On the use of Deep Boltzmann Machines for Road Signs Classification

BY

CARLO D’ERAMO
B.S., Politecnico di Milano, Milan, Italy, September 2011

THESIS

Submitted as partial fulfillment of the requirements for the degree of Master of Science in Computer Science in the Graduate College of the University of Illinois at Chicago, 2015

Chicago, Illinois

Defense Committee:

Brian Ziebart, Chair and Advisor
Tanya Berger-Wolf
Stefano Zanero, Politecnico di Milano
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 STATE OF THE ART</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Neural networks and objects classification</td>
<td>2</td>
</tr>
<tr>
<td>1.2 Boltzmann Machines</td>
<td>6</td>
</tr>
<tr>
<td>1.2.1 General Boltzmann Machines</td>
<td>6</td>
</tr>
<tr>
<td>1.2.2 Restricted Boltzmann Machines</td>
<td>14</td>
</tr>
<tr>
<td>1.2.3 Deep Belief Networks</td>
<td>16</td>
</tr>
<tr>
<td>1.2.4 Deep Boltzmann Machines</td>
<td>21</td>
</tr>
<tr>
<td>1.2.5 Multi-Prediction Deep Boltzmann Machines</td>
<td>25</td>
</tr>
<tr>
<td>2 OBJECTS RECOGNITION</td>
<td>29</td>
</tr>
<tr>
<td>2.1 Image and preprocessing</td>
<td>29</td>
</tr>
<tr>
<td>2.2 Neural networks and images</td>
<td>30</td>
</tr>
<tr>
<td>2.3 Object recognition with generative models</td>
<td>33</td>
</tr>
<tr>
<td>2.3.1 The German Traffic Sign Recognition Benchmark</td>
<td>35</td>
</tr>
<tr>
<td>3 PROJECT ARCHITECTURE AND IMPLEMENTATION</td>
<td>38</td>
</tr>
<tr>
<td>3.1 Libraries</td>
<td>38</td>
</tr>
<tr>
<td>3.2 Learning parameters</td>
<td>40</td>
</tr>
<tr>
<td>3.2.1 General parameters</td>
<td>40</td>
</tr>
<tr>
<td>3.2.2 Restricted Boltzmann Machine and Deep Boltzmann Machine parameters</td>
<td>42</td>
</tr>
<tr>
<td>3.2.3 Deep Boltzmann Machine finetuning parameters</td>
<td>43</td>
</tr>
<tr>
<td>3.2.4 Multi-Prediction Deep Boltzmann Machine parameters</td>
<td>43</td>
</tr>
<tr>
<td>3.3 Code implementation</td>
<td>44</td>
</tr>
<tr>
<td>3.3.1 German Traffic Sign Recognition Benchmark wrapper</td>
<td>45</td>
</tr>
<tr>
<td>3.3.2 Gaussian Restricted Boltzmann Machine training</td>
<td>46</td>
</tr>
<tr>
<td>3.3.3 Deep Boltzmann Machine training</td>
<td>47</td>
</tr>
<tr>
<td>3.3.4 Training of Multi-Prediction Deep Boltzmann Machine</td>
<td>50</td>
</tr>
<tr>
<td>3.4 German Traffic Sign Recognition Benchmark preprocessing</td>
<td>54</td>
</tr>
<tr>
<td>3.4.1 Enhancing contrast</td>
<td>55</td>
</tr>
<tr>
<td>3.4.2 Comments</td>
<td>58</td>
</tr>
<tr>
<td>4 EXPERIMENTAL RESULTS</td>
<td>61</td>
</tr>
<tr>
<td>4.1 MNIST</td>
<td>61</td>
</tr>
<tr>
<td>4.1.1 Greedy layer-wise pretraining</td>
<td>62</td>
</tr>
<tr>
<td>4.1.2 Deep Boltzmann Machine training</td>
<td>63</td>
</tr>
<tr>
<td>4.1.3 Deep Boltzmann Machine finetuning</td>
<td>64</td>
</tr>
<tr>
<td>CHAPTER</td>
<td>PAGE</td>
</tr>
<tr>
<td>---------</td>
<td>------</td>
</tr>
<tr>
<td>4.2</td>
<td>66</td>
</tr>
<tr>
<td>4.2.1</td>
<td>67</td>
</tr>
<tr>
<td>4.2.1.1</td>
<td>67</td>
</tr>
<tr>
<td>4.2.1.2</td>
<td>69</td>
</tr>
<tr>
<td>4.2.1.3</td>
<td>71</td>
</tr>
<tr>
<td>4.2.1.4</td>
<td>72</td>
</tr>
<tr>
<td>4.2.2</td>
<td>74</td>
</tr>
<tr>
<td>4.2.3</td>
<td>77</td>
</tr>
<tr>
<td>4.2.4</td>
<td>80</td>
</tr>
<tr>
<td>5</td>
<td>83</td>
</tr>
<tr>
<td>5.1</td>
<td>84</td>
</tr>
<tr>
<td>CITED LITERATURE</td>
<td>87</td>
</tr>
<tr>
<td>VITA</td>
<td>90</td>
</tr>
</tbody>
</table>
# LIST OF TABLES

<table>
<thead>
<tr>
<th>TABLE</th>
<th>MNIST - FIRST AND SECOND RBMS PARAMETERS FOR GREEDY LAYER-WISE PRETRAINING</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>MNIST - DBM PARAMETERS FOR TRAINING</td>
<td>62</td>
</tr>
<tr>
<td>II</td>
<td>MNIST - MLP PARAMETERS FOR FINETUNING</td>
<td>63</td>
</tr>
<tr>
<td>III</td>
<td>MNIST - MLP PARAMETERS FOR DROPOUT FINETUNING</td>
<td>65</td>
</tr>
<tr>
<td>IV</td>
<td>GTSRB - GRBM PARAMETERS</td>
<td>69</td>
</tr>
<tr>
<td>V</td>
<td>GTSRB - FIRST AND SECOND RBM PARAMETERS FOR GREEDY LAYER-WISE PRETRAINING</td>
<td>70</td>
</tr>
<tr>
<td>VI</td>
<td>GTSRB - DBM PARAMETERS FOR TRAINING</td>
<td>71</td>
</tr>
<tr>
<td>VII</td>
<td>GTSRB - MLP PARAMETERS FOR DROPOUT FINETUNING</td>
<td>72</td>
</tr>
<tr>
<td>VIII</td>
<td>BEST MODEL ERRORS FOR STANDARD GTSRB</td>
<td>73</td>
</tr>
<tr>
<td>IX</td>
<td>ERRORS FOR EACH PREPROCESSING METHOD</td>
<td>77</td>
</tr>
<tr>
<td>X</td>
<td>MLP FINAL PARAMETERS</td>
<td>78</td>
</tr>
<tr>
<td>XI</td>
<td>FINAL RESULTS</td>
<td>79</td>
</tr>
<tr>
<td>XII</td>
<td>MP-DBM PARAMETERS FOR TRAINING</td>
<td>80</td>
</tr>
<tr>
<td>FIGURE</td>
<td>PAGE</td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>27</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>31</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>35</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>36</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>36</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>48</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>53</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>55</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>56</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>57</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>58</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>59</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>60</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>64</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>65</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>70</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>73</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>75</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>75</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>75</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>76</td>
<td></td>
</tr>
<tr>
<td>FIGURE</td>
<td>PAGE</td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>HSV space reconstructions</td>
<td>76</td>
</tr>
<tr>
<td>32</td>
<td>Chromaticity space reconstructions</td>
<td>76</td>
</tr>
<tr>
<td>33</td>
<td>Dropout vs Standard backpropagation</td>
<td>79</td>
</tr>
<tr>
<td>34</td>
<td>DBM and MP-DBM test errors diagram</td>
<td>82</td>
</tr>
<tr>
<td>35</td>
<td>Examples of GTSDB images</td>
<td>85</td>
</tr>
</tbody>
</table>
LIST OF ABBREVIATIONS

BM Boltzmann Machine. vii, 6, 8, 14, 21, 24

CD Contrastive Divergence. vii, 14, 15, 68

CUDA Compute Unified Device Architecture. vii, 38


DBN Deep Belief Net. vii, ix, 16, 18, 20, 21, 34

GRBM Gaussian Restricted Boltzmann Machine. vii, 44, 46, 47, 55, 67–69, 74, 80, 86

GTSDB German Traffic Sign Detection Benchmark. vii, 84, 86

GTSRB German Traffic Sign Recognition Benchmark. vii, x, xi, 35, 37, 41, 42, 44, 45, 50, 54, 55, 64, 66, 67, 72, 77, 78, 81, 83, 84

HOG Histogram of Oriented Gradients. vii, 78

HSV Hue-Saturation Value. vii, 58, 74

IJCNN International Joint Conference on Neural Networks. vii, 35, 84

KL Kullback-Leibler. vii, 11, 14, 15

LDA Latent Dirichlet Allocation. vii, 78

LTS Long-Term Support. vii, 61

MLP Multi-Layer Perceptron. vii, 5, 24, 25, 28, 41–43, 48, 50, 64, 71, 77, 78, 81

MP Multi-Prediction. vii, 25–27
LIST OF ABBREVIATIONS (Continued)

MP-DBM Multi-Prediction Deep Boltzmann Machine. vii, 25, 26, 28, 34, 38, 40, 43, 44, 50, 80, 81, 83, 84, 86

MRF Markov Random Field. vii, 1

NCA Neighbourhood Component Analysis. vii, 34

RBM Restricted Boltzmann Machine. vii, 14–18, 20–23, 40–44, 47, 48, 50, 62, 63, 68–70

SAP Stochastic Approximation Procedure. vii, 12

SGD Stochastic Gradient Descent. vii, 26

SVM Support Vector Machine. vii, 34
SUMMARY

One of the most important issues in computer science is to give a computer some degree of intelligence in order to make it able to replace (or overcome) humans in some tasks. Indeed, there is a large variety of problems that can be easily carried out by humans, but it is still tough to teach a computer how to deal with them. Objects recognition is one of these problems: humans are very good at solving it, but we are still not able to build computers that can fully replace humans in this task.

Despite the fact that it is not possible to reproduce human performance at object classification, many machine learning models and algorithms have been successful at making computers able to classify objects belonging to small sets of them such as digits, letters, animals and so on. These techniques use a set of images called training set that is used to “teach” the features of an object (e.g.: colors, shape) to the model, in order to make it able to recognize instances of that object never seen before. In machine learning, there is a large variety of models and algorithms that have been developed during the years and it has been proved that there is not a single model that is always better than one another; thus, it is interesting to test and analyze the performance of models on different tasks in order to compare them and understand them better.

In this work we want to test the performance of deep generative models that are becoming increasingly interesting for objects recognition. We started studying the deep generative model of Deep Belief Net (DBN) and other related models. After some research, we focused on the
model of Deep Boltzmann Machines (DBMs) that we found to be one of the most effective generative models for classification. We focused on the results of Hinton and Salakhutdinov that, in their work on DBMs\cite{1}, propose a training algorithm for DBMs able to make them very efficient in classification. Indeed, in their experiments, they achieved high performance in digits and simple objects classification, overcoming a large number of machine learning models. These tasks were relatively simple and there are no many other experimental results showing the effectiveness of DBMs in more complex objects classification. We tested the possibilities of DBMs on the German Traffic Sign Recognition Benchmark (GTSRB) dataset\cite{2}, a collection of images of 43 classes of road signs, that we considered sufficiently complex to be an interesting test of DBMs effectiveness in complex objects classification.

We implemented the DBM training procedure proposed by Hinton and Salakhutdinov using a Python machine learning framework called Pylearn2\cite{3} in order to exploit its helpful features to build the model and to analyze it once trained. To become familiar with Pylearn2 and to have enough confidence on the correctness of our implementation, we first tried to reproduce their result on the MNIST dataset. Then, once we have been able to reach it, we started working on GTSRB. In this thesis, we show how we dealt with this dataset and the choices we made to reach our published results on it. Finally, we show how we have succeeded to improve on the classification performance of the best published generative model we have been able to find, confirming that DBM is one of the best generative models in classification tasks and that it can achieve high results also in complex objects classification.
In the next chapter we give an overview of discriminative and generative models, focusing on the deep generative models we used in this work. Moreover, we explain the training procedure we used, showing how it is able to efficiently train DBMs.

In Chapter 3, we discuss the issues related to the training of models for images classification and we present the GTSRB dataset and its features.

In Chapter 4, we explain Pylearn2 and the implementation of the DBM training procedure we made using it. We also show some optimization techniques we used to make images more suitable for training.

In Chapter 5, we show the experimental results we have been able to obtain.

Finally, in the last chapter, we resume our work and discuss some possible future developments of it.
CHAPTER 1

STATE OF THE ART

Machine learning has two opposed classes of models. One is composed of *discriminative* models that are models able to learn the dependence of an unobserved variable on an observed variable in a certain context. In other words, given the value of the observed variable, they are able to infer the value of the unobserved one.

The other class includes *generative* models that, instead, are full probabilistic models of all the variables involved.

Both discriminative and generative models are very often implemented through graphical models, that are probabilistic models able to model complex probability distributions represented with a graph composed of random variables (nodes) and conditional dependencies among them (edges). Random variables depend on or influence other ones in a certain context; for instance, dealing with object recognition, random variables can be the color of the object, its shape, its name and so on.

The most commonly used graphical models are Bayesian Networks and Markov Random Fields (MRFs); the former can be represented with a directed and acyclic graph, the latter have an undirected and cyclic graph. Thus, Bayesian Networks are not able to model cyclic dependencies like MRFs, but they can model induced dependencies whereas a MRF cannot. For both of models observed it is possible to infer the state of random variables given the real state of some of them. Unfortunately, for MRF with high tree-width exact inference is not...
possible because a particular element of the inference formula, called *partition function* (used to transform a normal value into a probabilistic value), considers all the possible configurations of random and observed variables making the complexity of the algorithm exponential. However inference can be approximated with methods such as *Monte Carlo Sampling*, without a big loss in terms of performance making them computable probabilistic models.

As stated in the introduction, our work consists of an object classification task using generative models on a complex dataset to evaluate the performance of them in discrimination and comparing them with the ones of discriminative models. To explain the concept of discrimination, we briefly introduce the discriminative model of neural networks, moving afterwards to the generative models we used.

**1.1 Neural networks and objects classification**

Neural networks\[^4\] are a discriminative model that has been proved to be very effective for classification and regression tasks. Our work deals only with a classification task and therefore this is the only class of problems that will be considered from now on in reviewing the state of the art. In this kind of problems, the aim is to obtain a model able to recognize objects never seen before assigning the right *class* to them. A class represents the identity of the object: for instance, if the task is to classify 4 types of animals, classes can be dog, cat, bird or fish. The neural network has to be trained in order to be able to discriminate among each class and assign the correct one to each example (the details of training will be explained soon).

Neural networks are directed graphs in which each node (neuron) has an *activation function* and a *bias*, and each edge (synapsis) has a *weight*. Neurons in a so called *feedforward* neural
network (Figure 1) can be grouped in layers that have incoming connections only with the previous layer and outcoming connections only with the following one. Moreover, no connections among neurons belonging to the same layer are allowed. In other words, the graph modeling a neural network is directed and the flow of information in a standard neural network goes from the beginning to the end of the network without going back.

Figure 1 shows a feedforward neural network composed of three layers: an input layer on the left, a hidden layer in the middle and an output layer on the right. The circles are the neurons of the network and the arrows represent the synapsis. As it can be seen, there are only connections between adjacent layers.

Layers are a fundamental aspect of neural networks due to their ability to capture features of data in order to obtain a more complex knowledge of them. Indeed, starting from the first layer, called visible layer, it is possible to activate neurons in the following layers (hidden layers) to make an abstraction of the input capturing more and more complex features through deeper layers\[5\]. The last layer, also called output layer, models the answer of the network to a specific input data.

Each neuron of a neural network has an activation function that has to be computed to obtain the output of a neuron and its formula is:

$$y_j = f\left(\sum_i W_{ij} x_i + b_j\right)$$ (1.1)

where \(j\) is the index of the neuron in the considered layer, \(i\) is the index of a neuron in the previous layer, \(x_i\) is the activation function value of neuron \(i\), \(W_{ij}\) and \(b_j\) are, respectively, the
weights of each edge connecting a neuron $i$ to the neuron $j$ and the bias of neuron $j$ and, finally, $f()$ is a function that can be linear or not. In classification tasks the output layer answer is the object class the network thinks current input data belongs to and can be computed using the activation function of the neurons in the output layer, modeling the answer, for instance, with a one-hot configuration where the number of neurons in the output layer is equal to the number of classes and only the one of them with the highest value of activation function is on.

To make the network able to give the correct answer given a certain input, i.e., to turn on the right neuron in the top layer, the weights and biases of the neural network need to be modified in order to make the network able to efficiently recognize a large amount of the object on which it has been trained. Efficient weights and biases can be found by means of gradient descent techniques, such as backpropagation, starting from a random initialization of them and changing them iteratively trying to reduce the number of errors obtained with the current configuration. Gradient descent moves the current configuration of parameters in the $n$-dimensional space ($n$ is the number of weights and biases) descending along the direction of the gradient of the error and, after a reasonable number of epochs, the algorithm is supposed to find good weights and biases.

Commonly, activation functions are sigmoidal or hyperbolic tangent functions and networks with activation functions of this form are called deterministic neural networks. A modified version of them uses these activation functions as probabilities of binary neurons, whose output is 1 or 0, to determine if the neuron is on or not. This type of networks are called stochastic neural networks and due to their probabilistic behaviors they are more likely, than other models,
to avoid local minima of the error function that the network is trying to minimize. Indeed, gradient descent procedure in deterministic neural networks does not make large jump across the error function and it is difficult for them to move from a local minima. A stochastic neural network makes the current error fluctuate randomly allowing it to escape from a local minima more easily. There are many other methods to increase performance of a network, but stochastic neurons have been presented because they are required in the models that we used in this work.

To reduce the error rate of the network, it is also critical to choose the right number of layers and neurons. The simplest model of neural network, called perceptron, is composed of only one neuron and it can only perform linear classification due to its very low complexity. It has been proved that only networks with a larger number of neurons are capable of non-linear classification or regression. In complex domains, there is the need to build large neural networks able to deal with their complexity and dimensions; one of these networks is the well-known Multi-Layer Perceptron (MLP) that is a bigger version of the perceptron. The machine learning branch of deep learning works with models with a large number of layers; for instance a deep neural network\(^\text{[6]}\) is a neural network with a relatively large number of layers often
containing many neurons. Deep learning models set the state of the art on many datasets and have become one of the most exploited ways to reach high results in a lot of object classification tasks.

1.2 Boltzmann Machines

In this work we focus on Boltzmann Machines (BMs)\[^7\] (Figure 2) and their variants exploring, in particular, Deep Boltzmann Machines (DBMs)\[^1\] which we will explain later in this chapter. BMs are one of the best generative models in object classification and our aim is to understand its potentialities in complex object classification and comparing it to the ones of pure discriminative models.

1.2.1 General Boltzmann Machines

A Boltzmann Machine is an energy-based undirected graphical model belonging to the group of generative stochastic neural networks. A BM contains a set of visible units \( v \in \{0, 1\}^V \) that models the input of the network (e.g., pixels of an image) and a set of hidden units \( h \in \{0, 1\}^U \) that act as feature detectors.

Figure 2 shows a BM composed of a visible layer on the bottom and a hidden layer on the top. The edges connect each neuron to all the others making a complete graph.

Each state \( \{v, h\} \) has an energy function defined as:

\[
E(v, h; \theta) = -\frac{1}{2} v^T L v - \frac{1}{2} h^T J h - v^T W h
\]  (1.2)
where $\theta = \{L, J, W\}$ are the model parameters: $L$, $J$ and $W$ represent visible-to-visible, hidden-to-hidden and visible-to-hidden symmetric connections. This energy function is used to define the probability that the model assigns to a joint configuration $\{v, h\}$:

$$P_{model}(v, h; \theta) = \frac{1}{\mathcal{Z}(\theta)} e^{-E(v, h; \theta)} \quad (1.3)$$

where the partition function $\mathcal{Z}(\theta)$:

$$\mathcal{Z}(\theta) = \sum_{v, h} e^{-E(v, h; \theta)} \quad (1.4)$$
takes into account all the possible joint configurations of visible and hidden units, thus transforming a scalar value into a probability value. The probability assigned to a vector of visible units $v$ is summed over all possible hidden vector $h$ configurations:

$$P_{model}(v; \theta) = \frac{P^*(v; \theta)}{Z(\theta)} = \frac{1}{Z(\theta)} \sum_h e^{-E(v, h; \theta)}$$

(1.5)

where $P^*$ denotes "unnormalized" probability.

We want to assign a greater probability to those states that give the correct prediction given a dataset example. In BMs this can be done with the adjustment of weights and biases lowering the energy of the states that would generate one of the samples coming from the training data. At the same time, we also want to lower the probability of obtaining data that is not sampled from the training data distribution. To this end, we allow the model to stochastically generate confabulations (or fantasies) and then raise accordingly the energy of the associated states. The change in a weight is then given by:

$$\frac{\partial \log(p(v))}{\partial w_{ij}} = \mathbb{E}_{P_{data}[v_i h_j]} - \mathbb{E}_{P_{model}[v_i h_j]}$$

(1.6)

where $\mathbb{E}_P$ is the expectation under the distribution specified by the subscript. In other words, both expectations indicates the frequency with which each pair of visible and hidden units (feature detectors) are simultaneously on when the network is driven under certain weights defining a specific probability distribution. The first element corresponds to the state obtained
when a training example is clamped to the visible layer. The second element is trickier and will be explained later.

It is useful to define the conditional probability functions of both type of units because they will be used for training:

\[
p(h_j = 1|v, h_{-j}) = g\left(\sum_i W_{ij} v_i + \sum_{m \neq j} J_{jm} h_m + b_j\right) \quad (1.7)
\]

\[
p(v_i = 1|h, v_{-i}) = g\left(\sum_j W_{ij} h_j + \sum_{k \neq i} L_{ik} v_k + a_i\right) \quad (1.8)
\]

where \(g(x)\) is the logistic function \(\frac{1}{1+e^{-x}}\), \(h_{-j}\) (and \(v_{-i}\)) are hidden and visible vectors without unit of index \(j\) (or \(i\)) and \(b_j\) and \(a_i\) are biases.

While the first element of the update formula (Equation 1.6) is an unbiased sample easily obtainable clamping a training example to the visible layer, the second element cannot be computed immediately. The procedure to calculate it involves starting from a random state of the visible units and performing Gibbs sampling until the network reaches thermal equilibrium, a situation where the probability distribution of the joint probability of the states has converged. A single step of Gibbs sampling consists of updating all hidden units in parallel using Equation 1.7 and updating all the visible units with Equation 1.8 afterwards. Reaching thermal equilibrium could be a very time consuming task and this could be a bottleneck in real cases.
In 1983 Hinton and Sejnowski\cite{7} derived the parameter update to perform gradient ascent in the log-likelihood:

\[
\Delta W = \alpha (\mathbb{E}_{\text{P}_{\text{data}}} [vh^T] - \mathbb{E}_{\text{P}_{\text{model}}} [vh^T])
\]

\[
\Delta L = \alpha (\mathbb{E}_{\text{P}_{\text{data}}} [vv^T] - \mathbb{E}_{\text{P}_{\text{model}}} [vv^T])
\]  

\[
\Delta J = \alpha (\mathbb{E}_{\text{P}_{\text{data}}} [hh^T] - \mathbb{E}_{\text{P}_{\text{model}}} [hh^T])
\]  

where \( \alpha \) is a learning rate and \( \mathbb{E} \) denotes an expectation. In particular, the data-dependent term \( \mathbb{E}_{\text{P}_{\text{data}}} \) is an expectation w.r.t. \( P_{\text{data}}(h, v; \theta) \), that is the completed data distribution.

Applying the chain rule we obtain \( P_{\text{data}}(h, v; \theta) = P(h|v; \theta)P_{\text{data}}(v) = P(h|v; \theta) \frac{1}{N} \sum_n \delta(v - v^n) \). \( P_{\text{data}}(v) \) is indeed the expectation that a certain visible vector is observed in the training set and it corresponds to the empirical distribution of data \( \frac{1}{N} \sum_n \delta(v - v^n) \) where \( \delta \) is the Dirac delta function\textsuperscript{1}. This is a formal way to count the number of examples of the training set \( v^n \) that are equal to the current one \( v \) because if \( v = v^n \) then \( \delta(v) = 1 \); otherwise it is 0. The expectation is then obtained dividing the sum by \( N \). The second term is the expectation with respect to the distribution defined by the model without any clamped visible vector, thus it is also called data-independent term.

Exact maximum likelihood learning cannot be used to compute either of these terms: the data-dependent term would require an exponential time in the number of hidden units and the time required to compute the model expectation term would be exponential in the number of both visible and hidden units. Hinton and Sejnowski\cite{7} proposed an algorithm to approximate

\textsuperscript{1}The Dirac delta function, also known as impulse function, is a function that is zero everywhere except at zero and has an integral of one over the entire real line.
these terms that consisted in running, during each iteration of learning, a separate Markov
chain for each training example from the dataset and a Markov chain to approximate the
model distribution. The problem with this procedure is that each chain requires a long time to
reach the stationary distribution. In 2012 Hinton and Salakhutdinov\cite{1} proposed a different ap-
proach to make the computation of both terms tractable by computing them with two different
independent methods:

- **Data-dependent expectations** are approximated with a *variational approach* that re-
  places the true posterior distribution over latent variables $P(h|v; \theta)$ given the current
  training vector $v$ by an approximate posterior $Q(h|v; \mu)$ and updates the parameters
  following the gradient of a lower bound on the log-likelihood:

  \[
  \log P(v; \theta) \geq \sum_h Q(h|v; \mu) \log P(v, h; \theta) + \mathcal{H}(Q)
  \]

  \[
  \geq \log P(v; \theta) - KL[Q(h|v; \mu)||P(h|v; \theta)]
  \]  

  (1.10)

  where $\mathcal{H}(Q)$ is the entropy functional. As it can be seen, variational learning tries to
  maximize the log-likelihood and at the same time it also tries to minimize the Kullback-Leibler (KL)
  divergence between the approximating and the true posterior distributions.

  Basing on the assumption that the distribution function of the data is usually unimodal,
  the true posterior is usually approximated with a *naive mean-field approach* that uses a
  *fully factorized distribution* (i.e., a distribution whose terms are all independent from one
another) \( Q(h|v; \mu) = \prod_j q(h_j) \) with \( q(h_j = 1) = \mu_j \). Using this distribution, the lower bound becomes:

\[
\log p(v; \theta) \geq \frac{1}{2} \sum_{i,k} L_{ik} v_i v_k + \frac{1}{2} \sum_{j,m} J_{jm} \mu_j \mu_m + \sum_{i,j} W_{ij} v_i \mu_j - \log Z(\theta) - \sum_j [\mu_j \log \mu_j + (1 - \mu_j) \log (1 - \mu_j)].
\]  

(1.11)

Then the learning procedure maximizes this lower bound with respect to the variational parameters \( \mu \) for fixed \( \theta \), by performing the following mean-field updates until convergence\(^1\):

\[
\mu_j \leftarrow g(\sum_i W_{ij} v_i + \sum_{m \neq j} J_{mj} \mu_m).
\]  

(1.12)

- **Model expectations** are approximated by means of a Stochastic Approximation Procedure (SAP) consisting of a sequence of parameters and states updates. The procedure begins with a set of initial parameters \( \theta^0 \) and an initial state \( x^0 \), randomly initialized; then the update phase samples a new state \( \tilde{x}^{t+1} \) from \( \tilde{x}^t \) using a transition operator that leaves \( p(\bullet; \theta^t) \) invariant (such as Gibbs sampling). This process is called Markov chain. To make the estimation of the expectation more efficient, \( M \) Markov chains are run simultaneously instead of only one and for each of them \( G \) Gibbs steps are performed. Once all Markov chains have been completed, the expectations \( \tilde{x}^{t+G,m} \), where \( m \) is the Markov

\(^1\)To speed up computations only a certain number of updates are done without affecting performance too much.
Figure 3: Restricted Boltzmann Machine

chain to which the vector belongs to, are averaged to find the final model expectation

\[ \tilde{x}^{t+G} = \frac{1}{M} \sum_{m} \tilde{x}^{t+G,m}. \]

Once both the expectations have been computed, weights are changed with the following weight update formulas:

\[
egin{align*}
W^{t+1} &= W^t + \alpha \left( \frac{1}{N} \sum_n v^n (\mu^n)^T - \frac{1}{M} \sum_m \tilde{v}^{t+G,m} (\tilde{h}^{t+G,m})^T \right) \\
J^{t+1} &= J^t + \alpha \left( \frac{1}{N} \sum_n \mu^n (\mu^n)^T - \frac{1}{M} \sum_m \tilde{h}^{t+G,m} (\tilde{h}^{t+G,m})^T \right) \quad (1.13) \\
L^{t+1} &= L^t + \alpha \left( \frac{1}{N} \sum_n v^n (v^n)^T - \frac{1}{M} \sum_m \tilde{v}^{t+G,m} (\tilde{v}^{t+G,m})^T \right).
\end{align*}
\]

This is the general procedure to train a general Boltzmann Machine, but it is very slow. Some constraints to the topology of the network can be applied in order to make the training faster, but at the price of having a less powerful model.
1.2.2 Restricted Boltzmann Machines

Restricted Boltzmann Machines\cite{8} are a type of BMs with two layers in which one layer is composed of visible units and the other one of hidden units. Moreover there are no visible-to-visible or hidden-to-hidden connections.

Figure 3 shows a Restricted Boltzmann Machine (RBM) composed of two layers with the visible one on the bottom and the hidden one on the top. As it can be seen, the RBM does not have connections among neurons belonging to the same layer making it a complete bipartite graph.

The energy function of a joint configuration \( \{ v, h \} \) in a RBM is the same as the one of the general BM case, but with \( J \) and \( W \) both set to 0:

\[
E(v, h; \theta) = -v^T Wh. \tag{1.14}
\]

Hinton\cite{9} proposed a solution that made the previously explained training technique computationally feasible using RBMs. The procedure tries to minimize the Contrastive Divergence (CD) which is the KL-divergence between the two distributions \( P^0 \parallel P^1_\theta \) where the subscript \( \theta \) indicates that the distribution is obtained using the current parameter vector \( \theta \) and the superscript indicates the number of performed Gibbs sampling steps: 0 stands for the initial state with the training vector clamped to the visible units and 1 stands for a state after one Gibbs sampling step. The algorithm starts by setting the state of the visible units clamping a training example and then performs one step of Gibbs sampling using the conditional probability
formulas (Equation 1.7 and Equation 1.8). This allows to obtain a reconstruction of the input
data which will be used for the weights update formula. One can also use CD-$k$ which simply
performs $k$ steps instead of one, but even though this choice gives a reconstruction which is
closer to the thermal equilibrium, it has the drawback to add variance (or noise) to it. In this
case the KL-Divergence that is minimized is $P_0 \parallel P_\theta^k$ where $k$ stands for a state after $k$ Gibbs
sampling steps. The update formula for CD (or CD-$k$) is:

$$\Delta w_{ij} = \epsilon (E_{P_{\text{data}}} v_i h_j - E_{P_{\text{recon}}} v_i h_j)$$ \hspace{1cm} (1.15)$$

with $E_{P_{\text{recon}}}$ replacing $E_{P_{\text{model}}}$ because the latter represents the distribution after convergence
$P^\infty$.

RBMs are an efficient way to find good feature detectors, but performance from this per-
spective can be improved by stacking several RBMs on top of each other with a procedure called
greedy layer-wise pretraining. As we have shown before, a RBM can be trained minimizing CD-$k$
and this procedure can be applied also for larger models that we introduce later. Starting
from the bottom RBM, it is possible to train each layer of RBMs till the top one. The bottom
RBM is the one that takes the input data from the training set by clamping a data vector into
its visible layer. After having trained the bottom one with CD-$k$, the following hidden layers
can be learned also with CD-$k$ using the hidden activities of the previous RBM as the visible
data for the higher level one. In other words, once we have obtained the trained weights of
a RBM and we have activated its hidden neurons sampling it from the posterior distribution
given the visible vector, we can use the sampled hidden state as the visible state of the RBM above that will be trained the same way. With slight changes, this is what is done to build a DBM that is one of the models we used for this work. However in order to understand DBMs better, it is advisable to analyze first a hybrid model called Deep Belief Net (DBN).

1.2.3 Deep Belief Networks

To understand the procedure that leads to a DBN it is better to discuss some related issues first. As shown in the introduction of this chapter, inference in a belief network (also known as Bayesian Network) is very difficult. As a matter of fact, the well-known phenomenon of explaining away often makes inference intractable and so the posterior distribution over hidden variables cannot be computed. To have tractable inference, it is desirable to have a factorial posterior distribution. Hinton\cite{10} shows a trick to eliminate the explaining away making the posterior distribution factorial; the trick involves the use of extra hidden layers, called complementary priors, with exactly the opposite correlations to those in the likelihood term coming from the data. This way, when a data vector is observed, the opposite correlations multiplied with the prior distribution eliminate themselves making the posterior factorial.

A model that implements such a trick is the one in Figure 4 where the rectangles represent the whole set of neurons in each layer and the arrows represent all the connections among layers. This is a semi-infinite model with tied weights in which the priors are complementary at every hidden layer. This network is clearly useless because it has no limit on the top and it can be considered just as a theoretical model: it should be limited in some way in order to make it practical. It can be shown that an infinite directed net is equivalent to a RBM\cite{10}. Indeed,
in the semi-infinite belief net, data can be generated starting with random data clamped in a infinitely deep layer and then performing top-down passes. In a RBM data can be generated starting with a random configuration of states in one of the two layers and then performing Gibbs sampling until reaching equilibrium distribution. It turned out that these procedures are equivalent and therefore an infinite belief net can be replaced with a RBM. In other words, the undirected connections of the RBM bring to the same model of an infinite belief net with tied weight matrices. Thus, the final network to be trained has the form of the one in Figure 5.

The learning algorithm of the hybrid model in Figure 5 initially trains the first layer of weights $W_0$ assuming that the weight matrices of each layer in the whole network have the
same values. As explained by Hinton, this is done in order to guarantee the existence of complementary priors. Once $W_0$ has been learned, it is freezeed and $W_0^T$ can be used to sample from the visible layer creating data for the first layer of hidden units. After the initialization of this layer, the second RBM can be trained the same way of the first and this procedure can be applied recursively to all other layers. Eventually the network will have all layers trained and tied directed weights at each layer.

The presented procedure is an unsupervised technique useful to exploit the performance of multiple RBMs. Even though it is reasonably powerful, its performance can be improved with a finetuning procedure that, starting from the pretrained parameters, trains the whole network finding even better ones. In the finetuning phase, the DBN is trained with a variation of the Wake-Sleep algorithm[11], an unsupervised training algorithm invented by Hinton in 1995 which can be used to train a so called Helmholtz machine (Figure 6). As shown in the figure, a Helmholtz machine has the same structure of a DBN with the exception of the RBM on the top that is not present; the dashed line indicate top-down connections and the continuous lines indicate bottom-up connections. The wake phase adjusts the generative weights (top-down) when the network is driven by the discriminative weights (bottom-up) and the sleep phase does the opposite.

Hinton shows that if the recognition weights are fixed, the update formula to adjust the generative weights is:

$$\Delta w_{kj} = \alpha s_k (s_j - p_j)$$  \hspace{1cm} (1.16)
Figure 5: Hybrid network with RBM on top

Figure 6: A three-layer Helmholtz machine
where $\alpha$ is the learning rate, $s_k$ and $s_j$ are the state of the layers $k$ and $j$ when the network is driven with recognition weights and $p_j$ is the probability that unit $j$ would turn on if it was driven by the current state in the layer above using the current generative weights.

Conversely, the update formula to adjust the recognition weights is:

$$\Delta w_{jk} = \alpha s_j (s_k - q_k)$$

where $\alpha$ is the learning rate, $s_j$ and $s_k$ are the state of the layers $j$ and $k$ when the network is driven with generative weights and $q_k$ is the probability that unit $k$ would turn on if it was driven by the current state in the layer below using the current discriminative weights.

Normally, Wake-Sleep starts initializing the top layer stochastically using only its bias to influence the probability of neurons of being on, but this cannot be done in a DBN because of the RBM on the top of it that would not produce samples from the generative model. However, Hinton (2006) shows that a contrastive version of wake-sleep can be applied to a DBN in order to make generative finetuning possible. This method involves the initial clamping of a training example-label pair from the training set and performing bottom-up passes till the RBM state is set. Then alternate Gibbs sampling allows the top RBM to produce a confabulation of the input. The confabulation is used to perform the Wake-Sleep algorithm in the previously explained way.

As previously specified, the RBM on the top of a DBN and the directed connections below it, make this model a hybrid between an undirected and a directed one. We now explain a
slightly different model that differs from a DBN because it has undirected connections between each layer and therefore it can be seen as a stack of RBMs (Figure 7: the associative memory can be seen at the top of the DBN). This model is called Deep Boltzmann Machine and, as stated before, it is the one we used the most for the purpose of our work.

1.2.4 Deep Boltzmann Machines

A DBM\textsuperscript{1} is a Boltzmann Machine composed of stacked RBMs and therefore it has no connections between non-adjacent layers nor intra-layer ones. A DBM, being a BM, can be trained with the training procedure for general BMs shown in section 1.2.1. Nevertheless, it is useful to initialize weights with a pretraining procedure which is a slightly modified version of the one for DBN which we have just presented with the ability to avoid the creation of asymmetric weights making it a stack of RBMs. This procedure considers the fact that after

![Figure 7: A Deep Belief Network and a Deep Boltzmann Machine](image)
learning the second-layer RBM, the approximation to the true posterior $P(h^{(1)}|v; W^{(1)}, W^{(2)})$ (where the superscript is the index of the hidden layer) can be computed in two different ways: the first way infers $h^{(1)}$ ignoring the second layer and uses $P(h^{(1)}|v; W^{(1)})$ and, on the other hand, the second alternative uses only the second layer ignoring the bottom one. This can be done performing two bottom-up passes and then using $P(h^{(1)}|h^{(2)}; W^{(2)})$.

Both methods overemphasize the included layer and underemphasize the other one making the inference of $h^{(1)}$ not sufficiently accurate. A trick to have a better inference is to take a geometric average of these two distributions using $\frac{1}{2}W^{(1)}$ and $\frac{1}{2}W^{(2)}$ (where the superscript is the index of the layer the weights vector belongs to) to infer $h^{(1)}$ after the initial two bottom up passes to infer $h^{(2)}$. This can be applied to a network with an arbitrary number of layers. In a network with more than two hidden layers, this can be done by alternating resampling the odd-numbered layers and the even-numbered layers and this procedure corresponds to alternate Gibbs Sampling in a DBM with the visible units clamped. It seems that a pretrained stack of RBMs can be composed to form a DBM halving all the weights, but unfortunately, this procedure has a problem with the first and last layer because for both of them there is only one input (the first hidden layer or the last but one layer) and therefore it is not legitimate to halve their weights to obtain the geometric average of incoming inputs. The simple trick to solve this problem is to constrain the bottom-up or top-down (respectively for visible or top layer’s) weights to be the double of the opposite ones during pretraining of the first and last RBM to compensate the lack of the second element in the geometric mean (Figure 8). Thus, a pretrained stack of RBMs, following this procedure, can be composed to form a DBM halving
all the weights except for the top-down ones in the first RBM and the bottom-up ones in the last RBM because this way it is possible to maintain their symmetry in each layer. Moreover, there is a method to correctly initialize the hidden states at the first step of inference when given a data vector, that consists in doubling the weights connecting the visible layer and the first hidden layer to compensate the lack of input making the inference correct.

Figure 8 shows the RBM pretraining on the left. It can be seen that the weights are doubled according to the previously explained assumption. After the pretraining, the DBM (on the right) can be build using the pretrained weights.
It is worth noting that Hinton and Salakhutdinov show that performing greedy layer-wise pretraining in a 2-layers DBM improves the variational bound, but they were not able to demonstrate the same for a DBM with \( n \) layers. Thus, there is no proof that using this technique to pretrain a DBM would work if the number of layers is greater than two even though it seems to work well in practice.

At the end of the pretraining phase and after the training procedure for general BMs has been applied, finetuning may be used to improve generative or discriminative performance. Hinton and Salakhutdinov\(^1\) explain how to improve discriminative performance in a 2-layers DBM by means of backpropagation; this is also one of the techniques we used in our work. Supervised finetuning with backpropagation is applied on a MLP that has the same configuration of the DBM except for the visible layer. Indeed, considering the fact that the first hidden layer takes inputs from the visible layer and the second hidden layer, we need to augment the input of the multilayer neural network with the sampled state of the second hidden layer of the DBM after the bottom-up passes as shown in Figure 9. This is done for every data in the training set creating a dataset with the same number of data as before augmented with the sampled state of the second hidden layer.

Eventually, a *softmax* layer is added at the top of the multilayer neural network to make it able to perform classification. A softmax layer is generally constrained to return the output of the network in a one-hot configuration turning on only the neuron with the highest value of activation function; in other words, it turns on only the most probable neuron that is the
most probable class that the network assigns to the input. Finally, the MLP is finetuned with backpropagation.

Figure 9 shows on the left the trained DBM. From left to the right the mean-field iterations are shown, with the trick of doubling the weights at the first iteration, as explained before. After the required number of mean-field iterations have been performed, the weights $W_2$ connecting the second layer to the first layer are used to connect the augmented input (the inferenced second layer) to the first layer and a softmax layer $y$ is added on the top.

1.2.5 Multi-Prediction Deep Boltzmann Machines

A new model based on DBMs, called Multi-Prediction Deep Boltzmann Machine (MP-DBM)\cite{12}, uses a different approach for training called Multi-Prediction (MP) training. The involved procedure trains the model "to predict any subset of variables given the complement of that subset of variables"\cite{12}. In other words, given some values contained in an example of the dataset, the model is trained to be good at predicting the missing values of that example.
Thus, dealing with images, the network is trained with examples from which some pixels are removed and the network learn how to predict them using the remaining ones.

Let $\mathcal{O}$ be a vector containing all variables that are observed during training. In unsupervised tasks, $\mathcal{O}$ corresponds to $v$ whereas in supervised ones $\mathcal{O} = [v, y]^T$. "Let $\mathcal{D}$ be a training set (i.e., a collection of values of $\mathcal{O}$). Let $S$ be a sequence of subsets of the possible indices of $\mathcal{O}$. Let $Q_i$ be the variational (e.g., mean-field) approximation to the joint of $\mathcal{O}_S$ and $h$ given $\mathcal{O}_{-S_i}$":\[12:\]

$$Q_i(\mathcal{O}_{S_i}, h) = \arg\min Q_{KL}(Q(\mathcal{O}_{S_i}, h)||P(\mathcal{O}_{S_i}, h|\mathcal{O}_{-S_i})).$$

(1.18)

There is not an explicit formula for $Q$ and it has to be computed with an iterative optimization process which runs mean-field updates to convergence. For simplicity, $Q$ is constrained to be factorial.

The MP-DBM is trained by using Stochastic Gradient Descent (SGD) on the MP objective function:

$$J(\mathcal{D}, \theta) = -\sum_{\mathcal{O} \in \mathcal{D}} \sum_i \log Q_i(\mathcal{O}_{S_i}).$$

(1.19)

As it can be seen, for each example of the dataset, a sum of several terms is considered. Each term indicates a subset of values to be removed from the example and the network will learn, from the remaining ones, how to predict them. During SGD training, minibatches of values of $\mathcal{O}$ and $S_i$ are sampled. $\mathcal{O}$ is sampled drawing an example from the training set whereas sampling $S_i$ means selecting each value of the example with a certain probability (generally
Figure 10 shows three different cases of MP steps. Each row has a neural network initialized differently from the other ones and each column represent a mean-field step: the leftmost column is the initial state, the central one shows the state during an intermediate step and the rightmost one is the state after one step. The black circles are the state that the network uses to learn it is 0.5). The selected variable will be the ones to be excluded from the training images and therefore they will be used as values to be predicted, the remainder of them will be used to train the network.
how to predict the blue ones. The white circles are the hidden states. The green arrows are the computational dependencies used to perform the mean-field steps.

This model has been developed in order to solve some issues related to DBMs. Firstly, it allows to avoid $L+2$ different phases of training (where $L$ is the number of layers of the DBM) reducing it to one phase of training. Moreover, while DBMs performance in classification tasks can be monitored only during the finetuning phase when a MLP is created using its pretrained weights, the classification error of a MP-DBM can be monitored from the start of training. Finally, it does not need greedy layer-wise pretraining of DBM that is known to find suboptimal solutions. The training procedure for each layer can be optimal if it would consider also the influence of deeper layers but, as we discussed before, it does not. A MP-DBM avoids this issue training a single model in a single phase.
As stated before, in this work we want to test the performance of DBM in image classification and, therefore, it is critical to understand how it is possible to deal with images in machine learning. In this chapter we show how images can be processed, before using them for the training algorithm, in order to make them more suitable for learning and we will provide an overview of the methods involved. After that, we focus on generative models and their published results on image classification tasks, discussing them and, finally, introducing the dataset we used to test the effectiveness of DBMs on complex images classification.

2.1 Image and preprocessing

A dataset content can be modified preprocessing each example in many possible ways to make it more suitable for training. Dealing with images, there are some common operations that are sometimes very useful or even mandatory. Considering the constant size of the input layer of a neural network (i.e., the number of neurons equal to the number of pixels of the image), if the dataset contains images of different sizes they must be resized to fit the size of the visible layer of the network. Thus, all images need to have the same size and cropping and/or resizing them can be one of the preprocessing steps.

Also the number of channels has to be considered together with the range of possible values of pixels. Grayscale images often have a range of $[0,255]$ for each pixel while in RGB images this
range is used for each color channel. This range needs to be small to be more suitable for training because large ranges are known to slow down the convergence of the training algorithm\cite{4}. For instance, it is very common to rescale the input range into a $[0, 1]$ or $[-1, 1]$ range. Another way to shrink the range is to modify it in such a way that the distribution of each example has zero mean and unit variance by means of data normalization, as done\cite{13} on CIFAR\cite{14} dataset. This allows, in some cases, to avoid learning some statistics in the data that are not significant for the training of the network.

An image can be modified with techniques such as whitening and contrast enhancement too. The former aims to decrease the degree of correlation of a pixel with its adjacent ones. This property is very desirable because it eliminates hidden structures of pixels inside the image. Indeed, if pixels are highly correlated among each other, a network will learn the whole structure composed by the correlated pixels and it will be unable to generalize its internal representation of the target object. The latter technique enhances the contrast of the images and this can help, for instance, in some situations where the target object is not clearly visible such as in dark images whose colors can be enhanced to make it more understandable.

For these reasons, when approaching a new datasets, it is critical to take all the preprocessing steps into consideration because performance of the training of a network can really improve if the right ones are used.

\section{2.2 Neural networks and images}

Our visual cortex is the result of years and years of evolution process that made it very efficient at recognizing the reality surrounding us. Dealing with computers and more particularly
with robots, it is quite reasonable to state that image recognition is a critical task. Robots can rely on various sensors and are able to perceive the environment around them. However none of these raw sensors would be useful without a system to elaborate the informations they get and acquire knowledge from them. This is an issue because retrieving informations is usually trivial, but the elaboration of them can be very difficult.

Even though it is very hard to achieve human image recognition performance, years of machine learning researches brought to the development of methods that guarantee high accuracy in certain tasks such as hand-written digits recognition (e.g., MNIST dataset\cite{15}), faces recognition (e.g.,\cite{16}) and even classification of general images (e.g., ImageNet\cite{17}).

MNIST is a dataset of 60000 training images and 10000 test images of 28x28 pixels of hand-written digits where each image contains a hand-written white digit from 0 to 9 on a black background (Figure 11). It is relatively simple because of the uniformity of its content: each
image has black and white colors, the digit is always centered and not rotated, there are no lights or shadows; the challenging feature is that they are different from each other according to the calligraphy of the writer. According to this fact, a model has to learn a generalized idea of the images it is learning to correctly classify the digits written by a writer never seen before. Many machine learning techniques have been able to reach a very low error rate on the test set of MNIST\textsuperscript{1}. As a matter of fact, MNIST has been very useful for the early models of machine learning, but nowadays it is way too simple to significantly prove the effectiveness of a model. Nevertheless, MNIST is often used as the starting point to test the model, before moving to more complex datasets.

In tasks where the images to be recognized may be pictures of animals, landscapes, objects and so on, it is very likely they have some problematic aspects. One of the main problems of these cases is the presence of a background behind the object to be classified. For instance, if the dataset is composed of pictures with animals in their environment, classification becomes hard due to the need for distinguishing the object from the background. To deal with this problem \textit{image segmentation} partitions an image into sets of pixels that are more meaningful and easier to analyze. This process applies a label to every pixel of the image according to some criteria; after that, pixels belonging to the same class or sharing some properties can be grouped in the same set. For instance, one criterion to segment an image may be to build sets with pixels with the same color, intensity or texture.

\textsuperscript{1}Results of the best models are published on the MNIST website\textsuperscript{[15]).}
Another issue is the possibility to have difficult images such as cases where the object is rotated, not centered or represented with different illumination. All of these are important problems that make the classification of complex images more difficult, but they are also useful because a network trained with them learns different representations of the same object and becomes able to classify it in different circumstances. In other words, they allow the network to be much more able to *generalization* that is very important in order to build an efficient model.

### 2.3 Object recognition with generative models

As stated before, the purpose of this thesis is to evaluate the efficiency of DBMs in discriminative tasks with a complex dataset; as a matter of fact, they have been proved to perform well on simple datasets, such as MNIST, but only few examples on more difficult datasets are available. In this work, we started using MNIST to replicate the results published by Hinton and Salakhutdinov\textsuperscript{[1]}, in order to have a prove of the correctness of the implementation of our training procedure. Then, we moved to test the effectiveness of DBMs on a more complex dataset that we introduce in the next section. Hinton and Salakhutdinov\textsuperscript{[1]} show the discriminative performance of DBMs in two experiments using MNIST and NORB datasets.

In the first experiment they have been able to reach the best published results (in 2012) of 0.95% error rate on the full test set of the permutation invariant version of MNIST. In another article\textsuperscript{[18]}, they show that the error rate becomes 0.79% if dropout\textsuperscript{1} is used in the training.

\textsuperscript{1}Dropout is a variant of standard backpropagation. It performs backpropagation steps turning off with a certain probability each neuron (generally 0.8 for visible neurons and 0.5 for hidden ones). This avoids the learning of hidden structures of correlated features that are not useful, and potentially harmful, for the training of the network.
finetuning phase. These results show that generative model can be able not only to reach, but also to overcome pure discriminative models. Indeed, the 0.79% test error outperformed regularized nonlinear Neighbourhood Component Analysis (NCA) (Salakhutdinov & Hinton, 2007), linear NCA (Goldberger, Roweis, Hinton & Salakhutdinov, 2004), a stack of greedily pretrained autoencoders (Bengio, Lamblin, Popovici & Larochelle, 2007) and DBNs\textsuperscript{10}.

The second experiment has been performed on the NORB dataset with the purpose of testing DBMs with a more difficult dataset than MNIST. NORB is a collection of 50 different 3D toys with 10 objects in each of five generic classes: cars, trucks, planes, animals and humans. Each object is photographed from different viewpoints and under various lighting conditions. The training set contains 24300 stereo images representing 25 objects, 5 per class, whereas the test set contains 24300 stereo images of the remaining different 25 objects. Each image has 96x96 pixels with integer grayscale values in the range $[0,255]$. With image preprocessing and, using a considerably large DBM and a trick that we use in our work and discuss in the next chapter, they have been able to reach an error rate of 10.8% on the full test set that outperformed the 11.6% obtained by Support Vector Machines (SVMs) (Bengio LeCun, 2007), 22.5% achieved by logistic regression and 18.4% achieved by the K-nearest neighbors (LeCun et al., 2004).

The same experiments were performed in order to compare the effectiveness of MP-DBMs and DBMs\textsuperscript{12}. The implemented MP-DBM has been able to reach a test error rate on the MNIST test set of 0.91% which is better than the 0.95% of DBM without dropout, but considerably worse of the one with dropout. The test error obtained on NORB is 10.6%, which is a slight improvement of the DBM one.
Both these results show the promising potential of these two models and this is the main reason why we chose to use them, focusing more on DBMs that give better performance, to test the effectiveness of generative models in classification of a complex dataset which we will present now.

2.3.1 The German Traffic Sign Recognition Benchmark

The German Traffic Sign Recognition Benchmark (GTSRB) dataset\cite{2} is taken from an image classification challenge held at the International Joint Conference on Neural Networks (IJCNN) in 2011. This challenge consisted in a multi-class, single-image classification task to be performed on a set of road signs photographed from a car in Germany. It comprises a training set of 39209 images from 43 classes of road signs and a test set of 12630 images. The examples are RGB pictures, not necessarily squared and their size varies from $15 \times 15$ to $250 \times 250$. As it can be seen in Figures 12 and 13, GTSRB contains both easier and more difficult images; in the latters the road signs can be out of focus, not centered, too bright, too dark and so on. Moreover the majority of the images have a complex background.
Images in the training set are organized in 43 folders, each containing picture of road signs of the same class. Furthermore, images in each folder are divided in groups of 30 examples of exactly the same road sign captured in 30 different positions that are closer and closer to it. Indeed, the 30 photos had been taken when the car was moving in the direction of the road sign and therefore it is possible to see the sign getting bigger and bigger as the car is approaching it (Figure 14). Together with the dataset, the coordinates of the bounding box (a rectangular section of the image inside which the road sign is inscribed) are available for each example, so it is possible to crop this rectangle from the initial image, removing part of the background and centering the sign.
This dataset is more difficult than both MNIST and NORB for some reasons. It has 43 classes of road signs that is much larger than the number of classes of the other datasets. Images can be out of focus, not centered and, like NORB, subject to different lighting conditions. One of the most difficult features of GTSRB is the background of the environment and, sometimes, even the obstacles in front of the road signs. Moreover, images are in RGB and this increases the dimension of each example and, as a consequence, the dimension of the input layer of the network. It could be possible to convert the pictures in grayscale, but intuitively (and as demonstrated by experiments that will be presented at the end) this causes a loss of information because colors of road signs are very informative; indeed, their colors give a hint about the identity of the sign. Colors of road signs can be: red, black, yellow, white, blue and, for road signs with traffic light, green. Colors and also the shapes of the signs, may be features that the network extract from the image and can help in the discrimination of the identity of the sign.

These features of GTSRB have been taken into account for the preprocessing phase and, later, we present and discuss each of the preprocessing steps we chose.
CHAPTER 3

PROJECT ARCHITECTURE AND IMPLEMENTATION

In this chapter we explain the choices we made for implementing and performing experiments on deep generative models focusing on the machine learning framework we decided to use and on some technical aspects about the algorithms to train our models and the image preprocessing techniques.

3.1 Libraries

To implement the DBM and MP-DBM we used Pylearn2[3], a machine learning framework written in Python[19] and Theano[20,21], a Python library used by Pylearn2.

Pylearn2 is based on Theano, a Python library that allows to define, optimize and evaluate efficiently mathematical expressions involving multi-dimensional arrays. It provides a tight integration with NumPy[22] and allows the use of GPU by compiling Python code into nVIDIA[23] Compute Unified Device Architecture (CUDA)[24] language that can be in turn executed by a nVIDIA GPU faster than on CPU. It has been fundamental for our work because it combines aspects of a computer algebra system with aspects of an optimizing compiler, allowing the computation of repeated complex mathematical expressions to run very fast.

Pylearn2 is a machine learning library written in Python and it is based on Theano. It offers a large variety of models, algorithms and other useful tools to make it easier to build, train and analyze machine learning models. For the purpose of this work, we decided to choose
Pylearn2 because of this reason and also for the ability of Pylearn2 to exploit Theano functions to enable the use of GPU.

Pylearn2 is composed of several packages in which all the necessary components of an experiment are organized in Python classes or scripts. For instance, each machine learning model is implemented as a Python class with the respective variables (e.g., the number of neurons in a neural network) and functions (e.g., the function to set a neural network weights). The standard way to implement a training procedure in Pylearn2 is to load a train variable, that is an object containing all the components of the experiment, and then run the main_loop function to perform the training of the model. The components of the experiment are read from a yaml file and they can be mandatory or optional. Mandatory ones are: the dataset, the model, the type of training algorithm and the cost function.

The optional components are extensions that allow to customize the behavior of the experiment such as:

- adding monitor channels to check validation or test error at each epoch,
- saving the best model found till the current epoch,
- adjusting learning rate or momentum during the training,

Besides to the components of the experiment, the yaml file also contains the value of the involved learning parameters. The advantage of using a yaml file is that all the details of each experiment are recorded in it making the experiment reproducible. Moreover, when trying several experiments with different parameters, the training script does not need to be changed except for the path of the yaml file to be read.
3.2 Learning parameters

To reach high performance with the DBM and MP-DBM, we had to tune the learning parameters of their training algorithms; for the DBM:

- RBMs training for the greedy layer-wise pretraining,
- DBM training,
- DBM finetuning,

and the MP-DBM training for the MP-DBM, that has only a single phase of training.

Some of these parameters are general and they are used in any of these algorithms whereas others are specific only for some of them. We explain all parameters we used starting from the general ones and, then, the parameters involved in any single training phase.

3.2.1 General parameters

- **Number of epochs** is the maximum number of epochs to train our models: if this parameter is set, the training stops after the specified number of epochs. This is not the only way to stop training, there are other techniques using other stopping conditions: for instance, *early stopping* stops the training after that the *validation error*\(^1\) begins to increase significantly even though the error on the training set continues to decrease. This

\(^1\)The percentage of error on the validation set.
happens when the network starts to overfit\textsuperscript{1} the training set and stopping the training with early stopping avoids this issue.

- **Number of layers** specifies how many layers compose the model. RBMs are composed of only one layer, DBMs are composed of a number of hidden layer equal to the number of stacked RBMs and the MLPs we used for DBM finetuning are composed of the number of hidden layers of the DBM plus the softmax layer.

- **Number of neurons per layer** specifies how many neurons belong to each layer.

- **Training algorithm**: in Pylearn\textsuperscript{2}, this parameter specifies whether *stochastic gradient descent*\textsuperscript{2} or *batch gradient descent*\textsuperscript{3} is used.

- **Batch size** determines the number of examples to be contained in each batch.

- **Learning rate** is the learning rate of the weights update equation.

- **Momentum** is the momentum of the weights update equation. This parameter helps to avoid excessive oscillations in narrow valleys of the error function.

\textsuperscript{1}Overfitting happens when a model learns to classify the examples in the training set, but is not able to correctly classify others never seen before.

\textsuperscript{2}Each weights update is performed using a single batch. An epoch finishes after that all batches have been used (i.e., after that a number of weights updates equal to the number of batches has been done).

\textsuperscript{3}Normally, batch gradient descent uses the whole training set for each weights update. However, in Pylearn\textsuperscript{2}, it can also be used like stochastic gradient descent, but with the difference that the step size along the direction of the gradient is computed with *partial line search*: a technique that computes the direction and step size of the update in a different way from the weights update formula used in stochastic gradient descent. Partial line search will not be explained here because we have not used it in our experiments on GTSRB.
• **Weights initialization** is the way the weights of our models, with the exception of the weights initialized with pretrained weights, are initialized\(^1\). In this work, we initialize weights randomly picking a value inside a specified range (e.g., \([-0.001;0.001]\)).

• **Biases initialization** is the way the biases of the neurons of our models, with the exception of the biases initialized with pretrained biases, are initialized\(^2\).

• **Weight decay** specifies the penalty to the cost function. This penalty is applied in order not to let the weights increase too much which is a well-known cause of overfitting.

• **Toronto sparsity** is a particular way to force the sparsity\(^3\) of the model. Here we do not explain this parameter because we do not use it in our experiments on GTSRB.

### 3.2.2 Restricted Boltzmann Machine and Deep Boltzmann Machine parameters

• **Number of Markov chains** specifies how many different Markov chains are simultaneously run. The results of the \(M\) Markov chains are finally averaged and used for the weights update formula.

---

\(^1\)The weights of the DBM are initialized with the pretrained weights of the RBMs and the MLP weights are initialized with the trained weights of the DBM.

\(^2\)The biases of the DBM’s neurons are initialized with the pretrained biases of the RBMs and the neurons biases of the MLP are initialized with the trained biases of the DBM.

\(^3\)A matrix is sparse when the majority of its elements is zero. Forcing sparsity in a neural network, means training it forcing the majority of its neurons to be inactive. This allows the network to learn relevant structures in the data similarly to what dropout does.
• **Number of Gibbs steps** specifies how many Gibbs steps are performed for each Markov chain.

• **Number of mean-field updates** is the number of updates to estimate the data-dependent distribution. In RBM this is set to 1 because this is enough to reach convergence.

### 3.2.3 Deep Boltzmann Machine finetuning parameters

The training of the DBM is followed by its finetuning which, as explained in Chapter 2, is done building a MLP using the trained weights and biases of the DBM, adding a softmax layer at the top, and augmenting the dataset together with the visible layer of the model. The parameters involved in this procedure are:

- **Number of mean-field updates** to initialize the second hidden layer’s neurons for data augmentation.

- **Number of line searches** specifies how many direction are checked in order to find the best direction to descent the gradient of the error in partial line search.

- **Dropout inclusion probabilities** specify the probability with which neurons are considered in the iterations of backpropagation. Backpropagation without dropout sets all these probabilities to 1.

### 3.2.4 Multi-Prediction Deep Boltzmann Machine parameters

The training of the MP-DBM involves:

- **Number of mean-field updates** to estimate the data-dependent distribution, as with DBMs.
• **Drop probability** of the visible neurons of the model. This parameter specifies the probability with which each visible neuron is dropped in the Multi-Prediction training algorithm.

Now, we show the implementation in Pylearn2 of the DBM and MP-DBM training procedures.

### 3.3 Code implementation

The whole procedure to train DBMs (pretraining, training and finetuning) described by Hinton and Salakhutdinov was not implemented in Pylearn2, while Contrastive Divergence to train RBMs, the algorithm to train DBMs, backpropagation and the training procedure of MP-DBMs were implemented. In addition to that, Pylearn2 requires a wrapper class for each dataset that has the purpose of loading the dataset in a data structure that Pylearn2 is able to read and use for training, but the wrapper for GTSRB was missing. Therefore we had to implement:

• the wrapper class for GTSRB,
• the training for a Gaussian Restricted Boltzmann Machine (GRBM), a model that will be explained later,
• the training for the DBM that merges the single phases of training,
• the training for the MP-DBM,
• scripts for preliminary experiments.

We explain each of this component in the following sections.
3.3.1 German Traffic Sign Recognition Benchmark wrapper

Pylearn2 offers wrapper classes for a lot of datasets that are generally used in machine learning, but not for GTSRB. To write it, we followed the technique used in other wrappers of image datasets adapting it to our case. To train a model, Pylearn2 requires a matrix $X$ of $N \times M$ dimensions where $N$ is the number of examples in the dataset and $M$ is the number of values per example. In supervised tasks, also the label matrix $Y$ is required and it can be specified using either a one-hot configuration or the label number of each example. In the one-hot case, the matrix has a number of rows equal to the number of examples in the dataset and a number of columns equal to the number of classes. Each row has all zeros except for the index of the class the example belongs to, that will be 1. In the case where the output is the label number, the number of rows are the same as before, but each row contains just an integer representing the class. The value of the pixels that are loaded in the $X$ matrix are first normalized in such a way that the pixels distribution of each image has a mean of zero and a variance of one, as explained in the data preprocessing section of the previous chapter.

Dealing with grayscale images, the $X$ matrix contains the values of the pixels of an image from top-left to bottom-right. For instance, taken the fifth image of the dataset, the value of the pixel with index $(i, j)$, will have index $(5, i * j + j)$ inside $X$. RGB images are modeled similarly, but their three channels are splitted in such a way that each row will contain the pixel

\footnote{If $i$ starts with 0. This is the case for the first index of the arrays in the math Python library Numpy that we used in this work.}
values of the image in $M \times 3$ columns inside which all the reds, all the greens and all the blues are stored in this order.

### 3.3.2 Gaussian Restricted Boltzmann Machine training

As discussed before, the input layer of a DBM is composed of stochastic binary units; however, this is not mandatory. For simple datasets (e.g., datasets with grayscale images), it is recommended to use binarized visible units because it reaches convergence faster, but this does not work for more complex datasets such as datasets with RGB images where binarization of the RGB values of the pixels reduces the accuracy of the input representation.

It is possible to use the activation function of the visible units, instead of the binarized one, but the hidden units are recommended to be always binary\[8\]. In this work, we used a technique\[1\] to train DBM on NORB dataset that requires to first train a GRBM on the images of the dataset, and then use the activations of its hidden neurons for each image as input of the DBM (Figure 15: the units on the bottom of the GRBM are the gaussian ones and the units on the top of the GRBM are the binary ones). A GRBM, indeed, solves the lack of accuracy of binarized logistic units replacing them with linear units with independent Gaussian noise. Its energy function is:

$$E(v, h) = \sum_i \frac{(v_i - a_i)^2}{2\sigma_i^2} - \sum_j b_j h_j - \sum_{i,j} \frac{v_i}{\sigma_i} h_j w_{ij}$$  \hspace{1cm} (3.1)$$

where $i$ is the index of a visible neuron, $j$ is the index of a hidden neuron, $v_i$ and $h_j$ are respectively the states of visible neuron $i$ and hidden neuron $j$, $a_i$ and $b_j$ are respectively visible
and hidden neuron biases, $w_{ij}$ is the weight between visible neuron $i$ and hidden neuron $j$ and $\sigma_i$ is the standard deviation of the Gaussian noise of visible neuron $i$.

The training of the GRBM has the same parameters of the training of the RBMs with the addition of $\sigma_i$ that can be initialized with an initial value and be learned as well as the weights and biases or kept constant during the training. The GRBM acts as a “preprocessing” step for the dataset because the DBM uses the features (e.g., hidden neurons states) of each input captured by the GRBM in its hidden nodes instead of the examples of the dataset. This is necessary because using a Gaussian visible layer directly on the DBM would make the training much slower because the training algorithm for DBM works well with binary units and, conversely, it requires a much greater number of weights updates with real number as inputs[25].

3.3.3 Deep Boltzmann Machine training

As discussed before, the training procedure for DBM involves the training of four models. Pylearn2 offers the training algorithm for all of them, but they are separated from each other. Thus, to implement the whole training procedure for DBMs, we needed to find a way to use all of them sequentially.

The procedure starts with the greedy layerwise pretraining of the RBMs. The first RBM is trained using the “preprocessed” examples from the GRBM as inputs. To do this, we used the Transformer_Dataset class for the first RBM, which takes the dataset to be transformed and the model that has to transform it, as inputs. This class transforms the data examples and the RBM uses them for its training. After that, the second RBM has to be trained using the
activation of the hidden states of the first RBM as inputs. This procedure is done by Algorithm 1.

Subsequently, weights and biases of the RBMs are clamped in the respective DBM layers that can be trained with the DBM training algorithm. After that, the dataset augmentation for training the MLP has to be performed. We implemented it using the $mf$ function, belonging to the DBM class, that performs mean-field inference on it for a certain number of steps selectable by the user. Then the MLP is created using an augmented visible layer with a number of neurons equal to the sum of the original visible neurons and the neurons of the second hidden layer of the DBM where, for each example of the dataset, the former contains its values and
Algorithm 1 Greedy layerwise pretraining

2. Randomly initialize weights and biases of GRBM.

// Contrastive Divergence for GRBM
3. for each batch b, b=1 to B do
4. for each training example i, i=1 to batch size do
5. Apply Contrastive Divergence with G Gibbs steps.
6. end for
7. end for
8. Make a dataset with the training examples preprocessed by GRBM. It will be used as dataset for RBM1.

// Contrastive Divergence for RBM1
10. for i=1 to number_of_epochs_for_RBM1 do
11. for each batch b, b=1 to B do
12. Apply Contrastive Divergence with G1 Gibbs steps.
13. end for
14. end for
15. Make a dataset with the GRBM preprocessed training examples preprocessing them with RBM1. It will be used as dataset of RBM2.

// Contrastive Divergence for RBM2
17. for i=1 to number_of_epochs_for_RBM2 do
18. for each batch b, b=1 to B do
19. Apply Contrastive Divergence with G2 Gibbs steps.
20. end for
21. end for
the latter contains the activation states of the second hidden layer’s neurons found before with
the mean-field steps with that example clamped into the visible layer.

As with the pretrained weights of the RBMs, weights and biases are initialized taking the
trained ones of the DBM and clamping them to the MLP, but the weights connecting the
added neurons in the augmented section of the visible layer to the first hidden layer’s ones, are
initialized with the respective weights connecting the first hidden layer to the second hidden
layer. This procedure is shown in Figure 16 where many arrows are omitted to make the picture
more readable. The figure on the left shows the activation states of the second hidden layer
after some mean-field steps with a training example clamped on the visible layer. The figure
on the right shows the augmented input of the MLP that is a concatenation of the training
example $i$ and the respective activation of the second hidden layer’s neurons. Eventually, the
MLP is trained with backpropagation. Note that the softmax layer that is added at the top of
the DBM to build the MLP, has biases and connected weights that are initialized randomly due
to the fact that they were not part of the pretrained DBM. However, this is not relevant because
they can be trained quickly without losing effectiveness. The whole training algorithm is shown
in Algorithm 2. Finally, Algorithm 3 briefly resumes the complete DBM training procedure
focusing on the way each model is used by the following model involved in the procedure.

3.3.4 Training of Multi-Prediction Deep Boltzmann Machine

MP-DBM needs just one model and one training procedure to be trained: the procedure
we implemented loads a unique yaml file and trains a MP-DBM according to that. However,
to reach our best performance on GTSRB with MP-DBM, we needed to finetune it, as done
Algorithm 2 DBM training script

2. Make a dataset with the training examples preprocessed by GRBM.
3. Clamp pretrained weights and biases into DBM.
4. // Training of DBM
   for i=1 to number_of_epochs_for_DBM do
      for each batch b, b=1 to B do
         update weights and biases of the DBM.
      end for
   end for
5. // Data augmentation
   for each preprocessed training example i do
      Perform MF mean-field steps to activate the hidden neurons of the DBM.
      Save activation state of the neurons in the second layer.
   end for
6. Make an augmented dataset concatenating the training examples with the respective activation of the neurons of the second hidden layer. It will be used as dataset of MLP.
7. Clamp pretrained weights and biases into MLP with the augmented visible layer.
8. Randomly initialize the weights and biases of the softmax layer of MLP.
9. // DBM finetuning
   for i=1 to number_of_epochs_for_MLP do
      for each batch b, b=1 to B do
         update weights and biases of MLP.
      end for
   end for
Algorithm 3 Complete DBM training procedure

// GRBM training
1. Randomly initialize weights and biases of GRBM.
2. Train GRBM.

// RBM1 training
3. Make a dataset with the training examples preprocessed by GRBM. It will be used as dataset of RBM1.
4. Randomly initialize weights and biases of RBM1.
5. Train RBM1.

// RBM2 training
6. Make a dataset with the GRBM preprocessed training examples preprocessing them with RBM1. It will be used as dataset of RBM2.
7. Randomly initialize weights and biases of RBM2.
8. Train RBM2.

// DBM training
9. Make a dataset with the training examples preprocessed by GRBM. It will be used as dataset of DBM.
10. Clamp pretrained weights and biases into DBM.
11. Train DBM.

// Data augmentation
12. Perform MF mean-field updates with each training example preprocessed by GRBM to activate the neurons of the second hidden layer and save their states.
13. Make an augmented dataset concatenating the training examples with the respective activation of the neurons of the second hidden layer. It will be used as dataset of MLP.

// DBM fine-tuning
14. Clamp pretrained weights and biases into MLP with the augmented visible layer.
15. Randomly initialize the weights and biases of the softmax layer of MLP.
16. Train MLP.
with the DBM. This is acceptable because finetuning has also been used to reach the published results on NORB dataset\cite{12}.

Pylearn2 tutorials implement several training procedures in order to show the suggested way to build up an experiment with Pylearn2. They are very different from each other and cover almost all kind of training procedures for a large variety of models showing how it is possible to perform them. However, testing the correctness of the implemented procedures on Pylearn2 is difficult, especially when the procedure is complex as the ours. Indeed, checking the code by inspection or debugging it, can be not enough to test its correctness: a wrong training procedure can finish its execution without errors and the trained model can give good performance despite the fact it has been trained with a wrong implementation of the training procedure. One way to check the correctness of an implementation is trying to replicate the results obtained in
the literature with the same procedure and comparing them: if the results are the same, it is strongly guaranteed that the procedure is correct. According to this method, we checked the correctness of our DBM training procedure trying to reach the same performance on MNIST published by Hinton and Salakhutdinov\cite{1}. We have been able to reproduce the 0.95% test error published in the article using all the instructions and parameters given in it and, for parameters that were not published in the paper, also checking the complete code published on Salakhutdinov’s website\cite{26}. This gives us enough confidence to state that the implementation of our DBM training procedure is correct and we also made a pull request on the GitHub\cite{27} repository of Pylearn2 that is now under revision.

3.4 German Traffic Sign Recognition Benchmark preprocessing

The implementation of the DBM training procedure has been followed by the analysis of GTSRB to understand the better way to approach it. We started considering the work done that obtained the best discriminative performance in the competition (0.54% test error)\cite{28}, where they show some preprocessing methods to enhance the contrast of images. We used these methods to train a relatively small model to understand which of them was the best in an acceptable training time and we planned to use it in our final model which would have taken more time to be trained. We also considered to use deblurring methods, but after some experiments we discarded this option because we saw that blurred images were correctly classified by the large majority of trained model, thus there was no need to use them.

Now we explain each preprocessing function we used showing how they change the images in Figure 17 and, in the next chapter, we present the experimental results we obtained using
them. For each preprocessing method, images were firstly cropped according to the bounding box coordinates, then resized at $32 \times 32$ to fit the number of visible neurons we chose for our GRBM and, after the transformation, normalized.

![Figure 17: Standard GTSRB examples](image)

### 3.4.1 Enhancing contrast

As shown in Figure 17, GTSRB contains a lot of dark images, thus contrast enhancing is very useful to make them more suitable for training.

Contrast can be enhanced rescaling the intensities of pixels in the range of 0 and 255. This method enlarges the small range of pixels intensities (e.g., 0-30) of dark images and rescales it between 0 and 255 enhancing the dark colors. This method can be applied on RGB images
modifying the range of each channel, but it is more appropriate with images in L*a*b* color space.

In L*a*b*, the intensity of a pixel is expressed by a single coordinate instead of one for each channel such as in RGB; therefore it is possible to enhance the contrast by rescaling the lightness coordinate of each pixel. To do this we first transformed each RGB image in L*a*b*, applied the method and then transformed the L*a*b* images in RGB obtaining the results shown in Figure 18.

Contrast can also be enhanced with the histogram equalization method that makes the histogram of an image flat. This histogram represents, for each pixel intensity between 0 and 255, how many pixels with that intensity there are in the image. In dark pictures, the histogram is clearly unbalanced on the left because of the presence of a lot of pixels with a low intensity.

Figure 18: Results of the application of pixels intensities rescaling to the images in Figure 17
This technique allows to balance the distribution of pixels intensities yielding to clearer images. We applied this technique to the lightness coordinate of the L*a*b* color space as before, obtaining the images in Figure 19.

Another technique, similar to the previous one, called adaptive histogram equalization, divides the image in several sections and then computes the histogram for each of them equalizing it as before. This helps to improve local contrast (Figure 20) and this has been very helpful in our work because we were interested in enhancing the contrast of the road sign with an histogram that does not take into account the intensity of pixels belonging to the background. Finally, since we were not sure that converting the L*a*b* images to RGB after the transformation was the right thing to do, we decided to make an experiment using the transformed images
in L*a*b* space directly. We preprocessed the L*a*b* images with adaptive histogram equalization because this method was the one that performed the best in the experiments presented in the following chapter and the resulting images are shown in Figure 21.

We tested also images converted in other color spaces without performing any transformations. Figure 22 shows the resulting images in both Hue-Saturation Value (HSV) and chromaticity spaces.

3.4.2 Comments

The preprocessing method that performed better was adaptive histogram equalization with RGB images. This may seem strange because some images preprocessed with this method look worse than the initial ones, but it is not right to state what image is better for a neural network
simply looking at it. Indeed, images that may look better for a human, are not guaranteed to be good also for a neural network, and vice versa.

Once we understood that adaptive histogram equalization was the best method to preprocess the images, we applied it and used the resulting images to do the final experiments with the purpose to reach the best classification rate, whose results are shown in the next chapter.
Figure 22: Results of the conversion of images in Figure 17 in HSV (left) and chromaticity (right) spaces.
All the experiments have been run on a server with Linux\textsuperscript{[29]} Ubuntu\textsuperscript{[30]} 14.04.1 Long-Term Support (LTS) (kernel GNU/Linux 3.13.0-32-generic x86\_64) with a nVIDIA TeslaC1060\textsuperscript{[31]} GPU, an Intel(R)\textsuperscript{[32]} Core(TM) i7\textsuperscript{[33]} CPU 920 @ 2.67GHz CPU with 8 cores, a 6GB RAM and the following version of the libraries we used: Numpy 1.8.2, Scipy\textsuperscript{[34]} 0.13.3, Theano 0.6.0 and Pylearn2 last commit 2b516212b5.

4.1 MNIST

The first experiments had the purpose to replicate the results of Hinton and Salakhutdinov\textsuperscript{[1]} on the MNIST dataset in order to test the correctness of our implementation of the DBM training procedure. To be as accurate as possible, we needed to collect all the learning parameters they used to reach their results and we were able to find them in the article and on the Matlab code available on Salakhutdinov’s website. The latter reports all the parameters values and this was helpful for some of them whose value was not reported by the former.

For a few parameters the values in the article and the ones in the code are different and we decided to choose the one published in the former considering them more reliable; we did not try both, but it is likely that they both work well. The whole MNIST training set of 60000 examples was used for training and the whole test set of 10000 examples for testing. We now
TABLE I: MNIST - FIRST AND SECOND RBMS PARAMETERS FOR GREEDY LAYER-WISE PRETRAINING

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>#Epochs</td>
<td>100</td>
<td>#Epochs</td>
<td>200</td>
</tr>
<tr>
<td>#Visible neurons</td>
<td>784</td>
<td>#Visible neurons</td>
<td>500</td>
</tr>
<tr>
<td>#Hidden neurons</td>
<td>500</td>
<td>#Hidden neurons</td>
<td>1000</td>
</tr>
<tr>
<td>Batch size</td>
<td>100</td>
<td>Batch size</td>
<td>100</td>
</tr>
<tr>
<td>Weights initialization</td>
<td>[-0.001;0.001] *</td>
<td>Weights initialization</td>
<td>[-0.01;0.01] *</td>
</tr>
<tr>
<td>Biases initialization</td>
<td>0</td>
<td>Biases initialization</td>
<td>0</td>
</tr>
<tr>
<td>Learning rate</td>
<td>0.05</td>
<td>Learning rate</td>
<td>0.05</td>
</tr>
<tr>
<td>Momentum</td>
<td>0.5-0.9*</td>
<td>Momentum</td>
<td>0.5-0.9*</td>
</tr>
<tr>
<td>#Markov chains</td>
<td>100</td>
<td>#Markov chains</td>
<td>100</td>
</tr>
<tr>
<td>#Gibbs steps</td>
<td>1</td>
<td>#Gibbs steps</td>
<td>5</td>
</tr>
</tbody>
</table>

explain the parameters used during each training phase and report their values (the values marked with * are commented within the paragraph).

4.1.1 Greedy layer-wise pretraining

The DBM trained by Hinton and Salakhutdinov is pretrained stacking 2 trained RBMs with the greedy layer-wise pretraining explained in Chapter 2. Table I summarizes the parameters used to train the two RBMs. The range next to the weights initialization parameter is the range from which the weights initialization values are picked and the notation used for the momentum indicates that it starts from 0.5 and increases linearly reaching the value 0.9 at the 6th epoch, thereafter it stays constant.
### TABLE II: MNIST - DBM PARAMETERS FOR TRAINING

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>#Epochs</td>
<td>500</td>
</tr>
<tr>
<td>#Visible neurons</td>
<td>784</td>
</tr>
<tr>
<td>#Hidden neurons</td>
<td>500-1000</td>
</tr>
<tr>
<td>Batch size</td>
<td>100</td>
</tr>
<tr>
<td>Weights initialization</td>
<td>-</td>
</tr>
<tr>
<td>Biases initialization</td>
<td>0</td>
</tr>
<tr>
<td>Learning rate</td>
<td>0.005*</td>
</tr>
<tr>
<td>Momentum</td>
<td>0.5-0.9*</td>
</tr>
<tr>
<td>#Mean-field inference steps</td>
<td>10</td>
</tr>
<tr>
<td>#Markov chains</td>
<td>100</td>
</tr>
<tr>
<td>#Gibbs steps</td>
<td>5</td>
</tr>
<tr>
<td>Weight decay</td>
<td>0.0002</td>
</tr>
</tbody>
</table>

### 4.1.2 Deep Boltzmann Machine training

After the greedy layer-wise pretraining, the RBMs are stacked to compose a DBM. The DBM is trained with the values in Table II. The weights and biases of the DBM are initialized using the RBMs trained ones. The momentum adjusting has been done the same way as for the RBMs. The learning rate starts from 0.005 and is decreased as $10/(2000 + t)$ where $t$ is the number of updates so far. After training, the quality of the weights learned by the DBM can be understood by checking how the model reconstructs some examples of the dataset. The reconstructions are made by clamping an example to the visible layer, estimating the hidden units with mean-field and finally performing one mean-field top-down pass to estimate the activation of the visible layer. Figure 23 shows 50 pairs of training examples and their respective reconstructions arranged one next to the other. For instance, the top left number is
an example of 8 taken from the training set and the number on its right is the reconstruction of it made by the DBM. As it can be seen, the reconstructions are very accurate and therefore it can be stated that the learned weights work well.

4.1.3 **Deep Boltzmann Machine finetuning**

The DBM has been finally used to build a MLP. The augmented dataset is computed performing 1 mean-field update for each example. Then the MLP is supervisedly trained with *conjugate gradient descent*, a method to solve unconstrained optimization problems that will not be discussed here since we have not used it for our work on GTSRB. The parameters are shown in Table III. Learning rate and momentum are omitted because conjugate gradient descent does not use them. After finetuning, the model reaches the 0.95% of test error at epoch 25 achieving the same result of Hinton and Salakhutdinov.

It is interesting to see that at the begininning of finetuning, after epoch 0 where the softmax layer weights are initialized randomly and this causes the initial test error to be very large, it suddenly becomes about 1.7% after one epoch (Figure 24), showing how the unsupervised training of the DBM is able to reach excellent discriminative results on MNIST. We wanted to
TABLE III: MNIST - MLP PARAMETERS FOR FINETUNING

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>#Epochs</td>
<td>100</td>
</tr>
<tr>
<td>#Visible neurons</td>
<td>1784</td>
</tr>
<tr>
<td>#Hidden neurons</td>
<td>500-1000-10</td>
</tr>
<tr>
<td>Batch size</td>
<td>5000</td>
</tr>
<tr>
<td>Weights initialization</td>
<td>[-0.05;0.05]</td>
</tr>
<tr>
<td>Biases initialization</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 24: MNIST - Test error diagram
test also the dropout technique\cite{18} trying to reach the 0.79\% of test error presented in the paper. The article does not report any details about the DBM training and we used the same learning parameters of the DBM of the experiments without dropout, reaching a 0.84\% test error that we considered acceptable. The parameters used to perform stochastic gradient descent with backpropagation using dropout are shown in Table IV. The momentum starts from 0.5 and it is increased linearly to 0.99 during the 500 epochs of training.

### 4.2 German Traffic Sign Recognition Benchmark

After we had enough confidence about the correctness of our implementation of the DBM training procedure, we started to test it on GTSRB. In all the experiments, the images were cropped according to their bounding box coordinates ensuring that the road sign is centered and the majority of background is cropped off the image. Then, before training, the images have been resized at the dimension of $32 \times 32$. With this dimension, the visible layer of the

\begin{table}[h]
\centering
\begin{tabular}{|c|c|}
\hline
Parameter & Value \\
\hline
#Epochs & 500 \\
#Visible neurons & 1784 \\
#Hidden neurons & 500-1000-10 \\
Batch size & 100 \\
Weights initialization & [-0.01;0.01] \\
Biases initialization & 0 \\
Learning rate & 1 \\
Momentum & 0.5-0.99* \\
Include probabilities & 0.8-0.5-0.5 \\
\hline
\end{tabular}
\caption{MNIST - MLP PARAMETERS FOR DROPOUT FINETUNING}
\end{table}
networks had to be composed of $32 \times 32 \times 3 = 3072$ neurons\(^1\). We initially considered to resize the images at the dimension of $48 \times 48$ that is the average dimension of the GTSRB images, which we recall varies from $15 \times 15$ to $250 \times 250$, but this would have resulted in a visible layer composed of $48 \times 48 \times 3 = 6912$ neurons that would have been too large to manage with our computational resources.

4.2.1 Preliminary experiments for parameters tuning

In the first experiments on GTSRB we tested the performance of the models using the images of GTSRB without preprocessing because we were planning to use the more efficient preprocessed images after that a good tuning of the parameters would have been found. Indeed, we wanted to evaluate the efficiency of the preprocessing methods without having to care about the parameters tuning, thus being able to compare them. These experiments have been done using a training set of 32200 examples, a validation set of 7009 examples and a test set of 12630 examples. In the next paragraphs we show the parameters we found to be the most effective and we discuss the experiments we did to find them.

4.2.1.1 Gaussian Restricted Boltzmann Machine training

As stated before, the GRBM has the purpose to be a “preprocessing step” for the GTSRB images to avoid the large amount of training time that a Gaussian layer for the visible layer of the DBM would require. As a matter of fact, the GRBM is the only model using the GTSRB images because the following ones use the hidden states activations of the GRBM for each

\(^1\)The multiplication by 3 takes into account the three color channels.
example; thus, it is critical to learn very good parameters for the GRBM in order to have a highly reliable activation of the hidden states for each example (i.e., a very accurate abstraction of them).

It is well-known that monitoring the learning of a RBM is not an easy task; there are no significant measures, such as validation or test error, to deduce the progress of learning. Hinton explains that the reconstruction error of the network can be used\cite{8}, but being conscious that it is a poor measure of the process of learning, indeed it is not the function that CD is trying to optimize; the reconstruction error is simply a measure of the differences between the value of the pixels of an image and the values of the pixels of the respective reconstruction. Moreover, the results of experiments cannot be compared using the reconstruction error. Indeed, being a measure of the differences between two neurons, if the number of neurons to be evaluated increases, also the reconstruction error increases. One can say that the reconstruction error can be used to compare networks with the same number of neurons for their visible layer, however this is not always true. If the networks are provided with input data normalized in two different ways (e.g., one receives data in the range $[0; 1]$ and the other receives data in the range $[-1; 1]$), the reconstruction error will be bigger for the network receiving the data with a wider range of values than the other. For these reasons, the reconstruction error cannot state if a network is better than another unless they receive the same data and they have the same number of visible neurons.

Since we needed the GRBM to be very efficient and that there is no other way to monitor its training, we decided to carefully monitor the reconstruction error of networks with the same
TABLE V: GTSRB - GRBM PARAMETERS

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>#Epochs</td>
<td>1000</td>
</tr>
<tr>
<td>#Visible neurons</td>
<td>3072</td>
</tr>
<tr>
<td>#Hidden neurons</td>
<td>4000</td>
</tr>
<tr>
<td>Batch size</td>
<td>100</td>
</tr>
<tr>
<td>Weights initialization</td>
<td>[-0.01;0.01]</td>
</tr>
<tr>
<td>Biases initialization</td>
<td>0</td>
</tr>
<tr>
<td>Learning rate</td>
<td>0.0001</td>
</tr>
<tr>
<td>#Gibbs steps</td>
<td>1</td>
</tr>
<tr>
<td>Sigma initialization</td>
<td>0.4</td>
</tr>
</tbody>
</table>

number of visible neurons and receiving the same input data. We did about ten experiments to tune the parameters in order to have the lowest reconstruction error. In Table V, we show the best learning parameters we found for the GRBM. The experiments showed that increasing the number of hidden neurons, decreases the reconstruction error of the network. Unfortunately, we had to limit the dimension of the network according to the capacity of our computational resource, but, as it can be seen in Figure 25, the reconstructions of the image are satisfying. The figure shows also that the dark images are not reconstructed as well as the clear ones. This issue has been solved with the experiments, explained in the following section, that involve image preprocessing methods.

4.2.1.2 Greedy layer-wise pretraining

The training of the RBMs has been done considering the same issues of the GRBM with the further problem that they do not reconstruct the images of the road sign. This made it harder to understand if the training of the RBM was going well or not, and even to understand
if the final model was good. To do this, we relied only on the reconstruction error, tuning the parameters according to it. We found that a lot of configurations of the parameters gave almost the same results. The best parameters we found, for both RBM, according to the reconstruction error are resumed in Table VI.

TABLE VI: GTSRB - FIRST AND SECOND RBM PARAMETERS FOR GREEDY LAYER-WISE PRETRAINING

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>#Epochs</td>
<td>600</td>
<td>#Epochs</td>
<td>600</td>
</tr>
<tr>
<td>#Visible neurons</td>
<td>4000</td>
<td>#Visible neurons</td>
<td>3000</td>
</tr>
<tr>
<td>#Hidden neurons</td>
<td>3000</td>
<td>#Hidden neurons</td>
<td>3000</td>
</tr>
<tr>
<td>Batch size</td>
<td>100</td>
<td>Batch size</td>
<td>100</td>
</tr>
<tr>
<td>Weights initialization</td>
<td>[-0.001;0.001]</td>
<td>Weights initialization</td>
<td>[-0.01;0.01]</td>
</tr>
<tr>
<td>Biases initialization</td>
<td>0</td>
<td>Biases initialization</td>
<td>0</td>
</tr>
<tr>
<td>Learning rate</td>
<td>0.05</td>
<td>Learning rate</td>
<td>0.05</td>
</tr>
<tr>
<td>#Gibbs steps</td>
<td>1</td>
<td>#Gibbs steps</td>
<td>3</td>
</tr>
</tbody>
</table>
TABLE VII: GTSRB - DBM PARAMETERS FOR TRAINING

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>#Epochs</td>
<td>500</td>
</tr>
<tr>
<td>#Visible neurons</td>
<td>4000</td>
</tr>
<tr>
<td>#Hidden neurons</td>
<td>3000-3000</td>
</tr>
<tr>
<td>Batch size</td>
<td>100</td>
</tr>
<tr>
<td>Weights initialization</td>
<td>-</td>
</tr>
<tr>
<td>Biases initialization</td>
<td>0</td>
</tr>
<tr>
<td>Learning rate</td>
<td>0.0005</td>
</tr>
<tr>
<td>Momentum</td>
<td>0.5-0.9*</td>
</tr>
<tr>
<td>#Mean field inference steps</td>
<td>10</td>
</tr>
<tr>
<td>#Markov chains</td>
<td>100</td>
</tr>
<tr>
<td>#Gibbs steps</td>
<td>5</td>
</tr>
<tr>
<td>Weight decay</td>
<td>0.001</td>
</tr>
</tbody>
</table>

4.2.1.3 Deep Boltzmann Machine training

The trained weights of the DBM determine the initial classification error of the DBM fine-tuning phase\(^1\). We tuned the DBM parameters in order to have a considerably lower initial classification error for the MLP in order to start the training from an already good set of weights. The most efficient parameters, considering computational time and classification performance, we found for this phase are resumed in Table VII. The momentum starts at 0.5 and linearly increases to 0.9 until the sixth epoch, after that it stays constant.

\(^1\)Actually, as explained before for MNIST, the initial test error is very high due to the fact that the softmax layer of the MLP has randomly initialized weights. However, after one epoch, the test error decreases significantly thanks to the DBM pretrained weights.
4.2.1.4 Deep Boltzmann Machine finetuning

The preliminary tests of DBM finetuning have been done using dropout because it is faster and almost always more effective than standard backpropagation. The activation functions of the hidden neurons have been set to be sigmoidal to have the same activation function of the DBM. The momentum starts at 0.5 and it is linearly increased to 0.99 until the 300th epoch. After having tuned all the parameters, we run the last of the preliminary experiments to test the performance we are able to reach on the not preprocessed GTSRB; the results of the model with the lowest validation error are reported in Table IX. Figure 26 shows the progress of the errors during backpropagation; as expected, the validation error is much lower than the test error. Indeed, in GTSRB the test set images are way worse than the training set one from which the images for the validation set have been picked.
TABLE IX: BEST MODEL ERRORS FOR STANDARD GTSRB

<table>
<thead>
<tr>
<th>Error</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Validation error</td>
<td>0.8%</td>
</tr>
<tr>
<td>Test error</td>
<td>9.51%</td>
</tr>
</tbody>
</table>

Figure 26: GTSRB - Test errors diagram for preliminary experiments
4.2.2 Experiments with preprocessed images

Once we tuned the parameters for each phase of training, we used them to analyze the effect of image preprocessing in terms of performance. Figures 27, 28, 29, 30, 31 and 32 show how the different type of preprocessed images are reconstructed and the classification errors of the best models found during DBM finetuning for each of these preprocessing methods are resumed in Table X. The reconstructions are very similar to the original images, confirming that the GRBM learns good weights independently from the preprocessing methods. Dealing with classification errors, the method that performed better on the validation set was pixels intensities rescaling, but the lowest test error has been reached using the resulting RGB images after adaptive histogram equalization. Since the test error reached with the latter is significantly lower than the test error reached with the former and that the two validation errors were almost the same, we decided to preprocess the dataset with adaptive histogram equalization for the final experiments.

The very high classification errors obtained with the images in L*a*b*, HSV and chromaticity spaces can be explained simply looking at the resulting images after preprocessing that are clearly worse than the ones in RGB spaces. On the other hand, it is interesting to observe that pixels intensities rescaling, that seems to make the best version of the images, performed worse than adaptive histogram equalization that, instead, seems to make worse images. However, this is not surprising: in machine learning what it is more understandable for a human is not necessarily the best also for neural networks.
Figure 27: Pixels intensities rescaling reconstructions

Figure 28: Histogram equalization reconstructions

Figure 29: RGB - Adaptive histogram equalization
Figure 30: L*a*b* - Adaptive histogram equalization

Figure 31: HSV space reconstructions

Figure 32: Chromaticity space reconstructions
TABLE X: ERRORS FOR EACH PREPROCESSING METHOD

<table>
<thead>
<tr>
<th>Model</th>
<th>Validation error</th>
<th>Test error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pixels intensities rescaling</td>
<td>0.54%</td>
<td>7%</td>
</tr>
<tr>
<td>Histogram equalization</td>
<td>0.73%</td>
<td>8.2%</td>
</tr>
<tr>
<td>RGB-Adaptive histogram equalization</td>
<td><strong>0.61%</strong></td>
<td><strong>6.08%</strong></td>
</tr>
<tr>
<td>L<em>a</em>b*-Adaptive histogram equalization</td>
<td>4.67%</td>
<td>11.2%</td>
</tr>
<tr>
<td>HSV space</td>
<td>4.04%</td>
<td>31.8%</td>
</tr>
<tr>
<td>Chromaticity space</td>
<td>5.28%</td>
<td>33.8%</td>
</tr>
</tbody>
</table>

4.2.3 Final experiments

The final experiments have been done in order to lower the test error as much as possible. To achieve it, we used the parameters we found during the tuning phase and preprocessed the images with adaptive histogram equalization. We also used the whole training set to have as much training examples as possible. Trying to tune the MLP’s parameters we found that there were no improvements on the test error rate, but we recalled that from the very preliminary experiments on GTSRB with simple MLPs, models with two layers composed of about 1000 neurons performed better than models with a larger number of them. Surprisingly, we found that training a DBM with less than 3000 neurons per hidden layer we were able to reach lower test errors, even if the reconstruction errors of the generative models were way larger than before, which is a prove of its unreliability.

Training a DBM of two layers of 1000 and 2000 neurons we were able to overcome the previous results of 6.08%, with a test error of 5.17%. Thereafter, we tried several experiments with different combinations of parameters. The learning parameters of the MLP that performed
TABLE XI: MLP FINAL PARAMETERS

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>#Epochs</td>
<td>1500</td>
</tr>
<tr>
<td>#Visible neurons</td>
<td>6000</td>
</tr>
<tr>
<td>#Hidden neurons</td>
<td>1000-2000-43</td>
</tr>
<tr>
<td>Activation function</td>
<td>Sigmoidal*</td>
</tr>
<tr>
<td>Batch size</td>
<td>100</td>
</tr>
<tr>
<td>Weights initialization</td>
<td>[-0.05;0.05]</td>
</tr>
<tr>
<td>Biases initialization</td>
<td>0</td>
</tr>
<tr>
<td>Learning rate</td>
<td>0.5</td>
</tr>
<tr>
<td>Momentum</td>
<td>0.5-0.75*</td>
</tr>
<tr>
<td>Inclusion probabilities</td>
<td>0.5-0.5-0.5*</td>
</tr>
<tr>
<td>#Mean field inference steps for data augmentation</td>
<td>1</td>
</tr>
</tbody>
</table>

the best are resumed in Table XI. The momentum starts from 0.5 and linearly increases to 0.75 until the 100th epoch. We found that dropout performed better than standard backpropagation as it can be seen in Figure 33 that shows the first 500 epochs of MLP finetuning with and without dropout. Surprisingly, we saw that dropping visible neurons with probability 0.5 worked better than the more common 0.2 probability. Table XII resumes the final results we had using the previously showed learning parameters for each of three methods. The test error obtained dropping the visible layer neurons with probability 0.5 is the best DBM test error we were able to find during this work and it overcomes the best results of generative models on this dataset. Indeed, this compares with the 4.32% which is the previous best test error of generative models on this dataset obtained with Latent Dirichlet Allocation (LDA)\(^1\).

\(^1\)This test error has been obtained with the generative model of LDA on the Histogram of Oriented Gradients (HOG) features available on the GTSRB website\(^2\).
**Figure 33: Dropout vs Standard backpropagation**

**TABLE XII: FINAL RESULTS**

<table>
<thead>
<tr>
<th>Method</th>
<th>Test error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard backpropagation</td>
<td>7.26%</td>
</tr>
<tr>
<td>Dropout (0.8 drop probability for visible neurons)</td>
<td>4.35%</td>
</tr>
<tr>
<td>Dropout (0.5 drop probability for visible neurons)</td>
<td>4.15%</td>
</tr>
</tbody>
</table>
### 4.2.4 Multi-Prediction Deep Boltzmann Machine experiments

The experiments with MP-DBM have been done after the DBM ones in order to compare them. The learning parameters of a MP-DBM are the same of a DBM with an additional parameter indicating the drop probability of visible neurons for the Multi-Prediction training. As stated before, the training of the MP-DBM requires only one phase of training and the test error can be monitored from the beginning of training\(^1\). This allowed us to tune the MP-DBM’s parameter (Table XIII, where the momentum starts from 0.5 and linearly increases to 0.75 until the 6\(^{th}\) epoch) having a look directly to the effect that they had on the test error.

The training of the MP-DBM gave us the very poor result of 44.7\% of test error. However, as shown in the NORB experiment\[^{[12]}\], MP-DBM finetuning is stated to be necessary to reach

---

\(^1\)To have a single training phase, the MP-DBM is required to have Gaussian visible neurons that the Multi-Prediction training is able to use avoiding the need of the preprocessing GRBM.
high performance on complex datasets; therefore, we finetuned the MP-DBM augmenting the data as before and using the same MLP parameters. At the end of MP-DBM finetuning we obtained a test error of 3.81%, that is the best test error on GTSRB that we were able to obtain in this work.

It is worth to point out that the test errors we obtained have been reached tuning the parameters to maximize the performance on the test set, despite the fact that this was not possible for the participants at the competition on GTSRB that were allowed to train their models only using a validation set. Indeed, the test set was provided only at the end of the competition to determine the best model. However, our purpose is to demonstrate the effectiveness of DBMs on GTSRB and the comparison with the models of the competition is purely a way to understand how DBMs perform compared to other models.

Figure 34 shows the progress of the test errors of both DBM and MP-DBM finetuning. This work confirmed that MP-DBMs can obtain slightly better performance than DBMs also on GTSRB dataset, even with a lower computational time.
Figure 34: DBM and MP-DBM test errors diagram
CHAPTER 5

CONCLUSION AND FUTURE DEVELOPMENTS

In this work we started from the research of Hinton and Salakhutdinov, that showed the excellent discriminative performance of DBMs on MNIST, with the purpose to evaluate their efficiency on a more complex dataset. We chose the GTSRB dataset, a collection of 43 classes of road signs photographed from real roads. The complexity of this dataset is motivated by the different light conditions affecting the images, the background, possible obstacles in front of the road signs or bad conditions of them.

To implement our experiments we used a Python machine learning framework called Pylearn2 integrating it with the whole DBM training algorithm, composed of greedy layer-wise pretraining, training and finetuning, that was missing before. Initially, we tried to replicate the results of Hinton and Salakhutdinov on MNIST to be sure of the correctness of our implementation; then, we began to analyze the GTSRB in order to understand how to deal with it. After having tuned the DBM parameters using the standard GTSRB images, we applied several preprocessing techniques to improve their quality in order to reach better discriminative performance. Finally, using the preprocessed images obtained with the preprocessing method that performed better, we made our final experiments with the purpose to reach the lowest test error.

We have been able to reach a 4.15% test error that allowed us to overcome the 4.32% published on the GTSRB website that, at the best of our knowledge, is the best published test error for generative models on this dataset. We also evaluated the performance of the MP-
DBM model, a variant of DBM that reached similar results on MNIST and NORB datasets, to compare their results on GTSRB. Using MP-DBMs we have been able to reach a 3.81% test error that is the best test error on GTSRB published in this work confirming that MP-DBM can achieve slightly better results than DBM.

Our work showed that DBMs and MP-DBMs can reach high discriminative performance even on more complex dataset than MNIST and NORB. It also showed that they have been able to overcome all generative models results on GTSRB suggesting that they are one of the best generative models for discriminative tasks.

5.1 Future developments - The German Traffic Sign Detection Benchmark

The German Traffic Sign Detection Benchmark (GTSDB) dataset is a collection of images of real roads, such as the pictures in Figure 35, introduced on the IJCNN in 2013. This dataset has to be used to train models able to detect road signs in the environment and, if requested, to classify them. To do this, the GTSDB images can be divided in a certain number of subimages that will be used as the input of model that has been trained on GTSRB. This model has the purpose to detect if the subimage contains a road sign and, if the answer is yes, to classify it. This is a possible application of DBMs that we had not explored in this work, but that can be done in the future.

Together with this task, it would be worth also to explore the possibilities of DBMs in real-time road signs classification. This is not a trivial tasks due to the amount of time that a single image requires to be classified with DBMs. Indeed, our model requires about 5 seconds to classify an image. This is explainable by the fact that, once the DBM has been finetuned, it is
Figure 35: Examples of GTSDB images
not possible to only perform bottom-up passes to infer the class of the road sign. In fact, before this phase, the image has to be preprocessed by the GRBM that produces the input of the DBM and this input has to be augmented by the DBM. In particular, the data augmentation is the most time demanding task that has the drawback to, apparently, prevent the possibility of using DBMs (and MP-DBMs) for real-time classification.

Thanks to the high discriminative performance reached with our models, this work can be used as the starting point to study the possibility of using DBMs for classification of GTSDB images and even to test, and improve, their efficiency in real-time road signs classification.
CITED LITERATURE


# VITA

## Carlo D’Eramo

<table>
<thead>
<tr>
<th>Education</th>
<th>B.S., Engineering of Computing Systems</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Politecnico di Milano</td>
</tr>
<tr>
<td></td>
<td>September 2011</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>M.S., Engineering of Computing Systems</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Politecnico di Milano</td>
</tr>
<tr>
<td></td>
<td>2011-2014</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>M.S., Computer Science</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>University of Illinois at Chicago, Chicago, IL</td>
</tr>
<tr>
<td></td>
<td>2012-2015</td>
</tr>
</tbody>
</table>