Université de Montréal

Training Deep Convolutional Architectures for Vision

par
Guillaume Desjardins

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Mémoire présenté à la Faculté des arts et des sciences
en vue de l'obtention du grade de Maître ès sciences (M.Sc.)
en informatique

Août, 2009

© Guillaume Desjardins, 2009.

Université de Montréal
Faculté des arts et des sciences

Ce mémoire intitulé:

Training Deep Convolutional Architectures for Vision

présenté par:

Guillaume Desjardins

a été évalué par un jury composé des personnes suivantes:

Pascal Vincent
président-rapporteur

Yoshua Bengio
directeur de recherche

Sébastien Roy
membre du jury

RÉSUMÉ

Les tâches de vision artificielle telles que la reconnaissance d'objets demeurent irrésolues à ce jour. Les algorithmes d'apprentissage tels que les Réseaux de Neurones Artificiels (RNA), représentent une approche prometteuse permettant d'apprendre des caractéristiques utiles pour ces tâches. Ce processus d'optimisation est néanmoins difficile. Les réseaux profonds à base de Machine de Boltzmann Restreintes (RBM) ont récemment été proposés afin de guider l'extraction de représentations intermédiaires, grâce à un algorithme d'apprentissage non-supervisé. Ce mémoire présente, par l'entremise de trois articles, des contributions à ce domaine de recherche.

Le premier article traite de la RBM convolutionelle. L'usage de champs réceptifs locaux ainsi que le regroupement d'unités cachées en couches partageant les même paramètres, réduit considérablement le nombre de paramètres à apprendre et engendre des détecteurs de caractéristiques locaux et équivariant aux translations. Ceci mène à des modèles ayant une meilleure vraisemblance, comparativement aux RBMs entraînées sur des segments d'images.

Le deuxième article est motivé par des découvertes récentes en neurosciences. Il analyse l'impact d'unités quadratiques sur des tâches de classification visuelles, ainsi que celui d'une nouvelle fonction d'activation. Nous observons que les RNAs à base d'unités quadratiques utilisant la fonction softsign, donnent de meilleures performances de généralisation.

Le dernière article quand à lui, offre une vision critique des algorithmes populaires d'entraînement de RBMs. Nous montrons que l'algorithme de Divergence Contrastive (CD) et la CD Persistente ne sont pas robustes : tous deux nécessitent une surface d'énergie relativement plate afin que leur chaîne négative puisse mixer. La PCD à "poids rapides" contourne ce problème en perturbant légèrement le modèle, cependant, ceci génère des échantillons bruités. L'usage de chaînes tempérées dans la phase négative est une façon robuste d'adresser ces problèmes et mène à de meilleurs modèles génératifs.

Mots clés: réseau de neurone, apprentissage profond, apprentissage non-supervisé, apprentissage supervisé, RBM, modèle à base d'énergie, tempered MCMC

ABSTRACT

High-level vision tasks such as generic object recognition remain out of reach for modern Artificial Intelligence systems. A promising approach involves learning algorithms, such as the Arficial Neural Network (ANN), which automatically learn to extract useful features for the task at hand. For ANNs, this represents a difficult optimization problem however. Deep Belief Networks have thus been proposed as a way to guide the discovery of intermediate representations, through a greedy unsupervised training of stacked Restricted Boltzmann Machines (RBM). The articles presented here-in represent contributions to this field of research.

The first article introduces the convolutional RBM. By mimicking local receptive fields and tying the parameters of hidden units within the same feature map, we considerably reduce the number of parameters to learn and enforce local, shift-equivariant feature detectors. This translates to better likelihood scores, compared to RBMs trained on small image patches.

In the second article, recent discoveries in neuroscience motivate an investigation into the impact of higher-order units on visual classification, along with the evaluation of a novel activation function. We show that ANNs with quadratic units using the softsign activation function offer better generalization error across several tasks.

Finally, the third article gives a critical look at recently proposed RBM training algorithms. We show that Contrastive Divergence (CD) and Persistent CD are brittle in that they require the energy landscape to be smooth in order for their negative chain to mix well. PCD with fast-weights addresses the issue by performing small model perturbations, but may result in spurious samples. We propose using simulated tempering to draw negative samples. This leads to better generative models and increased robustness to various hyperparameters.

Keywords: neural network, deep learning, unsupervised learning, supervised learning, RBM, energy-based model, tempered MCMC

CONTENTS

RÉSUMÉ . . . . **iii**

ABSTRACT . . . . **iv**

CONTENTS . . . . **v**

LIST OF TABLES . . . . **ix**

LIST OF FIGURES . . . . **x**

LIST OF APPENDICES . . . . **xi**

LIST OF ABBREVIATIONS . . . . **xii**

NOTATION . . . . **xiii**

DEDICATION . . . . **xv**

ACKNOWLEDGMENTS . . . . **xvi**

CHAPTER 1: INTRODUCTION **1**

1.1 Introduction to Machine Learning 2

1.1.1 What is Machine Learning 2

1.1.2 Empirical Risk Minimization 4

1.1.3 Parametric vs. Non-Parametric 5

1.1.4 Gradient Descent: a Generic Learning Algorithm 8

1.1.5 Overfitting, Regularization and Model Selection 9

1.2 Logistic Regression: a Probabilistic Linear Classifier 10

1.3 Artificial Neural Networks 13

1.3.1 Architecture 13

1.3.2 The Backpropagation Algorithm 15

1.3.3 Implementation Details 17
1.3.4 Challenges 18
1.4 Convolutional Networks 20
1.5 Alternative Models of Computation 22

CHAPTER 2: DEEP LEARNING 24
2.1 Boltzmann Machine 25
2.2 Markov Chains and Gibbs Sampling 27
2.2.1 Markov Chains 27
2.2.2 The Gibbs Sampler 28
2.3 Deep Belief Networks 29
2.3.1 Restricted Boltzmann Machine 29
2.3.2 Contrastive Divergence 31
2.3.3 Greedy Layer-Wise Training 32

CHAPTER 3: OVERVIEW OF THE FIRST PAPER 34
3.1 Context 34
3.2 Contributions 34
3.3 Comments 35

CHAPTER 4: EMPIRICAL EVALUATION OF CONVOLUTIONAL RBMS FOR VISION 36
4.1 Abstract 36
4.2 Introduction 36
4.3 Restricted Boltzmann Machines 37
4.4 Convolutional RBMs 39
4.4.1 Architecture of CRBMs 39
4.4.2 Contrastive Divergence for CRBMs 41
4.5 Experiments 42
4.6 Results and Discussion 44
4.7 Future work 46

4.8 Conclusion 46

CHAPTER 5: OVERVIEW OF THE SECOND PAPER 49

5.1 Context 49

5.2 Contributions 49

5.3 Comments 50

CHAPTER 6: QUADRATIC POLYNOMIALS LEARN BETTER IMAGE FEATURES 51

6.1 Abstract 51

6.2 Introduction 51

6.3 Model and Variations 55

6.3.1 Learning 56

6.3.2 Sparse Structure and Convolutional Structure 57

6.3.3 Support Vector Machines 59

6.4 Data Sets 59

6.4.1 Shape Classification 59

6.4.2 Flickr 60

6.4.3 Digit Classification 61

6.5 Results 61

6.5.1 Translation Invariance 62

6.5.2 Tanh vs. Softsign 63

6.6 Discussion 64

CHAPTER 7: OVERVIEW OF THE THIRD PAPER 67

7.1 Context 67

7.2 Contributions 68

7.3 Comments 69

CHAPTER 8: TEMPERED MARKOV CHAIN MONTE CARLO FOR TRAINING OF RESTRICTED BOLTZMANN MACHINES

. . . . **70**
8.1 Abstract 70
8.2 Introduction and Motivation 70
8.3 RBM Log-Likelihood Gradient and Contrastive Divergence 72
8.4 Tempered MCMC 73
8.5 Experimental Observations 75
8.5.1 CD-k: local learning ? 75
8.5.2 Limitations of PCD 77
8.5.3 Tempered MCMC for training and sampling RBMs 78
8.5.4 Tempered MCMC vs. CD and PCD 79
8.5.5 Estimating Likelihood on MNIST 81
8.6 Conclusion 83

CHAPTER 9: CONCLUSION **85**
9.1 Article Summaries and Discussion 85
9.1.1 Empirical Evaluation of Convolutional Architectures for Vision 85
9.1.2 Quadratic Polynomials Learn Better Image Features 86
9.1.3 Tempered Markov Chain Monte Carlo for Training of Restricted Boltzmann Machines 87
9.1.4 Discussion 87
9.2 Future Directions 88

BIBLIOGRAPHY **90**

LIST OF TABLES

4.1 Overview of CRBM Architectures Tested 45

6.1 Generalization Error 62
6.2 Results on Translation Invariance 63
6.3 Generalization Error: tanh vs. softsign 64

8.1 Quantitative Comparison of Generated Samples 83

LIST OF FIGURES

1.1 Linear Classifier 11

1.2 Sigmoid Function and Logistic Regression 13

1.3 Graphical Representation of Logistic Regression and a Multi-Layer Perceptron 14

1.4 Convolutional Neural Network 21

2.1 Boltzmann Machine and Restricted Boltzmann Machine 30

2.2 Deep Belief Network 32

4.1 Contrastive Divergence in convolutional RBMs 41

4.2 Example Training Data 43

4.3 Performance of RBM (dense, patch) vs. CRBMs 47

6.1 Sigmoid vs. Softsign 54

6.2 Architecture of Sparse and Convolutional Quadratic Networks 58

6.3 Samples from MNIST, Shapeset and Flickr Datasets 60

8.1 Samples Obtained with CD 76

8.2 Samples Obtained with PCD 78

8.3 Samples Obtained with PCD with fast weights 78

8.4 Samples Obtained with PCD 79

8.5 Results on Log-Likelihood Measurements 80

LIST OF APPENDICES

Appendix I: **Algorithms for Learning CRBMs** . . . . **xvii**

LIST OF ABBREVIATIONS

2D	Two Dimensional
AI	Artificial Intelligence
ANN	Artificial Neural Network
BM	Boltzmann Machine
CD	Contrastive Divergence
CNN	Convolutional Neural Network
CPU	Central Processing Unit
CRBM	Convolutional Restricted Boltzmann Machine
DBN	Deep Belief Network
FPCD	Persistent Contrastive Divergence with fast-weights
HOTLU	Higher-Order Threshold Logic Unit
I.I.D	Independent and Identically-Distributed
MCMC	Markov Chain Monte Carlo
ML	Machine Learning
MLP	Multi-Layer Perceptron
MSE	Mean-Squared Error
NLL	Negative Log-Likelihood
PCD	Persistent Contrastive Divergence
RBF	Radial Basis Function
RBM	Restricted Boltzmann Machine
SIFT	Scale Invariant Feature Transform
SVM	Support Vector Machines
XOR	Exclusive OR

NOTATION

$\mathscr{D}$	Dataset containing training examples and possibly labels/targets
$\mathscr{D}_{\text{valid}}$	Dataset containing examples $\notin \mathscr{D}$: used to choose optimal hyperparameters θ_H
$\mathscr{D}_{\text{test}}$	Dataset containing examples $\notin \mathscr{D}, \mathscr{D}_{\text{valid}}$: used to estimate generalization error
d_h	number of neurons in hidden layer
$E_{p(x)}$	Expectation with regards to probability density $p(x)$
f	Prediction function
f_θ	Function f parametrized by parameters θ
$g(x)$	discriminant function
$h^{(a)}, h$	vector of hidden units in MLP, before and after activation function
$\mathbf{I}_x$	indicator function (1 if x is true, 0 otherwise)
$K(x,y)$	Kernel function applied to points x and y
$KL(p_1, p_2)$	KL-divergence of probability densities p_1 and p_2
$\mathscr{L}$	Loss function used during minimization
L_β	size of mini-batch
m	Label predicted by prediction function f
$\mathscr{N}_{\mu,\sigma}$	Gaussian probability distribution with mean μ and covariance σ
$o^{(a)}, o$	vector of output units in MLP, before and after activation function
$p(x)$	Model estimate of the true empirical distribution $p_T(x)$
$p(x,y)$	Model estimate of the true joint-probability density function $p_T(x,y)$
$p(y\|x)$	Model estimate of the true conditional probability density function $p_T(y\|x)$
ϕ	Non-linear function
$\mathscr{R}(f, \mathscr{D})$	Empirical risk of prediction function f measured over dataset $\mathscr{D}$
$\mathbb{R}^d$	Set of all real numbers with dimensionality d

s	(non-linear) squashing function
θ	Set of parameters to learn
θ_H	Set of all hyperparameters
W	upper-case indicates a matrix, unless specified otherwise
w	lower-case indicates a vector, unless specified otherwise
W^T	transpose of matrix or vector
$W_{i\cdot}$	i-th row of matrix W
$W_{\cdot j}$	j-th column of matrix W
W_{ij}	element in i-th row and j-th column of matrix W
w_i	i-th element of vector w
$x^{(i)}$	i-th training example, $x^{(i)} \in \mathbb{R}^d$
$x \sim p(x)$	x is a sample drawn from probability density $p(x)$
$y^{(i)}$	Label or target for i-th training example
$z^{(i)}$	In supervised learning, $(x^{(i)}, y^{(i)})$, in unsupervised learning $(x^{(i)})$
Z	Normalization constant
(c, V, b, W)	parameters of MLP: offset vector c and weight matrix V of hidden layer; offset vector b and weight matrix W of output layer
$\left.\frac{\partial f}{\partial x}\right\|_{x=a}$	derivative of function f with respect to x, evaluated at $x = a$

À tous ceux qui m’ont encouragé à poursuivre dans cette nouvelle direction.
Sans vous, rien de ceci n’aurait été possible.

ACKNOWLEDGMENTS

First and foremost, I would like to thank Yoshua Bengio for taking me under his wing and giving me the tools to one-day (hopefully) succeed in research. His vast knowledge of machine learning is only trumped by his passion for research and understanding, a trait which is often contagious. By fostering creativity, independent thinking and open discussions, Yoshua has created a truly motivating and rewarding atmosphere within the lab, in which it is a pleasure to be a part of. I would also like to thank Aaron Courville for many thought provoking (and sometimes long) discussions, which have helped mold my current understanding of machine learning. Special thanks also go to James Bergstra and Olivier Breuleux not only for their work on Theano, but also for always making themselves available to help. Along the same lines, thanks to Frédéric Bastien, who despite being pulled in many directions, always finds time to help out.

Last but not least, thank you to my friends and family for their encouragements. Madeleine, thank you for putting up with the late hours, the ups and downs of research and for always being there for me (even when far away). This would not have been possible without you.

CHAPTER 1

INTRODUCTION

It is reported that Marvin Minsky, one of the founding fathers of Artificial Intelligence (AI), once assigned as a summer project, the task of building an artificial vision system capable of describing what it saw. Several decades later, visual object recognition still remains a largely unsolved problem. At the time of this writing, state-of-the-art results on Caltech 101 (a benchmark dataset containing objects of 101 categories to identify in natural images) hover around 65% accuracy [31]. So what makes object recognition and artificial perception so difficult ?

Real-world images are the result of complex interactions between lighting, scene geometry, textures and an observer (i.e. the human eye or a camera). Small variations at any stage of this image formation process can have profound effects on the resulting 2D image. For example, two identical pictures taken at different times of day may look more dissimilar (in terms of average euclidean distance between pixels) than pictures of different scenes taken in the same lighting conditions. Changes in viewpoint can also contribute to making a single object unrecognizable once rotated, if using a simple template matching approach[1]. The difficulty of generic object recognition is further compounded by the fact that two objects belonging to the same object category, may be more visually dissimilar, than objects of a competing class. Building a robust computer vision system therefore involves building a system which is invariant to many (if not all) of the afore-mentioned sources of variation.

Object recognition systems often use a two-stage pipeline to solve this problem. The first stage involves extracting a set of features from the input data, which is then used as input to a classification module. These features are often hand-crafted to be invariant to certain forms of variations. For example, SIFT features [43] have been shown to be robust to scale, lighting and small amounts of rotation. While these features are still

[1] Template matching consists in convolving the input with a prototypical image of the object of interest (or its sub-parts). The output of the convolution should be maximal at the object's location.

competitive on Caltech-101, their development requires extensive engineering. Also, it is not clear how one would engineer features to be robust across higher-level abstractions ("animal species" for example). It would be ideal if such features could be automatically learnt from training data. This would allow features to be tuned automatically in order to maximize the performance of the system, while also requiring the development of a single algorithm which would work across many settings. To this end, we turn to the field of Machine Learning, which has shown, since the inception of the Artificial Neural Network, that this is indeed an achievable goal.

In this chapter, we start by giving a brief overview of Machine Learning and explain the core principles behind the most common learning algorithms. In section 1.3, we explain in detail the Artificial Neural Network (ANN) and show how it can automatically perform feature extraction and classification. Sections 1.4 and 1.5 explore biologically motivated variants of ANNs which will be the focus of later chapters. We then build on this knowledge and explore in Chapter 2, recent developments in the field of neural network research : the Deep Belief Network, which embodies the principles of Deep Learning. Chapters 3-8 represent the core of this thesis and consist of three articles pertaining to the field of deep networks and ANNs applied to vision.

1.1 Introduction to Machine Learning

1.1.1 What is Machine Learning

Machine Learning (ML) is a sub-field of AI, which focuses on the statistical nature of learning. The goal of ML is to develop algorithms which learn directly from data by exploiting the statistical regularities present in the signal. Intelligence or intelligent behaviour, is thus regarded as the ability to apply this knowledge to novel situations. This concept is known as **generalization**. A learning algorithm can thus be described as any algorithm which takes as input a **training set** $\mathscr{D}$ and outputs a model or prediction function f. The quality of this learnt model is then determined by the accuracy of the prediction on a separate hold-out dataset known as the **test set** $\mathscr{D}_{\text{test}}$. A model which performs well on the training data $\mathscr{D}$ but poorly on the test data is said to be **overfitting**.

The exact nature of the datasets varies depending on the intended applications. In the context of supervised learning, the goal is to learn a mapping between a series of observations and associated targets. The training set can be written as $\mathcal{D} = \{(x^{(i)}, y^{(i)}); i = 1..n\}$ where $x^{(i)} \in \mathbb{R}^d$ is an input datum with target $y^{(i)}$. We will define $z^{(i)}$ as being the pair $(x^{(i)}, y^{(i)})$ and consider $z^{(i)}$ to be independent and identically distributed (IID) samples from the true underlying distribution $p_T(z)$. The target $y^{(i)}$ may be discrete or continuous. The discrete case corresponds to a **classification** task, where $y^{(i)} \in \{1, ..., m\}$ is one of m possible **categories** or **labels** to assign to input $x^{(i)}$. In object recognition, the $x^{(i)}$'s would correspond to the input images and the $y^{(i)}$'s to a numerical value indicating the type of object present within the image. From a probabilistic point of view, the general concept is to learn an estimate $p(y|x)$ of $p_T(y|x)$ directly from the training data $\{(x^{(i)}, y^{(i)})\}$. $p(y|x)$ is vector-valued and contains the class membership probabilities $p(y_j|x), j \in \{1, ..., m\}$ for all possible classes of input x. The predicted class is then given by $f(x) = \text{argmax}_j\, p(y_j|x)$. The resulting module is called a **classifier** and is a central building block of many object recognition systems. If the target y is continuous-valued, the problem is one of **regression**. The goal is then to generate an output so as to match a given statistic of $p_T(y|x)$, for example $E_{p_T(x,y)}(Y|X)$. Predicting the position of the object within the input images (as opposed to the nature of the object) would constitute a regression task.

When no target y is given, learning is said to be **unsupervised** and $z^{(i)} = x^{(i)}$. The learner then simply tries to model the input distribution $p_T(x)$, or aspects thereof. Probabilistic modeling of $p_T(y|x)$ is often referred to as **density estimation**, which strictly speaking, assumes that x is continuous-valued. Unsupervised learning is often used for exploratory data analysis, in order to gain a better understanding of the data. Algorithms such as k-means or mixtures of Gaussians for example, can be used to extract the most salient modes of a distribution and help to extract natural groupings within the data, a task known as **clustering**. One can also augment the model $p(x)$ by adding **hidden** or **latent** variables **h** to the model. $p(x)$ can then be rewritten as $p(x) = \sum_h p(x, h)$. The state of the hidden units being unknown, they must therefore be inferred by finding the most probable values for h. This task is known as **inference** and can be used to find "root

causes" or "explanations" for the visible data. As we will see later on in section 2.3.1, the Restricted Boltzmann Machine (RBM) works along this principle and is used to extract useful features from the data. Unsupervised learning can also be used to build **generative** models, where the trained system can output samples $\tilde{x} \sim p(x)$ which mimic the training distribution. A hybrid method also consists in using unsupervised learning to learn a model $p(x,y)$ of the joint distribution, which in turn can be used for classification.

Other variants of ML include semi-supervised learning where $\mathscr{D}$ consists of both labeled and unlabeled examples. Reinforcement learning deals with the problem of delayed reward, where the supervised signal (or reinforcement) is provided only after a series of actions have been taken and depends on the actions taken along the way. We are intentionally leaving out discussion of these sub-fields of ML as they are not directly relevant to the contents of this thesis.

1.1.2 Empirical Risk Minimization

While we have given a general definition of what a learning algorithm should look like, we still have not specified how the actual learning occurs. How do we actually obtain this function f from $\mathscr{D}$? Most ML algorithms utilize the **empirical risk minimization** strategy.

Given a **loss function** $\mathscr{L}$ and a dataset $\mathscr{D}$, the empirical risk is defined as:

$$\mathscr{R}(f,\mathscr{D}) = 1/n \sum_{i=1}^{n} \mathscr{L}(x^{(i)}, y^{(i)}; f) \tag{1.1}$$

Learning consists in finding the function or model f which minimizes the average loss across the training set. Learning should therefore return the function f such that:

$$f \leftarrow argmin_{f^*} \mathscr{R}(f^*, \mathscr{D})$$

This recipe for learning is problematic however. Indeed, there is an easy and trivial solution to this minimization process. The model can simply learn the training set by heart (i.e. by making a copy of the data in memory) and for each $x^{(i)}$ output the associated

$y^{(i)}$. Such a model would obtain the lowest value for $\mathscr{R}$. It would be absurd however, to claim that such a system has learnt anything useful about the data or to relate this model to "intelligent behaviour" of any kind. As mentioned previously, what we are ultimately interested in is the generalization capability of our model. To evaluate the true performance of a model, we therefore use $\mathscr{R}(f, \mathscr{D}_{\text{test}})$, the average risk across test set $\mathscr{D}_{\text{test}} = \{z_i; z_i \notin \mathscr{D}\}$.

The choice of loss function will vary depending on the application. Generally speaking however, the following loss functions are used:

- Classification: classification error. $\mathscr{L}_{classif.}(x^{(i)}, y^{(i)}; f) = \mathbf{I}_{f(x^{(i)})!=y^{(i)}}$, i.e. a unity loss whenever the predicted class label is not the correct one. For reasons we will explore in section 1.1.4, it can be advantageous for the loss function to be continuous and differentiable. In that case, probabilistic classifiers may use the conditional likelihood loss (section 1.2).

- Regression: mean-squared error loss. $\mathscr{L}_{MSE}(x^{(i)}, y^{(i)}; f) = (f(x^{(i)}) - y^{(i)})^2$. The empirical risk will thus be the average squared error between predicted values and the real targets.

- Density Estimation: negative-likelihood loss. $\mathscr{L}_{NLL}(x^{(i)}; f) = -\log f(x^{(i)})$, where $f(x)$ is the estimate $p(x)$ of the underlying distribution $p_T(x)$. In the case of parametric models indexed by parameters θ (section 1.1.3), this leads to the solution which maximizes $p(\mathscr{D}|\theta)$, an instance of the **maximum likelihood** solution.

1.1.3 Parametric vs. Non-Parametric

Machine learning algorithms can generally be split into two families: parametric and non-parametric methods.

Parametric algorithms are those which model a particular probability distribution, using a fixed set of parameters θ. The function or model is written as f_θ. In this setting, learning amounts to finding the optimal parameters θ so as to minimize $\mathscr{R}(f_\theta, \mathscr{D})$. The number of free parameters in θ, determines the **modeling capacity** of f and controls how

well one can approximate the distribution of interest. A typical parametric method for density estimation is the mixture of Gaussians model, which estimates $p(x)$ as the sum of multiple Gaussian probability distributions such that $p(x) = \sum_{k=1}^{K} \pi_k \mathcal{N}(x|\mu_k, \Sigma_k)$, where $\mathcal{N}(x|\mu_k, \Sigma_k)$ is a Gaussian distribution with mean μ_k and covariance matrix Σ_k. K is the total number of Gaussians used by the model and π_k is the weight associated to each Gaussian (with the constraint $\sum_{k=1}^{K} \pi_k = 1$). The optimal parameters $\theta = \{(\mu_k, \Sigma_k, \pi_k), k = 1..K\}$ can be determined through maximum likelihood by solving $\partial \mathcal{R} / \partial \theta = 0$. The main drawback of parametric methods is the constraints imposed by the choice of a fixed model. Choosing an improper model (e.g. a single Gaussian to model a multi-modal distribution) can result in poor performance. Parametric models of relevance to this thesis include logistic regression (section 1.2), artificial neural networks (section 1.3) and Deep Belief Networks (DBN) (section 2.3).

Non-parametric methods are more flexible in that they make no inherent assumptions about the distributions to model. A large family of non-parametric algorithms use the training data itself to model the distribution. The most basic non-parametric algorithm is the well-known histogram method. For input data $x^{(i)} \in \mathbb{R}^d$, the input data space is divided into k^d equally-sized bins. The probability density can then be estimated locally, within each bin as

$$p_i = \frac{n_i}{n\Delta_i},$$

where n_i is the number of data points falling within bin-i, n the total number of points in $\mathcal{D}$ and Δ_i the width of the bin. Parzen Windows is another hallmark non-parametric density estimation method which greatly improves on the histogram method. Instead of partitioning the space into equally sized bins and assigning probability mass in a discrete manner, each data point $x^{(i)} \in \mathcal{D}$ contributes an amount $1/nK(x^{(i)}, x)$ to $p(x)$, where $K(x^{(i)}, x)$ is a smooth kernel centered on $x^{(i)}$. The density estimate is therefore: $p(x) = 1/n \sum_{i=1}^{n} K(x^{(i)}, x)$, A popular choice for K is the multi-variate Gaussian density function with mean $\mu = x^{(i)}$. The variance σ^2 of this Gaussian kernel is referred to as a **hyperparameter** (and not a parameter) since it cannot be learnt by maximum likelihood on the training data. Indeed, learning the variance by minimizing $\mathcal{R}$ would result in a

null variance, possibly leading to an infinite loss on the test set. In the future, we will denote the set of hyperparameters required by a model f as θ_H. We will see later on in Chapter 8, a real-world usage scenario of how Parzen Windows can be used to estimate the density $p(x)$ defined by a Restricted Boltzmann Machine.

Parametric algorithms may also contain hyperparameters. Typical examples include learning rates (section 1.1.4), stopping criteria (sections 1.1.4, 1.1.5) and regularization constants (section 1.1.5). In practice, the distinction between parametric and non-parametric methods is also often blurry. For example, Artificial Neural Networks (ANN), Deep Belief Networks (DBN) and mixture models can be considered non-parametric if the number of hidden units or components in the mixture is parameterizable. A method for choosing good hyperparameter values is covered in section 1.1.5.

Many non-parametric methods are said to be **local methods**. This is the case when their prediction for test point $x^{(j)}$ depends on training data at a relatively short distance from $x^{(j)}$ (whether for classification, regression or density estimation). Local methods are thus much more prone to the **curse of dimensionality**, which states that the amount of data (cardinality of $\mathscr{D}$) required to span an input space $\mathbb{R}^d$ is exponential in the number of dimensions d. As such, the performance of these algorithms degrades significantly for higher values of d, unless compensated by an exponential increase in training data.

For vision applications, we are usually dealing with inputs of very large dimensionality. In the simplest case of hand-written digit recognition, such as the MNIST dataset [39], input images are of size 28x28 pixels and can easily scale up to hundreds of thousands of pixels for more complicated datasets. While the true dimensionality of the data (i.e the dimension of the manifold on which the training distribution is concentrated) is usually unknown and no doubt smaller than the raw number of pixels, the inherent complexity of vision problems suggests that this phenomenon is definitely at play. For this reason, this thesis will focus on **global methods**, which are not as prone to the curse of dimensionality and are much better suited to the problem at hand.

1.1.4 Gradient Descent: a Generic Learning Algorithm

Gradient Descent is a well-known first-order optimization technique for finding a minimum of a given function. For any multivariate function $f(x)$ (with $x = [x_1, ..., x_d]$) differentiable at point a, the direction of steepest ascent is given by the vector of partial derivatives, $[\frac{\partial f}{\partial x_1}\Big|_{x=a}, ..., \frac{\partial f}{\partial x_d}\Big|_{x=a}]$ at point a. If we initialize $x_0 = a$ and $||\frac{\partial f(a)}{\partial x}|| > 0$, performing an infinitesimal step in the opposite direction of this gradient is guaranteed to achieve a lower value for $f(x)$. This suggests the iterative algorithm of Algorithm 1 for minimizing the empirical risk $\mathscr{R}$.

Algorithm 1 `BatchGradDescentLearning`$(\mathscr{L}, f_\theta, \mathscr{D}, \varepsilon)$
L: loss function to minimize across training set $\mathscr{D}$
f_θ: prediction function parameterized by parameters θ
$\mathscr{D}$: dataset of training examples
ε: learning rate or step-size for gradient descent

Initialize model parameters of f_θ to $\tilde{\theta}$
while stopping condition is not met **do**
  Initialize $\frac{\partial \mathscr{R}}{\partial \theta}$ to 0
  for all $z^{(i)} \in \mathscr{D}$ **do**
    $\frac{\partial \mathscr{R}}{\partial \theta} \leftarrow \frac{\partial \mathscr{R}}{\partial \theta} + \frac{\partial \mathscr{L}(z^{(i)}, f_\theta)}{\partial \theta}|_{\theta=\tilde{\theta}}$
  end for
  $\tilde{\theta} \leftarrow \tilde{\theta} - \varepsilon \frac{\partial \mathscr{R}}{\partial \theta}$
  Update model parameters of f_θ to $\tilde{\theta}$
end while
return f_θ

The above procedure is known as a **batch learning** method, since it requires a complete pass through the training set $\mathscr{D}$ before performing a parameter update. In practice, this procedure is guaranteed to converge as long as certain conditions on the learning rate are satisfied[2]. Furthermore, if the loss function is convex in θ, it will converge to the global minimum. Gradient descent of non-convex functions may lead to local minima however.

An alternative learning algorithm, known as **stochastic gradient descent** has proven

[2]To guarantee convergence, ε_t must actually have a decreasing profile as a function of t, such that $lim_{t\to\infty} \sum_t \varepsilon_t = \infty$ and $lim_{t\to\infty} \sum_t \varepsilon_t^2 < \infty$.

very efficient in practice [39]. The trick is to get rid of the inner-most loop and update the parameters for each training example $z^{(i)}$. While this update does not follow the exact gradient of $\mathscr{R}(f,\mathscr{D})$, the redundancy in the training data (examples are IID) tends to make this algorithm converge much faster. The randomness introduced by the stochastic updates has also been shown to help with escaping from local minima. A hybrid method also exists, named **stochastic gradient with mini-batches**, which consists in updating the parameters every L_β training examples, where L_β is usually in the range $1 < L_\beta < 100$. This has several advantages, the first of which is to help reduce the variance of the gradient updates.[3]

How to initialize the parameters and when to stop the gradient descent procedure usually vary based on the application. In section 1.3, we will see how this is done in the case of Artificial Neural Networks.

1.1.5 Overfitting, Regularization and Model Selection

With all ML algorithms, great care must be taken so as not to overfit the training data. Non-parametric algorithms are especially prone to this problem. By simply memorizing the training data, they can achieve perfect prediction during training. In the case of Parzen Windows density estimation, we have already seen that this can lead to an infinite error on the test set if $\sigma = 0$, i.e. the worst generalization scenario possible. To select the optimal hyperparameters, one needs a separate hold-out set called the **validation set** which objectively measures the effect of the hyperparameters. A grid search can be used to measure $\mathscr{R}(f,\mathscr{D}_{\text{valid}})$ for various values of θ_H and the hyperparameters θ_H^* are chosen in order to minimize this empirical risk. Generalization error is then estimated as usual using hyperparameters θ_H^* for the model f.

Parametric models can also overfit training data if they have too much modeling capacity. For example, consider a dataset with N training examples and and a bijective function $\phi(x) : \mathbb{R}^d \rightarrow \mathbb{R}^n$. Any linear classifier with $N+1$ degrees of freedom can achieve zero classification error on the transformed dataset $\phi(\mathscr{D})$ (which will not necessarily

[3] For L_β chosen appropriately, the use of mini-batches can also help to speed up computations, by minimizing total memory accesses and maximizing cache usage within the CPU.

translate to better generalization). On the other hand, if modeling capacity is too small, the model f will exhibit poor performance both during training and testing. A good compromise is then to select a complex model and artificially control its capacity using a technique known as **regularization**.

Regularization involves adding a penalty term to the loss function $\mathscr{L}$ in order to discourage the parameters θ from reaching large values [6]. This modified loss function can be written as:

$$\mathscr{L}(f_\theta, \mathscr{D}) = \mathscr{L}_D(f_\theta, \mathscr{D}) + \lambda \mathscr{L}_\theta(\theta) \tag{1.2}$$

The indices in the terms of $\mathscr{L}_D$ and $\mathscr{L}_\theta$ are meant to differentiate the data-dependent loss from the regularization loss incurred by θ. The coefficient λ is a hyperparameter which controls the amount of regularization. The exact choice for $\mathscr{L}_\theta$ depends on the application. However, popular choices are **L2** and **L1 regularization** which penalize the L2 and L1 norm of each parameter in θ.

From here on in, we will use $\mathscr{L}$ to refer to the loss with or without regularization, depending on context. We shall also use the term **cross-validation** to refer to the use of a validation set for optimizing hyperparameters and **model selection** for the full training procedure (optimizing $\{\theta, \theta_H\}$).

1.2 Logistic Regression: a Probabilistic Linear Classifier

Linear classifiers are the simplest form of parametric models for binary classification. They split the input space into two subsets (corresponding to classes $\mathbf{C}_1$ and $\mathbf{C}_2$) using a linear **decision boundary** with equation:

$$g(x) = \mathbf{w}' \cdot x + b = \sum_j w_j x_j + b = 0 \tag{1.3}$$

$g(x)$ is referred to as the **discriminant function**. The **weights w** and input x are both column vectors in $\mathbb{R}^d$. The w_j's control the slope of the decision boundary, while the **offset** $b \in \mathbb{R}$ determines the exact position of this separating hyperplane. For a given set

of parameters $\theta = (\mathbf{w}, b)$, the output of the classifier is determined by which side of the decision boundary the test point falls in, as shown in Fig. 1.1. Formally, we define the **decision function** f as $f(x) = \begin{cases} \mathbf{C}_2 \text{ if } g(x) \geq 0 \\ \mathbf{C}_1 \text{ if } g(x) < 0 \end{cases}$.

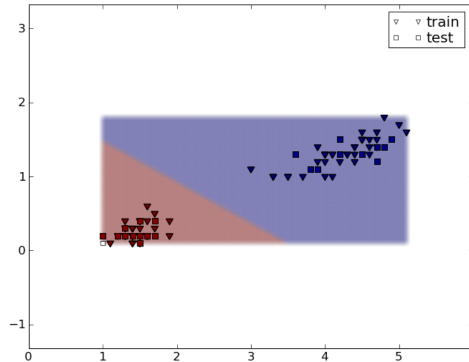


Figure 1.1: Linear classifier on the 2 first components of the Iris flower dataset [14]. Blue area corresponds to class $\mathbf{C}_1$ and red to class $\mathbf{C}_2$

Learning such a linear classifier amounts to finding the optimal parameters θ^*, so as to minimize the empirical risk using $\mathscr{L}_{classif.}$ as the loss function. Unfortunately, due to the step-wise nature of the decision function, this cost function is not smooth with respect to θ and as such, does not lend itself to gradient descent.

This problem can be easily overcome by using a smooth decision function. Geometrically, the value $g(x)$ represents the distance from point x to the decision boundary $g(x) = 0$. It can therefore be interpreted as encoding a "degree of belief" about the classification. The further a point x is from the boundary, the more confident the classifier is in its prediction. One possible solution, is therefore to modify the loss function to be null when the prediction is correct and proportional to $g(x)$ when not. This leads to the famous **perceptron** update rule, with loss function:

$$\mathscr{L}_{perceptron}(x^{(i)}, y^{(i)}; f, g) = -y^{(i)} g(x^{(i)}) \mathbb{I}_{f(x^{(i)}) != y^{(i)}} \tag{1.4}$$

By using a squashing function $s(g(x))$, such that $s : [-\infty, +\infty] \rightarrow [0, 1]$, our "degree of belief" can be interpreted as a probability. This leads to a probabilistic classifier called

logistic regression, which learns to predict $p(y=1|x)$ and is defined by Eqs. 1.5-1.6.

$$g(x) = \text{sigmoid}(w'x+b) \tag{1.5}$$

$$f(x) = \begin{cases} 1 \quad (\text{class } \mathbf{C}_2) & \text{if g(x) >= 0.5} \\ 0 \quad (\text{class } \mathbf{C}_1) & \text{if g(x) < 0.5} \end{cases} \tag{1.6}$$

The sigmoid function is defined as $\text{sigmoid}(x) = 1/(1+e^{-x})$. As shown in Figure 1.2(a), it is a monotonically increasing function, constrained to the unit interval. Around $x=0$, its behaviour is fairly linear, however its non-linearity becomes more pronounced as $|x|$ increases. By using the conditional entropy loss function of Eq. 1.7, the loss incurred by each data point is made proportional to the log-probability mass assigned to the incorrect class.

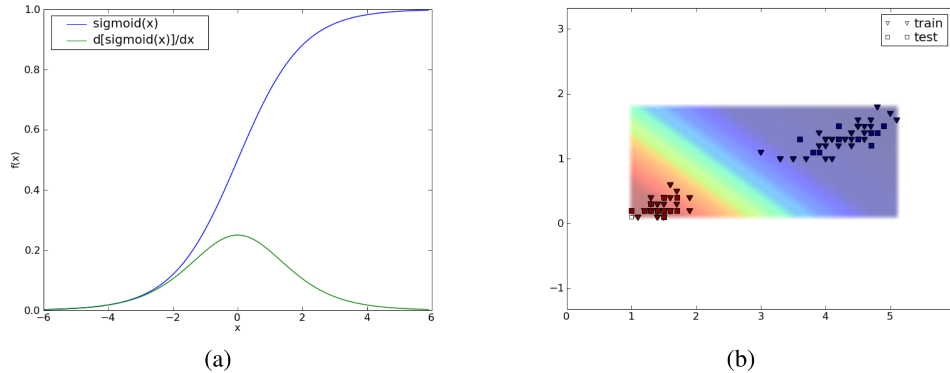
$$\mathscr{L}_{log.reg.}(x^{(i)}, y^{(i)}; g) = -y^{(i)} \log(g(x^{(i)})) - (1-y^{(i)}) \log(1-g(x^{(i)})) \tag{1.7}$$

Figure 1.2(b) shows the classification probability $p(y=1|x)$ for the Iris dataset. The color gradient going from bright red to dark blue corresponds to the interval of $p(y=1|x) \in [0,1]$. We can see that the classifier has maximal uncertainty around the decision boundary.

To obtain the parameter update rules for logistic regression, we can simply perform gradient descent on $\mathscr{R}$, since $\mathscr{L}_{log.reg.}$ is differentiable and smooth with respect to θ. We obtain the following gradients on $\mathbf{w}$ and b:

$$\frac{\partial \mathscr{R}}{\partial w_k} = \frac{1}{n} \sum_{i=1}^{n} [g(x^{(i)}) - y^{(i)}] x_k \tag{1.8}$$

$$\frac{\partial \mathscr{R}}{\partial b} = \frac{1}{n} \sum_{i=1}^{n} [g(x^{(i)}) - y^{(i)}] \tag{1.9}$$



(a) (b)

Figure 1.2: (a) Sigmoid logistic function and its first derivative (b) Classification probability p(y=1|x) of the logistic regression classifier, using the first two components of the Iris dataset as inputs.

1.3 Artificial Neural Networks

Artificial Neural Networks (ANN) are a natural extension of logistic regression to non-linearly separable data. For logistic regression to cope with the non-linear case, a preprocessing stage must first transform $\mathscr{D}$ such that the resulting dataset, $\mathscr{D}'$, becomes linearly separable. A standard linear classifier can then be used to separate $\mathscr{D}'$. The equation for this **generalized linear classifier** is given by:

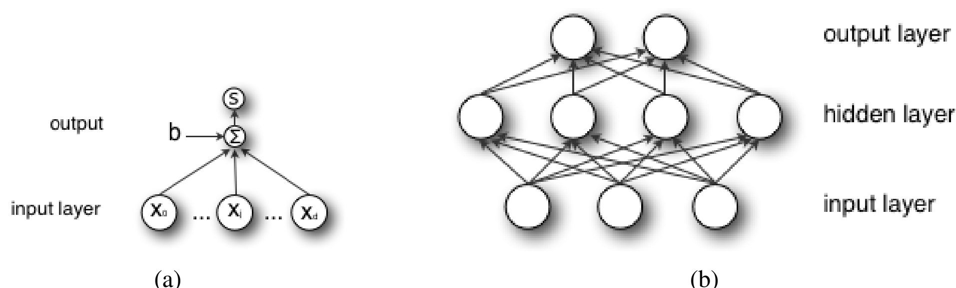
$$g(x) = \text{sigmoid}(\sum_j w_j \phi_j(x) + b), \tag{1.10}$$

where ϕ is a set of non-linear basis functions. The exact nature of ϕ obviously depends on the dataset, as such it would preferable to also learn this transformation from the data. We will see in the following sections that ANNs provide the mechanism for doing exactly this, by parameterizing the non-linear functions ϕ as a composition of logistic classifiers.

1.3.1 Architecture

ANNs are feed-forward probabilistic models which can be used both for regression and classification. The simplest form of ANN, the **Multi-Layer Perceptron** is shown

in Fig. 1.3(b) (for the simplest case of one hidden layer). Comparing this figure to the graphical depiction of logistic regression (Fig. 1.3(a)), it is clear that an MLP is simply a generalized linear classifier, where the pre-processing functions ϕ are themselves of the form $\phi_j(x) = \text{sigmoid}(W'_{j \cdot} x + b_j)$. We will refer to the outputs of the first-layer of logistic regressors as the **hidden units**, which together form the **hidden layer**. The **output layer** refers to the output of the last stage of logistic regression. In Figure 1.3(b), we have simplified the notation by merging the summation and non-linearities into a single entity, as well as omitting the contribution of the offsets b_j. From looking at this figure, it is also clear where the name "Artificial Neural Network" comes from. Much like the basic unit of the ANN, the biological neuron pools together a large number of inputs through its dendritic tree, performs a non-linear processing of its inputs (as in early integrate-and-fire models) generating an output on its single axon (through action potentials). In both cases, these units are organized into complex networks.



(a) (b)

Figure 1.3: (a) Graphical Representation of Logistic Regression. Directed connections from x_j to the summation node represent the weighted contributions $w_j x_j$. s represents the sigmoid activation function. (b) Multi-Layer Perceptron. Each unit in the hidden layer represents a logistic regression classifier. The hidden layer then forms the input to another stage of logistic classifiers.

Formally, a one-hidden layer MLP constitutes a function $f : \mathbb{R}^d \rightarrow \mathbb{R}^m$, such that:

$$f(x) = G(b + W(s(c + Vx))), \tag{1.11}$$

with vectors b and c, matrices W and V and activation functions G and s (typically fixed non-linearities like the sigmoid).

While the above formulation holds true for ANNs with only a single hidden layer, extending it to the multi-layer case is fairly straightforward.

In terms of notation, the hidden units h are obtained as $h(x) = s(c+Vx)$. $V \in \mathbb{R}^{d_h x d}$ is the weight matrix for connections going from the input to the hidden layer. Each row $V_{j\cdot}$ of V contains the weights for the j-th hidden unit, with V_{jk} being the weight from hidden unit h_j to input x_k with $j \in [1, d_h]$ and $k \in [1, d]$. Similarly, output units are obtained as $o(x) = G(b+Wh(x))$. W is the weight matrix connecting the hidden layer to output layer, with W_{ij} the weights connecting output o_i to hidden unit h_j with $i \in [1, m]$. $c \in \mathbb{R}^{d_h}$ and $b \in \mathbb{R}^m$ are the offsets for the hidden and output layers respectively.

For convenience, we will define $o^{(a)}$ and $h^{(a)}$ to be the values of the output and hidden layers before their respective activation functions (i.e. $o = G(o^{(a)}), h = s(h^{(a)})$).

The exact nature of G will depend on the application. For binary classification, a single output unit suffices and G can be the sigmoid activation function. For multi-class classification, G is the softmax activation function, defined as:

$$\text{softmax}_i(x) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

The single-layer MLP is of particular interest because it has been shown to be a **universal approximator** [27]. Given enough hidden units d_h, an MLP can learn to represent any continuous function to some fixed precision, hence capture classification boundaries of arbitrary complexity.

1.3.2 The Backpropagation Algorithm

In this section, we will briefly review the learning algorithm of ANNs. The derivation will be given for the MLP described by Eq. 1.11, with G being the identity function. This same procedure can however be generalized to any number of hidden layers and loss functions.

When the number of hidden units is fixed, MLPs are parametric models where $\theta = [V, c, W, b]$. As such, they can be trained to minimize the empirical risk using gradient descent. The general principle is to iteratively compute the loss $\mathscr{L}$ for a subset of $\mathscr{D}$,

calculate the gradients $\frac{\partial \mathscr{L}}{\partial \theta}$ using the **backpropagation algorithm** [54] and perform one step of gradient descent in an attempt to minimize the empirical risk $\mathscr{R}$ for the subsequent iteration.

Mathematically speaking, backpropagation exploits the chain-rule of derivation. We first start by writing the derivative of the loss function $\mathscr{L}(x^{(i)}, y^{(i)}; f)$ with respect to the output units $o_i^{(a)}$. Recall that $o_i^{(a)}$ is the activation of the units in the output layer, i.e. before the activation function (or non-linearity) has been applied.

$$\frac{\partial \mathscr{L}(x^{(i)}, y^{(i)})}{\partial o_i^{(a)}} = \frac{\partial \mathscr{L}}{\partial o_i} \frac{\partial o_i}{\partial o_i^{(a)}} = \frac{\partial \mathscr{L}}{\partial o_i} \left(\frac{\partial G(\chi)}{\partial \chi} \right) \Bigg|_{\chi = o_i^{(a)}} \equiv \delta_i \tag{1.12}$$

δ_i represents the "error signal" associated with unit o_i, which is back-propagated through the network and used to tune the parameters in the lower layers. Its use is inspired from [20].

From Eq. 1.12, we can easily derive the gradients with respect to parameters $[W, b]$ of the output layer. To simplify notation, we drop the parameters of the function $\mathscr{L}(x^{(i)}, y^{(i)})$ and simply write $\mathscr{L}$.

$$\frac{\partial \mathscr{L}}{\partial b_i} = \frac{\partial \mathscr{L}}{\partial o_i^{(a)}} \frac{\partial o_i^{(a)}}{\partial b_i} = \delta_i \tag{1.13}$$

$$\frac{\partial \mathscr{L}}{\partial W_{ij}} = \frac{\partial \mathscr{L}}{\partial o_i^{(a)}} \frac{\partial o_i^{(a)}}{\partial W_{ij}} = \delta_i h_j \tag{1.14}$$

To derive the gradients on $[V, c]$, we must first backpropagate the error $\frac{\partial \mathscr{L}}{\partial o_i}$, from the output units to the hidden units h_j. From the chain-rule of derivation we can write:

$$\frac{\partial \mathscr{L}}{\partial h_j} = \sum_i \frac{\partial \mathscr{L}}{\partial o_i^{(a)}} \frac{\partial o_i^{(a)}}{\partial h_j} = \sum_i \delta_i W_{ij} \tag{1.15}$$

$$\frac{\partial \mathscr{L}}{\partial h_j^{(a)}} = \frac{\partial \mathscr{L}}{\partial h_j} \frac{\partial h_j}{\partial h_j^{(a)}} = \sum_i [\delta_i W_{ij}] \left(\frac{\partial s(\chi)}{\partial \chi} \right) \Bigg|_{\chi = h_j^{(a)}} \equiv \delta_j \tag{1.16}$$

Again, we set δ_j to represent the error signal fed back from each hidden unit h_j.

Finally, from Eq. 1.16 we can now determine the gradients for the parameters $[V,c]$ of the hidden layer:

$$\frac{\partial \mathcal{L}}{\partial c_j} = \frac{\partial \mathcal{L}}{\partial h_j^{(a)}} \frac{\partial h_j^{(a)}}{\partial c_j} = \delta_j \tag{1.17}$$

$$\frac{\partial \mathcal{R}}{\partial V_{jk}} = \frac{\partial \mathcal{L}}{\partial h_j^{(a)}} \frac{\partial h_j^{(a)}}{\partial V_{jk}} = \delta_j x_k \tag{1.18}$$

1.3.3 Implementation Details

Combining the above equations with the gradient descent algorithm of Algorithm 1 constitutes the batch gradient descent algorithm for one-hidden layer MLPs. Algorithm 2 shows the backpropagation algorithm with stochastic updates. For each training example, we perform a **forward pass** to compute the predicted network output and associated loss (**fprop** in Algorithm 2). This loss is then used as input to the **downward pass** (**bprop** in Algorithm 2) which computes gradients for all parameters of the network. We then perform one step of stochastic gradient descent. An entire pass through the training set is referred to as an **epoch**. The algorithm can run for a fixed number of epochs or use a number of heuristics to decide when to stop (see section 1.3.4.2).

[37] outlines many useful tricks for making the backpropagation algorithm work better. Training patterns x should be normalized [4] and weights initialized to small random values, as a function of the neuron's fan-in. This ensures that units operate in the linear region of the sigmoid at the start of training and are thus provided with a strong learning signal. Targets y should also be chosen according to the type of non-linearity: $\{0,1\}$ for the sigmoid and $\{-1,1\}$ for the hyperbolic tangent *tanh* (an alternative to the sigmoid which is preferable according to [38]). Finally, choosing the appropriate learning rate ε is paramount to the success of this training procedure. In this thesis, we rely on first order gradient descent methods, combined with cross-validation for selecting optimal learning rates.

[4] Decorrelating the inputs x so that all component x_j of x are independent also helps to speedup convergence [37]. However it is not clear this is advisable when working with images, since we may actually want to preserve local correlations.

Algorithm 2 `TrainMLPStochastic`(D,ε)
D_n training set, containing pairs $(x^{(i)}, y^{(i)})$
ε learning rate
$\Delta\theta_{min}$ threshold value used to detect convergence of optimization
fprop: function which computes output $o(x) = f(x)$, according to Eq. 1.11
bprop: function which computes gradients on parameters b, W, c and V according to Eqs. 1.13-1.14 and Eqs. 1.17-1.18 respectively

```
Initialize c, b to zero vectors
Initialize V randomly from uniform distribution w/ range [−1/√d, +1/√d]
Initialize W randomly from uniform distribution w/ range [−1/√d_h, +1/√d_h]
continue ← true
θ^{t−1} ← (W, b, V, c)
while continue do
   Initialize dc, db, dV, dW to zeros
   for all (x^(i), y^(i)) in D_n do
      Get next input x^(i) and target y^(i) from D_n
      o^(i) ← fprop(x^(i))
      dW, db, dV, dc ← bprop(o^(i), y^(i))
      (W, b, V, c) ← (W, b, V, c) − ε · (dW, db, dV, dc)
   end for
   θ^t ← (W, b, V, c)
   if |θ^t − θ^{t−1}| < Δθ_min then
      continue ← false
   end if
   θ^{t−1} ← θ^t
end while
```

1.3.4 Challenges

1.3.4.1 Local Minima

The representational power of ANNs does come at a price. Because of composing several layers of non-linearities, the optimization problem becomes non-convex. There are therefore no guarantees that the resulting solution is a global minimum. As such, when optimizing ANNs, it is customary to run several iterations of the training algorithm from different random initial weights. The performance of the network as a whole can then be reported as the mean and standard deviation of $\mathscr{R}(f, \mathscr{D}_{\text{test}})$. This allows for a fair evaluation of ANN performance and a comparison to other convex learning algorithms

such as Support Vector Machines (SVM) [10]. Alternatively, one can also choose the seed used for random initialization by cross-validation.

As mentioned in section 1.1.4, stochastic gradient descent has been shown to help escape local minima. The idea is akin to Simulated Annealing [32]. By adding randomness or noise to the gradient, we are perturbing the system in such a way as to encourage further exploration of the space. In some cases, this small perturbation will be enough to escape a shallow local minimum.

While the use of gradient descent in a non-convex setting may be objectionable to some, it is worth reminding the reader that finding the global minima is not the ultimate goal of learning. Indeed, the ultimate goal is to achieve good generalization. As such, finding a good local optimum may be sufficient.

1.3.4.2 Overfitting

ANNs being universal approximators, they are very prone to overfitting. The model selection procedure described in section 1.1.5 must therefore be used to carefully select the number of layers and number of units n_h per layer. To control model capacity, ANNs can use an **early-stopping** procedure. By tracking the generalization performance during the training phase (using a validation set), it is possible to greatly reduce the sensitivity of the generalization error to the choice of network size [45]. Networks which have many more parameters than training examples can thus be used if learning is stopped before those networks are fully trained.

By tracking both training and validation errors during learning, it is possible to determine the optimal number of training epochs $\mathbf{e}^*$. During the first $\mathbf{e}^*$ epochs, training and validation errors are minimized concurrently. After $\mathbf{e}^*$ epochs however, validation error starts to increase (while training error is still being minimized).

Early stopping can be understood from the point of view of regularization (section 1.1.5). Since we initialize the weights to small random values, they will tend to increase throughout training. Stopping "early" (before $\mathscr{R}(f_\theta, \mathscr{D})$ is fully minimized) therefore prevents the parameters θ from reaching overly large values. This corresponds to an L2 regularization on the parameters [61].

1.4 Convolutional Networks

From Hubel and Wiesel's early work on the cat's visual cortex [29], we know there exists a complex arrangement of cells within the visual cortex. Cells are tiled in such a way as to cover the entire visual field, with each cell being only sensitive to a small sub-region called a **receptive field**. Two basic cell types were identified, with these very unique properties:

- simple cells (S) respond maximally to specific edge-like stimulus patterns within their receptive field. Their receptive field contains both excitatory and inhibitory regions.
- complex cells (C) respond maximally to the same set of stimulus as corresponding S cells, yet are locally invariant to their exact position.

This work is at the source of many neurally inspired models of computational vision: the NeoCognitron [16], HMAX [59] and LeNet-5 [40]. While they may differ in the details of their implementation, all these models share the same basic architecture, an example of which is shown in Fig. 1.4. They alternate layers of simple and complex units [5], arranged in 2D grids to mimic the visual field. Each unit at layer l is connected to a local subset of units at layer $l-1$, much like the receptive fields of Hubel and Wiesel. With the exception of this local connectivity, (S) units perform the same task as the artificial neurons of a standard neural network. The output of an (S) neuron can therefore be modeled with $h_i^{(S)}(x) = \text{sigmoid}(\sum_{j \in \text{rec field of } h_i} w_{ij} x_j + b)$, where i (as well as j) represents the 2D coordinates of a neuron in the hidden and visible layers respectively. The weights of an (S) neuron therefore represent a visual feature or template to which it responds maximally if present in its receptive field. (C) neurons receive the output from (S) units in their receptive fields and perform some kind of pooling function, such as computing the mean or max of their inputs. In doing so, they also act as a sub-sampling layer (i.e. fewer cells per retina area are necessary). This pooling is meant to replicate the invariance to position which was observed in (C) cells.

[5] For clarity, we use the word "unit" or "neuron" to refer to the artificial neuron and "cell" to refer to the biological neuron.

LeNet-5 additionally adds the constraint that all (S) neurons at a given layer are "replicated" across the entire visual field. These form **feature maps** which are shown in Fig. 1.4 as stacks of overlapping rectangles. (S) neurons within the same feature map share the same parameters W and each feature map contains a unique offset b. The result is a **Convolutional Neural Network** (CNN). CNNs get their name from the fact that the activations of neurons within feature map i can be written as $h_i(x) = \text{sigmoid}(W_i * x + b)$, where $*$ is the convolutional operator.

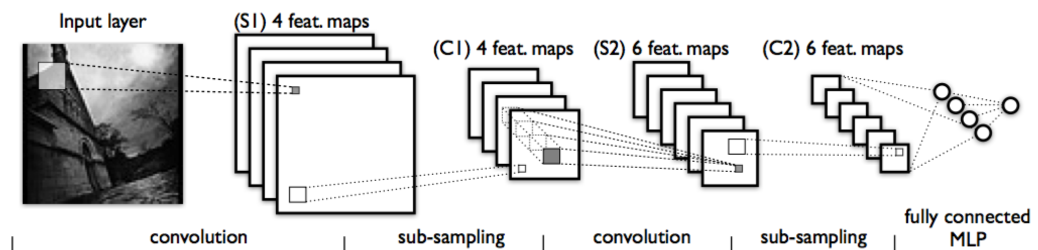


Figure 1.4: An example of a convolutional neural network, similar to LeNet-5. The CNN alternates convolutional and sub-sampling layers. Above, the input image is convolved with 4 filters generating 4 feature maps at layer (S1). Layer (C1) is then formed by down-sampling each feature map in (S1). Layer (S2) is similar to (S1), but uses 6 filters. Note that the receptive fields of units in (S2) span all 4 feature maps of (C1). The top-layers are fully-connected and form a standard MLP.

LeNet is of particular interest to this thesis, as it is the only model, of the 3 mentioned, which is trained through backpropagation. Since its inception, it has also achieved impressive results on a wide-array of visual recognition tasks which remain competitive to this day (0.95% classification error on MNIST [40]). The backpropagation algorithm of Eqs. 1.12-1.18. need only be modified slightly to account for the parameter sharing, by summing all parameter gradients originating from within the same feature map.

CNNs are very attractive models for vision. Features of interest (to which (S) cells are tuned) are detected regardless of the exact position of the stimulus. CNNs are thus naturally position equivariant, a property which would have had to be learnt in a traditional ANN. Also, the local structure of the receptive fields exploits the local correlation present in 2D images and after training, leads to **local feature detectors** such as edges

or corners in the first layer. The pyramidal nature of CNNs also means that higher-level units learn features which are more **global**, i.e. which span a larger field than the first layer. The pooling operation of complex cells may also provide some level of translation invariance, as well as invariance to small degrees of rotation [59]. This may help in making CNNs more robust.

Finally, CNNs massively cut-down on the amount of parameters which need to be learnt. Since we know that learning more parameters requires more training data [2], this helps the learning process. By controlling model capacity, CNNs also tend to achieve better generalization on vision problems.

1.5 Alternative Models of Computation

ANNs achieve non-linear behaviour by stacking multiple layers of simple units of the form $y(x) = s(Wx + b)$. Multiple layers are required because these simple units can only capture first-order correlations. Another solution is to use **higher-order units** which capture higher-order correlations such as the covariance between all pairs of input components (x_i, x_j). These were first introduced in [44] under the name of HOTLU (or high-order threshold logic unit). They showed that a simple second-order unit can learn the XOR logic function [6] in a single-pass of the training set. Formally, [17] defines higher order units as,

$$y_i(x) = s(T_0(x) + T_1(x) + T_2(x) + \ldots) \tag{1.19}$$

$$= s(b + \sum_j W_{ij} x_j + \sum_j \sum_k W_{ijk} x_j x_k + \ldots) \tag{1.20}$$

where the maximum index of T defines the order of the unit. While Minsky and Papert [44] claimed that the added complexity made them impractical to learn, Giles and Maxwell [17] showed that using prior information, higher-order units can be made invariant to certain transformations at a relatively small price. For example, shift invariance can be implemented very cheaply by a second-order unit, under the conditions that

[6] $XOR(i, j) = \begin{cases} +1 \text{ if } sign(i) = sign(j), \\ -1 \text{ if } sign(i) \neq sign(j). \end{cases}$.

$\{W_{ijk} = W_{i(j-m)(k-m)}; m \in \mathbb{N}, \forall j,k\}$. Having these built-in invariances is very advantageous. Since the network does not have to learn them from data, higher-order units can achieve better generalization with smaller training sets.

Computational neuroscience also provides additional arguments for higher order units. While the basic artificial neural unit introduced in section 1.3 vaguely resembles the architecture and behaviour of a biological neuron, there is no doubt that the real behaviour of a biological neuron is much more complex. Recently, Rust *et al.* [56] studied the behaviour of simple and complex cells in the early visual cortex of macaque monkeys, known as V1. They showed that the behaviour of simple (S) cells accounted for several linear filters, some of which were excitatory while others were inhibitory. Their model also showed a better fit to the cell's firing rate by taking into account pairs of filter responses. Their complete model, given in Eq. 6.1 of page 52, models cell behaviour as a weighted sum of squares of filter responses. This model will serve as inspiration to Chapter 6.

CHAPTER 2

DEEP LEARNING

From the discovery of the Perceptron, to the first AI winter and the discovery of the back-propagation algorithm, the history of connectionist methods has been a very tumultuous one. The latest chapter in neural network research involves moving past the standard Multi-Layer Perceptron (MLP) and into the field of **Deep Networks**: networks which are composed of many layers of non-linear transformations.

We have seen in Chapter 1 that the single-layer MLP is a universal approximator. Given enough hidden units and the ability to modify the parameters of the hidden and output layers, such MLPs can approximate any continuous function. While this revelation has been a strong argument in favor of neural networks, it fails to account for the complexity of the required networks. Taking inspiration from circuit theory, Håstad [19] states that a function which can be "compactly represented by a depth k architecture might require an exponential number of computational elements to be represented by a depth $k-1$ architecture". To become a true universal approximator, a shallow network such as the MLP, might thus require an exponential number of hidden units. From [2], we know the amount of training data required for good generalization is proportional to the number of parameters in the network. Training shallow networks might thus require an exponential amount of training data, a seemingly prohibitive task.

To make things worse, standard training of MLPs is purely supervised. This is problematic on two levels. Manual annotation of datasets is a very time-consuming and expensive task. One would thus benefit greatly from being able to use unlabeled data during the learning process. Second, it could be argued that to capture the real essence of a dataset $\mathscr{D}$, one would need to model the underlying joint-probability $p_T(x,y)$. The only learning signal used in supervised learning however, stems from the conditional-class probability $p(y=m|x)$. Since $p(x,y) = p(y|x)p(x)$, the use of the prior $p(x)$ in learning thus seems attractive.

In 2006, Hinton *et al.* [26] introduced the **Deep Belief Network** (DBN), a break-

through in the field of deep neural networks. They introduced a greedy layer-wise training procedure based on the **Restricted Boltzmann Machine** (RBM), which opened the door to learning deep hierarchical representations in an efficient manner. DBNs can also be used to initialize the weights of a deep feed-forward neural network. After a supervised fine-tuning stage[1], this unsupervised learning procedure leads to better generalization performance compared to traditional random initialization [5, 24].

The research presented in Chapters 4 and 8 was largely conducted to expand on this work. As such, this present chapter will focus on providing the reader with the necessary background material. Section 2.1 starts with an overview of Boltzmann Machines and their basic learning rule. We then proceed in section 2.2 with a short primer on Markov Chains and a particular form of Markov-Chain Monte Carlo (MCMC) sampling technique known as **Gibbs sampling**. From there, we will be able to cover the details of the DBN.

2.1 Boltzmann Machine

Boltzmann Machines (BM) [25] are probabilistic generative models which learn to model a distribution $p_T(x)$, by attempting to capture the underlying structure in the input. BMs contain a network of binary probabilistic units, which interact through weighted undirected connections. The probability of a unit s_i being "on" given its connected neighbours, is stochastically determined by the state of these neighbours, the strength of the weighted connections and the internal offset b_i. Positive weights w_{ij} indicate a tendency for units s_i and s_j to be "on" together, while $w_{ij} < 0$ indicates some form of inhibition. The entire network defines an energy function, defined as $\mathsf{E}(s) = -\sum_i \sum_{j>i} w_{ij} s_i s_j - \sum_i b_i x_i$. The stochastic update equation is then given by:

$$p(s_i = 1 | \{s_j : \forall j \neq i\}) = \text{sigmoid}(\sum_j w_{ij} s_j + b_i), \tag{2.1}$$

[1]The supervised fine-tuning stage consists in using the traditional supervised gradient descent algorithm of section 1.3.3, using the weights learnt during the layer-wise pre-training as initial starting conditions.

a stochastic version of the neuronal activation function found in ANNs. Under these conditions and at a stochastic equilibrium, it can also be shown that the probability of a given global configuration is given as

$$p(s) = \frac{1}{Z} e^{-\mathbb{E}(s)}. \tag{2.2}$$

High probability configurations therefore correspond to low-energy states.

Useful learning is made possible by splitting the units into **visible** and **hidden** units, as shown in Fig. 2.1(a), i.e. $s = (v, h)$. During training, visible units are driven by training samples $x^{(i)}$ and the hidden units are left free to converge to the equilibrium distribution. The goal of learning is then to modify the network parameters θ in such a way that $p(v) = \sum_h p(v, h)$ is approximately the same during training (with visible units clamped) and when the entire network is free-running. This amounts to maximizing the empirical log-likelihood

$$\frac{1}{N} \sum_{i=1}^{n} \log p(v = x^{(i)}). \tag{2.3}$$

From Eq. 2.3, we can derive a stochastic gradient over the parameters θ for training example $x^{(i)}$:

$$\left. \frac{\partial \log p(v)}{\partial \theta} \right|_{v=x^{(i)}} = -\sum_h p(h|v = x^{(i)}) \frac{\partial \mathbb{E}(x^{(i)}, h)}{\partial \theta} + \sum_{v,h} p(v, h) \frac{\partial \mathbb{E}(v, h)}{\partial \theta} \tag{2.4}$$

The above gradient is the sum of two terms, corresponding to the so-called **positive** and **negative phases**. The first term is an average over $p(h|v = x^{(i)})$ (i.e. probability over the hidden units given that the visible units are clamped to training data). It will act to decrease the energy of the training examples, referred to as **positive examples**. The second term, an average over $p(v, h)$, is of opposite sign and will thus act to increase the energy of configurations sampled from the model. These configurations are referred to as **negative examples**, as they are training examples which the network needs to *unlearn*. Together, this *push-pull* mechanism attempts to mold an energy landscape

where configurations with visible units corresponding to training examples have low-energy and all other configurations have high-energy.

To apply Eq. 2.4, we must first have a mechanism for obtaining samples from $p(h|v)$ and $p(v,h)$. The following section covers the basic principles of Markov Chains along with the Gibbs sampling algorithm, which will prove useful for this task.

2.2 Markov Chains and Gibbs Sampling

2.2.1 Markov Chains

A Markov Chain is defined as a stochastic process $\{X^{(n)} : n \in T, X^{(n)} \in \chi\}$ where the distribution of the random variable $X^{(n)}$ depends entirely on $X^{(n-1)}$. This can be written as:

$$p(X^{(n)}|X^{(0)},...,X^{(n-1)}) = p(X^{(n)}|X^{(n-1)})$$

The dynamics of the chain are thus entirely determined by the transition probability matrix P, whose elements p_{ij} determine the probability of making a transition from state i to state j. Given an initial state μ_0, the distribution at step n, is thus given by $\mu_0 P^n$. Chains of interest are those which are said to be irreducible and ergodic[2]. Under these conditions, a Markov chain will have a unique **stationary distribution** π such that $\pi P = \pi$ and the stationary distribution is the **limiting distribution** [67]. Mathematically, this translates to:

$$\lim_{n \to \infty} P^n_{ij} = \pi_j,\ \forall i. \tag{2.5}$$

An ergodic, irreducible Markov chain should therefore converge to its stationary distribution π if it is run for a sufficient number of steps. This is known as the **burn-in period**. This leads to an important result at the foundation of most MCMC sampling methods and which will prove useful for training Boltzmann Machines.

[2] Simply put, irreducibility implies that all states are accessible from each other with non-null probability. Chains are said to be ergodic if they are aperiodic and have states which revisit themselves in finite time and with probability 1. For further details, we refer the reader to [67].

An important property of Markov chains, is that, for any bounded function g [67]:

$$\lim_{N\to\infty} \frac{1}{N} \sum_{n=1}^{N} g(X^{(n)}) = E_\pi(g) = \sum_j g(j)\pi_j \tag{2.6}$$

While this only holds true in the limit of $N \to \infty$, in practice this means that samples obtained after a sufficient burn-in period can be treated as samples from π. The quality of the estimate $\hat{E_\pi}(g)$ is then determined by the **mixing rate** of the chain. The mixing rate relates to the amount of correlation between consecutive samples, with good mixing corresponding to zero or low correlation. In Chapter 8, we will see that good mixing is key to the successful training of RBMs.

2.2.2 The Gibbs Sampler

For the above to be useful for training Boltzmann Machines, we still require a mechanism for building a Markov chain with stationary distributions $p(h|v)$ and $p(v,h)$. This can be achieved by a process called **Gibbs sampling** [53]. Given a multivariate distribution $p(\mathbf{X} = X_1,...,X_p)$, the trick is to build a Markov chain with samples $X^{(i)}, i \in \mathbb{N}$ which, given the previous value $X^{(n)}$ of the chain state variable, has the transition probabilities as defined in Eqs. 2.7-2.10. For clarification, the superscript refers to the chain index within the Markov chain while the subscript is used to index a particular random variable (e.g. random variable formed by unit s_i in a Boltzmann machine)

$$X_1^{(n+1)} \sim p(x_1|x_2^{(n)}, x_3^{(n)}, ..., x_p^{(n)}) \tag{2.7}$$

$$X_2^{(n+1)} \sim p(x_2|x_1^{(n+1)}, x_3^{(n)}, ..., x_p^{(n)}) \tag{2.8}$$

$$... \tag{2.9}$$

$$X_p^{(n+1)} \sim p(x_p|x_1^{(n+1)}, x_2^{(n+1)}, ..., x_{p-1}^{(n+1)}) \tag{2.10}$$

Each variable is thus sampled independently, whilst keeping the other variables fixed. As an example, to sample from $p(h|v)$ for the BM of Fig. 2.1(a), we would build a chain as stated above with $X = (h_0, h_1, h_2)$ and inputs v clamped. To sample from $p(v,h)$, we

simply set $X = (\{v_i \forall i\}, \{h_j \forall j\})$. From Eq. 2.6, repeating the above procedure many times results in values $\mathbf{X}^{(n)}$ which can be treated as samples of $p(h|v)$ and $p(v,h)$.

Using Gibbs sampling to learn a BM is very expensive however. For each parameter update, one must run two full Markov chains to convergence, with each transition representing a full step of Gibbs sampling. For this reason, we now turn to the Restricted Boltzmann Machine, for which efficient approximations were devised.

2.3 Deep Belief Networks

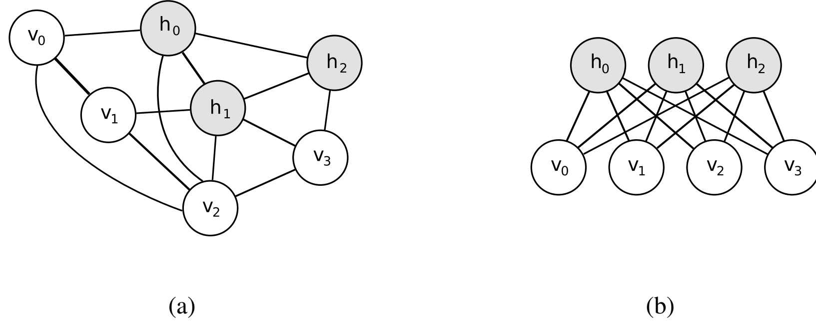
This section covers the core aspects of the Deep Belief Network [26]. We start with a description of the Restricted Boltzmann Machine and show how it improves upon the generic learning algorithm of a BM. We then tackle the Contrastive Divergence algorithm, a trick for speeding up the learning process even further, and finally show how RBMs can be stacked to learn deep representations of data.

2.3.1 Restricted Boltzmann Machine

Restricted Boltzmann Machines are variants of BMs, where visible-visible and hidden-hidden connections are prohibited. The energy function $\mathbb{E}(\mathbf{v},\mathbf{h})$ is thus defined by Eq. 2.11, where $\mathbf{W}$ represents the weights connecting hidden and visible units and $\mathbf{b}$, $\mathbf{c}$ are the offsets of the visible and hidden layers respectively.

$$\mathbb{E}(v,h) = -b'v - c'h - h'Wv \tag{2.11}$$

The biggest advantage of such an architecture is that the hidden units become conditionally independent, given the visible layer (and vice-versa). This is self-evident from looking at the graphical model of Fig. 2.1(b) and may also be derived from Eqs. 2.11



(a) (b)

Figure 2.1: Example of (a) Boltzmann Machine (b) Restricted Boltzmann Machine. Visible units are shown in white and hidden units in gray. For clarity, we omit the weights on each undirected connection along with the offset.

and 2.2. We can therefore write,

$$p(h|v) = \prod_i p(h_i|v) \tag{2.12}$$

$$p(v|h) = \prod_j p(v_j|h). \tag{2.13}$$

This greatly simplifies the learning rule of Eq. 2.4, as inference now becomes trivial and exact. As an example, we derive the gradient on W_{ij}. Gradients on the offsets can be obtained in a similar manner.

$$\left.\frac{\partial \log p(v)}{\partial \theta}\right|_{v=x^{(i)}} = -\sum_h \prod_i p(h_i|v=x^{(i)}) \left.\frac{\partial \mathbb{E}(v,h)}{\partial W_{ij}}\right|_{v=x^{(i)}} + \sum_{v,h} \prod_i p(h_i|v)p(v) \frac{\partial \mathbb{E}(v,h)}{\partial W_{ij}} \tag{2.14}$$

$$= -\sum_h p(h_i|x^{(i)}) h_i \cdot x_j^{(i)} + \sum_{v,h} p(h_i|v)p(v) h_i \cdot v_j \tag{2.15}$$

$$= -p(h_i = 1|x^{(i)}) \cdot x_j^{(i)} + \sum_v p(h_i = 1|v)p(v) \cdot v_j \tag{2.16}$$

$$= -x_j^{(i)} \cdot \text{sigmoid}(W_i \cdot x^{(i)} + c_i) + E_v[p(h_i|v) \cdot v_j] \tag{2.17}$$

As we can see from Eq. 2.17, the positive phase gradient is straightforward to compute. Computing the negative phase gradient still requires samples from $p(v)$ however.

How to get these negative samples is what sets most of the RBM training algorithms apart (see Chapter 8). Regardless of their peculiarities, they all exploit the fact that Gibbs sampling is very efficient in an RBM. Because units in one layer are conditionally independent given the other layer, getting $x^{(n+1)}$ from $x^{(n)}$ can be achieved in two steps:

$$h^{(n+1)} \sim \text{sigmoid}(W'x^{(n)} + c) \tag{2.18}$$

$$x^{(n+1)} \sim \text{sigmoid}(Wh^{(n+1)} + b) \tag{2.19}$$

For a more detailed derivation of all the above formulas, we refer the reader to [3].

2.3.2 Contrastive Divergence

The Contrastive Divergence learning algorithm [26] relies on the following two observations to speed-up learning:

1. since the Gibbs chain takes a long time to converge, initializing the chain with a training example $x^{(i)}$ (a sample of the distribution we wish to approximate) "should" help accelerate the burn-in process. Note in particular that when $p \approx p_T$, burn-in is immediate.

2. instead of letting the Markov chain converge to its equilibrium distribution before extracting a sample, run the chain for k-steps only. The resulting algorithm is referred to as "CD-k". Bengio and Delalleau [4] later showed that this approximation was warranted since the gradient of CD-k can be viewed as a series which converges to the true gradient and whose terms tend to 0 as $k \rightarrow \infty$.

While Chapter 8 will provide counter-arguments to the above statements, CD-1 has been found to work well in practice [26, 33]. CD-1 updates are illustrated in Fig. 4.1. For a given input $v^{(1)} = x^{(i)}$, they are given as follows:

$$\partial W_{ij} = -h_i^{(1)} \cdot v_j^{(1)} + p(h_i = 1|v^{(2)}) \cdot v_j^{(2)} \tag{2.20}$$

$$\partial \mathbf{c}_i = -h_i^{(1)} + p(h_i = 1|v^{(2)}) \tag{2.21}$$

$$\partial \mathbf{b}_j = -v_j^{(1)} + v_j^{(2)} \tag{2.22}$$

The parameter updates therefore encourage the visible-hidden correlations in the negative phase to match those in the positive phase.

2.3.3 Greedy Layer-Wise Training

Hinton showed in [26], that RBMs can be stacked and trained in a greedy manner to form so-called Deep Belief Networks. DBNs are graphical models which learn to extract a deep hierarchical representation of the training data.

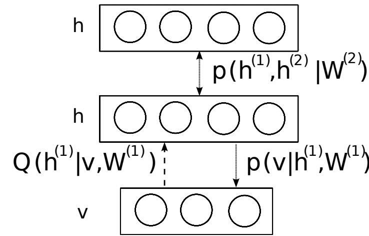


Figure 2.2: Example of a two-layer DBN. The posteriors $Q(h^{(l)}|v)$ are used to generate the representation at layer l. The top-two layers form an RBM which, together with $p(v|h^{(l)})$ form a generative model.

The principle is the following. Start by training a single RBM on the training distribution $p_T(x)$. Once the RBM is fully trained, freeze its weights and use its conditional distribution $p(h|v)$ (referred to as $Q(h|v)$ from now on) to generate a new distribution $p(x^{(1)})$, such that $x^{(1)} \sim \sum_x Q(h|v=x) p_T(v)$. This new distribution forms the training examples for the second layer RBM and the process is repeated until we reach an architecture with sufficient depth L. The resulting DBN is thus a graphical model as shown

in Fig. 2.2, which defines the following joint distribution [33]:

$$p(v,h^{(1)},...,h^{(L)}) = p(v|h^{(1)})p(h^{(1)}|h^{(2)})...p(h^{(L-2)}|h^{(L-1)})p(h^{(L-1)},h^{(L)}) \tag{2.23}$$

The arcs in bold define a generative model which can be used to sample from the model. The dashed arc illustrates the posteriors $Q^{(l)}(h|v)$ of the RBM used to generate the representation at layer l.

Why does such an algorithm work ? Taking as example a 2-layer DBN with hidden layers $h^{(1)}$ and $h^{(2)}$, Bengio [3] established that $\log p(v)$ can be rewritten as,

$$\begin{aligned}\log p(v) =& KL(Q(h^{(1)}|v)||p(h^{(1)}|v)) + H_{Q(h^{(1)}|v)} + \\ & \sum_h Q(h^{(1)}|v)(\log p(h^{(1)}) + \log p(v|h^{(1)}))\end{aligned} \tag{2.24}$$

$KL(Q(h^{(1)}|v)||p(h^{(1)}|v))$ represents the KL divergence between the posterior $Q(h^{(1)}|v)$ of the first RBM if it were standalone, and the probability $p(h^{(1)}|v)$ for the same layer but defined by the entire DBN (i.e. taking into account the prior $p(h^{(1)},h^{(2)})$ defined by the top-level RBM). H is the entropy function. It can be shown that if we initialize both hidden layers such that $W^{(2)} = {W^{(1)}}^T$, $Q(h^{(1)}|v) = p(h^{(1)}|v)$ and the KL divergence is null. First learning the first level RBM, then keeping W_1 fixed and optimizing Eq. 2.24 with respect to $W^{(2)}$ can thus only increase the likelihood $p(v)$. Also, notice that if we isolate the terms which depend only on $W^{(2)}$, we get: $\sum_h Q(h^{(1)}|v)p(h^{(1)})$. Optimizing this with respect to $W^{(2)}$ amounts to training a second-stage RBM, using the output of $Q(h^{(1)}|v)$ as the training distribution.

CHAPTER 3

OVERVIEW OF THE FIRST PAPER

Empirical Evaluation of Convolutional RBMs for Vision.
Desjardins, G. and Bengio, Y.
Technical Report 1327, Université de Montréal. Oct, 2008

3.1 Context

Deep Belief Networks, published in 2006 by Hinton *et al.* [26], introduced the idea of using unsupervised learning as a way to pretrain deep neural networks. Within that same year, several other research groups [5, 50] published similar findings. These groundbreaking papers generated a lot of excitement in the field of connectionism leading to a workshop on Deep Learning at the 2007 Neural Information Processing Systems (NIPS) conference. At the time this technical report was published, most applications of deep networks had focused on learning from small MNIST-like images [26, 33, 39]. Ranzato *et al.* [51] also explored performing unsupervised learning from small image patches and using the resulting filters to initialize the features of a larger convolutional architecture. Motivated by previous work on convolutional neural networks (see section 1.4), the goal of this work was to show that DBNs could benefit from having a convolutional architecture and eventually scale DBNs to larger images.

3.2 Contributions

This technical report lays the groundwork for convolutional DBNs. It starts by introducing the convolutional RBM (CRBM), which is a modification of the traditional RBM explored in section 2.3.1. Much like in the bottom layers of LeNet-5, hidden units have local receptive fields which span only a subset of the visible layer. They are also grouped into feature maps, which share the same parameters (weights and offsets). This allows hidden units to model local regions of input space, which share the same parametriza-

tion across the entire visual field. This work also explores various training algorithms for CRBMs. We start by showing that Contrastive Divergence can be easily adapted to account for the parameter sharing. We then show empirically that training CRBMs directly (i.e. on the full input image) is more efficient than training a fully-connected architecture on image patches, and then using this to initialize a convolutional network. To the best of our knowledge, this was the first reported implementation of such an architecture.

In terms of the authors' contributions, all of the underlying technical work was done by this author. It is however the result of joint-work with Yoshua Bengio, who guided the project to fruition through constructive feedback and thoughtful discussions. The technical writing was also entirely done by myself.

3.3 Comments

Since the writing of this report, most of the issues outlined as future work have been addressed and implemented. CRBMs have been successfully integrated as part of deep networks, the code optimized for larger datasets and max-pooling implemented as an additional step in the greedy layer-wise training procedure. The resulting architecture, dubbed LeDeepNet, has also been modified to support the use of Denoising Auto-Encoders as the basic building block [66].

Unfortunately, this work was never published. Implementation details, which have only recently been addressed, made LeDeepNet unsuitable for large-scale images such as Caltech 101 [13]. An unfortunate bug was also introduced during the rewrite, with serious consequences for the pre-training procedure.

Since then, Lee *et al.* [42] have published an award-winning paper, which integrates CRBMs in a full multi-layered probabilistic model, with very impressive results. They also introduce a probabilistic version of max-pooling, which not only allows for top-down interactions, but also naturally enforces local sparsity constraints. This work makes LeDeepNet somewhat obsolete. As such, future research directions must be re-evaluated. We take comfort in the fact that the technical report presented in the following chapter, was cited by [42, 49] as contemporary work in the development of CRBMs.

CHAPTER 4

EMPIRICAL EVALUATION OF CONVOLUTIONAL RBMS FOR VISION

4.1 Abstract

Convolutional Neural Networks have had great success in machine learning tasks involving vision and represent one of the early successes of deep networks. Local receptive fields and weight sharing make their architecture ideally suited for vision tasks by helping to enforce a prior based on our knowledge of natural images. This same prior could also be applied to recent developments in the field of deep networks, in order to tailor these new architectures for artificial vision. In this context, we show how the Restricted Boltzmann Machine (RBM), the building block of Deep Belief Networks, can be adapted to operate in a convolutional manner. We compare their performance to standard fully-connected RBMs on a simple visual learning task and show that the convolutional RBMs (CRBMs) converge to smaller values of the negative likelihood function. Our experiments also indicate that CRBMs are more efficient than standard RBMs trained on small image patches, with the CRBMs having faster convergence.

4.2 Introduction

Convolutional architectures have a long history in vision applications. They are largely inspired by models of the visual cortex and employ feature detectors which are sensitive to small regions of input space, called receptive fields. These detectors are replicated throughout the image and form so-called **feature maps**. In Artificial Neural Networks (ANN), this is achieved by forcing neurons within a feature map to have the same weights and offsets. This allows for the same feature to be detected at every point in the image. Feature maps are further grouped into layers and stacked, so that the output of one layer forms the input of the next. This pyramidal structure allows the initial layers to detect low-level features which are highly local in nature, such as edges or corners, which are then combined by the upper layers to generate more global and abstract

features.

The success of Convolutional Neural Networks (CNN), such as LeNet-5 [40] in artificial vision tasks like hand-written digit classification or object recognition, stems from their architecture and inherent constraints. The weight sharing within a feature map greatly reduces the number of free parameters which makes them less prone to overfitting. We can think of the locality constraints and position invariance as enforcing a prior based on our knowledge of natural images [3]. This acts as a regularization process which greatly facilitates their training. Historically, deep ANNs have been notoriously difficult to optimize. As we move towards deep architectures in machine learning, we should therefore try to leverage these concepts as CNNs represent one of the early successes of deep networks. In this report, we will therefore show how the same principles can be applied to the Restricted Boltzmann Machine (RBM), the building block of Deep Belief Networks (DBN). We will start by introducing the RBM and its training algorithm and then show how they can be adapted to operate in a convolutional manner. We will then showcase the experiments which were done, which seem to indicate that CRBMs are more efficient than traditional RBMs at learning to model images.

4.3 Restricted Boltzmann Machines

Boltzmann Machines are a probabilistic model, which define a joint energy between units in a **visible layer** v and a **hidden layer** h. In a Restricted Boltzmann Machine, connections are prohibited between units of the same layer. The energy $\mathbb{E}(v,h)$ is given by Eq. 4.1 and can be converted to a probability through the partition function Z defined below.

$$\mathbb{E}(v,h) = -b'v - c'h - h'Wv \tag{4.1}$$

$$P(v,h) = \frac{e^{-\mathbb{E}(v,h)}}{Z}, \text{ with } Z = \sum_{v,h} e^{-\mathbb{E}(v,h)} \tag{4.2}$$

Here, (b,c,W) are the offset and weight parameters θ of the model and have a definition similar to those of traditional neural networks. Learning in an RBM consists in

modifying θ in order to minimize the energy of observed configurations while increasing the energy of the other configurations. Since probability and energy have an inverse relationship, this increases the probability of observed data. The probability of observed v is obtained by marginalizing over the hidden layer, such that $P(v) = \sum_h P(v,h)$. To train the RBM, we need to estimate the gradient of the log-likelihood function $\log P(v)$:

$$\frac{\partial \log P(v)}{\partial \theta} = -\sum_h P(h|v)\frac{\partial \mathbf{E}(v,h)}{\partial \theta} + \sum_{v,h} P(v,h)\frac{\partial \mathbf{E}(v,h)}{\partial \theta}. \tag{4.3}$$

We refer the reader to [23, 26] for the presentation of RBMs and to [4] for derivations and further analysis of the log-likelihood gradient. Since the partition function is intractable, so is the computation of the above gradient. Contrastive Divergence (CD) approximates this gradient through a truncated Gibbs Markov chain. Starting from a valid training sample $x^{(1)}$ in the visible layer, CD-k generates samples ($x^{(t)}$, $y^{(t)}$) of the distribution according to $y^t \sim p(\mathbf{h}|\mathbf{v} = x^t)$ and $x^t \sim p(\mathbf{v}|\mathbf{h} = y^{t-1})$ with $t \in [1 \ldots k+1]$. Updates for weights W and offsets b and c are then performed in the direction given by ΔW, Δb and Δc respectively:

$$\Delta b = -x^{(1)} + x^{(k+1)} \tag{4.4}$$

$$\Delta c = -y^{(1)} + p(\mathbf{h} = 1|\mathbf{v} = x^{(k+1)}) \tag{4.5}$$

$$\Delta W = -y^{(1)\prime} \cdot x^{(1)} + p(\mathbf{h} = 1|\mathbf{v} = x^{(k)})' \cdot x^{(k+1)}. \tag{4.6}$$

where (abusing a bit notation), $p(\mathbf{h} = 1|\mathbf{v} = x^k)$ represents the vector whose elements are $p(\mathbf{h}_i = 1|\mathbf{v} = x^k)$. Pseudocode for CD is shown in the appendix (see `CRBM_CD`). Finally, it is important to note that sampling in an RBM is very efficient since the units in one layer are conditionally independent given the state of the other layer. For binary stochastic units, the activation probability of each unit h_i (or conversely v_i) is given by:

$$p(h_i = 1|v) = \text{sigmoid}(c_i + \sum_j W_{ij} v_i), \tag{4.7}$$

where sigmoid is the function defined as $\text{sigmoid}(x) = 1/(1+\exp(-x))$.

4.4 Convolutional RBMs

4.4.1 Architecture of CRBMs

In this section, we show how RBMs and CD can be adapted to work in a convolutional manner. There are two ways one can think of convolutional RBMs. From a theoretical point of view, they are simply very large fully-connected RBMs where each unit represents a particular feature at a certain position within the image. The weights W thus form a high-dimensional tensor where $W_{ij,mn,uv}$ refers to the weight connecting the i-th feature at pixel (m,n) of the hidden layer (with offset $c_{i,mn}$) to the j-th feature at pixel (u,v) of the visible layer (with offset $b_{j,uv}$). Their convolutional nature imposes specific constraints on the weights such that:

$$W_{ij,mn,uv} = W_{ij,m'n',(u+(m'-m))(v+(n'-n))} \; \forall \text{ hidden layer pixels } (m,n),(m',n') \quad (4.8)$$

$$W_{ij,mn,uv} = 0 \; \forall (u,v) \text{ such that } |u-m| > \alpha \text{ or } |v-n| > \beta \quad (4.9)$$

$$c_{i,mn} = c_i \; \forall \text{ pixels } (m,n) \text{ in hidden layer} \quad (4.10)$$

$$b_{j,uv} = b_j \; \forall \text{ pixels } (u,v) \text{ in visible layer} \quad (4.11)$$

where α and β determine the size of the receptive fields.

Alternatively, it can be easier to visualize CRBMs as simply performing a convolution using a standard RBM as the kernel. Using Fig. 4.1 as an example, the advantages offered by CRBMs become apparent. On vision tasks, the number of parameters required in a standard RBM grows with the dimension of the input image. In contrast, the complexity of a CRBM is solely determined by the size of the receptive field and the number of features we wish to extract, and does not depend on the input image. In this mindset, the hidden layer shown in Fig. 4.1 can be considered as having only 3 features (instead of 12) which together, form a multi-dimensional pixel within the hidden layer's feature map. At each pixel, hidden units are connected to a subset of the visible units, located within their receptive field (area shown in gray). Conceptually, these units once vectorized, act as the visible layer of a standard RBM. The CRBM shown therefore per-

forms an operation which is similar (but not equivalent) to that of a 3-(4x1) RBM, where the digits indicate the number of hidden and visible units respectively. The difference is only in the down-pass, when considering $P(v|h)$: the hidden units at nearby locations have overlapping receptive fields, and interact through them. When referring to the parameters of CRBMs, we will sometimes use an abridged notation where we drop the hidden layer pixel indices from the weight matrix W, since these weights are replicated throughout the feature map. Such weights will have the notation $W_{ij,uv}$. Similarly, b_j will refer to the offset of the j-th feature replicated at every pixel of the visible layer, and c_i the replicated hidden layer offset of the i-th feature.

At each step of the convolution, the state of a single pixel $p(h_{i,mn} = 1|v), \forall i$, can be inferred using Eq. 4.7. Repeating this operation for every pixel in the hidden layer constitutes the **upward pass** and generates a set of feature maps, which are local feature detectors operating at every position in the image. It is clear that such an implementation follows the constraints outlined in Eqs. 4.8-4.11.

Conceptually, inferring $p(v|h)$ works in a similar manner. However, this approach does not make for the most straight-forward nor the most efficient implementation. While upwards propagation can easily be implemented as a tensor-product of the visible units and the weight matrix W, determining $p(v_{j,uv}|h), \forall j$, one pixel at a time requires complex indexing of W. This can be side-stepped by again iterating over the pixels of the hidden layer. At each time step, hidden units h_i at pixel (m,n) contribute an amount $\delta v_{i,mn,uv} = \sum_j p(h_{i,mn}|v) W_{ij,m,n,uv}$ to the net input of units in their receptive field. The net input refers to the activation probability $p(v_{j,uv} = 1|h)$ before applying the non-linearity. The **downward pass** therefore consists in iterating over all pixels in the hidden layer and summing their contributions for each pixel in the visible layer. Once this is completed, the offset is added and we apply the sigmoid function, thus generating $p(v_{j,uv} = 1|h)$. The pseudocode for both the upward and downward pass is presented in the appendix (see `PropUp` and `PropDown`)

For purposes of notation (and following [39]), we will refer to the CRBM of Fig. 4.1 as being of type 3@2x2 - 1@3x3. This indicates that the CRBM contains one feature in

the visible layer (the gray-level pixel value) arranged in a 3x3 feature map, and 3 features in the hidden layer arranged in a 2x2 map. Note that this implicitly defines a receptive field of size 2x2.

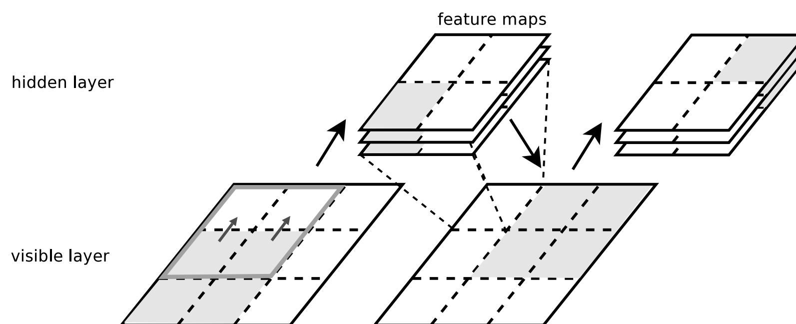


Figure 4.1: Contrastive Divergence in a convolutional RBM. In the above, the visible layer contains one feature which is arranged in a 3x3 feature map, representing the input image. The hidden layer contains 3 feature detectors. Since the units have a 2x2 receptive field, this generates a 2x2 feature map in the hidden layer. The CD algorithm remains unchanged propagating from the visible to the hidden layer during the positive phase, and down and back up again during the negative phase. In the above, the shaded areas represent a single time-step in the convolution, when propagating from the visible to the hidden layer. The state of the 3 hidden features shown in gray is inferred from the state of the visible units in their receptive field (also in gray). The grey outline and arrows represent the location of the next convolutional window. When inferring the state of the visible layer given the hidden layer, $p(v_{j,uv}|h)$ is inferred from all the hidden units which have pixel (u,v) in their receptive field.

,

4.4.2 Contrastive Divergence for CRBMs

The mechanisms for inferring the state of one layer given the other affords us the necessary tools for performing CD in a CRBM. As shown in Fig. 4.1, in its simplest form, CD-1 consists of the following steps:

1. Initialize the visible layer with a real image $x^{(1)}$ from the training distribution.

2. Perform an upward pass to infer pixel states in the hidden layer. Generate sample $y^{(1)} \sim p(h_{i,mn} = 1|v = x^{(1)})$, a multi-dimensional "image" containing the binary states of the feature maps.

3. Using sample $y^{(1)}$, perform a downward pass to infer pixel states in the visible layer. Generate sample image $x^{(2)} \sim p(v_{j,uv} = 1|h = y^{(1)})$.

4. Repeat upward pass using $x^{(2)}$ as data for the visible layer, to determine $p(h_{i,mn} = 1|v = x^{(2)})$

Before applying Eq. 4.4-4.6, we need to modify the learning procedure slightly to account for the parameter sharing. As in [40], this is done by computing the parameter gradients as if they were independent and then summing their individual contributions. As an example, the offset gradient of units in the visible layer is computed as $\Delta b_j = \sum_{uv} -x^{(1)}_{j,uv} + x^{(2)}_{j,uv}$. The same procedure can be applied to the offset in the hidden layer. Calculating the gradient for the weights is not as straightforward however. At each step of the convolution, we compute the tensor product of the hidden units at pixel (m,n) with the visible units within their receptive field. This value is either subtracted or added to ΔW depending on whether we are in the positive (upwards) or negative (downward) phase of CD.

4.5 Experiments

The goal of this experiment is to compare the performance of RBMs and CRBMs on visual learning tasks. Since measuring the likelihood of the data over the parameters of the model is intractable for RBMs, we rely on a generative procedure to estimate the negative-likelihood (NLL) throughout the training phase. This generative procedure is similar to the one described in [26] and involves the following steps: (1) we start by initializing the visible layer of the RBM with valid training data, (2) perform m steps of Gibbs sampling in the last RBM. The sampled vector in the visible layer is the result. Using this procedure, the quality of the model can be estimated by the proportion of valid generated images versus the total number of images generated. Let n_x be the number of examples generated by the RBM that are equal to input pattern x. To avoid assigning a probability 0 to any vector x, we consider the probability assigned to any x to be $\propto 1 + n_x$. More formally, let D_{train} be the training set containing N different training examples, x

a particular image configuration, n_x the number of images of type x generated and S the total number of images generated. The negative log-likelihood is estimated as follows:

$$NLL = \frac{-1}{|D_{train}|} \sum_{x \in D_{train}} \log \frac{1+n_x}{S+N} \tag{4.12}$$

Unfortunately, this method of estimating the NLL is rather costly in terms of memory since it requires us to build a histogram of all possible configurations of black or white pixels in an nxn image. So far, our experiments have thus been limited to small input images. The training data, shown below in Fig. 4.2, consists of small 4x4 binary images containing 21 different 2x2 or 3x3 shapes, which can be placed anywhere in the image. This gives a total of 123 different configurations and hopefully enough entropy in the source distribution to make reliable comparisons between the two models. Given $N = 123$ and generating $S = 10000$ samples, setting $n_x = S/N$ and $n_x = 0$ in Eq. 4.12 gives us the lower and upper bounds (respectively) of the NLL. The NLL estimator is thus bound to the following interval: $4.81 < NLL < 9.22$.

Figure 4.2: Subset of training data, which consists of small 4x4 images, containing simple 2x2 or 3x3 geometric figures (shown above) which can be positioned at 16 different positions.

We compared the performance of RBMs and CRBMs for the configurations shown in Table 4.1. The experiments were meant to encompass three separate test cases. In the first experiment, we start by comparing the learning dynamics of standard, fully-connected RBMs with 112 hidden and 16 visible units, to CRBMs having a limited receptive field of size 2x2. To provide a fair comparison, we increased the number of hidden units of the CRBM to 450 to keep the number of free parameters equal in both cases. The second phase of the experiment was meant to highlight the advantage of

learning in a convolutional manner, as opposed to simply using a standard RBM trained on a series of image patches. To this effect, we used an RBM with 450 hidden units and 4 visible units, in order to mimic a 2x2 receptive field. The RBM was then trained on consecutive image patches of size 2x2, thus mimicking a convolution operation. CD was applied at each image patch and thus generated 9 parameter updates per training image (i.e. the number of 2x2 patches which can be extracted from a 4x4 image). The resulting parameters were then used to initialize a compatible CRBM from which we could measure the NLL. In the last phase, we repeat the above experiments using 3x3 receptive fields. We experimented with several learning rates ε but found that in all cases, optimal performance was achieved with $\varepsilon = 0.1$. Those results are shown in section 6.5.

Furthermore, we also compared a variation of the CD learning algorithm discussed in section 4.4.2. Let us recall that parameter sharing in convolutional networks results in each pixel contributing in an additive manner to the parameter gradients. The variant consisted in averaging the parameter gradients of the offsets over the feature maps[1]. Updates to offset b_j for example, were thus computed as $\Delta b_j = (1/N_v) \sum_{u,v} \Delta b_{j,uv}$, where N_v is the number of pixels in the visible layer. We will refer to these experiments as *CRBM-sum* or *CRBM-mean*.

4.6 Results and Discussion

The result of these experiments are shown in Fig. 4.3. When using 2x2 receptive fields, we can clearly see that CRBMs and RBMs trained patch-wise offer the best performance, almost converging to the entropy of the source distribution. Fully-connected RBMs on the other hand, fail to reach this minimum value regardless of the number of hidden units. One could argue that given more training time, they might achieve the same results, as their curves seem to retain a small negative slope at 50000 epochs. However, the point remains that their convergence is significantly slower and thus sub-optimal. Among the local methods, *CRBM-mean* offers the best performance. It converges to the minimum faster than RBMs trained patch wise, where taking the mean or the sum of off-

[1]The same concept could also have been applied to the weights $W_{ij,uv}$. However this was shown, experimentally, to be very detrimental to the performance of CRBMs.

Table 4.1: Description of architectures used in our experiments. In order to perform a fair comparison between the above models, we adjusted the number of hidden and visible units to give approximately the same Degrees of Freedom (DoF) or number of free parameters. The RBM was also tested with an increased number of hidden units however, to determine if their poor performance was due to having too few hidden units for CD to work properly.

Type	Layers	DoF	Description
RBM	112 - 16	1792	RBM with 16 visible units and 112 hidden units.
	200 - 16	3200	RBM with 16 visible units and 200 hidden units.
	450 - 16	7200	RBM with 16 visible units and 450 hidden units.
RBM Patch	450 - 4	1800	RBM with 4 visible units and 450 hidden units. This RBM is trained on flattened 2x2 image patches, mimicking the effect of a convolution.
	200 - 9	1800	RBM with 9 visible units and 200 hidden units. This RBM is trained on flattened 3x3 image patches.
CRBM	450@3x3 - 1@4x4	1800	CRBM whose visible layer is a 4x4 image with unidimensional pixels. This CRBM contains 450 hidden units arranged in a 3x3 feature map (with receptive field of size 2x2).
	200@2x2 - 1@4x4	1800	This CRBM contains 200 hidden units arranged in a 2x2 feature map (with receptive field of size 3x3).

sets seems to have little or no effect. *CRBM-mean* offers surprisingly faster convergence than *CRBM-sum*, and thus seems to be the best method for the given training set. It is not quite understood why this is case, but it seems to imply that offsets might require a smaller learning rate than the weights.

These results are interesting for several reasons. First, this represents, to the authors' best knowledge, the first reported implementation of convolutional RBMs (by opposition to RBMs applied to image patches). Second, it is exciting to note that the advantages offered by CNNs, which have had tremendous success over the years, could also be of benefit to RBMs. Learning to model images using local methods does seem to offer a definite advantage, especially when learning is done in a convolutional manner. Although these results might appear counter-intuitive at first (since fully-connected architectures have more data at their disposal to learn a model of the distribution), we

believe this might be another example where "less is more" [11]. Having a limited receptive field could simplify the optimization process by making the learning criterion more smooth.

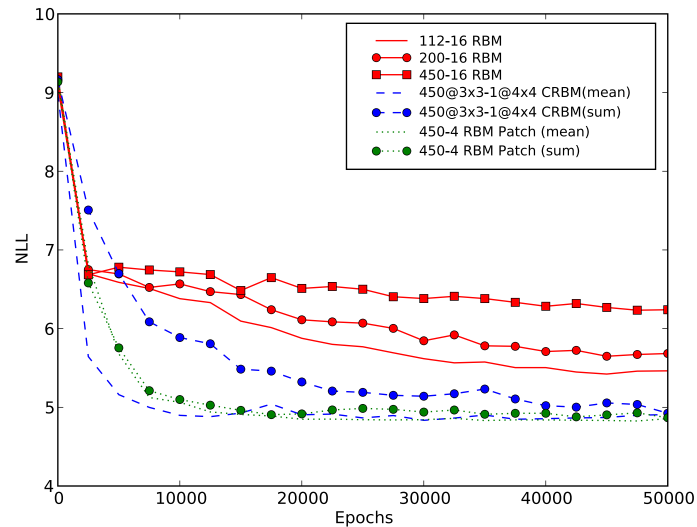
Also of interest is that these advantages seem to disappear as the size of the receptive field is increased to a value close to the size of the input image. In Fig. 4.3(b), CRBMs converge to about the same value and with the speed of regular RBMs. Only the *RBM Patch (mean)* seems to escape this local minimum. At this point, it is not clear why this is the case and warrants further investigation.

4.7 Future work

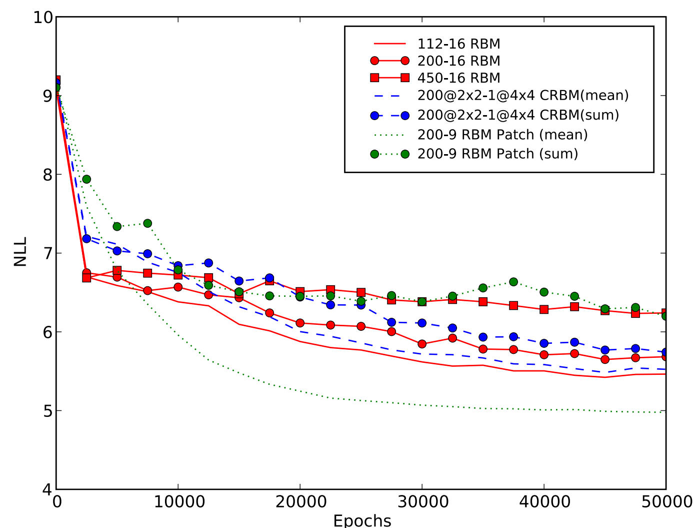
Much work is left to be done to evaluate the real-world performance of CRBMs in vision tasks. In the short term, these claims will need to be validated on larger-scale data sets, by increasing the factors of variation as well as the size the input images. To do this will require modifying the way we estimate the NLL during training, possibly using the method described in [58]. We also plan on training deep networks composed of stacks of CRBMs, complete with a supervised classification stage. This will enable us to estimate their performance on real-world vision tasks on datasets such as MNIST and NORB.

4.8 Conclusion

Convolutional architectures have proven to be very effective in machine learning applied to vision. By reducing the number of free parameters, the networks are more computationally efficient and less prone to overfitting. With the recent shift towards deep architectures in machine learning, in which the RBMs play a key role, it becomes interesting to see if RBMs can be modified in order to leverage these same basic principles. In this report, we have shown to the best of our knowledge, the first reported implementation of convolutional RBMs. Although this represents preliminary work, CRBMs show promise for use in vision applications. In this particular experiment, CRBMs (with proper receptive fields) offered the best performance, converging faster than RBMs trained on separate image patches. Fully-connected RBMs on the other



(a) Negative Likelihood of fully-connected RBMs, RBMs trained patch-wise and CRBMs, using 2x2 receptive fields. CRBMs and RBMs trained in patch mode almost converge to the entropy of the distribution. The RBMs do not reach this minimum, regardless of the number of hidden units. Of note, the CRBM in mean-mode seems to have the fastest convergence.



(b) Negative Likelihood of RBMs trained on 3x3 patches and CRBMs having 3x3 receptive fields. CRBMs and RBMs converge to approximately the same suboptimal value. RBM Patch in mean mode however, still offers a very good performance, again almost reaching the distribution entropy.

Figure 4.3: Comparing the performance of RBMs, CRBMs and RBMs trained on image patches.

hand, did not reach this minimum value. This confirms that convolutional architectures are better suited for vision and suggests that RBMs can be adapted to take advantage of this. Further work will focus on training CRBMs as part of deep architectures and applying them to complex discriminative tasks.

CHAPTER 5

OVERVIEW OF THE SECOND PAPER

Quadratic Polynomials Learn Better Image Features.
Bergstra, J., Desjardins, G., Lamblin, P. and Bengio, Y.
Submitted to the *International Conference on Machine Learning (ICML)*, 2009 (rejected)

5.1 Context

Research into deep networks is mostly focused on developing new training algorithms for artificial neural networks comprising many layers of non-linearity. As explained in [3], depth is however a subjective measure which depends on the set of allowed computations. Currently, most architectures or models have been based on a linear projection followed by a sigmoidal non-linearity (section 1.2). As we have seen in section 1.5 however, there may be certain advantages to using higher-order units capable of learning higher-order statistics (such as pair-wise correlations between input units). For a given level of computational complexity, a network composed of higher-order units may require fewer layers than one with only first-order units. This is a promising avenue of research as recent results have shown that performance starts decreasing in networks with more than 4 hidden layers, using current learning algorithms [12].

5.2 Contributions

Recent research in computational neuroscience also seems to justify the use of higher-order units in vision. Rust *et al.* [56] show that simple and complex cells in the visual cortex of macaque monkeys, exhibit much more complex behaviour than previously thought: behaviour which is very reminiscent of higher-order units.

The work presented in Chapter 6 uses this model as inspiration for experiments in object classification. The contributions are three-fold. First, we show that using the

model of [56] as a building block of ANNs can be done efficiently and translates to better performance on visual classification tasks. Second, we show that the non-linearity used in this model results in better generalization, compared to the traditional tanh activation function. We also investigate the issue of translation invariance in higher-order networks, which we discussed briefly in section 1.5.

With regards to the contributions of the authors, James Bergstra was the main driving force behind this work. He had previously studied other variants of the model presented in Chapter 6 and therefore had a very good insight into higher-order networks and some of the software was already in place. On the technical side, my main contribution was in adding support in Theano[1] for the various architectures tested in this paper (convolutional networks, with and without weight-sharing) and modifying the code to support all 3 architectures. Pascal Lamblin and I were in charge of the actual experiments, including the model selection procedure. This was not a trivial task given that these models have a large number of hyperparameters. It actually motivated the development of a "job management" tool[2] to streamline the process of launching experiments with many hyperparameters and storing the results in a format which facilitates their analysis. All authors contributed to analyzing these results. The writing was done by James Bergstra with input from Yoshua Bengio, who also helped direct the course of this research throughout.

5.3 Comments

The following article is a follow-up to a prior submission to the *Nature Neuroscience* journal by authors James Bergstra, Yoshua Bengio and Jerome Louradour. It was submitted to the 26-th *International Conference on Machine Learning* but unfortunately was not accepted for the conference. A revised version of the paper is currently under consideration for the 23-rd annual conference of the *Neural Information Processing Society*.

[1]Theano is an optimizing compiler developed by Olivier Breuleux and James Bergstra at the LISA laboratory (`www.iro.umontreal.ca/~lisa/`). More information on the project can be found at `http://lgcm.iro.umontreal.ca/`.

[2]JobMan, documentation to be made available soon on `http://lgcm.iro.umontreal.ca/`

CHAPTER 6

QUADRATIC POLYNOMIALS LEARN BETTER IMAGE FEATURES

6.1 Abstract

The affine-sigmoidal hidden unit (of the form $s(ax+b)$) is a crude predictor of neuron response in visual area V1. More descriptive models of V1 have been advanced that are no more computationally expensive, yet artificial neural network research continues to focus on networks of affine-sigmoidal models. This paper identifies two qualitative differences between the affine-sigmoidal hidden unit and a particular recent model of V1 response: a) the presence of a low-rank quadratic term in the argument to s, and b) the use of a gentler non-linearity than the tanh or logistic sigmoid. We evaluate these model ingredients by training single-layer neural networks to solve three image classification tasks. We experimented with fully-connected hidden units, as well as locally-connected units and convolutional units that more closely mimic the function and connectivity of the visual system. On all three tasks, both the quadratic interactions and the gentler non-linearity lead to significantly better generalization. The advantage of quadratic units was strongest in conjunction with sparse and convolutional hidden units.

6.2 Introduction

Artificial neural networks are among the earliest machine learning algorithms, and most are inspired by a particular simplification of the biological neuron: that each neuron applies a sigmoidal non-linearity (such as the logistic sigmoid, sigmoid) to an affine transform (parameter vector w, scalar b) of its input vector x:

$$\text{response} = \text{sigmoid}(w \cdot x + b) = \frac{1}{1 + \exp(-w \cdot x - b)}$$

Whereas the exploration of different kinds of models was active in the 1980's, artificial neural network researchers since then have settled on the affine-sigmoid model and the radial basis function (RBF) model [47, 48], which gave rise to Gaussian SVMs [7].

However, the affine-sigmoid model is a crude approximation of real neuron response. Recently, Rust *et al.* [56] described experiments in which they tested for linear and non-linear neuron responses among the simple and complex cells in the early vision system of the macaque monkey. They found that only the simplest cells responded to an input pattern x according to a formula like $\text{sigmoid}(wx+b)$. Their model (Eq. 6.1) fit spike-rate data better by incorporating separate non-linear terms for the excitation (E) and shunting inhibition (S) experienced by each cell.

$$\text{response} = \alpha + \frac{\beta E^{\zeta} - \delta S^{\zeta}}{1 + \gamma E^{\zeta} + \varepsilon S^{\zeta}} \tag{6.1}$$

$$E = \sqrt{\max(0, w'x)^2 + x'V'Vx} \tag{6.2}$$

$$S = \sqrt{x'U'Ux} \tag{6.3}$$

Equation 6.1 looks sigmoidal as a function of E, but the sharpness of the non-linearity is modulated by S. The constant scalar exponent ζ modulates the sensitivity of the function to both E and S. The constant scalars $\alpha, \beta, \delta, \gamma$, and ε control the dynamic range of the function. As in the affine-sigmoid function, a vector of weights w parametrizes a linear axis of increasing cell excitation. Most importantly, the low-rank matrices V and U capture second-order interactions between the neuron inputs by parametrizing subspaces where either positive or negative deviation from a particular value is equivalently significant. Affine-sigmoidal models can only approximate this sort of flexibility with increased depth and many more neurons in intermediate layers.

The use of quadratic functions, or even higher-order polynomials, is not new in neural network research. Minsky and Papert [44] used this idea to define the notion of *problem order*. For example, a first-order unit cannot implement the infamous XOR function, whereas a second-order unit can implement it. Later, Sigma-Pi networks were advanced in PDP-1 [55] as a way to model high-order polynomial computation within a

neuron dendritic tree. Giles and Maxwell [17] explained the utility of second-order and third-order polynomials in terms of their potential for group invariance. Their result is relevant to image processing (in raw raster format): there exist non-trivial second-order polynomials that are invariant to the translation of a subject across a background of zeros. The principle they draw on is that of spatial auto-correlation. Some second-order polynomials can be written as linear functions of the spatial auto-correlation of an image, and those polynomials are invariant to translation across a zero background. In contrast, the only first-order polynomials that are invariant to this sort of translation are zero-order polynomials, i.e., constants.

The difficulty with higher-order units, at least as they have traditionally been implemented, is that their use requires a great deal of space and time. The number of degrees of freedom in a higher-order unit (and amount of CPU time to evaluate the unit) is potentially $O(n^k)$ where n is the number of input dimensions and k the degree. In practice this has been prohibitive [17, 44]. For problems with many input dimensions (such as image classification), even a second-order model is too large. There have been methods for reducing the storage and CPU requirements: for example, by factoring the polynomial [60] or by manually selecting which second-order terms to parametrize [17, 63]. But these approaches suffer from other practical problems: the model in [60] appears difficult to train by gradient descent, and the approach in [63] requires knowledge of which terms to keep.

The formulation of [56] is interesting from a modeling perspective because it is flexible enough to implement the invariances described in [17], and at the same time, it can be made nearly as computationally cheap as the affine-sigmoidal model by choosing very low-rank U and V.

Another interesting aspect of the [56] formulation is the nature of sigmoidal non-linearity, as a function of E. Under a recombination of scalars $\alpha, \beta, \delta, \gamma, \varepsilon$, and S (considered constant) into a, b, c, the non-linearity has the form:

$$\text{response}(E) = a + \frac{E^{\zeta}}{b + cE^{\zeta}} \tag{6.4}$$

Considering the case of $a=0$ and $\zeta=b=c=1$ for simplicity, we can see that this non-linearity approaches its maximal value ($a+1/c=1.0$) more slowly than the tanh function (which is a symmetric equivalent of the logistic sigmoid) approaches its maximum (of 1.0). It approaches more slowly in the sense that the asymptotic limit of the ratio of their derivatives is 0.

$$\lim_{E\to\infty}\frac{\frac{d}{dE}\tanh(E)}{\frac{d}{dE}\left(\frac{E}{1+E}\right)}=\lim_{E\to\infty}\frac{\frac{d}{dE}\left(\frac{2}{1+\exp(-E)}\right)}{\frac{d}{dE}\left(\frac{E}{1+E}\right)}$$
$$=\lim_{x\to\infty}\exp(-x)x^2=0.$$

In order to have this sort of asymptotic behaviour from a feature activation function, we experimented with the following alternative to the tanh function, which we will refer to as the softsign function,

$$\text{softsign}(x)=\frac{x}{1+|x|}\tag{6.5}$$

illustrated in Figure 6.1.

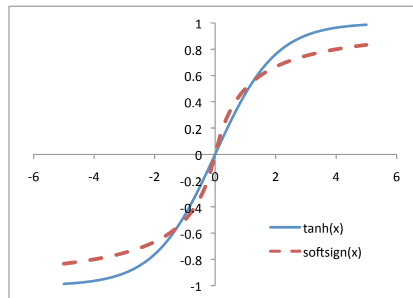


Figure 6.1: Standard sigmoidal activation function (tanh) versus the softsign, which converges polynomially instead of exponentially towards its asymptotes.

Finally, while not explicit in Eq. 6.1, it is well known that the receptive field of simple and complex cells in the V1 area of visual cortex are predominantly local [28]. They respond mainly to regions spanning from about $\frac{1}{4}$ of a degree up to a few degrees. This structure inspired the successful multilayer convolutional architecture of LeCun

et al. [39]. Inspired by their findings, we experimented with local receptive fields and convolutional hidden units.

This paper is patterned after early neural network research: we start from a descriptive (not mechanical) model of neural activation, simplify it, advance a few variations on it, and use it as a feature extractor feeding a linear classifier. With experiments on both artificial and real data, we show that quadratic activation functions, especially with softsign sigmoid functions, and in sparse and convolutional configurations, have interesting capacity that is not present in standard classification models such as neural networks or support vector machines with standard kernels.

6.3 Model and Variations

The basic model we investigate is a neural network based on hidden units (learned features) $h(x)$ in the form of

$$h(x) = s\left(\sum_{k=0}^{K}(A_k \cdot x)^2 + b \cdot x + c\right). \tag{6.6}$$

Here x is an input (such as an image in raster greyscale format), A is a matrix of rank K (row vector k denoted A_k), b is a vector of weights, and c is a threshold constant. We compare the tanh and softsign functions as candidates for s. This equation is simpler than the model of [56], but includes the technique of incorporating second-order interactions via a low-rank matrix A.

The positive semi-definite quadratic interaction matrix $\sum_k A_k' A_k$ can implement the sort of translation-invariant polynomial described by Giles and Maxwell [17] and Reid *et al.* [52]. The low-rank restriction limits the size and complexity of the pattern which can be recognized independently of its position, but with even a rank-2 A, a direction- and position-invariant edge detector is possible. The detector can be illustrated in 1-D with the matrix A such that $A_0 = (1,-1,1,-1,1,-1,\dots)$ and $A_1 = (-1,1,-1,1,-1,\dots)$. As an edge between a black (value 0) and white region (value 1) moves across the 1-D visual field, the response $A_0 \cdot x$ will oscillate between 0 and 1, and the response $A_1 \cdot x$ between

-1 and 0. The pair $(A_0 \cdot x, A_1 \cdot x)$ oscillates along an arc, tracing out a quarter-circle (though the exact curvature will depend on how the edge is rendered when it does not line up with a pixel boundary) with [constant] radius $\sum_{k=0}^{1}(A_k \cdot x)^2$. With higher-rank matrices, larger and more complex shapes can be detected in multiple positions. We do not claim that the supervised learning algorithm described below will learn an exactly invariant function, but these invariant functions are part of the family of functions that the model could learn.

6.3.1 Learning

Despite involving high-order non-linearities, the models presented here can be understood as single hidden-layer neural networks. In each model, the input images ($x \in \mathbb{R}^d$) are mapped to a *feature vector* by some trainable feature extractor ($h(x) \in \mathbb{R}^n$) and then classified by logistic regression, such that $y(x) = b_y + W_y h(x)$.

The prediction $y(x) \in \mathbb{R}^C$ can be interpreted as a vector of discriminant functions for C different classes. It is transformed into a probability distribution over classes by the softmax function given by

$$p(\text{class } i \,|\, x) = \frac{e^{y_i}}{\sum_j^C e^{y_j}}. \tag{6.7}$$

The fitting (learning) of b_y, W_y, and the parameters of h is accomplished by minimizing the average cross-entropy between the target distribution over labels targ and the distribution predicted by the softmax of $y(x)$:

$$\text{loss}(x, \text{targ}) = -\text{targ}' \log y(x). \tag{6.8}$$

Note that in ordinary classification problems such as those considered here targ is a one-of-C vector with a 1 at the position corresponding to the target class for pattern x.

We minimized this loss function using stochastic gradient descent on the feature parameters A, b, c and the logistic regression parameters b_y, W_y as in LeCun *et al.* [37]. In our experiments, we initialized the filter weights (A, b) from a zero-mean normal

distribution, and other weights (c, b_y, W_y) were initialized to zero. The variance of the Normal used for sampling affine parameters b was different from the variance of the Normal used for sampling quadratic parameters A. Both these variances were treated as hyperparameters.

The early-stopping criterion was based on a *best-epoch* heuristic. The validation set was consulted after each epoch to estimate the generalization error of the current classification model. If the generalization error of the current epoch is less than 0.96 of the generalization error at the best epoch, then the current epoch becomes the best epoch. This heuristic was used to reduce computational complexity, by reducing the frequency with which best models needed to be saved. The training procedure was stopped when the current epoch reached 1.75 times the best epoch, or a hard threshold of 800 epochs. While somewhat arbitrary, these heuristics were chosen such that the total training time remained practical (2-3 days at most), while also given enough time for models to escape from local minima. To give the search procedure enough time to get started, a minimum training time of 10 epochs was enforced. The model returned by this procedure is the one with the lowest observed validation-set classification error. Note that on account of the improvement threshold of .96, the returned model is not necessarily one of the best-epoch models.

The structural parameters of the models (number of hidden units, rank of A, step size during learning, initialization) were chosen by grid search and cross-validation. For each data set, and each kind of activation function, we searched for the best values. We tested learning rates in the range of 10^{-5} to 10^{-1}. We tested numbers of hidden units in the range of 100 to tens of thousands. We tested initial scale ranges from 0.01 to 0.1. We tested A of rank 0 (which is to say, no A matrix at all), 2, and 8. Due to the sheer number of experiments which were run, we only used a single random seed in our experiments.

6.3.2 Sparse Structure and Convolutional Structure

To further reduce the number of free parameters in Eq. 6.6 we introduced sparse structure to the image filters (A_k, b), exploiting the 2-D topological structure of image inputs. We know that neighboring pixels have greater dependency and that pixel loca-

tions can even be recovered from the patterns of correlation between pixels [35]. We also know that neurons in the human visual system are sparsely connected, with connections mainly involving neurons associated with spatially neighboring areas of the retina. The best-performing learning algorithms on some vision tasks such as digit classification are convolutional networks [36, 39], in which filters are both spatially sparse and shared across all the different filter locations. This motivated two variants explored here. In the first variant, the filters (both in the quadratic term A and in the linear term b) are spatially sparse. We restrict the rows of A and the linear component b to have non-zero values only in some $m \times m$ pixel image patch (we always chose the same patch for b and every row of A). We did not admit patches that overlapped the image boundary. For an image of $N \times N$ pixels, there are $(N-m+1)^2$ such patches. We thus added network capacity in logical blocks of $(N-m+1)^2$ hidden units called *feature maps* (Figure 6.2). This is unlike a fully-connected neural network, which may have any number of hidden units. The second variant is a convolutional model, i.e., a further restriction of the sparse model, wherein all of the filters in each feature map are constrained to be identical. For all datasets, we chose a value of $m = 5$. Hidden units for sparse and convolutional models therefore operated on 5×5 image patches.

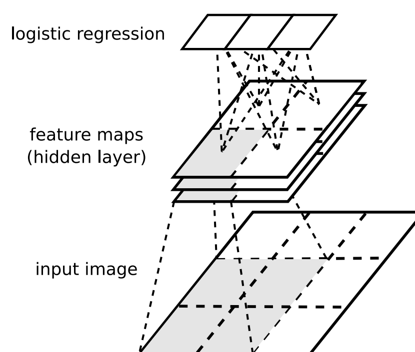


Figure 6.2: In the sparse and convolutional architectures, the hidden layer was arranged logically into feature maps. Each feature map corresponded to either the convolution of a single filter with the input image (convolutional), or the application of a 5×5 filter at every input position (sparse). Model capacity was adjusted via the number of feature maps. Classification was done by logistic regression on the concatenation of all feature maps.

6.3.3 Support Vector Machines

We compared our models with Support Vector Machines (SVMs). SVMs were used as multi-class classifiers using the one-against-all and the one-against-one approaches implemented in libSVM. [9]. Several popular kernels were tried–the linear, polynomial and Gaussian kernels– but the Gaussian kernel was consistently best. The validation data set was used to choose the kernel, the kernel's parameters, and the margin parameter C. Inputs to the SVM were scaled to have mean zero and unit variance. To find the best parameters C and γ for the Gaussian kernel $k(x,y) = e^{-\gamma||x-y||^2}$ we used a multi-level grid search.

6.4 Data Sets

6.4.1 Shape Classification

The SHAPESET data set contains greyscale images of size 32×32 showing a single flat-shaded shape on a uniform background (Figure 6.4.1, left). The examples contain regular shapes: circles, squares, and equilateral triangles. Images were generated by varying the type of shape, the position, size, orientation, and greyscale levels of the foreground and background. Each shape is constrained to fit entirely within the image, and to be large enough that its class can be distinguished at 32×32 resolution. Although we have not measured it formally, we estimate that the Bayes error rate of this classification task is less than 1%. For our experiments, we generated 10,000 training examples, 5,000 validation examples, and 5,000 test examples.

While some of these variabilities could be removed through trivial front-end processing, we preferred not to remove them. This data set represents a stepping stone towards real-world clutter and irregularity. Clutter, irregularity in object surfaces, lighting effects, and all the other subtleties whose omission make SHAPESET images look artificial are factors of variation that would only make classification harder than it already is in SHAPESET. Despite its visual simplicity, SHAPESET images include a number of relevant factors of variation that interact to make a difficult classification problem.

Figure 6.3: Sample inputs from SHAPESET (left) and FLICKR (center) and MNIST (right). SHAPESET contains circles, squares, equilateral triangles (image size 32×32). FLICKR contains 10 types of subject such as “animal”, “bird”, “beach”, and “city” (image size 75×75). MNIST contains handwritten digits 0-9 (image size 28×28).

6.4.2 Flickr

The images in the FLICKR data set (Figure 6.4.1, center) were collected from Flickr®[1] using the public API. Ten of the tags identified as most popular by Flickr were used as query terms and one thousand 75×75 images were downloaded for each tag and transformed to greyscale. The tags used were concrete nouns: “animals”, “baby”, “balloon”, “band”, “beach”, “bird”, “car”, “cat”, “church”, and “city”. The returned images correspond to the most relevant thousand (for a given tag) as decided by Flickr. We divided the image set into three: the training set contains 7,500 images, the validation set 1,000 and the test set 1,500.

To estimate the inherent difficulty of this classification problem, a small-scale experiment was conducted using Amazon’s Mechanical Turk service, in which human subjects were asked to classify 50 images into one correct category. These guesses were marked correct when they matched the query term used to download the image. Three subjects classified each image and a majority vote was used. The human error rate was about 20%. Most errors are the result of the ambiguous nature of some images (e.g., cat vs. animal or church vs. city). While this may render the dataset ill-suited for purposes of benchmarking, these ambiguities should not adversely affect the relative performance of each model tested.

[1] http://www.flickr.com

6.4.3 Digit Classification

We used MNIST (Figure 6.4.1, right) as a digit classification task. We used the full database, of 50,000 training images, 10,000 validation examples, and 10,000 test examples. The digits were left in their original format of 28×28 greyscale pixels [39].

6.5 Results

The first series of experiments was directed at evaluating the usefulness of the quadratic terms (the A matrix in Eq. 6.6). To that end, each architectural variant (dense, sparse, convolutional) was paired with a tanh sigmoid function and optimized for each data set for three amounts of quadratic capacity (affine, rank-2 A, and rank-8 A). The generalization error of the best model for each combination is listed in Table 6.1. In MNIST the rank-2 model outperformed the rank-8 model for all three architectures. In SHAPESET, the rank-8 model was best for dense and convolutional architectures, but rank-2 was best for the sparse architecture. In FLICKR, the rank-2 model was best for the dense architecture, but the rank-8 model was best for the sparse and convolutional variants. The affine model was not the best for any filter type or any data set. The rank-2 model in particular was always better than the affine one.

With regards to the different filter structures (also Table 6.1) there was not a consistent ranking across $K=0$, $K=2$, and $K=8$. In the affine case ($K=0$), the dense filters were consistently competitive, and clearly the best for SHAPESET. In the rank-2 case, the convolutional filters were best for MNIST and SHAPESET, but slightly worse than the dense filters on FLICKR. In the rank-8 case, the convolutional filters were clearly the best for all the image data sets. Sparse filters were consistently poorer than dense filters and convolutional ones, even though they shared the same sparsity pattern as the convolutional filters. This finding suggests that statistical efficiency is increased in these image-processing problems by sharing filters across the visual field.

To put these error rates into perspective, we compared them with an SVM model trained with the same inputs and outputs. A Gaussian SVM achieved an error rate of 1.4% on MNIST, which is better than the affine neural network, but slightly worse than

Table 6.1: The generalization error of models based on a tanh sigmoidal non-linearity. The codes used for the filter types are dense (D), sparse (S), and convolutional (C). The three columns of error rates are for affine models ($K = 0$), models whose A matrix has rank 2 ($K = 2$), and models with A of rank 8 ($K = 8$). The affine model is always worse than the best quadratic model, this is statistically significant (p-value under .05) in all cases except for the sparse models on FLICKR. SVM results are included for reference.

Data set	Filter type	Generalization Error (%) $K=0$	$K=2$	$K=8$	SVM
MNIST	D	1.9	**1.6**	1.7	1.4
	S	2.0	**1.6**	1.8	
	C	1.6	**1.3**	1.4	
SHAPESET	D	19.9	15.1	**11.8**	29.4
	S	40.4	**18.2**	19.6	
	C	57.0	12.1	**10.3**	
FLICKR	D	83.9	**79.9**	80.6	76.6
	S	81.9	81.5	**80.7**	
	C	87.8	80.9	**78.7**	

the single-layered quadratic network with convolutional features. [2]

A Gaussian SVM achieved an error rate of 29.6% on the SHAPESET task, which is worse than an affine neural network (19.9%), and much worse than a convolutional quadratic network (10.3%). On the FLICKR task, a Gaussian SVM achieved an error rate of 76.6%, which compares favorably to the best network that we tried (78.4%).

6.5.1 Translation Invariance

To investigate the hypothesis introduced above (following Adelson and Bergen [1], Giles and Maxwell [17]) that the advantage of quadratic units is related to their capacity for translation invariance, the following experiment was performed. The best models with and without quadratic units were identified for each of the three data sets. The best MNIST model with quadratic units had a rank-2 A matrix. The best SHAPESET and FLICKR models with quadratic units had rank-8 A matrices. Each of these models was re-evaluated on artificially translated training and validation sets, in which each of the examples was shifted by one pixel sideways, either horizontally, vertically, or both.

[2] Result from `http://yann.lecun.com/exdb/mnist/`.

When shifting, edge pixels opposite the direction of the shift were left unchanged.

Table 6.2: Artificially translating the data sets by one pixel in each direction illustrates the sensitivity of each model's error rate to object positioning. In each 3×3 grid, the center corresponds to the original data. For example, each upper left score corresponds to translation by 1 pixel to the left and 1 pixel upward. (Classification error in %)

	Training data						Validation data					
	Quadratic			Affine			Quadratic			Affine		
	8	3	4	10	4	6	8	3	4	9	4	5
MNIST	3	0	3	3	0	4	3	1	3	4	2	4
	5	3	8	6	5	9	4	3	6	5	4	8
	9	7	9	15	12	14	12	11	13	17	17	19
SHAPESET	7	3	7	13	3	12	11	10	12	17	17	18
	9	8	10	18	14	18	12	11	12	18	17	18
	74	72	76	83	83	83	81	81	81	83	83	84
FLICKR	76	75	77	83	82	82	81	79	81	83	83	84
	76	77	77	83	82	82	80	80	80	83	83	84

The classification accuracies of these best models on the artificially translated data are enumerated in Table 6.2. Generally, the performance of all models deteriorated substantially when objects were translated by just one pixel. The deterioration was more prominent in models which were achieving some success (those trained on MNIST and SHAPESET). In SHAPESET's training data, the quadratic model was more robust to translation; the worst quadratic model deterioration was from 3% to 10%, whereas the worst affine model deterioration was from 3% to 18%. In MNIST's training data, the quadratic model was slightly more robust to translation; the worst quadratic model deterioration was from 0% to 8%, whereas the worst affine model deterioration was from 0% to 10%. In FLICKR's training data and in all the validation data sets, the quadratic model was not more robust to translation than the affine one.

6.5.2 Tanh vs. Softsign

To compare the tanh and softsign sigmoid functions as transfer functions in neural networks, we restricted our family of models to those with dense filters. We varied again the number of quadratic terms K, and evaluated both tanh and softsign as the activation

Table 6.3: Generalization obtained when using tanh vs. softsign sigmoid functions for different ranks of quadratic interaction ($K = 0$, $K = 2$, $K = 8$). In every case but one (FLICKR, $K = 0$), the softsign error rate is lower than the tanh with p-value of .05.

Data set	Sigmoid type	Generalization Error (%) $K = 0$	$K = 2$	$K = 8$
MNIST	tanh	1.9	1.6	1.7
	softsign	**1.8**	**1.5**	**1.5**
SHAPESET	tanh	19.9	15.1	11.8
	softsign	**16.6**	**12.9**	**9.5**
FLICKR	tanh	**83.9**	79.9	80.6
	softsign	85.5	**78.8**	**78.4**

function s in Eq. 6.6. The generalization errors that were the result of optimizing these models for each data set are listed in Table 6.3. For the affine model, softsign was better on the MNIST and SHAPESET data sets but worse on FLICKR. For both quadratic models ($K = 2$, $K = 8$), softsign always resulted in better generalization. In every comparison between softsign and tanh their ordering is statistically significant at a p-value of .05.

6.6 Discussion

While much current research on learning in artificial neural networks deals with affine sigmoidal models, efforts to model neuron responses to stimuli have resulted in newer and qualitatively different models. Rust *et al.* [56] have put forward a more accurate biological model of visual area V1 that involves quadratic interactions between inputs as well as a different form of non-linearity. Our experiments evaluated the utility of these two elements of the Rust *et al.* [56] V1 response model:

1. the presence of a low-rank quadratic term: it was found to improve generalization on all three tested data sets;

2. the presence of a gentler, less saturating non-linearity (that converges polynomially rather than exponentially to its asymptotes): it was found to improve generalization on most of the settings tested.

We evaluated these elements in the context of fully-connected (dense), sparse, and convolutional single hidden-layer networks.

We found that the low-rank quadratic term was helpful in all three of our learning tasks. On MNIST the rank-2 quadratic models were best, and for SHAPESET and FLICKR the rank-8 quadratic models were best. For all data sets the affine models performed poorer than the quadratic models. When using quadratic units in sparse and convolutional filter configurations, the advantage of quadratic interactions was even greater than in the case of fully-connected units. Our best results were realized with quadratic interactions and convolutional filters. We conclude that the quadratic term is a useful ingredient to a neural network for classifying images.

The work of Adelson and Bergen [1], Giles and Maxwell [17] and Reid *et al.* [52] suggests that the value of the quadratic terms in our hidden units is that they facilitate the learning of translation-invariant functions. We tested this hypothesis by artificially translating the training and test sets and evaluating the most successful models for each data set. Contrary to the prediction of the translation-invariance hypothesis, we found that both quadratic and affine models were quite sensitive to our artificial single-pixel translations. Heeger [21] has argued that a model similar to the ones presented here implements contrast and luminance normalization, but an investigation of that hypothesis remains future work.

The gentler sigmoidal non-linearity (*softsign*) was helpful on all data sets, on both affine and quadratic models. The only case when it did not outperform the *tanh* sigmoid was in the case of the affine model on FLICKR (when both kinds of sigmoid achieved dismal performance). We conjecture that the advantage of the *softsign* is related to its gradient, which is larger than *tanh*'s for almost all the real domain. A larger gradient would reduce the severity of plateaus in the loss function, and yield a clearer learning signal for stochastic gradient descent.

Finally, the results obtained on FLICKR stand out as being very poor when compared to MNIST and SHAPESET. No doubt, this is due in part to the increased complexity of the task. Natural images are infinitely more complex than the images found in MNIST and SHAPESET. Solving such a classification task may thus require a deeper architec-

ture to deal with the many factors of variation. That being said, the 80% error rate on FLICKR is still much greater than the 66% error reported on Caltech-256 [18], a natural image dataset containing an even greater number of categories. This suggests other factors may be at play. First, the collection procedure for FLICKR was entirely automated and was thus much more fragile compared to that used for Caltech-256. Objects may thus be sub-optimal in describing their visual category and may also appear in significant clutter. Also, the mean number of images per category varies greatly between both datasets: 1000 for FLICKR versus 119 for Caltech-256. As such, the Caltech dataset may be more prone to the statistical biases reported in [46].

CHAPTER 7

OVERVIEW OF THE THIRD PAPER

Tempered Markov Chain Monte Carlo for training of Restricted Boltzmann Machines.

Desjardins, G., Courville, A., Bengio, Y., Vincent, P. and Delalleau, O.

Technical Report 1345, Dept. IRO, Université de Montréal

Submitted to *Neural Information Processing Systems (NIPS)*, 2009 (rejected)

Revised manuscript submitted to *Artificial Intelligence and Statistics (AISTATS)*, 2009

7.1 Context

In the following article, we depart slightly from the realm of convolutional neural networks to focus on the training algorithm of RBMs.

Since the publication of [26], Contrastive Divergence has become the learning algorithm of choice for RBMs. When training them as part of Deep Belief Networks, CD achieves state of the art performance on various classification tasks [26, 33, 42] and also leads to good generative models [26]. However, a standalone RBM does not make for a good generative model when trained with CD. This issue was first reported in [64] and resulted in an alternative training algorithm called Persistent Contrastive Divergence (PCD), which samples negative particles from a chain whose state is persistent (i.e the chain is not longer initialized with a training example at every weight update). While the technical details are reserved for Chapter 8, Tieleman [64] showed that this resulted in models with higher likelihood when compared to CD-1 and CD-10, at the price of a slower convergence. The issue of convergence speed was later addressed by Tieleman and Hinton [65] who proposed a variant called PCD with fast-weights.

The article presented in the following chapter follows in these footsteps. This work emerged from an initial evaluation of PCD and FPCD for training Convolutional DBNs. The limitations of these algorithms motivated the development of this novel training

method for RBMs.

7.2 Contributions

The following article compares recently proposed training algorithms for RBMs: mainly CD, PCD and FPCD. As in [64, 65], we investigate the relationship between the learning process and the mixing of the Markov chain in the negative phase. By focusing on this dynamic and how each algorithm molds the energy landscape, we show that using CD in the unsupervised learning phase can lead to a degeneracy where the energy is lowered for training data, but raised in its immediate vicinity. We also show that while PCD fixes this issue by exploring the energy surface globally, the Markov chain used in the negative phase still has the potential to get trapped in regions of high-probability, at which point the parameter updates will deviate from the true gradient. FPCD provides a mechanism for escaping from these regions, however the negative samples used to determine the gradient are not true samples of the model.

In the following paper, we introduce a novel training algorithm for RBMs, based on the tempered Markov Chain Monte Carlo (MCMC) sampling algorithm. By running multiple Markov chains at different temperatures, tempered MCMC affords us with a robust mechanism for exploring the energy landscape which in turn, leads to significantly better generative models. This is shown both through visualization and estimations of the log-likelihood.

In terms of the authors' contributions, the original idea of using tempered MCMC in the negative phase of PCD was that of Aaron Courville. The technical work (i.e re-implementing CD, PCD and FPCD, along with developing the tempered MCMC based approach) was done by myself, in strong collaboration with A. Courville and Yoshua Bengio. These authors were also heavily involved in the analysis of each algorithm's behaviour and in drawing out the main conclusions of the paper. In the original submission (see comments section), section 8.5.5 was mostly the work of Pascal Vincent (both technical and writing) and similarly, section 8.5.4 the work of Olivier Delalleau. The rest of the article was a collaboration between Yoshua Bengio (sections 8.1-8.3), Aaron

Courville (section 8.4) and myself (sections 8.5-8.5.3).

7.3 Comments

Since the original NIPS submission, the following article has gone through several changes. A few sections were reworked in order to address the feedback from the reviewers and sections 8.5-8.5.3 rewritten to help clarify our findings. The experiments of section 8.5.5 were also redone to stream-line the model selection procedure to account for the choice of sampling procedure. This resulted in new estimates of the log-likelihood of each model but did not change the main conclusions of the paper. A more in-depth discussion of these results was also added to this section.

CHAPTER 8

TEMPERED MARKOV CHAIN MONTE CARLO FOR TRAINING OF RESTRICTED BOLTZMANN MACHINES

8.1 Abstract

Alternating Gibbs sampling is the most common scheme used for sampling from Restricted Boltzmann Machines (RBM), a crucial component in deep architectures such as Deep Belief Networks. However, we find that it often does a very poor job of rendering the diversity of modes captured by the trained model. We suspect that this hinders the advantage that could in principle be brought by training algorithms relying on Gibbs sampling for uncovering spurious modes, such as the Persistent Contrastive Divergence algorithm. To alleviate this problem, we explore the use of tempered Markov Chain Monte-Carlo for sampling in RBMs. We find both through visualization of samples and measures of likelihood that it helps both sampling and learning.

8.2 Introduction and Motivation

Restricted Boltzmann Machines [15, 23, 62, 68] have attracted much attention in recent years because of their power of expression [34], because inference (of hidden variables h given visible variables x) is tractable and easy, and because they have been used very successfully as components in deep architectures [3] such as the Deep Belief Network [26]. Both generating samples and learning in most of the literature on Restricted Boltzmann Machines (RBMs) rely on variations on alternating Gibbs sampling, which exploits the bipartite structure of the graphical model (there are links only between visible and hidden variables, but not between visible or between hidden variables). RBMs and other Markov Random Fields and Boltzmann machines are energy-based models in which we can write $p(x) \propto e^{-\mathbb{E}(x)}$. The log-likelihood gradient of such models contains two main terms: the so-called positive phase contribution tells the model to decrease the energy of training example x and the so-called negative phase contribution tells the model

to increase the energy of all other points, in proportion to their probability according to the model. The negative phase term can be estimated by Monte-Carlo if one can sample from the model, but exact unbiased sampling is intractable, so different algorithms use different approximations.

The first and most common learning algorithm for RBMs is the Contrastive Divergence (CD) algorithm [22, 23]. It relies on a short alternating Gibbs Markov chain starting from the observed training example. This short chain yields a biased but low variance sample from the model, used to push up the energy of the most likely values (under the current model) near the current training example. It carves the energy landscape so as to have low values at the training points and higher values around them, but does not attempt to increase the energy (decrease the probability) of far-away probability modes, thereby possibly losing probability mass (and likelihood) there. After training an RBM with CD, in order to obtain good-looking samples, it is customary to start the Gibbs chain at a training example. As we find here, a single chain often does not mix well, visiting some probability modes often and others very rarely, so another common practice is to consider in parallel many chains all started from a different training example. However, this only sidesteps the problem of poor mixing.

The Persistent Contrastive Divergence (PCD) algorithm [64] was proposed to improve upon's CD limitation (pushing up only the energy of points near training examples). The idea is to keep a Markov chain (in practice several chains in parallel, for the reasons outlined above) to obtain the negative samples from an alternating Gibbs chain. Although the model is changing while we learn, we do not wait for these chains to converge after each update, with the reasoning that the parameter change is minor and the states which had high probability previously are still likely to have high probability after the parameter update.

Fast PCD [65] was later proposed to improve upon PCD's ability to visit spurious modes. Two sets of weights are maintained. The "slow weights" w_m represent the standard generative model. They are used in the positive phase of learning and to draw samples of the model using a regular alternating Gibbs chain. The negative phase however, uses an additional set of "fast weights". Negative particles are samples from a

persistent Markov chain with weights $w_m + v_m$, which creates a dynamic overlay on the energy surface defined by the model. Mixing is facilitated by using a large learning rate for parameters v_m, which is independent from that used for w_m (slow weights can therefore be fine-tuned using a decreasing learning rate with no impact on mixing). The fast weights v_m are pushed strongly towards zero with an L2 penalty ($0.05||v||^2$ in the pseudo-code provided in Tieleman and Hinton [65]), ensuring that their effect is only temporary. Both w_m and v_m are updated according to the sampling approximation of the log-likelihood gradient. Tieleman and Hinton [65] report substantial improvements in log-likelihood with FPCD in comparison to PCD and CD.

8.3 RBM Log-Likelihood Gradient and Contrastive Divergence

We formalize here the notation for some of the above discussion regarding RBMs and negative phase samples. Consider an energy-based model $p(s) \propto e^{-\mathbb{E}(s)}$, with $s = (\mathbf{x}, \mathbf{h})$, and marginal likelihood $p(\mathbf{x}) = \sum_{\mathbf{h}} e^{-\mathbb{E}(\mathbf{x},\mathbf{h})} / \sum_{\mathbf{x},\mathbf{h}} e^{-\mathbb{E}(\mathbf{x},\mathbf{h})}$. The marginal log-likelihood gradient with respect to some model parameter w_m has two terms

$$\frac{\partial \log p(\mathbf{x})}{\partial w_m} = -\sum_h P(h|x) \frac{\partial \mathbb{E}(s)}{\partial w_m} + \sum_s P(s) \frac{\partial \mathbb{E}(s)}{\partial w_m} \tag{8.1}$$

which are respectively called the positive phase and negative phase contributions[1]. Consider a Markov random field defined by statistics g_m, i.e.

$$p(s) \propto e^{-\sum_m w_m g_m(s)}.$$

Then the gradients in Eq. 8.1 are easily computed by

$$\frac{\partial \mathbb{E}(s)}{\partial w_m} = g_m(s).$$

In the following we consider RBMs with binary units x_j and h_i, and energy function $E(s) = -\mathbf{h}'W\mathbf{x} - \mathbf{h}'\mathbf{b} - \mathbf{x}'\mathbf{c}$. Here, the statistics of interest are $h_i x_j$, h_i and x_j, and we

[1]One should be careful that the signs of these terms in Eq. 8.1 do not match their name.

associate them with the parameters W_{ij}, b_i, and c_j respectively.

The Contrastive Divergence (CD) algorithm [22, 23] consists in approximating the sums in Eq. 8.1 by stochastic samples, i.e. in updating parameter w_m by

$$w_m \leftarrow w_m - \varepsilon \left(\frac{\partial \mathbf{E}(\mathbf{x}, \tilde{\mathbf{h}}_1)}{\partial w_m} - \frac{\partial \mathbf{E}(\tilde{\mathbf{x}}_k, \tilde{\mathbf{h}}_{k+1})}{\partial w_m} \right)$$

where $\tilde{\mathbf{h}}_{t+1}$ is sampled from the model conditional distribution $P(\mathbf{h}|\tilde{x}_t)$ (denoting by $\tilde{x}_0$ the training sample x used to initialize the Gibbs chain), $\tilde{\mathbf{x}}_t$ is sampled from $P(\mathbf{x}|\tilde{h}_{t-1})$, and ε is the learning rate of the update. Here, k is the number of alternating steps performed in the Gibbs chain: typically one uses $k = 1$ for efficiency reasons. This works well in practice, even though it may not be a good approximation of the log-likelihood gradient [4, 8].

8.4 Tempered MCMC

Markov Chain Monte Carlo methods provide a way of sampling from otherwise unmanageable distributions by means of sampling from a sequence of simpler *local* distributions. Despite the correlations induced between neighboring samples in the sequence, provided some very general conditions of the resulting Markov chain (such as ergodicity) are satisfied, samples are assured to converge to the target distribution.

However, they suffer from one very important drawback. Because these methods are based on local steps over the sample space, they are subject to becoming "stuck" in local maximum of probability density, over-representing certain modes of the distribution while under-representing others.

Parallel Tempering MCMC is one of a collection of methods (collectively referred to as Extended Ensemble Monte Carlo methods [30]) designed to overcome this shortcoming of standard MCMC methods. The strategy is simple: promote mixing between multiple modes of the distribution by drawing samples from smoothed versions of the target distribution. Provided the topology of the distribution is sufficiently smoothed, the local steps of standard MCMC methods are then able to leave the local regions of high

probability density to more fully explore the sampling space.

Consider the target distribution from which we wish to draw well mixing samples, given by:

$$p(x) = \frac{\exp(-\mathbb{E}(\mathbf{x}))}{Z}. \tag{8.2}$$

We create an extended system by augmenting the target distribution with an indexed temperature parameter:

$$p_{t_i}(x) = \frac{\exp(-\mathbb{E}(x)/t_i)}{Z(t_i)} \tag{8.3}$$

At high temperatures ($t_i >> 1$), the effect of the temperature parameter is to smooth the distribution, as the effective energies become more uniform (uniformly zero) over the sampling space.

In the case of Parallel tempering, the strategy is to simulate from multiple MCMC chains, each at one of an ordered sequence of temperatures t_i from temperature $t_0 = 1$ that samples from the distribution of interest (the target distribution) to a high temperature $t_T = \tau$, ie.

$$t_0 = 1 < t_1 < \cdots < t_i < \cdots < t_{T-1} < t_T = \tau.$$

At the high temperatures the chain mixes well but is not the distribution in which we are interested, so the following question remains: how do we make use of the well mixing chains running at high temperatures to improve sampling efficiency from our target distribution at $t_0 = 1$? In parallel tempering this question is addressed via the introduction of cross temperature state swaps. At each time-step, two neighbouring chains running at temperature t_k and t_{k+1} may exchange their particles x_k and x_{k+1} with an exchange probability given by:

$$r = \frac{p_k(x_{k+1})p_{k+1}(x_k)}{p_k(x_k)p_{k+1}(x_{k+1})} \tag{8.4}$$

For the family of Gibbs distribution (in which we are particularly interested in), this boils down to:

$$r = exp((\beta_k - \beta_{k+1}) \cdot (E(x_k) - E(x_{k+1}))), \tag{8.5}$$

where β_k is the inverse temperature parameter.

It is straightforward to see how this algorithm can be applied to the training of RBMs. Instead of running a single persistent Markov Chain as in PCD, multiple chains are run in parallel, each at their own temperature t_i. For each gradient update, all chains perform one step of Gibbs sampling after which state swaps are proposed between all neighbouring chains in a sequential manner. In this paper, swaps were proposed from high to low temperatures (between temperatures t_i and t_{i-1}), as a way to encourage the discovery of new modes. Empirical evidence seems to suggest that this ordering is not crucial to performance however. The negative particle used in the gradient update rule of Eq. 2.17 is then the particle at temperature t_0.

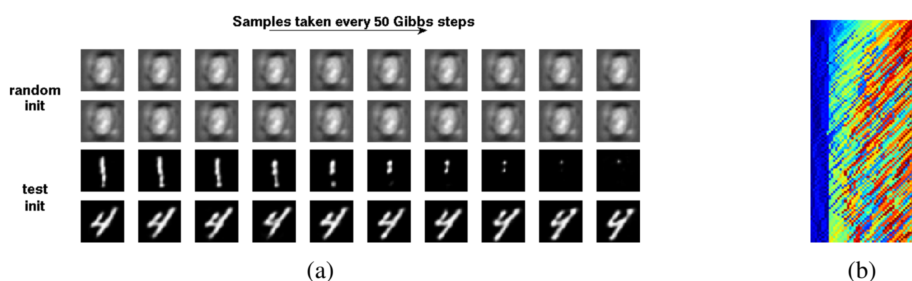
8.5 Experimental Observations

In [65], Tieleman and Hinton [65] highlight the importance of good sampling during the negative phase of RBM training. Without good mixing, negative particles can adversely affect the energy landscape by getting trapped in regions of high-probability and raising the energy level at that mode. In the extreme setting of a distribution containing few but well defined modes, combined with a high-learning rate, negative particles have the potential to pool together and cohesively undo the learning process. In this section, we will investigate the relation between mixing and learning for the most popular RBM training algorithms. The conclusions drawn from this analysis will serve as justification for the tempered MCMC method we propose.

8.5.1 CD-k: local learning ?

Because CD only runs the Markov Chain for a small number of steps, it is very susceptible to the mixing rate of the chain. Early in training, mixing is good since weights are initialized to small random values. As training progresses however, weights become larger and the energy landscape more peaked. For a fixed value of k, mixing thus degrades over time and leads to negative samples being increasingly correlated with training data. This can lead to a degeneracy where the energy of training examples

is lowered but increased in the immediate proximity, in effect forming an energy barrier around the wells formed by the data. When sampling from the resulting model, a Markov Chain initialized with a random state will thus fail to find the high-probability modes as the energy barrier defines a boundary of low-probability. This can be observed in Fig. 8.1(a). In this figure, each row represent samples from separate chains, with samples shown every 50 steps of Gibbs sampling. The top two chains are initialized randomly while the bottom two chains were initialized with data from the test set. The top chains never converge to a digit while the bottom chains exhibit fairly poor mixing.



(a) (b)

Figure 8.1: (a) Gibbs sampling from an RBM trained with CD, starting with a random initialization (2 top chains) vs. initializing with test images (b) Cross-temperature state swaps during MCMC sampling of an RBM trained with CD. Each color represents a particle originating at temperature t_k at time $t = 0$, as it jumps from one temperature to another. Temperatures in the range $[t_0, t_T]$ are shown from left to right. A bottleneck is clearly visible in the lower-temperature range.

This phenomenon can also be confirmed by using the tempered MCMC procedure to sample from an RBM trained with CD. Fig.8.1(b) shows a mixing plot of tempered MCMC sampling using 50 chains and a maximum temperature of 10. The first line of the image shows the state of the chains at the start of training, with each color representing a single particle x_i native to temperature t_i. Low temperatures are shown on the left and high temperatures on the right. Each subsequent line tracks the movement of the original particle through time. High acceptance ratios would cause the colors to become entangled after a certain number of iterations. For an RBM trained with CD however, there is a clear bottleneck in the lower temperature ranges, through which high energy particles do not go through. This gives more credibility to our theory of how CD-1

learning modifies the energy landscape. Positive training data pushes down hard on the energy landscape but only locally, while sharp ridges are formed around the wells by the negative phase. With enough time and tempered MCMC chains, one should theoretically converge on these wells. Our experience suggests that this does not happen in practice, which speaks to the sharpness of the energy landscape formed by CD.

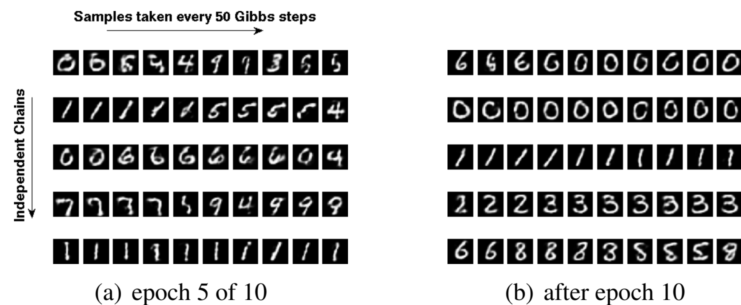
8.5.2 Limitations of PCD

[64] introduced persistent CD and recently PCD with fast-weights as ways to address the issue of bad mixing during the negative phase of learning. Maintaining a persistent chain has the benefit that particles explore the energy landscape more globally, pulling up the energy as they go along.

In Figure 8.2(a), we confirm that samples drawn during learning mix fairly well early on in training. The samples shown were collected midway through the learning procedure (epoch 5 of 10), from an RBM with 500 units trained with persistent CD. Fig. 8.2(b) tells a different story however. These samples were obtained after epoch 10, at which point learning was effectively stopped (learning rate of 0). We can see that mixing has degraded significantly.

We believe two factors are responsible for this. As mentioned previously, early on in training the energy landscape is smooth and mixing is good. With each weight update however, the positive phase creates wells which become progressively deeper. Eventually, negative samples become trapped in low-probability states and are unable to explore the energy landscape. This results in parameter updates which deviate from the true-likelihood gradient. This is very problematic since there is no early-stopping heuristics which can be used to stop learning in time to avoid this bad mixing.

The other factor to take into account is the effect of "fast weights" explored in [65]. The sampling procedure used during learning encourages the particles to mix, since each update renders the state of the negative particles less probable. Negative particles are therefore "encouraged" to move around in input space. The fast-weights algorithm (FPCD) exploits this idea by perturbing the model, but in a way which only affects the negative phase. Unfortunately, FPCD generates spurious samples as shown in Figure 8.3.



(a) epoch 5 of 10 (b) after epoch 10

Figure 8.2: Negative samples drawn from an RBM trained with PCD, at (a) epoch 5 of 10 (b) after learning is stopped. Notice that once the learning procedure is stopped, the mixing rate of the chains drops dramatically and samples become trapped in a minima of the energy landscape. Each row shows samples drawn from a single Gibbs chain, with 50 steps of Gibbs sampling between consecutive images.

These in effect, represent the paths which a negative particle must take to jump from one mode to the next. Not being true samples of the model however, they undoubtedly hurt the learning process since the parameter updates will not follow the true gradient.

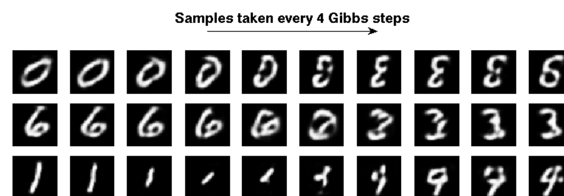
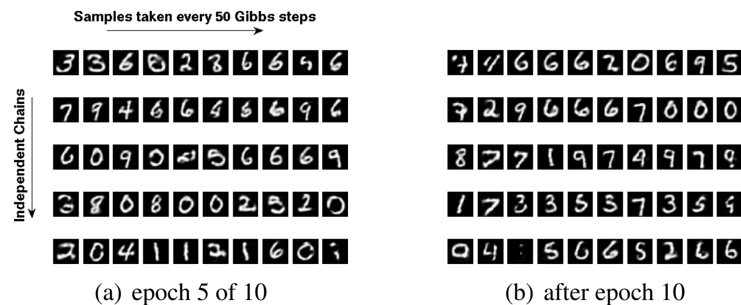


Figure 8.3: Samples obtained using the fast-weight sampling procedure.

8.5.3 Tempered MCMC for training and sampling RBMs

We now use the same experimental protocol to show that our RBM trained with tempered MCMC addresses the above problems in a straight-forward and principled manner. Figure 8.4 clearly shows that samples obtained from the fully trained model (once learning is stopped) mix extremely well.

The maximum temperature t_T and number of parallel chains to use were chosen somewhat arbitrarily. The maximum temperature was chosen by visualizing the samples from the top-most chain and making sure that the chain exhibited good mixing.



(a) epoch 5 of 10

(b) after epoch 10

Figure 8.4: Negative samples drawn from an RBM with 500 hidden units, trained with tempered MCMC, (a) at epoch 5 of 10 (b) after learning is stopped. It is clear that the resulting model exhibits much better mixing than in Fig. 8.2(b). Again, each row shows samples drawn from a single Gibbs chain, with 50 steps of Gibbs sampling between consecutive images.

The number of chains was chosen to be large enough such that the mixing plot of Figure 8.1(b) showed (i) a large number of cross-temperature state swaps and (ii) that a single particle x_i, on average, visited temperatures in the range $[t_0, t_T]$ with equal proportions.

With regards to the learning rate, we found that a decreasing learning rate schedule was necessary for the model to learn a good generative model. This should not be surprising, as it seems to echo the theoretical findings of Younes [69], which outlines a proof of convergence for profiles of the type a/t with small enough a. Empirically, we found that decreasing the learning rate linearly towards zero also worked well.

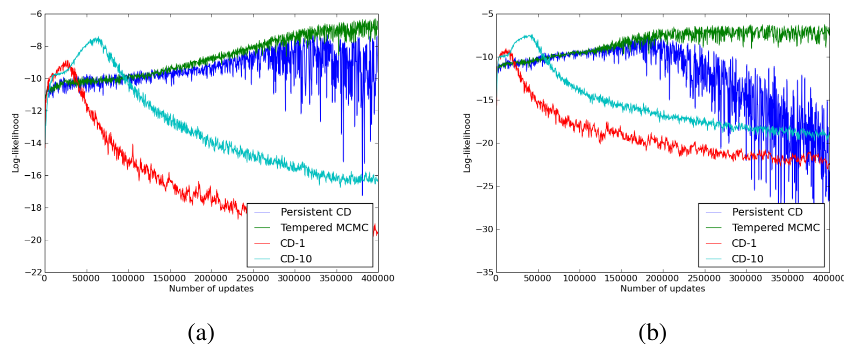
8.5.4 Tempered MCMC vs. CD and PCD

Here we consider a more quantitative assessment of the improvements offered by using tempered MCMC, compared to regular CD and PCD. In order to compute the exact log-likelihood of our models, we use a toy dataset of 4x4 binary pixel images. Half of the training examples are images which represent two black lines over a white background, such that pixels of each line are not adjacent to the other line's pixels. The lines are made of two pixels and the image is assumed to have a torus structure. The second half of the training examples are the same samples where black and white have been swapped. In total, this yields 320 valid images (out of the 2^{16} possible images)

which are all used as training data.

For all three algorithms, we performed 400,000 weight updates using mini-batches of size 8. This means that each training example is presented 10,000 times to each model. Learning rates were either held constant throughout learning or decreased linearly towards zero. No weight decay was used in any of the models. The other hyperparameters were varied as follows:

- number of hidden units in $\{10,20,30,40,50,100,200,300\}$
- initial learning rates in $\{0.1,0.05,0.01,0.005,0.001\}$
- for CD-k, a number k of steps in $\{1,3,5,10,25\}$
- for tempered MCMC, 50 chains spaced uniformly between $t_0 = 1$ and $t_T = 2$.



(a) (b)

Figure 8.5: Evolution of the exact log-likelihood of the training data as a function of epochs. (a)100 hidden units and a learning rate of 0.005 (b)100 hidden units and a learning rate of 0.01. The CD-k and PCD algorithms become unstable after a while, whereas the tempered MCMC method exhibits a much more reliable behaviour.

Figure 8.5 shows typical examples of the behaviour of the three algorithms. What usually happens is that after some time, the poor mixing of the chains used in CD-k and PCD leads to updates which actually *hurt* the modeling ability of the RBM. We observed that this phenomenon became worse as the learning rate increased. In contrast, the tempered MCMC version is much more stable and typically increases the log-likelihood in a pseudo-monotonic manner.

8.5.5 Estimating Likelihood on MNIST

Next we wanted to obtain a similar quantitative measure, that would provide for a more objective comparison of the algorithms considered, on a real-world problem.

An important characteristic of RBMs is that, while computing the exact likelihood under the learnt model is intractable, it is however *relatively* easy to generate samples from it. We thus set to compute a *quantitative* measure that could reflect the "quality" of the sample generation that the various trained models were capable of. More specifically, we want a numerical measure of how close the sample generation by a particular trained RBM is to the "true" but unknown distribution which produced the training samples $\mathscr{D}$. This will allow us to evaluate more objectively to what degree the considered training procedures are able to capture the distribution they were presented.

Notice that by formulating the question in this manner we link a trained model and a sampling procedure. What we evaluate is a (model + sampling-procedure) combination, i.e. the resulting measure depends not only on the model's parameters but also on the particular sampling procedure used after training to generate new samples. While it is natural here to use the same sampling procedure during post-training generation as was used during training, it is also possible to use after training a different procedure than what was used during training, e.g. train using simple CD but then generate using tempered MCMC.

The measure we consider is the average log probability of the samples in a held out test set $\mathscr{D}_{\text{test}}$ under a non-parametric Parzen Windows density estimation $\hat{p}_{\sigma,\mathscr{D}_{\text{s}}}$ based on the samples $\mathscr{D}_{\text{s}}$ generated by a given model that was trained on a train set $\mathscr{D}$. The test set $\mathscr{D}_{\text{test}}$ has n samples $x^{(1)},\ldots,x^{(n)}$ that originate from the same unknown distribution as the data $\mathscr{D}$ used for training. Similarly $\mathscr{D}_{\text{s}}$ has n' samples $s^{(0)},\ldots,s^{(n')}$ generated using a given (model + sampling-procedure).

Formally our sample generation quality measure is:

$$\ell(\mathscr{D}_{\text{test}},\mathscr{D}_{\text{s}}) = \frac{1}{n}\sum_{k=1}^{n}\log\hat{p}_{\sigma,\mathscr{D}_{\text{s}}}(x^{(k)}) \tag{8.6}$$

where $\hat{p}_{\sigma,\mathscr{D}_s}(x^{(k)})$ is the density evaluated at point $x^{(k)}$ obtained with a non-parametric kernel density estimator based on $\mathscr{D}_s$ with hyperparameter σ. i.e. $\hat{p}_{\sigma,\mathscr{D}_s}(\mathbf{x}) = \frac{1}{n'}\sum_{k=1}^{n'} K(\mathbf{x}; s^{(j)})$. We will use, as customary, a simple isotropic Gaussian kernel with standard deviation σ, i.e. $K = \mathscr{N}_{s^{(j)},\sigma}$.

In practice we proceeded as follows:

- Search for a good value of the kernel bandwidth σ. We perform a grid search trying to maximize the log probability of test samples $\mathscr{D}_{\text{test}}$ under $\hat{p}_{\sigma,\mathscr{D}_s}$ i.e. $\sigma \simeq \text{argmax}_{\sigma'} \ell(\mathscr{D}_{\text{test}}, \mathscr{D})$. We then keep this bandwidth hyperparameter fixed.

- Generate $\mathscr{D}_s$ made of n' samples from the considered RBM with the desired sampling procedure. For convenience[2] we choose $n' = 2n$ which is 20,000 for the standard MNIST test set.

- Compute generation quality $\ell(\mathscr{D}_{\text{test}}, \mathscr{D}_s)$ as defined in Eq. 8.6.

- Repeat these last two steps for all models and sampling procedures under consideration.

Table 8.1 reports the generation quality measure obtained for different combinations of training procedure (yielding a trained model) and post-training sample generation procedure. Model selection was performed by selecting, for each combination of (training procedure, sampling algorithm), the hyperparameters leading to the best likelihood. Including the sampling procedure in the model selection process allows for a fair comparison of training algorithms. Hyperparameters are thus selected such that the trained model is compatible with the sampling procedure.

As we can see, the tempered models have a significantly higher likelihood than all the other training algorithms, regardless of sampling procedure. It is interesting to note that in this case, sampling with tempered MCMC did not result in a higher likelihood. This may indicate that the parameter σ should be optimized independently for each model

[2]using the same sizes $n' = n$ allows us to compute the reverse $\ell(\mathscr{D}_s, \mathscr{D}_{\text{test}})$ without needing to search for a different hyperparameter σ.

Table 8.1: Log probability of test set samples under a non-parametric Parzen window estimator based on generated samples from models obtained with different training procedures. For reference the measure obtained on the training set is 239.88 ± 2.30 (± indicated standard error). As can be seen the TMCMC trained model largely dominates.

	Sample generation procedure		
Training procedure	TMCMC	Gibbs (random start)	Gibbs (test start)
TMCMC	208.26	**210.72**	209.83
FPCD	180.41	174.87	175.92
PCD	80.06	127.95	139.36
CD	-1978.67	-854.08	37.18

$\hat{p}_{\sigma,\mathscr{D}_s}(\mathbf{x})$. We leave this as future work. Also interesting to note: sampling CD or PCD-trained models with tempered MCMC results in a worse performance than with standard Gibbs sampling. This is definitely more pronounced in the case of CD and highlights the issues raised in section 8.5.1. As for PCD, this again confirms that mixing eventually breaks down during learning, after which negative particles fail to explore the energy landscape properly. Using tempered MCMC during training seems to avoid all these pitfalls.

8.6 Conclusion

We presented a new learning algorithm to train Restricted Boltzmann Machines, relying on the strengths of a more advanced sampling scheme than the ones typically used until now. The tempered MCMC sampling technique allows for a better mixing of the underlying chain used to generate samples from the model. We have shown that this results in better generative models, from qualitative and quantitative observations on real and simulated datasets. We have also shown that the use of tempering affords a higher reliability and increased robustness to learning rates and number of unsupervised training epochs. More experiments are still required however to assess the impact of this method on classification, especially in the context of deep architectures, where one typically stacks trained RBMs before using a global supervised criterion. Amongst other

questions, we would like to determine if and under which conditions a better generative model translates to an increase in classification performance.

CHAPTER 9

CONCLUSION

The work presented herein was motivated by the problem of artificial vision and object recognition which to date, remains largely unsolved. Failure of the "feature engineering" approach to yield robust object detectors seemed to justify the approach used in this thesis, based on the automatic learning of feature hierarchies using both supervised and unsupervised learning. To this end, the first two chapters focused on giving the reader the necessary background in Machine Learning in order to introduce the Artificial Neural Network and the Deep Belief Network which both form the backbone of this thesis. The core of the work relies on three articles which present separate, yet complementary contributions to the field. In this last chapter, we start by giving a brief summary of these contributions in light of recent developments. We then conclude by highlighting interesting areas for future research.

9.1 Article Summaries and Discussion

9.1.1 Empirical Evaluation of Convolutional Architectures for Vision

Motivated by previous work on CNNs [39], we introduced in Chapter 4 the Convolutional RBM, which mimics the architectural properties of hidden layers in CNNs. Hidden units are grouped into feature maps which share the same parametrization and are only connected to visible units within their receptive field. By reducing the number of parameters to learn and enforcing local, shift-equivariant feature detectors, the goal was to show that CRBMs could achieve better generalization when used as the building block of Convolutional DBNs. This work also represented an effort to move deep networks away from toy-problems (i.e small MNIST-sized images) and apply them to real-world images. By measuring the exact log-likelihood on small training images, we showed that CRBMs could learn to model the training distribution more efficiently than fully-connected RBMs.

This work represented a milestone on the road to building large Convolutional DBNs and motivated similar work on other deep architectures, such as the stacked Denoising Auto-Encoder (SdA). While this work remains unpublished to date, the advantage of convolutional architectures to vision is clear. In both cases, they result in better generalization on a wide variety of tasks.

Their true potential (and that of DBNs as a whole) is however best described by the recent work of Lee *et al.* [42]. In it, they show how Convolutional DBNs can be used to efficiently extract a hierarchical representation of data. Once trained on natural images, the first hidden layer learns to detect local edge-like features which are combined at the second layer to form object-part detectors and eventually form features specific to entire objects. The learnt features thus become increasingly correlated with object class as depth is increased. By learning such a hierarchical representation of data in an unsupervised manner, this work captures the essence of Deep Learning [3] and serves as inspiration to further pursue this line of research.

9.1.2 Quadratic Polynomials Learn Better Image Features

In section 1.5, we touched on the theoretical justifications of higher-order units. Inspired by the discovery that simple cells in the visual cortex exhibit similar behaviour [56], we set out in Chapter 6 to study their impact on visual learning tasks in both ANNs and CNNs. Using a low-rank approximation to the quadratic matrix, we reported across all architectures an increase in classification accuracy for several visual recognition tasks. The computational model of [56] also seemed to justify a novel activation function called $softsign$, characterized by a gentler slope than the traditional sigmoidal functions. The best results were obtained with architectures using both quadratic units and the $softsign$ activation function. Further work is still required to fully understand the above results, as the translation invariance hypothesis proved inconclusive.

9.1.3 Tempered Markov Chain Monte Carlo for Training of Restricted Boltzmann Machines

In the last part of this thesis, we turned our attention to the basic training algorithm of RBMs. Starting with the observation that RBMs trained with CD lead to poor generative models, we studied the recently proposed PCD and FPCD algorithms. While both offer notable improvements in performance, the PCD algorithm is still rather brittle. It relies on the energy surface being smooth in order for the negative chain to mix well. Excessive learning rates or training epochs can however lead to a peaked energy landscape which prevents proper mixing and consequently, learning. FPCD encourages mixing by performing small model perturbations. Unfortunately, this has the potential to generate spurious samples, meaning that the parameter updates will not follow the true gradient.

In Chapter 8, we showed that using tempered MCMC in the negative phase of PCD addresses these issues. This was confirmed both by visualization of samples, as well as estimation of the log-likelihood. By using a robust sampling technique, RBMs trained with tempered MCMC are also much less sensitive to the choice of learning rate and the number of unsupervised training epochs.

9.1.4 Discussion

In order to improve performance of artificial vision systems (and in the process gain a better understanding of what is required for learning complex tasks), we have approached the subject from several angles. From a high-level view, Chapter 4 dealt mostly with the issue of tailoring the network architecture to the task at hand, hence with the more general topic of using prior information to facilitate learning. As we have already mentioned, hard-coding prior information can be advantageous as it saves the machine from having to learn this prior. Given computational constraints (e.g. upper-bound on computation time), this may open the door to learning an even richer set of functions. While DBNs already rely on the unsupervised learning phase to learn a good prior of the data, they may still benefit from this strategy.

In section 1.5, we saw that quadratic units have the potential to learn certain invari-

ances, which we tried to exploit in Chapter 6. In the same manner, these invariances could be hard-coded within each unit by imposing certain constraints on the quadratic matrix. This may also help with the issue of computational complexity and make learning of Higher-Order Threshold Logic Units more practical.

Chapters 4 and 6 also show that biology can be a good source of inspiration for getting this prior information and more generally, that ML can benefit from advances in neuroscience. After all, the biological brain is the only known system capable of handling the complex tasks in which we are interested.

Finally, Chapter 8 focused on the unsupervised learning algorithm of RBMs. We showed that despite recent breakthroughs, much work still remains to understand learning in these energy-based models. Of particular importance is the realization that the learning procedure should not be constrained by the quality of the sampling method. This is a simple but powerful result. In addition to creating better generative models, section 8.5.4 shows that this has the potential to make the pre-training phase of DBNs much more robust. The number of unsupervised training epochs is usually fixed or determined by cross-validation. Too large a number would lead to heavily biased models while cross-validation would choose an overly conservative estimate (so as not to break mixing). Some previously published results may therefore be sub-optimal and may need revisiting.

9.2 Future Directions

In terms of future work, a first step will definitely be to replicate the findings of Lee *et al.* [41]. Of particular interest, is the use in [41, 42] of a sparsity criterion in the unsupervised training phase of the RBM. Sparsity ensures that only a subset of hidden units "explain" the visible layer and may be responsible for the quality of the learnt representation in [42]. Without sparsity, Convolutional RBMs have the potential to learn the identity function since the hidden representation is over-complete. From experience, they also seldom learn edge-like features in the first layer. Sparsity may be required for

this[1]. Their probabilistic max-pooling layer is also very attractive compared to the current feed-forward implementation, as it allows its inclusion in full probabilistic models like the Deep Boltzmann Machine [57].

With regards to the work on quadratic units, it would be interesting to integrate these units into deep architectures complete with a pre-training phase. Work has already started in this direction using the stacked Denoising Auto-Encoder. As mentioned previously, we would also like to hard-code prior knowledge of specific invariances (such as translation) into the quadratic term. This may give us the full benefit of higher-order units without the computational burden of having to learn these invariances.

Finally, there are many exciting avenues of research to pursue with regards to the unsupervised training of RBMs using the tempered MCMC approach. The first step will definitely involve learning to model the joint distribution $p_T(x,y)$ directly to measure its impact on classification performance, as in [64, 65]. This would be compared to the typical classification setting of DBNs (which combines pre-training and supervised fine-tuning) with tempered MCMC being used in the pre-training phase. This comparison and the added robustness of the unsupervised training may give new insights into the pre-training strategy of DBNs.

While this issue was not discussed in Chapter 8, we will also be investigating ways in which the number of chains, the maximum temperature and the temperature gaps between chains can be determined automatically. This would not only reduce the number of hyperparameters, but also increase robustness of the sampling method. By selecting these values dynamically, good mixing would be guaranteed regardless of the state of the energy landscape (i.e the state of the learner). It would also reduce computational complexity by using fewer chains early on in training and only adding extra chains when required. Finally, we would also like to apply this novel training algorithm to a wider array of architectures, including the general Boltzmann Machine.

[1] Personal communications with Honglak Lee.

BIBLIOGRAPHY

[1] Adelson, E. H. and Bergen, J. R. (1985). Spatiotemporal energy models for the perception of motion. *Journal of the Optical Society of America*, **2**(2), 284–99.

[2] Baum, E. B. and Haussler, D. (1989). What size net gives valid generalization? *Neural Computation*, **1**, 151–160.

[3] Bengio, Y. (2009). Learning deep architectures for AI. *Foundations and Trends in Machine Learning*, **2**(1). Also published as a book. Now Publishers, 2009.

[4] Bengio, Y. and Delalleau, O. (2009). Justifying and generalizing contrastive divergence. *Neural Computation*, **21**(6), 1601–1621.

[5] Bengio, Y., Lamblin, P., Popovici, D., and Larochelle, H. (2007). Greedy layer-wise training of deep networks. In B. Schölkopf, J. Platt, and T. Hoffman, editors, *Advances in Neural Information Processing Systems 19 (NIPS'06)*, pages 153–160. MIT Press.

[6] Bishop, C. M. (2006). *Pattern Recognition and Machine Learning*. Springer.

[7] Boser, B. E., Guyon, I. M., and Vapnik, V. N. (1992). A training algorithm for optimal margin classifiers. In *Fifth Annual Workshop on Computational Learning Theory*, pages 144–152, Pittsburgh. ACM.

[8] Carreira-Perpiñan, M. A. and Hinton, G. E. (2005). On contrastive divergence learning. In R. G. Cowell and Z. Ghahramani, editors, *Proceedings of the Tenth International Workshop on Artificial Intelligence and Statistics (AISTATS'05)*, pages 33–40. Society for Artificial Intelligence and Statistics.

[9] Chang, C.-C. and Lin, C.-J. (2001). *LIBSVM: a library for support vector machines*. Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

[10] Cortes, C. and Vapnik, V. (1995). Support vector networks. *Machine Learning*, **20**, 273–297.

[11] Elman, J. L. (1993). Learning and development in neural networks: The importance of starting small. *Cognition*, **48**, 781–799.

[12] Erhan, D., Manzagol, P.-A., Bengio, Y., Bengio, S., and Vincent, P. (2009). The difficulty of training deep architectures and the effect of unsupervised pre-training. In *Proceedings of the Twelfth International Conference on Artificial Intelligence and Statistics (AISTATS 2009)*, pages 153–160.

[13] Fei-Fei, L., Fergus, R., and Perona, P. (2004). Learning generative visual models from few training examples: An incremental bayesian approach tested on 101 object categories. page 178.

[14] Fisher, R. A. (1936). The use of multiple measurements in taxonomic problems. *Annals of Eugenics*, **7**, 179–188.

[15] Freund, Y. and Haussler, D. (1994). Unsupervised learning of distributions on binary vectors using two layer networks. Technical Report UCSC-CRL-94-25, University of California, Santa Cruz.

[16] Fukushima, K. (1980). Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, **36**, 193–202.

[17] Giles, C. L. and Maxwell, T. (1987). Learning, invariance, and generalization in high-order neural networks. *Applied Optics*, **26**(23), 4972.

[18] Griffin, G., Holub, A., and Perona, P. (2007). Caltech-256 object category dataset. Technical Report Technical Report 7694, California Institute of Technology.

[19] Håstad, J. (1986). Almost optimal lower bounds for small depth circuits. In *Proceedings of the 18th annual ACM Symposium on Theory of Computing*, pages 6–20, Berkeley, California. ACM Press.

[20] Haykin, S. (1998). *Neural Networks: A Comprehensive Foundation (2nd Edition)*. Prentice Hall, 2 edition.

[21] Heeger, D. J. (1992). Normalization of cell responses in cat striate cortex. *Visual Neuroscience*, **9**(2), 181–198.

[22] Hinton, G. E. (1999). Products of experts. In *Proceedings of the Ninth International Conference on Artificial Neural Networks (ICANN)*, volume 1, pages 1–6, Edinburgh, Scotland. IEE.

[23] Hinton, G. E. (2002). Training products of experts by minimizing contrastive divergence. *Neural Computation*, **14**, 1771–1800.

[24] Hinton, G. E. (2006). To recognize shapes, first learn to generate images. Technical Report UTML TR 2006-003, University of Toronto.

[25] Hinton, G. E., Sejnowski, T. J., and Ackley, D. H. (1984). Boltzmann machines: Constraint satisfaction networks that learn. Technical Report TR-CMU-CS-84-119, Carnegie-Mellon University, Dept. of Computer Science.

[26] Hinton, G. E., Osindero, S., and Teh, Y. (2006). A fast learning algorithm for deep belief nets. *Neural Computation*, **18**, 1527–1554.

[27] Hornik, K., Stinchcombe, M., and White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural Networks*, **2**, 359–366.

[28] Hubel, D. and Wiesel, T. (1968). Receptive fields and functional architecture of monkey striate cortex. *Journal of Physiology (London)*, **195**, 215–243.

[29] Hubel, D. H. and Wiesel, T. N. (1959). Receptive fields of single neurons in the cat's striate cortex. *Journal of Physiology*, **148**, 574–591.

[30] Iba, Y. (2001). Extended ensemble monte carlo. *International Journal of Modern Physics*, **C12**, 623–656.

[31] Kavukcuoglu, K., Ranzato, M., Fergus, R., and LeCun, Y. (2009). Learning invariant features through topographic filter maps. In *Proceedings of the Computer Vision and Pattern Recognition Conference (CVPR'09)*. IEEE.

[32] Kirkpatrick, S., Jr., C. D. G., , and Vecchi, M. P. (1983). Optimization by simulated annealing. *Science*, **220**, 671–680.

[33] Larochelle, H., Erhan, D., Courville, A., Bergstra, J., and Bengio, Y. (2007). An empirical evaluation of deep architectures on problems with many factors of variation. In Z. Ghahramani, editor, *Proceedings of the 24th International Conference on Machine Learning (ICML'07)*, pages 473–480. ACM.

[34] Le Roux, N. and Bengio, Y. (2008). Representational power of restricted Boltzmann machines and deep belief networks. *Neural Computation*, **20**(6), 1631–1649.

[35] Le Roux, N., Bengio, Y., Lamblin, P., Joliveau, M., and Kégl, B. (2008). Learning the 2-d topology of images. In J. Platt, D. Koller, Y. Singer, and S. Roweis, editors, *Advances in Neural Information Processing Systems 20 (NIPS'07)*, pages 841–848, Cambridge, MA. MIT Press.

[36] LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., and Jackel, L. D. (1989). Backpropagation applied to handwritten zip code recognition. *Neural Computation*, **1**(4), 541–551.

[37] LeCun, Y., Bottou, L., Orr, G. B., and Müller, K.-R. (1998a). Efficient backprop. In *Neural Networks, Tricks of the Trade*, Lecture Notes in Computer Science LNCS 1524. Springer Verlag.

[38] LeCun, Y., Bottou, L., Orr, G. B., and Müller, K.-R. (1998b). Efficient BackProp. In G. B. Orr and K.-R. Müller, editors, *Neural Networks: Tricks of the Trade*, pages 9–50. Springer.

[39] LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998c). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, **86**(11), 2278–2324.

[40] LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998d). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, **86**(11), 2278–2324.

[41] Lee, H., Ekanadham, C., and Ng, A. (2008). Sparse deep belief net model for visual area V2. In J. Platt, D. Koller, Y. Singer, and S. Roweis, editors, *Advances in Neural Information Processing Systems 20 (NIPS'07)*, pages 873–880. MIT Press, Cambridge, MA.

[42] Lee, H., Grosse, R., Ranganath, R., and Ng, A. Y. (2009). Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations. In L. Bottou and M. Littman, editors, *Proceedings of the Twenty-sixth International Conference on Machine Learning (ICML'09)*. ACM, Montreal (Qc), Canada.

[43] Lowe, D. G. (2004). Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, **60**(2), 91–110.

[44] Minsky, M. L. and Papert, S. A. (1969). *Perceptrons*. MIT Press, Cambridge.

[45] Morgan, N. and Bourlard, H. (1990). Generalization and parameter estimation in feedforward nets: some experiments. In D. Touretzky, editor, *Advances in Neural Information Processing Systems 2 (NIPS'89)*, pages 413–416, Denver, CO. Morgan Kaufmann.

[46] Pinto, N., Cox, D. D., and DiCarlo, J. J. (2008). Why is real-world visual object recognition hard? *PLoS Comput Biol*, **4**.

[47] Poggio, T. and Girosi, F. (1990). Networks for approximation and learning. *Proceedings of the IEEE*, **78**(9), 1481–1497.

[48] Powell, M. (1987). Radial basis functions for multivariable interpolation: A review.

[49] Raina, R., Madhavan, A., and Ng, A. Y. (2009). Large-scale deep unsupervised learning using graphics processors. In L. Bottou and M. Littman, editors, *Proceedings of the Twenty-sixth International Conference on Machine Learning (ICML'09)*, pages 873–880, New York, NY, USA. ACM.

[50] Ranzato, M., Poultney, C., Chopra, S., and LeCun, Y. (2007a). Efficient learning of sparse representations with an energy-based model. In B. Schölkopf, J. Platt,

and T. Hoffman, editors, *Advances in Neural Information Processing Systems 19 (NIPS'06)*, pages 1137–1144. MIT Press.

[51] Ranzato, M., Huang, F., Boureau, Y., and LeCun, Y. (2007b). Unsupervised learning of invariant feature hierarchies with applications to object recognition. In *Proceedings of the Computer Vision and Pattern Recognition Conference (CVPR'07)*. IEEE Press.

[52] Reid, M. B., Spirkovska, L., and Ochoa, E. (1989). Rapid training of higher-order neural networks for invariant pattern recognition. In *International Joint Conference on Neural Networks (IJCNN)*, Washington, DC, USA.

[53] Robert, C. P. and Casella, G. (1999). *Monte Carlo Statistical Methods*. Springer.

[54] Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986a). Learning representations by back-propagating errors. *Nature*, **323**, 533–536.

[55] Rumelhart, D. E., McClelland, J. L., and the PDP Research Group (1986b). *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, volume 1. MIT Press, Cambridge.

[56] Rust, N., Schwartz, O., Movshon, J. A., and Simoncelli, E. (2005). Spatiotemporal elements of macaque V1 receptive fields. *Neuron*, **46**(6), 945–956.

[57] Salakhutdinov, R. and Hinton, G. E. (2009). Deep Boltzmann machines. In *Proceedings of The Twelfth International Conference on Artificial Intelligence and Statistics (AISTATS'09)*, volume 5, pages 448–455.

[58] Salakhutdinov, R. and Murray, I. (2008). On the quantitative analysis of deep belief networks. In W. W. Cohen, A. McCallum, and S. T. Roweis, editors, *Proceedings of the Twenty-fifth International Conference on Machine Learning (ICML'08)*, volume 25, pages 872–879. ACM.

[59] Serre, T., Wolf, L., Bileschi, S., and Riesenhuber, M. (2007). Robust object recognition with cortex-like mechanisms. *IEEE Trans. Pattern Anal. Mach. Intell.*, **29**(3), 411–426. Member-Poggio, Tomaso.

[60] Shin, Y. and Ghosh, J. (1991). The pi-sigma network: An efficient higher-order neural network for pattern classification and function approximation. In *International Joint Conference on Neural Networks (IJCNN)*, Seattle, Washington, USA.

[61] Siöberg, J. and Ljung, L. (1992). Overtraining, regularization, and searching for minimum in neural networks. Technical report, Linköping University, S-581 83 Linköping, Sweden.

[62] Smolensky, P. (1986). Information processing in dynamical systems: Foundations of harmony theory. In D. E. Rumelhart and J. L. McClelland, editors, *Parallel Distributed Processing*, volume 1, chapter 6, pages 194–281. MIT Press, Cambridge.

[63] Spirkovska, L. and Reid, M. B. (1990). Connectivity strategies for higher-order neural networks applied to pattern recognition. *International Joint Conference on Neural Networks (IJCNN)*, **1**, 21–26.

[64] Tieleman, T. (2008). Training restricted boltzmann machines using approximations to the likelihood gradient. In W. W. Cohen, A. McCallum, and S. T. Roweis, editors, *Proceedings of the Twenty-fifth International Conference on Machine Learning (ICML'08)*, pages 1064–1071. ACM.

[65] Tieleman, T. and Hinton, G. (2009). Using fast weights to improve persistent contrastive divergence. In L. Bottou and M. Littman, editors, *Proceedings of the Twenty-sixth International Conference on Machine Learning (ICML'09)*, pages 1033–1040, New York, NY, USA. ACM.

[66] Vincent, P., Larochelle, H., Bengio, Y., and Manzagol, P.-A. (2008). Extracting and composing robust features with denoising autoencoders. In W. W. Cohen, A. McCallum, and S. T. Roweis, editors, *Proceedings of the Twenty-fifth International Conference on Machine Learning (ICML'08)*, pages 1096–1103. ACM.

[67] Wasserman, L. (2004). *All of Statistics - A Concise Course in Statistical Inference*. Springer.

[68] Welling, M., Rosen-Zvi, M., and Hinton, G. E. (2005). Exponential family harmoniums with an application to information retrieval. In L. Saul, Y. Weiss, and L. Bottou, editors, *Advances in Neural Information Processing Systems 17 (NIPS'04)*, pages 1481–1488, Cambridge, MA. MIT Press.

[69] Younes, L. (1998). On the convergence of markovian stochastic algorithms with rapidly decreasing ergodicity rates. In *Stochastics and Stochastics Models*, pages 177–228.

Appendix I

Algorithms for Learning CRBMs

Algorithm 3 `PropUp`($\mathbf{v}, \mathbf{W}, \mathbf{c}, rfdim$) *This function allows us to infer the state of the hidden layer h, given the state of the visible layer v. For performance reasons, an optional parameter allows us to directly compute the partial gradient update ∂W, calculated at each step of the convolution.*

$\mathbf{v}$ state of units in the visible layer of the CRBM

$\mathbf{W}$ is the CRBM weight matrix, of dimension (number of hidden features, width of receptive field, height of receptive field, number of visible feature)

$\mathbf{c}$ is the CRBM offset vector for the hidden units

$rfdim$ size of the hidden unit's receptive field

- • Initialize tensor ΔW, of same dimensions as W, to 0
- **for all** pixels (m,n) in hidden layer **do**
  - • $\mathscr{R}$
  - • set receptive field $\mathscr{RF} = \{(u,v) : m \leq u < m + rfdim, n \leq v < n + rfdim\}$
  - **for all** features i of pixel (m,n) **do**
    - • compute $\mathbf{h}_{i,mn} = \text{sigmoid}(\mathbf{c}_i + \sum_{j,(u,v) \in RF} \mathbf{W}_{ij,mn,uv} \mathbf{v}_{j,uv})$;
    - **for all** pixels (u,v) in hidden layer **do**
      - **for all** features j of pixel (u,v) **do**
        - • $\Delta W_{ij,mn,uv} = h_{i,mn} v_{j,uv}$
      - **end for**
    - **end for**
  - **end for**
- **end for**
- • return ΔW

Algorithm 4 `PropDown`($\mathbf{h},\mathbf{W},\mathbf{b},rfdim$) *This function allows us to infer the state of the visible layer v, given the state of the hidden layer h.*
h state of units in the hidden layer of the CRBM
W is the CRBM weight matrix, of dimension (number of hidden features, width of receptive field, height of receptive field, number of visible feature)
b is the CRBM offset vector for the visible units
$rfdim$ dimensions of the hidden unit's receptive field

for all pixels (u,v) in visible layer **do**
  • set $\mathbf{v}_{j,uv} = \mathbf{b}_j$
end for
for all pixels (m,n) in hidden layer **do**
  • set receptive field $\mathscr{RF} = \{(u,v) : m \leq u < m + rfdim, n \leq v < n + rfdim\}$
  for all pixels (u,v) of visible layer in $\mathscr{RF}$ **do**
    for all features j of pixel (u,v) **do**
      • compute $\mathbf{v}_{j,uv} = \mathbf{v}_{j,uv} + \mathbf{h}_{i,mn}\mathbf{W}_{ij,mn,uv}$
    end for
  end for
end for
• set $\mathbf{v} = \text{sigmoid}(\mathbf{v})$.

Algorithm 5 `CRBM_CD`($\mathbf{v}, \mathbf{h}, \mathbf{W}, \mathbf{b}, \mathbf{c}, rfdims$) *Description*
v state of units in the visible layer of the CRBM
h state of units in the hidden layer of the CRBM
W is the CRBM weight matrix, of dimension (number of hidden features, width of receptive field, height of receptive field, number of visible feature)
b is the CRBM offset vector for the visible units
c is the CRBM offset vector for the hidden units
$rfdims$ dimensions of the hidden unit's receptive field, of dimension (width of receptive field, height of receptive field)

- $\bullet x^{(0)} = \mathbf{v}$
- $\bullet \Delta W_u$ = `PropUp`($\mathbf{v}, \mathbf{W}, \mathbf{c}, rfdims$)
- **for all** pixels (m,n) in hidden layer **do**
  - **for all** features i of pixel (m,n) **do**
    - • sample $y^{(0)}_{i,mn}$ from $p(\mathbf{h}_{i,mn} = 1|\mathbf{v})$
  - **end for**
- **end for**
- $\bullet \mathbf{h} = y^{(0)}$
- • `PropDown`($\mathbf{h}, \mathbf{W}, \mathbf{b}, rfdims$)
- **for all** pixels (u,v) in visible layer **do**
  - **for all** features j of pixel (u,v) **do**
    - • sample $x^{(1)}_{j,uv}$ from $p(\mathbf{v}_{j,uv} = 1|\mathbf{h})$
  - **end for**
- **end for**
- $\bullet \mathbf{v} = x^{(1)}$
- $\bullet \Delta W_d$ = `PropUp`($\mathbf{v}, \mathbf{W}, \mathbf{c}, rfdims$)
- **for all** hidden features i **do**
  - • compute $\Delta c_i = \sum_{mn} -y^{(0)}_{i,mn} + y^{(1)}_{i,mn}$
- **end for**
- **for all** visible features j **do**
  - • compute $\Delta b_j = \sum_{uv} -x^{(0)}_{j,uv} + x^{(1)}_{j,uv}$
- **end for**
- $\bullet \Delta W = -\Delta W_u + \Delta W_d$
- • Apply gradient estimates Δc, Δb and ΔW to parameters $\mathbf{c}, \mathbf{b}, \mathbf{W}$