Hierarchical Convolutional Deep Learning in Computer Vision

by

Matthew D. Zeiler

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
Department of Computer Science
New York University
January, 2014

_____

Rob Fergus

Dedication

To Lisa

Acknowledgments

Joining the machine learning group at New York University was one of the best decisions of my life. I have to thank Rob Fergus for the continous support and for the many things he has taught me. He had a project ready for me as soon as I arrived at NYU, enabling me to jump into research immediately. His openness to meet at any time of day or night was crucial to my learning process and coming up with new ideas. His insights made debugging and implementing complicated algorithms relatively easy.

Additionally, this work would not have been possible without the help of Graham Taylor and Geoff Hinton who supervised me in my undergraduate thesis at the University of Toronto. They taught me the foundations of machine learning which I further built upon throughout my PhD. I had the opportunity to work with Graham directly for the first two years of my PhD as well. His patience with my persistent questions was essential to my progress in the early years of the program. I also had the pleasure of working with Geoff twice more during internships at Google. His intuition about what works in machine learning inspired me in the later portion of my PhD to think outside the box and produce some state-of-the-art results.

I would also like to thank all the students and postdocs at NYU for the interesting discussions and research they have all done. Especially Yann LeCun for the engaging chats and group meetings throughout my years at NYU.

Throughout my internships at Google during my last two summers of my PhD I learned a tremendous amount of implementation details that make training large-scale machine learning systems feasible. Without the knowledge I gained from this experience, my latest work would not have been possible. I would like to thank Marc'Aurelio Ranzato, Matthieu Devin, Rajat Monga, Greg Corrado, Vincent Vanhoucke and the rest of the Google Brain team for all the discussions and practical experience they taught me. I'd especially like to thank Jeff Dean for being an excellent mentor at Google, an insirpation for me in the field of computer science and for teaching me how to effectively scale

systems.

I thank my family for their continuing support in my studies and helping me with all the tough decisions along the way. Finally, I thank my girlfriend Lisa whose patience with my long hours of work and her loving support helped me get through the thick and thin of my PhD. Despite the exciting research I had the opportunity to do, it was her comedic upbeat attitude and encouragement that kept me going. Without her this would not have been possible.

Preface

The work in the first part of this thesis has been published in the following publications:

- M. Zeiler, D. Krishnan, G. Taylor, and R. Fergus. Deconvolutional networks. In *CVPR*, 2010.
- M. Zeiler, G. Taylor, and R. Fergus. Adaptive deconvolutional networks for mid and high level feature learning. In *ICCV*, 2011.
- M. Zeiler and R. Fergus. Differentiable pooling for hierarchical feature learning. *arXiv*, arXiv:1207.0151v1, 2013.
- M. Zeiler and R. Fergus. Stochastic pooling for regularization of deep convolutional neural networks. In *ICLR* , 2013.
- M. Zeiler and R. Fergus. Visualizing and understanding convolutional neural networks. In *ArXiv*, 2013.

The second part of the thesis includes other work done during my PhD that was not directly related to the work above:

- M. Zeiler, G. Taylor, L. Sigal, I. Matthews, and R. Fergus. Facial Expression Transfer with Input-Output Temporal Restricted Boltzmann Machines. *NIPS*, 2013.
- M Zeiler, M. Ranzato, R. Monga, M. Mao, K. Yang, Q. Le, P. Nguyen, A. Senior, V. Vanhoucke, J. Dean, G. Hinton. On Rectified Linear Units For Speech Processing. In *ICASSP*, 2013. [1]
- M. Zeiler. ADADELTA: An Adaptive Learning Rate Method. In *arXiv*, arXiv:1212.5701, 2012. [2]

[1] All this work was done while an intern at Google in 2012.
[2] Initial work was done while an intern at Google in 2012.

Abstract

It has long been the goal in computer vision to learn a hierarchy of features useful for object recognition. Spanning the two traditional paradigms of machine learning, unsupervised and supervised learning, we investigate the application of deep learning methods to tackle this challenging task and to learn robust representations of images.

We begin our investigation with the introduction of a novel unsupervised learning technique called deconvolutional networks. Based on convolutional sparse coding, we show this model learns interesting decompositions of images into parts without object label information. This method, which easily scales to large images, becomes increasingly invariant by learning multiple layers of feature extraction coupled with pooling layers. We introduce a novel pooling method called Gaussian pooling to enable these layers to store continuous location information while being differentiable, creating a unified objective function to optimize.

In the supervised learning domain, a well-established model for recognition of objects is the convolutional network. We introduce a new regularization method for convolutional networks called stochastic pooling which relies on sampling noise to prevent these powerful models from overfitting. Additionally, we show novel visualizations of these complex models to better understand what they learn and to provide insight on how to develop state-of-the-art architectures for large-scale classification of 1,000 different object categories.

We also investigate related problems in deep learning. First, we introduce a model for the task of mapping one high dimensional time series sequence onto another. Second, we address the choice of nonlinearity in neural networks, showing evidence that rectified linear units outperform others types in automatic speech recognition. Finally, we introduce a novel optimization method called ADADELTA which shows promising convergence speeds in practice whilst being robust to hyper-parameter selection.

Contents

List of Figures

List of Tables

List of Appendices

Chapter 1

Introduction

It has long been the goal of computer vision researchers to have a flexible representation of the visual world capable of recognizing objects in complex scenes. While this seems natural and simple to humans, the task of object recognition has been studied for many years with no robust solution to date. There have been successful methods that are adequate for specific classifying object categories such as faces but the problem of large-scale object recognition involving many different classes of objects remains largely unsolved.

This task is difficult due to the large number of objects in the world, the continuous set of viewpoints from which they can be viewed, and any number of external factors. These factors can include the lighting of objects in a scene, color variations for different instances of the same object, background clutter which can confuse classification, and occlusion which prevents the entire object from being seen. In addition to all these factors, there is also the possibility for several classes to look remarkably similar, a common occurrence when the number of objects to distinguish between becomes very large, or for objects in the same class to look very different such as when age, gender, version, make, etc. are present. See for example Figure 1.1 where images of dogs are shown at various poses and backgrounds with different colored fur. Despite all the dogs

Figure 1.1: An illustration of the difficulty in detecting various classes of objects. Shown from left to right are examples of a golden retriever, a Labrador retriever, a flat-coated retriever, and another Labrador retriever. Only slight variations in hair length or color set these breeds apart. We demonstrate in this thesis a model that can predict these classes correctly out of 1,000 known objects with 53%, 70%, 96%, and 63% confidence respectively despite the wide variety of pose, background, and colors of the dogs seen in these images.

being relatively similar, there are three distinct breeds shown. This can be confusing not only for a computer system, but for humans who are unfamiliar with the small details that make these breeds different. To be able to recognize a set of known objects like these in this vast world of variation seems like a daunting task. However, this is a task that even young infants can handle without much challenge, and it is also plausible that a computer system could be designed to handle all these variations and recognize objects in the visual world.

If a reliable object recognition system was designed, it could be useful in numerous applications. For instance, it could be useful for image search where understanding the content of images rather than relying on tags found in image captions could greatly improve the relevance of results. Image search based on an input image could also become feasible because only objects found in the images need to be matched. The field of surveillance could be improved by having systems looking for specific objects to be present or not in a given area. Car navigation could rely on a vision system to detect nearby cars or obstacles on the road and automatically determine how to safely position the car. These systems could also be used to aid blind people to see traffic signals, read signs, or move around new environments without additional help. These are just a few obvious examples, but with a reliable visual system capable of recognizing objects there

are many possible applications across a wide variety of domains.

In the past it has proved difficult to create hand crafted system to recognize these objects. Not only are there too many combinations of features needed to handle all the variations of the objects, this method seems like an implausible explanation for how the human visual system could work. Learning of some form takes place from the time we are born until we understand a large number of distinct objects. Therefore as the main crux of this thesis we attempt to determine what form of learning algorithm is applicable to the task of object recognition with the goal of producing a reliable system that could be useful for some of these aforementioned applications.

We chose to explore various forms of deep learning, that is, learning multiple layers of abstraction from the input images. As the number of classes, the amount of variability, and the number of training examples grow, deep learning out performs other methods. We investigate both unsupervised learning which requires no external guidance and supervised learning in which labelled examples of objects are utilized to train the model. Both forms of learning also exist in the human visual system, infants are given labels of select objects they see in the world, but can also cope with unseen objects by relating them to common things they have previously seen.

Learning structures from images without any supervision was the goal of our initial work on deconvolutional networks discussed in Chapter 3. By encouraging sparsity in the activations of an unsupervised learning method such as sparse coding [93], simple primitives such as oriented edges spontaneously emerge and resemble those found in the V1 area of the human visual pipeline. To easily build upon these primitives we first make the overall system more flexible whilst it simultaneously becoms more robust. Convolutional architectures, which share parameters at each spatial location over an image have the benefit of being equivariant to shifts in the input. That is, shifting the image in one direction creates a corresponding shift of the features responses in the same direction. This allows the representation to respond regardless of the object

position in the image. Combining the benefits of convolutions with sparse coding we show interesting edges, curves, and colors emerge while having an equivariant sparse distribution of features.

Building these simple primitives into more complex structures such as corners and curves seems like a plausible next step in building a robust object recognition system in an unsupervised way. By stacking multiple levels of sparse convolutional feature extractors we show these structures emerge sa the hierarchy ascends. A third layer on top of these mid-level features extracts more complex color variations and grating patterns similar to those proposed by Marr in the 1970's [83].

To further extend the idea of stacking convolutional sparse coding layers, we look toward convolutional networks [74] to see how those models handle invariance. Convolutional networks are essentially the inverse model to a deconvolutional network in that they apply feedforward filtering operations to images to extract interesting features. By adding a pooling layer between feature extraction layers, each higher layer feature handles complex invariances to different configurations of pixels in the input image. Incorporating similar pooling layers into our unsupervised learning approach with deconvolutional networks allows the third layer of the model to have a much larger receptive field and learn to represent parts of objects. Furthermore, an additional fourth layer in the model can learn to represent entire objects in a completely unsupervised fashion.

Having this hierarchical representation of objects proves useful for object recognition, improving performance as the hierarchy is ascended. This composition of low-level structure in multiple layers leading to high level object concepts is perhaps biologically plausible as well. V1 neurons are known to be selective to specific orientations [93] and to connect to several additional layers of processing with the final layers providing some representations of entire objects.

While the unsupervised deconvolutional networks are conceptually more ideal and can handle the nearly unlimited amounts of unlabelled data available, the classification per-

formance and inference speed does not necessarily scale well. A brief attempt at adding label information to the deconvolutional network suggested that extreme levels of sparsity were not useful for classification. This, along with recent success in large-scale object recognition by Krizhevsky *et al.* [67] using convolutional networks, led our research in the direction of these models which can take advantage of labelled information to tune the representation well for specific classes.

With the advent of rectified linear units (ReLU) [90], relatively sparse distributions can still be achieved in a convolutional network at a fraction of the computational cost of inference in a deconvolutional network. Since these rectified linear units are used in conjunction with a supervised objective function, the classification performance does not suffer from large amounts of sparsification as was the case with deconvolutional networks trained in an unsupervised fashion. By combining these ReLUs with a novel stochastic pooling procedure we show state of the art performance across a number of common benchmarks. This stochastic pooling provides a hyper-parameter free way of regularizing a large convolutional network such that it does not memorize or overfit the training data.

This approach was applied to several small scale object recognition benchmarks where the number of classes ranged up to 100 and the number of training examples was in the 10k regime. This is far from what even young children have been exposed to and likely not how the powerful visual system in our brains is trained. Therefore, we moved in the direction of more classes and data to tackle the ImageNet challenge for large-scale object recognition with 1000 classes and over 1 million training examples. Building a system for this dataset was challenging. By using deconvolutional networks as a tool to visualize the inner workings of a large convolutional network we achieved state-of-the-art results on the ImageNet benchmark as well as several other datasets. These visualizations also help understand the invariances the model can cope with and with collaboration with the neuroscience community could potentially lead to a better understanding of our visual systems.

The above work constitutes the main body of this work. In addition to investigating learning methods for object recognition, we also explore several other interesting problems in machine learning in the second part of this thesis. One such problem is that of facial expression transfer where the goal is to convert an input sequence of facial motions into a output sequence of a different person's corresponding motions. We derived a novel approach to this problem and surpassed existing methods in this useful task which can be applied directly to cartoon character animation.

Another problem that benefits from the use of rectified linear units is that of automatic speech recognition. We show faster training times and better generalization abilities for large scale speech recognition when using these types of nonlinearities. This work formed the basis for the production systems being used in various speech recognition products at Google.

Finally, we investigate the fundamental problem of optimization in machine learning and introduce a novel update rule. The proposed method is insensitive to hyperparameter settings and model architectures while providing fast convergence that is robust to large gradient magnitudes. We hope this method can be applied to many more problems in the future to improve convergence and generalization throughout machine learning.

Chapter 2

Background Literature

This chapter provides an outline of the basic principles upon which the remaining chapters are based with the aim of making this thesis self contained. The chapter also introduces some of the notation that will be used throughout this thesis.

The work in this thesis brings together previous methods from two major fields, computer vision and machine learning. Therefore this chapter is broken into two sections to describe some of the most common approaches used in each field. Additionally, background information about convolutions, a mathematical operation used throughout this thesis, is included in Appendix A.1.

2.1 Computer Vision

The field of computer vision has the broad goals of representing natural images in a way that can be useful for segmentation, object classification, object detection, image denoising and image de-blurring to name a few tasks. While all these tasks are seemingly disjoint, having a good underlying representation of an image is a common factor in all of them and relates to the work presented in this thesis. The entire field of computer vision is too broad to cover in depth, including a large body of work on hand-engineering

visual features which is not the focus of this thesis. Below are some prior art on learning representations in computer vision which relate to this thesis.

2.1.1 Sparse Coding

The goal of sparse coding is to learn an overcomplete set of basis functions that describe the input data [93]. Using a small number of such basis functions to reconstruction each input vector requires a sparse distribution of coefficients or features activations z, one for each basis function. For a given input vector, x, of N_d dimensions, the cost function C that sparse coding attempts to optimize is:

$$C = \frac{\lambda}{2}||Wz - x||_2^2 + |z|_1 \tag{2.1}$$

where W is a $N_d \times N_k$ matrix of basis functions and z are the feature activations as a $N_k \times 1$ vector for each example. To be over-complete N_k is chosen to be $> N_d$. Note, for simplicity here and throughout this thesis we omit summing the cost function over all examples x in the dataset $\mathcal{X}$. In practice the goal is to minimize the cost over the entire dataset and optimization may proceed using one example at a time, in mini-batches, or in full batch (entire dataset). Variables are defined as either being shared over the entire dataset, for example W in Eq. 2.1 or as being per-example variables such as x and z in Eq. 2.1. For a full list of notation see Appendix A.1 where all the methods mentioned in this thesis share a common set of notation in order to show their similarities.

This cost function is composed to two terms, the first being a reconstruction term that enforces the reconstruction from the inferred features, z, using the learned weight matrix W to be close to the inputs x for each data case. The second term encourages the inferred feature maps to become sparse in the ℓ_1 sense as a proxy for optimizing the true ℓ_0 sparsity which is non-differentiable [129].

To optimize this cost function, typically there is a alternating procedure between 1)

updating the weights W that are shared between all examples given each x and z, and 2) inferring the feature activations z for each example x given the current setting of the weights W.

In terms of updating the weights, gradient descent [106] is often employed [93]:

$$W = W - \eta \frac{\partial C}{\partial W} = W - \eta(Wz - x)z^T \tag{2.2}$$

where η is a learning rate chosen by hand. More advanced conjugate gradient methods can also be used to speed up the converge, reducing the number of optimization updates needed for each weight update [115].

Given a setting of the weights, the feature activations must be inferred. Many approaches utilize an alternating optimization approach such as iterative shrinkage and thresholding algorithm (ISTA) [18]. This alternates taking a gradient step to optimize the reconstruction term:

$$z' = z - \eta \left(\frac{\partial C}{\partial z} \right)_{recon} = z - \eta W^T(Wz - x) \tag{2.3}$$

followed by a step for the ℓ_1 sparsity term which has a closed form shrinkage operation:

$$z = max(0, z' - \lambda\eta) \tag{2.4}$$

This simple algorithm is fairly effective at optimizing the cost function in Eq. 2.1, however several other optimization algorithms have been designed for minimizing this sparse coding objective. Fast iterative shrinkage-thresholding algorithm (FISTA) [4] is the most immediate extension to the ISTA algorithm and changes the convergence properties to be $O(1/t^2)$ rather than the $O(1/t)$ convergence rate of ISTA, where t is there number of iterations. Additional efficient optimization methods for sparse coding include: atomic

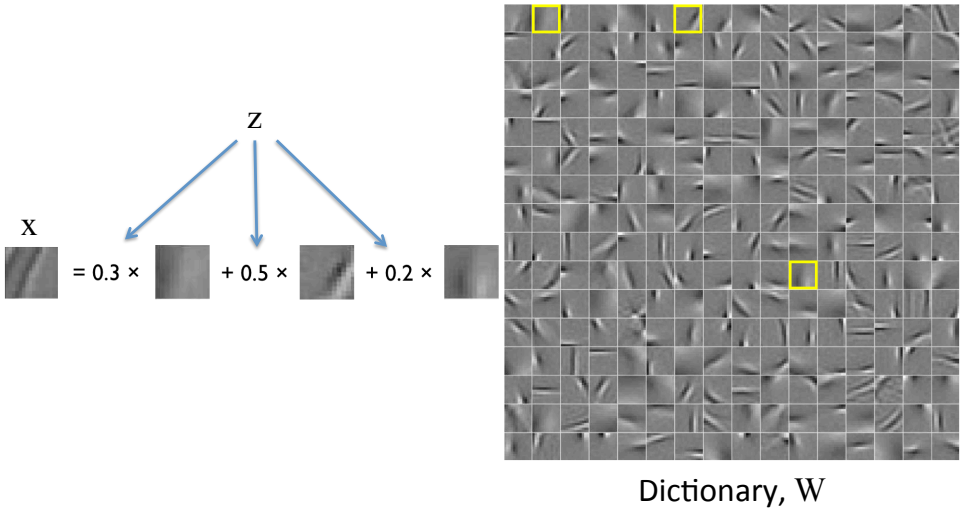

Figure 2.1: Sparse coding of an input patch x. Shown here are a few non-zero elements of the feature activations z which create a weighed sum reconstruction using the corresponding dictionary elements from W. Reproduced from Kavukcuoglu *et al.* [61].

decomposition by basis pursuit [20], continuation methods used by Geman [40] and Wang *et al.* [138], the feature-sign algorithm by Lee *et al.* [77], and the online primal-dual convex optimization toolbox by Mairal *et al.* [81, 82]. Regardless of the optimization method used, sparse coding has been shown to learn Gabor features when applied to patches of image, similar to those found in the V1 region of visual cortex [93] as shown in Figure 2.1.

A modification of sparse coding that can be helpful in computer vision is that of non-negative sparse coding [51, 52] or matrix factorization [76]. In this approach the feature activations z are constrained to be in the non-negative range, making the reconstruction model to be far simpler. This often has the benefit of the representation being easier to interpret as the basis function decomposition that results cannot arbitrarily create new solutions by subtracting other basis functions as we will show in later chapters. This may also be related to the success of rectified linear units (ReLU) [90] which have shown significant performance gains on feedforward models as we show in Chapter 10.

Two applications of the sparse coding method are: 1) image de-noising in which the goal is the recover a clean image given a noisy image corrupted by some process such as sensor noise, and 2) image de-blurring in which the goal is the recovery of a sharp image

given a blurry image typically created due to camera motion.

Sparse coding for image de-noising relies on using only a small number of feature activations such that the sparse reconstruction of the input signal cannot include all the detailed noise. Therefore, if a noisy input image is split into small patches and for each patch a sparse feature representation is obtained, the reconstruction of each patch will be a cleaner version of that portion of the image [54, 81].

In terms of de-blurring images, sparse coding can be used in a slightly different context. Fergus *et al.* [35] introduced the idea of decomposing a blury image into a sharp image and blur kernel while imposing a prior on the gradients of the image. Krishnan *et al.* [64] extended this approach to use sparse coding with hyper-Laplacian priors (instead of ℓ_1) to further tease apart the blur kernel and sharp image simultaneously given just the blurry image.

While sparsity alone produces interesting feature decompositions, it is not necessarily useful directly for classification, [102, 103]. As the sparseness of a solution increases, the discriminative information that remains in the few activations seems to be limited as we show in Section 4.4.5 of this thesis. Therefore it may be advantageous to combine sparse coding into a larger classification pipeline, as a regularization technique or use it simply as pre-training for other models.

2.1.2 Spatial Pyramid Matching

To build a better representation of an entire image we can combine low-level representations together. Low-level image features such as SIFT [80], HOG [26] or DAISY [139] have been shown to be useful for tasks such as image matching and retrieval [15]. These hand-crafted descriptors are designed to be slightly invariant to rotation, scale, translations, and brightness changes to varying degrees. Each feature is composed of an orientation binning stage that groups similarly oriented image structures together within local regions followed by a normalization stage that makes the resulting descriptor slightly

invariant to brightness variations. While these descriptors are sometimes useful for matching tasks, they only represent a small image region which is alone not very useful for object recognition.

To attempt to make low-level descriptors useful for object recognition, Lazebnik *et al.* [70] introduced the Spatial Pyramid Matching (SPM) method. This method has become the de facto standard method in computer vision for object recognition and has produced several state of the art results in the past. The first step of this method is to compute SIFT or HOG descriptors densely over over an image. Then a dictionary learning method such as K-means is used to learn a dictionary over these descriptors, representing each by a $1-of-N_k$ sparse feature vector. The third step involves computing histograms of neighboring feature vectors over multiple levels of regular grids placed over the image, forming a pyramid of grids from fine to coarse scale as shown in Figure 2.2. Finally, there is a concatenation step in which the histogram results from each grid in the pyramid are concatenated into a single vector per image and a Support Vector Machine (SVM) is trained in order to discriminate between these vectors.

The intuition behind the method is that the low-level hand-crafted descriptors are adequate for removing small variations in the input pixels. By quantizing these over a learned dictionary, they can be represented concisely knowing the N_k dictionary atoms. Then pooling on the pyramid of regions allows for objects to be recognized at multiple scales potentially and creates a high dimensional input making it easier for the SVM to discriminate examples.

Many SPM variants exist in which either the coding method or the pooling stage is varied. Replacing K-means (a form of hard partitioning) with sparse coding (soft partitioning) produces noticeable performance gains [11, 137, 141]. Combining local descriptors like SIFT into large more complex descriptors such as the macro-features of Boureau *et al.* [11] also improves performance. These essentially code larger image structures and are therefore more discriminative in practice.

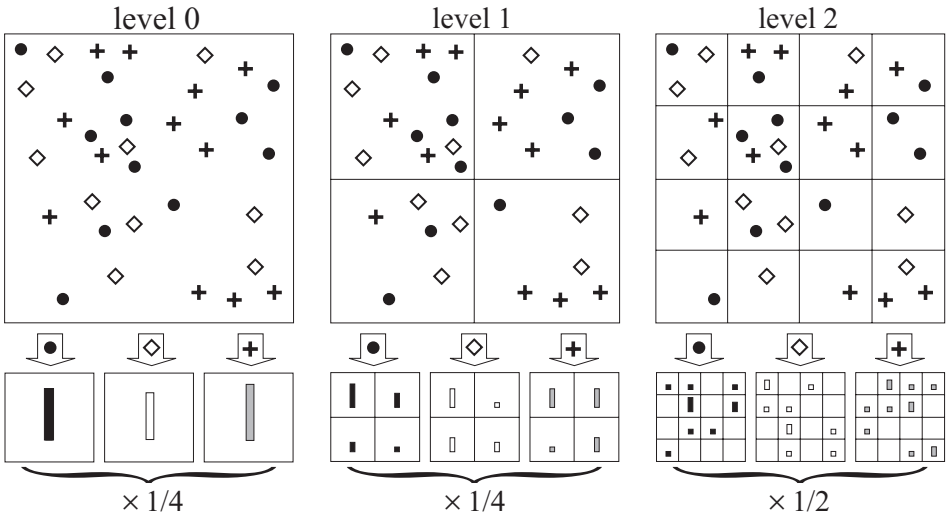

Figure 2.2: Spatial pyramid matching example where each type of marker represents a different dictionary atom determined by clustering SIFT features which are computed densely on the input image. These selected atoms are histogrammed over regular grids on the image and the resulting histograms are concatenated to create the final representation per image. Reproduced from Lazebnik *et al.* [70].

The histogram procedure in the canonical SPM approach is analogous to average pooling over the regions of the spatial pyramid. This is evident when the $1 - of - N_k$ sparse feature vector at each of the dense descriptor locations is considered as a set of binary activations at that location in 2-D feature maps. Replacing this average pooling with max pooling is known to improve performance [12], analogous to using max pooling in neural networks which shows improvements [56]. This pooling procedure can also be made more sophisticated by grouping the features to be pooled based on clustering in the input space (known as Locally-constrained Linear Coding) [137] or by learning the pooling regions instead of using regular grids [58].

2.1.3 Hierarchical Models

While sparse coding alone can learn interesting features from small image patches, it is the goal of hierarchical models to learn various levels of abstraction from an entire input image. Recent performance enhancements have been produced by Bo *et al.* [10] using

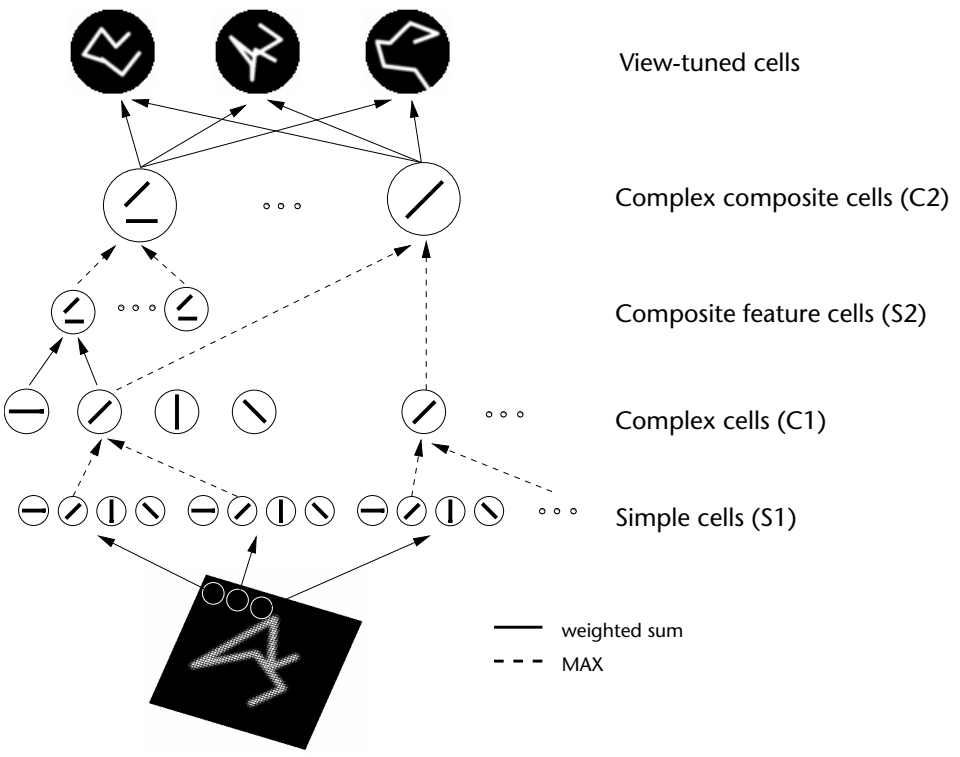

Figure 2.3: HMAX model consisting of layers of simple and complex cells. Simple cells represent a template matching phase, whereas complex cells represent an aggregation function to provide invariance. Reproduced from Riesenhuber and Poggio [101].

what is called multipath sparse coding. The idea is based on two extensions to SPMs. The first is to have different paths take as input different sized input patches. The second is to stack the sparse coding feature extractors in layers with pooling functions in between layers to provide invariance.

A wide range of other hierarchical image models have been proposed in computer vision. Relevant work includes that of Zhu and colleagues [131,145], in particular Guo *et al.* [43]. In these approaches, edges are composed using a hand-crafted set of image tokens into larger image structures. Grouping is performed via basis pursuit with intricate splitting and merging operations on image edges. The stochastic image grammars of Zhu and Mumford [145] also use fixed image primitives, as well as a complex Markov Chain Monte-Carlo (MCMC)-based scheme to parse scenes into object parts.

Zhu *et al.* [144] propose a top-down parts-and-structure model but it only reasons about

image edges, as provided by a standard edge detector instead of operating on pixels directly. The biologically inspired HMax model of Serre *et al.* [101, 114] uses exemplar templates in their intermediate representations, rather than learning conjunctions of edges (see Figure 2.3). Fidler and Leonardis [36, 37] propose a top-down model for object recognition which has an explicit notion of parts whose correspondence is reasoned about at each level. Hierarchical models have also been applied to other tasks than generic object recognition. Amit and Geman [2] and Jin and Geman [60] apply hierarchical models to deformed Latex digits and car license plate recognition. All of these hierarchical models show some resemblance to the work presented in this thesis.

2.2 Machine Learning

The field of machine learning is far too broad to cover in depth here. Therefore, we focus on a small portion of interesting neural network models and techniques that have influenced this work.

2.2.1 Autoencoders

One of the simplest neural networks for unsupervised learning is known as the autoencoder [13,106] (see Figure 2.4). The idea of this approach is to process an input example through multiple layers of a neural network in order to reconstruct the original input example at the output of the network. A given input x is mapped through neural net layers (sometimes referred to as a multi-layer perceptron (MLP)), each layer consisting of a matrix of weights W and biases b along with some form of nonlinearity such as a sigmoid function, tanh function, or rectified linear unit [90]. We refer to the features before the nonlinearity as z to be consistent with other notation and the activations after the nonlinearity as h. The output of each layer is:

$$z = Wx + b \Rightarrow h = \sigma(z) \tag{2.5}$$

where σ here is the sigmoid function but could be tanh, relu or other activation function. The target y for the autoencoder is simply the input x and the object to minimize is the squared distance between the output of the last layer $\hat{y} = h_l$ and the targets:

$$C = \sum_i (\hat{y}_i - y_i)^2 \tag{2.6}$$

Autoencoders can also be stacked and trained layer by layer. This is done by treating the activations of the layer below as input and targets for the current layer. Once all layers have been trained in this fashion, the entire network can be fine-tuned to reconstruct the original input as a deep autoencoder [133]. This can be a useful method for pretraining general deep networks [7] or for creating a compact codes from high dimensional data [50].

Several variations of the autoencoder have been proposed. Denoising autoencoders

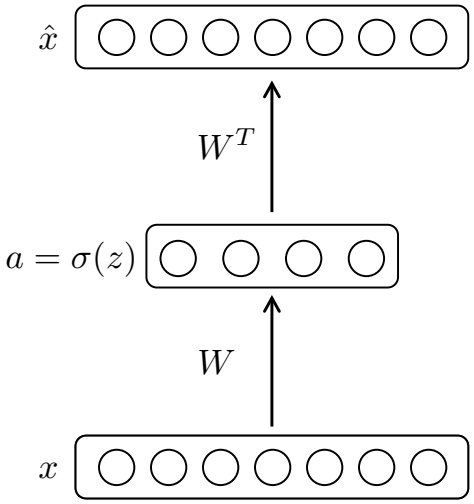

Figure 2.4: Autoencoder taking as input x, encoding it through weights W to get a set of feature responses z which are mapped through a nonlinearity σ to get activations h. These are input to higher layers which eventually output a reconstruction of the original input as $\hat{x}$. Note: biases b are omitted from this figure.

improves generalization by corrupting the input x while keeping the targets y noise-free [133,134]. This provides the autoencoder with many perturbations of the input and requires it to learn a mapping back to the original data. A similar approach is used in the transforming autoencoder [47], with a slightly different goal. An input image is shifted or modified with an affine transformation in a known way and the autoencoder must predict both the presence of a template feature and the transformation that the template must have gone through in order to reconstruct the input image. This makes the network robust to many variations in the input while being able to output pose parameters for each template it can detect. This is analogous to the work by Ranzato *et al.* [99] in which an autoencoder outputs pooling locations in order to place reconstructions from pooled features back in the optimal locations. This makes their autoencoder translation invariant and we show storing these locations is integral for reconstructing through multiple layers of a hierarchy in Chapter 4.

A similar approach to autoencoders for unsupervised learning is called predictive sparse decomposition (PSD) [61,96,97,100]. Traditionally, sparse coding is an expensive process as many iterations are needed to converge to a sparse solution. With PSD, the goal is to learn a fast approximate encoder for the solution of traditional sparse coding problems. This is analogous to the feedforward portion of a traditional autoencoder except the encoder and decoder are decoupled with a latent feature variable acting as a centerpiece for all terms of the object function. The three-fold objective is to reconstruct the image accurately from these features z through decoder matrix W_{dec}, make the features sparse as in sparse coding with an ℓ_1 prior, and have the feed-forward encoder weight W_{enc} produce outputs close to z:

$$C = \frac{1}{2}||W_{dec}z - x||_2^2 + \frac{1}{2}||W_{enc}x - z||_2^2 + \lambda|z|_1 \tag{2.7}$$

This algorithm learns interesting complex features in an unsupervised fashion with the benefits of avoiding the expensive sparsifying procedure.

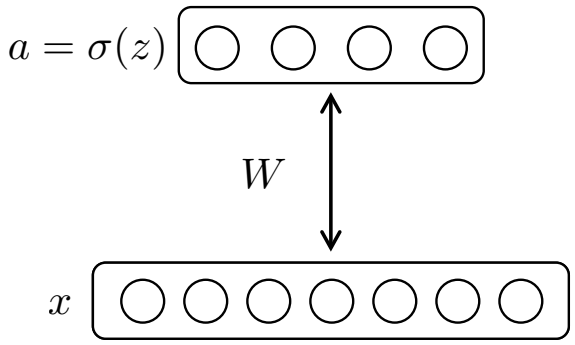

Figure 2.5: A Restricted Boltzmann Machine which has a joint distribution between visible units x and hidden unit activations h. The bidirectional weight matrix W maps between the bipartite set of units produce feature responses z that are mapped through a nonlinearity h before mapping back to the inputs through W^T.

2.2.2 Restricted Boltzmann Machines

A related model to autoencoders is called a Restricted Boltzmann Machine (RBM) [45, 119]. This is an extension of Boltzmann machines [1] in which the term restricted refers to the lack of connections between hidden units. This forms a bipartite graph between visible units x and hidden unit activations h (traditionally v and h in previous work) as shown in Figure 2.5. This generative model is governed by the following energy function to be minimized:

$$E(x,h) = -\sum_i {b^x}_i x_i - \sum_j {b^h}_j h_j - \sum_i \sum_j h_j W_{i,j} x_i \tag{2.8}$$

where b^x and b^h are the biases for the input units and hidden units respectively. This energy function can be interpreted as a joint probability to be maximized:

$$P(x,h) = \frac{e^{-E(x,h)}}{\sum_x \sum_h e^{-E(x,h)}} \tag{2.9}$$

where the denominator is the partition function which sums over all possible configurations. Training with an RBM typically uses an algorithm called contrastive diver-

gence [17, 45] where the goal is essentially to push down on the energy surface near data examples and push up on other points in the space.

A nice property of RBMs compared to other methods such as sparse coding is that inference is fast, involving a single feedforward encoding:

$$h = \sigma(z) = \sigma(Wx + b^h) \tag{2.10}$$

where σ is typically a logistic function which is sampled to create binary hidden unit activations.

RBMs have been used to initialize deep models in what is termed greedy layer-wise pretraining [7, 48, 49]. This approach is greedy in that the first layer is trained initially with no consideration of what will be trained on top. Once a good representation is learned in the first layer, an additional layer can be added on top which takes as input the feed-forward activations of the first layer for each input example. At this stage, the first layer in not trained and is simply used to produce training examples for the second layer. This procedure can continue for as many layers as desired. Afterwards, it is common practice to unroll the entire network and fine-tune all layers together with backpropagation of an objective function placed on top [7, 49]. This is beneficial when labelled training examples are scarce, as the unsupervised greedy pretraining can learning a good generative model on unlabelled data, which helps initialize a supervised training algorithm. This also provides an interesting learning algorithm for creating compact representations of the inputs [50].

RBMs have been extended in many ways since their introduction. Temporal versions [124, 128] can be applied to sequential data by conditioning on the past time steps to generate the current step. This produces a model exponentially more powerful than a hidden markov as the hidden state in a conditional RBM is distributed over all hidden units. We use an extension of this model in Chapter 9 to map from one temporal sequence

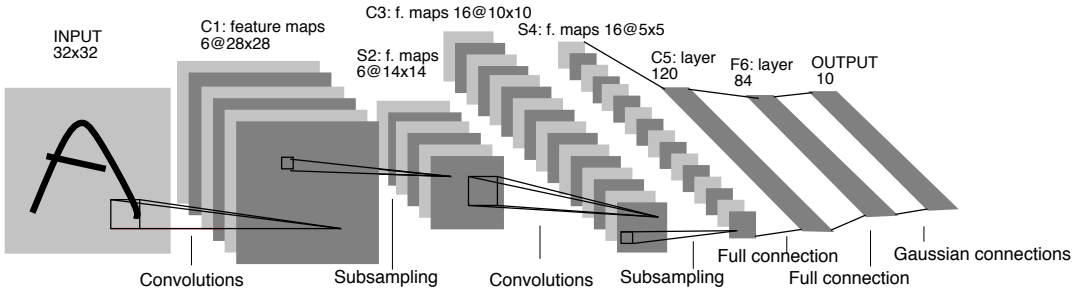

Figure 2.6: Convolutional network (LeNet5 [74]) composed of multiple alternating layers of convolutions and subsampling followed by a few fully connected layers and finally output units for discriminating between 10 classes of handwritten digits. Reproduced from LeCun *et al.* [74].

to another. Collaborative filtering is another application of RBMs [109] where filling in missing information conditioned on known information is critical. Furthermore, RBMs can be trained with a discriminative objective in addition to the generative objective in order to improve classification performance [69].

Deep Boltzmann machines (DBMs) [108] are a similar model to stacked RBMs except all the connections between layers are bidirectional. This allows DBMs to be jointly trained instead of doing greedy layer wise training. DBMs provide a powerful generative framework for high level reasoning of object classes [110] and for modelling complex object part relationships in natural images [32].

2.2.3 Convolutional Networks

The inspiration for much of the work in this thesis is the convolutional neural network (CNN) created by LeCun *et al.* [74] (see Figure 2.6). Images exhibit many spatial relationships between neighboring pixels that should not be affected by their location within the image [39]. To incorporate that into the model, a convolutional network learns a set of N_k filters $F = \{f_1, \ldots, f_{N_k}\}$ which are convolved with input image x to produce a set of N_k 2D features maps z:

$$z_k = f_k \otimes x \tag{2.11}$$

where $\otimes$ is the convolution operator. The learned filters are analogous to templates that are matched to every possible patch of the input image. When the filter correlates well with a region of the input image, the response in the corresponding feature map location is strong. The convolution operation has many beneficial properties: 1) weight sharing over the entire image reduces the number of parameters per response 2) local connectivity learns correlations between neighboring pixels, and 3) equivariance (i.e. an object shifted in the input image will simply shift the corresponding responses in a similar way).

Typically the convolutional responses are passed through a non-linear activation function such as sigmoids, tanh, or rectified linear units (ReLU) [90] to produce activation maps denoted h. Following this activation function, a pooling layer is typically present to provide invariance to slightly different input images. This pooling involves executing some operation P, typically average or max, over the activations within a small spatial region $\mathcal{G}$ of each map of activations:

$$p_{\mathcal{G}} = \max_{i \in \mathcal{G}} h_i \tag{2.12}$$

where i indexes the activations within the pooling region. Typically max pooling is preferred [12, 56] as it avoids cancellation of negative elements and prevents blurring of the activations and gradients throughout the network since the gradient is placed in a single location during backpropagation. After multiple convolutional and pooling layers, a convolutional network typically has one or more fully connected neural net layers with weights W and biases b before the final classifier. The entire network is trained with back-propagation of a supervised loss such as the cross entropy of a softmax classifier output and the target labels y represented as a $1-of-N_c$ vector where N_c is the number of classes to discriminate between:

$$C = -\sum_{c}^{N_c} y_c log(\hat{y}_c) \tag{2.13}$$

where $\hat{y}_c$ are the activations of the $l-th$ layer of the model h^l pushed through a softmax function:

$$\hat{y}_i = \text{softmax}(h_i^l) = \frac{e^{h_i^l}}{\sum_j e^{h_j^l}} \tag{2.14}$$

Other variants of convolutional networks have been proposed, such as tiled convolutional networks [71] which reduce the computations needed for convolutions by using a strided convolution. Also, multi-column convolutional networks which improve generalization abilities through model averaging of multiple networks [22].

Convolutional networks have been used to tackle many practical object recognition applications. For instance, they have been successful applied to recognizing digits and characters in documents [74] and house numbers [113]. Recently, fast computation libraries [23, 66] on GPUs have enabled large scale training of convolutional networks. Large convolutional networks have been used to produce impressive gains over state of the art computer vision techniques in the ImageNet challenge [67].

2.2.4 Other Convolutional Models

Many of the existing approaches discussed above have also been converted to the convolutional domain for image processing. The immediate advantages of being equivariant are crucial for a good image model. First, autoencoders have recently been extended to convolutional autoencoders [85] to encode simple shift invariances directly into the model. Stacking these convolutional autoencoders seems to learn interesting features when pooling is used and they can outperform CNN's on some benchmarks.

The HMAX model [101, 114] incorporates simple cells via convolution and complex cell

via max pooling inspired biological findings in the visual cortex. The predictive sparse decomposition (PSD) algorithm has been extended to learning convolutional features [62] in which corners, crosses, curves, circles, Gabors, etc. emerge from a single layer of encoding.

To extend convolutional models into the unsupervised training regime with RBMs Lee *et al.* [78] extended the work of Hinton *et al.* on Deep Belief Networks (DBN) [50] to be convolutional. The matrix multiplications in each RBM layer of a DBN are replaced with 2D convolutions. Additionally, a probabilistic form of max pooling is used to generate from each pooling region while providing invariance to the feedforward inference. This shows promising results in learning mid-level features such as curves and circles directly from pixels. Additionally, this model has been applied on top of SIFT vectors to learn discriminative features for classification [121].

2.2.5 Regularization Approaches

Neural networks are powerful models that given the opportunity can memorize training data. If a model has enough parameters, the performance on the training data typically converges towards perfection while the test performance degrades [89]. This well known side effect of training is known as overfitting and there have been many proposed approaches to overcome this.

Weight decay, weight tying or data augmentation are some common approaches. Weight decay is the simplest approach, which adds a term to the cost function to penalize the parameters in each dimension preventing it from exactly modelling the training data and therefore help generalize to new examples. Weight tying has already been shown in convolutional networks to reduce the number of parameters while still allowing models to learn good representations of the input data. Data augmentation is a method of boosting the size of the training set so that the model cannot memorize all of it. This may take several forms, for example elastic distortions have been applied on the

MNIST handwritten digit recognition challenge to generate additional realistic looking digits [23, 75, 116].

Translating an image is an integral part of the transforming autoencoder [47] and is also useful for inputs to other models [65, 136]. By taking small randomly located crops out of the original image and treating that as input to a network, the model sees a large variation of inputs that are labelled with the original class. This technique is essential in the impressive ImageNet results by Krizhevsky *et al.* due to the large number of parameters in their approach [67].

Another regularization technique is to encourage activations to have a sparse prior so the all the information about the training set cannot be captured in the representation. Several approaches have been proposed to induce this sparsity such as sparsifying logistic activations [100], a target sparsity penalty for each activation [125], or separating out activations into spike and slab units [25].

Finally, recent success has been shown with a regularization technique called Dropout [44]. The idea is to randomly set half of the activations in each hidden layer (and a smaller fraction of input dimensions) to 0. By doing this, the hidden units cannot co-adapt to each other being active, therefore they must learn a better representation of the input that generalizes well. Additionally, at test time, the hidden units are simply multiplied by the rate at which they are kept which acts like a form of model averaging over all possible instantiations of the model. Extensions to this method have been proposed where the weights, not the activations, are dropped [136], showing this to be an even better regularizer in certain cases.

Part I

Hierarchical Image Models

Chapter 3

Deconvolutional Networks

3.1 Introduction

Building robust low and mid-level image representations, beyond edge primitives, is a long-standing goal in vision. Many existing feature detectors spatially pool edge information which destroys cues such as edge intersections, parallelism and symmetry. In this chapter we introduce *deconvolutional networks* (DN), a framework that permits the unsupervised construction of hierarchical image representations. These representations can be used for both low-level tasks such as denoising, as well as providing features for object recognition. This forms the basis for the following two chapters which extend this early work on deconvolutional networks.

In a deconvolutional network, each level of the hierarchy groups information from the level beneath to form more complex features that exist over a larger scale in the image. Our grouping mechanism is sparsity: by encouraging parsimonious representations at each level of the hierarchy, features naturally assemble into more complex structures. However, as we demonstrate, sparsity itself is not enough – it must be deployed within the correct architecture to have the desired effect. We adopt a convolutional approach since it provides stable equivariant latent representations at each level which preserve

locality and thus facilitate the grouping behavior. Using the same parameters for learning each layer, our deconvolutional network can automatically extract rich features that correspond to mid-level concepts such as edge junctions, parallel lines, curves and basic geometric elements, such as rectangles. Remarkably, some of them look very similar to the mid-level tokens posited by Marr in his primal sketch theory [84] (see Figure 3.1).

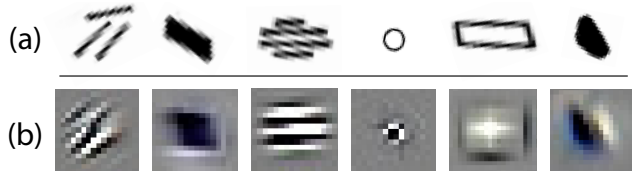

Figure 3.1: **(a)**: "Tokens" from Fig. 2-4 of *Vision* by D. Marr [84]. These idealized local groupings are proposed as an intermediate level of representation in Marr's primal sketch theory. **(b)**: Selected filters from the 3rd layer of our deconvolutional network, trained in an unsupervised fashion on real-world images.

Our proposed model is similar in spirit to the convolutional neural networks (CNN) of LeCun *et al.* [74], but quite different in operation. Convolutional networks are a bottom-up approach where the input signal is subjected to multiple layers of convolutions, non-linearities and sub-sampling. By contrast, each layer in our deconvolutional network is top-down; it seeks to generate the input signal by a sum over convolutions of the feature maps (as opposed to the input) with learned filters (see Figure 3.2). Given an input and a set of filters, inferring the feature map activations requires solving a multi-component deconvolution problem that is computationally challenging. In response, we use a range of tools from low-level vision, such as sparse image priors and efficient algorithms for image deblurring. Correspondingly, our work is an attempt to link high-level object recognition with low-level tasks like image deblurring through a unified architecture.

Deconvolutional networks are closely related to a number of "deep learning" methods [7, 48] from the machine learning community that attempt to extract feature hierarchies from data. Deep Belief networks (DBNs) [48] and hierarchies of sparse auto-encoders [56, 100, 133], like our approach, greedily construct layers from the image upwards in an unsupervised fashion. In these approaches, each layer consists of an encoder

and decoder[1]. The encoder provides a bottom-up mapping from the input to latent feature space while the decoder maps the latent features back to the input space, hopefully giving a reconstruction close to the original input. Going from the input directly to the latent representation without using the encoder is difficult because it requires solving an inference problem (multiple elements in the latent features are competing to explain each part of the input). As these models have been motivated to improve high-level tasks like recognition, an encoder is needed to perform fast, but highly approximate, inference to compute the latent representation at test time. However, during training the latent representation produced by performing top-down inference with the decoder is constrained to be close to the output of the encoder. Since the encoders are typically simple non-linear functions, they have the potential to significantly restrict the latent representation obtainable, producing sub-optimal features. Restricted Boltzmann Machines (RBM), the basic module of DBNs, have the additional constraint that the encoder and decoder must share weights. In deconvolutional networks, there is no encoder: we directly solve the inference problem by means of efficient optimization techniques. The idea is that by computing the features exactly (instead of approximately with an encoder) we can learn superior features.

Most unsupervised deep learning architectures are not convolutional, but work by Kavukcuoglu *et al.* [62] on convolutional PSD has been shown to learn interesting convolutional features with fast encoders and Lee *et al.* [78] has extended DBNs to be convolutional which learns high-level image features for recognition. This convolutional DBN is most similar to our deconvolutional network, with the main difference being that we use a decoder-only model as opposed to the symmetric encoder-decoder of the RBM layers.

Our work also has links to recent work in sparse image decompositions, as well as hierarchical representations. Lee *et al.* [77] and Mairal *et al.* [81, 82] have proposed efficient schemes for learning sparse over-complete decompositions of image patches [93], using a convex ℓ_1 sparsity term. We however perform sparse decomposition over the whole

[1]Convolutional networks can be regarded as a hierarchy of encoder-only layers [74].

image at once, not just for small image patches. As demonstrated by our experiments, this is vital to learning rich features. The key to making this work efficiently is to use a convolutional approach.

A range of hierarchical image models have been proposed. Particularly relevant is the work of Zhu and colleagues [131, 145], in particular Guo *et al.* [43]. Here, edges are composed using a hand-crafted set of image tokens into large-scale image structures. Grouping is performed via basis pursuit with intricate splitting and merging operations on image edges. The stochastic image grammars of Zhu and Mumford [145] also use fixed image primitives, as well as a complex Markov Chain Monte-Carlo (MCMC)-based scheme to parse scenes. Deconvolutional networks differ in two important ways: first, we learn image tokens completely automatically. Second, our inference scheme is far simpler than either of the above frameworks.

Zhu *et al.* [144] propose a top-down parts-and-structure model but it only reasons about image edges, as provided by a standard edge detector, unlike DNs which directly operates on pixels. The biologically inspired HMax model of Serre *et al.* [101, 114] use exemplar templates in their intermediate representations, rather than learning conjunctions of edges as we do. Fidler and Leonardis [36, 37] propose a top-down model for object recognition which has an explicit notion of parts whose correspondence is explicitly reasoned about at each level. In contrast, DNs approach simply performs a low-level deconvolution operations at each level, rather than attempting to solve a correspondence problem.

3.2 The Deconvolutional Network

We first consider a single deconvolutional network layer applied to an image. This layer takes as input an image x, composed of ${N_k}^0$ color channels $x = \{x_1, \ldots, x_{{N_k}^0}\}$. We represent each channel x_c of this image as a linear sum of ${N_k}^1$ latent feature maps

$z_k^1 = \{z_1^1, \ldots, z_{N_k{}^1}^1\}$ specific to this image [2] that are convolved with filters $f_{c,k}^1$ learned from a dataset of images $\mathcal{X}$:

$$\hat{x}_c^1 = \sum_{k=1}^{N_k{}^1} f_{c,k}^1 \otimes z_k^1 \tag{3.1}$$

where $\otimes$ denotes the convolution operator and $\hat{x}_c^1$ the resulting reconstruction from the layer 1 feature maps. If x_c is an $N_{x_{rows}} \times N_{x_{cols}}$ image and the filters are $N_{F_{rows}} \times N_{F_{cols}}$, then the latent feature maps are $N_{z_{rows}} \times N_{z_{cols}} = (N_{x_{rows}} + N_{F_{rows}} - 1) \times (N_{x_{cols}} + N_{F_{cols}} - 1)$ in size, i.e. a valid convolution is used for reconstruction. But Eq. 3.1 is an under-determined system, so to yield a unique solution we introduce a regularization term on z_k^1 that encourages sparsity in the latent feature maps. This gives us an overall cost function of the form:

$$C^1 = \frac{\lambda^1}{2} \sum_{c=1}^{N_k{}^0} \| \sum_{k=1}^{N_k{}^1} f_{c,k}^1 \otimes z_k^1 - x_c \|_2^2 + \sum_{k=1}^{N_k{}^1} |z_k^1|^\alpha \tag{3.2}$$

where we assume Gaussian noise on the reconstruction term and some sparse norm α for the regularization. Note that the sparse norm over all feature maps $|z^1|^\alpha$ is actually the α-norm on the vectorized version of matrix z^1, i.e. $|z^1|^\alpha = \sum_{k,i,j} |z_k^1(i,j)|^\alpha$ (where i and j index the spatial locations in z). Typically, $\alpha = 1$, although other values are possible, as described in Section 3.2.2. λ^1 is a constant that balances the relative contributions of the reconstruction of x and the sparsity of the feature maps z_k^1. To further simplify notation throughout this work, we replace the sum of convolutions with matrix notation as:

$$\hat{x}^1 = F^1 z^1 \equiv \sum_{k}^{N_k{}^1} f_{c,k}^1 \otimes z_k^1 \;\; \forall c \in N_k{}^0 \tag{3.3}$$

[2] The superscript 1 denotes layer 1 of the model, whereas superscript 0 in $N_k{}^0$ refers to the input image. Multiple layers will be described in later sections.

where we have been explicit to denote variables by their layer using the superscript 1 in this case. The collection of filters planes is denoted F^1 where $F^1 = \{f^1_{c,k}\ \forall k, c\}$ This makes the overall cost function from Eq. 3.2 much simpler:

$$C^1 = \frac{\lambda^1}{2}\|F^1 z^1 - x\|_2^2 + |z^1|^\alpha \tag{3.4}$$

Note that our model is top-down in nature: given the latent feature maps, we can synthesize an image. But unlike the sparse auto-encoder approach of Ranzato *et al.* [97], or DBNs [48], there is no mechanism for generating the feature maps from the input, apart from minimizing the cost function C^1 in Eq. 3.4. Many approaches focus on bottom-up inference, but we concentrate on obtaining high quality latent representations.

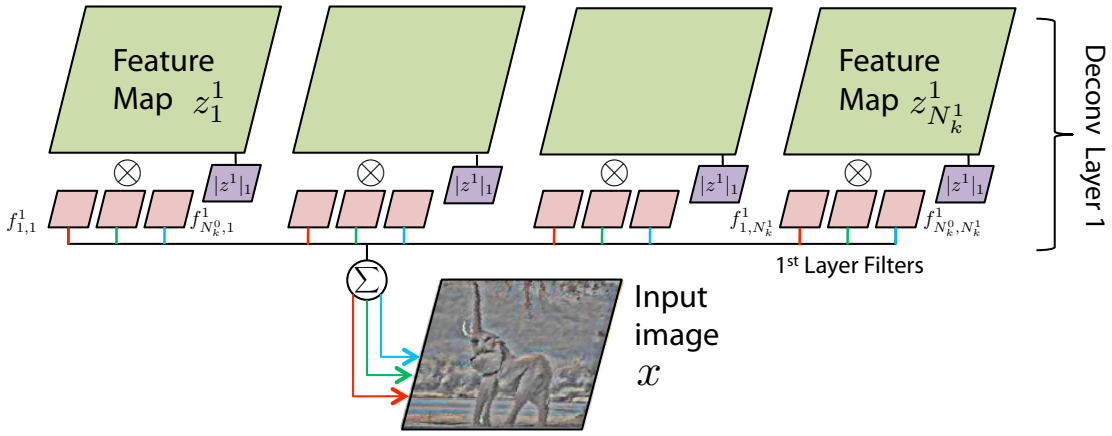

Figure 3.2: A single deconvolutional network layer (best viewed in color). In practice the first layer is fully connected, and thus the connectivity matrix g^1 is not shown.

In learning, described in Section 3.2.2, we use a set of images $x = \{x^1, \ldots, x^{N_x}\}$ for which we seek $\text{argmin}_{f,z}\, C^1$, the latent feature maps for each image and the filters [3]. Note that each image has its own set of feature maps while the filters are common to all images.

[3] We define $C^1 = \sum_{x \in \mathcal{X}} C^1(x)$, the sum of per-image cost.

3.2.1 Forming a hierarchy

The architecture described above produces sparse feature maps from a multi-channel input image. It can easily be stacked to form a hierarchy by treating the feature maps z_k^{L-1} of layer $L-1$ as input for layer L. In other words, layer L has as its input an image with ${N_k}^{L-1}$ channels being the number of feature maps at layer $L-1$. The cost function C^L for layer L is a generalization of Eq. 3.2, being:

$$\begin{aligned} C^L &= \frac{\lambda^L}{2} \sum_{c=1}^{{N_k}^{L-1}} \| \sum_{k=1}^{{N_k}^{L}} g_{c,k}^L (f_{c,k}^L \otimes z_k^L) - z_c^{L-1} \|_2^2 \\ &\quad + \sum_{k=1}^{{N_k}^{L}} |z_k^L|^\rho \end{aligned} \tag{3.5}$$

where z_c^{L-1} are the feature maps from the previous layer, and $g_{c,k}^L$ are elements of a fixed binary matrix that determines the connectivity between the feature maps at successive layers, i.e. whether z_k^L is connected to z_c^{L-1} or not [74]. In layer-1 we assume that $g_{c,k}^1$ is always 1, but in higher layers it can be sparse. We train the hierarchy from the bottom upwards, thus z_c^{L-1} is given from the results of inference with C^{L-1}. This structure is illustrated in Figure 3.2.

To further simplify this notation, we again use matrix form to write:

$$\hat{z}^{L-1} = g^L(F^L) z^L \equiv \sum_{k=1}^{{N_k}^{L}} g_{c,k}^L (f_{c,k}^L \otimes z_k^L) \;\; \forall c \in {N_k}^{L-1} \tag{3.6}$$

where $\hat{z}^{L-1}$ is the reconstruction of the feature maps of the layer below. Chaining these g, F, and z variables together can result in complicated expressions as we will see in later sections, therefore a *reconstruction* operator is introduced to further simplify the notation. This operator will be extended to include additional functions in later chapter and will be used throughout this work:

$$R^{L\to l} = g^l(F^l)\dots g^L(F^L)z^L \tag{3.7}$$

where L is the layer the reconstruction starts from and l is the layer the reconstruction ends at. If only one layer is given for the R operator, such as R^1, it is the layer from which the reconstruction begins and it is assumed to end at the input image pixel space, sometimes referred to as layer 0. This makes the overall cost for the first layer:

$$C^1 = \frac{\lambda^1}{2}\|R^1 z^1 - x\|_2^2 + |z^1|^\alpha \tag{3.8}$$

And for a higher layer L:

$$C^L = \frac{\lambda^L}{2}\|R^{L\to L-1} z^L - z^{L-1}\|_2^2 + |z^L|^\alpha \tag{3.9}$$

Unlike several other hierarchical models [56, 78, 97] we do not perform any pooling, subsampling or divisive normalization operations between layers at this stage, though this is addressed in later chapters.

3.2.2 Learning filters

To learn the filters, we alternately minimize C^L over the feature maps while keeping the filters fixed (i.e. perform inference) and then minimize C^L over the filters while keeping the feature maps fixed. This minimization is done in a layer-wise manner starting with the first layer where the inputs are the training images x. Details are given in Algorithm 1. We now describe how we learn the feature maps and filters by introducing a framework suited for large scale problems.

Inferring feature maps: Inferring the optimal feature maps z^L, given the filters and inputs is the crux of our approach. The sparsity constraint on z^L which prevents the

model from learning trivial solutions such as the identity function. When $\alpha = 1$ the minimization problem for the feature maps is convex and a wide range of techniques have been proposed [20, 77]. Although in theory the global minimum can always be found, in practice this is difficult as the problem is very poorly conditioned. This is due to the fact that elements in the feature maps are coupled to one another through the filters. One element in the map can be affected by another distant element, meaning that the minimization can take a very long time to converge to a good solution.

We tried a range of different minimization approaches to solve Eq. 3.5, including direct gradient descent, Iterative Reweighted Least Squares (IRLS) and stochastic gradient descent. We found that direct gradient descent suffers from the usual problem of flat-lining and thereby gives a poor solution. IRLS is too slow for large-scale problems with many input images. Stochastic gradient descent was found to require many thousands of iterations for convergence.

Instead, we introduce a more general framework that is suitable for any value of $\alpha > 0$, including pseudo-norms where $\alpha < 1$. The approach is a type of continuation method, as used by Geman [40] and Wang *et al.* [138]. Instead of optimizing Eq. 3.8 directly, we minimize an auxiliary cost function $\tilde{C}^L$ which incorporates auxiliary variables ω^L for each element in the feature maps z^L:

$$\begin{aligned} \tilde{C}^L &= \frac{\lambda^L}{2}\|R^{L\rightarrow L-1}z^L - z^{L-1}\|_2^2 \\ &+ \frac{\beta}{2}\|z^L - \omega^L\|_2^2 + |\omega^L|^\alpha \end{aligned} \tag{3.10}$$

where β is a continuation parameter. Introducing the auxiliary variables separates the convolution part of the cost function from the $|\cdot|^\alpha$ term. By doing so, an alternating form of minimization for z^L can be used. We first fix ω^L yielding a quadratic problem in z^L. Then, we fix z^L and solve a separable 1D problem for each element in ω^L. We call

these two stages the z and ω sub-problems respectively. As we alternate between these two steps, we slowly increase β from a small initial value until it strongly clamps z^L to ω^L. This has the effect of gradually introducing the sparsity constraint and gives good numerical stability in practice [64, 138]. We now consider each sub-problem.

z **sub-problem**: From Eq. 3.10, we see that we can solve for each z^L independently of the others. Here we take derivatives of $\tilde{C}^L$ w.r.t. z^L, assuming a fixed ω^L:

$$\frac{\partial \tilde{C}^L}{\partial z_k^L} = \lambda^L \sum_{c=1}^{N_k{}^{L-1}} g_{c,k}^L (f_{c,k}^{L^T} \otimes (\sum_{\tilde{k}=1}^{N_k{}^L} g_{c,\tilde{k}}^L (f_{c,\tilde{k}}^L \otimes z_{\tilde{k}}^L) - z_c^{L-1})) + \beta(z_k^L - \omega_k^L) \tag{3.11}$$

which can be written much more succinctly as:

$$\frac{\partial \tilde{C}^L}{\partial z^L} = \lambda^L R^{L-1 \rightarrow L^T} (R^{L \rightarrow L-1} z^L - z^{L-1}) + \beta(z^L - \omega^L) \tag{3.12}$$

where the first term in parenthesis is the error between the reconstruction and the feature maps in the layer below $e^L = (R^{L \rightarrow L-1} z^L - z^{L-1})$ and the second term in parenthesis is the error of the continuation variables. We have also introduced the *projection* operator, denoted as the transpose of the *reconstruction* operator $R^{L-1 \rightarrow L^T}$. The *projection* operator which is the backpropagation operator that undoes the reconstruction operator's sequence of operations back from l up to the layer where the reconstruction began L:

$$\nabla_z^L = R^{l \rightarrow L^T} e^L = g^L (F^L)^T \dots g^l (F^l)^T e^L \tag{3.13}$$

where ∇_z^L is the gradient of the cost function backpropagated up the model to layer L. Each transposed convolution sums over the input maps to a layer as:

$$g^L (F^L)^T e^{L-1} \equiv \sum_{c}^{N_k{}^{L-1}} g_{c,k}^L (f_{c,k}^{L^T} \otimes e_c^{L-1}) \;\; \forall k \in N_k{}^L \tag{3.14}$$

where if $(F^L)^T \equiv \texttt{flipud(fliplr(}f^L_{c,k}\texttt{))} \forall c \in {N_k}^{L-1}, \forall k \in {N_k}^L$ using Matlab notation. Although a variety of other sparse decomposition techniques [81,97] use stochastic gradient descent methods to update z^L_k for each k separately, this is not viable in a convolutional setting. Here, the various feature maps compete with each other to explain local structure in the most compact way. This requires us to simultaneously optimize over $z^L = \{z^L_1, \ldots, z^L_{{N_k}^L}\}$. Setting $\frac{\partial \tilde{C}}{\partial z^L} = 0$, the optimal z^L is the solution to the following ${N_k}^L(N_{z_{rows}})(N_{z_{cols}})$ linear system:

$$A \begin{pmatrix} z^L_1 \\ . \\ z^L_{{N_k}^L} \end{pmatrix} = \begin{pmatrix} \sum_{c=1}^{{N_k}^{L-1}} F^{L^T}_{c,1} z^{L-1}_c + \frac{\beta}{\lambda}\omega^L_1 \\ . \\ \sum_{c=1}^{{N_k}^{L-1}} F^{L^T}_{c,{N_k}^L} z^{L-1}_c + \frac{\beta}{\lambda}\omega^L_{{N_k}^L} \end{pmatrix} \tag{3.15}$$

where

$$A = \begin{pmatrix} \sum_{c=1}^{{N_k}^{L-1}} F^{L^T}_{c,1} F^L_{c,1} + \frac{\beta}{\lambda} I & . & \sum_{c=1}^{{N_k}^{L-1}} F^L_{c,1} F^{L^T}_{c,{N_k}^L} \\ . & . & . \\ \sum_{c=1}^{{N_k}^{L-1}} F^{L^T}_{c,{N_k}^L} F^L_{c,1} & . & \sum_{c=1}^{{N_k}^{L-1}} F^{L^T}_{c,{N_k}^L} F^L_{c,{N_k}^L} + \frac{\beta}{\lambda} I \end{pmatrix} \tag{3.16}$$

The matrix A represents the outer product of each filter F convolved with itself, being a N_k by N_k set of larger filter planes. Eq. 3.15 can be effectively minimized by conjugate gradient (CG) descent. Note that A never needs to be formed since the Az product can be directly computed using convolution operations inside the CG iteration. Each Az product requires $2{N_k}^{L-1}{N_k}^L$ convolutions of filters with the $(N_{z_{rows}})(N_{z_{cols}})$ feature maps and can easily be parallelized.

Although some speed-up might be gained by using FFTs in place of spatial convolutions [14], particularly if the filter size $N_{F_{rows}}$ by $N_{F_{cols}}$ is large, this can introduce boundary effects in the feature maps – therefore solving in the spatial domain was used throughout this work.

ω **sub-problem**: Given fixed z^L, finding the optimal ω^L requires solving a 1D optimization problem for each element in the feature map. If $\alpha = 1$ then, following Wang

et al. [138], ω^L has a closed-form solution given by:

$$\omega^L = \max(|z^L| - \frac{1}{\beta}, 0)\frac{z^L}{|z^L|} \tag{3.17}$$

where all operations are element-wise. Alternatively for arbitrary values of $\alpha > 0$, the optimal solution can be computed via a lookup-table [64]. This permits us to impose more aggressive forms of sparsity than $\alpha = 1$.

Filter updates : With ω^L fixed and z^L computed for a fixed i, we use the following for gradient updates for each filter plan $f^L_{c,k}$:

$$\frac{\partial \tilde{C}^L}{\partial f^L_{c,k}} = \lambda^L {z^L_k}^T \otimes (\sum_{\tilde{k}=1}^{{N_k}^L} g^L_{c,\tilde{k}}(f^L_{c,\tilde{k}} \otimes z^L_{\tilde{k}}) - z^{L-1}_c) = \lambda^L {z^L}^T (R^{L\rightarrow L-1} z^L - z^{L-1}) \tag{3.18}$$

where ${z^L}^T$ performs a convolution between the reconstruction error in parenthesis and the rotated versions of the feature maps. The overall learning procedure is summarized in Algorithm 1. In practice, instead of updating a single image, we operate on mini-batches of images at once to run inference in parallel and filter learning using the average gradient over the mini-batch.

3.2.3 Image representation/reconstruction

To use the model for image reconstruction, we first decompose an input image by using the learned filters F to find the latent representation z. We explain the procedure for a 2 layer model. We first infer the feature maps z^1 for layer 1 using the input x and the filters F^1 by minimizing C^1. Next we update the feature maps for layer 2, z^2 in an alternating fashion. In step 1, we first minimize the reconstruction error between the original image x and the reconstruction $\hat{x}^2$ from z^2, projecting z^2 through F^2 and F^1 to the image:

Algorithm 1 : Learning a single layer, L, of the deconvolutional network.

Require: Input to layer L: Training images $x \in \mathcal{X}$ if $L = 1$, otherwise z^{L-1}
Require: # feature maps ${N_k}^L$, connectivity g^L
Require: Regularization weight λ^L, # epochs N_e
Require: Continuation parameters: $\beta_0, \beta_{\text{Inc}}, \beta_{\text{Max}}$
Initialize feature maps and filters $z \sim \mathcal{N}(0, \epsilon)$, $F \sim \mathcal{N}(0, \epsilon)$
for epoch $= 1 : N_e$ **do**
  for $im = 1 : N_x$ **do** %% note: z^L and ω^L are per-image
    $\beta = \beta_0$
    while $\beta < \beta_{\text{Max}}$ **do**
      Given z^L, solve for ω^L using Eq. 3.17
      Given ω^L, solve for z^L using Eq. 3.15
      $\beta = \beta \cdot \beta_{\text{Inc}}$
    end while
  end for
  Update F^L using gradient descent on Eq. 3.18
end for
Output: Filters F

$$\hat{x}^2 = R^2 z^2 = \sum_{k=1}^{{N_k}^1} g^1_{c,k} (f^1_{c,k} \otimes \sum_{\tilde{k}=1}^{{N_k}^2} g^2_{k,\tilde{k}} (f^2_{k,\tilde{k}} \otimes z^2_{\tilde{k}})) \quad \forall c \in {N_k}^0 \tag{3.19}$$

thus minimizing the cost:

$$\frac{\lambda^2}{2} \|R^2 z^2 - x\|_2^2 + |z^2| \tag{3.20}$$

In step 2, we minimize the reconstruction error w.r.t. z^1:

$$\frac{\lambda^2}{2} \|R^{2\to 1} z^2 - z^1\|_2^2 + |z^2| \tag{3.21}$$

We alternate between steps 1 and 2, using conjugate gradient descent in both. Once z^2 has converged, we reconstruct x by projecting back to the image via F^2 and F^1:

$$\hat{x}^2 = R^2 z^2 \tag{3.22}$$

An important detail is the addition of an extra feature map z^0 per input map of layer 1 that connects to the image via a constant uniform filter F^0. Unlike the sparsity priors on the other feature maps, z^0 has an ℓ_2 prior on the gradients of z^0, i.e. the prior is of

the form $\|\nabla z^0\|^2$. These maps capture the low-frequency components, leaving the high-frequency edge structure to be modeled by the learned filters. Given that the filters were learned on high-pass filtered images, the z^0 maps assist in reconstructing raw unprocessed images.

3.3 Experiments

In our experiments, we train on two datasets of 100×100 images, one containing natural scenes of fruits and vegetables and the other consisting of scenes of urban environments. In all our experiments, unless otherwise stated, the same learning settings were used for all layers, namely: $H=7$, $\lambda=1$, $\alpha=1$, $\beta_0=1$, $\beta_{\text{Inc}}=6$, $\beta_{\text{Max}}=10^5$, $N_e=3$.

3.3.1 Learning multi-layer deconvolutional filters

With the settings described above we trained a separate 3 layer model for each dataset, using an identical architecture. The first layer had 9 feature maps fully-connected to the input. The second layer had 45 maps: 36 were connected to pairs of maps in the first layer, and the remainder were singly-connected. The third layer had 150 feature maps, each of which was connected to a random pair of second layer feature maps. In Figure 3.7 and Figure 3.8 we show the filters that spontaneously emerge, projected back into pixel space.

The first layer in each model learns Gabor-style filters, although for the city images they are not evenly distributed in orientation, preferring vertical and horizontal structures. The second layer filters comprise an assorted set of V2-like elements, with center-surround, corners, T-junctions, angle-junctions and curves.

The third layer filters are highly diverse. Those from the model trained on food images (Figure 3.7) comprise several types: oriented gratings (rows 1–4); blobs (D8, E7, H9); box-like structures (B10, F12) and others that capture parallel and converging lines

(C12, J11). The filters trained on city images (Figure 3.8) capture line groupings in horizontal and vertical configurations. These include: conjunctions of T-junctions (C15, G11); boxes (D14, E4) and various parallel lines (B15, D8, I3). Some of the filters are representative of the tokens shown in Fig. 2-4 of Marr [84] (see Figure 3.1).

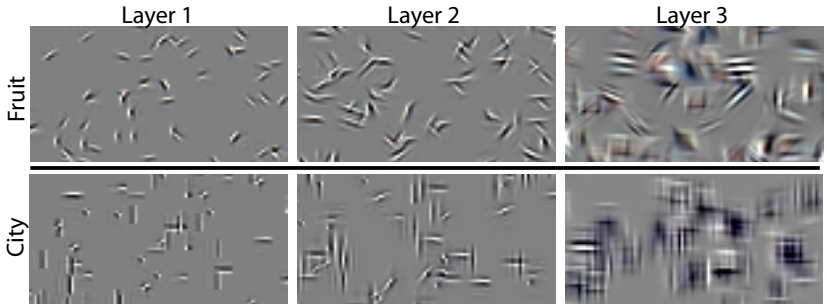

Figure 3.3: Samples from the layers of two deconvolutional network models, trained on fruit (top) or city (bottom) images.

Since our model is generative, we can sample from it. In Figure 3.3 we show samples from the two different models from each level projected down into pixel space. The samples were drawn using the relative firing frequencies of each feature from the training set.

3.3.2 Comparison to patch-based decomposition

To demonstrate the benefits of imposing sparsity within a convolutional architecture, we compare our model to the patch-based sparse decomposition approach of Mairal *et al.* [81]. Using the SPAMS code accompanying [81] we performed a patch-based decomposition of the two image sets, using 100 dictionary elements. The resulting filters are shown in Figure 3.4(left). We then attempted to build a hierarchical 2 layer model by taking the sparse output vectors from each image patch and arranging them into a map over the image. Applying the SPAMS code to this map produces the 2^{nd} layer filters shown in Figure 3.4(right). While larger in scale than the 1^{st} layer filters, they are generally Gabor-like and do not show the diverse edge conjunctions present in our 2^{nd} layer filters. To probe this result, we visualize the latent feature maps of our convolutional decomposition and Mairal *et al.* .'s patch-based decomposition in Figure 3.5.

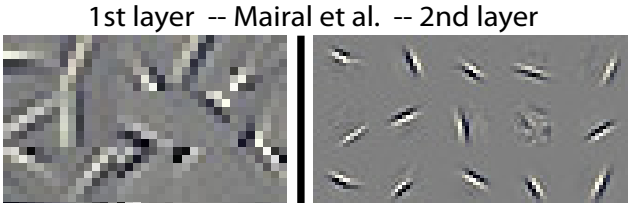

Figure 3.4: Examples of 1st and 2nd layer filters learned using the patch-based sparse deconvolution approach of Mairal *et al.* [81], applied to the food dataset. While the first layer filters look similar to ours, the 2nd layer filters are merely larger versions of the 1st layer filters, lacking the edge compositions found in our 2nd layer (see Figure 3.7 and Figure 3.8).

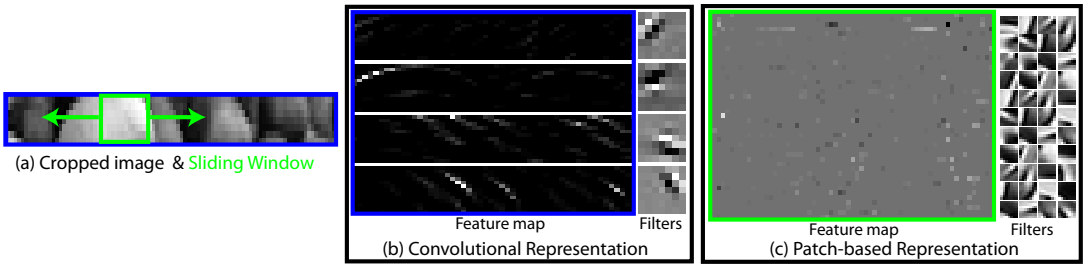

Figure 3.5: A comparison of convolutional and patch-based sparse representations for a crop from a natural image (a). (b): Sparse convolutional decomposition of (a). Note the smoothly varying feature maps that preserve spatial locality. (c): Patch-based convolutional decomposition of (a) using a sliding window (green). Each column in the feature map corresponds to the sparse vector over the filters for a given x-location of the sliding window. As the sliding window moves the latent representation is highly unstable, changing rapidly across edges. Without a stable representation, stacking the layers will not yield higher-order filters, as demonstrated in Figure 3.4.

3.3.3 Caltech-101 object recognition

We now demonstrate how deconvolutional networks can be used in an object recognition setting. As we are primarily interested in image representation, we compare to other methods using a common framework of one or more layers of feature extraction, followed by Spatial Pyramid Matching [70]. We use the standard Caltech-101 dataset for evaluating classification performance, but we would like to emphasize that the filters of our DN have been learned using a generic, disparate training set: a concatenation of the natural and city images. The Caltech-101 images are only used for supervised training[4] of the

[4]The 150x150 pixel contrast normalized gray images used for classification were connected to 8 feature maps in layer 2. Second layer maps were connected singly and in every possible pair to the layer 1 maps,

Table 3.1: Recognition performance on Caltech-101.

# training examples	15	30
DN-1 (KM)	$57.7 \pm 1.0\%$	$65.8 \pm 1.3\%$
DN-2 (KM)	$57.0 \pm 0.8\%$	$65.5 \pm 1.0\%$
DN-(1+2) (KM)	$\mathbf{58.6} \pm 0.7\%$	$\mathbf{66.9} \pm 1.1\%$
Lazebnik *et al.* [70]	56.4%	$64.6 \pm 0.7\%$
Jarret *et al.* [56]	–	$65.6 \pm 1.0\%$
Lee *et al.* [78] layer-1	$53.2 \pm 1.2\%$	$60.5 \pm 1.1\%$
Lee *et al.* [78] layer-1+2	$57.7 \pm 1.5\%$	$65.4 \pm 0.5\%$
Zhang *et al.* [143]	$59.1 \pm 0.6\%$	$66.2 \pm 0.5\%$

classifier.

Our baseline is the method of Lazebnik *et al.* [70] where SIFT descriptors are computed densely over the image, followed by Spatial Pyramid Matching. To compare our latent representation with this approach, we densely constructed descriptors[5] from layer 1 (DN-1) and layer 2 (DN-2) feature activations. These were then vector quantized using K-means (KM) into 1000 clusters and grouped into a spatial pyramid from which an SVM histogram intersection kernel was computed for classification. Results for 10-fold cross validation with 15 and 30 images training per category are reported in Table 3.1.

Our method slightly outperforms the SIFT-based approach [70] as well as other multi-stage convolutional feature-learning methods such as convolutional DBNs [78] and feed-forward convolutional networks [56]. We achieved the best performance when we concatenated the spatial pyramids of both layers before computing the SVM histogram intersection kernels: denoted DN-(1+2).

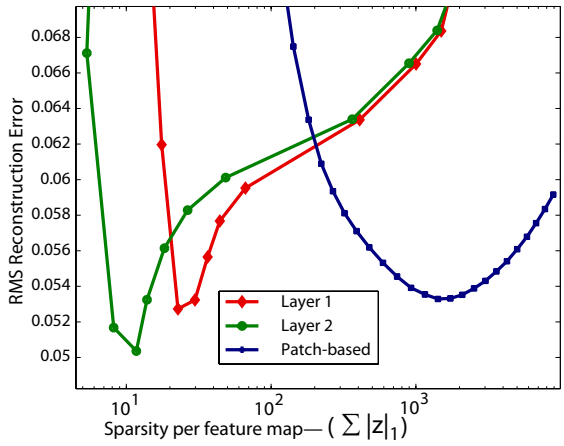

Figure 3.6: Exploring the trade-off between sparsity and denoising performance for our 1st and 2nd layer representations (red and green respectively), as well as the patch-based approach of Mairal *et al.* [81] (blue). Our 2nd layer representation simultaneously achieves a lower reconstruction error and sparser feature maps.

3.3.4 Denoising images

Given that our learned representation can be used for synthesis as well as analysis, we explore the ability of a two layer model to denoise images. Applying Gaussian noise to an image with a SNR of 13.84dB, the first layer of our model was able to reduce the noise to 16.31dB. Further, using the latent features of our second layer to reconstruct the image, the noise was reduced to a SNR of 18.01dB.

We also explore the relative sparsity of the feature maps in the 1st and 2nd layers of our model as we vary λ. In Figure 3.6 we plot the average sparsity of each feature map against RMS reconstruction error, we see that the feature maps at layer 2 are sparser and give a lower reconstruction error, than those of layer 1. We also plot the same curve for the patch-based sparse decomposition of Mairal *et al.* [81]. In this framework, inference is performed separately for each image patch and since patches overlap, a much larger number of latent features are needed to represent the image. The curve was produced

for a total of 36 layer 2 feature maps. α=0.8, λ^1=10, and λ^2=1 were used to maintain more discriminative information in the feature maps.

[5] Activations from each layer were split into overlapping 16x16 patches at a stride of 2 pixels. The absolute value of activations in each patch were pooled by a factor of 4 then grouped in 4x4 regions on each of 8 layer 1 feature maps giving a 128-D descriptor per patch and grouped in 2x2 regions on each of 36 layer 2 maps leading to 144-D layer 2 descriptors.

by varying the number of active dictionary atoms per patch in reconstruction.

3.3.5 Inference timings

Our efficient optimization scheme makes it feasible to perform exact inference in a convolutional setting. Alternate approaches [78] rely on simple non-linear encoders to perform approximate inference. Our scheme is linear in the number of filters and pixels in the image (5.8 ± 1.0secs/filter/megapixel) Thus for 150×150 images used in the Caltech 101 experiments, using the architecture described in Section 3.3.1, inferences takes 2.5s, 10s, 55s layers-1,2,3 respectively. Due to the small filter sizes, learning incurs only a 10% overhead relative to inference. While our algorithm is slow compared to approaches that use bottom-up encoders, heavy use of the convolution operator makes it amenable to parallelization and GPU-based implementations which we show to give between 1 and 2 orders of magnitude speed-up in later chapters. Additional performance gains are also shown when pooling is introduced between layers.

3.4 Discussion

In this chapter we introduced deconvolutional networks: a conceptually simple framework for learning sparse, over-complete feature hierarchies. Applying this framework to natural images produces a highly diverse set of filters that capture high-order image structure beyond edge primitives. These arise without the need for hyper-parameter tuning or additional modules, such as local contrast normalization, max-pooling and rectification [56]. Our approach relies on robust optimization techniques to minimize the poorly conditioned cost functions that arise in the convolutional setting. Supplemental images, video, and code for this chapter can be found at: `http://www.cs.nyu.edu/~zeiler/pubs/cvpr2010/`.

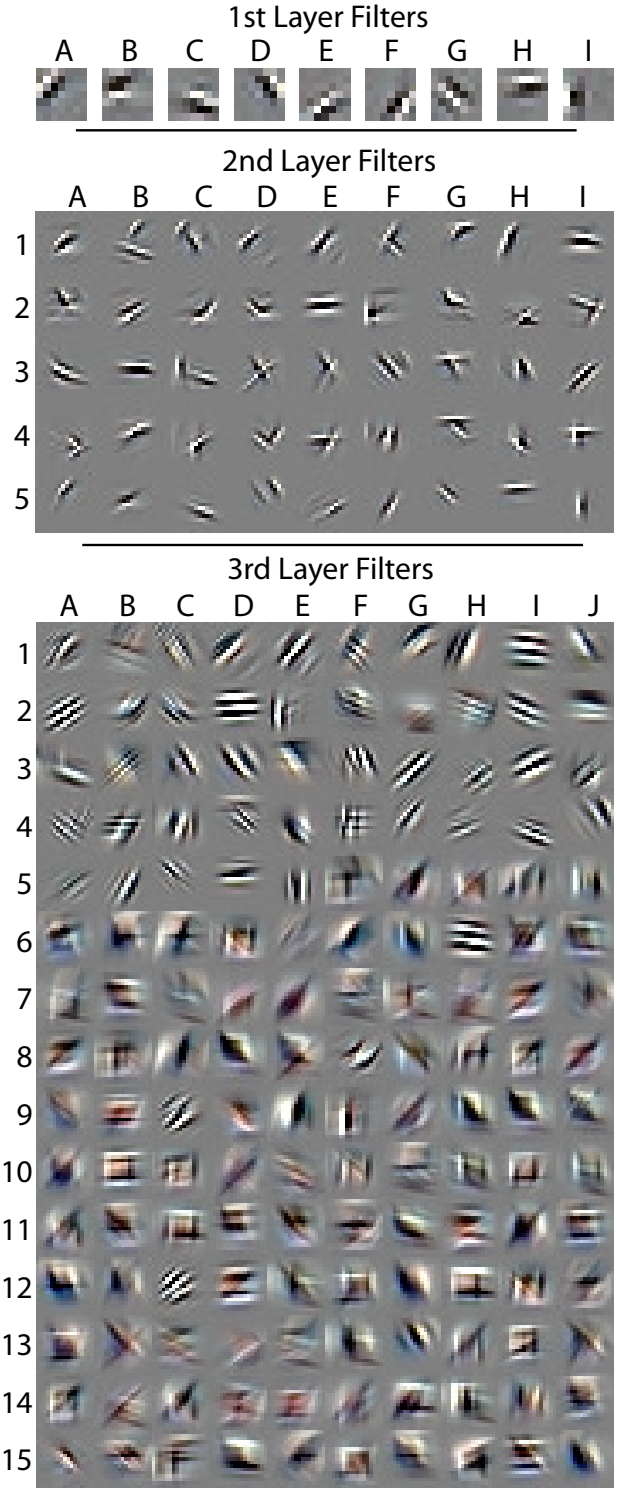

Figure 3.7: Filters from each layer in our model, trained on food scenes. Note the rich diversity of filters and their increasing complexity with each layer. In contrast to the filters shown in Figure 3.8, the filters are evenly distributed over orientation.

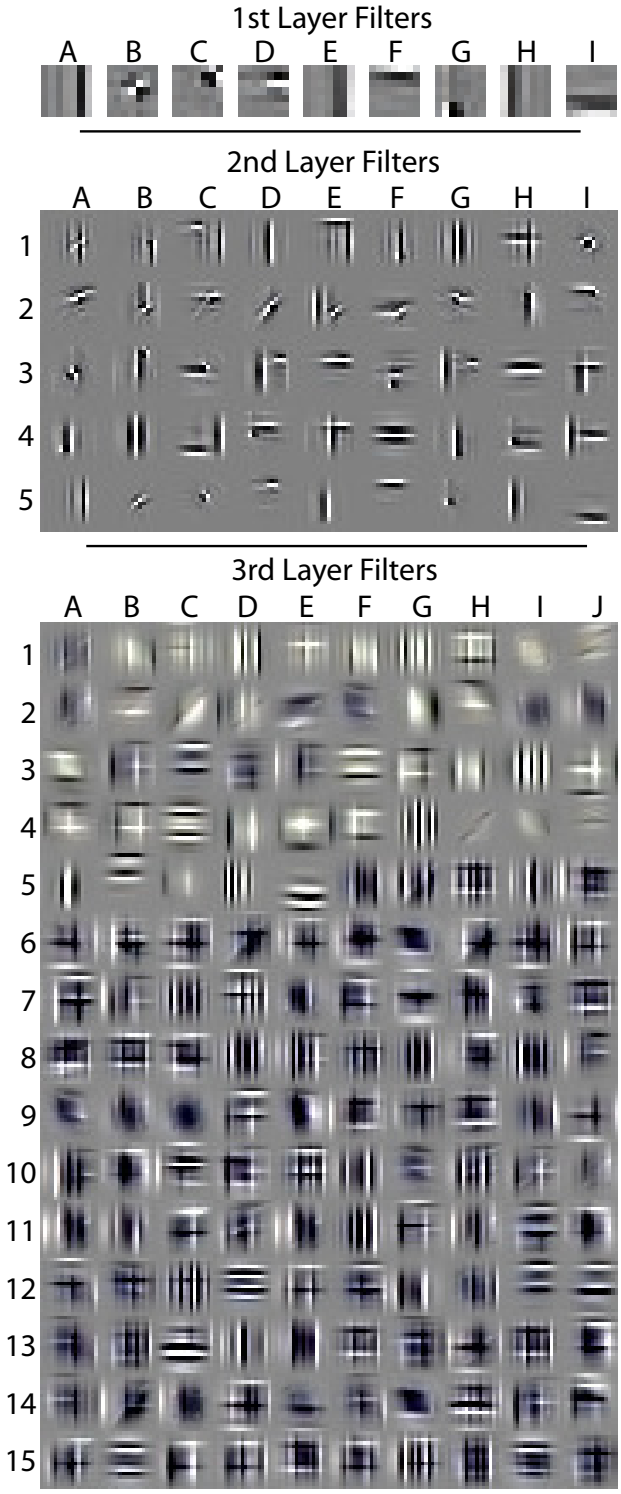

Figure 3.8: Filters from each layer in our model, trained on the city dataset. Note the predominance of horizontal and vertical structures.

Chapter 4

Adaptive Deconvolutional Networks for Mid and High Level Feature Learning

4.1 Introduction

For many tasks in vision, the critical problem is discovering good image representations that are invariant to many changes in input. For example, the advent of local image descriptors such as SIFT and HOG has precipitated dramatic progress in matching and object recognition. Interestingly, many of the successful representations are quite similar [139], essentially involving the calculation of edge gradients, followed by some histogram or pooling operation. While this is effective at capturing low-level image structure, the challenge is to find representations appropriate for mid and high-level structures, i.e. corners, junctions, and object parts, which are surely important for understanding images.

The previous chapter introduced deconvolutional networks which can learn interesting

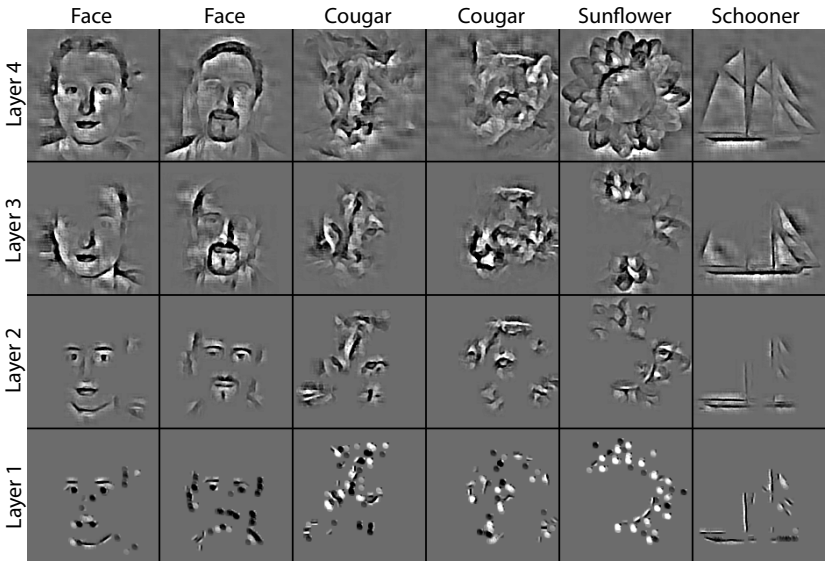

Figure 4.1: Top-down parts-based image decomposition with an adaptive deconvolutional network. Each column corresponds to a different input image under the same model. Row 1 shows a single activation of a 4^{th} layer feature map projected into image space. Conditional on the activations in the layer above, we also take a subset of 5, 25 and 125 active features in layers 3, 2 and 1 respectively and visualize them in image space (rows 2-4). The activations reveal mid and high level primitives learned by our model. In practice there are many more activations such that the complete set sharply reconstructs the *entire* image from each layer.

compositions of edge primitives into corner, curve and parallel line features. However, due to some crucial missing components in the model, the features learned in higher layers do not represent object parts or entire objects. In this chapter we extend deconvolutional networks such that when trained on natural images, the layers of the model capture image information in a variety of forms: low-level edges, mid-level edge junctions, high-level object parts and complete objects. We propose novel solutions to two fundamental problems associated with all types of feature hierarchies, not only deconvolutional networks, in order to make this possible.

The first problem relates to invariance: while edges only vary in orientation and scale, larger-scale structures are more variable. Trying to explicitly record all possible shapes of t-junction or corners, for example, would lead to a model that is exponential in the number of primitives. Hence invariance is crucial for modeling mid and high-level

structure.

The second problem relates to the layer-by-layer training scheme employed in hierarchical models, such as deep belief networks [48, 78] and convolutional sparse coding [19, 62]. Lacking a method to efficiently train all layers with respect to the input, these models are trained greedily from the bottom up, using the output of the previous layer as input for the next. The major drawback to this paradigm is that the image pixels are discarded after the first layer, thus higher layers of the model have an increasingly diluted connection to the input. This makes learning fragile and impractical for models beyond a few layers.

Our solution to both these issues is to introduce a set of latent switch variables, computed for each image, that locally adapt the model's filters to the observed data. Hence, a relatively simple model can capture wide variability in image structure. The switches also provide a direct path to the input, even from high layers in the model, allowing each layer to be trained with respect to the image, rather than the output of the previous layer. As we demonstrate, this makes learning far more robust. Additionally, the switches enable the use of an efficient training method, allowing us to learn models with many layers and hundreds of feature maps on thousands of images.

Some related approaches have been proposed in the past to learn hierarchies of features, but these previous attempts have not directly tackled both of these two problems in the unsupervised learning setting. Convolutional neural networks (CNN) [74], like deconvolutional networks, produce a hierarchy of latent feature maps via learned filters. However, they process images bottom-up and are trained discriminatively and purely supervised, while our approach is top-down (generative) and unsupervised. Predictive Sparse Decomposition (PSD) [56] adds a sparse coding component to CNNs that allows unsupervised training. In contrast to our model, each layer only reconstructs the layer below. Additional shift invariance can be incorporated as in [99] by recording transformation parameters for use in reconstruction. This is similar to the work presented here,

storing location information from max pooling operations, but was previously used in a single layer autoencoders whereas we extend this to train multiple layers of deconvolutions.

This limitation is shared by Deep Belief Networks (DBNs) [48,78] which are comprised of layers of Restricted Boltzmann Machines. Each RBM layer, conditional on its input, has a factored representation that does not directly perform explaining away. Also, training is relatively slow.

The closest approaches to ours are those based on convolutional sparse coding [19, 62]. Like PSD and DBNs, each layer only attempts to reconstruct the output of the layer below. Additionally, they manually impose sparse connectivity between the feature maps of different layers, thus limiting the complexity of the learned representation. This is similar to the first chapter on deconvolutional networks, where connectivity could be sparse in higher layers. In contrast, in this chapter and the remainder of this work we found full connectivity to work well for learning complex structures and eliminates the choice of connectivity structure to use. Additional differences include: the lack of pooling layers and inefficient inference schemes [19] that do not scale.

Our model performs a decomposition of the full image, in the spirit of Zhu and Mumford [145] and Tu and Zhu [131]. This differs from other hierarchical models, such as Fidler and Leonardis [36] and Zhu *et al.* [144], that only model a stable sub-set of image structures at each level rather than all pixels. Another key aspect of our approach is that we learn the decomposition from natural images. Several other hierarchical models such as Serre *et al.*'s HMax [101, 114] and Guo *et al.* [43] use hand-crafted features at each layer.

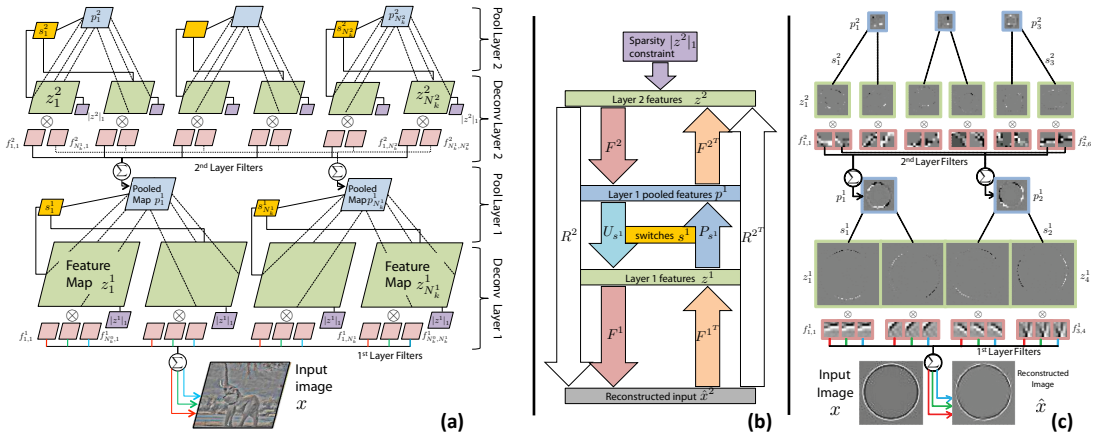

Figure 4.2: **(a):** A visualization of two layers of our model. Each layer consists of a deconvolution and a max-pooling. The deconvolution layer is a convolutional form of sparse coding that decomposes input image x into feature maps z^1 (green) and learned filters F^1 (red), which convolve together and sum to reconstruct x. The filters have N_c planes, each used to reconstruct a different channel of the input image. Each z map is penalized by a per-element ℓ_1 sparsity term (purple). The max-pooling layer pools within and between feature maps, reducing them in size and number to give pooled maps p (blue). The locations of the maxima in each pooling region are recorded in switches s (yellow). The second deconvolution/pooling layer is conceptually identical to the first, but now has two input channels rather than three. In practice, we have many more feature maps per layer and have up to 4 layers in total. **(b):** A block diagram view of the inference operations within the model for layer 2. See Section 4.2.1 for an explanation. **(c):** A toy instantiation of the model on the left, trained using a single input image of a (contrast-normalized) circle. The switches and sparsity terms are not shown. Note the sparse feature maps (green) and the effect of the pooling operations (blue). Since the input is grayscale, the planes of the 1st layer filters are identical.

4.2 Deconvolutional Networks with Adaptive Pooling

Our model produces an over-complete image representation that can be used as input to standard object classifiers. Unlike many image representations, ours is learned from natural images and, given a new image, requires inference to compute. The model decomposes an image in a hierarchical fashion using multiple alternating layers of convolutional sparse coding (deconvolution) and max-pooling. Each of the deconvolution layers attempts to directly minimize the reconstruction error of the input image under a sparsity constraint on an over-complete set of feature maps. The cost function $C^L(x)$

for layer L comprises two terms [1]: (i) a likelihood term that keeps the reconstruction of the input $\hat{x}^L$ close to the original input image x; (ii) a regularization term that penalizes the ℓ_1 norm of the 2D feature maps z^L on which the reconstruction $\hat{x}^L$ depends. The relative weighting of the two terms is controlled by λ^L:

$$C^L = \frac{\lambda^L}{2}\|\hat{x}^L - x\|_2^2 + |z^L|_1 \tag{4.1}$$

Unlike existing approaches [19, 62], our convolutional sparse coding layers attempt to directly minimize the reconstruction error of the input image, rather than the output of the layer below.

Deconvolution: Consider the first layer of the model, as shown in Figure 4.2(a). The reconstruction $\hat{x}^1$ from the first layer, which remains the same as in the previous chapter, is formed by convolving each of the 2D feature maps z^1 with filters F^1 and sums them:

$$\hat{x}^1 = F^1 z^1 = R^1 z^1 \tag{4.2}$$

We describe the inference scheme used to discover an optimal z^1 and the closely related learning approach for estimating F^1 in Sections 4.2.1 and 4.2.2 respectively, which differs slightly from the previous chapter.

Pooling: On top of each deconvolutional layer, we perform a 3D max-pooling operation on the feature maps z. This allows the feature maps of the layer above to capture structure at a larger scale than the current layer. The pooling is 3D in that it occurs both spatially (within each 2D z map) and also between adjacent maps, as shown in Figure 4.3. Within each neighborhood of z we record both the value and location of the maximum (irrespective of sign). *Pooled maps* p store the values, while *switches* s record the locations.

[1] We utilize L instead of l from this point on to distinguish the top layer where reconstruction begins, L, from an intermediate layer l.

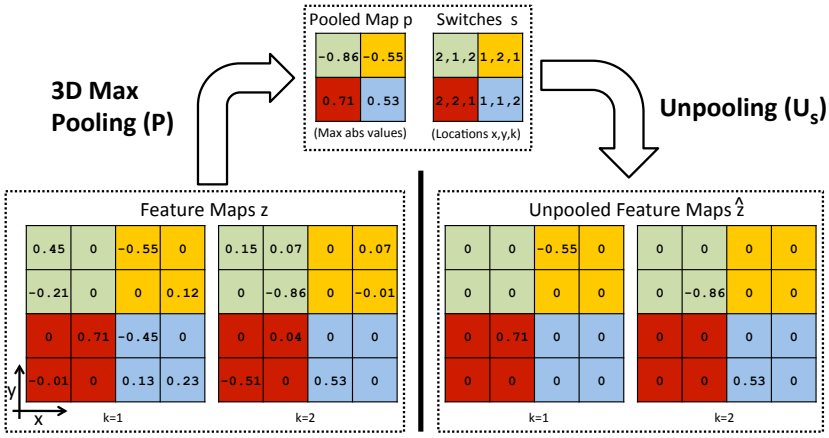

Figure 4.3: An example of 3D max pooling using a neighborhood of size $2 \times 2 \times 2$, as indicated by the colors. The pooling operation P is applied to the feature maps z, yielding pooled maps p and switches s that record the location of the maximum (irrespective of sign). Given the pooled maps and switches, we can also perform an unpooling operation U_s which inserts the pooled values in the appropriate locations in the feature maps, with the remaining elements being set to zero.

Our model uses two distinct forms of pooling operation on the feature maps z. The first, shown in Figure 4.3, treats the switches s as an output: $[p, s] = Pz$. The second takes the switches s as an input, where they specify which elements in z are copied into p. If s is fixed, then this is a linear operation which can be written as $p = P_s z$, with P_s being a binary selection matrix, set by switches s. This latter operation is used when computer gradients up the model.

The corresponding unpooling operation U_s, shown in Figure 4.3, takes the elements in p and places them in z at the locations specified by s, the remaining elements being set to zero: $\hat{z} = U_s p$. Note that this is also a linear operation for fixed s and that $U_s = P_s^T$.

Multiple Layers: The architecture remains the same for higher layers in the model but there are typically a larger number of feature maps ${N_k}^l$ in higher layers. Starting reconstruction from layer L we reconstruct the input pixels through the filters and switches of the layers below. We extend the *reconstruction* operator R^L to take the feature maps z^L from layer L and alternately convolve (F) and unpool them (U_s) down to the input:

$$\hat{x}^L = F^1 U_{s^1} F^2 U_{s^2} \dots F^L z^L = R^L z^L \tag{4.3}$$

Note that $\hat{x}^L$ depends on the feature maps z^L from the current layer but not those beneath[2]. However, the reconstruction operator R^L does depend on the pooling switches in the intermediate layers $(s^{L-1}, \dots, s^1)$ since they determine the unpooling operations $U_{s^{L-1}}, \dots, U_{s^1}$. These switches are configured by the values of $z^{L-1} \dots z^1$ from previous iterations.

We also redefine the *projection* operator R^{L^T} which takes error signal e^L at the input and projects it back up to the feature maps of any layer $l \leq L$. Given previously determined switches in the lower layers, $s^1, \dots, s^{L-1}$, the new operator maps upwards through convolutions and pooling operations:

$$R^{L^T} = F^{L^T} P_{s^{L-1}} F^{L-1^T} P_{s^{l-2}} \dots P_{s^1} F^{1^T} \tag{4.4}$$

A crucial property of our model is that *given the switches* s, both the reconstruction R^L and projection operators R^{L^T} are linear, thus allowing the gradients to be easily computed, even in models with many layers, making inference and learning straightforward. Figure 4.2(a) illustrates two layers of deconvolution and pooling within our model. Figure 4.2(b) shows how the reconstruction and projection operators are made up of the filtering, pooling and unpooling operations.

4.2.1 Inference

For a given layer L, inference involves finding the feature maps z^L that minimize C^L, given an input image x and filters F. For each layer we need to solve a large ℓ_1 convolutional sparse coding problem and we adapt the ISTA scheme of Beck and Teboulle [4].

[2]In other words, when we project down to the image, we do not impose sparsity on any of the intermediate layer reconstructions $\hat{z}^{L-1}, \dots, \hat{z}^1$

This uses an iterative framework of gradient and shrinkage steps.

Gradient step: This involves taking a step in the direction of the gradient g^L of the reconstruction term of Eq. 4.1, with respect to z^L:

$$\nabla_z^L = R^{L^T}(R^L z^L - x) = R^{L^T} e^L \tag{4.5}$$

To compute the gradient, we take feature maps z^L and, using the filters and switch settings of the layers below, reconstruct the input $\hat{x}^L = R^L z^L$. We then compute the reconstruction error $\hat{x}^L - x$. This is then propagated back up the network using R^{L^T} which alternately filters (F^T) and pools it (P_s) up to layer L, yielding the gradient ∇_z^L. This process is visualized in Figure 4.2(middle) for a two layer model.

Once we have the gradient ∇_z^L, we then can update z^L:

$$z^L = z^L - \lambda^L \eta^L \nabla_z^L \tag{4.6}$$

where the η^L parameter sets the size of the gradient step.

Shrinkage step: Following the gradient step, we perform a per-element shrinkage operation that clamps small elements in z^L to zero, thus increasing its sparsity:

$$z^L = \max(|z^L| - \eta^L, 0)\text{sign}(z^L) \tag{4.7}$$

Pooling/unpooling: We then update the switches s^L of the current layer by performing a pooling operation[3] $[p^L, s^L] = P(z^L)$, immediately followed by an unpooling operation $z^L = U_{s^L} p^L$. This fulfills two purposes: (i) it ensures that we can accurately reconstruct the input through the pooling operation, when building additional layers on top and (ii)

[3]This is the form of pooling shown in Figure 4.3 that treats the switches as an output. It is not the same as the form of pooling used in projection operator R^T, where they are an input

Algorithm 2 Adaptive Deconvolutional Networks

Require: Training set $\mathcal{X}$, # layers N_l, # epochs N_e, # ISTA steps N_t
Require: Regularization weights λ^l, # feature maps ${N_k}^l$ $\forall l \in N_l$
Require: Shrinkage parameters η^l $\forall l \in N_l$
for $L = 1 : N_l$ **do** %% Loop over layers to reconstruct from
  Init. features/filters: $z^L \sim \mathcal{N}(0, \epsilon)$, $F^L \sim \mathcal{N}(0, \epsilon)$
  for epoch $= 1 : N_e$ **do** %% Epoch iteration
    for $im = 1 : N_x$ **do** %% Loop over images
      for $t = 1 : N_t$ **do** %% ISTA iteration
        Reconstruct input: $\hat{x}^L = R^L z_k$
        Compute reconstruction error: $e = \hat{x}^L - x$
        Propagate error up to layer L: $\nabla_z^L = {R^L}^T e^L$
        Take gradient step: $z^L = z^L - \lambda^L \eta^L \nabla_z^L$
        Perform shrinkage: $z^L = \max(|z^L| - \eta^L, 0)\text{sign}(z^L)$
        Pool z^L, updating the switches s^L: $[p^L, s^L] = P(z^L)$
        Unpool p^L, using s^L to give z^L: $z^L = U_{s^L} p^L$
      end for
    end for
    Update F^L by solving Eq. 4.8 using CG
  end for
end for
Output: filters F, feature maps z and switches s.

it updates the switches to reflect the revised values of the feature maps. Once inference has converged, the switches will be fixed, ready for training the layer above. Hence, a secondary goal of inference is to determine the optimal switch settings in the current layer.

Overall iteration: A single ISTA iteration consists of each of the three steps above: gradient, shrinkage and pooling/unpooling. During inference we perform 10 ISTA iterations per image for each layer.

Both the reconstruction R and propagation R^T operations are very quick, just consisting of convolutions, summations, pooling and unpooling operations, all of which are amenable to parallelization. This makes it possible to efficiently solve the system in Eq. 4.1, even with massively over-complete layers where z^L may be up to 10^5 in length.

Note that while the gradient step is linear, the model as a whole is not. The non-linearity arises from two sources: (i) sparsity, as induced by the shrinkage Eq. 4.7, and (ii) the settings of the switches s which alter the pooling/unpooling within R^L.

4.2.2 Learning

In learning the goal is to estimate the filters F in the model, which are shared across all images $\mathcal{X} = \{x^1, \ldots, x, \ldots, x^{N_x}\}$. For a given layer L, we perform inference to compute z^L. Taking derivatives of Eq. 4.1 with respect to F^L and setting to zero, we obtain the following linear system in F^L [4]:

$$\left(z^{L^T} P_{s^{L-1}} R^{L-1^T}\right) \hat{x}^L = \left(z^{L^T} P_{s^{L-1}} R^{L-1^T}\right) x \tag{4.8}$$

where $\hat{x}$ is the reconstruction of the input using the current value of F^L. We have omitted the sum over images within an mini-batch which appear on both sides of the equation to simplify notation. Note, the reconstruction operator R^L is image specific because it uses the location of pooling switches s^L in each layer L below L to operate. We solve this system using linear conjugate gradients (CG). The matrix-vector product of the left-hand side is computed efficiently by mapping down to the input and back up again using the R^L and R^{L^T} operations. After solving Eq. 4.8, we normalize F^L to have unit length. The overall algorithm for learning all layers of the model is given in Algorithm 1. We alternate small steps in z^L and F^L, using a single ISTA step per epoch to infer z^L and two CG iterations for F^L, repeated over 10 epochs. The procedure for inference is identical, except the F^L update on line 15 is not performed and we use a single epoch with 10 ISTA steps.

4.3 Application to object recognition

Our model is purely unsupervised and so must be combined with a classifier to perform object recognition. In view of its simplicity and performance, we use the Spatial Pyramid Matching (SPM) of Lazebnik *et al.* [70].

[4] F^L appears in the reconstruction: $\hat{x}^L = R^L z^L = R^{L-1} U_{s^{L-1}} F^L z^L$.

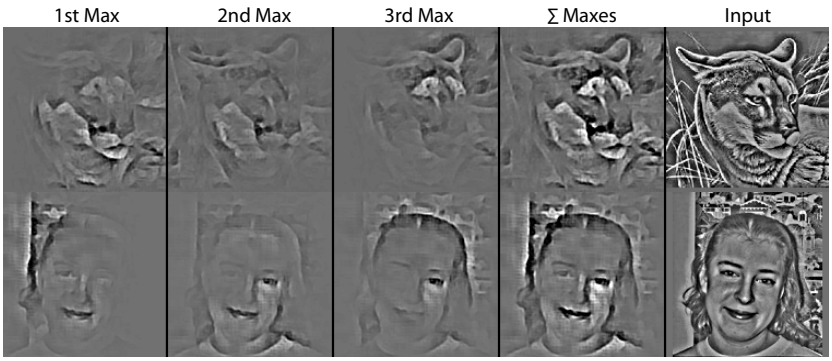

Figure 4.4: Col 1–3: The largest 3 absolute activations in the 4th layer projected down to pixel space for 2 different images. Note how distinct structures are reconstructed, despite the model being entirely unsupervised. See Section 4.3 for details on their use for recognition. Col 4–5: sum of first 3 columns; original input image.

Given a new image, performing inference with our model decomposes it into multiple layers of feature maps and switch configurations. We now describe a novel approach for using this decomposition in conjunction with the SPM classifier.

While the filters are shared between images, the switch settings are not, thus the feature maps of two images are not directly comparable since they use different bases R^L. For example, in Figure 4.1 each 4th layer top-down decomposition begins from the same feature map, yet gives quite different reconstructions. This shows two key aspects of the model: (i) within a class or between similar classes, the decompositions share similar parts and focus on particular regions of the image; and (ii) the adaptability of the switch settings allows the model to learn complex interactions with other classes. However, this makes direct use of the higher-level feature maps problematic for classification and we propose a different approach.

For each image im, we take the set of the N_m largest absolute activations from the top layer feature maps and project each one *separately* down to the input to create N_m different images $(\hat{x}_1, \ldots, \hat{x}_{N_m})$, each containing various image parts generated by our model. This only makes sense for high layers with large receptive fields. In Figure 4.4 we show the pixel space reconstructions of the top $N_m = 3$ 4th layer activations inferred for 2 different images. Note how they contain good reconstructions of select image structures,

as extracted by the model, while neighboring content is suppressed, providing a soft decomposition. For example, the 2nd max for the face reconstructs the left eye, mouth, and left shoulder, but little else. Conversely, the 3rd max focuses on reconstructing the hair. The structures within each max reconstruction consist of textured regions (e.g. shading of the cougar), as well as edge structures. They also tend to reconstruct the object better than the background.

Instead of directly inputting $\hat{x}_1, \ldots, \hat{x}_{N_m}$ to the SPM, we instead use the corresponding reconstructions of the 1st layer feature maps (i.e. $\hat{z}^1_1, \ldots, \hat{z}^1_{N_m}$), since activations at this layer are roughly equivalent to unnormalized SIFT features (the standard SPM input [70]). After computing separate pyramids for each $\hat{z}^1_m$, we average all N_m of them to give a single pyramid for each image. We can also apply SPM to the actual 1st layer feature maps z^1_k, which are far denser and have even coverage of the image[5]. The pyramids of the two can be combined to boost performance.

4.4 Experiments

We train our model on the entire training set of 3060 images from the Caltech-101 dataset (30 images per class with 101 object classes and 1 background class).

Pre-processing: Each image is converted to gray-scale and resized to 150×150 (zero padding to preserve the aspect ratio). Local subtractive and divisive normalization (i.e. the patch around each pixel should have zero mean and unit norm) is applied using a 13×13 Gaussian filter with $\sigma = 5$.

Model architecture: We use a 4 layer model, with 7×7 filters, and $E = 10$ epochs of training. Various parameters, timings and statistics are shown in Table 4.1. Due to the efficient inference scheme, we are able to train with many more feature maps and more data than other approaches, such as [19, 62, 78]. By the 4th layer, the receptive field

[5]Specific details: pixel spacing=2, patch size=16, codebook size=2000.

Property	Layer 1	Layer 2	Layer 3	Layer 4
# Feature maps $N_k{}^l$	15	50	100	150
Pooling size	3x3x3	3x3x2	3x3x2	3x3x2
λ^l	2	0.1	0.005	0.001
η^l	10^{-3}	10^{-4}	10^{-6}	10^{-8}
CPU Inference time	0.12 s	0.21 s	0.38 s	0.54 s
GPU Inference time	0.006 s	0.01 s	0.03 s	0.05 s
z pixel field	7x7	21x21	63x63	189x189
Feature map dims	156x156	58x58	26x26	15x15
# Filter Params	735	7,350	122,500	367,500
Total # z & s	378,560	178,200	71,650	37,500

Table 4.1: Parameter settings (top 4 rows) and statistics (lower 5 rows) of our model.

of each feature map element (z pixel field) covers the entire image, making it suitable for the novel feature extraction process described in Section 4.3. At lower layers of the model, the representation has many latent variables (i.e. z's and s's) but as we ascend, the number drops. Counterbalancing this trend, the number of filter parameters grows dramatically as we ascend and the top layers of the model are able to learn object specific structures.

Timings: With 3060 training images and $N_e = 10$ epochs, it takes around 5 hours to train the entire 4 layer model using a MATLAB implementation on a six-core CPU. For inference, a single epoch suffices with 10 ISTA iterations at each layer. The total inference time per image is 1.25 secs (see Table 4.1 for each layer). Additionally, since the algorithm is highly parallel at each layer, the use of a Nvidia GTX 480 GPU reduces training time down to 30 mins for the entire model and 0.1 secs to infer each image.

4.4.1 Model visualization

The top-down nature of our model makes it easy to inspect what it has learned. In Figure 4.5 we visualize the filters in the model by taking each feature map separately and picking the single largest absolute activation over the entire training set. Using the switch settings particular to that activation we project it down to the input pixel space. At layer 1 (Figure 4.5(a)), we see a range of oriented Gabors of differing frequencies and some

Our model - layer 1	$67.8 \pm 1.2\%$
Our model - layer 4	$69.8 \pm 1.2\%$
Our model - layer 1 + 4	$\mathbf{71.0 \pm 1.0}\%$
Chen *et al.* [19] layer-1+2 (ConvFA)	$65.7 \pm 0.7\%$
Kavukcuoglu *et al.* [62] (ConvSC)	$65.7 \pm 0.7\%$
Zeiler *et al.* [142] layer-1+2 (DN)	$66.9 \pm 1.1\%$
Boureau *et al.* [11] (Macrofeatures)	$70.9 \pm 1.0\%$
Jarrett *et al.* [56] (PSD)	$65.6 \pm 1.0\%$
Lazebnik *et al.* [70] (SPM)	$64.6 \pm 0.7\%$
Lee *et al.* [78] layer-1+2 (CDBN)	$65.4 \pm 0.5\%$
Our Caltech-256 Model - layer 1+4	$\mathbf{70.5 \pm 1.1}\%$

Table 4.2: Recognition performance on Caltech-101 compared to other approaches grouped by similarity (from top). Group 1: our approach; Group 2: related convolutional sparse coding methods with SPM classifier; Group 3: other methods using SPM classifier; group 4: our model with filters trained on Caltech-256 images.

DC filters. In layer 2 (Figure 4.5(b)), a range of edge junctions and curves can be seen, built from combinations of the 1st layer filters. For select filters (highlighted in color), we expand to show the 25 strongest activations across all images. Each group shows clustering with a certain degree of variation produced by the specific switch settings for that particular activation. See, for example, the sliding configuration of the T-junction (blue box). Reflecting their large receptive field, the filters in layer 3 (Figure 4.5(c)) show a range of complex compositions. The highlighted boxes show that the model is able to cluster quite complex structures. Note that the groupings produced are quite different to a pixel-space clustering of image patches since they are: (i) far from rectangular in shape; (ii) utilize the adaptable geometric transformations offered by the switches below. The 4th layer filters (Figure 4.5(d)) show fairly complete reconstructions of entire objects with groupings amongst objects of the same class or of similar shape.

To understand the relative sizes of each projection we also show the receptive fields for layers 1-4 in Figure 4.5(e). Finally, reconstructions from each layer of the model of 4 example input images are shown in Figure 4.5(f). Note that unlike related models, such as Lee *et al.* [78], sharp image edges are preserved in the reconstructions, even from layer 4.

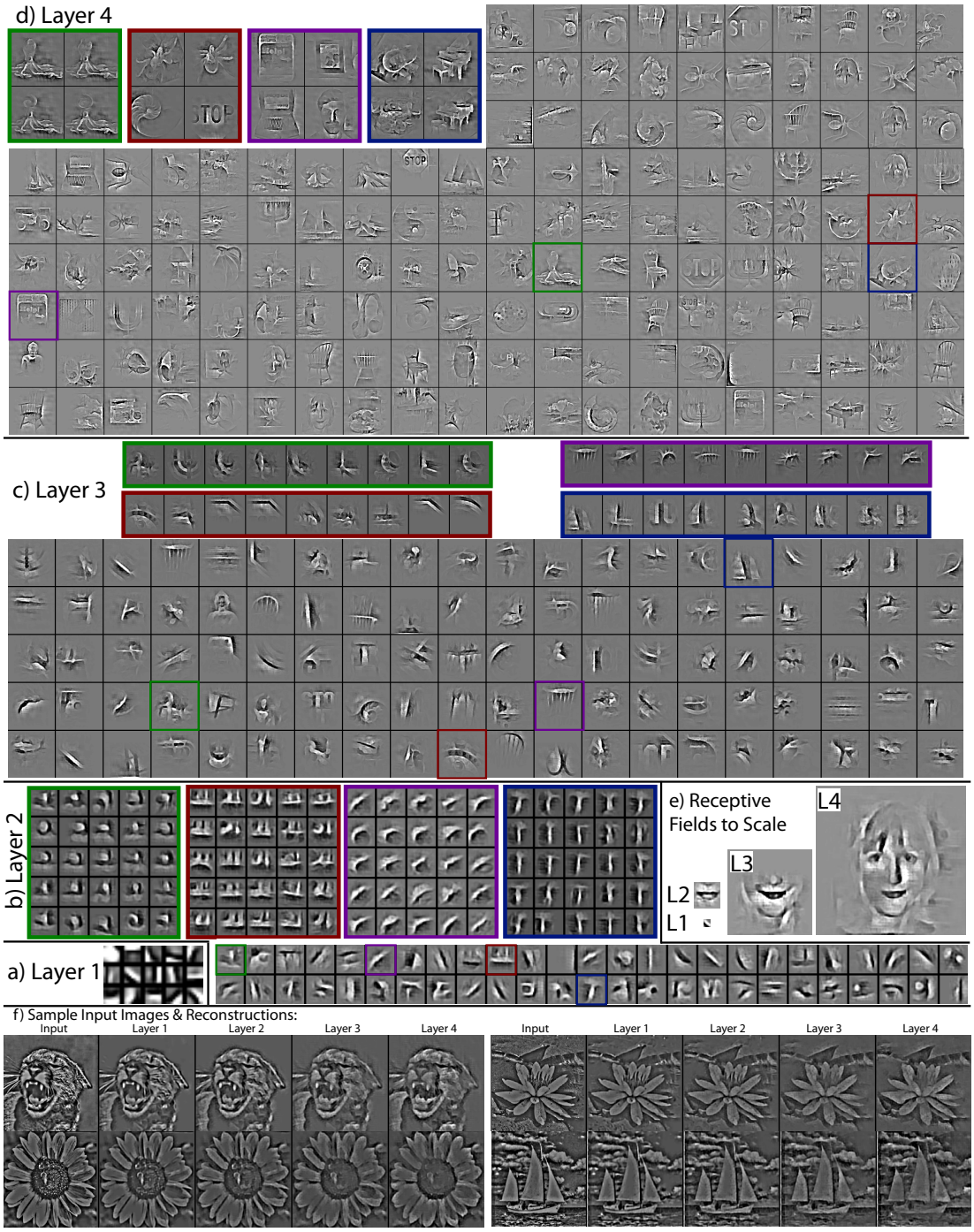

Figure 4.5: a-d) Visualizations of the filters learned in each layer of our model with zoom-ins showing the variability of select features. e) An illustration of the relative receptive field sizes. f) Image reconstructions for each layer. See Section 4.4.1 for explanation. This figure is best viewed in electronic form.

4.4.2 Evaluation on Caltech-101

We use $N_m = 50$ decompositions from our model to produce input for training the Spatial Pyramid Match (SPM) classifier of Lazebnik *et al.* [70]. The classification results on the Caltech-101 test set are shown in Table 4.2.[6]

Applying the SPM classifier to layer 1 features z^1 from our model produces similar results (67.8%) to many other approaches, including those using convolutional sparse coding (2nd group in Table 4.2). However, using the 50 max decompositions from layer 4 in the SPM classifier, as detailed in Section 4.3, we obtain a significant performance improvement of 2%, surpassing the majority of hierarchical and sparse coding approaches that also use the same SPM classifier (middle two groups in Table 4.2). Summing the SVM kernels resulting from the max activations from layer 4 and the layer 1 features, we achieve 71.0%. The only approach based on the SPM with comparable performance is that of Boureau *et al.* [11], based on Macrofeatures. Current state-of-the-art techniques [11,137,141] use forms of soft quantization of the descriptors, in place of the hard k-means quantization used in the SPM (e.g. Wang *et al.* [137] obtain 73.4% using a sparse-coding based classifier).

4.4.3 Evaluation on Caltech-256

Using the same training and evaluation parameters as for Caltech-101, we evaluated our model on the more difficult Caltech-256 dataset (see Table 4.3). By training our model on 30 images in each of the 256 categories and using $N_m = 50$ decompositions as before, we show a gain of 3.7% over SIFT features with the SPM classifier [141].

[6] In Table 4.2 and Table 4.3, we only consider approaches based on a single feature type. Approaches that combine hundreds of different features with multiple kernel learning methods outperform the methods listed. Also, only hard quantizatino approaches are shown.

Our model - layer 1	$31.2 \pm 1.0\%$
Our model - layer 4	$30.1 \pm 0.9\%$
Our model - layer 1 + 4	$\mathbf{33.2 \pm 0.8}\%$
Yang *et al.* [141] (SPM)	$29.5 \pm 0.5\%$
Our Caltech-101 Model - layer 1+4	$\mathbf{33.9 \pm 1.1}\%$

Table 4.3: Caltech-256 recognition performance of our model and a similar SPM method. Our Caltech-101 model was also evaluated.

4.4.4 Transfer learning

By using filters trained on Caltech-101, then classifying Caltech-256 and vice versa we can test how well our model generalizes to new images. In both cases, classification performance remains within errors of the original results (see Table 4.2 and Table 4.3) showing the adaptability of our model to generalize to new instances and entirely new classes.

4.4.5 Classification and reconstruction relationship

While we have shown sparsity to be useful for learning high level features and decomposing images into a hierarchy of parts, it is not necessarily a strong cue for classification as Rigamonti *et al.* [103] note. By varying the λ^l parameter at each layer for inference, holding the filters fixed, we analyzed the tradeoff of sparsity and classification performance on Caltech-101 in Figure 4.6. With higher λ^l values, sparsity is reduced, reconstructions

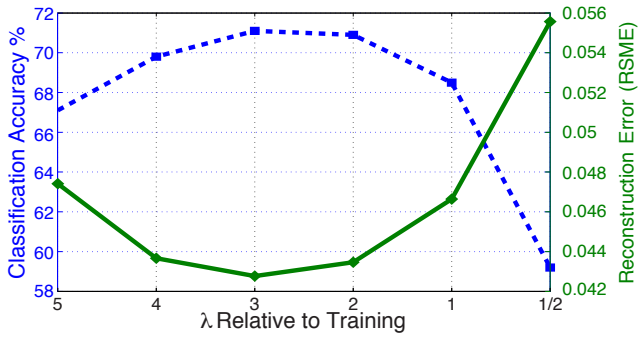

Figure 4.6: Relationship between reconstruction (solid line) and classification rates (dotted line) for varied amounts of sparsity.

improve, and recognition rates increase. The optimum appears around $3\times$ the λ^l used for training, after which the increased λ^l results in larger ISTA steps that introduce instability in optimization.[7]

4.4.6 Analysis of switch settings

Other deep models lack explicit pooling switches, thus during reconstruction either place a single activation in the center of each pool [19], or distribute it equally over all locations [78]. Figure 4.7 demonstrates the utility of switches: we (i) reconstruct using the top 25 activations in layers 2,3 and 4 for different forms of switch behavior; (ii) sum the resulting reconstructions and (iii) classify Caltech-101 using the layer 1 features (as before).

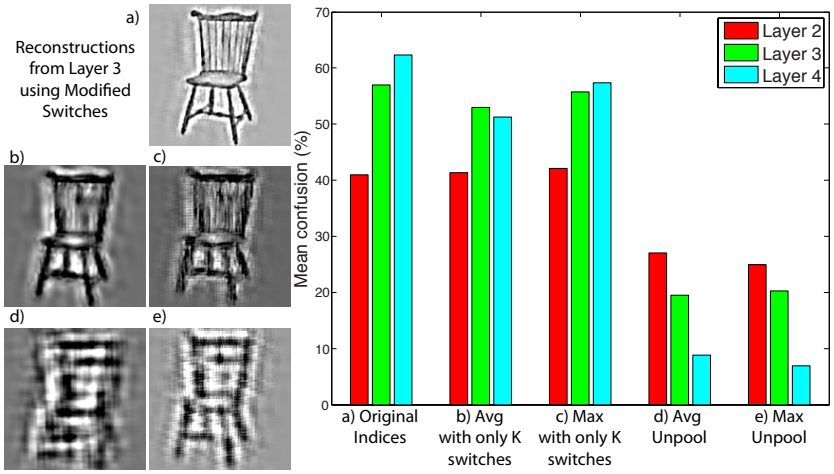

Figure 4.7: Switch importance for reconstruction and classification.

It is evident from the layer 3 reconstructions shown in Figure 4.7(a) that retaining all max locations allows for sharp reconstructions while average unpooling Figure 4.7(b,d) causes blur and using the center indices in max unpooling Figure 4.7(c,e) causes jitter with corresponding drops in recognition performance. When reconstruction, maintaining the proper k switches Figure 4.7(b,c) is crucial for selecting the proper feature maps in lower layers, so preventing extreme deformations of the objects (see Figure 4.7(d,e)) which leads to severely reduced recognition performance.

[7] The results in Table 4.2 and Table 4.3 used $2\times$ λ^l.

4.5 Discussion

The novel methods introduced in this chapter allow us to reliably learn models with many layers. As we ascend the layers, the switches in our model allow the filters to adapt to increasingly variable input patterns. The model is thus able to capture mid and high-level features that generalize between classes. Using these features with standard classifiers gives competitive rates on Caltech-101 and Caltech-256. The generality of our learned representation is demonstrated by its ability to generalize to datasets on which it was not trained, while maintaining a comparable performance. Matlab code for this chapter is available at `www.matthewzeiler.com/pubs/iccv2011/`.

Chapter 5

Differentiable Pooling for Hierarchical Feature Learning

5.1 Introduction

The max pooling in the previous chapter allowed deconvolutional networks to scale to large images and represent entire objects using a deep combination of convolutions and unpooling for reconstruction. Each reconstruction involves the unpooling operation which uses the discrete switch locations in each pooling region to place reconstructed activations in place. This provides decent reconstructions in deep models, but information is lost in this discretization step. This chapter proposes a novel pooling method which provides continuous translation and scale parameterization for each pooling region that is also differentiable. This facilitates a unified objective function which involves reconstruction from the pooled features of the top layer, as opposed to the unpooled features as in the last chapter, while allowing pooling parameters to be jointly optimized in all layers.

In many hierarchical image representations spatial pooling plays an important role in providing invariance to local perturbations of the input and allowing higher-level features

to model large portions of the image. Sum and max pooling are the most common forms, with max being typically preferred (see Boureau *et al.* [12] for an analysis).

This chapter uses a parametric form of pooling that can be directly integrated into the overall objective function of many hierarchical models. Using a Gaussian parametric model, we can directly optimize the mean and variance of each Gaussian pooling region during inference to minimize a global objective function. This contrasts with existing pooling methods that just optimize a local criterion (e.g. max over a region). Adjusting the variance of each Gaussian allows a smooth transition between selecting a single element (akin to max pooling) over the pooling region, or averaging over it (like a sum operation).

Integrating pooling into the objective facilitates joint training and inference across all layers of the hierarchy, something that is often a major issue in many deep models. During training, most approaches build up layer-by-layer, holding the output of the layer beneath fixed. However, this is sub-optimal, since the features in the low-layers cannot use top-down information from a higher layer to improve them. A few approaches do perform full joint training of the layers, notably the Deep Boltzmann Machine [108], and Eslami *et al.* [32], as applied to images, and the Deep Energy Models of Ngiam *et al.* [92]. We demonstrate our differentiable pooling in a third model with this capability, the deconvolutional networks introduced earlier. We show how joint inference and training of all layers is possible, using this differentiable pooling. However, differentiable pooling is not confined to the deconvolutional network model – it is capable of being incorporated into many existing hierarchical models as well.

The latent variables that control the Gaussians in our pooling scheme store location information ("where"), distinct from the features that capture appearance ("what"). This separation of what/where is also present in Ranzato *et al.* [99], the transforming auto-encoders of Hinton *et al.* [47], and the adaptive deconvolutional networks from the previous chapter.

In this work, we also explore a number of secondary issues that help with training deep models: non-negativity constraints; different forms of sparsity; overcoming local minima during inference and different sparsity levels during training and testing.

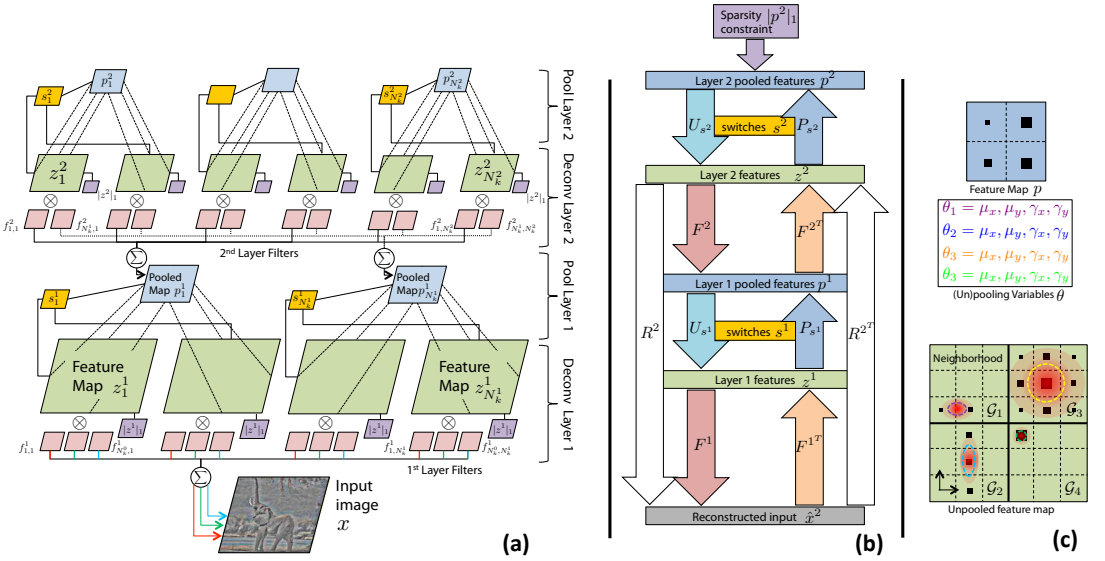

Figure 5.1: (a): A 2-layer model architecture. (b): Schematic of inference in a two layer model. (c): Illustration of the Gaussian parameterization used in our differentiable pooling.

5.2 Deconvolution Networks with Differentiable Pooling

We explain our contributions in the context of a deconvolutional network, introduced in the previous two chapters. This model is a hierarchical form of convolutional sparse coding that can learn invariant image features in an unsupervised manner. Its simplicity allows the easy integration of differentiable pooling and is amenable to joint inference over all layers.

Let us start by reviewing a single deconvolutional network layer, presented with an input image x (having ${N_k}^0$ color channels). The goal is to produce a reconstruction $\hat{x}$ from sparse pooled features p, that is close to x. We achieve this by minimizing:

$$C = \frac{\lambda}{2}\|\hat{x} - x\|_2^2 + |p|_\alpha \tag{5.1}$$

where λ is a hyper-parameter that controls the influence of the reconstruction term and the layer $[\,]^L$ notation will be omitted for clarity until needed. p consists of a set of N_k 2-D feature maps, thus forming an over complete-basis. To give a unique solution, a sparsity constraint on p is needed and we use an element-wise pseudo-norm where $0.5 \leq \alpha \leq 1$. The reconstruction $\hat{x}$ is produced from p by two alternating operations: *Unpooling* and *Convolution*. This is different from previous chapters where the reconstruction began from the unpooled features z. The difference involves a generalization to the unpooling operation which is explained next.

5.2.1 Unpooling

In the unpooling sub-stage, each 2D feature map p_k undergoes an unpooling operation to produce a larger 2D *unpooled* feature map z_k[1]. Each element j in p_k influences a small neighborhood $\mathcal{G}_j$ (typically 2×2 or 3×3) in the unpooled map z_k, via a set of weights $w_k(i)$ within the neighborhood:

$$z_k(i) = w_k(i)p_k(j) \quad \forall i \in \mathcal{G}_j \tag{5.2}$$

We constrain the weights $w_k(i)$ to have unit ℓ_2-norm, as this makes the unpooling operation invertible[2]. The inverse *pooling* operation computes each element j in p_k as the sum of weights $w_k(i)$ in neighborhood $\mathcal{G}_j$ of the unpooled map z_k:

$$p_k(j) = \sum_{i \in \mathcal{G}_j} w_k(i)z_k(j) \tag{5.3}$$

[1] 3D (un)pooling is also possible, as explored previous.

[2] Combining Eqs. 5.2 and 5.3, we have $p_k(j) = \sum_i w_k^2(i)p_k(j)$, hence $\sum_i w_k^2(i)$=1.

In the previous chapter, max (un)pooling was used, equivalent to $w_k(i)$ being all zero, except for a single element set to 1. In this work, we consider more general $w_k(i)$'s, as detailed in Section 5.2.3, treating them as latent variables which will be inferred for each input image. Note that each element in p has its own set of w's.

For the rest of this chapter, we consider the neighborhoods $\mathcal{G}_j$ to be non-overlapping, but the above formulation generalizes to overlapping regions as well. For brevity, we write the unpooling operation as a single linear matrix, parameterized by weights w: $z = U_w p$.

5.2.2 Updated Cost Function

The feature maps exist solely at the top of the model where the reconstruction begins from a layer L (there are no explicit features in intermediate layers, l). Thus, the *reconstruction* can be updated to include the variables of the intermediate layers, the filters F^l and unpooling weights w^l:

$$\hat{x}^L = F^1 U_{w^1} F^2 U_{w^2} \dots F^L U_{w^L} p^L = R^L p^L \tag{5.4}$$

where F^l and U_{w^l} are the convolutional and unpooling operations from each layer l. This lets us write the overall objective for a multi-layer model (shown here for a single image, x, but optimized over a set of images $x^1, \dots, x^{\mathcal{X}}$ during training):

$$C^L = \frac{\lambda}{2} \| R^L p^L - x \|_2^2 + |p^L|_\alpha \tag{5.5}$$

This integrated formulation allows the straightforward optimization of the features p^L of the top layer and the filters F^l and the (un)pooling weights w^l of lower layers to minimize a single objective function. A multi-layer model is shown in Figure 5.1(a). While most other models also learn filters and features, the pooling operation is typically fixed. Direct optimization of Eq. 5.5 with respect to w is one the main contributions of this

work and is described in Section 5.2.3.

Note that, given fixed weights w, the reconstruction is linear in p, thus Eq. 5.5 describes a tri-linear model, with w coding position (where) information about the (what) features p. Since the reconstruction R^L is linear, the reconstruction term is easily differentiable. The derivative of R^L is:

$$R^{L^T} = U_{w^L}^T F^{L^T} \dots U_{w^1}^T F^{1^T} \tag{5.6}$$

which is the updated *projection* operator. This takes a signal at the input and repeatedly convolves (using flipped versions of the filters at each layer) and pools (using weights w^l) all the way up to the features. This is a key operation for both inference and learning, as described in Section 5.3 and Section 5.4 respectively. Figure 5.1(b) illustrates the reconstruction and forward propagation operations.

The objective function in Eq. 5.5 differs from the original deconvolutional network formulation in several important ways. First, sparsity is imposed directly on p^L, as opposed to z^L. This integrates pooling into the objective function, allowing it to become part of the inference. Second, previously only $\alpha = 1$ was considered, whereas we consider the hyper-Laplacian ($\alpha < 1$) sparsity here which enforce a stronger sparsity penalty. Third, p^L is non-negative, as opposed to previously where there was no such constraint. Fourth, and most importantly, by inferring the optimal (un)pooling weights w we directly minimize the objective function of the model. Fixed sum or max pooling, employed by other approaches, is a local heuristic that has no clear relationship to the overall cost.

5.2.3 Differentiable Pooling

We impose a parametric form on the (un)pooling weights w to ensure that the features are invariant to small changes in the input. The pooling would otherwise be able to memorize perfectly the unpooled features, giving "lossless" pooling which would not generalize at all.

The parametric model we use is a 2D axis-aligned Gaussian, with mean (μ_x, μ_y) and precision (γ_x, γ_y) over the pooling neighborhood $\mathcal{G}_j$, introduced in Section 5.2.1. The Gaussian is normalized within the extent of the pooling region to give weights w whose square sums to 1 (thus giving unit ℓ_2 norm):

$$w(i) = \frac{\sqrt{v(i)}}{\sqrt{\sum_{i' \in \mathcal{G}} v(i')}} \tag{5.7}$$

where $v(i)$ is value of the Gaussian for element i, at the spatial locations $\text{sp}_x(i), \text{sp}_y(i)$ within the neighborhood $\mathcal{G}_j$:

$$v(i) = e^{-[\frac{\gamma_x}{2}(\text{sp}_x(i)-\mu_x)^2 + \frac{\gamma_y}{2}(\text{sp}_y(i)-\mu_y)^2]} \tag{5.8}$$

Figure 5.1(c) shows an illustration of this parameterization. For brevity, we let $\theta_j = \{\mu_x, \mu_y, \gamma_x, \gamma_y\}$ be the parameters for neighborhood $\mathcal{G}_j$. We thus rewrite the unpooling operation in U_{w^l} as U_{θ^l}. The reconstruction operator R^L from layer L is also parameterized by θ from each layer below L which makes this an image dependent operator, but we omit the θ in the notation for R to avoid clutter. The Gaussian representation has several advantages over existing sum or max pooling:

- Varying the mean of the Gaussian selects a particular region in the unpooled feature map, just like max pooling. This makes the feature invariant to small translations within the unpooled maps.
- Varying the precision of the Gaussian allows a smooth variation between max and sum operations (high and low precision respectively).
- Changes in precision allow invariance to small scale changes in the unpooled features. For example, the width of an edge can easily be altered by adjusting the variance (see Figure 5.2(c)).

- The continuous nature of the Gaussian allows sub-pixel reconstruction that avoids aliasing artifacts, which can occur with max pooling. See Figure 5.5 for an illustration of this.
- The Gaussian representation is differentiable, i.e. the gradient of Eq. 5.5 with respect to θ_j has analytic form, as detailed in Section 5.3.2.

5.2.4 Non-Negativity

In standard sparse coding and other learning methods both the feature activations and the learned parameters can be positive or negative. This contrasts with our model, in which we enforce non-negativity.

This is motivated by several factors. First, there is no notion of negative intensities or objects in the visual world. Second, the Gaussian parameterization used in the differentiable pooling scheme, described in Section 5.2.3 has positive weights, so cannot represent individual negative values in the unpooled feature maps. Third, there is some biological evidence for non-negative representations within the brain [52]. Finally, we find experimentally that non-negativity reduces the flexibility of the model, encouraging it to learn good representations. The features computed at test-time have improved classification performance, compared with models without this constraint (see Section 5.6.4).

5.2.5 Hyper-Laplacian Sparsity

Most sparse coding models utilize the ℓ_1-norm to enforce a sparsity constraint on the features [93], as a proxy for optimizing ℓ_0 sparsity [129]. However, a drawback of this form of regularization is that it gives the same cost to two elements being 0.5 versus a single elements at 1 and the other at 0, even though the latter has a lower ℓ_0 cost.

To encourage features with lower ℓ_0 cost, we use a pseudo-norm $\ell_{0.5}$ (i.e. $\alpha = 0.5$ in Eq. 5.5) inspired by Krishnan and Fergus [64], which aggressively pushes small elements

toward zero. To optimize this, we experimented with closed form optimization techniques in [64], but settled on gradient descent for simplicity.

5.3 Inference

During inference, the filters F at all layers are fixed and the objective is to find the features p and (un)pooling variables θ for all neighborhoods and all layers that minimize Eq. 5.5. We do this by alternating between updating the features p and the Gaussian variables θ, while holding the other fixed.

5.3.1 Feature Updates

For a given layer L, we seek the features p^L that minimize C^L (Eq. 5.5), given an input image x, filters $F^1, \ldots, F^L$ and unpooling variables $\theta^1, \ldots, \theta^L$. This is a large convolutional sparse coding problem and we adapt the ISTA scheme of Beck and Teboulle [4]. This uses an iterative framework of gradient and shrinkage steps.

Gradient step: The gradient of C^L with respect to p^L is:

$$\nabla_p^L = \frac{\partial C^L}{\partial p^L} = R^{L^T}(R^L p^L - x) \tag{5.9}$$

This involves first reconstructing the input from the current features: $\hat{x}^L = R^L p^L$, computing the error signal $e^L = (\hat{x}^L - x)$, and then forward propagating this up to compute the top layer gradient $\nabla_p^L = R^{L^T} e^L$. Given the gradient, we then can update p^L:

$$p^L = p^L - \lambda^L \eta_{p^L} \nabla_p^L \tag{5.10}$$

where the η_{p^L} parameter sets the size of the gradient step.

Shrinkage step: Following the gradient step, we perform a per-element shrinkage operation that clamps small elements in p^L to zero, increasing its sparsity. For $\alpha = 1$, we use the standard ℓ_1 shrinkage:

$$p^L = \max(|p^L| - \eta_{p^L}, 0) \cdot \text{sign}(p^L) \tag{5.11}$$

For $\alpha = 0.5$, we step in the direction of the gradient:

$$p^L = p^L - \eta_{p^L} \frac{1}{2} \sqrt{|p^L|}^{-1} \cdot \text{sign}(p^L) \tag{5.12}$$

Projection step: After shrinking small elements away, the solution is then projected onto the non-negative set:

$$p^L = \max(p^L, 0) \tag{5.13}$$

Step size calculation: In order to set a learning rate for the feature map optimization, we employ an estimation technique for steepest descent problems [115] which uses the gradients $\nabla_p^L = \frac{\partial C^L}{\partial p^L}$:

$$\eta_{p^L} = \frac{\nabla_p^{L^T} \nabla_p^L}{\nabla_p^{L^T} R^{L^T} R^L \nabla_p^L} \tag{5.14}$$

Automating the step-size computation has two advantages. First, each layer requires a significantly different learning rate on account of the differences in architecture, making it hard to set manually. Second, by computing the step-size before each gradient step, each ISTA iteration makes good progress at reducing the overall cost. In practice, we find fixed step-sizes to be significantly inferior.

∇_p^L is computed once per mini-batch. For efficiency, instead of computing the denominator in Eq. 5.14 for each image, we estimate it by selecting a small portion ($\sim$10%) of each mini-batch and then averaging the final learning rates.

Reset step: Repeated optimization of the objective function tends to get stuck in local minima as it proceeds over the dataset for several epochs. We found a simple and effective way to overcome this problem. By setting all feature maps p^L to 0 every few epochs (essentially re-initializing inference), cleaner filters and better performing features can be learned, as demonstrated in Section 5.6.5.

This reset may be explained as follows. During alternating inference and learning stages, the model can overfit a mini-batch of data by optimizing either the filters or feature maps too much. This causes the model to lock up in a state where no new feature map elements can turn on because the reconstruction performance is sufficient to have only a small error propagating forward to the feature level. Since no new features turn on after shrinkage, the filters remain fixed as they continue to get the similar gradients. This can happen early in the learning procedure when the filters are still not optimal and therefore the learned representation suffers. By resetting the feature maps, at the next epoch the model has to reevaluate how to reconstruct the image from scratch, and can therefore turn on the optimal feature elements and continue to optimize the filters.

5.3.2 (Un)pooling Variable Updates

Given a model with N_l layers, we wish to update the (un)pooling variables θ^l at each intermediate layer l to optimize the objective C^L where the layer we reconstruct from L is the top layer N_l of the model. We assume that the filters $F_1, \ldots, F_{N_l}$ and features p^L are fixed.

The gradients for the pooling variables θ^l involve combining, at layer l, the forward propagated error signal with the top down reconstruction signal. This combined signal then drives the update of the pooling variables. More formally:

$$\frac{\partial C^L}{\partial U_{\theta^l}} = R^{l^T}(R^L p^L - x) \cdot (R^{(L \to l)} p^L) \tag{5.15}$$

where $R^{L\to l}$ is the top down reconstruction from layer L feature maps to layer l feature maps and R^{l^T} is the error propagation up to z^l.

With the chosen Gaussian parameterization of the pooling regions, the chain rule can be used to compute the gradient for each parameter $\theta^l = \{\mu_x, \mu_y, \gamma_x, \gamma_y\}$:

$$\nabla_\theta^l = \frac{\partial C^L}{\partial \theta^l(j)} = \sum_{\tilde{i}\in\mathcal{G}_j} \frac{\partial C^L}{\partial U_{\theta^l}(\tilde{i})} \frac{\partial U_{\theta^l}(\tilde{i})}{\partial w(\tilde{i})} \sum_{i\in\mathcal{G}_j} \frac{\partial w(\tilde{i})}{\partial v(i)} \frac{\partial v(i)}{\partial \theta^l(j)} \tag{5.16}$$

where j is the neighborhood index,

$$\frac{\partial U_{\theta^l}(\tilde{i})}{\partial w(\tilde{i})} = \hat{p}^l(\tilde{i}) = (R^{(L\to l)} p^L)(\tilde{i}) \tag{5.17}$$

$$\frac{\partial w(\tilde{i})}{\partial v(i)} = (\sum_{n\in\mathcal{G}} v(n))^{-1}[w(i)] \tag{5.18}$$

$$\frac{\partial w(\tilde{i})}{\partial v(\tilde{i})} = (\sum_{n\in\mathcal{G}} v(n))^{-1}[1 - w(\tilde{i})] \tag{5.19}$$

$$\frac{\partial v(i)}{\partial \mu_x(j)} = \gamma_x(j)(\mathrm{sp}_x(i) - \mu_x(j))v(i) \tag{5.20}$$

$$\frac{\partial v(i)}{\partial \mu_y(j)} = \gamma_y(j)(\mathrm{sp}_y(i) - \mu_y(j))v(i) \tag{5.21}$$

$$\frac{\partial v(i)}{\partial \gamma_x(j)} = -\frac{1}{2}(\mathrm{sp}_x(i) - \mu_x(j))^2 v(i) \tag{5.22}$$

$$\frac{\partial v(i)}{\partial \gamma_y(j)} = -\frac{1}{2}(\mathrm{sp}_y(i) - \mu_y(j))^2 v(i) \tag{5.23}$$

Algorithm 3 Learning with Differentiable Pooling in deconvolutional networks

Require: Training set $\mathcal{X}$, # layers N_l, # epochs N_e, # ISTA steps N_t
Require: Regularization coefficients λ^l, # feature maps ${N_k}^l, \forall l < N_l$
Require: Pooling step sizes η_{U^l}

for $L = 1 : N_l$ **do** %% Loop over the layers to reconstruct from
  Init. features/filters: $p^L \sim 0$, $F^L \sim \mathcal{N}(0, \epsilon)$
  Init. switches: $\theta^l = Fit({R^l}^T x) \ \ \forall l < L$
  for epoch $= 1 : N_e$ **do** %% Epoch iteration
    for $im = 1 : N_x$ **do** %% Loop over images. Note: z, p, and θ are per-image
      for $t = 1 : N_t$ **do** %% ISTA iteration
        Reconstruct input: $\hat{x}^L = R^L p^L$
        Compute reconstruction error: $e^L = \hat{x}^L - x$
        Propagate error up to layer L: $\nabla_p^L = {R^L}^T e^L$
        Estimate step size η_{p^L} as in Eq. 5.14
        Take gradient step on p: $p^L = p^L - \lambda^L \eta_{p^L} \nabla_p^L$
        Perform shrink: $p^L = \max(|p^L| - \eta_{p^L}, 0)\text{sign}(p^L)$
        Project to positive: $p^L = \max(p^L, 0)$
        for $l = 1 : L$ **do** %% Loop over lower layers
          Take gradient step on θ: $\theta^l = \theta^l - \lambda^L \eta_{U^l} \nabla_\theta^l$
        end for
      end for
    end for
    Update F^L by solving Eq. 5.25 using CG
    Project F^L to positive and unit length
  end for
end for
Output: filters F, feature maps p and pooling variables θ.

where $\text{sp}_x(i)$ and $\text{sp}_y(i)$ are the coordinates within the pooling neighborhood $\mathcal{G}_j$.

Once the complete gradient is computed as in Eq. 5.16, we do a gradient step on each pooling variable:

$$\theta^l = \theta^l - \lambda^L \eta_{U^l} \nabla_\theta^l \tag{5.24}$$

using a fixed step size η_{U^l}. We experimented with a similar step size to Eq. 5.14 for the pooling parameters, however found the estimates to be unstable, likely due to the nonlinear derivatives involved in the Gaussian pooling.

5.4 Learning

After inference of the feature maps for the top layer and (un)pooling variables for all layers is complete, the filters in each layer are updated. This is done using the gradient with respect to each layer's filters:

$$\frac{\partial C^L}{\partial (F_{c,k})^l} = \lambda^L [R^{l-1^T}(R^L p^L - x)]_c \otimes [(U_{\theta^l} R^{(L \to l)} p^L)]_k \tag{5.25}$$

where the left term is the bottom up error signal propagated up to the feature maps p^{l-1} below the given filters via R^{l-1^T}, and the right term is the top down reconstruction to the unpooled feature maps z^l. The gradient is therefore the convolution between all combinations of input error maps to the layer (indexed by c) and the unpooled feature maps reconstructed from above (indexed by k), resulting in updates of each filter plane $F^l_{c,k}$, for each layer l.

In practice we use batch conjugate gradient updates for learning the filters as the model is linear in F^l once the feature maps and pooling parameters are inferred. After 2 steps of conjugate gradients, the filters are projected to be nonnegative and renormalized to unit ℓ_2 length.

5.4.1 Joint Inference

The objective function explicitly constrains the reconstruction from the top layer features to be close to the input image. From this we can calculate gradients for each layer's filters and pooling variables while optimizing the top level features maps. Therefore for each image we can infer the local shifts and scalings of low level features as the high level concepts develop.

We have found that pre-training the first layer in one phase of training and then using the pooling variables and learned layer 1 filters to initialize a second phase of training

works best. The second phase of training optimizes the second layer objective from which we can update p^2, U_{w^2}, U_{w^1}, F^2, and F^1 jointly. If care is not taken in this joint update, the first layer features can trade off representation power with the second layer filters. This can result in the second layer filters capturing the details while the first layer filters become dots. To avoid this problem, after the first phase of training we hold F^1 fixed and optimize the remaining variables jointly. Thus, while the filters are learned layer-by-layer, inference is always performed jointly across all layers. This has the nice property that these low level parts can move and scale as the U_{w^1} variables are optimized while the high level concepts are learned.

5.5 Initialization of Parameters

Before training, the filter parameters are initialized to zero mean Gaussian distributed random values with $1e^{-2}$ standard deviation. After this random initialization, the filters are projected to be non-negative and normalized to unit length before training begins.

Before inference, either at the start of training or at test time, we initialize the features maps to 0. This creates a reconstruction of 0 in the pixel space, therefore the initial gradient being propagated up the network is $-x$. This is similar to a feedforward network for the first iteration of inference. While forward propagating this signal up the network we can leverage the Gaussian parameterization of the pooling regions to fit these pooling parameters using moment matching. That is, at each layer, we extract the optimal pooling parameter that fit this bottom up signal. This provides a natural initialization to both the pooling variables at each layer and the top level feature activations given the input image and the filter initialization.

5.6 Experiments

Evaluation on MNIST We choose to evaluate our model on the MNIST handwritten

digit classification task. This dataset provides a relatively large number of training instances per class, has many other results to compare to, and allows easy interpretation of how a trained model is decomposing each image.

Pre-processing: The inputs were the unprocessed MNIST digits at 28x28 resolution. Since no preprocessing was done, the elements remained nonnegative.

Model architecture: We trained a 2 layer model with 5x5 filters in each layer and 2x2 non-overlapping pooling regions. The first layer contained 16 feature maps and the second layer contained 48 features maps. Each of these 48 feature maps connect randomly to 8 different layer 1 feature maps through the second layer filters. These sizes were chosen comparable to our previous work while being more amenable to GPU processing. The receptive fields of the second layer features are 14x14 pixels with this configuration, or one quarter the input image size.

Classification: One motivation of this work was to analyze how the classification pipeline of the previous chapter could be simplified by making the top level features of the network more informative. Therefore, in these experiments we simply treat the top level activations inferred for each image as input to a linear SVM [33].

The only post processing done to these high level activations is that overlapping patches are extracted and pooled, analogous to the dense SIFT processing which is shown by many computer vision researchers to improve results [11]. This step provides an expansion in the number of inputs, allowing the linear SVM to operate in a higher dimensional space. For layer 1 classification these patches were 9x9 elements of the layer 1 features maps. For layer 2 they were 6x6 patches, roughly the same ratio to the feature map size as for layer 1. These patches were concatenated as input to the classifier. Throughout the experiments we did not combine features from multiple layers, concatenating only layer 1 patches together for layer 1 classification and only layer 2 features together for layer 2 classification. These final inputs to the classifier were each normalized to unit length.

Hyperparameters: By cross validating on a 50,000 train and 10,000 validation set of MNIST images, we found that $\lambda^1 = 2$ and $\lambda^2 = 0.5$ gave optimal classification performance. Each layer was trained with 100 ISTA steps/epoch for 50 epochs (passes through the dataset). After epoch 25, the feature maps were reset to 0 during training. At test time, we found higher $\lambda^1 = 5$ and $\lambda^2 = 5$ improved classification, as did optimizing for only 50 ISTA steps of inference.

5.6.1 Model visualization

By visualizing the filters and features maps of the model, we can easily understand what it has learned. In Figure 5.2 (a) we demonstrate sharp reconstructions of the input images from the second layer features maps. In Figure 5.2 (b) we display the raw filter coefficients for layer 1 which have learned small pieces of strokes. By incorporating the pooling parameters into the layer, these filters are robust to small changes in the input.

Visualizing these invariances of a model can be helpful in understanding the inputs the model is sensitive to. Searching through the dataset of inferred feature map activations and selecting the maximum element per feature map to project downward into the pixel space is one way of visualizing these invariances. However, these selected elements are only exemplars of inputs that most strongly activated that feature. In Figure 5.2(c) we show a more representative selection of invariances by instead selecting a feature activation to be projected down based on sampling from the distribution of activations for that feature in the dataset. This gives a less biased view of what activates that feature than selecting the largest few activations from the dataset. Once a sample is selected for a given feature map, the pooling variables corresponding to the image from which the activation was selected are used in the unpooling stages to do the top down visualization.

Examining the 16 sample visualizations for each feature in Figure 5.2(c) shows the scale and shifts that the Gaussian pooling provides to these relatively simple first layer fil-

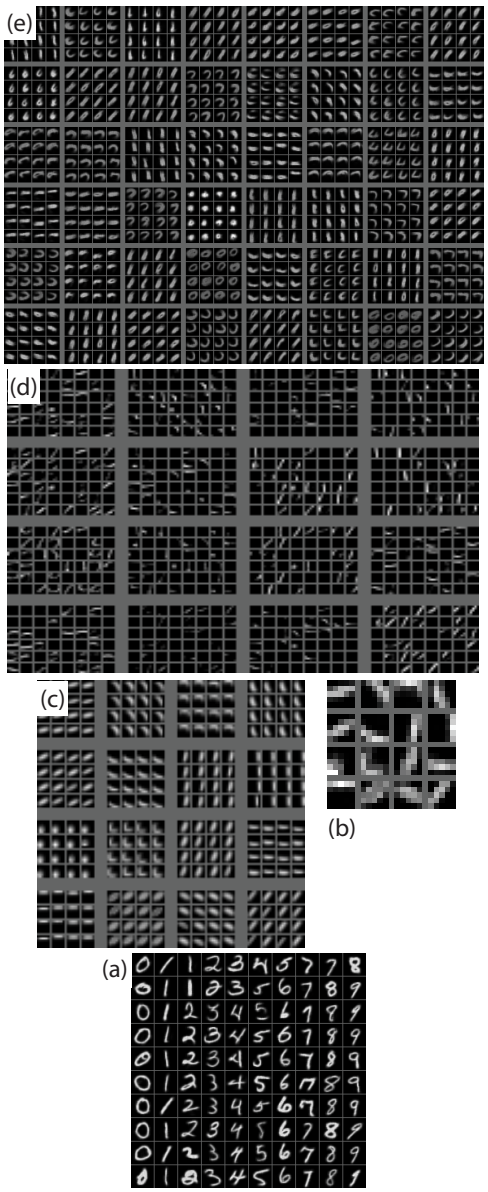

Figure 5.2: Visualization of the trained model: (a) reconstructions from layer 2, (b) the 16 layer 1 filter weights, (c) invariance visualization for layer 1 incorporating unpooling and convolutions (see Section 5.6.1 for details) (d) layer 2 filter weights (shown as 16 groups of filter planes connecting to all 48 layer 2 maps), (e) layer 2 pixel space invariance visualization of features projected down from samples of the layer 2 feature distribution (see Section 5.6.1).

ters. We can continue to analyze the model by viewing the layer 2 filters planes in Figure 5.2(d). Each of the 48 second layer features has 16 filter planes (shown in separate groups), one connecting to each of the layer 1 feature maps. While the second layer

filters are difficult to understand directly, we can visualize the learned representation of the second layer by projecting down all the way to the pixel space through layer 1. Figure 5.2(e) shows for each of the 48 feature maps a 4x4 grid of pixel space projections obtained by sampling 16 activations from the distribution of activations of each layer 2 feature and projecting down via alternating convolution and unpooling with the corresponding pooling variables separately for each activation.

While analyzing the features in pixel space is informative, we have also found it is useful to view the features as decompositions of an input image to know how the model is representing the data. One possible method of displaying the decomposition is by coloring each pixel of the reconstruction according to which feature it came from. Each feature is assigned a hue (in no particular order) and the associated reconstruction produced then defines the saturation of that color. The resulting image therefore depicts the high level feature assignments. Pixels with brownish colors indicate a summation of several colors (features) together. Note that the input images themselves are grayscale – the colors are just for visualization purposes.

In Figure 5.3(d) we show such a reconstruction from layer 1 for the original image in (e). To understand the model we also show the layer 1 feature map activations in (a) with their corresponding color assignment around them. Notice the sparse distribution of activations can reconstruct the entire image by utilizing the Gaussian pooling and layer 1 filters in (c). Figure 5.3(b) shows the result of this unpooling operation on the feature maps. Notice in the orange and purple boxes the elongated lines in the unpooled maps, made possible by a low precision in one dimension.

Figure 5.4 takes this analysis one step further by using the second layer of the model. Starting from 3 features in the layer 2 feature maps as shown in (a), they are unpooled (as shown in (b)) and then convolved with the second layer filters to reconstruct many elements down on to the first layer features maps (c). These are further unpooled to (d) where again you can see the benefits of the Gaussian pooling smoothly transitioning

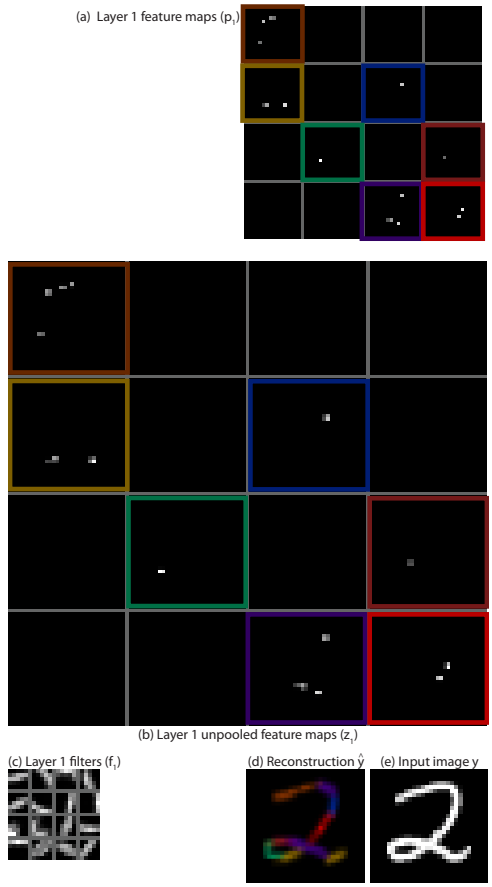

Figure 5.3: One layer decomposition of a digit into parts. From the top down the layer 1 feature maps (a) are unpooled into (b) and convolved with (e) to produce the reconstruction (f). The colors in the reconstruction simply represent which feature the reconstructed pixel came from.

between non-overlapping pooling regions. These are finally convolved with first layer filters (e) to give the decomposition shown in (f). Notice how long range structures are grouped into common features in the higher layer compared to the layer 1 decomposition

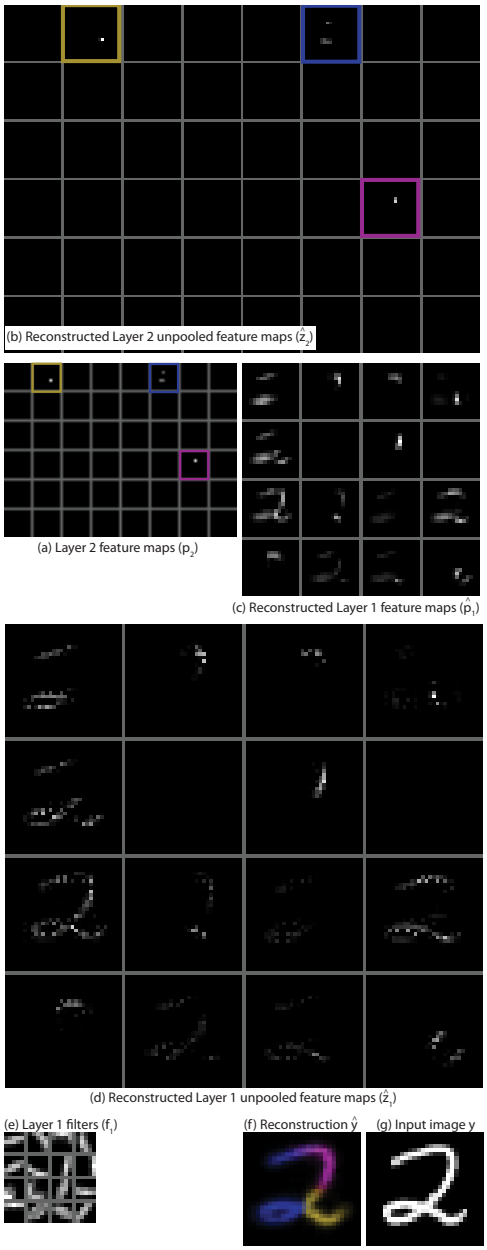

Figure 5.4: Two layer decomposition of a digit into parts. From the top down the layer 2 feature maps (a) are unpooled into (b) and convolved within the layer 2 filters to produce the reconstruction of layer 1 feature maps (c). These are unpooled into (c) and convolved with (e) to produce (f), colored according to the layer 2 feature that the reconstructed pixel it was reconstructed from.

of Figure 5.3.

5.6.2 Max Pooling vs Gaussian Pooling

The discrete locations that max pooling allows within a region are a limiting factor in the reconstruction quality of the model. Figure 5.5 (bottom) shows a significant aliasing effect is present in the visualizations of the model when max pooling is used. With the complex interactions between positive and negative elements removed, the model is not able to form smooth transitions between non overlapping pooling regions even though the filters used in the succeeding convolution sublayer have overlap between regions. Using the Gaussian pooling, the model can infer the desired precisions and means in order to optimize the reconstruction quality from high layers of the model.

This fine tuning of reconstruction allows for improvements without significantly varying the features activations (ie. maintains or decreases the sparsity while adjusting the pooling parameters). This is confirmed in Figure 5.6 where we break down the cost function into the reconstruction and regularization terms. In this figure we also display the ℓ_0 sparsity of each model as this can directly be used for comparison.

The Gaussian pooling significantly outperforms Max pooling in terms of optimizing the objective. By not being able to adjust the pooling variables to optimize the overall cost, Max pooling plateaus despite running for many epochs. Additionally it has a much higher ℓ_0 cost throughout training. In contrast, the ℓ_0 cost with Gaussian pooling decreases smoothly throughout training because the model can fine tune the pooling parameters to explain much more with each feature activation. This property is shown in Table 5.1 to significantly improve classification performance compared to Max pooling when stacking.

	Layer 1	Layer 2
Max Pooling	1.30%	1.25%
Gaussian Pooling	1.38%	0.84%

Table 5.1: MNIST error rate of Max pooling versus Gaussian pooling for 1 and 2 layer models. Note the performance improvement when stacking layer with Gaussian pooling.

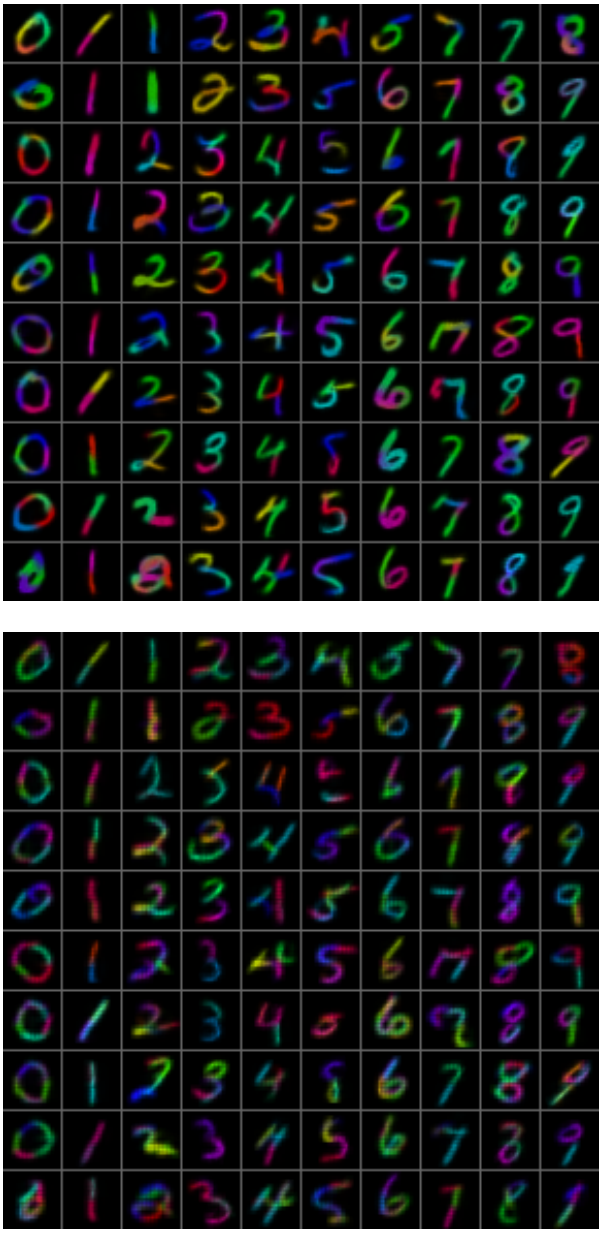

Figure 5.5: Feature decomposition comparison between Gaussian pooling (top) and max pooling (bottom). Each reconstructed pixel's color corresponds to the layer 2 feature map it was reconstructed from. Note the reuse of similar strokes in digits of a different class. Aliasing artifacts are present in the reconstructions using max pooling – see Section 5.6.2.

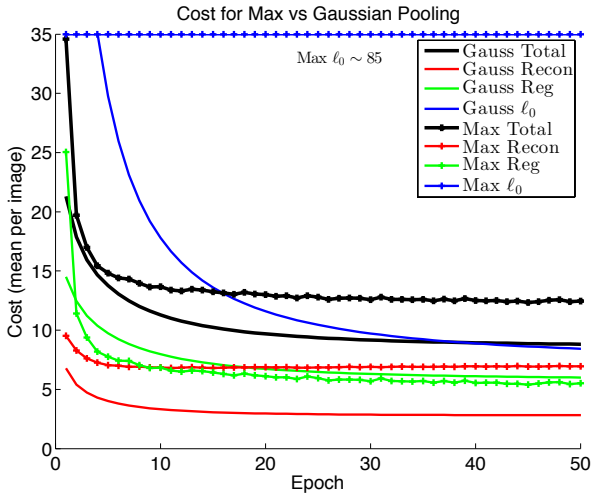

Figure 5.6: Breakdown of cost function into reconstruction and regularization terms for Max and Gaussian pooling for 2 layer models. Gaussian pooling gives consistently lower cost than max pooling. Furthermore, the ℓ_0 sparsity (shown in blue), is significantly lower for the Gaussian pooling, although not explicitly part of the cost.

5.6.3 Joint Inference

One of the main criticisms of sparse coding methods is that inference must be conducted even at test time due to the lack of a feedforward connection to encode the features. In our approach we discovered two fundamental techniques that mitigate this drawback.

The first is that running a joint inference procedure over both layers of our network improves the classification performance compared to running each layer separately. Instead of inferring the feature maps and pooling variables for the first layer and then using these pooling variables to initialize the second layer inference (2 phases), we can directly run inference with a two layer model. The differentiable pooling allows us to infer the pooling variables of both layers in addition to the layer 2 feature values simultaneously in 1 phase. At the first iteration of inference we leverage the ability to fit the Gaussian pooling parameters in a feed forward way as mentioned in Section 5.5. This halves the number of inference iterations needed by not requiring any first layer inference prior to inferring the second layer.

To examine this first discovery in depth we considered several combinations of how to

joint train and then run inference at test time with this model. During training we have found both qualitatively in terms of feature diversity and quantitatively in terms of classification performance that training in separate phases, one for each layer of the model, works better than jointly training both layers from scratch. In the second phase of training, when optimizing for reconstruction from the second layer feature maps, the first layer pooling variables and filters can either be updated or held fixed. Each row of Table 5.2 examines each combination of these updates during training. We can see that the optimal training scheme was with fixed first layer filters but pooling updates on both layers. This made the system more stable while still allowing these first layer filters to move and scale as needed by updating the first layer pooling variables.

In all cases we see a significant reduction in error rates when doing inference in 1 phase. The middle column of the table shows this 1 phase inference, but without optimizing the first layer pooling parameters U^1 whereas the last column does optimize U^1. We see an improvement in updating U^1 for all but the last row which was trained without U^1 and so is accustomed to that type of inference. This improvement with joint inference of U^1, U^2, and p^2 is a key finding which is only possible with differentiable pooling.

Training	Infer 2 phases	Infer 1 phase (no U^1)	Infer 1 phase
Updating F^1 U_{w^1}	1.79%	1.63%	1.40%
Updating F^1	1.71%	1.21%	1.10%
Updating U_{w^1}	1.39%	1.04%	0.84%
No Layer 1 Updates	1.46%	0.99%	1.03%

Table 5.2: Comparison of joint training techniques. Each row is a trained two layer model that updates select variables in layer 1 during training (in addition to F^2 and U_{w^2}). The three columns use these models but run inference at test time in 2 phases, 1 phase without updating U^1, and 1 phase with all updates respectively.

The second discovery that reduces evaluation time is that running the same number of ISTA iterations as was done during training does not give optimal classification performance, possibly due to over-sparsification of the features. Similarly running with too few iterations also reduces performance. Figure 5.7 shows a plot comparing the number of ISTA iterations to the classification performance with an optimum at 50 ISTA steps,

half the number used during training.

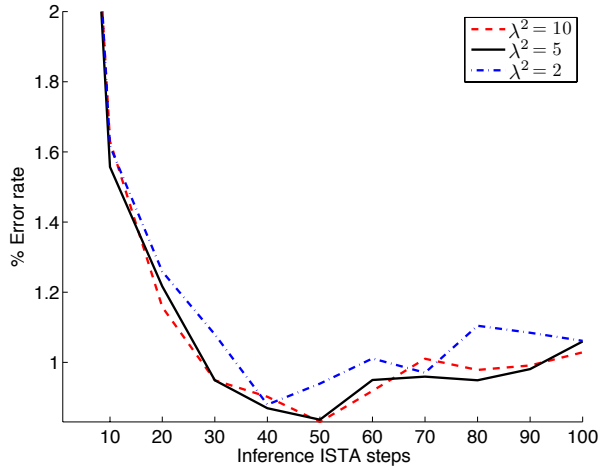

Figure 5.7: Comparison of MNIST digit classification errors versus number of ISTA steps used during inference.

5.6.4 Effects of Non-Negativity

With negative elements present in the system, many possible solutions can be found during optimization. This happens because subtractions allow the removal of portions of high level features. This has the effect of making them less discriminative because the model can change parameters in-between the high level feature activations and the input image in order to reconstruct better while assigning less meaning to the feature activations themselves.

To show this is not an artifact of the Gaussian pooling being more suited to nonnegative systems (due to the summation over the pooling region possibly leading to cancellations if negatives are present), we include comparison in Table 5.3 to Max pooling. In both cases, enforcing positivity via projected gradient descent improves the discriminative information preserved in the features.

	Positive/Negative	Non-negative
Max Pooling	2.04%	1.25%
Gaussian Pooling	2.32%	0.84%

Table 5.3: MNIST error rate for Max and Gaussian models trained with and without the non-negativity constraint.

5.6.5 Effects of Feature Reset

When training the model on MNIST, some less than optimal filters are learned when not resetting the feature maps. For example, in Figure 5.8 (c) many of these layer 1 filters are block-like such as the 3rd row, 2nd column. However this same feature in (a) improves if the feature maps are reset to 0 once half way through training. This single reset is enough to encourage the filters to specialize and improve. Similarly, the layer 2 pixel visualizations in (b) have much more variation due to the reset compared to (d) which did not have the reset. In particular, notice many blob-like features learned in (d) without reset such as the 2nd and 5th rows of the 1st column that improve in (b). These larger, more varied features learned with the reset help improve classification performance as shown in Table 5.4.

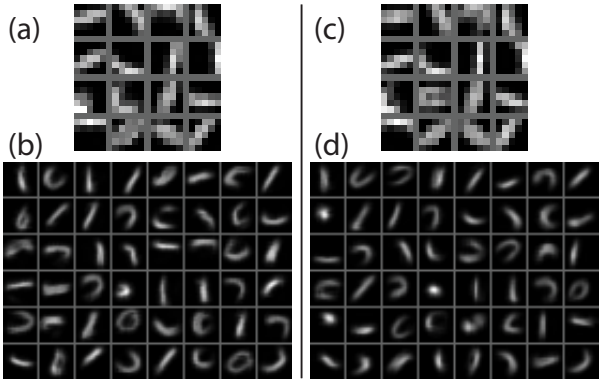

Figure 5.8: Qualitative difference in first layer filters with (left) and without (right) resetting of the feature maps.

Trained with No Reset	Trained with Reset
1.00%	0.84%

Table 5.4: MNIST error rates for 2 layer models trained with and without resetting the feature maps.

5.6.6 Effects of Hyper-Laplacian Sparsity

It has previously been shown that sparsity encourages learning of distinctive features, however it is not necessarily useful for classification [103]. We analyze this in the context of hyper-Laplacian sparsity applied to both training and inference. In this comparison we trained two models, one with a ℓ_1 prior on the feature maps and the other with a $\ell_{0.5}$ prior. Once trained, we took each model and ran inference with both ℓ_1 and $\ell_{0.5}$ priors. For reference the ℓ_0 sparsity for the training runs was 4.2 for the $\ell_{0.5}$ regularized training and 20.2 for the ℓ_1 regularized training with the same $\lambda^2 = 0.5$ setting. Since the amount of sparsity can also be controlled during inference by the λ^2 parameter, we plot in Figure 5.9 the classification performance for various λ^2 settings in these four model combinations.

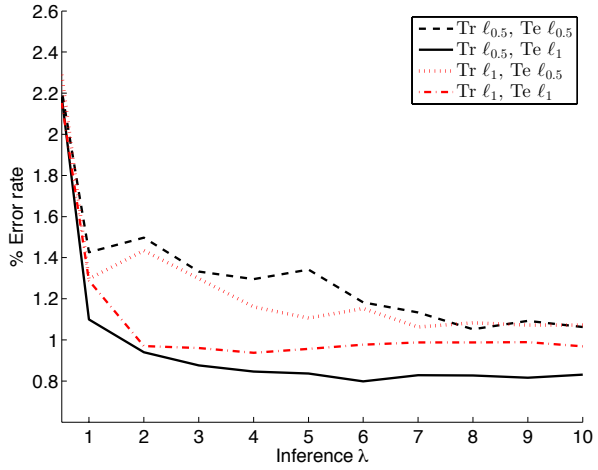

Figure 5.9: Error rates for ℓ_1 and $\ell_{0.5}$ priors used in training and inference.

Interestingly, utilizing the added sparsity during training enforced by the $\ell_{0.5}$ while using the more relaxed ℓ_1 prior for inference is the optimal combination for all λ settings. This suggests sparsity is useful during training to learn meaningful features, but is not as useful for inference at test time.

	Pre-training	Fine-tuning
Our Method	0.84%	–
CDBN (1+2 layers) [78]	0.82%	–
DBN (3 layers) [49] [50]	2.5%	1.18%
DBM (2 layers) [108]	–	0.95%

Table 5.5: MNIST errors rates for related generative models.

5.6.7 Comparison to Other Methods

We chose the MNIST dataset for its large number of results to compare to. Of these, deep learning methods typically fall into one of two categories, 1) those that are completely unsupervised and have a simple classifier on top, or 2) those that are fine-tunes discriminatively with labels. Our method falls into the first category as it is completely unsupervised during training, and only the linear SVM applied on top has access to the label information of the training set. We do not back propagate this information through the network, but this would be an interesting future direction to pursue. Table 5.5 shows our method is competitive with other deep generative models, even surpassing several which use discriminative fine tuning.

5.7 Discussion

In this chapter we introduced the concept of differentiable pooling for deep learning methods. Also, we demonstrated that joint training a deconvolutional network with differential pooling improves performance, positivity encourages the model to learn better representations, and that there is an optimal amount of sparsity to be used during training and inference. Finally, we introduced a simple resetting scheme to avoid local minimum and learn better features. We believe many of the approaches and findings in this work are applicable not only to deconvolutional networks but also to sparse coding and other deep learning methods in general.

Chapter 6

Stochastic Pooling for Regularization of Deep Convolutional Neural Networks

6.1 Transition

The previous three chapters introduced the deconvolutional network and several novel variations of them. These models were shown to learn interesting low, mid, and high level features of objects. The unsupervised learning in deconvolutional networks found natural groupings at each layer of similar looking object features with no label information provided to the system. These groupings were useful for object classification on classic computer vision benchmarks using spatial pyramids and linear SVM classifiers. However, since there was no discriminative information provided during training, such as class labels, it proved difficult to outperform state of the art classification methods using deconvolutional networks directly.

We attempted to make deconvolutional networks perform better for large-scale object classification. A classification term added to the overall cost function along with the reconstruction term and sparsity could encourage the latent features p to group based on class. However, we found tuning the multiple coefficients which trade off the three terms was too difficult and made the model unstable. Additionally, to scale to large datasets with millions of images rather than the thousands of images used thus far is not necessarily possible with the current deconvolutional network framework. Multiple iterations for optimizing the sparse coding object is slow when operating on millions of images. Additionally, the memory requirements to store the top layer feature maps and all lower layer pooling parameters is infeasible. Approximations of the sparse coding iterations were attempted where only the previous number of nonzero elements were saved for each image (i.e. the ℓ_0 sparsity) and on the next epoch for that same image pooling parameters were set in a feedforward fashion just as the Gaussian pooling parameters were initialized in the previous chapter while the shrinkage threshold for the first ISTA iteration was set to restore the saved ℓ_0 sparsity. While this did eliminate the memory issues and seemed to speed up the sparsification process initially, the overall objective function required nearly the same number of iterations to reach comparable objective values.

It was this memory constraint and lack of improvement with a discriminative objective that led to investigate fully supervised training with convolutional networks. Additionally, at nearly the same time the impressive work of [67] on large convolutional networks applied to ImageNet came out, which gave us additional motivation to transition into exploring convolutional networks.

The operations of a convolutional network are simply the opposite of a deconvolution network, the forward prop in a convolutional network being the backprop for a deconvolutional network and vice versa. This allowed a simple transition to convolutional networks using our expertise from training deconvolutional networks so we could investigate how convolutional networks could be improved and understood. This investigation

makes up the remainder of our work on hierarchical image models.

6.2 Introduction

We begin our work on convolutional networks by determining how the models can effectively scale to many parameters. Neural network models, including convolutional neural networks, are prone to over-fitting due to their high capacity. A range of regularization techniques are used to prevent this, such as weight decay, weight tying and the augmentation of the training set with transformed copies [89]. These allow the training of larger capacity models than would otherwise be possible, which yield superior test performance compared to smaller un-regularized models.

Dropout, recently proposed by Hinton *et al.* [44], is another regularization approach that stochastically sets half the activations within a layer to zero for each training sample during training. It has been shown to deliver significant gains in performance across a wide range of problems, although the reasons for its efficacy are not yet fully understood.

A drawback to dropout is that it does not seem to have the same benefits for convolutional layers, which are common in many networks designed for vision tasks. A generalization of dropout called DropConnect [136] could potentially be applied to convolutional layers though this would require a carefully crafted implementation. In this work, we propose a novel type of regularization for convolutional layers that enables the training of larger models without over-fitting, and produces superior performance on recognition tasks.

The key idea is to make the pooling that occurs in each convolutional layer a stochastic process. Conventional forms of pooling such as average and max are deterministic, the latter selecting the largest activation in each pooling region. In our stochastic pooling, the selected activation is drawn from a multinomial distribution formed by the activations within the pooling region.

An alternate view of stochastic pooling is that it is equivalent to standard max pooling

but with many copies of an input image, each having small local deformations. This is similar to explicit elastic deformations of the input images [117], which delivers excellent MNIST performance. Other types of data augmentation, such as flipping and cropping differ in that they are global image transformations. Furthermore, using stochastic pooling in a multi-layer model gives an exponential number of deformations since the selections in higher layers are independent of those below.

6.3 Review of Convolutional Networks

Our stochastic pooling scheme is designed for use in a standard convolutional neural network architecture. We first review this model, along with conventional pooling schemes, before introducing our novel stochastic pooling approach.

A classical convolutional network is composed of alternating layers of convolution and pooling (i.e. subsampling). The aim of the first convolutional layer is to extract patterns found within local regions of the input images that are common throughout the dataset. This is done by convolving a template or filter over the input image pixels, computing the inner product of the template at every location in the image and outputting this as a feature maps z, one for each filter f in the layer. This output is a measure of how well the template matches each portion of the image. A non-linear function $\sigma()$ is then applied element-wise to each feature map z: $h = \sigma(z)$. The resulting activations h are then passed to the pooling layer. This aggregates the information within a set of small local regions, $\mathcal{G}$, producing a pooled feature map p (of smaller size) as output. Denoting the aggregation function as $P()$, for each feature map z we have:

$$p_j = P(\sigma(z_i)) \quad \forall i \in \mathcal{G}_j \tag{6.1}$$

where $\mathcal{G}_j$ is pooling region j in feature map z and i is the index of each element within it.

The motivation behind pooling is that the activations in the pooled map p are less sensitive to the precise locations of structures within the image than the original feature map z. In a multi-layer model, the convolutional layers, which take the pooled maps as input, can thus extract features that are increasingly invariant to local transformations of the input image. This is important for classification tasks, since these transformations obfuscate the object identity.

A range of functions can be used for $\sigma()$, with $tanh()$ and logistic functions being popular choices. In this work we use a rectified linear function $\sigma(z) = max(0, z)$ as the non-linearity. In general, this has been shown [90] to have significant benefits over $tanh()$ or logistic functions (also see the evidence in Chapter 10 of this thesis). However, it is especially suited to our pooling mechanism since: (i) our formulation involves the non-negativity of elements in the pooling regions and (ii) the clipping of negative responses introduces zeros into the pooling regions, ensuring that the stochastic sampling is selecting from a few specific locations (those with strong responses), rather than all possible locations in the region.

There are two conventional choices for P: average and max. The former takes the arithmetic mean of the elements in each pooling region:

$$p_j = \frac{1}{|\mathcal{G}_j|} \sum_{i \in \mathcal{G}_j} h_i \tag{6.2}$$

while the max operation selects the largest element:

$$p_j = \max_{i \in \mathcal{G}_j} h_i \tag{6.3}$$

Both types of pooling have drawbacks when training deep convolutional networks. In average pooling, all elements in a pooling region are considered, even if many have low magnitude. When combined with linear rectification non-linearities, this has the effect of

down-weighting strong activations since many zero elements are included in the average. Even worse, with $tanh()$ non-linearities, strong positive and negative activations can cancel each other out, leading to small pooled responses.

While max pooling does not suffer from these drawbacks, we find it easily overfits the training set in practice, making it hard to generalize well to test examples. Our proposed pooling scheme has the advantages of max pooling but its stochastic nature helps prevent over-fitting.

6.4 Stochastic Pooling

In stochastic pooling, we select the pooled map response by sampling from a multinomial distribution formed from the activations of each pooling region. More precisely, we first compute the probabilities $P()$ for each region j by normalizing the activations within the region:

$$P(loc = i) = \frac{h_i}{\sum_{n \in \mathcal{G}_j} h_n} \tag{6.4}$$

We then sample from the multinomial distribution based on $P(loc)$ to pick a location loc within the region. The pooled activation is then simply h_{loc}:

$$p_j = h_{loc} \text{ where } loc \sim P(P(loc = 1), \ldots, P(loc = |\mathcal{G}_j|)) \tag{6.5}$$

The procedure is illustrated in Figure 6.1. The samples for each pooling region in each layer for each training example are drawn independently to one another. When back-propagating through the network this same selected location loc is used to direct the gradient back through the pooling region, analogous to back-propagation with max pooling.

Max pooling only captures the strongest activation of the filter template with the input

for each region. However, there may be additional activations in the same pooling region that should be taken into account when passing information up the network and stochastic pooling ensures that these non-maximal activations will also be utilized.

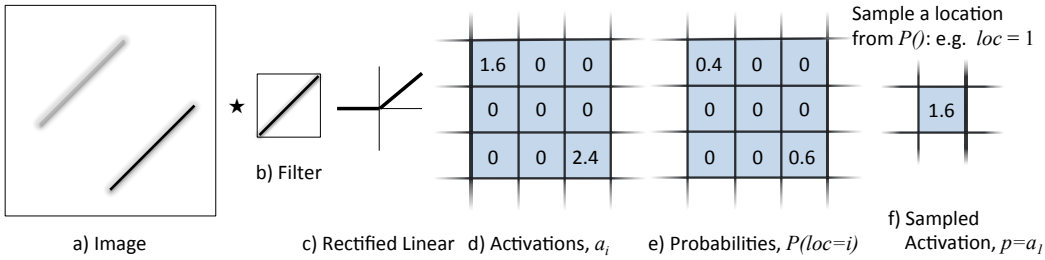

Figure 6.1: Toy example illustrating stochastic pooling. a) Input image. b) Convolutional filter. c) Rectified linear function. d) Resulting activations within a given pooling region. e) Probabilities based on the activations. f) Sampled activation. Note that the selected element for the pooling region may not be the largest element. Stochastic pooling can thus represent multi-modal distributions of activations within a region.

6.4.1 Probabilistic Weighting at Test Time

Using stochastic pooling at test time introduces noise into the network's predictions which we found to degrade performance (see Section 6.5.7). Instead, we use a probabilistic form of averaging. In this, the activations in each region are weighted by the probability $P(loc = i)$ (see Eq. 6.4) and summed:

$$p_j = \sum_{i \in \mathcal{G}_j} P(loc = i) h_i \tag{6.6}$$

This differs from standard average pooling because each element has a potentially different weighting and the denominator is the sum of activations $\sum_{i \in \mathcal{G}_j} h_i$, rather than the pooling region size $|\mathcal{G}_j|$. In practice, using conventional average (or sum) pooling results in a huge performance drop (see Section 6.5.7).

Our probabilistic weighting can be viewed as a form of model averaging in which each setting of the locations *loc* in the pooling regions defines a new model. At training time,

sampling to get new locations produces a new model since the connection structure throughout the network is modified. At test time, using the probabilities instead of sampling, we effectively get an estimate of averaging over all of these possible models without having to instantiate them. Given a network architecture with $N_{\mathcal{G}}$ different pooling regions, each of size $N_{i \in \mathcal{G}}$, the number of possible models is ${N_{\mathcal{G}}}^{N_{i \in \mathcal{G}}}$ where $N_{\mathcal{G}}$ can be in the 10^4-10^6 range and $N_{i \in \mathcal{G}}$ is typically 4,9, or 16 for example (corresponding to 2×2, 3×3 or 4×4 pooling regions). This is a significantly larger number than the model averaging that occurs in dropout [44], where $N_{i \in \mathcal{G}} = 2$ always (since an activation is either present or not). In Section 6.5.7 we confirm that using this probability weighting achieves similar performance compared to using a large number of model instantiations, while requiring only one pass through the network.

Using the probabilities for sampling at training time and for weighting the activations at test time leads to what was state-of-the-art performance on many common benchmarks at the time of publication.

6.5 Experiments

6.5.1 Overview

We compare our method to average and max pooling on a variety of image classification tasks. In all experiments we use mini-batch gradient descent with momentum to optimize the cross entropy between our network's prediction of the class and the ground truth labels. For a given parameter θ at time t the weight updates added to the parameters, $\Delta\theta_t$ are $\Delta\theta_t = 0.9\Delta\theta_{t-1} - \eta\nabla\theta_t$ where $\nabla\theta_t$ is the gradient of the cost function with respect to that parameter at time t averaged over the batch and η is a learning rate set by hand.

All experiments were conducted using an extremely efficient C++ GPU convolution library [66] wrapped in MATLAB using GPUmat [132], which allowed for rapid develop-

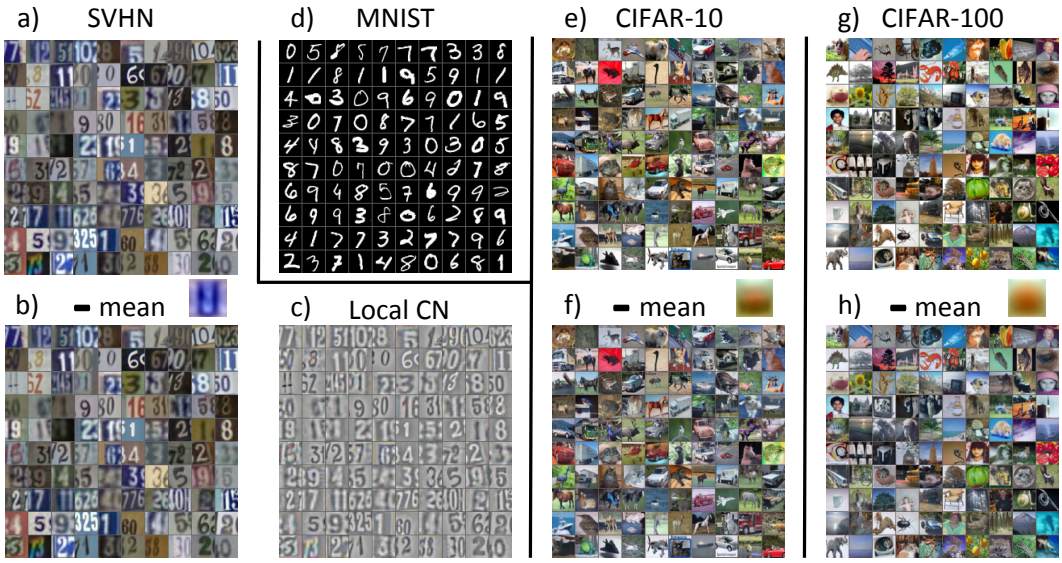

Figure 6.2: A selection of images from each of the datasets we evaluated. The top row shows the raw images while the bottom row are the preprocessed versions of the images we used for training. The CIFAR datasets (f,h) show slight changes by subtracting the per pixel mean, whereas SVHN (b) is almost indistinguishable from the original images. This prompted the use of local contrast normalization (c) to normalize the extreme brightness variations and color changes for SVHN.

ment and experimentation. We begin with the same network layout as in Hinton *et al.* 's dropout work [44], which has 3 convolutional layers with 5x5 filters and 64 feature maps per layer with rectified linear units as their outputs. We use this same model and train for 280 epochs in all experiments aside from one additional model in Section 6.5.5 that has 128 feature maps in layer 3 and is trained for 500 epochs. Unless otherwise specified we use 3×3 pooling with stride 2 (i.e. neighboring pooling regions overlap by 1 element along the borders) for each of the 3 pooling layers. Additionally, after each pooling layer there is a response normalization layer (as in [44]), which normalizes the pooling outputs at each location over a subset of neighboring feature maps. This typically helps training by suppressing extremely large outputs allowed by the rectified linear units as well as helps neighboring features communicate. Finally, we use a single fully-connected layer with soft-max outputs to produce the network's class predictions. We applied this model to four different datasets: MNIST, CIFAR-10, CIFAR-100 and Street View House Numbers (SVHN), see Figure 6.2 for examples images.

6.5.2 CIFAR-10

We begin our experiments with the CIFAR-10 dataset where convolutional networks and methods such as dropout are known to work well [44,65]. This dataset is composed of 10 classes of natural images with 50,000 training examples in total, 5,000 per class. Each image is an RGB image of size 32x32 taken from the tiny images dataset and labeled by hand. For this dataset we scale to [0,1] and follow Hinton *et al.* 's [44] approach of subtracting the per-pixel mean computed over the dataset from each image as shown in Figure 6.2(f).

Cross-validating with a set of 5,000 CIFAR-10 training images, we found a good value for the learning rate η to be 10^{-2} for convolutional layers and 1 for the final softmax output layer. These rates were annealed linearly throughout training to $1/100th$ of their original values. Additionally, we found a small weight decay of 0.001 to improve performance and was applied to all layers. These hyper-parameter settings found through cross-validation were used for all other datasets in our experiments.

Using the same network architecture described above, we trained three models using average, max and stochastic pooling respectively and compare their performance. Figure 6.3 shows the progression of train and test errors over 280 training epochs. Stochastic pooling avoids over-fitting, unlike average and max pooling, and produces less test errors. Table 6.1 compares the test performance of the three pooling approaches to the current state-of-the-art result on CIFAR-10 which uses no data augmentation but adds dropout on an additional locally connected layer [44]. Stochastic pooling surpasses this result by 0.47% using the same architecture but without requiring the locally connected layer. This is near the state of the art results found by tuning hyperparameters with Gaussian Processes [120] or the even more recent work on Maxout networks by Goodfellow *et al.* [41].

[1] Weight decay prevented training errors from reaching 0 with average and stochastic pooling methods and required the high number of epochs for training. All methods performed slightly better with weight decay.

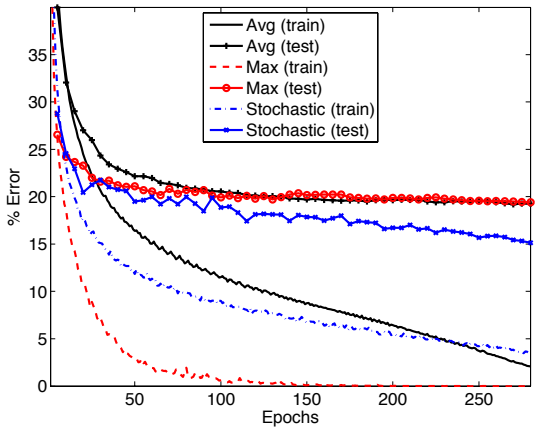

Figure 6.3: CIFAR-10 train and test error rates throughout training for average, max, and stochastic pooling. Max and average pooling test errors plateau as those methods overfit. With stochastic pooling, training error remains higher while test errors continue to decrease.

	Train Error %	Test Error %
3-layer Conv. Net [44]	–	16.6
3-layer Conv. Net + 1 Locally Conn. layer with dropout [44]	–	15.6
3-layer Conv. Net + GP [120]	–	14.93
Maxout Networks [41]	–	**12.93**
Avg Pooling	1.92	19.24
Max Pooling	0.0	19.40
Stochastic Pooling	3.40	**15.13**

Table 6.1: CIFAR-10 Classification performance for various pooling methods in our model compared to the state-of-the-art performance [44] with and without dropout, with Gaussian process hyperparameter tuning, or using maxout networks.

To determine the effect of the pooling region size on the behavior of the system with stochastic pooling, we compare the CIFAR-10 train and test set performance for 5x5, 4x4, 3x3, and 2x2 pooling sizes throughout the network in Figure 6.4. The optimal size appears to be 3x3, with smaller regions over-fitting and larger regions possibly being too noisy during training. At all sizes the stochastic pooling outperforms both max and average pooling.

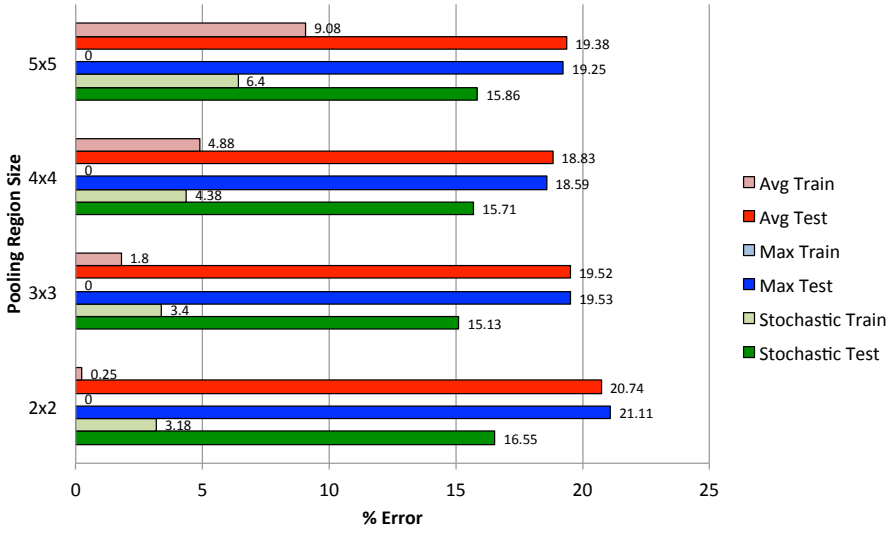

Figure 6.4: CIFAR-10 train and test error rates for various pooling region sizes with each method.

6.5.3 MNIST

The MNIST digit classification task is composed of 28x28 images of the 10 handwritten digits [75]. There are 60,000 training images with 10,000 test images in this benchmark. The images are scaled to [0,1] and we do not perform any other pre-processing.

During training, the error using both stochastic pooling and max pooling dropped quickly, but the latter completely overfit the training data. Weight decay prevented average pooling from over-fitting, but had an inferior performance to the other two methods. Table 6.2 compares the three pooling approaches to state-of-the-art methods on MNIST, which also utilize convolutional networks. Stochastic pooling outperforms all other methods that do not use data augmentation methods such as jittering or elastic distortions [72]. The current state-of-the-art single model approach by Ciresan *et al.* [23] uses elastic distortions to augment the original training set. As stochastic pooling is a different type of regularization, it could be combined with data augmentation to further improve performance.

	Train Error %	Test Error %
2-layer Conv. Net + 2-layer Classifier [56]	–	0.53
6-layer Conv. Net + 2-layer Classifier + elastic distortions [23]	–	0.35
Maxout Networks [41]	–	**0.45**
Avg Pooling	0.57	0.83
Max Pooling	0.04	0.55
Stochastic Pooling	0.33	**0.47**

Table 6.2: MNIST Classification performance for various pooling methods. Rows 1 & 2 show the current state-of-the-art approaches. More recently maxout networks have been shown to surpass our results [41].

6.5.4 CIFAR-100

The CIFAR-100 dataset is another subset of the tiny images dataset, but with 100 classes [65]. There are 50,000 training examples in total (500 per class) and 10,000 test examples. As with the CIFAR-10, we scale to [0,1] and subtract the per-pixel mean from each image as shown in Figure 6.2(h). Due to the limited number of training examples per class, typical pooling methods used in convolutional networks do not perform well, as shown in Table 6.3. Stochastic pooling outperforms these methods by preventing over-fitting and surpasses what we was the state-of-the-art method by 2.66% and nearing the more recent Maxout Network results [41].

	Train Error %	Test Error %
Receptive Field Learning [57]	–	45.17
Maxout Networks [41]	–	**38.57**
Avg Pooling	11.20	47.77
Max Pooling	0.17	50.90
Stochastic Pooling	21.22	**42.51**

Table 6.3: CIFAR-100 Classification performance for various pooling methods compared to the previous state-of-the-art method based on receptive field learning. More recently maxout networks have been shown to improve upon our results [41].

6.5.5 Street View House Numbers

The Street View House Numbers (SVHN) dataset is composed of 604,388 images (using both the difficult training set and simpler extra set) and 26,032 test images [91]. The goal of this task is to classify the digit in the center of each cropped 32x32 color image.

This is a difficult real world problem since multiple digits may be visible within each image. The practical application of this is to classify house numbers throughout Google's street view database of images.

We found that subtracting the per-pixel mean from each image did not really modify the statistics of the images (see Figure 6.2(b)) and left large variations of brightness and color that could make classification more difficult. Instead, we utilized local contrast normalization (as in [113]) on each of the three RGB channels to pre-process the images Figure 6.2(c). This normalized the brightness and color variations and helped training proceed quickly on this relatively large dataset.

Despite having significant amounts of training data, a large convolutional network can still overfit. For this dataset, we train an additional model for 500 epochs with 64, 64 and 128 feature maps in layers 1, 2 and 3 respectively. Our stochastic pooling helps to prevent overfitting even in this large model (denoted 64-64-128 in Table 6.4), despite training for a long time. The existing state-of-the-art on this dataset is the multi-stage convolutional network of Sermanet *et al.* [113], but stochastic pooling beats this by 2.10% (relative gain of 43%) similar to the recent Maxout results [41].

	Train Error %	Test Error %
Multi-Stage Conv. Net + 2-layer Classifier [113]	–	5.03
Multi-Stage Conv. Net + 2-layer Classifer + padding [113]	–	4.90
Maxout Networks [41]	–	**2.72**
64-64-64 Avg Pooling	1.83	3.98
64-64-64 Max Pooling	0.38	3.65
64-64-64 Stochastic Pooling	1.72	3.13
64-64-128 Avg Pooling	1.65	3.72
64-64-128 Max Pooling	0.13	3.81
64-64-128 Stochastic Pooling	1.41	**2.80**

Table 6.4: SVHN Classification performance for various pooling methods in our model with 64 or 128 layer 3 feature maps compared to state-of-the-art results with and without data augmentation. Maxout network recently edged out stochastic pooling in this benchmark as well [41].

6.5.6 Reduced Training Set Size

To further illustrate the ability of stochastic pooling to prevent over-fitting, we reduced the training set size on MINST and CIFAR-10 datasets. Figure 6.5 shows test performance when training on a random selection of only 1000, 2000, 3000, 5000, 10000, half, or the full training set. In most cases, stochastic pooling overfits less than the other pooling approaches.

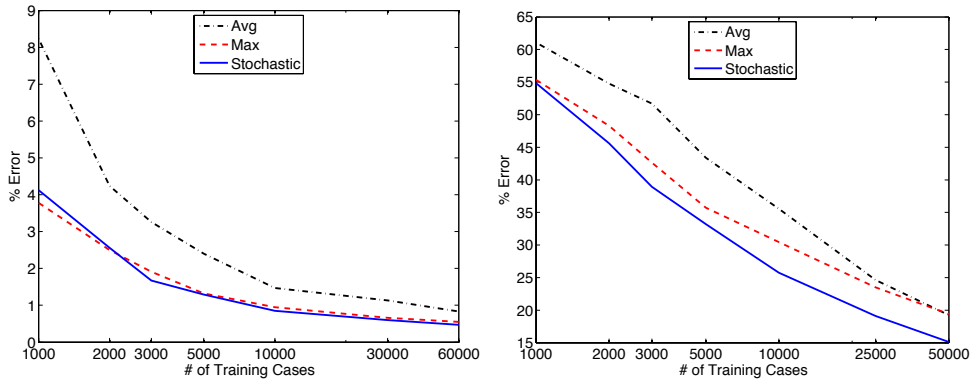

Figure 6.5: Test error when training with reduced dataset sizes (on log scale) for MNIST (left) and CIFAR-10 (right). Stochastic pooling generally overfits the least.

6.5.7 Importance of Model Averaging

To analyze the importance of stochastic sampling at training time and probability weighting at test time, we use different methods of pooling when training and testing on CIFAR-10 (see Table 6.5). Choosing the locations stochastically at test time degrades performance slightly as could be expected, however it still outperforms models where max or average pooling are used at test time. To confirm that probability weighting is a valid approximation to averaging many models, we draw N samples of the pooling locations throughout the network and average the output probabilities from those N models (denoted Stochastic-N in Table 6.5). As N increases, the results approach the probability weighting method, but have the obvious downside of an N-fold increase in computations.

Using a model trained with max or average pooling and using stochastic pooling at test time performs poorly. This suggests that training with stochastic pooling, which incorporates non-maximal elements and sampling noise, makes the model more robust at test time. Furthermore, if these non-maximal elements are not utilized correctly or the scale produced by the pooling function is not correct, such as if average pooling is used at test time, a drastic performance hit is seen.

When using probability weighting during training, the network easily over-fits and performs sub-optimally at test time using any of the pooling methods. However, the benefits of probability weighting at test time are seen when the model has specifically been trained to utilize it through either probability weighting or stochastic pooling at training time.

Train Method	Test Method	Train Error %	Test Error %
Stochastic Pooling	Probability Weighting	3.20	**15.20**
Stochastic Pooling	Stochastic Pooling	3.20	17.49
Stochastic Pooling	Stochastic-10 Pooling	3.20	15.51
Stochastic Pooling	Stochastic-100 Pooling	3.20	**15.12**
Stochastic Pooling	Max Pooling	3.20	17.66
Stochastic Pooling	Avg Pooling	3.20	53.50
Probability Weighting	Probability Weighting	0.0	19.40
Probability Weighting	Stochastic Pooling	0.0	24.00
Probability Weighting	Max Pooling	0.0	22.45
Probability Weighting	Avg Pooling	0.0	58.97
Max Pooling	Max Pooling	0.0	19.40
Max Pooling	Stochastic Pooling	0.0	32.75
Max Pooling	Probability Weighting	0.0	30.00
Avg Pooling	Avg Pooling	1.92	19.24
Avg Pooling	Stochastic Pooling	1.92	44.25
Avg Pooling	Probability Weighting	1.92	40.09

Table 6.5: CIFAR-10 Classification performance for various train and test combinations of pooling methods. The best performance is obtained by using stochastic pooling when training (to prevent over-fitting), while using the probability weighting at test time.

6.5.8 Visualizations

Some insight into the mechanism of stochastic pooling can be gained by using a deconvolutional network to provide a novel visualization of our trained convolutional network. The deconvolutional network has the same components (pooling, filtering) as a convo-

lutional network but are inverted to act as a top-down decoder that maps the top-layer feature maps back to the input pixels. The unpooling operation uses the stochastically chosen locations selected during the forward pass. The deconvolution network filters (now applied to the feature maps, rather than the input) are the transpose of the feed-forward filters, as in an auto-encoder with tied encoder/decoder weights. We repeat this top-down process until the input pixel level is reached, producing the visualizations in Figure 6.6. With max pooling, many of the input image edges are present, but average pooling produces a reconstruction with no discernible structure. Figure 6.6(a) shows 16 examples of pixel-space reconstructions for different location samples throughout the network. The reconstructions are similar to the max pooling case, but as the pooling locations change they result in small local deformations of the visualized image.

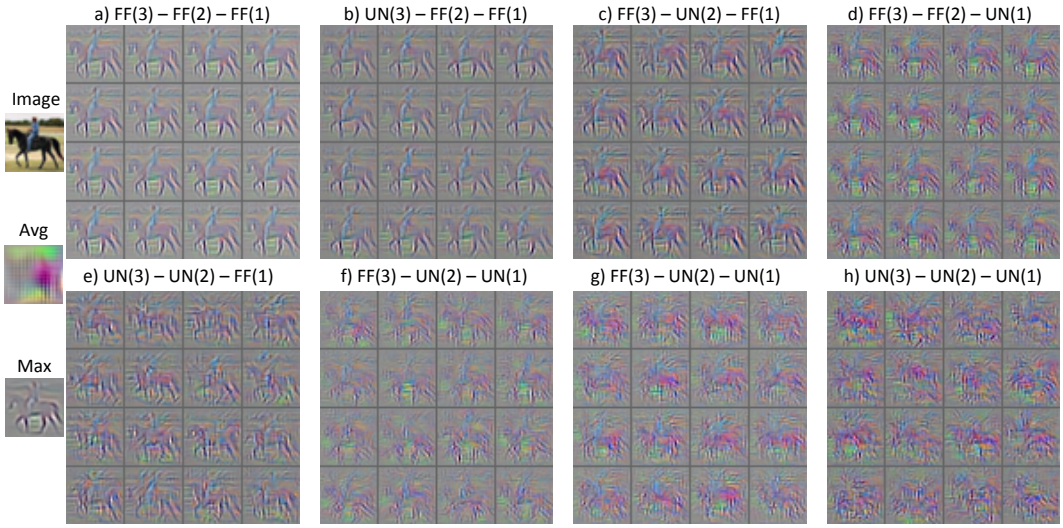

Figure 6.6: Top down visualizations from the third layer feature map activations for the horse image (far left). Max and average pooling visualizations are also shown on the left. (a)–(h): Each image in a 4x4 block is one instantiation of the pooling locations using stochastic pooling. For sampling the locations, each layer (indicated in parenthesis) can either use: (i) the multinomial distribution over a pooling region derived from the feed-forward (FF) activations as in Eq. 6.4, or (ii) a uniform (UN) distribution. We can see that the feed-forward probabilities encode much of the structure in the image, as almost all of it is lost when uniform sampling is used, especially in the lower layers.

Despite the stochastic nature of the model, the multinomial distributions effectively capture the regularities of the data. To demonstrate this, we compare the outputs produced

by a deconvolutional network when sampling using the feedforward (FF) probabilities versus sampling from uniform (UN) distributions. In contrast to Figure 6.6(a) which uses only feedforward probabilities, Figure 6.6(b-h) replace one or more of the pooling layers' distributions with uniform distributions. The feed forward probabilities encode significant structural information, especially in the lower layers of the model. Additional visualizations and videos of the sampling process are provided as supplementary material at `www.matthewzeiler.com/pubs/iclr2013/`.

6.6 Discussion

We propose a simple and effective stochastic pooling strategy that can be combined with any other forms of regularization such as weight decay, dropout, data augmentation, etc. to prevent over-fitting when training deep convolutional networks. The method is intuitive, selecting from information the network is already providing, as opposed to methods such as dropout which throw information away. We show what was state-of-the-art performance at the time of publication on numerous datasets, when comparing to other approaches that do not employ data augmentation. Furthermore, our method has negligible computational overhead and no hyper-parameters to tune, thus can be swapped into to any existing convolutional network architecture.

Chapter 7

Visualizing and Understanding Convolutional Neural Networks

7.1 Introduction

One solution to the problem of overfitting, namely stochastic pooling, was introduced in the previous chapter. This was shown to enable state of the art performance on a variety of datasets. However, the datasets we used are relatively small in terms of pixel dimensions and number of images, too small to fully utilize the representational power of a large convolutional network. Such large convolutional neural network models have recently demonstrated impressive classification performance on the ImageNet benchmark (Krizhevsky *et al.* [67]). Despite this, there is no clear understanding of why they perform so well, or how they might be improved. In this chapter we address both issues.

We introduce a novel visualization technique based on deconvolutional networks that gives insight into the function of intermediate feature layers and the operation of the classifier. We also systematically analyze the performance contribution of individual layers to determine their importance. This enables us to find model architectures that outperform Krizhevsky *et al.* on the ImageNet classification benchmark. We show our

ImageNet model generalizes well to other datasets: when the softmax classifier is re-trained, it convincingly beats the current state-of-the-art results on Caltech-101 and Caltech-256 datasets.

Since their introduction by LeCun *et al.* [73] in the early 1990's, convolutional neural networks have demonstrated excellent performance at tasks such as hand-written digit classification and face detection. Recently, several papers have shown that they can also deliver outstanding performance on more challenging visual classification tasks. Ciresan *et al.* [22] demonstrate state-of-the-art performance on NORB and CIFAR-10 datasets. Most notably, Krizhevsky *et al.* [67] show record beating performance on the ImageNet 2012 classification benchmark, with their convnet model achieving an error rate of 16.4%, compared to the 2nd place result of 26.1%. Several factors are responsible for this renewed interest in convnet models: (i) the availability of much larger training sets, with millions of labeled examples; (ii) powerful GPU implementations, making the training of very large models practical and (iii) better model regularization strategies, such as Dropout [44].

Despite this encouraging progress, there is still little insight into the internal operation and behavior of these complex models, or how they achieve such good performances. From a scientific standpoint, this is deeply unsatisfactory. Without clear understanding of how and why they work, the development of better models is reduced to trial-and-error. In this work we introduce a visualization technique that reveals the input stimuli that excite individual feature maps at any layer in the model. It also allows us to observe the evolution of features during training and to diagnose potential problems with the model. The visualization technique we propose uses a multi-layered deconvolutional network to project the feature activations back to the input pixel space. We also perform a sensitivity analysis of the classifier output by occluding portions of the input image, revealing which parts of the scene are important for classification.

Using these tools, we start with the architecture of Krizhevsky *et al.* [67] and explore

different architectures, discovering ones that outperform their results on ImageNet. We then explore the generalization ability of the model to other datasets, just retraining the softmax classifier on top. As such, this is a form of supervised pre-training, which contrasts with the unsupervised pre-training methods popularized by Hinton *et al.* [49] and others [7, 133].

Visualizing features to gain intuition about the network is common practice, but mostly limited to the 1st layer where projections to pixel space are possible. In higher layers this is not the case, and there are limited methods for interpreting activity. Erhan *et al.* [31] find the optimal stimulus for each unit by performing gradient descent in image space to maximize the unit's activation. This requires a careful initialization and does not give any information about the unit's invariances. Motivated by the latter's short-coming, Le *et al.* [71] (extending an idea by Berkes and Wiskott [9]) show how the Hessian of a given unit may be computed numerically around the optimal response, giving some insight into invariances. The problem is that for higher layers, the invariances are extremely complex so are poorly captured by a simple quadratic approximation. Our approach, by contrast, provides a non-parametric view of invariance, showing which patterns from the training set activate the feature map.

7.2 Approach

Before introducing our novel visualization technique, we recap the convnet models to which they will be applied.

We use standard fully supervised convnet models throughout the work, as defined by LeCun *et al.* [73] and Krizhevsky *et al.* [67]. These models map a color 2D input image x_i, via a series of layers, to a probability vector $\hat{y}_i$ over the C different classes. Each layer consists of (i) convolution of the previous layer output (or, in the case of the 1st layer, the input image) with a set of learned filters; (ii) passing the responses through a rectified linear function ($relu(x) = \max(x, 0)$); (iii) [optionally] max pooling over

local neighborhoods and (iv) [optionally] a local contrast operation that normalizes the responses across feature maps. For more details of these operations, see Krizhevsky *et al.* [67] and Jarrett *et al.* [56]. The top few layers of the network are conventional fully-connected networks and the final layer is a softmax classifier. Figure 7.2 shows the model used in many of our experiments.

We train these models using a large set of N_x labeled images $\{x, y\}$, where label y_i is a discrete variable indicating the true class. A cross-entropy loss function, suitable for image classification, is used to compare $\hat{y}_i$ and y_i. The parameters of the network (filters in the convolutional layers, weight matrices in the fully-connected layers and biases) are trained by back-propagating the derivative of the loss with respect to the parameters throughout the network, and updating the parameters via stochastic gradient descent. Full details of training are given in Section 7.3.1.

7.2.1 Visualization with a Deconvolutional Network

Understanding the operation of a convnet requires interpreting the feature activity in intermediate layers. We present a novel way to *map these activities back to the input pixel space*, showing what input pattern originally caused a given activation in the feature maps. We perform this mapping with a deconvolutional network (deconvnet). A deconvnet can be thought of as a convnet model that uses the same components (filtering, pooling) but in reverse, so instead of mapping pixels to features does the opposite. In previous chapters, deconvnets were proposed as a way of performing unsupervised learning. Here, they are not used in any learning capacity, just as a probe of the convnet.

To examine a convnet, a deconvnet is attached to each of its layers, as illustrated in Figure 7.1(top), providing a continuous path back to image pixels. To start, an input image is presented to the convnet and features computed throughout the layers. To examine a given convnet activation, we set all other activations in the layer to zero and

pass the feature maps as input to the attached deconvnet layer. Then we successively (i) unpool, (ii) rectify and (iii) filter to reconstruct the activity in the layer beneath that gave rise to the chosen activation. This is then repeated until input pixel space is reached.

Unpooling: In the convnet, the max pooling operation is non-invertible, however we can obtain an approximate inverse by recording the locations of the maxima within each pooling region in a set of *switch* variables. In the deconvnet, the unpooling operation uses these switches to place the reconstructions from the layer above into appropriate locations, preserving the structure of the stimulus. See Figure 7.1(bottom) for a visualization of the procedure.

Rectification: The convnet uses *relu* non-linearities, which rectify the feature maps. To obtain valid reconstructions, we pass the reconstructed signal through a *relu* non-linearity.

Filtering: The convnet uses learned filters to convolve the feature maps from the previous layer. To invert this, the deconvnet uses transposed versions of the same filters, but applied to the rectified maps, not the output of the layer beneath. In practice this means flipping each filter vertically and horizontally.

Projecting down from higher layers uses the switch settings generated by the max pooling in the convnet on the way up. As these switch settings are peculiar to a given input image, the reconstruction obtained from a single activation is thus a small piece of the original input image, with structures weighted according to their contribution toward to the feature activation. Since the model is trained discriminatively, they implicitly show which parts of the input image are discriminative.

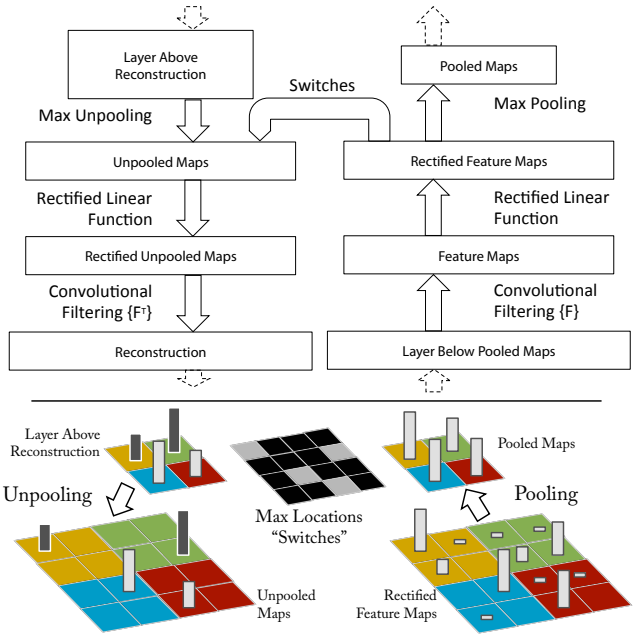

Figure 7.1: Top: A deconvnet layer (left) attached to a convnet layer (right). The deconvnet will reconstruct an approximate version of the convnet features from the layer beneath. Bottom: An illustration of the unpooling operation in the deconvnet, using *switches* which record the location of the local max in each pooling region (colored zones) during pooling in the convnet.

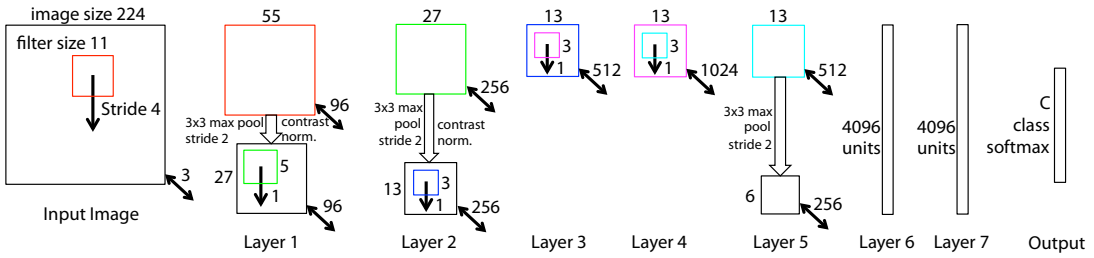

Figure 7.2: Architecture of our 8 layer convnet model. A 224 by 224 crop of an image (with 3 color planes) is presented as the input. This is convolved with 96 different 1st layer filters (red), each of size 11 by 11, using a stride of 4 in both x and y. The resulting feature maps are then: (i) passed through a rectified linear function (not shown), (ii) pooled (max within 3x3 regions, using stride 2) and (iii) contrast normalized across feature maps to give 96 different 27 by 27 element feature maps. Similar operations are repeated in layers 2,3,4,5. The last two layers are fully connected, taking features from the top convolutional layer as input in vector form ($6 \cdot 6 \cdot 256 = 9216$ dimensions). The final layer is a C-way softmax function, C being the number of classes. All filters and feature maps are square in shape.

7.3 Experiments

We start by training a large convolutional network model on the ImageNet dataset, using the exact architecture specified in Krizhevsky *et al.* [67] and attempt to replicate their result on the validation set. The ImageNet dataset [28] consists of 1.3M/65k/100k training/validation/test examples, spread over 1000 categories. Details of the training procedure are given in Section 7.3.1 below. As shown in Table 7.2, we achieve error rate within 0.1% of their reported value on the ImageNet 2012 validation set.

We now explore a range of different model architectures in an attempt to understand the relative importance of each layer. In Table 7.1, we modify the size of (a) the convolutional layers, (b) the fully connected layers and (c) both sections of the model. Decreasing each part separately only results in a modest performance drop. This is surprising for the fully connected layers, given that they contain the majority of the model's parameters. However, decreasing both serverly affects performance, showing the importance of having a minimum depth to the model. Altering the number of units in the fully connected layers (2048 or 8192 vs 4096) makes little difference to performance. Increasing the size of the convolutional layers 3,4,5 to 512-1024-512 maps, from 384-384-256, does give a gain in performance, but the model starts to over-fit due to the big increase in number of parameters. The overfitting is more pronounced when increasing the size of both the convolutional and fully connected layers.

Error %	Train Top-1	Val Top-1	Val Top-5
Our replication of Krizhevsky *et al.* [67], 1 convnet	35.1	40.5	18.1
With removed layers 3,4	41.8	45.4	22.1
With removed layer 7	27.4	40.0	18.4
With removed layers 6,7	27.4	44.8	22.4
With removed Layers 3,4,6,7	71.1	71.3	50.1
With layers 6,7: 2048 units	40.3	41.7	18.8
With layers 6,7: 4096 units as per [67]	35.1	40.5	18.1
With layers 6,7: 8192 units	26.8	40.0	18.1
Our model	28.7	**38.3**	**16.4**
Our model, layers 6,7: 8192 units	21.4	**38.0**	**16.5**

Table 7.1: ImageNet 2012 classification error rates with various architectural changes to our ImageNet model.

Error %	Val Top-1	Val Top-5	Test Top-5
Krizhevsky *et al.* [67], 1 convnet	40.7	18.2	−−
Krizhevsky *et al.* [67], 5 convnets	38.1	16.4	16.4
Krizhevsky *et al.* [67], 1 convnets*	39.0	16.6	−−
Krizhevsky *et al.* [67], 7 convnets*	36.7	15.4	15.3
Our replication of Krizhevsky *et al.* [67], 1 convnet	40.5	18.1	−−
1 convnet as per Figure 7.2	38.3	16.4	16.5
5 convnets as per Figure 7.2	36.6	15.3	15.3

Table 7.2: ImageNet 2012 classification error rates. The $*$ indicates models that were trained on both ImageNet 2011 and 2012 training sets with an additional convolution layer.

The experiments in Table 7.1 show that by increasing the number of feature maps in the middle layers, the model of [67] may be improved upon. Figure 7.2 shows the best performing architecture, which has a dramatically larger layers 3,4 and 5. When evaluated on the Imagenet 2012 validation set, it significantly outperforms [67], beating their single model result by 1.8% (see Table 7.2). When we combine multiple models, we obtain a test error of **15.3**%, which matches the absolute best performance on this dataset, despite only using the much smaller 2012 training set. We note that this error is almost half that of the top non-convnet entry in the ImageNet 2012 classification challenge, which obtained 26.1% error.

7.3.1 Training Details

The models were trained on the ImageNet 2012 training set (1.3 million images, spread over 1000 different classes). Each RGB image was preprocessed by resizing the smallest dimension to 256, cropping the center 256x256 region, subtracting the per-pixel mean (across all images) and then using 10 different sub-crops of size 224x224 (corners + center with(out) horizontal flips). Stochastic gradient descent with a mini-batch size of 128 was used to update the parameters, starting with a learning rate of 10^{-2}, in conjunction with a momentum term of 0.9. Dropout [44] is used in the fully connected layers (6 and 7) with a rate of 0.5. We manually anneal the learning rate throughout training, decreasing it when the validation set error flatlines. All weights are randomly initialized 10^{-2} standard deviation and biases are set to 0.

As in [67], we produce multiple different crops and flips of each training example to boost training set size. We stopped training after 70 epochs, which took around 24 days on a single GTX580 GPU, using an implementation based on [67]. One further difference is that the sparse connections used in Krizhevsky's layers 3,4,5 (due to the model being split across 2 GPUs) are replaced with dense connections in our models.

7.4 Convnet Visualization

Using the model described in Section 7.3.1, we now use the deconvnet to visualize the feature activations on the ImageNet validation set.

7.4.1 Feature Evolution during Training

Figure 7.3 visualizes the progression during training of the strongest activation (across all training examples) within a given feature map projected back to pixel space. Sudden jumps in appearance result from a change in the image from which the strongest activation originates. Due to space constraints, only a randomly selected subset of feature maps are visualized and zooming is needed to see the details clearly. As expected, the first layer filters consist of Gabors and low-frequency color. The 2nd layer features are more complex, corresponding to conjunctions of edges and color patterns. The 3rd layer features show larger image parts. Within a given feature projection, significant variations in contrast can be seen, showing which parts of the image contribute most to the activation and thus are most discriminative, e.g. the lips and eyes on the persons face (Row 12). The visualization from the 4th and 5th layer show activations that respond to complex objects. Note that little of the scene background is reconstructed, since it is irrelevant to predicting the class.

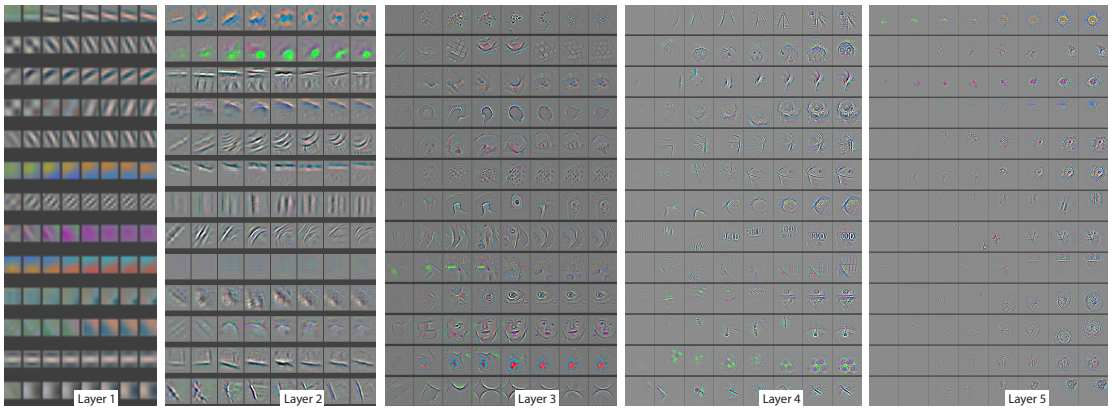

Figure 7.3: Evolution of model features through training. Each layer's features are displayed in a different block. Within each block, we show a randomly chosen subset of features at epochs [1,2,5,10,20,30,40,64]. The visualization shows the strongest activation (across all training examples) for a given feature map, projected down to pixel space using our deconvnet approach. Color contrast is artificially enhanced and the figure is best viewed in electronic form.

7.4.2 Feature Invariance

Figure 7.4 shows feature visualizations from our model once training is complete. However, instead of showing the single strongest activation for a given feature map, we show the top 9 activations. Projecting each separately down to pixel space reveals the different structures that excite a given feature map, hence showing its invariance to input deformations. Layer 2 responds to corners and other edge/color conjunctions. Layer 3 has more complex invariances, capturing similar textures (e.g. mesh patterns (Row 1, Col 1); grilles (R3,C5)). Layer 4 shows significant variation, but is more class-specific: faces of children (R4,C4); bear/koala tongue/nose (R3,C6). Layer 5 shows entire objects with significant pose variation, e.g. keyboards (R1,C11).

Figure 7.5 shows 5 sample images being translated, rotated and scaled by varying degrees while looking at the changes in the feature vectors from the top and bottom layers of the model, relative to the untransformed feature. Small transformations have a dramatic effect in the first layer of the model, but a lesser impact at the top feature layer, being quasi-linear for translation & scaling. The network output is stable to translations and scalings, but only slightly to rotation around the original upright orientation.

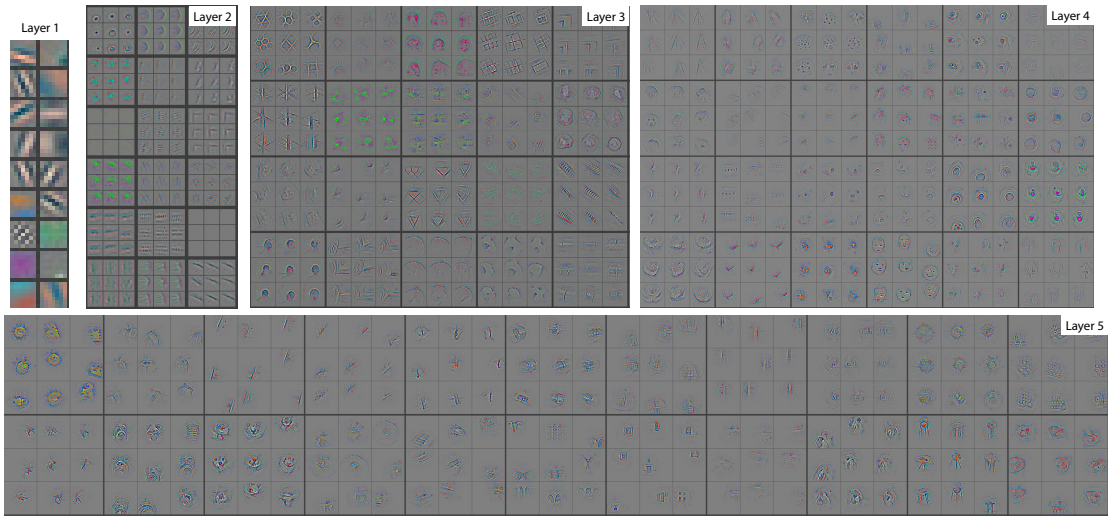

Figure 7.4: Visualization of features in a fully trained model. For layers 2-5 we show the top 9 activations in a random subset of feature maps across the validation data, projected down to pixel space using our deconvolutional network approach. Note: (i) the strong grouping within each feature map, (ii) greater invariance at higher layers and (iii) exaggeration of discriminative parts of the image, e.g. eyes and noses of the cats, dogs & similar animals (layer 5, 2nd row, cols 2 and 3). Best viewed in electronic form.

7.4.3 Occlusion Sensitivity

With image classification approaches, a natural question is if the model is truly classifying the object alone, or if it is using the surrounding context. Figure 7.6 attempts to answer this question by systematically occluding different portions of the input image with a grey square, and monitoring the output of the classifier. The examples clearly show the model is localizing the objects within the scene, as the probability of the correct class drops significantly when the object is occluded. Figure 7.6 also shows visualizations from the strongest feature map of the top convolution layer, in addition to activity in this map as a function of occluder position. When the occluder covers the image region that appears in the visualization, we see a strong drop in activity in the feature map. This shows that the visualization genuinely corresponds to the image structure that stimulates that feature map, hence validating the other visualizations in Figure 7.3 and Figure 7.4.

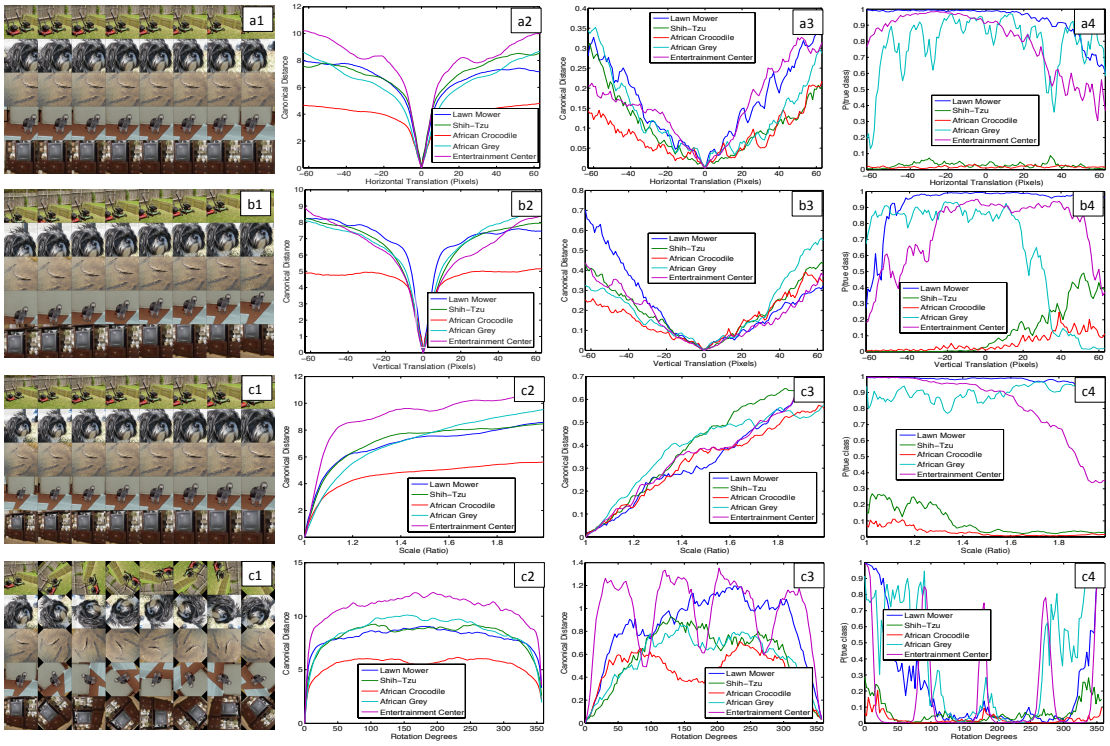

Figure 7.5: Analysis of vertical translation, scale, and rotation invariance within the model (rows a-c respectively). Col 1: 5 example images undergoing the transformations. Col 2 & 3: Euclidean distance between feature vectors from the original and transformed images in layers 1 and 7 respectively. Col 4: the probability of the true label for each image, as the image is transformed.

7.4.4 Correspondence Analysis

Deep models differ from many existing recognition approaches in that there is no explicit mechanism for establishing correspondence between specific object parts in different images (e.g. eyes and noses for faces). However, an intriguing possibility is that deep models might be *implicitly* computing them. To explore this, we take 5 randomly drawn dog images with frontal pose and systematically mask out the same part of the face in each image (e.g. all left eyes, see Figure 7.7). For each image i, we then compute: $\epsilon_i^l = h_i^l - \tilde{h}_i^l$, where h_i^l and $\tilde{h}_i^l$ are the feature vectors at layer l for the original and occluded images respectively. We then measure the consistency of this difference vector ϵ between all related image pairs (i, j): $\Delta_l = \sum_{i,j=1,i\neq j}^{5} \mathcal{H}(\text{sign}(\epsilon_i^l), \text{sign}(\epsilon_j^l))$, where $\mathcal{H}$ is Hamming distance. A lower value indicates greater consistency in the change resulting

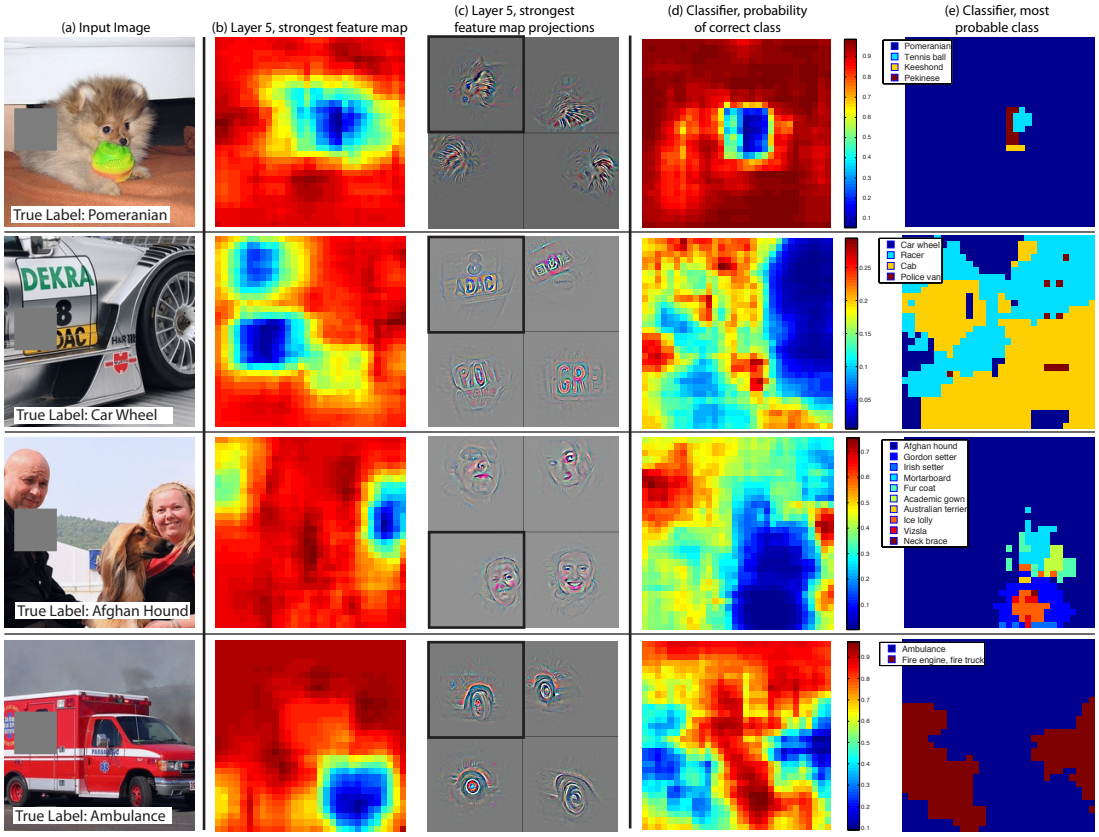

Figure 7.6: Four test examples where we systematically cover up different portions of the scene with a gray square (1st column) and see how the top (layer 5) feature maps ((b) & (c)) and classifier output ((d) & (e)) changes. (b): for each position of the gray scale, we record the total activation in one layer 5 feature map (the one with the strongest response in the unoccluded image). (c): a visualization of this feature map projected down into the input image (black square), along with visualizations of this map from other images. The first row example shows the strongest feature to be the dog's face. When this is covered-up the activity in the feature map decreases (blue area in (b)). (d): a map of correct class probability, as a function of the position of the gray square. E.g. when the dog's face is obscured, the probability for "pomeranian" drops significantly. (e): the most probable label as a function of occluder position. E.g. in the 1st row, for most locations it is "pomeranian", but if the dog's face is obscured but not the ball, then it predicts "tennis ball". In the 2nd example, text on the car is the strongest feature in layer 5, but the classifier is most sensitive to the wheel. The 3rd example contains multiple objects. The strongest feature in layer 5 picks out the faces, but the classifier is sensitive to the dog (blue region in (d)), since it uses multiple feature maps. In the 4th example, the front wheel is the strongest feature, but the output depends on many parts of the vehicle.

from the masking operation, hence tighter correspondence between the same object parts in different images. In Table 7.3 we compare the Δ score for three parts of the face (left

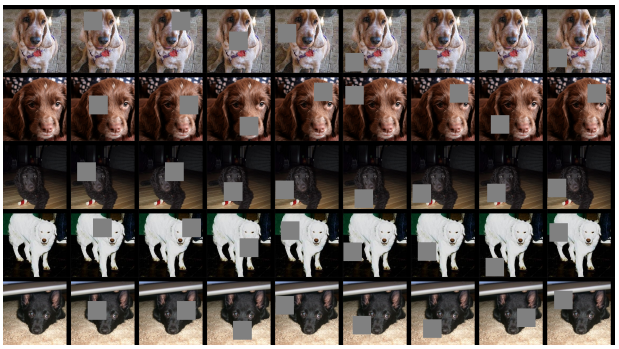

Figure 7.7: Images used for correspondence experiments. Col 1: Original image. Col 2,3,4: Occlusion of the right eye, left eye, and nose respectively. Other columns show examples of random occlusions.

eye, right eye and nose) to random parts of the object, using features from layer $l = 5$ and $l = 7$. The lower score for these parts, relative to random object regions, for the layer 5 features show the model does establish some degree of correspondence. We do not see the same gap in the layer 7 results because the spatial extend of the layer 5 features is not present in the fully connected layer 7 features and these higher features are more discriminative between the different breeds of dog.

Occlusion Location	Mean Feature Sign Change Layer 5	Mean Feature Sign Change Layer 7
Right Eye	0.067 ± 0.007	0.069 ± 0.015
Left Eye	0.069 ± 0.007	0.068 ± 0.013
Nose	0.079 ± 0.017	0.069 ± 0.011
Random	0.107 ± 0.017	0.073 ± 0.014

Table 7.3: Measure of correspondence for different object parts in 5 different dog images. The lower scores for the eyes and nose (compared to random object parts) show the model implicitly establishing some form of correspondence of parts at layer 5 in the model. At layer 7, the scores are more similar, perhaps due to upper layers trying to discriminate between the different breeds of dog or their topology being fully connected as opposed to convolutional.

7.5 Feature Generalization

The experiments above show the importance of the convolutional part of our ImageNet model in obtaining state-of-the-art performance. This is supported by the visualizations

of Figure 7.4 which show the complex invariances learned in the convolutional layers. We now explore the ability of these feature extraction layers to generalize to other datasets, namely Caltech-101 [34], Caltech-256 [42] and PASCAL VOC 2012. To do this, we keep layers 1-7 of our ImageNet-trained model fixed and train a new softmax classifier on top (for the appropriate number of classes) using the training images of the new dataset. Since the softmax contains relatively few parameters, it can be trained quickly from a relatively small number of examples, as is the case for certain datasets.

This approach is a *supervised* form of pre-training, since the bulk of the model parameters have been learned in a supervised fashion on the ImageNet data. This prevents direct comparisons to existing algorithms since they did not use the ImageNet data during training. However, the results do give an absolute assessment of the performance of features extracted by our network. We also try a second strategy of training a model from scratch, i.e. resetting layers 1-7 to random values and train them, as well as the softmax, on the training images of the dataset.

Caltech-101: We follow the procedure of [34] and randomly select 15 or 30 images per class for training and test on up to 50 images per class reporting the average of the per-class accuracies in Table 7.4, using 5 train/test folds. Training took 17 minutes for 30 images/class. The pre-trained model beats the best reported result for 30 images/class from [10] by 2.9%. The convnet model trained from scratch on only Caltech-101 images however does terribly, only achieving 46.5%.

# Train	Acc % 15/class	Acc % 30/class
Bo *et al.* [10]	–	81.4 ± 0.33
Yang *et al.* [59]	73.2	84.3
Non-pretrained convnet	22.8 ± 1.5	46.5 ± 1.7
ImageNet-pretrained convnet	$\mathbf{83.8 \pm 0.5}$	$\mathbf{86.5 \pm 0.5}$

Table 7.4: Caltech-101 classification accuracy for our convnet models, against two leading alternate approaches.

Caltech-256: We follow the procedure of [42], selecting 15, 30, 45, or 60 training images per class, reporting the average of the per-class accuracies in Table 7.5. Our ImageNet-pretrained model beats the state-of-the-art results obtained by [10] by a significant mar-

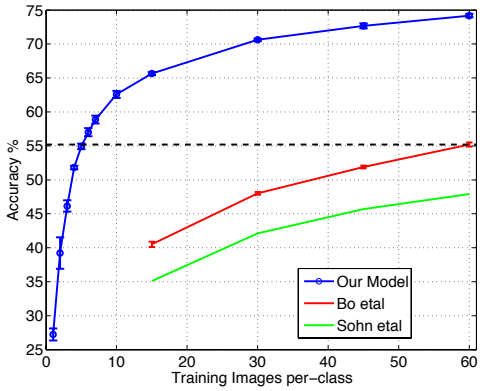

Figure 7.8: Caltech-256 classification performance as the number of training images per class is varied. Using only 6 training examples per class with our pre-trained feature extractor, we surpass best reported result by Bo *et al.* [10].

gin: 74.2% vs 55.2% for 60 training images/class. However, as with Caltech-101, the model trained from scratch does poorly. In Figure 7.8, we explore the "one-shot learning" [34] regime. With our pre-trained model, just 6 Caltech-256 training images per class are needed to beat the leading method using 10 times as many images. This shows the power of the ImageNet feature extractor.

# Train	Acc % 15/class	Acc % 30/class	Acc % 45/class	Acc % 60/class
Sohn *et al.* [121]	35.1	42.1	45.7	47.9
Bo *et al.* [10]	40.5 ± 0.4	48.0 ± 0.2	51.9 ± 0.2	55.2 ± 0.3
Non-pretr.	9.0 ± 1.4	22.5 ± 0.7	31.2 ± 0.5	38.8 ± 1.4
ImageNet-pretr.	$\mathbf{65.7 \pm 0.2}$	$\mathbf{70.6 \pm 0.2}$	$\mathbf{72.7 \pm 0.4}$	$\mathbf{74.2 \pm 0.3}$

Table 7.5: Caltech 256 classification accuracies.

PASCAL 2012: We used the standard training and validation images to train a 20-way softmax on top of the ImageNet-pretrained convnet. This is not ideal, as PASCAL images can contain multiple objects and our model just provides a single exclusive prediction for each image. Table 7.6 shows the results on the test set. The PASCAL and ImageNet images are quite different in nature, the former being full scenes unlike the latter. This may explain our mean performance being 3.2% lower than the leading [140] result, however we do beat them on 5 classes, sometimes by large margins.

Acc %	Sande *et al.* [111]	Yan *et al.* [140]	Ours	Acc %	Sande *et al.* [111]	Yan *et al.* [140]	Ours
Airplane	92.0	**97.3**	96.0	Dining tab	63.2	**77.8**	67.7
Bicycle	74.2	**84.2**	77.1	Dog	68.9	83.0	**87.8**
Bird	73.0	80.8	**88.4**	Horse	78.2	**87.5**	86.0
Boat	77.5	85.3	**85.5**	Motorbike	81.0	**90.1**	85.1
Bottle	54.3	**60.8**	55.8	Person	91.6	**95.0**	90.9
Bus	85.2	**89.9**	85.8	Potted pl	55.9	**57.8**	52.2
Car	81.9	**86.8**	78.6	Sheep	69.4	79.2	**83.6**
Cat	76.4	89.3	**91.2**	Sofa	65.4	**73.4**	61.1
Chair	65.2	**75.4**	65.0	Train	86.7	**94.5**	91.8
Cow	63.2	**77.8**	74.4	Tv	77.4	**80.7**	76.1
Mean	74.3	**82.2**	79.0	# won	0	**15**	5

Table 7.6: PASCAL 2012 classification results, comparing our Imagenet-pretrained convnet against the leading two methods Sande *et al.* [111] and Yan *et al.* [140].

7.5.1 Layer-by-Layer Performance Breakdown

We explore how discriminative the features in each layer of our Imagenet-pretrained model are. We do this by varying the number of layers retained from the ImageNet model and place either a linear SVM or softmax classifier on top. Table 7.7 shows results on Caltech-101 and Caltech-256. For both datasets, a steady improvement can be seen as we ascend the model, with best results being obtained by using all layers. This supports the premise that as the feature hierarchies become deeper, they learn increasing powerful features.

	Cal-101 (30/class)	Cal-256 (60/class)
SVM (1)	44.8 ± 0.7	24.6 ± 0.4
SVM (2)	66.2 ± 0.5	39.6 ± 0.3
SVM (3)	72.3 ± 0.4	46.0 ± 0.3
SVM (4)	76.6 ± 0.4	51.3 ± 0.1
SVM (5)	$\mathbf{86.2 \pm 0.8}$	65.6 ± 0.3
SVM (7)	$\mathbf{85.5 \pm 0.4}$	$\mathbf{71.7 \pm 0.2}$
Softmax (5)	82.9 ± 0.4	65.7 ± 0.5
Softmax (7)	$\mathbf{85.4 \pm 0.4}$	$\mathbf{72.6 \pm 0.1}$

Table 7.7: Analysis of the discriminative information contained in each layer of feature maps within our ImageNet-pretrained convnet. We train either a linear SVM or softmax on features from different layers (as indicated in brackets) from the convnet. Higher layers generally produce more discriminative features.

7.6 Discussion

In this chapter we explored large convolutional neural network models, trained for image classification, in a number ways. First, we systematically modified the network architecture to reveal that having a minimum depth to the network, rather than any individual section, is vital to the model's performance. Expanding the size of these layers results in models who performance beats that of [67]. We then presented a novel way to visualize the activity within the model. This reveals the features to be far from random, uninterpretable patterns. Rather, they show many intuitively desirable properties such as compositionally, increasing invariance and class discrimination as we ascend the layers. We also demonstrated through a series of occlusion experiments that the model, while trained for classification, is highly sensitive to local structure in the image and is not just using broad scene context. The occlusion techniques also allowed us to probe how the model may be implicitly establishing correspondence with stable object parts in the upper feature layers, when determining its prediction. Finally, we showed how the ImageNet trained model can generalize well to other datasets. For Caltech-101 and Caltech-256, the datasets are similar enough that we can beat the best reported results, in the latter case by a significant margin. This result brings into question the utility of benchmarks with small (i.e. $< 10^4$) training sets. Our convnet model generalized less well to the PASCAL data, perhaps suffering from dataset bias [130], although it was still within 3.2% of the best reported result, despite no tuning for the task. For example, our performance might improve if a different loss function was used that permitted multiple objects per image.

Chapter 8

Part 1 Conclusion

8.1 Summary of Contributions

The goal of this portion of the thesis was to explore different methods for learning hierarchical features from natural images. Throughout this work we utilized machine learning techniques as opposed to hand-crafted features in order to solve problems in computer vision. The methods introduced prove useful both for unsupervised learning of deep convolutional architectures as well as an understanding of what traditional feedforward architectures actually learn.

In Chapter 3 we introduced a novel unsupervised learning approach called deconvolutional networks. The approach incorporates multiple layers of stacked deconvolutions with the objective of optimizing the reconstruction of the previous layer's feature while forcing the current layer features to be sparse. This is done through an iterative optimization technique which is shown to learn interesting low level edge structures and mid-level corners, curves, etc. in an unsupervised fashion. These models can be applied to both denoising natural images and object classification.

To provide more invariance into deconvolutional networks so that high level object parts

can be represented, we introduced the use of switches in Chapter 4. These encode the maximum activations within pooling regions at each layer so that reconstruction can proceed from a high level of the model down to the input pixels. This objective ensures that deep models can still represent the input pixels well as opposed to just being able to reconstruct the layer below. This was shown to learn interesting high level features such as object parts and entire objects that group natural via unsupervised learning. Competitive classification performance was shown to result models trained in this way.

To extend this idea of using pooling parameters to represent the invariances at multiple levels of the model, Chapter 5 introduced a continuous representation based on 2D Gaussians in each pooling region. The mean and variance of the Gaussians can be optimized via backpropagation of the overall objective function which reconstructs from the highest level pooled features. This allows lower layer parts to adjust as the high level parts are learned. This joint training of all layers was shown to improve reconstruction performance while improving the discriminative information contained in the higher level features.

Our expertise on training deconvolutional networks was then applied to training and understanding convolutional networks beginning in Chapter 6. A novel regularization technique called stochastic pooling was introduced to prevent convolutional networks from overfitting the training set when data is scarce. This led to state of the art performance on several common benchmarks using this method which has no parameters and can be swapped in to existing models.

Finally, we investigated what is learned by convolutional networks using a novel visualization technique based on deconvolutional networks. In doing so we revealed ways to improve performance of the model for large scale object recognition while showing the diversity and invariances of the features learned at each layer of the model. This work enabled state of the art classification performance on the large scale ImageNet benchmark as well as several smaller benchmarks upon which the model was tested. We hope

this understanding of what occurs at all layers of a complex artificial neural network model will enable future work on better understanding of how these models relate to the functioning of the visual cortex.

8.2 Future Directions

Several important directions of research based on this thesis are worthwhile investigating:

Improving Classification with Deconvolutional Networks: While the deconvolutional network was shown to learn interesting features, we found that the pooling parameters contained a great deal of information. Using these parameters in addition to the high level features could provide one mechanism for improving classification performance of deconvolutional networks. Further work on incorporating a classification term into the objective should be done to pull the features towards natural groupings based on object class.

Scaling Deconvolutional Networks: The inference methods used for deconvolutional methods are inherently slow as they involve multiple iterations to be performed. Fast approximation techniques based on feedforward initializations and strong shrinkage operators should be investigated. Doing so would require only the features and pooling parameters for the current mini-batch of images to be stored, allowing deconvolutional networks to scale to large datasets. This combined with the previous objective could enable these models to leverage labelled data when available, while scaling to the vast amounts of unlabelled data in the world.

Extend Convolutional Models for detection and multiple objects: Convolutional networks have been known to performance well for object classification for many years and they have recently been applied to very large datasets. However, future work is needed for methods of applying convolutional networks to object detection and to cope with multiple objects being present in a scene at once.

Part II

Other Work

The previous chapters of this thesis were focused on solving complex computer vision tasks. The following few chapters relate to work done during my thesis that is not directly related to computer vision. The Chapter 9 involves modelling sequences of motion capture. Chapter 10 applies to speech recognition. While Chapter 11 describes an optimization technique to improve gradient descent for various problems.

Chapter 9

Facial Expression Transfer with Input-Output Temporal Restricted Boltzmann Machines

9.1 Introduction

In this chapter we work with a different modality of data, motion capture, presented as sequences of markers representing facial expressions of human. We introduce a new type of Temporal Restricted Boltzmann Machine that defines a probability distribution over an output sequence conditional on an input sequence. It shares the desirable properties of RBMs: efficient exact inference, an exponentially more expressive latent state than HMMs, and the ability to model nonlinear structure and dynamics. We apply our model to a challenging real-world graphics problem: facial expression transfer. Our results demonstrate improved performance over several baselines modeling high-dimensional 2D and 3D data.

Modeling temporal dependence is an important consideration in many learning prob-

lems. One can capture temporal structure either explicitly in the model architecture, or implicitly through latent variables which can act as a "memory". Feedforward neural networks which incorporate fixed delays into their architecture are an example of the former. A limitation of these models is that temporal context is fixed by the architecture instead of inferred from the data. To address this shortcoming, recurrent neural networks incorporate connections between the latent variables at different time steps. This enables them to capture arbitrary dynamics, yet they are more difficult to train [8].

Another family of dynamical models that has received much attention are probabilistic models such as Hidden Markov Models and more general Dynamic Bayes nets. Due to their statistical structure, they are perhaps more interpretable than their neural-network counterparts. Such models can be separated into two classes [123]: tractable models, which permit an exact and efficient procedure for inferring the posterior distribution over latent variables, and intractable models which require approximate inference. Tractable models such as Linear Dynamical Systems and HMMs are widely applied and well understood. However, they are limited in the types of structure that they can capture. These limitations are exactly what permit simple exact inference. Intractable models, such as Switching LDS, Factorial HMMs, and other more complex variants of DBNs permit more complex regularities to be learned from data. This comes at the cost of using approximate inference schemes, for example, Gibbs sampling or variational inference, which introduce either a computational burden or poorly approximate the true posterior.

In this chapter we focus on Temporal Restricted Boltzmann Machines [123,128], a family of models that permits tractable inference but allows much more complicated structure to be extracted from time series data. Models of this class have a number of attractive properties: 1) They employ a *distributed* state space where multiple factors interact to explain the data; 2) They permit nonlinear dynamics and multimodal predictions; and 3) Although maximum likelihood is intractable for these models, there exists a simple and efficient approximate learning algorithm that works well in practice.

We concentrate on modeling the distribution of an output sequence conditional on an input sequence. Recurrent neural networks address this problem, though in a non-probabilistic sense. The Input-Output HMM [6] extends HMMs by conditioning both their dynamics and emission model on an input sequence. However, the IOHMM is representationally limited by its simple discrete state in the same way as a HMM. Therefore we extend TRBMs to cope with input-output sequence pairs. Given the conditional nature of a TRBM (its hidden states and observations are conditioned on short histories of these variables), conditioning on an external input is a natural extension to this model.

Several real-world problems involve sequence-to-sequence mappings. This includes motion-style transfer [53], economic forecasting with external indicators [86], and various tasks in natural language processing [24]. Sequence classification is a special case of this setting, where a scalar target is conditioned on an input sequence. In this chapter, we consider facial expression transfer, a well-known problem in computer graphics, in which a sequence of facial movements is mapped onto another. Current methods considered by the graphics community are typically linear (e.g., methods based on blendshape mapping) and they do not take into account dynamical aspects of the facial motion itself. This makes it difficult to retarget the facial articulations involved in speech. We propose a model that can encode a complex nonlinear mapping from the motion of one individual to another which captures facial geometry and dynamics of both source and target.

In the following subsections we discuss several latent variable models which can map an input sequence to an output sequence. We also briefly review our application field: facial expression transfer.

9.1.1 Temporal models

Among probabilistic models, the Input-Output HMM [6] is most similar to the architecture we propose. Like the HMM, the IOHMM is a generative model of sequences but it models the distribution of an output sequence conditional on an input, while

the HMM simply models the distribution of an output sequence. The IOHMM is also trained with a more discriminative-style EM-based learning paradigm than HMMs. A similarity between IOHMMs and TRBMs is that in both models, the dynamics and emission distributions are formulated as neural networks. However, the IOHMM state space is a multinomial while TRBMs have binary latent states. A K-state TRBM can thus represent the history of a time series using 2^K state configurations while IOHMMs are restricted to K settings.

The Continuous Profile Model [79] is a rich and robust extension of dynamic time warping that can be applied to many time series in parallel. The CPM has a discrete state-space and requires an input sequence. Therefore it is a type of conditional HMM. However, unlike the IOHMM and our proposed model, the input is unobserved, making learning completely unsupervised.

Our approach is also related to the many proposed techniques for supervised learning with structured outputs. The problem of simultaneously predicting multiple, correlated variables has received a great deal of recent attention [3]. Many of these models, including the one we propose, are formally defined as undirected graphs whose potential functions are functions of some input. In Graph Transformer Networks [74] the dependency structure on the outputs is chosen to be sequential, which decouples the graph into pairwise potentials. Conditional Random Fields [68] are a special case of this model with linear potential functions. These models are trained discriminatively, typically with gradient descent, where our model is trained generatively using an approximate algorithm.

9.1.2 Facial expression transfer

Facial expression transfer, also called motion retargeting or cross-mapping, is the act of adapting the motion of an actor to a target character. It, as well as the related fields of facial performance capture and performance-driven animation, have been very active research areas over the last several years.

According to a review by Pighin [94], the two most important considerations for this task are facial model parameterization (called "the rig" in the graphics industry) and the nature of the chosen cross-mapping. A popular parameterization is "blendshapes" where a rig is a set of linearly combined facial expressions each controlled by a scalar weight. Retargeting amounts to estimating a set of blending weights at each frame of the source data that accurately reconstructs the target frame. There are many different ways of selecting blendshapes, from simply selecting a set of sufficient frames from the data, to creating models based on principal components analysis. Another common parameterization is to simply represent the face by its vertex, polygon or spline geometry. The downside of this approach is that this representation has many more degrees of freedom than are present in an actual facial expression.

A linear function is the most common choice for cross-mapping. While it is simple to estimate from data, it cannot produce subtle nonlinear motion required for realistic graphics applications. An example of this approach is [21] which uses a parametric model based on eigen-points to reliably synthesize simple facial expressions but ultimately fails to capture more subtle details. Vlasic et al. [135] have proposed a multi-linear mapping where variation in appearance across the source and target is explicitly separated from the variation in facial expression. None of these models explicitly incorporate dynamics into the mapping, which is a limitation addressed by our approach.

Finally, we note that Susskind et al. [122] have used RBMs for facial expression generation, but not retargeting. Their work is focused on static rather than temporal data.

9.2 Modeling dynamics with Temporal Restricted Boltzmann Machines

In this section we review the Temporal Restricted Boltzmann Machine. We then introduce the Input-Output Temporal Restricted Boltzmann Machine which extends the

architecture to model an output sequence conditional on an input sequence.

9.2.1 Temporal Restricted Boltzmann Machines

A Restricted Boltzmann Machine [118] is a bipartite Markov Random Field consisting of a layer of stochastic observed variables ("visible units") connected to a layer of stochastic latent variables ("hidden units"). The absence of connections between hidden units ensures they are conditionally independent given a setting of the visible units, and vice-versa. This simplifies inference and learning.

The RBM can be extended to model temporal data by conditioning its visible units and/or hidden units on a short history of their activations. This model is called a Temporal Restricted Boltzmann Machine [123]. Conditioning the model on the previous settings of the hidden units complicates inference. Although one can approximate the posterior distribution with the filtering distribution (treating the past setting of the hidden units as fixed), we choose to use a simplified form of the model which conditions only on previous *visible* states [128]. This model inherits the most important computational properties of the standard RBM: simple, exact inference and efficient approximate learning.

RBMs typically have binary observed variables and binary latent variables but to model real-valued data (e.g., the parameterization of a face), we can use a modified form of the TRBM with conditionally independent linear-Gaussian observed variables [38]. The model, depicted in Figure 9.1 (left), defines a joint probability distribution over a real-valued representation of the current frame of data, x_t, and a collection of binary latent variables, h_t, where $h_{k,t} \in \{0, 1\}$:

$$P(x_t, h_t | x_{<t}) = \frac{\exp\left(-E(x_t, h_t | x_{<t})\right)}{Z(x_{<t})} \tag{9.1}$$

For notational simplicity, we concatenate a short history of data at $t-1,\ldots,t-\tau$ into a vector which we call $x_{<t}$. The distribution specified by Eq. 9.1 is conditional on this

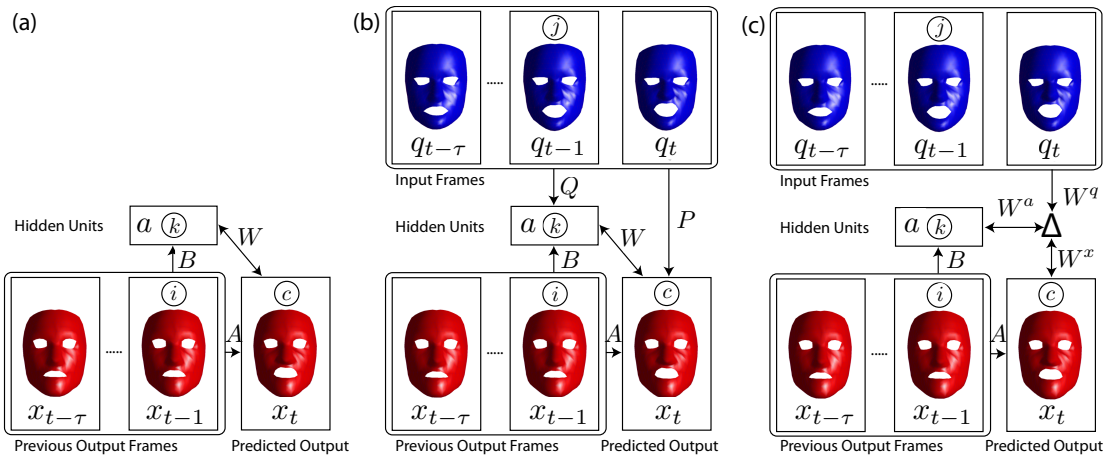

Figure 9.1: Left: A Temporal Restricted Boltzmann Machine. Middle: An Input-Output Temporal Restricted Boltzmann Machine. Right: A factored third-order IOTRBM (FIOTRBM).

history and normalized by a quantity Z which is intractable to compute exactly[1] but not needed for inference nor learning.

The joint distribution is characterized by an "energy function":

$$E(x_t, h_t|x_{<t}) = \sum_c \frac{1}{2}(x_{c,t} - \hat{b^x}_{c,t})^2 - \sum_k h_{k,t}\hat{b^h}_{k,t} - \sum_{ck} W_{ck}x_{c,t}h_{k,t} \tag{9.2}$$

which captures pairwise interactions between variables, assigning high energy to improbable configurations and low energy to probable configurations. In the first term, each visible unit contributes a quadratic penalty that depends on its deviation from a "dynamic mean" determined by the history:

$$\hat{b^x}_{c,t} = {b^x}_c + \sum_i A_{ic}x_{i,<t} \tag{9.3}$$

where i indexes the dimensions of the history vector. Weight matrix A and offset vector b^x (with elements ${b^x}_c$) parameterize the autoregressive relationship between the history

[1]To compute Z exactly we would need to integrate over the joint space of all possible output configurations and all settings of the binary latent variables.

and current frame of data. Each hidden unit activation h_k contributes a linear offset to the energy which is also a function of the history:

$$\hat{b^h}_{k,t} = {b^h}_k + \sum_i B_{ik} x_{i,<t}. \tag{9.4}$$

Weight matrix B and offset b^h (with elements ${b^h}_k$) parameterize the relationship between the history and the latent variables. The final term of Eq. 9.2 is a bi-linear constraint on the interaction between the current setting of the visible units and hidden units, characterized by matrix W.

The density for observation x_t conditioned on the past can be expressed by marginalizing out the binary hidden units in Eq. 9.1:

$$P(x_t|x_{<t}) = \sum_{h_t} P(x_t, h_t|x_{<t}) = \frac{\sum_{h_t} \exp\left(-E(x_t, h_t|x_{<t})\right)}{Z(x_{<t})} \tag{9.5}$$

while the probability of observing a *sequence*, $x_{(\tau+1):T}$, given an τ-frame history $x_{1:\tau}$, is simply the product of all the local conditional probabilities up to time T, the length of a sequence:

$$P(x_{(\tau+1):T}|x_{1:\tau}) = \prod_{t=\tau+1}^{T} P(x_t|x_{<t}). \tag{9.6}$$

The TRBM has been used to generate and denoise sequences [123, 128], as well as a prior in multi-view person tracking [127]. In all cases, it requires an initialization, $x_{1:\tau}$, to perform these tasks. Alternatively, by learning a prior model of $x_{1:\tau}$ it could easily extended to model sequences non-conditionally, i.e., defining $P(x_{1:T})$.

9.2.2 Input-Output Temporal Restricted Boltzmann Machines

Ultimately we are interested in learning a probabilistic mapping from an input sequence, $q_{1:T}$ to an output sequence, $x_{1:T}$. In other words, we seek a model that defines $P(x_{1:T}|q_{1:T})$. However, the TRBM only defines a distribution over an output sequence $P(x_{1:T})$. Extending this model to learn an input-output mapping is the primary contribution of this chapter. Without loss of generality, we will assume that in addition to having access to the complete history of the input, we also have access to the first τ frames of the output. Therefore we seek to model $P(x_{(\tau+1):T}|x_{1:\tau}, q_{1:T})$. By placing an τ^{th} order Markov assumption on the current output, x_t, that is, assuming conditional independence on all other variables given an τ-frame history of x_t and an $\tau+1$-frame history of the input sequence $q_{<=t}$ (up to and including time t), we can operate in an online setting:

$$P(x_{(\tau+1):T}|x_{1:\tau}, q_{1:T}) = \prod_{t=\tau+1}^{T} P(x_t|x_{<t}, q_{<=t}). \tag{9.7}$$

where we have used the shorthand $q_{<=t}$ to describe a vector that concatenates a window over the input at time $t, t-1, \ldots, t-\tau$. Note that in an offline setting, it is simple to generalize the model by conditioning the term inside the product on an arbitrary window of the source (which may include source observations past time t).

We can easily adapt the TRBM to model $P(x_t|x_{<t}, q_{<=t})$ by modifying its energy function to incorporate the input. The general form of energy function remains the same as Eq. 9.2 but it is now also conditioned on $q_{<=t}$ by redefining the dynamic biases (Eq. 9.3 and Eq. 9.4) as follows:

$$\hat{b^x}_{ct} = {b^x}_c + \sum_i A_{ic} x_{i,<t} + \sum_j P_{jc} q_{j,<=t} \tag{9.8}$$

$$\hat{b^h}_{kt} = {b^h}_k + \sum_i B_{ik} x_{i,<t} + \sum_j Q_{jk} q_{j,<=t} \tag{9.9}$$

where j is an index over elements of the input vector. Therefore the matrix P ties the input linearly to the output (much like existing simple models) but the matrix Q also allows the input to nonlinearly interact with the output through the latent variables h. We call this model an Input-Output Temporal Restricted Boltzmann Machine (IOTRBM). It is depicted in Figure 9.1 (middle).

A desirable criterion for training the model is to maximize the conditional log likelihood of the data:

$$C = \sum_{t=\tau+1}^{T} \log P(x_t | x_{<t}, q_{<=t}). \tag{9.10}$$

However, the gradient of the cost defined in Eq. 9.10 with respect to the model parameters $\theta = \{W, A, B, P, Q, b^x, b^h\}$ is difficult to compute analytically due to the normalization constant Z. Therefore, Contrastive Divergence (CD) learning is typically used in place of maximum likelihood. It follows the approximate gradient of an objective function that is the difference between two Kullback-Leibler divergences [45]. It is widely used in practice and tends to produce good generative models [16].

The CD updates for the IOTRBM have a common form (see the supplementary material for details [2]):

$$\Delta\theta_i \propto \sum_{t=\tau+1}^{T} \left\langle \frac{\partial E(x_t, h_t | x_{<t}, q_{<=t})}{\partial \theta_i} \right\rangle_{\text{data}} - \left\langle \frac{\partial E(x_t, h_t | x_{<t}, q_{<=t})}{\partial \theta_i} \right\rangle_{\text{recon}} \tag{9.11}$$

[2] `http://www.matthewzeiler.com/pubs/nips2011/`

where $\langle\cdot\rangle_{\text{data}}$ is an expectation with respect to the training data distribution, and $\langle\cdot\rangle_{\text{recon}}$ is the M-step reconstruction distribution as obtained by alternating Gibbs sampling, starting with the visible units clamped to the training data. The input and output history stay fixed during Gibbs sampling. CD requires two main operations: 1) sampling the latent variables, given a window of the input and output,

$$P(h_{k,t} = 1|x_t, x_{<t}, q_{<=t}) = \left(1 + \exp(-\sum_c W_{ck}x_{c,t} - \hat{b^h}_{kt})\right)^{-1}, \tag{9.12}$$

and 2) reconstructing the output data, given the latent variables:

$$\hat{x}_{c,t}|h_t, x_{<t}, q_{<=t} \sim \mathcal{N}\left(x_{ct}; \sum_k W_{ck}h_{k,t} + \hat{b^x}_{c,t}, 1\right). \tag{9.13}$$

Eq. 9.12 and Eq. 9.13 are alternated M times to arrive at the M-step quantities used in the weight updates. More details are given in Section 9.3.

9.2.3 Factored Third-order Input-Output Temporal Restricted Boltzmann Machines

In an IOTRBM the input and target history can only modify the hidden units and current output through additive biases. There has been recent interest in exploring higher-order RBMs in which variables interact multiplicatively [87, 98, 126]. Figure 9.1 (right) shows an IOTRBM whose parameters W, Q and P have been replaced by a three-way weight tensor defining a multiplicative interaction between the three sets of variables. The introduction of the tensor results in the number of model parameters becoming cubic and therefore we factor the tensor into three matrices: W^q, W^h, and W^x. These parameters connect the input, hidden units, and current target, respectively to a set of deterministic units which modulate the connections between variables. The introduction of these factors corresponds to a kind of low-rank approximation to the original interaction tensor, that uses $O(K^2)$ parameters instead of $O(K^3)$.

The energy function of this model is:

$$E(x_t, h_t|x_{<t}, q_{<=t}) = \sum_c \frac{1}{2}(x_{c,t} - \hat{b^x}_{c,t})^2 - \sum_k h_{k,t}\hat{b^h}_{k,t} - \sum_f \sum_{ckj} W^x_{cf} W^h_{kf} W^q_{jf} x_{c,t} h_{k,t} q_{j,<=t} \tag{9.14}$$

where f indexes factors and $\hat{b^x}_{c,t}$ and $\hat{b^h}_{k,t}$ are defined by Eq. 9.3 and Eq. 9.4 respectively. Weight updates all have the same form as Eq. 9.11 (see the supplementary material for details [3]). The conditional distribution of the latent variables given the other variables becomes,

$$P(h_{k,t} = 1|x_t, x_{<t}, q_{<=t}) = \left(1 + \exp(-\sum_f W^h_{kf} \sum_c W^x_{cf} x_{c,t} \sum_j W^q_{jf} q_{j,<=t} - \hat{b^h}_{kt})\right)^{-1} \tag{9.15}$$

and the reconstruction distribution becomes,

$$x_{c,t}|h_t, x_{<t}, q_{<=t} \sim \mathcal{N}\left(x_{ct}; \sum_f W^x_{cf} \sum_k W^h_{kf} h_{k,t} \sum_j W^q_{jf} q_{j,<=t} + \hat{b^x}_{c,t}, 1\right). \tag{9.16}$$

9.3 Experiments

We evaluate the IOTRBM on two facial expression transfer datasets, one based on 2D motion capture and the other on 3D motion capture. On both datasets we compare our model against three baselines:

Linear regression (LR): We perform a regularized linear regression between each frame of the input to each frame of the output. The model is solved analytically by least squares. The regularization parameter is set by cross-validation on the training set.

τ^{th}-order Autoregressive[4] model (AR): This model improves on linear regression

[3] `http://www.matthewzeiler.com/pubs/nips2011/`

[4] This model considers the history of the source when predicting the target so it is not purely autoregressive.

by also considering linear dynamics through the history of the input and output. Again through regularized least squares we fit a matrix that maps from a concatenation of the $(\tau + 1)$-frame input window $q_{<=t}$ and τ-frame target window, $x_{<t}$.

Multi-layer perceptron: A nonlinear model with one deterministic hidden layer, the same cardinality as the IOTRBM. The input is the concatenation of the source and target history, the output is the current target frame. We train with a nonlinear conjugate gradient method.

These baselines were chosen to highlight the main difference of our approach over the majority of techniques proposed for this application, namely the consideration of dynamics and the use of a nonlinear mapping through latent variables. We also tried an IORBM, that is, an IOTRBM with no target history. It consistently performed worse than the IOTRBM, and we do not report its results.

Details of learning All models saw a window of 4 input frames (3 previous + 1 current) and 6 previous output frames, with the exception of linear regression which only saw the current input. For the IOTRBM models, we found that initializing the parameters A and P to the solution found by the autoregressive model gave slightly better results. All other parameters were initialized to small random values. For CD learning we set the learning rates for A and P to 10^{-6} and for all other parameters to 10^{-3}. This was done to prevent strong correlations from dominating early in learning. All parameters used a fixed weight decay of 0.02 and momentum of 0.75. As suggested by [126], we added a small amount of Gaussian noise ($\sigma = 0.1$) to the output history to make the model more robust to unseen outputs during prediction (recall that the model sees true outputs at training time, but is fed back predictions at test time).

9.3.1 2D facial expression transfer

The first dataset we consider consists of facial motion capture of two subjects who were asked to speak the same phrases. It has 186 trials, totaling 10414 fames per subject.

Each frame is 180 dimensional, representing the x and y position of 90 facial markers. Each pair of sequences has been manually time-aligned based on a phonetic transcription so they are synchronized between subjects.

	RMS Marker Error (mm)						
Model \ Split	S1	S2	S3	S4	S5	S6	Mean
Linear regression	6.19	6.18	6.19	5.85	6.13	6.34	6.15 ± 0.15
Autoregressive	5.43	**5.22**	5.67	5.37	5.37	5.76	5.47 ± 0.20
MLP	5.30	5.28	5.76	5.31	5.28	5.31	5.37 ± 0.19
IOTRBM	**5.31**	5.27	**5.71**	**5.14**	**5.17**	**5.08**	**5.28 ± 0.22**
FIOTRBM	5.41	5.43	5.76	5.42	5.45	5.46	5.49 ± 0.13

Table 9.1: 2D dataset. Mean RMS error on test output sequences.

	Input noise			Output noise			Input & Output Noise		
Model \ Noise	0.01	0.1	1	0.01	0.1	1	0.01	0.1	1
Linear regression	6.48	15.05	136.2	N/A					
Autoregressive	5.83	10.48	84.40	5.78	7.24	36.19	5.85	11.26	94.35
MLP	5.40	5.42	6.80	5.40	5.43	6.37	5.40	5.43	7.55
IOTRBM	**5.06**	**5.07**	**5.39**	**5.07**	**5.18**	8.48	**5.07**	**5.17**	8.57
FIOTRBM	5.46	5.46	5.66	5.46	5.46	**5.56**	5.46	5.46	**5.82**

Table 9.2: 2D dataset. Mean RMS error (in millimeters) under noisy input and output history (Split 6).

Preprocessing

We found the original data to exhibit significant random relative motion between the two faces throughout the entire sequences which could not reasonably be modeled. Therefore, we transformed the data with an affine transform on all markers in each frame such that a select few nose and skull markers per frame (stationary facial locations) were approximately fixed relative to the first frame of the source sequences. Both the input and output were reduced to 30 dimensions by retaining only their first 30 principal components. This maintained 99.9% of the variance in the data. Finally, the data was normalized to have zero mean and scaled by the average standard deviation of all the elements in the training set.

We evaluate the various methods on 6 random arbitrary splits of the dataset. In each case, 150 complete sequences are maintained for training and the remaining 36 sequences are used for testing. Each model is presented with the first 6 frames of the true test output and successive 4-frame windows of the true test input. The exception is the linear regression model, which only sees the current input. Therefore prediction is measured from the 7^{th} frame onward.

The IOTRBM produces its final output by initializing its visible units with the current previous frame plus a small amount of Gaussian noise and then performing 30 alternating Gibbs steps. At the last step, we do not sample the hidden units. This predicted output frame now becomes the most recent frame in the output history and we iterate forward. The results show a IOTRBM with 30 hidden units. We also tried a model with 100 hidden units which performed slightly worse. Finally, we include the performance of a factored, third-order IOTRBM. This model used 30 hidden units and 50 factors.

We report RMS marker error in millimeters where the mean is taken over all markers, frames and test sequences (Table 9.1). Not surprisingly, the IOTRBM consistently outperforms linear regression. In all but two splits (where performance is comparable) the IOTRBM outperforms the AR model. Mean performance over the splits shows an advantage to our approach. This is also qualitatively apparent in videos that superimpose the true target with predictions from the model [5]. We encourage the reader to view the videos, as certain aesthetic properties such as the tradeoff between smoothness and responsiveness are not captured by RMS error. We observed that on the 2D dataset, the FIOTRBM had no advantage over the simpler IOTRBM.

To compare the robustness of each model to corrupted inputs or outputs, we added various amounts of white Gaussian noise to the input window, output history initialization or both during retargeting with a trained model. This is performed for data split S6 (though we observed similar results for other splits). The performance of each model is given in Table 9.2.

[5] See `http://www.matthewzeiler.com/pubs/nips2011/`

The IOTRBM generally outperforms the baseline models in the presence of noise. This is most apparent in the case of input noise: the scenario we would most likely find in practice. However, under low to moderate output noise, we note that the IOTRBM is robust, to the point that it does not even require a valid N frame output initialization to produce a sensible retargeting. Interestingly, we also observe the FIOTRBM performing well under high-noise conditions.

9.3.2 3D facial expression transfer

The second dataset we consider consists of facial motion capture data of two subjects, asked to perform a set of isolated facial movements based on FACS. The movements are more exaggerated than the speech performed in the 2D set. The dataset consists of two trials, totaling 1050 frames per subject. In contrast to the 2D set, the marker set used differs between subjects. The first subject has 313 markers (939 dimensions per frame) and the second subject has 332 markers (996 dimensions per frame). There is no correspondence between marker sets.

Preprocessing The 3D data was not spatially aligned. Both the input and output were PCA-reduced to 50 dimensions (99.9% of variance). We then normalized in the same way as for the 2D data.

We evaluate performance on 5 random splits of the 3D dataset, shown in Table 9.3. The IOTRBM and FIOTRBM models considered have identical architectures to the ones used for 2D data. We found empirically that increasing the noise level of the output history to $\sigma = 1$ improved generalization on this smaller dataset.

Similar to the experiments with 2D data, the IOTRBM consistently outperforms the autoregressive model. However, it does not outperform the MLP. Interestingly, the factored, third-order model considerably improves on the performance of the standard IOTRBM and the MLP. Figure 9.2 visualizes the predictions made by the FIOTRBM. We

	RMS Marker Error (mm)					
Split	S1	S2	S3	S4	S5	Mean
Autoregressive	2.12	2.98	2.44	2.26	2.46	2.45 ± 0.33
MLP	1.98	1.58	1.69	1.51	**1.39**	1.63 ± 0.22
IOTRBM	1.98	2.62	2.37	2.11	2.27	2.27 ± 0.25
FIOTRBM	**1.70**	**1.54**	**1.55**	**1.42**	1.48	**1.54 ± 0.10**

Table 9.3: 3D dataset. Mean RMS error on test output sequences.

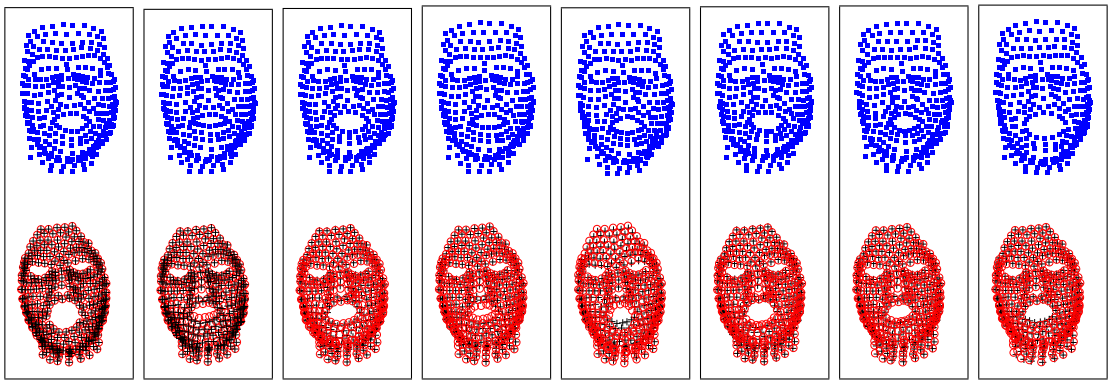

Figure 9.2: Retargeting with the third-order factored TRBM. We show every 30th frame. The top row shows the input. The bottom row shows the true target (circles) and the prediction from our model (crosses). This figure is best viewed in electronic form and zoomed.

also refer the reader to videos included as supplementary material [6]. These demonstrate a qualitative improvement of our models over the baselines considered.

9.4 Discussion

We have introduced the Input-Output Temporal Restricted Boltzmann Machine, a probabilistic model for learning mappings between sequences. We presented two variants of the model, one with pairwise and one with third-order multiplicative interactions. Our experiments so far are limited to dynamic facial expression transfer, but nothing restricts the model to this domain.

Current methods for facial expression transfer are unable to factor out style in the retargeted motion, making it difficult to adjust the emotional content of the resulting facial

[6] `http://www.matthewzeiler.com/pubs/nips2011/`

animation. We are therefore interested in exploring extensions of our model that include style-based contextual variables (c.f., [126]).

Acknowledgments

The authors thank Rafael Tena and Sarah Hilder for assisting with data collection and annotation.

Matlab code

Code is available at: `http://www.matthewzeiler.com/pubs/nips2011/`.

Chapter 10

On Rectified Linear Units for Speech Processing

10.1 Introduction

In addition to the previous applications shown thus far, deep neural networks have recently become the gold standard for acoustic modeling in speech recognition systems. The key computational unit of a deep network is a linear projection followed by a point-wise non-linearity, which is typically a logistic. In this work, we show that we can improve generalization and make training of deep networks faster and simpler by substituting the logistic units with rectified linear units. These units are linear when their input is positive and zero otherwise. In a supervised setting, we can successfully train very deep nets from random initialization on a LVCSR (large vocabulary continuous speech recognition) task achieving lower word error rates than using a logistic network with the same topology. All our experiments are executed in a distributed environment using several hundred machines and several hundred of hours of speech data.

Recent years have seen a surge of interest in neural networks for acoustic modeling in speech recognition systems. Compared to traditional Gaussian Mixture Models (GMMs),

neural networks have two main advantages. They scale better with the input dimensionality allowing the use of larger context windows and they automatically learn discriminative features from data alleviating the problem of manually engineering and selecting features. Together these two factors have yielded dramatic improvements in terms of word error rate.

In their seminal work, Mohamed et al. [88] proposed to use a system composed of many layers of logistic units. In order to overcome the notoriously difficult problem of optimizing very deep networks, they proposed to use a layer-wise unsupervised learning algorithm, called RBM [45], as a way to provide a sensible initialization and they demonstrated significant improvements over the baseline GMM.

One issue with this training procedure is that tracking convergence of RBMs is difficult and the overall layer-wise training procedure is laborious and time-consuming, even when using specialized hardware like GPUs. This challenge continues to motivate researchers to design better unsupervised algorithms [95, 107] and better optimization methods for training deep neural nets [29, 63].

Inspired by recent work on deep learning for vision applications [67, 90], we propose to replace the logistic non-linearity with a half-rectification non-linearity which is linear for positive values and zero otherwise. Because of the shape of this non-linearity, we call the resulting deep network a "hinge neural network" (HNN), and the units that compose the HNN "rectified linear units" (ReLUs) [90].

This small change brings several advantages. First, it eliminates the necessity to have a "pretraining" phase using unsupervised learning. We demonstrate empirically that we can easily and successfully train extremely deep networks even from random initialization. Second, the convergence of HNN is faster than in a regular logistic neural net with the same topology. Third, HNN is very simple to optimize. Even vanilla stochastic gradient descent with constant learning rate yields very good accuracy. Fourth, HNN generalizes better than its logistic counterpart. And finally, rectified linear units are

faster to compute because they do not require exponentiation and division, with an overall speed up of 25% on the 4 hidden layer neural network we tested on.

We conjecture that the reason why rectified linear units are so beneficial for efficient learning of deep neural nets is twofold. From the optimization perspective, HNN is piecewise linear. If we restrict our attention to the units that are non-zero, the whole system reduces to a linear convex system whose optimization is straightforward even using first order optimizers. Secondly, HNN generalizes better because the internal representation produced by the network is much more regularized. Unlike logistic units that produce small positive values when the input is not aligned with the internal weights, rectified linear units output exact zeros. For instance, we found that in average about 80% of the units in a HNN are zero after training. Improved generalization can be seen as the effect of the increased sparsity of the internal representation or also by interpreting HNN as a system with stacked binary linear SVM's (as opposed to logistic regression classifiers), and it is well known that such classifiers enjoy better generalization properties.

Overall, we demonstrate a clear advantage of ReLU's over logistic units in the supervised setting. Our empirical validation uses a recently introduced distributed framework [27] that allows us to train on four hundred machines using 1600 cores to process 26 million frames every day.

10.2 Supervised Learning

Our supervised learning set up is conventional, except for the use of the proposed activation function. The network is given both an input x (typically a few consecutive frames of a spectrogram representation) and a label representing the state of the HMM for that input. The network processes the input through a sequence of non-linear transformations. In particular, at the l-th layer the network computes:

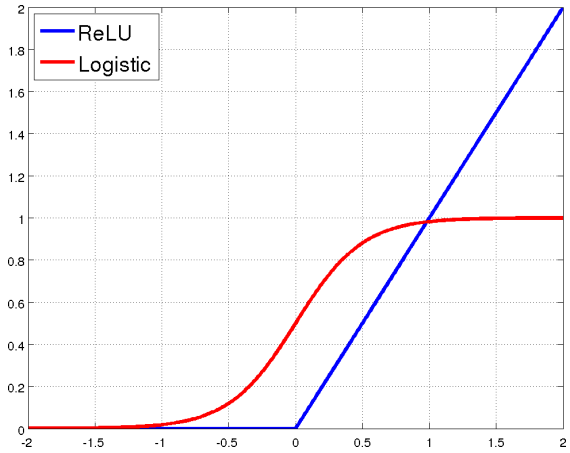

Figure 10.1: The proposed non-linearity, ReLU, and the standard neural network non-linearity, logistic.

$$h^l = \sigma(W^l h^{l-1} + b^l) \tag{10.1}$$

where $W^l \in \mathbb{R}^{N_k{}^l \times N_k{}^{l-1}}$ is a matrix of trainable weights, $b^l \in \mathbb{R}^{N_k{}^l}$ is a vector of trainable biases, and $h^{l-1} \in \mathbb{R}^{N_k{}^{l-1}}$ is the $(l-1)$-th hidden layer (or the input x if l is equal to 0) and $h^l \in \mathbb{R}^{N_k{}^l}$ is the l-th hidden layer.

In our work, we propose to use as f the following point-wise non-linear function: $\sigma(u) = \max(0, z)$. The resulting unit in the network is dubbed ReLU [90]. We have also experimented and compared to other functions as well. We tested the widely used *logistic*, $\sigma(z) = 1/(1 + \exp(-z))$, and *hyperbolic tangent*, $\sigma(z) = \tanh(z)$. Since hyperbolic tangent performed slightly worse than logistic, we only report the latter for our baseline comparisons.

In order to predict the label, the topmost layer of the network uses a softmax non-linearity which outputs probability values. If the network has N_l layers, then the prediction for the probability of the c-th class is:

$$P(c|x) = \frac{\exp(W_c^{N_l} h^{N_l - 1} + b_c^{N_l})}{\sum_{j=1}^{N_c} \exp(W_j^{N_l} h^{N_l} + b_j^{N_l})} \tag{10.2}$$

where $W_j^{N_l}$ is j-th row of the last layer weight matrix, $b_j^{N_l}$ is the j-th entry in the last layer vector of biases and N_c is the number of classes.

Training the parameters of the network (the weight matrices and biases at all layers) is performed by minimizing the cross entropy loss over the training set. The contribution of each sample x to the loss is:

$$C^{\text{sup}} = -\sum_{j=1}^{C} \mathbf{t}_j \log P(j|x;\theta) \tag{10.3}$$

where $\mathbf{t}$ is a 1-of-C encoding of the target class label and θ collectively denotes all parameters of the neural network, namely $\{(W^l, b^l), i = 0, \ldots, N_l\}$. We will discuss in sec. 10.3 how we minimize this loss function.

10.3 Learning in a Distributed Framework

In order to support training on vast amount of data in very short time, we use a recently proposed distributed framework [27]. The hidden units of the network are partitioned across several machines and each machine further parallelizes computation across several cores. Parallel distributed computation is used across the samples in a mini-batch as well as across the nodes of the neural network.

In the experiments of sec. 10.4 we use this framework and learn the parameters of the system by *asynchronous* stochastic gradient descent (SGD) [27]. Training proceeds as follows. The network is replicated $N_{replicas}$ times. Each replica is an exact copy of the model, with possibly slightly stale parameters and operating on a random subset of the training data. Besides the $N_{replicas}$ replicas, there is also a sharded parameter server hosting the most updated version of the parameters. Once a model replica has finished computing the gradients on its mini-batch, it sends them to the parameter server which uses them to update the parameters. Finally, the parameter server sends back

an updated copy of the parameters to that model replica. This mechanism allows many model replicas to work concurrently but asynchronously on the same training problem and to quickly update the parameters, while being tolerant to machine failure and high latency.

In this work, we investigated three different ways to update the parameters in the parameter server. Let θ_t^l be the l-th parameter after $t-1$ weight updates. In vanilla SGD the parameters are updated using:

$$\theta_{t+1}^l = \theta_t^l - \eta \nabla_{\theta_t^l} \tag{10.4}$$

where η is the learning rate. In SGD with ADAGRAD [27, 30], each parameter has its own adaptive learning rate; the parameter update is

$$\theta_{t+1}^l = \theta_t^l - \eta_t^l \nabla_{\theta_t^l} , \qquad \eta_t^l = \frac{\eta}{\sqrt{\sum_{\tau=1}^t (\nabla_{\theta_\tau^l})^2}} \tag{10.5}$$

Finally, in SGD with momentum the parameters are updated by:

$$\theta_{t+1}^l = \theta_t^l - \eta \Delta_{t+1}^l \tag{10.6}$$

with $\Delta_{t+1}^l = 0.9\Delta_t^l + \nabla_{\theta_t^l}$. While ADAGRAD aims at gently scaling and annealing learning rates, momentum speeds up learning along those gradient directions that are persistent during training.

10.4 Experiments

All experiments are performed using several hundred hours of US English data collected using Voice Search, Voice IME and read data at Google. The test set follows the same

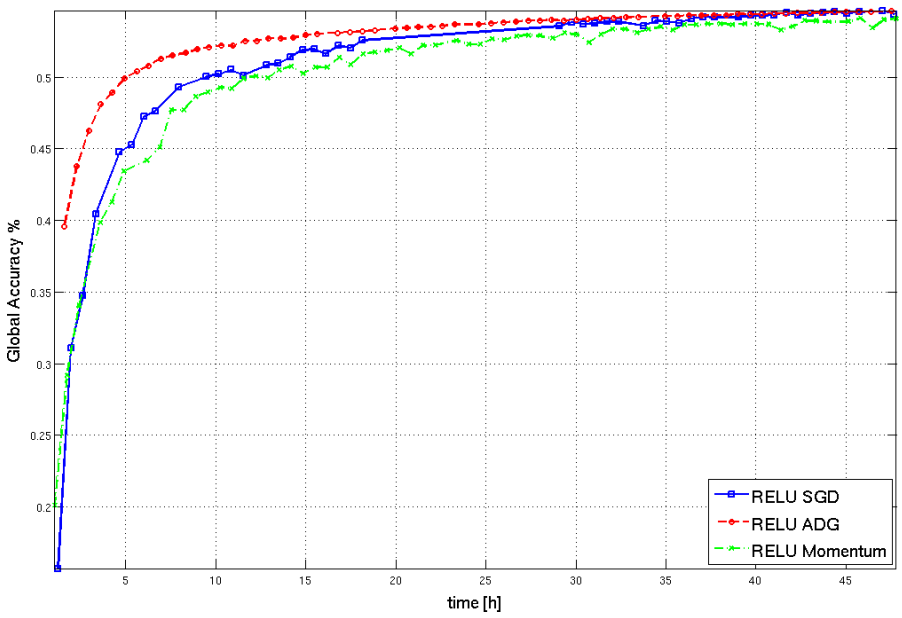

Figure 10.2: Test set frame accuracy as a function of time of a 4 hidden layer HNN trained with different optimizers.

distribution of the training set but uses independent sources. The set up for the hybrid decoding is exactly the same as the one described in earlier work [55].

In the supervised setting, a baseline GMM-HMM system is trained and used to generate 8000 context dependent senones. This system is also used to produce senone labels for every input frame using forced alignment. These labels are the target for the supervised network.

The input to the network consists of 26 consecutive frames, each comprising 40 log-energy filter bank outputs. The overall input dimensionality is 1040, although spectral analysis reveals that 95% of the variance is concentrated in the first leading 100 dimensional principal components.

Finally, all layers of our networks have 2560 hidden units and training has been performed by partitioning each network across 4 machines using up to 4 CPUs each. The number of model replicas $N_{replicas}$ has been set to 100.

In the first experiment shown in fig. 10.2, we compare the three different optimization

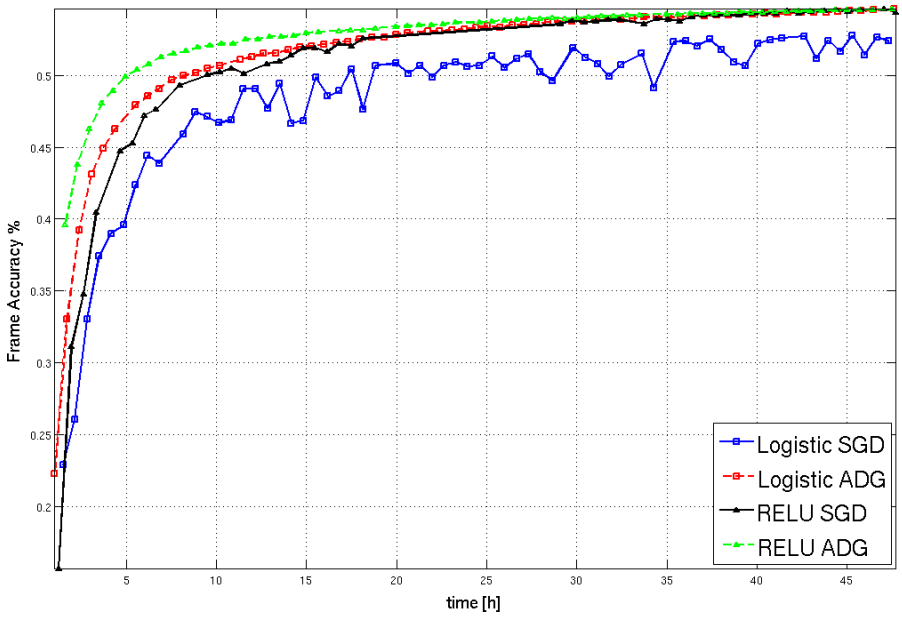

Figure 10.3: Frame accuracy as a function of time for a 4 hidden layer neural net trained with either logistic or ReLUs and using as optimizer either SGD or SGD with ADAGRAD (ADG).

strategies we described in sec. 10.3, on a 4 hidden layer HNN initialized at random. In terms of wall clock time to reach a given frame accuracy on the validation set, ADAGRAD exhibits the fastest convergence time, although plain SGD eventually reaches the same overall frame accuracy. Momentum instead performs slightly worse.

Similar findings were observed using a network with logistic units. However, plain SGD does not perform as well as ADAGRAD in this case, see fig. 10.3. Unlike HNN, a logistic network does need accelerated first order methods to yield good frame accuracy. It seems that optimization is much harder in logistic networks than HNNs. Fig. 10.3 shows that a logistic network trained with ADAGRAD can achieve the same accuracy as a HNN trained with either ADAGRAD or even plain SGD. However, we found that the performance in terms of word error rate is superior when using HNN and SGD. The word error rate of a logistic network trained with ADAGRAD is 11.8% (slightly better than when training using SGD), while the word error rate of HNN is 11.7 and 11.4% when using ADAGRAD and SGD, respectively. Since a difference of 0.1% is statistically

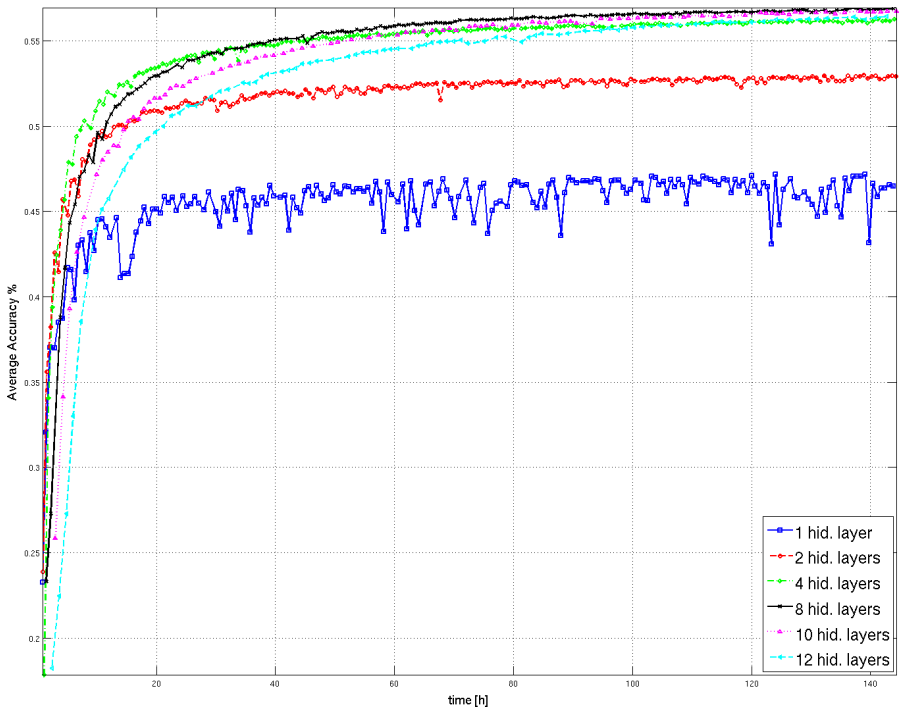

Figure 10.4: Validation frame accuracy over time using HNN with different number of hidden layers and SGD.

Nr. hid. layers	1	2	4	8	10	12
WER %	16.0	12.8	11.4	10.9	11.0	11.1

Table 10.1: Word error rate of HNN with varying number of hidden layers.

significant in our data set, we conclude that HNNs are not only easier to train but also they generalize better.

Finally, fig. 10.4 shows that extremely deep HNNs (we tested up to 12 hidden layers) can be successfully trained from random initialization. Since we did not allocate more resources for the deeper networks, their compute and convergence time is slower. However, they do not get stuck in the optimization and produce among the best results. Table 10.1 reports the corresponding word error rates on the test set. The 8 hidden layer neural network produces the best rate of 10.9%, but this is closely followed by the 10 and 12 hidden layer HNN.

We also tested very deep logistic networks from random initialization but did observe

that the optimization gets stuck when using 8 hidden layers and more. After one week, 8 hidden layers logistic network could only achieve a word error rate of 12.0%.

10.5 Discussion

In this empirical study we advocate the use of ReLU in deep networks since a) they are easier to optimize, b) they converge faster, c) they generalize better and d) they are faster to compute. This confirms the previous chapter results where good results where obtained in computer vision models using rectified linear units.

Chapter 11

ADADELTA: An Adaptive Learning Rate Method

11.1 Introduction

The most common method for training deep neural networks is stochastic gradient descent. This was used in all the previous chapters and typically can be applied to any differentiable objective function. However, it is somewhat unsatisfying that this simply first order method which requires a scalar hand-tuned learning rate is the only method that is widely used.

In this final chapter, we present a novel per-dimension learning rate method for gradient descent called ADADELTA. The method dynamically adapts over time using only first order information and has minimal computational overhead beyond vanilla stochastic gradient descent. The method requires no manual tuning of a learning rate and appears robust to noisy gradient information, different model architecture choices, various data modalities and selection of hyperparameters. We show promising results compared to other methods on the MNIST digit classification task using a single machine and on a large scale voice dataset in a distributed cluster environment.

Many machine learning algorithms can be framed as deriving updates to a set of parameters θ in order to optimize an objective function C. This often involves some iterative procedure which applies changes to the parameters, $\Delta\theta$ at each iteration of the algorithm. Denoting the parameters at the t-th iteration as θ_t, this simple update rule becomes:

$$\theta_{t+1} = \theta_t + \Delta\theta_t \tag{11.1}$$

In this chapter we consider gradient descent algorithms which attempt to optimize the objective function by following the steepest descent direction given by the negative of the gradient $\nabla_{\theta t}$. This general approach can be applied to update any parameters for which a derivative can be obtained:

$$\Delta\theta_t = -\eta\nabla_{\theta t} \tag{11.2}$$

where $\nabla_{\theta t}$ is the gradient of the parameters at the t-th iteration $\frac{\partial C(\theta_t)}{\partial \theta_t}$ and η is a learning rate which controls how large of a step to take in the direction of the negative gradient. Following this negative gradient for each new sample or batch of samples chosen from the dataset gives a local estimate of which direction minimizes the cost and is referred to as stochastic gradient descent (SGD) [104]. While often simple to derive the gradients for each parameter analytically, the gradient descent algorithm requires the learning rate hyperparameter to be chosen by hand.

Setting the learning rate typically involves a tuning procedure in which the highest possible learning rate is used. Choosing higher than this rate can cause the system to diverge in terms of the objective function, and choosing a lower rate results in slow learning. Determining a good learning rate becomes more of an art than a science for many problems. To further complicate the learning rate selection, it is often advantageous to use a large learning rate early on in training and a small learning rate near the end.

This work attempts to alleviate the task of choosing a learning rate by introducing a new dynamic learning rate that is computed on a per-dimension basis using only first order information. This requires a trivial amount of extra computation per iteration over gradient descent. Additionally, while there are some hyper parameters in this method, we have found their selection does not drastically alter the results. The benefits of this approach are as follows:

- no manual setting of a learning rate.
- insensitive to hyperparameters.
- separate dynamic learning rate per-dimension.
- minimal computation over gradient descent.
- robust to large gradients, noise and architecture choice.
- applicable in both local or distributed environments.

11.2 Other Optimization Techniques

There are many modifications to the gradient descent algorithm. The most powerful such modification is Newton's method which requires second order derivatives of the cost function:

$$\Delta\theta_t = H_t^{-1}\nabla_{\theta t} \tag{11.3}$$

where H_t^{-1} is the inverse of the Hessian matrix of second derivatives computed at iteration t. This determines the optimal step size to take for quadratic problems, but unfortunately is prohibitive to compute in practice for large models. Therefore, many additional approaches have been proposed to either improve the use of first order information or to approximate the second order information.

11.2.1 Learning Rate Annealing

There have been several attempts to use heuristics for estimating a good learning rate at each iteration of gradient descent. These either attempt to speed up learning when suitable or to slow down learning near a local minima. Here we consider the latter.

When gradient descent nears a minima in the cost surface, the parameter values can oscillate back and forth around the minima. One method to prevent this is to slow down the parameter updates by decreasing the learning rate. This can be done manually when the validation accuracy appears to plateau. Alternatively, learning rate schedules have been proposed [104] to automatically anneal the learning rate based on how many epochs through the data have been completed. These approaches typically add additional hyperparameters to control how quickly the learning rate decays.

11.2.2 Per-Dimension First Order Methods

The heuristic annealing procedure discussed above modifies a single global learning rate that applies to all dimensions of the parameters. Since each dimension of the parameter vector can relate to the overall cost in vastly different ways, a per-dimension learning rate that can compensate for these differences is often advantageous.

Momentum

One method of speeding up training per-dimension is the momentum method [105]. This is perhaps the simplest extension to SGD that has been successfully used for decades. The main idea behind momentum is to accelerate progress along dimensions in which gradients consistently have the same sign and to slow progress along dimensions where the sign of the gradients continues to change. In momentum this accomplished by keeping track of past parameter updates with an exponential decay:

$$\Delta\theta_t = \rho\Delta\theta_{t-1} - \eta\nabla_{\theta t} \tag{11.4}$$

where ρ is a constant controlling the decay of the previous parameter updates. This gives a nice intuitive improvement over SGD when optimizing difficult cost surfaces such as a long narrow valley. The gradients along the valley, despite being much smaller than the gradients across the valley, are typically in the same direction and thus the momentum term accumulates to speed up progress. In SGD the progress along the valley would be slow since the gradient magnitude is small and the fixed global learning rate shared by all dimensions cannot speed up progress. Choosing a higher learning rate for SGD may help but the dimension across the valley would then also make larger parameter updates which could lead to oscillations back as forth across the valley.

These oscillations are mitigated when using momentum because the sign of the gradient changes and thus the momentum term damps down these updates to slow progress across the valley. Again, this occurs per-dimension and therefore the progress along the valley is unaffected. Note however that this per-dimension adjustment is not a scaling of the learning rate, but is an additive term that can help with more complex cost functions. For example, an axis aligned valley in a cost function can be optimized well with a small learning rate down the valley and a high learning rate along the valley, but this cannot be done by scaling per-dimension learning rates when the valley is not axis-aligned. Momentum on the other hand is additive and thus can cope with such optimizations.

ADAGRAD

A recent first order method called ADAGRAD [30] has shown remarkably good results on large scale learning tasks in a distributed environment [27]. This method relies on only first order information but has some properties of second order methods and annealing. The update rule for ADAGRAD is as follows:

$$\Delta\theta_t = -\frac{\eta}{\sqrt{\sum_{\tau=1}^{t} \nabla_{\theta\tau}^2}} \nabla_{\theta t} \tag{11.5}$$

Here the denominator computes the ℓ_2 norm of all previous gradients on a per-dimension basis and η is a global learning rate shared by all dimensions.

While there is the hand tuned global learning rate, each dimension has its own dynamic rate. Since this dynamic rate grows with the inverse of the gradient magnitudes, large gradients have smaller learning rates and small gradients have large learning rates. This has the nice property, as in second order methods, that the progress along each dimension evens out over time. This is very beneficial for training deep neural networks since the scale of the gradients in each layer is often different by several orders of magnitude, so the optimal learning rate should take that into account. Additionally, this accumulation of gradient in the denominator has the same effects as annealing, reducing the learning rate over time.

Since the magnitudes of gradients are factored out in ADAGRAD, this method can be sensitive to initial conditions of the parameters and the corresponding gradients. If the initial gradients are large, the learning rates will be low for the remainder of training. This can be combatted by increasing the global learning rate, making the ADAGRAD method sensitive to the choice of learning rate. Also, due to the continual accumulation of squared gradients in the denominator, the learning rate will continue to decrease throughout training, eventually reaching to zero and stopping training completely. We created our ADADELTA method to overcome the sensitivity to the hyperparameter selection as well as to avoid the continual decay of the learning rates.

11.2.3 Methods Using Second Order Information

Whereas the above methods only utilized gradient and function evaluations in order to optimize the objective, second order methods such as Newton's method or quasi-

Newton methods make use of the Hessian matrix or approximations to it. While this provides additional curvature information useful for optimization, computing accurate second order information is often expensive.

Since computing the entire Hessian matrix of second derivatives is too computationally expensive for large models, Becker and LeCun [5] proposed a diagonal approximation to the Hessian. This diagonal approximation can be computed with one additional forward and back-propagation through the model, effectively doubling the computation over SGD. Once the diagonal of the Hessian is computed, $diag(H)$, the update rule becomes:

$$\Delta\theta_t = -\frac{1}{|\text{diag}(H_t)| + \mu} \nabla_{\theta t} \tag{11.6}$$

where the absolute value of this diagonal Hessian is used to ensure the negative gradient direction is always followed and μ is a small constant to improve the conditioning of the Hessian for regions of small curvature.

A recent method by Schaul *et al.* [112] incorporating the diagonal Hessian with ADAGRAD-like terms has been introduced to alleviate the need for hand specified learning rates. This method uses the following update rule:

$$\Delta\theta_t = -\frac{1}{|\text{diag}(H_t)|} \frac{E[\nabla_{\theta t-\tau:t}]^2}{E[\nabla_{\theta t-\tau:t}^2]} \nabla_{\theta t} \tag{11.7}$$

where $E[\nabla_{\theta t-\tau:t}]$ is the expected value of the previous τ gradients and $E[\nabla_{t-\tau:t}^2]$ is the expected value of squared gradients over the same window τ. Schaul *et al.* also introduce a heuristic for this window size τ (see [112] for more details).

11.3 ADADELTA Method

The idea presented in this chapter was derived from ADAGRAD [30] in order to improve upon the two main drawbacks of the method: 1) the continual decay of learning rates throughout training, and 2) the need for a manually selected global learning rate. After deriving our method we noticed several similarities to Schaul *et al.* [112], which will be compared to below.

In the ADAGRAD method the denominator accumulates the squared gradients from each iteration starting at the beginning of training. Since each term is positive, this accumulated sum continues to grow throughout training, effectively shrinking the learning rate on each dimension. After many iterations, this learning rate will become infinitesimally small.

11.3.1 Idea 1: Accumulate Over Window

Instead of accumulating the sum of squared gradients over all time, we restricted the window of past gradients that are accumulated to be some fixed size τ (instead of size t where t is the current iteration as in ADAGRAD). With this windowed accumulation the denominator of ADAGRAD cannot accumulate to infinity and instead becomes a local estimate using recent gradients. This ensures that learning continues to make progress even after many iterations of updates have been done.

Since storing τ previous squared gradients is inefficient, our method implements this accumulation as an exponentially decaying average of the squared gradients. Assume at time t this running average is $E[\nabla_\theta{}^2]_t$ then we compute:

$$E[\nabla_\theta{}^2]_t = \rho\, E[\nabla_\theta{}^2]_{t-1} + (1-\rho)\, \nabla_{\theta_t}^2 \tag{11.8}$$

where ρ is a decay constant similar to that used in the momentum method. Since we

require the square root of this quantity in the parameter updates, this effectively becomes the RMS of previous squared gradients up to time t:

$$\text{RMS}[\nabla_\theta]_t = \sqrt{E[\nabla_\theta{}^2]_t + \epsilon} \tag{11.9}$$

where a constant ϵ is added to better condition the denominator as in [5]. The resulting parameter update is then:

$$\Delta\theta_t = -\frac{\eta}{\text{RMS}[\nabla_\theta]_t} \nabla_{\theta t} \tag{11.10}$$

The formulation presented in Eq. 11.10 was recently proprosed by Hinton *et al.* [46] under the name of RMSPROP and will be compared to the ADADELTA formulation which combines this approach with the following idea.

11.3.2 Idea 2: Correct Units with Hessian Approximation

When considering the parameter updates, $\Delta\theta$, being applied to θ, the units should match. That is, if the parameter had some hypothetical units, the changes to the parameter should be changes in those units as well. When considering SGD, Momentum, or ADAGRAD, we can see that this is not the case. The units in SGD and Momentum relate to the gradient, not the parameter:

$$\text{units of } \Delta\theta \propto \text{units of } \nabla_\theta \propto \frac{\partial f}{\partial \theta} \propto \frac{1}{\text{units of } \theta} \tag{11.11}$$

assuming the cost function, f, is unit-less. ADAGRAD also does not have correct units since the update involves ratios of gradient quantities, hence the update is unit-less.

In contrast, second order methods such as Newton's method that use Hessian information or an approximation to the Hessian do have the correct units for the parameter updates:

Algorithm 4 Computing ADADELTA update at time t

Require: Decay rate ρ, Constant ϵ
Require: Initial parameter θ_1
1: Initialize accumulation variables $E[{\nabla_\theta}^2]_0 = 0$, $E[\Delta\theta^2]_0 = 0$
2: **for** $t = 1 : T$ **do** %% Loop over # of updates
3: Compute Gradient: $\nabla_{\theta t}$
4: Accumulate Gradient: $E[{\nabla_\theta}^2]_t = \rho E[{\nabla_\theta}^2]_{t-1} + (1-\rho){\nabla_\theta}_t^2$
5: Compute Update: $\Delta\theta_t = -\frac{\text{RMS}[\Delta\theta]_{t-1}}{\text{RMS}[\nabla_\theta]_t}\ \nabla_{\theta t}$
6: Accumulate Updates: $E[\Delta\theta^2]_t = \rho E[\Delta\theta^2]_{t-1} + (1-\rho)\Delta\theta_t^2$
7: Apply Update: $\theta_{t+1} = \theta_t + \Delta\theta_t$
8: **end for**

$$\Delta\theta \propto H^{-1}\nabla_\theta \propto \frac{\frac{\partial f}{\partial\theta}}{\frac{\partial^2 f}{\partial\theta^2}} \propto \text{units of } \theta \tag{11.12}$$

Noticing this mismatch of units we considered terms to add to Eq. 11.10 in order for the units of the update to match the units of the parameters. Since second order methods have correct units, we rearrange Newton's method (assuming a diagonal Hessian) for the inverse of the second derivative to determine the quantities involved:

$$\Delta\theta = \frac{\frac{\partial f}{\partial\theta}}{\frac{\partial^2 f}{\partial\theta^2}} \Rightarrow \frac{1}{\frac{\partial^2 f}{\partial\theta^2}} = \frac{\Delta\theta}{\frac{\partial f}{\partial\theta}} \tag{11.13}$$

Since the RMS of the previous gradients is already represented in the denominator in Eq. 11.10, a quantity related to the $\Delta\theta$ quantity should be present in the numerator. $\Delta\theta_t$ for the current time step is not known, so we assume the curvature is locally smooth and approximate $\Delta\theta_t$ by compute the exponentially decaying RMS over a window of size τ of previous $\Delta\theta$ to give the ADADELTA method:

$$\Delta\theta_t = -\frac{\text{RMS}[\Delta\theta]_{t-1}}{\text{RMS}[\nabla_\theta]_t}\ \nabla_{\theta t} \tag{11.14}$$

where the same constant ϵ is added to the numerator RMS as well. This constant serves

the purpose both to start off the first iteration where $\Delta\theta_0 = 0$ and to ensure progress continues to be made even if previous updates become small.

This derivation makes the assumption of diagonal curvature so that the second derivatives could easily be rearranged. Furthermore, this is an approximation to the diagonal Hessian using only RMS measures of ∇_θ and $\Delta\theta$. This approximation is always positive as in Becker and LeCun [5], ensuring the update direction follows the negative gradient at each step.

In Eq. 11.14 the $\text{RMS}[\Delta\theta]_{t-1}$ quantity lags behind the denominator by 1 time step, due to the recurrence relationship for $\Delta\theta_t$. An interesting side effect of this is that the system is robust to large sudden gradients which act to increase the denominator, reducing the effective learning rate at the current time step, before the numerator can react.

The method in Eq. 11.14 uses only first order information and has some properties from each of the discussed methods. The negative gradient direction for the current iteration $-\nabla_{\theta t}$ is always followed as in SGD. The numerator acts as an acceleration term, accumulating previous gradients over a window of time as in momentum. The denominator is related to ADAGRAD in that the squared gradient information per-dimension helps to even out the progress made in each dimension, but is computed over a window to ensure progress is made later in training. Finally, the method relates to Schaul *et al.* 's in that some approximation to the Hessian is made, but instead costs only one gradient computation per iteration by leveraging information from past updates. For the complete algorithm details see Algorithm 11.3.2.

11.4 Experiments

We evaluate our method on two tasks using several different neural network architectures. We train the neural networks using SGD, Momentum, ADAGRAD, RMSPROP, and ADADELTA in a supervised fashion to minimize the cross entropy objective between

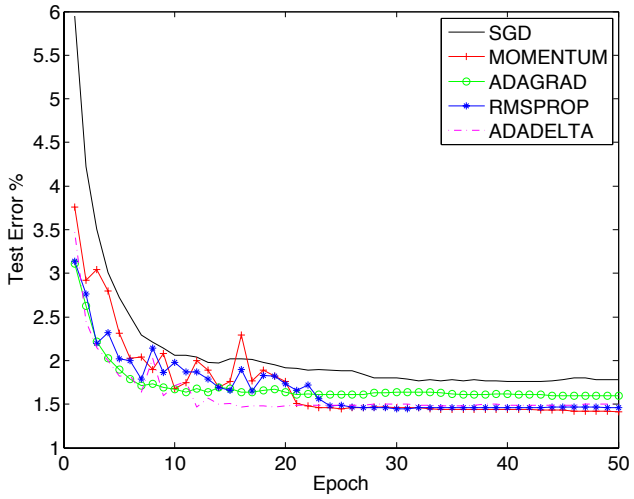

Figure 11.1: Comparison of learning rate methods on MNIST digit classification for 50 epochs.

the network output and ground truth labels. Comparisons are done both on a local computer and in a distributed compute cluster.

11.4.1 Handwritten Digit Classification

In our first set of experiments we train a neural network on the MNIST handwritten digit classification task. For comparison with Schaul's *et al.* method we trained with tanh nonlinearities and 500 hidden units in the first layer followed by 300 hidden units in the second layer, with the final softmax output layer on top. Our method was trained on mini-batches of 100 images per batch for 6 epochs through the training set. Setting the hyperparameters to $\epsilon = 1e-6$ and $\rho = 0.95$ we achieve 2.00% test set error compared to the 2.10% of Schaul *et al.* While this is nowhere near convergence it gives a sense of how quickly the algorithms can optimize the classification objective.

To further analyze various methods to convergence, we train the same neural network with 500 hidden units in the first layer, 300 hidden units in the second layer and rectified linear activation functions in both layers for 50 epochs. We notice that rectified linear units work better in practice than tanh as in the previous chapter, and their non-

saturating nature further tests each of the methods at coping with large variations of activations and gradients.

	SGD	MOMENTUM	ADAGRAD
$\eta = 1e^{0}$	**2.26%**	89.68%	43.76%
$\eta = 1e^{-1}$	2.51%	**2.03%**	2.82%
$\eta = 1e^{-2}$	7.02%	2.68%	**1.79%**
$\eta = 1e^{-3}$	17.01%	6.98%	5.21%
$\eta = 1e^{-4}$	58.10%	16.98%	12.59%

Table 11.1: MNIST test error rates after 6 epochs of training for various hyperparameter settings using SGD, MOMENTUM, and ADAGRAD.

	$\epsilon = 1e^{-2}$	$\epsilon = 1e^{-4}$	$\epsilon = 1e^{-6}$	$\epsilon = 1e^{-8}$
$\eta = 1e^{0}$	90.26%	90.26%	90.26%	90.26%
$\eta = 1e^{-1}$	2.33%	89.72%	81.46%	89.72%
$\eta = 1e^{-2}$	2.46%	1.99%	3.37%	3.70%
$\eta = 1e^{-3}$	5.39%	2.58%	**1.93%**	2.18%
$\eta = 1e^{-4}$	11.03%	5.57%	3.03%	2.78%

Table 11.2: MNIST test error rate after 6 epochs for various hyperparameter settings using RMSPROP. This uses a setting of $\rho = 0.95$ and varies the other two parameters, the lreatning rate η and constant in denomaintor ϵ so in practice more turning is needed for optimal RMSPROP performance.

	$\epsilon = 1e^{-2}$	$\epsilon = 1e^{-4}$	$\epsilon = 1e^{-6}$	$\epsilon = 1e^{-8}$
$\rho = 0.9$	2.59%	2.05%	1.90%	2.29%
$\rho = 0.95$	2.58%	1.99%	**1.83%**	2.13%
$\rho = 0.99$	2.32%	2.28%	2.05%	2.00%

Table 11.3: MNIST test error rate after 6 epochs for various hyperparameter settings using ADADELTA.

In Figure 11.1 we compare SGD, Momentum, ADAGRAD, RMSPROP, and ADADELTA in optimizing the test set errors. The unaltered SGD method does the worst in this case, whereas adding the momentum term to it significantly improves performance. ADAGRAD performs well for the first 10 epochs of training, after which it slows down due to the accumulations in the denominator which continually increase. ADADELTA matches the fast initial convergence of ADAGRAD while continuing to reduce the test error, converging near the best performance which occurs with momentum. Using only Idea 1 (ie. RMSPROP) performs well when the manual learning rate is optimally adjusted to $\eta = 1e^{-2}$, but converges to 2.5% error with a setting of $\eta = 1e^{-3}$, showing the utility of the numerator of ADADELTA to auto-tune this rate per dimension.

11.4.2 Sensitivity to Hyperparameters

While momentum converged to a better final solution than ADADELTA after many epochs of training, it was very sensitive to the learning rate selection, as was SGD and ADAGRAD. In Table 11.1 we vary the learning rates for each method and show the test set errors after 6 epochs of training using rectified linear units as the activation function. The optimal settings from each column were used to generate Figure 11.1. With SGD, Momentum, or ADAGRAD the learning rate needs to be set to the correct order of magnitude, above which the solutions typically diverge and below which the optimization proceeds slowly. RMSPROP needs a well tuned learning rate as well, plus two additional coefficients, the constant in the denominator η and the decay ρ for the denominator running average (see Table 11.2). We can see that these results are highly variable for each method, compared to ADADELTA in Table 11.3 in which the two hyperparameters do not significantly alter performance.

11.4.3 Effective Learning Rates

To investigate some of the properties of ADADELTA we plot in Figure 11.2 the step sizes and parameter updates of 10 randomly selected dimensions in each of the 3 weight matrices throughout training. There are several interesting things evident in this figure. First, the step sizes, or effective learning rates (all terms except $\nabla_{\theta t}$ from Eq. 11.14) shown in the left portion of the figure are larger for the lower layers of the network and much smaller for the top layer at the beginning of training. This property of ADADELTA helps to balance the fact that lower layers have smaller gradients due to the diminishing gradient problem in neural networks and thus should have larger learning rates.

Secondly, near the end of training these step sizes converge to 1. This is typically a high learning rate that would lead to divergence in most methods, however this convergence towards 1 only occurs near the end of training when the gradients and parameter updates are small. In this scenario, the ϵ constants in the numerator and denominator dominate

the past gradients and parameter updates, converging to the learning rate of 1.

This leads to the last interesting property of ADADELTA which is that when the step sizes become 1, the parameter updates (shown on the right of Figure 11.2) tend towards zero. This occurs smoothly for each of the weight matrices effectively operating as if an annealing schedule was present.

However, having no explicit annealing schedule imposed on the learning rate could be why momentum with the proper hyperparameters outperforms ADADELTA later in training as seen in Figure 11.1. With momentum, oscillations that can occur near a minima are smoothed out, whereas with ADADELTA these can accumulate in the numerator. An annealing schedule could possibly be added to the ADADELTA method to counteract this in future work.

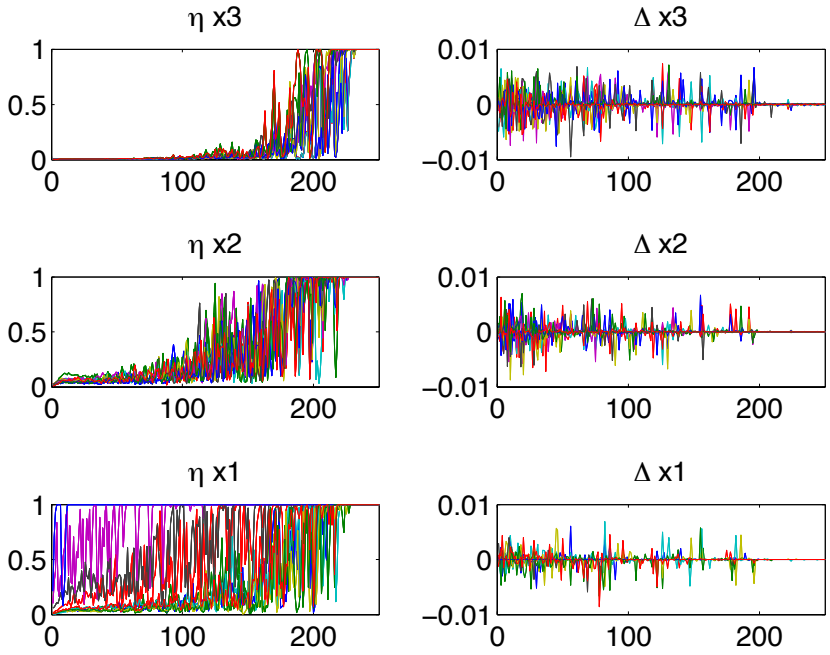

Figure 11.2: Step sizes and parameter updates shown every 60 batches during training the MNIST network with tanh nonlinearities for 25 epochs. Left: Step sizes for 10 randomly selected dimensions of each of the 3 weight matrices of the network. Right: Parameters changes for the same 10 dimensions for each of the 3 weight matrices. Note the large step sizes in lower layers that help compensate for vanishing gradients that occur with backpropagation.

11.4.4 Speech Data

In the next set of experiments we trained a large-scale neural network with 4 hidden layers on several hundred hours of US English data collected using Voice Search, Voice IME, and read data. The network was trained using the distributed system of [27] in which a centralized parameter server accumulates the gradient information reported back from several replicas of the neural network. In our experiments we used either 100 or 200 such replica networks to test the performance of ADADELTA in a highly distributed environment.

The neural network is setup as in [55] where the inputs are 26 frames of audio, each consisting of 40 log-energy filter bank outputs. The outputs of the network were 8,000 senone labels produced from a GMM-HMM system using forced alignment with the input frames. Each hidden layer of the neural network had 2560 hidden units and was trained with either logistic or rectified linear nonlinearities.

Figure 11.3 shows the performance of the ADADELTA method when using 100 network replicas. Notice our method initially converges faster and outperforms ADAGRAD throughout training in terms of frame classification accuracy on the test set. The same settings of $\epsilon = 1e^{-6}$ and $\rho = 0.95$ from the MNIST experiments were used for this setup.

When training with rectified linear units and using 200 model replicas we also used the same settings of hyperparameters (see Figure 11.4). Despite having 200 replicas which inherently introduces significant amount of noise to the gradient accumulations, the ADADELTA method performs well, quickly converging to the same frame accuracy as the other methods.

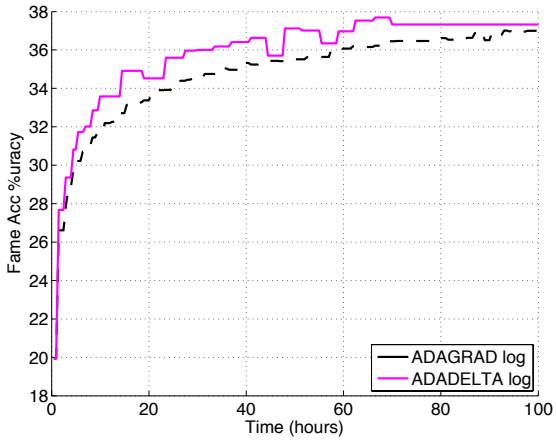

Figure 11.3: Comparison of ADAGRAD and ADADELTA on the Speech Dataset with 100 replicas using logistic nonlinearities.

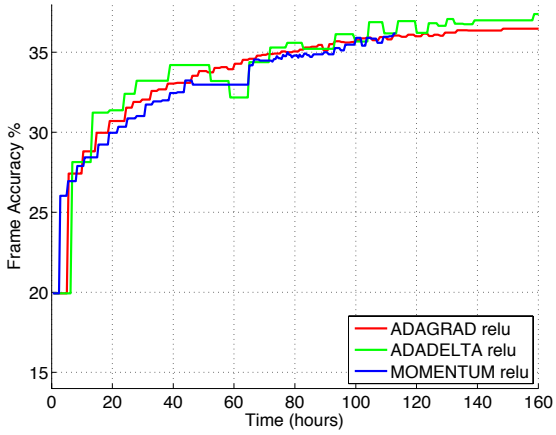

Figure 11.4: Comparison of ADAGRAD, Momentum, and ADADELTA on the Speech Dataset with 200 replicas using rectified linear nonlinearities.

11.5 Discussion

In this chapter we introduced a new learning rate method based on only first order information which shows promising result on MNIST and a large scale Speech recognition dataset. This method has trivial computational overhead compared to SGD while providing a per-dimension learning rate. Despite the wide variation of input data types, number of hidden units, nonlinearities and number of distributed replicas, the hyperparameters did not need to be tuned, showing that ADADELTA is a robust learning rate method that can be applied in a variety of situations.

Acknowledgments We thank Geoff Hinton, Yoram Singer, Ke Yang, Marc'Aurelio Ranzato and Jeff Dean for the helpful comments and discussions regarding this work.

Chapter 12

Part 2 Conclusion

12.1 Summary of Contributions

In this second half of this thesis we investigated three different areas of machine learning research. In Chapter 9 we introduced a novel method called the Input-Output Temporal Restricted Boltzmann Machine for mapping an input sequence to an output sequence. The IOTRBM is a generative model of the sequence conditioned on a few frames of previous outputs and the entire input sequence. We showed superior results to similar methods in the real world application of mapping motion capture of facial expressions between two subjects.

In Chapter 10 we moved on to the task of large vocabulary continuous speech recognition to determine how rectified linear units could improve performance in this challenging task. The results show that rectified linear units train faster and generalize better than competing activation functions such as the logistic function or tanh nonlinearities. This work formed the basis for improvements to the speech recognition system at Google that is now used in several production systems.

In the final chapter, Chapter 11, of this thesis we explore the optimization problem in

machine learning. Many algorithms rely on gradient descent with a single hand-tuned learning rate for optimization. Building on the success of the ADAGRAD method we designed a novel optimization technique which seems robust in many situations. This method is insensitive to the settings of its hyperparameters and converges well in both local and distributed computing environments. We believe it is a promising approach that could be applied to many machine learning problems.

12.2 Future Directions

Style Variables for Facial Expression Transfer: The IOTRBM presented in Chapter 9 was sufficient for transferring motions of one subject's face onto another when they were saying the same sentence with the same emotion. However, it would be interesting to decouple what is being said from the emotion by using additional style variables. Arbitrary emotions could be mapped between one another during retargeting by modifying the style variables. This could be useful for animating cartoon characters easily using human recordings that have generic emotions.

Integrating Speech Acoustic and Language Models: The speech recognition pipeline presented in Chapter 10 showed good improvements when rectified linear units were used. However, the overall objective of converting audio signals into predictions at the word-level is not directly optimized in this system because the neural network acoustic model and the HMM language model are not integrated. An interesting direction would be to combined both components into a single neural network with the objective to directly predict words from the audio signals themselves.

Improving ADADELTA Near Convergence: We showed fast convergence in Chapter 11 with ADADELTA, but we also found the final performance to be slightly behind that of the traditional momentum method. This could be due to lack of an annealing component in the algorithm, which is further exacerbated by accumulating only positive terms in the numerator. Further investigation is needed into the behaviour of

ADADELTA near convergence to determine why this method appears not to settle to good minima and if a momentum-like term could be added to improve performance.

Part III

Appendices

A Math Details

A.1 Convolution

A common operation throughout computer vision, machine learning, and especially this work is the convolution operation, typically denoted as $\otimes$. Convolution can be viewed as a mathematical operation between two inputs f and g that results in an output similar to one of the outputs being smoothed or modulated by the other. Convolution is directly related to correlation which is simply involves the operations, except one of the inputs in reversed. For example, correlation in 2D with a filter is the same as convolution with the filter being flipped up/down followed by flipped left/right (or rotated 180 degrees). This is evident from the formula for convolution in 2D where the filter f is of size $(2N_{F_{rows}} + 1) \times (2N_{F_{cols}} + 1)$ for simplicity:

$$(f \otimes g)[x, y] = \sum_{i=-N_{F_{rows}}}^{N_{F_{rows}}} \sum_{j=N_{F_{cols}}}^{N_{F_{cols}}} f[i, j]g[x - i, y - j] \tag{12.1}$$

and the formula correlation:

$$(f \otimes g)[x, y] = \sum_{i=-N_{F_{rows}}}^{N_{F_{rows}}} \sum_{j=N_{F_{cols}}}^{N_{F_{cols}}} f[i, j]g[x + i, y + j] \tag{12.2}$$

Correlation simply does the dot product of the filter centered at position $[x, y]$ in the image g. Convolution does this same dot product but with the 180 degree rotated filter. This equation can easily be understood with a figure (Figure 12.1, where f is a filter that is slid over an input image g.

An interesting property of convolution that is crucial to understanding the gradient descent operations in this work is that the gradient of a convolution operation is the correlation operation of the backpropagated errors. That is, it is the convolution of the backpropagated error signal with the 180 degree rotated filter. Conversely, the gradient

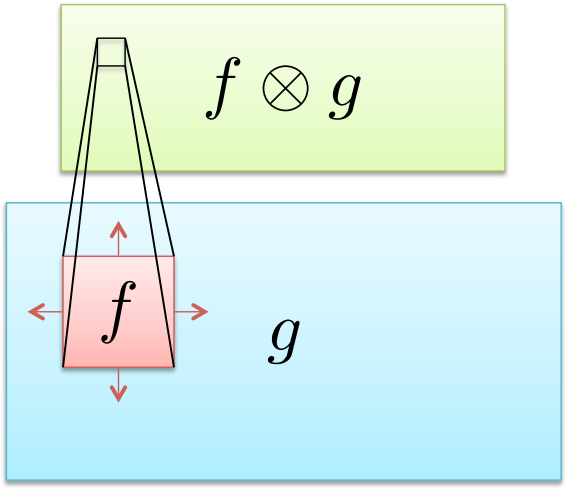

Figure 12.1: f is shown as a typical filter in a convolutional network and g is an image we want to convolve with this filter.

of a correlation is the convolution of the backpropagated errors.

One final thing to consider with convolutions is what to do at the boundaries of the image. There are three common types of convolutions to handle this in different ways. Valid convolutions choose the limits of g and f such that the filter overlap with the image never extends beyond the image boundaries, making the results smaller than the input image. Same convolutions allow g and y to range up to the size of the image, with half of the filter extending beyond the image boundary for edge cases, essentially extending the image by that amount with zeros and making the result the same size as the image. Finally full convolutions extend the boundary of the image with zeros such that only a single row, column, or corner pixel of the filter overlaps with the image making the result larger than the image. All three of these convolutional are used throughout this work in various ways and are related. For example, doing a valid convolution in one direction to get activations requires a full convolution during backpropagation to compute the proper gradients and vice versa. A same convolution simply uses a same convolution for gradients as well.

A.2 Mathematical Notation

The goal of choosing the notation used throughout this thesis was to have a unified view of different methods so that it is simple to see how they are related. Below is the notation used throughout the thesis for all methods split up into some high level categories for better presentation.

Variable	Description
x	input example of N_d dimensions
y	target or outputs
$\hat{y}$	predictions of targets y
$\mathcal{X}$	dataset of input examples
$\mathcal{Y}$	dataset of target values
N_c	number of classes in data
N_d	number of input dimensions
N_x	number of examples in dataset

Table 12.1: Dataset related variables.

Variable	Description
$N_{x_{rows}}$	number of rows of pixels in input x
$N_{x_{cols}}$	number of cols of pixels in input x
$N_{F_{rows}}$	number of rows of pixels in filters f
$N_{F_{cols}}$	number of cols of pixels in filters f
$N_{z_{rows}}$	number of rows of pixels in feature maps z
$N_{z_{cols}}$	number of cols of pixels in feature maps z
N_l	number of layers of a model
N_k	number of features in a layer, with ${N_k}^0$ being image colors
$N_{\mathcal{G}}$	number of pooling regions
$N_{i \in \mathcal{G}}$	number of elements in pooling region
g	connectivity matrix indicating connections between feature maps

Table 12.2: Architecture related variables.

Variable	Description
z	features before nonlinearity
ω	auxiliary features for optimization
σ	sigmoid or nonlinearity in general
h	features after nonlinearity
s	max pooling switch variables
p	pooled feature maps

Table 12.3: Intermediate variable names within a model, specific to an image.

Variable	Description
C	cost function to be optimized
E	energy function
N_e	number of epochs for training
N_t	number of optimization iterations per batch
t	step number for optimization
N_m	number of maxes used in reconstructions
∇	gradient of cost function, in some cases this is written as $\frac{\partial C}{\partial \theta}$
∇_z^L	gradient of cost function with respect to z from a L model.
H	the Hessian matrix of second derivatives
Δ	change in parameter applied as update during optimization
$N_{replicas}$	number of model replicas used to train with asynchronous SGD

Table 12.4: Optimization related concepts.

Variable	Description
P	probability
Z	partition function for probability distributions
b^x	RBM a bias variable for visible units
b^h	RBM a bias variable for hidden units
A	RBM matrix from previous visibles to current visibles
B	RBM matrix from previous visibles to current hiddens
q	IOTRBM source input sequence units
P	IOTRBM matrix from previous source visibles to current target visibles
Q	IOTRBM matrix from previous source visibles to current hiddens
τ	IOTRBM time window of previous inputs
$x_{<t}$	history of data at $t-1,\ldots,t-\tau$ as a vector
$x_{<=t}$	history of data at $t,\ldots,t-\tau$ as a vector

Table 12.5: RBM related concepts.

Variable	Description
P	pooling function. Differs from probability which is written as a function $P()$
U	unpooling function
loc	location with a pooling region
s	max pooling switch variables
$\mathcal{G}$	the group of elements in pooling region
v	the unnormalized Gaussian pooling weights
w	the normalized Gaussian pooling weights
μ_x	the mean pooling variable in the x dimension
μ_y	the mean pooling variable in the y dimension
γ_x	the variance pooling variable in the x dimension
γ_y	the variance pooling variable in the y dimension
sp_x	the spatial x location within a pooling region
sp_y	the spatial y location within a pooling region

Table 12.6: Pooling related variables.

Variable	Description
θ	notation for generic collection of parameters
W	weight matrix in sparse coding and full layers
b	bias in neural network
f	single plane filter weights in deconv and conv
F	all planes of filter weights in deconv and conv
W_{dec}	decoder weight matrix in PSD
W_{enc}	encoder weight matrix in PSD

Table 12.7: Learnable parameters names.

Variable	Description
im	index of image from dataset
i	general indexing variable
j	general indexing variable
c	indexing variable for color channels or number of classes depending on context
k	indexing variable for features
m	indexing variable for max reconstructions
n	indexing variable for pooling group normalizations
l	layer of the model (always a superscript)
L	top layer from which reconstruction begin

Table 12.8: Various indexing variables.

Variable	Description
η	learning rate
ρ	momentum coefficient or decay factor
λ	coefficient on sparse coding prior
β	coefficient used in continuation method

Table 12.9: Tunable hyper-parameters.

Variable	Description
$R^{L \to l}$	*reconstruction* operator form layer L to l
R^L	*reconstruction* operator form layer L to pixels
R^T	*projection* operator, backpropagates errors of the *reconstruction* operator
$\otimes$	convolution operator
$[\,]^T$	transpose operator, for filters this is `flipud(fliplr([ ]))`
$[\,]^l$	superscript refers to the layer. If not shown, refers to variable in general, ex. z
$[\,]_{i,j}$	multiple indexes can appear in subscripts
$[\,]$	no subscript refers to the entire set, such as $z^1 = \{z_1^1, \ldots, z_{N_k 1}^1\}$
$\tilde{}$	to differentiate the same index used for different loops
ℓ_α	generic norm notation, treating variables as vectors, also written as $\|\,\|^\alpha$
α	the exponent of an ℓ_α, $\|\,\|$ refers to $\alpha = 1$
ℓ_1/ℓ_2	sparsity inducing norm
ℓ_0	sparsity inducing norm
$\ell_{0.5}$	sparsity inducing norm
ℓ_1	sparsity inducing norm
ℓ_2	error norm or weight decay regularizer

Table 12.10: Math operators.

Variable	Description
ISTA	Iterative Shrinkage and Thresholding Algorithm
FISTA	Fast Iterative Shrinkage and Thresholding Algorithm
SPM	Spatial Pyramid Matching
SIFT	Scale-Invariant Feature Transform
HOG	Histogram of Oriented Gradients
SVM	Support Vector Machine
CNN	Convolutional Neural Network
DN	Deconvolutional Network
RBM	Restricted Boltzmann Machine
PSD	Predictive Sparse Decomposition
DBN	Deep Belief Network
HMM	Hidden Markov Model
IOHMM	Input-Output Hidden Markov Model
TRBM	Temporal Restricted Boltzmann Machine
IOTRBM	Input-Output Temporal Restricted Boltzmann Machine
FIOTRBM	Factored Input-Output Temporal Restricted Boltzmann Machine
CPM	Continuous Profile Model
LR	Linear Regression
AR	Autoregressive model
MLP	Multi-Layered Perceptron
IRLS	Iterative Reweighted Least Squares
LVCSR	Large Vocabulary Continuous Speech Recognition
HNN	Hinge Neural Networks

Table 12.11: Acronyms used throughout this work.

Part IV

Bibliography

Bibliography

[1] D. H. Ackley, G. E. Hinton, and T. Sejnowski. A learning algorithm for boltzmann machines. *Cognitive Sciences*, 9:147–162, 1985.

[2] Y. Amit and D. Geman. A computational model for visual selection. *Neural Computation*, 11(7):1691–1715, 1999.

[3] G. H. Bakir, T. Hofmann, B. Schölkopf, A. J. Smola, B. Taskar, and S. V. N. Vishwanathan. *Predicting Structured Data*. MIT Press, 2007.

[4] A. Beck and M. Teboulle. A fast iterative shrinkage-thresholding algorithm for linear inverse problems. *SIAM Journal on Imaging Sciences*, 2(1):183–202, 2009.

[5] S. Becker and Y. LeCun. Improving the convergence of back-propagation learning with second order methods. Technical report, Department of Computer Science, University of Toronto, Toronto, ON, Canada, 1988.

[6] Y. Bengio and P. Frasconi. An input/output HMM architecture. In G. Tesauro, D. S. Touretzky, and T. K. Leen, editors, *Proc. NIPS 7*, pages 427–434, 1995.

[7] Y. Bengio, P. Lamblin, D. Popovici, and H. Larochelle. Greedy layer-wise training of deep networks. In *NIPS*, pages 153–160, 2007.

[8] Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166, 1994.

[9] P. Berkes and L. Wiskott. On the analysis and interpretation of inhomogeneous quadratic forms as receptive fields. *Neural Computation*, 2006.

[10] L. Bo, X. Ren, and D. Fox. Multipath sparse coding using hierarchical matching pursuit. In *CVPR*, 2013.

[11] Y. Boureau, F. Bach, Y. LeCun, and J. Ponce. Learning mid-level features for recognition. In *CVPR*. IEEE, 2010.

[12] Y. Boureau, J. Ponce, and Y. LeCun. A theoretical analysis of feature pooling in vision algorithms. In *ICML*, 2010.

[13] H. Bourlard and Y. Kamp. Auto-association by multilayer perceptrons and singular value decomposition. *Biological cybernetics*, 59(4-5):291–294, 1988.

[14] H. Bristow, A. Eriksson, and S. Lucey. Fast convolutional sparse coding. In *CVPR*, 2013.

[15] M. Brown and D. G. Lowe. Automatic panoramic image stitching using invariant features. *International Journal of Computer Vision*, 74(1):59–73, 2007.

[16] M. Carreira-Perpinan and G. Hinton. On contrastive divergence learning. In *AISTATS*, pages 59–66, 2005.

[17] M. A. Carreira-Perpinan and G. E. Hinton. On contrastive divergence learning. In *Artificial Intelligence and Statistics*, volume 2005, page 17, 2005.

[18] A. Chambolle, R. A. De Vore, N.-Y. Lee, and B. J. Lucier. Nonlinear wavelet image processing: Variational problems, compression, and noise removal through wavelet shrinkage. *Image Processing, IEEE Transactions on*, 7(3):319–335, 1998.

[19] B. Chen, G. Sapiro, D. Dunson, and L. Carin. Deep learning with hierarichal convolutional factor analysis. *JMLR*, page Submitted, 2010.

[20] S. Chen, D. Donoho, and M. Saunders. Atomic decomposition by basis pursuit. *SIAM J. Sci Comp.*, 20(1):33–61, 1999.

[21] E. Chuang and C. Bregler. Performance driven facial animation using blendshape interpolation. Technical report, Stanford University, 2002.

[22] D. C. Ciresan, J. Meier, and J. Schmidhuber. Multi-column deep neural networks for image classification. In *CVPR*, 2012.

[23] D. C. Ciresan, U. Meier, J. Masci, L. M. Gambardella, and J. Schmidhuber. Flexible, high performance convolutional neural networks for image classification. In *IJCAI*, 2011.

[24] R. Collobert and J. Weston. A unified architecture for natural language processing: deep neural networks with multitask learning. In *ICML*, pages 160–167, 2008.

[25] A. Courville, J. Bergstra, and Y. Bengio. The spike and slab restricted boltzmann machine. In *AISTATS*, 2011.

[26] N. Dalal and B. Triggs. Histograms of oriented gradients for human detection. In *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, volume 1, pages 886–893. IEEE, 2005.

[27] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, Q. Le, M. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Ng. Large scale distributed deep networks. In *NIPS*, 2012.

[28] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.

[29] L. Deng, B. Hutchinson, and D. Yu. Parallel training of deep stacking networks. In *Interspeech*, 2012.

[30] J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online leaning and stochastic optimization. In *COLT*, 2010.

[31] D. Erhan, Y. Bengio, A. Courville, and P. Vincent. Visualizing higher-layer features of a deep network. In *Technical report, University of Montreal*, 2009.

[32] S. Eslami, N. Heess, and J. Winn. The shape boltzmann machine: a strong model of object shape. In *CVPR*, 2012.

[33] R. E. Fan, K. W. Chang, C. J. Hsieh, X. R. Wang, and C. J. Lin. Liblinear: A library for large linear classification. *Journal of Machine Learning Research*, 9:1871–1874, 2008.

[34] L. Fei-fei, R. Fergus, and P. Perona. One-shot learning of object categories. *IEEE Trans. PAMI*, 2006.

[35] R. Fergus, B. Singh, A. Hertzmann, S. T. Roweis, and W. Freeman. Removing camera shake from a single photograph. *ACM Transactions on Graphics, SIGGRAPH 2006 Conference Proceedings, Boston, MA*, 25:787–794, 2006.

[36] S. Fidler, M. Boben, and A. Leonardis. Similarity-based cross-layered hierarchical representation for object categorization. In *CVPR*, 2008.

[37] S. Fidler and A. Leonardis. Towards scalable representations of object categories: Learning a hierarchy of parts. In *CVPR*, 2007.

[38] Y. Freund and D. Haussler. Unsupervised learning of distributions of binary vectors using 2-layer networks. In *Proc. NIPS 4*, 1992.

[39] K. Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, 36:193202, 1980.

[40] D. Geman and Y. C. Nonlinear image recovery with half-quadratic regularization. *PAMI*, 4:932–946, 1995.

[41] I. Goodfellow, D. Warde-Farley, M. Mirza, A. Courville, and Y. Bengio. Maxout networks. *JMLR*, 3(28):1319–1327, 2013.

[42] G. Griffin, A. Holub, and P. Perona. The caltech 256. In *Caltech Technical Report*, 2006.

[43] C. E. Guo, S. C. Zhu, and Y. N. Wu. Primal sketch: Integrating texture and structure. *CVIU*, 106:5–19, 2007.

[44] G. Hinton, N. Srivastave, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv*, arXiv:1207.0580, 2012.

[45] G. E. Hinton. Training products of experts by minimizing contrastive divergence. *Neural Comput.*, 14(8):1771–1800, 2002.

[46] G. E. Hinton. Lecture 6e. In *Coursera course on Neural Networks for Machine Learning*, 2012.

[47] G. E. Hinton, A. Krizhevsky, and S. Wang. Transforming auto-encoders. In *ICANN-11*, 2011.

[48] G. E. Hinton, S. Osindero, and Y. W. Teh. A fast learning algorithm for deep belief nets. *Neural Comput.*, 18(7):1527–1554, 2006.

[49] G. E. Hinton, S. Osindero, and Y. The. A fast learning algorithm for deep belief nets. *Neural Computation*, 18:1527–1554, 2006.

[50] G. E. Hinton and R. R. Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, 2006.

[51] P. O. Hoyer. Non-negative sparse coding. pages 557–565, 2002.

[52] P. O. Hoyer. Modeling receptive fields with non-negative sparse coding. *Neurocomputing*, 52-54:547–552, 2008.

[53] E. Hsu, K. Pulli, and J. Popović. Style translation for human motion. *ACM Trans. Graph.*, 24(3):1082–1089, 2005.

[54] A. Hyvärinen, P. Hoyer, and E. Oja. Image denoising by sparse code shrinkage. In *Intelligent Signal Processing*. Citeseer, 1999.

[55] N. Jaitly, P. Nguyen, A. Senior, and V. Vanhoucke. Application of pretrained deep neural networks to large vocabulary speech recognition. In *Interspeech*, 2012.

[56] K. Jarrett, K. Kavukcuoglu, M. Ranzato, and Y. LeCun. What is the best multi-stage architecture for object recognition? In *ICCV*, 2009.

[57] Y. Jia and C. Huang. Beyond spatial pyramids: Receptive field learning for pooled image features. In *NIPS Workshops*, 2011.

[58] Y. Jia, C. Huang, and T. Darrell. Beyond spatial pyramids: Receptive field learning for pooled. In *CVPR*, 2012.

[59] Y. Jianchao, Y. Kai, G. Yihong, and H. Thomas. Linear spatial pyramid matching using sparse coding for image classification. In *CVPR*, 2009.

[60] Y. Jin and S. Geman. Context and hierarchy in a probabilistic image model. In *CVPR*, 2006.

[61] K. Kavukcuoglu, M. Ranzato, and Y. LeCun. Fast inference in sparse coding algorithms with applications to object recognition. In *Computational and Biological Learning Lab, Courant Institute, NYU*, 2008.

[62] K. Kavukcuoglu, P. Sermanet, Y. Boureau, K. Gregor, M. Mathieu, and Y. LeCun. Learning convolutional feature hierachies for visual recognition. In *NIPS*, 2010.

[63] B. Kingsbury, T. Sainath, and H. Soltau. Scalable minimum bayes risk training of deep neural network acoustic models using distributed hessian-free optimization. In *Interspeech*, 2012.

[64] D. Krishnan and R. Fergus. Analytic Hyper-Laplacian Priors for Fast Image Deconvolution. In *NIPS*, 2009.

[65] A. Krizhevsky. Learning multiple layers of featurs from tiny images. Technical Report TR-2009, University of Toronto, 2009.

[66] A. Krizhevsky. cuda-convnet. `http://code.google.com/p/cuda-convnet/`, 2012.

[67] A. Krizhevsky, I. Sutskever, and G. Hinton. Imagenet classification with deep convolutional neural networks. In *NIPS*, 2012.

[68] J. Lafferty, A. McCallum, and F. Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *Proc. ICML*, pages 282–289, 2001.

[69] H. Larochelle and Y. Bengio. Classification using discriminative restricted boltzmann machines. In *Proc. ICML*, pages 536–543, 2008.

[70] S. Lazebnik, C. Schmid, and J. Ponce. Beyond bags of features: Spatial pyramid matching for recognizing natural scene categories. In *CVPR*, 2006.

[71] Q. V. Le, J. Ngiam, Z. Chen, D. Chia, P. Koh, and A. Y. Ng. Tiled convolutional neural networks. In *NIPS*, 2010.

[72] Y. LeCun. The MNIST database. `http://yann.lecun.com/exdb/mnist/`, 2012.

[73] 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 Comput.*, 1(4):541–551, 1989.

[74] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *IEEE*, 86(11):2278–24, 1998.

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

[76] D. D. Lee and H. S. Seung. Learning the parts of objects by non-negative matrix factorization. *Nature*, 401(6755):788–791, 1999.

[77] H. Lee, A. Battle, R. Raina, and A. Y. Ng. Efficient sparse coding algorithms. In *NIPS*, pages 801–808, 2007.

[78] H. Lee, R. Grosse, R. Ranganath, and A. Y. Ng. Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations. In *ICML*, pages 609–616, 2009.

[79] J. Listgarten, R. Neal, S. Roweis, and A. Emili. Multiple alignment of continuous time series. In *Proc. NIPS 17*, 2005.

[80] D. G. Lowe. Object recognition from local scale-invariant features. In *Computer vision, 1999. The proceedings of the seventh IEEE international conference on*, volume 2, pages 1150–1157. Ieee, 1999.

[81] J. Mairal, F. Bach, J. Ponce, and G. Sapiro. Online dictionary learning for sparse coding. In *ICML*, 2009.

[82] J. Mairal, F. Bach, J. Ponce, G. Sapiro, and A. Zisserman. Supervised dictionary learning. In *NIPS*, 2008.

[83] D. Marr. Early processing of visual information. *Phil. Trans. R. Soc. Lond. B.*, 275:483–524, 1976.

[84] D. Marr. *Vision*. Freeman, San Francisco, 1982.

[85] J. Masci, U. Meier, D. Ciresan, and J. Schmidhuber. Stacked convolutional auto-encoders for hierarchical feature extraction. In *ICANN-11*, 2011.

[86] A. Mateo, A. Muñoz, and J. García-González. Modeling and forecasting electricity prices with input/output hidden Markov models. *IEEE Trans. on Power Systems*, 20(1):13–24, 1995.

[87] R. Memisevic and G. Hinton. Learning to represent spatial transformations with factored higher-order Boltzmann machines. *Neural Comput*, 22(6):1473–92, 2010.

[88] A. Mohamed, G. Dahl, and G. Hinton. Deep belief networks for phone recognition. NIPS 22 workshop on deep learning for speech recognition, 2009.

[89] G. Montavon, G. Orr, and K.-R. Muller, editors. *Neural Networks: Tricks of the Trade*. Springer, San Francisco, 2012.

[90] V. Nair and G. Hinton. Rectified linear units improve restricted boltzmann machines. In *ICML*, 2010.

[91] Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu, and A. Y. Ng. Reading digits in natural images with unsupervised feature learning. In *NIPS Workshop*, 2011.

[92] J. Ngiam, Z. Chen, P. Koh, and A. Ng. Learning deep energy models. In *ICML*, 2011.

[93] B. A. Olshausen and D. J. Field. Sparse coding with an overcomplete basis set: A strategy employed by V1? *Vision Research*, 37(23):3311–3325, 1997.

[94] F. Pighin and J. P. Lewis. Facial motion retargeting. In *ACM SIGGRAPH 2006 Courses*, SIGGRAPH '06, New York, NY, USA, 2006. ACM.

[95] C. Plahl, T. Sainath, B. Ramabhadran, and D. Nahamoo. Improved pre-training of deep belief networks using sparse encoding symmetric machines. In *ICASSP*, 2012.

[96] M. Ranzato, Y. Boureau, S. Chopra, and Y. LeCun. A unified energy-based framework for unsupervised learning. In *AISTATS*, 2007.

[97] M. Ranzato, Y. Boureau, and Y. LeCun. Sparse feature learning for deep belief networks. In *NIPS*. MIT Press, 2008.

[98] M. Ranzato and G. E. Hinton. Modeling pixel means and covariances using factorized Third-Order boltzmann machines. In *Proc. CVPR*, pages 2551–2558, 2010.

[99] M. Ranzato, F. Huang, Y. Boureau, and Y. LeCun. Unsupervised learning of invariant feature hierarchies with applications to object reocgnition. In *CVPR*, 2007.

[100] M. Ranzato, C. S. Poultney, S. Chopra, and Y. LeCun. Efficient learning of sparse representations with an energy-based model. In *NIPS*, pages 1137–1144, 2006.

[101] M. Riesenhuber and T. Poggio. Hierarchical models of object recognition in cortex. *Nature Neuroscience*, 2(11):1019–1025, 1999.

[102] R. Rigamonti, M. Brown, and V. Lepetit. Is sparsity really relevant for image classification? Technical Report 152499, Ecole Polytechnique Federale de Lausanne (EPFL), 2010.

[103] R. Rigamonti, M. Brown, and V. Lepetit. Are sparse representations really relevant for image classification? In *CVPR*, pages 1545–1552, 2011.

[104] H. Robinds and S. Monro. A stochastic approximation method. *Annals of Mathematical Statistics*, 22:400–407, 1951.

[105] D. Rumelhart, G. Hinton, and R. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986.

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

[107] T. Sainath, B. Kingsbury, and B. Ramabhadran. Auto-encoder bottleneck features using deep belief networks. In *ICASSP*, 2012.

[108] R. Salakhutdinov and G. Hinton. Deep boltzmann machines. In *AISTATS*, volume 5, pages 448–455, 2009.

[109] R. Salakhutdinov, A. Mnih, and G. E. Hinton. Restricted boltzmann machines for collaborative filtering. In *ICML*, pages 791–798, 2007.

[110] R. Salakhutdinov, J. Tenenbaum, and A. Torralba. Learning to learn with compound hierarchical-deep models advances in neural information processing systems 25. In *NIPS*, 2012.

[111] K. Sande, J. Uijlings, C. Snoek, and A. Smeulders. Hybrid coding for selective search. In *PASCAL VOC Classification Challenge 2012*, 2012.

[112] T. Schaul, S. Zhang, and Y. LeCun. No more pesky learning rates. arXiv:1206.1106, 2012.

[113] P. Sermanet, S. Chintala, and Y. LeCun. Convolutional neural networks applied to house numbers digit classification. In *ICPR*, 2012.

[114] T. Serre, L. Wolf, and T. Poggio. Object recognition with features inspired by visual cortex. In *CVPR*, 2005.

[115] J. R. Shewchuk. An introduction to the conjugate gradient method without the agonizing pain. *Neural Comput.*, 49(CS-94-125):64, 1994.

[116] D. Simard, P.Y. Steinkraus and J. Platt. Best practices for convolutional neural networks. In *ICDAR*, 2003.

[117] P. Simard, D. Steinkraus, and J. Platt. Best practices for convolutional neural networks applied to visual document analysis. In *ICDAR*, 2003.

[118] P. Smolensky. Information processing in dynamical systems: Foundations of harmony theory. In D. E. Rumelhart, J. L. McClelland, et al., editors, *Parallel Distributed Processing: Volume 1: Foundations*, pages 194–281. MIT Press, Cambridge, MA, 1986.

[119] P. Smolensky, D. E. Rumelhart, and J. L. McClelland. *Parallel Distributed Processing: Volume 1: Foundations*. MIT Press, 1986.

[120] J. Snoek, H. Larochelle, and R. P. Adams. Practical bayesian optimization of machine learning algorithms. In *NIPS*, 2012.

[121] K. Sohn, D. Jung, H. Lee, and A. Hero III. Efficient learning of sparse, distributed, convolutional feature representations for object recognition. In *ICCV*, 2011.

[122] J. Susskind, G. Hinton, J. Movellan, and A. Anderson. Generating facial expressions with deep belief nets. In *Affective Computing, Focus on Emotion Expression, Synthesis and Recognition*. I-TECH Education and Publishing, 2008.

[123] I. Sutskever and G. Hinton. Learning multilevel distributed representations for high-dimensional sequences. In *Proc. AISTATS*, 2007.

[124] I. Sutskever, G. E. Hinton, and G. Taylor. The recurrent temporal restricted boltzmann machine. In *NIPS*, 2008.

[125] K. Swersky, D. Tarlow, I. Sutskever, R. Salakhutdinov, R. S. Zemel, and R. P. Adams. Cardinality restricted boltzmann machines. In *NIPS*, 2012.

[126] G. Taylor and G. Hinton. Factored conditional restricted Boltzmann machines for modeling motion style. In *Proc. ICML*, pages 1025–1032, 2009.

[127] G. Taylor, L. Sigal, D. Fleet, and G. Hinton. Dynamical binary latent variable models for 3d human pose tracking. In *Proc. CVPR*, 2010.

[128] G. W. Taylor, G. E. Hinton, and S. Roweis. Modeling human motion using binary latent variables. In *NIPS*, pages 1345–1352, 2007.

[129] R. Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society*, 58, 1996.

[130] A. Torralba and A. A. Efros. Unbiased look at dataset bias. In *CVPR*, 2011.

[131] Z. W. Tu and S. C. Zhu. Parsing images into regions, curves, and curve groups. *IJCV*, 69(2):223–249, August 2006.

[132] `http://gp-you.org/`. GPUmat. `http://sourceforge.net/projects/gpumat/`, 2012.

[133] P. Vincent, H. Larochelle, Y. Bengio, and P. A. Manzagol. Extracting and composing robust features with denoising autoencoders. In *ICML*, pages 1096–1103, 2008.

[134] P. Vincent, H. Larochelle, I. Lajoie, Y. Bengio, and P.-A. Manzagol. Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. *The Journal of Machine Learning Research*, 11:3371–3408, 2010.

[135] D. Vlasic, M. Brand, H. Pfister, and J. Popović. Face transfer with multilinear models. In *ACM SIGGRAPH 2005*, pages 426–433, 2005.

[136] L. Wan, M. D. Zeiler, S. Zhang, Y. LeCun, and R. Fergus. Regularization of neural networks using dropconnect. In *ICML*, 2013.

[137] J. Wang, J. Yang, K. Yu, T. Huang, and Y. Gong. Locality-constrained linear coding for image classification. In *CVPR*, 2010.

[138] Y. Wang, J. Yang, W. Yin, and Y. Zhang. A new alternating minimization algorithm for total variation image reconstruction. *SIAM J. Imag. Sci.*, 1(3):248–272, 2008.

[139] S. Winder, G. Hua, and M. Brown. Picking the best daisy. In *CVPR*, 2009.

[140] S. Yan, J. Dong, Q. Chen, Z. Song, Y. Pan, W. Xia, Z. Huang, Y. Hua, and S. Shen. Generalized hierarchical matching for sub-category aware object classification. In *PASCAL VOC Classification Challenge 2012*, 2012.

[141] J. Yang, K. Yu, Y. Gong, and T. Huang. Linear spatial pyramid matching using sparse coding for image classification. In *CVPR*, 2009.

[142] M. Zeiler, D. Krishnan, G. Taylor, and R. Fergus. Deconvolutional networks. In *CVPR*, 2010.

[143] H. Zhang, A. C. Berg, M. Maire, and J. Malik. Svm-knn: Discriminative nearest neighbor classification for visual category recognition. In *CVPR*, 2006.

[144] L. Zhu, Y. Chen, and A. L. Yuille. Learning a hierarchical deformable template for rapid deformable object parsing. *PAMI*, March 2009.

[145] S. Zhu and D. Mumford. A stochastic grammar of images. *Foundations and Trends in Comp. Graphics and Vision*, 2(4):259–362, 2006.