Optimizing Neural Networks that Generate Images

by

Tijmen Tieleman

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

Abstract

Optimizing Neural Networks that Generate Images

Tijmen Tieleman
Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto
2014

Image recognition, also known as computer vision, is one of the most prominent applications of neural networks. The image recognition methods presented in this thesis are based on the reverse process: generating images. Generating images is easier than recognizing them, for the computer systems that we have today. This work leverages the ability to generate images, for the purpose of recognizing other images.

One part of this thesis introduces a thorough implementation of this “analysis by synthesis” idea in a sophisticated autoencoder. Half of the image generation system (namely the structure of the system) is hard-coded; the other half (the content inside that structure) is learned. At the same time as this image generation system is being learned, an accompanying image recognition system is learning to extract descriptions from images. Learning together, these two components develop an excellent understanding of the provided data.

The second part of the thesis is an algorithm for training undirected generative models, by making use of a powerful interaction between training and a Markov Chain whose task is to produce samples from the model. This algorithm is shown to work well on image data, but is equally applicable to undirected generative models of other types of data.

Acknowledgements

The primary acknowledgement is, of course, to Geoff Hinton. He taught me neural networks, and he inspired me by always taking very good care of his team and setting high standards for himself. When time comes that I lead a team, I will pass this on.

My additional thesis committee members, Radford Neal and Rich Zemel, have seen to it that this thesis is accessible to a far larger circle of people. For work as specialized as a PhD thesis, it is easy to forget that one must communicate in a way that not only the very closest colleagues will understand. Radford and Rich helped me achieve this.

My thanks also go to the three professors who volunteered to join the committee for the final examination: Yoshua Bengio, Raquel Urtasun, and Russ Salakhutdinov. Thank you for volunteering to read through an extensive piece of work and providing your perspectives.

Of course, there are others who were involved in the process of creating this thesis. After all, it was a long process, and such a thing is never done alone. My colleagues have made the group an environment where I felt at home. I want to mention especially Ilya Sutskever.

Lastly, a project of this magnitude is not just something professional, but also something personal: it carries the signature of a whole person. Therefore, I wish to thank all those who taught me and who supported me, and enabled me to get to the point where I could write this thesis: from family to internship supervisor, from my first computer programming teacher to my tango teacher, and from friends to all those giants on whose shoulders I have the privilege to stand.

Contents

List of Figures

0.1 Introduction, contributions, and structure of this thesis

Neural networks are among the most successful methods for computer vision (Krizhevsky et al., 2012), and are well-suited to take full advantage of the steadily increasing amounts of data and compute power at our disposal. There are different approaches to creating neural networks for image recognition. One method is creating a neural network with as much muscle as possible, and then training it with labeled data (Ciresan et al., 2010; Krizhevsky et al., 2012). Another approach starts by learning to generate images, before trying to recognize them. That second approach is the subject of this thesis.

Neural networks can generate images in various ways. The most widely used method is that of *(probabilistic) generative models*. These, again, come in a variety of flavours (see Section 1.5.2 in the literature review). One contribution of this thesis is an algorithm for training probabilistic generative models of the undirected flavour. The algorithm capitalizes on a powerful interaction between learning and exploration in such models, and is described in Section 3.

A somewhat less popular type of neural network that generates images is (the decoder component of) autoencoders. Unlike probabilistic generative models, vanilla autoencoders are deterministic, but the decoder component of an autoencoder can still generate different images because it takes input. Usually, that input, known as the *code* and produced from the input by the encoder component, is smaller than the image that is being generated, and is intended to be in a higher level, *more meaningful* representation or “language”. All of this sounds terribly imprecise, and that’s because we typically let the learning algorithm “choose” what the representation should be. This is the spirit of machine learning and it’s good, but one can also try to combine the power of machine learning with the power of engineering. Section 2 describes an autoencoder where part of the representation language is engineered and part is learned.

Section 1 is a literature review of deep neural networks.

All three chapters have their own introduction & conclusion, and can be read independently, although more knowledge is assumed in chapters 2 and 3 than in the literature review.

The unifying theme in this thesis is the aim of learning good internal representations of images, by learning to generate images from those internal representations. This is an increasingly widely pursued area of research, which now has its own conference (the International Conference on Learning Representations). Section 3 does it with a probabilistic generative model, and Section 2 does it with an autoencoder.

0.1.1 Contributions

I would like to highlight three research contributions in this thesis:

- The “Fast Persistent Contrastive Divergence” (FPCD) algorithm for training undirected generative models. This algorithm works by running an exploration alongside the training, and uses an interaction between learning and exploration to ensure that the exploration component keeps moving rapidly, without requiring the learning component to use a large learning rate. The efficient exploration is created by developing temporary aversion to wherever the explorer finds itself, thus forcing it to move on.

- A demonstration of a domain-specific data generation system guiding the learning of a data recognition system. This is done with an autoencoder, of which the decoder is built with domain-specific knowledge, and the encoder is an off-the-shelf neural network. By training these together, the encoder absorbs much of the domain-specific knowledge that has been put into the decoder. This is useful because domain-specific knowledge is often easier to describe as a data generation system than in a data recognition system.

- A demonstration that the power of componential systems engineering can be combined with the power of machine learning. The previously mentioned autoencoder consists of about a thousand[1] small components that together form an elaborately engineered system of the kind that is usually associated with physical or software engineering, but not with machine learning. Many of these thousand are learning components, which end up working together with the non-learning ones. This shows that machine learning and sophisticated componential engineering can be combined effectively.

[1] Depending on how one counts, the number could come out anywhere between a few hundred and well over a thousand.

Chapter 1

Literature review: deep neural networks

1.1 Introduction

In recent years, the machine learning community has shown a growing interest in deep neural networks, DNNs: systems composed of multiple groups (layers) of units, with the "lower" layers being connected to the input data and the "higher" layers being for internal data processing, connected only indirectly to the input data. Many researchers have found such systems to perform better, in practice, than "shallow" networks, which have no such division of tasks among the different units. Many of us have hypothesized about why such networks often work better, and have performed empirical or theoretical analyses of the nature of this advantage and how to maximize it.

Some clear and convincing conclusions have been drawn, but many questions remain unanswered. As in many fields of science, some groups believe firmly in the value of one type of system, while others believe those systems to have no future, and always deal with another type of system. Part of the problem, in the analysis of DNNs, is that there are many different phenomena at play, all interacting. Together with the fact that neural networks are largely black boxes, difficult to inspect, this makes it easy to misinterpret experimental outcomes. It also makes the discussion difficult: there are some popular buzzwords, but exactly what phenomenon they denote is not always well agreed.

This chapter is an attempt to separate some of those phenomena, at least on paper. Each chapter deals with another component of the strategy that can be used in dealing with DNNs. The value of each of these strategies is a debated issue: they all have advantages and disadvantages, and, therefore, great fans and fierce opponents. In this chapter, I do not attempt to take a stance in those debates. Instead, I hope to get a grip on what exactly the discussions are about, and to separate questions wherever possible. Which questions belong in the debate about unsupervised training, and which are instead about generative training? Which advantages and disadvantages come from having multilayer systems, and which come from greedy training? Such questions are the ones that I seek to answer, by listing, as separately as possible, the various issues that are known to be at play in DNNs.

1.1.1 Success stories

Deep neural networks (DNNs) have been used with some success on many different tasks, with many different types of data. Many applications are on pixel image data, including scans of handwritten digits (Hinton et al., 2006), photographs of toy objects (LeCun et al., 2004), and real life photographs (Fei-Fei et al., 2004; Torralba et al., 2007). However, many models have proven quite adaptable to work on data with a time dimension, such as speech (Mohamed et al., 2009; Mohamed et al., 2012), highly processed real-valued data (Taylor et al., 2007), and video (Sutskever & Hinton, 2007). They have also been applied to data with less easily visualizable structure, such as bag-of-words descriptions of documents (Salakhutdinov & Hinton, 2009b).

In general, DNNs seem to be most useful on high-dimensional data with strong and complicated statistical dependencies.

1.2 Using multiple layers

Before going into *how* best to use DNNs, we must consider why and when to use them.

1.2.1 Hierarchically structured data

Many types of data have a hierarchical structure. Images are, on the lowest level, pixels, but those pixels together form edges and colour gradients. Those edges and gradients are the building blocks for simple shapes, which in turn are the elements of more complex shapes. Those shapes are the parts of objects, and the objects form larger and larger groups, which eventually become a visual scene.

For speech data, the most basic measurements are sequences of air pressure readings. These are typically converted to spectrograms, which, for speech, form phonemes, words, phrases, sentences, and eventually coherent messages.

The reason why data is often structured this way is that the world, of which this data is a fairly direct observation, is similarly structured. Objects have parts, and the way they translate to sensory data reflects that.

This hierarchical structure is what inspires many researchers to build hierarchical networks to deal with these types of data. The first layer of units gets to interpret the pixels (or spectrograms) as lines and colour gradients (or frequency gradients). The next layer takes that processed version of the data, and processes it some more, in ways that would be difficult to do on the raw data. The next layer up goes to an even higher-level representation, etc. Or, more generally speaking, there are high level constructs to observe in data, that are best seen as combinations of slightly lower level constructs, which again say how even lower level patterns relate to each other. If the data indeed has such structure, then the hope is that DNNs will be able to take advantage of it.

Seen this way, DNNs are an attempt to analyse data in the form & structure of its underlying causes: the structured world. DNNs represent the assumption (or “prior”) that there is such a structured underlying reality.

1.2.2 Theoretical considerations on representational power

Any smooth function can be well approximated by very shallow networks of exponential width, or very narrow networks of exponential depth (Sutskever & Hinton, 2008). When we consider networks of less than exponential size, there are no “universal approximator” results, but small network depth is known to be a limiting factor on the family of functions that can be expressed efficiently by a network of simple units (e.g. threshold units) (Minsky & Papert, 1969; Allender, 1996; Bengio & LeCun, 2007; Bengio, 2009). Intuitively, this means that deep networks are more complicated and therefore potentially more difficult to train, but have more potential than shallow networks.

1.2.3 Neuroscience evidence

Another motivation, for some researchers, is the evidence from neuroscience suggesting that human and animal brains have exactly such deep, multilayer structure. The fact that this was developed by evolution suggests that it is a sensible approach.

1.2.4 When DNN’s do not help

On other tasks, DNNs are not the best technique to use. On data that has little structure, trying to extract deeply hidden concepts from the data is simply not the right way to go. On data that needs no processing in stages, DNNs can’t do much good, and the task should be left to more specialized methods, such as Support Vector Machines (SVM’s) (Cortes & Vapnik, 1995) for classification, or Gaussian Processes (GP’s) (Rasmussen, 2006) for regression. However, on more difficult tasks with high-dimensional input with strong dependencies, DNNs can hope to make supervised tasks easier by transforming the input to a high-level representation, which may be easier to handle for SVM’s, GP’s, or other specialized methods.

(Salakhutdinov & Murray, 2008) produced some evidence suggesting that the task of building a data distribution model of the MNIST handwritten digits (LeCun & Cortes, 1998) may be one where DNNs cannot help much: Deep Belief Networks (DBN’s) (Hinton et al., 2006) no longer outperformed Restricted Boltzmann Machines (RBM’s) (Smolensky, 1986), if plenty of compute time was available and a good training algorithm was used for the RBM. However, this finding has not been duplicated for the supervised discrimination task.

On that same dataset, with the task of distribution modeling, I noticed[1] that the representation in the third layer of units of a DBN was very similar to that of the first layer, i.e. the raw data. This, again, suggests that using multiple layers does not help much on that task.

1.3 Training layer by layer, greedily

Once we choose to use a multilayer network, and decide on unit types and network architecture, there is the question of how to train it.

One could choose an objective function and try to optimize that with a numerical optimizer (such as gradient descent). Alternatively, one could train the layers one at a time, bottom-up, greedily (i.e. without changing the layer parameters after training the layer). After such greedy training, full-model training can be used to fine-tune the model parameters, effectively turning the greedy training into a heuristic initialization phase, which is why it is often called “pretraining”.

Pretraining is done using an objective function that is thought (or proven) to be related to the objective function for the full model, but generally not the same.

[1] Unpublished work.

Greedy layer-wise training can be done for feature extraction purposes (Bengio et al., 2007), be it unsupervised (training the layer as an autoencoder) or supervised (training the layer as a hidden layer in a classification model), or for distribution modeling purposes, be it in a directed model (Hinton et al., 2006) or an undirected model (Salakhutdinov & Hinton, 2009a).

Many papers have used this strategy: (Hinton et al., 2006; Salakhutdinov & Hinton, 2009a; Lee et al., 2009; Nair & Hinton, 2010a; Salakhutdinov & Hinton, 2009b; Hinton & Salakhutdinov, 2006; Deselaers et al., 2009; Mohamed et al., 2012), to name just a few.

1.3.1 In favour of greedy, layer-wise training

The main inspiration for greedy layer-wise training of DNNs is the difficulty of training the entire model from scratch. The popular optimization technique of gradient descent runs into two problems. First, if the model parameters are initialized to small values, then many training objective functions have small gradients w.r.t. the parameters of the lowest layers, for reasons explained in (Bengio et al., 1994). Second, many believe that the optimization lacks the ability to change the parameters enough to find a truly good parameter setting if the optimization is not started close to a good solution. Although these two observations have been called into question recently by (Martens, 2010), they form the cornerstone of the case for greedy layer-wise training.

A way to mitigate these problems is to use heuristics to find parameter settings that are, hopefully, somewhat sensible and therefore close to a good parameter setting. Greedy layer-wise training is an attempt to do that. Not only can it yield somewhat sensible parameter settings; it will also reduce the problem of the small gradients by eliminating the need for initialization with small parameter values.

1.3.2 Restricted Boltzmann Machines

RBM basics

One of the commonly used methods for pretraining a model layer by layer is the Restricted Boltzmann Machine (RBM). An RBM is an energy-based model for unsupervised learning (Hinton, 2002; Smolensky, 1986). It consists of two layers of units: one *visible*, to represent the data, and one *hidden*. RBM's are typically used for unsupervised learning, i.e. for modelling a probability distribution over visible vectors[2]. If the visible units are divided into two groups, they can also be used to model the joint distribution of, for example, images composed of binary pixels and their class labels.

The simplest RBM is one where all units are binary, and we focus on that case. However, this special case can easily be generalized to other *harmoniums* (Smolensky, 1986; Welling et al., 2005) in which the units have Gaussian, Poisson, multinomial, or other distributions in the exponential family. Most of the analysis and algorithms mentioned here are equally applicable to those other cases.

A binary RBM has three sets of parameters: a bias to each visible unit, a bias to each hidden unit, and a connection weight between each pair of a visible unit and a hidden unit.

Standard notation is to use i for indices of visible units and j for indices of hidden units. w_{ij} denotes the strength of the connection between the i^{th} visible unit and the j^{th} hidden unit. b_i denotes the bias to the i^{th} visible unit, and a_j is the bias to the j^{th} hidden unit. v_i denotes the state of the i^{th} visible unit, and similarly h_j denotes the state of the j^{th} hidden unit. θ denotes the collection of learnable

[2] However, (Hinton et al., 2006; Larochelle & Bengio, 2008; Schmah et al., 2010) illustrate how RBMs can be used for classification.

parameters, i.e. the combination of the biases to the visible layer, the biases to the hidden layer, and the weights between the two layers.

An *energy* function is defined on states: $E_\theta(v, h) = -\sum_{i,j} v_i h_j w_{ij} - \sum_i v_i b_i - \sum_j h_j a_j$. This is replaced by a different function for RBMs that have units that aren't binary.

Through these energies, probabilities are defined as $P_\theta(\vec{v}, \vec{h}) = \exp(-E_\theta(\vec{v}, \vec{h}))/Z_\theta$ where Z_θ is the normalizing constant $Z_\theta = \sum_{\vec{v}', \vec{h}'} \exp(-E_\theta(\vec{v}', \vec{h}'))$.

The probability of a data point (represented by the state $\vec{v}$ of the visible layer) is defined as the marginal: $P_\theta(\vec{v}) = \sum_{\vec{h}} P_\theta(\vec{v}, \vec{h}) = \sum_{\vec{h}} \exp(-E_\theta(\vec{v}, \vec{h}))/Z_\theta$

Given a training data distribution D, the training data log likelihood objective function is $\phi(\theta) = \mathop{\mathrm{E}}_{\vec{v} \sim D}[\log P_\theta(\vec{v})]$. This can be conveniently rewritten as $\phi(\theta) = \phi^+(\theta) - \phi^-(\theta)$, where $\phi^+(\theta) = \mathop{\mathrm{E}}_{\vec{v} \sim D}[\log \sum_{\vec{h}} \exp(-E_\theta(\vec{v}, \vec{h}))]$, and $\phi^-(\theta) = \log Z_\theta = \log \sum_{\vec{v}, \vec{h}} \exp(-E_\theta(\vec{v}, \vec{h}))$. Notice that $\phi^-(\theta)$ does not depend on the training data.

Gradients

The *positive* gradient $\nabla\phi^+$ is simple: $\nabla\phi^+ = \mathop{\mathrm{E}}_{\vec{v} \sim D, \vec{h} \sim P_\theta(\cdot|\vec{v})}[\nabla(-E_\theta(\vec{v}, \vec{h}))]$. For w_{ij} in a binary RBM, this is: $\frac{\partial\phi^+}{\partial w_{ij}} = \mathop{\mathrm{E}}_{\vec{v} \sim D}[v_i \cdot P_\theta(h_j = 1|\vec{v})]$. See Section 4.2 for details. It corresponds to reducing the energy of the training cases, and it can easily be computed exactly.

The *negative* gradient for energy-based models is $\nabla\phi^- = \mathop{\mathrm{E}}_{\vec{v}, \vec{h} \sim P_\theta}[\nabla(-E_\theta(\vec{v}, \vec{h}))]$. For w_{ij} in a binary RBM, this is $\frac{\partial\phi^-}{\partial w_{ij}} = P_\theta(v_i = 1, h_j = 1)$. See Section 4.3 for details. It corresponds to increasing the energy of those configurations to which the model assigns high probability; sometimes this is called "unlearning" the model distribution. It is expensive to compute exactly; in general, not even an unbiased estimate can be obtained in less than exponential time. Such an unbiased estimate would require samples from the model, and getting those samples is intractable. Because it is the intractable part, this "unlearning" is the most interesting part of the gradient.

To get an efficient approximation of $\nabla\phi^-$, one uses some algorithm to *approximately* sample from the model.

Contrastive Divergence

The Contrastive Divergence algorithm with 1 step (CD-1) (Hinton, 2002; Bengio & Delalleau, 2007; Sutskever & Tieleman, 2010) is a widely used method for efficiently approximating $\nabla\phi^-$. It attempts to at least estimate the *direction* of the gradient somewhat accurately, even if the estimate of the size is likely to be far from accurate. To get an approximate sample from the model, the algorithm starts a Markov Chain at one of the training data points used to estimate $\nabla\phi^+$, performs some number of full Gibbs updates (e.g. 10), and treats the resulting configuration as a sample from the model.

The algorithm has a parameter: the number of Gibbs updates that are performed. For example, CD-10 is where we perform 10 Gibbs updates in going from training data to the approximate sample.

Mean field CD (Welling & Hinton, 2002), abbreviated MF CD, is a variation on CD where we never make any stochastic choices, and just substitute (conditional) probabilities for the binary states that CD usually works with. This has the advantage of being a deterministic gradient estimate, which means that larger learning rates can be used safely.

1.3.3 Disadvantages

The main disadvantage of greedy training, assuming that it is used to initialize an optimization procedure that optimizes all parameters jointly, is that it introduces more meta-parameters. Instead of one optimization task, there are two - each with their own meta-parameters. If those meta-parameters of the pretraining phase are not chosen well, the pretraining might do more harm than good.

Another problem is that typically, the pretraining objective function is not the same as the final objective function. Sometimes, the pretraining optimizes a bound on the final objective function, and is therefore theoretically guaranteed to improve (a bound on) the final objective function, though in practice the conditions required for this guarantee are seldom met (Hinton et al., 2006; Salakhutdinov & Hinton, 2009a). For supervised tasks there is no such guarantee even in theory. This makes it all the more difficult to choose the two sets of meta-parameters well, and means that this method must be seen as entirely heuristic, although it has consistently been found to work well.

1.3.4 Greedy, but with more consideration for the higher layers to come

Usually, greedy layer-wise training means that each layer is trained in a way that would be most appropriate if there were no other layers to be trained on top of it. For example, in (Hinton et al., 2006), each layer is trained as an undirected top layer for a DBN, even though in the end the intermediate layers are not used that way. An alternative is to acknowledge that most of those greedily trained layers are going to be intermediate layers, and to try to prepare them for that task.

For lower layers in a DNN, it is important that much of the information that is in the data is preserved in the transformation that is applied by that layer. Encouraging good autoencoding is, therefore, important in the lower layers (Tieleman, 2007; Le Roux & Bengio, 2008), while the higher layers should focus more on modeling or transforming the data.

1.4 Unsupervised (pre)training

If the task at hand is an unsupervised task, then of course training is unsupervised. However, even when the task is a supervised one, some unsupervised training can be a good idea. That is what this section is about.

Training then happens in two stages: first, the unsupervised initialization or “pre-training”, and second, the supervised training - much like greedy layer-wise training can be initialization for training that considers all layers. Because this strategy is often combined with the greedy pretraining strategy and the generative training strategy, it is easy to confuse them. However, greedy pretraining can well be used without unsupervised training (Bengio et al., 2007), and unsupervised pretraining often happens with autoencoders as opposed to generative methods, so it can be analyzed somewhat independently.

Unsupervised pretraining comes in two main flavours: *generative* pretraining (see Section 1.5), and *feature extraction* pretraining (e.g. using autoencoders). Some algorithms, such as CD (Hinton, 2002) and some explicit gradient mixing strategies, have a bit of both flavours.

Like greedy pretraining, unsupervised pretraining must be considered a heuristic, because the pretraining objective function is at best weakly related to the final (supervised) objective function. In the case of generative pretraining, there is no clear theoretical connection between the two, and it is theoretically possible that the pretraining produces a model completely useless for any supervised task (see Section 1.5.5). On the other hand, if the unsupervised pretraining uses the autoencoder objective function, there is some connection: both autoencoders and supervised classification or regression use the intermediate layers as (typically deterministic) feature extraction layers whose task it is to re-encode the input data in some form.

Greedy layer-wise pretraining is almost always unsupervised, so a number of papers that claim to have benefitted from unsupervised pretraining can be found in Section 1.3.

1.4.1 In favour of unsupervised pretraining

The main motivation for unsupervised training is that there is much potentially useful information in the inputs X that might be ignored by simple methods that try to learn $P(Y|X)$ immediately. For example, in the case of image classification, the images themselves contain much information (images can rarely be described by a few bits only), while the information content of the label is just logarithmic in the number of classes. By taking a closer look at the X first, one can often find some important structure in it, that can make $P(Y|X)$ easier to model. From looking at the X data alone, one can often produce meaningful alternative encodings of X. This is not an attempt to use more data, but rather an attempt to use the available data differently. Unsupervised pretraining is a way to emphasize structure in the X data, in order to make better use of it.

Because there is typically much more information in unsupervised learning signals, one can use it to train networks with many more parameters, before overfitting becomes an issue. Of course this means training with an objective function different from the final one, but the hope is that a network produced in such a way will be close, in parameter space, to a network that performs well on the final task.

Unsupervised pretraining can make use of unlabeled data, and is therefore an example of semi-supervised learning (Zhu, 2005). An interesting example of using unlabeled data from a slightly different source is given in (Lee et al., 2009). For many datasets, however, simply making good use of the information in the input data distribution has proven to be a big win over simple supervised-only training.

Often, unsupervised pretraining is combined with other strategies, such as greedy pretraining and generative training. This can make it difficult to isolate the merits of those individual strategies. However, (Bengio et al., 2007) includes a comparison of greedy unsupervised pretraining vs. greedy supervised pretraining, and reports that the unsupervised version outperforms the supervised version. Their hypothesis is that supervised training fails to produce information-preserving feature detectors (see Section Section 1.3.4), but of course the method may also suffer from overfitting.

1.4.2 Disadvantages

The same disadvantages as with greedy pretraining apply: this is heuristic pretraining, with inherently more meta-parameters, and the risk that the two training phases won't connect well because their objective functions are quite different.

In the case of unsupervised pretraining for a final supervised task, the second problem shows up as the fear that the feature extraction units created by the unsupervised pretraining will not contain the information needed for the supervised task. In the case of a generatively pretrained multilayer network, with the classification or regression using only the inferred states of the top level units, some information that was in the input data may simply be absent in that top layer representation (see the random bit experiment in Section 1.5.3).

This problem is less likely to show up if the unsupervised pretraining was not *generative* but done for *feature extraction*, using autoencoders (again, see Section 1.5.3). However, even with autoencoders, it is quite possible that the extracted features focus mostly on parts of the input that are entirely irrelevant to the supervised task. For example, in image labelling tasks, the task is usually to label the foreground, but unsupervised pretraining may well focus on the background (depending on what distinguishes foreground from background, and depending on the unsupervised pretraining objective function that is used).

One way to deal with that particular problem is to segment out the region of interest in the input data. However, segmentation is another difficult task. Another method is to modify the input to make the background less interesting for the unsupervised objective function, for example by eliminating big differences due to lighting (Nair, 2010).

However, this problem is not going to go away entirely, as long as pretraining is done with a different objective function than the final training.

1.5 Distribution modeling versus transformation sequence learning

If one chooses to pretrain using an unsupervised objective function, there are two main classes of those: *generative* (also known as *distribution modeling*) objective functions, and *transformation sequence learning* objective functions, which learn a sequence of representation transformations: each layer of units in a DNN embodies a different representation of data, and the layers of weights between them embody the transformations.

Although these are two separate classes of objective functions, it should be noted that there are strong connections between the two, and that a model trained with one can do well on the other. A model trained with a generative objective function does, typically, also define a representation transformation. Likewise, some models that are not explicitly trained as a probability distribution can still be thought of as defining a probability distribution (Bengio et al., 2013b; Bengio & Thibodeau-Laufer, 2013; Bengio, 2013).

If one chooses generative pretraining for a supervised task, the most common[3] approach is to convert the model of $P(X)$ into a module that transforms X into another representation, and then to use that other representation as the input to a supervised algorithm. This is done by using a latent variable model for the generative learning, and then using (possibly approximate) inference to go from raw input to a factorial distribution over the latent variable states[4]. That factorial distribution will specify a probability of turning on, for each latent variable, and those probabilities are then used as the transformed data.

1.5.1 Transformation sequence learning

Representation transformation learning is typically done with simple feedforward representation inference, and a simple noise model that defines how to measure the difference between the intended output and the actual last representation of the data[5].

[3] A common alternative is to train a generative model of inputs and labels, and then use the posterior label probability, given the input, for classification (Hinton et al., 2006; Larochelle & Bengio, 2008).

[4] With generative DNNs, the most common approach is to use only the latent variables in the top layer

[5] For general real-valued inputs, a common noise model is a spherical Gaussian with fixed variance. For binary inputs, it makes sense to use the KL divergence between the intended output and the Bernoulli distribution that's defined by taking

For pretraining with a transformation sequence objective function, we need a target output. This can be the same as the input (an autoencoder), the same as the target output that will be used after the pretraining phase (Bengio et al., 2007), or simply the target outputs from a different but related dataset (Sermanet et al., 2014). After training, we discard all but the first representation transformation (or the first few), we use that first transformation to re-represent our data, and we train a new model on the data in this new representation.

Most such systems, in particular autoencoders, have been trained entirely deterministically. Recently, "denoising autoencoders" (Vincent et al., 2008) and "dropout" (see Section 1.8.4) have changed that. These attempt to produce an encoder that is more robust to noise, by randomly setting some units' states to 0 for the encoder, while still requiring that the original (now omitted) input be reconstructed.

1.5.2 Distribution modeling: directed versus undirected

Generative models can be further classified as *undirected* models and *directed* models.

Undirected models

Single layer models A Restricted Boltzmann Machine (RBM, see Section 1.3.2), is a single layer undirected generative model.

Multilayer models The Deep Boltzmann Machine (DBM) (Salakhutdinov & Hinton, 2009a), also used in (Lee et al., 2009) with convolutional connectivity, is an undirected *multilayer*[6] generative model. Technically speaking, it is simply a Boltzmann Machine (Hinton & Sejnowski, 1986), but its units are organized in layers, which enables a greedy pretraining scheme (Salakhutdinov & Hinton, 2009a) similar to the one for DBN's in (Hinton et al., 2006). The DBN pretraining scheme is not designed for training DBMs[7], but remarkably still seems to have worked acceptably in (Lee et al., 2009).

Training The training data log likelihood gradient for undirected models is the difference of the gradient of unnormalized data log likelihood (the "positive" component), and the gradient of the normalization term (the "negative" component). For all but very restricted connectivity architectures, the negative component is intractable and must be approximated, typically using approximate sampling from the model. For undirected single hidden layer models that do not include hidden-to-hidden connections, like RBMs, the positive component can be calculated exactly with little computation; for less restricted architectures, the positive component can be approximated using simple "mean field" variational inference (Wainwright & Jordan, 2003; Salakhutdinov & Hinton, 2009a).

Mean field variational approximation with factorial variational distributions is not appropriate for the negative component. Such a factorial distribution is unimodal[8], and this is often acceptable for a distribution that's conditional on the state of the visible units (that's what happens when we approximate the positive component). After all, that conditioning is typically a major restriction, leaving room for essentially only one "interpretation" (state of the hidden units). However, the distribution of a good

the actual output as a probability.

[6] Multilayer neural networks are usually called "deep" neural networks.

[7] In a DBM, a unit receives input from both of its adjacent layers (if there are two adjacent layers). However, in the lower layers of a DBN, a unit only receives input from the layer above it (the layer of its "parent" nodes). If we use a pretraining method that's designed for DBNs, and then use the produced model as a DBM, those units will be receiving input from twice the number of units for which it was designed.

[8] For binary variables.

generative model, not conditioned on anything (this is what the negative component is about), tends to be multimodal: it is trained to give high probability to a variety of training cases. Therefore, a mean field variational approximation is a poor choice for modeling the unconditional distribution.

The approximate samples required for the negative component can be obtained in a number of ways (Tieleman, 2008). The Contrastive Divergence (CD) algorithm (Hinton, 2002) does it by running a Markov Chain for a short time, starting at training data. In the case of binary units, "Pseudo Likelihood" (Besag, 1986) can be seen as an extreme case of CD, where the Markov Chain transition operator performs as few changes as possible. The "Persistent Contrastive Divergence" (PCD) algorithm (Tieleman, 2008; Salakhutdinov & Hinton, 2009a) uses the same approach as CD except that the Markov Chain state is maintained between gradient estimates. This has a useful interaction with the learning algorithm, which causes rapid mixing. This interaction is reported on and exploited further in Section 3.

Comparing undirected models to directed models Undirected layered models with the typical restriction that there be no intra-layer connections have one main advantage over directed models: the units of a layer are conditionally independent given the states of the units of adjacent layers, which may result in more efficient Gibbs sampling. Directed models lack this feature because of the "explaining away" (Pearl, 1988) effect which means that nodes in the same layer are not conditionally independent given the state of their children.

One disadvantage of undirected models is that drawing samples from exactly the model distribution (not conditioning on anything) typically requires exponential time. A related issue is that gradient estimation for training undirected models includes a "negative phase", which almost always has to be approximated.

Directed models

Variations Directed *multilayer* models include Sigmoid Belief Networks (SBN's) (Neal, 1992) and Deep Belief Networks (DBN's) (Hinton et al., 2006). DBN's are technically a hybrid of directed and undirected models, but here they are classified as directed multilayer models, because most of their layers are directed.

Model interpretation In directed models, the presence of lower layers does not change the distribution described by the higher layers. Sampling from a directed model (i.e. when we're not conditioning on anything) can be described as drawing a sample from the distribution defined at the top, and then stochastically propagating that one sample through the lower layers. In SBN's, which are purely directed, this sampling at the top (when we're not conditioning on anything) is trivial because the units of the top layer are independent. In a DBN, the top two layers of units are an RBM, from which exact sampling is intractable.

Training The training procedure for directed models involves sampling from posterior given the state of the visible units. Because this is intractable, approximate inference is used in practice.

The approximate inference may be implemented as a Markov Chain (Neal, 1992), or using variational inference parameters (Hinton et al., 1995; Hinton et al., 2006).

1.5.3 The difference, illustrated by the random bit experiment

Representation transformation learning can be quite different from distribution learning, as is well illustrated by the following thought experiment. Let the training data be binary vectors of fixed length, and let one of those bits (say the first one) be statistically independent from the other bits, in the training set. What will unsupervised learning do with that bit?

An autoencoder will try to reconstruct that bit from the hidden representation, and will therefore try to set one unit in the hidden layer apart for modeling this independent bit. That way, the bit will indeed be reconstructed accurately.

A generative model will be able to describe that bit with very little modeling resources: only the base rate needs to be noted, and most models can do this using a bias. A latent variable model has no reason to use any latent variables for dealing with this bit.

This difference illustrates well how different the two objective functions can be.

As a result of those differences, the latent representation found by generative models will be uninformative about the value of that bit, while an autoencoder will try to produce an internal representation that specifies exactly what state the bit was in.

If the independent bit was random and irrelevant noise, then the internal representation produced by generative models is obviously preferable. If, however, the bit specifies information relevant the supervised task, then the autoencoder is doing better. The same applies when there is a bit that only *seems* to be independent from the other units, because the dependencies are not of a type that the model naturally detects.

1.5.4 In favour of distribution modeling

Theoretical advantages

For some generative models (DBN's and DBM's), there are greedy layer-wise training procedures for which the training objective function is a lower bound on the full-model objective function. Even though in practice these training procedures are not followed exactly, their existence is theoretically appealing.

Empirical advantages

Many papers claim to have benefitted from generative training (Bengio et al., 2007; Lee et al., 2009; Hinton et al., 2006; Salakhutdinov & Hinton, 2009a; Nair & Hinton, 2010a). One problem, however, with empirically evaluating generative training is that it is often combined with greedy unsupervised pretraining and other methods, and the authors conclude “deep networks are great”, without being quite sure whether to attribute their success to having multiple layers, to having unsupervised training, to using greedy pretraining, to using generative pretraining, to thorough meta-parameter tuning, or to luck. However, Bengio and Larochelle have done many reasonably controlled comparisons between training layers as RBM's using the Contrastive Divergence (CD) (Hinton, 2002) algorithm, and training them as autoencoders, and usually the RBM method performs best. (Bengio, 2009) hypothesizes that this is because CD-1 training is closer to true generative training (in RBM's). There seems to be a general sentiment (see also (Hinton, 2002)) that training with the exact likelihood gradient would be best, if only there would be a way to get it quickly and with little noise.

More sophisticated inference

After a generative model is trained, it is often used as a method for re-encoding the data, in a (hopefully) more semantic representation. For a generative model, this encoding step is often simply computing the conditional probability of the hidden units to turn on given that the visible units represent the original input. Those probabilities, sometimes computed exactly and sometimes just approximately, then become the new representation of the original input. Regardless whether this inference is exact (e.g. when the model is an RBM) or approximate (e.g. when the model is a DBN), the computation is often done by a simple deterministic feedforward neural network. In this case, the best encoding that such a generative model could theoretically produce is the same as the best encoding that an autoencoder could theoretically produce[9].

However, when the encoding is more sophisticated than simply a deterministic feed-forward (greedy) pass, there is a clear theoretical argument explaining why generative models may well produce better representations. If, in a generative model, proper inference is done, i.e. also taking into account the units above a layer when choosing a state for the units of that layer, then this top-down influence might help disambiguate the input. Such inference, however, is rarely used (in (Salakhutdinov & Hinton, 2009a) it is used for training).

1.5.5 In favour of transformation sequence learning

An extreme version of the random bit problem

An extreme case of the random bit example is a generative model that does all modeling in the first 3 layers and does not use layers 4 and 5 at all. The inferred state of those top layers will carry zero information about the input, and will therefore be useless for any supervised task. I don't know of any situations where something like this was observed in practice, nor would it be a problem if one simply used the states of *all* hidden units as the encoding. However, the thought experiment does highlight how unsupervised training, especially generative training, uses a different objective function than classification and can, therefore, be less-than-ideal when the application of the network is classification.

Generative models are hard to evaluate

Another drawback of generative models is that many, such as Sigmoid Belief Networks (Neal, 1992), Deep Boltzmann Machines (Salakhutdinov & Hinton, 2009a), RBM's and DBN's, are difficult to evaluate. Autoencoders and purely discriminately trained models come with a tractable objective function that can be measured at any time, on the training data or on held-out validation data, to assess training progress and compare meta-parameter settings. A generatively trained model that is going to be used, in the end, for classification, can of course be evaluated once the entire model is finished and the classification performance can be measured, but a first layer cannot be evaluated this way before all other layers are fully trained. This makes algorithm experimentation and meta-parameter choosing much more difficult.

One popular way to monitor the progress of RBM training is to track training data reconstruction error, but that is hard to interpret well (Tieleman, 2007). (Salakhutdinov & Murray, 2008) have come a little closer to evaluating $P(x)$ under many generative models, but their method takes much time and

[9] However, generative models have their advantages during training, which means that they might end up closer to that best possible encoding.

is not entirely reliable. (Schulz et al., 2010) thoroughly analyses these two diagnostics, and concludes that neither is fully satisfactory.

In RBM's, one can track overfitting directly by comparing the free energy of training data versus validation data. This does not show unambiguously whether or not the validation data log probability is increasing or decreasing, but when the model is overfitting, this will produce a warning (the problem is that it might also produce that warning before the validation data log probability begins to decrease).

1.6 Preprocessing data

Some DNN users preprocess the data before giving it to the DNN. Sometimes that is necessary to get the data in a format that can be handled by their preferred type of DNN; at other times it is not strictly necessary but nonetheless helps by producing data that is more suited to the strengths and weaknesses of the DNN.

A third motivation for preprocessing is the feeling that without it, the first layer of the DNN will just learn to do roughly that same preprocessing, and doing it manually saves time and reduces the required number of layers. For example, on image data, many types of networks learn units in the first layer that are very similar to Gabor filters. On the other hand, selecting the right instances from the space of Gabor filters (location; scale; orientation) is a difficult task, and may still be solved best by a learning algorithm.

(Mohamed et al., 2009) is an example of extensive preprocessing, which significantly helped the model.

(Nair & Hinton, 2010a) used preprocessing to reduce the data dimensionality without significantly changing the nature of the data. The aim was to reduce compute time.

(Tieleman & Hinton, 2009) used preprocessing to turn grey scale image data into roughly binary data, which was easier for the network to handle.

Subtracting the mean and dividing by the standard deviation is a simple method for bringing the data to a standard scale.

Another commonly used preprocessing method is whitening the data. Without strong pairwise correlations in the input, the model may be able to learn something more interesting about the data than just the correlations that it would probably learn on unwhitened data. Principal Component Analysis can be used to decorrelate data and reduce its dimensionality without losing much information.

A completely different idea is to use the DNN itself as a preprocessing step. (Salakhutdinov & Hinton, 2009b) uses an DNN to process bag-of-words descriptions of documents into short binary codes, which enables a simple look-up program to perform the ultimate task of, given a new document, finding other documents similar to it.

1.7 Numerical Optimization

Numerical optimization is the main loop in most NN computer programs, but it most papers give it little attention: they use stochastic gradient descent, possibly with momentum, and seem to assume that there's little else that can be done about optimization. However, more advanced optimization strategies do exist, and can make a big difference.

(Hinton & Salakhutdinov, 2006; Martens, 2010) have shown that putting effort into the optimization can be well worth the effort. (Martens, 2010), in particular, has shown that a better optimizer can enable a whole new class of models (very deep or recurrent ones) to learn well in practice, much like the concept of greedy pretraining (Hinton et al., 2006) did[10]. For those who are reluctant to switch to another optimizer than gradient descent, there are alternatives that might be worthwhile, such as using short runs to carefully select a learning rate, or even a learning rate schedule. The still-common attitude that gradient descent is the only usable optimizer is simply mistaken, as (Martens, 2010) showed very convincingly.

1.7.1 Solutions found (or not found) by gradient descent

To the best of my knowledge, researchers who train networks using gradient descent (the de facto standard) never reach a local optimum of their objective function. Note that this casts the many arguments about local optima versus global optima in a somewhat suspicious light, and suggests that we refrain from claiming "local optimum!" when what we really observe is the more general phenomenon of "optimizer failure". One may plot the value of the objective function and notice that it roughly levels out at some point, but this is in no way a guarantee that no further improvement would be achieved with more optimization: it only means that if further improvement is to be achieved, it is going to require significantly more effort. Of course this optimization failure can serve a useful regularization, like early stopping, but usually one wants to have more control over the regularization than this allows.

1.7.2 Monte Carlo gradient estimates

One of the reasons why gradient descent is popular is that often the best available optimization information is an unbiased but stochastic gradient estimate, and gradient descent is one of the few algorithms that don't need more accurate information.

Sometimes, like with the Contrastive Divergence (Hinton, 2002) gradient estimator, this stochasticity is an inevitable part of the gradient estimator. At other times, however, the only source of stochasticity is the use of mini-batches. (Schraudolph & Graepel, 2003; Hinton & Salakhutdinov, 2006) suggest a way to combine the use of mini-batches with the use of optimizers that require exact gradient information. Their outer loop is over mini-batches, and a short inner loop takes such a mini-batch and optimizes the model parameters for a few iterations using only the data in that mini-batch. The underlying optimizer can be anything (such as the method of conjugate gradients).

1.7.3 Getting better hardware instead of software

An approach that has proven more popular than trying to create better optimization software, is to find better hardware. (Raina et al., 2009; Mnih, 2009; Tieleman, 2010) found that many machine learning algorithms can be accelerated greatly using machinery originally designed as graphics processors but now somewhat specialized for scientific computation.

[10] Later, (Sutskever et al., 2013) showed that intelligent model initialization can be sufficient in getting these same models to work.

1.8 Desirable unit properties

1.8.1 Sparsity

Recent empirical and theoretical findings suggest that it is good to have units that are rarely active. This is usually called sparsity, not to be confused with sparsity of parameter matrices.

Advantages of sparsity

(Ranzato et al., 2006; Lee et al., 2009; Nair & Hinton, 2010a), among others, claim to have benefitted from sparsity, be it as a natural phenomenon or induced by adding sparsity-inducing components to the training objective function.

Interpretability The most commonly heard argument in favour of sparsity is increased interpretability: the extent to which a unit's state is informative, e.g. for classifying the input. If a unit is active in the latent representation of only few input patterns, then it seems more plausible that the unit responds to something fairly specific and hopefully identifiable in the input. In a classifier, if such a unit is active, then that is a clear signal to the classifier as to what is going on in the input. Compare this to a unit that is often active. Such a unit corresponds to something that is present in many (say half) of the inputs, and is therefore a less informative signal when it is on[11].

Revives 'dead' units (Nair, 2010) mentions that a particular sparsity target, e.g. 5%, can help the learning process by quickly reviving units that would otherwise be inactive for all or almost all training cases and under the model distribution. In RBM's, such units get almost zero gradient, and thus take a long time to develop into something more useful, when gradient descent is used.

How to make sparsely active units

Often, sparse activity occurs without explicit effort to enforce it, especially in higher layers (see (Lee et al., 2009) and the online demonstration of (Hinton et al., 2006)). (Lee et al., 2009) reported strong correlations between those sparse higher layer units and the image class label, which suggests that those higher layer units respond to complex, semantically meaningful, rarely present features, such as large parts of objects in the input image.

However, one may want to explicitly encourage sparsity by including something to that effect in the training objective function. (Nair & Hinton, 2010a) and (Ranzato et al., 2006) present two methods for doing so. Both use a running historical average of each unit's activity, and reduce activity when that historical average is large, to arrive at the desired mean activity.

Another approach is to enforce sparsity with a hard constraint: by using multinomial latent variables. (Larochelle et al., 2010) did some experiments with those, but not very thoroughly, because the paper was mostly about another model. However, this is more restricting than just sparsity, because it groups the latent variables and then enforces that no more than one per group can be active. Fully general sparsity is enforced in the Cardinality Restricted Boltzmann Machine (Swersky et al., 2012), where the hidden units can have any state they want, as long as no more than some specified number k of them are on.

[11] However, it does turn on more often, so on average it does convey more information (its entropy is higher).

1.8.2 Invariances

Invariances are highly valued by some researchers. (Nair, 2010) even describes the whole visual processing task as an invariance task. The idea motivating the search for invariances is that many types of transformations of input can be large changes in terms of Euclidian distance in the raw input space, but minor changes semantically. For example, if the data is audio data, recorded from a person speaking, then he may speak a little slower, or with a higher pitch, and thus change the data dramatically in terms of raw input, while still saying exactly the same words and conveying the same message. A good listening system, therefore, has to produce some outputs that are largely invariant to pitch and speed. Conversely, a system that is invariant to such features of the input may well be focusing on the high level meaning. That is why some researchers attempt to force their systems to have such invariances.

For multilayer networks, however, it is important to distinguish between output invariance and intermediate representation invariance. The desired invariances are mostly network output invariances, and this does not mean that the internal representation has to be invariant as well. Forcing invariances on layers other than the output layer may, therefore, be counterproductive.

In image processing, a system may seek to have output invariant specifically to translation, rotation (two or three dimensional), scale, lighting (LeCun et al., 2004), partial occlusion, or other features that the researcher deems to be “inessential” aspects of the input. Alternatively, a system can try to be invariant to less well-specified aspects of its data, as is done in (Wiskott & Sejnowski, 2002), where the system is trained to be invariant to whatever is changing on a short time scale, without being told explicitly what that is.

How to make invariant units

Some built-in exact translation invariance: convolutional DNN’s with max-pooling Convolutional DNNs with pooling (also known as subsampling) (LeCun et al., 1998; Krizhevsky et al., 2012; Abdel-Hamid et al., 2012) have some translation invariance built into their architecture. The first several layers of these networks are alternating convolution layers and pooling layers. A convolution layer applies a small (e.g. 5x5) local linear filter at all locations in the preceding layer, followed by a nonlinearity like the logistic function $y = \frac{1}{1+\exp(-x)}$.

A pooling layer follows a convolution layer (which typically has many units) and summarizes the information of the convolution units in a smaller number of units. A unit in a pooling layer receives input from a small neighbourhood (e.g. 2x2) of adjacent convolution units in the layer below. The state of the pooling unit is either the maximum or the sum or the mean of the states of those convolution units. Correspondingly, it is called max-pooling, sum-pooling, or average-pooling. The effect of max-pooling is that the strongest convolution filter response in a 2x2 region is passed on to the next layer, through the pooling unit, but *which* of those 2x2 positions produced that strongest response is unknown to the next layer. Thus, the state of the units in the next layer is invariant to small translations of the input, if they’re small enough that the strongest filter response within a pool stays within that pool and no stronger response is moved (translated) into the pool.

However, the credit for the good performance of convolutional DNNs with pooling is hard to attribute, because these networks also use weight sharing, which is seen by many as a good idea in general, and the subsampling serves not only for the invariance but also to reduce the number of units in the network, which is often desirable given the large numbers of units that convolutional DNNs typically have. Unfortunately, researchers rarely choose to disentangle these effects.

More general, approximate invariances using calculus: Lie transformation groups (Simard et al., 1996) approximates well-defined image transformations (such as rotation) linearly in pixel space. These approximations can be used to make template matchers or feature extraction systems which are insensitive to small transformations along the invariance directions. For bigger transformations, e.g. bigger rotations, this approach fails because the effect of the transformation on the raw input values is no longer well approximated by a linear function.

Brute force for arbitrary well-defined transformations with few degrees of freedom (Frey & Jojic, 2000) uses iteration to simply try a variety of transformations of the input, in an attempt to find one that the classifier can deal with (both at training time and at test time). This is not restricted to small transformations, but if the transformation has many degrees of freedom, then the space of transformations is too large to search all of it.

Using a transformation function to create more training data An alternative approach, when one has a transformation function that describes the desired invariance, is to use it to create more training data. See for example (Hinton, 1987; Simard et al., 2003; Decoste & Schoelkopf, 2002; Ciresan et al., 2010). This has the advantage that the specification of the model architecture does not get more complicated, and that it can be done with any type of model. (Ciresan et al., 2010) took this approach very far and achieved impressive results: they use a simple model architecture, but because they made a lot of extra training data, it performs well.

Like the above brute-force search approach, the approach of making extra data has the weakness that only a small space of transformations can be used exhaustively, but that problem takes a different form, here. As long as one can draw samples from the space of transformations, one can generate a nearly infinite dataset on the fly, which on the surface may sound quite sufficient. However, if it's a large space, then learning will inevitable see only a small fraction of it, and good results will require good generalization. If the learning system is good at generalizing, then the method of expanding the dataset using transformations can do well, as (Ciresan et al., 2010) showed. However, it is less direct than building the invariance into the model, and therefore tends to be *much* less efficient. That is the main weakness of this method.

The method differs from what happens in convolutional DNNs in an important way: making extra data only enforces invariance at the output layer: intermediate representations are not forced to be invariant. This can be an advantage: while the state of output units should indeed be invariant to various transformations, it is not at all clear that the same applies to hidden units.

Breeder learning: learning to create more training data The transformation function that is used to create additional data is usually a fairly simple one, applied directly on the raw data, such as translation or scaling on an image[12]. In a more semantic representation of data, such as the representation that might be used by hidden units in a DNN (Bengio et al., 2013a), or the representation that goes into a data generation system that's handmade for this purpose (Nair et al., 2008), far more changes can be made that keep the transformed data looking, subjectively, reasonable. The "breeder learning" method

[12] Or, more generally, affine transformations.

uses this idea to partially learn the transformation function (which generates more training data) (Nair et al., 2008).

Breeder learning requires a parameterized black box data generator (not learned), which we can denote as a function G: *code* → *image*. The general idea is that the system learns to produce codes that $G(\cdot)$ will turn into meaningful training data. The procedure starts by learning a function H: *image* → *code*, implemented as a deterministic neural feedforward network. $H(\cdot)$ learns to approximate the inverse of $G(\cdot)$, in the vicinity of the training data.

If $H(\cdot)$ is learned successfully, then $G(H(x)) \approx x$ for $x \in$ training data. When that is achieved, good variations on training data case x can be made by $G(H(x) + \delta)$, where δ is a random small vector in code space, i.e. the input space of $G(\cdot)$ and the output space of $H(\cdot)$. The label of the variation can be the same as that of x.

In (Nair et al., 2008), $G(\cdot)$ is a simple computer program that uses a physical model of handwriting: it has a "pen" (which has some parameters), which is moved around by springs (whose varying stiffnesses are also parameters). Thus, the *code* for an image consists of a few numbers describing the pen, and some more numbers describing the behaviour of the springs.

Writing a parameterized data generation procedure is a powerful way of describing data, which is explored in a different way in Section 2.

1.8.3 Local connectivity

When there is a notion of relatedness between the various dimensions in the input, such as proximity of pixels (in the case of pixel image input), one may wish to have units that are connected to only those input units that represent a (small) neighbourhood. With image data this means that a unit is connected to a specific region of the input. With audio data this could mean that a unit is connected only to those inputs representing a small band of sound frequencies (Abdel-Hamid et al., 2012). If a group of units are "locally connected", then the closeness measure applies to those units as well, so the next layer can again be locally connected.

The main appeal of locally connected units is that local connectivity in the lower layers of an DNN goes well with the idea of hierarchically structured data (see Section 1.2.1).

Another advantage has to do with the number of model parameters. If the local connectivity is built into the model, then the model will have fewer parameters than it would have with full global connectivity. This can be advantageous if it helps as a regularizer. It can also speed up the computations.

How to make locally connected units

In some situations, the learning will choose approximately local connectivity without being forced to. Rarely will all units have local connectivity, but many latent units might have strong connections only to input units in a small region. CD-1 training often results in many fairly locally connected units (Hinton, 2006).

Local connectivity can also be encouraged by including a component to that effect in the objective function. L1 weight decay can cause locality, by encouraging units to have few connections, which typically end up being to units representing a neighbourhood.

Another approach to engineering local connectivity is to build it into the model architecture, either with parameter sharing (LeCun et al., 1998) or without (Gregor & LeCun, 2010). This can serve as regularization, by reducing the number of parameters, and as a computational efficiency optimization, because unlike with L1 weight decay, where many connections still exist albeit with strength zero, with these methods there really are fewer connections, so the computations can be optimized to really skip the eliminated connections.

1.8.4 Robustness and independence

If a unit in a network can only make a useful contribution if all other units in the network do exactly the right thing, then the system isn't very reliable. In particular, it might fail to do well on data on which it wasn't trained, i.e. data where the subtleties of how the various input variables go together are slightly different from what they are in the training data. It would be much better if a unit can do a reasonable job even if its neighbours aren't behaving exactly the way they've been behaving on the training set. This is the idea behind the *dropout* regularization method for deterministic feedforward neural networks (Hinton et al., 2012).

The method is quite simple: every time when a gradient is computed on a training case, some fraction[13] of the units[14] in the network are temporarily eliminated, i.e. their state is set to zero so that they don't get to do their job. This forces the remaining units to learn to operate without getting to count on the presence of their neighbours. As a result, they learn to become more independent, and they work better in unexpected situations i.e. on test data. This has worked well with both logistic units and rectified linear units.

This is a great regularizer, but at test time the noise isn't helpful. At test time, we don't drop out units 50% of the time, but instead we halve their outgoing weights. The effect is roughly[15] the same in expectation.

Denoising autoencoders (Vincent et al., 2008) can be thought of as a special case of dropout, where only the input units get dropped out.

1.9 Meta-parameters in DNNs

DNNs always have quite a few meta-parameters: there are many choices to be made about architecture (number of layers, layer sizes, unit types), optimization strategy (learning rate schedule, momentum, etc.), objective function (regularization constants), and data preprocessing. Sometimes, one has strong intuitions about what strategy is best; at other times, one would ideally like to try many alternatives, preferably in an automated fashion. In the past few years, this issue has been receiving more attention, and some advances have been made. One of those advances is the crude but effective method of using large clusters of computers to try many alternatives (Erhan et al., 2009). Another is the use of machine learning itself to do the meta-parameter optimization (Snoek et al., 2012; Swersky et al., 2013; Bergstra et al., 2011; Snoek et al., 2013).

[13] 50% is a reasonable choice in most situations.

[14] This includes both the hidden units and the input units, though (Hinton et al., 2012) recommends "dropping out" only a smaller fraction (20%) of the input units.

[15] It isn't exactly the same because there are nonlinearities in the network.

1.9.1 Model architecture parameters

Model architecture has the largest number of meta-parameters and receives the most attention. Even when we're sticking to fairly established types of networks, there are many choices to be made.

Number of layers

There seems to be little agreed wisdom as to what is a good number of layers to use. (Hinton et al., 2006) uses three layers of hidden units, but does not explain how that choice was made. (Salakhutdinov & Murray, 2008) suggests that for getting good test data log probability in a generative model of the MNIST distribution, more than one hidden layer (i.e. an RBM) didn't help. (Mohamed et al., 2009) chooses the number of layers using validation set performance, and found that for their task up to 8 layers can help.

Larger numbers of hidden layers have recently become feasible and successful, because of advances in optimization (Martens, 2010; Sutskever et al., 2013). However, there are still no clear conclusions about this question.

Width of layers

Usually, all layer sizes are of the same order of magnitude, unless there are very specific reasons to make them different, such as in (Salakhutdinov & Hinton, 2009b). (Hinton et al., 2006) uses a top layer that is significantly larger than the other layers, to have a powerful generative model[16]. In general, large layers often help, but the question is which layers benefit most from the additional compute resources required to train large layers. (Erhan et al., 2009) suggests making all layers have the same size, which seems to work reasonably well and eliminates some meta-parameters. On the other hand, one must keep in mind that not all layers serve the same purpose.

Unit types

The most commonly used unit in DNNs today is still the logistic unit[17], which produces outputs in the range $(0, 1)$ through the function $\sigma(x) = \frac{1}{1+\exp(-x)}$. However, rectified linear units (Nair & Hinton, 2010b; Krizhevsky et al., 2012) are less susceptible to the problem of vanishing gradients, and are gaining some popularity. Another idea that is being experimented with (including in Section 2) is a unit that squares its input.

1.9.2 Meta-parameter optimization

Tweaking by hand

Still the most common approach is to optimize them by hand, and simply report the chosen value (or to not mention anything at all in the paper). This method is a natural extension of tweaking the algorithm

[16] In a DBN, one may view the top layer as generating high-level descriptions of data, and the other layers as a program for turning those descriptions into the right data format. With this interpretation, the finding that the top layer had to be bigger in (Hinton et al., 2006) suggests that the task of generating high-level descriptions of handwritten digit images with an RBM requires more units than the task of translating those high-level descriptions into pixels. To an extent this agrees with the set-up that ended up working best in Section 2.

[17] The tanh variation, which has a range of $(-1, 1)$ through the function $\tanh(x) = \frac{\exp(2x)-1}{\exp(2x)+1} = \sigma(2x) \cdot 2 - 1$, is also popular.

itself by hand, and requires few experimental runs and no additional programming effort. That often makes this the only feasible method for experiments that require large amounts of computation.

However, being only weakly separated from algorithm design, it increases the risk of inadvertently tuning to the *test set*. For large training and test sets drawn from the same distribution, this cannot really be much of an issue, but for small test sets it might. Machine learning competitions now solve this problem by publishing training and validation sets, but keeping the test set secret and allowing only a limited number of model evaluations on the test set.

Another drawback of hand-tuning meta-parameters is that meta-parameter sensitivity goes unnoticed to the readers of the paper, and quite possibly to the researcher himself.

The third problem with this method is that the researcher does not know well whether the chosen meta-parameter settings are close to optimal - unless he puts a lot of time into investigating them carefully.

The problem that most discouraged me from this approach is that it can take quite a bit of the researcher's time.

Grid search and random search

The next least programming intensive approach is to do a grid search over some meta-parameter values, as was done quite massively in (Erhan et al., 2009; Pinto et al., 2009). This method requires a bit more programming effort, but not much: it involves little more than a few nested loops, and some outcome visualization effort. This approach has the advantage that meta-parameter interactions and sensitivity can be identified, and that there are more optimality guarantees. Also, because it requires more programming effort, researchers are more likely to remember to do meta-parameter selection based on a validation set, instead of the test set.

The obvious disadvantage of this method is that it requires many runs to be performed, and that the number of meta-parameters that can be tuned is only logarithmic in the number of runs. For toy problems this solution may work fine, but for larger experiments, only a few meta-parameters can be tuned well.

A big improvement on grid search is random search, which is not more difficult to implement. It may sound rather trivial, but it can be much more efficient (Bergstra & Bengio, 2012). Imagine there is a meta-parameter that, in practice, has no effect. Grid search would suffer a lot from adding this to the collection of searched meta-parameters. Random search, on the other hand, wouldn't suffer at all.

More sophisticated automated meta-parameter search

Grid search and random search have the advantage of being automated, that many machines can be used, and that experiments can proceed without the researcher's intervention. However, they are clearly not ideal. Imagine that a setting a_1 for meta-parameter A never seems to work well, and the same for some setting b_1 of meta-parameter B. Then it seems pointless to perform the run that combines a_1 and b_1: more promising values should be given priority in the search. Such heuristics can be implemented in computer programs, to some extent, for example using non-parametric regression methods to predict both the performance of meta-parameter settings and the variance of that performance. This method was used in (Tieleman & Hinton, 2009), and has recently been studied far more thoroughly (Snoek et al., 2012; Swersky et al., 2013; Bergstra et al., 2011; Snoek et al., 2013).

The advantage is that the search is performed more intelligently. The disadvantage is that it requires more programming effort. However, efforts are underway to make proper software packages for this; clearly, this is not something that every researcher should have to implement on his own.

1.10 Conclusion

DNNs currently enjoy great popularity, but are difficult to study systematically. The reason is probably that they are difficult enough to get to work well, let alone study them with thoroughly controlled experiments and theoretical investigations. Large numbers of meta-parameters, long run times, and the black box nature of neural networks, are significant obstacles.

Thorough empirical analyses, such as (Erhan et al., 2009), are of limited relevance because they necessarily involve shorter runs than the long runs that are more typical of research in practice.

The large number of meta-parameters means that everybody's experiments are different in at least a few ways, and therefore difficult to compare.

Theoretical results can sometimes lead to, or support, important insights. Often, however, the bounds that can be proven are too loose to be relevant for practitioners.

It is often tempting to draw grand conclusions, and generalization is of course part of a scientist's task. However, the above difficulties have meant that many conclusions about DNNs had to be revoked later on. Learning by CD gradient estimation works, but not, as one might have initially thought, because one step of Gibbs sampling is enough to get reasonable samples from the model. Learning using Persistent CD (Tieleman, 2008) gradient estimates works, but not because there is sufficient mixing in that Markov Chain (Tieleman & Hinton, 2009). Learning using variational inference works, but not necessarily because the inference is close to correct. Rather, there is a subtle interaction between the learning and the gradient estimation, which makes the inference correct, or in the case of PCD makes the mixing work. Also, it is my experience that the simplest of sanity checks, like histograms of unit activities and gradients, often produce highly unexpected findings. All this should caution against drawing conclusions, however appealing they may be.

Despite these obstacles, considerable progress is being made in solving important practical problems. Classification performance of carefully manually tuned algorithms on sometimes somewhat artificial datasets may not be exactly the "meta objective function" that one truly cares about, but it can serve well as a first filter. New models are being proposed frequently, and bit by bit their properties are learned, even if only by observing on what types of datasets and tasks they work best. For now, this is a useful way to continue. However, continued investigation of the effects and interactions of all those meta-parameters, as is happening in research today, is the right way forward.

Chapter 2

Autoencoders with domain-specific decoders

2.1 Introduction

This chapter describes a method of learning to understand images, that is based on learning to generate images from small codes.

2.1.1 General idea

This approach is based on including domain-specific knowledge in the learning system. Adding domain-specific knowledge to neural networks is quite a common practice, and often leads to improved performance. Machine learning combined with human knowledge is, apparently, more powerful than machine learning alone.

Building knowledge into a data recognition system

In image processing, the most ubiquitous example of this practice is the use of convolutional networks with pooling (see Section 1.8.2). We believe that in the lowest layers, feature detectors should be local and should be doing more or less the same in every location, so we force this replication. If we also believe that some invariance to location is desirable, we add pooling to force this invariance.

In audio recognition, most systems use a lot of domain-specific knowledge: the Fourier transform, Cepstral transform, and sometimes convolution over the time (Lang et al., 1990) or frequency domains (Abdel-Hamid et al., 2012). Some of this is data preprocessing; some of it is built into the neural network architecture.

The above are examples of the most obvious approach to using domain-specific knowledge for building a data recognition network: simply build the knowledge into that data recognition network.

Building knowledge into a data generation system

In this chapter, I study an alternative: building the knowledge into a data generation system, which can then guide the training of a data recognition network. This approach is called *analysis by synthesis.* It was described by (Halle & Stevens, 1959; Halle & Stevens, 1962) for the domain of speech data, and has been used in many machine learning models. Autoencoders and probabilistic generative models (especially causal ones like SBNs) clearly embody this principle. In computer vision the approach is ubiquitous, with Active Appearance Models (Cootes et al., 2001) being an elegant illustration of its power (see also Section 2.8.1). (Nair et al., 2008) takes the idea further by treating the data generation system as entirely a black box.

Recently, the work on transforming autoencoders started to introduce more specific network architecture ideas in autoencoder-like systems (Hinton et al., 2011). The approach of that paper inspired the work presented in this chapter. Its central innovation is to introduce a type of composite unit in the code layer of an autoencoder. These composite units contain not just one value, but a fixed number of values (3, in that paper), with enforced interpretations as a triple of (Δx, Δy, intensity), describing the appearance of a visual component of the image, like an edge. These units are called "capsules" because they encapsulate a fully detailed description of one visual component. A model typically has a modest number of such capsules (like 30), each describing its own visual component.

The question is of course how to enforce the interpretation of 90 code units as $30 \cdot 3$ capsule descriptions. The main method that (Hinton et al., 2011) uses for this is to slightly modify the way the training data is presented to the model[1]. The work presented in this chapter takes a different approach: the data is presented in the standard autoencoder fashion, the encoder is general-purpose, but the decoder is specifically designed for the concept of visual capsules[2].

More concretely, this means that the extraction of a "code", i.e. a description of the image, will be performed by a general-purpose neural network, while the reconstruction of the image from the code will be performed by a system that has been enriched with knowledge that is specific to the domain of visual data.

The motivation for this approach lies in the fact that human knowledge about image generation is easier to describe to a computer than our knowledge about image recognition. Put more plainly, engineered computer graphics is easier than engineered computer vision. The proposed method consists of a general-purpose encoder network, which learns to produce a vector graphics[3] description of the input image (i.e. computer vision), and a domain-specific decoder that renders the vector graphics, producing the output of the system (i.e. computer graphics). Together, the two form an autoencoder: the learning tries to make the rendered output as close as possible to the input. See Figure 2.1. We tell the system (part of) how to verify that a particular vector graphics description is an accurate description of the input image. Theoretically speaking, that doesn't guarantee that we can also extract a good description (unless P=NP), but we can train a powerful encoder to do that job quite well.

[1] The paper also uses some capsule-specific architecture in both the encoder and the decoder.

[2] Capsules can be applied in this fashion to any type of data for which we can describe a process that generates such data from parameter values. (Jaitly & Hinton, 2013) does it for speech data.

[3] The expression "vector graphics" may suggest that it deals only with straight lines, but straight lines are only the simplest form of vector graphics. In general, vector graphics refers to math-heavy descriptions of images, and is usually contrasted with raster graphics (a.k.a. bitmaps). The model of this chapter describes images using such concepts as rotation and shearing, and makes good use of the mathematical properties of those operations for the purposes of learning and describing recursively structured objects. Linear algebra is an essential ingredient of this technique, so it must be called vector graphics. However, it should be noted that vector graphics is a broad category that includes many different techniques.

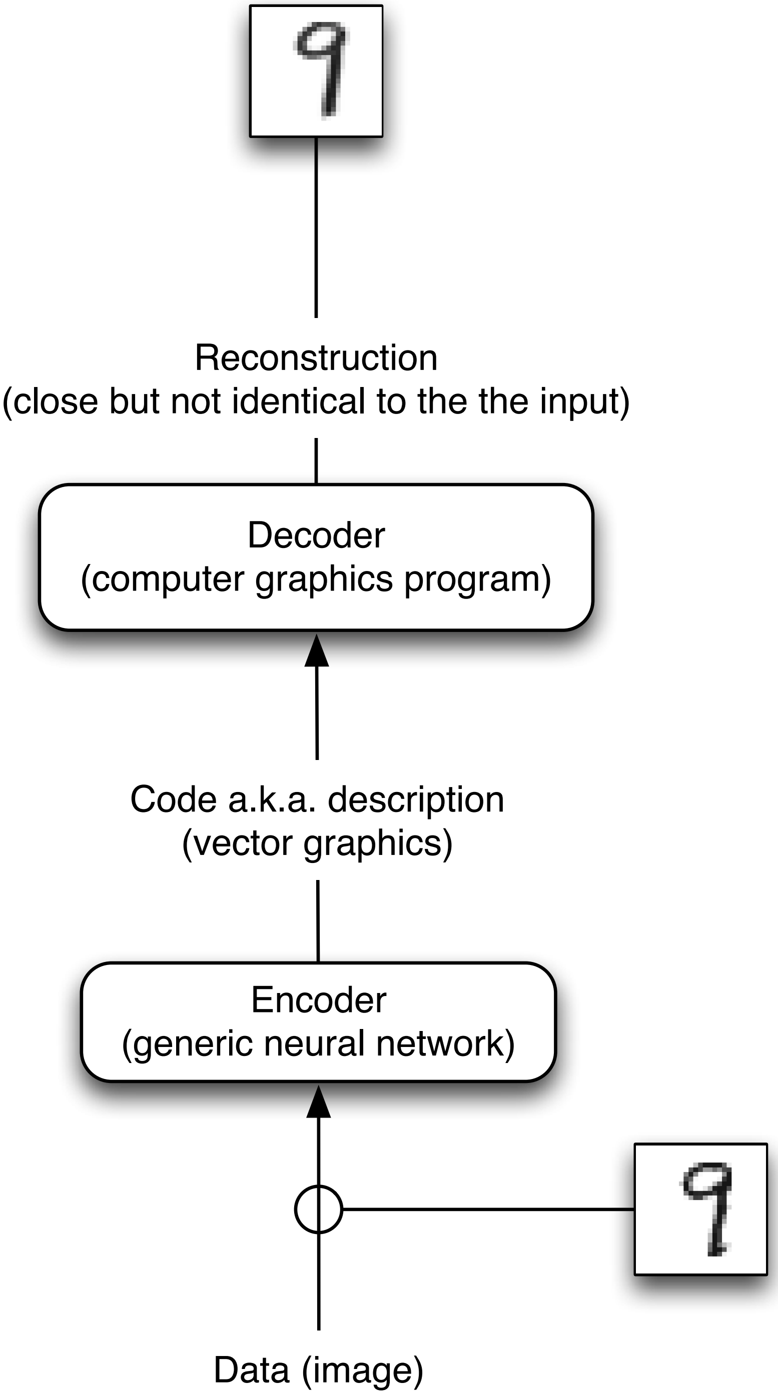

Figure 2.1: The computer graphics-based autoencoder

Interpretation as a sparse-code autoencoder

A second motivation is found when one looks at sparse coding (Olshausen & Field, 1996). As an autoencoder, this is more or less the following: a regular autoencoder, with a single hidden layer, that has been forced to use sparse codes. After training, each code unit in such a sparse autoencoder has a fixed[4] contribution that it makes to the reconstruction whenever it's activated. Those standard contributions are different for each code unit, but when one visualizes them for all code units, one often sees that many units' contributions are almost the same. For example, there might be 100 units that each contribute a straight line, the only difference being just a minor change in location or angle, or some other difference that may be quite significant in terms of squared difference in pixel space, but that to the human eye looks like a semantically nearly insignificant difference. Alternatively, and more commonly, there will be many units that are all slightly different Gabor filters.

If one uses such an autoencoder to create descriptions of images, for further processing by another system, all of this near-replication feels sub-optimal. Instead of describing which one of 100 minute variations on the concept of "a line" is found in the image, it would be better to describe what location, rotation, etcetera, is applied to one basic version. That would convey the meaning in a distributed representation, with higher precision, and would be better for the next stage of processing. That is exactly what the capsules-based autoencoder is attempting to produce.

As a result, we find much less of this kind of replication in the decoder[5]. Even more strikingly, the decoder units don't learn Gabor filters at all. This suggests that all this replication, and even the Gabor filters as a concept, are no longer necessary because we added parameters for the location and other attributes of the decoder units[6].

Final use of the system

After the autoencoder has been trained, we discard the decoder (the graphics program), and are left with an encoder network that produces vector graphics descriptions of images. A vector graphics description of an image is easier to interpret, for a computer, than a pixel array description of the same image. We can use these descriptions as input to another learning algorithm, giving it an easier job than it would have if it would use the pixel-based description as input. An example of that is demonstrated in this chapter.

2.1.2 Purpose of this work

The purpose of this project is not in finding the solution to an application-level task. Instead, it is a proof-of-concept, the concept being the use of a partially hard-coded data generation system to train

[4] Or almost fixed: a code unit's activation can be quantified by a scalar, representing the extent to which the unit is activated. The standard contribution to the reconstruction is then multiplied by that scalar.

[5] Part of the decoder is handmade and fixed, but the typical contributions to the reconstruction, now called "templates", are learned, and what they learn has little of this repetition.

[6] Gabor filters as output units have several strengths. One strength is that they're useful for slightly moving over edges in the output: adding a multiple of a well-chosen Gabor filter to an image has the effect of slightly moving an edge. Another strength of Gabors is the way they can be combined linearly. After one Gabor unit has moved an edge a bit, another Gabor can move it some more, or move it back a little. Thus, high precision in terms of location can be achieved by linearly combining multiple Gabors. In mathematical terms: a Gabor is a localized sine wave, and linear combinations of out-of-phase sine waves can make sine waves of any phase. Thus, if your output units are required to operate in a fixed location, Gabors are an excellent choice, because linear combinations of them can be used to approximate a (slightly) variable-location Gabor. However, when output units are no longer restricted to one location, this trick of combining Gabors no longer adds value. This may be the reason why allowing translation made the Gabors disappear: the model no longer needs to restrict itself to output units that can be slightly moved by linear combination.

a data recognition system. I implement the data generation system as a component in an otherwise standard machine learning set-up (an autoencoder), and show that it leads to good data recognition.

In the process, I implement a sophisticated partially-learning data generation system in an autoencoder, which is then trained using gradients. The next twenty pages are full of diagrams describing the data generation system. I describe it one component at a time, because it's too complex to draw the details of all components in a diagram that comfortably fits on one page. The computer program implements that system and therefore has an internal organization that's essentially the Python version of these diagrams. Computer-assisted differentiation of the entire system then allows a gradient-based training approach.

Thus, the second take-home message for the neural networks research community is that it's quite doable to create systems in a gradient-based learner that are many times more complicated than typical neural networks. The most complicated neural networks combine dropout, multiple layers, convolution, pooling, and perhaps a few more tricks, but this still rather bleakly contrasts with the sophisticated componential systems that engineers in every other discipline use. That difference may lead one to conclude that neural networks are unsuitable for sophisticated designed structure. This work proves otherwise: we can create systems that have both the sophisticated componential structure that other engineers take for granted, and the ability to automatically train most of the components using a dataset.

To enable this combination, a few guidelines must be kept in mind. First of all, the entire system must always be differentiable. Second, we must ensure that as the gradients are propagated through the various components, they don't vanish or explode too badly. Third, we must use a numerical optimizer that can learn dozens of different groups of parameters at the same time, all of which have different semantics, which usually leads to different gradient characteristics. Optimizers that use, for example, a single fixed learning rate, are ill-suited for such a task.

2.2 The encoder

The encoder is emphatically not the crucial ingredient of this model. Most of this project is about how to give the decoder the tools to generate data from small meaningful codes. However, an encoder is still needed, so here's what I did.

For the encoder I used a deterministic feedforward DNN. Its input is the original image; its output is an attempt at a vector graphics description of that image, in the language that the decoder requires. It is a fairly standard neural network. No part of the encoder is specialized for this task; no part of it is particularly innovative; and the details are not central to the contribution of this thesis. The fanciest bit of it is that I used dropout (see Section 1.8.4) for regularization[7], and skip-layer connections for efficient learning.

Of course, just saying that the encoder is a feedforward DNN leaves many things to be decided: the number of layers, the type of units, regularization details, etc. I did not investigate all of these very extensively, because the defining component of the system is not the encoder but the decoder. Here, I describe the encoder architecture that I ended up using, without claiming that it is the best possible for the task. The reader is encouraged to skip these details for now, and focus first on understanding the decoder.

A diagram of the encoder is shown in Figure 2.2. For details, read the following subsections.

[7] On MNIST, it made for significantly better reconstruction and classification performance.

I did very little experimentation with the encoder architecture, because the emphasis of this work is on the decoder, and the encoder was working just fine. However, in an impressive case of "don't assume that you know things without having tried them"[8], a simple one-hidden-layer encoder (with the same total number of units, all logistic) turned out to work just as well on the MNIST dataset. This strongly supports the idea that the sophisticated decoder is indeed the crucial ingredient. However, I still believe that computer vision is not an easy task for a neural network. MNIST and TFD aren't quite complete computer vision tasks: they're small, they're grey scale, and MNIST is particularly easy because most pixels are near-saturated. High-resolution RGB images of interesting scenes are more challenging. For such future work, I suspect that a more sophisticated (multi-layer) encoder will make a big difference.

2.2.1 Number of layers and connectivity

Having three layers of units between the input (the image) and the output (the vector graphics description) worked well.

This means that there are four layers of weights: input → hidden1, hidden1 → hidden2, hidden2 → hidden3, and hidden3 → output. To avoid the problems of vanishing gradients, I added skip-connections from the first two hidden layers straight to the output. Thus, if there would be vanishing gradients, at least the chain input → hidden1 → output would still learn (it's not that deep), and after that's learned, hidden1 would have meaningful values, so the chain hidden1 → hidden2 → output can be learned, etc. These skip connections made a big difference in reconstruction error and in classification of the MNIST dataset: with skip-connections the error rate was around 1.7%, and without skip-connections it was 9.9%. It should be noted that with proper initialization (Sutskever et al., 2013), skip-connections would probably not have been necessary.

2.2.2 Layer size and type

After some exploration, I found that a decent set-up is to have first, two hidden layers of units that use the logistic nonlinearity, and then a third hidden layer of units with a squaring nonlinearity[9]. For simplicity, all three of those layers have the same number of units.

Some exploration showed that a total of 2,000 hidden units is a decent choice[10], so each hidden layer has 666 units.

All of this could probably be tweaked for better performance, but these values are simple and work well enough. Again, the encoder is not the defining component of the system.

2.3 The code language and the decoder

The decoder architecture is engineered using domain-specific knowledge. It is designed to decode codes in a specific code language; a specific type of vector graphics description. That language and the design of the decoder are described in this section.

[8] My thanks to Radford Neal and Rich Zemel for requesting this.

[9] The system also works quite well with exclusively logistic units, but using squaring units in that third layer did reduce the MNIST classification (see Section 2.6.6) error rate from 2.4% to 1.7%.

[10] With a total of 1,000 hidden units (instead of 2,000), MNIST classification error rate went from 1.7% to 6.4%.

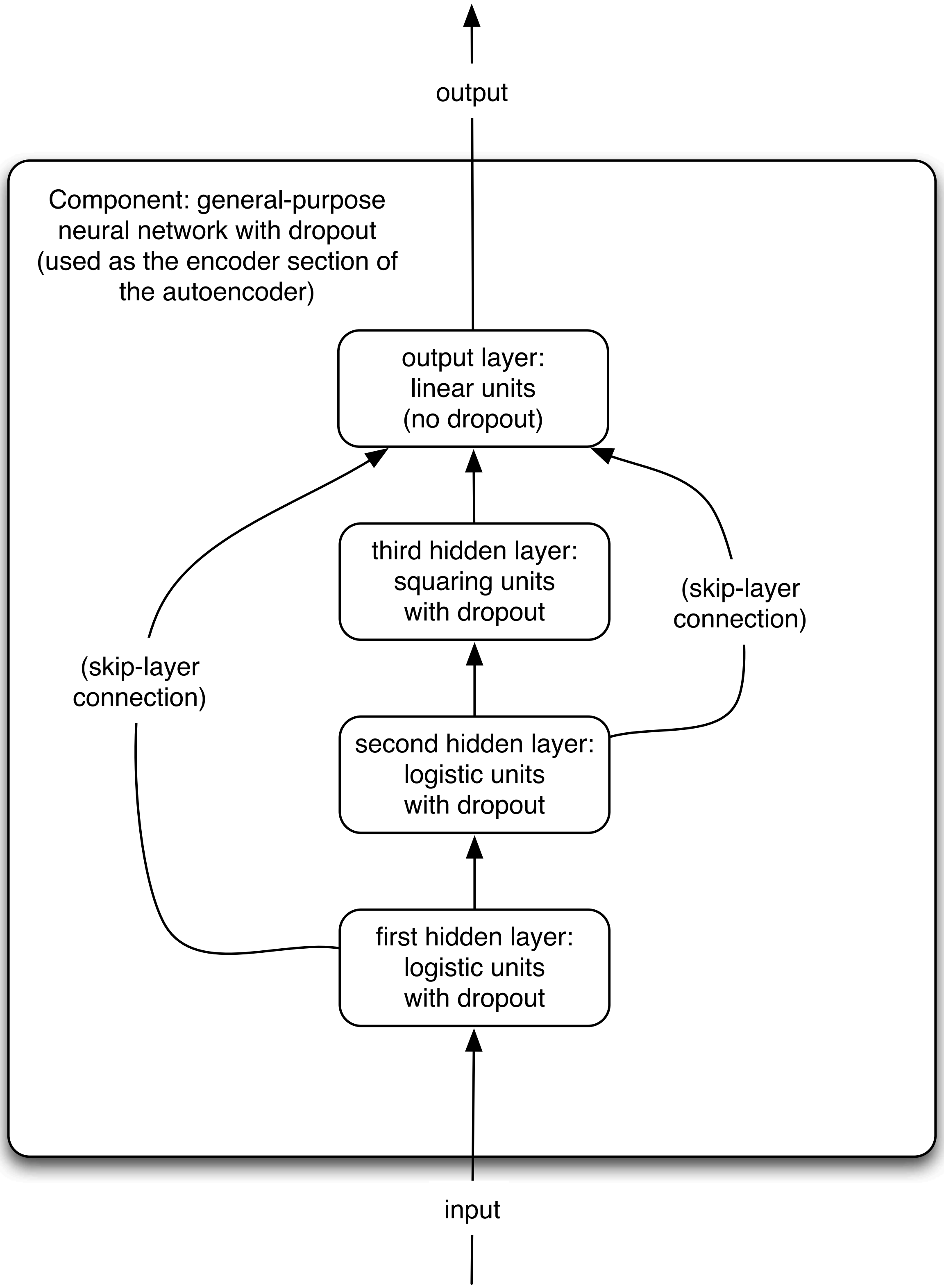
output
Component: general-purpose neural network with dropout (used as the encoder section of the autoencoder)
output layer: linear units (no dropout)
third hidden layer: squaring units with dropout
(skip-layer connection)
(skip-layer connection)
second hidden layer: logistic units with dropout
first hidden layer: logistic units with dropout
input

Figure 2.2: Component: a general-purpose neural network with dropout (GPNN). It's a fairly standard DNN, with dropout, skip-connections, and two types of units. All connections are global: there are no restricted receptive fields, and there is no weight-sharing (of course this could be added). The output layer simply adds up all input that it receives. When this system serves as the encoder section of an autoencoder, the original data is the input, and the "code" will be the output. In our case, the "code" consists of the pose of all capsules in the model.

2.3.1 Capsules in transforming autoencoders

Because the decoder for this work is quite elaborate, it must be introduced step by step. The first step is the work of (Hinton et al., 2011), which introduces the concept of a "capsule". A capsule serves roughly the same purpose as a unit in the code layer of any autoencoder, but is more sophisticated and thus serves the purpose more effectively. In particular, it contains more than just one value, and the designer of the system imposes an interpretation on the various values.

In (Hinton et al., 2011), a capsule represents a component of an image, like an object in a scene, with a specific location and brightness. Thus, the capsule contains three values, which together make up the "pose" or configuration of the image component: x, y, and brightness. Since this pose includes not only the geometric position but also the *intensity*, we might call it an "iPose".

Also, every capsule has a learned image "template" associated with it: a platonic ideal of the object. This template is constant, but its appearances vary: it shows up at different locations and with different brightness values. In the decoder, the capsule essentially copies the template into the model output, at the location and brightness that are specified in the three code values a.k.a. the "pose". Thus, the "computer graphics-based" decoder is limited to taking a fixed template per capsule, translating it in 2D, and multiplying it by an intensity. The pose is different for each data case and is produced by the encoder in a parametric way. A capsule's template, on the other hand, is the same for every data case, i.e. it is a learned constant and does not come from the encoder.

It is important to carefully distinguish between a capsule's instantiation and its intrinsic nature. The intrinsic nature is just the template. This is learned, but after the model has been learned it becomes a constant. While the model is learning, it is constant in the sense that it's the same for every data case. The instantiation, on the other hand, is typically different for different data cases.

2.3.2 More powerful capsules: enabling full affine transformations

The next step is to add more parameters to the pose: instead of just translation and intensity (2+1 values), capsules now allow general affine transformations and intensity. Thus, they have 6 + 1 values in their pose: six for the details of the affine transformation, and, as before, one for the intensity.

This adds much flexibility to the system. The capsule can now project its template not only in any location and with any intensity, but also with any scaling (along two axes), rotation, and shearing.

This brings us much closer to the way computer graphics systems work. In a computer graphics system, a scene is described as a long list of objects, each with its own transformation. Those transformations are described by homogeneous coordinate transformation matrices in three dimensions. In this work, we use just two dimensions to keep things simple, but the principle is the same.

The reason why computer graphics systems use homogeneous coordinate transformation matrices is that they are incredibly convenient. To combine the effects of multiple transformations, one only needs to multiply their matrices. This makes it easy to use objects that are composed of parts, recursively if necessary. To get the position, orientation, scaling, etc., of a part of some composite object, one only needs to multiply together two matrices: first, the matrix that describes the coordinate frame of the composite object with respect to the global coordinate frame; and second, the matrix that describes the coordinate frame of the part with respect to the coordinate frame of the composite object. Matrix multiplications can be built into a differentiable graphics-based decoder in an autoencoder, so all these possibilities become available when capsules understand full affine transformations. In Section 2.3.4, the convenience and success of this approach is demonstrated.

Of course, the decoder in our learning system is humbler than a full-scale modern computer graphics system. This decoder does not deal with three-dimensional graphics, with lighting, with atmospheric influences, etc. In principle, nothing prevents that, but more compute power will be needed before all of that can realistically be done.

A diagram of the rendering (output generation) process of a simple (atomic) capsule is shown in Figure 2.3; details are described in the following subsections.

Multiple coordinate frames

The computer graphics framework is essential, especially for the system that uses composite objects, so I must introduce some computer graphics notation now.

The central concept is that of a coordinate frame. In computer graphics, coordinates are three-dimensional, but having only a two-dimensional world does not change the underlying ideas.

The most obvious coordinate frame is that of the world. One can think of this as the most official or absolute coordinates, like GPS coordinates. An insight that made computer graphics much easier was to accept that this is not the only coordinate frame worth thinking about. We also have the coordinate frame of an object (composite or atomic), and the viewer's coordinate frame.

Each of these is a different set of axes: they can all be used to describe the same locations, but the descriptions will look different. If we use homogeneous coordinates, the translation between coordinate frames becomes a matrix-vector multiplication. In a two-dimensional world, a location is normally described by just two numbers, x and y, but using homogeneous coordinates, those are made the first two values in a vector of three. The third value will always be 1 (that's a bit of a simplification, but for the purposes of this work it will do).

To translate a location description from object coordinates (homogeneous) to world coordinates (homogeneous), we simply multiply the coordinate vector by the (object → world) transformation matrix. If we then wish to know where that location is in the viewer's field of vision, we multiply that world coordinate vector by the (world → viewer) matrix. Because these two transformations are both linear, the composite transformation is linear, too, and is described by the (object → viewer) matrix, which we get by multiplying the (object → world) transformation matrix and the (world → viewer) transformation matrix. Naturally, the reverse transformation is obtained by multiplying by the inverse of the matrix: $(\text{viewer} \rightarrow \text{object}) = (\text{object} \rightarrow \text{viewer})^{-1}$.

Because the third value of every coordinate vector must be 1, the third row of every coordinate transformation matrix must be $[0, 0, 1]$, leaving six degrees of freedom.

The autoencoder in computer graphics language

Using this computer graphics framework, the autoencoder can be described as follows.

The templates are the objects. In their own coordinate frame, they are of unit size, i.e. a template pixel that's right in the middle of the template is at position (0.5, 0.5) in template/object coordinates.

The geometric part of the pose, i.e. the affine transformation (6 degrees of freedom), describes the transformation between the object coordinate frame and the world coordinate frame, which is also the camera coordinate frame[11]. This coordinate frame is also of unit size: an output pixel right in the middle

[11] In this simplified computer graphics system, having a separate camera coordinate frame doesn't help. If we would have a separate camera coordinate frame, the step from the object coordinate frame to the final (camera) coordinate frame

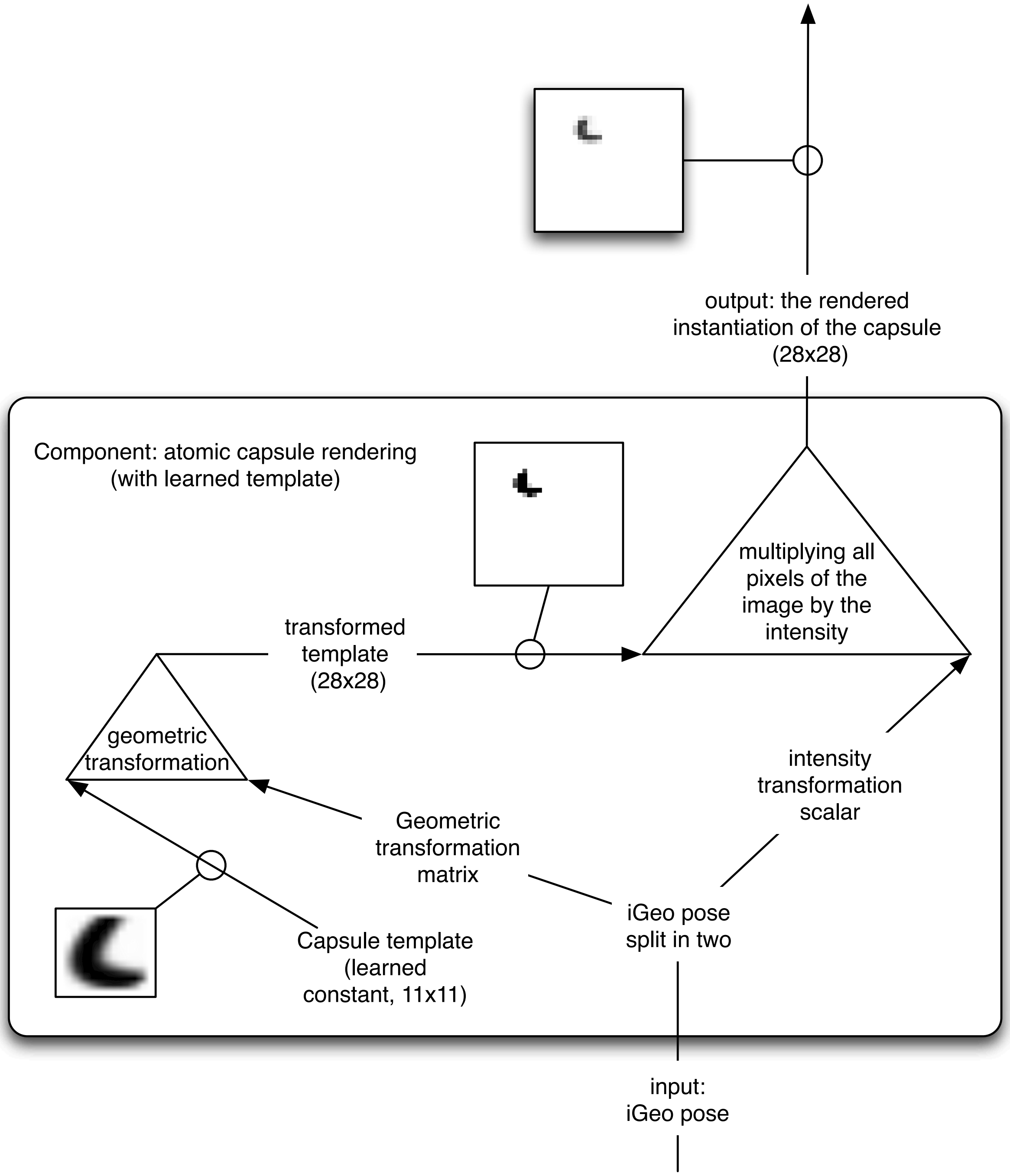
output: the rendered instantiation of the capsule (28x28)
Component: atomic capsule rendering (with learned template)
multiplying all pixels of the image by the intensity
transformed template (28x28)
geometric transformation
intensity transformation scalar
Geometric transformation matrix
iGeo pose split in two
Capsule template (learned constant, 11x11)
input: iGeo pose

Figure 2.3: Component: Atomic capsule rendering (ACR). "iGeo pose" means a pose specification that includes both the intensity transformation multiplier and the geometric transformation matrix. The little circles on some lines, and their attached square figures, illustrate an example of the type of message that's passed through that line in the system.

of the output is at world/camera coordinates (0.5, 0.5).

Transformation representations

There are several ways in which the pose could define the transformation between the object coordinate frame and the world coordinate frame.

Perhaps the most intuitive approach is that the six pose values that are dedicated to the geometric transformation are simply the first two rows of the 3x3 matrix (object → world). The third row is fixed to be $[0, 0, 1]$. That way, it directly tells us where a pixel of the template ends up in the output image.

However, there are other, better, representations.

The renderer-friendly approach

Rendering really requires the opposite of the above: given a pixel location in the output image, it needs to know where in the object(s) to look to find out what to draw in that output pixel. The corresponding location in the object frame (the template image) will typically not be integer, so we use a bilinear interpolation of the four surrounding template pixels.

Thus, what the renderer really needs is the (world → object) transformation matrix. Of course that's simply the inverse of the (object → world) matrix, and matrix inverse is a differentiable operation, so in theory it doesn't really matter which is represented by the pose. However, in practice, matrix inverses are best avoided. Having a matrix inverse in the decoder means that there can be unpleasant boundary cases and near-boundary cases which would lead to large gradients. Therefore, it's better that the geometric transformation part of the pose directly represent (the first two rows of) the (world → object) matrix.

The human approach

There is an even better way to represent transformations. Finding the natural representation of things is the theme of this entire research project, and geometric transformations, too, have more natural representations. How would a human like to describe the transformation? He might say something like this: "Okay, we have this template. We're going to rotate it 30 degrees counter-clockwise, shear it to the right 1%, then scale up horizontally by +120% and vertically by +15%, and then we're going to move it into the upper right corner of the output." That's quite different from writing down the first two rows of a homogeneous coordinate transformation matrix. It still involves six degrees of freedom, but they're expressed in a much more intuitive way; much easier for humans to deal with. Experiments showed that it's also easier for a neural network. Perhaps it wouldn't make a difference if we used a highly second-order-aware numerical optimizer like (Martens, 2010). However, with the relatively simple optimizer that I ended up using, disentangling the factors of variation this way certainly worked quite a bit better.

would proceed via an intermediate coordinate frame (the world frame), but there's no reason why the model couldn't simply learn to make the world and camera frames identical, or make the object and world coordinate frames identical. Normally, in computer graphics, we're dealing with a largely static world in which it makes sense to have objects in a standard configuration (aligned with the horizon) and a moving camera which is less restricted. However, in this simplified situation, we do not have that.

The order of the transformations

The more intuitive transformation description above describes the (object → world) transformation as a composition of four elementary transformations: rotation, shearing, scaling, and translation, in that order. This sequence of transformations is implemented as a sequence of multiplied transformation matrices; an illustration of the process (including some additional details which are explained later) is included at the end of this section in Figure 2.4.

The order matters here, i.e. these four transformations are not all commutative. To illustrate this, consider a rotation of 90 degrees clockwise, and a translation of 10 units to the right. If we apply rotation first and translation second, as the above example does, then (0,0) is translated to (10,0). However, if we first apply translation, which takes (0,0) to (10,0), and then rotation, we end up at (0,-10). The rotation still means that the object will be turned, but the translation now means something quite different, and less intuitive. This interaction is undesirable and can be avoided by applying the rotation before the translation. In fact, translation non-intuitively interacts with the other two elementary transformation (scaling and shearing) as well, so it must be applied as the very last.

Avoiding inverses

Thus, the human-style transformation description makes two important choices. First, it chooses to describe the (object → world) coordinate transformation, instead of the (world → object) coordinate transformation that the graphics renderer would prefer. The second choice is the order of the primitive transformations: first rotation, then shearing, then scaling, and finally translation. There is some justification for that order, as described above, but it is to an extent arbitrary. For example, I cannot think of any compelling argument for why rotation should come before shearing. I chose to write the example with this specific order because experiments with the autoencoder showed that it was the order that was most convenient for a neural network. However, with hindsight it's clearly also a reasonable choice.

Of course, we still want to avoid inverting anything: the inverse of a scaling transformation involves a division, and divisions aren't pleasant when one needs well-behaved gradients. The solution is to construct the (world → object) transformation matrix after all, but to do so using these insights about what makes for a learner-friendly representation. The (world → object) matrix is the inverse of the (object → world) matrix, so we must reverse the order of the elementary transformations: the (world → object) matrix will be constructed as the product of these four matrices in this order: a translation matrix; a scaling matrix; a shearing matrix; and a rotation matrix. With this decomposition, the fact that it's the (world → object) transformation instead of the more intuitive (object → world) transformation is not very troubling anymore: for example, the (world → object) rotation transformation in radians is simply minus the (object → world) rotation in radians, and having to change the sign of its output does not burden a neural network.

Learning logarithms vs. learning raw values

The scaling values are multipliers, and the most natural representation for multipliers is as their logarithm. It would make sense, therefore, to include in the pose not the scaling values, but the logs of those values.

I tried that and it didn't work well. One of the problems is that when the scaling value and its log get large, the gradient for the log gets large, too. Another problem occurs because of the practicalities of training this model using a GPU: when the network learns a bit too eagerly and then makes one big mistake on one unexpected data case, producing a large log scaling value, the scaling value itself can become too large for a 32-bit floating point number in a computer. If one doesn't write various complicating exception rules, this leads to *inf* and *nan* values in the model, which is unacceptable.

Therefore, I chose to include not the log of the scaling multiplier, but the scaling multiplier itself, in a capsule's pose. This means that it's more difficult to learn extremely small or extremely large scaling values, but it still worked in practice.

Choosing the origin

If you take an image, in a coordinate frame where the lower left corner is the origin, and you decide to rotate the whole thing counter-clockwise by 90 degrees, the image ends up not only rotated but also moved a lot. The same applies if you decide to scale or shear it. This is counter-intuitive, and therefore not ideal when a person has to design transformations. Experiments verified that it is similarly problematic when a neural network is designing them.

All of these problems get much smaller if the origin is in the middle of the image. This can be achieved as follows. Before applying the learned transformation, we apply a fixed transformation that moves the image so that the origin is in the centre. Then, we apply the learned transformation, which can now nicely assume that the origin is in the centre. Last, we apply another fixed transformation that moves the (now transformed) image back to more standard coordinates, i.e. a coordinate frame where the origin is in the same corner where it originally was.

This change, while not changing the number of learnable parameters or degrees of freedom in the transformation, means that the transformation is learned in an even more natural language: a language where rotation means rotation around the middle, instead of rotation around a corner, and likewise for scaling and shearing.

Summary: the learner-friendly approach

The "nice" (learner-friendly) description still has six degrees of freedom:

- Two, t_0 and t_1, for translation along the two axes.
- Two, s_0 and s_1, for scaling along the two axes.
- One, z, for shearing.
- One, θ, for rotation.

See Figure 2.4 for a diagram and some more details.

In the diagrams, I call this "nice" representation of a geometric transformation "geoNice", and if an intensity is specified, too, it's "iGeoNice". For notational convenience only (i.e. to reduce clutter in the diagrams), the "iGeoNice to iGeo(matrix)" component is introduced in Figure 2.5.

Overview: capsules with full affine transformations

Figure 2.6 shows multiple capsules in an autoencoder.

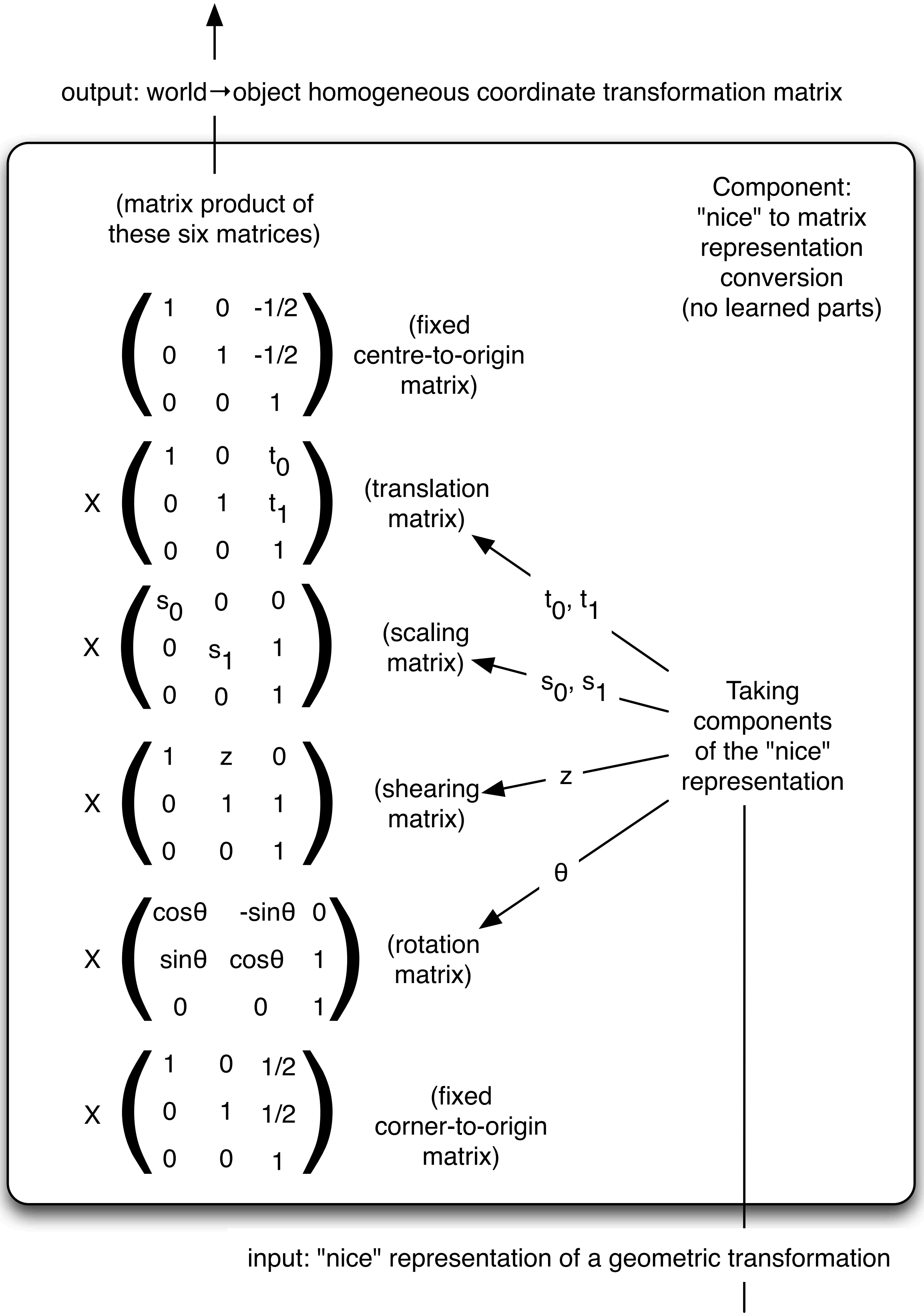
output: world→object homogeneous coordinate transformation matrix
(matrix product of these six matrices)
Component: "nice" to matrix representation conversion (no learned parts)
1 0 -1/2
0 1 -1/2
0 0 1
(fixed centre-to-origin matrix)
X
1 0 t0
0 1 t1
0 0 1
(translation matrix)
X
s0 0 0
0 s1 1
0 0 1
(scaling matrix)
X
1 z 0
0 1 1
0 0 1
(shearing matrix)
X
cosθ -sinθ 0
sinθ cosθ 1
0 0 1
(rotation matrix)
X
1 0 1/2
0 1 1/2
0 0 1
(fixed corner-to-origin matrix)
t0, t1
s0, s1
z
θ
Taking components of the "nice" representation
input: "nice" representation of a geometric transformation

Figure 2.4: Component: “nice” to matrix (NTM) representation conversion for geometric transformations. This component allows other components to describe geometric transformations in a natural or “nice” format. This component produces a matrix, using as input that natural representation. Error derivatives of the matrix are backpropagated through this component, and come out at the bottom as error derivatives for the 6 numbers in the natural representation.

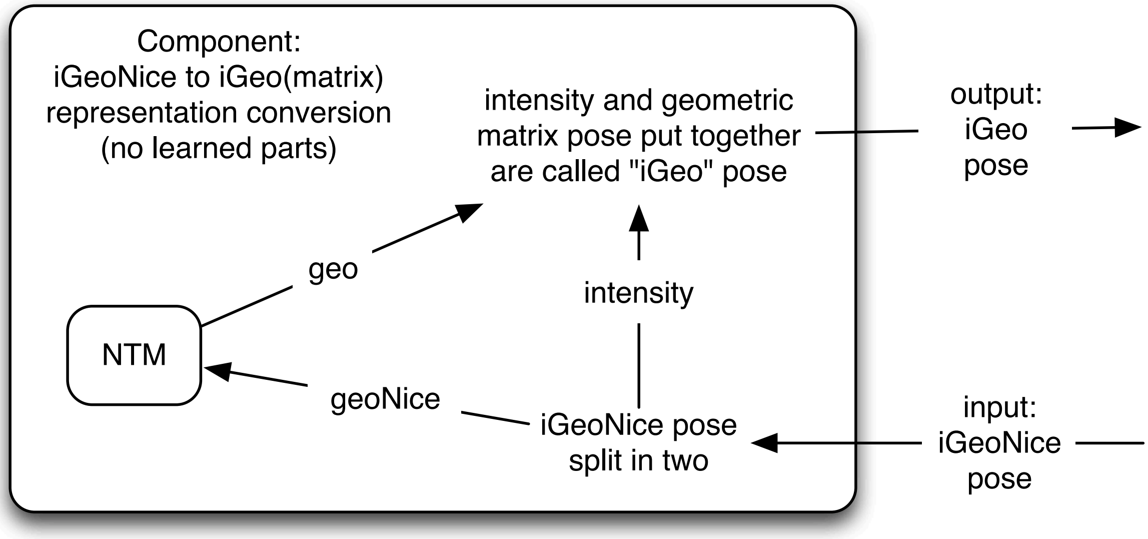

Figure 2.5: Component: iGeoNice to iGeo(matrix) (INTM) representation conversion for geometric transformations. “iGeoNice” means a specification of both an intensity scalar and the “nice” (i.e. learner-friendly) representation of a geometric transformation. In contrast, “iGeo” or means an intensity scalar and the matrix representation of a geometric transformation. Notice that this component doesn’t describe any new computations; its purpose is notational only. NTM: Figure 2.4.

2.3.3 Output models

After each capsule has computed its contribution to the output, the contributions from all capsules must be combined, somehow, and there are different ways to do this.

The most intuitive way is to simply add them up. For each output pixel, we ask what contribution to it is made by each of the capsules, and we add those up to get the final value of the pixel.

However, there are more sophisticated approaches, which in my experiments worked better. When we’re deciding what the value of a particular output pixel should be, we could use the largest of the capsules’ contributions, instead of the sum. This nonlinearity allows the model to sweep many things under the rug: only the dominant capsule will get to decide a pixel’s intensity. It is therefore more natural to speak of different capsules’ “suggestions” instead of their “contributions”.

The problem is that it makes many gradients go to zero: the learning signal disappears for all capsules except the one with the greatest output to a particular pixel. What works better is a softened version of the max function: $\text{maxSoft}(\vec{a}) = \log(\sum \exp(\vec{a}))$. However, that has a problem, too: if the capsules’ outputs are close to each other, then this function is so soft that it becomes linear again, like using the sum. What worked best for MNIST was a compromise: $\text{maxSemiSoft}(\vec{a}) = \frac{\log(\sum \exp(\vec{a} \cdot 100))}{100}$. This combines some of the differentiability of maxSoft with some of the selectiveness of the hard max function.

Thus, if a capsule doesn’t wish to draw on some part of the image, it should output zero there, so that the max will come from another capsule. If a capsule does wish to draw a part of the image, and other capsules draw there, too, then the result is that whichever one draws with more intensity determines the outcome. This is not entirely satisfactory, because it means that drawing dark details (i.e. intensity near 0) on a bright background (i.e. intensity near 1) is impossible. However, if capsules only draw non-zero output where they’re sure that they know the right intensity, then it works.

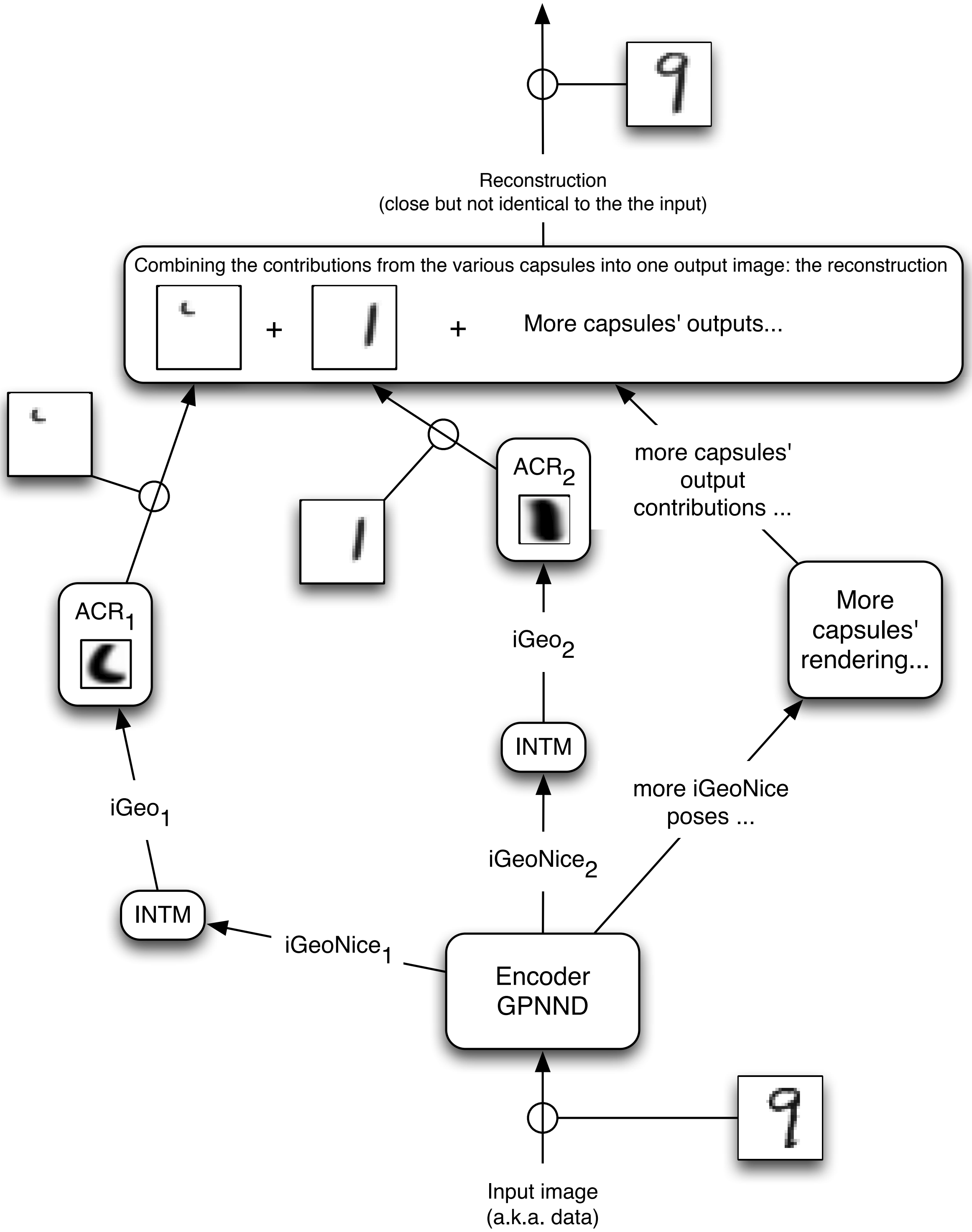
Reconstruction
(close but not identical to the the input)
Combining the contributions from the various capsules into one output image: the reconstruction
+
+
More capsules' outputs...
ACR$_2$
more capsules' output contributions ...
ACR$_1$
iGeo$_2$
More capsules' rendering...
INTM
more iGeoNice poses ...
iGeo$_1$
iGeoNice$_2$
INTM
iGeoNice$_1$
Encoder GPNND
Input image
(a.k.a. data)

Figure 2.6: A relatively simple autoencoder, where the decoder consists of multiple simple (atomic) capsules. Each capsule gets its pose from the partly shared encoder (which has shared hidden units), and the capsules' outputs are combined to make the model's final output. Each capsule has its own learned template. "iGeoNice" means a specification of both an intensity scalar and the "nice" (i.e. learner-friendly) representation of a geometric transformation. In contrast, "iGeo" means an intensity scalar and the matrix representation of a geometric transformation. GPNN: Figure 2.2. INTM: Figure 2.5. ACR: Figure 2.3.

One small modification can be made. When, say, 10 capsules all suggest a pixel intensity of zero, this maxSemiSoft function will not output zero, but rather $\frac{\log(\sum \exp(\vec{0} \cdot 100))}{100} \approx 0.023$. To make it easier for the system to really output zero, I subtract that value after computing maxSemiSoft, to produce the final pixel intensity.

Other ideas are $\sigma(\sum \vec{a})$ with $\sigma(x) = \frac{1}{1+\exp(-x)}$, $\max(\sum \vec{a}, 1)$, or a softened version of the latter, but none of these worked very well in my experiments.

2.3.4 Representing composite objects: composite capsules

The next step is the creation of capsules that represent composite objects, i.e. constellations of parts which can all be moved together. I call these composite capsules (CCs). The parts of these composites are represented by atomic capsules (ACs), which are what I described above.

The next few subsections describe the details of this set-up; for an overview diagram see Figure 2.7.

A new coordinate frame

As in computer graphics, the orientation ("pose") of the parts of a composite object will be internally described relative to the coordinate frame of the composite; not relative to the world coordinate frame. This makes sense: relative to the composite objects, the parts are always in exactly the same configuration (for rigid composite objects), or approximately the same configuration (for somewhat less rigid composite objects). When we need to know the orientation of a part relative to the world (e.g. when we're rendering the image), we can calculate it by multiplying the part-in-composite matrix and the composite-in-world matrix, but when we're describing the scene, it's easier to just talk about the parts relative to the composite, and about the composite relative to the world.

In the autoencoder, a composite capsule (CC) will have a "CC-in-world" pose that describes its orientation relative to the world. The parts of the composite will be atomic capsules (ACs), and their "AC-in-CC" pose describes their orientation relative to the CC.

The same two-stage set-up applies to the intensity scalar: the final intensity of an AC's appearance will be the product of the AC's nominal intensity (which is relative to its CC) and the CC's nominal intensity (which is relative to the world, i.e. it is absolute).

Thus, we now have two kinds of poses in the code layer of the autoencoder: CC-in-world poses, and AC-in-CC poses. Both types consist of a geometric transformation (a matrix) and an intensity transformation (1 multiplicative scalar), i.e. both are "iGeo" poses.

The pose of a composite (relative to the world) can be multiplied by the pose of a part (relative to the composite) to give us the pose of the part (relative to the world). That process is best described in a diagram: see Figure 2.8.

More CC pose: a CC's internal shape distortion

If a CC represented a rigid composite object, we would need nothing more than is described above. However, that would take away much of the purpose of having CCs: a rigid CC isn't more powerful than an AC. ACs are rigid by nature: they have a constant template. Having multiple rigid CCs that are forced to use (copies of) the same ACs would still make some sense because it forces extensive use of the templates of those ACs, but flexible CC's can be made, too, and are better.

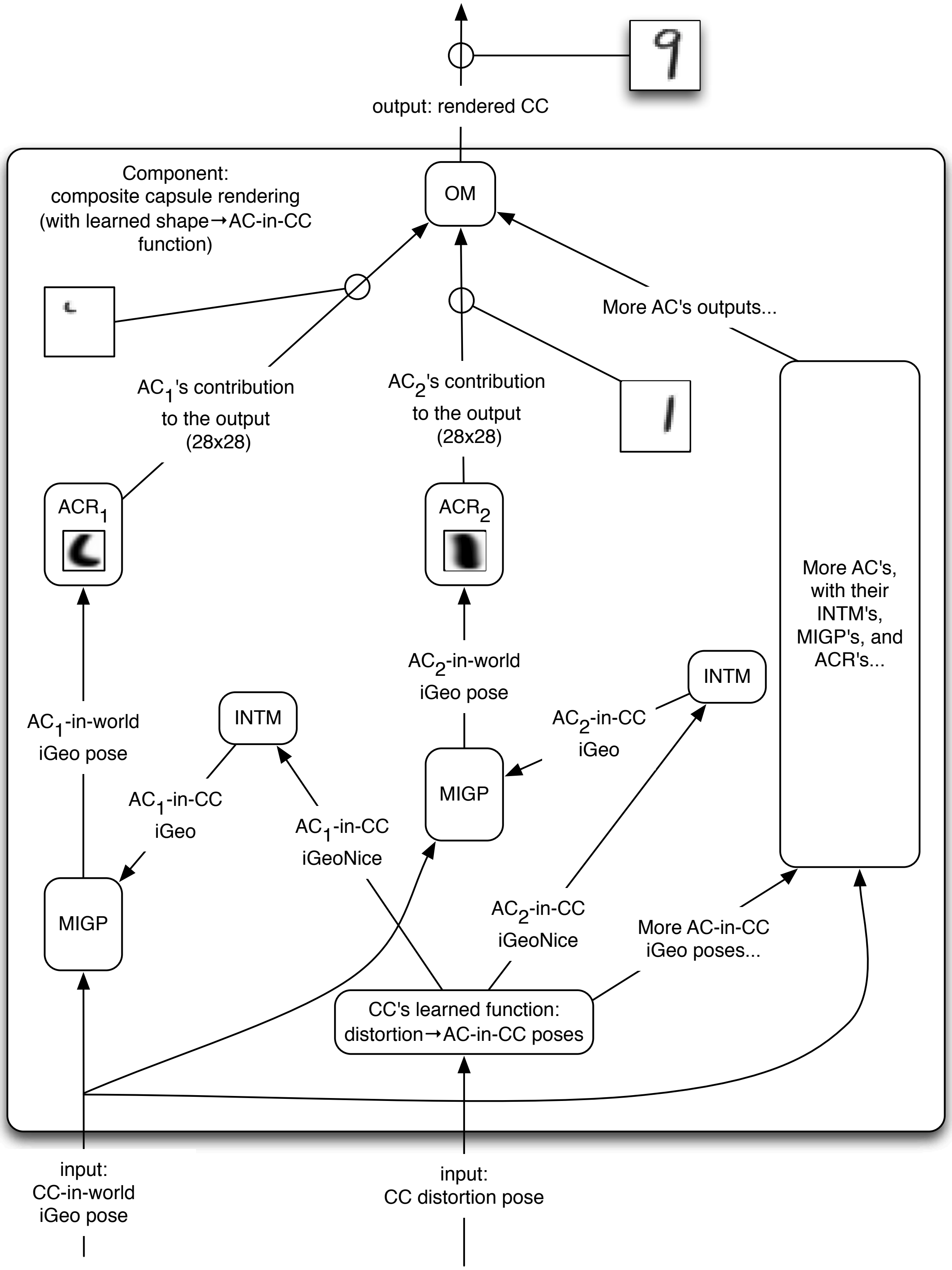

Figure 2.7: Component: composite capsule rendering (CCR). Only two of the ACs of this CC are shown, but there can be any number of them. INTM: Figure 2.5. MIGP: Figure 2.8. ACR: Figure 2.3. OM: Section 2.3.3.

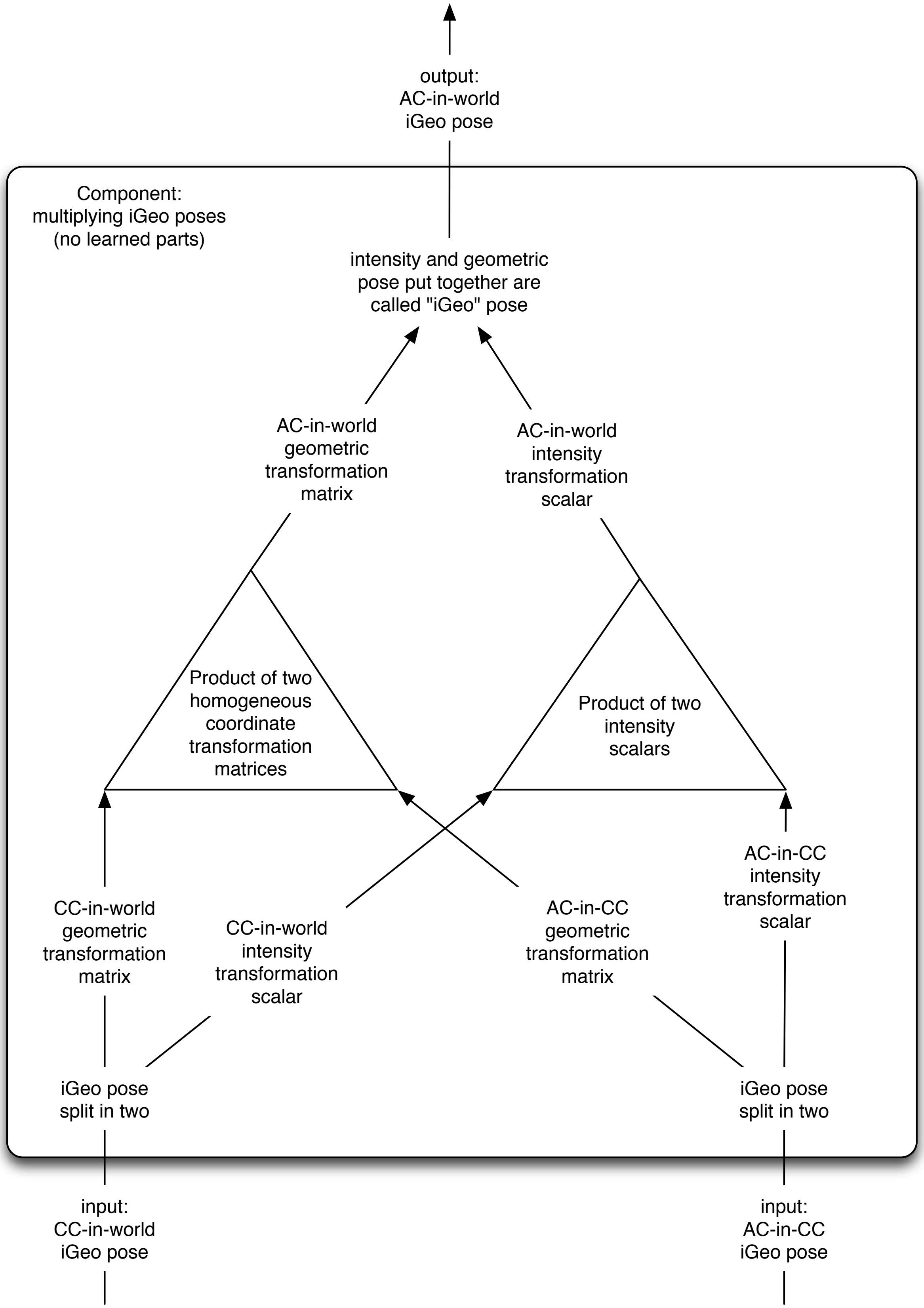

Figure 2.8: Component: Multiplying iGeo poses (MIGP). “iGeo pose” means a pose specification that includes both the intensity transformation multiplier and the geometric transformation matrix.

This can be done by allowing a CC to vary not only its geometric and intensity pose, but also to slightly vary its internal shape: the poses of its component ACs relative to the CC. One way of doing so is by defining a CC's pose to also include all of its AC-in-CC poses, but that leads to overly large CC poses, and there is a better way.

A CC's pose must, of course, include its geometric orientation and an intensity, but we'll add a third component: several "distortion" variables that describe how the AC-in-CC poses of the components of the CC vary from what's typical for this particular CC. A CC is, then, defined by the learned function that maps this "distortion" component of its pose to the AC-in-CC poses of its components. That function is the intrinsic nature of a CC (constant over cases, but learned), like a template is the intrinsic nature of an AC (constant over cases, but learned). In my experiments, that learned function is implemented as a DNN with the same architecture as the encoder (see Section 2.2), except that it doesn't use dropout (Section 1.8.4).

Because there are only few distortion variables, only a few kinds of changes can be made in a CC's appearance: the CC cannot be "distorted" into just about any shape. With the introduction of dropout on the distortion variables (explained in Section 2.4.4), a CC gets an identifiable default configuration of its ACs, and the values of the distortion variables describe only relatively minor deviations from that default (see Figure 2.22).

2.3.5 Multiple CCs

As described above, a CC's pose is decoded to an output image. An intermediate step in this process is the poses for the component ACs:

CC pose → AC poses → output image.

This is analogous to the decoding process of a regular autoencoder where the decoder has a hidden layer:

Code layer → decoder's hidden layer → output.

In such a regular autoencoder, we can choose to make the code layer larger without violating any underlying assumptions. Similarly, in the CC-based autoencoder, we can give a CC more *distortion* variables. Those more numerous distortion variables then cooperatively decide on the AC-in-CC poses.

What we cannot do, however, is create multiple CCs and ask them to share the ACs. The assumption is that ACs live next to each other in their own isolated world, and that world is a single CC. If there are to be multiple CCs, they should each have their own ACs. The structure must be such a hierarchy. However, it turns out that by carefully blurring some boundaries, slightly diverging from the hierarchical structure, we can make multiple CCs share ACs, on the condition that (roughly speaking) only one CC is active at the same time. A formal description follows.

If we must obey the hierarchical structure to the letter, we only have two options. We can decide that a single CC is enough (this is somewhat limiting), or we can decide that we need multiple CCs and that they'll all have their own ACs (this makes for a large total number of ACs, which is quite costly).

However, there's a compromise that allows multiple CCs without requiring much more computation time: the multiple CCs can form a mixture, much like a mixture of experts. It is crucial that, most of the time, the mixture manager makes a clear choice for just one of the CCs.

Of course, in a mixture each CC still needs to have its own ACs, but if only one CC will be activated by the mixture manager, then only that one CC's set of ACs will actually have an effect, so we have not increased the number of ACs that need to be dealt with (rendered, gradient backpropagated, etc). For datasets that are well described by a mixture, like MNIST, this is ideal.

A mixture manager has to make differentiable decisions, so during learning it doesn't really activate only one of the mixture components. It tries all components, but uses weights to do more learning in the "more responsible" components. This presents an obstacle to the proposed efficiency optimization.

The solution is to encourage the manager to express very strong preferences most of the time[12] : for a given data case, the manager should choose a single mixture component with high confidence. Then, in some places we'll ignore the fact that the manager still has a little bit of uncertainty about which component should handle the data case. This is the blurring of boundaries that I mentioned above.

Encouraging low entropy

A traditional mixture manager (Figure 2.9) has a softmax output. It works with values $\{x_i\}$ for each candidate component i, which are then converted to probabilities $\{y_i\}$ by the formula $y_i = \exp(x_i - \log(\sum_j \exp(x_j)))$. If the manager is confident about which mixture component should be used, then this $\{y_i\}$ distribution will have low entropy: it will be almost exactly one-of-N, i.e. one y_i value will be almost 1 and the others will be almost 0. However, things don't have to go this way: the manager is allowed to produce a high entropy $\{y_i\}$ distribution, by having the largest few x_i values close together.

The first step is to modify the manager by adding i.i.d. random variables $\{\xi_i\}$ to $\{x_i\}$ before the normalization step, like (Salakhutdinov & Hinton, 2009b) did to get nearly binary codes. This way, the probabilities are instead $y_i = \exp(x_i + \xi_i - \log(\sum_j \exp(x_j + \xi_j)))$ (Figure 2.10). These can still have quite some entropy, but the manager is now thoroughly discouraged from making the largest few x_i values be close together: if they are, then the $\{y_i\}$ distribution will be significantly influenced by the random $\{\xi_i\}$, effectively overriding some of the $\{x_i\}$, i.e. the manager's wishes. No manager likes to give up control like that, so the manager will eventually learn to produce low entropy choices by always making x_i for the chosen component i much larger than the other x values.

A ξ distribution that worked well in my experiments was the uniform distribution over the interval $[0, 10]$. On MNIST, with 25 mixture components, this led to the most favoured mixture component having a y_i value of 0.99 or more, for 80% of the data cases, and 0.9 or more for 90% of the data. Having 25 mixture components means that the greatest possible amount of entropy in $\{y_i\}$ is $\log 25$; typically the actual amount of entropy came out to only about 2.5% of that, because of this modified manager. It might have been even less if the system weren't also trying to keep all options open, to an extent (see Section 2.4.1 and Section 2.4.5).

For the second step, we must first take a look at what the traditional mixture of experts set-up with CCs would be (see Figure 2.11). There, each expert (each CC, in our case) gets to suggest an output, and the degree to which they're mistaken (the difference from the original, squared) is weighted by $\{y_i\}$ from the manager, to produce the network's error value for a particular data case.

Given that, by adding the random $\{\xi_i\}$, we have a manager that will typically make low-entropy choices, we can optimize by sweeping the last small bit of entropy under the rug, as follows.

There will be multiple CCs, all of them with their own pose $\vec{p_i}$ as suggested by the encoder, and all of them with their own y_i from the mixture manager. Typically, one of the $\{y_i\}$ will be nearly 1; the

[12] It would be bad to have such discrete choices *all the time*: that would eliminate the gradients that we need for learning. The system does not seem to fall into this trap, possibly because of a light weight decay (see Section 2.4.5). See below for some statistics on how strong the manager's preferences typically are.

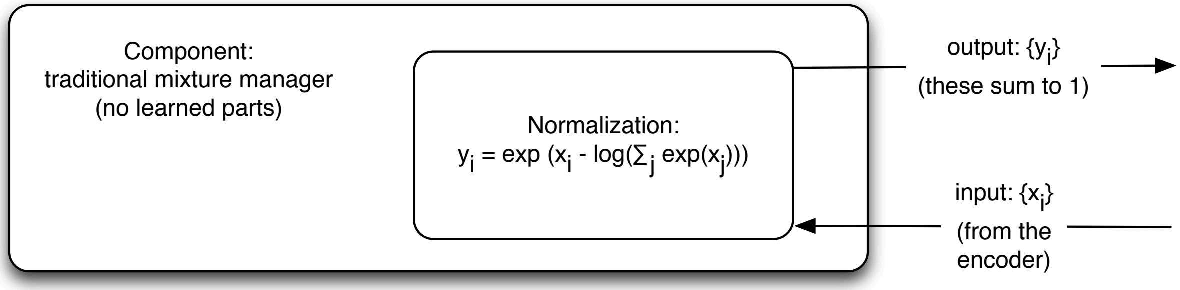

Figure 2.9: Component: traditional mixture manager (TMM). It's just a mathematical function.

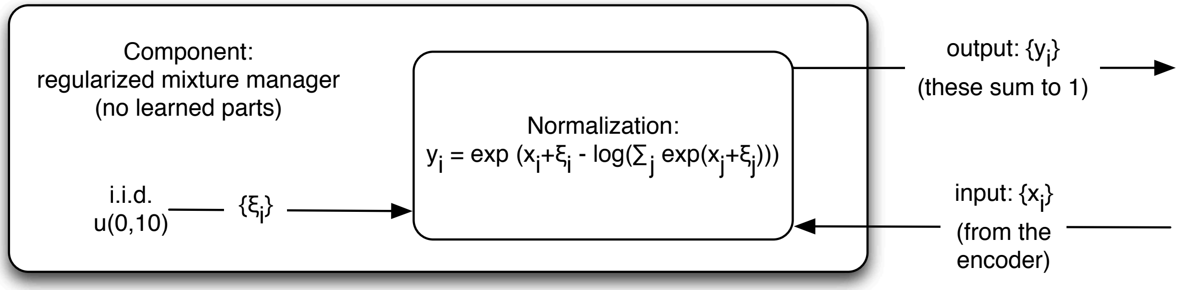

Figure 2.10: Component: regularized mixture manager (RMM). The sizable random influence encourages the model to make the $\{x_i\}$ represent very strong preferences.

others will be nearly 0. We can extract the pose of the selected CC in a differentiable way as follows: $\vec{p^*} = \sum_i y_i \cdot \vec{p_i}$. That concept of "the selected CC" is getting blurred here because it's no longer strictly just one of the CCs, but it still works in practice. This has some similarities to what happens in *mean field* variational inference in generative models: instead of treating the state of a unit as "probably (99% chance) a one, but maybe (1% chance) a 0", we treat it as 0.99.

Then, the chosen CC must apply its *distortion → AC-in-CC poses* function. For extra uniformity among the CCs, and to share some of the learning signal, the learned implementation of this function can be partially shared among the different CCs, as follows. The function is implemented in the decoder as a deterministic multilayer feedforward neural network that takes as input the *distortion* values (of course) and a one-of-N $\{y_i\}$ indication of which CC's function is to be applied. Thus, which CC's *distortion → AC-in-CC poses* function is computed is determined by which of the $\{y_i\}$ is 1. When in practice the $\{y_i\}$ are not exactly one-of-N but are almost one-of-N, we can provide the almost-one-of-N $\{y_i\}$ values instead of exactly-one-of-N $\{y_i\}$ values and it'll still work fine, and this way, everything can be differentiable. The *distortion → AC-in-CC poses* DNN is the same general-purpose neural network as the encoder (Figure 2.2), except that no dropout is applied.

This optimized mixture is shown in Figure 2.12.

When the $\{y_i\}$ have zero entropy, the two ways of implementing a mixture are equivalent, except that with this second set-up there's more shared learning in the *distortion → AC-in-CC poses* function. When the $\{y_i\}$ have little entropy, the two implementations are almost equivalent, because everything in the system is differentiable. When the $\{y_i\}$ have a significant amount of entropy, the semantics of this optimized implementation aren't very meaningful: the poses of multiple CCs are averaged, and the

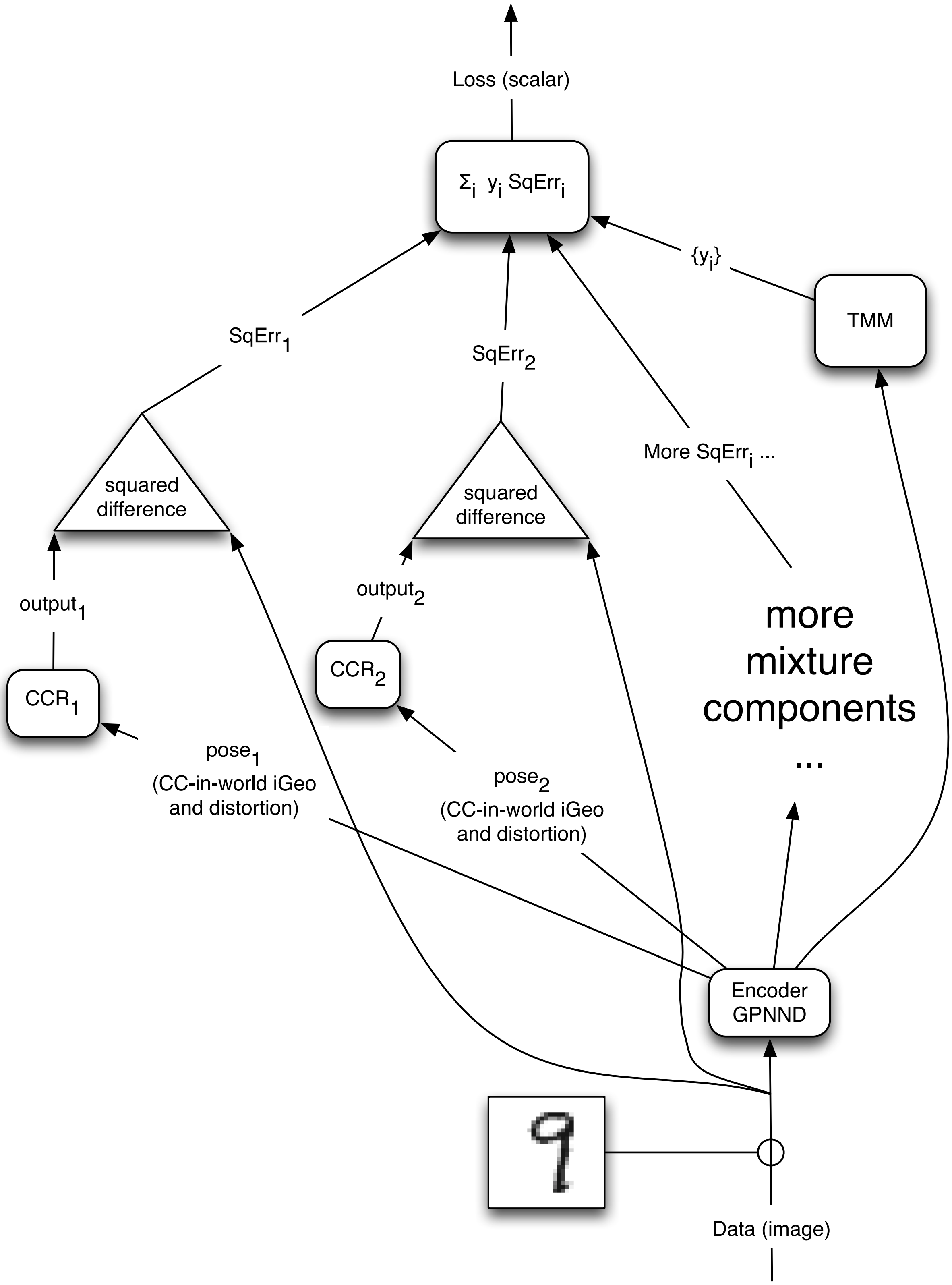

Figure 2.11: A mixture-of-CCs autoencoder, set up to mimic a conventional mixture of experts model. Only two CCs are shown for brevity. CCs have learned components, so each CCR is different. GPNN: Figure 2.2. CCR: Figure 2.7. TMM: Figure 2.9.

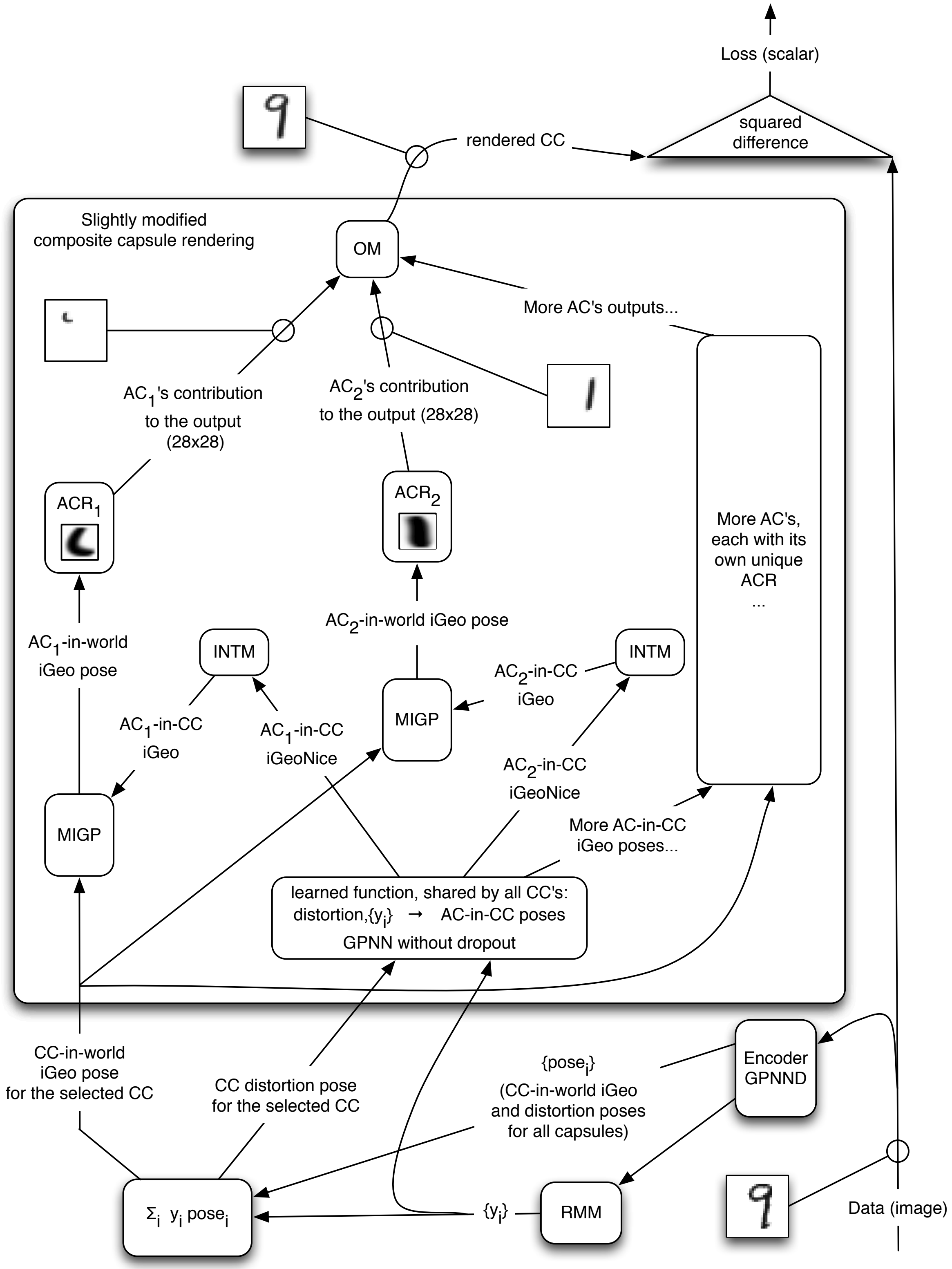
Loss (scalar)
squared difference
rendered CC
Slightly modified composite capsule rendering
OM
More AC's outputs...
AC$_1$'s contribution to the output (28x28)
AC$_2$'s contribution to the output (28x28)
ACR$_1$
ACR$_2$
More AC's, each with its own unique ACR ...
AC$_2$-in-world iGeo pose
AC$_1$-in-world iGeo pose
INTM
INTM
MIGP
MIGP
AC$_2$-in-CC iGeo
AC$_1$-in-CC iGeo
AC$_1$-in-CC iGeoNice
AC$_2$-in-CC iGeoNice
More AC-in-CC iGeo poses...
learned function, shared by all CC's: distortion,{y$_i$} → AC-in-CC poses
GPNN without dropout
CC-in-world iGeo pose for the selected CC
CC distortion pose for the selected CC
{pose$_i$} (CC-in-world iGeo and distortion poses for all capsules)
Encoder GPNND
Σ_i y$_i$ pose$_i$
{y$_i$}
RMM
Data (image)

Figure 2.12: The optimized mixture implementation. The section in the large box is almost the same as Figure 2.7: only the *distortion* → *AC-in-CC poses* function is different (see the bottom of the box). GPNN: Figure 2.2. RMM: Figure 2.10. INTM: Figure 2.5. MIGP: Figure 2.8. ACR: Figure 2.3. OM: Section 2.3.3.

implementations of the *distortion* → *AC-in-CC poses* functions for different CCs are blended. However, the manager has a strong incentive to learn nearly exactly one-of-N $\{y_i\}$, so this situation occurs rarely.

Of course, this optimization only really helps when it's costly to apply the *pose* → *output* function for lots of mixture components (lots of CCs, here). In our case, this is costly because rendering and backpropagating gradients through the rendering take up a significant chunk of the compute time, even with the optimization.

Lastly, note that this implementation of a mixture nicely emphasizes how a mixture is equivalent to a function with an additional categorical parameter.

Ambiguity in the mixture

Sometimes, a data case may come up that can be represented either by one CC with one geometric transformation, or by another CC with a different geometric transformation. Imagine, for example, that the image shows a somewhat rotated square. If there's a CC that's good at producing squares (maybe it can handle rectangles of various widths and heights as specified in its *distortion* variables), it can reproduce this image nicely. If there's another CC that's particularly good at producing diamond shapes (with, perhaps, variations in edge thickness as specified in its *distortion* variables), than that one can also reproduce the image, but it will need a different geometric transformation. The system can handle such ambiguity gracefully, because every CC is instantiated with its own geometric transformation. The flexibility would have been lost if there would be only one global geometric transformation, that would be applied to whichever CC is activated by the manager.

2.4 Regularization

2.4.1 Three reasons to regularize in general

There are different reasons why regularizing any kind of neural network often helps.

Avoiding overfitting

The best known reason is overfitting prevention: by keeping the learning capacity small, one can avoid situations where the training data is handled well by the trained network, but other data from the same distribution isn't.

Avoiding undesirable solutions

Another reason, somewhat less frequently cited, is that well-chosen regularization can sometimes prevent the network from learning to do the assigned task in ways that the designer considers undesirable, even if it also works on held-out data. An example of this type of regularization is making the code layer of an autoencoder into an information bottleneck[13]. Without that bottleneck, the autoencoder would be

[13] It is debatable whether an information bottleneck should be called *regularization*. Most things that we routinely call regularization are soft constraints that have nothing to do with the model architecture; an information bottleneck is clearly not in that category. On the other hand, information bottlenecks in the architecture have much in common with conventional regularization methods such as weight decay: their purpose is to limit the abilities of the model in some way, without significantly reducing other aspects of its abilities (the model can still have as many units as we have hardware capacity for). Weight decay effectively prevents the model from learning many large weights (which could lead to overfitting); an information bottleneck requires an efficient code (which similarly aids generalization). Thus, there is an argument for calling an information bottleneck *regularization*. I find this an appealing argument, but it is obviously a

able to reconstruct all data well, including held-out data, but even though that's the objective function, it's not what the designer really wants. Usually, the reason for training autoencoders is not some desire to be able to reconstruct things, but rather the wish to learn a useful representation of the data[14]. That discrepancy explains the need for regularization of this type.

Gradient maintenance

A still less commonly emphasized reason for regularizing is the desire to keep gradients from going to zero prematurely. It arises because gradient-based learning is always trying to reduce reconstruction error *in the short term*, which can lead to decisions that are harmful in the long run.

Imagine, for example, that the encoder of a mixture-of-CCs autoencoder has learned never to use mixture component #13, possibly because its random initialization was so poor that the best way forward in the short term was to simply disable it entirely. Now that component #13 is never used any more, its template has no effect on the model's performance and therefore has zero gradients and will never change. It remains underdeveloped, and the mixture manager will never want to use component #13. Obviously, this is an undesirable situation. It could be compared to a kid in high school who never learns to play hockey because nobody ever wants him on the team, so he gets zero practice and indeed never develops hockey skills.

It would be much better if the mixture manager would, every once in a while, give #13 a bit of a second chance. Whenever that happens, its template would get some learning signal, and eventually it would become useful, and the mixture manager would eventually offer #13 a permanent job. In other words, the manager should keep exploring alternatives, and should avoid writing off possibilities too harshly, unless that is really necessary.

The manager can be gently encouraged to give such second chances. Other components of the system are susceptible to the same type of vicious cycles and also need to be encouraged to give second chances. That way, every component is always at least a little bit involved in the action, and we don't have these prematurely disappearing gradients.

2.4.2 Regularizing qualities of the capsules concept

Because of the nature of this domain-specific decoder, the autoencoder is not very susceptible to the first two problems that regularization usually aims to curb: overfitting and undesirable but well-reconstructing solutions.

Small code

One of the bottlenecks that prevent these problems is the size of the code layer. The code layer of the mixture of CCs in Figure 2.12 contains a categorical variable indicating which mixture component to use (out of e.g. 25 components), an intensity and a geometric transformation of the component (which have a fixed interpretation in the decoder and therefore automatically generalize), and some distortion variables (anywhere for 5 to 30, in most of my experiments). The distortion variables can contain quite some information, and their interpretation by the decoder is not fixed, so they could be involved in overfitting. However, there are not many of them. There is no need to have many, because much of

subjective choice.

[14] We could use an autoencoder for the purpose of reconstructing: if we give it a small code layer, and it still learns to reconstruct well, we have a lossy compression algorithm.

the information about the input image can be encoded in the intensity and geometric transformation, and in the mixture component indicator variable: the CCs learn good models of different types of input images, and all it takes to activate the right CC is that one indicator variable.

Largely fixed decoder

Consider the decoder of the AC-based autoencoder of Figure 2.6. If we have 10 ACs, each of which has a template of 10x10 pixels, then those 1,000 pixel intensities are the only learnable parameters of the decoder. If we have at least a few thousand images to train on (many modern datasets contain millions of images), those 1,000 parameters will be rather tied down by having to participate in reconstructing a few thousand images. They will not have much flexibility left for memorizing individual training cases, i.e. for overfitting.

The decoder of the CC-based autoencoder (Figure 2.12) does contain many parameters, namely in the *distortion,*$\{y_i\} \rightarrow$ *AC-in-CC poses* function. However, that CC-based autoencoder can also be seen as an AC-based autoencoder, where the encoder is bigger and contains the *distortion,*$\{y_i\} \rightarrow$ *AC-in-CC poses* function, and the decoder has again only something like 1,000 learnable parameters. Therefore, the above argument still applies.

Componential structure

The above effects on their own would still leave the possibility of the encoder failing to produce good descriptions of images on which it was not trained. However, the code language is also quite componential: one AC's pose does not affect another AC's output. Therefore, the encoder is likely to learn a similarly componential approach, which helps prevent overfitting. Every component of the encoder (say every AC's pose encoder) has to work well for every training case, and such an encoder component is smaller than the entire encoder and is therefore more efficiently restricted by the training data.

2.4.3 Additional regularization

The capsules-based autoencoder does need regularization, but not for the usual reasons.

The model needs a large number of learnable parameters, because computer vision is not easy, and the encoder is doing computer vision. However, overfitting is not much of an issue, because of the model architecture, as explained above.

Avoiding undesirable solutions is a bit of an issue, and inspired some of the regularization strategies that I use. These are helpful but not essential.

The main reason for having regularization is the least well-known of the three: maintaining non-zero gradients. The system has many components, all interacting in learnable and sometimes complicated ways. Because of that, there are several ways in which a component could be disabled prematurely. One example is discarding mixture component #13 as I mentioned before. Another example would be deciding to add translation to a capsule's pose that ensures that it is never inside the output image any more. This would immediately eliminate all gradients related to that capsule.

Because of these dangers, regularization is needed, mostly to keep the many pieces working together in an at least mildly reasonable way, so that better ways can be learned by gradient-based optimization. That can be done by weight decay (most pathological behaviour would involve some large learned values somewhere in the system), but also by more problem-specific regularization methods.

Section 2.4.4 lists the various regularization strategies that I found helpful for preventing undesirable solutions. Section 2.4.5 lists regularization strategies that are intended for gradient maintenance.

2.4.4 Avoiding undesirable solutions in the capsules-based autoencoder

Sometimes, autoencoder training can lead to solutions (learned functions) that are decent or even good at reconstructing the input, but don't really do what the researcher was hoping for. This is because the objective function, "reconstruct the input well", is not what a researcher typically cares about. Because of this, autoencoder regularization is an active area of research; see for example (Rifai et al., 2011).

Dropout on the distortion variables

Dropout (see Section 1.8.4) on the hidden units in a neural network tends to make the units operate more independently, and thus avoids overfitting much like the componentiality of capsules can. Dropout can be applied to the distortion variables of a CC's pose, and have the same effect: the individual distortion variables learn to operate more independently of each other. This makes a big difference.

As a second effect, it prevents a CC from generating very different output for different data cases. The CC will want to always produce the same output, or, failing that, not overly different outputs for different data cases. To understand this, imagine that a CC produces very different outputs for different data cases. If, for some data case, it is to produce an output very different from what it typically produces, it will be told to do so by its distortion variables. If those variables are subject to dropout (i.e. to being set to zero at times), then there's a significant probability that the CC will not "get the message" (because the message dropped out), and instead of that very unusual output, it will produce a much more typical output, thus incurring a large reconstruction error. Therefore, when the distortion variables in a CC's pose are subject to dropout, the CC will prefer that each distortion variable describes only a small variation on the CC's typical output.

For the same reasons, the distortion values will have a mean value close to zero, relative to their standard deviation[15]: if the distortion values are approximately zero, dropout cannot cause much damage. Thus, a CC gets a clear default output shape, namely the shape that it produces when all distortion values are zero. This is shown in Section 2.6.4 (especially Figure 2.18).

In a mixture of CCs, each mixture component covers some manifold in the data space, and all mixture components together are to cover the most densely populated part of the data space. If it's a mixture of CCs whose distortion variables are subject to dropout, there's a pressure to make each of those manifolds quite local, as opposed to stretching from one end of the data space to another. This makes for very identifiable clusters (details are in Section 2.6.6), taking MNIST classification error rate down from 12% to 1.7%.

Disallowing negative intensities and "negative ink"

Another undesirable solution to be avoided is one that gets overly creative with the use of "negative ink". Consider the case of a capsules-based autoencoder where the individual ACs' outputs are simply added up to produce the model's output. The researcher may prefer that a capsule's influence on the output should be non-negative, i.e. that it should only be adding things to the output, and never subtracting

[15] In typical runs, the mean value for the distortion values was found to be ranging from -0.2 to +0.2 standard deviations away from zero, i.e. quite close to zero indeed. Without dropout on the distortion values, in one run the mean distortion values ranged from -1.1 to +1.5 standard deviations away from zero.

things. In short, perhaps the capsules should not produce “negative ink”, even if that would, in the short term, lead to slightly better reconstructions. In the long term, the researcher may think that such “negative ink” is just not “natural” (whatever that word means exactly), and will therefore not lead to good solutions.

Non-negative output from ACs can be enforced by requiring that both the templates and the intensities (which are part of the poses) be non-negative. This could be done in a few different ways:

- By adding weight decay but only on negative template pixels, and by similarly penalizing negative intensities. This implements the non-negativity wish in a soft way, and requires one to choose the weight decay strength.

- For the template pixels (which are simply learned parameters), it could also be done by simply refusing to make them negative, even if they’re zero and they have a negative gradient. This implements a hard constraint. However, this cannot easily be done for the intensities, because those are not learned parameters but functions of learned parameters and the input data.

- Another way to implement a hard constraint is to learn not template pixels, but f^{-1} of template pixels for some f. Those f^{-1}(template) values are then transformed by $f(\cdot)$ to produce the actual template pixels. This not only applies to learned template pixels, but also to dynamically generated intensity values. If $f(\cdot)$ is a non-negative function, this will have the desired effect. One such function is $f(x) = \max(x, 0)$, but that has the drawback that its gradient is zero when the input goes below zero. Another idea would be $f(x) = \log(1 + \exp(x))$, or $f(x) = \frac{1}{10}\log(1 + \exp(x \cdot 10))$ if we want to make it easier to produce near-zero outputs. That last function is what I used whenever I felt that a capsule’s contribution should always be non-negative.

For classification of the MNIST digits database, disallowing negative ink halved the error rate (from 3.3% to 1.7%).

2.4.5 Gradient maintenance in the capsules-based autoencoder

Gradient maintenance by weight decay

The most commonly used form of regularization in neural networks is L2 weight decay, and I decided to use it here, too, albeit with one small change. Usually, weight decay is implemented as an additional loss of $\alpha \sum_i \theta_i^2$, where α is a meta-parameter that specifies how strong the weight decay is, and θ is the vector of all learnable parameters.

Instead of that, I used $\alpha \sum_i \max(0, |\theta_i| - 0.1)^2$. This means that only the part that is more than 0.1 away from zero is penalized. I did this because I consider values that are 0.1 (or less) to be no less acceptable than values that are exactly zero. In fact, a value of 0.1 has an advantage over a value of zero, when it comes to gradient maintenance: a weight of zero means that a connection is entirely disabled, and this makes zero gradients more likely. However, I did not thoroughly investigate if this change is important or not. It doesn’t seem likely to be harmful, and with the computer automatically computing the derivative of the loss function, it wasn’t much of an added burden.

Such weight decay can, in theory, prevent all of the problems that arise from the model having learned some unnecessarily extreme values, like the thought experiment with the unfortunate mixture component #13. Weight decay would (indirectly) tell the mixture manager to only express such an extremely negative opinion of #13 if even a minutely less negative one would already make for significantly worse performance, which typically won't be the case.

Because the main purpose of the weight decay was not capacity control but just "gradient maintenance", it didn't have to be strong. I used $\alpha = 10^{-6}$, which is not much[16].

A similar decay rule could also very reasonably be applied to not the weights but the states (or inputs) of all hidden units in a network[17], but I didn't experiment with that. I did, however, apply a bit of such decay to the values in the code layer of the autoencoder.

Gradient maintenance by preventing excessive scaling

One specific cause of disappearing gradients is when an AC's template is always scaled up so much that only a few pixels of it are visible in the output. When that happens, the other pixels get zero gradients. Another problem that may arise is when for different cases a different part of the template is in the output: again, the effect is that typically only a few pixels of the template are used.

It's a bit tricky to express "the number of template pixels that are not visible in the output because of scaling" as a simple mathematical function, but there's a surrogate that works well enough in practice. When the geometric transformation (template → world) has a large determinant, the template is being scaled up a lot. We can express a penalty in terms of the determinant. It won't correspond exactly to "don't scale too much of the template outside the output", but it'll be better than nothing. In my experiments, it solved the problem.

For the implementation details, consider an autoencoder like Figure 2.6 or Figure 2.12. The "geo" part of the "iGeo" pose that goes into the ACR components represents the (world → template) coordinate transformation. We would like to say that the determinant of the (template → world) transformation should not be too large, but we can instead say that the determinant of the (world → template) transformation should not be too small. The penalty term is $\alpha \cdot \max(0, s - \det(\text{world} \rightarrow \text{template}))^2$, where α is the tunable strength of the penalty and s is the smallest determinant that we're entirely happy with. I set s to the value that would make the template exactly as large as the entire output.

Gradient maintenance by nondeterministic mixture component selection

The mixture manager presented in Section 2.3.5 gets random variables added to its preferences, making the actual decisions a bit unpredictable. The main goal here is encouraging low entropy to enable the optimization outlined in that section.

The noise means that sometimes, a mixture component will be activated that the manager didn't really intend to activate. Despite the fact that it wasn't intentional, the suddenly activated mixture component still has to reconstruct the input as accurately as possible. In other words, it still receives a training signal. Thus, when the mixture manager hasn't quite made up its mind yet about which of two mixture components should be handling a particular input, on average both will get a learning signal. Therefore, also the perhaps slightly less favoured component gets a chance to learn, and if it learns very

[16] Without weight decay, the MNIST classification error rate (see Section 2.6.6) went from 1.7% to 2.0%.

[17] It wouldn't be "weight decay", but it might be called "intermediate value penalization". In mathematical notation we can write it as ϕ_{ivp}, i.e. the component of the objective function ϕ that comes from intermediate value penalization: $\phi_{\text{ivp}}(\theta) = \alpha \underset{x \sim D}{\mathrm{E}}[\|h(x, \theta)\|_2^2]$, where α is the regularization coefficient, D is the training data set, θ is the collection of learnable model parameters, and $h(x, \theta)$ is the function that computes the unit values for training case x with model parameters θ. I consider it to be related to weight decay because the formula for that is quite similar: $\phi_{\text{wd}}(\theta) = \alpha\|\theta\|_2^2$

well, the mixture manager might start to favour it more after all. Thus, this noisy manager also has some "gradient maintenance" effect.

Also, the mixture flexibility means that we can, after training, artificially select a particular mixture component to see how it would reconstruct the input (see Section 2.6.5).

Gradient maintenance by encouraging the use of all mixture components

In Section 2.4.1, I mentioned the undesirable scenario of a certain mixture component (let's call it component #13) never getting activated by the mixture manager. Such non-inclusive behaviour on the part of the manager can be explicitly discouraged by adding another bit of regularization to the objective function.

Training typically proceeds with *mini-batches*, especially given the current trend towards parallelization of compute power. Let's say that there are 500 data cases in a mini-batch. If there are 25 mixture components, it would be reasonable to suppose that all 25 of them are used for at least one of the 500 data cases. If all 25 components on average get the same work load, then the probability of one component having no work at all in a batch is about $3 \cdot 10^{-8}$, i.e. quite small. However, if the manager does not distribute work evenly, there might be a mixture component that gets no work at all in an entire batch. The idea is to punish the manager whenever such a thing happens.

We take the $\{y_i\}$ (these are 25 values per data case if we have 25 mixture components), and average them over the 500 training cases of the batch. Now we have 25 values that describe the share of the batch that was assigned to each mixture component. Of those 25 numbers we take the logarithm; we add up those logarithms; we multiply that by some α that indicates how urgent this directive to the manager is; and the result of all that is added to the objective function, to be maximized[18].

The same can be said with a formula: add $\alpha \sum_i \log(\frac{1}{N} \sum_j y_i^j)$, where N is the number of data cases in the batch, and y_i^j is the responsibility that the mixture manager assigns to mixture component i on data case j.

This way, if the manager does avoid the use of, say, mixture component #13, this component of the objective function will have a very low (i.e. unfavourable) value, and there will be a gradient pushing towards using mixture component #13 more.

The scheme is a little vulnerable to very bad luck: if a batch shows up that, by coincidence, has no cases for a particular mixture component, there will be a severe penalty. It is, however, not an extreme penalty, and therefore also not an extreme gradient. The system can be made more robust by using a historic average of the $\{y_i\}$, instead of the batch average. However, I am not under the impression that using the batch average caused any problems, and it does eliminate the need for tracking history and incorporating it into the automated gradient computation.

2.4.6 Regularizing the encoder

All of the above regularization strategies deal with the code layer, and together they are quite sufficient (many are not even essential). However, performance can still be improved somewhat by also regularizing the encoder, using dropout.

The encoder is a large neural network with a difficult job to do: computer vision. The language of the descriptions that it needs to find is nearly fixed (unlike in a regular autoencoder), and the componentiality

[18] If you're working with a numerical optimizer that insists on minimizing instead of maximizing, just add a minus sign.

of that language further reduces the danger of overfitting, but some overfitting can still happen. Dropout works well to combat that. Dropout is a very simple and reliable method, and in all of my experience with it, I have never come across a situation where a dropout rate of 50% was far from optimal. That value is what I used in the encoder.

Again, this is far from essential, but it does help a bit.

2.5 Other implementation notes

Before I describe a few more details of my implementation, I must emphasize that this is only one implementation of the idea, and it could be done quite differently, and undoubtedly better. The main idea is to include domain-specific knowledge in a sophisticated componentially designed data generation system, so that a data recognition system can then learn from it. That can be done in many ways. That being said, here's a bit more about how I did it.

2.5.1 Computer-assisted model handling

Because the model is more complex than a typical neural network, bigger software tools are needed than just a simple script or two. I created a system that allows one to describe a model as a mathematical function: from the input(s), we calculate intermediate values; from those, we create further intermediate values; and in the end, we compute the output value. In this case, the input is an image and a vector θ that contains all learnable parameters, and the output is the objective function value. This notation allows for automatic differentiation, for computer-assisted visualization, and for componentiality by defining helper functions which are then invoked from the main function (or from other helpers).

Having such a tool makes the development effort much easier, especially when one needs to try lots of different variations on a model. It is quite a natural and flexible tool, and I was certainly not the first to think of it. A more developed system for the same purpose is Theano (Bergstra et al., 2010), and the only reason why I did not use Theano is that it wasn't yet very mature when I started this research project. However, it is obviously a wheel that should not be re-invented by every researcher.

2.5.2 Using GPU's

Machine learning programs usually require large amounts of compute power, and nowadays Graphics Processing Units (GPUs) are best at providing that. I used GPUs through the Cudamat+Gnumpy interface for Python (Mnih, 2009; Tieleman, 2010).

Most computations are standard (matrix multiplications, simple elementwise operations) and are available in any GPU API. However, somewhat ironically given what "GPU" stands for, I spent a lot of time implementing AC rendering and its derivatives on GPUs, in the CUDA programming language. These computations take up the biggest chunk of compute time, and are somewhat tricky to implement on a GPU, given a GPU's aversion to conditional statements and difficulty with enabling large amounts of efficient randomly accessible memory.

2.5.3 The numerical optimizer

The problem

A simple neural network with one hidden layer has four groups of learnable parameters: input → hidden connection strengths, hidden → output connection strengths, hidden biases, and output biases.

The presented mixture of CCs (Figure 2.12), on the other hand, can reasonably be said to have 62 groups of learnable parameters, such as:

- The ACs' templates.
- The encoder's skip-connection weights between the second hidden layer and the geometric CC-in-world translation outputs.
- The *distortion* → *AC-in-CC poses* function's connection weights between the third hidden layer and the AC-in-CC intensity values.
- The encoder's skip-connection weights between the first hidden layer and the mixture manager's $\{x_i\}$.

It's debatable whether the biases to geometric translation values (in pixels) should be counted as separate from the biases to geometric rotation values (in radians), but whatever counting method you consider to be reasonable, the number will far exceed four.

These different groups of parameters have different types of effects and are expressed in different units, and therefore have different gradient characteristics: different gradient sizes, different internal second-order characteristics, different interactions with the other 61 groups, and different sensitivities to the various sources of randomness in the model.

Because of all those differences, I didn't choose the commonly used algorithm of simple gradient descent with a single fixed learning rate. Something more sophisticated is needed.

The main idea: conservative learning

Philosophically more satisfying would be to have 62 different learning rates, but in practice it's easier to have individually adapting learning rates. There are many ways to implement adaptive learning rates, and most of those ways have meta-parameters that need to be set by hand. Setting 62 sets of adaptivity parameters is no good either, so I designed an adaptive learning rate algorithm with emphasis on being cautious: if in doubt, *any doubt at all*, then use a small learning rate. This means that it won't be optimal for any of the 62 groups of parameters, but it won't cause any disasters either. Especially in research situations, it's better to have to run a program five times as long because of sub-optimality than to spend very much more time writing the program, in search of 62 optimal strategies. The philosophy of this approach is similar, in spirit, to the way computer science theorists design "optimal" algorithms that are guaranteed to achieve their goal in the shortest possible amount of time, up to a constant multiplicative factor which could be quite large but is considered to be unimportant.

The optimizer that I designed for this work is not, in my opinion, a core component. Other good optimizers might do the trick just as well, or better. I just made something relatively simple but very robust, not aiming for (or caring about) stellar performance. All of this work is a proof-of-concept anyway, so optimality is not a major concern. Because of that, I have not thoroughly compared variations of the optimizer, nor have I compared its performance to that of other optimizers. The main point of this work is in the model design, not in the optimizer. I do, however, include a brief description of the optimizer here, for completeness.

Four suggestions for $\Delta\theta_i$

The optimizer deals with each learnable parameter individually: it makes no attempt to understand the interactions between different parameters i.e. the off-diagonal terms of the Hessian. When deciding by what amount to change a learnable parameter, it comes up with four different suggestions, each based on a different type of analysis, and then goes with the most conservative of the four, i.e. the smallest of the four proposed changes in the parameter value. The hope is that such conservatism, while probably not good if we need very fast learning, will at least avoid disasters, for all 62 different types of parameters. The four proposals are the following, with $g_i = \frac{\partial \phi}{\partial \theta_i}$ standing for the mini-batch objective function gradient for the parameter under consideration:

- The first rule, i.e. the first suggested value for $\Delta\theta_i$, is simply a maximum on the (absolute) change in parameter value: $\Delta\theta_i = \alpha_1 \cdot \text{sign}(g_i)$. This rule exists only as a last resort measure to prevent extreme changes in parameter value, and is almost never the dominant one.
- The second suggestion for $\Delta\theta_i$ is a maximum learning rate: $\Delta\theta_i = \alpha_2 \cdot g_i$.
- The third rule aims to limit the typical size of the updates: $\Delta\theta_i = \alpha_3 \cdot \frac{g_i}{\sqrt{\widehat{g_i^2}}}$, where $\widehat{g_i^2}$ is an estimate of the mean of g_i^2 over mini-batches, and is implemented as a historical decaying average of that squared gradient (see more details below). In rmsprop (Hinton, 2012), this is the only rule.
- The fourth rule is in place to recognize that if a gradient is larger, that might in fact be reason to change the parameter value less rapidly. It can also be explained as aiming to limit the amount by which we're attempting to change ϕ on a single update. $\Delta\theta_i = \alpha_4 \cdot \frac{g_i}{\widehat{g_i^2}}$, with the same $\widehat{g_i^2}$ as in the third rule. If, by a rescaling of a parameter[19], the effect of small changes in the value of that parameter is made ten times greater, then intuitively, the reasonable thing to do is to make ten times smaller changes. This fourth rule is in place to respect that insight.

All of these rules have their strengths and weaknesses, but the hope is that by always taking whichever of the four makes the most conservative suggestion, we'll always be reasonably safe. It seems to have worked well enough in practice.

The algorithm has four α_i meta-parameters, which I explored manually to find good values[20].

Conservatively tracking historical gradient size

The algorithm also has meta-parameters in the choice of how to compute $\widehat{g_i}$. The typical way to compute this is with an iteration that makes an update every time g_i is computed: $\widehat{g_i}_{\text{new}} \leftarrow \frac{1}{\gamma} \cdot g_i^2 + (1 - \frac{1}{\gamma}) \cdot \widehat{g_i}_{\text{old}}$. Here, γ can be thought of as the amount of time until a memory has significantly decayed. A large γ has the advantage that large gradient values (which signal a need for proceeding with caution) are remembered longer; a small γ has the advantage that large gradient values have an almost immediate

[19] Here, a "rescaling" means that the model would use $\theta_i \cdot 10$ where it previously used simply θ_i. This doesn't fundamentally change what the model is doing, but it does make the effect of an infinitesimal change in θ_i ten times greater.

[20] Recently, effective systems have been developed that automate the task of finding good values for meta-parameters (Snoek et al., 2012; Swersky et al., 2013; Bergstra et al., 2011; Snoek et al., 2013). In my opinion, they are now sufficiently mature, and I recommend their use for situations like these.

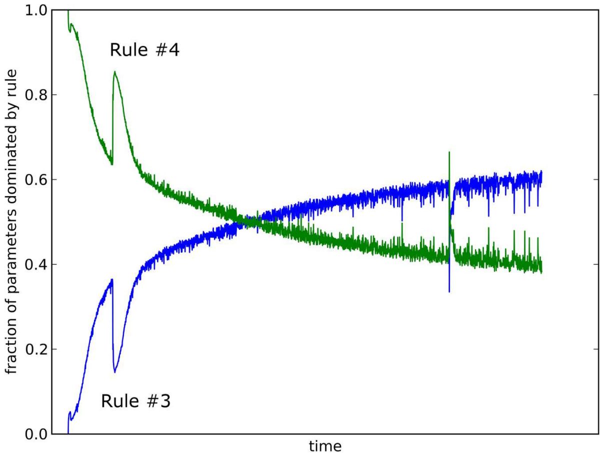

Figure 2.13: The change, over time, in the fraction of learned parameters of which the update size is determined by rule #3, and by rule #4. The other two rules didn't dominate for a significant fraction of the parameters.

effect when they occur. I ended up using three decaying averages $\widehat{g_i}$, each with a different γ: one with $\gamma = \exp(0) = 1$, one with $\gamma = \exp(3)$, and one with $\gamma = \exp(6)$. The actual $\widehat{g_i}$ used by the above rules three and four is then whichever of these three moving averages has the most alarming assessment of gradient size, i.e. it's whichever of these three gives the largest number.

As again becomes clear here, this optimizer is full of heuristics. I have not found time to study it thoroughly, but it seems to have done its job acceptably.

The two dominant rules

After finding good values for $\{\alpha_i\}$, one can record which of these four rules tend to dominate most often, i.e. which rules tend to make the most conservative suggestions. In practice, the third and fourth rule tend to dominate. At first, the fourth rule dominates for all parameters (this is just an artefact of initialization), and then, slowly, the third rule becomes dominant for more parameters. A plot of this process, for a typical run, is shown in Figure 2.13. It shows how initially, rule #4 dominates, but that's because the initial gradient size estimates are high (another bit of conservatism), and rule #4 is the most sensitive to large gradients. Then, over time, rule #3, which aims for same-size changes in the parameter value, gets to dominate more and more. Sometimes rule #4 dominates a bit more again, when there are more large (or noisy) gradients. When those subside, rule #3 takes over again. Rule #1 is almost never used: it is only an emergency break. Rule #2, i.e. the simplest form of gradient-based optimization, turns out to be simply useless and is never used, with the $\{\alpha_i\}$ values that turned out to work best. Over time, rule #4 dominates fewer and fewer parameter updates, but even at the end of training, rule #4 still dominates for 40% of the parameters.

2.6 Experiments on MNIST

Experiments on MNIST unambiguously showed the potential of this method. The domain-specific decoder really captures the nature of the MNIST images, making it easy for gradient-based learning to fill in the details.

2.6.1 Details of the model

On MNIST, a mixture model works well. I used the model described in Figure 2.12, with 25 mixture components (CCs). One might expect a mixture of 10 components to do best, but having some extra worked better[21]. There is significant variation within digit classes, and apparently the model sometimes wants to treat different versions of a digit separately.

I wanted to use 10 ACs, with 10x10 pixel templates, but to reduce the risk of confusion in an already complicated program I decided to not use the same number too often, and opted for 9 ACs with 11x11 pixel templates. It probably doesn't make a big difference. Classification error rate is quite robust with respect to the number of ACs: with anywhere between 9 and 50 ACs, classification worked well (typically around 1.7% error rate, albeit with variance), but 3 ACs is not enough (classification error rate went up to 5%).

30 distortion parameters per CC, dropped out with a rate of 1/2, worked well, but further exploration revealed that any number of distortion parameters between 5 and 50 works decently. More than 50 didn't work well, with classification error rate (see Section 2.6.6) going up to 3% (with 70 distortion parameters) and 7% (with 100 distortion parameters).

At the output, the ACs' contributions are combined using the formula $\frac{\log(\sum \exp(\vec{a} \cdot 100))}{100}$, as explained in Section 2.3.3. After computing the model output that way, the standard squared error objective function is used.

2.6.2 Reconstructions, and ACs' contributions

Figure 2.14 shows reconstructions, and how they're created, for a typical run with the aforementioned settings. The 9 ACs have all found a role for themselves, though for some images, some ACs are not participating in the reconstruction, i.e. they have intensity near zero. The 9 templates are shown in Figure 2.15. Note that the contributions in Figure 2.14 are affine & intensity transformed versions of the corresponding templates.

Notice that, like most autoencoders, the system fails to reproduce some small irregularities: the reconstruction is essentially a smoothed version of the original. This is not very different from the way a person would reconstruct images, if he would only be allowed to look at the image for a little while before having to draw a reconstruction from memory.

Included in the figure caption is the reconstruction error, both with and without dropout. Here, *without dropout* means that the model (which was training *with* dropout) is run in a modified form, to have something deterministic. Section 1.8.4 describes the change, except that one thing is different from what I wrote in that section: for the distortion values, which are normally subject to dropout, I don't halve the outgoing weights. I tried that, and it made for *worse* reconstructions. Halving the outgoing weights is equivalent to halving the distortion values. My hypothesis is that the distortion units are few enough that they matter individually, unlike most units in a neural network. Therefore, the individual values choose a typical scale, during training. Halving the value takes it away from that scale.

Another interesting observation can be made from Figure 2.14: each capsule has its own typical place of application. To verify this, Figure 2.16 shows where the capsules typically place their contribution. Clearly, each has its own area.

[21] 10 mixture components was not enough: it led to a classification error rate (see Section 2.6.6) of 10%. 100 mixture components worked as well as 30 components, with an error rate of about 1.7%.

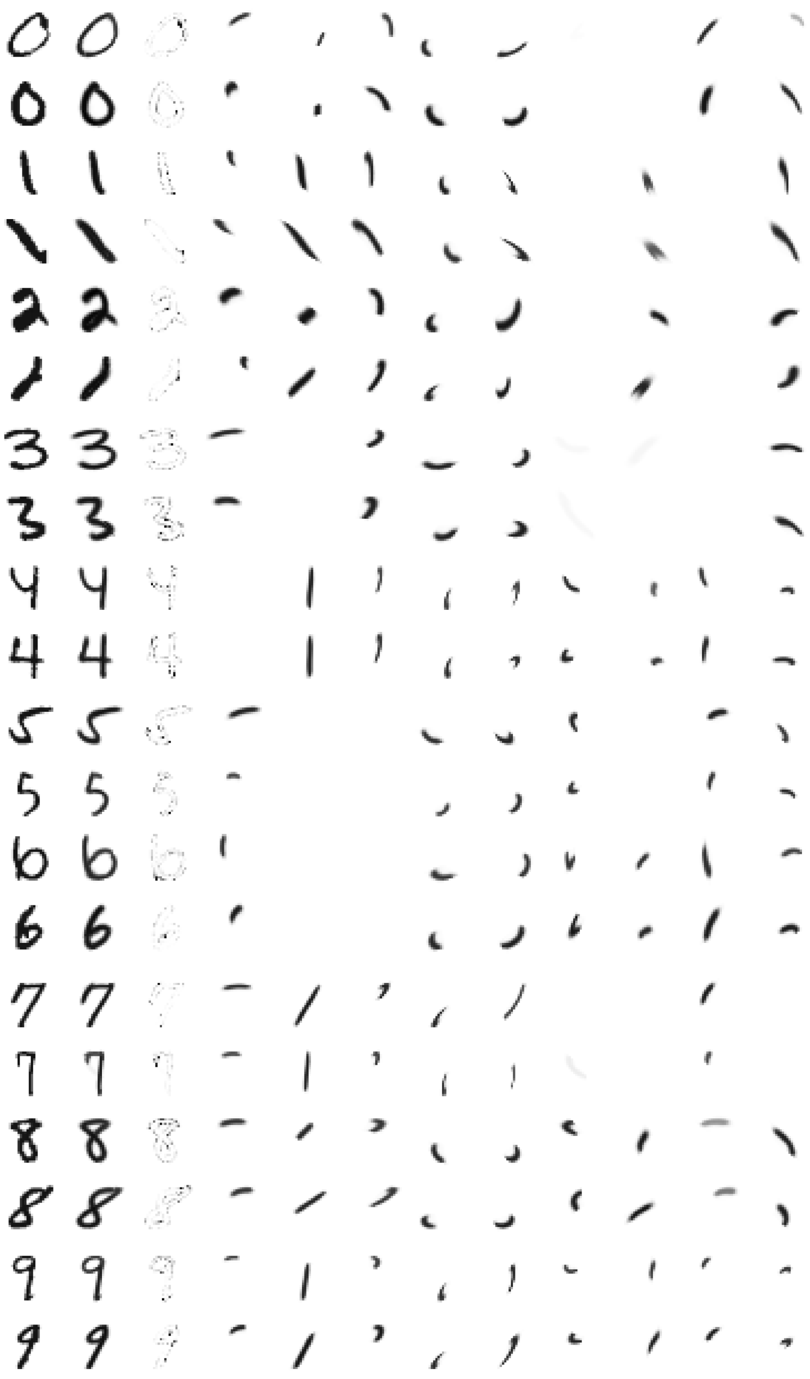

Figure 2.14: The model's reconstructions of twenty images. Columns: original image; reconstructed image; mismatch; first AC's contribution; second AC's contribution; etc. The squared reconstruction error per pixel $\frac{1}{N}\sum_{i=1}^{N}(x_i - \widehat{x_i})^2$, where N is the number of pixels and $\widehat{x_i}$ is the model's reconstruction of pixel i, is 0.0095 (15% of data variance) on training data with dropout (i.e. the training objective function), is 0.0057 (9% of data variance) on training data when dropout is disabled, is 0.0104 (15% of data variance) on held-out data with dropout, and is 0.0069 (10% of data variance) on held-out data without dropout. Notice how for two images of the same digit class, a capsule usually takes on the same role in both of the images. For example, even though the two images of the digit 5 look quite different, the first capsule represents the upper bar of the digit in both cases. This indicates that the model has learned to understand that the two images aren't all that different, even though their Euclidian distance in pixel space is large. This insight is acquired entirely unsupervised, but can be used for classification, as shown in Section 2.6.6.

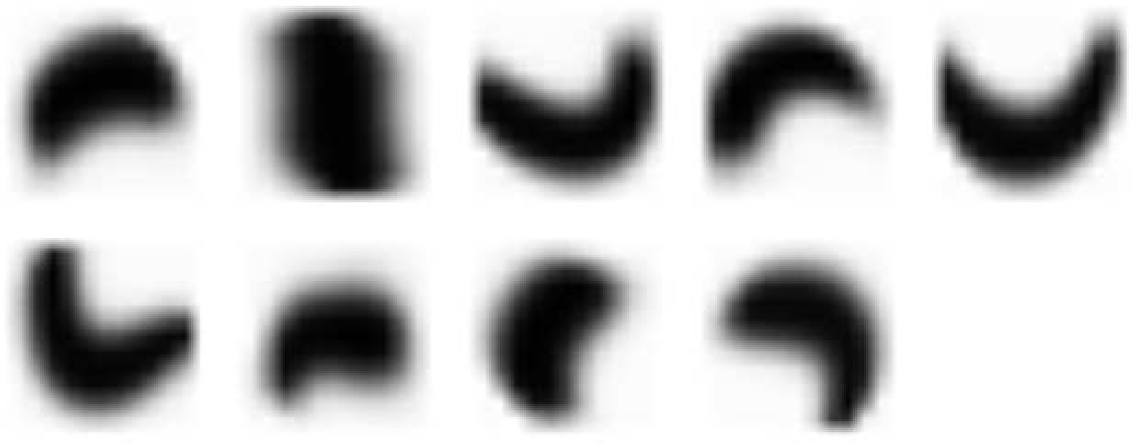

Figure 2.15: The model's learned templates. Each is 11x11 pixels, but in the model these pixels represent points instead of little squares. Between those 11x11 points, bilinear interpolation is used. The result of that interpolation is shown here. One clearly sees pieces of pen stroke. Figure 2.14 shows how those are combined to form digits.

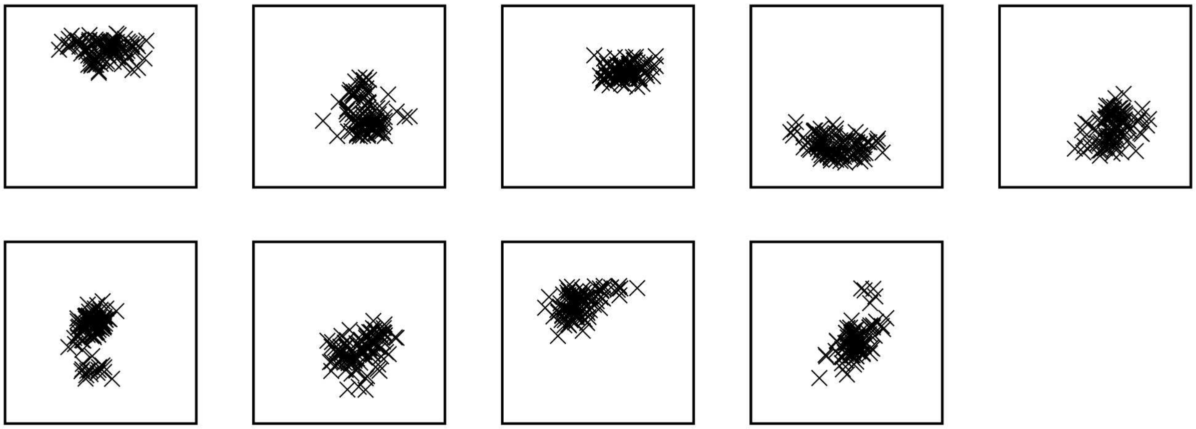

Figure 2.16: The places where each capsule is applied. This shows the centres of gravity for the capsules' contributions. The first of the 9 squares shows where the centre of gravity for the first capsule's contribution is, for 100 MNIST images: usually somewhere in the upper half of the image. The order of the 9 capsules is the same as in the other figures. Compare this to Figure 2.14.

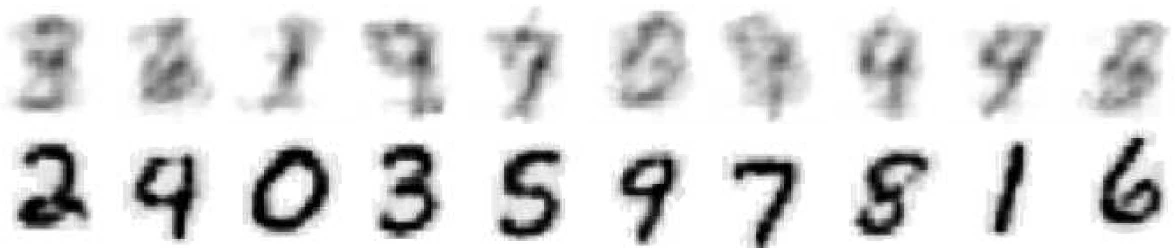

Figure 2.17: Top: the means of a mixture of 10 factor analyzers, trained on the raw pixel values of the MNIST dataset. Bottom: the means of a mixture of 10 factor analyzers, trained on the AC iGeo pose values of the MNIST dataset, as interpreted by the encoder of the capsule-based autoencoder. Visualization is done by the decoder of the autoencoder.

2.6.3 Modeling the MNIST distribution with a simple generative model

The idea is that these MNIST images are better represented as collections of pen strokes, placed appropriately, than as collections of pixel intensities. This hypothesis suggests that simple generative model of collections of pen strokes should work better than a simple generative model of pixel intensities. I tested that by building a Mixture of Factor Analyzers (MFA) of pixel intensities, and another MFA of the pose values that the encoder of my autoencoder produces when given the MNIST images.

First, I trained a mixture of 10 Factor Analyzers on the pixel intensities in MNIST. The means of the 10 factors are shown in Figure 2.17 (top). It is clear that, in the space of pixel intensities, a model as simple as an MFA fails to accurately describe the MNIST dataset.

Second, I ran all of the MNIST images through the capsule-based autoencoder, and recorded the resulting AC iGeo poses. Thus we have a different description for every MNIST image. On the collection of those descriptions I then ran a mixture of 10 factor analyzers. The 10 learned means are in AC iGeo pose space, but can be translated to image space by the decoder of the autoencoder. The result of doing that is shown in Figure 2.17 (bottom). Here we see that the MFA found 10 mixture components that describe the data much more accurately. The conclusion is that in AC iGeo pose space, the structure of the MNIST dataset is sufficiently simple and straightforward that even an MFA, i.e. *a very simple model*, can do a good job modelling it.

2.6.4 The effect of individual distortion parameters

A CC has three different types of pose variables: intensity, geometric transformation, and distortion. The effect of the first two is pre-defined, but the effect of the distortion variables is learned. To visualize what meaning a CC has chosen to give to its distortion variables, we first ask it to produce its most typical output image: we give it typical intensity and geometric transformation values, and distortion values of zero. Zero distortion is not that strange for a CC, because it's used to the distortion values getting dropped out. The output, given this pose, gives a good sense of what sort of images the CC typically produces.

Next, we focus on the first distortion variable. If we give it a different value (while keeping the other distortion variables set to zero), the CC will produce different output. If we give it fairly negative values, like -2σ where σ is the standard deviation of observed values of that distortion variable, we get to see how far it typically changes the output. If we give it the value $+2\sigma$, we see the other end of the spectrum, and intermediate values give somewhat less transformed outputs. This procedure can be repeated for the other distortion variables. Thus, we get an overview of which variation each distortion variable has come to stand for. The result of that, for two CCs in the mixture, is shown in Figure 2.18.

Each row in the figure describes a different distortion variable (the CCs in this model had 8 distortion variables). In the middle of the row is the image that results when the variable is set to 0. To the left, the result is shown of more and more negative values; to the right are positive values. Of course, this way we only see images where all but one of the distortion variables have value zero, so we don't see how the various distortion variables interact. Nevertheless it's informative.

I chose to show this for a model where the CCs had fewer than 30 distortion variables, because the more distortion variables there are, the smaller the effect of a single one becomes. With 8 distortion variables, each has a significant influence, which makes this method of visualization more effective.

2.6.5 The mixture: all components try to reconstruct

After looking at the internal workings of a single CC, we can also look at what the mixture is doing. As mentioned in Section 2.4.5, even mixture components that are not usually chosen to handle a particular image need to be somewhat prepared to handle it anyway, because of the stochastic behaviour of the mixture manager. Figure 2.19 shows the reconstructions of 20 different images, by each of the 25 mixture components.

2.6.6 Clustering as semi-supervised learning

Most autoencoders end up as data pre-processing modules for other systems like classifiers: the classifier uses not the raw (original) representation of the input for a training case, but instead the code that the autoencoder assigns to that input. We can choose to do that with just a small part of the code: just the mixture component index, i.e. nothing more than just that one categorical variable. Another way of saying this is that we let the mixture-of-CCs autoencoder cluster the data into 25 clusters (if we have 25 CCs), and then we ask the classifier to assign a label to each cluster. All inputs that the mixture manager assigns to one cluster get the same label.

Thus, training the classifier is a very small job: it has to choose 25 labels (namely one for each cluster). In this example, the classifier only gets to make 25 choices, with 10 options per choice. That's quite different from learning a million neural network weights, or choosing thousands of support vectors from a collection of sixty thousand. On the one hand, this means that it won't be able to do a good job if within many clusters there are lots of data cases that need to be given different labels. On the other hand, the classifier only needs a tiny bit of training data to properly make its decision worth an amount of information of $\log_2(10^{25})$ bits.

We can turn this set-up into a semi-supervised learning situation, as follows. First, all training data is provided without labels, and the mixture-of-CCs autoencoder gets trained on that. Second, the classifier is given the mixture manager's clustering result, and from each of the 25 clusters it gets to pick one training case for which it will be given the correct label. That might be, for example, the one training case that gets assigned to the cluster with the highest confidence, by the mixture manager. The classifier then puts the label of that training case on the entire cluster.

At test time, we get an unlabeled image, and ask the mixture manager to which cluster the image belongs. The system will then output, as its guessed label, the label that was assigned to that cluster by the classifier during "training".

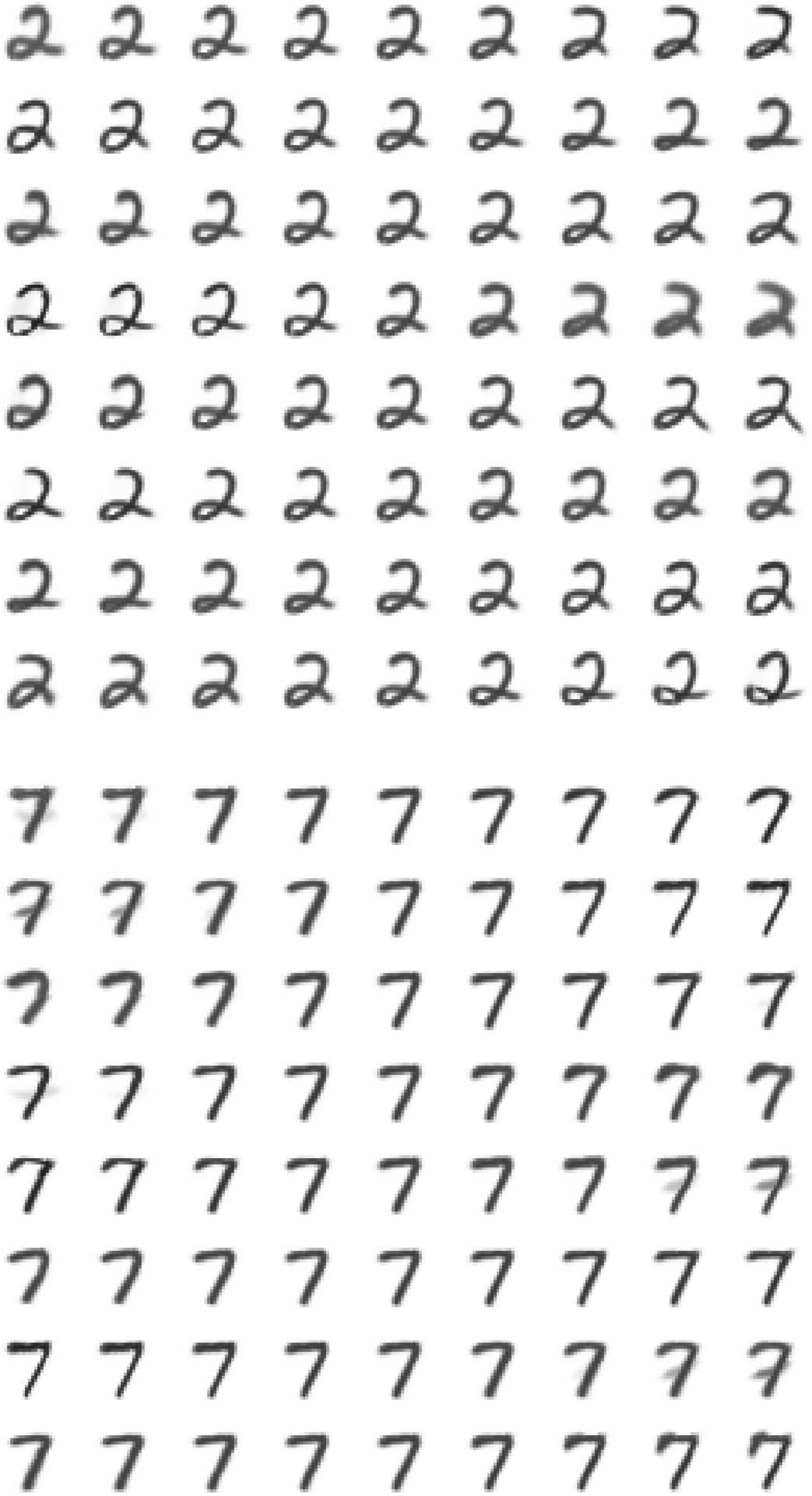

Figure 2.18: The effect of changing individual distortion variables, for two CCs. Notice how most distortion variables control only one or two aspects of the appearance: they're fairly specific. This is encouraged by applying dropout to the distortion variables (see Section 2.4.4).

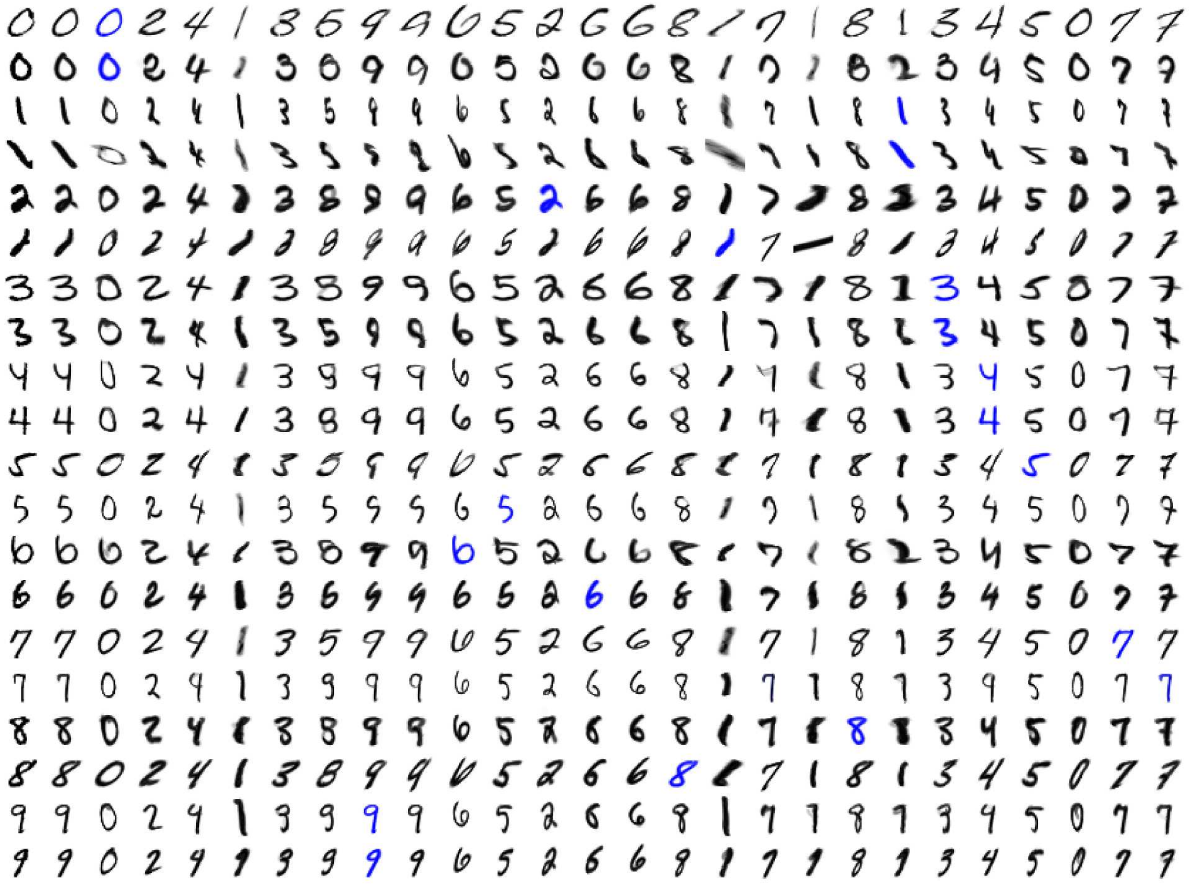

Figure 2.19: Reconstructions by different mixture components. First column: the original image. Second column: the reconstruction by the chosen mixture component. Third column: the reconstruction by the first mixture component. Fourth column: the reconstruction by the second mixture component; etc. The mixture component that the manager chooses is indicated by a blue reconstruction. Notice that each mixture component is, of course, specialized to one type of images, but has quite some flexibility by using different poses (distortion, geometric, and intensity). This allows for some creative attempts to still somewhat reasonably reconstruct images that the component is definitely not specialized in.

This system trains with only 25 labeled cases[22]. The rest of the information comes from the unlabeled data, and the domain-specific knowledge that has been put into the decoder section of the autoencoder. Applied to the model described above, it gets a good error rate on the task of MNIST classification. 11 runs with different random seeds gave a mean error rate of 1.74%, with standard deviation = 0.4%, min = 1.30%, and max = 2.59%. Given that it was trained with only 25 labeled cases, this is impressive. It shows that the clustering, which is performed without any labeled data, closely matches the human perception of how these images should be clustered.

This is not the first method to attempt classification of the MNIST dataset with only a few labeled cases. (Ranzato et al., 2007) trains multiple translation-invariant layers of feature detectors, unsupervised. Supervised training can then be done on the learned representations. When they train these feature detectors on all of MNIST (unsupervised), and then train a simple classifier on top (supervised, using only 2,000 labeled training cases), that model achieves a 2.5% error rate on the MNIST test set. With 5,000 labeled training cases the error rate drops to 1.5%. More recently, (Bruna & Mallat, 2010) built a system of invariant feature detectors that achieved a 1.7% error rate on MNIST using 2,000 training cases, not using any additional unlabeled training data. Although these numbers cannot be compared directly to those that of the capsule-based autoencoder described in this work[23], it does become very clear that a 1.74% error rate with only 25 labels in the training data is excellent.

2.7 Experiments on the TFD

I ran roughly the same model on the images of faces from the Toronto Face Database (Susskind et al., 2010). However, face images don't come in 10 groups as naturally as digit images do, so a mixture isn't appropriate here. Instead, I used a single CC, with 10 distortion parameters and 20 ACs (a face consists of more pieces than a digit). This task is more difficult than MNIST, so to get reasonable reconstructions with not too large a model, a dropout rate of 10% instead of 50% worked better.

The same figures can be made as for the MNIST experiment: Figure 2.20, Figure 2.21, and Figure 2.22.

Clearly, the system performed less well on the TFD than it did on MNIST. The reconstructions are not totally unreasonable but are definitely not as good as they were on MNIST: TFD reconstruction error is 29% of the data variance, compared to 10% for MNIST. However, the 10 degrees of freedom that the model chooses as distortion variables are definitely interesting (see Figure 2.22). Some of the ACs' contributions correspond to identifiable parts of a face, like the third last AC in Figure 2.20, but many are not the very local and identifiable "mouth" or "eye" kind of contribution that I was hoping to see.

What exactly causes the model to struggle with TFD is an open question. Images of faces are far more complex[24] than images of digits, but I do not know exactly which of these complexities is the problematic one. Whichever one it is, the result is that with the set-up that I investigated, very accurate reconstructions are not achieved. In other words, there is substantial underfitting. This is also seen when one compares performance on training data to performance on test data: there's no appreciable

[22] The system needs to pick a label for each of its 25 clusters, so it needs 25 labeled training cases: one representative training case per cluster. In my experiments, getting a label for the training case that most confidently gets assigned to the cluster is entirely sufficient. I found that using labels for all training cases that go to a cluster, and then choosing the majority label as the label to stick on the cluster, results in the same cluster labellings.

[23] For one, the results described here are for 25 *chosen* labels.

[24] An image of a face has lighting variations, it's truly real-valued as opposed to nearly binary, and it has many components that can vary, either individually or together.

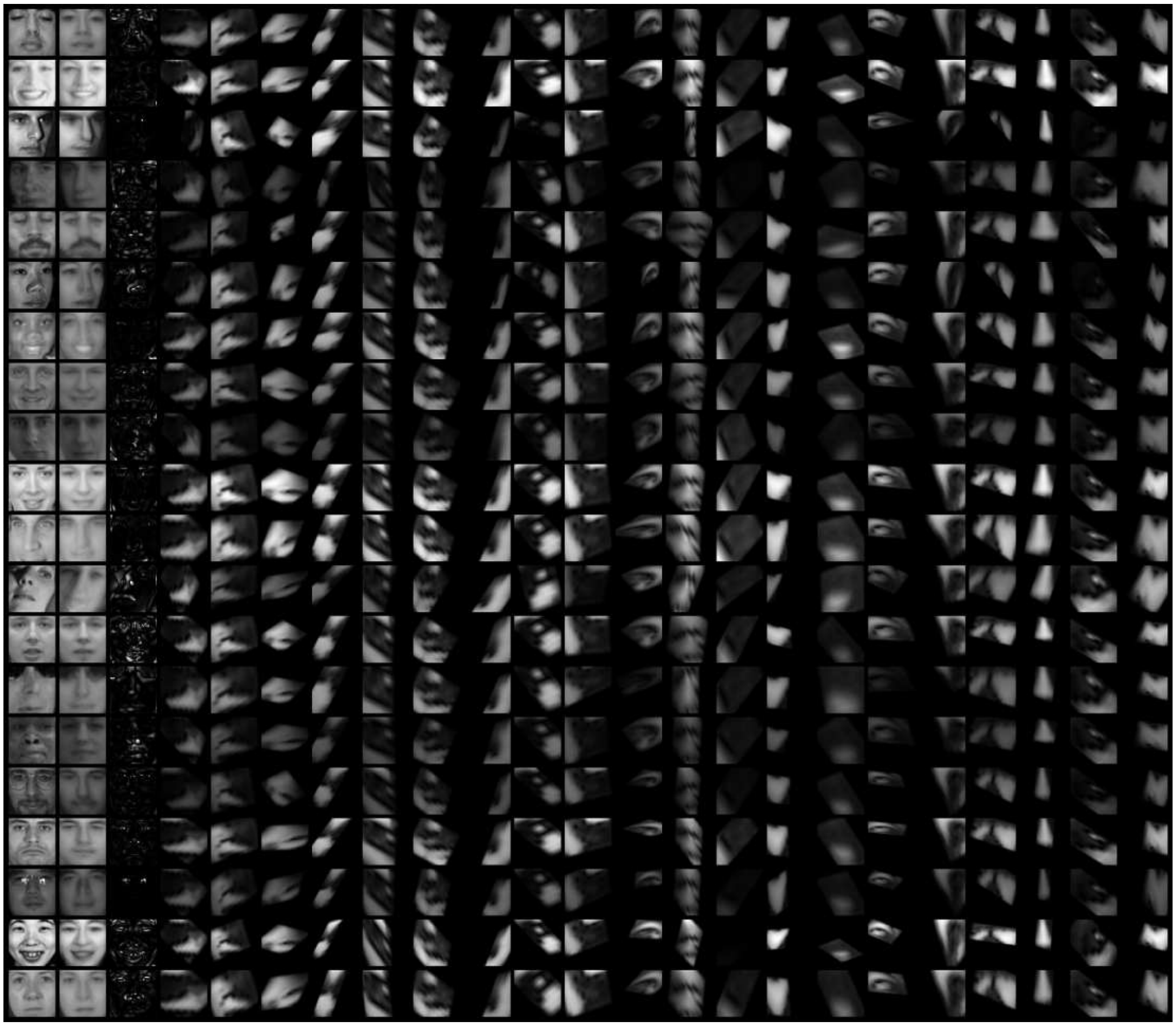

Figure 2.20: The model's reconstructions of twenty images. Columns: original image; reconstructed image; mismatch; first AC's contribution; second AC's contribution; etc. The squared reconstruction error per pixel is 0.0061 (34% of data variance) with dropout and 0.0053 (29% of data variance) without.

difference, not even when the system is run without dropout.

A reasonable hypothesis is that it takes more than 10 numbers to describe the specifics of a face, so with 10 distortion variables the model just isn't expressive enough. (Cootes et al., 2001) uses 55 pose parameters. Adding more distortion variables is one of the first things I want to explore next.

2.7.1 Multi-channel ACs

One concern is that the components of faces vary in more ways than just affine transformations: lighting variation causes differences in shadow patterns and intensity, and the simple ACs that I implemented are not flexible enough to model that well. To investigate if that was the main stumbling block, I made a different kind of ACs that can handle intensity gradients, and smooth blending of images: multi-channel ACs.

A multi-channel AC contains more than one template (say 3), and for each template, there is an

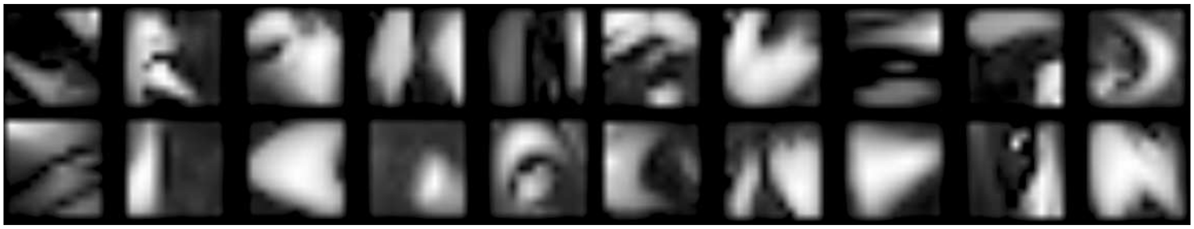

Figure 2.21: The model's learned templates. Some of the templates are usually rotated, and are therefore hard to recognize as parts of faces. Compare this figure to Figure 2.20.

intensity parameter in the AC's pose. In the rendering process, the templates would be linearly combined, weighted by their intensities. If there are three templates, then an AC has 9 pose parameters (6 geometric and 3 intensity) instead of 7 (6 geometric and 1 intensity), i.e. a modest increase[25]. It does, however, provide significant additional flexibility.

A multi-channel AC representing some three-dimensional object could, for example, have three templates describing the appearance for three different positions of the light source. Interpolation between those three templates would be a useful approximation to fully modelling appearance for every possible light source. In computer graphics, a "directional light source" has not only an intensity but also two degrees of freedom in its location[26], which could be translated to these three intensity pose values by a simple interpolation.

If the output model (see Section 2.3.3) is to simply add up the contributions from all ACs, multi-channel ACs could be emulated by simply having multiple ACs that always produce output together in the same location. However, this emulation does not work with the maxSemiSoft output model. Also, replacing one three-channel AC with three single-channel ACs does introduce much more compute work and many more pose variables (21 instead of 9, for the above example). Therefore, multi-channel ACs could potentially make a big difference.

In a toy experiment, multi-channel ACs showed that they can indeed learn to accurately model intensity gradients. The dataset for this toy experiment consisted of images of one face (the same face for the entire dataset), with an intensity gradient as well as an overall intensity multiplier. Thus, there are three degrees of freedom (the overall intensity multiplier, the intensity gradient direction, and the intensity gradient slope). A single three-channel AC was trained to model this, and did so successfully (see Figure 2.23).

However, when applied to the task of modeling the TFD images, three-channel ACs did not do significantly better than the original single-channel ACs. Given the same amount of wall clock time, they did worse, because rendering a multi-channel AC, and backpropagating derivatives through that rendering process, takes more time than it does for a single-channel AC. This suggests that the intensity gradients in TFD images are not the main problem. However, negative results are always ambiguous, hard to interpret, so until we have a positive result, such interpretation is essentially guesswork.

[25] It does make for significantly more template parameters, but the number of decoder parameters does not appear to be an issue, because a CC has many, many more and doesn't overfit either.

[26] There are only two degrees of freedom in its location, because its distance is fixed to be infinite.

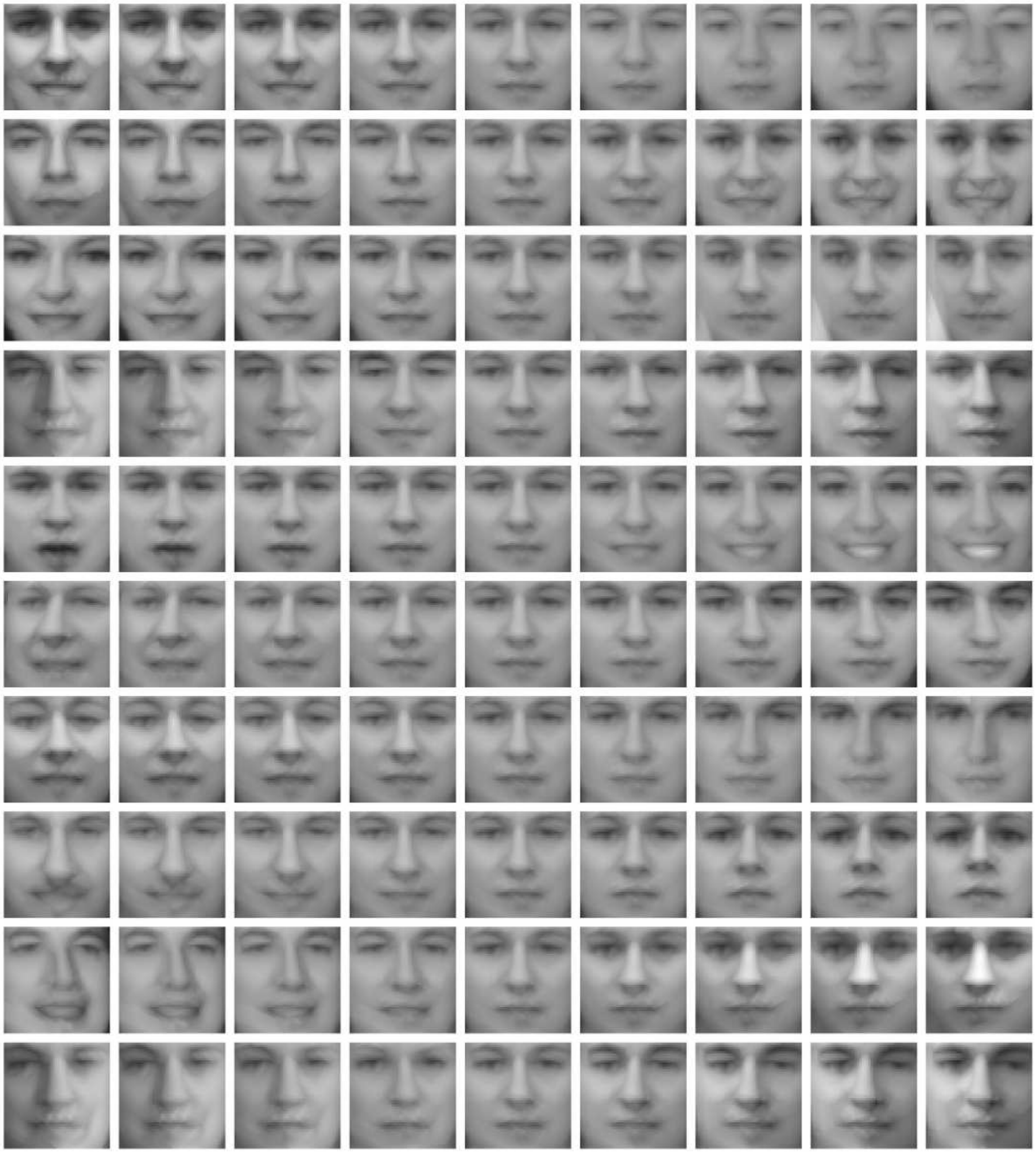

Figure 2.22: The effect of changing individual distortion variables, for the one and only CC.

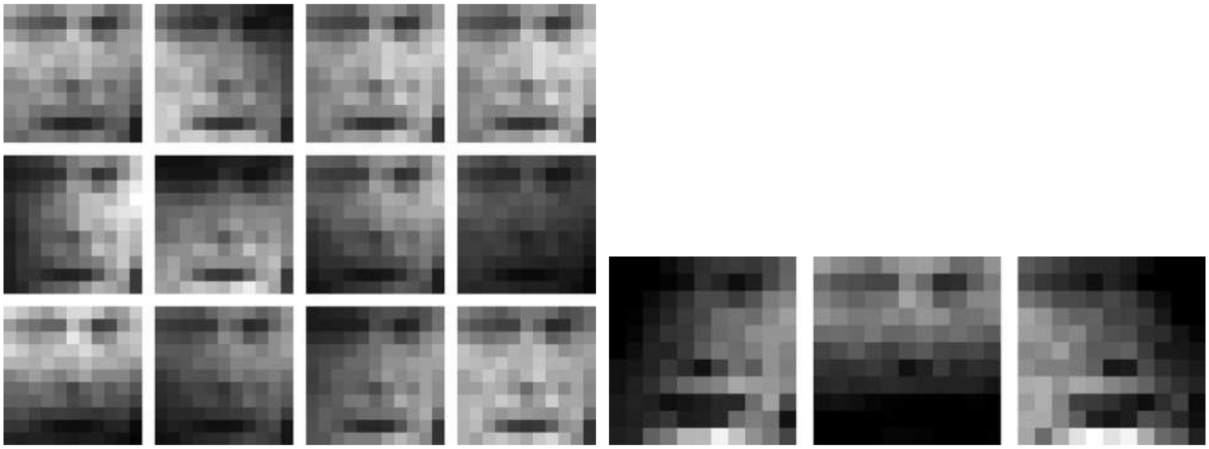

Figure 2.23: The *intensity gradient* toy experiment that confirmed that a three-channel AC can learn simple intensity variations with three degrees of freedom. Left: 12 samples of the dataset. If you squint, they change from abstract art to faces. Right: the three channels of the learned template.

2.7.2 Another output model: explicit depth modelling

Another hypothesis is that the output model is inappropriate for modeling images of faces. There is something unsatisfactory about the $\text{maxSemiSoft}(\vec{a}) = \frac{\log(\sum \exp(\vec{a} \cdot 100))}{100}$ output model: if the image is inverted (i.e. black → white and white → black), the output model would work quite differently, and possibly worse. For a more concrete example, notice how this output model nicely allows drawing bright foreground objects on a dark background (brightness dominates and thus becomes foreground), but doesn't allow for drawing dark foreground objects on a bright background (the background would dominate and hide the intended foreground objects). For MNIST this is fine, because in MNIST the foreground is always of high intensity while the background is of low intensity[27], but not every type of image is like that. In particular, for the TFD images it already feels much less natural. It gets even more unpleasant when the images are in full colour instead of just grey scale.

A solution is that the output model could make the foreground vs. background issue explicit. The idea is to add a second "template", much like in multi-channel ACs. This second template contains not an image for display, but rather a pixel grid where each pixel indicates to what extent the corresponding pixel of the template is considered to be foreground or background. This added "template" could be thought of as the depth (distance) value of the pixel. It is a poor man's implementation of 3-dimensional occlusion, in an otherwise still 2-dimensional world.

Optionally, one could also add an extra pose parameter, that could modify this "depth" value, additively or otherwise. This feature could also be integrated into the recursive CC structure, by specifying that a CC's "depth" is automatically added to the depth of its component ACs.

With this output model, one can try to make the depths of different AC contributions so different that the final value of every reconstruction pixel is determined almost exclusively by just one of the ACs, although it won't be the same AC for every pixel. That way, the image is segmented: every pixel is assigned to one of the visual components (the ACs' outputs).

Experiments with this output model are in progress, but unfinished and therefore not in the scope

[27] That is, if one interprets pen stroke as high intensity, which does not match reality but is nevertheless the standard approach.

of this thesis.

2.8 Comparison to similar models

2.8.1 Active Appearance Models

Active Appearance Models (AAM) (Cootes et al., 2001) have much in common with the approach described here. Both AAMs and this autoencoder model several types of variation that together determine the reconstruction of the given image:

- Global shape transformation. In an AAM, this is a similarity transformation, i.e. translation, rotation, and uniform scaling (no shearing or axil-specific scaling). In the autoencoder, it's a full affine transformation (CC geometric pose).

- Global intensity transformation. In an AAM, this is both multiplicative and additive. In the autoencoder it's only multiplicative (CC intensity).

- Local shape transformation. In an AAM, this is a linear transformation of the coordinates of hand-picked anchor points like eye pupils and mouth corners for a face, relative to the globally transformed constellation. In the autoencoder this is an affine transformation of the parts (the ACs) relative to the constellation (the CC), with a natural representation of the affine transformation parameters being modeled by the *distortion → AC-in-CC poses* DNN whose input is the distortion values.

- Local intensity transformation. In an AAM, this is a linear transformation of texture pixel intensities, before the shape transformation. In the autoencoder, this is the ACs' intensity multiplier, or more generally the channels' intensity multipliers in a multi-channel AC (Section 2.7.1). This AC-in-CC intensity scalar is nonlinearly produced by the *distortion → AC-in-CC poses* DNN.

Thus, both models are very direct implementations of the analysis-by-synthesis paradigm, and comparison is straightforward. There are significant differences:

- The encoder is completely different. In an AAM, the encoder is non-parametric, i.e. given a new image, we start with a guess for the pose and optimize from there. There is no learned encoder. On the one hand, this gives hope for more flexibility and more case-specific fine-tuning of the pose; on the other hand, it is slower.

- The training set is different, too. An AAM uses a training set where the same set of anchor points is annotated in every training case. The autoencoder just uses images.

- Although for AAMs the training set is more expensive, for the autoencoder the training procedure is expensive - that is the price of its flexibility. An AAM is simple enough that training takes very little time. The autoencoder, on the other hand, must not only learn the details of its more sophisticated decoder (i.e. create the code language), but must also learn an encoder to do the code inference that for AAMs is part of the training data.

- As mentioned above, the transformations in an AAM are linear functions of the pose, whereas the autoencoder uses nonlinear (DNN) transformations.

- The background is handled differently. An AAM doesn't try to model it; the autoencoder does. This is a major weakness of the autoencoder: if it cannot model the entire image, it will typically also fail to model just the foreground. Perhaps the autoencoder should be allowed to output "background" pixels in its reconstruction, where instead of the reconstruction error there would be a constant amount of loss per pixel. If this works, it could be a powerful segmentation method: it looks at a large image, finds something it can model with few pose parameters, and marks the rest of the image as "future work" (i.e. the algorithm can be repeated).

- AAMs are simpler and require more expensive training data, but they have been successfully applied to images of a size and complexity that is not commonly handled with DNNs today. Thus, AAM-like models are state of the art in practical computer vision, whereas the autoencoder is a proof of concept, with less success today but more promise for the future.

2.8.2 Recursive Compositional Models

Recursive Compositional Models (RCMs) (Zhu & Yuille, 2006; Zhu et al., 2008a; Zhu et al., 2011) go further: they define a hierarchy consisting of an arbitrary number of layers.

RCMs are quite different from Active Appearance Models and the autoencoder, in two ways: they are energy-based models with discrete variables and a Gibbs distribution over their states, and they are conditional models, so they never generate the image. What they do have in common with the autoencoder is the compositional (hierarchical) set-up.

To create that hierarchy, the latent variables of an RCM are laid out in a tree[28] configuration, with the scope of the top unit (the root) being the entire object, and the scope of lower layer units being parts and sub-parts. The connections between parents and children (and among siblings) make additive contributions to a global energy function. In some RCMs, all latent variables have interactions with the image; in other versions, only the leaves of the tree do so.

Energy-based models require more complicated inference than the simple deterministic feedforward networks of the autoencoder. However, with sufficient restrictions, inference can be made quite doable. RCMs do this by placing the nodes in a tree configuration and by allowing only a discrete set of possible values for each node. That way, dynamic programming becomes feasible.

RCMs have a curious and exciting feature, very different from the autoencoders: in an RCM, the nodes at different levels (in the autoencoder we would say the *poses* at different levels) are all of the same type. In the autoencoder, higher level capsules have to have more pose variables, but RCMs take a different approach. In an RCM, the state of a high level unit does not fully describe the state of everything in its subtree; rather, it *summarizes* it. Thus, RCMs elegantly address the issue of constant length descriptions at different scales. The autoencoder cannot do that.

An RCM can work without the high level units fully describing the state of their entire subtree, because RCMs are energy-based stochastic models. Thus, if a high level unit doesn't describe every detail of what "its" descendants are doing, the descendants can still do their job. In a deterministic autoencoder this is impossible: the state of the ancestor has to fully determine the states of its descendants. Thus, in the autoencoder, if more layers are to be added (see Section 2.9.1), the higher layer capsules will have

[28] RCMs can also be set up with an almost-tree graph: all nodes that share the same parent are fully interconnected. To retain the efficiency that a tree enables, one must, in this case, demand that a node cannot have many children, e.g. no more than 5.

to have sizeable states, i.e. a large number of distortion values. This is not ideal, but it's a consequence of the deterministic nature. RCMs do not suffer from that.

This *summarization* approach also explains how a discrete set of unit values can be sufficient: they are not expected to fully describe everything anyway.

The training data for RCMs varies between implementations. Some require the full state of all latent variables to be specified in the training data; others require less: e.g. only the state of the bottom layer of latent variables, or even just a collection of images of the same object with different backgrounds (Zhu et al., 2008b).

2.8.3 Breeder learning

Breeder learning (see Section 1.8.2 for an overview) uses a learned encoder and a fully hard-wired black box decoder. After the concession that no part of the decoder is going to get learned and that we won't be able to backpropagate gradients through it to assist in training the encoder, it is of course very nice that we can use any decoder that we want. Decoders that involve very extensive computations and discrete choices (such as *while* loops) would be difficult to incorporate into the capsule-based autoencoder, but are no problem at all for breeder learning. This means that breeder learning can handle more sophisticated decoders, and do so at a lower cost in terms of software engineering.

It is not entirely satisfactory that no part of the *breeder learning* decoder can be learned. However, we have excellent off-the-shelf generative models for many types of data, especially visual data. Breeder learning could be applied with industrial scale computer graphics systems today, and that certainly can't be said of the capsule-based autoencoder.

There is, unfortunately, a limitation on the decoders that can be used even in breeder learning: the code has to be fixed-size, and discrete code elements are a problem. However, as the work in this thesis shows, things that are discrete in nature can often be nicely approximated in a continuous way.

2.9 Directions of further exploration

The modular set-up of the proposed autoencoders is quite flexible, and allows for variations in many places. Below are some ideas that I feel are worth exploring.

2.9.1 More decoder layers: applying CCs recursively

The most obvious extension is to make more extensive use of the fact that there's recursive structure in this "computer graphics" decoder. Typical computer graphics systems certainly do make extensive use of this.

Composite capsules could very well contain other *composite* capsules as components. The only conceptual change is that a composite must then not only produce *iGeo* poses for its components, but also provide the *distortion* pose for each component. This is only a small change.

More difficulties might arise from the fact that a composite-of-composites has more atomic pieces: if the composite-of-composites contains 10 composites-of-atomics, and each of those contain 10 atomic capsules, then rendering the composite-of-composites requires $10 \cdot 10 = 100$ ACs being rendered. Rendering more ACs, and propagating gradients through that rendering process, could well become the compute time bottleneck. It is already a significant cost with 9 ACs. However, it may be possible to save quite a bit on this expensive, as explained below.

2.9.2 Less learning, for ACs

A large fraction of the compute time is devoted to finding gradients for the templates. This will become even more dominant with more ACs and larger outputs. However, there are ways to cut down on this cost.

Conceptually the simplest approach is to use fixed templates. They'd have to be somewhat reasonable, but perhaps this can be done. After all, in computer graphics programs, the composite objects are usually represented as, ultimately, collections of coloured triangles. With the right componential set-up, the atomic pieces can be quite simple.

Another approach would be to only compute the templates' gradients on some iterations of the optimization, and to pretend that they are zero on other iterations. One such strategy would be to compute the gradients for the first 10% of the optimization, and thereafter only once every ten iterations. It seems reasonable to hope that the templates don't need to change very much any more, after a while.

2.9.3 More sophisticated computer graphics

The computer graphics-based decoder that was implemented for this project was, of course, rather simple, compared to mature computer graphics programs: it is only two-dimensional, it doesn't have complicated lighting and atmospheric effects, and it doesn't have occlusion. All of that could be implemented.

Most three-dimensional effects, as well as lighting, are nicely differentiable. Occlusion might present some difficulties. When one object passes through another that was previously hiding it, and the two objects have exactly the same surface normal, there is a non-differentiable change in the image.

However, some intrinsically non-differentiable effects can be made differentiable using a reasonable approximation, as has been demonstrated with the mixture manager in this work. Occlusion could be handled similarly, by making its effect “soft” and encouraging the system not to linger in the soft region too long. It worked for the mixture manager, and it worked for (Salakhutdinov & Hinton, 2009b).

It should be noted that switching from a two-dimensional object model to a three-dimensional one makes the space much larger but doesn't necessarily require hugely more work. With surface meshes, computer graphics systems can use roughly two-dimensional objects (curved a bit for the extra dimension) to show a three-dimensional world (on a two-dimensional retina).

However, implementing full graphics will require quite a bit more compute power, especially if it is also to handle high resolution images.

2.9.4 Greedy layer-wise training

There are several options for training this system greedily, layer by layer, like the systems discussed in Section 1.3.

Training the encoder greedily

The simplest idea is to train the encoder using the now standard tricks mentioned in Section 1.3. For example, one could first train an RBM on the raw data; then train an RBM on the data representations in the hidden layer of the first RBM; and then use the data representations in the hidden layer of the second RBM as input to the encoder in something like Figure 2.12, with or without fine-tuning the weights of those two RBM's.

This is a reasonable idea and might be an improvement indeed. However, one of the motivations for such pre-training is a worry about vanishing gradients, and those are already dealt with by the skip-connections in Figure 2.2. Nevertheless, it might be a nice fast way of initializing the system.

Training the decoder greedily

A pre-training strategy more in the data generation spirit of the proposed autoencoders is to focus on the decoder.

One would first train an autoencoder where the decoder contains only ACs, like Figure 2.6, perhaps on small patches of the raw data. This would produce some well-trained ACs, i.e. good templates.

Next, we'd throw away everything but the ACs (the templates), and set up an autoencoder with a two-stage decoder, like Figure 2.12, to be trained on the same raw data as before. However, the templates for the ACs would be fixed to the previously learned ones. This would avoid having to learn everything from scratch, and it would have far less computation to do (see Section 2.9.2).

After that system has learned good CCs, we'd again throw away everything but the learned capsules (the CCs, i.e. their *distortion* → *AC-in-CC poses* functions), and set up a system with a three-stage decoder, where the capsules are recursive CCs (see Section 2.9.1), containing the previously learned CCs as (fixed) components. Etcetera.

Of course, this could be combined with pre-training the encoder, or one could choose to re-use (part of) the trained encoders as initialization, instead of throwing them away each time. Also, one could choose between either fine-tuning the previously learned capsules (at the cost of more computation), or freezing them entirely as suggested above.

2.9.5 Improving the encoder

Because of the nature of this project, most of the above suggestions concern only the decoder. However, the encoder could also be improved.

Standard tricks

The most obvious addition to the encoder would be some convolutional-style locality and weight sharing, or perhaps slightly less weight sharing as is done in (Ranzato et al., 2010). There is every reason to believe that this would help.

Another addition that can be made to the encoder is a fixed layer of edge detectors. Good edge detection systems are available, and one could insert one of those as the first layer of the encoder. However, the job of the whole system is to reconstruct the entire input, including all grey scale or colour intensities, and edge detectors throw this away.

A way around this problem is to change the objective function: the target output could be the edges of the image, as opposed to the image itself. That way, we are effectively back to the algorithm as described in this thesis; it's just that the data is now edge data instead of raw image data. The hope would be that this edge data is easier to interpret and reconstruct than raw image data. That is an interesting open question, and deserves a spot on the "future work" list.

One standard method that should probably be avoided is pooling with subsampling[29] after a convolutional layer (see Section 1.8.2). The reason is that pooling deliberately throws away some high-precision location information, and that information is essential for accurate reconstruction.

Domain-specific tricks

One can also try to improve the encoder in more sophisticated ways, that rely on the structure of the decoder and the code layer. If the decoder is a single CC, as it was in the presented experiments on the TFD, one could demand that the encoder first produce only the first half of the pose: the iGeo transformation. After the encoder provides that, the input image would be normalized by applying the inverse of the given iGeo transformation. A second encoder would then be tasked with deciding on the *distortion* part of the pose, using the normalized image as input. Working with normalized data is often easier, so this might work better than requiring the encoder to decide on the iGeo transformation and the distortion values in one go, from the raw input image.

One could go even further and allow the second encoder to slightly change the iGeo transformation, if it feels that the first encoder did an imperfect job. This can be made recursive, and it brings us closer to the idea of a saccading fovea, as was explored in (Larochelle & Hinton, 2010). There is no reason why we cannot proceed further in that direction, using explicitly represented geometric transformations as a tool.

2.9.6 Model selection

The traditional approach to model selection in neural networks research is to train many models, independently, and to go with whichever one works best. See Section 1.9 for more details about this - especially the recent advances in automating this process.

It is also possible to do model selection during training: one can add or eliminate units whenever that seems to be called for. This applies to both the atomic units and the capsules, even the composite capsules. Section 2.4.5 describes that sometimes, the model fails to use a capsule, *by accident*. In that case, a gentle pressure can correct the problem. However, if there are more capsules than are truly needed, then it makes sense to stop using the spare ones, *on purpose*. The objective function as it is now will always favour the use of more capsules, but something can be added to it to encourage the system to stop using a capsule if that capsule really isn't helping. For example, one could add a small L1 penalty on the sum of the capsule's template pixels (for atomic capsules). Then, if the capsule isn't helping much, the template pixels will all go to 0. If that happens, the optimizer can detect that (if it checks for such things every once in a while), and remove such discarded capsules from the model. Similarly, if all capsules are being used, the optimizer can go in and add a few more. Thus, the optimizer can choose the number of capsules, over time. This is more satisfactory than trying ten models in parallel, which only differ in their number of capsules.

[29] Unless there is a lot of overlap between the pools; ideally a stride of just 1.

2.10 Conclusion

2.10.1 Applicability

The proposed autoencoder relies on one central assumption: that the visual scene is componential. The more componential it is, the better the system can do.

Every image has some global attributes: the specifics of the lighting, the specifics of the atmosphere, and some general properties of the setting. Objects have global attributes as well. For example, when modelling a human face, the person's skin colour is a global attribute, and their dominant emotion also has effects on more than just the mouth. A handwritten digit has a global stroke width. Such global variables go against the assumption of componentiality. The model can handle them by including a large number of CC distortion variables, but it's not what the model is best at.

Therefore, modeling a person is probably more difficult than modeling a car. A car has fairly independent components: the wheels (position & possibly decoration), the frame (shape & colour), the lights (on / off), the passengers (present / absent).

Another type of image that is quite componential is the kind that people take and upload to social media. People are uploading many images a day, and storing these could be an issue. If the rate of uploads increases quicker than the price of storage media decreases, then some day there would be a need for better lossy compression than JPEG. This type of autoencoder might be that compression (the code layer could be made into a small compressed description of the image), and fortunately, there would be no shortage of training data.

In short, this model is probably best with data that is highly componential, i.e. where a part of the image can be modelled with little knowledge about the rest of the image.

2.10.2 Take-home messages

The main idea of this approach is to train a data recognition system by creating a data generation system that's half engineered and half learned. The experiments with MNIST and the TFD show that this approach has potential: on these datasets, the learnable part of the data generation system did indeed learn to work well, in the context of the engineered structure. The data recognition system learned well, too. In the case of MNIST, a nice quantitative result verified that things can be done this way that had not been achieved before: the model achieved good classification performance, using only 25 labelled examples.

There are two take-home messages. The first is that sophisticated half engineered domain-specific knowledge can be put into a data generation system, which is often easier than placing it in a recognition system. A recognition system can absorb the domain-specific knowledge when it's trained jointly with the generation system, with minimal danger of overfitting. This approach is suitable for any type of data where we can describe the structure that a good generation system would have. Visual data is not the only domain where that is the case, but it may be the most promising one, because we have very good image generation systems. However, care must be taken to ensure that the data generation framework is sufficiently flexible for the type of data that it's asked to generate. The particular data generation system used in this work seems to be more appropriate to the MNIST dataset than to the TFD images.

The second take-home message is that learning and sophisticated engineering can go together. Systems with elaborate engineered componential structure[30] can be made, where the details are filled in by learning. This requires that the effect of learnable parameters always be differentiable, which isn't trivial but not impossible either. With some engineering tricks, it's possible to use gradient-based learning for intrinsically discrete choices, like those of a mixture manager.

[30] The function that computes the loss value from the input image and the learnable parameter vector θ consists of 945 atomic steps in a Python program. The word "atomic", here, only refers to the Python perspective: some of these "atomic" components represent decidedly composite mathematical functions like log-sum-exp, or the application of the geometric transformation. On the other hand, some of these components that in Python can be called atomic may feel so insignificant that one might want to call them sub-atomic in a semantic sense. Anyway, these 945 steps, whatever we call them, can be categorized as 284 mathematical operations and 661 bits of "administration work", like reshaping matrices. The function that computes the gradient is automatically generated from this model function, and consists of 1063 steps. The point is that this is a highly componential system.

Chapter 3

Training undirected generative models

3.1 Introduction

In this chapter I address the question of how to train undirected graphical models, also known as energy-based models. There are many interesting and successful applications of these models, on a variety of types of data, but here I focus on the question of how to train them, regardless what the type of data and the details of the task are.

3.1.1 Focus on data log probability

The most common way to train generative models is to try to maximize the (log) probability of the data under the probability distribution that the model defines. This kind of training usually proceeds iteratively by computing or estimating the gradient of the sum of log probability of the training data, with respect to the learnable parameters, and changing the parameters in that direction[1]. That is the approach taken in this chapter. Therefore, its central question is how best to estimate the gradient of the training data log probability. It should be made clear that there are other ways to train generative models, like score matching (Hyvärinen & Dayan, 2005) or training the system to stay close to the training data under some Markov Chain transition operator. However, this chapter is all about estimating the data log probability gradient.

3.1.2 Likelihood gradient: two parts

The algorithms studied in this chapter are, roughly speaking, applicable to any type of undirected graphical model. The experiments were performed mostly on RBM's, for which the *positive* part of the gradient is fully tractable, but the algorithms can also be adapted to work on more general models; see for example (Salakhutdinov & Hinton, 2009a).

As mentioned in Section 1.3.2, a typical training algorithm for an undirected graphical model consists of gradient-based optimization, and the choice of approximation to the *negative* part of the likelihood gradient is the most distinctive part of the algorithm.

[1] Or at least in a direction that has a positive dot product with that gradient.

3.1.3 Global unlearning

In the past decade, two general types of such approximations have emerged. Intuitively the most appealing one is *global unlearning* (see Section 1.3.2), where $\nabla\phi^-$ is estimated from attempted samples from the model distribution, which are independent of the training data. This is appealing because energy-based models are *globally normalized* (see Section 1.5.2) so the most correct thing to do is to do unlearning on configurations taken from the global distribution.

3.1.4 Local unlearning

Global unlearning gradient estimators are philosophically appealing but have significant weaknesses compared to *local unlearning* gradient estimators, which have been popular for some time because of their easy computation, low variance, and success in creating the building blocks for multilayer models; see for example (Besag, 1986; Hinton, 2002; Hinton et al., 2006). In *local unlearning* algorithms, $\nabla\phi^-$ is estimated from configurations that are somehow close to the training data. This could be very close, as in pseudo likelihood (PL) (Besag, 1986), or somewhat close, as in CD-1, or in fact not that close at all, as happens in CD-k for large k at the cost of more computation time. However, in all cases of local unlearning the configurations on which the negative gradient estimate is based are somehow tied to a training case.

3.1.5 Outline of the chapter

This chapter investigates those two classes of training algorithms. First, Section 3.2 introduces two global unlearning algorithms and their connections to other methods like Herding. Next, Section 3.3 and Section 3.4 present a range of experiments comparing the various algorithms. Section 3.5 goes over some applications of the algorithms. Finally, Section 3.6 concludes the chapter by taking a step back and looking at the bigger picture again.

3.2 Global unlearning algorithms

3.2.1 Persistent Contrastive Divergence

CD-1 is fast, has low variance, and is a reasonable approximation to the likelihood gradient, but it is still significantly different from the likelihood gradient when the mixing rate is low. This can be seen by drawing samples from the distribution that it learns (see Figure 3.2). CD-n for greater n produces more accurate gradient estimates (in expectation) than CD-1 (Bengio & Delalleau, 2007), but requires more computation by a factor of $O(n)$. In (Neal, 1992), a solution to this problem is suggested for Sigmoid Belief Networks and general (i.e. not Restricted) Boltzmann Machines. In the context of RBMs, the idea is as follows.

What we need for approximating $\nabla\phi^-$ is a sample from the model distribution. The standard way to get it is by using a Markov Chain, but running a chain for many steps is too time-consuming. However, between parameter updates, the model changes only slightly. We can take advantage of that by initializing a Markov Chain at the state in which it ended for the previous model. This initialization is often fairly close to the model distribution, even though the model has changed a bit in the parameter update. Neal uses this approach with Sigmoid Belief Networks to approximately sample from the posterior distribution over hidden layer states given the visible layer state. For RBMs, the situation is even better: there is only one distribution from which we need samples, as opposed to one distribution per training data point. Thus, the algorithm can be used to produce gradient estimates *online* or using mini-batches, using only a few training data points for the positive part of each gradient estimate, and only a few 'fantasy' points for the negative part. Each time a mini-batch is processed, the fantasy points are updated by one or more full steps of the Markov Chain.

Of course this is still an approximation, because the model does change slightly with each parameter update. With infinitesimally small learning rate it becomes exact, but it also seems to work well with realistic learning rates.

In (Tieleman, 2008), which introduced the method to the research community investigating Restricted Boltzmann Machines, I called this algorithm *Persistent Contrastive Divergence* (PCD), to emphasize that the Markov Chain is not reset between parameter updates. This community was unaware that statisticians have used essentially the same method for a lot longer under the name *stochastic approximation*, and have made thorough theoretical analyses; see (Robbins & Monro, 1951; Kushner & Clark, 1978).

(Salakhutdinov & Hinton, 2009a) uses the same approach for estimating $\nabla\phi^-$ in a more complex energy-based model: a multi-layer Boltzmann Machine. Because that model is more complex, it also requires an approximation to $\nabla\phi^+$, but $\nabla\phi^-$ and $\nabla\phi^+$ are still estimated separately.

The PCD algorithm can be implemented in various ways. One could, for example, choose to randomly reset some of the Markov Chains at regular intervals. My initial experiments showed that a reasonable implementation is as follows: no Markov Chains get reset, ever; one full Gibbs update is done on each of the Markov Chains for each gradient estimate; and the number of Markov Chains is equal to the number of training data points in a mini-batch.

3.2.2 Why PCD works

As mentioned in Section 1.3.2, the training data likelihood objective function for an RBM consists of two components, ϕ^+ and ϕ^-, of which only the latter has an intractable gradient: $\nabla\phi^- = \underset{\vec{v},\vec{h}\sim P_\theta}{\mathrm{E}}[\nabla(-E_\theta(\vec{v},\vec{h}))]$. The gradient of ϕ^+ is tractable: $\nabla\phi^+ = \underset{\vec{v}\sim D,\vec{h}\sim P_\theta(\cdot|\vec{v})}{\mathrm{E}}[\nabla(-E_\theta(\vec{v},\vec{h}))]$. D denotes the training data distribution.

Gradient estimation using the R distribution

There is another way of writing $\nabla\phi$. Let R be some distribution over the data space, that is used for estimating $\nabla\phi$. For now, let R be the model's distribution P_θ, as a constant, i.e. R will remain the same when θ changes, though of course P_θ *will* change. With that particular choice of R, we can write $\nabla\phi = \nabla(KL(R||P_\theta) - KL(D||P_\theta))$ (see Section 4.1 for a proof, assuming that specific choice of R).

If we're using that perfect R, the sought gradient $\nabla\phi$ is exactly equal to $\nabla(KL(R||P_\theta) - KL(D||P_\theta))$, but this is trivial because then $KL(R||P_\theta) = 0$, and it doesn't suggest an algorithm. What we can do, however, is use a different R: we choose $R \neq P_\theta$. In this case, $\nabla\phi$ will not in general be equal to $\nabla(KL(R||P_\theta) - KL(D||P_\theta))$. However, because all the involved functions are smooth, if R is sufficiently close to P_θ, then $\nabla(KL(R||P_\theta) - KL(D||P_\theta))$ will be a useful approximation to $\nabla\theta$. If, in addition, R is such that $\nabla(KL(R||P_\theta) - KL(D||P_\theta))$ can be efficiently computed or approximated, we have an algorithm for approximating $\nabla\theta$.

Many gradient approximation algorithms can indeed be described as using an approximate R: one that is not exactly the model distribution P_θ. In CD-1, the approximate R is the distribution of configurations that we reach after performing one full Gibbs update from randomly chosen training data cases. In pseudo-likelihood, the approximate R is the distribution reached by resampling one variable, chosen uniformly at random. And in PCD, the approximate R is a mixture of delta distributions, one for each of the persistent Markov Chains, also known as the fantasy particles. For example, if we have 100 persistent Markov Chains, R will be the probability distribution that has 1% of its probability mass on each of the Markov Chains' states.

Learning makes the approximation worse

By changing the model parameters θ in the direction of $\nabla\phi = \nabla(KL(R||P_\theta) - KL(D||P_\theta))$, using such an approximate R, we tend to increase $KL(R||P_\theta)$ while decreasing $KL(D||P_\theta)$.

Decreasing $KL(D||P_\theta)$ is a very reasonable thing to do: we're reducing the difference between the training data distribution and the model distribution. This corresponds to following the gradient of ϕ.

Increasing $KL(R||P_\theta)$ is less obviously the right thing to do. We're pushing to make R, our approximation to P_θ, as different from P_θ as possible. This is in contrast to variational learning, which also uses an approximating distribution: there, the effect of learning is that the approximating distribution becomes a better approximation. For PCD, however, the effect of learning is that the approximating distribution gets worse. The model distribution is trying to get away from R, which attempts to approximate it.

The interaction between learning and the Gibbs transition operator

Making R, our collection of, say, 100 *fantasy* configurations, a bad approximation to P_θ, may seem undesirable but has a beneficial side-effect that can only be understood by considering the interaction of the learning and the Gibbs updates on the fantasy particles. After several weight updates have had the effect of making R and P_θ quite different, the Gibbs updates on the R particles will become well "motivated" to quickly move them towards P_θ again. A Markov Chain that's in a low probability configuration will typically change to quite a different configuration when it makes its next transition. This means that the R particles are typically moving around at high speed, always chasing P_θ that's being made different from R by the $KL(R||P_\theta)$ part of the gradient. This has the effect that the R particles are rapidly moving around the state space; much more rapidly than a regular Markov Chain with Gibbs transitions would if there were no learning going on at the same time.

The energy landscape

Another way to explain the phenomenon is by looking at the energy landscape that the RBM defines. The R particles are located somewhere, and the $KL(R||P_\theta)$ part of the gradient has the effect of increasing the energy in and near[2] those places. After the energy has been raised somewhat and the probability therefore reduced, the R particles quickly move on to another place of higher probability, i.e. they're "rolling" downhill in the energy landscape.

[2] Configurations that are similar to a specific R particle will also tend to see their energy increase. In an RBM, a configuration is a highly distributed object. The energy, being a sum of typically small terms based on which interactions are "on" in a configuration, tends to vary only a little between very two similar configurations. Thus, when the energy of an R particle is increased, the energy of similar configurations tends to increase as well.

This analysis explains several previously unexplained phenomena of PCD:

- The R particles of the PCD algorithm are always moving much more rapidly than one would expect if one looked at a Markov Chain on a static energy landscape, i.e. after the model has been trained.
- PCD never seemed to be very good at fine-tuning weights with a small learning rate: when the learning rate is reduced, the artificially enhanced "mixing rate" of the fantasy particles is equally reduced.
- PCD performs much worse when one adds *momentum* to the optimizer. The above analysis reveals why it is important that θ be able to change quickly, and not move sluggishly in one direction, which is the effect of momentum. If it takes θ a long time to stop raising the energy in a location where many R particles were not too long ago, then the favourable interaction between learning and optimization is slowed down.

3.2.3 Fast PCD: the main idea

Rapid "mixing" *and* a small learning rate

The artificially large "mixing rate" that is created by the interaction between learning and the transition operator of the Markov Chain is exploited more explicitly in the "Fast PCD" (FPCD) algorithm. In PCD, this large "mixing rate" is much reduced when the learning rate is reduced, and that is what FPCD is designed to remedy.

Introducing fast weights

In FPCD, there are two energy landscapes. The first one is that of the evolving θ that we're trying to learn, and it changes relatively slowly, because learning is something to do carefully. We call this the "slow model" with its "slow energy landscape", defined by the "slow weights" (called θ or θ_{slow}).

The other energy landscape is changing more rapidly. It is not the one that we're trying to learn, and it is discarded when the algorithm terminates. Its sole purpose is to provide a stream of approximate samples from the "slow" model, and it achieves that in roughly the way Herding works (see Section 3.2.5). We call this other model the "fast model", defined by the "fast weights".

The fast model learns with a large learning rate, so as to capitalize on the favourable interaction between learning and the Markov Chains, but it is also strongly "pulled back" (decayed) towards the slow model. This way, the approximate samples that it provides remain close to the distribution of the slow model. After the publication of FPCD, it has even been suggested that this artificial "mixing rate" is so good that it should be used to draw samples from trained models (Breuleux et al., 2010; Breuleux et al., 2011)[3].

A natural way to implement this pair of models is to explicitly represent both the slow weights $\theta = \theta_{\text{slow}}$ and the difference $\theta_{\triangle} = \theta_{\text{fast}} - \theta_{\text{slow}}$ between the fast weights and the slow weights. We call this $\theta_{\triangle}$ the "fast overlay", because it represents a relatively thin overlay on the energy landscape: the fast energy landscape is equal to the slow landscape with the overlay added on top. The fast weights θ_{fast} are not explicitly represented, but are instead a function of the slow weights and the fast overlay,

[3] Those sampling methods use a training set, and are thus a mix of two ideas: sampling from a model, and herding (which can be thought of as sampling from a distribution that is implied by the given dataset).

as $\theta_{\text{fast}} = \theta_{\text{slow}} + \theta_{\Delta}$. With this framework, instead of saying that the fast weights decay towards the slow weights, we say that the overlay decays towards zero - but clearly this is just a matter of notation, because the effect is the same.

3.2.4 Fast PCD: pseudocode

Below is a pseudocode description of the FPCD algorithm. For simplicity, considerations such as momentum and weight decay (on the slow model) have been left out, but these are easy to incorporate.

```
Program parameters:
    • A schedule of slow (a.k.a. regular) learning rates, used in step 6 below.
    • A schedule of fast learning rates, used in step 7 below. A constant fast learning rate is recommended.
    • A decay rate $\lambda < 1$ for $\theta_{\Delta}$. Recommended value: $\lambda = \frac{19}{20}$
    • The number $N_f$ of fantasy particles. The recommended value here is the same as the mini-batch size, which in my case was 100.
Initialization:
    • Initialize $\theta_{\text{slow}}$ to small random values.
    • Initialize $\theta_{\Delta}$ to all zeros.
    • Initialize the $N_f$ Markov Chains $\vec{v}^{-}$ to all zero states.
Then repeat:
    1. Get the next batch of training data, $\vec{v}^{+}$.
    2. Calculate $\vec{h}^{+} = P_{\theta}(\vec{h}|\vec{v}^{+}, \theta_{\text{slow}})$, i.e. activation probabilities inferred using the slow model. Calculate the positive gradient $g^{+} = {\vec{v}^{+}}^{T}\vec{h}^{+}$.
    3. Calculate $\vec{h}^{-} = P(\vec{h}|\vec{v}^{-}, \theta_{\text{slow}} + \theta_{\Delta})$, i.e. activation probabilities inferred using the fast model. Calculate the negative gradient $g^{-} = {\vec{v}^{-}}^{T}\vec{h}^{-}$.
    4. Update $\vec{v}^{-}$ = sample from $P(v|\vec{h}^{-}, \theta_{\text{slow}} + \theta_{\Delta})$, i.e. one full Gibbs update on the negative data, using the fast weights.
    5. Calculate the full gradient $g = g^{+} - g^{-}$.
    6. Update $\theta_{\text{slow}} = \theta_{\text{slow}} + g \cdot$ regular learning rate for this iteration.
    7. Update $\theta_{\Delta} = \theta_{\Delta} \cdot \lambda + g \cdot$ fast learning rate for this iteration.
```

Notice that when the fast learning rate is zero, we get the original PCD algorithm. This means that one can interpolate between the two algorithms by choosing a small “fast learning rate”.

What value for N_f (the number of fantasy particles) is optimal is an open question. The above recommendation is based solely on the fact that it is simple and it worked - I have not thoroughly explored alternatives. There is something to be said for having a large number of fantasy particles: only a large ensemble can hope to approximate the model distribution well. On the other hand, it also requires more compute time: using 1 particle and performing 100 updates on it at every step clearly has advantages over using 100 particles with just 1 update at a time. 1 particle with 100 updates can move further in each iteration, and if 100 particles with 1 update end up staying close together, 1 particle with 100 updates is clearly preferable. On the other hand, with Moore’s law having turned towards greater parallelism rather than greater clock speed, 100 particles also have an obvious advantage. A second reason to prefer 100 particles with 1 update per iteration is that using 1 particle with 100 updates per θ update might not be much better than 1 particle with just 1 update: this algorithm is for models that are so stiff that a regular Markov Chain essentially gets nowhere, and that is what 100 updates without changing θ would be. However, this is simply an open question.

3.2.5 Related algorithms

Herding

PCD is closely related to a process called “Herding” (Welling, 2009b; Welling, 2009a). If one replaces the Gibbs transition operator in PCD with a transition operator that deterministically goes to the greatest probability (i.e. lowest energy) configuration, the resulting process is Herding. Herding has been shown to also work with less rigorously energy-minimizing transition operators (Welling, 2009a), and may therefore be helpful in analyzing what is happening to the fantasy particles of PCD.

Herding is quite different from learning a parametric model. In Herding, the weights are always changing rapidly and are not the object of interest. Instead, we record the R states over time, and the distribution of that collection is, in a sense, the way the model architecture perceives the training data. Some basic statistics of the distribution are guaranteed to match those of the training data exactly, in the limit of long herding sequences.

Two-timescale stochastic approximation algorithms

PCD is an instance of the broad class of systems that is known in the statistics literature as “stochastic approximation” (Robbins & Monro, 1951; Bharath & Borkar, 1999). FPCD is an instance of the “two-timescale” variety of stochastic approximation algorithms. In two-timescale stochastic approximation algorithms, two random variables evolve simultaneously: one slowly and one rapidly. In the case of FPCD, the slow one is θ_{slow} and the rapidly moving evolving one is $\theta_{\text{fast}} = \theta_{\text{slow}} + \theta_{\Delta}$.

In idealized mathematical analysis aiming to prove almost sure convergence, the rapidly evolving variable, viewed from the perspective of the slowly evolving one, will tend, in the limit of the process, towards evolving so rapidly that it approaches equilibrium. Conversely, in the limit, from the perspective of the rapidly evolving variable, the slowly evolving variable will tend to zero movement, i.e. it becomes static. FPCD, being an algorithm for a practical purpose, instead of a mathematical analysis for a theory purpose, has no *limit* and the slowly learning model need not come entirely to a halt. However, it should start to move at least somewhat more slowly over time, i.e. the “regular” learning rate should decay.

The Wang-Landau algorithm

PCD and FPCD are also related to the Wang-Landau (WL) algorithm.

WL serves not for learning a model, but for analyzing a fixed probability distribution. However, it doesn’t do this by producing samples of the distribution. Rather, it aims to estimate the size of different regions of the probability space. What connects it to PCD, FPCD, and Herding, is its use of a random walk that develops aversion to areas where it spends much time, in order to keep moving around rapidly.

WL is described in (Wang & Landau, 2001) in a way that takes advantage of the discrete energy levels of simple Potts and Ising models. The authors note that although they apply a random walk to energy space, the same type of walk could be applied to any other parameterization of space. Thus generalized, the idea is as follows.

The algorithm starts by dividing the probability space in a number of regions (in the original paper, every energy level is a region). It expects to visit each region, and to keep some data about each region, so the number cannot be too large, but it can be in the thousands. The algorithm then sets up an aversion scalar for each region; these scalars are all initialized to 1.

Then it runs a modified Markov Chain on the probability distribution. The modification is that it makes its transition decisions not based on the original probability of the various points, but instead based on their probability divided by the aversion scalar of the region to which the point belongs. After every transition, the region in which this random walk currently finds itself gets more aversion: its aversion scalar is multiplied by a small constant greater than 1. As in Herding/PCD/FPCD, this has the effect of encouraging the random walk to move to another part of the space relatively soon, and thus ensures better exploration.

The effect of this policy is that in the end, all regions are visited roughly equally often. When that is accomplished, we can estimate the relative sizes of the regions *under the original probability distribution*: the recorded aversions are used as those estimates. While having a good estimate of the marginal probabilities of the regions doesn’t provide samples, it does provide much valuable insight.

Note that after the random walk, WL no longer has any interest in the visited states - this in contrast to PCD/FPCD/Herding.

For more details and analysis, see the original publication (Wang & Landau, 2001).

A variation, more like FPCD, is described in (Zhou et al., 2006). It eliminates the requirement that the state space be small enough to visit every point with a random walk. It does this by developing aversion to a localized region around the point that’s visited. However, this means that computing the aversion (as the sum of regional aversions) now takes time linear in the number of iterations, as opposed to being an O(1) look-up. Another problem is that to produce the same kind of estimates of the region sizes as the original WL algorithm does, one still needs a loop over all discrete points in the space[4].

[4] The authors only mention that “we calculate the thermodynamic quantities (...) with numerical integral”. However, I cannot think of a way this might be done exactly without such a loop. The authors demonstrate their algorithm on a task where the sampled space is small enough that such a loop is indeed feasible.

Differences and similarities

All of these algorithms capitalize on the rapid exploration that a random walk is forced into when it develops aversion to wherever it is. However, there are significant differences.

Random walk histogram One distinguishing factor is in the type of random walk state histogram that these methods aim for. WL is based on a flat histogram, where all regions are visited roughly equally often. Herding and FPCD, on the other hand, aim for a histogram like that of the given model or data. Thus, Herding and FPCD are much more like traditional Markov Chain Monte Carlo algorithms than WL.

Relation to datasets The second major difference between these various algorithms is in what the algorithm is trying to analyze. WL and FPCD seek to analyze a probability distribution that's defined by, in the case of neural networks, learned weights of a parametric model. Herding, on the other hand, analyzes a dataset.

However, FPCD also depends on a dataset[5]; WL is the only one of these algorithms that does not use a dataset. FPCD's dependence on a dataset means that it is a biased sampler, and the same applies to (Breuleux et al., 2011). FPCD's fast model runs a learning algorithm (roughly speaking PCD) with that dataset, and as a result, its "samples" will be biased towards that dataset[6]. Even if that biasing training data set would be equal to the model distribution, the pseudo-samples that FPCD would collect (with the "slow learning rate" set to zero) would still be biased, except in the limit of a vanishing "fast learning rate", but there the advantage of the algorithm is lost. FPCD's sampling method is inherently approximate; its main justification is that it seems to work in practice, despite being approximate.

The strengths and weaknesses of binning WL creates a smaller search space by binning: every bin of states in the original space becomes a state in the reduced space. Aversion is then applied to specific bins (Wang & Landau, 2001), or local regions of neighbouring bins (Zhou et al., 2006). If all goes well, the random walk will properly explore each bin, and the algorithm produces good estimates of the size of each bin.

However, if the bins are too large, the lack of aversion-based exploration within a bin becomes problematic. If the bins are too small, the random walk (which needs to visit them all) takes too much time.

Applying aversion to local regions of bins, instead of single bins, can improve exploration, as (Zhou et al., 2006) found. However, runtime remains at the very least linear in the number of bins.

Herding and FPCD can be thought of as applying aversion to quite a large area, instead of just one bin or a group of neighbouring bins, as pointed out by (Breuleux et al., 2011). For example, if the random walk visits the configuration where all units are on (in a binary RBM), every configuration in the whole search space will receive aversion proportional to the number of units and pairs of units that

[5] The dataset dependence of FPCD's fast model could perhaps be eliminated. We could replace the positive gradient estimate for the fast model by a slowly decaying historical average of the negative gradient estimates. The same could be tried with (Breuleux et al., 2011). However, this is only speculation; I have not attempted it in practice.

[6] A highly informal argument suggests that this might not be a problem in practice, just like CD-1 can produce a highly biased gradient estimate but still tend to move roughly in the right direction. The problem is that the "negative gradient" estimate is computed with pseudo-samples that are not quite samples from the model but are somewhat biased towards the training data. However, the "positive gradient" estimate will still be based on data that's even closer to the training set, namely the training set itself. Thus, the difference between the "positive gradient" estimate and the "negative gradient" estimate may still point in approximately the right direction.

are on in the configuration. Thus, Herding and FPCD avoid the need for binning to create a small space. FPCD pays for this by giving up on all interesting theoretical correctness guarantees.

Conclusion The approximate sampling component of the FPCD algorithm differs from Herding in that it's for sampling from a model with weights, albeit (slowly) evolving weights. It differs from the WL algorithm in that it aims for the standard Markov Chain histogram (instead of a flat histogram), in that its aversion effect is highly distributed, and in that it does not produce estimates of the size of designated chunks of the probability distribution. Thus, these three algorithms use the same principle of aversion-for-exploration in very different ways and to different ends.

3.3 Experiments: global unlearning vs. local unlearning

I start this section with its simple conclusion: experiments show clearly that for training an undirected model that is going to have to achieve results on its own (instead of being combined with other models), global unlearning is preferable to local unlearning.

What follows is the details of a series of experiments with different tasks, different image data sets, different undirected models, and different local and global unlearning algorithms, all of which point to the above conclusion.

3.3.1 General set-up of the experiments

For all experiments I used the mini-batch learning procedure, using 100 training cases for each gradient estimate.

The learning rates used in the experiments are not constant. In practice, decaying learning rates often work better. In these experiments, the learning rate was linearly decayed from some initial learning rate to zero, over the duration of the learning. Preliminary experiments showed that this often works better than the $\frac{1}{t}$ schedule suggested in (Robbins & Monro, 1951), which is preferable when infinitely much time is available for the optimization.

Some experiment parameters, such as the number of hidden units and the size of the mini-batches, were fixed. However, the initial learning rate was chosen using a validation set, as was weight decay for the (short) experiments on the MNIST patches and artificial data. For each algorithm, each task, and each training duration, 30 runs were performed with evaluation on validation data, trying to find the settings that worked best. Then a choice of initial learning rate and, for the shorter experiments, weight decay, were made, and with those chosen settings, 10 more runs were performed, evaluating on test data. This resulted in 10 test performance numbers, which were summarized by their average and standard deviation (shown as error bars).

The local unlearning algorithms that I used are several variations on the CD algorithm (CD-1, CD-10, and MF CD) as introduced in Section 1.3.2, and the pseudo-likelihood (PL) algorithm (Besag, 1986). The global unlearning algorithms used here are PCD and FPCD.

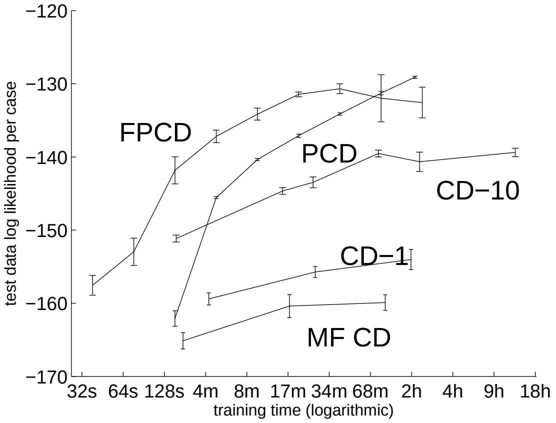

Figure 3.1: Modeling MNIST data with 25 hidden units (exact log likelihood)

3.3.2 Evaluating data likelihood exactly

Task

There are several ways of evaluating how well an RBM has been trained, but the most natural one is the data likelihood function. That is, after all, the objective function of which we attempt to find the gradient with a reasonable approximation. Data likelihood can be computed exactly for small models (up to about 25 hidden units), and can be estimated for larger models. For this first series of experiments I wanted to have exact evaluation, so I used an RBM with 25 binary hidden units.

Data

The data set for this task was the MNIST dataset of handwritten digit images (LeCun & Cortes, 1998). The images are 28 by 28 pixels, and the data set consists of 60,000 training cases and 10,000 test cases. To have a validation set, I split the official training set of 60,000 cases into a training set of 50,000 cases and a validation set of 10,000 cases. To have binary data, I treat the pixel intensities as probabilities. Each time a binary data point is required, a real-valued MNIST image is binarized by sampling from the given Bernoulli distribution for each pixel. Thus, in effect, the data set is a mixture of 70,000 factorial distributions: one for each of the data points in the MNIST data set. The visible layer of the RBM consists of 784 binary units.

Results

The results are shown in Figure 3.1, and are quite clear: if we're training a globally normalized model like an RBM, global unlearning is the way to go. Of the local unlearning algorithms, CD-10 performs best, probably because it is the least local in its unlearning.

Samples

We can also draw samples from a trained model, to get a feeling for what distribution it learned. This gives a subjective, inexact evaluation, but is nevertheless useful. In Figure 3.2, samples are shown from two different models: one trained with PCD (global unlearning), and one trained with CD-1 (local

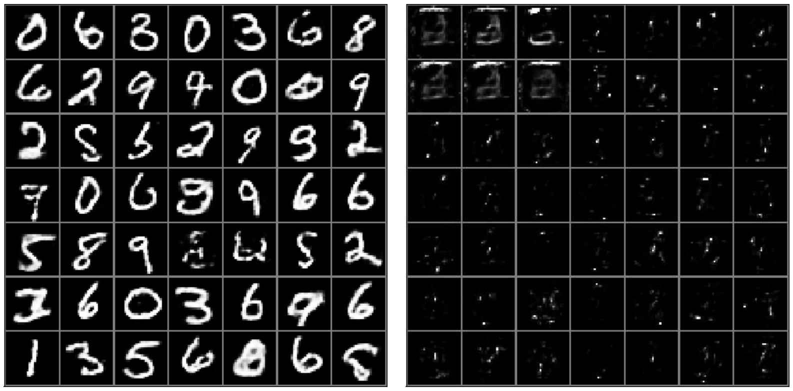

Figure 3.2: Samples from an RBM that was trained using PCD (left) and an RBM that was trained using CD-1 (right). Clearly, CD-1 did not produce an accurate model of the MNIST digits. Notice, however, that some of the CD-1 samples vaguely resemble a three.

unlearning). Both models had 500 binary hidden units. Again, the model that was trained with global unlearning is clearly doing a better job.

3.3.3 Using an RBM for classification

Another way to compare training algorithms is to look at what happens if we turn the generative model into a classification engine. This can be done with an RBM by adding a visible unit that represents the label. The result is called a classification RBM; see (Hinton et al., 2006; Larochelle & Bengio, 2008). The label unit is a multinomial unit, and its conditional probability distribution is computed by the softmax function. During training, the right value for this unit is included in the (labeled) training set. At test time, we perform classification by computing which of the possible classes has the greatest conditional probability, given the regular input values.

This model can be used for classification, and at the same time it builds a generative model of the input/label combinations in the training set. When one is only interested in classification, the fact that a generative model is also built has the pleasant effect of being a good regularizer. For these experiments, I only measure classification performance: the fraction of test cases that are correctly classified.

These experiments, too, were run on the MNIST data set.

The results are shown in Figure 3.3 and again show that global unlearning does a better job at this task than local unlearning.

3.3.4 A fully visible model

Task

The third model is significantly different: a fully visible, fully connected Markov Random Field (MRF) (see for example (Wainwright & Jordan, 2003)). One can use the PCD algorithm on it, although it looks a bit different in this case. To have exact test data log likelihood measurements, I used a small model, with only 25 units.

The data set for these experiments was obtained by taking small patches of 5 by 5 pixels, from the MNIST images. To have somewhat smooth-looking data, I binarized by thresholding at 1/2. The 70,000

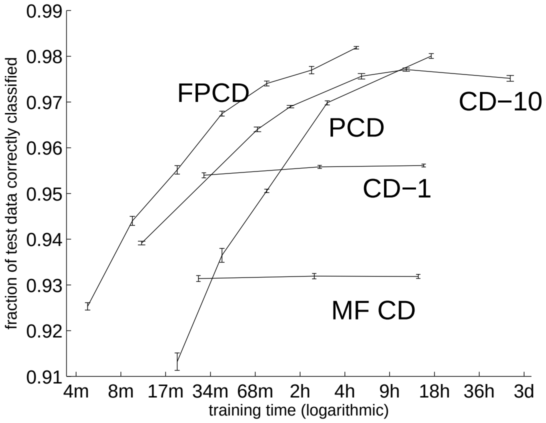

Figure 3.3: Classification of MNIST data

MNIST data points were thus turned into 70,000 times $(28 - 5 + 1)^2 = 4,032,000$ patches. This data set was split into training (60%), validation (20%), and test (20%) sets.

Algorithms

Three different training algorithms were compared:

- Pseudo-likelihood.
- Persistent Contrastive Divergence. PCD for fully visible MRFs is a bit different from PCD for RBMs. A pleasant difference is that, because there are no hidden units, $\frac{\partial \phi^+}{\partial \theta}$ is constant, so it can be precomputed for the entire training set. Thus, no variance results from the use of mini-batches, and the training set can be discarded after $\frac{\partial \phi^+}{\partial \theta}$ is computed over it. An unpleasant difference is that the Markov Chain defined by Gibbs sampling has slower mixing: MRFs with connections between the visible units lack the pleasant property of RBMs that all visible units can be updated at the same time (see, however, (Martens & Sutskever, 2010), which explores a way to overcome this weakness).
- Training by following the exact training data likelihood gradient. This is very slow, and just barely feasible for this toy model experiment, but gives a reasonable upper bound on how well training could work. It is not an exact upper bound, for a variety of reasons, but if runtime were not an issue, most researchers would prefer it to all algorithms that are used in practice.

Results

Pseudo-Likelihood (PL) training works reasonably well on this data set, but it does not produce the best probability models. This is probably because PL optimizes a different objective function. As a result, PL needed early stopping to avoid departing too much from the data likelihood objective function: the optimal learning rate was always more or less inversely proportional to the duration of the optimization. Even with only a few seconds of training time, the best test data likelihood is already achieved: -5.35 (nats per data case).

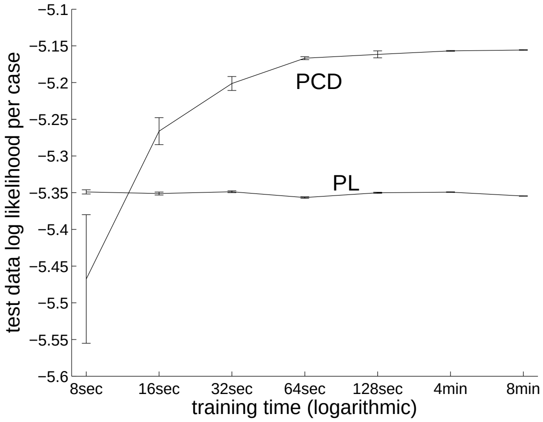

Figure 3.4: Training a fully visible MRF

PCD training does go more in the direction of the data likelihood function - in the limit of the learning rate going to zero, it gives its exact gradient. Thus, PCD did profit from having more time to run. Figure 3.4 shows the performance. The asymptotic test data log likelihood for PCD is -5.15.

Training based on the exact training data likelihood gradient (slow but possible to compute) also ended up with a test data log likelihood of -5.15, supporting the idea that in the limit of enough training time, PCD has the same result as this almost best-possible algorithm. It just takes less time in practice.

This value of -5.15 "nats" per case is, however, quite different from the entropy of the training data distribution, which is approximately 4.78 nats[7]. This difference is likely due to the fact that the model has insufficient complexity to completely learn the training data distribution.

As a side-note, the training data log likelihood is only 0.004 nats better than the test data log likelihood. This small difference is not a great surprise, because the data set is large and the model is small.

3.3.5 Evaluating data likelihood under a full-size model

Approximating likelihood

Comparing data likelihood under toy models is important, and nice because one has exact results, but is not entirely satisfactory. However, likelihood under a full-size model is intractable because it requires one to compute the normalization constant Z of the RBM. To still have some sort of comparison, I used a procedure for estimating the normalization constant Z, that was developed in our group (Salakhutdinov & Murray, 2008). The algorithm works for any number of hidden units, but it comes with few guarantees in practice. Nonetheless, it can be used to complement other results.

Data

For these experiments, an artificial data set was created by combining the outlines of rectangles and triangles in an image. Because this data set is artificially generated, there is, in practice, an unlimited

[7] This estimate was obtained by treating the dataset of 4,032,000 cases as a mixture of 4,032,000 delta distributions. There is great overlap: some input patterns appear many times. Therefore, calculating entropy this way has some justification. However, it is inevitably an underestimate of the number that would be found if MNIST had had more cases.

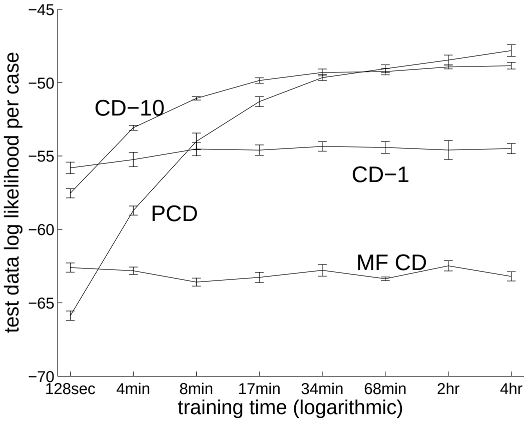

Figure 3.5: Modeling artificial data. Shown on the y axis is approximate test data log likelihood.

amount of it, which has the benefit of shedding some light on the reasons for using weight decay.

Model

The model for this task was a fully binary RBM with 100 hidden units.

Results

In Figure 3.5 we see essentially the same as what happened on the MNIST tasks. MF CD is clearly the worst of the algorithms, CD-1 works better, and CD-10 and PCD work best, with CD-10 being preferable when little time is available and PCD being better if more time is available.

Weight decay

This data set was artificially generated, so there was an infinite amount of data available. Thus, one might think that the use of weight decay serves no purpose. However, all four algorithms did work best with some weight decay. One reason for this is probably that weight decay regularization can help to keep the model somewhat reasonable, which can be important when it's trained with a poor gradient approximation. Another reason is that all of these algorithms are somewhat dependent on the mixing rate of the Markov Chain defined by the Gibbs sampler, and that mixing rate is usually higher when the parameters of the model are smaller (if the parameters would be simply scaled down, the mixing rate is guaranteed to be higher). Thus, weight decay keeps the model mixing reasonably well, and makes all of these algorithms work better.

- MF CD, which is in a sense the most approximate algorithm of them all, needs most weight decay, probably to keep its oddities under control. MF CD worked best with a weight decay strength of 10^{-3}.
- CD-1 does introduce some noise in the update procedure, which itself has a regularizing effect (Hinton et al., 2012). As a result it required less weight decay: $3 \cdot 10^{-4}$.
- CD-10 performs more updates, and can therefore still do somewhat of a decent job when mixing is poorer. The best weight decay value for CD-10 turned out to be approximately $1.3 \cdot 10^{-4}$.

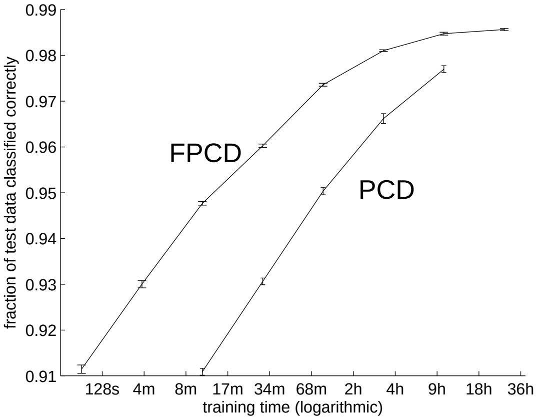

Figure 3.6: Classifying MNIST data (the number of hidden units is chosen using validation data).

- Finally, the mixing mechanism used by PCD is even better, but it is still based on the Gibbs sampler, so it, too, works better with some weight decay. The best weight decay strength for PCD was approximately $2.5 \cdot 10^{-5}$.

3.4 Experiments: FPCD vs. PCD

This section presents some comparisons of the performance of PCD and FPCD. Both use global unlearning, and they are similar in spirit, but FPCD was designed to be faster by allowing a large “fast learning rate” which ensures that the fantasy particles rapidly explore the state space. The general impression is indeed that FPCD is faster, but otherwise quite similar to PCD. The greatest difference in learning speed is observed when the amount of training time is small, relative to the size of the model. This suggests that FPCD should be a particularly good choice for training large models, where one expects training time to be a major bottleneck.

For some other experiments that investigate what meta-parameter settings work well for FPCD, see the paper that introduced it (Tieleman & Hinton, 2009).

3.4.1 Comparisons with fixed model architecture

Figure 3.1 and Figure 3.3 show FPCD generally outperforming PCD, though most clearly so with little runtime and a large model. Training the toy model for two hours appears not to be such a situation: there, the FPCD performance becomes a bit less predictable.

3.4.2 Comparison with different architectures

This task was again classification of the MNIST images, but for these experiments, the number of hidden units was not fixed, but was chosen using the validation set. The initial learning rate was also chosen using the validation set. The additional FPCD parameters were set using the heuristically chosen values mentioned in (Tieleman & Hinton, 2009). The result is displayed in Figure 3.6.

As was observed on the smaller tasks, FPCD takes, on average, about one quarter of the time required by PCD to get the same performance.

It is interesting to see what number of hidden units worked best for PCD versus FPCD. Given the same amount of training time, more hidden units means fewer parameter updates - the product of these two numbers has to be approximately constant. It turned out that for FPCD, the optimal number of hidden units is significantly larger than for PCD. For example, given 9 hours of training time, FPCD worked best with about 1200 hidden units, while PCD worked best with about 700 hidden units. This ratio of close to 2 was fairly consistent for different amounts of training time. The most natural explanation is that FPCD has the greatest advantage when fewer parameter updates can be done. That is why FPCD works best with a more powerful model, even though that means fewer parameter updates. PCD needs many parameter updates, and thus cannot afford many hidden units. Again, this supports the hypothesis that FPCD is particularly advantageous for large tasks, where the amount of training time is a bottleneck.

3.5 Survey of models that use FPCD

FPCD has been used in a variety of projects, most of them models of images.

(Ranzato et al., 2010; Ranzato et al., 2011) describe a sophisticated and effective model of images in their raw representation: pixel intensity arrays. It uses two types of hidden units: units that affect only the mean of the image pixels and units that affect only the covariance of the image pixels. Both types can be marginalized out to produce an unnormalized probability of an image vector. Hybrid Monte Carlo on this proposed image vector is then used as the transition operator[8]. Their model is very stiff, so FPCD was needed to make the "negative phase" exploration work well. Ranzato mentioned that FPCD was the only training method that produced models from which he could sample well (personal communication, 2011). (Kivinen & Williams, 2012) uses a variation on this model for learning models of regular texture images. It adds a few units that are used to switch between different textures, in the same model.

Another approach is presented by (Ngiam et al., 2011). It, too, uses FPCD with a Hybrid Monte Carlo transition operator, but on a simple Restricted Boltzmann Machine. The model also uses image encoding units, but they are not part of the RBM.

The spike-and-slab RBM (Courville et al., 2011b) has also been successfully trained using FPCD (Luo et al., 2012). Again, FPCD's strength was most visible when dealing with stiff models.

The failings of FPCD have also been investigated. (Fischer & Igel, 2010) echoes the conclusion of Section 3.3.5 by highlighting the need for regularization, even for powerful methods like FPCD. (Desjardins et al., 2010) points out the inherently approximate nature of the sampling in FPCD: the weights of the fast model can be quite different from those of the regular model (even though only temporarily so), and while this allows good exploration, it can also lead to "negative particle" states that are not representative of the training data and presumably not of the model distribution. (Salakhutdinov, 2010) voices the same concern.

3.6 Conclusion

Creating a good generative model of images is not easy. Generating images of handwritten symbols of just 10 classes, monochrome, with most pixels being saturated, in images of 28x28, is doable, but is not

[8] This is in contrast to my own experiments, which use Gibbs sampling and do not integrate out the hidden units.

exactly the ultimate goal. High resolution colour images of many different types of scenes, with many objects in them and many different lighting conditions, is quite a step up. Despite some exciting progress (Ranzato et al., 2010; Courville et al., 2011a), at this time that task is still beyond what we can do well by just learning.

One of the problems in creating a model of such data is that only a tiny fraction of the state space deserves to have any significant probability. In practice, that means that we're usually stuck with either a poor model, or a model with Markov Chains that mix extremely slowly. It has to be a very stiff model, allowing its variables little freedom, and such a thing is never good for mixing rates (Bengio et al., 2013a).

FPCD seems like it is still one of the best tools that we have for this task, today. With a large "fast learning rate", its rapidly changing exploratory energy function can quickly cover great distances in the state space, without destroying anything permanently because the "slow model" is not changing much at all. (Ranzato et al., 2010) uses FPCD on such a stiff model, with decent results.

However, there is an alternative method. Layer-by-layer training of deep models, as is done in (Hinton et al., 2006), may be better suited for this task after all. The lower layers can learn to re-encode the data in a space where the restrictions are less rigid, where the correlations are weaker. Then, the higher layers will have a far easier job (Bengio et al., 2013a). In this set-up, the lower layers do not have to build a generative model of the data; only the top layer has to do so. In this situation, CD-1 has shown itself to be an excellent training algorithm for the lower layers, although the top layer still has to be trained with a global unlearning algorithm. CD-1, being much like autoencoder training, is good at preserving a lot of information in the hidden units. Preserving information is not important for the top layer, but it is essential in the lower layers.

In Section 3.3.5 I reported that weight decay tends to help even when overfitting is not a possibility. This adds to the suspicion that no matter how good the exploration strategy of an algorithm is, stiff models with hard-to-cross energy barriers are still going to cause problems. That strengthens the argument for first transforming the data space in a learned way, using greedy layerwise training of a deep architecture, before attempting to model it.

In conclusion, I must therefore make very clear that my experiments have only shown global unlearning to be superior for tasks where the model will have to perform well *on its own.* If a model has a job to do in conjunction with other models, as happens in greedy section-by-section training algorithms, the situation is more complicated, and such set-ups may be the best hope for a generative model of realistic images.

Chapter 4

Appendix: detailed derivations

4.1 Rewriting the RBM training objective function

This section details the rewriting of $\nabla\phi$ for energy-based models as $\nabla\phi = \nabla(KL(R||P_\theta) - KL(D||P_\theta))$. Here, R is equal to P_θ, for the θ value at which we seek $\nabla\phi = \nabla\phi(\theta)$. R is a constant: it does not change when θ changes, i.e. $\nabla R \equiv \nabla_\theta R = 0$.

D stands for the training data distribution.

4.1.1 Informal derivation

An elegant but slightly informal way to show that $\nabla\phi = \nabla(KL(R||P_\theta) - KL(D||P_\theta))$ is as follows.

Clearly, at the given value for θ, $KL(R||P_\theta) = 0$. When θ changes, P_θ will change but R will not, so we can write $KL(R||P_\theta)$ as a function of θ: $f(\theta) \equiv KL(R||P_\theta)$. It is a differentiable function, and its minimum is at the given θ: there, $f = 0$, and everywhere else, $f \geq 0$. Because θ is a minimum of f, $\nabla f = 0$ at θ. Therefore, $\nabla KL(R||P_\theta) = 0$, so the claim $\nabla\phi = \nabla(KL(R||P_\theta) - KL(D||P_\theta))$ can be simplified to $\nabla\phi = -KL(D||P_\theta))$.

Because $\phi(\theta) = \mathop{\mathrm{E}}_{\vec{x}\sim D}[\log P_\theta(\vec{x})]$ and $KL(D||P_\theta) = \mathop{\mathrm{E}}_{\vec{x}\sim D}[\log D(\vec{x}) - \log P_\theta(\vec{x})] = -H(D) + \mathop{\mathrm{E}}_{\vec{x}\sim D}[-\log P_\theta(\vec{x})]$ and $\nabla - H(D) = 0$, the claimed equality is clearly true.

4.1.2 More formal derivation

A more rigorous derivation is as follows.

Step one is to show that $\nabla KL(R||P_\theta) = 0$, in a more formal way.

$$
\begin{aligned}
& \nabla KL(R||P_\theta) && \\
&= \nabla \mathop{\mathrm{E}}_{\vec{x}\sim R}[\log R(\vec{x}) - \log P_\theta(\vec{x})] && \text{By the definition of KL} \\
&= \nabla(-H(R)) - \nabla \mathop{\mathrm{E}}_{\vec{x}\sim R}[\log P_\theta(\vec{x})] && \text{Splitting the differentiation} \\
&= -\nabla \mathop{\mathrm{E}}_{\vec{x}\sim R}[\log P_\theta(\vec{x})] && R \text{ is constant, w.r.t. } \theta \\
&= - \mathop{\mathrm{E}}_{\vec{x}\sim R}[\nabla \log P_\theta(\vec{x})] && R \text{ is constant, w.r.t. } \theta \\
&= - \mathop{\mathrm{E}}_{\vec{x}\sim R}\left[\frac{1}{P_\theta(\vec{x})} \nabla P_\theta(\vec{x})\right] && \text{Derivative rule for logarithms} \\
&= - \mathop{\mathrm{E}}_{\vec{x}\sim P_\theta}\left[\frac{1}{P_\theta(\vec{x})} \nabla P_\theta(\vec{x})\right] && R = P_\theta\text{, when we're not taking its derivative} \\
&= - \sum_{\vec{x}} P_\theta(\vec{x}) \frac{1}{P_\theta(\vec{x})} \nabla P_\theta(\vec{x}) && \text{Definition of discrete expectation} \\
&= - \sum_{\vec{x}} \nabla P_\theta(\vec{x}) && \text{Cancellation} \\
&= -\nabla \sum_{\vec{x}} P_\theta(\vec{x}) && \text{Exchange differentiation and finite summation} \\
&= -\nabla 1 && \text{Probability distributions sum to 1} \\
&= 0 &&
\end{aligned}
$$

Now it remains to be shown that $\nabla\phi = \nabla(-KL(D||P_\theta))$, which is a shorter derivation:

$$
\begin{aligned}
& \nabla(-KL(D||P_\theta)) && \\
&= \nabla(- \mathop{\mathrm{E}}_{\vec{x}\sim D}[\log D(\vec{x}) - \log P_\theta(\vec{x})] && \text{By the definition of KL} \\
&= \nabla(H(D) + \mathop{\mathrm{E}}_{\vec{x}\sim D}[\log P_\theta(\vec{x})] && \text{Splitting the expectation} \\
&= \nabla H(D) + \nabla \mathop{\mathrm{E}}_{\vec{x}\sim D}[\log P_\theta(\vec{x})] && \text{Splitting the differentiation} \\
&= \nabla \mathop{\mathrm{E}}_{\vec{x}\sim D}[\log P_\theta(\vec{x})] && D \text{ does not depend on } \theta \\
&= \nabla\phi && \text{By the definition of } \phi
\end{aligned}
$$

Those two derivations together show that $\nabla\phi = \nabla(KL(R||P_\theta) - KL(D||P_\theta))$.

4.2 The positive gradient for RBMs

This section describes the *positive* part of the gradient for RBMs: first for general energy-based models, and second specifically for binary RBMs.

4.2.1 For general energy-based models

ϕ^+ is defined as $\phi^+(\theta) = \mathop{\mathrm{E}}_{\vec{v}\sim D}\left[\log \sum_{\vec{h}} \exp(-E_\theta(\vec{v},\vec{h}))\right]$. Below is the derivation for the gradient of ϕ^+ w.r.t. θ. It's a fairly straightforward application of standard differentiation rules:

$$\begin{aligned}
\nabla\phi^+ &= \nabla \mathop{\mathrm{E}}_{\vec{v}\sim D}\left[\log \sum_{\vec{h}} \exp(-E_\theta(\vec{v},\vec{h}))\right] && \text{By the definition of } \phi^+ \\
&= \mathop{\mathrm{E}}_{\vec{v}\sim D}\left[\nabla \log \sum_{\vec{h}} \exp(-E_\theta(\vec{v},\vec{h}))\right] && \text{Rearranging} \\
&= \mathop{\mathrm{E}}_{\vec{v}\sim D}\left[\frac{\nabla \sum_{\vec{h}} \exp(-E_\theta(\vec{v},\vec{h}))}{\sum_{\vec{h}} \exp(-E_\theta(\vec{v},\vec{h}))}\right] && \text{Derivative of log} \\
&= \mathop{\mathrm{E}}_{\vec{v}\sim D}\left[\frac{\sum_{\vec{h}} \nabla \exp(-E_\theta(\vec{v},\vec{h}))}{\sum_{\vec{h}} \exp(-E_\theta(\vec{v},\vec{h}))}\right] && \text{Rearranging} \\
&= \mathop{\mathrm{E}}_{\vec{v}\sim D}\left[\frac{\sum_{\vec{h}} \exp(-E_\theta(\vec{v},\vec{h}))\nabla(-E_\theta(\vec{v},\vec{h}))}{\sum_{\vec{h}} \exp(-E_\theta(\vec{v},\vec{h}))}\right] && \text{Derivative of exp} \\
&= \mathop{\mathrm{E}}_{\vec{v}\sim D}\left[\frac{\sum_{\vec{h}} \exp(-E_\theta(\vec{v},\vec{h}))/Z_\theta \nabla(-E_\theta(\vec{v},\vec{h}))}{\sum_{\vec{h}} \exp(-E_\theta(\vec{v},\vec{h}))/Z_\theta}\right] && \text{Divide by } \frac{Z_\theta}{Z_\theta} \\
&= \mathop{\mathrm{E}}_{\vec{v}\sim D}\left[\frac{\sum_{\vec{h}} P_\theta(\vec{v},\vec{h})\nabla(-E_\theta(\vec{v},\vec{h}))}{\sum_{\vec{h}} \exp(-E_\theta(\vec{v},\vec{h}))/Z_\theta}\right] && \text{By definition of } P_\theta(\vec{v},\vec{h}) \\
&= \mathop{\mathrm{E}}_{\vec{v}\sim D}\left[\frac{\sum_{\vec{h}} P_\theta(\vec{v},\vec{h})\nabla(-E_\theta(\vec{v},\vec{h}))}{P_\theta(\vec{v})}\right] && \text{By definition of } P_\theta(\vec{v}) \\
&= \mathop{\mathrm{E}}_{\vec{v}\sim D}\left[\sum_{\vec{h}} \frac{P_\theta(\vec{v},\vec{h})}{P_\theta(\vec{v})}\nabla(-E_\theta(\vec{v},\vec{h}))\right] && \text{Rearranging} \\
&= \mathop{\mathrm{E}}_{\vec{v}\sim D}\left[\sum_{\vec{h}} P_\theta(\vec{h}|\vec{v})\nabla(-E_\theta(\vec{v},\vec{h}))\right] && \text{Conditional probability} \\
&= \mathop{\mathrm{E}}_{\vec{v}\sim D}\left[\mathop{\mathrm{E}}_{\vec{h}\sim P_\theta(\cdot|\vec{v})}\left[\nabla(-E_\theta(\vec{v},\vec{h}))\right]\right] && \text{Discrete expectation}
\end{aligned}$$

4.2.2 For binary RBMs

For a binary RBM, the above simplifies to $\frac{\partial \phi^+}{\partial w_{ij}} = \underset{\vec{v} \sim D}{\mathrm{E}} [v_i \cdot P_\theta(h_j = 1|\vec{v})]$, as shown below. The key insight is that $\frac{\partial - E_\theta(\vec{v},\vec{h})}{\partial w_{ij}} = [v_i = 1 \wedge h_j = 1]$, by the definition of the energy for a binary RBM.

$$\begin{aligned}
\frac{\partial \phi^+}{\partial w_{ij}} &= \underset{\vec{v} \sim D}{\mathrm{E}} \left[\underset{\vec{h} \sim P_\theta(\cdot|\vec{v})}{\mathrm{E}} \left[\frac{\partial - E_\theta(\vec{v},\vec{h})}{\partial w_{ij}} \right] \right] && \text{See previous section} \\
&= \underset{\vec{v} \sim D}{\mathrm{E}} \left[\underset{\vec{h} \sim P_\theta(\cdot|\vec{v})}{\mathrm{E}} [v_i = 1 \wedge h_j = 1] \right] && \text{By the definition of } E_\theta \\
&= \underset{\vec{v} \sim D}{\mathrm{E}} [P_\theta(v_i = 1 \wedge h_j = 1|\vec{v})] && \text{Expectation of an indicator variable} \\
&= \underset{\vec{v} \sim D}{\mathrm{E}} [v_i \cdot P_\theta(h_j = 1|\vec{v})] && v_i \text{ is binary}
\end{aligned}$$

Following a very similar line of reasoning, one finds that $\frac{\partial \phi^+}{\partial b_i} = P_{\vec{v} \sim D}(v_i = 1)$ and $\frac{\partial \phi^+}{\partial a_j} = E_{\vec{v} \sim D} [P_\theta(h_j = 1|\vec{v})]$.

Informally, the conclusion is that increasing ϕ^+ corresponds to strengthening the connections between pairs (v_i, h_j) that are often both on, with $\vec{v} \sim D$.

The above can be easily computed as a sum over the training cases, because $P_\theta(h_j = 1|\vec{v})$ is tractable: it is simply $P_\theta(h_j = 1|\vec{v}) = \sigma(b_j + \sum_i v_i \cdot w_{ij})$, where $\sigma(x) \equiv \frac{1}{1+\exp(-x)}$.

4.3 The negative gradient for RBMs

The derivation of the *negative* part of the gradient is a bit simpler but relies on the same ideas as the *positive* part.

4.3.1 For general energy-based models

ϕ^- is defined as $\phi^-(\theta) = \log Z_\theta = \log \sum_{\vec{v},\vec{h}} \exp(-E_\theta(\vec{v},\vec{h}))$. This section shows how the gradient w.r.t. θ comes out as $\nabla \phi^- = \underset{\vec{v},\vec{h} \sim P_\theta}{\mathrm{E}} \left[\nabla(-E_\theta(\vec{v},\vec{h})) \right]$.

$$
\begin{aligned}
\nabla\phi^- &= \nabla \log Z_\theta && \text{By the definition of } \phi^- \\
\nabla\phi^- &= \frac{1}{Z_\theta}\nabla Z_\theta && \text{Derivative of } \log \\
\nabla\phi^- &= \frac{1}{Z_\theta}\nabla \sum_{\vec{v},\vec{h}} \exp(-E_\theta(\vec{v},\vec{h})) && \text{By the definition of } Z_\theta \\
\nabla\phi^- &= \frac{1}{Z_\theta}\sum_{\vec{v},\vec{h}} \nabla \exp(-E_\theta(\vec{v},\vec{h})) && \text{Rearranging} \\
\nabla\phi^- &= \frac{1}{Z_\theta}\sum_{\vec{v},\vec{h}} \exp(-E_\theta(\vec{v},\vec{h}))\nabla(-E_\theta(\vec{v},\vec{h})) && \text{Derivative of } \exp \\
\nabla\phi^- &= \sum_{\vec{v},\vec{h}} \frac{\exp(-E_\theta(\vec{v},\vec{h}))}{Z_\theta}\nabla(-E_\theta(\vec{v},\vec{h})) && \text{Rearranging} \\
\nabla\phi^- &= \sum_{\vec{v},\vec{h}} P_\theta(\vec{v},\vec{h})\nabla(-E_\theta(\vec{v},\vec{h})) && \text{By the definition of } P_\theta(\vec{v},\vec{h}) \\
\nabla\phi^- &= \mathop{\mathrm{E}}_{\vec{v},\vec{h}\sim P_\theta}\left[\nabla(-E_\theta(\vec{v},\vec{h}))\right] && \text{By the definition discrete expectation}
\end{aligned}
$$

Informally, the conclusion is that increasing ϕ^- corresponds to decreasing $E_\theta(\vec{v},\vec{h})$ for $(\vec{v},\vec{h})$ pairs that have great probability.

4.3.2 For binary RBMs

In a binary RBM, $\nabla\phi^- = \mathop{\mathrm{E}}_{\vec{v},\vec{h}\sim P_\theta}\left[\nabla(-E_\theta(\vec{v},\vec{h}))\right]$ comes out as $\frac{\partial\phi^-}{\partial w_{ij}} = P_\theta(v_i = 1, h_j = 1)$.

We start with $\frac{\partial\phi^-}{\partial w_{ij}}$ from the general form: $\frac{\partial\phi^-}{\partial w_{ij}} = \mathop{\mathrm{E}}_{\vec{v},\vec{h}\sim P_\theta}\left[\frac{\partial - E_\theta(\vec{v},\vec{h})}{\partial w_{ij}}\right]$.

Because of the definition of $E_\theta(\vec{v},\vec{h})$ for a binary RBM, $\frac{\partial - E_\theta(\vec{v},\vec{h})}{\partial w_{ij}} = [v_i = 1 \wedge h_j = 1]$, i.e. that derivative is 1 if visible unit #i and hidden unit #j are both on, and 0 otherwise. We can proceed as follows:

$$
\begin{aligned}
\frac{\partial\phi^-}{\partial w_{ij}} &= \mathop{\mathrm{E}}_{\vec{v},\vec{h}\sim P_\theta}\left[\frac{\partial - E_\theta(\vec{v},\vec{h})}{\partial w_{ij}}\right] && \text{This is the general form} \\
&= \mathop{\mathrm{E}}_{\vec{v},\vec{h}\sim P_\theta}[v_i = 1 \wedge h_j = 1] && \text{From the definition of } E_\theta \\
&= P_\theta(v_i = 1 \wedge h_j = 1) && \text{Expectation of an indicator variable}
\end{aligned}
$$

Essentially the same argument leads to the conclusion that $\frac{\partial\phi^-}{\partial b_i} = P_\theta(v_i = 1)$ and $\frac{\partial\phi^-}{\partial a_j} = P_\theta(h_j = 1)$.

Bibliography

Abdel-Hamid, O., Mohamed, A.-r., Jiang, H., & Penn, G. (2012). Applying convolutional neural networks concepts to hybrid nn-hmm model for speech recognition. *Acoustics, Speech and Signal Processing (ICASSP), 2012 IEEE International Conference on* (pp. 4277–4280).

Allender, E. (1996). Circuit complexity before the dawn of the new millennium. *Foundations of Software Technology and Theoretical Computer Science* (pp. 1–18).

Bengio, Y. (2009). *Learning deep architectures for AI.* Now Publishers Inc.

Bengio, Y. (2013). Deep learning of representations: Looking forward. In *Statistical language and speech processing*, 1–37. Springer.

Bengio, Y., & Delalleau, O. (2007). *Justifying and generalizing contrastive divergence* (Technical Report 1311). Université de Montréal.

Bengio, Y., Lamblin, P., Popovici, D., & Larochelle, H. (2007). Greedy layer-wise training of deep networks. *Advances in Neural Information Processing Systems 19: Proceedings of the 2006 Conference* (p. 153).

Bengio, Y., & LeCun, Y. (2007). Scaling learning algorithms towards AI. *Large-Scale Kernel Machines.*

Bengio, Y., Mesnil, G., Dauphin, Y., & Rifai, S. (2013a). Better mixing via deep representations. .

Bengio, Y., Simard, P., & Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, *5*, 157–166.

Bengio, Y., & Thibodeau-Laufer, É. (2013). Deep generative stochastic networks trainable by backprop. *arXiv preprint arXiv:1306.1091.*

Bengio, Y., Yao, L., Alain, G., & Vincent, P. (2013b). Generalized denoising auto-encoders as generative models. *Advances in Neural Information Processing Systems* (pp. 899–907).

Bergstra, J., Bardenet, R., Bengio, Y., Kégl, B., et al. (2011). Algorithms for hyper-parameter optimization. *NIPS* (pp. 2546–2554).

Bergstra, J., & Bengio, Y. (2012). Random search for hyper-parameter optimization. *The Journal of Machine Learning Research*, *13*, 281–305.

Bergstra, J., Breuleux, O., Bastien, F., Lamblin, P., Pascanu, R., Desjardins, G., Turian, J., Warde-Farley, D., & Bengio, Y. (2010). Theano: a CPU and GPU math expression compiler. *Proceedings of the Python for Scientific Computing Conference (SciPy).* Austin, TX. Oral Presentation.

Besag, J. (1986). On the statistical analysis of dirty pictures. *Journal of the Royal Statistical Society B*, *48*, 259–302.

Bharath, B., & Borkar, V. S. (1999). Stochastic approximation algorithms: Overview and recent trends. *Sadhana*, *24*, 425–452.

Breuleux, O., Bengio, Y., & Vincent, P. (2010). Unlearning for better mixing. *Universite de Montreal/DIRO*.

Breuleux, O., Bengio, Y., & Vincent, P. (2011). Quickly generating representative samples from an rbm-derived process. *Neural computation*, *23*, 2058–2073.

Bruna, J., & Mallat, S. (2010). Classification with scattering operators. *arXiv preprint arXiv:1011.3023*.

Ciresan, D. C., Meier, U., Gambardella, L. M., & Schmidhuber, J. (2010). Deep, big, simple neural nets for handwritten digit recognition. *Neural computation*, *22*, 3207–3220.

Cootes, T. F., Edwards, G. J., Taylor, C. J., et al. (2001). Active appearance models. *IEEE Transactions on pattern analysis and machine intelligence*, *23*, 681–685.

Cortes, C., & Vapnik, V. (1995). Support-vector networks. *Machine learning*, *20*, 273–297.

Courville, A., Bergstra, J., & Bengio, Y. (2011a). Unsupervised models of images by spike-and-slab RBMs. .

Courville, A. C., Bergstra, J., & Bengio, Y. (2011b). A spike and slab restricted boltzmann machine. *International Conference on Artificial Intelligence and Statistics* (pp. 233–241).

Decoste, D., & Schoelkopf, B. (2002). Training invariant support vector machines. *Machine Learning*, *46*, 161–190.

Deselaers, T., Hasan, S., Bender, O., & Ney, H. (2009). A deep learning approach to machine transliteration. *Proceedings of the Fourth Workshop on Statistical Machine Translation* (pp. 233–241).

Desjardins, G., Courville, A. C., Bengio, Y., Vincent, P., & Delalleau, O. (2010). Tempered markov chain monte carlo for training of restricted boltzmann machines. *International Conference on Artificial Intelligence and Statistics* (pp. 145–152).

Erhan, D., Manzagol, P. A., Bengio, Y., Bengio, S., & Vincent, P. (2009). The difficulty of training deep architectures and the effect of unsupervised pre-training. *Proceedings of The Twelfth International Conference on Artificial Intelligence and Statistics (AISTATS 09)* (pp. 153–160).

Fei-Fei, L., Fergus, R., & Perona, P. (2004). Learning generative visual models from few training examples: an incremental Bayesian approach tested on 101 object categories. IEEE. CVPR 2004. *Workshop on Generative-Model Based Vision*.

Fischer, A., & Igel, C. (2010). Empirical analysis of the divergence of gibbs sampling based learning algorithms for restricted boltzmann machines. In *Artificial neural networks–icann 2010*, 208–217. Springer.

Frey, B. J., & Jojic, N. (2000). Transformation-invariant clustering and dimensionality reduction using em. *IEEE Transactions on Pattern Analysis and Machine Intelligence*.

Gregor, K., & LeCun, Y. (2010). Emergence of Complex-Like Cells in a Temporal Product Network with Local Receptive Fields. *Arxiv preprint arXiv:1006.0448*.

Halle, M., & Stevens, K. (1962). Speech recognition: A model and a program for research. *Information Theory, IRE Transactions on*, *8*, 155–159.

Halle, M., & Stevens, K. N. (1959). Analysis by synthesis. *Proc. Seminar on Speech Compression and Processing*.

Hinton, G. (1987). Learning translation invariant recognition in a massively parallel networks. *PARLE Parallel Architectures and Languages Europe* (pp. 1–13).

Hinton, G. (2012). Lecture 6.5: rmsprop: divide the gradient by a running average of its recent magnitude. *Coursera: Neural Networks for Machine Learning*.

Hinton, G., Krizhevsky, A., & Wang, S. (2011). Transforming auto-encoders. *Artificial Neural Networks and Machine Learning–ICANN 2011*, 44–51.

Hinton, G., & Salakhutdinov, R. (2006). Reducing the Dimensionality of Data with Neural Networks. *Science*, *313*, 504–507.

Hinton, G. E. (2002). Training Products of Experts by Minimizing Contrastive Divergence. *Neural Computation*, *14*, 1771–1800.

Hinton, G. E. (2006). *To Recognize Shapes, First Learn to Generate Images* (Technical Report UTML TR 2006-004). University of Toronto, Department of Computer Science.

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

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

Hinton, G. E., & Sejnowski, T. J. (1986). Learning and relearning in Boltzmann machines. *Mit Press Computational Models Of Cognition And Perception Series*, 282–317.

Hinton, G. E., Srivastava, N., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. R. (2012). Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*.

Hyvärinen, A., & Dayan, P. (2005). Estimation of non-normalized statistical models by score matching. *Journal of Machine Learning Research*, *6*.

Jaitly, N., & Hinton, G. E. (2013). Using an autoencoder with deformable templates to discover features for automated speech recognition.

Kivinen, J. J., & Williams, C. (2012). Multiple texture boltzmann machines. *International Conference on Artificial Intelligence and Statistics* (pp. 638–646).

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). ImageNet classification with deep convolutional neural networks. *NIPS* (p. 4).

Kushner, H., & Clark, D. (1978). Stochastic approximation methods for constrained and unconstrained systems.

Lang, K. J., Waibel, A. H., & Hinton, G. E. (1990). A time-delay neural network architecture for isolated word recognition. *Neural networks*, *3*, 23–43.

Larochelle, H., & Bengio, Y. (2008). Classification using discriminative Restricted Boltzmann Machines. *Proceedings of the 25th international conference on Machine learning* (pp. 536–543).

Larochelle, H., Bengio, Y., & Turian, J. (2010). Tractable multivariate binary density estimation and the Restricted Boltzmann Forest.

Larochelle, H., & Hinton, G. E. (2010). Learning to combine foveal glimpses with a third-order boltzmann machine. *Advances in neural information processing systems* (pp. 1243–1251).

Le Roux, N., & Bengio, Y. (2008). Representational power of restricted boltzmann machines and deep belief networks. *Neural Computation*, *20*, 1631–1649.

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

LeCun, Y., & Cortes, C. (1998). The MNIST database of handwritten digits. `http://yann.lecun.com/exdb/mnist/`.

LeCun, Y., Huang, F. J., & Bottou, L. (2004). Learning methods for generic object recognition with invariance to pose and lighting. *Proceedings of the 2004 IEEE Computer Society Conference on Computer Vision and Pattern Recognition* (pp. 97–104).

Lee, H., Grosse, R., Ranganath, R., & Ng, A. Y. (2009). Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations. *Proceedings of the 26th Annual International Conference on Machine Learning* (pp. 609–616).

Luo, H., Carrier, P. L., Courville, A., & Bengio, Y. (2012). Texture modeling with convolutional spike-and-slab rbms and deep extensions. *arXiv preprint arXiv:1211.5687*.

Martens, J. (2010). Deep learning via Hessian-free optimization. *Proceedings of the 27th international conference on Machine learning*.

Martens, J., & Sutskever, I. (2010). Parallelizable sampling of markov random fields. *International Conference on Artificial Intelligence and Statistics* (pp. 517–524).

Minsky, M. L., & Papert, S. (1969). *Perceptrons: An introduction to computational geometry*. MIT Press.

Mnih, V. (2009). *CUDAMat: A CUDA-based matrix class for Python* (Technical Report UTML TR 2009-004). University of Toronto, Department of Computer Science.

Mohamed, A., Dahl, G., & Hinton, G. (2009). Deep Belief Networks for phone recognition.

Mohamed, A., Dahl, G. E., & Hinton, G. E. (2012). Acoustic modeling using deep belief networks. *Audio, Speech, and Language Processing, IEEE Transactions on*, *20*, 14 –22.

Nair, V. (2010). *Visual object recognition using generative models of images*. Doctoral dissertation, University of Toronto.

Nair, V., & Hinton, G. (2010a). 3-d object recognition with deep belief nets. *Advances in Neural Information Processing Systems*.

Nair, V., & Hinton, G. E. (2010b). Rectified linear units improve Restricted Boltzmann Machines. *Proceedings of the 27th International Conference on Machine Learning (ICML-10)* (pp. 807–814).

Nair, V., Susskind, J., & Hinton, G. (2008). Analysis-by-Synthesis by Learning to Invert Generative Black Boxes. *Artificial Neural Networks-ICANN 2008*, 971–981.

Neal, R. M. (1992). Connectionist learning of belief networks. *Artificial Intelligence*, *56*, 71–113.

Ngiam, J., Chen, Z., Koh, P. W., & Ng, A. Y. (2011). Learning deep energy models. *Proceedings of the 28th International Conference on Machine Learning (ICML-11)* (pp. 1105–1112).

Olshausen, B. A., & Field, D. J. (1996). Emergence of simple-cell receptive field properties by learning a sparse code for natural images. *Nature*, *381*, 607.

Pearl, J. (1988). *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Morgan Kaufmann.

Pinto, N., Doukhan, D., DiCarlo, J. J., & Cox, D. D. (2009). A high-throughput screening approach to discovering good forms of biologically inspired visual representation.

Raina, R., Madhavan, A., & Ng, A. Y. (2009). Large-scale deep unsupervised learning using graphics processors. *Proceedings of the 26th Annual International Conference on Machine Learning* (pp. 873–880).

Ranzato, M., Huang, F. J., Boureau, Y.-L., & Lecun, Y. (2007). Unsupervised learning of invariant feature hierarchies with applications to object recognition. *Computer Vision and Pattern Recognition, 2007. CVPR'07. IEEE Conference on* (pp. 1–8).

Ranzato, M., Susskind, J., Mnih, V., & Hinton, G. (2011). On deep generative models with applications to recognition. *Computer Vision and Pattern Recognition (CVPR), 2011 IEEE Conference on* (pp. 2857–2864).

Ranzato, M. A., Mnih, V., & Hinton, G. (2010). Generating more realistic images using gated mrf's. *Advances in Neural Information Processing Systems* (pp. 2002–2010).

Ranzato, M. A., Poultney, C., Chopra, S., & LeCun, Y. (2006). Efficient learning of sparse representations with an energy-based model. *Advances in neural information processing systems*, *19*.

Rasmussen, C. E. (2006). Gaussian processes in machine learning. *Advanced Lectures on Machine Learning*, 63–71.

Rifai, S., Vincent, P., Muller, X., Glorot, X., & Bengio, Y. (2011). Contractive auto-encoders: Explicit invariance during feature extraction. *Proceedings of the 28th International Conference on Machine Learning (ICML-11)* (pp. 833–840).

Robbins, H., & Monro, S. (1951). A Stochastic Approximation Method. *The Annals of Mathematical Statistics*, *22*, 400–407.

Salakhutdinov, R. (2010). Learning deep boltzmann machines using adaptive mcmc. *Proceedings of the 27th International Conference on Machine Learning (ICML-10)* (pp. 943–950).

Salakhutdinov, R., & Hinton, G. (2009a). Deep Boltzmann Machines. *Proceedings of the International Conference on Artificial Intelligence and Statistics* (pp. 448–455).

Salakhutdinov, R., & Hinton, G. (2009b). Semantic hashing. *International Journal of Approximate Reasoning*, *50*, 969–978.

Salakhutdinov, R., & Murray, I. (2008). On the quantitative analysis of deep belief networks. *Proceedings of the International Conference on Machine Learning.*

Schmah, T., Yourganov, G., Zemel, R. S., Hinton, G. E., Small, S. L., & Strother, S. C. (2010). Comparing classification methods for longitudinal fmri studies. *Neural Computation*, *22*, 2729–2762.

Schraudolph, N. N., & Graepel, T. (2003). Combining Conjugate Direction Methods with Stochastic Approximation of Gradients. *Proc. 9^{th} Intl. Workshop Artificial Intelligence and Statistics (AIstats)* (pp. 7–13). Key West, Florida: Society for Artificial Intelligence and Statistics.

Schulz, H., Müller, A., & Behnke, S. (2010). Investigating Convergence of Restricted Boltzmann Machine Learning.

Sermanet, P., Eigen, D., Zhang, X., Mathieu, M., Fergus, R., & LeCun, Y. (2014). Overfeat: Integrated recognition, localization and detection using convolutional networks. *International Conference on Learning Representations (ICLR 2014).* CBLS.

Simard, P., LeCun, Y., Denker, J., & Victorri, B. (1996). Transformation invariance in pattern recognition: tangent distance and tangent propagation. *Neural networks: tricks of the trade*, 549–550.

Simard, P. Y., Steinkraus, D., & Platt, J. (2003). Best practices for convolutional neural networks applied to visual document analysis. *International Conference on Document Analysis and Recogntion (ICDAR), IEEE Computer Society, Los Alamitos* (pp. 958–962).

Smolensky, P. (1986). *Information processing in dynamical systems: foundations of harmony theory.* MIT Press Cambridge, MA, USA.

Snoek, J., Larochelle, H., & Adams, R. P. (2012). Practical bayesian optimization of machine learning algorithms. *Neural Information Processing Systems.*

Snoek, J., Swersky, K., Zemel, R., & Adams, R. P. (2013). Input warping for bayesian optimization of non-stationary functions. *Advances in Neural Information Processing Systems Workshop on Bayesian Optimization.*

Susskind, J. M., Anderson, A. K., & Hinton, G. E. (2010). *The Toronto Face Database* (Technical Report UTML TR 2010-001). University of Toronto, Department of Computer Science.

Sutskever, I., & Hinton, G. E. (2007). Learning multilevel distributed representations for high-dimensional sequences. *Proceeding of the Eleventh International Conference on Artificial Intelligence and Statistics* (pp. 544–551).

Sutskever, I., & Hinton, G. E. (2008). Deep, narrow sigmoid belief networks are universal approximators. *Neural computation*, *20*, 2629–2636.

Sutskever, I., Martens, J., Dahl, G., & Hinton, G. (2013). On the importance of initialization and momentum in deep learning. *Proceedings of the 30th International Conference on Machine Learning (ICML-13)* (pp. 1139–1147).

Sutskever, I., & Tieleman, T. (2010). On the convergence properties of contrastive divergence. *Proc. Conference on AI and Statistics (AI-Stats).*

Swersky, K., Snoek, J., & Adams, R. P. (2013). Multi-task bayesian optimization. *Advances in Neural Information Processing Systems.*

Swersky, K., Tarlow, D., Sutskever, I., Salakhutdinov, R., Zemel, R. S., & Adams, R. P. (2012). Cardinality restricted boltzmann machines. *NIPS* (pp. 3302–3310).

Taylor, G. W., Hinton, G. E., & Roweis, S. T. (2007). Modeling human motion using binary latent variables. *Advances in Neural Information Processing Systems*, *19*, 1345.

Tieleman, T. (2007). Some investigations into energy-based models. Master's thesis, University of Toronto.

Tieleman, T. (2008). Training restricted Boltzmann machines using approximations to the likelihood gradient. *Proceedings of the 25th international conference on Machine learning* (pp. 1064–1071).

Tieleman, T. (2010). *Gnumpy: an easy way to use GPU boards in Python* (Technical Report UTML TR 2010-002). University of Toronto, Department of Computer Science.

Tieleman, T., & Hinton, G. E. (2009). Using Fast Weights to Improve Persistent Contrastive Divergence. *Proceedings of the 26th international conference on Machine learning* (pp. 1033–1040).

Torralba, A., Fergus, R., & Freeman, W. T. (2007). Object and scene recognition in tiny images. *J. Vis.*, *7*, 193–193.

Vincent, P., Larochelle, H., Bengio, Y., & Manzagol, P. A. (2008). Extracting and composing robust features with denoising autoencoders. *Proceedings of the 25th international conference on Machine learning* (pp. 1096–1103).

Wainwright, M. J., & Jordan, M. I. (2003). Graphical models, exponential families, and variational inference. *UC Berkeley, Dept. of Statistics, Technical Report*, *649*.

Wang, F., & Landau, D. P. (2001). Efficient, multiple-range random walk algorithm to calculate the density of states. *Physical Review Letters*, *86*, 2050.

Welling, M. (2009a). Herding dynamic weights for partially observed random field models. *Proceedings of the Twenty-Fifth Conference on Uncertainty in Artificial Intelligence* (pp. 599–606).

Welling, M. (2009b). Herding dynamical weights to learn. *Proceedings of the 26th Annual International Conference on Machine Learning* (pp. 1121–1128).

Welling, M., & Hinton, G. E. (2002). A new learning algorithm for mean field Boltzmann machines. *Proceedings of the International Conference on Artificial Neural Networks* (pp. 351–357). Springer.

Welling, M., Rosen-Zvi, M., & Hinton, G. (2005). Exponential family harmoniums with an application to information retrieval. *Advances in Neural Information Processing Systems*, *17*, 1481–1488.

Wiskott, L., & Sejnowski, T. J. (2002). Slow feature analysis: Unsupervised learning of invariances. *Neural computation*, *14*, 715–770.

Zhou, C., Schulthess, T. C., Torbrügge, S., & Landau, D. (2006). Wang-landau algorithm for continuous models and joint density of states. *Physical review letters*, *96*, 120201.

Zhu, L., Chen, Y., Ye, X., & Yuille, A. (2008a). Structure-perceptron learning of a hierarchical log-linear model. *Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on* (pp. 1–8).

Zhu, L., & Yuille, A. L. (2006). A hierarchical compositional system for rapid object detection. *Department of Statistics, UCLA.*

Zhu, L. L., Chen, Y., & Yuille, A. (2011). Recursive compositional models for vision: Description and review of recent work. *Journal of Mathematical Imaging and Vision*, *41*, 122–146.

Zhu, L. L., Lin, C., Huang, H., Chen, Y., & Yuille, A. (2008b). Unsupervised structure learning: Hierarchical recursive composition, suspicious coincidence and competitive exclusion. In *Computer vision–eccv 2008*, 759–773. Springer.

Zhu, X. (2005). *Semi-supervised learning literature survey* (Technical Report 1530). Computer Sciences, University of Wisconsin-Madison.