THE SPATIAL INDUCTIVE BIAS OF DEEP LEARNING

by
Benjamin R. Mitchell

A dissertation submitted to The Johns Hopkins University in conformity with the requirements for the degree of Doctor of Philosophy.

Baltimore, Maryland
March, 2017

Abstract

In the past few years, Deep Learning has become the method of choice for producing state-of-the-art results on machine learning problems involving images, text, and speech. The explosion of interest in these techniques has resulted in a large number of successful applications of deep learning, but relatively few studies exploring the nature of and reason for that success.

This dissertation is motivated by a desire to understand and reproduce the performance characteristics of deep learning systems, particularly Convolutional Neural Networks (CNNs). One factor in the success of CNNs is that they have an inductive bias that assumes a certain type of spatial structure is present in the data. We give a formal definition of how this type of *spatial structure* can be characterised, along with some statistical tools for testing whether spatial structure is present in a given dataset. These tools are applied to several standard image datasets, and the results are analyzed.

ABSTRACT

We demonstrate that CNNs rely heavily on the presence of such structure, and then show several ways that a similar bias can be introduced into other methods. The first is a partition-based method for training Restricted Boltzmann Machines and Deep Belief Networks, which is able to speed up convergence significantly without changing the overall representational power of the network. The second is a deep partitioned version of Principal Component Analysis, which demonstrates that a spatial bias can be useful even in a model that is non-connectionist and completely linear. The third is a variation on projective Random Forests, which shows that we can introduce a spatial bias with only minor changes to the algorithm, and no externally imposed partitioning is required. In each case, we can show that introducing a spatial bias results in improved performance on spatial data.

Primary Reader: John Sheppard

Secondary Readers: Greg Hager, Raman Arora, Carey Priebe

Acknowledgments

This dissertation would not have been possible without the help and support of many people. Chief among these is my adviser, John Sheppard, who has provided me with encouragement and support over the long period this dissertation represents. He has been a constant friend and advocate on my behalf, in spite of the geographic distance which has separated us for much of this period. He has my deep gratitude for all his help.

I would also like to thank my on-campus adviser Greg Hager and the other members of my committee, Raman Arora and Carey Priebe, each of whom has offered advice and input at many stages of this process. Additionally, part of this research was conducted using computational resources at the Maryland Advanced Research Computing Center (MARCC), which Greg Hager provided me access to.

Hasari Tosun has been a collaborator on several papers that tie into this body of work as well, and provided many valuable contributions. I am also indebted to many other members of the Numerical Intelligent Systems Lab for their ideas and suggestions. Likewise, discussions with Joshua Vogelstein and Tyler Tomita resulted in several productive directions for work that is described in this dissertation.

This work was primarily supported by the Johns Hopkins University Computer Science Department; I will always be grateful for their willingness to let me keep going, even when the way forward wasn't always clear.

Finally, this work would not have been possible without the love and support of my family and friends, especially my wife Katy. I could not have done it alone.

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

1.1 Deep Learning

The topic of *deep learning* has seen an explosion of interest in the past few years, both in the machine learning community and in the media at large. This attention has been the result of some spectacular successes using the technique, advancing the state of the art for a number of difficult machine learning problems. Historically, deep learning has mostly been applied to computer vision problems (i.e., learning from digital images), but these days deep learning is being applied to problems in a wide range of other fields as well, including speech recognition, linguistics, bioinformatics, and game AI [5, 19, 23, 124, 139].

Deep learning developed as part of the field of *connectionist models*, as a way of harnessing the increased representational power that arises from the composition of nonlinear functions (connectionist models are discussed in greater depth in Chapter 2). By adding extra layers to a multilayer perceptron, for instance, a greater range of functions can be encoded efficiently, and more abstraction is possible. Unfortunately, this increased theoretical power has historically proven difficult to realize in practice. Simply adding lots of layers to a traditional multilayer perceptron tends to backfire, since the more layers are present, the more difficult it becomes to train the layers near the "bottom" using traditional gradient-descent based learning algorithms.

The techniques that have come to be labeled *deep learning* are methods of getting around this problem, in one way or another, enabling networks with many layers to be trained efficiently. There are now a variety of different deep learning algorithms that have been proposed, but the vast majority still fit this simple description.

Much of the original inspiration for connectionist models, including deep learning, came from studies of neurology, and in particular the study of the neurons in the primary visual cortex of the brain [89]. Early work by Rosenblatt suggested that, "A perceptron is first and foremost a brain model, not an invention for pattern recognition," [114] but this view did not persist. While there is still work being done in the field of neural modeling, the majority of modern connectionist models are *not* designed to simulate biological neural systems, and are in fact a poor model for actual neural tissue. However, many of the basic architectures still in use can trace their roots back to our understanding of brains, limited as that understanding is.

For this reason, the success of a particular connectionist model (deep learning included) is sometimes waved off as simply being a result of neuromorphic design. However, even if it were the case that a system worked because it was modeled after brain tissue, that would not be an explanation of *why* that system worked, in either the natural or the artificial context.

To an engineer who simply wants to solve a problem, this may not be a concern; so long as the tool works, the fact that it is treated as a black box is irrelevant. To such an engineer, the results speak for themselves, because results are all that matter.

To a scientist, however, this approach is unsatisfying. Science seeks not just to accomplish things, but also to understand things. The scientist wants to know *why* and *how* a tool works, and ideally *where* or *when* it will work as well.

Partly, this is driven by pure scientific curiosity; we value knowledge and understanding for their own sakes. But it is also true that a deep understanding of a thing often allows us to see new ways of improving that thing, or an entirely new purpose to which the thing may be put. There are also many contexts in which the "interpretability" of a model is important, such as cases where the purpose of the model is to help humans understand the data, where the model must be able to explain its decisions to a human, or where we need to be certain when the model will work and when it will fail. In cases like these, a black box approach may not be sufficient to meet all design goals.

In this dissertation, we will explore some of these questions of understanding as they apply to deep learning. We will attempt to understand *why* deep learning works, as well as *when* it is likely to succeed or fail; we will both review previous work on understanding deep learning and present our own work extending this knowledge. From there, we will test our understanding by trying to apply similar principles to a variety of different models, some of which are not connectionist models at all. In so doing, we hope both to improve understanding of deep learning and to offer ways to leverage that understanding in a much broader range of models than just the connectionist systems used for deep learning currently.

Some deep learning researchers have expressed the view that deep learning is fundamentally not amenable to examination and explanation, because its power comes from complex compositions of nonlinear functions and highly distributed representations [45, 55]. While not everyone agrees with this view, this idea has led to a great deal of emphasis on simply trying to tweak existing deep learning systems to obtain state-of-the-art performance on some particular problem. In that type of work, objective performance is the primary measurement of relevance.

The work we present in this dissertation comes at the problem from a different perspective; we are not seeking to improve the state-of-the-art directly, but rather to take a more “basic science” approach to understanding why deep learning works so well in the first place. Our work often focuses on comparing the relative performance of a baseline method to a modified version that incorporates a potential source of performance advantage. We often design experiments intentionally to handicap one or both methods, in order to make the comparison as fair as possible so we can be sure that any difference in performance can only result from one factor. By separating potential sources of performance advantage and examining them independently, our hope is that, in the long run, the overall state-of-the-art can be improved, but state-of-the-art performance is not our immediate goal.

1.2 Terms and Notation

While detailed definitions of many terms are provided in later chapters, our hypotheses and contributions make use of a few terms that may lead to confusion if the reader is not clear on precisely what we mean by them. This section will briefly discuss the different ways these terms are sometimes used, and what we mean when we use them in this work. A summary of our mathematical notation appears at the end of the section.

1.2.1 Machine Learning

The term *machine learning* can mean slightly different things depending on context; for the purposes of this dissertation, we will use a fairly broad definition: a machine learning algorithm is one that takes data samples as input, and generates a problem solver as an output (i.e., its output is another algorithm). This definition includes supervised and unsupervised learning, as well as semi-supervised learning and reinforcement learning. The terms *machine learning* and *statistical pattern recognition* are often used interchangeably, and indeed most machine learning techniques do rely on the existence of statistical patterns in the data.

In this work, we will use the term *machine learning* because some techniques explicitly model statistical patterns (for example, probabilistic graphical modeling), while others only implicitly rely on the existence of such patterns (for example, inductive logic programming); this can lead to confusion when using the term *statistical pattern recognition* to refer to the broader class of learning algorithms.

1.2.2 Bias

The term *bias* can also be used to mean a variety of different things. In this work, we will primarily use the notion of *inductive bias*, as well as the Bayesian statistical meaning of the word: a bias is some expectation we have *a priori*, and is often referred to as a “prior belief,” “prior probability distribution,” or just a “prior” for short. In Bayesian statistics, a model is built by combining information extracted from samples with information encoded in the bias. A bias can be useful for many things; for example, to incorporate outside knowledge from a domain expert into a model, or to help avoid overfitting.

Importantly, the statistical use of the term *bias* is distinct from the way the term is used in the social sciences to describe a form of discrimination (e.g., racial bias or *implicit bias* [104]). While there are obvious similarities between the uses, there are some important differences as well. In the social context, bias is often viewed as something that we must work to minimize or eliminate (e.g., in the context of lending or hiring practices biased by race, gender, or some other marker), even if it is actually supported by the data (e.g., income disparities between different groups may be very real, but we still want to treat them equally). When the term is used conversationally or in the media, it generally has this meaning.

From a frequentist statistical point of view, bias is simply the difference between the model and the true distribution; that is, it is fundamentally a measure of error in the model. In this context, all bias is considered to be bad, and an *unbiased estimator* is considered to be ideal. However, it is worth noting that this logic only holds under the assumption that our samples are drawn independently and identically distributed (IID) from the underlying distribution, which may not always be the case for real-world data.

The Bayesian statistical notion is a bit broader, so we must be more careful; in the Bayesian interpretation of probability, probabilistic estimates represent degrees of belief, and the prior distribution represents a way to model belief in the absence of evidence. Here, we may use the term *prior belief* rather than *bias*, but mathematically the two concepts are closely related. Some forms of prior belief are still undesirable (e.g., our beliefs are wrong, or introduce a selection bias in a sampling process), but other forms are helpful (e.g., a bias created by a domain expert to augment a sparse dataset, or to compensate for a known problem with our sampling method).

The final meaning of *bias* we will consider here is *inductive bias*; in this meaning, all machine learning relies on some sort of bias. This type of bias is frequently implicit in the technique, and is also referred to as *model bias* or *representation bias.* Rather than a source of statistical error, this type of bias can be thought of as a set of assumptions being made (usually *a priori*) about which possible solutions are worth considering, which should be preferred, and which can be ignored. Without a bias of this type, machine learning not would be possible.

1.2.3 The Need for Biases

In his seminal paper "The Need for Biases in Learning Generalizations [98]," Tom Mitchell highlighted the importance of bias by pointing out that without some form of inductive bias to restrict the hypothesis space, the basic task of machine learning is impossible. Generalization, in this context, is the ability to fit a model to some samples, and then make good predictions about future samples based on that model. For supervised learning, this means taking a set of solved examples and learning to correctly solve novel examples; the ability to solve novel examples is generalization.

What Mitchell pointed out is that there is always an infinite set of models that will correctly fit any finite set of samples. To illustrate this, consider the very simple case of two points in the Cartesian plane; there can be only one linear function that will pass through both points. But if we do not limit ourselves to linear functions, it quickly becomes clear that there are lots of other functions that will pass through these two points. In fact, we can demonstrate that the set of functions that passes through these two points is an infinite set; consider functions of the form $A \cdot \sin(B \cdot x + C) + D$. If we can find a set of values for the parameters that fit our two points, we can generate an infinite number of phase-shifted alternatives that will also fit the points by varying the parameter C alone (by adding multiples of π). We can also generate an infinite number of functions that will fit our points by varying A and B; if we vary all the parameters, we can see that we have an uncountably infinite set of functions passing through our two points, and we have not even considered the class of polynomials yet, let alone the space of all functions.

A truly unbiased evaluation is based purely on how well a model fits the data, meaning any of these infinite models that fit our points is equally "good;" they all fit our data perfectly. However, in practice some will be better than others at modeling the "true" underlying distribution our data was sampled from. Therefore, we need some way of refining our notion of "good" so we can choose a model from this infinite set; in machine learning, "good" ideally means a model that generalizes well. This preference for one model over another in spite of the fact that they both match the data equally well is the definition of *inductive bias*.

One example of a commonly used inductive bias is Occam's Razor, which is a bias towards simplicity. Occam's Razor says that we should choose the simplest model that correctly fits the data. Note that this does not necessarily mean this model is the best one possible; in particular, if we get more data, we may need to revise the model, and if our data is noisy, a naïve application (i.e., trying to fit the data perfectly) can lead to overfitting. Empirically, however, Occam's Razor has tended to be a good starting point, and it can generally be found underpinning most scientific theories.

A more concrete example of a representation bias is the type of model used to fit some data. Do we use a polynomial model, a mixture model, or a non-parametric model? Each type of model is well suited to some problems and poorly suited to others. If we choose a type of model that is poorly suited to our data, the learned model is unlikely to represent the underlying distribution accurately, and will therefore tend not to generalize well.

The observation that bias is critical to generalization (and therefore to all machine learning) is important, not just because it helps us understand what machine learning is about, but also because it suggests that we should be directly studying bias, so that we can create better learning algorithms. Mitchell's paper concludes,

> "If biases and initial knowledge are at the heart of the ability to generalize beyond observed data, then efforts to study machine learning must focus on the combined use [of] prior knowledge, biases, and observation in guiding the learning process. It would be wise to make the biases and their use in controlling learning just as explicit as past research has made the observations and their use." [98]

Our work follows this tradition of explicitly examining and making use of the biases of machine learning techniques.

1.2.4 Local vs Spatially Local

Our use of the terms *local* and *spatially local* in this work are different than the normal usage of the term *local*. A full definition of our usage can be found in Chapter 4, but we will briefly contrast the two uses here, since a misunderstanding could lead to a great deal of confusion.

The standard usage of the term *local* refers to short distances in data-space, meaning distance as measured between two feature vectors; this is the way the term *local* is used in contexts like ISOMAP or Locally Linear Embedding, where distance is frequently something like the L2 norm of the difference between two vectors in $\mathbb{R}^n$ (see Section 2.3.2.3 for more information on these techniques). When we use the term *spatially local*, we are talking about locality in feature-space, where distance is measured not between data points, but between feature sampling locations, using an approach borrowed from the field of Spatial Statistics.

In Spatial Statistics, each sample is associated with a sampling location, in addition to the sampled value. This sampling location can be thought of as meta-information, giving us a way of measuring “spatial” distance between the sampling locations of different features. In an image, for example, it is natural to use the image coordinates of a pixel as its sampling location, meaning adjacent pixels would be highly “local,” while pixels from opposite sides of the image would be “non-local.” This means that if we treat an image as a feature vector, each feature (i.e., each element of the vector) has an associated sampling location based on its pixel coordinates. *Spatially local* describes distance between features, not distance between data vectors.

a, b, x, y	scalars
$\mathbf{M}, \mathbf{D}, \mathbf{S}$	matrices or sets
$\mathbf{x}, \mathbf{y}, \mathbf{u}, \mathbf{v}$	vectors
$\mathbf{x}^\top$	transpose of vector $\mathbf{x}$
x_i	ith element of vector $\mathbf{x}$
x_{ij}	the element at row i, column j of matrix $\mathbf{X}$
$x \sim p$	x is sampled from distribution p

Table 1.1: *Conventions for mathematical notation*

1.2.5 Multi-scale and hierarchical models

Much of our work deals with breaking data down in ways that resemble the approaches taken in multi-scale learning and multi-resolution analysis. However, the term "multi-scale" is used inconsistently, and is sometimes interpreted to mean a particular type of image recognition problem, in which objects of the same class may appear at different scales (i.e. a significant degree of scale invariance is required). Similarly, several of our models could be correctly described as "hierarchical," but the term is already in use to label other types of techniques, such as Hierarchical Clustering. For terms of this type, we will attempt to be clear about when we are using them as names, and when we are using them as descriptions.

1.2.6 Notation

Key notational conventions can be found in Table 1.1.

1.3 Statement of Main Hypothesis

Our basic hypothesis is that deep learning methods like Convolutional Neural Networks (CNNs) have several built-in inductive biases that help explain their success, and that one of these biases is the assumption of *spatially localized structure* in the data.

This basic hypothesis may be fairly intuitive for many deep learning researchers; in fact, the use of spatial information is mentioned as an advantage of CNNs in some of the earliest work [81, 82]. However, it is a claim made without any evidence or citations to back it up, so we want to demonstrate that it is true before moving on. We have demonstrated instances in which other pieces of untested 'conventional wisdom' about spatial structure has proved to be inaccurate (see Chapter 6), so it is important to experimentally confirm this hypothesis before making use of it.

Once confirmed, we explore the much more interesting question of how we can add this type of bias explicitly to other techniques, and whether doing so will allow us to improve the performance of a wide variety of techniques when applied to spatial data. Here, our hypothesis is that applying a spatial bias to other techniques should be both possible and beneficial when spatial data are involved, even in the context of non-connectionist techniques.

1.4 Overview of Contributions

The major contributions to the field made in this dissertation are as follows:

- We define *spatially local structure* for use in a standard machine learning framework, and we demonstrate some methods for analyzing data to determine how much spatially local structure is present.
    - We provide both intuitive and formal statistical definitions of *local structure* with respect to features (as opposed to vectors), both in the general case and in the special case of *spatially local structure*.
    - We adapt tools from the field of Spatial Statistics for analyzing spatially local structure in image datasets.
    - We use these tools to demonstrate that several standard image datasets contain significant spatially local structure.
    - We apply a random permutation to the feature order in these datasets, and show that these permuted datasets have no spatially local structure.
- We show that Convolutional Neural Networks have a built-in assumption that their input data contains spatially local structure, and that their performance is hurt significantly if this assumption proves false.

- We demonstrate that other techniques can have similar spatially-aware biases explicitly imposed, and that doing so improves their performance when the assumption of local structure proves true, but hurts their performance when it proves false.

  - We develop a spatially-partitioned training scheme for Restricted Boltzmann Machines that takes advantage of spatial structure to allow for faster training and more accurate results.
  - We develop a partition-trained Deep Belief Network using our partitioned Restricted Boltzmann Machine training algorithm, and show that the resulting network not only has performance advantages, but also preserves spatial structure in the higher layers that is lost using traditional training methods.
  - We develop a deep, spatial version of Principal Component Analysis that takes advantage of spatial structure, even when constrained in all other ways to be comparable to standard Principal Component Analysis.
  - We develop Spatially-Biased Projective Random Forests that can take advantage of spatial structure in the data, even using a very weak form of spatial bias.

1.5 Organization

This chapter, Chapter 1, contains an introduction to the main problems addressed in this dissertation, as well as an overview of our contributions to the field. It also contains a summary of notation, and definitions of a few key terms that are important to understanding our contributions.

Chapter 2 is a review of background work that may be helpful in understanding and contextualizing our work. It contains a brief history of the fields of connectionist models and deep learning. The chapter also contains reviews of several other methods, unrelated to deep learning or connectionism, that are important to understanding the novel work described in later chapters.

Chapter 3 introduced the various datasets used in our experiments; each dataset is described, and examples are given.

Chapter 4 describes and defines *local structure* in detail, both in general and in the special case of *spatially local structure*. It introduces a framework for how structural bias can be introduced, and describes several ways in which data can be partitioned to take advantage of local structure. Finally, it explores how data can be analysed to reveal the presence (or absence) of spatially local structure. This includes an introduction to several techniques from the field of Spatial Statistics, how they can be applied to image data, and how the results are interpreted. This process is then demonstrated experimentally on several standard computer vision benchmarking datasets. Randomly permuted versions of these datasets are created and are then analyzed the same way, allowing for comparisons between data with and without spatial structure.

Chapter 5 examines the behavior of Convolutional Neural Networks in the presence and absence of spatial structure, using the data described and analysed in the previous chapters. This serves to confirm the importance spatial structure in Convolutional Neural Networks, by demonstrating how performance is adversely impacted by the absence of spatial structure.

Chapter 6 presents a spatially local training algorithm for Restricted Boltzmann Machines, and experimentally demonstrates the performance advantages of such an algorithm. It then explores the impact of using such a training algorithm in each layer of a Deep Belief Network, demonstrating not only quantitative performance gains, but qualitative differences in spatial structure in hidden layer activations.

Chapter 7 presents a non-connectionist model constructed using the principles of spatial data partitioning. The model is a simple one based on Principal Component Analysis, ensuring that the experimentally demonstrated performance advantage of the spatially local model can have no source other than the spatial structure of the model. We also show that model using local structure relies on the presence of local structure for its performance, while the baseline model does not.

Chapter 8 presents a second non-connectionist model, this one based on Random Forests. This model uses a different mechanism to introduce a spatial bias than those described in previous chapters, and is radically different from traditional deep learning algorithms. In spite of this, we are still able to demonstrate that the spatially biased version has a performance advantage on data with spatial structure.

Chapter 9 concludes the dissertation, and contains an overview of what we were able to conclude from our analysis and experiments. It again summarises our contributions to the field, this time with reference to the work presented in the previous chapters. Finally, it describes some of the many ways our work could be extended and built upon in the future.

Chapter 2

Background

In this chapter, we cover many existing techniques and concepts that are useful in understanding our work. Sections relating to techniques with which the reader is already knowledgable can safely be skimmed, or skipped entirely. Alternatively, the chapter can be bypassed entirely, and then used as a reference if things become confusing later.

We begin the chapter with a review of the connectionist models that standard deep learning uses as building blocks. We then give a brief history of connectionism and deep learning. Finally, we cover relevant non-connectionist techniques.

2.1 Connectionist Models

Connectionist models are generally represented as graph structures, with nodes (or *units*) representing integrating elements and edges (or *connections*) representing connection strengths. When parallels are drawn to biological neural tissues, the nodes represent neurons, and the edges represent synapses.

In general, a node takes the intensity values from its incoming edges and computes a function based on those values, the output of which is the *activation level* of that node. This function is called an *activation function* or an *aggregation function.* The activation is then put onto each outgoing edge, where it is multiplied by the weight associated with that edge. In typical network operation, some nodes will have their values clamped (*input nodes*), and some nodes will have their values examined (*output nodes*). Other nodes in the network are called *hidden nodes*, because their activations are neither set nor examined externally.

Typically, an activation function is some kind of nonlinear squashing function applied to the sum of the inputs (see Figure 2.1). The Heaviside step-edge function was used in the earliest networks, with a suitable threshold to give a binary output; later networks used sigmoid activation functions, which can be viewed as a smoothed version of the Heaviside function. The activation function, frequently labeled $g(\cdot)$, takes as input an activation vector $\mathbf{x} \in \mathbb{R}^n$ and a weight vector $\mathbf{w} \in \mathbb{R}^n$. Using this

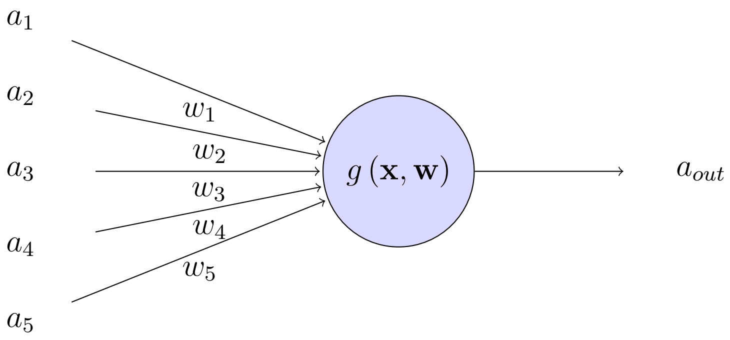


Figure 2.1: *An example connectionist node with five incoming edges and one outgoing edge, with an activation function $g(\cdot)$.*

notation, one possible sigmoid activation function is the logistic function:

$$g(\mathbf{x}, \mathbf{w}) = \frac{1}{1 + e^{-\left(\sum w_i x_i\right)}}$$

The *architecture* of a connectionist model is a description of the network graph topology; i.e., how many nodes are there, and which ones are connected by edges. There are many types of connectionist model architectures; here, we will primarily concern ourselves with the subset known as *feed-forward networks*. A feed-forward network is an acyclic directed graph, where the input nodes have no incoming edges and the output nodes have no outgoing connections. In this type of network, nodes are often organized into *layers*, where a layer is a subset of nodes that are all the same hop-count distance from the input nodes. Layers are typically indexed up from the input nodes, so nodes in the first hidden layer have incoming edges from input nodes, nodes in the second hidden layer have incoming edges from nodes in the first

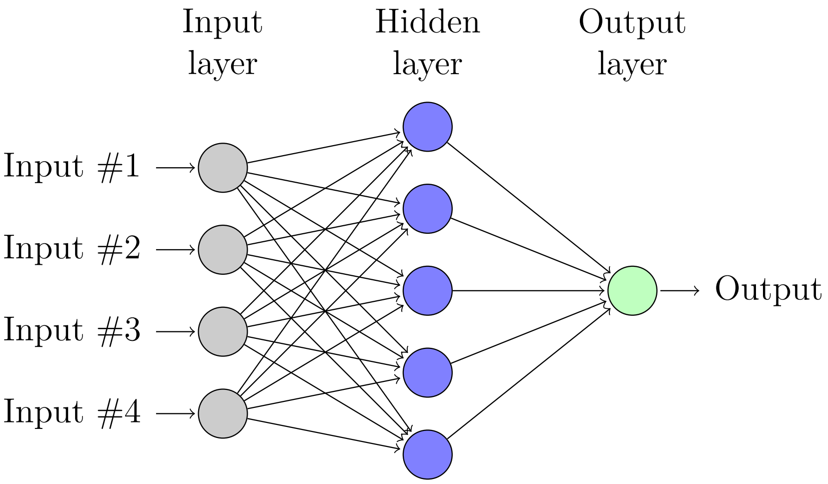


Figure 2.2: *An example fully connected feed-forward network with four inputs, five hidden nodes, and one output.*

hidden layer, and so forth. Activation values can then be computed one layer at a time, starting with clamped values at the input nodes and propagating activations up to the output nodes. A network in which every node in each layer is connected to every node in the layer above is known as a *fully connected* feed-forward network (see Figure 2.2).

2.1.1 Types of Connectionist Models

A *perceptron* is a fully-connected feed-forward connectionist network that uses a linear activation function with a threshold to yield a binary output. It generally does not contain more than one hidden layer.

A *multilayer perceptron* (MLP), on the other hand, uses a continuous, nonlinear activation function (often a logistic or hyperbolic tangent function); it is still a fully-connected feed-forward network and almost always contains one or more hidden layer.

An *artificial neural network* (ANN) in general is a broader term for this type of connectionist model; it is not restricted in architecture or activation function, but is generally only used to describe directed-graph networks.

2.1.2 A Brief History of Connectionist Models

The field that has become known as connectionist modeling derives originally from work done by McCulloch and Pitts, who were trying to understand the vision system in the brains of animals. Their first major contribution was the 1943 paper [95] that presented a simple mathematical model of the behavior of neurons, based on the observed fact that neuronal activity was based on discrete electrical impulses, which meant that communication was essentially binary in nature. This binary nature allowed them to apply a logic-based interpretation, though the authors do point out that the logical behavior at a high level does *not* describe or explain the underlying functionality of actual neurons (in particular, they note that actual neurons learn using what amount to continuously variable parameters, which their model does not fully for). While the model had some limitations, it was the first attempt to describe how neural tissue could perform formal computations.

Their second major contribution was the 1959 paper [89] that examined the functional characteristics of the visual system of frogs, using a combination of electron microscopy to determine numbers and types of cells, and fine electrical probes to examine electrical activity of axon bundles in the brains of living frogs (the frogs were sedated and had a small flap of bone opened to allow the probes to be used, but were otherwise normal). By exposing the frogs to a variety of different visual stimuli while recording the electrical activity in various different cells, they were able to determine that the frog's visual system was *not* presenting the brain with a set of point-based measures of light intensity, but rather was sending the brain a sparse set of signals that represented complex functions of the actual visual stimulus. In effect, the frog's visual system presents the brain with simple features based primarily on movement (that is, change in the visual stimuli) which allows it to respond to prey-like visual stimuli and predator-like visual stimuli, but is largely insensitive to static visual stimuli, regardless of what those stimuli are. The realization that the neural pathways coming from the retina represented highly complex functions of their inputs was an unexpected one, and led to a new way of thinking about the role of visual neurons.

At around the same time, Rosenblatt published the 1957 paper [113] that introduced the term *perceptron*. The perceptron was the first practical application of the logical model developed by McCulloch and Pitts to solve perceptual problems. The perceptron, as originally conceived, was a custom-designed electronic machine that used a grid of photoreceptors as input and a set of potentiometers that could be adjusted by motors during training. The original system was able to learn to recognise simple patterns, and this success was met with wild media speculation about what the system would be able to accomplish. This speculation was encouraged by Rosenblatt's own claims about the revolutionary power of his system, leading to predictions that human-like artificial intelligence would soon be possible.

Unfortunately, the actual system, while interesting, was fairly limited in the types of patterns it could recognise. This led to problems when it failed to fulfil the grandiose claims that had been made early on, and resulted in a spike of early interest and funding followed by a backlash in which interest and funding largely dried up for over a decade (an event often referred to as the "AI winter"). This is often said to have been the result of the 1969 publication of a book by Minsky and Papert [96], though there is some debate over the extent to which this was the intended interpretation of their work.

The main failing highlighted in the book was that a perceptron with only one layer of hidden nodes is a linear classifier, and can therefore only be trained to classify linearly separable problems. Since many problems of interest are not linearly separable (including XOR and parity), this is a significant drawback. The power of perceptron-style networks to encode non-linear functions was actually described in the original 1943 paper [95], but it requires more complex architectures. Rosenblatt's algorithm for learning the connection weights was not able to handle networks with more than two hops between an input and an output, meaning that a single hidden layer was all that could be used effectively.

The solution to this was a training algorithm that allowed networks with more than one layer to be trained. The algorithm is known as *error backpropagation* [116], and it works by applying the chain rule to derive updates for connection weights based on how much they are responsible for an incorrect output. For the chain rule to be used, however, the weight aggregation function must be continuously differentiable; the step-edge Heaviside functions used in early perceptrons were not. By using a sigmoid function instead, connection weights could be learned for networks with multiple levels of hidden nodes. This allowed for much more complex problems to be solved, because the sigmoid function, which can be thought of as a step-edge with a smoothing kernel applied, has the added benefit of providing a smoother fitness landscape for gradient descent.

The math underlying this algorithm for continuous functions was derived in the 1960s [13, 14, 68], but it was not actually applied to perceptron weight learning until later. Werbos initially suggested such a possible application in his 1974 dissertation [140], followed by a basic application a few years later, but the 1986 publication by Rumelhart and Hinton [116] is generally considered to be the first example of modern backprop.

The combination of the error backpropagation algorithm with the increased computational resources that became available during that time period led to a renewal of interest in the area of artificial neural network research that has continued to the present.

2.1.3 Autoencoders and RBMs

Autoencoders and RBMs are two standard methods used as building blocks for deep learning. Both methods allow a distributed representation to be learned from input data in an unsupervised fashion; connectionist models are used to encode these representations. Once trained, this representation can then be used to train another model, which takes the output of the first model as its input. This process is repeated, and the components are stacked to form a deep network (see Section 2.2.3). Note that, while the two techniques are different, the encoding networks they produce can often be used interchangeably, which can lead to confusion (especially when deep autoencoders use RBMs to pre-train their layers).

2.1.3.1 Autoencoders

An *autoencoder* [55] is a type of feed-forward neural network that is trained to predict its own input; that is, the training signal for the output nodes is the same as for the input nodes. One or more fully-connected hidden layers are typically used, generally with fewer nodes in the hidden layer(s) than in the input and output layers (see Figure 2.3). Since it is trained to minimise reconstruction error, the optimization task is essentially one of compression or dimensionality reduction. Sometimes, other optimization criteria may be imposed as well, such as sparseness of encoding, which can lead to more neurologically plausible results [103]. Alternatively, by stochastically dropping some inputs, a *de-noising autoencoder* can be trained to reconstruct full vectors from lossy ones [138].

Autoencoders with many hidden layers are generally "pre-trained" one layer at a time, from the bottom up; the first hidden layer is trained to reconstruct the inputs, then the weights between the inputs and first hidden layer are fixed, and the second hidden layer is trained to reconstruct the first hidden layer activations, and so forth. Once all layers have been pre-trained, a final pass of tuning all the weights using standard gradient descent can be performed; the pre-training allows the algorithm to be initialized in a sufficiently good local basin to achieve high performance.

Once trained, the hidden layer activations can be used as an encoded representation of the input, much like the output of any other dimensionality reduction or feature extraction technique. For an input vector $\mathbf{x} \in \mathbb{R}^n$, a trained autoencoder can be treated as an encoding function $f(\mathbf{x}) \to \mathbf{h}$, where $\mathbf{h}$ is the activations of the hidden nodes. For a single hidden layer made up of m nodes with a $n \times m$ weight matrix $\mathbf{W}$ and a bias $\mathbf{b}$, the mapping is $f(\mathbf{x}) = g(\mathbf{W}\mathbf{x} + \mathbf{b})$, where g is an aggregation function.

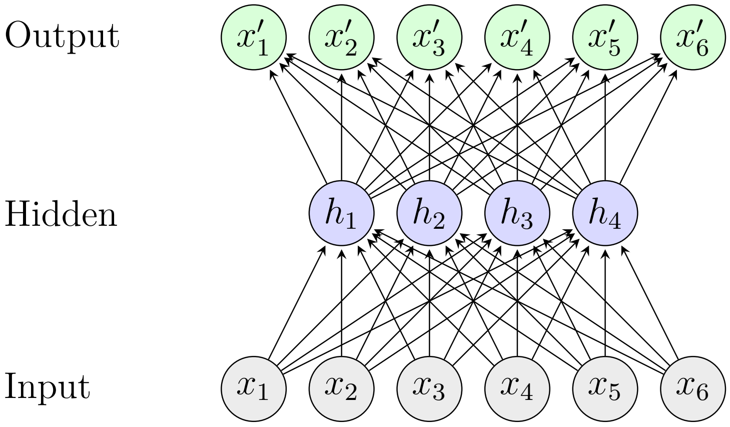


Figure 2.3: *Basic architecture of an autoencoder network; the network is trained to minimize the difference between the inputs* $\mathbf{x}$ *and the outputs* $\mathbf{x}'$*.*

As with MLP networks, sigmoid functions such as logistic or hyper-tangent are the most common choice for aggregation function. If a linear function is used instead of a sigmoid, then a standard autoencoder tends to converge to the same encoding as PCA does (see 2.3.2.1 for details on PCA). However, the ability to use non-linear activation functions gives autoencoders more representational power than traditional PCA (which is a linear technique).

2.1.3.2 Restricted Boltzmann Machines

A Restricted Boltzmann Machine (RBM) is type of connectionist model designed to optimize for reconstruction error, much like an auto-encoder. While RBMs can be used as a type of auto-encoder, they are based on quite different principles from the more traditional perceptron-style auto-encoders described above.

The RBM model was first proposed as a simplification of a full Boltzmann Machine [31] by Smolensky in 1986 [126] under the name Harmonium. A Boltzmann Machine is a probabilistic version of a Hopfield network, meaning it is primarily auto-associative in nature. The goal is to learn patterns inherent in a set of input samples by building connections that encode those patterns. However, where classic Hopfield networks and Boltzmann Machines are uniform in construction (i.e., all nodes are equal), an RBM has two distinct types of nodes: *visible nodes*, and *hidden nodes*.

The distinction is that *visible nodes* correspond with features of the observed samples, where *hidden nodes* do not. Much like with other connectionist models, hidden nodes end up encoding a complex, distributed representation of the visible nodes; in effect, they become the model representation of the data.

Unlike a perceptron, in an RBM there is no distinction between *input* and *output* nodes, but rather the *visible nodes* function as both. An RBM is therefore represented as an undirected graph. The visible nodes and the hidden nodes are normally fully connected, but there are no connections between the nodes in each category, so the result is a complete bipartite graph. This constraint is the “restriction” which differentiates RBMs from classic Boltzmann Machines. We will use $\mathbf{x}$ to denote visible nodes, and $\mathbf{h}$ to denote hidden nodes; see Figure 2.4 for an example RBM.

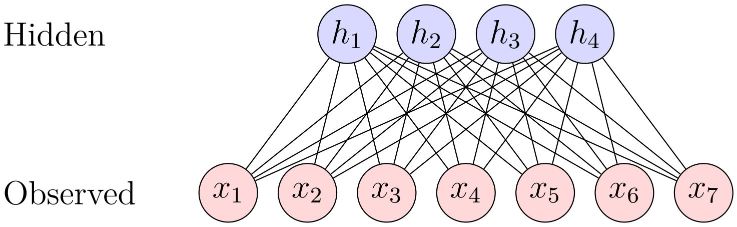


Figure 2.4: *Basic architecture of a Restricted Boltzmann Machine with seven observable nodes* $\mathbf{x}$ *and four hidden nodes* $\mathbf{h}$*.*

The key insight (and the origin of the name) comes from the fact that this type of model can be represented as a Boltzmann energy distribution [88]. The Boltzmann energy distribution is a concept developed by physicists to describe a system of particles, where it models the likelihood of the system being in any given configuration as a function of the amount of energy the system has in that state, as well as the temperature of the system.

RBMs make use of the math but detach it from its original physical meaning, instead taking advantage of the fact that we can use a Boltzmann distribution to efficiently learn a set of parameters (i.e., edge weights in the graph) that will cause the sample data to be produced with high likelihood. The equation for

$$p(\mathbf{x}, \mathbf{h}) = \frac{e^{(-E(\mathbf{x},\mathbf{h}))}}{Z},$$

where E is the energy of the state, and Z is the partition function, which defines configurations over all possible states:

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

The conditional probability can be written in terms of the energy function as follows:

$$p(\mathbf{h}|\mathbf{x}) = \frac{e^{(-E(\mathbf{x},\mathbf{h}))}}{\sum_{\mathbf{h}} e^{(-E(\mathbf{x},\mathbf{h}))}}$$

The probability of data $p(\mathbf{x})$ is then obtained by marginalizing over the hidden vector $\mathbf{h}$:

$$\begin{aligned} p(\mathbf{x}) &= \sum_{\mathbf{h}} p(\mathbf{x},\mathbf{h}) \\ &= \sum_{\mathbf{h}} \frac{e^{(-E(\mathbf{x},\mathbf{h}))}}{Z} \end{aligned}$$

Calculating $p(\mathbf{x},\mathbf{h})$ exactly is not tractable due to the partition function, Z; however, the conditional probability, $p(\mathbf{h}|\mathbf{x}) = p(\mathbf{x},\mathbf{h})/\sum_{h'} p(\mathbf{x},\mathbf{h}')$, has a rather simple form. To differentiate from the current vector $\mathbf{h}$, we use $\mathbf{h}'$ to represents all possible hidden vectors (configurations) of size n. Then given our definition of the Boltzmann distribution, we obtain

$$p(\mathbf{h}|\mathbf{x}) = \frac{e^{(\mathbf{h}^\top \mathbf{W}\mathbf{x}+\mathbf{b}^\top \mathbf{x}+\mathbf{c}^\top \mathbf{h})}/Z}{\sum_{h' \in \{0,1\}^n} e^{(\mathbf{h}'^\top \mathbf{W}\mathbf{x}+\mathbf{b}^\top \mathbf{x}+\mathbf{c}^\top \mathbf{h}')}/Z}.$$

Although an algorithm for training RBMs was known in the early 1980s [31], it did not scale well on the hardware available at the time. RBMs began to gain popularity 2002, when Hinton *et al.* developed Contrastive Divergence, a more efficient training method based on Gibbs Sampling [54]. Since then, RBMs have been used widely as basic components of deep learning systems [9, 55, 56]. RBMs have also been applied successfully to classification tasks [22, 76, 92], as well as other learning tasks such as Collaborative Filtering [118].

The Contrastive Divergence (CD) method provides a reasonable approximation to the likelihood gradient of the energy function. Algorithm 2.1 shows pseudocode for training RBMs using a one step Contrastive Divergence method. The algorithm accepts a sample data instance and a set of model parameters: weight vector ($\mathbf{W}$), visible layer bias vector ($\mathbf{b}$), hidden layer bias vector ($\mathbf{c}$), and learning rate (α). It updates the model parameters in two phases, referred to as the *positive* phase and the *negative* phase respectively.

First, in the positive phase (lines 1-2), the probability of the hidden node is calculated for all hidden nodes, given the visible vector. In the negative phase (lines

Algorithm 2.1 Pseudocode for one-step contrastive divergence, CD-1($\mathbf{x}_i$, α)

Input: $\mathbf{x}_i$: data sample, α: learning rate.

Output: $\mathbf{W}$: weight vector, $\mathbf{b}$: visible node bias vector, $\mathbf{c}$: hidden node bias vector

```
    Positive Phase:
1:  $\mathbf{x}^0 \leftarrow \mathbf{x_i}$
2:  $\mathbf{h}^0 \leftarrow \sigma(\mathbf{c} + \mathbf{W}\mathbf{x})$
    Negative Phase
3:  $\tilde{\mathbf{h}}^{\mathbf{0}} \sim \mathbf{p}(\mathbf{h}|\mathbf{x}^{\mathbf{0}})$
4:  $\tilde{\mathbf{x}} \sim \mathbf{p}(\mathbf{x}|\tilde{\mathbf{h}}^{\mathbf{0}})$
5:  $\mathbf{h}^1 \leftarrow \sigma(\mathbf{c} + \mathbf{W}\tilde{\mathbf{x}})$
    Update parameters:
6:  $\mathbf{b} \leftarrow \mathbf{b} + \alpha(\mathbf{x}^{\mathbf{0}} - \tilde{\mathbf{x}})$
7:  $\mathbf{c} \leftarrow \mathbf{c} + \alpha(\mathbf{h}^{\mathbf{0}} - \mathbf{h}^{\mathbf{1}})$
8:  $\mathbf{W} \leftarrow \mathbf{W} + \alpha(\mathbf{h}^{\mathbf{0}}\mathbf{x}^{\mathbf{0}} - \mathbf{h}^{\mathbf{1}}\tilde{\mathbf{x}})$
```

3-5), the probability of each hidden node is determined by sampling from the model. First a sample of points for the hidden nodes is drawn based on the current estimate of the distribution $\mathbf{p}(\mathbf{h}|\mathbf{x}^{\mathbf{0}})$ (line 3). Using these sampled points $\mathbf{h}^{\mathbf{0}}$, the current estimate of $\mathbf{p}(\mathbf{x}|\mathbf{h})$ is used to sample points for the visible nodes $\mathbf{x}$ (line 4). Finally, on line 5, the probabilities of the hidden nodes are updated based on the sampled vector for the visible nodes. The parameters of the network are updated on lines 6-8. Contrastive Divergence need not be limited to one forward and backward pass in this matter, and Algorithm 2.1 can be extended by creating a loop around lines 1-5. Then for $k > 1$, the positive and negative phases are repeated k times before the parameters are updated.

The CD-1 algorithm (i.e, Contrastive Divergence with one step) has proven to be sufficient for many applications [8, 133]. CD-k is rarely used, because resetting the

Markov chain after each parameter update is inefficient (as the model has already changed [133]). As an alternative, Tieleman modified the Contrastive Divergence method by making Markov chains persistent [133]; one or more persistent chains are kept during training, leading to greater efficiency for the multi-step case. Even so, many applications have demonstrated minimal (if any) improvement in performance using persistent Markov chains.

2.2 A Brief History of Deep Learning

While deep learning has become wildly popular only recently, it has been an area of active study for over 35 years. In fact, the desire for “deep” connectionist networks can be traced back to the first models developed by McCulloch & Pitts [95]. From the beginning, the goal was to build a model of the primary visual cortex, which was already known to exhibit a deep hierarchical structure by the 1950s. The reason early models were “shallow” is simply that shallow models are much easier to train than deep ones. It was the lack of an ability to train connection weights for multi-layer networks, pointed out by Minsky & Papert [96] in the late 1960s, that is frequently blamed for the decline of interest (and funding) for connectionist models that took place in the following decade.

The first solution to this problem was the error gradient backpropagation training technique, which used a continuously differentiable activation function in combination with the chain rule to derive "error" values for non-output nodes. This technique proved very successful for networks with a layer or two of hidden nodes, allowing networks to be trained to solve non-linear problems. However, as more hidden layers were added, gradient diffusion and overfitting resulted in a failure to converge to a good solution. This is often referred to as the problem of *vanishing* or *exploding gradients*. The chain rule involves taking the product of node activations to derive an error gradient for hidden nodes, so the deeper the network is the closer the products will get to zero (if the activation function is always less than one) or sometimes $\pm\infty$ (if the activation function can be greater than one).

There have been attempts to work around this problem going back many years, with one of the earliest successes being Fukushima's Neocognitron [39]. For most of the intervening time, interest in the field was relatively small, with the majority of publications coming from the labs of Hinton, Bengio, and LeCun. Convolutional Networks [82] and Deep Belief Networks [53] are the two primary deep network types that have been studied significantly. There have been other deep networks models proposed [6, 42], but they have not had as significant an impact on the field.

All of these techniques have offered ways to minimize or sidestep the problem of vanishing gradients. Additionally, they all share the same neuromorphic approach, basing their functionality on the neuroanatomy of the primary visual cortex (some more loosely than others), and they were all initially targeted at computer vision problems.

While this history goes back over 35 years, deep learning did not receive much attention until recently. In fact, the term “deep learning” did not come into common usage in the machine learning community until around 2008. The explosion of interest we have seen since then was sparked largely by the success of Convolutional Networks and Deep Belief Networks on computer vision problems. In the last few years, Convolutional Networks have become the dominant deep learning paradigm for many types of real-world problems, but other techniques are also still studied and used.

2.2.1 Neocognitron

The first model which was “deep” in the modern sense of the word was Fukushima’s Neocognitron [39] (which predated Rumelhart’s introduction of backprop). Not only is the system continuing to be studied today [40], but it has also been highly influential in the design of many other deep learning techniques. It was explicitly designed to be an abstract model of primary visual cortex (V1) neural tissue in animals as it was understood during the 1960s (see [62, 63, 75, 89]). It is worth noting that more recent neurological studies have given us a significantly more detailed understanding of some of these processes; here, we will only concern ourselves with the work that most heavily influenced the design of the Neocognitron network, and even then we leave out much of the neural complexity that was not modeled in the network.

In abstract terms, Neocognitron is a connectionist network composed of two distinct types of nodes, called S-cells and C-cells. A full network consists of alternating layers of the two types of nodes in a feed-forward architecture. The layers are not fully-connected, but instead use a restricted connection scheme.

2.2.1.1 Cell Types

S-cells are modeled after a class of neurons in the primary visual cortex (V1) called ‘simple’ cells [63]. A simple cell acts as a type of detector; its activity is normally low, but spikes when it is presented with an input that matches the particular pattern that cell is looking for. In the visual cortex, the first layer of simple cells are connected to the retina through the lateral geniculate body (LGB) via retinal ganglion cells each of which is activated by either a bright spot (‘on’-center) or a dark spot (‘off’-center) in a small, localized region of the retina [75]. Each simple cell has a “receptive field” that characterises what part of the retina can impact its activity (see Figure 2.5). Each cell also has a pattern of light and dark that, when applied to its patch of retina, will cause a spike in its response. Simple cells have both excitatory and inhibitory regions; this allows them to respond to very specific patterns, such as oriented edges, or bands of a

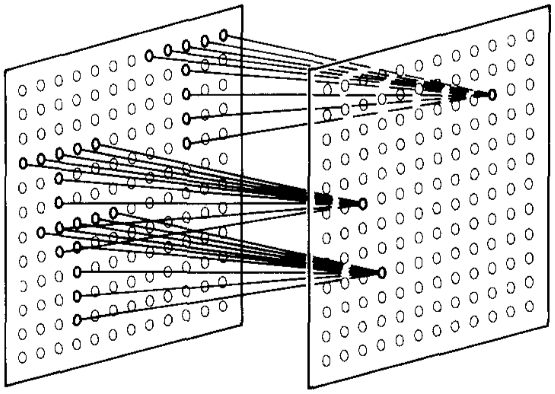


Figure 2.5: Illustration showing the input interconnections to the cells within a single cell-plane, adapted from [39]

particular width [62]. Each cell responds only to a single pattern, and then only when the location of that pattern on the retina corresponds with the location of the pattern in the receptive field of the cell. Hubel & Weisel described them as 'simple' because the patterns they respond to are fixed, and fairly easy to characterize, but they can still respond to much more complex patterns than the simple center/surround spots that activate retinal ganglion cells.

C-cells are modeled after a type of 'complex' cell [63], which are the other primary class of cells described by Hubel & Wiesel. The essential difference is that 'complex' cells respond to patterns that cannot be easily characterised by a simple spatial form. For example, a cell that activates primarily based not just on a shape, but on having that shape move in a certain way, or a cell that does not activate in the presence of a stimulus, but produced a brief spike in activity when that stimulus is removed. The particular type of complex cell modeled in a Neocognitron is one that responds to a

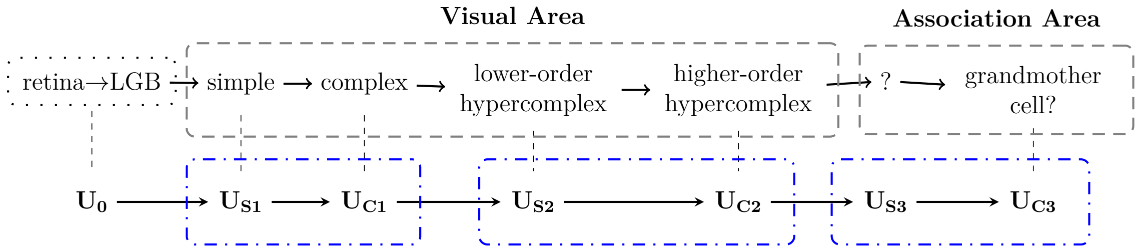


Figure 2.6: *Correspondence between the hierarchy model by Hubel and Weisel, and the neural network of the neocognitron, adapted from [39]*

pattern with a degree of spatial invariance; that is, it is looking for a pattern that is smaller than its receptive field, and activates when that pattern is present regardless of the specific location of the pattern within the overall receptive field.

Hubel & Weisel also describe 'lower-order hypercomplex' cells and 'higher-order hypercomplex' cells, which behave similarly to 'simple' and 'complex' cells respectively. The distinction is that they use the activations of 'complex' cells as their inputs, rather than using input directly from the retina. For the purposes of Neocognitron, these are treated as being just another set of S-cells and C-cells, the distinction being only whether the inputs come from raw pixel intensities or from the outputs of a previous stage of the network. The correspondence between cell types in the Hubel & Weisel model and a Neocognitron network is show in Figure 2.6.

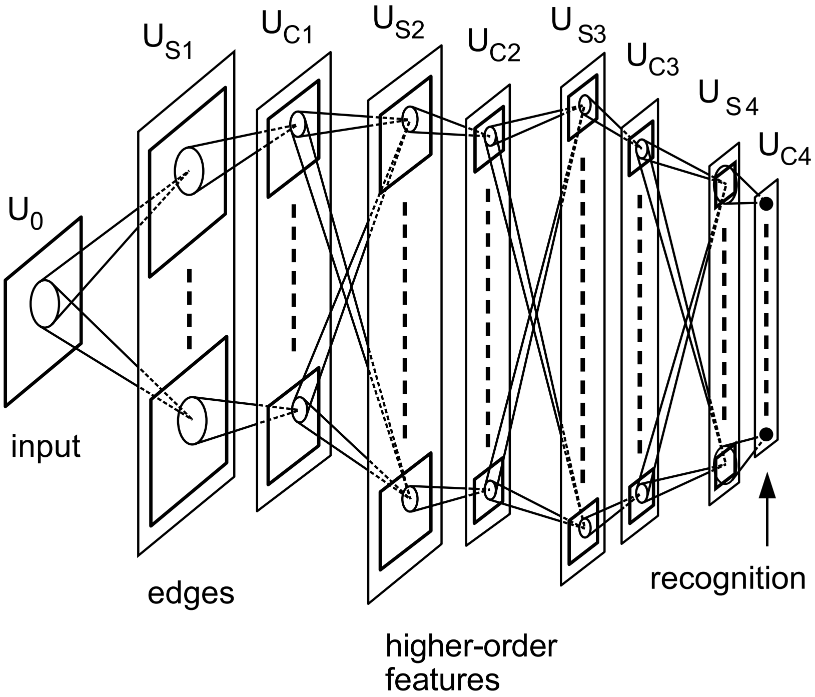


Figure 2.7: *The architecture of a Neocognitron, adapted from [40]*

2.2.1.2 Architecture

The overall architecture of a Neocognitron network consists of a series of "modular structures," each of which consists of a layer of S-cells followed by a layer of C-cells. The input layer is a 2D array of units treated as a binary image.

Within a layer, cells are grouped into "cell-planes," where all the cells in a plane have receptive fields that respond to the same pattern, with the difference being only the region of the input that they consider. This can be thought of as each "cell-plane" representing the output of a single local feature-detector applied at each position in the input image (see Figure 2.7).

The numerical operation of the C-cells is fairly simple; each C-cell receives input from all S-cells in a fixed size region of the preceding layer. The weights are fixed, and defined in such a way that activation of any of the inputs is sufficient to activate the output; in effect, the C-cell performs a logical AND on its inputs. Since the weights are fixed, no learning takes place in these layers.

The S-cells are somewhat more complicated; they receive input from a region in the same way as the C-cells, but their output function is based on learned connections. The mathematical model is based on the Hubel & Weisel model of 'simple' cells, meaning it involves a pattern of excitatory and inhibitory connections that combine to yield a cell that responds only to a particular pattern.

2.2.1.3 Training

Training a Neocognitron network (described as "self-organization") is an unsupervised Hebbian learning process, and works by using a form of competitive learning to incrementally update the weights in the network. After each presentation of an input, a few "representative" S-cells are picked in each S-layer based on which cells show the highest activation for that input. The selection is competitive, meaning only one cell can be selected in any given cell-plane. Weakly activating cell-planes may have no representatives chosen, in which case their weights will not be updated. As with other forms of competitive learning, the chosen unit is updated to make it respond even more strongly and specifically to the pattern that was presented by increasing the excitatory value of connections where the pattern is high, and increasing the inhibitory value of all other connections to compensate (i.e., so that uniform input continues to produce a low output value).

No supervised learning is done; the network is trained purely as an (unsupervised) feature extractor. While this is partly because the intention was to model the self-organizational pattern discovery seen in neural tissue, it is also worth remembering that the error backpropagation algorithm had not yet been applied to neural networks when the Neocognitron model was first described.

2.2.1.4 Applications

In its earliest versions [39], the Neocognitron was able to learn clusterings that corresponded very closely to the classes of a 5-class input problem. The problem used was in effect a very small digit recogniser, mapping images of the numbers 0-4 to distinct units in the highest layer. This was robust to small changes in scale, placement, shape, and additive or subtractive noise, meaning that the same output unit would activate regardless of these changes to the input.

What made this result impressive was the fact that an unsupervised system had effectively learned a set of classes that corresponded well with the concept labels a human would apply to the same data. However, the approach had difficulty scaling;

when applied to a 10-digit input set, the network proved to be very sensitive to initialization and parameter settings, and did not always learn to distinguish between all ten digits. The computational limitations of the time and the complexity of the algorithm meant that it was highly limited in terms of practical applicability, but the invariance it captured was something that has historically been a weak point of most machine learning techniques, including less structured feed-forward networks such as perceptrons.

2.2.2 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) were originally developed by LeCun as connectionist networks for handwritten-digit classification as a part of his dissertation work [79]. The most recognizable form is probably that described in 1998 [82] as LeNet5, though earlier versions had at least some of the same characteristics a decade prior [81].

A CNN can be viewed as a fairly straightforward translation of a Neocognitron architecture to use more standard feed-forward perceptron style unit activation functions. The primary advantage of doing this lies in the error-backpropagation rule which had recently been developed [116] for this type of network; by using perceptron style units, LeCun was able to train the network using supervised learning. This made CNN style networks more practical for real-world tasks such as handwritten

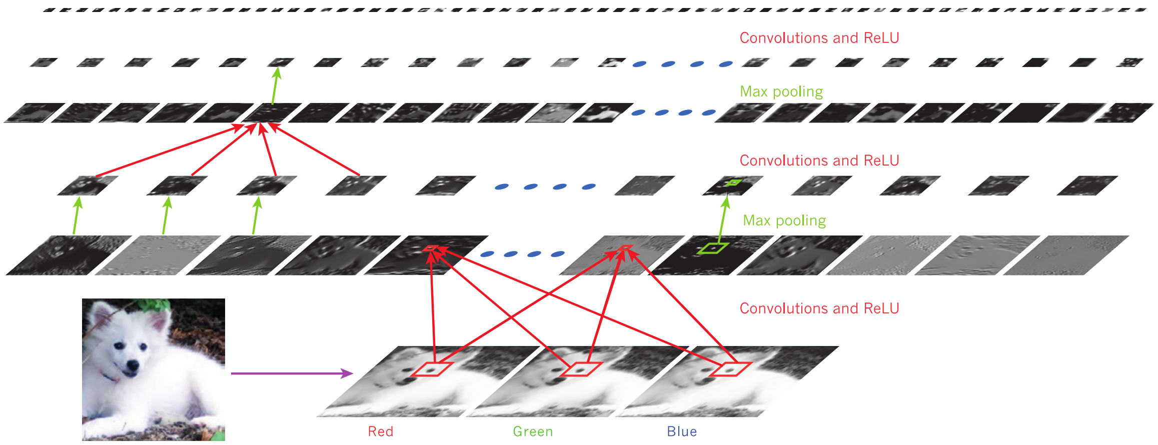


Figure 2.8: Architecture of a convolutional network. *The outputs (not the filters) of each layer (horizontally) of a typical convolutional network architecture applied to the image of a Samoyed dog (bottom left; and RGB (red, green, blue) inputs, bottom right). Each rectangular image is a feature map corresponding to the output for one of the learned features, detected at each of the image positions. Information flows bottom up, with lower-level features acting as oriented edge detectors, and a score is computed for each image class in output. ReLU, rectified linear unit. Adapted from [80].*

digit recognition; one of the big early successes for CNNs was automatic processing of zip-codes (for mail) and check values (for banks). Such systems were in use by banks and post offices long before the modern use of CNNs for more difficult computer vision tasks [82].

Where Fukushima was originally focused on creating plausible models of the primary visual cortex in biological systems, LeCun was more interested in the ability of the system to solve real-world problems. The inspiration for the two models was the same, however, and LeCun's work follows from Fukushima's (and Hubel & Weisel's) quite clearly. The language used to describe the two models is different, however, even when it is describing functionally equivalent concepts.

For instance, the CNN literature tends to use the term “unit” where the Neocognitron literature uses the term “cell,” but the meaning of these terms is basically the same. The feature-extracting “S-cells” become “convolutional units,” and the resolution-reducing “C-cells” become “sub-sampling” or “pooling units” (we will tend to use the latter term, since it is more common in use today, but earlier work uses the former term to describe the same part of the network). What Fukushima referred to as a “cell-plane” is called a “feature map” by LeCun, but is architecturally identical. What Fukushima calls the “response field” of a unit is frequently referred to as a filter or a kernel.

Besides linguistics, there are two big differences between the early Neocognitron models and the early CNN models. First, CNNs used the sigmoid activation functions typical of fully connected feed-forward networks, rather than the more directly neuromorphic activation functions used in Neocognitrons. Second, early CNNs included trainable parameters for both convolutional units and pooling units; in Neocognitrons, only the S-cells had trainable parameters.

2.2.2.1 Convolutional Layers

Units in a convolutional layer are organized in the same basic way as S-cells in a Neocognitron; units are organized into groups based on weight-sharing. Each of k feature extractors (the weights for which are learned during training) will have a number of associated units which taken together are called a feature map; the complete convolutional layer will be made up of k feature maps (the size of each feature map is the same as the size of the input to the layer, minus the width of the feature extraction filter, since the entire filter must be inside the image boundaries). Each feature map effectively contains the response of the associated feature extractor applied at each possible location in the input image. LeCun observed that the process of calculating the activations for the units in a feature map was mathematically equivalent to convolving the input with the weight matrix for the feature extractor in question, which is where the name of the technique came from. Just like a Neocognitron, units within a layer of a CNN are organized so as to correspond spatially with the layout of the inputs below them; that is, units which are adjacent in a feature map are generated by applying the feature extractor at adjacent locations in the input.

The activation of a convolutional unit is computed simply as the product of the inputs and their corresponding connection weights, added together and then fed through a squashing function. In early work, this was a standard logistic or hyperbolic tangent function (generally with fixed scaling parameters set to encourage solutions in the plastic region of the function) [82]. In more recent work, great success has come from using alternatives here, particularly rectified sigmoid [85, 108], and rectified linear units (ReLU) [100] or the similar but smoother SoftPlus [29]. Many formulations will also incorporate parameters for the amplitude and slope (i.e., multipliers that al-

$$\text{Logistic} \equiv \frac{1}{1+e^{(-a)}}$$
$$\text{Hypertangent} \equiv \tanh(a)$$
$$\text{Rectified linear} \equiv \max(0,a)$$
$$\text{SoftPlus} \equiv \log(1+e^{(a)})$$

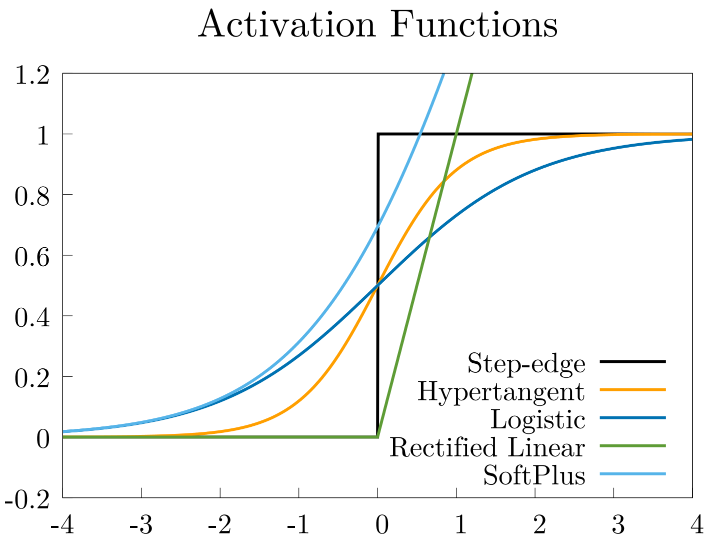


Figure 2.9: *Some possible activation functions that can be used in neural networks, where 'a' is the weighted sum of the incoming activations. They are sometimes scaled vertically or horizontally depending on application.*

low the functions to be scaled vertically and horizontally respectively), but these are generally constants set by a designer rather than parameters learned by the network. They can be used to encourage the network to find solutions in the plastic region of the functions. This can be desirable because trying to drive the network to produce a value that can only be achieved in the limit as the input goes to ±∞ leads to extreme weight values, which has been shown to hurt the generalization ability of a network. See Figure 2.9 for a comparison of several common activation functions.

Particularly for the "rectified" functions, there are a number of alternate functions that approximate them (one example is given above), as well as variations such as "noisy" [100] or "leaky" [93] rectifiers. Generally, this is an attempt to retain the advantageous properties but with a function that produces a nicer gradient to learn on.

In some later CNNs, this activation function is followed by applying some form of local contrast normalization [85] (sometimes referred to as LeCun Local Contrast Normalization, or LeCun LCN). This is another operation that is done both because it seems to model visual cortex behavior and because it is frequently useful in practice. In particular, it can help in creating invariance to changes in color or illumination; local contrast normalization tends to produce overall network behavior that focuses on shape rather than intensity. This can also be seen as a way of re-introducing a mechanism similar to the balancing of excitatory and inhibitory weights in the S-cells of a Neocognitron.

2.2.2.2 Pooling Layers

Units in a pooling (or sub-sampling) layer work in a way similar to the C-cells in a Neocognitron; the big difference is that pooling units have trainable parameters. Just like a C-cell in a Neocognitron, the basic function of a pooling unit in a CNN is to reduce the spatial resolution of its input by combining several input values into a single output value. Unlike convolutional units, the receptive fields of pooling units generally do not overlap, so the result is that the output of a pooling layer will be something like a sub-sampled version of the convolutional layer below it.

In most CNNs, the input to a pooling unit is a fixed-size 2×2 region of a feature map, though there have been experiments using other sizes. The activation of a pooling unit in early CNNs was calculated by taking the average of the inputs, applying (trainable) additive and multiplicative biases, and then passing the result through a sigmoid function [82]; different values for the biases allowed the units to perform operations like simple averaging or to approximate a logical AND or OR of the inputs. In more modern CNNs, the pooling units often just perform a soft-max operation on their inputs [85]. This means pooling units in modern CNNs have actually converged back to the behavior of C-cells in early Neocognitrons.

2.2.2.3 Architecture

Like Neocognitrons, CNNs are composed of alternating layers of feature-extracting convolutional layers and resolution-reducing pooling layers (see Figure 2.7). Most of the network structure for the two models is very similar; the main differences are in how the unit activations are computed, and what the top of the network looks like.

Since CNNs were designed to be used in a supervised learning environment, they need some sort of trainable output signal. Where a Neocognitron simply reduces features to an abstract representation and stops, a CNN adds several layers of fully connected feed-forward nodes on the top. This can be thought of as a deep, convolutional "feature extraction" stage followed by a traditional multilayer neural network as a “classifier” making use of the features. However, unlike most such setups, the two stages are trained in concert using a uniform training algorithm.

For the first convolutional layer C_1, which is connected directly to the input, there will be a total of k_1 feature maps, where k_1 is the number of filters used. Each pooling layer will have a size that is 1/4 the size of its inputs, assuming a 2×2 pooling region. What is less straightforward, however, is the size of the remaining convolutional layers. At a given layer n, we can choose k_n as the number of filters to apply, but if we apply each of these filters to each of the k_{n-1} feature maps in the preceding layer, we will wind up with exponential growth as we add layers. Not only is this computationally problematic, but giving the network too many free parameters tends to result in poor performance.

Several alternatives have been presented for combatting this problem. One of the simplest involves applying a filter to each of the maps in the previous layer, and simply averaging the result. A more interesting alternative is to treat the set of 2D feature maps as slices, and stack them into a 3D volume. The receptive field of the convolutional units in the next layer also becomes 3D; in effect, each unit now takes input from the same spatial region in each of the input feature maps.

This avoids the exponential increase in units, but it still introduces a large number of free parameters. In some instances, only a subset of the feature maps in the previous layer are used for each unit; in LeNet-5, for example, the first convolutional layer had

6 filters, and the second convolutional layer had 16 filters. 6 of these received input from 3 feature maps in the first layer, 9 received input from 4 feature maps in the previous layer, and the final one received input from all 6 feature maps in the previous layer [85].

In classic CNNs, after several stages the convolutional filters will be the same size as the feature maps from the previous stage; at this point, each filter will produce a single output. It is these outputs which are used as the inputs for the fully connected classification network at the top.

In more recent work, a wide range of variations have been proposed. One recent architectural idea that has shown promise is making the networks significantly deeper; the simplest method for doing this is to add multiple convolutional layers between each pooling layer. By using larger input images, keeping the filters small, and using pooling layers less frequently, much deeper networks can be created. The architecture proposed by Simonyan & Zisserman [125], frequently referred to as “VGG,” can contain up to 19 convolutional layers; LeNet-5 by contrast contains only 3 convolutional layers. He et al. [50] constructed networks with over 100 layers, though they found some other changes to the network were required to get good performance from such very-deep CNNs (in particular, the addition of fixed-weight shortcut connections so the layers could learn residuals rather than original functions). There have also been explorations of different types of pooling operations [86], or even discarding pooling entirely in favor of simply downsampling by increasing the stride of convolutional layers [127].

2.2.2.4 Training

Unlike Neocognitrons, early CNNs were trained in a purely supervised manner, using the same error backpropagation gradient descent algorithm as other feed-forward neural networks. This was applied to the weights and biases for both convolutional units and pooling units.

Part of what differentiates convolutional units from S-cells is that where S-cells had a built-in regularization constraint (i.e., the ‘inhibitory’ weights had to sum to the same total as the ‘excitatory’ weights), the weights for convolutional units are free parameters (as is the case in most artificial neural networks). In large part, this was due to the recent development of gradient-descent based methods for training this type of network weight in multi-layer perceptron style networks. Where Neocognitrons were designed to self-organize (i.e., learn in an unsupervised fashion), CNNs were designed from the beginning to solve supervised learning problems.

Modern CNNs are still trained using a form of error backpropagation; stochastic gradient descent (SGD) is the standard practice. There are many variations that can improve performance, including standard SGD modifications like adding a momentum term [130]. Mini-batches are often used as a compromise between full batch gradient descent and completely online single-sample SGD [90]. Performing data normalization on a per-mini-batch basis has also become common since Ioffe & Szegedy introduced it [65]; this technique is referred to as Batch Normalization (sometimes abbreviated BatchNorm). Batch Normalization is one of several methods that can be used to combat overfitting; others include Dropout [57] and DropConnect [139]. There has also been work on finding better ways of initializing network weights before training [97].

2.2.2.5 Applications

The observation that unit activations can be computed via convolution allowed for computational optimizations which, combined with access to better hardware, allowed early CNNs to be trained more efficiently than Neocognitrons. The earliest successes were primarily on handwritten digit classification and resulted in several production systems built around CNNs [82].

While useful, for many years CNNs were only applied to relatively small datasets, and could not be made to scale to more challenging computer vision tasks. There were a number of important improvements in the architecture and training algorithms that were introduced, but the community studying CNNs remained fairly small.

More recently, the availability of large datasets like ImageNet [145] has meant that researchers are no longer limited to low resolution, low sample-count datasets like MNIST [83] or NORB [84]. At the same time, the availability of off-the-shelf GPUs capable of general purpose computing resulted in possibilities for massively parallel computation. The operation of CNNs lends itself well to parallelization, because each unit in a layer is data-parallel with all other units in that layer.

The 2012 paper by Krizhevsky, Sutskever, and Hinton [73] described a CNN that was trained on the subset of ImageNet released for the ILSVRC-2010 competition; this dataset is composed of 1.2 million images in 1000 classes. The original ImageNet images are of varying resolutions, so the images were cropped and scaled to be a uniform 256 × 256 pixels. While this is still low compared to modern digital camera resolutions, it was much higher than the previous datasets that CNNs had been applied to historically. This CNN (which took 5 days to train on a pair of top-of-the-line graphics cards) achieved results that outperformed the best existing techniques by huge margins, in some cases cutting the error rate nearly in half, and the authors noted that, “all of our experiments suggest that our results can be improved simply by waiting for faster GPUs and bigger datasets to become available” [73].

Following the publication of this paper, there has been a massive spike in interest in CNNs, both in academia and in industry. At the time of writing, CNNs are uncontested for state-of-the-art performance on a variety of computer vision problems, and they are in use in a wide range of production systems at companies like Google, Facebook, and Microsoft. However, it is worth remembering that CNNs are achieving these successes on a particular type of problems, namely those with spatio-temporal structure (which includes images, audio, and text); there is no violation of the No Free Lunch theorems [142] involved in their success, and there are plenty of non-spatial problems for which they are not the best choice.

2.2.3 Deep Belief Networks

Deep Belief Networks (DBNs) were developed by Hinton [53, 55] and Bengio [9] as a type of deep network based on a probabilistic graphical modelling framework. While both DBNs and CNNs are deep connectionist models, they are significantly different both in terms of architecture and in terms of training. In particular, where CNNs and Neocognitrons use a rigid network structure with layers of qualitatively distinct unit types, DBNs generally use fully connected layers in which all units operate uniformly. Thus a DBN is closer in architecture to a traditional MLP network than it is to a CNN.

However, DBNs are also fundamentally unlike perceptrons in that they are not strictly feed-forward, and are trained in an unsupervised fashion. DBNs are more closely related to undirected, associative-memory models such as Hopfield networks [61] than they are to models trained with gradient descent. This means a DBN can be thought of as a probabilistic model of the data, and can be used in a generative capacity. If class labels are part of the data at training time, a DBN can be used

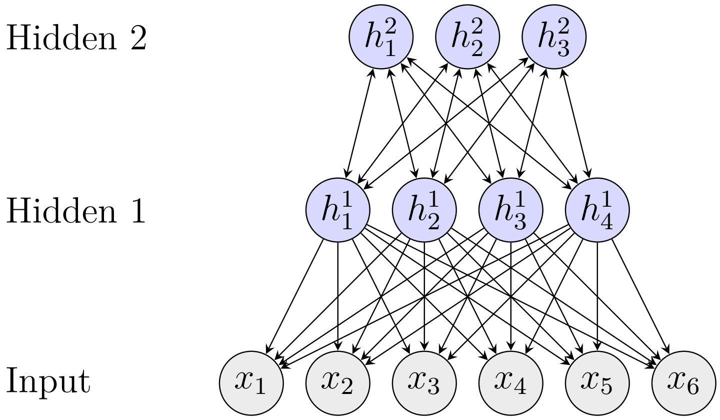


Figure 2.10: *Basic architecture of a Deep Belief Network with two hidden layers.*

as a classifier by presenting the available information and then taking a maximum-likelihood approach to selecting a class label, but such a label is not treated any differently than any other observable variable.

Thus, while DBNs can be used as a discriminative model, they can also be used for other inference tasks in a uniform way. This makes them particularly well suited to tasks where some features may be noisy, obscured, or missing entirely [9]. While a properly regularized CNN may be able to perform classification even in the presence of noise, there is no clean way to handle "missing" features. Additionally, there are many problems in which the goal is not classification, but de-noising or recovering missing information [143]; for these tasks, DBNs have obvious advantages.

2.2.3.1 Architecture

Classic DBNs are described as a probabilistic graphical model, where the two topmost layers are undirected and the rest of the network is a direct acyclic graph [53] (see Figure 2.10). The initial idea was to build up a "deep" network by training one layer at a time in a greedy fashion, to avoid the problems of "explaining away" that otherwise make training a deep probabilistic graphical model infeasible.

Nodes in a DBN are treated as latent variables; much like an MLP, layers are fully-connected to the layers above and below, but there are no connections between nodes in the same layer. Taken together, the nodes in a given layer encode a probability distribution over the nodes in the layer below them. Originally, each layer had fewer nodes than the layer below it, resulting in distributed representations that became increasingly low-dimensional and "abstract" as depth increased [53]. Later work described variants that added sparsity constraints to force less distributed encodings, which may require higher unit counts in the upper layers [109].

The connection scheme means that any two adjacent layers of a DBN form a bipartite graph encoding a probability distribution; this is equivalent to a Restricted Boltzmann Machine (RBM) [53] (see Section 2.1.3.2 for details on RBMs). DBNs take advantage of this fact by using the Contrastive Divergence RBM training algorithm to train the layers of the network.

2.2.3.2 Training

DBNs are traditionally trained one layer at a time in an unsupervised fashion [53]. Starting with the raw inputs, a layer is added to form an RBM, which is then trained using Contrastive Divergence (see Section 2.1.3.2 for details on RBM training). Once the training process is complete, the weights for the RBM are fixed, and a new layer is added. This new layer uses the activation of the previous layer as its inputs; since the weights in the layers below are fixed, we can treat all the previous layers as the input process for the new RBM, which is again trained using Contrastive Divergence.

This process is repeated for each layer in the network. It has been referred to as *greedy layer-wise training* [9], because each layer is trained independently of the others. The process of training a layer is “greedy” because it does not take into account what impact its learned encoding will have on layers above it in the network.

Once all layers of the network have been trained using this process, the network can be fine-tuned for a particular task using supervised learning. In this case, the initial construction and training of the DBN can be thought of as a complicated weight-initialization heuristic, in which case the process may be described as *unsupervised greedy layer-wise pre-training*, or just *unsupervised pre-training*.

It has been demonstrated experimentally that this unsupervised pre-training step results in networks that significantly outperform similarly architected networks trained purely using supervised learning, even if the supervised learning is done in a greedy layer-wise fashion [8, 9]. Some of the only previous work attempting to explore the inductive biases of deep learning examined the question of why unsupervised pre-training helps, and concluded that it largely functioned as a very robust form of regularization [30].

Several variations on the training algorithm have been proposed to incorporate additional terms to constrain the energy function of the RBMs (including the introduction of explicit sparsity and regularization terms) [109], or to increase the speed of training and convergence [117]. It has also been shown that using more traditional auto-associative networks in place of RBMs can yield similar results [9, 143].

2.2.3.3 Applications

For simple image classification problems like MNIST, DBNs compare favorably to "shallow" methods like SVMs and MLPs [9, 53, 77]. However, DBNs have so far been unable to scale to very large and difficult problems such as ImageNet; since the report that CNNs could successfully scale to such problems by using using parallel computing on GPUs, DBNs have fallen out of favor as an image classification technique. They have continued to be used for reconstruction tasks on images [143], and have been used for modeling semantic content in documents [56].

The main difficulty with DBN training is that contrastive divergence does not scale and parallelize as easily as the algorithms used to train CNNs. CNNs use small response regions and weight-sharing to drastically cut down on the number of free parameters (compared to fully-connected networks), and their operation is one directional. Because RBMs are undirected models, information flows in both directions along the edges; because they are fully connected, the activity of a given node can depend (indirectly) on every single weight and activation in the network. The result is that the cost of one training epoch for an RBM grows rapidly with increasing numbers of inputs.

2.3 Feature Extraction

The deep learning methods described above can be thought of as singular models, but they can also be thought of as a deep "feature extraction" network, followed by a "shallow" method that computes the final output. CNNs, in particular, are often described this way, with the deep convolutional part feeding into a shallow fully-connected feed-forward network that learns to perform the specified task. In this context, it is worth comparing deep networks to other methods that perform a mapping from an initial "raw" data-space to some other feature-space that is more convenient for a follow-on technique.

Feature selection techniques generally work by choosing a small subset of the features from the original feature vectors. In effect, the new feature-space is an axis-aligned subspace of the original feature-space.

Dimensionality reduction techniques are a broader category that encompasses feature selection, but does not restrict the new feature-space to be an axis-aligned subspace of the original. In general, the only requirement is that the new feature-space have a lower dimensionality than the original one. Techniques of this sort are generally applied to help combat what is referred to as the "curse of dimensionality". The "curse" refers to a set of phenomena associated with high dimensional data that can make machine learning either ineffective or computationally intractable.

Feature extraction is the broadest label, and simply means that the feature-space being used is different from the original. It encompasses dimensionality reduction, but places no restrictions on the relative dimensionalities of the new and old spaces. Thus, it is possible to have a feature extraction technique that results in outputs that are higher-dimensional than the inputs (sparse coding and kernel methods are common examples).

Feature engineering is a term sometimes used to describe a mapping of any of these types being created by a human engineer, as opposed to algorithmic methods for learning mappings automatically from data. This dissertation will focus on automated learning, rather than knowledge engineering performed by humans.

For our purposes, the end result of all these techniques can be described as a function that maps "raw" feature vectors from their original feature-space to some other feature-space. Without loss of generality, we will call such a mapping function a "feature-space projection." In this work, we will primarily use real-valued vector-spaces, in which case such a projection function can be written as

$$f : \mathbb{R}^n \rightarrow \mathbb{R}^m,$$

where f is the feature-space projection function, and n and m are the dimensionalities of the original and projected spaces, respectively.

2.3.1 Feature Selection

The term *feature selection* is generally used to describe a class of techniques that work to reduce the dimensionality of the data space by simply eliminating some dimensions. Given a set of attributes or features, a subset of features is selected for use, and the rest are discarded. Many different heuristics have been used to determine which features to keep and which to discard. While the details of such a selection criterion are inherently domain specific, there are two distinct high-level goals: reducing the size of the data space, and improving the performance of the system that uses the data. Often, both are desirable, but there are applications in which one or the other is more important.

Feature selection is generally performed in contexts where it is believed that the majority of features will not provide useful information for the chosen exploitation task. In such a problem, these extra features can serve as "distractors," and removing them can help reduce overfitting by preventing the learning algorithm from using irrelevant features to discriminate between data points that are equivalent when considering only relevant features. Feature selection can also help improve the convergence of local search algorithms such as gradient descent. Removing irrelevant features reduces the size of the hypothesis space, and can result in gradients which are smoother and easier to calculate.

Within the feature-space projection framework, a feature selection function simply maps from an original high dimensional vector to a new lower dimensional vector by concatenating the selected attributes into a new vector and discarding the rest. If m features are selected from an original set of n, the projection function would be $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$, where the space $\mathbb{R}^m$ is an axis-aligned subspace of $\mathbb{R}^n$, and the function $f(\mathbf{x})$ simply returns the linear projection of $\mathbf{x}$ into the subspace $\mathbb{R}^m$. The dimensions (or features) not shared by $\mathbb{R}^n$ and $\mathbb{R}^m$ are discarded, and the values for the remaining dimensions are unchanged.

2.3.2 Dimensionality Reduction

Dimensionality reduction describes a broad class of methods that attempt to take long data vectors and transform them into shorter ones, ideally without the loss of task-relevant information. Many different types of algorithms can be used to do this, and what information qualifies as task-relevant is by definition problem-specific, so no one algorithm will be best for all situations.

One use of dimensionality reduction is *data compression*, in which the task is the reduction in total size of some data while minimizing loss of information. Some compression schemes are lossless, meaning the size of the data is reduced without any loss of information; others are lossy, meaning that some information is irrecoverably lost in the transformation. Lossy algorithms can generally compress the data more efficiently (that is, to a smaller final size), but many problems cannot tolerate the loss of information. The goal is to find a mapping function

$$C : \mathbb{R}^n \to \mathbb{R}^k,$$

where $k < n$. The ratio

$$R_C = \frac{n}{k}$$

is known as the *compression ratio*, and the primary goal of data compression is to maximize this ratio.

Techniques in data compression rely on finding regularities in data that can allow the data to be represented using fewer bits of information. One commonly used regularity is the existence of repeated patterns; if a given pattern, or sub-vector, appears multiple times in a data vector, the data vector can be compressed by storing the pattern once, and then replacing the matching sub-vectors with references to the stored pattern.

Techniques in dimensionality reduction more broadly rely on similar regularities and algorithms. The difference is that in standard compression, we are only concerned with the ability to compress known data. In the broader context, we are often concerned with the ability to compress *un*-known data. This is the concept of *generalization* as applied to data modeling.

The requirement of good generalization changes the pattern-finding problem; in compression, if a pattern exists more than once in the data, the compression ratio can be increased by storing that pattern in a codebook. Compression ratio is a very simple performance measure, but when comparing lossless compression algorithms, it is generally the primary factor we are concerned with (time and space needed for compression/decompression are sometimes of interest as well).

Performance measures based on good generalization are more complex, and more difficult to satisfy. For a set of real-valued vectors, finding compressible patterns in each vector and storing them will generally result in a longer representation of a given novel vector than that novel vector's original length. This is due to the fact that we are storing a potentially very large number of patterns that are present in only a small number of vectors. In the context of generalization, we are therefore less concerned with generating lossless decompositions, since they tend to result in overfitting.

When performing standard compression, we can easily measure how much information is lost in the process, and can make firm theoretical claims that a given algorithm produces a "lossless" compression scheme. If we need to compress vectors that we never saw when generating the compression scheme, such claims can no longer be well-grounded. Instead, we must use an indirect measurement of information loss, such as average loss over a given testing set, or mean expected loss for a given distribution.

In order to achieve good compression but minimize loss of information, we want to store patterns that are highly likely to appear in a novel vector, and not those that are highly unlikely to appear. This likelihood is generally not known *a priori*, but can be estimated if we make the standard assumptions that the novel vector is drawn from the same distribution as the training data, and that all samples are independent and identically distributed (IID).

If we are interested in feature extraction, we generally want to extract a consistent set of features, which means that we need a fixed-length decomposition, and one in which given output elements have a consistent interpretation. Standard codebooking does not fit these requirements, nor do many other standard algorithms designed for compression; the underlying ideas and mathematics, however, remain quite similar.

In particular, while standard variable-length codebooks will not work, a type of fixed-length codebook can be used, and often performs quite well. Known as *vector quantization*, the idea is to pick a set of template vectors, and then represent each data vector with a reference to the template that it most closely matches. The loss of information in this case will be the difference between the vector and its template. If the templates are well chosen, this can lead to a high compression ratio and low loss of information for amenable types of data.

Standard vector quantization works on one data vector at a time, attempting to map each data vector to a single template. This is effectively a clustering algorithm; find clusters (templates), then use cluster membership to represent points. The trouble with this is that in order to get low loss of information, we generally need a large number of clusters. Since we need to store all the cluster means as full-length vectors, the compression ratio is often not as good as we might like.

Vector quantization fundamentally takes advantage of the assumption that the data has meaningful structure to it; that is, we assume the data are not distributed uniformly across the entire space in which they are defined, but rather are predominantly confined to smaller local regions or manifolds. The fact that vector quantization appears to work for many types of real-world data suggests that this assumption is often well founded.

Mathematically, we can describe a set D of vectors in $\mathbb{R}^n$ using a set of template vectors T, yielding a resultant "compressed" set $D^{'}$ as

$$\begin{aligned} D &= \{\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_m\}, \mathbf{v}_j \in \mathbb{R}^n \\ T &= \{\mathbf{t}_0, \mathbf{t}_1, \ldots, \mathbf{t}_k\}, \mathbf{t}_j \in \mathbb{R}^n \\ D^{'} &= \{u_0, u_1, \ldots, u_m\}, u_j \in \mathbb{N}[1, k], \end{aligned}$$

where the compression ratio, R_C, is the ratio of the total description lengths,

$$R_C = \frac{m \cdot n}{m + (k \cdot n)},$$

given the simplifying assumption that the representation length of a single $u_j \in \mathbb{N}[0, k]$ is similar to the representation length of one vector element, $v_{j_i} \in \mathbb{R}$. A more detailed computation would require a specification of the representation length of the integer and real-number data types being used.

2.3.2.1 Principal Component Analysis

Principal Component Analysis (PCA) [67, 105] is a classic example of a dimensionality reduction technique; like vector quantization, it uses templates, but it represents data points as a linear combination of templates, rather than just a single template. It is a method for taking a set of points in one orthogonal basis and creating a linear transformation matrix which projects those points into a new orthogonal basis. The new basis vectors are parallel to the eigenvectors of the covariance matrix of the original data. For this reason, PCA is sometimes referred to as eigen-decomposition. The advantage of this is that each (scalar) eigenvalue is related to how much of the variance of the original data set lies along the dimension represented by the corresponding eigenvector. This property means that low eigenvalue basis vectors can be discarded with little loss of information. The basis vectors themselves can be viewed as templates in the original dataspace. The encoding can be reversed by treating the low-dimensional coordinates as weights applied to a linear combination of the basis vectors.

If all basis vectors are used, the projection is lossless. PCA for dimensionality reduction works by retaining the basis vectors associated with the top k eigenvalues, and discarding the rest. Projecting into this basis then functions as an encoding into $\mathbb{R}^k$. The value of k is selected by a human engineer, often by examining a plot of the eigenvalues to select a point of diminishing returns.

Using PCA in this way can be seen as a way of generating a linear transformation to k dimensions that is optimal with respect to mean squared reconstruction error. It is worth remembering, however, that mean squared reconstruction error is not always an accurate measure of encoding quality for all types of exploitation tasks. Additionally, better reconstruction error may be achievable if we allow for non-linear projections.

Within the feature-space projection framework, PCA generates an f function that takes an original data vector and projects it into the space defined by the selected eigenvectors. Given the singular value decomposition of the original data matrix into $\mathbf{UDV}^\top$ (where the columns of $\mathbf{U}$ and $\mathbf{V}$ form orthogonal bases to transform to and from the eigenspace, and $\mathbf{D}$ is a diagonal matrix containing the singular values), this is accomplished by multiplying input vectors by the first k rows of $\mathbf{V}^\top$, where k is the desired output dimensionality:

$$f(\mathbf{x}) = \mathbf{V}^\top \cdot \mathbf{x}$$

If a non-linear projection is desired, this approach can be extended to Kernel Principal Component Analysis (KPCA) [120]. Many other variations and optimizations have been proposed; of particular interest in our own work is an iterative approximation algorithm that allows the first k eigenvectors to be found without the computational expense of working with the full covariance matrix [4].

2.3.2.2 Discriminant Analysis

PCA-based techniques are designed to minimize reconstruction error, which is useful for modeling but is not necessarily optimal for discriminative tasks such as classification. The related class of *discriminant analysis* techniques choose basis vectors to maximize inter-class discrimination rather than to minimize loss of information. The best known of these is Fischer's Discriminant Analysis [35]. The term "Linear Discriminant Analysis (LDA)" is sometimes used to refer to Fischer's technique, but this usage can lead to confusion since this term is also used to refer to a binary classification algorithm that relies on the same underlying mathematics.

Fischer's technique works by looking at the ratio of inter-class variance to intra-class variance. As with PCA, singular value decomposition can be used to find a set of eigenvectors whose corresponding eigenvalues describe their importance. The multi-class version was described by Rao [110], and works by multiplying the inverse covariance (Σ^{-1}) with the inter-class scatter matrix Σ_b:

$$\Sigma_b = \frac{1}{c}\sum_{i=1}^{c}(\mu_i - \mu)(\mu_i - \mu)^\top.$$

The eigenvalues of $(\Sigma^{-1}\Sigma_b)$ correspond to class separation, meaning we can rank the eigenvectors by how well they separate points.

The top k eigenvectors can then be used to form a basis that will give the best possible separation between classes. As with PCA, we can then use the top k rows of $\mathbf{V}^\top$ to create a feature space projection function

$$f(\mathbf{x}) = \mathbf{V}^\top \cdot \mathbf{x}.$$

2.3.2.3 Isomap & Locally-Linear Embedding

Isomap is a technique designed to fit a low dimensional manifold to a set of data [132]. As with kernel methods such as support vector machines [119], the idea is that data which are not linearly separable under any linear projection may be linearly separable under a non-linear projection. Unlike kernel methods, however, Isomap takes the approach of explicitly trying to fit the data to an k-dimensional manifold (where k is less than the dimensionality of the data vectors). The most common example is data that form a (sometimes extruded) spiral, which needs to be "unrolled" before it can be linearly separable.

The underlying assumption is that the high-dimensional data exist on (or near) a low-dimensional manifold, and that distance on the manifold is of much stronger relevance for the chosen exploitation task than distance in the high-dimensional data-space. The algorithm proceeds from this assumption by assigning each data point a "neighborhood" of nearby points to which it is connected with edges in an undirected graph. Once this graph is built, pairwise geodesic distances are computed for each pair of points using shortest-path costs in the graph. These pairwise geodesic distances are then used to perform standard multidimensional scaling (MDS) [74, 122, 123].

Locally-Linear Embedding (LLE) was developed independently at around the same time as Isomap [115] and follows the same conceptual approach, though the mathematical and algorithmic approaches differ. In LLE, low-dimensional points are represented as linear combinations of eigenvectors rather than as the type of low-dimensional Euclidean embedding created by MDS.

It is important to note that the use of the term *local* in this context refers to pairs of data points that have a low value for some pairwise distance measure. This is locality in data-space and is distinct from the feature-space based notion of locality described in Chapter 4.

2.3.2.4 Hierarchical Clustering

Hierarchical clustering techniques such as agglomerative or divisive clustering [66] can also be seen as a way of creating feature-space projections. In either the agglomerative or the divisive case, each level of the clustering hierarchy can be viewed as a single feature-space projection function that maps data vectors to their corresponding cluster membership. Thus, for a hierarchy of depth d, with original data vectors in $\mathbb{R}^n$, we have a set of feature-space projection functions F:

$$\begin{aligned} F = [f^1 :\ & \mathbb{R}^n \to \mathbb{R}^{m_1},\ \dots\ , \\ f^i :\ & \mathbb{R}^n \to \mathbb{R}^{m_i},\ \dots\ , \\ f^d :\ & \mathbb{R}^n \to \mathbb{R}^{m_d}], \end{aligned}$$

where m_i is the dimensionality of the cluster membership vector for the clustering produced by level i of the clustering hierarchy. This output dimensionality is particularly important when the clustering allows multiple-membership or fuzzy-membership. The output dimensionality will be equal to the number of clusters present at a given level of the clustering hierarchy. For single-membership clustering, this results in a “one-hot” style encoding. If multiple or fuzzy membership is allowed, there may be non-zero values along multiple dimensions, each describing a degree of membership in the corresponding cluster.

2.3.2.5 Hierarchical Self-Organizing Maps

Hierarchical Self-Organizing Maps [111] are essentially a form of divisive clustering. A Kohonen-style Self-Organizing Map (SOM) [70] is trained to split the data into some number of subsets; each of these subsets is then used to train another SOM, and the data continues to be recursively split until some termination criterion is met. This results in a relatively standard clustering hierarchy, it just uses a competitive learning mechanism for choosing the clusters. Mathematically, all that has changed from the previous case is the method used for generating the cluster labels. From a practical standpoint, SOMs have some important differences from more traditional clustering algorithms, such as the distinction between distance in data space and distance in cluster space, and the fact that cluster centers are affected by a local neighborhood of other cluster centers.

2.3.3 Multi-Resolution Analysis

Traditional Multi-Resolution Analysis (MRA) [25, 94, 112] is one of the few non-connectionist algorithms that does make some use of spatial locality (as described in Chapter 4). Like CNNs, MRA was originally designed to analyze spatio-temporal data. Unlike CNNs, MRA came from a signal processing background, and has no strong ties to connectionism, though there are still occasional mentions of evidence that animal vision systems have a hierarchical structure of this kind [1]. Additionally, one of the primary goals of MRA is to introduce scale invariance; this is distinct from CNNs, where translational invariance is also considered to be of high importance (though each method also allows several other forms of invariance to varying degrees). In fact, retaining information about the spatial localization of patterns was described as an intentional goal of some of the early work in this field [1].

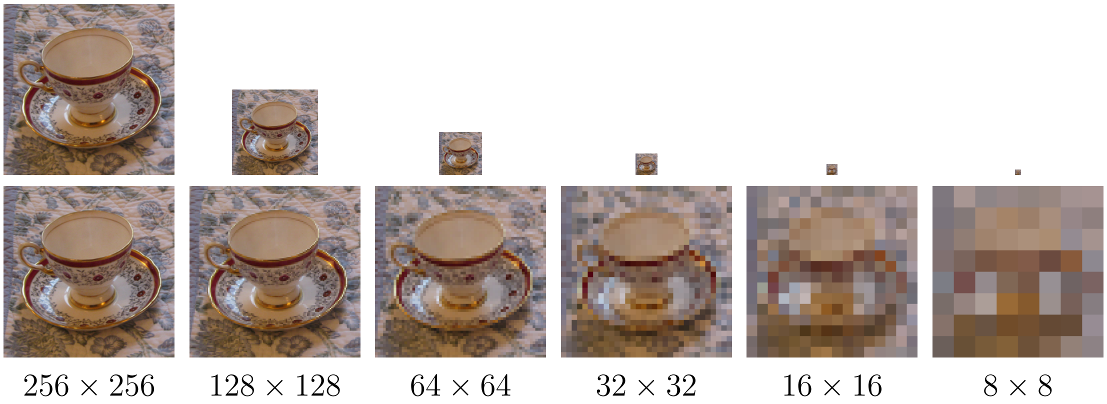


Figure 2.11: *An image pyramid generated from an image of a teacup. Above are downsampled versions of the original image; below are versions of the downsampled images rescaled to the size of the original. Resolutions of the downsampled version are listed below each column.*

2.3.3.1 Image Pyramids

The simplest way of analysing an image at multiple resolutions is simply to generate multiple versions of the image, each at a different resolution, and then analyze each version separately.

One way of doing this is to start with an image, and then downsample by a factor of 2 in each dimension. The result will be an image with the same basic content, but at a lower resolution. If this process is repeated, the resulting versions can be thought of as a vertical “stack” where resolution decreases going up the stack. This type of structure is referred to as a *pyramid* style decomposition [1, 15, 21]. See Figure 2.11 for an example.

There are a number of variations on this basic method that offer improved performance. First, while the downsampling can be a simple sub-sampling scheme in which one sample is taken from each 2×2 group of pixels, this tends to result in significant aliasing. As a result, most formulations apply a low-pass filter of some sort prior to downsampling. This can be as simple as applying a Gaussian blur to an image before sub-sampling, and then averaging the pixels from the 2×2 region of the blurred image [1]. This is sometimes called a *Gaussian pyramid*; the example in Figure 2.11 is of this type.

This type of pyramid will naturally tend to loose high-frequency information such as edges; in many applications, this is not desirable, so alternatives can be used. A *Laplacian pyramid* [15] can be generated by replacing the Gaussian filter with a Laplacian filter, which can be thought of as using a band-pass filter instead of a low-pass one. Another way to think of this is as subtracting the "blurred" image from the original, resulting in an image that brings out the high-frequency features. When the downsampling is done after this, the result is that edges are enhanced, taking up proportionally more of the smaller images.

A complete image pyramid takes only $1^1/_3$ times the space of the original image to store, and operations on the smaller versions are computationally cheaper than on the larger image, so pyramids can be seen as an efficient data structure. For example, they can be used to test for template matching at multiple resolutions much faster than could be done by using multiple scaled versions of the template [1]. Pyramids are also frequently used in computer graphics, where they are often referred to as *mipmaps* [141], and are deployed to increase texture fill speed while decreasing aliasing.

2.3.3.2 Wavelets

Image pyramids represent image information at multiple scales, but they are intended to represent visually interpretable information. Wavelet analysis discards the visual understandability of the pyramid in favor of using basis functions to encode each level of the pyramid [25, 94]. Intuitively, this is similar to a Fourier transform, which converts a signal from a time domain representation to a frequency domain representation. A value in the frequency domain encodes the amplitude of a sinusoidal waveform of corresponding frequency in the time domain, along with a phase offset (encoded as the imaginary component of the complex number). The goal is to pick a set of waveforms that, when composed, re-construct the original signal.

Wavelet analysis does something intuitively similar, but instead of using sinusoidal functions, it uses some other set of functions as a basis to encode the signal. It is these basis functions that are referred to as *wavelets*, $\psi(x)$ (this terminology was introduced in [49]). It is important to note that these are not arbitrarily chosen basis functions; in particular, it is desirable to have functions that provide good localization (as opposed to the sinusoidal functions used in traditional Fourier analysis), and form an orthonormal basis of $L^2(\mathbb{R})$ (i.e., Lebesgue spaces). Haar functions were used as an early example, and were later shown to be a special case of the more general class of wavelet functions described by Daubechies [24].

For one-dimensional (i.e., time-series) data, a *continuous wavelet transform* (CWT) can be defined as:

$$\Psi_x^{\psi}(\tau, s) = \frac{1}{\sqrt{|s|}} \int x(t)\psi\left(\frac{t-\tau}{s}\right) dt$$

where ψ is the basis wavelet, τ and s are the translation and scale parameters, and $x(t)$ is the (time-domain) signal being analyzed. Since Ψ is a function of τ and s, we need to compute this at a range of values, but so long as our data is band-limited (which any data from actual sensors must be), the ranges will be bounded. In theory, this must be done for a continuous range of values of s and τ, though in practice this can be approximated by a sufficiently fine-grained grid search. Even with this limitation, however, analytically computing the above integral for a large range of parameter values tends to be computationally intractable.

For this reason, the approach taken in wavelet-based multi-resolution analysis is typically to use a *discrete wavelet transform* (DWT). This can be thought of as performing a CWT where the scale parameter s is discretized on a log-scale grid; this is because at lower frequencies, the lower spatial resolution of the wavelet means fewer samples are required. The offset parameter τ is then discretized based on s; at higher frequencies (i.e., more compressed versions of the basis wavelet), τ samples are closer together.

Computationally, however, the DWT is generally done by sub-sampling the data, rather than by compressing the wavelet (as in image pyramids, the data is convolved with a filter before sub-sampling to avoid aliasing). This results in a more computationally efficient approach.

For computer image problems, s is generally sampled at powers of 2 (just like an image pyramid), and τ (now 2-dimensional) is sampled linearly in s. This results in something that looks rather like the image pyramids described earlier, but now instead of a low resolution image at the top of the pyramid, we have low-frequency wavelets (meaning a low value of s and only a few different values of τ). At the base of the pyramid, we have many high-frequency wavelets (meaning wavelets with a high value of s with densely sampled offsets).

It is also common to take the approach of modeling residuals, meaning the top layer attempts to fit the data, but the layer below it attempts to fit the difference between the data and the top layer. Modeling residuals in this fashion gives us greater independence between the layers and tends to result in better models.

Like image pyramids, MRA can be used as a form of compression; in fact, the JPEG-2000 image format offers wavelet based methods for both lossy and lossless compression, though the format has not seen widespread adoption at the time of writing.

From a machine learning standpoint, wavelets combine many of the advantageous properties of Fourier analysis and image pyramid based multi-scale analysis. The primary practical disadvantage in comparison with CNNs is that CNNs use learned filters (rather than pre-defined ones), and can handle more types of invariance.

2.3.4 Multi-Scale Learning

Multi-Scale is a term which is not always used consistently. This can lead to confusion, but under some interpretations the concept is closely related to our work. We will describe a few of the uses of the term here, and indicate how our work relates to them.

The term "multi-scale" has been used to describe approaches that apply non-wavelet based methods to multi-resolution problems (i.e. problems where the same "object" may appear at vastly different scales). Recently, attempts have been made to address multi-scale problems using a variation on CNNs [91]. This work uses an approach called *spatial pyramidal pooling*, which was applied to allow CNNs to handle non-uniform resolutions in input images [51]. Unfortunately, current approaches still require all input resolutions that will be used to be explicitly enumerated, with a separate network being trained for each resolution. Weight sharing between these networks reduces the computational overhead, but these methods are still not fully able to handle arbitrary resolution inputs.

As with MRA, the intent is to learn representations that can be matched at multiple scales. These approaches make implicit use of local spatial structure, as well as multi-scale invariances in the data. The work presented in this dissertation does not attempt to make use of multi-scale invariance in its inputs, though this is an interesting direction for future work. Of the work presented, the technique described in Chapter 7 is the closest to being applicable in this type of multi-scale setting, though the approach described in that chapter is limited by the fact that different levels of the hierarchy do not interpret feature values the same way. If the features at different levels could be "standardized," we believe such a hierarchy could be used in a multi-scale context relatively easily, though we have not yet tested this hypothesis experimentally.

However, the term "multi-scale" is also used in several other ways. In the context of deep learning, it has been used to describe the practice of feeding the outputs of multiple hidden layers to the final classifier (as opposed to using only the final layer) [28, 121]. Similarly, it has been used to describe a model that used a wavelet transform to produce inputs to a "multi-scale" set of predictive models [144], or a model that uses a "multi-scale" set of Gaussian filters to pre-process data for segmentation [32]. Using this broader definition of the term, much of the work in this dissertation can be viewed as taking advantage of "multi-scale" information. The technique described in Chapter 7, in particular, could easily be used as a multi-scale feature extractor simply by giving the final classifier access to the intermediate level encodings.

2.3.5 Random Forests

Random Forests are an ensemble method using decision trees. Over the years, a number of variants have been proposed, but all share the property that decisions are made based on an ensemble of trees, which are trained using some sort of constraint that makes them function as “weak” classifiers. If there is sufficient independence between the trees (i.e., if they make errors independently of one another), they have several nice properties from both theoretical and practical standpoints.

The first description of an ensemble of intentionally-weakened decision trees was by Ho [58, 59]; each tree was trained using a random subset of the available features. This is effectively a partitioned training scheme, with conceptual similarities to what we describe in Section 4.2 (though we were not aware of Ho’s formulation when we began our own). The idea of an ensemble of classifiers trained from random partitioned subspaces was explored further by Kleinberg [69], who called it “stochastic modeling.” Unlike our work, however, neither Ho nor Kleinberg were interested in using these partitions to capture spatial structure explicitly; their use of partitions was simply to create weak-but-independent classifiers.

The formulation of random forests from Amit & Geman [3] described doing the recursive node-splitting in a stochastic way, which also used the notion of random partitioned subspaces, but the subspaces were created per-node rather than per-tree.

Breiman [12] extended this approach by using a bagging scheme such that each tree was trained on a subset of examples (Ho trained on the full set of examples, but each tree had a fixed subset of the features for each example), and each node was trained using a subset of the features. Breiman also described several variants, including "Forest-RI" in which each split used a single feature (i.e., a subspace of size 1), and "Forest-RC" in which each split used a random combination of features (i.e., a subspace of size c).

Tomita *et al.* [134] further generalized this to base decisions on weighted-linear combinations of features (i.e., projections onto arbitrarily oriented subspaces). This approach allows for the number of features to be used to be randomized on a per-node basis, and the weighted combination allows for arbitrarily oriented decision boundary hyperplanes.

2.3.5.1 Decision Trees

Decision trees are a type of decision algorithm that predates modern computer science by a considerable margin; they are simple, intuitive, and their operation is transparent (i.e., a human can look at a decision tree and understand the decision-making process). Historically, decision trees were created manually by domain experts, but in the field of machine learning they are created in an automated fashion.

While many methods have been suggested, the majority of tree-learning algorithms boil down to a greedy algorithm for recursively splitting a dataset. At the root node, this algorithm is applied to generate a set of children; the algorithm is recursively applied to each child. The base case results in terminal (or leaf) nodes, which are associated with decision values. Decision trees can be produced for either classification or regression (the term classification-and-regression-tree (CART) was introduced by Breiman *et al.* [11] to encompass both variants); in this work, we are primarily interested in classification.

Early work by Quinlan [106, 107] suggested several variants using the concepts of entropy and information gain for node splitting criteria. Breimen [11], working independently at around the same time, suggested the Gini purity index as a node splitting criterion. There have since been a wide range of alternative methods proposed; here, we will concentrate on these two, since they are what is typically used in random forests.

The information gain based heuristic suggested by Quinlan [106] works by minimizing entropy, which is an information theoretic measure of how many bits of information are required on average to encode a sample. Entropy is minimized when the set is homogeneous, since for a uniform set all samples will be the same, and therefore no information is required to encode the value of a sample.

For a set of examples S, with k classes, let s_i be the set of examples of class i. Then the entropy of S is given by:

$$H(S) = -\sum_{i=1}^{k} P(s_i) \log_2(P(s_i))$$

Since we do not know the true probability $P(s_i)$, we estimate it by using counts:

$$P(s_i) \approx \frac{|s_i|}{\sum_{j=1}^{k} |s_j|} = \frac{|s_i|}{|S|}$$

If we substitute this into the entropy equation, we get

$$H(S) = -\sum_{i=1}^{k} \frac{|s_i|}{|S|} \log_2 \left(\frac{|s_i|}{|S|} \right).$$

In the context of classification, we want to minimize entropy, since the goal is to get disjoint subsets of examples in which all examples share a class label (i.e., all examples are classified correctly by the tree). To determine the value of a particular candidate splitting criterion, we can compute how much entropy is reduced by performing such a split; this is called information gain. If a candidate split criterion α assigns examples to m disjoint subsets, we can define the combined entropy of those subsets as

$$H(S|\alpha) = -\sum_{j=1}^{m} \left(\frac{|s^j|}{|S|} H(s^j) \right),$$

where s^j is the set of examples that are assigned to subset j by α. Plugging in $H(s^j)$, we get

$$H(S|\alpha) = -\sum_{j=1}^{m} \left(\frac{|s^j|}{|S|} \left(-\sum_{i=1}^{k} \frac{|s_i^j|}{|s^j|} \log_2 \left(\frac{|s_i^j|}{|s^j|} \right) \right) \right),$$

where s_i^j is the set of examples that are of class i and are assigned to subset j by α. The information gain for α can then be written as:

$$IG(\alpha) = H(S) - H(S|\alpha)$$

The Gini impurity measure proposed by Breiman [11] as an alternative is intuitively similar, measuring the likelihood that a randomly selected element of a set will be misclassified by a label drawn from a distribution based on the proportion of set elements belonging to each class. As with entropy, the Gini impurity is minimized when the set is homogeneous.

Again, for a set of examples S, with k classes, let s_i be the set of examples of class i. Then the Gini impurity of S is given by:

$$\begin{aligned} I_G(S) &= \sum_{i=1}^{k} P(s_i)\,(1 - P(s_i)) \\ &= 1 - \sum_{i=1}^{k} P(s_i)^2 \end{aligned}$$

which is approximated with occurrence counts as

$$I_G(S) \approx 1 - \sum_{i=1}^{k} \left(\frac{|s^i|}{|S|} \right)^2 .$$

Both methods are applied in a *greedy* fashion, in that the training algorithm selects the split that reduces entropy or impurity as much as possible in a single step. While optimizing for overall tree behavior might be ideal, global optimization of decision trees has been shown to be NP-complete [64].

In early work, the candidate split criteria (α) were just single attribute values [106]. Later work used combinations of attributes to generate "oblique" decision trees [52, 99]. In general α can be any function that assigns data vectors to one of m discrete values. For each of these values, a child node is added to the node in question, and training samples are assigned to each child node by applying α to the set of training samples at the current node.

The recursive algorithm normally terminates when the set of training samples at a node is uniform (i.e., they all have the same class value, and/or they all have identical values for all attributes), though this tends to result in overfitting. Several early stopping and pruning methods have been suggested to alleviate this, but none have met with universal success.

2.3.5.2 Random Decision Forests

As mentioned previously, there are a number of ways to build an ensemble out of decision trees; the only fundamental requirement is that the trees be built in a stochastic way. As with any ensemble, the goal is to have the members make mistakes in independent ways, such that the ensemble as a whole has higher performance than any of the members. In reality, complete independence is generally impractical at best, and performance is still limited by the ability of the member classifiers to generalize.

In Breiman's classic formulation [12], a random forest is an ensemble of decision trees, each of which is trained using some random "parameter" vector Θ. In practice, random values are normally generated on the fly by sampling from a (pseudo-)random number generator as needed, but the idea is the same.

While there are many methods that fit this definition, the term "random forest" is often used to describe Breiman's Forest-RI, which uses orthogonal trees [59] (i.e., splits are always axis-parallel). The basic procedure is to start each tree with a subset of the full set of training examples, a method called *bagging* [10]. Bagging works particularly well for unstable learning methods (i.e., sensitive to small changes in the training set), which greedily-learned decision trees tend to be.

Each tree is then trained using a greedy recursive algorithm similar to the one described above, but with much more randomness (and, therefore, less aggressively greedy). At a given node, the algorithm picks c candidate features as possible splitting criteria. For each possible candidate, an optimal splitting value is chosen by testing possible splitting values against a metric (generally information gain or Gini). The candidate with the best score is chosen as the winner, and is used to split the current node. Other than this, the overall recursive algorithm is the same as for basic decision trees as described above.

Tomita et al. [134] presented a generalized algorithm for Projective Random Forests, which both encompasses most previously described random forest variants and allows for new variants. The basic idea is that, rather than selecting a single feature (as in Forest-RI) or a fixed number of features (as in Forest-RC) to use as a splitting criterion, we can use any projection down to a scalar value. Forest-RI can be thought of as a projection onto an (orthogonal) axis, and Forest-RC can be thought of as a projection onto an oblique axis. Rather than using a fixed value for the number of features to use in such a projection, we can generate projections randomly; this results in a more flexible process, but so long as we use a linear projection, it

Algorithm 2.2 Pseudocode for Random Projection Forests, which generalizes a wide range of previously proposed decision forests, adapted from [134]. $\mathbf{x_i}$ are feature vectors, y_i are class labels, and $\mathbf{A}$ is a projection matrix; $a \sim b$ indicates a is sampled from b

```
Input: data: $\mathbf{D} = (\mathbf{x_i}, y_i) \in (\mathbb{R}^p \times Y)$ for $i \in [n]$, tree rules, distributions on projection
    matricies: $\mathbf{A} \sim f_A(D)$,
Output: decision trees, predictions
for each tree do
    Sample training data to obtain a bag $(\bar{X}, \bar{y})$
    Create a root node with this bag as its sample set
    for each leaf node do
        Let $\tilde{\mathbf{X}} = \mathbf{A}^\top \bar{\mathbf{X}}$, where $\mathbf{A} \sim f_A(D)$
        Find the "best" split coordinate $k^*$ in $\tilde{X}$
        Find the "best" split value $t^*(k^*)$ for this coordinate
        Split $\mathbf{X}$ according to whether $\tilde{\mathbf{X}}_i > t^*(k^*)$
        Assign each child node as a leaf or terminal node using stopping criteria
    end for
end for
```

still ultimately boils down to making a decision based on a threshold applied to a weighted linear combination of features. Pseudocode for this algorithm can be found in Algorithm 2.2.

Much theoretical work has been done to characterize the properties of random forests (see, for example, [12, 58, 59, 69]). Like all ensemble classifiers, random forests can out-perform their member classifiers to a degree that is related to the level of independence between the members (in terms of their likelihood of making an incorrect prediction for any given example).

One of the biggest strengths of random forests is the fact that adding more trees to the forest will tend to improve overall ensemble performance, with no upper bound on the number of trees. While a point of diminishing returns may be reached, adding extra trees does not (on average) result in decreased performance on novel testing data. For this reason, decision forests are often described as being immune to overfitting.

It is worth noting, however, that this is only true when comparing forests that differ only in the number of trees but have all trees generated using the same way. In particular, we have found in our own experiments that increasing the modeling power of the individual decision trees can result in lower overall forest performance due to overfitting, regardless of how many trees are generated (see Chapter 8).

2.3.5.3 Applications

Random forests have been successfully applied to a wide range of problems, and have been shown to outperform other classifiers on a wide range of benchmarks [16, 33]. The latter tested 117 distinct classifiers on 121 datasets from the UCI machine learning repository. The results of both of these papers was that random forests were the overall winner, followed closely by SVMs. Studies like these suggest that random forests are a good choice for arbitrarily selected problems, i.e. those for which we can make no particular assumptions safely.

However, as the authors of these studies point out, the UCI repository does not cover the complete space of interesting machine learning problems. In particular, most of the problems are relatively low dimensional, and even the largest had only

262 features. Compared to high resolution digital images, which may have tens of millions of features, this is a small number. When these methods are compared to CNNs on image classification problems, they do not fare nearly so well.

Chapter 3

Image Datasets

In this chapter, we describe several image classification datasets used in our experiments and analysis. These particular datasets were selected mostly for their widespread use in benchmarking image classification techniques and for being datasets on which deep learning algorithms have been shown to perform well. They span a wide range of size and task difficulty, but all are fundamentally the same type of problem, allowing results to be easily compared between them. Note that not all datasets were used for all experiments (generally for computational reasons). For the sake of comparison, we provide state-of-the-art results from the literature in Table 3.1, though we did not attempt to replicate these results ourselves. Because we did not control these experiments, we will treat these results as anecdotal, and no conclusions should be drawn about relative performance compared to our methods.

Dataset	Classification Error	Source
MNIST	0.21%	Wan *et al.*, 2013 [139]
SVHN	1.69%	Lee *et al.*, 2016 [86]
CIFAR-10	3.47%	Graham *et al.*, 2015 [46]
CIFAR-100	24.28%	Clevert *et al.*, 2016 [18]
ImageNet (ILSVRC 2012)	16.5%	Szegedy *et al.*, 2016 [131]

Table 3.1: *State of the art classification performance on standard image benchmarking datasets. In each case, the system producing the state-of-the-art result is some form of Convolutional Network.*

3.1 MNIST

The MNIST (Mixed National Institute of Standards and Technology) database is a dataset of handwritten digits, constructed from NIST's SD-3 and SD-1 databases. MNIST is distributed as 60,000 training samples plus 10,000 testing samples. Each image is 28×28 pixels, and encodes a single handwritten digit (0 to 9). The task is to recognise the digit, effectively a 10-class classification problem. The data is heavily pre-processed, with raw images having been thresholded, binarized, and cleaned to get a black digit on a white background. The binary digit images were then cropped and scaled to fit in a 20×20 region, resulting in a low-resolution grayscale image with the original digit aspect ratio maintained. The digits are then centered in the final 28×28 image, resulting in a 4 pixel wide white border around every image. See Figure 3.1 for some sample images. It was introduced in [82], and can be obtained from [83].

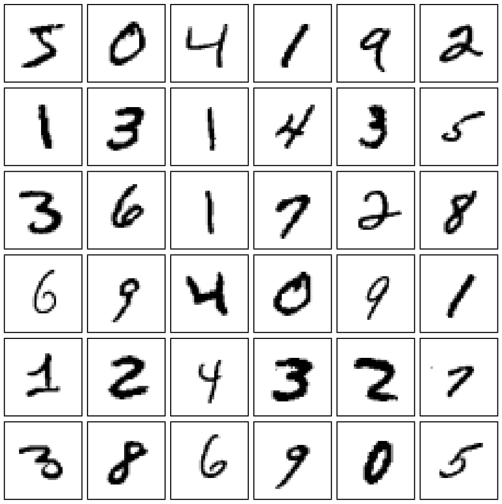


Figure 3.1: *Sample images from the MNIST dataset. Each example is a hand-written digit which has been heavily preprocessed and scaled to* 28×28 *pixels.*

Classifying MNIST was one of the first big successes of deep learning, at a time when it was considered a challenging problem. Today, MNIST is largely regarded as a "toy" problem, as it is fairly small and relatively easy compared to other image classification datasets. State-of-the art at the time of writing is less than 0.21% classification error [139].

3.2 SVHN

The Street View House Numbers (SVHN) dataset is another digit classification problem, but with digits extracted from house numbers in Google Street View images. SVHN is distributed as 73,252 training samples, 26,032 testing examples, and

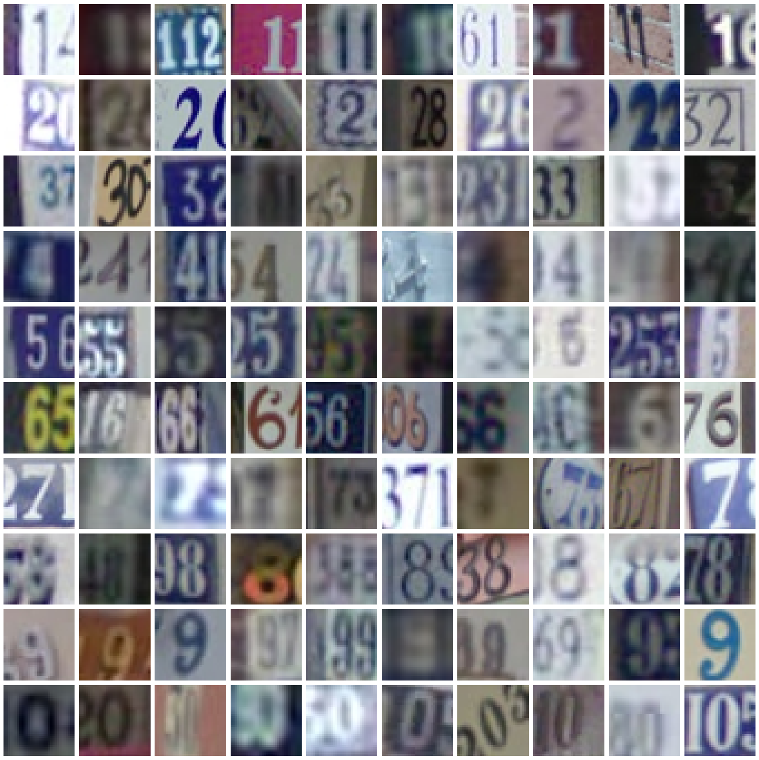


Figure 3.2: *Sample images from the SVHN dataset. Each example is a digit from a house number in a Google Street View image which has been cropped and downscaled to* 32×32 *pixels.*

531,131 "extra" samples. It can be obtained either as a set of 32×32 pixel images, each centered on a digit, or as the full-size original images along with bounding-box information for each digit; both versions are color images. For our experiments, we used the 32×32 pixel images, which were generated by cropping a square image patch based on the bounding-box information. The patches took the larger of the two bounding box dimensions, and extended the other to make the region square, so the patches retain the original aspect ratio, but are scaled down to a uniform size. No other preprocessing is done, so the images are fairly heterogeneous compared to

MNIST. In particular, both background and foreground color and texture are highly variable, orientation is not uniform, and most images contain some or all of the adjacent digit(s) in the house number, which tend to function as distractors. See Figure 3.2 for some sample images. SVHN was introduced in [102] and can be obtained from [101]

The basic task is similar to MNIST, but the lack of preprocessing makes the task significantly harder. This is another dataset which has proved to be a good fit for deep learning, with most high-performing solutions involving some form of CNN. State-of-the-art at the time of writing is less than 1.7% classification error [86].

3.3 CIFAR

The CIFAR dataset (Canadian Institute for Advanced Research) is two subsets of CSAIL's 80 Million Tiny Images dataset [135] which have been hand-labeled to enable their use as an image classification dataset. There are two versions, CIFAR-10 and CIFAR-100; we used the CIFAR-10 version in our work. CIFAR-10 and CIFAR-100 are non-overlapping subsets, but each was generated in the same way. The difference is that CIFAR-10 is comprised of 10 distinct classes with 6000 examples of each, and CIFAR-100 is comprised of 100 distinct classes with 600 examples of each (the sets of classes do not overlap). Each is distributed as a set of 50,000 training examples

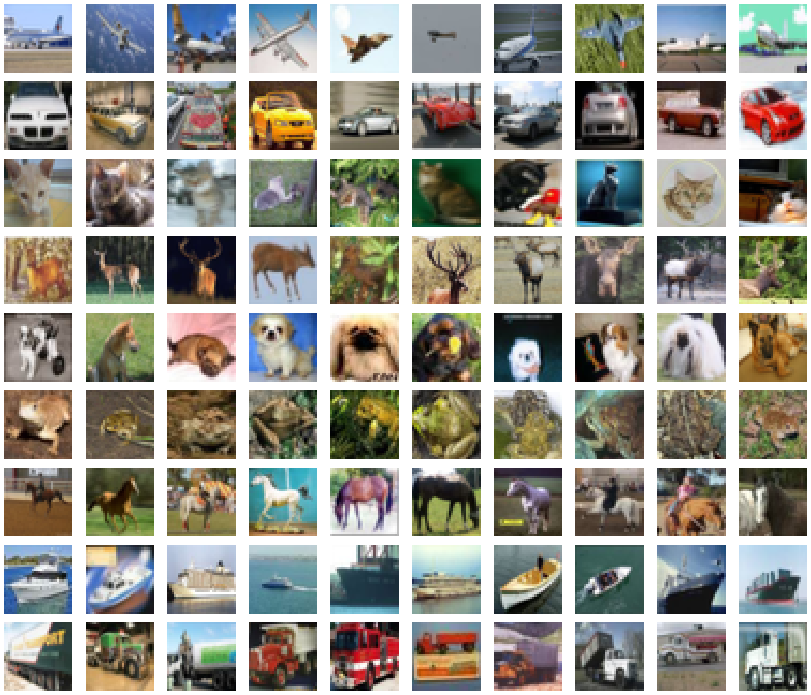


Figure 3.3: *Sample images from the CIFAR-10 dataset. Each example is an image of an object which has been cropped and downscaled to* 32 × 32 *pixels.*

and 10,000 testing examples. The original Tiny Images data was gathered by feeding search terms into several image search engines and downloading the results, each of which was downsampled into a 32 × 32 pixel color image with "noisy" labels (i.e., the search term used to obtain the image). The CIFAR datasets are subsets for which humans examined the images to ensure that only correctly labeled examples with a clear foreground object were included. Other than scaling to 32 × 32 pixels, no preprocessing was done. The classes are broad enough that there is significant variation between the actual foreground object given a particular label, as well as variations in scale, orientation, location, and foreground and background color and texture. Some images contain foreground objects that are partly occluded but are still recognisable by a human. See Figure 3.3 for some sample images. It was introduced in [71] and can be obtained from [72]

The task here is significantly harder than the digit classification datasets; while CIFAR-10 is still a 10 class problem, the class “dog” contains a great deal more variation than the class “7”. Additionally, the total number of training examples for each class is relatively small, especially for CIFAR-100. Again, high-performing systems tend to be variations on CNNs. State-of-the-art at the time of writing is less than 3.5% classification error for CIFAR-10 [46] and less than 25% classification error for CIFAR-100 [18].

3.4 ImageNet

ImageNet is a large-scale image database in which images are associated with nouns from WordNet. This allows the images to be organized into a “concept” hierarchy, which allows for classification at different levels of abstraction, among other things. The raw images are scraped from the web, but they are curated and annotated by hand. At the time of writing, there were around 16 million images in the dataset, representing over 20,000 concepts, referred to as “synsets” in WordNet parlance; WordNet contains around 80,000 noun-concepts in total. No preprocessing

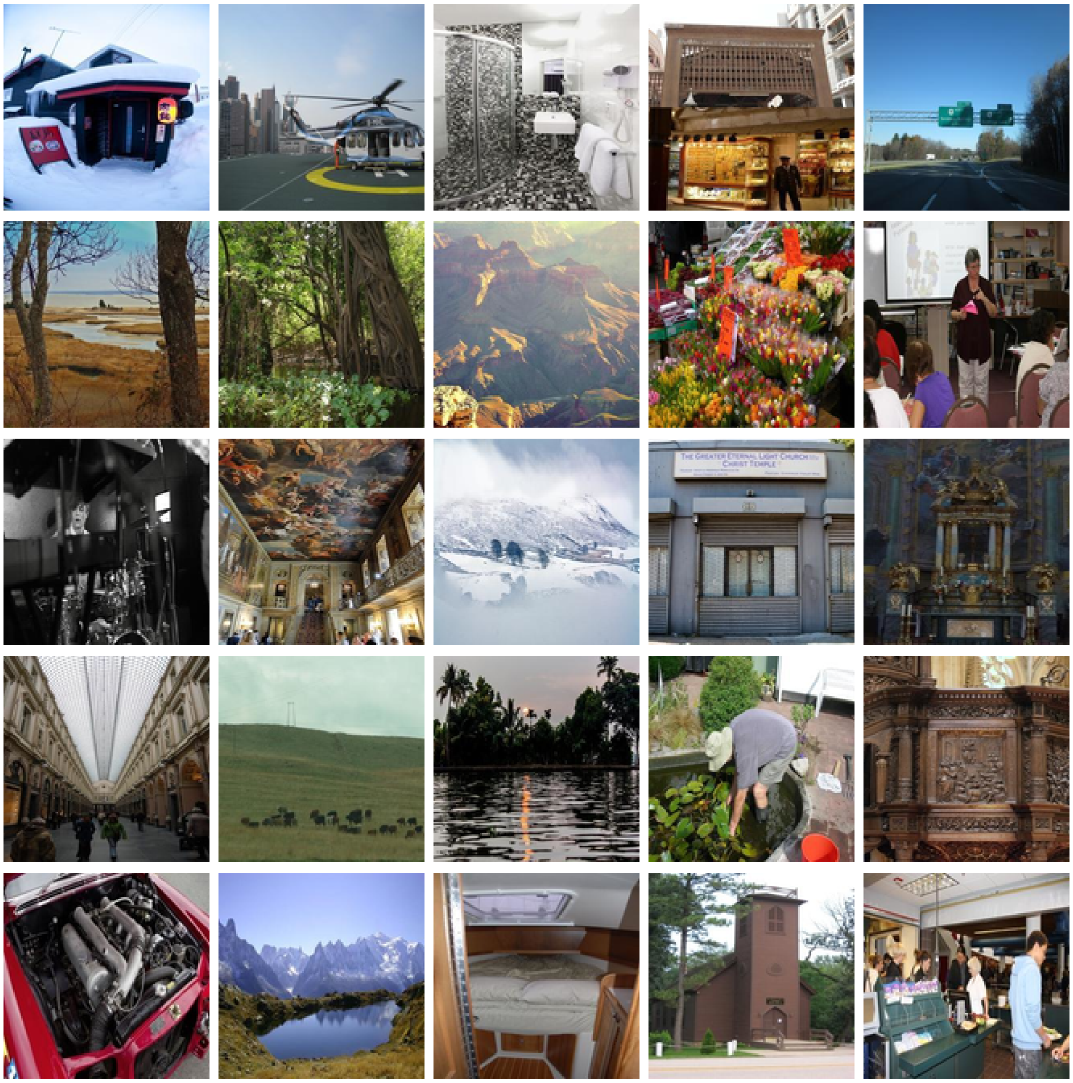


Figure 3.4: *Sample images from the ImageNet dataset. Each example is an image scraped from the web which has been hand-tagged with WordNet synsets; in the ILSVRC2015-Places2 version we used, images were scaled to* 256×256 *pixels.*

has been done, and since the images were not created by the maintainers, the images are highly non-uniform. There is a wide range of image shape and resolutions, quality, and style, and the images were taken under a wide range of different conditions. A variety of subsets of ImageNet have been created for different machine learning competitions; the work we present used the "small" (256 × 256 pixel) version of the ILSVRC2015-Places2 dataset, which is approximately 115Gb in size. This version of the dataset was introduced in [146] and can be obtained from [145] (for non-commercial research and educational purposes only). See Figure 3.4 for some sample images.

The classification task here is extremely difficult and remains very much an unsolved problem. The total dataset is big enough to be intractable without specialized hardware and algorithms, and the lack of a uniform image resolution also presents problems for many standard techniques; for this reason, most work is done using only a portion of the images, and they are often scaled down to a more manageable (and standardized) resolution. Even the large-scale distributed training algorithms developed at Google [26], which used a cluster with over 10,000 CPU cores, required the images to be rescaled down to 100 × 100 pixels, though they were able to train on all 16 million of these small images by sharding the dataset across the cluster. This work achieved a classification accuracy of 16%, meaning the error rate was 84%. While this is significantly better than chance for a 20,000 class problem, 16 out of 100 is still pretty poor performance. This problem is so difficult that even on the smaller subsets, results on ImageNet classification are frequently reported as "5-guesses" error, meaning that if any of the top 5 guesses by the classifier are correct, it is counted

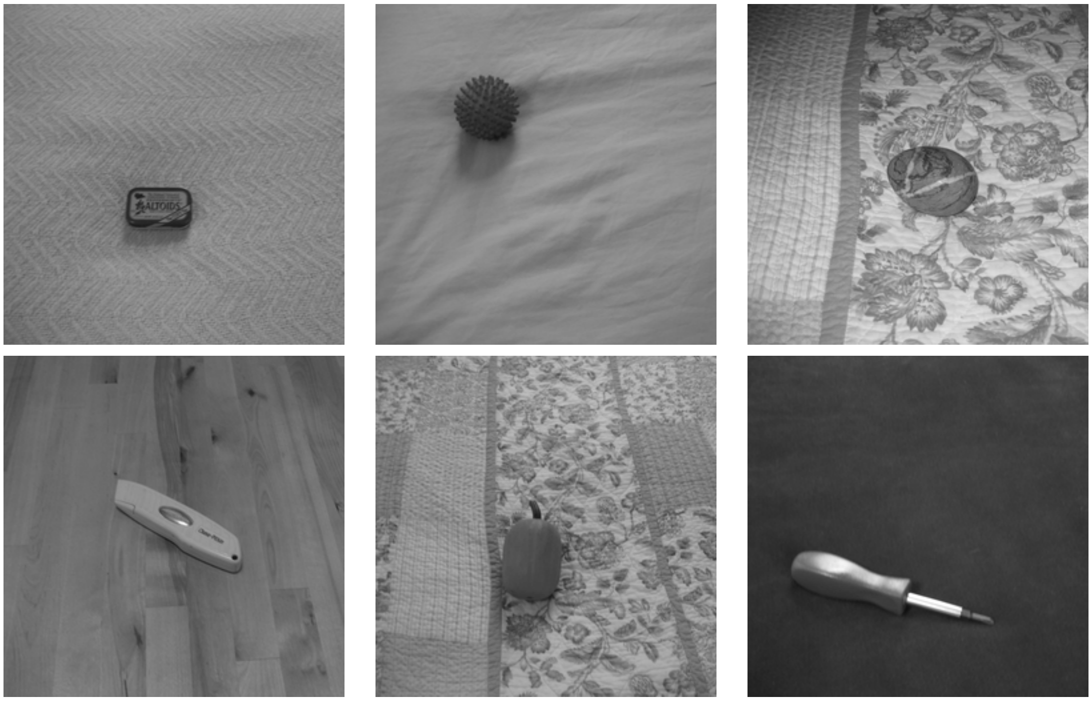


Figure 3.5: *Some example images from the Simple Objects dataset. Each image is a photograph of a foreground object against one of 5 backgrounds. Images were cropped and downscaled to* 512×512*,* 256×256*, and* 128×128*, giving three different "versions" of the dataset.*

as a success. The best results reported at the time of writing is 16.5% error on the ILSVRC 2012 subset [131].

3.5 Simple Objects

This is a small object recognition dataset we created for some early experiments. There are 600 images of 10 foreground objects taken against 5 different backgrounds; we used 512×512, 256×256, and 128×128 pixel versions so we could test behavior at different scales. This dataset is very sparse and very small by modern standards; at the time, we lacked the compute resources to handle large datasets of high-resolution images. We created this novel data set so that we could have natural images (i.e., not artificially generated or composited) in multiple resolutions. Since the release of ImageNet, this dataset is of largely historical interest. See Figure 3.5 for some sample images.

Chapter 4

Spatially Local Structure

Our basic hypothesis is that many deep learning techniques share an inductive bias that assumes certain structural properties are satisfied by the data. If true, it makes sense this would give them an advantage, but only when working on data that conforms to this assumption. Before we can test this hypothesis experimentally, we need to first define in a more precise and formal way what this assumption is. We also need to explore the realm of statistical techniques that will let us test how badly this assumption is violated by a given dataset. This, along with methods of artificially creating or destroying such structure, will allow us to devise a set of experiments to determine whether our hypothesis is supported by actual systems.

In addition, once we have clearly defined these assumptions, we will be able to impose these assumptions on models where they are *not* built in implicitly. Not only will this serve as another test of the hypothesis, but it also has the potential to let us boost the performance of these other methods.

4.1 Local Structure

First, let us be clear about what we mean by *spatially local structure*, or just *local structure* in general. In many machine learning and statistical contexts, the term *structure* is used to describe properties of a dataset. In these contexts, we are looking at a set of examples and trying to discover patterns formed by the distribution of examples. The "structure" in this case consists of relationships between different examples. For instance, if our data points are real valued vectors, we would be comparing the vectors to one another, so any "structure" would have the same dimensionality as our data.

We want to distinguish this from *local structure*, which we define to be patterns or statistical relationships between different features. The distinction here is between looking at structure across a dataset, and looking at structure across the dimensions of the data space. In this work, we will use the term *global structure* to refer to the "standard" use of the term *structure*, as a way to help us keep these concepts distinct.

Our use of the term *local* is also a bit different from the way it is often used, in that we are referring to locality in terms of features *within* a data vector. The more standard usage of *local* as it is applied in contexts like Locally Linear Embedding [115] or Isomap [132] refers to locality in terms of distances between complete data vectors. The basic notion of locality is the same; in both instances, it is a way of describing how close two things are to each other. The difference is whether we are measuring distance between data vectors, or between features.

Global structure lets us ask questions like, “if this image contains a dog, how likely is it to also contain a cat?” *Local structure* lets us ask questions like, “if this pixel is red, how likely is the pixel next to it to be blue?” Given how much more interesting the answer to the first question is, it is unsurprising that machine learning tends to concern itself primarily with global structure. Here, we concern ourselves with local structure, not so much because it is more interesting by itself, but because it offers us a chance to simplify some of the combinatorially intractable problems that global structure analysis tends to run into.

In fact, many statistical modelling techniques already make some use of local structure; the independence relationships encoded in a Bayesian network, for example, can be thought of as encoding a form of (conditional) local structure. The main distinguishing feature of our work is the application and examination of assumptions about local structure that are built into a model. In particular, we will be focussing on structure in data with an intuitive notion of spatial distance between features, hence the term *spatially local structure*.

While it is less well known than some others, there has long been a branch of statistics dedicated to the examination and modeling of spatially local structure. That field is Spatial Statistics, and we will be making use of a number of concepts and methods drawn from spatial statistics in our work. We will go into more detail about the theory and the math in section 4.3, but a deep understanding of the math is not required for a high-level conceptual understanding of the deep local structure hypothesis. We will also not make extensive use of the language and terminology of spatial statistics. This is a conscious choice, made largely because our data do not always meet the standard assumptions made in the spatial statistics literature, which could make our use of that terminology misleading.

4.1.1 Locality

In its most general form, the term *locality* as we use it here describes statistical dependence relationships. Information is *local* to some subspace if it has no dependencies with anything outside that subspace. Formally, for data in $\mathbb{R}^n$, we say that a statistical relationship is *localized* to a subspace $C \subset \mathbb{R}^n$ if that relationship can be completely captured using only C (i.e., $\mathbb{R}^n \setminus C$ can be discarded with no loss of information). In other words, local information is information that is encapsulated within a particular subspace and is statistically independent of everything outside of that subspace. We can also speak of *conditional locality*, which is equivalent to saying a subspace is conditionally independent from the rest of the data given some other (possibly latent) value. While statistically (and combinatorially) weaker, this type of conditional independence is generally a better fit for real-world data, so it will often wind up being of more practical use.

Formally, then, *local structure* is simply structure (in the statistical sense) which is localized. Like most strong statistical properties, in real-wold data the question is not whether we can prove complete independence (or even conditional independence), but how close to that ideal we can get. In practice, the question is not a binary one of presence or absence, but rather a question of membership; how *much* local structure is present?

This is a more complex question than it seems because there are actually two factors involved. Given some subspace, it is relatively straightforward to determine how strongly elements of that subspace are correlated with other elements both inside and outside of that subspace. If elements within the subspace are highly correlated with each other and have low correlation with elements outside the subspace, we can say that subspace shows high locality.

What makes things difficult is that this type of analysis relies on knowing what elements should be in the subspace. We are left with a version of the model selection problem; how many subspaces should we use, and how should we assign elements to them? Like most versions of the model selection problem, this one is ill posed without some kind of modeling bias, and intractable unless that bias significantly restricts the hypothesis space.

We will use the term *partitioning* to refer to this model selection process of decomposing a feature-space into a set of subspaces. In practice, this means taking feature vectors and assigning each element of a vector to one or more *partitions*. Each partition corresponds to a particular subspace, based on a *partitioning scheme* that defines this mapping. The process takes a single "full-length" vector as input, and outputs several smaller vectors, one for each subspace. By applying this process to each vector in a dataset, we can get a *partitioned dataset*. In general, a partitioned dataset will contain more vectors than the original dataset, but each vector will be shorter (i.e., it will have fewer features); the ratios involved will depend on the partitioning scheme used. For a very simple example, if we start out with 10 vectors in $\mathbb{R}^4$, and our partitioning scheme simply splits each vector in half with no overlap, the partitioned dataset would contain 20 vectors in $\mathbb{R}^2$. A formalism, as well as several illustrative examples, are presented in Section 4.2.

As with most forms of model selection, attempting to learn an optimal partitioning scheme for a given task is a form of the meta-learning problem [129], and we will not attempt to solve it here. Instead, we will use the familiar approach of solving the model selection problem manually, and then using automated techniques to optimize the parameters of the model. Just as most connectionist networks or graphical models begin with a defined network architecture, we will begin with a defined partitioning scheme. As with most such models, this will be created based on a combination of prior knowledge and trial-and-error, and will mean performance may be limited by a poor choice of model. The full meta-learning problem, though a fascinating one, must be left for future work as it is beyond the scope of this dissertation.

4.1.2 Spatial Locality

While the general case is of theoretical interest, it is of limited practical applicability because of the aforementioned model selection problem. In practice, therefore, we need a way of picking a sensible model before we can start to look for patterns.

One special case of general locality is *spatial locality*, which we can now define formally as a partitioning scheme where partitions are based on spatial distance between features in our data vectors. This only makes sense when the original features have some notion of "spatial" arrangement, and in which we have an expectation that spatial distance might correlate with statistical independence. Fortunately, both of these appear to be reasonable expectations for a number of types of real-world data. One common example is digital images, for which each "vector" is a spatially organized grid of pixel values. When we say "spatial distance" here, we mean the distance between corresponding pixels in that grid. In principle, we can use any distance measure we like, though in practice we will mostly use the L^2-norm.

Square regions of an image, sometimes called “patches” or “windows,” are used in a partitioned fashion in a wide range of image processing techniques, though the statistical independence relationships implied by such a framework are generally not talked about explicitly. However, since a wide range of successful techniques rely on this type of decomposition (see Chapter 2), it seems reasonable to think this might be an effective partitioning scheme.

For many of our experiments, we will stick to using regularly spaced and sized spatial regions for partitioning, following many of the conventions used by other techniques. This allows us to exploit assumptions that already seem to be working, reducing the model selection problem to simply a choice of the size and spacing of our patches. This also allows us to more easily and directly compare our results to other existing work. However, this is not the only option available to us; see Chapter 8 for an example of a spatial partitioning that is neither compact nor square.

While we will restrict our experiments to image analysis, the theory applies equally well to other applications. However, “spatial” may have other meanings for different types of data. For example, many applications have data that are not sampled on a grid; fields like geology or meteorology often have data in which each value has a corresponding sampling location. In this case, a spatial decomposition would probably be based on the sampling locations, rather than on the relative position of the elements in a data vector.

We can also imagine non-spatial contexts in which intuitive notions of locality may be productively applied; temporal locality, in particular, seems like a natural extension, and again one that is implicitly used by many existing time-series analysis techniques. We consider this to be another promising area for future work.

4.1.3 Deep Locality

In most real-world data, we would not expect completely local structure, meaning that there will be some information that cannot be captured locally. Another way of putting this is that partitioning the dataset would result in a net loss of information (from an information-theoretic standpoint). This sort of real-world violation of strong model assumptions is very common. For example, the Markov assumption and the Naïve Bayes assumption are both routinely used despite the fact that they are not perfectly adhered to; in spite of this, both have proven very useful in practice.

If we want the local structure assumption to be similarly useful, we must ensure that it both matches real-world data as well as possible, and degrades as gracefully as possible when not fully met. Conditional locality will frequently be of use here, as it will allow us to construct a hierarchical model of (almost) local patterns. For example, if we have two partitions that are mostly independent, we may be able to make them more independent by creating a latent variable upon which both are conditioned. This latent variable is less "local" since it effectively spans a larger subspace (the union of the two partitions). However, it is still somewhat local, since it still only spans a portion of the original data space (unless the union of the two partitions does, in fact, cover the entire space).

Again, we will use spatial structure in images to illustrate this idea. If we have an image that has been partitioned into 16 non-overlapping patches of equal size (e.g., by laying a 4×4 grid over it), we can construct "local" models for the patches, and then use latent variables to capture whatever information the local models were unable to capture. We can even go a step further and make several latent variable models, each of which covers only a subset of the local "patch" models. For example, let us say that we will create four latent models, one for each quadrant of the original image. Each latent model will then be linked to four of the original models, and the four original models will be conditionally independent from each other given the latent model.

Each of these latent models is a local model, but they indirectly span a larger region of the original image (a quarter, instead of a sixteenth), so they are in some sense *less* local. We can then repeat this process, creating another set of latent variable models, which will be used to make the first set of latent models conditionally independent of one another. This process can be repeated until the point where the result is a single latent variable model that indirectly spans the entire original image.

The result is a hierarchical model, where each level of the hierarchy can be thought of as modeling the image at a certain level of granularity or abstraction. In an ideal case, each model will account for as much information as can be captured locally, and the models in the next level up the hierarchy will encode *only* the residual information that is available over the larger area it implicitly spans. Latent Tree Graphical Models [17] and Infinite Latent Feature Models [43, 44] have some similarities to this approach, though they make no attempt to make use of spatial structure.

We will use the term *deep locality* to refer to this idea, since it is closely related to the way deep learning models work, if a bit broader and more generic. In an ideal case, the hierarchical model taken as a whole would give us a parsimonious-but-lossless encoding of the original data. Since this will not occur with real data, we will once again try to characterize how well a given dataset lends itself to this kind of modeling.

In its broadest form, the concept of deep locality can be used with any type of partitioning and modeling processes. In this dissertation, we will focus our attention on *deep spatial locality*, which is the special case in which spatial distance is used to decide which models to join at each level of the hierarchy. The image model described above is an example of this. The broader case will be left largely to future work, though as previously noted we expect it to be easily applicable in other spatio-temporal domains, including natural language processing and time-series analysis.

4.2 Feature-space Partitioning

To describe the statistical effects of partitioning on data vectors more formally, we will first introduce notation to describe different partitioning schemes. To begin with, we will restrict our notation to real-valued vector spaces. In principle, there is no reason that most of the math cannot be applied to other types of feature spaces, such as multivariate discrete or heterogeneous, but we will leave such applications for future work.

If our input data space is $\mathbb{R}^n$, we define a partitioning function π as

$$\pi : \mathbb{R}^n \to \left\{\mathbf{u}_1, \ldots, \mathbf{u}_k\right\}, \mathbf{u}_i \in \mathbb{R}^s,$$

which takes a single input vector in $\mathbb{R}^n$ and produces a set of k output vectors in $\mathbb{R}^s$, where $s < n$. In principal, $\mathbb{R}^s$ can be any s-dimensional space; the math does not preclude the use of complex linear transformations, or even non-linear ones. In practice, it is generally unclear how such a complex transform would be chosen, so the standard approach is simply to keep a subset of the original vector elements. In such a case, each partition can be thought of as a feature selection process (selecting a subset of s features from n possibilities). The k subspaces are not required to be disjoint, and in fact many existing techniques, including CNNs, use overlapping subsets.

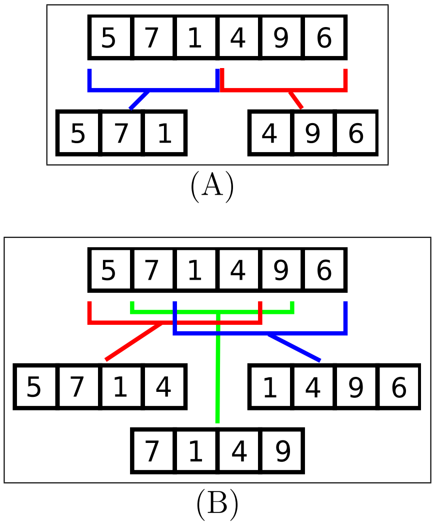


Figure 4.1: *Simple examples of vector partitioning. (A) breaking a vector into two disjoint sub-vectors, and (B) breaking it into three overlapping sub-vectors.*

As an illustrative example, let us consider a partitioning function that simply splits vectors of length 6 into two equal length halves; in this case, $n = 6$, $s = 3$, and $k = 2$ (Figure 4.1-A). As mentioned, partition membership need not be mutually exclusive, and partitions may overlap. So, we might have another partitioning function that takes our length 6 vectors and produces 3 output sub-vectors of length 4, by having a size-4 partition window placed at the front, middle, and end of the original vector. This would give us $n = 6$, $s = 4$, $k = 3$ (Figure 4.1-B).

In general, it must be the case that $1 \leq s \leq n$ and $1 \leq k \leq n$. There are more precise constraints between s and k, but the exact form will depend on the type of partitioning being done. In the simple case given above, $s + k \leq n + 1$, because the longer the sub-vector is, the fewer size-s windows can be fit into the original vector without repetition. If we do not want overlap, this is further restricted to $s \cdot k \leq n$.

Partitioning functions can also be constructed that respect structural properties of the original data vectors. For example, in the case of vectors representing images, there is a natural 2-dimensional layout to the vector elements (pixels). While it is possible to treat an image as a single (long) vector and partition it in the way described above, it would make more sense to treat the image as a 2D grid and partition it into smaller 2D patches. For example, an 8×8 pixel image might be split into a set of 4×4 pixel sub-vectors. In general, it is desirable for adjacencies and distances between elements of the original data vector to be preserved in the partitioned sub-vector, especially if the intent is to exploit spatially local characteristics of the data. See Figure 4.2 for an example of a simple 2D partitioning.

If we want to talk about what statistical information is destroyed by the partitioning and what is preserved, we need to be able to reference more than one vector at a time since a single sample contains no statistical information in isolation. Therefore, we will assume a dataset composed of m samples, where each sample $\mathbf{x}_i$ is composed of n real-valued features:

$$\mathbf{D} = \{\mathbf{x}_1, \ldots, \mathbf{x}_m\}, \mathbf{x}_i \in \mathbb{R}^n$$

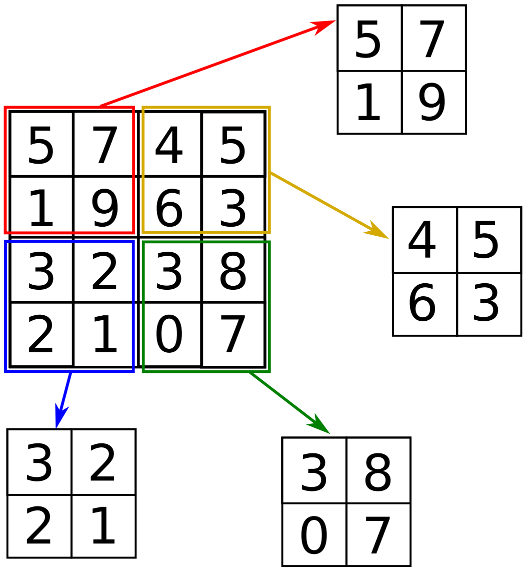


Figure 4.2: *An example of partitioning a vector with a 2-dimensional inherent structure. In this case, 4 equal sized partitions are created with no overlap.*

We can treat each feature as a random variable X^j, where $j \in \{1, \dots, n\}$, meaning a vector is interpreted as $\mathbf{x}_i = \{X_i^1, \dots, X_i^n\}$. We will use $\mathbf{X}^j$ in the absence of any subscript to denote the set of all the jth vector features in the dataset,

$$\mathbf{X}^j = \{X_1^j, \dots X_m^j\}.$$

We can now describe the statistical properties of these random variables across the dataset. For any individual feature i, we can compute statistical measures like mean and variance. For any given pair of features (i, j), we can compute statistical measures like correlation, covariance, mean-deltas, and so forth; each vector in $\mathbf{D}$

provides one sample for each feature, so the number of samples will be the cardinality of $\mathbf{D}$.

By applying a partitioning function π to every element of a dataset $\mathbf{D}$, we can generate a new dataset, which will be a set of the sets created by applying π to the $\mathbf{x}_i$:

$$\mathbf{D}_\pi = \{\{\pi(\mathbf{x}_1)\}, \ldots, \{\pi(\mathbf{x}_m)\}\}$$

In some instances, we may be able to treat the partitioned sub-vectors uniformly, in which case we may want to combine them into a single set by taking the union of the subsets created by the partitioning,

$$\mathbf{D}_\pi = \{\pi(\mathbf{x}_1) \cup \ldots \cup \pi(\mathbf{x}_m)\},$$

but this will not be appropriate for all types of data. In cases where all features are interpreted the same way, this is generally fine, but if some features need to be interpreted differently, we cannot take the union this way. It also helps if the data is isotropic (i.e., changes are independent of direction), and spatial invariance is expected (i.e., a given pattern is equally likely to occur anywhere in the data vector). Natural image data tends to fit these assumptions fairly well, as do many types of geological and meteorological data (for which this type of analysis is called *geostatistics*), and location-based medical, social, and economic datasets (e.g., for the analysis of cancer clusters). This is also true of many types of temporal data, including audio, natural language text, and generic time-series data.

The main advantage of taking the union is that we get a larger number of samples, which allows for better estimation; the term “weight sharing” as it is used in the context of CNNs [82] is implicitly describing this type of union. As an example, for natural image data, taking the union is generally safe since we frequently want to create spatially invariant models anyway. In a more structured dataset, such as the MNIST dataset (see Section 3.1), we do not expect complete spatial invariance in the data (the digit images are centered and surrounded by a white border, so the statistical properties of pixels near the edge and pixels near the center will be different), so taking the union would be a trade-off between having more samples but making an (incorrect) assumption that they were all samples from the same distribution. Whether or not this trade-off is a good one will depend on just how badly the assumptions are violated; as with the Naïve Bayes and Markov assumptions, the benefits can outweigh the downsides even if the assumptions are not met perfectly.

Within the partitioned dataset, we will need to keep track of which random variables in the original dataset map to which elements of the partitioned dataset. If we do so, we will then be in a position to describe which statistical quantities can still be calculated, and which quantities cannot be calculated because the variables involved are not in the same partition.

So long as the partitioning function preserves locally adjacent blocks of the original vectors, it will be the case that we can still measure statistical properties of feature pairs that are "near" each other in the original vectors, but we will be unable to measure statistical properties of features that are "far apart" in the original vectors. In other words, spatially local statistical information can be preserved, even if much of the non-local information may be lost.

4.2.1 Types of Partitioning Functions

The simplest way of encoding local information is to simply partition a feature-space into a set of sub-spaces; each sub-space will capture only the statistical information local to the features it includes. This is closely related to the feature selection problem, but the goal is to group features into related subsets, rather than to reduce the overall number of features used. In the absence of perfect knowledge about what features should be grouped together, there are a variety of heuristics that can be used to partition a feature-space. Currently, these heuristics are mostly based on spatio-temporal structure; we consider the investigation of heuristics for partitioning based on generalized local structure to be a promising direction for future work.

4.2.2 Partition Shape

The first aspect to consider is what the size of each partition should be, and how to assign features to partitions. Models like CNNs and Neocognitrons have traditionally used square, contiguous image regions; when a CNN is described as using filters of size 5×5, it is a description of both the size of each partition (25 features) and the way that features are assigned to partitions. The input features are treated as a 2D array, a central locus is chosen, and then all features with both X and Y coordinates less than 3 units away from the locus are grouped together in a partition.

While this is an intuitive and natural partition shape for the square images traditionally used as input for CNNs, it is not the only way we could imagine doing a partitioning. For instance, there is no reason that the filters need to be square; mathematically and algorithmically, using 5×7 filters works the same way that using 5×5 filters does.

Images are not only spatial, but the spatial coordinates are on a fixed grid; for other types of spatial data, it might make more sense to pick a feature as a locus and then group with it all features with locations less than some distance d away (in effect, all features within a hypersphere of some radius). Alternatively, we might take the k nearest neighbors and group them together.

In temporal data, partitions are usually made by treating the time value as the primary criterion for grouping features. So, for example, a window of 5 seconds might be used to define a partition, and all the samples inside that window would be grouped into a subspace.

4.2.3 Overlapping Partitions

Partitioning can be done either in a way that ensures all subsets are disjoint or in a way that results in overlap, meaning some features are present in more than one subset. If partitions do not overlap, then the total size of the partitioned dataset will be no greater than the size of the un-partitioned dataset. If overlap is present, the size of the partitioned dataset will be strictly greater than the size of the un-partitioned data; the greater the degree of overlap, the greater the size difference. For this reason, using overlapping partitions can be viewed as a form of data augmentation.

For a CNN, the amount of overlap is determined by the relative magnitude of the filter width and stride parameters. If the stride is greater than or equal to the width, there will be no overlap; maximal overlap occurs when the stride is 1.

Whether overlap is desirable will depend on the application; more overlap gives a better chance that a particular local relationship will be captured, but for some algorithms overlap creates undesirable dependencies between parts of the model that would otherwise be independent. For example, when training partitioned RBMs (see Chapter 6), overlapping partitions result in slower training because the dominant factor in RBM training time is clique size. By keeping partitions disjoint, the number of mutually dependent parameters is kept low, allowing for rapid iteration and convergence.

4.2.4 Hierarchical Partitioning

It is also possible to set up hierarchical partitioning schemes; for example, a standard quad-tree style decomposition can be thought of as a hierarchical spatial partitioning. Within any given level of the hierarchy, the partitions of a standard quad-tree decomposition will be disjoint. Across levels, however, this will not be true; each feature will be present in exactly one partition at each level, so the total number of replications of a given feature will be equal to the number of levels in the quad-tree.

CNNs do something conceptually similar, but mathematically different, because the “features” present at the second layer are not features that are present in the input. These features are a result of feature extraction, rather than feature-space partitioning, so a standard CNN does not technically use a hierarchical feature-space partitioning, even if it implicitly results in something similar.

4.2.5 Stochastic Partitioning

Another way of defining partitions is to derive them from a function, which need not be deterministic. A classic example of this is the way Random Forests (see

Section 2.3.5) are built; what Breiman calls “Random Input Forests” (Forest-RI) [12] are made by choosing a single feature at random to use as a candidate split criterion. This is equivalent to creating a partition with dimension 1, where membership in the partition is stochastic (in classic Random Forests, the feature to use in the partition is drawn from a uniform distribution over the set of features being used for the tree in question).

Breiman also discusses “Random Combination Forests” (Forest-RC) which are made by choosing some small fixed number of features L, again drawn uniformly from the available features. After partitioning, the L features are projected down to a scalar (i.e., the candidate decision criterion is a weighted linear combination of the selected features).

While previous work with random forests has always done partitioning by selecting features from a uniform distribution, there is no reason other distributions cannot be used. In particular, just as spatially aware heuristics can be used with deterministic partitions, they can also be used with stochastically generated partitions. In Chapter 8, we discuss a variant in which partitions are not chosen by sampling from a uniform distribution, but instead from a Gaussian distribution with a randomly selected mean; this can be seen as a non-deterministic relative of the fixed-size spatial regions centered on a spatial locus described earlier.

4.3 Spatial Statistics

To analyze the spatial behavior of different learning algorithms, we need tools to measure and describe spatial structure in data. In this work, we use two different classes of methods for analyzing spatial structure. The first is to measure how inter-feature variance and correlation changes with spatial distance between features. This gives us a quantitative measure but tells us only about pair-wise relationships. In addition, it is only meaningful as an average. The second is a more holistic qualitative analysis that can be done by using dimensionality reduction to project the data onto a 2-dimensional plane so it can be visualized. This lets us see what kind of clusters or structure exists in the data and compare the amount of structure present in different datasets.

Our interest in spatial locality means that we would be wise to draw from the field of statistics for spatial data, also known as spatial statistics. While some of the techniques used in geostatistics, for example, will not be relevant to all types of data, there are a number of principles and techniques that can productively be borrowed and used in our analysis of the impact of partitioning, particularly when using a spatially meaningful partitioning function.

The field of spatial statistics was developed largely for analyzing data gathered from processes that were expected to vary based on the location at which the sample was taken. Common examples are geological, meteorological, or epidemiological data. Many of the principles of spatial statistics originated from the study of geological data, referred to as geostatistics. Example problems include prediction of crop yields based on field location, prediction of weather, temperature, pollution, and other environmental conditions based on both temporal and spatial information, and characterization of local epidemiological problems, such as discovery of cancer clusters [20].

The basic problem addressed by spatial statistics is that most statistical models start with fairly strong assumptions about the nature of the distribution being modeled. In particular, it is very common to assume that the data are sampled independent and identically distributed (IID) from a distribution which is stationary and homoscedastic.

This can be problematic in several ways. First, the model may make assumptions that are *too strong* and are violated by the data, with the result that the model may seem to capture trends that are not actually present in the data. Effectively, the space represented by the model is strictly smaller than the space represented by the true distribution. If we consider modeling as a search process, this effectively means the hypothesis space excludes the true solution. Depending on the degree to which the assumptions are violated, the result may be a model that is wildly inaccurate. This can be highly problematic, but it is generally fairly easy to discover by evaluating the performance of a model on novel (or held-out) data.

Alternatively, the model may make assumptions that are *too weak*, with the result that the model may have difficulty capturing trends that are present in the data. This is often more subtle, since there may be no drastic disagreement between the model and the data; here, the space represented by the model may be *larger* than the space represented by the distribution. The result tends to be a model that underperforms, but not in any way that demonstrates a clear mismatch with the underlying distribution.

Spatial statistics is an attempt to address the latter problem as much as the former. In particular, it allows us to incorporate extra information into our model to describe the spatial (or spatio-temporal) circumstances under which a particular sample was taken. This means that spatially localized trends can be captured, which would be impossible to characterize if we treated all samples as being IID.

We will follow the notational conventions of Cressie [20] for describing spatial statistics. Given a distribution $\mathbf{Z}$, the basic model used for spatial statistics is to assume that a each data sample has a spatial location $\mathbf{s} \in \mathbb{R}^n$, where location is generally modeled as position in an n-dimensional Euclidean space. For any location $\mathbf{s}$, $\mathbf{Z}(\mathbf{s})$ is the distribution (or random process) evaluated at that location. Note that, unless this is a deterministic process, multiple samples drawn from the same spatial location need not agree. Temporal information can either be included as a part of the "location," or explicitly separated into its own variable, making a spatio-temporal process $\mathbf{Z}(\mathbf{s}, t)$. A dataset is then a set of samples drawn at particular locations; for a set of locations $D \subset \mathbb{R}^n$, we can write the random process as:

$$\{\mathbf{Z}(\mathbf{s}) : \mathbf{s} \in D\}$$

The primary assumptions built into this model are that the distribution may vary as a function of spatial (or temporal) location, but that for any given location the distribution will be stationary and samples will be drawn IID from that distribution. It is also generally expected that the distribution will be smoothly varying to some degree, meaning that nearby spatial locations will tend to have similar distributions; in other words, the locations are treated as values from a continuous (n-dimensional) range, rather than as a set of discrete labels.

4.3.1 Variograms

The primary statistical tool we will borrow is the *variogram*, which provides a way of measuring how important spatial distance is to statistical correlation (i.e., how smoothly does $\mathbf{Z}$ vary as a function of spatial distance). Given two samples from a spatial distribution $\mathbf{Z}$, sampled at locations $\mathbf{s}_1$ and $\mathbf{s}_2$, $(\mathbf{Z}(\mathbf{s}_1) - \mathbf{Z}(\mathbf{s}_2))$ is the difference between the *values* of those samples, and $(\mathbf{s}_1 - \mathbf{s}_2)$ is the *distance* (typically Euclidean) between spatial locations at which those samples were taken. If there exists a $\gamma(\cdot)$ such that

$$var(\mathbf{Z}(\mathbf{s}_1) - \mathbf{Z}(\mathbf{s}_2)) = 2\gamma(\mathbf{s}_1 - \mathbf{s}_2), \forall \mathbf{s}_1, \mathbf{s}_2 \in D,$$

where D is the set of all possible sampling locations, then $2\gamma(\cdot)$ is called a *variogram*; note that it is a function of the distance between sampling locations.

In geostatistics, variograms are often used for a type of predictive modeling called *Kriging* [20, 41], which means that the assumptions must be fairly closely held. Kriging is a predictive method that uses variograms to minimize mean-squared-error, and essentially generates a predicted value for a novel location by interpolating between samples from nearby locations.

Kriging is performed using $\gamma(\cdot)$, which is referred to as a *semi-variogram*; if we let h be the distance between an existing sample and a location we would like to predict a value for, $h = \mathbf{s}_0 - \mathbf{s}_{new}$, we can write the semivariogram as:

$$\gamma(h) = cov(0) - cov(h)$$

where $cov(\cdot)$ is the covariance with respect to $\mathbf{s}_0$, defined in terms of the expected value E:

$$
\begin{aligned}
cov(0) &= var(\mathbf{Z}(\mathbf{s}_0) \\
cov(h) &= E\left\{\mathbf{Z}(\mathbf{s}_0 + h) \cdot \mathbf{Z}(\mathbf{s}_0)\right\} - E\left\{\mathbf{Z}(\mathbf{s}_0)\right\}^2
\end{aligned}
$$

For a variogram (or a semivariogram) to be defined properly, some fairly strong assumptions about the data must be made, including but not limited to it being sampled IID from a static, isotropic spatial distribution. Kriging was developed as a means of making predictions for geological data (for mining purposes), which is a setting in which the variogram assumptions tend to hold reasonably well. In the case of the image data we will be analyzing, the assumptions are less accurate; in particular, in geostatistics multiple samples from the same location will tend to give the same (or at least similar) results. In image data, there is no expectation that a particular pixel location will have similar values in different images. This means applying Kriging to images would make no sense at all.

Fortunately, we will mainly use variograms as a descriptive tool, so the fact that a variogram would make a poor predictive model for our data is not a problem. Even if the assumptions are not met perfectly, an empirical estimate of a variogram can still be a useful tool for measuring the presence and extent of local structure in data, and it is in this capacity we will use them in our experiments.

For image data, where features are sampled from an evenly spaced grid (referred to in spatial statistics as lattice models), a variogram can almost be thought of as a kind of histogram. The 'bins' are inter-feature (i.e., sampling location) distances rounded to the nearest integer, and the height of each 'column' is the mean of the variance of the sample value differences between all feature pairs that are that distance apart.

To generate a variogram plot, for every pair of features (i, j), take the element-wise difference $X^i - X^j$, and then compute the variance of the resultant set $var(X^i - X^j)$. Next, for each distance d in the range $[0, d_{max}]$, compute the mean of all the feature-pair-variances between feature-pairs that are separated by distance d. These averages are then plotted against the distances.

We can make a similar histogram using covariances, correlations, mutual information, or any other pair-wise statistical measure between feature-pairs of a given distance. The terms *covariogram* ($C(\cdot)$) and *correlogram* ($\rho(\cdot)$) can be used to describe these functions, but much like the variogram they are frequently not well-defined in practice, and we must use mean-based estimates:

$$cov\left(\mathbf{Z}(\mathbf{s}_1, \mathbf{s}_2)\right) = C(\mathbf{s}_1 - \mathbf{s}_2) \forall \mathbf{s} \in D$$

$$\rho(h) = C(h)/C(0)$$

For a more in depth treatment of variograms, covariograms, correlograms, and spatial statistics in general, the reader is directed to Cressie's [20] and Gelfand *et al.*'s [41] books on the subject.

4.3.2 t-SNE

While pairwise statistical plots offer a nice quantitative depiction of spatial structure, they are limited in scope both by only measuring pairwise relationships and by the fact that the assumptions that underlie the statistical models are poorly conformed to. We wanted a more qualitative and holistic method to complement these measures; by embedding our data in a 2-dimensional space, we enable easy visual comparisons of how much structure is present.

To embed our data into a plane for visualization, we used *t-Distributed Stochastic Neighbor Embedding* (t-SNE) [27]. t-SNE was chosen because it is a dimensionality reduction technique that is particularly designed for visualizing high dimensional data. The method builds a map in which distances between points reflect similarities in the data. It embeds high-dimensional data in lower dimensional space by minimizing the discrepancy between pairwise statistical relationships in the high and low dimensional spaces.

t-SNE is an embedding algorithm in much the same way as Multi-dimensional Scaling (MDS) [74, 122, 123] or Locally Linear Embedding (LLE) [115] (see Section 2.3.2.3); the primary difference is the optimization being done. MDS is designed to preserve overall distances, meaning that it is as important to maintain pairwise distance even for pairs that are very far apart. t-SNE is closer to techniques like LLE, in that it focuses more on preserving distances between pairs that are close by in the original data-space. Unlike LLE, however, t-SNE does not encode point locations in terms of their neighbors, but as absolute locations the way MDS does.

For a dataset of n points, let $i, j \in [1, n]$ be indices, and let $x_i \in \mathbf{X}$ and $y_i \in \mathbf{Y}$ refer to the ith data point of the original dataset and the low-dimensional equivalent respectively. Given a candidate embedding, t-SNE first calculates all pairwise Euclidean distances between data points in each space. The pairwise Euclidean distance between x_i and x_j is used to calculate a conditional probability, $p_{j|i}$, which is the probability that x_i would pick x_j as its neighbor. This probability is based on a Gaussian centered at x_i, with a variance based on sampling density (in densely sampled regions, the variance will be smaller than in more sparsely sampled regions). Similarly, pairwise conditional probabilities $q_{j|i}$ are calculated for each pair (y_i, y_j) in the low-dimensional embedding. As an objective function, t-SNE tries to minimize the discrepancies between the conditional probabilities for corresponding pairs in the high dimensional and low dimensional spaces by using *Kullback-Leibler* divergence (KL divergence):

$$D_{KL}(P\|Q) = \sum_i P(i) \log \frac{P(i)}{Q(i)}$$

This is an intractable global optimization problem, so gradient descent is used to find a local optimum.

One drawback of t-SNE is that for large, high-dimensional datasets, even the local search can be quite slow. In such cases, PCA is sometimes used as a pre-processing step to speed up the computation and suppresses high-frequency noise [27]. A typical example might retain the top 30 eigenvectors, and project the original data into the eigen-basis. t-SNE would then be applied to this 30-dimensional dataset to reduce it to a 2-dimensional set for visualization.

The resulting 2D plots make the structure (or lack thereof) readily apparent. Since the optimization is done on pair-wise distances between original data vectors, feature ordering (i.e., spatially local structure) in the high-dimensional data does not change the qualitative properties of the low-dimensional data significantly. Moreover, since the mapping is non-linear and non-parametric, it is relatively insensitive to whether information is encoded using sparse or distributed representations. As a result, t-SNE allows us to examine the presence of structure without having to worry about the form of that structure impacting our analysis.

4.4 Identifying Structure in Image Data

Since spatial structure is a statistical pattern, theoretical analysis is of limited use without empirical evaluation of the patterns in actual data. We performed a series of experiments on several different image datasets, both to look for statistical evidence of spatial structure and to evaluate the impact of applying a spatial bias to different techniques. See Chapter 3 for a description of the original datasets.

4.4.1 Randomly Permuted Datasets

Since we wanted to evaluate the ways different learning algorithms interacted with spatial structure, we created *permuted* versions of each of the datasets listed above to produce a non-spatial baseline.

To create these permuted data sets, we generated random permutations of the same length as the data vectors. A random permutation of length k is simply a list of the integers $[1, k]$ in a random order; the likelihood of a given value being assigned to a given index is uniform for all value/index pairs. We then use the permutation to define a mapping we can apply to our data vectors: each dimension i of the input vector is mapped to the dimension corresponding to the ith value of the random permutation in the output vector. The mapping is 1-to-1 and onto, and is lossless from an information-theoretic standpoint, since we are only changing the order the features appear in the input.

By applying a random permutation to each vector in a dataset, we get a permuted dataset; note that the *same* random permutation is applied to *all* the input vectors, including both training and testing samples. This means that while the generation of the mapping is randomized, once a permutation has been generated its operation on input vectors is deterministic (and easily reversible). See Figure 4.3 for some examples of images before and after random permutation.

For our experiments, the purpose of the permutation is to disrupt any spatial structure that might exist, while leaving all non-spatial structure intact. Any algorithm that does not consider spatial structure should show the same performance on both an original dataset and the permuted version of that dataset. An algorithm that does rely on spatial structure, however, should show lower performance when applied to the permuted data instead of the original data.

4.4.2 Locality Analysis

For four standard image datasets, we generated spatial information plots as described in Section 4.3. Here, we show the variogram and mean-correlation plots for both the original and the permuted versions of the MNIST dataset (see Section 3.1),

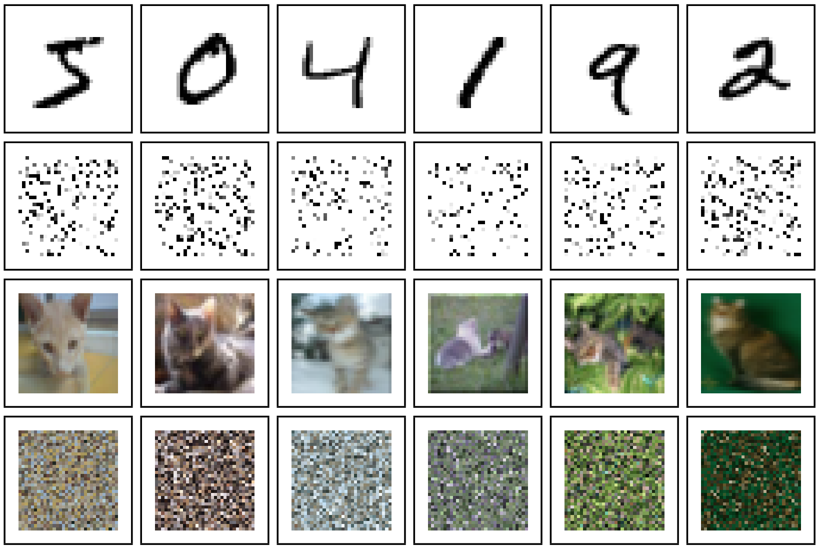


Figure 4.3: *Randomly permuted image examples. The first row shows original MNIST images, the second row shows permuted versions of the same images. The third and fourth rows do the same for CIFAR-10 images.*

the SVHN dataset (see Section 3.2), the CIFAR-10 dataset (see Section 3.3), and the ImageNet ILSVRC2015-Places2 dataset (see Section 3.4).

Figure 4.4 shows the variograms and mean-correlation plots for both the original MNIST training data and for the randomly permuted version. To generate a variogram, we calculated $var(X^i - X^j)$ for each pair i, j, and then for each distance, plotted the mean of all the variances of feature pairs that distance apart. For the mean-correlation plots, we calculated $corr(X^i, X^j)$ for each pair i, j, and then for each distance, plotted the mean of the correlations of feature pairs that distance apart.

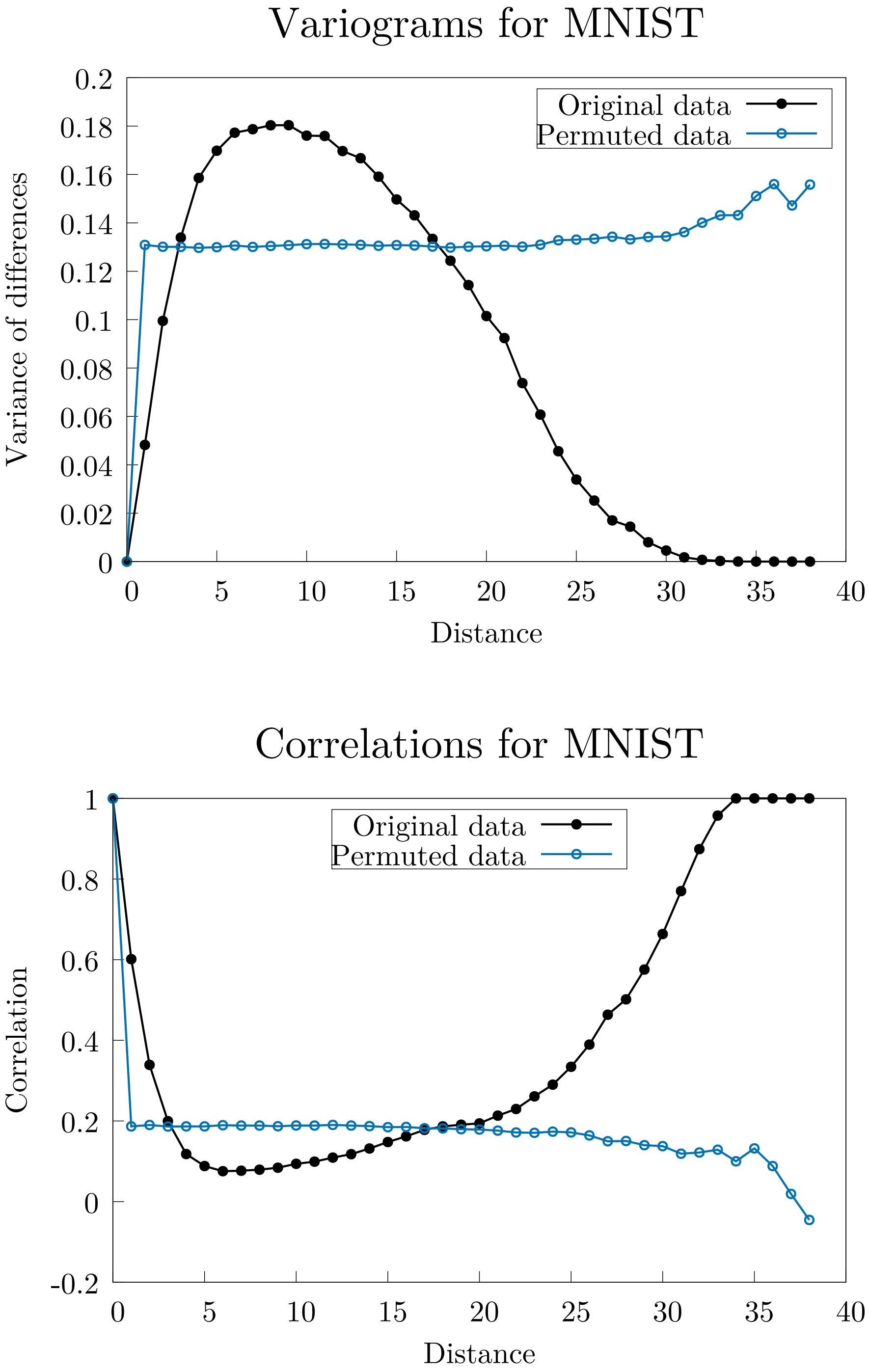


Figure 4.4: *Variogram and correlation plots for the MNIST dataset, before and after random permutation of feature ordering.*

Figures 4.5, 4.6 and 4.7 show the same process applied to the SVHN, CIFAR-10, and ImageNet ILSVRC2015-Places2 datasets, respectively.

First, we note that for each dataset, the variogram plots and the correlation plots tend to contain qualitatively similar information; while they are not exactly the inverse of one another, the same basic trends are present. We include both versions here for completeness and to demonstrate the similarities, but for most purposes it should be sufficient to simply pick one type of plot and use it.

In Figure 4.4, we can examine variogram and mean-correlation plots for original and permuted versions of the MNIST dataset to draw conclusions about the spatial structure present in that data. At distance 0, there is no variance and perfect correlation, since any feature always has the same value as itself, regardless of the data source. Beyond that, however, the variance and correlation plots for the permuted dataset are basically flat, indicating that after permutation there remains no significant relationship between statistical information and spatial distribution for feature pairs. The deviations from this trend in the last few points in each plot are caused by the reduced sample size available for the extreme distances (i.e., only pixels in opposing corners). Generating a new random permutation causes significant fluctuation in these last few values, but the rest are stable.

This lack of spatial structure is the expected behavior for *any* dataset that has been permuted randomly, since the permutation is specifically intended to make the

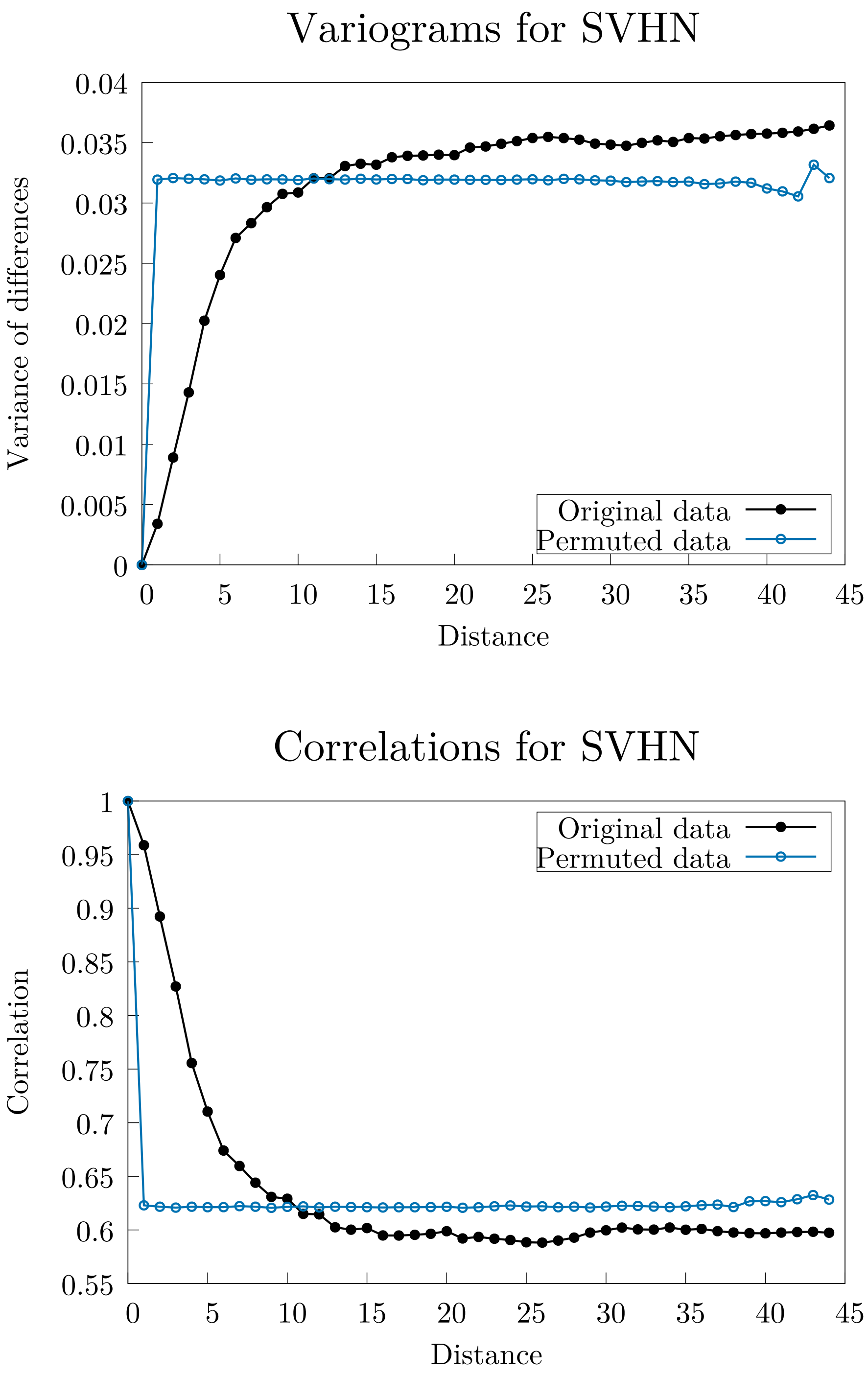


Figure 4.5: *Variogram and correlation plots for the SVHN dataset, before and after random permutation of feature ordering.*

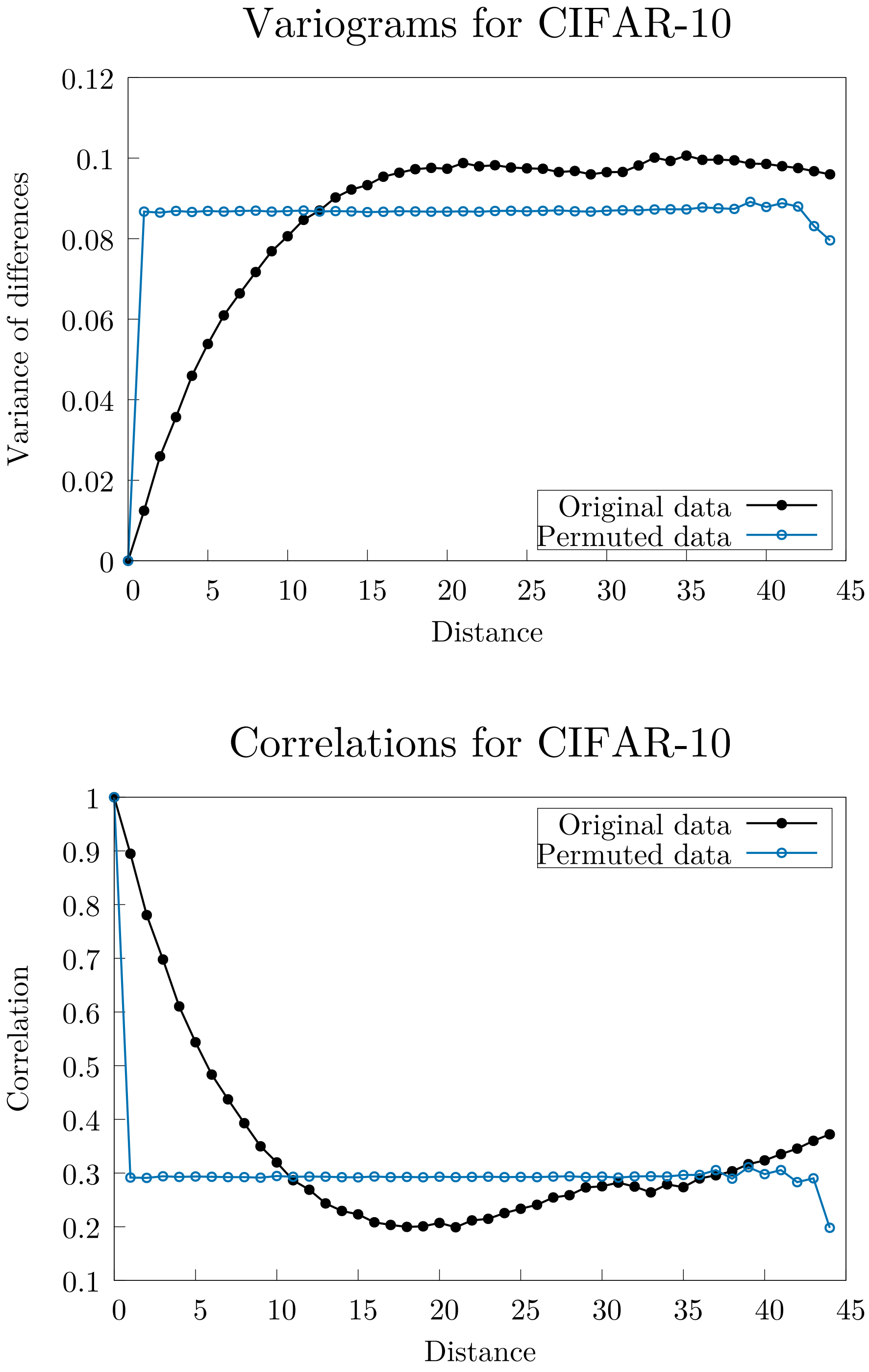
spatial distribution uniformly random. Examining the plots for the other datasets (Figures 4.5–4.7) demonstrate that this property holds for the permuted versions of all datasets, just as expected.

The plots for the original data, on the other hand, exhibit some interesting structure. The first thing to note is that for all datasets, adjacent features have high correlation, and the variance of their inter-pair differences is low. As the feature-pairs get farther apart, the degree of correlation drops off, but the rate differs between the datasets.

The MNIST data (Figure 4.4) shows the most rapid falloff; by a distance of 5 pixels, there remains on average little statistical relationship between feature pairs. This is likely related to the average line-width of the hand-written digits, which is generally 2-3 pixels. The fact that correlation improves again as distances increase is an artifact of the construction of the MNIST dataset. In particular, the MNIST images all have a centered digit surrounded by a white border. Since background pixels have the same value in *all* images, correlation scales with the likelihood that both members of a pair are background. The greater the distance between a pair of features, the more likely that both features will be a part of the background; in fact, beyond a distance of 30 (note that maximum distance is $28 \times \sqrt{2} \approx 40$) both pixels are *guaranteed* to be background pixels, meaning they are guaranteed to always have the same value. Note that most correlation measures are actually undefined when correspondence is perfect; for the purposes of Figure 4.4, we have set these values to 1.

The SVHN data (Figure 4.5) shows a falloff that is slower than that shown by the MNIST data, but still has a relatively steep slope for the first 5-10 pixel-units of distance. Since this dataset is also composed of numerical digits that have been centered and scaled, it makes sense that we would see a similar pattern. However, unlike the MNIST images, the SVHN images are not otherwise preprocessed. This means we do not see the increase in correlation towards the end of the plot that we do with MNIST; there is no universal background color, so distant pixels will not have artificially inflated correlations. However, the backgrounds do tend to be somewhat uniform within any given image, since they tend to be mostly paint, shingle, brick, or whatever other surface the house numbers are attached to.

The CIFAR-10 data (Figure 4.6) has an even slower falloff of correlation than the SVHN data, with both a shallower initial slope and a gentler corner at the top. This is due to the fact that the CIFAR images have much greater variability than the previous datasets (in terms of the shape, scale, location, etc., of the foreground object, as well as in terms of backgrounds). Here, it takes around 15 pixels distance before there appears to be little remaining statistical relationship on average; since the images are 32×32 pixels in size, this is about half the width of an image.



Variograms for CIFAR-10

Correlations for CIFAR-10

Figure 4.6: *Variogram and correlation plots for the CIFAR-10 dataset, before and after random permutation of feature ordering.*

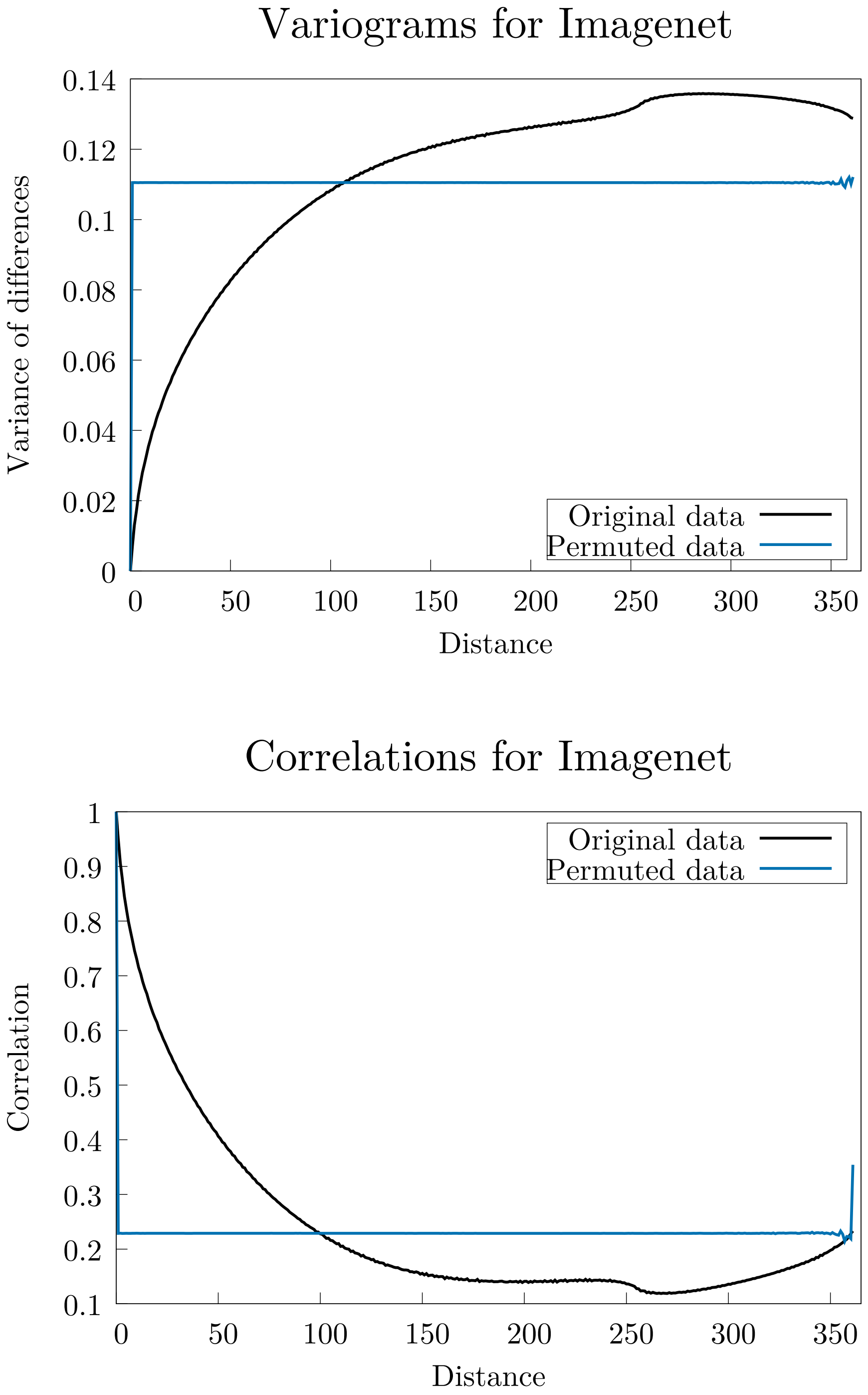


Figure 4.7: *Variogram and correlation plots for the ImageNet ILSVRC2015 dataset, before and after random permutation of feature ordering.*

The ImageNet data (Figure 4.7) show a curve with a very similar overall shape to the CIFAR-10 data but with a different range. Where the CIFAR-10 images are 32×32 pixels, the ImageNet images are 256×256 pixels, so the overall range of possible distances is much greater in the latter dataset. Proportional to image resolution, the curves are very similar, but in absolute terms the ImageNet plot has a much shallower slope. Additionally, the ImageNet data produce a variance curve that does not fully plateau at half the image width the way the CIFAR-10 curve does, likely because the foreground objects are even less uniform in type and placement.

Both the ImageNet curves and the CIFAR-10 curves show an odd “bump” near the end; in both cases, this occurs at the point where the distance becomes greater than the width of the images. In effect, pairs with distances greater than this can only come from areas near diagonally opposed corners, which causes the number of samples to drop rapidly. This effect is not present in the MNIST or SVHN data, because those datasets have more uniform backgrounds.

Taken together, these results are a strong empirical demonstration that image data does exhibit spatially localized structure, and that this structure can be eliminated by randomly permuting the ordering of the features in a dataset.

The pattern exhibited by the ImageNet dataset is likely to be the most representative of arbitrary natural image data (i.e., images that have not been significantly preprocessed and do not represent a narrow or uniform class of subjects). The fact that the CIFAR-10 data show the same pattern in miniature gives us confidence that this is the case, while also highlighting the fact that spatial structure is a fundamental property of the underlying data, and not a function of image resolution.

Chapter 5

Spatial Bias in Convolutional Networks

5.1 Motivation

Proponents of convolutional networks have long expressed a belief that spatial information is being exploited by CNNs, and that this one advantage they have over unstructured approaches (see for example page 6 of [82]). No citations or evidence are given; instead, this claim is generally presented as being so obvious that no evidence is required. It is a sensible enough claim that this has tended to be sufficient for most readers, but it has also never been terribly important. The performance numbers produced by CNNs demonstrate the overall utility of the method, which tends to be the point of such papers, so the veracity of this claim is largely irrelevant.

We in no way dispute the claim that CNNs exploit spatially local information; rather, our work relies heavily on it being true. Since we are explicitly looking at spatially local information, we wanted empirical evidence that spatial information was being used by CNNs, largely so we can demonstrate that other techniques can be induced to produce similar behavior under similar circumstances. This will then be evidence that these other methods are making similar assumptions (and making use of similar information) to CNNs.

5.2 Methodology

The simplest way to test for the impact of structure on CNNs is to compare their behavior in the presence and absence of that structure. In particular, we used several standard datasets that CNNs are known to perform well on (see Sections 3.1 and 3.3), and compared the behavior of CNNs with fully connected feed-forward networks on these datasets. We then randomly permuted each dataset (as described in Section 4.4.1) to produce data that was identical in all ways other than the spatial ordering of the features, and applied both techniques again.

The expectation is that a method that makes no use of spatially local information will perform identically on both original and permuted versions of the data. For techniques that rely on spatial information, permuting the data should result in lower performance than can be achieved on the original data.

It should be noted that while we attempted to find reasonable values for the model parameters, better performance for these techniques is reported elsewhere in the literature; our goal was not to achieve state-of-the-art performance, but rather to observe how performance is impacted by the presence (or absence) of spatial information.

5.3 Experiments

The CNN we trained for the MNIST dataset (see Section 3.1) used two convolutional layers, with 16 and 32 filters respectively. All convolutional kernels were 5×5 with local contrast normalization. Pooling regions were 2×2 and used max-out pooling. The network was topped with two fully-connected layers, with 512 and 128 units respectively. The network was trained using SGD with mini-batches and batch normalization.

We also trained a standard multi-layer perceptron (MLP) separately on the same data. The architecture and training was the same as for the two-layer output network used for the CNN; 512 units in the first hidden layer, 128 in the second, and trained using SGD with mini-batches [90] and batch normalization [65].

We trained the same models on both the original MNIST data and a randomly permuted version of the MNIST data. In each case, the training and testing sets provided as part of MNIST were used for training and evaluation. The results of these experiments can be seen in Table 5.1.

For the CIFAR-10 dataset (see Section 3.3), we used a VGG-like network [125], trained using SGD with mini-batches [90], batch-normalization [65], and dropout [57]. The fully connected network on top consisted of two layers, each with 512 units.

The MLP architecture used for the CIFAR-10 dataset was two hidden layers, the first with 4092 units and the second with 2048 units. Like the CNN, the MLP was trained using dropout and batch-normalization.

Again, the models were trained on both the original CIFAR-10 data and on a randomly permuted version of the CIFAR-10 data.

Each experiment was replicated 10 times, and the results reported in in Table 5.1 represent means over these 10 samples.

	MNIST	Permuted MNIST	CIFAR-10	Permuted CIFAR-10
MLP	98.41%	98.40%	60.69%	61.27%
CNN	99.54%	97.76%	92.34%	56.46%

Table 5.1: *Classification accuracy on MNIST and CIFAR-10 test sets (average over 10 trials). MLP is a feed-forward multi-layer perceptron, CNN is a convolutional neural network. Performance is reported on both original datasets, and randomly permuted versions of the datasets.*

5.4 Discussion

In Table 5.1, we can see that CNNs are able to outperform MLPs on both MNIST and CIFAR-10 by wide margins, just as expected. CIFAR-10 is known to be a significantly more difficult problem, so it is no surprise that accuracy is always better on MNIST, even though the networks used on the CIFAR-10 data are more powerful.

To test for statistical significance, we applied a paired Wilcoxon signed rank test to each pair of experimental conditions, with the null hypothesis in each case being that the two conditions were equivalent.

For our purposes, the result that is of primary interest here is the comparison between performance on permuted data and unmodified data. In both the case of MNIST and the case of CIFAR-10, we find that the MLPs behave the same way regardless of whether the data have been permuted or not (p-values for the null hypothesis are $\gg 0.05$). This is a strong indication that the MLPs are not making use of any spatial structure.

The CNNs, on the other hand, show significantly worse performance on the permuted data than on the original data (p-values are < 0.006 for MNIST and < 0.002 for CIFAR-10). In fact, for both MNIST and CIFAR the CNN on permuted data performs significantly worse than the basic MLP (p-values are < 0.006 for MNIST and < 0.002 for CIFAR-10). In the case of the MNIST data, where the exact same architecture as the MLP was used as the output network for the CNN, this indicates that not only is the CNN making use of spatial information when it is present, it is reliant on the spatial information to such an extent that removing the spatial data makes the CNN perform worse than the identity function.

5.5 Conclusions

Overall, these results strongly support the hypothesis that CNNs not only make use of spatially local information, but are reliant upon it. Thus, we conclude that CNNs have an inductive bias that assumes the presence of spatially local structure, and can move on to the question of how such a bias can be introduced into other methods.

In CNNs, this bias is a result of the network architecture. Any given unit in a convolutional layer has incoming connections from a small, spatially localized group of the units in the layer below. The size of this group is the size of the convolutional filter that the unit represents. This is effectively a spatial partitioning of the input data.

Chapter 6

Partitioned Training of RBMs and DBNs

6.1 Motivation

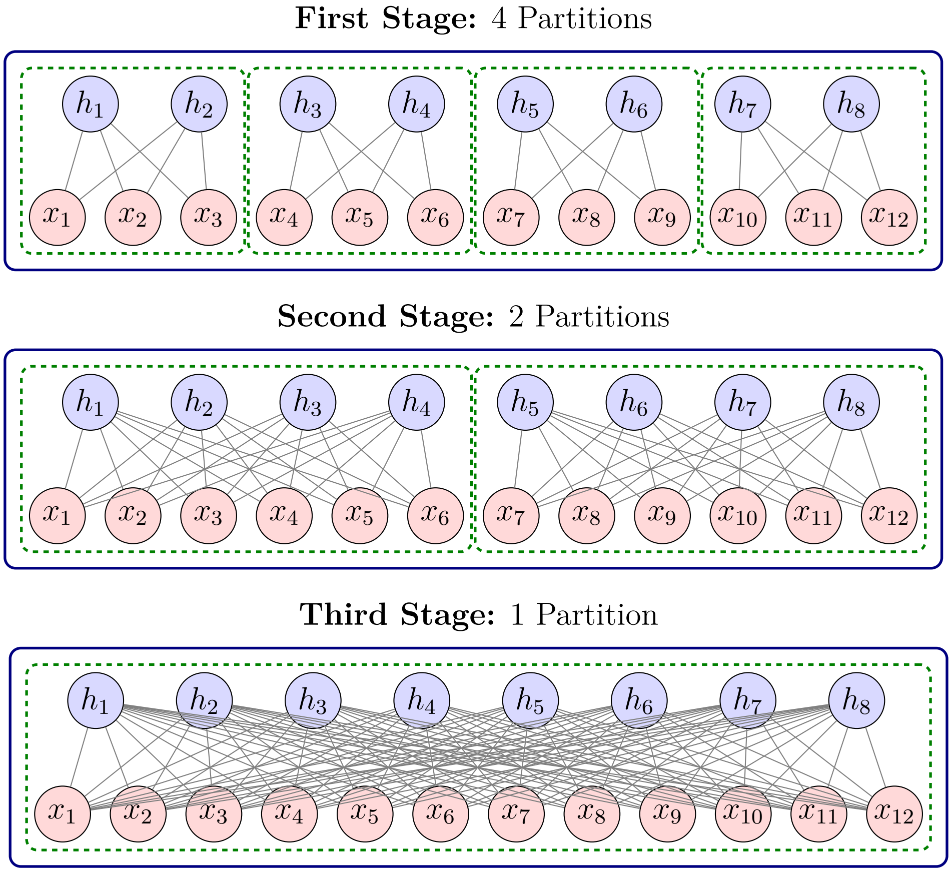
Having demonstrated that CNNs rely on assumptions about spatially local structure, we wanted to incorporate this type of assumption into other models. Restricted Boltzmann Machines (RBMs) are another connectionist technique associated with deep learning, but they are much more free-form networks; where CNNs have complex partitioning and weight-sharing architectures, RBMs are normally fully-connected (see Section 2.1.3.2 for more information about RBMs). This leads to a behavior on spatial data more like multi-layer perceptrons than like CNNs (see Chapter 5 for what this behavior looks like for spatial and non-spatial data).

RBMs are an interesting candidate for introduction of a spatial bias in their own right, but they are also the building blocks for Deep Belief Nets (DBNs), which have been the primary alternative to CNNs within the deep-learning literature (See Section 2.2.3 for more information about DBNs).

6.2 Partitioned RBMs

As a means of utilizing spatially local information in RBMs, we proposed a training method for RBMs that partitions the network into several sub-networks that are trained independently and incrementally re-combined until only a single, all-inclusive partition is left [136]. With the *partitioned RBM* method, training involves several levels of partitioning and training. At each level, the RBM is partitioned into multiple RBMs as shown in Figure 6.1. In this figure, the partitions do not overlap, although [136] also demonstrates additional improvement in accuracy when some amount of overlap is permitted. See Section 4.2 for a more detailed discussion of partitioning functions.

In this case, the advantage of a disjoint partitioning is that all the RBMs at a given level can be trained simultaneously in a way that is both data- and task-parallel.


First Stage: 4 Partitions

Second Stage: 2 Partitions

Third Stage: 1 Partition

Figure 6.1: *Example partitioning approach for an RBM, using 4 partitions in the first stage, 2 in the second stage, and 1 in the final stage. Note that the learned weights are retained from the previous stage.*

If the partitions overlap, the sub-networks can no longer be trained independently, because multiple sub-networks will be updating the same parameters. While some parallelism can still be achieved, extra communication and synchronization mechanisms are required; Fortier *et al.* [36, 37, 38] have described one approach for allowing parallelization and distribution of optimization for overlapping networks.

Once a partitioning function has been defined, each partitioned RBM is trained using a set of sub-vectors from a corresponding partition of the training dataset; in effect, each RBM is trained on “instances” that contain only the features that correspond to the input nodes assigned to that RBM partition.

Once these RBMs have been trained, a new partitioning with fewer splits is created, which forms the basis for the next level up. Figure 6.1 shows an example in which the first layer has 4 distinct RBMs, the second layer has 2, and the final layer has 1. The power of the method comes from the fact that all levels share the same weight matrix. This means that during training, each RBM updates its own part of the globally shared weights; because the partitions do not overlap, there is no data dependency (which would prevent easy parallelization), but there is ultimately only one set of weights. The result is that the later levels begin their training with weights that have been pre-initialized by the earlier levels. The only exception is the units that were not connected in the previous partitioning; here, new connections must be initialized randomly, but this is a small percentage of the overall weights at any given stage.

The reason this is useful is that, when RBMs are small (in terms of number of nodes/weights being updated), they can be trained more quickly. While the results of the disjoint training will not be perfect (because they lack the full data vectors, they will be missing some relationships), they can be allowed to run for many more epochs

Algorithm 6.3 Pseudocode for the partitioned RBM training method (see Algorithm 2.1 for the contrastive divergence algorithm). Note that the same weight matrix $\mathbf{W}$ is used for the entire training process. $a \sim b$ indicates a is sampled from b. See Section 4.2 for details about partitioning functions.

```
Input: $\mathbf{D}$: training data set, $\pi_j$, $j \in [1..k]$: partition functions, $\alpha$: learning rate.
Output: $\mathbf{W}$: weight vector,
for $j = 1$ to $k$ do
    $\mathbf{D}_{\pi_j} \leftarrow \pi_j(\mathbf{D})$
    for all partitions in $\pi_j$ do
        repeat
            $\mathbf{x} \sim \mathbf{D}_{\pi_j}$
            CD-1$(\mathbf{x}, \mathbf{W}, \alpha)$
        until convergence
    end for
end for
```

in a given time frame. After each iteration, the RBMs become larger from recombining, but training requires many fewer epochs than normal to converge because most weights are much closer to their optimal values than would be the case with random initialization. This enables the overall Partitioned-RBM training method to achieve higher performance over a given training interval than using the traditional RBM training method, even though both ultimately produce networks that are identical in structure. See Algorithm 6.3 for pseudocode.

6.3 Partition Trained DBNs

Since unsupervised pre-training of DBNs is an iterative process of training RBMs to reproduce their input, it is fairly straightforward to substitute our partitioned RBM training method for the standard one. The result is a partition trained DBN.

Like a normal DBN, a partition trained DBN is trained one layer at a time, using Partitioned-RBM training for each layer. Unlike the Convolutional Deep Belief Network (CDBN) model [87], the resulting network is still a traditional DBN in its architecture and operation. Like Partitioned-RBM, a CDBN uses partitioning for its filter-response units, but the partitions are permanent and persist even in the final trained model. The outputs are then combined with probabilistic max-pooling, resulting in an architecture that looks more like a CNN than a DBN. A CDBN is trained using standard sampling techniques. What differs is the architecture and the probabilistic max-pooling operation. Our partitioned training method, on the other hand, is not a new architecture but rather a new method of training a standard DBN architecture. Thus, a partition trained DBN can be compared directly to a conventionally trained DBN. Doing the same comparison with a CDBN is much less informative, as the two models have fundamentally different representational power.

6.4 Methodology

We compared standard RBMs to Partitioned RBMs on the MNIST dataset; see Section 3.1 for details on the dataset. This dataset was chosen largely because it is a good fit for RBMs of any kind; its nearly-binary nature and relatively low resolution allow for RBMs to be trained relatively efficiently, and it is easy to compare our results to the existing literature.

We measured the performance of the RBMs using reconstruction error, which is defined to be the mean difference between the original and reconstructed images. We used a binary reconstruction error, using a fixed threshold value of 30 to map pixels in the range $[0 - 255]$ to a binary 1 or 0 for the original images.

To get the reconstructed image from the RBM, we first sampled a hidden node activation vector from the RBM model for a given input image, and then sampled a visible node activation vector based on the sampled hidden activations. The resulting vector is also binarized and used in conjunction with the original vector to calculate reconstruction error for length n vectors:

$$E(\mathbf{x}, \mathbf{x}') = \frac{1}{n} \sum_{i=1}^{n} I(x_i \neq x_i') \tag{6.1}$$

To examine what statistical information is being exploited by the Partitioned RBMs, we performed a set of experiments training RBMs on both the original MNIST data and a randomly permuted version of the dataset (see Section 4.4.1). By comparing the performance of the different types of RBMs on both the original data and the permuted data, we can see whether they make use of spatially local structure.

We also ran experiments using Class RBMs [78], meaning that an extra set of visible nodes containing class labels was added to the fully connected RBM. These class label values were clamped during training, and then used for class prediction on testing data, which allows a more direct comparison with the supervised learning techniques used in other chapters of this dissertation.

Each RBM was run for 20 iterations, and the error rates reported are the mean values from 10-fold cross-validation (not using MNIST's predefined split between training and test data).

We also performed experiments in which the partitioned training method was used for each layer of a DBN and compared this to architecturally equivalent DBNs trained using the normal method. As with the RBMs, our performance measure was reconstruction error, calculated by propagating an input image all the way to the last hidden layer of Deep Belief Network, then reconstructing it by reverse propagation to the visible layer, and applying Equation 6.1 as above.

To examine whether spatially local features are being disrupted in the DBN training process, we constructed a 2-layer DBN, where each layer is composed of 784 hidden nodes. One version of this network was trained using the partitioned RBM

Configuration	Samples	Original data	Permuted Data
Single RBM	60000	2.46%	2.44%
Single RBM	30000	2.55%	2.55%
RBM-28	60000	3.32%	7.00%
RBM-20	50000	2.20%	6.42%
RBM-15	40000	1.87%	6.13%
RBM-10	30000	1.64%	5.00%
RBM-5	25000	1.49%	3.88%
RBM-2	20000	1.44%	2.89%
RBM-1	30000	1.42%	2.24%

Table 6.1: *Reconstruction error of traditional and partition-trained RBMs on the MNIST dataset*

training method, and the other was trained conventionally. We then calculated the variograms and t-SNE plots for the output of hidden nodes at each layer, using the methods described in Section 4.3.

6.5 Results and Discussion

6.5.1 Partitioned-RBM results

Table 6.1 shows the results of our RBM experiments. In the configuration column, Single RBM indicates the RBM was trained on the full data vectors (i.e., no partitioning); RBM-n indicates a Partitioned-RBM with n partitions. RBM-1 is equivalent to Single RBM in terms of its configuration, but the RBM-1 is trained on fewer samples, since each successive RBM-n configuration starts with the output of the previous configuration. The Samples column gives the number of training instances that were used to train the given RBM, selected at random from the total training set; these training set sizes were chosen such that training all the RBM-n configurations matched the time complexity of training a Single RBM (i.e., both methods were given the same total number of weight updates).

For the original MNIST dataset, the Partitioned-RBM outperforms the Single RBM not only for the RBM-1, but in all configurations except RBM-28. This means that a single partitioned training step is already sufficient to produce good performance, even though there are still many feature pairs that are unconnected. Additionally, when a Single RBM is trained using the same reduced-size dataset as the final level of the Partition-RBM, its performance decreases even further. By design, the computational complexity of the full stack of Partitioned-RBMs is comparable to that of the Single RBM trained on the entire dataset; however, it is evident that significantly less computation was necessary for the Partitioned-RBM to yield superior performance. This behavior holds for classification as well as reconstruction error. As can be seen in Table 6.2, when using Class RBMs the partition-trained RBM produces higher classification accuracy while simultaneously being more computationally efficient.

For the Partitioned-RBM, reconstruction error on the permuted dataset is significantly worse ($p > 99.99\%$ using a paired t-test) than on the original dataset.

Configuration	Samples (10^3)	Accuracy	Chain Operations (10^{10})
Single RBM	54	96.97%	131.71
Partitioned-RBM-(16-4-1)	54-50-30	97.18 %	78.42

Table 6.2: *Classification accuracy of traditional and partition-trained RBMs on the MNIST dataset*

Permutation has no statistically significant impact on the performance of the standard Single RBM, as we would expect. From this result, we are led to the conclusion that the Partitioned-RBM is making use of statistical information that is spatially localized, where the Single RBM is not. The permutation results in no overall loss of information for the Single RBM; it simply re-orders the elements of the vectors. For this reason, any pair-wise correlation between a given pair of features will be unaltered by the permutation. The unchanged behavior of the Single RBM is therefore exactly what we would expect.

Things are different for the Partitioned-RBM, however, since the location of the two features in a given pair will be altered. This means that two features that would have been assigned to the same partition in the original dataset might *not* be assigned to the same partition in the permuted dataset. Since each piece of a Partitioned-RBM only has access to features in its partition, this means that whatever statistical information was contained in the correlation between this pair of features is no longer available to the Partitioned-RBM.

Configuration	Samples (10^3)	Error
Single RBM	54	2.59%
Partitioned-RBM-(16-4-1)	54-50-30	1.05%

Table 6.3: *Reconstruction error of traditional and partition-trained 2-layer DBNs on the MNIST dataset*

In fact, the Partitioned-RBM will always be cut off from a great many of the pairwise feature correlations; the difference between the original and permuted datasets is simply in *which* correlations are lost. The fact that the Partitioned-RBM performs significantly worse on the permuted dataset implies that not all correlations are of equal value. In particular, it means that correlations between pairs of features that are spatially proximate in the original data are more important to the success of the algorithm than correlations between arbitrary pairs (which will, on average, be significantly farther apart in the original image). Given the spatial structure in the data (see Section 4.4.2), these results demonstrate that the Partitioned-RBM is making use of spatially local statistical information in the MNIST dataset to achieve its performance advantage.

6.5.2 Partition Trained DBN results

Table 6.3 shows the results of training a two-layer DBN. The configuration column specifies training method used for each layer; "Single RBM" indicates the traditional RBM training method and "Partitioned-RBM" indicates our partitioned RBM training method. The number of partitions in each training stage is defined in parentheses; (16-4-1) indicates that we trained each Partitioned-RBM first with 16 splits, then 4, and finally trained as a single partition. Note again that each successive Partitioned-RBM configuration starts with the output of the previous configuration, as described previously. The Samples column gives the number of training instances, selected at random from the total training set, that were used to train the given DBN. Again, as the number of partitions decreases, we decrease the training set size to match the time complexity of the full Partitioned-RBM training process to that of the Single RBM.

As with individual RBMs, a DBN trained using Partitioned-RBM significantly outperforms the one trained using the traditional method ($p > 99.99\%$ using a paired t-test). By design, the computational complexity of the full stack of Partitioned-RBMs is comparable to or faster than that of the Single RBM trained on the entire dataset; however, it is again evident that significantly less computation is necessary for the Partitioned-RBM to yield superior performance.

We also generated variogram and mean-correlation plots (as described in Section 4.3) for several digits at multiple levels of the DBN (plots for the full MNIST dataset are in Section 4.4.2). Figure 6.2 shows the variogram plots for subsets of the data corresponding to digits 0, 5 and 9 (other digits are omitted for brevity, but look similar). The first column shows variograms of the raw input vectors for each subset,

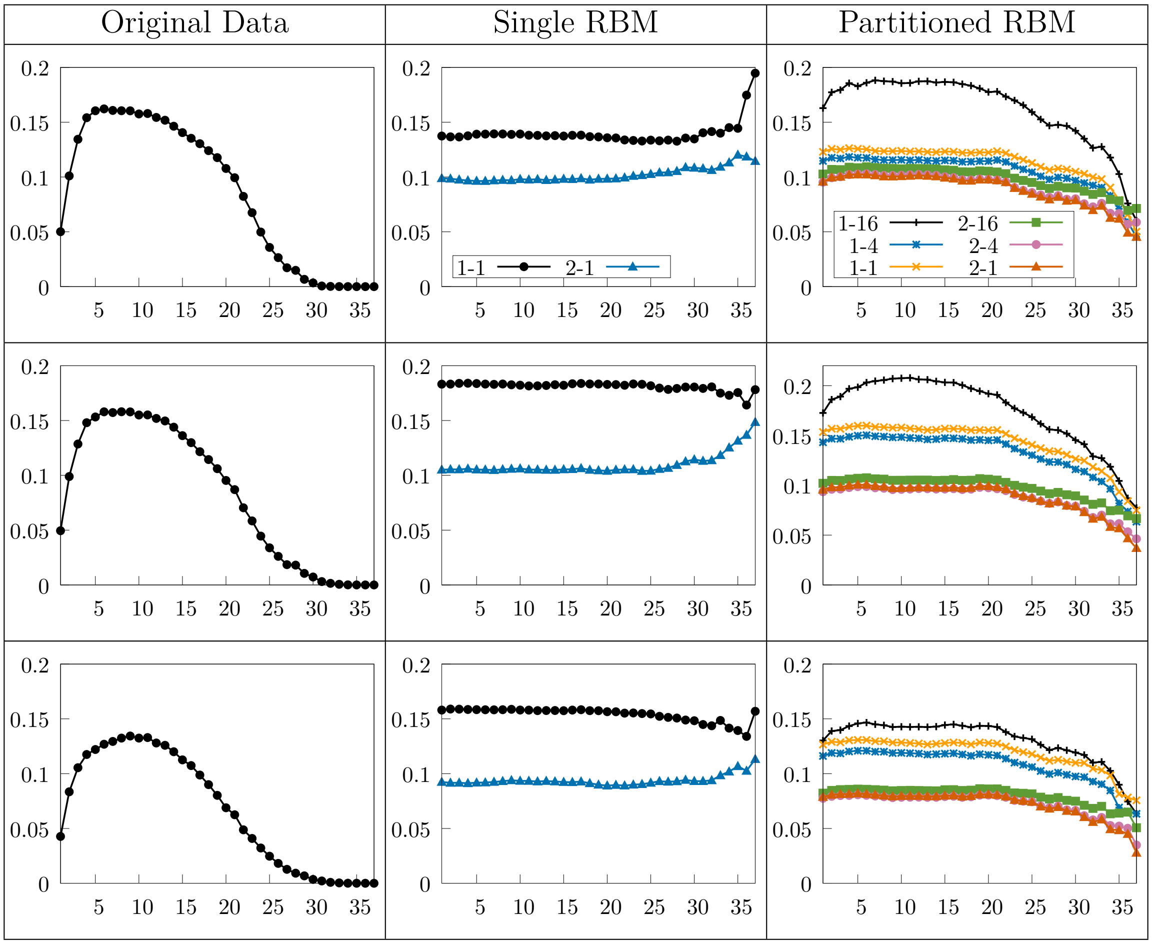


Figure 6.2: *Variograms: labels 0, 5 and 9 (from top to bottom). Labels of the form $N-P$ indicate data for hidden layer N of a Deep Belief Network based on Partitioned-RBMs with P partitions. First column corresponds to result of original digits. Second column corresponds to Single RBM and last column corresponds to Partitioned-RBM*

the second column shows results of the traditionally trained DBN, and the third shows results of the partition trained DBN; each graph has lines for both the first and second layers of the DBN. For all digits, the partitioned training method produces an “arch” pattern consistent with the original digit plot. In comparison, the hidden layers of

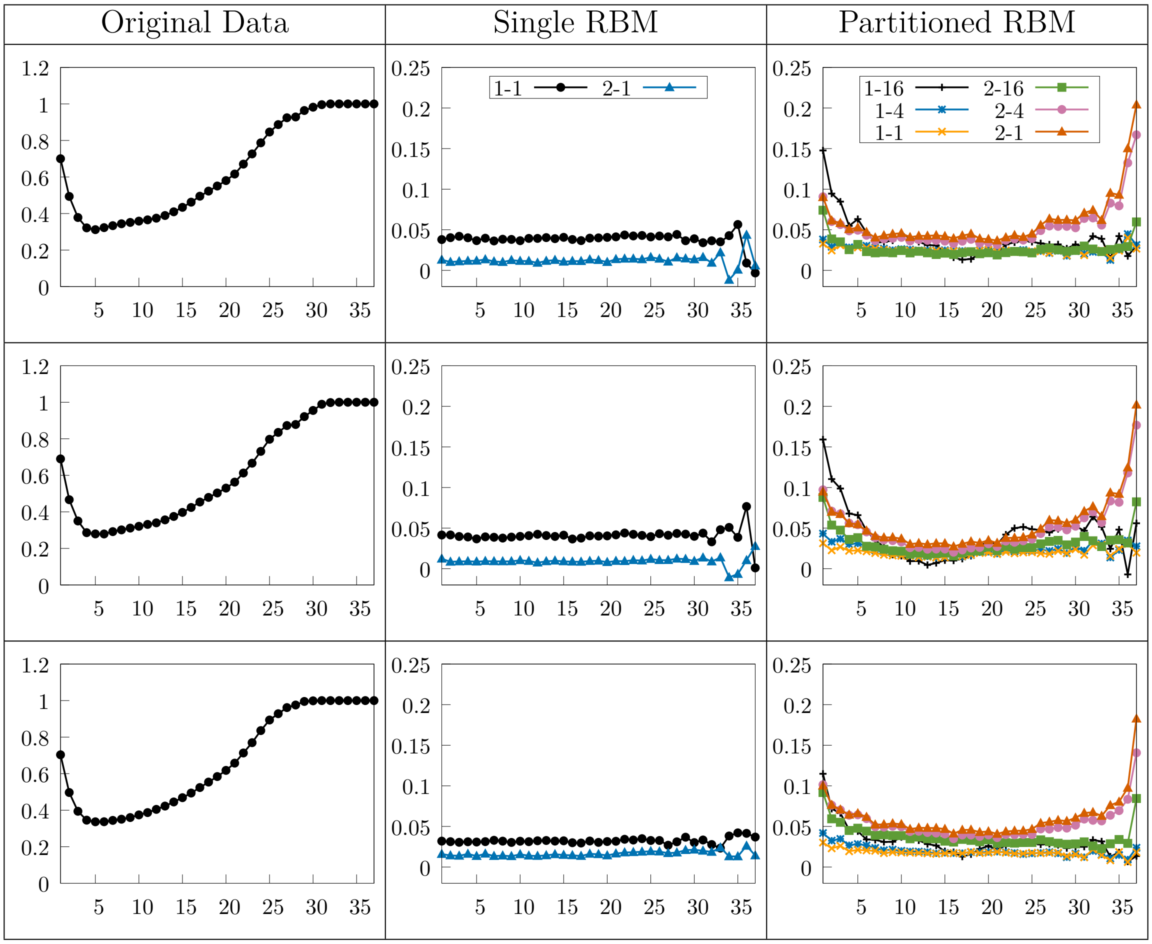
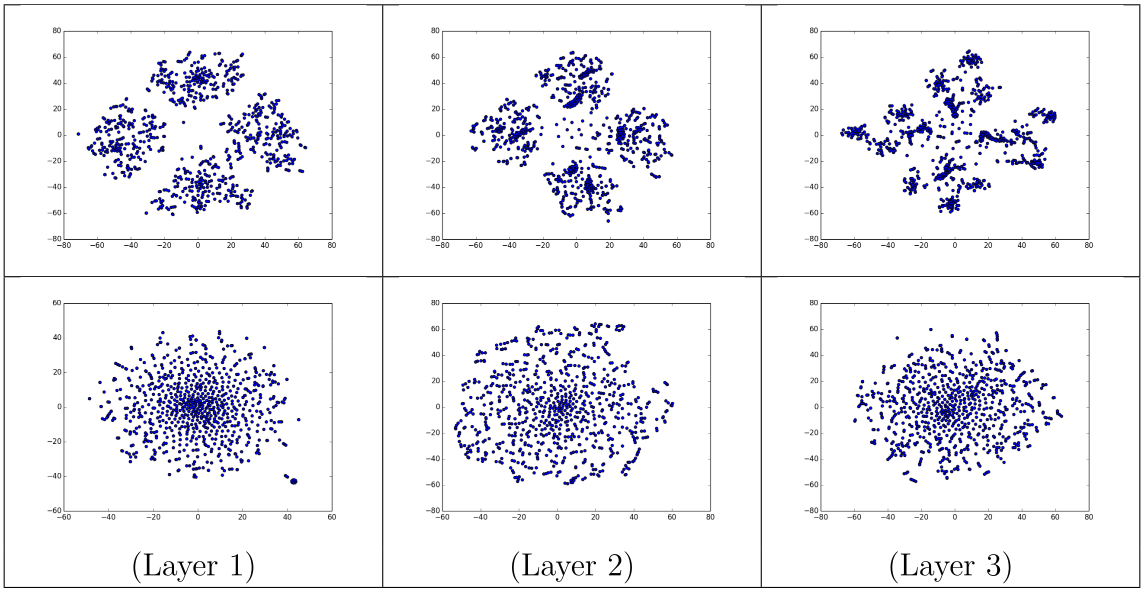


Figure 6.3: *Mean-correlation: labels 0, 5 and 9 (from top to bottom). Labels of the form N−P indicate data for hidden layer N of a Deep Belief Network based on Partitioned-RBMs with P partitions. First column corresponds to result of original digits. Second column corresponds to Single RBM and last column corresponds to Partitioned-RBM.*

the traditionally trained DBN do not preserve any relationship between distance and difference.

Figure 6.3 shows the mean-correlation plots for digits 0, 5 and 9 of the MNIST data. As with variograms, the first column depicts digits, the second column shows

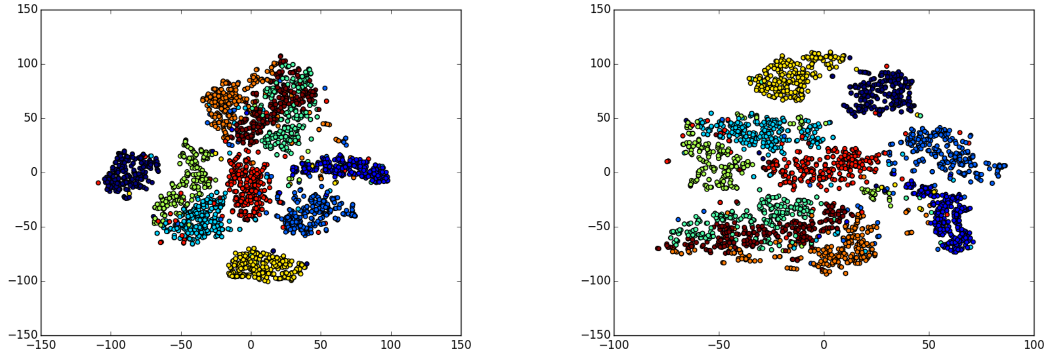


(Layer 1) (Layer 2) (Layer 3)

Figure 6.4: *t-SNE plots for DBN hidden node activations on the MNIST dataset. The first row shows results for the partition trained DBN, the second row shows results for the traditionally trained DBN.*

results of the traditionally trained DBN, and the third column shows results of the partition trained DBN. These are consistent with the variogram results: The Single RBM training method destroys any relationship between distance and mean correlation, while the raw data and the Partitioned-RBM output both show correlation that changes with distance, indicating that spatial information has been preserved.

We also used t-SNE (as described in Section 4.3) to visualize the activations at the hidden nodes. To apply t-SNE to hidden nodes, we generated sample points by setting a selected hidden node to 1.0 and all other hidden nodes to 0.0, and then computing corresponding input node activations. Thus, the weights between that hidden node and all visible nodes captures a “feature” (this can also be referred to as a filter, or



(A)

(B)

Figure 6.5: *t-SNE plots for both original (A) and permuted (B) versions of the MNIST dataset; color represents class label.*

template, depending on context). For this experiment, we constructed a 3-layer Deep Belief Network where each layer has 784 nodes. Results are shown in Figure 6.4. The first row shows the results for the partitioned training method and the second row for the traditional training method. Columns correspond to network layers 1–3. The scatter plot of activations shows that the partitioned training method produces some natural clusters, whereas the traditional training method output closely approximates a zero-mean Gaussian.

To ensure that the t-SNE method is correctly accounting for possible re-ordering of features, we applied t-SNE to both original and permuted versions of the MNIST dataset. Figure 6.5 shows results of these experiments. The left figure is for original data and right figure corresponds to permuted data. The permuted data generates a t-SNE plot with qualitatively similar structure to that generated from the original

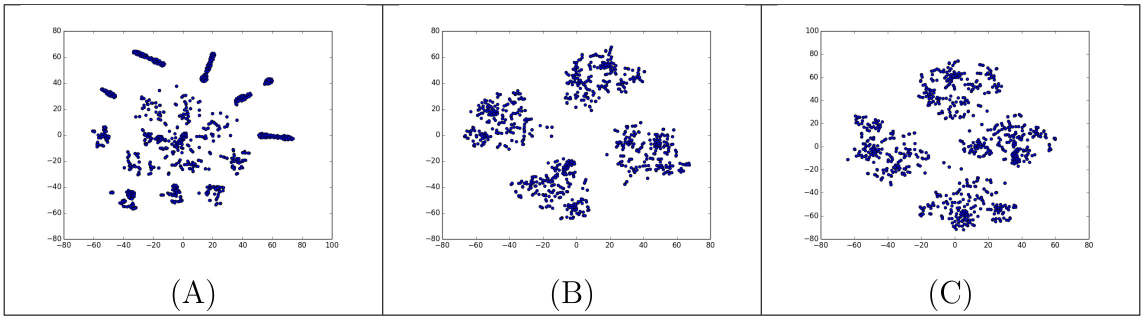


Figure 6.6: *t-SNE mappings for hidden node activations of different Partitioned-RBM configurations on the MNIST dataset; (A) 16 partitions, (B) 4 partitions, (C) 1 partition*

data; importantly, this resembles the output generated from the Partitioned-RBM, but it does not resemble the Gaussian-like output generated from the traditional RBM.

To explore how diffusion progresses across the configurations of the Partitioned-RBM, we paused the training between stages (i.e., just before the number of partitions was decreased). As the number of partitions changed, we plotted the t-SNE mapping for the first hidden layer of the DBN (first RBM). Figure 6.6 shows that structure continues to be present, though some consolidation does take place as partitions are joined.

All of these results support our original hypothesis and offer a clear demonstration that partitioning can be used to introduce a spatial bias into a technique that would otherwise lack one.

6.6 Conclusions

We have begun to explore one of the potential inductive biases that deep learning techniques leverage to achieve their performance. Our experimental results suggest that data partitioning techniques can enable algorithms to make use of spatially local information. This can be the case even when the final learned model has the same architecture as the unmodified version. Further, when the models are stacked, the hidden node activations retain spatial information, which is not the case in traditionally trained DBNs. Finally, the disruption of spatial structure hurts the performance of the spatially biased method, confirming our expectations that the spatial partitioning is only helpful when spatial structure is present.

Chapter 7

Deep Partitioned PCA

7.1 Motivation

One of our first attempts to introduce a spatially local structure bias into a non-connectionist model is a variant on PCA (see 2.3.2.1 for details on PCA as a base technique). We will refer to this system as *Deep Partitioned PCA* (DPPCA). The goal in designing the system was to perform feature extraction in a way that gained some of the structural properties of CNNs, but with as few other advantages as possible. PCA was chosen as a base technique in large part because it is a linear technique that is simple and well understood.

The intent of choosing a linear model was to be sure that any performance difference between "deep" and "flat" versions of PCA did not come from the extra mathematical power inherent in stacked non-linearities. Both standard PCA and DPPCA have the same mathematical power (since composed linear functions have no more power than a single linear function). Thus, so long as the total number of output features generated by each model is the same, there should be no representational advantage to one technique over the other.

PCA is also a deterministic technique, which produces a result that is guaranteed to optimize a known criterion. In particular, if the top k principal components are used as a basis to generate a length-k feature vector, there can be no other linear projection into $\mathbb{R}^k$ that yields a lower average reconstruction error [2]. Additionally, PCA is an algorithm with no parameters or hyper-parameters to tune (other than the choice of k), which further reduces the possible impact of experimental design on the results.

Finally, since we wanted to look at image data to make our work comparable to prior work on CNNs, PCA was a reasonable choice. It has been used as a dimensionality reduction technique for many years in the field of computer vision (see, for example, [137]), which means that readers are likely to be familiar with it.

7.2 DPPCA

DPPCA is a feature extraction hierarchy based on a partitioning of the original feature space that resembles the partitioning used by CNNs. In our experiments on image data, we used a quad-tree [34] style partitioning; the bottom level of the partitioning was a set of non-overlapping 4×4 pixel patches. PCA was applied to the set of all 4×4 patches from all images in the training dataset and the top k eigenvectors were used as a reduced-dimensionality basis. This can be thought of as comparable to a CNN architecture in which the stride length of the filters is equal to their width; by using the union of the sets of 4×4 patches, we make the same assumptions that "weight sharing" in CNNs does. Unlike a standard CNN, however, we chose to use disjoint partitions rather than overlapping ones, so as to avoid giving the deep method any extra advantages.

Each 4×4 patch was projected into this new basis, and the reduced-dimensionality patches were then joined back together in their original order. The result of this was that each original data vector (i.e., image) had its dimensionality multiplied by $^k/_{16}$ (Figure 7.1). These reduced dimensionality images formed the next layer up in the hierarchy after the raw data.

This process was repeated, using the newly created layer as the data for the split-PCA-join process to create the next layer up. At every layer after the first, the dimensionality was reduced by a fixed factor of 4 (i.e. a factor of 2 in each dimension).

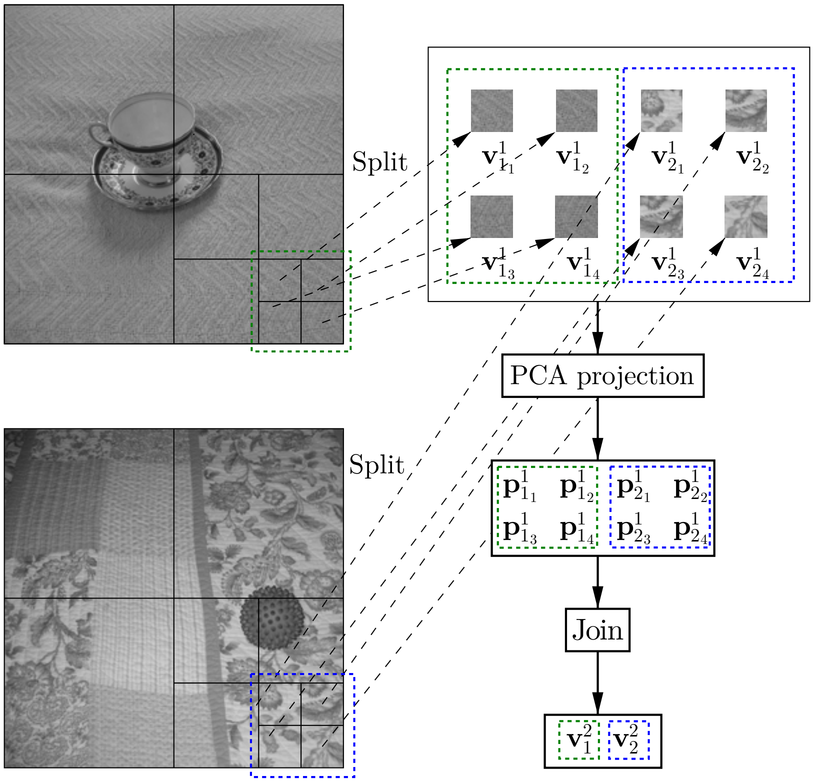


Figure 7.1: *An example of how DPPCA works on a pair of images from our dataset. The $\mathbf{v}^1$ are vectors in S^1, the $\mathbf{v}^2$ are vectors in S^2, and the $\mathbf{p}^1$ are vectors in P^1. Note only a small number of the vectors from each of these levels are shown.*

The process was terminated when the remaining data was too small to split, and the entire image was represented by a single k-dimensional feature vector at the "top" of the hierarchy.

As an illustration, if our original data is a set of m vectors, each length n, then we start with a raw data matrix $\mathbf{D_1}$, which is $m \times n$. In the case of images, this means each row of the matrix is an image. The level one split data matrix, $\mathbf{S_1}$, in our hierarchy is generated by recursively splitting the vectors in $\mathbf{D_1}$ down to $4 \times 4 = 16$ dimensional patches, so it will be a $(m \cdot \frac{n}{16}) \times 16$ matrix. We apply PCA to $\mathbf{S_1}$, extract the top k eigenvectors into $\mathbf{F_1}$, and use $\mathbf{F_1}$ as a basis to project the vectors of $\mathbf{S_1}$ into. Applying the projection to the vectors in $\mathbf{S_1}$ results in $\mathbf{P_1}$, an $(m \cdot \frac{n}{16}) \times k$ matrix. Adjacent vectors in $\mathbf{P_1}$ are then joined (using the inverse of the splitting operator), resulting in $\mathbf{S_2}$, which is $(m \cdot \frac{n}{16 \cdot 4}) \times (4 \cdot k)$. If we continue to recursively join the $\mathbf{S^2}$ data, we will get $\mathbf{D_2}$, which is an $m \times (n \cdot \frac{k}{16})$ matrix. Alternatively, we can simply apply PCA directly to $\mathbf{S_2}$ and avoid some extra split/join operations when building the hierarchy.

In general, for layer l, $\mathbf{D}_l$ will be $m \times (n \cdot \frac{k}{16 \cdot 4^l})$. When $16 \cdot 4^l = n$, the hierarchy is complete, giving a top layer with dimensions of $m \times k$. We note that the math only works cleanly for raw data vectors whose length is a power of four; this means we need square images with power of two widths. While there are several ways this constraint could be relaxed (by changing the split/join operation), we leave them for future work.

Pseudocode for this algorithm is given in Algorithm 7.4. The *SplitQuads* function (line 3) does a quad-tree style split of an image into four equal-sized sub-images, and the *JoinQuads* function (line 8) inverts this operation. The function $PCA(M, k)$ (line 6) computes an eigen-decomposition of the covariance of M and then returns the top k eigenvectors. The matrix multiplication in line 7 projects the data into the new eigen-basis. The **for** loop in lines 2–4 does the "recursive" splitting of data, and the **for** loop on lines 5–9 builds the feature hierarchy one level at a time.

Algorithm 7.4 DPPCA

```
Input: input matrix $\mathbf{X}$, depth of hierarchy $m$, number of eigenvectors to keep $k$
Output: feature-space hierarchy $F$, projected data hierarchy $\mathbf{D}$
$\mathbf{D}_1 \leftarrow \mathbf{X}$
for $i = 1$ to $m$ do
    $\mathbf{D}_1 \leftarrow \text{SplitQuads}(\mathbf{D}_1)$
end for
for $i = 1$ to $m$ do
    $\mathbf{F}_i \leftarrow \text{PCA}(\mathbf{D}_i,\ k)$
    $\mathbf{P} \leftarrow \mathbf{F}_i\mathbf{D}_i$
    $\mathbf{D}_{i+1} \leftarrow \text{JoinQuads}(\mathbf{P})$
end for
return $F$, $D$
```

Since our goal was to examine the benefits of performing "deep" dimensionality reduction in this hierarchical fashion, we used only the top layer as the "output" of the overall DPPCA technique. This allowed us to compare the k-dimensional feature vector produced by DPPCA directly to the k-dimensional feature vector produced simply by applying PCA directly to the raw image data and projecting into the top-k eigen-basis (which we will call "Flat PCA" for the sake of clarity). We will leave the exploration of using lower layers of the hierarchy as outputs for future work.

The DPPCA model has less theoretical power than most previous deep learning models, due to the fact that it is a linear model. This is in contrast to the complex non-linear functions used in other deep models, which often include things like inter-layer adaptive contrast normalization, soft-max functions, and sigmoid or rectified-linear input aggregation. Additionally, some deep methods do not restrict themselves to feed-forward operations [6], making the overall behavior of the system that much more complicated.

The inherent power of the repeated non-linear aggregation used in standard deep learning techniques makes it difficult to tell how much of the performance of those techniques is due to the spatially organized feature partitioning, and how much is due to the stacked non-linear functions. It was for this reason that we designed DP-PCA to have as little non-linearity as possible. While this handicaps its performance in comparison to something like a CNN, it allows us to examine the effects of the hierarchical partitioning much more cleanly than would be possible using non-linear aggregation (or using a non-linear feature extractor at each layer instead of PCA, which would also introduce significant non-linearity into the result).

We also chose to use a non-overlapping partitioning; using overlapping partitions when generating the PCA-based mapping would likely improve results, but can be viewed as giving DPPCA an augmented dataset (i.e., more data overall than flat PCA would have access too). Again, this choice was made to ensure DPPCA had no possible source of advantage other than the hierarchical partitioning.

7.3 Methodology

One of the most basic deep-structure hypotheses is that real-world data contains deep structure, and exploiting this structure will yield improved performance on machine learning tasks. To test this hypothesis in a non-connectionist setting, we designed experiments to compare the performance of a standard (shallow) feature extractor directly with a deep feature hierarchy using the same extractor. As described, our goal in choosing PCA was to have a well-understood feature extractor that could be used to expose the differences between deep and flat feature extraction; we have no expectation that it will produce optimal classification results. For our purposes here, the differences between deep and flat are more important than the absolute performances. In an application where performance is the primary goal, the best feature extractor available should be used.

We used the custom dataset described in Section 3.5, in three different image resolutions (128×128, 256×256, and 512×512 pixels). For each image resolution, we did experiments using both 5×2 cross-validation, and 10-fold cross-validation. Five-by-two has some nice theoretical properties, but due to the relatively small size of our dataset, the accuracy achievable by 10-fold cross-validation was higher. We report the results for both methods.

For each experiment, a dataset was split into “train” and “test” sets using one of the validation methods, and the training set was used to generate two feature spaces.

The first used Flat PCA to generate 16 features, and the second used DPPCA to generate 16 features. The dimensionality of the resultant feature space was the same for both techniques. Due to the length of our data vectors and the computational resources available to us at the time, operations on the full covariance matrix proved intractable. To work around this, we used an iterative PCA algorithm [4] to generate only the first 16 eigenvectors.

The training data was then projected into both 16-dimensional feature spaces, and the projected training data was used to train two standard classifiers. Once the classifiers were trained, the testing data was projected into each feature space and presented to the corresponding classifier to evaluate its performance.

We used two classifiers, a simple Nearest Neighbor classifier and a Support Vector Machine. As with the choice of feature extractor, we chose simple, widely-used, deterministic classification algorithms. While we performed a few experiments to make sure we had reasonable parameters for the SVM (i.e., kernel type, degree, etc.), we make no claim that these classifiers will yield the highest possible performance on the task. Again, the goal was to use simple algorithms to make the difference between the deep and flat feature extraction as clear as possible.

Finally, we performed the same experiments on a randomly permuted version of the dataset (see 4.4.1). This permutation effectively erases any local structure in the

Width	Validation	Classifier	Flat	Deep
128	10-fold	KNN	52.56%	53.20%
128	10-fold	SVM	45.40%	**48.09**%
128	5x2	KNN	43.84%	**44.93**%
128	5x2	SVM	37.77%	**39.63**%
256	10-fold	KNN	51.26%	52.08%
256	10-fold	SVM	45.04%	46.71%
256	5x2	KNN	42.33%	**43.54**%
256	5x2	SVM	36.28%	37.83%
512	10-fold	KNN	50.83%	52.60%
512	10-fold	SVM	43.59%	46.57%
512	5x2	KNN	43.87%	**45.03**%
512	5x2	SVM	36.61%	**38.47**%

Table 7.1: *Classification accuracy on the MNIST dataset. Each score is averaged over the samples created by the indicated validation technique. Bold numbers indicate that the advantage a technique showed was significant ($p \geq .95$ using two-sided paired Wilcoxon).*

data, while preserving global statistical properties. This was done to test whether the deep architecture was truly making use of local structure or not.

7.4 Results and Discussion

We ran both 10-fold and 5×2 cross-validation in combination with each image size and classifier. The results of these experiments are summarized in Table 7.1 by giving the mean accuracy achieved by each group of 10 experiments (one per validation fold). As can be seen in this table, Deep Partitioned PCA achieves a higher mean accuracy than Flat PCA in all cases; the overall mean improvement achieved by DPPCA is 1.16%. While this difference is small, it is highly significant; a two-sided paired

Width	Validation	Classifier	Deep:Flat Wins	Margin
128	10-fold	KNN	50%:20%	2.94%:4.08%
128	10-fold	SVM	90%:10%	3.14%:1.66%
128	5x2	KNN	60%:20%	1.93%:0.33%
128	5x2	SVM	80%:10%	2.45%:1.00%
256	10-fold	KNN	50%:30%	3.24%:2.70%
256	10-fold	SVM	60%:40%	6.03%:4.93%
256	5x2	KNN	70%:20%	1.87%:0.49%
256	5x2	SVM	70%:30%	2.82%:1.43%
512	10-fold	KNN	50%:30%	4.90%:2.18%
512	10-fold	SVM	50%:30%	8.17%:3.81%
512	5x2	KNN	80%:20%	1.57%:0.50%
512	5x2	SVM	80%:20%	2.53%:0.83%
Average			65.83%:23.33%	3.47%:2.00%

Table 7.2: *Percentage of validation runs in which one technique out-performed the other. In cases where performance was the same, no winner is listed. The margin is the amount that the winning technique won by, averaged over the instances in which that technique won.*

Width	Flat MSE	Deep MSE	Flat Accuracy	Deep Accuracy
128	6.53	6.96	44.89%	46.47%
256	14.08	13.90	43.73%	45.04%
512	29.92	29.51	43.73%	45.67%

Table 7.3: *Mean squared reconstruction error and average classification accuracy for the different techniques and image sizes. The results are averaged over all experiments on images with the indicated resolution.*

Wilcoxon test yields a p-value of $p = 1.86 \times 10^{-7}$ for the null hypothesis that the two methods produce equivalent results.

The absolute values of the accuracies are low in all cases, though well above random chance for a 10 class problem. This seems to be due largely to the difficulty of the problem and the small number of samples; as a baseline, we performed experiments using simple CNNs, and were unable to obtain accuracy above 48% (Table 7.5). It is

Width	Flat Original	Flat Permuted	Deep Original	Deep Permuted
128	43.84%	44.07%	44.93%	35.89%
256	42.33%	41.02%	43.54%	31.11%
512	43.87%	43.25%	45.03%	28.59%

Table 7.4: *Classification error on randomly permuted images. Results reported using 5x2 validation and the nearest neighbor classifier; results for other methods were similar.*

Flat PCA	Deep PCA	CNN
43.84%	44.93%	48%

Table 7.5: *Classification error on original images. Results reported using 5x2 validation. Nearest neighbor classifier was used for both PCA based methods.*

possible that with more parameter tuning we could do better, but it seems likely that no technique will perform much better for this data. Our technique offers relatively competitive performance despite using a simple, non-connectionist architecture. Additionally, it is far more computationally efficient; the Convolutional Networks took several orders of magnitude longer to train (both methods were able to take advantage of GPU processing, so neither had an advantage in parallelism).

Additionally, while object recognition is known to be a hard problem, it is likely that we could achieve better results using something other than PCA to do feature extraction, since PCA tends to work best for vision problems after lots of pre-processing (e.g., see the original Eigenfaces work [137]). Most work with CNNs also does more preprocessing than we used here; in particular, local contrastive normalization would likely improve the performance of either technique. As previously stated, we wanted a simple, general algorithm for our feature extractor, with as little preprocessing as possible. Our goal was to examine the role of deep structure in learning, not create a state-of-the-art classifier system.

Looking at the p-values for each experiment individually (highlights in the last column of Table 7.1), we see that the 5×2 cross-validation gives much better significance. In fact, in all but one of the 5×2 experiments, DPPCA was better by a statistically significant margin ($p \geq 0.95$). In the one instance where it failed to meet this significance, it was only off by about 2% ($p = 0.93$). The 10-fold cross-validation runs, on the other hand, had poor significance results but higher absolute performance scores. This result should not be surprising given the small size of the dataset; the 10-fold method has more training data, so it can achieve better performance, but much less testing data, which handicaps its ability to produce a wide and consistent margin.

While DPPCA is better on average for every cross-validation run, it was not always better for every single training fold. Table 7.2 shows how often each algorithm beat the other during each 10-experiment validation run, as well as the margin of that victory. In cases where the two algorithms tied, neither was counted as winning. We note that DPPCA not only wins more frequently, but when it wins it does so by a larger margin.

Due the relatively small data sets, we saw a large variance between different test/train splits; there could be as much as a 15% difference in accuracy (the same for both techniques) between the different splits of a single 10-fold cross-validation run. This behavior suggests that the number of training samples was a limiting factor in the final performance of the classifiers. Additionally, the average accuracy achieved during the 10-fold cross-validation experiments was around 10% higher than that achieved during the 5×2 cross-validation experiments, which lends support to this hypothesis. The difference between the classification accuracy achieved by the "flat" and "deep" methods was rarely more than a few percent, but it also showed a very small variance, proving to be quite stable across all the different experiments. Thus, we expect that a larger data set would show improved accuracies for both flat and deep methods, but we do not expect the difference between flat and deep would be impacted greatly.

Table 7.3 shows the average mean-squared reconstruction error (MSE) that results from projecting into and then out of each feature space, along with the average accuracy for each method. These results do not show any significant difference in MSE between the flat and deep techniques. As expected, increasing the length of the raw data vectors while leaving the dimensionality of the generated feature space fixed leads to higher MSE. Higher classification accuracy without higher MSE suggests that the deep technique is doing a better job of keeping "meaningful" features.

Table 7.4 shows the results of randomly permuting the data before applying the learning process. The fact that permutation makes no significant difference for Flat PCA is exactly what we would expect; since PCA works by looking an global statistical properties, the order in which the features appear makes no difference in the projected data. In the case of the deep technique, however, there is a significant difference; performance on the permuted dataset is far worse than on the unmodified images.

7.5 Conclusions

The central result of this work is that spatial partitioning and deep architectures can yield improved results even without connectionist models, and that this performance still seems to be due to a bias assuming the presence of spatially local structure. While many authors have claimed that deep techniques can learn abstract features, it has never been demonstrated that this property holds even without complex connectionist models. We have demonstrated that this property is, at least in part, created directly by the structure of the deep spatially local partitioning, and not just by the interactions of non-linearities in a multi-layer connectionist network. Both our deep and flat techniques have the same length output, and the same total amount of input data, meaning that neither one has an information-theoretic advantage in its representational power. Additionally, removing the local structure from the data via random permutation results in a significant performance loss for the deep technique only, once again demonstrating that the spatial bias is having the impact we predicted.

These experiments provide further evidence that our original hypotheses hold. In particular, they continue to provide evidence that images have spatial structure that fits the local structure bias reasonably well, and they confirm that we can use partitioning to take advantage of this structure even in models which are non-connectionist and entirely linear.

Chapter 8

Spatial-biased Random Forests

8.1 Motivation

While the approaches discussed in the previous sections apply the ideas of spatially local structure to models beyond CNNs, they all still take the basic approach of operating on spatially partitioned sub-vectors in a hierarchical fashion. Our next step was to apply the same bias to a technique not related to CNNs in any way, to demonstrate that a spatial bias can still improve performance.

For this set of experiments, we chose Random Forests (RFs) (see Section 2.3.5 for a full description). In particular, we used the style of random-projection Random Forest described in [134] as Randomer Forests (RerFs). We will refer to our modified technique as *Spatial-biased projective random forests*, or SBPRFs.

For the sake of clarity and consistency, we will use Breiman's terminology [12] and call the basic, single-index random forest algorithm "Forest-RI" (i.e., Forest using Random Indexes). We will sometimes refer to the RandomerForest algorithm as "Forest-RP" (Forest using Random Projections), and our novel spatially-biased algorithm as "Forest-RS" (Forest using Random Structured projections) when doing so will make things easier to understand.

Like PCA, Random Forests are a well-known and widely studied technique with many nice theoretical properties. There are a wide variety of problems for which Random Forests outperform other learning algorithms [16, 33], but computer vision problems do not generally fall into this class. Therefore, we were interested to see how much performance benefit there was in applying a spatial bias to random forests for these tasks.

It is worth noting that classic RFs that select a single feature to use as a splitting criterion struggle with problems that require examination of multiple features simultaneously (such as XOR or parity problems); RerFs use a linear combination of a random subset of features at each node, which gives them much greater flexibility in this type of task.

Even with random projections, however, RFs do not have a good mechanism for handling certain types of invariance. By contrast, one of the strengths of CNNs is that the repeated pooling steps introduces a significant degree of spatial invariance, which is useful in some types of vision tasks. Tasks like handwritten digit classification or face recognition often have significant pre-processing done to ensure that spatial invariance is not required (i.e., each image is scaled and centered the same way), but more challenging tasks like general object recognition do not offer this type of preprocessing (CIFAR and ImageNet are examples of this type of problem). This means that RFs, even with a spatial bias, are not currently good candidates for these problems. Our goal in this work is to show that RFs can be made to take advantage of spatially local information; we view the introduction of spatial invariance to be an important direction for future work.

8.2 SBPRFs

Standard RerFs work by using random projections of the data to generate node-splitting candidates (see Section 2.3.5 for details). SBPRFs work according to the same general tree-creation algorithm but with a modification to the way in which the random projections are generated.

In a standard RerF, candidate projections are created by choosing features at random from an n dimensional feature vector, with each feature having a likelihood of being chosen of $^1/_k$ for some k much less than n. Each feature is then assigned a weight in the range $[-1, 1]$. These random feature-weight sets can then be used as basis vectors; projecting data vectors onto these basis vectors is equivalent to computing the weighted linear combination for that data vector, and produces a scalar output. The scalar projection value is then considered as a candidate for splitting the tree node, in the same way as for any other type of random forest.

In SBPRFs, the difference is that the features for each projection are chosen using a different distribution. In a standard RerF, features are chosen according to a uniform distribution. In a SBPRF, features are chosen according to a spatially-biased distribution.

For spatial data (such as images), each candidate projection has a "center" location sampled from a uniform distribution across each spatial dimension of the input (e.g., $\{X, Y\}$ for a grayscale image, $\{X, Y, C\}$ for a color image). This location is then used as the mean of a Gaussian distribution from which features are sampled. The variance of the Gaussian is a function of the tree-depth of the node; as depth increases, so does variance (the variance is also scaled based on the size of each dimension, in the case of not all spatial dimensions being equal in size). Note that samples drawn from the Gaussian are also subject to the bounds of the actual data space, so the sampling process on the whole does not represent a true Gaussian distribution but one which is bounded by the edges of the image.

See Algorithm 8.5 for pseudocode showing how to create a candidate projection. $\mathbf{v}$ will be determined by the data; for example, the CIFAR-10 dataset (see Section 3.3) contains 32 × 32 pixel images with 3 color values per pixel, so the result would be something like $\mathbf{v} = \{32, 32, 3\}$ and $n = 3$. The order of the values will depend on the way the values are laid out in memory (i.e. are the three color values for a given pixel adjacent, or are we storing all the red values, followed by all the green values and all the blue values).

The effect is that decision nodes near the root of a tree will tend to have projections based on features tightly clustered in a small spatial region. As tree depth increases, the features will tend to become more widely spread out, until eventually there is little spatial bias remaining, and the sampling distribution begins to resemble a uniform one again.

This sampling process tends to mean that each tree in the forest will, at its root node, effectively focus its "attention" on a particular spatial region. This will encourage the discovery of spatially local statistical relationships in the data. The gradual increase of the spatial region size is necessary to allow the trees to eventually capture statistical relationships that are non-local; if the sampling area is kept small at all tree nodes, tree performance is decreased.

With a sufficient number of trees in the forest, the overall forest will still spread its feature-samples across the data space in an approximately uniform way; the difference is that the sampling is now heteroscedastic thanks to the per-node spatial bias.

Algorithm 8.5 Pseudocode for an algorithm to generate a spatially biased random projection. Can be used to generate the rows of the matrix **A** used in Algorithm 2.2. $a \sim b$ indicates that a is sampled from b. Note that k may also be sampled from a Gaussian distribution; α and β are tunable parameters.

```
Input: Spatial stride vector $\mathbf{v}$ with $n$ elements, tree depth of the current node $d$,
    scaling parameters $\alpha$ and $\beta$, non-zero components to generate $k$
Output: Output projection $\mathbf{f}$
$\mathbf{f} \leftarrow \mathbf{0}$
for each non-zero component do
    for $i = 1$ to $n$ do
        $\mu \sim \text{Uniform}(1, v_i)$
        $\sigma \leftarrow (\alpha_i + (\beta_i \cdot d)) \cdot v_i$
        repeat
            $idx_i \sim \text{Gaussian}(\mu, \sigma)$
        until $1 \leq idx_i \leq v_i$
    end for
    $o \leftarrow idx_1$
    for $i = 2$ to $n$ do
        $o \leftarrow o \cdot v_{i-1}$
        $o \leftarrow o + idx_i$
    end for
    $f_o \sim \text{Uniform}(-1.0, 1.0)$
end for
return $\mathbf{f}$
```

8.3 Methodology

We chose two standard computer vision datasets to test the impact of spatially biasing the random projections, namely the MNIST and SVHN datasets (described in detail in 3.1 and 3.2 respectively). For each dataset, we used the provided test-train split, to allow easy comparison of our results to those obtained by others using different techniques. Our performance measure was classification accuracy on the testing set.

Experiments were performed using basic, single-index random forests (Forest-RI), standard RerFs (Forest-RP) and SBPRFs (Forest-RS); several meta-parameters were examined, and values were chosen experimentally (see Section 8.4 below).

In all cases, the learning algorithm is recursive, with each node of the tree taking a training set (consisting of the training examples assigned to that node by its parent, or the full set of examples assigned to the tree if it is the root node). A set of c candidate splitting projections are generated, using either a single randomly selected feature (Forest-RI), unbiased random projections (Forest-RP) or spatially biased random projections (Forest-RS); for the latter two techniques, each projection has (on average) k non-zero components. For each candidate, the data are sorted according to the projected value, and mid-points between each pair of adjacent values are considered as possible split points. The optimal split point is determined by computing a utility value for each possible split point; for our experiments, we used information gain as our utility estimate, though other measures of purity can be used. We tried Gini impurity, but found that information gain performed slightly better on our data.

The termination criteria for the recursive tree-building algorithms were: having a completely pure sample (i.e., all examples are the same class), having fewer than n_{min} total samples, or failure to find any split that improves utility by more than ϵ. In practice, we found there was no benefit from using non-zero values of ϵ for our data.

8.4 Results and Discussion

8.4.1 Parameter Tuning

We experimented with several parameters for our algorithms to find values that produced good results. As with our other experiments, this was not an exhaustive search, since we are more interested in the relative performance of the spatial and non-spatial techniques. For the parameters shared by multiple algorithms, all algorithms showed the same trends. This allowed us to use the same values for all versions, resulting in a fair comparison without disadvantaging any algorithm.

For all types of forests, each tree was trained using a randomly chosen subset of the available training examples; the size of this subset is a parameter of the algorithm. Experimentally, values that were too low resulted in underfitting regardless of forest size, and values that were too high resulted in performance that did not scale as well with increasing forest size. In effect, if each tree trains on the full set, there is less independence between the members of the ensemble, so adding more members has less benefit. For the results reported here, each tree was trained with 70% of the training examples. This is a similar number to what others have suggested is reasonable [12].

For all types of forests, increasing the number of candidates splits c to be considered at each node tended to improve performance up to some point of diminishing returns. For the experiments reported here, we used a value of 200.

For the MNIST dataset, the minimum number of examples needed before a node was split (n_{min}) was set to 1, meaning the algorithm ran until nodes were pure or no candidate split was able to reduce entropy. Higher values were found to hurt performance across all techniques. For the SVHN dataset, we used a value of 3 for computational reasons; with $n_{min} = 1$, the larger forests outgrew the 1Tb of main memory available on our systems.

For both Forest-RP and Forest-RS, the average number of non-zero projection components k was set to 3, as this seemed optimal for both techniques on our data. Higher values were computationally more expensive, and significantly higher values were found to negatively impact the generalization of both techniques, though the effect was stronger for Forest-RP than for Forest-RS. In fact, larger values for this parameter frequently resulted in both Forest-RP and Forest-RS performing *worse* than Forest-RI on novel testing data, though Forest-RS continued to out-perform Forest-RP.

For Forest-RS, there were additional parameters controlling how the variance of the Gaussian used to select features behaved. The results were not particularly sensitive to small changes, but we did try a range of values. For the experiments reported here, the variance was calculated as:

$$\sigma = (0.05 + (0.015 \cdot d)) \cdot w$$

	MNIST	SVHN
Forest-RI	0.967	0.701
Forest-RP	0.971	0.706
Forest-RS	0.974	0.722

Table 8.1: *Classification accuracy for random forest variants on MNIST and SVHN data. Forest-RI uses one feature to split nodes, Forest-RP uses a random weighted combination of features, and Forest-RS uses a spatially biased random weighted combination of features.*

where d is the tree depth of the current node, and w is the width of the current image dimension (in our experiments, the images used were square, so width and height were equivalent). This resulted in fairly tight clusters near the root of the tree, and widely spread ones near the leaves. Sample coordinates outside the image boundaries were re-sampled from the same distribution until valid coordinates were obtained.

8.4.2 Performance Analysis

Table 8.1 shows the final classification accuracy on testing data of each ensemble using the parameters described above. The MNIST results come from a forest of 1008 trees, and the SVHN results come from a forest of 384 trees.

In all cases, Forest-RS outperforms Forest-RP, which in turn outperforms Forest-RI. This ordering was extremely consistent across our tuning experiments so long as projective methods were not given too many non-zero components. For the null hypothesis that two techniques were equivalent, a paired Wilcoxon signed rank test gave a likelihood of $p \leq 2 \times 10^{-16}$ for all pairs using the data from Figure 8.1.

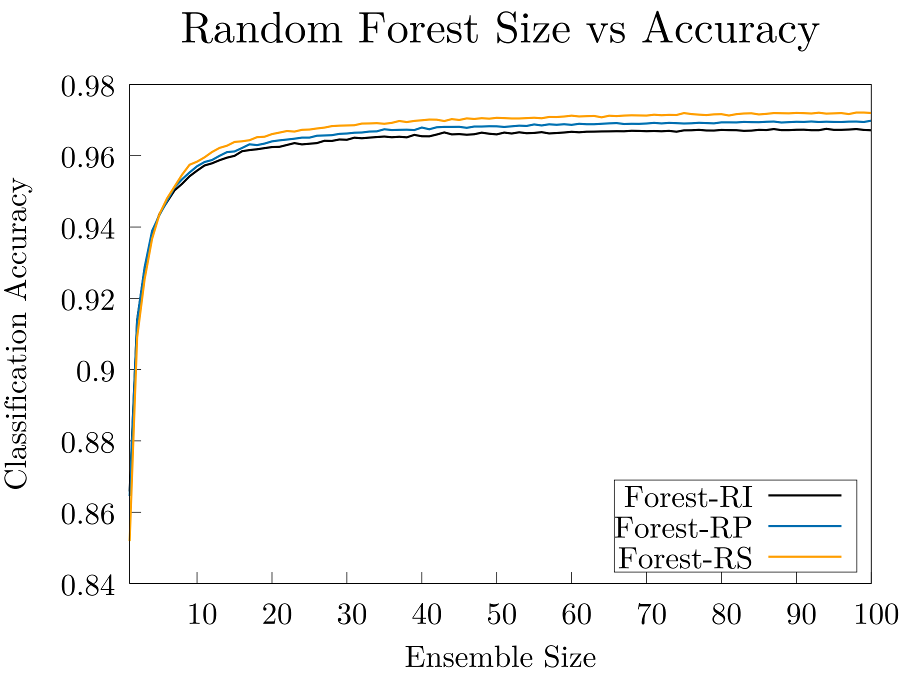


Figure 8.1: *Plot of classification accuracy vs number of trees in the forest for the MNIST dataset.*

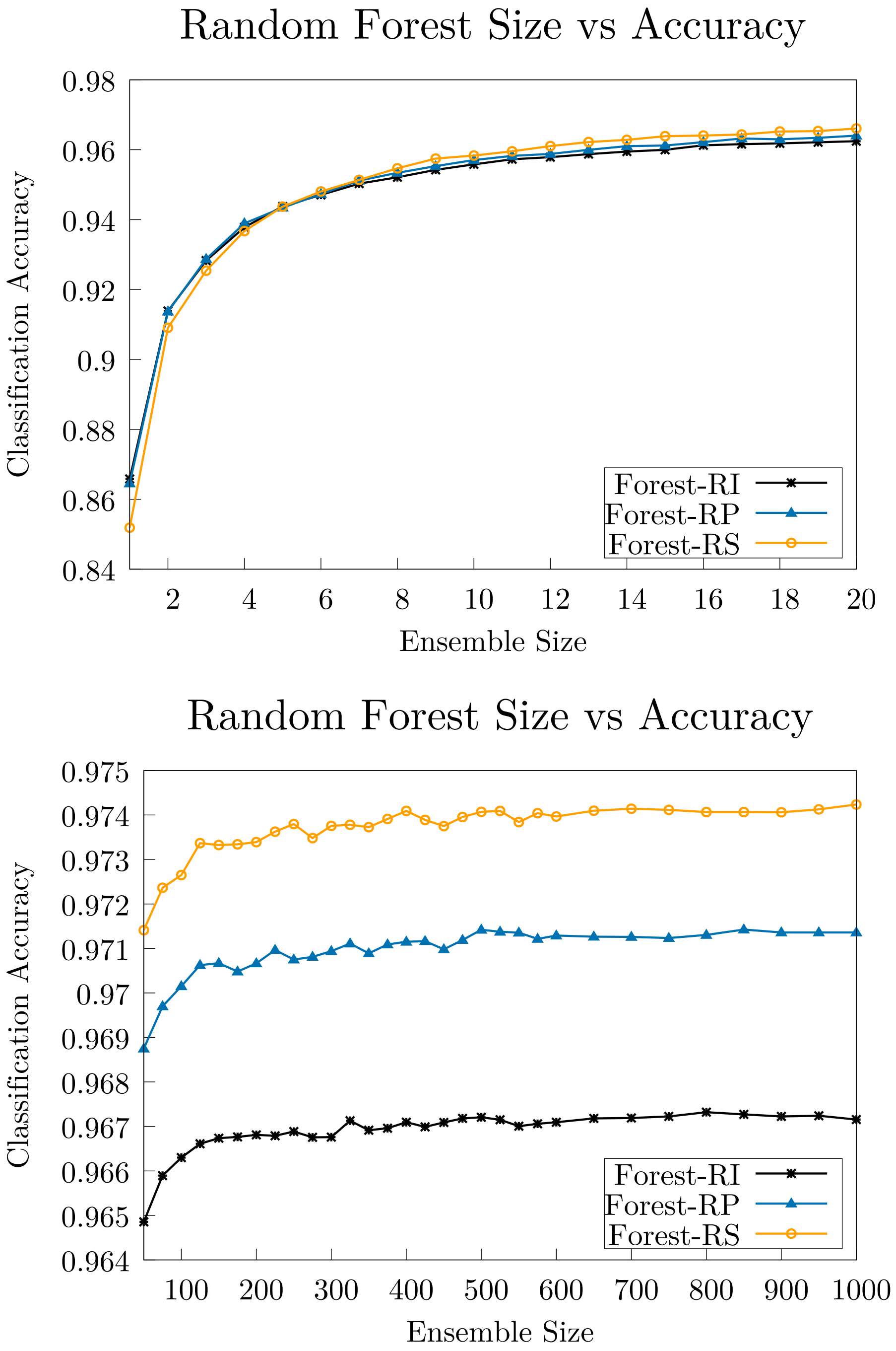


Figure 8.2: *Plots of classification accuracy vs number of trees in the forest for the MNIST dataset. The first plot shows behavior for small ensemble sizes, the second plot zooms out to show the behavior for large ensemble sizes.*

The performance of all techniques is better on MNIST than SVHN, which is not surprising given that SVHN is a harder problem. None of these techniques is competitive with CNNs on these datasets, but again this is unsurprising given that decision trees have no mechanism for handling several types of invariance that CNNs allow to some degree, including translation, scale, and local-contrast.

Figure 8.1 shows the performance of ensembles of varying sizes on the MNIST dataset. For each ensemble size value, 20 sample forests containing that many random trees were tested, and their performance was averaged. This was done for each size in the range of 1 to 100.

Figure 8.2 contains similar plots, but with different ranges to examine the behavior at different scales. We can see that for very small ensembles (less than 5 trees), Forest-RS does not perform as well as the other methods. For ensembles larger than 5 trees, the ordering of the techniques is once again consistently in line with the final results. This suggests that part of the performance advantage of the Forest-RS technique may be that the individual trees are slightly weaker but have greater independence, resulting in more power from combining them.

The second plot in Figure 8.2 shows the behavior for very large numbers of trees. The plot starts at an ensemble size of 50, so the range of accuracies represented is much narrower than in the other plots. This plot shows that performance improvement mostly levels off after the first 200 or so trees, increasing only very slowly after that.

After about 500 trees, the remaining fluctuations appear to be mostly noise, though it is possible that on average some very slow improvement will continue to occur.

This is in line with other work on random forests, which often suggests that there is no upper bound to how many trees can productively be combined. In particular, the claim is sometimes made that random forests are immune to overfitting, because increasing the complexity of the model (by adding more trees) will never cause the expected performance to decrease.

Our results support this idea to the extent that more trees will never reduce expected performance. However, in our experiments it also became clear that this claim relies on a very narrow definition of "overfitting." In particular, increasing the power of the individual trees in the ensemble *can* result in lower overall performance, regardless of how many trees are in the ensemble; we saw this when using values of k greater than 3 for both the Forest-RP and Forest-RS models.

Additionally, the test-set performance of our forests was often considerably lower than their train-set performance; for the MNIST dataset, the forest classified the training data with zero error, and for the SVHN dataset, the forest classified the training data with more than 98% accuracy, as compared to the testing data accuracies reported in Table 8.1. This indicates that the model on the whole is fitted better to the training data than the testing data, which is an alternative (and broader) definition of "overfitting."

8.5 Conclusions

Overall, these results demonstrate that it is possible to introduce a spatial bias into random forests, and to do so without a strongly defined spatial partitioning scheme. Even with a relatively weak form of spatial bias, incorporating such a bias improves the performance of random forests on spatial data. Additionally, the weaker bias means that unlike many of the previously presented methods, applying this method to randomly permuted data should result in it being mathematically equivalent to the Forest-RP algorithm; this means it will have lower performance than on the spatial data, but it should not fall below the performance of the baseline method. This suggests that this type of bias is more robust to the possible absence of spatial structure, which is a potential advantage over CNNs and other fixed-partition methods.

Chapter 9

Conclusions

9.1 Overview

This dissertation began with the hypothesis that Convolutional Neural Networks have an inductive bias which assumes the presence of spatially local structure. Through experimental and theoretical analysis, we demonstrated that image data does exhibit this type of structure, and that Convolutional Neural Networks rely heavily on its presence.

We devised a mathematical framework for describing this type of bias, which both encompassed existing methods and suggested ways of introducing local structure assumptions into other methods. We then applied this insight to a variety of different techniques. We used a spatially partitioned training scheme to improve the performance and convergence speed of Restricted Boltzmann Machines, and showed that Deep Belief Networks trained this way retain spatial structure at all levels of depth, while traditionally trained ones do not.

We also created a Deep Partitioned Principal Component Analysis algorithm, which demonstrated that similar behavior can be produced by a non-connectionist model, even when the model is completely linear. Finally, we introduced a stochastic spatial bias into random forests, and demonstrated that we can make use of a spatial bias even without a fixed spatial partitioning the inputs.

9.2 Main Contributions

9.2.1 Local Structure

In Chapter 4, we defined the concept of *local structure* as a statistical property of subspaces in a dataset. We provided both intuitive conceptual descriptions and formal statistical ones for how local structure can be defined and what it looks like.

We then described the special case of *spatially local structure*, in which the features of a data vector have some natural spatial organization to them. In this case, we can describe statistical information local to particular spatial regions of our input vectors. This formulation allows us to apply spatial statistics techniques to our data, as well as offering a framework for describing the architecture and behavior of techniques like Convolutional Neural Networks.

We also defined a formalism for feature-space partitioning and described a number of different types of partitioning functions that can be used for different types of problems.

Finally, we described some statistical tools that can be used to analyze spatially local structure in data. We then applied these tools to our data (described in Chapter 3), and examined the results. Finally, we described a way to destroy spatial structure by applying a random permutation to the feature ordering of data vectors, and showed that our analytic tools demonstrated that these “permuted” datasets no longer contained spatially local structure.

9.2.2 Spatial Bias in Convolutional Neural Networks

In Chapter 5, we used both the original and permuted versions of several image datasets to train basic multi-layer-perceptron style networks, showing that both versions of the data produced the same results. This demonstrated that the random permutation had not impacted the overall learnability of the data. When we trained Convolutional Neural Networks on the same data, however, we demonstrated that while they were able to significantly outperform the fully connected networks on the original data, they were unable to learn effectively from the permuted data. In fact, since the final layers of a Convolutional Neural Network form a fully connected network, the use of convolutional layers appears to be worse than the identity function, actively hurting the ability of the top layers to learn on the permuted data.

This confirms that Convolutional Neural Networks rely on the existence of spatially local structure, while fully-connected networks do not. It also helps confirm that the analytical tools presented in the previous chapters are measuring the same kind of local structure that Convolutional Neural Networks take advantage of.

9.2.3 Partitioned Training of RBMs and DBNs

In Chapter 6, we introduced a method for training Restricted Boltzmann Machines based on a spatial partitioning of the input data, and showed that this training method resulted in faster convergence and lower reconstruction error rates in comparison to the standard training method. We also showed that, much like Convolutional Neural Networks, this performance advantage only applied to data with spatial structure.

We then used this partitioned training method to train successive layers of a Deep Belief Network, and showed that such a network again outperforms a naïvely trained one on spatial data. We also showed that the hidden node activations of each layer in a partition-trained network retained spatial patterns, while the Deep Belief Network trained using the traditional algorithm did not display local structure in its hidden nodes.

9.2.4 Deep Partitioned PCA

In Chapter 7, we introduced a hierarchical, spatially partitioned variation on Principal Component Analysis. As with the other spatial models, we showed that this technique produced better features for classification than traditional, non-spatial Principal Component Analysis. Also as with the other spatial models, we showed that the spatial algorithm had significantly lower performance on randomly permuted data, while the non-spatial algorithm did not.

What makes this result interesting is that the technique presented is entirely linear, and is not a connectionist model. This demonstrates that the ability to make use of spatially local information is not simply an emergent feature of complex connectionist architectures and composed non-linear functions.

9.2.5 Spatial-biased Random Forests

In Chapter 8, we introduced a spatially biased variation on the Random Forest ensemble learning algorithm. This model differed from the others in that there was no fixed partitioning of data, but rather a stochastic partitioning generated on a per-node basis. Since standard random forests already make use of a stochastic per-node partitioning, the only change we made in our method was to the distribution used for generating the partitions. Where standard random forests use a uniform distribution, we used a Gaussian distribution with a randomly chosen mean.

We showed that spatially-biased random forests are able to out-perform otherwise equivalent unbiased random forests when applied to spatial data. Unlike the other techniques, this one uses a much weaker spatial bias; in fact, its performance on non-spatial data will be the same as the unbiased version, so it can be considered more robust against the absence of spatial information than the other spatial methods described.

9.3 Future Work

The work presented in this dissertation represents only the beginnings of an inquiry into the inductive biases that have allowed Convolutional Neural Networks to achieve such high performance on spatio-temporal learning tasks. As such, there are a number of interesting directions for further inquiry, including the introduction of spatial bias into more models, the application of local structure learning to non-spatio-temporal data, and the examination of additional biases present in Convolutional Neural Networks.

9.3.1 Adding Spatial Bias to Existing Techniques

We have demonstrated several ways of adding spatial bias to existing techniques, and shown in each case that they result in improved performance relative to non spatially biased versions of the same techniques. In many ways, however, the work performed so far is as much proof-of-concept as it is practical, since Convolutional Neural Networks still have superior overall performance for the standard computer vision tasks we examined.

We believe that in time the concept of feature-space partitioning may become a powerful tool in the broader arsenal of machine learning, such that selecting a partitioning scheme might be just one more choice made by an engineer, rather than being viewed as a stand-alone technique. In the shorter term, however, there are both many learning algorithms that might benefit from the introduction of spatial bias, and many ways in which such bias could be introduced.

9.3.2 Applications to Non-spatio-temporal Data

The problems examined in this dissertation have been restricted to standard computer vision benchmarking tasks performed mostly on standard datasets. This was done because we were specifically targeting the tasks on which Convolutional Neural Networks had demonstrated such success. Additionally, the use of standard datasets makes it easy for researchers familiar with these data understand our results and compare them to other work. However, we believe that the broader idea of partitioned learning need not be restricted to image tasks. A first step would be application to other, non-image spatio-temporal data.

The more interesting (and more difficult) task is to discover ways of generating useful partitioning schemes for data that is not inherently spatio-temporal in nature. This will likely require significant theoretical and statistical work, as well as practical experimental demonstrations. We believe that probabilistic graphical modeling and latent variable analysis may be a promising starting point for such an analysis. Work by Strasser *et al.* [128, 129] is one possible starting point.

If heuristics for partitioning arbitrary data could be created, it could enable performance improvement for the broader space of machine learning problems, including everything from multivariate discrete data to multi-view and heterogeneous data. It is additionally worth noting that partitioning can also provide ways of increasing parallelization. In the past decade, an increasing fraction of the Moore's Law increase in transistor counts has been dedicated to improving parallelism, rather than increasing clockspeed, so the ability to take advantage of this hardware is correspondingly becoming more important in machine learning.

Finally, we have shown that under some circumstances partitioning can significantly decrease time to convergence when it is applied to algorithms that scale poorly with things like clique size. For these techniques, simple increases in computational power may never be sufficient to make them practical at big-data scales; feature-space partitioning may be a way of helping these techniques scale more effectively.

9.3.3 Other Biases of Deep Learning

While our work presents strong evidence that Convolutional Neural Networks have an inductive bias that assumes spatially local structure is present, this is by no means the *only* inductive bias of the method. Given how successful the technique has been, we believe further investigation is required to tease apart all of the different biases that allow it to work so well.

For example, Convolutional Neural Networks frequently produce high-level features that show high degrees of invariance to things like position and scale; the work we presented in this dissertation does not have these properties. The ability to understand and model the way these invariances are introduced would again allow us to introduce similar invariances into other techniques.

Additionally, other deep learning models could be examined for inductive biases. For example, several deep recurrent network models have been demonstrated to have useful and interesting properties, including Long Short-Term Memories [47, 48, 60] and Neural Abstraction Pyramids [6, 7]. An examination what inductive biases enable these models to work would be of significant benefit to the field of machine learning as a whole.

Bibliography

[1] E. H. Adelson, C. H. Anderson, J. R. Bergen, P. J. Burt, and J. M. Ogden, “Pyramid Methods in Image Processing,” *RCA Engineer*, vol. 29-6, 1984.

[2] E. Alpaydin, *Introduction to Machine Learning.* MIT Press, 2004.

[3] Y. Amit and D. Geman, “Shape quantization and recognition with randomized trees,” *Neural Computation*, vol. 9, no. 7, pp. 1545–1588.

[4] M. Andrecut, “Parallel gpu implementation of iterative pca algorithms,” *Journal of Computational Biology*, vol. 16, no. 11, 2008.

[5] D. C. P. S. P. Baldi, “Deep autoencoder neural networks for gene ontology annotation predictions,” in *Proceedings of ACM Conference on Bioinformatics, Computational Biology, and Health Informatics (ACM BCB).* New York, NY, USA: ACM, 2014, pp. 533–540.

[6] S. Behnke, *Hierarchical Neural Networks for Image Interpretation.* Springer, 2003.

[7] S. Behnke and R. Rojas, "Neural abstraction pyramid: a hierarchical image understanding architecture," *Proceedings of International Joint Conference on Neural Networks (IJCNN)*, vol. 2, pp. 820–825, 1998.

[8] Y. Bengio, "Learning deep architectures for AI," *Foundations and Trends in Machine Learning*, vol. 2, no. 1, pp. 1–127, 2009, also published as a book. Now Publishers, 2009.

[9] Y. Bengio, P. Lamblin, D. Popovici, and H. Larochelle, "Greedy layer-wise training of deep networks," *Proceedings of Neural Information Processing Systems (NIPS)*, vol. 19, pp. 153–160, 2007.

[10] L. Breiman, "Bagging predictors," *Machine Learning*, vol. 24, no. 2, pp. 123–140, 1996.

[11] L. Breiman, J. Friedman, C. Stone, and R. Olshen, *Classification and Regression Trees*, ser. The Wadsworth and Brooks-Cole statistics-probability series. Taylor & Francis, 1984.

[12] L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001.

[13] A. E. Bryson, “A gradient method for optimizing multi-stage allocation processes,” in *Proceedings of Harvard University Symposium on Digital Computers and their Applications*, 1961.

[14] A. E. Bryson and Y.-C. Ho, *Applied optimal control: optimization, estimation, and control.* Blaisdell Publishing Company, 1969.

[15] P. J. Burt and E. H. Adelson, “The laplacian pyramid as a compact image code,” *IEEE Transactions on Communications*, vol. COM-31, no. 4, pp. 532–540, April 1983.

[16] R. Caruana, N. Karampatziakis, and A. Yessenalina, “An empirical evaluation of supervised learning in high dimensions,” in *Proceedings of the International Conference on Machine Learning (ICML)*, 2008.

[17] M. J. Choi, V. Y. Tan, A. Anandkumar, and A. S. Willsky, “Learning latent tree graphical models,” *Journal of Machine Learning Research*, vol. 12, no. May, pp. 1771–1812, 2011.

[18] D.-A. Clevert, T. Unterthiner, and S. Hochreiter, “Fast and accurate deep network learning by exponential linear units (ELUs),” in *Proceedings of International Conference on Learning Representations (ICLR)*, 2016.

[19] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. Kuksa, "Natural language processing (almost) from scratch," *Journal of Machine Learning Research*, p. 2537, 2011.

[20] N. Cressie, *Statistics for Spatial Data.* Wiley-Interscience, 1993.

[21] J. L. Crowley, "A representation for visual information," Carnegie Mellon University, Tech. Rep. CMU-RI-TR-82-7, 1981.

[22] G. Dahl, R. Adams, and H. Larochelle, "Training restricted Boltzmann machines on word observations," in *Proceedings of International Conference on Machine Learning (ICML).* ACM, 2012, pp. 679–686.

[23] G. Dahl, D. Yu, L. Deng, and A. Acero, "Context-dependent pre-trained deep neural networks for large vocabulary speech recognition," in *IEEE Transactions on Audio, Speech, and Language Processing*, 2012.

[24] I. Daubechies, "Orthonormal bases of compactly supported wavelets," *Communications on Pure and Applied Mathematics*, vol. 41, no. 7, pp. 909–996, 1988.

[25] I. Daubechies, "Ten lectures on wavelets," *Cmbs-Nsf Regional Conference Series in Applied Mathematics, Society for Industrias and Applied Mathematics*, vol. 61, 1998.

[26] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, Q. Le, M. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Ng, "Large scale distributed deep networks," in *Proceedings of Neural Information Processing Systems (NIPS)*, 2012.

[27] L. V. der Maaten and G. E. Hinton, "Visualizing data using t-sne," *Journal of Machine Learning Research*, vol. 9, no. 2579-2605, p. 85, 2008.

[28] S. Dong, G. Luo, G. Sun, K. Wang, and H. Zhang, "A combined multi-scale deep learning and random forests approach for direct left ventricular volumes estimation in 3d echocardiography," in *Computing in Cardiology Conference (CinC)*, Sept 2016, pp. 889–892.

[29] C. Dugas, Y. Bengio, F. Bélisle, C. Nadeau, and R. Garcia, "Incorporating second-order functional knowledge for better option pricing," in *Proceedings of Neural Information Processing Systems (NIPS)*, 2001, pp. 472–478.

[30] D. Erhan, Y. Bengio, A. Courville, P. Manzagol, and P. Vincent, "Why does unsupervised pre-training help deep learning?" *Journal of Machine Learning Research*, vol. 11, pp. 625–660, 2010.

[31] S. E. Fahlmann and G. E. Hinton, "Massively parallel architectures for ai: Netl, thistle, and boltzmann machines," in *Proceedings of National Conference on Artificial Intelligence (AAAI)*, 1983.

[32] M. Farhan, P. Ruusuvuori, M. Emmenlauer, P. Rämö, C. Dehio, and O. Yli-Harja, "Multi-scale gaussian representation and outline-learning based cell image segmentation," *BMC Bioinformatics*, vol. 14, no. 10, p. S6, 2013. [Online]. Available: http://dx.doi.org/10.1186/1471-2105-14-S10-S6

[33] M. Fernandez-Delgado, E. Cernadas, S. Barro, and D. Amorim, "Do we need hundreds of classifiers to solve real world classification problems?" *Journal of Machine Learning Research (JMLR)*, vol. 15, no. 1, pp. 3133–3181, October 2014.

[34] R. Finkel and J. Bentley, "Quad trees, a data structure for retrieval on composite keys," *Acta Informatica*, vol. 4, pp. 1–9, 1974.

[35] R. Fisher, "The use of multiple measurements in taxonomic problems," *Annals of Eugenics*, vol. 7, pp. 179–188.

[36] N. Fortier, J. W. Sheppard, and K. Pillai, "DOSI: Training artificial neural networks using overlapping swarm intelligence with local credit assignment," in *Proceedings of Joint International Conference on Soft Computing and Intelligent Systems (SCIS) and International Symposium on Advanced Intelligent Systems (ISIS).* IEEE, 2012, pp. 1420–1425.

[37] N. Fortier, J. W. Sheppard, and S. Strasser, “Abductive inference in bayesian networks using overlapping swarm intelligence,” *Soft Computing*, pp. 1–21, May 2014.

[38] N. Fortier, J. W. Sheppard, and S. Strasser, “Learning bayesian classifiers using overlapping swarm intelligence,” in *Proceedings of the Symposium on Swarm Intelligence (SIS)*, December 2014, pp. 205–212.

[39] K. Fukushima, “Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position,” *Biological Cybernetics*, vol. 36, no. 4, pp. 193–202, 1980.

[40] K. Fukushima, “Training multi-layered neural network neocognitron,” *Neural Networks*, vol. 40, pp. 18–31, April 2013.

[41] A. Gelfand, P. Diggle, M. Fuentes, and P. Guttor, *Handbook of Spatial Statistic.* CRC Press, 2010.

[42] D. George and J. Hawkins, “Invariant Pattern Recognition using Bayesian Inference on Hierarchical Sequences,” *Proceedings of International Joint Conference on Nerual Networks (IJCNN)*, 2005.

[43] Z. Ghahramani and P. S. Griffiths, Thomas L, “Bayesian nonparametric latent feature models,” *Bayesian Statistics*, vol. 8, pp. 1–25, 2007.

[44] Z. Ghahramani and T. L. Griffiths, “Infinite latent feature models and the indian buffet process,” in *Advances in neural information processing systems*, 2005, pp. 475–482.

[45] I. Goodfellow, Y. Bengio, and A. Courville, “Deep learning,” 2016, book in preparation for MIT Press. [Online]. Available: http://www.deeplearningbook.org

[46] B. Graham, “Fractional max-pooling,” *Computing Research Repository (CoRR)*, 2014.

[47] A. Graves, A. rahman Mohamed, and G. Hinton, “Speech recognition with deep recurrent neural networks,” in *Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2013, pp. 6645–6649.

[48] K. Greff, R. K. Srivastava, J. Koutník, B. R. Steunebrink, and J. Schmidhuber, “LSTM: A search space odyssey,” *Computing Research Repository (CoRR)*, vol. abs/1503.04069, 2015. [Online]. Available: http://arxiv.org/abs/1503.04069

[49] A. Grossman and J. Morlet, “Decomposition of hardy functions into square integrable wavelets of constant shape,” 1984.

[50] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” *Computing Research Repository (CoRR)*, 2015.

[51] K. He, X. Zhang, S. Ren, and J. Sun, “Spatial pyramid pooling in deep convolutional networks for visual recognition,” *IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI)*, vol. 37, no. 9, 2015.

[52] D. Heath, S. Kasif, and S. Salzberg, “Induction of oblique decision trees,” in *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, 1993, pp. 1002–1007.

[53] G. E. Hinton, S. Osindero, and Y. W. Teh, “A fast learning algortihm for deep belief nets,” *Neural Computation*, vol. 18, pp. 1527–1554, 2006.

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

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

[56] G. E. Hinton and R. Salakhutdinov, “Discovering binary codes for documents by learning deep generative models,” *Topics in Cognitive Science*, vol. 3, no. 1, pp. 74–91, 2011.

[57] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, “Improving neural networks by preventing co-adaptation of feature detectors,” *Computing Research Repository (CoRR)*, 2012.

[58] T. Ho, “Random decision forests,” in *Proceedings of International Conference on Document Analysis and Recognition (ICDAR)*, 1995, pp. 278–282.

[59] T. Ho, “The random subspace method for constructing decision forests,” *IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI)*, vol. 20, no. 8, pp. 832–844, 1998.

[60] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” vol. 9, no. 8, pp. 1735–1780, 1997.

[61] J. Hopfield, “Neural networks and physical systems with emergent collective computational abilities,” *Proceedins of the National Academy of Science (PNAS)*, vol. 79, no. 8, pp. 2554–2558, 1982.

[62] D. Hubel and T. Wiesel, “Receptive fields of single neurones in the cat’s striate cortex,” *Journal of Phisiology*, vol. 148, pp. 574–591, 1959.

[63] D. Hubel and T. Wiesel, “Receptive fields, binocular interaction and functional architecture in the cat’s visual cortex,” *The Journal of Physiology*, vol. 160, pp. 106–154, 1962.

[64] L. Hyafil and R. L. Rivest, “Constructing optimal binary decision trees is np-complete,” *Information Processing Letters*, vol. 5, no. 1, pp. 15–17, 1976.

[65] S. Ioffe and C. Szegedy, “Batch normalization: Accelerating deep network training by reducing internal covariate shift,” *Computing Research Repository (CoRR)*, 2015.

[66] A. Jain, M. Murty, and P. Flynn, “Data clustering: A review,” *ACM Computing Surveys*, 1999.

[67] I. Jolliffe, *Principal Components Analysis.* Springer, 2002.

[68] H. J. Kelley, “Gradient theory of optimal flight paths,” *ARS Journal*, vol. 30, no. 10, pp. 947–954, 1960.

[69] E. Kleinberg, “An overtraining-resistant stochastic modeling method for pattern recognition,” *The Annals of Statistics*, vol. 24, no. 6, pp. 2319–2349, 1996.

[70] T. Kohonen, *Self-Organizing Maps*, Third Edition ed. Springer, 2001.

[71] A. Krizhevsky, “Learning multiple layers of features from tiny images,” University of Toronto CS Department, Tech. Rep., 2009.

[72] A. Krizhevsky, V. Nair, and G. E. Hinton. Cifar-10 and cifar-100 datasets. Accessed 2016-08-01. [Online]. Available: http://www.cs.toronto.edu/~kriz/cifar.html

[73] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” vol. 25, pp. 1097 – 1105, 2012.

[74] J. B. Kruskal and M. Wish, *Multidimensional Scaling.*, ser. Sage University Paper. Quantitative Applications in the Social Sciences. SAGE Publications, Inc, 1978.

[75] S. Kuffler, "Discharge patterns and functional organization of mammalian retina," *Journal of Neurophysiology*, vol. 16, pp. 37–68, 1953.

[76] H. Larochelle and Y. Bengio, "Classification using discriminative restricted Boltzmann machines," in *Proceedings of International Conference on Machine Learning (ICML).* ACM, 2008, pp. 536–543.

[77] H. Larochelle, D. Erhan, A. Courville, J. Bergstra, and Y. Bengio, "An empirical evaluation of deep architectures on problems with many factors of variation," in *Proceedings of International Conference on Machine Learning (ICML)*, 2007, pp. 473–480.

[78] H. Larochelle, M. Mandel, R. Pascanu, and Y. Bengio, "Learning algorithms for the classification restricted boltzmann machine," *Journal of Machine Learning Research*, vol. 13, pp. 643–669, 2012.

[79] Y. LeCun, "Modèles connexionnistes de l'apprentissage," Ph.D. dissertation, Université Pierre et Marie Curie, Paris, France, 1987.

[80] Y. LeCun, Y. Bengio, and G. E. Hinton, “Deep learning,” *Nature*, vol. 521, pp. 436–444, 2015.

[81] Y. LeCun, B. Boser, J. Denker, D. Henderson, R. Howard, W. Hubbard, and L. Jackel, “Backpropagation applied to handwritten zip code recognition,” *Neural Computation*, vol. 1:4, pp. 541–551, 1989.

[82] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[83] Y. LeCun, C. Cortes, and C. Burges. The MNIST database of handwritten digits. Accessed 2016-08-01. [Online]. Available: http://yann.lecun.com/exdb/mnist/

[84] Y. LeCun, F. Huang, and L. Bottou, “Learning methods for generic object recognition with invariance to pose and lighting,” *Proceedings of Computer Vision and Pattern Recognition (CVPR)*, vol. 2, pp. 97–104, 2004.

[85] Y. LeCun, K. Kavukcuoglu, and C. Farabet, “Convolutional networks and applications in vision,” in *Proceedins of International Symposium on Circuits and Systems (ISCAS)*, 2010.

[86] C.-Y. Lee, P. W. Gallagher, and Z. Tu, “Generalizing pooling functions in convolutional neural networks: Mixed, gated, and tree,” in *Proceedings of International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2016, pp. 464–472.

[87] H. Lee, R. Grosse, R. Ranganath, and A. Ng, “Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations,” in *Proceedings of International Conference on Machine Learning (ICML).* ACM, 2009, pp. 609–616.

[88] D. Lemons, *A student’s guide to entropy.* Cambridge University Press, 2013.

[89] Y. Lettvin, H. Maturana, W. McCulloch, and W. Pitts, “What the frog’s eye tells the frog’s brain,” *Institute of Radio Engineering (IRE)*, vol. 47, pp. 1940–1951, 1959.

[90] M. Li, T. Zhang, Y. Chen, and A. Smola, “Efficient mini-batch training for stochastic optimization,” in *Proceedings of Conference on Knowledge Discovery and Datamining (KDD)*, 2014.

[91] Q. Liu, R. Hang, H. Song, and Z. Li, “Learning multi-scale deep features for high-resolution satellite image classification,” *Computing Research Repository (CoRR)*, 2016.

[92] J. Louradour and H. Larochelle, "Classification of sets using restricted Boltzmann machines," in *Proceedings of Conference on Uncertainty in Artificial Intelligence (UAI).* AUAI, 2011, pp. 463–470.

[93] A. Maas, A. Hannun, and A. Ng, "Rectifier nonlinearities improve neural network acoustic models," in *Proceedings of International Conference on Machine Learning (ICML)*, 2013.

[94] S. G. Mallat, "A Theory for multiresolution signal decomposition: The wavelet representation," *IEEE Transactions on Patern Analysis and Machine Intelligence (PAMI)*, vol. 11, pp. 674–693, July 1989.

[95] W. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *Bulletin of Mathematical Biophysics*, vol. 5, pp. 115–133, 1943.

[96] M. Minsky and S. Papert, *Perceptrons: an introduction to computational geometry.* MIT Press, 1969.

[97] D. Mishkin and J. Matas, "All you need is a good init," in *Proceedings of International Conference on Learning Representations (ICLR)*, 2016.

[98] T. M. Mitchell, "The need for biases in learning generalizations," Rutgers University, New Brunswick, NJ, Tech. Rep. CBM-TR-117.

[99] S. K. Murthy, S. Kasif, S. Salzberg, and R. Beigel, “Oc1: A randomized algorithm for building oblique decision trees,” in *Proceedings of National Conference on Artificial Intelligence (AAAI)*, vol. 93, 1993, pp. 322–327.

[100] V. Nair and G. E. Hinton, “Rectified linear units improve restricted boltzmann machines,” in *Proceedings of International Conference on Machine Learning (ICML)*, 2010.

[101] Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu, and A. Ng. The street view house numbers (svhn) dataset. Accessed 2016-08-01. [Online]. Available: http://ufldl.stanford.edu/housenumbers

[102] Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu, and A. Ng, “Reading digits in natural images with unsupervised feature learning,” *Proceedings of NIPS Workshop on Deep Learning and Unsupervised Feature Learning*, 2011.

[103] B. Olshausen *et al.*, “Emergence of simple-cell receptive field properties by learning a sparse code for natural images,” *Nature*, vol. 381, no. 6583, pp. 607–609, 1996.

[104] B. Payne and B. Gawronski, “A history of implicit social cognition: Where is it coming from? where is it now? where is it going?” in *Handbook of implicit social cognition: Measurement, theory, and applications.* Guilford Press, 2010, pp. 1–17.

[105] K. Pearson, "On lines and planes of closest fit to systems of points in space," *Philosophical Magazine Series 6*, vol. 2, no. 11, pp. 559–572, 1901.

[106] J. Quinlan, "Induction of decision trees," *Machine Learning*, vol. 1, pp. 81–106, 1986.

[107] J. Quinlan, *C4.5: Programs for Machine Learning.* Morgan Kaufmann Publishers, 1993.

[108] T. Randen and J. H. Husoy, "Filtering for texture classification: A comparative study," *IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI)*, vol. 21, no. 4, pp. 291–310, 1999.

[109] M. Ranzato, Y. Boureau, and Y. LeCun, "Sparse feature learning for deep belief networks," *Proceedings of Neural Information Processing Systems (NIPS)*, 2007.

[110] C. R. Rao, "The utilization of multiple measurements in problems of biological classification," *Journal of the Royal Statistical Society. Series B*, vol. 10, no. 2, pp. 159–203, 1948.

[111] A. Rauber, D. Merkl, and M. Dittenbach, "The Growing Hierarchical Self-Organizing Map: Exploratory Analysis of High-Dimensional Data," *IEEE Transactions on Neural Networks*, vol. 13, no. 6, pp. 1331–1341, November 2002.

[112] O. Rioul and M. Vetterli, “Wavelets and signal processing,” *IEEE Signal Processing Magazine*, vol. 8, no. 4, pp. 14–38, 1991.

[113] F. Rosenblatt, *The Perceptron–a perceiving and recognizing automaton.* Cornell Aeronautical Laboratory, 1957, vol. Report 85-460-1.

[114] F. Rosenblatt, *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms.* Spartan Books, 1962.

[115] S. Roweis and L. Saul, “Nonlinear dimensionality reduction by locally linear embedding,” *Science*, vol. 290, no. 5500, pp. 2323–2326, 2000.

[116] D. Rumelhart, G. E. Hinton, and R. Williams, “Learning representations by back-propagating errors,” *Nature*, vol. 323, pp. 533–536, 1986.

[117] R. Salakhutdinov and G. E. Hinton, “Deep Boltzmann machines,” in *Proceedings of International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2009, pp. 448–455.

[118] R. Salakhutdinov, A. Mnih, and G. E. Hinton, “Restricted Boltzmann machines for collaborative filtering,” in *Proceedings of International Conference on Machine Learning (ICML).* ACM, 2007, pp. 791–798.

[119] B. Schölkopf, C. J. Burges, and A. J. Smola, *Advances in kernel methods support vector learning.* MIT Press, 1998.

[120] B. Schölkopf, A. Smola, and K.-R. Müller, “Kernel principal component analysis,” *International Conference on Artificial Neural Networks(ICANN)*, vol. 1327, pp. 583–588, 1997.

[121] P. Sermanet and Y. LeCun, “Traffic sign recognition with multi-scale convolutional networks,” in *International Joint Conference on Neural Networks (IJCNN)*, July 2011, pp. 2809–2813.

[122] R. N. Shepard, “The analysis of proximities: Multidimensional scaling with an unknown distance function. i,” *Psychometrika*, vol. 27, pp. 125–140, 1962.

[123] R. N. Shepard, “The analysis of proximities: Multidimensional scaling with an unknown distance function. ii,” *Psychometrika*, vol. 27, pp. 219–246, 1962.

[124] D. Silver, A. Huang, C. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, “Mastering the game of go with deep neural networks and tree search,” *Nature*, vol. 529, pp. 484–489, 2016.

[125] K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” in *Proceedings of Interantional Conference on Learning Representations (ICLR)*, 2015.

[126] P. Smolensky, “Information processing in dynamical systems: Foundations of harmony theory,” in *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1*, D. E. Rumelhart and J. L. McClelland, Eds. Cambridge, MA, USA: MIT Press, 1986, pp. 194–281.

[127] J. Springenberg, A. Dosovitskiy, T. Brox, and M. Reidmiller, “Striving for simplicity: The all convolutional net,” in *Workshop at the International Converence on Learning Representations (ICLR)*, 2015.

[128] S. Strasser, J. W. Sheppard, N. dortier, and R. Goodman, “Factored evolutionary algorithms,” *IEEE Transactions on Evolutionary Computation*, vol. PP, no. 99, pp. 1–1, 2016.

[129] S. T. Strasser, “Factored evolutionary algorithms: Cooperative coevolutionary optimization with overlap,” Ph.D. dissertation, Montana State University, 2016.

[130] I. Sutskever, J. Martens, G. Dahl, and G. E. Hinton, “On the importance of initialization and momentum in deep learning,” *Journal of Machine Learning Research (JMLR)*, vol. 28, pp. 1139–1147, 2013.

[131] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi, “Inception-v4, inception-resnet and the impact of residual connections on learning,” in *Proceedings of International Conference on Learning Representations (ICLR)*.

[132] J. Tenenbaum, V. deSilva, and J. Langford, "A global geometric framework for nonlinear dimensionality reduction," *Science*, vol. 290, pp. 2319–2323, 2000.

[133] T. Tieleman, "Training restricted Boltzmann machines using approximations to the likelihood gradient," in *Proceedings of International Conference on Machine Learning (ICML).* ACM, 2008, pp. 1064–1071.

[134] T. M. Tomita, M. Maggioni, and J. T. Vogelstein, "Randomer forests," *Computing Research Repository (CoRR)*, 2015.

[135] A. Torralba, R. Fergus, and W. Freeman, "Tiny images," MIT CSAIL, Tech. Rep., 2007.

[136] H. Tosun and J. W. Sheppard, "Training restricted Boltzmann machines with overlapping partitions," in *Machine Learning and Knowledge Discovery in Databases (ECML-PKDD)*, ser. Lecture Notes in Computer Science. Springer, 2014, vol. 8726, pp. 195–208.

[137] M. Turk and A. Pentland, "Face recognition using eigenfaces," *Proceedings of Computer Vision and Pattern Recognition (CVPR)*, pp. 586–591, 1991.

[138] P. Vincent, H. Larochelle, Y. Bengio, and P. Manzagol, "Extracting and composing robust features with denoising autoencoders," in *Proceedings of International Conference on Machine Learning (ICML)*, 2008.

[139] L. Wan, M. Zeiler, S. Zhang, Y. LeCun, and R. Fergus, "Regularization of neural network using dropconnect," in *Proceedings of Interational Conference on Machine Learning (ICML)*, 2013.

[140] P. Werbos, "Beyond regression: New tools for prediction and analysis in the behavioral sciences," Ph.D. dissertation, Harvard University, 1974.

[141] L. Williams, "Pyramidal parametrics," in *Computer Graphics (Proceedings of SIGGRAPH)*, 1983.

[142] D. H. Wolpert and W. G. Macready, "No free lunch theorems for search," The Santa Fe Institute, Santa Fe, NM, Tech. Rep. SFI-TR-95-02-010.

[143] J. Xie, L. Xu, and E. Chen, "Image denoising and inpainting with deep neural networks," in *Proceedings of Neural Information Processing Systems (NIPS)*, 2012, pp. 341–349.

[144] X. Zhao, J. Wan, G. Ren, J. Liu, J. Chang, and D. Yu, "Multi-scale dbns regression model and its application in wind speed forecasting," in *Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC)*, Oct 2016, pp. 1355–1359.

[145] B. Zhou, A. Khosla, A. Lapedriza, A. Torralba, and A. Oliva. Ilsvrc2015 results. Accessed 2016-08-01. [Online]. Available: http://image-net.org/challenges/LSVRC/2015/download-images-3j16.php

[146] B. Zhou, A. Khosla, A. Lapedriza, A. Torralba, and A. Oliva, "Places2: A large-scale database for scene understanding," *pre-print*, 2015.

Vita

Benjamin R. Mitchell was born in 1983 in New Haven, CT, USA. He received a B.A. in Computer Science from Swarthmore College in 2005, and a M.S.E. in Computer Science from the Johns Hopkins University in 2008. He received a certification from the JHU Preparing Future Faculty Teaching Academy in 2016.

He has worked as a Teaching Assistant and a Research Assistant from 2005 to 2008, and he has been an Instructor at the Johns Hopkins University since 2009. He has taught courses including Introductory Programming in Java, Intermediate Programming in C/C++, Artificial Intelligence, and Computer Ethics. In 2015, he received the Professor Joel Dean Award for Excellence in Teaching, and he was a finalist for the Whiting School of Engineering Excellence in Teaching Award in 2016.

In addition to the field of machine learning, he has peer-reviewed publications in fields including operating systems, mobile robotics, medical robotics, and semantic modeling.