Learning to Extract Parameterized Features by Predicting Transformed Images

Sida Wang

Department of Computer Science

University of Toronto

A thesis submitted for the degree of

Bachelor of Applied Science

April 2011

Acknowledgements

I would like to acknowledge, above all, my advisor Geoffrey Hinton for his continued support and enthusiasm in this research. I learned a lot under his excellent supervision and this research would not have been possible without him. In fact, many ideas in this thesis are his or started by his suggestions. Mistakes, bugs in here are certainly due to me since I implemented everything in this thesis.

I thank Dr. Marc'Aurelio Ranzato for many early help and suggestions to get me up to speed. Any progress was greatly sped up by the availability of cudamat [1] by Volodymyr Mnih and Gnumpy by Tijmen Tieleman [2]. These excellent tools makes implementing learning algorithms on the GPU much easier. Finally, I thank everyone in the machine learning group for the great academic atmosphere.

Abstract

Interesting data such as images, videos and speech are often highly structured - real photos are far from a random set of pixels. Because of this structure, the pixel representation is inefficient. Performance on tasks like classification, tracking, object recognition etc. can be improved by first extracting features. A variety of methods have been developed for feature extraction from adaptive methods like PCA, k-means, Gaussian mixture models, restricted Boltzmann machine (RBM), autoencoder to hand-crafted features like wavelets, oriented Gabor filters, SIFT etc.

Individual components within these adaptive representations are either independent from each other or hard to interpret. In PCA, k-means, and GMM, the representation of a horizontal edge at one location has nothing to do with the representation of another horizontal edge at some other location. On the other hand, codes extracted by RBM and deep autoencoder are hard to interpret and change unpredictably under simple transformations. To overcome this limitation, parameterized features with explicit pose parameters are desired. Although pose is already present in hand-crafted features like SIFT features (the pose is in its descriptor), such representations have never been learned. In this thesis, we train "capsules" to perform sophisticated computation, and encapsulate the result in a small vector of instantiation parameters representing the pose. The method used to obtain these capsules and extract these instantiation parameters, called the transforming autoencoder, is introduced. The transforming autoencoder is trained on pairs of images related by a transformation while having direct access to this underlying transformation that we

would like to capture. Under this scheme, the instantiation parameters are forced to take on a desired meaning without their values being explicitly specified.

In addition to its fundamental appeal, the usefulness of such parameterized representations is then demonstrated through classification on MNIST digits and its ability of generalizing to transformed test digits that come from a different distribution than the training digits.

Contents

Chapter 1

Introduction

1.1 Introduction

Interesting data such as images, videos and speech are highly structured - real photos are far from a random set of pixels. Because of this structure, pixel representation of an image is highly inefficient. For example, real 640×480 images lie on or near a lower dimensional manifold within the full $640 \times 480 = 30720$ dimensional space. Interesting structures and useful information are all buried in the pixel representation, so it is difficult to do anything useful with them.

Performance on various tasks like object recognition, tracking, image compression etc. can be improved by first extracting suitable representations, or features [3; 4]. Extracting useful features is an important goal in unsupervised learning and there are many classical methods for this task like PCA, k-means, Gaussian mixtures [5], restricted Boltzmann machine (RBM) [6], etc. On the other hand, there are many hand-crafted features like wavelets [7], oriented Gabor filters [8], SIFT [4], etc. All of the adaptive features are specific and do not capture common relationships - the representation of a horizontal edge at one location often has nothing to do with the representation of another horizontal edge at a nearby location. They are usually represented by different feature vectors that are independent from each other.

This independence, or ignorance of relationships allows these methods to be more general, but also unsatisfying for tasks like image representation where these

basic spatial relationships are extremely important and highly predictable. It would be better if a feature for a horizontal edge represents not only its presence, but also where it is, making its relationship with other similar edges apparent. In other words, the representation for a visual entity should have instantiation parameters that explicitly capture some desired properties. SIFT, with its descriptor, is an example of such a parameterized feature.

However, such parameterized representation has never been learned from data. We show that the transforming autoencoder is an efficient method to learn representation with instantiation parameters.

There are many benefits to having parameterized features. One type of parameterization of particular interest is of geometric information. Any object recognition system should be robust to simple affine transformations. If all pixels in an image are translated 2 pixels to the right, or rotated 10 degrees clockwise, the identity of an object in the image should not change, although the pixel representation might have changed considerably. Parameterized features are ideal for capturing this type of information.

In addition, an object can be defined in terms of the spatial relatinships between its parts and spatial relationships between these parts and the whole [9]. Having these parameterized features is a first and concrete step towards representing objects in this way. In addition to its fundamental appeal, the usefulness of this representation is demonstrated through classification tasks on the MNIST dataset and the representation's ability to generalize to transformed test images distributed differently than the training images.

1.2 What is the transforming autoencoder

1.2.1 The autoencoder

The autoencoder is a multi-layer neural network [1] used to extract compact codes from structured data like natural images [10]. The basic idea is simple to explain. If we force the output of a multi-layer neural network to be the same as its input, and we put a tiny layer in the middle to form a bottleneck, then the value of this

[1] See A for basic background information on neural network and how to train them

bottleneck layer is forced to be an efficient code of the input. Due to its many layers and the non-linearities, the codes found by an autoencoder can be compact and powerful.

For example, the input data might be small images $x \in \mathbb{R}^{28\times 28=784}$, the bottleneck might be 30 real numbers $c = f(x) \in \mathbb{R}^{30}$. The output $y = g(c) \in \mathbb{R}^{784}$ can be trained to reconstruct the input, for example, by minimizing the reconstruction error between output y and target x : $||y - x||^2$ or the cross-entropy error if logistic units are used.

1.2.2 The transforming autoencoder

A compact code can be extracted using an autoencoder. However, this code changes in an unpredictable way when the input is transformed. Usually this is not a problem, but a consequence of having a compact code with high information content per bit. However, if the input is transformed in a way that does not change its content, then we would like the code to change in a predictable way that reflects the underlying transformation. For instance, shifting the whole image by a few pixels usually does not change the perceived content of the image. However, the pixel representation can be very different and the autoencoder code might change unpredictably. In the case of images, transformations that we are most interested in are affine transformations. Similarly for speech signals, timing translations and amplitute scalings do not affect the perceived content but they alter the wave representation significantly. [1]

We would like slightly rotated, or slightly translated images to have similarly related representations that have predictable relationships to each other. The transforming autoencoder is a method to learn such representations. The transforming autoencoder consists of many capsules. Each capsule is an independent sub-network used to extract a single parameterized feature representing a single visual entity. Each parameterized feature consists of a single gating unit, which indicates if a visual entity is present and some instantiation parameters, which represent the pose. The transforming autoencoder can be trained by asking it to

[1] In the sense that the i_{th} pixel and the amplitude $A(t = t_0)$ can be very different after the transformation is applied. The rest of this work will be focused on images, in particular, on MNIST digits.

predict a transformed version of its input image, given the original input image and what transformation is performed. During training, the transforming autoencoder does not have to be told of the instantiation parameter values - it is how these instantiation parameters should change that is specified. After training, the transforming autoencoder codes its input in terms of a small set of compact (say, 2-9 dimensions) instantiation parameters of visual entities, one per capsule. Each capsule learns its own visual entity. Then the transforming autoencoder is able to reconstruct its output from these compact and meaningful instantiation parameters. Within the transforming autoencoder, independent capsules perform sophisticated computation, and encapsulate (thus the name capsule) their results in some instantiation parameters. [1]

Particular meanings are forced onto these instantiation parameters by specifying how they change. For example, the instantiation parameters can be position coordinates of visual entities or coordinate transformation matrices. If we train a transforming autoencoder to use position coordinates, the following should happen. When an image is slightly translated, the same capsules are activated (gating units are on), but the instantiation parameters should be different by exactly the same amount of translation. Thus, we can obtain information about spatial relationships in a compact, and usable form. This idea applies not only to images, but also to temporal shift in speech signals and other structured transformations we would like to explicitly deal with. Not only do we get compact codes like in the autoencoder, we also enforce what these codes, or instantiation parameters, ought to represent by specifying how these parameters should change.

1.3 Relationship to other works

The core problem that transforming autoencoder addresses is extracting features that deals with important transformations of known types explicitly. There are

[1] We use visual entity to mean things in the image; parameterized feature to mean gating units and instantiation parameters; instantiation parameters to mean the compact pose representation obtained by the transforming autoencoder; capsule to mean the hardware (or sub-network) used to compute these instantiation parameters. We avoid the general word feature from now on because it could mean either the representation (parameterized feature, and instantiation parameters are examples of representation) or things in the image (visual entity)

many previous approaches to this problem from architectural level solutions such as convolutional neural networks [11], to engineered features such as SIFT [4], and to model level solutions that try to separate content from the pose and model pose explicitly [12; 13]. Convolutional neural networks deal with translation at an architectural level by applying replicated feature detectors all over the image and subsample or pool these features to achieve invariance to translations in its upper level features [14]. However, in achieving this invariance, detailed location information is lost. Trying to deal with other invariances this way by convolving with different orientations, different scales, etc. causes the number of operations to grow multiplicatively. For example, if there are $T = 10 \times 10$ translations, $R = 360/10 = 36$ rotations, and $S = 5$ scales, then we have to replicate $TRS = 18000$ copies of each feature detector. More generally, if each of the K invariances needs M replicated detectors, then we need M^K replications in total. This is exponential in K. We show that the transforming autoencoder can address this problem by extending it from simple translations to rotation, and then to full affine transformations.

Engineered features such as SIFT keep precise location information. The idea behind many engineered features is that an image should be represented by a set of characteristic points such as corners, or even edges. These characteristic points should be consistently extracted from images of the same object in different viewpoints. Then a feature descriptor would concisely summarize each particular feature in a way that has good retrieval properties (low false positive rate, high true positive rate). However, neither the feature descriptor nor the extraction method is learned from data. The transforming autoencoder achieves these desired properties of engineered features through learning. The advantage here is that features obtained by a transforming autoencoder do not have to be designed for each specific application, thus making this task much more routine. Transforming autoencoders also make it easy to scale with the computational power and the number of parameters.

Models that attempt to separately represent content from pose inspired this work. But these models are often unsuitable for spatial transformations in images, although they are more general and can achieve reasonable results. For example, bilinear models [13] can be applied to distinct objects like letters and faces where

the respective poses, font and viewpoint, are very different things. A recent work on a transformation model using 3-way factored RBM [12] is another example. Like the transforming autoencoder, this model is also trained on pairs of images differing by a transformation. It does not need to be told of the transformation but can infer it from the training data. However, the 3-way factored RBM has trouble generalizing to spatial transformations with more than 2 degrees of freedom. One reason is that the 3-way factored RBM does not have the right representation for spatial transformations, but it instead deals with the many possible transformations using its distributed internal representations, which is actually inefficient. Its hidden units cannot code quantity in a convincing way, even if real valued units like rectified linear units (ReLU) [15] are used. In the case of translations, it would code "shift 3 pixels to the left" very differently from "shift 5 pixels to the left". This thesis would have been on 3-way factored RBM if it worked efficiently. Based on my experience using ReLU, the 3-way factored RBM seems to prefer 1-in-N encoding as it becomes highly trained on translations, when there are enough units. In comparison, the transforming autoencoder takes its input data into a compact domain where the transformations of interest are naturally represented in terms of matrices or position vectors.

In summary, the aim of this research is to obtain parameterized features similar in spirit to SIFT in a way that adapts to the task at hand. However, we would like to do so by specifying how the instantiation parameters should vary under transformations of interest without hand engineering the features for each particular application. Compared to convolutional neural networks, instantiation parameters obtained by transforming autoencoder should vary in a specified way under transformations of interest, instead of losing information by being invariant.

Chapter 2

The transforming autoencoder

2.1 The architecture of transforming autoencoder

2.1.1 A high level overview

The transforming autoencoder consists of many capsules. Each capsule is an independent sub-network used to extract a single set of instantiation parameters. Each parameterized feature consists of a single gating unit, indicating if a visual entity is present and some instantiation parameters, representing the pose. [1]

We introduce the idea in terms of 2d translations of images. Instantiation parameters (2 dimensional) should represent the position of the visual entity that its capsule is responsible for. Since we know the position differences between our input and output vectors, we can add to the predicted position this known difference, and ask the network to reconstruct the correct output. Adding this difference forces a meaning onto the code extracted so the network has to extract positions of visual entities in the image. Although each capsule processes information independently, it cannot cheat by extracting a trivial position since the code extracted has to contain all the information needed to reconstruct the output. Therefore, each capsule tends to learn to represent a different visual entity from other capsules.

It would be more satisfying (and theoretically possible) if the network did not need to be told of this known transformation such as in [12; 13]. But this

[1] See A.4 for terminology information.

transformation is readily available and can serve to force the autoencoder to find an appropriate representation. This information is available to us. When we saccade our eye around this page, we can incorporate information from this new view to what we obtained before because we know how much we saccade. Similar information would be available in robotics application as well through the control, and sensor systems. So we explore what happens when we have direct access to such transformations, and introduce an architecture that takes advantage of this information.

2.1.2 Details of the transforming autoencoder

We first demonstrate the concept by dealing with image translations and then generalize to full affine transformations later. Suppose the raw data $w_i \in \mathbb{R}^{784}$ are 28×28 images of MNIST digits. We may apply simple geometric transformations (in this case just translation) to w^i to generate our training data set, here I use the superscript to denote training case i.

$$\{x^i = T(s_x^i, w^i), t^i = T(s_t^i, w^i), s^i\}_{i=1}^M$$

For each training case (dropping the superscript for cleaner notation), $T(s, x) \in \mathbb{R}^{784}$ takes $x \in \mathbb{R}^{784}$ to the image of x shifted by $s \in \mathbb{R}^2$. $s = s_t - s_x$ is the difference between the underlying translations of t and x. For example, $s_t, s_x \in \mathbb{R}^2$ might be distributed according to a spherical Gaussian with zero mean and variance σ^2.

The transforming autoencoder is now easy to define. Let the transforming autoencoder have N capsules. Each capsule has recognition hidden units $H_r \in \mathbb{R}^{N_r}$, instantiation parameter units $c \in \mathbb{R}^2$ gating unit $p \in (0, 1)$, and generative hidden units $H_g \in \mathbb{R}^{N_g}$. The input to each capsule is the whole input image, and capsules are independent from each other. [1] Each capsule produces an output $y \in \mathbb{R}^{784}$, these outputs are added up linearly $Y = \sum_j y_j \in \mathbb{R}^{784}$ and the final output is $O = \sigma(Y + b_o) \in \mathbb{R}^{784}$. Where $\sigma(x) = \frac{1}{1+e^{-x}}$ is component-wise sigmoid function, and $b_o \in \mathbb{R}^{784}$ is the output bias. The output of the network O should be like the target t.

[1] some evidence suggests that capsules with local recognition fields may work better, at least when the computing performance gain is taken into account. See 4.4.

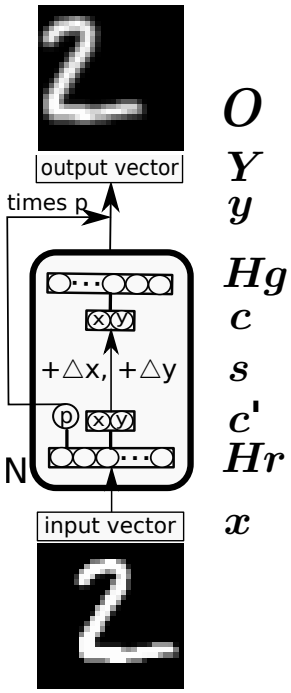

Figure 2.1: Simplest transforming autoencoder architecture for translations. N is the number of capsules. Each of the N capsules is gated by unit p at the final output of that capsule.

Let x, y be the input and output of an individual capsule. Then, we have,

$$\begin{aligned} y &= p(W_{hy}H_g) \in \mathbb{R}^{784} \\ H_g &= \sigma(W_{cg}c + b_g) \in (0,1)^{N_g} \\ p &= \sigma(W_{hp}H_r + b_p) \in (0,1) \\ c' &= c + s \in \mathbb{R}^2 \\ c &= W_{hc}H_r + b_c \in \mathbb{R}^2 \\ H_r &= \sigma(W_{xh}x + b_r) \in (0,1)^{N_r} \end{aligned} \tag{2.1}$$

Where $c = c(x),\ p = p(x)$ are values of the instantiation parameters and gating units of the current capsule. And $W_{xh}, W_{hc}, W_{cp}, W_{cg}, W_{hy}$ are, respectively, weight matrix from input to recognition hidden units, from recognition hidden units to instantiation parameters, from recognition hidden units to gating

units, from instantiation parameter units to generative hidden units, and from generative hidden units to output. See figure 2.1 for the schematic of this transforming autoencoder with labels. Training is done by fixing this architecture and minimizing the cross-entropy error between O and t. Basically, we want the final network function F where $O = F(s, x)$ to be like the translation function T where $t = T(s, x)$. However, F has to do so by using a small number of capsules and extracting a set of compact instantiation parameters $c \in \mathbb{R}^2$, one for each capsule.

Note how $c' = c + s$. In a way, the instantiation parameters extracted by each capsule are "corrupted" by the translation vector s. c and p are just a function of the input, while the output is predicted only from c' and p. So c' has to represent the content of the input while passing on the amount of translation to reconstruct the output. It is thus almost impossible for c to represent anything other than the position coordinates of visual entities in the input. However, the transforming autoencoder did not have supervision signals for the exact instantiation parameters of each visual entity, it only had information about how these parameters should change under a global transformation. This is based on the assumption that instantiation parameters that represent pose information should change in a consistent way under global transformations. Providing explicit supervision is an alternative method, but impossible unless we know exactly what the instantiation parameters should be (then there is little point). Although we have no idea which capsule should deal with which visual entity, nor what the instantiation parameters are, as long as we know how they should change, then we can train the transforming autoencoder. Luckily for this method, most transformations we would like to represent and be invariant for object recognition are global transformations such as translation, rotation, scaling, affine, homography, brightness, contrast, etc.

2.2 Training the transforming autoencoder

2.2.1 How to train

We can then train this autoencoder using back-propagation [5] (See also A). For training data $\{x, t, s\}$, we minimize the cross-entropy objective function,

$$CE(t, O) = -\sum_{j=1}^{784} t_j \log(O_j) + (1 - t_j) \log(1 - O_j)$$

In reality, we use mini-batches of size 100. See more details in A.1.

2.2.2 The transforming autoencoder on translations of MNIST

I train the transforming autoencoder described above on the MNIST dataset. The data is translated by an amount distributed according to a 2d isotropic Gaussian with $\sigma = 3$ and then $\sigma = 10$. I use a weight decay of 0.0001, mini-batches of size 100 and my implementation runs on an Nvidia GPU board. For conceptual details please see A and for implementation details please see C. The tables and figures to follow show squared reconstruction errors, reconstructions, and generative fields grouped by capsules. The reconstructions shown are on test data the transforming autoencoder has not seen before. However, these test data are easier than the training data, since the inputs are also transformed in the training data, whereas the inputs shown in the figures are centered.

case	training sqr. error	testing sqr. error
100 capsules $\sigma = 10$	5.31	5.61
100 capsules $\sigma = 3$	3.58	3.75

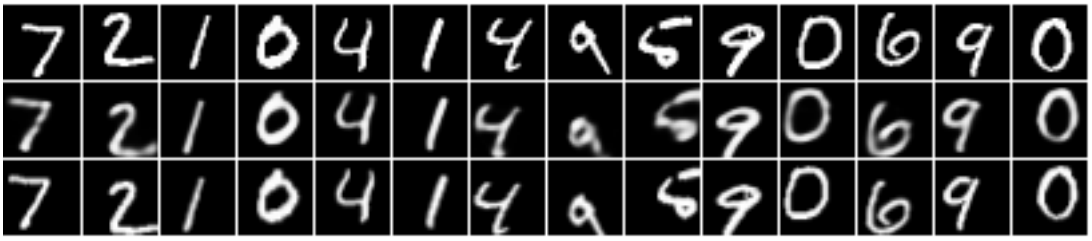

Figure 2.2: Reconstructions of translated digits using a transforming autoencoder with 100 capsules, each with 10 recognition units and 10 generative units. **top row**: input, **middle row**: reconstruction, **bottom row**: target output

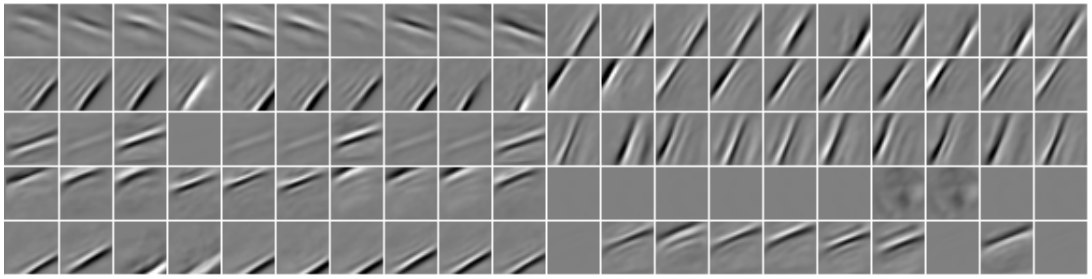

Figure 2.3: All 10 generative units of the first 10 (out of 100) capsules which produced the above reconstructions

Figure 2.4: Reconstructions of translated digits using a transforming autoencoder with 100 capsules, each with 10 recognition units and 10 generative units. **top row**: input, **middle row**: reconstruction, **bottom row**: target output

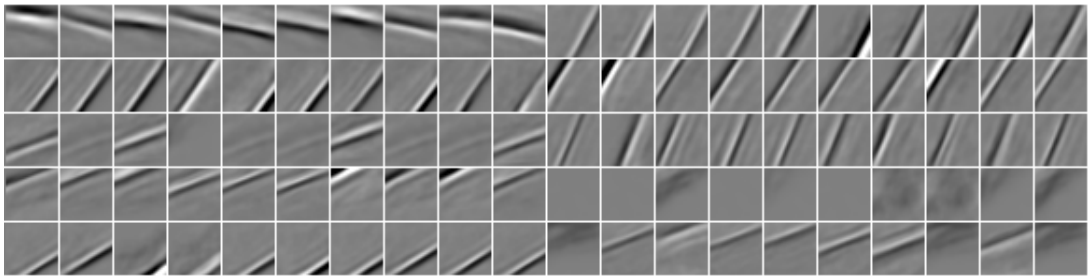

Figure 2.5: All 10 generative units of the first 10 (out of 100) capsules which produced the above reconstructions

2.2.3 The transforming autoencoder on more complex transformations

Convolutional neural network deals with translations just fine. In fact, the fields learned by the transforming autoencoder trained on translations do look like replicated features. However, convolutional neural network would have trouble with more complicated transformations as discussed in 1.3. The transforming

autoencoder can be more easily applied to more complex transformations.

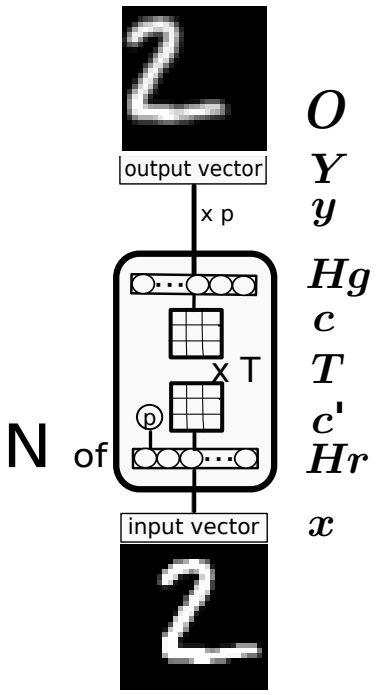

Figure 2.6: Transforming autoencoder that extract a full matrix. Similar to figure 2.1

There are two obvious approaches to more complex image transformations. One is to simply introduce explicit parameters for orientation, scaling, or shear in addition to x and y. For example, to deal with rotation and translation, we introduce capsule parameters x, y, θ that are to be predicted and we provide $\Delta x, \Delta y$ and $\Delta\theta$ that are known where θ is the orientation parameter. Equation (2.1) still applies in this case, with one additional parameter (now $c' = c + s \in \mathbb{R}^3$ instead of $\mathbb{R}^2$).

But perhaps a more coherent approach to spatial transformations is to deal with all 2d affine transformations (translation, rotation, scaling and shearing) at once. Each capsule outputs 9 instantiation parameters which is treated as a 3×3 homography or affine transformation matrix A. A known transformation matrix $T \in {}^3\mathbb{R}^3$ is then applied to A to get matrix TA which is used to predict the output image. This type of change is conceptually simple under the transforming autoencoder framework, equation (2.1) is the only thing that changes, where

$c' = c + s$ now becomes $c' = Tc$. See A for more details and A.2 for how training data was generated in this case.

The transforming autoencoder shown in figures 2.7 and 2.8 has 50 capsules of 20 hidden units each. Each capsule outputs parameters x, y, θ. The transforming autoencoder shown in figures 2.9 and 2.10 has 25 capsules of 40 hidden units each. Each capsule outputs 9 instantiation parameters treated as a 3×3 matrix A.

Case	training error	testing error
50 capsules on translation and rotation	13	13
25 capsules on full affine	5.9	6.2

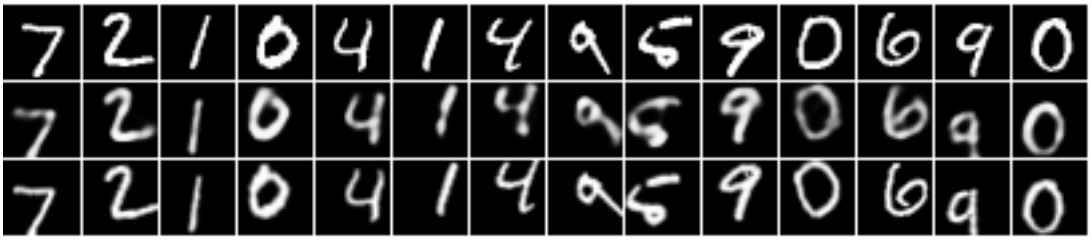

Figure 2.7: Reconstructions of translations and rotations of digits using a transforming autoencoder with 50 capsules, each having 20 recognition units and 20 generative units. Each capsule outputs x, y, θ as instantiation parameters. **top row**: input, **middle row**: reconstruction, **bottom row**: target output

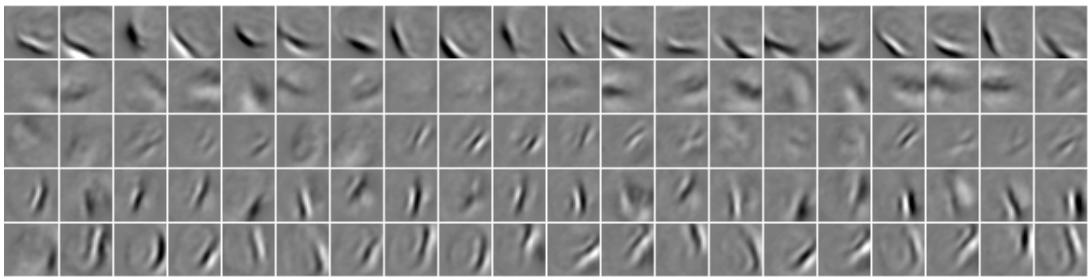

Figure 2.8: all 20 generative fields of the first 5 capsules of the transforming autoencoder that produced reconstructions in figure 2.7

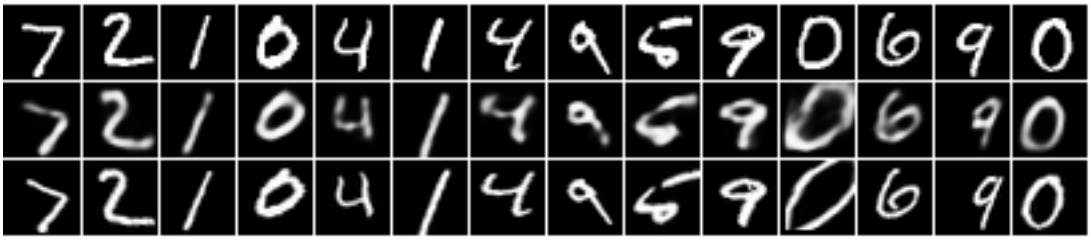

Figure 2.9: Full affine reconstructions using a transforming autoencoder with 25 capsules, each having 40 recognition units and 40 generative units. Each capsule outputs a full affine matrix $A \in^3 \mathbb{R}^3$ as instantiation parameters. **top row**: input, **middle row**: reconstruction, **bottom row**: target output

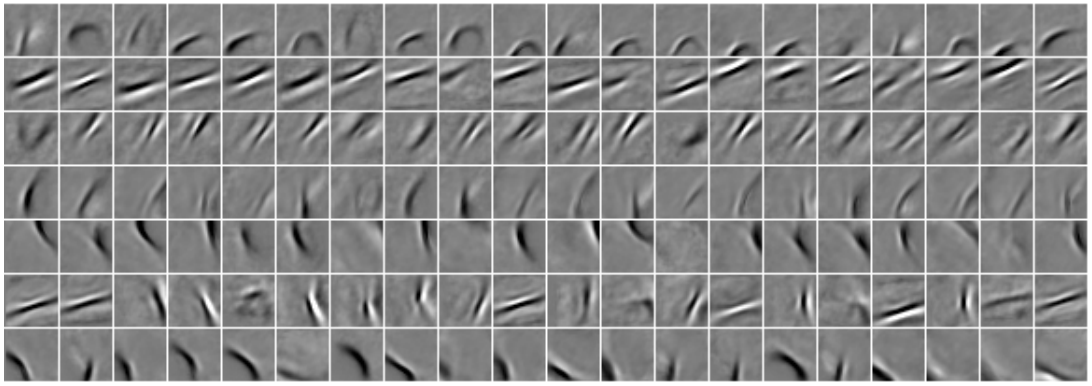

Figure 2.10: The first 20 (out of 40) generative units of the first 7 (out of 25) capsules which produced reconstruction in figure 2.9.

2.3 How does the transforming autoencoder work

The reconstructions are decent, and the filters are interesting looking, but how do the capsules really work?

Here we investigate how instantiation parameters change as we translate the input image. Figures 2.11 and 2.12 are plots of instantiation parameters as the input image is translated.

When transforming autoencoder is trained on small range of transformations, the capsules indeed output coordinates of the visual entities that they are responsible for. The situation is a little complicated for transforming autoencoder trained on larger transformations. The input images can only be shifted so much, whereas it is acceptable for the target image to shift to outside of the box. In

this case, the generative units get most of the training signals and the recognition units are not nearly as good. On the other hand, it might just be that recognition fields need to be messier compared to the generative fields. See 4.2 for more discussions of this point. See figure 2.13 for a comparison between the generative and recognition fields.

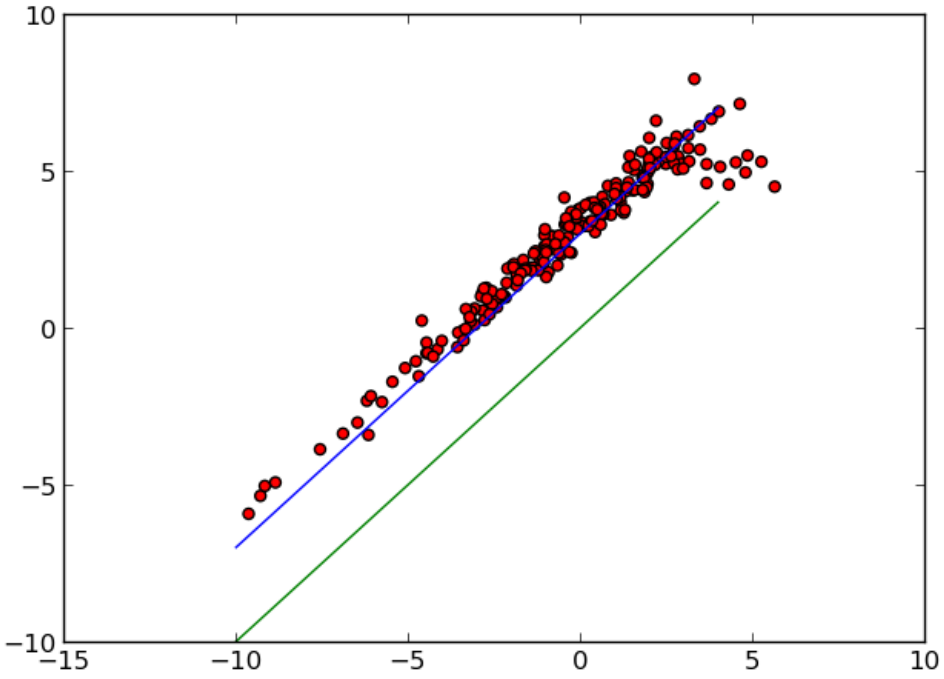

Figure 2.11: Red scatter: the instantiation parameter xs of capsule 0 on a set of input images shifted by 3 pixels to the right vs. the instantiation parameter xs of capsule 0 on the original set of input images; blue line: $y = x + 3$; green line: $y = x$. The red scatter should fall on blue line if the x instantiation parameter do what they are supposed to. The scatter would fall on the green line if instantiation parameters do not change as the input shifts. x and y has nothing to do with input x and linear output y, here they are just coordinates.

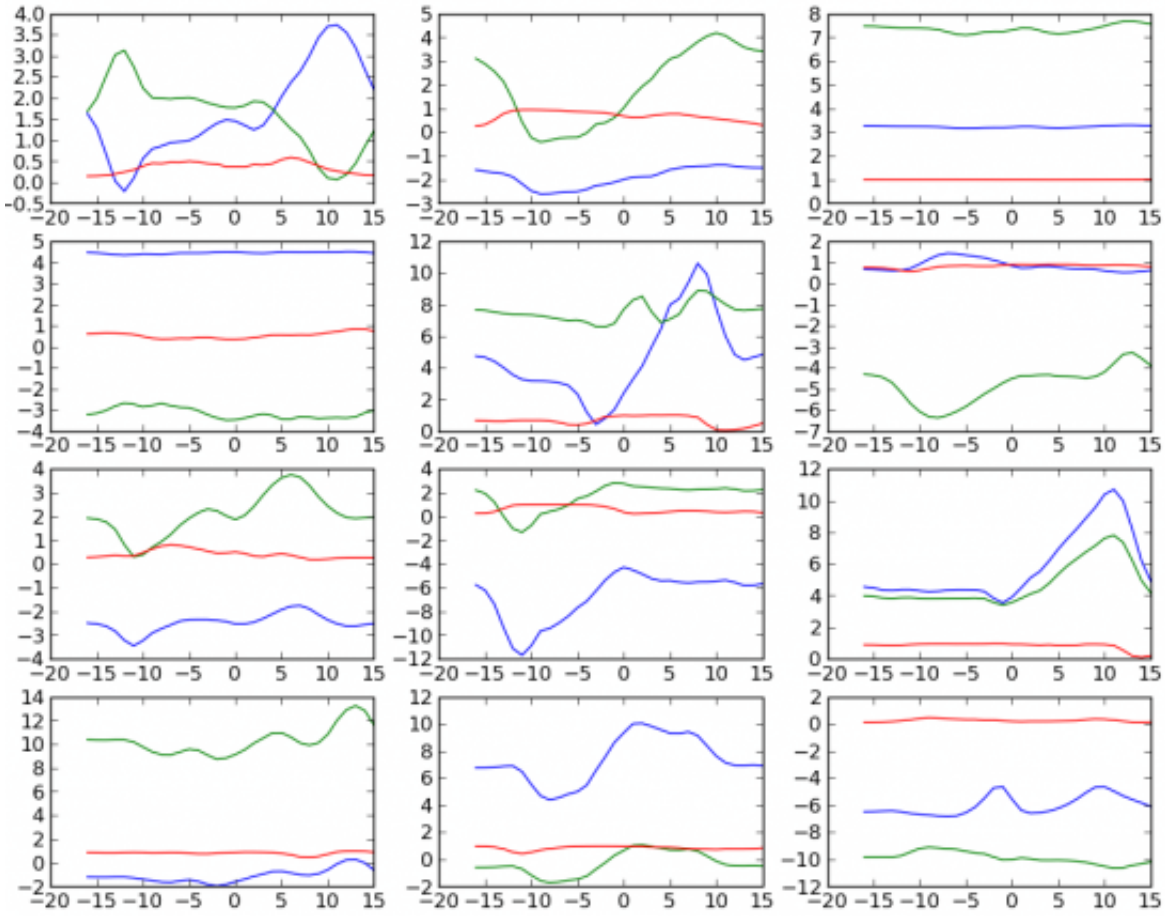

Figure 2.12: Plotted is the capsule outputs as a particular input image shifts in direction $(1, 1)$. red line: gating units, blue line: x outputs, green line: y outputs. Each subplot is for a different capsule.

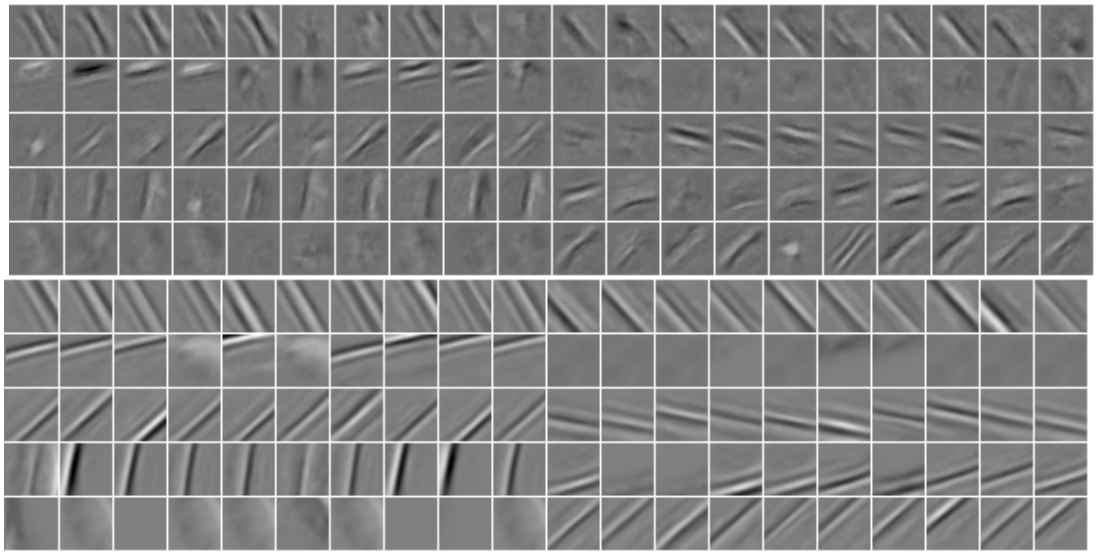

Figure 2.13: Messier recognition fields (top) vs. generative fields (bottom). See 4.2 for more discussions on why this might be so.

Chapter 3

Using the transforming autoencoder for classification

3.1 Classification for MNIST digits

To demonstrate the usefulness of instantiation parameters extracted by capsules, we first apply them to the classical MNIST classification task. The best result so far is obtained using a 6-layer neural networks. The number of units, from visible (784 units) to final output (10 classes) is the following: 784-2500-2000-1500-1000-500-10. That net obtained a result of 35 errors out of 10000 and it was trained on elastically deformed data [16]. [1]

Training raw MNIST digits on logistic regression get 12% errors. This can be compared with applying logistic regression to instantiation parameters obtained by the transforming autoencoder, which got 2.9% errors.

In addition to the dataset, a collection of classification results can be found at http://yann.lecun.com/exdb/mnist/.

[1] All errors quoted in this chapter are test errors on the test set of 10000 cases. The number of errors divided by 10000 gives the percent error rate, which is most often quoted from now on.

3.2 Applying the transforming autoencoder to MNIST classification

3.2.1 Baseline results

I first train a transforming autoencoder with 100 capsules on MNIST digits. The original 784 dimensional data vector is transformed into a code that has 200 dimensions for position and 100 dimensions for gating units. To obtain some baseline results, I simply apply logistic regression to the 200 dimensional position units (100 sets of instantiation parameters, 2 in each set) and 100 dimensional gating units. Logistic regression on the 200 position units gets 292 errors (2.9% errors rate) on the test set of 10000 digits. Logistic regression on the 100 dimensional gating units gets 7% errors. Neither results are exceptional, but logistic regression on instantiation parameters (2.9%) performed much better than logistic regression on raw pixels (12%) despite being more compact, better than the 8.4% errors obtained by logistic regression with deskewing, and better than some nearest neighbors results (5% errors). This is also comparable to the 1998 result which used a neural network with one hidden layer of 1000 units (4.5% errors) and even some deeper neural networks results [14]. However, logistic regression on instantiation parameters is not as good the 1.6% error obtained by neural networks that are carefully trained [14]. Better performance on lower dimensional data suggests that the representation of a digit by a small set of instantiation parameters is sensible. Logistic regression is known to benefit from higher input dimensions and applying logistic regression to instantiation parameters does not take advantage of the nature of these instantiation parameters at all. Nevertheless, performance is better.

3.2.2 Classification using pairwise differences

Using pairwise differences between instantiation parameters obtained from a 100 capsules transforming autoencoder, it got 107 errors out of the 10000 digits test set. It got 115 errors on the validation set, which suggests no overfitting and perhaps that the test set is easier. This classifier was trained using deformed

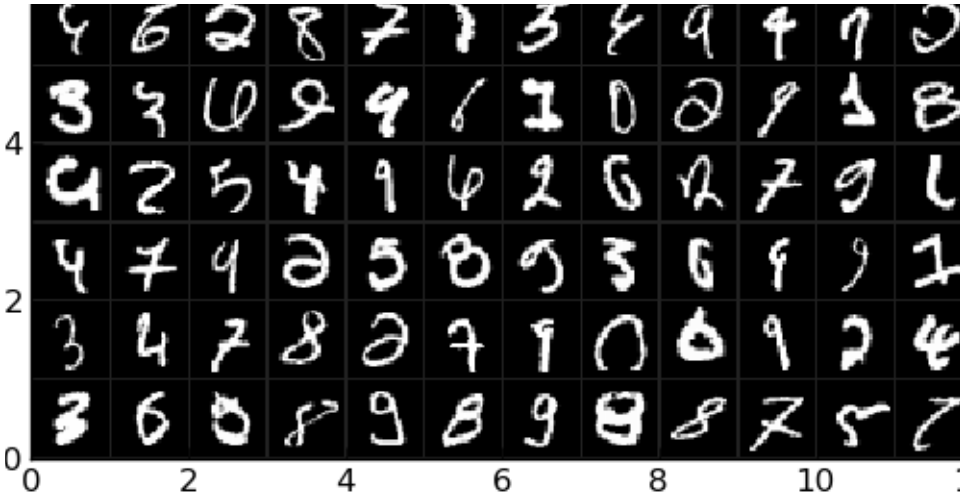

Figure 3.1: 72 samples from 292 miss-classified digits using logistic regression on instantiation parameters.

training set and details to reproduce this result are provided in C.

3.2.3 An overview of how pairwise differences work

One appealing way of describing an object is in terms of the relationship between its parts [9]. Instantiation parameters extracted by transforming autoencoder fit this approach nicely. I would like to use a few English letters to demonstrate this idea. A dot on top of a vertical stroke defines the letter i. Two vertical strokes slightly apart with a horizontal stroke roughly in the middle describes the letter H. The transforming autoencoder extracts instantiation parameters, and has a gating unit indicating whether the visual entity represented is there. We may now look for coincidences of co-occurring visual entities that are especially discriminative. Having two vertical strokes slightly apart strongly indicates the presence of an upper case H, U, or N, it is less likely to be an M, or a W, since the distance required is a little bigger. However, M, or W are still more likely than T given the two vertical strokes. If there is also a horizontal stroke somewhere in between the vertical strokes, then the letter in question is almost certainly an H. Within a small range, the instantiation parameters extracted by each capsule vary by roughly the same amount when the input is translated.

So the differences between instantiation parameters from pairs of capsules are roughly invariant to translations. Thus, if two capsules both dealing with vertical strokes are active, and their positions differ by an amount just right for H, U or N, then this coincidence is strongly indicative that the object present is, in fact, H, U or N. By looking at all pairs, a rather computational expensive task, we can find all these indicative coincidences. This is still a suboptimal way of taking advantage of such information, since these first level capsules really only work over a limited range, but it is simple enough to be tried out quickly.

3.2.4 Mathematical details

Each capsule i extracts instantiation parameters representing position coordinates $x_i, y_i \in \mathbb{R}$ and gating unit $p_i \in (0, 1)$. Define differences matrix D as,

$$d_{ij} = \sigma_x^2(x_i - x_j + b_{ij}^x)^2 + \sigma_y^2(y_i - y_j + b_{ij}^y)^2$$

And define the input matrix F to logistic regression as,

$$f_{ij} = \frac{p_i p_j}{1 + d_{ij}}$$

Where b_{ij}^x, b_{ij}^y are bias terms to be learned, σ_x^2, σ_y^2 are learned scale parameters indicating how acurate a particular coincidence should be. Then f_{ij}s are simply put into a neural network with softmax outputs. The derivatives of the log probability of the correct class are then back-propagated to learn $b_{ij}^x, b_{ij}^y, \sigma_x^2, \sigma_y^2$ but without back-propagating to the transforming autoencoder. This method works fairly well and gets 165 test errors trained only on the original 50000 training data. If it is trained on deformed images as well (see A.2 for how data is generated), then test errors get down to 107 (out of 10000).

3.3 Generalizing to translated test data

3.3.1 Description of this task

What happens if we train a classifier only on centered digits, and then test on translated digits? Now the test set and training set really come from different distributions, which violates a fundamental assumption in machine learning. Thus, this task is almost hopeless for generic systems without prior knowledge on how to generalize a particular task. But because pairwise differences between capsule parameters should be invariant to small translations, the pairwise differences classifier described above might generalize to shifted digits that are not in the training set.

I translate each test input image by a small and random amount $v \in \mathbb{R}^2$ where $v|\sigma \sim \mathcal{N}(0, \sigma^2)$. I train the classifier on the original MNIST training set only and test on translations of the MNIST test set. The transforming autoencoder, however, is trained on the translated training set as well.

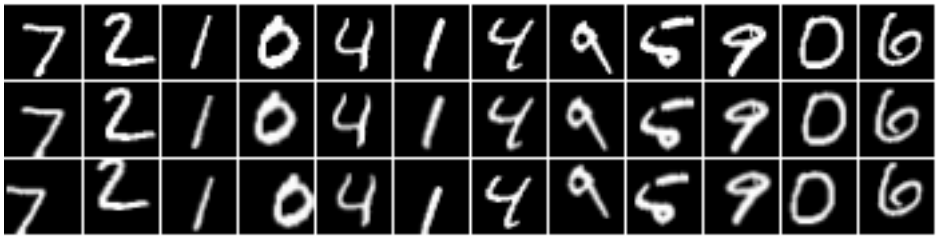

Figure 3.2: Sample of input digits used for this task. **top row**: original digits, **middle**: shifted with $\sigma = 1$, **bottom row**: shifted with $\sigma = 3$

3.3.2 Results on this task

3 types of classifiers are trained for comparisons: Logistic regression on raw digits (simple LR), logistic regression on instantiation parameters without gating units (capsule LR), and finally the pairwise differences classifiers (PDC). Then these 3 classifiers are tested on translated test sets with different amounts of translation specified by σ. The results are as expected.

Case	Training error	Testing error rate
simple LR	8.4%	8.06%
simple LR $\sigma = 1$	–	23%
simple LR $\sigma = 3$	–	68%
simple LR $\sigma = 6$	–	84%
capsule LR	3.53%	3.41%
capsule LR $\sigma = 1$	–	7.2%
capsule LR $\sigma = 3$	–	40%
capsule LR $\sigma = 6$	–	69%
PDC	1.45%	1.65%
PDC $\sigma = 1$	–	2.4%
PDC $\sigma = 3$	–	14.5%
PDC $\sigma = 6$	–	47%

Looking at figure 3.3, we see that logistic regression on instantiation parameters does much better than logistic regression on raw digits. This is probably because capsules are trained using translations, so it somewhat "understands" translation and represents translated digits with more invariance than in the pixel space. The pairwise differences classifier generalizes much better to translated testing data as expected. See B for even more detailed table of numbers, and what error cases look like. Note these results are a little worse than reported in 3.2.2 because they are only trained on the original training set.

Training the transforming autoencoder also takes a fair amount of computational effort, but it does not require labels. So if we just have a small set of labeled data, and a large amount of unlabeled data, we can extract instantiation parameters using transforming autoencoder first, in an unsupervised way, and then train a classifier on these instantiation parameters. Of course, we could have transformed the labeled training set in training the classifier, but that does not diminish the transforming autoencoder's ability of extracting structures without labels. In conclusion, this experiment suggests that the resulting classifier can achieve a better generalization just by using these instantiation parameters. And even better generalization can be achieved by using a specially designed classifier like the pairwise differences classifier.

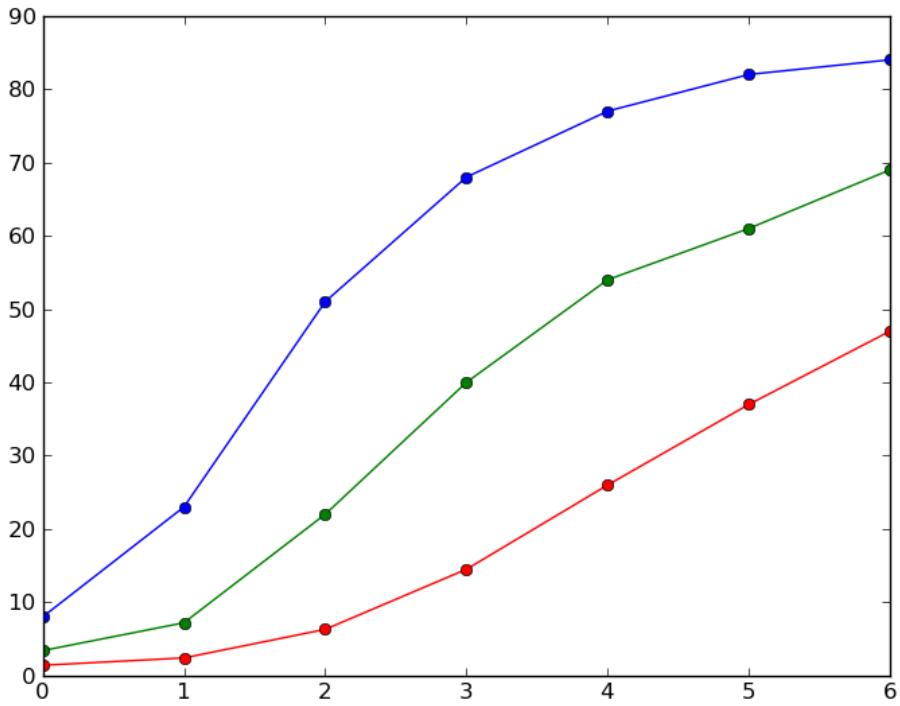

Figure 3.3: Percent error vs. translation size σ on test set. Red line at the bottom is for the pairwise differences classifier; green line in the middle is for capsules LR; blue line at the top is for simple LR. The pairwise differences classifier does much better at generalizing.

Chapter 4

Tricks and future works

4.1 Improving performance by adding layers

The kind of computation demanded by the transforming autoencoder is rather complex. It was unable to achieve state of the art performance using just 4 layers of weights. In particular, the back-propagation algorithm is not very good at deep networks. The gradient at the recognition layers is of lower quality than fresh gradient in the generative layer (see 4.2 for more discussions of this point). An obvious thing to do is adding one or more pre-trained layers below the transforming autoencoder. So the transforming autoencoder would start with more reasonable features than raw pixels. Alternatively, one may try a more powerful optimizer such as the Hessian-free optimizer [17]. Since this is a situation where the training data are essentially unlimited, using a deep net and applying a powerful optimizer is a natural thing to do. However, some preliminary comparisons suggest this did not make too much difference.

4.2 Regularized transforming autoencoder

The most exciting aspect about this work is that once we represent images in the domain of instantiation parameters, we can then proceed to build a hierarchy on top of that. However, to do this, we would need high quality recognition layers. We typically get messier recognition weights than generative weights partially

because not enough gradient information is passed back through the bottleneck see figure 2.13 for comparisons. It might therefore be worthwhile to add regularization in the middle layer that provides additional information consistent with the goal of transforming autoencoder.

Using the notation introduced in 2.1.2: if $x' = T(s, x)$ we would like $c(x') = c(x) + s$. To do so, we may add the term

$$||c(x') - (c(x) + s)||^2$$

to our objective function to be minimized in addition to reconstructing the output. This is an idea due to Navdeep Jaitly.

I am not certain if the messier recognition units are due to lower quality gradients or a more fundamental problem. We can also argue that the recognition task is inherently more difficult and requires messier fields. The cost of communicating N data points under some generative model is $T = C(G) + N \cdot C(P) + N \cdot C(R)$. Where $C(G)$ is the one-time information cost of communicating the generative model. P represents the parameters used to generate each data point and R is the residual error (the generative model is not perfect), $C(P)$ and $C(R)$ are their respective information costs. We note that this cost has nothing to do with the recognition process - only the generative part of the model needs to be communicated. It might be difficult to obtain P. In other words, communicating the recognition model, R, might have a high cost. However, this cost $C(R)$ is completely ignored in communicating data points. This idea is introduced in [18].

4.3 Deeper transforming autoencoder

One strange characteristics of this transforming autoencoder architecture is that capsules are independent from each other. This is reasonable for small transformations, but must be extremely inefficient for larger transformations. To achieve big transformations under the current architecture, the generative fields of each unit must cover the entire range. It can do so by cheating a little.

By comparing figures 4.1, 4.2, and 4.3, we see that the generative fields become progressively more global as the range of translation trained gets larger. A more

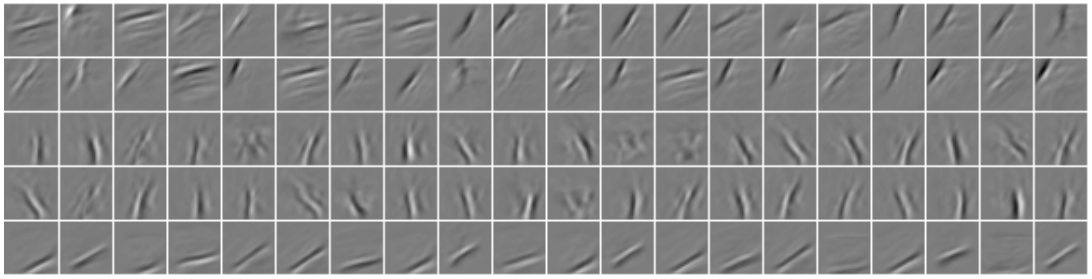

Figure 4.1: Generative fields obtained when trained on small transformations (-3 to 3). They are localized and generally do the right thing - output coordinates

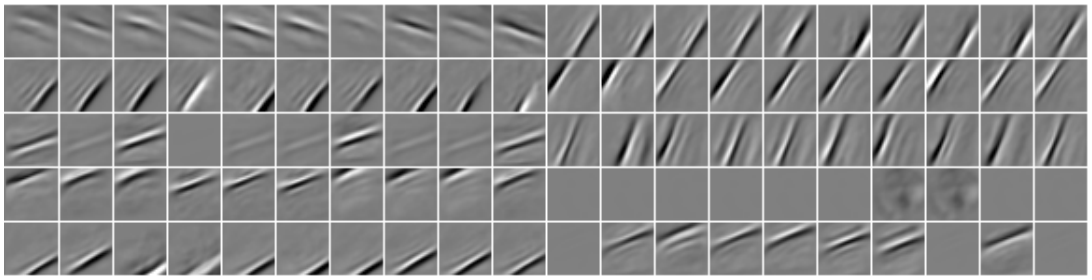

Figure 4.2: Generative fields obtained when trained on slightly larger transformations. $\sigma = 3$

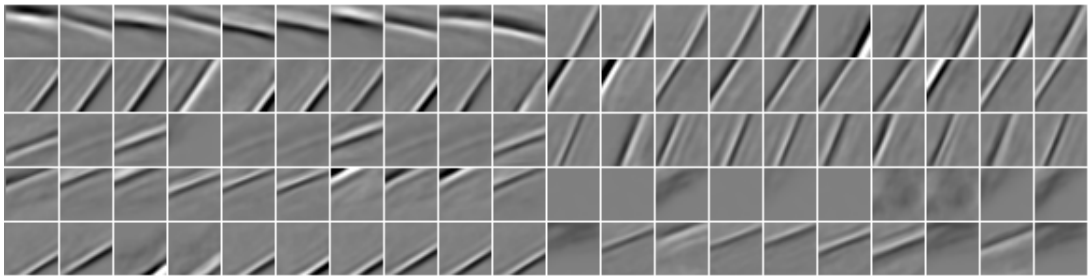

Figure 4.3: Generative fields obtained when trained on large translations. $\sigma = 10$

sensible way of doing this is perhaps by training the transforming autoencoder using small transformations and then build layers on top of the parameter units and introduce dependence to deal with large transformations. A higher level that can redirect one capsule to another when the original capsule goes out of range is desired. This is different from adding more layers to make our simple transforming autoencoder more powerful as discussed in 4.1.

4.4 Transforming autoencoder with local fields

Alex Krizhevsky was able to achieve better classification results using the full matrix version of transforming autoencoder with localized recognition fields.

Chapter 5

Conclusions

This work showed how to learn features similar in spirit to SIFT that have explicitly parameterized poses (section 1.1). We demonstrated that the transforming autoencoder is an efficient way of extracting such parameterized features (section 1.2.2). Each parameterized feature has explicit instantiation parameters for the visual entity that it represents in an image, and a gating unit indicating the presence of such a visual entity. We show that these parameterized features can be learned just by specifying how their instantiation parameters should change under global transformations, instead of having supervision signals for their exact values (subsections 2.1.1. 2.1.2). This specified change forces a meaning onto the instantiation parameters, so they will learn to represent position coordinates, orientations, or even coordinate transformation matrices (subsection 2.2.3).

We then demonstrated the utility of such parameterized features through classification for MNIST digits (chapter 3). Logistic regression on the instantiation parameters got better results than logistic regression on raw pixels. Logistic regression on the instantiation parameters also got better generalization abilities, showing that these parameterized features are more "invariant". Furthermore, we obtained low classification errors (104 errors / 10000 test cases) (subsection 3.2.2) and much better generalization to transformed test digits using the pairwise differences classifier (section 3.3). The generalization task here refers to testing on translated test digits while only training on centered training digits.

We also briefly explored potential methods of improving the current transforming autoencoder, such as adding more layers, adding regularization, making

it deeper, and using local recognition fields (chapter 4).

Appendix A

Neural network and training details

For the purpose of being more self-contained. I include a small appendix on neural networks relating to techniques used in this thesis and give references to more details where applicable.

A.1 Neural network

An artificial neural network is an adaptive learning method with a bunch of units (like neurons) that are connected with other units. In particular, a feed-forward neural network consists of several layers of units, from the input to the output. Units in each layer are only and fully connected to other units in adjacent layers. In a feed-forward neural network, we may sort layers from input to output. Then latter layers do not influence earlier layers. These connections are weighted, and the weights reflect how strongly an earlier unit should influence a latter unit. All neural networks discussed in this thesis are feed-forward neural networks.

As neural network is a well-established topic, I immediately refer readers to better introductions such as chapter 5 of David MacKay's textbook [19] on neural network, which is freely available online at http://www.inference.phy.cam.

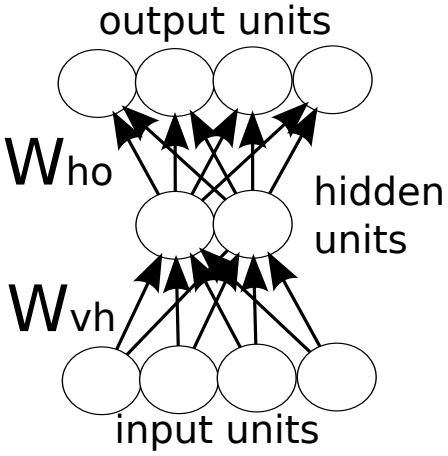

Figure A.1: Neural network with 4 input units, 2 hidden units, and 4 output units.

ac.uk/mackay/itprnn/book.html. Neural network is also introduced in Chris Bishop's book [5], also in chapter 5.

A neural network can be trained by back-propagation which is also explained in [5]. Basically, we find the gradient of some objective function and move our parameters (weights) in the opposite direction to the gradient. Some possible objective functions: $O = ||t - y||^2$ is the squared distance between the output of a neural network and the target;

$$CE(t, O) = -\sum_{j=1}^{N} t_j \log(O_j) + (1 - t_j) \log(1 - O_j)$$

is the cross-entropy function, which is most natural for logistic output units. Here N is the dimension of the output.

A.1.1 Back-propagation using stochastic gradient descent

In practice, evaluating the exact gradient requires all training data, which contains 60000 cases for MNIST. This is wasteful, we instead use mini-batches of 100 cases for each batch. The gradient is evaluated for each batch and the parameters are updated.

A.2 Generating translated/deformed data

I used the SciPy package to generate translated, rotated, deformed data. Each image is first blurred before being transformed for non-discrete transformations. This step can be skipped when images are only translated by integer pixels, or rotated by 90, 180, or 270.

To get deformed images from an affine transformation, I used the following transformation matrix.

$$T = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} + \sigma \begin{pmatrix} X_{11} & X_{12} \\ X_{21} & X_{22} \end{pmatrix}$$

Where $X_{ij} \sim \mathcal{N}(0, 1)$. I typically use $\sigma = 0.05, 0.1$.

A.3 Miscellaneous techniques used in this thesis

A.3.1 Weight decay

A small weight decay is typically used ranging from 1e-4 to 2e-5. The exactly magnitude within this range is not found to matter.

A.3.2 Curriculum learning

Learning on small transformations before going to large transformations help the learning speed. I am not as sure about the final convergence value.

A.3.3 Learning schedule

I typically use a large learning rate to start like 0.1 and gradually move to 0.001 at the end of training.

A.4 Terminologies

We use visual entity to mean things in the image; parameterized feature to mean gating units and instantiation parameters; instantiation parameters to mean the compact pose representation obtained by the transforming autoencoder; capsule to mean the hardware (or sub-network) used to compute these instantiation parameters. We avoid the general word feature because it could mean either the representation (parameterized feature, and instantiation parameters are examples of representation) or things in the image (visual entity)

Appendix B

Detailed classification results

B.1 Table of results

Simple logistic regression, logistic regression on capsules, pairwise differences classifier and finally the pairwise differences classifier trained on translated image with $\sigma_{\texttt{train}} = 1$. σ specified in the table is the standard deviation of normally distributed translation vector applied to test images.

Case	Training error	Testing error rate
simple LR	8.4%	8.06%
simple LR $\sigma = 1$	–	23%
simple LR $\sigma = 2$	–	51%
simple LR $\sigma = 3$	–	68%
simple LR $\sigma = 4$	–	77%
simple LR $\sigma = 5$	–	82%
simple LR $\sigma = 6$	–	84%
capsules LR	3.53%	3.41%
capsules LR $\sigma = 1$	–	7.2%
capsules LR $\sigma = 2$	–	22%
capsules LR $\sigma = 3$	–	40%
capsules LR $\sigma = 4$	–	54%
capsules LR $\sigma = 5$	–	61%
capsules LR $\sigma = 6$	–	69%
PDC	1.1%	1.4%
PDC $\sigma = 1$	–	2.4%
PDC $\sigma = 2$	–	6.3%
PDC $\sigma = 3$	–	14.5%
PDC $\sigma = 4$	–	26%
PDC $\sigma = 5$	–	37%
PDC $\sigma = 6$	–	47%
PDCt	1.1%	1.4%
PDCt $\sigma = 1$	–	1.7%
PDCt $\sigma = 2$	–	2.5%
PDCt $\sigma = 3$	–	6.5%
PDCt $\sigma = 4$	–	15%
PDCt $\sigma = 5$	–	25%
PDCt $\sigma = 6$	–	35%

Figure B.1: errors made by the pairwise difference classifier

[8, 8, 8, 0, 9, 2, 7, 8, 1, 3, 9, 9],
[0, 9, 5, 9, 5, 2, 4, 9, 5, 1, 7, 3],
[3, 3, 3, 9, 3, 8, 7, 3, 7, 4, 9, 5],
[7, 9, 8, 9, 9, 0, 9, 9, 1, 2, 8, 1],
[5, 1, 4, 8, 4, 0, 1, 4, 4, 9, 3, 0],
[9, 5, 5, 8, 0, 7, 2, 9, 0, 1, 7, 0],
[3, 3, 2, 8, 3, 3, 9, 7, 7, 3, 1, 8],
[8, 7, 5, 9, 3, 5, 8, 0, 5, 6, 9, 8],
[4, 9, 6, 9, 8, 8, 9, 3, 8, 8, 3, 3],
[9, 9, 3, 5, 5, 5, 5, 5, 2, 9, 5, 7],
[8, 3, 6, 5, 5, 9, 2, 2, 2, 2, 4, 3],
[7, 7, 5, 6, 5, 9, 8, 8, 7, 3, 9, X]

Figure B.2: Labels of the 143 errors made by this classifier. These labels correspond to the figure B.1 and are all wrong.

B.2 Error cases

B.3 Error cases when generalizing to translated test data

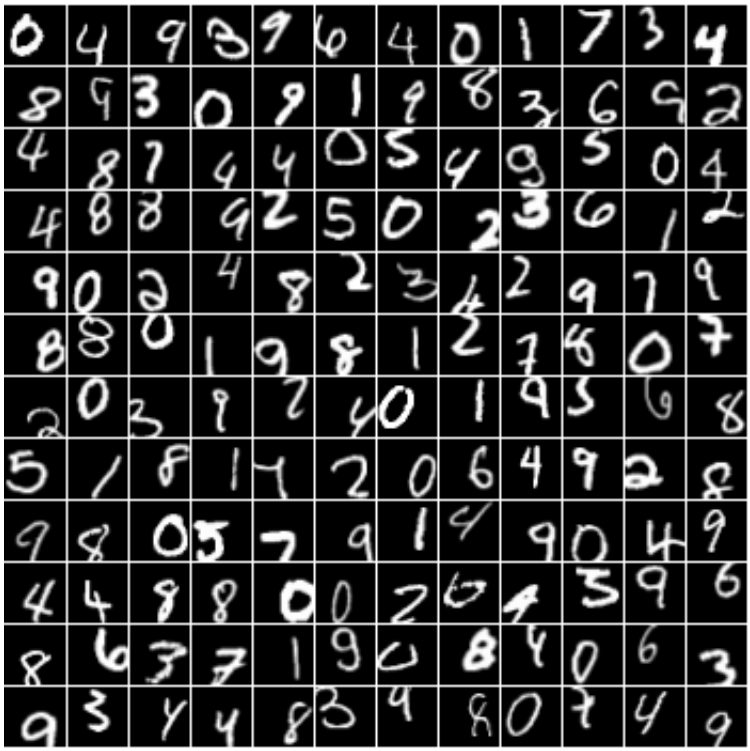

Figure B.3: errors made by the pairwise differences classifier when inputs are shifted by 3. Roughly 1450 errors were made, this is only a sample of them.

[6, 7, 7, 8, 1, 1, 9, 3, 8, 2, 2, 9],
[5, 5, 1, 5, 7, 4, 1, 7, 2, 5, 5, 5],
[8, 1, 1, 6, 9, 2, 3, 7, 3, 7, 1, 5],
[1, 6, 1, 4, 4, 3, 7, 1, 2, 2, 7, 4],
[1, 9, 3, 7, 7, 4, 2, 1, 4, 6, 8, 6],
[1, 3, 4, 7, 8, 9, 4, 4, 9, 6, 9, 3],
[5, 9, 5, 8, 4, 1, 7, 4, 7, 1, 4, 5],
[9, 7, 3, 4, 3, 7, 1, 1, 9, 8, 3, 5],
[7, 9, 1, 3, 9, 1, 4, 8, 7, 7, 5, 7],
[1, 5, 9, 9, 1, 9, 7, 7, 6, 5, 7, 2],
[9, 1, 9, 5, 4, 5, 7, 6, 7, 7, 1, 9],
[5, 5, 7, 9, 7, 2, 1, 4, 2, 1, 1, 5]

Figure B.4: Selected wrong labels corresponding to figure B.3.

Figure B.5: 121 errors made by the pairwise difference classifier trained on translations as well. These 121 errors can be compared to the 143 errors in figure B.1. The same net is trained slightly more here than in figure B.1.

```
Outputs
array([[8, 6, 0, 9, 2, 2, 7, 8, 1, 4, 8],
       [9, 0, 5, 3, 2, 9, 1, 7, 3, 3, 3],
       [3, 9, 3, 8, 7, 3, 4, 5, 9, 0, 9],
       [9, 1, 2, 8, 1, 8, 4, 1, 0, 4, 7],
       [9, 2, 5, 8, 3, 2, 9, 3, 9, 1, 8],
       [0, 5, 3, 3, 9, 2, 8, 4, 9, 8, 7],
       [8, 7, 7, 2, 7, 5, 9, 1, 8, 0, 5],
       [9, 8, 4, 9, 4, 9, 4, 8, 3, 8, 3],
       [9, 9, 8, 3, 5, 5, 5, 5, 3, 2, 9],
       [1, 7, 6, 1, 3, 6, 5, 2, 2, 2, 2],
       [8, 4, 3, 7, 3, 7, 5, 6, 9, 8, 7]])

Target
array([[9, 4, 6, 4, 8, 8, 2, 2, 2, 7, 9],
       [8, 6, 6, 7, 7, 4, 7, 5, 8, 5, 2],
       [7, 7, 9, 4, 3, 9, 9, 6, 7, 2, 3],
       [4, 6, 1, 0, 9, 6, 2, 6, 9, 7, 4],
       [8, 3, 9, 6, 9, 1, 7, 7, 4, 9, 4],
       [5, 8, 9, 8, 5, 7, 7, 9, 7, 2, 9],
       [2, 9, 3, 3, 8, 6, 8, 6, 9, 8, 3],
       [4, 0, 8, 7, 6, 7, 1, 7, 5, 3, 5],
       [5, 3, 3, 9, 9, 9, 9, 9, 9, 9, 8],
       [7, 0, 0, 4, 2, 0, 8, 7, 7, 7, 7],
       [9, 9, 0, 2, 6, 9, 6, 5, 4, 2, 9]],
```

Figure B.6: Selected wrong labels and correct labels corresponding to figure B.5.

Appendix C

Reproducing results in this thesis

C.1 Prerequisites

To run my code, one needs to obtain cudamat [1] and gnumpy [2].

cudamat: http://code.google.com/p/cudamat/

gnumpy: http://www.cs.toronto.edu/~tijmen/gnumpy.html

which in turn requires npmat (see documentation on the gnumpy page). Some more general prerequisites are numpy, scipy, matplotlib which can be found in apt on Ubuntu.

CUDA is a general requirement if one wants to take advantage of GPU computing. My code is only tested on CUDA 3.0 and CUDA 3.2

C.2 Obtaining the code and weights

One can find links to python code, weights and documentations at http://www.cs.toronto.edu/~sidaw/thesis. This site is subject to change. In case this site goes down, contact the author. The author should welcome other communications about this research as well.

Few most important points.

- tnetg.py implements the transforming autoencoder. Use trainer.py to interact with tnetg.
- ff.py implements the classification stuff. Use classmnist.py to interact with ff.py.

References

[1] V. Mnih, "Cudamat: a CUDA-based matrix class for python," Tech. Rep. UTML TR 2009-004, Department of Computer Science, University of Toronto, November 2009. i, 42

[2] T. Tieleman, "Gnumpy: an easy way to use GPU boards in Python," Tech. Rep. UTML TR 2010-002, University of Toronto, Department of Computer Science, 2010. i, 42

[3] C. P. Papageorgiou, M. Oren, and T. Poggio, "A general framework for object detection," *Computer Vision, IEEE International Conference on*, vol. 0, p. 555, 1998. 1

[4] D. G. Lowe, "Object recognition from local scale-invariant features," in *ICCV*, pp. 1150–1157, 1999. 1, 5

[5] C. M. Bishop, *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer, 1 ed., 2007. 1, 10, 33

[6] D. H. Ackley, G. E. Hinton, and T. J. Sejnowski, "A learning algorithm for Boltzmann Machines," *Cognitive Science*, vol. 9, pp. 147–169, 1985. 1

[7] M. Antonini, M. Barlaud, P. Mathieu, and I. Daubechies, "Image coding using the wavelet transform," *IEEE Trans. on Image Processing*, vol. 2, Apr. 1992. 1

[8] J. G. Daugman, "Complete discrete 2-D gabor transforms by neural networks for image analysis and compression," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 36, pp. 1169–1179, July 1988. 1

[9] G. E. Hinton and L. A. Parsons, "Frames of reference and mental imagery," in *Attention and Performance IX*, pp. 261–277, Hillsdale, NJ: Erlbaum, 1981. 2, 21

[10] G. E. Hinton and R. R. Salakhutdinov, "Reducing the dimensionality of data with neural networks," *Science*, vol. 313, no. 5786, pp. 504–507, 2006. 2

[11] Y. L. Cun and Y. Bengio, "Convolutional networks for images, speech, and time series," in *The Handbook of Brain Theory and Neural Networks* (M. A. Arbib, ed.), pp. 255–258, Cambridge, Massachusetts: MIT Press, 1995. 5

[12] R. Memisevic and G. E. Hinton, "Learning to represent spatial transformations with factored higher-order boltzmann machines," *Neural Computation*, vol. 22, no. 6, pp. 1473–1492, 2010. 5, 6, 7

[13] J. B. Tenenbaum and W. T. Freeman, "Separating style and content with bilinear models," *Neural Comput.*, vol. 12, pp. 1247–1283, June 2000. 5, 7

[14] Y. Lecun, L. Jackel, L. Bottou, A. Brunot, C. Cortes, J. Denker, H. Drucker, I. Guyon, U. Muller, E. Sackinger, P. Simard, and V. Vapnik, "Comparison of learning algorithms for handwritten digit recognition," June 14 1995. 5, 20

[15] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *ICML* (J. Fürnkranz and T. Joachims, eds.), pp. 807–814, Omnipress, 2010. 6

[16] U. M. L. M. G. D.C. Ciresan and J. Schmidhuber, "Deep big simple neural nets excel on handwritten digit recognition," *CoRR*, vol. abs/1003.0358, 2010. informal publication. 19

[17] J. Martens, "Deep learning via hessian-free optimization," in *ICML* (J. Fürnkranz and T. Joachims, eds.), pp. 735–742, Omnipress, 2010. 26

[18] G. E. Hinton and R. S. Zemel, "Autoencoders, minimum description length and helmholtz free energy," in *Advances in Neural Information Processing Systems 6*, pp. 3–10, Morgan Kaufmann, 1994. 27

[19] D. J. C. MacKay, *Information Theory, Inference, and Learning Algorithms*. Cambridge University Press, 2003. Available from `http://www.inference.phy.cam.ac.uk/mackay/itila/`. 32