DEMYSTIFYING UNSUPERVISED FEATURE LEARNING

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Adam Coates
September 2012

Abstract

Machine learning is a key component of state-of-the-art systems in many application domains. Applied to many kinds of raw data, however, most learning algorithms are unable to make good predictions. In order to succeed, most learning algorithms are applied instead to "features" that represent higher-level concepts extracted from the raw data. These features, developed by expert practitioners in each field, encode important prior knowledge about the task that the learning algorithm would be unable to discover on its own from (often limited) labeled training examples. Unfortunately, engineering good feature representations for new applications is extremely difficult. For the most challenging applications in AI, like computer vision, the search for good features and higher-level image representations is vast and ongoing.

In this work we study a class of algorithms that attempt to learn feature representations automatically from unlabeled data that is often easy to obtain in large quantities. Though many such algorithms have been proposed and have achieved high marks on benchmark tasks, it has not been fully understood what causes some algorithms to perform well and others to perform poorly. It has thus been difficult to identify any key directions in which the algorithms might be improved in order to significantly advance the state of the art. To address this issue, we will present results from an in-depth scientific study of a variety of factors that can affect the performance of feature-learning algorithms. Through a detailed analysis, a surprising picture emerges: we find that many schemes succeed or fail as a result of a few (easily overlooked) factors that are often orthogonal to the particular learning methods involved. In fact, by focusing solely on these factors it is possible to achieve state-of-the-art performance on common benchmarks using quite simple algorithms. More

importantly, however, a main contribution of this line of research has been to identify very simple yet highly scalable feature learning methods that, by virtue of focusing on the most critical properties identified in our study, are highly successful in many settings: the proposed algorithms consistently achieve top performance on benchmarks, have been successfully deployed in realistic computer vision applications, and are even capable of discovering high-level concepts like object classes without any supervision.

For my parents, Paul and Kate, and Lisa. And for my grandfather:
"I don't know how it happened, but you grandkids got so darn smart."
— George Schirle

Acknowledgments

I don't understand why, but I have had the very best of virtually everything in life, so it is only with great difficulty that I try to sufficiently acknowledge the many people that have made my education and this thesis possible. I have had the great fortune of being raised in an amazing family, taught by excellent teachers my entire life, cared for by great friends, and challenged by bright students and coworkers at every turn. Even counting the many all-nighters, I do not think I have done anything especially extraordinary to deserve the many gifts I have received over the years, but I am always trying to live up to those advantages; I hope that this thesis is one step along the way to doing so.

First, it goes without saying that my parents, Paul and Kate, have been major collaborators in life. I was always able to hack away at my own path in the world without terribly much to worry about. It is clear now, older as I am, how great a privilege this was—it is barely an accomplishment to have done well in high school and in college when so much other weight has been carried by fantastic parents. I am not sure how to thank them for this except to share with them the credit for all that I do.

In addition to my parents, the rest of my family deserves a great deal of thanks. My brothers, Bob and Kevin, if only tenuously at times, have always been supportive. Sometimes that support comes in the form of poking, prodding, teasing, or bear-hugging; but I try to think of it as all the same (positive) thing. They have been my playmates for longer than anyone else on the planet, and that has continued to be true throughout my PhD study, even when we can't actually be in the same place. My gratitude also extends to my grandparents (and not just by propagating credit

through my parents). Thanks especially to my grandfather, George Schirle, who I blame for my sense of humor. Though "Gramps" spent his entire career at Sears, his reach in the world is large and growing through his love for family and especially us grandkids. I could be no more blessed myself than to leave behind as much joy in the world as he has.

And, not least amongst family, I thank my wife and best friend Lisa. She is the least expected treasure I have found or ever expect to find. I cannot imagine having made it this far without her encouragement, patience and loyalty. On the other side of every hill she is waiting; and I will never manage to thank her sufficiently for that feat.

My friends, too, need a nod. To Joe Karam, Bryan Lung, Kapil Jain, Justin and Varun Tansuwan, Katherine Chou, Shantanu Tarafdar, Rob French, Helene Butler and Connie Nelson: thank you. I am an incredibly tricky friend to maintain, yet somehow this crowd of special human beings has managed. They have dragged me out to eat, to movies, to bowling, to ball games, and more when I was pretty sure that work could not wait. Finally thanks to Benedict Tse who, as a coworker and friend, was my inspiration for making many changes in life.

My education has been the product of many mentors. In high school I was fortunate to have many excellent teachers, not to mention the support of amazing staff and counselors. Calistoga High School is small, but I simply could not have asked for a better education. Much gratitude goes to Ivan Miller, my math teacher, who took me on countless field trips to math and computer events. His encouragement and direction, I think, were major ingredients in my acceptance to Stanford as an undergrad. Thanks also to Mark Greenway, Gary Guttman and Tom Abbey of the CHS humanities programs for encouraging me to become more than just a math nerd.

In grad school, I have had the ridiculous luck of being mentored by two amazing researchers: Pieter Abbeel and, of course, my advisor Andrew Ng. Pieter and I worked countless hours on autonomous helicopters, trying relentlessly to get them to fly aerobatic routines. During the several years we spent working on the project together we enjoyed many late-night brainstorming and debugging sessions. These were some of the events that helped me decide to apply to PhD programs. Thinking

back, those were also the times when I learned the most about how to do research, how to think about problems, and—importantly for me—how to keep my eye on the ball. Even more, outside of the lab, Pieter has always been a great friend and I continue to look forward to our chats.

I don't know where to start thanking Andrew, since our history together is now nearly 10 years long. I don't think I can imagine any professional quality that Andrew has not helped me to improve, sometimes up from zero. Paper writing, giving technical talks, choosing research topics, staying organized, mentoring students—he has made quite a large investment of time in my upbringing as a researcher. For the many many many items in my "advice to self" file that are attributable to him and for the example he has set, I am most thankful.

I also want to thank the many excellent undergraduate and MS students I have worked with over the years since I began research at Stanford (many of whom are co-authors on the handful of papers we have published together): Paul Baumstarck, Eric Berger, Blake Carpenter, Carl Case, Eric Liang, Tim Hunter, Brody Huval, Sanjeev Satheesh, Bipin Suresh, Tao Wang, and David Wu. Aside from the knowledge I have gained from working with each of them, I cannot begin to thank them for their patience with my novice (but hopefully improving!) advice and direction. I continue to enjoy working with Tao, David, and Brody on new research even as I write this thesis. Constantly I am reminded by their skill and hard work of the rewards of research and of being a Stanford student.

Of course, I have spent a great deal of time with the other PhD students in the AI lab, especially in Andrew's research group. Alan Asbeck, Varun Ganapathi, Honglak Lee, Quoc Le, Andrew Maas, Dave Jackson, Jiquan Ngiam, Alex Karpenko, Zico Kolter, Morgan Quigley, Rajat Raina, Andrew Saxe, Richard Socher, Yirong Shen, and Will Zou have all had to put up with my wandering into their office at some point[1]—sometimes to talk about research, but usually to bother them with something random. Their tolerance for my distractions is extraordinary, and I often feel that this is the part of Stanford that made me feel most at home. They have laughed at terrible technical jokes, shared their insights into problems not remotely

[1]By the length of that list, you can see that I still do a great deal of wandering.

related to anybody's research, and commiserated when NIPS reviews came back sour. Especially I give thanks to Quoc, Andrew S. and Andrew M. for allowing me to pick their brains repeatedly (sometimes needlessly or for entertainment only). And also to Morgan and Zico who, in addition to letting me borrow their great minds on occasion, have also been great companions throughout grad school. Indeed, if I could pick one thing to carry through to my life beyond the PhD program, it would be our lunch-time walks and conversations.

Finally, thanks to the excellent technical support, staff and administrators of the Computer Science Department who have kept our computers running and helped me navigate through 3 degree programs. Thanks also to the contributors and administrators of the Stanford Graduate Fellowship who have made my study financially feasible—I hope that I will succeed in passing on this generosity to others.

My thanks go to all of these people; this is a journey I never expected to be on and barely imagined finishing. I am here thanks to their contributions; I hope they have enjoyed the ride as much as I have.

Contents

Abstract **iv**

Acknowledgments **vii**

1 Introduction **1**

1.1 Supervised Learning, Self-Taught Learning, and Unsupervised Feature Learning . . . 3

1.2 Overview of Contributions . . . 5

1.3 Outline . . . 6

1.4 First Published Appearances . . . 8

2 Background **10**

2.1 Feature Representations . . . 10

2.1.1 Features in Supervised Learning . . . 10

2.1.2 Feature Hierarchies . . . 12

2.2 Locally Connected, Convolutional Features for Images . . . 12

2.3 Template for Unsupervised Feature Learning . . . 14

2.4 Unsupervised Learning Algorithms . . . 16

2.4.1 Clustering . . . 16

2.4.2 Orthogonal Matching Pursuit . . . 18

2.4.3 Sparse Coding . . . 19

2.4.4 Auto-encoders . . . 20

2.5 Feature Encodings . . . 22

2.6 Summary . . . 23

3 A Scientific Study of UFL **24**
3.1 Related Work . . . 25
3.2 Standard Feature Learning Framework for Image Recognition . . . 27
3.2.1 Feature Learning from Patches . . . 28
3.3 Experimental Results and Analysis . . . 36
3.3.1 Effect of pipeline parameters . . . 36
3.3.2 Encoding versus training . . . 45
3.4 Summary . . . 54

4 Selecting Receptive Fields in Deep Networks **55**
4.1 Introduction . . . 55
4.2 Related Work . . . 57
4.3 Algorithm Details . . . 59
4.3.1 Similarity of Input Features . . . 60
4.3.2 Selecting Local Receptive Fields . . . 61
4.3.3 Approximate Similarity . . . 61
4.3.4 Learning Architecture . . . 63
4.3.5 Feature Hierarchy Details . . . 64
4.4 Experimental Results . . . 65
4.4.1 Comparison on CIFAR-10 . . . 66
4.4.2 Comparison on STL-10 . . . 69
4.5 Summary . . . 70

5 Application to Scene Text Recognition **71**
5.1 Introduction . . . 71
5.2 Related Work . . . 72
5.3 Feature Learning Architecture . . . 73
5.3.1 Feature extraction . . . 74
5.3.2 Text detector training . . . 74
5.3.3 Character classifier training . . . 74
5.4 Detection and Recognition Experiments . . . 76
5.4.1 Detection . . . 76

5.4.2 Character Recognition 78
5.5 Summary 79

6 Emergence of Object-Selective Features **80**
6.1 Introduction 80
6.2 Algorithms 82
6.2.1 Learning Selective Features (Simple Cells) 82
6.2.2 Learning Invariant Features (Complex Cells) 83
6.3 Algorithm Behavior 84
6.3.1 Feature Hierarchy 86
6.4 Experiments 87
6.4.1 Low-Level Simple and Complex Cell Visualizations 88
6.4.2 Higher-Level Simple and Complex Cells 88
6.5 Related Work 92
6.6 Summary 94

7 Conclusions **96**

Bibliography **98**

List of Tables

3.1 UFL system test results on CIFAR-10 43
3.2 UFL system test results on NORB (normalized-uniform) 44
3.3 CIFAR-10 CV accuracy for encoder vs. training algorithm comparison 46
3.4 CIFAR-10 results for encoder / training algorithm combinations . . . 48
3.5 NORB (jittered-cluttered) test accuracy for encoding / training algorithm combinations 49
3.6 Caltech 101 test accuracy for encoder / training algorithm combinations 50

4.1 RF learning results on CIFAR-10 test set 68
4.2 RF learning results on small CIFAR-10 test set 68
4.3 RF learning results on STL-10 test 69

5.1 Performance comparison on ICDAR 2003 character datasets 78

6.1 Quantiative comparison of selectivity of cell types for face images . . 89

List of Figures

2.1 Notation and setup of locally connected, convolutional features. . . . 13
2.2 Standard template for constructing unsupervised feature learning systems. . . . 15
2.3 Diagram of a 1-layer auto-encoder. . . . 21

3.1 Diagram of image recognition pipeline. . . . 34
3.2 CIFAR-10 learned basis comparison 38
3.3 Comparisons for whitening and feature count 39
3.4 Comparisons for change in step size 41
3.5 Comparison of receptive field sizes 42
3.6 Performance of sparse coding vs. soft-threshold encoding on small datasets 52

4.1 Learned RF examples 67
4.2 Learned 2nd layer feature examples 67

5.1 Bases learned from scene text data 73
5.2 Synthetic and distorted scene text data 75
5.3 ICDAR 2003 dataset detection results 77
5.4 Text detection examples 77
5.5 Character recognition accuracy on ICDAR 2003 dataset 78

6.1 Visualization of simple and complex cell learning algorithm behavior . 86
6.2 Network architecture and dataset samples 88
6.3 First layer feature visualizations 90

6.4 Visualizations of 2nd layer simple and complex cells 95

Chapter 1

Introduction

In machine learning applications our typical goal is to learn a mapping from input patterns to an output value. For instance, a common task in computer vision is to learn to map an input image, represented by pixel intensity values, to a "label" that identifies the object pictured in the image. To learn this mapping, we furnish an algorithm with a large number of examples of correct input and output pairs and the algorithm must learn a mapping that can predict the correct output value when presented with a novel input. Learning to make such predictions directly, however, turns out to be quite difficult: the mapping from pixel intensities to object class labels, for instance, is an extremely complex, non-linear function that most machine learning algorithms cannot discover on their own. As a result, researchers in many fields attempt to engineer "features" that transform the raw input data into a new representation (a vector of feature values) that makes the important characteristics of the input more apparent. This simplifies the mapping that must be learned and can lead to much better performance. Unfortunately, the process of engineering features for each new application is arduous. For extremely complex tasks like computer vision or speech recognition, the complexity of the necessary representations is so great that building features based on prior knowledge is extraordinarily difficult.

In this thesis, we will study algorithms that attempt to *learn* the feature representations themselves from data. Such algorithms, in principle, could develop features that can more accurately represent the important qualities of the input data and

surpass the level of complexity and variety of features that could reasonably be imagined or implemented by human experts. Yet while many such algorithms have been proposed, and a growing field of researchers have gained experience in employing them, they can often be as complex and unwieldy as the hand-built features that they replace. As a result, while hopes are high that such methods might alleviate the requirement to engineer features once and for all, it has been unclear exactly what characteristics of these algorithms make them successful and in what ways they could clearly be made better. For instance, many algorithms that are apparently quite similar often differ dramatically in performance while others that are apparently very different may achieve similar scores on common benchmarks. Similarly, though we often have a strong intuitive sense for the types of features our algorithms should discover (such as object parts or object classes in images) it has been unclear whether existing algorithms could learn these features or whether more sophisticated approaches are necessary.

This work attacks these problems through an in-depth empirical study of existing algorithms, attempting to tease out the fundamental ingredients that make some algorithms more successful than others. From this study, we will identify a number of critical components. One key result is that the *scale* of our representations (the number of features, for instance) is a crucial factor in determining performance. Using these results, we will identify a very simple yet highly successful feature learning system that is not only competitive on many common benchmarks but is also a state-of-the-art performer in real-world applications like scene text recognition. Further, we will see that essentially the same ideas, with just a few extra ingredients, can discover the kinds of high level concepts we expect: the system that we develop turns out to be capable of discovering a commonly occurring object class (human faces) in unlabeled images using no supervision at all.

1.1 Supervised Learning, Self-Taught Learning, and Unsupervised Feature Learning

The classic machine learning setting described above, where we are given a set of labeled examples and attempt to learn to predict correct labels for novel inputs, is known as "supervised learning". This framework encompasses a vast number of algorithms and successful real-world applications. In this setting, we assume that we are provided with a set of m example input vectors $x^{(i)}, i = 1, \ldots, m$ and their corresponding (correct) labels $y^{(i)}$. For classification problems (which we will work with throughout this thesis), we have $y^{(i)} \in \{0, 1\}$. The goal of supervised learning is to train a function $y = f(x; \theta)$ to predict the correct label y for a novel input vector x. The vector θ is a set of parameters optimized by the learning algorithm to make $f(x; \theta)$ as good a predictor as possible.

If we choose a very simple class of functions $f(x; \theta)$, say linear $f(x; \theta) = \theta^\top x$, then f will only be able to make very simple predictions from its inputs. On the other hand, if we have relatively little training data, we will not be able to train a complex function $f(x; \theta)$ that has many parameters due to the risk of overfitting (and hence poor performance on test data). As a result, most applications employ "features": We transform each n-dimensional input $x \in \mathbb{R}^n$ into a new representation $\phi \in \mathbb{R}^K$ composed of K features, and then apply our learning algorithm to these feature vectors to learn a function $f(x; \theta) = \theta^\top \phi$. The feature vectors ϕ are computed directly from x using a function $\phi = \Phi(x)$ that is often designed by hand for each application. When we can choose good features, this scheme works much better than using raw inputs and does not necessitate training extremely large numbers of parameters from our labeled data. We can think of the feature functions $\Phi(x)$ as providing a way for us to encode prior knowledge into the learning system. Each feature value often represents a "higher-level" concept that we do not expect our supervised learning algorithm to find given only the labeled data. For instance, when working with images, good features often try to detect edges, shapes, or textures that might convey important high-level information about the input.

For the hardest applications, though, it can be extremely difficult to invent new

features by hand. Other applications may be highly specialized or novel and thus do not attract sufficient attention to foment the creation of new features when needed. One way around the problem is to use very large quantities of labeled data [22, 3, 84, 13, 79, 12, 51], and then train a highly expressive class of functions f (e.g., a deep neural network). This approach has seen some success, but is limited by our ability to acquire enough labeled data for every task that we wish to solve.

The self-taught learning (STL) framework has been proposed [67] as one alternative to using only labeled data. In self-taught learning we are additionally given an extremely large set of *unlabeled* examples. These examples often come from a similar but much broader distribution than our labeled examples. For instance, in an object recognition task, our labeled examples might contain cropped images of different types of objects, yet our unlabeled examples might be random snippets cropped from images downloaded from the Internet that do not necessarily contain any particular object at all. (This contrasts self-taught learning from semi-supervised learning [10] where we assume that the unlabeled images *could* be labeled, but that the label has not been provided.) A major advantage of this setup is that unlabeled data is very easy to acquire and, conceivably, could help our algorithms to understand the basic properties of the underlying data distribution (the distribution of the $x^{(i)}$) so that subsequent learning tasks are easier. We may think of this as a way for algorithms to acquire prior knowledge or experience from unlabeled examples in addition to labeled examples. Because unlabeled data is plentiful, we can potentially train highly sophisticated models that would be impossible to train from only labeled data and thus supplant much of the prior knowledge that we would otherwise need to encode by hand.

Unsupervised Feature Learning (UFL) is one approach to tackling problems within the self-taught learning framework. The main goal here is to learn, from only unlabeled examples, a useful feature representation $\Phi(x)$ that can then be used for other tasks (e.g., for supervised learning with a linear function class as above). This can be done by letting $\Phi(x)$ be parametrized by a set of parameters Θ. The goal of a UFL algorithm is to tune Θ so that the resulting feature vectors $\phi = \Phi(x; \Theta)$ are a "better" representation than the raw inputs x. Note that while many of the evaluation

tasks considered in our work are supervised classification tasks, our goal is to learn higher-level features that are generally useful for many tasks. Thus we will usually work with UFL algorithms as a "black box": they take in unlabeled data and learn parameters Θ without any integration with the particular task for which the features $\Phi(x;\Theta)$ will be used. We will present a simple template for building algorithms that do this in Chapter 2. Much of the rest of this thesis will be concerned with analyzing these algorithms and finding the best combinations of ingredients based on our experiments.

1.2 Overview of Contributions

A significant degree of confusion has surrounded the many algorithms and applications that may be included under the heading "unsupervised feature learning". Many of these algorithms can be viewed as different implementations of similar ideas and yet their relative performance on benchmarks can vary dramatically. The reasons for these basic differences have not been studied thoroughly, and thus it has been hard to determine ways to make progress (either on individual algorithms or on the entire class of methods generally). Indeed, with increasingly sophisticated models performance on benchmarks generally improved without making clear exactly which ideas led to the improvement.

As a result of this situation, a major focus of this work is to perform an extensive study of several key factors that can play a role in the performance of various feature-learning algorithms, like the number of features that we learn or the ways in which we pre-process or slice up our data before applying a UFL method. Often these factors are not peculiar to a specific algorithm, and we will be able to draw broad conclusions that hold not only for specific algorithms but hold broadly for a large number of feature learning methods.

Among these key factors it will become apparent that one of the drivers behind progress is *scale*: algorithms that can learn large numbers of features are consistently better performers. Motivated by this, and based on our empirical study, we will develop a highly successful and very scalable feature learning method. The resulting

system is an excellent performer on many standard benchmarks and is also useful for practical applications. More importantly, however, we show that the same basic components are capable of constructing sophisticated representations that capture high-level concepts like object classes, and complex invariances in images. All of these results rest critically on a few basic ideas that can be implemented quite easily yet become powerful as a result of our focus on large scale implementation.

1.3 Outline

This thesis will proceed as follows:

Chapter 2: Background. Chapter 2 will cover some background material on learning hierarchical feature representations. This literature, which includes much work from neural networks and unsupervised learning, is extremely large as a result of decades of development. We will focus on a few major ingredients as well as terminology that we have adopted from this literature. A key component of our work will be unsupervised learning algorithms, which are the primary learning mechanism in most of our systems. We will review several such algorithms that will show up in our later work and also point out that these algorithms are often implementing similar ideas in slightly different ways.

Chapter 3: A Scientific Study of UFL. We will begin with our study of feature learning algorithms. Noting that many implementations in the literature differ along just a few axes, we will systematically study the effect of several parameters common to many different algorithms. We will do this by laying down a simple "pipeline" into which we can substitute many different feature learning modules. It will turn out that the parameters that control the setup of this pipeline are often far more important than how we go about learning the features themselves. In some cases these parameters are numerical hyperparameters that we can select via cross-validation or rules of thumb based on our results. Other factors, such as how we encode features using the results of our unsupervised learning algorithm, can also have a very large effect. But the

main impact of this study has been to lessen the focus on new models for unsupervised learning and divert our attention to the more critical issues identified in our experiments: scalability and careful choice of the various exogenous parameters in the system. This will motivate subsequent work on scalable, easily tuned systems that zero in on these issues rather than developing new learning methods.

Chapter 4: Selecting Receptive Fields in Deep Networks. Many algorithms assume that the "connectivity" of features to inputs is known *a priori*. It is easy enough to connect every feature to every input, but in these cases the unsupervised learning procedure must learn parameters for all of the pair-wise (feature-to-input) relationships. When we have extremely large numbers of inputs and features, this becomes prohibitive for many learning algorithms. We can solve this problem by hand-coding the connectivity in order to reduce the computational burden, but this solution turns out to be ineffective for deep networks (for reasons we will explain). In order to improve the scalability of algorithms for training deep networks, we will introduce a method for determining the connectivity of networks rapidly before the unsupervised learning phase. As a result, we are able to train much larger networks and employ unsupervised learning schemes that normally would not scale well to the very large representations we consider.

Chapter 5: Application to Scene Text Recognition. Though feature learning methods are showing a great deal of promise for learning complex concepts that would be difficult to identify with hand-coded features, the results of this work are already yielding benefits in realistic applications. Due to the recent proliferation of mobile phones with cameras, an interesting computer vision application has emerged: detection and recognition of text in natural images. This problem is much more difficult than document character recognition—which is largely solved—and has received some attention from machine learning practitioners in the past. As a result, a handful of engineered systems with custom features have been built, but the problem has not been solved satisfactorily for widespread

use. In this chapter we will apply our feature learning system in an essentially "off the shelf" manner to the problem of scene text recognition and find that, indeed, this approach is competitive, and usually superior, with more sophisticated systems developed elsewhere. This work concretely demonstrates the benefits that may be obtained by focusing on learned feature representations to supplant hand-engineered features.

Chapter 6: Emergence of Object-Selective Features. Though a great deal of work on feature learning focuses on discriminative tasks, there are many interesting questions to be asked about what types of concepts such algorithms are capable of learning on their own. Ideally, a sufficiently sophisticated method should discover high-level concepts like object classes in images, or syllables and words in audio clips. Intuitively we believe that these types of features capture the important high-level information present in the data. Yet despite some results suggesting that a few existing algorithms could discover such structure (like decomposition of objects into object parts) in highly restrictive scenarios, it has remained unknown whether additional ingredients might be necessary to achieve similar results in the general case. Based on our prior results we will show that, in fact, no additional sophistication is necessary: at very large scale, even very "obvious" notions of selectivity and invariance—already present in a wide range of algorithms—are enough to yield multi-layered networks of features that are selective for a commonly occurring object (human faces) with robustness to surprisingly complex types of distortion.

1.4 First Published Appearances

Much of the work presented here has appeared first in other publications. The results of Chapter 3 appeared in [16] and [17], though the discussion here is amplified. The receptive field learning procedure of Chapter 4 first appeared in [18]. Chapter 5 includes results on scene text recognition that first appeared in [14]. Finally, our work in Chapter 6 on the emergence of object-selective features from our systems trained

on unlabeled imagery covers results from [15] with some additional discussion.

Chapter 2

Background

Much work in Unsupervised Feature Learning adopts terminology and algorithmic components from neural networks and unsupervised learning. In this chapter we will cover some of the important components from this literature that we will re-use frequently.

2.1 Feature Representations

2.1.1 Features in Supervised Learning

A typical goal in supervised learning is to fit a function $\hat{y} = f(x; \theta)$ to a given set of example input-output pairs $(x^{(i)}, y^{(i)}), i = 1, \ldots, m$ where each $x^{(i)}$ is a vector of input values and $y^{(i)}$ is a corresponding vector of target outputs. We will often concatenate these vectors column-wise into matrices X and Y. So, for instance, $X_{ji} = x_j^{(i)}$. To train the predictor f we typically optimize over the parameters θ to find the optimal choice $\theta^\star$ that minimizes the expected value of a loss function $\mathcal{L}$ on the training data:

$$\theta^\star = \arg\min_\theta \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\hat{y}, y^{(i)}).$$

For example, in linear regression we may minimize squared error over linear functions $\hat{y} = f(x;\theta) = \theta^\top x$:

$$\begin{aligned}\theta^\star =& \arg\min_\theta \ \frac{1}{m}\sum_{i=1}^{m} ||\theta^\top x^{(i)} - y^{(i)}||_2^2 \\ =& \arg\min_\theta \ ||\theta^\top X - Y||_{\mathcal{F}}^2.\end{aligned}$$

Of course, linear functions as above are not sufficiently expressive for many tasks and thus we previously introduced the notion of feature functions $\Phi(x)$ to first transform our inputs $x \in \mathbb{R}^n$ into feature vectors $\phi \in \mathbb{R}^K$ that may be more useful for predicting y. Given such feature functions, we can simply train f by minimizing $\mathcal{L}(f(\Phi(x^{(i)});\theta), y^{(i)})$ on the training data. Since the functions $\Phi(x)$ may be chosen arbitrarily (even producing feature vectors of larger or smaller dimensionality than x) we may construct very complex functions $f(x;\theta) = \theta^\top \Phi(x)$. If we are clever in our choice of features, this may allow us to make much better predictions of y than would be possible using the raw inputs x directly.

In this thesis we will be primarily concerned with the case that the feature functions $\Phi(x)$ are themselves parametrized by a set of values Θ that may be trained or tuned. For example, one simple choice that we will use frequently is:

$$\Phi(x;\Theta) = g(Wx + b) \tag{2.1}$$

where the parameters $\Theta = (W, b)$ are the parameters of an affine (vector-valued) function, and $g(\cdot)$ is some simple element-wise nonlinear function. In a supervised learning setting like the one above, our goal would be to tune both θ (the parameters of $f(\cdot)$) and Θ (the parameters of $\Phi(\cdot)$) to perform as well as possible on the final prediction task:

$$\theta^\star, \Theta^\star = \arg\min_{\theta,\Theta} \ \sum_i \mathcal{L}(f(\Phi(x^{(i)};\Theta);\theta), y^{(i)})$$

Since the function $\Phi(\cdot)$ is nonlinear, this optimization is more difficult than the linear case above, but in practice can be solved with off-the-shelf numerical optimizers when

$\Phi(\cdot)$ is very simple (e.g., as in Eq. (2.1)).

2.1.2 Feature Hierarchies

In the preceding we allowed for a single tunable feature function $\Phi(x;\Theta)$ so that our final prediction could be computed by composition of the functions f and Φ ($\hat{y} = f(\Phi(x;\Theta);\theta)$), where both f and Φ took very simple forms (e.g., f could be linear, and Φ could be a linear function followed by an element-wise nonlinearity). In general, we can build more sophisticated functions by composing multiple "layers" of nonlinear functions in succession:

$$\hat{y} = f(\Phi^{(L)}(\Phi^{(L-1)}(\dots\Phi^{(1)}(x;\Theta^{(1)});\dots\Theta^{(L-1)});\Theta^{(L)});\theta) \tag{2.2}$$

where L is the number of layers and $\Theta^{(1)},\dots,\Theta^{(L)}$ are the parameters for each layer of features. Intuitively, we may think of the feature functions $\Phi^{(l)}(\cdot)$ as taking in a "lower level" representation of x and converting it to a "higher level" K_l-dimensional representation that is better for making predictions. These layers form a hierarchy of increasingly sophisticated features, built from multiple non-linear operations applied to the raw input vector.

2.2 Locally Connected, Convolutional Features for Images

Most of the experiments in this thesis will involve features trained from unlabeled images. That is, each input vector $x^{(i)} \in \mathbb{R}^n$ represents a two-dimensional array of pixel intensities. For a w-by-w pixel image with c channels per pixel we have $n = w^2c$. In this setting, we can incorporate several additional assumptions to reduce computational and data requirements.

When we define features as affine functions of the input values (as in Eq. (2.1)), the number of parameters is $Kn + n$, which grows linearly in both the number of inputs and features. Typically, we aim to set K to be a constant factor multiple of n

(often called the "overcompleteness" factor). For instance, we might choose $K = 10n$ and thus the number of parameters becomes $O(n^2)$. For modest-sized images (e.g., 32-by-32 pixel color images, which gives $n = 3072$) we will often be unable to train all of the parameters effectively given the time and data available. To reduce the number of parameters (and computational cost), it is common to enforce certain constraints on the structure of the parameters Θ that exploit the known properties of images.

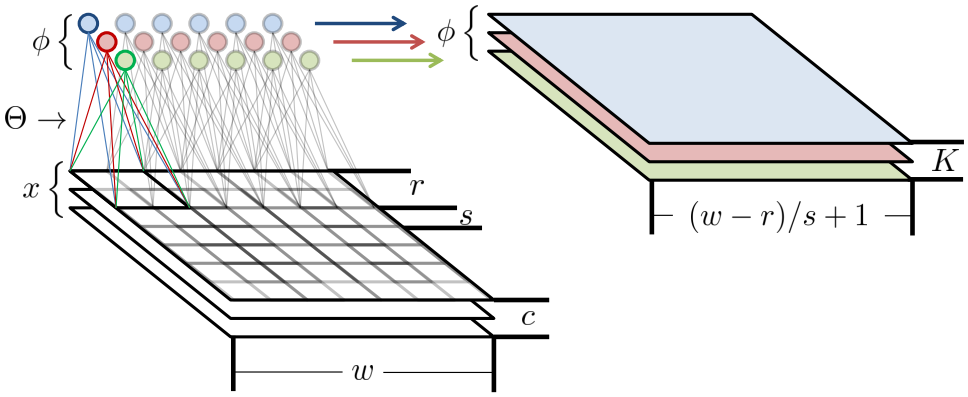


Figure 2.1: Notation and setup of locally connected, convolutional features.

First, we will reduce the total number of parameters to be trained by using *locally connected* features. Rather than allowing each feature value ϕ_k to depend on the entire input vector through $\Phi(x)$, we will restrict each ϕ_k to depend only on a subset of the inputs. Concretely, for images, we divide the image into (possibly overlapping) square regions of size r-by-r pixels, separated by s pixels each. The region size r is often called the "receptive field" size, and s the "step" size or "stride". Each feature is restricted to depend on only the inputs in one such region. This situation is depicted in Figure 2.1. For a linear parametrization Wx it is easy to see that this reduces the number of parameters from $O(n^2)$ to just $O(r^2c)$, which is a dramatic reduction in computational and data requirements in many applications. Note that when an equal number K of features are connected to each receptive field, the resulting feature vector ϕ may also be interpreted as a 3D-array. Letting $R = (w - r)/s + 1$, the resulting array has spatial dimensions R-by-R and K channels. [1] In this context, each channel

[1] We will often use K to refer to the number of features for each receptive field when we use

is known as a "map".

In addition to restricting the connectivity of the features, it is also useful to reduce the number of unique parameters by sharing parameter values across features. Concretely, suppose we wish to have K features for each r-by-r region in Figure 2.1. The result can be represented as a R-by-R-by-K array of feature values, so we have $\phi \in \mathbb{R}^{K'}$ with $K' = K \cdot R^2$. If $\Theta^{[i,j]}$ denotes the parameters learned from the (i,j)'th receptive field of the image, a common trick is to require that $\Theta^{[1,1]} = \Theta^{[1,2]} = \ldots = \Theta^{[R,R]} = \Theta$. Thus, we just train parameters Θ that get "re-used" for every receptive field. Referred to as "weight tying", this constraint reduces the number of unique features to be learned from $K \cdot R^2$ to just K. In the special case where the step size $s = 1$ pixel, this is also called a "convolutional" architecture, since the set of parameters Θ may be interpreted as a bank of filters that are convolved with the input image.

The weight-tying trick relies on the assumption that the input distribution is stationary with respect to 2D translations. That is, it relies on the (reasonable) assertion that the statistical structure of a r-by-r window of the input image is not affected by which sub-window is chosen. In this case, it is likely that parameters Θ learned for one receptive field (or from patches extracted from randomly chosen receptive fields) will yield features representing any particular receptive field just as well. This may not be true for some types of images (e.g., tightly cropped images where the borders may have different statistics than the center), but in most cases this subtlety is safely ignored.

2.3 Template for Unsupervised Feature Learning

In order to train the parameters Θ associated with the feature representations, many approaches are available. A great deal of work has focused on supervised methods that train the entire hierarchy of features directly to minimize the top-level objective [50, 51, 40, 12]. In unsupervised feature learning, the features are learned from unlabeled data—that is, we aim to train Θ using only unlabeled examples. Many

convolutional systems instead of the total number of features.

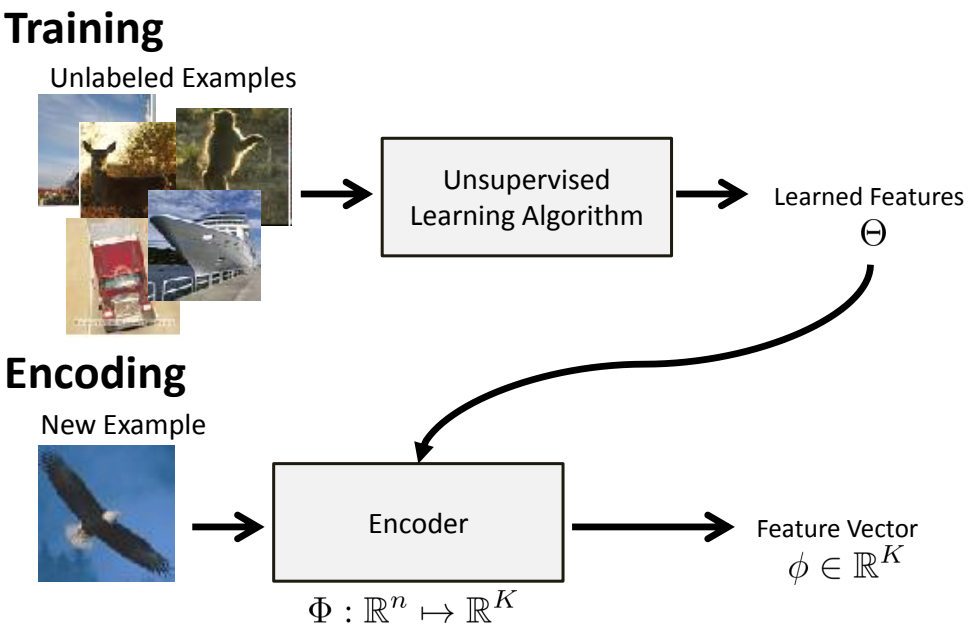


Figure 2.2: A standard template that encompasses a wide range of unsupervised feature learning algorithms.

unsupervised feature learning algorithms can be seen as instantiations of a common template wrapped around two components: (i) an unsupervised learning algorithm that trains the parameters Θ, and (ii) a definition for the feature map $\Phi(x;\Theta)$.

The basic template is pictured in Figure 2.2. A set of unlabeled examples (e.g., images) are provided to the unsupervised learning algorithm. The unsupervised learning algorithm takes in $x^{(i)}$ and outputs a trained set of parameters Θ that, in some sense, encode knowledge about the distribution of the data $x^{(i)}$. These parameters are used to define the feature functions $\Phi(x;\Theta)$, which are then applied to *labeled* data points to yield feature vectors ϕ. Along with labels y, the feature vectors may be passed to any machine learning algorithm to predict y from ϕ. Note that while $\Phi(x;\Theta)$ may often be closely related to the particular unsupervised learning method, in general it can be chosen separately as an arbitrary function of x and Θ.

2.4 Unsupervised Learning Algorithms

The first component needed to define an unsupervised feature learning system following the template in Figure 2.2 is an unsupervised learning algorithm. A few notable algorithms that we will use later in this thesis are reviewed here, though we will also introduce a few other choices later.

2.4.1 Clustering

Mixture of Gaussians

A simple form of unsupervised learning algorithm is clustering, which posits that the data is generated from a mixture model with K components:

$$P(x) = \sum_{k=1}^{K} \pi_k P_k(x; \Theta_k)$$

where π_k are the prior probabilities of x being sampled from each component, and P_k are component distributions parametrized by Θ_k. A common case is the Gaussian mixture model (GMM) where $P(x; \Theta_k) = \mathcal{N}(x; \mu_k, \Sigma_k)$ for mean and covariance parameters μ_k and Σ_k. For a given dataset, all of the parameters in this probabilistic model, $\Theta = (\mu, \Sigma, \pi)$, can be estimated using the Expectation-Maximization algorithm [21].

K-means: Euclidean Distance

K-means clustering is a simple algorithm that seeks cluster centers $D^{(k)} \in \mathbb{R}^n, k = 1, \ldots, K$ and assignments $C^{(i)} \in \{1, 2, \ldots, K\}$ of the training samples $x^{(i)}$ to clusters that minimize the distance between data points and their cluster centers. Specifically, we find D and C by minimizing:

$$\underset{C,D}{\text{minimize}} \sum_i ||D^{(C^{(i)})} - x^{(i)}||_2^2$$

which is accomplished by an alternating optimization over C and D:

$$\begin{aligned}&\text{Repeat until convergence:}\\ &C := \sum_i ||D^{(C^{(i)})} - x^{(i)}||_2^2\\ &D := \arg\min_{\hat{D}} \sum_i ||D^{(C^{(i)})} - x^{(i)}||_2^2.\end{aligned}$$

This may be simplified to:

$$\begin{aligned}&\text{Repeat until convergence:}\\ &C^{(i)} := \arg\min_k ||D^{(k)} - x^{(i)}||_2^2, \forall i\\ &D^{(k)} := \frac{\sum_i \mathbb{1}\{C^{(i)} == k\} x^{(i)}}{\sum_i \mathbb{1}\{C^{(i)} == k\}}, \forall k\end{aligned}$$

where the last line is simply the arithmetic mean of the samples $x^{(i)}$ assigned to the k'th cluster. The main advantage of K-means clustering over more sophisticated models is speed and simplicity: the computations above are easy to implement, easy to parallelize (over examples), and do not require any parameter tuning.

K-means: Spherical

A variant of K-means that we will use is "spherical" K-means, sometimes also called "gain-shape vector quantization" [23, 101]. This algorithm minimizes:

$$\begin{aligned}\underset{D,z}{\text{minimize}} \quad & \sum_i ||Dz^{(i)} - x^{(i)}||_2^2 \\ \text{subject to} \quad & ||D^{(k)}||_2 = 1, \forall k \\ \text{and} \quad & ||z^{(i)}||_0 \leq 1, \forall i\end{aligned} \tag{2.3}$$

where the vectors $z^{(i)} \in \mathbb{R}^K$ are called "code vectors", and $||z^{(i)}||_0 \leq 1$ means that each code vector may have at most a single non-zero element. This may similarly be

optimized by an alternating iteration over z and D, given by:

$$\begin{aligned}
&\text{Repeat until convergence:} \\
&z_k^{(i)} := \begin{cases} D^{(k)\top} x^{(i)} & \text{if } k == \arg\max_j |D^{(j)\top} x^{(i)}| \\ 0 & \text{otherwise} \end{cases} \quad \forall i, k \\
&D := XZ^\top \\
&D^{(k)} := D^{(k)} / ||D^{(k)}||_2.
\end{aligned}$$

In the above algorithm Z represents the column-wise concatenation of the code vectors $z^{(i)}$, and similarly for X. In short, this iteration determines which element of each $z^{(i)}$ should be non-zero to minimize Eq. (2.3) by a brute-force check, then sets each $z^{(i)}$ to its optimal value. The optimal (normalized) choice of each $D^{(k)}$ is then computed holding the $z^{(i)}$ fixed.

2.4.2 Orthogonal Matching Pursuit

We can generalize the optimization problem for spherical K-means above to allow multiple non-zero entries in $z^{(i)}$. This enables each code vector to represent more complex patterns. Specifically, we would like to solve:

$$\begin{aligned}
\underset{D,z}{\text{minimize}} \quad & \sum_i ||Dz^{(i)} - x^{(i)}||_2^2 \\
\text{subject to} \quad & ||D^{(k)}||_2 = 1, \forall k \\
\text{and} \quad & ||z^{(i)}||_0 \leq \Lambda, \forall i
\end{aligned} \tag{2.4}$$

where again $||z^{(i)}||_0$ is the number of non-zero elements in $z^{(i)}$, and Λ is the largest number of non-zero elements allowed to represent each $x^{(i)}$. Unfortunately, this optimization is difficult since the constraint is non-convex and we cannot search over all choices for the non-zero entries of $z^{(i)}$ as with the spherical K-means algorithm above.

In order to perform an alternating optimization like the one used for K-means,

we can compute the code vectors $z^{(i)}$ approximately using Orthogonal Matching Pursuit [65, 5] to compute codes with at most Λ non-zeros (which we refer to as "OMP-Λ"). For a single input $x^{(i)}$, OMP-Λ begins with $z^{(i)} = 0$ and at each iteration greedily selects an element of $z^{(i)}$ to be made non-zero to minimize the remaining reconstruction error. After each selection, $z^{(i)}$ is updated to minimize $||Dz^{(i)} - x^{(i)}||_2^2$ over $z^{(i)}$ allowing only the selected elements to be non-zero. Note that OMP-1 is just spherical K-means.

2.4.3 Sparse Coding

Sparse Coding is a neurologically-inspired algorithm [62] that encourages the vectors $z^{(i)}$ to be sparse but does not have a hard limit on the number of non-zero entries. Specifically, sparse coding solves the following minimization problem:

$$\begin{aligned} \underset{D,z}{\text{minimize}} \quad & \sum_{i=1}^{m} ||Dz^{(i)} - x^{(i)}||_2^2 + \lambda ||z^{(i)}||_1 \\ \text{subject to} \quad & ||D^{(k)}||_2^2 \leq 1, \forall k. \end{aligned}$$

This problem is similar to the minimization for spherical K-means, but the requirement for $z^{(i)}$ to have a single non-zero element has been replaced by a penalty on the 1-norm ($||\cdot||_1$) of $z^{(i)}$. Such penalties encourage $z^{(i)}$ to have many zero entries (roughly controlled by the penalty weight λ).

Holding D fixed in the optimization above results in m separate optimization problems over the m code vectors $z^{(i)}$. These are convex optimization problems and can be solved efficiently [83, 24]. Meanwhile, holding all of the $z^{(i)}$ fixed, the optimization over D is convex and can be solved quickly. This motivates an alternating

algorithm to estimate D:

$$\begin{aligned} &\text{Repeat until convergence:} \\ &z^{(i)} := \arg\min_{z} \; ||Dz - x^{(i)}||_2^2 + \lambda||z||_1, \forall i \\ &D := \arg\min_{\hat{D}} \; ||\hat{D}Z - X||_{\mathcal{F}}^2 \\ &\qquad \text{subject to} \;\; ||\hat{D}^{(k)}||_2^2 \leq 1. \end{aligned} \tag{2.5}$$

Note that unlike K-means, which assigns each sample to a single cluster, sparse coding uses a *distributed* representation that represents $x^{(i)}$ in terms of multiple basis vectors from D while still requiring the code vector $z^{(i)}$ to be parsimonius (i.e., sparse). The main disadvantage of this approach is that we must solve for every example the optimizations in Eq. 2.5. Specialized algorithms [52, 95], and optimized solvers are available for these problems though the expense is still extremely high compared to simpler methods like K-means clustering.

2.4.4 Auto-encoders

All of the algorithms above may be seen as minimizing the distortion incurred by representing each $x^{(i)}$ by the corresponding code vector $z^{(i)}$, subject to the $z^{(i)}$ satisfying a constraint that requires it to be "simpler" than the original $x^{(i)}$ (namely, by requiring all but a few elements to be zero). In each case we might need to actually solve another optimization problem in order to find the code vector $z^{(i)}$ at each iteration, which is potentially expensive.

An auto-encoder is a type of neural network that may be seen as learning a very simple function to compute $z^{(i)}$ rapidly, but with similar requirements that the $z^{(i)}$ be sparse and yield a good reconstruction of the original input $x^{(i)}$. For instance, a simple "single layer" auto-encoder, as depicted in Figure 2.3, computes $z = g(W^{(1)}x + b^{(1)})$ for a scalar nonlinearity $g(\cdot)$, with trainable parameters $W^{(1)}$ and $b^{(1)}$. The output layer $\hat{x}$ is then computed by another affine transformation $\hat{x} = W^{(2)}z + b^{(2)}$. The parameters $W^{(\cdot)}, b^{(\cdot)}$ are then trained to ensure good reconstruction of x after encoding to z and

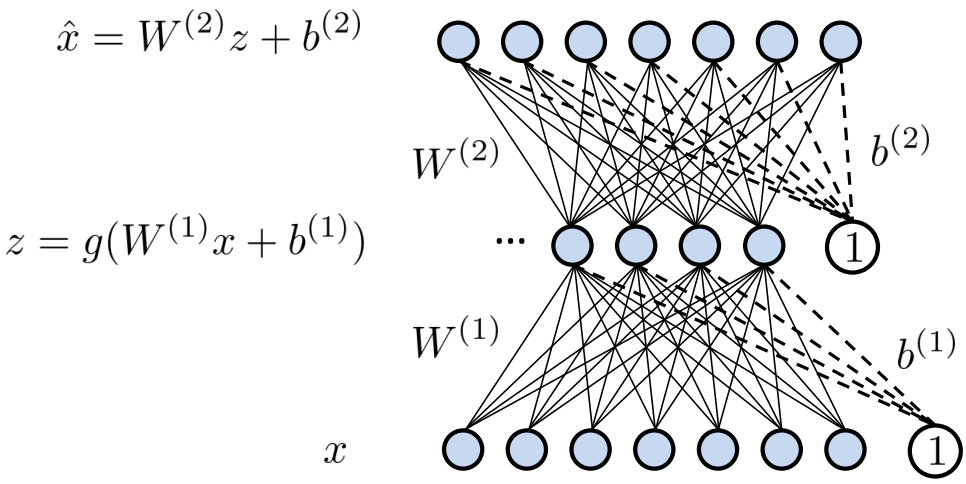


Figure 2.3: Diagram of a 1-layer auto-encoder.

then decoding to $\hat{x}$:

$$\underset{W^{(\cdot)},b^{(\cdot)}}{\text{minimize}} \sum_i ||W^{(2)}g(W^{(1)}x^{(i)} + b^{(1)}) + b^{(2)} - x^{(i)}||_2^2$$

The optimization above usually requires some form of regularization to avoid trivial or degenerate solutions, and these often depend on the choice of nonlinearity $g(\cdot)$. A popular choice of regularization is sparsity, where we penalize $z = g(W^{(1)}x + b^{(1)})$ so that its activations tend to have mostly small values with a few large values. When $g(\cdot)$ is the logistic sigmoid $g(a) = (1 + \exp(-a))^{-1}$, one such penalty is:

$$\mathcal{S}(z) = \sum_k \text{KL}(\text{Ber}(z_k)||\text{Ber}(\gamma))$$

where $\text{KL}(A||B)$ is the Kullback-Leibler divergence between distributions A and B, $\text{Ber}(p)$ is a Bernoulli distribution with mean parameter p, and γ is a tunable hyperparameter. In practice, this smooth penalty tends to be easier to work with when using sigmoid units than alternatives (e.g., the 1-norm penalty used by sparse coding).

A second form of regularization is to tie together the weights $W^{(1)}$ for the "encoding" layer and $W^{(2)}$ for the "decoding" layer. Specifically, we often use $W^{(1)} = W^{(2)\top} = W$. This tends to encourage W to have columns that are more orthogonal

and discourages arbitrary re-scaling of the weights (e.g., choosing $W^{(1)}$ very small and $W^{(2)}$ very large).

Combining these two forms of regularization, training a typical auto-encoder requires solving the following minimization:

$$\underset{W,b^{(1)},b^{(2)}}{\text{minimize}} \sum_i ||W^\top g(Wx^{(i)} + b^{(1)}) + b^{(2)} - x^{(i)}||_2^2 + \lambda \mathcal{S}(g(Wx^{(i)} + b^{(1)})).$$

This optimization can be performed with off-the-shelf numerical solvers since the network is very simple (using only a single nonlinear layer). The needed gradients my be computed quickly via the back-propagation [76] algorithm. Though this approach can be tricky to get working, it has the advantage that it learns "distributed" representations much like sparse coding, but has a very fast way to compute $z^{(i)}$ for a new input.

2.5 Feature Encodings

The second component required in the template of Figure 2.2 is a feature function $\Phi(x; \Theta)$ that maps an input $x \in \mathbb{R}^n$ to a new representation $\phi \in \mathbb{R}^K$. For many choices of unsupervised learning algorithm, a natural choice of Φ is clear. For instance, for all of the unsupervised learning algorithms above where we defined a "code vector" z corresponding to each x, we can simply take $\Phi(x; \Theta) = z$. For example, with the spherical K-means algorithm, we might choose

$$\Phi(x; D) = \begin{cases} D^{(k)\top} x & \text{if } k == \arg\max_j \ D^{(j)\top} x \\ 0 & \text{otherwise.} \end{cases}$$

Though one choice of Φ might be suggested by the unsupervised learning algorithm (as above), there is no strict reason that we must use this particular Φ. In practice, we can choose Φ arbitrarily. For instance, we may choose $\Phi : \mathbb{R}^n \to \mathbb{R}^{K'}$ where K' is a different number of features from K, the number of clusters or basis vectors that might be trained by the unsupervised learning algorithm. We will consider the effect

of non-standard choices of feature functions in Chapter 3.

2.6 Summary

In this chapter we introduced the notion of feature representations as a generic way of transforming data into a form that is more useful for making predictions. Though one can, in principle, train these from labeled data directly on the end task, a major advantage of having trainable features is that we can potentially learn them from *unlabeled* data. This gives us the opportunity to acquire useful prior knowledge, in the form of the parameters Θ, from background data that makes subsequent learning tasks easier.

We introduced a practical set of building blocks from which many unsupervised feature learning systems may be built. For instance, we showed how the basic definitions of features may be restricted to "locally connected" and "convolutional" features, useful for applications to image data. In addition, we proposed a standard template that encompasses a large number of feature learning algorithms used in practice. This template allows us to build a UFL method by incorporating (i) any unsupervised learning algorithm, and (ii) any function that maps an input x to features ϕ using the learned model parameters Θ. The rest of this thesis explores algorithms built from these ingredients, seeking the key factors that separate high-performing systems from low-performing ones, and aiming to extend the model in ways that leverage these insights.

Chapter 3

A Scientific Study of Unsupervised Feature Learning

Unsupervised Feature Learning (UFL) algorithms of the sort described in Chapter 2 can be constructed from a variety of different components and often have many "hyper-parameters" associated with their implementation: pre-processing stages, parameters that determine local connectivity (as in Section 2.2), the choice of unsupervised learning algorithm, the number of features to learn, and the feature mapping $\Phi(x; \Theta)$. Though there are many examples of successful applications in the literature, there is not much guidance for how to choose these many parameters. Indeed, many results in the past have appeared contradictory: apparently similar algorithms have achieved very different results on benchmarks while some very different algorithms end up with similar results. In this chapter we present the results of an in-depth scientific study into many of the factors that can affect performance in order to make sense of prior results and to understand what makes a UFL system work well. Surprisingly, we will find that good performance is not related to the sophistication of our learning algorithms—instead, other (sometimes neglected) factors turn out to be much more important.

3.1 Related Work

One area of feature learning that has been thoroughly studied is the choice of unsupervised learning algorithm. Since the introduction of unsupervised pre-training [31], many new schemes for stacking layers of features to build "deep" representations have been proposed. Most have focused on creating new training algorithms to build single-layer models that are composed to build deeper structures. Among the algorithms considered in the literature are sparse-coding [62, 52, 96], RBMs [31, 45], sparse RBMs [53], sparse auto-encoders [28, 71], denoising auto-encoders [89], "factored" [70] and mean-covariance [69] RBMs, as well as many others [68, 54, 99]. Thus, amongst the many components of feature learning architectures, the unsupervised learning module appears to be the most heavily scrutinized.

On top of the many models proposed, some consensus has emerged on which combinations of algorithms are fastest or which tend to yield the highest performance on benchmarks. For example, K-means clustering is extremely fast and widely used in computer vision, yet sparse coding has been shown to yield consistently better features [6, 42, 96]. Thus, fast algorithms and approximations have been devised to make sparse coding more practical on large problems [30, 95]. Other authors, however, have chosen to dissect algorithms like sparse coding in search of its strengths and weaknesses, with an eye toward developing new encodings. For example, [99] have argued for "locality preserving" encodings based on the idea that such encodings allow higher-level systems to learn functions across the data manifold more easily, and they show that this method is often superior to sparse coding. Yet many of these previous results do not carefully control for other differences between implementations, or attempt to decompose the system in a way that clearly shows which parts of the pipeline yield higher performance. In this chapter, we will try to remedy this by a more careful set of controlled experiments aimed at understanding exactly which algorithms perform best and why. We note that some of the basic results of these experiments have appeared in more limited experiments; for instance, it has been shown several times that model parameters Θ trained with K-means, but combined with non-standard encodings Φ can yield high performance [90, 87].

Beyond the study of learning algorithms and feature encodings, some work has considered the impact of other choices in these feature learning systems, especially the choice of network architecture. Jarret et al. [40], for instance, have considered the impact of changes to the "pooling" strategies frequently employed between layers of features, as well as different forms of normalization and rectification between layers. Similarly, Boureau et al. have considered the impact of coding strategies and different types of pooling, both in practice [6] and in theory [7]. Our experiments follow in this vein, but consider instead the structure of single-layer networks—before feature pooling, and orthogonal to the choice of algorithm or feature encoding.

of our experiments is very closely related to "visual words" models used in computer vision. Specifically, the K-means clustering algorithm is used to learn a set of centroids that are then used to map inputs into a new feature space. For instance, in the "bag of words" and spatial pyramid models [19, 47, 26, 94] it is typical to map an input x to a 1-of-K coded feature vector (i.e., one uses a "Hard Assignment" encoding for $\Phi(x;\Theta)$, as detailed in Section 3.2.1). This kind of quantization makes K-means learning very fast, but results in crude features. Thus, other authors have used "soft" assignments (e.g., Gaussian activations) to improve performance of the encoding stage [87]. Similarly, these types of features have been trained recursively to construct multiple layers of features [2]. The effects of pooling and choice of activation function or coding scheme have similarly been studied for these models [47, 87, 60]. The primary distinction that should be drawn with the experiments here is that we use a much wider array of pipeline combinations in a large battery of controlled experiments: we use many different unsupervised learning algorithms and feature encodings, and consider the effect of changes to our image processing pipeline across algorithms in a deliberate search for the specific changes that are *consistently* related to high recognition performance across datasets and across learning schemes.

3.2 Standard Feature Learning Framework for Image Recognition

To study the effect of various parameters, we will construct a standard pipeline for learning image features using the basic concepts in Chapter 2. The pipeline will enable us to easily alter various parameters, including the choice of unsupervised learning algorithm and feature mapping, in a systematic way to test the effect of each on (supervised) image recognition benchmarks. In this chapter all of our experiments will be performed with a single layer of learned features and a fixed post-processing stage (feature pooling) in order to simplify the experiments. We will consider training multiple layers of features in later chapters.

At a high-level, our system performs the following steps to learn a feature representation from a set of unlabeled images:

1. Extract small patches from random locations in unlabeled training images.
2. Apply a pre-processing stage to the patches.
3. Train model parameters Θ from these patches using one of several unsupervised learning algorithms.

Given the learned feature parameters and a set of labeled training images we can then perform feature extraction and classification:

1. Define a feature encoding $\Phi(x; \Theta)$ that maps a patch x to a feature vector ϕ.
2. Extract features using Φ from equally spaced sub-patches covering the input image.
3. Pool features together over regions of the input image to reduce the total number of feature values.
4. Train a linear classifier to predict image labels given the pooled feature vectors.

This pipeline is simply an extension of the basic feature learning framework described in Chapter 2 (see Figure 2.1) with both local connectivity and weight tying. That is,

features are evaluated for small patches, rather than over the whole image and the same parameters Θ are used to extract features from many patches within a larger image. A few additional steps were added to the basic recipe that are specific to the image recognition experiments we will use as a benchmark.

We will now describe the components of this pipeline and its parameters in more detail.

3.2.1 Feature Learning from Patches

As mentioned above, we begin by extracting random sub-patches from unlabeled input images. Each patch has dimension r-by-r and has c channels,[1] with r referred to as the "receptive field size". Each r-by-r patch can be represented as a vector in $\mathbb{R}^n$ of pixel intensity values, with $n = r \cdot r \cdot c$. We then construct a dataset of m randomly sampled patches, $x^{(1)}, ..., x^{(m)}$, where $x^{(i)} \in \mathbb{R}^n$. Recall that we denote by X the entire dataset concatenated column-wise into a matrix of n rows and m columns. Given this dataset, we apply the pre-processing and unsupervised learning steps.

Pre-processing

It is common practice to perform several simple normalization steps before attempting to generate features from image data. For our experiments, we assume that every patch $x^{(i)}$ is normalized by subtracting the mean and dividing by the standard deviation of its elements. Note that this corresponds to local brightness and contrast normalization within the original (larger) image.

After normalizing each input vector, the entire dataset X may optionally be whitened [39], which removes linear correlations amongst the input values. Here

[1] For example, if the input image is represented in (R,G,B) color, then it has three channels.

we will use ZCA whitening:

$$\begin{aligned}
\mu &:= \text{mean}(X) \\
\Sigma &:= \text{cov}(X) \\
\% \text{ Find V and D such that } VDV^\top &== \Sigma : \\
[V, D] &:= \text{eig}(\Sigma) \\
x^{(i)}_{\text{centered}} &:= x^{(i)} - \mu \\
x^{(i)}_{\text{whitened}} &:= VD^{-1/2}V^\top x^{(i)}_{\text{centered}}.
\end{aligned}$$

We then use $x^{(i)}_{\text{whitened}}$ as the input to the rest of our pipeline, though we will continue to denote it merely as $x^{(i)}$. While this pre-process is commonly used in deep learning work (e.g., [70]) it is less frequently employed in computer vision.

Unsupervised learning

After pre-processing, an unsupervised learning algorithm is applied to train a model, parameterized by Θ, of the unlabeled patches. For the purposes of the experiments in this chapter, we will view the unsupervised learning module as a function that takes in the unlabeled patches X and outputs a trained set of model parameters Θ. These parameters are used (however we like) to define the feature mapping Φ. We briefly summarize the unsupervised learning algorithms that we will use in this chapter, along with a few notes about their specific implementation and a description of the parameters Θ that they produce. The first of these were introduced in Section 2.4; several additional algorithms are briefly described here as well.

1. **Gaussian mixture model (GMM):** We estimate a Gaussian mixture model with K components from the unlabeled patches X. As is common practice, we run a single iteration of K-means to initialize the means of the mixture model.[2] The algorithm outputs parameters $\Theta = (\mu_\cdot, \Sigma_\cdot, \pi)$ that represent the means,

[2]When K-means is run to convergence we have found that the mixture model does not lead to feature representations substantially different from the K-means result.

covariances and prior probabilities, respectively, of each mixture component. In all of our experiments we restrict the covariance matrices $\Sigma_\cdot$ to be diagonal.

2. **K-means clustering:** In our experiments we will use both variants of K-means (Euclidean and spherical) introduced in Section 2.4.1. In either case the algorithm outputs $\Theta = (D)$, where D is the matrix whose columns $D^{(k)}, k = 1, \ldots, K$ are the centroids learned by K-means. For spherical K-means, we have $||D^{(k)}||_2 = 1$.

3. **Sparse coding (SC):** A dictionary of basis vectors is trained as in Section 2.4.3. The algorithm outputs $\Theta = (D)$, where D is the matrix whose columns $D^{(k)}$ are the K learned basis vectors. Our implementation renormalizes the basis vectors $D^{(k)}$ to unit length after each optimization over D. This step tends to prevent individual basis vectors from shrinking to zero (a problem analogous to empty clusters in K-means). We use the coordinate descent algorithm [95] to solve for the sparse codes (implemented by [56]) and a standard least-squares solver to compute D given the matrix of codes Z.

4. **Orthogonal matching pursuit (OMP-Λ):** We use orthogonal matching pursuit as in Section 2.4.2 to train a dictionary $\Theta = (D)$, similarly to sparse coding. As with sparse coding and K-means, we normalize the basis vectors $D^{(k)}$ to unit length after each iteration to prevent basis vectors from shrinking to zero. The implementation of [56] is used to compute the OMP code vectors. Note again that OMP-1 is equivalent to spherical K-means.

5. **Sparse auto-encoder (SAE):** We train an auto-encoder with K hidden nodes using back-propagation to minimize squared reconstruction error. As in Section 2.4.4, we use an additional penalty term that encourages the units to maintain a low average activation [53, 28] (sparsity) and tied encoding/decoding weights. The training algorithm outputs $\Theta = (W, b^{(1)}, b^{(2)})$, where $W \in \mathbb{R}^{K \times n}$ is the matrix of weights and $b^{(1)} \in \mathbb{R}^K, b^{(2)} \in \mathbb{R}^n$ are the biases for the encoding and decoding layers, respectively.

   There are several hyper-parameters used by the training algorithm (e.g., the

target activation γ). These parameters were chosen using a cross-validation procedure for each of our benchmarks. A separate cross-validation run is used for each choice of the receptive field size, r (the dimension of the unlabeled image patches).[3]

6. **Sparse restricted Boltzmann machine (SRBM):** The restricted Boltzmann machine (RBM) is an undirected graphical model with K binary hidden variables. Sparse RBMs can be trained using the contrastive divergence approximation [32] with the same type of sparsity penalty as the auto-encoders. The training also produces weights W and biases b. As above, the necessary hyper-parameters for the learning algorithm are determined by cross-validation for each receptive field size.

7. **Randomly sampled patches (RP):** As a baseline unsupervised learning algorithm, we represent the input data $x^{(i)}$ with a set of randomly chosen exemplars. Specifically, this algorithm generates $\Theta = (D)$ where the columns of D are normalized vectors sampled randomly from amongst the $x^{(i)}$.

8. **Random weights (R):** It has also been shown that completely random values used in place of learned parameters can perform surprisingly well in some tasks [40, 80]. Thus, we have also tried using $\Theta = (D)$, where the columns of the dictionary $D \in \mathbb{R}^{K \times n}$ are vectors sampled from a unit normal distribution, normalized to unit length.

Feature Encodings

After running an unsupervised learning algorithm to train model parameters Θ from the unlabeled patches X, we must define a mapping from a new patch x to a corresponding feature vector ϕ. As explained in Section 2.5 this function may be chosen arbitrarily, though there is usually a "natural" choice of mapping for a particular unsupervised learning algorithm. Here we summarize for concreteness all of the feature

[3] Ideally, we would perform this cross-validation for every choice of parameters, but the expense is prohibitive for the number of experiments we perform here.

mappings $\Phi(x;\Theta)$ that will be used in our experiments. The first of these correspond to the "natural" encodings for the unsupervised learning algorithms above, but the last few are chosen without reference to a particular learning algorithm.

1. **Soft assignment (SA)** Given means and covariances $\Theta = (\mu_\cdot, \Sigma_\cdot)$, the feature vector for x is defined as:

$$\phi_k = \frac{1}{(2\pi)^{n/2}|\Sigma_k|^{1/2}} \exp\left(-\frac{1}{2}(x-\mu_k)^\top \Sigma_k^{-1}(x-\mu_k)\right).$$

   We call this the "natural" encoding for the mixture of Gaussian model since each feature is the probability that x was generated by component k given that it belongs to cluster k. This approach has been used frequently in prior art [87, 2, 6].

2. **Hard assignment:** Given centroids $D^{(k)}$, the feature vector for x is defined as:

$$\phi_k = \begin{cases} 1 & \text{if } k = \arg\min_j \ ||D^{(j)} - x||_2^2 \\ 0 & \text{otherwise.} \end{cases}$$

   This is the natural encoding for K-means with Euclidean distance, used frequently in previous work [19, 94, 47].

3. **Sparse coding (SC):** Given a learned dictionary D from one of the above algorithms, we solve for the sparse code $z \in \mathbb{R}^K$ for x by minimizing (2.5) with D fixed. Note that the choice of λ in this case may be different from that used during training. We then take:

$$\phi_k = \max\{0, z_k\}$$
$$\phi_{k+n} = \max\{0, -z_k\}$$

That is, we split the positive and negative components of the sparse code z into separate features. Note that this results in $\phi \in \mathbb{R}^{2K}$. This allows the higher-level parts of the system (i.e., the classifier) to weight positive and negative responses differently if necessary.[4]

4. **Orthogonal matching pursuit (OMP-Λ):** As above, we compute z given x and D using OMP-Λ to yield at most Λ non-zeros. Given z, the features $\phi \in \mathbb{R}^{2K}$ are defined as for sparse coding above. In the special case that $\Lambda = 1$, z will have just one non-zero element (equal to $D^{(k)\top}x$, for one choice of k). This is the "one hot" encoding, which is the natural encoding for Spherical K-means (equivalent to OMP-1).

5. **Sigmoid (S)** For parameters $\Theta = (W, b^{(1)})$ learned by auto-encoder or RBM training, we use the hidden unit responses as features:

$$\begin{aligned} \phi_k &= g(W^{(k)}x + b_k^{(1)}) \\ \phi_{k+d} &= g(-W^{(k)}x + b_k^{(1)}) \quad \% \text{ Optional} \end{aligned}$$

where $W^{(k)}$ is the k'th row of W, and g is the logistic sigmoid function. Note that we allow for a "two-sided" encoding here so that the Sigmoid encoder is not at a disadvantage compared to sparse coding during some of our later experiments.

6. **Triangle:** Given centroids D, we define ϕ by:

$$\phi_k = \max\{0, \text{mean}(v) - v_k\}$$

where $v_k = ||x - D^{(k)}||_2$ and $\text{mean}(v)$ is the mean of the elements of v. This activation function outputs 0 for any feature ϕ_k where the distance from x to the centroid $D^{(k)}$ is "above average". In practice, this means that roughly half of the features will be set to 0. This can be thought of as a very simple form of

[4]This polarity splitting has always improved performance in our experiments, and can be thought of as non-negative sparse coding with the dictionary $[-D\ D]$.

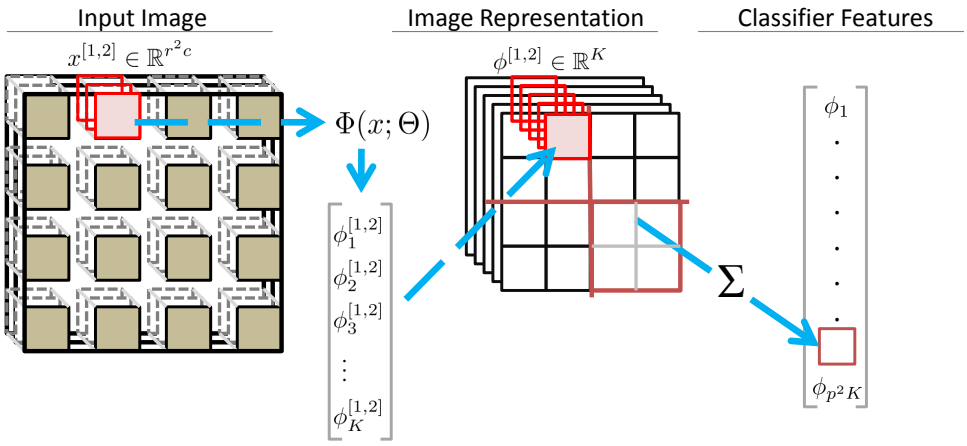


Figure 3.1: Diagram of our image recognition pipeline. We use locally connected features and convolutions as detailed in Section 2.2 (see also Figure 2.1). This yields features $\phi^{[i,j]}$ for each patch $x^{[i,j]}$, which are pooled over local (non-overlapping) regions of a p-by-p grid to yield a vector for classification. (The diagram shows $p = 2$.)

"competition" between features.

7. **Soft threshold (T):** Another very simple encoding is a feed-forward "soft threshold" with a fixed threshold α. For a dictionary D of learned basis vectors:

$$\phi_k = \max\left\{0, D^{(k)\top} x - \alpha\right\}$$
$$\phi_{k+n} = \max\left\{0, -D^{(k)\top} x - \alpha\right\}$$

This function has become popular in other feature learning architectures based on many different learning algorithms [41, 42, 58, 46], and is often referred to as a "shrinkage" function for its role in regularization and sparse coding algorithms [30].

Convolution

Given learned parameters Θ and an applicable feature encoding $\Phi : \mathbb{R}^n \mapsto \mathbb{R}^K$, we are now able to map any r-by-r pixel input patch (with c channels) to a K-dimensional feature vector ϕ. As described in Section 2.2, rather than learning different features

(a different set of parameters Θ) for each possible r-by-r sub-patch of the image, we can simply re-use the same $\Phi(x;\Theta)$ to extract features from each sub-patch. More concretely, let $x^{[i,j]}$ represent the (i,j)'th sub-patch (with c channels) of an image. As in Figure 3.1, the upper-left corner of this r-by-r pixel patch is at pixel $((i-1)s,(j-1)s)$ (for step size s pixels between patches). We apply Φ to each of these sub-patches to yield a feature vector $\phi^{[i,j]}$ for every location, then concatenate them into a $(r-w)/s$-by-$(r-w)/s$-by-K dimensional representation of the original image. This array can be interpreted as an image of reduced spatial dimension but with K channels.

Feature Pooling

In this chapter, we will use a simple "pooling" stage to reduce the features before classification. Pooling aggregates feature responses from a small spatial region in order to introduce invariance to small translations of the input image and to reduce the total number of features given to the classification stage. Such techniques are a mainstay of computer vision systems [96, 47, 19]. Here, we use average pooling over square sub-regions of the image. Specifically, given an R-by-R-by-K array of feature responses computed as above (where $R = (r-w)/s$), we reduce the representation to p-by-p-by-K responses by computing the mean over non-overlapping sub-windows of p-by-p grid for each of the K channels (see Figure 3.1).

Classifier Training

In the experiments of this chapter, we will benchmark feature representations in image recognition problems. In these problems we are given a labeled dataset $x^{(i)}, y^{(i)}, i = 1,\ldots,m$ where the $x^{(i)}$ are images and the $y^{(i)} \in \{1,\ldots,\mathcal{C}\}$ are object class labels. To perform classification, we apply the feature extraction pipeline described above to the image $x^{(i)}$ to yield the pooled feature representation, which we denote $\phi^{(i)}$ (overloading our above notation where ϕ referred to the feature representation of a single patch). Each array of pooled features is stretched into a vector and passed to our classification stage.

To perform the multi-class classification we use 1-versus-all classification with linear L2 SVMs [9]. Specifically, for class C we train a linear classifier as:

$$\underset{\theta_C}{\text{minimize}} \sum_i (\max\{0, 1 - \mathcal{Y}_C^{(i)}({\theta_C}^\top \phi^{(i)})\})^2 + \lambda ||\theta_C||_2^2$$

where $\mathcal{Y}_C^{(i)} = 1$ when $y^{(i)} == C$ and $\mathcal{Y}_C^{(i)} = -1$ otherwise. The final classification decision, is given by:

$$\hat{y}^{(i)} = f(x^{(i)}; \theta) = \arg\max_C \ \theta_C^\top \phi^{(i)}.$$

The regularization parameter λ is determined by cross-validation on training data for all of our experiments.

3.3 Experimental Results and Analysis

Using the pipeline introduced above, we have performed an extensive analysis of how various components and parameters in the system affect image recognition performance. Though recognition performance is not a general measurement of feature quality, we have found that certain key factors are almost universally associated with higher performance regardless of dataset or the exact application. In our experiments we will use several common image recognition datasets to benchmark various instantiations of the above pipeline. We will see that the trends identified in these experiments continue to hold for a realistic application (scene text recognition) in Chapter 5.

3.3.1 Effect of pipeline parameters

The pipeline described above includes many adjustable parameters. In addition to the unsupervised learning algorithm and feature encoding method needed to define a feature learning system, many of these parameters are a part of the surrounding pipeline. Specifically, we must also choose:

1. Whether to perform whitening.
2. The number of features, K, to train.
3. The step size (stride), s, between sub-patches where features are extracted.
4. The receptive field size, r.

Here we will analyze the effect of each of these parameters on the CIFAR-10 [45] and NORB [51] datasets. First, we will evaluate the effects of these parameters using cross-validation on the CIFAR-10 training set. We will then report the results achieved on both CIFAR-10 and NORB test sets using each unsupervised learning algorithm and the parameter settings that our analysis suggests is best overall (i.e., in our final results, we use the same settings for all algorithms).[5]

For our unsupervised learning algorithms we will use Gaussian mixture models trained with EM, K-means with Euclidean distance, sparse auto-encoders, and sparse RBMs. In each case we use the "natural" encoding associated with each algorithm, except for K-means where we will also use the "triangle" encoding defined in Section 6. For these experiments, we do not use the "two-sided" encoding for the auto-encoders and RBMs (i.e., we only use $\phi_k = g(Wx + b^{(1)})$, and omit $\phi_{k+n} = g(-Wx + b^{(1)})$).

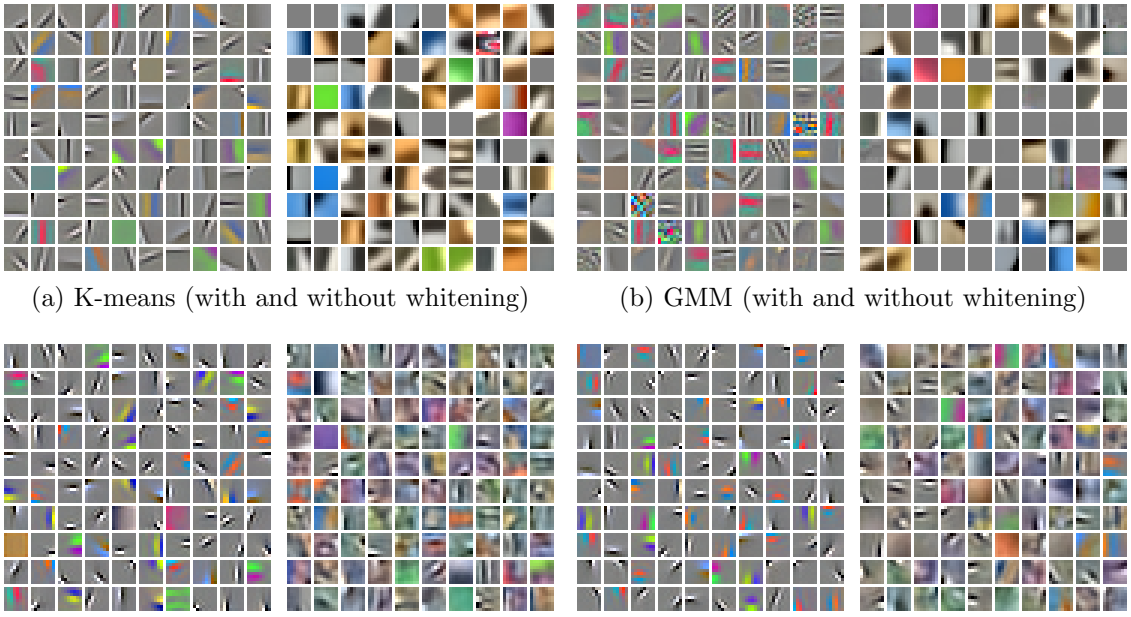
Our basic testing procedure is as follows. For each of these unsupervised learning algorithms, we will train a single layer of features using either whitened data or raw data and a choice of the parameters K, s, and r. We then train a linear classifier as described in Section 3.2.1, and finally test the classifier on a holdout set (for our main analysis) or the test set (for our final results in this section).

Feature visualization

Before we present our quantitative analysis of classification results, we first show visualizations of the learned feature representations.

The bases (or centroids) learned by sparse autoencoders, sparse RBMs, K-means, and Gaussian mixture models are shown in Figure 3.2 for 8 pixel receptive fields. It is

[5] To clarify: The parameters used in our final evaluation are those that achieved the best (average) cross-validation performance across all models: whitening, 1 pixel stride, 6 pixel receptive field, and 1600 features.



(a) K-means (with and without whitening) (b) GMM (with and without whitening)

(c) Sparse Autoencoder (with and without whitening) (d) Sparse RBM (with and without whitening)

Figure 3.2: Randomly selected bases (or centroids) trained on CIFAR-10 images using different learning algorithms. Best viewed in color.

well-known that autoencoders and RBMs yield localized filters that resemble Gabor filters and we can see this in our results both when using whitened data and, to a lesser extent, raw data. However, these visualizations also show that similar results can be achieved using clustering algorithms. In particular, while clustering raw data leads to centroids consistent with those in [26] and [88], we see that clustering whitened data yields sharply localized filters that are very similar to those learned by the other algorithms. Thus, it appears that such features are easy to learn with clustering methods (without any parameter tweaking) as a result of whitening.

Effect of whitening

We now move on to our characterization of performance on various axes of parameters, starting with the effect of whitening, which visibly changes the learned bases (or centroids) as seen in Figure 3.2. Figure 3.3 shows the performance for all of our

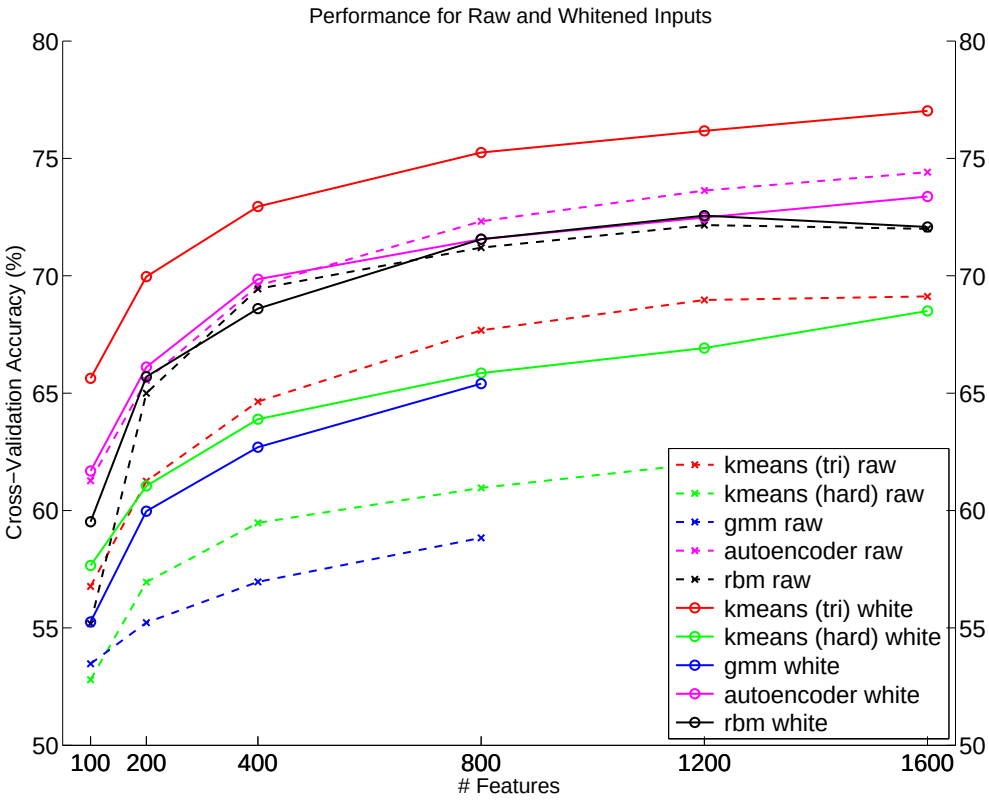


Figure 3.3: Effect of whitening and number of bases (or centroids).

algorithms as a function of the number of features (which we will discuss in the next section) both with and without whitening. These experiments used a stride of $s = 1$ pixel and $r = 6$ pixel receptive field.

For sparse autoencoders and RBMs, the effect of whitening is somewhat ambiguous. When using only 100 features, there is a significant benefit of whitening for sparse RBMs, but this advantage disappears with larger numbers of features. For the clustering algorithms, however, we see that whitening is a crucial pre-process since the clustering algorithms cannot handle the correlations in the data.[6]

Clustering algorithms have been applied successfully to raw pixel inputs in the past [26, 88] but these applications did not use whitened input data. Our results suggest that improved performance might be obtained by incorporating whitening.

[6]Our GMM implementation uses diagonal covariances and K-means uses Euclidean distance.

Number of features

Our experiments considered feature representations with 100, 200, 400, 800, 1200, and 1600 learned features.[7] Figure 3.3 clearly shows the effect of increasing the number of learned features: all algorithms generally achieved higher performance by learning more features as expected.

Surprisingly, K-means clustering coupled with the "triangle" activation function and whitening achieves the highest performance. This is particularly notable since K-means requires no tuning whatsoever, unlike the sparse auto-encoder and sparse RBMs which require us to choose several hyper-parameters to ensure reasonable results.

Effect of step size

The step size or stride s used in our framework is the spacing between patches where feature values will be extracted (see Figure 2.1). Frequently, learning systems will use a stride $s > 1$ because computing the feature mapping is very expensive. For instance, sparse coding requires us to solve an optimization problem for each such patch, which may be prohibitive for a stride of 1. It is reasonable to ask, then, how much this compromise costs in terms of performance for the algorithms we consider (which all have the property that their feature mapping can be computed extremely quickly). In this experiment, we fixed the number of features (1600) and receptive field size (6 pixels), and vary the stride over 1, 2, 4, and 8 pixels. The results are shown in Figure 3.4. (We do not report results with GMMs, since training models of this size was impractical.)

The plot shows a clear downward trend in performance with increasing step size as expected. However, the magnitude of the change is striking: for even a stride of $s = 2$, we suffer a loss of 3% or more accuracy, and for $s = 4$ we lose at least 5%. These differences can be significant in comparison to the choice of algorithm. For instance, a sparse RBM with stride of 2 performed comparably to the simple hard-assignment K-means scheme using a stride of 1—one of the simplest possible algorithms we could

[7] We found that training Gaussian mixture models with more than 800 components was often difficult and always extremely slow. Thus we only ran this algorithm with up to 800 components.

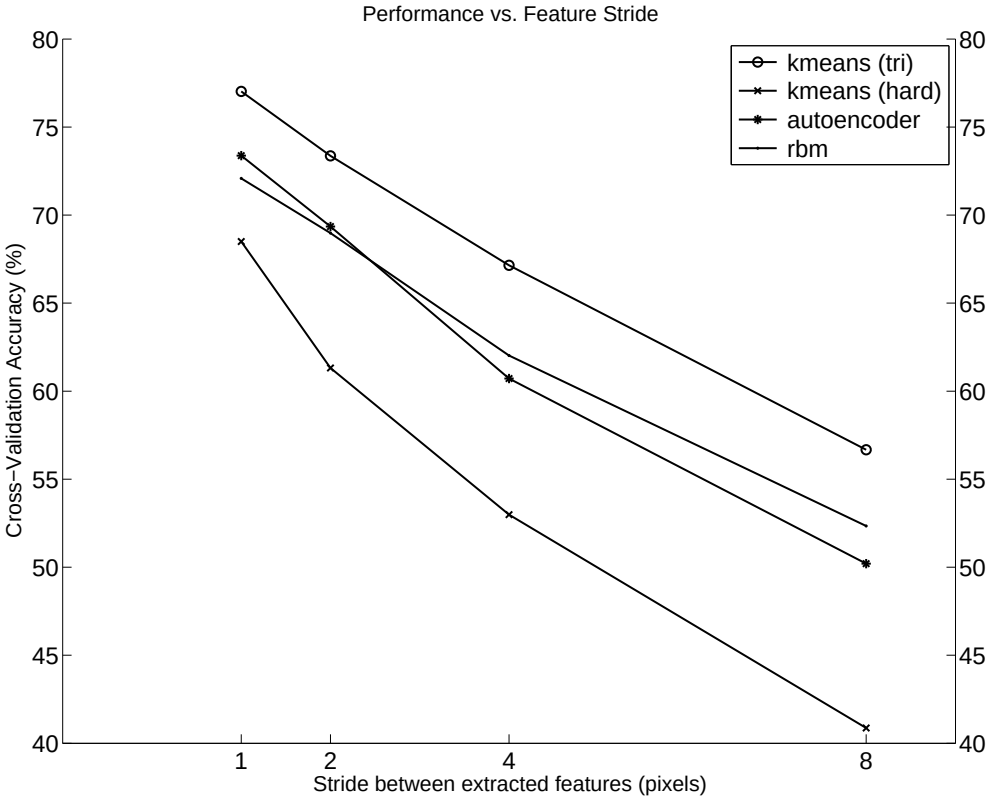


Figure 3.4: Effect of step size.

have chosen for unsupervised learning (and certainly much simpler than a sparse RBM).

Effect of receptive field size

Finally, we also evaluate the effect of receptive field size. Given a scalable algorithm, it's possible that leveraging it to learn larger receptive fields could allow us to recognize more complex features that cover a larger region of the image. On the other hand, this increases the dimensionality of the space that the algorithm must cover and may require us to learn more features or use more data. As a result, given the same amount of data and using the same number of features, it is not clear whether this is a worthwhile investment. In this experiment, we tested receptive field sizes of 6, 8,

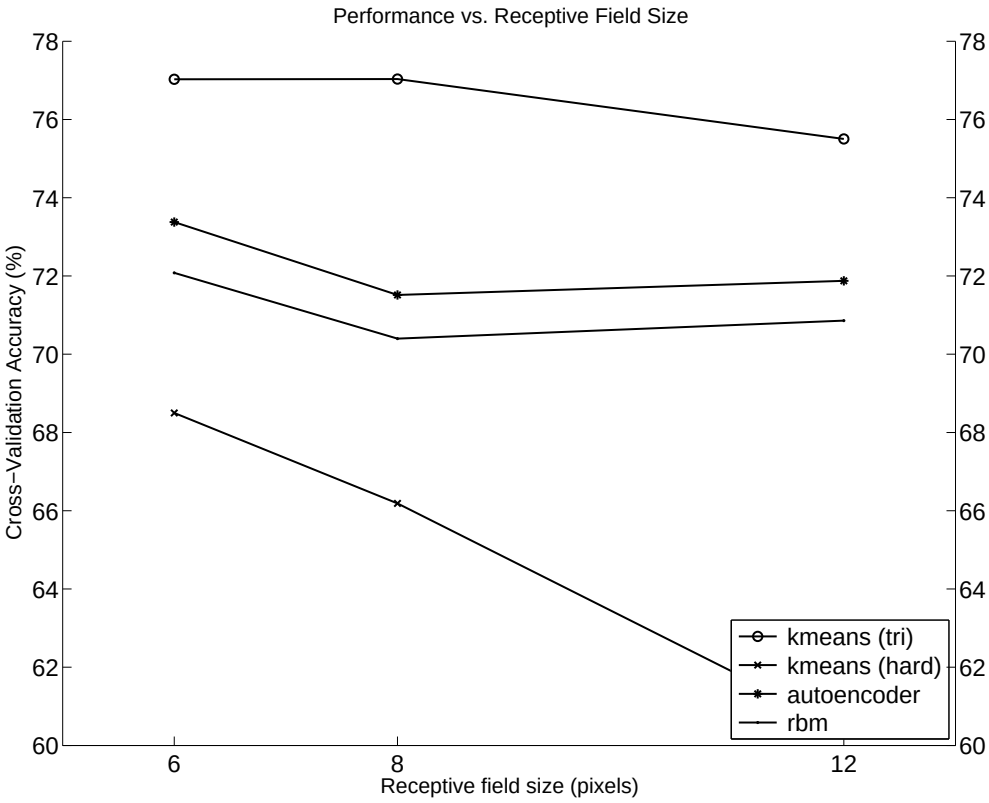


Figure 3.5: Effect of receptive field size.

and 12 pixels. For other parameters, we used whitening, stride of 1 pixel, and 1600 bases (or centroids).

The summary results are shown in Figure 3.5. Overall, the 6 pixel receptive field worked best. Meanwhile, 12 pixel receptive fields were similar or worse than 6 or 8 pixels. Thus, if we have computational resource to spare, our results suggest that it is better to spend it on reducing stride and expanding the number of learned features.

Unfortunately, unlike for the other parameters, the receptive field size does require some form of cross validation in order to make an informed choice. Our experiments do suggest, though, that even very small receptive fields can work well (with pooling) and are worth considering. This is especially important if reducing the input size allows us to use a smaller stride or more features which both have large positive impact on results. On the other hand, the proper receptive field size is related to the

Table 3.1: Test recognition accuracy on CIFAR-10

Algorithm	Accuracy
10K linear random projections (reported in [69])	36.0%
Raw pixels (reported in [45])	37.3%
GIST (384 dimension) (from [69])	54.7%
10K Gaussian RBM (2 layers) [45]	56.6%
RBM with backpropagation [45]	64.8%
3-Way Factored RBM (3 layers) [70]	65.3%
Mean-covariance RBM (3 layers) [69]	71.0%
Improved Local Coord. Coding [98]	74.5%
Conv. Deep Belief Net (2 layers) [46]	78.9%
Sparse auto-encoder	73.4%
Sparse RBM	72.4%
K-means (Hard)	68.6%
K-means (Triangle)	77.9%
K-means (Triangle, 4000 features)	**79.6%**

spatial extent of dependencies in the data—we should try to choose r so that pixel intensities more than r pixels apart tend to be almost independent. A more general way to pick receptive fields is the topic of Chapter 4.

Full test results on CIFAR-10 and NORB

We have shown that whitening, a stride of 1 pixel, a 6 pixel receptive field, and a large number of features works best on average a range of different training methods for CIFAR-10. Using these parameters we ran the full pipeline on the entire CIFAR-10 training set, trained a SVM classifier and tested on the standard CIFAR-10 test set. Our final test results on the CIFAR-10 data set with these settings are reported in Table 3.1 along with results from other prior publications. Quite surprisingly, K-means clustering (with Euclidean distance) paired with the "triangle" encoding scheme attains very high performance (77.9%) with 1600 features. Based on this success, we can improve the results further simply by increasing the number of features to 4000. Using these features, our test accuracy increases to 79.6%.

Based on the analysis above, we have also run each of these algorithms on the NORB "normalized uniform" dataset. We use all of the same parameters as for

Table 3.2: Test recognition accuracy (and error) for NORB (normalized-uniform)

Algorithm	Accuracy (error)
Conv. Neural Network [51]	93.4% (6.6%)
Deep Boltzmann Machine [78]	92.8% (7.2%)
Deep Belief Network [57]	95.0% (5.0%)
(Best result of [40])	94.4% (5.6%)
Deep neural network [85]	**97.13% (2.87%)**
Sparse auto-encoder	96.9% (3.1%)
Sparse RBM	96.2% (3.8%)
K-means (Hard)	96.9% (3.1%)
K-means (Triangle)	97.0% (3.0%)
K-means (Triangle, 4000 features)	**97.21% (2.79%)**

CIFAR-10, including the 6 pixel receptive field size. The results are summarized in Table 3.2. Here, all of the algorithms achieve very high performance. Again, K-means with the "triangle" activation achieves the highest performance. When using 4000 features as for CIFAR, we achieve 97.21% accuracy. We note, however, that the other results are very similar regardless of the algorithm used. This suggests that the main source of performance here is from our choice of network structure, not from the particular choice of unsupervised learning algorithm.

Discussion

At the time of their initial publication [16], the system above achieved the highest known performance on the benchmarks we tested. Considering the simplicity of the system compared to contemporary methods, this outcome is rather surprising—it is not clear, on first inspection, exactly what in this particular set of experiments allows us to achieve such high performance compared to prior work. We believe that the main explanation for the performance gain is, in fact, our choice of network parameters since almost all of the algorithms performed favorably relative to previous results.

Each of the network parameters (feature count, stride and receptive field size) we have tested potentially confers a significant benefit on performance. For instance, large numbers of features (regardless of how they're trained) gives us many non-linear

projections of the data. Unlike simple linear projections, which have limited representational power, it is well-known that using extremely large numbers of non-linear projections can make data closer to linearly separable and thus easier to classify. Hence, larger numbers of features may be uniformly beneficial, regardless of the training algorithm.

The dramatic impact of changes to the stride parameter may be partly explained by the work of Boureau [7]. By setting the stride small, a larger number of samples are incorporated into each pooling area which was shown both theoretically and empirically to improve results. It is also likely that high-frequency features (edges) are more accurately identified using a dense sampling.

Finally, the receptive field size, which we chose by cross-validation appears to be important as well. It appears that large receptive fields result in a space that is simply too large to cover effectively with a small number of nonlinear features. For instance, because our features often include shifted copies of edges, increasing the receptive field size also increases the amount of redundancy we can expect in our filters. This caveat might be ameliorated by training convolutionally [54, 51, 42]. Note that small receptive fields might also increase the number of samples used in pooling and thus have a small effect similar to using a smaller stride.

3.3.2 Encoding versus training

A critical design decision that must be made when constructing a feature learning system is the choice of unsupervised learning algorithm and feature encoding. Above, we evaluated the effect of altering the various "parameters" that control how features are constructed and extracted from images given a fixed choice of the training algorithm that yields parameters Θ and feature encoding function $\Phi(x; \Theta)$. In this section we take the alternate approach: we hold the pipeline parameters fixed and study the effect of altering the training and encoding methods. Specifically, we note that (i) many of the algorithms described above may be seen as a "black box" that takes in an unlabeled dataset X and learn a set of basis functions $\Theta = (D)$ (referred to variously as "weights", a "codebook", or a "dictionary"), and (ii) many encoding

Table 3.3: Cross-validation results for combinations of learning algorithms and encoders on CIFAR-10. All numbers are percent accuracy. The reported accuracies are from 5-fold cross validation on the CIFAR training set, maximizing over the choice of hyper-parameters for both the training and encoding algorithm. I.e., these are the best results we can obtain using the given combination of training and encoding algorithms if we choose the hyper-parameters to maximize the CV accuracy.

Train \ Encoder	Natural	SC	OMP 1	OMP 10	T
R	70.5	74.0	65.8	68.6	73.2
RP	76.0	76.6	70.1	71.6	78.1
RBM	74.1	76.7	69.5	72.9	78.3
SAE	72.9	76.5	68.8	71.5	76.7
SC	77.9	78.5	70.8	75.3	78.5
OMP 1	71.4	78.7	71.4	76.0	78.9
OMP 2	73.8	78.5	71.0	75.8	79.0
OMP 5	75.4	78.8	71.0	76.1	79.1
OMP 10	75.3	79.0	70.7	75.3	79.4

algorithms define a mapping from a vector x to feature vector ϕ given D agnostic to how D was trained. In this section, we exploit the ability to "mix and match" these training algorithms and encodings to analyze the contributions of each module in a controlled setting.

We will analyze the benefits of sparse coding both as a training algorithm and as an encoding strategy in comparison to Spherical K-means, Orthogonal Matching Pursuit, sparse RBMs and sparse auto-encoders, randomly sampled patches and randomly populated dictionaries. Refer again to Section 2.4 for a summary of these algorithms. For a particular choice of unsupervised learning algorithm, we will select a single compatible encoding function Φ from those in Section 3.2.1 and then benchmark the performance on CIFAR-10, Caltech-101, and NORB datasets.

Comparison on CIFAR-10

Our first and most expansive set of experiments are conducted on the CIFAR-10 dataset [45]. Here, we perform a full comparison of all of the learning and encoding algorithms. In particular, we train the dictionary with 1600 entries from whitened, 6 by 6 pixel color image patches (108-dimensional vectors), using sparse coding (SC), orthogonal matching pursuit (OMP) with $\Lambda = 1, 2, 5, 10$, sparse RBMs (RBM), sparse auto-encoders (SAE), randomly sampled image patches (RP), and random weights (R).

For each dictionary learned with the algorithms above, we then extract features not only using the "natural" encoding associated with the learning algorithm, but also with other compatible encodings from the ones described in Section 2.5. Specifically, we use sparse coding, with $\lambda \in \{0.5, 0.75, 1.0, 1.25, 1.5\}$, OMP with $\Lambda = 1, 10$, and soft thresholding (T) with $\alpha \in \{0.1, 0.25, 0.5, 1.0\}$. After computing the features for a combination of dictionary and encoding method, we construct a final feature vector by average pooling over the 4 image quadrants, yielding $4 \times 2 \times 1600 = 12800$ features. We report the best 5-fold cross-validation results, maximizing over the choice of hyper-parameters[8], for each combination of training algorithm and encoding algorithm in Table 3.3.

From these numbers, a handful of trends are readily apparent. First, we note that the first column (which pairs each learning algorithm with its standard encoder) shows that sparse coding is superior to all of the other methods by a fairly significant margin, with 77.9% accuracy. OMP-1 (Spherical K-means) is far worse (71.4%). However, we do get surprisingly close with OMP-10 and random patches. If we look at the results in the remaining columns, it becomes clear that this is not due to the learned basis functions: when using sparse coding as the activation (column 2), *all* of the dictionaries, except the Random dictionary, perform competitively. This suggests that the strength of sparse coding on CIFAR comes not from the learned

[8]Note that for sparse coding this means that the number reported for the "natural" encoding is for the best choice of λ when using the same penalty for both training and encoding. The number in the "sparse coding" column is the best performance possible when choosing *different* λ for training and encoding.

Table 3.4: Test results for some of the best systems of Table 3.3 on CIFAR-10. All numbers are percent accuracy.

Train / Encoder	Test Acc.
RP / T	79.1%
SC / SC	78.8%
SC / T	78.9%
OMP 1 / SC	78.8%
OMP 1 / T	79.4%
OMP 10 / T	80.1%
OMP 1 / T ($d = 6000$)	**81.5**%
Euclidean K-means / Triangle [16] 1600 features	77.9%
Euclidean K-means / Triangle [16] 4000 features	79.6%
Improved LCC [98]	74.5%
Conv. DBN [46]	78.9%
Deep neural net [12]	80.49%

basis functions, but primarily from the encoding mechanism.

Another striking result of these experiments is the success of the soft threshold activation function. Despite using only a feed-forward non-linearity with a *fixed* threshold, this encoding also performs uniformly well across dictionaries, and as well or even better than sparse coding.

We next take several of the best performing systems according to the cross-validation results in Table 3.3, re-train the classifiers on the full CIFAR training set and then test them on the standard test set. The final test results are reported in Table 3.4. We note several key numbers. First, using a dictionary of random patches and a soft threshold, we obtain 79.1% accuracy. This is very surprising since this algorithm requires *no training* beyond the choice of the threshold ($\alpha = 0.25$). All of the other results are similar, with just more than 1% separating them. The best overall system identified by cross-validation was OMP-10 with the soft threshold, achieving 80.1% accuracy.

In addition, we note that it is often possible to achieve better performance simply by using much larger dictionaries [87]. This is easily achieved with inexpensive training and encoding algorithms like OMP-1 and the soft-threshold. If we use a dictionary with $d = 6000$ basis vectors, we can achieve 81.5% accuracy—better than

Table 3.5: Test accuracies for the NORB jittered-cluttered dataset. All numbers are percent accuracy.

TRAIN / ENCODER	NATURAL	SC ($\lambda = 1$)	T ($\alpha = 0.5$)
R	91.9	93.8	93.1
RP	92.8	**95.0**	93.6
SC $\lambda = 1$	94.1	94.1	93.5
OMP 1	90.9	94.2	92.6
CONV.NET [81]			**94.4%**
SVM-CONV.NET [33]			94.1%
RELU RBM [58]			84.8%

those obtained with euclidean K-means in Table 3.1, and the best known result on CIFAR-10 when these results were first published [17].

Experiments on NORB

We also perform experiments on the NORB (jittered-cluttered) dataset [51]. Each 108x108 image includes 2 gray stereo channels. We resized the images to 96x96 pixels and average-pool over a 5x5 grid. We train on the first 2 folds of training data (58320 examples), and test on both folds of test data (58320 examples). Based on our experience with CIFAR, we chose fixed values for hyper-parameters for these experiments. For sparse coding, we have used $\lambda = 1.0$ and for the soft threshold $\alpha = 0.5$, though the test results are mostly insensitive to these choices.

We report test errors achieved with the natural encoder for each dictionary as well as sparse coding and the soft threshold in Table 3.5. Again we see that the soft threshold, even when coupled with randomly sampled patches, performs nearly as well as sparse coding. Though performance is slightly lower, we note that its best showing (93.6%) is achieved with far less labor: the sparse coding system requires over 7 hours to run on 40 2.26GHz cores, while the soft threshold scheme requires just 1 hour. In addition, we also see that sparse coding performs comparably regardless of which training algorithm we use. Surprisingly, when using random patches we

Table 3.6: Test results for the Caltech 101 dataset. Numbers are percent accuracy (and standard deviation) with 30 training images per class.

	SC ($\lambda = 0.15$)	T ($\alpha = 0.5$)
R	67.2% (0.8%)	66.6% (0.2%)
RP	72.6% (0.9%)	64.3% (1.2%)
SC	72.6% (0.9%)	67.7% (0.3%)
OMP 1	71.9% (0.9%)	63.2% (1.4%)
SC-SPM [96]		73.2% (0.54%)
Boureau et al., 2010 [6]		75.7% (1.1%)
Jarret et al., 2009 [40]		65.5%

achieve 95.0% accuracy—better than previously published results for this dataset. For comparison, a contemporary convolutional neural network system (using max pooling) [81] achieved 94.4% on this dataset.

Experiments on Caltech 101

Finally, we also performed experiments on the Caltech 101 dataset. For these experiments, we adopted the system of [96]. This system uses SIFT descriptors as the input to feature learning instead of raw pixels. In particular, SIFT descriptors are extracted from each image over a grid. This yields a representation of the image as a set of 128-dimensional vectors, with one descriptor representing each patch of the grid. These vectors become the inputs $x \in \mathbb{R}^{128}$ to the training and encoding algorithms and play the same role as the patches of pixels in our previous experiments.

Given the inputs $x^{(i)}$, a dictionary is constructed as before using random noise (R), randomly sampled descriptors (RP), sparse coding (SC), or spherical K-means (OMP 1). After performing the encoding, the features are pooled using max-pooling in a 3-level spatial pyramid [47] (i.e., we pool over 4x4, 2x2, and 1x1 grids). We use 30 training examples per class in our experiments, and report the average accuracy over 5 samplings of the training and test sets in Table 3.6. We use $\lambda = 0.15$ (the same used in [96]), and again $\alpha = 0.5$ for the soft threshold.

As can be seen in Table 3.6 the results are similar, though not identical to those on CIFAR and NORB. First, these results confirm that the choice of dictionary is

not especially critical: when using sparse coding as the encoder, we can use randomly sampled descriptors and achieve high performance. However, it appears that the soft threshold works less well for this dataset. One shortcoming of the soft threshold activation is the use of a constant threshold. If we instead use a variable threshold (and a dictionary trained with sparse coding), setting α dynamically to yield exactly 20 non-zeros for each example, we achieve 70.1% accuracy ($\pm$0.9%). Still a gap remains, which appears to be a result of having few training examples.

Discussion

Our experiments above emphasize primarily that the choice of encoding scheme can often be more important than how we train the parameters Θ of our features. Interestingly though, sparse coding—a very sophisticated encoder—is often no better in terms of final image recognition performance than a simple soft-threshold nonlinearity. We add a few comments here on what may be behind these particular results.

Sparse coding and small datasets First, a primary distinction between the Caltech 101 dataset and the CIFAR and NORB datasets is the number of available labeled training examples (just 30 per class for Caltech 101). In the situation where we have little labeled data regularization and prior knowledge become much more important, as we have very few labels for supervised training. It turns out that sparse coding excels in this scenario: it yields a feature vector that works well even when we have very few labels, and even when we use simple algorithms to populate the dictionary.

We have verified this phenomenon on the CIFAR and STL-10 [16] datasets. We began with a dictionary composed of random patches. We then tested the performance of the sparse-coding and soft-threshold encoders when the SVM training procedure is limited to a small number of labeled examples. For CIFAR, the average test performance over 5 folds of labeled data, for various numbers of labeled examples, is plotted in Figure 3.6. There it can be seen that the performance of sparse coding and the soft-threshold are essentially identical when we use large labeled training sets, but that

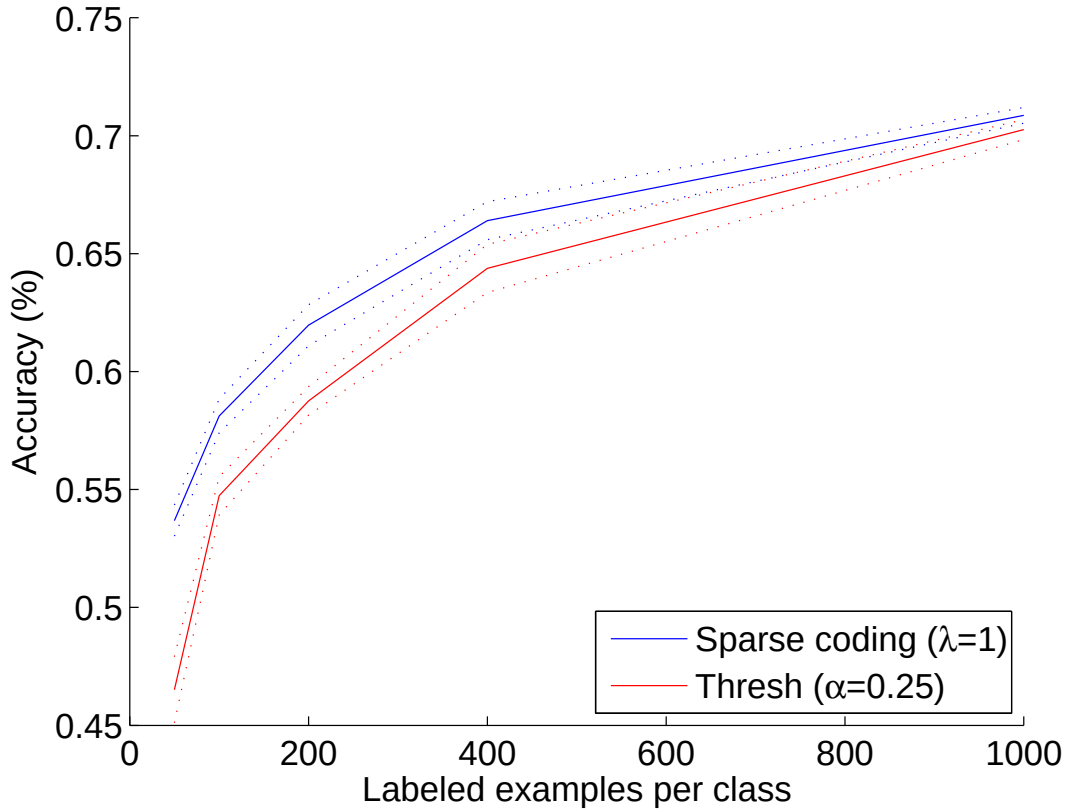


Figure 3.6: Performance of sparse coding and soft-threshold activations with small quantities of labeled data on CIFAR-10.

sparse coding performs much better for smaller numbers of examples. The STL-10 dataset similarly emphasizes smaller labeled datasets (100 examples per fold), though providing additional unlabeled data. On this dataset, the same phenomenon is apparent: on 32x32 downsampled images, sparse coding ($\lambda = 1.0$) achieves 59.0% average accuracy ($\pm 0.8\%$), while the soft-threshold ($\alpha = 0.25$) achieves 54.9% ($\pm 0.4\%$). Thus it appears that sparse coding yields a representation that is consistently better when we do not have many labeled examples, though both results are better than those previously reported in [16].

Dictionary Learning Our results have shown that the main advantage of sparse coding is as an encoder, and that the choice of basis functions has little effect on performance. Indeed, we can obtain performance on par with any of the learning algorithms tested simply by sampling random patches from the data. This indicates that the main value of the dictionary is to provide a highly overcomplete basis on which to project the data before applying an encoder, but that the exact structure of these basis functions (which comprise the bulk of the parameters that we would normally need to estimate) is less critical than the choice of encoding. All that appears necessary for success in these experiments is to choose the basis to roughly tile the space of the input data. This increases the chances that a few basis vectors will be near to an input, yielding a large activation that is useful for identifying the location of the input on the data manifold later [61, 99]. This explains why spherical K-means (OMP-1) is quite capable of competing with more complex algorithms: it simply ensures that there is at least one dictionary entry near any densely populated areas of the input space. We expect that learning is more crucial if we use small dictionaries, since we would then need to be more careful to pick basis functions that span the space of inputs equitably—in such cases, the requirement that dictionaries produce a *sparse* basis (where only a few projections $D^{(k)\top}x$ yield significant responses) may be important.

Caveats It is worth noting a couple of caveats to the analysis above. First, all of our results are based on a common pipeline that incorporates a number of "special purpose" components, including contrast normalization and feature pooling. These are necessary in order to make our end task practical, and to provide a single (and already widely used) reference point for comparison. Nevertheless, it is conceivable that some of our conclusions are influenced by the particulars of the pipeline. All of the results above are also based on a single layer of features and thus may not hold for multi-layered representations. Despite this shortcoming in our analysis we have found that this pipeline is remarkably successful in real applications (see Chapter 5 for one) and on many different benchmarks where previous authors have used many different types of architectures. More importantly still, this analysis has served as

the seed for our own later work in this area: in later chapters, we will continue to use the surprisingly versatile spherical K-means algorithm and soft-threshold encoding, and pay special attention to factors such as scalability.

3.4 Summary

In this chapter we proposed a standard image processing pipeline that covered a wide array of possible architectures similar to those used in computer vision applications [19, 96, 47, 87] and in feature learning and deep learning work [54, 46, 51]. This has allowed us to systematically test the impact of changes in the parameters and algorithms that are part of the pipeline and to discover the main areas where more attention might yield improvements in performance. Surprisingly, the main contributors to high performance are factors that are mostly unrelated to the choice of *training* algorithm—the parameters that define the feature extraction pipeline (receptive field size r, step size s, number of features K and the use of whitening), and the choice of encoding Φ are apparently more important than the exact training algorithm. These observations have served as the jumping off point for the remaining chapters of this thesis. Rather than developing new learning algorithms and more sophisticated models of data, we instead focus on scalability and on improvements to the other components of our pipeline that are apparently more crucial.

Chapter 4

Selecting Receptive Fields in Deep Networks

4.1 Introduction

An important practical concern in building multi-layered ("deep") feature representations is to specify how the features in each layer connect to the features in the layers beneath. Traditionally, the number of parameters in models for visual tasks is reduced by restricting higher level features to depend only on a "receptive field" of lower-level inputs. For instance, in Section 2.2 we described a simple heuristic to choose the connectivity for computer vision applications: each feature ϕ_k depends only on a small rectangular area within a larger image instead of allowing every feature to depend on the entire image [51, 54]. When we use linear filters to represent our features (where there is one parameter for each input pixel in the receptive field) this trick dramatically reduces the number of parameters that must be trained and is a key element of several state-of-the-art systems [12, 81, 17]. In this chapter we propose a method to automatically choose such receptive fields in situations where we do not know how to specify them by hand—a situation that, as we will explain, is commonly encountered in deep feature representations.

There are now many known results indicating that large hierarchies of features with thousands of unique feature extractors are top competitors in applications and

benchmarks (e.g., [12, 33, 81] and the results of Chapter 3). A major obstacle to scaling up feature representations further is the blowup in the number of parameters: if we aim to estimate a dictionary $D \in \mathbb{R}^{K \times n}$ using an algorithm like K-means or sparse coding, then for n input features, a complete representation with $K = n$ features requires a matrix of n^2 weights—one weight for every feature and input. This blowup leads to a number of practical problems: (i) it becomes difficult to represent, and even more difficult to update, the entire matrix D during learning, (ii) feature extraction becomes extremely slow, and (iii) many algorithms and techniques (like whitening and local contrast normalization) are difficult to generalize to large, unstructured input domains. As mentioned above, we can solve this problem by limiting the "fan in" to each feature by connecting each feature extractor to a small receptive field of inputs. In this chapter we will introduce a method that chooses these receptive fields automatically during unsupervised training of deep feature representations. The scheme can operate without prior knowledge of the underlying data and is applicable to virtually any UFL pipeline, including any of the unsupervised feature learning methods introduced previously in this thesis. In our experiments, we will show that when this method is combined with our K-means-based pipeline from Chapter 3, we can construct highly scalable systems that achieve even higher accuracy on CIFAR-10 and STL datasets.

It may not be clear yet why it is necessary to have an automated way to choose receptive fields since, after all, it is already common practice to pick receptive fields simply based on prior knowledge. However, this type of solution is insufficient for large, deep representations. For instance, in local receptive field architectures for image data, we typically train a bank of linear filters $\Theta = (D)$ using an unsupervised learning algorithm and then define features $\Phi(x; \Theta)$ to be extracted from small image patches. These features can then be extracted from sub-patches of a larger image in a convolutional manner. As an example, if we train 100 6-by-6 pixel filters and convolve them with a 32-by-32 pixel input, then we will get a 27-by-27-by-100 array of features. Each 2D grid of 27-by-27 feature responses for a single filter is frequently called a "map" [51, 12]. Though there are still spatial relationships amongst the feature values within each map, it is not clear how two features in *different* maps are

related. Thus when we train a second layer of features we must typically resort to connecting each feature to every input map or to a random subset of maps [40, 12] (though we may still take advantage of the remaining spatial organization within each map). At even higher layers of deep feature hierarchies, this problem becomes extreme: our array of responses will have very small spatial resolution (e.g., 1-by-1) yet will have a large number of maps and thus we can no longer make use of spatial receptive fields. This problem is exacerbated further when we try to use very large numbers of maps which are often necessary to achieve top performance [12, 16].

In this chapter we propose a way to address the problem of choosing receptive fields that is not only a flexible addition to feature learning pipelines, but that can scale up to the extremely large networks of features used in state-of-the-art systems. In our method we select local receptive fields that group together (pre-trained) lower-level features according to a pairwise similarity metric between features. Each receptive field is constructed using a greedy selection scheme so that it contains features that are similar according to the similarity metric. Given the learned receptive fields (groups of features) we can subsequently apply an unsupervised learning method independently over each receptive field. Using this method in conjunction with the pipeline proposed in Chapter 3, we demonstrate the ability to train multi-layered networks of features using only spherical K-means as our unsupervised learning module. All of our results are achieved without supervised fine-tuning[1], and thus rely heavily on the success of the unsupervised learning procedure. Nevertheless, this system attains performance on the CIFAR-10 and STL datasets better than the best contemporary results.

4.2 Related Work

While much work has focused on different representations for deep networks, an orthogonal line of work has investigated the effect of network structure on performance of these systems. Much of this line of inquiry has sought to identify the best choices of network parameters such as size, activation function, pooling method and so on [40,

[1]That is, we do not need to tune the parameters $\Theta^{(1)}, \Theta^{(2)}, \ldots$ jointly using a supervised objective, which has been necessary to achieve improved results for deep representations in the past.

6, 58, 81], similar to our study in Chapter 3. Through these investigations a handful of key factors have been identified that strongly influence performance (such as the type of pooling, activation function, and number of features). These studies, however, do not address the finer-grained questions of how to choose the internal structure of deep representations directly.

Other authors have tackled the problem of architecture selection more generally. One approach is to search for the best architecture. For instance, Saxe et al. [80] propose using randomly initialized networks (forgoing the expense of training) to search for a high-performing structure. Pinto et al. [66], on the other hand, use a screening procedure to choose from amongst large numbers of randomly composed networks, collecting the best performing networks.

More powerful modeling and optimization techniques have also been used for learning the structure of deep networks in-situ. For instance, Adams et al. [1] use a non-parametric Bayesian prior to jointly infer the depth and number of hidden units at each layer of a deep belief network during training. Zhang and Chan [102] use an L1 penalization scheme to zero out many of the connections in an otherwise bipartite structure. Unfortunately, these methods require optimizations that are as complex or expensive as the algorithms they augment, thus making it difficult to achieve computational gains from any architectural knowledge discovered.

In this chapter, the receptive fields will be built by analyzing the relationships between feature responses rather than relying on prior knowledge of their organization. A popular alternative solution is to impose topographic organization on the feature outputs during training. In general, these learning algorithms train a set of features (usually linear filters) such that features nearby in a pre-specified topography share certain characteristics. The Topographic ICA algorithm [37], for instance, uses a probabilistic model that implies that nearby features in the topography have correlated variances (i.e., energies). This statistical measure of similarity is motivated by empirical observations of neurons and has been used in other analytical models [82]. Similar methods can be obtained by imposing group sparsity constraints so that features within a group tend to be on or off at the same time [27, 29]. These methods

have many advantages but require us to specify a topography first, then solve a large-scale optimization problem in order to organize our features according to the given topographic layout. This will typically involve many epochs of training and repeated feature evaluations in order to succeed. Here, we perform this procedure in reverse: our features are pre-trained using whatever UFL method we like, then we will extract a useful grouping of the features post-hoc. This approach has the advantage that it can be scaled to large distributed clusters and is very generic, allowing us to potentially use different types of grouping criteria and learning strategies in the future with few changes. In that respect, part of the novelty in this approach is to convert existing notions of topography and statistical dependence in deep representations into a highly scalable "wrapper method" that can be re-used with other algorithms.

4.3 Algorithm Details

In this section we will describe our approach to selecting the connections between the feature values ϕ and their lower-level inputs x (i.e., how to "learn" the receptive field structure of the high-level features) from an arbitrary set of data based on a particular pairwise similarity metric: square-correlation of feature responses.[2] We will then explain how our method integrates with a typical learning pipeline and, in particular, how to couple our algorithm with the feature learning system proposed in Chapter 3.

In what follows, we assume that we are given a dataset X of input vectors $x^{(i)}, i \in \{1, \ldots, m\}$, with elements $x_j^{(i)}$. These vectors may be raw values (e.g., pixel values) but will usually be features computed by lower layers of a deep network.[3]

[2]Though we use this metric throughout, and propose some extensions, it can be replaced by many other choices such as the mutual information between two features.

[3]We will often refer to the elements $x_1, x_2, \ldots$ as "inputs" or "input features" to disambiguate from $\phi_1, \phi_2, \ldots$, which we refer to as "output features" or simply "features".

4.3.1 Similarity of Input Features

In order to group two input features together, we must first define a similarity metric between them. Ideally, we should group together input elements whos values are closely related (e.g., because they respond to similar patterns or tend to appear together). By putting these input features in the same receptive field, we allow their relationship to be modeled more finely by higher level learning algorithms. Meanwhile, it also makes sense to model seemingly independent subsets of input features separately, and thus we would like such inputs to end up in different receptive fields.

A number of criteria might be used to quantify this type of relationship. One popular choice is "square correlation" of feature responses, which partly underpins the Topographic ICA [37] algorithm. The idea is that if our dataset X consists of linearly uncorrelated input features (as can be obtained by applying a whitening procedure), then a measure of the higher-order dependence between two inputs can be obtained by looking at the correlation of their energies (squared responses). In particular, if we have $\mathbb{E}[x] = 0$ and $\mathbb{E}\left[xx^\top\right] = I$, then we will define the similarity between inputs x_j and x_k as the correlation between the squared responses:

$$S[x_j, x_k] = \text{corr}(x_j^2, x_k^2) = \mathbb{E}\left[x_j^2 x_k^2 - 1\right] / \sqrt{\mathbb{E}\left[x_j^4 - 1\right] \mathbb{E}\left[x_k^4 - 1\right]}.$$

This metric is easy to compute by first whitening our input dataset with ZCA[4] whitening [4], then computing the pairwise similarities between all of the input features:

$$S_{j,k} \equiv S_X[x_j, x_k] \equiv \frac{\sum_i {x_j^{(i)}}^2 {x_k^{(i)}}^2 - 1}{\sqrt{\sum_i ({x_j^{(i)}}^4 - 1) \sum_i ({x_k^{(i)}}^4 - 1)}}. \tag{4.1}$$

This computation is completely practical for fewer than 5000 input features. For fewer than 10000 inputs it is feasible but somewhat arduous: we must not only hold a 10000x10000 matrix in memory but we must also whiten our 10000-dimensional dataset—requiring a singular value or eigenvalue decomposition. We will explain

[4] If $\mathbb{E}\left[xx^\top\right] = \Sigma = VDV^\top$, ZCA whitening uses the transform $P = VD^{-1/2}V^\top$ to compute the whitened vector $\hat{x}$ as $\hat{x} = Px$.

how this expense can be avoided in Section 4.3.3, after we describe our receptive field learning procedure.

4.3.2 Selecting Local Receptive Fields

We now assume that we have available to us the matrix of pairwise similarities between input features $S_{j,k}$ computed as above. Our goal is to construct "receptive fields": sets of input features R_n, $n = 1, \ldots, N$ whose responses will become the inputs to one or more higher-level features. We would like for each R_n to contain pairs of features with large values of $S_{j,k}$. We might achieve this using various agglomerative or spectral clustering methods, but we have found that a simple greedy procedure works well: we choose one feature as a seed, and then group it with its nearest neighbors according to the similarities $S_{j,k}$. In detail, we first select N rows, $j_1, \ldots, j_N$, of the matrix S at random (corresponding to a random choice of features x_{j_n} to be the seed of each group). We then construct a receptive field R_n that contains the features x_k corresponding to the top T values of $S_{j_n,k}$. We typically use $T = 200$, though our results are not too sensitive to this parameter. Upon completion, we have N (possibly overlapping) receptive fields R_n that can be used during training of the next layer of features.

4.3.3 Approximate Similarity

Computing the similarity matrix $S_{j,k}$ using square correlation is practical for fairly large numbers of input features using the obvious procedure given above. However, if we want to learn receptive fields over huge numbers of input features (as arise, for instance, when we use hundreds or thousands of maps), we may often be unable to compute S directly. For instance, as explained above, if we use square correlation as our similarity criterion then we must perform whitening over a large number of features.

Note, however, that the greedy grouping scheme requires only N rows of the matrix. Thus, provided we can compute $S_{j,k}$ for a single pair of input features, we can avoid storing the entire matrix S. To avoid performing the whitening step for all

of the input features, we can instead perform pair-wise whitening between features. Specifically, to compute the squared correlation of x_j and x_k, we whiten the jth and kth inputs of X together (independently of all other columns), then compute the square correlation between the whitened values $\hat{x}_j$ and $\hat{x}_k$. Though this procedure is not equivalent to performing full whitening, it appears to yield effective estimates for the squared correlation between two features in practice. For instance, for a given "seed", the receptive field chosen using this approximation typically overlaps with the "true" receptive field (computed with full whitening) by 70% or more. More importantly, our final results (Section 4.4) are unchanged compared to the exact procedure.

Compared to the "brute force" computation of the similarity matrix, the approximation described above is very fast and easy to distribute across a cluster of machines. Specifically, the 2x2 ZCA whitening transform for a pair of features can be computed analytically, and thus we can express the pair-wise square correlations analytically as a function of the original inputs without having to numerically perform the whitening on all pairs. If we assume that all of the input features of $x^{(i)}$ are zero-mean and unit variance, then we have:

$$\hat{x}_j^{(i)} = \frac{1}{2}((\gamma_{jk} + \beta_{jk})x_j^{(i)} + (\gamma_{jk} - \beta_{jk})x_k^{(i)})$$
$$\hat{x}_k^{(i)} = \frac{1}{2}((\gamma_{jk} - \beta_{jk})x_j^{(i)} + (\gamma_{jk} + \beta_{jk})x_k^{(i)})$$

where $\beta_{jk} = (1-\alpha_{jk})^{-1/2}$, $\gamma_{jk} = (1+\alpha_{jk})^{-1/2}$ and α_{jk} is the correlation between x_j and x_k. Substituting $\hat{x}^{(i)}$ for $x^{(i)}$ in Equation 4.1 and expanding yields an expression for the similarity $S_{j,k}$ in terms of the pair-wise moments, $\mathbb{E}\left[x_j^4\right], \mathbb{E}\left[x_j^3 x_k\right], \mathbb{E}\left[x_j^2 x_k^2\right], \mathbb{E}\left[x_j x_k^3\right]$, and $\mathbb{E}\left[x_k^4\right]$, for each pair of features. We can compute these statistics in a single pass over the dataset, compute $S_{j,k}$, and then select the receptive fields based on the results. Many alternative methods (e.g., Topographic ICA) would require some form of distributed optimization algorithm to achieve a similar result, which requires many feed-forward and feed-back passes over the dataset. In contrast, the above method is typically about as costly as a single feed-forward pass (to compute the feature values $x^{(i)}$) and is thus very fast compared to other conceivable solutions.

4.3.4 Learning Architecture

For our experiments, we will continue using the architecture of Chapter 3, which we previously applied with success to image recognition problems. Specifically, we will use one of the top-performing systems from Section 3.3.2: spherical K-means to train the parameters $\Theta^{(l)}$ for the l'th layer of features, and a soft threshold nonlinearity for our feature encoding. In this section we will briefly review this specific system as it will be used in conjunction with our receptive field learning approach, but it should be noted that our basic method is equally applicable to many other choices of processing pipeline and unsupervised learning algorithm.

Let $x^{(i)}, i = 1, \ldots, m$ be a dataset composed of a large number of 3-channel (RGB), 6-by-6 pixel image patches extracted from random locations in unlabeled training images. Then our system from Section 3.3.2 applies the following procedure to learn a new representation of these image patches:

1. Normalize each example $x^{(i)}$ by subtracting out the mean and dividing by the norm. Apply a ZCA whitening transform to $x^{(i)}$ to yield $\hat{x}^{(i)}$.

2. Apply spherical K-means to obtain a (normalized) set of linear filters ("dictionary"), $\Theta = (\mathcal{D})$.

3. Define a mapping from the whitened input vectors $\hat{x}^{(i)}$ to output features given the dictionary $\mathcal{D}$. We use the soft threshold encoding that computes each feature $\phi_j^{(i)}$ as $\phi_j^{(i)} = \max\{0, \mathcal{D}^{(j)\top}\hat{x}^{(i)} - \alpha\}$ for a fixed threshold α.

The computed feature values for each example, $\phi^{(i)}$, become the new representation for the patch $x^{(i)}$. We then apply the feature extractor $\Phi(x; \Theta)$ to a larger image convolutionally as described in Section 3.2.1 with a stride $s = 1$.

Clearly we can modify this procedure to use choices of receptive fields other than 6-by-6 patches of images. Concretely, given a 32-by-32 pixel image, we could break the vector of 3072 pixel intensities into arbitrary overlapping subsets R_n where each R_n includes a subset of the RGB values of the whole image. Then we apply the procedure outlined above to each set of inputs R_n independently, followed by concatenating all of the extracted features. In general, if X is now any training set (not necessarily

image patches), we can define X_{R_n} as the training set X reduced to include only those inputs from one receptive field, R_n (that is, we simply discard all of the input features from X that do not correspond to features in R_n). We may then apply the feature learning and extraction methods above to each reduced dataset X_{R_n} separately, just as we would for the hand-chosen patch receptive fields used in previous work.

4.3.5 Feature Hierarchy Details

The above components, conceptually, allow us to lump together arbitrary types and quantities of data into our unlabeled training set and then automatically partition them into receptive fields in order to learn higher-level features. The automated receptive field selection can choose receptive fields that span multiple feature maps, but the receptive fields will often span only small spatial areas (since features extracted from locations far apart tend to appear nearly independent). Thus, we will also exploit spatial knowledge to enable us to use large numbers of maps rather than trying to treat the entire input as unstructured data. Note that this is mainly to reduce the expense of feature extraction and to allow us to use spatial pooling (which introduces some invariance between layers of features); the receptive field selection method itself can be applied to hundreds of thousands of inputs. We now detail the network structure used for our experiments that incorporates this structure.

First, there is little point in applying the receptive field learning method to the raw pixel layer. Thus, we use 6-by-6 pixel receptive fields with a step size $s = 1$ pixel between them for the first layer of features just as in our previous experiments. If the first layer contains K_1 maps (i.e., K_1 dictionary elements learned by K-means), then a 32-by-32 pixel color image takes on a 27-by-27-by-K_1 representation after the first layer of (convolutional) feature extraction. Second, depending on the unsupervised learning module, it can be difficult to learn features that are invariant to image transformations like translation. This is handled traditionally by incorporating "pooling" layers [6, 51]. Here we use average pooling over adjacent, disjoint 3-by-3 spatial blocks. Thus, applied to the 27-by-27-by-K_1 representation from layer 1, this yields a 9-by-9-by-K_1 pooled representation.

After extracting the 9-by-9-by-K_1 pooled representation from the first two layers, we apply our receptive field selection method. We could certainly apply the algorithm to the entire high-dimensional representation. As explained above, it is useful to retain spatial structure so that we can perform spatial pooling and convolutional feature extraction. Rather than applying our algorithm to the entire input, we apply the receptive field learning to 2-by-2 spatial regions within the 9-by-9-by-K_1 pooled representation. Thus the receptive field learning algorithm must find receptive fields to cover $2 \times 2 \times K_1$ inputs. The next layer of feature learning then operates on each receptive field within the 2-by-2 spatial regions separately. This is similar to the structure commonly employed by prior work [12, 40], but here we are able to choose receptive fields that span several feature maps in a deliberate way while also exploiting knowledge of the spatial structure.

In our experiments we will benchmark our system on image recognition datasets using $K_1 = 1600$ first layer maps and $K_2 = 3200$ second layer maps learned from $N = 32$ receptive fields. When we use three layers, we apply an additional 2-by-2 average pooling stage to the layer 2 outputs (with stride of 1) and then train $K_3 = 3200$ third layer maps (again with $N = 32$ receptive fields). To construct a final feature representation for classification, the outputs of the first and second layers of trained features are average-pooled over quadrants as in our previous results (Section 3.2.1). Thus, our first layer of features result in $1600 \times 4 = 6400$ values in the final feature vector, and our second layer of features results in $3200 \times 4 = 12800$ values. When using a third layer, we use average pooling over the entire image to yield 3200 additional feature values. As in our previous results, we combine the features from all of the pooling stages into a single long vector and use these for training and testing in a L2-SVM (Section 3.2.1).

4.4 Experimental Results

We again benchmark this method on visual recognition problems: the CIFAR-10 and STL datasets. In addition to training on the full CIFAR training set, we also provide results of our method when we use only 400 training examples per class to compare

with other single-layer results in Figure 3.6.

The CIFAR-10 examples are all 32-by-32 pixel color images. For the STL dataset, we downsample the (96 pixel) images to 32 pixels. We use the pipeline detailed in Section 4.3.4, with spherical K-means to train up to 3 layers. For each set of experiments we provide test results for 1 to 3 layers of features, where the receptive fields for the 2nd and 3rd layers of features are learned using the method of Section 4.3.2 and square-correlation for the similarity metric.

For comparison, we also provide test results in each case using several alternative receptive field choices. In particular, we have also tested architectures where we use a single receptive field ($N = 1$) where R_1 contains all of the inputs, and random receptive fields ($N = 32$) where R_n is filled according to the same algorithm as in Section 4.3.2, but where the matrix S is set to random values. The first method corresponds to the "completely connected", brute-force case described earlier, while the second is the "randomly connected" case. Note that in these cases we use the same spatial organization outlined in Section 4.3.5. For instance, the completely-connected layers are connected to all the maps within a 2-by-2 spatial window. Finally, we also provide test results using a large 1st layer representation ($K_1 = 4800$ maps) to verify that the performance gains we achieve are not merely the result of passing more projections of the data to the supervised classification stage.

4.4.1 Comparison on CIFAR-10

Learned 2nd-layer Receptive Fields and Features

Before we look at classification results, we first inspect the learned features and their receptive fields from the second layer (i.e., the features that take the pooled first-layer responses as their input). Figure 4.1 shows two typical examples of receptive fields chosen by our method when using square-correlation as the similarity metric. In both of the examples, the receptive field incorporates filters with similar orientation tuning but varying phase, frequency and, sometimes, varying color. The position of the filters within each window indicates its location in the 2-by-2 region considered by the learning algorithm. As we might expect, the filters in each group are visibly

similar to those placed together by topographic methods like TICA that use related criteria.

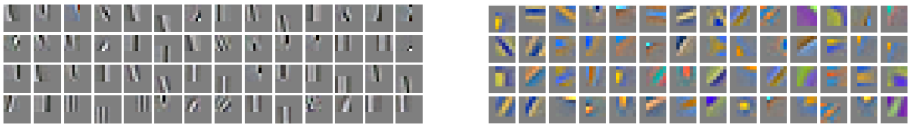


Figure 4.1: Two examples of receptive fields chosen from 2-by-2-by-1600 image representations. Each box shows the low-level filter and its position (ignoring pooling) in the 2-by-2 area considered by the algorithm. Only the most strongly dependent features from the $T = 200$ total features are shown. (Best viewed in color.)

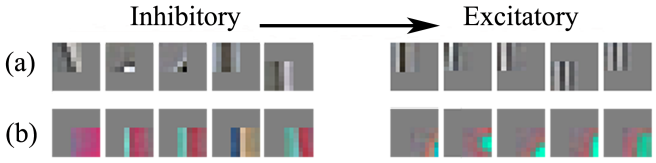


Figure 4.2: Most inhibitory (left) and excitatory (right) filters for two 2nd-layer features. (Best viewed in color.)

We also visualize some of the higher-level features constructed by the vector quantization algorithm when applied to these two receptive fields. The filters obtained from VQ assign weights to each of the lower level features in the receptive field. Those with a high positive weight are "excitatory" inputs (tending to lead to a high response when these input features are active) and those with a large negative weight are "inhibitory" inputs (tending to result in low filter responses). The 5 most inhibitory and excitatory inputs for two learned features are shown in Figure 4.2 (one from each receptive field in Figure 4.1). For instance, the two most excitatory filters of feature (a) tend to select for long, narrow vertical bars, inhibiting responses of wide bars.

Classification Results

We have tested our method on the task of image recognition using the CIFAR training and testing labels. Table 4.1 details our results using the full CIFAR dataset with various settings. We first note the comparison of our 2nd layer results with the

Table 4.1: Results on CIFAR-10 (full)

Architecture	**Accuracy (%)**
1 Layer	78.3%
1 Layer (4800 maps)	80.6%
2 Layers (Single RF)	77.4%
2 Layers (Random RF)	77.6%
2 Layers (Learned RF)	81.2%
3 Layers (Learned RF)	**82.0%**
Spherical K-means (6000 maps) [17]	81.5%
Conv. DBN [46]	78.9%
Deep NN [12]	80.49%

Table 4.2: Results on CIFAR-10 (400 ex. per class)

Architecture	**Accuracy (%)**
1 Layer	64.6% (±0.8%)
1 Layer (4800 maps)	63.7% (±0.7%)
2 Layers (Single RF)	65.8% (±0.3%)
2 Layers (Random RF)	65.8% (±0.9%)
2 Layers (Learned RF)	69.2% (±0.7%)
3 Layers (Learned RF)	**70.7%** (±0.7%)
Sparse coding (1 layer) [17]	66.4% (±0.8%)
VQ (1 layer) [17]	64.4% (±1.0%)

alternative of a single large 1st layer using an equivalent number of maps (4800) and see that, indeed, our 2nd layer created with learned receptive fields performs better (81.2% vs. 80.6%). We also see that the random and single receptive field choices work poorly, barely matching the *smaller* single-layer network. This confirms our belief that grouping together similar features is necessary to allow our unsupervised learning module (spherical K-means) to identify useful higher-level structure in the data. Finally, with a third layer of features, we achieve results exceeding our previous high (Table 3.4) on the full CIFAR dataset with 82.0% accuracy.

It is difficult to assess the strength of feature learning methods on the full CIFAR

Table 4.3: Classification Results on STL-10

Architecture	Accuracy (%)
1 Layer	54.5% (±0.8%)
1 Layer (4800 maps)	53.8% (±1.6%)
2 Layers (Single RF)	55.0% (±0.8%)
2 Layers (Random RF)	54.4% (±1.2%)
2 Layers (Learned RF)	58.9% (±1.1%)
3 Layers (Learned RF)	**60.1**% (±1.0%)
Sparse coding (1 layer) [17]	59.0% (±0.8%)
VQ (1 layer) [17]	54.9% (±0.4%)

dataset because the performance may be attributed to the success of the supervised SVM training and not the unsupervised feature training. For this reason we have also performed classification using 400 labeled examples per class.[5] Our results for this scenario are in Table 4.2. There we see that our 2-layer architecture significantly outperforms our 1-layer system as well as the two 1-layer architectures developed in Section 3.3.2. As with the full CIFAR dataset, we note that it was not possible to achieve equivalent performance by merely expanding the first layer or by using either of the alternative receptive field structures (which, again, make minimal gains over a single layer).

4.4.2 Comparison on STL-10

Finally, we also tested our algorithm on the STL-10 dataset [16]. Compared to CIFAR, STL provides many fewer labeled training examples (allowing 100 labeled instances per class for each training fold). Instead of relying on labeled data, one tries to learn from the provided unlabeled dataset, which contains images from a distribution that is similar to the labeled set but broader. We used the same architecture for this dataset as for CIFAR, but rather than train our features each time on the labeled training fold (which is too small), we use 20000 examples taken from the unlabeled dataset. Our results are reported in Table 4.3.

[5]Our networks are still trained unsupervised from the entire training set.

Here we see increasing performance with higher levels of features once more, achieving state-of-the-art performance with our 3-layered model. This is especially notable since the higher level features have been trained purely from unlabeled data. We note, one more time, that none of the alternative architectures (which roughly represent common practice for training deep networks) makes significant gains over the single layer system.

4.5 Summary

This chapter presented a mechanism to automatically learn the "receptive fields" used in deep feature representations. Though our previous systems (and much of prior art) rely on known spatial structure to restrict the connectivity of input features to output features, we showed that this becomes impractical when using an extremely large number of maps (i.e., very large choices of K in our unsupervised learning stage) since many scalable algorithms like K-means produced unorganized features whose relationships are not clear (unlike much more expensive topographic methods). To remedy this problem, we proposed to use pair-wise dependency tests amongst input features to greedily group inputs into receptive fields. Once these groups are identified, each receptive field is used to train new higher-level features that are later concatenated. In our results, we showed that this approach was not only superior to other "obvious" choices of connectivity but also better than very large single-layer systems that were consistently successful in benchmarks. Importantly, this method is a convenient "wrapper" that can be used as an off-the-shelf tool combined with any choice of UFL algorithm to help manage extremely large, higher-level feature representations where more traditional spatio-temporal local receptive fields are unhelpful or impossible to employ successfully.

Chapter 5

Application to Scene Text Recognition

5.1 Introduction

Detection of text and identification of characters in scene images is a challenging visual recognition problem. As in much of computer vision, the challenges posed by the complexity of these images have been combated with hand-designed features [20, 97, 92] and models that incorporate various pieces of high-level prior knowledge [91, 64]. In this chapter, we produce results from our feature learning systems applied to problems in this domain. Thus we attempt to use the algorithms developed in Chapters 2 & 3 to learn the necessary features directly from the data as an alternative to using purpose-built, text-specific features or models. Among our results, we have achieved performance among the best known on the ICDAR 2003 character recognition dataset using essentially an off-the-shelf instantiation of the pipeline in Chapter 3.

In contrast to more classical OCR problems, where the characters are typically monotone on fixed backgrounds, character recognition in scene images is potentially far more complicated due to the many possible variations in background, lighting, texture and font. As a result, building complete systems for these scenarios requires us to invent representations that account for all of these types of variations. Indeed, significant effort has gone into creating such systems, with top performers integrating

dozens of cleverly combined features and processing stages [64]. In contrast, the main goal of this thesis has been to study and identify promising algorithms that can learn higher level representations of data automatically for new tasks, thus avoiding some of this engineering effort. Scene text recognition is an interesting type of problem in this respect, since feature learning systems may be especially valuable when specialized features are clearly needed but it is difficult to build them by hand.

In this chapter, we will aim to determine to what extent the basic systems developed in earlier chapters may be useful in scene text detection (where we try to identify regions of text in an image) and character recognition (where we try to classify a small image as one of several characters in an alphabet). Specifically, we will again employ a system like the ones in Section 3.3.2 to learn image features for these tasks and then a supervised classification stage. Among our results, we will show the effect on recognition performance as we increase the number of learned features, and that it is possible to obtain performance comparable to or better than state-of-the-art systems.

5.2 Related Work

Scene text recognition has generated significant interest from many branches of research. While it is now possible to achieve extremely high performance on tasks such as digit recognition in controlled settings [68], the task of detecting and labeling characters in complex scenes remains an active research topic. However, many of the methods used for scene text detection and character recognition are predicated on purpose-built systems specific to the new task. For text detection, for instance, solutions have ranged from simple off-the-shelf classifiers trained on hand-coded features [11] to multi-stage pipelines combining many different algorithms [63, 64]. Common features include edge features, texture descriptors, and shape contexts [20]. Meanwhile, various flavors of probabilistic model have also been applied [91, 93, 25], folding many forms of prior knowledge into the detection and recognition system.

On the other hand, some systems with highly flexible learning schemes attempt to learn all necessary information from labeled data with minimal prior knowledge.

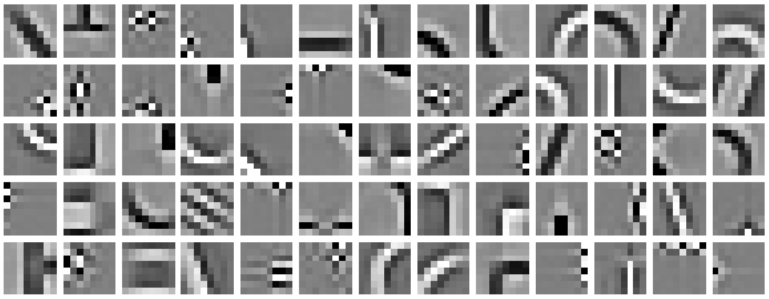


Figure 5.1: A small subset of the dictionary elements learned from grayscale, 8-by-8 pixel image patches extracted from the ICDAR 2003 dataset.

For instance, multi-layered neural network architectures have been applied to character recognition and are competitive with other leading methods [77]. This mirrors the success of such approaches in more traditional document and hand-written text recognition systems [50].

5.3 Feature Learning Architecture

The system we use to learn and extract features for this application is essentially identical to the one used in Section 3.3.2 with straight-forward changes to the parameters. Specifically, we use spherical K-means for the unsupervised learning algorithm, and again the soft-threshold function for $\Phi(x; \Theta)$. We use the convolutional architecture proposed in Section 2.2 with receptive field size $r = 8$ pixels and step size $s = 1$ pixel. That is, we use K-means to train dictionary parameters $\Theta = (D)$ from 8-by-8 pixel grayscale patches cropped from images of scene text, and then use the learned parameters to extract features using $\Phi(x; \Theta)$ from 8-by-8 sub-patches of a larger image.

Shown in Figure 5.1 are a set of dictionary elements (columns of D visualized as patches) resulting from application of spherical K-means to whitened patches extracted from small images obtained from the ICDAR 2003 dataset. Note that the features are specialized to the data—some elements correspond to short, curved strokes rather than simply to edges since such features are very common in images with text compared to natural images.

5.3.1 Feature extraction

Both our detector and character classifier consider 32-by-32 pixel images. To compute the feature representation of the 32-by-32 image, we compute the representation described above for every 8-by-8 sub-patch of the input, yielding a 25-by-25-by-K representation. This representation is then spatially pooled (as in Section 3.2.1) over 9 blocks in a 3-by-3 grid over the image, yielding a final feature vector ϕ with $9K$ features for this image.

5.3.2 Text detector training

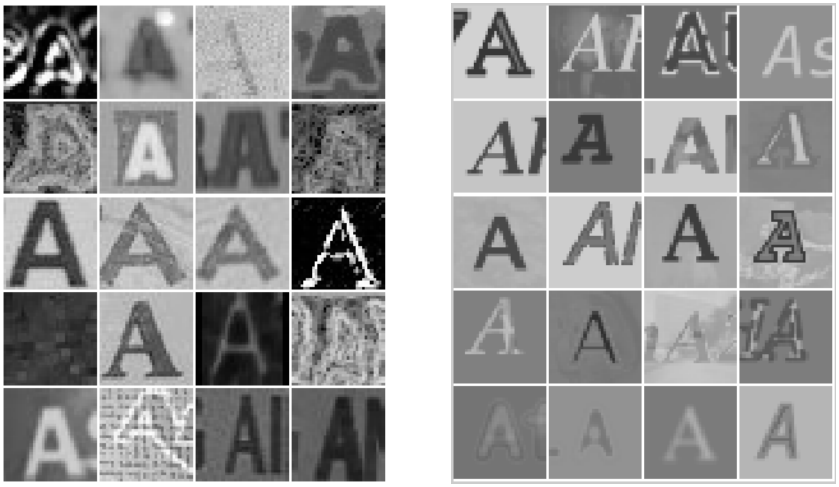
For text detection, we train a binary classifier that aims to distinguish 32-by-32 windows that contain text from windows that do not. We build a training set for this classifier by extracting 32-by-32 windows from the ICDAR 2003 training dataset, using the word bounding boxes to decide whether a window is text or non-text.[1] With this procedure, we harvest a set of 60000 32-by-32 windows for training (30000 positive, 30000 negative) from the ICDAR training data. We then use the feature extraction method described above to convert each image into a $9K$-dimensional feature vector. These feature vectors and the ground-truth "text" and "not text" labels acquired from the bounding boxes are then used to train a linear SVM. We will later use our feature extractor and the trained classifier for detection in the "sliding window" fashion.

5.3.3 Character classifier training

For character classification, we also use a fixed-sized input image of 32-by-32 pixels. Unlike the detector, however, the images are not "text" and "non-text" images, but images of cropped characters acquired from labeled datasets.[2] To build a character

[1] We define a window as "text" if 80% of the window's area is within a text region, and the window's width or height is within 30% of the width or height (respectively) of the ground-truth region. The latter condition ensures that the detector tends to detector characters of size similar to the window.

[2] Typically, input images from public datasets are already cropped to the boundaries of the character. Since our classifier uses a fixed-sized window, we re-cropped characters from the original



(a) Distorted ICDAR examples (b) Synthetic examples

Figure 5.2: Augmented training examples.

recognizer, we extract features from these 32-by-32 images and then train a 63-class (26 characters, upper and lower case, 10 digits) SVM classifier using the 1-versus-all approach. Other than the choice of input data and class labels, the feature extraction procedure is identical to the one used for the detection system.

Since we can produce large numbers of features using our feature learning system, over-fitting becomes a serious problem when training from the (relatively) small character datasets currently available. To help mitigate this problem, we have combined data from multiple sources. In particular, we have compiled our training data from the ICDAR 2003 training images [55], Weinman et al.'s sign reading dataset [91], and the English subset of the Chars74k dataset [20]. Our combined training set contains approximately 12400 labeled character images.

With large numbers of features, it is useful to have even more data. To satisfy these needs, we have also experimented with synthetic augmentations of these datasets. In particular, we have added synthetic examples that are copies of the ICDAR training samples with random distortions and image filters applied (see Figure 5.2(a)), as well as artificial examples of rendered characters blended with random scenery images (Figure 5.2(b)). With these examples included, our dataset includes a total of 49200

images using an enclosing window of the proper size.

images.

5.4 Detection and Recognition Experiments

We now present experimental results achieved with the system above, demonstrating the impact of being able to train increasing numbers of features. Specifically, for detection and character recognition, we trained our classifiers with increasing numbers of learned features (increasing values of K) and in each case evaluated the results on the ICDAR 2003 test sets for text detection and character recognition.

5.4.1 Detection

To evaluate our detector over a large input image, we take the classifier trained as in Section 5.3.2 and compute the features and classifier output for each 32-by-32 window of the image. We perform this process at multiple scales and then, for each location in the original image assign it a score equal to the maximum classifier output achieved at any scale. By this mechanism, we label each pixel with a score according to whether that pixel is part of a block of text. These scores are then thresholded to yield binary decisions at each pixel. By varying the threshold and using the ICDAR bounding boxes as per-pixel labels, we sweep out a precision-recall curve and report the area under the curve as our final performance measure.

Figure 5.3 plots the area under the precision-recall curve for our detector for varying numbers of features. It is seen there that performance improves consistently as we increase the number of features: our detector improves from roughly 0.3 AUC, to 0.45 AUC simply by including more features. While our performance is not yet comparable to top performing systems it is notable that our approach included virtually no prior knowledge. In contrast, Pan et al.'s recent state-of-the-art system [64] involves multiple highly tuned processing stages incorporating several sets of expert-chosen features.

Note that these numbers are per-pixel accuracies (i.e., the performance of the detector in identifying, for a single window, whether it is text or non-text). In practice,

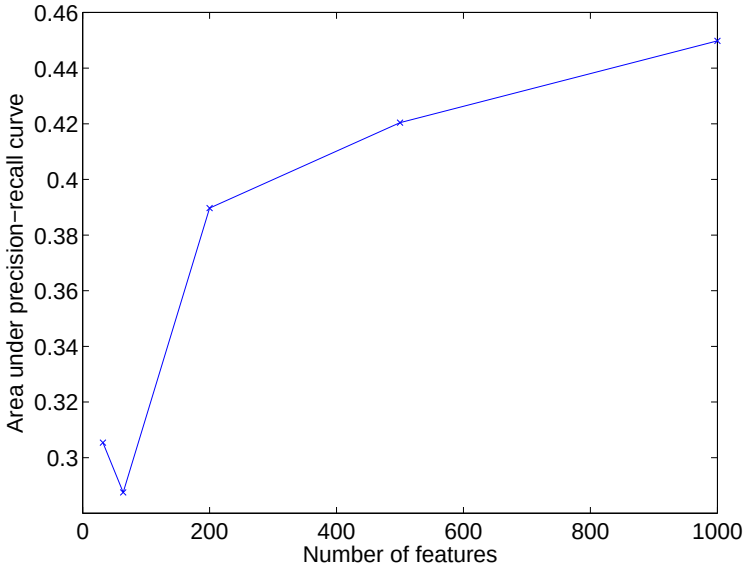


Figure 5.3: Area under PR curve as a function of number of learned features.

(a) ICDAR test image (b) Text detector scores

Figure 5.4: Example text detection classifier outputs.

the predicted labels of adjacent windows are highly correlated and thus the outputs include large contiguous "clumps" of positively and negatively labeled windows that could be passed on for more processing. A typical result generated by our detector is shown in Figure 5.4.

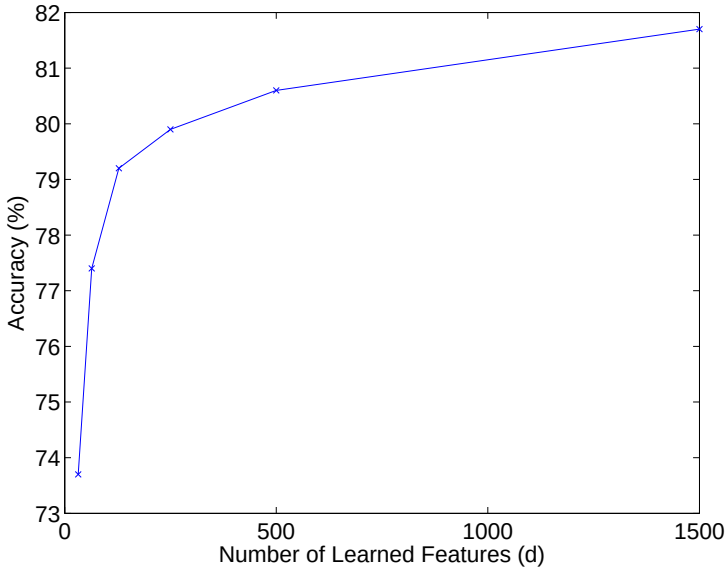


Figure 5.5: Character classification accuracy (62-way) on ICDAR 2003 test set as a function of the number of learned features.

Table 5.1: Test recognition accuracy on ICDAR 2003 character sets. (Dataset-Classes)

Algorithm	Test-62	Sample-62	Sample-36
Neumann and Matas, 2010 [59]	67.0%[7]	-	-
Yokobayashi et al., 2006 [97]	-	81.4%	-
Saidane and Garcia, 2007 [77]	-	-	84.5%
Our system	81.7%	81.4%	85.5%

5.4.2 Character Recognition

As with the detectors, we trained our character classifiers with varying numbers of features K on the combined training set described in Section 5.3. We then tested this classifier on the ICDAR 2003 test set, which contains 5198 test characters from 62 classes (10 digits, 26 upper- and 26 lower-case letters). The average classification accuracy on the ICDAR test set for increasing numbers of features is plotted in

[7] Achieved without pre-segmented characters.

Figure 5.5. Again, we see that accuracy climbs as a function of the number of features. Note that the accuracy for the largest system (1500 features) is the highest, at 81.7% for the 62-way classification problem. This is comparable or superior to other (purpose-built) systems tested on the same problem. For instance, the system in [97], achieves 81.4% on the smaller ICDAR "sample" set where we, too, achieve 81.4%. The authors of [77], employing a supervised convolutional network, achieve 84.5% on this dataset when it is collapsed to a 36-way problem (removing case sensitivity). In that scenario, our system achieves 85.5% with 1500 features. These results are summarized in comparison to other work in Table 5.1.

5.5 Summary

In this chapter we presented results from an application of our feature learning pipeline to scene text detection and character recognition. The basic system we used is essentially identical to the best performer from Chapter 3, emphasizing that these types of learning schemes can successfully construct feature representations that perform well on new, realistic tasks. Interestingly, the basic phenomenon identified earlier that showed increasing performance with larger numbers of features appears to hold for text detection and character recognition as well. (See, e.g., Figure 3.3.) Thus, while much research has focused on developing by hand the models and features used in scene-text applications, our results point out that it may be possible to achieve high performance using a more automated and generic approach. With more scalable and sophisticated feature learning algorithms currently being developed by machine learning researchers, it is possible that the approaches pursued here might achieve performance well beyond what is possible through other methods that rely heavily on hand-coded prior knowledge.

Chapter 6

Emergence of Object-Selective Features

6.1 Introduction

Previously in this thesis we have studied feature learning algorithms geared toward constructing feature representations for supervised tasks. In our experiments and application to scene text, we found that these algorithms were apparently able to learn useful high-level features without labels. Yet in the end we trained our features from labeled datasets (ignoring the labels) and used a supervised learning algorithm to learn to detect patterns like object classes that the unsupervised learning algorithm is not expected to find on its own. An interesting open question is whether unsupervised feature learning algorithms of the sort proposed in Chapter 3 are able to construct features, without the benefit of supervision, that can identify high-level concepts like frequently-occurring object classes. It is already known that this can be achieved when the dataset is sufficiently restricted that object classes are clearly defined (typically closely cropped images) and occur very frequently [54, 100, 103]. In this chapter our goal is to test whether unsupervised feature learning algorithms can achieve a similar result without any supervision at all.

The setting we consider for our experiments is a challenging one. We have harvested a dataset of 1.4 million image thumbnails from YouTube and extracted roughly

57 million 32-by-32 pixel patches at random locations and scales. These patches are very different from those found in the labeled datasets like CIFAR-10 [45] used for our other experiments. The overwhelming majority of patches in our dataset appear to be random clutter. In the cases where such a patch contains an identifiable object, it may well be scaled, arbitrarily cropped, or uncentered. As a result, it is very unclear what makes an "object class" in this type of patch dataset, and less clear that a completely unsupervised learning algorithm could manage to create "object-selective" features able to distinguish an object from the wide variety of clutter without some other type of supervision.

In order to have some hope of success, we can identify several key properties that our feature learning algorithm should likely have. First, since identifiable objects show up very rarely, it is clear that we are obliged to train from extremely large datasets. We have no way of controlling how often a particular object shows up and thus enough data must be used to ensure that an object class is seen many times—often enough that it cannot be disregarded as random clutter. Second, we are also likely to need a very large number of features. Training too few features will cause us to "underfit" the distribution, forcing the learning algorithm to ignore rare events like objects. Finally, we should aim to build features that incorporate invariance so that features respond not just to a specific pattern (e.g., an object at a single location and scale), but to a range of patterns that collectively belong to the same object class (e.g., the same object seen at many locations and scales). Unfortunately, these desiderata are difficult to achieve at once: typical off-the-shelf methods for building invariant hierarchies of features are not sufficiently scalable to be able to train many thousands of features from our 57 million patch dataset using our cluster of 30 machines.

In the preceding chapters we have already identified a learning algorithm (spherical K-means) capable of training large numbers of *selective* features from large datasets. That is, we are able to train many thousands of linear filters that respond whenever a given input appears similar to the filter. In this chapter, we propose a similarly scalable algorithm for building *invariant* features that respond to a range of related patterns. Surprisingly, we find that despite the simplicity of these algorithms we are nevertheless able to discover high-level features sensitive to the most commonly

occurring object class present in our dataset: human faces. In fact, we find that these features are better face detectors than a linear filter trained from labeled data, achieving up to 86% AUC compared to 77% on labeled diagnostic data. Thus, our results emphasize that not only can unsupervised feature learning algorithms discover object-selective features with no labeled data, but that such features can potentially perform better than supervised detectors due to their deep, nonlinear representation. A key point of these experiments is that the basic behavior of our feature learning algorithms is extremely similar to existing methods for building invariant feature hierarchies, suggesting that other popular feature learning methods currently available may also be able to achieve such results if run at large enough scale.

6.2 Algorithms

Our system is built on two separate learning modules: (i) an algorithm to learn selective features (linear filters that respond to a specific input pattern), and (ii) an algorithm to combine the selective features into invariant features (that respond to a spectrum of gradually changing patterns). We will refer to these features as "simple cells" and "complex cells" respectively, in analogy to previous work and to biological cells [35] with (very loosely) related response properties. Following other popular systems [72, 50, 37, 36] we will then use these two algorithms to build alternating layers of simple cell and complex cell features.

6.2.1 Learning Selective Features (Simple Cells)

The first module in our learning system trains a bank of linear filters to represent our selective "simple cell" features. For this purpose we use the spherical K-means system of Chapters 3-5 which has been a very successful approach to large-scale feature learning.

The algorithm is given a set of input vectors $x^{(i)} \in \mathbb{R}^n, i = 1, \ldots, m$. These vectors are pre-processed by removing the mean and normalizing each example, then performing PCA whitening. We then learn a dictionary $\Theta = D \in \mathbb{R}^{n \times K}$ of linear

filters using spherical K-means.

Given the linear filters D, we then define the responses of the learned simple cell features as $s^{(i)} = g(a^{(i)})$ where $a^{(i)} = D^\top x^{(i)}$ and $g(\cdot)$ is a nonlinear activation function. In our experiments we will typically use $g(a) = |a|$ for the first layer of simple cells, and $g(a) = a$ for the second.[1]

6.2.2 Learning Invariant Features (Complex Cells)

To construct invariant complex cell features, a common approach is to create "pooling units" that combine the responses of lower-level simple cells. In this work, we use max-pooling units [72, 54, 81]. Specifically, given a vector of simple cell responses $s^{(i)}$, we will train complex cell features whose responses are given by:

$$c_j^{(i)} = \max_{k \in G_j} s_k^{(i)}$$

where G_j is a set that specifies which simple cells the j'th complex cell should pool over. Thus, the complex cell c_j is an invariant feature that responds significantly to any of the patterns represented by simple cells in its group.

Each group G_j should specify a set of simple cells that are, in some sense, similar to one another. In convolutional neural networks [50], for instance, each group is hard-coded to include translated copies of the same filter resulting in complex cell responses c_j that are invariant to small translations. Some algorithms [37, 27] fix the groups G_j ahead of time then optimize the simple cell filters D so that the simple cells in each group share a particular form of statistical dependence. In our system, we will use linear correlation of the linear filter responses as our similarity metric, $\mathbb{E}[a_k a_l]$, and construct groups G_j that combine similar features according to this metric. Computing the similarity directly would normally require us to estimate the correlations from data, but since the inputs $x^{(i)}$ are whitened we can instead compute the similarity directly from the filter weights:

$$\mathbb{E}[a_k a_l] = \mathbb{E}\left[D^{(k)\top} x^{(i)} x^{(i)\top} D^{(l)}\right] = D^{(k)\top} D^{(l)}.$$

[1] This allows us to train roughly half as many simple cell features for the first layer.

For convenience in the following, we will actually use the dissimilarity between features, defined as $d(k,l) = ||D^{(k)} - D^{(l)}||_2 = \sqrt{2 - 2 * \mathbb{E}\left[a_k a_l\right]}$.

To construct the groups G, we will use a version of single-link agglomerative clustering to combine sets of features that have low dissimilarity according to $d(k,l)$.[2] To construct a single group G_0 we begin by choosing a random simple cell filter, say $D^{(k)}$, as the first member. We then search for candidate cells to be added to the group by computing $d(k,l)$ for each simple cell filter $D^{(l)}$ and add $D^{(l)}$ to the group if $d(k,l)$ is less than some limit τ. The algorithm then continues to expand G_0 by adding any additional simple cells that are closer than τ to any one of the simple cells already in the group. This procedure continues until there are no more cells to be added, or until the diameter of the group (the dissimilarity between the two furthest cells in the group) reaches a limit Δ.[3]

This procedure can be executed, quite rapidly, in parallel for a large number of randomly chosen simple cells to act as the "seed" cell, thus allowing us to train many complex cells at once. Compared to the simple cell learning procedure, the computational cost is extremely small even for our rudimentary implementation. In practice, we often generate many groups (e.g., several thousand) and then keep only a random subset of the largest groups. This ensures that we do not end up with many groups that pool over very few simple cells (and hence yield complex cells c_j that are not especially invariant).

6.3 Algorithm Behavior

Though it seems plausible that pooling simple cells with similar-looking filters according to $d(k,l)$ as above should give us some form of invariant feature, it may not yet be clear why this form of invariance is desirable. To explain, we will consider a simple "toy" data distribution where the behavior of these algorithms is more clear. Specifically, we will generate three heavy-tailed random variables X, Y, Z according

[2]Since the first layer uses $g(a) = |a|$, we actually use $d(k,l) = \min\{||D^{(k)} - D^{(l)}||_2, ||D^{(k)} + D^{(l)}||_2\}$ to account for $-D^{(l)}$ and $+D^{(l)}$ being essentially the same feature.

[3]In our experiments, we will use $\tau = 0.3$ for the first layer of complex cells, and $\tau = 1.0$ for the second layer. In all cases we use $\Delta = 1.5 > \sqrt{2}$.

to:

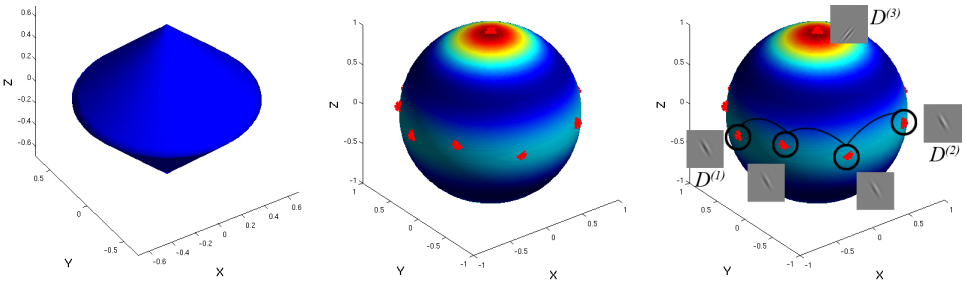
$$\sigma_1, \sigma_2 \sim \mathcal{L}(0, \lambda)$$
$$e_1, e_2, e_3 \sim \mathcal{N}(0, 1)$$
$$X = e_1\sigma_1, Y = e_2\sigma_1, Z = e_3\sigma_2$$

Here, σ_1, σ_2 are scale parameters sampled independently from a Laplace distribution, and e_1, e_2, e_3 are sampled independently from a unit Gaussian. The result is that Z is independent of both X and Y, but X and Y are not independent due to their shared scale parameter σ_1 [37]. An isocontour of the density of this distribution is shown in Figure 6.1a.

Other popular algorithms [37, 36, 27] for learning complex-cell features are designed to identify X and Y as features to be pooled together due to the correlation in their energies (scales). One empirical motivation for this kind of invariance comes from natural images: if we have three simple-cell filter responses $a_1 = D^{(1)\top}x$, $a_2 = D^{(2)\top}x$, $a_3 = D^{(3)\top}x$ where $D^{(1)}$ and $D^{(2)}$ are Gabor filters in quadrature phase, but $D^{(3)}$ is a Gabor filter at a different orientation, then the responses a_1, a_2, a_3 will tend to have a distribution very similar to the model of X, Y, Z above [38]. By pooling together the responses of a_1 and a_2 a complex cell is able to detect an edge of fixed orientation invariant to small translations. This model also makes sense for higher-level invariances where X and Y do not merely represent responses of linear filters on image patches but feature responses in a deep network. Indeed, the X–Y plane in Figure 6.1a is referred to as an "invariant subspace" [43].

Our combination of simple cell and complex cell learning algorithms above tend to learn this same type of invariance. After whitening and normalization, the data points X, Y, Z drawn from the distribution above will lie (roughly) on a sphere. The density of these data points is pictured in Figure 6.1b, where it can be seen that the highest density areas are in a "belt" in the X–Y plane and at the poles along the Z axis with a low-density region in between. If we apply our K-means clustering method to this dataset we get the centroids shown as $*$ marks in Figure 6.1b. From this picture it is clear what a subsequent application of our single-link clustering

algorithm will do: it will try to string together the centroids around the "belt" that forms the invariant subspace and avoid connecting them to the (distant) centroids at the poles. Max-pooling over the responses of these filters will result in a complex cell that responds consistently to points in the X–Y plane, but not in the Z direction—that is, we end up with an invariant feature detector very similar to those constructed by existing methods. Figure 6.1c depicts this result, along with visualizations of the hypothetical gabor filters $D^{(1)}, D^{(2)}, D^{(3)}$ described above that might correspond to the learned centroids.



(a) (b) (c)

Figure 6.1: (a) An isocontour of a sparse probability distribution over variables X, Y, and Z. (See text for additional detail.) (b) A visualization of the spherical density obtained from the distribution in (a) after normalization. Red areas are high density and dark blue areas are low density. Centroids learned by K-means from this data are shown on the surface of the sphere as red * marks. (c) A pooling unit identified by applying single-link clustering to the centroids (black links join pooled filters). (See text.)

6.3.1 Feature Hierarchy

Now that we have defined our simple and complex cell learning algorithms, we can use them to train alternating layers of selective and invariant features. We will train 4 layers total, 2 of each type. The architecture we use is pictured in 6.2a.

Our first layer of simple cell features are locally connected to 16 non-overlapping 8-by-8 pixel patches within the 32-by-32 pixel image. These features are trained by building a dataset of 8-by-8 patches and passing them to our simple cell learning

procedure to train 6400 first-layer filters $D \in \mathbb{R}^{64\times 6400}$. We apply our complex cell learning procedure to this bank of filters to find 128 pooling groups $G_1, G_2, \ldots, G_{128}$. Using these results, we can extract our simple cell and complex cell features from each 8-by-8 pixel subpatch of the 32-by-32 image. Specifically, the linear filters D are used to extract the first layer simple cell responses $s_i^{(p)} = g(D^{(i)\top} x^{(p)})$ where $x^{(p)}, p = 1, .., 16$ are the 16 subpatches of the 32-by-32 image. We then compute the complex cell feature responses $c_j^{(p)} = \max_{k\in G_j} s_k^{(p)}$ for each patch.

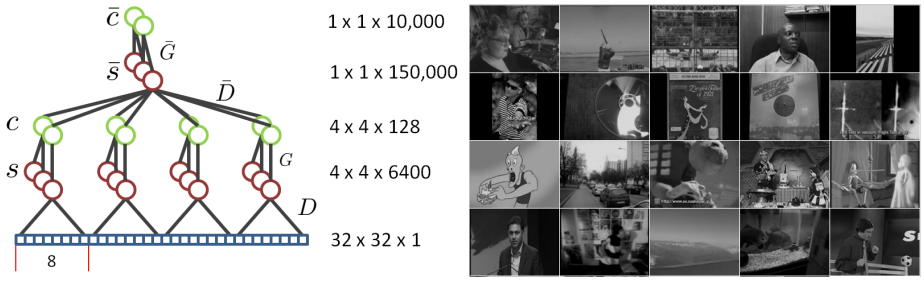
Once complete, we have an array of 128-by-4-by-4 = 2048 complex cell responses c representing each 32-by-32 image. These responses are then used to form a new dataset from which to learn a second layer of simple cells with K-means. In our experiments we train 150,000 second layer simple cells. We denote the second layer of learned filters as $\bar{D}$, and the second layer simple cell responses as $\bar{s} = \bar{D}^\top c$. Applying again our complex cell learning procedure to $\bar{D}$, we obtain pooling groups $\bar{G}$, and complex cells $\bar{c}$ defined analogously.

6.4 Experiments

As described above, our algorithm is trained from patches harvested from YouTube thumbnails downloaded from the web. Specifically, we downloaded the thumbnails for over 1.4 million YouTube videos[4], some of which are shown in Figure 6.2b. These images were downsampled to 128-by-96 pixels and converted to grayscale. We then cropped 57 million randomly selected 32-by-32 pixel patches from these images to form our unlabeled training set. No supervision is used here—thus most patches represent partial views of objects or clutter at differing scales. We applied our algorithm as described above to these images on a cluster of 30 machines. The entire training procedure takes approximately 3 days to run—virtually all of it consumed training the 150,000 high-level simple cells ($\bar{D}$).[5]

[4]We cannot select videos at random, so we query videos under each YouTube category ("Pets & Animals", "Science & Technology", etc.) along with a date (e.g., "January 2001").

[5]Though this is a fairly long run, we note that 1 iteration of K-means is cheaper than a single batch gradient step for most other methods able to learn high-level invariant features. We expect that these experiments would be impossible to perform in a reasonable amount of time on our cluster with another algorithm.



(a) (b)

Figure 6.2: (a) Cross-section of network architecture used for experiments. Full layer sizes are shown at right. (b) Randomly selected 128-by-96 images from our dataset.

6.4.1 Low-Level Simple and Complex Cell Visualizations

Before checking our higher-layer features for object-selectivity, we will first visualize the learned low-level filters D and pooling groups G to demonstrate that they are, in fact, quite similar to those learned by other well-known algorithms. It is already known that our K-means-based algorithm learns simple-cell-like filters (e.g., edge-like features as well as spots, curves) as can be seen in Figure 6.3a.

To visualize the learned complex cells, we can inspect the simple cell filters that belong to each of the pooling groups. The filters for several pooling groups are visualized in Figure 6.3b. There it can easily be seen that the filters, as expected, represent a spectrum of very similar image structures. Though many pairs of filters may be extremely similar[6], there are also other pairs that differ significantly and are included in the group due to the single-link clustering method. Note that some of our groups are composed of similar edges at differing locations, and thus appear to learn translation invariance as we expect.

6.4.2 Higher-Level Simple and Complex Cells

We will now investigate the learned higher layer simple cell and complex cell features, $\bar{s}$ and $\bar{c}$, to see if any of them have managed to become selective for an object class.

[6]Some filters have reversed polarity due to our use of absolute-value rectification during training of the first layer.

	Best 32-by-32 simple cell	Best in $\bar{s}$	Best in $\bar{c}$	Supervised Linear SVM
AUC	64%	86%	80%	77%

Table 6.1: Area under PR curve for different cells on our face detection validation set. Only the SVM uses labeled data.

In particular, the most commonly occurring object class in these video thumbnails is human faces (even though we estimate that much less than 0.1% of patches contain a well-framed face). Thus, we would like to know whether our algorithm has learned any higher-level features selective for human faces at varying locations and scales. To locate such features if they exist, we use labeled images from the "Labeled Faces in the Wild" (LFW) dataset [34]. We constructed a dataset composed of several hundred thousand non-face images as well as tens of thousands of known face images from the LFW dataset.

To test whether any of the $\bar{s}$ simple cell features are selective for faces, we used each feature by itself as a "detector" on the labeled dataset: we compute the area under the precision-recall curve (AUC) obtained when using each single feature's response $\bar{s}_k$ as a simple classifier. Indeed, it turns out that there *are* a handful of high-level features that tend to be good detectors for faces. The precision-recall curves for the best 5 detectors are shown in Figure 6.3c (top curves); the best of these achieves 86% AUC. We visualize 16 of the simple cell features identified by this procedure[7] in Figure 6.4(a) along with a sampling of the image patches that activate the first of these cells strongly. There it can be seen that these simple cells are selective for faces located at differing locations and at varying scales. Within each group the faces may differ slightly due to the learned invariance provided by the complex cells in the lower layer (and thus the mean of each group of images appears quite blurry).

On first sight, it appears that this result could have been obtained by applying our simple cell learning procedure directly to the 32-by-32 images without any attempts at incorporating local invariance as done here. That is, rather than training D (the

[7]We visualize the higher-level features by averaging together the 100 unlabeled images from our YouTube dataset that elicit the strongest activation.

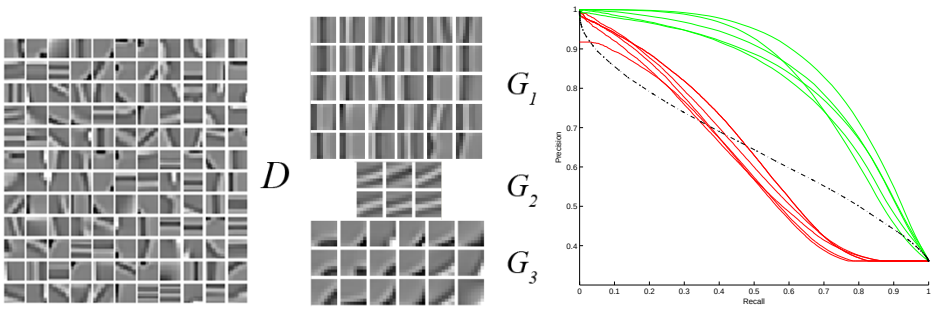


Figure 6.3: (a) First layer simple cell filters learned by K-means. (b) Sets of simple cell filters belonging to three pooling groups learned by our complex cell training algorithm. (c) Precision-Recall curves showing selectivity for human faces of 5 low-level simple cells trained from a full 32-by-32 patch (red curves, bottom) versus 5 higher-level simple cells (green curves, top). Performance of the best linear filter found by SVM from labeled data is also shown (black dotted curve middle).

first-layer filters) from 8-by-8 patches, we could simply train D directly from the 32-by-32 images. This turns out not to be successful. The lower curves in Figure 6.3c are the precision-recall curves for the best 5 simple cells found in this way. It can be seen quite clearly that the higher-level features are dramatically better detectors than simple cells built directly from pixels[8] (only 64% AUC).

As a second control experiment we also trained a linear SVM from half of the labeled data using only pixels as input (contrast-normalized and whitened). The PR curve for this linear classifier is shown in Figure 6.3c as a black dotted line. There we can see that the supervised linear classifier is significantly better (77% AUC) than the 32-by-32 linear simple cells. On the other hand, it does not perform as well as the higher level simple cells learned by our system even though it is likely the best possible linear detector.

Finally, we have applied the same complex-cell learning procedure as before to the

[8]These simple cells where trained by applying K-means to normalized, whitened 32-by-32 pixel patches from a smaller unlabeled set known to have a higher concentration of faces. As a result, a handful of centroids look roughly like face exemplars and hence can be considered simple "template matchers". When trained on the full dataset (which contains far fewer faces), K-means learns only edge and arc features which perform much worse (about 45% AUC).

higher-level simple cell filters. Due to the invariance introduced at the lower layers, two simple cells that detect faces at slightly different locations or scales will often have very similar filter weights and thus we expect our algorithm to find and combine these simple cells into higher-level invariant features.

We will visualize our higher-level complex cell features $\bar{c}$ to see what they detect. To visualize these features, we can simply look at visualizations for all of the simple cells in each of the groups $\bar{G}$. These visualizations show us the set of patches that strongly activate each simple cell, and hence also activate the complex cell. The results of such a visualization for one group that was found to contain only face-selective cells is shown in Figure 6.4c. There it can be seen that this single "complex cell" selects for faces at multiple positions and scales. A sampling of image patches collected from the unlabeled data that strongly activate the corresponding complex cell are shown in Figure 6.4d. We can see that the complex cell detects many faces but at a much wider variety of positions and scales compared to the simple cells, demonstrating that even "higher level" invariances are being captured, including scale invariance. Benchmarked on our labeled set, this complex cell achieves 80.0% AUC—somewhat worse than the very best simple cells, but still in the top 10 performing cells in the entire network. Interestingly, the qualitative results in Figure 6.4d are excellent, and we believe these images represent even greater range of variations than those in the labeled set. Thus the 80% AUC number may somewhat under-rate the quality of these features.

These results suggest that the basic notions of invariance and selectivity that underpin many popular feature learning algorithms may be sufficient to discover the kinds of high-level features that we desire, possibly including whole object classes robust to both local and global variations. Indeed, using extremely simple implementations of selective and invariant features closely related to existing algorithms, we have found that is possible to build features with high selectivity for a coherent, commonly occurring object class. Though human faces occur only very rarely in our very large dataset, it is clear that the complex cell visualized Figure 6.4d is adept at spotting them amongst tens of millions of images. The enabler for these results is the scalability of the algorithms we have employed, suggesting that other systems can

likely achieve similar results to the ones shown here if their computational limitations are overcome.

6.5 Related Work

The method that we propose here close connections to a wide array of prior work, as we have noted throughout this paper. For instance, the basic notions of selectivity and invariance that drive our system can be identified in many other algorithms. For instance, group sparse coding methods [27] and Topographic ICA [37, 38] build invariances by pooling simple cells that lie in an invariant subspace, identified by strong scale correlations between the cell responses. The advantage of the statistical criterion embodied by these algorithms to learn invariance is that they can often determine which features should be pooled together even when the simple cell filters are orthogonal (where they would be too far apart for our algorithm to recognize that their relationship since we use only linear correlation). Our results suggest that while this type of invariance is very useful, there are other ways to achieve a similar effect.

Our approach can also be connected with methods that attempt to model the geometric (e.g., manifold) structure of the input space. For instance, Contractive Auto-Encoders [74, 73], Local Coordinate Coding [99], and Locality-constrained Linear Coding [90] tend to learn sparse linear filters while also attempting to model the manifold structure staked out by these filters (sometimes termed "anchor points"). One plausible interpretation of our results, suggested by Figure 6.1b, is that with extremely overcomplete dictionaries it becomes possible to "walk" between distant anchor points on the image manifold, thus constructing pooling units that make our complex cells invariant to shifts along this manifold. [8] use similar intuitions to propose a clustering method that is quite similar to our own approach, though the method in their work is used to build pooling regions for supervised learning applications.

One of our key results, the unsupervised discovery of features selective for human faces, is fairly unique in the literature. Results of this kind have appeared in many guises in more restricted settings. For instance, [54] learned Deep Belief Network

models that decomposed object classes like faces, cars, and motorbikes into parts using a probabilistic version of translation-invariant max-pooling. Similarly, [100] has shown results of a similar flavor on the Caltech recognition datasets. [103] showed that a probabilistic model (with some hand-coded geometric knowledge) is able to correctly recover clusters containing 20 known object class silhouettes from outlines in the LabelMe dataset. Other authors have also shown the ability to discover some form of global invariance or manifold structure (e.g., as seen in the results of embedding algorithms [86, 75]) when trained in similarly restricted settings.

The methods above, however, are usually discovering decompositions of objects into parts or some other structure that is far more apparent when we are using labeled, *tightly cropped* images. Even if we do not use the labels themselves the labeled examples are, by construction, highly clustered: faces will be better-separated from other objects because there are no partial faces or random clutter. In our dataset, no supervision is used except to probe the representation post hoc. Our system nevertheless appears capable of identifying faces in the far more general scenario that we have proposed.

Finally, we also note the recent, extensive findings of Le et al. [49]. In that work an extremely large 9-layer neural network based on a TICA-like learning algorithm [48, 37] also appears to be capable of identifying a wide variety of object classes (including cats and upper-bodies of people) seen in YouTube videos. Our results complement this work in several key ways. First, by training on smaller randomly cropped patches, we show that object-selectivity may still be obtained even when objects are framed properly within the image only rarely. By contrast, the input images used by Le et al. are so large (200-by-200 pixels) it is more likely to be the case that a full object may be found somewhere in the image.[9] Second, and perhaps more important, we have shown that the key concepts of selectivity and invariance (sparse selective filters combined with invariant-subspace pooling) present in their system may also be implemented in a very different way using scalable clustering algorithms, allowing us to achieve results reminiscent of theirs but using a vastly smaller amount of computing

[9]Of course, their network still performs the difficult feat of learning invariance to long-range translations within this window.

power. (We used 240 cores, while their large-scale system is composed of 16,000 cores.) In combination, these results point strongly to the conclusion that highly scalable implementations of *existing* feature-learning concepts can discover very sophisticated high-level representations.

6.6 Summary

In this chapter we presented a variation on our previous feature learning systems composed of highly scalable learning methods. To learn simple selective features ("simple cells"), we applied K-means clustering, which we have used throughout this thesis. In addition, we have replaced the hard-coded spatial invariances (pooling) used in Chapters 3-4 with a second algorithm: agglomerative clustering, which stitches the simple cells together into invariant features ("complex cells"). Based on the observation that K-means tends to tile the invariant subspaces of the input data distribution, we argue that these invariant features are essentially the same as those identified by other (much more expensive) learning algorithms. We showed that these two components are, in fact, capable of learning complicated high-level representations in large scale experiments on unlabeled images from YouTube. Specifically, we showed that higher level simple cells could learn to detect human faces without any supervision at all, and that our complex-cell learning procedure could combine these into even higher-level invariances. These results indicate that we are apparently equipped with many of the key principles needed to achieve such results and that a critical remaining puzzle is how to scale up more sophisticated algorithmic solutions to the sizes needed to capture more object classes and even more complex invariances.

(a) (b) (c)

(d)

Figure 6.4: Visualizations. (a) A collection of patches from our unlabeled dataset that maximally activate one of the high-level simple cells from $\bar{s}$. (b) The mean of the top stimuli for a handful of face-selective cells in $\bar{s}$. (c) Visualization of the face-selective cells that belong to one of the complex cells in $\bar{c}$ discovered by the single-link clustering algorithm applied to $\bar{D}$. (d) A collection of unlabeled patches that elicit a strong response from the complex cell visualized in (c) — virtually all are faces, at a variety of scales and positions. Compare to (a).

Chapter 7

Conclusions

This thesis has presented a detailed study of a variety of unsupervised feature learning algorithms. In particular, we have sought to develop a relatively flexible set of ingredients from which to construct UFL systems: a common template for image recognition algorithms, many choices of unsupervised learning algorithms, a variety of encoding schemes, and multiple ways of cobbling these pieces together into a hierarchy of learned features. Using these ingredients as a starting point, we have evaluated a wide range of variations and benchmarked them on common recognition tasks. Though such results can often be highly application-specific, it turns out that there are certain trends that hold true across many types of image recognition problems: feature representations with very large numbers of features, large datasets and otherwise simple learning algorithms are often top performers, even enabling state-of-the-art results on applications like scene-text recognition. More important, taken to their limits, we have even found that feature-learning systems with similar components can learn to identify complex patterns like human faces (complete with translation and scale invariance) without any supervision. These results suggest, critically, that the components needed to learn successful hierarchies of features are not necessarily complex, and that the key pieces of the puzzle that deserve added attention are often unrelated to the learning algorithm itself.

More specifically, some of the main findings of this work have been:

1. Even simple unsupervised learning algorithms that find sparse linear projections

of data are sufficient to learn layers of features.

2. The choice of "architecture" (how features are connected to inputs, and the type of encoding Φ used), is often more critical than the unsupervised learning algorithm. In practice, larger representations are almost universally better performers.

3. Combined with very simple algorithms to build invariant features (similar in behavior to existing methods [43, 44, 37, 27]), we can construct high-level invariant features selective for a commonly occurring object class.

This last result is useful not because it suggests a *new algorithm*. Instead it shows that known notions of selectivity and invariance, even implemented with stunning simplicity, are sufficient to generate complicated but intuitive high-level features. The key to achieving this result was, as suggested by our earlier analysis, to refocus resources on problems of scalability.

Overall, the take-home message of this line of work is that scalability and exogenous system parameters play a remarkably large role in the success or failure of feature learning algorithms. When we push these design choices to their limits, we have been able to repeatedly best state-of-the-art benchmark scores and even construct very competitive systems in fields where we have little prior expertise (as in our scene text recognition work of Chapter 5). Similarly, with only a little extra algorithmic novelty, we are starting to see hints of the ability to discover complex features like objects. These promising early results emphasize the importance of carefully teasing out the major contributors to success in feature learning systems and then optimizing along those dimensions; indeed, this thesis has shown that on a modest scale the payoff may be substantial.

Bibliography

[1] R. Adams, H. Wallach, and Z. Ghahramani. Learning the structure of deep sparse graphical models. In *International Conference on AI and Statistics*, 2010.

[2] A. Agarwal and B. Triggs. Hyperfeatures: Multilevel local coding for visual recognition. In *European Conference on Computer Vision*, 2006.

[3] M. Banko and E. Brill. Scaling to very very large corpora for natural language disambiguation. In *39th Annual Meeting on Association for Computational Linguistics*, 2001.

[4] A. Bell and T. J. Sejnowski. The 'independent components' of natural scenes are edge filters. *Vision Research*, 37, 1997.

[5] T. Blumensath and M. E. Davies. On the difference between orthogonal matching pursuit and orthogonal least squares. Unpublished manuscript, 2007.

[6] Y. Boureau, F. Bach, Y. LeCun, and J. Ponce. Learning mid-level features for recognition. In *Computer Vision and Pattern Recognition*, 2010.

[7] Y. Boureau, J. Ponce, and Y. LeCun. A theoretical analysis of feature pooling in visual recognition. In *International Conference on Machine Learning*, 2010.

[8] Y. Boureau, N. L. Roux, F. Bach, J. Ponce, and Y. LeCun. Ask the locals: multi-way local pooling for image recognition. In *International Conference on Computer Vision*, 2011.

[9] C. J. Burges and D. J. Crisp. Uniqueness of the svm solution. In *Advances in Neural Information Processing Systems*, pages 223–229, 1999.

[10] O. Chapelle, B. Schölkopf, and A. Zien. *Semi-supervised learning*. MIT Press, Cambridge, Mass, 2006.

[11] X. Chen and A. Yuille. Detecting and reading text in natural scenes. In *Computer Vision and Pattern Recognition*, volume 2, 2004.

[12] D. Ciresan, U. Meier, J. Masci, L. M. Gambardella, and J. Schmidhuber. High-performance neural networks for visual object classification. *Computing Research Repository*, 2011. http://arxiv.org/abs/1102.0183.

[13] A. Coates, P. Baumstarck, Q. Le, and A. Y. Ng. Scalable learning for object detection with GPU hardware. In *IROS*, 2009.

[14] A. Coates, B. Carpenter, C. Case, S. Satheesh, B. Suresh, T. Wang, D. J. Wu, and A. Y. Ng. Text detection and character recognition in scene images with unsupervised feature learning. In *International Conference on Document Analysis and Recognition*, 2011.

[15] A. Coates, A. Karpathy, and A. Y. Ng. On the emergence of object-selective features in unsupervised feature learning. Unpublished manuscript, 2012.

[16] A. Coates, H. Lee, and A. Y. Ng. An analysis of single-layer networks in unsupervised feature learning. In *International Conference on AI and Statistics*, 2011.

[17] A. Coates and A. Y. Ng. The importance of encoding versus training with sparse coding and vector quantization. In *International Conference on Machine Learning*, 2011.

[18] A. Coates and A. Y. Ng. Selecting receptive fields in deep networks. In *Advances in Neural Information Processing Systems*, 2011.

[19] G. Csurka, C. Dance, L. Fan, J. Willamowski, and C. Bray. Visual categorization with bags of keypoints. In *ECCV Workshop on Statistical Learning in Computer Vision*, 2004.

[20] T. E. de Campos, B. R. Babu, and M. Varma. Character recognition in natural images. In *Proceedings of the International Conference on Computer Vision Theory and Applications, Lisbon, Portugal*, February 2009.

[21] A. Dempster, N. Laird, and D. Rubin. Maximum likelihood from incomplete data via the em algorithm. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 1–38, 1977.

[22] J. Deng, A. Berg, K. Li, and L. Fei-Fei. What does classifying more than 10,000 image categories tell us? In *12th European Conference of Computer Vision*, 2010.

[23] I. S. Dhillon and D. M. Modha. Concept decompositions for large sparse text data using clustering. *Machine Learning*, 42(1), 2001.

[24] B. Efron, T. Hastie, I. Johnstone, and R. Tibshirani. Least angle regression. *The Annals of statistics*, 32(2):407–499, 2004.

[25] X. Fan and G. Fan. Graphical Models for Joint Segmentation and Recognition of License Plate Characters. *IEEE Signal Processing Letters*, 16(1), 2009.

[26] L. Fei-Fei and P. Perona. A Bayesian hierarchical model for learning natural scene categories. In *Computer Vision and Pattern Recognition*, 2005.

[27] P. Garrigues and B. Olshausen. Group sparse coding with a laplacian scale mixture prior. In *Advances in Neural Information Processing Systems*, 2010.

[28] I. Goodfellow, Q. Le, A. Saxe, H. Lee, and A. Ng. Measuring invariances in deep networks. In *Advances in Neural Information Processing Systems*, 2009.

[29] K. Gregor and Y. LeCun. Emergence of complex-like cells in a temporal product network with local receptive fields. *Computing Research Repository*, 2010. http://arxiv.org/abs/1006.0448.

[30] K. Gregor and Y. LeCun. Learning fast approximations of sparse coding. In *International Conference on Machine Learning*, 2010.

[31] G. Hinton, S. Osindero, and Y. Teh. A fast learning algorithm for deep belief nets. *Neural Computation*, 18(7), 2006.

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

[33] F. Huang and Y. LeCun. Large-scale learning with SVM and convolutional nets for generic object categorization. In *Computer Vision and Pattern Recognition*, 2006.

[34] G. B. Huang, M. Ramesh, T. Berg, and E. Learned-Miller. Labeled faces in the wild: A database for studying face recognition in unconstrained environments. Technical Report 07-49, University of Massachusetts, Amherst, October 2007.

[35] D. Hubel and T. Wiesel. Receptive fields of single neurones in the cat's striate cortex. *The Journal of physiology*, 148(3):574–591, 1959.

[36] A. Hyvärinen and P. Hoyer. Emergence of phase-and shift-invariant features by decomposition of natural images into independent feature subspaces. *Neural Computation*, 12(7):1705–1720, 2000.

[37] A. Hyvärinen, P. Hoyer, and M. Inki. Topographic independent component analysis. *Neural Computation*, 13(7):1527–1558, 2001.

[38] A. Hyvärinen, J. Hurri, and P. Hoyer. *Natural Image Statistics*. Springer-Verlag, 2009.

[39] A. Hyvärinen and E. Oja. Independent component analysis: algorithms and applications. *Neural networks*, 13(4-5), 2000.

[40] K. Jarrett, K. Kavukcuoglu, M. Ranzato, and Y. LeCun. What is the best multi-stage architecture for object recognition? In *International Conference on Computer Vision*, 2009.

[41] K. Kavukcuoglu, M. Ranzato, and Y. LeCun. Fast inference in sparse coding algorithms with applications to object recognition. Technical Report CBLL-TR-2008-12-01, Computational and Biological Learning Lab, Courant Institute, NYU, 2008.

[42] K. Kavukcuoglu, P. Sermanet, Y. Boureau, K. Gregor, M. Mathieu, and Y. LeCun. Learning convolutional feature hierarchies for visual recognition. In *Advances in Neural Information Processing Systems*, 2010.

[43] T. Kohonen. Emergence of invariant-feature detectors in self-organization. In M. Palaniswami et al., editor, *Computational Intelligence, A Dynamic System Perspective*, pages 17–31. IEEE Press, New York, 1995.

[44] T. Kohonen. Emergence of invariant-feature detectors in the adaptive-subspace self-organizing map. In *Biological Cybernetics*, volume 75, pages 281–291, 1996.

[45] A. Krizhevsky. Learning multiple layers of features from Tiny Images. Master's thesis, Dept. of Comp. Sci., University of Toronto, 2009.

[46] A. Krizhevsky. Convolutional Deep Belief Networks on CIFAR-10. Unpublished manuscript, 2010.

[47] S. Lazebnik, C. Schmid, and J. Ponce. Beyond bags of features: Spatial pyramid matching for recognizing natural scene categories. In *Computer Vision and Pattern Recognition*, 2006.

[48] Q. Le, A. Karpenko, J. Ngiam, and A. Ng. ICA with reconstruction cost for efficient overcomplete feature learning. In *Advances in Neural Information Processing Systems*, 2011.

[49] Q. Le, M. Ranzato, R. Monga, M. Devin, K. Chen, G. Corrado, J. Dean, and A. Ng. Building high-level features using large scale unsupervised learning. In *International Conference on Machine Learning*, 2012.

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

[51] Y. LeCun, F. Huang, and L. Bottou. Learning methods for generic object recognition with invariance to pose and lighting. In *Computer Vision and Pattern Recognition*, 2004.

[52] H. Lee, A. Battle, R. Raina, and A. Y. Ng. Efficient sparse coding algorithms. In *Advances in Neural Information Processing Systems*, 2007.

[53] H. Lee, C. Ekanadham, and A. Y. Ng. Sparse deep belief net model for visual area V2. In *Advances in neural information processing systems*, 2008.

[54] H. Lee, R. Grosse, R. Ranganath, and A. Y. Ng. Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations. In *International Conference on Machine Learning*, 2009.

[55] S. Lucas, A. Panaretos, L. Sosa, A. Tang, S. Wong, and R. Young. ICDAR 2003 robust reading competitions. *International Conference on Document Analysis and Recognition*, 2003.

[56] J. Mairal, F. Bach, J. Ponce, and G. Sapiro. Online learning for matrix factorization and sparse coding. *Journal of Machine Learning Research*, 11:19–60, 2010.

[57] V. Nair and G. E. Hinton. 3D object recognition with deep belief nets. In *Advances in Neural Information Processing Systems*, 2009.

[58] V. Nair and G. E. Hinton. Rectified Linear Units Improve Restricted Boltzmann Machines. In *International Conference on Machine Learning*, 2010.

[59] L. Neumann and J. Matas. A method for text localization and recognition in real-world images. In *Asian Conference on Computer Vision*, 2010.

[60] E. Nowak, F. Jurie, and B. Triggs. Sampling strategies for bag-of-features image classification. In *European Conference on Computer Vision*, 2006.

[61] B. Olshausen and D. Field. Sparse coding of sensory inputs. *Current opinion in neurobiology*, 14(4), 2004.

[62] B. A. Olshausen and D. J. Field. Emergence of simple-cell receptive field properties by learning a sparse code for natural images. *Nature*, 381(6583):607–609, 1996.

[63] Y. Pan, X. Hou, and C. Liu. A robust system to detect and localize texts in natural scene images. In *International Workshop on Document Analysis Systems*, 2008.

[64] Y. Pan, X. Hou, and C. Liu. Text localization in natural scene images based on conditional random field. In *International Conference on Document Analysis and Recognition*, 2009.

[65] Y. C. Pati, R. Rezaifar, and P. S. Krishnaprasad. Orthogonal matching pursuit: recursive function approximation with applications to wavelet decomposition. In *Asilomar Conference on Signals, Systems and Computers*, November 1993.

[66] N. Pinto, D. Doukhan, J. J. DiCarlo, and D. D. Cox. A high-throughput screening approach to discovering good forms of biologically inspired visual representation. *PLoS Comput Biol*, 2009.

[67] R. Raina, A. Battle, H. Lee, B. Packer, and A. Ng. Self-taught learning: transfer learning from unlabeled data. In *24th International Conference on Machine learning*, 2007.

[68] M. Ranzato, Y. Boureau, and Y. LeCun. Sparse feature learning for deep belief networks. In *Advances in Neural Information Processing Systems*, 2007.

[69] M. Ranzato and G. E. Hinton. Modeling Pixel Means and Covariances Using Factorized Third-Order Boltzmann Machines. In *Computer Vision and Pattern Recognition*, 2010.

[70] M. Ranzato, A. Krizhevsky, and G. E. Hinton. Factored 3-way Restricted Boltzmann Machines for Modeling Natural Images. In *Thirteenth International Conference on Artificial Intelligence and Statistics*, 2010.

[71] M. Ranzato, C. Poultney, S. Chopra, and Y. LeCun. Efficient learning of sparse representations with an energy-based model. In *Advances in Neural Information Processing Systems*, 2007.

[72] M. Riesenhuber and T. Poggio. Hierarchical models of object recognition in cortex. *Nature neuroscience*, 2, 1999.

[73] S. Rifai, Y. Dauphin, P. Vincent, Y. Bengio, and X. Muller. The manifold tangent classifier. In *Advances in Neural Information Processing*, 2011.

[74] S. Rifai, P. Vincent, X. Muller, X. Glorot, and Y. Bengio. Contractive auto-encoders: Explicit invariance during feature extraction. In *International Conference on Machine Learning*, 2011.

[75] S. Roweis and L. Saul. Nonlinear dimensionality reduction by locally linear embedding. *Science*, 290(5500):2323—2326, December 2000.

[76] D. Rumelhart, G. Hintont, and R. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.

[77] Z. Saidane and C. Garcia. Automatic scene text recognition using a convolutional neural network. In *Workshop on Camera-Based Document Analysis and Recognition*, 2007.

[78] R. Salakhutdinov and G. E. Hinton. Deep Boltzmann Machines. In *Twelfth International Conference on Artificial Intelligence and Statistics*, 2009.

[79] B. Sapp, A. Saxena, and A. Y. Ng. A fast data collection and augmentation procedure for object recognition. In *AAAI Twenty-Third Conference on Artificial Intelligence*, 2008.

[80] A. Saxe, P. Koh, Z. Chen, M. Bhand, B. Suresh, and A. Y. Ng. On random weights and unsupervised feature learning. In *International Conference on Machine Learning*, 2011.

[81] D. Scherer, A. Mller, and S. Behnke. Evaluation of pooling operations in convolutional architectures for object recognition. In *International Conference on Artificial Neural Networks*, 2010.

[82] E. Simoncelli and O. Schwartz. Modeling surround suppression in v1 neurons with a statistically derived normalization model. *Advances in Neural Information Processing Systems*, 1998.

[83] R. Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 267–288, 1996.

[84] A. Torralba, R. Fergus, and W. Freeman. 80 million tiny images: a large dataset for non-parametric object and scene recognition. In *Transactions on Pattern Analysis and Machine Intelligence*, 2007.

[85] R. Uetz and S. Behnke. Large-scale object recognition with CUDA-accelerated hierarchical neural networks. In *Intelligent Computing and Intelligent Systems*, 2009.

[86] L. van der Maaten and G. Hinton. Visualizing high-dimensional data using t-SNE. *Journal of Machine Learning Research*, 9:2579—2605, November 2008.

[87] J. C. van Gemert, J. M. Geusebroek, C. J. Veenman, and A. W. M. Smeulders. Kernel codebooks for scene categorization. In *European Conference on Computer Vision*, 2008.

[88] M. Varma and A. Zisserman. A statistical approach to material classification using image patch exemplars. In *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2006.

[89] P. Vincent, H. Larochelle, Y. Bengio, and P. Manzagol. Extracting and composing robust features with denoising autoencoders. In *International Conference on Machine Learning*, 2008.

[90] J. Wang, J. Yang, K. Yu, F. Lv, T. Huang, and Y. Gong. Locality-constrained linear coding for image classification. In *Computer Vision and Pattern Recognition*, 2010.

[91] J. Weinman, E. Learned-Miller, and A. R. Hanson. Scene text recognition using similarity and a lexicon with sparse belief propagation. *Transactions on Pattern Analysis and Machine Intelligence*, 31(10), 2009.

[92] J. J. Weinman. Typographical features for scene text recognition. In *Proc. IAPR International Conference on Pattern Recognition*, pages 3987–3990, Aug. 2010.

[93] J. J. Weinman, E. Learned-Miller, and A. R. Hanson. A discriminative semi-markov model for robust scene text recognition. In *Proc. IAPR International Conference on Pattern Recognition*, Dec. 2008.

[94] J. Winn, A. Criminisi, and T. Minka. Object categorization by learned universal visual dictionary. In *International Conference on Computer Vision*, 2005.

[95] T. Wu and K. Lange. Coordinate descent algorithms for lasso penalized regression. *Annals of Applied Statistics*, 2(1), 2008.

[96] J. Yang, K. Yu, Y. Gong, and T. S. Huang. Linear spatial pyramid matching using sparse coding for image classification. In *Computer Vision and Pattern Recognition*, 2009.

[97] M. Yokobayashi and T. Wakahara. Binarization and recognition of degraded characters using a maximum separability axis in color space and gat correlation. In *International Conference on Pattern Recognition*, volume 2, pages 885–888, 2006.

[98] K. Yu and T. Zhang. Improved local coordinate coding using local tangents. In *International Conference on Machine Learning*, 2010.

[99] K. Yu, T. Zhang, and Y. Gong. Nonlinear learning using local coordinate coding. In *Advances in Neural Information Processing Systems*, 2009.

[100] M. D. Zeiler, G. W. Taylor, and R. Fergus. Adaptive deconvolutional networks for mid and high level feature learning. In *International Conference on Computer Vision*, 2011.

[101] C. Zetzsche, G. Krieger, and B. Wegmann. The atoms of vision: Cartesian or polar? *Journal of the Optical Society of America*, 16(7), July 1999.

[102] K. Zhang and L. Chan. Ica with sparse connections. *Intelligent Data Engineering and Automated Learning*, 2006.

[103] L. Zhu, Y. Chen, A. Torralba, W. Freeman, and A. Yuille. Part and Appearance Sharing: Recursive Compositional Models for Multi-View Multi-Object Detection. In *Computer Vision and Pattern Recognition*, 2010.