VISUAL OBJECT RECOGNITION USING GENERATIVE MODELS OF IMAGES

by

Vinod Nair

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto

Abstract

Visual Object Recognition Using Generative Models of Images

Vinod Nair
Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto
2010

Visual object recognition is one of the key human capabilities that we would like machines to have. The problem is the following: given an image of an object (e.g. someone's face), predict its label (e.g. that person's name) from a set of possible object labels. The predominant approach to solving the recognition problem has been to learn a discriminative model, i.e. a model of the conditional probability $P(l|v)$ over possible object labels l given an image v.

Here we consider an alternative class of models, broadly referred to as *generative models*, that learns the latent structure of the image so as to explain how it was generated. This is in contrast to discriminative models, which dedicate their parameters exclusively to representing the conditional distribution $P(l|v)$. Making finer distinctions among generative models, we consider a supervised generative model of the joint distribution $P(v, l)$ over image-label pairs, an unsupervised generative model of the distribution $P(v)$ over images alone, and an unsupervised *reconstructive* model, which includes models such as autoencoders that can reconstruct a given image, but do not define a proper distribution over images. The goal of this thesis is to empirically demonstrate various ways of using these models for object recognition. Its main conclusion is that such models are not only useful for recognition, but can even outperform purely discriminative models on difficult recognition tasks.

We explore four types of applications of generative/reconstructive models for recognition: 1) incorporating complex domain knowledge into the learning by inverting a synthesis model, 2) using the latent image representations of generative/reconstructive models for recognition, 3) optimizing a hybrid generative-discriminative loss function, and 4) creating additional synthetic data for training more accurate discriminative models. Taken together, the results for these applications support the idea that generative/reconstructive models and unsupervised learning have a key role to play in building object recognition systems.

Acknowledgements

It was fun to watch Geoff Hinton's relentlessly creative mind come up with insights and ideas one after the other, without rest. Thanks to Geoff for generously sharing some of those ideas with me. I've yet to meet anyone more original in his thinking or with a livelier imagination. And he has shown me by example what it takes to do really *great* work – the commitment, focus and resourcefulness needed to be the best at whatever one does. I suspect that having seen this example will turn out to be by far the most useful lesson I got out of my PhD education.

Thanks to my committee members Rich Zemel, Allan Jepson and Yann LeCun for taking time off their busy schedules to read the thesis and give feedback. Their advice and suggestions improved the thesis a lot.

A big thank you to all the members of the UofT Machine Learning group, and in particular my office mates Andriy Mnih, Russ Salakhutdinov, Ilya Sutskever, Graham Taylor and Tijmen Tieleman, for all their help and encouragement. Special thanks to Josh Susskind for his collaboration on the face work presented in Chapter 3, as well as helping me prepare most of the figures in the thesis.

Finally, I thank my parents for their unconditional support and love. They helped me keep things in perspective throughout the PhD process, which let me get through it without going insane :-)

Contents

Chapter 1

Introduction

Among the human capabilities that we would like machines to have, the ability to recognize objects visually ranks high on the list. We can look around us and recognize familiar objects without much trouble. Getting a computer with a camera connected to it to do the same is the main goal of computer vision research. The holy grail is a general-purpose artificial vision system that can rapidly recognize instances of objects from a large catalogue of object types in a real-world scene.

Such a system does not currently exist, but if it did, it would have no trouble finding a wide variety of applications. Face recognition is perhaps the most well-known application, where the task is to identify a person from the image of his or her face. Here the 'objects' are the faces of different people currently known to the recognition system. Given the image of some scene as input, it has to decide whether any instances of the known faces appear within it. Other applications include better searching of image databases, robot navigation, improved image/video compression, automated surveillance, and so on. There are simply too many to list – by definition, any task that can make use of someone looking at a scene and identifying the objects in view is a potential application. Since machines are usually much cheaper than humans and do not get bored or tired, artificial systems will be much more widely useful than their human counterparts.

Despite their advantages, artificial vision systems are not yet in widespread use as they are far from matching the accuracy, speed and generality of human vision. Object recognition is hard because the appearance of the object can vary with lighting, viewpoint, changes in shape, occlusion etc. For example consider two images of the same person's face, one taken indoors and the other outdoors. If we examine the numerical values of the pixels in the two images, they can be very different. But an accurate object recognition system must still report the same face identity in both settings. More generally, the output of the system must be *invariant* to any changes in the input pixels that do not correspond to a change in object identity, while at the same time being sensitive to those that do. Solving the invariance problem has turned out to be extremely difficult for computers, even though our visual system seems to solve it without apparent effort.

The thesis will focus on the problem of visual object recognition. We consider a specialized version of the problem where an object has already been localized within a larger scene, and only its identity remains to be decided. So the input to the recognition system is a fixed-size image with the object roughly centred at a standard scale. There is a pre-specified set of object types from which the system chooses a label to assign to its input image. This formulation in effect reduces object recognition to image classification. The problem of localizing an object within a scene, i.e. *segmentation*, is also important and cannot be avoided when building a practical recognition system. But the core challenge of *visual invariance* still remains, so one can make useful progress just by concentrating on the image classification task in isolation.

Classification strictly requires only a model of the conditional probability of the object label l given an image v. The standard approach to recognition has been to define a parameterized *discriminative model* $P(l|v, \theta_d)$, where the parameter vector θ_d is estimated by maximum likelihood learning on a set of image-label pairs. Alternatively one can define a *generative model* $P(v, l|\theta_g)$ representing the joint distribution and estimate θ_g, again by maximizing likelihood. At test time, the conditional probability $P(l|v, \theta_g)$ is given by:

$$P(l|v, \theta_g) = \frac{P(v, l|\theta_g)}{\sum_{l'} P(v, l'|\theta_g)}. \tag{1.1}$$

It is not immediately obvious why the generative approach for estimating the conditional probability is useful. Fitting distributions to high-dimensional data (such as images) by maximum likelihood is analytically intractable for all but the simplest models. Discriminative models on the other hand can often be trained using the exact gradient of the log-likelihood. So the added difficulty of learning the joint distribution appears unnecessary at first glance.

Going a bit further, we can consider an unsupervised generative model that represents the distribution $P(v)$ over images alone, without the labels. It is even less clear what role such a model can play since the required conditional probability cannot be computed from it. Similarly, we can consider non-probabilistic counterparts like PCA and nonlinear autoencoders, which do not define a distribution over images but can be used to reconstruct a given input image. Such *reconstructive* models also cannot be used directly to compute the conditional probability and initially appears useless for recognition.

The main goal of this thesis is to *empirically demonstrate various ways of using generative and reconstructive models of images for object recognition*. The overall conclusion that comes out of it is that *these models are not only broadly useful for recognition, but can even outperform purely discriminative models on difficult recognition tasks*. This may sound counter-intuitive in the context of the above arguments, but the results presented in the upcoming chapters provide convincing support for it. The thesis also serves a secondary purpose as a collection of tricks for building better recognition systems using generative/reconstructive models. So even the pragmatic engineer with no ideological interest in generative models and unsupervised learning will hopefully find some tricks in here that are worth adding to his or her toolkit.

We first summarize the different applications of generative/reconstructive models demonstrated in the thesis. The applications are not all new or unique to the thesis, but we list them here to emphasize that these models are useful for recognition, and that they are useful in more than just one way.

1. **Incorporating complex domain knowledge into the learning:** Consider a graphics program that can synthesize realistic images of objects by manipulating a set of input variables. For example, it may be a physically-based model of handwritten digits with inputs like muscle stiffnesses, thickness of the pen, ink darkness etc. Given these inputs, the program internally simulates the physics of the arm trajectory to output a digit image. By using such a forward model in an analysis-by-synthesis loop, we can define a reconstructive representation of digits that exploits the knowledge built into the graphics program. Discriminative models allow limited forms of domain knowledge into the learning, e.g. sharing parameters in a convolutional network based on the 2D topology of images (LeCun et al. [1998]), and enforcing known invariances that the classification function should satisfy (Simard et al. [1996]). But much richer domain knowledge can be expressed naturally as a synthesis model, which can then be used for recognition by learning its 'inverse'.

2. **Recognition based on image representations learned by unsupervised models:** The hidden representation inferred by an unsupervised model compactly describes an image such that it can be reconstructed from that description. When used as part of a discriminative model, such a representation can give better recognition accuracy than the raw pixels. Unsupervised models are

less susceptible to overfitting than purely discriminative ones. There are two reasons for this: first, informally we can think of unsupervised learning as a large multi-task learning problem in which the model simultaneously predicts many variables (as many as there are inputs), while discriminative learning involves predicting just one (the label). Secondly, typical object recognition problems have many more unlabeled training cases available than labeled ones. Both of these factors allow unsupervised learning to put more constraints on the model parameters, and support more parameters and richer representations without overfitting.

One way to take advantage of this property is to rely on unsupervised learning to do most of the heavy-lifting when training an object recognition model, and use the small number of labeled cases for discriminative training in a more limited way. For example, unsupervised learning can be used to extract a large set of features, with the role of discriminative training limited to learning a low-capacity classifier on top of those features or to just fine-tuning those features discriminatively (instead of learning them from scratch). Deep Belief Nets (Hinton et al. [2006]) are a good example of this approach.

3. **Optimizing a hybrid generative-discriminative objective function:** As described earlier, a discriminative model is optimized to represent the conditional distribution $P(l|v)$, while a supervised generative model is optimized to represent $P(v, l)$. One can consider using a hybrid objective function that is a weighted sum of the above two objectives. A number of papers (e.g. Bouchard and Triggs [2004], Holub and Perona [2005], Raina et al. [2003]) have tried this idea and empirically shown it to give better accuracy than optimizing either one of the individual objectives in isolation. The results in the thesis lend more support to this idea.

   Although simply taking a weighted sum may seem ad hoc, Bishop and Lasserre [2007] have shown that it can be interpreted as a principled way of compensating for model mis-specification when learning a model of $P(v, l)$. From the point of view of discriminative learning, the advantage of the hybrid objective is that it allows unlabeled images to be included into the learning, which has a strong regularizing effect.

4. **Creating extra training images for discriminative models:** This is perhaps the simplest way to use generative/reconstructive models for object recognition. Take existing labeled training images, infer their hidden representations under an unsupervised model, randomly perturb those representations, and compute the images corresponding to them. The reasoning is that small random changes in feature space are likely to correspond to semantically meaningful changes in pixel space that do not change the class. The resulting synthetic images can then be used as additional examples for training a discriminative model with better accuracy.

1.1 Contributions of the thesis

As a whole, the thesis provides empirical support for the idea that generative models and unsupervised learning are useful for object recognition. This is its main contribution. The more specific ones are listed below:

1. We present a new algorithm called *breeder learning* for inverting a given synthesis model and obtaining its corresponding analysis model. The algorithm allows a pre-existing synthesis model to be used as part of an analysis-by-synthesis loop.

2. We describe a physically-based synthesis model for images of handwritten digits that simulates the actual hand-drawing process to generate the images. This model is inverted using breeder learning, and the resulting analysis-by-synthesis system is used for classifying digit images.

3. Maximum likelihood learning of a mixture model whose components are Restricted Boltzmann Machines is intractable because of the need to compute the probability of a data vector under a component RBM. We show how by defining the mixture model such that the mixing proportions are implicitly determined by the model parameters (rather than treating them as explicit parameters), it becomes tractable to learn a mixture of RBMs using Contrastive Divergence.

4. We introduce a new type of top-level model for Deep Belief Networks (DBNs) that allows separate sets of features to be learned for each object class.

5. We show how optimizing a hybrid generative-discriminative objective function avoids the poor Markov chain mixing that affects a top-level RBM model for a DBN. It gives better results than a purely generative top-level model on a difficult 3D object recognition task.

1.2 Outline of the thesis

We end this chapter with a short summary of each of the upcoming chapters:

1. **Chapter 2: Approaches to Learning Visual Invariance for Object Recognition** We give a brief review of some of the existing approaches to solving the invariance problem.

2. **Chapter 3: Analysis-by-Synthesis by Learning to Invert a Black Box Synthesis Model** This chapter introduces the breeder learning algorithm and shows how it can be used as part of an analysis-by-synthesis approach to modeling images. We present some proof-of-concept type results. The material in this chapter is based on Nair et al. [2008].

3. **Chapter 4: Inverting a Physically-Based Synthesis Model of Handwritten Digit Images** This chapter is a detailed application of breeder learning for a recognition task. We consider the problem of classifying handwritten digit images. The synthesis model simulates the physics of an arm trajectory drawing a digit. The corresponding analysis model is learned using breeder learning. Various ways of applying the learned model to the recognition task are considered. The material in this chapter comes from Hinton and Nair [2006].

4. **Chapter 5: An Implicit Mixture of Restricted Boltzmann Machines** This chapter shows how a seemingly intractable mixture of RBMs can be learned if we are willing to give up on having mixing proportions as explicit parameters. We also consider a simple way of using such a model for recognition. The material in this chapter first appeared in Nair and Hinton [2008].

5. **Chapter 6: 3D Object Recognition with Deep Belief Networks** This chapter presents a new top-level model for a DBN, as well as a hybrid generative-discriminative algorithm for training it. We apply the new model to a 3D object recognition task that requires invariance to pose, lighting, and intra-class shape variations. We also consider a semi-supervised version of the task and show how unlabeled data improves discriminative performance. The material in this chapter will appear in Nair and Hinton [2009].

Chapter 2

Approaches to Learning Visual Invariance

This chapter reviews various methods described in the literature for solving the visual invariance problem. The main ones considered are Lie group-based methods, slow feature analysis, multi-stage Hubel-Wiesel architecture, interest region descriptors, and higher order models, such as higher order Boltzmann machines and multilinear extensions of PCA and ICA.

2.1 Introduction

Learning invariant image representations is one of the most basic problems in computer vision. The human visual system has the ability to recognize common objects at different positions, scales, orientations, viewpoints, illuminations etc. Giving artificial vision systems similar capabilities is a difficult problem.

There are two types of learning problem we can consider regarding visual invariance. In the first type, a set of transformations is pre-specified based on prior knowledge, and the goal is to learn a model that is invariant to these known transformations of the image input. This is the problem that is addressed by the vast majority of papers on visual invariance. In many real-world applications there is much prior knowledge about which transformations the model should be invariant to. So learning algorithms that are designed to use pre-specified transformations are very useful in practice.

In the second type of problem, the invariant transformations are *not* pre-specified. Instead, transformation invariance of the learned model comes through generalization. This is a much harder problem, and fewer papers have attempted it (Hinton [1987], Memisevic and Hinton [2007]). For example, Memisevic and Hinton [2007] describe a model that learns to represent transformations between pairs of images without any built-in knowledge about specific transformations. This type of learning is more general and potentially more useful.

The rest of the chapter describes a number of learning approaches that have been proposed for solving the visual invariance problem.

2.2 Learning invariance to Lie transformation groups

Consider an image transformation t_α, parameterized by α. For example t_α could be translation on a pixel grid, where α is a 2D vector that specifies the number of pixels for horizontal and vertical translation. The set of transformations t_α corresponding to all possible values of α form a *group* if (1) the composition of any two transformations is another transformation in the same set (*closure*), (2) the composition order does not matter (*associativity*), (3) there is a unique *identity* transformation, and (4) every transformation in the set has a unique *inverse* transformation. Typical image transformations, such as

translation with wrap-around across image boundaries, satisfy the above conditions. For example, assuming wrap-around, two translations can be composed into another translation, the order in which two translations are applied does not matter, zero translation corresponds to identity, and every translation t_α can be undone by $t_{-\alpha}$.

A *Lie transformation group* satisfies the additional condition that t_α is differentiable with respect to α. This means that applying t_α ($\alpha \in \Re^m$) to an image $I \in \Re^n$ produces an m-dimensional (typically nonlinear) manifold in image space.

The main advantage of Lie transformation groups is that they provide a mathematically convenient way of modeling many image transformations that are of practical relevance. Since t_α is a continuous function of α, it becomes possible to consider infinitesimally small transformations. If we express the effect of t_α on I as a Taylor series, then an infinitesimal transformation can be represented by truncating all the terms in the series that are higher than first-order. The result is an image representation that is linear in α. Therefore, infinitesimally small transformations of I lie on an m-dimensional plane in image space, and this plane is tangent to the manifold of transformed images at I. As a result, in practice, Lie groups allow small discrete transformations to be modeled approximately using standard linear algebra.

Another advantage of Lie transformation groups is that, under certain assumptions, it is possible to derive a closed form expression for the entire Taylor series (i.e. without truncating any terms). The basic idea is to first define a model for infinitesimally small transformations. Given this model, arbitrarily large transformations can be created by composing infinitely many small transformations. Then a closed form expression for this infinite composition is derived based on the model for small transformations. So Lie groups can be used to define mathematically convenient models of arbitrarily large transformations as well.

Note that discrete image transformations (e.g. pixel-wise translation) do not form Lie groups because they do not satisfy the differentiability condition. But we can always convert a discrete pixel image into a smooth, differentiable mapping from $\Re^2$ (the 2D pixel coordinates) to $\Re$ (pixel values) by convolving it with a 2D Gaussian filter. Then it becomes sensible to consider infinitesimally small translations, rotations etc. applied to the continuous version of the original image. Some transformations are inherently non-smooth (e.g. reflection) and does not have an equivalent continuous version, so they do not form Lie groups as well.

An example of applying Lie groups to learning invariant image representations is the work by Simard et al. [1996]. They propose two different approaches for learning an image classifier that is approximately invariant to a set of pre-specified transformations, (1) *tangent distance* and (2) *tangent propagation*. Let $s(I, \alpha)$ denote the image generated by applying the transformation t_α to I ($I \in \Re^n$, $\alpha \in \Re^m$, as before). Also, let $\alpha = 0$ correspond to the identity transformation, i.e. $s(I, 0) = I$. Computing the Taylor series expansion of $s(I, \alpha)$ with respect to α and truncating all terms higher than first-order gives

$$s(I, \alpha) \approx I + J\alpha, \tag{2.1}$$

where $J = \frac{\partial s(I,\alpha)}{\partial \alpha}|_{\alpha=0}$ is the Jacobian matrix of s. (The expression for computing J depends on the set of invariant transformations that we pre-specify, and can be derived analytically for many common types of transformations, as shown in Simard et al. [1996].) This is the m-dimensional tangent plane that approximates the manifold in the vicinity of I.

The basic idea of tangent distance is the following: given two images I_1 and I_2, compute the tangent planes for both of them, and then let the tangent distance between the images to be the distance between their corresponding planes. Note that, by definition, any two images that lie on the same tangent plane will have zero tangent distance between them. This makes it approximately invariant to small transformations because images generated by such transformations would lie on (or very close to) the tangent

plane. The results on the MNIST classification task show that a nearest neighbour classifier using tangent distance significantly outperforms Euclidean distance. Examples of transformations that were used in this task are translation, rotation, scaling, and thickening.

Tangent propagation is a way of incorporating approximate transformation invariance into a classifier learned by gradient-based optimization. In addition to the usual cost function that is used to learn the classifier, an extra regularizing cost is added to enforce approximate invariance of the classifier to a set of pre-specified transformations. Suppose $G_w(I)$ is the function being learned (parameterized by the vector w) using a training set of N images $I_1, ..., I_N$. We want it to be approximately invariant to the transformation t_α of its input for small α. In other words, we want G_w to have approximately zero gradient along the directions in image space in which the transformations tend to push a training image. The extra cost E_r added to the usual training cost function is:

$$E_r = \sum_{i=1}^{N} \left| \frac{\partial G_w(s(I_i, \alpha))}{\partial \alpha} \right|^2_{\alpha=0}.$$

The main insight from tangent propagation is that it shows how the differentiability property of Lie transformation groups can be exploited to incorporate invariance in gradient-based learning methods. The alternative is to apply a large, discrete set of small transformations to the training images, include these synthetic images in the training set with the same label as the original images, and then train a classifier as usual. Tangent propagation achieves the same effect in a much more direct manner by taking advantage of the differentiability of the transformations.

Another way of using the tangent plane approximation to the continuous transformation manifold is proposed in Hinton et al. [1997]. They train a mixture of factor analyzers with an additional term in the cost function that penalizes poor reconstructions of tangent vectors at each training image. The result is that the model learns to reconstruct not only the training images, but also the transformed versions of those images produced by a set of pre-specified continuous transformations. So the overall model consists of a set of local, linear models that together form a global, nonlinear model of the image data.

Papers by Rao and Ruderman [1999], and Miao and Rao [2007], are other examples of learning visual invariance using Lie group theory. (The latter is mostly a longer version of the former.) Unlike Simard *et al.*, they do not truncate the Taylor series expansion of $s(I, \alpha)$ (as in equation 2.1). So their model can handle arbitrarily large transformations. But they assume that the gradient of the transformation $s(I, \alpha)$ (with respect to α) at an image I depends linearly on I. In other words, each column of the Jacobian matrix $J = \frac{\partial s(I,\alpha)}{\partial \alpha}|_{\alpha=0}$ is a linear function of I:

$$J = [G_1 I \ \mid \ G_2 I \ \mid \ ... \ \mid \ G_m I],$$

where G_i is an $n \times n$ matrix. If we substitute this assumption into the vector form of the Taylor series, we get:

$$s(I, \alpha) = \sum_{j=0}^{\infty} \left[\frac{1}{j!} (\left(\sum_{i=1}^{m} \alpha_i G_i \right)^j s(I, \alpha') \right]_{\alpha'=0} = \left(\sum_{j=0}^{\infty} \left[\frac{1}{j!} \left(\sum_{i=1}^{m} \alpha_i G_i \right)^j \right] \right) I, \tag{2.2}$$

$$= e^{\left(\sum_{i=1}^{m} \alpha_i G_i\right)} I. \tag{2.3}$$

So the linear gradient assumption results in a closed form expression for the Taylor series in terms of the matrix exponential. We can think of equation 2.3 as defining a generative model in which the latent representation consists of a "normalized" image I and a transformation α, and the model parameters are

the matrices G_i. Given particular values for I and α, the observed image is generated by transforming I according to α using $s(I, \alpha)$.

To learn this model, Rao and Ruderman [1999] note that for infinitesimally small values of α, we can truncate the higher order terms in the Taylor series. The resulting generative model is *bilinear* in I and α. (Bilinear models are discussed in more detail in section 2.9.) So

$$s(I, \alpha) \approx I + J\alpha = I + \left(\sum_{i=1}^{m} \alpha_i G_i \right) I = I + \left(\sum_{k=1}^{n} \sum_{i=1}^{m} \alpha_i I_k G_{ik} \right),$$

where α_i is the i^{th} component of α, I_k is the k^{th} pixel of I, and G_{ik} is the k^{th} column of the matrix G_i. They first learn this bilinear model using pairs of images that are related by an unknown, infinitesimally small, transformation. Once the matrices G_i are learned this way, they simply substitute them into equation 2.3 to obtain the model for arbitrarily large transformations. So the assumption of infinitesimal transformations is used only for training, just to learn the model parameters. Since the same parameters are used in the large-transformation model, there is no need for such an assumption at test time.

The observed image I' is assumed to be generated by $s(I, \alpha)$ and then corrupted by additive Gaussian noise with an identity covariance matrix. The maximum likelihood solution is given by minimizing the squared error between the observed image and its reconstruction by the model. The training images are generated by taking small natural image patches and applying small transformations to them (e.g. subpixel translations). At training time, the latent "normalized" image I is assumed to be given (it's the original, un-transformed patch), and α is inferred by gradient descent on the squared error between the artificially transformed image and its reconstruction by the model. With I and α both known, the model parameters G_i are updated by the negative gradient of the squared reconstruction error. At test time, I and α are inferred by coordinate descent on the squared error between the test image and its reconstruction by the large-transformation model in equation 2.3.

The later paper by Miao and Rao [2007] apply the above model to video sequences, where they assume that two consecutive frames undergo a transformation that is small enough for the bilinear model to hold approximately. This is a more realistic application than the previous one because the transformed images are not generated synthetically. However, the video sequence they use is fairly simple (a camera undergoing 1D translation above a toy town scene). Their results show that "temporal slowness" can be used to correctly learn a transformation model (at least for a simple sequence) without an explicit supervisory signal specifying what the transformations are.

2.3 Slow feature analysis

Temporal slowness is the idea that high level visual representations of a scene should vary more slowly over time than the low level sensory inputs. For example, as a person moves about a room, the identities of the (stationary) objects in the room should remain constant even though the retinal input is changing dramatically. In the case of Miao and Rao [2007], they assume that the latent image I stays constant across two consecutive frames, and the changes between the frames can be explained as a transformation of I. This idea was proposed by Hinton [1989] as a general principle for learning invariant visual representations without a teacher. An early example of an actual implementation of the idea is the paper by Foldiak [1991].

Here we look at a more recent paper by Wiskott and Sejnowski [2002]. They propose an algorithm for learning a slowly-varying latent representation of temporal signals. The representation is linear, and the cost function used for learning is the time-averaged magnitude of the latent representation's temporal derivative. In the following description, we use the same notation as in the original paper.

The input is a time-varying J-dimensional vector $\mathbf{x}(t)$. The goal is to learn a J-dimensional representation $\mathbf{y}(t)$ such that the temporal derivatives dy_j/dt (for $j \in 1, ..., J$) are small. $\mathbf{y}(t)$ is assumed to be a linear, instantaneous function of $\mathbf{x}$: i.e., $\mathbf{y}(t) = W\mathbf{x}(t)$, where W is a $J \times J$ matrix. The j^{th} column of W, $\mathbf{w}_j$, is computed such that it minimizes $\langle (dy_j/dt)^2 \rangle$ under the following constraints:

$$\langle y_j \rangle = 0,$$

$$\langle y_j^2 \rangle = 1,$$

$$\langle y_{j'} y_j \rangle = 0 \quad \forall j' < j.$$

(The symbol $\langle\rangle$ denotes time-averaging.) The first constraint is used only to simplify the expressions for the other two constraints and is not essential. The second constraint of unit variance excludes the trivial constant solution $y_j(t) = 0$. The third constraint of decorrelated components for $\mathbf{y}(\mathbf{t})$ excludes solutions where the components are simply replicas of each other.

Assume that the input $\mathbf{x}(t)$ is whitened (zero mean, identity covariance). Using the relationship $y_j = \mathbf{w}_j^T \mathbf{x}$ the above constraints can be re-written as

$$\langle y_j \rangle = \mathbf{w}_j^T \langle \mathbf{x} \rangle = 0,$$

$$\langle y_j^2 \rangle = \mathbf{w}_j^T \langle \mathbf{x}\mathbf{x}^T \rangle \mathbf{w}_j = \mathbf{w}_j^T \mathbf{w}_j = 1,$$

$$\langle y_{j'} y_j \rangle = \mathbf{w}_{j'}^T \langle \mathbf{x}\mathbf{x}^T \rangle \mathbf{w}_j = \mathbf{w}_{j'}^T \mathbf{w}_j = 0 \quad \forall j' < j.$$

These constraints specify that the matrix W must be orthogonal. In addition, its j^{th} column must minimize

$$\langle (dy_j/dt)^2 \rangle = \mathbf{w}_{j'}^T \langle \frac{d\mathbf{x}}{dt} \frac{d\mathbf{x}}{dt}^T \rangle \mathbf{w}_j.$$

Such a matrix W is given by the eigenvalue decomposition of $\langle \frac{d\mathbf{x}}{dt} \frac{d\mathbf{x}}{dt}^T \rangle$. (In practice the temporal derivative is approximated by finite differences.) So the columns of W are the eigenvectors of $\langle \frac{d\mathbf{x}}{dt} \frac{d\mathbf{x}}{dt}^T \rangle$. The components of $\mathbf{y}(t)$ can be ranked from slowest to fastest by sorting their corresponding eigenvalues from smallest to largest.

This particular formulation of slow feature analysis by Wiskott and Sejnowski does not maximize the likelihood of the training data. So it is not clear in what sense the learned representation is meaningful other than that it varies slowly in time. Slowness alone need not produce meaningful representations of the input signal. For example, simply lowpass filtering the input signal can produce an uninteresting slow representation. Such a solution is avoided in this formulation by forcing $\mathbf{y}(t)$ to depend only on the value of $\mathbf{x}$ for time t. Even with such restrictions, it is not obvious why the learned representation should be interesting.

Also, the above approach cannot be directly applied to a model with a learnable nonlinearity. (The authors mention using a *fixed* nonlinearity on the inputs, which of course is always possible.) The reason is based on an insight from methods that learn a nonlinear similarity metric between data vectors (Salakhutdinov and Hinton [2007]). One way to learn a similarity metric is to train a parametric mapping from the high dimensional data space to a low-dimensional output space in which Euclidean distance is semantically meaningful. Suppose the mapping is trained to maximize the mutual information between the outputs of two similar data vectors. Computing entropy exactly is expensive for multi-dimensional real-valued outputs. A tractable approximation is to assume Gaussian distributed outputs and compute the entropy as the log determinant of the Gaussian's covariance matrix. But this approximation allows a trivial solution in which a sufficiently flexible nonlinear mapping makes the individual output entropies arbitrarily large (by making the log determinant large), while keeping the joint output entropy not much

larger than any of the individual entropies, thus making the mutual information arbitrarily large. The approximation can be safely used only to learn a linear mapping, because a linear mapping cannot simultaneously achieve large individual entropies, an identity covariance matrix, and a joint entropy that is not much larger than any one individual entropy. Using a nonlinear mapping, it is possible to satisfy all three of these constraints. The same argument applies to the above formulation of slow feature analysis.

2.4 Models based on discretized transformations

If the set of transformations that a model should be invariant to is discrete and small, then a brute-force approach to invariance might be practical. For example, many face detection systems do an exhaustive search for faces in an image at a discrete set of scales and positions. These systems consist of a local, fixed-scale face detector which is applied to local patches in the image at all possible scale-position combinations. So the idea is to build a transformation-normalized face model and then add on invariance by exhaustively applying all possible transformations to the normalized model. This is a general method for achieving invariance to a discrete set of pre-specified transformations. It can be seen as a special case of the "normalize first and then recognize" strategy where the number of possible transformations is small, so it is feasible to run the recognizer on all possible normalizations.

The advantage of the above approach is that the transformations can be arbitrary – unlike in other approaches, they need not be continuous or small. The disadvantage is that if there are different types of transformations (translation, rotation, scaling etc.), then all possible combinations of them need to be considered, and so the set of transformations grows exponentially with the number of types. For n types of transformations, with m of each type, there are m^n possible combinations. Even a modest number of transformation types can be too expensive computationally.

Frey and Jojic [2000] present a set of models based on the above approach. They treat transformation as a discrete latent variable, and model the transformation-normalized image by (1) a Gaussian, (2) a mixture of Gaussians, (3) a factor analyzer, and (4) a mixture of factor analyzers (Ghahramani and Hinton [1996]). The pre-specified transformations are implemented by fixed permutation matrices (one matrix per transformation). So the observed image vector is related linearly to the transformation-normalized image vector. Once the discrete latent variables (the transformation and the mixture component) in a model are clamped, it reduces to a standard linear-Gaussian model. The discrete variables can be marginalized out by brute-force summation over all their possible settings. Assuming the summation is tractable, all four models can be trained unsupervised with EM, without any labels specifying which transformations are observed in the training images.

The *transformed Gaussian model* uses a Gaussian distribution over the transformation-normalized image and a multinomial distribution over the transformations. It tries to model the data as various transformed versions of a single prototype image. The *transformed mixture of Gaussians model* is an extension in which each mixture component is a transformed Gaussian model. So instead of one prototype, now there are multiple prototypes, each of which are transformed according to prototype-specific transformation probabilities. The *transformed component analysis* (TCA) model is the same as the transformed Gaussian model, except the distribution over the normalized image is given by a factor analyzer. The *mixture of TCA* (MTCA) model uses a TCA model as its component distribution. In all these models the normalized image's distribution is shared across all transformations, which significantly reduces the number of parameters compared to, say, a mixture model that has one component per prototype-transformation pair.

A modification of the MTCA model makes it possible to combine the local, tangent plane approximation to the continuous transformation manifold with the global, nonlinear approximation to the man-

ifold given by the discrete set of transformations. (This is assuming that all the transformations being considered are continuous.) The trick, suggested in Hinton et al. [1997], is to replace each factor loading matrix in MTCA with a matrix that contains the tangent vectors computed at the corresponding mixture component's mean. This fits a plane which approximates the transformation manifold locally at the component's mean. The overall model then consists of a set of locally linear models that together form a nonlinear model of the transformation manifold.

2.5 Invariance in face detectors

Almost all recent face detectors described in the literature are translation and scale invariant. A standard approach is to first train a classifier to discriminate between faces and non-faces. The face patches used to train this classifier are pre-segmented and normalized (e.g., the eye locations are forced to have the same pixel coordinates across all face patches). So the classifier itself is not explicitly designed to have translation and scale invariance with respect to its input. At test time, the input to the face detector is an image that contains an unknown number of faces in it. The faces (if any) are detected by independently applying the binary classifier to local windows in the image at all possible locations and scales. When a local window matches the position and the scale of a face in the image, the binary classifier outputs the face class label, and the window is then taken to be a detected face. The search over locations and scales must be done over a sufficient range of values and with enough resolution to make sure that none of the legitimate faces in the input image are missed.

The same brute-force approach can potentially be applied to achieve invariance to facial pose. The binary classifier can, in principle, be trained on faces in all possible poses. But, as explained in Jones and Viola [2003], in practice it is difficult to train such a single classifier accurately. For some non-frontal poses (e.g. profile views), the face image will contain a significant amount of background. So the problem of segmentation becomes much more important than in the case of only frontal poses. Also, standard methods for normalizing the training faces (such as placing the eyes at fixed pixel coordinates) are no longer meaningful when all poses are considered together.

Rowley et al. [1997] build a rotation-invariant face detector using a classifier trained on faces in the frontal, up-right pose. The system can only deal with in-plane rotations of faces, which unfortunately rules out a large number of poses that produce out-of-plane rotations. Their detector consists of a neural network that is trained to estimate the rotation angle of a face. They first apply this network to a local image window. The estimated angle is used to rotate the local image patch so that the (presumed) face becomes up-right. The binary classifier is then be applied to the rotation-normalized face. In the case of a non-face patch, the rotation estimator will output a meaningless angle, and rotating the patch by that angle is unlikely to hurt the subsequent classification step.

Jones and Viola [2003] propose a similar system, but instead of training a single classifier on up-right faces, they discretize the entire range of orientations into a small number of classes and train one classifier per orientation class. A decision tree classifier is used to assign an image patch to one of these orientation classes. Although the underlying task is regression, they treat it as classification. Then the face/non-face classifier trained for that orientation class is used to decide whether the patch is a face. The drawback of the general detection strategy adopted by both Rowley et al. [1997] and Jones and Viola [2003] is that if the orientation estimator fails, then the face detector will fail as well. So the detector can only be as good as the orientation estimator itself.

The original Viola-Jones face detector (Viola and Jones [2001]) has only scale and position invariances, but they are implemented in an interesting way. The face/non-face classifier has a cascade structure, with each stage of the cascade being a stricter test for a face than the earlier stages. A stage can reject an image patch as a non-face, in which case no further processing is done on that patch. But

a patch which is labeled as a face by one stage is passed on to the next stage for further checking. Only those patches that pass through all the stages are reported as detected faces. So only the very first stage of the classifier is applied exhaustively at all scale-position combinations. The first stage contains a small number of low-level features, making it somewhat like a *learned* low-level interest region detector for faces. The results show that, in practice, even a simple, fast test in the first stage can rule out vast regions of the input image as non-face. Therefore most of the work for scale and position invariance is done through low-level processing that is fast and efficient.

Osadchy, Miller and LeCun (Osadchy et al. [2004]) present another approach to pose-invariant face detection. They train a convolutional neural network to simultaneously perform face/non-face classification and 3D pose estimation. Instead of representing the pose with 3 numbers, they overparameterize it with a 9D representation. (The mapping from the pose to the 9D representation is specified by hand.) Since specifying the pose requires only three degrees of freedom, the valid poses form a 3D manifold in the 9D output space. A convolutional neural network is trained to take a face image patch as input and compute its corresponding correct 9D pose representation as output, while for a non-face patch, the network is trained to compute a 9D vector that is far away from the 3D manifold of valid poses. At test time, a patch is classified by first inferring its 9D pose representation using the convolutional network. This 9D vector is then projected on to the 3D manifold of valid poses. If the distance between the original 9D vector and its projection is greater than a threshold, then the patch is classified as non-face. Otherwise it is classified as a face with the pose given by the projection. The results show that training the network to do the two tasks simultaneously results in better accuracy for both of them than independently training a separate network for each task. The trick here is to train a face/non-face classifier with labels that are much more informative than binary labels. The extra information provided by the pose label results in a much better classifier.

2.6 The Multi-stage Hubel-Wiesel Architecture

This is a class of models for invariant pattern recognition that is based on the following approach:
(1) Replicate local image features by applying various amounts of whatever transformation the recognition needs to be invariant to. For example, to achieve rotational invariance, a local feature is replicated by applying various amounts of rotation to it.
(2) Pool the activations of the replicas over a small range of transformations into a single activation that is invariant to which of the replicas in the pool is active. For example, a pool is considered active if any one of the rotated versions of a feature in that pool is active. So its activation is invariant to rotation. Then learn features on these pooled activations to produce 'features of features'.
(3) Gradually build up invariance by repeating the above two steps over many layers of features.
The above strategy is inspired by experimental results from neuroscience about the mammalian visual cortex. Position and scale invariances seem to be built up hierarchically in the ventral stream using transformed replicas of features, and the receptive field size tends to increase with the hierarchy level. The name *multi-stage Hubel-Wiesel architecture* was suggested in Ranzato et al. [2007]. Examples of models that have this architecture are convolutional neural networks (LeCun et al. [1998]), Neocognitron (Fukushima [2007]) and HMAX (Riesenhuber and Poggio [1999]). They differ mostly in the details of how the three steps are implemented.

Convolutional neural networks learn local features that are replicated across all possible pixel positions via convolution. Features over a small local neighbourhood are pooled together by an average. This makes a second level feature invariant to changes in the positions of the first level features within its own pool, because the averaging operation is invariant to permutations of its inputs. Similarly, the second level features are replicated by convolution and pooled to form the inputs to the third level features, and

so on. As more levels are added, the features become increasingly position invariant. A fully connected network is usually applied on the activations of the highest-level convolutional features to compute the final output of the network (e.g. a class label), and the entire hierarchy is trained simultaneously, end-to-end. A feedforward pass through a convolutional network is computationally efficient because image convolution is a fast operation in modern CPUs. This makes translation invariance relatively cheap to implement. Incorporating other types of invariances can be more expensive: for example, rotational and scale invariance can be included by first generating various rotated and scaled versions of the input image and then applying the convolutional filters to them. But the large number of combinations of transformations makes this approach expensive.

HMAX[1] is meant to be a model for theoretical neuroscientists studying object recognition in the human visual cortex. The pooling of feature activations is done by a max operation. So among all the feature activations in a pool, the largest one is picked to be the pool's activation. Also, learning is done only at the highest level of the network, and the lower level features are hand-coded. The features in the lowest level are oriented bar and edge detectors. The learned component of the architecture is a Gaussian Radial Basis Function network that simply stores the top-level feature activations for all the training images and trains a linear network from the feature activations to the desired output (e.g. the pose of an object) by supervised learning. The idea is that the same, fixed, set of feature layers are used for many different visual inference tasks, and task-specific learning occurs only at a high level, to convert the top-level feature activations into desired outputs.

One limitation of the above models is that they are limited to producing invariance to only small, local transformations in the image because allowing larger invariances will reduce the discriminative power of the features. Two different objects containing the same feature but at different positions cannot be discriminated by a fully translation-invariant detector for that feature. One way to get around this limitation is to explicitly model not just the "what" information, but the "where" information as well.

It is easy to explicitly retain the "where" information in the HMAX model simply by storing which feature in a pool produced the maximum activation. Ranzato et al. [2007] incorporate this idea into learning an autoencoder-type network with convolutional kernels that are essentially "movable parts" of objects. The trick is to learn the kernels by backpropagating gradients only from the image locations where they fired most strongly, rather than from all locations in the image (as would be done in a standard convolutional network). This can be done by setting the feature map of a kernel (i.e. the output of convolving the kernel with the image) to zero everywhere except at the location of the maximum activation. The kernel parameters are then updated using gradients computed only for that location. So if two different objects have the same local feature ("part") but at different positions, the model can learn a single feature for that part and still discriminate between the two objects by its location of maximum activation.

The learning algorithm used in Ranzato et al. [2007] is an extension of an earlier unsupervised learning algorithm for training an encoder-decoder architecture (Ranzato et al. [2006]). As in the earlier algorithm, the main steps are: (1) infer a hidden representation, or code, for the input image by forward propagation through the encoder, (2) optimize the code with respect to reconstruction error by backpropagation through the decoder, (3) update the encoder parameters with the optimized code as its target output, and (4) update the decoder parameters with the optimized code as its input and the image as the target output. The new algorithm uses modified versions of the same steps. For an encoder with a convolutional architecture, the code now includes the pixel coordinates of the maximum activation location for each kernel. This extended part of the code is kept fixed in step 2. The feature maps are set to zero at all locations except those corresponding to the maximum activation (one location per feature map). Steps 3 and 4 remain unchanged. Results in Ranzato et al. [2007] show that applying this

[1] HMAX stands for "Hierarchical Model and X" (http://riesenhuberlab.neuro.georgetown.edu/hmax.html)

algorithm to MNIST produces kernels that resemble "movable parts" for handwritten digit images.

The strategy used in this paper to make the features translation-invariant is practical only because there is a simple method for estimating the position of a local feature within an image. But it is not clear how this approach can be extended to other types of transformations, such as rotation. Also, the assumption here is that a particular part/feature can appear only once in an image, because only a single location in the feature map is kept. This seems like a strong assumption.

2.7 Local interest region detectors and descriptors

An important class of algorithms for invariant visual recognition is based on the idea of first detecting a set of local regions in an image that are "interesting" in some sense[2], and then computing descriptions of those regions. Both the *detection* and *description* steps are designed to be approximately invariant to common image transformations (e.g. affine transformations). The hope is that the same descriptors can be reliably re-computed from a transformed version of the original image so that the two images can be matched even under large transformations. A well-known example of this type of algorithm is SIFT (Scale Invariant Feature Transform) by Lowe [2003].

What makes the local descriptor approach different from other approaches to invariance is that it is purely low level. The algorithms for detecting and describing the local regions have no built-in notion of objects or any such high level image representations. Instead, they are simply trying to find image patches that are highly re-detectable under various transformations that are of interest in typical vision applications (e.g. affine transformations, illumination changes etc.). The generic, low level nature of these algorithms makes them a useful front end for a wide variety of applications, such as object recognition, image retrieval, and visual SLAM (simultaneous localization and mapping).

A large number of algorithms have been proposed for both region detection and description. A review and performance evaluation of these methods is presented in Mikolajczyk and Schmid [2005]. It considers five region detection methods and ten region descriptors. The region detectors are all based on similar ideas and differ only in the computational details, so only the particular method used by SIFT is described here. (Not surprisingly, one of the conclusions of Mikolajczyk and Schmid [2005] is that the choice of the region detector does not significantly affect a descriptor's performance.) There is more variety in the ideas for region descriptors, although many of the best performers are essentially variants of the one used by SIFT (e.g. shape context (Belongie et al. [2002]), and gradient location and orientation histogram, or GLOH (Mikolajczyk and Schmid [2005])).

SIFT's region detector aims to find patches in the input image that are likely to be re-detectable across scale changes of the image. It performs a brute-force search over all possible positions in various scaled versions of the image. At each scale, the response of a difference-of-Gaussian (DoG) filter is computed for all image positions. A scale-stable region is taken to be centred at a pixel location that produces a local extremum in the DoG responses across scales. The intuition is this: consider an idealized image containing a white circle on a black background. A DoG filter that starts off with a small receptive field and gradually gets scaled up will produce its maximum (or minimum, if pixel colours are flipped) response when the on-centre part of its receptive field fully fits the circle, regardless of the scale of the circle. Therefore the circle can be detected at any scale by locating an extremum in the filter response as a function of scale. In real images, this criterion tends to find blob-like homogenous regions. Note that the detector is invariant to scaling and rotation of the image, but not to more general affine transformations. Affine-invariant alternatives have been proposed (Mikolajczyk and Schmid [2005]), but in practice, the invariances in the region descriptor can compensate for the detector's lack of affine

[2] "Interesting" typically means that it is possible to extract a description of the region that is stable over a range of viewing conditions.

invariance (Lowe [2003]).

Once the scale-stable image locations are identified, descriptors are computed for the pixel neighbourhoods centred on those locations. Each location has a scale associated with it. The descriptor for a location is computed using a 16×16 pixel patch from the original image rescaled by the location's scale. So a 16×16 neighbourhood may correspond to a much larger neighbourhood in the original image. The region is then assigned an orientation by computing a histogram of local image gradient orientations (discretized into 36 bins) from the 16×16 patch. The bin with the most mass is selected to be the region's orientation. If many bins have significant mass, then multiple regions are created at the same location, but each with a different orientation.

So each region has its own location, scale and orientation, which together define a region-specific 2D coordinate system. The region's descriptor is computed with respect to this coordinate system. As long as a region's axes are estimated correctly from a transformed version of the original image, its descriptor will remain the same. This is in effect the low level version of the object recognition approach that first estimates an object-centric reference frame and then describes the object relative to that reference frame.

The descriptor consists of histograms of gradient orientations computed from the 16×16 patch. Gradient orientation is discretized into 8 bins. The 16×16 patch is divided into 16 non-overlapping, smaller patches, each of which is 4×4. One gradient orientation histogram (8 numbers) is computed per smaller patch to produce a $16 * 8 = 128$ dimensional, real-valued descriptor for the region. The histogram-based representation provides invariance to the specific spatial configuration of the gradients. The 128-D feature vector is normalized to unit length for approximate illumination invariance.

One approach to object recognition using SIFT is based on nearest neighbour matching of descriptors. Given a test image containing an unknown set of objects and background clutter, the descriptors computed from the image are compared to descriptors computed from a database of reference images of known objects. Euclidean distance is used to measure the similarity between two feature vectors. Since the SIFT features in the test image need not all be produced by objects, there has to be a way of rejecting some of them as background clutter. One method is to reject a test image feature as 'background' if it matches multiple features in the database with similar distances as its nearest database feature. In other words, for a feature to be considered 'foreground', it has to match to a single database feature much better than to any other database feature. Once a set of test image features is tentatively matched to a particular object in the database, each of them provides an independent prediction for the position, scale and orientation of that object in the test image. If the predictions are in reasonable agreement, then the object is taken to be present in the image. The biggest advantages of this approach to recognition is that (1) there is no need for a pixel-wise segmentation of the image, and (2) by using local features, recognition is possible even under occlusion.

The constellation model by Fergus et al. [2003] is another approach to object recognition using local image descriptors. It uses the Kadir-Brady region detector, which is similar to the one used by SIFT. At each pixel location in the image, the detector computes the entropy of the histogram of pixel intensities in a circular region of various scales (i.e. radii) centred at that location. Treating the entropy $H(s)$ as a function of scale s, image locations that produce a local maximum of $H(s)$ (over some fixed range of scales) are selected as interesting. This detector tends to find blob-like regions in the image.

In the constellation model, interest regions are used as candidates for parts of objects. Each object consists of a set of parts, and an object-specific model specifies distributions for the relative locations and appearances of those parts. Given a test image containing at most one object, the region detector is applied to the image to find interest regions. All possible assignments of interest regions to the parts of an object are considered (including not assigning any regions to a part, which allows for occlusions). Any region not assigned to an object belongs to the background. A particular assignment of the regions to the parts of an object in effect defines an object-based reference frame to explain those regions. Relative to that reference frame, the probability of the regions being produced by the object is computed. This

allows the recognition system to decide whether an object is present in the image and if so, which object it is. Here the segmentation problem is solved in a brute force way by considering all possible interpretations of the detected regions as belonging to the object or the background. This is practical only because the number of parts in an object model and the number of detected regions in an image are both restricted to be small. The approximate scale invariance of the region detector makes the recognition less sensitive to scale changes in the objects.

2.8 Perceptrons and SVMs

Minsky and Papert [1988] consider the invariance properties of perceptrons. Their *group invariance theorem* shows that the perceptron can only implement simple transformation invariant classifiers that check whether the area occupied by the active pixels in a binary image is greater than some threshold. It cannot implement a classifier that, for example, distinguishes two different characters with the same number of active pixels regardless of where they appear in the image. This is not surprising since transformation invariant classification of two different input patterns would require nonlinear features of the input image. Perceptrons can be made more powerful with higher order units that combine many input units into one (i.e. make the inputs nonlinear features of the pixels). For example, Giles and Maxwell [1987] show that it is possible to handcraft features for a translation invariant perceptron by multiplying together pairs of input units to produce a higher order input for the perceptron.

The impossibility results for the perceptron no longer apply once nonlinear hidden units are included in a neural network. Hinton [1987] showed that a feedforward neural network with hidden units can learn to do translation invariant classification of different input patterns. The network is trained to discriminate among sixteen patterns, each of which is allowed to undergo 1D translation (with wrap around) without changing the class label. The training set contains only a subset of all possible combinations of patterns and translations, so the network cannot simply memorize each pattern at every possible position. The results show that it is able to generalize correctly to familiar patterns appearing in novel positions. This work is one of the few examples in the literature where a model learns invariance to a transformation without having it pre-specified and built into the model by hand.

A number of methods have been proposed for incorporating invariances into support vector machines. DeCoste and Scholkopf [2002] describe two simple methods: (1) *virtual support vectors* and (2) the *kernel jittering*. The virtual support vector method has three steps: (1) train an oridinary SVM on the training set, (2) expand the training set with artificial examples generated by applying the desired transformations on the support vectors of the trained SVM, and (3) train a new SVM on the expanded training set. Steps 2 and 3 can be repeated if necessary. The main advantage of this approach is that the training is computationally cheaper than applying the transformations to *all* training cases and then learning on the resulting (much larger) set. Since the learning computational cost for SVMs grows quadratically with the number of training cases, the savings can be large. This approach has been shown to do well on the MNIST classification task, with an error rate of 0.56% (DeCoste and Scholkopf [2002]). Its drawback is the assumption that the transformed versions of only the support vectors are needed to learn an invariant classifier. For example, a training case that is originally not a support vector may turn into one once a transformation is applied to it. The virtual support vector method ignores this possibility.

Kernel jittering uses an ordinary kernel which may not have the desired invariances, to define a new kernel that does have them. The output of the new kernel for a pair of vectors is computed by (1) applying all desired transformations to one of the vectors, (2) evaluating the ordinary kernel on the resulting vector pairs, and (3) outputting the result for only the nearest pair of vectors (nearest according to the oridinary kernel). Under some assumptions, this new "jittered" kernel satisfies Mercer's conditions and can be used in an SVM. Decoste and Scholkopf mention that in practice, the conditions

are almost always satisfied. The main drawback of this approach is that kernel evaluations become expensive if the set of possible transformations is large.

Scholkopf et al. [1998] suggest another approach that incorporates approximate local invariance by modifying the dot product between two data vectors. Consider a modified linear SVM classifier of the form

$$f(\mathbf{x}) = sgn(\sum_{i=1}^{l} \alpha_i y_i (B\mathbf{x} \cdot B\mathbf{x}_i) + b), \tag{2.4}$$

where $\mathbf{x}$ is the input vector, $f(\mathbf{x})$ is the binary classification function, $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_l, y_l)\}$ is the training set, α_i are the learned multipliers, and b is a scalar bias. The matrix B is a linear pre-processor applied to the classifier's input. We want to choose B such that the resulting classifier is approximately invariant to a pre-specified transformation (e.g. translation of the input). Once B is determined somehow, the modified SVM is learned in the usual way by maximizing the following expression:

$$\sum_{i=1}^{l} \alpha_i - \frac{1}{2} \sum_{i,k=1}^{l} \alpha_i y_i \alpha_k y_k (B\mathbf{x}_i \cdot B\mathbf{x}_k), \tag{2.5}$$

subject to the constraints $\alpha_i \geq 0$, $\sum_{i=1}^{l} \alpha_i y_i = 0$.

B is selected as follows. Let $g(\mathbf{x}) = \sum_{i=1}^{l} \alpha_i y_i (B\mathbf{x} \cdot B\mathbf{x}_i) + b$, so $f(\mathbf{x}) = sgn(g(\mathbf{x}))$. Transformation invariance of $g(\mathbf{x})$ is sufficient to make the classifier invariant. (This is not a necessary condition because even if a transformation changes $g(\mathbf{x})$, but not its sign, it will not affect the classifier output.) We can enforce local invariance of $g(\mathbf{x})$ to a differentiable, 1-parameter transformation $\mathcal{L}_t$ by making sure that $g(\mathcal{L}_t\mathbf{x})$ does not vary with respect to t at the identity transformation, taken to be at $t = 0$. So we need to minimize the following regularizer:

$$\frac{1}{l} \sum_{i=1}^{l} \left(\frac{\partial g(\mathcal{L}_t \mathbf{x}_i)}{\partial t} \bigg|_{t=0} \right)^2. \tag{2.6}$$

Substituting the expression for $g(\mathbf{x})$ into equation 2.6 gives

$$\frac{1}{l} \sum_{i=1}^{l} \left(\frac{\partial}{\partial t} \left(\sum_{k=1}^{l} \alpha_k y_k (B\mathcal{L}_t \mathbf{x}_i \cdot B\mathbf{x}_k) \right) \bigg|_{t=0} \right)^2. \tag{2.7}$$

After some manipulations, this can be re-written as

$$\sum_{i,k=1}^{l} \alpha_i y_i \alpha_k y_k (B\mathbf{x}_i \cdot BCB^T B\mathbf{x}_k), \tag{2.8}$$

where

$$C = \frac{1}{l} \sum_{i=1}^{l} \left(\frac{\partial \mathcal{L}_t \mathbf{x}_i}{\partial t} \bigg|_{t=0} \right) \left(\frac{\partial \mathcal{L}_t \mathbf{x}_i}{\partial t} \bigg|_{t=0} \right)^T. \tag{2.9}$$

Note that the tangent vectors $\frac{\partial \mathcal{L}_t \mathbf{x}_i}{\partial t}\Big|_{t=0}$ have zero mean since the transformation gradient is evaluated at the identity transformation ($t = 0$). Therefore C is a sample covariance matrix of tangent vectors.

The simplest way to include the regularizer (equation 2.8) in the SVM objective function (equation 2.5) is to pick B such that $BCB^T = I$. Then the regularizer becomes identical to the second term of the objective function, and maximizing the objective makes the regularization term small, as we

want. BCB^T can be made equal to identity with the choice $B = C^{-1/2}$. If C is invertible, its inverse square root exists because C is a non-negative matrix. If C is not invertible, then it is replaced by $C_\lambda = (1 - \lambda)C + \lambda I$ for $0 < \lambda \leq 1$. C_λ is invertible and non-negative because C is a non-negative matrix. Scholkopf et al. [1998] call it the *tangent covariance matrix*.

We can think of the linear pre-processing done by B as a form of transformation-specific whitening. C (or C_λ) is positive definite, so it can be decomposed as $C = SDS^T$, where S is an orthogonal matrix containing the eigenvectors of C, and D is a diagonal matrix containing the corresponding eigenvalues. So $B = C^{-1/2} = SD^{-1/2}S^T$. Since the dot product is not affected by an orthogonal transformation, we get that $B\mathbf{x}_i \cdot B\mathbf{x}_k = (D^{-1/2}S^T)\mathbf{x}_i \cdot (D^{-1/2}S^T)\mathbf{x}_k$. Computing the dot product of the vectors pre-processed by B is equivalent to first projecting each input vector onto the eigenvectors of C, scaling the components of this projection by the inverse square root of the eigenvalues, and then computing the dot product of the scaled projection vectors. The eigenvectors of C with the biggest eigenvalues are the directions in input space along which the transformation gradient has the highest variance. Those components of the projected vector are scaled down by the square root of the corresponding eigenvalues, as done in whitening. As a result, the dot product after pre-processing by B is less sensitive to the transformation than the dot product in the original input space. This makes the linear SVM classifier approximately invariant to the transformation.

2.9 Higher order models

Many of the standard methods for learning models of image data use a single set of hidden factors to explain the observed data (e.g. PCA, ICA, RBM). One way to extend these models is to use two different sets of hidden units and form three-way cliques containing one visible unit and two hidden units, one from each of three sets of units. If the hidden units in one of the sets are clamped to a particular configuration, the model reduces to the original form with a single (unclamped) set of hidden units modeling the visible units. But now the parameters of this reduced model are a function of the clamped hidden configuration, instead of being constant as in the original form. So the new model is a higher order version of the original one. In general, the number of different sets of hidden units can be arbitrarily large.

The main attraction of higher order models is that the activities of one set of hidden units can modulate the interaction between the remaining sets of hidden units and the visible units. A model with only one set of hidden units also has this property, but in a much weaker sense. For example, clamping a hidden unit in an RBM also produces a different model over the remaining hidden units and the visible units by changing the biases into the visible units. But a higher order Boltzmann machine allows much richer ways of modulating the model. A third order Boltzmann machine with two sets of hidden units and three-way cliques (a clique contains one unit from each of three sets) can use one set of hidden units to modulate an *entire* RBM between the other set of hidden units and the visible units. This is more powerful than just changing the biases into the visible units.

Higher order models are useful for learning invariant representations. Consider a third order Boltzmann machine trained on images in such a way that one set of hidden units represents shape and the other represents viewpoint[3]. Then at test time, given an image, the shape and viewpoint representations can be inferred by prolonged Gibbs sampling that alternatively updates the two sets of hidden units with the visible units clamped. So a viewpoint representation is inferred simultaneously with a shape representation that is normalized for that viewpoint. This model implements the idea of doing invariant recognition by imposing a view-specific reference frame on the object and interpreting shape

[3]For example, during training constrain the activities of the shape units to be the same for images containing the same shape but different viewpoints, and constrain the viewpoint units similarly.

with respect to that frame. However, unlike other works that *first* infers a viewpoint and *then* infers the shape with respect to that viewpoint, this model infers both simultaneously. The idea of simultaneous viewpoint-shape inference by iterative settling in a three-way model was originally suggested by Hinton [1981], although that paper did not use higher order Boltzmann machines to implement the idea.

One of the main drawbacks of higher order models is that the number of parameters grows exponentially with clique size. Suppose that there are m separate sets of units, each containing n units. If we restrict the m-way cliques so that a clique contains exactly one unit from each set (the same restriction an RBM has in its 2-way cliques), then there are n^m possible cliques. Allowing one free parameter per clique, which is the most obvious way of extending a basic model to higher order, will require n^m parameters. For example, a model with three sets of 1000 units each has a billion parameters in it, so a massive training set is needed to fit it.

The alternative is to regularize the model so that its effective number of degrees of freedom is much smaller than n^m. One way is to allow only one free parameter *per unit* in each set, and then define the parameter for an m-way clique as the product of the parameters of the m units in that clique. It is equivalent to taking the full m-dimensional array of parameters and factoring it as the outer product of m vectors, each one n-dimensional. So in the case of $m = 2$, the matrix of parameters is given by the outer product of two n-dimensional vectors. In the case of $m = 3$, the 3D array of parameters is given by the outer product of three n-dimensional vectors, and so on. With this factorization strategy, the number of free parameters is only $n \cdot m$, so the exponential growth with respect to m is avoided. In general, one can define the parameter array to be the sum of k such separate factorizations, which allows $n \cdot m \cdot k$ free parameters. For the $m = 2$ case this corresponds to representing the full parameter matrix as the product of two rank-k matrices.

A possible disadvantage of factorization is that optimizing the model parameters can be difficult. A model defined in terms of the product of two scalar parameters has the following degeneracy: multiplying one parameter by a constant and dividing the other parameter by the same constant results in the same model. This can be a problem for gradient-based optimization since any particular setting of the parameters has an infinite number of equivalent settings in its vicinity. But with appropriate regularization (such L2 weight cost), the degeneracy can be removed. Another trick mentioned in the literature is coordinate descent: in a set of parameters that are multiplied together, optimize only one of them at a time and keep the remaining parameters in the product fixed.

The rest of this section will describe various types of higher order models and how they are used for learning invariant visual representations. We first explain higher order Boltzmann machines, and then look at higher order extensions of PCA and ICA.

2.9.1 Higher order Boltzmann machines

Higher order Boltzmann machines were first described in Sejnowski [1986]. Memisevic and Hinton [2007] describe the first application of such a model to real data. They use a third order Boltzmann machine to learn a model of transformations between pairs of images. It consists of two sets of visible units $\mathbf{x}$ and $\mathbf{y}$, and one set of hidden units $\mathbf{h}$. It represents the conditional distribution $p(\mathbf{y}, \mathbf{h}|\mathbf{x})$, defined as follows:

$$p(\mathbf{y}, \mathbf{h}|\mathbf{x}) = \frac{exp(-E(\mathbf{y}, \mathbf{h}; \mathbf{x}))}{\sum_{\mathbf{y},\mathbf{h}} exp(-E(\mathbf{y}, \mathbf{h}; \mathbf{x}))}, \tag{2.10}$$

where

$$E(\mathbf{y}, \mathbf{h}; \mathbf{x}) = -\sum_{i,j,k} W_{ijk} x_i y_j h_k. \tag{2.11}$$

W_{ijk} is the 3D array of parameters for the model. The array is not factorized, so it has one free parameter per three-way clique.

Once $\mathbf{x}$ is clamped to a particular configuration, the model reduces to an RBM with $\mathbf{y}$ as the visible vector and $\mathbf{h}$ as the hidden vector. But the parameters of that RBM are a function of $\mathbf{x}$. As can be seen in equation 2.11, a particular setting of the x_i's will add various slices of the 3D array together to define the weight matrix of the RBM. Since $\mathbf{x}$ is assumed to be always observed, inference and learning steps conveniently become identical to those of an RBM. This is possible only for the special case in which two of the three sets of units are observed. In the case of two sets of hidden units and one set of visible units, inference is much more expensive because it requires prolonged Gibbs sampling between the two sets of hidden units.

2.9.2 Bilinear models

Tenenbaum and Freeman [2002] use a bilinear model to decompose data into two factors corresponding to what they call *style* and *content*. They consider two kinds of bilinear models, *symmetric* and *asymmetric*. In the symmetric model, a k-dimensional data vector y is given by

$$y = \sum_{i,j} w_{ij} a_i^s b_j^c, \tag{2.12}$$

where the vector a^s represents style, b^c represents content, and w_{ij} are k-dimensional basis vectors. So the data vector is generated as a linear combination of basis vectors where the combination coefficients are given by the outer product of the style and content representations. This model treats both style and content symmetrically, i.e. mathematically there is no distinction between the style and content vectors.

In the asymmetric model, the data vector y is given by

$$y = \sum_{j} w_j^s b_j^c, \tag{2.13}$$

where w_j^s are style-specific basis vectors and b^c is the content representation. So the data vector is generated by linearly combining the style-specific basis vectors using the components of the content vector as the coefficients. Here there is a distinction, or asymmetry, between style and content in that style determines the basis functions while content determines how they are combined. The asymmetric model can be derived from the symmetric one by combining the style-specific terms in equation 2.12 as follows:

$$w_j^s = \sum_{j} w_{ij} a_i^s. \tag{2.14}$$

Both models are learned by minimizing the sum of squared errors between the training data vectors and their reconstructions. For the asymmetric model, learning is particularly simple and can be done with the basic matrix SVD. The training cases are assumed to be column vectors labeled by their (discrete) style and content classes. (The number of training cases for all possible style-content pairs are assumed to be the same. If there are multiple examples per style-content pair, they are averaged together to form a single training case.) The entire training set is arranged as a matrix such that each column contains a single content class, with the vectors corresponding to multiple styles concatenated together as one long column vector. So there are as many columns as content classes, and the number of rows is given by the product of the number of style classes and the dimensionality of the data vector. Let this matrix be $\bar{Y}$. It can be decomposed as

$$\bar{Y} = WB, \tag{2.15}$$

where the columns of W contain the style-specific bases and the columns of B are the coefficients for combining the bases. This decomposition can be computed by applying SVD to $\bar{Y}$

$$\bar{Y} = USV^T, \tag{2.16}$$

and setting $W = US$ and $B = V^T$. By throwing out the columns of W and rows of B corresponding to the smallest singular values in S, the dimensionality of the content representation can be reduced.

For the symmetric model, the learning is more complicated and involves multiple iterations of SVD. The basic idea is to compute the style and content representations for the training cases by alternatively keeping one of them fixed and optimizing for the other. Each such optimization step can be done using SVD. This iterative procedure is guaranteed to converge to a local minimum of the squared reconstruction error. Once the style and content representations are computed, the only unknown in equation 2.12 are the basis vectors w_{ij}, and they can be solved for analytically.

To describe the learning algorithm, first we define the *vector transpose* of a matrix. Consider the matrix $\bar{Y}$ as defined before, where each column is a concatenation of column vectors from the same content class but in different styles. The *vector transpose* $\bar{Y}^{VT}$ of such a matrix is created by horizontally concatenating the column vectors from the same content class, as separate columns, rather than vertically as a single long column vector (see figure 5 in Tenenbaum and Freeman [2002]). The vector transpose is analogous to the ordinary transpose if we view $\bar{Y}$ as a 2D array whose elements are column vectors – the vector transpose re-arranges those column vectors in the same way as the ordinary transpose re-arranges the scalar elements of a matrix.

Re-writing equation 2.12 in matrix form, we get that

$$\bar{Y} = [W^{VT}A]^{VT}B, \tag{2.17}$$

$$\bar{Y}^{VT} = [WB]^{VT}A. \tag{2.18}$$

The iterative learning is initialized by first computing B from the SVD of $\bar{Y}$ (just as in the asymmetric model learning). Since B is orthogonal, $B^{-1} = B^T$, and therefore $[\bar{Y}B^T]^{VT} = W^{VT}A$. So then A can be estimated from the SVD of $[\bar{Y}B^T]^{VT}$. A is orthogonal as well, which gives $[\bar{Y}^{VT}A^T]^{VT} = WB$. Now B can be re-estimated from the SVD of $[\bar{Y}^{VT}A^T]^{VT}$, and the whole procedure is repeated again. Once the iterations converge, W can be solved for analytically.

Grimes and Rao [2005] describe a variant of the style-content bilinear model where the style and content representations are assumed to be sparse. They use the same cost function (sum of squared reconstruction error) as in Tenenbaum and Freeman's symmetric model given by equation 2.12, with two extra additive terms to enforce sparsity of the two factors. Instead of using SVD, they train the model by gradient descent.

Grimes and Rao describe some interesting problems that they ran into while trying to make the learning algorithm work. Optimizing the squared reconstruction error (without any sparsity costs) by gradient descent did not work and got stuck in poor local minima. Including the extra sparsity costs caused the style and content representations to shrink to 0 in magnitude (thus achieving low sparsity cost) while making the weights extremely large so that the reconstructions have roughly the same numerical scale as the training data. To prevent this, they scale down the weights at each learning iteration by a gain factor so as to maintain a desired variance level for the activities of the style and content units in the model. Another problem is that the style and content representations can be multiplied and divided, respectively, by a constant without changing the output of the model. This is because the output depends only on the product of the style and content activities, not on their individual values. As a result, the parameter space contains many points near each other that all have exactly the same cost,

making optimization difficult. The problem is solved by minimizing the cost function with respect to one factor until near convergence while keeping the other factor fixed (basically coordinate descent).

Their training data consists of natural image patches. Translations are applied to these patches (e.g. ± 3 pixels both horizontally and vertically), and the model is trained on the resulting patches so that the content representation remains invariant to translation while the style representation remains invariant to different patches undergoing the same translation. This is achieved by first inferring the content representation of a set of training patches that have *not* been translated (i.e., the zero-translation view is the canonical view of the patches). Then the content representation is kept clamped while the input patches are translated, and the style representations are inferred. To keep the style representation invariant to translation, the style vectors for the *same* translation but *different* patches are adapted towards their mean. The cost function gradient is computed using the resulting style and content vectors, and the parameters are updated. The results show that the model manages to learn localized features that are translation-invariant.

2.9.3 Multilinear models

Vasilescu and Terzopoulos [2002] present an algorithm for learning a multilinear model of data. They extend PCA to arbitrary order by defining a higher order version of SVD. The training data is now assumed to be arranged as a multi-dimensional array.

The higher order SVD decomposes the N^{th}-order data array $\mathcal{D}$ into the product of a *core tensor* (analogous to the matrix of singular values in ordinary SVD) and a set of orthogonal matrices, one for each dimension of $\mathcal{D}$. Multiplication of a multi-dimensional array and a matrix is defined by a new operation called the *tensor product*. Its exact definition is not necessary to understand the high-level idea. Roughly, the higher order SVD is computed by "re-shaping" the N-dimensional array $\mathcal{D}$ as a matrix in N different ways, and then applying ordinary SVD to each of those N matrices. The results of these N runs of SVD are then combined to define the decomposition of the data tensor.

More specifically, given an N^{th}-order array $\mathcal{D}$, a *higher-order SVD* can be defined as follows: for each of the N dimensions of the array, construct an ordinary matrix by collecting the set of vectors obtained by varying that dimension while keeping all other dimensions fixed, and arranging them as column vectors. Compute the ordinary SVD of this matrix, and keep only the orthogonal basis for the column space from the result (the matrix U in USV^T). This procedure produces N matrices, $U_1, ..., U_N$.

Once the matrices $U_1, ..., U_N$ have been computed, it is straightforward to use them along with $\mathcal{D}$ to solve for the core tensor. The result is a decomposition of $\mathcal{D}$ in terms of the core tensor and N matrices. This decomposition is called the N-mode SVD. Since there are many ways to define the notions of rank and orthogonality for tensors, this is not the only way to define a higher-order generalization of SVD. N-mode SVD becomes identical to the ordinary SVD when applied to a a matrix.

Unfortunately, one of the most important properties of ordinary SVD does not carry over to its higher-order generalization. It is *not* the case that truncating the lower corner of the core tensor of $\mathcal{D}$ produces the best low rank approximation (in the squared error sense) of $\mathcal{D}$. (See an example of this in Lathauwer et al. [2000].) But in practice truncating the core tensor tends to produce good approximations to the data tensor.

More recent work by Vasilescu and Terzopoulos [2005] extends the above ideas to derive a multilinear version of independent component analysis as well. If we treat whitening as a pre-processing step for ICA, then ICA can be seen as an extension of PCA. Once the data is whitened, the mixing matrix that ICA is trying to estimate must be orthogonal (assuming the data was truly generated by an ICA model). So the ICA solution can be computed by modifying the PCA solution with an unknown rotation matrix. Analogously, we can define a multilinear version of ICA by modifying the multilinear PCA solution described above.

First, the N-mode SVD is computed as before. For each of the $U_1, ..., U_N$ matrices given by the N-mode SVD, the corresponding rotation matrices $W_1, ..., W_N$ are computed by doing ordinary ICA on the data matrix given by flattening the multi-dimensional array along each of its N dimensions. Then the solution found by N-mode SVD is modified by replacing U_i by $U_i W_i^{-1} W_i = C_i W_i$. Then the matrices W_i are absorbed into the core tensor from N-mode SVD to produce a new tensor product decomposition of the multi-dimensional array.

Chapter 3

Analysis-by-Synthesis by Learning to Invert a Black Box Synthesis Model

In this chapter we describe a new way of learning to infer reconstructive representations of data. Its advantage is that it can incorporate into the learning complex domain knowledge about how the data was generated. This potentially allows the representation to capture the true degrees of freedom in the data better than those learned by generic models like PCA or autoencoders.

Briefly, the premise of the chapter is this: we can often express knowledge about the underlying generative process of the data in the form of a *synthesis model*. For example, if the data is generated by a well-understood physical process, the model may be a simulation of it. The simulation may be controlled by a set of variables that can be smoothly changed to produce different data vectors, and these variables form the inputs to the synthesis model. By learning the corresponding *analysis model*, i.e. a mapping from a data vector to the synthesis inputs, it becomes possible to use the inputs as a reconstructive representation. Such a representation takes advantage of the domain knowledge built into the model.

Learning the analysis model is difficult – in a typical application, we only have the inputs to the function (data vectors) and not their corresponding target outputs (the inputs to the synthesis model that would reconstruct those data vectors). We describe a way of training a feedforward neural network that starts with just one labeled case (input-output pair) and uses the synthesis model to "breed" more labeled cases. As learning proceeds, the training set of input-output pairs evolves and the target output vectors that the analysis model assigns to unlabeled data vectors converge to the correct values of the control variables.

The explanation of the algorithm and its results are spread over two chapters. This chapter presents the algorithm and shows two simple applications of it, just to verify that it does work. The next chapter presents a more extensive application where we consider different ways of using the reconstructive representation for recognition.

3.1 Introduction

"Analysis-by-synthesis" is the idea of explaining an observed data vector (e.g. an image) in terms of a compact set of hidden causes that generated it. A *synthesis model* specifies how the hidden causes produce the data vector. An *analysis model* is the inverse mapping – it infers the causes from a given data vector. In coding terms, the analysis and synthesis models are the encoder and decoder, respectively, and the hidden causes represent a *code vector*. The composite of the two models should implement the identity function: inferring the code vector from a data vector followed by synthesizing from the code

vector should reconstruct the original data vector.

Here we consider the following problem: given a training set of data vectors and a synthesis model for that data, learn the corresponding analysis model. We will assume that data vectors and code vectors are both real-valued. For example, suppose that we have a face dataset and a graphics program that can generate any realistic face image. This program may have a set of inputs (e.g. pose, lighting, facial muscle activations) that can be smoothly varied to create any face. The task is to learn an analysis model that infers from a face image the graphics inputs that will accurately reconstruct that image. The inputs to the synthesis model can be seen as a reconstructive representation of the data vector.

Note that this is a *new type of problem* that existing learning algorithms are not designed to solve. Here we assume that a synthesis model of the data is given as part of the problem, and the goal is to learn the inverse of that particular model. In contrast, algorithms such as PCA, factor analysis, and ICA simply assume specific parametric forms for the synthesis model and fit the parameters to the data. A nonlinear autoencoder learns separate encoder and decoder models simultaneously by also assuming them to be of a specific parametric form with an analytic gradient. As explained later, our problem is more difficult than the ones solved by these standard methods.

3.2 Motivation

There are two main motivations for this work. First, it is a way to incorporate domain knowledge, via a synthesis model, into the analysis-by-synthesis framework for modeling data. Synthesis models are a natural way of expressing complex prior knowledge. For example, in modeling face images, knowledge about facial muscles and skin and how they interact to produce different expressions can be expressed as a physics-based graphics model. Having a simulation of the underlying physical process built into the model gives its inputs (i.e. the code vector) useful semantics. This approach can help mitigate the model mis-specification problem that affects simple generative or reconstructive models whose parametric forms are not powerful enough to correctly capture the true generative process.

Second, solving the above problem is a way to directly take advantage of *existing* models from computer graphics for learning representations of image data. Enormous effort has already been expended on building graphics models (see e.g. face models by Lee et al. [1995], and Sifakis et al. [2005]), and now they can be used for building better vision models. Successfully inverting realistic graphics programs will result in image representations that can improve object recognition and image coding.

3.3 Overview of our approach

Our goal is to design one learning algorithm that can be used to invert many different synthesis models. The algorithm we propose treats the synthesis model as a "black box" function that can be evaluated as many times as necessary, but knowledge of its internal details is not directly available. In particular, we assume that the gradient of the synthesis function is not known[1]. Decoupling the learning of the analysis model from the specific design details of the synthesis model allows different synthesis models to be inverted without changing the algorithm.

The learning problem as stated so far is very general. We do not solve this general version – instead, we consider a restricted setting in which the distribution in data space is assumed to be the result of combining a *unimodal distribution in code space* with a *deterministic, nonlinear synthesis model*. ICA is somewhat similar in that it also assumes a simple distribution in code space and a deterministic mapping to data space to explain the data distribution. Unlike ICA, here we do not assume that the

[1] In practice there may not be an analytic expression for the gradient of a complex graphics program.

components of the code vector are independent or that the synthesis mapping has a known parametric form. Here we are given a set of samples from the data distribution, and access to the synthesis model as a black box function. But we are not given any samples from the code distribution, nor do we know the exact form of this distribution to draw samples from it.

Whether the assumption of a unimodal code distribution is valid for a particular application depends strongly on the synthesis model itself. Currently we do not have a formal description of the full set of conditions on the synthesis model for its inverse to be learnable by our algorithm. This is one of the limitations that needs to be addressed in future work. Consider the linear case in which the synthesis model multiplies the input vector by an unknown matrix to produce the output vector. The elements of the matrix can be discovered trivially by evaluating the synthesis function on the standard basis vectors. So the problem is interesting only for the nonlinear case. The three synthesis models that we apply the algorithm to (in this chapter and next) are nonlinear, and in all three cases, it successfully learns an approximate inverse. While these results are encouraging, the generality of the algorithm and the conditions under which it will work still need to be characterized formally.

The goal of the algorithm is to learn a regression mapping from data space to code space. There are three difficulties: first, only the regression function's inputs (data vectors) are given for learning – the corresponding target outputs are unknown[2]. If they were known, then the problem reduces to standard supervised learning. For a complex synthesis model, we expect that inferring a high-dimensional code vector for a given data vector is too hard to do "by hand", ruling out the possibility of hand-labeling a large set of data vectors with their target outputs.

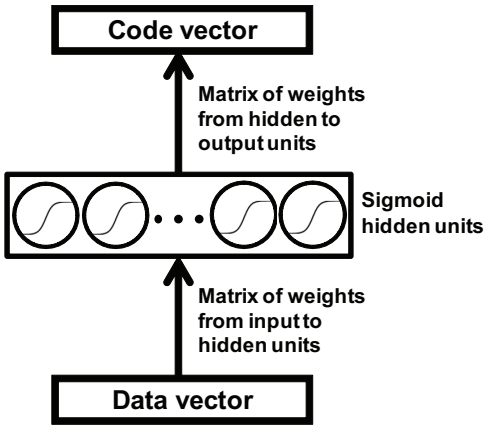
Second, the black box's gradient is unknown, so learning cannot be done by propagating the gradient of the data reconstruction error through the black box. Estimating it numerically by finite differencing is too inefficient. If the black box gradient were known, then learning becomes similar to that of an autoencoder whose decoder part is pre-specified and fixed.

Third, codes corresponding to the real data occupy only a very small volume in code space. For example, consider a face model that simulates muscles with springs. Humans cannot independently control each facial muscle, so the muscle activations are dependent on each other. Therefore only a small subspace of possible spring states correspond to valid facial configurations. This makes it impractical to naively take *uniform random samples* from code space, generate data vectors from those samples using the black box, and learn the recognition model from the resulting input-output pairs. Such an approach will waste almost all the capacity of the analysis model on "junk" training cases far away from the real data that we are interested in modeling.

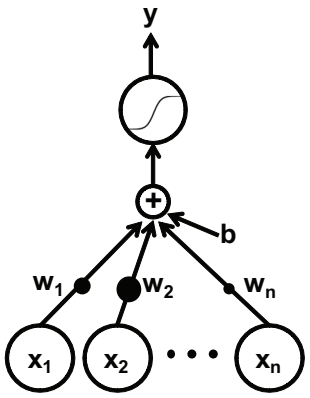
Our approach addresses these three difficulties. We assume that we are given a single point in code space, referred to as the *prototype*, near the mode of the (unknown) code distribution. Its purpose is to restrict the learning to the relevant part of code space. The algorithm randomly perturbs the prototype to compute a set of nearby code vectors from which their corresponding data vectors are generated using the synthesis model. The analysis model is trained by standard supervised learning on the resulting input-output pairs. In the subsequent learning iterations, codes even further away from the prototype are sampled and the corresponding input-output pairs are trained on. With more learning the sampling procedure produces code vectors from an increasingly broader distribution. The details are in section 3.4. As the algorithm "breeds" its own labeled training cases, we refer to it as *breeder learning*.

We use breeder learning to invert two different black boxes in this chapter, one for images of eyes (section 3.5.1) and the other for faces (section 3.5.2). In the former case, we got the software for the graphics model from its authors (Moriyama et al. [2006]) and simply used it as a black box subroutine in our algorithm. This shows the usefulness of de-coupling the design details of the synthesis model

[2]This is the more likely scenario in practice – the data vectors are usually supplied without any extra quantitative information about how they were generated, except for maybe class labels.



(a) Feedforward network with one hidden layer of sigmoid units.



(b) A single sigmoid unit.

Figure 3.1: (a) Architecture of the analysis neural network used in all our applications. The output units can be either sigmoid or linear, depending on the application. (b) A sigmoid unit is implemented using the *logistic function* as the squashing nonlinearity. So the overall function implemented by one such unit is $y = \frac{1}{1+\exp(-b-\sum_{i=1}^{n} w_i x_i)}$ where $x_1, ..., x_n$ are the inputs, $w_1, ..., w_n$ are learnable weights on the inputs, and b is a learnable scalar bias. y lies in the interval $[0, 1]$.

from the learning.

3.4 Breeder learning

Breeder learning is not specific to any particular parametric form for the analysis model. But as we will see, the set of input-output pairs is generated dynamically during training, so the model must be learnable in an online manner. We choose a feedforward neural network with a single hidden layer (see figure 3.1) to implement the analysis model.

When picking the prototype, we want to avoid outlying points far away from the high probability region of the code space. So we pick the prototype to be a point that produces a "realistic" image even under small random perturbations. Finding such a point by hand is assumed to be tractable. This of course depends on the particular synthesis model being inverted, but for the three applications considered in the thesis, constructing the prototype has been straightforward.

Breeder learning relies on the analysis network itself, as it is being learned, to find new code vectors to train on. To start off the search in the relevant part of the code space, the parameters (weights) of the analysis network should be initialized such that its output is approximately the prototype early on in training (regardless of which data vector it sees as its input). This can be done by setting the biases into the network's output units to produce the prototype code vector and all other parameters to small random values. As a result, early in training, the network's output will be affected by its input only weakly, and determined almost entirely by the output biases.

The complete set of inputs to the algorithm are 1) the synthesis model, 2) the prototype, and 3) a training set of data vectors. The final output is an analysis model trained to accurately infer from a data vector its corresponding code vector.

Once initialized, the weights are updated iteratively. Each iteration has four main steps (see figures 3.2, 3.3 and 3.4):

1. Data vectors from the training set are input to the current analysis network to infer their corre-

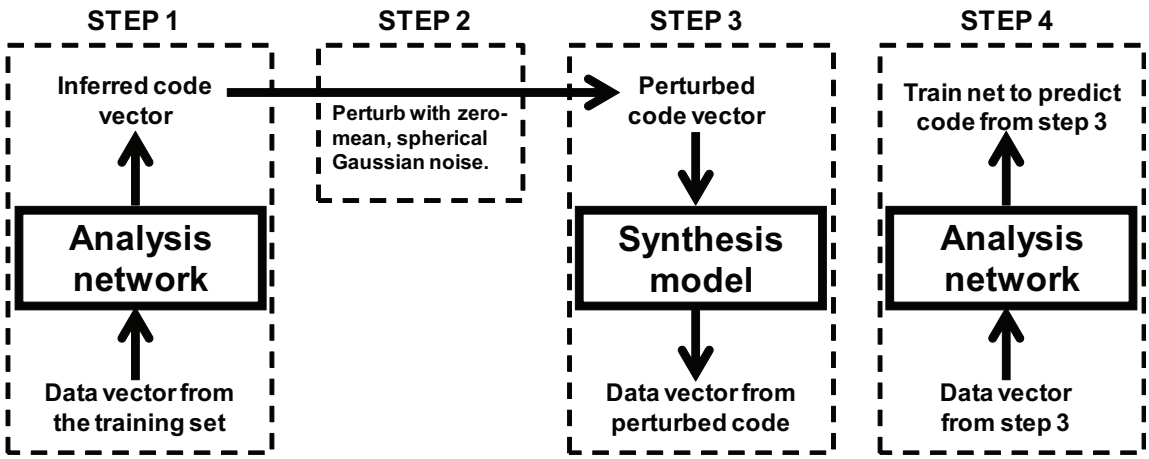


Figure 3.2: The main steps in a single iteration of the breeder learning algorithm.

sponding code vectors.

2. These codes are perturbed by adding zero-mean, spherical Gaussian noise with user-specified variance. In the case where the components of the code vector are restricted to the interval $[0, 1]$ and are represented by logistic sigmoid units at the output layer of the analysis network, the noise is added to the *input* of the unit. So the perturbed codes will still be in the interval $[0, 1]$. If the output units are linear (i.e. their values are in $[-\infty, \infty]$), the noise is added directly to their values.

3. The synthesis model is applied on the perturbed codes to produce their corresponding data vectors. The noisy codes and the data vectors generated from them form a set of input-output pairs on which the analysis network can be trained.

4. The weights are updated by the negative gradient of the squared error between the target code and the network's prediction.

Since the network weights are changing at each iteration, the codes inferred in step 1 for the same data vectors in the training set will change from one iteration to the next. So the input-output pairs that are used to update the weights are also changing throughout training. These pairs are used only for a single update and then thrown away. They are not re-used in future iterations.

Because of how the analysis network is initialized, it first learns to invert the synthesis model in a small neighbourhood around the prototype. The early noisy codes will be minor variations of the prototype, so the input-output pairs will not be very diverse. At this point the network can correctly infer the codes for only a small subset of X, i.e., those that are near the prototype's corresponding data vector. Randomly perturbing the outputs allows the network to discover codes slightly farther away from the prototype. Training on them expands the region in code space that the network can correctly handle.

In subsequent iterations, the network will correctly infer the codes for a few more real data vectors. Perturbing their codes generates new ones that are even farther from the prototype. As learning progresses, the training pairs become increasingly diverse, as the codes come from a larger region in code space. The network eventually learns to handle the entire region of code space corresponding to the real data vectors.

Some notes about the algorithm:

• The algorithm is not defined in terms of directly optimizing a loss function with respect to the training data vectors (e.g. optimizing the squared reconstruction error loss over data vectors). But empirically,

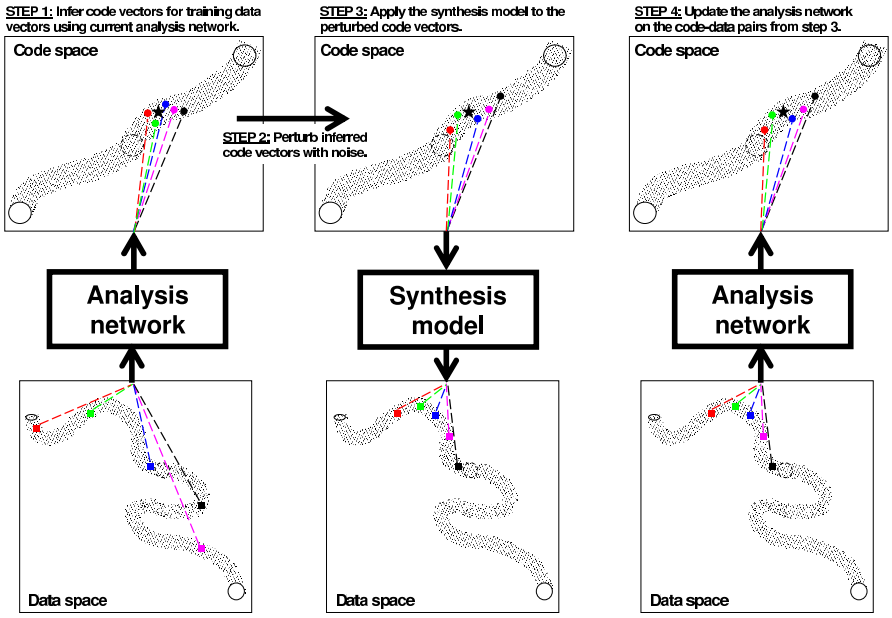


Figure 3.3: A cartoon summary of the main steps in a single iteration of the breeder learning algorithm. The manifolds in both data space and code space are "fat" in the sense that adding small amounts of noise to a point do not typically take a point off the manifold. In step 1, the training data vectors (squares), which come from all over the manifold in image space, are used as inputs to the current analysis network to infer some code vectors. Early on in training, these code vectors (circles) will be close to the prototype (star). They are perturbed with noise in step 2 to produce new code vectors. In step 3, the perturbed code vectors are used as input to the synthesis model to generate the corresponding data vectors. In step 4, the analysis network's weights are updated using the data vectors from step 3 as inputs, and their corresponding code vectors as the target outputs.

the data reconstruction error drops almost monotonically during training. A formal analysis of what loss function the algorithm is optimizing remains to be done.

• The algorithm is not doing a naive random search in code space. Instead, it uses the current analysis network itself to produce new codes to learn on. So the network's ability to correctly generalize to previously unseen data vectors is being exploited in the search. Section 3.5 shows that it allows the algorithm to discover codes that correspond to real data vectors much more efficiently than a random search. If the network generalizes incorrectly, the learning can become unstable and move away from a good solution. We have observed this behaviour in a small minority of the runs of the algorithm.

• The amount of noise used to perturb the code vectors (i.e. the variance of the spherical Gaussian noise) is set by trial-and-error. Too much noise makes the learning unstable, while too little makes it slow. For the applications we have tried the algorithm on, setting the noise level correctly was not difficult.

• The set of code-data pairs that the network learns on starts off mostly homogeneous, with all the code vectors close to the prototype. With more learning, the code vectors become more diverse as they start to come farther away from the prototype. Eventually the learning discovers the correct code vec-

Algorithm for training an analysis network A_w parameterized by weight vector w:

Given: Training set X of n data vectors $\{x_1, x_2, ..., x_n\}$, a black box synthesis model S, prototype code vector p.

Initialization: Set output biases of A_w using p, and the remaining weights to samples from a zero-mean Gaussian with a small standard deviation.

Weight update computed using the i^{th} (unlabeled) training case x_i:
Let y_i be the code vector inferred from x_i using the current analysis network A_w.

1. $y_i = A_w(x_i)$.
2. Perturb y_i randomly to create y'_i.
3. $x'_i = S(y'_i)$.
4. Supervised learning on (x'_i, y'_i):
   (a) $y''_i = A_w(x'_i)$.
   (b) $E = \|y'_i - y''_i\|^2$.
   (c) $w \leftarrow w - \eta \frac{\partial E}{\partial w}$.

Figure 3.4: Summary of the breeder learning algorithm. Although the above description is in terms of updating w using a single input-output pair at a time, in practice we average the gradient estimates from a *mini-batch* of such pairs to compute a single update.

tors for all the data vectors in the original (unlabeled) training set. Once the analysis network reaches an approximately correct solution, it will stay there since it is being trained on small perturbations of the correct code vectors and their corresponding data vectors.

• One underlying assumption is that Euclidean distance in code space is a more semantically meaningful way of assessing similarity than any generic distance metric in data space. Therefore small random perturbations in code space should produce semantically similar data vectors that may nevertheless have a large Euclidean distance between them.

• It is possible to formulate a mixture version of the algorithm that can handle far-apart modes in code space. It would require creating one prototype per mode. We have not yet tried this possibility.

• During training there is no attempt to filter out those synthetic data vectors (x'_i in figure 3.4) that are highly dissimilar from the data vectors in the training set ($\{x_1, x_2, ..., x_n\}$ in figure 3.4). Generic similarity metrics in data space can be highly misleading, and such filtering is likely to make the learning worse. Without filtering the network will occasionally learn on "junk" training cases. But if such cases are rare, their effect on the network is small.

Relationship to wake-sleep algorithm: The wake-sleep algorithm (Hinton et al. [1995]) was proposed for simultaneously learning two directed networks, one for analysis, or *recognition*, and the other for synthesis, or *generation*. (These networks consist of stochastic binary units, so both code and data vectors are binary.) The recognition network first infers codes for the real data vectors, which are then applied as inputs to the generative network to produce synthetic data vectors. The input-output pairs of one network are used to train the other in a supervised manner (with the roles of the input and output reversed).

Our approach here is similar in spirit, except only the recognition network is learned. By solving this more restricted problem we avoid the two drawbacks of wake-sleep. First, the wake-sleep recognition network wastes capacity by learning to invert the generative network on data vectors with low probability under the true data distribution. This is because early on in training the generative network is poor at producing the real data vectors. We avoid the problem by using 1) a good synthesis model throughout learning, and 2) a prototype to restrict the learning to the relevant part of the code space.

The second drawback of wake-sleep is 'mode averaging': if there are two different codes that can generate the same data vector, then the recognition network learns to infer a code that is neither, but a blend of the two. In our case the analysis network picks whichever code it happens to see first while training and learns to infer that code. The other code would simply be ignored. Based on similar premises, a modified version of wake-sleep has been used successfully in Deep Belief Nets (Hinton et al. [2006]) as a way of fine-tuning recognition and generative networks after an initial pre-training stage.

Random-code learning: A simpler alternative to breeder learning is to 1) sample an isotropic Gaussian centred on the prototype code vector and with a fixed variance, 2) generate synthetic data vectors from these samples, and 3) train the recognition network on the resulting pairs. In our experiments (section 3.5) this alternative consistently performs worse than breeder learning. If the Gaussian's variance is too large, many of the sampled codes will correspond to junk data vectors. If it is too small, it will almost never see valid codes that happen to be far away from the prototype. So the particular way in which breeder learning creates new codes is crucial for its success and cannot be replaced by a naive random search.

3.5 Results

The rest of the chapter describes two applications of breeder learning: inverting a synthesis model for images of eyes (Moriyama et al. [2006]), and an active appearance model for faces (Cootes et al. [2001]). In both cases we learn to infer a reconstructive representation for images by taking a synthesis model from the literature and simply "plugging it in" as the black box into breeder learning. These applications show our algorithm's usefulness for exploiting an existing synthesis model to define a compact representation of the data.

3.5.1 Inverting a 2D model of eye images

The black box is a 2D model of eye images proposed by Moriyama et al. [2006]. They use knowledge about the eye's anatomy to define a model parameterized by high-level properties of the eye, such as gaze direction and how open the eyelid is. Since breeder learning does not need to know the model's internal details, we explain them only briefly here. See Moriyama et al. [2006] for a full description.

Based on its inputs, the synthesis model first computes a set of polygonal curves that represent the 2D shape of the sclera, iris, upper eyelid, lower eyelid, and the corners of the eye. Once the shape is computed, we use a simple texture model to generate a 32×64 grayscale image from it (see figure 3.5). In total there are eight inputs to the black box, all scaled to be in the range $[0, 1]$. (These inputs affect only the shape; the texture model is fixed.) Given this black box and a training set of real eye images, we use breeder learning to learn the corresponding analysis model.

Dataset and training details: We use 1272 eye images collected from faces of people acting out different expressions. We normalize all images to be 32×64, and apply histogram equalization to remove lighting variations. See the odd-numbered columns of figure 3.7 for example images. Since the eye images come from faces with many different expressions and ethnicities, they contain a wide

variety of shapes and represent a difficult shape modeling task. We select the prototype code to be the vector with all components set to 0.5, which is the midpoint of each code dimension's range of possible values. From the set of 1272 images, 872 are used for training, 200 for validation and 200 for testing.

The analysis network has 2048 input units ($32 \times 64 = 2048$ pixels), 100 logistic sigmoid units in the hidden layer, and 8 sigmoid units in the output layer. A code vector is randomly perturbed during learning by adding zero-mean Gaussian noise with a standard deviation of 0.25 to the total input of each code unit. Training is stopped when the root mean squared error (RMSE) of the validation images is minimized. The recognition network trained by breeder learning achieves its best performance on the validation set after about 1900 epochs.

Figure 3.6 shows the RMSE achieved by breeder learning on the validation set as training proceeds. Random-code learning is unable to improve the RMSE beyond a certain value and starts overfitting because it only sees training cases from a limited region around the prototype. We tried various values for the variance of random-code learning, and the results shown are for the one that gave the best performance on the validation set.

Figure 3.7 shows examples of test images reconstructed by the recognition network trained with breeder learning. The inferred boundaries of the sclera and iris regions are superimposed on the real image. Notice that the network is able to correctly infer the codes for eyes with significantly different shapes. This is despite the limited texture model used by the black box.

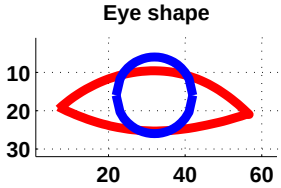
3.5.2 Inverting an active appearance model of faces

We now consider inverting an active appearance model (AAM) of face images (Cootes et al. [2001]). The AAM is a popular nonlinear synthesis model that incorporates knowledge about facial shape and texture to learn a low-dimensional representation of faces. Unlike the eye model, here the black box itself is learned from data, but this difference is irrelevant from the point of view of breeder learning.

Our implementation of the AAM follows Cootes et al. [2001]. Again, we only give a brief overview of it here. It consists of separate PCA models for facial shape and texture, whose outputs are combined via a nonlinear warp to generate the face image. As in Cootes et al. [2001], we apply PCA again to the shape and texture representations of the training images to produce an "appearance" model of faces.

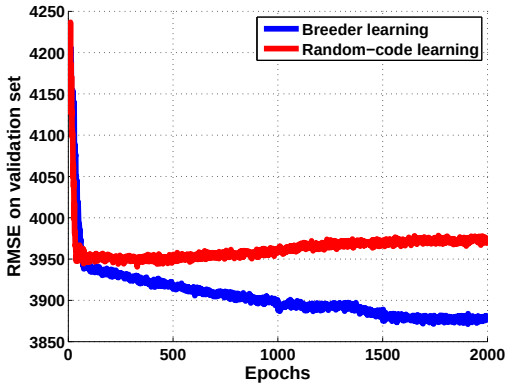
We first train the AAM using a set of face images, and then use it as a *fixed* black box for breeder learning. The face images are of size 30×30, and the AAM's appearance representation (i.e., the code vector) is chosen to be 60-dimensional. So for the purposes of breeder learning, we treat the AAM as a black box that takes 60 real-valued inputs and produces a 30×30 face image as output. (Note that the AAM learning procedure itself computes the codes for its *training* images as part of learning, so they are known, but we do not use them when learning the recognition network. On the other hand the correct codes for the *test* images are truly unknown.)

Dataset and training details: We use 400 frontal faces (histogram-equalized) containing different expressions and identities. The dataset is split into 300 training images, 50 validation images, and 50



Eye region	Pixel value
Iris	0
Sclera	255
Specularity	255
Background	128 + Noise

Figure 3.5: The synthesis black box for eyes: shape model (left), texture model (table on the right), and the image generated by applying the texture to the shape.



Algorithm	Test RMSE
Breeder learning	3902.56
Random-code learning	3977.37

Figure 3.6: Validation set RMSE (left graph) during training, and test set RMSE (above table) after training, for breeder and random-code learning algorithms on the eye dataset.

test images. None of the identities in the test set appear in the training and validation sets, so at test time, the recognition network has to generalize correctly to unseen identities (rather than unseen images of familiar identities). Note that only the 300 training and 50 validation images are used in the learning of the AAM itself.

The analysis network has 900 input units, a hidden layer of 100 logistic sigmoid units, and 60 linear output units. We select the origin of the code space, corresponding to the face with the mean shape and mean texture, as the prototype code. Since the network's output units are linear, the code vectors are perturbed during learning by adding zero-mean Gaussian noise (with 0.1 standard deviation) directly to the outputs. The analysis network trained by breeder learning achieves its best performance on the validation set after slightly fewer than 3400 epochs.

Figure 3.8 shows the RMSE results. Interestingly, the best reconstruction error achieved by breeder learning on the validation set is below that of the AAM itself (dashed line in the graph). This means that the net is able to find codes that are better in the squared pixel error sense than the ones found by the AAM learning. Example reconstructions of test faces are shown in figure 3.9. In most cases, the network reconstructs the face with approximately the correct expression and identity. In contrast, the reconstructions computed by the network learned with random-code learning are visually much worse and most of them resemble the face corresponding to the prototype code.

3.6 Iterative refinement of reconstructions with a synthesis network

So far analysis, or inference, has been treated as a purely *bottom-up* computation. A key property of analysis-by-synthesis is the use of top-down knowledge in the synthesis model to improve inference via

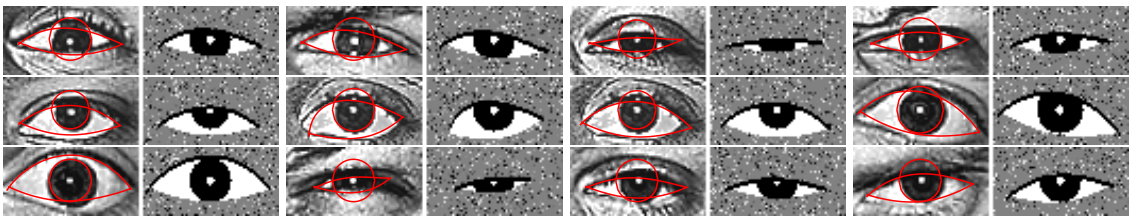
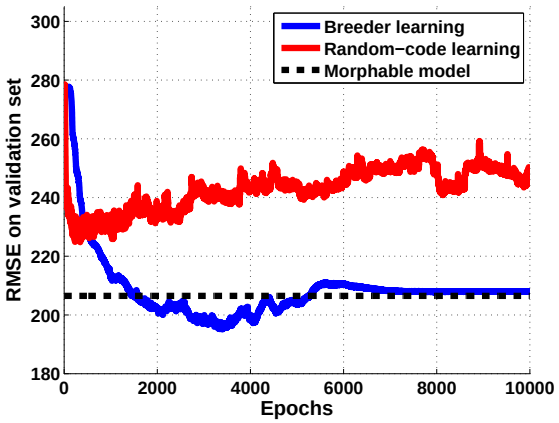


Figure 3.7: Test image reconstructions computed by the recognition network trained with breeder learning. The odd columns show the eye images with the superimposed curves describing the shape inferred by the recognition network. The even columns show the reconstructions computed by applying texture to the inferred shapes. The images should be viewed on screen to see the colour of the curves properly.



Algorithm	Test RMSE
Breeder learning	200.17
Random-code learning	229.31

Figure 3.8: Validation set RMSE (left graph) during training, and test set RMSE (above table) after training, for breeder and random-code learning algorithms on the face dataset.

a feedback loop that minimizes an error measure in data space itself. In our case implementing such a feedback loop requires knowing the gradient of the synthesis black box and so it is not possible. But once a fully-trained analysis network is available, an alternative approach becomes possible.

The idea is to approximate the function implemented by the black box with a *synthesis neural network* (Jordan and Rumelhart [1992]). This network emulates the black box: it takes a code vector as input and computes the corresponding data vector as output. Once such a synthesis network is trained, an approximate gradient of the data reconstruction error with respect to the code can be computed analytically by backpropagation. As a result, analysis now becomes a gradient-based iterative optimization problem that minimizes reconstruction error in data space.

Training the synthesis network is possible only because a fully-trained analysis model is already available. It provides the synthesis network with approximately correct target codes for the training images. Given these code-image pairs, training reduces to a standard supervised learning task. The analysis network restricts the learning to the small part of data space that contains the real data, thus making it practical. Without it, the synthesis network would have to be trained to emulate the black box everywhere in code space, which is impractical.

Figure 3.10 shows the main steps of the closed loop inference procedure using the analysis and synthesis networks (both fully trained). The initial code is computed by a bottom-up pass through the analysis network as before. But unlike in open-loop recognition, this initial estimate is subsequently refined by (approximate) gradient descent on the squared error between the data vector and its reconstruction. The iterations continue until the squared error stops improving. The details of each step are given in figure 3.11.

Figure 3.9: Test image reconstructions computed using the analysis network trained with breeder learning.

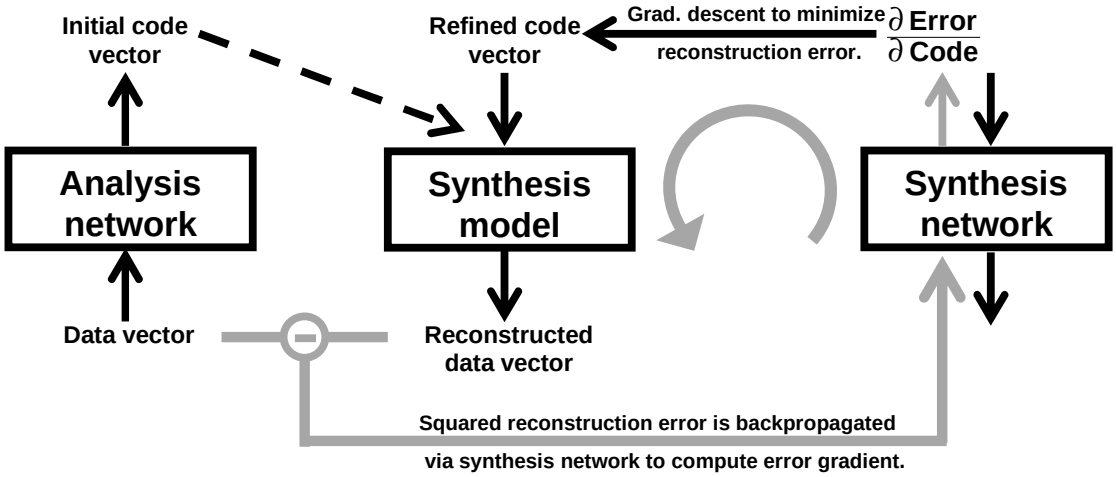


Figure 3.10: The main steps of the iterative inference procedure using both the analysis and synthesis networks. Note that the gradient computation proceeds in the *opposite* direction of the connections in the synthesis network.

Algorithm for closed-loop analysis of data vector x:

Given: Synthesis black box S, analysis network A_w, synthesis network $S_{w'}$.

Initialization: $y = A_w(x)$.

For each refinement iteration:

1. $x' = S(y)$.
2. $E = \|x - x'\|^2$.
3. Compute $\frac{\partial E}{\partial y}$ by backpropagation through $S_{w'}$.
4. $y \leftarrow y - \eta \frac{\partial E}{\partial y}$.

Figure 3.11: The closed-loop analysis algorithm using a synthesis neural network.

We learned a synthesis network to emulate the AAM and then used it to refine the reconstructions of faces. The average improvement in squared pixel error for the validation and test sets are 6.28% and 5.41%, respectively. It should be emphasized that the closed-loop analysis algorithm is used only as a way of fine-tuning the initial open-loop code estimate, which is already a very good solution. This side-steps the issue of whether a generic distance metric such as Euclidean distance can be used to correctly measure similarity in data space. Here we use Euclidean distance to measure *local* similarity only, i.e. to decide how an already good reconstruction can be made a little bit better. So minimizing Euclidean distance can be sensible for fine-tuning, even if it is prone to get stuck in shallow local minima when starting from a random solution.

3.7 Conclusions

Breeder learning is a new tool for engineers building analysis models. By taking advantage of the rich domain knowledge in a synthesis model, it can learn to infer better representations of data than those learned by standard methods such as PCA or autoencoders. Inverting complex physically-based synthesis models is an especially promising application of breeder learning. As the next chapter shows, successfully inverting such models can result in representations that are useful for object recognition.

Although the empirical results for breeder learning (including those in the next chapter) have been very encouraging, we do not yet have a good theoretical understanding of the algorithm and the conditions under which it can be expected to work. We have made some high-level comparisons to existing methods like ICA and the wake-sleep algorithm, but a formal analysis of breeder learning would be needed to make strong claims about its generality and usefulness. Such an analysis will be tricky because the model being learned itself is being used to decide what it will learn.

Chapter 4

Inverting a Physics-Based Synthesis Model of Handwritten Digit Images

This chapter presents another application of breeder learning, this time for inverting a synthesis model of handwritten digit images. Unlike the previous chapter, here we show how the reconstructive representation learned this way can be used for classification. The inversion problem is more difficult relative to the applications we saw earlier, and the results bring into full view the usefulness of breeder learning, and more generally, inverting synthesis models.

The synthesis model itself is interesting, so we describe it some detail even though breeder learning does not need the internal details. The model numerically simulates the physical process of handwriting as a highly idealized mass-spring system. This is not a new idea (e.g. Eden [1962], Hollerbach [1981]), but our contribution here is to make the simulation realistic enough for it to be actually useful in learning an accurate reconstructive representation of real handwritten digits.

We consider a number of different ways of using the learned representation for digit classification. Results for the MNIST dataset (section A.1) show that inverting synthesis models can be very useful for improving classification accuracy.

4.1 Introduction

One of the first proposed applications of the analysis-by-synthesis approach was the recognition of handwriting using a synthesis model that involved pairs of opposing springs. A parameterized simulation of the motor acts that produce handwriting should provide a natural way of characterizing it. For example, the images of twos in figure 4.1 are far apart in pixel space as measured by Euclidean distance, but they are all produced by very similar motor acts. In fact, these images were generated by first inferring the code vector (i.e. our synthesis model's inputs) for the left-most image, and then applying small random perturbations to it and generating from the perturbed codes. As we will see later, those inputs are a time sequence of spring stiffnesses, but figure 4.1 makes it clear that they form a meaningful representation of such images.

4.2 A Physics-based Synthesis Model for Handwritten Digits

The synthesis model relies on a simple physics-based description of how the arm moves when drawing a digit. We approximate the motor act of drawing as an arm moving on top of a horizontal drawing surface with a pen attached to its end. Inspired by arm movement models in the biomechanical literature, we

Figure 4.1: An MNIST image of a two (leftmost image) and the additional images generated from it by inferring its code vector and randomly perturbing that code. The pixel space representations are very different, but they are all clearly twos.

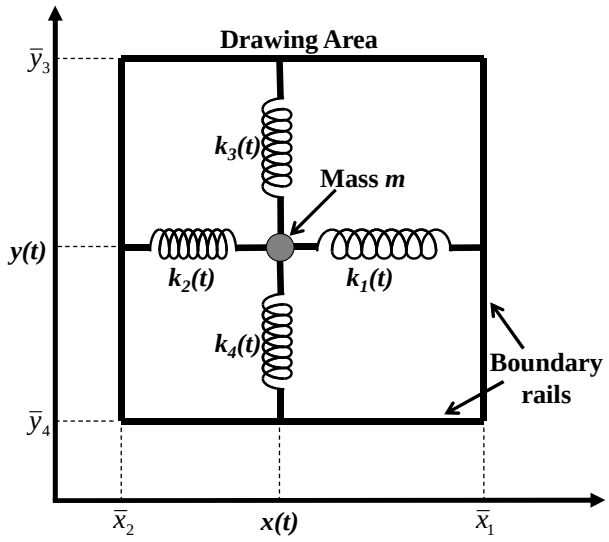


Figure 4.2: The mass-spring system we use to simulate the physics of drawing a handwritten digit.

use the mass-spring system shown in figure 4.2 to model the muscle forces exerted on the pen. The two pairs of opposing linear springs represent antagonistic muscle pairs in the arm. The mass represents the pen in contact with the drawing surface. The other end of each spring slides without friction along rails that mark out the boundaries of the drawing area. This allows us to treat the force vector generated by an individual spring to be axis-aligned and one-dimensional, which simplifies the simulation of the system.

There are two main steps in generating a digit: 1) simulating the the mass-spring system to compute a pen trajectory, and 2) applying ink to the trajectory. We now describe the details of these steps.

4.2.1 Computing the pen trajectory

The stiffnesses of the four springs are varied over time so that a time-varying force is applied on the mass. Each spring exerts a force (directed along an axis) on the mass according to Hooke's law:

$$F(t) = k(t)(x(t) - \bar{x}), \tag{4.1}$$

where $F(t)$ is the force at time t, $k(t)$ is the spring's stiffness, $x(t)$ is the (one-dimensional) position of the end of the spring attached to the mass, and $\bar{x}$ is the rest length.

The behaviour of the system is numerically simulated over a fixed number of discrete time steps. We assume that the four spring stiffness values at each time step are given as inputs. In the very first time step, the position of the mass is set to the equilibrium point determined by the initial stiffness values.

This is the position at which the net force on the mass is zero, and its x-coordinate is given as:

$$x[1] = \frac{\bar{x}_1 \cdot k_1[1] + \bar{x}_2 \cdot k_2[1]}{k_1[1] + k_2[1]} \tag{4.2}$$

Here we use the square bracket notation $[\cdot]$ for the discrete time index. The expression for the y-coordinate is analogous. The mass is assumed to have zero velocity and acceleration at the first time step.

At each subsequent time step n, the acceleration, velocity, and position of the mass m along the x-axis, denoted by $a_x[n]$, $v_x[n]$ and $x[n]$ respectively, are computed as:

$$a_x[n] = \frac{k_1[n] \cdot (\bar{x}_1 - x[n-1]) + k_2[n] \cdot (\bar{x}_2 - x[n-1])}{m}, \tag{4.3}$$

$$v_x[n] = (1 - \eta) \cdot v_x[n-1] + a_x[n], \tag{4.4}$$

$$x[n] = x[n-1] + v_x[n], \tag{4.5}$$

where η is a viscosity parameter. The expressions for the corresponding quantities along the y-axis are analogous. Therefore, given a sequence of stiffness values for the four springs as input, the above computation produces a pen trajectory in the drawing area.

In our implementation, the drawing area has size 78 units on each side. The mass and viscosity are set to fixed values selected by trial and error. It is possible to treat the mass and viscosity parameters as externally specified inputs to the synthesis model, but using fixed values seems to work well enough.

To compute a digit's complete pen trajectory, the simulation is carried out for a fixed number of time steps. Different digit classes require different numbers of steps to draw. For example, a one can be drawn with a quick, short stroke, while an eight has a circuitous trajectory that needs many temporal variations in the force, which can be simulated only with a large number of steps. To accommodate such differences, we use a different number of time steps per class. Most classes use 17 steps, but it can be as low as 10 (for ones) and as high as 20 (for eights). Digit instances within the same class are drawn using the same number of time steps.

4.2.2 Applying ink to the trajectory

We now describe one particular method for inking the trajectory. It is fast and generates fairly realistic-looking ink. Inking is done in two steps: first, the trajectory is thinly traced out on a pixel grid. Then the ink is given the desired thickness and brightness by convolving the trace image with a kernel. See figure 4.3 for a summary of the main steps.

A 36×36 pixel grid is overlaid at the centre of the drawing area. Any part of the trajectory that goes outside this grid is not inked. The sequence of pen positions is upsampled by linearly interpolating three points between two consecutive points on the trajectory. We call the original trajectory points 'coarse-grain' points and the newly interpolated ones 'fine-grain'.

The coordinates of the points are real-valued, so some kind of discretization is necessary to relate them to pixel coordinates. This is done by applying ink on the four pixels nearest to each point on the trajectory. A fixed amount of ink is split among the four pixels using bilinear interpolation (so the closer a pixel is to the trajectory point, the greater the fraction of ink it gets). Coarse-grain points contribute 2 units of ink to their four nearest pixels. The fine-grain points also contribute ink the same way, except the amount of ink they contribute is zero if they are less than one pixel apart and rises linearly to the same amount as the coarse-grain points if they are more than two pixels apart. This prevents a large amount of ink clumping in a small part of the image if the fine-grain points happen to fall closely together. The result is a thin trace of the trajectory on the pixel grid.

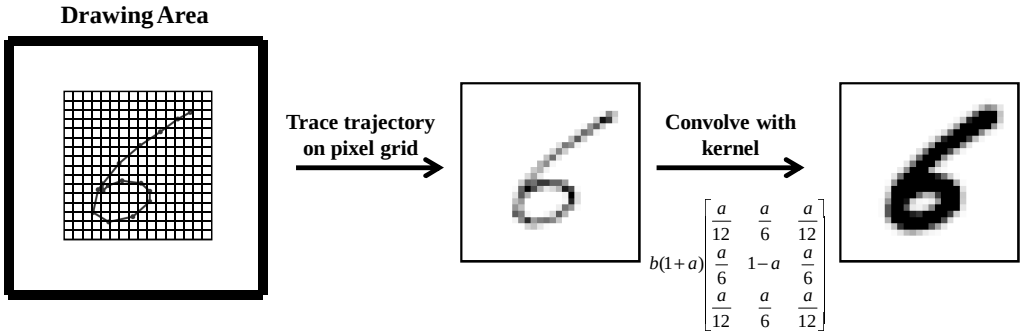


Figure 4.3: Main steps in applying ink to the trajectory.

The trace image is then convolved with a 3×3 kernel to give the ink the desired thickness and brightness. The entries of the kernel are computed based on two scalar inputs which determine the brightness and thickness of the stroke. The pixel values in the convolution output are clipped to lie in the interval $[0, 1]$. Finally, a 28×28 window is cropped from the middle of the 36×36 pixel grid to produce an image of the same size as those in the MNIST database.

This inking algorithm generates reasonably realistic output. In particular, the ink looks more realistic than what can be generated by applying Gaussian blur to the trace image. Real images from the MNIST database tend to have ink that sharply drops off in intensity at the edges, which cannot be created correctly with Gaussian blur. One limitation of the current method is that the same brightness and thickness parameters are used to ink the entire trajectory. So it is not possible to generate significant variations in the ink brightness and thickness along the trajectory. This is a good approximation for most real images, but in some cases such variations are useful.

To summarize, the complete set of inputs to the synthesis model are a) the sequence of stiffnesses for the four springs (four scalars per time step), and b) two scalars for ink thickness and brightness. Given these inputs, the model first simulates the mass-spring physics to compute the pen trajectory. Then it puts ink on the trajectory to produce a grayscale image as the final output. Viewed from outside, the model's full set of inputs is treated as simply one long vector. The sequence interpretation that its components have inside the model is not exploited when the analysis network computes a code vector. We refer to the code vector as a *motor program*, in the sense that it is a sequence of commands computed to carry out a motor act.

As mentioned before, the number of time steps used in the simulation varies across digit classes. Therefore the dimensionality of the code vector depends on the class, making it awkward to learn a single analysis network for all classes. So we train a separate analysis network for each one. Keeping the classes separate also makes it possible to incorporate some other class-specific attributes into the synthesis model that are described next.

4.3 Class-specific modifications to the basic synthesis model

We now describe two enhancements of the synthesis model that apply only to some of the digit classes. The basic model as described so far assumes that the pen always stays in contact with the drawing surface and applies ink throughout its trajectory, which is inappropriate for certain digit classes. The enhancements are meant to get around this problem.

4.3.1 Lifting the pen off the paper

Drawing some instances of fours and fives requires temporarily lifting the pen off the paper. For both classes, there are two different kinds of motor acts people commonly use to draw the digit: one that requires lifting the pen, and another that does not. The former is more general as it can be used to reconstruct an instance drawn by either method. We adapt the generative model to simulate the lifting of the pen. The simplest way to do this is to turn off the ink for a fixed subset of time steps along the trajectory. In the initial tracing of the trajectory on the pixel grid (section 4.2.2), both coarse- and fine-grain points corresponding to a pre-determined subset of consecutive time steps do not contribute any ink to the pixel grid. The subsequent convolution step is carried out just as before.

We use this trick to draw all instances of fours and fives. By placing the 'inkless' time steps of the trajectory at the appropriate image locations, it is possible to accurately reconstruct any instance. The reconstruction results in section 4.5 show that the analysis neural network does learn to place the time steps properly to produce convincing reconstructions.

4.3.2 Adding an extra stroke

People often draw an extra dash through the middle of sevens and at the bottom of ones. In the MNIST database, about 2.2% of ones and 13% of sevens are dashed, so they are too common to be simply ignored. We model such images with an additional motor program that draws just the dash. Note that all images contain the conventional version of the digit, with a small percentage containing an extra dash. The problem here is different from the one we face with fours and fives: there are two distinct parts, with one part appearing rarely. Learning a motor program for each part separately is easy, and then both versions of the digit can be generated by deciding whether or not to run the motor program for the dash.

To generate the dashed version, first the trajectories of the main part of the digit (i.e. the 'normal' seven) and the dash are computed separately. They are then superimposed and traced out on the same pixel grid, and the trace image is convolved with the same kernel.

4.4 Training a neural network to infer the motor program from an image

Trying to invert a synthesis model such as the mass-spring simulator makes it clear why an algorithm like breeder learning is necessary. The training images do not come pre-labeled with their corresponding correct motor programs or any other kind of quantitative measurements on how they were actually generated. The alternative of autoencoder-style learning by minimizing the pixel reconstruction error requires the gradient of the output image with respect to the code vector.

Training the analysis network is a straightforward application of breeder learning. Of the three ingredients that the algorithm needs (synthesis model, training set of images, and prototype), we have already described the first one. Details of the MNIST dataset are in section A.1. The method for creating the prototype is described next.

4.4.1 Creating the prototype motor program

The prototype should be a point on or close to the manifold of motor programs for a digit class. Since the mapping from a motor program to the image it generates is highly non-linear, it is difficult to create a prototype by trial-and-error search for the stiffness values. We have created a graphical user interface in Matlab (figure 4.4) that allows the user to interactively choose the stiffness values. The interface contains four slider controls corresponding to the spring stiffnesses for a selected time step. When the slider positions are changed, immediate visual feedback is produced by showing how the shape of the

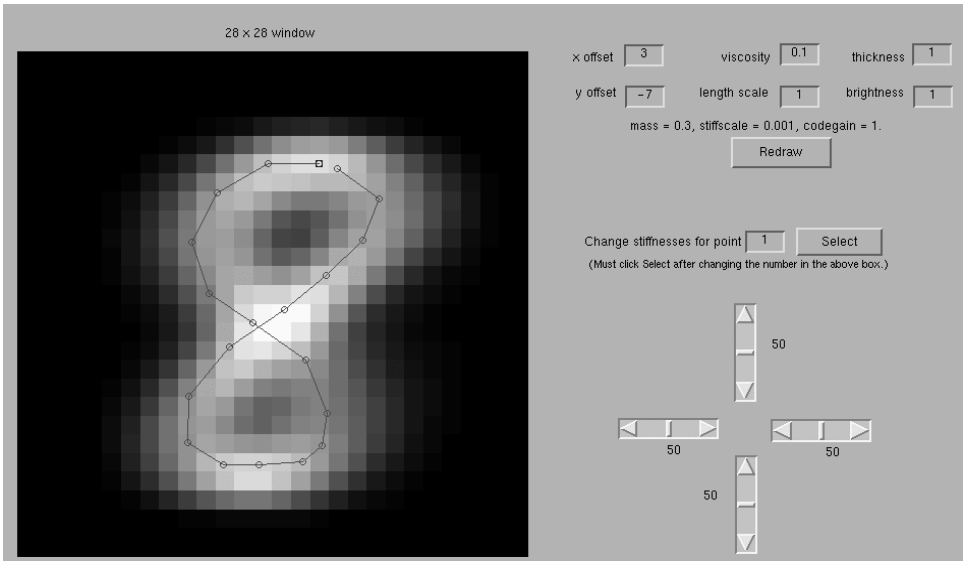


Figure 4.4: User interface for creating a prototype.

corresponding pen trajectory changes. This way the user can set the stiffness values 'visually' for all the time steps in a few minutes. To further simplify the prototype creation, we use the mean image of a class as the target shape for setting the stiffness values so that there is a clear visual goal. The user sets the stiffness sliders such that the pen trajectory approximately traces out the mean image. The thickness and brightness parameters are set conservatively to mid-range values.

4.4.2 Details of the learning

We train a separate analysis network for each digit class. The same learning algorithm is repeated for each digit class with a training set that contains only images from that class. In all cases the network has the three layer architecture shown in figure 3.1, with 784 pixels at the input layer, the motor program at the output layer, and a hidden layer in between. Both the hidden and output layers use logistic sigmoid units. The hidden layer contains 400 units. The dimensionality of the motor program depends on the number of time steps in the mass-spring simulation, which can be different for different classes. The general expression is $4 * n + 2$ where n is the number of time steps – there are 4 stiffness values per time step, and two additional values specifying the ink thickness and brightness. Most classes use 17 steps, so the dimensionality is 70 for those classes.

Dashed ones and sevens are treated effectively as two extra classes, with their own analysis networks trained on datasets containing only the dashed cases. Both of these analysis networks consist of two sub-networks: one for computing the motor program of the main part of the digit, and another for computing the motor program of the dash. The former is initialized to the analysis network trained for the regular version of the digit (i.e. dashless ones and sevens), and the latter is initialized randomly. The two sub-networks are then trained simultaneously with breeder learning. Given a dashed image as input, they both infer motor programs for their respective parts, which are then fed into the synthesis model to produce a single image as output. The component network for computing the motor program of the dash contains 100 hidden units. The other component starts off well-trained (its parameters have already been trained on dashless images), so the subsequent training on the dashed cases modifies its parameters only slightly.

Altogether we train 12 separate analysis networks (0-9, dashed one, dashed seven). Except for the training sets, prototypes, and the minor differences in the synthesis models, the overall learning procedure is identical across all 12 networks.

The biases into the output units are initialized to the logits (inverse of the logistic) of the prototype motor program. As explained in the previous chapter, this has the effect that if all other weights in the network are set to zero, then its output will be exactly the prototype. The rest of the weights in the network are initialized to independent samples from a zero-mean Gaussian with a standard deviation of 0.01.

For each class, we use a training set of 4400 images and a validation set of 1000 images (except for the two dashed digit classes, which have many fewer training cases). The training set is split into 'mini-batches' of 100 images, with one weight update done per mini-batch. Fully stochastic updates (i.e. one update per training case) can be very noisy because the actual input-output pairs that the network learns on are being dynamically generated and the distribution of those pairs is changing throughout training.

We measure training progress by computing the squared pixel reconstruction error the network achieves on the validation set. Training is stopped once this error is minimized. For almost all classes, training ends within 5000 epochs, which takes about 10 hours in Matlab on a 3GHz Intel Xeon machine.

Figure 4.5 shows how reconstructions of some training images of eights change as the analysis network is being trained. The reconstructions start off poor, but as training progresses, they improve (see the figure caption for an explanation).

4.5 Reconstruction results

Figure 4.6 shows validation set images for all classes except fours and fives that are reconstructed from the motor program inferred with their corresponding class-specific analysis network. Note that the motor program representation is able to handle the wide variety of shapes and ink within a class and produce qualitatively convincing reconstructions.

Figure 4.7 shows reconstruction examples of fours and fives from the validation set. Unlike for the other classes, here the synthesis model lifts the pen off the paper at certain pre-specified time steps. The reconstructions show how the analysis networks skillfully exploit this feature to good effect. They learn to pace the pen so that when it gets lifted, it is at spots along the trajectory that do not require any ink. Note that the same synthesis models can also be used to reconstruct even those fours and fives that do *not* require lifting the pen.

Figure 4.8 shows examples of validation images of dashed ones and sevens reconstructed using the extra stroke.

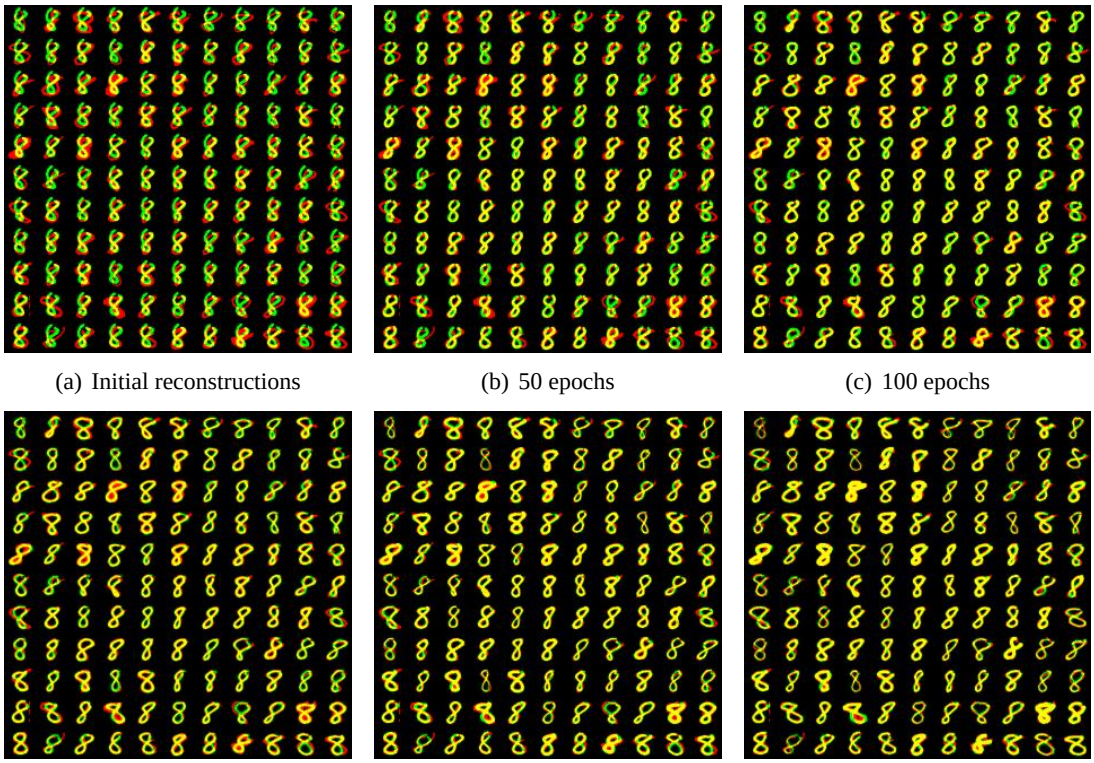
4.6 Iterative refinement of reconstructions with a synthesis network

As explained in the previous chapter, once the analysis network is trained, it becomes tractable to train a synthesis network to act as a smooth, differentiable approximation of the black box synthesis model. Then, using the gradient of the synthesis network, a motor program can be refined iteratively by minimizing the squared pixel reconstruction error.

Under the iterative inference procedure, the motor program for an image is initialized to its open-loop estimate computed by the analysis network. The main steps for refining this estimate are summarized below:

Repeat until the pixel reconstruction error converges:

1. Reconstruct from the current estimate of the motor program using the black box synthesis model.



(a) Initial reconstructions (b) 50 epochs (c) 100 epochs

(d) 150 epochs (e) 200 epochs (f) 1000 epochs

Figure 4.5: The image sequence shows how the analysis network for the digit class eight improves as learning progresses. Each step in the sequence shows MNIST training cases of eights in red, with the reconstructions in green computed using the analysis network after different number of epochs of training. (Since red + green = yellow, images that are reconstructed well should consist almost entirely of yellow pixels.) Before training begins, the motor program inferred by the analysis network for any input image will be close to the prototype. This is why at the beginning the reconstructions look almost identical regardless of which image is at the input. As breeder learning progresses, it produces code vectors that are increasingly farther away from the prototype and trains the analysis network on the corresponding input-output pairs. As a result the analysis network can infer different motor programs far away from the prototype for different input images. The final step in the sequence (after 1000 epochs of training) shows that the network is eventually able to reconstruct a wide variety of input images well.

2. Compute the gradient of squared pixel error with respect to the motor program by backpropagation through the synthesis network.

3. Update the current estimate of the motor program in the direction of the negative gradient to get a new estimate.

We train twelve synthesis networks, one each for the ten classes, dashed ones, and dashed sevens. Learning is supervised since the analysis networks can be used to label the training images with their approximately correct motor programs. Figure 4.9 shows an example of how closed-loop inference improves squared pixel reconstruction error. Iterative inference also helps to improve classification error significantly, as we will see in the next section.

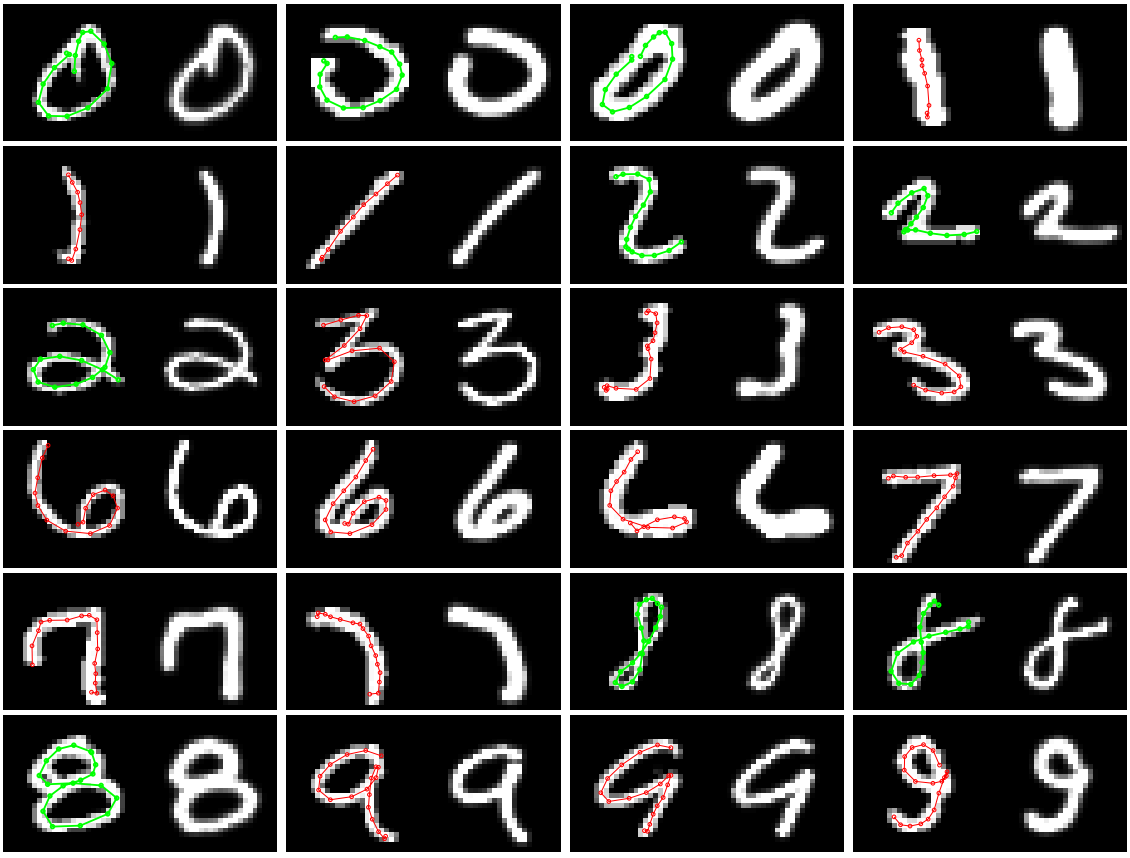


Figure 4.6: Examples of validation set images reconstructed by their corresponding model. In each case the original image is on the left and the reconstruction is on the right. Superimposed on the original image in colour is the pen trajectory. The dots along the trajectory indicate the time steps. (These images should be viewed on the screen to see the colour properly.)

4.7 Improving the synthesis model with additional learning

One of the drawbacks of the current synthesis model is that the ink it produces has constant thickness along the pen trajectory and the texture tends to be too smooth. This approximation is poor for the many MNIST images in which the thickness varies noticeably along the trajectory and the texture is jagged. Allowing separate thickness and brightness values *per time step* can partially address the problem. The motor program representation is extended to now have six numbers (4 stiffness values, thickness, brightness) per time step. The new analysis network for predicting the extended motor program can be initialized using the weights of the old one so that it starts off by predicting the same thickness and brightness values at each time step. Further training of the weights will then let the values vary with time. We have not yet tried this extension of the synthesis model.

More generally, the issue here is how to improve on a synthesis model that does not always produce completely realistic images. One solution is to train a neural network to predict the pixel difference between the input image and its reconstruction computed by the analysis-by-synthesis loop. This network takes the digit image as input, and computes the pixel residual as output. The residual can then be added to the reconstruction computed by the analysis-by-synthesis loop to produce the final reconstruction. The purpose of the residual network is to capture whatever aspects of the image that the synthesis model is incapable of generating correctly. We can think of the hidden units of the residual network as

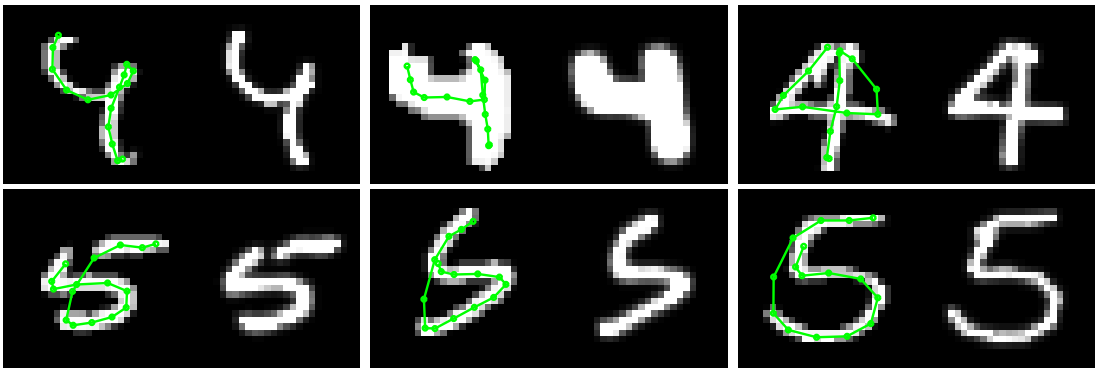


Figure 4.7: Reconstructions of validation set 4's and 5's with the synthesis model allowed to lift the pen off the paper at fixed time steps. When drawing 4's, the ink is turned off for timesteps 9 and 10. For 5's ink is turned off for timesteps 13 and 14. The pen trajectory for 5's starts with the downward vertical stroke, does the open loop at the bottom, moves back up to the top with the pen lifted, and finally finishes with the top horizontal stroke. (These images should be viewed on the screen to see the colour properly.)

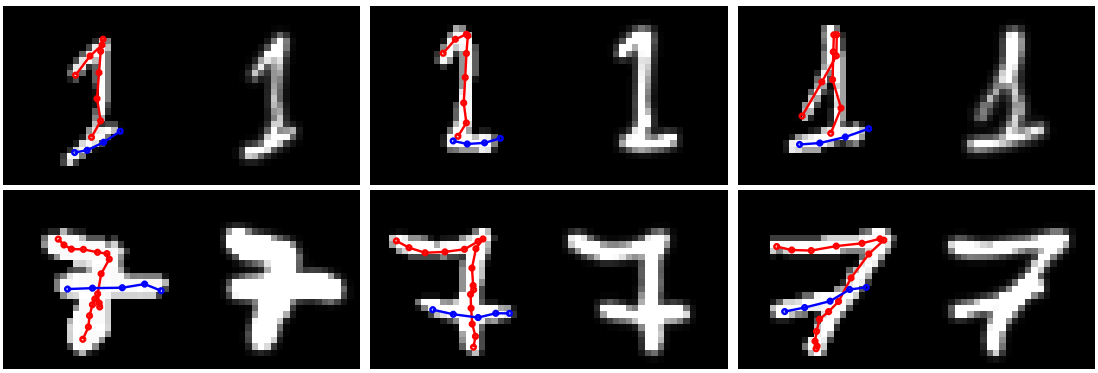


Figure 4.8: Examples of dashed ones and sevens reconstructed using a second stroke. The pen trajectory for the dash is shown in blue, superimposed on the original image. (These images should be viewed on the screen to see the colour properly.)

an extension of the original code vector. And its hidden-to-output weights can be seen as a *learnable* extension of the original, fixed, synthesis model. Note that the residual network can potentially use the original code vector itself as an extra input to predict the residual. The analysis and residual networks can be trained jointly to minimize the pixel reconstruction error. This is a general way of improving any synthesis model within the breeder learning framework.

4.8 Evaluating the usefulness of the motor program representation for classification

The reconstruction results in the previous section represent one way of evaluating whether motor programs are sensible for modeling handwritten digits. Another way is to use them for a high-level inference task such as classification. Many standard learning algorithms have been evaluated on the MNIST classification task, which makes it a useful benchmark for evaluating new algorithms.

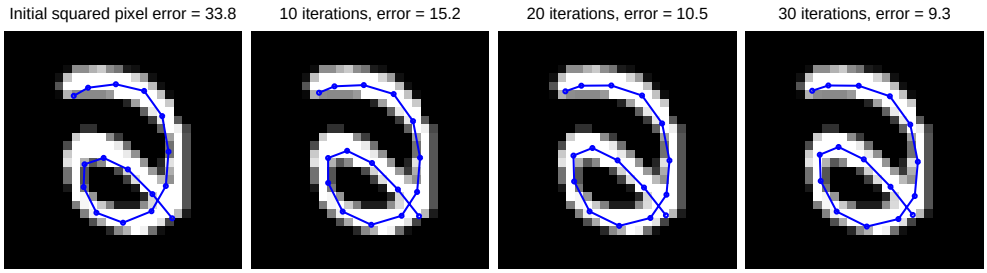


Figure 4.9: An example of how iterative refinement improves reconstruction. The image sequence above shows an MNIST image with its inferred trajectory superimposed on it. As more iterations of closed-loop inference are done, the trajectory fits the shape of the digit better. After 30 iterations, the squared pixel error is almost a quarter of its initial value. (These images should be viewed on the screen to see the colour properly.)

We have tried three different ways of building a classifier:

1. **Energy-based approach:** Use the motor program representation to assign to a test image a small set of scores, or 'energies', that measure the 'badness-of-fit' of the image under each class-specific model. Then feed the energies as input to a logistic regression classifier which converts them into a distribution over the ten class labels. The logistic regression parameters are the only ones trained discriminatively in the overall system and make up only a small fraction of all the parameters.

2. **Synthetic data approach:** New synthetic training images are generated by randomly perturbing the motor programs inferred from the MNIST images. These images can then be used as additional training cases for improving the performance of any discriminative model.

3. **Feature pre-training approach:** Take the features from all the class-specific analysis networks, use them to initialize the features in the first layer of a fully-connected feedforward neural network classifier, and train the classifier. This is an example of 'pre-training' features as part of a reconstructive model first, followed by discriminative fine-tuning.

The details of each of these approaches and their results are explained next. For the energy-based and synthetic data approaches, we use the iterative, closed-loop inference method of section 4.6 to compute the code from an image. In the former case, the energy values for a test image are computed after doing closed-loop inference on it. In the latter case, the codes that are perturbed to generate synthetic images are computed using closed-loop inference.

4.8.1 Energy-based approach

Given a test image, we want to use the ten class-specific models to predict its label. A simple way to do that is to ask each model to reconstruct the image, and then pick the class with the smallest pixel reconstruction error. We can use a more general version of this idea by computing multiple energy values (one of which can be reconstruction error) for the image under each model. Each energy measures how poorly the image fits under a model, with the correct class model hopefully having lower energies than the other models.

When there are multiple energies per class, picking the class with the 'lowest energy' is no longer straightforward. The proper method is to convert the energies into probabilities (by exponentiating the negative energy and dividing by the partition function), and then combine the probabilities into a conditional probability over labels. But computing the partition function involves a sum over all possible

images, so that's intractable. A tractable, but less proper, alternative is to use the energies as input to a logistic regression classifier. The classifier can learn to combine the energies and reconcile their different units in such a way that the classification accuracy is optimized.

We use three types of energies: 1) pixel reconstruction error, 2) energy under a Restricted Boltzmann Machine (RBM) model of the trajectory for a class, and 3) energy under a PCA model of the image residual (the difference between an image and its reconstruction) for a class. Each type contributes ten scores, one for each class, for a total of thirty values per test image. The classifier then computes the distribution over ten classes from its 30D input. It has a total of 310 parameters ($30 \times 10 + 10$ biases into the output units).

Note that for ones and sevens, we reconstruct the image using both the dashed and dashless models and pick the one with the lower reconstruction error (rather than keeping both). The trajectory model only uses the trajectory for the main part of the digit and ignores the dash, so there is only one model for both types. The image residual model is also shared between the dashed and dashless versions. That is why there are only ten energies per type rather than twelve.

We use the reconstruction computed from the motor program refined using the iterative inference procedure, instead of the one computed from the initial motor program inferred by the analysis network. Using the refined reconstruction reduces classification error of the energy-based approach by approximately 22%.

Image reconstruction energy: Sum of squared differences is used to measure the quality of the image reconstruction computed from the motor program. However, the simple L_2 distance between the image and its reconstruction is sensitive to even small alignment differences between the two shapes being compared. We take advantage of the inferred trajectory to make the comparison less sensitive to such problems.

The idea is to check whether locally shifting ink patches along the pen trajectory can improve the match between the image and its reconstruction. At each time step along the trajectory, a 5×5 window is placed on the reconstructed image centred at the pen coordinates for that time step. The patch covered by this window is then shifted up, down, left, and right, and at each position, the sum of squared differences is computed between the original image and the (now modified) reconstruction. The patch position that results in the lowest error is selected and the reconstructed image is modified to use that patch position. The same procedure is repeated for each of the remaining time steps. In effect, a greedy local search is done by wiggling the ink along the trajectory to find a reconstruction with a lower sum of squared differences. This method improves classification accuracy over using naive L_2 distance by approximately 18%.

Free energy under an RBM model of trajectories: When an analysis network is used to reconstruct an image outside of its own class, it tends to produce a contorted pen trajectory in an effort to explain as much of the ink as possible. For example, figure 4.10 shows how the two network and the three network reconstruct a two image. The three network achieves a better sum of squared pixel error by generating a highly contorted trajectory. The contortion can become even more pronounced with iterative refinement using the synthesis network. Since such contortions are not typical of the network's class, they provide useful information about the true class of the image.

We fit an RBM to the trajectories inferred for the images in each class by the correct class-specific analysis network. Once trained, it can be used to assign an energy to a test image's trajectory computed for that class. A highly unusual trajectory will be assigned a high energy, which can be useful for predicting the class label. Since the pen coordinates are real-valued, we use Gaussian visible units Hinton and Salakhutdinov [2006] to model them. We use 100 binary-valued hidden units. Training is

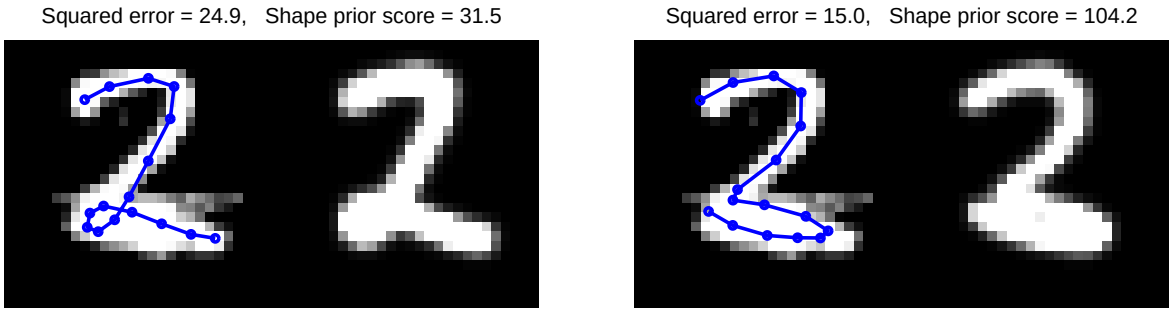


Figure 4.10: Results of reconstructing the same MNIST digit by two different class models. Within each pair of digits shown above, the MNIST digit is shown on the left of the pair and its reconstruction on the right. The digit pair on the left shows the two-model's reconstruction results, and the pair on the right shows the three-model's result. The inferred trajectory is superimposed on the original image. The three-model sharply bends the bottom of its trajectory to better explain the ink and achieves a better pixel reconstruction error, but such bending is highly unusual for threes. An RBM model of trajectories can penalize an unusual trajectory by assigning it a high free energy. (These images should be viewed on the screen to see the colour properly.)

done using the Contrastive Divergence algorithm (Hinton [2002]).

Let $\mathbf{v}$ be the vector of visible units, and $\mathbf{h}$ be the vector of hidden units. For an RBM with unit-variance Gaussian visible units and binary hidden units, the energy function is given by:

$$E(\mathbf{v}, \mathbf{h}) = \sum_i \frac{(v_i - b_i)^2}{2} - \sum_j b_j h_j - \sum_{i,j} v_i h_j W_{ij}, \tag{4.6}$$

where v_i is the i^{th} visible unit, h_j is the j^{th} hidden unit, W_{ij} is the weight between v_i and h_j, and b_i and b_j denote the biases for the v_i and h_j units respectively. $E(\mathbf{v}, \mathbf{h})$ is defined only for a full configuration of units that specifies the values of both $\mathbf{v}$ and $\mathbf{h}$. For a *partial* configuration in which only the values of the units in $\mathbf{v}$ are given, we can define an analogous quantity called *free energy*:

$$F(\mathbf{v}) = \sum_i \frac{(v_i - b_i)^2}{2} - \sum_j \left(P(h_j|\mathbf{v})t_j + P(h_j|\mathbf{v}) \log(P(h_j|\mathbf{v})) + (1 - P(h_j|\mathbf{v})) \log(1 - P(h_j|\mathbf{v}))\right), \tag{4.7}$$

where t_j is the total input into the j^{th} hidden unit. Like $E(\mathbf{v}, \mathbf{h})$, $F(\mathbf{v})$ can also be interpreted as measuring 'badness', but for a partially observed configuration $\mathbf{v}$. $F(\mathbf{v})$ takes into account the distribution induced by $\mathbf{v}$ over the unobserved units $\mathbf{h}$.

Energy under a PCA model of residual images: For each digit class, we fit a PCA model to the arithmetic difference between an image and its reconstruction by the correct class-specific motor program. This can be thought of as fitting a simple linear model to the leftover image structure not captured by the motor program. There is one such model per class.

At test time we project the residual image computed for each class onto the relevant PCA hyperplane and compute the squared distance between the residual and its projection. The distance is an additional energy value that assesses how well a class-specific model explains the test image.

The thirty energy values computed this way for a test image form the input to a logistic regression classifier. Its output unit is a discrete variable l that uses 1-of-K encoding to represent K different

Model	# of discriminatively trained parameters	% Test error
Logistic regression on pixels	7850	7.28
Fully-connected neural network (800 sigmoid hidden units)	636010	1.60
Energy-based approach	310	1.50

Table 4.1: MNIST test classification error rate of the energy-based approach compared to two baseline discriminative models.

labels. Given an input vector $\mathbf{x}$ consisting of the thirty energies, the probability of k^{th} class is:

$$P(l_k = 1|\mathbf{x}) = \frac{\exp(b_k + \sum_j W_{jk} x_j)}{\sum_{k'=1}^{K} \exp(b_{k'} + \sum_j W_{jk'} x_j)}, \tag{4.8}$$

where the matrix of weights W and the biases b_k are the learnable parameters of the classifier. The parameters are trained to minimize the cross entropy between the predicted and true distributions over the labels for a training case (the true distribution assigns probability 1 to the correct label, 0 for all others).

The energy-based approach to classification has a 1.50% error rate on the MNIST test set. This result is significantly worse than a number of discriminative models (e.g. a convolutional neural network has an error rate of only 0.89% (Ranzato et al. [2007])), but as table 4.1 shows, it is still better than some baseline models with many more parameters.

4.8.2 Synthetic data approach

We create new images in the same way breeder learning creates new samples to train on, by corrupting motor programs of MNIST images with noise. Initially we used the inking algorithm described in section 4.2.2. However, the classification results of the synthetic data approach improve significantly when using an alternative inking algorithm described next.

The ink generated by 2D convolution has smoother texture than real images. We get around this problem with a method inspired by work on patch-based generative models of images, such as image quilting (Efros and Freeman [2001]) and epitomes (Jojic et al. [2003]). It 'stitches' together patches from real MNIST images to produce new images. However, the problem here is somewhat simpler than the one solved by image quilting and epitomes because the patches are restricted to lie along the pen trajectory, which is inherently one-dimensional.

Figure 4.11 provides an overview of the algorithm. The input is a pen trajectory and the output is a 28×28 image with ink applied to that trajectory. The basic idea is to apply ink over short spans of the trajectory by copying ink segments from the corresponding spans along the trajectories of real MNIST images. A source image is selected based on 1) a trajectory with similar shape as the input trajectory over the current span being inked, and 2) an ink segment in that span that has good 'continuity' with segments previously placed on the input trajectory.

We use a span of 9 time steps for applying a single ink segment to the input trajectory. The longer this time span, the more realistic the ink will look in the output image since more of the ink comes from the same source image. But a longer span may also result in a worse match between the shapes of the input trajectory and a real image's trajectory. So it is necessary to keep the time span short. But if it is too short, the ink can look choppy and ragged. By 'ink segment' we mean the set of pixels that are

within a 5×5 neighbourhood of all the coarse and fine grain points on the trajectory within a particular time span.

We now describe the steps for selecting a single ink segment to place on the input trajectory. Shape similarity between two trajectories over a particular time span is measured by the sum of squared differences between the pen position coordinates at corresponding time steps. Both trajectories are aligned to have the same coordinates for the first time step of the span so that the shape comparison is translation invariant. Only MNIST images from the same class as the image being generated are considered for this comparison. A subset of MNIST images (e.g. 100) is initially selected based on shape similarity as candidate 'sources' for the current ink segment.

One source image is then selected based on continuity of the current ink segment with the previous segment. To make the transition between consecutive ink segments appear smooth, we use a 50% overlap between two consecutive time spans. The selected image is the one with the smallest sum of squared pixel differences between the second half of the previous segment in the output image and the first half of the current segment in the source image. Since the very first segment of the input trajectory has no continuity constraint, an image is selected randomly from the subset. Once a source image is selected, pixels from its segment are copied over to the corresponding pixel locations in the output image. Note that each segment is selected greedily, based on how well it overlaps with only the immediately preceding ink segment. The greedy approach is computationally efficient.

The new inking algorithm is used to generate 1.2 million synthetic images by adding zero-mean Gaussian noise with standard deviation 0.1 to the logits of the motor programs of 50,000 MNIST images. (The remaining 10,000 images are set aside as a validation set, so they are not used to generate synthetic images.) Each of the 50,000 images is perturbed 24 times to create the full set.

Classification results: We consider three types of classifiers: 1) k-nearest neighbour (kNN) classifier, 2) fully-connected neural network, and 3) convolutional neural network. Since kNN with 1.2 million extra images is very slow, we only use half of them. For the fully-connected net, we use a single hidden layer of 800 hidden units with a 10-way softmax unit at the output. For the convolutional net, the architecture is as follows: the first convolutional layer contains 25 kernels each of size 5×5, the second convolutional layer has 50 kernels also of size 5×5, followed by a fully connected layer with 200 sigmoid units, and finally a 10-way softmax unit at the output. The parameters in both types of networks are initialized randomly and trained to minimize the cross entropy loss.

In addition to the 1.2 million 'motor-distorted' images, we have also tried small, random affine transformations on the MNIST images to generate more images. Training on such 'affine distortions'

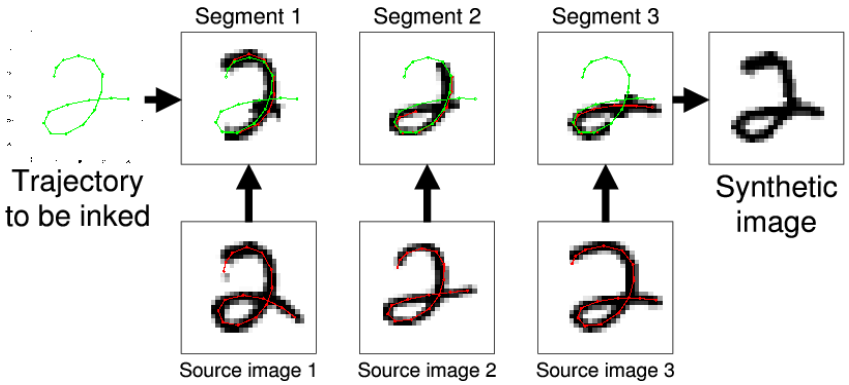


Figure 4.11: The patch-based inking algorithm cuts out ink segments from MNIST images and stitches them together to put ink on a new trajectory and generate a synthetic image (top row, right).

k-Nearest Neighbour models		
Classifier details	**Synthetic data**	**% Error**
L_2 distance	None	3.09
L_3 distance	None	2.83
L_2 distance	600K motor distortions	2.43
L_3 distance	600K motor distortions	2.24
Fully-connected feedforward neural networks		
Single hidden layer, 800 hidden units, logistic nonlinearity	None	1.60
	0.5 million affine distortions	0.97
	1.2 million motor distortions	0.75
	0.5 million affine + 1.2 million motor distortions	0.65
	1.7 million affine distortions	0.98
Convolutional neural networks		
Ranzato et al. [2007]	None	0.89
Convolutional net (25-50-200-10)	0.5 million affine + 1.2 million motor distortions	0.73

Table 4.2: MNIST test classification error rate for various discriminative models when trained with extra data generated from motor programs and affine distortions.

is a commonly used trick for improving classifier performance. Some of our results suggest that motor distortions are more useful and contain information that cannot be produced through random affine transformations, but we have not done an extensive comparison between the two types of distortions. We use 500,000 affine-distorted images (10 draws of random affine transformations per image $\times$ 50,000 training images), so in total there are 1.7 million synthetic images.

Table 4.2 shows the test error rates for the three types of classifiers. The motor distortions improve the performance in every single case for all three classifiers, even a sophisticated one like the convolutional net which already performs well without any extra data. For fully-connected nets, using only affine distortions does not perform as well as using only motor distortions, even when more of the former are used. But when the two are combined, the error rate is lower than what is achieved with either one in isolation. The 65 test mistakes made by the best net in table 4.2 are shown in figure 4.12.

When a network (both fully-connected and convolutional) is trained on the synthetic images, mini-batches are created by using an equal number of real and synthetic images (e.g. a mini-batch of 1000 images will contain 500 MNIST images and 500 synthetic ones). Therefore the relative influence of the real and synthetic images on the parameters is equal even though the synthetic images outnumber the real ones roughly by a factor 30.

4.8.3 Feature pre-training approach

Finally, we consider taking the features learned by the analysis networks and plugging them into a classifier. This is an example of first learning features using an unsupervised algorithm and then fine-tuning them with a supervised one.

We use the 4000 features from the ten class-specific analysis networks (400 features per network, excluding the networks for dashed ones and sevens) to initialize the first layer of a fully-connected feedforward net. These features then connect to a softmax unit at the output. The parameters in the output layer are randomly initialized. They are updated 100 times while keeping the pre-trained features fixed. After that, all the parameters are updated together.

Figure 4.12: 65 errors on the MNIST test set made by the net with 800 hidden units and trained on 1.7 million synthetic training cases, in addition to the original MNIST training set. Above each image, we show the true label (left of the arrow) and the predicted label. Some of the cases are arguably errors in the human labeling.

We also consider using the same 4000 features to initialize the first layer of a convolutional net. One difficulty here is that the kernels typically used in the first layer tend to be much smaller than the image itself (e.g. 5×5 in the previous section's net), while features of fully connected nets have the same size as the input image. The solution we have tried is to select, from each 28×28 feature, the 5×5 window with the largest L_2 norm and use it as a kernel. In other words, select the 5×5 window that has the strongest influence on a feature's output and ignore the rest.

Now we end up with 4000 kernels. A typical convolutional net has only tens of kernels in the first layer. We use the following heuristic to select a subset of the 4000: we go through the kernels in descending order of L_2 norm, and pick one if it has a cosine distance greater than some threshold (set by hand) from all the previous picks. (The first kernel is always selected by default.) If the minimum distance condition is not met, the kernel is not kept. Using this rule, 25 kernels are selected for the first layer. The remaining layers of the net are of the same size as those in the previous section (50 kernels in the second layer, followed by a fully-connected layer of 200 units, and then a softmax output unit) and they are initialized randomly. To see how high the performance of a convolutional net can be pushed, we combine both pre-training and synthetic data to train it. The 25 kernels picked by the above heuristic are shown in figure 4.13.

The results are shown in table 4.3. The error rate for the fully-connected net should be compared to

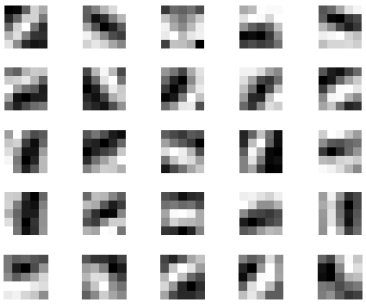


Figure 4.13: Pre-trained kernels.

Classifier with pre-trained features in the first layer	**Synthetic data**	**% Error**
Fully-connected feedforward net with a single hidden layer of 4000 logistic sigmoid units.	None	0.91
Convolutional net (25-50-200-10)	0.5 million affine + 1.2 million motor distortions	0.53

Table 4.3: MNIST test classification error rate for a fully-connected net and a convolutional net when the first layer is initialized using pre-trained features from the ten class-specific analysis networks.

the 1.60% obtained without either pre-training or synthetic data. The 0.53% result for the convolutional net would tie for the fourth best error rate (as of September 2009) among the long list of results for a variety of models shown on the MNIST webpage[1]. It should also be compared to a convolutional net without pre-training or synthetic data (0.89% by Ranzato et al. [2007]), and without pre-training but with synthetic data (0.73% from table 4.2).

As a final result, figure 4.14 shows a subset of the features of the fully-connected net before and after discriminative fine-tuning. These are the 100 features with the biggest L_2 distance between the 'before' and 'after' versions. Qualitatively they still look very similar, which supports the idea that the discriminative part of the learning does not need to significantly modify the result of pre-training to achieve high accuracy.

4.8.4 Further ideas for using the motor program representation

Another possible use of the motor program representation for classification is as a regularizer that encourages the classifier to be invariant to small changes in the motor program. This idea is inspired by the tangent propagation algorithm (see section 2.2), which proposed adding to the usual loss function an extra regularizer term that penalizes the norm of the gradient of the classifier output with respect to certain transformations (e.g. translation, rotation etc.) of the input image. The penalty encourages the classifier to be invariant to small such transformations.

Similarly, we propose a regularizer that penalizes the L_2 norm of the gradient of the classifier output with respect to the motor program of the input image. The intuition is that small changes in the motor program should not affect the class label, so ideally the gradient's norm should be 0. The penalty on the norm enforces it as a soft constraint.

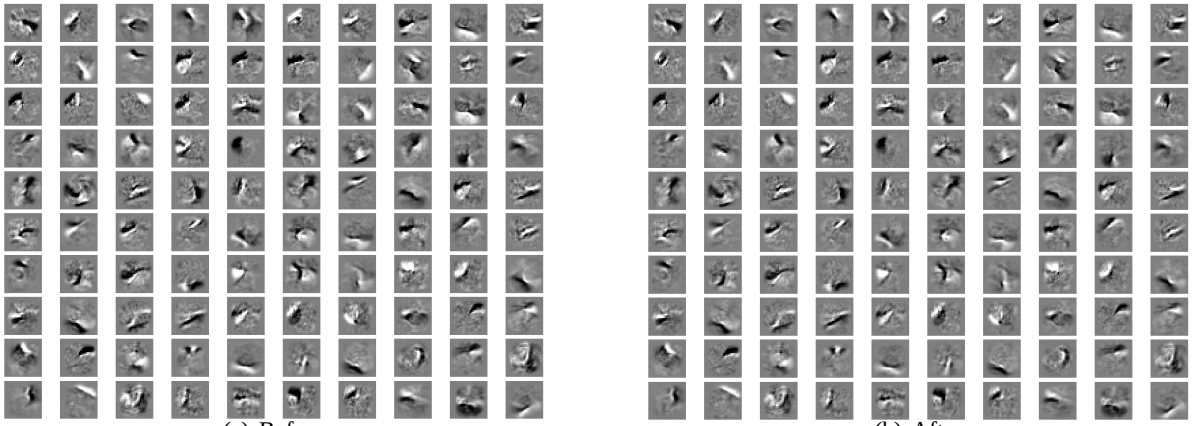
To implement the regularizer, we need its gradient with respect to the classifier parameters. The

[1] http://yann.lecun.com/exdb/mnist/

synthesis networks define a mapping from motor programs to images, and the classifier is the mapping from an image to the label. So the composite mapping from a motor program to the label is just a fully-connected feedforward neural net. The first and second derivatives of this function are straightforward to compute, and from those the gradient of the regularizer can be computed. The implementation is left as future work.

4.9 Conclusions

The results in this chapter show that inverting a synthesis model can be useful for object recognition. They also show that breeder learning works well enough to learn the approximate inverse of a non-trivial synthesis model like the mass-spring model. Even though the empirical behaviour of the algorithm is sensible, more work needs to be done to understand it better theoretically.



(a) Before (b) After

Figure 4.14: Features before and after fine-tuning.

Chapter 5

Implicit Mixtures of Restricted Boltzmann Machines

We present a mixture model whose components are Restricted Boltzmann Machines (RBMs). This possibility has not been considered before because computing the partition function of an RBM is intractable, which makes learning a mixture of RBMs by maximum likelihood intractable as well. However, when formulated as a third-order RBM, such a mixture model *can* be learned tractably using Contrastive Divergence. The energy function of the model captures three-way interactions among visible units, hidden units, and a single hidden discrete variable that represents the cluster label. The distinguishing feature of this model is that, unlike other mixture models, the mixing proportions are not explicitly parameterized. Instead, they are defined implicitly via the energy function and depend on all the parameters in the model. We present results for the MNIST and NORB datasets showing that the implicit mixture of RBMs learns clusters that reflect the class structure in the data.

5.1 Introduction

Our main motivation is to develop a model that can simultaneously learn a *clustering* of the data as well as a *cluster-specific latent representation*. This can be useful if the dataset consists of several subsets that require very different features to describe. For example, as shown in section 5.4, when we apply a mixture of two RBMs to unlabeled images of handwritten two's and three's from MNIST, it discovers clusters that correspond to those classes and learns class-specific features entirely unsupervised. Mixture of Factor Analyzers (MFA) (Ghahramani and Hinton [1996]) is another model that learns a clustering and a latent representation together. In that case learning is a straightforward application of the Expectation Minimization (EM) algorithm because all the quantities required by EM are tractable to compute. This is not the case for a mixture of RBMs, as explained next. Also, MFA is a *directed* model, while the model we propose is *undirected*.

A typical mixture model is composed of a number of separately parameterized density models each of which has two important properties:

1. There is an efficient way to compute the probability density (or mass) of a datapoint under each model.

2. There is an efficient way to change the parameters of each model so as to maximize or increase the sum of the log probabilities it assigns to a set of datapoints.

The mixture is created by assigning a mixing proportion to each of the component models and it is typically fitted by using the EM algorithm that alternates between two steps. The E-step uses property 1

to compute the posterior probability that each datapoint came from each of the component models. The posterior is also called the "responsibility" of each model for a datapoint. The M-step uses property 2 to update the parameters of each model to raise the responsibility-weighted sum of the log probabilities it assigns to the datapoints. The M-step also changes the mixing proportions of the component models to match the proportion of the training data that they are responsible for.

Restricted Boltzmann Machines (Hinton [2002]) model binary data-vectors using binary latent variables. They are considerably more powerful than mixture of multivariate Bernoulli models[1] because they allow many of the latent variables to be on simultaneously so the number of alternative latent state vectors is exponential in the number of latent variables rather than being linear in this number as it is with a mixture of Bernoullis. An RBM with N hidden units can be viewed as a mixture of 2^N Bernoulli models, one per latent state vector, with a lot of parameter sharing between the 2^N component models and with the 2^N mixing proportions being implicitly determined by the same parameters. It can also be viewed as a product of N "uni-Bernoulli" models (plus one Bernoulli model that is implemented by the visible biases). A uni-Bernoulli model is a mixture of a uniform and a Bernoulli. The weights of a hidden unit define the i^{th} probability in its Bernoulli model as $p_i = \sigma(w_i)$, and the bias, b, of a hidden unit defines the mixing proportion of the Bernoulli in its uni-Bernoulli as $\sigma(b)$, where $\sigma(x) = (1 + \exp(-x))^{-1}$.

The modeling power of an RBM can always be increased by increasing the number of hidden units (Roux and Bengio [2008]) or by adding extra hidden layers (Sutskever and Hinton [2008]), but for datasets that contain several distinctly different types of data, such as images of different object classes, it would be more appropriate to use a mixture of RBM's. The mixture could be used to model the raw data or some preprocessed representation that has already extracted features that are shared by different classes. Unfortunately, RBM's cannot easily be used as the components of mixture models because they lack property 1: It is easy to compute the *unnormalized* density that an RBM assigns to a datapoint, but the normalization term is exponentially expensive to compute exactly and even approximating it is extremely time-consuming (Salakhutdinov and Murray [2008]). There is also no efficient way to modify the parameters of an RBM so that the log probability of the data is guaranteed to increase, but there are good approximate methods (Hinton [2002]) so this is not the main problem. This chapter describes a way of fitting a mixture of RBM's without explicitly computing the partition function of each RBM.

Our approach in effect trades off one intractable problem – computing the partition function of an RBM – for another – exact maximum likelihood learning of an RBM. Nevertheless, it is a good trade because we know how to deal with the latter using a tractable, well-tested approximation, i.e. Contrastive Divergence (Hinton [2002]). One side-effect of the trade is that we no longer have mixing proportions as explicit parameters. But, as the results show, this does not appear to make the mixture model any less useful.

5.2 The model

We start with the energy function for an RBM and then modify it to define the implicit mixture of RBMs. To simplify the description, we assume that the visible and hidden variables of the RBM are binary. The formulation below can be adapted to other (non-binary) types of variables (e.g., see Welling et al. [2005]).

The energy function for an RBM is

$$E(\mathbf{v}, \mathbf{h}) = -\sum_{i,j} W^R_{ij} v_i h_j, \tag{5.1}$$

[1] A multivariate Bernoulli model consists of a set of probabilities, one per component of the binary data vector.

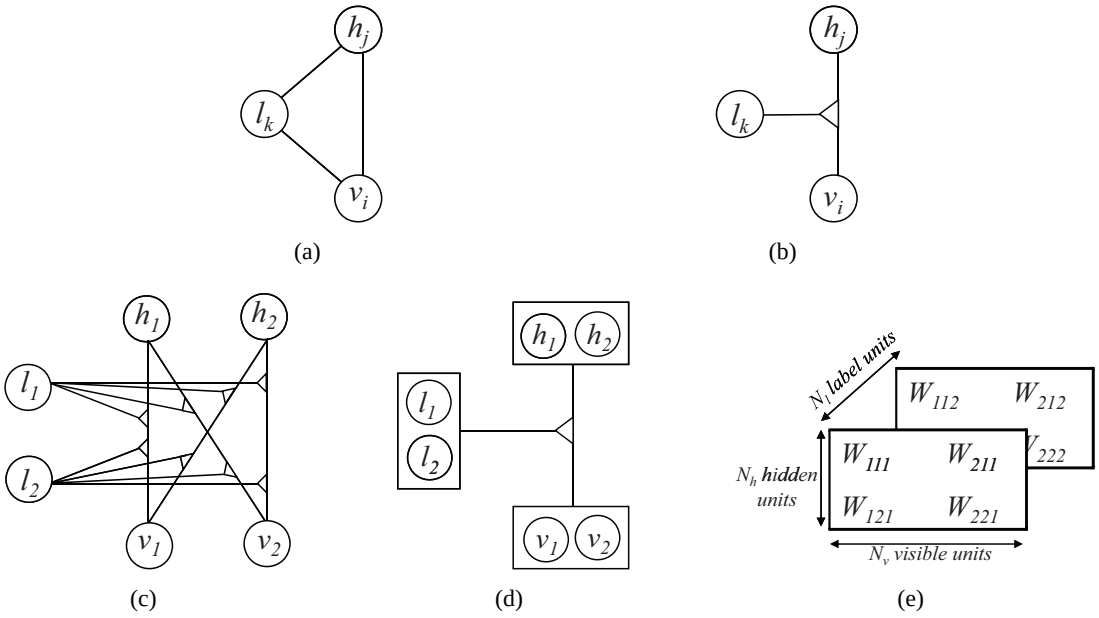


Figure 5.1: The Implicit Mixture of Restricted Boltzmann Machines. **(a)** Every clique in the model contains a visible unit, hidden unit, and label unit. **(b)** Our shorthand notation for representing the clique in (a). **(c)** A model with two of each unit type. There is one clique for every possible triplet of units created by selecting one of each type. The "restricted" architecture precludes connections between units of the same type. **(d)** Our shorthand notation for representing the model in (c). **(e)** The 3D array of parameters for the model in (c).

where $\mathbf{v}$ is a vector of visible (observed) variables, $\mathbf{h}$ is a vector of hidden variables, and W^R is a matrix of parameters that capture pairwise interactions between the visible and hidden variables. We omit biases for clarity. Now consider extending this model by including a discrete variable $\mathbf{l}$ with K possible states, represented as a K-dimensional binary vector with 1-of-K activation. Defining the energy function in terms of *three-way interactions* among the components of $\mathbf{v}$, $\mathbf{h}$, and $\mathbf{l}$ gives

$$E(\mathbf{v}, \mathbf{h}, \mathbf{l}) = -\sum_{i,j,k} W^I_{ijk} v_i h_j l_k, \tag{5.2}$$

where W^I is a *3D array* of parameters. Each slice of this array along the $\mathbf{l}$-dimension is a matrix that corresponds to the parameters of each of the K component RBMs. The joint distribution is

$$P(\mathbf{v}, \mathbf{h}, \mathbf{l}) = \frac{\exp(-E(\mathbf{v}, \mathbf{h}, \mathbf{l}))}{Z_I}, \tag{5.3}$$

where

$$Z_I = \sum_{\mathbf{u},\mathbf{g},\mathbf{y}} \exp(-E(\mathbf{u}, \mathbf{g}, \mathbf{y})) \tag{5.4}$$

is the partition function of the implicit mixture model. Re-writing the joint distribution in the usual mixture model form gives

$$P(\mathbf{v}) = \sum_{\mathbf{h},\mathbf{l}} P(\mathbf{v}, \mathbf{h}, \mathbf{l}) = \sum_{k=1}^{K} \sum_{\mathbf{h}} P(\mathbf{v}, \mathbf{h} | l_k = 1) P(l_k = 1). \tag{5.5}$$

Equation 5.5 defines the implicit mixture of RBMs. $P(\mathbf{v}, \mathbf{h}|l_k = 1)$ is the k^{th} component RBM's distribution, parameterized by the k^{th} slice of W^I along the **l**-dimension. Unlike in a typical mixture model, the mixing proportion $P(l_k = 1)$ is not a separate parameter in our model. Instead, it is *implicitly* defined via the energy function in equation 5.2. Figure 5.1 gives a visual description of the implicit mixture model's structure.

5.3 Learning

Given a set of N training cases $\{\mathbf{v}^1, ..., \mathbf{v}^N\}$, we want to learn the parameters of the implicit mixture model by maximizing the log likelihood $L = \sum_{n=1}^{N} \log P(\mathbf{v}^n)$ with respect to W^I. We use gradient-based optimization to do this. The expression for the gradient is

$$\frac{\partial L}{\partial W^I} = N \left\langle \frac{\partial E(\mathbf{v}, \mathbf{h}, \mathbf{l})}{\partial W^I} \right\rangle_{P(\mathbf{v},\mathbf{h},\mathbf{l})} - \sum_{n=1}^{N} \left\langle \frac{\partial E(\mathbf{v}^n, \mathbf{h}, \mathbf{l})}{\partial W^I} \right\rangle_{P(\mathbf{h},\mathbf{l}|\mathbf{v}^n)}, \tag{5.6}$$

where $\langle \rangle_{P()}$ denotes an expectation with respect to the distribution $P()$. The two expectations in equation 5.6 can be estimated by sample means if unbiased samples can be generated from the corresponding distributions. The conditional distribution $P(\mathbf{h}, \mathbf{l}|\mathbf{v}^\alpha)$ is easy to sample from, but sampling the joint distribution $P(\mathbf{v}, \mathbf{h}, \mathbf{l})$ requires prolonged Gibbs sampling and is intractable in practice. We get around this problem by using the Contrastive Divergence (CD) learning algorithm Hinton [2002], which has been found to be effective for training a variety of energy-based models (e.g. Roth and Black [2005], Roth and Black [2007], Welling et al. [2005], He et al. [2004]).

Sampling the conditional distributions: We now describe how to sample the conditional distributions $P(\mathbf{h}, \mathbf{l}|\mathbf{v})$ and $P(\mathbf{v}|\mathbf{h}, \mathbf{l})$, which are the main operations required for CD learning. The second case is easy: given $l_k = 1$, we select the k^{th} component RBM of the mixture model and then sample from its conditional distribution $P_k(\mathbf{v}|\mathbf{h})$. The bipartite structure of the RBM makes this distribution factorial. So the i^{th} visible unit is drawn independently of the other units from the Bernoulli distribution

$$P(v_i = 1|\mathbf{h}, l_k = 1) = \frac{1}{1 + \exp(-\sum_j W^I_{ijk} h_j)}. \tag{5.7}$$

Sampling $P(\mathbf{h}, \mathbf{l}|\mathbf{v})$ is done in two steps. First, the K-way discrete distribution $P(\mathbf{l}|\mathbf{v})$ is computed (see below) and sampled. Then, given $l_k = 1$, we select the k^{th} component RBM and sample from its conditional distribution $P_k(\mathbf{h}|\mathbf{v})$. Again, this distribution is factorial, and the j^{th} hidden unit is drawn from the Bernoulli distribution

$$P(h_j = 1|\mathbf{v}, l_k = 1) = \frac{1}{1 + \exp(-\sum_i W^I_{ijk} v_i)}. \tag{5.8}$$

To compute $P(\mathbf{l}|\mathbf{v})$ we first note that

$$P(l_k = 1|\mathbf{v}) \propto \exp(-F(\mathbf{v}, l_k = 1)), \tag{5.9}$$

where the *free energy* $F(\mathbf{v}, l_k = 1)$ is given by

$$F(\mathbf{v}, l_k = 1) = -\sum_j \log(1 + \exp(\sum_i W^I_{ijk} v_i)). \tag{5.10}$$

If the number of possible states of $\mathbf{l}$ is small enough, then it is practical to compute the quantity $F(\mathbf{v}, l_k = 1)$ for every k by brute-force. So we can compute

$$P(l_k = 1|\mathbf{v}) = \frac{\exp(-F(\mathbf{v}, l_k = 1))}{\sum_{k'} \exp(-F(\mathbf{v}, l_{k'} = 1))}. \tag{5.11}$$

Equation 5.11 defines the *responsibility* of the k^{th} component RBM for the data vector $\mathbf{v}$.

Contrastive divergence learning: Below is a summary of the steps in the CD learning for the implicit mixture model.

1. For a training vector $\mathbf{v}_+$, pick a component RBM by sampling the responsibilities $P(l_k = 1|\mathbf{v}_+)$. Let m be the index of the selected RBM.
2. Sample $\mathbf{h}_+ \sim P_l(\mathbf{h}|\mathbf{v}_+)$.
3. Compute the outer product $\mathbf{D}_m^+ = \mathbf{v}_+\mathbf{h}_+^T$.
4. Sample $\mathbf{v}_- \sim P_m(\mathbf{v}|\mathbf{h}_+)$.
5. Pick a component RBM by sampling the responsibilities $P(l_k = 1|\mathbf{v}_-)$. Let q be the index of the selected RBM.
6. Sample $\mathbf{h}_- \sim P_q(\mathbf{h}|\mathbf{v}_-)$.
7. Compute the outer product $\mathbf{D}_q^- = \mathbf{v}_-\mathbf{h}_-^T$.

Repeating the above steps for a mini-batch of N_b training cases results in two sets of outer products for each component k in the mixture model: $S_k^+ = \{\mathbf{D}_{k1}^+, ..., \mathbf{D}_{kM}^+\}$ and $S_k^- = \{\mathbf{D}_{k1}^-, ..., \mathbf{D}_{kQ}^-\}$. Then the approximate likelihood gradient (averaged over the mini-batch) for the k^{th} component RBM is

$$\frac{1}{N_b}\frac{\partial L}{\partial W_k^I} \approx \frac{1}{N_b}\left(\sum_{i=1}^{M}\mathbf{D}_{ki}^+ - \sum_{j=1}^{Q}\mathbf{D}_{kj}^-\right). \tag{5.12}$$

Note that to compute the outer products $\mathbf{D}^+$ and $\mathbf{D}^-$ for a given training vector, the component RBMs are selected through *two separate stochastic picks*. Therefore the sets S_k^+ and S_k^- need not be of the same size because the choice of the mixture component can be different for $\mathbf{v}_+$ and $\mathbf{v}_-$.

Scaling free energies with a temperature parameter: In practice, the above learning algorithm causes all the training cases to be captured by a single component RBM, and the other components to be left unused. This is because free energy is an unnormalized quantity that can have very different numerical scales across the RBMs. One RBM may happen to produce much smaller free energies than the rest because of random differences in the initial parameter values, and thus end up with high responsibilities for most training cases. Even if all the component RBMs are initialized to the exact same initial parameter values, the problem can still arise after a few noisy weight updates. A heuristic to get around this problem is to use a temperature parameter T when computing the responsibilities:

$$P(l_k = 1|\mathbf{v}) = \frac{\exp(-F(\mathbf{v}, l_k = 1)/T)}{\sum_{k'} \exp(-F(\mathbf{v}, l_{k'} = 1)/T)}. \tag{5.13}$$

By choosing a large enough T, we can make sure that random scale differences in the free energies do not lead to the above collapse problem. One possibility is to start with a large T and then gradually anneal it as learning progresses. In our experiments we found that using a constant T works just as well as annealing, so we keep it fixed. Note that this is not equivalent to simply dividing the weights of all the RBMs by a factor of T.

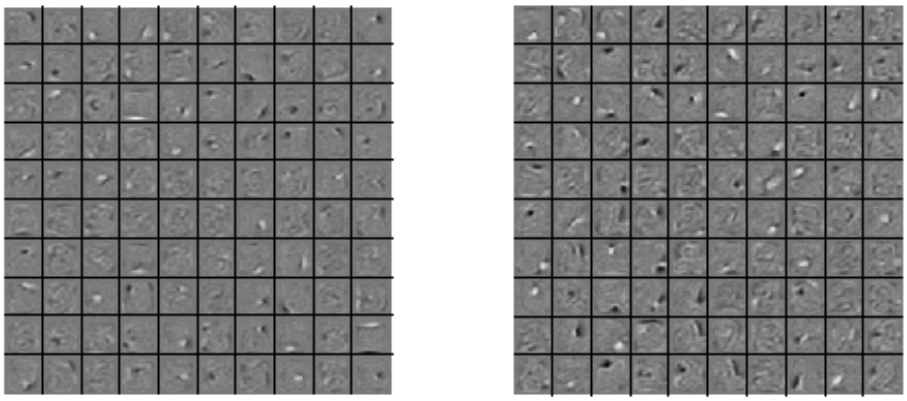


Figure 5.2: Features of the mixture model with two component RBMs trained on unlabeled images of handwritten two's and three's from MNIST. Those for the "two-RBM" are on the right. The features resemble what one would get if two RBMs are trained separately, one on twos only and the other on threes only. But this requires knowing the labels of the images.

5.4 Results

We apply the implicit mixture of RBMs to two datasets, MNIST and NORB. Details of both of these datasets can be found in section A.2. We use MNIST mainly as a sanity check, and most of our results are for the more difficult NORB dataset.

Evaluation method: Since computing the exact partition function of an RBM is intractable, it is not possible to directly evaluate the quality of our mixture model's fit to the data, e.g., by computing the log probability of a test set under the model. Recently it was shown that Annealed Importance Sampling can be used to tractably approximate the partition function of an RBM (Salakhutdinov and Murray [2008]). While this is an attractive option to consider in future work, for now we use the less direct but computationally cheaper approach of evaluating the model by using it in a classification task.

A reasonable evaluation criterion for a mixture modeling algorithm is that it should be able to find clusters that are mostly 'pure' with respect to class labels. That is, the set of data vectors for which a particular mixture component has high responsibilities should have the same class label. So it should be possible to accurately predict the class label of a given data vector from the responsibilities of the different mixture components for that vector. Once a mixture model is fully trained, we evaluate it by training a classifier that takes as input the responsibilities of the mixture components for a data vector and predicts its class label. The goodness of the mixture model is measured by the test set prediction accuracy of this classifier.

5.4.1 Results for MNIST

Before attempting to learn a good mixture model of the whole MNIST dataset, we tried two simpler modeling tasks. First, we fitted an implicit mixture of two RBMs with 100 hidden units each to an unlabeled dataset consisting of 4,000 twos and 4,000 threes. As we hoped, almost all of the twos were modeled by one RBM and almost all of the threes by the other. Figure 5.2 shows the features learned by each RBM (the one that models twos is on the right). These features are sensible for detecting the kind of strokes that appear in images of handwritten twos and threes.

On 2042 held-out test cases of twos and threes, there were only 24 classification errors when an image was assigned the label of the most probable RBM. This compares very favorably with logistic

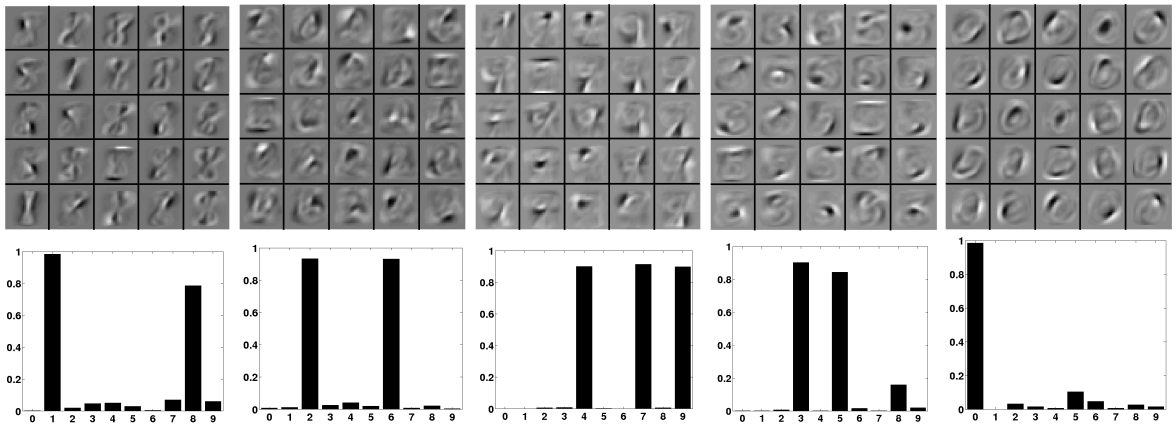


Figure 5.3: Features of the mixture model with five component RBMs trained on all ten classes of MNIST images.

regression which needs 8000 labels in addition to the images and gives 36 errors on the test set even when using a penalty on the squared weights whose magnitude is set using a validation set. Logistic regression also gives a good indication of the performance that could be expected from fitting a mixture of two Gaussians with a shared covariance matrix, because logistic regression is equivalent to fitting such a mixture discriminatively.

We then tried fitting an implicit mixture model with only five component RBMs, each with 25 hidden units, to the entire training set. We purposely make the model very small so that it is possible to visually inspect the features and the responsibilities of the component RBMs and understand what each component is modeling. This is meant to qualitatively confirm that the algorithm can learn a sensible clustering of the MNIST data. (Of course, the model will have poor classification accuracy as there are more classes than clusters, so it will merge multiple classes into a single cluster.) The features of the component RBMs are shown in figure 5.3 (top row). The plots in the bottom row show the fraction of training images for each of the ten classes that are hard-assigned to each component. The learning algorithm has produced a sensible mixture model in that visually similar digit classes are combined under the same mixture component. For example, ones and eights require many similar features, so they are captured with a single RBM (leftmost in fig. 5.3). Similarly, images of fours, sevens, and nines are all visually similar, and they are modeled together by one RBM (middle of fig. 5.3).

We have also trained larger models with many more mixture components. As the number of components increase, we expect the model to partition the image space more finely, with the different components specializing on various sub-classes of digits. If they specialize in a way that respects the class boundaries, then their responsibilities for a data vector will become a better predictor of its class label.

The component RBMs use binary units both in the visible and hidden layers. The image dimensionality is 784 (28×28 pixels). We have tried various settings for the number of mixture components (from 20 to 120 in steps of 20) and a component's hidden layer size (50, 100, 200, 500). Classification accuracy increases with more components, until 80 components. Additional components give slightly worse results. The hidden layer size is set to 100, but 200 and 500 also produce similar accuracies. Out of the 60,000 training images in MNIST, we use 50,000 to train the mixture model and the classifier, and the remaining 10,000 as a validation set for early stopping. The final models are then tested on a separate test set of 10,000 images.

Once the mixture model is trained, we train a logistic regression classifier to predict the class la-

Logistic regression classifier input	% Test error
Unnormalized responsibilities	3.36%
Pixels	7.28%

Table 5.1: MNIST test set error rates when a logistic regression classifier is trained on the unnormalized responsibilities of the implicit mixture model versus on the raw pixels.

bel from the responsibilities[2]. It has as many inputs as there are mixture components, and a ten-way softmax over the class labels at the output. With 80 components, there are only $80 \cdot 10 + 10 = 810$ parameters in the classifier (including the 10 output biases). In our experiments, classification accuracy is consistently higher when *unnormalized* responsibilities are used as the classifier input, instead of the actual responsibilities. These unnormalized values have no proper probabilistic interpretation, but they allow for better classification, so we use them in all our experiments.

Table 5.1 shows the classification error rate of the resulting classifier on the MNIST test set. As a simple baseline comparison, we train a logistic regression classifier that predicts the class label from the raw pixels. This classifier has 7850 parameters and yet the mixture-based classifier has less than half the error rate. The unnormalized responsibilities therefore contain a significant amount of information about the class labels of the images, which indicates that the implicit mixture model has learned clusters that mostly agree with the class boundaries, even though it is not given any class information during training.

5.4.2 Results for NORB

NORB is a much more difficult dataset than MNIST because the images are of very different classes of 3D objects (instead of 2D patterns) shown from different viewpoints and under various lighting conditions. The pixels are also no longer binary-valued, but instead span the grayscale range $[0, 255]$. So binary units are no longer appropriate for the visible layer of the component RBMs. Gaussian visible units have previously been shown to be effective for modeling grayscale images (Hinton and Salakhutdinov [2006]), and therefore we use them here. See Hinton and Salakhutdinov [2006] for details about Gaussian units (or see equation 4.6 in the previous chapter for the energy function of an RBM with Gaussian visible units with unit variance and binary hidden units). As in that paper, the variance of the units is fixed to 1.

Empirically, learning an RBM with Gaussian visible units has been found to require a greater number of weight updates than an RBM with binary visible units. This problem becomes even worse in our case since a large number of RBMs have to be trained simultaneously. We avoid it by first training a single RBM with Gaussian visible units and binary hidden units on the raw pixel data, and then treating the activities of its hidden layer as pre-processed data to which the implicit mixture model is applied. Since the hidden layer activities of the pre-processing RBM are binary, the mixture model can now be trained efficiently with binary units in the visible layer[3]. Once trained, the low-level RBM acts as a fixed pre-processing step that converts the raw grayscale images into binary vectors. Its parameters are not modified further when training the mixture model. Figure 5.4 shows the components of the complete

[2]Note that the mixture model parameters are kept fixed when training the classifier, so the learning of the mixture model is entirely unsupervised.

[3]We actually use the real-valued probabilities of the hidden units as the data, and we also use real-valued probabilities for the reconstructions. On other tasks, the learning gives similar results using binary values sampled from these real-valued probabilities but is slower.

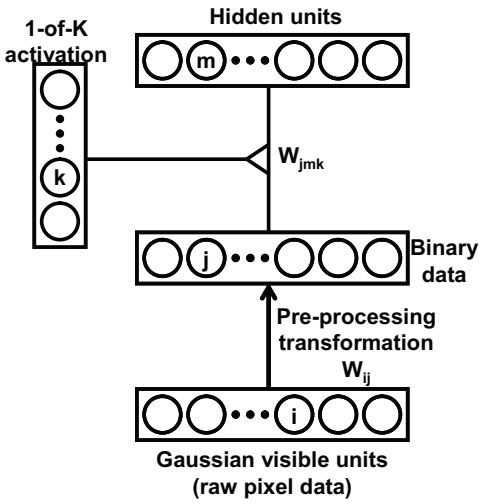


Figure 5.4: Implicit mixture model used for MNORB. A pre-processing transformation converts the raw pixels, represented by Gaussian visible units, into the activation probabilities of stochastic binary units. The mixture model is trained on the real-valued probabilities rather than binary activities of the units.

model.

A difficulty with training the implicit mixture model (or any other mixture model) on NORB is that the 'natural' clusters in the dataset correspond to the six lighting conditions instead of the five object classes. The objects themselves are small (in terms of area) relative to the background, while lighting affects the entire image. Any clustering signal provided by the object classes will be weak compared to the effect of large lighting changes. So we simplify the dataset slightly by normalizing the lighting variations across images. Each image is multiplied by a scalar such that all images have the same average pixel value. This crude normalization significantly reduces the interference of the lighting on the mixture learning[4]. Finally, to speed up experiments, we subsample the images from 96×96 to 32×32 and use only one image of the stereo pair. We refer to this dataset as 'Modified NORB' or 'MNORB'. It contains 24,300 training images and an equal number of test images. From the training set, 4,300 are set aside as a validation set for early stopping.

We use 2000 binary hidden units for the preprocessing RBM, so the input dimensionality of the implicit mixture model is 2000. We have tried many different settings for the number of mixture components and the hidden layer size of the components. The best classification results are given by 100 components, each with 500 hidden units. This model has about $100 \cdot 500 \cdot 2000 = 10^8$ parameters, and takes about 10 days to train on an Intel Xeon 3Ghz processor.

Table 5.4.2 shows the test set error rates for a logistic regression classifier trained on various input representations. As mentioned earlier, Mixture of Factor Analyzers (MFA) is similar to the implicit mixture of RBMs in that it also learns a clustering while simultaneously learning a latent representation per cluster component. But it is a directed model based on linear-Gaussian representations, and it can be learned tractably by maximizing likelihood with EM. We train MFA on the raw pixel data of MNORB. The MFA model that gives the best classification accuracy (shown in table 5.4.2) has 100 component Factor Analyzers with 100 factors each. (Note that simply making the number of learnable parameters equal is not enough to match the capacities of the different models because RBMs use binary latent representations, while FAs use continuous representations. So it is not easy to strictly control for

[4]The normalization does not completely remove lighting information from the data. A logistic regression classifier can still predict the lighting label with 18% test set error when trained and tested on normalized images.

Logistic regression classifier input	% Test error
Unnormalized responsibilities computed by the implicit mixture of RBMs	14.65%
Probabilities computed by the transformation W_{ij} in fig 5.4 (i.e. the *pre-processed representation*)	16.07%
Raw pixels	20.60%
Unnormalized responsibilities of an MFA model trained on the pre-processed representation in fig 5.4	22.65%
Unnormalized responsibilities of an MFA model trained on raw pixels	24.57%
Unnormalized responsibilities of a Mixture of Bernoullis model trained on the pre-processed representation in fig 5.4	28.53%

Table 5.2: MNORB Test set error rates for a logistic regression classifier with different types of input representations.

capacity when comparing these models.)

A mixture of multivariate Bernoulli distributions (see *e.g.* section 9.3.3 of Bishop [2006]) is similar to an implicit mixture model whose component RBMs have no hidden units and only visible biases as trainable parameters. The differences are that a Bernoulli mixture is a directed model, it has explicitly parameterized mixing proportions, and maximum likelihood learning with EM is tractable. We train this model with 100 components on the activation probabilities of the preprocessing RBM's hidden units. The classification error rate for this model is shown in table 5.4.2.

These results show that the implicit mixture of RBMs has learned clusters that reflect the class structure in the data. By the classification accuracy criterion, the implicit mixture is also better than MFA. The results also confirm that the lack of explicitly parameterized mixing proportions does not prevent the implicit mixture model from discovering interesting cluster structure in the data.

5.5 Conclusions

We have presented a tractable formulation of a mixture of RBMs. That such a formulation is even possible is a surprising discovery. The key insight here is that the mixture model can be cast as a third-order RBM, provided we are willing to abandon explicitly parameterized mixing proportions. Then it can be learned tractably using contrastive divergence. As future work, it would be interesting to explore whether these ideas can be extended to modeling time-series data.

Chapter 6

3D Object Recognition with Deep Belief Nets

We introduce a new type of top-level model for Deep Belief Nets and evaluate it on a 3D object recognition task. The top-level model is a third-order Boltzmann machine, trained using a hybrid algorithm that combines both generative and discriminative gradients. Performance is evaluated on the NORB database (*normalized-uniform* version), which contains stereo-pair images of objects under different lighting conditions from different viewpoints. Our model achieves 6.5% error on the test set, which is close to the best published result for NORB (5.9%) using a convolutional neural net that has built-in knowledge of translation invariance. It substantially outperforms shallow models such as SVMs (11.6%). DBNs are especially suited for semi-supervised learning, and to demonstrate this we consider a modified version of the NORB recognition task in which additional *unlabeled* images are created by applying small translations to the images in the database. With the extra unlabeled data (and the same amount of labeled data as before), our model achieves 5.2% error.

6.1 Introduction

Recent work on DBNs (Larochelle et al. [2007], Lee et al. [2009]) has shown that it is possible to learn multiple layers of nonlinear features that are useful for object classification without requiring labeled data. The features are trained one layer at a time as an RBM using CD, or as some form of autoencoder (Vincent et al. [2008], Ranzato et al. [2007]), and the feature activations learned by one module become the data for training the next module. After a pre-training phase that learns layers of features which are good at modeling the statistical structure in a set of unlabeled images, supervised backpropagation can be used to fine-tune the features for classification (Hinton and Salakhutdinov [2006]). Alternatively, classification can be performed by learning a top layer of features that models the joint density of the class labels and the highest layer of unsupervised features (Hinton et al. [2006]). These unsupervised features (plus the class labels) then become the penultimate layer of the deep belief net (Hinton et al. [2006]).

Early work on deep belief nets was evaluated using the MNIST dataset of handwritten digits (Hinton et al. [2006]) which has the advantage that a few million parameters are adequate for modeling most of the structure in the domain. For 3D object classification, however, many more parameters are probably required to allow a DBN with no prior knowledge of spatial structure to capture all of the variations caused by lighting and viewpoint. It is not yet clear how well DBNs perform at 3D object classification when compared with shallow techniques such as SVMs (Vapnik [1998], DeCoste and Scholkopf [2002]) or deep discriminative techniques like convolutional neural networks (LeCun et al. [1998]).

In this chapter, we describe a better type of top-level model for deep belief nets and show that if the top-level model is trained using a combination of generative and discriminative gradients (Hinton [2006], Kelm et al. [2006], Larochelle and Bengio [2008]) there is no need to backpropagate discriminative gradients to fine-tune the earlier layers of features. We evaluate the model on NORB (LeCun et al. [2004]), which is a carefully designed object recognition task that requires generalization to novel object instances under varying lighting conditions and viewpoints. Our model significantly outperforms SVMs, and it also outperforms convolutional neural nets when given additional *unlabeled* data produced by small translations of the training images.

We use RBMs trained with one-step contrastive divergence as our basic module for learning layers of features. These are fully described elsewhere (Hinton et al. [2006], Bengio et al. [2007]) and the reader is referred to those sources for details.

6.2 A Third-Order Restricted Boltzmann Machine as the Top-Level Model

Until now, the only top-level model that has been considered for a DBN is an RBM with two types of visible units (one for the label, another for the penultimate feature vector) and a hidden layer, with bipartite connections between the visible and hidden layers. We now consider an alternative model for the top-level joint distribution in which the class label multiplicatively interacts with *both* the penultimate layer units and the hidden units to determine the energy of a full configuration. It is the same as the implicit mixture model introduced in the previous chapter, except now the discrete (cluster) label variable is no longer hidden, and the number of components in the model is fixed to be the number of object classes. Changing the label variable from hidden to observed affects the inference and learning procedures. As explained before, the model is a Boltzmann machine with three-way cliques, each containing a penultimate layer unit v_i, a hidden unit h_j, and a label unit l_k. See the figure in the previous chapter (figure 5.1) for a summary of the architecture.

We quickly review the energy function and the probability model of the third-order RBM. Consider the case where the components of $\mathbf{v}$ and $\mathbf{h}$ are stochastic binary units, and $\mathbf{l}$ is a discrete variable with K states represented by 1-of-K encoding. The model can be defined in terms of its energy function

$$E(\mathbf{v}, \mathbf{h}, \mathbf{l}) = -\sum_{i,j,k} W_{ijk} v_i h_j l_k, \tag{6.1}$$

where W_{ijk} is a learnable scalar parameter. (We omit bias terms in all expressions for clarity.) The probability of a full configuration $\{\mathbf{v}, \mathbf{h}, \mathbf{l}\}$ is then

$$P(\mathbf{v}, \mathbf{h}, \mathbf{l}) = \frac{\exp(-E(\mathbf{v}, \mathbf{h}, \mathbf{l}))}{Z}, \tag{6.2}$$

where $Z = \sum_{\mathbf{v}', \mathbf{h}', \mathbf{l}'} \exp(-E(\mathbf{v}', \mathbf{h}', \mathbf{l}'))$ is the partition function. Marginalizing over $\mathbf{h}$ gives the distribution over $\mathbf{v}$ and $\mathbf{l}$ alone.

The main difference between the new top-level model and the bipartite one used in earlier DBNs is that now the class label multiplicatively modulates how the visible and hidden units contribute to the energy of a full configuration. If the label's k^{th} unit is 1 (and the rest are 0), then the k^{th} slice of the 3D array determines the energy function. In the case of soft activations (i.e. more than one label has non-zero probability), a weighted blend of the array's slices specifies the energy function. The earlier top-level (RBM) model limits the label's effect to changing the biases into the hidden units, which modifies only how the hidden units contribute to the energy of a full configuration. There is no direct interaction between the label and the visible units. Introducing direct interactions among all three sets of variables allows the model to learn features that are dedicated to each class. This is a useful property

when the object classes have substantially different appearances that require very different features to describe. Unlike an RBM, the model structure is not bipartite, but it is still "restricted" in the sense that there are no direct connections between two units of the same type.

Note that the third-order model is not the same as training a separate RBM for each class. A collection of independently trained class-specific RBMs represents the distribution $P(\mathbf{v}|\mathbf{l})$, while the third-order model represents the joint distribution $P(\mathbf{v},\mathbf{l}) = P(\mathbf{v}|\mathbf{l})P(\mathbf{l})$. This distinction will become important later when we consider the hybrid generative-discriminative learning algorithm.

6.2.1 Inference

The distributions that we would like to be able to infer are $P(\mathbf{l}|\mathbf{v})$ (to classify an input), and $P(\mathbf{v},\mathbf{l}|\mathbf{h})$ and $P(\mathbf{h}|\mathbf{v},\mathbf{l})$ (for CD learning). Fortunately, all three distributions are tractable to sample from exactly. The simplest case is $P(\mathbf{h}|\mathbf{v},\mathbf{l})$. Once $\mathbf{l}$ is observed, the model reduces to an RBM whose parameters are the k^{th} slice of the 3D parameter array. As a result $P(\mathbf{h}|\mathbf{v},\mathbf{l})$ is a factorized distribution that can be sampled exactly.

For a third-order RBM model with N_v visible units, N_h hidden units and N_l class labels, the distribution $P(\mathbf{l}|\mathbf{v})$ can be exactly computed in $O(N_v N_h N_l)$ time. This result follows from two observations: 1) setting $l_k = 1$ reduces the model to an RBM defined by the k^{th} slice of the array, and 2) the negative log probability of $\mathbf{v}$, up to an additive constant, under this RBM is the free energy:

$$F_k(\mathbf{v}) = -\sum_{j=1}^{N_h} \log(1 + \exp(\sum_{i=1}^{N_v} W_{ijk} v_i)). \tag{6.3}$$

The idea is to first compute $F_k(\mathbf{v})$ for each setting of the label, and then convert them to a discrete distribution by taking the softmax of the negative free energies:

$$P(\mathbf{l}_k = 1|\mathbf{v}) = \frac{\exp(-F_k(\mathbf{v}))}{\sum_{k=1}^{N_l} \exp(-F_k(\mathbf{v}))}. \tag{6.4}$$

Equation 6.3 requires $O(N_v N_h)$ computation, which is repeated N_l times for a total of $O(N_v N_h N_l)$ computation.

We can use the same method to compute $P(\mathbf{l}|\mathbf{h})$. Simply switch the role of $\mathbf{v}$ and $\mathbf{h}$ in equation 6.3 to compute the free energy of $\mathbf{h}$ under the k^{th} RBM. (This is possible since the model is symmetric with respect to $\mathbf{v}$ and $\mathbf{h}$.) Then convert the resulting N_l free energies to the probabilities $P(\mathbf{l}_k = 1|\mathbf{h})$ with the softmax function.

Now it becomes possible to exactly sample $P(\mathbf{v},\mathbf{l}|\mathbf{h})$ by first sampling $\tilde{l} \sim P(\mathbf{l}|\mathbf{h})$. Suppose $\tilde{l_k} = 1$. Then the model reduces to its k^{th}-slice RBM from which $\tilde{\mathbf{v}} \sim P(\mathbf{v}|\mathbf{h},\tilde{\mathbf{l}}_k = 1)$ can be easily sampled. The final result $\{\tilde{\mathbf{v}},\tilde{\mathbf{l}}\}$ is an unbiased sample from $P(\mathbf{v},\mathbf{l}|\mathbf{h})$.

6.2.2 Learning

Given a set of N labeled training cases $\{(\mathbf{v}_1,\mathbf{l}_1),(\mathbf{v}_2,\mathbf{l}_2),...,(\mathbf{v}_N,\mathbf{l}_N)\}$, we want to learn the 3D parameter array W for the third-order model. When trained as the top-level model of a DBN, the visible vector $\mathbf{v}$ is a penultimate layer feature vector. We can also train the model directly on images as a shallow model, in which case $\mathbf{v}$ is an image (in row vector form). In both cases the label $\mathbf{l}$ represents the N_l object categories using 1-of-N_l encoding. For the same reasons as in the case of an RBM, maximum likelihood learning is intractable here as well, so we rely on Contrastive Divergence learning instead. CD was originally formulated in the context of the RBM and its bipartite architecture, but here we extend it to the non-bipartite architecture of the third-order RBM model.

An unbiased estimate of the maximum likelihood gradient can be computed by running a Markov chain that alternatively samples $P(\mathbf{h}|\mathbf{v},\mathbf{l})$ and $P(\mathbf{v},\mathbf{l}|\mathbf{h})$ until it reaches equilibrium. Contrastive divergence uses the parameter updates given by three half-steps of this chain, with the chain initialized from a training case (rather than a random state). As explained in section 6.2.1, both of these distributions are easy to sample from. The steps for computing the CD parameter updates are summarized below:

Contrastive divergence learning of $P(\mathbf{v},\mathbf{l})$:

1. Given a labeled training pair $\{\mathbf{v}^+, \mathbf{l}_k^+ = 1\}$, sample $\mathbf{h}^+ \sim P(\mathbf{h}^+|\mathbf{v}^+, \mathbf{l}_k^+ = 1)$.
2. Compute the outer product $D_k^+ = \mathbf{v}^+\mathbf{h}^{+T}$.
3. Sample $\{\mathbf{v}^-, \mathbf{l}^-\} \sim P(\mathbf{v},\mathbf{l}|\mathbf{h}^+)$. Let m be the index of the component of $\mathbf{l}^-$ set to 1.
4. Sample $\mathbf{h}^- \sim P(\mathbf{h}|\mathbf{v}^-, \mathbf{l}_m^- = 1)$.
5. Compute the outer product $D_m^- = \mathbf{v}^-\mathbf{h}^{-T}$.

Let $W_{\cdot,\cdot,k}$ denote the $N_h \times N_v$ matrix of parameters corresponding to the k^{th} slice along the label dimension of the 3D array. Then the CD update for $W_{\cdot,\cdot,k}$ is:

$$\Delta W_{\cdot,\cdot,k} = D_k^+ - D_k^-, \tag{6.5}$$

$$W_{\cdot,\cdot,k} \leftarrow W_{\cdot,\cdot,k} + \eta \Delta W_{\cdot,\cdot,k}, \tag{6.6}$$

where η is a learning rate parameter. Typically, the updates computed from a "mini-batch" of training cases (a small subset of the entire training set) are averaged together into one update and then applied to the parameters.

6.3 Combining Gradients for Generative and Discriminative Models

In practice the Markov chain used in the learning of $P(\mathbf{v},\mathbf{l})$ can suffer from slow mixing. In particular, the label $\mathbf{l}^-$ generated in step 3 above is unlikely to be different from the true label $\mathbf{l}^+$ of the training case used in step 1. The chain has a tendency to stay "stuck" on the same state for the label variable, even if it is run for a large number of steps. This is because in the positive phase the hidden unit activities are inferred with the label clamped to its true value. So the hidden activities contain information about the true label, which gives it an advantage over the other labels. We have observed that empirically, the label rarely changes from its original setting even after many Markov chain steps.

Consider the extreme case where we initialize the Markov chain with a training pair $\{\mathbf{v}^+, \mathbf{l}_k^+ = 1\}$ and the label variable *never* changes from its initial state during the chain's entire run. In effect, the model that ends up being learned is a class-conditional generative distribution $P(\mathbf{v}|\mathbf{l}_k = 1)$, represented by the k^{th} slice RBM. The parameter updates are identical to those for training N_l independent RBMs, one per class, with only the training cases of each class being used to learn the RBM for that class. Note that this is very different from the model in section 6.2: here the energy functions implemented by the class-conditional RBMs are learned independently and their energy units are not commensurate with each other.

Alternatively, we can optimize the *same* set of parameters to represent yet another distribution, $P(\mathbf{l}|\mathbf{v})$. The advantage in this case is that the *exact* gradient needed for maximum likelihood learning, $\partial log P(\mathbf{l}|\mathbf{v})/\partial W$, can be computed in $O(N_v N_h N_l)$ time. The gradient expression can be derived with some straightforward differentiation of equation 6.4. The disadvantage is that it cannot make use of unlabeled data. Also, as the results show, learning a purely discriminative model at the top level of a DBN gives much worse performance.

However, now a new way of learning $P(\mathbf{v}, \mathbf{l})$ becomes apparent: we can optimize the parameters by using *a weighted sum of the gradients* for $\log P(\mathbf{v}|\mathbf{l})$ and $\log P(\mathbf{l}|\mathbf{v})$. As explained below, this approach 1) avoids the slow mixing of the CD learning for $P(\mathbf{v}, \mathbf{l})$, and 2) allows learning with both labeled and unlabeled data. In our experiments, a model trained with this hybrid learning algorithm has the highest classification accuracy, beating both a generative model trained using CD as well as a purely discriminative model. The main steps of the algorithm are listed below.

Hybrid learning algorithm for $P(\mathbf{v}, \mathbf{l})$:
Let $\{\mathbf{v}^+, \mathbf{l}_k^+ = 1\}$ be a labeled training case.
Generative update: CD learning of $P(\mathbf{v}|\mathbf{l})$

1. Sample $\mathbf{h}^+ \sim P(\mathbf{h}|\mathbf{v}^+, \mathbf{l}_k^+ = 1)$.
2. Compute the outer product $D_k^+ = \mathbf{v}^+\mathbf{h}^{+T}$.
3. Sample $\mathbf{v}^- \sim P(\mathbf{v}|\mathbf{h}^+, \mathbf{l}_k^+ = 1)$.
4. Sample $\mathbf{h}^- \sim P(\mathbf{h}|\mathbf{v}^-, \mathbf{l}_k^+ = 1)$.
5. Compute the outer product $D_k^- = \mathbf{v}^-\mathbf{h}^{-T}$.
6. Compute update $\Delta W_{\cdot,\cdot,k}^g = D_k^+ - D_k^-$.

Discriminative update: ML learning of $P(\mathbf{l}|\mathbf{v})$

1. Compute $\log P(\mathbf{l}_c = 1|\mathbf{v}^+)$ for $c \in \{1, ..., N_l\}$.
2. Using the result from step 1 and the true label $\mathbf{l}_k^+ = 1$, compute the update $\Delta W_{\cdot,\cdot,k}^d = \partial \log P(\mathbf{l}|\mathbf{v})/\partial W_{\cdot,\cdot,c}$ for $c \in \{1, ..., N_l\}$.

The two types of update for the c^{th} slice of the array $W_{\cdot,\cdot,c}$ are then combined by a weighted sum:

$$W_{\cdot,\cdot,c} \leftarrow W_{\cdot,\cdot,c} + \eta(\Delta W_{\cdot,\cdot,c}^g + \lambda \Delta W_{\cdot,\cdot,c}^d), \tag{6.7}$$

where λ is a parameter that sets the relative weighting of the generative and discriminative updates, and η is the learning rate. As before, the updates from a mini-batch of training cases can be averaged together and applied as a single update to the parameters.

Note that the generative part in the above algorithm is simply CD learning of the RBM for the k^{th} class. The earlier problem of slow mixing does not appear in the hybrid algorithm because the chain in the generative part does not involve sampling the label.

Semi-supervised learning: The hybrid learning algorithm can also make use of *unlabeled* training cases by treating their labels as missing inputs. The model first infers the missing label by sampling $P(\mathbf{l}|\mathbf{v}_u)$ for an unlabeled training case $\mathbf{v}_u$. The generative update is then computed by treating the inferred label as the true label. (The discriminative update will always be zero in this case.) Therefore the unlabeled training cases contribute an extra generative term to the parameter update.

6.3.1 Cost Function of the Hybrid Algorithm

For the *discriminative* version of the third-order RBM, the 3D parameter array W is updated by the negative gradient of the training set's negative log-likelihood:

$$L_D(W) = -\sum_{n=1}^{N} \log(P(\mathbf{l}_n|\mathbf{v}_n, W)), \tag{6.8}$$

where $(\mathbf{v}_n, \mathbf{l}_n)$ is a labeled training case and $P(\mathbf{l}_n|\mathbf{v}_n, W)$ is given by 6.4. The exact gradient of equation 6.8 can be computed tractably. For the *class-conditional, generative* version, the negative log-likelihood expression is:

$$L_{CG}(W) = -\sum_{n=1}^{N} \log(P(\mathbf{v}_n|\mathbf{l}_n, W)), \tag{6.9}$$

where $P(\mathbf{v}_n|\mathbf{l}_n, W)$ is the probability of $\mathbf{v}_n$ under its corresponding class-specific RBM. In this case we can use the CD approximation to the gradient of equation 6.9 to update W.

The cost function optimized by the hybrid learning algorithm is

$$L_{CG}(W) + \lambda L_D(W), \tag{6.10}$$

where λ is the user-set parameter mentioned before. In our experiments, we have tried values for λ from 0.1 to 20. The best classification results are achieved for $\lambda = 5$, but nearby values also give similar results, so the algorithm does not appear to be extremely sensitive to its exact value. Note that the value of λ is not a good indicator of the relative size of the contributions made by the two types of gradients. The generative gradient tends to have a much bigger L_2 norm than the discriminative one (in some experiments we have seen two orders of magnitude difference). Setting λ to 5 therefore does not mean that the discriminative gradient makes a contribution 5 times bigger than the generative gradient to the weight update.

6.3.2 Interpretation of the Hybrid Algorithm

Hybrid learning resembles pseudo-likelihood learning: instead of maximizing $P(\mathbf{v}, \mathbf{l})$ directly, the optimization relies on the two conditional distributions $P(\mathbf{v}|\mathbf{l})$ and $P(\mathbf{l}|\mathbf{v})$. So one informal interpretation of the algorithm is that it is still approximately learning the joint distribution.

A more rigorous interpretation can be found in Bishop and Lasserre [2007]. They consider a cost function that is slightly different from equation 6.10, but it can be shown that their analysis applies to the hybrid algorithm's cost function as well. They show that the kind of blended learning done by the hybrid algorithm can be interpreted as learning a joint distribution model. The main effect of the blending is to compensate for the model mis-specification problem suffered by the original generative model.

6.4 Sparsity

Discriminative performance is improved by using binary features that are only rarely active. Sparse activities are achieved by specifying a desired probability of being active, $p << 1$, and then adding an additional penalty term that encourages an exponentially decaying average, q, of the actual probability of being active to be close to p. The natural error measure to use is the cross entropy between the desired and actual distributions: $p \log q + (1-p)\log(1-q)$. For logistic units this has a simple derivative of $p-q$ with respect to the total input to a unit. This derivative is used to adjust both the bias and the incoming weights of each hidden unit. We tried various values for p and 0.1 worked well. In addition to specifying

p it is necessary to specify how fast the estimate of q decays. We used $q_{new} = 0.9 * q_{old} + 0.1 * q_{current}$ where $q_{current}$ is the average probability of activation for the current mini-batch of 100 training cases. It is also necessary to specify how strong the penalty term should be, but this is easy to set empirically. We multiply the penalty gradient by a coefficient that is chosen to ensure that, on average, q is close to p but there is still significant variation among the q values for different hidden units. This prevents the penalty term from dominating the learning. One added advantage of this sparseness penalty is that it revives any hidden units whose average activities are much lower than p.

6.5 Evaluating DBNs on the NORB Object Recognition Task

Pre-processing: See section A.2 for details about NORB. A single training (and test) case in NORB is a stereo-pair of grayscale images, each of size 96×96. To speed up experiments, we reduce dimensionality by using a "foveal" image representation. The central 64×64 portion of an image is kept at its original resolution. The remaining 16 pixel-wide ring around it is compressed by replacing non-overlapping square blocks of pixels with the average value of a block. We split the ring into four smaller ones: the outermost ring has 8×8 blocks, followed by a ring of 4×4 blocks, and finally two innermost rings of 2×2 blocks. The foveal representation reduces the dimensionality of a stereo-pair from 18432 to 8976. All our models treat the stereo-pair images as 8976-dimensional vectors[1]. We do not use the lighting normalization that was used in the previous chapter when training the implicit mixture model on NORB.

A crucial property of the NORB dataset is that the object instances are split into two disjoint sets to define the training and test sets. Therefore at test time the model must generalize to new instances of the same object classes.

6.5.1 Training Details

Model architecture: The two main decisions to make when training DBNs are the number of hidden layers to greedily pre-train and the number of hidden units to use in each layer. To simplify the experiments we constrain the number of hidden units to be the same at all layers (including the top-level model). We have tried hidden layer sizes of 2000, 4000, and 8000 units. We have also tried models with two, one, or no greedily pre-trained hidden layers. To avoid clutter, only the results for the best settings of these two parameters are given. The best classification results are given by the DBN with one greedily pre-trained sparse hidden layer of 4000 units (regardless of the type of top-level model).

A DBN trained on the pre-processed input with one greedily pre-trained layer of 4000 hidden units and a third-order model on top of it, also with 4000 hidden units, has roughly 116 million learnable parameters in total. This is roughly two orders of magnitude more parameters than some of the early DBNs trained on the MNIST images (Hinton et al. [2006], Larochelle et al. [2007]). Training such a model in Matlab on an Intel Xeon 3GHz machine takes almost two weeks. See a recent paper by Raina et al. [2009] that uses GPUs to train a deep model with roughly the same number of parameters much more quickly. We put Gaussian units at the lowest (pixel) layer of the DBN.

Early stopping: Previous papers that report results on NORB (LeCun et al. [2004], Bengio and LeCun [2007]) do not hold out a subset of the training images for early stopping. They train models on the full training set and report classification accuracy on the test set. To make our results comparable, we also train the top-level model with the full training set. We use a subset of the *test* images for early stopping, and compute classification accuracy on the remaining test cases. To estimate the accuracy on the *entire*

[1]Knowledge about image topology is used only along the (mostly empty) borders, and not in the central portion that actually contains the object.

test set, we train the model twice, each time using a different, non-overlapping subset of the test data for early stopping and computing the accuracy on the remainder. The mean of the two estimates is reported as the accuracy for the full set. The validation set and the partial test set we use still split the object instances disjointly, so at test time the model must still generalize to new instances.

6.6 Results

The results are presented in three parts: part 1 compares deep models to shallow ones, all trained using CD. Part 2 compares CD to the hybrid learning algorithm for training the top-level model of a DBN. Part 3 compares DBNs trained with and without unlabeled data, using either CD or the hybrid algorithm at the top level. For comparison, table 6.1 lists results for some discriminative models on the normalized-uniform NORB test set (without any pre-processing). The results for our DBN models range from 11.9% to 5.2%.

Model	**Error**
Logistic regression	19.6%
kNN (k=1) (LeCun et al. [2004])	18.4%
Gaussian kernel SVM (Bengio and LeCun [2007])	11.6%
Convolutional neural net (Bengio and LeCun [2007])	6.0%
Convolutional net + SVM (Bengio and LeCun [2007])	5.9%

Table 6.1: Test set error rates for discriminative models on normalized-uniform NORB without any pre-processing.

6.6.1 Deep vs. Shallow Models Trained with CD

We consider here DBNs with one greedily pre-trained layer. Its shallow counterpart trains the top-level model directly on the pixels (using Gaussian visible units[2]), with no pre-trained layers in between. Using CD as the learning algorithm (for both greedy pre-training and at the top-level) with the two types of top-level models gives us four possibilities to compare. The test error rates for these four models(see table 6.2) show that one greedily pre-trained layer reduces the error substantially, even without any subsequent fine-tuning of the pre-trained layer.

Model	**RBM with label unit**	**Third-order Restricted Boltzmann Machine**
Shallow	22.8%	20.8%
Deep	11.9%	7.6%

Table 6.2: NORB test set error rates for deep and shallow models trained using CD with two types of top-level models.

The third-order RBM model outperforms the standard RBM top-level model when they both have the *same number of hidden units*, but a better comparison might be to match the number of *parameters* by increasing the hidden layer size of the standard RBM model by five times (i.e. 20000 hidden units). We have tried training such an RBM, but the error rate is worse than the RBM with 4000 hidden units.

[2]When training the shallow model with Gaussian visible units, the free energy expression in equation 6.3 (for binary visible units) must be changed appropriately. The new expression is given by equation 4.7.

6.6.2 Hybrid vs. CD Learning for the Top-level Model

We now compare the two alternatives for training the top-level model of a DBN. There are four possible combinations of top-level models and learning algorithms, and table 6.3 lists their error rates. All these DBNs share the same greedily pre-trained first layer – only the top-level model differs among them.

	Top-level Model	
Learning algorithm	RBM with label unit	Third-order Restricted Boltzmann machine
CD	11.9%	7.6%
Hybrid	10.4%	6.5%

Table 6.3: NORB test set error rates for top-level models trained using CD and the hybrid learning algorithms.

The lower error rates of hybrid learning are partly due to its ability to avoid the poor mixing of the label variable when CD is used to learn the joint density $P(v, l)$ and partly due to its greater emphasis on discrimination (but with regularization provided by also learning $P(v|l)$).

6.6.3 Semi-supervised vs. Supervised Learning

In this final part, we show how DBNs can take advantage of unlabeled data to improve the classification error. We create additional images from the original NORB training set by applying global translations of 2, 4, and 6 pixels in eight directions (two horizontal, two vertical and four diagonal directions) to the original stereo-pair images[3]. These "jittered" images are treated as extra *unlabeled* training cases that are combined with the original labeled cases to form a much larger training set.

Note that the jittered images could have been assigned the same label as the images they were created from. By treating them as unlabeled, the goal is to test whether improving the unsupervised, generative part of the learning alone can improve discriminative performance.

The unlabeled images here are specially designed to help the features become less sensitive to small shifts of the image. This is clearly artificial – we normally would not expect unlabeled data to provide information about a useful invariance so directly. Another way to simulate a semi-supervised learning task is to hold out the labels of some of the original training images and treat them as unlabeled. But this does not provide any extra data for pre-training the lower layer features since there is no distinction between labeled and unlabeled images in the greedy pre-training phase. Also, the original NORB training set is already relatively small (only 24,300 cases) and holding out labels will make it even smaller.

Instead of artificially crippling the discriminative part of the training with fewer labels, we add a separate unlabeled set to the full labeled set and measure how much improvement the generative part of the training can make given the best possible contribution by the discriminative part. This is a stricter assessment of the usefulness of the generative part. It also corresponds to what one would do in a real application, i.e. not hold out any labeled data and try to maximize classification accuracy.

There are two ways to use unlabeled data:

1. Use it for greedy pre-training of the lower layers only, and then train the top-level model as before, with only labeled data and the hybrid algorithm.

2. Use it for learning the top-level model as well, this time with the semi-supervised variant of the hybrid algorithm at the top-level.

Table 6.4 lists the results for both options.

[3]The same translation is applied to both images in the stereo-pair.

Top-level model (hybrid learning only)	Unlabeled jitter for greedy pre-training?	Unlabeled jitter at the top-level?	Error
RBM with label unit	No	No	10.4%
	No	Yes	10.5%
	Yes	No	9.0%
Third-order model	No	No	6.5%
	No	Yes	7.1%
	Yes	No	5.3%
	Yes	Yes	5.2%

Table 6.4: NORB test set error rates for DBNs trained with and without unlabeled data, and using the hybrid learning algorithm at the top-level.

The key conclusion from table 6.4 is that simply using more *unlabeled* training data in the unsupervised, greedy pre-training phase alone can significantly improve the classification accuracy of the DBN. It allows a third-order top-level model to reduce its error from 6.5% to 5.3%, which beats the current best published result for normalized-uniform NORB *without using any extra labeled data*. Using more unlabeled data also at the top level further improves accuracy, but only slightly, to 5.2%. When the unlabeled data is used only for training the top-level model (and not for pre-training the first layer), the results become *worse* than not using unlabeled data at all. It appears that the main effect of unlabeled data is to produce better features in the pre-trained layer. Without those improved features in the first layer, using unlabeled data to train the top-level model does not help.

Now consider a discriminative model at the top, representing the distribution $P(l|v)$. Unlike in the generative case, the exact gradient of the log-likelihood is tractable to compute. Table 6.5 shows the results of some discriminative models. These models use the same greedily pre-trained lower layer, learned with unlabeled jitter. The only difference is in how the parameters of the top-level are initialized.

Initialization of top-level parameters	Use jittered images as labeled?	Error
Random	No	13.4%
Random	Yes	7.1%
DBN top-level model with 5.2% error	Yes	5.0%

Table 6.5: NORB test set error rates for discriminative third-order models (i.e. third-order models trained to represent $P(\mathbf{l}|\mathbf{v})$) at the top level. They all use the same greedily pre-trained lower layer. The only difference among them is how the parameters are initialized.

We compare training the discriminative top-level model "from scratch" (random initialization) versus initializing its parameters to those of a generative model learned by the hybrid algorithm. We also compare the effect of using the jittered images as extra *labeled* cases. As mentioned before, it is possible to assign the jittered images the same labels as the original NORB images they are generated from, which expands the labeled training set by 25 times.

The bottom two rows of table 6.5 compare the accuracy of a discriminative third-order model with

and without generative pre-training. Generative pre-training significantly improves accuracy, but using the labels of the jittered images for the subsequent discriminative training only makes a small additional improvement. We have also noticed that fine-tuning the lower layer discriminatively leads to rapid overfitting which quickly makes the test error much higher than 5.0%.

6.7 Conclusions

Our results make a strong case for the use of generative modeling in object recognition. The main two points are:
1) Unsupervised, greedy, generative modeling can learn representations of the input images that support much more accurate object recognition than the raw pixel representation.
2) Including $P(\mathbf{v}|\mathbf{l})$ in the objective function for training the top-level model is much better than using $P(\mathbf{l}|\mathbf{v})$ alone, or learning $P(\mathbf{v}, \mathbf{l})$ using CD.

In future work we plan to factorize the third-order Boltzmann machine as described in Taylor and Hinton [2009] so that some of the top-level features can be shared across classes.

Chapter 7

Conclusions

7.1 Summary of the Thesis

The aim of the thesis was to show that generative/reconstructive models of images are useful for object recognition. The thesis demonstrated this idea in four different ways: 1) incorporating complex domain knowledge into the learning by inverting a synthesis model, 2) using the latent image representations of generative/reconstructive models for recognition, 3) optimizing a hybrid generative-discriminative loss function, and 4) creating additional synthetic data for better training of discriminative models. We end with a summary of the key results in the thesis.

Chapter 3 presented the breeder learning algorithm for learning an analysis model given a synthesis model and a set of images. We applied it to three different synthesis models which generate images of eyes, faces, and in chapter 4, handwritten digits. The results support two observations: 1) the input variables of a synthesis model can be a sensible reconstructive representation for the class of images it is designed to produce, and 2) empirically, breeder learning appears general enough to invert fairly different synthesis models. Once an analysis model is learned, it becomes possible to train a neural network to act as an emulator of the synthesis model. With such an emulator, inferring the synthesis inputs from an image becomes an iterative optimization problem that directly optimizes pixel reconstruction error. As the reconstruction results for face images showed, iterative inference gives better reconstruction error than one-step inference.

Chapter 4 applied breeder learning to a physically-based model of handwritten digits. The model consists of a simple mass-spring system in which the mass represents the tip of the pen and the four springs correspond to arm muscles. By varying the spring stiffnesses over time, the mass can be made to move along a particular trajectory. A digit image is then generated by applying ink on the trajectory. The resulting "motor program" representation's usefulness for classification is evaluated on the MNIST dataset in three different ways. The first method assigns to a test image a set of energies under class-specific models, and uses them as input to a logistic regression classifier. Even with orders of magnitude fewer discriminatively trained parameters, this method still outperforms baseline models such as a pixel-based logistic regression classifier or a fully-connected feedforward neural network. The second method creates more labeled training cases by corrupting the motor programs of the original MNIST training images. These synthetic examples turned out to significantly improve the performance of every type of classifier we tried – kNN, fully-connected feedforward neural networks and convolutional neural networks. The third method takes the features from the analysis models, puts them in a neural network classifier, and trains them discriminatively. When applied to a convolutional network and trained with additional synthetic examples, this method resulted in a top-5 error rate on the MNIST prediction task.

Chapter 5 looked at the problem of simultaneously learning a clustering of the data and a cluster-

specific latent representation. In particular, we proposed a new mixture model in which the component distributions are represented by RBMs. The traditional way of formulating such a model, with the mixing proportions as explicit parameters and the component distributions as RBMs, is intractable because the probability of a data vector under an RBM is intractable to compute. Surprisingly, this intractability is avoided by letting the mixing proportions be implicitly defined by the parameters of the model. The resulting mixture model can be trained with contrastive divergence. We showed that such a model is useful for learning cluster-specific features. By using the responsibilities of an image as input to a logistic regression classifier, it is possible to outperform the same classifier trained directly on pixels.

Chapter 6 presented an application of Deep Belief Networks to the NORB 3D object recognition task. We modified the original DBN model with a new type of top-level model and a new hybrid generative-discriminative algorithm for training it. It produced results that were close to the state-of-the-art performance given by a convolutional network that has knowledge about 2D image topology and local translation invariance hand-coded into it. When we used semi-supervised learning – an advantage our model has over purely discriminative models like convolutional networks – with unlabeled images created by applying small global shifts to the original training images, our model achieved the current best published result on the NORB dataset.

7.2 Limitations of Generative Models

While the thesis highlighted the advantages of generative/reconstructive models, they also have a number of disadvantages. At a broad level, we can identify three basic limitations:

1. **Approximating the true generative distribution poorly:** When we train a standard parametric generative model (e.g. RBM) on real-world images with a limited number of parameters and no built-in domain knowledge, it usually learns only a crude approximation to the real generative system that produced those images. For example, an RBM with say, 500 hidden units, trained on handwritten digit images is unlikely to figure out the physics of hand muscles, even if a very large training set is used. The problem is that a generic model with a small number of parameters is not expressive enough to represent the true image distribution.

   One way to deal with this problem is to build into the model, by hand, detailed domain knowledge about how the images are generated. Then the model may need only a small number of learnable parameters to approximate the real system well enough. Applications of breeder learning in chapters 3 and 4 are examples of this approach. Breeder learning is especially sensible when domain knowledge already exists in the form of a realistic graphics program, in which case the additional hand-engineering effort is small.

   Alternatively, the generative model's expressiveness can be increased by using a larger number of parameters and many layers of nonlinear features. This approach aims to keep the hand-designing effort to a minimum and rely as much as possible on the data itself to build the model. Deep Belief Nets and Deep Boltzmann Machines (Salakhutdinov and Hinton [2009]) are examples of such large-scale, deep, generative models. With more parameters, optimizing a nonconvex cost function for learning becomes more difficult. The main advance of DBNs is the greedy layer-wise learning strategy that is designed to find better solutions to the optimization problem than those found by simultaneously updating all model parameters throughout learning.

2. **Need for segmented input:** Generative models try to explain *all* the structure in their input, regardless of whether that structure is directly relevant for the object recognition task. This is a disadvantage when objects are unsegmented and appear superimposed on structured background

(e.g. faces in uncontrolled scenes collected off the web). A generative model wastes capacity trying to model the background structure, while a discriminative model can get away without segmentation by learning to detect only those differences across object classes that are reliably predictive of the label. The problem is that generative models need to explain everything, and without segmentation, there are too many things going on in the image to explain.

So for a generative model to be effective, it needs to be used in combination with a reliable image segmentation algorithm. Without it, applications of generative models will be limited mostly to datasets that have no structured background, such as MNIST and normalized-uniform NORB. Unfortunately, segmentation itself is a difficult unsolved problem. In this context, the ability of discriminative models to ignore background clutter is an advantage. But as segmentation algorithms improve, the importance of this advantage will decline.

3. **Intractable model evaluation:** Given two generative models, deciding which one is a better fit to the data is done by computing the probability each of them assigns to some held-out set of data. This computation is intractable for many interesting classes of generative models. For example, computing the probability of a data vector under an RBM (and similar energy-based models) requires computing the partition function, which involves summing an exponential number of terms. The sum is intractable for all but small, toy RBMs.

   Salakhutdinov and Murray [2008] have used Annealed Importance Sampling to approximate the partition function of an RBM, but it can give an estimate with a large error without indicating that the estimate is unreliable. In the absence of a direct estimate, even an upper bound on the partition function can be useful, as it gives a lower bound on the probability of a data vector. Then the lower bound for two models can be compared for a partial indication of which one may be better. But computing a strict upper bound on the partition function is also intractable.

   Not being able to directly measure a generative model's quality is a disadvantage, and perhaps *the* main obstacle to developing better generative models. In practice, it means that there is no objective way to decide, e.g., whether an RBM with 100 hidden units is better than one with 200. Discriminative models do not have this disadvantage since they can be easily evaluated by their classification error on a held-out set. When training RBMs and DBNs on images, one can try to assess progress during training by computing the model's squared pixel reconstruction error, or visually checking whether the filters learned by the model look sensible. But these are at best very indirect indicators of the quality of the model. For example, an RBM learned with CD can show a deceptively large improvement in pixel reconstruction error during training simply by getting the Markov chain to not mix well. There is clearly a need for better ways of evaluating RBMs, and generative models in general.

7.3 Looking Ahead

As these limitations show, learning good generative models is a long-term research undertaking rather than a solved problem. Nevertheless, the results in this thesis have hopefully convinced the reader that generative modeling has immediate applications to object recognition, and that further advances will lead to better results. Unlabeled images are becoming easier to collect in large quantities from the internet. GPUs, multi-core CPUs and cheaper RAM are making it more and more tractable to train much larger scale models than those currently being used. In this context, the tools and ideas presented here will only increase in their importance.

Going forward what we need are better generative models with better ways of training them and performing inference with them. Recent developments like DBNs are a step in that direction. With

more such advances, eventually it should become practical to learn a complex, large-scale generative model that approximates the true underlying generative process much more closely than current models do. Once that happens, the full potential of generative modeling and unsupervised learning for object recognition will be achieved, and computer vision systems will approach human-level performance at object recognition.

Bibliography

S. Belongie, J. Malik, and J. Puzicha. Shape matching and object recognition using shape contexts. *IEEE Trans. on PAMI*, 2(4):509–522, April 2002.

Y. Bengio and Y. LeCun. Scaling learning algorithms towards AI. 2007.

Y. Bengio, P. Lamblin, P. Popovici, and H. Larochelle. Greedy Layer-Wise Training of Deep Networks. In *NIPS*, 2007.

C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

C. M. Bishop and J. Lasserre. Generative or discriminative? getting the best of both worlds. *Bayesian Statistics*, pages 3–24, 2007.

G. Bouchard and B. Triggs. The trade-off between generative and discriminative classifiers. In *16th International Symposium on Computational Statistics*, pages 721–728, 2004.

T. F. Cootes, G. J. Edwards, and C. J. Taylor. Active appearance models. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 23(6):681–685, 2001.

D. DeCoste and B. Scholkopf. Training Invariant Support Vector Machines. *Machine Learning*, 46: 161–190, 2002.

M. Eden. Handwriting and pattern recognition. *IRE Transactions on Information Theory*, IT-8 (2): 160–166, 1962.

A. A. Efros and W. T. Freeman. Image quilting for texture synthesis and transfer. In *SIGGRAPH*, 2001.

R. Fergus, P. Perona, and A. Zisserman. Object class recognition by unsupervised scale-invariant learning. In *Proc. IEEE Conf. Computer Vision and Pattern Recognition*, 2003.

P. Foldiak. Learning invariance from transformation sequences. *Neural Computation*, 3:194–200, 1991.

B. Frey and N. Jojic. Transformation-invariant clustering and dimensionality reduction using em. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 1000–1014, 2000.

K. Fukushima. Neocognitron. *Scholarpedia*, 2007.

Z. Ghahramani and G. E. Hinton. The EM Algorithm for Mixtures of Factor Analyzers. *Technical Report CRG-TR-96-1, Dept. of Computer Science, University of Toronto*, 1996.

C. L. Giles and T. Maxwell. Learning, invariance, and generalization in high-order neural networks. *Applied Optics*, 26(23):4972–4978, 1987.

D. B. Grimes and R. P. N. Rao. Bilinear sparse coding for invariant vision. *Neural Computation*, 17: 47–73, 2005.

X. He, R. S. Zemel, and M. A. Carreira-Perpinan. Multiscale conditional random fields for image labeling. In *CVPR*, pages 695–702, 2004.

G. E. Hinton. Connectionist learning procedures. *Artificial Intelligence*, 40(1-3):185–234, 1989.

G. E. Hinton. A parallel computation that assigns canonical object-based frames of reference. In *Proc. of the 7th IJCAI*, 1981.

G. E. Hinton. Learning translation invariant recognition in a massively parallel network. In *PARLE: Parallel Architectures and Languages Europe*, pages 1–13, 1987.

G. E. Hinton. To Recognize Shapes, First Learn to Generate Images. *Technical Report UTML TR 2006-04, Dept. of Computer Science, University of Toronto*, 2006.

G. E. Hinton. Training products of experts by minimizing contrastive divergence. *Neural Computation*, 14(8):1711–1800, 2002.

G. E. Hinton and V. Nair. Inferring motor programs from images of handwritten digits. In *Neural information processing systems*, 2006.

G. E. Hinton and R. Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313:504–507, 2006.

G. E. Hinton, P. Dayan, B. J. Frey, and R. M. Neal. The wake-sleep algorithm for unsupervised neural networks. *Science*, 268:1158–1161, 1995.

G. E. Hinton, P. Dayan, and M. Revow. Modeling the manifolds of images of handwritten digits. *IEEE Trans. on Neural Networks*, 8(1):65–74, 1997.

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

J.M. Hollerbach. An oscillation theory of handwriting. *Biological Cybernetics*, 39:139–156, 1981.

A. Holub and P. Perona. A discriminative framework for modelling object classes. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2005.

N. Jojic, B. J. Frey, and A. Kannan. Epitomic analysis of appearance and shape. In *ICCV*, 2003.

M. Jones and P. Viola. Fast multi-view face detection. In *MERL TR2003-96*, 2003.

M. Jordan and D. Rumelhart. Forward models: Supervised learning with a distal teacher. *Cognitive Science*, 16:307–354, 1992.

M. Kelm, C. Pal, and A. McCallum. Combining Generative and Discriminative Methods for Pixel Classification with Multi-Conditional Learning. In *ICPR*, 2006.

H. Larochelle and Y. Bengio. Classification Using Discriminative Restricted Boltzmann Machines. In *ICML*, pages 536–543, 2008.

H. Larochelle, D. Erhan, A. Courville, J. Bergstra, and Y. Bengio. An empirical evaluation of deep architectures on problems with many factors of variation. In *ICML*, pages 473–480, 2007.

L. Lathauwer, B. Moor, and J. Vandewalle. A multilinear singular value decomposition. *SIAM Journal on Matrix Analysis and Applications*, 21(4):1253–1278, 2000.

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

Y. LeCun, F. J. Huang, and L. Bottou. Learning methods for generic object recognition with invariance to pose and lighting. In *CVPR*, Washington, D.C., 2004.

H. Lee, R. Grosse, R. Ranganath, and A. Ng. Convolutional Deep Belief Networks for Scalable Unsupervised Learning of Hierarchical Representations. In *ICML*, 2009.

Y. Lee, D. Terzopoulos, and K. Waters. Realistic modeling for facial animation. In *SIGGRAPH*, 1995.

D. Lowe. Distinctive image features from scale-invariant keypoints. In *International Journal of Computer Vision*, volume 20, pages 91–110, 2003.

R. Memisevic and G. E. Hinton. Unsupervised learning of image transformations. In *Proc. IEEE Conf. Computer Vision and Pattern Recognition*, 2007.

X. Miao and R. P. N. Rao. Learning the lie groups of visual invariance. *Neural Computation*, 19: 2665–2693, 2007.

K. Mikolajczyk and C. Schmid. A performance evaluation of local descriptors. *IEEE Trans. on PAMI*, 27(10):1615–1630, October 2005.

M. Minsky and S. Papert. *Perceptrons*. MIT Press, 2 edition, 1988.

T. Moriyama, T. Kanade, J. Xiao, and J. F. Cohn. Meticulously detailed eye region model and its application to analysis of facial images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 28(5):738–752, 2006.

V. Nair and G. E. Hinton. Implicit mixtures of restricted boltzmann machines. In *Neural information processing systems*, 2008.

V. Nair and G. E. Hinton. 3d object recognition with deep belief nets. In *Neural information processing systems*, 2009.

V. Nair, J. Susskind, and G. E. Hinton. Analysis-by-synthesis by learning to invert generative black boxes. In *International Conference on Artificial Neural Networks*, 2008.

M. Osadchy, M. L. Miller, and Y. LeCun. Synergistic face detection and pose estimation with energy-based models. 2004.

R. Raina, Y. Shen, A. Ng, and A. McCallum. Classification with hybrid generative/discriminative models. pages 545–552, 2003.

R. Raina, A. Madhavan, and A. Ng. Large-scale Deep Unsupervised Learning using Graphics Processors. In *ICML*, 2009.

Marc'Aurelio Ranzato, Chris Poultney, Sumit Chopra, and Yann LeCun. Efficient learning of sparse representations with an energy-based model. In *NIPS*. MIT Press, 2006.

Marc'Aurelio Ranzato, Fu-Jie Huang, Y-Lan Boureau, and Yann LeCun. Unsupervised learning of invariant feature hierarchies with applications to object recognition. In *Proc. Computer Vision and Pattern Recognition Conference (CVPR'07)*. IEEE Press, 2007.

R. P. N. Rao and Daniel Ruderman. Learning lie groups for invariant visual perception. 1999.

M. Riesenhuber and T. Poggio. Hierarchical models of object recognition in cortex. *Nature Neuroscience*, 2:1019–1025, 1999.

S. Roth and M. J. Black. Fields of experts: A framework for learning image priors. In *CVPR*, pages 860–867, 2005.

S. Roth and M. J. Black. Steerable Random Fields. In *ICCV*, 2007.

N. Le Roux and Y. Bengio. Representational Power of Restricted Boltzmann Machines and Deep Belief Networks. *Neural Computation*, 20:1631–1649, 2008.

H. Rowley, S. Baluja, and T. Kanade. Rotation invariant neural network-based face detection. 1997.

R. Salakhutdinov and G. Hinton. Learning a nonlinear embedding by preserving class neighbourhood structure. In *AISTATS*, 2007.

R. Salakhutdinov and G. E. Hinton. Deep Boltzmann machines. In *Proceedings of the International Conference on Artificial Intelligence and Statistics*, volume 5, pages 448–455, 2009.

R. Salakhutdinov and I. Murray. On the quantitative analysis of Deep Belief Networks. In *ICML*, Helsinki, 2008.

B. Scholkopf, P. Simard, A. Smola, and V. Vapnik. Prior knowledge in support vector kernels. In *NIPS 10*, 1998.

T. J. Sejnowski. Higher-order boltzmann machines. In *Neural Networks for Computing*, pages 398–403. American Institute of Physics, 1986.

E. Sifakis, I. Neverov, and R. Fedkiw. Automatic determination of facial muscle activations from sparse motion capture marker data. In *SIGGRAPH*, 2005.

P. Y. Simard, Y. LeCun, J. Denker, and B. Victorri. Transformation invariance in pattern recognition-tangent distance and tangent propagation. pages 239–27, 1996.

I. Sutskever and G. E. Hinton. Deep Narrow Sigmoid Belief Networks are Universal Approximators. *Neural Computation*, 20:2629–2636, 2008.

G. Taylor and G. E. Hinton. Factored Conditional Restricted Boltzmann Machines for Modeling Motion Style. In *ICML*, 2009.

J. B. Tenenbaum and W. T. Freeman. Separating style and content with bilinear models. *Neural Computation*, 12:1247–1283, 2002.

V. Vapnik. *Statistical Learning Theory*. John Wiley and Sons, 1998.

M. O. A. Vasilescu and D. Terzopoulos. Multilinear independent components analysis. 2005.

M. O. A. Vasilescu and D. Terzopoulos. Multilinear analysis of image ensembles: Tensorfaces. In *Proceedings of the European Conference on Computer Vision*, pages 447–460, Copenhagen, Denmark, 2002.

P. Vincent, H. Larochelle, Y. Bengio, and P. A. Manzagol. Extracting and Composing Robust Features with Denoising Autoencoders. In *ICML*, 2008.

P. Viola and M. Jones. Robust real-time object detection. In *Proc. of Workshop on Statistical and Computational Theories of Vision*, 2001.

M. Welling, M. Rosen-Zvi, and G. E. Hinton. Exponential family harmoniums with an application to information retrieval. In *NIPS 17*, 2005.

L. Wiskott and T. Sejnowski. Slow feature analysis: Unsupervised learning of invariances. *Neural Computation*, 14:715–770, 2002.

Appendix A

Datasets

A.1 MNIST

The dataset contains images of handwritten digits belonging to ten different classes (0 to 9). The digits are size-normalized and centred within a 28×28 image. Examples are shown in figure A.1. The training-test split of the dataset is pre-specified by its creators, so there is no ambiguity about how to measure performance. The training set has 60,000 images in total and the test set has 10,000 images. The pixels are real-valued and lie in the interval $[0, 1]$, with most values at the extremes of the interval. The dataset can be downloaded from http://yann.lecun.com/exdb/mnist/.

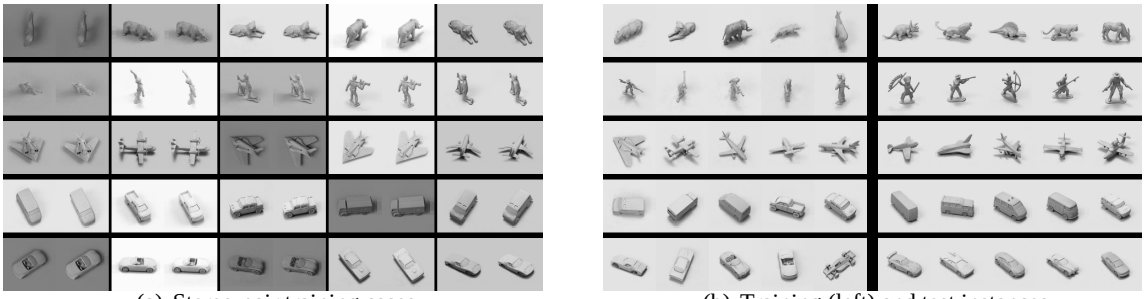
Figure A.1: Randomly selected examples from the MNIST training set.

A.2 NORB

NORB contains stereo-pair, grayscale images of toy objects under controlled lighting conditions and viewpoints. The five object classes are *animals*, *humans*, *planes*, *trucks*, and *cars*. The dataset comes in two versions, *normalized-uniform* and *jittered-cluttered*. In this thesis we only use the *normalized-uniform* version, which shows objects at a normalized scale and position with a uniform background. The dimensions of each image in the stereo-pair are 96×96. Examples are shown in figure A.2(a).

There are 10 different instances of each object class, imaged under 6 illuminations and 162 viewpoints (18 azimuth values $\times$ 9 elevation values). The instances are split into two disjoint sets (pre-specified in the database) of five each to define the training and test sets, both containing 24,300 cases (5 object types $\times$ 5 instances $\times$ 6 illuminations $\times$ 162 viewpoints). So at test time a trained model has

to recognize *unseen instances* of the same object classes. Figure A.2(b) shows the training (left) and test instances of each class in one row. Figure A.3 shows the various viewpoints from which each object is photographed. Figure A.3 shows the various lighting conditions. For more details, see LeCun et al. [2004]. The dataset is available at http://www.cs.nyu.edu/∼ylclab/data/norb-v1.0/.



(a) Stereo-pair training cases (b) Training (left) and test instances

Figure A.2: (a) Randomly selected examples from the NORB training set for the five different classes. Each row corresponds to one class. (b) All ten instances of each class with only one image from the stereo-pair, shown in the same lighting condition.

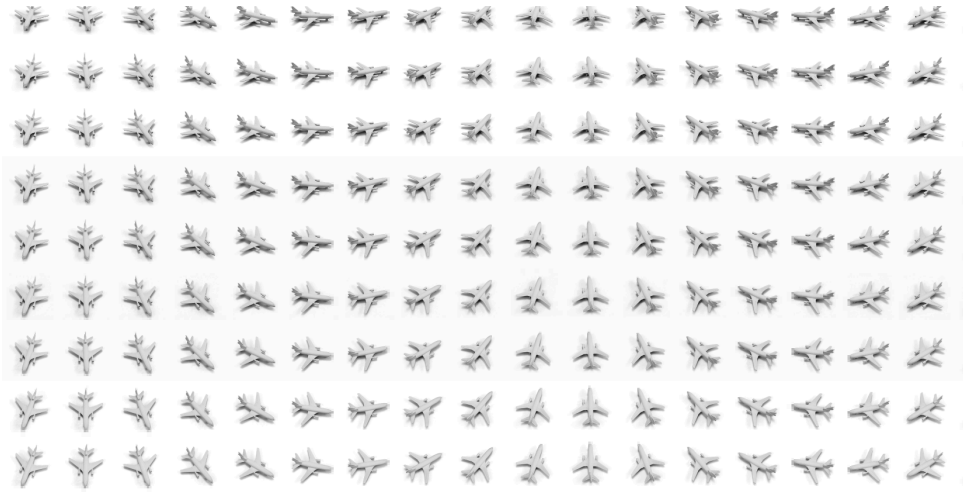


Figure A.3: The viewpoints from which each object in NORB is photographed. There are 9 elevation values (one per row) and 18 azimuth values (one per column) for a total of 162 viewpoints.

Figure A.4: The six lighting conditions in NORB.