

Departement of Informatics
University of Fribourg (Switzerland)

Deep Learning Feature Extraction for Image Processing

Thesis

presented to the Faculty of Science of the University of Fribourg (Switzerland)
in consideration for the award of the academic grade of
Doctor of Philosophy in Computer Science

by

Baptiste Wicht

from

Le Mouret, Fribourg (Switzerland)

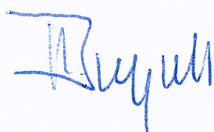
Thesis N° 2048
UniPrint, Fribourg
2017

Accepted by the Faculty of Science of the University of Fribourg (Switzerland)
upon the recommendation of:

- Prof. Edy Portmann, University of Fribourg (Jury President)
- Prof. Rolf Ingold, University of Fribourg (Director)
- Prof. Jean Hennebert, University of Fribourg (Co-Director)
- Prof. Volkmar Frinken, University of Kyushu, Japan (Expert)
- Dr. Andreas Fischer, University of Fribourg (Expert)
- Dr. Robert Van Kommer, EPFL, Lausanne (Expert)

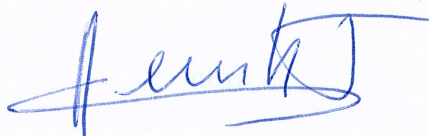
Fribourg, October 6th, 2017

Thesis Director



Prof. Rolf Ingold

Thesis Co-Director



Prof. Jean Hennebert

Faculty Dean



Prof. Christian Bochet

Version abrégée

Dans le cadre de cette thèse, nous proposons d'utiliser des méthodologies permettant d'extraire automatiquement des caractéristiques importantes depuis des images. Nous nous sommes intéressés tout spécialement à l'évaluation de ces caractéristiques comparées à des caractéristiques définies à la main. Plus précisément, nous nous sommes intéressés à l'entraînement non-supervisé de modèles Restricted Boltzmann Machine (RBM) et Convolutional Restricted Boltzmann Machine (CRBM). Ces modèles ont relancés l'engouement de cette décade pour les technologies dites de Deep Learning. En effet, ces dernières années les approches basées sur les auto-encodeurs et tout spécialement les auto-encodeurs convolutionnels ont été de plus en plus utilisées. Pour ces raisons, un objectif de cette thèse est de comparer l'approche CRBM avec l'approche auto-encodeur convolutionnel.

Le cadre de ce travail est défini par plusieurs tâches d'intelligence artificielle. La première, la reconnaissance de nombres écrits à la main, est analysée pour voir comment le pré-entraînement non-supervisé, une technique introduite avec les Deep Belief Networks (DBNs), d'un modèle peut améliorer l'entraînement de réseaux de neurones. Le second, la détection et la reconnaissance de Sudoku dans des images, consiste à évaluer les performances des modèles DBN et Convolutional Deep Belief Network (CDBN) pour la classification d'images de basse qualité. Finalement, les caractéristiques sont apprises de façon complètement non-supervisée pour une tâche de repérage de mot-clé depuis des images et sont comparées avec des caractéristiques bien connues, définies à la main. De plus, cette thèse est aussi orientée autour d'un axe de développement logiciel. En effet, une librairie de réseaux de neurones a été développée durant la thèse pour explorer les différentes optimisations qui sont possibles pour entraîner les modèles aussi vite que possible.

Cette thèse est séparée en plusieurs chapitres, ainsi:

Chapitre 1 Introduction Ce chapitre présente le contexte de la thèse. Une série de questions de recherche est proposée pour définir plus en détail le contexte de la recherche. Il propose aussi une description détaillée du contenu de ce rapport.

Chapitre 2 Fundamentals Dans ce chapitre, les différentes techniques utilisées pour cette thèse sont détaillées. Cela comprend l'intelligence artificielle en général et les réseaux de neurones, en passant par les techniques plus spécifiques telles que le modèle RBM et son entraînement.

Chapitre 3 Semi-Supervised Training Ce chapitre explore les avantages pos-

sibles du pré-entraînement d'un réseau de neurones en utilisant une technique non-supervisée. Les résultats montrent que cela permet de fortement régulariser l'entraînement du réseau. L'état de ces techniques en vue des différentes améliorations modernes relatives à l'entraînement des réseaux est aussi discuté.

Chapitre 4 Framework Une librairie complète pour l'entraînement de réseaux de neurones a été développée pour cette thèse. Ce chapitre présente toutes les fonctionnalités incluses dans la librairie ainsi que les nombreuses optimisations qui ont été réalisées afin de rendre l'entraînement de réseau de neurones aussi rapide que possible. Les choix d'implémentation relatifs aux différents modèles RBM sont aussi discutés.

Chapitre 5 Sudoku Recognition L'entraînement semi-supervisé est utilisé pour entraîner un système de classification pour des nombres aussi bien générés par ordinateur qu'écrits à la main, depuis des images de Sudoku. Une base de données de ces images a été collectée avec des caméras de téléphones portables dans des journaux suisses. Il est montré que le système final est capable de reconnaître la plupart des nombres, en capturant des caractéristiques importantes des images, autant pour les nombres cursifs que pour les nombres générés par ordinateur.

Chapitre 6 Keyword Spotting L'apprentissage entièrement non-supervisé est analysé dans ce chapitre, dans le contexte du repérage de mot-clés dans des documents historiques et modernes. Un système complet est présenté avec deux systèmes de classification différents. Les caractéristiques apprises sont comparées avec plusieurs caractéristiques de référence et les résultats montrent leur supériorité dans presque toutes les conditions testées.

Chapitre 7 Auto-encoders Dans ce chapitre, nous avons comparé les auto-encodeurs standards, basés sur des réseaux de neurones, avec les modèles RBM et CRBM. La même tâche de repérage de mot-clé, présentée dans le Chapitre 6, est utilisée pour l'évaluation des différentes techniques. Il est montré que les deux familles ont des performances relativement similaires. Nous observons que les auto-encodeurs standards sont généralement plus simples à entraîner et à configurer. Cela peut expliquer leur domination dans la communauté de recherche.

Chapitre 8 Conclusion Le chapitre final de cette thèse dresse les conclusions sur les différentes expériences et tente de répondre aux questions qui ont été définies dans l'Introduction.

Abstract

In this thesis, we propose to use methodologies that automatically learn how to extract relevant features from images. We are especially interested in evaluating how these features compare against handcrafted features. More precisely, we are interested in the unsupervised training that is used for the Restricted Boltzmann Machine (RBM) and Convolutional Restricted Boltzmann Machine (CRBM) models. These models relaunched the Deep Learning interest of the last decade. During the time of this thesis, the auto-encoders approach, especially Convolutional Auto-Encoders (CAE) have been used more and more. Therefore, one objective of this thesis is also to compare the CRBM approach with the CAE approach.

The scope of this work is defined by several machine learning tasks. The first one, handwritten digit recognition, is analysed to see how much the unsupervised pretraining technique introduced with the Deep Belief Network (DBN) model improves the training of neural networks. The second, detection and recognition of Sudoku in images, is evaluating the efficiency of DBN and Convolutional Deep Belief Network (CDBN) models for classification of images of poor quality. Finally, features are learned fully unsupervised from images for a keyword spotting task and are compared against well-known handcrafted features. Moreover, the thesis was also oriented around a software engineering axis. Indeed, a complete machine learning framework was developed during this thesis to explore possible optimizations and possible algorithms in order to train the tested models as fast as possible.

This thesis is split into several chapters, as follows:

Chapter 1 Introduction This introductory chapter presents the context of the thesis. Several research questions are also proposed to define further the context of the research and a detailed outline of the thesis is provided.

Chapter 2 Fundamentals In this chapter, the background of the different techniques used in this research is laid out, from machine learning to the RBM model and its unsupervised training with the Contrastive Divergence (CD) algorithm.

Chapter 3 Semi-Supervised Training This chapter explores the advantages of pretraining a neural network using unsupervised techniques in order to produce a robust initialization of the weights. Our work shows that unsupervised pretraining acts as a strong regularizer. The state of unsupervised pretraining in view of the recent improvements in training is also discussed.

Chapter 4 Framework We developed a machine learning framework for the purpose of this research. The different features that were implemented as well as the many performance optimizations that were developed are presented here. The implementation choices regarding the different RBM models are also discussed.

Chapter 5 Sudoku Recognition Unsupervised pretraining is used to train a recognizer for mixed inputs including computer-generated and handwritten digits extracted from Sudoku images. A database of images has been collected using smartphone cameras in Swiss newspapers. It is shown that the final system is able to efficiently recognize most of the detected digits, capturing relevant features for both printed and handwritten digit recognition.

Chapter 6 Keyword Spotting Fully unsupervised learning with convolutional models is explored in this chapter, in the context of handwritten keyword spotting, on historical and modern documents. A complete system is designed, with two different classifiers. The learned features are compared against several reference handcrafted feature sets and are outperforming them under almost all conditions.

Chapter 7 Auto-encoders In this chapter, we compared regular auto-encoders, based on Artificial Neural Networks (ANNs) against the RBM and CRBM models. The same keyword spotting task as in Chapter 6 is used for the evaluation. It is shown that both families are exhibiting similar performance. We observe that auto-encoders are generally more simple to train, which may explain their current predominance of use in the research community.

Chapter 8 Conclusion The final chapter draws the conclusions on the different experiments and addresses the questions that were stated in the Introduction.

Acknowledgments

Although this thesis results of the compilation of my own efforts, I would like to acknowledge and express my gratitude to the following people for their valuable time and assistance, without whom the completion of this project would not have been possible:

- My advisor, Prof. Dr. Jean Hennebert, for giving me the opportunity to work on this thesis, for his support during these years and especially for all the Belgian beer and fun provided to the team!
- Prof. Dr. Rolf Ingold for acting as dean of this thesis
- Dr. Volkmar Frinken and Dr. Robert Van Kommen for acting as experts for this research
- My girlfriend, Bao Li Zheng for supporting me during this long thesis and making me believe I could finish it.
- Dr. Andreas Fischer for all the help and quality advice provided during the course of this thesis.
- Dr. Frédéric Bapst for reviewing this thesis and providing quality comments for its improvements
- All the people of room C10.08, for all the fun and terrible discussions we had during our lunches and coffees together.
- My fellow labmates for the stimulating discussions.
- My family for believing in me and supporting me.
- Bjarne Stroustrup for creating C++, the perfect programming language allowing me to develop the framework necessary for this research.
- Amon Amarth, Arch Enemy, Kalmah, Sabaton and Wintersun for providing me with the necessary musical environment during my research.

Contents

1	Introduction	1
1.1	General context: Feature extraction	1
1.2	Specific context: Image processing	3
1.3	Research questions	4
1.4	Outline of the thesis	5
I	Theory	9
2	Fundamentals	11
2.1	Introduction to Machine Learning	12
2.2	Neural Network	14
2.3	Deep Learning	19
2.4	Restricted Boltzman Machine	20
2.5	Deep Belief Network	41
2.6	Convolutional Restricted Boltzmann Machine	43
2.7	Convolutional Deep Belief Network	48
2.8	Variants of Restricted Boltzmann Machine	48
3	Semi-Supervised Training	57
3.1	Introduction	57
3.2	Advantages of pretraining	59
3.3	Advances in Neural Network training	60
3.4	Summary	62
4	Framework	67
4.1	Deep Learning Library	67

4.2	Models	68
4.3	Visualization	71
4.4	Performance	73
4.5	Preprocessor	78
4.6	Evaluation	79
II	Applications	83
5	Sudoku Recognition	85
5.1	Introduction	85
5.2	Data set	86
5.3	State of the art	88
5.4	Sudoku Digit Detection	90
5.5	Digit Classifiers	92
5.6	Results	102
5.7	Performance	104
5.8	Summary and potential extensions	105
6	Keyword Spotting	109
6.1	Introduction	110
6.2	Keyword Spotting	113
6.3	Data sets	116
6.4	Reference features	118
6.5	Keyword Spotting System	119
6.6	Results	125
6.7	Grayscale images	137
6.8	Efficiency	140
6.9	Summary	142
7	Auto-encoders	149
7.1	Introduction	149
7.2	Experimental Evaluation	151
7.3	Dense Auto-Encoders	152
7.4	Convolutional Auto-Encoders	155

7.5	Hybrid Auto-Encoders	160
7.6	Denoising Auto-Encoders	162
7.7	Results	165
7.8	Summary	166
III	Conclusion	171
8	Conclusion	173
8.1	Synthesis	173
8.2	Perspectives	176
IV	Appendices	179
A	Detailed Performance Analysis	181
A.1	Introduction	181
A.2	Configuration	182
A.3	Valid Convolution	182
A.4	Full Convolution	184
A.5	GPU Performance	184
A.6	References for Appendix A	192
B	Framework Evaluation	193
B.1	Introduction	193
B.2	Empirical Evaluation	195
B.3	General Evaluation	210
B.4	Conclusion	213
B.5	References for Appendix B	214
	References	217
	Index	233
	Glossary	235
	Acronyms	237

List of Figures	241
List of Tables	249
List of Algorithms	255
Resume	257
List of publications	259

Symbols and notation

Below we summarize the notation used throughout this thesis. Vectors are denoted as boldface lowercase letter, such as \mathbf{a} . Matrices are denoted as boldface uppercase letters, such as \mathbf{A} . Constants are uppercase letters, such as N . Indices inside ranges are lowercase letter, such as i . Unless indicated otherwise, vectors and matrices are considered as being row-major.

Machine Learning

\mathbf{o}	Raw input of a system, from which features \mathbf{x} are extracted
\mathbf{x}	Input of a system
\mathbf{y}	Expected output of a system
\hat{y}	Obtained output of a system
ϵ	Learning Rate
α	Momentum
λ	Weight-cost for weight decay
∇	Gradient
θ	Parameters of a model
$J(\theta)$	Cost function
W	Weights of a neural model
b	Biases of the visible units of a neural model
c	Biases of the hidden units of a neural model

Energy Based Models

$p(x)$	Probability density function
$p(x \mid y)$	Conditional probability of x given y
$E(\mathbf{v}, \mathbf{h})$	Energy of a joint configuration
$F(\mathbf{v})$	Free Energy of a visible pattern

$\mathbb{E}_{\mathbf{h}}$	Expectation over \mathbf{h}
Z	Partition function
p	Activation probability of a unit
s	Sample state of a unit
\mathbf{v}	Vector of visible unit probabilities
\mathbf{v}'	Vector of visible unit states
\mathbf{h}	Vector of hidden unit probabilities
\mathbf{h}'	Vector of hidden unit states

Mathematical symbols

\triangleq	defined as
$ x $	Absolute value
$\sigma(x)$	Logistic sigmoid function
$\exp(x)$	Exponential function e^x
$N(\mu, \sigma^2)$	Normal distribution of mean μ and variance σ^2
$Unif(a, b)$	Uniform distribution between a and b
$O(\cdot)$	Big-O complexity

Linear algebra

\mathbf{A}^T	Transpose of a matrix
$Tr(\mathbf{A})$	Trace of a matrix
$\mathbf{a} \cdot \mathbf{b}$	Dot product of two vectors
$\mathbf{a} \odot \mathbf{b}$	Hadamard product between two vectors
$\mathbf{a} \otimes \mathbf{b}$	Outer product between two vectors
$\tilde{\mathbf{A}}$	Matrix \mathbf{A} flipped horizontally and vertically
$\mathbf{A} * \mathbf{B}$	Matrix multiplication of the matrix \mathbf{A} and the matrix \mathbf{B}
$\mathbf{I} \bullet_v \mathbf{K}$	Valid convolution of the matrix \mathbf{I} by the matrix \mathbf{K}
$\mathbf{I} \bullet_f \mathbf{K}$	Full convolution of the matrix \mathbf{I} by the matrix \mathbf{K}
$\mathcal{F}(\mathbf{A})$	Fast Fourier Transform of the matrix \mathbf{A}
$\mathcal{F}^{-1}(\mathbf{A})$	Inverse Fast Fourier Transform of the matrix \mathbf{A}

Chapter 1

Introduction

*If you think it's simple, then you
have misunderstood the problem*

Bjarne Stroustrup

Contents

1.1	General context: Feature extraction	1
1.2	Specific context: Image processing	3
1.3	Research questions	4
1.4	Outline of the thesis	5

Machine Learning can be described as the science of making a computer take a decision without telling it how exactly to perform this task. For this, a model is presented with a large quantity of data and the associated expected responses. From this data, the model is able to learn a mathematical model of how to bind the response, or label, to the input, in such a way that it can predict the correct response, generalizing for inputs that have not yet been seen. This form of learning using labels is known as supervised learning.

1.1 General context: Feature extraction

Nowadays, Machine Learning is used to analyze more and more data and the available data is becoming more and more complex. In the last decade, the advent of Deep Learning helped in creating more efficient learning models. Many Machine Learning tasks are targeting *classification* problems. Such systems work in a way similar to what is shown on Figure 1.1. First, features are extracted from the input data. This can be seen as creating a new representation of the data specifically for the current task. A classification system is then learned on top of these features to achieve the task. Once trained, the system should now be able to be used on

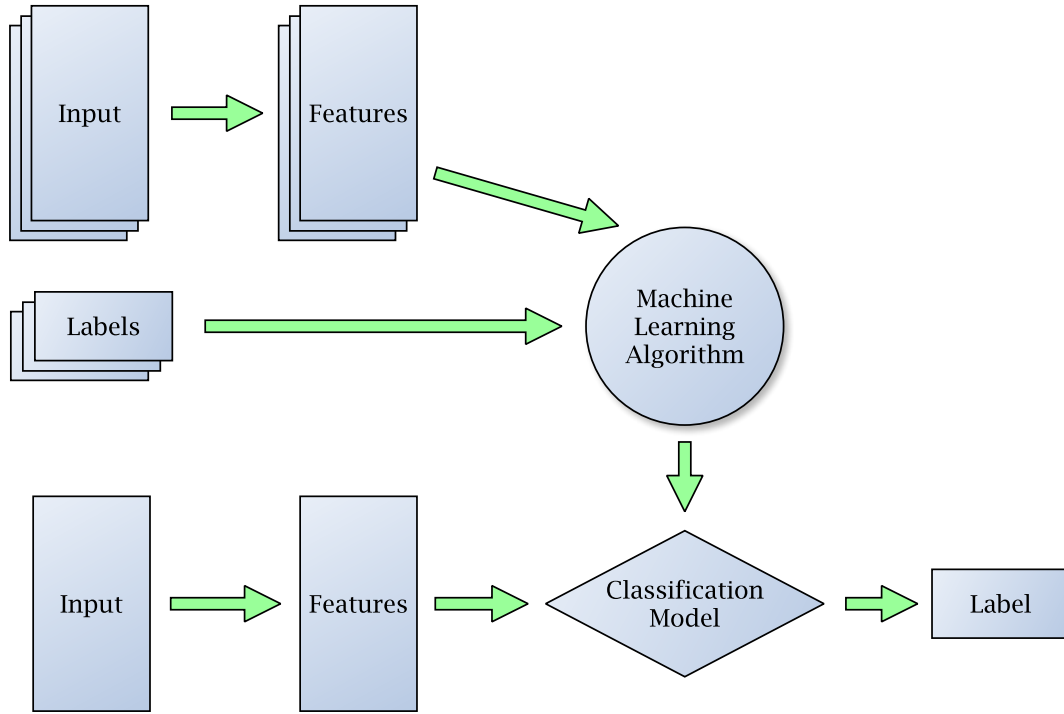


Figure 1.1: A typical supervised Machine Learning model, for classification tasks. The model is trained using some algorithm and some training data. It can then be used to predict a label from any input.

data that have not yet been seen during the training phase and accurately predict its response, in this case, the class label. Often, and especially until recent years, the features extracted from the input were *handcrafted* features. This qualification means that such features are designed especially for the input data and the task at hand. They are generally tied not only to the type of data, for instance images of handwritten words, but to a specific subset such as images of English handwritten words written with ink on parchment. Most of these features are generally not robust to change.

Another approach to extract features from the data is to learn a feature extractor using Machine Learning. Instead of building a system to classify some images, a learning system is built to extract features from the input. In the case of images, this means that the network is learning higher-level features directly from the input pixels. We believe that this approach is superior to using handcrafted features, for several reasons. By training a model on each data set, the trained model can be adapted to many types of inputs whereas handcrafted features may require hand-tuning for each data set. Moreover, this approach should not need an expert knowledge of the images being analyzed.

The main idea behind this thesis is the in-depth analysis of techniques for feature extraction from data. As shown in the following section, a special focus is given to feature extraction from images and more specifically from handwriting. The class

of feature learning methods analysed in this thesis is also scoped by the previous work of Geoffrey Hinton, the so-called "Hinton Approach".

1.2 Specific context: Image processing

A large part of the work of Machine Learning nowadays is in relation with images. There are very large collections of images available freely on the Internet. Even labeled image data sets are somewhat abundant compared to other fields of research. On the other hand, images have a tendency of being complex. Although they are generally easy to understand by the human, they may present a lot of difficulty to a computer model. Moreover, they also can be of high dimensionality, making them harder to process in a reasonable amount of time. Finally, images can be very different depending on the conditions in which they were taken. The quality of the capture makes large differences in how the images can be analyzed.

Because of their complexity and dimensionality, most of the ongoing research is using some methods to find spatial dependencies inside the image to be able to learn features that can be shared across the complete image rather than be specific to pixels. For this, the Convolutional Neural Network (CNN) and Long Short Term Memory (LSTM) models have proven very successful. In this research, we will focus on the use of convolutional models to solve some problems of image analysis. By the use of convolution, the network is learning feature detectors shared among all locations of an image. For instance, one of its feature detectors could detect a red dot anywhere in the image, rather than detecting a red dot in the upper left corner. Convolution also has the advantage that the number of weights that need to be learned is much smaller than the number of weights for a standard neural network. When learning deep representations of images, the goal is that each layer learns a more specialized feature set on top of the previous layer features. For instance, if we were trying to detect images of cars in a large data set, the first layer could learn to detect oriented edges, the second layer to detect car parts and the last one to combine parts to form different types of cars.

Deep Learning, a seemingly new trend in Machine Learning, has proven very successful in solving difficult problems such as classifying very large images or data with many classes. Research in Deep Learning was restarted when Hinton and Salakhutdinov introduced a new approach, able to pretrain a neural network in an unsupervised manner. This particular approach, the "Hinton approach" consists in using Restricted Boltzmann Machine (RBM), Convolutional Restricted Boltzmann Machine (CRBM) and Deep Belief Network (DBN) either to improve classification results via pretraining or to extract features from images in an unsupervised way. While Deep Learning evolved in several other directions starting from this breakthrough, in this thesis, we are particularly interested on the original approach of Hinton about Deep Learning.

In the broader concept of image processing, we also focused especially on handwritten inputs for which still challenging tasks are in front of the research community.

More specifically, contributions are done in this thesis in the context of two different experiments: mixed printed and handwritten digit recognition for Sudoku Recognition and Handwritten Keyword Spotting. This task allowed us to observe the effect of pretraining a neural network with RBM and CRBM. Pragmatically, profiting from an already existing Sudoku Recognition data set, we have developed a system able to detect and recognize a Sudoku puzzle in an image taken from a phone camera in a newspaper. Chapter 5 is partially related to this experiment. In a second time, we have focused on the task of Handwritten Keyword Spotting, consisting in finding all the occurrences of a given word in a set of images from historical documents, without using transcription. This problem has the advantage that there are several baseline handcrafted feature sets for it, making it a good basis for comparing against automatically generated feature sets. Moreover, the available baseline systems are generally using two very different types of classifier, making it perfectly suited to test how features are tied to a particular Machine Learning model. The results of using unsupervised learning for feature extraction on the task of Handwritten Keyword Spotting are presented in Chapter 6. Then, taking advantage of this system as a way to compare different feature extractors, the RBM models are compared against regular auto-encoders. The results of the comparison between these systems are presented in Chapter 7.

1.3 Research questions

In this thesis, we are using neural networks to extract features from images. Specifically, we are using the "Hinton" Deep Learning approach with the RBM and CRBM models to learn a deep representation of the input. It was already shown that these techniques can be used to improve a classification model by providing a good initialization of its weights. They are able to extract features from images that can then be used for classification by other advanced classifier such as Support Vector Machine (SVM). Our constitutive hypothesis is that such techniques are able to learn good feature representation from images and that such features can be used even with basic classifiers, such as template matching techniques. In this thesis, we focus on the advantages of using automatic feature learning rather than handcrafted features.

Under the assumption that the RBM and CRBM models can generate good features from images, we propose the following scientific questions that this thesis will try to address:

1. What are the advantages of unsupervised pretraining in the context of training neural networks for classification ?
2. Does pretraining work in the same way for standard models and convolutional models ?
3. What are the advantages of automatic feature extraction compared to handcrafted features ?

4. How much tuning of the model and training is necessary to generate features to be used by basic classifiers ?
5. Can the same features easily be used by different classifiers ?
6. How does the RBM and CRBM approaches compare to other alternatives for feature extraction ?

Further to these more fundamental questions, we also try to address specific questions that are related to the conducted experiments:

1. In the context of Sudoku Recognition, can a single network efficiently learn from two different types of inputs (computer-generated and handwritten) ?
2. In the context of Handwritten Keyword Spotting, are features learned on grayscale images superior to features learned on binary images ?
3. How does the machine learning framework developed for this research compare to other popular alternatives in terms of accuracy and runtime performance ?

1.4 Outline of the thesis

This thesis is organized as follows.

- **Chapter 1 Introduction**

In this first Chapter, the general and the specific contexts of this thesis are introduced, respectively feature extraction and image processing. After this, the constitutive hypothesis and the scientific and specific questions that this thesis addresses are presented. Finally, the different chapters of this report are outlined.

Part I Theory

- **Chapter 2 Fundamentals**

In this Chapter, the fundamental background on which this thesis is based is described. This starts with an introduction to Machine Learning and especially to Deep Learning. It then goes on to introduce the RBM model which is the main building block of this research. How to train this model is covered in detail. From there, building upon the RBM, higher level networks are introduced by stacking. The convolutional versions of the standard models are also considered as well as some of the most popular variants of the RBM.

- **Chapter 3 Semi-Supervised Training**

The third Chapter focuses on the unsupervised pretraining of feed-forward neural networks in order to improve supervised training in a second phase, in the context of classification problems. It presents the general concept of pretraining and its advantages. It then provides a list of the most important improvements that have been made to neural network training since the advent of pretraining. Finally, a summary of this form of pretraining is presented.

- **Chapter 4 Framework**

Chapter 4 presents the Deep Learning library that was developed during the course of this research. This library was used to perform all the experiments presented in this thesis. The features of the framework are presented in detail as well as the most important points of implementation such as performance considerations and memory consumption.

Part II Applications

- **Chapter 5 Sudoku Recognition**

The fifth Chapter summarizes the experiments that were performed on detection and recognition of mixed printed and handwritten digits collected in the context of Sudoku puzzle images taken with a phone camera. The data set that was collected and used for this research is presented, as well as the state of the art for this problem. The different steps of the detection of the puzzle are also outlined. Then, the different classifiers that are used for this task are presented. Finally, the results and runtime performance of the proposed systems are analyzed and the overall results of the experiments are summarized.

- **Chapter 6 Keyword Spotting**

This Chapter investigates the use of fully unsupervised feature learning with convolutional models for a keyword spotting task on handwritten documents. First, the state of the art for this task is analyzed. Then, the features learned with this model are compared against three reference feature sets applied on three different data sets. Two different classifiers are used and compared, a simple classifier and a complex learning-based classifier. The results obtained under these different configurations are discussed in detail, as well as the efficiency of the different models. Finally, conclusions are drawn regarding the use of these features compared to standard features.

- **Chapter 7 Auto-encoders**

In this final evaluation, the features generated from RBM and CRBM are compared to the features generated by regular auto-encoders on the keyword spotting task. Several variants of models are investigated, from dense auto-encoders to convolutional auto-encoders, as well as deep and stacked

auto-encoders and denoising auto-encoders. Finally, the advantages and disadvantages of the RBM and CRBM approaches are summarized compared to these alternatives.

Part III Conclusion

- **Chapter 8 Conclusion**

This Chapter concludes the thesis. The scientific questions addressed in this Introduction are recapitulated and detailed answers are provided for each of them. Directions for future work are also presented.

Part I

Theory

Chapter 2

Fundamentals

*In deep learning, the algorithms we use now are versions of
the algorithms we were developing in the 1980s and the 1990s*

Geoffrey Hinton

Contents

2.1	Introduction to Machine Learning	12
2.1.1	Mathematical formalism	13
2.2	Neural Network	14
2.3	Deep Learning	19
2.4	Restricted Boltzman Machine	20
2.4.1	Binary-Binary Restricted Boltzmann Machine	23
2.4.2	Training	24
2.4.3	Contrastive Divergence	25
2.4.4	Monitoring the learning progress	32
2.4.5	Persistent Contrastive Divergence	35
2.4.6	Other types of units	36
2.5	Deep Belief Network	41
2.6	Convolutional Restricted Boltzmann Machine	43
2.6.1	Convolutional Sparsity regularization	45
2.6.2	Probabilistic Max Pooling	46
2.7	Convolutional Deep Belief Network	48
2.8	Variants of Restricted Boltzmann Machine	48
2.8.1	Mean-Covariance RBM	49
2.8.2	Discriminative RBM	49
2.8.3	Temporal RBM	50

2.8.4	Spike and Slab RBM	50
2.8.5	Deep Boltzmann Machine	50

2.1 Introduction to Machine Learning

This thesis focuses on extracting features from images directly with Machine Learning rather than relying on handcrafted feature extractors. Machine Learning is a science aiming at getting machines to learn solutions to specific problems without being explicitly programmed into doing so (Murphy, 2012). Many Machine Learning solutions have been derived from our knowledge of the human brain.

Machine Learning models are used to solve two main tasks: *classification* and *regression* (Bishop, 2006). *Classification* is the task of assigning a label to an input while *regression* generates a continuous output. A classification problem would be, for example, to detect images of cats in a large set of images while a regression problem would be to predict the price of a house given its characteristics.

To solve these two problems, three approaches are particularly considered:

- *Supervised learning*: The model is learned from the input and the expected output data. This is the most common form of learning. Indeed, most training algorithms are computing an initial output, comparing it with the expected output and adapting the system to be closer to the expected data.
- *Unsupervised learning*: The model is learned only from the input data. This approach is particularly useful in practice since unlabeled data is abundant while labeled data is more scarce and requires a lot of effort to collect. However, due to its unsupervised characteristic, this technique cannot be used directly for classification. It is rather used to automatically discover structures in the input space such as the presence of clusters.
- *Semi-Supervised learning* (or *Semi-Unsupervised learning*): Both kinds of data are used to train the model. The model is first pretrained using unsupervised data and then improved with supervised data. There is generally more unsupervised data available and this approach makes use of this advantage. It is also possible to train a model using both types of data at once, but this is more complex and rarely used.

In this thesis, we are particularly interested in the use of unlabeled data, either for fully unsupervised learning and feature extraction or for later supervised training, for classification problems.

2.1.1 Mathematical formalism

The objective of supervised learning can be generalized as learning a good approximation of an unknown function $\mathbf{y} = f(\mathbf{x})$, where \mathbf{x} denotes an input vector and \mathbf{y} the expected output. The learning process assumes the availability of representative training examples $(\mathbf{x}_n, \mathbf{y}_n)$. The goal is to learn a mapping function $\hat{\mathbf{y}}$, through the learning process, that approximates $f(\mathbf{x})$:

$$\hat{\mathbf{y}} = \hat{f}(\mathbf{x}) \quad (2.1)$$

The learning objective is to make $\hat{\mathbf{y}}$ as close as possible to the actual \mathbf{y} , a property known as *convergence*. The learned mapping function uses some parameters, or weights. The set of parameters is generally denoted as θ , thus the mapping function is sometimes defined as $\hat{f}_\theta(\mathbf{x})$. The learning process will progressively adapt these weights in order to make the $\hat{\mathbf{y}}$ vector converge to \mathbf{y} . In order to do this, the learning process will use a *cost function* $J(\theta)$. The cost function is measuring how good the mapping function is with the given parameters. It is defined differently for each model and each training procedure. However, the objective will always be to minimize $J(\theta)$. Generally, learning algorithms are looking for the minimum of the cost function by iteratively moving the parameters in the opposite direction of the gradient of the cost function with:

$$\theta_i = \theta_i - \epsilon \frac{\partial J(\theta)}{\partial \theta_i} \quad (2.2)$$

where ϵ is the learning rate, expressing the size of the step that is done in the direction of the gradient descent. This is then repeated, until convergence is reached or more generally some set goal is reached, such as going below some classification error on an independent validation set. This approach is called gradient descent and is the most popular training algorithm for neural networks. From this basic idea, there exist many variations of training algorithms (Ngiam et al., 2011).

In classical machine learning, the input \mathbf{x} is obtained through a preliminary step, called *feature extraction*, aiming at converting the *raw* input \mathbf{o} into a better suited representation \mathbf{x} :

$$\mathbf{x} = g(\mathbf{o}) \quad (2.3)$$

Recently and especially since the advent of Deep Learning (See Section 2.3), the full model is generally learned directly from the raw input, such as pixels from images, and the network itself is learning new representations of the data. Therefore, the input data is directly the raw input data:

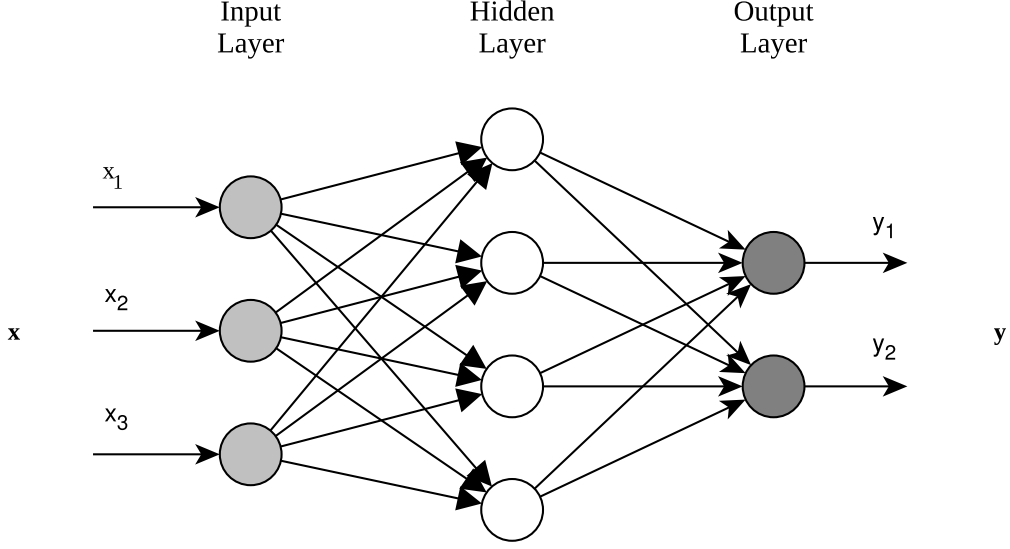


Figure 2.1: An exemplary feed-forward neural network with three layers. The input layer will receive the input data, a vector \mathbf{x} of D elements, in this case $D = 3$. The hidden layer will compute a new representation of the input. Finally, the output layer will compute the answer $\tilde{\mathbf{y}}$ of the network from the hidden representation. Assuming an adequate learning process with convergence, the $\tilde{\mathbf{y}}$ may represent the probabilities of observing 2 objects in the exemplary image. Each edge between two neurons is associated to a weight.

$$\mathbf{x} = \mathbf{o} \quad (2.4)$$

This is one of the advantages of Deep Learning, for which feature extraction becomes less necessary. Nevertheless, it has been shown that combining standard feature extractors and features extracted with Deep Learning can improve the performance of the overall system at the cost of an extra complexity (Lan et al., 2016; Sargano, Angelov, and Habib, 2017; Majtner, Yildirim-Yayilgan, and Hardeberg, 2016).

2.2 Neural Network

An Artificial Neural Network (ANN), or neural network, is a Machine Learning model inspired by cells in the human brain. We may refer to (Murphy, 2012) as a general introduction to ANNs. Such networks are used to approximate a function depending on a very large number of inputs. The concept of neural network is central to this thesis and to Machine Learning in general.

A neural network is typically designed in layers and is characterized by three main features:

1. Its architecture is defined by its number of layers, the number of neurons in each layer and the connections between the neurons. In the most simple case of feed-forward neural networks, there are only connections between the neurons of one layer and the neurons of the next layer. The first layer is called the input layer, the last layer is the output layer and the intermediate layers are called hidden layers. An example of neural network with three layers is shown in Figure 2.1. Each neuron has an associated value and this value is either given by the data (for the input layer) or computed from the set of its inputs, given by the previous layer. Before being passed to the next layer, the output of a layer is generally passed through an activation.
2. The activation function defines the mathematical function that is used to compute the state of a neuron based on the values of its inputs. To avoid the output being a linear representation of the inputs, the activation function is generally a non-linear function such as the hyperbolic tangent or the logistic sigmoid. The connections between the neurons, sometimes called synapses, have an associated weight that is used so that each neuron can weight individually each of its inputs. These weights are representing the "intelligence" of the system. The architecture of the network plus its trained weights is considered as the trained model.
3. The last characteristic of a neural network is the learning rule that is used to update its weights so that the function is approximated as well as possible. It is not considered as a part of the model since it is not used anymore after training.

The original neural network model, called the perceptron, was designed in 1958 (Rosenblatt, 1958). This model is the simplest form of neural network with two layers and a very simple activation function based on addition and multiplication. Moreover, this model is guaranteed to converge in a finite number of steps (Rosenblatt, 1958). Minsky et al. demonstrated that models with a single layer are only able to learn linearly separable functions. Moreover, they also have shown that adding a single hidden layer to the model solves this problem (Minsky and Papert, 1969). One of the first multi-layer models that was introduced was the Neocognitron model (Fukushima, 1980).

Neural networks with hidden layers are generally called Multi-Layer Perceptrons (MLPs). It was shown that such models can be qualified of universal approximators. Indeed, a sufficiently large MLP could, in principle, approximate any function (Hornik, Stinchcombe, and White, 1989). The backpropagation algorithm, was introduced in 1982 as a general learning procedure for neural network (Werbos, 1982). It was later improved to handle more types of neural networks (LeCun, Boser, et al., 1989). The main form of backpropagation is the Stochastic Gradient Descent (SGD) algorithm (often used as synonym of backpropagation). This algorithm is the most used nowadays to train neural networks. This iterative algorithm presents one input of the problem to the network, computes the values of the output layer and compares them with the expected output. The difference,

called the error, is then propagated back to each layer from the output layer to the input layer. For each layer, the gradients of the errors are computed and applied to its weights. This process is repeated for each example from the data set and then repeated again as long as the objective classification error is not reached. One application of the entire data set is called an epoch.

The network must learn a general solution and not a solution that is too tightly coupled with the training examples, a problem known as overfitting. Indeed, the goal is not to classify these training samples, but rather to classify samples from an independent test set composed of unseen data. For this, regularization methods are used during training. There are several methods for this, the most simple techniques are simply constraining the weights to not grow too large while other techniques are adding stochastic information during training to not let the network know too much about the exact training samples (Srivastava et al., 2014). The use of data augmentation (generating artificial samples from the existing samples) is also a very efficient technique to solve overfitting (Jarrett, Kavukcuoglu, Lecun, et al., 2009).

There are two main families of neural network:

1. Feed-forward neural network: This is the earliest form of neural network and arguably the simplest. The defining point of a feed-forward neural network is the absence of cycles in the graph of the connections between neurons. The information moves only in one direction, from the input layer to the output layer, generally without any jumps around layers. The examples elaborated in this Section are feed-forward neural networks.
2. Recurrent Neural Networks (RNNs): There are cycles in the graph of the neuron connections. These cycles are time "delayed", giving the network the capability to take an internal state and which gives it temporal-like behaviour. These networks have proved very efficient for speech recognition (Sak, A. W. Senior, and Beaufays, 2014) and handwriting recognition (Graves et al., 2009) and even for image classification (Pinheiro and Collobert, 2014) in which they are able to learn the local dependencies between pixels. At the time of writing this document, the most used and successful recurrent model is the Long Short Term Memory (LSTM) model (Hochreiter and Schmidhuber, 1997).

This thesis solely focuses on feed-forward neural networks. There are several families of feed-forward networks depending on the connectivity of the neurons. In this work, we are most interested in two main families: Fully Connected Neural Networks (FCNNs) and Convolutional Neural Networks (CNNs).

The topology of a FCNN, or dense network, is the most classical one where every neuron from one layer is directly connected to every neuron in the next layer (See Figure 2.1 for instance). For an image input, it means that every output neuron is connected to every pixel in the input image. Each layer of such a network has

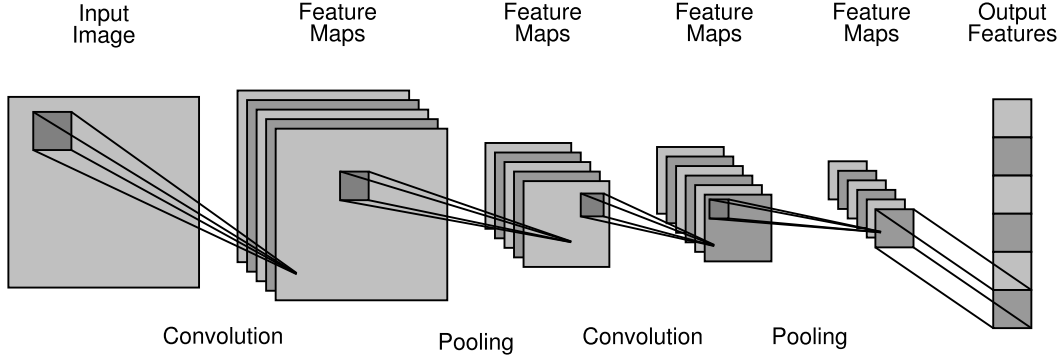


Figure 2.2: A Convolutional Neural Network with two convolutional layers and two pooling layers. The input layer will receive the input data. The final layer is simply the concatenation of the feature maps of the previous layer.

a matrix \mathbf{W} of weights connecting the input (or visible) units \mathbf{v} and the output (or hidden) units \mathbf{h} . It also has a vector of visible biases \mathbf{b} . Instead of being a linear combination of the inputs, the networks are generally using a non-linear activation function $N(\mathbf{x})$ such as the logistic sigmoid or the hyperbolic tangent in the mapping function. For instance, for a network with one layer and an activation function $N(\mathbf{x})$, the mapping function is:

$$\hat{\mathbf{y}} = \mathbf{h} = \hat{f}(\mathbf{x}) = N(\mathbf{b} + \mathbf{x} * \mathbf{W}) \quad (2.5)$$

When a network with several layers is used, a mapping function is used for each layer and the outputs from the first layer are passed to the next layer. For instance, with one hidden layer:

$$\mathbf{h}_1 = \hat{f}_1(\mathbf{x}) = N_1(\mathbf{b}_1 + \mathbf{x} * \mathbf{W}_1) \quad (2.6)$$

$$\hat{\mathbf{y}} = \mathbf{h}_2 = \hat{f}_2(\mathbf{h}_1) = N_2(\mathbf{b}_2 + \mathbf{h}_1 * \mathbf{W}_2) \quad (2.7)$$

This is easily generalized to more layers. The mapping function of the complete network is the combination of the mapping functions of each layer. The training objective will be to select the weights and biases for each layer in order to minimize the overall cost function.

In CNNs, the connectivity between layers is done using a convolution, generally two-dimensional. In such a model, each set of weights is shared on the entire image rather than being tied to a spatial location. This model has been developed from the model of the visual cortex where the neurons connected to the retina are arranged in layers and each layer has access to a larger receptive field than the previous one (Hubel and Wiesel, 1962). Moreover, neurons adjacent to each other

also have adjacent receptive fields in the retina. In convolutional networks, each output neuron is connected to only a portion of the input space (LeCun, Bottou, et al., 1998). An example of CNN is shown in Figure 2.2. Inputs of type image in a CNN are generally three-dimensional along the x and y axes and the color channels with respectively, dimensions N_V^1 for the width, N_V^2 for the height and C for the number of channels. The number of channels is generally the number of color channels in image inputs (one for grayscale and binary and three for color images). There are generally K groups of output neurons in that configuration and each group has a set of weights \mathbf{W}_k (only one set per group, not per neuron, thus the name of shared weights) to compute the output values from the set of inputs. Since the units are connected using a two-dimensional convolution, each set of weights is a two-dimensional convolutional kernel ($[N_K^1, N_K^2]$). Moreover, the input channels are generally combined together and do not appear in the outputs. Therefore, there is also a set of weights per input channel, making the matrix \mathbf{W} four-dimensional ($[C, K, N_K^1, N_K^2]$). Thus, the output is also three-dimensional (K , the output width N_H^1 and the output height N_H^2). Each of the K groups is producing a two-dimensional feature map using its own convolutional matrix. Generally, each group also has one bias \mathbf{b}_k . If we consider images as input, each output neuron would be connected to a small window in the image. Contrary to a fully-connected ANN, a CNN will generate a set of feature maps for each input. Each convolutional layer has access to a larger part of the input than its predecessor. Convolutional layer are often followed by pooling layers that are shrinking the representation. This model is more recent than fully-connected networks and is the current state of the art for image analysis problems with feed-forward neural networks. The mapping function can be defined similarly as for the dense layer. For instance, for one convolutional layer:

$$\hat{\mathbf{y}}^k = \mathbf{H}^k = \hat{f}(\mathbf{x}) = N(\mathbf{b}^k + \sum_{c=1}^C (\tilde{\mathbf{W}}_c^k \bullet_v \mathbf{x})) \quad (2.8)$$

When several layers are used, the combination is the same as for a fully-connected network, the output of one layer becomes the input of the next layer. However, generally, the expected output \mathbf{y} is not three-dimensional. Therefore, convolutional layers are generally followed by final dense layers for computing the final $\hat{\mathbf{y}}$. Although this network is now composed of both dense and convolutional layers, the result is still called a CNN. For a network with one convolutional layer and one dense layer, the mapping function can be composed as such:

$$\mathbf{H}_1^k = \hat{f}(\mathbf{x}) = N_1(\mathbf{b}^k + \sum_{c=1}^C (\tilde{\mathbf{W}}_c^k \bullet_v \mathbf{x})) \quad (2.9)$$

$$h1 = \text{flatten}([H_1^0, H_1^1, \dots, H_1^K]) \quad (2.10)$$

$$\hat{\mathbf{y}} = \mathbf{h}_2 = \hat{f}_2(\mathbf{h}_1) = N_2(\mathbf{b}_2 + \mathbf{h}_1 * \mathbf{W}_2) \quad (2.11)$$

The three-dimensional matrix output of the first layer is flattened as a vector for the second layer. Again, this can easily be expanded for more layers. In theory, networks composed of arbitrary number of layers could be created. Nevertheless, networks with many layers are harder to train for both computational reasons and problems such as the vanishing gradient (See the next Section).

2.3 Deep Learning

While generally presented as a new trend in Machine Learning, Deep Learning is already several decades old. Indeed, as seen in the previous Section, it dates back to the first multi-layer ANN, introduced in the 1980s. However, training such models was impractical, due to the long training times and the small processing power available at the time. Moreover, these structures also suffered from the so-called vanishing gradient problem (Hochreiter, 1991). When training a deep neural network with gradient descent, each weight receives an update proportional to the gradient of the error with respect to the current weight. These gradients are generally small numbers (in the $[-1, 1]$ range). For a network with N layers, the gradient of the first layer is composed of the multiplication of N of these small numbers, resulting in an even smaller number. Due to this problem, networks with more than one hidden layer were difficult to train. Moreover, the weights of these networks were randomly initialized, making the convergence of the training highly dependent on the quality of the random initialization. Indeed, training may be stuck in a local minima due to poor initial conditions. Several very advanced training algorithm have been developed to overcome this issue (Ngiam et al., 2011), such as Marquardt (Hagan and Menhaj, 1994), Conjugate Gradient (CG) (Blue and Grother, 1992) or Limited-Memory Broyden-Fletcher-Goldfarb-Shanno (L-BFGS) (Byrd et al., 1995). However, these techniques, while successful, make the training of deep models a very complicated process. Finally, the training of neural networks only uses labeled data, which are costly to produce, almost all data being unlabeled in practice. Due to these issues, simpler models such as the Support Vector Machine (SVM) became the popular choice in the 1990s and early 2000s and progress in Deep Learning was cut short and was only researched by few groups.

Deep Learning regained adoption after Hinton and Salakhutdinov presented a novel way to pre-train a multi-layer feed-forward neural network, in an efficient manner, one layer at a time, using several Restricted Boltzmann Machines (RBMs) stacked in a model called a Deep Belief Network (DBN) (G. E. Hinton, Osindero, and Teh, 2006; G. E. Hinton and Salakhutdinov, 2006; G. E. Hinton, 2007). The complete network was then fine-tuned using standard backpropagation. This particular method achieved state of the art results and beat the best SVM at the time, without considering against results using data augmentation. Since then, Deep Learning gained wide adoption and it restarted research on deep networks, especially CNNs. From this point on, Deep Learning technologies achieved numerous outstanding results.

We can observe that there are two "branches" of Deep Learning. The first, the "Hinton approach", is following the Hinton original approach and uses RBM and its variants to initialize the weights of large neural networks that are then fine-tuned for classification. These networks can also be directly used for feature extraction. The second "branch" has followed the restart of Deep Learning to train larger and larger neural networks. This has led to many new techniques such as Dropout (Srivastava et al., 2014) and Batch Normalization (Ioffe and Szegedy, 2015). This also led to the design of very deep networks such as the Inception Network (Szegedy, Liu, et al., 2015) or the Residual Network (He et al., 2016) composed of more than twenty layers. These new techniques and models allowed researchers to train networks that were thought to be untrainable a decade ago. Although the first "branch" was very popular at first and led to several breakthroughs, the second branch ended up overshadowing it with purely supervised learning (LeCun, Bengio, and G. E. Hinton, 2015). Chapter 3 analyzes the reasons of this decline.

Deep networks have also largely been helped by the advances in hardware which is much faster than thirty years ago and the introduction of Graphical Processing Units (GPUs) as a general-purpose processing unit. When the new training techniques and new hardware are used together for training learning models, it is now possible to train very large networks orders of magnitude faster than it used to be at the beginning of Deep Learning research. This somehow decreases the importance of unsupervised pretraining as an initialization of the weights to speed up training since training is now significantly easier than it used to be.

Deep Learning comprises a large number of models and concepts such as MLP, DBN, CNN, RNN and Deep SVM. This thesis focuses on the family of networks based on the RBM and Convolutional Restricted Boltzmann Machine (CRBM) models, the so-called "Hinton approach", detailed in the next Section.

2.4 Restricted Boltzman Machine

A Restricted Boltzmann Machine (RBM) is a generative stochastic ANN. It is a model especially made to learn a probability distribution over the inputs. They were initially introduced by Smolensky et al. in 1986, under the name Harmonium (Smolensky, 1986). Although they are quite ancient models, they only rose to a large audience after Hinton et al. invented a fast learning algorithm to train them in 2002 (G. E. Hinton, 2002).

An RBM is a variant of the normal class of Boltzmann Machine, proposed by Hinton et al. (Ackley, G. E. Hinton, and Sejnowski, 1985). Figure 2.3 illustrates an RBM. It is made of two layers, a visible layer and a hidden layer. Both layers contain a certain number of units (or neurons). In an RBM, the "restricted" means that neurons form a bipartite graph, i.e. there are no connections between units of the same group. This special restriction makes for more efficient algorithms to train the model, contrary to the general class of Boltzmann Machines. In essence, an RBM is a simple ANN with a single visible layer and an output layer and

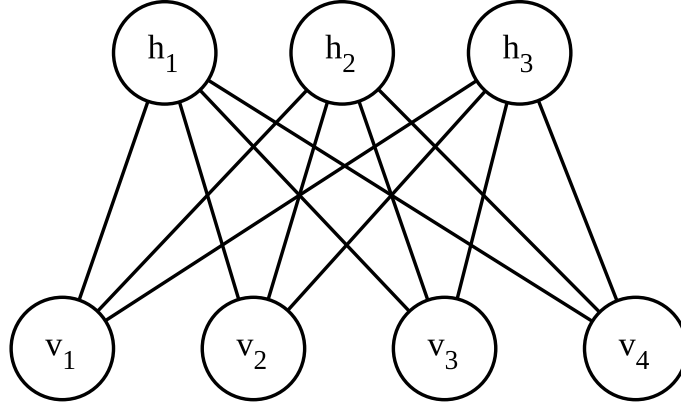


Figure 2.3: Abstract view of a Restricted Boltzmann Machine, with four visible units v and three hidden units h . There are no connections between units of the same layer.

no hidden layers in between. It can also be viewed as an unfolded neural network with three layers, the output layer having the same number of neurons as the input layer, and using a common set of weights for both layers. The main difference is how the model is trained and in its ability to reconstruct its input from its output.

An RBM model can be represented with a vector \mathbf{v} of M visible units, a vector \mathbf{h} of N hidden units, a matrix \mathbf{W} of weights connecting the visible and the hidden units ($[M \times N]$ matrix), a vector \mathbf{b} of M visible biases and a vector \mathbf{c} of N hidden biases. As a standard neural network, the weights are directly connected, meaning that an output neuron is connected to each input neuron. This means that the mapping function uses the matrix multiplication operation to connect the two layers. Contrary to standard neural networks, a unit has a probability and a state and both are used during training. The activation probability is the probability of a unit to be activated while the state is the result of sampling the probability. When an RBM is used as a feature extractor, the extracted features are the activation probabilities of the output (hidden) layer.

In Energy Based Models (EBMs), any configuration of the units has a scalar energy (Hopfield, 1982). The energy of a joint configuration can be used to compute its probability. In the case of an EBM with hidden units, such as the RBM, the probability of a joint configuration of the visible and hidden units can be expressed directly:

$$p(\mathbf{v}, \mathbf{h}) = \frac{1}{Z} e^{-E(\mathbf{v}, \mathbf{h})} \quad (2.12)$$

where Z represents the *partition function* of the model. It can be computed using the sum of the energies of all the possible joint configurations of the visible and hidden units:

$$Z = \sum_{\mathbf{v}, \mathbf{h}} e^{-E(\mathbf{v}, \mathbf{h})} \quad (2.13)$$

However, computing Z is in practice untractable as it involves a summation over all potential combinations of (\mathbf{v}, \mathbf{h}) vectors. If θ represents the parameters of the system ($\theta = (\mathbf{W}, \mathbf{b}, \mathbf{c})$ for a standard RBM), $Z(\theta)$ is called the *normalization constant* of the model.

The probability of the observed variables can also be expressed in terms of the probabilities of the joint configurations:

$$p(\mathbf{v}) = \frac{1}{Z} \sum_{\mathbf{h}} e^{-E(\mathbf{v}, \mathbf{h})} \quad (2.14)$$

The energy of a joint configuration of the visible and hidden units in an RBM can be defined in terms of its components:

$$E(\mathbf{v}, \mathbf{h}) = -(\mathbf{b} \cdot \mathbf{v}) - (\mathbf{c} \cdot \mathbf{h}) - (\mathbf{h} \cdot (\mathbf{W} * \mathbf{v})) \quad (2.15)$$

The free energy $F(\mathbf{v})$ of an EBM for a visible vector \mathbf{v} corresponds to the energy that a configuration would need to have the same probability as all of the configurations containing \mathbf{v} :

$$e^{-F(\mathbf{v})} = \sum_{\mathbf{h}} e^{-E(\mathbf{v}, \mathbf{h})} \quad (2.16)$$

$$F(\mathbf{v}) = -\log \sum_{\mathbf{h}} e^{-E(\mathbf{v}, \mathbf{h})} \quad (2.17)$$

The free energy can also be used to express $p(\mathbf{v})$:

$$p(\mathbf{v}) = \frac{1}{Z} e^{-F(\mathbf{v})} \quad (2.18)$$

From the definition of $F(\mathbf{v})$ and $E(\mathbf{v}, \mathbf{h})$, the free energy function of an RBM can be completely derived:

$$F(\mathbf{v}) = -(\mathbf{b} \cdot \mathbf{v}) - \sum_{\mathbf{v}} \log \sum_{\mathbf{h}} e^{\mathbf{h}(\mathbf{c} + \mathbf{W}\mathbf{v})} \quad (2.19)$$

Since a lower energy for a pattern corresponds to an higher probability for this pattern to be generated, the training of the RBM will try to minimize the free energy of the training patterns and to maximize the free energy of the other patterns. In other words, the training shall maximize $p(\mathbf{v})$, implying a minimization of $F(\mathbf{v})$.

Free energy itself has several uses. It can be used as a way to monitor overfitting, by comparing the difference between the free energy on the training set and the validation set. The partition function being the same for both sets with the same model, the comparison of free energies is meaningful. It can also be used for classification when separate RBMs are used for each class. For each sample to classify, the free energy of this sample on each RBM is computed and then a softmax classifier is trained on top of the free energies.

Since visible and hidden units are conditionally independent given one-another, the general probability activation functions can be defined, for a general case, as:

$$p(\mathbf{h}|\mathbf{v}) = \prod_{\mathbf{h}} p(\mathbf{h}|\mathbf{v}) \quad (2.20)$$

$$p(\mathbf{v}|\mathbf{h}) = \prod_{\mathbf{v}} p(\mathbf{v}|\mathbf{h}) \quad (2.21)$$

In practice, the general formulas for the RBM are not used as such but are simplified for each sub-type of RBM, based on the types of visible and hidden units. Once simplified, these formulas allow to make inference in an RBM relatively straightforward.

2.4.1 Binary-Binary Restricted Boltzmann Machine

The original RBM was developed for binary-valued inputs. It consists of binary visible units and binary hidden units. A binary unit is based on the logistic sigmoid activation function and uses Bernoulli sampling to sample its state. Binary units are also called Bernoulli units. This is the most common type of RBM. When both units are binary, the activation functions can be defined as:

$$\mathbf{h}_j = p(\mathbf{h}_j = 1|\mathbf{v}) = \sigma(\mathbf{c}_j + \sum_{i=1}^M \mathbf{v}_i \mathbf{W}_{i,j}) \quad (2.22)$$

$$\mathbf{v}_i = p(\mathbf{v}_i = 1|\mathbf{h}) = \sigma(\mathbf{b}_i + \sum_{j=1}^N \mathbf{h}_j \mathbf{W}_{i,j}) \quad (2.23)$$

or in vector form:

$$\mathbf{h} = \sigma(\mathbf{c} + \mathbf{v} * \mathbf{W}) \quad (2.24)$$

$$\mathbf{v} = \sigma(\mathbf{b} + \mathbf{W} * \mathbf{h}) \quad (2.25)$$

where the logistic sigmoid function σ is defined as:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \quad (2.26)$$

The result of the activation function for a unit is called its activation probability. The states of the units are obtained by sampling these activation probabilities. Each type of unit has a sampling function $S(x)$ that is used to obtain the state of the unit from its activation. For binary units, the states are obtained using Bernoulli sampling:

$$S(x) = \begin{cases} 1 & \text{if } \mathbf{x} > \text{Unif}(0, 1) \\ 0 & \text{otherwise} \end{cases} \quad (2.27)$$

$$\mathbf{h}'_j = S(\mathbf{h}_j) \quad (2.28)$$

$$\mathbf{v}'_j = S(\mathbf{v}_j) \quad (2.29)$$

For example the state \mathbf{h}'_j is set to 1 if the computed activation \mathbf{h}_j is bigger than a sample in an uniform distribution computed between 0 and 1. These states are generally only used during learning.

When both layers are made of binary units, the free energy function further simplifies to:

$$F(\mathbf{v}) = -(\mathbf{b} \cdot \mathbf{v}) - \sum_{j=1}^N \log(1 + \exp(\mathbf{c}_j + \sum_{i=1}^M \mathbf{v}_i \mathbf{W}_{i,j})) \quad (2.30)$$

This particular RBM model is the most common and most general variant of RBM. However, since the introduction of the RBM, different types of units have been developed, for various purposes, as shown in Section 2.4.6. Moreover, different variants of RBM have also been proposed (See Section 2.8).

2.4.2 Training

As indicated before, the goal of training an RBM is to make it model the input distribution as well as possible. For this, the input samples must be made more

likely and the other samples less likely during training. In other words, the probabilities of the input samples $p(\mathbf{x})$ must be maximized. For this, the following cost function can be defined:

$$J = -\frac{1}{N} \prod_{i=1}^N p(\mathbf{x}_i) \quad (2.31)$$

In practice, logarithms of the probabilities are generally preferred in order to improve stability:

$$J = \frac{1}{N} \sum_{i=1}^N -\log(p(\mathbf{x}_i)) \quad (2.32)$$

This is equivalent to the average of Negative Log-Likelihood (NLL) cost function. To minimize this cost function, stochastic gradient descent can be used, implying the derivative of the function for one sample with respect to the parameters of the system:

$$J(\theta) = \frac{\partial -\log(p(\mathbf{x}_i))}{\partial \theta} \quad (2.33)$$

$$= \underbrace{\mathbb{E}_{\mathbf{h}} \left[\frac{\partial E(\mathbf{x}_i, \mathbf{h})}{\partial \theta} \middle| \mathbf{x}_i \right]}_{\text{positive phase}} - \underbrace{\mathbb{E}_{\mathbf{x}, \mathbf{h}} \left[\frac{\partial E(\mathbf{x}, \mathbf{h})}{\partial \theta} \right]}_{\text{negative phase}} \quad (2.34)$$

The two parts of the gradients are generally referred as the positive and negative phases. The gradients of the positive phase can be carried out since it is conditioned on the value of a training sample. However, the gradients of the negative phase are intractable. Therefore, it is necessary to approximate this negative term in order to efficiently perform gradient descent. For this, the Contrastive Divergence (CD) algorithm is generally used, as seen in the following Section.

2.4.3 Contrastive Divergence

The general learning algorithm for Boltzmann Machine (G. E. Hinton and Sejnowski, 1986) is fairly simple. It tries to minimize the Kullback-Leibler divergence (Kullback and Leibler, 1951) of the real distribution and the distribution generated by the model. From this, the gradient G can be derived easily and a gradient descent procedure can be used to optimize the distribution. However, this procedure is slow to converge in practice and the time to reach a temperature equilibrium grows exponentially with the size of the model. This is the main reason why general Boltzmann Machines did not have many practical applications.

While an RBM could be trained using the general Boltzmann Machine learning algorithm, its *bipartite graph* property allows for more efficient training. The most often used algorithm to train an RBM is the Contrastive Divergence (CD) algorithm. This algorithm has been initially developed to train Product of Experts (PoE) (G. E. Hinton, 2002) and has then been adapted for RBM (G. E. Hinton, Osindero, and Teh, 2006). Indeed, since the probability of generating a visible vector is linked to the probabilities of generating it by each of the hidden unit individually (Freund and Haussler, 1994), an RBM can be viewed as a PoE with one expert for each hidden unit.

Samples from an RBM can be obtained using Gibbs Sampling, a Markov Chain Monte-Carlo (MCMC) algorithm, and running a Markov chain to convergence. The CD algorithm is based on this sampling process, but does two specific optimizations. First, instead of starting at a random state of the visible units, it starts at the state of a training vector. Moreover, instead of waiting for the convergence of the Markov chain, it simply takes its state after K steps. This has the advantage of not requiring to perform alternating Gibbs Sampling for many iterations. Overall, these improvements are making CD a very fast algorithm. This learning rule is closely related to the gradient of the difference of two Kullback-Liebler divergences. Although the objective does not exactly follow the gradient of any function (Sutskever and Tieleman, 2010), it has been successful in many applications.

Going back to Equation 2.34, this is equivalent to replacing the expectation of the model by an estimate obtained by Gibbs sampling from the input.

$$\underbrace{\mathbb{E}_{\mathbf{h}} \left[\frac{\partial E(\mathbf{x}_i, \mathbf{h})}{\partial \theta} \middle| \mathbf{x}_i \right]}_{\text{positive phase}} \approx \frac{\partial E(\mathbf{x}_i, p(\mathbf{h}|\mathbf{x}_i))}{\partial \theta} \quad (2.35)$$

$$\underbrace{\mathbb{E}_{\mathbf{x}, \mathbf{h}} \left[\frac{\partial E(\mathbf{x}, \mathbf{h})}{\partial \theta} \right]}_{\text{negative phase}} \approx \frac{\partial E(\tilde{\mathbf{x}}, \tilde{\mathbf{h}})}{\partial \theta} \quad (2.36)$$

where $\tilde{\mathbf{x}}$ and $\tilde{\mathbf{h}}$ are obtained by Gibbs sampling. In fact, it can be shown that for a Binary-Binary RBM, the positive phase value is not an approximation but the correct value. Only the negative phase is approximate using Gibbs sampling and CD. The partial derivatives of the $E(\mathbf{x}, \mathbf{h})$ with respect to the weights can be obtained:

$$\frac{\partial E(\mathbf{x}, \mathbf{h})}{\partial \mathbf{W}_{j,k}} = \frac{\partial - (\sum_{j',k'} \mathbf{W}_{j',k'} \mathbf{h}_{j'} \mathbf{x}_{k'} - \sum_{k'} \mathbf{c}_{k'} \mathbf{x}_{k'} - \sum_{j'} \mathbf{b}_{j'} \mathbf{h}_{j'})}{\partial \mathbf{W}_{j,k}} \quad (2.37)$$

$$= \frac{\partial - (\sum_{j',k'} \mathbf{W}_{j',k'} h_{j'} \mathbf{x}_{k'})}{\partial \mathbf{W}_{j,k}} \quad (2.38)$$

$$= -(\mathbf{h}_j \mathbf{x}_k) \quad (2.39)$$

Similarly, the partial derivatives with respect to the hidden biases can be obtained:

$$\frac{\partial E(\mathbf{x}, \mathbf{h})}{\partial \mathbf{b}_j} = \frac{\partial - (\sum_{j', k'} \mathbf{W}_{j', k'} \mathbf{h}_{j'} \mathbf{x}_{k'} - \sum_{k'} \mathbf{c}_{k'} \mathbf{x}_{k'} - \sum_{j'} \mathbf{b}_{j'} \mathbf{h}_{j'})}{\partial \mathbf{b}_j} \quad (2.40)$$

$$= \frac{\partial - (\sum_{j'} \mathbf{b}_{j'} \mathbf{h}_{j'})}{\partial \mathbf{b}_j} \quad (2.41)$$

$$= -\mathbf{h}_j \quad (2.42)$$

and, finally, with respect to the visible biases:

$$\frac{\partial E(\mathbf{x}, \mathbf{h})}{\partial \mathbf{c}_k} = \frac{\partial - (\sum_{j', k'} \mathbf{W}_{j', k'} \mathbf{h}_{j'} \mathbf{x}_{k'} - \sum_{k'} \mathbf{c}_{k'} \mathbf{x}_{k'} - \sum_{j'} \mathbf{b}_{j'} \mathbf{h}_{j'})}{\partial \mathbf{c}_k} \quad (2.43)$$

$$= \frac{\partial - (\sum_{k'} \mathbf{c}_{k'} \mathbf{x}_{k'})}{\partial \mathbf{c}_k} \quad (2.44)$$

$$= -\mathbf{x}_k \quad (2.45)$$

To summarize, the following derivatives in vector form are used:

$$\frac{\partial E(\mathbf{x}, \mathbf{h})}{\partial \mathbf{W}} = -(\mathbf{h} \otimes \mathbf{x}) = -\mathbf{h} \mathbf{x}^T \quad (2.46)$$

$$\frac{\partial E(\mathbf{x}, \mathbf{h})}{\partial \mathbf{b}} = -\mathbf{h} \quad (2.47)$$

$$\frac{\partial E(\mathbf{x}, \mathbf{h})}{\partial \mathbf{c}} = -\mathbf{x} \quad (2.48)$$

From this, the complete update rules using the positive and negative phases are obtained:

$$\mathbf{W} = \mathbf{W} + \epsilon(\mathbf{v}^0 \otimes \mathbf{h}^0 - \mathbf{v}^K \otimes \mathbf{h}^K) \quad (2.49)$$

$$\mathbf{b} = \mathbf{b} + \epsilon(\mathbf{v}^0 - \mathbf{v}^K) \quad (2.50)$$

$$\mathbf{c} = \mathbf{c} + \epsilon(\mathbf{h}^0 - \mathbf{h}^K) \quad (2.51)$$

where $\mathbf{v}^0 \otimes \mathbf{h}^0$ represents the expectations under the input distribution and $\mathbf{v}^K \otimes \mathbf{h}^K$ represents the expectations driven by the model.

The algorithm is illustrated in Figure 2.4. It is very similar to the stochastic gradient descent procedure used to train an ANN with backpropagation. The CD algorithm can be done with a certain number K of steps of Gibbs sampling (called CD- K). While increasing the number of steps highly improves the quality of learning as an approximation of the maximum log-likelihood, CD-1 is generally sufficient in practice when the RBM is used to learn to extract features or to

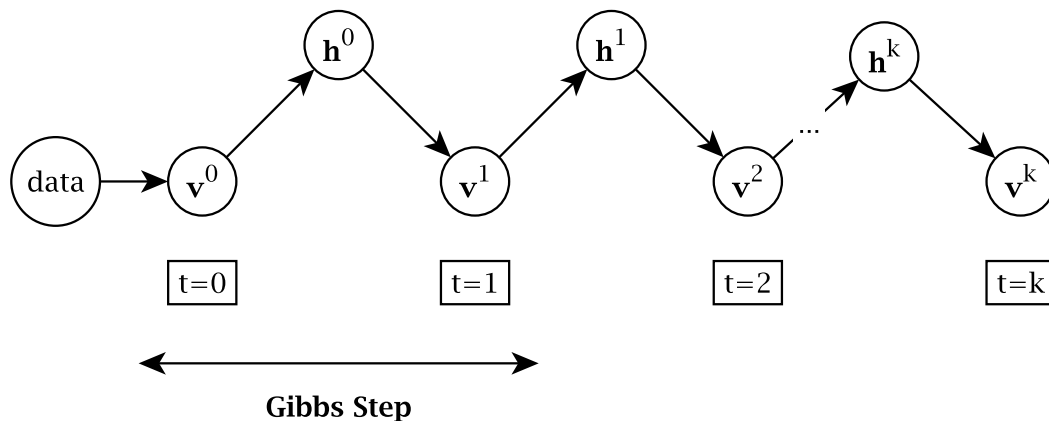


Figure 2.4: Graphical representation of the Contrastive Divergence Algorithm. The algorithm CD-K stops at $t=K$. Each iteration performs a full Gibbs step.

Algorithm 2.1 Standard CD-K algorithm. This performs one epoch of training through the complete training set.

for all training sample $\mathbf{x} \in$ training set **do**

$\mathbf{v}^0 = \mathbf{x}$

$\mathbf{h}^0 = p(\mathbf{h}|\mathbf{v}^0)$

$\mathbf{h}'^0 = S(\mathbf{h}^0)$

for $k \leftarrow 1, K$ **do**

$\mathbf{v}^k = p(\mathbf{v}|\mathbf{h}'^{k-1})$

$\mathbf{h}^k = p(\mathbf{h}|\mathbf{v}^k)$

$\mathbf{h}'^k = S(\mathbf{h}^k)$

end for

$\mathbf{W}^{pos} = \mathbf{v}^0 \otimes \mathbf{h}^0$

$\mathbf{W}^{neg} = \mathbf{v}^K \otimes \mathbf{h}^K$

$\nabla \mathbf{W} = \epsilon(\mathbf{W}^{pos} - \mathbf{W}^{neg})$

$\nabla \mathbf{b} = \epsilon(\mathbf{v}^0 - \mathbf{v}^K)$

$\nabla \mathbf{c} = \epsilon(\mathbf{h}^0 - \mathbf{h}^K)$

$\mathbf{W} = \mathbf{W} + \nabla \mathbf{W}$

$\mathbf{b} = \mathbf{b} + \nabla \mathbf{b}$

$\mathbf{c} = \mathbf{c} + \nabla \mathbf{c}$

end for

initialize the weights of a neural network (G. E. Hinton, 2012). Algorithm 2.1 defines the steps necessary for one epoch of CD-K.

This procedure is repeated for a certain number of epochs, until an acceptable convergence is reached. The condition for stopping the training is not trivial and depends on how the RBM will be used (see Section 2.4.4).

Similarly to the training of an ANN, momentum is generally used to speed up the training. Momentum (Bertsekas, 1999) is a technique that limits the oscillations of the gradient descent and accelerates the learning in the correct direction. This is done by adding a fraction of the update at the past step to the current update. This can be seen as pushing a ball down the hill. It will accumulate momentum downhill, the target of training. When momentum is used, the weights update equations must be updated as follows:

$$\nabla \mathbf{W}(t) = \alpha \nabla \mathbf{W}(t-1) + \epsilon(\mathbf{v}^0 \otimes \mathbf{h}^0 - \mathbf{v}^K \otimes \mathbf{h}^K) \quad (2.52)$$

$$\nabla \mathbf{b}(t) = \alpha \nabla \mathbf{b}(t-1) + \epsilon(\mathbf{v}^0 - \mathbf{v}^1) \quad (2.53)$$

$$\nabla \mathbf{c}(t) = \alpha \nabla \mathbf{c}(t-1) + \epsilon(\mathbf{h}^0 - \mathbf{h}^1) \quad (2.54)$$

The momentum (α) is generally set to a value between 0.5 and 0.9. With a value of zero, the momentum has no effect.

Similarly, regularization such as weight decay is often applied to the gradients. This improves the mixing rate of the Gibbs Markov Chain, reduces overfitting to the training data and makes units smoother. This is achieved by adding an extra term (the penalty function $P(\mathbf{i})$, \mathbf{i} being the vector to regularize, typically the gradients of the weight matrix) to the gradient (before momentum and before applying the gradients to the weights):

$$\nabla \mathbf{W} = \nabla \mathbf{W} - P(\nabla \mathbf{W}) \quad (2.55)$$

The penalty function is generally chosen to penalize large weights. There are several possible penalty functions, L1 and L2 penalty functions being the most used ones:

$$L1(\mathbf{i}) = \lambda * |\mathbf{i}| \quad (2.56)$$

$$L2(\mathbf{i}) = \lambda * \mathbf{i} \quad (2.57)$$

L1 regularization is sometimes preferred for images since it may result in more localized features, while L2 regularization is often used by default in other cases. These functions are generally parametrized using a weight cost (λ). Values for λ are generally ranging from $1e^{-2}$ to $1e^{-5}$. It is also possible to use both L1 and L2 regularization techniques at the same time. Although it is rarely done, weight decay can also be applied on the gradients of the biases.

In practice, there are three ways to apply gradient descent on a network:

1. Stochastic (or Online) Gradient Descent. The gradients are computed for a single example and directly applied to the weights
2. Batch Gradient Descent: The gradients are computed for the complete data set and only then applied to the weights
3. Mini-Batch Gradient Descent: The gradients are computed for a small subset of size B before being applied to the weights.

The same three options are available for CD training. In practice, mini-batch training is generally preferred over the other two versions. Unless the error manifold of the model is very smooth, which is rarely the case (there are often a lot of local minima), Mini-Batch and Stochastic Gradient Descent perform significantly better than Batch Gradient Descent. It is also generally faster to converge, since the weights are updated much more often. Moreover, mini-batch has several advantages over stochastic gradient descent. First, it allows the gradients to be smoothed, avoiding issues with very large outliers. It is also much faster, allowing vectorization of the gradient computation.

Algorithm 2.2 Mini-batch CD-k algorithm, one epoch

for all mini batch $b \in$ training set **do**

for all image $\mathbf{x} \in b$ **do**

$\mathbf{v}^0 = \mathbf{x}$

$\mathbf{h}^0 = p(\mathbf{h}|\mathbf{v}^0)$

$\mathbf{h}'^0 = S(\mathbf{h}^0)$

for $i \leftarrow 1, K$ **do**

$\mathbf{v}^k = p(\mathbf{v}|\mathbf{h}'^{k-1})$

$\mathbf{h}^k = p(\mathbf{h}|\mathbf{v}^k)$

$\mathbf{h}'^k = S(\mathbf{h}^k)$

end for

$\mathbf{W}^{pos} \leftarrow \mathbf{W}^{pos} + \mathbf{v}^0 \otimes \mathbf{h}^0$

$\mathbf{W}^{neg} \leftarrow \mathbf{W}^{neg} + \mathbf{v}^K \otimes \mathbf{h}^K$

end for

$\nabla \mathbf{W} = \frac{\epsilon}{B} (\mathbf{W}^{pos} - \mathbf{W}^{neg})$

$\nabla \mathbf{b} = \frac{\epsilon}{B} (\mathbf{v}^0 - \mathbf{v}^K)$

$\nabla \mathbf{c} = \frac{\epsilon}{B} (\mathbf{h}^0 - \mathbf{h}^K)$

$\mathbf{W} = \mathbf{W} + \nabla \mathbf{W}$

$\mathbf{b} = \mathbf{b} + \nabla \mathbf{b}$

$\mathbf{c} = \mathbf{c} + \nabla \mathbf{c}$

end for

For mini-batch gradient descent, the entire data set is cut into small mini-batches of size B (generally ranging from 20 to 300 samples, depending on the data set).

The gradients are then computed on the complete mini-batch before being applied to the weights. The algorithm remains mostly the same, except that all the activations and gradients are computed on each sample of the mini-batch before being applied to the weights. Before being added to the weights, the sum of the gradients must be divided by the size of the mini-batch to avoid the update being dependent on the mini-batch size. Algorithm 2.2 describes the procedure in detail.

For the sake of consistency between different descent techniques, in this thesis, an epoch is considered as one training pass through the complete data set. This means that several updates to the weights are made during one epoch. One iteration is one update of the weights. On the other hand, some other research are describing an epoch as one update to the weights.

2.4.3.1 Sparsity regularization

Generally, hidden units that are rarely active are more useful than those that are active often. Moreover, sparse units are easier to interpret and are generally representing better features when the network is not meant to be fine-tuned. By default, binary hidden units have a tendency of being active about half of the time.

Sparsity of the binary hidden units can be enforced by setting a target p for the activation probability of each hidden unit. By using an additional penalty term, the actual probability q can be encouraged to be close to p . The standard way to compute this penalty term comes from (Nair and G. E. Hinton, 2009). An exponentially decaying average is used to estimate the actual probability of being active for each unit:

$$q_{new} = \kappa q_{old} + (1 - \kappa) q_{current} \quad (2.58)$$

The exponential decay is controlled with κ , it is called the sparsity decay rate. When trained with mini-batch CD, $q_{current}$ is the mean activation probability of the given hidden unit for the last mini-batch. Using the cross-entropy between the desired sparsity and the actual sparsity, the penalty z for each unit can be easily derived for binary units:

$$z = \omega * (q - p) \quad (2.59)$$

The derivative is scaled by ω for both the biases and the weights. This parameter is representing the learning rate of the sparsity, also called the sparsity cost. The decay rate is generally set to 0.9 and the sparsity cost must be chosen so that the average sparsity is leaning towards p while not interfering too much with the main learning objective. It is important that the penalty vector z is applied to the weight gradients and the biases gradients. If applied to only one of them, the other will compete in the opposite direction. It is also possible to use a global sparsity

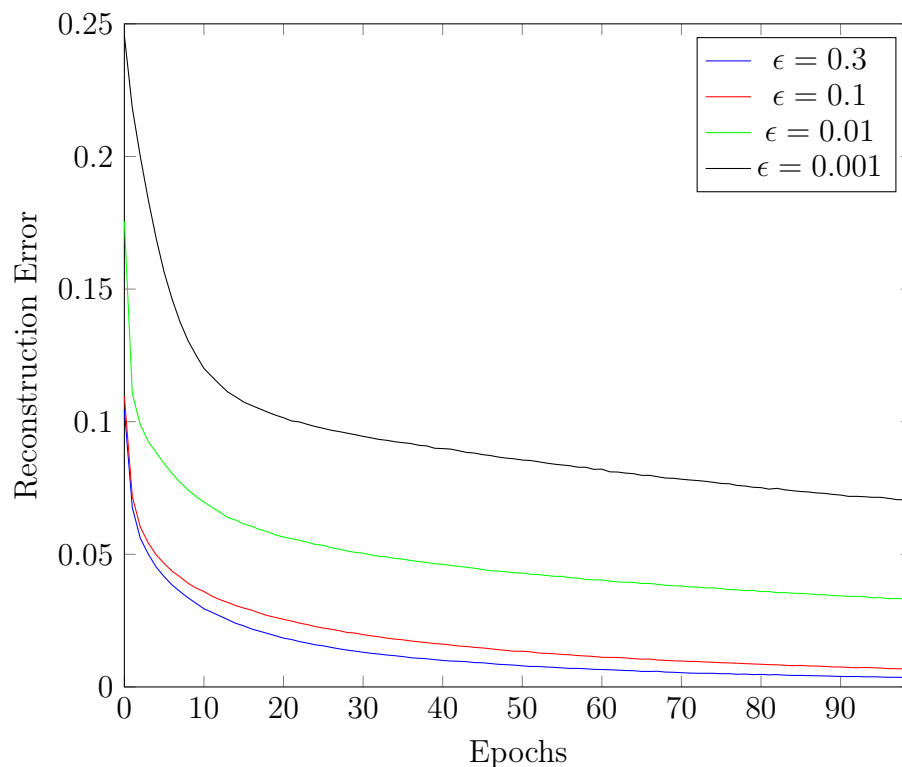


Figure 2.5: Impact of different learning rates on Contrastive Divergence training. This is for a subset of the MNIST data set.

target. However, this generally results in worse weights than using a target for each hidden unit.

While not its primary objective, this regularization method has also the advantage that units that are almost dead (with very low activation probability) will be revived so that their probability will be close to p . This may avoid having too many dead units.

2.4.4 Monitoring the learning progress

Since CD trains an RBM to reconstruct its input, the most obvious and simplest way to monitor the learning of an RBM is to compute the reconstruction error over the data set. This is a good metric if the RBM is especially trained for reconstruction. However, if the model is trained for later classification in a DBN or to extract features for another network, it rarely is an adequate metric. Also, when the objective is to use an RBM to model the probability distribution of the input, this is an inadequate metric. In that case, approximating the likelihood is generally performed (see Section 2.4.5). For instance, Figure 2.5 shows the evolution of the reconstruction error of an RBM over time with different learning rates. Generally, a higher learning rate will lead to a faster convergence of the reconstruction error. However, depending on the final objective, a large learning

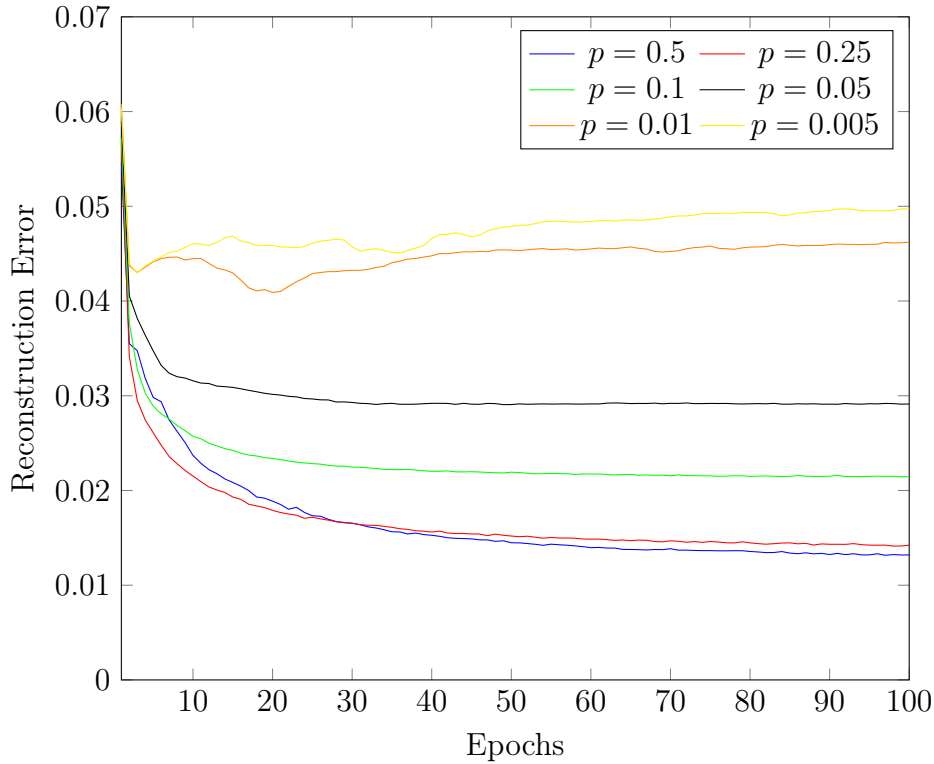


Figure 2.6: Impact of different sparsity targets on Contrastive Divergence training, with the same learning rate. This is for a subset of the MNIST data set. When the target is too small, the convergence of the reconstruction error is impeded.

rate is not necessarily a good idea. Indeed, a too high learning rate may also cause the learning to diverge or the network to overfit.

This technique can also be used to observe the impact of sparsity regularization during the training of an RBM. Figure 2.6 shows the reconstruction error rate during training with different sparsity target values. This was generated from a small network with 200 hidden units. Once the sparsity target is too small, the training does not converge to a small reconstruction error rate because there are not enough activated hidden units to compute a good representation of the data.

When the RBM is used to extract features or layers of features (see Section 2.5) for classification, it is generally possible to test the efficiency of these features by obtaining the final classification score. However, it generally takes a lot of time to perform the full classification training and testing, and measures that are obtainable without further training are generally preferred. Indeed, a measure of the quality during pretraining itself is preferable. This is the main reason why the reconstruction error rate is still often used, at least to verify that the network is learning correctly.

When the data has spatial or temporal structure, i.e. images or speech samples, visually displaying the weights can be very informative. If the weights look structured, it may be a good information about what the network has learned.



Figure 2.7: Visual representation of the features learned by an RBM on the MNIST data set.

Figure 2.7 shows the features learned by an RBM when trained on the MNIST data set (LeCun, Cortes, and Burges, 1998). It can clearly be seen that some filters have learned to detect single digits.

The basic idea for drawing filters is to scale the values of the weights in the grayscale range (from 0 to 255) and displaying these values. For fully-connected layers, each hidden unit is connected to every input unit and therefore can be represented as an image of the same size as the input. For convolutional models, each filter is represented as a grayscale image of the kernel size. When the input is a color image, there are generally three input channels and therefore the number of kernels is also multiplied by three. With that, the kernels can also be represented in color by drawing an image with all three channels for each filter. When the model has several layers, it is more difficult to display the weights of the upper layers. One solution is to take the training image that has the highest activation on each filter, but this does not tell anything about why this image acts strongly. One other way is to use another convolutional network to project the activation back in the input pixel space (Matthew D. Zeiler and Fergus, 2014). It is also possible to perform a linear combination of the weights between layers to obtain a good visualization

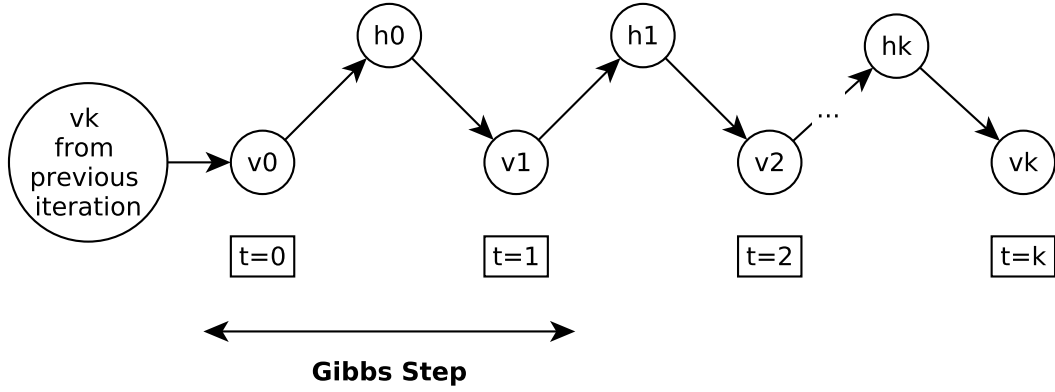


Figure 2.8: Graphical representation of the Persistent Contrastive Divergence Algorithm. Only the initialization of v_0 differs from CD. An iteration is one update of the weights.

(Erhan, Bengio, A. Courville, and Vincent, 2009).

Drawing the shape of the filters is the most used visual tool for monitoring RBM, but there are other ways to visually debug the training of an RBM, with histograms, probabilities and even three-dimensional filters (Yosinski and Lipson, 2012).

2.4.5 Persistent Contrastive Divergence

While CD-k is fast and is a reasonable approximation to the likelihood gradient and is able to substantially reduce the reconstruction error over the training epochs, it is still quite different from the likelihood gradient. To alleviate these issues, Persistent Contrastive Divergence (PCD) has been designed (Tieleman, 2008; Tieleman and G. E. Hinton, 2009). In this algorithm, instead of starting the Gibbs sampling at a data point at each parameter update, "persistent" Markov chains are kept and are not reset when the parameters are updated. This is presented in Figure 2.8. If the learning rate is small enough, this may produce better models than simply using CD, by improving the mixing rate of the Gibbs chains. Like CD, PCD can be performed with different numbers of Gibbs steps.

This algorithm can be implemented in many ways, for instance by resetting at interval the Markov chains or by randomly choosing Markov chains to reset. However, in practice, the Markov chains are never reset, there are as many Markov chains as there are samples in a mini-batch and a full Gibbs sampling step is done on each Markov chain for each parameter update. For this reason, PCD is rarely done without mini-batch training. The CD algorithm (see Section 2.4.3) can easily be transformed into PCD when using mini-batch. The only difference being that the visible units are only set to the value data points at the beginning of the training and not before each Gibbs sampling step. Moreover, the learning rate is

typically significantly smaller than for regular CD training.

PCD generally outperforms CD when the objective is to build a density model of the input data. In that case, the algorithm is closer to the optimal algorithm for modeling the probability distribution of the input. However, it is less accurate in generating reconstructions of the input. Therefore, the reconstruction error rate is not a good information on how the network is training. Since computing the exact likelihood is computationally intractable, it is necessary to use a proxy to the likelihood. The simplest method is to use the Pseudo-Likelihood (PL) proxy. For an RBM with binary units, assuming that all bits are independent, it can be computed as in the following:

$$PL(\mathbf{x}) = \prod_{i=1}^M p(\mathbf{x}_i | \mathbf{x}_{-i}) \quad (2.60)$$

$$\log PL(\mathbf{x}) = \sum_{i=1}^M \log p(\mathbf{x}_i | \mathbf{x}_{-i}) \quad (2.61)$$

In this formulas, \mathbf{x}_{-i} represents the set of all bits inside \mathbf{x} except for the bit i . Therefore, $\log PL(\mathbf{x})$ is a summation of the log probabilities of each bit of the hidden representation, conditioned by the state of all other bits. This is an expensive computation for the training most RBM.

There exist several faster approximations to the PL. The second approach is to estimate the normalization constant Z of the model. Once it is known, the likelihood can be computed in a straightforward manner (See Equation 2.12). The most used technique to approximate the normalization constant is Annealed Importance Sampling (AIS) (Neal, 2001; Salakhutdinov, 2009). AIS estimates the normalization constant of a model by computing the ratio between the normalization constants of two distributions. By choosing a base distribution from which the exact normalization constant can be computed (for instance, an RBM with zero-weights and zero-biases), the normalization constant of the target RBM can be estimated.

2.4.6 Other types of units

While binary hidden units are very efficient on binary-valued input data, it is not always possible to binarize data sets. Moreover, they sometimes prove slow to converge or are overfitting early in the training. For these reasons, several other types of units were developed for RBM.

Figure 2.9 shows the aspect of some of the activation functions used in this Section.

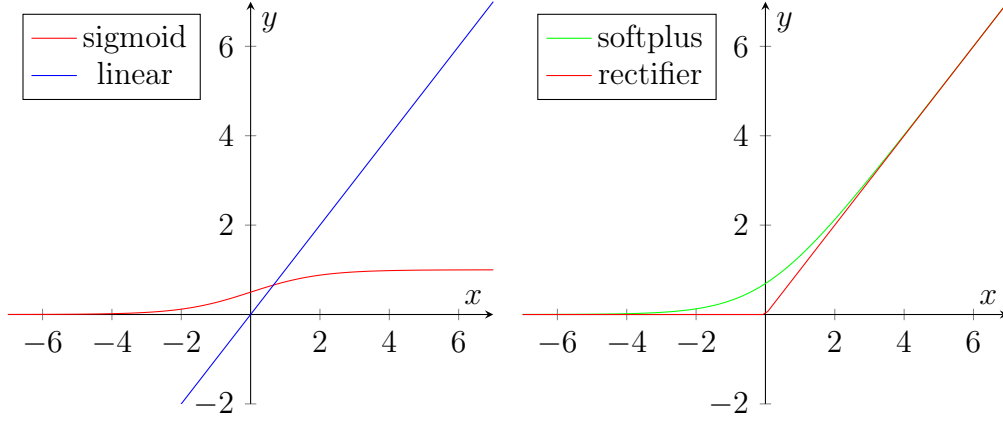


Figure 2.9: Visual aspect of the main activation functions used for an RBM. On the left, the logistic sigmoid and identity functions. On the right, the softplus and rectifier functions.

2.4.6.1 Softmax hidden unit

The softmax function is a generalization of the logistic function. It is typically used at the last layer of artificial neural networks for classification, where only one label is valid for each given input. These networks can then be trained using a variant of logistic loss regression. This was also adopted in RBM. Softmax units are a special type of units whose states are mutually constrained so that exactly one of the states has the value 1, while all others have the value 0. The state of a softmax unit is computed using the softmax function:

$$\mathbf{i} = \mathbf{c} + \mathbf{v}\mathbf{W} \quad (2.62)$$

$$\mathbf{h} = p(\mathbf{h} = 1|\mathbf{v}) = \frac{e^{\mathbf{i}_j}}{\sum_{i=1}^N e^{\mathbf{i}_j}} \quad (2.63)$$

In practice, this formula proves highly numerically instable. Therefore, a more stable version is generally used:

$$\mathbf{i} = \mathbf{c} + \mathbf{v}\mathbf{W} \quad (2.64)$$

$$m = \max_{j=1}^N \mathbf{i}_j \quad (2.65)$$

$$\mathbf{h} = p(\mathbf{h} = 1|v) = \frac{e^{\mathbf{i}-m}}{\sum_{j=1}^N e^{\mathbf{i}_j-m}} \quad (2.66)$$

The hidden units are sampled so that only one hidden unit is sampled to one while the other units are all sampled to zero:

$$\mathbf{h}'_j = \begin{cases} 1 & \text{if } \mathbf{h}_j == \max_{i=1}^N \mathbf{h}_i \\ 0 & \text{otherwise} \end{cases} \quad (2.67)$$

In practice, this type of layer is only rarely pretrained. This layer is generally only used when the network is designed for classification. Moreover, since it is the last layer of the network, it is going to receive direct gradients from the classification errors during fine-tuning. For this reason, pretraining the layer would not result in any significant advantage for fine-tuning.

2.4.6.2 Gaussian unit

As their name implies, binary visible units are best used when the input data is binary. However, it is not always possible to binarize the input data. In such cases, Gaussian visible units are better suited to handle real-valued input data. They are linear units with independent Gaussian noise. The energy functions of such an RBM can be computed as follows:

$$E(\mathbf{v}, \mathbf{h}) = \sum_{i=1}^M \frac{(\mathbf{v}_i - \mathbf{b}_i)^2}{2\sigma_i^2} - \sum_{j=1}^N c_j \mathbf{h}_j - \sum_{i=1}^M \sum_{j=1}^N \frac{\mathbf{v}_i}{\sigma_i} \mathbf{h}_j \mathbf{W}_{i,j} \quad (2.68)$$

$$F(\mathbf{v}) = \sum_{i=1}^M \frac{(c_i - \mathbf{v}_i)^2}{2\sigma_i^2} - \sum_{j=1}^N \log(1 + \exp(\sum_{i=1}^N \frac{\mathbf{v}_i}{\sigma_i} \mathbf{W}_{i,j} + \mathbf{b}_j)) \quad (2.69)$$

From this, the probability function for a visible unit becomes:

$$p_i = p(v_i | \mathbf{h}) = N(b_i + \sigma_i \sum_{j=1}^n h_j W_{i,j}, \sigma_i^2) \quad (2.70)$$

It is also necessary to divide the value of the visible units by their variance when computing the hidden units activation probabilities.

While it is possible to learn the variance of the noise for each visible unit, this is difficult with CD. In practice, Gaussian visible units are used on data that has been normalized to have zero mean and unit variance. This allows the reconstruction to be *noise-free*:

$$\mathbf{v} = p(\mathbf{v} = 1 | \mathbf{h}) = \mathbf{b} + \mathbf{W} * \mathbf{h} \quad (2.71)$$

$$\mathbf{v}' = \mathbf{v} + N(0, 1) \quad (2.72)$$

And it simplifies the energy function:

$$E(\mathbf{v}, \mathbf{h}) = \sum_{i=1}^M \frac{(\mathbf{v}_i - \mathbf{b}_i)^2}{2} + \sum_{j=1}^N \mathbf{c}_j \mathbf{h}_j - \sum_{i=1}^M \sum_{j=1}^N \mathbf{v}_i \mathbf{h}_j \mathbf{W}_{i,j} \quad (2.73)$$

A Gaussian-Binary RBM (also called a Gaussian-Bernoulli RBM) needs a learning rate that is one or two orders of magnitude lower than a Binary-Binary RBM. Moreover, the learning needs more epochs to converge and is less stable.

Gaussian hidden units are also possible, but have very few usages in practice. Indeed, they are very complicated and slow to learn and require extensive optimization to have a good model. Moreover, Gaussian hidden units do not offer significant improvement when compared with binary units or Rectified Linear Units (see Section 2.4.6.3). When both hidden and visible units are Gaussian, the energy of the RBM becomes:

$$E(\mathbf{v}, \mathbf{h}) = \sum_{i=1}^M \frac{(\mathbf{v}_i - \mathbf{b}_i)^2}{2} + \sum_{j=1}^N \frac{(\mathbf{h}_j - \mathbf{c}_j)^2}{2} - \sum_{i=1}^M \sum_{j=1}^N \mathbf{v}_i \mathbf{h}_j \mathbf{W}_{i,j} \quad (2.74)$$

2.4.6.3 Rectified Linear Unit

Rectified Linear Units (ReLUs) have demonstrated to be able to learn better features than binary units (Nair and G. E. Hinton, 2010). Moreover, they are capable of preserving more information about the input distribution. Since their introduction for RBM, they have been widely adopted for other deep learning models (Matthew D. Zeiler, Ranzato, et al., 2013; G. E. Dahl, Sainath, and G. E. Hinton, 2013; Krizhevsky, Sutskever, and G. E. Hinton, 2012) and have achieved very good performance.

Originally, the activation function of Rectified Linear Units has been defined using the softplus function:

$$f(x) = \ln(1 + e^x) \quad (2.75)$$

In practice, it is generally approximated as the max function (or hard max):

$$f(x) = \max(x, 0) = \begin{cases} x & : \text{if } x \geq 0 \\ 0 & : \text{if } x < 0 \end{cases} \quad (2.76)$$

The difference between the softplus function and the approximation can be observed in Figure 2.9. When using binary units, with the logistic sigmoid function, the gradient vanishes as we increase the input x . This is not the case with ReLU function. Moreover, when using the hard max approximation, the activation probabilities are sparse after passing through the ReLU function. Finally, this function is much faster to compute than the sigmoid function which requires exponentiation and division.

There exist several variants of ReLU functions (B. Xu et al., 2015). The main problem with ReLU is that a unit can irreversibly "die" (be turned off). If a large gradient causes the neuron never to activate again, the gradients for this neuron will always be zero from this point, causing the neuron to "die". To alleviate this issue, Leaky ReLUs were introduced with a small change to the function:

$$f(x) = \max(x, 0) = \begin{cases} x & : \text{if } x \geq 0 \\ ax & : \text{if } x < 0 \end{cases} \quad (2.77)$$

where a is set to a small constant in $[0, 1]$. In some cases, this produces better results than standard ReLU by not letting die some units. In order to avoid having to find a value for a , it is possible to learn this value during training, in a model called Parametric ReLU (He et al., 2015). Another improvement for Leaky ReLU is to randomize the a constant during training, using a uniform distribution between l and u , $\frac{1}{\text{Unif}(l,u)}$ (called Randomized Leaky ReLU). The most used distribution for this model is $\text{Unif}(3, 8)$ ($l = 3, u = 8$) (He et al., 2015). During testing, a is set to $\frac{2}{l+u}$. In practice, a random a seems the best alternative, by avoiding dying neurons and by improving control over overfitting.

In an RBM model, ReLU activation probabilities and samples can be calculated as follows:

$$\mathbf{i} = \mathbf{b} + \mathbf{v} * \mathbf{W} \quad (2.78)$$

$$\mathbf{h} = p(\mathbf{h} = 1 | \mathbf{v}) = \max(\mathbf{i}, 0) \quad (2.79)$$

$$\mathbf{h}' = \max(\mathbf{i} + N(0, \sigma(\mathbf{i})), 0) \quad (2.80)$$

Sometimes the variance of the Normal distribution in Equation 2.80 is simply set to 1 directly.

In some cases, ReLU can also be capped to a maximum value of U (called RELU-U here) (Krizhevsky, 2010). This encourages the RBM model to learn sparse features earlier during training and is computationally more efficient than other sparsity regularization methods. RELU-U units are computed in the following manner:

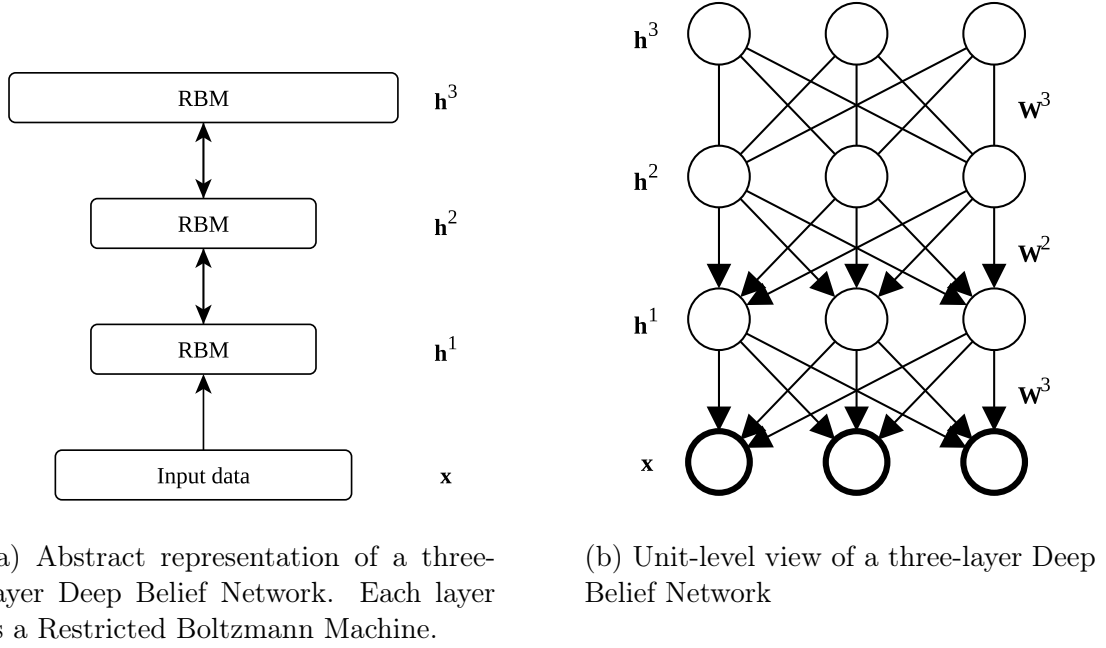


Figure 2.10: Visual representation of a Deep Belief Network

$$\mathbf{i} = \mathbf{b} + \mathbf{v} * \mathbf{W} \quad (2.81)$$

$$\mathbf{h} = p(\mathbf{h} = 1|\mathbf{v}) = \min(\max(\mathbf{i}, 0), U) \quad (2.82)$$

$$\sigma(\mathbf{i}) = \begin{cases} 1 & : \text{if } x = 0 \text{ or } x = U \\ 0 & : \text{if } 0 < x < U \end{cases} \quad (2.83)$$

$$\mathbf{h}' = \min(\max(\mathbf{i} + N(0, \sigma(\mathbf{i})), 0), U) \quad (2.84)$$

ReLU-1 could typically be used in the first layer of a network to avoid a large network simply memoizing the inputs. The units of the following layers could then be capped to higher values, such as ReLU-6, to enforce sparsity.

Although the ReLU function could theoretically be used in an RBM for visible units as well, it is generally not the case. Indeed, Gaussian visible units are a better representations for standard input patterns than ReLU. Therefore, a good model is to use Gaussian visible units for inputs that cannot be binarized and RELU hidden units.

2.5 Deep Belief Network

While an RBM may be able to learn features from simple input data, it is limited in what it can represent. For this reason, they are generally stacked together in order to form a higher level network. A DBN is a generative model consisting of

Algorithm 2.3 Deep Belief Network training algorithm

```

S = samples from training set
I = S
for all layer  $l \in$  network do
    train  $l$  with I
    save weights of  $l$ 
    I = features of  $l$  computed from I
end for
if fine tuning then
    L = labels from training set
    Fine-tune network with I and L
end if

```

several layers. Each layer of a DBN is an RBM. Although this is not covered in this thesis, it is also possible to stack any kind of auto-encoder as the basic block of the DBN. Figure 2.10 presents a DBN with three layers. This structure was proposed in (G. E. Hinton and Salakhutdinov, 2006) with an efficient learning algorithm. A DBN is first pretrained, layer by layer, using CD on each RBM layer. This is a fast procedure since each layer is trained one after another, freezing the weights after the training of a given layer. This procedure is detailed in Algorithm 2.3.

The idea behind this greedy layer-wise pretraining algorithm is that each model in the sequence of layers is learning from a different representation of the input. Each model performs a non-linear transformation and produces the input of the next layer. If the algorithm makes changes to the weights of the higher level, it is guaranteed to improve the generative model (G. E. Hinton, Osindero, and Teh, 2006). This guarantee only holds if the general Boltzmann Machine algorithm is used, which is not the case in practice where CD is used. However, adding extra layers is still guaranteed to improve imperfect models if each layer is learned long enough. This does help the following supervised fine-tuning procedure by restricting the parameters to particular regions in the model space. As for feature extraction, a DBN can learn features that are relevant to the input in that they are able to reconstruct the input using these features.

If the network is trained for classification, once each layer has been pretrained, it can be trained (fine-tuned) similarly to standard neural networks, using a back-propagation algorithm such as SGD or CG. In that case, it works exactly in the same way as a Deep Neural Network. The unsupervised pretraining acts as an initialization of the weights that helps the network generalize and prevents standard backpropagation issues such as being stuck in a local minima or the vanishing gradient problem. This better initialization allows for fast convergence and generally requires less refinements of the fine-tuning step. This is the first contribution of the restarted Deep Learning models. This model achieved state of the art for the MNIST database (LeCun, Cortes, and Burges, 1998). It is also possible to train the last layer, with softmax units, directly with CD by integrating the labels as inputs of the last layer and not perform any backpropagation. While this produces

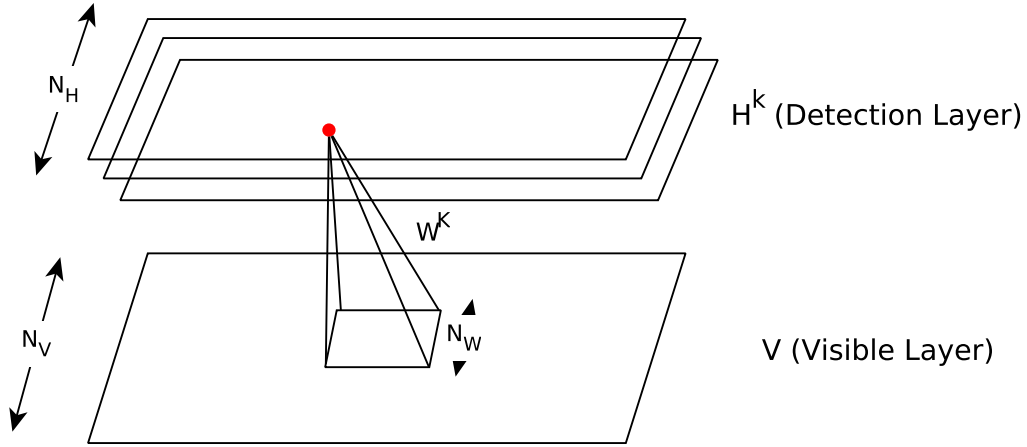


Figure 2.11: Visual representation of a Convolutional Restricted Boltzmann Machine.

good results, this does not perform as well as fine tuning the model since only the last layer is adapted.

2.6 Convolutional Restricted Boltzman Machine

While RBM and DBN models have proved very successful in solving several problems, they are not well-adapted for natural images modeling. Indeed, RBMs are known to have issues in capturing local dependencies between pixels (Taylor and G. E. Hinton, 2009). Moreover, since they are fully-connected models, they are not able to scale to large input samples, such as realistically-sized natural images. This comes from two reasons. First, realistic natural images are often composed of a very large number of pixels, making training very slow. For large images, RBM becomes computationally intractable. Moreover, objects being able to appear anywhere inside an image, translation invariance becomes an important characteristic of a training model (max pooling also helps in that matter). RBMs share the same limitations as standard fully-connected neural networks and do not cope well with local translations in the image, unless various transformations are performed (Sohn and H. Lee, 2012). For these reasons, translation invariance was integrated into RBM using convolution as a solution, resulting in the CRBM model (H. Lee, Grosse, et al., 2009). While RBM is the model equivalent to a fully-connected network, the CRBM is the equivalent of a CNN. The model learns feature detectors shared among all locations of an image, following previous work (LeCun,

Boser, et al., 1989; Grosse, Kwong, and Ng, 2007). These feature detectors will learn to detect local features occurring at any position in the image. This has the advantage that the features are invariant to translations of the input image and that the model is able to learn local dependencies between pixels, resulting in much better performance for natural images. Moreover, since each output neuron is only connected to a subset of the input, there are much less weights to learn and the network is easily able to scale to large images.

A CRBM remains very similar to an RBM but the activation of units is done using convolution rather than multiplication and weights are shared for all locations of the image. As shown in Figure 2.11, a CRBM is made of two layers, one visible layer \mathbf{V} and one hidden layer \mathbf{H} . The input layer (or visible) \mathbf{V} is a matrix of $C \times N_V^1 \times N_V^2$ visible units. C is the number of channels in the input, typically the number of color channels in color images for the first layer of the network. When CRBMs are stacked together, the following layers have a number of channels corresponding to the number of filters of the previous layer. The hidden layer \mathbf{H} consists of K groups (or bases) and each group is made of $N_H^1 \times N_H^2$ hidden units (for a total of $KN_H^1 N_H^2$ hidden units). Each of the K bases is associated with a $N_W^1 \times N_W^2$ convolutional filter (by convolutional properties $N_W^n \triangleq N_V^n - N_H^n + 1$). Here, we consider the basic case of convolution with unit-stride and no zero-padding, but the model can easily be extended to support non-unit strides and zero-padding. Since the matrix of weights \mathbf{W} is not shared between channels, there are K groups for each channel and the filters for each channel are independent, leading to a total number of weights of $CKN_W^1 N_W^2$. Each hidden base has a bias \mathbf{b}_k and each channel has a bias \mathbf{c}_c . Several models for the biases have been used, for instance (Krizhevsky, 2010) does not share biases between filters (one bias per position in the filter).

While CRBMs are generally used for two-dimensional inputs, it is also completely possible to adapt them for one-dimensional input such as speech (H. Lee, Pham, et al., 2009). In this case, all the structures have one less dimension and the one-dimensional convolution is used to compute the activation probabilities of the units. The case of color images is easily handled by a CRBM. In this case, there is a set of weights for each input channel and the results of the convolution is averaged over each channel to generate the hidden activation probabilities. The exact same procedure is used when CRBMs are stacked since the second layer will obtain a K -dimensional input from the previous layer. While rarely done in practice, it is also possible to not average over the input channels. In that particular case, there will be KC output feature maps. Moreover, the model can also be adapted to use higher levels of convolution, such as a real three-dimensional convolution (e.g. a spatial convolution over volumes).

Using the valid convolution, the energy function of a network with binary visible and hidden units can be defined as:

$$\begin{aligned}
E(\mathbf{v}, \mathbf{h}) = & - \sum_{c=1}^C \sum_{k=1}^K \sum_{i=1}^{N_H^1} \sum_{j=1}^{N_H^2} (\text{Tr}(\mathbf{h}_{i,j}^k{}^T (\tilde{\mathbf{W}}_c^k \bullet_v \mathbf{v}_c)_{i,j})) \\
& - \sum_{k=1}^K \sum_{i=1}^{N_H^1} \sum_{j=1}^{N_H^2} (\mathbf{b}_k \mathbf{h}_{i,j}^k) \\
& - \sum_{c=1}^C \sum_{i=1}^{N_V^1} \sum_{j=1}^{N_V^2} (\mathbf{c}_c \mathbf{v}_{c,i,j})
\end{aligned} \tag{2.85}$$

Using the valid convolution for the hidden units and the full convolution for the visible units, the activation functions for binary hidden units can be defined as follows:

$$\mathbf{h}_{i,j}^k = p(\mathbf{h}_{i,j}^k = 1 | \mathbf{v}) = \sigma\left(\left(\sum_{c=1}^C \tilde{\mathbf{W}}_c^k \bullet_v \mathbf{v}_c\right)_{i,j} + \mathbf{b}_k\right) \tag{2.86}$$

$$\mathbf{v}_{i,j}^c = p(\mathbf{v}_{i,j}^c = 1 | \mathbf{h}) = \sigma\left(\left(\sum_{k=1}^K \mathbf{W}_c^k \bullet_f \mathbf{h}^k\right)_{i,j} + \mathbf{c}_c\right) \tag{2.87}$$

The sampling function remains the same as for the RBM. It is also straightforward to adapt the formulas for other types of units since they all receive the same input.

A CRBM is trained similarly to an RBM, with an adapted version of the formulas to compute the gradients of the positive and negative phases (See Algorithm 2.1):

$$\mathbf{W}_{c,k}^{pos} = \mathbf{v}_c^0 \bullet_v \tilde{\mathbf{h}}_k^0 \tag{2.88}$$

$$\mathbf{W}_{c,k}^{neg} = \mathbf{v}_c^1 \bullet_v \tilde{\mathbf{h}}_k^1 \tag{2.89}$$

2.6.1 Convolutional Sparsity regularization

Although sparsity is important for RBM (see Section 2.4.3.1), it becomes much more important for CRBM. Indeed, the model is highly overcomplete, because the size of the output representation is in fact larger than the size of the input representation. Unless the size of the filters is very large, which does not generally happen in practice, the model is overcomplete almost by a factor K . In practice, overcomplete models have a tendency of learning trivial solutions, such as the identity function, instead of learning generic feature detectors. While this can achieve a good reconstruction error rate, this kind of feature is not useful for passing to another classifier and is highly tied to the training set. The most common solution to avoid this issue is to force hidden units to be rarely active,

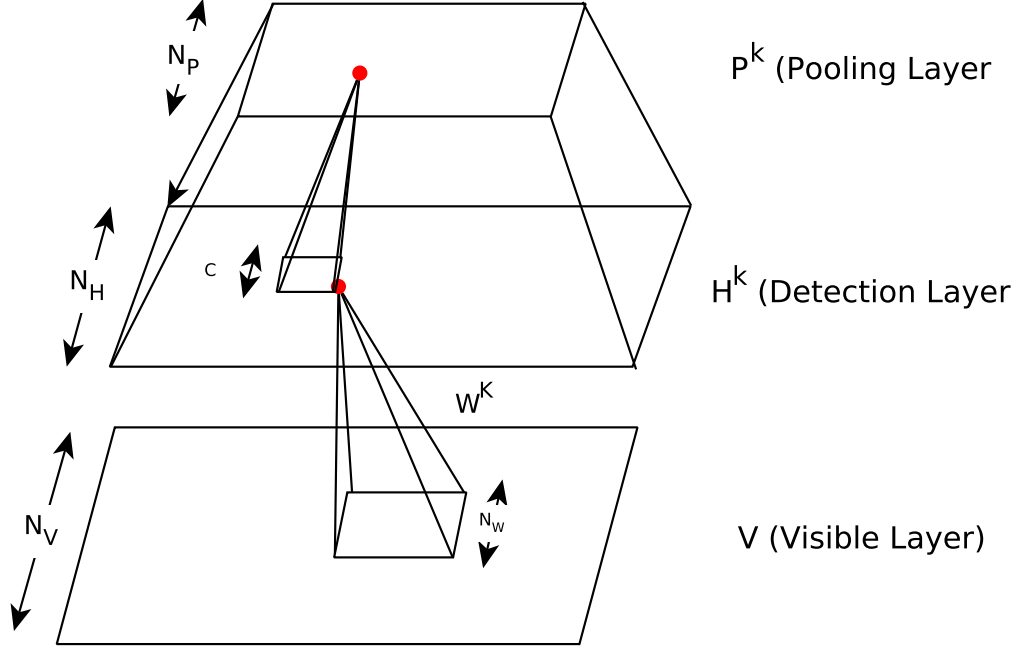


Figure 2.12: Visual representation of Probabilistic Max Pooling applied to a Convolutional Restricted Boltzmann Machine.

therefore a given stimulus in the input would only activate a small fraction of the hidden units (Olshausen and Field, 1996). Another solution is to use ReLU units that are by themselves sparser than binary hidden units (See Section 2.4.6.3).

The most common approach for sparsity regularization in a CRBM is slightly different from the general approach for an RBM (H. Lee, Ekanadham, and Ng, 2008). A penalty is only applied to the hidden biases:

$$\mathbf{b}_k = \mathbf{s}_k - p \quad (2.90)$$

$$\mathbf{z}_k = \mathbf{b}_k * \omega \quad (2.91)$$

Where p is a target sparsity and ω is a sparsity learning rate. It is important that both the learning rate of the weights and the learning rate of the sparsity are tuned accordingly so that both reconstruction and sparsity can be learned effectively.

2.6.2 Probabilistic Max Pooling

Generally, higher levels of a neural network encode information about progressively larger input regions. CNNs use pooling layers to shrink the representation

by a given factor (Y.-L. Boureau, Ponce, and LeCun, 2010). Specifically, max pooling computes the maximum activation of all the units in a given region of the feature map and Average Pooling computes the average activation of the region. In practice, pooling is done over small regions.

The aforementioned pooling operators are intended for feed-forward networks. CRBM is, by nature, a generative model, supporting both top-down and bottom-up inference. Using standard pooling layers would prevent top-down inference. Therefore, the Probabilistic Max Pooling (PMP) operator was designed so that inference is based on a max-pooling behaviour (H. Lee, Grosse, et al., 2009). This operator shrinks each dimension of the representation by a factor C . Using this operator improves translation-invariance, reduces the computational cost (by reducing the size of inputs for the following layers) and also reduces the size of the output feature maps, which may be critical for feature extraction.

A CRBM with PMP is equivalent to a CRBM but has a third layer, the pooling layer \mathbf{P} , consisting of K groups of $N_P^1 \times N_P^2$ units ($N_P^n \triangleq N_H^n/C$). Figure 2.12 presents a graphical representation of a CRBM using Probabilistic Max Pooling. The energy function of this model can be defined as follows:

$$\begin{aligned}
 E(\mathbf{v}, \mathbf{h}) = & - \sum_{c=1}^C \sum_{k=1}^K \sum_{i=1}^{N_H^1} \sum_{j=1}^{N_H^2} (\mathbf{h}_{i,j}^k (\tilde{\mathbf{W}}_c^k \bullet_v \mathbf{v}_c)_{i,j}) \\
 & - \sum_{k=1}^K \sum_{i=1}^{N_H^1} \sum_{j=1}^{N_H^2} (\mathbf{b}_k \mathbf{h}_{i,j}^k) \\
 & - \sum_{c=1}^C \sum_{i=1}^{N_V^1} \sum_{j=1}^{N_V^2} (\mathbf{c}_c \mathbf{v}_{c,i,j})
 \end{aligned} \tag{2.92}$$

For binary visible and hidden units, the activation probabilities of each unit can be computed using the following formulas:

$$\mathbf{v} = p(\mathbf{v}_{c,i,j} = 1 | \mathbf{h}) = \sigma(\mathbf{c}_c + \sum_k (\mathbf{W}_c^k \bullet_f \mathbf{h}^k)_{i,j}) \tag{2.93}$$

$$B_\alpha \triangleq (i, j) : h_{i,j} \text{ belongs to block } \alpha \tag{2.94}$$

$$\mathbf{I}(h_{i,j}^k) \triangleq \mathbf{b}_k + (\tilde{\mathbf{W}}^k \bullet_v \mathbf{v})_{i,j} \tag{2.95}$$

$$\mathbf{h} = p(\mathbf{h}_{i,j}^k = 1 | \mathbf{v}) = \frac{\exp(\mathbf{I}(h_{i,j}^k))}{1 + \sum_{i',j' \in \beta_\alpha} \exp(\mathbf{I}(h_{i',j'}^k))} \tag{2.96}$$

$$\mathbf{P} = p(\mathbf{p}_\alpha^k = 0 | \mathbf{v}) = \frac{1}{1 + \sum_{i',j' \in \beta_\alpha} \exp(\mathbf{I}(h_{i',j'}^k))} \tag{2.97}$$

The detection units in a block B_α are connected to the pooling unit p_α in a single potential. This has the effect that only at most one detection unit may be

activated. The pooling unit is activated only if a detection unit of its group is activated. This means that each block can take on $C^2 + 1$ different values.

This special CRBM is especially meant to work with binary hidden units since the pooling units are meant to be binary as well. On the other hand, the visible units can be adapted to other units such as Gaussian units.

2.7 Convolutional Deep Belief Network

Such as the DBN is a network where RBM are stacked, the Convolutional Deep Belief Network (CDBN) model (H. Lee, Grosse, et al., 2009) is a network composed of several CRBM one after another. It is generally pretrained by greedily training each CRBM in an unsupervised manner to initialize the weights of the networks. It can then be fine-tuned or used for feature extraction. In practice, this model is more often used solely for feature extraction rather than for initializing the weights of a neural network as was the original purpose of DBN. It is nevertheless possible to fine-tune it in the same way as a CNN (Krizhevsky, 2010). It is typical to use either Probabilistic Max Pooling (PMP) or regular max pooling layers between the layers to improve the robustness of the learned features and control overfitting. Using standard max pooling layers between CRBM has the disadvantage of losing the ability of doing both bottom-up and top-down inference. However, it has the advantage of being easy to fine-tune using standard backpropagation algorithms.

Figure 2.13 shows the features learned by a three-layer CDBN with PMP (H. Lee, Grosse, et al., 2009) on images from the Caltech-101 data set. It can clearly be observed that the first layer learned filters for oriented edges while the second one learned object parts. Finally, the last layer acts as a detector of higher-level features, almost complete objects.

2.8 Variants of Restricted Boltzmann Machine

The standard RBM and the CRBM are the most commonly used types of RBM. However, several other variations of RBM have been developed. Most models were developed to solve a task that is difficult to solve with the RBM. The principal goal is to perform better than the Gaussian-Bernoulli RBM for real-valued inputs, such as modeling natural images or time series. These models are less general-purpose than the standard RBM and as such have less applicability. However, they are typically exhibiting better performance for very specific tasks. In practice, with the exception of the Deep Boltzmann Machine (See Section 2.8.5), these models have been much less used than the RBM or CRBM models, which are more simple and more generic models.

This Section briefly describes some of the popular variants of RBM and their possible applications. Many more variations were developed, however this Section is not meant to be exhaustive.

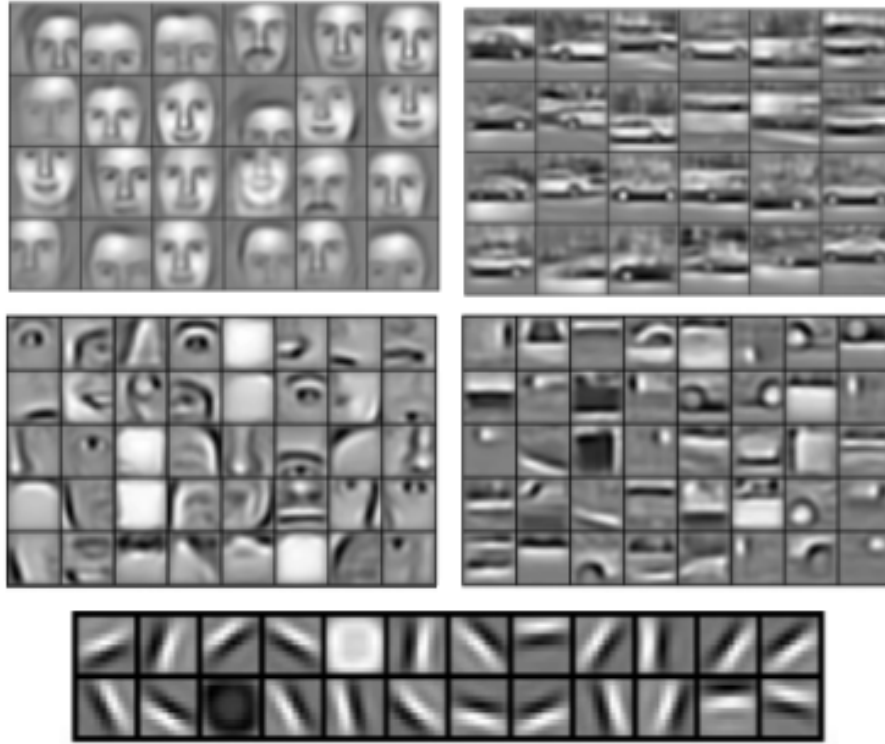


Figure 2.13: Features learned by a three-level CDBN with Probabilistic Max Pooling on two different Caltech 101 categories. From (H. Lee, Grosse, et al., 2009).

2.8.1 Mean-Covariance RBM

A mean-covariance RBM (mcRBM) (Ranzato and G. E. Hinton, 2010; Ranzato, Krizhevsky, et al., 2010; G. Dahl et al., 2010) is an alternative to the Gaussian-Bernoulli RBM to handle real-valued inputs. It is typically used as the first layer of a DBN. This model has two separate groups of hidden units: the mean units and the precision units (or covariance units). The mean units are similar to the units of a Gaussian-Bernoulli RBM. What makes this model interesting is the precision units. They are enforcing smoothness constraints in the input data. When a constraint is violated, the precision unit is turned off. At any time, the set of turned-on precision units specifies a covariance matrix specific to the sample. When put together, each image is represented using a set of binary latent features modeling the mean and another set modeling covariance. This model has been successfully applied to both natural images and acoustic data.

2.8.2 Discriminative RBM

Typically, RBM are used as feature extractors for higher level classifiers or to initialize the weights of a feed-forward neural network. To achieve classification, the RBM can be used to model the joint distribution of the inputs x and the associated classes y . The model can then be trained using standard CD algorithm.

However, this will train the model in having a good $p(x, y)$ approximation. Instead, a Discriminative RBM (DRBM) (Larochelle and Bengio, 2008) changes the training objective so that it optimizes $p(y|x)$ instead. Using this new objective, the DRBM can be trained using CD. Moreover, this model can also be used as a feature extractor, enforcing the features to be more discriminative than if they were generated using a standard RBM model.

2.8.3 Temporal RBM

The Temporal RBM (TRBM) (Taylor, G. E. Hinton, and Roweis, 2006; Sutskever and G. E. Hinton, 2007) is a sequence of RBMs that are arranged so that the state of the RBM at a given step only depends on the state of the RBM in the previous timestep. Learning can be achieved using CD, however exact inference is intractable and so inference is realized using a heuristic approximation. The model was later extended to the Recurrent Temporal RBM (Sutskever, G. E. Hinton, and Taylor, 2009). While being very similar to a TRBM, this variant makes exact inference possible and improves the computation of the gradient. For instance, these models were able to generate videos of balls bouncing. The TRBM was also described in similar ways as the Conditional RBM (Taylor and G. E. Hinton, 2009) (Mnih, Larochelle, and G. E. Hinton, 2012).

2.8.4 Spike and Slab RBM

The *spike and slab* RBM (ssRBM) (A. C. Courville, Bergstra, and Bengio, 2011) differs from a Gaussian-Bernoulli RBM in that each hidden unit is associated with a binary *spike* variable and a real value *slab* vector of K features. Where a Gaussian-Bernoulli RBM model continuous input with only binary latent variables, this model uses both binary and real variables to represent the latent features. It can be trained efficiently using a slightly different version of CD. This model proved to generate very potent filters on whitened color images and was able to achieve very good classification results when coupled with logistic regression.

2.8.5 Deep Boltzmann Machine

A DBN (See Section 2.5) is merely a stack of RBM, trained in a layer-by-layer way and not a multilayer Boltzmann Machine (G. E. Hinton and Salakhutdinov, 2006). A Deep Boltzmann Machine (DBM) (Salakhutdinov and G. E. Hinton, 2009; G. E. Hinton and Salakhutdinov, 2012) has undirected connections between all layers, contrary to a DBN in which only the connections between the top two layers are undirected and the other connections are top-to-bottom directed. This can be seen on Figure 2.14. Contrary to a DBN, the inference can also perform top-down feedback, after the typical bottom-up pass. This allows the DBM to better propagate uncertainty.

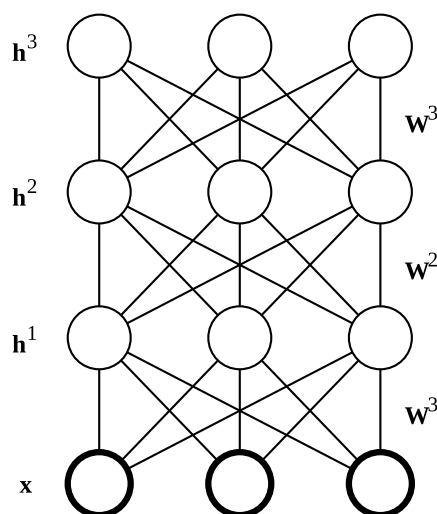


Figure 2.14: Three-layer Deep Boltzmann Machine

A DBM is also greedily pretrained layer-by-layer, but in such a way that allows top-down feedback to be achieved and dispatching more work to the higher levels of the network by configuring the tied weights. This means that it is not enough to train each layer as an RBM and then copy the pretrained weights into the network. For this, there are several techniques (I. Goodfellow, Bengio, and A. Courville, 2016). Once pretrained, its weights can also be used to initialize the weights of a neural network that can then be fine-tuned using standard techniques. More interestingly and contrary to DBNs, it can also be generatively fine-tuned and proved to be more efficient than a DBN in estimating the probability distribution over the input samples (G. E. Hinton and Salakhutdinov, 2012).

References for Chapter 2

- Ackley, David H, Geoffrey E. Hinton, and Terrence J. Sejnowski (1985). “A learning algorithm for boltzmann machines”. In: *Cognitive science* 9.1, pp. 147–169 (cit. on p. 20).
- Bertsekas, Dimitri P. (1999). *Nonlinear programming* (cit. on p. 29).
- Bishop, Christopher M. (2006). *Pattern recognition and Machine Learning*. Springer (cit. on p. 12).
- Blue, James L. and Patrick J. Grother (1992). “Training feed-forward neural networks using conjugate gradients”. In: *SPIE/IS&T 1992 Symposium on Electronic Imaging: Science and Technology*. International Society for Optics and Photonics, pp. 179–190 (cit. on p. 19).
- Boureau, Y-Lan, Jean Ponce, and Yann LeCun (2010). “A theoretical analysis of feature pooling in visual recognition”. In: *Proceedings of the Int. Conf. on Machine Learning*, pp. 111–118 (cit. on p. 47).

- Byrd, Richard H. et al. (1995). “A limited memory algorithm for bound constrained optimization”. In: *SIAM Journal on Scientific Computing* 16.5, pp. 1190–1208 (cit. on p. 19).
- Courville, Aaron C., James Bergstra, and Yoshua Bengio (2011). “A spike and slab restricted Boltzmann machine”. In: *International Conference on Artificial Intelligence and Statistics*, pp. 233–241 (cit. on p. 50).
- Dahl, George E., Tara N. Sainath, and Geoffrey E. Hinton (2013). “Improving deep neural networks for LVCSR using rectified linear units and dropout”. In: *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*. IEEE, pp. 8609–8613 (cit. on p. 39).
- Dahl, George et al. (2010). “Phone recognition with the mean-covariance restricted Boltzmann machine”. In: *Advances in neural information processing systems*, pp. 469–477 (cit. on p. 49).
- Erhan, Dumitru, Yoshua Bengio, Aaron Courville, and Pascal Vincent (2009). “Visualizing higher-layer features of a deep network”. In: *University of Montreal* 1341 (cit. on p. 35).
- Freund, Yoav and David Haussler (1994). *Unsupervised learning of distributions of binary vectors using two layer networks*. Computer Research Laboratory [University of California, Santa Cruz] (cit. on p. 26).
- Fukushima, Kunihiko (1980). “Neocognitron: A Self-Organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position”. In: *Biological Cybernetics* 36, pp. 193–202 (cit. on p. 15).
- Goodfellow, Ian, Yoshua Bengio, and Aaron Courville (2016). “Deep Learning”. Book in preparation for MIT Press. URL: <http://www.deeplearningbook.org> (cit. on p. 51).
- Graves, Alex et al. (2009). “A Novel Connectionist System for Unconstrained Handwriting Recognition”. In: *IEEE transactions on pattern analysis and machine intelligence* 31.5, pp. 855–868 (cit. on p. 16).
- Grosse, Roger, Helen Kwong, and Andrew Y. Ng (2007). “Shift-invariant sparse coding for audio classification”. In: *Proceedings of the Twenty-third Conference on Uncertainty in Artificial Intelligence* (cit. on p. 44).
- Hagan, Martin T. and Mohammad B. Menhaj (1994). “Training feedforward networks with the Marquardt algorithm”. In: *Neural Networks, IEEE Transactions on* 5.6, pp. 989–993 (cit. on p. 19).
- He, Kaiming et al. (2015). “Delving deep into rectifiers: Surpassing human-level performance on imagenet classification”. In: *Proceedings of the IEEE International Conference on Computer Vision*, pp. 1026–1034 (cit. on p. 40).
- (2016). “Deep residual learning for image recognition”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 770–778 (cit. on p. 20).
- Hinton, Geoffrey E. (2002). “Training products of experts by minimizing contrastive divergence”. In: *Neural computation* 14.8, pp. 1771–1800 (cit. on pp. 20, 26).
- (2007). “Learning multiple layers of representation”. In: *Trends in Cognitive Sciences* 11, pp. 428–434 (cit. on p. 19).

- (2012). “A Practical Guide to Training Restricted Boltzmann Machines.” In: *Neural Networks: Tricks of the Trade (2nd ed.)* Ed. by Grégoire Montavon, Genevieve B. Orr, and Klaus-Robert Müller. Vol. 7700. Lecture Notes in Computer Science. Springer, pp. 599–619. ISBN: 978-3-642-35288-1. URL: <http://dblp.uni-trier.de/db/series/lncs/lncs7700.html#Hinton12> (cit. on pp. 29, 68, 73, 93).
- Hinton, Geoffrey E., Simon Osindero, and Yee-Whye Teh (2006). “A fast learning algorithm for deep belief nets”. In: *Neural computation* 18.7, pp. 1527–1554 (cit. on pp. 19, 26, 42, 98).
- Hinton, Geoffrey E. and Ruslan R. Salakhutdinov (2006). “Reducing the dimensionality of data with neural networks”. In: *Science* 313.5786, pp. 504–507 (cit. on pp. 19, 42, 50, 57, 86).
- (2012). “A better way to pretrain deep boltzmann machines”. In: *Advances in Neural Information Processing Systems*, pp. 2447–2455 (cit. on pp. 50, 51).
- Hinton, Geoffrey E. and Terrence J. Sejnowski (1986). “Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1”. In: ed. by David E. Rumelhart, James L. McClelland, and CORPORATE PDP Research Group. Cambridge, MA, USA: MIT Press. Chap. Learning and Relearning in Boltzmann Machines, pp. 282–317. ISBN: 0-262-68053-X. URL: <http://dl.acm.org/citation.cfm?id=104279.104291> (cit. on p. 25).
- Hochreiter, Sepp (1991). *Untersuchungen zu dynamischen neuronalen Netzen. Diploma thesis, Institut für Informatik, Lehrstuhl Prof. Brauer, Technische Universität München* (cit. on pp. 19, 60).
- Hochreiter, Sepp and Jürgen Schmidhuber (1997). “Long Short-Term Memory”. In: *Neural Comput.* 9.8, pp. 1735–1780. ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735. URL: <http://dx.doi.org/10.1162/neco.1997.9.8.1735> (cit. on p. 16).
- Hopfield, John J. (1982). “Neural networks and physical systems with emergent collective computational abilities”. In: *Proceedings of the national academy of sciences* 79.8, pp. 2554–2558 (cit. on p. 21).
- Hornik, Kurt, Maxwell Stinchcombe, and Halbert White (1989). “Multilayer feed-forward networks are universal approximators”. In: *Neural networks* 2.5, pp. 359–366 (cit. on p. 15).
- Hubel, David H. and Torsten N. Wiesel (1962). “Receptive fields, binocular interaction and functional architecture in the cat’s visual cortex”. In: *The Journal of physiology*, pp. 106–154. URL: <http://www.ncbi.nlm.nih.gov/pmc/articles/PMC1359523/pdf/jphysiol01247-0121.pdf> (cit. on p. 17).
- Ioffe, Sergey and Christian Szegedy (2015). “Batch normalization: Accelerating deep network training by reducing internal covariate shift”. In: *arXiv preprint arXiv:1502.03167* (cit. on pp. 20, 62).
- Jarrett, Kevin, Koray Kavukcuoglu, Yann Lecun, et al. (2009). “What is the best multi-stage architecture for object recognition?” In: *2009 IEEE 12th International Conference on Computer Vision*. IEEE, pp. 2146–2153 (cit. on pp. 16, 58, 71).

- Krizhevsky, Alex (2010). *Convolutional deep belief networks on cifar-10* (cit. on pp. 40, 44, 48, 131).
- Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton (2012). “Imagenet classification with deep convolutional neural networks”. In: *Advances in neural information processing systems*, pp. 1097–1105 (cit. on pp. 39, 62).
- Kullback, Solomon and Richard A. Leibler (1951). “On information and sufficiency”. In: *The annals of mathematical statistics* 22.1, pp. 79–86 (cit. on p. 25).
- Lan, Zhenzhong et al. (2016). “The best of both worlds: Combining data-independent and data-driven approaches for action recognition”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pp. 123–132 (cit. on p. 14).
- Larochelle, Hugo and Yoshua Bengio (2008). “Classification using discriminative restricted Boltzmann machines”. In: *Proceedings of the 25th international conference on Machine learning*. ACM, pp. 536–543 (cit. on p. 50).
- LeCun, Yann, Yoshua Bengio, and Geoffrey E. Hinton (2015). “Deep learning”. In: *Nature* 521.7553, pp. 436–444 (cit. on pp. 20, 60, 63).
- LeCun, Yann, Bernhard Boser, et al. (1989). “Backpropagation Applied to Handwritten Zip Code Recognition”. In: *Neural Comput.* 1.4, pp. 541–551. ISSN: 0899-7667. DOI: 10.1162/neco.1989.1.4.541. URL: <http://dx.doi.org/10.1162/neco.1989.1.4.541> (cit. on pp. 15, 43).
- LeCun, Yann, Léon Bottou, et al. (1998). “Gradient-based learning applied to document recognition”. In: *Proceedings of the IEEE* 86.11, pp. 2278–2324. ISSN: 0018-9219. DOI: 10.1109/5.726791 (cit. on pp. 18, 87, 99, 197).
- LeCun, Yann, Corinna Cortes, and Christopher J. C. Burges (1998). *The MNIST database of handwritten digits* (cit. on pp. 34, 42).
- Lee, Honglak, Chaitanya Ekanadham, and Andrew Y. Ng (2008). “Sparse deep belief net model for visual area V2”. In: *Advances in neural information processing systems*, pp. 873–880 (cit. on pp. 46, 69, 131).
- Lee, Honglak, Roger Grosse, et al. (2009). “Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations”. In: *Proceedings of the 26th Annual International Conference on Machine Learning*. ACM, pp. 609–616 (cit. on pp. 43, 47–49, 69, 99).
- Lee, Honglak, Peter Pham, et al. (2009). “Unsupervised feature learning for audio classification using convolutional deep belief networks”. In: *Advances in neural information processing systems*, pp. 1096–1104 (cit. on p. 44).
- Majtner, Tomas, Sule Yildirim-Yayilgan, and Jon Yngve Hardeberg (2016). “Combining deep learning and hand-crafted features for skin lesion classification”. In: *Image Processing Theory Tools and Applications (IPTA), 2016 6th International Conference on*. IEEE, pp. 1–6 (cit. on p. 14).
- Minsky, Marvin and Seymour Papert (1969). *Perceptrons: An essay in computational geometry*. en. Cambridge, MA: MIT Press (cit. on p. 15).
- Mnih, Volodymyr, Hugo Larochelle, and Geoffrey E. Hinton (2012). “Conditional restricted Boltzmann machines for structured output prediction”. In: *arXiv preprint arXiv:1202.3748* (cit. on p. 50).

- Murphy, Kevin P. (2012). *Machine learning: a probabilistic perspective*. MIT press (cit. on pp. 12, 14).
- Nair, Vinod and Geoffrey E. Hinton (2009). “3D object recognition with deep belief nets”. In: *Advances in Neural Information Processing Systems*, pp. 1339–1347 (cit. on p. 31).
- (2010). “Rectified linear units improve restricted boltzmann machines”. In: *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pp. 807–814 (cit. on pp. 39, 60).
- Neal, Radford M. (2001). “Annealed importance sampling”. In: *Statistics and Computing* 11.2, pp. 125–139 (cit. on p. 36).
- Ngiam, Jiquan et al. (2011). “On optimization methods for deep learning”. In: *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pp. 265–272 (cit. on pp. 13, 19).
- Olshausen, Bruno A. and David J. Field (1996). “Emergence of simple-cell receptive field properties by learning a sparse code for natural images”. In: *Nature* 381.6583, pp. 607–609 (cit. on p. 46).
- Pinheiro, Pedro O. and Ronan Collobert (2014). “Recurrent Convolutional Neural Networks for Scene Labeling”. In: *Proceedings of the 31th International Conference on Machine Learning (ICML-14)*, pp. 82–90 (cit. on p. 16).
- Ranzato, Marc’Aurelio and Geoffrey E. Hinton (2010). “Modeling pixel means and covariances using factorized third-order Boltzmann machines”. In: *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on*. IEEE, pp. 2551–2558 (cit. on p. 49).
- Ranzato, Marc’Aurelio, Alex Krizhevsky, et al. (2010). “Factored 3-way restricted Boltzmann machines for modeling natural images”. In: *International conference on artificial intelligence and statistics*, pp. 621–628 (cit. on p. 49).
- Rosenblatt, Frank (1958). “The perceptron: a probabilistic model for information storage and organization in the brain.” In: *Psychological review* 65.6, p. 386 (cit. on p. 15).
- Sak, Hasim, Andrew W Senior, and Françoise Beaufays (2014). “Long short-term memory recurrent neural network architectures for large scale acoustic modeling.” In: *INTERSPEECH*, pp. 338–342 (cit. on p. 16).
- Salakhutdinov, Ruslan R. (2009). “Learning in Markov random fields using tempered transitions”. In: *Advances in neural information processing systems*, pp. 1598–1606 (cit. on p. 36).
- Salakhutdinov, Ruslan R. and Geoffrey E. Hinton (2009). “Deep boltzmann machines”. In: *International conference on artificial intelligence and statistics*, pp. 448–455 (cit. on p. 50).
- Sargano, Allah Bux, Plamen Angelov, and Zulfiqar Habib (2017). “A comprehensive review on handcrafted and learning-based action representation approaches for human activity recognition”. In: *Applied Sciences* 7.1, p. 110 (cit. on p. 14).
- Smolensky, Paul (1986). “Information processing in dynamical systems: Foundations of harmony theory”. In: (cit. on p. 20).
- Sohn, Kihyuk and Honglak Lee (2012). “Learning Invariant Representations with Local Transformations”. In: *ICML* (cit. on p. 43).

- Srivastava, Nitish et al. (2014). “Dropout: a simple way to prevent neural networks from overfitting.” In: *Journal of Machine Learning Research* 15.1, pp. 1929–1958 (cit. on pp. 16, 20, 61).
- Sutskever, Ilya and Geoffrey E. Hinton (2007). “Learning multilevel distributed representations for high-dimensional sequences”. In: *International Conference on Artificial Intelligence and Statistics*, pp. 548–555 (cit. on p. 50).
- Sutskever, Ilya, Geoffrey E. Hinton, and Graham W. Taylor (2009). “The recurrent temporal restricted boltzmann machine”. In: *Advances in Neural Information Processing Systems*, pp. 1601–1608 (cit. on p. 50).
- Sutskever, Ilya and Tijmen Tieleman (2010). “On the Convergence Properties of Contrastive Divergence.” In: *AISTATS*. Vol. 9, pp. 789–795 (cit. on p. 26).
- Szegedy, Christian, Wei Liu, et al. (2015). “Going deeper with convolutions”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 1–9 (cit. on pp. 20, 62).
- Taylor, Graham W. and Geoffrey E. Hinton (2009). “Factored Conditional restricted Boltzmann machines for modeling motion style”. In: *Proceedings of the 26th annual international conference on machine learning*. ACM, pp. 1025–1032 (cit. on pp. 43, 50).
- Taylor, Graham W., Geoffrey E. Hinton, and Sam T. Roweis (2006). “Modeling human motion using binary latent variables”. In: *Advances in neural information processing systems*, pp. 1345–1352 (cit. on p. 50).
- Tieleman, Tijmen (2008). “Training restricted Boltzmann machines using approximations to the likelihood gradient”. In: *Proceedings of the 25th international conference on Machine learning*. ACM, pp. 1064–1071 (cit. on p. 35).
- Tieleman, Tijmen and Geoffrey E. Hinton (2009). “Using fast weights to improve persistent contrastive divergence”. In: *Proceedings of the 26th Annual International Conference on Machine Learning*. ACM, pp. 1033–1040 (cit. on p. 35).
- Werbos, Paul J. (1982). “Applications of advances in nonlinear sensitivity analysis”. In: *System modeling and optimization*. Springer, pp. 762–770 (cit. on p. 15).
- Xu, Bing et al. (2015). “Empirical evaluation of rectified activations in convolutional network”. In: *arXiv preprint arXiv:1505.00853* (cit. on p. 40).
- Yosinski, Jason and Hod Lipson (2012). “Visually debugging restricted boltzmann machine training with a 3d example”. In: *Representation Learning Workshop, 29th International Conference on Machine Learning* (cit. on p. 35).
- Zeiler, Matthew D. and Rob Fergus (2014). “Visualizing and understanding convolutional networks”. In: *European Conference on Computer Vision*. Springer, pp. 818–833 (cit. on p. 34).
- Zeiler, Matthew D., Marc’Aurelio Ranzato, et al. (2013). “On rectified linear units for speech processing”. In: *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*. IEEE, pp. 3517–3521 (cit. on p. 39).

Chapter 3

Semi-Supervised Training

*It's fine to celebrate success but it is more
important to heed the lessons of failure*

Bill Gates

Contents

3.1	Introduction	57
3.2	Advantages of pretraining	59
3.3	Advances in Neural Network training	60
3.4	Summary	62

3.1 Introduction

Semi-Supervised Training generally refers to the case when a neural network that is to be used for classification is first pretrained, layer-by-layer, using an unsupervised training algorithm. Finally, the network can be trained using a standard training algorithm, for classification or prediction. In the case of a Deep Belief Network (DBN), each layer is trained, in turn, using a Restricted Boltzmann Machine (RBM), with Contrastive Divergence (CD). This is called semi-supervised since the pretraining is done in an unsupervised manner, while the training itself (fine-tuning the network) is done in a, more classical, supervised way. In the first step, the network is trained to reconstruct the input, potentially using large quantities of data as no labels are necessary. The network is only able to perform classification once fine-tuned using labelled training data. This form of training was introduced by Hinton et al. (G. E. Hinton and Salakhutdinov, 2006). It is interesting that this is close to human learning since most of what humans learn is unsupervised, no labels are necessary for recognizing shapes for instance. The pretraining acts as a good initialization of the network weights. This helps the

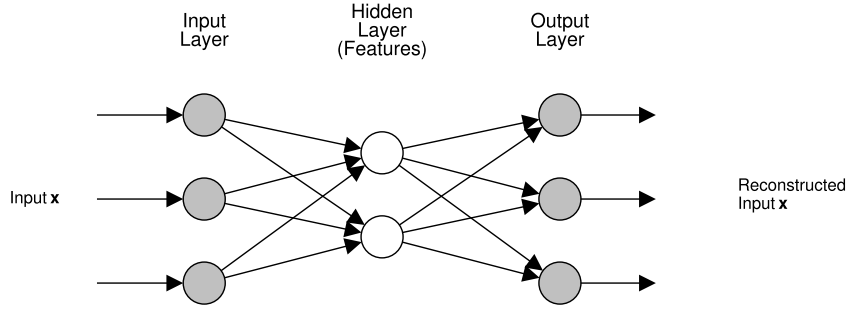


Figure 3.1: Example of an auto-encoder neural network. The network is trained to reconstruct its input \mathbf{x} , targetting to obtain a $\hat{\mathbf{x}}$ output as close as possible to the input \mathbf{x} . The activations of the hidden layer represents the features that the network tries to learn.

supervised fine-tuning to complete faster and to find better solutions. At the time of the original publication, this new technique outperformed the state of the art of Digit Recognition on the MNIST data set (considering only non-convolutional results without any data augmentation). Since then, the state of the art has evolved again (Jarrett, Kavukcuoglu, Lecun, et al., 2009; Ciregan, Meier, and Schmidhuber, 2012), but this result restarted the interest in neural networks and more specifically in larger and deeper networks, leading to what is now known as Deep Learning.

This technique is not limited to the RBM model. In practice, it is possible to pretrain a neural network using other types of auto-encoders to get a good first initialization of the weights. There exist several types of auto-encoders (Bengio, 2009). Generally, auto-encoders are a special form of Artificial Neural Network (ANN). Figure 3.1 shows an example of a simple auto-encoder ANN. The most basic auto-encoder is a network with one hidden layer in between the input and the output layer that have matching dimensions. The training is not done using labels, but it is done using the inputs as the expected output. The features generated by the hidden layer is what is kept once the network is trained. This form of architecture is sometimes called a mirror architecture since the layers around the features layers are mirrored versions of each other. Generally, the part of the network on the right of the features layer is dropped once the network is trained. Denoising Auto-Encoder (DAE) are trained not directly with the input but with a noisy (corrupted) version of them, to improve the robustness of the learned features (Vincent, Larochelle, Bengio, et al., 2008). Moreover, an auto-encoder model can also be stacked in Stacked Auto-Encoder (SAE) or in Stacked Denoising Auto-Encoder (SDAE) and convolutional variants have also been developed (Masci et al., 2011), following the same idea.

This pretraining of neural networks does not perform feature extraction per se. Indeed, once the network is fine-tuned, the features that are generated by the intermediate layers are altered to achieve the network goal. Nevertheless, the

pretraining is performed so that a better representation of the input is generated, thus performing feature extraction.

Since pretraining seems to help the training of large ANNs, some research experimented with combined unsupervised and supervised gradients during training (Zamora-Martinez, Munoz-Almaraz, and Pardo, 2016) with good success. This is performed by using a loss composed of both the supervised and unsupervised losses. The training starts with a focus on the unsupervised loss and is getting more and focuses on the supervised loss.

3.2 Advantages of pretraining

Although the results empirically show that unsupervised pretraining helps the overall training of deep neural networks for classification, it is not directly clear why this is the case. Indeed, the objectives of the two phases are not nearly the same. The first training objective tries to minimize the reconstruction error of the network. The second tries to minimize its classification error. A lot of research has been done to explain why this pretraining phase helps the fine-tuning (Erhan, Bengio, A. Courville, Manzagol, et al., 2010; Erhan, Manzagol, et al., 2009) (Bengio, Lamblin, et al., 2007; Larochelle, Bengio, et al., 2009).

The basic and reasonable explanation of why it helps is simply that it provides a good initialization of the weights, mitigating the difficulty of the optimization problem. Numerous experiments and research were performed to verify and clarify this basic explanation. The first interesting fact is that the pretraining need not necessarily been done with RBM and DBN. Indeed, it was shown, empirically, that training each layer as an auto-encoder or using another form of supervised pretraining leads to improvements when compared with a deep neural network that is not pretrained (random initialization of the weights) (Bengio, Lamblin, et al., 2007). It was shown that carefully pretraining a neural network using a layer-by-layer training with auto-encoders led to performance very close to that of the DBN model. When the model is trained as a denoising auto-encoder (Vincent, Larochelle, Bengio, et al., 2008), it performs at least as well as a DBN (Erhan, Manzagol, et al., 2009). In each case, the pretrained networks yielded significantly better performance than the network simply trained in a standard way. Moreover, the study also showed that pretrained shallow networks are able to compete against non-pretrained deep networks (Vincent, Larochelle, Bengio, et al., 2008). In a further experiment, it was shown that initializing the network in a way that meaningfully represents the input yield a better generalization once the weights are fine-tuned. More importantly, the higher layers are already initialized with a representation of higher level, whereas a standard initialization would initialize each layer independently. Moreover, it was also demonstrated that while very efficient on data in which the input and the target are highly correlated, pretraining is not adequate when there is no particular relation between input and output, which is the case in some regression problems.

More specifically, experiments have shown that the more layers a network has, the more interesting pretraining becomes (Erhan, Manzagol, et al., 2009). Pretraining also helps reducing the variance of different trained models due to the initial random initialization of the weights (before the pretraining itself). However, it is interesting to note that the classification error goes up when increasing too much the number of layers even with pretraining. When experimenting with the size of the layers, it was shown that while pretraining is very efficient for large layers, it hurts performance when working with small networks. Since pretraining acts as a regularizer method, the limited capacity of such networks does not allow further restriction. Indeed, regularized is generally more helpful when the number of parameters is large. This is one of the reasons why small softmax layers are not pretrained.

The results and conclusions have been confirmed by others (Erhan, Bengio, A. Courville, Manzagol, et al., 2010) (Larochelle, Bengio, et al., 2009). The important conclusion is that pretraining a neural network in an unsupervised way before training it for classification acts as a regularizer and provides, in the end, a better generalization.

3.3 Advances in Neural Network training

The different studies that are reported in Section 3.2 show that pretraining a deep neural network greatly improves its training and allows to reach better performance of the final fine-tuned network. However, since then, several novel neural network training techniques have been introduced. These advances allowed researchers to train deeper and deeper networks without pretraining.

The first of these improvements is the use of a new activation function, the Rectifier function. Units using this new function are called Rectified Linear Units (ReLUs). Although they were first introduced for Boltzmann machines (Nair and G. E. Hinton, 2010), ReLUs have proven an excellent activation function for general ANN training. The function is very simple:

$$f(x) = \max(0, x) \tag{3.1}$$

These units have many advantages over the logistic sigmoid or the hyperbolic tangent activation functions. First, and probably foremost, they are not exposed to the vanishing gradient problem (Hochreiter, 1991) and their gradient cannot grow too high. Moreover, since they are inherently sparse, it is not necessary to apply a sparsity regularization method to the training (see Section 2.4.3.1). Moreover, they are also more efficient to compute than most other activation functions. All these advantages made it the most used activation function in deep neural networks nowadays (LeCun, Bengio, and G. E. Hinton, 2015). One problem with this activation function is that units can "die", but there are several solutions to this problem (see Section 2.4.6.3).

New regularization techniques have also been proposed, especially to help control overfitting. One major advance was the introduction of Dropout (G. E. Hinton, Srivastava, et al., 2012; Srivastava et al., 2014; Baldi and Sadowski, 2013). During training, at each epoch, each neuron has a probability p of being dropped out of the network. In practice, $p = 0.5$ is the most used value. This means that, at each epoch, a possibly different network will be trained. Under these conditions, a neuron cannot adapt too much to other neurons (a problem known as co-adaptation). This significantly reduces overfitting, especially since this is equivalent to training a large number of smaller networks that are less subject to overfitting. It has been demonstrated that Dropout is the first order equivalent of an L2 regularization after some feature scaling (Wager, Wang, and P. S. Liang, 2013). It is important that at test time, the weights be multiplied by the dropout probability. Dropout can be generalized to DropConnect (Wan et al., 2013), in which connections between neurons are dropped with a probability $1 - p$ rather than dropping neurons directly. This has proved more efficient than Dropout on several problems. One disadvantage of Dropout is that it requires more epochs for training the complete network since only a smaller portion of the network is trained at each iteration. Moreover, while these techniques lead to very large improvements in fully-connected layers, they do not improve much the performance of convolutional layers. For this, special forms of stochastic pooling layers were introduced and were shown to improve dropout effects on training convolutional layers (Matthew D. Zeiler and Fergus, 2013; Wu and Gu, 2015). These models are using a similar form of dropping but applied to the pooling layer (Max Pooling Dropout) rather than on the convolutional layer itself. It was observed that Dropout was not helpful for small data sets and its improvements for large data sets was varying significantly from one data set size to another (Srivastava, 2013)

Not only was the training of neural networks largely improved by these techniques, but it was also greatly sped up by different techniques. While Graphical Processing Units (GPUs) were originally intended for processing graphics with a lot of small elements, their massive processing capabilities led some people to use them in a more general way, a technique known as General Purpose Graphical Processing Unit (GPGPU) (Owens et al., 2007). Central Processing Units (CPUs) are built for a very wide variety of applications and are fast for executing one task. However, due to their high complexity and applicability, only few cores can be packed on the same die. On the other hand, GPU are built especially to process data in parallel (the idea being that each pixel can be rendered independently). So, while CPU can provide the best single thread performance, GPU can provide many more levels of parallelism. Several operations used in neural network training can be processed with many threads, for instance the convolution or the general matrix-matrix multiplication. Early researches have claimed GPU speedups in the order of one or two orders of magnitude faster over CPU implementations (N. K. Govindaraju et al., 2008; Silberstein et al., 2008; Z. Yang, Zhu, and Pu, 2008). However, it is more likely that the speedup is between one and ten times faster when both the CPU and the GPU code are carefully optimized (V. W. Lee et al., 2010). GPUs have known many successes in Machine Learning and especially

with neural network (Krizhevsky, Sutskever, and G. E. Hinton, 2012; Jia et al., 2014; Simonyan and Zisserman, 2014). GPUs are now considered by most as the go-to architecture for large neural networks training. Nevertheless, it is still possible to obtain very good performance using only CPU when using adequate performance optimizations (Wicht, Fischer, and Hennebert, 2016c). Section A.5 presents comparisons on CPU versus GPU performance on several operations.

Another improvement to speed up training is called Batch Normalization (Ioffe and Szegedy, 2015). While it does not help much the quality of the final trained model, it makes training much faster. Using this technique led to up to 14 times faster training in some cases. Each training mini-batch is normalized so that the distribution of the network activations do not vary too much during training when the parameters are updated. This normalization is performed on the inputs of each layer. This highly reduces the dependence on training hyperparameters in deep neural networks, since each layer will receive similar inputs. This has the advantage of allowing larger learning rates during training by not letting the gradients grow or decay too much. Batch Normalization by itself does not improve a lot the final results, but it facilitates the training so that it is more effective and much faster. It was also shown that this technique was acting as a regularizer and was able to remove the need for Dropout during training, thus speeding the training even more.

When these techniques are combined together, researchers are now able to train very large networks in a matter of hours, or incredibly deep networks such as the GoogLeNet network (Szegedy, Liu, et al., 2015) in a matter of weeks, whereas training such networks a decade ago would have taken weeks, respectively months.

Now that very large networks can be trained in reasonable time, it also becomes possible to make better use of data augmentation, to improve results even more. Data augmentation consists in creating new versions of the input samples present in the data set to have more data to train the model. This is especially useful when working with images, for which it is relatively straightforward to create new versions. The most common class of augmentation relies on affine transformations such as rotation, scaling or translation of the image pixels. In practice, it is often more efficient to use elastic transformations on images. In this case, a random displacement field is generated and applied to each pixel (Simard, Steinkraus, and Platt, 2003). Moreover, it is also possible to add random noise to the images. Finally, another simple technique is to extract random crops from an image. For instance, if the input is a 256x256 image, there are 1024 224x224 crops that could be considered (Krizhevsky, Sutskever, and G. E. Hinton, 2012). At test time, the activation probabilities are generally averaged over several crops of the test image.

3.4 Summary

Unsupervised pretraining is the technique that relaunched a very strong interest in Deep Learning. By pretraining a neural network in an unsupervised manner before

fine-tuning it for classification, researchers were able to reach new state of the art performance. This started a new wave of research in neural network training. This wave of research led to many new techniques that allowed researchers to train larger and larger networks more efficiently and with a fraction of the time that was necessary to train them only a few years ago.

Although it launched this new wave of research and interest, unsupervised pre-training was soon overshadowed by these new techniques with which pretraining was not fully necessary anymore. Although pretraining could still probably provide a better initialization of the weights for the new generation of neural networks, this would make training more complicated at the benefit of a few epochs of training gained.

It was recently demonstrated that unsupervised pretraining is still useful on small labeled data sets (LeCun, Bengio, and G. E. Hinton, 2015), especially on data sets for which large amount of unlabeled data is also available. This was also shown later in (O. E. David and Netanyahu, 2016) where a very small data set for painter classification was complemented with large amount of unlabeled data and achieved state of the art performance. Finally, it is also possible that unsupervised training become important again in the near future (LeCun, Bengio, and G. E. Hinton, 2015) with new unsupervised training techniques.

References for Chapter 3

- Baldi, Pierre and Peter J. Sadowski (2013). “Understanding Dropout”. In: *Advances in Neural Information Processing Systems 26*. Ed. by C. J. C. Burges et al. Curran Associates, Inc., pp. 2814–2822. URL: <http://papers.nips.cc/paper/4878-understanding-dropout.pdf> (cit. on p. 61).
- Bengio, Yoshua (2009). “Learning deep architectures for AI”. In: *Foundations and trends® in Machine Learning* 2.1, pp. 1–127 (cit. on pp. 58, 112, 149, 150).
- Bengio, Yoshua, Pascal Lamblin, et al. (2007). “Greedy layer-wise training of deep networks”. In: *Advances in neural information processing systems* 19, p. 153 (cit. on p. 59).
- Ciregan, Dan, Ueli Meier, and Jürgen Schmidhuber (2012). “Multi-column deep neural networks for image classification”. In: *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*. IEEE, pp. 3642–3649 (cit. on p. 58).
- David, Omid E. and Nathan S. Netanyahu (2016). “DeepPainter: Painter Classification Using Deep Convolutional Autoencoders”. In: *International Conference on Artificial Neural Networks*. Springer, pp. 20–28 (cit. on p. 63).
- Erhan, Dumitru, Yoshua Bengio, Aaron Courville, Pierre-Antoine Manzagol, et al. (2010). “Why does unsupervised pre-training help deep learning?” In: *Journal of Machine Learning Research* 11.Feb, pp. 625–660 (cit. on pp. 59, 60, 167).
- Erhan, Dumitru, Pierre-Antoine Manzagol, et al. (2009). “The Difficulty of Training Deep Architectures and the Effect of Unsupervised Pre-Training.” In: *AISTATS*. Vol. 5, pp. 153–160 (cit. on pp. 59, 60).

- Govindaraju, Naga K. et al. (2008). “High performance discrete Fourier transforms on graphics processors”. In: *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. IEEE Press, p. 2 (cit. on p. 61).
- Hinton, Geoffrey E. and Ruslan R. Salakhutdinov (2006). “Reducing the dimensionality of data with neural networks”. In: *Science* 313.5786, pp. 504–507 (cit. on pp. 19, 42, 50, 57, 86).
- Hinton, Geoffrey E., Nitish Srivastava, et al. (2012). “Improving neural networks by preventing co-adaptation of feature detectors”. In: *arXiv preprint arXiv:1207.0580* (cit. on pp. 61, 71).
- Hochreiter, Sepp (1991). *Untersuchungen zu dynamischen neuronalen Netzen. Diploma thesis, Institut für Informatik, Lehrstuhl Prof. Brauer, Technische Universität München* (cit. on pp. 19, 60).
- Ioffe, Sergey and Christian Szegedy (2015). “Batch normalization: Accelerating deep network training by reducing internal covariate shift”. In: *arXiv preprint arXiv:1502.03167* (cit. on pp. 20, 62).
- Jarrett, Kevin, Koray Kavukcuoglu, Yann Lecun, et al. (2009). “What is the best multi-stage architecture for object recognition?” In: *2009 IEEE 12th International Conference on Computer Vision*. IEEE, pp. 2146–2153 (cit. on pp. 16, 58, 71).
- Jia, Yangqing et al. (2014). “Caffe: Convolutional Architecture for Fast Feature Embedding”. In: *arXiv preprint arXiv:1408.5093* (cit. on pp. 62, 79, 194).
- Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton (2012). “Imagenet classification with deep convolutional neural networks”. In: *Advances in neural information processing systems*, pp. 1097–1105 (cit. on pp. 39, 62).
- Larochelle, Hugo, Yoshua Bengio, et al. (2009). “Exploring strategies for training deep neural networks”. In: *Journal of Machine Learning Research* 10.Jan, pp. 1–40 (cit. on pp. 59, 60, 167).
- LeCun, Yann, Yoshua Bengio, and Geoffrey E. Hinton (2015). “Deep learning”. In: *Nature* 521.7553, pp. 436–444 (cit. on pp. 20, 60, 63).
- Lee, Victor W. et al. (2010). “Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU”. In: *ACM SIGARCH Computer Architecture News* 38.3, pp. 451–460 (cit. on pp. 61, 76, 185, 187, 189).
- Masci, Jonathan et al. (2011). “Stacked convolutional auto-encoders for hierarchical feature extraction”. In: *Artificial Neural Networks and Machine Learning—ICANN 2011*, pp. 52–59 (cit. on pp. 58, 150).
- Nair, Vinod and Geoffrey E. Hinton (2010). “Rectified linear units improve restricted boltzmann machines”. In: *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pp. 807–814 (cit. on pp. 39, 60).
- Owens, John D. et al. (2007). “A survey of general-purpose computation on graphics hardware”. In: *Computer graphics forum*. Vol. 26. 1. Wiley Online Library, pp. 80–113 (cit. on pp. 61, 184).
- Silberstein, Mark et al. (2008). “Efficient computation of sum-products on GPUs through software-managed cache”. In: *Proceedings of the 22nd annual international conference on Supercomputing*. ACM, pp. 309–318 (cit. on p. 61).

- Simard, Patrice, David Steinkraus, and John C. Platt (2003). “Best Practices for Convolutional Neural Networks Applied to Visual Document Analysis.” In: *ICDAR*. Vol. 3, pp. 958–962 (cit. on pp. 62, 71).
- Simonyan, Karen and Andrew Zisserman (2014). “Very deep convolutional networks for large-scale image recognition”. In: *arXiv preprint arXiv:1409.1556* (cit. on p. 62).
- Srivastava, Nitish (2013). “Improving neural networks with dropout”. PhD thesis. University of Toronto (cit. on p. 61).
- Srivastava, Nitish et al. (2014). “Dropout: a simple way to prevent neural networks from overfitting.” In: *Journal of Machine Learning Research* 15.1, pp. 1929–1958 (cit. on pp. 16, 20, 61).
- Szegedy, Christian, Wei Liu, et al. (2015). “Going deeper with convolutions”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 1–9 (cit. on pp. 20, 62).
- Vincent, Pascal, Hugo Larochelle, Yoshua Bengio, et al. (2008). “Extracting and composing robust features with denoising autoencoders”. In: *Proceedings of the 25th international conference on Machine learning*. ACM, pp. 1096–1103 (cit. on pp. 58, 59, 151).
- Wager, Stefan, Sida Wang, and Percy S. Liang (2013). “Dropout Training as Adaptive Regularization”. In: *Advances in Neural Information Processing Systems 26*. Ed. by C. J. C. Burges et al. Curran Associates, Inc., pp. 351–359. URL: <http://papers.nips.cc/paper/4882-dropout-training-as-adaptive-regularization.pdf> (cit. on p. 61).
- Wan, Li et al. (2013). “Regularization of neural networks using dropconnect”. In: *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, pp. 1058–1066 (cit. on pp. 61, 197).
- Wicht, Baptiste, Andreas Fischer, and Jean Hennebert (2016c). “On CPU Performance Optimization of Restricted Boltzmann Machine and Convolutional RBM”. In: *IAPR Workshop on Artificial Neural Networks in Pattern Recognition*. Springer International Publishing, pp. 163–174 (cit. on pp. 62, 75).
- Wu, Haibing and Xiaodong Gu (2015). “Towards dropout training for convolutional neural networks”. In: *Neural Networks* 71, pp. 1–10 (cit. on p. 61).
- Yang, Zhiyi, Yating Zhu, and Yong Pu (2008). “Parallel image processing based on CUDA”. In: *Computer Science and Software Engineering, 2008 International Conference on*. Vol. 3. IEEE, pp. 198–201 (cit. on p. 61).
- Zamora-Martinez, Francisco, Javier Munoz-Almaraz, and Juan Pardo (2016). “Integration of Unsupervised and Supervised Criteria for Deep Neural Networks Training”. In: *International Conference on Artificial Neural Networks*. Springer, pp. 55–62 (cit. on p. 59).
- Zeiler, Matthew D. and Rob Fergus (2013). “Stochastic pooling for regularization of deep convolutional neural networks”. In: *arXiv preprint arXiv:1301.3557* (cit. on p. 61).

Chapter 4

Framework

*C++ does not give you performance, it
gives you control over performance*
Chandler Carruth

Contents

4.1	Deep Learning Library	67
4.2	Models	68
4.2.1	Restricted Boltzmann Machine	68
4.2.2	Convolutional Restricted Boltzmann Machine	69
4.2.3	Deep Belief Network	69
4.2.4	Neural Network	70
4.3	Visualization	71
4.4	Performance	73
4.4.1	Matrix Multiplication	73
4.4.2	Convolution	73
4.4.3	Smart Expression Templates	75
4.4.4	Parallelization	76
4.4.5	GPU	76
4.4.6	Memory	77
4.5	Preprocessor	78
4.6	Evaluation	79

4.1 Deep Learning Library

To support the research done during this thesis, a complete research framework has been developed and made open source and available to the scientific commu-

nity. This framework, called Deep Learning Library (DLL), has been developed in a generic manner to make it useful to other researchers. The rationale behind the development of the library rather than use an existing Machine Learning library is two-fold. First, at the beginning of this thesis, while Machine Learning libraries for standard Artificial Neural Networks (ANNs) and Convolutional Neural Networks (CNNs) were numerous and fully-featured, there was no support for Restricted Boltzmann Machine (RBM) and Deep Belief Network (DBN) other than small standalone, undocumented, programs. Moreover, from a research point of view, implementing a Machine Learning model from scratch is an excellent way of reaching a maximum understanding of the model.

The library is not tied to any experiments done during the course of this project and should be usable for any project requiring the use of Deep Learning. However, the features that have been selected for implementation are the models and algorithms required to support this thesis. Therefore this framework may not be as feature-complete as other popular Machine Learning libraries.

DLL has been implemented in C++, for performance and practical reasons. The framework has been made publicly available, under the terms of the open-source MIT license, to other Machine Learning researchers¹.

To see how DLL compares against other popular machine learning frameworks, in terms of learning quality and performance, see Appendix B.

4.2 Models

Several Machine Learning models have been implemented in the library during the course of this project. Since this research was heavily focusing on the RBM and Convolutional Restricted Boltzmann Machine (CRBM) models, this framework is also focused on these models. Nevertheless, several alternative models are also supported by DLL (See Section 4.2.4).

4.2.1 Restricted Boltzmann Machine

The library has full support for RBM. They can be used either as standard feature extractor, as an auto-encoder, as a denoising auto-encoder or to perform the pretraining of a DBN. The RBM model and its training procedure follows Hinton model and advices (G. E. Hinton, 2012).

The standard binary and Gaussian units are supported. Moreover, hidden units can also be Rectified Linear Units (ReLUs), capped-ReLUs (ReLU-n) or softmax units. These units allow for a large number of networks to be designed to achieve various goals.

An RBM can be trained using two different algorithms: Contrastive Divergence

¹<http://github.com/wichtounet/dll>

(CD) (see Section 2.4.3) or Persistent Contrastive Divergence (PCD) (see Section 2.4.5). For both methods, several options are supported by the framework:

- Using momentum to speed up and stabilize the learning convergence
- Computing the gradients on a mini-batch basis rather than per sample to improve efficiency and gradient quality and speed up training
- Using weight decay on the weight updates to reduce overfitting and improve the mixing rate of the Gibbs sampling
- Enforcing sparsity of the hidden units activations to generate features that are easier to interpret and more robust
- Using different initialization methods for the weights of the model
- Randomizing the order in which the samples are learned by the network

4.2.2 Convolutional Restricted Boltzmann Machine

Two versions of CRBM have been implemented, both following Honglak Lee's models (H. Lee, Grosse, et al., 2009). The first model is a standard two layers model. This follows the same principle as an RBM in which the activation probabilities of the hidden units are computed from visible units using a valid convolution. The second implementation uses Probabilistic Max Pooling (PMP). Since the pooling layer changes the activation function of the hidden layer, this is implemented as a second model rather than a subsequent pooling layer. For the first model, Max Pooling can be achieved, in a non-generative way, by using a standard pooling layer after the CRBM (see Section 4.2.4).

The same training algorithms and refinements as RBM (see Section 4.2.1) have been implemented. A special method for Sparsity regularization of the model is implemented, following the method introduced in (H. Lee, Ekanadham, and Ng, 2008).

4.2.3 Deep Belief Network

The DBN model is implemented as a stack of layers. Each layer receives inputs from the previous layer and forwards its output to the next one.

In DLL, the DBN is implemented as a more standard neural network than a regular DBN, i.e. layers that are not RBM-based are supported. The network can only be pretrained when the layers are of an auto-encoder type.

It supports a patches extraction layer, which extracts small image patches from a source image. Normalization and binarization layers have been implemented as well. Finally, the model also supports standard layers from ANN such as pooling

layers. In DLL, a DBN is a synonym of network and acts dependently of the layers it is composed of. For instance, a network made of standard convolutional layers followed by fully-connected layers will not support top-down inference.

When the DBN is pretrained, each of its layers is trained, layer by layer, in an unsupervised way. The first layer is trained using the input data, while the following layers are trained using the activation probabilities of their predecessor. Some layers are not pretrained: softmax layers, pooling layers and standard neural network layers.

In this library, the same data structure is used for both the DBN and the Convolutional Deep Belief Network (CDBN) models, only fine-tuning procedures differ from one model to another. Standard and convolutional layers can be mixed inside the network model.

The DBN model can be fine-tuned using either one of two different methods:

- A standard backpropagation algorithm (Stochastic Gradient Descent (SGD)) (LeCun, 1985). While being very simple to implement, this method is very effective. Refinements such as momentum and weight decay have been implemented. Moreover, this implementation supports mini-batch training, meaning that a batch of data will be processed before the gradients are computed. This makes for more efficient computation and for more stable training. The implemented algorithm is able to handle fully-connected layers and convolutional layers as well as pooling layers.
- A Complex nonlinear Conjugate Gradient optimization method (Shewchuk, 1994; Fletcher and Colin, 1964; Rasmussen, 2006). A Polack-Ribiere flavor of the Conjugate Gradient method is used to find the search directions. The step sizes are guessed with a line search using cubic and quadratic polynomial approximations and a Wolfe-Powell stopping criterion. While this training is generally more efficient and faster to converge than backpropagation, it is very complex and contains several hyper-parameters that are difficult to tune for the problem. Moreover, it is highly tied to the underlying model. For these reasons, it is often more convenient to use a simpler backpropagation technique. In DLL, this implementation supports only RBM layers, contrary to the SGD implementation that supports all types of neural layers.

Moreover, the DBN can also be used as feature extractor directly, by giving it the input and extracting the activation probabilities of the last layer of the network. Classification support with SVM (Chang and Lin, 2011) using these features is also integrated inside the library.

4.2.4 Neural Network

RBM and DBN models are often used conjointly with standard neural networks and convolutional neural networks. They also can be used with standard utility

layers that often complement neural layers. For this reason, the framework was augmented to support several more classical layers.

An RBM model, once trained, is highly similar to an ANN without any hidden units between the input units and the output units. Support for fully-connected (dense) layers has been implemented, following the same implementation principles. In the same idea, support for convolutional layers has been integrated, in a manner similar to CRBM and with the same performance optimizations.

Standard ANNs and CNNs can be trained using a Stochastic Gradient Descent optimizer. The optimizer has been implemented to handle hybrid networks as well, for instance convolutional layers followed by dense layers or a network where some layers have been pretrained as RBM. The Conjugate Gradient optimizer has not been adapted for neural network, the SGD optimizer being sufficient for most experiments. Several refinements have been implemented as well. To avoid overfitting, Dropout can be used while training (G. E. Hinton, Srivastava, et al., 2012), stochastically removing units from the network during training. Standard gradient descent uses very simple rules to update the weights and the biases of the network. Over the years, many optimization algorithms have been developed (Ruder, 2016). Support for the most popular optimization algorithms have been integrated in the framework: Momentum (Qian, 1999), Nesterov Accelerated Gradients (Nesterov, 1983), Adagrad (Duchi, Hazan, and Singer, 2011), Adadelta (Matthew D Zeiler, 2012), Adam (D. Kingma and Ba, 2014) and RMSProp (Tieleman and G. E. Hinton, 2012).

Several utility layers (non-neural layers) are available in the framework. The most used utility layers in neural networks are pooling layers. Max Pooling and Average Pooling layers have been implemented and are supported by the standard back-propagation training implementation. Normalization layers such as Rectification and Local Contrast Normalization (LCN) layers were also implemented (Jarrett, Kavukcuoglu, Lecun, et al., 2009). It is also possible to augment the input data to improve the training performance. For this, affine and elastic distortions (Simard, Steinkraus, and Platt, 2003) are used to create new versions of the samples to train on. Transformation layers to scale, binarize or zero-mean normalize the inputs of the network or the outputs of some layer are supported as well. Finally, the framework has integrated support for extracting patches from images directly inside the network. In this configuration, the final output of the network is a collection of feature vectors, one feature vector per patch. This integration results in significantly lower memory usage than generating the patches a priori and then concatenating the results outside the network.

4.3 Visualization

Weights learned by an RBM or a CRBM are often complex to interpret. Comparing two sets of features is not a trivial task. One solution that is often used is to look at the visual aspect of the learned weights. For instance, in the case of digit

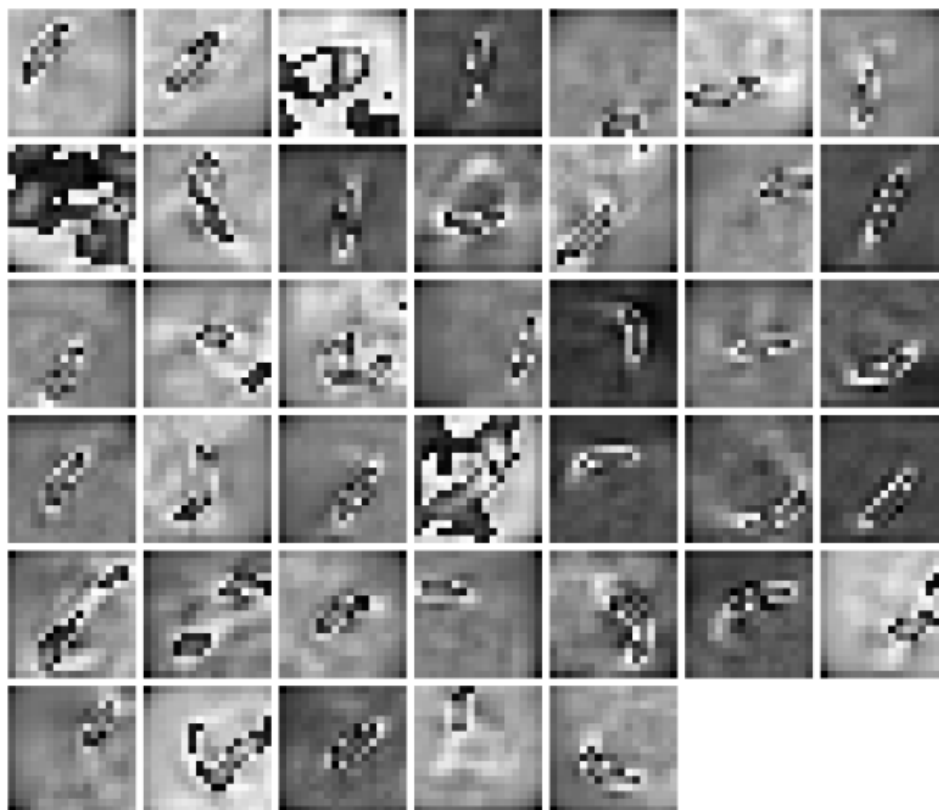


Figure 4.1: Visualization of the convolutional filters learned with a Convolutional RBM in DLL, on the MNIST dataset. These are the weights of the input layer, showing that it is learning to detect various strokes.

recognition, a filter can be a detector for the digit 1 while another can be a detector for the digit 9. In the case of multi-layers networks, the filters of the first layer may detect strokes, while the subsequent layers are learning edges.

For this reason, basic visualization capabilities have been added to the framework. Due to time constraints, the visualizer is only displaying the weights of the network being trained. The visualizer is built as a watcher of the training procedure and is updated every time a training epoch is completed. The visualizer has been implemented on top of the OpenCV library (Bradski, 2000). The weight values are directly converted to grayscale and no combination of layer weights is performed (see Section 2.4.4). Figure 4.1 shows an example of filters learned with the library on handwritten digits by a CRBM. While this visualization can be used on any of the layer during pretraining, it does only show them independently and therefore is not very visually appealing for layers other than the input layer. Indeed, the following layers will learn from the feature representation instead of from the image and therefore the filters may not be directly recognizable without going back to the input space (See Section 2.4.4).

4.4 Performance

A naive implementation of the RBM and CRBM models and their training algorithms will usually lead to low efficiency. When devising the implementation of such models, it is very important to pay attention to the performance.

4.4.1 Matrix Multiplication

When using CD to train an RBM with a mini-batch technique, the weights can be applied to all the inputs of the mini-batch in one step using a large matrix-matrix multiplication instead of several vector-matrix multiplication (G. E. Hinton, 2012). A matrix-matrix multiplication is a very complex algorithm to optimize and typically requires a doubly-blocked implementation (Goto and Van De Geijn, 2008). Fortunately, there already exists heavily-optimized of the algorithm, in the form of Basic Linear Algebra Subprograms (BLAS) libraries (Lawson et al., 1979). These libraries are providing several important linear algebra primitives such as the matrix-vector or the matrix-matrix multiplication operations. The Intel Math Kernel Library (MKL) has been chosen as BLAS library for the framework. The MKL library is also used to perform the outer product needed for the gradient computations, as well as some other minor operations.

4.4.2 Convolution

Training a CRBM requires two forms of convolution, a valid convolution to compute the activation probabilities of the hidden units and a full convolution for the visible units. A full convolution can be expressed in terms of a valid convolution with some amount of padding of the input matrix. Convolution operations are known to be computationally heavy and memory-bound. An example of a valid convolution of a 5x5 image by a 3x3 kernel is shown in Figure 4.2. The equation for the two-dimensional valid convolution of an image \mathbf{I} with a kernel \mathbf{K} into a output image \mathbf{C} can be obtained as such:

$$\mathbf{C}_{i,j} = \sum_k^K \sum_l^L \mathbf{I}_{i+k,j+l} * \mathbf{K}_{k,l} \quad (4.1)$$

In that case, the kernel is already reversed prior to the convolution, leading to what is known as a cross correlation that is more efficient to implement.

Modern Central Processing Units (CPUs) have advanced vectorization capabilities, allowing them to perform several floating point operations during the same cycle, known as Single Instruction Multiple Data (SIMD). The first optimization that is done on convolution is to use fine-tuned vectorized algorithm instead of a standard algorithm. Such algorithms have been written using either Streaming SIMD Extensions (SSE) or Advanced Vector eXtensions (AVX) for faster convolution. The

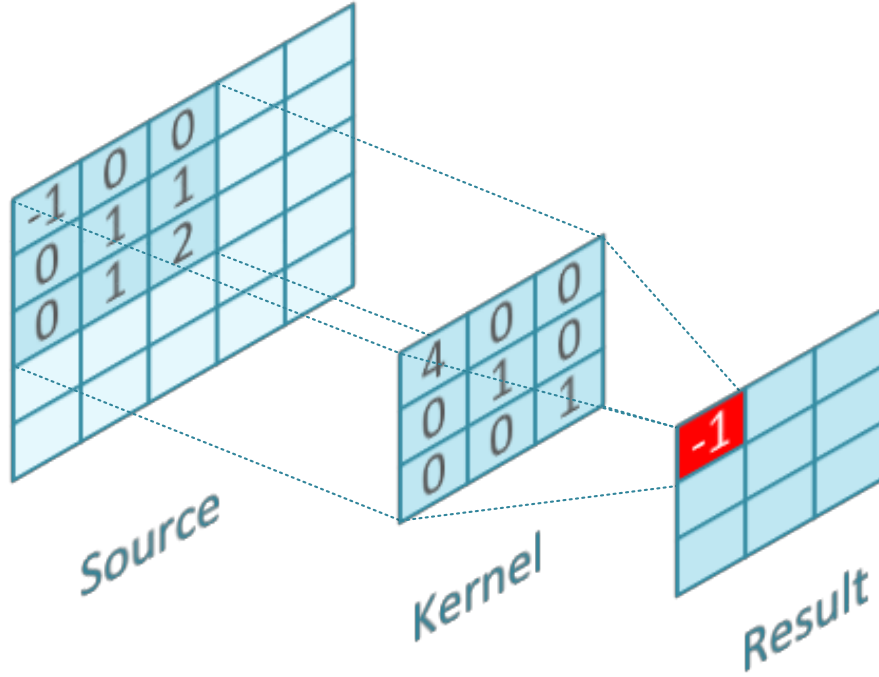


Figure 4.2: A 'valid' convolution of a 5x5 image with a 3x3 kernel. The kernel will be applied to every possible position inside the image.

choice between SSE and AVX is done given the capabilities of the machines and the dimensions of the convolution. For instance, AVX is able to perform operations on eight single-precision floating point operations with one instruction. In practice, we observed that a valid convolution convolution on single-precision floating point numbers is almost eight times faster than the same implementation without vectorization. However, this speedup highly depends on the dimensions of the kernel. When the kernel dimensions are not big enough or are not a multiple of the vector size, the kernel and the image are padded so that vectorization is more efficient. This significantly improves the performance of the vectorized implementations.

Algorithm 4.1 Compute a valid convolution $\mathbf{C} = \mathbf{I} \bullet_v \mathbf{W}$ with a Matrix-Matrix Multiplication

$\mathbf{W}' = \text{reshape}(\tilde{\mathbf{W}}, [1, k_1 k_2])$
 $\mathbf{I}' = \text{matrix}(k_1 k_2, c_1 c_2)$
 $\mathbf{I}' = \text{im2col}(\mathbf{I}, [k_1 k_2])$
 $\mathbf{C} = \mathbf{W}' * \mathbf{I}'$

A valid convolution is performed for each convolution filter of the CRBM for the same input image. It is possible to speed up this operation by replacing the K convolutions by one large matrix matrix multiplication (Ren and L. Xu, 2015). For some configurations of convolution and when a highly-optimized version of matrix-matrix multiplication is available, this significantly improves the performance of the valid convolution. Algorithm 4.1 describes the necessary steps for

this computation (*im2col* rearranges image blocks into columns). When this is done for several images at once, the speedups are significant. Benchmark details are available in Appendix A.3. When a full mini-batch is processed at once, the computation can be speedup further using parallelization.

A full convolution can be expressed in terms of a Discrete Fourier Transform (DFT). Indeed, the convolution theorem states that a convolution in the time domain correspond to a pointwise multiplication in the frequency domain (Bracewell, 1965). When the DFT operation is performed with a Fast Fourier Transform (FFT) implementation, it is very efficient. This process is detailed in Algorithm 4.2. The convolution needs to be large enough for the speedup to be interesting. Moreover, since the same image is convolved several times with different weights and the same weights are applied several times to different images, the complete process can be improved by precomputing all the DFT transformations. Benchmark details are available in Appendix A.4.

More details on the performed optimizations of the framework are available in (Wicht, Fischer, and Hennebert, 2016c).

Algorithm 4.2 Compute a full convolution $\mathbf{C} = \mathbf{I} * \mathbf{W}$ with FFT

$\mathbf{I}' = \mathbf{I}$ zero-padded to the output size
 $\mathbf{W}' = \mathbf{W}$ zero-padded to the output size
 $\mathbf{C}' = \mathcal{F}(\mathbf{I}') \cdot \mathcal{F}(\mathbf{W}')$
 $\mathbf{C} = \mathcal{F}^{-1}(\mathbf{C}')$

4.4.3 Smart Expression Templates

While most of the time is contained inside computation-intensive kernels (convolutions and matrix multiplications), computations of activation functions and gradients are still significant if not implemented efficiently. These operations can generally be expressed mathematically as element-wise operations on large vectors or matrices. To implement such expressions as efficiently as possible, while keeping the code clear, Smart Expression Templates (Iglberger et al., 2012a; Iglberger et al., 2012b) have been used.

Smart Expression Templates is a technique allowing to write an expression in a program very close to its mathematical form, while avoiding the overhead of writing such expression inefficiently. This technique allows the expression to be evaluated in one pass over the different matrices or vectors and allows the expression to be vectorized whenever possible. The matrix and vector computation code has been decoupled from the main framework into an independent library (ETL²). For instance, Listing 4.1 shows how the activation probabilities of a RBM are computed inside DLL. The notation is very close to the mathematical form.

²<http://github.com/wichtounet/etl>

Listing 4.1 Example of Smart Expression Templates in DLL to define the activation probabilities of a Restricted Boltzmann Machine.

```
h = sigmoid(b + v * w)
v = sigmoid(c + transpose(w * transpose(h)))
```

In practice, this could also allow more advanced techniques such as automatic differentiation or symbolic computation to be used. A complete neural network could be defined in terms of a graph and the gradients could be computed directly from the graph to update the weights.

4.4.4 Parallelization

Parallelization has been applied at several levels to improve the performance of training a neural network. First, the general matrix expressions can be computed using several threads to maximize the CPU usage. Moreover, the compute-intensive algorithms such as the convolution algorithms have been parallelized to use as many threads as available on the machine. This proved to have the biggest impact of parallelization. When it is available and more efficient to use, a parallel version of the BLAS library is also used.

When training RBM models, and especially CRBM models, one mini-batch at a time, most of the operations are only working on one image at a time. Therefore, they are very good candidates to parallelize, since they are completely independent. These independent operations are executed using several threads, speeding up the overall process.

4.4.5 GPU

Due to their very high parallelization support, Graphical Processing Units (GPUs) are able to perform some operations significantly faster than the CPU. Indeed, a GPU has two or three orders of magnitude more threads than a standard CPU. However, each of these threads is significantly slower than a CPU thread. In practice, on average, a fully-optimized GPU implementation is between two and three times faster than a fully-optimized CPU implementation (V. W. Lee et al., 2010). Nevertheless, this difference can highly vary from one operation to another.

The main difficulty in replacing standard algorithms by GPU-implemented algorithms is the high cost moving data from the CPU memory to the GPU memory and vice-versa. For these reasons, it is only interesting to use GPU-optimized algorithms when the cost of memory transfer is lower than the time gained by using the GPU. Matrix multiplication algorithms are very good candidates being extremely computationally intensive for large matrices. It is also very interesting when it is possible to make several operations together without going back to the GPU. For instance, $A * B * C$ could be done with the GPU without ever storing the result of

$A * B$ in CPU memory and therefore making the GPU version significantly faster than the overall CPU version.

The framework takes advantage of the GPU by replacing some CPU operations by GPU operations when it is more interesting (decided heuristically with a threshold on the size of the matrices and the type of operation). To ease the development of the library and its usage, the library does not make use of CUDA code directly, but relies on various libraries that are made available by NVIDIA to perform several computationally-intensive functions on GPU. This also has the advantage that these routines are already highly-optimized, for several different GPUs. Currently, the framework uses the GPU for the following operations:

- Matrix-Matrix multiplications with the NVIDIA CUDA Basic Linear Algebra Subprograms (CuBLAS) library
- FFT and Inverse FFTs of large matrices with the NVIDIA CUDA Fast Fourier Transform (CuFFT) library
- Convolutions and batched convolutions with the NVIDIA CUDA Deep Neural Network (CuDNN) library

Moreover, the framework also avoids going back through CPU for sequence of operations that can be done entirely on GPU (such as $A * B * C$). However, the framework cannot train a network entirely on GPU like other machine learning frameworks.

The use of GPU is seamless, changing a simple compiler option will make the library use the GPU libraries available on the target machine.

4.4.6 Memory

Neural networks have a high memory consumption. It is greatly increased when the results of all the filters of a convolutional layer are computed at once (for performance reasons), meaning that large intermediate matrices need to be kept in memory. When training the network in a layer-by-layer manner, a lot of intermediate data is kept in memory for the following layers. Since the first convolutional layers of a network are typically augmenting the data, this means a direct multiplication of the data set size. This is not a problem when working on small data sets such as MNIST where images are 28x28 in dimensions, but this quickly becomes an issue when the data set contains realistically-sized images or contains much more images. For these reasons, several optimizations have been performed to handle large data sets while keeping the memory consumption as low as possible.

The first optimization is to choose the best data type for the weights and the results of the different layers. Using a single-precision floating point type instead of a double-precision type halves the necessary memory. Moreover, this also has

the advantage that vectorized expressions can handle twice more data in one instruction and may improve performance of some fine-tuned algorithms by a factor of two. On GPU, some frameworks are going even further by using half-precision floating point numbers (Courbariaux, Bengio, and J.-P. David, 2014). However, this is not natively available on CPU.

Secondly, the workspace size of some algorithms can also be optimized. For instance, when computing the state of the hidden units in a CRBM, instead of computing K convolutions and then accumulating their results, a large amount of memory can be saved by accumulating the results after each convolution. This optimization is performed for several operations, resulting in large savings of memory in some cases (depending on the dimensions of the networks).

Finally, some memory can also be saved by not allocating memory for some layers. For instance, normalization layers can simply be processed inline and applied directly to the memory of the previous layer, thus saving a full copy of the memory of the previous layer. This is done for every layer that does not change the dimensions of the input and does not need any training.

For very large data sets, these optimizations are not enough to be able to train a model completely in memory. In such cases, an out-of-memory mode has been implemented so that very few information are kept in memory at any given time and the data is assumed to be costly to retrieve (typically from disk). This comes at the cost of significantly decreasing the performance of the system. To minimize this decrease, two things are very important. First, the training algorithm (CD here) should always have enough work and not starve. For this, it becomes interesting to use a separate thread to compute the next batch of input for the training of the current layer. Secondly, it is important to minimize the number of time the same data is read from disk and it is equally important to read as much data in one time as possible (for disk and memory optimization reasons). For this, several mini-batches of training data are always read at once. The number of mini-batches read at once depends on how large a mini-batch is in memory and can be configured accordingly. For the first layer of the network, this change has less effect, but the intermediate results of the network are not kept completely in memory for the subsequent layers. Data is passed in batch from the data set through the already trained layers to form the training data of the currently trained layer. This means that the features from the first layers will be computed several times for the same data. Therefore, it becomes important that inference is as fast as possible. The gain for fine-tuning is less interesting since it is not done one layer at a time and therefore does not need to store full intermediate results.

4.5 Preprocessor

All the features of the framework can be directly used as a header-only C++ library. This allows for the highest level of customization and the best performance. Nevertheless, not all Machine Learning researchers are familiar with programming

in general or with the C++ programming language in particular or simply do not have the time to create a new application dedicated to using the framework for their problem. For these reasons, a simple preprocessor system has been developed, allowing the researcher to use the library by simply describing the network and its options in a text file, in a manner similar to other popular Machine Learning libraries such as Caffe (Jia et al., 2014). While complex tuning of the network in the training phase cannot be achieved with this method, this makes the configuration and training of the network much easier. Every Machine Learning researcher should be able to use this front-end to create Deep Learning models. The most common options and features of the framework are available in this operating mode. Behind the scenes, the preprocessor is simply generating a C++ program with the necessary code to implement the configured options. The generated program is then compiled and run directly. Listing 4.2 shows an example of a configuration for a three layer network.

Listing 4.2 Example of a three-layer network defined with the DLL preprocessor

```
network:
  conv:
    channels: 1
    v1: 28
    v2: 28
    filters: 10
    w1: 5
    w2: 5
    activation: relu
  dense:
    hidden: 250
    activation: relu
  dense:
    hidden: 10
    activation: softmax

options:
  training:
    epochs: 100
    batch: 10
```

4.6 Evaluation

It is important for the framework to be compared against other equivalent available frameworks in order to validate its results and to judge its merits. Therefore, it was decided to compare the framework against five popular machine learning frameworks. Six different experiments have been performed, on three different data sets. The evaluation is done on two different parameters. First, the quality of the learned models is evaluated. Indeed, the networks trained with DLL should not be significantly different from a network with the same parameters trained by

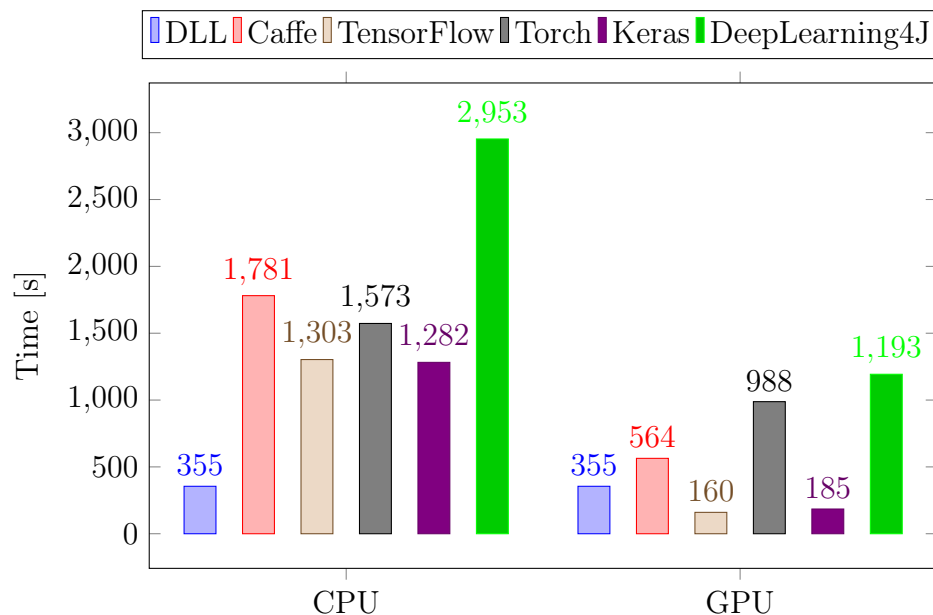


Figure 4.3: Training time performance comparison of the frameworks on a Convolutional Neural Network experiment, on CPU and on GPU.

another framework. Secondly, since one of the main focus of this framework was on efficiency, the training time of the models has also been evaluated.

Overall, the DLL framework was able to train networks with similar accuracies as the other tested frameworks. Moreover, on each experiment, it was the fastest framework for training a neural network on CPU and was generally competitive on GPU as well. For instance, Figure 4.3 shows the time necessary to train a CNN for each of the different frameworks on the MNIST data set on the CPU and on the GPU. All the details of the evaluation are presented in Appendix B.

References for Chapter 4

- Bracewell, Ron (1965). “The Fourier transform and its applications”. In: *New York* 5 (cit. on p. 75).
- Bradski, Gary (2000). “The OpenCV library”. In: *Dr. Dobb’s Journal of Software Tools* (cit. on pp. 72, 91).
- Chang, Chih-Chung and Chih-Jen Lin (2011). “LIBSVM: A library for support vector machines”. In: *ACM Transactions on Intelligent Systems and Technology* 2 (3). Software available at <http://www.csie.ntu.edu.tw/~cjlin/libsvm>, 27:1–27:27 (cit. on p. 70).
- Courbariaux, Matthieu, Yoshua Bengio, and Jean-Pierre David (2014). “Training deep neural networks with low precision multiplications”. In: *arXiv preprint arXiv:1412.7024* (cit. on p. 78).

- Duchi, John, Elad Hazan, and Yoram Singer (2011). “Adaptive subgradient methods for online learning and stochastic optimization”. In: *Journal of Machine Learning Research* 12.Jul, pp. 2121–2159 (cit. on p. 71).
- Fletcher, Reeves and Reeves M. Colin (1964). “Function minimization by conjugate gradients”. In: *The Computer Journal* 7.2, pp. 149–154. DOI: 10.1093/comjnl/7.2.149. URL: <http://dx.doi.org/10.1093/comjnl/7.2.149> (cit. on pp. 70, 97).
- Goto, Kazushige and Robert Van De Geijn (2008). “High-performance implementation of the level-3 BLAS”. In: *ACM Transactions on Mathematical Software (TOMS)* 35.1, p. 4 (cit. on p. 73).
- Hinton, Geoffrey E. (2012). “A Practical Guide to Training Restricted Boltzmann Machines.” In: *Neural Networks: Tricks of the Trade (2nd ed.)* Ed. by Grégoire Montavon, Genevieve B. Orr, and Klaus-Robert Müller. Vol. 7700. Lecture Notes in Computer Science. Springer, pp. 599–619. ISBN: 978-3-642-35288-1. URL: <http://dblp.uni-trier.de/db/series/lncs/lncs7700.html#Hinton12> (cit. on pp. 29, 68, 73, 93).
- Hinton, Geoffrey E., Nitish Srivastava, et al. (2012). “Improving neural networks by preventing co-adaptation of feature detectors”. In: *arXiv preprint arXiv:1207.0580* (cit. on pp. 61, 71).
- Iglberger, Klaus et al. (2012a). “Expression templates revisited: a performance analysis of current methodologies”. In: *SIAM Journal on Scientific Computing* 34.2, pp. C42–C69 (cit. on p. 75).
- (2012b). “High performance smart expression template math libraries”. In: *High Performance Computing and Simulation (HPCS), 2012 International Conference on*. IEEE, pp. 367–373 (cit. on p. 75).
- Jarrett, Kevin, Koray Kavukcuoglu, Yann Lecun, et al. (2009). “What is the best multi-stage architecture for object recognition?” In: *2009 IEEE 12th International Conference on Computer Vision*. IEEE, pp. 2146–2153 (cit. on pp. 16, 58, 71).
- Jia, Yangqing et al. (2014). “Caffe: Convolutional Architecture for Fast Feature Embedding”. In: *arXiv preprint arXiv:1408.5093* (cit. on pp. 62, 79, 194).
- Kingma, Diederik and Jimmy Ba (2014). “Adam: A method for stochastic optimization”. In: *arXiv preprint arXiv:1412.6980* (cit. on p. 71).
- Lawson, Chuck L. et al. (1979). “Basic linear algebra subprograms for Fortran usage”. In: *ACM Transactions on Mathematical Software (TOMS)* 5.3, pp. 308–323 (cit. on p. 73).
- LeCun, Yann (1985). “Une procédure d’apprentissage pour réseau à seuil asymétrique (a Learning Scheme for Asymmetric Threshold Networks)”. In: *Proceedings of Cognitiva 85*. Paris, France, pp. 599–604 (cit. on p. 70).
- Lee, Honglak, Chaitanya Ekanadham, and Andrew Y. Ng (2008). “Sparse deep belief net model for visual area V2”. In: *Advances in neural information processing systems*, pp. 873–880 (cit. on pp. 46, 69, 131).
- Lee, Honglak, Roger Grosse, et al. (2009). “Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations”. In: *Proceedings of*

- the 26th Annual International Conference on Machine Learning*. ACM, pp. 609–616 (cit. on pp. 43, 47–49, 69, 99).
- Lee, Victor W. et al. (2010). “Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU”. In: *ACM SIGARCH Computer Architecture News* 38.3, pp. 451–460 (cit. on pp. 61, 76, 185, 187, 189).
- Nesterov, Yurii (1983). “A method for unconstrained convex minimization problem with the rate of convergence $O(1/k^2)$ ”. In: *Doklady ANSSSR*. Vol. 269. 3, pp. 543–547 (cit. on p. 71).
- Qian, Ning (1999). “On the momentum term in gradient descent learning algorithms”. In: *Neural networks* 12.1, pp. 145–151 (cit. on p. 71).
- Rasmussen, Carl Edward (2006). *Minimize a differentiable multivariate function, implementation in Matlab*. URL: <http://learning.eng.cam.ac.uk/~carl/code/minimize/minimize.m> (cit. on pp. 70, 97).
- Ren, Jimmy S. J. and Li Xu (2015). “On Vectorization of Deep Convolutional Neural Networks for Vision Tasks”. In: *CoRR* abs/1501.07338. URL: <http://arxiv.org/abs/1501.07338> (cit. on p. 74).
- Ruder, Sebastian (2016). “An overview of gradient descent optimization algorithms”. In: *CoRR* abs/1609.04747. URL: <http://arxiv.org/abs/1609.04747> (cit. on p. 71).
- Shewchuk, Jonathan R. (1994). *An Introduction to the Conjugate Gradient Method Without the Agonizing Pain*. Tech. rep. Pittsburgh, PA, USA (cit. on pp. 70, 97).
- Simard, Patrice, David Steinkraus, and John C. Platt (2003). “Best Practices for Convolutional Neural Networks Applied to Visual Document Analysis.” In: *ICDAR*. Vol. 3, pp. 958–962 (cit. on pp. 62, 71).
- Tieleman, Tijmen and Geoffrey E. Hinton (2012). *Lecture 6.5—RmsProp: Divide the gradient by a running average of its recent magnitude*. COURSERA: Neural Networks for Machine Learning (cit. on p. 71).
- Wicht, Baptiste, Andreas Fischer, and Jean Hennebert (2016c). “On CPU Performance Optimization of Restricted Boltzmann Machine and Convolutional RBM”. In: *IAPR Workshop on Artificial Neural Networks in Pattern Recognition*. Springer International Publishing, pp. 163–174 (cit. on pp. 62, 75).
- Zeiler, Matthew D (2012). “ADADELTA: an adaptive learning rate method”. In: *arXiv preprint arXiv:1212.5701* (cit. on p. 71).

Part II

Applications

Chapter 5

Sudoku Recognition

*The first step is to establish that something
is possible; then probability will occur*

Elon Musk

Contents

5.1	Introduction	85
5.2	Data set	86
5.3	State of the art	88
5.3.1	Camera-based OCR	89
5.4	Sudoku Digit Detection	90
5.5	Digit Classifiers	92
5.5.1	System I: Fully-Connected DBN	92
5.5.2	System II: Convolutional DBN	99
5.6	Results	102
5.7	Performance	104
5.8	Summary and potential extensions	105

5.1 Introduction

The first question this research is trying to address is how much difference the unsupervised pretraining of an Artificial Neural Network (ANN) is impacting fully-connected networks and convolutional networks. Another question we are addressing is about the capability of the deep networks to perform feature extraction on different types of inputs with the same neural network.

To answer these questions and as first research experiment, the detection and recognition of Sudoku puzzles in images taken from Swiss newspapers with cell

phone cameras has been chosen. This particular problem was chosen for several reasons. First, a data set was already available in the research team, it was only necessary to complete and finalize it. Moreover, due to the nature of cell phone cameras and images taken from newspapers, the task is not necessarily trivial, making it interesting to work on. Finally, since the recognition step necessary in this problem is a form of digit recognition and this problem has already been addressed with Restricted Boltzmann Machine (RBM) and Deep Belief Network (DBN) (G. E. Hinton and Salakhutdinov, 2006), it was taken as an adequate problem to start with. In a second time, the data set was extended with images containing both handwritten and computer printed digits.

The Sudoku is a famous Japanese logic puzzle. It is played on a 9x9 grid with numbers from 1 to 9. At the beginning of the game, the grid is partially filled, with some cells being left empty. The goal of the game is to fill these empty cells with numbers so that every row, every column and every 3x3 block contain all numbers only once. This is a game that is often present in newspapers and that is very appreciated nowadays. Figure 5.1 gives an example of Sudoku, before and after resolution.

In this chapter, the overall system that was implemented to detect and recognize Sudoku puzzles will be presented. First, the data set that was collected for this purpose is presented. Then, the state of the art in Sudoku detection and recognition is analyzed. The different classification systems are presented in detail and their results are compared. The effect of pretraining is evaluated and compared for both data sets and systems. Then, the overall results are presented. The run-time performance of the system is also evaluated. Finally, conclusions about this experiment are drawn.

5.2 Data set

To the best of our knowledge, at the beginning of the experiment, no free data set of Sudoku images was available. The research on this problem was only presenting results obtained on few images, not publicly available and therefore were impossible to reproduce or to compare with other results. For these reasons, it was decided to publish a data set that would allow future research to be done on publicly available data.

A data set of Sudoku images was built on top of a smaller existing data set of images from the research lab, gathered by Patrick Anagnostaras (Anagnostaras, 2008). The Sudoku Recognition data set (SRD)¹ (Wicht and Hennebert, 2015) is composed of 200 Sudoku images taken from different Swiss newspapers and from various cell phones. The data set is split into a training set of 160 images and a test set of 40 images. Due to the small size of the data set, the use of a validation set is left to the discretion of the users. The pictures include parts of the newspaper

¹https://www.github.com/wichtounet/sudoku_dataset

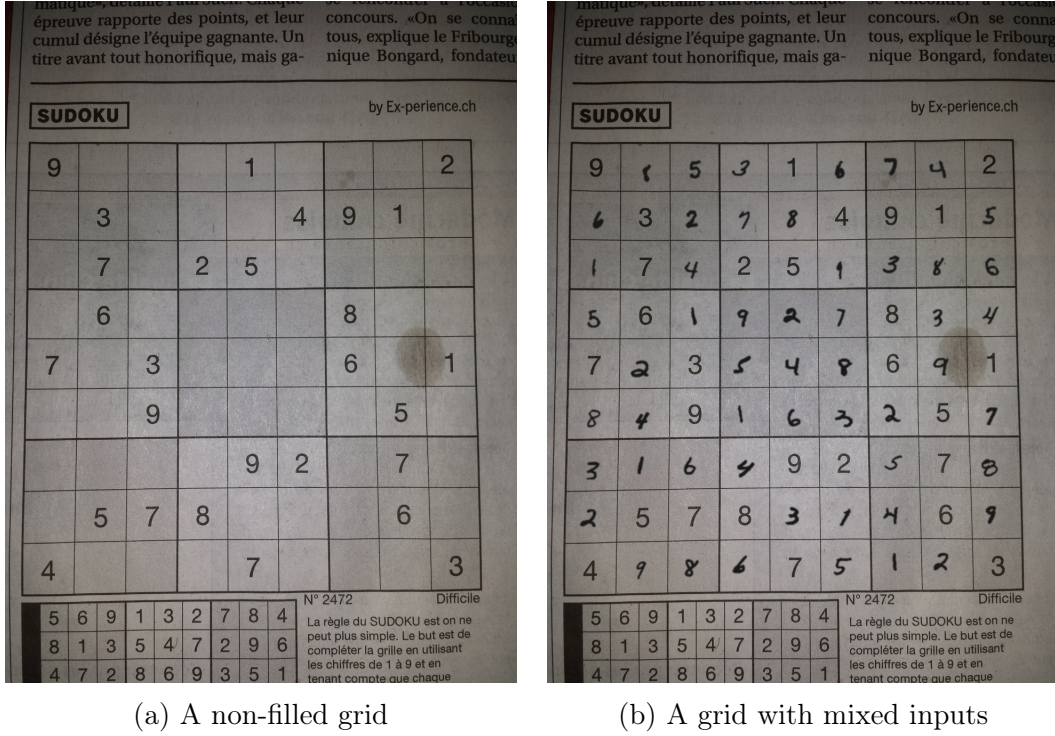


Figure 5.1: Exemplary images from the Sudoku Recognition data set (SRD)

surrounding the puzzle, including sometimes parts of other Sudoku puzzles, which makes the grid detection in itself somehow challenging. The pictures were taken so that the conditions are very different from one image to another, including shadows, blur and illumination gradients. Moreover, the nature of newspapers makes some images distorted because of non-flat pages.

One specific question that was of interest in this work was to determine whether a neural network would be able to easily handle two different types of inputs, i.e. printed and handwritten digits. For that purpose, the data set was enhanced with grids that were already filled. In that case, each grid contains both handwritten and printed digits. Because not enough filled grid images were available, the data set was augmented by filling the empty cells of existing puzzles with distorted images from the MNIST data set (LeCun, Bottou, et al., 1998). The MNIST images are used for drawing the ink of the new synthetic digits. The cells were detected using a algorithm (see Sudoku puzzle Detection) and handwritten digits were inserted at a random position around the center of the cell. Six different colors were used as ink. Finally, a light Gaussian blur is used to make the result more realistic.

Figure 5.1 presents two exemplary images from the data set. Figure 5.1a shows the original image with an empty grid while Figure 5.1b shows the same image in its synthetically filled version (mixed inputs).

To summarize, there are three versions of the data set:

- *V1*: The original version (Wicht and Hennebert, 2014), with 160 images.
- *V2*: The improved version (Wicht and Hennebert, 2015), with 200 images
- *mixed*: Synthetically augmented version of *V2* with fully-filled grids, with 200 images

In this chapter, the results are collected on *V2* and *mixed*. Since *V2* is mostly an improved version of *V1*, results are not collected on *V1*.

5.3 State of the art

While a lot of research was done on strategies and algorithms for solving the Sudoku puzzle, few existing research focuses specifically on detection and recognition of Sudoku images. Nevertheless, there have been a few attempts at this problem.

A. Van Horn proposed a complete system to recognize and solve Sudoku puzzle from images, based on a Hough transform (Van Horn, 2012). The puzzle is detected from its four corners using the intersections of the detected lines. For recognition, they center the digits inside its cell and use a ANN to classify them. Interestingly, the empty cells are not detected a priori but are passed directly to the classifier. This has been tested on a very small set of images. Moreover, the test set used for the results is not publicly available.

Another system was proposed the same year by Simha et al (Simha, Suraj, and Ahobala, 2012). The image is first binarized using adaptive thresholding and all the components connected to the border of the images are removed. This allows later recognition steps to perform better by having less noise and less components candidates. A second Connected Components algorithm is used to identify the largest component area inside the image. This component is used as the outer grid of the Sudoku puzzle. Digits inside the grid are also located by their components. Then, the detected digit images are assigned to a cell by applying a virtual 9x9 grid on top of the puzzle. Finally, classification is performed using a very simple template matching strategy. Again, no results computed on a publicly available data set are available.

Since neither of the two systems existing at the time of the experiment were tested on a publicly available data set, it was not possible to compare our system directly with these systems. This is one of the reason the Sudoku Recognition data set (SRD) (see Section 5.2) was gathered and published.

More results were published after the analysis and proposal of our system. The first one was proposed by Ly et al. (Ly and Vo, 2015). They also focus on Sudoku images taken from magazines with a camera. In their system, the grid is detected using a Hough transform on a binarized image. From there, the angle of the grid is computed and the puzzle is rotated to help character recognition. The grid is split into 81 cells by using a template matching technique with decreasing template

size until all cells are found. Finally, the digits are recognized using a neural network. No details were provided on the tests performed on the system. Kamal et al. designed a system for detecting and solving Sudoku puzzles especially for implementation on a Field Programmable Gate Array (FPGA) (Kamal, Chawla, and Goel, 2015). In this system, local thresholding is applied to the gray-scale image to binarize it. Then, a Hough transform is used to detect the biggest quadrilateral inside the image. The digits are extracted from the grid using a division in equally sized cells. Finally, some form of Optical Character Recognition (OCR) is used to recognize the digits, but few information is given on that part. The system was tested on a private data set of thirty images.

5.3.1 Camera-based OCR

All the pictures from the data set were taken in newspapers with phone camera. The rationale behind this choice was to increase the difficulty of the task and make it more interesting and somehow closer to what a real-life application would look like. Indeed, text detection and recognition of images acquired from high quality scanners is a long-time studied problem. There have been some extremely efficient solutions proposed, see (Impedovo, Ottaviano, and Occhinegro, 1991) for an exhaustive survey. As such, it is generally now considered an easy problem, although there still exists some difficult edge cases, such as very complex historical documents.

When the images of text are coming from optical camera, the problem remains challenging for several reasons. Indeed, scanners have been conceived especially for the task of digitalizing text documents, and later adapted to handle images as well. They are very well tuned to this task, contrary to cameras that are tuned to take scene images. Scanners are more accurate and especially more stable whereas different cameras may produce very different pictures for the same original scene and their quality is rarely on par with scanners. One caveat in phone cameras is also that focus is not always good with optical zoom, although modern smartphones have been improving over the years. Moreover, whereas a scanner works in a constrained and well-defined environment by controlling the illumination, pictures can be taken with cameras in very different conditions. The background is generally different from one picture to another and is subject to different lighting conditions and may exhibit shadows and gradients of illumination. Finally, the angle at which the picture is taken is entirely dependent on the person taking the picture whereas scanned documents are generally almost perfectly aligned.

For a reference and more details about the challenges of camera-based analysis, a complete survey of has been made available by Liang et al. (J. Liang, Doermann, and Li, 2005). The different steps that are commonly performed to solve this issue, such as localization, normalization and binarization, are presented with existing solutions. It is shown that, although some solutions exist, it remains a challenging problem and that many sub-problems are still unsolved. On the specific context of Mobile-Based OCR, the different challenges are presented in (Jain

et al., 2013). This work analyzes and compares the different potential solutions to these challenges. Many approaches rely on preprocessing steps that allow images taken with a phone-based camera to be handled by standard existing techniques. The conclusion is that, although solutions are getting better, there is still room for improvement in this particular area.

When pictures are taken in a newspaper, new difficult conditions may appear. The main difficulty is that the newspaper pages are not flat. This may lead to images distorted in three dimensions making them much harder to analyze. Moreover, since images are taken from different newspapers, the font styles and font sizes are likely to be different from one editor to another. Finally, the Sudoku puzzle is rarely well isolated on the page. Indeed, there are different objects (for instance images, text or simply lines) surrounding the object of interest.

5.4 Sudoku Digit Detection

The first step before trying to recognize the digits is to extract them from the Sudoku image. There are two main approaches to this problem:

1. *Top-Down*: Detect the grid, then the cells and finally extract the digits from each detected cell.
2. *Bottom-up*: Detect the digits first and then build the grid from the digit positions in the image.

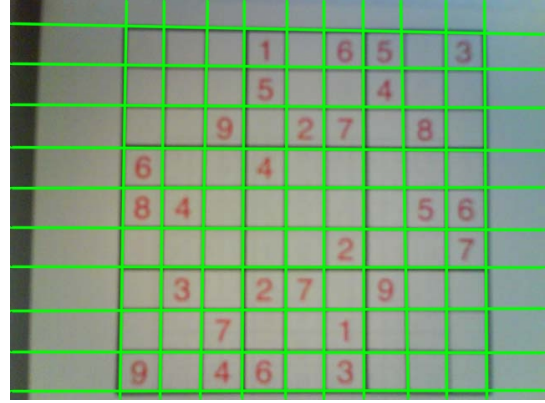
The most used technique in similar cases is to use a top-down approach, which is what was selected. A bottom-up approach could have been solved by machine learning techniques using text detection models (Coates, Carpenter, et al., 2011). But since text detection was not the focus of this research, standard image processing techniques were chosen to solve the problem in a top-down approach. Moreover, since some cells are empty, bottom-up detection would not necessarily have been trivial.

To detect each digit inside the image, the following steps are performed:

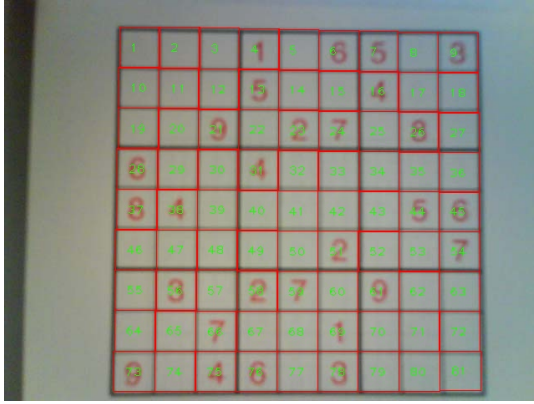
1. The Canny algorithm (Canny, 1986) is used to detect the edges of the image. From these edges, segments of lines are reconstructed with a Progressive Probabilistic Hough Transform (Matas, Galambos, and Kittler, 2000).
2. Segments belonging to the same line are merged together using a simple Connected Component analysis (Ronse and Devijver, 1984) and some heuristics dedicated to the problem.
3. Intersections between the detected lines are then computed. The best cluster of intersections (a perfect cluster of a puzzle has 100 intersections) is then selected and its four corner points are computed as the corners of the grid.



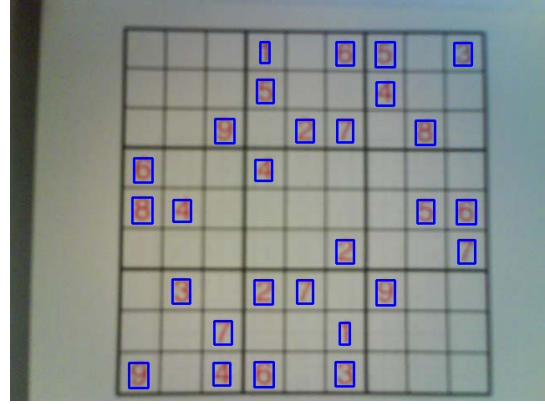
(a) The original image



(b) The detected lines



(c) The detected grid cells and their indices



(d) The final detected digits

Figure 5.2: The main steps for detection of a Sudoku puzzle inside an image, from the original image to the detected digits.

4. The detected grid is split in 9x9 cells and the digit inside each cell, if any, is isolated using a Contour Detection algorithm (Suzuki and Abe, 1985).
5. The isolated digits are centered inside a white image and then resized to a 32x32 image. The final image is extracted from the source color image and then binarized using another binarization scheme for a better result.

The image processing algorithms of the OpenCV library (Bradski, 2000) were used. The steps are the same for detection of the *V2* and *mixed* data sets. However, since there are no empty cells in *mixed* grids, several heuristics have been chosen to be more aggressive since there must be a digit in each cell. Moreover, some tests are also not necessary. Overall, the results of the detection are significantly better for the *mixed* data set than for the *V2* data set.

Complete information about these steps is available in (Wicht and Hennebert, 2014). Figure 5.2 shows the results of the four main steps computed on an exemplary image from the data set.

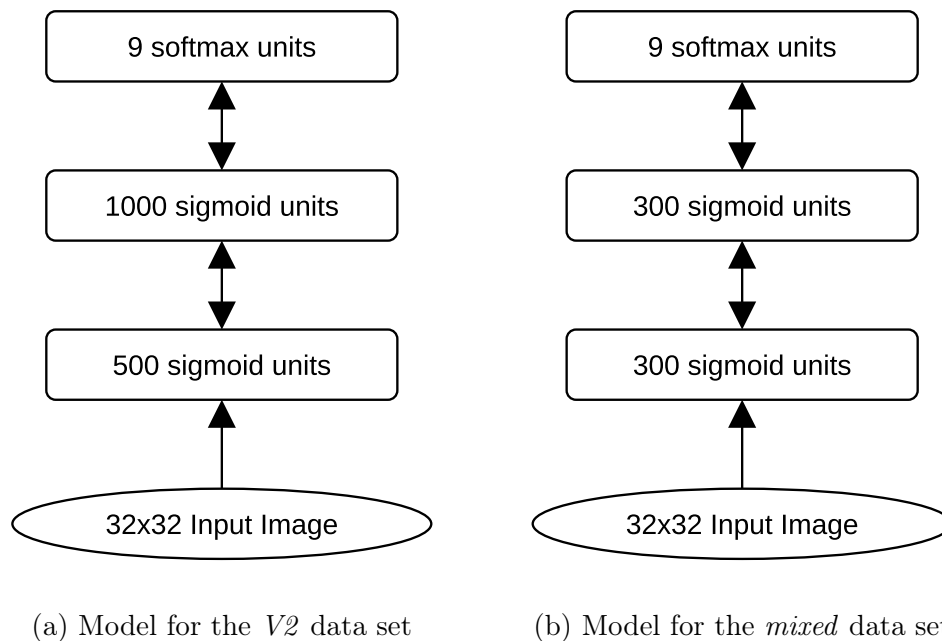


Figure 5.3: Abstract view of the Deep Belief Network used for classification of the Sudoku digits. Each layer of the network is an RBM.

5.5 Digit Classifiers

After the complete detection process has been performed, each 32x32 digit binary image has to be classified. Only digits are classified, the empty cells are eliminated by the detection process. To solve this task, two different systems were developed. The detection is performed in the same manner independently of the classifier system.

Both data sets have 160 images for training and 40 images for testing. On the *V2* training set, the detection process returns 4656 digits to train the system, while there are 1156 digits to classify at test-time. On the *mixed* data set, there are 12960 training digits and 3240 test images to classify.

The systems are evaluated using two metrics. The accuracy at the cell-level, i.e. the percentage of cells correctly classified over the entire data set, and the accuracy at the Sudoku level, i.e. the percentage of Sudoku correctly classified. A Sudoku is correctly classified if its 81 cells are correctly classified.

5.5.1 System I: Fully-Connected DBN

The first system is a standard DBN, with fully-connected layers only. The two DBN models used for classification are depicted in Figure 5.3.

Several architectures have been considered for this network. The final architecture is the one that brought the better classification accuracy while still being

reasonably-sized. In both cases, a three-layer network yielded the best classification performance. When more layers are being added to the network, the final classification did not improve more, while the training and inference times were going up. For these reasons, focus was made on a three-layer network. Regarding the number of hidden units in each layer, large number of combinations were tested and the configuration the most suited for classification was selected. With the final number of hidden units, variations of about 5% to 10% in their numbers do not change the final classification error. The size of the input images was chosen as an average of every extracted cell so that the centering and scaling would not alter the images too significantly.

Both networks are made of three RBM layers. In both cases, the input layer has the same number of units, corresponding to the dimensionality of the images, namely 32x32 visible units. For the *V2* data set, the first layer has 500 hidden sigmoid units. The second layer is made of 1000 hidden sigmoid units and the last layer has nine hidden softmax units. This network contains 1'021'000 parameters to learn. For the second network (for the *mixed* data set), it was necessary to drastically reduce the size of the network due to major overfitting issues, as explained below. The final network has 300 sigmoid hidden units in the first and second layer and the standard nine softmax units for the last layer. Thus, this network has significantly less parameters than the first, only 399'900 weights. It should be noted that the size of the first network could be reduced without significant impact on performance.

The weights of each RBM are initialized from a Gaussian distribution of mean 0.0 and variance 0.01. The first layer visible biases are initialized using $\mathbf{c}_i = \log(\frac{\mathbf{p}_i}{1-\mathbf{p}_i})$ where \mathbf{p}_i is the probability of unit i being 1 in the training set, following the advice from (G. E. Hinton, 2012). All the other biases (visible and hidden) are initialized to zero.

Each sigmoid layer is pretrained, in turn, as an RBM with Contrastive Divergence (CD), with only one Gibbs step (CD_1). While it is possible to use CD to pretrain a softmax layer, it does not help the fine-tuning step. Generally, due to the small number of softmax units, pretraining this kind of layer tends to hurt the classification performance, therefore they have not been pretrained. Mini-batch training is used to speedup training, with 32 images per mini-batch. The complete data set is randomly shuffled before each epoch. The learning rate ϵ is set to 0.1 for each layer. L2 weight decay is used on each layer on the weights (not on the biases) to reduce overfitting. Momentum learning is used to increase the speed of learning and stabilize the training. The momentum α is fixed to 0.9 for each layer. Using a variable momentum has been experimented with, however, it only decreased the stability of training while not helping the final classification error, therefore only a fixed momentum was used in the end. For the *V2* network, each layer is trained for 50 full epochs.

Finally, once the pretraining is done, the complete network is trained for classification with the cell labels using a Stochastic Gradient Descent (SGD) optimization method. Again, mini-batch training is used, with 32 images per mini-batch. The learning rate ϵ is kept to 0.1 for the entire training. The data set is randomly

shuffled before each epoch. A momentum α of 0.9 is also used and L2 weight decay is applied on all weights. The network is trained for 100 epochs.

In the case of the model for the *mixed* data set, it was necessary to be more careful than for the first data set. There are two major differences between the data sets that are impacting training. First, the second data set has about three times more training samples, making it easier to train. Moreover, the detection is much more efficient on the second data set than on the first. Overall, this makes the training highly prone to overfitting. For this reason, a smaller network was used. Moreover, the parameters needed to be tuned accordingly. A smaller learning rate ϵ was used (0.03 instead of 0.1) and an higher L2 weight cost λ (0.005 in place of 0.0002). Finally, the number of pretraining steps was reduced and the training was stopped once the error stopped decreasing on a small validation set, while the first network was stopped after a fixed number of epochs). With the final set of parameters, the overfitting was largely reduced but still remains an issue. In practice, it would be more efficient to use advanced techniques such as Dropout and Data Augmentation (see Section 3.3). However, to keep the models simple for this experiment, it was decided to not use any of these advanced techniques.

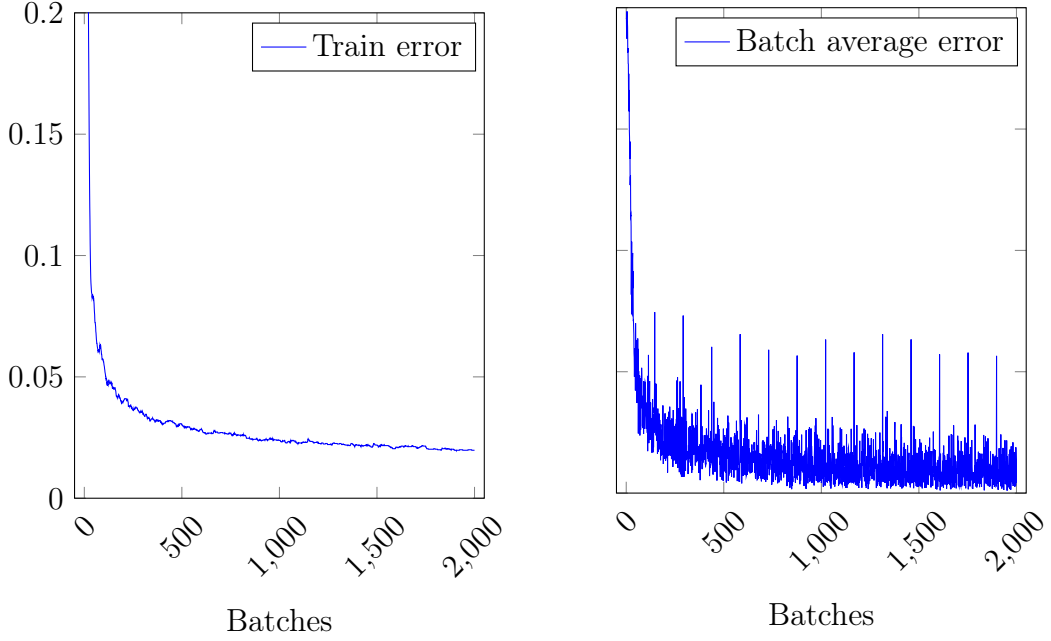
Once the network is trained for classification, it is trivial to use it to classify an image. The activation probabilities of the first layer are computed from the image and are passed to the second layer. The second layer computes its features and passes them to the softmax layer. The outputs of the softmax layer are the probabilities of each label. The unit with the highest activation probability indicates the most probable digit. This final label is selected for evaluation.

Results

The best trained network achieved an accuracy of 97.75% on the *V2* test set. An error analysis shows that the main reason of errors lies in the quality of the detection process being quite poor at detecting empty cells and passing them to the classifier. If the errors due to the detection (empty cells not eliminated by the detection process) are not taken into account, the accuracy is about 98.8% without retraining. On the *mixed* test set, the achieved accuracy is 96.3%. In this case, the problem is not related to detection but rather to significant overfitting of the network.

Figure 5.4a shows the evolution of the training error during fine-tuning on *V2* data set. This follows a rather standard evolution with fast learning at the beginning and slower evolution in the end. In Figure 5.4b, the mini-batch error over time is presented. Again the same kind of curve can be observed. Nevertheless, it can also be observed that not all mini-batches are alike, with some important outliers.

More interestingly, to see the impact of unsupervised pretraining, Figure 5.5 shows the training error over time, during fine-tuning, with different numbers of pretraining epoch. These results are gathered on the *V2* data set. The first interesting fact about this figure is that fine-tuning after pretraining is much more stable than



(a) Evolution of the training error during the first 2000 mini-batches epochs of fine-tuning the Sudoku Recognition network, for the system I on the V2 data set. (b) Evolution of the mini-batch error during the first 2000 mini-batches epochs of fine-tuning the Sudoku Recognition network, for the system I on the V2 data set.

Figure 5.4: Evolution of the error for training the Sudoku Recognition Network, during fine-tuning, after unsupervised pretraining on the V2 data set.

training without any pretraining. This can be observed on the pretrained curves that have very small variations compared to the blue curve which varies a lot even after more than a thousand batches. It seems that pretraining smoothes out the training by not letting it diverge too much. Interestingly, pretraining the network for only one epoch does not help the training error to converge faster, as can be observed on the red curve. Indeed, although it speeds up the early training and stabilizes the error, the error slowly becomes worse than when the network is not pretrained. On the other hand, when the network is pretrained for enough epochs, the training error converges faster and stays lower than the error of the randomly-initialized network. Generally, the more epochs of pretraining are performed, the faster the training error converges. However, in this particular case, pretraining the network does not help break the plateau that can be seen in the Figure. Although 100 epochs of pretraining converges faster than 50 epochs, it does not help diminish the final classification error. Therefore, 50 epochs of CD were performed for the final results.

For comparison, the evolution of the training error on the *mixed* data set is presented in Figure 5.6. It is interesting to see that it is very different from the previous results (see Figure 5.5). First, the error is going down in a smoother way, there is no long plateau at the end of training. This is because the data set is in fact easier than the V2 data set. Moreover, while the effect of pretraining as

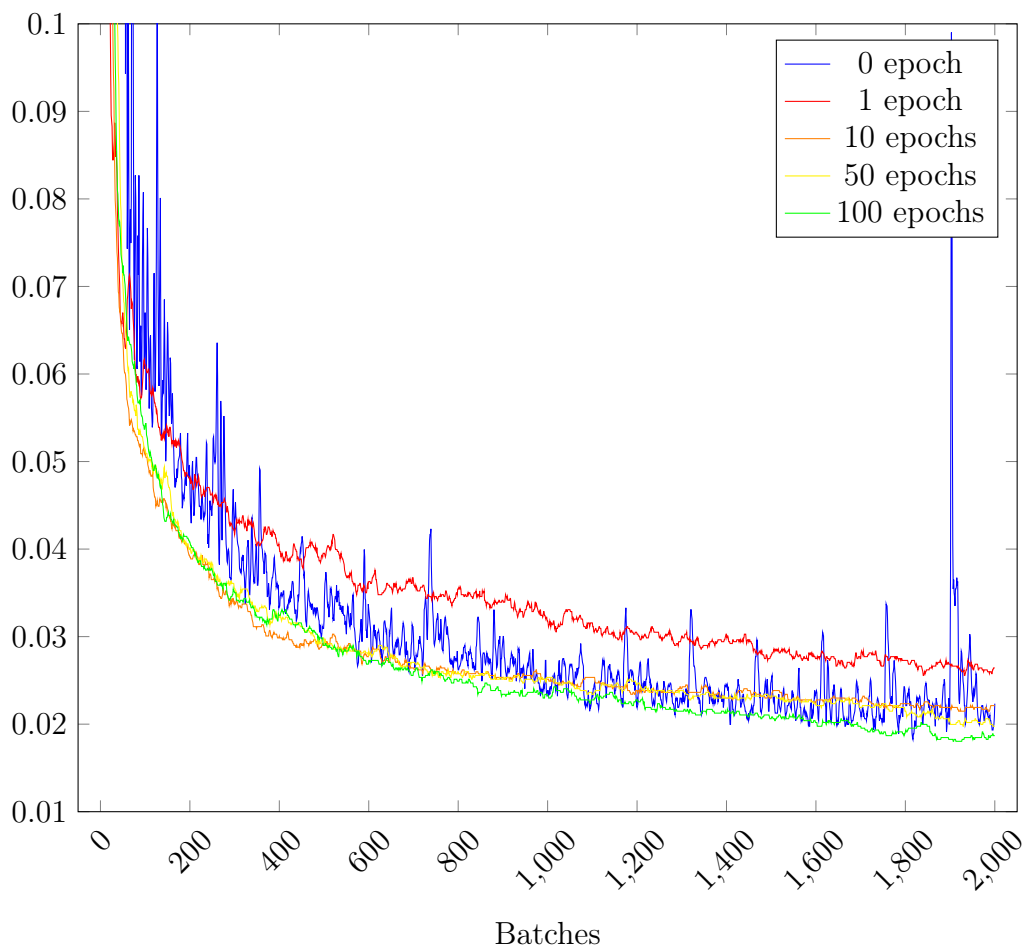


Figure 5.5: Evolution of the training error during the first 2000 mini-batches epochs of fine-tuning the Sudoku Recognition network, with different epochs of unsupervised pretraining, for the DBN system, on the $V2$ data set.

a regularization method in the first case was very clear, it is not so significant in this case. Indeed, to overcome the overfitting issues, stronger L2 regularization, smaller learning rates and higher number of epochs have been used. Since the network is already highly regularized, the effect of pretraining is less important. The main difference in the different number of epochs is that the network without pretraining is learning much faster than even the network with only one epoch of pretraining. However, while the training error is going down very quickly, the test error is very high on this network. Using even one epoch of pretraining largely reduces this issue of overfitting. Moreover, the more epochs of pretraining are used the faster the network is learning while still being regularized. However, after some point, the training is again prone to too much overfitting. This is the case where using 50 epochs of pretraining, while seemingly better on the graph, is in fact worse than the network pretrained for 25 epochs. In the end, the best results were achieved with around 25 epochs of pretraining.

The results presented in this section for the first system and the $V2$ data set are

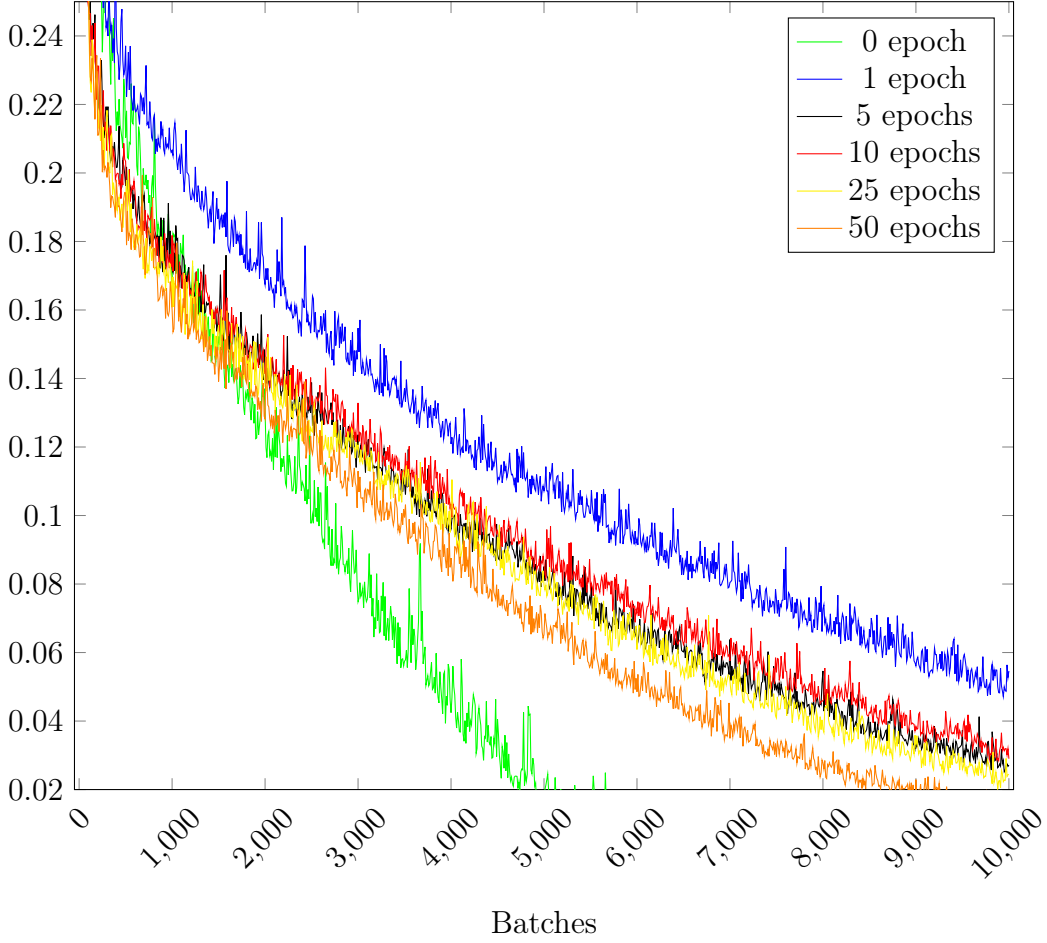
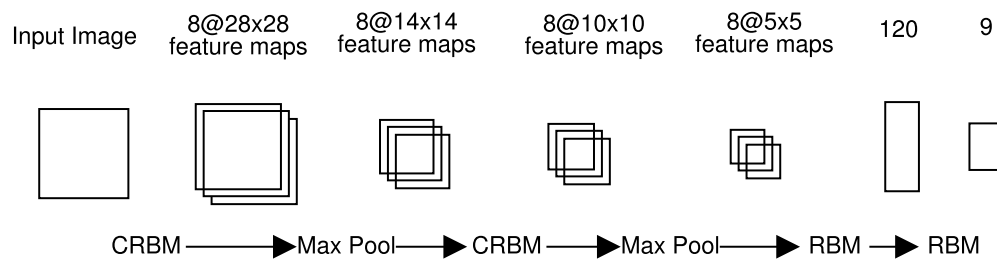


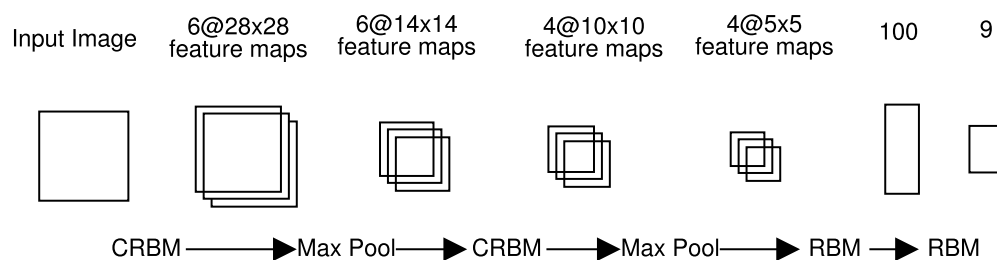
Figure 5.6: Evolution of the training error during the first 10000 mini-batches epochs of fine-tuning the Sudoku Recognition network, with different epochs of unsupervised pretraining, for the System I, on the *mixed* data set.

different from those published in (Wicht and Hennebert, 2014), being slightly worse. Indeed, there are several differences between these experiments. First, the present experiment has been conducted on a much more recent version of the deep learning framework, which may explain some differences. More importantly, the present results are computed on the second version of the data set which contains 200 images instead of 160 images. Moreover, the new version of the data set contains new images that are very different from the previous one, making it an harder task. Then, the results are based on the results of the detection made in (Wicht and Hennebert, 2015) which is tailored to handle mixed handwritten and computer-generated digits. Finally, while the original results were based on a special Polack-Ribiere flavor of the nonlinear Conjugate Gradient (CG) optimization method (Shewchuk, 1994; Fletcher and Colin, 1964; Rasmussen, 2006), the present results are computed using a more classical SGD optimization method.

The classification results obtained on the *V2* and *mixed* data sets are significantly worse than those obtained on the MNIST digit detection task (1.25% reported



(a) V2 data set



(b) mixed data set

Figure 5.7: Abstract view of the Convolutional DBN used for classification of the Sudoku digits.

in (G. E. Hinton, Osindero, and Teh, 2006)) with a similar DBN. Although both tasks are apparently similar, there are several major differences. First, the MNIST data set offers 60000 images for training while the current experiment has less than 5000. Results for the V2 data set would probably improve if there were more training examples. Moreover, the MNIST data set contains only very clean images (the handwriting is not easy, but the images contain few noise) while images here are extracted from difficult images and the preprocessing steps do not do a perfect job of removing the noise. Also, while MNIST images have been scanned on handwritten document, the Sudoku images are taken from phone camera on newspaper. The resulting quality is not as good or as stable as those of a scanner. Furthermore, both data sets contain some digits that are computer-written while MNIST only contains handwritten digits. Finally, the images for training are extracted from the digit detector which is not perfect. Some digits are extracted from cells that are empty (incorrectly detected) and some digit images still contain lines from the Sudoku puzzle. Therefore, an error rate higher than those obtained on MNIST is explainable.

5.5.2 System II: Convolutional DBN

In order to study the effect of pretraining on convolutional layers and compare it to the effect on dense layers, the second classification system that has been implemented for this task is a Convolutional Deep Belief Network (CDBN) (H. Lee, Grosse, et al., 2009). While being based on the same principle, a major difference between this system and the first system is that it also has convolutional layers in front of the fully connected layers. The network has two convolutional layers, each followed by a max pooling layer reducing each dimension by a factor of two. After these first four layers, two fully-connected layers are used to perform the classification. The first three layers are using logistic sigmoid hidden units. The last fully-connected layer uses softmax hidden units. The trained layers are pretrained as Convolutional Restricted Boltzmann Machine (CRBM) and RBM. Figure 5.7 presents the details of the network for the two data sets. The network has much less parameters than the fully connected versions. Indeed, the convolutional layers being locally connected, it needs fewer parameters. There are no direct connection from each input pixel to each output feature. Each output feature is only connected to a small window of the input image. The weights of each output feature inside each group are shared. Moreover, since both max pooling layers are reducing the dimensionality of the feature map by a factor of four, the following layers have less inputs to handle. Overall, the first network has 26880 parameters while the second one has 11650 parameters, to be compared with more than one million for the first network of the first system. When they are compared with state of the art Convolutional Neural Networks (CNNs), these networks appear to be very small. For the first network, larger layers were not necessary due to the nature of the data set. For the second data set, the network was kept small in order to limit the problems due to overfitting and our will to stay with simple regularization techniques.

The architecture that has been chosen for this task is similar to a very standard CNN architecture (LeCun, Bottou, et al., 1998), but the neural layers are being pretrained. Although there are few parameters to learn, the system is still largely impacted by overfitting. This is because of the over-complete representation of the convolutional layers. Indeed, the learned representation is larger than the input representation, making the network prone to learn very simple solutions. A first solution to solve this problem is to make the network as small as possible to limit over-completeness. However, this also limits the learning capabilities of the model. The second simple solution that was adopted is to reduce the learning rate and train for more epochs. Moreover, weight decay is also used while trained, in order to limit the number of free parameters, thus reducing overfitting. Finally, pretraining itself being a regularization method, it should help reducing the overall difficulty in training the network. When this is combined with a smart choice of pretraining hyper-parameters, it was found out to reduce overfitting. Ideally, it would be necessary to use advanced techniques for overfitting such as Dropout and Data Augmentation (see Section 3.3). However, we wanted to keep the model simple to be able to compare it to the other systems.

The weights of the RBMs and CRBMs are initialized from a Gaussian distribution of mean 0.0 and variance 0.01. The hidden biases of the CRBMs are initialized to -0.1 and all the other biases are zero-initialized.

The networks are trained in a similar manner as the first system. Each RBM and CRBM is trained, layer by layer, using mini-batch CD with one Gibbs step (CD_1). L2 weight decay and momentum are also used to improve learning. The learning rates ϵ have been very carefully tuned in order to avoid overfitting at this stage. The first CRBM learning rate is set to 0.001, the second CRBM uses 0.0001 and the third RBM uses 0.001. Due to the smaller learning rates, it was necessary to pretrain the layers for 100 epochs. The data set is shuffled before each epoch.

Using the cell labels, the complete network is trained for classification using mini-batch SGD. Momentum and L2 weight decay are used for each network. For the first network, the learning rate ϵ is set to 0.08. The network is trained until the error goes below some threshold. The second network uses a smaller learning rate (0.03). The L2 weight cost λ is set to 0.0005.

Results

The best trained network for this system achieves an accuracy of 98.12% on the *V2* data set and an accuracy of 98.92% on the *mixed* data set. For the first data set, the result is really close to the result of the first system (97.75%). This result is very difficult to improve on since most of the errors are coming from errors in the detection step. However, the result on the *mixed* data set were significantly improved compared to the first system (96.3%).

While this system produced very good results on the *mixed* data set, its optimization proved far from trivial. Indeed, as it was the case for the first system, this data set is very prone to overfitting. It was necessary to test a wide range of training parameters and network dimensions in order to find a network with the maximum accuracy. Nevertheless, once carefully configured and trained, this system proved significantly better than the system based on the fully-connected DBN.

To see if the impact of pretraining was the same on convolutional neural networks as on fully connected neural networks, the evolution of the training error during fine-tuning was observed, with different numbers of epochs for layer-wise pretraining. These results are presented in Figure 5.8. The first interesting thing to observe is that even one epoch of pretraining provides a much better initial value of the weights rather than a random initialization. Indeed, it can be seen on the figure that even after 3000 mini-batches, the training error is still significantly lower than when the network is not pretrained at all. However, it does not seem to act as strongly as a regularization method as it did for the fully-connected DBN. Indeed, the curves with more epochs of pretraining are only slightly more stable than the curves with less epochs. This can be explained by the lower number of weights of this system compared to the System I. Overall, this seems to indicate that pretraining still does provide an excellent initialization of the weights for a

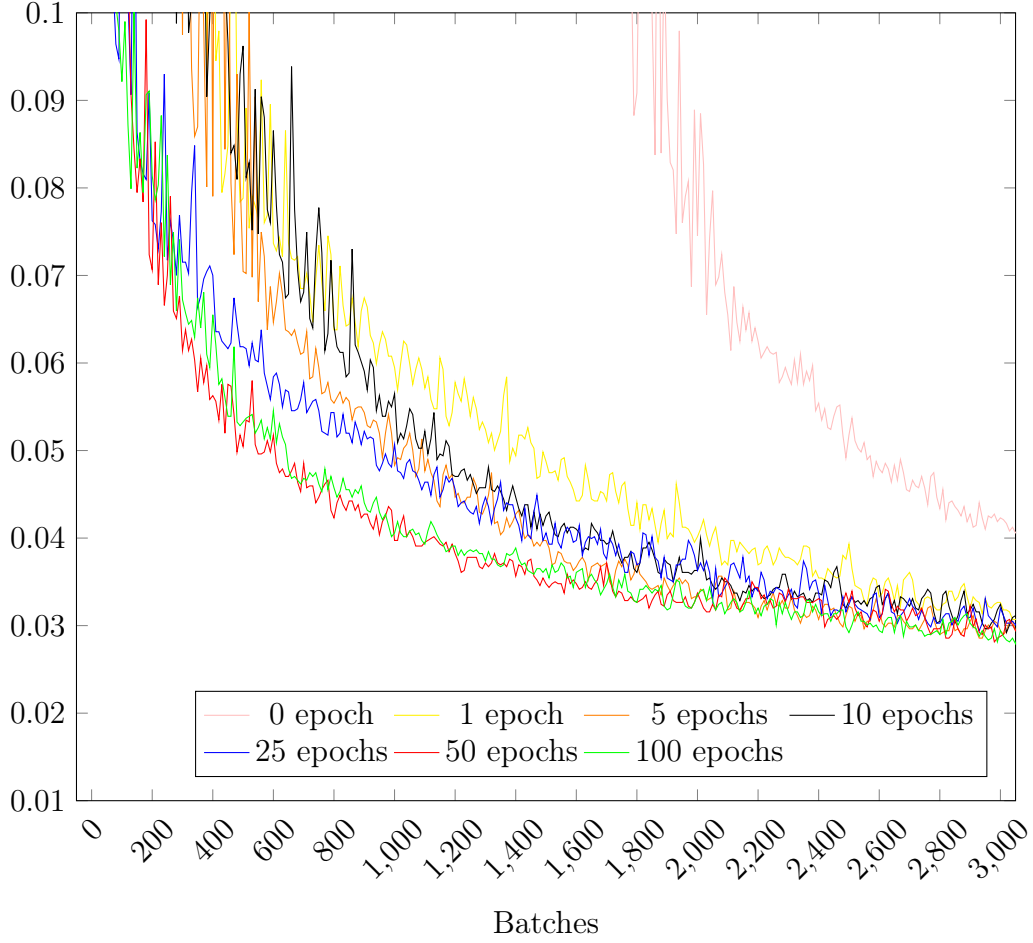


Figure 5.8: Evolution of the training error during the first 3000 mini-batches epochs of fine-tuning the Sudoku Recognition network, with different epochs of unsupervised pretraining, for the Convolutional DBN system, on the *V2* data set.

CDBN but has less of a regularization effect as for the fully-connected DBN.

Figure 5.9 shows the results of the same experiments on the *mixed* data set. Again, it is clear that the pretraining epochs help a lot in providing a good initialization of the weights. Indeed, the error without pretraining (the pink curve) has a training error that is significantly higher than the models with even 1 epoch of pretraining. On the other hand, the regularization effect of pretraining is definitely not present here. Indeed, all curves are very unstable during learning. This confirms the results from the *V2* data set, by showing that pretraining in a CDBN and by extension in a CNN provides a good initialization of the weights but has a much smaller regularization effect than on fully-connected networks.

The results for the *mixed* data set are very different from those published in (Wicht and Hennebert, 2015). The main difference is that the results presented in the paper are using a CDBN only for feature extraction while a Support Vector Machine (SVM) is used for classification from these features. Moreover, the results were also conducted on different versions of the deep learning framework which may

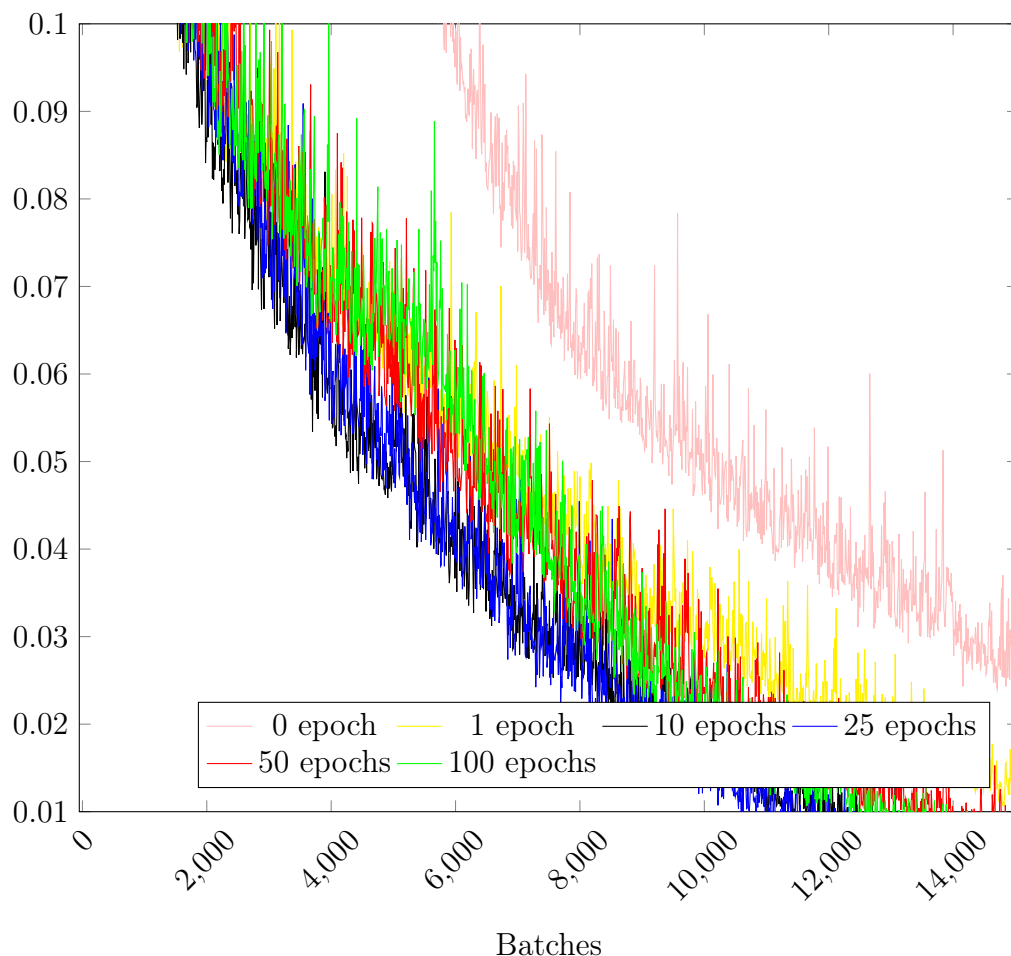


Figure 5.9: Evolution of the training error during the first 15000 mini-batches epochs of fine-tuning the Sudoku Recognition network, with different epochs of unsupervised pretraining, for the Convolutional DBN system, on the *mixed* data set.

explain some small differences.

5.6 Results

The final results obtained by the proposed systems are summarized in Table 5.1 for both data sets and for both systems.

For the *V2* data set, 33 Sudoku puzzles out of 40 in the test set are perfectly recognized (every one of the 81 cells is correctly classified), averaging to a 82.5% accuracy. There is no difference between the systems at the Sudoku-level, but the second system is a bit better at cell-level accuracy. While this may seem low, there are still very few errors on each Sudoku. Indeed, when considering the accuracy at the level of a cell, 98.68% of the cells are correctly classified with System I. When the errors are considered individually, more than 70% of them are coming

Level	System I	System II	Level	System I	System II
Cell-Level	98.68%	98.79%	Cell-Level	97.01%	99.14%
Sudoku-Level	82.5%	82.5%	Sudoku-Level	65%	92.5%

(a) *V2* data set(b) *mixed* data set

Table 5.1: Accuracy of Sudoku Detection and Recognition system. A Sudoku is correctly recognized if every one of its 81 cells is correctly classified. System I is based on a fully-connected DBN while System II is based on a Convolutional DBN.

Step	<i>V2</i>	<i>mixed</i>
Layer 1	37	35
Layer 2	57	23
Fine-Tuning	178	68
Total	290	147

(a) System I

Step	<i>V2</i>	<i>mixed</i>
CRBM 1	27	286
CRBM 2	15	179
RBM 1	4	30
Fine-Tuning	934	4028
Total	994	4547

(b) System II

Table 5.2: Training time for the Sudoku Recognition System, in seconds. The total time includes the time necessary to load the data and detect the digits.

from detection issues with empty cells. It is either a digit detected inside a cell that was empty or an empty cell detected where there was a digit. Regarding the classification errors, most of the errors are coming from digits that are very similar such as 5 and 6. Moreover, the networks are being trained on the results of the detection, inducing more errors in the models, explaining the large outliers in the training figures. Considering the low quality of several images and the difficulty of Camera-based OCR and newspaper conditions (see Section 5.3.1), these results were deemed satisfying. If they were to be improved, a better detection process and a large amount of data would probably be necessary.

For the *mixed* data set, the results are very different. Since the detection results are much better on this data set, the results are highly depending on the quality of the classification system itself. The first system performs poorly, due to high unsolved overfitting of the model, classifying correctly only 65% of the Sudoku images. On the other hand, the second system excels at this with 92.5% of the Sudoku images perfectly classified. Although the results on cell-level of the System I are acceptable (97.01% accuracy), this still makes a lot of cells incorrectly classified and therefore a poor overall score at the Sudoku level. With 99.14% of the cells correctly classified by the System II, almost all Sudoku are correctly classified.

Step	Min	Max	Mean	Median
Image Loading	2272	78019	10640	2413
Line Detection	34570	69903	44084	39267
Grid Detection	202	28748	5884	230
Digit Detection	25747	78310	33791	31031
Digit Recognition	3027	4876	3827	3819
Total	67881	206192	96099	76293

Table 5.3: Computing time, in microseconds, necessary for each step of the proposed system for Sudoku Detection and Recognition. With the System I on the *V2* data set.

5.7 Performance

The experiments in this section were conducted using the optimized implementations of the developed framework (see Section 4.4). The hardware configuration is presented in Section A.2. The times necessary for training the different networks is presented in detail in Table 5.2.

Overall, it can clearly be seen that the fully-connected DBN (System I) is much faster to train than the convolutional DBN (System II). This comes principally from the fact that fully-connected layers are much easier to optimize (almost all the time is contained in the matrix-matrix multiplication kernel) than convolutional layers. Moreover, a CNN generally tends to take longer to converge than fully-connected models.

For the System I on the *V2* data set, the complete network takes less than five minutes to train. More than half the time is spent in fine-tuning which is more computationally intensive than pretraining since it works on the complete network rather than one layer at a time. The pretraining time for each layer mostly depends on its dimensions. The trained networks being relatively small and the data set being very small, the training times were not expected to be critical. Nevertheless, when pretraining and fine-tuning the neural network without the performance optimizations of the framework, the network is around two orders of magnitude slower to train, completing in 6.5 hours. For the *mixed* data set, it is even faster. This may seem counter-intuitive since the *mixed* data set has several times more training samples. However, the second network is smaller than the first one and less pretraining epochs are performed. Finally, the supervised training is stopped early once a certain goal is reached, therefore less epochs of fine-tuning are performed.

For the System II, a large difference can be observed in training time between the two data sets. First, this is because the data set *V2* is three times smaller than the *mixed* data set. Moreover, on the first data set, only 25 epochs of pretraining are performed compared to 100 for the second data set. Finally, the learning rate was reduced on the second data set and therefore more epochs of training are necessary to reach the training goal.

Overall, it is clear that training convolutional neural networks is much more time-consuming than standard fully-connected neural networks. However, with a carefully optimized implementation, this can still be done in very reasonable time. Moreover, it can also be observed that the cost of pretraining is small compared to the cost of fine-tuning the full network.

The times necessary to compute each step of the detection and recognition of the system are presented in Table 5.3 for the System I on the *V2* data set. On average, it takes about 100 milliseconds to completely detect and recognize a Sudoku in an image. For realistically sized images, this time remains reasonable, but there is still room for improvement. The time necessary to detect and recognize a Sudoku image is largely dominated by the loading and detection of the image. Indeed, the recognition itself only takes about 4% of the complete analysis time. The detection uses several complex image analysis algorithms from the OpenCV library. Moreover, the detection step was not optimized for performance as much as the performance of the classification by the network. Recognition with the system II is slower, with an average of 8732us per image. This is about 2.3 times slower than the first system. Nevertheless, this still remains a small portion of the overall analysis time.

5.8 Summary and potential extensions

This chapter reported results about the experiments in Sudoku detection and recognition in pictures taken with phone cameras in various Swiss newspapers. Since earlier research for this problem was not performed on publicly available data, a new database was created and published: the Sudoku Recognition data set (SRD). This data set is composed of 200 images of Sudoku puzzles. A second version of the data set with fully-filled puzzles is also available.

The detection of the grid and of the digits is performed using several standard image processing techniques. Then, the recognition is handled by two different systems: A fully-connected DBN and a convolutional CDBN. Both systems are pretrained with CD and fine-tuned with mini-batch SGD. On the first data set, both systems were able to perfectly recognize 82.5% of the Sudoku puzzles. On the second data set, the system II has proved more resilient to overfitting, although its optimization has not been trivial, correctly classifying 92.5% of the puzzles, compared to 65% for the first system.

The impact of pretraining the neural network prior to training it for classification has been analyzed. When looking at the convergence of the training error over time, it can be seen that the error curve is much more stable when the network is pretrained. Moreover, when the network is pretrained for sufficient epochs, the training error converges faster. This shows that pretraining does indeed act as a regularizer. Moreover, this also shows that pretraining provides a good initialization of the weights. This second fact can be especially observed in the training of the System II where the training is significantly faster with a pretrained network.

However, although the training is smoother and faster, the final classifier model is not much different with or without pretraining.

All the networks were trained in a small amount of time. The longest training took about 75 minutes for the System II on the second data set. On the other hand, the fastest network was trained in 2.5 minutes with the System I on the same data set. On average, it takes about 100ms to process a complete Sudoku from an image. The recognition is highly optimized and only takes between 4% and 8% of the analysis on average.

Overall, the problem of detecting and recognizing a Sudoku Puzzle on pictures taken from a mobile-based camera in a newspaper is not necessarily an easy task. A large deal of the difficulty lies in the detection of the grid and the digits, impacting the classifier. On the second version of the data set where all cells are filled, the detection is much easier and the results are more tied to the quality of the recognition model. As for the question of whether a single network can learn inputs of two different types, with handwritten and computer-generated digits, it was not as much of a problem as was expected at the beginning, especially with the convolutional network that proved very good at this task.

If this experiment was to be taken further, it would be necessary to improve the detection quality in priority. Indeed, the poor detection results are making the training more difficult, especially on the *V2* data set. Passing the empty cells directly to the recognizer instead of handling them would surely improve the detection results. A full end-to-end approach with CNN would probably be the most efficient technique for this problem, but would require a complete overhaul of the system. Then, it would also be interesting to resolve the overfitting issues by using Dropout and augmenting the training samples by using affine and elastic distortions. An interesting approach would be to use the results of the solver in order to fix some mistakes at the recognition level. Finally, handling more difficult situations like large rotations of the image would also be interesting to investigate.

References for Chapter 5

- Anagnostaras, Patrick (2008). *Sudoclick - automatic recognition of Sudoku grids for mobile phone* (cit. on p. 86).
- Bradski, Gary (2000). “The OpenCV library”. In: *Dr. Dobb’s Journal of Software Tools* (cit. on pp. 72, 91).
- Canny, John (1986). “A Computational Approach to Edge Detection”. In: *IEEE Transactions Pattern Analysis Mach. Intelligence* 8.6, pp. 679–698. ISSN: 0162-8828. DOI: 10.1109/TPAMI.1986.4767851. URL: <http://dx.doi.org/10.1109/TPAMI.1986.4767851> (cit. on p. 90).
- Coates, Adam, Blake Carpenter, et al. (2011). “Text detection and character recognition in scene images with unsupervised feature learning”. In: *2011 International Conference on Document Analysis and Recognition*. IEEE, pp. 440–445 (cit. on p. 90).

- Fletcher, Reeves and Reeves M. Colin (1964). “Function minimization by conjugate gradients”. In: *The Computer Journal* 7.2, pp. 149–154. DOI: 10.1093/comjnl/7.2.149. URL: <http://dx.doi.org/10.1093/comjnl/7.2.149> (cit. on pp. 70, 97).
- Hinton, Geoffrey E. (2012). “A Practical Guide to Training Restricted Boltzmann Machines.” In: *Neural Networks: Tricks of the Trade (2nd ed.)* Ed. by Grégoire Montavon, Genevieve B. Orr, and Klaus-Robert Müller. Vol. 7700. Lecture Notes in Computer Science. Springer, pp. 599–619. ISBN: 978-3-642-35288-1. URL: <http://dblp.uni-trier.de/db/series/lncs/lncs7700.html#Hinton12> (cit. on pp. 29, 68, 73, 93).
- Hinton, Geoffrey E., Simon Osindero, and Yee-Whye Teh (2006). “A fast learning algorithm for deep belief nets”. In: *Neural computation* 18.7, pp. 1527–1554 (cit. on pp. 19, 26, 42, 98).
- Hinton, Geoffrey E. and Ruslan R. Salakhutdinov (2006). “Reducing the dimensionality of data with neural networks”. In: *Science* 313.5786, pp. 504–507 (cit. on pp. 19, 42, 50, 57, 86).
- Impedovo, S, L Ottaviano, and S Occhinegro (1991). “Optical character recognition—a survey”. In: *International Journal of Pattern Recognition and Artificial Intelligence* 5.01n02, pp. 1–24 (cit. on p. 89).
- Jain, Atishay et al. (2013). “Fundamental Challenges to Mobile Based OCR”. In: vol. 2. 5, pp. 86–101 (cit. on p. 89).
- Kamal, Snigdha, Simarpreet Singh Chawla, and Nidhi Goel (2015). “Identification of numbers and positions using MATLAB to solve Sudoku on FPGA”. In: *2015 Annual IEEE India Conference (INDICON)*. IEEE, pp. 1–6 (cit. on p. 89).
- LeCun, Yann, Léon Bottou, et al. (1998). “Gradient-based learning applied to document recognition”. In: *Proceedings of the IEEE* 86.11, pp. 2278–2324. ISSN: 0018-9219. DOI: 10.1109/5.726791 (cit. on pp. 18, 87, 99, 197).
- Lee, Honglak, Roger Grosse, et al. (2009). “Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations”. In: *Proceedings of the 26th Annual International Conference on Machine Learning*. ACM, pp. 609–616 (cit. on pp. 43, 47–49, 69, 99).
- Liang, Jian, David Doermann, and Huiping Li (2005). “Camera-based analysis of text and documents: a survey”. In: *International Journal of Document Analysis and Recognition (IJDAR)* 7.2-3, pp. 84–104 (cit. on p. 89).
- Ly, Tanh Minh and Dung Trung Vo (2015). “A novel and automatic character extraction and recognition for Sudoku puzzle solving”. In: *Advanced Technologies for Communications (ATC), 2015 International Conference on*. IEEE, pp. 546–550 (cit. on p. 88).
- Matas, Jiri, Chris Galambos, and Josef Kittler (2000). “Robust Detection of Lines Using the Progressive Probabilistic Hough Transform”. In: *Comput. Vis. Image Underst.* 78.1, pp. 119–137. ISSN: 1077-3142. DOI: 10.1006/cviu.1999.0831. URL: <http://dx.doi.org/10.1006/cviu.1999.0831> (cit. on p. 90).

- Rasmussen, Carl Edward (2006). *Minimize a differentiable multivariate function, implementation in Matlab*. URL: <http://learning.eng.cam.ac.uk/carl/code/minimize/minimize.m> (cit. on pp. 70, 97).
- Ronse, Christian and Pierre A. Devijver (1984). *Connected Components in Binary Images: The Detection Problem*. New York, NY, USA: John Wiley & Sons, Inc. ISBN: 0-86380-016-5 (cit. on p. 90).
- Shewchuk, Jonathan R. (1994). *An Introduction to the Conjugate Gradient Method Without the Agonizing Pain*. Tech. rep. Pittsburgh, PA, USA (cit. on pp. 70, 97).
- Simha, Pramod, K. V. Suraj, and Tejas Ahobala (2012). “Recognition of numbers and position using image processing techniques for solving Sudoku Puzzles”. In: *Advances in Engineering, Science and Management (ICAESM), 2012*. Nagapattinam, Tamil Nadu: IEEE, pp. 1–5. ISBN: 978-1-4673-0213-5 (cit. on p. 88).
- Suzuki, Satoshi and Keiichi Abe (1985). “Topological structural analysis of digitized binary images by border following.” In: *Computer Vision, Graphics, and Image Processing* 30, pp. 32–46 (cit. on p. 91).
- Van Horn, Adam (2012). *Extraction of Sudoku Puzzles using the Hough transform*. Tech. rep. University of Kansas, Department of Electrical Engineering and Computer Science (cit. on p. 88).
- Wicht, Baptiste and Jean Hennebert (2014). “Camera-based Sudoku recognition with deep belief network”. In: *Soft Computing and Pattern Recognition (SoC-PaR), 2014 6th International Conference of*, pp. 83–88. DOI: 10.1109/SOCPAR.2014.7007986 (cit. on pp. 88, 91, 97).
- (2015). “Mixed handwritten and printed digit recognition in Sudoku with Convolutional Deep Belief Network”. In: *Document Analysis and Recognition (ICDAR), 2015 13th International Conference on*. IEEE, pp. 861–865 (cit. on pp. 86, 88, 97, 101).

Chapter 6

Keyword Spotting

The only problem is time

Seth MacFarlane

Contents

6.1	Introduction	110
6.1.1	Feature learning	111
6.2	Keyword Spotting	113
6.2.1	State of the art	113
6.3	Data sets	116
6.3.1	IAM off-line database	117
6.3.2	George Washington database	117
6.3.3	Parzival database	117
6.4	Reference features	118
6.5	Keyword Spotting System	119
6.5.1	Feature extraction	120
6.5.2	Dynamic Time Warping	122
6.5.3	Hidden Markov Model	123
6.6	Results	125
6.6.1	Experimental Evaluation	125
6.6.2	Dynamic Time Warping	126
6.6.3	Hidden Markov Model	134
6.7	Grayscale images	137
6.8	Efficiency	140
6.9	Summary	142

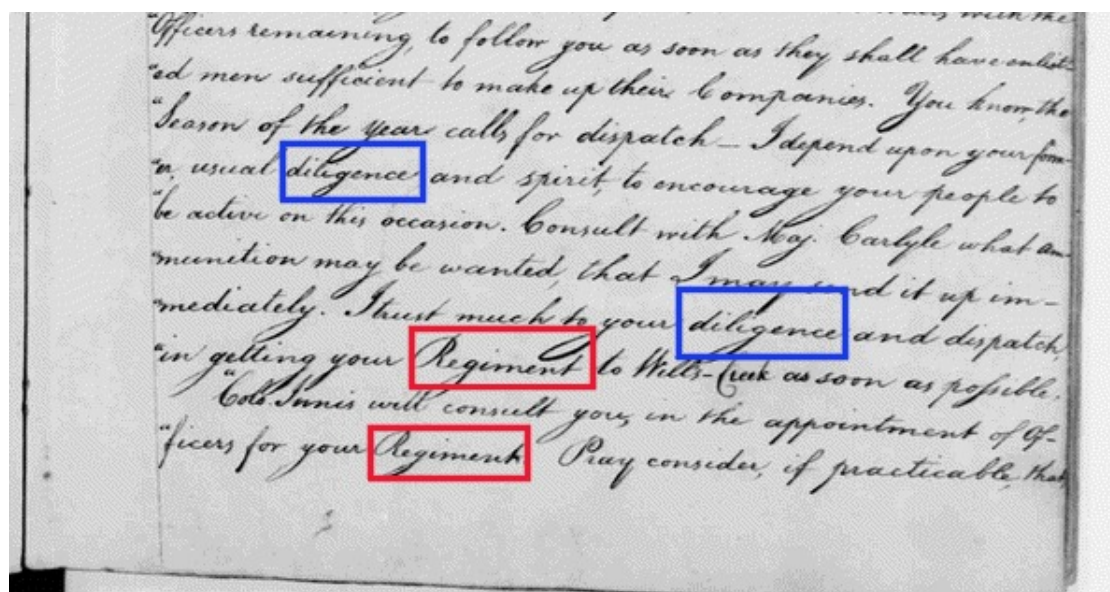


Figure 6.1: Spotting of the keywords Regiment and Diligence in a handwritten letter from George Washington.

6.1 Introduction

Semi-supervised training was covered in detail in Chapter 3 and Chapter 5. With such training scheme, the network is first pretrained using fully unsupervised data. This pretraining step aims at generating a good initial state of the network. It also leverages on the inherent structure of the data to extract features able to reconstruct the input data. The training then uses supervised data to fine-tune the model for its task (classification or regression). With this fine-tuning step, the final model does not directly offer the extracted features anymore and does not directly rely on them. Nevertheless, the structure of the network remains the same and fine-tuned features could still be used for another task. It merely uses the extracted features as a good initialization of the weights to make fine-tuning more efficient. It was also seen in the previous chapters that numerous improvements on neural network training have rendered this pretraining technique less interesting and have made the training of neural networks easier. The present chapter focuses on fully unsupervised feature learning. In that case, the model is trained only to extract features from a data set and is not altered to do another task in a second phase. It means that the model is trained entirely without labels. If classification is the final goal, the features can be directly passed to another model that uses these features instead of directly using the input or some handcrafted features. An advantage of unsupervised feature extraction is that it does not require any supervised data for training. Labeled data being much harder to obtain than unlabeled material, this is a strong advantage. Of course, if classification is the final goal, a classifier would still need to be trained on top of the learned representation and such training would require labeled data.

To test how the features learned automatically with a Restricted Boltzmann Machine (RBM) or a Convolutional Restricted Boltzmann Machine (CRBM) could be used in an image analysis problem, it was decided to focus on handwritten keyword spotting on historical documents. Figure 6.1 shows a simple example of word spotting. Keyword spotting consists in finding occurrences of a given word in a corpus of handwritten pages, without transcribing them. This task was chosen for several reasons. First, there exist numerous data sets for this task. Moreover, it is not a trivial task and there still is a lot of research going on in this field. Then, the classifiers used to perform keyword spotting from the extracted features are quite different, ranging from very simple template matching system to complex Recurrent Neural Network (RNN) architectures. Finally, most of the research in this field is using handcrafted features especially designed for the task. Since there are several available feature sets for this problem that are known to be working well, this is a good opportunity to compare handcrafted features with features automatically learned in an unsupervised manner. This is also a good task to evaluate the performance of the learned features that can be used by different classifiers and how difficult it is to tune the features for the keyword spotting task.

This chapter describes the results that were obtained for this experiment. First, the task of keyword spotting in handwritten historical documents is presented in detail. Then, the state of the art for this task is explored. After that, the three different data sets that have been used in our experiments are presented. The following section describes the reference feature sets that are used to compare our learned features against existing handcrafted features. Then, the complete system is described, from the feature extraction to the classification with either of the two selected classifiers. The overall results are then presented in detail for each of the classifiers and system optimization is discussed. Finally, the summary provides conclusions as well as an outlook of some possible improvements for this experiment.

6.1.1 Feature learning

Extracting features from images can be done in two manners. First, the most classical way is to design features especially tuned for the task and sometimes even tuned for a specific data set. These features are called *handcrafted features* since an algorithm was designed manually to extract them, incorporating a priori information on the specificities of the data. Some of the handcrafted features are very simple, while some more recent feature sets are very complex and generally highly-dimensional. The second category of features consists in automatically learning features from the images using machine learning. This is a solution that is used more and more since the advent of Deep Learning. These features are called *learned features*.

Handcrafted features have several disadvantages.

1. They require expert knowledge of the data and of the problem. Few hand-

crafted features can be applied to many different data sets for the same problem. They often need to be tuned specifically for each data set.

2. They require time-consuming human-tuning of the features. Developing a set of features for a data set is not trivial and may take a large amount of time, even for experts of the field.
3. They do not generalize well to other data set showing new characteristics. While they generally perform very well on the data sets for which they were designed, slight changes in the data set may lead to poor performance.

There are many techniques to learn features from a data set, either supervised learning or unsupervised learning (Bengio, A. Courville, and Vincent, 2013). Historically, supervised learning was used to learn a feature extractor, either with simple dictionary learning or with an Artificial Neural Network (ANN). There are several techniques than can be used for this task, K-means (Lloyd, 1982), Principal Component Analysis (PCA) (Hotelling, 1933), auto-encoders (Bengio, 2009) and the family of RBM models, on which this experiment focuses. Since the advent of Deep Learning, unsupervised learning has been used more and more to extract features from data, especially from images. A comparison of the efficiency of some of these models is presented in (Coates, H. Lee, and Ng, 2010).

Learned features are trying to overcome the issues of handcrafted features. Since they can be trained on any data set, they generalize very well to change. Moreover, they are generally able to cope well with unknown examples, due to the higher generalization capabilities of neural networks. Finally, they do not require expert knowledge of the data and should not require a lot of human-labour time. On the other hand, they also have some disadvantages:

1. They do require expert knowledge of the system that is used to learn them. Neural networks can sometimes be very complicated to train and unsupervised training may reveal more complicated to manage than standard classification training.
2. They may take a large amount of time to train. Some very large models need several days for training on large data set, where handcrafted features do not need to be trained.
3. The system to extract features needs to be trained on each data set. It generally means that using a new data set or a new format of data needs a full retraining of the model, or at least an adaptation of the model using new data to complement the old model. This requires some time for training and some knowledge of the training system.
4. They may be slow to compute. Depending on the complexity of the network, it may be time-consuming to extract features from the images. They are faster than some complex handcrafted features but are generally slower than simple ones.

5. They are often complex. Sometimes they exhibit a very visual nature, but it is not always possible to understand the features from an human point of view, whereas handcrafted features are, most of the time, designed to "make sense".

Overall, the research reported in this chapter is attempting to verify the hypothesis that learned features are superior to handcrafted features in terms of task performance. For fair comparison, we selected the state of the art handcrafted features on the task of handwritten keyword spotting and compare them to our learned features.

6.2 Keyword Spotting

Keyword spotting, or word spotting, consists in retrieving information from documents based on a keyword query. The query can be done by-example by providing an image of the searched keyword or by-string by providing the searched keyword itself. This research focuses on query-by-example (QBE). It can also be done by retrieving lines containing the keyword rather than retrieving the occurrence of the word, this has the advantage of not necessarily requiring word segmentation of the lines (Fischer, Keller, et al., 2012). This experiment focuses solely on handwritten documents (letters and historical documents). However, word spotting can be applied to computer-written documents as well: The same keyword spotting task exists in other domains such as speech or video signal, where the search is performed in the set of documents without transcribing the documents.

For this research, the query is a template image of the searched keyword. For each available word image, the word spotting system must take the decision of whether the image contains the word from the query. In typical systems, the decision is taken according to a threshold over some measure of distance $ds(\mathbf{X}, \mathbf{K})$ between the image \mathbf{X} and the template image of the keyword \mathbf{K} :

$$ds(\mathbf{X}, \mathbf{K}) < \mathbf{T} \quad (6.1)$$

In practice, T will be chosen depending on whether precision or recall is more important. It is also possible to use a different threshold for each keyword rather than a global threshold.

6.2.1 State of the art

Keyword spotting was first proposed for speech analysis (Rose and Paul, 1990; Myers, Rabiner, and Rosenberg, 1980). It was then adapted for poorly printed documents a few years later (Chen, Wilcox, and Bloomberg, 1993; Kuo and

Agazzi, 1994) and finally for handwritten text analysis (Raghavan Manmatha, C. Han, and Riseman, 1996). It is still a widely researched problem.

Keyword spotting solutions can be separated in two families. First, *template-based* methods are using a template image for the query keyword and comparing this template with each word image of the document. These techniques are generally very simple, but do have the advantage that it is easy to get template images. Moreover, they do not require any knowledge of the language being processed. On the other hand, spotting out-of-vocabulary keywords is not possible, since a template image is necessary for each keyword to be spotted. The second family of models, *learning-based* systems, are using machine learning techniques to learn word models. They generally exhibit better generalization and are able to spot arbitrary keywords. However, they need labeled data and a certain knowledge of the underlying alphabet. They are also generally more complex models and require some training time. Both families of models require some form of features to perform keyword spotting.

6.2.1.1 Template-based

Template-based approaches are computing a distance between the query image and the current image being inspected. Early works were based on computing a single distance between two images. For instance, in (Raghavan Manmatha, C. Han, and Riseman, 1996), the Scott And Longuet-Higgins (SLH) distance (Scott and Longuet-Higgins, 1991) was used. Using some global characteristics of the image, new features have been devised. In (Zhang, Srihari, and Huang, 2003), several binary features were explored and (Bhardwaj, Jose, and V. Govindaraju, 2008) used the moments from binary images to form higher-level features.

While global features bring some interesting results, most of the recent and most successful work is based on local features. In that case, features are computed in some subpart of the image and then these subparts are compared between the two images. The complete feature vector for an image is computed by concatenating the local features. These kinds of distances are generally more robust. Several strategies for locality of features have been devised. For instance, (Leydier et al., 2009) used gradient angle features combined with some form of elastic matching and (Rothfeder, Feng, and Toni M. Rath, 2003) used features computed at the border of the images. One very successful combination is the use of a local sliding window with Dynamic Time Warping (DTW) (Toni M. Rath and Raghavan Manmatha, 2003). The sliding window is producing the locality in the features. Since then, the use of DTW has become a very well established technique and can be used with several different features, such as closed contours (Adamek, O'Connor, and Smeaton, 2007) or word profiles (Tony M. Rath and Rudrapatna Manmatha, 2007).

Recently, local gradient features, similar to Histogram of Oriented Gradient (HOG) features have been very successful for keyword spotting (Rodriguez and Perronnin, 2008; Terasawa and Tanaka, 2009). In (Kovalchuk, Wolf, and Dershowitz, 2014),

a very large feature descriptor is created from HOG and Local Binary Pattern (LBP) features. This large set is then pooled to a size that is the same for each word and can be compared by a simple euclidean distance. Howe used energy minimization scoring on top of a special inkball model to perform spotting (Howe, 2013). Almazan et al. developed a special feature set than can be applied to both the template images and the query strings (Almazán et al., 2014). The proposed feature set, named Pyramid Histogram of Oriented Characters (PHOC), makes it very easy to perform keyword spotting by a simple nearest neighbour technique. Another very compact and efficient representation was proposed by (Rusiñol et al., 2015) allowing segmentation-free spotting. Retsinas et al. used Projection of Oriented Gradients (POG) features with a direct feature comparison to achieve very successful results (Retsinas et al., 2016).

6.2.1.2 Learning-based

Historically, all learning-based approaches were learning word-level models. By using the posterior probabilities of a Hidden Markov Model (HMM) (Raghavan Manmatha, C. Han, and Riseman, 1996) proposed a general approach for word spotting. Several similar approaches were later proposed, for instance using symmetric half plane HMM in (Choisy, 2007) or using Fischer kernels to replace the posterior probabilities in (Perronnin and Rodriguez-Serrano, 2009).

One drawback of word-level models is that they are unable to spot out-of-vocabulary words. This is why several works are investigating character-level models, that are more robust and from which it is easy to create word-level models for classification. At first, character generalized Hidden Markov Model (gHMM) were trained using character template images (Forsyth et al., 2005; Chan, Ziftci, and Forsyth, 2006). A similar approach was used in (Cao and V. Govindaraju, 2007) with Gabor features for each character. However, it is not trivial to extract such character images from historical handwritten documents.

Since the lexicon is known, character models can be learned without segmentation from word-level or line-level models. From these models, word-level and filler models can be composed to create a robust scoring method. An earlier work (El-Yacoubi, Gilloux, and Bertille, 2002) used this approach for an address reading task. This was applied again for unconstrained keyword spotting in (Thomas, Chatelain, Heutte, and Paquet, 2010). These techniques are segmentation-free and the model is trained on complete text lines. These HMM approaches have the disadvantage of needing a lexicon. An improvement over this was proposed in (Fischer, Keller, et al., 2012), not comparing against other lexicon words but using a character-based confidence model to perform scoring. Another approach using RNNs at character level with Long Short Term Memory (LSTM) was proposed (Frinken et al., 2012), outperforming both HMM and DTW on a similar set of features. HMM have also been successfully combined with deep neural networks (Thomas, Chatelain, Heutte, Paquet, and Kessentini, 2015).

Recently, more end-to-end approaches have been developed. One technique is to

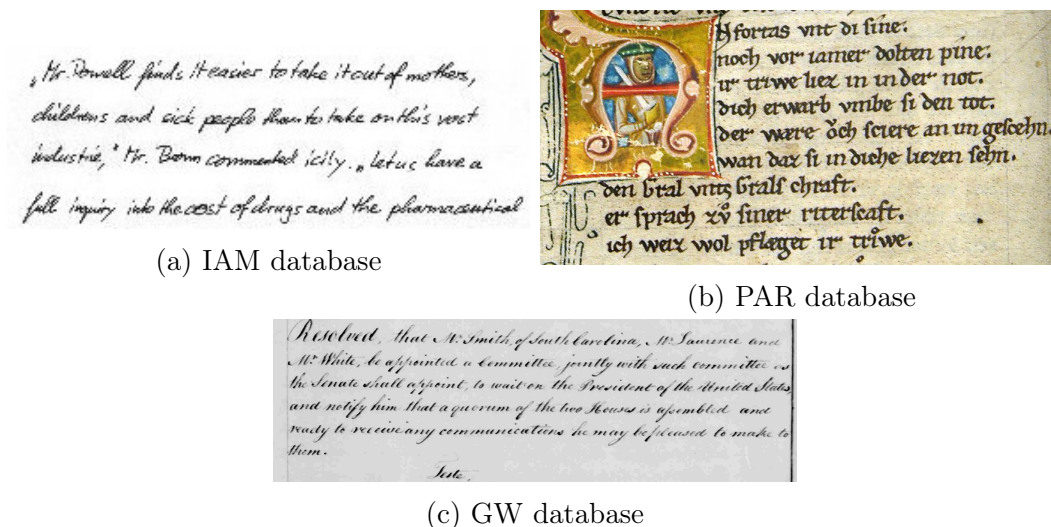


Figure 6.2: Extracts from exemplary pages from the three data sets used for keyword spotting.

use a large Convolutional Neural Network (CNN) pretrained on existing data set with many images such as ImageNet and then adapted for the keyword spotting data set (Sharma and Sankar, 2015). Another technique, designed for text spotting in natural images, was presented in (Jaderberg, Vedaldi, and Zisserman, 2014) where a sequence of CNN using maxout units is used to extract features from small images. Sudholt et al (Sudholt and Fink, 2016) used a deep CNN to estimate PHOC features. Similar results were achieved by combining the text and image embeddings into one convolutional representation (Krishnan, Dutta, and Jawahar, 2016).

Generally, learning-based models can be used with arbitrary feature sets. Therefore, most of the feature sets presented in the Section 6.2.1.1 can also be used with the models presented in this section.

6.3 Data sets

For evaluating the features automatically learned using the proposed system, three different data sets have been selected: two single writer data sets (PAR and GW) and one multiple-writer (IAM). The data sets are presented in detail in the following sections. Figure 6.2 shows examples of parts of pages from the three data sets.

The proposed evaluation does not directly use the original data sets, but uses the versions made available by (Fischer, Keller, et al., 2012). These versions already contain normalized and segmented line images and word images. Using this prepared data set allows us to insulate our analysis out of the segmentation task, making the assumption of a perfect segmentation.

Data set	Language	Writers	Pages	Lines	Words
GW	English	2	20	656	4894
PAR	Ancient German	3	47	4477	23478
IAM	Modern English	657	1539	13353	115320

Table 6.1: Statistics on the entire data sets used for keyword spotting.

6.3.1 IAM off-line database

IAM is a data set collected by the Institute of Computer Science and Applied Mathematics (IAM) from the University of Bern in Switzerland. It contains modern English handwriting works. The writers were told to reproduce text coming from the Lancaster-Oslo-Bergen (LOB) corpus (Johansson, Leech, and Goodluck, 1978). This data set contains very different writing styles and all the text has been written on paper with contemporary pens. Figure 6.2a shows one example from IAM.

The IAM data set is large, with 1539 pages and 657 writers. Table 6.1 gives some statistics about this set. The layout of this data set is significantly simpler than historical documents, its difficulties lie in the difference in writing styles. Moreover, the data set was collected so that the test set, the validation set and the training set each contain text from different writers. This makes it necessary to be able to work very efficiently with unknown writing styles.

6.3.2 George Washington database

The George Washington (GW) data set (Lavrenko, Toni M. Rath, and Raghavan Manmatha, 2004) is a collection of letters written by George Washington and one of his associates, in English. It is written with ink, on paper. The secondary writer has been trying to mimic the writing style of George Washington, therefore the two writing styles are very close and this data set is often considered a single-writer data set. Figure 6.2c presents one example from this set.

This is a very small data set, with only 20 pages of relatively short letters. More statistics about this data set are available in Table 6.1. The layout of the pages is quite simple. Aside from the text, the pages are containing page numbers, some rulers and signatures. Due to its small size, experiments on this data set were done on four different cross-validation sets. When single results are presented, they are averaged over the four validation runs.

6.3.3 Parzival database

The Parzival (PAR) data set (Fischer, Wüthrich, et al., 2009) contains the poem Parzival, written by Wolfram von Eschenbach, in the 13th century, in medieval German. It is written with ink on parchment. The copy that is considered here

is the copy held in Switzerland, by the Abbey Library of Saint Gall. While it has been written by three different writers, the styles are very similar. Moreover, one of the writer largely dominates the others in terms of pages. Therefore, the data set is also considered as single-writer data set. An exemplary image is shown in Figure 6.2b.

This data set contains 47 pages of manuscript. More statistics are available in Table 6.1. The style is very rich and contains ornaments and marginal notes. The front pages have been slightly degraded by time, with small holes in the parchment, large stains and wrinkles. Moreover, there are also a few lines that are overlapping when they contain long descenders or ascenders characters.

6.4 Reference features

For evaluating the performance of the automatically learned features, three different features sets were selected for comparison. These feature sets were selected because they are known to work well for keyword spotting and they have all been tested on at least one of the selected data sets. The reference feature sets are:

- `Marti2001` from (Marti and Bunke, 2001): This set of features is well-known in the field of handwriting and has been used repeatedly and successfully. It is a very simple set of nine heuristic features, extracted at each column of the image. There are three global features: the percentage of black pixels, the center of gravity and the second order moment. The remaining six features are local: the position of the lower and upper contours, the gradients of both contours, the number of black to white transitions and the percentage of black pixels between the two contours. These features are lacking in robustness because they are only local to each column, aside from gradients from column to column.
- `Rodriguez2008` from (Rodriguez and Perronnin, 2008): This set of features is inspired from the SIFT descriptor. It uses local gradient histogram features with overlapping horizontal sliding window. The sliding window is applied from left to right, on every column of the image. The window is then separated into a grid. Histograms of gradient orientations are accumulated in each cell of the grid. The features at each of the position are normalized so that the components of each cell sum to one. Since each window is split in a 4x4 grid and each cell has 8 bins of gradients, there are 128 features for each column of the image. These features have an important advantage over the column features of Marti since they have a much larger context for each features. There still is features for each column, but these features are depending on a significantly larger context, making the features significantly more robust.
- `Terasawa2009` from (Terasawa and Tanaka, 2009): This set of features is very similar to `Rodriguez2008` but makes several changes. First, the

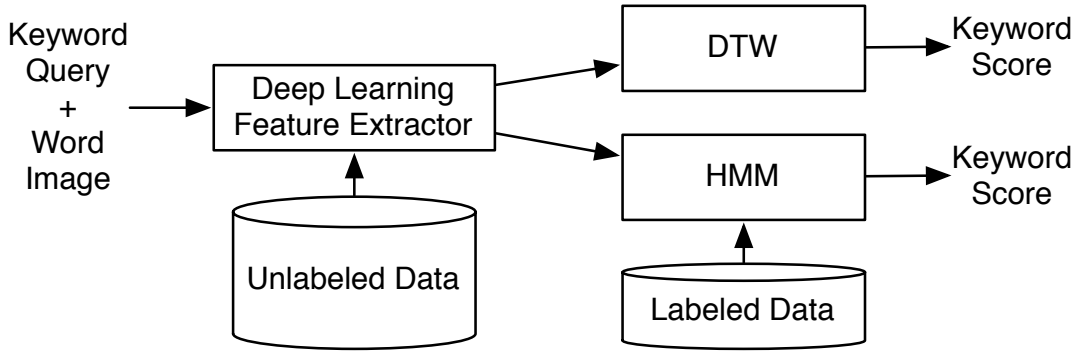


Figure 6.3: The Keyword Spotting system developed during this research. The system extracts features directly from pixels and passes them to a DTW or an HMM classifier to perform keyword spotting.

sliding window is significantly narrower (4x2 blocks). On top of that, there is no horizontal overlapping inside the window, only vertical overlapping, this means that the width of the block is restricted to that of the window. The horizontal overlapping is achieved by the sliding window itself. Unlike the SIFT reference, the signed gradient of orientation is used in place of the unsigned gradient. They are also using more bins, 16 bins for each cell. In total, there are 196 features for each column of the image. These features also have the advantage of robustness because of the large context for each feature column.

The implementations have been reproduced inside our keyword spotting system. The parameters for each implementation have been extracted from the original research papers.

6.5 Keyword Spotting System

An overall view of the proposed Keyword Spotting system is given in Figure 6.3. The feature extractor is trained using the available unsupervised data for each training data set. Once trained, it is able to generate features from any image. Then, these features are passed to the classifier. The system was tested using two different classifiers. The first system, DTW, is a pure template-based classifier that does not require any training and as such the complete system does not require any labeled data, making it a hybrid model (part template-based and part learning-based). The second classifier that was used, HMM, is trained to recognize word models, using the extracted features. The classifier is only responsible for scoring each of the images using its own dissimilarity measures. From this, Equation 6.1 is used to decide whether the image contains the keyword query, independently of the classifier.

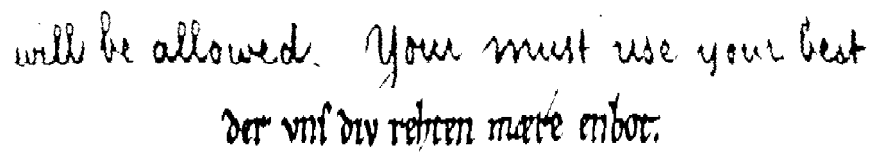


Figure 6.4: Two images of preprocessed lines from the GW (top) and PAR (bottom) data sets.

When using DTW as the classifier, the system needs an example image for the query, while it does only need a word transcription (in the language sense) when using HMM. Both systems have advantages and disadvantages, but using these two systems is interesting to observe how we can learn features generic enough to be used by two very different classifiers. It is also interesting to see how much impact better features can have on different classifiers.

Our evaluations solely focus on spotting at the word-level, thus requiring segmentation of the text lines into words. Line spotting can be performed at the line-level with DTW, but still requires word segmentation. However, when using HMM as the classifier, the model can easily be adapted in order to perform line spotting without requiring any segmentation into words, as shown in (Fischer, Keller, et al., 2012).

This section describes the feature extraction system that is used to get features from the image. Then, both classifiers are described into more details in the following sections.

6.5.1 Feature extraction

Features are extracted in two steps. In a first time, the images are preprocessed in order to increase the accuracy of keyword spotting. Then, features are extracted from each image using a sliding window technique and a neural network that is learning to extract features from the images in an unsupervised manner.

6.5.1.1 Preprocessing

The presented system works on binary word images. A simple global threshold is used to binarize the images. On some of the data sets, a local edge enhancement technique has also been performed (Fischer, Indermühle, et al., 2010). The line images are then segmented into word images. Finally, the system normalizes all the word images to remove the skew of the text by rotation and its slant using a shear transformation. The height of the images is also normalized with respect to the upper and lower baselines. Finally, the width of the lines is normalized with respect to its number of letters. (Marti and Bunke, 2001) describes this process into more details. These steps are actually already performed in the images from the benchmark data sets that we used. Therefore, segmentation errors are not

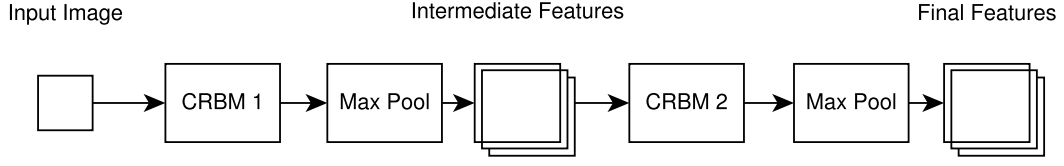


Figure 6.5: The Convolutional Deep Belief Network used for feature extraction in the Keyword Spotting System.

taken into account for the presented results. These are the base images that will be presented to the reference systems. Figure 6.4 shows two examples of preprocessed lines from the data sets. These preprocessed images are also used for each reference feature set, so this step is completely independent of the system being tested.

For the feature extractor, the height of the entire image is scaled down by a factor of three. This is not performed for the reference handcrafted features that take as input the uncompressed images.

6.5.1.2 Feature Extraction

To extract and learn the features, the proposed system uses a horizontal sliding window of width W and of the same height H as the original compressed image. The sliding window moves, from left to right, one pixel at a time, over the entire image. For the windows that are intersecting with the borders of the image, outside pixels are considered to be background pixels, set to pixel values of zero. Overall, from an $N \times H$ image the system will extract N patches of dimensions $W \times H$.

The model used for feature extraction is presented in Figure 6.5. The feature extractor system is only presented with these patches, it does not have any knowledge of the entire image. From this collection of images, distributed into a training set, a validation set and a test set, a Convolutional Deep Belief Network (CDBN) model is trained to extract features. The model is trained using Contrastive Divergence (CD), in a completely unsupervised procedure. Moreover, the training is only done layer by layer, not through the entire model. The model is composed of two CRBM models stacked together. To improve translation invariance and the robustness of the features, each CRBM layer is followed by a Max Pooling layer. This also has the advantage of reducing the size of the output feature map, which is important for some classifiers. Each pooling layer shrinks the representation by a factor C in each dimension. In this work, the pooling is non-overlapping, namely, the stride is always equal to the pooling factor ($S = C$). Using a standard max pooling instead of using Probabilistic Max Pooling (PMP) means that top-down inference through the model will not be possible. However, PMP is not defined for units other than binary, so standard max pooling was used. This also has the advantage of efficiency because max pooling is a very simple operation.

An image \mathbf{X} is transformed into N patches \mathbf{x}_n extracted by the same sliding window

procedure. Given the trained CDBN, each patch is then passed through each layer of the network to compute its activation probabilities. The features of each patch are then concatenated for the entire image:

$$f(\mathbf{x}) = [\text{CDBN}(\mathbf{x}) \text{ activation probabilities}] \quad (6.2)$$

$$F(\mathbf{X}) = [f(\mathbf{x}_1), f(\mathbf{x}_2), \dots, f(\mathbf{x}_N)] \quad (6.3)$$

The classifiers are then using $F(\mathbf{X})$ as input to compute their dissimilarity measure.

6.5.2 Dynamic Time Warping

Dynamic Time Warping (DTW) is a technique used to find an optimal alignment between two sequences of potentially different length. The two sequences are warped non-linearly so that they match each other. For each point in one sequence, DTW will find a corresponding point in the other sequence.

DTW was first introduced to solve speech problems (Myers, Rabiner, and Rosenberg, 1980) and has since also been very well-established in the field of keyword spotting (Toni M. Rath and Raghavan Manmatha, 2003; Tony M. Rath and Rudrapatna Manmatha, 2007). It has been used successfully with many different types of features (Rodriguez and Perronnin, 2008; Terasawa and Tanaka, 2009; Wicht, Fischer, and Hennebert, 2016b). Interestingly, it also has been combined directly with a neural network (Iwana, Seiichi, and Frinken, 2016).

Before the DTW distance is computed, the features are normalized so that each feature vector has zero-mean and unit variance. The cost of an alignment is the sum of the $d(\mathbf{x}, \mathbf{y})$ distances of each aligned pair. This system uses the squared Euclidean as the distance measured $d(\mathbf{x}, \mathbf{y})$:

$$d(\mathbf{x}, \mathbf{y}) = \sum_{i=1}^N (\mathbf{x}_i - \mathbf{y}_i)^2 \quad (6.4)$$

The DTW distance $D(F(\mathbf{X}), F(\mathbf{Y}))$ of two feature vector sequences $F(\mathbf{X})$ and $F(\mathbf{Y})$ is given by the minimum alignment cost, found by dynamic programming. For speeding up the process and improving the results, a Sakoe-Chiba band (Sakoe and Chiba, 1978) is used. This constrains the search of the path to be within a band of a certain width around the shortest path, as illustrated in Figure 6.6. The final distance is normalized with respect to the warping path (length of the optimal alignment). When several occurrences of the keyword are available in the training set, the example that minimizes the distance for the currently tested image is selected. The distance over the $F(\mathbf{X})$ function (see Equation 6.3) is used as the final dissimilarity measure between a word image \mathbf{X} and a keyword string \mathbf{K}

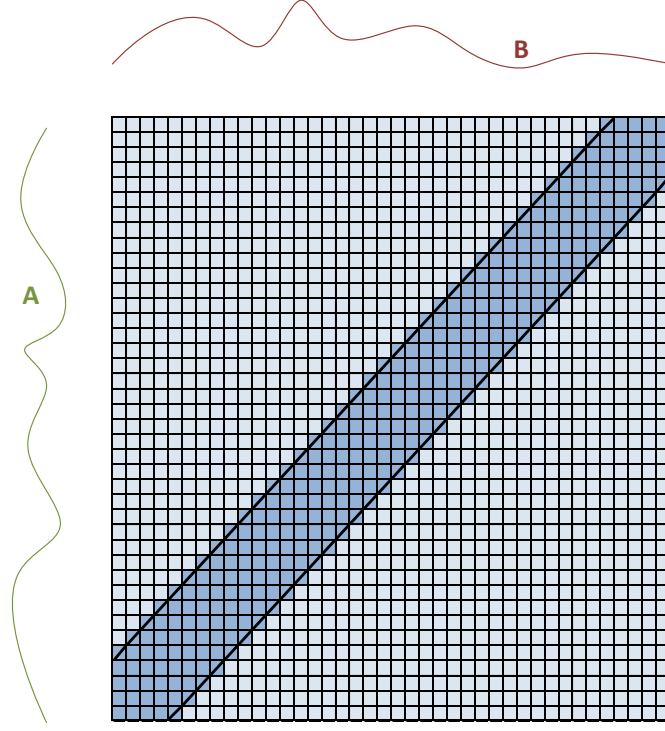


Figure 6.6: Sakoe-Chiba band constraint for Dynamic Time Warping. The warping path is constrained to the darker zone. Each point of the sequence A will be matched to a point in the sequence B.

$$ds(\mathbf{X}, \mathbf{K}) = \min_j^N D(F(\mathbf{X}), F(\mathbf{T}_j)) \quad (6.5)$$

using each of the N available template image $\mathbf{T}_1, \dots, \mathbf{T}_n$ of the keyword \mathbf{K} . Since it is a simple problem to solve, the DTW implementation has been implemented directly in our experiment.

6.5.3 Hidden Markov Model

A Hidden Markov Model (HMM) is a statistical model, principally used for modelling sequences of features that show a state-based nature. The sequential nature of speech and handwriting makes it a very good candidate for these kinds of problems. It models the probability distribution of the features over a series of consecutive observations. This model has been introduced in a series of papers between the late 1960s and the early 1970s by Baum et al. (Rabiner, 1989). HMMs have been used repeatedly and successfully for modeling handwritten text (Plötz and Fink, 2009) and for keyword spotting too (Choisy, 2007; Chan, Ziftci, and Forsyth, 2006; Rodriguez and Perronnin, 2008; Terasawa and Tanaka, 2009).

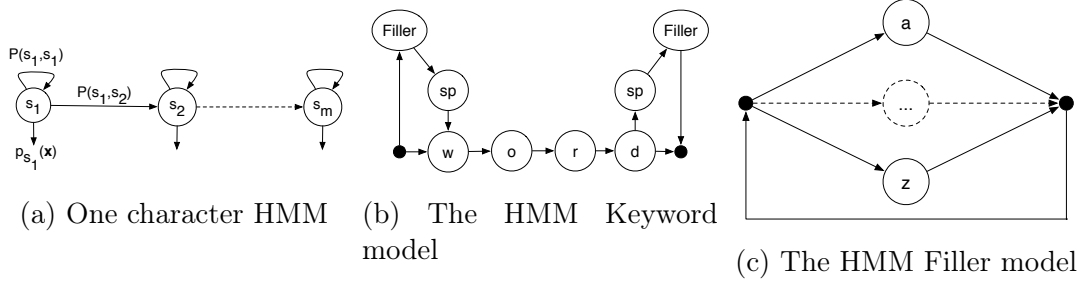


Figure 6.7: The Hidden Markov Models (HMMs) used for keyword spotting.

The technique described in (Fischer, Keller, et al., 2012) is the one that was selected as the base for an HMM classifier. The difference between the two classifier is that this experiment performs word spotting while the original performs line spotting, but this only slightly changes the model. Figure 6.7 presents the different models used for keyword spotting. The system trains character HMMs for each character contained in the available training set. Each model is trained using the Baum-Welch algorithm (Rabiner, 1989). Once each character HMM is trained, they can be used for recognition using the following technique. For an input keyword query, a keyword model is created by aggregating character HMMs. Using the Viterbi algorithm (Viterbi, 1967), this model can be used to compute the log-likelihood score of an input image, $\log p(\mathbf{X}|\mathbf{K})$, looking at the best path of state sequence in the HMM. This will represent the probability of the given image being the word query. Another model, the filler model, is constructed from the character models in order to model an arbitrary sequence of characters. Again using Viterbi, this second model is used to compute a score, $\log p(\mathbf{X}|F)$. This score is representing the general conformance of the image compared to the trained word models and acts as a regularizer. This will allow to normalize the word score with respect to the writing style, allowing to compare scores directly and improving generalization to unknown handwriting styles. To achieve this, log-odds scoring (Barrett, Hughey, and Karplus, 1997) is used, subtracting the filler score from the word score. The final score is also normalized with respect to the length of the keyword (in pixels) N_K , resulting in the following final dissimilarity measure:

$$ds(\mathbf{X}, \mathbf{K}) = \frac{\log p(\mathbf{X}|\mathbf{K}) - \log p(\mathbf{X}|F)}{N_K} \quad (6.6)$$

This is then used directly for keyword spotting. The HMM models and training algorithms provided by the HTK toolkit ¹ have been used for this experiment.

¹<http://htk.eng.cam.ac.uk>

6.6 Results

Following the same experimental evaluation, the results are presented separately for the two classifiers that are used in the overall system. The various parameters that have been necessary to tune to improve the results are also presented for each system.

6.6.1 Experimental Evaluation

For performance evaluation, two different scenarios are considered. The precision of the system is the metric that is measured. In both cases, the evaluation uses the complete test set of each data set. First, in the *local* scenario, each keyword has an associated *local threshold*. This scenario measures the Mean Average Precision (MAP) of the spotting system. The second scenario is the *global* scenario in which there is only one *global threshold*. This is to measure the Average Precision (AP) of the system. The `trec_eval` software system² is used to compute both values (Fischer, Keller, et al., 2012). The thresholds are never set explicitly, `trec_eval` is handling this as well. Both scenarios could be used in practice. The local scenario can use a global threshold for out-of-vocabulary words and the local thresholds can be tuned on existing data sets to achieve better performance.

For both scenarios, the precision and recall of the system are also computed. These values can be obtained using the number of false negatives (FN), false positives (FP) and true positives (TP). Since the values will depend on the threshold being select, they are defined as threshold functions. Using these values, the recall $R(T)$ and precision $P(T)$ can be computed as follows:

$$R(T) = \frac{TP(T)}{TP(T) + FN(T)} \quad (6.7)$$

$$P(T) = \frac{TP(T)}{TP(T) + FP(T)} \quad (6.8)$$

and presented in the form of a recall-precision curve, once averaged over all queries. These curves are also computed separately in the two different scenarios. These results are also generated with `trec_eval`.

The source code for the implementation of the proposed system as well as the implementation of the reference feature sets is available online³. This also includes code for evaluation with DTW and HMM.

²http://trec.nist.gov/trec_eval

³https://github.com/wichtounet/word_spotting

Table 6.2: Mean Average Precision (MAP) and Average Precision (AP) for the different features when using DTW as classifier. The best feature set is indicated in bold. The absolute and relative improvements and error reduction of the proposed feature set over the best baseline is also mentioned.

	GW		PAR		IAM	
System	AP	MAP	AP	MAP	AP	MAP
Marti2001	40.11	55.74	63.73	60.39	5.00	19.89
Rodriguez2008	37.84	64.72	57.69	49.33	0.79	9.48
Terasawa2009	42.97	66.37	69.72	74.10	0.33	6.01
Proposed	53.71	67.26	70.77	75.11	0.1	2.62
Abs. Improvement	10.74	0.89	1.05	1.01	-	-
Rel. Improvement	24.99%	1.34%	1.50%	1.36%	-	-
Error Reduction	18.83%	2.64%	3.46%	3.89%	-	-

6.6.2 Dynamic Time Warping

First the results of the evaluation with the DTW classifier are discussed. The overall results are presented in Table 6.2 for each data set. The very poor performance shown on the IAM data set with either the reference features or the proposed feature set are explainable. Indeed, this data set is peculiar in that none of the training writing style are present in the test set. In that case, basic template matching strategies are known to have difficulties (Fischer, Keller, et al., 2012). Therefore, a performance comparison in that case would not be conclusive.

The proposed system outperforms every baseline on every data set. Since *Terasawa2009* is always the best of the baseline, the performance of the proposed system will only be compared to it. On the GW data set, there is a very significant improvement in performance in the global scenario performance, resulting in almost 25% improvement or a reduction of the error of 18%. On the other hand, the improvement on the local scenario is not really significant. On the PAR data set, the system reduced the error by about 3.5% against the best baseline. This may not be a very large reduction, but the performance of the second baseline is already quite good. On the IAM data set, as mentioned, the results are not conclusive with DTW.

Overall, even with a basic classifier such as the DTW, the features are exhibiting excellent performance. Especially, the learned features are more stable from one data set to another compared to the reference features which shows quite low performance on the global scenario of the GW data set, while being very good on the global scenario of the PAR data set. This demonstrates one advantage of the learned features that can be trained on different data set to improve the results. Moreover, the excellent generalization on the global scenario on GW tends to confirm this point.

For more details, Figure 6.8 presents the performance on the GW data set in the form of Recall-Precision plots. The results for the PAR data set are shown in

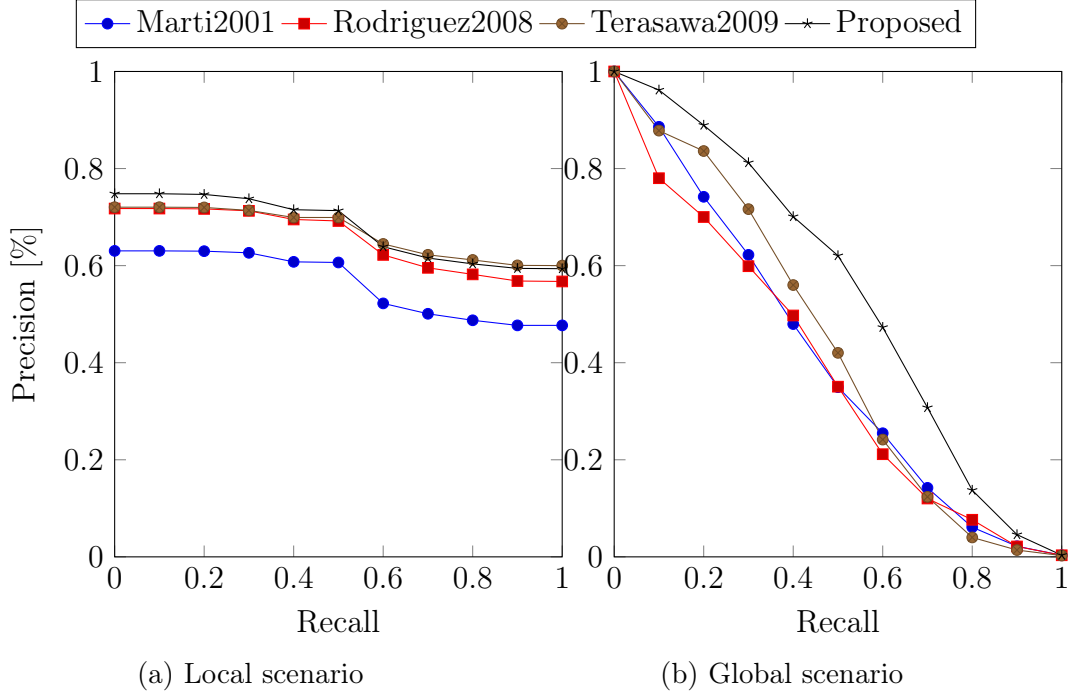


Figure 6.8: Recall-Precision performance of the Keyword Spotting System, with DTW, on the GW data set.

Figure 6.9. This confirms the MAP and AP results obtained before. Indeed, a significant difference can be observed in the global scenario of GW. However, on the other cases, the proposed system is just slightly ahead of *Terasawa2009*.

While the performance achieved by the proposed system is good, its optimization has not been trivial. Indeed, the system has many training parameters and the architecture itself can be highly customized. Moreover, the DTW being a very simple classifier, it proved very sensitive to the output of the system, rendering the task of system optimization even more complex. The following sections are detailing the principal points of optimization that were made in order for the system to work as well as presented. They also describe the final optimal parameters that were used.

6.6.2.1 Architecture

The first and most important aspect of system optimization was to select a correct architecture for the feature extraction system.

The number of layers was the first parameter to tune. A network with one layer had two disadvantages for the purpose of feature extraction. The generated features were not high-level enough to distinguish different patches. Moreover, with only one layer, too many features were generated by the network. This is an issue with DTW for which having too many features has a negative impact on performance. From the beginning, it was necessary to concentrate the research on

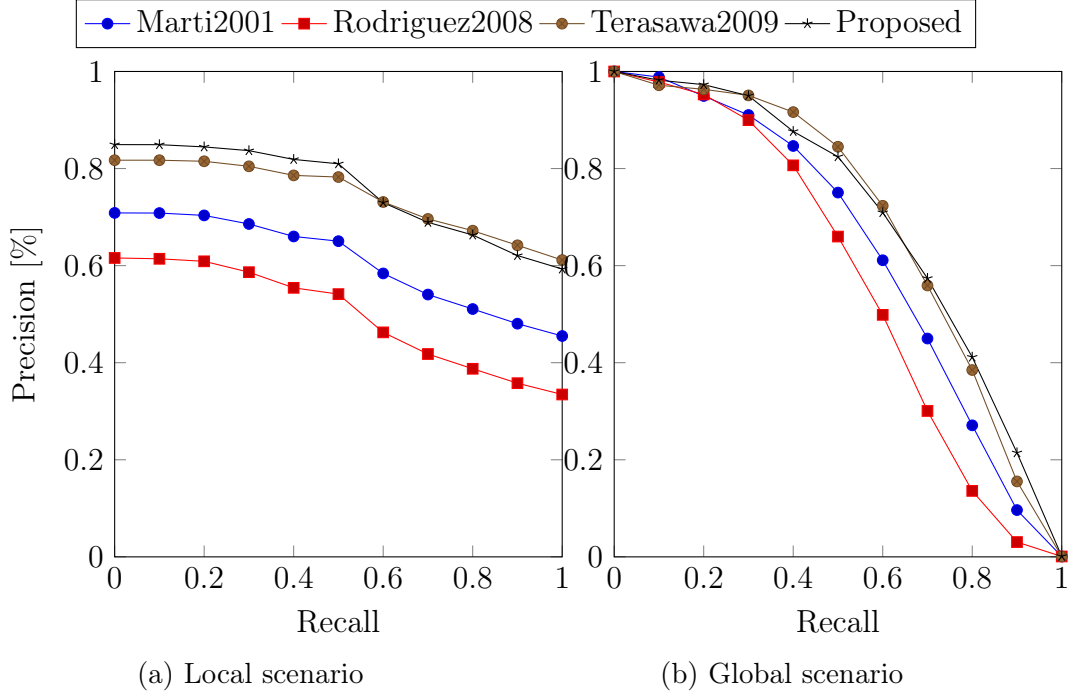


Figure 6.9: Recall-Precision performance of the Keyword Spotting System, with DTW, on the PAR data set.

networks yielding low numbers of output features in order for the DTW results to be adequate. Networks with three layers were also investigated, but the inputs are not complex enough for such a network to learn relevant features. Indeed, the network is only learning on a small patch of the image. Moreover, if an image patch goes through three convolutional layers and three pooling layers, the end result would be a very small feature map.

The size of the convolutional filters was also optimized. Several configurations yielded almost similar results. In every case, the second layer used 3×3 filters. It was the best configuration for the second layer in every tested configuration. For the first layer, 9×9 , 7×7 and 5×5 filters yielded approximately the same results, in order of preference. Moreover, 9×9 reduced the size of the image more than other configuration and thus improved the runtime performance of the overall system. Therefore, it was decided to keep these filter shapes for each network.

The stride of the sliding window was important but has few opportunities for optimization. In practice, only strides of one and two pixels performed well. When the stride was increased more than that the performance decreased very quickly. The reason is that the stride reduces the number of output feature maps linearly. It was found that the length of a word is in itself a very important feature for keyword spotting. Therefore, diminishing this length by more than a factor of two was too much for the performance of the system. Since a stride of one pixel performed slightly better overall than a stride of 2, it was kept at 1 for each data set.

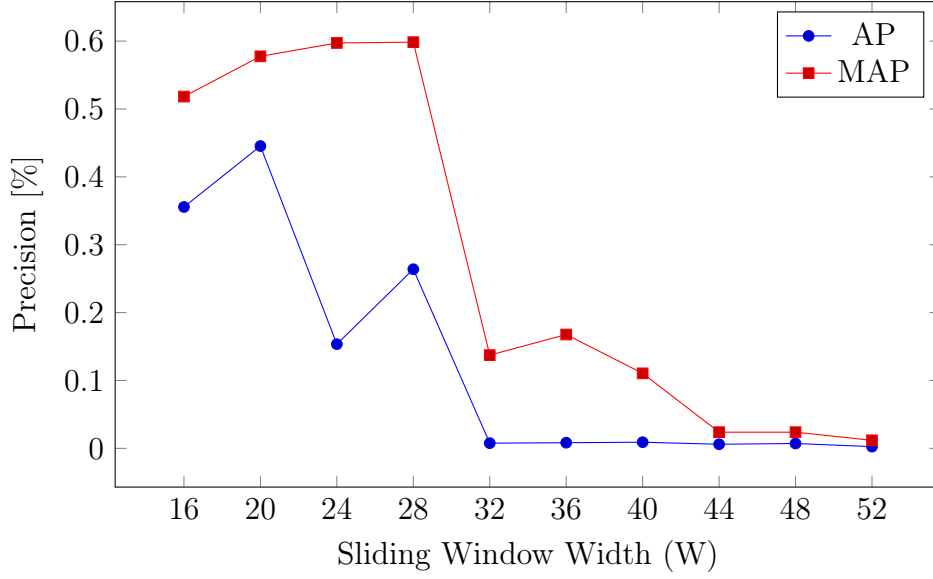


Figure 6.10: Keyword spotting on the George Washington dataset with different widths of the sliding window.

Another very important parameter was the width of the window on which the network is trained. This parameter has several implications. First, it indicates how much context is given to the feature extractor. Indeed, the feature extractor only see information about the current window and learns features inside this window. A larger window would give more context to the feature extraction. Since padding in the borders is used, the width of the window does not impact the number of output feature maps. However, larger windows also increase the amount of overlap since the window is only moved one pixel at a time. Finally, the width of the window is directly related to the size of one output feature map. It is not directly possible to choose any size since it must be handled by the two convolutional layers and the two pooling layers. It would be possible to ease this restriction by using padding in the convolutions and pooling operations, but it was decided not to use it for ease of implementation. The values compatible with our system are the numbers multiple of 4, starting from 16.

Figure 6.10 shows the AP and MAP performance of the system using different widths for the sliding window. The differences are very important between the different configurations. When considering only the MAP, the best width would be of 28 pixels, with values between 16 and 24 producing good results. From 32 pixels and wider windows, the MAP performance starts to get very low. However, fewer configurations are producing good AP results. Indeed, only 16 and 20 window widths are performing adequately. For windows larger than 20 pixels, the AP performance decreases very quickly, reaching almost zero at 32. These results are similar when using the other data sets. In some cases, a window of 16 pixels is more adequate, but overall, 20 pixels is the window size that produces the best results. Therefore, the window width was set to $W = 20$ for all configurations. Interestingly, this is approximately the width of a character in the images.

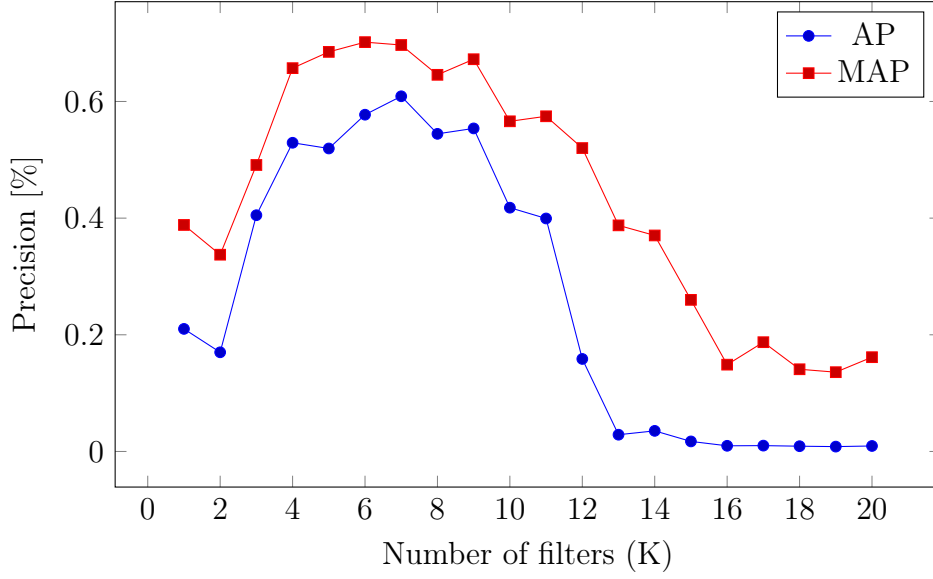


Figure 6.11: Keyword spotting on the George Washington dataset with different numbers of convolutional filters in each layer.

Finally, once all the other architecture parameters are set, what remains is the number of filters of the network. This parameter defines the number of different convolutional filters that are learned on the images. It is directly related to the learning capacity of the network. Figure 6.11 shows the performance achieved with different numbers of filters in the convolutional layers. There are very significant differences in performance between the different values of K . For the tested data set, the best values are between 5 and 11. After 11, the performance decreases very quickly. The Average Precision is almost 0 starting from $K = 13$ and the Mean Average Precision also decreases quickly. It can be seen on this data set that $K = 7$ is the parameter that maximizes both the AP and MAP performance. On the PAR and IAM data set, the same phenomenon can be observed but slightly more to the right. Indeed, it was found that $K = 12$ is the best configuration for the other two data sets.

6.6.2.2 Types of units

Several types of units can be used in a CRBM model. Moreover, contrary to standard neural networks, an RBM has visible and hidden units and these two units can be of different types. In this particular case, the input patches are binary, therefore, the visible units of the first CRBM are best kept binary (using the logistic sigmoid as activation function).

For the hidden units, there are several possible choices. The basic idea is to use binary hidden units, which are generally working adequately in almost all situations. Another possibility is to use Gaussian units, however, they are generally best used as visible units and they can make the training highly unstable. Another

Parameter	Layer 1	Layer 2
Epochs	10	10
Learning Rate ϵ	1×10^{-4}	1×10^{-6}
Momentum α	0.9	0.9
L2 decay λ	2×10^{-4}	5×10^{-4}
Batch Size (GW)	128	128
Batch Size (PAR/IAM)	256	256
Clip Gradients	5.0	5.0

Table 6.3: Training parameters for the feature extraction system for keyword spotting, used for Contrastive Divergence training.

solution is to use Rectified Linear Unit (ReLU) units as hidden units. When ReLU units are used as hidden units, it is still possible to use binary visible units in the second layer if the activation probabilities are sparse enough, otherwise Gaussian or ReLUs must be used as visible units in the second layer.

In our case, the visible units are always binary visible units for both layers. This activation function is very stable and very easy to learn. Originally, binary hidden units were used for the GW data set and ReLU for the larger data sets (Wicht, Fischer, and Hennebert, 2016b; Wicht, Fischer, and Hennebert, 2016a). It is difficult to train ReLU on the small number of images of the GW data set. Therefore, binary hidden units were used. In that case, it was important to enforce the units to have very sparse activation. Without sparsity, the results were significantly worse. The simple comparison done in the DTW is only well suited to sparse units. For this, the method described in (H. Lee, Ekanadham, and Ng, 2008) was used and performed very well for this experiment. However, better results were then achieved using capped ReLU (Krizhevsky, 2010). These units are capped to some values which help make them sparser and makes the training of the network much more stable for large networks and especially when large amounts of data is available. The final network uses RELU-1 units in the first layer and RELU-6 units in the second layer. These units proved to generate robust features and were easily handled by the DTW. Moreover, the training with capped units was significantly more stable than when using standard ReLUs.

6.6.2.3 Training parameters

A CDBN system has many training parameters and there are many possible refinements that can be used during training. Importantly, the training parameters are independent from one layer to another and while some of them can be kept the same, it has been necessary to tune some of the parameters for each layer. Moreover, some of the parameters have much more impact than others.

Table 6.3 shows the different training parameters that were used to train the proposed system. Each layer is trained for 10 epochs of CD. On the GW data set, 10 epochs proved the best around all cross validation sets. Some of the sets were

demonstrating better results with slightly fewer epochs, but on average over the four cross validation sets, 10 epochs was the best. On the larger data sets, fewer epochs can be used to save some time, because of the very large number of patches that are extracted, but going to 10 epochs did not hurt the performance and was kept the same for all data sets for sake of convenience. The learning rate ϵ was the most important parameter to tune for this experiment. It was necessary to use a smaller learning rate for the second layer because of the much larger number of input channels. Momentum was necessary to use to stabilize the training and make it faster. Both the learning rate and the momentum α were kept constant during the entire duration of training. L2 weight decay has been used on all the weights, but not on the biases, in order to help generalization. A higher weight cost λ is used on the second layer to make learning more stable. A weight decay rate of 0.9 proved efficient for both layers. To avoid some instability in learning ReLU, the gradients were clipped to 5.0 in both layers. This was especially interesting in learning for the larger data sets.

The only training parameter changing between the data sets is the batch size. A batch size of 256 was used for the PAR and IAM data sets that are significantly larger than the GW data set for which the batch size of 128 was used. This does not make a significant difference in spotting performance, but does speed up the training significantly.

6.6.2.4 Normalization

Since the DTW classification is performed by comparing the features of two images using the Euclidean distance, it is very important to ensure that the features are as much comparable as possible. If features are of different scales, the overall distance will not be meaningful.

The most important optimization is to scale each feature in a standard range. While some work such as (Fischer, Keller, et al., 2012) uses linear scaling to scale the final features in the range $[0, 1]$, it was found out in this experiment that scaling each feature so that they have zero-mean and unit-variance was more efficient (called mean scaling here). This feature scaling is performed just before passing the features to the DTW operator. This feature scaling is also performed for the reference feature sets. It was in fact more important for the reference feature set because the features are less sparse and more variable in range than the feature learned by the proposed system.

Before the features are scaled and classified with DTW, the features \mathbf{x}_n of each patch are normalized using a variant of L2 normalization:

$$\mathbf{x}_n = \frac{\mathbf{x}_n}{\sqrt{\sum_{n'}^N (\mathbf{x}_{n'} + \mathbf{x}_{n'}) + e^2}} \quad (6.9)$$

This is the equivalent of dividing the feature vectors by their L2-norm. This

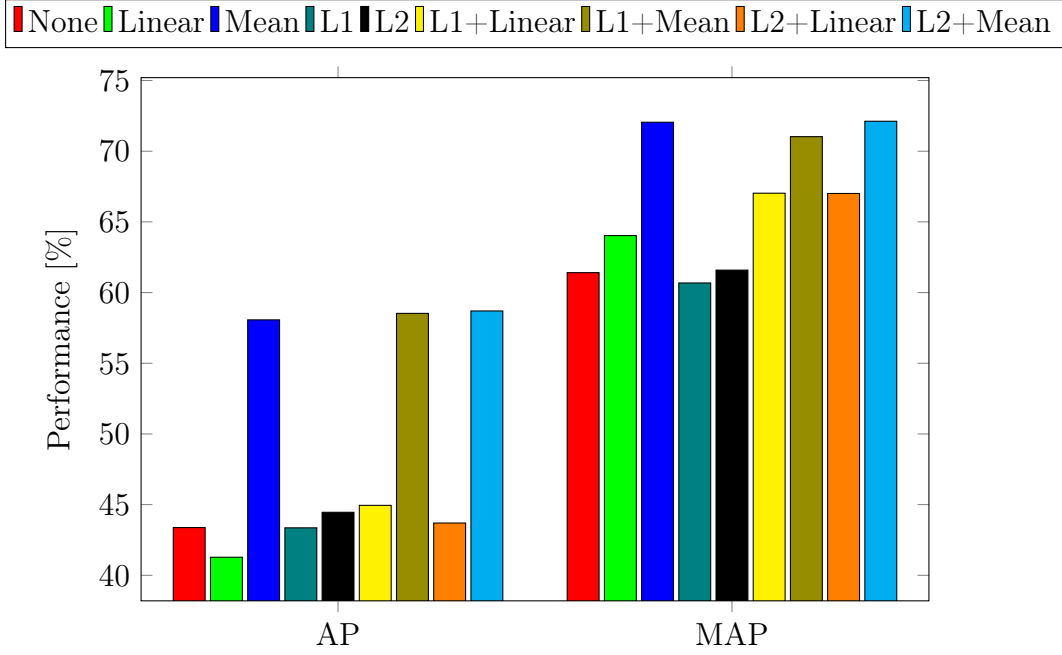


Figure 6.12: Impact of normalization on the keyword spotting features with the DTW classifier, on the GW data set

follows the block normalization techniques seen in HOG (Dalal and Triggs, 2005). The variable e should be set to a small constant, it was set to 16 in the present experiment, and the exact value should not, hopefully, be important. This is only performed for the proposed features and not for the reference features which are already normalized by nature. L1-normalization was also experimented with:

$$\mathbf{x}_n = \frac{\mathbf{x}_n}{\sum_{n'}^N \mathbf{x}_{n'}} \quad (6.10)$$

Several techniques and combinations of techniques have been tested on the GW data set. Figure 6.12 shows the results with different normalization schemes on the GW data set. The configurations are easy to test since the model does not need retraining. The results have been obtained by using the best trained model for all tests and changing only the normalization process before the DTW classification is performed.

It is interesting to note that the differences are more important in the global scenario (the AP measure) than in the local scenario. Indeed, this is especially showing that mean scaling of the features improves very significantly the results of the global scenario. This can be explained as AP uses a global threshold for which a normalization is expected to help smoothing out local variabilities. On the local scenario, mean scaling is still significantly better than linear scaling, but the differences are smaller. While there is a slight difference between L1 and L2 normalization, it is very small. Although adding L1 and L2 normalization to linear

Table 6.4: Mean Average Precision (MAP) and Average Precision (AP) for the different features when using HMM as classifier. The best feature set is indicated in bold. The absolute and relative improvements and error reduction of the proposed feature set over the best baseline is also mentioned.

	GW		PAR		IAM	
System	AP	MAP	AP	MAP	AP	MAP
Marti2001	66.59	81.97	85.03	89.57	52.50	69.92
Rodriguez2008	32.65	59.52	9.29	19.30	4.20	18.61
Terasawa2009	73.02	83.23	91.27	91.88	57.17	72.58
Proposed	73.17	87.90	92.19	94.40	64.57	74.49
Abs. Improvement	0.15	4.67	0.92	2.52	7.4	1.91
Rel. Improvement	0.20%	5.61%	0.99%	2.66%	12.94%	2.63%
Error Reduction	0.55%	27.84%	11.77%	45.0%	20.88%	7.48%

scaling improves the results over simply linear scaling, the difference is not significant over mean scaling. However, when considering all the data sets and all the cross validation sets, a slight improvement was found when using L2 normalization plus mean scaling when compared to simply mean scaling, therefore, it was kept in the final system. Since the focus of the work was not on normalization, other forms of normalization such as using the features of the entire data set or of the entire page have not been explored.

6.6.3 Hidden Markov Model

The results using the HMM classifier are discussed here. The character HMMs used for evaluation of the proposed features use 3 Gaussian mixtures per state for the GW data set, 5 for PAR and 7 for IAM. For maximum performance, the technique from (Günter and Bunke, 2003) has been followed, computing the number of states for each character model from the mean width of the letters in the images.

The overall results with an HMM classifier are presented in Table 6.4 for each data set. The reason for the very low performance of the *Rodriguez2008* features with the HMM classifier has not been found. Nevertheless, since *Terasawa2009* is always better with DTW classifier and has excellent performance with the HMM classifier as well, it has been considered as the best baseline and the proposed features are compared against it.

From the results, it is clear that HMM is a much better classifier for word spotting than DTW. Indeed, except for *Rodriguez2008*, the results with all the feature sets are significantly better than with DTW. Moreover, there are also less differences between the features of *Marti2001* and *Terasawa2009*. This probably comes from the fact that the original HMM system was developed and tuned for *Marti2001* features (Fischer, Keller, et al., 2012). Although the results of the best baseline are already very good, results from the proposed system are better in every situation.

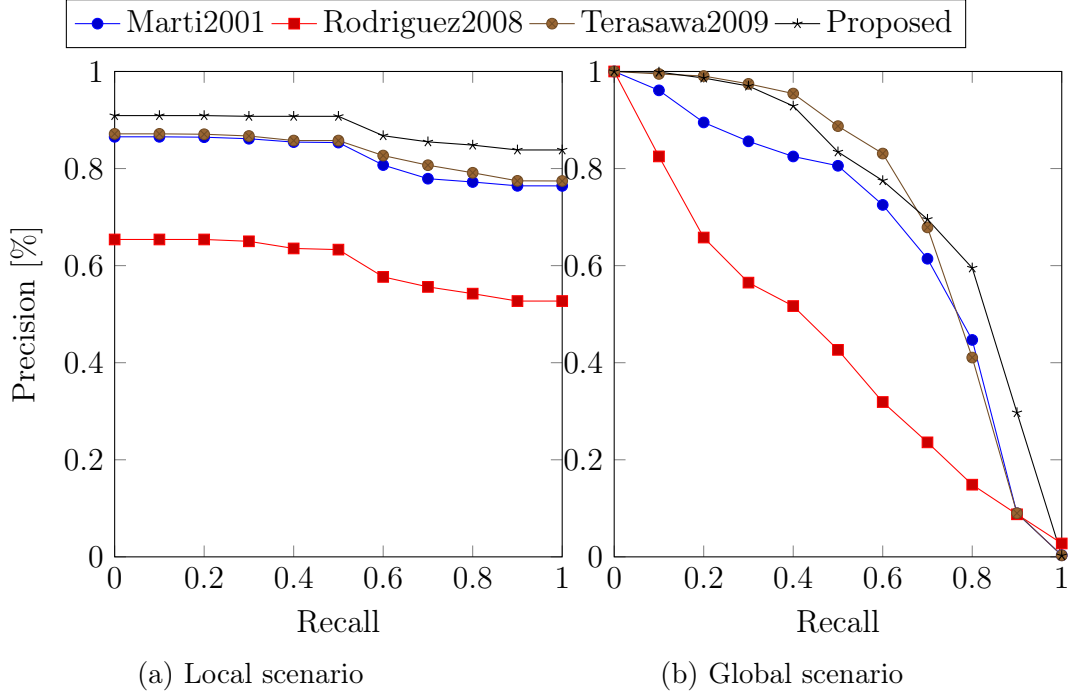


Figure 6.13: Recall-Precision performance of the Keyword Spotting System, with HMM, on the GW data set.

Since the accuracies are high, it is more interesting to look at the relative error reduction rather than the improvement in accuracy. Except for the local scenario on GW, the error reductions of the proposed system are significant. In the best case, in the global scenario of the PAR data set, the error is reduced by 45%. On the GW and PAR data sets, the improvements are larger for the local scenario. On the IAM data set, the error is reduced more in the global scenario than in the local one. The system is proving to be able to generate good features for HMM.

For more details, Figure 6.13 presents the results for the GW data set in the form of Recall-Precision plots. The plots for PAR and IAM data sets are shown in Figure 6.14, respectively Figure 6.15. These curves confirm the overall results. It is clear that the *Rodriguez2008* features have some issues with HMM. It is also clear that the features are very close together in terms in performance. Looking especially at the PAR results, the proposed system learns features that are very good for both scenarios, with a small advantage over the second baseline. Even the very simple features from the *Marti2001* baseline are doing very well with this classifier, not being too inferior to *Terasawa2009*.

Overall, very few optimizations of the parameters were done for the HMM classifier. Indeed, the parameters obtained after the extensive tuning of the parameters for DTW have been working very well. Moreover, since it was a goal not to tune the parameters independently for each classifier and since DTW is less efficient than HMM, fine-tuning the system for DTW was a better choice than fine-tuning it for HMM. This is showing that the learned features can indeed be used by two

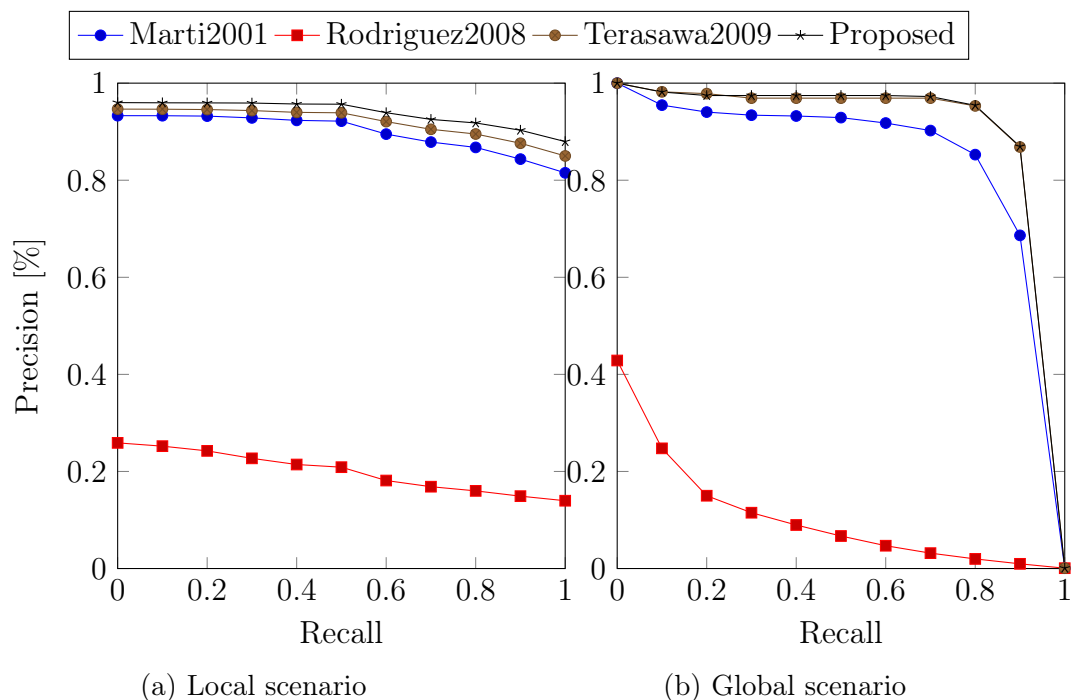


Figure 6.14: Recall-Precision performance of the Keyword Spotting System, with HMM, on the PAR data set.

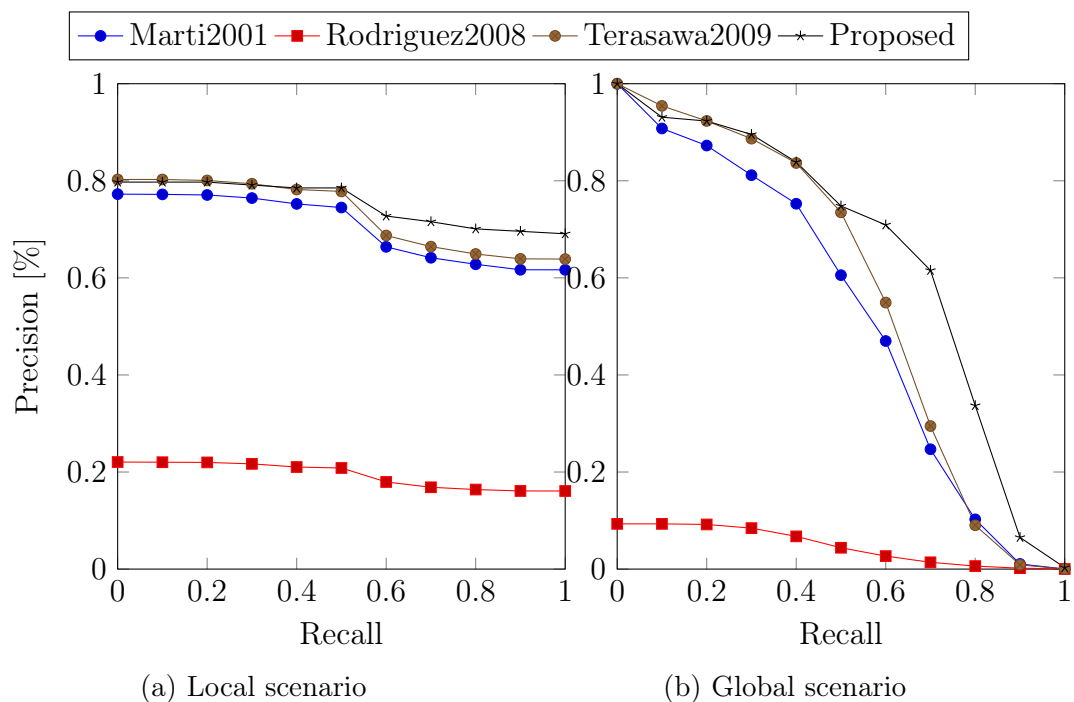


Figure 6.15: Recall-Precision performance of the Keyword Spotting System, with HMM, on the PAR data set.

very different classifiers without tuning for both of them. The only thing that was tuned was the number of Gaussian mixtures for each model. Numbers of Gaussian mixtures from 1 to 10 were evaluated on each data set and 3, 5 and 7 were selected for the GW, PAR and IAM data sets, respectively. It is highly possible that better results could be obtained with HMM if the parameters of the feature extractor model were tuned accordingly and independently of DTW.

6.7 Grayscale images

All the results presented in this chapter have been obtained on normalized binary images from the documents. However, binarization of historical documents is not trivial and may result in a loss of information that could potentially be used by a feature extractor. This section presents results on how the proposed system works using grayscale images.

Ideally, the grayscale images would be directly extracted from the source images. However, there does not exist any version of the used data sets for which grayscale images are also available. The first experiment that was realized was to use the word locations available in the Histogram data base (Stauffer, Fischer, and Riesen, 2016) to extract the words from the George Washington data set. However, the word locations from this data set proved of low quality, being very coarse. Indeed, even when most of the noise was removed, the images normalized in height and binarized with Wolf and Jolion advanced technique (Christian, Jolion, and Chassaing, 2002), the results with any of the reference feature sets were very poor.

Since no grayscale images of sufficient quality were available for any of the data sets, it was decided to generate grayscale images from the normalized binary images. For this, a Gaussian Blur was applied to each word image before it was cut into patches, artificially creating grayscale images. For this, a Gaussian kernel with a standard deviation of 2.0 in both directions was applied.

To handle grayscale values in the feature extractor and learn from them, there are two main solutions. If the values are scaled down to the $[0, 1]$ range, they can be considered as probabilities and be handled by binary visible units. Nevertheless, better results are generally observed when using Gaussian visible units. For a noise-free reconstruction, the inputs are normalized to have zero mean and unit variance. This was confirmed in these experiments where Gaussian visible units proved more efficient than binary visible units. Gaussian visible units were also used in the second layer for a small improvement in performance, but binary visible units could have been used as well. When using Gaussian visible units and ReLU in the layers, it was necessary to reduce the learning rate ϵ of the first layer by two orders of magnitude (from 1×10^{-4} to 1×10^{-6}). With greater learning rates, the training diverged very quickly. Indeed, with Gaussian visible units, there is no upper bound to the components of the reconstruction, leading to instability in training, requiring smaller learning rates. The learning rate of the second layer was already small enough (1×10^{-6}) to handle Gaussian visible units and was not

Table 6.5: Mean Average Precision (MAP) and Average Precision (AP) for the system with grayscale images, with both classifiers. The best feature set is indicated in bold. The absolute and relative improvements and error reduction of the proposed feature set over the system with binary images.

	GW (DTW)		GW (HMM)	
System	AP	MAP	AP	MAP
Marti2001	40.11	55.74	66.59	81.97
Rodriguez2008	37.84	64.72	32.65	59.52
Terasawa2009	42.97	66.37	73.02	83.23
Proposed (Binary)	53.71	67.26	73.17	87.90
Proposed (Grayscale)	53.96	68.34	76.55	86.68
Abs. Improvement	0.25	1.08	3.38	-1.22
Rel. Improvement	0.46%	1.58%	4.4%	-1.4%
Error Reduction	1.46%	3.41%	14.41%	-9.2%

changed. Even with the smaller learning rate, the training with Gaussian visible units and ReLU hidden units is very unstable. Even when increasing the number of epochs, we observed significant differences between different trainings.

The results obtained with this system when using DTW as classifier are presented in Table 6.5. The Recall-Precision curves for this case are shown in Figure 6.16. The improvements for the global scenario are small, around 1.5% of error reduction. On the local scenario, the improvements are more significant with around 3.4% error reduction. When observing the results of the local scenario in the Recall-Precision curves, it can be seen that most of the benefit is obtained by extra precision on the tail of the curves. On the global scenario, the difference is much smaller when observing the curves.

When using HMM as a classifier (results in the last two columns of Table 6.5), the results are more mixed. Indeed, in the global scenario, the error is reduced by more than 14%. However, in the local scenario, the error is increased by about 9%. The Recall-Precision curves for these scenarios are shown in Figure 6.17. The difference can be clearly observed. Indeed, the system with grayscale images performs significantly worse on the local scenario, while a significant increase in performance can be observed in the global scenario.

Overall, there is some potential in using grayscale images rather than binary images. Nevertheless, it would be more interesting to use the real grayscale images rather than artificially creating them from binary images. Indeed the mitigated results may also comes from the fact that it is difficult to recreate an information that was discarded during the binarization process. However, the lack of segmented grayscale word images makes this harder to achieve. Finally, the instability on learning with Gaussian visible units and ReLU hidden units makes it difficult to learn a good model.

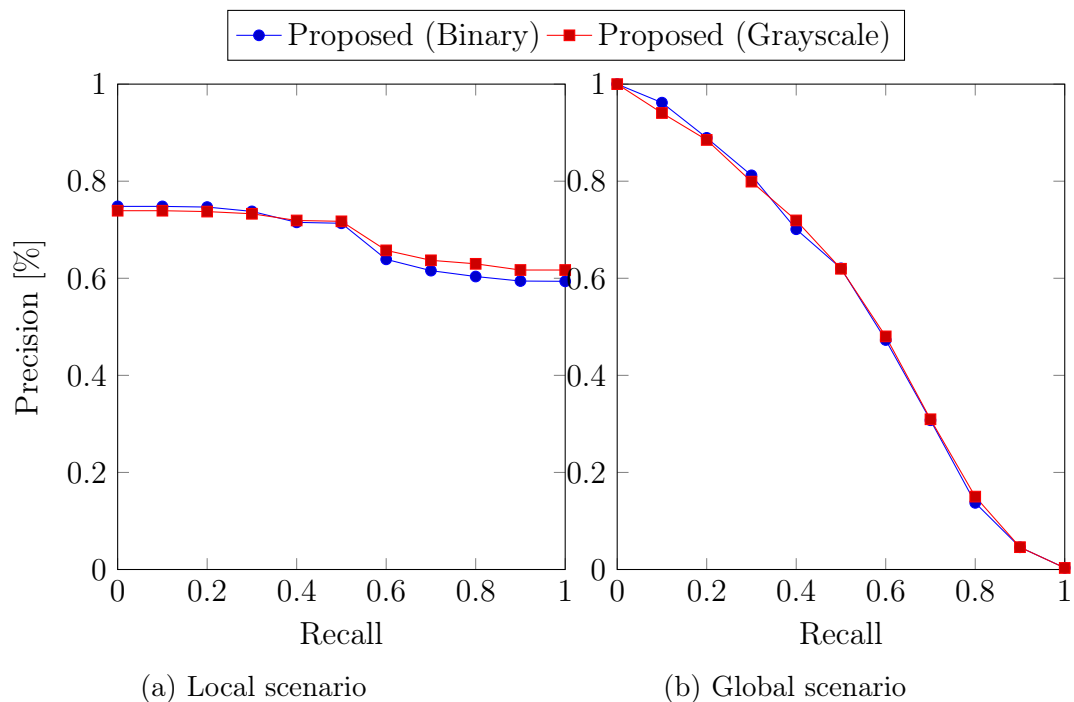


Figure 6.16: Recall-Precision performance of the Keyword Spotting System, using grayscale images, with DTW, on the GW data set.

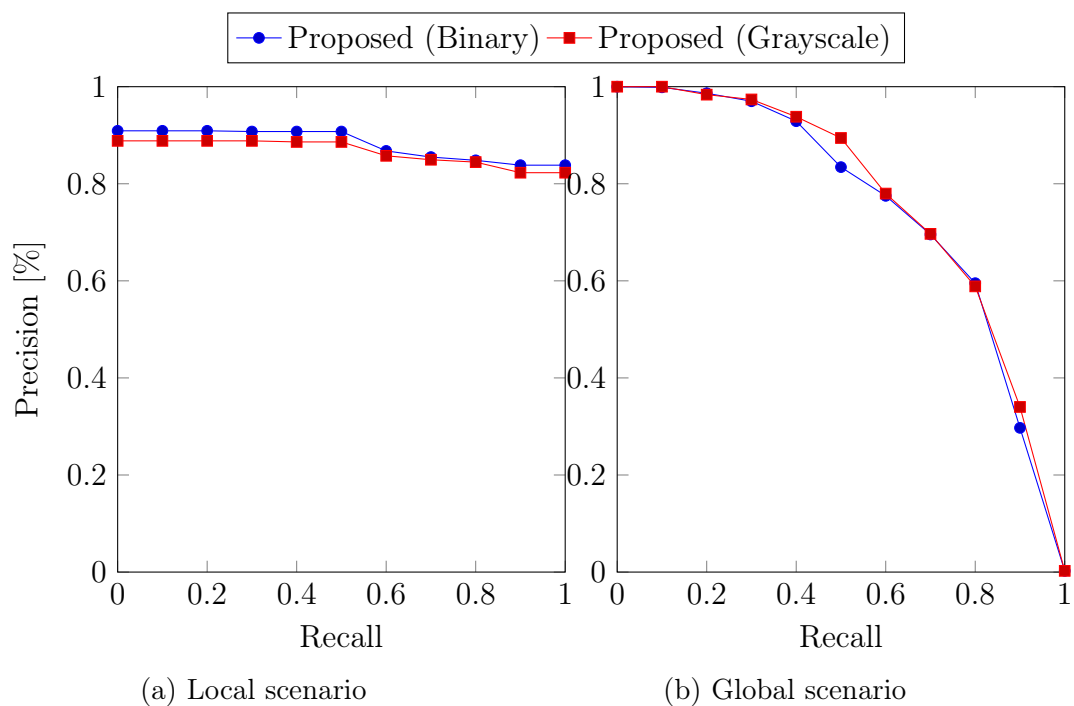


Figure 6.17: Recall-Precision performance of the Keyword Spotting System, on grayscale images, with HMM, on the GW data set.

Table 6.6: Time necessary to evaluate the results of one set of the GW data set with each feature set, the time for training the HMM is included. All results are in seconds.

System	DTW	HMM
Marti2001	19.54	81.12
Rodriguez2008	47.95	280.43
Terasawa2009	71.36	643.196
Proposed	26.11	391.545

6.8 Efficiency

As shown in the previous sections, the keyword spotting accuracy of the proposed system is very satisfying. Nevertheless, it is a complex system that needs training and from which it is not trivial to compute features. Therefore, it is also important to consider the runtime efficiency of the system, both for training the feature extractor and for computing the features and evaluating them. The hardware configuration is detailed in Section A.2.

Table 6.6 presents the time necessary to perform keyword spotting for one cross validation set of the GW data set. The training time of the feature extractor is not taken into account. The time necessary for the evaluation with DTW is mostly related to the feature extraction and only loosely to the feature dimensionality. The fastest system is *Marti2001* which is also the simplest and smallest feature set. For the same reasons, the proposed system is very fast with DTW because it has been optimized for feature extraction and its dimensionality is not critical for DTW. On the other hand, the used implementation of the other two reference feature sets is not optimized for feature extraction and therefore, there is a large overhead at this point. Nevertheless, even the slowest system is able to be completely evaluated on this data set in about 71 seconds. The timings with HMM are more interesting. Indeed, in that case, the time is dominated by the time spent inside the HMM training and the HMM evaluation using the Viterbi algorithm. For this, the dimensionality of the features is very important. This can be shown in the results where the feature set are ordered by their dimensionality. *Marti2001* features are very small and thus are the fastest of the feature set. *Terasawa2009* has the highest dimensionality, which makes it the slowest. In the end, the proposed system is positioning itself between *Rodriguez2008* and *Terasawa2009*. Overall, the significant advantage of DTW over HMM in terms of performance is also partially explained by the use of the Sakoe-Chiba band, significantly reducing the number of operations necessary.

Contrary to the other baselines, the proposed system needs to be trained. It is important that the system can be trained in reasonable time for the system to be used. Table 6.7 presents the time necessary to train the feature extractor on each data set. For comparison, the training time of the HMM is also included as well as the time necessary for evaluation with the HMM classifier. On the

Table 6.7: Time necessary to train the feature extractor. L1 and L2 are the times necessary to train the first layer, respectively the second layer. For comparison, the times for training and evaluation with HMM are included. All results are in seconds.

Data set	Complete	L1	L2	HMM Train	HMM Eval
GW cv1	1690.82	1241	449	44.744	391.01
GW cv2	1771.88	1300	471	44.983	458.36
GW cv3	1762.64	1295	465	46.435	328.21
GW cv4	1714.17	1256	458	45.12	376.46
PAR	8038	5219	2818	401.22	8932.28
IAM	53130	32019	22110	11715.8	48235.1

Table 6.8: Number of patches for each data set and number of patches processed by seconds by each layer during training.

Data set	Patches	L1	L2
GW cv1	370139	2983	8248
GW cv2	375955	2890	7982
GW cv3	381126	2938	8187
GW cv4	375310	2988	8194
PAR	1118197	2142	3968
IAM	4532408	1461	2049

GW data set, training the full model takes slightly less than half an hour. Even if this is a short training time, this is a very large overhead compared to less than one minute for HMM classifier and about six minutes for the evaluation. Nevertheless, it remains a relatively low training time. Training the CDBN takes about two and a quarter hours for the PAR data set. This is still very large compared to the seven minutes needed to train the HMM, but it is only as long as the evaluation itself with HMM. The training on the images from IAM is very time-consuming, taking more than 14 hours for the full model. Nevertheless, it is already two hours shorter than the HMM evaluation and training which is also very time consuming. Overall, training the model incurs a significant overhead when compared to standard features, especially on small data sets. On large data sets where the HMM evaluation takes significantly longer, the overhead is less significant, but it still takes about the same time to train the model than to evaluate it with the HMM classifier, including the HMM training time.

Table 6.8 presents detailed results for each layer with the number of patches processed per second for each layer. Although the number of images in each data set is not necessarily high, the number of patches generated from them is important. It should be clear that the network is able to process many patches per second and is very efficient. On GW data set, the first layer is able to process 3000 patches per second, while the second layer is able to process more than 8000 patches per second. The second layer has much smaller inputs than the first layer, therefore it

is much faster to process patches. On the PAR data set, the speed is significantly lower. Indeed, the first layer is able to process about 2100 patches per second while the second layer only processes 4000 images per second. The results are lower for two reasons. First, there are more feature maps in each layer than on GW. Moreover, due to the larger data sets, there are many more cache faults generated by the processor and the data locality is significantly worse. On the IAM data sets, the speed is even slower, with 1400 patches per second for the first layer and 2000 for the second layer. Since there are too many patches in this data set to fit in memory on the machine used of these experiments, the patches are regenerated from the original images for each mini-batch. This has a very large overhead. If the patches were able to fit in main memory, the speed would be almost equivalent to the speed on the PAR data set. Considering that these results are generated on a low-end Central Processing Unit (CPU) without any Graphical Processing Unit (GPU) acceleration, the training times can be considered rather acceptable

Overall, the proposed system is able to compete with the other baselines for evaluation, being even faster than the *Terasawa2009* baseline. The feature extraction process is already fast and could even be optimized further if necessary. However, the system needs to be trained and the training time of the complete system is significant. Nevertheless, considering the large number of patches with which the system is trained, the overall system can still be trained in very satisfying times. Indeed, the system is able to train up to 8000 patches per second, on a low-end CPU machine. This is only possible due to the performance optimizations that were developed in the used framework (See Section 4.4).

6.9 Summary

In this chapter, results about experiments in Handwritten Keyword Spotting in Historical documents were reported. Unsupervised Learning was used to automatically learn a feature extractor over small patches generated using a sliding window, over a word image. These features are then passed over to a classifier, either DTW or HMM, to take the final word spotting decision. The performance of the generated features is then compared to three reference feature sets, on three different data sets.

On every tested situation, the proposed feature extractor outperformed each reference feature set. When using DTW as a classifier the system was able to reduce the error by up to 18%. When HMM is used, the error is reduced by up to 45%. When using binary images transformed into grayscale using a Gaussian blur, the error is reduced by up to another 3.4% when using DTW. However, it has more mixed results with HMM, improving the performance in the global scenario but reducing it in the local scenario.

Although the proposed system outperformed every baseline and proved very efficient in learning features from the tested data sets, its optimization has proved rather challenging. Indeed, not only does the model have many design parameters,

there are several training hyper-parameters. Moreover, when Gaussian visible units are used with ReLU, the training of the model revealed not very stable, resulting in significant differences between different trained models.

Regarding CPU efficiency, the proposed system can generate features very quickly and is only slightly slower than the most basic of the reference feature sets. On the other hand, the feature extractor needs to be trained for each data set. While this process is fast on small data sets such as GW, several hours are necessary for PAR and IAM data sets. When DTW is used as classifier, this is a very large overall overhead while for HMM the relative overhead is less important since the classifier itself requires significant time to train.

While the results presented in this chapter are very satisfactory, several improvements of the system are still possible. To see if the same set of learned features could be successfully used with two different classifiers, the same parameters of the model were used for both the DTW and the HMM classifiers. Since DTW proved the least efficient of the two, the parameters were mostly tuned to work well with DTW. It should be possible to optimize further the parameters for HMM in order to obtain even superior results. Another possibility to improve the results further, especially for the smaller GW data set, would be to augment the data set with affine and elastic distortions to generate better features. Also, training the model to reconstruct clean patches from some noisy images, could be a way to obtain more robust features (Tang, Salakhutdinov, and G. E. Hinton, 2012) (Cho, 2013). While DTW and HMM are both good classifiers for word spotting, they may not be the most efficient models for the learned features. Indeed, recurrent models such as the LSTM were shown to be better than HMM in some cases (Frinken et al., 2012). Therefore, it would be interesting to test LSTM as classifier with the automatically learned features. While it was seen that transforming the binary images into grayscale images with a Gaussian filter could slightly improve the performance, it would be even more interesting to work on real grayscale images if a good word segmentation was available. Moreover, the stability of the system when using grayscale images could also been improved. Finally, since the system is now able to perform word spotting with good performance, it would be interesting to develop an application or a web service from it and make it available to a larger audience.

References for Chapter 6

- Adamek, Tomasz, Noel E. O'Connor, and Alan F. Smeaton (2007). "Word matching using single closed contours for indexing handwritten historical documents". In: *Int. Journal of Document Analysis and Recognition* 9, pp. 153–165 (cit. on p. 114).
- Almazán, Jon et al. (2014). "Word spotting and recognition with embedded attributes". In: *IEEE transactions on pattern analysis and machine intelligence* 36.12, pp. 2552–2566 (cit. on p. 115).

- Barrett, Christian, Richard Hughey, and Kevin Karplus (1997). “Scoring hidden Markov models”. In: *Computer applications in the biosciences: CABIOS* 13.2, pp. 191–199 (cit. on p. 124).
- Bengio, Yoshua (2009). “Learning deep architectures for AI”. In: *Foundations and trends® in Machine Learning* 2.1, pp. 1–127 (cit. on pp. 58, 112, 149, 150).
- Bengio, Yoshua, Aaron Courville, and Pascal Vincent (2013). “Representation learning: A review and new perspectives”. In: *IEEE transactions on pattern analysis and machine intelligence* 35.8, pp. 1798–1828 (cit. on p. 112).
- Bhardwaj, Anurag, Damien Jose, and Venu Govindaraju (2008). “Script Independent Word Spotting in Multilingual Documents.” In: *IJCNLP*, pp. 48–54 (cit. on p. 114).
- Cao, Huaigu and Venu Govindaraju (2007). “Template-free word spotting in low-quality manuscripts”. In: *Proceedings of the 6th International Conference on Advances in Pattern Recognition*, pp. 135–139 (cit. on p. 115).
- Chan, Jim, Celal Ziftci, and David Forsyth (2006). “Searching off-line Arabic documents”. In: *Proceedings of the IEEE Conf. on Computer Vision and Pattern Recognition*. Vol. 2. IEEE, pp. 1455–1462 (cit. on pp. 115, 123).
- Chen, Francine R., Lynn U. Wilcox, and Dun S. Bloomberg (1993). “Word spotting in scanned images using Hidden Markov Models”. In: *Proceedings of the IEEE Int. Conf. on Acoustics Speech and Signal Processing*. Vol. 5. IEEE, pp. 1–4 (cit. on p. 113).
- Cho, Kyunghyun (2013). “Boltzmann machines and denoising autoencoders for image denoising”. In: *arXiv preprint arXiv:1301.3468* (cit. on pp. 143, 162, 167).
- Choisy, Christophe (2007). “Dynamic handwritten keyword spotting based on the NSHP-HMM”. In: *Proceedings of the IEEE Int. Conf. on Document Analysis and Recognition*. Vol. 1. IEEE, pp. 242–246 (cit. on pp. 115, 123).
- Christian, Wolf, Jean-Michel Jolion, and Françoise Chassaing (2002). “Text Localization, Enhancement and Binarization in Multimedia Documents”. In: *Proceedings of the International Conference on Pattern Recognition*. Vol. 2, pp. 1037–1040 (cit. on p. 137).
- Coates, Adam, Honglak Lee, and Andrew Y. Ng (2010). “An analysis of single-layer networks in unsupervised feature learning”. In: *Ann Arbor* 1001.48109, p. 2 (cit. on pp. 112, 167).
- Dalal, Navneet and Bill Triggs (2005). “Histograms of oriented gradients for human detection”. In: *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR’05)*. Vol. 1. IEEE, pp. 886–893 (cit. on p. 133).
- Fischer, Andreas, Emanuel Indermühle, et al. (2010). “Ground truth creation for handwriting recognition in historical documents”. In: *Proceedings of the IAPR Int. Workshop on Document Analysis Systems*. ACM, pp. 3–10 (cit. on p. 120).
- Fischer, Andreas, Andreas Keller, et al. (2012). “Lexicon-free handwritten word spotting using character HMMs”. In: *Pattern Recognition Letters* 33, pp. 934–942 (cit. on pp. 113, 115, 116, 120, 124–126, 132, 134).
- Fischer, Andreas, Markus Wüthrich, et al. (2009). “Automatic transcription of handwritten medieval documents”. In: *Proceedings of the IEEE Int. Conf. on Virtual Systems and Multimedia*. IEEE, pp. 137–142 (cit. on p. 117).

- Forsyth, David et al. (2005). “Making latin manuscripts searchable using gHMMs”. In: *Proceedings of the Advances in Neural Information Processing Systems*. Vol. 17. MIT Press, p. 385 (cit. on p. 115).
- Frinken, Volkmar et al. (2012). “A novel word spotting method based on recurrent neural networks”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 34, pp. 211–224 (cit. on pp. 115, 143).
- Günter, Simon and Horst Bunke (2003). “Optimizing the Number of States, Training Iterations and Gaussians in an HMM-based Handwritten Word Recognizer”. In: *Proceedings of the Seventh International Conference on Document Analysis and Recognition*. IEEE, pp. 472–476 (cit. on p. 134).
- Hotelling, Harold (1933). “Analysis of a complex of statistical variables into principal components”. In: *J. Educ. Psych.* 24 (cit. on p. 112).
- Howe, Nicholas R. (2013). “Part-structured inkball models for one-shot handwritten word spotting”. In: *2013 12th International Conference on Document Analysis and Recognition*. IEEE, pp. 582–586 (cit. on p. 115).
- Iwana, Brian, Uchida Seiichi, and Volkmar Frinken (2016). “A Robust Dissimilarity-based Neural Network for Temporal Pattern Recognition”. In: *International Conference on Frontiers in Handwriting Recognition (ICFHR) 2016* (cit. on p. 122).
- Jaderberg, Max, Andrea Vedaldi, and Andrew Zisserman (2014). “Deep features for text spotting”. In: *European conference on computer vision*. Springer, pp. 512–528 (cit. on p. 116).
- Johansson, Stig, Geoffrey N Leech, and Helen Goodluck (1978). *Manual of Information to Accompany the Lancaster-Oslo/Bergen Corpus of British English, for Use with Digital Computer*. Department of English, University of Oslo (cit. on p. 117).
- Kovalchuk, Alon, Lior Wolf, and Nachum Dershowitz (2014). “A Simple and Fast Word Spotting Method”. In: *14th International Conference on Frontiers in Handwriting Recognition, ICFHR 2014, Crete, Greece, September 1-4, 2014*, pp. 3–8 (cit. on p. 114).
- Krishnan, Praveen, Kartik Dutta, and C. V. Jawahar (2016). “Deep Feature Embedding for Accurate Recognition and Retrieval of Handwritten Text”. In: *International Conference on Frontiers in Handwriting Recognition (ICFHR) 2016* (cit. on p. 116).
- Krizhevsky, Alex (2010). *Convolutional deep belief networks on cifar-10* (cit. on pp. 40, 44, 48, 131).
- Kuo, Shyh-shiaw and Oscar E. Agazzi (1994). “Keyword spotting in poorly printed documents using pseudo 2-D Hidden Markov Models”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 16, pp. 842–848 (cit. on p. 113).
- Lavrenko, Victor, Toni M. Rath, and Raghavan Manmatha (2004). “Holistic word recognition for handwritten historical documents”. In: *Proceedings of the Int. Workshop on Document Image Analysis for Libraries*. IEEE, pp. 278–287 (cit. on p. 117).
- Lee, Honglak, Chaitanya Ekanadham, and Andrew Y. Ng (2008). “Sparse deep belief net model for visual area V2”. In: *Advances in neural information processing systems*, pp. 873–880 (cit. on pp. 46, 69, 131).

- Leydier, Yann et al. (2009). “Towards an omnilingual word retrieval system for ancient manuscripts”. In: *Pattern Recognition* 42.9, pp. 2089–2105 (cit. on p. 114).
- Lloyd, Stuart (1982). “Least squares quantization in PCM”. In: *IEEE transactions on information theory* 28.2, pp. 129–137 (cit. on p. 112).
- Manmatha, Raghavan, Chengfeng Han, and Edward M. Riseman (1996). “Word spotting: A new approach to indexing handwriting”. In: *Proceedings of the IEEE Conf. on Computer Vision and Pattern Recognition*. IEEE, pp. 631–637 (cit. on pp. 114, 115).
- Marti, Urs V. and Horst Bunke (2001). “Using a statistical language model to improve the performance of an HMM-based cursive handwriting recognition system”. In: *Int. Journal of Pattern Recognition and Artificial Intelligence* 15, pp. 65–90 (cit. on pp. 118, 120).
- Myers, Cory, Lawrence R. Rabiner, and Andrew Rosenberg (1980). “An investigation of the use of Dynamic Time Warping for word spotting and connected speech recognition”. In: *Proceedings of the IEEE Int. Conf. on Acoustics Speech and Signal Processing*. Vol. 5. IEEE, pp. 173–177 (cit. on pp. 113, 122).
- Perronnin, Florent and Jose A. Rodriguez-Serrano (2009). “Fisher kernels for handwritten word-spotting”. In: *2009 10th International Conference on Document Analysis and Recognition*. IEEE, pp. 106–110 (cit. on p. 115).
- Plötz, Thomas and Gernot A. Fink (2009). “Markov models for offline handwriting recognition: a survey”. In: *International Journal on Document Analysis and Recognition (IJDAR)* 12.4, pp. 269–298 (cit. on p. 123).
- Rabiner, Lawrence R. (1989). “A tutorial on hidden Markov models and selected applications in speech recognition”. In: *Proceedings of the IEEE*, pp. 257–286 (cit. on pp. 123, 124).
- Rath, Toni M. and Raghavan Manmatha (2003). “Word image matching using Dynamic Time Warping”. In: *Proceedings of the IEEE Conf. on Computer Vision and Pattern Recognition*. Vol. 2. IEEE, pp. 521–527 (cit. on pp. 114, 122).
- Rath, Tony M. and Rudrapatna Manmatha (2007). “Word spotting for historical documents”. In: *Int. Journal of Document Analysis and Recognition (IJDAR)* 9, pp. 139–152 (cit. on pp. 114, 122).
- Retsinas, George et al. (2016). “Keyword Spotting in Handwritten Documents Using Projections of Oriented Gradients”. In: *2016 12th IAPR Workshop on Document Analysis Systems (DAS)*, pp. 411–416 (cit. on p. 115).
- Rodriguez, Jose A. and Florent Perronnin (2008). “Local gradient histogram features for word spotting in unconstrained handwritten documents”. In: *Proceedings of the Int. Conf. on Frontiers in Handwriting Recognition*, pp. 7–12 (cit. on pp. 114, 118, 122, 123).
- Rose, Richard C. and Douglas B. Paul (1990). “A Hidden Markov Model based keyword recognition system”. In: *Proceedings of the Int. Conf. on Acoustics Speech, and Signal Processing*. IEEE, pp. 129–132 (cit. on p. 113).
- Rothfeder, Jamie L., Shaolei Feng, and Toni M. Rath (2003). “Using corner feature correspondences to rank word images by similarity”. In: *Computer Vision and Pattern Recognition Workshop, 2003. CVPRW’03. Conference on*. Vol. 3. IEEE, pp. 30–30 (cit. on p. 114).

- Rusiñol, Marçal et al. (2015). “Efficient segmentation-free keyword spotting in historical document collections”. In: *Pattern Recognition* 48.2, pp. 545–555 (cit. on p. 115).
- Sakoe, Hiroaki and Seibi Chiba (1978). “Dynamic programming algorithm optimization for spoken word recognition”. In: *IEEE Transactions on Acoustics Speech and Signal Processing* 26, pp. 43–49 (cit. on p. 122).
- Scott, Guy L. and Christopher H. Longuet-Higgins (1991). “An algorithm for associating the features of two images”. In: *Proceedings of the Royal Society of London B: Biological Sciences* 244.1309, pp. 21–26 (cit. on p. 114).
- Sharma, Arjun and Pramod Sankar (2015). “Adapting off-the-shelf CNNs for word spotting and recognition”. In: *2015 13th International Conference on Document Analysis and Recognition (ICDAR)*, pp. 986–990 (cit. on p. 116).
- Stauffer, Michael, Andreas Fischer, and Kaspar Riesen (2016). “A Novel Graph Database for Handwritten Word Images”. In: *Structural, Syntactic, and Statistical Pattern Recognition - Joint IAPR International Workshop, S+SSPR 2016, Mérida, Mexico, November 29 - December 2, 2016, Proceedings*, pp. 553–563 (cit. on p. 137).
- Sudholt, Sebastian and Gernot A. Fink (2016). “PHOCNet: A Deep Convolutional Neural Network for Word Spotting in Handwritten Documents”. In: *International Conference on Frontiers in Handwriting Recognition (ICFHR) 2016* (cit. on p. 116).
- Tang, Yichuan, Ruslan R. Salakhutdinov, and Geoffrey E. Hinton (2012). “Robust boltzmann machines for recognition and denoising”. In: *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*. IEEE, pp. 2264–2271 (cit. on pp. 143, 162).
- Terasawa, Kengo and Yuzuru Tanaka (2009). “Slit style HOG feature for document image word spotting”. In: *Proceedings of the IEEE Int. Conf. on Document Analysis and Recognition*. IEEE, pp. 116–120 (cit. on pp. 114, 118, 122, 123).
- Thomas, Simon, Clément Chatelain, Laurent Heutte, and Thierry Paquet (2010). “An information extraction model for unconstrained handwritten documents”. In: *Pattern Recognition (ICPR), 2010 20th International Conference on*. IEEE, pp. 3412–3415 (cit. on p. 115).
- Thomas, Simon, Clément Chatelain, Laurent Heutte, Thierry Paquet, and Yousri Kessentini (2015). “A deep HMM model for multiple keywords spotting in handwritten documents”. In: *Pattern Analysis and Applications* 18.4, pp. 1003–1015 (cit. on p. 115).
- Viterbi, Andrew (1967). “Error bounds for convolutional codes and an asymptotically optimum decoding algorithm”. In: *IEEE transactions on Information Theory* 13.2, pp. 260–269 (cit. on p. 124).
- Wicht, Baptiste, Andreas Fischer, and Jean Hennebert (2016a). “Deep Learning Features for Handwritten Keyword Spotting”. In: *International Conference on Pattern Recognition (ICPR)* (cit. on p. 131).
- (2016b). “Keyword Spotting with Convolutional Deep Belief Networks and Dynamic Time Warping”. In: *International Conference on Artificial Neural Networks*. Springer International Publishing, pp. 113–120 (cit. on pp. 122, 131).

- El-Yacoubi, Mounim A., Michel Gilloux, and Jean-Michel Bertille (2002). “A statistical approach for phrase location and recognition within a text line: an application to street name recognition”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 24.2, pp. 172–188 (cit. on p. 115).
- Zhang, Bin, Sargur N Srihari, and Chen Huang (2003). “Word image retrieval using binary features”. In: *Proceedings of the Electronic Imaging 2004*. Int. Society for Optics and Photonics, pp. 45–53 (cit. on p. 114).

Chapter 7

Auto-encoders

*We know a lot of things, but what
we don't know is a lot more*

Edward Witten

Contents

7.1	Introduction	149
7.2	Experimental Evaluation	151
7.3	Dense Auto-Encoders	152
7.4	Convolutional Auto-Encoders	155
7.5	Hybrid Auto-Encoders	160
7.6	Denoising Auto-Encoders	162
7.7	Results	165
7.8	Summary	166

7.1 Introduction

This thesis is based on the predicate that the Restricted Boltzmann Machine (RBM) and Convolutional Restricted Boltzmann Machine (CRBM) models are well-performing models able to efficiently extract features from images when trained in an unsupervised manner. However, these models are not the only ones that are capable of automatically learning a new representation from data in an unsupervised way. There are several alternatives. This chapter describes some of these alternative models, namely the auto-encoders, and goes on to compare some of them on a feature extraction task.

In essence, a RBM is an auto-encoder model, also called an auto-associator or Diabolo network (Bengio, 2009; Rumelhart, G. E. Hinton, and Williams, 1988;

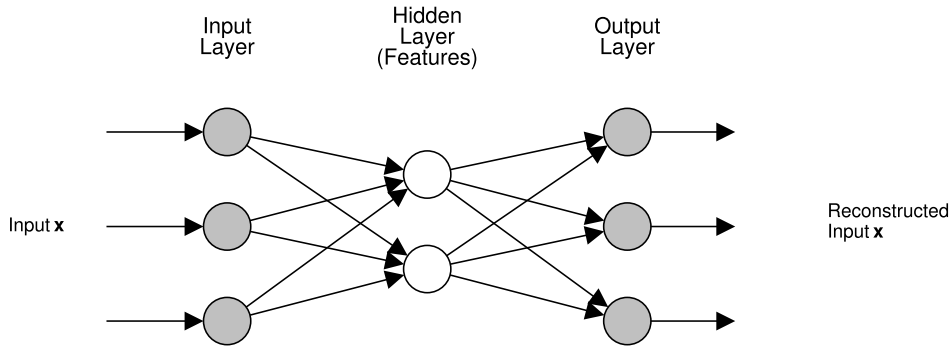


Figure 7.1: Example of an auto-encoder neural network. The network is trained as a standard neural network. Only the hidden layer is used for feature extraction.

(G. E. Hinton and Zemel, 1994). These models are trained to encode the input in some different representation and reconstruct the input from this learned representation. Therefore, the expected output is the input itself, no labels are necessary for training. There are generally two parts to these models, the encoder network and the decoder network. Once the full network has been trained, the encoder network can be used to extract features from data. The learned representation needs not necessarily to be smaller than the input representation. While this would intuitively allow the auto-encoder learn the identity function as the internal representation, it was shown that, in practice, an auto-encoder trained with Stochastic Gradient Descent (SGD) could yield some useful representations, bigger than the input (Bengio, LeCun, and al., 2007).

The simplest form of auto-encoder is a Multi-Layer Perceptron (MLP) with one hidden layer, as shown in Figure 7.1. The network can be trained with standard back propagation techniques using the input as the expected target. From this simple form of auto-encoder, several variations can be derived. The principle can also be applied to convolutional auto-encoders to reconstruct images by learning convolutional filters instead of a flat representation (Masci et al., 2011). Convolutional auto-encoders are generally learning better features on images and sound data sets. Fully connected auto-encoders are very simple to implement since the decoder layer is simply a fully-connected layer with transposed dimensions. For convolutional auto-encoder, it is slightly more complicated. The transposed operation being the deconvolution can be implemented with some form of full convolution (Noh, Hong, and B. Han, 2015). Single-layer models can also be composed to create higher level representations. This can be achieved using two different techniques (Bengio, 2009). Several simple models can be trained in a chain, each reconstructing the representation of the previous one. It is also possible to create a Deep Neural Network by adding more hidden layers before the representation layer and the corresponding inverse layers after it.

A very successful technique for training auto-encoders is to corrupt the input and training the network to reconstruct the clean input. These models are called De-

noising Auto-Encoders (DAEs) (Vincent, Larochelle, Bengio, et al., 2008; Vincent, Larochelle, Lajoie, et al., 2010). The goal is to make the features more robust by not presenting them the real images. These models proved very efficient and are able to learn very powerful representations. Since this does not change the model itself but only the data on which it is trained, it is very straightforward to apply this principle to any auto-encoder.

An alternative to denoising auto-encoders is to force the model to learn a sparse representation of the input data. These models are called Sparse Auto-Encoders (Olshausen and Field, 1997; Y.-l. Boureau, LeCun, and Ranzato, 2008; Ranzato and LeCun, 2007; Doi, Balcan, and Lewicki, 2005). Generally, these models have a larger representation than other models, but the units are activated only rarely. These models are generally trained by adding a sparsity term to the loss function.

More recently, even more advanced models for auto-encoders have been proposed. The Variational Auto-Encoder (VAE) is a special form of auto-encoder that learns the distribution over the data and a set of latent variables (D. P. Kingma and Welling, 2013; Rezende, Mohamed, and Wierstra, 2014). These models are very good at generating samples from scratch or from corrupted inputs. The model can also be adapted for semi-supervised learning (D. P. Kingma, Mohamed, et al., 2014). Another successful variant of the VAE is the Conditional Variational Auto-Encoder (CVAE) that takes advantage of side information such as labels to improve the output representation (Sohn, H. Lee, and Yan, 2015). Another powerful model is the Generative Adversarial Network (GAN) (I. J. Goodfellow et al., 2014). In this model, two networks are trained by competing against each other. The first model is trying to generate fake images that are able to fool the second model which must decide if the input is real or not. This model is able to generate very realistic images from simple labels. From this model, Adversarial Auto-Encoder (AAE) can be derived (Makhzani et al., 2015; Radford, Metz, and Chintala, 2015) and can learn very robust representations. Several attempts were also made to combine the strengths of the VAE and GAN models (Mescheder, Nowozin, and Geiger, 2017; Larsen, Sønderby, and Winther, 2015). Overall, these two families of models and their combinations have achieved excellent performance mostly in generating or distorting images and extracting robust features.

This chapter compares some of the simple approaches to feature extraction against the RBM approach and summarizes the state of the RBM approach. More specifically, dense and convolutional auto-encoders, deep and stacked auto-encoders, hybrid auto-encoders and denoising auto-encoders are experimented with.

7.2 Experimental Evaluation

In the following sections, several approaches to automatic feature extraction learning will be compared to the approach based on RBM models. It is not a trivial task to compare features together and evaluate them. We re-used the Keyword Spotting experiment and the evaluation framework to compare the different fea-

tures learned with auto-encoders. For this, each model learns features of the same dimensionality and these features are passed to a simple Dynamic Time Warping (DTW) classifier, for keyword spotting. Using a very simple classifier has the advantage that the features are compared as directly as possible. The DTW is used with a Sakoe-Chiba band of 0.05 for each model. The George Washington data set was selected for this task in to be able to test many feature sets in a reasonable time. Features are learned directly from the patches of the George Washington data set, using the first cross validation set. The evaluation is done using Average Precision (AP) and Mean Average Precision (MAP), in two different scenarios. The full details of the evaluation for keyword spotting are explained in detail in Section 6.6.1.

The experiments were developed using the DLL framework. For reference and for reproducibility of the results, the source code used to perform these experiments is available online¹. This includes the C++ code for each of the model for each of the experiments as well as the necessary code for the data set loading and the evaluation of the different features.

7.3 Dense Auto-Encoders

The first experiment that is done is to compare dense auto-encoders and RBM models. For this, models with the same feature dimensionality are learned on the same data and are compared.

A dense auto-encoder model is a very simple neural network. It has two fully-connected layers, the second layer being the transpose of the first one, as shown in Figure 7.1. The tested networks are all using sigmoid activation functions. The first layer has N hidden units and the second layer has the same number of hidden units as the first layer has input units. It is trained exactly as a standard Artificial Neural Network (ANN). This model is trained with Mini-Batch gradient descent, with momentum and L2 weight decay. Each model is trained for 10 epochs. The input data set is shuffled before each epoch. The RBM has only one layer, with N hidden units. It is trained with Contrastive Divergence (CD) for 10 epochs. It is also trained with momentum and L2 weight decay. Each epoch is trained with a different random permutation of the inputs. For both models, it has been necessary to reduce the learning rate as N increases.

The results for this experiment are presented in Figure 7.2, for both scenarios. There are some significant differences between the two models. The most important one is when generating only 10 features from the 800 input features. In that case, the dense auto-encoder excels while the RBM seems to fail learning a good representation. One possible explanation for this could be that the RBM tries to learn the input distribution, not only the reconstruction. It is possible that 10 features are not enough for this task. Overall, the dense auto-encoder has a better

¹https://github.com/wichtounet/word_spotting

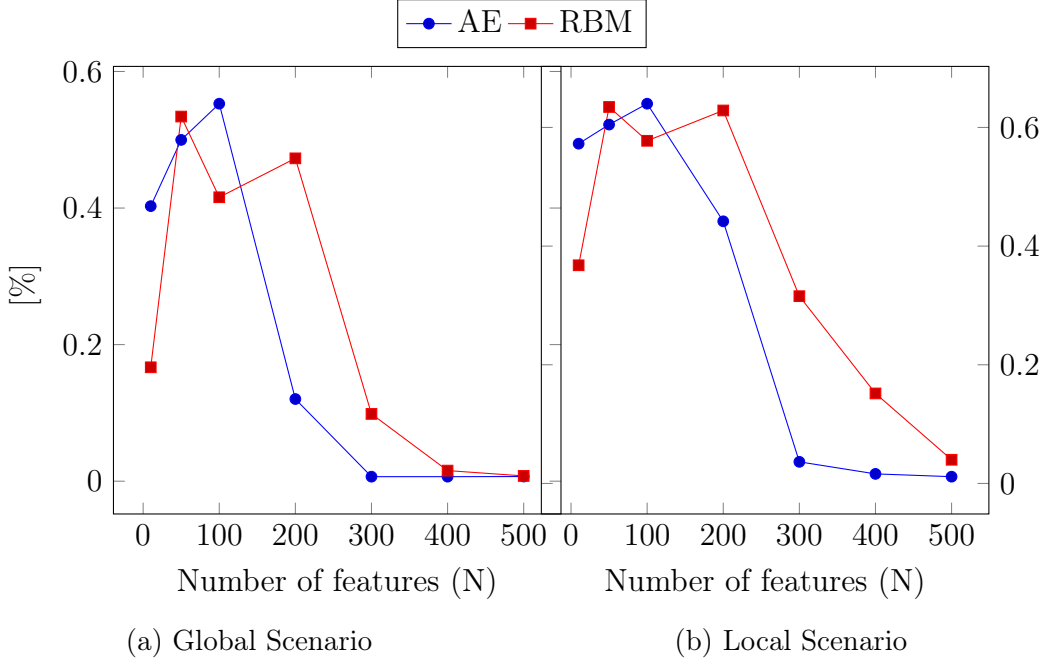


Figure 7.2: Comparison of Dense Auto-Encoders and RBM on a Keyword Spotting task, using the George Washington data set.

performance for a number of features below 200. For larger number of features, the RBM is always better. Overall, if the average performance over the number of features is considered, the RBM is 7.2% better in the global scenario and 16.9% better in the local one.

In practice, it is generally more powerful to use neural networks with several layers in order to extract more abstract features at each layer. The same observation stands true for auto-encoders. There are two different models for auto-encoders with multiple layers. The first idea is simply to add more encoding and decoding layers into the network, creating a deep auto-encoder. It is also possible to follow the RBM model and train auto-encoders independently one after another in a layerwise fashion, being a stacked auto-encoder. Both models are illustrated in Figure 7.3.

To compare a model with several RBM layers, also called a Deep Belief Network (DBN) and multi-layer auto-encoders, the same experiment was run again with a first intermediate layer of 200 units. The three models are trained in a similar manner as before. Another possibility, not explored here, is to combine both approaches by initializing the weights of a deep model using its stacked version (or RBMs) and then training again the deep model (Vincent, Larochelle, Lajoie, et al., 2010).

The results can be seen in Figure 7.4. These results are very interesting on several points. First, it can be observed that the stacked auto-encoder is producing better features than the deep auto-encoder, by about 30% in the global scenario and by about 21% in the local one. There are several possible explanations for this

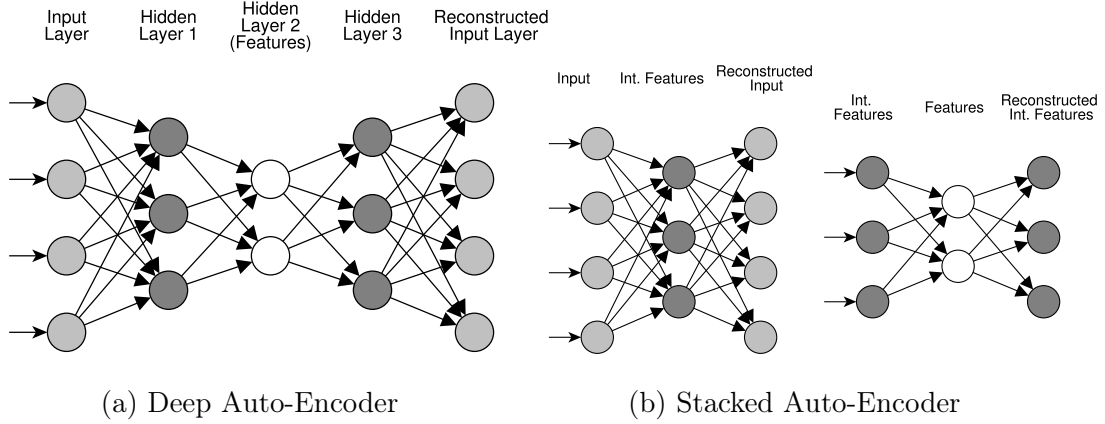


Figure 7.3: Comparison of Deep Auto-Encoder and Stacked Auto-Encoder

observation. First, it is generally easier to train several small models than a large model. Moreover, while it would make sense that combining the layers together would improve the results, having them separated can prevent too much adaptation of the layers to the input data, making the learned features more generic. Another advantage is to help the system to create multiple layers of abstraction, by creating good features at each level. This is not necessarily the case when deep neural networks are trained. When comparing the DBN and the stacked auto-encoder, the same effect with 10 features can be observed as in the previous example. The DBN shows poor performance with few features. By increasing the number of features, the DBN is able to extract features, but is worse in the global scenario (by about 11%) and better in the local scenario by about 9%. On the other side, the stacked auto-encoder model fails early on for large numbers of features. When the single-layer models are compared with the two-layer models, the deep model does not improve over the single-layer and even reduces the performance in the local scenario. On the other hand, the stacked model significantly improves the performance of the auto-encoder. For the RBM, the improvements are significant as well but smaller than for the auto-encoder. It must also be taken into account that the input samples are not very complex and as such may not profit much from deep features.

Overall, it seems like both models are performing well on this task and both have different strengths, with an advantage for the RBM for single-layer model and for the stacked auto-encoder with two layers. It is also observed that stacking several auto-encoders one after another can produce better results than a multi-layer auto-encoder. Being trained in a layer-by-layer manner makes the RBM faster to train, by about 20% in this case. It can also be seen that RBMs are performing better than auto-encoders when the number of features is large. On simple single-layer models, the training is similarly simple for both models. When several layers are used, RBM training can be cumbersome due to the high number of hyper parameters involved. Indeed, the number of parameters is linearly dependent on the number of RBMs used. While it is also possible to assign different learning

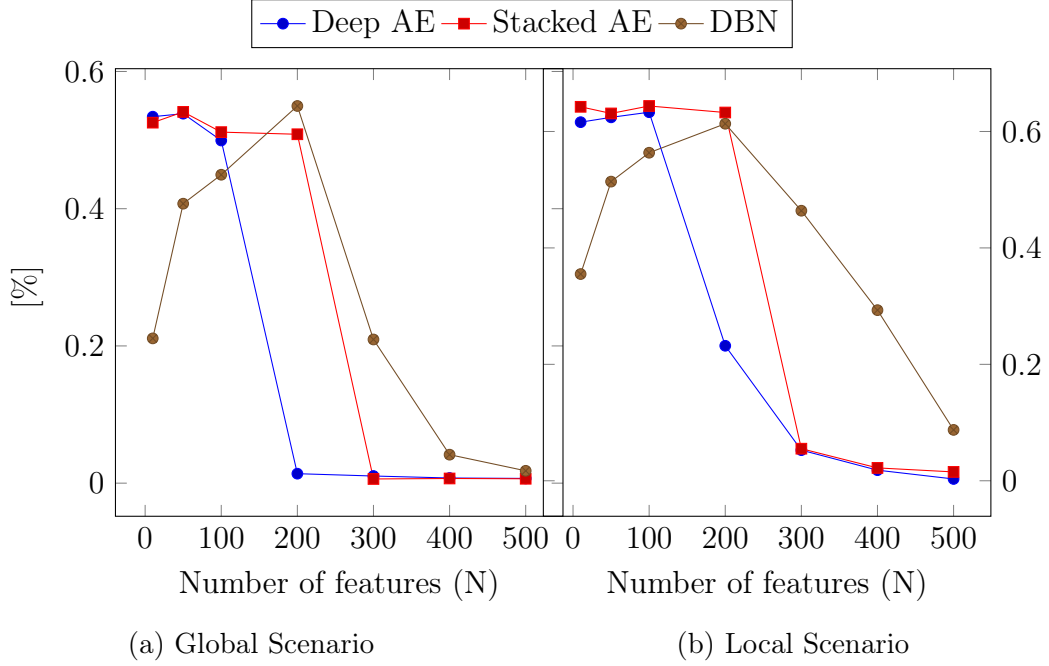


Figure 7.4: Comparison of Deep and Stacked Dense Auto-Encoders and DBN on a Keyword Spotting task, using the George Washington data set.

rates to different layers in an ANN, this is rarely done and as such makes it easier to train multi-layer neural networks than stacked RBMs. When the auto-encoders are stacked, the model becomes as complicated to train as the DBN model.

7.4 Convolutional Auto-Encoders

The second experiment that is performed is to compare models based on the Convolutional Auto-Encoder (CAE) and CRBM models. Again, models with the same architecture will be trained for the two different techniques on the same data and then compared against each other.

The convolutional auto-encoder has two layers like the dense auto-encoders, but the two layers are different. The first layer is a standard convolutional layer while the second layer is a so-called deconvolutional layer. The first layer performs a valid convolution and the second performs a full convolution, effectively getting back to the original dimensionality. This can be observed in Figure 7.5. This model is trained with standard Mini-Batch gradient descent, with the same parameters and refinements as the dense auto-encoder (see Section 7.3). The CRBM is simpler, having only one layer. It is also trained using the same techniques seen in the previous section. For both models, the size of the kernel has been fixed to 17x17. While this is a rather large kernel size, it was necessary in order to reduce the dimensionality enough for the DTW to give adequate results. Different numbers of filters (K) are tested for each model. The learning rates had to be adapted for

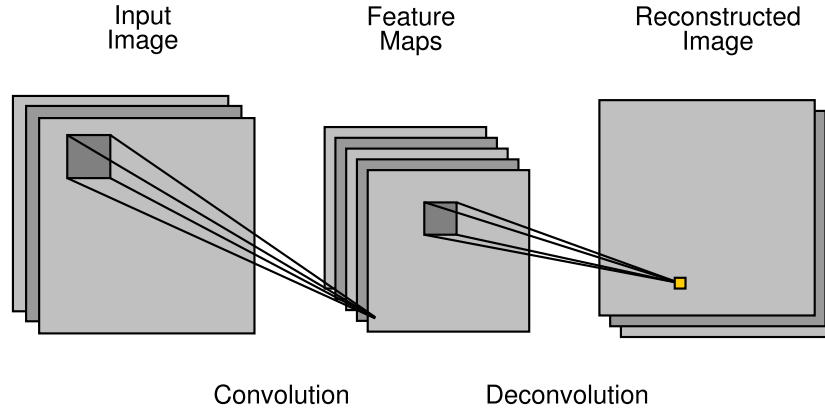


Figure 7.5: Example of a convolutional auto-encoder neural network. The network is trained as a standard neural network. Only the first layer is used for feature extraction.

the different numbers of filters. Moreover, the learning rate for the CRBM had to be set one order of magnitude lower than for the auto-encoder.

Figure 7.6 presents the results for this experiment, for both scenarios. The results are only good for small numbers of kernels. This confirms the fact that DTW is not able to handle too many features. There are some very significant differences between the models. In both scenarios, the CRBM model is superior to the auto-encoder model. In the global scenario, it is more than 90% better while it is only 12.82% better in the local scenario, when considering the average performance over different number of features. This shows a better generalization of the features for the CRBM in that case.

Since DTW is expecting features with small dimensionality, another tool to improve the results is to use pooling to reduce the size of the features and to improve their generalization. For convolutional auto-encoder, this means adding a max pooling layer, followed by an upsample layer. This means already four layers in the network. For the CRBM, there are two choices. The first is simply to add a max pooling layer after the CRBM layer. In this case, the training has no notion of pooling at all, the same features will be generated as previously, but they will be pooled. It is also possible to integrate pooling in the training using Probabilistic Max Pooling (PMP). Both models will be compared here against the auto-encoder. Again, each system will have the same architecture and different numbers of filters will be tested. The size of each kernel is kept to 17x17 and the features are pooled using a 2x2 kernel, with a stride of two, effectively dividing the number of features by four.

Figure 7.7 shows the results obtained when pooling the features. To see if adding pooling inside the network helps learning in the first layer, the features of both the first layer (CAE-L1) and the second layer (CAE-L2) of the auto-encoder are

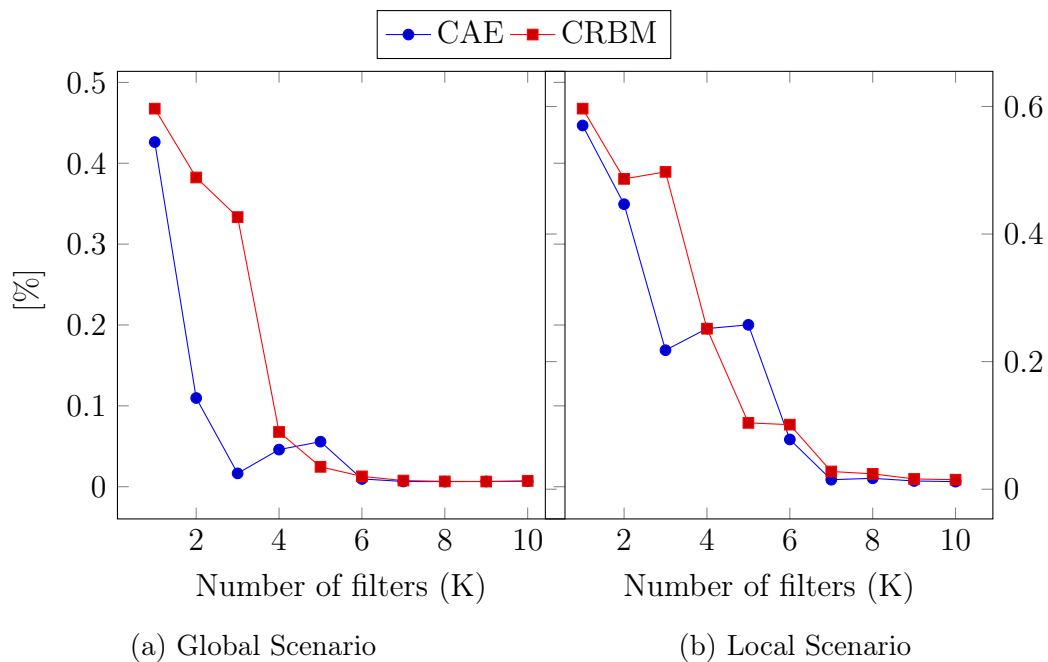


Figure 7.6: Comparison of Convolutional Auto-Encoders and CRBMs on a Keyword Spotting task, using the George Washington data set.

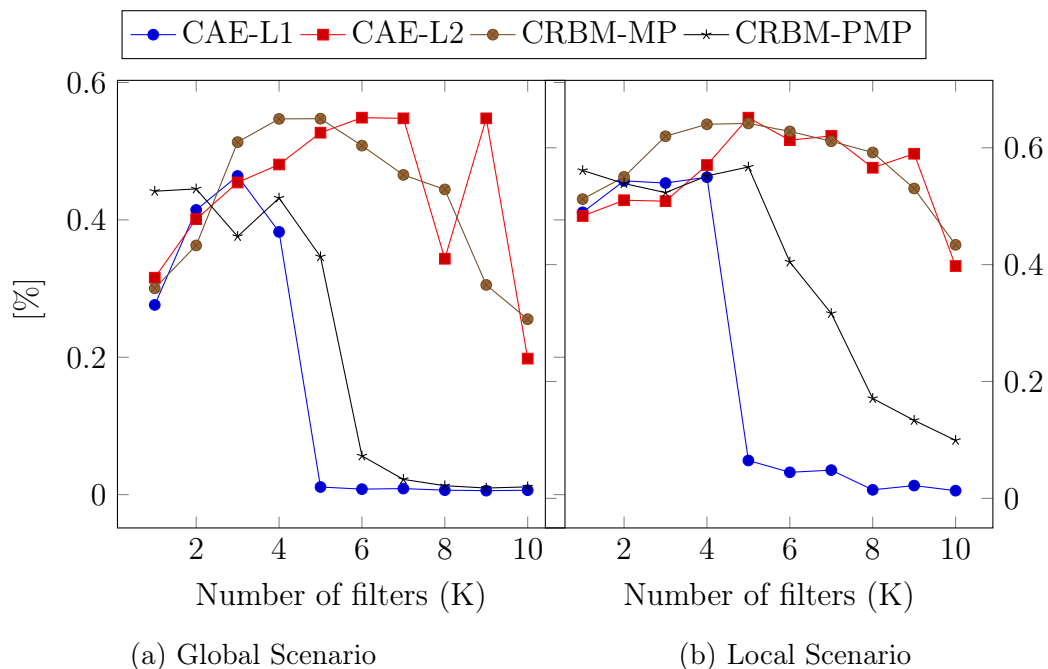


Figure 7.7: Comparison of Convolutional Auto-Encoders and CRBMs, with pooling, on a Keyword Spotting task, using the George Washington data set.

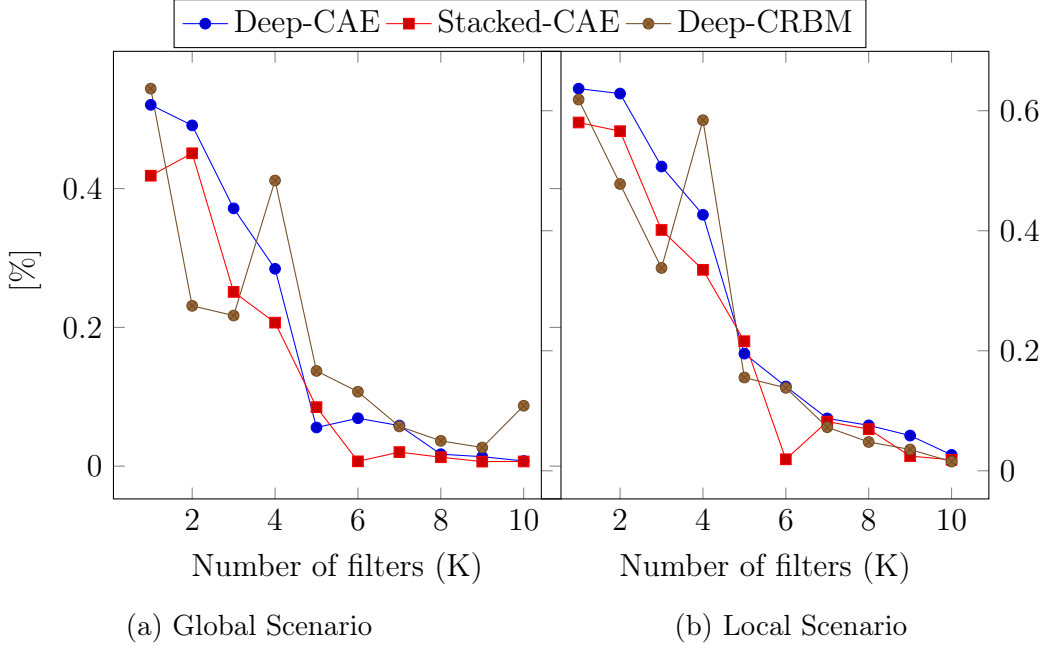


Figure 7.8: Comparison of Deep Convolutional Auto-Encoders and Deep CRBM on a Keyword Spotting task, using the George Washington data set.

evaluated. Indeed, it can be observed by looking at CAE-L1, that the results are significantly better than the results obtained by CAE in the previous experiment. This means that not only does adding a pooling layer improve the quality of the features, it does also strongly improve the learning of the previous layers. However, for DTW, the features after pooling (CAE-L2) are stronger than the unpooled features (CAE-L1). When comparing CRBM-MP and CRBM-PMP, it can be observed that the latter is less efficient. Although it has very strong features with small number of filters, it fails very quickly as the number of filters increases. In practice, during this thesis, it has been observed several times that the use of PMP decreased the stability and was only really interesting when sparsity regularization was applied to binary hidden features. The features learned by the CRBM-MP system and the CAE-L2 features are almost equivalent. CAE-L2 is about 6% better in the global scenario while CRBM-MP is about 4.5% better in the local scenario. The features learned by CRBM-MP are more stable than the one learned by CAE-L2, but the performance peaks are higher for the latter.

As for dense auto-encoders, it is also possible to have multi-layer convolutional auto-encoders. Again, there are two possible models, deep and stacked (See Section 7.3 for details). The CRBM model can also be stacked in the Convolutional Deep Belief Network (CDBN) model. The first experiment with two layers is using 9x9 kernels in both layers and no pooling. The first convolution layer is set to have 6 filters. The number of filters of the second layer will change from 1 to 10 and the same training parameters as before are used.

Figure 7.8 presents the performance of the three deep models. Overall, the results obtained with these models are not very good. They are almost comparable to

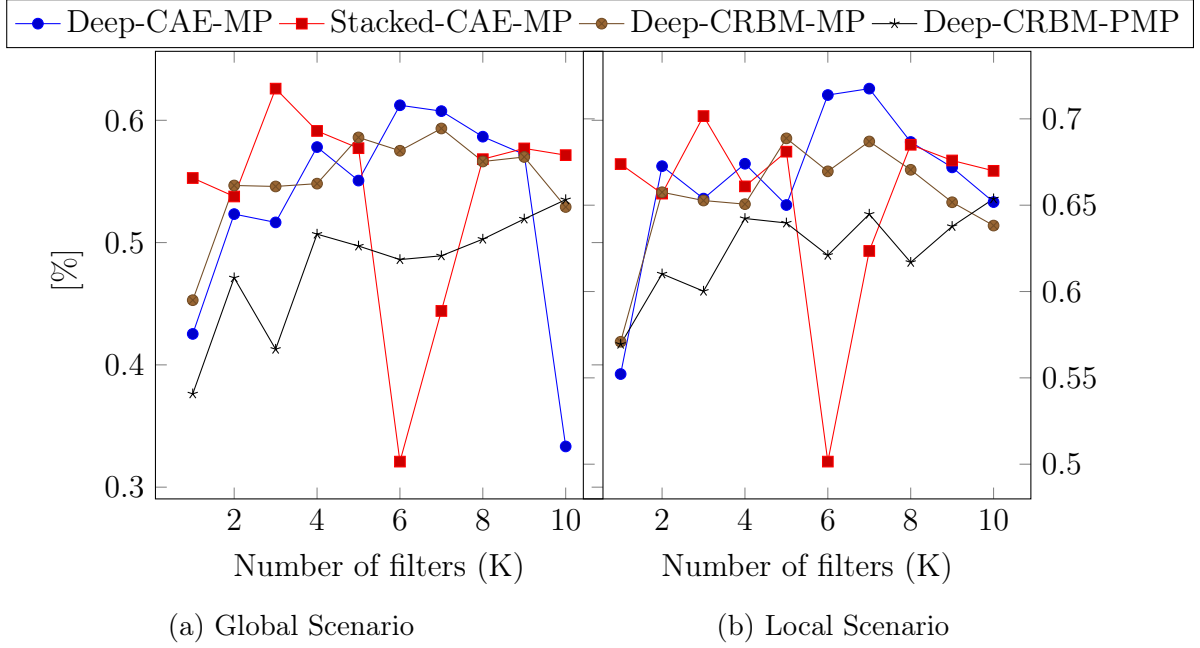


Figure 7.9: Comparison of Deep Convolutional Auto-Encoders and Deep CRBM on a Keyword Spotting task, with pooling layers, using the George Washington data set.

the results with a single layer and no pooling. Contrary to the previous deep results, the stacked auto-encoder performs significantly worse in both scenarios than the deep auto-encoder. In this experiment, the deep CRBM model does not perform very well, being 6.3% worse in the global scenario and 15% worse in the local scenario. It is only slightly better than the stacked auto-encoder. Another important difference between the models is the time necessary to train. Indeed, it is significantly faster to train the CDBN model than the deep auto-encoder. It takes almost twice the time to train it. The stacked auto-encoder is also slower than the CDBN but by a small margin of around 15%.

Since max pooling was shown to improve the results of single-layer models, the next experiment includes pooling layers in the three deep models. Again, both the CRBM model with max pooling and the CRBM with PMP are compared. The first layer has six 9x9 filters and the second layer has a variable number of 3x3 filters. Both convolutional layers are followed by a max pooling layer with a ratio of two in each dimension. With this configuration, the deep convolutional auto-encoder is composed of eight layers.

Figure 7.9 presents the performance of these four models. These deep convolutional models achieve the best performance observed in this chapter. They are reaching excellent spotting performance. Contrary to the deep dense models, the stacked model does not perform significantly better than the deep model. Both models are performing almost equivalently. Indeed, it is worse by 1.7% in the local scenario and slightly better in the global scenario (by 1.1%). However, the performance of the deep model is more stable over different configurations. Secondly, the CRBM

model with standard max pooling performs significantly better than the CRBM with PMP, by about 15% in the global scenario and 4.7% in the local one. Both the deep auto-encoder and the CDBN with max pooling are performing very well on this experiment. The deep auto-encoder is performing slightly better in the local scenario (by about 1.6%), while the CDBN is about 2% better in the global scenario. However, we note that the highest peaks of performance are achieved by the deep CAE model.

Overall, both the CAE and CRBM models are achieving very good performance for extracting features from images. In this experiment, pooling was shown to be very important to achieve maximum performance. This helps learning better features for the auto-encoders and also helps DTW by significantly reducing the number of features to compare. When multiple-layer models are used, the performance can be significantly improved. It was seen that the stacked and deep models for CAE performs about the same. It was also seen that the CRBM model with PMP performs worse than the CRBM model with simple max pooling layers. The CRBM models are generally more stable than the CAE models over different configurations. Finally, the best results were observed for the deep CAE model.

7.5 Hybrid Auto-Encoders

In the two previous sections, dense and convolutional auto-encoders have been experimented with. In standard neural networks, convolutional layers are often followed by a fully-connected layer to perform classification. It is also possible to build an auto-encoder using both kinds of layers, using the same architecture. We call this an hybrid auto-encoder in this section.

The first hybrid model uses a convolutional layer with six 9x9 kernels, followed by a fully-connected layer with variable number of hidden units. Both the deep and stacked versions of the auto-encoders are tested.

The results are presented in Figure 7.10. In the global scenario, the stacked model is stronger than the deep model, while it seems slightly worse in the local scenario. Moreover, the deep model seems more unstable with different configurations and was more delicate to train. As for the hybrid DBN, it is significantly worse in the local scenario, by about 3.8% and slightly better in the global scenario by about 1.2%. Overall, the results are rather good, better than using only the convolutional layer and better than the dense models.

The second hybrid experiment still uses a first convolutional layer with six 9x9 kernels, but it is followed by a max pooling layer with a pooling ratio of two in both dimensions. Again, it is followed by a fully-connected layer with variable numbers of hidden units. For the CRBM model, both a standard max pooling layer and PMP are tested.

The results are presented in Figure 7.11. Overall, the results achieved are very good. The deep CRBM model with max pooling layer is the worst model in this

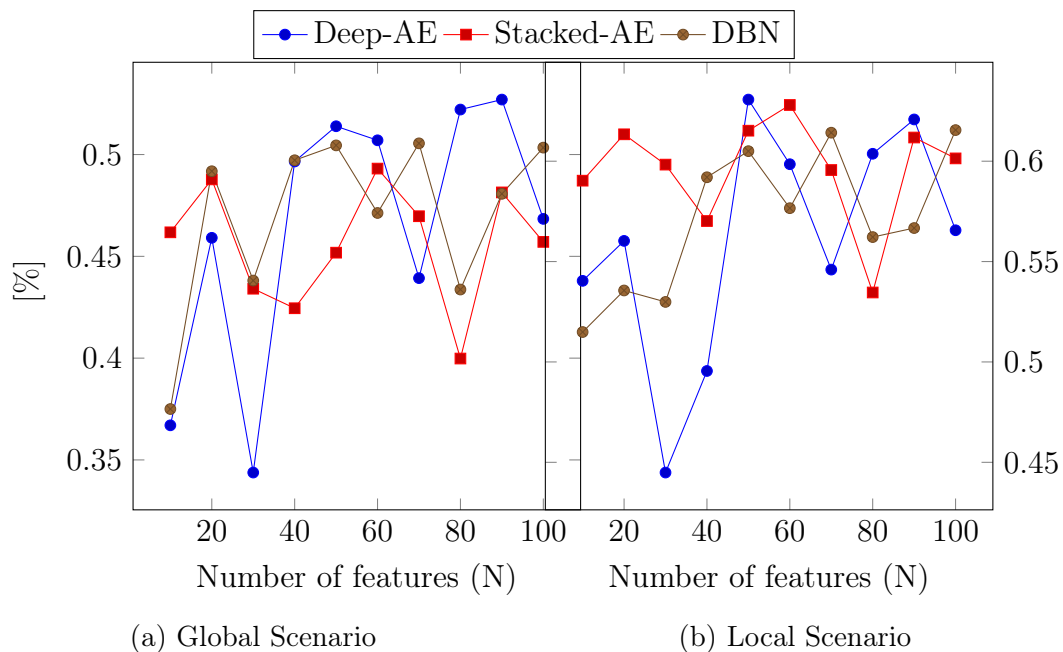


Figure 7.10: Comparison of Deep and Stacked Hybrid Auto-Encoders and Hybrid DBN on a Keyword Spotting task, using the George Washington data set.

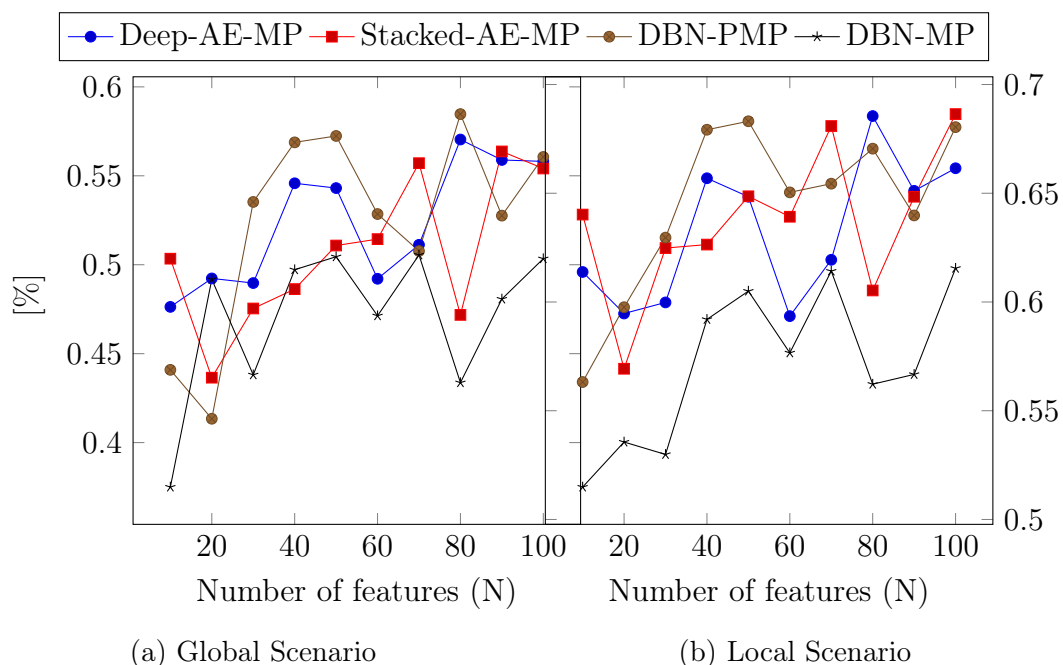


Figure 7.11: Comparison of Deep and Stacked Hybrid Auto-Encoders and Hybrid DBN, with pooling, on a Keyword Spotting task, using the George Washington data set.

experiment, being 11% worse in the global scenario and 12% in the local scenario than the CRBM with PMP. The stacked model is significantly worse in the global scenario but slightly better in the local one. Overall, the best model is the CRBM with PMP which is very slightly better in the global scenario and about 1.2% better in the local one than the second best model. The two hybrid models based on RBM have proved more difficult to tune than the standard models, especially with respect to the learning rate. They produce results with large differences between different training parameters.

Overall, both types of models are doing a good job on these hybrid experiments. The hybrid model with max pooling is able to achieve excellent performance. The CRBM with PMP is proving very efficient in this particular case. While the RBM family has generally better overall performance, it does not have the same peak performance as the simpler auto-encoder. Moreover, it is also more difficult to train the RBM models due to more parameters to configure. Nevertheless, their training remains significantly faster than standard auto-encoders.

7.6 Denoising Auto-Encoders

A simple way to improve the robustness of the features learned by an auto-encoder is to make them reconstruct a clean image from noisy images. The goal is to improve the robustness of the features and prevent weak features that are too much coupled to the input images. This technique can easily be applied to the training of an auto-encoder since it does not change the training significantly. The reconstruction is done from a noisy image instead of doing it from the clean image. It is important that the loss is computed using the clean image and the reconstructed image. This process is not the same as learning Image Denoising. Indeed, in this configuration, the model will be trained with potentially new images at each epoch. Although RBM and derived models are able to learn Image Denoising problems (Tang, Salakhutdinov, and G. E. Hinton, 2012; Cho, 2013), they are not always well suited to be trained as denoising auto-encoder. Indeed, their training is already stochastic, via the Bernoulli sampling and already accounting for noise. Therefore, adding more noise to the training may not be as efficient as it is on standard auto-encoder and may be detrimental. To compare how both models are impacted by applying this technique, several models are trained with varying amount of noise in the input. The noise model that is used is simply to have a probability to set a feature to zero in the input, this probability being the noise level.

The first model that is being trained is the single-layer dense model from Section 7.3. The model with 50 hidden units was selected since both models were very close on this configuration. The model is trained again using the same parameters with some level of noise.

Figure 7.12 shows the results of this first denoising experiment. When no noise is applied, the results are almost similar, as was already seen previously. However,

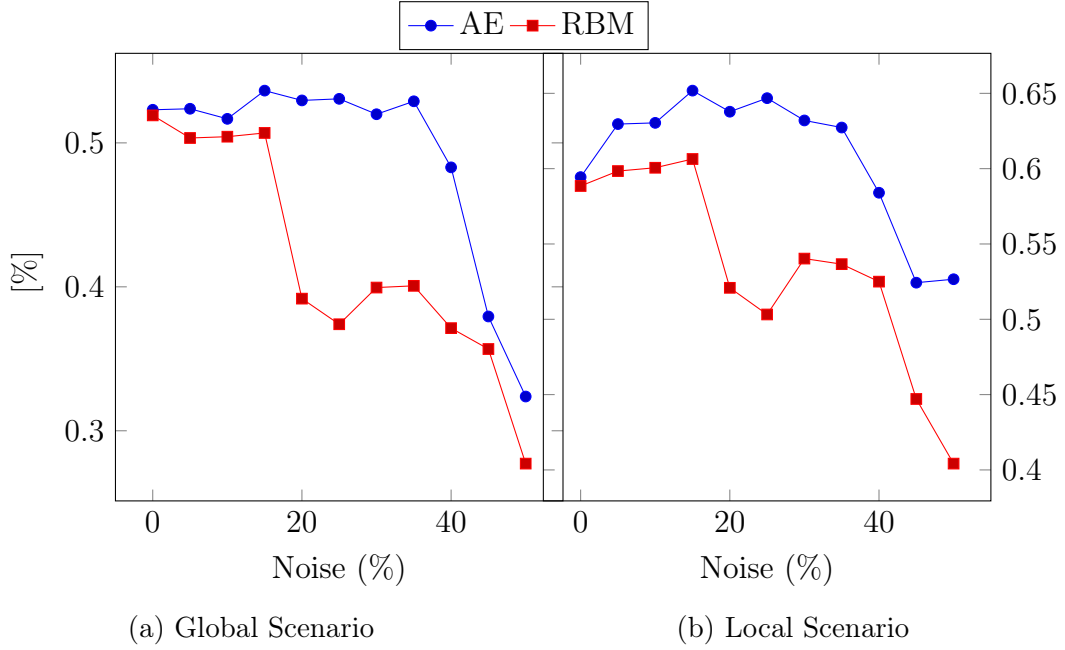


Figure 7.12: Comparison of Denoising Dense Auto-Encoders and Denoising RBM on a Keyword Spotting task, using the George Washington data set.

when some noise is applied, the differences between the two models are rather significant. Indeed, while the RBM is able to withstand about 15% of noise and manages a small improvement in the local scenario, the auto-encoder has very good results up to 35% of noise. Moreover, the improvements for the auto-encoder are more significant, almost 10% in the local scenario and about 2.5% in the global scenario. In this experiment, the denoising auto-encoder is clearly better than the RBM.

The second model that is tested is the single-layer convolutional model with pooling layer from Section 7.4. The first layer has been fixed to five filters. Different amounts of noise are tested.

The results are presented in Figure 7.13. The results are showing several very interesting trends. It can be observed that convolutional models are more resistant to noise and can handle up to 50% noise without too much reduction of performance. This is especially true for CRBM models which are improving quite significantly until the end. Overall, the CRBM model with standard pooling is the best model in this case, especially in the global scenario. Very interestingly, the CRBM model with PMP is taking heavy advantage of the noise. It is starting with very low performance without noise, but its performance is improving steeply as the noise level increases. For the highest level of noise, it performs as well as the CRBM model with standard pooling.

The last denoising model is the deep model with pooling layers from Section 7.4. The first layer has six filters, while the second one has been fixed to five convolutional filters. Different amounts of noise are tested. For the stacked models, the

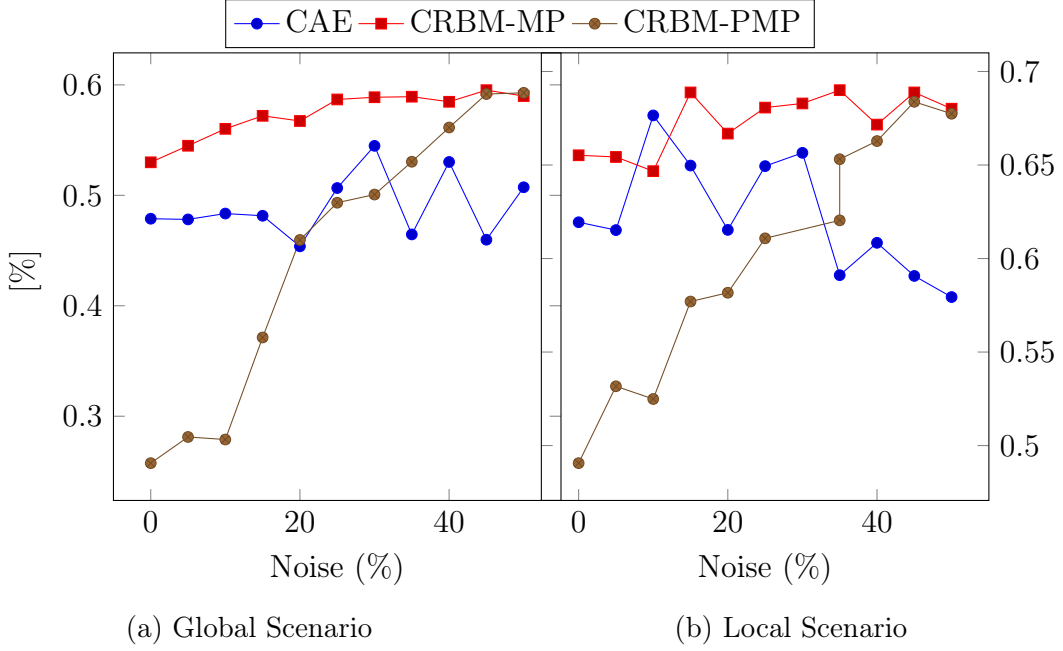


Figure 7.13: Comparison of Denoising Convolutional Auto-Encoders and Denoising CRBM, with pooling, on a Keyword Spotting task, using the George Washington data set.

noise is applied to the input layer of each stacked model.

Figure 7.14 presents the results for this last experiment. First, it can be observed that the stacked model performs worse than the deep model (results 20% and 25% of noise notwithstanding). The stacked model has noise before each of the convolutional layers while the deep model only sees the noise once. It seems that applying noise in the inner layers is detrimental to the performance for this model and task. This may also explain the fact that the deep auto-encoder performs better in this situation than the deep CRBM. Moreover, the CRBM with PMP does not perform as well as it did in the previous experiment in which it was shown to improve significantly with high level of noise. Again, this is due to the noise applied at both layers.

It is clear that corrupting the samples during the training can significantly improve the quality of the features generated by an auto-encoder. While this does not improve the performance of a fully connected RBM, it does improve significantly the performance of the CRBM model, especially when using PMP. When models are stacked, noise applied to the second layer seems detrimental to the quality of the features for this task. However, it improves the performance of the deep model, probably because the noise is only applied to the input and not in front of each stacked model. It would be interesting to compare the difference with the noise only applied to the first stacked model.

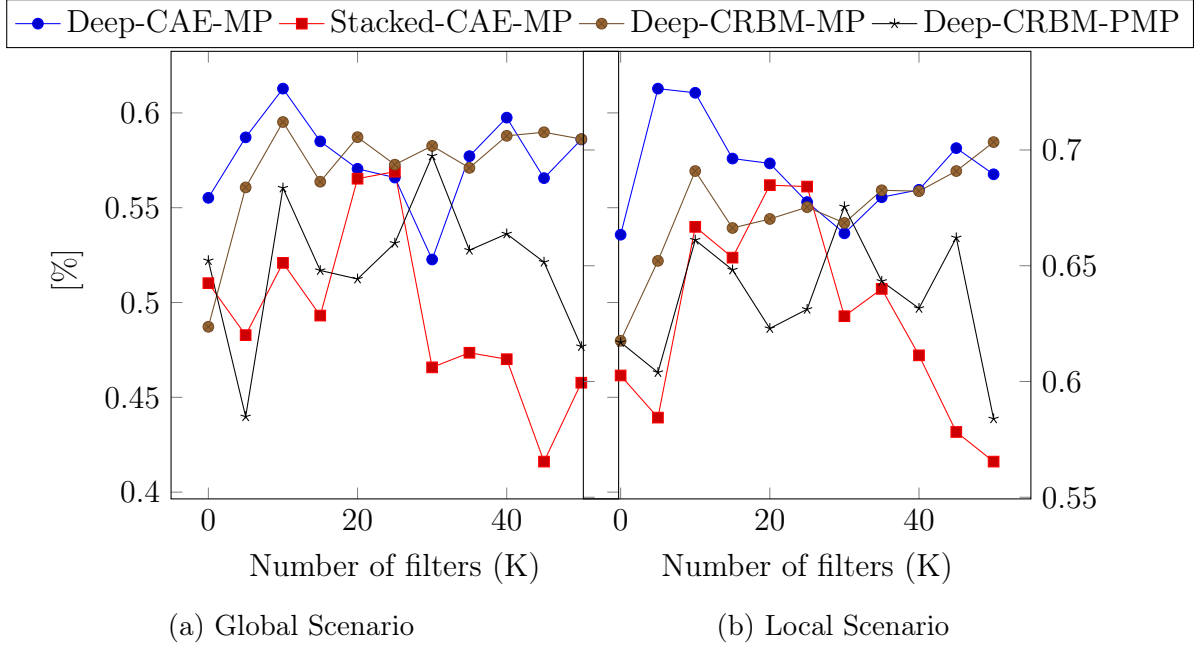


Figure 7.14: Comparison of Denoising Deep Convolutional Auto-Encoders and Denoising Deep CRBM on a Keyword Spotting task, with pooling layers, using the George Washington data set.

7.7 Results

For an overview, Table 7.1 presents a summary of all the results acquired during this experiment. Overall, the best absolute result has been achieved by models based on auto-encoders. The best Average Precision has been achieved by the Deep convolutional model with max pooling (62.6%). The best Mean Average Precision is obtained with the Denoising Deep Convolutional model (72.7%). The best RBM configurations are not far behind with, respectively for AP and MAP, the denoising dense model and the denoising convolutional model (59.5%) and the denoising deep convolutional model (70.3%).

Although the RBM and CRBM models are outperformed in terms of absolute performance, they generally had better overall performance when considering the average performance over different configurations. This is important in practice when the parameters are not intensively tuned for a specific data set.

In the Keyword Spotting experiment (See Chapter 6), better results were obtained for the CRBM models. Indeed, more configurations were explored and more advanced techniques such as sparsity and Rectified Linear Unit (ReLU) features were experimented with. Therefore, it is also possible that the results obtained in this present experiment can be further improved.

Table 7.1: Mean Average Precision (MAP) and Average Precision (AP) for each different model tested during this experiment. Each model includes the best result obtained with the RBM model and the equivalent auto-encoder model. A bold number indicates the best for each experiment.

Model	AE		RBM	
	AP	MAP	AP	MAP
Dense	55.3	64.0	53.4	63.5
Deep Dense	54.1	64.4	55.0	61.3
Convolutional	42.6	57.0	46.8	59.7
Convolutional Pooling	54.9	65.2	54.7	64.2
Deep Conv.	52.1	63.7	54.4	61.9
Deep Conv. Pooling	62.6	71.8	59.3	68.9
Hybrid	52.7	63.1	50.6	61.6
Hybrid Pooling	57.0	68.6	58.5	67.9
Denoising Dense.	53.6	65.2	59.5	60.6
Denoising Conv.	54.5	67.7	59.5	69.0
Denoising Deep Conv.	61.3	72.7	59.0	70.3
Best Result	62.6	72.7	59.5	70.3

7.8 Summary

In this chapter, the features generated by RBM models were compared to the features coming from auto-encoders based on neural networks. Overall, both models have good performance. In the case of single-layer dense models, the RBM proved very powerful but was outperformed by the stacked auto-encoder when multiple layers are used. The CRBM has shown better overall performance and resilience to different configuration than the convolutional auto-encoder. Nevertheless, the highest performance was observed with the latter. The same was observed for hybrid models. The denoising auto-encoder showed better performance than the fully-connected RBM. However, the CRBM was significantly better and more resilient to noise than the single-layer convolutional layer. But again, the highest performance was achieved with the deep denoising convolutional auto-encoder.

In general, RBM models are more complicated to train than the equivalent auto-encoder. CD training is a slightly more complex training method than SGD and has more hyper parameters to tune. Moreover, for some of the experiments, it was necessary to use different training parameters for different layers of the network. This was not necessary for training the auto-encoders. Also, the RBM models are apparently more sensitive to the chosen hyper-parameters and to the random initialization. Another advantage of the auto-encoders, not directly related to this experiment, is that they are trained in the same manner as classification neural network. Therefore, it is easy to find support for auto-encoders in most of the existing machine learning frameworks. This is not the case for RBM for which the support is rather poor.

One advantage of the RBM models over auto-encoders is the significantly faster training. This is especially true for deep models. A deep CAE with two layers and pooling is made of eight layers. This means that significant time will be spent forwarding the data to the end of the network and then back propagating the error to the first layer. This is not the case in the CRBM where backpropagation only occurs inside a layer. For large deep CAE models, the equivalent CDBN was more than three times faster to train. The stacked CAE model alleviates a part of this issue since back propagation does only go through one layer. But there are still more layers and this makes CD training more efficient than SGD.

The main goal of this chapter was to compare the capabilities of RBM models and standard auto-encoders. Although keyword spotting was selected as experiment, this chapter was not meant as an extensive exploration of the use of auto-encoders for keyword spotting. Indeed, in practice, other parameters would have been necessary to test such as learning sparse representation and other types of units such as ReLU. Moreover, the chosen architectures were not tuned as much as it would be necessary to obtain the very best results for the task. Nevertheless, excellent spotting results were achieved, even with this relatively limited exploration of the parameters.

The comparison between the two families of model is not extensive and is only performed on one task and one data set. The results could differ for other tasks or different data sets. Both families of models were also compared several times in different research work. Larochelle et al. showed several examples in which stacked RBM models were outperforming stacked auto-encoders, for unsupervised pretraining (Larochelle, Bengio, et al., 2009). For semi-supervised learning, Erhan et al. observed qualitatively similar results between denoising auto-encoders and RBMs (Erhan, Bengio, A. Courville, Manzagol, et al., 2010). Coates et al. observed slightly better performance for single-layer sparse auto-encoder compared to sparse RBM on two different data sets for object recognition (Coates, H. Lee, and Ng, 2010). Vincent et al. showed almost similar performance for feature learning for a DBN and a Stacked Auto-Encoder (SAE) when denoising was used to train both models (Vincent, Larochelle, Lajoie, et al., 2010). Cho performed a comparison of DAE and RBM for image denoising (Cho, 2013) and found that RBMs were more robust to noise, achieving better performance when the noise level was high but being outperformed on small noise-level images.

References for Chapter 7

- Bengio, Yoshua (2009). “Learning deep architectures for AI”. In: *Foundations and trends® in Machine Learning* 2.1, pp. 1–127 (cit. on pp. 58, 112, 149, 150).
- Bengio, Yoshua, Yann LeCun, and al. (2007). “Scaling learning algorithms towards AI”. In: *Large-scale kernel machines* 34.5 (cit. on p. 150).
- Boureau, Y-lan, Yann LeCun, and Marc’Aurelio Ranzato (2008). “Sparse feature learning for deep belief networks”. In: *Advances in neural information processing systems*, pp. 1185–1192 (cit. on p. 151).

- Cho, Kyunghyun (2013). “Boltzmann machines and denoising autoencoders for image denoising”. In: *arXiv preprint arXiv:1301.3468* (cit. on pp. 143, 162, 167).
- Coates, Adam, Honglak Lee, and Andrew Y. Ng (2010). “An analysis of single-layer networks in unsupervised feature learning”. In: *Ann Arbor* 1001.48109, p. 2 (cit. on pp. 112, 167).
- Doi, Eizaburo, Doru C. Balcan, and Michael S. Lewicki (2005). “A theoretical analysis of robust coding over noisy overcomplete channels”. In: *Advances in neural information processing systems*, pp. 307–314 (cit. on p. 151).
- Erhan, Dumitru, Yoshua Bengio, Aaron Courville, Pierre-Antoine Manzagol, et al. (2010). “Why does unsupervised pre-training help deep learning?”. In: *Journal of Machine Learning Research* 11.Feb, pp. 625–660 (cit. on pp. 59, 60, 167).
- Goodfellow, Ian J. et al. (2014). “Generative adversarial nets”. In: *Advances in Neural Information Processing Systems*, pp. 2672–2680 (cit. on p. 151).
- Hinton, Geoffrey E. and Richard S. Zemel (1994). “Autoencoders, minimum description length, and Helmholtz free energy”. In: *Advances in neural information processing systems*, pp. 3–3 (cit. on p. 150).
- Kingma, Diederik P., Shakir Mohamed, et al. (2014). “Semi-supervised learning with deep generative models”. In: *Advances in Neural Information Processing Systems*, pp. 3581–3589 (cit. on p. 151).
- Kingma, Diederik P. and Max Welling (2013). “Auto-encoding variational bayes”. In: *arXiv preprint arXiv:1312.6114* (cit. on p. 151).
- Larochelle, Hugo, Yoshua Bengio, et al. (2009). “Exploring strategies for training deep neural networks”. In: *Journal of Machine Learning Research* 10.Jan, pp. 1–40 (cit. on pp. 59, 60, 167).
- Larsen, Anders Boesen Lindbo, Søren Kaae Sønderby, and Ole Winther (2015). “Autoencoding beyond pixels using a learned similarity metric”. In: *CoRR* abs/1512.09300. URL: <http://arxiv.org/abs/1512.09300> (cit. on p. 151).
- Makhzani, Alireza et al. (2015). “Adversarial autoencoders”. In: *arXiv preprint arXiv:1511.05644* (cit. on p. 151).
- Masci, Jonathan et al. (2011). “Stacked convolutional auto-encoders for hierarchical feature extraction”. In: *Artificial Neural Networks and Machine Learning—ICANN 2011*, pp. 52–59 (cit. on pp. 58, 150).
- Mescheder, Lars M., Sebastian Nowozin, and Andreas Geiger (2017). “Adversarial Variational Bayes: Unifying Variational Autoencoders and Generative Adversarial Networks”. In: *CoRR* abs/1701.04722. URL: <http://arxiv.org/abs/1701.04722> (cit. on p. 151).
- Noh, Hyeonwoo, Seunghoon Hong, and Bohyung Han (2015). “Learning deconvolution network for semantic segmentation”. In: *Proceedings of the IEEE International Conference on Computer Vision*, pp. 1520–1528 (cit. on p. 150).
- Olshausen, Bruno A. and David J. Field (1997). “Sparse coding with an overcomplete basis set: A strategy employed by V1?”. In: *Vision research* 37.23, pp. 3311–3325 (cit. on p. 151).
- Radford, Alec, Luke Metz, and Soumith Chintala (2015). “Unsupervised representation learning with deep convolutional generative adversarial networks”. In: *arXiv preprint arXiv:1511.06434* (cit. on p. 151).

- Ranzato, Marc'Aurelio and Yann LeCun (2007). "A sparse and locally shift invariant feature extractor applied to document images". In: *Ninth International Conference on Document Analysis and Recognition (ICDAR 2007)*. Vol. 2. IEEE, pp. 1213–1217 (cit. on p. 151).
- Rezende, Danilo Jimenez, Shakir Mohamed, and Daan Wierstra (2014). "Stochastic backpropagation and approximate inference in deep generative models". In: *arXiv preprint arXiv:1401.4082* (cit. on p. 151).
- Rumelhart, David E., Geoffrey E. Hinton, and Ronald J. Williams (1988). "Learning representations by back-propagating errors". In: *Cognitive modeling* 5.3, p. 1 (cit. on p. 149).
- Sohn, Kihyuk, Honglak Lee, and Xinchun Yan (2015). "Learning structured output representation using deep conditional generative models". In: *Advances in Neural Information Processing Systems*, pp. 3483–3491 (cit. on p. 151).
- Tang, Yichuan, Ruslan R. Salakhutdinov, and Geoffrey E. Hinton (2012). "Robust boltzmann machines for recognition and denoising". In: *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*. IEEE, pp. 2264–2271 (cit. on pp. 143, 162).
- Vincent, Pascal, Hugo Larochelle, Yoshua Bengio, et al. (2008). "Extracting and composing robust features with denoising autoencoders". In: *Proceedings of the 25th international conference on Machine learning*. ACM, pp. 1096–1103 (cit. on pp. 58, 59, 151).
- Vincent, Pascal, Hugo Larochelle, Isabelle Lajoie, et al. (2010). "Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion". In: *Journal of Machine Learning Research* 11.Dec, pp. 3371–3408 (cit. on pp. 151, 153, 167).

Part III

Conclusion

Chapter 8

Conclusion

*All we have to decide is what to do
with the time that is given to us*

J.R.R. Tolkien

Contents

8.1	Synthesis	173
8.2	Perspectives	176

8.1 Synthesis

In this thesis, the use of Deep Learning techniques to automatically extract features from images in an unsupervised manner was investigated. More specifically, the research focuses on the original branch of Deep Learning, the so-called "Hinton approach", using the Restricted Boltzmann Machine (RBM) and Convolutional Restricted Boltzmann Machine (CRBM) models. This approach was validated using several experiments in order to observe its effectiveness and also the associated difficulties.

In the Introduction, several scientific research questions were presented. Our contributions in regard to these questions are summarized below:

1. The first question concerned the advantages of unsupervised pretraining in the context of classification problems.
It was seen during our experiment on Sudoku recognition that unsupervised pretraining acted as a strong regularizer. It was shown that the training was much faster after pretraining than it was without. Moreover, it was also observed that pretraining was avoiding overfitting in early stages of training.
2. The second question was whether pretraining was acting in the same way for dense layers and for convolutional layers.

On a digit recognition task, it was observed that there were some significant differences in how unsupervised pretraining was impacting the training of fully-connected layers and convolutional layers. Indeed, the regularization effect is not significant for convolutional layers. However, it can clearly be observed that it provides a very good initialization of the weights that allows to significantly speed up the training of the network.

3. The third question was about the advantages of automatic feature learning when compared to handcrafted features.

On a keyword spotting task on handwritten historical documents, a convolutional feature extractor trained to automatically learn features from patches of word images was able to outperform three different sets of handcrafted features. The results were confirmed on three different data sets. However, even if the features are learned automatically, it is still necessary to configure a large number of parameters and good results were requesting a significant amount of tuning.

4. The next one concerned the difficulty of generating features for basic classifiers.

When working with basic classifiers such as the Dynamic Time Warping (DTW) for keyword spotting, the dimensionality of the features has been a very important factor. Moreover, the normalization of the features has also been a factor of significant importance. Nevertheless, it was shown that with sufficient tuning of the network architecture and training parameters, it was possible to generate good features for basic classifiers and achieve better performance than the reference feature sets on several data sets.

5. The fifth question was whether it was possible to use the same features with different classifiers.

When using the same feature extractor model on two very different classifiers, DTW and Hidden Markov Model (HMM), it was shown that it was possible to achieve very good performance in both cases. This is showing that the features that can be learned with these models are rather generic. This was made without any changes to the architecture or training parameters between the two classifiers. However, different classifiers have different strengths and weaknesses and therefore it is very difficult to optimize for very different classifiers without losing overall performance.

6. The last question concerned the difference between the RBM and CRBM approach for feature learning compared to different alternatives and more specifically standard auto-encoders.

On a keyword spotting task, the performance of RBM models and regular auto-encoders have been compared. This showed an advantage for RBM models when using a single layer and an advantage to deep auto-encoder models when several layers were used. This also showed that it was more complex to train and tune an RBM than a simple auto-encoder. Once carefully tuned, both models were capable of generating very good features and a

large number of models can be composed to generate different features more suited to a specific task.

The contributions regarding the specific questions can be summarized as such:

1. The first question was whether a single network can learn from two different types of inputs in the context of Sudoku recognition.

By using a single network to learn both handwritten and computer generated digits on a Sudoku recognition task, it was observed that it was not necessary to separate these inputs before training. Using a single network with enough learning capacity is more than enough for this type of dual inputs.

2. The second specific question was whether features learned on grayscale images are better than features learned on binary images in the context of handwritten Keyword Spotting.

Since no grayscale segmented word images were available for training our model, it was trained on binary images on which a Gaussian blur filter was applied, making them artificially grayscale. When using DTW as a classifier, the performance of the system was significantly improved. However, when using HMM, the performance was improved in one scenario and reduced in another. Although there is potential in learning from grayscale images rather than binary images, the learning process using Gaussian visible units is very unstable and needs lower learning rates to work.

3. The last specific questions addressed the performance of the framework developed during the course of this thesis in comparison to popular machine learning frameworks.

On four different machine learning experiments, the training quality of our developed framework (DLL) proved to be comparable to the other five tested frameworks in terms of accuracy. Moreover, it was also the framework with the most features for RBM and CRBM training and usage. Nevertheless, the framework clearly lacks features for regular neural network training. On the performance side, it is among the fastest on Central Processing Unit (CPU), thanks to many processor and memory optimizations that were included in the developed framework. However, it is not taking advantage enough of the Graphical Processing Unit (GPU), for which few optimizations were included.

To summarize, it was found during this research that it is possible to generate very robust features from images. On several experiments, the features learned by the trained models were superior to handcrafted features. Moreover, learned features have several advantages over them. Indeed, as they are learned on data, they are easier to adapt to different data sets. Finally, we believe that there is still a large potential for unsupervised learning for feature extraction, especially when working on images.

More specifically, this research has shown that the RBM model and its variants were good candidates to learn a feature extractor on several experiments. We also

performed a comparison of RBM with auto-encoders on a feature extraction task using the same data set. The auto-encoders show advantages and disadvantages over RBM. They shown, on that specific data set and when using DTW as a classifier, slightly better overall performance than RBM and revealed easier to train with less hyper-parameters. However, RBM models revealed more stable in terms of performance across larger rangers of model architectures making them good candidates in context of of shared configurations. Moreover, while Deep Learning via RBM and CRBM (the "Hinton approach") was the new trend for a few years after 2006, it was quickly superseded by new generations of Convolutional Neural Networks and new techniques for training larger and larger models. These models have new capabilities and are easier to train, especially since they do not require semi-supervised training. Moreover, it should also be mentioned that while unsupervised pretraining helps both in regularizing the training and in providing a good initialization of the weights, it also increases the complexity of training the network. Nevertheless, it was shown that for several problems, unsupervised pretraining was still useful. Indeed, for problems with a small amount of available labeled data, the use of unlabeled data which usually comes in large quantities can still greatly improve the final results. However, problems with large amount of labeled data many not anymore benefit from unsupervised pretraining.

8.2 Perspectives

A usable and general-purpose framework was developed during the course of this research. Although it is ready to be used by other researchers and compared well to other popular machine learning tools, more work is necessary in order to make it more competitive. While its RBM and CRBM support is excellent, it would be necessary to enhance its general neural network learning facilities, for instance adding Batch normalization or supporting a wider range of activation functions. While the framework supports a large range of feed-forward neural networks, support for Recurrent Neural Networks (RNNs) such as Long Short Term Memorys (LSTMs) would also be a great advantage. It would also be very important to make better use of the GPU to accelerate training and be more competitive in this area. Moreover, making the framework easier to use for the research community and expanding its documentation would also be necessary.

Regarding the Sudoku experiment, several improvements would still be necessary to complete it. Indeed, the detection process is still very complicated in terms of the used algorithms and its performance are not highly satisfactory. Since the classification model is trained on the detect images, it is important that the detection are as accurate as possible. It would be more adequate to use a learning approach for the detection as well.

The keyword spotting results could also be improved. It was shown that auto-encoders are also very good candidates to generate features for this task on the George Washington data set when using DTW. It would prove highly interesting to

complete this evaluation with several data sets and different classifiers. Moreover, it would also be very interesting to use auto-encoders to their full extent and fine-tune the architecture to use them instead of the CRBM to see if better performance can be obtained. Fine-tuning the system for the HMM classifier would also surely to better final results. Moreover, instead of using a patch-by-patch approach for feature extraction, a holistic approach could also greatly improve the performance, but such holistic features are significantly more difficult to generate with these techniques, especially considering the dimensionality and variability of the images.

If we may now draw a final word, we observed in this thesis and through the work of many other researchers that machine learning techniques have grown more and more powerful over the last decade. Big steps have been taken with many new techniques such as deep networks trained layer-by-layer for feature extraction and approaches that have allowed to reach robustness by incorporating different granularities of information. However, while such systems are achieving outstanding results, they have also grown more and more complex. For solving a given task, a researcher is now overwhelmed by the choice of different model families with few insights on which one to pick. Once a model is chosen, there is also a large number of hyper-parameters to tune to reach the best results. The community is using more and more the term *cooking* to express this frequent random exploration of models and associated hyper-parameters. Along the work undertaken in this thesis and through discussions with colleagues of research, we could observe that, for a given task, the *cooking time* is increasing while our understanding of what is going on in the models is decreasing. Often, the reasons why a given model is superior to another are not especially clear. Nevertheless, let's see the optimistic side of this : it will surely trigger new research thesis in the direction of simplification and interpretability of deep approaches.

Part IV

Appendices

Appendix A

Detailed Performance Analysis

*Premature optimization is the root
of all evil*

Donald Knuth

Contents

A.1	Introduction	181
A.2	Configuration	182
A.3	Valid Convolution	182
A.4	Full Convolution	184
A.5	GPU Performance	184
A.5.1	Matrix-Matrix Multiplication	185
A.5.2	Fast Fourier Transform	187
A.5.3	Batch Valid Convolution	187
A.5.4	Batch Full Convolution	190
A.5.5	Conclusion	190
A.6	References for Appendix A	192

A.1 Introduction

This appendix presents detailed results about some performance benchmarking that has been performed during the course of this thesis. The performance of different implementations of the valid and full convolution operations is compared in the first two sections. Then, the performance of Graphical Processing Unit (GPU) implementations is compared to the performance of optimized Central Processing Unit (CPU) implementations.

A.2 Configuration

All the results presented in this chapter have been computed on a Gentoo Linux machine, on a Linux kernel 4.4, with 12 GB of RAM, running an Intel® Core™ i7-2600. The processor is running at a frequency of 3.40GHz (CPU frequency scaling has been disabled for the purpose of these tests). Both Streaming SIMD Extensions (SSE) and Advanced Vector eXtensions (AVX) were enabled on the machine. BLAS operations are executed with the Intel® Math Kernel Library (MKL). The GPU used for the benchmarks is a Nvidia Geforce GTX 960 card. This is a consumer desktop card, that is especially optimized for single-precision workloads. CUDA 8.0.44 was used as well as CUDNN v5.

The tests were written in C++ using our own libraries: Deep Learning Library (DLL)¹ and Expression Templates Library (ETL)² (See Chapter 4 for details). Each program has been compiled with GNU Compiler Collection (GCC) 4.9, with compiler flags selected for maximum native performance.

A.3 Valid Convolution

Convolution operations are known to be computationally very heavy and highly memory-bound. A standard implementation of a valid convolution is easy to develop but is rarely efficient enough for use in a Convolutional Neural Network (CNN) or a Convolutional Restricted Boltzmann Machine (CRBM). The first way to come around this issue is to use a vectorized implementation of the algorithm to perform several floating point operations in one CPU cycle. The second possible optimization is to reduce the convolution operation to a General Matrix Multiplication (GEMM) operation by rearranging the input matrix (see Section 4.4.2). Table A.1 shows the performance of these different versions for a single-precision valid convolution. The vectorized convolution is always faster than the other implementations. The fact that the GEMM implementation is slower than the vectorized version (and sometimes slower than the standard version) comes from the overhead of the matrix rearranging and from the fact that this computes a vector-matrix multiplication rather than a matrix-matrix multiplication. This can also be observed by the very small difference between the different GEMM implementation. Indeed, if the bulk of the computation was done inside the matrix multiplication, the difference would be much larger between naive matrix multiplication and the MKL implementation. Moreover, the parallel matrix-matrix multiplication is slower than the serial version.

In practice and especially for neural network training, a single image is almost always convolved with several kernels and the results are either summed or all kept, depending on the situation. The reduction to the multiplication becomes more interesting since the slow matrix transformation for rearranging the input

¹<https://github.com/wichtounet/dll/>

²<https://github.com/wichtounet/etl/>

Table A.1: Performance of a single-precision valid convolution, in microseconds. A is the naive matrix multiplication, B is the MKL version and C is the parallel MKL version. The speedup is comparing the GEMM(B) version with the Vectorized version. The Vectorized version uses SSE and AVX to perform up to 8 single-precision floating points in one CPU cycle.

Image Kernel	12x12 5x5	16x16 5x5	16x16 9x9	28x28 9x9	50x50 17x17	128x128 17x17	128x128 31x31	256x256 31x31
Standard	1.686	3.710	4.594	28.236	295.778	3207	8154	43471
Vectorized	0.805	1.651	1.229	9.067	66.045	694	1841	10660
GEMM(A)	1.732	2.563	5.253	15.741	174.217	2930	13253	71439
GEMM(B)	1.664	2.162	4.666	13.317	156.708	2884	12946	67289
GEMM(C)	1.713	2.554	5.111	23.750	153.984	3034	13815	66949
Speedup	0.481	0.744	0.253	0.690	0.430	0.210	0.131	0.149

Table A.2: Performance of a single-precision valid convolution of 1 image with 100 kernels, in milliseconds. The speedup is comparing the GEMM(C) version with the Vectorized version. The Vectorized version uses SSE and AVX to perform up to 8 single-precision floating points in one CPU cycle.

Image Kernel	12x12 5x5	16x16 5x5	16x16 9x9	28x28 9x9	50x50 17x17	128x128 17x17	128x128 31x31	256x256 31x31
Standard	0.953	2.126	2.040	12.653	82.027	889.029	1527	8075
Vectorized	0.081	0.179	0.196	0.964	6.621	70.181	188.208	1062
GEMM(A)	1.561	1.633	1.629	2.131	7.351	67.983	174.021	921
GEMM(B)	0.014	0.026	0.039	0.181	2.290	19.818	55.596	293
GEMM(C)	0.009	0.014	0.029	0.102	0.766	9.266	25.911	119
Speedup	9	12.78	6.758	9.45	8.643	7.574	7.263	8.924

only needs to be done once for the image and can be used for several kernels. Moreover, a single matrix-matrix multiplication is performed per image instead of one inefficient vector-matrix multiplication per kernel. Results when convolving an image with 100 kernels are presented in Table A.2. In that case, the parallel GEMM version is always significantly faster than the vectorized version. The speedups are ranging from 6.7 to 12.7, depending on the configuration of the convolutions. The speedup gets even better when more kernels are used. This time it becomes clear that the performance of the MKL is largely superior to a naive implementation. Moreover, since the matrices are now much bigger, it becomes interesting to use the parallel version of the MKL that is about twice faster than the standard version on this experiment.

From the results, it can be deduced that a valid convolution of one image with one kernel should always be computed using a vectorized algorithm. When the image is convolved with enough kernels at once, it should be reduced to a matrix multiplication algorithm. Moreover, a parallel implementation of the matrix multiplication should be used.

A.4 Full Convolution

A full convolution is very similar to a valid convolution except that it produces an image bigger than the source image ($O \triangleq I + K - 1$). This makes the code harder to optimize than for the full convolution since it contains several conditions that limit the possibilities of vectorization. Nevertheless, it is still possible to write an optimized version with SSE and AVX, by computing several floating point operations in one CPU cycle. Another possible optimization is to reduce the full convolution to a Fast Fourier Transform (FFT) operation (see Section 4.4.2). There is some overhead since both input matrices must be padded with zeroes to the size of the output matrix. Moreover, the performance is highly dependent on the efficiency of the FFT implementation. In our tests, the FFT implementation from the Intel MKL library was used. Table A.3 shows the performance of these different versions for a single-precision full convolution. For very small dimensions, even the vectorized version does not beat the standard version and the FFT version only gets better than the vectorized from medium images. The FFT speedup ranges from 0.2 to 33.5 and are generally increasing with the image and kernel size. Even for such small images, it already becomes interesting to perform the FFT in parallel. This is observed as the parallel version of the FFT begins to be faster than the standard version at the same point where the FFT becomes faster than the vectorized version. Moreover, for large images, even an handcrafted FFT can be faster than the vectorized version.

Again, a single image is often convolved with N kernels at once to produce N output images. Handling this special case directly in the implementation leads to better performance. Indeed, the FFT and padding of the input image only needs to be done once and more computations can be vectorized. Results for the full convolution of an image with 100 kernels are presented in Table A.4. It can be observed that the FFT becomes interesting even for smaller matrices. Moreover, the FFT speedups are a bit higher than before. Finally, the overall speedup (FFT version compared to the standard version) are significantly higher. Indeed, the FFT version for the largest configuration is 116 times faster than the standard version while it only was 67 times faster when considering only one kernel at once.

To conclude, a full convolution of an image with one or more kernels should be computed by a vectorized algorithm for small matrices and by a FFT reduction for large matrices. When the dimensions are large enough, a parallel version of the FFT computation should also be considered.

A.5 GPU Performance

Over the last years, the use of GPU as computational unit for machine learning has increased exponentially. Indeed, one can make use of their massive parallel capabilities to compute some operations very quickly (Owens et al., 2007). While some very massive speedups were claimed early, it is likely that the speedups are

Table A.3: Performance of a single-precision full convolution, in microseconds. The speedup is comparing the FFT(C) version with the Vectorized version. The Vectorized version uses SSE and AVX to perform up to 8 single-precision floating points in one CPU cycle.

Image Kernel	12x12 5x5	16x16 5x5	16x16 9x9	28x28 9x9	50x50 17x17	128x128 17x17	128x128 31x31	256x256 31x31
Standard	4.165	7.119	21.411	62.198	690.085	4430	14501	57436
Vectorized	4.527	7.461	16.913	45.658	375.189	2202	6941	28495
FFT(A)	22.499	34.862	44.086	88.335	601.149	1357	27399	11945
FFT(B)	10.563	21.739	23.821	39.652	98.685	311.238	1565	2083
FFT(C)	17.041	34.716	32.849	42.511	81.734	187.001	596	850
Speedup	0.265	0.21	0.51	1.07	4.590	11.775	11.645	33.523

Table A.4: Performance of a single-precision full convolution of 1 image with 100 kernels, in milliseconds. The speedup is comparing the FFT(C) version with the Vectorized version. The Vectorized version uses SSE and AVX to perform up to 8 single-precision floating points in one CPU cycle.

Image Kernel	12x12 5x5	16x16 5x5	16x16 9x9	28x28 9x9	50x50 17x17	128x128 17x17	128x128 31x31	256x256 31x31
Standard	2.670	4.461	10.689	29.137	210.461	1238	2964	11172
Vectorized	0.460	0.755	1.706	4.585	36.657	199.421	699	2835
FFT(A)	2.591	2.737	3.647	7.130	43.931	113.872	1869	913
FFT(B)	0.477	1.098	1.317	2.433	6.931	26.552	107.597	178.677
FFT(C)	0.816	1.339	1.553	2.160	4.928	18.359	52.245	96.296
Speedup	0.57	0.56	1.09	2.12	7.43	10.862	13.37	29.44

ranging from 2 to 10 times faster (V. W. Lee et al., 2010) and some operation are not necessarily faster on a GPU.

As a reference, this section compares the performance of different algorithms on CPU and GPU. For each operation, the time used to transfer the data from CPU to GPU and back is included in the measure. In practice, this time could be mitigated if all operations are done on the GPU.

For benchmarking, four different operations have been selected. These operations have been selected for their use in neural network training. In each case, several CPU versions are compared to the GPU version.

A.5.1 Matrix-Matrix Multiplication

The matrix-matrix multiplication is a very expensive operation, with a time complexity of $O(N^3)$. Although there exists algorithm with lower complexity, in practice, they are slower than a carefully optimized standard algorithm.

This benchmark compares five implementations of Matrix-Matrix multiplication:

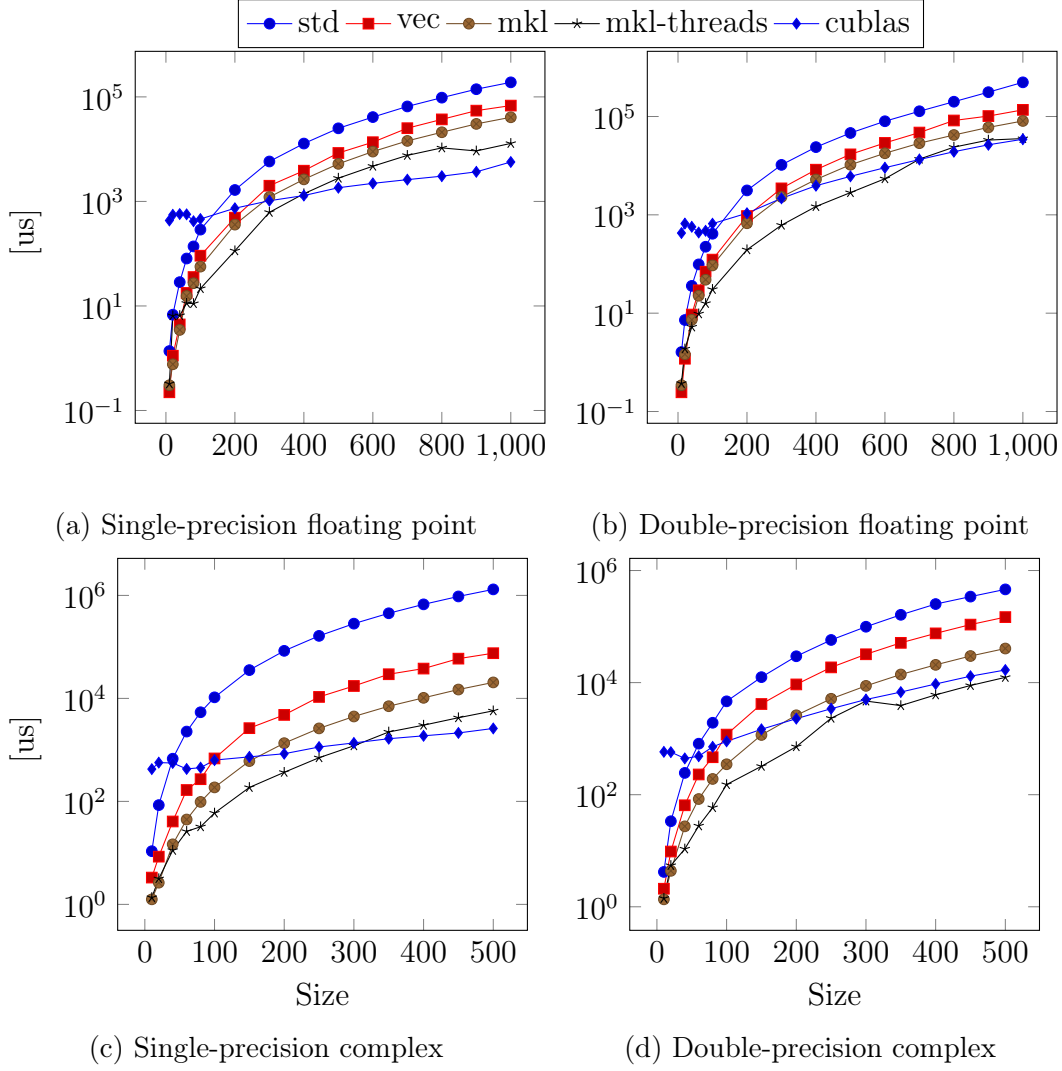


Figure A.1: Comparison of the performance of Matrix-Matrix Multiplication on CPU and GPU, on different floating point precisions.

- *std*: A naive CPU implementation
- *vec*: A carefully tuned CPU implementation of our doing
- *mkl*: The CPU implementation from the Intel MKL library
- *mkl-threads*: The CPU parallel implementation from the Intel MKL library
- *cublas*: The GPU implementation from the NVIDIA CUBLAS library

Figure A.1 presents the results of this benchmark with different floating point precisions. Of all the CPU implementations, apart from very small sizes, the *mkl-threads* version is clearly, and as expected, the fastest version. On a single-precision floating point benchmark, which is the most used in deep learning, it can be observed that the GPU version starts overperforming the CPU version when

multiplying matrices bigger than 400x400. After this point, the GPU version is about two times faster than the CPU version. On a double-precision benchmark, the CPU implementation is always at least as fast as the GPU version even for large matrices. On complex numbers, the same conclusions can be observed, with the GPU being faster on single precision and slower on double precision. The single precision results confirm the results presented in (V. W. Lee et al., 2010) for the SGEMM operation.

The large difference between single-precision and double-precision comes from the fact that the tested GPU is a consumer card which is especially optimized for single precision. Only some of the professional GPUs from NVIDIA are optimized for double-precision as well.

A.5.2 Fast Fourier Transform

The FFT computes the Discrete Fourier Transform (DFT) of a sequence in an optimized manner, with a complexity of $O(N \log N)$. This is an operation that is very computation-intensive.

Four FFT implementations are compared:

- *std*: Handcrafted, slightly optimized, parallel for several transforms, implementation
- *mk1*: The CPU implementation from the Intel MKL library
- *mk1-threads*: The parallel CPU implementation from the Intel MKL library
- *cufft*: The GPU implementation from the NVIDIA CUFFT library

Figure A.2 presents the results of this benchmark. The first row is a single transform of the given size, while the second line is 512 transforms of the given size. It is very interesting to see that even on a single precision workload, the GPU version is very close to the best CPU version, but slightly slower. When several transforms are done at once, the GPU version does not come close to the CPU version, being around two times slower. This probably shows that the GPU version is optimized for large FFT and not for multiple smaller FFTs. It is also interesting that the parallel MKL version is not significantly faster than the sequential MKL version.

A.5.3 Batch Valid Convolution

The valid convolution is used during forward propagation in CNNs. Generally, this is done at once for several batches of images with multiple input channels and several kernels at once. This is an operation that is sometimes referred as a 4D convolution, but the convolutions are done on 2D images.

Several different implementations are compared:

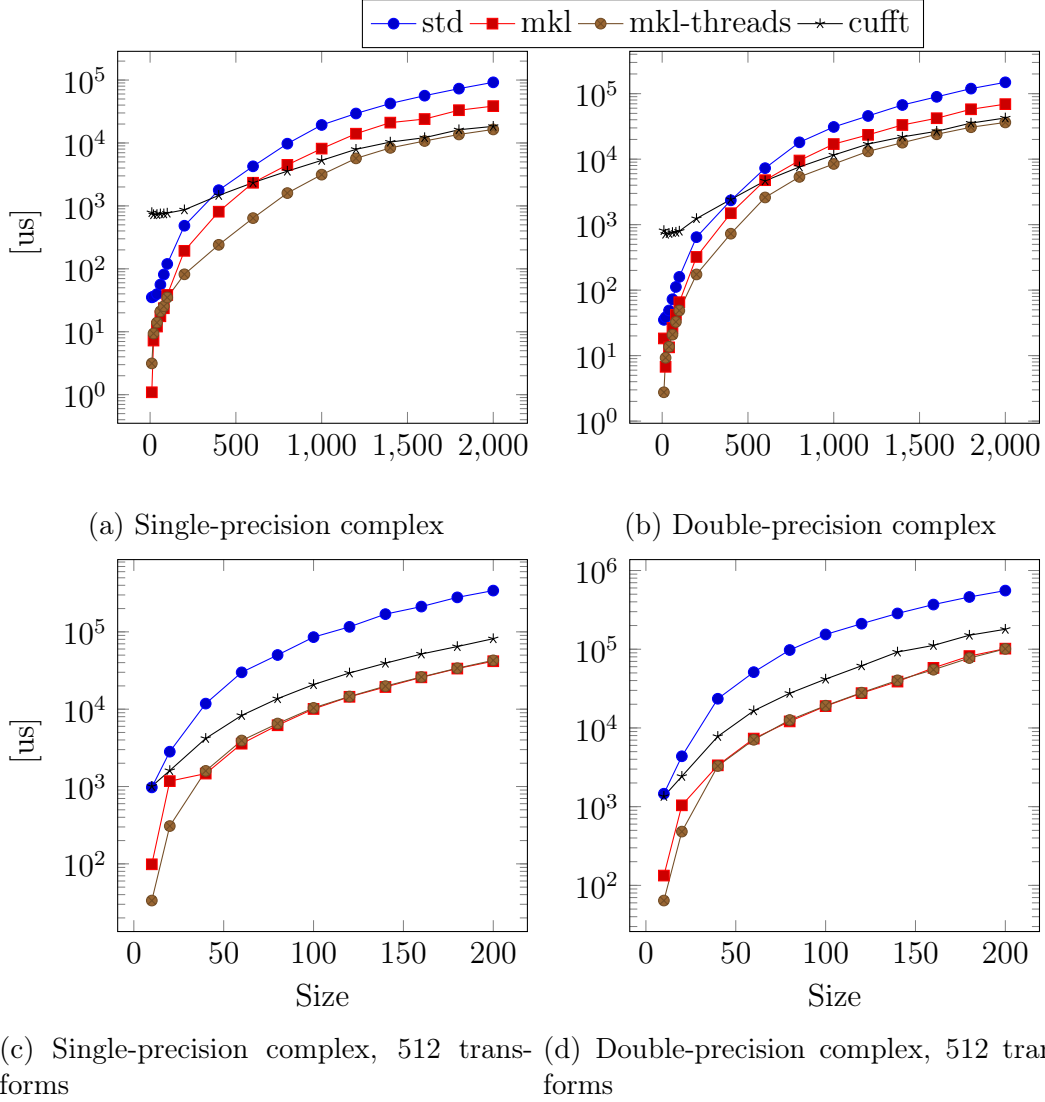


Figure A.2: Comparison of the performance of the Fast-Fourier Transform on CPU and GPU, on different floating point precisions.

- *vec*: Optimized CPU parallel version
- *blas*: Convolution with Matrix-Matrix multiplication, parallel version (see Section 4.4.2)
- *cudnn*: GPU version from the NVIDIA CUDNN library

A simple handcrafted algorithms is not used since it would be too slow already for the dimensions tested in this benchmark. From our evaluation of performance, an unoptimized algorithm for this operation is almost two orders of magnitude slower than the other tested alternatives. Unfortunately, there is no standard CPU implementation of the convolution such as BLAS.

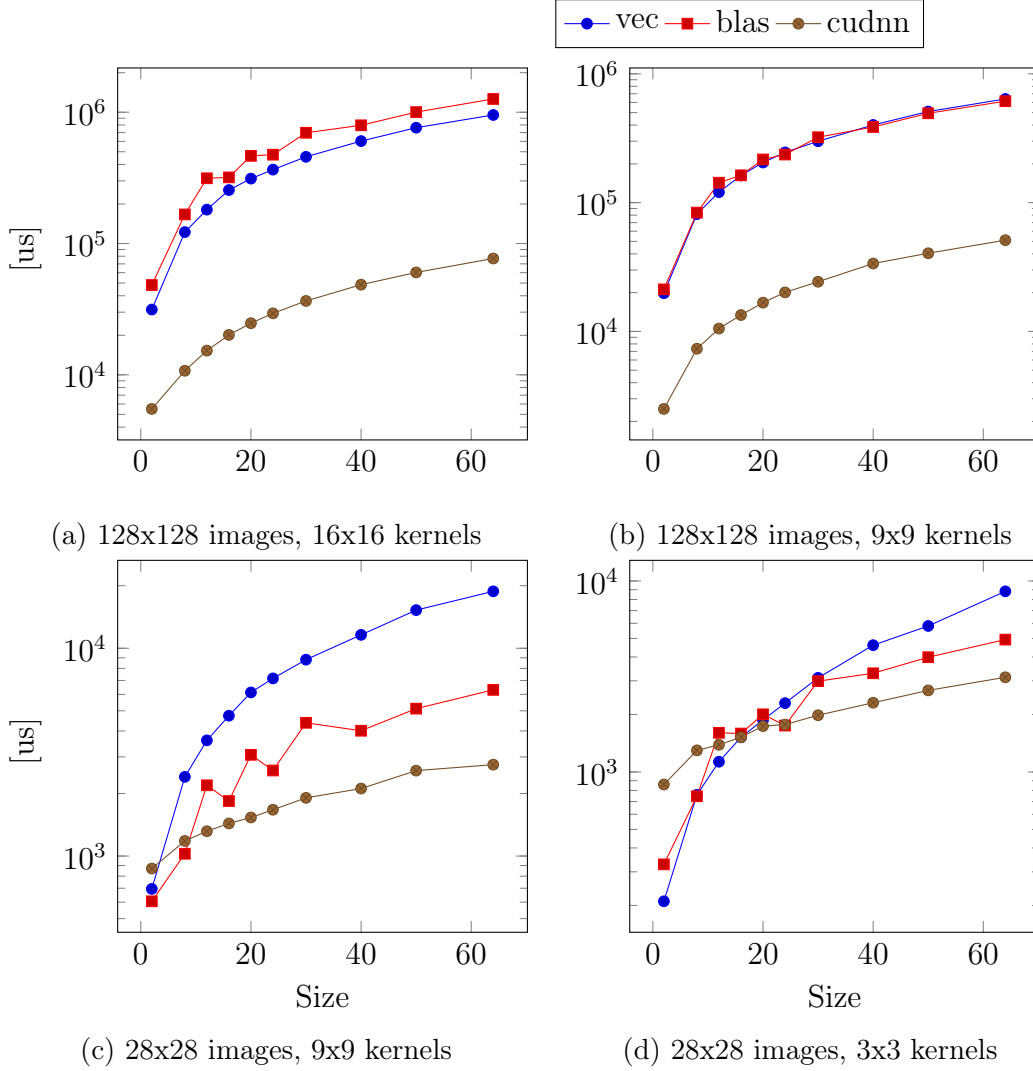


Figure A.3: Comparison of the performance of the Batch Valid Convolution on CPU and GPU, with different image and kernel sizes.

Since this operation is only used for training CNNs, only its single-precision version is benchmarked.

Figure A.3 details the result of this test. Several different image and kernel sizes are tested with increasing numbers of images. The number of kernels and the number of input channels is kept constant. On this benchmark, it is quite clear that the GPU version is much faster than the other versions. On large images (results from the first row), the *cudnn* version is more than one order of magnitude faster than any of the CPU versions. And this is the case even for very few images. When images are small, the differences are less important, but the *cudnn* version still manages to be at least twice faster than *vec* and *blas*, except with very small number of images. In this case, the GPU version really shines with the very high computational cost of the convolution operation. Interestingly, the speedups observed in this experiment are higher than those observed in (V. W. Lee et al.,

2010). This may be explained by a slower CPU implementation used here or by the excellent performance of the CUDNN library that was not available at the time.

A.5.4 Batch Full Convolution

The full convolution is used during back propagation in CNNs. Again, this is done on several images and with several kernels at once. The main difference between this operation and the valid version is that it is harder to optimize since there are many border cases to handle.

For this operation, three implementations are compared:

- *vec*: Optimized CPU parallel version
- *fft*: Convolution with Fast Fourier Transform, parallel version (see Section 4.4.2)
- *cudnn*: GPU version from the NVIDIA CUDNN library

Again, no unoptimized naive algorithm is benchmarked, it would be several orders of magnitude slower than the optimized CPU versions. Since this operation is only used for training CNNs, only its single-precision version is measured.

Figure A.4 presents the performance of the different versions. Again, several different image and kernel sizes are tested and the number of images is increased. On this benchmark, it's definitely clear that the *cudnn* version is much faster than the two CPU contenders. The GPU version is not significantly faster than the valid version, but the CPU versions are much slower on a full convolution than on a valid convolution. In fact, it is much more difficult to optimize for a full convolution given the amount of padding that is performed and the very high memory bandwidth required. On some of the configuration tested, the *cudnn* version is almost two orders of magnitude faster than the *fft* or *vec* versions.

A.5.5 Conclusion

While a GPU version is quickly much faster than a naive version of complex algorithms, the speedups are much lower when compared with a highly optimized CPU implementation. Moreover, the cost of transferring data to and from the GPU is not negligible. This is why it is generally only interesting to use the GPU for large problems or for very computation-heavy problems. Otherwise, the gain of extra parallelization is canceled by the cost of transferring the data.

Moreover, it is also very important to consider the precision of computation. Indeed, most GPUs are highly optimized for single precision but their double precision performance is quite poor. This is not an issue for Artificial Neural Network

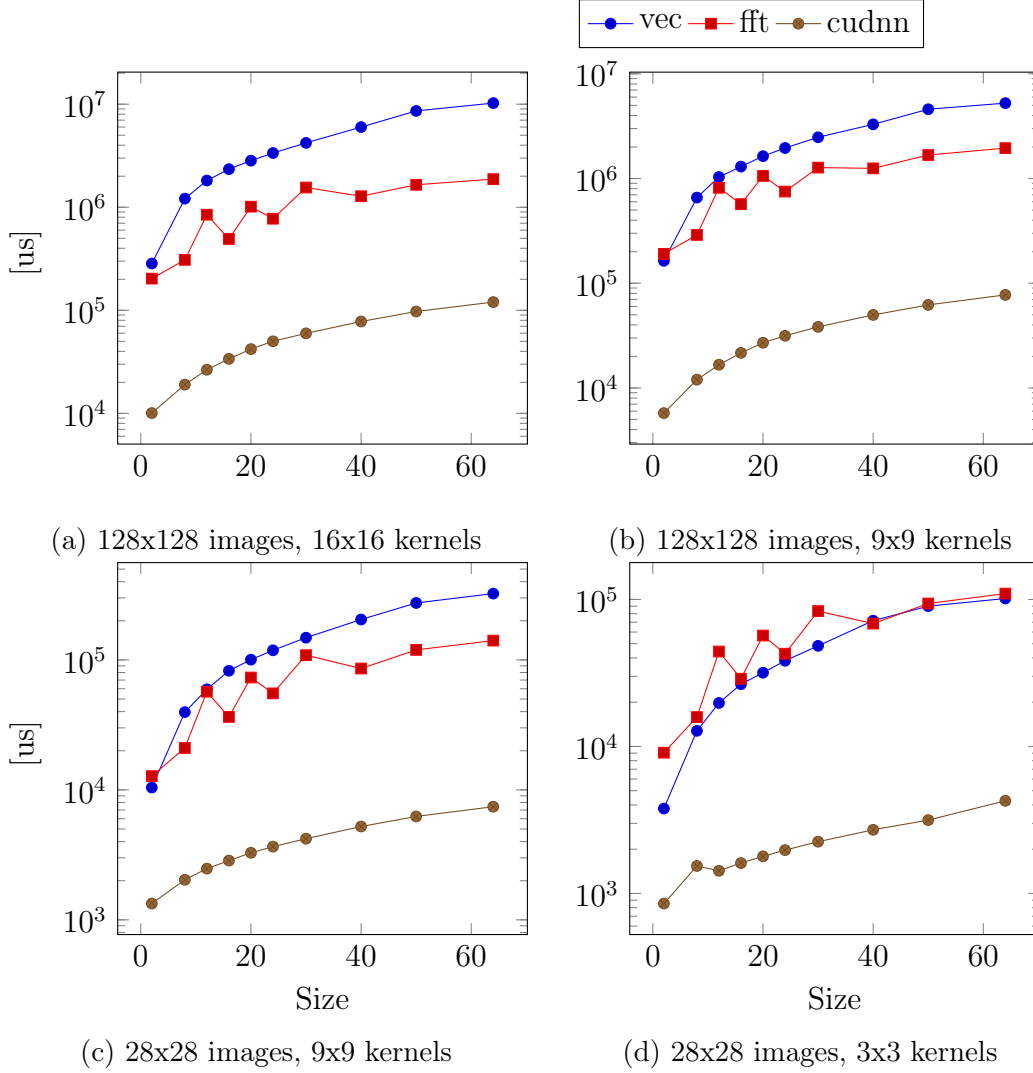


Figure A.4: Comparison of the performance of the Batch Full Convolution on CPU and GPU, with different image and kernel sizes.

(ANN) for which single-precision is enough, but may be an issue for other machine learning models for which it is not precise enough. The general trend in neural network learning with GPUs is even to work with half-precision floating point numbers for more performance. However, there is no such precision on CPU without doing the computations with higher precision and doing conversions.

The speedups that can be gained by using a GPU implementation also differ greatly from one problem to another. As seen in this section, the a matrix-matrix multiplication can be around two times faster, while the gains are not significant for an FFT. On the other hand, the speedup on a convolution operation is very significant, ranging from one to two orders of magnitude faster. This shows the high potential of GPU for CNN and Deep Learning in general.

The results presented here should not be taken as general advice on which version is the fastest. There are large differences between different types of CPU and types

of GPUs and the actual speedup will greatly vary depending on what is available. There also may be faster alternatives to the implementations that have been tested here. Moreover, depending on the type of machine learning model and the exact architecture, the needs will change. In the end, the only sensible thing to do is to benchmark according to the expected needs of the model.

A.6 References for Appendix A

- Lee, Victor W. et al. (2010). “Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU”. In: *ACM SIGARCH Computer Architecture News* 38.3, pp. 451–460 (cit. on pp. 61, 76, 185, 187, 189).
- Owens, John D. et al. (2007). “A survey of general-purpose computation on graphics hardware”. In: *Computer graphics forum*. Vol. 26. 1. Wiley Online Library, pp. 80–113 (cit. on pp. 61, 184).

Appendix B

Framework Evaluation

*Sharing is good, and with digital
technology, sharing is easy*

Richard Stallman

Contents

B.1 Introduction	193
B.2 Empirical Evaluation	195
B.2.1 MNIST	197
B.2.2 CIFAR-10	206
B.2.3 ImageNet	208
B.3 General Evaluation	210
B.3.1 Caffe	210
B.3.2 TensorFlow	211
B.3.3 Keras	212
B.3.4 DeepLearning4J	212
B.3.5 Torch	213
B.4 Conclusion	213
B.5 References for Appendix B	214

B.1 Introduction

Deep Learning Library (DLL) is the framework that was developed to support the research done during this thesis (See Chapter 4 for details). The features of the framework are highly related to the Restricted Boltzmann Machine (RBM) and

Convolutional Restricted Boltzmann Machine (CRBM) models. However, it is also able to train regular neural network models and its usage is not tightly coupled to this thesis. In order to see how this framework performs in practice and evaluate its potential for other researchers, it is compared with several other frameworks on a few experiments.

There are two main families of machine learning frameworks:

1. **Low-Level Frameworks:** In this case, the basic building block of the framework is simply the ability to describe tensors in matrix and mathematical terms. From this basic ability, higher level building blocks such as neural network layers can be described. This allows a very large range of models to be implemented with the framework. However, it generally means that the use of the framework is somewhat more complicated. It is the best choice when new models are being developed. Some of these frameworks provide general training algorithms or automatic differentiation capabilities.
2. **High-Level Frameworks:** The framework directly provides machine learning building blocks such as convolutional layers or a ReLU layer and training capabilities. These kind of frameworks are generally easier to use but allow less customization. Moreover, it is also generally harder to develop custom models without editing directly the code of the framework itself. It is the best choice when using existing models, already supported by the framework and not needing extensive customization.

DLL is in itself a high-level framework offering various machine learning modules. Since it is based on ETL which is a low-level tensor-like library, it could also be described as a low-level framework, but the features would not be enough on themselves to make a fully usable machine learning framework. Nevertheless, the fact that it is based on a low-level framework helps with the extensibility of the framework and its development.

For comparison with DLL, the following frameworks have been selected:

1. **Caffe** (Jia et al., 2014): A high-level machine learning framework started by The Berkeley Vision and Learning Center (BVLC). The focus of this framework is speed and expression. The project is developed in C++. It is optimized for both Central Processing Unit (CPU) and Graphical Processing Unit (GPU). Although it can be used directly with a low-level C++ API, it is generally best used using its special descriptor language to describe the model and its training parameters in a text file. It cannot be distributed on several computers.
2. **TensorFlow** (Abadi and al., 2015): A low-level library using data flow graphs to perform numerical computations. Although very general, this library is mainly used for machine learning. It was originally developed and used internally by Google and has since been open-sourced. It can be used on

CPU and GPU. It is a very recent library, first released in 2015. It has a very large set of features. The core of the system is written in C++, but the front end features are only available in Python. The computation graph can be distributed across several nodes. It provides automatic differentiation of the mathematical expressions.

3. Torch (Collobert, Kavukcuoglu, and Farabet, 2011): A low-level machine learning framework. It is a really mature project, started in 2002, one of the earliest machine learning framework. It is written in C, C++ and Lua and is used through a Lua front-end. It has support for both CPU and GPU. Although it is low-level, it provides several high-level modules. It cannot be directly distributed but there are some third party libraries allowing to distribute Torch computations. It does not have any support for automatic differentiation.
4. Keras (Chollet, 2015): A high-level machine learning library. It is a recent project that runs on top of either Theano or TensorFlow as a backend. It is written in Python and can only be used with Python. It provides a large number of high level modules that ease the development of machine learning software. It can also be used on several machines via TensorFlow backend, but not trivially.
5. DeepLearning4J (Team, 2015): A distributed deep learning framework. This framework is written in Java, C and C++ but can only be used with Java. It works with GPU and CPU. It has a very large set of features. It is supported commercially by Skymind. It can easily be distributed across several machines.

These frameworks have been selected for being among the most popular in the deep learning community and for being close to the DLL framework itself.

The frameworks are first compared empirically on several different simple machine learning experiments. On each of these experiments, the training time on CPU and GPU is computed as well as the final test accuracy. Moreover, the difficulty in creating and training the different models is also discussed. Then, the state of each framework is summarized in a general way. Finally, conclusions are drawn as to how the DLL framework compares to the other tested software and what should be done in order to make it better.

B.2 Empirical Evaluation

For an empirical evaluation and comparison of the different frameworks, several different experiments have been designed. Three different data sets have been selected for their popularity: MNIST, CIFAR-10 and ImageNet.

The goal of these experiments is not to reach state of the art performance, it is simply to compare accuracy and speed of different implementations of the same

model. Finally, since every framework has some different set of features, the experiments are limited to the common subset of features that all tested frameworks support. Nevertheless, some of the experiments have only been performed on some of the frameworks because of the lack of RBM or CRBM support or because of issues found with some of the frameworks.

The computer and configuration used for these tests is described in Section A.2.

The frameworks have been installed with the following details:

- DLL: Used from the sources directly with the last version available at the time of this writing (Git commit 99c49ed). In CPU mode, the experiments are compiled with support for the parallel MKL library. When GPU mode is selected, they are compiled to support the NVidia CUFFT, CUBLAS and CUDNN libraries.
- Caffe: Installed from the sources, from Git commit 5a201dd, in GPU mode. The path to the custom installation of MKL and CUDA was set. Following the documentation indicating that it could be faster or slower depending on the situation, CUDNN mode was not enabled. The switch between CPU and GPU is done through the solver mode property.
- TensorFlow: Installed directly from the official binary provided by the project, in a virtual Python environment. The version 0.12 RC0, with CUDA 8.0 was selected, with Python 3.4. The switch between CPU and GPU is done by letting CUDA know about the available device or indicating that there is no GPU available, changing a simple environment variable.
- Keras: The version 1.1.2 was installed directly from the official installer in a virtual Python environment, on Python 3.4. The TensorFlow backend was used. The switch between CPU and GPU is done as it is for TensorFlow.
- Torch: Installed directly from the sources, from Git commit 426e298. The path to the custom MKL and CUDA installation was set. A different version of each experiment was necessary to handle CUDA.
- DeepLearning4J: The version 0.7.0 was installed with Maven. The CPU or GPU backend is simply selected when the application is built by Maven, by selecting the correct Java libraries to use.

For each framework and tested case, the best result out of three runs has been used as the final result.

For reference and for an easy reproduction of the results, the source code used to perform these experiments is available online¹. This includes code for each of the model for each of the experiment as well as the different configuration scripts that are used to select the CPU and GPU mode.

¹<https://github.com/wichtounet/frameworks>

Since DLL is generally not faster on GPU mode than on CPU, the results have been taken in CPU mode for this framework.

B.2.1 MNIST

MNIST is a data set for digit recognition (LeCun, Bottou, et al., 1998). It contains 60'000 small images of digits for training and 10'000 images for testing. Each image is grayscale and is of the same 28×28 size, leading to an input size of 784. It is very well known and has been used over and over again with almost all existing machine learning methods. The current state of the art for the data set is using a large Convolutional Neural Network (CNN) trained using DropConnect (Wan et al., 2013). This technique achieved an error rate of 0.21%, which is better than the human recognition rate.

Although this data set is now considered easy, it is a good candidate for this kind of experiments since the expected performance of a neural network is well known. And there is already code available for this data set and task for most of the existing frameworks.

This task is solved using two different models: one fully-connected model and one convolutional model, in order to see the performance difference between different models for the tested frameworks.

B.2.1.1 Fully-Connected Neural Network

The first experiment that is performed is to use a simple fully-connected Artificial Neural Network (ANN) to perform digit recognition on the MNIST data set.

The network that is tested is a three-layer network with 500 hidden units in the first layer and 250 hidden units in the second layer. The last layer has 10 hidden units for classification of the digits. The first two layers are using sigmoid activation function and the network is trained with a softmax cross entropy loss.

The network is trained with Mini-Batch Gradient Descent, for 50 epochs. The learning rate is kept constant at 0.1 during training and the momentum is set to 0.9. Batches of 100 images are used to train and test the network. After each epoch, the accuracy on the training set is computed and the accuracy on the test set is computed after all epochs have completed.

Table B.1 shows the final classification error on the test set for each framework. The evolution of the training error at each epoch is presented in Figure B.1. It is interesting that there are very significant differences between the frameworks. It seems that the frameworks are handling very differently some of the parameters of the network. On average, DLL, TensorFlow and Keras were the most stable frameworks for this experiment and are always exhibiting the best final accuracy. This can also be seen in the evolution of the training error where all three frameworks are very close. Interestingly, TensorFlow shows a very unstable error during the first 20 epochs of training. DeepLearning4J also exhibits good performance.

Framework	Error (CPU)	Error (GPU)
DLL	1.97%	1.97%
Caffe	1.67%	3.48%
TensorFlow	1.82%	1.83%
Torch	7.19%	7.52%
Keras	1.86%	1.79%
DeepLearning4J	3.29%	3.335%

Table B.1: Final test error obtained by each framework on the Fully-Connected Neural Network experiment, on the MNIST data set, on CPU and GPU. The numbers in bold are from the best performing frameworks.

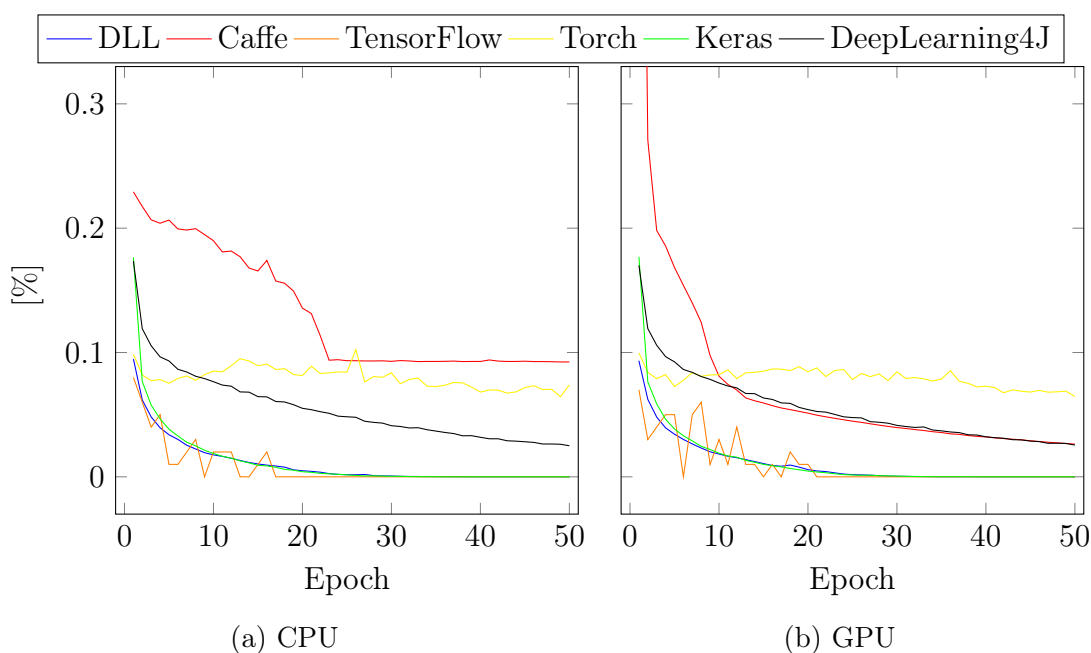


Figure B.1: Evolution of the training error for the different frameworks on the Fully-Connected Neural Network experiment, on the MNIST data set.

Overall, Torch is the framework with the worst accuracies in this experiment. It seems that some of the frameworks are handling gradients and learning rate very differently than others.

Figure B.2 shows the runtime performance of the different frameworks for this experiment. On CPU, DLL is the fastest framework, being about 35% faster than the frameworks at the second place, TensorFlow and Keras. The other three frameworks are significantly slower. Torch is around four times slower than DLL while DeepLearning4J and Caffe are around five times slower than DLL. On GPU, Caffe is achieving very impressive performance being around 14 times faster than its CPU version. It is then followed by Keras and TensorFlow. DLL manages to be faster than both Torch and DeepLearning4J even without using the GPU. It is also interesting to note that Keras and TensorFlow on CPU are faster than Torch

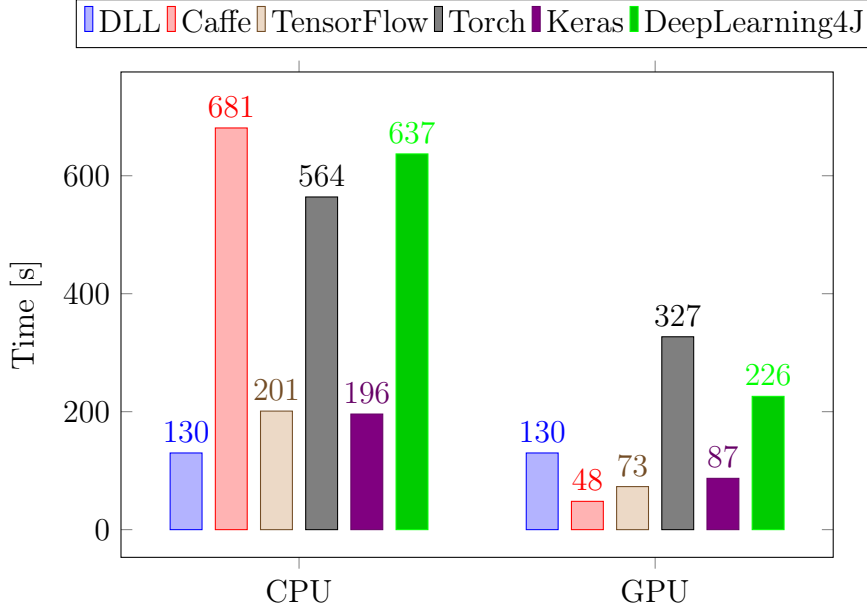


Figure B.2: Training time performance comparison of the frameworks on a Fully-Connected Neural Network experiment, on the MNIST data set, on CPU and on GPU. DLL has only been used in CPU mode.

and DeepLearning4J on GPU. Not all frameworks have the same focus when it comes to speed.

On this experiment, the DLL framework has the same training and test accuracies as the best other frameworks and is the fastest framework on CPU. Even without using GPU, it still remains competitive with the other frameworks when they take advantage of the GPU.

B.2.1.2 Convolutional Neural Network

The second experiment that is performed is the same task as the first one but using a small CNN to solve the problem.

The network is composed of the following layers. A convolutional layer with 8 kernels of size 5×5 , followed by a max pooling layer with a ratio of two in both dimensions. Then, again a convolutional layer with 8 kernels of size 5×5 , followed by another max pooling layer with the same ratios. Finally, one fully-connected layer with 150 hidden units and the final fully-connected layer with 10 units. The two convolutional layers are followed by a sigmoid activation function, as well as the first dense layers, while the final fully-connected layer uses softmax activation. The network is trained using the same parameters that were used in the first experiment.

The final test error obtained by each framework is presented in Table B.2. Figure B.3 presents the evolution of the training error at each epoch. Overall, there are

Framework	Error (CPU)	Error (GPU)
DLL	1.69%	1.69%
Caffe	1.51%	1.58%
TensorFlow	1.2%	1.4%
Torch	2.58%	2.89%
Keras	1.11%	0.96%
DeepLearning4J	1.72%	1.72%

Table B.2: Final test error obtained by each framework on the Convolutional Neural Network experiment, on the MNIST data set, on CPU and GPU. The numbers in bold are from the best performing frameworks.

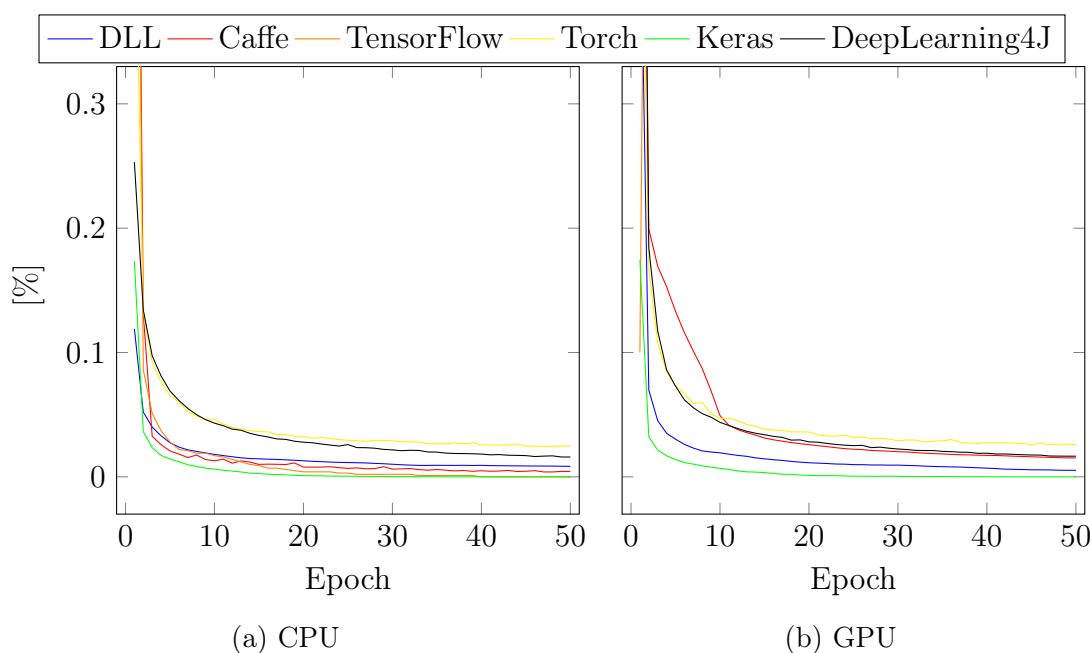


Figure B.3: Evolution of the training error for the different frameworks on the Convolutional Neural Network experiment, on the MNIST data set.

less differences between the different frameworks than with the first experiment, with Torch falling behind with its twice lower accuracy. All the other frameworks have equivalent performance. The small differences between the top frameworks may be explained by different seeds used in training. The DLL framework is quite accurate when compared with the other frameworks. It can also be seen that the frameworks are generally offering very similar performance between their CPU and GPU versions. Looking at the curves, they are also very similar from one framework to the other, with Torch and DeepLearning4J falling behind again.

Figure B.4 shows how the different libraries are performing for this experiment, in terms in training time. There are very significant differences between the different candidates. Again, DLL is the fastest framework on CPU, this time very clearly. Indeed, it is 3.6 times faster than the following frameworks, TensorFlow and Keras.

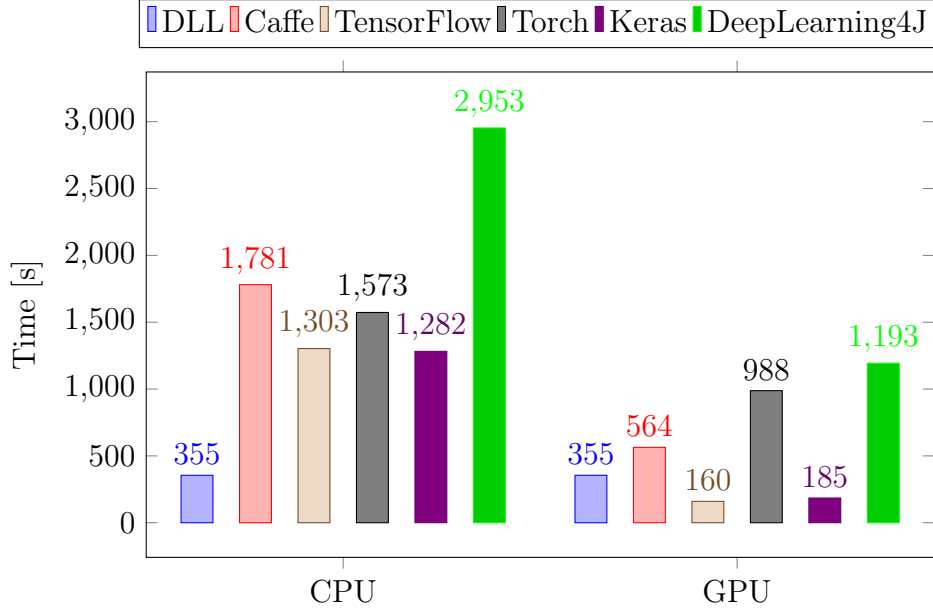


Figure B.4: Training time performance comparison of the frameworks on a Convolutional Neural Network experiment, on the MNIST data set, on CPU and on GPU. DLL has only been used in CPU mode.

The slowest framework on CPU is DeepLearning4J which is eight times slower than DLL. On GPU, the results are very different. Keras and TensorFlow are by far the fastest frameworks on GPU. Even though DLL is not accelerated by GPU, it is still significantly faster than Caffe, Torch and DeepLearning4J although they are significantly accelerated by GPU. Moreover, it is only twice slower than the two most efficient frameworks on GPU. This clearly shows that most framework are not optimized for convolution on CPU. The CPU convolution implementations of DLL were highly optimized for CPU, while most frameworks focus solely on GPU.

On this second experiment, DLL exhibits accuracy similar to the other best frameworks on both CPU and GPU. On the performance side, it is the fastest of the frameworks on CPU, with a significant speedup and has adequate performance when compared with frameworks taking advantage of GPU, being only slower than TensorFlow and Keras.

B.2.1.3 Restricted Boltzmann Machine

The next experiment is to train a RBM model on the MNIST data set and test its reconstruction capabilities. It is very difficult to estimate the quality of different RBMs especially depending on what the end goal is. Therefore, only the reconstruction error rate will be considered as a performance measure, even if it is known as a poor measure of performance. It mainly serves as a comparison of the behavior of the different frameworks.

Unfortunately, only few frameworks have support for RBM. Caffe has no support

Framework	Error (CPU)	Error (GPU)
DLL	0.864%	0.864%
TensorFlow	1.173%	1.179%
Torch	6.7850%	N/A
DeepLearning4J	25.761%	25.761%

Table B.3: Final test reconstruction error obtained by each framework on the Restricted Boltzmann Machine experiment, on CPU and GPU. The numbers in bold are from the best performing frameworks.

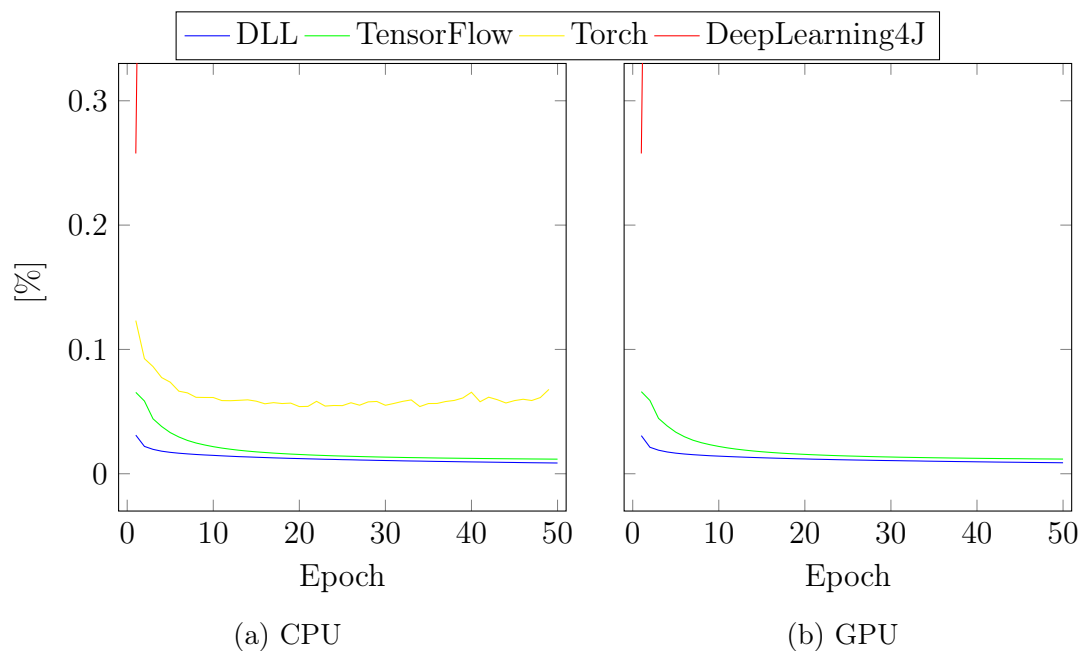


Figure B.5: Evolution of the reconstruction error for the different frameworks on the Restricted Boltzmann Machine experiment, on the MNIST data set.. The curve for DeepLearning4J is going up and staying very high after this point.

at all for RBM and Deep Belief Network (DBN). In the tested frameworks other than DLL, only DeepLearning4J has integrated RBM support. Torch has unofficial support for RBM, which is what was used here. Unfortunately, it was not possible to adapt this for GPU, therefore only CPU results are presented for Torch. TensorFlow does not have integrated support for RBM, but since it is a general purpose computation library, it is relatively simple to build a basic RBM. Keras also has some unofficial support, but it is highly out of date with the official version and it was unfortunately not possible to make it work.

The model that is tested is a single RBM with 784 visible units (the pixels of a MNIST image) and 500 hidden units. All the units are binary sigmoid units. The model is trained using Contrastive Divergence (CD) with Mini-Batch, for 50 epochs. The learning rate is kept at 0.1 during the entire training and the momentum is set to 0.9. Mini-batches of 100 images are used for training.

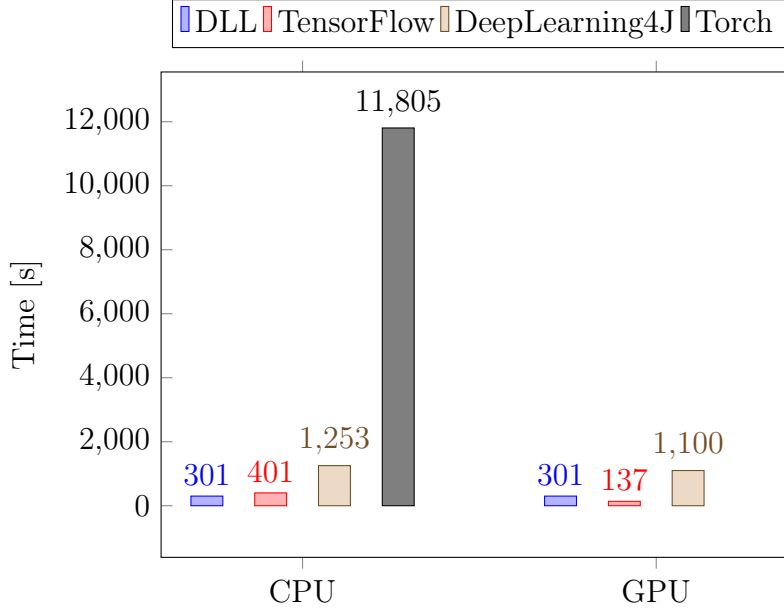


Figure B.6: Training time performance comparison of the frameworks on a Restricted Boltzmann Machine training experiment, on the MNIST data set, on CPU and on GPU. DLL was only used in CPU mode.

The final reconstruction error of each framework is shown in Table B.3. Figure B.5 presents the evolution of the training reconstruction error at each epoch. There are very large discrepancies between the different implementations of RBM. Unfortunately, it was impossible to make DeepLearning4J learn a good model. Indeed, whatever the training parameters the model was given, it was not learning and the best reconstruction error is in fact the reconstruction error from the model before training. No answer to this problem was given by the maintainers of the library. Even if Torch is not on the same performance level as DLL and TensorFlow, it is still learning reasonably well and provides an acceptable reconstruction error once trained for long enough. Both DLL and TensorFlow are providing a good final reconstruction error. As seen in the learning curves, they are learning approximately at the same rate, with DLL being faster in the early epochs of training, which may simply come from a better initialization.

Figure B.6 presents the time necessary to train the RBM using the different frameworks. The very long time necessary to Torch for training the RBM is partially explained by the fact that the training is done in a single-threaded manner. However, the code is made in such a way that adding more threads is even slower. This is not directly related to Torch itself but rather to the unofficial Torch RBM library that was used for this experiment which seems to be very poorly optimized. As for the other frameworks, DLL is about 25% faster than TensorFlow but both libraries are exhibiting good training times. DeepLearning4J is reasonably fast but already four times slower than DLL. When using GPU, TensorFlow is clearly the fastest, with DLL behind. The very small acceleration of DeepLearning4J by GPU can be

Framework	Error (CPU)	Error (GPU)
DLL	0.965%	0.965%
TensorFlow	1.08%	1.074%

Table B.4: Final test reconstruction error obtained by each framework on the Convolutional Restricted Boltzmann Machine experiment, on CPU and GPU. The numbers in bold are from the best performing frameworks.

partially explained by the fact that reconstruction of samples does not seem to be accelerated while the training itself is accelerated and thus a significant amount of time is spent in computing the reconstruction error.

Overall, DLL is the fastest at training a RBM on CPU and remains competitive even when other frameworks are taking advantage of GPU. The reconstruction performance of the models trained by the framework is comparable to that of the other frameworks. Moreover, it is also the framework that has the most features for RBM and that is the most simple to build around.

B.2.1.4 Convolutional RBM

The final experiment is similar to the previous one, but using a CRBM instead of an RBM. It is also evaluated using the reconstruction error as was the previous experiment.

When it comes to CRBM, the choice of frameworks supporting it is even more scarce than it was for RBM. From the frameworks having support for RBM, it was only possible to use TensorFlow for a CRBM implementation. For this, the previous implementation was adapted to a CRBM. DeepLearning4J has no support for CRBM and Torch has not even any unofficial support for it.

The model that is trained is a single layer CRBM with 8 convolutional filters of size 5×5 . It is trained with CD with Mini-Batch, for 50 epochs. The learning rate is set to 0.001 during the entire training and the momentum is set to 0.9. Batches of 100 images are used for training.

The final reconstruction error of each framework is shown in Table B.4. Figure B.7 presents the evolution of the training reconstruction error at each epoch. There are not very significant differences between the frameworks. DLL is a bit faster to learn reconstruction and ends up with a better reconstruction rate but both implementations are achieving a very good reconstruction rate.

The time necessary to train the model with both frameworks is presented on Figure B.8 for GPU and CPU. DLL achieves an excellent performance on CPU, being almost three times faster than TensorFlow. Once it is given access to a GPU, TensorFlow becomes much faster, making TensorFlow about 2.5 times faster than DLL.

The DLL framework is exhibiting similar reconstruction performance for CRBM than TensorFlow, but has real integrated support for CRBM. It is especially fast

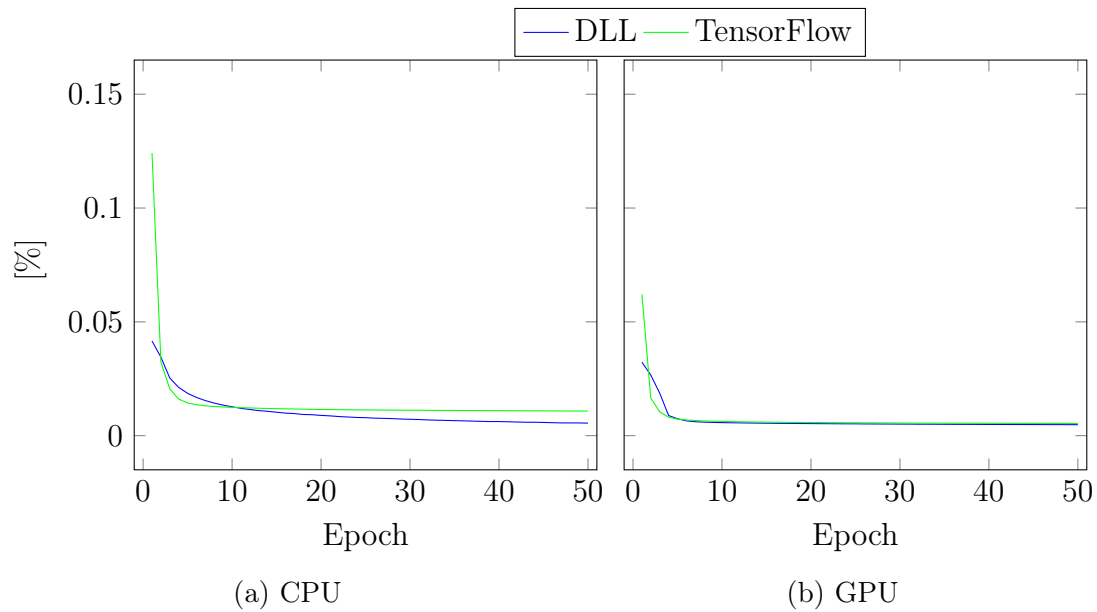


Figure B.7: Evolution of the reconstruction error for the different frameworks on the Convolutional Restricted Boltzmann Machine experiment, on the MNIST data set.

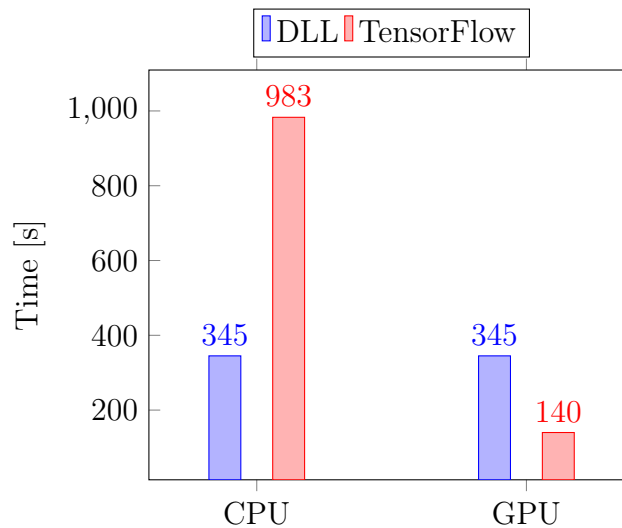


Figure B.8: Training time performance comparison of the frameworks on a Convolutional Restricted Boltzmann Machine training experiment, on the MNIST data set, on CPU and on GPU. DLL was only used in CPU mode.

Framework	Error (CPU)	Error (GPU)
DLL	40.31%	40.31%
Caffe	90%	36.17%
TensorFlow	38.4%	36.9%
Torch	41.56%	37.84%
Keras	37.13%	35.09%

Table B.5: Final test error obtained by each framework on the Convolutional Neural Network experiment, on the CIFAR-10 data set, on CPU and GPU. The numbers in bold are from the best performing frameworks.

on CPU but falls behind TensorFlow with a GPU.

B.2.2 CIFAR-10

Although MNIST is a good benchmark data set, it is very small and its images are also very small and grayscale, making it not really representative of the images handled by recent networks. Therefore, CIFAR-10 was used as the second data set for experiments. CIFAR-10 is a data set for object recognition (Krizhevsky and G. Hinton, 2009). It contains 60'000 images of ten different classes. Each image is a 32×32 color image, with three color channels, for a total of 3072 inputs. 50'000 images are used for training and 10'000 are used for testing. Although very similar to MNIST, it is significantly more complicated to obtain good results and the state of the art accuracy is significantly lower. At this time, the best accuracy achieved on this data set is 96.5% (Graham, 2014), achieved with a special max pooling operator.

Once again, we use a convolutional network for this experiment. The network is similar to the one used for the previous MNIST experiment, but with significantly more convolutional filters. The first convolutional layer has 12 filters of size 5×5 and is followed by a 2×2 max pooling layer. The third layer is again convolutional with 24 filters of size 3×3 and again pooled with 2×2 max pooling. These layers are followed by a dense layer of 64 units and a final softmax layer with ten output units, representing the final classes. All the hidden units are using Rectified Linear Units (ReLUs). The network is trained similarly as the previous networks, but with a smaller learning rate (0.001).

Unfortunately, we did not succeed in using DeepLearning4J for this experiment. Indeed, the training always stopped with an error, apparently related to the official data loader for this data set. No answers or solution was provided by the maintainers of the library at the time of this writing. Thus, this framework has been removed from this experiment.

The final classification error of each framework is shown in Table B.5. The large error rates for this experiment are not surprising given the small size of the network being trained and the short training. Except for Caffe in CPU mode, every

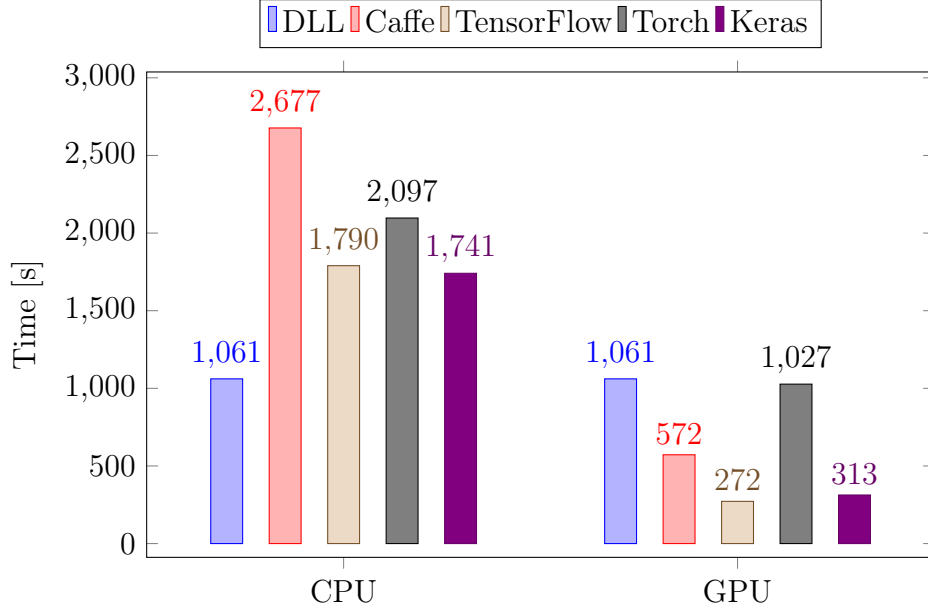


Figure B.9: Training time performance comparison of the frameworks on a Convolutional Neural Network experiment, on the CIFAR-10 data set, on CPU and on GPU. DLL has only been used in CPU mode.

framework achieves similar performance on this task. For an unknown reason, Caffe training was very unstable and was generating Not-a-Number (NaN) results very quickly during training.

Figure B.9 presents the training times for this task, for each framework. Although the speedups are less significant than for the previous CNN experiment, DLL is still the fastest framework when using only a CPU. Indeed, it is about 40% faster than TensorFlow and Keras and even twice faster than Torch and 2.6 times faster than Caffe. On GPU, DLL has the same speed as Torch, at the last position, but is significantly slower than the leading frameworks. Indeed, TensorFlow and Keras are about four times faster than DLL while Caffe is almost twice faster. The difference in the speedups can be explained by the significantly larger number of filters in the convolutional layers and the larger input images. This may indicate that most frameworks have been more optimized for larger networks. This also confirms that GPU computations are achieving the best performance when a large amount of data is available for each computation.

Overall, DLL is able to train a CNN achieving the same accuracy as the other frameworks for the CIFAR-10 object recognition task and is able to train it faster than them when using a CPU. However, DLL is among the slowest frameworks when a GPU is available.

B.2.3 ImageNet

The previous two data sets are containing relatively small images and contains relatively few images. Therefore, the last experiment is done on ImageNet. ImageNet is a very large database of natural color images, organized in many categories. In total, there are more than 14 millions of images and more than 20'000 categories. In this experiment, we are considering the sub part defined for the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) 2012 (Russakovsky et al., 2015). This sub data set contains 50'000 validation images, 100'000 test images and around 1'200'000 training images. All the images are sorted into 1000 categories. It still considered as a challenging data set but many recent breakthroughs have been achieved on this task. At the time of this writing, the best classification error is 3.08%, achieved with the Inception network (Szegedy, Ioffe, et al., 2016)

Prior to training, all the images have been resized to a size of 256×256 , for a total input size of 196'608 pixels (250 times more than for MNIST). No data augmentation is performed for training, the images are taken directly, contrary to most state of the art results that are using random cropping. Due to the high dimensionality of the images and the very large number of images available in the data set, it is impossible, in our environment, to keep the data set in memory. This makes training more difficult and more time-consuming since images are loaded several times from the file system.

While every reference frameworks had official training for the previous MNIST and CIFAR-10 CNN experiments, only Caffe provides an official and up-to-date code for training with ILSVRC 2012. The code for DeepLearning4J has been based on an official reader for data sets similar to ImageNet. For the other frameworks, simple data reading has been implemented using the tools available in each of them.

In order to solve this task, a large network is used. It starts with five convolutional layers, using a 3×3 filter and one pixel of zero-padding around the image, thus the output size is the same as the input size. Each of these five layers is followed by ReLU activation and max pooling with a 2×2 kernel. They are followed by one dense layer with 2048 ReLU units and a fully-connected softmax layer with 1000 classification units corresponding to the classes.

To keep the time of the experiment reasonable, the network is only trained for five epochs. After the training, the accuracy over the training set is computed. The networks are trained using mini-batches of 128 images. However, since it was not possible to use this parameter for DeepLearning4J and Torch, they are using mini-batches of 64 images. Indeed, both required more than 12GB of memory with 128 images in a mini-batch.

The final classification training accuracy of each framework is shown in Table B.6. Unfortunately, DeepLearning4J was too slow to complete the five epochs in our environment, therefore the final results were not computed. Overall, each framework is exhibiting comparable accuracy on this data set even with very few training

Framework	Error (CPU)	Error (GPU)
DLL	30.09%	30.09%
Caffe	30.28%	33.03%
TensorFlow	32.12%	32.88%
Torch	25.28%	27.39%
Keras	31.73%	30.23%

Table B.6: Final training error obtained by each framework on the Convolutional Neural Network experiment, on the ImageNet data set (ILSVRC 2012), on CPU and GPU. The numbers in bold are from the best performing frameworks.

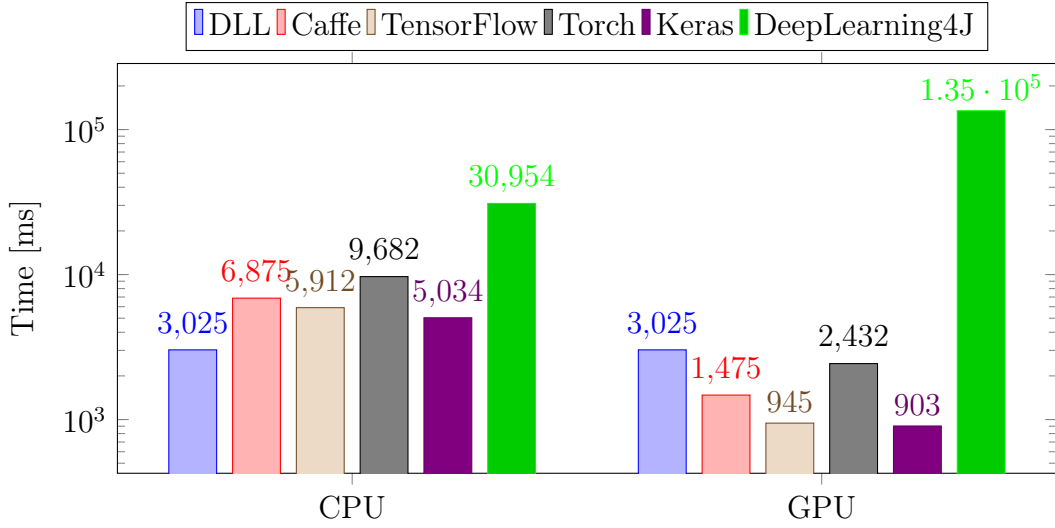


Figure B.10: Training time performance comparison of the frameworks on the ImageNet task, on CPU and on GPU. The time scale is logarithmic. The time is the average time necessary for the training of one batch of 128 elements. DLL has only been used in CPU mode.

epochs.

In order to make comparison more straightforward, the average training time for one batch is used rather than the full training time. For the frameworks using batches of 64 images (Torch and DeepLearning4J), the result is the training time for two batches. Figure B.10 presents these results. Since DeepLearning4J is much slower than the other frameworks, the results are presented in logarithmic form. The definite reason for this inefficiency was not found, but several similar reports were found online indicating very large difference in performance between GPU card models. Once again, DLL manages to be the fastest framework on CPU, even with this very large model. Indeed, it is about 40% faster than Keras and two times faster than TensorFlow and Caffe and more than three times faster than Torch. All the frameworks are much faster than DeepLearning4J in both training modes. When using the GPU, DLL is slightly slower than Torch, but already two times slower than Caffe and about three times slower than Keras and TensorFlow.

On this last experiment, DLL manages to train a very large network with similar accuracies as the other reference frameworks. Moreover, it is able to train it significantly faster than they are when only the CPU is available. Once GPU is used, it is among the slowest frameworks.

B.3 General Evaluation

This section summarizes the state of the different frameworks as to give an overview of what they are good for and what are their lacks. It is important to note that this evaluation only results from the previous experiments and as such is not an extensive evaluation and may be highly suggestive.

B.3.1 Caffe

Caffe has a large set of features, especially when focused on neural networks. The framework is generally up to date with recent technologies. However, its fundamental architecture makes it a poor candidate to implement Recurrent Neural Networks (RNNs) and makes it highly inflexible in general. It is not directly extensible without working inside Caffe code. Moreover, Caffe has no support for RBM or CRBM.

For some, one advantage of Caffe is that no programming code is necessary to create and train machine learning models. Indeed, everything can be done by describing the model and describing the solver to train it. However, protobuf, the language chosen for the description of the network is not the simplest one to work with and the structure of the description is not always easy to create or even to understand. Moreover, it is also difficult to change how it is working since the only access is through the descriptor, so it cannot be integrated directly in another application. The switch from CPU to GPU is very simple, a simple property in the solver descriptor and the training will be done in the selected mode. A problem in Caffe is that, contrary to all the other frameworks, it has absolutely no notion of epochs, only of iterations. This makes porting some examples to Caffe non-intuitive and makes comparing results more difficult than it should be.

The installation is not necessarily trivial and there are several large dependencies that are necessary such as Boost, protobuf, gflags and several other libraries. Moreover, the installation with GPU support is not evident since it has very strong dependencies on the version of CUDA and of CUDNN it is supporting.

The CPU mode of Caffe was the most unstable of all the tested frameworks. It was very difficult to make it work on some simple examples, even on the examples from the official distribution. Several errors such as double free corruption or experiments ending with Not-a-Number (NaN) results were encountered. This can be observed on the first experiment where the Caffe version trained with CPU produces the worst results, by far. On the other hand, it is generally much better

when training the model in GPU mode.

As for the performance, it has average performance on CPU and good performance on GPU but it is only the fastest network on the small fully-connected experiment.

Due to its small customization abilities, it is best used when using existing models, already written in Caffe descriptor language and when planning to only use GPU.

B.3.2 TensorFlow

TensorFlow is a recent, very modern, low-level framework that is evolving really quickly. At its base, it is not limited to machine learning. Indeed, it is mainly a framework for optimized numerical computation using data-flow graphs with automatic differentiation.

Due to its low-level nature, it has very few features related to machine learning. Indeed, it does not even have any support for Multi-Layer Perceptron (MLP). While it has a few high level structures, such as Long Short Term Memory (LSTM) cells, it provides mainly tools such as softmax or ReLU functions that needs to be composed to form machine learning models. However, it does have a very good support for various optimization algorithms, such as Stochastic Gradient Descent (SGD) and Limited-Memory Broyden-Fletcher-Goldfarb-Shanno (L-BFGS), and its automatic differentiation feature makes it easy to develop simple models without having to work out the gradients of the model. This makes it very easy to experiment with new variations of models or to extend existing code. On the other hand, it is rather complicated to create simple models such as CNN compared to some other frameworks.

The installation of TensorFlow can be relatively easy, but it highly depends on the platform on which installation is to be made. Indeed, installers are provided for several popular platforms which makes it very simple to install. For other platforms, installation from the sources is necessary and is far from trivial and requires many non-obvious dependencies.

Its runtime performance is really good. It is generally the fastest framework on GPU, sometimes being several times faster than the second (not including Keras, based on TensorFlow). For some CNNs on CPU, it could be faster, being two times slower than DLL.

It is very easy to switch between CPU and GPU, a simple environment variable in the system allows to change it without modification of the source code or without recompilation.

Overall, it is an excellent candidate for experimentation with new models or variations of existing models. Due to its growing popularity, there is a large number of examples available, making it easy to work with existing systems.

B.3.3 Keras

Keras is a high-level framework based on TensorFlow. It can also be used as a frontend to Theano, but this was not tested in these experiments.

While TensorFlow is very low-level and often makes it necessary to write mathematical expressions to train a network, Keras adds support for higher level structures such as dense and convolutional layers directly. It also has a real notion of network. It has a large panel of features. Unfortunately, it does not have any official support for RBM and the only unofficial support for it is highly outdated and is not able to work with the recent versions of Keras.

Since it is based on TensorFlow, it is also extremely fast. Indeed, on GPU it is always on the top frameworks. On CPU, except for convolutional networks where it is dominated by DLL, it is among the fastest frameworks. Moreover, direct access to the tensor backend is also possible to make changes or extend the framework with more manipulation. The switch from CPU to GPU is the same than TensorFlow, with a simple environment variable to change.

The installation is fairly simple, but requires either Theano or TensorFlow, which are not always trivial to install.

Overall, it is an excellent high level framework that can also be customized, with very good performance, both on CPU and GPU.

B.3.4 DeepLearning4J

DeepLearning4J is a high-level framework with a large set of features. It has integrated support for distributed computing.

It has integrated support for many neural networks models. However, the way networks are built in the code is not especially intuitive and doing other tasks than just classification with the models is not practical. Customization or extension of existing models is not easy due to the framework complex architecture. There is official support for RBM, but it is very limited and very unstable. Indeed, it was not possible during the experiment to successfully train a simple RBM for reconstruction.

The installation is relatively easy if it is done with Maven. Otherwise, it requires a very large number of Java dependencies. The switch from CPU to GPU is simple but requires a recompilation of the program.

This framework is one of the slowest that was tested in this comparison. On fully-connected networks, it was next to last in front of Torch but was last when convolutional layers were added.

Overall, it is a good framework with many features but low performance. It is probably a good fit if Java is already used in other research projects or if distributed computing is necessary.

B.3.5 Torch

Torch is a mature low-level framework. At its core, it is also a tensor library like TensorFlow, but lacks an automatic differentiation support.

Although models can be written directly using tensors, Torch also offers high level structures such as neural network layers. It is easy to create new models with Torch. This makes it easier to work with than TensorFlow for some models. However, training of the models is less intuitive than with TensorFlow. In practice, it was always one of the frameworks with the worst accuracy on the tested models. There is unofficial support for RBM, but it is extremely slow, being 27 times slower than DLL. There is no support for CRBM.

However, although higher level structures are available in Torch, it is still more verbose to write code than with other high level framework with sometimes a lot of code required to train a model. Moreover, it was also the only framework that required changing several lines of code for switching from CPU to GPU.

The installation of the framework is highly unclear. It is not obvious how to configure the BLAS library that is to be used by the framework. Moreover, the installation process also tries to install, by itself, several system packages.

As for the performance of training, it is one of the slowest frameworks, with DeepLearning4J. It is always slower than DLL, even when using a GPU.

Overall, Torch proposes almost the same features than TensorFlow but is significantly slower, less practical to use and much less active at the time of this writing. On the other hand, it does have the advantage of having several higher level structures available.

B.4 Conclusion

This chapter compared several machine learning frameworks on six different experiments using three different data sets. The goal was to compare DLL with popular frameworks and see what are its strengths and weaknesses.

Overall, DLL proved on par with the other frameworks in terms of classification performance and reconstruction capabilities. This shows that the framework is working correctly, at least on the tested cases. However, for neural networks, it is still lacking a few options. For instance, it does not have support for advanced techniques such as Batch Normalization.

As for usability, for the tested experiments, DLL was among the best frameworks. It required only very few lines of code. However, it has less options and lacks in customization abilities and extension points. The switch from CPU to GPU is also seamless, but requires a recompilation of the program to take effect. It is the framework that has the best RBM and CRBM support, being designed for this, at its core.

In terms of performance, DLL is always the fastest framework on CPU in all our experiments. However, it does not take enough advantage of the GPU. When using the GPU, it is still competitive for small networks but is significantly slower for larger networks than the other reference frameworks.

In view of this evaluation, several things will be necessary to improve DLL and make it an appealing framework. First, it is necessary to take full advantage of GPU. Currently, only very few operations are performed on GPU and the data is moved from CPU to GPU too many times during training. Even if its CPU performance is already very good, there is still room for improvement to make it the fastest framework on CPU. Moreover, in order for DLL to be used as a general machine learning framework, it would be necessary to add more advanced features like Batch Normalization. Support for RNNs, and more specifically LSTMs, would also be a plus for the framework. Finalization of its layer system so that new layers can be added easily should also be done. Another small improvement would be to be able to change at runtime the CPU or GPU mode instead of requiring a recompilation. Finally, improvement of the descriptor system that allows to create models without writing C++ code would also improve the usability of the framework by a large factor.

This evaluation of frameworks does not aim at being extensive and has only been performed for evaluating DLL, the framework that has been developed during this thesis. For another point of view, Bahrampour et al. performed an extensive comparison of five frameworks (Caffe, Neon, TensorFlow, Theano, and Torch) (Bahrampour et al., 2015). A comparison of the main features of a large number of Deep Learning frameworks is also available online (Wikipedia, 2016).

B.5 References for Appendix B

- Abadi, Martin and al. (2015). *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. URL: <http://tensorflow.org/> (cit. on p. 194).
- Bahrampour, Soheil et al. (2015). “Comparative Study of Deep Learning Software Frameworks”. In: *arXiv preprint arXiv:1511.06435* (cit. on p. 214).
- Chollet, François (2015). *keras*. <https://github.com/fchollet/keras> (cit. on p. 195).
- Collobert, Ronan, Koray Kavukcuoglu, and Clément Farabet (2011). “Torch7: A Matlab-like Environment for Machine Learning”. In: *BigLearn, NIPS Workshop* (cit. on p. 195).
- Graham, Benjamin (2014). “Fractional max-pooling”. In: *arXiv preprint arXiv:1412.6071* (cit. on p. 206).
- Jia, Yangqing et al. (2014). “Caffe: Convolutional Architecture for Fast Feature Embedding”. In: *arXiv preprint arXiv:1408.5093* (cit. on pp. 62, 79, 194).
- Krizhevsky, Alex and Geoffrey Hinton (2009). “Learning multiple layers of features from tiny images”. In: (cit. on p. 206).

- LeCun, Yann, Léon Bottou, et al. (1998). “Gradient-based learning applied to document recognition”. In: *Proceedings of the IEEE* 86.11, pp. 2278–2324. ISSN: 0018-9219. DOI: 10.1109/5.726791 (cit. on pp. 18, 87, 99, 197).
- Russakovsky, Olga et al. (2015). “ImageNet Large Scale Visual Recognition Challenge”. In: *International Journal of Computer Vision (IJCV)* 115.3, pp. 211–252. DOI: 10.1007/s11263-015-0816-y (cit. on p. 208).
- Szegedy, Christian, Sergey Ioffe, et al. (2016). “Inception-v4, inception-resnet and the impact of residual connections on learning”. In: *arXiv preprint arXiv:1602.07261* (cit. on p. 208).
- Team, Deeplearning4j Development (2015). *Deeplearning4j: Open-source distributed deep learning for the JVM*. URL: <http://deeplearning4j.org> (cit. on p. 195).
- Wan, Li et al. (2013). “Regularization of neural networks using dropconnect”. In: *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, pp. 1058–1066 (cit. on pp. 61, 197).
- Wikipedia (2016). *Comparison of Deep Learning software*. Online; accessed 05-December-2016. URL: https://en.wikipedia.org/wiki/Comparison_of_deep_learning_software (cit. on p. 214).

References

- Abadi, Martin and al. (2015). *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. URL: <http://tensorflow.org/> (cit. on p. 194).
- Ackley, David H, Geoffrey E. Hinton, and Terrence J. Sejnowski (1985). “A learning algorithm for boltzmann machines”. In: *Cognitive science* 9.1, pp. 147–169 (cit. on p. 20).
- Adamek, Tomasz, Noel E. O’Connor, and Alan F. Smeaton (2007). “Word matching using single closed contours for indexing handwritten historical documents”. In: *Int. Journal of Document Analysis and Recognition* 9, pp. 153–165 (cit. on p. 114).
- Almazán, Jon et al. (2014). “Word spotting and recognition with embedded attributes”. In: *IEEE transactions on pattern analysis and machine intelligence* 36.12, pp. 2552–2566 (cit. on p. 115).
- Anagnostaras, Patrick (2008). *Sudoclick - automatic recognition of Sudoku grids for mobile phone* (cit. on p. 86).
- Bahrampour, Soheil et al. (2015). “Comparative Study of Deep Learning Software Frameworks”. In: *arXiv preprint arXiv:1511.06435* (cit. on p. 214).
- Baldi, Pierre and Peter J. Sadowski (2013). “Understanding Dropout”. In: *Advances in Neural Information Processing Systems 26*. Ed. by C. J. C. Burges et al. Curran Associates, Inc., pp. 2814–2822. URL: <http://papers.nips.cc/paper/4878-understanding-dropout.pdf> (cit. on p. 61).
- Barrett, Christian, Richard Hughey, and Kevin Karplus (1997). “Scoring hidden Markov models”. In: *Computer applications in the biosciences: CABIOS* 13.2, pp. 191–199 (cit. on p. 124).
- Bengio, Yoshua (2009). “Learning deep architectures for AI”. In: *Foundations and trends® in Machine Learning* 2.1, pp. 1–127 (cit. on pp. 58, 112, 149, 150).
- Bengio, Yoshua, Aaron Courville, and Pascal Vincent (2013). “Representation learning: A review and new perspectives”. In: *IEEE transactions on pattern analysis and machine intelligence* 35.8, pp. 1798–1828 (cit. on p. 112).
- Bengio, Yoshua, Pascal Lamblin, et al. (2007). “Greedy layer-wise training of deep networks”. In: *Advances in neural information processing systems* 19, p. 153 (cit. on p. 59).
- Bengio, Yoshua, Yann LeCun, and al. (2007). “Scaling learning algorithms towards AI”. In: *Large-scale kernel machines* 34.5 (cit. on p. 150).
- Bertsekas, Dimitri P. (1999). *Nonlinear programming* (cit. on p. 29).

- Bhardwaj, Anurag, Damien Jose, and Venu Govindaraju (2008). “Script Independent Word Spotting in Multilingual Documents.” In: *IJCNLP*, pp. 48–54 (cit. on p. 114).
- Bishop, Christopher M. (2006). *Pattern recognition and Machine Learning*. Springer (cit. on p. 12).
- Blue, James L. and Patrick J. Grother (1992). “Training feed-forward neural networks using conjugate gradients”. In: *SPIE/IS&T 1992 Symposium on Electronic Imaging: Science and Technology*. International Society for Optics and Photonics, pp. 179–190 (cit. on p. 19).
- Boureau, Y-lan, Yann LeCun, and Marc’Aurelio Ranzato (2008). “Sparse feature learning for deep belief networks”. In: *Advances in neural information processing systems*, pp. 1185–1192 (cit. on p. 151).
- Boureau, Y-Lan, Jean Ponce, and Yann LeCun (2010). “A theoretical analysis of feature pooling in visual recognition”. In: *Proceedings of the Int. Conf. on Machine Learning*, pp. 111–118 (cit. on p. 47).
- Bracewell, Ron (1965). “The Fourier transform and its applications”. In: *New York* 5 (cit. on p. 75).
- Bradski, Gary (2000). “The OpenCV library”. In: *Dr. Dobb’s Journal of Software Tools* (cit. on pp. 72, 91).
- Byrd, Richard H. et al. (1995). “A limited memory algorithm for bound constrained optimization”. In: *SIAM Journal on Scientific Computing* 16.5, pp. 1190–1208 (cit. on p. 19).
- Canny, John (1986). “A Computational Approach to Edge Detection”. In: *IEEE Transactions Pattern Analysis Mach. Intelligence* 8.6, pp. 679–698. ISSN: 0162-8828. DOI: 10.1109/TPAMI.1986.4767851. URL: <http://dx.doi.org/10.1109/TPAMI.1986.4767851> (cit. on p. 90).
- Cao, Huaigu and Venu Govindaraju (2007). “Template-free word spotting in low-quality manuscripts”. In: *Proceedings of the 6th International Conference on Advances in Pattern Recognition*, pp. 135–139 (cit. on p. 115).
- Chan, Jim, Celal Ziftci, and David Forsyth (2006). “Searching off-line Arabic documents”. In: *Proceedings of the IEEE Conf. on Computer Vision and Pattern Recognition*. Vol. 2. IEEE, pp. 1455–1462 (cit. on pp. 115, 123).
- Chang, Chih-Chung and Chih-Jen Lin (2011). “LIBSVM: A library for support vector machines”. In: *ACM Transactions on Intelligent Systems and Technology* 2 (3). Software available at <http://www.csie.ntu.edu.tw/~cjlin/libsvm>, 27:1–27:27 (cit. on p. 70).
- Chen, Francine R., Lynn U. Wilcox, and Dun S. Bloomberg (1993). “Word spotting in scanned images using Hidden Markov Models”. In: *Proceedings of the IEEE Int. Conf. on Acoustics Speech and Signal Processing*. Vol. 5. IEEE, pp. 1–4 (cit. on p. 113).
- Cho, Kyunghyun (2013). “Boltzmann machines and denoising autoencoders for image denoising”. In: *arXiv preprint arXiv:1301.3468* (cit. on pp. 143, 162, 167).
- Choisy, Christophe (2007). “Dynamic handwritten keyword spotting based on the NSHP-HMM”. In: *Proceedings of the IEEE Int. Conf. on Document Analysis and Recognition*. Vol. 1. IEEE, pp. 242–246 (cit. on pp. 115, 123).

- Chollet, François (2015). *keras*. <https://github.com/fchollet/keras> (cit. on p. 195).
- Christian, Wolf, Jean-Michel Jolion, and Françoise Chassaing (2002). “Text Localization, Enhancement and Binarization in Multimedia Documents”. In: *Proceedings of the International Conference on Pattern Recognition*. Vol. 2, pp. 1037–1040 (cit. on p. 137).
- Ciregan, Dan, Ueli Meier, and Jürgen Schmidhuber (2012). “Multi-column deep neural networks for image classification”. In: *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*. IEEE, pp. 3642–3649 (cit. on p. 58).
- Coates, Adam, Blake Carpenter, et al. (2011). “Text detection and character recognition in scene images with unsupervised feature learning”. In: *2011 International Conference on Document Analysis and Recognition*. IEEE, pp. 440–445 (cit. on p. 90).
- Coates, Adam, Honglak Lee, and Andrew Y. Ng (2010). “An analysis of single-layer networks in unsupervised feature learning”. In: *Ann Arbor* 1001.48109, p. 2 (cit. on pp. 112, 167).
- Collobert, Ronan, Koray Kavukcuoglu, and Clément Farabet (2011). “Torch7: A Matlab-like Environment for Machine Learning”. In: *BigLearn, NIPS Workshop* (cit. on p. 195).
- Courbariaux, Matthieu, Yoshua Bengio, and Jean-Pierre David (2014). “Training deep neural networks with low precision multiplications”. In: *arXiv preprint arXiv:1412.7024* (cit. on p. 78).
- Courville, Aaron C., James Bergstra, and Yoshua Bengio (2011). “A spike and slab restricted Boltzmann machine”. In: *International Conference on Artificial Intelligence and Statistics*, pp. 233–241 (cit. on p. 50).
- Dahl, George E., Tara N. Sainath, and Geoffrey E. Hinton (2013). “Improving deep neural networks for LVCSR using rectified linear units and dropout”. In: *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*. IEEE, pp. 8609–8613 (cit. on p. 39).
- Dahl, George et al. (2010). “Phone recognition with the mean-covariance restricted Boltzmann machine”. In: *Advances in neural information processing systems*, pp. 469–477 (cit. on p. 49).
- Dalal, Navneet and Bill Triggs (2005). “Histograms of oriented gradients for human detection”. In: *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR’05)*. Vol. 1. IEEE, pp. 886–893 (cit. on p. 133).
- David, Omid E. and Nathan S. Netanyahu (2016). “DeepPainter: Painter Classification Using Deep Convolutional Autoencoders”. In: *International Conference on Artificial Neural Networks*. Springer, pp. 20–28 (cit. on p. 63).
- Doi, Eizaburo, Doru C. Balcan, and Michael S. Lewicki (2005). “A theoretical analysis of robust coding over noisy overcomplete channels”. In: *Advances in neural information processing systems*, pp. 307–314 (cit. on p. 151).
- Duchi, John, Elad Hazan, and Yoram Singer (2011). “Adaptive subgradient methods for online learning and stochastic optimization”. In: *Journal of Machine Learning Research* 12.Jul, pp. 2121–2159 (cit. on p. 71).

- Erhan, Dumitru, Yoshua Bengio, Aaron Courville, Pierre-Antoine Manzagol, et al. (2010). “Why does unsupervised pre-training help deep learning?” In: *Journal of Machine Learning Research* 11.Feb, pp. 625–660 (cit. on pp. 59, 60, 167).
- Erhan, Dumitru, Yoshua Bengio, Aaron Courville, and Pascal Vincent (2009). “Visualizing higher-layer features of a deep network”. In: *University of Montreal* 1341 (cit. on p. 35).
- Erhan, Dumitru, Pierre-Antoine Manzagol, et al. (2009). “The Difficulty of Training Deep Architectures and the Effect of Unsupervised Pre-Training.” In: *AISTATS*. Vol. 5, pp. 153–160 (cit. on pp. 59, 60).
- Fischer, Andreas, Emanuel Indermühle, et al. (2010). “Ground truth creation for handwriting recognition in historical documents”. In: *Proceedings of the IAPR Int. Workshop on Document Analysis Systems*. ACM, pp. 3–10 (cit. on p. 120).
- Fischer, Andreas, Andreas Keller, et al. (2012). “Lexicon-free handwritten word spotting using character HMMs”. In: *Pattern Recognition Letters* 33, pp. 934–942 (cit. on pp. 113, 115, 116, 120, 124–126, 132, 134).
- Fischer, Andreas, Markus Wüthrich, et al. (2009). “Automatic transcription of handwritten medieval documents”. In: *Proceedings of the IEEE Int. Conf. on Virtual Systems and Multimedia*. IEEE, pp. 137–142 (cit. on p. 117).
- Fletcher, Reeves and Reeves M. Colin (1964). “Function minimization by conjugate gradients”. In: *The Computer Journal* 7.2, pp. 149–154. DOI: 10.1093/comjnl/7.2.149. URL: <http://dx.doi.org/10.1093/comjnl/7.2.149> (cit. on pp. 70, 97).
- Forsyth, David et al. (2005). “Making latin manuscripts searchable using gHMMs”. In: *Proceedings of the Advances in Neural Information Processing Systems*. Vol. 17. MIT Press, p. 385 (cit. on p. 115).
- Freund, Yoav and David Haussler (1994). *Unsupervised learning of distributions of binary vectors using two layer networks*. Computer Research Laboratory [University of California, Santa Cruz] (cit. on p. 26).
- Frinken, Volkmar et al. (2012). “A novel word spotting method based on recurrent neural networks”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 34, pp. 211–224 (cit. on pp. 115, 143).
- Fukushima, Kunihiko (1980). “Neocognitron: A Self-Organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position”. In: *Biological Cybernetics* 36, pp. 193–202 (cit. on p. 15).
- Goodfellow, Ian J. et al. (2014). “Generative adversarial nets”. In: *Advances in Neural Information Processing Systems*, pp. 2672–2680 (cit. on p. 151).
- Goodfellow, Ian, Yoshua Bengio, and Aaron Courville (2016). “Deep Learning”. Book in preparation for MIT Press. URL: <http://www.deeplearningbook.org> (cit. on p. 51).
- Goto, Kazushige and Robert Van De Geijn (2008). “High-performance implementation of the level-3 BLAS”. In: *ACM Transactions on Mathematical Software (TOMS)* 35.1, p. 4 (cit. on p. 73).
- Govindaraju, Naga K. et al. (2008). “High performance discrete Fourier transforms on graphics processors”. In: *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. IEEE Press, p. 2 (cit. on p. 61).

- Graham, Benjamin (2014). “Fractional max-pooling”. In: *arXiv preprint arXiv:1412.6071* (cit. on p. 206).
- Graves, Alex et al. (2009). “A Novel Connectionist System for Unconstrained Handwriting Recognition”. In: *IEEE transactions on pattern analysis and machine intelligence* 31.5, pp. 855–868 (cit. on p. 16).
- Grosse, Roger, Helen Kwong, and Andrew Y. Ng (2007). “Shift-invariant sparse coding for audio classification”. In: *Proceedings of the Twenty-third Conference on Uncertainty in Artificial Intelligence* (cit. on p. 44).
- Günter, Simon and Horst Bunke (2003). “Optimizing the Number of States, Training Iterations and Gaussians in an HMM-based Handwritten Word Recognizer”. In: *Proceedings of the Seventh International Conference on Document Analysis and Recognition*. IEEE, pp. 472–476 (cit. on p. 134).
- Hagan, Martin T. and Mohammad B. Menhaj (1994). “Training feedforward networks with the Marquardt algorithm”. In: *Neural Networks, IEEE Transactions on* 5.6, pp. 989–993 (cit. on p. 19).
- He, Kaiming et al. (2015). “Delving deep into rectifiers: Surpassing human-level performance on imagenet classification”. In: *Proceedings of the IEEE International Conference on Computer Vision*, pp. 1026–1034 (cit. on p. 40).
- (2016). “Deep residual learning for image recognition”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 770–778 (cit. on p. 20).
- Hinton, Geoffrey E. (2002). “Training products of experts by minimizing contrastive divergence”. In: *Neural computation* 14.8, pp. 1771–1800 (cit. on pp. 20, 26).
- (2007). “Learning multiple layers of representation”. In: *Trends in Cognitive Sciences* 11, pp. 428–434 (cit. on p. 19).
- (2012). “A Practical Guide to Training Restricted Boltzmann Machines.” In: *Neural Networks: Tricks of the Trade (2nd ed.)* Ed. by Grégoire Montavon, Genevieve B. Orr, and Klaus-Robert Müller. Vol. 7700. Lecture Notes in Computer Science. Springer, pp. 599–619. ISBN: 978-3-642-35288-1. URL: <http://dblp.uni-trier.de/db/series/lncs/lncs7700.html#Hinton12> (cit. on pp. 29, 68, 73, 93).
- Hinton, Geoffrey E., Simon Osindero, and Yee-Whye Teh (2006). “A fast learning algorithm for deep belief nets”. In: *Neural computation* 18.7, pp. 1527–1554 (cit. on pp. 19, 26, 42, 98).
- Hinton, Geoffrey E. and Ruslan R. Salakhutdinov (2006). “Reducing the dimensionality of data with neural networks”. In: *Science* 313.5786, pp. 504–507 (cit. on pp. 19, 42, 50, 57, 86).
- (2012). “A better way to pretrain deep boltzmann machines”. In: *Advances in Neural Information Processing Systems*, pp. 2447–2455 (cit. on pp. 50, 51).
- Hinton, Geoffrey E. and Terrence J. Sejnowski (1986). “Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1”. In: ed. by David E. Rumelhart, James L. McClelland, and CORPORATE PDP Research Group. Cambridge, MA, USA: MIT Press. Chap. Learning and Relearning in

- Boltzmann Machines, pp. 282–317. ISBN: 0-262-68053-X. URL: <http://dl.acm.org/citation.cfm?id=104279.104291> (cit. on p. 25).
- Hinton, Geoffrey E., Nitish Srivastava, et al. (2012). “Improving neural networks by preventing co-adaptation of feature detectors”. In: *arXiv preprint arXiv:1207.0580* (cit. on pp. 61, 71).
- Hinton, Geoffrey E. and Richard S. Zemel (1994). “Autoencoders, minimum description length, and Helmholtz free energy”. In: *Advances in neural information processing systems*, pp. 3–3 (cit. on p. 150).
- Hochreiter, Sepp (1991). *Untersuchungen zu dynamischen neuronalen Netzen. Diploma thesis, Institut für Informatik, Lehrstuhl Prof. Brauer, Technische Universität München* (cit. on pp. 19, 60).
- Hochreiter, Sepp and Jürgen Schmidhuber (1997). “Long Short-Term Memory”. In: *Neural Comput.* 9.8, pp. 1735–1780. ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735. URL: <http://dx.doi.org/10.1162/neco.1997.9.8.1735> (cit. on p. 16).
- Hopfield, John J. (1982). “Neural networks and physical systems with emergent collective computational abilities”. In: *Proceedings of the national academy of sciences* 79.8, pp. 2554–2558 (cit. on p. 21).
- Hornik, Kurt, Maxwell Stinchcombe, and Halbert White (1989). “Multilayer feed-forward networks are universal approximators”. In: *Neural networks* 2.5, pp. 359–366 (cit. on p. 15).
- Hotelling, Harold (1933). “Analysis of a complex of statistical variables into principal components”. In: *J. Educ. Psych.* 24 (cit. on p. 112).
- Howe, Nicholas R. (2013). “Part-structured inkball models for one-shot handwritten word spotting”. In: *2013 12th International Conference on Document Analysis and Recognition*. IEEE, pp. 582–586 (cit. on p. 115).
- Hubel, David H. and Torsten N. Wiesel (1962). “Receptive fields, binocular interaction and functional architecture in the cat’s visual cortex”. In: *The Journal of physiology*, pp. 106–154. URL: <http://www.ncbi.nlm.nih.gov/pmc/articles/PMC1359523/pdf/jphysiol01247-0121.pdf> (cit. on p. 17).
- Iglberger, Klaus et al. (2012a). “Expression templates revisited: a performance analysis of current methodologies”. In: *SIAM Journal on Scientific Computing* 34.2, pp. C42–C69 (cit. on p. 75).
- (2012b). “High performance smart expression template math libraries”. In: *High Performance Computing and Simulation (HPCS), 2012 International Conference on*. IEEE, pp. 367–373 (cit. on p. 75).
- Impedovo, S, L Ottaviano, and S Occhinegro (1991). “Optical character recognition—a survey”. In: *International Journal of Pattern Recognition and Artificial Intelligence* 5.01n02, pp. 1–24 (cit. on p. 89).
- Ioffe, Sergey and Christian Szegedy (2015). “Batch normalization: Accelerating deep network training by reducing internal covariate shift”. In: *arXiv preprint arXiv:1502.03167* (cit. on pp. 20, 62).
- Iwana, Brian, Uchida Seiichi, and Volkmar Frinken (2016). “A Robust Dissimilarity-based Neural Network for Temporal Pattern Recognition”. In: *International Conference on Frontiers in Handwriting Recognition (ICFHR) 2016* (cit. on p. 122).

- Jaderberg, Max, Andrea Vedaldi, and Andrew Zisserman (2014). “Deep features for text spotting”. In: *European conference on computer vision*. Springer, pp. 512–528 (cit. on p. 116).
- Jain, Atishay et al. (2013). “Fundamental Challenges to Mobile Based OCR”. In: vol. 2. 5, pp. 86–101 (cit. on p. 89).
- Jarrett, Kevin, Koray Kavukcuoglu, Yann Lecun, et al. (2009). “What is the best multi-stage architecture for object recognition?” In: *2009 IEEE 12th International Conference on Computer Vision*. IEEE, pp. 2146–2153 (cit. on pp. 16, 58, 71).
- Jia, Yangqing et al. (2014). “Caffe: Convolutional Architecture for Fast Feature Embedding”. In: *arXiv preprint arXiv:1408.5093* (cit. on pp. 62, 79, 194).
- Johansson, Stig, Geoffrey N Leech, and Helen Goodluck (1978). *Manual of Information to Accompany the Lancaster-Oslo/Bergen Corpus of British English, for Use with Digital Computer*. Department of English, University of Oslo (cit. on p. 117).
- Kamal, Snigdha, Simarpreet Singh Chawla, and Nidhi Goel (2015). “Identification of numbers and positions using MATLAB to solve Sudoku on FPGA”. In: *2015 Annual IEEE India Conference (INDICON)*. IEEE, pp. 1–6 (cit. on p. 89).
- Kingma, Diederik P., Shakir Mohamed, et al. (2014). “Semi-supervised learning with deep generative models”. In: *Advances in Neural Information Processing Systems*, pp. 3581–3589 (cit. on p. 151).
- Kingma, Diederik P. and Max Welling (2013). “Auto-encoding variational bayes”. In: *arXiv preprint arXiv:1312.6114* (cit. on p. 151).
- Kingma, Diederik and Jimmy Ba (2014). “Adam: A method for stochastic optimization”. In: *arXiv preprint arXiv:1412.6980* (cit. on p. 71).
- Kovalchuk, Alon, Lior Wolf, and Nachum Dershowitz (2014). “A Simple and Fast Word Spotting Method”. In: *14th International Conference on Frontiers in Handwriting Recognition, ICFHR 2014, Crete, Greece, September 1-4, 2014*, pp. 3–8 (cit. on p. 114).
- Krishnan, Praveen, Kartik Dutta, and C. V. Jawahar (2016). “Deep Feature Embedding for Accurate Recognition and Retrieval of Handwritten Text”. In: *International Conference on Frontiers in Handwriting Recognition (ICFHR) 2016* (cit. on p. 116).
- Krizhevsky, Alex (2010). *Convolutional deep belief networks on cifar-10* (cit. on pp. 40, 44, 48, 131).
- Krizhevsky, Alex and Geoffrey Hinton (2009). “Learning multiple layers of features from tiny images”. In: (cit. on p. 206).
- Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton (2012). “Imagenet classification with deep convolutional neural networks”. In: *Advances in neural information processing systems*, pp. 1097–1105 (cit. on pp. 39, 62).
- Kullback, Solomon and Richard A. Leibler (1951). “On information and sufficiency”. In: *The annals of mathematical statistics* 22.1, pp. 79–86 (cit. on p. 25).
- Kuo, Shyh-shiaw and Oscar E. Agazzi (1994). “Keyword spotting in poorly printed documents using pseudo 2-D Hidden Markov Models”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 16, pp. 842–848 (cit. on p. 113).

- Lan, Zhenzhong et al. (2016). “The best of both worlds: Combining data-independent and data-driven approaches for action recognition”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pp. 123–132 (cit. on p. 14).
- Larochelle, Hugo and Yoshua Bengio (2008). “Classification using discriminative restricted Boltzmann machines”. In: *Proceedings of the 25th international conference on Machine learning*. ACM, pp. 536–543 (cit. on p. 50).
- Larochelle, Hugo, Yoshua Bengio, et al. (2009). “Exploring strategies for training deep neural networks”. In: *Journal of Machine Learning Research* 10, Jan, pp. 1–40 (cit. on pp. 59, 60, 167).
- Larsen, Anders Boesen Lindbo, Søren Kaae Sønderby, and Ole Winther (2015). “Autoencoding beyond pixels using a learned similarity metric”. In: *CoRR* abs/1512.09300. URL: <http://arxiv.org/abs/1512.09300> (cit. on p. 151).
- Lavrenko, Victor, Toni M. Rath, and Raghavan Manmatha (2004). “Holistic word recognition for handwritten historical documents”. In: *Proceedings of the Int. Workshop on Document Image Analysis for Libraries*. IEEE, pp. 278–287 (cit. on p. 117).
- Lawson, Chuck L. et al. (1979). “Basic linear algebra subprograms for Fortran usage”. In: *ACM Transactions on Mathematical Software (TOMS)* 5.3, pp. 308–323 (cit. on p. 73).
- LeCun, Yann (1985). “Une procédure d’apprentissage pour réseau à seuil asymétrique (a Learning Scheme for Asymmetric Threshold Networks)”. In: *Proceedings of Cognitiva 85*. Paris, France, pp. 599–604 (cit. on p. 70).
- LeCun, Yann, Yoshua Bengio, and Geoffrey E. Hinton (2015). “Deep learning”. In: *Nature* 521.7553, pp. 436–444 (cit. on pp. 20, 60, 63).
- LeCun, Yann, Bernhard Boser, et al. (1989). “Backpropagation Applied to Handwritten Zip Code Recognition”. In: *Neural Comput.* 1.4, pp. 541–551. ISSN: 0899-7667. DOI: 10.1162/neco.1989.1.4.541. URL: <http://dx.doi.org/10.1162/neco.1989.1.4.541> (cit. on pp. 15, 43).
- LeCun, Yann, Léon Bottou, et al. (1998). “Gradient-based learning applied to document recognition”. In: *Proceedings of the IEEE* 86.11, pp. 2278–2324. ISSN: 0018-9219. DOI: 10.1109/5.726791 (cit. on pp. 18, 87, 99, 197).
- LeCun, Yann, Corinna Cortes, and Christopher J. C. Burges (1998). *The MNIST database of handwritten digits* (cit. on pp. 34, 42).
- Lee, Honglak, Chaitanya Ekanadham, and Andrew Y. Ng (2008). “Sparse deep belief net model for visual area V2”. In: *Advances in neural information processing systems*, pp. 873–880 (cit. on pp. 46, 69, 131).
- Lee, Honglak, Roger Grosse, et al. (2009). “Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations”. In: *Proceedings of the 26th Annual International Conference on Machine Learning*. ACM, pp. 609–616 (cit. on pp. 43, 47–49, 69, 99).
- Lee, Honglak, Peter Pham, et al. (2009). “Unsupervised feature learning for audio classification using convolutional deep belief networks”. In: *Advances in neural information processing systems*, pp. 1096–1104 (cit. on p. 44).

- Lee, Victor W. et al. (2010). “Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU”. In: *ACM SIGARCH Computer Architecture News* 38.3, pp. 451–460 (cit. on pp. 61, 76, 185, 187, 189).
- Leydier, Yann et al. (2009). “Towards an omnilingual word retrieval system for ancient manuscripts”. In: *Pattern Recognition* 42.9, pp. 2089–2105 (cit. on p. 114).
- Liang, Jian, David Doermann, and Huiping Li (2005). “Camera-based analysis of text and documents: a survey”. In: *International Journal of Document Analysis and Recognition (IJDAR)* 7.2-3, pp. 84–104 (cit. on p. 89).
- Lloyd, Stuart (1982). “Least squares quantization in PCM”. In: *IEEE transactions on information theory* 28.2, pp. 129–137 (cit. on p. 112).
- Ly, Thanh Minh and Dung Trung Vo (2015). “A novel and automatic character extraction and recognition for Sudoku puzzle solving”. In: *Advanced Technologies for Communications (ATC), 2015 International Conference on*. IEEE, pp. 546–550 (cit. on p. 88).
- Majtner, Tomas, Sule Yildirim-Yayilgan, and Jon Yngve Hardeberg (2016). “Combining deep learning and hand-crafted features for skin lesion classification”. In: *Image Processing Theory Tools and Applications (IPTA), 2016 6th International Conference on*. IEEE, pp. 1–6 (cit. on p. 14).
- Makhzani, Alireza et al. (2015). “Adversarial autoencoders”. In: *arXiv preprint arXiv:1511.05644* (cit. on p. 151).
- Manmatha, Raghavan, Chengfeng Han, and Edward M. Riseman (1996). “Word spotting: A new approach to indexing handwriting”. In: *Proceedings of the IEEE Conf. on Computer Vision and Pattern Recognition*. IEEE, pp. 631–637 (cit. on pp. 114, 115).
- Marti, Urs V. and Horst Bunke (2001). “Using a statistical language model to improve the performance of an HMM-based cursive handwriting recognition system”. In: *Int. Journal of Pattern Recognition and Artificial Intelligence* 15, pp. 65–90 (cit. on pp. 118, 120).
- Masci, Jonathan et al. (2011). “Stacked convolutional auto-encoders for hierarchical feature extraction”. In: *Artificial Neural Networks and Machine Learning-ICANN 2011*, pp. 52–59 (cit. on pp. 58, 150).
- Matas, Jiri, Chris Galambos, and Josef Kittler (2000). “Robust Detection of Lines Using the Progressive Probabilistic Hough Transform”. In: *Comput. Vis. Image Underst.* 78.1, pp. 119–137. ISSN: 1077-3142. DOI: 10.1006/cviu.1999.0831. URL: <http://dx.doi.org/10.1006/cviu.1999.0831> (cit. on p. 90).
- Mescheder, Lars M., Sebastian Nowozin, and Andreas Geiger (2017). “Adversarial Variational Bayes: Unifying Variational Autoencoders and Generative Adversarial Networks”. In: *CoRR* abs/1701.04722. URL: <http://arxiv.org/abs/1701.04722> (cit. on p. 151).
- Minsky, Marvin and Seymour Papert (1969). *Perceptrons: An essay in computational geometry*. en. Cambridge, MA: MIT Press (cit. on p. 15).
- Mnih, Volodymyr, Hugo Larochelle, and Geoffrey E. Hinton (2012). “Conditional restricted Boltzmann machines for structured output prediction”. In: *arXiv preprint arXiv:1202.3748* (cit. on p. 50).

- Murphy, Kevin P. (2012). *Machine learning: a probabilistic perspective*. MIT press (cit. on pp. 12, 14).
- Myers, Cory, Lawrence R. Rabiner, and Andrew Rosenberg (1980). “An investigation of the use of Dynamic Time Warping for word spotting and connected speech recognition”. In: *Proceedings of the IEEE Int. Conf. on Acoustics Speech and Signal Processing*. Vol. 5. IEEE, pp. 173–177 (cit. on pp. 113, 122).
- Nair, Vinod and Geoffrey E. Hinton (2009). “3D object recognition with deep belief nets”. In: *Advances in Neural Information Processing Systems*, pp. 1339–1347 (cit. on p. 31).
- (2010). “Rectified linear units improve restricted boltzmann machines”. In: *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pp. 807–814 (cit. on pp. 39, 60).
- Neal, Radford M. (2001). “Annealed importance sampling”. In: *Statistics and Computing* 11.2, pp. 125–139 (cit. on p. 36).
- Nesterov, Yurii (1983). “A method for unconstrained convex minimization problem with the rate of convergence $O(1/k^2)$ ”. In: *Doklady ANSSSR*. Vol. 269. 3, pp. 543–547 (cit. on p. 71).
- Ngiam, Jiquan et al. (2011). “On optimization methods for deep learning”. In: *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pp. 265–272 (cit. on pp. 13, 19).
- Noh, Hyeonwoo, Seunghoon Hong, and Bohyung Han (2015). “Learning deconvolution network for semantic segmentation”. In: *Proceedings of the IEEE International Conference on Computer Vision*, pp. 1520–1528 (cit. on p. 150).
- Olshausen, Bruno A. and David J. Field (1996). “Emergence of simple-cell receptive field properties by learning a sparse code for natural images”. In: *Nature* 381.6583, pp. 607–609 (cit. on p. 46).
- (1997). “Sparse coding with an overcomplete basis set: A strategy employed by V1?” In: *Vision research* 37.23, pp. 3311–3325 (cit. on p. 151).
- Owens, John D. et al. (2007). “A survey of general-purpose computation on graphics hardware”. In: *Computer graphics forum*. Vol. 26. 1. Wiley Online Library, pp. 80–113 (cit. on pp. 61, 184).
- Perronnin, Florent and Jose A. Rodriguez-Serrano (2009). “Fisher kernels for handwritten word-spotting”. In: *2009 10th International Conference on Document Analysis and Recognition*. IEEE, pp. 106–110 (cit. on p. 115).
- Pinheiro, Pedro O. and Ronan Collobert (2014). “Recurrent Convolutional Neural Networks for Scene Labeling”. In: *Proceedings of the 31th International Conference on Machine Learning (ICML-14)*, pp. 82–90 (cit. on p. 16).
- Plötz, Thomas and Gernot A. Fink (2009). “Markov models for offline handwriting recognition: a survey”. In: *International Journal on Document Analysis and Recognition (IJDAR)* 12.4, pp. 269–298 (cit. on p. 123).
- Qian, Ning (1999). “On the momentum term in gradient descent learning algorithms”. In: *Neural networks* 12.1, pp. 145–151 (cit. on p. 71).
- Rabiner, Lawrence R. (1989). “A tutorial on hidden Markov models and selected applications in speech recognition”. In: *Proceedings of the IEEE*, pp. 257–286 (cit. on pp. 123, 124).

- Radford, Alec, Luke Metz, and Soumith Chintala (2015). “Unsupervised representation learning with deep convolutional generative adversarial networks”. In: *arXiv preprint arXiv:1511.06434* (cit. on p. 151).
- Ranzato, Marc’Aurelio and Geoffrey E. Hinton (2010). “Modeling pixel means and covariances using factorized third-order Boltzmann machines”. In: *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on*. IEEE, pp. 2551–2558 (cit. on p. 49).
- Ranzato, Marc’Aurelio, Alex Krizhevsky, et al. (2010). “Factored 3-way restricted Boltzmann machines for modeling natural images”. In: *International conference on artificial intelligence and statistics*, pp. 621–628 (cit. on p. 49).
- Ranzato, Marc’Aurelio and Yann LeCun (2007). “A sparse and locally shift invariant feature extractor applied to document images”. In: *Ninth International Conference on Document Analysis and Recognition (ICDAR 2007)*. Vol. 2. IEEE, pp. 1213–1217 (cit. on p. 151).
- Rasmussen, Carl Edward (2006). *Minimize a differentiable multivariate function, implementation in Matlab*. URL: <http://learning.eng.cam.ac.uk/carl/code/minimize/minimize.m> (cit. on pp. 70, 97).
- Rath, Toni M. and Raghavan Manmatha (2003). “Word image matching using Dynamic Time Warping”. In: *Proceedings of the IEEE Conf. on Computer Vision and Pattern Recognition*. Vol. 2. IEEE, pp. 521–527 (cit. on pp. 114, 122).
- Rath, Tony M. and Rudrapatna Manmatha (2007). “Word spotting for historical documents”. In: *Int. Journal of Document Analysis and Recognition (IJДАР)* 9, pp. 139–152 (cit. on pp. 114, 122).
- Ren, Jimmy S. J. and Li Xu (2015). “On Vectorization of Deep Convolutional Neural Networks for Vision Tasks”. In: *CoRR* abs/1501.07338. URL: <http://arxiv.org/abs/1501.07338> (cit. on p. 74).
- Retsinas, George et al. (2016). “Keyword Spotting in Handwritten Documents Using Projections of Oriented Gradients”. In: *2016 12th IAPR Workshop on Document Analysis Systems (DAS)*, pp. 411–416 (cit. on p. 115).
- Rezende, Danilo Jimenez, Shakir Mohamed, and Daan Wierstra (2014). “Stochastic backpropagation and approximate inference in deep generative models”. In: *arXiv preprint arXiv:1401.4082* (cit. on p. 151).
- Rodriguez, Jose A. and Florent Perronnin (2008). “Local gradient histogram features for word spotting in unconstrained handwritten documents”. In: *Proceedings of the Int. Conf. on Frontiers in Handwriting Recognition*, pp. 7–12 (cit. on pp. 114, 118, 122, 123).
- Ronse, Christian and Pierre A. Devijver (1984). *Connected Components in Binary Images: The Detection Problem*. New York, NY, USA: John Wiley & Sons, Inc. ISBN: 0-86380-016-5 (cit. on p. 90).
- Rose, Richard C. and Douglas B. Paul (1990). “A Hidden Markov Model based keyword recognition system”. In: *Proceedings of the Int. Conf. on Acoustics Speech, and Signal Processing*. IEEE, pp. 129–132 (cit. on p. 113).
- Rosenblatt, Frank (1958). “The perceptron: a probabilistic model for information storage and organization in the brain.” In: *Psychological review* 65.6, p. 386 (cit. on p. 15).

- Rothfeder, Jamie L., Shaolei Feng, and Toni M. Rath (2003). “Using corner feature correspondences to rank word images by similarity”. In: *Computer Vision and Pattern Recognition Workshop, 2003. CVPRW’03. Conference on*. Vol. 3. IEEE, pp. 30–30 (cit. on p. 114).
- Ruder, Sebastian (2016). “An overview of gradient descent optimization algorithms”. In: *CoRR* abs/1609.04747. URL: <http://arxiv.org/abs/1609.04747> (cit. on p. 71).
- Rumelhart, David E., Geoffrey E. Hinton, and Ronald J. Williams (1988). “Learning representations by back-propagating errors”. In: *Cognitive modeling* 5.3, p. 1 (cit. on p. 149).
- Rusiñol, Marçal et al. (2015). “Efficient segmentation-free keyword spotting in historical document collections”. In: *Pattern Recognition* 48.2, pp. 545–555 (cit. on p. 115).
- Russakovsky, Olga et al. (2015). “ImageNet Large Scale Visual Recognition Challenge”. In: *International Journal of Computer Vision (IJCV)* 115.3, pp. 211–252. DOI: 10.1007/s11263-015-0816-y (cit. on p. 208).
- Sak, Hasim, Andrew W Senior, and Françoise Beaufays (2014). “Long short-term memory recurrent neural network architectures for large scale acoustic modeling.” In: *INTERSPEECH*, pp. 338–342 (cit. on p. 16).
- Sakoe, Hiroaki and Seibi Chiba (1978). “Dynamic programming algorithm optimization for spoken word recognition”. In: *IEEE Transactions on Acoustics Speech and Signal Processing* 26, pp. 43–49 (cit. on p. 122).
- Salakhutdinov, Ruslan R. (2009). “Learning in Markov random fields using tempered transitions”. In: *Advances in neural information processing systems*, pp. 1598–1606 (cit. on p. 36).
- Salakhutdinov, Ruslan R. and Geoffrey E. Hinton (2009). “Deep boltzmann machines”. In: *International conference on artificial intelligence and statistics*, pp. 448–455 (cit. on p. 50).
- Sargano, Allah Bux, Plamen Angelov, and Zulfiqar Habib (2017). “A comprehensive review on handcrafted and learning-based action representation approaches for human activity recognition”. In: *Applied Sciences* 7.1, p. 110 (cit. on p. 14).
- Scott, Guy L. and Christopher H. Longuet-Higgins (1991). “An algorithm for associating the features of two images”. In: *Proceedings of the Royal Society of London B: Biological Sciences* 244.1309, pp. 21–26 (cit. on p. 114).
- Sharma, Arjun and Pramod Sankar (2015). “Adapting off-the-shelf CNNs for word spotting and recognition”. In: *2015 13th International Conference on Document Analysis and Recognition (ICDAR)*, pp. 986–990 (cit. on p. 116).
- Shewchuk, Jonathan R. (1994). *An Introduction to the Conjugate Gradient Method Without the Agonizing Pain*. Tech. rep. Pittsburgh, PA, USA (cit. on pp. 70, 97).
- Silberstein, Mark et al. (2008). “Efficient computation of sum-products on GPUs through software-managed cache”. In: *Proceedings of the 22nd annual international conference on Supercomputing*. ACM, pp. 309–318 (cit. on p. 61).

- Simard, Patrice, David Steinkraus, and John C. Platt (2003). “Best Practices for Convolutional Neural Networks Applied to Visual Document Analysis.” In: *ICDAR*. Vol. 3, pp. 958–962 (cit. on pp. 62, 71).
- Simha, Pramod, K. V. Suraj, and Tejas Ahobala (2012). “Recognition of numbers and position using image processing techniques for solving Sudoku Puzzles”. In: *Advances in Engineering, Science and Management (ICAESM), 2012*. Nagapattinam, Tamil Nadu: IEEE, pp. 1–5. ISBN: 978-1-4673-0213-5 (cit. on p. 88).
- Simonyan, Karen and Andrew Zisserman (2014). “Very deep convolutional networks for large-scale image recognition”. In: *arXiv preprint arXiv:1409.1556* (cit. on p. 62).
- Smolensky, Paul (1986). “Information processing in dynamical systems: Foundations of harmony theory”. In: (cit. on p. 20).
- Sohn, Kihyuk and Honglak Lee (2012). “Learning Invariant Representations with Local Transformations”. In: *ICML* (cit. on p. 43).
- Sohn, Kihyuk, Honglak Lee, and Xinchun Yan (2015). “Learning structured output representation using deep conditional generative models”. In: *Advances in Neural Information Processing Systems*, pp. 3483–3491 (cit. on p. 151).
- Srivastava, Nitish (2013). “Improving neural networks with dropout”. PhD thesis. University of Toronto (cit. on p. 61).
- Srivastava, Nitish et al. (2014). “Dropout: a simple way to prevent neural networks from overfitting.” In: *Journal of Machine Learning Research* 15.1, pp. 1929–1958 (cit. on pp. 16, 20, 61).
- Stauffer, Michael, Andreas Fischer, and Kaspar Riesen (2016). “A Novel Graph Database for Handwritten Word Images”. In: *Structural, Syntactic, and Statistical Pattern Recognition - Joint IAPR International Workshop, S+SSPR 2016, Mérida, Mexico, November 29 - December 2, 2016, Proceedings*, pp. 553–563 (cit. on p. 137).
- Sudholt, Sebastian and Gernot A. Fink (2016). “PHOCNet: A Deep Convolutional Neural Network for Word Spotting in Handwritten Documents”. In: *International Conference on Frontiers in Handwriting Recognition (ICFHR) 2016* (cit. on p. 116).
- Sutskever, Ilya and Geoffrey E. Hinton (2007). “Learning multilevel distributed representations for high-dimensional sequences”. In: *International Conference on Artificial Intelligence and Statistics*, pp. 548–555 (cit. on p. 50).
- Sutskever, Ilya, Geoffrey E. Hinton, and Graham W. Taylor (2009). “The recurrent temporal restricted boltzmann machine”. In: *Advances in Neural Information Processing Systems*, pp. 1601–1608 (cit. on p. 50).
- Sutskever, Ilya and Tijmen Tieleman (2010). “On the Convergence Properties of Contrastive Divergence.” In: *AISTATS*. Vol. 9, pp. 789–795 (cit. on p. 26).
- Suzuki, Satoshi and Keiichi Abe (1985). “Topological structural analysis of digitized binary images by border following.” In: *Computer Vision, Graphics, and Image Processing* 30, pp. 32–46 (cit. on p. 91).
- Szegedy, Christian, Sergey Ioffe, et al. (2016). “Inception-v4, inception-resnet and the impact of residual connections on learning”. In: *arXiv preprint arXiv:1602.07261* (cit. on p. 208).

- Szegedy, Christian, Wei Liu, et al. (2015). “Going deeper with convolutions”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 1–9 (cit. on pp. 20, 62).
- Tang, Yichuan, Ruslan R. Salakhutdinov, and Geoffrey E. Hinton (2012). “Robust boltzmann machines for recognition and denoising”. In: *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*. IEEE, pp. 2264–2271 (cit. on pp. 143, 162).
- Taylor, Graham W. and Geoffrey E. Hinton (2009). “Factored Conditional restricted Boltzmann machines for modeling motion style”. In: *Proceedings of the 26th annual international conference on machine learning*. ACM, pp. 1025–1032 (cit. on pp. 43, 50).
- Taylor, Graham W., Geoffrey E. Hinton, and Sam T. Roweis (2006). “Modeling human motion using binary latent variables”. In: *Advances in neural information processing systems*, pp. 1345–1352 (cit. on p. 50).
- Team, Deeplearning4j Development (2015). *Deeplearning4j: Open-source distributed deep learning for the JVM*. URL: <http://deeplearning4j.org> (cit. on p. 195).
- Terasawa, Kengo and Yuzuru Tanaka (2009). “Slit style HOG feature for document image word spotting”. In: *Proceedings of the IEEE Int. Conf. on Document Analysis and Recognition*. IEEE, pp. 116–120 (cit. on pp. 114, 118, 122, 123).
- Thomas, Simon, Clément Chatelain, Laurent Heutte, and Thierry Paquet (2010). “An information extraction model for unconstrained handwritten documents”. In: *Pattern Recognition (ICPR), 2010 20th International Conference on*. IEEE, pp. 3412–3415 (cit. on p. 115).
- Thomas, Simon, Clément Chatelain, Laurent Heutte, Thierry Paquet, and Yousri Kessentini (2015). “A deep HMM model for multiple keywords spotting in handwritten documents”. In: *Pattern Analysis and Applications* 18.4, pp. 1003–1015 (cit. on p. 115).
- Tieleman, Tijmen (2008). “Training restricted Boltzmann machines using approximations to the likelihood gradient”. In: *Proceedings of the 25th international conference on Machine learning*. ACM, pp. 1064–1071 (cit. on p. 35).
- Tieleman, Tijmen and Geoffrey E. Hinton (2009). “Using fast weights to improve persistent contrastive divergence”. In: *Proceedings of the 26th Annual International Conference on Machine Learning*. ACM, pp. 1033–1040 (cit. on p. 35).
- (2012). *Lecture 6.5—RmsProp: Divide the gradient by a running average of its recent magnitude*. COURSERA: Neural Networks for Machine Learning (cit. on p. 71).
- Van Horn, Adam (2012). *Extraction of Sudoku Puzzles using the Hough transform*. Tech. rep. University of Kansas, Department of Electrical Engineering and Computer Science (cit. on p. 88).
- Vincent, Pascal, Hugo Larochelle, Yoshua Bengio, et al. (2008). “Extracting and composing robust features with denoising autoencoders”. In: *Proceedings of the 25th international conference on Machine learning*. ACM, pp. 1096–1103 (cit. on pp. 58, 59, 151).

- Vincent, Pascal, Hugo Larochelle, Isabelle Lajoie, et al. (2010). “Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion”. In: *Journal of Machine Learning Research* 11.Dec, pp. 3371–3408 (cit. on pp. 151, 153, 167).
- Viterbi, Andrew (1967). “Error bounds for convolutional codes and an asymptotically optimum decoding algorithm”. In: *IEEE transactions on Information Theory* 13.2, pp. 260–269 (cit. on p. 124).
- Wager, Stefan, Sida Wang, and Percy S. Liang (2013). “Dropout Training as Adaptive Regularization”. In: *Advances in Neural Information Processing Systems* 26. Ed. by C. J. C. Burges et al. Curran Associates, Inc., pp. 351–359. URL: <http://papers.nips.cc/paper/4882-dropout-training-as-adaptive-regularization.pdf> (cit. on p. 61).
- Wan, Li et al. (2013). “Regularization of neural networks using dropconnect”. In: *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, pp. 1058–1066 (cit. on pp. 61, 197).
- Werbos, Paul J. (1982). “Applications of advances in nonlinear sensitivity analysis”. In: *System modeling and optimization*. Springer, pp. 762–770 (cit. on p. 15).
- Wicht, Baptiste, Andreas Fischer, and Jean Hennebert (2016a). “Deep Learning Features for Handwritten Keyword Spotting”. In: *International Conference on Pattern Recognition (ICPR)* (cit. on p. 131).
- (2016b). “Keyword Spotting with Convolutional Deep Belief Networks and Dynamic Time Warping”. In: *International Conference on Artificial Neural Networks*. Springer International Publishing, pp. 113–120 (cit. on pp. 122, 131).
- (2016c). “On CPU Performance Optimization of Restricted Boltzmann Machine and Convolutional RBM”. In: *IAPR Workshop on Artificial Neural Networks in Pattern Recognition*. Springer International Publishing, pp. 163–174 (cit. on pp. 62, 75).
- Wicht, Baptiste and Jean Hennebert (2014). “Camera-based Sudoku recognition with deep belief network”. In: *Soft Computing and Pattern Recognition (SoC-PaR), 2014 6th International Conference of*, pp. 83–88. DOI: 10.1109/SOCPAR.2014.7007986 (cit. on pp. 88, 91, 97).
- (2015). “Mixed handwritten and printed digit recognition in Sudoku with Convolutional Deep Belief Network”. In: *Document Analysis and Recognition (ICDAR), 2015 13th International Conference on*. IEEE, pp. 861–865 (cit. on pp. 86, 88, 97, 101).
- Wikipedia (2016). *Comparison of Deep Learning software*. Online; accessed 05-December-2016. URL: https://en.wikipedia.org/wiki/Comparison_of_deep_learning_software (cit. on p. 214).
- Wu, Haibing and Xiaodong Gu (2015). “Towards dropout training for convolutional neural networks”. In: *Neural Networks* 71, pp. 1–10 (cit. on p. 61).
- Xu, Bing et al. (2015). “Empirical evaluation of rectified activations in convolutional network”. In: *arXiv preprint arXiv:1505.00853* (cit. on p. 40).
- El-Yacoubi, Mounim A., Michel Gilloux, and Jean-Michel Bertille (2002). “A statistical approach for phrase location and recognition within a text line: an appli-

- cation to street name recognition”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 24.2, pp. 172–188 (cit. on p. 115).
- Yang, Zhiyi, Yating Zhu, and Yong Pu (2008). “Parallel image processing based on CUDA”. In: *Computer Science and Software Engineering, 2008 International Conference on*. Vol. 3. IEEE, pp. 198–201 (cit. on p. 61).
- Yosinski, Jason and Hod Lipson (2012). “Visually debugging restricted boltzmann machine training with a 3d example”. In: *Representation Learning Workshop, 29th International Conference on Machine Learning* (cit. on p. 35).
- Zamora-Martinez, Francisco, Javier Munoz-Almaraz, and Juan Pardo (2016). “Integration of Unsupervised and Supervised Criteria for Deep Neural Networks Training”. In: *International Conference on Artificial Neural Networks*. Springer, pp. 55–62 (cit. on p. 59).
- Zeiler, Matthew D (2012). “ADADELTA: an adaptive learning rate method”. In: *arXiv preprint arXiv:1212.5701* (cit. on p. 71).
- Zeiler, Matthew D. and Rob Fergus (2013). “Stochastic pooling for regularization of deep convolutional neural networks”. In: *arXiv preprint arXiv:1301.3557* (cit. on p. 61).
- (2014). “Visualizing and understanding convolutional networks”. In: *European Conference on Computer Vision*. Springer, pp. 818–833 (cit. on p. 34).
- Zeiler, Matthew D., Marc’Aurelio Ranzato, et al. (2013). “On rectified linear units for speech processing”. In: *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*. IEEE, pp. 3517–3521 (cit. on p. 39).
- Zhang, Bin, Sargur N Srihari, and Chen Huang (2003). “Word image retrieval using binary features”. In: *Proceedings of the Electronic Imaging 2004*. Int. Society for Optics and Photonics, pp. 45–53 (cit. on p. 114).

Index

- ANN, 14, 70, 197
- auto-encoder, 41, 149
 - convolutional, 34, 150
 - denoising, 150
 - dense, 150
 - sparse, 151
 - variational, 151
- CNN, 16, 187, 190, 199, 206, 208
- Contrastive Divergence, 25, 201
 - Persistent, 35
- Convolution, 73
 - full, 44, 75, 155, 184, 190
 - valid, 44, 74, 155, 182, 187
- CPU, 73, 181, 195
- Data set, 86
 - CIFAR-10, 195, 206
 - GW, 116, 117
 - IAM, 116, 117
 - ImageNet, 195, 208
 - MNIST, 33, 58, 72, 87, 96, 151, 195, 197
 - PAR, 116, 117
- DBM, 50
- DBN, 41, 69
 - Convolutional, 48, 69
- Deep Learning, 3, 19
- DLL, 67, 193
- DTW, 122, 126, 151
- ETL, 181
- Expression Templates, 75
- GPU, 76, 181, 184, 190, 195
- HMM, 123, 134
- MLP, 70, 197
- Pooling, 18, 46
 - Average, 71
 - Max, 43, 46, 48, 69, 71, 193
 - Probabilistic Max, 46, 48, 69
- RBM, 20, 48, 68, 149, 193, 201
 - Convolutional, 43, 45, 69, 149, 193, 204
 - Discriminative, 49
 - Mean-Covariance, 49
 - Spike and Slab, 50
 - Temporal, 50
- RELU, 39, 60, 68, 130
- Sakoe-Chiba band, 122, 151
- sparsity, 31, 33, 40, 45, 69, 130
- spotting, 110, 151, 175
- Sudoku, 85, 175
- SVM, 19, 101
- vanishing gradient, 19, 60
- vectorization, 73, 182, 184

Glossary

auto-encoder A model able to encode some input using an intermediate representation and decode from this representation into the original input space. Often used to extract features from input data.

convolution A convolution is an operation in which a function (the kernel) is applied at each position of some input function, resulting in an output function. Visually, this can be seen as applying a 2D kernel matrix at each possible position of an image and computing the dot product at each of these positions, resulting in an output matrix filled with these local products .

CUDA Parallel computing platform allowing developers to use NVidia Graphics Processing Units (GPUs) for general purpose processing.

full convolution A convolution returning the complete result of the convolution, this is the default convolution. By definition, the size of the result is $Output \triangleq Input + Filter - 1$.

MNIST Standard data set for digit recognition. While considered an easy task, this data set is often used as a benchmark for new algorithms.

valid convolution A convolution returning only the parts of the convolution that can be computed without zero-padding. By definition, the size of the result is $Output \triangleq Input - Filter + 1$.

vanishing gradient A problem occurring when training neural networks with several layers in which the gradients of a layer are becoming increasingly small.

Acronyms

AAE Adversarial Auto-Encoder.

AIS Annealed Importance Sampling.

ANN Artificial Neural Network.

AP Average Precision.

AVX Advanced Vector eXtensions.

BLAS Basic Linear Algebra Subprograms.

CAE Convolutional Auto-Encoder.

CD Contrastive Divergence.

CDBN Convolutional Deep Belief Network.

CG Conjugate Gradient.

CNN Convolutional Neural Network.

CPU Central Processing Unit.

CRBM Convolutional Restricted Boltzmann Machine.

CVAE Conditional Variational Auto-Encoder.

DAE Denoising Auto-Encoder.

DBM Deep Boltzmann Machine.

DBN Deep Belief Network.

DFT Discrete Fourier Transform.

DLL Deep Learning Library.

DTW Dynamic Time Warping.

EBM Energy Based Model.

FCNN Fully Connected Neural Network.

FFT Fast Fourier Transform.

FPGA Field Programmable Gate Array.

GAN Generative Adversarial Network.

gHMM generalized Hidden Markov Model.

GPGPU General Purpose Graphical Processing Unit.

GPU Graphical Processing Unit.

HMM Hidden Markov Model.

HOG Histogram of Oriented Gradient.

L-BFGS Limited-Memory Broyden-Fletcher-Goldfarb-Shanno.

LBP Local Binary Pattern.

LCN Local Contrast Normalization.

LSTM Long Short Term Memory.

MAP Mean Average Precision.

MCMC Markov Chain Monte-Carlo.

MKL Math Kernel Library.

MLP Multi-Layer Perceptron.

NaN Not-a-Number.

OCR Optical Character Recognition.

PCA Principal Component Analysis.

PCD Persistent Contrastive Divergence.

PHOC Pyramid Histogram of Oriented Characters.

PL Pseudo-Likelihood.

PMP Probabilistic Max Pooling.

PoE Product of Experts.

POG Projection of Oriented Gradients.

RBM Restricted Boltzmann Machine.

ReLU Rectified Linear Unit.

RNN Recurrent Neural Network.

SAE Stacked Auto-Encoder.

SDAE Stacked Denoising Auto-Encoder.

SGD Stochastic Gradient Descent.

SIMD Single Instruction Multiple Data.

SSE Streaming SIMD Extensions.

SVM Support Vector Machine.

VAE Variational Auto-Encoder.

List of Figures

1.1	A typical supervised Machine Learning model, for classification tasks. The model is trained using some algorithm and some training data. It can then be used to predict a label from any input.	2
2.1	An exemplary feed-forward neural network with three layers. The input layer will receive the input data, a vector \mathbf{x} of D elements, in this case $D = 3$. The hidden layer will compute a new representation of the input. Finally, the output layer will compute the answer $\tilde{\mathbf{y}}$ of the network from the hidden representation. Assuming an adequate learning process with convergence, the $\tilde{\mathbf{y}}$ may represent the probabilities of observing 2 objects in the exemplary image. Each edge between two neurons is associated to a weight.	14
2.2	A Convolutional Neural Network with two convolutional layers and two pooling layers. The input layer will receive the input data. The final layer is simply the concatenation of the feature maps of the previous layer.	17
2.3	Abstract view of a Restricted Boltzmann Machine, with four visible units \mathbf{v} and three hidden units \mathbf{h} . There are no connections between units of the same layer.	21
2.4	Graphical representation of the Contrastive Divergence Algorithm. The algorithm CD-K stops at $t=K$. Each iteration performs a full Gibbs step.	28
2.5	Impact of different learning rates on Contrastive Divergence training. This is for a subset of the MNIST data set.	32
2.6	Impact of different sparsity targets on Contrastive Divergence training, with the same learning rate. This is for a subset of the MNIST data set. When the target is too small, the convergence of the reconstruction error is impeded.	33
2.7	Visual representation of the features learned by an RBM on the MNIST data set.	34

2.8	Graphical representation of the Persistent Contrastive Divergence Algorithm. Only the initialization of v_0 differs from CD. An iteration is one update of the weights.	35
2.9	Visual aspect of the main activation functions used for an RBM. On the left, the logistic sigmoid and identity functions. On the right, the sofplus and rectifier functions.	37
2.10	Visual representation of a Deep Belief Network	41
a	Abstract representation of a three-layer Deep Belief Network. Each layer is a Restricted Boltzmann Machine.	41
b	Unit-level view of a three-layer Deep Belief Network	41
2.11	Visual representation of a Convolutional Restricted Boltzmann Machine.	43
2.12	Visual representation of Probabilistic Max Pooling applied to a Convolutional Restricted Boltzmann Machine.	46
2.13	Features learned by a three-level CDBN with Probabilistic Max Pooling on two different Caltech 101 categories. From (H. Lee, Grosse, et al., 2009).	49
2.14	Three-layer Deep Boltzmann Machine	51
3.1	Example of an auto-encoder neural network. The network is trained to reconstruct its input \mathbf{x} , targetting to obtain a $\hat{\mathbf{x}}$ output as close as possible to the input \mathbf{x} . The activations of the hidden layer represents the features that the network tries to learn.	58
4.1	Visualization of the convolutional filters learned with a Convolutional RBM in DLL, on the MNIST dataset. These are the weights of the input layer, showing that it is learning to detect various strokes.	72
4.2	A 'valid' convolution of a 5x5 image with a 3x3 kernel. The kernel will be applied to every possible position inside the image.	74
4.3	Training time performance comparison of the frameworks on a Convolutional Neural Network experiment, on CPU and on GPU.	80
5.1	Exemplary images from the Sudoku Recognition data set (SRD)	87
a	A non-filled grid	87
b	A grid with mixed inputs	87
5.2	The main steps for detection of a Sudoku puzzle inside an image, from the original image to the detected digits.	91
a	The original image	91
b	The detected lines	91

c	The detected grid cells and their indices	91
d	The final detected digits	91
5.3	Abstract view of the Deep Belief Network used for classification of the Sudoku digits. Each layer of the network is an RBM.	92
a	Model for the <i>V2</i> data set	92
b	Model for the <i>mixed</i> data set	92
5.4	Evolution of the error for training the Sudoku Recognition Network, during fine-tuning, after unsupervised pretraining on the <i>V2</i> data set.	95
a	Evolution of the training error during the first 2000 mini-batches epochs of fine-tuning the Sudoku Recognition network, for the system I on the <i>V2</i> data set.	95
b	Evolution of the mini-batch error during the first 2000 mini-batches epochs of fine-tuning the Sudoku Recognition network, for the system I on the <i>V2</i> data set.	95
5.5	Evolution of the training error during the first 2000 mini-batches epochs of fine-tuning the Sudoku Recognition network, with different epochs of unsupervised pretraining, for the DBN system, on the <i>V2</i> data set.	96
5.6	Evolution of the training error during the first 10000 mini-batches epochs of fine-tuning the Sudoku Recognition network, with different epochs of unsupervised pretraining, for the System I, on the <i>mixed</i> data set.	97
5.7	Abstract view of the Convolutional DBN used for classification of the Sudoku digits.	98
a	<i>V2</i> data set	98
b	<i>mixed</i> data set	98
5.8	Evolution of the training error during the first 3000 mini-batches epochs of fine-tuning the Sudoku Recognition network, with different epochs of unsupervised pretraining, for the Convolutional DBN system, on the <i>V2</i> data set.	101
5.9	Evolution of the training error during the first 15000 mini-batches epochs of fine-tuning the Sudoku Recognition network, with different epochs of unsupervised pretraining, for the Convolutional DBN system, on the <i>mixed</i> data set.	102
6.1	Spotting of the keywords Regiment and Diligence in a handwritten letter from George Washington.	110
6.2	Extracts from exemplary pages from the three data sets used for keyword spotting.	116

a	IAM database	116
b	PAR database	116
c	GW database	116
6.3	The Keyword Spotting system developped during this research. The system extracts features directly from pixels and passes them to a DTW or an HMM classifier to perform keyword spotting.	119
6.4	Two images of preprocessed lines from the GW (top) and PAR (bottom) data sets.	120
6.5	The Convolutional Deep Belief Network used for feature extraction in the Keyword Spotting System.	121
6.6	Sakoe-Chiba band constraint for Dynamic Time Warping. The warping path is constrained to the darker zone. Each point of the sequence A will be matched to a point in the sequence B.	123
6.7	The Hidden Markov Models (HMMs) used for keyword spotting. . .	124
a	One character HMM	124
b	The HMM Keyword model	124
c	The HMM Filler model	124
6.8	Recall-Precision performance of the Keyword Spotting System, with DTW, on the GW data set.	127
a	Local scenario	127
b	Global scenario	127
6.9	Recall-Precision performance of the Keyword Spotting System, with DTW, on the PAR data set.	128
a	Local scenario	128
b	Global scenario	128
6.10	Keyword spotting on the George Washington dataset with different widths of the sliding window.	129
6.11	Keyword spotting on the George Washington dataset with different numbers of convolutional filters in each layer.	130
6.12	Impact of normalization on the keyword spotting features with the DTW classifier, on the GW data set	133
6.13	Recall-Precision performance of the Keyword Spotting System, with HMM, on the GW data set.	135
a	Local scenario	135
b	Global scenario	135
6.14	Recall-Precision performance of the Keyword Spotting System, with HMM, on the PAR data set.	136

a	Local scenario	136
b	Global scenario	136
6.15	Recall-Precision performance of the Keyword Spotting System, with HMM, on the PAR data set.	136
a	Local scenario	136
b	Global scenario	136
6.16	Recall-Precision performance of the Keyword Spotting System, using grayscale images, with DTW, on the GW data set.	139
a	Local scenario	139
b	Global scenario	139
6.17	Recall-Precision performance of the Keyword Spotting System, on grayscale images, with HMM, on the GW data set.	139
a	Local scenario	139
b	Global scenario	139
7.1	Example of an auto-encoder neural network. The network is trained as a standard neural network. Only the hidden layer is used for feature extraction.	150
7.2	Comparison of Dense Auto-Encoders and RBM on a Keyword Spotting task, using the George Washington data set.	153
a	Global Scenario	153
b	Local Scenario	153
7.3	Comparison of Deep Auto-Encoder and Stacked Auto-Encoder . . .	154
a	Deep Auto-Encoder	154
b	Stacked Auto-Encoder	154
7.4	Comparison of Deep and Stacked Dense Auto-Encoders and DBN on a Keyword Spotting task, using the George Washington data set.	155
a	Global Scenario	155
b	Local Scenario	155
7.5	Example of a convolutional auto-encoder neural network. The network is trained as a standard neural network. Only the first layer is used for feature extraction.	156
7.6	Comparison of Convolutional Auto-Encoders and CRBMs on a Keyword Spotting task, using the George Washington data set.	157
a	Global Scenario	157
b	Local Scenario	157

7.7	Comparison of Convolutional Auto-Encoders and CRBMs, with pooling, on a Keyword Spotting task, using the George Washington data set.	157
a	Global Scenario	157
b	Local Scenario	157
7.8	Comparison of Deep Convolutional Auto-Encoders and Deep CRBM on a Keyword Spotting task, using the George Washington data set.	158
a	Global Scenario	158
b	Local Scenario	158
7.9	Comparison of Deep Convolutional Auto-Encoders and Deep CRBM on a Keyword Spotting task, with pooling layers, using the George Washington data set.	159
a	Global Scenario	159
b	Local Scenario	159
7.10	Comparison of Deep and Stacked Hybrid Auto-Encoders and Hybrid DBN on a Keyword Spotting task, using the George Washington data set.	161
a	Global Scenario	161
b	Local Scenario	161
7.11	Comparison of Deep and Stacked Hybrid Auto-Encoders and Hybrid DBN, with pooling, on a Keyword Spotting task, using the George Washington data set.	161
a	Global Scenario	161
b	Local Scenario	161
7.12	Comparison of Denoising Dense Auto-Encoders and Denoising RBM on a Keyword Spotting task, using the George Washington data set.	163
a	Global Scenario	163
b	Local Scenario	163
7.13	Comparison of Denoising Convolutional Auto-Encoders and Denoising CRBM, with pooling, on a Keyword Spotting task, using the George Washington data set.	164
a	Global Scenario	164
b	Local Scenario	164
7.14	Comparison of Denoising Deep Convolutional Auto-Encoders and Denoising Deep CRBM on a Keyword Spotting task, with pooling layers, using the George Washington data set.	165
a	Global Scenario	165

b	Local Scenario	165
A.1	Comparison of the performance of Matrix-Matrix Multiplication on CPU and GPU, on different floating point precisions.	186
a	Single-precision floating point	186
b	Double-precision floating point	186
c	Single-precision complex	186
d	Double-precision complex	186
A.2	Comparison of the performance of the Fast-Fourier Transform on CPU and GPU, on different floating point precisions.	188
a	Single-precision complex	188
b	Double-precision complex	188
c	Single-precision complex, 512 transforms	188
d	Double-precision complex, 512 transforms	188
A.3	Comparison of the performance of the Batch Valid Convolution on CPU and GPU, with different image and kernel sizes.	189
a	128x128 images, 16x16 kernels	189
b	128x128 images, 9x9 kernels	189
c	28x28 images, 9x9 kernels	189
d	28x28 images, 3x3 kernels	189
A.4	Comparison of the performance of the Batch Full Convolution on CPU and GPU, with different image and kernel sizes.	191
a	128x128 images, 16x16 kernels	191
b	128x128 images, 9x9 kernels	191
c	28x28 images, 9x9 kernels	191
d	28x28 images, 3x3 kernels	191
B.1	Evolution of the training error for the different frameworks on the Fully-Connected Neural Network experiment, on the MNIST data set.	198
a	CPU	198
b	GPU	198
B.2	Training time performance comparison of the frameworks on a Fully-Connected Neural Network experiment, on the MNIST data set, on CPU and on GPU. DLL has only been used in CPU mode.	199
B.3	Evolution of the training error for the different frameworks on the Convolutional Neural Network experiment, on the MNIST data set.	200

a	CPU	200
b	GPU	200
B.4	Training time performance comparison of the frameworks on a Convolutional Neural Network experiment, on the MNIST data set, on CPU and on GPU. DLL has only been used in CPU mode.	201
B.5	Evolution of the reconstruction error for the different frameworks on the Restricted Boltzmann Machine experiment, on the MNIST data set.. The curve for DeepLearning4J is going up and staying very high after this point.	202
a	CPU	202
b	GPU	202
B.6	Training time performance comparison of the frameworks on a Restricted Boltzmann Machine training experiment, on the MNIST data set, on CPU and on GPU. DLL was only used in CPU mode.	203
B.7	Evolution of the reconstruction error for the different frameworks on the Convolutional Restricted Boltzmann Machine experiment, on the MNIST data set.	205
a	CPU	205
b	GPU	205
B.8	Training time performance comparison of the frameworks on a Convolutional Restricted Boltzmann Machine training experiment, on the MNIST data set, on CPU and on GPU. DLL was only used in CPU mode.	205
B.9	Training time performance comparison of the frameworks on a Convolutional Neural Network experiment, on the CIFAR-10 data set, on CPU and on GPU. DLL has only been used in CPU mode.	207
B.10	Training time performance comparison of the frameworks on the ImageNet task, on CPU and on GPU. The time scale is logarithmic. The time is the average time necessary for the training of one batch of 128 elements. DLL has only been used in CPU mode.	209

List of Tables

5.1	Accuracy of Sudoku Detection and Recognition system. A Sudoku is correctly recognized if every one of its 81 cells is correctly classified. System I is based on a fully-connected DBN while System II is based on a Convolutional DBN.	103
a	<i>V2</i> data set	103
b	<i>mixed</i> data set	103
5.2	Training time for the Sudoku Recognition System, in seconds. The total time includes the time necessary to load the data and detect the digits.	103
a	System I	103
b	System II	103
5.3	Computing time, in microseconds, necessary for each step of the proposed system for Sudoku Detection and Recognition. With the System I on the <i>V2</i> data set.	104
6.1	Statistics on the entire data sets used for keyword spotting.	117
6.2	Mean Average Precision (MAP) and Average Precision (AP) for the different features when using DTW as classifier. The best feature set is indicated in bold. The absolute and relative improvements and error reduction of the proposed feature set over the best baseline is also mentioned.	126
6.3	Training parameters for the feature extraction system for keyword spotting, used for Contrastive Divergence training.	131
6.4	Mean Average Precision (MAP) and Average Precision (AP) for the different features when using HMM as classifier. The best feature set is indicated in bold. The absolute and relative improvements and error reduction of the proposed feature set over the best baseline is also mentioned.	134

6.5	Mean Average Precision (MAP) and Average Precision (AP) for the system with grayscale images, with both classifiers. The best feature set is indicated in bold. The absolute and relative improvements and error reduction of the proposed feature set over the system with binary images.	138
6.6	Time necessary to evaluate the results of one set of the GW data set with each feature set, the time for training the HMM is included. All results are in seconds.	140
6.7	Time necessary to train the feature extractor. L1 and L2 are the times necessary to train the first layer, respectively the second layer. For comparison, the times for training and evaluation with HMM are included. All results are in seconds.	141
6.8	Number of patches for each data set and number of patches processed by seconds by each layer during training.	141
7.1	Mean Average Precision (MAP) and Average Precision (AP) for each different model tested during this experiment. Each model includes the best result obtained with the RBM model and the equivalent auto-encoder model. A bold number indicates the best for each experiment.	166
A.1	Performance of a single-precision valid convolution, in microseconds. A is the naive matrix multiplication, B is the MKL version and C is the parallel MKL version. The speedup is comparing the GEMM(B) version with the Vectorized version. The Vectorized version uses SSE and AVX to perform up to 8 single-precision floating points in one CPU cycle.	183
A.2	Performance of a single-precision valid convolution of 1 image with 100 kernels, in milliseconds. The speedup is comparing the GEMM(C) version with the Vectorized version. The Vectorized version uses SSE and AVX to perform up to 8 single-precision floating points in one CPU cycle.	183
A.3	Performance of a single-precision full convolution, in microseconds. The speedup is comparing the FFT(C) version with the Vectorized version. The Vectorized version uses SSE and AVX to perform up to 8 single-precision floating points in one CPU cycle.	185
A.4	Performance of a single-precision full convolution of 1 image with 100 kernels, in milliseconds. The speedup is comparing the FFT(C) version with the Vectorized version. The Vectorized version uses SSE and AVX to perform up to 8 single-precision floating points in one CPU cycle.	185

B.1	Final test error obtained by each framework on the Fully-Connected Neural Network experiment, on the MNIST data set, on CPU and GPU. The numbers in bold are from the best performing frameworks.	198
B.2	Final test error obtained by each framework on the Convolutional Neural Network experiment, on the MNIST data set, on CPU and GPU. The numbers in bold are from the best performing frameworks.	200
B.3	Final test reconstruction error obtained by each framework on the Restricted Boltzmann Machine experiment, on CPU and GPU. The numbers in bold are from the best performing frameworks.	202
B.4	Final test reconstruction error obtained by each framework on the Convolutional Restricted Boltzmann Machine experiment, on CPU and GPU. The numbers in bold are from the best performing frameworks.	204
B.5	Final test error obtained by each framework on the Convolutional Neural Network experiment, on the CIFAR-10 data set, on CPU and GPU. The numbers in bold are from the best performing frameworks.	206
B.6	Final training error obtained by each framework on the Convolutional Neural Network experiment, on the ImageNet data set (ILSVRC 2012), on CPU and GPU. The numbers in bold are from the best performing frameworks.	209

List of Listings

- 4.1 Example of Smart Expression Templates in DLL to define the activation probabilities of a Restricted Boltzmann Machine. 76
- 4.2 Example of a three-layer network defined with the DLL preprocessor 79

List of Algorithms

2.1	Standard CD-K algorithm. This performs one epoch of training through the complete training set.	28
2.2	Mini-batch CD-k algorithm, one epoch	30
2.3	Deep Belief Network training algorithm	42
4.1	Compute a valid convolution $\mathbf{C} = \mathbf{I} \bullet_v \mathbf{W}$ with a Matrix-Matrix Multiplication	74
4.2	Compute a full convolution $\mathbf{C} = \mathbf{I} * \mathbf{W}$ with FFT	75

Resume

Education

- **2013–2017:** Ph.D in Computer Science, University of Fribourg, Fribourg, advisors: Jean Hennebert, Andreas Fischer, Rolf Ingold
- **2011–2013:** Master of Science, Computer Science, HES-SO University of applied Sciences, Western Switzerland, advisor: Omar Abou Khaled
- **2008–2011:** Bachelor of Science, Computer Science, University of applied Sciences, Fribourg, advisor: Frédéric Bapst

Experience

- **2013–present:** Scientific Collaborator, HEIA-FR, Fribourg, CH, Machine Learning, Phonetic algorithms, Performance and Benchmarking
- **2011–2012:** Software Engineer, HEIA-FR, Development of school intranet applications with Sharepoint and .NET.
- **2004–2008:** Apprentice in Computer Science, SITEL, Fribourg, CH, Servers, network infrastructure, Java, Java EE, SQL Server.

School projects

- **Master Thesis**, Cache-Friendly Profile-Guided Optimization, Lawrence Berkeley National Laboratory, Berkeley, CA, Sampling-based Profile-Guided optimization pass and loop fusion pass for GCC
- **Semester Project**, Concurrent Binary Search Trees Implementation for multicore, HES-SO, Development and comparisons of several concurrent binary trees implementations
- **Bachelor Thesis**, Inlining Assistance for Large-Scale OO Applications, Lawrence Berkeley National Laboratory, Berkeley, CA, Static analysis of compiled program to find the most interesting functions to inline

-
- **Semester Project**, Checked Overflow in Java Code, HEIA-FR, Automatically detect arithmetic anomalies in Java code

Technical Experience

- Extremely proficient with
 - *languages*: C, C++, Java
 - *technologies*: L^AT_EX, Bash, ZSH, Git, Vim, Git, Subversion
 - *OSs*: Linux, Gentoo, Ubuntu, Windows
 - *others*: Machine learning, performance benchmarking, profiling
- Have experience with
 - *languages*: Intel Assembly, PHP, C#, Python, Prolog, HTML/CSS/JavaScript
 - *technologies*: Jenkins, Sonar, Visual Studio, Android, Sharepoint
 - *others*: Compiler Theory, GCC, Clang, operating system development

Certifications

- SCJP, Sun Certified Java Programmer, Edition 6, Sun/Oracle, 2010

Languages

- French: Native proficiency
- English: Bilingual proficiency
- German: Elementary proficiency

Personal Projects

- **DLL**: C++ Library for Deep Learning, Restricted Boltzmann Machine (RBM), Deep Belief Network and Convolution RBM.
- **ETL**: Expression Templates library, in C++, for matrix and vector computations. Matrix Multiplication, Convolutions, Fast Fourier Transform, ...
- **Thor**: 64-bit operating system in C++ and assembly, multi-processing, with networking and hard drive capabilities
- **EDDI**: Compiler of custom programming language, in C++, with aggressive optimizations for 64-bit Intel processors.

List of publications

Conference Papers

1. B. Wicht and J. Hennebert, Camera-based Sudoku recognition with Deep Belief Network, In Proceedings of the 6th IEEE International Conference of Soft Computing and Pattern Recognition (SoCPaR), pages 83-88, August, 2014
2. B. Wicht and J. Hennebert, Mixed handwritten and printed digit recognition in Sudoku with Convolutional Deep Belief Network, In Proceedings of the 13th IAPR International Conference on Document Analysis and Recognition (ICDAR), pages 861-865, August, 2015
3. B. Wicht, A. Fischer and J. Hennebert, Keyword Spotting with Convolutional Deep Belief Networks and Dynamic Time Warping, In Proceedings of the International Conference on Artificial Neural Networks (ICANN), pages 113-120, June, 2016
4. N. Howe, A. Fischer and B. Wicht, Inkball Models as Features for Handwriting Recognition, In Proceedings of the International Conference on Frontiers of Handwriting Recognition (ICFHR), October, 2016
5. B. Wicht, A. Fischer and J. Hennebert, Deep Learning Features for Handwritten Keyword Spotting, In Proceedings of the International Conference on Pattern Recognition (ICPR), December, 2016
6. B. Wicht, A. Fischer and J. Hennebert, DLL: A Blazing Fast Deep Neural Network Library, *submitted* to the Proceedings of the International Conference on Pattern Recognition (ICPR), 2018

Workshops

1. B. Wicht, J. Hennebert, Deep Learning Feature Extraction for Natural Text Analysis, In Proceedings of the Doctoral Workshop on Distributed Systems, 2015

-
2. B. Wicht, A. Fischer and J. Hennebert, On CPU Performance Optimization of Restricted Boltzmann Machine and Convolutional RBM, In Proceedings of the IAPR Workshop on Artificial Neural Networks in Pattern Recognition (ANNPR), pages 163-174, September, 2016

