

Université de Montréal

Understanding deep architectures and the effect of unsupervised pre-training

par
Dumitru Erhan

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

Thèse présentée à la Faculté des arts et des sciences
en vue de l'obtention du grade de Philosophiæ Doctor (Ph.D.)
en Informatique

Octobre, 2010

© Dumitru Erhan, 2010.

Université de Montréal
Faculté des arts et des sciences

Cette thèse intitulée :

Understanding deep architectures and the effect of unsupervised pre-training

présentée par :
Dumitru Erhan

a été évaluée par un jury constitué des personnes suivantes :

Alain Tapp,	président-rapporteur
Yoshua Bengio,	directeur de recherche
Max Mignotte,	membre du jury
Andrew Y. Ng,	examineur externe
Pierre Duchesne,	représentant du doyen de la F.A.S.

Thèse acceptée le : 23 février 2011

Résumé

CETTE THÈSE porte sur une classe d'algorithmes d'apprentissage appelés *architectures profondes*. Il existe des résultats qui indiquent que les représentations peu profondes et locales ne sont pas suffisantes pour la modélisation des fonctions comportant plusieurs facteurs de variation. Nous sommes particulièrement intéressés par ce genre de données car nous espérons qu'un agent intelligent sera en mesure d'apprendre à les modéliser automatiquement ; l'hypothèse est que les architectures profondes sont mieux adaptées pour les modéliser.

Les travaux de Hinton *et al.* (2006) furent une véritable percée, car l'idée d'utiliser un algorithme d'apprentissage non-supervisé, les *machines de Boltzmann restreintes*, pour l'initialisation des poids d'un réseau de neurones supervisé a été cruciale pour entraîner l'architecture profonde la plus populaire, soit les réseaux de neurones artificiels avec des poids totalement connectés. Cette idée a été reprise et reproduite avec succès dans plusieurs contextes et avec une variété de modèles.

Dans le cadre de cette thèse, nous considérons les architectures profondes comme des *biais inductifs*. Ces biais sont représentés non seulement par les modèles eux-mêmes, mais aussi par les méthodes d'entraînement qui sont souvent utilisés en conjonction avec ceux-ci. Nous désirons définir les raisons pour lesquelles cette classe de fonctions généralise bien, les situations auxquelles ces fonctions pourront être appliquées, ainsi que les descriptions qualitatives de telles fonctions.

L'objectif de cette thèse est d'obtenir une meilleure compréhension du succès des architectures profondes. Dans le premier article, nous testons la concordance entre nos intuitions—que les réseaux profonds sont nécessaires pour mieux apprendre avec des données comportant plusieurs facteurs de variation—et les résultats empiriques. Le second article est une étude approfondie de la question : pourquoi l'apprentissage non-supervisé aide à mieux généraliser dans un réseau profond ? Nous explorons et évaluons plusieurs hypothèses tentant d'élucider le fonctionnement de ces modèles. Finalement, le troisième article cherche à définir de façon qualitative les fonctions modélisées par un réseau profond. Ces visualisations facilitent l'interprétation des représentations et invariances modélisées par une architecture profonde.

Mots-clés : apprentissage automatique, réseaux de neurones artificiels, architectures profondes, apprentissage non-supervisé, visualisation.



Summary

THIS THESIS studies a class of algorithms called *deep architectures*. We argue that models that are based on a shallow composition of local features are not appropriate for the set of real-world functions and datasets that are of interest to us, namely data with many factors of variation. Modelling such functions and datasets is important if we are hoping to create an intelligent agent that can learn from complicated data. Deep architectures are hypothesized to be a step in the right direction, as they are compositions of nonlinearities and can learn compact distributed representations of data with many factors of variation.

Training fully-connected artificial neural networks—the most common form of a deep architecture—was not possible before Hinton *et al.* (2006) showed that one can use stacks of unsupervised Restricted Boltzmann Machines to initialize or pre-train a supervised multi-layer network. This breakthrough has been influential, as the basic idea of using unsupervised learning to improve generalization in deep networks has been reproduced in a multitude of other settings and models.

In this thesis, we cast the deep learning ideas and techniques as defining a special kind of inductive bias. This bias is defined not only by the kind of functions that are eventually represented by such deep models, but also by the learning process that is commonly used for them. This work is a study of the reasons for why this class of functions generalizes well, the situations where they should work well, and the qualitative statements that one could make about such functions.

This thesis is thus an attempt to understand why deep architectures work. In the first of the articles presented we study the question of how well our intuitions about the need for deep models correspond to functions that they can actually model well. In the second article we perform an in-depth study of why unsupervised pre-training helps deep learning and explore a variety of hypotheses that give us an intuition for the dynamics of learning in such architectures. Finally, in the third article, we want to better understand what a deep architecture models, qualitatively speaking. Our visualization approach enables us to understand the representations and invariances modelled and learned by deeper layers.

Keywords: machine learning, artificial neural networks, deep architectures, unsupervised learning, visualization.



Contents

Résumé	v
Summary	vii
Contents	ix
List of Figures	xiii
List of Tables	xv
Glossary	xvii
Acknowledgements	xix
1 Introduction	1
1.1 Artificial Intelligence	3
1.2 Machine Learning	4
1.3 The Generalization Problem	7
1.3.1 Training and Empirical Risk Minimization	8
1.3.2 Validation and Structural Risk Minimization	10
1.3.3 The Bayesian approach	12
1.4 The Inevitability of Inductive Bias in Machine Learning	15
2 Previous work: from perceptrons to deep architectures 19	19
2.1 Perceptrons, neural networks and backpropagation	19
2.2 Kernel machines	24
2.3 Revisiting the generalization problem	28
2.3.1 Local kernel machines	29
2.3.2 Shallow architectures	30
2.3.3 The need for an appropriate inductive bias	32
2.4 Deep Belief Networks	35
2.4.1 Restricted Boltzmann Machines	36
2.4.2 Greedy Layer-wise training of DBNs	38
2.5 Exploring the greedy unsupervised principle	40
2.6 Recent developments in deep architectures	42

2.6.1	Deep Belief Networks	42
2.6.2	Denoising Auto-encoders	48
2.6.3	Energy-based frameworks	49
2.6.4	Semi-supervised embedding for Natural Language Processing and kernel-based approaches	51
2.7	Understanding the bias induced by unsupervised pre- training and deep architectures	52
3	Presentation of the first article	55
3.1	Article details	55
3.2	Context	55
3.3	Contributions	56
3.4	Comments	57
4	An empirical evaluation of deep architectures on prob- lems with many factors of variation	59
4.1	Introduction	59
4.1.1	Shallow and Deep Architectures	60
4.1.2	Scaling to Harder Learning Problems	61
4.2	Learning Algorithms with Deep Architectures	62
4.2.1	Deep Belief Networks and Restricted Boltzmann Machines	62
4.2.2	Stacked Autoassociators	64
4.3	Benchmark Tasks	65
4.3.1	Variations on Digit Recognition	65
4.3.2	Discrimination between Tall and Wide Rectangles	67
4.3.3	Recognition of Convex Sets	67
4.4	Experiments	68
4.4.1	Benchmark Results	70
4.4.2	Impact of Background Pixel Correlation	71
4.5	Conclusion and Future Work	73
5	Presentation of the second article	75
5.1	Article details	75
5.2	Context	75
5.3	Contributions	76
5.4	Comments	78
6	Why Does Unsupervised Pre-training Help Deep Learn- ing?	81
6.1	Introduction	81
6.2	The Challenges of Deep Learning	84

6.3	Unsupervised Pre-training Acts as a Regularizer	86
6.4	Previous Relevant Work	88
6.4.1	Related Semi-Supervised Methods	88
6.4.2	Early Stopping as a Form of Regularization	89
6.5	Experimental Setup and Methodology	90
6.5.1	Models	90
6.5.2	Deep Belief Networks	91
6.5.3	Stacked Denoising Auto-Encoders	92
6.5.4	Data Sets	94
6.5.5	Setup	94
6.6	The Effect of Unsupervised Pre-training	96
6.6.1	Better Generalization	96
6.6.2	Visualization of Features	98
6.6.3	Visualization of Model Trajectories During Learning	99
6.6.4	Implications	103
6.7	The Role of Unsupervised Pre-training	103
6.7.1	Experiment 1: Does Pre-training Provide a Better Conditioning Process for Supervised Learning?	104
6.7.2	Experiment 2: The Effect of Pre-training on Training Error	106
6.7.3	Experiment 3: The Influence of the Layer Size	107
6.7.4	Experiment 4: Challenging the Optimization Hypothesis	108
6.7.5	Experiment 5: Comparing pre-training to L_1 and L_2 regularization	110
6.7.6	Summary of Findings: Experiments 1-5	110
6.8	The Online Learning Setting	110
6.8.1	Experiment 6: Effect of Pre-training with Very Large Data Sets	111
6.8.2	Experiment 7: The Effect of Example Ordering	113
6.8.3	Experiment 8: Pre-training only k layers	115
6.9	Discussion and Conclusions	116
7	Presentation of the third article	123
7.1	Article details	123
7.2	Context	123
7.3	Contributions	124
7.4	Comments	125
8	Understanding Representations in Deep Architectures	127
8.1	Introduction	127

8.2	Previous work	129
8.2.1	Linear combination of previous units	129
8.2.2	Output unit sampling	130
8.2.3	Optimal stimulus analysis for quadratic forms	130
8.3	The models	131
8.4	How to obtain filter-like representations for deep units	133
8.4.1	Sampling from a unit of a Deep Belief Network	133
8.4.2	Maximizing the activation	133
8.4.3	Connections between methods	135
8.4.4	First investigations into visualizing upper layer units	136
8.5	Uncovering Invariance Manifolds	142
8.5.1	Invariance Manifolds	144
8.5.2	Results	146
8.5.3	Measuring invariance	148
8.6	Conclusions and Future Work	149
9	Conclusion	153
9.1	Main findings	154
9.2	Speculative remarks	155
	References	157

List of Figures

1.1	Illustration of 4 different learning problems.	6
1.2	Illustration of 32-example dataset and a polynomial fit.	7
1.3	Dependence of polynomial fit on the hyper-parametres	11
2.1	The perceptron model.	20
2.2	Two learning problems, one solvable by the Perceptron and one that cannot be solved using a linear model. . .	20
2.3	Examples of activation functions	21
2.4	A single-hidden-layer neural network	22
2.5	A 2D binary classification problem	25
2.6	Examples of shallow architectures	31
2.7	A Restricted Boltzmann Machine	36
2.8	Greedy layer-wise training of a Deep Belief Network . .	39
2.9	Greedy layer-wise training of a Stacked Auto-Associator	41
4.6	Samples from <i>convex</i>	68
4.7	Samples with progressively less pixel correlation in the background.	72
4.8	Classification error on MNIST examples with progressively less pixel correlation in the background.	72
6.1	Effect of depth on performance for a model trained without unsupervised pre-training and with unsupervised pre-training	97
6.2	Histograms of test errors obtained on MNIST with or without pre-training	97
6.3	Visualization of filters learned by a DBN trained on InfiniteMNIST	98
6.4	Visualization of filters learned by a network without pre-training, trained on InfiniteMNIST	99
6.5	2D visualizations with <i>t</i> SNE of the functions represented by 50 networks with and 50 networks without pre-training	101
6.6	2D visualization with ISOMAP of the functions represented by 50 networks with and 50 networks without pre-training.	102

6.7	Training errors obtained on Shapese t when stepping in parameter space around a converged model in 7 random gradient directions.	103
6.8	Evolution without pre-training and with pre-training on MNIST of the log of the test NLL plotted against the log of the train NLL as training proceeds	106
6.9	Effect of layer size on the changes brought by unsupervised pre-training	108
6.10	For MNIST , a plot of the log(train NLL) vs. log(test NLL) at each epoch of training. The top layer is constrained to 20 units.	109
6.11	Comparison between 1 and 3-layer networks trained on InfiniteMNIST . Online classification error, computed as an average over a block of last 100,000 errors.	112
6.12	Error of 1-layer network with RBM pre-training and without, on the 10 million examples used for training it	113
6.13	Variance of the output of a trained network with 1 layer. The variance is computed as a function of the point at which we vary the training samples.	114
6.14	Testing the influence of selectively pre-training certain layers on the train and test errors	116
8.1	Activation maximization applied on MNIST	138
8.2	Activation Maximization (AM) applied on Natural Image Patches and Caltech Silhouettes	141
8.3	Visualization of 6 units from the second hidden layer of a DBN trained on MNIST and natural image patches.	142
8.4	Visualization of 36 units from the second hidden layer of a DBN trained on MNIST and 144 units from the second hidden layer of a DBN trained on natural image patches	143
8.5	Illustration of the invariance manifold tracing technique in 3D.	145
8.6	Output filter minima for the output units corresponding to digits 4 and 5, as well as a set of invariance manifolds for each	147
8.7	Measuring the invariance of the units from different layers.	148



List of Tables

4.1	Results on the benchmark for problems with factors of variation	71
6.1	Effect of various initialization strategies on 1 and 2-layer architectures.	105



Glossary

AA	Auto-Associator
AM	Activation Maximization
ANN	Artificial Neural Network
CD	Contrastive Divergence
CE	Cross-Entropy
CRBM	Conditional Restricted Boltzmann Machine
DBM	Deep Belief Network
DBM	Deep Boltzmann Machine
EBM	Energy-Based Model
ERM	Empirical Risk Minimization
FPCD	Fast Persistent Contrastive Divergence
GP	Gaussian Processes
HMM	Hidden Markov Models
ICA	Independent Components Analysis
KL	Kullback-Leibler (divergence)
LLE	Locally-Linear Embedding
MAP	Maximum a-posteriori
MCMC	Markov Chain Monte Carlo
MDL	Minimum Description Length
ML	Maximum Likelihood
MLP	Multi-Layer Perceptron
MRF	Markov Random Field
MSE	Mean Squared Error
NCA	Neighbourhood Components Analysis

NLL	Negative Log-Likelihood
NLP	Natural Language Processing
NN	Neural Network
PCA	Principal Components Analysis
PCD	Persistent Contrastive Divergence
PDF	Probability Distribution Function
PL	Pseudo-Likelihood
PSD	Predictive Sparse Decomposition
RBF	Radial Basis Function
RBM	Restricted Boltzmann Machine
SAA	Stacked Auto-Associator
SAE	Stacked Auto-Encoder
SDAE	Stacked Denoising Auto-Encoder
SESM	Symmetric Encoding Sparse Machine
SFA	Slow Feature Analysis
SNE	Stochastic Neighbourhood Embedding
SRM	Structural Risk Minimization
SSL	Semi-Supervised Learning
SV	Support Vector
SVM	Support Vector Machine
VC	Vapnik-Chervonenkis
ZCA	Zero-phase Components Analysis



Acknowledgements

First and foremost, I extend my gratitude to Yoshua Bengio, the fearless leader of the LISA lab and my advisor who, at the onset of my PhD in 2006 warned me that he was embarking the whole lab on a long-term project—the study of deep architectures—which may fail (it did not). I thank him for his frankness, for his tireless and passionate attitude towards research, for the great advice, for making it possible to participate in this long-term project, and for being an inspiration throughout the MSc and PhD years at Université de Montréal.

Aaron Courville deserves special recognition for being an amazing person to work with. His pursuit of perfection—of results, of our writing, of our arguments—has considerably improved this thesis and the work contained within. I wish to thank the LISA lab members for being such a great gang, for the many great discussions we have had together, and for being great co-authors. Hugo Larochelle, James Bergstra, Nicolas Le Roux, Pierre-Antoine Manzagol, Douglas Eck, Guillaume Desjardins, Olivier Delalleau, Joseph Turian, Pascal Lamblin, Pascal Vincent—it has been a pleasure working and sharing the same lab with you.

I also wish to acknowledge Samy Bengio and the people at Google Research with whom I collaborated during the past two years for providing me with a great opportunity to test some of my ideas in a large-scale industrial setting.

Nicolas Chapados is responsible for the \LaTeX template that makes this document so much more readable. Pierre-Antoine Manzagol and James Bergstra have provided me with useful feedback and suffered through reading the drafts of this document. The committee members have made this thesis better in a variety of ways, many thanks to them as well.

Stella, thank you for having been the perfect “partner in crime” during this whole PhD experience and for having stuck with me, against the odds. Finally, this thesis would not exist without the indefatigable support of my parents, Nadejda and Ion, whose optimism has never faded and who always believed in me.

To my parents

1

Introduction

THE FIELD OF Artificial Intelligence (AI) is at the intersection of a broad spectrum of disciplines, ranging from theoretical computer science, to statistics, many branches of mathematics, computational neuroscience, robotics, cognitive psychology, and linguistics, among others. Its aims are to better understand intelligence, especially human, and to eventually create such intelligence. The reasons for wanting to create it are varied, but the hope is that through such creation one can better understand the world around us, make scientific progress and introduce ways to benefit society at large.

Since research in AI is so broad in its potential scope, there is a great variety of questions that researchers in this field can tackle: whether they relate to parallels between human (or animal intelligence) and the kind of intelligence that one can create with a computer program, or to the theoretical properties and limitations of algorithms that are commonly used in the literature.

Machine Learning (ML) is a subfield of AI, whose main subject is the study of learning from data. Just as AI is at the intersection of several disciplines, so is Machine Learning: it is admittedly difficult to draw a sharp line that differentiates the two, however research in ML tends to concentrate on the more computational aspects of artificial intelligence, especially as they relate to learning from data. Whereas, for instance, a good portion of AI research is devoted to studying agents situated in a world, classical ML is typically concentrated on the *modelling* aspects of the same problem.

Both AI and ML research is heavily influenced by our perception and understanding of human and animal intelligence. Early advances in neuroscience, especially the computational aspects of it, have shaped the kind of algorithms and models that constituted the beginnings of ML research. The reason for that is simple: brains are the what makes human intelligent and they provide an example that we can take inspiration from. By investigating them, we can not only advance the state of our knowledge regarding human intelligence, but we can hope to create new, intelligent models of the world.

In this thesis, we explore a class of machine learning algorithms that are inspired by our understanding of the neural connectivity and

organization of the brain, called *deep architectures**. The main goal of this thesis is to advance the understanding of this class of learning algorithms. Sometimes our work took inspiration from our understanding of the human brain to better understand these algorithms, and other times it made it possible to frame the effects of these algorithms in terms of standard machine learning concepts. Ultimately, the contribution of this thesis is to ask and provide answers to questions that further our understanding of a large class of state-of-the-art models in Machine Learning and to make sure that these answers enable researchers in the field to improve on this state-of-the-art.

This thesis is structured as follows:

Chapter 1: defining intelligence, artificial intelligence, machine learning, and the fundamental problem of machine learning: the problem of generalization.

Chapter 2 : exploring attempts at to solving this generalization problem, all the while paying attention to advances in our understanding of the short-comings of such solutions. In the same chapter, we describe *deep architectures*, the new class of algorithms that appear to possess qualities that make it possible to overcome said short-comings.

Chapters 3 and 4 : inquiry into these deep architectures, their properties and their limitations. We begin by presenting our work on investigating how susceptible deep architectures are to certain variations of interesting learning problems, fundamentally and empirically.

Chapters 5 and 6 : we pin down certain ingredients from deep architectures into well-known Machine Learning concepts, and formulate a coherent hypothesis that gives us an idea as to why deep architectures can better solve the generalization problem.

Chapters 7 and 8 : we undertake the mission of building (mathematical) tools for qualitatively understanding deep architecture models.

Chapter 9 : provides a synthesis and discussion of our hypotheses, results, and tools.

*. Throughout this thesis, the definition of deep architectures includes the structure of the model as well as the algorithms used to learn in them.

1.1 Artificial Intelligence

There are various types and definitions of intelligence and intelligent behaviour. The type of intelligence that we will be referring to in this thesis is the kind found in humans and most animals. It is the kind of intelligence that allows us to infer meaningful abstractions when faced with an environment, be it through the transformation of the activations of photo-receptive cells of the retina into mental images or through the transformation of air vibrations into useful mental representations such as words. This is definitely not the only manifestation of intelligence, as, in this discussion, we do not cover issues such as action, survival or reproduction, but limit ourselves to the passive forms of intelligence.

AI in general, and Machine Learning specifically, is concerned, among other things, with emulating intelligence. It is not necessary to copy the exact biological or physical mechanisms of humans in order to recreate their capabilities; indeed, one can treat humans as black boxes and instead build machines that behave *like* humans (in the *input-output* sense), but that do not necessarily *function* like humans. As long as the machine can produce the same kind of intelligence that we observe in humans (or even better one), we should be reasonably satisfied*.

However, it would be wasteful to disregard the knowledge that we can obtain by investigating the mechanisms occurring in the human brain. This is because not only are humans in the unique position of pondering over their own intelligence, they are also the owners of brains that have evolved to intelligence unmatched by other animals on Earth. Human brains are thus successful examples of self-introspective intelligence, and therefore analyzing the principles behind the human brain (its organization, connectivity, evolution throughout lifetime and so forth) could help us greatly in understanding the nature of intelligence.

It is very likely that there exists a number of ways of creating a system or agent that is intelligent—yet the mere existence of the brain suggests a prior over these possibilities, each of them being, in a vague sense, a computational view or model of how to learn and manipulate representations of the world. This means that we are better off at

*. A famous argument against this idea is the *Chinese Room Experiment*. Searle (1999) argues that if he could convincingly simulate the behaviour of a native Chinese speaker, by following input/output instructions, he could conceivably pass the Turing test without actually understanding Chinese. In the context of the discussion on creating intelligence, this thesis will take a computational view on the theory of mind, arguing that we can view the brain as a mechanism for creating and manipulating representations. In this respect, the quest to emulate these mechanisms is not as short-sighted as it might seem.

trying to search for a good model of the world in the vicinity of the model specified by the brain, than by starting in a completely random place.

It is probably this kind of thinking that led the first researchers of AI to the creation of the Perceptron (Rosenblatt, 1958), one of the first mathematical idealizations of the biological neuron (an “artificial neuron”). This was possible because of the advances in neuroscience, which analyzed the behaviour of biological neurons in the presence of an action potential. Nonlinear differentiable activation functions and an efficient way of training networks of one or more layers of such artificial neurons gave rise to a whole subfield of AI/ML—Artificial Neural Networks (ANN).

ANNs are neither truthful representations of the biological and chemical processes that happen in the brain, nor are they replacements for the brain (yet). They do provide however an example of how researchers can get *inspiration* from the brain and advance the state of the art in AI. As described in Chapter 2, ANN research is experiencing a revival, thanks to the discovery of useful and efficient ways of training many layers of neurons. The techniques for training many layers of neurons, as well as the underlying models, are what we shall usually refer to as *deep architectures* throughout this thesis.

This work expands on this research by investigating the reasons why such techniques work. We are motivated by the fact that despite the recent advances in multilayer ANNs and despite the impressive experimental results, we lack a thorough understanding of what makes them work. Thus, we are looking for *insights* into these models. It is worth noting that we are not aiming for a better understanding of the brain as such; though we do not exclude the possibility of finding interesting parallels with results obtained by cognitive psychologists or computational neuroscientists, the research that is described in this thesis aims rather at a better understanding of the algorithms and models that we currently use.

1.2 Machine Learning

It is hard to imagine an intelligent system or agent that is not capable of learning from its own experience. Therefore, we do not simply want a system that is able to come up with meaningful abstractions and representations of the world, given an environment. We want a system that is able to *learn* to come up with such abstractions and manipulate them, given a series of environments and time.

We formalize certain basic notions of Machine Learning in the following, so as to make the further discussions more precise. As mentioned in the above, Machine Learning posits as a problem the creation of agents, systems, or algorithms that learn, either from data or from their environment. *Data* can in principle be anything that we can quantify in some way, but we shall usually deal with a set of vectors \mathbf{x} called \mathcal{X} , coming from an \mathbb{R}^D space. A *model* of the data is a function f , parametrized by a set of parameters θ , which maps elements of \mathcal{X} to some space \mathbb{R}^T . We shall denote the output of f on \mathcal{X} as the set $\mathcal{O} = f(\mathcal{X})$, with $\mathbf{o} \in \mathcal{O}$, while \mathcal{Y} denotes the set associated with \mathcal{X} , corresponding to the “true” (or optimal) model of the data. The task is then to find a procedure that would modify θ in such a way that $\mathbf{o} = f(\mathbf{x})$ would be as similar as possible to \mathbf{y} , given \mathbf{x} , with an emphasis on *generalizing* to *unseen* vectors \mathbf{x} in \mathbb{R}^D that are likely to come from the same distribution as the elements of \mathcal{X} . In other words, we are interested in finding that “true” model of the data, though this is an oversimplification (for presentation purposes), as in reality we operate on *probability distributions* and there does not necessarily exist a one-to-one mapping from \mathcal{X} to \mathcal{Y} ; rather, we are interested in finding or approximating the joint distribution $P_{\mathbf{x} \in \mathcal{X}, \mathbf{y} \in \mathcal{Y}}(\mathbf{x}, \mathbf{y})$, with an emphasis on obtaining the conditional $P(\mathbf{y}|\mathbf{x})$.

A *learning algorithm* specifies the sequence of steps that modifies the parameters of f in order to attain some objective \mathcal{L} . Usually, \mathcal{L} is some cost that penalizes the dissimilarity between \mathbf{o} and \mathbf{y} for some \mathbf{x} . The learning algorithm is used to solve a specific instance of a *learning problem*. Depending on the type of \mathbf{y} , we shall generally distinguish between the following types of problems:

classification : \mathcal{Y} has finite cardinality. If the cardinality of $\mathcal{Y} = 2$, we call this a binary classification problem. If the cardinality of $\mathcal{Y} > 2$, we call it a multiclass classification problem. The unique members of \mathcal{Y} are called *labels* and are given to the algorithm prior to learning.

regression : \mathcal{Y} is an infinite and possibly unbounded subset of \mathbb{R}^T . The values of \mathcal{Y} are called *targets* and are given to the algorithm prior to learning.

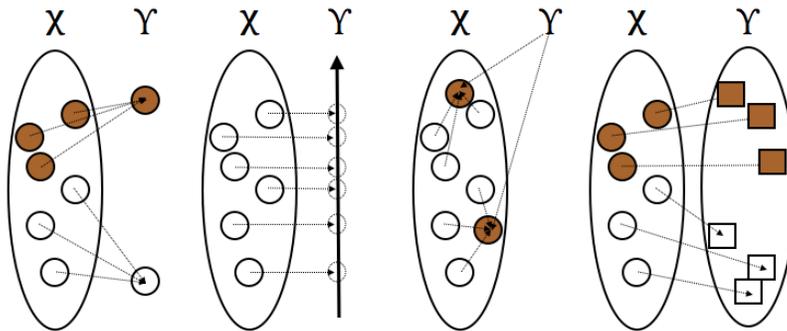
clustering : \mathcal{Y} is either a set of vectors from \mathbb{R}^D (which can be a subset of \mathcal{X}) or a set of indices, neither given to the algorithm before or during training. The cardinality of \mathcal{Y} is smaller than the cardinality of \mathcal{X} .

embedding / projection : \mathcal{Y} is a set of vectors in \mathbb{R}^T (where T can be larger *or* smaller than D) called *embeddings* that are *not* given

to the algorithm prior to or during learning. The cardinality of \mathcal{Y} is equal to the cardinality of \mathcal{X} .

These four problems are illustrated in Figure 1.1. Note that we gloss over many variations in between these problems (and wholly different problems, such as *reinforcement learning*), since the examples shown in Figure 1.1 are the ones of particular interest for this thesis. Also note that classification and regression are examples of *supervised learning problems*, while clustering and embedding / projection are examples of *unsupervised learning problems*, the difference being the presence or the absence of \mathcal{Y} during learning. Finally, this thesis will find special interest in problems that are at the intersection of supervised and unsupervised learning: these are called *semi-supervised learning problems*.

► **Figure 1.1.** Illustration of 4 different learning problems. From left to right: a binary classification problem, a regression problem, clustering with cluster centers being part of the domain of \mathcal{X} and an embedding / projection problem



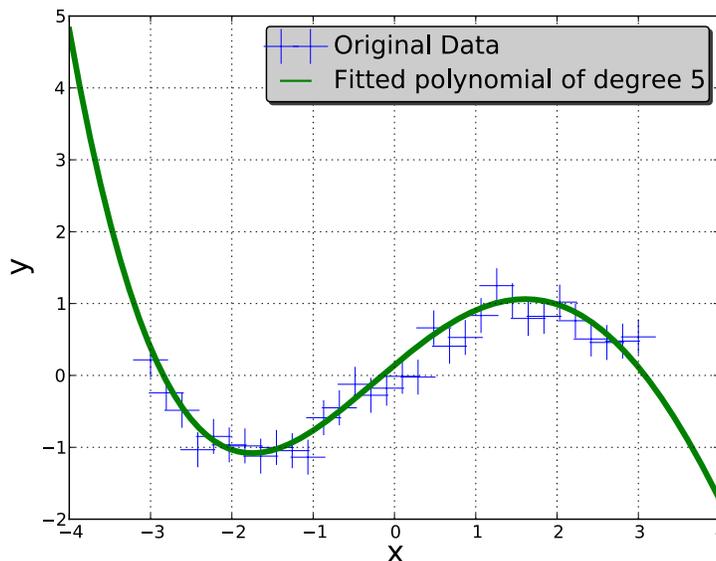
As described in the above, a learning algorithm is a *functional* that takes as inputs $\mathcal{X}, \mathcal{Y}, \mathcal{L}$ and a set of parameters θ and returns a modified set of these parameters θ^* , or, equivalently, a function f_{θ^*} which can then be applied to new input instances. This process is called *training*. Note that in case of clustering and embedding / projection, \mathcal{Y} are not actually given to the learning algorithm. In those cases, the learning problem is to find a good way of *representing* \mathcal{X} , rather than a way of approximating the relationship between \mathcal{X} and \mathcal{Y} . There is a variety of reasons for wanting a better representation, but the simplest one is that a good representation could, in principle, reveal interesting structure in \mathcal{X} (as shown in Figure 1.1, where the projections of \mathcal{X} separate better the examples from the two classes). Another reason for wanting a good representation is by viewing the problem of modelling \mathcal{X} as that of *density estimation*, i.e., constructing a probability distribution function from the input instances given to us via \mathcal{X} .

Finally, some notational aspects: the column vector corresponding to the D -dimensional input with index i in the dataset is \mathbf{x}^i . Its j th component: x_j^i . The corresponding target output is y^i , while the output

predicted by some model is \hat{y}^i . A column weight vector is θ and its j th component is w_j . A weight matrix is W and the element on the i th row and j th column is W_{ij} .

1.3 The Generalization Problem

Consider a simple example (in Figure 1.2), which is a 1-dimensional 32-example dataset (blue crosses), generated by $y^i = \sin(x^i) + N(0, 0.25)$ where $N(0, 0.25)$ is a univariate Normal noise with mean zero and standard deviation of 0.25. The learning algorithm does not see the generating function, but just the (x^i, y^i) pairs. This is a supervised learning problem, where the algorithm has to perform regression. The subset of the input set $\mathcal{X} \times \mathcal{Y}$ that we shall use for getting θ^* is called a *training set*, denoted by D_{train} . In our 32-example case, this is represented by half of the examples (randomly chosen).



▲ **Figure 1.2.** Illustration of a 1-dimensional 32-example dataset (blue crosses), generated by $y^i = \sin(x^i) + N(0, 0.25)$ where $N(0, 0.25)$ is a univariate Normal with mean zero and standard deviation of 0.25, $x \in [-3; 3]$, and $i = 1, \dots, 32$. The green curve corresponds to a polynomial of degree 5, found by minimizing the Mean Squared Error (MSE).

This serves as the prototypical scenario for the discussion on the fact that solving the learning algorithm—finding a function or its parameters that minimizes \mathcal{L} given the training data—is not an easy endeavour. First, there is an infinite number of functions that can provide

an error-free mapping from the given inputs to the given outputs*. We are thus presented with a few important questions: how can one even enumerate or search through these functions? Does it matter which function we ultimately choose? And if yes, is there a criterion that we can use to choose from these functions?

1.3.1 Training and Empirical Risk Minimization

In the most general case possible, without making any further assumptions, we cannot search through all the possible function mappings in finite time. So we need to restrict ourselves to a class of functions which we can either enumerate or search through easily. In this particular case, we could restrict ourselves to the model class of polynomials, meaning that we will seek functions of the type

$$f_k(x, \theta) = \theta_0 + \sum_{j=1}^k \theta_j x^j$$

where x^j is x to the j th power and where we will denote $\theta = \theta_0, \dots, \theta_k$.

In general, models of the data can either be *parametric* or *non-parametric*. A *parametric* model class has a *fixed* capacity for any size of the training set, whereas a *nonparametric* model's capacity generally *grows* with the size of the training set. The capacity of a model class is roughly the cardinality of the set of functions represented by it. This is just one of the few definitions for this concept; intuitively, it states that if a model class has greater capacity it can model more (and more complex) functions, since there are, in principle, more functions to choose or search from. The polynomial model that we investigate corresponds to a parametric model of the data. This is because, in our example, a polynomial of degree k is a class of functions, as it represents all functions which have the form of a polynomial of degree k , meaning the weight vectors of all polynomials of degree k ; the number of such functions[†] is fixed during training and does not grow with the size of the data.

Assuming that we can efficiently find the polynomial of degree k that “fits” the data best, then the problem should be easily solved. Unfortunately, there are certain steps in this procedure which need to

*. Moreover, it is not at all clear that we want an error-free mapping, since we, as designers of the learning problem generated the data with noise. However, the learning algorithm does not have, generally speaking, access to such information

†. The number of such functions is infinite, but we can make statements that polynomials of degree k are a subset of polynomials of degree $k + 1$ and hence the former class of functions has fewer functions.

be further refined and which play a major role in the kind of model we will get at the end. First, we need to define what we mean by “fit”: how do we measure whether a certain function fits the data? We can take a Mean-Squared Error (MSE) approach and define the degree of fit between the predictions $\hat{y}^i = f_k(x^i, \theta)$ of this function and the outputs as

$$E(k, \theta, D_{train}) = \sum_{i=1}^{N_{train}} (\hat{y}^i - y^i)^2$$

where x^i is the i th 1-dimensional input example, y^i is the corresponding output, k is the degree of the polynomial, and θ the parameter vector corresponding to the specific polynomial that we are analyzing. $E(k, \theta, D_{train})$ is what we called a *loss functional* in Section 1.2.

The principle of defining a function class and choosing a function from this set, which minimizes a given training error is called *Empirical Risk Minimization* (ERM) (Vapnik and Chervonenkis, 1971). What one typically wants to minimize is the *true risk*, meaning the loss functional evaluated over the entire distribution $p(\mathbf{x}, y)$, but generally we cannot do that and have to make do with the finite training set that is at our disposal.

The choice of Mean-Squared Error might seem arbitrary and this is true in the absence of other information. It turns out that it is a natural option when solving a regression problem. Assume we have a loss $L(y, f(\mathbf{x}))$ and a distribution $p(\mathbf{x}, y)$ over which we want to compute this loss. The expected loss is then

$$E_{\mathbf{x}, y}[L] = \int \int L(y, f(\mathbf{x})) p(\mathbf{x}, y) d\mathbf{x} dy$$

If the chosen loss is $(y - f(\mathbf{x}))^2$, then we can differentiate $E_{\mathbf{x}, y}[L]$ with respect to $f(\mathbf{x})$. When setting to zero, the function $f(\mathbf{x})$ that minimizes this expected loss becomes $E_y[y|\mathbf{x}]$, which is the conditional expectation of y given \mathbf{x} , also called the *regression function*.

In our particular case, for a given k , the fact that $f_k(x, \theta)$ is a linear function in θ and that MSE is a quadratic function of θ implies that we can efficiently find the θ that minimizes $E(k, \theta, D_{train})$ (this is equivalent to performing least-squares regression with a simple linear model). The solution to this optimization problem is unique, given k and D_{train} ; let us denote it by θ^* . Figure 1.4(a) shows $E(k, \theta^*, D_{train})$, the training error, for each k . As one would expect, the larger k is, the better the fit and when $k \geq N_{train} - 1$, the MSE will necessarily be zero. Naturally, the question is then: how to choose k ?

1.3.2 Validation and Structural Risk Minimization

One of the standard procedures for verifying the quality of a model *validation*: it is the computation of a loss functional \mathcal{L}_{valid} (which could be different from \mathcal{L}) on some (statistically independent from D_{train}) subset of $\mathcal{X} \times \mathcal{Y}$ called the *validation set* D_{valid} . Validation is needed for choosing between the *hyperparameters* of a learning algorithm—those parameters which are not directly optimized during training, but which still influence the quality of the parameters θ^* that are found by applying the Empirical Risk Minimization principle. A model class indexed by some hyperparameters λ is a subset of functions from \mathcal{X} to \mathcal{Y} whose parameters are θ_λ . The tuple $(\lambda, \theta_\lambda^*)$ that minimizes \mathcal{L}_{valid} is the one we should choose as our model.

In our example, the only hyper-parameter is k . cannot guide ourselves by $E(k, \theta^*, D_{train})$ for choosing an optimal k . The validation principle requires us to split the data available into a training set, which we did by defining D_{train} as a random half of the data, and a validation set D_{valid} , the other random half in our example. Then, one uses the validation loss functional on D_{valid} to make decisions about hyper-parameters. In our example, the validation loss functional that we will define is going to be MSE again.

In Figure 1.4(a) we see a curve that is common in many machine learning experiments: there seems to be a k^* (at around $k = 5$) for which the error on the validation set (the *validation error*) is lowest. Any model with $k < k^*$ seems to *underfit* the data, meaning that it does not have enough capacity (large enough number of functions from which to choose), whereas any model with $k > k^*$ seems to *overfit* the data, meaning it likely has too much capacity.

To get an unbiased estimate of the performance of the model represented by $k = k^*$, we would need to test the best model on a *test set* D_{test} (independent of D_{train} and D_{valid}) using some loss functional, which is normally L_{valid} . This unbiased estimate of the performance of the model is also called the *generalization error* and choosing a model class which allows to us efficiently obtain the lowest generalization error for a given learning problem is the central problem in Machine Learning.

While the method by which we chose k —the one that minimizes the validation error—is popular and intuitively sound, it is certainly not the only way in which we could proceed. For example, the principle of “Occam’s Razor”, which says that one should choose the *simplest* model that explains the data Blumer *et al.* (1987), has also been investigated. More generally, the procedure for giving preference to certain configurations of the parameters at the expense of others is called *reg-*

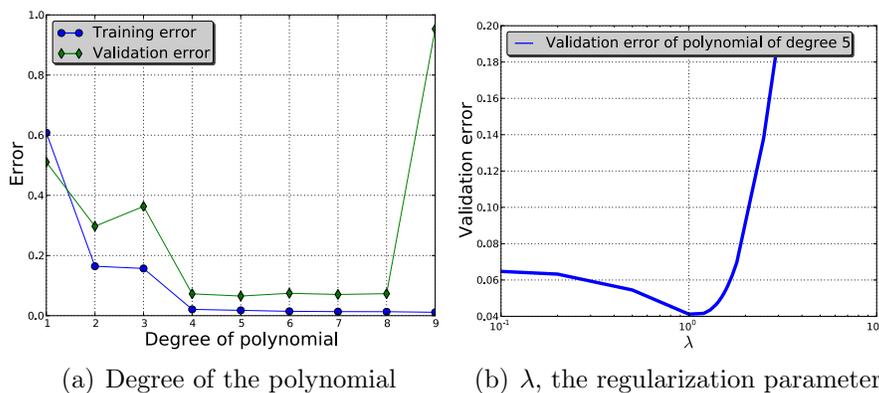
ularization. Both parametric and nonparametric model classes can be *regularized*: this means that one can impose constraints on the type and richness of functions that can be used or learned, i.e. we can constrain the capacity of the model class. Regularization typically acts as a trade-off mechanism between fitting the training data and having a good model for previously unseen examples.

Getting back to our example, let us take the class of functions that is represented by the polynomials of degree k^* . A specific instance from this class is in Figure 1.2. One way of constraining the parameters of this function class is to minimize it by modifying the loss function such that

$$E(k, \lambda, \theta, D_{train}) = \sum_{i=1}^{N_{train}} (f_k(x^i, \theta) - y^i)^2 + \lambda \|\theta\|^2$$

Minimizing this loss function is otherwise known as ridge regression (Höerl and Kennard, 1970).

The principle of adding a constraint to the process of Empirical Risk Minimization so as to favour certain functions in a given class is called *Structural Risk Minimization* (SRM) (Vapnik, 1982). Assuming one can measure the complexity of a given model or model class (norm of parameters, degree of polynomial, etc.), the SRM principle is a general way of choosing among models, by simply selecting the one whose sum of the empirical risk and complexity level is minimal. In this sense, SRM is a trade-off between the data fit and the complexity of a given model and is a principled way of performing regularization.



◀ **Figure 1.3.** MSE as a function of the degree of the polynomial and of λ , the regularization parameter (for $k = 5$, the degree that minimizes the error on the validation set)

In our case, the $\lambda \|\theta\|^2$ regularizer introduces a penalty proportional to the square of the magnitude of each weight, which in the particular case of polynomials, will encourage functions with small parameter values, i.e., polynomials with small coefficients. Figure 1.4(b) shows the

value of the training and validation errors as a function of λ , the regularization value. Note that this is not, in the strictest sense, the application of the Structural Risk Minimization principle. SRM would make us choose λ to minimize $E(k, \lambda, \theta, D_{train}) + C(k, \lambda, \theta)$, where $C(k, \lambda, \theta)$ is a measure of complexity of the model specified by k, θ and λ (and not the *value* of k, θ and λ). It is unfortunately not obvious to define such a complexity measure; fortunately, simply evaluating $E(k, \lambda, \theta, D_{valid})$ is a reasonable and very popular alternative. As with k, λ trades off *simplicity* with *data fit* and, as before, there seems to exist a value of λ that minimizes such a trade-off; like before, we can interpret any model with λ smaller or larger than this optimal value as underfitting or overfitting, respectively.

Even after performing regularized polynomial fitting on our data, the result in Figure 1.2 should tell us that our job of finding a class of functions that generalizes well is far from finished. While the polynomial of degree $k^* = 5$ found with MSE is able to model the underlying function well within the data bounds, it is clearly mismatched with the target function, as it does not extrapolate well. This mismatch is a consequence of our choice to restrict ourselves to polynomials in our search: no polynomial will be really able to model a sinusoid, and we are bound to have numerical and stability issues if we have to resort to fitting polynomials of very high degree. This is an important lesson as it shows that even when a model is theoretically guaranteed to be able to represent any function*, it might not be at all efficient and one might run into the trouble of the target function simply being mismatched with the class of functions considered during learning.

1.3.3 The Bayesian approach

So far we have taken a *function approximation* approach to modelling: we have defined the result of learning a *single* θ^* , given to us by a combination of a learning objective functional and a model selection scheme. This θ^* represents the set of parameters or function that we believe, based on measures such as the lowest validation error, would best generalize to unseen data.

The basic idea of the Bayesian approach is that within a given class of functions (specified, say, by a combination of hyper-parameters), several θ could seem as appropriate choices: when predicting the output for a previously unseen \mathbf{x}_{new} , one has to somehow integrate or average over all of these choices, weighing them in a principled man-

*. It is easy to convince oneself that a polynomial can approximate any reasonably smooth function of the input, given sufficient data and a large enough degree.

ner. Formally, if $f(\mathbf{x}, \theta)$ is how we parametrize our function class (e.g., polynomials of a certain degree), then we should be making decisions by

$$E_{\theta}(y|\mathbf{x}_{new}, D_{train}) = \int f(\mathbf{x}_{new}, \theta)P(\theta|D_{train})d\theta$$

where we weigh each prediction by $P(\theta|D_{train})$, the *posterior* distribution of the parameters given the training data. Unfortunately, this integral is intractable for a lot of functions that are of interest to us and most often one has to resort to approximations, such as in the procedure for computing, approximating or sampling from $P(\theta|D_{train})$.

To obtain $P(\theta|D_{train})$ one uses the Bayes theorem:

$$P(\theta|D_{train}) = P(D_{train}|\theta)P(\theta)/P(D_{train})$$

Central to the Bayesian approach is the notion of a *prior* over the parameters: $P(\theta)$ encodes our prior belief in the distribution of θ without having seen any training data. $P(D_{train}|\theta)$ is the *likelihood* of the data given a setting of the parameters: we shall see in the following that the analysis of this distribution is of interest to us.

One of the more useful approximations that one could make during the process of applying the principles of Bayesian inference is the MAP (Maximum A-Posteriori) approach, which is the principle of choosing the θ that maximizes the posterior

$$\theta^{MAP} = \arg \max_{\theta} P(\theta|D_{train}) = \arg \max_{\theta} P(D_{train}|\theta)P(\theta)$$

The intuition is that the mode of the posterior distribution is a good point estimate of the entire distribution. For certain simple distributions (such as the multivariate Gaussian) the mode is indeed a representative point estimate*.

Depending on the shape of the prior or the likelihood, computing θ^{MAP} can be difficult or intractable, because the posterior distribution could be multi-modal. Often the solution is to use the principle of Maximum Likelihood, which states that one should choose θ to maximize

$$\theta^{ML} = \arg \max_{\theta} P(D_{train}|\theta)$$

which is the likelihood of the data given the parameters. There are connections between the ERM/SRM and ML/MAP principles, respectively. To reveal these connections within our polynomial modelling framework, we have to cast the problem in a probabilistic setting, since

*. Naturally, the MAP estimate may be a poor approximation for a multi-modal posterior distribution.

no mathematical object that we looked at so far in our discussion on polynomial fitting defines a probability of any sorts. If we take our example of fitting a polynomial to the data, we can define a distribution that represents the uncertainty that we have in our prediction for a given $f_k(x, \theta)$:

$$p(y|x, \theta) = N(y|f_k(x, \theta), \sigma^2)$$

where $f_k(x, \theta)$ is the polynomial as defined before and $N(y|f_k(x, \theta), \sigma^2)$ is a Gaussian distribution centered at $f_k(x, \theta)$ with a standard deviation of σ .

Assuming that our data is i.i.d. and drawn from this distribution, for a given training set (x^i, y^i) , where $i = 1, \dots, N_{train}$, the conditional likelihood of the data given the parameters is

$$p(D_{train}|\theta) = \prod_{i=1}^{N_{train}} N(y^i|f_k(x^i, \theta), \sigma^2)$$

Maximizing $p(D_{train}|\theta)$ implies taking the log, differentiating wrt. to θ and setting to zero: one can then see that maximizing the likelihood is equivalent to minimizing the Mean Squared Error on the training set. Thus, when assuming an Gaussian model for our probabilistic predictions, the ML solution—call it θ^{ML} —is equivalent to the MSE method we used when applying the ERM principle before.

Likewise, if we incorporate a prior $P(\theta) = N(\mathbf{0}, \lambda\mathbf{I})$ into our predictions, then maximizing the posterior $P(\theta|D_{train}) \propto P(D_{train}|\theta)P(\theta)$ with respect to θ means finding the θ that minimizes

$$\sum_{i=1}^{N_{train}} (f_k(x^i, \theta) - y^i)^2 + \lambda\|\theta\|^2$$

Therefore, we see that the squared penalty we used to regularize when applying the SRM principle is equivalent to introducing an isotropic Gaussian prior on the weights (with a standard deviation that is equivalent to λ in our model) and finding the MAP solution— θ^{MAP} —to the problem.

With either θ^{ML} or θ^{MAP} we can make predictions for previously unseen inputs by simply evaluating $p(y|x, \theta^{ML})$ or $p(y|x, \theta^{MAP})$ for a given x . A full Bayesian treatment, as mentioned in the above, requires us to not simply come up with a point estimate of θ , but marginalize over the entire posterior distribution when $P(\theta|D_{train})$ when making decisions. Thus, we would like to be able to calculate distributions of the form

$$p(y|x, D_{train}) = \int p(y|x, \theta)p(\theta|D_{train})d\theta$$

For the distributions that we considered so far (uni and multi-variate Gaussians) this can be done analytically, since computing the integrals is a matter of convolving two Gaussians. Thus, $p(y|x, D_{train})$ is a Gaussian itself, whose mean and standard deviation are functions of x , the unseen input.

Often, one can use complicated multi-modal prior distributions if one makes approximations, either in the inference or in the learning process, or both. However, taking the Bayesian approach to modelling and inferring point estimates or posterior parameter distributions is almost always a trade-off between the mathematical convenience of having tractable distributions (i.e., over which one can tractably evaluate integrals or sums) and having powerful models of the data.

1.4 The Inevitability of Inductive Bias in Machine Learning

What transpires from the discussion so far is that there is a multitude of choices that one makes before arriving at a model that is satisfactory:

- The class of functions to consider for modelling: polynomial, perceptrons, artificial neural networks, etc.
- The general principle for optimization or inference in this class of functions: maximum likelihood or ERM, MAP or SRM, fully Bayesian, etc.
- Model choice or selection procedure: the choice of the loss functional, including how to split the available data into training, validation and test sets.
- Model constraints: how to *regularize* the model such that certain parameter configurations are more preferred than others or, equivalently, which prior distribution to use in a Bayesian framework.

One might question whether automatic machine learning is possible without us making so many choices. It is instructive to think of the functions that we are searching through as *hypotheses* that explain our data. The choices that we make at each stage during learning either exclude certain hypotheses from our search altogether* or change the likelihood with which our search procedure will consider them†.

*. For instance, the choice of considering only polynomials of order k excludes all the other hypotheses

†. The penalty on large parameter values will make them more unlikely.

Every time we make a choice of this type, we are *biasing* the optimization or search through the hypothesis space. A fundamental result in Machine Learning is that bias-free learning is not possible (Mitchell, 1990), for there is always generally an infinite number of hypotheses that explain the data equally well. Equivalently, it is not possible to *generalize* without having an *inductive bias*, which allows us to define a preference for certain hypotheses vs. others; without an inductive bias, we will not be able to have a meaningful way of learning from data.

Another fundamental result in optimization and machine learning, the No-Free Lunch theorem (Wolpert, 1996), states that for every learning algorithm (that generates hypotheses from data) there always exists a distribution of examples on which this model will do badly. In other words, no completely general-purpose learning algorithm can exist, therefore every learning algorithm must contain restrictions—implicit or explicit biases—on the class of functions that it can learn or represent.

Biases can come in different flavours, but generally we can distinguish between *representational* and *procedural* biases (Gordon and Desjardins, 1995). A representational inductive bias is one that makes the hypothesis space incapable of modelling all possible hypotheses that are compatible with the training data, because of the limitations on the kind of hypotheses that can be constructed. A representational bias can be *strong* if the hypothesis space that corresponds to it is small; conversely, the bias is weak if the hypothesis space is large (Utgoff, 1986). Analytically, one can measure the strength of a bias via the Vapnik-Chervonenkis (VC) dimension (Vapnik and Chervonenkis, 1971), which is an indirect measure of the capacity of the model*. Finally, one can also measure the *correctness* (Utgoff, 1986) of a bias: namely, whether the hypothesis space defined by it contains the “correct” hypothesis or not.

A procedural bias makes the search or inference procedure itself biased towards certain hypotheses. Maximum likelihood, Occam’s razor, MAP are all example of such procedural biases. Another popular way of encoding a preference for a given hypothesis is the principle of Minimum Description Length (MDL) (Solomonoff, 1964; Rissanen, 1983), which specifies that preference should be given to hypotheses that allows for the best (or shortest) encoding of the data†. In one way or another, inductive biases are present in all machine learning algorithms for otherwise, as Mitchell (1990) put it “*an unbiased learning system’s*

*. Defined by the cardinality of the set of hypotheses that this model can represent or, more precisely, by the largest set of points that the model can shatter.

†. The MDL principle can be viewed as a variant of Occam’s razor.

ability to classify new instances is no better than if it simply stored all the training instances and performed a lookup when asked to classify a subsequent instance”.

Since inductive bias is *always* present, we have a certain freedom to choose when designing the process of learning. Frequently, the choice is made towards a weak representational bias, therefore making sure that the hypothesis space considered is the most general possible class of functions over which we can define a preference; this is to ensure that our representational bias is *correct*, i.e., does not exclude the target hypothesis. But we should always be aware that our choice must be made such that *interesting* and complicated functions or hypotheses are actually learnable (in finite time), using standard algorithms applied on functionals that take this model class as inputs. This is true even in the context of models that are in principle able to model arbitrary input-output mappings*: inductive bias is still an important tool, since not all these models actually allow for interesting and complicated functions to be learned with finite amounts of data, or time, or simply a number of examples which is not exponential in the number of intrinsic variations in the data.

Modern Machine Learning problems have a few properties which make the process of training, or approximating the relationship between \mathcal{X} and \mathcal{Y} and coming up with hypotheses that generalize well, quite difficult. Any technique for solving the generalization problem must scale well in the number of dimensions of the inputs \mathbf{x} *and* the number of examples considered. Typical ML problems have complex input variations and noise, either in the labelling process or in the data itself, and simple approaches that rely on enumerating the data variations could be extremely inefficient. The generalization problem is even more acute in such a setting, because, as we shall see in Chapter 2, some of the typical representational and procedural inductive biases and assumptions—related to the classes of functions, or the regularity in the input data—simply do not hold for realistic machine learning datasets and problems.

In this thesis, we study the deep architectures. Any instance of these models (and associated learning procedures) is a parametric model of the data, but we are free to choose the capacity of the model based on validation data; this makes the model family non-parametric as well. The combination of these properties is desirable, for it allows us to tune the size of the hypothesis space in a principled manner. One is also able to leverage unsupervised data during the training process, a step which

*. These mappings typically have reasonable properties, such as functions with compact support and having certain continuity properties.

we show crucial in Chapter 6. Their hierarchical structure makes the models compact and highly nonlinear representations of the data and its variations and, empirically, they show great promise in a variety of application domains. Thus, deep architectures provide an interesting inductive bias and this thesis is an exploration of the consequences that this inductive bias has on the hypotheses that we can learn from data.

2

Previous work: from perceptrons to deep architectures

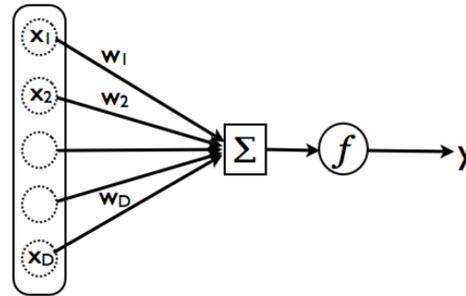
IN THIS CHAPTER we review some important developments in Machine Learning, with an emphasis on Artificial Neural Networks. By any measure, it is not meant to be a comprehensive overview of the history of the field, but rather a background on the work that shaped the ideas presented in the later parts; there is also more emphasis on the state of the art in the field.

2.1 Perceptrons, neural networks and backpropagation

As previously mentioned, one of the first brain-inspired AI models was the Perceptron by Rosenblatt (1958), presented in Figure 2.1. Sidestepping the complications that arise with modelling real, biological neurons, where differential equations govern their behavior, perceptrons are *artificial neurons* whose behavior is a highly simplified version of the real neurons. The basic idea is that a neuron receives inputs from other neurons (the connections between them corresponding to dendrites in a biological neuron), sums these inputs and passes the sum through an *activation function*. The output of the activation function is then passed to the next neuron (corresponds to an axon). As shown in Figure 2.2 (left), the activation function of a perceptron is similar to the one used in biological neurons—a step function (also called the Heaviside function, see Figure 2.3). It corresponds to the neuron “firing” (or being “on”) when the weighted input sums to a value above a threshold.

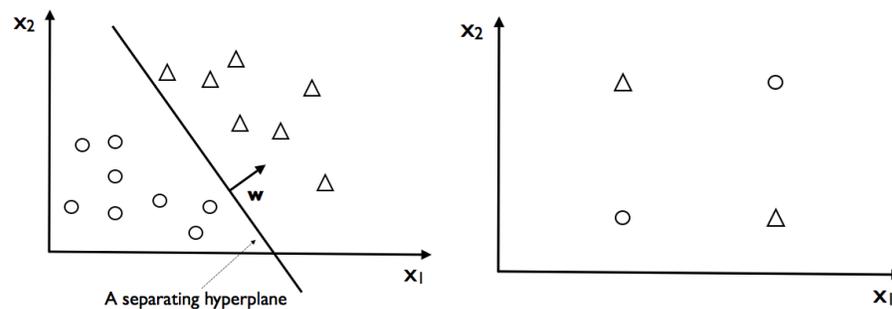
The perceptron algorithm came with a learning rule for modifying its weights to attain a classification objective. Importantly, under certain conditions, this rule will converge to a set of weights that gives zero errors. One such condition is the linear separability of the input examples, as illustrated in Figure 2.2 (left): linear separability means that there exists a hyperplane in the D -dimensional space of \mathbf{x} that can separate (error-free) the input examples. If such a hyperplane exists,

► **Figure 2.1.** The perceptron: a mathematical idealization of the biological neuron.



then the perceptron rule will converge to it; otherwise, there is no guarantee of that happening. A finding with significant impact at that time was that for a variety of problems, linear separability does not apply. Minsky and Papert (1969) have demonstrated that perceptrons are not able to solve seemingly simple problems such as the XOR-problem*, which is not linearly separable (see Figure 2.2).

► **Figure 2.2.** Left: a binary classification problem where there exists a separating hyperplane. Right: the XOR problem, where no separating hyperplane exists and where the perceptron algorithm will not converge



This work was influential enough to slow down research into ANNs for a period of 15 years. Refinements and improvements to the basic linear threshold model were certainly being done, and Widrow and Lehr (1990) present a comprehensive examination of the history of single-layer networks in that period of time; however, a breakthrough was needed in order to overcome the issue of linear separability.

For this breakthrough to occur, two important things had to happen:

- The realization that *differentiable nonlinear* activations allowed much easier numerical optimization.
- An efficient way of training of networks containing such elements.

*. The inputs are all the possible combinations for two bits and the outputs are the respective application of the XOR function on these inputs

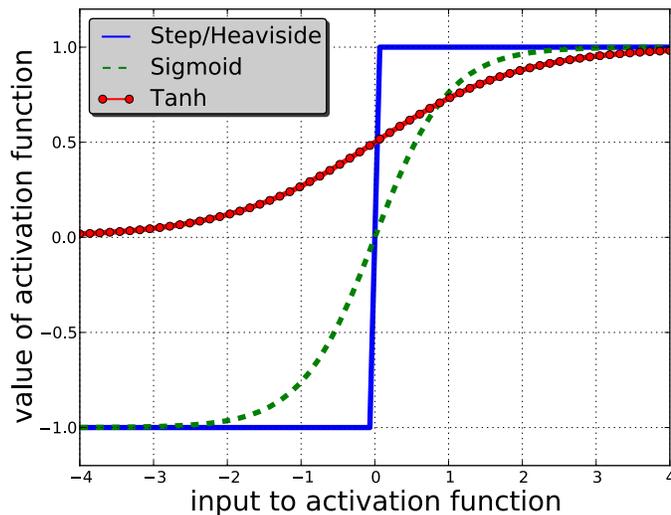
The *backpropagation* algorithm, discovered independently by several researchers in a span of 12 years (Werbos, 1974; Parker, 1985; LeCun, 1986; Rumelhart *et al.*, 1986) provided just that. First, the Heaviside step function was replaced by a *sigmoid* as the activation function. The latter is a family of functions, having roughly an “S”-shape, closely related to the damped nonlinear response that a biological neuron has*. Typical examples from this family are the logistic sigmoid

$$\text{sigm}(\mathbf{x}) = \frac{1}{1 + \exp(-\mathbf{x})}$$

and the hyperbolic tangent

$$\text{tanh}(\mathbf{x}) = \frac{\exp(\mathbf{x}) - \exp(-\mathbf{x})}{\exp(\mathbf{x}) + \exp(-\mathbf{x})}.$$

Both are illustrated in Figure 2.3.



◀ **Figure 2.3.** Illustration of activation functions: the step function, the sigmoid, and the hyperbolic tangent

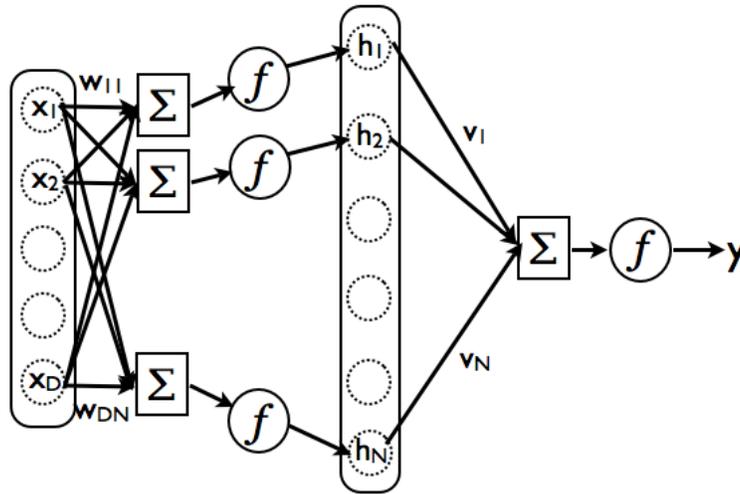
Replacing either of these formulas as the transfer function of a perceptron does not make a significant difference, since the resulting model is a logistic regression model, which suffers from the same issue of linear separability. The key insight is shown in Figure 2.4: one can now build *layers* of such units[†] and the more sigmoidal units we add to a layer,

*. It can also be viewed it as a “soft” step function

†. One could have built multiple layers of perceptrons, too, in a similar fashion. However, the discontinuous and non-differentiable activation function did not lend itself easily to a learning algorithm such as backpropagation, presented below.

the more nonlinear the relationship between the input and the output becomes. This is because the output is now computed as a weighted sum of outputs of these units. Sigmoidal units are then called “hidden units” and the layers containing them are “hidden layers”.

► **Figure 2.4.** A single-hidden-layer neural network that takes a D dimensional vector as input, nonlinearly projects it to a N -dimensional hidden unit space and retrieves an output by a weighted combination of the hidden units followed by the application of an output transfer function



Formally, a single-output, single-hidden-layer neural network, as represented in Figure 2.4 computes the following mathematical function:

$$\hat{y} = f(\mathbf{v}' \text{sigm}(W\mathbf{x})) \quad (2.1)$$

where W is the input-to-hidden weight matrix, \mathbf{v} are the hidden-to-output weights*, \mathbf{x} is the input and $f()$ is the *output transfer function*: it transfers the output of the network into the domain that is problem-specific (the domain of y , the label/target corresponding to its estimate computed by the network, \hat{y}). This formalization can be easily extended to an arbitrary number of hidden units, hidden layers and outputs.

Why is a differentiable activation function important? The above-mentioned researchers had another key insight: the output of equation 2.1 is differentiable with respect to all of its parameters. Given a differentiable *objective function*, $E(\hat{y}, y, x)$ that can compute some measure of error between the prediction of the network \hat{y} and the true y corresponding to some \mathbf{x} , we can now compute the *gradient* of the error with respect to the parameters of the network.

For each of the problems described in Section 1.2 one can find suitable differentiable objective functions, as there is a natural pairing be-

*. Typically, a neural network implementation contains a bias vector/term as well, which can be seen as a weight that does not depend on the input

tween the output transfer function, the error used and the problem to be solved:

- In the case of *regression*, the *squared error* is the most popular choice: $E(\hat{y}, y, x) = (\hat{y} - y)^2$ and the output transfer function is simply the identity.
- With *binary classification*, the output transfer function is usually a logistic sigmoid and \hat{y} can be interpreted as the *probability* $p(y = 1|\mathbf{x})$; the objective function is then the *cross-entropy* $E(\hat{y}, y, x) = -y \log \hat{y} - (1 - y) \log(1 - \hat{y})$, assuming $y \in \{0, 1\}$, which is simply $-\log p(y|\mathbf{x})$, i.e. the negative log-likelihood.
- For *multiclass classification problems*, the so-called *softmax* is the natural choice for the output transfer function as it projects the output of the network onto a set of probabilities that sum to one, corresponding to $p(y_k|\mathbf{x})$, where k is an index into the classes:

$$p(y_k|\mathbf{x}) = \text{softmax}(t_k) = \frac{\exp(t_k)}{\sum_j \exp(t_j)}$$

where \mathbf{t} is a vector of outputs of the network. By transforming y into a *one-hot vector** we can use the negative likelihood $-\log p(y|x)$, too.

- For finding an *embedding* of the input \mathbf{x} one view the network as *encoding* the input into the output domain. The process of *decoding* will transform the output of the network into the same domain as \mathbf{x} and use either the squared error or the cross-entropy to minimize the disparity between the original input and the decoded one. The activations of the hidden layer could then act as the embedding of the data, meaning that they are a new (and hopefully better) representation for the input.

The gradient gives the direction of the *steepest ascent* in the error space. Therefore, by moving in the direction opposite of it, we can perform gradient descent and modify the parameters of the network so as to reach a (local) minimum in the error space. Computing the gradient efficiently was the other ingredient to the success of neural networks after Rumelhart *et al.* (1986)'s publication. The main idea is that that we can recursively compute the gradient with respect to the outputs of each unit, once the gradients for the layer above are known; hence the name *backpropagation*.

This recursive structure gives rise to an efficient algorithm for computing the required gradients since a complete backpropagation pass takes only $O(k)$ (with k being the number of weights). This was a

*. Corresponding to a vector that is all zero except at position k where k is the class number

significant gain in performance, compared by Bishop (1995) to the invention of the Fast Fourier Transform algorithm. The fact that we are able to compute the gradient efficiently does not imply a good method of using it to optimize the weights of a neural network. LeCun *et al.* (1998) provide some discussion on the relative merits of each method, when applied to neural networks, the conclusion being that stochastic gradient updates in small minibatches and a mildly adaptive learning rate schedule works well in practice.

An interesting finding is that neural networks with one hidden layer can, in principle, approximate any continuous and bounded function (Hornik *et al.*, 1989). However, using backpropagation and gradient descent does not guarantee the convergence to the optimal solution, since the error space is highly non-convex and contains a large number of local minima. Also, Hornik *et al.* (1989)'s result gives no guarantee that a one-hidden layer network is a statistically efficient representation for the learning problem at hand (representing the data may require an exponential number of units, parameters and/or examples to capture the target functions). In spite of these limitations, for a while, neural networks with hidden units were the state of the art in many areas and therefore had become quickly very popular in applications areas of Machine Learning.

Experimental evidence by Bengio *et al.* (2007) suggests that training multiple layers with backpropagation gets stuck in poor local minima or plateaus, and results obtained with 3 or more layers were worse than those obtained with networks that have one or two layers. Coupled with Hornik *et al.* (1989)'s finding, *single-layer* networks were a much more popular choice. Therefore, while multiple layers certainly added modeling capacity to the network, they were never widely used. We have thus come to one of the questions that has recently drawn the attention of many researchers in the field: *What is needed to efficiently train a neural network with many layers?* And why would we want that? Simply because a multi-layer network could be more efficient at compactly representing the target function that we want to learn. In the following, to explore several answers to these questions, ranging from application specific modifications to more general principles that seem to be applicable in a variety of settings.

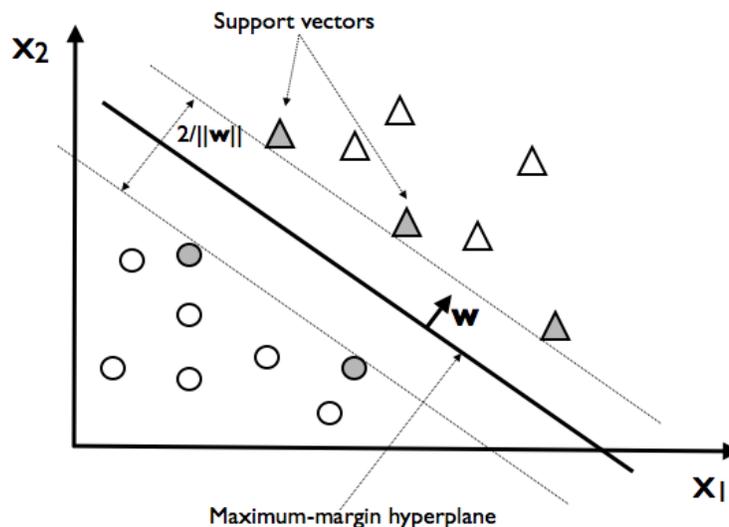
2.2 Kernel machines

While backpropagation has certainly popularized neural networks to the point of near ubiquity in Machine Learning, there are certain is-

sues that one deals with when using ANNs. First, for the vast majority of architectures and objective functions, the cost optimized by gradient descent is not convex in the parameters of the network. Therefore it is sensitive to local minima or initialization of the weights. Secondly, while there are entire papers dedicated to “tips and tricks” for training neural networks, such as by LeCun *et al.* (1998), it is never quite possible to explore in a satisfactory manner the set of hyperparameters that a neural network has; the mere existence of lengthy papers on such “tips and tricks” is a testament to the fact that obtaining good performance with neural networks is not an easy endeavour.

Such issues have fuelled research into alternative ways of modelling data. Over the last 10-15 years, the most popular alternative was the usage of *large-margin* techniques, especially Support Vector Machines with nonlinear kernels. Nonlinear kernels and margin-based approaches are two distinct ideas that are only tied to each other insofar they produce interesting models when used together. We describe them each in turn.

Going back to the basic Perceptron, consider the problem shown in Figure 2.5. In this setting, we ask the algorithm to learn a linear decision boundary that would separate 2D examples from each class. Note that the classes *are* linearly separable and that there is an infinite number of hyperplanes separating them. The perceptron learning rule for modifying the weights \mathbf{w} will choose one of these hyperplanes.



◀ **Figure 2.5.** A binary classification problem (circles vs. triangles) in 2D. The classes are linearly separable, but there is a unique maximum-margin separating hyperplane. This hyperplane is specified by the support vectors of the data or by its corresponding weight vector \mathbf{w}

Without making any further assumptions, the hyperplane chosen by the perceptron learning rule is as good as any other, as long as it

separates the two classes perfectly. The main hypothesis behind large-margin classifiers is that the hyperplane that *maximizes the margin* between itself and the closest examples from each class is to be preferred. Intuitively this makes sense: by placing the decision boundary as far as possible from examples of each class (but still “in between”, obviously) we are hoping to minimize the probability of making a mistake at test time. Note also that the solution to the problem, i.e. the hyperplane that separates the classes with the largest margin, is fully specified by the examples at the margin. We shall see shortly that this is a crucial ingredient for algorithms such as Support Vector Machines.

This is the intuition. Formally speaking, in the case of binary classification, the problem is as follows: given D_{train} , containing (\mathbf{x}^i, y^i) pairs, with $\mathbf{x}^i \in \mathbb{R}^D$ and $y^i = \pm 1$, we want to find a \mathbf{w} that makes the data linearly separable and whose normal—the separating hyperplane—has the largest margin. The optimization problem is then

$$\text{minimize } \frac{1}{2} \|\mathbf{w}\|^2 \quad (2.2)$$

$$\text{s.t. } y^i(\mathbf{w}'\mathbf{x}^i - b) \geq 1, \forall i \quad (2.3)$$

where we used the fact that the distance between the two hyperplanes that separate that data (with no other examples in between) is $\frac{2}{\|\mathbf{w}\|}$ (see Figure 2.5). As shown by Vapnik (1998), this optimization problem has a dual formulation:

$$\text{minimize } \sum_i \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y^i y^j \mathbf{x}^i \mathbf{x}^j \quad (2.4)$$

$$\text{s.t. } \sum_i \alpha_i y^i = 0 \text{ and } \alpha_i \geq 0, \forall i \quad (2.5)$$

with $\mathbf{w} = \sum_i \alpha_i y^i \mathbf{x}^i$. In the solution that is found by solving the above optimization problem, points corresponding to $\alpha_i > 0$ are called “support vectors” and they lie on the hyperplanes on the margin (they are shaded in Figure 2.5). Points with $\alpha_i = 0$ will not be part of the solution and they can in principle be discarded after finding the optimal α . This is the basic linear Support Vector Machine (SVM) algorithm.

As presented in the above, the problem of finding the maximum-margin separating hyperplane is a convex optimization problem. This means that no matter what the initial estimate of \mathbf{w} is, the algorithm will converge to a unique solution in finite time. If the two classes are indeed linearly separable then the algorithm is hyperparameter-free*.

*. Cortes and Vapnik (1995) modify the basic algorithm by introducing *slack variables* to allow for non-separable cases.

Linear SVMs, as described up to now, are not necessarily very useful, even given the large-margin criterion, if the underlying problem is not linearly separable or very noisy. An extension of the linear model, which had by far the most impact in this field was the so-called “kernel trick”. Proposed by Aizerman *et al.* (1964) and popularized by Boser *et al.* (1992), this trick relied on the observation that the solution of the optimization problem in equation 2.4 relies solely on dot products between the training examples. If we projected the original data into some high-dimensional (or even infinite-dimensional) Euclidean \mathcal{H} space using a mapping

$$\Phi : \mathbb{R}^D \rightarrow \mathcal{H}$$

and used those projections as training examples, then the solution would only depend on dot-products of the form $\Phi(\mathbf{x}^i) \cdot \Phi(\mathbf{x}^j)$. If there exists a function, called a *kernel*, that allows us to compute $K(\mathbf{x}^i, \mathbf{x}^j) = \Phi(\mathbf{x}^i) \cdot \Phi(\mathbf{x}^j)$ without explicitly requiring $\Phi(\mathbf{x}^i)$ and $\Phi(\mathbf{x}^j)$ then we can simply replace $\mathbf{x}^i \cdot \mathbf{x}^j$ with $K(\mathbf{x}^i, \mathbf{x}^j)$ in our optimization problem and be done.

Of course, for this to work, there are some restrictions of the kinds of kernels that we can use. Vapnik (1998) provides the details of these so-called *Mercer kernels*. It suffices to say that such kernels allow the optimization problem to remain convex by guaranteeing a positive semi-definite Gram matrix* for any training set. The most popular choices for kernels are extremely simple:

RBF kernel : $K(\mathbf{x}^i, \mathbf{x}^j) = \exp\left(-\frac{\|\mathbf{x}^i - \mathbf{x}^j\|^2}{\sigma^2}\right)$, where σ is the *width* of the kernel, a hyperparameter.

Polynomial kernel : $K(\mathbf{x}^i, \mathbf{x}^j) = (1 + \mathbf{x}^i \cdot \mathbf{x}^j)^d$, where d is the degree of the polynomial, a hyperparameter as well.

Note that in the case of the RBF kernel the underlying projection Φ maps its input onto an infinite-dimensional Hilbert space. It is in this space that we will find the separating hyperplane; however, we will not be able to get a closed form for it. The solution, like in the linear SVM case, will be an expansion over the Support Vectors:

$$f(\mathbf{x}^{test}) = \sum_{i=1}^{N_{SV}} \alpha_i y^i K(\mathbf{s}^i, \mathbf{x}^{test}) + b \quad (2.6)$$

where \mathbf{s}_i are the Support Vectors, i.e. those training points for which the learned α_i is non-zero. This time around, we need to iterate through the set of SVs *every time* we need to classify a new example. What is

*. The matrix of kernel evaluations for all data points in the training set

worse is that Steinwart (2003) has shown that the *expected value* of N_{SV} scales with the size of the training set, unless the training error is zero*. The computational burden of using SVMs does not stop here, however: even with very clever techniques, the time complexity of *training* a Support Vector Machine is at least $O(N_{SV} \cdot N)$ (where N is the number of examples in the training set), which a lot of the times means training time that is quadratic in the number of examples; in practice, this is closer to $O(N^p)$, with $2 < p < 3$. This is unacceptable in an online learning scenario, for instance, and quickly becomes unmanageable for very large datasets.

The most important thing about the kernel trick is that it allowed for *nonlinear solutions* to be found. Like Neural Networks, the kernel trick generalized the perceptron to a nonlinear setting, but in a rather different way. Using either the RBF or the polynomial kernel greatly increased the capacity of the SVM model, while preserving the convexity of the optimization problem. Moreover, the total number of hyperparameters (typically just 2) allowed for a quasi-complete grid search to be performed. Finally, Hammer and Gersmann (2003) show that SVMs with nonlinear kernels (such as the RBF or polynomial kernel) can approximate any measurable or continuous function up to any desired accuracy, given *enough training examples*.

The kernel trick, convexity, ease of use and firm grounding in the principles of Statistical Learning Theory were key ingredients for the popularity of Support Vector Machines. Another such ingredient was the empirical evidence: SVMs had become the state-of-the-art in many application areas and for many standard Machine Learning benchmark tasks, such as digit classification (Schölkopf and Smola, 2002). The clear separation between the choice of the kernel—which could be fitted to the task at hand—and the optimization algorithm, which worked for any suitable kernel, helped popularize the technique, despite the relatively high computational requirements.

2.3 Revisiting the generalization problem

In Section 1.4 we discussed the notion of inductive biases and how they are inevitable in Machine Learning. In this Chapter, we have

*. Specifically, $N_{SV}/N \rightarrow 2\varepsilon_{\Phi}$, as the training set size increases, where ε_{Φ} is the minimum possible error achieved by a linear classifier in the feature space defined by Φ

shown how it is possible to take a very simple Machine Learning algorithm — the Perceptron — and make it more general* in a few different ways. In the following, we discuss the consequences of making these choices, frame the choices as inductive biases, and describe potential problems with applying these solutions to modelling data with many underlying variabilities.

2.3.1 Local kernel machines

The bulk of the predictive power of an SVM (as presented in equation 2.6) of the model lies within the kernel used, i.e. in the similarity measure used to compare the test point with the support vectors. In the case of the RBF kernel this similarity measure is a function of the Euclidean distance between the two. Using such similarity measures only makes sense in a case where a test point's neighborhood is meaningful, i.e. if this neighborhood (in the Euclidean distance sense) is populated[†] and if a certain *smoothness prior*[‡] applies. In high dimensions (the case we are interested in), the lack of neighbors will reduce the SVM algorithm estimator to 1-nearest neighbor or a constant, both of which are known to be not good.

The smoothness prior is, however, more defensible and has been a major hypothesis in most non-parametric methods that are based on *local*, i.e. neighborhood-based, estimators for similarity. The reason it is popular is because it seems quite natural for an estimator to be a smooth function of its input. And yet, as shown by Bengio *et al.* (2006) and Bengio and LeCun (2007) it is easy to find examples where seemingly small deviations in the input produce big changes in the Euclidean distance between the original and the transformation.

The simplest case is that of an alphanumeric character being translated along some axis. In our *mental representation* of the digit, such a transformation is very smooth and almost negligible as we do not change our opinion of the label of the character. Yet if we computed the Euclidean distance between two images containing translated versions of the same character, even a 1-pixel translation could induce big changes in the distance computed. The smoothness intuition breaks down because our vision system is *automatically* computing a representation in which such transformations as rotation, scaling and translation are *smooth*.

*. In the sense of increasing its capacity.

†. Otherwise the similarity with the support vectors is close to zero

‡. The notion that given $x \approx y$, $f(x) \approx f(y)$, with f being the target function to be learned

Another example is a decision boundary between two classes, which takes the shape of a sinusoid. This sinusoid separating the two classes *can* be modeled by an SVM with an RBF kernel, given sufficient a number of support vectors. Yet as shown by Bengio *et al.* (2006) the number of support vectors needed for accurately modeling such a boundary with an RBF kernel is proportional to the number of “bumps” in the sinusoid. Clearly, we can expect decision boundaries for realistic problems (image classification with translation, rotation, scaling, deformations, backgrounds) to change rapidly locally. Therefore we should also expect local kernels such as the RBF kernel to be an inefficient way of representing such boundaries, especially in high-dimensions. An extreme example for the latter is the *d-bits parity problem* (deciding whether a given number of bits has an even number of ones or not). It can be shown that in order to solve this problem with RBF kernels and SVMs one would need at least 2^{d-1} support vectors!

On the other hand, the sinusoid itself can be compactly represented by a mathematical description and parity can be solved with a hierarchical tree-like model whose depth grows only logarithmically in d and size linearly in d . These examples point to the idea that one is able to come up with *compact representations* for the problems at hand. One could argue that such representations assume strong prior knowledge about the task at hand. Yet we shall argue that the parity example and, especially, the image translation example, provide us with a principle that can be generalized to a variety of problems: that of building *hierarchies of representations*.

2.3.2 Shallow architectures

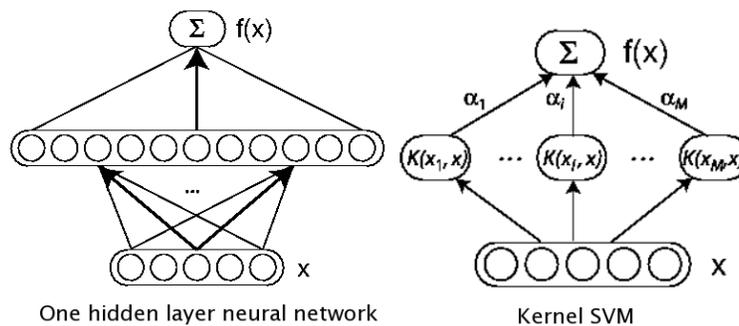
Clearly, the example of image translation is not a hard one to “fix” in practice. There is even a solution* for learning with Support Vector Machines in such a scenario by modifying the kernel to be invariant to small local translations, rotations or scalings (Decoste and Schölkopf, 2002). However, hand-crafting kernels is not a solution that is feasible in the long-term: the sheer number of variations that natural images can have and the complexity of some of these transformations (such as backgrounds) produces a combinatorial explosion that is hard to deal with.

Work has been done on constructing convex optimization procedures for training Support Vector Machines with convex combinations of kernels; variations of this idea formed the basis of a popular research

*. This solution is somewhat impractical, as it involves generating extra support vectors by translating the original ones.

field called Multiple Kernel Learning (Bach *et al.*, 2004). Such work addresses the issue of no single kernel being the right choice all the time (in a finite sample size case) and, in a sense, introduces a hierarchical representation by adding another “layer”.

Bengio and LeCun (2007) argue that the Machine Learning community should be moving towards models that learn hierarchical representations of the data. It is argued that having *deep architectures* is a good way of overcoming the shortcomings of the *shallow architectures* described above. A shallow architecture is defined as a model with one layer of nonlinear processing units: in a neural network, this is a layer of sigmoidal units; in an SVM, this is the application of the nonlinear kernel for each support vector (both are illustrated in Figure 2.6). Therefore, **a deep architecture is defined as compositions of many layers of adaptive nonlinear processing units.**



◀ **Figure 2.6.** Shallow architectures: one hidden-layer neural network (left) and nonlinear Support Vector Machines (right)

As noted in Sections 2.1 and 2.2, both one-hidden layer neural networks and SVMs with RBF kernels are known to be able to model any mapping from input to output, as long as sufficient samples (and hidden units) are allowed. So why care about deep architectures then? The answer lies in the *efficiency* of the learned representations. The d -bit parity problem illustrates this perfectly: a shallow architecture needs an exponential number of units to solve it, while a deep representation can do some with a tree whose depth grows with $\log d$. Bengio and LeCun (2007) cite other such examples, mainly from Boolean circuit theory, that show that many functions that can be representing compactly with a deep architecture require a much larger number of components using a shallow one.

Bengio (2009) demonstrates examples from monotone weighted threshold circuits, where an architecture of depth k requires a polynomial number of units* to represent a certain class of functions, while an architecture of depth $k - 1$ requires an *exponential* number of such units.

*. Polynomial in d

These results do not directly apply to real-life architectures and non-toy problems such as image classification. Yet they should be taken as a strong indication that learning deep architectures could potentially be worthwhile because of the gains in the efficiency of representations.

2.3.3 The need for an appropriate inductive bias

So far, we have encountered a few of the inductive biases that are at the basis of numerous Machine Learning algorithms and research. An important class is the one of **smoothness or locality assumptions**: this is the intuition that smooth changes in the input space imply smooth changes in the output space. More specifically, this means that small perturbations of an input should not change the corresponding output.

This assumption is implicit in the choice of the RBF kernel from Section 2.2: for a given test point, the decision function that is learned computes the Euclidean distance between it and the support vectors and scales the decision by the variance and an exponential. The smoothness assumption certainly seems natural and is ubiquitous in many of the Machine Learning algorithms that one encounters.

In this particular example, smoothness is a property of the decision function or boundary with respect to the changes in the input. A function or a hypothesis can also be smooth in terms of its parameters or in terms of the internal representation that the model learns. We argued in Section 2.3.1 that input smoothness—or simply *the locality assumption*—might not be a good idea when the data that one is trying to model has certain more realistic properties.

A related inductive bias is the **manifold assumption**: while we have not explicitly encountered it so far, it is frequently used in Machine Learning. It assumes that the data lies on a lower-dimensional manifold that is embedded in a high-dimensional space that is presented to us in the training data. A variety of algorithms use this modelling assumption for extracting a better representation of the input: Laplacian Eigenmaps (Belkin and Niyogi, 2002), ISOMAP (Tenenbaum *et al.*, 2000) and LLE (Roweis and Saul, 2000) are representative examples. Most of these techniques also assume some sort of input smoothness or locality in the Euclidean or Riemannian space of the inputs.

Bengio and LeCun (2007) provide an example that we have already touched upon in Section 2.3.1: let us consider the manifold of examples coming from a single class and their class-invariant transformations, such as translation, rotation, scaling and so forth. Each of these transformations (and combinations thereof) creates a highly non-linear

surface in the *input space*. Most of the methods that incorporate the manifold assumption will be able to represent this manifold well enough if one of the two conditions are met: either the manifold is smooth as a function of the input (not true of the transformations considered) or it is densely sampled. Bengio and LeCun (2007) argues that the latter implicit assumption of high-density sampling of data points near the curvature of the manifold is unrealistic given a manifold with high curvature, such as the one described above. Thus there is a need to consider models that do not necessarily depend on a high density of examples about the curvature of the manifold, methods that would be able to model such curvature in a more parametric fashion.

The **principle of large-margin** classification (or regression) is a clear example of a procedural inductive bias, for one is favouring a certain class of hypotheses (those that maximize the margin) to the detriment of others. While it has an excellent intuitive and geometric interpretations, as well as good theoretical properties (Cortes and Vapnik, 1995; Vapnik, 1998), this principle is still not enough to handle many difficult learning tasks.

All these biases try to solve the generalization problem by encoding certain intuitions that make it possible to overcome the inherent ill-posedness of the problem of learning. All of them have great success, empirically speaking, but we have argued that at least some of these inductive biases have certain fundamental limitations. To be sure, we shall not argue that a method should be completely oblivious to input smoothness or locality and should take advantage of it when possible. We are also not suggesting that the kind of problems (we called them *realistic* in the above) that we consider will have completely arbitrary non-smooth decision boundaries*. The argument is simply that, generally speaking, one cannot expect the data and the associated decision boundaries between classes to be well-behaved and smooth[†] and that we need our inductive biases to reflect that.

So far we have presented several of the arguments made by Bengio and LeCun (2007) and Bengio (2009), which demonstrate the limitations of shallow architectures. We have argued in the above that shallow architectures such as an SVM with a local kernel (RBF, for instance) are not well suited for modelling highly-varying decision boundaries where the manifold assumption need not apply.

There are several ways to think about how to overcome this problem. A principled, yet potentially very time-consuming way, is to hand-craft models that are able, for particular instances of complicated func-

*. It is hard to imagine generalizing in such a general scenario.

†. Like the well-known Swiss roll toy problem (Tenenbaum *et al.*, 2000).

tions, to simplify the learning problem. This is the approach of incorporating translation or rotation invariance in models applied on image classification tasks, such as described by Decoste and Schölkopf (2002). A different approach is presented by the hypothesis that models with multiple compositions of non-linearities are simply better suited for such tasks, since, as argued by Bengio and LeCun (2007), such composition appears to give non-local properties to a deep architecture. The *belief* is that ultimately, the latter approach will prevail, partly because of the effort that one has to put into designing or changing methods to be invariant to all transformations and partly because of the fact that for methods like SVMs with RBF kernels the size of the training set could potentially scale combinatorially with the number of possible transformations.

Before jumping to the description of the methods that we shall employ throughout this thesis, we need to make one more thing more precise. Throughout the discussion on the need of having methods that generalize well non-locally and that can represent efficiently (hierarchically) data and decision boundaries, we have only hinted at what these realistic complicated data and decision boundaries are. At a higher level, this is the set of tasks whereby one must extract abstract concepts from data. More specifically, these tasks include inferring meaning from images, texts, sounds and relating these to each other in meaningful ways (i.e., finding similarities). These are the same tasks that we expect the human brain to accomplish.

So why not make sure that the class of models that we consider can actually efficiently represent such concepts or functions? This is a *basic objective of the work in deep architectures*. It should be emphasized again that the real-life need and usefulness of deep architectures for such tasks is only a *hypothesis* so far. In the light of results presented in Chapters 4 and 6, as well as the overview in Section 2.6, the evidence supporting this hypothesis is strong; there are also theorems that show that certain types of deep architectures are able to model quasi-arbitrary input-to-output mappings. However, there is no definitive proof that deep architectures are better than shallow ones. Likely, there *cannot* be such a proof, since the desire to have a function class represented by compositions of multiple non-linearities is simply an *inductive bias*, and no inductive bias is universally better than any other, per the No-Free Lunch theorem.

Thus the refined version of the deep hypothesis is simply stating that for a given class of complex, interesting and very non-local or non-linear functions of the input deep architectures could be more appropriate than shallow ones. This class of functions, called the *AI-set*

by Bengio and LeCun (2007) (because they correspond to what an intelligent agent should be able to learn from data), is of interest to researchers in Machine Learning, as it represents a challenging set of learning problems and advances the field towards a very meaningful goal.

2.4 Deep Belief Networks

Assuming that one is convinced of the need for deep architectures, how does one go about building such hierarchies of composed nonlinearities? Would any such hierarchy work or does one require special algorithms to train them?

Convolutional Networks (LeCun *et al.*, 1998) are one of the earliest examples of successful deep architectures and are based on the idea of emulating some of the aspects of the visual system*. These multi-layer networks were inspired by earlier work on the Neocognitron (Fukushima *et al.*, 1983). Constraints on weights—such as weight sharing, to emulate convolutions—and sparse layers made it possible to train much more efficiently the whole network using backpropagation. Convolutional Networks of 5-6 layers can be easily trained and typically give state-of-the-art results on benchmarks such as MNIST.

However, training *unconstrained* neural networks with an equivalent number of layers using backpropagation was a difficult problem up to now. It is believed that the presence of many local minima and symmetries made it hard for backpropagation to find a good set of parameters (Bengio, 2009). Deep Belief Networks are a promising solution to this problem.

Deep Belief Networks (DBN), introduced by Hinton *et al.* (2006), is the culmination of efforts to successfully train densely connected multi-layer neural networks. Such networks provided a deep, nonparametric[†] and *nonlocal* representations of the input. These networks can also be trained efficiently with algorithms that are only *linear* in the size of the training set and they allow one to compute representations for unseen samples efficiently. Moreover, the ideas behind DBNs turned out to be some rather general principles that could and would be applied to other architectures as well. The key ingredients that made the DBN approach work are:

*. One can argue that the visual system of primates is a good example of a hierarchical architecture for processing streams of visual data.

†. In the sense of allowing ourselves to increase the number of hidden units with more data

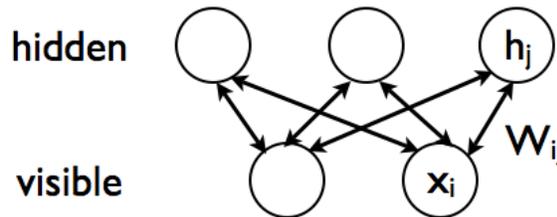
- Training layers in a greedy, layer-wise fashion.
- Unsupervised learning before supervised training.
- The usage of Restricted Boltzmann Machines (RBM).
- The efficiency of Contrastive Divergence.

We start by presenting RBMs and Contrastive Divergence.

2.4.1 Restricted Boltzmann Machines

The basic building block of a DBN is a Restricted Boltzmann Machine (Smolensky, 1986; Hinton and Sejnowski, 1986). A Boltzmann Machine (Hinton *et al.*, 1984; Hinton and Sejnowski, 1986; Ackley *et al.*, 1985) is a network of symmetrically connected stochastic units, whose probability of being “on” is the sigmoid of the weighted sum of the states of the other units. An RBM is a special type of Boltzmann Machine, in which connectivity is restricted, inference is simple and tractable and which has an efficient learning algorithm for learning the weights between units. A basic RBM is presented in Figure 2.7.

► **Figure 2.7.** A Restricted Boltzmann Machine with visible units \mathbf{x} , hidden units \mathbf{h} and weight matrix W . Biases \mathbf{b} and \mathbf{c} have been omitted.



An RBM is composed of two types of units: visible units $x_i, i \in 1 \dots D$ and hidden units $h_j, j \in 1 \dots N$. Visible units correspond to observed inputs, and \mathbf{h} is a vector of unobserved latent variables. A matrix of weights W connects \mathbf{x} and \mathbf{h} . Mathematically, an RBM specifies a bi-partite graph over \mathbf{x} and \mathbf{h} using W , as well as a joint probability distribution function over visibles and hidden

$$P(\mathbf{x}, \mathbf{h}) = \frac{1}{Z} \exp(-\mathbf{h}'W\mathbf{x} - \mathbf{b}'\mathbf{x} - \mathbf{c}'\mathbf{h})$$

where Z is the normalization constant and \mathbf{b} and \mathbf{c} are the so-called *visible unit biases* and *hidden unit biases*, respectively. As written above, both the visible and the hidden units in this RBM are so-called Bernoulli units—they are 0/1-valued—but extensions to other members of the exponential family are possible (Welling *et al.*, 2005; Bengio *et al.*, 2007).

In RBM terminology, we define

$$\text{energy}(\mathbf{x}, \mathbf{h}) = \mathbf{h}'W\mathbf{x} + \mathbf{b}'\mathbf{x} + \mathbf{c}'\mathbf{h}$$

which is the so-called *energy function*. An RBM is a special graphical model in that the problem of *inference** is particularly easy to solve: because of the bi-partite nature of the graph, it is easy to show that

$$p(x_i = 1|\mathbf{h}) = \text{sigm} \left(b_i + \sum_j W_{jk} h_j \right)$$

$$p(h_j = 1|\mathbf{x}) = \text{sigm} \left(c_j + \sum_i W_{ij} x_i \right)$$

Not only is inference possible without any approximations, but the distributions $p(\mathbf{h}|\mathbf{x})$ and $p(\mathbf{x}|\mathbf{h})$ factorize completely! This is important since it makes the inference problem in an RBM extremely efficient and one uses this fact to get an efficient *learning* algorithm as well.

In principle, *learning* in an RBM, by, for instance, modifying W , \mathbf{b} , and \mathbf{c} to maximize the likelihood of the data $p(\mathbf{x})$, is intractable since computing the latter involves a summation over all the possible states of \mathbf{h} (we assumed that each h_j is 0/1-valued, therefore an \mathbf{h} with N units will have 2^N states over which we need to sum). However, consider a Gibbs Markov chain, where we repeatedly sample \mathbf{h} given \mathbf{x} , \mathbf{x} given \mathbf{h} and so on. It can be shown (Geman and Geman, 1984) that such a procedure will give rise to an unbiased sample from the joint $p(\mathbf{x}, \mathbf{h})$ as we sample infinitely many times.

Let us denote $(\mathbf{x}_k, \mathbf{h}_k)$ the k -th sample from this Markov chain. If we knew $(\mathbf{x}_\infty, \mathbf{h}_\infty)$, we could use it to compute an unbiased estimate of the gradient of the log-likelihood with respect to any of the parameters of the RBM. *Contrastive Divergence* (Hinton, 2002) is the idea that one does not need to let the Gibbs chain reach equilibrium in order to compute the log-likelihood. Let \mathbf{x}_0 be a training sample, then:

$$\log p(\mathbf{x}_0) = \log \sum_{\mathbf{h}} p(\mathbf{x}_0, \mathbf{h}) = \log \sum_{\mathbf{h}} \exp(-\text{energy}(\mathbf{x}_0, \mathbf{h})) - \log \sum_{\mathbf{h}, \mathbf{x}} \exp(-\text{energy}(\mathbf{x}, \mathbf{h}))$$

Therefore, the gradient of the log-likelihood wrt. to, say, W is

$$\frac{\partial \log p(\mathbf{x}_0)}{\partial W} = - \sum_{\mathbf{h}_0} p(\mathbf{h}_0|\mathbf{x}_0) \frac{\partial \text{energy}(\mathbf{x}_0, \mathbf{h}_0)}{\partial W} + \sum_{\mathbf{x}_k, \mathbf{h}_k} p(\mathbf{x}_k, \mathbf{h}_k) \frac{\partial \text{energy}(\mathbf{x}_k, \mathbf{h}_k)}{\partial W}$$

and an unbiased sample of this gradient would be

$$- \frac{\partial \text{energy}(\mathbf{x}_0, \mathbf{h}_0)}{\partial W} + \frac{\partial \text{energy}(\mathbf{x}_k, \mathbf{h}_k)}{\partial W}$$

*. Inferring the distribution of one set of variables given the rest

when $k \rightarrow \infty$. Contrastive Divergence is the idea of simply taking $k = 1$ (i.e. run the Gibbs chain for one step) and using $(\mathbf{x}_1, \mathbf{h}_1)$ for computing an *estimator* of the gradient of the log-likelihood. We shall call the gradient estimate obtained after k steps of Gibbs sampling CD_k , with CD_1 corresponding to the original Contrastive Divergence. Thus,

$$CD_k(W) = -\mathbf{h}_0 \mathbf{x}'_0 + \mathbf{h}_k \mathbf{x}'_k .$$

Intuitively, CD_k is modifying the *energy* of particular configurations of the RBM such that samples from the true distribution $(\mathbf{x}_0, \mathbf{h}_0)$ get lower energy, while samples from the distribution generated by the RBM $(\mathbf{x}_k, \mathbf{h}_k)$ get higher energy. Interestingly, such energy-landscape modifications are *non-local*, because an RBM specifies a probability distribution over all the possible configurations of (\mathbf{x}, \mathbf{h}) and this has to sum to unity.

It is not clear that Contrastive Divergence should work: the CD_k algorithm not only has variance due to sampling, is also has a bias. Carreira-Perpiñan and Hinton (2005) show that while this approximation *is* a biased estimate of the true gradient, this bias is generally small. Bengio and Delalleau (2009) study this bias and show how it decreases with k . They also show that the *sign* of the CD estimator is most often right (which is what matters most for gradient descent). It was also found that the CD_1 estimate is generally close to the maximum likelihood estimate and that one could in principle use CD_1 as an *initialization strategy* before running a more expensive log-likelihood computation (such as CD_k , for a big k).

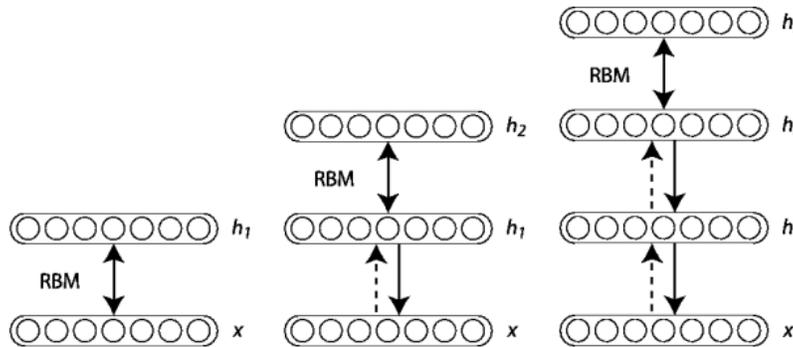
A disadvantage of CD is the inability of computing an objective function of performance, so we do not have any stopping criterion, because there is no closed form for the objective function minimized by CD (and true likelihood computation is normally intractable); moreover, Sutskever and Tieleman (2010) show that CD is not the gradient of any function. Practically speaking, CD_1 is a relatively fast way of learning in an RBM and has, so far, produced excellent results when used as a building block for a DBN.

2.4.2 Greedy Layer-wise training of DBNs

RBMs, by themselves, are *shallow* models of the data. They learn an unsupervised representation of the input and can also be used to generate samples from their distribution. They are most useful however as part of a deep architecture called Deep Belief Networks.

Deep Belief Networks rely upon the very simple idea that one can build hierarchical (deep) models of the data in a greedy layer-wise fash-

ion, as illustrated in Figure 2.8. One models an input distribution \mathcal{X}



◀ **Figure 2.8.** The process of greedy layer-wise training of a Deep Belief Network. RBMs are trained sequentially to model their input, with previous layers’ parameters being fixed. Figure by Larochelle et al. (2007)

by training an RBM with *CD* for a certain number of steps. For each $\mathbf{x}^i \in \mathcal{X}$ one can then recover a posterior distribution $p(\mathbf{h}^i | \mathbf{x}^i)$ (from equations 2.7) over the hidden units of the RBM, also known as a hidden representation. These hidden representations are then used as an *empirical distribution* (i.e. a training set) for another RBM, called the second-layer RBM, which will, in turn, model these representations with another set of hidden units and whose parameters will be learned via *CD* as well. This process can be iterated as many times as needed. It should be stressed that only one layer is trained at a time, in a feed-forward fashion, with the previous layers’ parameters being “frozen”.

Such a greedy layer-wise procedure can be justified using a variational bound on the log-likelihood as we add more and more layers (Hinton *et al.*, 2006; Le Roux and Bengio, 2008). At the end of training, one can obtain the top-level hidden representations by simply computing the posterior probabilities for each layer in turn. We have already seen that these conditionals are identical to layers in a normal feed-forward neural network, therefore, once trained, a DBN is identical performance-wise to a neural network (for computing hidden representations).

Once built this way, one can either continue training the DBN using a variant of the wake-sleep algorithm (Hinton *et al.*, 1995, 2006) to build a better generative model of the data or one can treat the learned weights as initial values for a feed-forward multi-layer neural network. Such a network can then be used in conjunction with any standard objective function for training MLPs: Hinton and Salakhutdinov (2006) used the reconstruction error cost to build a deep auto-encoder for dimensionality reduction purposes, while Bengio *et al.* (2007) added a

supervised layer on top of the last layer learned by a DBN and used such networks for classification problems.

Hinton *et al.* (2006) showed that on the classical MNIST benchmark* DBNs outperformed other black-box methods (i.e. methods that do not incorporate prior knowledge about the digits in their training), by obtaining a classification error rate of 1.2%. This was an important result since it showed that training multilayer neural networks is feasible, that this method produces state-of-the-art results, and that a deep architecture is better than the state of the art shallow architecture (Support Vector Machines). The paper has also basically launched the field of deep architectures, advances in which are described below.

2.5 Exploring the greedy unsupervised principle

Hinton *et al.* (2006)'s work was a breakthrough in that it presented for the first time an efficient and generic way of training a neural network with more layers than traditionally used. Bengio *et al.* (2007) explored in more depth the general principles behind Hinton *et al.* (2006)'s approach and found that

- there is evidence to support the hypothesis that greedy layer-wise initialization of weights helps with the *optimization* process, by placing the initial weights in a region with a good local minimum[†].
- importantly, greedy *unsupervised* initialization helps more than greedy *supervised* initialization.

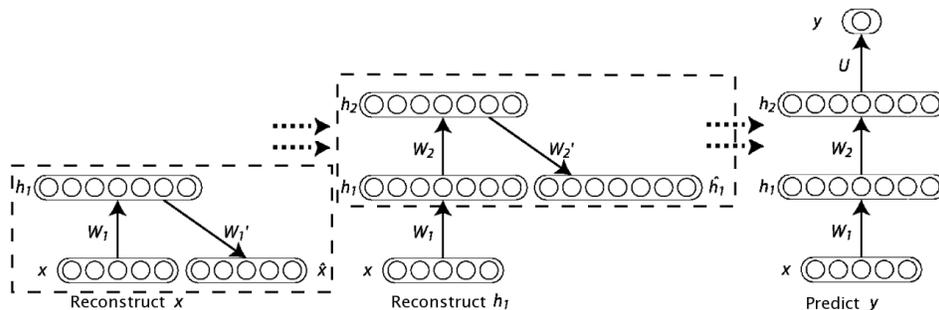
To highlight these points, Bengio *et al.* (2007) have also proposed an alternate way of training a multilayer neural network, called Stacked Auto-Associators (SAA). Shown in Figure 2.9, these models take the main idea of DBNs—greedy construction of hierarchical neural network representations—and apply it to the case of auto-associators (Anderson and Mozer, 1981). The latter is an architecture of feed-forward neural networks which can be best described as attempting to reconstruct its input. A basic auto-associator is shown in Figure 2.9 (leftmost). Assuming input are in $[0; 1]^D$, the auto-associator (AA) model is simply

$$\begin{aligned}\mathbf{h} &= \text{sigm}(W\mathbf{x} + \mathbf{b}) \\ \hat{\mathbf{x}} &= \text{sigm}(W'\mathbf{h} + \mathbf{c})\end{aligned}$$

*. Digit classification: <http://yann.lecun.com/exdb/mnist/>

†. However, see Section 6.7.4 for a refined version of these experiments.

where $\hat{\mathbf{x}}$ is called the *reconstruction* of \mathbf{x} . The parameters of the network are typically optimized via backpropagation and gradient descent, with the objective function (called the *reconstruction error*) being either the squared loss $\|\mathbf{x} - \hat{\mathbf{x}}\|^2$ or the cross-entropy $-\sum_{i=1}^D x_i \log \hat{x}_i + (1 - x_i) \log(1 - \hat{x}_i)$.



◀ **Figure 2.9.** Greedy layer-wise training of a Stacked Auto-Associator. Auto-associators are trained sequentially to reconstruct their input, with previous layers' parameters being fixed. A supervised layer is added on top and backpropagation is then used to optimize the whole network. By Larochelle et al. (2007)

At the end of training, a representation \mathbf{h}^i can be found for each example \mathbf{x}^i . As with DBNs, AAs can be “stacked” by considering these \mathbf{h}^i as examples to another auto-associator and so on until one is satisfied with the number of layers in the network. Afterward, a supervised layer can be added on top of the last layer and the whole network can then be trained using backpropagation.

One of the interesting hypotheses presented by Hinton *et al.* (2006) was that unsupervised initialization is key to obtaining good performance in a deep network. Bengio *et al.* (2007) and Larochelle *et al.* (2009) verify this by greedily stacking *supervised* one-hidden-layer neural networks (where the output layer is discarded after training each layer). It was observed that such a strategy degrades the performance of the network quite significantly. Partially supervised training, where the unsupervised gradient is added to the supervised one, seems to be a good compromise in RBMs (Larochelle and Bengio, 2008), but controlling the trade-off between the two is still an open question.

While RBMs/DBNs are generative models of the data, auto-associators are not. This means that we can sample from the distribution learned by an RBM/DBN and observe it. This is especially important for sanity checking. Whereas an RBM in conjunction with CD, is, in a sense, learning to reconstruct the input *distribution*, the auto-associator learns to reconstruct a specific example. While it is possible to compute an out of sample estimate of the objective function that is minimized by the auto-associator, this is not possible for an RBM: the objective function minimized by Contrastive Divergence cannot be computed, and

the likelihood of an input under the current model is intractable for all but the smallest models. Note that Bengio and Delalleau (2009) show that the gradient of the reconstruction error in an auto-encoder is related and is an approximation of the CD_1 gradient estimator in an RBM. However, initializing a multi-layer deep network with RBMs works slightly better in practice than doing so with auto-associators (details are in Chapter 4).

2.6 Recent developments in deep architectures

The broader applicability of the unsupervised greedy layer-wise initialization of a deep network and the strong results on MNIST have influenced many groups of researchers to further pursue the research on deep architectures. In the following, we describe most of the recent work on the subject.

2.6.1 Deep Belief Networks

Goldberger *et al.* (2005) presented a method for linearly transforming the input space such that in the transformed space the k -NN algorithm performs well. This *Neighbourhood Component Analysis* (NCA) criterion has been applied to training a Deep Belief Network as well (Salakhutdinov and Hinton, 2007a). This criterion makes the network learn what is essentially a nonlinear NCA transformation in which k -NN should perform well. Adding it improved the results on MNIST to 1.08%. Interestingly, the authors experimented with allocating certain (output) units to performing unsupervised learning and others to supervised learning. The latter split gives a error rate of 0.97% on MNIST.

Apart from impressive results on MNIST, DBNs have been used for learning a deep auto-encoder model of the data. Trained on MNIST digits, a two-dimensional code learned by a DBN produces well-separated “clouds” of classes, and outperformed PCA and Latent Semantic Analysis on reducing the dimensionality when trained on a corpus of documents (Hinton and Salakhutdinov, 2006). A so-called “semantic hashing” method extends the basic RBM model for word count data using a “Constrained Poisson Model” (Gehler *et al.*, 2006). The deep representations learned by the DBN are 0/1-valued and, therefore, allow for fast bit counting routines based on the Hamming distance to be used for

searching and document retrieval. They also outperform the TF-IDF representation accuracy-wise (Salakhutdinov and Hinton, 2007b).

Deep Belief Networks have also been used for learning the covariance kernels for Gaussian processes (Salakhutdinov and Hinton, 2008). In that setting, stacked RBMs are used to initialize a deep neural network as described in the above. Afterward, a Gaussian covariance matrix is initialized with the top-level features learned by the DBN and its parameters are then optimized using maximum likelihood learning and the supervised labels from the dataset. Such an approach can leverage vast amounts of unlabelled data for providing an initial estimate for the covariance matrix. Yet its drawback is the computational efficiency: each gradient update of the covariance matrix is $O(N^3)$, where N is the number of labeled examples.

Osindero and Hinton (2008) show that it is possible to add *lateral connections* between the visible units of an RBM, by modifying the energy function to local interactions between visible units. When trained on a set of natural image patches, such lateral connections have the effect of *whitening* the data, i.e. they capture and filter out the pairwise correlations between pixels. More interestingly, when used on hidden layers (in the same greedy procedure that is used for training DBNs), not simply on the input layer, these lateral connections seem to give rise to a subjectively better model of natural images (when compared to a normal RBM), as they tend to improve the coherence between pixels. The images generated by the DBN/RBMs with lateral connections are also closer, statistically speaking, to natural images.

Sparsity constraints have also been investigated in the context of DBNs. Lee *et al.* (2008) present a method for modifying the biases of the hidden units of an RBM so as to encourage sparse representations for modelling images. Earlier work includes learning topographic models of natural images with so-called “products of student- t experts” (Osindero *et al.*, 2006; Hinton *et al.*, 2006), which also recovers Gabor-like filters.

Nair and Hinton (2010) show that by replacing the binary units with rectified linear units one can improve the performance of RBMs and stacks of RBMs on two image modelling tasks, NORB (LeCun *et al.*, 2004) and Labeled Faces in the Wild (Huang *et al.*, 2007). This is in line with work done by Bengio and Glorot (2010), who observed that in the case of a modified version of auto-encoders (presented in Section 2.6.2), the choice of nonlinearity plays a big role, especially in how well stochastic gradient descent can optimize deep architectures: their conclusion is that the standard sigmoid nonlinearity is less suited than the hyperbolic tangent, which is in turn worse than a rectifier

non-linearity.

Convolutional Deep Belief Networks

Convolutional Deep Belief Networks (Lee *et al.*, 2009) are a significant advance in making Deep Belief Networks scalable for large high-dimensional data. They are a successful attempt at creating a model that has the desirable properties of a Convolutional Neural Network (LeCun *et al.*, 1998)—translation invariance (via the convolutional structure and a probabilistic max-pooling operator) and appropriateness for sequential and image data—and the desirable properties of Deep Belief Networks—the generative model and the fact that unsupervised learning can be done in a natural way. The probabilistic max-pooling techniques makes it possible to have a probabilistically sound inference process: hierarchical inference can be done in a top-down and bottom-up way on full-sized images. Moreover, the results show that the features learned are hierarchical in nature and the model can obtain state-of-the-art results on MNIST and Caltech-101 (Fei-Fei *et al.*, 2004).

Discriminative RBMs

Discriminative RBMs (Larochelle and Bengio, 2008) are another way of training an RBM. The main idea is that of training an RBM as a classifier. This is possible by making the target or label part of the (visible) input and exploiting the fact that in a classification setting one can typically afford to enumerate all possible classes (unless there are very many classes, obviously). Let \mathbf{x} be the input as before and y be the label. It turns out that computing $p(y|\mathbf{x})$ is tractable and exact (Salakhutdinov *et al.*, 2007). This also means that one can compute the *exact gradient* of $p(y|\mathbf{x})$ wrt. to the model parameters and perform gradient descent afterward. Interestingly, the best results are obtained by a *hybrid* model which trains an RBM using both Contrastive Divergence *and* the discriminative criterion $p(y|\mathbf{x})$.

Deep Boltzmann Machines

An approach closely related to RBMs stacked into Deep Belief Networks is that of *Deep Boltzmann Machines* (DBM) by Salakhutdinov and Hinton (2009). DBMs are different from DBNs in that they are a graphical model in which all connections between layers are undirected. This allows for the (approximate) inference procedure in this model to

include top-down influences, making it possible to better model contextual effects and better incorporate the prior represented by higher-level abstractions in upper layers. Another interesting property of DBMs is that the optimization procedure changes all the parameters at the same time, overcoming the greedy layer-wise “constraint” that typical training of stacked RBMs imposes.

Salakhutdinov and Hinton (2009)’s version of Deep Boltzmann Machines is however considerably slower than a comparable DBN, because of the need for iterative inference, a relaxation process involving all the layers trying to find a coherent “explanation” of the input. Salakhutdinov and Larochelle (2010) suggest a different approximate inference scheme, which uses a separate model that initializes the values of the latent variables of the DBM in one bottom-up pass. This “learning to do inference” scheme is at most 3 times slower than a DBN. Salakhutdinov and Hinton (2009) show that Deep Boltzmann Machines can obtain impressive empirical results and have the current record on the best performance on MNIST without using prior knowledge: 0.95%.

Modelling high-dimensional time-series

Hierarchical representations for time series, based on the RBM/DBN ideas, have also been explored: Taylor *et al.* (2007) showed how compositions of Conditional Restricted Boltzmann Machines (CRBM) could be used for learning nonlinear generative models of human motion data. The CRBM model is similar in spirit to the gated RBM model, used by Memisevic and Hinton (2007) and Memisevic and Hinton (2010) for modelling and unsupervised learning of natural image transformations. Taylor and Hinton (2009) extended their CRBM work by including multiplicative three-way interactions and factoring these interactions to reduce the computational complexity of the problem. Ranzato *et al.* (2010) and Ranzato and Hinton (2010) use factored 3-way interactions in an RBM to model natural images: the key difference between this work and that of Taylor and Hinton (2009) being that the latter presents *conditional* 3-way models, whereas the former presents *joint* 3-way models.

Taylor *et al.* (2010) have explored using the so-called Implicit Mixtures of CRBMs for human pose tracking. They show that learning is efficient with such models and that they can learn coherent models of several activities. Models similar to Ranzato and Hinton (2010)’s factored 3-way CRBM have been used for obtaining state-of-the-art results (Dahl *et al.*, 2010) on the TIMIT (Fisher *et al.*, 1986) phone recognition task. Finally, Lee *et al.* (2009) obtain good performance on

a variety of audio classification tasks using their Convolutional DBN approach.

Learning and inference strategies in RBMs

Alternatives and improvements to Contrastive Divergence for training RBMs have been investigated, too. Tieleman (2008) presents the Persistent Contrastive Divergence (PCD) algorithm, which is based on the idea that between the updates to the parameters of an RBM the model represented by the RBM changes only slightly. Thus we can use initialize the Markov chain at the next time step at the state at which it ended in the previous step. This works better than standard *CD* and is just as efficient. An extension is the Fast Weights PCD (Tieleman and Hinton, 2009) or FPCD algorithm which introduces an auxiliary set of “fast weights” that make the Markov chain mix faster. The faster mixing makes the learning process convergence faster as well.

Tempered MCMC (Desjardins *et al.*, 2010) has been used to improve mixing of the Markov chain during the negative phase of CD, such that it explores the modes of the distribution better. It seems to improve both the quality of samples and learning. Related work by Salakhutdinov (2010) explored using tempered transitions to encourage mixing and fine-tuning of the parameters in a Markov Random Field.

Herding (Welling, 2009b,a) is a rather different strategy: instead of trying to estimate the parameters of an RBM, it simply aims to produce samples similar to those from the training set via a dynamical system. An interesting property of this technique is that the distribution of the generated samples cannot be expressed (at any given time) as a simple function of the parameters, but is defined by the dynamics of the generating procedure. Breuleux *et al.* (2010) introduce an idea that extends the FPCD technique of Tieleman and Hinton (2009) with notions from Herding. This Rates-FPCD sampler for RBMs mixes well and it can also improve the model as one collects from samples from the model.

There has been some work on comparing several of the inductive principles for training RBMs: Marlin *et al.* (2009) perform several experiments on a variety of datasets and come to the conclusion that among Contrastive Divergence, Stochastic Maximum Likelihood (presented as PCD in the above), Ratio Matching and Pseudo-Likelihood, the principle of Stochastic Maximum Likelihood seemed most robust and better performing.

Analysis of RBMs, DBNs and related models

Advances have been made in understanding the theoretical properties of Deep Belief Networks. Sutskever and Hinton (2008) show that deep belief networks which are *narrow* can in principle approximate any input distribution. The number of layers needed to do so might be *exponential* in the dimensionality of the input. The authors also show that adding a hidden layer can increase the representational power of a DBN and show an algorithm for greedily learning a DBN that can approximate any distribution. Le Roux and Bengio (2008) show that an RBM is universal approximator as well (with an exponential number of hidden units) for discrete-valued input distributions and pose several open questions regarding the expressive power of DBNs with more than one layer. Le Roux and Bengio (2010) follow up on this work and prove that deep but thin DBNs do not need more parameters than shallow and fat architectures for universal approximation.

There is progress in providing a tool for comparing two RBMs with different architectures, for purposes of model selection. Salakhutdinov and Murray (2008) show that it is possible to approximate the ratio of normalization constants of two RBMs using Annealed Importance Sampling (Murray and Salakhutdinov, 2009), an importance sampling method that works well in high dimensions. In the same paper, it is also shown how one can estimate a bound on the log-probability that a multilayer DBN assigns to test inputs. Both of these results allow one to estimate the performance of an RBM/DBN as a *generative* model of the data. Using these measures, the authors have also discovered that gradually increasing the number of Gibbs sampling steps during training with Contrastive Divergence gives much better results as compared to CD_1 , in terms of input likelihood (but not as clearly in terms of finding features that are good for a deep classifier).

On the other hand, Long and Servedio (2010) provides certain worst-case scenario analyses of RBMs which state that it is NP-hard to approximate even coarsely the partition function of an RBM. The authors speculate that these worst-case scenario hardness results do not necessarily contradict real-world experience with RBMs, as most often the parameters of models typically used are rather small; work and analysis by Bengio and Delalleau (2009), seems to confirm this observation, as they show that the bias of CD is proportional, in part, to the magnitude of the weights of an RBM.

Interesting developments have been made in trying to understand the invariances that are learned by certain deep architectures, notably DBNs. We have already mentioned that Lee *et al.* (2008)'s results suggest that sparse RBMs learn Gabor-filter-like features when trained on

natural images. Lee *et al.* (2009)’s follow-up work suggests that second layer filters correspond to the kind of features that neurons from the V2 cortical area encode: namely grating filters and other compositions of first layer features. Notably, the distribution of the responses of the units from the second layer to standard benchmark inputs corresponds to the distribution of V2 neurons. Goodfellow *et al.* (2009) present an analysis of Convolutional Deep Belief Networks and Stacked Auto-Associators in which they perform an interesting analysis of the invariance properties of these architectures: they find that generally, deeper layers are more invariant (when averaging across a range of transformations), but that for specific input transformations (e.g. translation) this does not necessarily hold.

While research into better understanding RBMs is still ongoing, the variety of work on the subject has made it possible to have common wisdom of tips and tricks that make them work: Hinton (2010) is one work that contains such knowledge.

2.6.2 Denoising Auto-encoders

Previously, we argued that the principle of initializing a network using a greedy unsupervised procedure was more general than if applied to the case of RBMs stacked in a DBN. Stacked Auto-Associators are an example where such a procedure works. A very promising new development is a modification of Stacked Auto-Associators called *Denoising Auto-Associators* by Vincent *et al.* (2008). *Algorithmically*, it is just a small modification to the basic SAA algorithm: instead of obtaining

$$\begin{aligned}\mathbf{h} &= \text{sigm}(W\mathbf{x} + \mathbf{b}) \\ \hat{\mathbf{x}} &= \text{sigm}(W\mathbf{h} + \mathbf{c})\end{aligned}$$

and using $\hat{\mathbf{x}}$ to minimize the reconstruction error $R(\hat{\mathbf{x}}, \mathbf{x})$, the idea is to compute $\hat{\mathbf{h}} = \text{sigm}(W\tilde{\mathbf{x}} + \mathbf{b})$ where $\tilde{\mathbf{x}}$ is some random perturbation (corruption) of \mathbf{x} . Typical perturbations include flipping some bits of \mathbf{x} or setting them to a default value. Given, $\hat{\mathbf{h}}$, we obtain $\hat{\tilde{x}}$ and minimize $R(\hat{\tilde{x}}, \mathbf{x})$, *not* $R(\hat{\tilde{x}}, \tilde{\mathbf{x}})$.

Conceptually and empirically, the differences are rather important. On its surface, this process makes the auto-encoder learn to predict random perturbations of \mathbf{x} from itself. It is also making the auto-encoder predict *parts* of \mathbf{x} from the non-perturbed parts; essentially, it is learning how to “fill in the blanks” of its input. Therefore, the model is learning features that are more *robust* to perturbations in the input.

Vincent *et al.* (2008) show that the corruption process can be viewed as a way of projecting the training data from the underlying manifold to the outside of the manifold. The denoising process can be seen as a way to learn the inverse mapping, which projects corrupted points back onto the manifold. When applied to the datasets described in Chapter 4 a deep architecture whose layers are trained using the denoising principle outperforms or equals all the deep and shallow architectures considered. A qualitative analysis of the learned features reveals that they become more localized and more “pronounced” as the level of corruption is increased. Moreover, for several datasets that were considered, the optimal level of corruption (the one that gives rise to the best supervised model) is relatively high: up to 40% of the inputs were zeroed by the corruption process.

Given the results obtained by Vincent *et al.* (2008), deep architectures with denoising auto-encoders is a model that can (almost) be considered a replacement for Deep Belief Networks. The reason is the sheer simplicity with which we can modify it to suit our needs: we can apply any of the tips and tricks that classical neural network research (weight sharing or convolutions, regularization, direct input-to-output connections etc). While the modeling assumptions of the DBN generative model are good to have, deep networks with denoising auto-encoders strike a better balance between good results and flexibility in changing the model to suit one’s need.

An illustration of this is the work of Larochelle *et al.* (2009) for adding asymmetric lateral connections in the hidden layers of the auto-encoders in the network. Compared to the approach of Osindero and Hinton (2008), adding these connections is much simpler conceptually and computationally, as it simply corresponds to adding an extra square weight matrix in between the hidden units of a denoising auto-encoder. Adding these connections improves performance on two character recognition tasks, the hypothesis being that lateral connections make it possible to introduce higher-order dependencies between units.

2.6.3 Energy-based frameworks

Energy-Based Models (EBMs), described by (LeCun *et al.*, 2006), are a framework for describing dependencies between variables by associating a scalar value, called energy, to each configuration of these variables. In such a framework, one can perform two tasks: the first is *inference*, which is finding the configuration of a given set of variables that minimizes the energy, given a fixed value for the rest of the variables. The second task is *learning*, which is finding an energy func-

tion that gives higher energy to empirically observed configurations of variables and lower ones to unobserved configurations. Many of the common models and algorithms can be cast in the EBM framework: this includes RBMs trained with CD, (denoising) auto-encoders and convolutional networks. This also includes models for which inference is not as easy, such as sparse coding Olshausen and Field (1996) or the Symmetric Encoding Sparse Machine described below.

An prototypical example of a model class that is part of EBMs is the *encoder-decoder* framework, proposed by Ranzato *et al.* (2008), based on their previous work (Ranzato *et al.*, 2007). In this work, they define an encoder as transforming the input into some representation and the decoder as the inverse process. In an RBM, this corresponds to computing the conditionals in equation 2.7 (likewise in an auto-encoder). Like in a classical auto-encoder, the weights are symmetric. A sparsity constraint is imposed on the representations learned by the encoder, by modifying the loss function to incorporate a term that corresponds to a factorized student- t prior over the representations. The loss function, which is minimized wrt. to W (during the process of *learning*), becomes:

$$L(\mathbf{h}, \mathbf{x}) = \|\mathbf{h} - W\mathbf{x}\|_2^2 + \|\mathbf{x} - W'\text{sigm}(\mathbf{h})\|_2^2 + \text{sparsity}(\mathbf{h}) + \|W\|_1$$

where the first term makes the output of the encoder as similar as possible to \mathbf{h} , the second makes the reconstruction as similar as possible to the input \mathbf{x} and the other two terms encourage sparsity in representations and weights. The loss is minimized with a coordinate-descent algorithm: first, given W and \mathbf{x} an \mathbf{h}^* is found that minimizes the loss. Using \mathbf{h}^* , we then take one step of gradient descent to modify W such that the loss is decreased. Note that once trained, the system, called Symmetric Encoding Sparse Machine (SESM) can be used as such: there is no need for optimization when recovering \mathbf{h} (during the process of *inference*) for a test sample is needed.

Ranzato *et al.* (2008) have found through experimentation on MNIST that SESM provides a good trade-off between the entropy of the learned representations and reconstruction error. Their results also point to the fact that the algorithm achieving the best such trade-off is the one that performs best in classification (which implies also that having a good reconstruction error does not guarantee good supervised performance). Finally, they have used the SESM module as a way to initialize a deeper architecture, with two layers, in a greedy layer wise fashion, in the same spirit as DBNs and SAAs. A peculiar result was obtained: if one constrains the last layer of the network to 10 units only, the features detected at the second level look like 10 prototype digits, even

though no supervised data was ever used by the algorithm.

A similar architecture was used by Ranzato *et al.* (2007) to learn a hierarchical network for image classification. The difference being that more prior knowledge about images were incorporated in the encoder-decoder framework, by using convolutions or weight sharing, with the ultimate goal of learning hierarchies of invariant features. A multi-layer network is initialized with such modules and then trained with a supervised layer on top; it achieves an impressive 0.64% on MNIST. A very similar technique was applied for image document compression (Ranzato and LeCun, 2007)

Jarrett *et al.* (2009) explore the variety of effects that come into play when designing a multi-stage (deep) architecture for object recognition. These effects include the choice of non-linearities, usage of unsupervised learning, and depth of the architecture, among many others. They design their experiments with an energy-based framework in mind, and consider encoder-decoder architectures that include convolutions, rectification layers, local contrast normalization, as well as comparisons between two pooling strategies (max and mean). Results on Caltech 101 (Fei-Fei *et al.*, 2004), NORB (LeCun *et al.*, 2004), and MNIST (LeCun *et al.*, 1998) suggest that supervised fine-tuning (as opposed to hand-crafted or random filters), unsupervised learning, depth, and absolute value rectification are essential for obtaining good performance; they obtain 0.53% classification error on MNIST.

2.6.4 Semi-supervised embedding for Natural Language Processing and kernel-based approaches

Collobert and Weston (2008) have extended work on using neural network for semantic role extraction (Collobert and Weston, 2007; Bengio *et al.*, 2003) by building a deep architecture for solving a host of Natural Language Processing problems at once. In a multi-task framework, they use ideas from convolutional neural networks to train a network for predicting parts of speech, chunks, named entity tags, semantic roles and semantically similar words. One crucial aspect of this network is the training of an unsupervised language model, where unlabelled data is leveraged in the following way: the network is asked to decide whether the word in the middle of the sentence is related to the context or not. At the end of training, the embeddings are used as initializations for the features learned by the multi-task network. This semi-supervised multi-task network is able to deliver state of the art results without any hand-crafted features.

Cho and Saul (2010a,b) present a novel idea of deep kernels. The

premise of the work is to view deep architectures as not necessarily compositions of shallow non-linearities, but as a shallow architecture where there the representation (in this case, the kernel) is itself deep. The approach is to construct a kernel that mimics the computation of a deep neural network: these arc-cosine kernel functions induce sparse, non-negative distributed representations that are similar to the ones produced by single-layer threshold networks. By composing these kernels recursively, one can obtain a “deep” representation for the similarity between two inputs. In their experiments with datasets presented in Chapter 4, where they use these deep kernels in conjunction with an SVM, they obtain results that outperform significantly SVMs with RBF kernels and that compare favourably with DBNs.

2.7 Understanding the bias induced by unsupervised pre-training and deep architectures

In this chapter, we have seen that the general idea of a Deep Belief Network—use unsupervised learning to help with learning a deep architecture—has been used in a variety of settings. These strategies differ in the general philosophy: be it by taking a generative or probabilistic modelings perspective (DBNs, DBMs) or by using neural networks for approximating an unsupervised objective (Stacked Auto-Encoders, including the denoising version), or by taking an energy-based approach (SESM, PSD, etc.). What is impressive is that the idea of semi-supervised deep learning can manifest itself in many ways and can have a positive impact, despite the obvious differences in implementation. What follows is a study of the why these general principles that are behind DBNs make such a difference.

In Section 1.4 we argued that in order to make it possible to generalize to unseen examples, one is compelled to have an inductive bias. In Section 2.3.3 we presented evidence and intuition for having an appropriate inductive bias, one that would make it possible for learning algorithms to model complicated functions that are similar to what an intelligent agent would need to model. We posited that deep architectures—compositions of layers of nonlinear features—could be an appropriate inductive bias as there is evidence to support the hypothesis that they can represent more compactly certain classes of functions that are of interest to us.

The mere existence of a class of models (deep architectures) that can *represent* compactly nonlinear input-to-output mappings does not imply a good and efficient technique for learning such mappings. In Sections 2.4, 2.5 and 2.6 we described several of the approaches that researchers in the field use in order to make learning in these predominantly multi-layer neural network models work.

We have seen that the common thread in these approaches is the usage of unsupervised learning as a “pre-training” mechanism for initializing the weights of the deep architecture. These semi-supervised learning approaches can be seen as potentially the good way of overcoming the limitations of shallow architectures, of training deep models and of achieving good generalization performance on complicated problems of interest.

However, deep learning it is still only one of the many inductive biases that are now common in Machine Learning. So it is only natural to ask ourselves what exactly do these biases mean in terms of how they change the class of functions that our models are able to represent. Helping to answer these questions is one of the largest contributions of this thesis.

Tackling this problem head-on is difficult: it is not clear how to, generally speaking, characterize a class of learned functions and show how it is different from any other set of functions. To make it possible, we need to be able to formulate the right kind of questions. One can view these questions as starting points for *explanatory hypotheses*, which will ultimately make it possible to have a better understanding of what deep architectures do. In principle, we would like answers for the following questions:

Why do deep methods work? We have posited hypotheses for the need to have models that are able to represent complicated nonlinear input-output mappings and we gave plenty of examples of successful methods for learning with such models. These methods have a common recipe: we would like to understand why this recipe gives rise to better generalization performance. Notably, **what are the crucial ingredients in the recipe?**

Are there any intrinsic limitations in the used recipe? It would be interesting to characterize the class of functions that is problematic wrt. to the training methods that are commonly used. Having such examples could help us in not only understanding the limitations of deep architectures (and of the commonly used techniques for training them), but would allow improving the training methods in order to overcome these limitations.

How can we analyze and characterize these models? Is there a qualitative way in which we could characterize the functions learned by a given instance of a deep architecture? Can we gain more insight into these functions via such qualitative analyses?

This thesis is thus an exploration of how the choice of doing unsupervised pre-training and of using deep architectures shapes the function classes that we learn:

- First, in Chapters 3 and 4, we explore the empirical effect of unsupervised pre-training and of depth. The central question that this work answers is *when are deep architectures most appropriate?* We come up with datasets that directly test Bengio and LeCun (2007)’s hypotheses related to deep architectures being more appropriate for modelling datasets (or function classes) with many variations.
- Second, in Chapters 5 and 6, we try to understand the mechanisms behind the success of deep architectures in such settings. We posit hypotheses related to how unsupervised pre-training could be shaping the function class that is modelled by deep architectures. Essentially, we investigate the question *why does unsupervised pre-training help deep learning?*
- Finally, in Chapters 7 and 8, we explore the qualitative consequences of choosing these algorithms and models on the functions that are being learned, in an attempt to answer the question *what qualitative invariances are learned by deep architectures?* More concretely, we introduce and present tools for gaining insights into the invariances that are being modelled by deep architectures.

3

Presentation of the first article

3.1 Article details

An Empirical Evaluation of Deep Architectures on Problems with Many Factors of Variation

Hugo Larochelle, Dumitru Erhan, Aaron Courville, James Bergstra and Yoshua Bengio. In the *Proceedings of the 24th International Conference on Machine Learning (ICML 2007)*, pages 473–480. Corvallis, OR, 2007.

Note on my personal contribution: I participated at designing the experiments, datasets and their analysis. I also ran most the experiments with the shallow architectures and I contributed to the writing of the article.

3.2 Context

As of 2007, the field of deep architectures was still in early development. Only a handful of papers (Bengio *et al.*, 2007; Ranzato *et al.*, 2007) had explored the principles laid out by (Hinton *et al.*, 2006), who introduced Deep Belief Networks. The aim of this first article is to empirically investigate certain hypotheses described by Bengio and LeCun (2007). The claims can be summarized as stating that for interesting, real-world problems in which input data has many variations, shallow architectures, especially ones using local kernels as basis elements, are unlikely to scale well. Instead, one would need deep non-local representations that would capture, using compact yet highly non-linear function classes, the complicated variations in the data.

The challenge was to come up with a series of controlled experiments where we can generate a lot of data that has many variations and which would allow us to test hypotheses related to the advantage of deep architectures vs. shallow ones, as well as the advantage of pre-trained architectures vs. ones without pre-training. Thus far, the nascent field

of deep architectures had used only MNIST as a benchmark, and it was not clear how well the conclusions made by Hinton *et al.* (2006), Bengio *et al.* (2007), and Ranzato *et al.* (2007) would generalize to other, perhaps more complicated datasets.

The paper had thus two aims: to test the above mentioned hypotheses on a series of datasets and to construct these datasets in such a way that they would make sure these hypotheses are best testable.

3.3 Contributions

The main contributions of this article are as follows. First, we established a framework for creating data with many controlled factors of variation. These datasets are only a step in the direction of what Bengio and LeCun (2007) meant by the “AI-Set” tasks (namely, tasks that an intelligent agent should be able to solve). Nonetheless, these datasets extend MNIST (and others) in a meaningfully complicated way and our results lend credence to the hypothesis that the inductive bias specified by deep architectures is most appropriate for such data (meaning data with many factors of variation).

A first observation that we make is that meaningful variations in the data can dramatically alter the performance, be that of deep or of shallow architectures. Adding natural image backgrounds to MNIST decreased performance in a *very significant way*, across the board. Another observation is that not all variations are created equal: adding random noise background does not decrease performance as much. The *kind* of noise makes quite a difference: the amount of correlation or structure in the noise is directly responsible for the degradation of the performance of deep architectures.

Pre-training really seems to help, both for shallow and for deep architectures. On some datasets, depth did not have as much of a pronounced effect as pre-training. Perhaps this is the result of the relatively mediocre performance of the first layer (on, say, the natural image background data): it is plausible that a deep model is not necessarily going to perform well if its shallow version is far from being a good model of the data.

Ultimately, pre-training is the key ingredient: in most cases, it is the ingredient for the good performance, but in some cases, we posit, it is at least partly responsible for the bad performance. During unsupervised pre-training, if supervised signal is not strong enough in the data, pre-training could discard it (as most of the unsupervised learning algorithms will try to model the salient features of the data). However, this

is a fundamental loss that we should expect to potentially incur if doing a two-stage process of learning (unsupervised followed by supervised); and this potential is illustrated by the results on mnist-noise-variations.

3.4 Comments

This paper was a direct verification of the hypotheses that deep non-local inductive biases are better than shallow local inductive biases on problems with many factors of variation. The datasets are the paper's most enduring contribution and they have been used in a number of papers as benchmark data (Li, 2010; Marlin *et al.*, 2009; Vincent *et al.*, 2008; Le Roux *et al.*, 2008; Cho and Saul, 2010a,b; Larochelle *et al.*, 2009). An important update to the results is the work of Vincent *et al.* (2008), which introduces Stacked Denoising Auto-Encoders and which shows that they either match or beat the performance of DBNs on the same datasets.

While these datasets are being perceived by certain Machine Learning researchers as tools for testing the “deep hypothesis” *, one can rightly criticize them as being still rather “toyish” in their size and complexity. Advances in computational efficiency should allow us to move on from MNIST-like data to large-scale data and this is definitely an interesting avenue for future work.

*. See <http://hunch.net/?p=1467> for a blog post by John Langford on how boosted decision trees by Li (2010) can do well on these datasets.

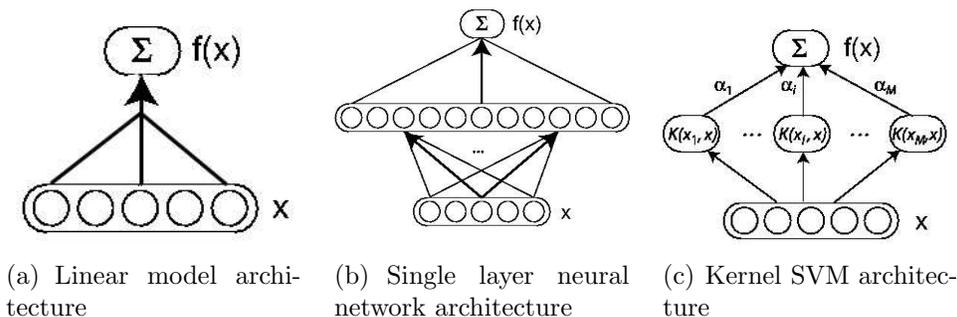
An empirical evaluation of deep architectures on problems with many factors of variation

RECENTLY, several learning algorithms relying on models with deep architectures have been proposed. Though they have demonstrated impressive performance, to date, they have only been evaluated on relatively simple problems such as digit recognition in a controlled environment, for which many machine learning algorithms already report reasonable results. Here, we present a series of experiments which indicate that these models show promise in solving harder learning problems that exhibit many factors of variation. These models are compared with well-established algorithms such as Support Vector Machines and single hidden-layer feed-forward neural networks.

4.1 Introduction

Several recent empirical and theoretical results have brought deep architectures to the attention of the machine learning community: they have been used, with good results, for dimensionality reduction (Hinton and Salakhutdinov, 2006; Salakhutdinov and Hinton, 2007a), and classification of digits from the MNIST data set (Hinton *et al.*, 2006; Bengio *et al.*, 2007). A core contribution of this body of work is the training strategy for a family of computational models that is similar or identical to traditional multilayer perceptrons with sigmoidal hidden units. Traditional gradient-based optimization strategies are not effective when the gradient must be propagated across multiple nonlinearities. Hinton (2006) gives empirical evidence that a sequential, greedy, optimization of the weights of each layer using the generative training criterion of a Restricted Boltzmann Machine tends to initialize the weights such that global gradient-based optimization can work. Bengio *et al.* (2007) showed that this procedure also worked using the autoassociator unsupervised training criterion and empirically studied the sequential, greedy layer-wise strategy. However, to date, the only empirical comparison on classification problems between these deep training algorithms and the state-of-the-art has been on MNIST, on which many algorithms are relatively successful and in which the classes

► **Figure 4.1.** *Examples of models with shallow architectures.*



are known to be well separated in the input space. It remains to be seen whether the advantages seen in the MNIST dataset are observed in other more challenging tasks.

Ultimately, we would like algorithms with the capacity to capture the complex structure found in language and vision tasks. These problems are characterized by many factors of variation that interact in nonlinear ways and make learning difficult. For example, the NORB dataset introduced by LeCun *et al.* (2004) features toys in real scenes, in various lighting, orientation, clutter, and degrees of occlusion. In that work, they demonstrate that existing general algorithms (Gaussian SVMs) perform poorly. In this work, we propose a suite of datasets that spans some of the territory between MNIST and NORB—starting with MNIST, and introducing multiple factors of variation such as rotation and background manipulations. These toy datasets allow us to test the limits of current state-of-the-art algorithms, and explore the behavior of the newer deep-architecture training procedures, *with architectures not tailored to machine vision*. In a very limited but significant way, we believe that these problems are closer to “real world” tasks, and can serve as milestones on the road to AI.

4.1.1 Shallow and Deep Architectures

We define a *shallow model* as a model with very few layers of composition, e.g. linear models, one-hidden-layer neural networks and kernel SVMs (see figure 4.1). On the other hand, *deep architecture models* are such that their output is the result of the composition of some number of computational units, commensurate with the amount of data one can possibly collect, i.e. not exponential in the characteristics of the problem such as the number of factors of variation or the number of inputs. These units are generally organized in layers so that the many levels of computation can be composed.

A function may appear complex from the point of view of a local

non-parametric learning algorithm such as a Gaussian kernel machine, because it has many variations, such as the *sine* function. On the other hand, the Kolmogorov complexity of that function could be small, and it could be representable efficiently with a deep architecture. See Bengio and LeCun (2007) for more discussion on this subject, and pointers to the circuit complexity theory literature showing that shallow circuits can require exponentially more components than deeper circuits.

However, optimizing deep architectures is computationally challenging. It was believed until recently impractical to train deep neural networks (except Convolutional Neural Networks (LeCun *et al.*, 1989)), as iterative optimization procedures tended to get stuck near poor local minima. Fortunately, effective optimization procedures using unsupervised learning have recently been proposed and have demonstrated impressive performance for deep architectures.

4.1.2 Scaling to Harder Learning Problems

Though there are benchmarks to evaluate generic learning algorithms (e.g. the UCI Machine Learning Repository) many of these proposed learning problems do not possess the kind of complexity we address here.

We are interested in problems for which the underlying data distribution can be thought as the product of factor distributions, which means that a sample corresponds to a combination of particular values for these factors. For example, in a digit recognition task, the factors might be the scaling, rotation angle, deviation from the center of the image and the background of the image. Note how some of these factors (such as the background) may be very high-dimensional. In natural language processing, factors which influence the distribution over words in a document include topic, style and various characteristics of the author. In speech recognition, potential factors can be the gender of the speaker, the background noise and the amount of echo in the environment. In these important settings, it is not feasible to collect enough data to cover the input space effectively; especially when these factors vary independently.

Research in incorporating factors of variation into learning procedures has been abundant. A lot of the published results refer to learning invariance in the domain of digit recognition and most of these techniques are engineered for a specific set of invariances. For instance, Decoste and Schölkopf (2002) present a thorough review that discusses the problem of incorporating prior knowledge into the training procedure of kernel-based methods. More specifically, they discuss prior

knowledge about invariances such as translations, rotations etc. Three main methods are described:

1. hand-engineered kernel functions,
2. artificial generation of transformed examples (the so-called *Virtual SV* method),
3. and a combination of the two: engineered kernels that generate artificial examples (e.g. *kernel jittering*).

The main drawback of these methods, from our point of view, is that domain experts are required to explicitly identify the types of invariances that need to be modeled. Furthermore these invariances are highly problem-specific. While there are cases for which manually crafted invariant features are readily available, it is difficult in general to construct invariant features.

We are interested in learning procedures and architectures that would *automatically* discover and represent such invariances (ideally, in an efficient manner). We believe that one good way of achieving such goals is to have procedures that learn high-level features (“abstractions”) that build on lower-level features. One of the main goals of this paper is thus to examine empirically the link between high-level feature extraction and different types of invariances. We start by describing two architectures that are designed for extracting high-level features.

4.2 Learning Algorithms with Deep Architectures

Hinton *et al.* (2006) introduced a greedy layer-wise *unsupervised* learning algorithm for Deep Belief Networks (DBN). This training strategy for such networks was subsequently analyzed by Bengio *et al.* (2007) who concluded that it is an important ingredient in effective optimization and training of deep networks. While lower layers of a DBN extract “low-level features” from the input observation \mathbf{x} , the upper layers are supposed to represent more “abstract” concepts that explain \mathbf{x} .

4.2.1 Deep Belief Networks and Restricted Boltzmann Machines

For classification, a DBN model with ℓ layers models the joint distribution between target y , observed variables x_j and i hidden layers \mathbf{h}^k made of all binary units h_i^k , as follows:

$$P(\mathbf{x}, \mathbf{h}^1, \dots, \mathbf{h}^\ell) = \left(\prod_{k=1}^{\ell-2} P(\mathbf{h}^k | \mathbf{h}^{k+1}) \right) P(y, \mathbf{h}^{\ell-1}, \mathbf{h}^\ell)$$

where $\mathbf{x} = \mathbf{h}^0$, $P(\mathbf{h}^k | \mathbf{h}^{k+1})$ has the form given by equation 4.1 and $P(y, \mathbf{h}^{\ell-1}, \mathbf{h}^\ell)$ is a Restricted Boltzmann Machine (RBM), with the bottom layer being the concatenation of y and $\mathbf{h}^{\ell-1}$ and the top layer is \mathbf{h}^ℓ .

An RBM with n hidden units is a parametric model of the joint distribution between hidden variables h_i and observed variables x_j of the form:

$$P(\mathbf{x}, \mathbf{h}) \propto e^{\mathbf{h}'W\mathbf{x} + b'\mathbf{x} + c'\mathbf{h}}$$

with parameters $\theta = (W, b, c)$. If we restrict h_i and x_j to be binary units, it is straightforward to show that

$$P(\mathbf{x}|\mathbf{h}) = \prod_i P(x_i|\mathbf{h}) = \prod_i \text{sigm}(b_i + \sum_j W_{ji}h_j) \quad (4.1)$$

where *sigm* is the logistic sigmoid function, and $P(\mathbf{h}|\mathbf{x})$ also has a similar form:

$$P(\mathbf{h}|\mathbf{x}) = \prod_j P(h_j|\mathbf{x}) = \prod_j \text{sigm}(c_j + \sum_i W_{ji}x_i) \quad (4.2)$$

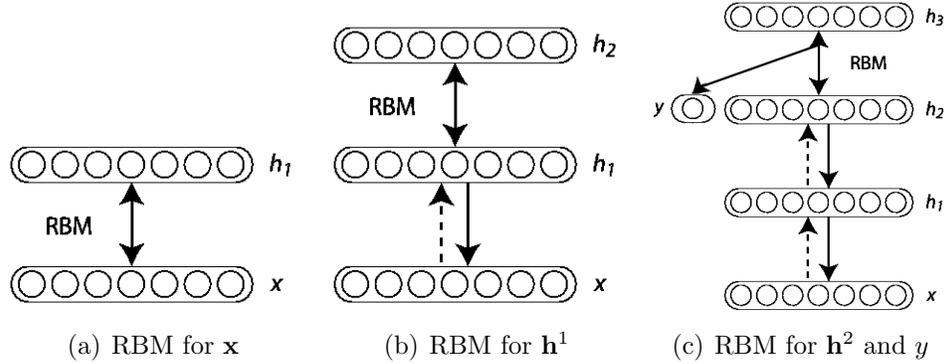
The RBM form can be generalized to other conditional distributions besides the binomial, including continuous variables. See Welling *et al.* (2005) for a generalization of RBM models to conditional distributions from the exponential family.

RBM models can be trained by gradient descent. Although $P(\mathbf{x})$ is not tractable in an RBM, the Contrastive Divergence gradient (Hinton, 2002) is a good stochastic approximation of $\frac{\partial \log P(\mathbf{x})}{\partial \theta}$. The contrastive divergence stochastic gradient can be used to initialize each layer of a DBN as an RBM. The number of layers can be increased greedily, with the newly added top layer trained as an RBM to model the output of the previous layers. When initializing the weights to \mathbf{h}^ℓ , an RBM is trained to model the concatenation of y and $\mathbf{h}^{\ell-1}$. This iterative pre-training procedure is illustrated in figure 4.2.

Using a mean-field approximation of the conditional distribution of layer $\mathbf{h}^{\ell-1}$, we can compute a representation $\hat{\mathbf{h}}^{\ell-1}$ for the input by setting $\hat{\mathbf{h}}^0 = \mathbf{x}$ and iteratively computing $\hat{\mathbf{h}}^k = P(\mathbf{h}^k | \hat{\mathbf{h}}^{k-1})$ using equation 4.2. We then compute the probability of all classes given the approximately inferred value $\hat{\mathbf{h}}^{\ell-1}$ for $\mathbf{h}^{\ell-1}$ using the following expression:

$$P(y|\hat{\mathbf{h}}^{\ell-1}) = \sum_{\mathbf{h}^\ell} P(y, \mathbf{h}^\ell | \hat{\mathbf{h}}^{\ell-1})$$

► **Figure 4.2.** Iterative pre-training construction of a Deep Belief Network.



which can be calculated efficiently. The network can then be fine-tuned according to this estimation of the class probabilities by maximizing the log-likelihood of the class assignments in a training set using standard back-propagation.

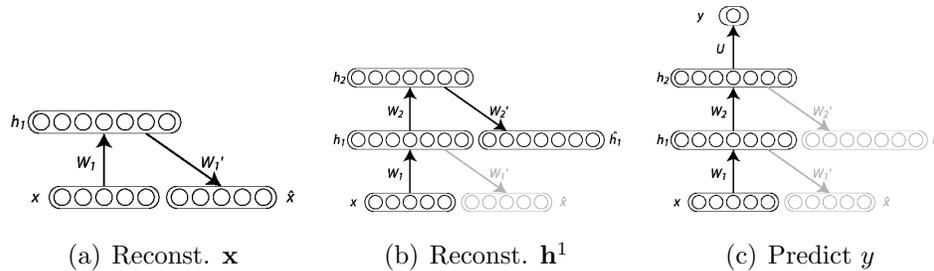
4.2.2 Stacked Autoassociators

As demonstrated by Bengio *et al.* (2007), the idea of successively extracting non-linear features that “explain” variations of the features at the previous level can be applied not only to RBMs but also to *autoassociators*. An autoassociator is simply a model (usually a one-hidden-layer neural network) trained to reproduce its input by forcing the computations to flow through a “bottleneck” representation. Here we used the following architecture for autoassociators. Let \mathbf{x} be the input of the autoassociator, with $x_i \in [0, 1]$, interpreted as the probability for the bit to be 1. For a layer with weight matrix W , hidden biases column vector b and input biases column vector c , the reconstruction probability for bit i is $p_i(\mathbf{x})$, with the vector of probabilities:

$$p(\mathbf{x}) = \text{sigm}(c + W \text{sigm}(b + W'\mathbf{x})).$$

The training criterion for the layer is the average of negative log-likelihoods for predicting \mathbf{x} from $p(\mathbf{x})$. For example, if \mathbf{x} is interpreted either as a sequence of bits or a sequence of bit probabilities, we minimize the reconstruction cross-entropy:

$$R = - \sum_i x_i \log p_i(\mathbf{x}) + (1 - x_i) \log(1 - p_i(\mathbf{x})).$$



◀ **Figure 4.3.** Iterative training construction of the Stacked Autoassociators model.

See Bengio *et al.* (2007) for more details. Once an autoassociator is trained, its internal “bottleneck” representation (here, $\text{sigm}(b + W'\mathbf{x})$) can be used as the input for training a second autoassociator etc. Figure 4.3 illustrates this iterative training procedure. The stacked autoassociators can then be fine-tuned with respect to a supervised training criterion (adding a predictive output layer on top), using back-propagation to compute gradient on parameters of all layers.

4.3 Benchmark Tasks

In order to study the capacity of these algorithms to scale to learning problems with many factors of variation, we have generated datasets where we can identify some of these factors of variation explicitly. We focused on vision problems, mostly because they are easier to generate and analyze. In all cases, the classification problem has a balanced class distribution.

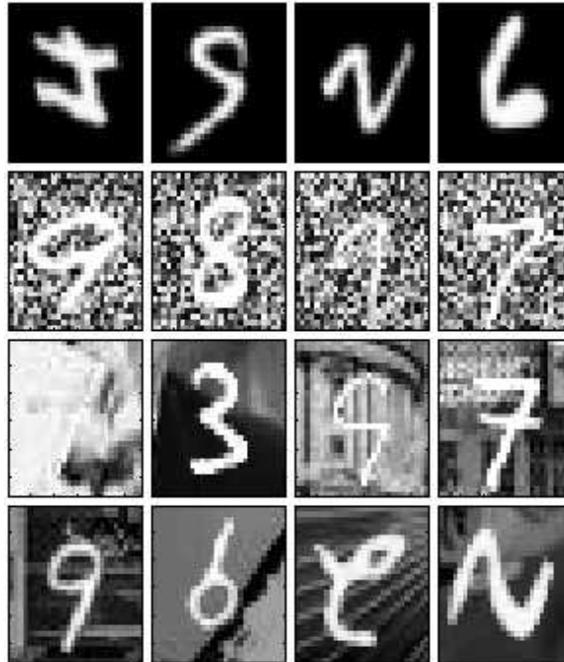
4.3.1 Variations on Digit Recognition

Models with deep architectures have been shown to perform competitively on the MNIST digit recognition dataset (Hinton *et al.*, 2006; Bengio *et al.*, 2007; Salakhutdinov and Hinton, 2007a). In this series of experiments, we construct new datasets by adding additional factors of variation to the MNIST images. The generative process used to generate the datasets is as follows:

1. Pick sample $(x, y) \in \mathcal{X}$ from the digit recognition dataset;
2. Create a perturbed version \hat{x} of x according to some factors of variation;
3. Add (\hat{x}, y) to a new dataset $\hat{\mathcal{X}}$;

4. Go back to 1 until enough samples are generated.

► **Figure 4.4.** From top to bottom, samples from *mnist-rot*, *mnist-back-rand*, *mnist-back-image*, *mnist-rot-back-image*.



Introducing multiple factors of variation leads to the following benchmarks:

mnist-rot: the digits were rotated by an angle generated uniformly between 0 and 2π radians. Thus the factors of variation are the rotation angle and those already contained in MNIST, such as hand writing style;

mnist-back-rand: a random background was inserted in the digit image. Each pixel value of the background was generated uniformly between 0 and 255;

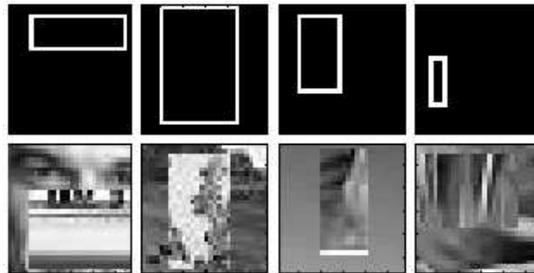
mnist-back-image: a random patch from a black and white image was used as the background for the digit image. The patches were extracted randomly from a set of 20 images downloaded from the internet. Patches which had low pixel variance (i.e. contained little texture) were ignored;

mnist-rot-back-image: the perturbations used in *mnist-rot* and *mnist-back-image* were combined.

These 4 databases have 10000, 2000 and 50000 samples in their training, validation and test sets respectively. Figure 4.4 shows samples from these datasets.

4.3.2 Discrimination between Tall and Wide Rectangles

In this task, a learning algorithm needs to recognize whether a rectangle contained in an image has a larger width or length. The rectangle can be situated anywhere in the 28×28 pixel image. We generated two datasets for this problem:



◀ **Figure 4.5.** From top to bottom, samples from *rectangles* and *rectangles-image*.

rectangles: the pixels corresponding to the border of the rectangle has a value of 255, 0 otherwise. The height and width of the rectangles were sampled uniformly, but when their difference was smaller than 3 pixels the samples were rejected. The top left corner of the rectangles was also sampled uniformly, constrained so that the whole rectangle would fit in the image;

rectangles-image: the border and inside of the rectangles corresponds to an image patch and a background patch is also sampled. The image patches are extracted from one of the 20 images used for *mnist-back-image*. Sampling of the rectangles is essentially the same as for *rectangles*, but the area covered by the rectangles was constrained to be between 25% and 75% of the total image, the length and width of the rectangles were forced to be of at least 10 and their difference was forced to be of at least 5 pixels.

We generated training sets of size 1000 and 10000 and validation sets of size 200 and 2000 for *rectangles* and *rectangles-image* respectively. The test sets were of size 50000 in both cases. Samples for these two tasks are displayed in figure 4.5.

4.3.3 Recognition of Convex Sets

The task of discriminating between tall and wide rectangles was designed to exhibit the learning algorithms' ability to process certain image shapes and learn their properties. Following the same principle, we designed another learning problem which consists in indicating if a set of pixels forms a convex set.

Like the MNIST dataset, the convex and non-convex datasets both consist of images of 28×28 pixels. The convex sets consist of a single convex region with pixels of value 255 (white). Candidate convex images were constructed by taking the intersection of a random number of half-planes whose location and orientation were chosen uniformly at random.

Candidate non-convex images were constructed by taking the union of a random number of convex sets generated as above. The candidate non-convex images were then tested by checking a convexity condition for every pair of pixels in the non-convex set. Those sets that failed the convexity test were added to the dataset. The parameters for generating the convex and non-convex sets were balanced to ensure that the mean number of pixels in the set is the same.

The generated training, validation and test sets are of size 6000, 2000 and 50000 respectively. Samples for this tasks are displayed in figure 4.6.

► **Figure 4.6.** *Samples from convex, where the first, fourth, fifth and last samples correspond to convex white pixel sets.*



4.4 Experiments

We performed experiments on the proposed benchmarks in order to compare the performance of models with deep architectures with other popular generic classification algorithms.

In addition to the Deep Belief Network (denoted DBN-3) and Stacked Autoassociators (denoted SAA-3) models, we conducted experiments with a single hidden-layer DBN (DBN-1), a single hidden-layer neural network (NNet), SVM models with Gaussian (SVM_{rbf}) and polynomial (SVM_{poly}) kernels.

In all cases, model selection was performed using a validation set. For NNet, the best combination of number of hidden units (varying from 25 to 700), learning rate (from 0.0001 to 0.1) and decrease constant (from 0 to 10^{-6}) of stochastic gradient descent and weight decay penalization (from 0 to 10^{-5}) was selected using a grid search.

For DBN-3 and SAA-3, both because of the large number of hyperparameters and because these models can necessitate more than a day

to train, we could not perform a full grid search in the space of hyper-parameters. For both models, the number of hidden units per layer must be chosen, in addition to all other optimization parameters (learning rates for the unsupervised and supervised phases, stopping criteria of the unsupervised phase, etc.). The hyper-parameter search procedure we used alternates between fixing a neural network architecture and searching for good optimization hyper-parameters in a manner similar to coordinate descent. See <http://www.iro.umontreal.ca/~lisa/icml2007> for more details about this procedure. In general, we tested from 50 to 150 different configurations of hyper-parameters for DBN-3 and SAA-3. The layer sizes varied in the intervals [500, 3000], [500, 4000] and [1000, 6000] respectively for the first, second and third layer and the learning rates varied between 0.0001 and 0.1. In the case of the single hidden layer DBN-1 model, we allowed ourselves to test for much larger hidden layer sizes, in order to balance the number of parameters between it and the DBN-3 models we tested.

For all neural networks, we used early stopping based on the classification error of the model on the validation set. However during the initial unsupervised training of DBN-3, the intractability of the RBM training criterion precluded the use of early stopping. Instead, we tested 50 or 100 unsupervised learning epochs for each layer and selected the best choice based on the final accuracy of the model on the validation set.

The experiments with the NNet, DBN-1, DBN-3 and SAA-3 models were conducted using the PLearn* library, an Open Source C++ library for machine learning which was developed and is actively used in our lab.

In the case of SVMs with Gaussian kernels, we performed a two-stage grid search for the width of the kernel and the soft-margin parameter. In the first stage, we searched through a coarse logarithmic grid ranging from $\sigma = 10^{-7}$ to 1 and $C = 0.1$ to 10^5 . In the second stage, we performed a more fine-grained search in the vicinity of that tuple (σ, C) that gave the best validation error. In the case of the polynomial kernel, the strategy was the same, except that we searched through all possible degrees of the polynomial up to 20, rendering the fine-grained search on this parameter useless. Conforming to common practice, we also allowed the SVM models to be retrained on the concatenation of the training and validation set using the selected hyper-parameters. Throughout the experiments we used the publicly available library libSVM (Chang and Lin, 2001), version 2.83.

For all datasets, the input was normalized to have values between 0

*. See <http://www.plearn.org/>

and 1. When the input was binary (i.e. for *rectangles* and *convex*), the Deep Belief Network model used binary input units and when the input was in $[0, 1]^n$ (i.e. for *mnist-rot*, *mnist-back-rand*, *mnist-back-imag*, *mnist-rot-back-image* and *rectangles-image*) it used truncated exponential input units (Bengio *et al.*, 2007).

4.4.1 Benchmark Results

The classification performances for the different learning algorithms on the different datasets of the benchmark are reported in table 4.1. As a reference for the variations on digit recognition experiments, we also include the algorithms' performance on the original MNIST database, with training, validation and test sets of size 10000, 2000 and 50000 respectively. Note that the training set size is significantly smaller than that typically used.

The reader should be aware that new versions of the datasets containing rotations have been generated. There was an issue in the previous versions, on which the original article was based, namely with the way rotated digits were generated, which increased the range of values a digit pixel could have. For instance, this issue made it easier to discern digits from the image background in the *mnist-rot-back-image* dataset. New results for these datasets have been generated and are reported along with the other benchmark results.

There are several conclusions which can be drawn from these results. First, taken together, deep architecture models show globally the best performance. For all datasets, either DBN-1, DBN-3 or SAA-3 are among the best performing models (within the confidence intervals). Five times out of 8 the best accuracy is obtained with a deep architecture model (either DBN-3 or SAA-3). This is especially true in three cases: *mnist-back-rand*, *mnist-back-image*, *mnist-rot-back-image*, where they perform better by a large margin. Also, deep architecture models consistently improve on NNet, which is basically a shallow and totally supervised version of the deep architecture models.

Second, the improvement provided by deep architecture models is most notable for factors of variation related to background, especially in the case of random background, where DBN-3 almost reaches its performance on *mnist-basic*. It seems however that not all of the invariances can be learned just as easily—an example is the one of rotation, where the deep architectures outperform SVMs only by a small margin. \mathbf{SVM}_{rbf} does achieve an impressive result; we believe that this is possible because of the large number of samples in the training set (the input space is well populated) and because there is only one factor applied (contrast this with the score we obtain with \mathbf{SVM}_{rbf} on

Dataset	SVM _{rbf}	SVM _{poly}	NNet	DBN-1	SAA-3	DBN-3
<i>mnist-basic</i>	3.03±0.15	3.69±0.17	4.69±0.19	3.94±0.17	3.46±0.16	3.11±0.15
<i>mnist-rot</i>	10.38±0.27	13.61±0.30	17.62±0.33	12.11±0.29	11.43±0.28	12.30±0.29
<i>mnist-rot</i>	11.11±0.28	15.42±0.32	18.11±0.34	10.30±0.27	10.30±0.27	14.69±0.31
<i>mnist-back-rand</i>	14.58±0.31	16.62±0.33	20.04±0.35	9.80±0.26	11.28±0.28	6.73±0.22
<i>mnist-back-image</i>	22.61±0.37	24.01±0.37	27.41±0.39	16.15±0.32	23.00±0.37	16.31±0.32
<i>mnist-rot-back-image</i>	32.62±0.41	37.59±0.42	42.17±0.43	31.84±0.41	24.09±0.37	28.51±0.40
<i>mnist-rot-back-image</i>	55.18±0.44	56.41±0.43	62.16±0.43	47.39±0.44	51.93±0.44	52.51±0.40
<i>rectangles</i>	2.15±0.13	2.15±0.13	7.16±0.23	4.71±0.19	2.41±0.13	2.60±0.14
<i>rectangles-image</i>	24.04±0.37	24.05±0.37	33.20±0.41	23.69±0.37	24.05±0.37	22.50±0.37
<i>convex</i>	19.13±0.34	19.82±0.35	32.25±0.41	19.92±0.35	18.41±0.34	18.63±0.34

Table 4.1. Results on the benchmark for problems with factors of variation (in percentages). The best performance as well as those with overlapping confidence intervals are marked in bold. The strike-through text represents results with incorrectly generated data (see text for an explanation).

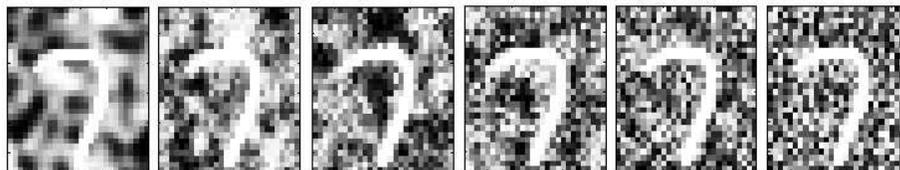
mnist-rot-back-image where the presence of two factors creates a less well-behaved input space)

4.4.2 Impact of Background Pixel Correlation

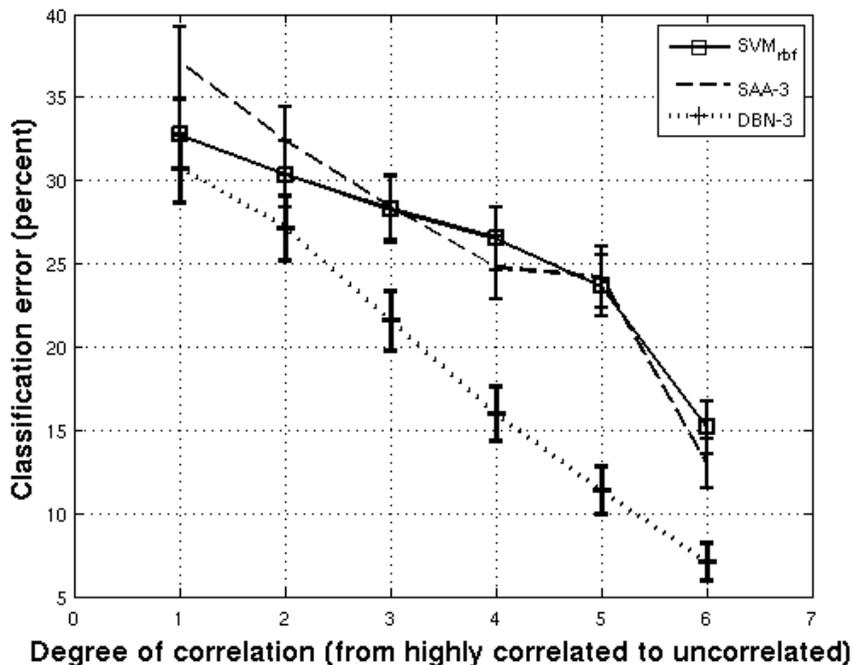
Looking at the results obtained on *mnist-back-rand* and *mnist-back-image* by the different algorithms, it seems that pixel correlation contained in the background images is the key element that worsens the performances. To explore the disparity in performance of the learning algorithms between MNIST with independent noise and MNIST on a background image datasets, we made a series of datasets of MNIST digits superimposed on a background of correlated noisy pixel values.

Correlated pixel noise was sampled from a zero-mean multivariate Gaussian distribution of dimension equal to the number of pixels: $s \sim \mathcal{N}(0, \Sigma)$. The covariance matrix, Σ , is specified by a convex combination of an identity matrix and a Gaussian kernel function (with bandwidth $\sigma = 6$) with mixing coefficient γ . The Gaussian kernel induced a neighborhood correlation structure among pixels such that nearby pixels are more correlated than pixels further apart. For each sample from $\mathcal{N}(0, \Sigma)$, the pixel values p (ranging from 0 to 255) were determined by passing elements of s through the standard error function $p_i = erf(s_i/\sqrt{2})$ and multiplying by 255. We generated six datasets with varying degrees of neighborhood correlation by setting the mixture weight γ to the values $\{0, 0.2, 0.4, 0.6, 0.8, 1\}$. The marginal distributions for each pixel p_i is uniform $[0,1]$ for each value of γ . Figure 4.7

► **Figure 4.7.** From left to right, samples with progressively less pixel correlation in the background.



► **Figure 4.8.** Classification error of SVM_{rbf} , SAA-3 and DBN-3 on MNIST examples with progressively less pixel correlation in the background.



shows some samples from the 6 different tasks.

We ran experiments on these 6 datasets, in order to measure the impact of background pixel correlation on the classification performance. Figure 4.8 shows a comparison of the results obtained by DBN-3, SAA-3 and SVM_{rbf} . In the case of the deep models, we used the same layer sizes for all six experiments. The selected layer sizes had good performance on both *mnist-back-image* and *mnist-back-rand*. However, we did vary the hyper-parameters related to the optimization of the deep networks and chose the best ones for each problems based on the validation set performance. All hyper-parameters of SVM_{rbf} were chosen according to the same procedure.

It can be seen that, as the amount of background pixel correlation increases, the classification performance of all three algorithms degrade. This is coherent with the results obtained on *mnist-back-image* and

mnist-back-rand. This also indicates that, as the factors of variation become more complex in their interaction with the input space, the relative advantage brought by DBN-3 and SAA-3 diminishes. This observation is preoccupying and implies that learning algorithms such as DBN-3 and SAA-3 will eventually need to be adapted in order to scale to harder, potentially “real life” problem.

One might argue that it is unfair to maintain the same layer sizes of the deep architecture models in the previous experiment, as it is likely that the model will need more capacity as the input distribution becomes more complex. This is a valid point, but given that, in the case of DBN-3 we already used a fairly large network (the first, second and third layers had respectively 3000, 2000 and 2000 hidden units), scaling the size of the network to even bigger hidden layers implies serious computational issues. Also, for even more complex datasets such as the NORB dataset (LeCun *et al.*, 2004), which consists in 108×108 stereo images of objects from different categories with many factors of variation such as lighting conditions, elevation, azimuth and background, the size of the deep models becomes too large to even fit in memory. In our preliminary experiments where we subsampled the images to be 54×54 pixels, the biggest models we were able to train only reached 51.6% (DBN-3) and 48.0% (SAA-3), whereas SVM_{rbf} reached 43.6% and NNet reached 43.2%. Hence, a natural next step for learning algorithms for deep architecture models would be to find a way for them to use their capacity to more directly model features of the data that are more predictive of the target value.

Further details of our experiments and links to downloadable versions of the datasets are available online at: <http://www.iro.umontreal.ca/~lisa/icml2007>

4.5 Conclusion and Future Work

We presented a series of experiments which show that deep architecture models tend to outperform other shallow models such as SVMs and single hidden-layer feed-forward neural networks. We also analyzed the relationships between the performance of these learning algorithms and certain properties of the problems that we considered. In particular, we provided empirical evidence that they compare favorably to other state-of-the-art learning algorithms on learning problems with many factors of variation, but only up to a certain point where the data distribution becomes too complex and computational constraints become an important issue.

Acknowledgments

We would like to thank Yann LeCun for suggestions and discussions. We thank the anonymous reviewers who gave useful comments that improved the paper. This work was supported by NSERC, MITACS and the Canada Research Chairs.

5

Presentation of the second article

5.1 Article details

Why Does Unsupervised Pre-training Help Deep Learning? Dumitru Erhan, Yoshua Bengio, Aaron Courville, Pierre-Antoine Manzagol, Pascal Vincent, and Samy Bengio. *Journal of Machine Learning Research*, 11(Feb):625–660, 2010.

Note on my personal contribution: I participated at the design and analysis of all the experiments, ran the vast majority of the experiments, and wrote major parts of the article.

5.2 Context

Since the publication of the seminal work of Hinton *et al.* (2006), the approach of combining unsupervised pre-training with supervised learning for training deep architectures has been gaining traction in the community. Mostly, this was done by employing a two-phase technique or by optimizing the two objectives simultaneously. The popularity of these techniques is evidenced by the review of the recent developments in Section 2.6, where we also show that such models were able to obtain state-of-the-art performance on a variety of Machine Learning problems, ranging from vision to speech and language modelling.

In Chapter 4 we verified empirically some of the assumptions that we had about deep architectures and the effect of unsupervised pre-training. One of the more important conclusions was that pre-training is quite effective in terms of obtaining good performance on a variety of datasets that were constructed to obtain artificial and complicated variations. Another conclusion was that pre-training might hurt performance if there is a lot of irrelevant input structure that can be learned during unsupervised training and if the supervised signal is comparatively not strong enough in the data.

Unsupervised pre-training is clearly a crucial ingredient, one that deserves special analysis in the context of deep architectures. How-

ever, we do not really understand why it does work so well and what mechanisms are behind it. It seems quite interesting and startling that by simply changing the starting point of supervised back-propagation algorithm we get results that are so dramatically different. The motivation of this paper was to explore such issues, by postulating hypotheses related to the effect of unsupervised pre-training.

5.3 Contributions

The contributions of this paper can be summarized as having empirically investigated the main hypotheses related to the effects of unsupervised pre-training. The first part of the investigation tested basic hypotheses related to the overall generalization advantage of deep learning and unsupervised pre-training, in a much larger scale than had been previously done. We observed that increasing the depth of the networks without unsupervised pre-training increases the probability of finding bad local minima when starting from random initialization; unsupervised pre-training is much more robust in such a setting. Moreover, unsupervised pre-training leads to better generalization and allows deeper networks to be trained successfully.

Our visualizations of the networks in function and input spaces point to the fact that networks with unsupervised pre-training explore regions of the parameter space that are different and disjoint from those initialized randomly. These results were an impetus for designing a series of experiments that would test the **role** of unsupervised pre-training. We posited three explanatory hypotheses:

- The **conditioning** hypothesis states that the advantage of pre-training comes from the weights being large and better conditioning the gradient descent process.
- The **optimization** hypothesis suggests that pre-training leads to better generalization because it helps with the *optimization* process.
- The **regularization** hypothesis stipulates that unsupervised pre-training puts the parameters in a region of the space in which training error is not necessarily better than when starting at random, but which systematically yields better generalization and reduces the over-fitting effects; this being the hallmark of a useful regularization effect.

Our experiments do not provide enough evidence to support the conditioning hypothesis. Several of our initial results show that networks initialized with unsupervised pre-training obtain *higher* training

errors compared to the ones initialized randomly. This is consistent with a regularization interpretation; moreover, by adding capacity constraints, pre-trained models' generalization error suffers, adding more evidence to the regularization hypothesis.

Such experiments called for investigations into the effect of unsupervised pre-training for larger-scale datasets. This is a setting in which we discovered a surprising result (apparently, in contradiction with the earlier results): pre-trained networks retain their advantage even when followed by a very large number of supervised updates, in an online learning scenario. On the face of it, this result does not necessarily support either the regularization or the optimization hypothesis. However, further refinements to these experiments show that pre-training acts as a variance reduction technique in online learning. We can and do compute the *training error* in such a setting: pre-trained networks seem to perform better. Online error, training error and generalization error are essentially converging asymptotically to each other in such a scenario, thus what pre-training seems to be doing is better optimization of the generalization error.

On the other hand, we argue that the most of the signs still point to a regularization hypothesis in our case, as we can cast the systematic choice of the initial value for supervised learning (in a non-convex optimization problem) as one way of regularizing learning. This refined interpretation of pre-training as regularization is important as it allows us to learn more about the dynamics of learning in the two-stage training process of a deep architecture. What this teaches us is that, to oversimplify, pre-training “chooses” the hyper-quadrant of the weight space in which the parameters of the network will end up in; stochastic gradient with sigmoidal nonlinearities does not seem to be able to escape it.

Thus, unsupervised pre-training is a clear example of a procedural inductive bias, as described in Section 1.4. Recall that a procedural inductive bias is one that makes learning favour certain hypotheses (that explain the data) to the detriment of others. Unsupervised pre-training it is a procedural bias in that it will make *supervised* learning favour hypotheses which are good *unsupervised* models of the data. Therein lies the strength and perhaps the weakness of using the pre-training approach: as argued in the paper, pre-training will only be useful insofar modelling $P(X)$ is useful for modelling $P(Y|X)$ with the same architecture.

5.4 Comments

This is a paper that has investigated, mainly through empirical means, a variety of hypotheses for why unsupervised pre-training for deep architectures works. Our experiments have shed light on dynamics of learning during the two-stage process of unsupervised learning followed by supervised learning. We have concluded that pre-training acts as a variance reduction technique and has most of the qualities that we associate with a regularizer.

An important side-effect of our results is that online learning could, in principle, suffer from being stuck in a basin of attraction from which it cannot escape. This is an effect of stochastic gradient descent which, as we argued, “overfits” the early examples presented to it. An interesting connection could be made between this effect and the one discussed in Chapter 4. There, we observed that deep architectures learning unsupervised features that are ultimately not going to be useful for supervised learning might not perform well; this was the case with the data where the input structure was rich enough to overwhelm the supervised signal. This inability of stochastic gradient descent to escape the basin of attraction it ends up in could potentially hurt us in the long-run, especially as we try to move to online learning where the input-output distribution is changing over time or is too rich to be captured by the first million examples.

One should note that we attempted to use standard, state-of-the-art and/or common wisdom in training the deep architectures that we considered in our study. While the evidence that we put forward to support the regularization hypothesis is significant, there is a variety of ways in which our results could be extended. This could come from using better inference/sampling/learning schemes for RBMs and DBNs, such as (F)PCD (Tieleman, 2008; Tieleman and Hinton, 2009) or Tempered MCMC (Desjardins *et al.*, 2010), by using nonlinearities that do not suffer from the same drawbacks as standard sigmoids, such as rectified linear units (Nair and Hinton, 2010) or hyperbolic tangents (Bengio and Glorot, 2010), or by using better random initialization strategies for weights (Bengio and Glorot, 2010). Such strategies have made it possible to train networks that are randomly initialized and obtain much better performance on standard benchmarks, compared to previous strategies for training simple feed-forward neural networks. Nonetheless, we believe that the general conclusion of this article is most likely still valid: whichever mechanism for initializing the weights of a neural network is chosen—be that clever(er) randomness (and the associated non-linearity) or pre-training—one can always see the choice of the ini-

tial point of a non-convex optimization procedure as a regularizer. It is also likely that our conclusion about stochastic gradient “overfitting” to early examples holds as well, since stochastic gradient descent is present in all of these mentioned approaches.

Therefore, an important direction of future research could be the exploration of better optimization methods for the supervised objective function. Martens (2010) explored a Hessian-free second order method for optimizing a randomly initialized neural network. Their motivation lies in the fact that a first-order method (such as stochastic gradient descent) is blind to the curvature of the error space and that a second-order method could solve the underfitting problem (the higher training error) that one observes with deep architectures. Their results using deep *auto-encoders* seem to suggest that the Hessian-free method is able to obtain lower training and testing reconstruction errors compared to simply using pre-training. Extending such results into the same supervised learning framework (including classification error results) that we explored in this chapter could certainly provide us with more insights into how to improve learning in deep architectures and whether the regularization hypothesis can be confirmed in the scenario where one uses stronger (higher-order) optimization methods during supervised learning.

6

Why Does Unsupervised Pre-training Help Deep Learning?

MUCH RECENT research has been devoted to learning algorithms for deep architectures such as Deep Belief Networks and stacks of auto-encoder variants, with impressive results obtained in several areas, mostly on vision and language data sets. The best results obtained on supervised learning tasks involve an unsupervised learning component, usually in an unsupervised pre-training phase. Even though these new algorithms have enabled training deep models, many questions remain as to the nature of this difficult learning problem. The main question investigated here is the following: how does unsupervised pre-training work? Answering this questions is important if learning in deep architectures is to be further improved. We propose several explanatory hypotheses and test them through extensive simulations. We empirically show the influence of pre-training with respect to architecture depth, model capacity, and number of training examples. The experiments confirm and clarify the advantage of unsupervised pre-training. The results suggest that unsupervised pre-training guides the learning towards basins of attraction of minima that support better generalization from the training data set; the evidence from these results supports a regularization explanation for the effect of pre-training.

6.1 Introduction

Deep learning methods aim at learning feature hierarchies with features from higher levels of the hierarchy formed by the composition of lower level features. They include learning methods for a wide array of *deep architectures* (Bengio, 2009 provides a survey), including neural networks with many hidden layers (Bengio *et al.*, 2007; Ranzato *et al.*, 2007; Vincent *et al.*, 2008; Collobert and Weston, 2008) and graphical models with many levels of hidden variables (Hinton *et al.*, 2006), among others (Zhu *et al.*, 2009; Weston *et al.*, 2008). Theoretical results (Yao, 1985; Håstad, 1986; Håstad and Goldmann, 1991; Bengio *et al.*, 2006), reviewed and discussed by Bengio and LeCun (2007), suggest that in order to learn the kind of complicated functions that can

represent high-level abstractions (e.g., in vision, language, and other AI-level tasks), one may need *deep architectures*. The recent surge in experimental work in the field seems to support this notion, accumulating evidence that in challenging AI-related tasks—such as computer vision (Bengio *et al.*, 2007; Ranzato *et al.*, 2007; Larochelle *et al.*, 2007; Ranzato *et al.*, 2008; Lee *et al.*, 2009; Mobahi *et al.*, 2009; Osindero and Hinton, 2008), natural language processing (NLP) (Collobert and Weston, 2008; Weston *et al.*, 2008), robotics (Hadsell *et al.*, 2008), or information retrieval (Salakhutdinov and Hinton, 2007b; Salakhutdinov *et al.*, 2007)—deep learning methods significantly out-perform comparable but shallow competitors, and often match or beat the state-of-the-art.

These recent demonstrations of the potential of deep learning algorithms were achieved despite the serious challenge of training models with many layers of adaptive parameters. In virtually all instances of deep learning, the objective function is a highly non-convex function of the parameters, with the potential for many distinct *local minima* in the model parameter space. The principal difficulty is that not all of these minima provide equivalent generalization errors and, we suggest, that for deep architectures, the standard training schemes (based on random initialization) tend to place the parameters in regions of the parameters space that generalize poorly—as was frequently observed empirically but rarely reported (Bengio and LeCun, 2007).

The breakthrough to effective training strategies for deep architectures came in 2006 with the algorithms for training deep belief networks (DBN) (Hinton *et al.*, 2006) and stacked auto-encoders (Ranzato *et al.*, 2007; Bengio *et al.*, 2007), which are all based on a similar approach: greedy layer-wise unsupervised pre-training followed by supervised fine-tuning. Each layer is pre-trained with an unsupervised learning algorithm, learning a nonlinear transformation of its input (the output of the previous layer) that captures the main variations in its input. This unsupervised pre-training sets the stage for a final training phase where the deep architecture is fine-tuned with respect to a supervised training criterion with gradient-based optimization. While the improvement in performance of trained deep models offered by the pre-training strategy is impressive, little is understood about the mechanisms underlying this success.

The objective of this paper is to explore, through extensive experimentation, how unsupervised pre-training works to render learning deep architectures more effective and why they appear to work so much better than traditional neural network training methods. There are a few reasonable hypotheses why unsupervised pre-training might work.

One possibility is that unsupervised pre-training acts as a kind of network pre-conditioner, putting the parameter values in the appropriate range for further supervised training. Another possibility, suggested by Bengio *et al.* (2007), is that unsupervised pre-training initializes the model to a point in parameter space that somehow renders the optimization process more effective, in the sense of achieving a lower minimum of the empirical cost function.

Here, we argue that our experiments support a view of unsupervised pre-training as an unusual form of *regularization*: minimizing variance and introducing bias towards configurations of the parameter space that are useful for unsupervised learning. This perspective places unsupervised pre-training well within the family of recently developed semi-supervised methods. The unsupervised pre-training approach is, however, unique among semi-supervised training strategies in that it acts by defining a particular initialization point for standard supervised training rather than either modifying the supervised objective function (Barron, 1991) or explicitly imposing constraints on the parameters throughout training (Lasserre *et al.*, 2006). This type of initialization-as-regularization strategy has precedence in the neural networks literature, in the shape of the early stopping idea (Sjöberg and Ljung, 1995; Amari *et al.*, 1997), and in the Hidden Markov Models (HMM) community (Bahl *et al.*, 1986; Povey and Woodland, 2002) where it was found that first training an HMM as a generative model was essential (as an initialization step) before fine-tuning it discriminatively. We suggest that, in the highly non-convex situation of training a deep architecture, defining a particular initialization point *implicitly* imposes constraints on the parameters in that it specifies which minima (out of a very large number of possible minima) of the cost function are allowed. In this way, it may be possible to think of unsupervised pre-training as being related to the approach of Lasserre *et al.* (2006).

Another important and distinct property of the unsupervised pre-training strategy is that in the standard situation of training using stochastic gradient descent, the beneficial generalization effects due to pre-training do not appear to diminish as the number of labeled examples grows very large. We argue that this is a consequence of the combination of the non-convexity (multi-modality) of the objective function and the dependency of the stochastic gradient descent method on example ordering. We find that early changes in the parameters have a greater impact on the final region (basin of attraction of the descent procedure) in which the learner ends up. In particular, unsupervised pre-training sets the parameter in a region from which better basins of attraction can be reached, in terms of generalization. Hence, although

unsupervised pre-training is a regularizer, it can have a positive effect on the training objective when the number of training examples is large.

As previously stated, this paper is concerned with an experimental assessment of the various competing hypotheses regarding the role of unsupervised pre-training in the recent success of deep learning methods. To this end, we present a series of experiments design to pit these hypotheses against one another in an attempt to resolve some of the mystery surrounding the effectiveness of unsupervised pre-training.

In the first set of experiments (in Section 6.6), we establish the effect of unsupervised pre-training on improving the generalization error of trained deep architectures. In this section we also exploit dimensionality reduction techniques to illustrate how unsupervised pre-training affects the location of minima in parameter space.

In the second set of experiments (in Section 6.7), we directly compare the two alternative hypotheses (pre-training as a pre-conditioner; and pre-training as an optimization scheme) against the hypothesis that unsupervised pre-training is a regularization strategy. In the final set of experiments, (in Section 6.8), we explore the role of unsupervised pre-training in the online learning setting, where the number of available training examples grows very large. In these experiments, we test key aspects of our hypothesis relating to the topology of the cost function and the role of unsupervised pre-training in manipulating the region of parameter space from which supervised training is initiated.

Before delving into the experiments, we begin with a more in-depth view of the challenges in training deep architectures and how we believe unsupervised pre-training works towards overcoming these challenges.

6.2 The Challenges of Deep Learning

In this section, we present a perspective on why standard training of deep models through gradient backpropagation appears to be so difficult. First, it is important to establish what we mean in stating that training is difficult.

We believe the central challenge in training deep architectures is dealing with the strong dependencies that exist during training between the parameters across layers. One way to conceive the difficulty of the problem is that we must simultaneously:

1. adapt the lower layers in order to provide adequate input to the final (end of training) setting of the upper layers

2. adapt the upper layers to make good use of the final (end of training) setting of the lower layers.

The second problem is easy on its own (i.e., when the final setting of the other layers is known). It is not clear how difficult is the first one, and we conjecture that a particular difficulty arises when both sets of layers must be learned jointly, as the gradient of the objective function is limited to a local measure given the current setting of other parameters. Furthermore, because with enough capacity the top two layers can easily overfit the training set, training error does not necessarily reveal the difficulty in optimizing the lower layers. As shown in our experiments here, the standard training schemes tend to place the parameters in regions of the parameters space that generalize poorly.

A separate but related issue appears if we focus our consideration of traditional training methods for deep architectures on stochastic gradient descent. A sequence of examples along with an online gradient descent procedure defines a trajectory in parameter space, which converges in some sense (the error does not improve anymore, maybe because we are near a local minimum). The hypothesis is that small perturbations of that trajectory (either by initialization or by changes in which examples are seen when) have more effect early on. Early in the process of following the stochastic gradient, changes in the weights tend to increase their magnitude and, consequently, the amount of non-linearity of the network increases. As this happens, the set of regions accessible by stochastic gradient descent on samples of the training distribution becomes smaller. Early on in training small perturbations allow the model parameters to switch from one basin to a nearby one, whereas later on (typically with larger parameter values), it is unlikely to “escape” from such a basin of attraction. Hence the early examples can have a larger influence and, in practice, trap the model parameters in particular regions of parameter space that correspond to the specific and arbitrary ordering of the training examples.* An important consequence of this phenomenon is that even in the presence of a very large (effectively infinite) amounts of supervised data, stochastic gradient descent is subject to a degree of *overfitting* to the training data presented early in the training process. In that sense, unsupervised pre-training interacts intimately with the optimization process, and when the number of training examples becomes large, its positive effect is seen not only on generalization error but also on training error.

*. This process seems similar to the “critical period” phenomena observed in neuroscience and psychology (Bornstein, 1987).

6.3 Unsupervised Pre-training Acts as a Regularizer

As stated in the introduction, we believe that greedy layer-wise unsupervised pre-training overcomes the challenges of deep learning by introducing a useful prior to the *supervised fine-tuning* training procedure. We claim that the regularization effect is a consequence of the pre-training procedure establishing an initialization point of the fine-tuning procedure inside a region of parameter space in which the parameters are henceforth restricted. The parameters are restricted to a relatively small volume of parameter space that is delineated by the boundary of the *local basin of attraction* of the supervised fine-tuning cost function.

The pre-training procedure increases the magnitude of the weights and in standard deep models, with a sigmoidal nonlinearity, this has the effect of rendering both the function more nonlinear and the cost function locally more complicated with more topological features such as peaks, troughs and plateaus. The existence of these topological features renders the parameter space locally more difficult to travel significant distances via a gradient descent procedure. This is the core of the restrictive property imposed by the pre-training procedure and hence the basis of its regularizing properties.

But unsupervised pre-training restricts the parameters to particular regions: those that correspond to capturing structure in the input distribution $P(X)$. To simply state that unsupervised pre-training is a regularization strategy somewhat undermines the significance of its effectiveness. Not all regularizers are created equal and, in comparison to standard regularization schemes such as L_1 and L_2 parameter penalization, unsupervised pre-training is dramatically effective. We believe the credit for its success can be attributed to the unsupervised training criteria optimized during unsupervised pre-training.

During each phase of the greedy unsupervised training strategy, layers are trained to represent the dominant factors of variation extant in the data. This has the effect of leveraging knowledge of X to form, at each layer, a representation of X consisting of statistically reliable features of X that can then be used to predict the output (usually a class label) Y . This perspective places unsupervised pre-training well within the family of learning strategies collectively known as semi-supervised methods. As with other recent work demonstrating the effectiveness of semi-supervised methods in regularizing model parameters, we claim that the effectiveness of the unsupervised pre-training strategy is limited to the extent that learning $P(X)$ is helpful in learning $P(Y|X)$.

Here, we find transformations of X —learned features—that are predictive of the main factors of variation in $P(X)$, and when the pre-training strategy is effective,^{*} some of these learned features of X are also predictive of Y . In the context of deep learning, the greedy unsupervised strategy may also have a special function. To some degree it resolves the problem of simultaneously learning the parameters at all layers (mentioned in Section 6.2) by introducing a proxy criterion. This proxy criterion encourages significant factors of variation, present in the input data, to be represented in intermediate layers.

To clarify this line of reasoning, we can formalize the effect of unsupervised pre-training in inducing a prior distribution over the parameters. Let us assume that parameters are forced to be chosen in a bounded region $\mathcal{S} \subset \mathbb{R}^d$. Let \mathcal{S} be split in regions $\{R_k\}$ that are the basins of attraction of descent procedures in the training error (note that $\{R_k\}$ depends on the training set, but the dependency decreases as the number of examples increases). We have $\cup_k R_k = \mathcal{S}$ and $R_i \cap R_j = \emptyset$ for $i \neq j$. Let $v_k = \int 1_{\theta \in R_k} d\theta$ be the volume associated with region R_k (where θ are our model’s parameters). Let r_k be the probability that a purely random initialization (according to our initialization procedure, which factorizes across parameters) lands in R_k , and let π_k be the probability that pre-training (following a random initialization) lands in R_k , that is, $\sum_k r_k = \sum_k \pi_k = 1$. We can now take into account the initialization procedure as a regularization term:

$$\text{regularizer} = -\log P(\theta).$$

For pre-trained models, the prior is

$$P_{\text{pre-training}}(\theta) = \sum_k 1_{\theta \in R_k} \pi_k / v_k.$$

For the models without unsupervised pre-training, the prior is

$$P_{\text{no-pre-training}}(\theta) = \sum_k 1_{\theta \in R_k} r_k / v_k.$$

One can verify that $P_{\text{pre-training}}(\theta \in R_k) = \pi_k$ and $P_{\text{no-pre-training}}(\theta \in R_k) = r_k$. When π_k is tiny, the penalty is high when $\theta \in R_k$, with unsupervised pre-training. The derivative of this regularizer is zero almost everywhere because we have chosen a uniform prior inside each region R_k . Hence, to take the regularizer into account, and having a generative model $P_{\text{pre-training}}(\theta)$ for θ (i.e., this is the unsupervised

*. Acting as a form of (data-dependent) “prior” on the parameters, as we are about to formalize.

pre-training procedure), it is reasonable to sample an initial θ from it (knowing that from this point on the penalty will not increase during the iterative minimization of the training criterion), and this is exactly how the pre-trained models are obtained in our experiments.

Note that this formalization is just an illustration: it is there to simply show how one could conceptually think of an initialization point as a regularizer and should not be taken as a literal interpretation of how regularization is explicitly achieved, since we do not have an analytic formula for computing the π_k 's and v_k 's. Instead these are implicitly defined by the whole unsupervised pre-training procedure.

6.4 Previous Relevant Work

We start with an overview of the literature on semi-supervised learning (SSL), since the SSL framework is essentially the one in which we operate as well.

6.4.1 Related Semi-Supervised Methods

It has been recognized for some time that generative models are less prone to overfitting than discriminant ones (Ng and Jordan, 2002). Consider input variable X and target variable Y . Whereas a discriminant model focuses on $P(Y|X)$, a generative model focuses on $P(X, Y)$ (often parametrized as $P(X|Y)P(Y)$), that is, it also cares about getting $P(X)$ right, which can reduce the freedom of fitting the data when the ultimate goal is only to predict Y given X .

Exploiting information about $P(X)$ to improve generalization of a classifier has been the driving idea behind semi-supervised learning (Chapelle *et al.*, 2006). For example, one can use unsupervised learning to map X into a representation (also called embedding) such that two examples \mathbf{x}_1 and \mathbf{x}_2 that belong to the same cluster (or are reachable through a short path going through neighboring examples in the training set) end up having nearby embeddings. One can then use supervised learning (e.g., a linear classifier) in that new space and achieve better generalization in many cases (Belkin and Niyogi, 2002; Chapelle *et al.*, 2003). A long-standing variant of this approach is the application of Principal Components Analysis as a pre-processing step before applying a classifier (on the projected data). In these models the data is first transformed in a new representation using unsupervised learning, and a supervised classifier is stacked on top, learning to map the data in this new representation into class predictions.

Instead of having separate unsupervised and supervised components in the model, one can consider models in which $P(X)$ (or $P(X, Y)$) and $P(Y|X)$ share parameters (or whose parameters are connected in some way), and one can trade-off the supervised criterion $-\log P(Y|X)$ with the unsupervised or generative one ($-\log P(X)$ or $-\log P(X, Y)$). It can then be seen that the generative criterion corresponds to a particular form of prior (Lasserre *et al.*, 2006), namely that the structure of $P(X)$ is connected to the structure of $P(Y|X)$ in a way that is captured by the shared parametrization. By controlling how much of the generative criterion is included in the total criterion, one can find a better trade-off than with a purely generative or a purely discriminative training criterion (Lasserre *et al.*, 2006; Larochelle and Bengio, 2008).

In the context of deep architectures, a very interesting application of these ideas involves adding an unsupervised embedding criterion at each layer (or only one intermediate layer) to a traditional supervised criterion (Weston *et al.*, 2008). This has been shown to be a powerful semi-supervised learning strategy, and is an alternative to the kind of algorithms described and evaluated in this paper, which also combine unsupervised learning with supervised learning.

In the context of scarcity of labelled data (and abundance of unlabelled data), deep architectures have shown promise as well. Salakhutdinov and Hinton (2008) describe a method for learning the covariance matrix of a Gaussian Process, in which the usage of unlabelled examples for modeling $P(X)$ improves $P(Y|X)$ quite significantly. Note that such a result is to be expected: with few labelled samples, modeling $P(X)$ usually helps. Our results show that even in the context of *abundant labelled data*, unsupervised pre-training still has a pronounced positive effect on generalization: a somewhat surprising conclusion.

6.4.2 Early Stopping as a Form of Regularization

We stated that pre-training as initialization can be seen as restricting the optimization procedure to a relatively small volume of parameter space that corresponds to a local basin of attraction of the supervised cost function. Early stopping can be seen as having a similar effect, by constraining the optimization procedure to a region of the parameter space that is close to the initial configuration of parameters. With τ the number of training iterations and η the learning rate used in the update procedure, $\tau\eta$ can be seen as the reciprocal of a regularization parameter. Indeed, restricting either quantity restricts the area of parameter space reachable from the starting point. In the case of the optimization of a simple linear model (initialized at the origin) using

a quadratic error function and simple gradient descent, early stopping will have a similar effect to traditional regularization.

Thus, in both pre-training and early stopping, the parameters of the supervised cost function are constrained to be close to their initial values.* A more formal treatment of early stopping as regularization is given by Sjöberg and Ljung (1995) and Amari *et al.* (1997). There is no equivalent treatment of pre-training, but this paper sheds some light on the effects of such initialization in the case of deep architectures.

6.5 Experimental Setup and Methodology

In this section, we describe the setting in which we test the hypothesis introduced in Section 6.3 and previously proposed hypotheses. The section includes a description of the deep architectures used, the data sets and the details necessary to reproduce our results.

6.5.1 Models

All of the successful methods (Hinton *et al.*, 2006; Hinton and Salakhutdinov, 2006; Bengio *et al.*, 2007; Ranzato *et al.*, 2007; Vincent *et al.*, 2008; Weston *et al.*, 2008; Ranzato *et al.*, 2008; Lee *et al.*, 2008) in the literature for training deep architectures have something in common: they rely on an unsupervised learning algorithm that provides a training signal at the level of a single layer. Most work in two main phases. In a first phase, *unsupervised pre-training*, all layers are initialized using this layer-wise unsupervised learning signal. In a second phase, *fine-tuning*, a global training criterion (a prediction error, using labels in the case of a supervised task) is minimized. In the algorithms initially proposed (Hinton *et al.*, 2006; Bengio *et al.*, 2007; Ranzato *et al.*, 2007), the unsupervised pre-training is done in a greedy layer-wise fashion: at stage k , the k -th layer is trained (with respect to an unsupervised criterion) using as input the output of the previous layer, and while the previous layers are kept fixed.

We shall consider two deep architectures as representatives of two families of models encountered in the deep learning literature.

*. In the case of pre-training the “initial values” of the parameters for the supervised phase are those that were obtained at the end of pre-training.

6.5.2 Deep Belief Networks

The first model is the Deep Belief Net (DBN) by Hinton *et al.* (2006), obtained by training and stacking several layers of Restricted Boltzmann Machines (RBM) in a greedy manner. Once this stack of RBMs is trained, it can be used to initialize a multi-layer neural network for classification.

An RBM with n hidden units is a Markov Random Field (MRF) for the joint distribution between hidden variables h_i and observed variables x_j such that $P(\mathbf{h}|\mathbf{x})$ and $P(\mathbf{x}|\mathbf{h})$ factorize, that is, $P(\mathbf{h}|\mathbf{x}) = \prod_i P(h_i|\mathbf{x})$ and $P(\mathbf{x}|\mathbf{h}) = \prod_j P(x_j|\mathbf{h})$. The sufficient statistics of the MRF are typically h_i , x_j and $h_i x_j$, which gives rise to the following joint distribution:

$$P(\mathbf{x}, \mathbf{h}) \propto e^{\mathbf{h}'W\mathbf{x} + b'\mathbf{x} + c'\mathbf{h}}$$

with corresponding parameters $\theta = (W, b, c)$ (with $'$ denoting transpose, c_i associated with h_i , b_j with x_j , and W_{ij} with $h_i x_j$). If we restrict h_i and x_j to be binary units, it is straightforward to show that

$$\begin{aligned} P(\mathbf{x}|\mathbf{h}) &= \prod_j P(x_j|\mathbf{h}) \quad \text{with} \\ P(x_j = 1|\mathbf{h}) &= \text{sigmoid}(b_j + \sum_i W_{ij} h_i). \end{aligned}$$

where $\text{sigmoid}(\mathbf{a}) = 1/(1 + \exp(-\mathbf{a}))$ (applied element-wise on a vector \mathbf{a}), and $P(\mathbf{h}|\mathbf{x})$ also has a similar form:

$$\begin{aligned} P(\mathbf{h}|\mathbf{x}) &= \prod_i P(h_i|\mathbf{x}) \quad \text{with} \\ P(h_i = 1|\mathbf{x}) &= \text{sigmoid}(c_i + \sum_j W_{ij} x_j). \end{aligned}$$

The RBM form can be generalized to other conditional distributions besides the binomial, including continuous variables. Welling *et al.* (2005) describe a generalization of RBM models to conditional distributions from the exponential family.

RBM models can be trained by approximate stochastic gradient descent. Although $P(\mathbf{x})$ is not tractable in an RBM, the Contrastive Divergence estimator (Hinton, 2002) is a good stochastic approximation of $\frac{\partial \log P(\mathbf{x})}{\partial \theta}$, in that it very often has the same sign (Bengio and Delalleau, 2009).

A DBN is a multi-layer generative model with layer variables h_0 (the input or visible layer), h_1 , h_2 , etc. The top two layers have a joint distribution which is an RBM, and $P(h_k|h_{k+1})$ are parametrized in the

same way as for an RBM. Hence a 2-layer DBN is an RBM, and a stack of RBMs share parametrization with a corresponding DBN. The contrastive divergence update direction can be used to initialize each layer of a DBN as an RBM, as follows. Consider the first layer of the DBN trained as an RBM P_1 with hidden layer h_1 and visible layer v_1 . We can train a second RBM P_2 that models (in its visible layer) the samples h_1 from $P_1(h_1|v_1)$ when v_1 is sampled from the training data set. It can be shown that this maximizes a lower bound on the log-likelihood of the DBN. The number of layers can be increased greedily, with the newly added top layer trained as an RBM to model the samples produced by chaining the posteriors $P(h_k|h_{k-1})$ of the lower layers (starting from h_0 from the training data set).

The parameters of a DBN or of a stack of RBMs also correspond to the parameters of a deterministic feed-forward multi-layer neural network. The i -th unit of the k -th layer of the neural network outputs $\hat{h}_{ki} = \text{sigmoid}(c_{ki} + \sum_j W_{kij} \hat{h}_{k-1,j})$, using the parameters c_k and W_k of the k -th layer of the DBN. Hence, once the stack of RBMs or the DBN is trained, one can use those parameters to initialize the first layers of a corresponding multi-layer neural network. One or more additional layers can be added to map the top-level features \hat{h}_k to the predictions associated with a target variable (here the probabilities associated with each class in a classification task). Bengio (2009) provides more details on RBMs and DBNs, and a survey of related models and deep architectures.

6.5.3 Stacked Denoising Auto-Encoders

The second model, by Vincent *et al.* (2008), is the so-called Stacked Denoising Auto-Encoder (SDAE). It borrows the greedy principle from DBNs, but uses denoising auto-encoders as a building block for unsupervised modeling. An auto-encoder learns an encoder $h(\cdot)$ and a decoder $g(\cdot)$ whose composition approaches the identity for examples in the training set, that is, $g(h(\mathbf{x})) \approx \mathbf{x}$ for \mathbf{x} in the training set.

Assuming that some constraint prevents $g(h(\cdot))$ from being the identity for arbitrary arguments, the auto-encoder has to capture statistical structure in the training set in order to minimize reconstruction error. However, with a high capacity code ($h(\mathbf{x})$ has too many dimensions), a regular auto-encoder could potentially learn a trivial encoding. Note that there is an intimate connection between minimizing reconstruction error for auto-encoders and contrastive divergence training for RBMs, as both can be shown to approximate a log-likelihood gradient (Bengio and Delalleau, 2009).

The *denoising auto-encoder* (Vincent *et al.*, 2008; Seung, 1998; LeCun, 1987; Gallinari *et al.*, 1987) is a stochastic variant of the ordinary auto-encoder with the distinctive property that even with a high capacity model, it cannot learn the identity mapping. A denoising auto-encoder is explicitly trained to denoise a corrupted version of its input. Its training criterion can also be viewed as a variational lower bound on the likelihood of a specific generative model. It has been shown on an array of data sets to perform significantly better than ordinary auto-encoders and similarly or better than RBMs when stacked into a deep supervised architecture (Vincent *et al.*, 2008). Another way to prevent regular auto-encoders with more code units than inputs to learn the identity is to restrict the capacity of the representation by imposing sparsity on the code (Ranzato *et al.*, 2007, 2008).

We now summarize the training algorithm of the Stacked Denoising Auto-Encoders. More details are given by Vincent *et al.* (2008). Each denoising auto-encoder operates on its inputs \mathbf{x} , either the raw inputs or the outputs of the previous layer. The denoising auto-encoder is trained to reconstruct \mathbf{x} from a stochastically corrupted (noisy) transformation of it. The output of each denoising auto-encoder is the “code vector” $h(\mathbf{x})$, not to confuse with the reconstruction obtained by applying the decoder to that code vector. In our experiments $h(\mathbf{x}) = \text{sigmoid}(\mathbf{b} + W\mathbf{x})$ is an ordinary neural network layer, with hidden unit biases \mathbf{b} , and weight matrix W . Let $C(\mathbf{x})$ represent a stochastic corruption of \mathbf{x} . As done by Vincent *et al.* (2008), we set $C_i(\mathbf{x}) = x_i$ or 0, with a random subset (of a fixed size) selected for zeroing. We have also considered a salt and pepper noise, where we select a random subset of a fixed size and set $C_i(\mathbf{x}) = \text{Bernoulli}(0.5)$. The denoised “reconstruction” is obtained from the noisy input with $\hat{\mathbf{x}} = \text{sigmoid}(\mathbf{c} + W^T h(C(\mathbf{x})))$, using biases \mathbf{c} and the transpose of the feed-forward weights W . In the experiments on images, both the raw input x_i and its reconstruction \hat{x}_i for a particular pixel i can be interpreted as a Bernoulli probability for that pixel: the probability of painting the pixel as black at that location. We denote $\text{CE}(\mathbf{x}|\hat{\mathbf{x}}) = \sum_i \text{CE}(x_i|\hat{x}_i)$ the sum of the component-wise cross-entropy between the Bernoulli probability distributions associated with each element of \mathbf{x} and its reconstruction probabilities $\hat{\mathbf{x}}$: $\text{CE}(\mathbf{x}|\hat{\mathbf{x}}) = -\sum_i (x_i \log \hat{x}_i + (1 - x_i) \log (1 - \hat{x}_i))$. The Bernoulli model only makes sense when the input components and their reconstruction are in $[0, 1]$; another option is to use a Gaussian model, which corresponds to a Mean Squared Error (MSE) criterion.

With either DBN or SDAE, an output logistic regression layer is added after unsupervised training. This layer uses softmax (multinomial logistic regression) units to estimate $P(\text{class}|\mathbf{x}) = \text{softmax}_{\text{class}}(\mathbf{a})$,

where a_i is a linear combination of outputs from the top hidden layer. The whole network is then trained as usual for multi-layer perceptrons, to minimize the output (negative log-likelihood) prediction error.

6.5.4 Data Sets

We experimented on three data sets, with the motivation that our experiments would help understand previously presented results with deep architectures, which were mostly with the MNIST data set and variations (Hinton *et al.*, 2006; Bengio *et al.*, 2007; Ranzato *et al.*, 2007; Larochelle *et al.*, 2007; Vincent *et al.*, 2008):

MNIST the digit classification data set by LeCun *et al.* (1998), containing 60,000 training and 10,000 testing examples of 28x28 handwritten digits in gray-scale.

InfiniteMNIST a data set by Loosli *et al.* (2007), which is an extension of MNIST from which one can obtain a quasi-infinite number of examples. The samples are obtained by performing random elastic deformations of the original MNIST digits. In this data set, there is only one set of examples, and the models will be compared by their (online) performance on it.

Shaperset is a synthetic data set with a controlled range of geometric invariances. The underlying task is binary classification of 10×10 images of triangles and squares. The examples show images of shapes with many variations, such as size, orientation and gray-level. The data set is composed of 50000 training, 10000 validation and 10000 test images.*

6.5.5 Setup

The models used are

1. Deep Belief Networks containing Bernoulli RBM layers,
2. Stacked Denoising Auto-Encoders with Bernoulli input units, and
3. standard feed-forward multi-layer neural networks,

each with 1–5 hidden layers. Each hidden layer contains the same number of hidden units, which is a hyperparameter. The other hyperparameters are the unsupervised and supervised learning rates, the L_2

*. The data set can be downloaded from <http://www.iro.umontreal.ca/~lisa/twiki/bin/view.cgi/Public/ShapersetDataForJMLR>.

penalty / weight decay,^{*} and the fraction of stochastically corrupted inputs (for the SDAE). For MNIST, the number of supervised and unsupervised passes through the data (epochs) is 50 and 50 per layer, respectively. With InfiniteMNIST, we perform 2.5 million unsupervised updates followed by 7.5 million supervised updates.[†] The standard feed-forward networks are trained using 10 million supervised updates. For MNIST, model selection is done by choosing the hyperparameters that optimize the supervised (classification) error on the validation set. For InfiniteMNIST, we use the average online error over the last million examples for hyperparameter selection. In all cases, purely stochastic gradient updates are applied.

The experiments involve the training of deep architectures with a variable number of layers with and without unsupervised pre-training. For a given layer, weights are initialized using random samples from uniform $[-1/\sqrt{k}, 1/\sqrt{k}]$, where k is the number of connections that a unit receives from the previous layer (the fan-in). Either supervised gradient descent or unsupervised pre-training follows.

In most cases (for MNIST), we first launched a number of experiments using a cross-product of hyperparameter values[‡] applied to 10 different random initialization seeds. We then selected the hyperparameter sets giving the best validation error for each combination of model (with or without pre-training), number of layers, and number of training iterations. Using these hyper-parameters, we launched experiments using an additional 400 initialization seeds. For InfiniteMNIST, only one seed is considered (an arbitrarily chosen value).

In the discussions below we sometimes use the word **apparent local minimum** to mean the solution obtained after training, when no further noticeable progress seems achievable by stochastic gradient descent. It is possible that these are not really near a true local minimum (there could be a tiny ravine towards significant improvement, not accessible by gradient descent), but it is clear that these end-points represent regions where gradient descent is stuck. Note also that when we write of number of layers it is to be understood as the number of *hidden* layers in the network.

*. A penalizing term $\lambda\|\theta\|_2^2$ is added to the supervised objective, where θ are the weights of the network, and λ is a hyper-parameter modulating the strength of the penalty.

†. The number of examples was chosen to be as large as possible, while still allowing for the exploration a variety of hyper-parameters.

‡. Number of hidden units $\in \{400, 800, 1200\}$; learning rate $\in \{0.1, 0.05, 0.02, 0.01, 0.005\}$; ℓ_2 penalty coefficient $\lambda \in \{10^{-4}, 10^{-5}, 10^{-6}, 0\}$; pre-training learning rate $\in \{0.01, 0.005, 0.002, 0.001, 0.0005\}$; corruption probability $\in \{0.0, 0.1, 0.25, 0.4\}$; tied weights $\in \{\text{yes}, \text{no}\}$.

6.6 The Effect of Unsupervised Pre-training

We start by a presentation of large-scale simulations that were intended to confirm some of the previously published results about deep architectures. In the process of analyzing them, we start making connections to our hypotheses and motivate the experiments that follow.

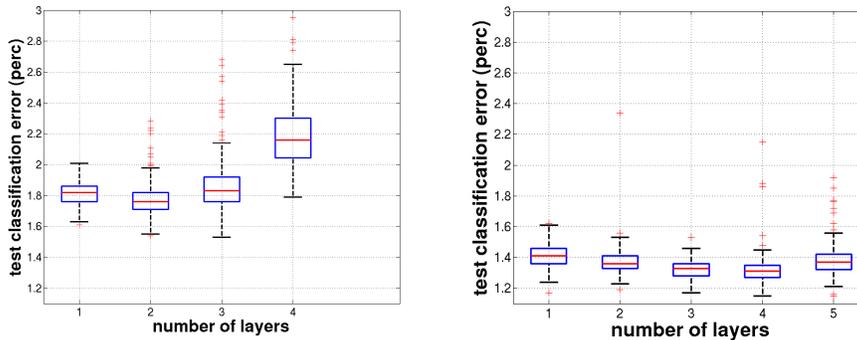
6.6.1 Better Generalization

When choosing the number of units per layer, the learning rate and the number of training iterations to optimize classification error on the validation set, unsupervised pre-training gives substantially lower test classification error than no pre-training, for the same depth or for smaller depth on various vision data sets (Ranzato *et al.*, 2007; Bengio *et al.*, 2007; Larochelle *et al.*, 2009, 2007; Vincent *et al.*, 2008) no larger than the MNIST digit data set (experiments reported from 10,000 to 50,000 training examples).

Such work was performed with only one or a handful of different random initialization seeds, so one of the goals of this study was to ascertain the effect of the random seed used when initializing ordinary neural networks (deep or shallow) and the pre-training procedure. For this purpose, between 50 and 400 different seeds were used to obtain the graphics on MNIST.

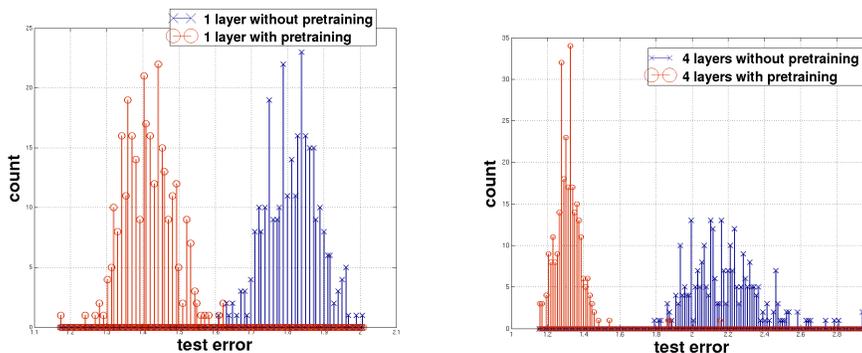
Figure 6.1 shows the resulting distribution of test classification error, obtained with and without pre-training, as we increase the depth of the network. Figure 6.6.1 shows these distributions as histograms in the case of 1 and 4 layers. As can be seen in Figure 6.1, unsupervised pre-training allows classification error to go down steadily as we move from 1 to 4 hidden layers, whereas without pre-training the error goes up after 2 hidden layers. It should also be noted that we were unable to effectively train 5-layer models without use of unsupervised pre-training. Not only is the error obtained on average with unsupervised pre-training systematically lower than without the pre-training, it appears also more robust to the random initialization. With unsupervised pre-training the variance stays at about the same level up to 4 hidden layers, with the number of bad outliers growing slowly.

Contrast this with the case without pre-training: the variance and number of bad outliers grows sharply as we increase the number of layers beyond 2. The gain obtained with unsupervised pre-training is more pronounced as we increase the number of layers, as is the gain in robustness to random initialization. This can be seen in Figure 6.6.1. The



▲ **Figure 6.1.** Effect of depth on performance for a model trained (**left**) without unsupervised pre-training and (**right**) with unsupervised pre-training, for 1 to 5 hidden layers (networks with 5 layers failed to converge to a solution, without the use of unsupervised pre-training). Experiments on MNIST. Box plots show the distribution of errors associated with 400 different initialization seeds (top and bottom quartiles in box, plus outliers beyond top and bottom quartiles). Other hyperparameters are optimized away (on the validation set). Increasing depth seems to increase the probability of finding poor apparent local minima.

increase in error variance and mean for deeper architectures without pre-training suggests that **increasing depth increases the probability of finding poor apparent local minima** when starting from random initialization. It is also interesting to note the low variance and small spread of errors obtained with 400 seeds with unsupervised pre-training: it suggests that **unsupervised pre-training is robust with respect to the random initialization seed** (the one used to initialize parameters before pre-training).



◀ **Figure 6.2.** Histograms presenting the test errors obtained on MNIST using models trained with or without pre-training (400 different initializations each). **Left:** 1 hidden layer. **Right:** 4 hidden layers.

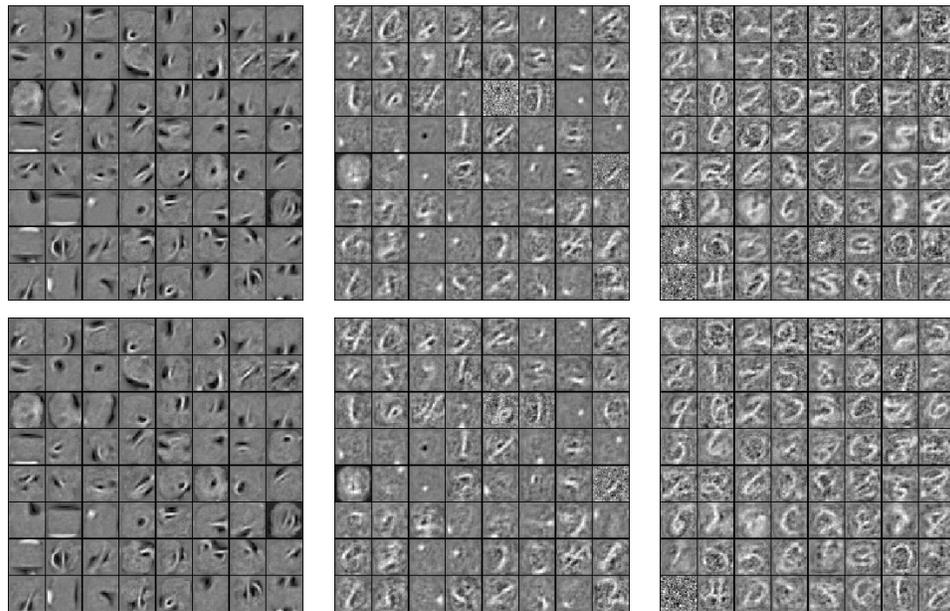
These experiments show that the variance of final test error with

respect to initialization random seed is larger without pre-training, and this effect is magnified for deeper architectures. It should however be noted that there is a limit to the success of this technique: performance degrades for 5 layers on this problem.

6.6.2 Visualization of Features

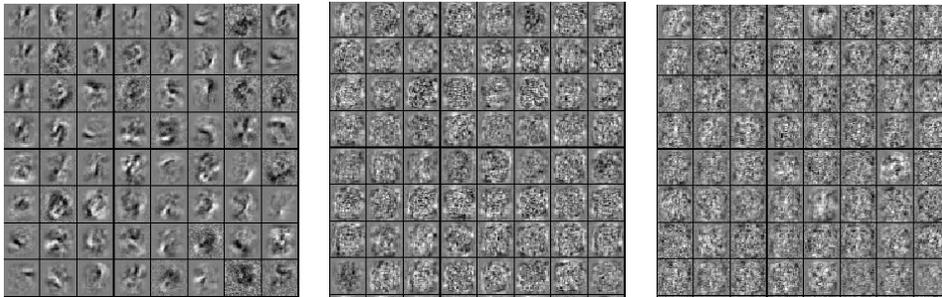
Figure 6.3 shows the weights (called filters) of the first layer of the DBN before and after supervised fine-tuning. For visualizing what units do on the 2nd and 3rd layer, we used the activation maximization technique described in Chapter 8: to visualize what a unit responds most to, the method looks for the bounded input pattern that maximizes the activation of a given unit. This is an optimization problem which is solved by performing gradient ascent in the space of the inputs, to find a local maximum of the activation function. Interestingly, nearly the same maximal activation input pattern is recovered from most random initializations of the input pattern.

► **Figure 6.3.** Visualization of filters learned by a DBN trained on *InfiniteMNIST*. The top figures contain a visualization of filters after pre-training, while the bottoms ones picture the same units after supervised fine-tuning; from left to right: units from the 1st, 2nd and 3rd layers, respectively.



For comparison, we have also visualized the filters of a network for 1–3 layers in which no pre-training was performed (Figure 6.4). While the first layer filters do seem to correspond to localized features, 2nd and 3rd layers are not as interpretable anymore. Qualitatively speaking, filters from the bottom row of Figure 6.3 and those from Figure 6.4 have little in common, which is an interesting conclusion in

itself. In addition, there seems to be more interesting visual structures in the features learned in networks with unsupervised pre-training.



◀ **Figure 6.4.** Visualization of filters learned by a network without pre-training, trained on *InfiniteMNIST*. The filters are shown after supervised training; from left to right: units from the 1st, 2nd and 3rd layers, respectively. The visualizations are produced with the technique described in Chapter 8

Several interesting conclusions can be drawn from Figure 6.3. First, supervised fine-tuning (after unsupervised pre-training), even with 7.5 million updates, does not change the weights in a significant way (at least visually): they seem stuck in a certain region of weight space, and the sign of weights does not change after fine-tuning (hence the same pattern is seen visually). Second, different layers change differently: the first layer changes least, while supervised training has more effect when performed on the 3rd layer. Such observations are consistent with the predictions made by our hypothesis: namely that the early dynamics of stochastic gradient descent, the dynamics induced by unsupervised pre-training, can “lock” the training in a region of the parameter space that is essentially inaccessible for models that are trained in a purely supervised way.

Finally, the features increase in complexity as we add more layers. First layer weights seem to encode basic stroke-like detectors, second layer weights seem to detect digit parts, while top layer weights detect entire digits. The features are more complicated as we add more layers, and displaying only one image for each “feature” does not do justice to the non-linear nature of that feature. For example, it does not show the *set of patterns* on which the feature is highly active (or highly inactive).

While Figures 6.3–6.4 show only the filters obtained on *InfiniteMNIST*, the visualizations are similar when applied on *MNIST*. Likewise, the features obtained with SDAE result in qualitatively similar conclusions; Chapter 8 gives more details.

6.6.3 Visualization of Model Trajectories During Learning

Visualizing the learned features allows for a qualitative comparison of the training strategies for deep architectures. However it is not

useful for investigating how these strategies are influenced by random initialization, as the features learned from multiple initializations look similar. If it was possible for us to visualize a variety of models at the same time, it would allow us to explore our hypothesis, and ascertain to what degree and how the set of pre-trained models (for different random seeds) is far from the set of models without pre-training. Do these two sets cover very different regions in parameter space? Are parameter trajectories getting stuck in many different apparent local minima?

Unfortunately, it is not possible to directly compare parameter values of two architectures, because many permutations of the same parameters give rise to the same model. However, one can take a functional approximation approach in which we compare the function (from input to output) represented by each network, rather than comparing the parameters. The function is the infinite ordered set of output values associated with all possible inputs, and it can be approximated with a finite number of inputs (preferably plausible ones). To visualize the trajectories followed during training, we use the following procedure. For a given model, we compute and concatenate all its outputs on the test set examples as one long vector summarizing where it stands in “function space”. We get one such vector for each partially trained model (at each training iteration). This allows us to plot many learning trajectories, one for each initialization seed, with or without pre-training. Using a dimensionality reduction algorithm we then map these vectors to a two-dimensional space for visualization.* Figures 6.5 and 6.6 present the results using dimensionality reduction techniques that focus respectively on local[†] and global structure.[‡] Each point is colored according to the training iteration, to help follow the trajectory movement.

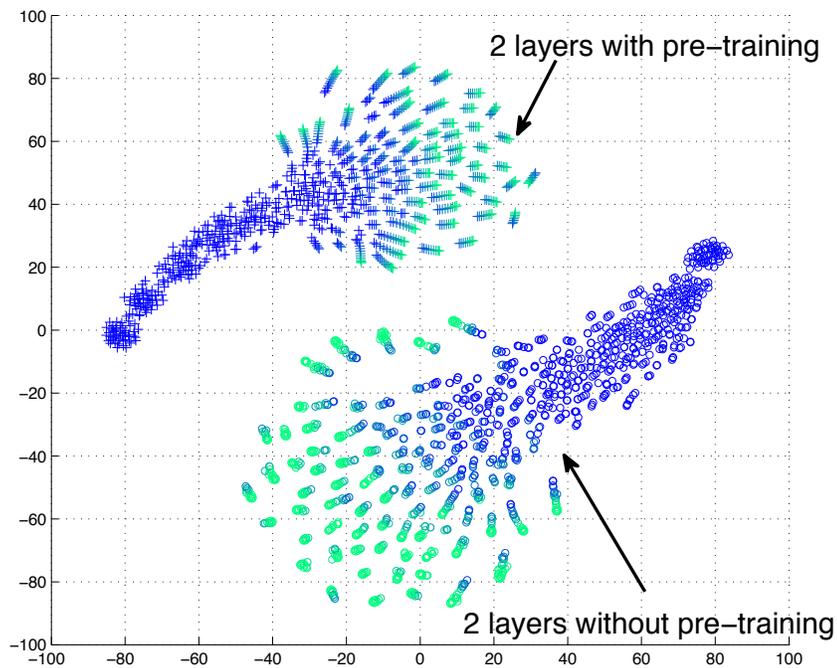
What seems to come out of these visualizations is the following:

1. The pre-trained and not pre-trained models start and *stay* in different regions of function space.
2. From the visualization focusing on local structure (Figure 6.5) we see that all trajectories of a given type (with pre-training or without) initially move together. However, at some point (after about 7 epochs) the different trajectories (corresponding to different random seeds) diverge (slowing down into elongated jets)

*. Note that we can and do project the models with and without pre-training at the same time, so as to visualize them in the same space.

†. t-Distributed Stochastic Neighbor Embedding, or tSNE, by van der Maaten and Hinton (2008), with the default parameters available in the public implementation: <http://ict.ewi.tudelft.nl/~lvandermaaten/t-SNE.html>.

‡. Isomap by Tenenbaum *et al.* (2000), with one connected component.



◀ **Figure 6.5.** 2D visualizations with *tSNE* of the functions represented by 50 networks with and 50 networks without pre-training, as supervised training proceeds over MNIST. See Section 6.6.3 for an explanation. Color from dark blue to cyan and red indicates a progression in training iterations (training is longer without pre-training). The plot shows models with 2 hidden layers but results are similar with other depths.

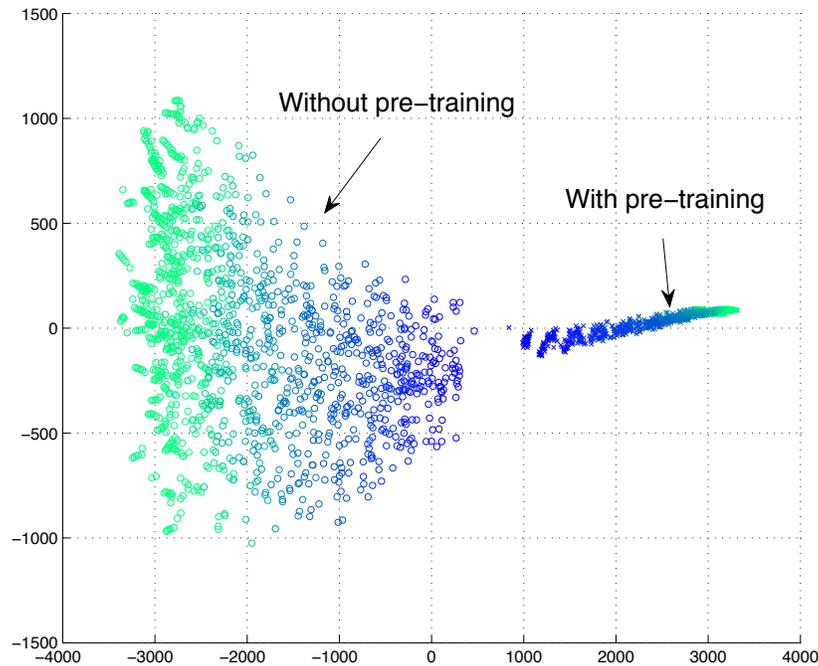
and never get back close to each other (this is more true for trajectories of networks without pre-training). This suggests that each trajectory moves into a different apparent local minimum.*

3. From the visualization focusing on global structure (Figure 6.6), we see that the pre-trained models are plausibly in a disjoint and much smaller region of space than the not pre-trained models (as far as we can interpret ISOMAP results this way). In fact, from the standpoint of the functions found without pre-training, the pre-trained solutions look all the same, and their self-similarity increases (variance across seeds decreases) during training, while the opposite is observed without pre-training. This is consistent with the formalization of pre-training from Section 6.3, in which we described a theoretical justification for viewing unsupervised pre-training as a regularizer; there, the probabilities of pre-training parameters landing in a basin of attraction is small.

The visualizations of the training trajectories do seem to confirm

*. One may wonder if the divergence points correspond to a turning point in terms of overfitting. As shall be seen in Figure 6.8, the test error does not improve much after the 7th epoch, which reinforces this hypothesis.

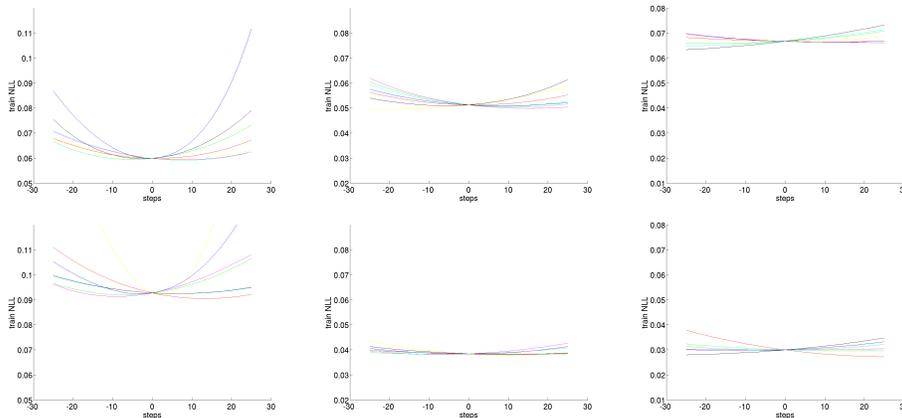
► **Figure 6.6.** 2D visualization with ISOMAP of the functions represented by 50 networks with and 50 networks without pre-training, as supervised training proceeds over MNIST. See Section 6.6.3 for an explanation. Color from dark blue to cyan indicates a progression in training iterations (training is longer without pre-training). The plot shows models with 2 hidden layers but results are similar with other depths.



our suspicions. It is difficult to guarantee that each trajectory actually does end up in a different local minimum (corresponding to a different function and not only to different parameters). However, all tests performed (visual inspection of trajectories in function space, but also estimation of second derivatives in the directions of all the estimated eigenvectors of the Jacobian not reported in details here) were consistent with that interpretation.

We have also analyzed models obtained at the end of training, to visualize the training criterion in the neighborhood of the parameter vector θ^* obtained. This is achieved by randomly sampling a direction v (from the stochastic gradient directions) and by plotting the training criterion around θ^* in that direction, that is, at $\theta = \theta^* + \alpha v$, for $\alpha \in \{-2.5, -2.4, \dots, -0.1, 0, 0.1, \dots, 2.4, 2.5\}$, and v normalized ($\|v\| = 1$). This analysis is visualized in Figure 6.7. The error curves look close to quadratic and we seem to be near a local minimum in all directions investigated, as opposed to a saddle point or a plateau. A more definite answer could be given by computing the full Hessian eigenspectrum, which would be expensive. Figure 6.7 also suggests that the error landscape is a bit flatter in the case of unsupervised pre-training, and

flatter for deeper architectures.



▲ **Figure 6.7.** Training errors obtained on *Shapaset* when stepping in parameter space around a converged model in 7 random gradient directions (stepsize of 0.1). **Top:** no pre-training. **Bottom:** with unsupervised pre-training. **Left:** 1 hidden layer. **Middle:** 2 hidden layers. **Right:** 3 hidden layers. Compare also with Figure 6.8, where 1-layer networks with unsupervised pre-training obtain higher training errors.

6.6.4 Implications

The series of results presented so far show a picture that is consistent with our hypothesis. Better generalization that seems to be robust to random initializations is indeed achieved by pre-trained models, which indicates that unsupervised learning of $P(X)$ is helpful in learning $P(Y|X)$. The function space landscapes that we visualized point to the fact that there are many apparent local minima. The pre-trained models seem to end up in distinct regions of these error landscapes (and, implicitly, in different parts of the parameter space). This is both seen from the function space trajectories and from the fact that the visualizations of the learned features are qualitatively very different from those obtained by models without pre-training.

6.7 The Role of Unsupervised Pre-training

The observations so far in this paper confirm that starting the supervised optimization from pre-trained weights rather than from ran-

domly initialized weights consistently yields better performing classifiers on MNIST. To better understand where this advantage came from, it is important to realize that the *supervised objective being optimized is exactly the same in both cases*. The gradient-based optimization procedure is also the same. The only thing that differs is the starting point in parameter space: either picked at random or obtained after unsupervised pre-training (which also starts from a random initialization).

Deep architectures, since they are built from the composition of several layers of non-linearities, yield an error surface that is non-convex and hard to optimize, with the suspected presence of many local minima (as also shown by the above visualizations). A gradient-based optimization should thus end in the apparent local minimum of whatever *basin of attraction* we started from. From this perspective, the advantage of unsupervised pre-training could be that it puts us in a region of parameter space where basins of attraction run deeper than when picking starting parameters at random. The advantage would be due to a better **optimization**.

Now it might also be the case that unsupervised pre-training puts us in a region of parameter space in which training error is not necessarily better than when starting at random (or possibly worse), but which systematically yields better generalization (test error). Such behavior would be indicative of a **regularization** effect. Note that the two forms of explanation are *not necessarily mutually exclusive*.

Finally, a very simple explanation could be the most obvious one: namely the disparity in the magnitude of the weights (or more generally, the marginal distribution of the weights) at the start of the supervised training phase. We shall analyze (and rule out) this hypothesis first.

6.7.1 Experiment 1: Does Pre-training Provide a Better Conditioning Process for Supervised Learning?

Typically gradient descent training of the deep model is initialized with randomly assigned weights, small enough to be in the linear region of the parameter space (close to zero for most neural network and DBN models). It is reasonable to ask if the advantage imparted by having an initial unsupervised pre-training phase is simply due to the weights being larger and therefore somehow providing a better “conditioning” of the initial values for the optimization process; we wanted to rule out this possibility.

By conditioning, we mean the range and marginal distribution from which we draw initial weights. In other words, could we get the same

performance advantage as unsupervised pre-training if we were still drawing the initial weights independently, but from a more suitable distribution than the uniform $[-1/\sqrt{k}, 1/\sqrt{k}]$? To verify this, we performed unsupervised pre-training, and computed marginal histograms for each layer’s pre-trained weights and biases (one histogram per each layer’s weights and biases). We then resampled new “initial” random weights and biases according to these histograms (independently for each parameter), and performed fine-tuning from there. The resulting parameters have the same marginal statistics as those obtained after unsupervised pre-training, but not the same joint distribution.

Two scenarios can be imagined. In the first, the initialization from marginals would lead to significantly better performance than the standard initialization (when no pre-training is used). This would mean that unsupervised pre-training does provide a better marginal conditioning of the weights. In the second scenario, the marginals would lead to performance similar to or worse than that without pre-training.*

initialization.	Uniform	Histogram	Unsup.pre-tr.
1 layer	1.81 ± 0.07	1.94 ± 0.09	1.41 ± 0.07
2 layers	1.77 ± 0.10	1.69 ± 0.11	1.37 ± 0.09

Table 6.1. *Effect of various initialization strategies on 1 and 2-layer architectures: independent uniform densities (one per parameter), independent densities from the marginals after unsupervised pre-training, or unsupervised pre-training (which samples the parameters in a highly dependent way so that they collaborate to make up good denoising auto-encoders.) Experiments on MNIST, numbers are mean and standard deviation of test errors (across different initialization seeds).*

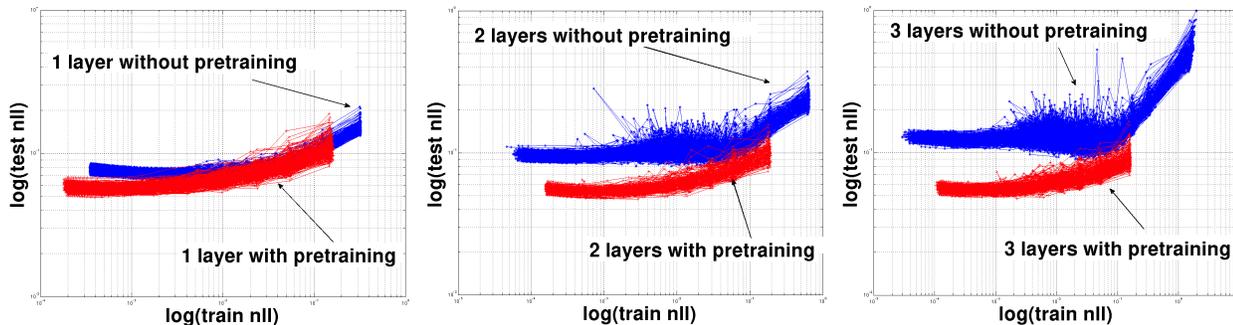
What we observe in Table 6.1 seems to fall within the first scenario. However, while initializing the weights to match the marginal distributions at the end of pre-training appears to slightly improve the generalization error on MNIST for 2 hidden layers, the difference is not significant and it is far from fully accounting for the discrepancy between the pre-trained and non-pre-trained results.

This experiment constitutes evidence against the preconditioning hypothesis, but does not exclude either the optimization hypothesis or the regularization hypothesis.

*. We observed that the distribution of weights after unsupervised pre-training is fat-tailed. It is conceivable that sampling from such a distribution in order to initialize a deep architecture might actually *hurt* the performance of a deep architecture (compared to random initialization from a uniform distribution).

6.7.2 Experiment 2: The Effect of Pre-training on Training Error

The optimization and regularization hypotheses diverge on their prediction on how unsupervised pre-training should affect the training error: the former predicts that unsupervised pre-training should result in a lower training error, while the latter predicts the opposite. To ascertain the influence of these two possible explanatory factors, we looked at the test cost (Negative Log Likelihood on test data) obtained as a function of the training cost, along the trajectory followed in parameter space by the optimization procedure. Figure 6.8 shows 400 of these curves started from a point in parameter space obtained from random initialization, that is, without pre-training (blue), and 400 started from pre-trained parameters (red).



▲ **Figure 6.8.** Evolution without pre-training (blue) and with pre-training (red) on MNIST of the log of the test NLL plotted against the log of the train NLL as training proceeds. Each of the 2×400 curves represents a different initialization. The errors are measured after each pass over the data. The rightmost points were measured after the first pass of gradient updates. Since training error tends to decrease during training, the trajectories run from right (high training error) to left (low training error). Trajectories moving up (as we go leftward) indicate a form of overfitting. All trajectories are plotted on top of each other.

The experiments were performed for networks with 1, 2 and 3 hidden layers. As can be seen in Figure 6.8, while for 1 hidden layer, unsupervised pre-training reaches lower training cost than no pre-training, hinting towards a better optimization, this is not necessarily the case for the deeper networks. The remarkable observation is rather that, *at a same training cost level, the pre-trained models systematically yield a lower test cost than the randomly initialized ones.* The advantage appears to be one of *better generalization rather than merely a better*

optimization procedure.

This brings us to the following result: unsupervised pre-training appears to have a similar effect to that of a good regularizer or a good “prior” on the parameters, even though no explicit regularization term is apparent in the cost being optimized. As we stated in the hypothesis, it might be reasoned that restricting the possible starting points in parameter space to those that minimize the unsupervised pre-training criterion (as with the SDAE), does in effect restrict the set of possible final configurations for parameter values. Like regularizers in general, unsupervised pre-training (in this case, with denoising auto-encoders) might thus be seen as decreasing the variance and introducing a bias (towards parameter configurations suitable for performing denoising). Unlike ordinary regularizers, unsupervised pre-training does so in a data-dependent manner.

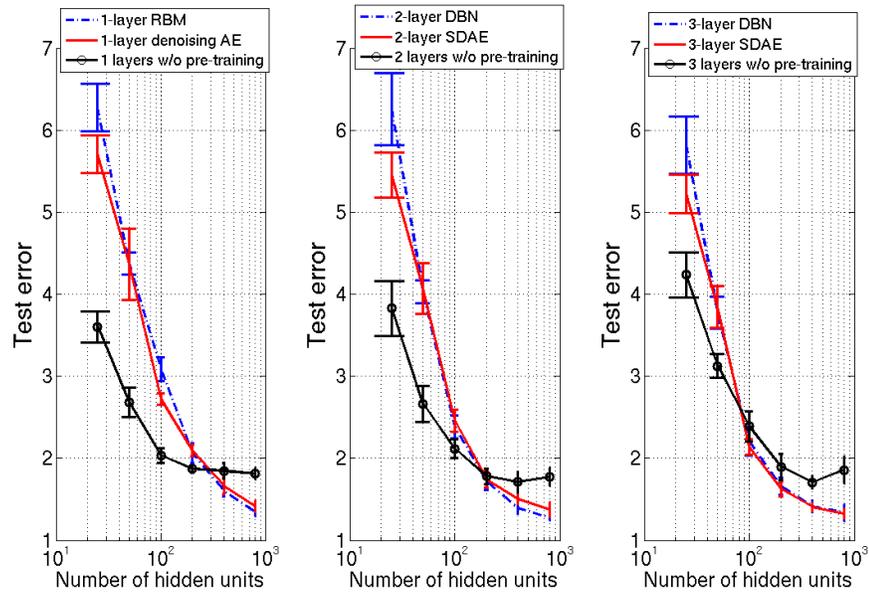
6.7.3 Experiment 3: The Influence of the Layer Size

Another signature characteristic of regularization is that the effectiveness of regularization increases as capacity (e.g., the number of hidden units) increases, effectively trading off one constraint on the model complexity for another. In this experiment we explore the relationship between the number of units per layer and the effectiveness of unsupervised pre-training. The hypothesis that unsupervised pre-training acts as a regularizer would suggest that we should see a trend of increasing effectiveness of unsupervised pre-training as the number of units per layer are increased.

We trained models on MNIST with and without pre-training using increasing layer sizes: 25, 50, 100, 200, 400, 800 units per layer. Results are shown in Figure 6.9. Qualitatively similar results were obtained on *Shapaset*. In the case of SDAE, we were expecting the denoising pre-training procedure to help classification performance most for large layers; this is because the denoising pre-training allows useful representations to be learned in the over-complete case, in which a layer is larger than its input (Vincent *et al.*, 2008). What we observe is a more systematic effect: while unsupervised pre-training helps for larger layers and deeper networks, it also appears to hurt for too small networks.

Figure 6.9 also shows that DBNs behave qualitatively like SDAEs, in the sense that unsupervised pre-training architectures with smaller layers hurts performance. Experiments on *InfiniteMNIST* reveal results that are qualitatively the same. Such an experiment seemingly points to a re-verification of the regularization hypothesis. In this case,

► **Figure 6.9.** Effect of layer size on the changes brought by unsupervised pre-training, for networks with 1, 2 or 3 hidden layers. Experiments on MNIST. Error bars have a height of two standard deviations (over initialization seed). Pre-training hurts for smaller layer sizes and shallower networks, but it helps for all depths for larger networks.



it would seem that unsupervised pre-training acts as an additional regularizer for both DBN and SDAE models—on top of the regularization provided by the small size of the hidden layers. As the model size decreases from 800 hidden units, the generalization error increases, and it increases more with unsupervised pre-training presumably because of the extra regularization effect: small networks have a limited capacity already so further restricting it (or introducing an additional bias) can harm generalization. Such a result seems incompatible with a pure optimization effect. We also obtain the result that DBNs and SDAEs seem to have qualitatively similar effects as pre-training strategies.

The effect can be explained in terms of the role of unsupervised pre-training as promoting input transformations (in the hidden layers) that are useful at capturing the main variations in the input distribution $P(X)$. It may be that only a small subset of these variations are relevant for predicting the class label Y . When the hidden layers are small it is less likely that the transformations for predicting Y are included in the lot learned by unsupervised pre-training.

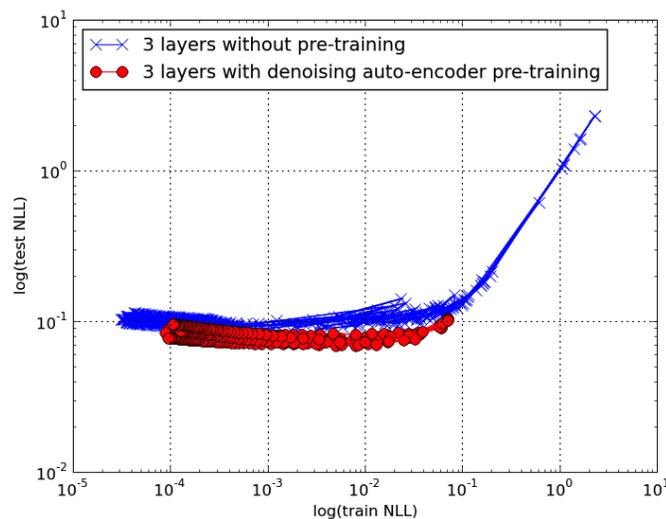
6.7.4 Experiment 4: Challenging the Optimization Hypothesis

Experiments 1–3 results are consistent with the regularization hypothesis and Experiments 2–3 would appear to directly support the

regularization hypothesis over the alternative—that unsupervised pre-training aids in optimizing the deep model objective function.

In the literature there is some support for the optimization hypothesis. Bengio *et al.* (2007) constrained the top layer of a deep network to have 20 units and measured the training error of networks with and without pre-training. The idea was to prevent the networks from over-fitting the training error simply with the top hidden layer, thus to make it clearer whether some optimization effect (of the lower layers) was going on. The reported training and test errors were lower for pre-trained networks. One problem with the experimental paradigm used by Bengio *et al.* (2007) is their use of early stopping. This is problematic because, as previously mentioned, early stopping is itself a regularizer, and it can influence greatly the training error that is obtained. It is conceivable that if Bengio *et al.* (2007) had run the models to convergence, the results could have been different. We needed to verify this.

Figure 6.10 shows what happens without early stopping. The training error is still higher for pre-trained networks even though the generalization error is lower. This result now favors the regularization hypothesis against the optimization story. What may have happened is that early stopping prevented the networks without pre-training from moving too much towards their apparent local minimum.



◀ **Figure 6.10.** For MNIST, a plot of the $\log(\text{train NLL})$ vs. $\log(\text{test NLL})$ at each epoch of training. The top layer is constrained to 20 units.

6.7.5 Experiment 5: Comparing pre-training to L_1 and L_2 regularization

An alternative hypothesis would be that classical ways of regularizing could perhaps achieve the same effect as unsupervised pre-training. We investigated the effect of L_1 and L_2 regularization (i.e., adding a $\|\theta\|_1$ or $\|\theta\|_2^2$ term to the supervised objective function) in a network without pre-training. We found that while in the case of MNIST a small penalty can in principle help, the gain is nowhere near as large as it is with pre-training. For InfiniteMNIST, the optimal amount of L_1 and L_2 regularization is zero.*

This is not an entirely surprising finding: not all regularizers are created equal and these results are consistent with the literature on semi-supervised training that shows that unsupervised learning can be exploited as a particularly effective form of regularization.

6.7.6 Summary of Findings: Experiments 1-5

So far, the results obtained from the previous experiments point towards a pretty clear explanation of the effect of unsupervised pre-training: namely, that its effect is a regularization effect. We have seen that it is not simply sufficient to sample random weights with the same magnitude: the (data-dependent) unsupervised initialization is crucial. We have also observed that canonical regularizers (L_1/L_2 penalties on the weights) do not achieve the same level of performance.

The most compelling pieces of evidence in support of the regularization hypothesis are Figures 6.8 and 6.9. The alternative explanation—that unsupervised pre-training has an optimization effect—suggested by Bengio *et al.* (2007) doesn't seem to be supported by our experimental setup.

6.8 The Online Learning Setting

Our hypothesis included not only the statistical/phenomenological hypothesis that unsupervised pre-training acted as a regularizer, but also contains a mechanism for how such behavior arises both as a consequence of the dynamic nature of training—following a stochastic gradient through two phases of training and as a consequence of the non-convexity of the supervised objective function.

*. Which is consistent with the classical view of regularization, in which its effect should diminish as we add more and more data.

In our hypothesis, we posited that early examples induce changes in the magnitude of the weights that increase the amount of non-linearity of the network, which in turn decreases the number of regions accessible to the stochastic gradient descent procedure. This means that the early examples (be they pre-training examples or otherwise) determine the basin of attraction for the remainder of training; this also means that the early examples have a disproportionate influence on the configuration of parameters of the trained models.

One consequence to the hypothesized mechanism is that we would predict that in the online learning setting with unbounded or very large data sets, the behavior of unsupervised pre-training would diverge from the behavior of a canonical regularizer (L_1/L_2). This is because the effectiveness of a canonical regularizer **decreases** as the data set grows, whereas the effectiveness of unsupervised pre-training as a regularizer is **maintained** as the data set grows.

Note that stochastic gradient descent in online learning is a stochastic gradient descent optimization of the generalization error, so good online error in principle implies that we are optimizing well the generalization error. Indeed, each gradient $\frac{\partial L(x,y)}{\partial \theta}$ for example (x, y) (with $L(x, y)$ the supervised loss with input x and label y) sampled from the true generating distribution $P(x, y)$ is an unbiased Monte-Carlo estimator of the true gradient of generalization error, that is, $\sum_y \int_x \frac{\partial L(x,y)}{\partial \theta} P(x, y) dx$.

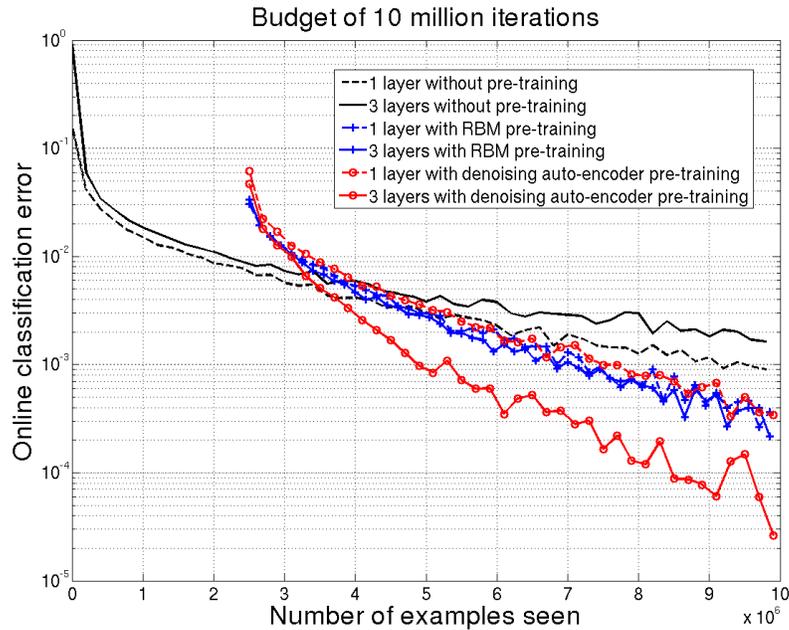
In this section we empirically challenge this aspect of the hypothesis and show that the evidence does indeed support our hypothesis over what is more typically expected from a regularizer.

6.8.1 Experiment 6: Effect of Pre-training with Very Large Data Sets

The results presented here are perhaps the most surprising findings of this paper. Figure 6.11 shows the online classification error (on the next block of examples, as a moving average) for 6 architectures that are trained on `InfiniteMNIST`: 1 and 3-layer DBNs, 1 and 3-layer SDAE, as well as 1 and 3-layer networks without pre-training.

We can draw several observations from these experiments. First, 3-layer networks without pre-training are worse at generalization, compared to the 1-layer equivalent. This confirms the hypothesis that even in an online setting, optimization of deep networks is harder than shallow ones. Second, 3-layer SDAE models seem to generalize better than 3-layer DBNs. Finally and most importantly, the pre-training advantage does not vanish as the number of training examples increases, on the contrary.

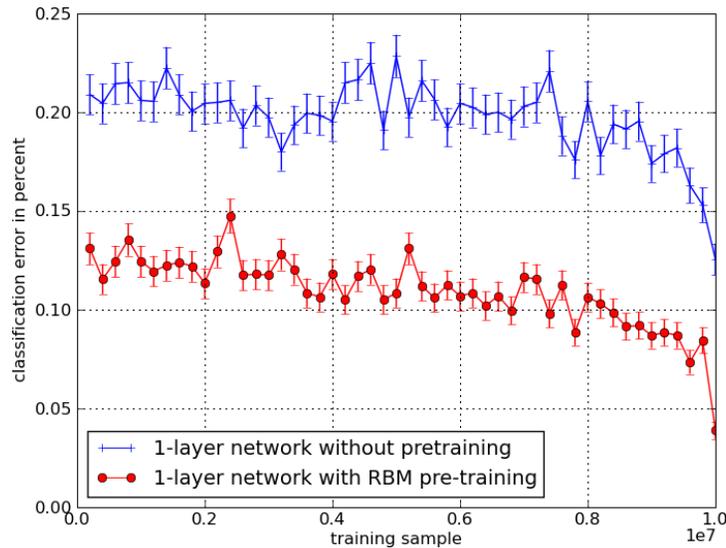
► **Figure 6.11.** Comparison between 1 and 3-layer networks trained on *InfiniteMNIST*. Online classification error, computed as an average over a block of last 100,000 errors.



Note that the number of hidden units of each model is a hyperparameter.* So theoretical results suggest that 1-layer networks without pre-training should in principle be able to represent the input distribution as capacity and data grow. Instead, without pre-training, the networks are not able to take advantage of the additional capacity, which again points towards the optimization explanation. It is clear, however, that **the starting point of the non-convex optimization matters**, even for networks that are seemingly “easier” to optimize (1-layer ones), which supports our hypothesis.

Another experiment that shows the effects of large-scale online stochastic non-convex optimization is shown in Figure 6.12. In the setting of *InfiniteMNIST*, we compute the error on the *training set*, in the same order that we presented the examples to the models. We observe several interesting results: first, note that both models are better at classifying more recently seen examples. This is a natural effect of stochastic gradient descent with a constant learning rate (which gives exponentially more weight to recent examples). Note also that examples at the beginning of training are essentially like test examples for both models, in terms of error. Finally, we observe that the pre-trained model is better across the board *on the training set*. This fits

*. This number was chosen individually for each model s.t. the error on the last 1 million examples is minimized. In practice, this meant 2000 units for 1-layer networks and 1000 units/layer for 3-layer networks.



◀ **Figure 6.12.** Error of 1-layer network with RBM pre-training and without, on the 10 million examples used for training it. The errors are calculated in the same order (from left to right, above) as the examples were presented during training. Each error bar corresponds to a block of consecutive training examples.

well with the optimization hypothesis, since it shows that unsupervised pre-training has an optimization effect.

What happens in this setting is that the training and generalization errors converge as the empirical distribution (defined by the training set) converges to the true data distribution. These results show that the effectiveness of unsupervised pre-training does not diminish with increasing data set sizes. This would be unexpected from a superficial understanding of unsupervised pre-training as a regularization method. However it is entirely consistent with our interpretation, stated in our hypothesis, of the role of unsupervised pre-training in the online setting with stochastic gradient descent training on a non-convex objective function.

6.8.2 Experiment 7: The Effect of Example Ordering

The hypothesized mechanism implies, due to the dynamics of learning—the increase in weight magnitude and non-linearity as training proceeds, as well as the dependence of the basin of attraction on early data—that, when training with stochastic gradient descent, we should see increased sensitivity to early examples. In the case of `InfiniteMNIST` we operate in an online stochastic optimization regime, where we try to find a local minimum of a highly non-convex objective function. It is then interesting to study to what extent the outcome of this optimization is influenced by the examples seen at different points during training,

and whether the early examples have a stronger influence (which would not be the case with a convex objective).

To quantify the variance of the outcome with respect to training samples at different points during training, and to compare these variances for models with and without pre-training, we proceeded with the following experiment. Given a data set with 10 million examples, we vary (by resampling) the first million examples (across 10 different random draws, sampling a different set of 1 million examples each time) and keep the other ones fixed. After training the (10) models, we measure the variance (across the 10 draws) of the *output* of the networks on a fixed test set (i.e., we measure the variance in function space). We then vary the next million examples in the same fashion, and so on, to see how much each of the ten parts of the training set influenced the final function.

► **Figure 6.13.** Variance of the output of a trained network with 1 layer. The variance is computed as a function of the point at which we vary the training samples. Note that the 0.25 mark corresponds to the start of supervised fine-tuning.

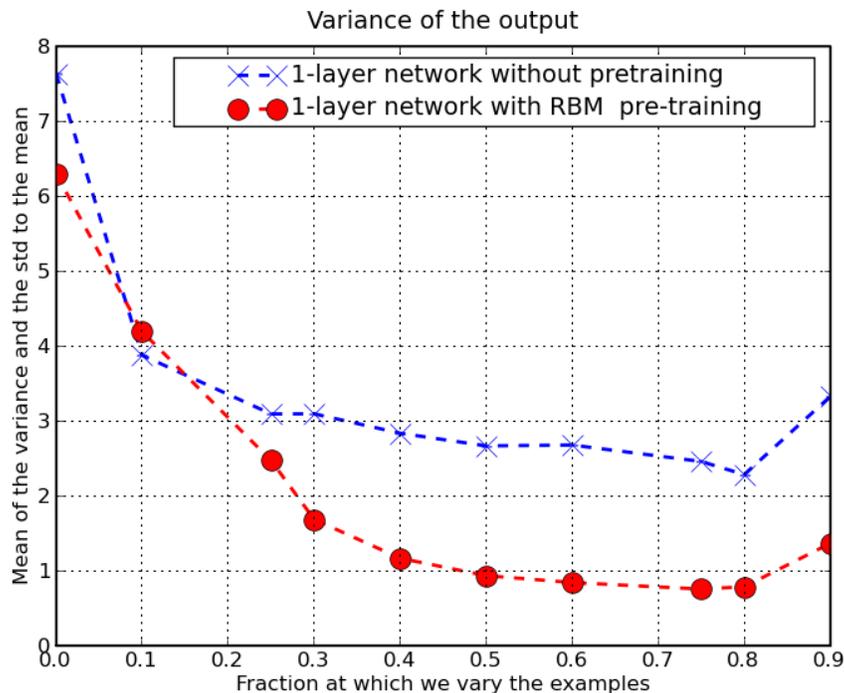


Figure 6.13 shows the outcome of such an analysis. The samples at the beginning* do seem to influence the output of the networks more than the ones at the end. However, this variance is *lower* for the networks that have been pre-trained. In addition to that, one should note that the variance of pre-trained network at 0.25 (i.e., the variance

*. Which are *unsupervised* examples, for the red curve, until the 0.25 mark in Figure 6.13.

of the output as a function of the first samples used for supervised training) is *lower* than the variance of the supervised network at 0.0. Such results imply that unsupervised pre-training can be seen as a sort of variance reduction technique, consistent with a regularization hypothesis. Finally, both networks are more influenced by the *last examples* used for optimization, which is simply due to the fact that we use stochastic gradient with a constant learning rate, where the most recent examples' gradient has a greater influence.

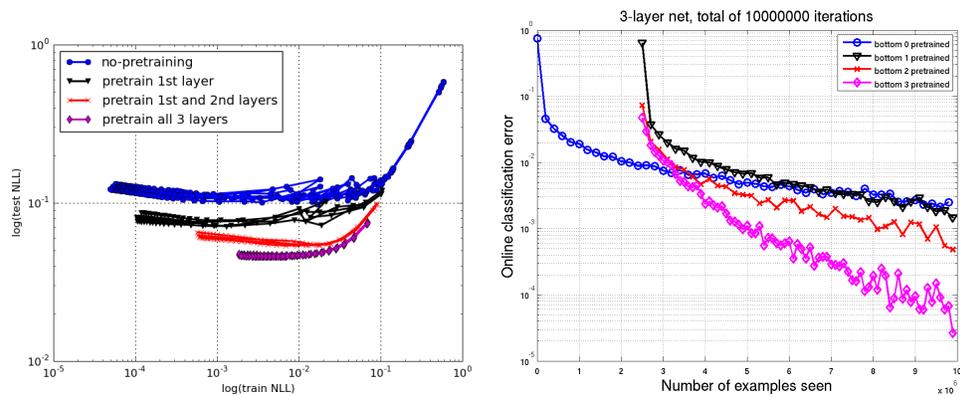
These results are consistent with what our hypothesis predicts: both the fact that early examples have greater influence (i.e., the variance is higher) and that pre-trained models seem to reduce this variance are in agreement with what we would have expected.

6.8.3 Experiment 8: Pre-training only k layers

From Figure 6.11 we can see that unsupervised pre-training makes quite a difference for 3 layers, on `InfiniteMNIST`. In Figure 6.14 we explore the link between depth and unsupervised pre-training in more detail. The setup is as follows: for both `MNIST` and `InfiniteMNIST` we pre-train only the bottom k layers and randomly initialize the top $n - k$ layers in the usual way. In this experiment, $n = 3$ and we vary k from 0 (which corresponds to a network with no pre-training) to $k = n$ (which corresponds to the normal pre-trained case).

For `MNIST`, we plot the $\log(\text{train NLL})$ vs. $\log(\text{test NLL})$ trajectories, where each point corresponds to a measurement after a certain number of epochs. The trajectories go roughly from the right to left and from top to bottom, corresponding to the lowering of the training and test errors. We can also see that models overfit from a certain point onwards.

For `InfiniteMNIST`, we simply show the online error. The results are ambiguous w.r.t the difficulty of optimizing the lower layers versus the higher ones. We would have expected that the largest incremental benefit came from pre-training the first layer or first two layers. It is true for the first two layers, but not the first. As we pre-train more layers, the models become better at generalization. In the case of the finite `MNIST`, note how the final training error (after the same number of epochs) becomes *worse* with pre-training of more layers. This clearly brings additional support to the regularization explanation.



▲ **Figure 6.14.** On the left: for *MNIST*, a plot of the $\log(\text{train NLL})$ vs. $\log(\text{test NLL})$ at each epoch of training. We pre-train the first layer, the first two layers and all three layers using RBMs and randomly initialize the other layers; we also compare with the network whose layers are all randomly initialized. On the right: *InfiniteMNIST*, the online classification error. We pre-train the first layer, the first two layers or all three layers using denoising auto-encoders and leave the rest of the network randomly initialized.

6.9 Discussion and Conclusions

We have shown that unsupervised pre-training adds robustness to a deep architecture. The same set of results also suggests that increasing the depth of an architecture that is not pre-trained increases the probability of finding poor apparent local minima. Pre-trained networks give consistently better generalization. Our visualizations point to the observations that pre-trained networks learn qualitatively different features (if networks are visualized in the weight space) compared to networks without pre-training. Moreover, the trajectories of networks with different initialization seeds seem to fall into many distinct apparent local minima, which are again different (and seemingly far apart) depending on whether we use pre-training or not.

We have shown that unsupervised pre-training is not simply a way of getting a good initial marginal distribution, and that it captures more intricate dependencies between parameters. One of our findings is that deep networks with unsupervised pre-training seem to exhibit some properties of a regularizer: with small enough layers, pre-trained deep architectures are systematically worse than randomly initialized deep architectures. Moreover, when the layers are big enough, the pre-trained models obtain worse training errors, but better generalization

performance. Additionally, we have re-done an experiment which purportedly showed that unsupervised pre-training can be explained with an optimization hypothesis and observed a regularization effect instead. We also showed that classical regularization techniques (such as L_1/L_2 penalties on the network weights) cannot achieve the same performance as unsupervised pre-training, and that the effect of unsupervised pre-training does not go away with more training data, so if unsupervised pre-training is a regularizer, it is certainly of a rather different kind.

The two unsupervised pre-training strategies considered—denoising auto-encoders and Restricted Boltzmann Machines—seem to produce qualitatively similar observations. We have observed that, surprisingly, the pre-training advantage is present even in the case of really large training sets, pointing towards the conclusion that the starting point in the non-convex optimization problem is indeed quite important; a fact confirmed by our visualizations of filters at various levels in the network. Finally, the other important set of results show that unsupervised pre-training acts like a variance reduction technique, yet a network with pre-training has a lower training error on a very large data set, which supports an optimization interpretation of the effect of pre-training.

How do we make sense of all these results? The contradiction between what looks like regularization effects and what looks like optimization effects appears, on the surface, unresolved. Instead of sticking to these labels, we attempted to draw a hypothesis, described in Section 6.3 about the dynamics of learning in an architecture that is trained using two phases (unsupervised pre-training and supervised fine-tuning), which we believe to be consistent with all the above results.

This hypothesis suggests that there are consequences of the non-convexity of the supervised objective function, which we observed in various ways throughout our experiments. One of these consequences is that early examples have a big influence on the outcome of training and this is one of the reasons why in a large-scale setting the influence of unsupervised pre-training is still present. Throughout this paper, we have delved on the idea that the basin of attraction induced by the early examples (in conjunction with unsupervised pre-training) is, for all practical purposes, a basin from which supervised training does not escape.

This effect can be observed from the various visualizations and performance evaluations that we made. *Unsupervised pre-training, as a regularizer that only influences the starting point of supervised training, has an effect that, contrary to classical regularizers, does not disappear with more data* (at least as far as we can see from our results). Basically, unsupervised pre-training favors hidden units that compute

features of the input X that correspond to major factors of variation in the true $P(X)$. Assuming that some of these are near features useful at predicting variations in Y , unsupervised pre-training sets up the parameters near a solution of low predictive generalization error.

One of the main messages that our results imply is that the optimization of a non-convex objective function with stochastic gradient descent presents challenges for analysis, especially in a regime with large amounts of data. Our analysis so far shows that it is possible for networks that are trained in such a regime to be influenced more by early examples. This can pose problems in scenarios where we would like our networks to be able to capture more of the information in later examples, that is, when training from very large data sets and trying to capture a lot of information from them.

One interesting realization is that with a small training set, we do not usually put a lot of importance on minimizing the training error, because overfitting is a major issue; the training error is not a good way to distinguish between the generalization performance of two models. In that setting, unsupervised pre-training helps to find apparent local minima that have better generalization error. With a large training set, as we saw in Figure 6.12, the empirical and true distributions converge. In such a scenario, *finding a better apparent local minimum will matter and stronger (better) optimization strategies should have a significant impact on generalization when the training set is very large*. Note also that it would be interesting to extend our experimental techniques to the problem of training deep auto-encoders (with a bottleneck), where previous results (Hinton and Salakhutdinov, 2006) show that not only test error but also training error is greatly reduced by unsupervised pre-training, which is a strong indicator of an optimization effect. We hypothesize that the presence of the bottleneck is a crucial element that distinguishes the deep auto-encoders from the deep classifiers studied here.

In spite of months of CPU time on a cluster devoted to the experiments described here (which is orders of magnitude more than most previous work in this area), more could certainly be done to better understand these effects. Our original goal was to have well-controlled experiments with well understood data sets. It was not to advance a particular algorithm but rather to try to better understand a phenomenon that has been well documented elsewhere. Nonetheless, our results are limited by the data sets used and it is plausible that different conclusions could be drawn, should the same experiments be carried out on other data.

Our results suggest that optimization in deep networks is a compli-

cated problem that is influenced in great part by the early examples during training. Future work should clarify this hypothesis. If it is true and we want our learners to capture really complicated distributions from very large training sets, it may mean that we should consider learning algorithms that reduce the effect of the early examples, allowing parameters to escape from the attractors in which current learning dynamics get stuck.

The observations reported here suggest more detailed explanations than those already discussed, which could be tested in future work. We hypothesize that the factors of variation present in the input distribution are disentangled more and more as we go from the input layer to higher-levels of the feature hierarchy. This is coherent with observations of increasing invariance to geometric transformations in DBNs trained on images (Goodfellow *et al.*, 2009), as well as by visualizing the variations in input images generated by sampling from the model (Hinton, 2007; Susskind *et al.*, 2008), or when considering the preferred input associated with different units at different depths (Lee *et al.*, 2009; Erhan *et al.*, 2010). As a result, during early stages of learning, the upper layers (those that typically learn quickly) would have access to a more robust representation of the input and are less likely to be hindered by the entangling of factors variations present in the input. If this disentangling hypothesis is correct, it would help to explain how unsupervised pre-training can address the chicken-and-egg issue explained in Section 6.2: the lower layers of a supervised deep architecture need the upper layers to define what they should extract, and vice-versa. Instead, the lower layers can extract robust and disentangled representations of the factors of variation and the upper layers select and combine the appropriate factors (sometimes not all at the top hidden layer). Note that as factors of variation are disentangled, it could also happen that some of them are not propagated upward (before fine-tuning), because RBMs do not try to represent in their hidden layer input bits that are independent.

To further explain why smaller hidden layers yield worse performance with pre-training than without (Figure 6.9), one may hypothesize further that, for some data sets, the leading factors of variation present in $P(X)$ (presumably the only ones captured in a smaller layer) are less predictive of Y than random projections* can be, precisely because of the hypothesized disentangling effect. With enough hidden units, unsupervised pre-training may extract among the larger set of learned features some that are highly predictive of Y (more so than random projections). This additional hypothesis could be tested by

*. Meaning the random initialization of hidden layers.

measuring the mutual information between each hidden unit and the object categories (as done by Lee *et al.*, 2009), as the number of hidden units is varied (like in Figure 6.9). It is expected that the unit with the most mutual information will be less informative with pre-training when the number of hidden units is too small, and more informative with pre-training when the number of hidden units is large enough.

Under the hypothesis that we have proposed in Section 6.3, the following result is unaccounted for: in Figure 6.8(a), training error is lower with pre-training when there is only one hidden layer, but worse with more layers. This may be explained by the following additional hypothesis. Although each layer extracts information about Y in some of its features, it is not guaranteed that all of that information is preserved when moving to higher layers. One may suspect this in particular for RBMs, which would not encode in their hidden layer any input bits that would be marginally independent of the others, because these bits would be explained by the visible biases: perfect disentangling of Y from the other factors of variation in X may yield marginally independent bits about Y . Although supervised fine-tuning should help to bubble up that information towards the output layer, it might be more difficult to do so for deeper networks, explaining the above-stated feature of Figure 6.8. Instead, in the case of a single hidden layer, less information about Y would have been dropped (if at all), making the job of the supervised output layer easier. This is consistent with earlier results (Larochelle *et al.*, 2009) showing that for several data sets supervised fine-tuning significantly improves classification error, when the output layer only takes input from the top hidden layer. This hypothesis is also consistent with the observation made here (Figure 6.1) that unsupervised pre-training actually does not help (and can hurt) for too deep networks.

In addition to exploring the above hypotheses, future work should include an investigation of the connection between the results presented in this paper and by Hinton and Salakhutdinov (2006), where it seems to be hard to obtain a good training reconstruction error with deep auto-encoders (in an unsupervised setting) without performing pre-training. Other avenues for future work include the analysis and understanding of deep semi-supervised techniques where one does not separate between the pre-training phase and the supervised phase, such as work by Weston *et al.* (2008) and Larochelle and Bengio (2008). Such algorithms fall more squarely into the realm of semi-supervised methods. We expect that analyses similar to the ones we performed would be potentially harder, but perhaps revealing as well.

Many open questions remain towards understanding and improving

deep architectures. Our conviction is that devising improved strategies for learning in deep architectures requires a more profound understanding of the difficulties that we face with them. This work helps with such understanding via extensive simulations and puts forward a hypothesis explaining the mechanisms behind unsupervised pre-training, which is well supported by our results.

Acknowledgements

This research was supported by funding from NSERC, MITACS, FQRNT, and the Canada Research Chairs. The authors also would like to thank the editor and reviewers, as well as Fernando Pereira for their helpful comments and suggestions.

7

Presentation of the third article

7.1 Article details

Dumitru Erhan, Aaron Courville, and Yoshua Bengio. *Understanding Representations Learned by Deep Architectures*. Under review at *Neural Computation*, October 2010.

Note on my personal contribution: I participated at the design and analysis of the experiments, ran all the experiments, and wrote major parts of the article.

7.2 Context

A lot of effort has been put into empirical comparisons of the variety of pre-training strategies for deep architectures. Another way of analyzing a specific instance of a deep architecture is by visualizing the parameters of the network. If the input is interpretable visually (images, spectrograms etc.), the linear parameters of a first layer can be visualized in the input space. Such visualizations (called “filters”) can reveal the features that units from the first layer have captured and could be used for comparing strategies for training them.

Rooted in neuroscience, where one is typically interested in finding the response of a unit given a range of inputs, this technique has been successfully used in a variety of Machine Learning settings. For instance, Olshausen and Field (1996) show that their sparse coding approach to modelling natural images recovers Gabor-like filters. In the context of deep architectures, Osindero and Hinton (2008), Lee *et al.* (2008), and Vincent *et al.* (2008) use such visualizations to differentiate the effects of their proposed unsupervised pre-training algorithm from previous work.

Assuming that deep structures are indeed more appropriate for modelling data and functions with many variations, it would certainly be interesting to try to qualitatively compare models based on the second

(and higher) layers of the architecture, and not only the first. Unfortunately, unlike the first-layer case (where, for a given unit, its activation is linear in the input), finding the filter does not have an easy solution. The function represented by each of these units is non-convex and it is not at all clear that one can even represent each of these units as one single filter.

7.3 Contributions

The contribution of this paper can be summarized as describing and introducing new ways of visualizing the functions learned by deep architectures. The approach that we took was one of visualizing in the input space, to obtain filter-like responses for each of the units from the deep layers.

The introduced methods are that of sampling by clamping and activation maximization. The sampling method is mostly useful in a DBN, where we can “clamp” the given hidden unit to a certain value (usually 1) and start an MCMC procedure that samples the input units. This defines a distribution of inputs which characterizes the filter. Activation Maximization is the idea that a unit can be characterized by the input to which this unit responds maximally. This idea can be cast as an optimization problem which can be solved by gradient descent with constraints. We have also analyzed a previously used method for visualizing hidden units, by Lee *et al.* (2008), which used linear combinations of first layer filters for visualization. Our work has also uncovered connections between this methods, both empirically and formally.

The methods described produce interesting and complementary visualizations. A surprising aspect of our results is that activation maximization consistently finds local maxima that appear equivalent, even though in theory the function represented by a single hidden unit in a deep layer is fraught with local minima. The filter-like representations also look interpretable and they provide some evidence to the hypothesis that deeper layers learn more complicated features of the input.

The second part of this paper extends the activation maximization method to answer the following questions: can we characterize a hidden unit by more than simply the input to which it responds maximally? Can we find a manifold of filter-like inputs that would correspond to the invariances that this unit describes? It turns out that extending activation maximization to handle these questions is once again a simple optimization problem. These invariance manifolds make it possible

for us to check that the deep architectures under consideration have indeed learned interesting features of the input. The method also enables us to define a way of measuring invariance that does not depend on hand-specifying a list of invariances. We confirm previous work of Goodfellow *et al.* (2009) who observed that deeper layers are more invariant than the layers closer to the input.

7.4 Comments

This work was an inquiry into how to qualitatively assess the class of functions that is represented by deep architectures with and without pre-training. Describing this inductive bias or class of functions is important and is complementary to the approach taken in Chapter 4, where we postulated that pre-trained deep architectures are better than other models at modelling datasets with certain characteristics: here, we want to characterize invariances or manifolds in the data space, represented by units from a deep architecture, which is trained on a specific dataset.

This work is only a step in the direction of trying to understand the invariances and representations. A few directions that would be interesting to pursue are that of characterizing more specifically the invariance manifold and of defining a clear invariance metric based on the invariance curves presented in the paper. Hopefully, these two tasks can be done at the same time, so that we can make statements that enable us to decompose the invariances represented by a filter or a layer and that make it possible for us to gain more qualitative insight into deep architectures.

Understanding Representations Learned in Deep Architectures

DEEP ARCHITECTURES have demonstrated state-of-the-art performance in a variety of settings, especially with vision datasets. Deep learning algorithms are based on learning several levels of *representation* of the input. Beyond test-set performance, there is a need for *qualitative* comparisons of the solutions learned by various deep architectures, focused on those learned representations. One of the goals of our research is to improve tools for finding good qualitative interpretations of high level features learned by such models. We also seek to gain insight into the invariances learned by deep networks. To this end, we contrast and compare several techniques for finding such interpretations. We applied our techniques on Stacked Denoising Auto-Encoders and Deep Belief Networks, trained on several vision datasets. We show that consistent filter-like interpretation is possible and simple to accomplish at the unit level. The tools developed make it possible to analyze deep models in more depth and accomplish the tracing of *invariance manifolds* for each of the hidden units. We hope that such techniques will allow researchers in deep architectures to understand more of how and why deep architectures work.

8.1 Introduction

Until 2006, it was not known how to efficiently learn deep hierarchies of features with a densely-connected neural network of many layers. The breakthrough, by Hinton *et al.* (2006), came with the realization that unsupervised models such as Restricted Boltzmann Machines (RBMs) can be used to initialize the network in a region of the parameter space that makes it easier to subsequently find good minima of the supervised objective, i.e., which give good generalization error. The greedy, layer-wise unsupervised initialization of a network can also be carried out by using auto-associators and related models (Bengio *et al.*, 2007; Ranzato *et al.*, 2007). Recently, there has been a surge in research on training deep architectures: Bengio (2009) gives a comprehensive review.

There exists a flurry of ideas on how pre-training should be done, how to better train deep models and how to, in general, learn better hierarchical representations of data. There has also been some progress in better understanding the effect of unsupervised pre-training and its role as a regularizer (Erhan *et al.*, 2010). And while *quantitative* analyses and comparisons of various strategies, models and techniques exist, and visualizations of the first layer representations are common in the literature, one area where more work needs to be done is the *qualitative* analysis of representations learned beyond the first level. Qualitative analysis would bring us insights into the models used, and would allow us to compare them beyond merely measuring performance on a held-out dataset.

We want to understand what the models have learned: what features of the data models have captured and which ones they have not. Answers to that question would help tackle issues that are potentially difficult to address with a purely quantitative approach. For instance, what is the difference between the representations learned by a Deep Belief Network (DBN) and a Stacked Denoising Auto-Encoder (SDAE), when both models perform similarly on the same test set? It would also be helpful in providing evidence to support the hypothesis that deep representations are capturing and disentangling interesting features of the data.

To better understand what models learn, we set as an aim the exploration of ways to visualize what a unit activates in an *arbitrary layer* of a deep network. The goal is to have this visualization in the *input space* (of images), while remaining computationally efficient, and to make it as general as possible (in the sense of it being applicable to a large class of neural-network-like models).

For a first layer unit, given its quasi-linear response (ignoring the sigmoidal nonlinearity), a typical visualization is simply showing in the input space (e.g. as an image) the input weights of the unit, also called the filters or “receptive fields”. This is particularly convenient when the inputs are images or waveforms, which can be visually interpreted by humans. Often, these filters take the shape of stroke detectors, when trained on digit data, or edge detectors (Gabor filters) when trained on natural image patches (Hinton *et al.*, 2006; Osindero and Hinton, 2008; Larochelle *et al.*, 2009).

For higher-level (deeper) layers, one could approach the problem from a few different angles. One approach is to devise sampling techniques. For instance, Deep Belief Nets by Hinton *et al.* (2006) have an associated generative procedure, and one could potentially use such a procedure to gain insight into what an individual hidden unit repre-

sents; we introduce such an approach in this work. Note that methods that rely on sampling will likely produce output similar to examples from the training distribution and one might need to further process the samples in order to get a picture of what the unit represents. A second approach, introduced in this paper, is inspired by the idea of maximizing the response of a given unit. One of the experimental findings of this investigation is quite surprising: despite its limitations (local minima), this method was able to find coherent filter-like representations for deeper units. A third approach, by Lee *et al.* (2008), produces a filter-like representation for deeper units from a linear combination of lower-level filters. Our results appear consistent across various datasets and techniques.

In this paper, we compare and contrast these techniques qualitatively on several image datasets, and we also explore connections between all of them. Even if we obtain a “filter”-like representation of a unit from a deep layer, it does not tell us the whole picture, because of the nonlinear relationship between the input and the unit response. One way of getting more insight into such a nonlinear unit is by testing its invariance against a specific set of variations in the input, e.g. rotations (Goodfellow *et al.*, 2009). We argue in this paper that it is useful to seek a *set* of invariances, or *invariance manifolds* for each of these units. In particular, we explore a general method that is not tied to a specific list of invariances. Such an invariance analysis could be a way to gain more insight into what the units of those layers capture. A contribution of this paper is the introduction of a few general tools that make the feature and invariance analysis of deeper layers possible.

8.2 Previous work

We briefly go over previous attempts at solving the visualization and invariance problem, in contexts similar to ours.

8.2.1 Linear combination of previous units

Lee *et al.* (2008) showed one way of visualizing the activation pattern of units in the second hidden layer of a Deep Belief Network (Hinton *et al.*, 2006). They made the assumption that a unit can be characterized by the filters of the previous layer to which it is most strongly connected*. By taking a weighted linear combination of the previous

*. i.e. whose weight to the upper unit is large in magnitude

layer filters—where the weight of the filters is its weight to the unit considered—they show that a Deep Belief Network, trained on natural images, will tend to learn “corner detectors” at the second layer. Lee *et al.* (2009) used an extended version of this method for visualizing units of the third layer: by simply weighing the “filters” found at the second layer by their connections to the third layer, and choosing again the largest weights.

Such a technique is simple and efficient. One disadvantage is that it is not clear how to automatically choose the appropriate number of filters to keep at each layer. Moreover, by selecting only the very few most strongly connected filters from the first layer, one can potentially get a misleading picture when there is not a small group of large weights but rather many smaller and similar-magnitude weights into a unit. Finally, this method also bypasses the nonlinearities between layers, which may be an important part of the model. One motivation for this paper is to validate whether the patterns obtained by Lee *et al.* (2008) are similar to those obtained by the other methods explored here.

8.2.2 Output unit sampling

Consider a Deep Belief Network with several layers. A typical scenario is where the top layer is an RBM that sees as its visible input a concatenation of the representation produced by lower levels and a one-hot vector indicating the class label. In that case, one can “clamp” the label vector to a particular configuration and sample from a particular class distribution $p(\mathbf{x}|\text{class} = k)$. Such a procedure, first described by Hinton *et al.* (2006), makes it possible to “visualize” output units, as distributions in the input space. As described in section 8.4.1, such a procedure can be extended to an arbitrary unit in the network.

It is sometimes difficult to obtain samples that cover well the modes of a Boltzmann Machine or RBM distribution, and these sampling-based visualizations cannot be applied to other deep architectures such as those based on auto-encoders (Bengio *et al.*, 2007; Ranzato *et al.*, 2007; Larochelle *et al.*, 2007; Ranzato *et al.*, 2008; Vincent *et al.*, 2008) or on semi-supervised learning of similarity-preserving embeddings at each level (Weston *et al.*, 2008). Moreover, sampling produces a *distribution* for each unit: figuring out relevant statistics of that distribution (e.g., the modes) is potentially not straightforward.

8.2.3 Optimal stimulus analysis for quadratic forms

Berkes and Wiskott (2006) start with an idea, inspired by neuro-

physiological experiments, of computing the optimal excitatory (and inhibitory) stimulus, in the for quadratic functions of the input, which are learned using Slow Feature Analysis (SFA). The limitation to quadratic forms of the input makes it possible to find the optimal stimulus, i.e. the one maximizing the activation, relatively easily.

Berkes and Wiskott (2006) also consider an invariance analysis of the optimal stimulus, whereby one finds transformations of the input to which the quadratic form is most insensitive. This method of finding invariance is using the geodetic path, meaning the path along a sphere (norm constraint, in this case), which has the smallest “acceleration”^{*} as possible.

These ideas are the closest in spirit to the work that we introduce in this paper, related to maximizing the response of a given unit. The key differences, on which we elaborate in section 8.5, are that we consider general nonlinear functions of the input (and not just quadratic forms) and our invariance analysis is a more direct and more non-local application of the idea that the directions of invariance should be the ones in which the function value (activation) drops least for such general nonlinear functions.

8.3 The models

For our analysis, we shall consider two deep architectures as representatives of two families of models encountered in the deep learning literature. The first model is a Deep Belief Net (DBN) (Hinton *et al.*, 2006), obtained by training and stacking three layers of Restricted Boltzmann Machines (RBM) in a greedy manner. This means that we trained an RBM with Contrastive Divergence (Hinton, 2002), we fixed the parameters of this RBM, and then trained another RBM to model the hidden layer representations of the first level RBM. This process can be repeated to yield a deep architecture that is an unsupervised model of the training distribution, a generative model of the data from which one can easily obtain samples from a trained model. DBNs have been described numerous times in the literature, please refer to Bengio (2009) and Hinton *et al.* (2006) for further details.

The second model, introduced by Vincent *et al.* (2008), is the so-called Stacked Denoising Auto-Encoder (SDAE). It borrows the greedy principle from DBNs, but uses denoising auto-encoders as a building block for unsupervised modelling. An auto-encoder learns an encoder

*. of the considered function, in this case the activation function.

$h(\cdot)$ and a decoder $g(\cdot)$ whose composition approaches the identity for examples in the training set, i.e., $g(h(\mathbf{x})) \approx \mathbf{x}$ for \mathbf{x} in the training set. The *denoising auto-encoder* is a stochastic variant of the ordinary auto-encoder, which is explicitly trained to denoise a corrupted version of its input. It has been shown on an array of datasets to perform significantly better than ordinary auto-encoders and similarly or better than RBMs when stacked into a deep supervised architecture (Vincent *et al.*, 2008).

We now summarize the training algorithm of the Stacked Denoising Auto-Encoders. More details are given by Vincent *et al.* (2008). Each denoising auto-encoder operates on its inputs \mathbf{x} , either the raw inputs or the outputs of the previous layer. The denoising auto-encoder is trained to reconstruct \mathbf{x} from a stochastically corrupted (noisy) transformation of it. The representation learned by each denoising auto-encoder is the “code vector” $h(\mathbf{x})$. In our experiments $h(\mathbf{x}) = \text{sigmoid}(\mathbf{b} + W\mathbf{x})$ is an ordinary neural network layer, with hidden unit biases \mathbf{b} , weight matrix W , and $\text{sigmoid}(\mathbf{a}) = 1/(1 + \exp(-\mathbf{a}))$ (applied element-wise on a vector \mathbf{a}). Let $C(\mathbf{x})$ represent a stochastic corruption of \mathbf{x} . As done by Vincent *et al.* (2008), we randomly set $C_i(\mathbf{x}) = x_i$ or 0. A fixed-size random subset of \mathbf{x} is selected for zeroing. We have also considered a salt and pepper noise, where we select a random subset of a fixed size and set $C_i(\mathbf{x}) = \text{Bernoulli}(0.5)$. The “reconstruction” is obtained from the noisy input with $\hat{\mathbf{x}} = \text{sigmoid}(\mathbf{c} + W^T h(C(\mathbf{x})))$, using biases \mathbf{c} and the transpose of the feed-forward weights W . When training denoising auto-encoders on images, both the raw input x_i and its reconstruction \hat{x}_i for a particular pixel i can be interpreted as a Bernoulli probability for that pixel: the probability of painting the pixel as black at that location. We denote by $\text{KL}(\mathbf{x}||\hat{\mathbf{x}}) = \sum_i \text{KL}(x_i||\hat{x}_i)$ the sum of component-wise KL divergences between the Bernoulli probability distributions associated with each element of \mathbf{x} and its reconstruction probabilities $\hat{\mathbf{x}}$: $\text{KL}(\mathbf{x}||\hat{\mathbf{x}}) = -\sum_i (x_i \log \hat{x}_i + (1 - x_i) \log (1 - \hat{x}_i))$. The Bernoulli model only makes sense when the input components and their reconstruction are in $[0, 1]$; another option is to use a Gaussian model, which corresponds to a Mean Squared Error (MSE) criterion.

For each unlabelled example \mathbf{x} , a stochastic gradient estimator is then obtained by computing $\partial \text{KL}(\mathbf{x}||\hat{\mathbf{x}})/\partial \theta$ for $\theta = (\mathbf{b}, \mathbf{c}, W)$. The gradient is stochastic because of sampling the example \mathbf{x} and because of the stochastic corruption $C(\mathbf{x})$. Stochastic gradient descent $\theta \leftarrow \theta - \epsilon \cdot \partial \text{KL}(\mathbf{x}||\hat{\mathbf{x}})/\partial \theta$ is then performed with learning rate ϵ , for a fixed number of pre-training iterations.

8.4 How to obtain filter-like representations for deep units

We shall start our analysis by introducing the tools to obtain a filter-like representation for units belonging to a deep layer.

8.4.1 Sampling from a unit of a Deep Belief Network

Consider a Deep Belief Network with j layers, as described in section 8.3. In particular, layers $j - 1$ and j form an RBM from which we can sample using block Gibbs sampling, which successively samples from $p(\mathbf{h}_{j-1}|\mathbf{h}_j)$ and $p(\mathbf{h}_j|\mathbf{h}_{j-1})$, denoting by \mathbf{h}_j the binary vector of units from layer j . Along this Markov chain, we propose to “clamp” unit h_{ij} , and only this unit, to 1. We can then sample inputs \mathbf{x} by performing ancestral top-down sampling in the directed belief network going from layer $j - 1$ to the input, in the DBN; as mentioned in section 8.2.2, this procedure is similar to experiments done by Hinton *et al.* (2006) for output units. This produces a distribution that we shall denote by $p_j(\mathbf{x}|h_{ij} = 1)$ where h_{ij} is the unit that is clamped, and p_j denotes the depth- j DBN containing only the first j layers.

In essence, with this method, we use the distribution $p_j(\mathbf{x}|h_{ij} = 1)$ to characterize h_{ij} . We can characterize the unit by samples from this distribution or summarize the information by computing the expectation $E[\mathbf{x}|h_{ij} = 1]$. This method has, essentially, no hyperparameters except the number of samples that we use to estimate the expectation. It is relatively efficient provided the Markov chain at layer j mixes well, which is not always the case, unfortunately, as illustrated previously (Tieleman and Hinton, 2009; Desjardins *et al.*, 2010).

Note that this method is only applicable to models from which one can (efficiently) sample and this is a rather important restriction if one’s goal is to come up with general methods for inspecting such deep architectures; for instance, it cannot be applied architectures based on auto-encoders (Bengio *et al.*, 2007; Ranzato *et al.*, 2007; Larochelle *et al.*, 2007; Ranzato *et al.*, 2008; Vincent *et al.*, 2008) or on semi-supervised learning of similarity-preserving embeddings at each level (Weston *et al.*, 2008).

8.4.2 Maximizing the activation

We introduce a new idea: we look for input patterns of bounded norm which maximize the *activation** of a given hidden unit; since the

*. The total sum of the input to the unit from the previous layer plus its bias.

activation of a unit in the first layer is a linear function of the input, in the case of the first layer, this input pattern is proportional to the filter itself, i.e., $\mathbf{x} \cdot w$ is maximized for $\mathbf{x} \propto w$ (keeping $\|\mathbf{x}\|$ fixed).

The reasoning behind this idea is that a pattern to which the unit is responding maximally could be a good initial representation of what a unit is doing*. One simple way of doing this is to find, for a given unit, the input samples (from either the training or the test set) that give rise to the highest activation of the unit. Unfortunately, this still leaves us with the problem of choosing how many samples to keep for each unit and the problem of how to “combine” these samples. Ideally, we would like to find out what these samples have in common, i.e. to be able to synthesize a representation from them. Furthermore, it may be that only some elements of the input vector contribute to the high activation, and it may not be easy to determine the relevant elements simply by inspection.

Note that we restricted ourselves needlessly to searching for an input pattern from *the training or test sets*, or simply from the set of all valid patterns. We can take a more general view and *maximizing the activation of a unit* as an optimization problem. Let θ denote our neural network parameters (weights and biases) and let $h_{ij}(\theta, \mathbf{x})$ be the activation of a given unit i from a given layer j in the network; h_{ij} is a function of both θ and the input sample \mathbf{x} . Assuming a fixed θ (for instance, the parameters after training the network), we can formalize this approach as searching for

$$\mathbf{x}^* = \arg \max_{\mathbf{x} \text{ s.t. } \|\mathbf{x}\|=\rho} h_{ij}(\theta, \mathbf{x}).$$

This is, in general, a non-convex optimization problem. But it is a problem for which we can at least try to find a local minimum. This can be done most easily by performing simple *gradient ascent*[†] in the input space, i.e. computing the gradient of $h_{ij}(\theta, \mathbf{x})$ and moving \mathbf{x} in the direction of this gradient.

Two scenarios are possible after the optimization converges: the same (qualitative) minimum is found when starting from different random initializations or two or more local minima are found. In both cases, the unit can then be characterized by the minimum or set of minima found. In the latter case, one can either average the results, or choose the one which maximizes the activation, or display all the local minima obtained to characterize that unit.

*. This is the reasoning for visualizing first-layer filters in the input space, too: they are the inputs to which the unit responds maximally.

†. Since we are trying to *maximize* h_{ij} .

This optimization technique (which we call “activation maximization”, or AM) is applicable to any network in which we can compute the above gradients. Like any gradient descent technique, it does involve a choice of hyperparameters: in particular, the learning rate and a stopping criterion (the maximum number of gradient ascent updates, in our experiments).

8.4.3 Connections between methods

There is an interesting link between the method of maximizing the activation and the sampling method from section 8.4.1. By definition, $E[\mathbf{x}|h_{ij} = 1] = \int \mathbf{x}p_j(\mathbf{x}|h_{ij} = 1)d\mathbf{x}$. If we consider the extreme case where the distribution concentrates at \mathbf{x}^+ , $p_j(\mathbf{x}|h_{ij} = 1) \approx \delta_{\mathbf{x}^+}(\mathbf{x})$, then the expectation is $E[\mathbf{x}|h_{ij} = 1] = \mathbf{x}^+$. On the other hand, when applying the activation maximization (AM) technique to a DBN, we are approximately * looking for $\arg \max_{\mathbf{x}} p(h_{ij} = 1|\mathbf{x})$, since this probability is monotonic in the (pre-sigmoid) activation of unit h_{ij} . Using Bayes’ rule and the concentration assumption about $p(\mathbf{x}|h_{ij} = 1)$, we find that

$$p(h_{ij} = 1|\mathbf{x}) = \frac{p(\mathbf{x}|h_{ij} = 1)p(h_{ij} = 1)}{p(\mathbf{x})} = \frac{\delta_{\mathbf{x}^+}(\mathbf{x})p(h_{ij} = 1)}{p(\mathbf{x})}$$

This is zero everywhere except at \mathbf{x}^+ so under our assumption, $\arg \max_{\mathbf{x}} p(h_{ij} = 1|\mathbf{x}) = \mathbf{x}^+$.

More generally, one can show that if $p(\mathbf{x}|h_{ij} = 1)$ concentrates sufficiently around \mathbf{x}^+ compared to $p(\mathbf{x})$, then the two methods (expected value over samples vs AM) should produce very similar results. Generally speaking, it is easy to imagine how such an assumption could be untrue because of the nonlinearities involved. In fact, what we observe is that although the samples or their average may look like **training examples**, the images obtained by AM look more like **image parts**, which may be a more accurate representation of what the particular units do (by opposition to all the other units involved in the sampled patterns). This subtlety is key and it highlights the ways in which they are different and complementary.

There is also a link between the gradient updates for maximizing the activation of a unit and finding the linear combination of weights as described by Lee *et al.* (2009). Take, for instance h_{i2} , i.e. the activation of unit i from layer 2 with $h_{i2} = \mathbf{v}'\text{sigmoid}(W\mathbf{x})$, with \mathbf{v} being the unit’s

*. because of the approximate optimization and because the true posteriors are intractable for higher layers, and only approximated by the corresponding neural network unit outputs.

weights and W being the first layer weight matrix. Then $\partial h_{i2}/\partial \mathbf{x} = \mathbf{v}' \text{diag}(\text{sigmoid}(W\mathbf{x}) * (\mathbf{1} - \text{sigmoid}(W\mathbf{x})))W$, where $*$ is the element-wise multiplication, diag is the operator that creates a diagonal matrix from a vector, and $\mathbf{1}$ is a vector filled with ones. If the units of the first layer do not saturate, then $\partial h_{i2}/\partial \mathbf{x}$ points roughly in the direction of $\mathbf{v}'W$, which can be approximated by taking the terms with the largest absolute value of \mathbf{v}_i .

8.4.4 First investigations into visualizing upper layer units

We shall begin with an investigation into the feasibility of using these methods for our stated purpose (obtaining informative filter-like representations). In the course of these experiments, we will also be able to compare these methods and observe their relative merits in action. More importantly, these experiments will build a basis for our explorations of invariance manifolds in the latter sections.

We used three datasets to validate our hypotheses:

- An extended version of the MNIST digit classification dataset, by Loosli *et al.* (2007), in which elastic deformations of digits are generated stochastically. We used 2.5 million examples as training data, where each example is a 28×28 gray-scale image.
- A collection of 100,000 greyscale 12×12 patches of natural images, generated from the collection of whitened natural image patches by Olshausen and Field (1996).
- *Caltech Silhouettes*, a simplified version of the Caltech-101 dataset (Fei-Fei *et al.*, 2004), in which the shape of the target object was extracted and the entire image was binarized into a foreground and a background (Marlin *et al.*, 2009). The dataset contains approximately 4,100 images of size 28×28 from 101 categories, with at least 20 and at most 100 examples from each class*.

The visualization procedures were tested on the models described in section 8.3: Deep Belief Nets (DBNs) and Stacked Denoising Auto-Encoders (SDAE). The hyperparameters are: unsupervised and supervised learning rates, number of hidden units per layer, and the amount of noise in the case of SDAE; they were chosen to minimize the classification error on MNIST and Caltech Silhouettes, respectively[†] or the

*. The data can be downloaded from <http://people.cs.ubc.ca/~bmarlin/data/index.shtml>

†. We are choosing our hyperparameters based on the supervised objective. This objective is computed by using the unsupervised networks as initial parameters for supervised backpropagation. We chose to select the hyperparameters based on the classification error because for this problem we do have an objective criterion for

reconstruction error* on natural images, for a given validation set. For MNIST and Caltech Silhouettes, we show the results obtained after unsupervised training only; this allows us to compare all the methods (since it does not make sense to sample from a DBN after the supervised fine-tuning with backpropagation stage). For the SDAE applied on natural images, we used salt and pepper noise as a corruption technique, as opposed to the zero-masking noise described by Vincent *et al.* (2008): such symmetric noise seems to work better with natural images. For the DBN we used a Gaussian input layer when modelling natural images; these are more appropriate than the standard Bernoulli units, given the distribution of pixel grey levels in such patches (Bengio *et al.*, 2007; Larochelle *et al.*, 2009).

In the case of AM (section 8.4.2, Activation Maximization), the procedure is as follows for a given unit from either the second or the third layer: we initialize \mathbf{x} to a vector of 28×28 or 12×12 dimensions in which each pixel is sampled independently from a uniform over $[0; 1]$. We then compute the gradient of the activation of the unit w.r.t. \mathbf{x} and make a step in the gradient direction. The gradient updates are continued until convergence, i.e. until the activation does not increase faster than a threshold rate. Note that after each gradient update, the current estimate of \mathbf{x}^* is re-normalized to the average norm of examples from the respective dataset[†]. There is no constraint that the resulting values in \mathbf{x}^* be in the domain of the training/test set values. For instance, we experimented with making sure that the values of \mathbf{x}^* are in $[0; 1]$ (for MNIST), but this produced worse results. On the other hand, the goal is to find a “filter”-like result and a constraint that this “filter” is strictly in the same domain as the input image may not be necessary. Finally, the same optimal value (i.e. the one that seems to maximize activation) for the learning rate of the gradient ascent works for all the units from the same layer.

Sampling from a DBN is done as described in section 8.4.1, by running the randomly-initialized Markov chain and top-down sampling every 100 Gibbs steps. In the case of the method described in sec-

comparing networks, which is not the case for the natural image data.

*. For RBMs, the reconstruction error is obtained by treating the RBM as an auto-encoder and computing a deterministic value using either the KL divergence or the MSE, as appropriate. The reconstruction error of the first layer RBM is used for model selection.

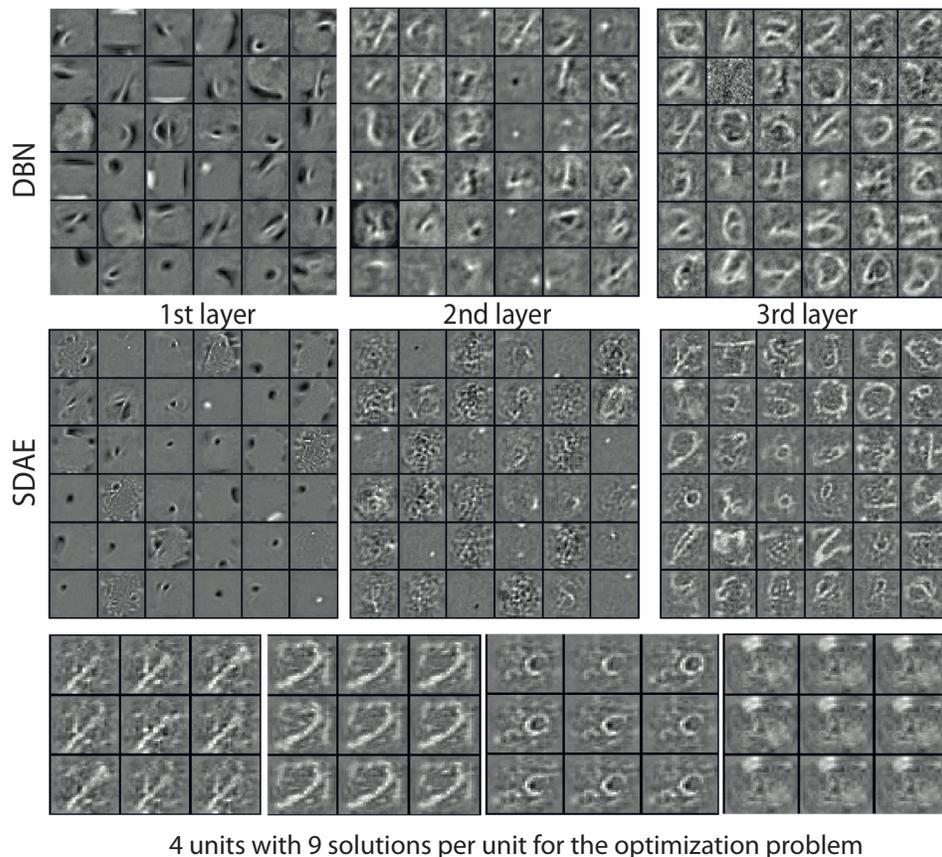
†. Such a procedure is essentially a stochastic gradient method with projection to the constraint at each step. It is possible to use better and more complicated optimization methods—such as conjugate gradient—but this adds unnecessary complexity (because of the constraint) and, in our experiments, did not lead to different conclusions.

tion 8.2.1, the (subjective) optimal number of previous layer filters was taken to be 100.

Activation Maximization

We begin by the analysis of the **activation maximization** method (AM). Figures 8.1 and 8.2 contain the results of the optimization of units from the 2nd and 3rd layers of a DBN and an SDAE, along with the first layer filters. Figure 8.1 shows such an analysis for MNIST, while Figure 8.2 shows it for the natural image data and Caltech Silhouettes.

► **Figure 8.1.** Activation maximization (AM) applied on MNIST. **First two rows:** visualization of 36 units from the first (1st column), second (2nd column) and third (3rd column) hidden layers of a DBN (top) and SDAE (middle), using the technique of maximizing the activation of the hidden unit. **Bottom row:** 4 examples of the solutions to the optimization problem for units in the 3rd layer of the SDAE, from 9 random initializations.



To test the dependence of this gradient ascent on the initial conditions, 9 different random initializations were tried*. The retained “filter” corresponding to each unit is the one (out of the 9 random initializations) which maximizes the activation. In the same figures we

*. If one chooses a training set examples as a starting point, the same types of local minima are found. Our goal was to show that this method is robust even when initialized with examples that are very far from the training set.

also show the variations found by the different random initializations for a given unit from the 3rd layer. **Surprisingly, most random initializations yield roughly the same prominent input pattern.** Moreover, we measured the maximum values for the activation function to be quite close to each other (not shown). Such results are relatively surprising, given that, generally speaking, the activation function of a third layer unit is a highly non-convex function of its input. Therefore, either we are consistently lucky, or we are not sampling from the whole space, or, at least in these particular cases (a network trained on MNIST digits, Caltech Silhouettes, or natural images), the activation functions of the units tend to be more “unimodal”.

One important point is that, qualitatively speaking, the filters at the 3rd layer look interpretable and quite complex. For MNIST, some look like pseudo-digits. In the case of natural images, we can observe grating filters at the second layer of DBNs and complicated units that detect, for instance, corners at the second and third layer of SDAE; some of the units have the same characteristics that we would associate with V2-area units (Lee *et al.*, 2008). For Caltech Silhouettes, a few of the units look like whole-object class detectors (faces, for instance), but most seem to simply encode for the presence or absence of parts of objects (likely meaning that the third layer units have managed to learn a decomposition of the input space features that is not as simple as just whole-object-class detection). Such results also suggest that higher level units do indeed learn *meaningful* combinations of lower level features.

Note that the first layer filters obtained by the SDAE when trained on natural images are Gabor-like features. It is interesting that in the case of the DBN, the filters that minimized the reconstruction error*, i.e. those that are pictured in Figure 8.2 (top-left corner), do not have the same low-frequency and sparsity properties like the ones found by the first-level denoising auto-encoder†. Yet at the second layer **the filters found by activation maximization are a mixture of Gabor-like features and grating filters.** This shows that appearances can be deceiving: we might have dismissed the RBM whose weights are shown in Figure 8.2 as a bad model of natural images had we looked

*. Which is only a proxy for the actual objective function that is minimized by a stack of RBMs.

†. It *is* possible to obtain Gabor-like features with RBMs—work by Osindero and Hinton (2008) shows that—but in our case these filters were never those that minimized the reconstruction error of an RBM. This points to a larger issue: it appears that using different learning rates for Contrastive Divergence learning will induce features that are *qualitatively different*, depending on the value of the learning rate.

only at the first layer filters, but the global qualitative assessment of this model, which includes the visualization of the second and third layers, points to the fact that the 3-layer DBN is in effect learning something quite interesting. Such a result suggests that qualitative model comparison (between SDAE and DBNs in this case) cannot rely entirely on first-layer filter visualizations.

Sampling a unit

We now turn to the **sampling technique** described in section 8.4.1. Figure 8.3 shows samples obtained by clamping a second layer unit to 1; both MNIST and natural image patches are considered. In the case of natural image patches, the distributions are roughly unimodal, in that the samples are of the same pattern, for a given unit. For MNIST, the situation is slightly more delicate: there seem to be one or two modes for each unit*. The *average* input (the expectation of the distribution), as seen in Figure 8.4.4, then looks like a digit or a superposition of two digits.

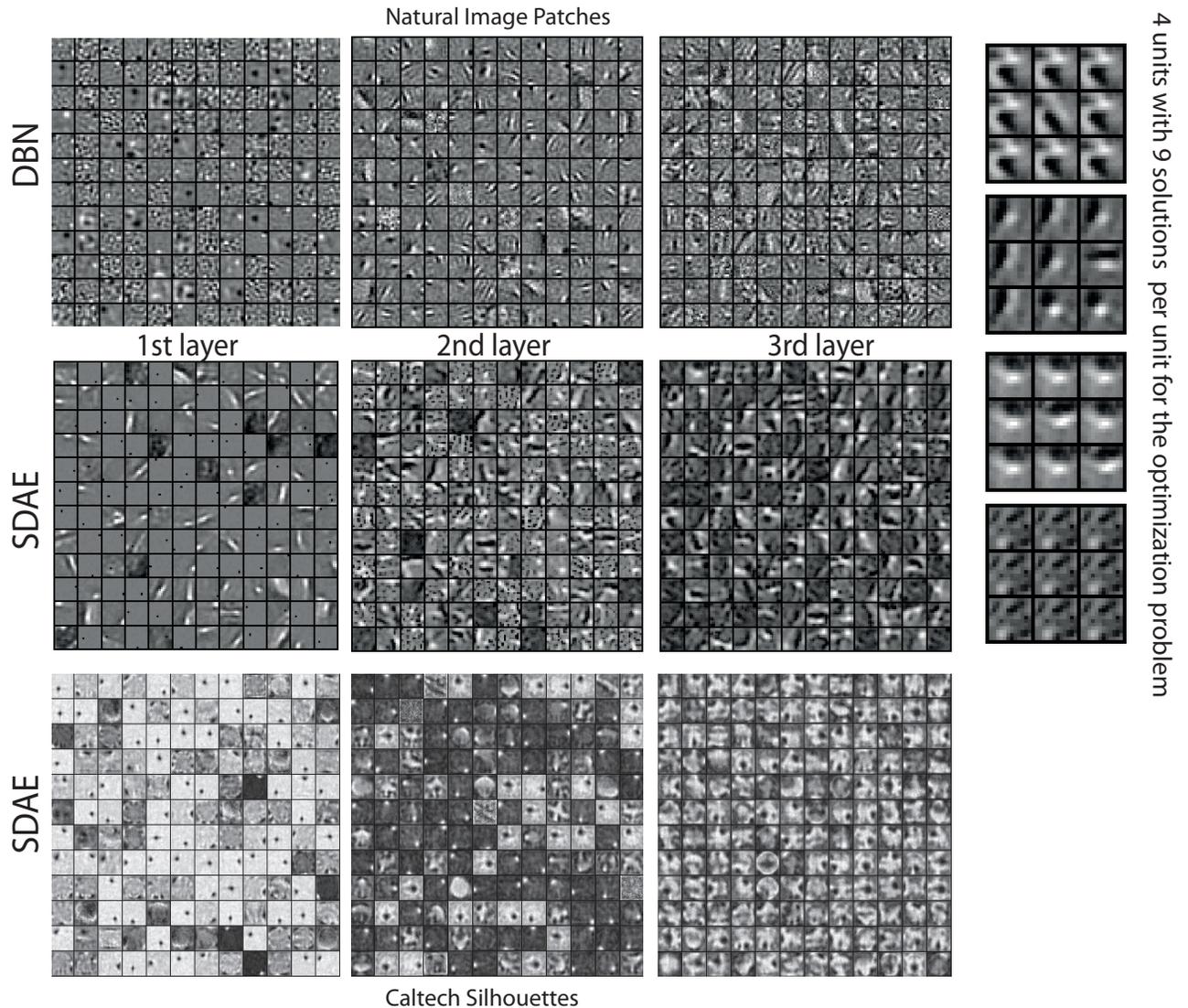
Note that unlike the results of AM, the samples are much more likely to be part of the underlying distribution of examples (digits or patches). AM seems to produce *features* and it is up to us to decide which examples would “fit” these features; the sampling method produces *examples* and it leaves it to us decide which features these examples have in common. In this respect, the two techniques serve complementary purposes.

Comparison of methods

In Figure 8.4.4, we can see a comparison of the three techniques: activation maximization, hidden unit sampling, and the **linear combination method**, introduced by Lee *et al.* (2008) and as described in section 8.2.1. The methods are tested on the second layer of a DBN trained on MNIST. In the above, we noted links between the three techniques. The experiments show that many of the filters found by the three methods share some features, but have some differences as well. In general, linear combination of previous layer filters and AM were quite similar, highlighting parts, whereas sampling produced full examples.

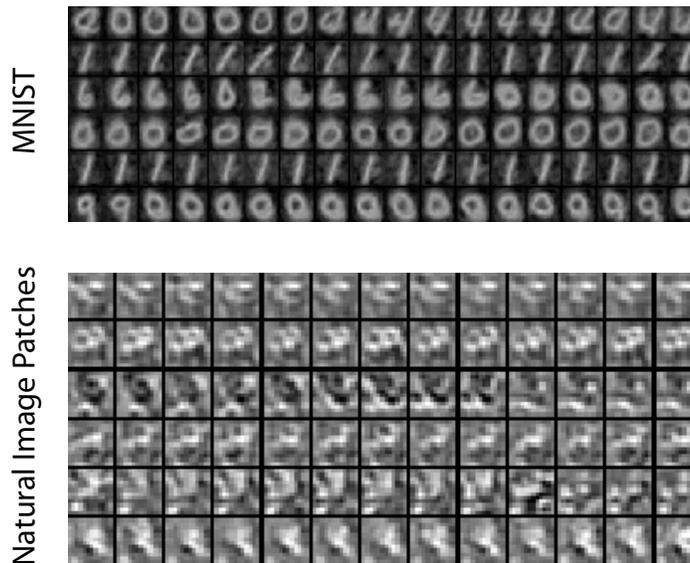
Unfortunately, we do not have an objective measure that would allow us to compare the three methods, but visually we believe that AM produces more interesting and useful results: by comparison, the

*. This result was obtained with multiple restarts and 20,000 Gibbs steps



▲ **Figure 8.2.** Activation Maximization (AM) applied on Natural Image Patches (top and middle row) and Caltech Silhouettes (bottom row). Visualization of 144 units from the first (1st column), second (2nd column) and third (3rd column) hidden layers of a DBN (top row) and an SDAE (middle and bottom rows), using the technique of maximizing the activation of the hidden unit. **In the 4th column:** 4 examples of the solutions to the optimization problem for units in the 3rd layer of the SDAE, subject to 9 random initializations, for natural images.

► **Figure 8.3.** Visualization of 6 units from the second hidden layer of a DBN trained on MNIST (**top**) and natural image patches (**bottom**). The visualizations are produced by sampling from the DBN and clamping the respective unit to 1. Each unit’s distribution is a row of samples; the mean of each row is in the first column of Figure 8.4.4 (left).

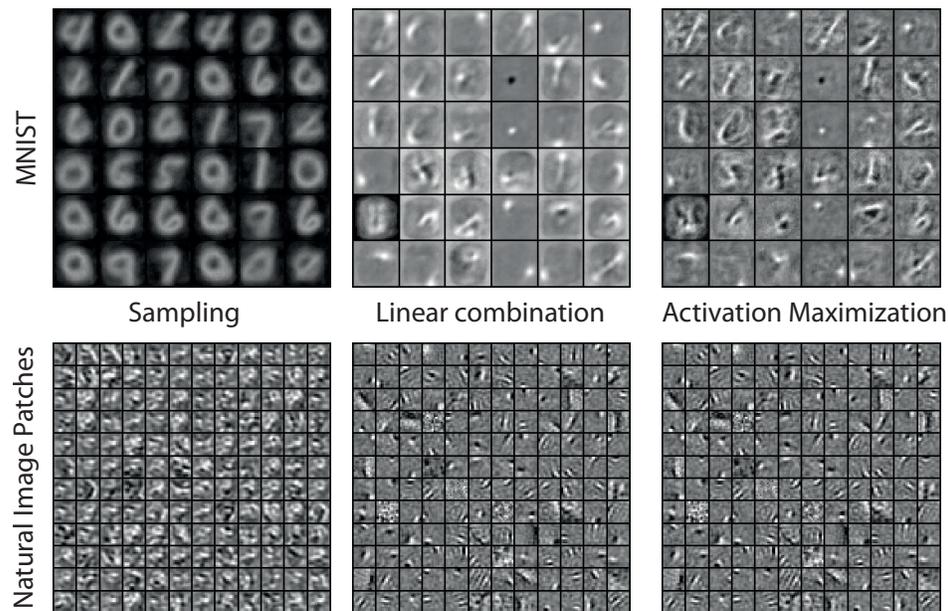


average samples from the DBN are almost always in the shape of a digit (for MNIST), while the linear combination method seems to find only parts of the features that are found by AM, which tends to find sharper patterns.

AM is applicable to a very large class of models, is conceptually simple and produces high quality visualizations. Moreover, the technique lends itself to easy, but quite powerful extensions, as we shall explore next.

8.5 Uncovering Invariance Manifolds

Thus far our goal has been to obtain a filter-like representation for each unit of the upper layers. Obtaining such filters is an interesting development and it allows us to see that upper layer units correspond to more complicated filters (sometimes even “template detectors”) and verify some hypotheses that we had about deep architectures: namely that they learn to model interesting features at higher levels, that units at those levels correspond to more complicated V2-area like units etc. However, such filter-like representations only characterize a point in the input space: they don’t really describe the invariances captured by each unit or each layer. The second part of our inquiry will address this issue.



◀ **Figure 8.4.** Visualization of 36 units from the second hidden layer of a DBN trained on MNIST (**top**) and 144 units from the second hidden layer of a DBN trained on natural image patches (**bottom**). Left: sampling with clamping, Centre: linear combination of previous layer filters, Right: maximizing the activation of the unit. Black is negative, white is positive and gray is zero.

A simple approach to solving this problem is by extending our activation maximization approach to computing some second order visualization. One way was presented in section 8.2.3, by Berkes and Wiskott (2006)*: compute geodesic paths (paths on the norm constraint / sphere), starting at the maximum of the activation function, which have the smallest rate of change. Another solution is to compute the Hessian at the local maximum and analyze the directions of principal invariance, corresponding to the eigenvectors of the Hessian with the smallest eigenvalues, by moving in the direction of those eigenvectors (starting from the optimum), while remaining on the norm sphere. For quadratic forms and in the context of Slow Feature Analysis, such an approach seemed to be fruitful (Berkes and Wiskott, 2002, 2006).

Our attempts at replicating the latter analysis in the context of AM and arbitrary units in the deep layers were not as successful: the eigenvectors point in directions that did not reveal useful insights, as far as we could tell. Our intuition is that such directions are really a very local measure around the maximum and may not be meaningful farther away from it. This locality effect is present in the geodesic path method of Berkes and Wiskott (2006), where the authors suggest that this method is only applicable in “a small neighbourhood” of the maximum. We would like a method that would trace an invariance

*. for quadratic functions of the input

manifold that corresponds to the unit, and we want this manifold to be less local (with respect to the maximum found via AM). Ideally, we would like to see what pattern of activations it is most invariant to or what manifold this unit “traces” in the input space. Finally, our intuition suggests that these directions of invariance should correspond, roughly speaking, to the changes of the optimum that produce the smallest decrease in the activation value, and we would like a more direct way of achieving this.

8.5.1 Invariance Manifolds

A simple way of achieving such goals is to start with the result given to us by AM and move as far as possible from it while keeping the activation as large as possible. Formally, let \mathbf{x}_{opt} be the (best) local optimum found by AM for a given unit. Then we re-formulate our optimization problem as follows:

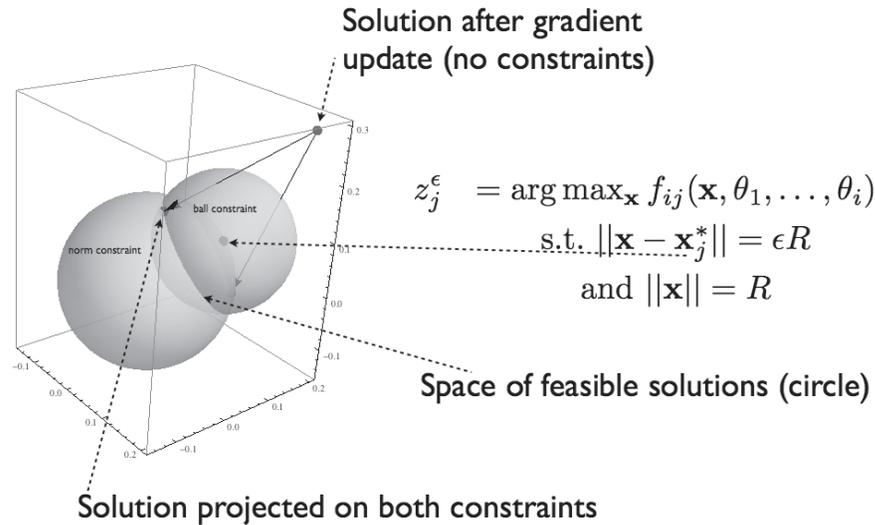
$$\mathbf{x}_\varepsilon^* = \arg \max_{\mathbf{x} \text{ s.t. } \|\mathbf{x}\|=\rho \text{ and } \|\mathbf{x}-\mathbf{x}_{opt}\|=\varepsilon\rho} h_{ij}(\theta, \mathbf{x}).$$

where $0 \leq \varepsilon \leq 2$. By varying ε we can construct a one-dimensional manifold—represented by the solutions \mathbf{x}_ε^* in increasing order of ε —that has our desired properties*.

Note that, as before, we require our solutions \mathbf{x}_ε^* to be of some fixed norm as well (ρ , as before); removing such a constraint makes the optimization problem ill-behaved (the objective function could otherwise potentially increase without bound). The optimization problem can again be solved with simple gradient descent, starting from a random point in the space of feasible solutions and projecting to the space of feasible solutions at each step; projecting exactly onto both constraints is more complicated than the simple AM (Activation Maximization) with one norm constraint, but it follows from a straightforward algebraic computation.

Figure 8.5 illustrates this process for an optimization problem in 3 dimensions. We remind the reader that for simplicity this procedure is a sequence of gradient steps followed by projection to the constraints. Note that the projection operation always has two solutions (on the opposite sides of the feasible solutions circle/hypersphere, in our case) – we always pick the one that results in the highest activation value.

*. At $\varepsilon = 2$ the two (hyper-)spheres corresponding to the two constraints intersect at exactly one point. If ε is larger than 2, then the constraint cannot be satisfied anymore, since the spheres do not intersect (one is inside the other). See Figure 8.5.



▲ **Figure 8.5.** Illustration of the invariance manifold tracing technique in 3D. \mathbf{x}_j^* is the activation maximization result for unit j , R is the average norm of our inputs, and ϵR is the distance from \mathbf{x}_j^* that we want our solutions to be. After each gradient step (towards maximizing f_{ij}), we project the current solution such that it satisfies the constraints; there are two such projections possible—for the next iteration of the optimization problem, we choose the one with the highest activation value.

As discussed in the introduction to this section, when analyzing the directions of invariance, as given to us by the eigenvectors of the Hessian at the local maximum \mathbf{x}_{opt} , we did not observe any qualitatively interesting results. Our hypothesis is that there are many local directions—corresponding roughly to changing the background—and moving in those directions will not decrease the activation of the given unit*. Such an effect can also occur with our invariance manifold technique: the optimization procedure could conceivably move \mathbf{x}_ϵ^* into directions that are of no interest to us (from a model analysis point of view)†.

A way to counteract this effect is to move only in directions where there is variance in the data or, equivalently, dampen the directions in

*. In other words, the learning procedure has managed to make units invariant to small background transformations.

†. An interesting parallel can be made with an experiment that we performed, in which, instead of Activation Maximization we *minimize* the activation for each unit. The same “background effect” was observed. This suggests that the “activation landscape” of a hidden unit is similar to a ridge, in that there are a few directions of invariance—which are not easy to find—and quite a number of directions in which we can move and decrease the activation significantly.

which there is no variance in the training data. More specifically, this can be accomplished by computing the whitening matrix W , via the zero-phase whitening (also called ZCA) transform (Bell and Sejnowski, 1997). This is the matrix which, when multiplied with $\mathbf{x} \in D_{train}$ spheres the data, i.e., $\text{Cov}(\mathbf{y}) = I$, where $\mathbf{y} = W\mathbf{x}$. Starting from $\mathbf{y}_{opt} = W\mathbf{x}_{opt}$, the search becomes:

$$\mathbf{y}_{\varepsilon}^* = \arg \max_{\mathbf{y} \text{ s.t. } \|W^{-1}\mathbf{y}\|=\rho \text{ and } \|W^{-1}(\mathbf{y}-\mathbf{y}_{opt})\|=\varepsilon\rho} h_{ij}(\theta, W^{-1}\mathbf{y}) \quad (8.1)$$

That is, scale the directions in which we move by the amount of variance that the training data exhibits in those directions. Algorithm 1 contains the details of this procedure in pseudo-code format.

Algorithm 1 Pseudo-code of the invariance computation procedure (eq. 8.1), using the whitening matrix to scale the directions in which we proceed. The $\text{projection}(\mathbf{y}_{new}, \text{constraints}(\rho, \varepsilon, \mathbf{y}_{opt}))$ operator signifies the function that projects \mathbf{y}_{new} s.t. $\|W^{-1}\mathbf{y}_{new}\| = \rho$ and $\|W^{-1}(\mathbf{y}_{new} - \mathbf{y}_{opt})\| = \varepsilon\rho$.

Require: \mathbf{x}_{opt} , W , and a learning rate μ

```

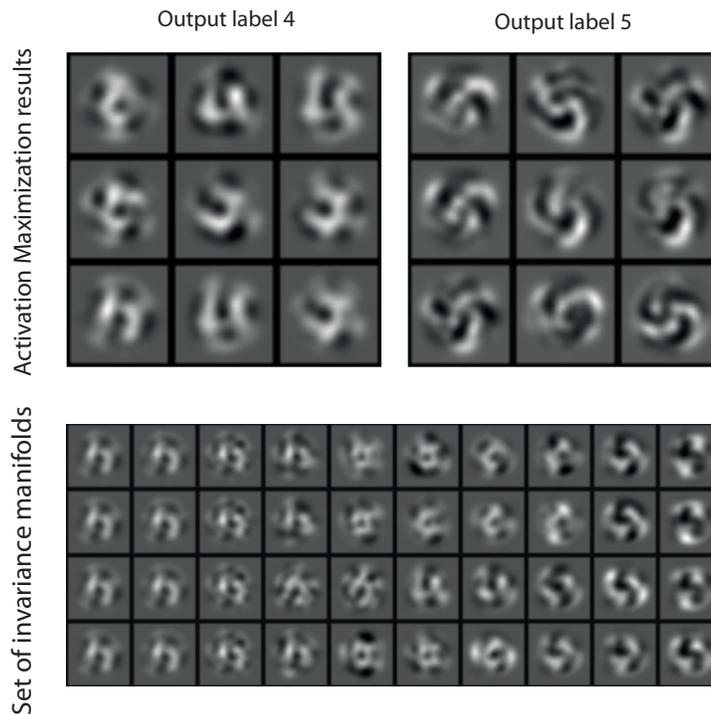
 $\mathbf{y}_{opt} = W\mathbf{x}_{opt}$ 
 $\mathbf{y}_{current} = \mathbf{y}_{opt}$ 
while not converged do
   $\mathbf{y}_{new} = \mathbf{y}_{current} + \mu \cdot \frac{\partial(h_{ij}(\theta, W^{-1}\mathbf{y}_{current}))}{\partial\mathbf{y}_{current}}$ 
   $\mathbf{y}_{current} = \text{projection}(\mathbf{y}_{new}, \text{constraints}(\rho, \varepsilon, \mathbf{y}_{opt}))$ 
end while
 $\mathbf{y}_{\varepsilon}^* = \mathbf{y}_{current}$ 
return  $\mathbf{y}_{\varepsilon}^*$ 

```

8.5.2 Results

We applied this method to a variant of the MNIST dataset, called *mnist-rot*, first presented by Larochelle *et al.* (2007). This is a dataset that contains rotated MNIST digits (random rotations, angle between $-\pi$ and π) and is being used in the community as a good check for empirically evaluating whether a given deep architecture is able to capture the rotational invariance in the data.

A sanity check for the invariance manifold technique just presented is to apply it to one of the 10 *output* units, corresponding to the predictions of the network for a given label. The hypothesis is that the



◀ **Figure 8.6.** Upper: output filter minima for the output units corresponding to digits 4 and 5 (upper). Lower: A set of invariance manifolds corresponding to digit 4, all starting from the same point (the best activation maximization result) and with a small random perturbation at the beginning of optimization; a row is one such trajectory / invariance manifold

results of the optimization technique on such units should be most interpretable (compared to other units in the network) and should be quite revealing of the invariances that are captured by the process of supervised learning.

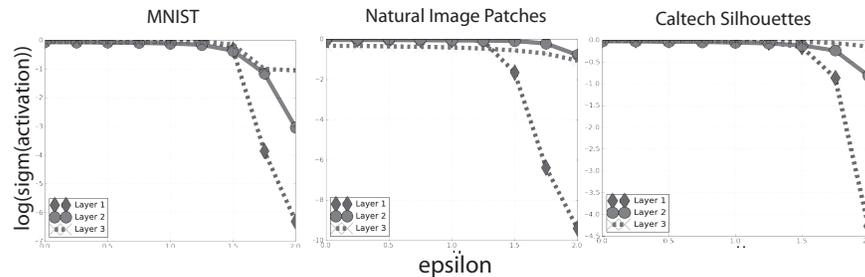
Figure 8.6 (upper) presents several runs of AM on the output units corresponding to labels 4 and 5. A key observation in this case is that \mathbf{x}_{opt} does not appear to be unimodal (as a function of random starting points). In fact, it would have been surprising otherwise: for instance, it is unlikely that the distribution of all rotated four-digits can be “captured” by a prototypical “four”. Instead, we see a variety of rotated four- and five-digits.

Figure 8.6 (lower) contains an invariance manifold analysis: we picked the \mathbf{x}_{opt} for the four-digit AM which had the highest activation value and then did four trials in which we varied the starting point of the optimization; this results in a *set of invariances* that characterize this unit. In fact, this was the only element of uncertainty in the optimization process—for a given ε we used the previous $\mathbf{x}_{\varepsilon-\delta}^*$ (meaning the solution with a slightly smaller ε) as the starting point. The startling observation is that even when only the very initial condition is changed, the invariance manifolds (from left to right on each row)

become quite different. These manifolds also seem to be interpretable, as they are capturing the various rotations that the output unit seems to be able to model.

8.5.3 Measuring invariance

Using the invariance manifold tool we can get an idea of the invariance for a given deep architecture model. Indeed, note that the activation value of a given unit j from a layer i , $h_{ij}(\mathbf{x}_\varepsilon^*)$ as one varies ε , can be considered as an indicator of invariance for a given unit: *the slower the unit's activation decreases as we increase ε the more invariant it is*. The intuition is the following: a unit whose activation drops down slowly has “carved” a manifold of the input space that is sufficiently large that even if we go far away from \mathbf{x}_{opt} we can still maintain a high level of activation. Conversely, a unit whose activation drops down very fast has carved a small region of the space is therefore only responsible for only a few variations in the input data.



▲ **Figure 8.7.** *Measuring the invariance of the units from different layers. From left to right, experiments on MNIST, Natural Image Patches and Caltech Silhouettes, with SDAE. The y-axis plots the sigmoid of the activation (in the log domain, for clarity) vs. the ε with which we move. The “speed” with which the curves decrease is what should be compared (layer 1 vs. layer 2 vs. layer 3).*

There is no established notion of a measure of invariance of a given unit in such a network. We argue however that, in a sense, our intuition can be used to reach a rather generic notion of invariance. Furthermore, to compute it, one does not need to specify a given *type* of invariance (though, as we shall see later in the discussion, this is also a limitation). This is in contrast with the work of Goodfellow *et al.* (2009), where the authors specify a series of input deformations (rotations, translations, etc) and an invariance measure that is computed for each unit.

The main hypothesis that researchers in deep architectures have is that the upper layers of the models become more invariant to input transformations, presumably because of the increased level of abstraction represented by upper layers. Using our approach, this becomes a testable hypothesis: we simply need to compute the activation $h_{ij}(\mathbf{x}_\varepsilon^*)$ of each unit as ε increases, for all the units in a given layer. Figure 8.7 contains such an analysis, for MNIST, mnist-rot and Natural Image Patches. We observe that in all cases the slope of the activation decrease (as ε increases) is *smaller for the first layer units compared to the second layer ones*; the second layer slopes for MNIST and Caltech Silhouettes are smaller than the third layer slopes as well. One could use this method to define a scalar measure of invariance, for instance from the area under the curve, which can then be used to compare models against each other. What the figure provides is new evidence to support the earlier observations of Goodfellow *et al.* (2009) that, in general, units from upper layers appear more invariant than those in the lower layer.

8.6 Conclusions and Future Work

We started from a simple desire: to better understand the solution that is learned and represented by a deep architecture, by investigating the response of individual units in the network. Like the analysis of individual neurons in the brain by neuroscientists (Dayan and Abbott, 2001, chapter 2.2), this approach has limitations, but we believe that such visualization techniques can help understand the nature of the functions learned by the network.

We describe three techniques for visualizing deep layers: activation maximization (AM) and sampling from an arbitrary unit are both new (to the best of our knowledge) and introduced in this work, while the linear combination technique had been previously introduced by Lee *et al.* (2008). We show the intuitive similarities between them and compared and contrasted them on three datasets. Our results confirm our intuitions about the hierarchical representations learned by deep architectures: namely that the higher layer units represent features that possess (meaningfully) more complicated structure and correspond to combinations of lower-layer features. The three techniques considered for visualization give rise to meaningfully different results: as posited in the introduction, we found that a sampling-based method produces a distribution of training-set-like samples, which may require further

processing to make sense of what specifically the chosen units captures. Conversely, AM (and, to a lesser extent, the linear combination method) make it possible to get a “part”-like representation of each unit, an arguably more interpretable representation.

We also find that the two deep architectures considered learn quite different features. An unexpected result (Figure 8.2) is the discovery that, for natural image patches, uninformative-looking first-layer filters of a Deep Belief Network do not necessarily tell the whole story: we show that second-layer units can model edge detectors and grating filters in the same model. The implication of this result is that higher-layer units can be an important tool for comparing models and provides a justification for seeking to understand and visualize what the upper-layer units in a deep architecture do; such a result should be interpreted in the context of the standard approach used in many papers on deep architectures (Osindero and Hinton, 2008; Larochelle *et al.*, 2009), which is that of simply looking at first-layer filters, in addition to test error performance.

Further leveraging the AM methodology, we turn to the question of exploring the invariances that are learned by individual units in a network. Again, we can cast this question as an optimization problem. We explore this in detail for the output units of a supervised network trained on rotated digits. These explorations confirm our intuitions that these manifolds essentially capture the kinds of general invariances present in the data and learned by the model. Finally, this investigation naturally provides a way to visualize and measure invariance. This experiment allowed us to compare layers in a fairly generic way (with respect to how “invariant” the average activation of a unit is, as we move away from the result of AM), without actually specifying the set of invariances by hand, or generating data in any way; as with AM, this invariance analysis is applicable to a large class of deep architectures.

The same procedures (AM and invariance analyses) can be applied to the weights obtained after *supervised* learning and the observations are similar: convergence occurs and features seem more complicated at higher layers. We have already performed a basic analysis along these lines—in Erhan *et al.* (2010), Figures 3 and 4, where we show the influence of pre-training on a deep network. However, we feel that more work is needed in order to better understand the qualitative effect of pre-training for supervised learning and visualization/invariance analysis tools could be helpful in this respect.

We would be interested in comparing with Goodfellow *et al.* (2009)’s approach of hand-crafted input transformations (such as translations, rotations etc.), and the measurements of invariance of upper-layer units

as a function of these transformations. Our belief is that analysis methods that rely on specific invariances are limited in the story they can tell us, because we would like to measure invariance to variations that are not known a priori. The method we presented in this paper is generic with respect to the input transformations and is thus a generic way of measuring invariance; in this sense, it is an interesting alternative to Goodfellow *et al.* (2009)'s approach. Nonetheless, one could reasonably question the interpretability of the invariance manifolds that our method uncovers. Would it be possible to project or decompose the manifold of a given unit to a set of known invariances? Could we group the units of the layer according to certain types of invariance?

Future research will concentrate on exploring such questions. Ideally, a future method for analysis would be able to detail, for a given unit, the level of invariance with respect to (for example) rotation, translation and scaling of the input data and provide us with an idea of how invariant it is to other transformations of the input that are not in the list. Our work is a step in such a direction. The analysis in Figure 8.5 could be extended such that the search space of the invariance manifold is limited to inputs corresponding to only rotations (or only translations or etc.) of \mathbf{x}_j^* , the AM output; by computing curves such as the ones in Figure 8.7, for each such transformation separately, one could then come up with at least a *relative* notion of invariance, meaning that we could understand whether a unit is more invariant to rotations or to translations. From there, we could compare entire layers or model instances, and we might also be able to compare the behaviour of higher level units in a deep network to features and invariances that are presumed to be encoded by the higher levels of the visual cortex Lee *et al.* (2008).

Acknowledgments

We would like to thank Pascal Vincent and Ian Goodfellow for helpful discussions.

9

Conclusion

SCIENCE is the process of building testable hypotheses about the natural world and doing experiments to validate these hypotheses, thus increasing the corpus of shared knowledge. The goal of researchers in Machine Learning is to come up with hypotheses that best explain the mechanisms via which an agent or a system can learn from data and generalize to unseen examples. These hypotheses can take the shape of models, ways of learning with them, or the choices that we have to make when we select one of these models. Collectively, we have called the process of establishing all these choices as the *inductive bias* that we, as researchers, introduce into the process.

Fundamentally, there cannot be a learning algorithm that is universally applicable and that can generalize across all possible data distributions. But just like science is concerned with testing hypotheses related to the *natural world*, we argue in this thesis that the function classes that we consider when coming up with new ML hypotheses should be such that they model best data and functions that are most interesting to us from the point of view of creating intelligent agents.

We have argued that models that have an inductive bias towards locality in the input space and a shallow representation are not sufficient for the class of functions that are of interest to us. This “AI-Set” contains functions with many degrees of variation, that are highly non-local as a function of their input, and that have high manifold curvature that requires dense input sampling for representation. The AI-Set of functions is what we expect an intelligent agent to be able to learn.

Deep architectures, defined as compositions of nonlinear processing units, were suggested as the solution to these problems. Hinton *et al.* (2006)’s breakthrough work on using stacked Restricted Boltzmann Machines for unsupervised learning to initialize the supervised learning process of a multi-layer neural network was seminal in that it made it possible to efficiently train such compositions of nonlinear processing units. The state-of-the-art results in a variety of application domains and the replication of the general principles to other unsupervised techniques and deep models popularized the field in the past few years. It is only natural therefore to try to understand the reasons for the success of these methods.

9.1 Main findings

This thesis is an inquiry into the general principles—the “recipe”—of deep architectures. Each of the presented articles is asking a question about these models and the ways used to train them, in an attempt to glean insights that will further our understanding of them.

Chapter 4 is an inquiry into how easy it is to model the class of functions that we defined as the “AI-set” (or some approximation of it) using deep architectures/ Our choice of datasets to test such hypotheses was a step beyond the standard datasets that had been used previously. Are deep architectures indeed more appropriate for modelling data with many variations? Evidence from this chapter suggests that this is indeed the case and that unsupervised pre-training is crucial in obtaining good performance on datasets with many variations.

Chapter 6 is an exploration of a central issue in deep learning: why does unsupervised pre-training work? What exactly does unsupervised pre-training change in terms of the functions that a deep architecture can model and how does it do it? Our extensive experiments support a regularization explanation. Regularization occurs by shaping the distribution of initialization points that supervised learning has access to. In this sense, we argued that unsupervised pre-training is a form of procedural inductive bias. Another important conclusion from this work is that stochastic gradient descent seems to be influenced disproportionately by examples seen early during training.

Finally, in Chapter 8 we wanted to know: can we qualitatively describe the function class that is represented by a trained deep network? Can we get an insight into them through such descriptions? We discussed a few techniques for this kind of introspection (two of them being introduced in the article) and found that we can characterize the deep architectures considered, and the functions modelled by them, at the unit level. We have also extended one of the techniques in such a way that it allows us to trace an invariance manifold in the input space, thereby providing us with an even richer description of these functions, and making it possible to define invariance curves that can be used to compare the agglomerated invariance properties of whole layers.

As discussed in each of the chapters presented this work, in the light of work published afterward, refinements to the hypotheses exposed could certainly be made. Clearly, the investigation into the reasons for the success of deep architectures and the effect of pre-training are not finished yet. The work in Chapter 4 could be extended by constructing datasets that are closer to real-life situations (larger-scale, more complicated variations). The experiments in Chapter 6 can be extended

by analyzing more recently proposed deep architectures and optimization strategies, while the experiments in Chapter 8 could benefit from further improvements to the techniques such that the results are more interpretable.

Nonetheless, we believe that the work presented by these three articles shows clearly the importance of unsupervised pre-training and the mechanisms via which depth and pre-training shape the class of functions that is being learned by neural networks considered. In this sense, this work has accomplished its goal of elucidating the inductive bias represented by deep architectures.

9.2 Speculative remarks

As of November 4, 2010, the original DBN article (Hinton *et al.*, 2006), published in 2006, has 336 citations according to Google Scholar. The field has thus grown very rapidly and the work on deep architectures seems to have quite an impact, beyond Machine Learning as well. The DBN approach has led to a quasi-renaissance of neural networks. This is an exciting time for research in neural networks, because they are very flexible and powerful models, and the work of Hinton *et al.* (2006), as well as subsequent publications, have enabled researchers to obtain state-of-the-art results consistently in a variety of settings.

The field is moving in many different directions, a lot of them with the potential for exciting new advances. For example, neural networks have been proposed (Caruana, 1997) as a natural solution for solving the multi-task learning problem*. It is plausible that having a deep representation helps—or is even necessary—in order to build a representation that allows for multi-task learning to work.

Multi-modality is another example of a direction in which the field could be moving. Coming up with a joint representation for various modalities of the training data (text, image, sound, etc.) is intuitively possible with deep architectures. Problems such as image or music annotation and description or, more generally, learning problems with rich input data could certainly benefit from such advances.

Computing power is finally catching up with the model sizes that allow us to train models on realistic input sizes. However, advances still need to be made if want deep architectures to be able to model

*. This is the principle that states that instead of learning a single task at a time, one can share model parameters and training data to learn multiple tasks at the same time.

images with millions of pixels or to train models on billions of such images. Convolutional approaches are one solution, but methods for parallelizing training or coming up with spars(er) representations should also help. Additionally, we predict that more powerful optimization methods will be explored, as models sizes will eventually become so large that one will be faced with an under-fitting issue during learning. Second-order methods such as the Hessian-free approach of Martens (2010) could impact the field quite significantly.



Bibliography

- Ackley, D. H., G. E. Hinton, and T. J. Sejnowski (1985). A Learning Algorithm for Boltzmann Machines. *Cognitive Science* 9, 147–169. (Cited on page 36.)
- Aizerman, M. A., E. M. Braverman, and L. I. Rozonoer (1964). Theoretical Foundations of the Potential Function Method in Pattern Recognition Learning. *Automation and Remote Control* 25, 821–837. (Cited on page 27.)
- Amari, S., N. Murata, K.-R. Müller, M. Finke, and H. H. Yang (1997). Asymptotic statistical theory of overtraining and cross-validation. *IEEE Transactions on Neural Networks* 8(5), 985–996. (Cited on pages 83 and 90.)
- Anderson, J. A. and M. C. Mozer (1981). Categorization and Selective Neurons. In G. E. HINTON and J. A. ANDERSON (Eds.), *Parallel Models of Associative Memory*, pp. 213–236. Hillsdale: Lawrence Erlbaum. (Cited on page 40.)
- Bach, F., G. Lanckriet, and M. Jordan (2004). Multiple kernel learning, conic duality, and the SMO algorithm. In C. E. BRODLEY (Ed.), *Proceedings of the Twenty-first International Conference on Machine Learning (ICML'04)*, pp. 6. ACM. (Cited on page 31.)
- Bahl, L., P. Brown, P. deSouza, and R. Mercer (1986). Maximum mutual information estimation of hidden Markov parameters for speech recognition. In *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, Tokyo, Japan, pp. 49–52. (Cited on page 83.)
- Barron, A. E. (1991). Complexity Regularization with Application to Artificial Neural Networks. In G. ROUSSAS (Ed.), *Nonparametric Functional Estimation and Related Topics*, pp. 561–576. Kluwer Academic Publishers. (Cited on page 83.)
- Belkin, M. and P. Niyogi (2002). Laplacian Eigenmaps and Spectral Techniques for Embedding and Clustering. In T. DIETTERICH,

- S. BECKER, and Z. GHARAMANI (Eds.), *Advances in Neural Information Processing Systems 14 (NIPS'01)*, Cambridge, MA. MIT Press. (Cited on pages 32 and 88.)
- Bell, A. and T. J. Sejnowski (1997). The independent components of natural scenes are edge filters. *Vision Research* 37, 3327–3338. (Cited on page 146.)
- Bengio, Y. (2009). Learning deep architectures for AI. *Foundations and Trends in Machine Learning* 2(1), 1–127. Also published as a book. Now Publishers, 2009. (Cited on pages 31, 33, 35, 81, 92, 127 and 131.)
- Bengio, Y. and O. Delalleau (2009, June). Justifying and Generalizing Contrastive Divergence. *Neural Computation* 21(6), 1601–1621. (Cited on pages 38, 42, 47, 91 and 92.)
- Bengio, Y., O. Delalleau, and N. Le Roux (2006). The Curse of Highly Variable Functions for Local Kernel Machines. In Y. WEISS, B. SCHÖLKOPF, and J. PLATT (Eds.), *Advances in Neural Information Processing Systems 18 (NIPS'05)*, pp. 107–114. Cambridge, MA: MIT Press. (Cited on pages 29, 30 and 81.)
- Bengio, Y., R. Ducharme, P. Vincent, and C. Jauvin (2003). A Neural Probabilistic Language Model. *Journal of Machine Learning Research* 3, 1137–1155. (Cited on page 51.)
- Bengio, Y. and X. Glorot (2010, May). Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of AIS-TATS 2010*, Volume 9, pp. 249–256. (Cited on pages 43 and 78.)
- Bengio, Y., P. Lamblin, D. Popovici, and H. Larochelle (2007). Greedy Layer-Wise Training of Deep Networks. In B. SCHÖLKOPF, J. PLATT, and T. HOFFMAN (Eds.), *Advances in Neural Information Processing Systems 19 (NIPS'06)*, pp. 153–160. MIT Press. (Cited on pages 24, 36, 39, 40, 41, 55, 56, 59, 62, 64, 65, 70, 81, 82, 83, 90, 94, 96, 109, 110, 127, 130, 133 and 137.)
- Bengio, Y. and Y. LeCun (2007). Scaling Learning Algorithms towards AI. In L. BOTTOU, O. CHAPPELLE, D. DECOSTE, and J. WESTON (Eds.), *Large Scale Kernel Machines*. MIT Press. (Cited on pages 29, 31, 32, 33, 34, 35, 54, 55, 56, 61, 81 and 82.)
- Berkes, P. and L. Wiskott (2002). Applying Slow Feature Analysis to Image Sequences Yields a Rich Repertoire of Complex Cell Properties. In J. R. DORRONSORO (Ed.), *Proc. Intl. Conf. on Artificial*

- Neural Networks - ICANN'02*, Lecture Notes in Computer Science, pp. 81–86. Springer. (Cited on page 143.)
- Berkes, P. and L. Wiskott (2006). On the analysis and interpretation of inhomogeneous quadratic forms as receptive fields. *Neural Computation* 18(8), 1868–1895. (Cited on pages 130, 131 and 143.)
- Bishop, C. (1995). *Neural Networks for Pattern Recognition*. London, UK: Oxford University Press. (Cited on page 24.)
- Blumer, A., A. Ehrenfeucht, D. Haussler, and M. Warmuth (1987). Occam's razor. *Inf. Proc. Let.* 24, 377–380. (Cited on page 10.)
- Bornstein, M. H. (1987). *Sensitive periods in development: interdisciplinary perspectives / edited by Marc H. Bornstein*. Hillsdale, N.J.: Lawrence Erlbaum Associates. (Cited on page 85.)
- Boser, B. E., I. M. Guyon, and V. N. Vapnik (1992). A training algorithm for optimal margin classifiers. In *COLT '92: Proceedings of the fifth annual workshop on Computational learning theory*, New York, NY, USA, pp. 144–152. ACM. (Cited on page 27.)
- Breuleux, O., Y. Bengio, and P. Vincent (2010). Unlearning for Better Mixing. Technical Report 1349, Université de Montréal/DIRO. (Cited on page 46.)
- Carreira-Perpiñan, M. A. and G. E. Hinton (2005). On Contrastive Divergence Learning. In R. G. COWELL and Z. GHAMRANI (Eds.), *Proceedings of the Tenth International Workshop on Artificial Intelligence and Statistics (AISTATS'05)*, pp. 33–40. Society for Artificial Intelligence and Statistics. (Cited on page 38.)
- Caruana, R. (1997). Multitask Learning. *Machine Learning* 28(1), 41–75. (Cited on page 155.)
- Chang, C.-C. and C.-J. Lin (2001). *LIBSVM: a library for support vector machines*. Software available at <http://www.csie.ntu.edu.tw/~cjlin/libsvm>. (Cited on page 69.)
- Chapelle, O., B. Schölkopf, and A. Zien (Eds.) (2006). *Semi-Supervised Learning*. Cambridge, MA: MIT Press. (Cited on page 88.)
- Chapelle, O., J. Weston, and B. Schölkopf (2003). Cluster kernels for semi-supervised learning. In S. BECKER, S. THRUN, and K. OBERMAYER (Eds.), *Advances in Neural Information Processing Systems 15 (NIPS'02)*, Cambridge, MA, pp. 585–592. MIT Press. (Cited on page 88.)

- Cho, Y. and L. Saul (2010a). Kernel Methods for Deep Learning. In Y. BENGIO, D. SCHUURMANS, C. WILLIAMS, J. LAFFERTY, and A. CULOTTA (Eds.), *Advances in Neural Information Processing Systems 22 (NIPS'09)*, pp. 342–350. NIPS Foundation. (Cited on pages 51 and 57.)
- Cho, Y. and L. K. Saul (2010b). Large-Margin Classification in Infinite Neural Networks. *Neural Computation* 22(10), 2678–2697. (Cited on pages 51 and 57.)
- Cohen, W. W., A. McCallum, and S. T. Roweis (Eds.) (2008). *Proceedings of the Twenty-fifth International Conference on Machine Learning (ICML'08)*. ACM. (Cited on pages 164 and 172.)
- Collobert, R. and J. Weston (2007). Fast Semantic Extraction Using a Novel Neural Network Architecture. In *ACL 2007, Proceedings of the 45th Annual Meeting of the Association for Computational Linguistics, June 23-30, 2007, Prague, Czech Republic*. The Association for Computer Linguistics. (Cited on page 51.)
- Collobert, R. and J. Weston (2008). A Unified Architecture for Natural Language Processing: Deep Neural Networks with Multitask Learning. In W. W. COHEN, A. MCCALLUM, and S. T. ROWEIS (Eds.), *Proceedings of the Twenty-fifth International Conference on Machine Learning (ICML'08)*, pp. 160–167. ACM. (Cited on pages 51, 81 and 82.)
- Cortes, C. and V. Vapnik (1995). Support Vector Networks. *Machine Learning* 20, 273–297. (Cited on pages 26 and 33.)
- Dahl, G. E., M. Ranzato, A. Mohamed, and G. E. Hinton (2010). Phone Recognition with the Mean-Covariance Restricted Boltzmann Machine. In *Advances in Neural Information Processing Systems (NIPS)*. (Cited on page 45.)
- Dayan, P. and L. F. Abbott (2001). *Theoretical Neuroscience*. The MIT Press. (Cited on page 149.)
- Decoste, D. and B. Schölkopf (2002). Training invariant support vector machines. *Machine Learning* 46, 161–190. (Cited on pages 30, 34 and 61.)
- Desjardins, G., A. Courville, Y. Bengio, P. Vincent, and O. Delalleau (2010, May). Tempered Markov Chain Monte Carlo for training of Restricted Boltzmann Machine. In *JMLR W&CP: Proceedings*

- of the Thirteenth International Conference on Artificial Intelligence and Statistics (AISTATS 2010)*, Volume 9, pp. 145–152. (Cited on pages 46, 78 and 133.)
- Erhan, D., Y. Bengio, A. Courville, P.-A. Manzagol, P. Vincent, and S. Bengio (2010, February). Why Does Unsupervised Pre-training Help Deep Learning? *Journal of Machine Learning Research* 11, 625–660. (Cited on pages 128 and 150.)
- Erhan, D., A. Courville, and Y. Bengio (2010, October). Understanding Representations Learned in Deep Architectures. Technical Report 1355, Université de Montréal/DIRO. (Cited on page 119.)
- Fei-Fei, L., R. Fergus, and P. Perona (2004). Learning Generative Visual Models from Few Training Examples: An Incremental Bayesian Approach Tested on 101 Object Categories. In *Conference on Computer Vision and Pattern Recognition Workshop (CVPR 2004)*, pp. 178. (Cited on pages 44, 51 and 136.)
- Fisher, W., G. Doddington, and K. Goudie-Marshall (1986). The DARPA speech recognition research database: specifications and status. In *Proc. DARPA Workshop on Speech Recognition*, pp. 93–99. (Cited on page 45.)
- Fukushima, K., S. Miyake, and T. Ito (1983). Neocognitron: A Neural Network Model for a Mechanism of Visual Pattern Recognition. *IEEE Transactions on Systems, Man, and Cybernetics* 13(5), 826–834. (Cited on page 35.)
- Gallinari, P., Y. LeCun, S. Thiria, and F. Fogelman-Soulie (1987). Memoires associatives distribuees. In *Proceedings of COGNITIVA 87*, Paris, La Villette. (Cited on page 93.)
- Gehler, P. V., A. D. Holub, and M. Welling (2006). The rate adapting poisson model for information retrieval and object recognition. In W. W. COHEN and A. MOORE (Eds.), *Proceedings of the Twenty-three International Conference on Machine Learning (ICML'06)*, New York, NY, USA, pp. 337–344. ACM. (Cited on page 42.)
- Geman, S. and D. Geman (1984, November). Stochastic Relaxation, Gibbs Distributions, and the Bayesian Restoration of Images. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 6, 721–741. (Cited on page 37.)

- Goldberger, J., S. Roweis, G. E. Hinton, and R. Salakhutdinov (2005). Neighbourhood Components Analysis. In L. SAUL, Y. WEISS, and L. BOTTOU (Eds.), *Advances in Neural Information Processing Systems 17 (NIPS'04)*. MIT Press. (Cited on page 42.)
- Goodfellow, I., Q. Le, A. Saxe, and A. Ng (2009). Measuring Invariances in Deep Networks. In Y. BENGIO, D. SCHUURMANS, C. WILLIAMS, J. LAFFERTY, and A. CULOTTA (Eds.), *Advances in Neural Information Processing Systems 22 (NIPS'09)*, pp. 646–654. (Cited on pages 48, 119, 125, 129, 148, 149, 150 and 151.)
- Gordon, D. and M. Desjardins (1995). Evaluation and selection of biases in machine learning. *Machine Learning 20*(1), 5–22. (Cited on page 16.)
- Hadsell, R., A. Erkan, P. Sermanet, M. Scoffier, U. Muller, and Y. LeCun (2008). Deep Belief Net Learning in a Long-Range Vision System for Autonomous Off-Road Driving. In *Proc. Intelligent Robots and Systems (IROS'08)*, pp. 628–633. (Cited on page 82.)
- Hammer, B. and K. Gersmann (2003). A Note on the Universal Approximation Capability of Support Vector Machines. *Neural Process. Lett. 17*(1), 43–53. (Cited on page 28.)
- Håstad, J. (1986). Almost optimal lower bounds for small depth circuits. In *Proceedings of the 18th annual ACM Symposium on Theory of Computing*, Berkeley, California, pp. 6–20. ACM Press. (Cited on page 81.)
- Håstad, J. and M. Goldmann (1991). On the power of small-depth threshold circuits. *Computational Complexity 1*, 113–129. (Cited on page 81.)
- Hinton, G. E. (2002). Training products of experts by minimizing contrastive divergence. *Neural Computation 14*, 1771–1800. (Cited on pages 37, 63, 91 and 131.)
- Hinton, G. E. (2006). To recognize shapes, first learn to generate images. Technical Report UTML TR 2006-003, University of Toronto. (Cited on page 59.)
- Hinton, G. E. (2007). To recognize shapes, first learn to generate images. In P. CISEK, T. DREW, and J. KALASKA (Eds.), *Computational Neuroscience: Theoretical Insights into Brain Function*. Elsevier. (Cited on page 119.)

- Hinton, G. E. (2010). A practical guide to training restricted Boltzmann machines. Technical Report UTML TR 2010-003, Department of Computer Science, University of Toronto. (Cited on page 48.)
- Hinton, G. E., P. Dayan, B. J. Frey, and R. M. Neal (1995). The wake-sleep algorithm for unsupervised neural networks. *Science* 268, 1558–1161. (Cited on page 39.)
- Hinton, G. E., S. Osindero, and Y. Teh (2006). A fast learning algorithm for deep belief nets. *Neural Computation* 18, 1527–1554. (Cited on pages v, vii, 35, 39, 40, 41, 55, 56, 59, 62, 65, 75, 81, 82, 90, 91, 94, 127, 128, 129, 130, 131, 133, 153 and 155.)
- Hinton, G. E., S. Osindero, M. Welling, and Y. Teh (2006, July–August). Unsupervised Discovery of Non-Linear Structure using Contrastive Backpropagation. *Cognitive Science* 30(4), 725–731. (Cited on page 43.)
- Hinton, G. E. and R. Salakhutdinov (2006, July). Reducing the dimensionality of data with neural networks. *Science* 313(5786), 504–507. (Cited on pages 39, 42, 59, 90, 118 and 120.)
- Hinton, G. E. and T. J. Sejnowski (1986). Learning and relearning in Boltzmann machines. In D. E. RUMELHART and J. L. MCCLELLAND (Eds.), *Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Volume 1: Foundations*, pp. 282–317. Cambridge, MA: MIT Press. (Cited on page 36.)
- Hinton, G. E., T. J. Sejnowski, and D. H. Ackley (1984). Boltzmann machines: Constraint satisfaction networks that learn. Technical Report TR-CMU-CS-84-119, Carnegie-Mellon University, Dept. of Computer Science. (Cited on page 36.)
- Hoerl, A. and R. Kennard (1970). Ridge regression: biased estimation for non-orthogonal problems. *Technometrics* 12, 55–67. (Cited on page 11.)
- Hornik, K., M. Stinchcombe, and H. White (1989). Multilayer Feed-forward Networks Are Universal Approximators. *Neural Networks* 2, 359–366. (Cited on page 24.)
- Huang, G. B., M. Ramesh, T. Berg, and E. Learned-Miller (2007, October). Labeled Faces in the Wild: A Database for Studying Face Recognition in Unconstrained Environments. Technical Report 07-49, University of Massachusetts, Amherst. (Cited on page 43.)

- Jarrett, K., K. Kavukcuoglu, M. Ranzato, and Y. LeCun (2009). What is the Best Multi-Stage Architecture for Object Recognition? In *Proc. International Conference on Computer Vision (ICCV'09)*. IEEE. (Cited on page 51.)
- Larochelle, H. and Y. Bengio (2008). Classification using Discriminative Restricted Boltzmann Machines. See Cohen *et al.* (2008), pp. 536–543. (Cited on pages 41, 44, 89 and 120.)
- Larochelle, H., Y. Bengio, J. Louradour, and P. Lamblin (2009, January). Exploring Strategies for Training Deep Neural Networks. *Journal of Machine Learning Research* 10, 1–40. (Cited on pages 41, 96, 120, 128, 137 and 150.)
- Larochelle, H., D. Erhan, A. Courville, J. Bergstra, and Y. Bengio (2007). An Empirical Evaluation of Deep Architectures on Problems with Many Factors of Variation. In Z. GHAMRANI (Ed.), *Proceedings of the 24th International Conference on Machine Learning (ICML'07)*, pp. 473–480. ACM. (Cited on pages 39, 41, 82, 94, 96, 130, 133 and 146.)
- Larochelle, H., D. Erhan, and P. Vincent (2009, April). Deep Learning using Robust Interdependent Codes. In *Proceedings of the Twelfth International Conference on Artificial Intelligence and Statistics (AISTATS 2009)*, pp. 312–319. (Cited on pages 49 and 57.)
- Lasserre, J. A., C. M. Bishop, and T. P. Minka (2006). Principled Hybrids of Generative and Discriminative Models. In *Proceedings of the Computer Vision and Pattern Recognition Conference (CVPR'06)*, Washington, DC, USA, pp. 87–94. IEEE Computer Society. (Cited on pages 83 and 89.)
- Le Roux, N. and Y. Bengio (2008, June). Representational Power of Restricted Boltzmann Machines and Deep Belief Networks. *Neural Computation* 20(6), 1631–1649. (Cited on pages 39 and 47.)
- Le Roux, N. and Y. Bengio (2010, August). Deep Belief Networks are Compact Universal Approximators. *Neural Computation* 22(8), 2192–2207. (Cited on page 47.)
- Le Roux, N., P.-A. Manzagol, and Y. Bengio (2008). Topmoumoute online natural gradient algorithm. In J. PLATT, D. KOLLER, Y. SINGER, and S. ROWEIS (Eds.), *Advances in Neural Information Processing Systems 20 (NIPS'07)*, Cambridge, MA. MIT Press. (Cited on page 57.)

- LeCun, Y. (1986). Learning Processes in an Asymmetric Threshold Network. In F. FOGELMAN-SOULIÉ, E. BIENENSTOCK, and G. WEISBUCH (Eds.), *Disordered Systems and Biological Organization*, pp. 233–240. Les Houches, France: Springer-Verlag. (Cited on page 21.)
- LeCun, Y. (1987). *Modèles connexionistes de l'apprentissage*. Ph. D. thesis, Université de Paris VI. (Cited on page 93.)
- LeCun, Y., B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel (1989). Backpropagation Applied to Handwritten Zip Code Recognition. *Neural Computation* 1(4), 541–551. (Cited on page 61.)
- LeCun, Y., L. Bottou, Y. Bengio, and P. Haffner (1998, November). Gradient-Based Learning Applied to Document Recognition. *Proceedings of the IEEE* 86(11), 2278–2324. (Cited on pages 35, 44, 51 and 94.)
- LeCun, Y., L. Bottou, G. B. Orr, and K.-R. Müller (1998). Efficient BackProp. In G. B. ORR and K.-R. MÜLLER (Eds.), *Neural Networks: Tricks of the Trade*, pp. 9–50. Springer. (Cited on pages 24 and 25.)
- LeCun, Y., S. Chopra, R. Hadsell, M.-A. Ranzato, and F.-J. Huang (2006). A Tutorial on Energy-Based Learning. In G. BAKIR, T. HOFMAN, B. SCHOLKOPF, A. SMOLA, and B. TASKAR (Eds.), *Predicting Structured Data*, pp. 191–246. MIT Press. (Cited on page 49.)
- LeCun, Y., F.-J. Huang, and L. Bottou (2004). Learning Methods for Generic Object Recognition with Invariance to Pose and Lighting. In *Proceedings of the Computer Vision and Pattern Recognition Conference (CVPR'04)*, Volume 2, Los Alamitos, CA, USA, pp. 97–104. IEEE Computer Society. (Cited on pages 43, 51, 60 and 73.)
- Lee, H., C. Ekanadham, and A. Ng (2008). Sparse deep belief net model for visual area V2. In J. PLATT, D. KOLLER, Y. SINGER, and S. ROWEIS (Eds.), *Advances in Neural Information Processing Systems 20 (NIPS'07)*, pp. 873–880. Cambridge, MA: MIT Press. (Cited on pages 43, 47, 90, 123, 124, 129, 130, 139, 140, 149 and 151.)
- Lee, H., R. Grosse, R. Ranganath, and A. Y. Ng (2009). Convolutional deep belief networks for scalable unsupervised learning of hierarchical

- representations. In L. BOTTOU and M. LITTMAN (Eds.), *Proceedings of the Twenty-sixth International Conference on Machine Learning (ICML'09)*. Montreal (Qc), Canada: ACM. (Cited on pages 44, 82, 119, 120, 130 and 135.)
- Lee, H., P. Pham, Y. Largman, and A. Ng (2009). Unsupervised feature learning for audio classification using convolutional deep belief networks. In Y. BENGIO, D. SCHUURMANS, C. WILLIAMS, J. LAFERTY, and A. CULOTTA (Eds.), *Advances in Neural Information Processing Systems 22 (NIPS'09)*, pp. 1096–1104. (Cited on pages 45 and 48.)
- Li, P. (2010). Robust LogitBoost and Adaptive Base Class (ABC) LogitBoost. In *In Proceedings of Uncertainty on Artificial Intelligence (UAI 2010)*. (Cited on page 57.)
- Long, P. M. and R. A. Servedio (2010). Restricted Boltzmann Machines are Hard to Approximately Evaluate or Simulate. In *Proceedings of the 27th International Conference on Machine Learning (ICML'10)*. (Cited on page 47.)
- Loosli, G., S. Canu, and L. Bottou (2007). Training Invariant Support Vector Machines using Selective Sampling. In L. BOTTOU, O. CHAPELLE, D. DECOSTE, and J. WESTON (Eds.), *Large Scale Kernel Machines*, pp. 301–320. Cambridge, MA.: MIT Press. (Cited on pages 94 and 136.)
- Marlin, B., K. Swersky, B. Chen, and N. de Freitas (2009). Inductive Principles for Restricted Boltzmann Machine Learning. In *Proceedings of The Thirteenth International Conference on Artificial Intelligence and Statistics (AISTATS'10)*, Volume 9, pp. 509–516. (Cited on pages 46, 57 and 136.)
- Martens, J. (2010, June). Deep Learning via Hessian-free Optimization. In L. BOTTOU and M. LITTMAN (Eds.), *Proceedings of the Twenty-seventh International Conference on Machine Learning (ICML-10)*, pp. 735–742. ACM. (Cited on pages 79 and 156.)
- Memisevic, R. and G. E. Hinton (2007). Unsupervised learning of image transformations. In *Proceedings of the Computer Vision and Pattern Recognition Conference (CVPR'07)*. (Cited on page 45.)
- Memisevic, R. and G. E. Hinton (2010, June). Learning to Represent Spatial Transformations with Factored Higher-Order Boltzmann Machines. *Neural Computation* 22(6), 1473–1492. (Cited on page 45.)

- Minsky, M. L. and S. A. Papert (1969). *Perceptrons*. Cambridge: MIT Press. (Cited on page 20.)
- Mitchell, T. M. (1990). The Need for Biases in Learning Generalizations. In J. SHAVLIK and T. DIETTERICH (Eds.), *Readings in Machine Learning*, pp. 184–191. Morgan Kaufmann. (Cited on page 16.)
- Mobahi, H., R. Collobert, and J. Weston (2009, June). Deep Learning from Temporal Coherence in Video. In L. BOTTOU and M. LITTMAN (Eds.), *Proceedings of the 26th International Conference on Machine Learning*, Montreal, pp. 737–744. Omnipress. (Cited on page 82.)
- Murray, I. and R. Salakhutdinov (2009). Evaluating probabilities under high-dimensional latent variable models. In D. KOLLER, D. SCHURMANS, Y. BENGIO, and L. BOTTOU (Eds.), *Advances in Neural Information Processing Systems 21 (NIPS'08)*, Volume 21, pp. 1137–1144. (Cited on page 47.)
- Nair, V. and G. E. Hinton (2010). Rectified Linear Units Improve Restricted Boltzmann Machines. In L. BOTTOU and M. LITTMAN (Eds.), *Proceedings of the Twenty-seventh International Conference on Machine Learning (ICML-10)*, pp. 807–814. ACM. (Cited on pages 43 and 78.)
- Ng, A. Y. and M. I. Jordan (2002). On Discriminative vs. Generative Classifiers: A comparison of logistic regression and naive Bayes. In T. DIETTERICH, S. BECKER, and Z. GHAHRAMANI (Eds.), *Advances in Neural Information Processing Systems 14 (NIPS'01)*, pp. 841–848. (Cited on page 88.)
- Olshausen, B. A. and D. J. Field (1996). Emergence of simple-cell receptive field properties by learning a sparse code for natural images. *Nature* 381, 607–609. (Cited on pages 50, 123 and 136.)
- Osindero, S. and G. E. Hinton (2008). Modeling image patches with a directed hierarchy of Markov random field. In J. PLATT, D. KOLLER, Y. SINGER, and S. ROWEIS (Eds.), *Advances in Neural Information Processing Systems 20 (NIPS'07)*, Cambridge, MA, pp. 1121–1128. MIT Press. (Cited on pages 43, 49, 82, 123, 128, 139 and 150.)
- Osindero, S., M. Welling, and G. E. Hinton (2006). Topographic Product Models Applied to Natural Scene Statistics. *Neural Computation* 18(2), 381–414. (Cited on page 43.)

- Parker, D. B. (1985). Learning Logic. Technical Report TR-47, Center for Computational Research in Economics and Management Science, Massachusetts Institute of Technology, Cambridge, MA. (Cited on page 21.)
- Povley, D. and P. Woodland (2002). Minimum error and I-smoothing for improved discriminative training. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing (ICASSP'2002)*, Volume 1, Orlando, Florida, USA, pp. I-105–I-108. IEEE. (Cited on page 83.)
- Ranzato, M., Y.-L. Boureau, and Y. LeCun (2008). Sparse feature learning for deep belief networks. In J. PLATT, D. KOLLER, Y. SINGER, and S. ROWEIS (Eds.), *Advances in Neural Information Processing Systems 20 (NIPS'07)*, Cambridge, MA, pp. 1185–1192. MIT Press. (Cited on pages 50, 82, 90, 93, 130 and 133.)
- Ranzato, M. and G. Hinton (2010). Modeling pixel means and covariances using factorized third-order Boltzmann machines. In *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on*, pp. 2551–2558. IEEE. (Cited on page 45.)
- Ranzato, M., F. Huang, Y. Boureau, and Y. LeCun (2007). 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. (Cited on page 51.)
- Ranzato, M., A. Krizhevsky, and G. Hinton (2010). Factored 3-way restricted Boltzmann machines for modeling natural images. In Y. W. TEH and M. TITTERINGTON (Eds.), *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics (AISTATS 2010)*, *JMLR W&CP*, Volume 9, pp. 621–628. (Cited on page 45.)
- Ranzato, M. and Y. LeCun (2007). A Sparse and Locally Shift Invariant Feature Extractor Applied to Document Images. In *International Conference on Document Analysis and Recognition (ICDAR'07)*, Washington, DC, USA, pp. 1213–1217. IEEE Computer Society. (Cited on page 51.)
- Ranzato, M., C. Poultney, S. Chopra, and Y. LeCun (2007). Efficient Learning of Sparse Representations with an Energy-Based Model. In B. SCHÖLKOPF, J. PLATT, and T. HOFFMAN (Eds.), *Advances*

- in Neural Information Processing Systems 19 (NIPS'06)*, pp. 1137–1144. MIT Press. (Cited on pages 50, 55, 56, 81, 82, 90, 93, 94, 96, 127, 130 and 133.)
- Rissanen, J. (1983). A universal data compression system. *IEEE Transactions on Information Theory* 29, 656–664. (Cited on page 16.)
- Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review* 65, 386–408. (Cited on pages 4 and 19.)
- Roweis, S. and L. K. Saul (2000, December). Nonlinear dimensionality reduction by locally linear embedding. *Science* 290(5500), 2323–2326. (Cited on page 32.)
- Rumelhart, D. E., G. E. Hinton, and R. J. Williams (1986). Learning Representations by Back-Propagating Errors. *Nature* 323, 533–536. (Cited on pages 21 and 23.)
- Salakhutdinov, R. (2010). Learning in Markov Random Fields using Tempered Transitions. In Y. BENGIO, D. SCHUURMANS, C. WILLIAMS, J. LAFFERTY, and A. CULOTTA (Eds.), *Advances in Neural Information Processing Systems 22 (NIPS'09)*. (Cited on page 46.)
- Salakhutdinov, R. and G. E. Hinton (2007a). Learning a Nonlinear Embedding by Preserving Class Neighbourhood Structure. In *Proceedings of the Eleventh International Conference on Artificial Intelligence and Statistics (AISTATS'07)*, San Juan, Porto Rico. Omnipress. (Cited on pages 42, 59 and 65.)
- Salakhutdinov, R. and G. E. Hinton (2007b). Semantic Hashing. In *Proceedings of the 2007 Workshop on Information Retrieval and applications of Graphical Models (SIGIR 2007)*, Amsterdam. Elsevier. (Cited on pages 43 and 82.)
- Salakhutdinov, R. and G. E. Hinton (2008). Using Deep Belief Nets to Learn Covariance Kernels for Gaussian Processes. In J. PLATT, D. KOLLER, Y. SINGER, and S. ROWEIS (Eds.), *Advances in Neural Information Processing Systems 20 (NIPS'07)*, Cambridge, MA, pp. 1249–1256. MIT Press. (Cited on pages 43 and 89.)
- Salakhutdinov, R. and G. E. Hinton (2009). Deep Boltzmann Machines. In *Proceedings of The Twelfth International Conference on Artificial Intelligence and Statistics (AISTATS'09)*, Volume 5, pp. 448–455. (Cited on pages 44 and 45.)

- Salakhutdinov, R. and H. Larochelle (2010). Efficient Learning of Deep Boltzmann Machines. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics (AISTATS 2010)*, *JMLR W&CP*, Volume 9, Sardinia, Italy, pp. 693–700. (Cited on page 45.)
- Salakhutdinov, R., A. Mnih, and G. E. Hinton (2007). Restricted Boltzmann machines for collaborative filtering. In Z. GHAHRAMANI (Ed.), *Proceedings of the Twenty-fourth International Conference on Machine Learning (ICML'07)*, New York, NY, USA, pp. 791–798. ACM. (Cited on pages 44 and 82.)
- Salakhutdinov, R. and I. Murray (2008). On the Quantitative Analysis of Deep Belief Networks. In W. W. COHEN, A. MCCALLUM, and S. T. ROWEIS (Eds.), *Proceedings of the Twenty-fifth International Conference on Machine Learning (ICML'08)*, Volume 25, pp. 872–879. ACM. (Cited on page 47.)
- Schölkopf, B. and A. J. Smola (2002). *Learning with Kernels: Support Vector Machines, Regularization, Optimization and Beyond*. Cambridge, MA: MIT Press. (Cited on page 28.)
- Searle, J. (1999). The Chinese Room. In R. WILSON and F. KEIL (Eds.), *The MIT Encyclopedia of the Cognitive Sciences*. Cambridge, MA: MIT Press. (Cited on page 3.)
- Seung, S. H. (1998). Learning continuous attractors in recurrent networks. In M. JORDAN, M. KEARNS, and S. SOLLA (Eds.), *Advances in Neural Information Processing Systems 10 (NIPS'97)*, pp. 654–660. MIT Press. (Cited on page 93.)
- Sjöberg, J. and L. Ljung (1995). Overtraining, regularization and searching for a minimum, with application to neural networks. *International Journal of Control* 62(6), 1391–1407. (Cited on pages 83 and 90.)
- Smolensky, P. (1986). Information Processing in Dynamical Systems: Foundations of Harmony Theory. In D. E. RUMELHART and J. L. MCCLELLAND (Eds.), *Parallel Distributed Processing*, Volume 1, Chapter 6, pp. 194–281. Cambridge: MIT Press. (Cited on page 36.)
- Solomonoff, R. J. (1964). A formal theory of inductive inference. *Information and Control* 7, 1–22, 224–254. (Cited on page 16.)
- Steinwart, I. (2003). Sparseness of support vector machines. *Journal of Machine Learning Research* 4, 1071–1105. (Cited on page 28.)

- Susskind, J. M., G. E., J. R. Movellan, and A. K. Anderson (2008). Generating Facial Expressions with Deep Belief Nets. In V. KORDIC (Ed.), *Affective Computing, Emotion Modelling, Synthesis and Recognition*, pp. 421–440. ARS Publishers. (Cited on page 119.)
- Sutskever, I. and G. E. Hinton (2008). Deep Narrow Sigmoid Belief Networks are Universal Approximators. *Neural Computation* 20(11), 2629–2636. (Cited on page 47.)
- Sutskever, I. and T. Tieleman (2010). On the Convergence Properties of Contrastive Divergence. In Y. W. TEH and M. TITTERINGTON (Eds.), *Proc. of the International Conference on Artificial Intelligence and Statistics (AISTATS)*, Volume 9, pp. 789–795. (Cited on page 38.)
- Taylor, G. and G. Hinton (2009, June). Factored Conditional Restricted Boltzmann Machines for Modeling Motion Style. In L. BOTTOU and M. LITTMAN (Eds.), *Proceedings of the 26th International Conference on Machine Learning (ICML'09)*, Montreal, pp. 1025–1032. Omnipress. (Cited on page 45.)
- Taylor, G., G. E. Hinton, and S. Roweis (2007). Modeling Human Motion Using Binary Latent Variables. In B. SCHÖLKOPF, J. PLATT, and T. HOFFMAN (Eds.), *Advances in Neural Information Processing Systems 19 (NIPS'06)*, pp. 1345–1352. Cambridge, MA: MIT Press. (Cited on page 45.)
- Taylor, G., L. Sigal, D. Fleet, and G. Hinton (2010). Dynamic binary latent variable models for 3D pose tracking. In *Proc. Conference on Computer Vision and Pattern Recognition (CVPR'2010)*. (Cited on page 45.)
- Tenenbaum, J., V. de Silva, and J. C. Langford (2000, December). A Global Geometric Framework for Nonlinear Dimensionality Reduction. *Science* 290(5500), 2319–2323. (Cited on pages 32, 33 and 100.)
- Tieleman, T. (2008). Training restricted Boltzmann machines using approximations to the likelihood gradient. In W. W. COHEN, A. MCCALLUM, and S. T. ROWEIS (Eds.), *Proceedings of the Twenty-fifth International Conference on Machine Learning (ICML'08)*, pp. 1064–1071. ACM. (Cited on pages 46 and 78.)
- Tieleman, T. and G. Hinton (2009). Using Fast Weights to Improve Persistent Contrastive Divergence. In L. BOTTOU and M. LITTMAN (Eds.), *Proceedings of the Twenty-sixth International Conference on*

- Machine Learning (ICML'09)*, pp. 1033–1040. ACM. (Cited on pages 46, 78 and 133.)
- Utgoff, P. (1986). Shift of bias for inductive concept learning. *Machine learning: An artificial intelligence approach 2*, 107–148. (Cited on page 16.)
- van der Maaten, L. and G. E. Hinton (2008, November). Visualizing Data using t-SNE. *Journal of Machine Learning Research 9*, 2579–2605. (Cited on page 100.)
- Vapnik, V. (1998). *Statistical Learning Theory*. Wiley, Lecture Notes in Economics and Mathematical Systems, volume 454. (Cited on pages 26, 27 and 33.)
- Vapnik, V. N. (1982). *Estimation of Dependences Based on Empirical Data*. Berlin: Springer-Verlag. (Cited on page 11.)
- Vapnik, V. N. and A. Y. Chervonenkis (1971). On the Uniform Convergence of Relative Frequencies of Events to Their Probabilities. *Theory of Probability and Its Applications 16*, 264–280. (Cited on pages 9 and 16.)
- Vincent, P., H. Larochelle, Y. Bengio, and P.-A. Manzagol (2008). Extracting and Composing Robust Features with Denoising Autoencoders. See Cohen *et al.* (2008), pp. 1096–1103. (Cited on pages 48, 49, 57, 81, 90, 92, 93, 94, 96, 107, 123, 130, 131, 132, 133 and 137.)
- Welling, M. (2009a). Herding Dynamic Weights for Partially Observed Random Field Models. In *Proceedings of the 25th Conference in Uncertainty in Artificial Intelligence (UAI'09)*. Morgan Kaufmann. (Cited on page 46.)
- Welling, M. (2009b). Herding Dynamic Weights to Learn. In L. BOTTOU and M. LITTMAN (Eds.), *Proceedings of the Twenty-sixth International Conference on Machine Learning (ICML'09)*. ACM. (Cited on page 46.)
- Welling, M., M. Rosen-Zvi, and G. E. Hinton (2005). Exponential Family Harmoniums with an Application to Information Retrieval. In L. SAUL, Y. WEISS, and L. BOTTOU (Eds.), *Advances in Neural Information Processing Systems 17 (NIPS'04)*, Cambridge, MA, pp. 1481–1488. MIT Press. (Cited on pages 36, 63 and 91.)

- Werbos, P. (1974). *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*. Ph. D. thesis, Harvard University. (Cited on page 21.)
- Weston, J., F. Ratle, and R. Collobert (2008). Deep Learning via Semi-Supervised Embedding. In W. W. COHEN, A. MCCALLUM, and S. T. ROWEIS (Eds.), *Proceedings of the Twenty-fifth International Conference on Machine Learning (ICML '08)*, New York, NY, USA, pp. 1168–1175. ACM. (Cited on pages 81, 82, 89, 90, 120, 130 and 133.)
- Widrow, B. and M. Lehr (1990, September). 30 Years of Adaptive Neural Networks: Perceptron, Madaline, and Backpropagation. *Proceedings of the IEEE* 78(9), 1415–1442. (Cited on page 20.)
- Wolpert, D. H. (1996). The lack of a priori distinction between learning algorithms. *Neural Computation* 8(7), 1341–1390. (Cited on page 16.)
- Yao, A. (1985). Separating the polynomial-time hierarchy by oracles. In *Proceedings of the 26th Annual IEEE Symposium on Foundations of Computer Science*, pp. 1–10. (Cited on page 81.)
- Zhu, L., Y. Chen, and A. Yuille (2009). Unsupervised Learning of Probabilistic Grammar-Markov Models for Object Categories. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 31(1), 114–128. (Cited on page 81.)