Uniwersytet Jagielloński
Wydział Matematyki i Informatyki
INSTYTUT INFORMATYKI
Studia dzienne

Nr indeksu: 1082571

Bartosz Witkowski

Klasyfikacja obrazów za pomocą autoenkoderów.

Opiekun pracy magisterskiej:
dr Bartosz Zieliński

Opracowano zgodnie z Ustawą o prawie autorskim i prawach pokrewnych z dnia 4 lutego 1994 r. (Dz.U. 1994 nr 24 poz. 83) wraz z nowelizacją z dnia 25 lipca 2003 r. (Dz.U. 2003 nr 166 poz. 1610) oraz z dnia 1 kwietnia 2004 r. (Dz.U. 2004 nr 91 poz. 869)

Kraków 2013

Jagiellonian University
Faculty of Mathematics and Computer Science
INSTITUTE OF COMPUTER SCIENCE

Bartosz Witkowski

Autoencoders for image classification.

Supervisor:
Bartosz Zieliński PhD

Kraków 2013

Contents

Nomenclature ii

Introduction iii

1 Theoretical Background 1
1.1 Image Recognition and Learning Models 1
1.2 Artificial Neural Networks 1
1.3 Deep Neural Networks and Deep Learning 3
1.4 Autoencoders and Sparsity 3
1.5 Convolutional Networks 5
1.5.1 Convolution 5
1.5.2 Pooling 7
1.5.3 Convolutional and Pooling Layers 8
1.6 Related Work 8

2 Problem Formulation 9
2.1 MNIST 9

3 Methodology 12
3.1 Training Parameters and Network Topology 12
3.2 Architecture-I – Stacked Autoencoders 12
3.2.1 Greedy layerwise training 12
3.2.2 Training Protocol 13
3.2.3 Training Parameters and Methods 14
3.3 Architecture-II – Stacked Convolutional Autoencoders 15
3.3.1 Training Protocol 15
3.3.2 Training parameters and Methods 18
3.4 Visualizing features 18

4 Results 19
4.1 Unsupervised Learning 19
4.2 Supervised Learning 19

5 Conclusions and Future Work 23

Nomenclature

$W^{(l)}$ the weight matrix associated with layer l.

$b^{(l)}$ the bias vector associated with layer l.

s_l the size of the layer l.

n_Ω the number of layers of a neural network.

x the input set.

y the label set.

$x^{(i)}$ i-th input vector.

$y^{(i)}$ i-th label vector.

$a^{(l)}$ activation of layer l.

$\sigma(\bullet)$ sigmoid activation function.

$g(\bullet)$ softmax activation function.

$J(\bullet)$ the model error function.

$h_{W,b}(\bullet)$ the hypothesis of the model.

λ the weight decay parameter.

ρ the sparsity parameter.

$\hat{\rho}$ the empiric sparsity of a network.

β the sparsity weight.

Δ stopping parameter.

ConvNet convolutional network

t_i non-labeled training data for the n-stage deep network.

$\hat{t}_i$ non-labeled training data for the n-stage autoencoder in Architecture-II.

Introduction

Computer Vision problems are among the better studied problems of machine learning. The most common dilemma of these methods is the choice of features we use to train the classifier. In this paper we describe how to implement two architectures of neural-network based classifiers that find good features automatically. The classifiers are tested on a very popular digit recognition benchmark and compared to state-of-the-art results as well as naive classification methods that do not use features.

The aim of this work was twofold. The first goal was to show that the methodologies similar to the ones used in [LRM+12] can be applied to solve machine learning problems on commodity hardware. The second goal was to implement a deep-learning algorithms from scratch. Both of those goals were completed successfully.

This thesis consists of five chapters. The first chapter introduces the theory and motivation behind deep learning and techniques used in this work. In the second chapter we present the data set used to validate the accuracy of the learning models introduced in the next chapter. Thus, the third chapter presents the architectures of two learning models, their parameters and training methodology. The fourth chapter presents an analysis of the obtained results and compares them to state-of-the-art and naive training methods. The fifth chapter summarizes this thesis and discusses possible future work. Finally, the implementation details of the library and brief usage guide is given in the appendix.

Keywords: Computer vision, machine learning, classification, neural networks, deep learning, convolutional networks, pooling, MNIST.

Chapter 1

Theoretical Background

1.1 Image Recognition and Learning Models

Image recognition problems such as handwritten digit classification require the interpretation of images. Usually, computer algorithms used for image interpretation do not base on raw pixel values but use some intermediate higher level representation [Ben09, p. 3] such as SIFT [Low99] or HoG [DT05]. The same algorithm of decomposing a problem into subproblems was observed in humans as a way to solve complicated tasks [Ben09, p. 3] [Hin10, p 177].
Let $X \in \mathbb{R}^p$ denote a real valued random input vector and $Y \in \mathbb{R}$ a real valued random output variable. A **learning model**, or a predictive model, is a function $f(X)$ that predicts Y based on X given as an input [HTF09, p. 18]. We can judge how well a learning model works using a loss (or error) function which penalizes errors in prediction.
We will consider a learning model which uses decomposition to solve image recognition tasks. But first we need to introduce artificial neural networks which the models are based on.

1.2 Artificial Neural Networks

The term **neural network** encompasses a large class of models and learning methods [HTF09, p. 392]. The central idea is to extract linear combinations of the inputs as derived features and then model the target as a nonlinear function of these features [HTF09, p. 389].
A feedforward neural network of depth n is a n-stage regression or classification model, typically represented by a network diagram [HTF09, p. 392] such as the one in figure 1.1. Stages in the learning model correspond to layers in the neural networks. The units of the neural network, or neurons, are connected to each other layer-wise.
We will represent those connections as a matrix of weights $W^{(L)}$. Where $W_{ij}^{(L)}$ is the weight of the connection between the unit i in the layer L and unit j in the layer $L+1$. We will assume that the layer $W^{(1)}$ is at the bottom and the layer with the highest index is at the top. Each layer in the neural network has a size (the number of neurons) denoted as s_i where $i = 1 \dots n_\Omega$ and n_Ω is the index of the last layer. We also add a 'special' bias unit to each layer which always equals 1.

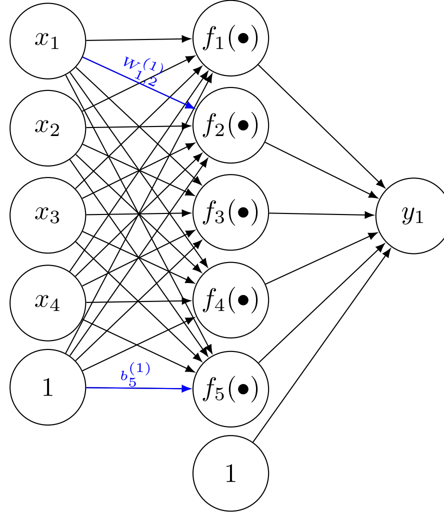

Figure 1.1: A feedforward neural network with 4 input units, 5 hidden units, and 1 output unit. Biases are represented as additional units with constant input set to 1. The layer sizes are: $s_1 = 5$, $s_2 = 6$, $s_3 = s_{n_\Omega} = 1$. The weight $W^{(1)}_{1,2}$ and the bias $b^{(1)}_5$ weight are highlighted. f_i $i = 1 \dots 5$ correspond to activation functions (see below).

When modeling K-class classification with neural networks [1] , the resulting network should have K units at the top layer, with the k-th unit modeling the probability that the input belongs to class k [HTF09, p. 392]. The units in the middle layers of the network are called hidden units. [HTF09, p 394].
The outputs of layer l are called activations and are computed based on linear combinations of inputs and the bias unit[HTF09, p. 392] in the following way:

$$a^{(l)} = f^{(l)}(z^{(l)})$$

where

$$z^{(1)} = W^{(1)}x + b^{(1)}$$
$$z^{(n)} = W^{(n)}a^{(n-1)} + b^{(n)} \quad \forall n = 2 \dots n_\Omega$$

, $W^{(l)}$ is the matrix of weights of layer l, and $b^{(l)}$ is the vector of bias weights (or simply biases) of layer l, and the function $f^{(l)} : \mathbb{R}^{s_l} \to \mathbb{R}^{s_l}$ is an activation function. Althought in theory we could use different activations functions for each neuron (like shown in figure 1.1) in practice only one type of activation function for each layer is used. We will use two types of activation functions, a non linear sigmoid activation function:

$$\sigma(z)^{(l)}_i = \frac{1}{1 + e^{-z_i}} \quad \forall i = 1 \dots s_l, l = 1 \dots n_\Omega$$

and the soft-max activation function used as the last layer (classifier) for K-class classification:

$$g(z_k) = \frac{e^{z_k}}{\sum_{i=1}^{K} e^{z_i}} \quad \forall k = 1 \dots K$$

where z is a vector, and k is the class index. A neural network with an input layer and a softmax layer (a layer with the softmax activation function) forms a **softmax classifier**.

[1] Classification where the possible number of classes is K

A neural network has to be trained (weights and biases have to be found) to fit the training data. Typically, when training feedforward networks we use an average sum-of-squared errors as an error function which has to be minimized [HTF09, p. 395]:

$$J(W,b) = \frac{1}{m}\sum_{i=1}^{m}\left(\frac{1}{2}\left\|h_{W,b}(x^{(i)}) - y^{(i)}\right\|^2\right)$$

where m is the training set size, $x^{(i)}$ is the i-th training sample with the label $y^{(i)}$, and $h_{W,b}(\cdot)$ returns the output of last layer of the network.
To prevent the network from overfitting the training data we add a regularization term controlled by λ (called **weight decay**). It has been shown that such a factor keeps the weights small and decreases overfitting [HTF09, p. 398].
The error function with regularization applied to the last layer looks as follows:

$$J_{\text{regularized}}(W,b) = \left[\frac{1}{m}\sum_{i=1}^{m}\left(\frac{1}{2}\left\|h_{W,b}(x^{(i)}) - y^{(i)}\right\|^2\right)\right] + \frac{\lambda}{2}\sum_{i=1}^{s_{n_\Omega - 1}}\sum_{j=1}^{s_{n_\Omega}}\left(W_{ji}^{(n_\Omega - 1)}\right)^2$$

The gradient of the error function (needed for optimization) can be calculated using the chain rule for differentiation [HTF09, p. 395]. Using the stochastic gradient descent algorithm (sometimes called backpropagation algorithm in the context of neural networks [HTF09, p. 395]), or some other optimization method we can find the minimum of the error function, and thus the best parameters for the network.

1.3 Deep Neural Networks and Deep Learning

As it turns out, deep (meaning with more then 1-2 hidden layers) vanilla[2] neural networks perform *worse* than neural networks with one or two hidden layers [Ben09, p. 6, 31]. While in theory deep neural networks have at least the same expressive power as shallow neural networks [Ben09, p.32], and can learn complex hierarchies of features, in practice we find that they are prone to being stuck in local minima during the training phase. That's why deep vanilla neural networks are not appropriate candidates for a learning model that uses decomposition.
Fortunately, in 2003 Hinton et al. introduced Deep Belief Networks [HOT06] which use Restricted Boltzmann Machines - a variant of neural networks with stochastic binary units [HOT06, p. 2-3] that overcome the training problems of vanilla neural networks. For this purpose, they proposed a layerwise training algorithm which greedily trains one layer at a time - which exploits an unsupervised learning algorithm for each layer [HOT06, p 5]. The details of the greedy layerwise training protocol will be discussed in chapter 3. Shortly after, related algorithms based on auto-encoders were proposed [Ben09, p. 6].

1.4 Autoencoders and Sparsity

An autoencoder is a neural network (see Fig. 1.2) that is trained to encode an input x into some representation $c(x)$ so that the input can be reconstructed from that representation [Ben09, p. 45]. While the identity function - copying inputs to outputs - is not a useful function to learn, we can add additional constraints on either the network, or the error criterion in hopes that the network captures some useful features of the input [Ben09, p. 45] (in contrast a vanilla neural network with $s_1 \leq s_2$ would probably just learn the identity function).

[2]Neural networks trained using only backpropagation.

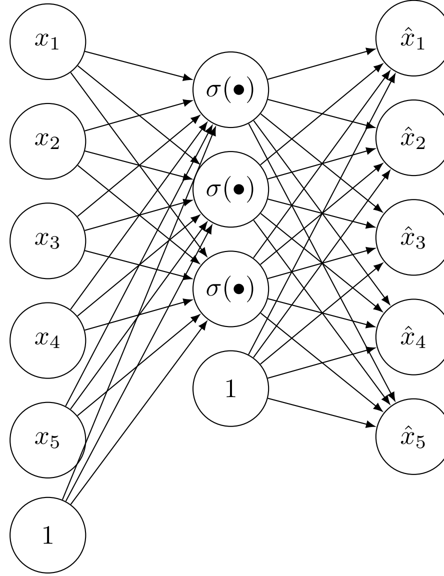

Figure 1.2: An example of an autoencoder with 5 input units and 3 hidden units (features).

After successful[3] training, the autoencoder it should decompose the inputs into a combination of hidden layer activations. In this context we say that a trained autoencoder has learned **features**. The amount of features is equal to the amount of neurons in the hidden layer.
One possible constraint is to enforce network sparsity. We can measure the average activations of the neurons in the second layer:

$$\hat{\rho}_j = \frac{1}{m}\sum_{i=1}^{m}\left[a_j^{(2)}(x^{(i)})\right] \quad \forall j = 1 \dots s_2$$

and add a penalty to the error function which will prevent the activations from straying too far from some desired mean activation ρ (the **sparsity parameter**). The resulting autoencoder is called a **sparse autoencoder**.
One candidate for the penalty function is the Kullback-Leibler divergence:

$$\mathrm{KL}(\rho||\hat{\rho}) = \sum_{j=1}^{s_2} \rho \log \frac{\rho}{\hat{\rho}_j} + (1-\rho)\log\frac{1-\rho}{1-\hat{\rho}_j}$$

which penalizes ρ from straying too far from $\hat{\rho}$ as seen in figure 1.3.

[3]The training may be unsuccessful if the input data is random and highly uncorrelated - the network will not deduce any useful generalization in that case.

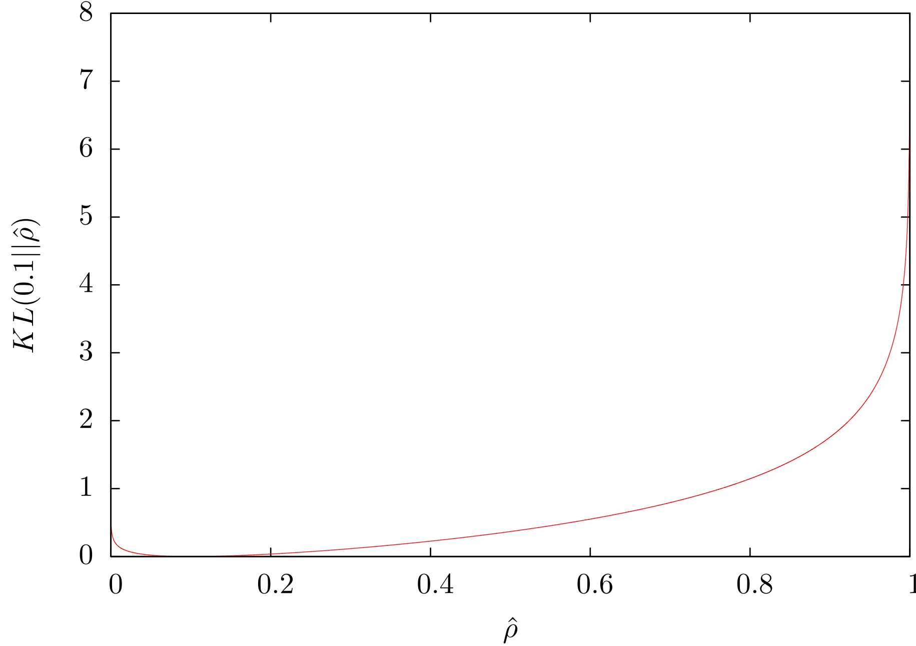

Figure 1.3: The KL penalty function for $\rho = 0.1$

An error function which uses both the sparsity constraint and regularization can be expressed as:

$$J_{\text{sparse}}(W, b) = J_{\text{regularized}}(W, b) + \beta \cdot \text{KL}(\rho||\hat{\rho})$$

where β is called the **sparsity constraint** and controls the sparsity penalty.
We can train layers of autoencoders using the greedy layerwise training algorithm similar to training deep belief networks: we train the first autoencoder using unsupervised learning, then we use the hidden layer activations as inputs for the second layer, and so forth.

1.5 Convolutional Networks

Another type of deep neural networks that perform better then vanilla neural networks are convolutional neural networks or ConvNets [Ben09, p 43]. ConvNets are inspired by the human visual systems structure and work by exploiting local connections [Ben09, p. 43-44] through two operations - convolution and sub-sampling/pooling.

1.5.1 Convolution

"Convolutional neural networks are organized in layers of two types: convolutional layers and sub-sampling layers. Each layer has a topographic structure, i.e., each neuron is associated with a fixed two- dimensional position that corresponds to a location in the input image, along with a receptive field (the region of the input image that influences the response of the neuron). At each location of each layer, there are a number of different neurons, each with its set of input weights, associated with neurons in a rectangular patch in the previous layer. The same set of weights, but a different input rectangular patch, are associated with neurons at different locations. [...] One untested hypothesis is that the small fan-in of these neurons (few inputs per neuron) helps gradients to propagate through so many layers without diffusing so much as to become useless." [Ben09, p. 44]

The convolution operation can be imagined as convolving an image with a neural network (or a single layer) - analogously to convolving an image with some function. The windowed input is fed to the neural network:

$$\begin{aligned}
&\forall i = 1 \dots \mathrm{width}(\mathrm{sample}) \cdot \mathrm{height}(\mathrm{sample}) \\
&\forall r = 1 \dots Y \\
&\forall c = 1 \dots X \\
\hat{x}(r,c)_i &= \mathrm{image}\left(r + \left\lfloor \frac{i}{\mathrm{width}(\mathrm{image})} \right\rfloor, c + \mathrm{mod}(i, \mathrm{width}(\mathrm{image}))\right) \\
a_{r,c} &= \mathrm{nnet}(\hat{x}(r,c))
\end{aligned}$$

producing an output matrix of activation vectors (illustrated in figure 1.4):

$$\begin{bmatrix}
a_{1,1} & a_{2,1} & a_{3,1} & \dots & a_{X,1} \\
a_{1,2} & a_{2,2} & a_{3,2} & \dots & a_{X,2} \\
\vdots & & & \ddots & \dots \\
a_{1,Y} & a_{2,Y} & a_{3,Y} & \dots & a_{X,Y}
\end{bmatrix}$$

where:

$$\begin{aligned}
X &= \mathrm{width}(\mathrm{image}) - \mathrm{width}(\mathrm{sample}) + 1 \\
Y &= \mathrm{height}(\mathrm{image}) - \mathrm{height}(\mathrm{sample}) + 1,
\end{aligned}$$

$\mathrm{mod}(x, y)$ is the result of modulo division, and nnet(x) is a function that returns the activation of the last layer of a neural network given the input.
Once again we assume that s_1 is the size of the input layer of the ConvNet, s_{n_Ω} is the size of the output layer with n_Ω being its index. Additionally, both sample and image are rectangular regions but the input itself is flat (a vector of $\mathbb{R}^{s_1}$) and has to be converted. The conversion function will be called **flatten**.

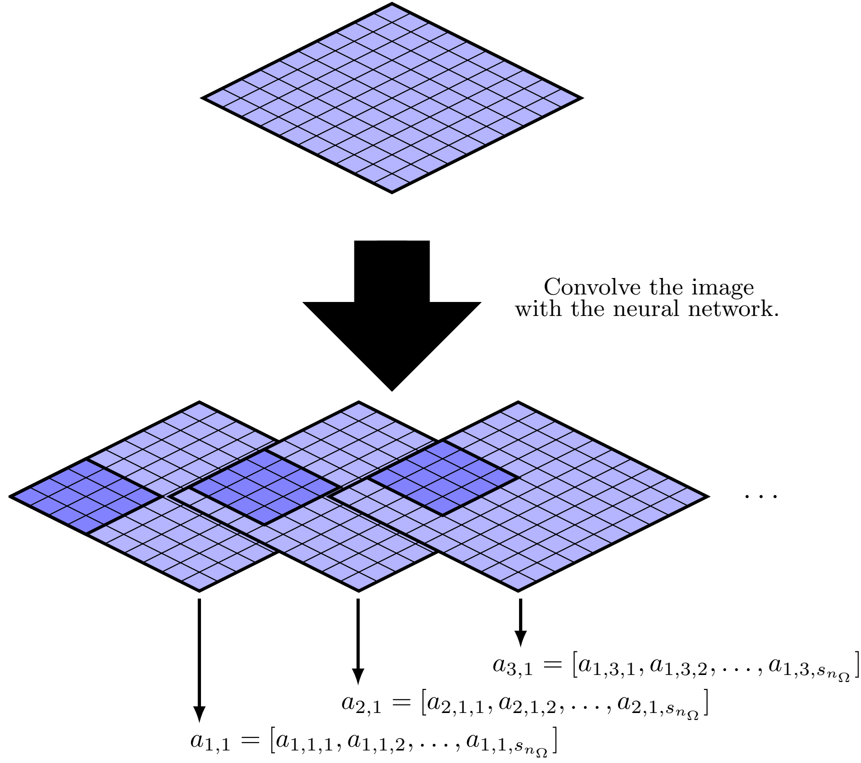

Figure 1.4: Example of the convolution process. The vector of activations $a_{i,j}$ corresponds to the output of the neural network and consists of values $a_{i,j,1}$ through $a_{i,j,s_{n_\Omega}}$

1.5.2 Pooling

Another operation used by ConvNets is pooling (See Fig 1.5). Pooling is a biologically inspired operation that reduces the dimensionality of the input [SCL12, p. 2]. Consider a matrix I: $m \times n$, pooling applies some operation to the values of the matrix in a spacial neighborhood. For a rectangular neighborhood $r \times r$ the output matrix size will be $\lfloor \frac{m}{r} \rfloor \times \lfloor \frac{n}{r} \rfloor$ [4]. A single cell of the output matrix is calculated as follows:

$$\forall y = 1 \ldots \left\lfloor \frac{m}{r} \right\rfloor$$

$$\forall x = 1 \ldots \left\lfloor \frac{n}{r} \right\rfloor$$

$$O_{y,x} = \lim_{p \to P} \left(\sum_{i=1}^{r} \sum_{j=1}^{r} I\left(i + yr, j + xr\right)^p \times G(i,j) \right)^{\frac{1}{p}}$$

where G is the gauss kernel I is the input matrix. The actual implementation uses $P = \infty$ which corresponds max operation (see also footnote 4, and figure 1.5). This variant of pooling is called L_p-pooling.

[4] The actual implementation simulates max-pooling by using the maximum of activations (for efficiency reasons the kernel operation is omitted). Because of this it is possible to preserve the edges and the actual size is $\lceil \frac{m}{r} \rceil \times \lceil \frac{n}{r} \rceil$

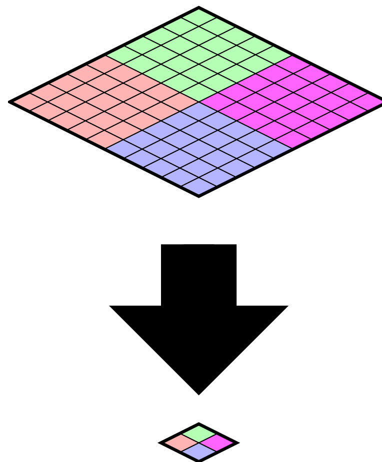

Figure 1.5: The pooling operation decreases the amount of features of each input source region (also called the receptive field).

This decreases the number of trainable parameters and exploits locality - only the most prominent features from a region are picked and given as the output.

1.5.3 Convolutional and Pooling Layers

Conceptually pooling and convolution may be modeled by layers: the convolution operation can be viewed as a not-fully connected layer[5]. Pooling can also be expressed as a not fully connected layer – where the L_p-pooling operation is the activation function, and the output layer size is dependent on pooling size.

1.6 Related Work

The methods and architectures used in this work are heavily inspired by two papers: "Building High-level Features Using Large Scale Unsupervised Learning" [LRM$^+$12] and "Convolutional Neural Networks Applied to House Numbers Digit Classification" [SCL12]. In this paper we use smaller datasets and simpler algorithms. By this virtue the presented methods are more suited to experimentation on commodity hardware. Especially in contrast to the team behind [LRM$^+$12] which used a cluster consisting of 1000 computers with 16 cores each. One of the objectives of this work is to show that deep learning using simple architectures and small datasets is possible without clusters of high-end computers.

[5]And non-fully-connected layers can be viewed as layers with some weights always equal to 0

Chapter 2

Problem Formulation

2.1 MNIST

The MNIST [LC] data set of handwritten digits is a very popular benchmark for image recognition tasks. It has a training set of 60,000 examples, and a test set of 10,000 examples. The images are 28×28 in size, the amount of images belonging to a specific class is presented in table 2.1.

k	training	test
0	5923	980
1	6742	1135
2	5958	1032
3	6131	1010
4	5842	982
5	5421	892
6	5918	958
7	6265	1028
8	5851	974
9	5949	1009

Table 2.1: The amount of images belonging to the class k in the training and test sets.

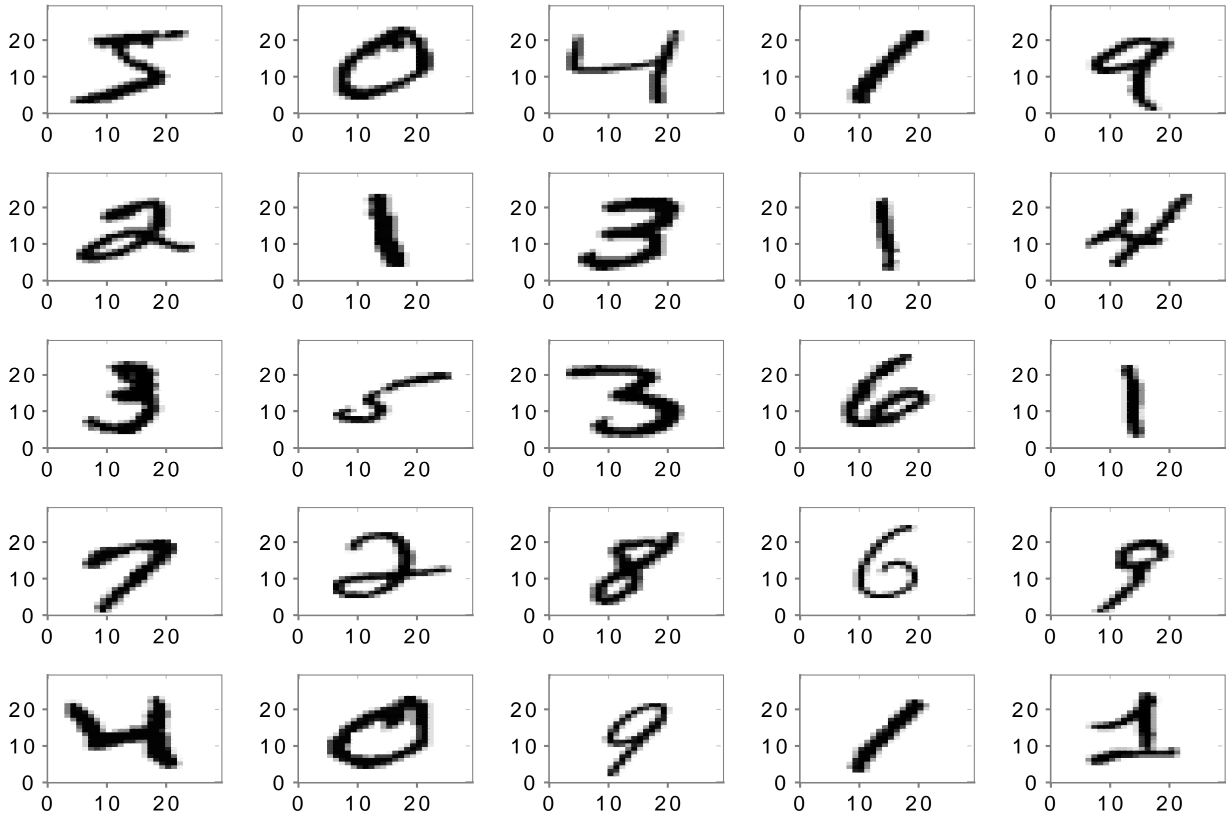

Figure 2.1: Samples from the MNIST training set. The number on the axes correspond to pixel coordinates.

We will consider the quality of digit classification on the MNIST data set using the provided test set. The classification accuracy will be expressed as a ratio of correctly classified digits to the size of the test set.
Because of its popularity the MNIST data set is a de facto standard benchmark for learning methods. The official MNIST website [mni] contains a list of learning models along with a error rate on the provided test set. Some of those methods worked on a different version of the data set: *"Some of those experiments used a version of the database where the input images where deskewed (by computing the principal axis of the shape that is closest to the vertical, and shifting the lines so as to make it vertical). In some other experiments, the training set was augmented with artificially distorted versions of the original training samples. The distortions are random combinations of shifts, scaling, skewing, and compression."* [mni]. Those experiments may have a better performance compared to methods that did not use an augmented test set or data preprocessing (a common heuristic to improve a model performance is to increase the amount of training data).
Currently the state-of-the-art method is a committee of 35 ConvNets with the architecture that starts with an input layer, followed by a convolutional layer with 20 maps (the amount of convolutions), followed by a pooling layer, a convolutional layer with 40 maps, a pooling layer, a fully connected layer with 150 neurons and an output layer with 10 outputs (one per class). For future reference the network will be summarized as: 1-20-P-40-P-150-10. The networks were trained on a training set with elastic distortions and width normalization *We perform experiments on the original and six preprocessed datasets. [...] we [...] normalize the width of all characters to 10, 12, 14, 16, 18 and 20 pixels* [CMGS11]. Each member of the committee was trained on different image

resolution, the whole model was trained on a super-computer with 4 GPUs.
We will also compare the results of this thesis to simpler networks such as a 2 layer neural network with 300 hidden units, a softmax classifier and a linear classifier (which can be thought of as a one layer neural network with a linear activation function).
The results of the above methods are shown in chapter 4, along with our results.

Chapter 3

Methodology

In this chapter we will describe the architectures and methods used to classify the images from the database presented in the previous chapter.

3.1 Training Parameters and Network Topology

The parameters of both architectures were chosen heuristically – the sparsity parameter ρ, sparsity weight b, and weight decay parameter λ was chosen after some initial experiments on a small set of the MNIST data. The stopping criterion Δ was chosen after training the first autoencoder for Architecture-I – and motivated mostly by diminishing returns in further optimisation.
The topology for Architecture-I was motivated mostly by ease of training autoencoders – 200 deemed a good number of features for 28×28 images, and in practice training the autoencoders wasn't very time consuming.
The topology for Architecture-II was chosen to preserve as much data as possible while at the same time presenting the methodology. Subimages of size 7×7 were chosen over size 5×5 for the first stage autoencoder based on softmax classifier performance on the training data.
Further information about the parameters and topology is given in the following sections.

3.2 Architecture-I – Stacked Autoencoders

First, we consider a simple architecture built of stacked autoencoders. The fully trained model built by stacking parts of autoencoders can be summarized as a 784–200–200–200–10 deep network (see figure 3.6).

3.2.1 Greedy layerwise training

The greedy layerwise training protocol can be summarized as follows: to construct a deep pretrained network of n layers we divide the learning into n **stages**. In each stage we exploit an unsupervised training algorithm: in the first stage we train an autoencoder on the provided training data sans labels. After traning the resulting autoencoder will learn the features of this data. Next we map the training data to the feature space (in practice by cutting out the last layer of autoencoder so that inputs are mapped to hidden layer activations – features). The mapped data is then used to train the next stage autoencoder. The training follows layer by layer until the last one. The last layer is trained as a classifier (not as an autoencoder) using supervised learning.

3.2.2 Training Protocol

The first stage autoencoder (see figure 3.1) is trained on t^*: the first 30000 images (out of 60000) taken from the training set x (which we will name t_1).

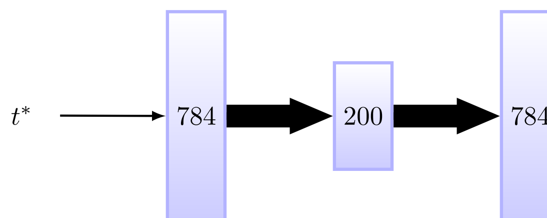

Figure 3.1: Training the first stage autoencoder

After training the first network we strip out of the last layer obtaining the network n_1 (see figure 3.2). We then obtain the activations of the network for the next stage training data (we call them t_2 see figure 3.3).

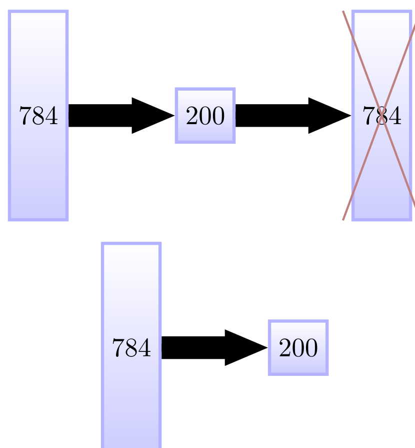

Figure 3.2: The network n_1

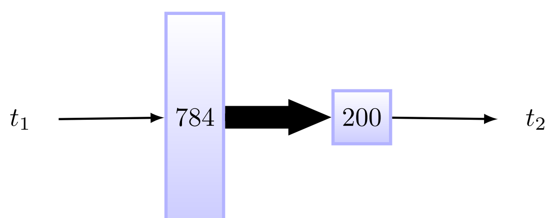

Figure 3.3: Obtaining the training data t_2

Similarly as before, we use the training data t_2 to train a new neural network n_2 with 200 input units, and 200 feature units (and 200 output units which will be stripped out after the unsupervised training phase). Once again, we map the training data t_2 to the network activations resulting in training data t_3 (see figure 3.4).
We continue this protocol for the third stage of the network, obtaining the network n_3 and training data t_4 (see figure 3.4).

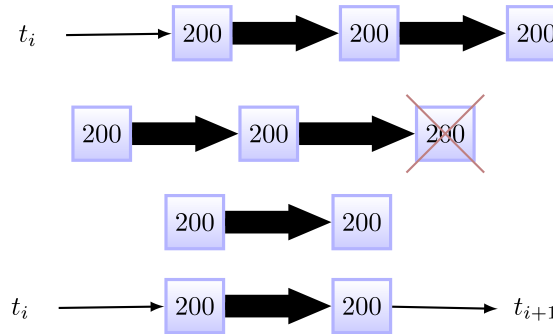

Figure 3.4: Training, and training data acquisition for stages 2 and 3 ($i = 2, 3$)

For the final stage of the network we concatenate the training data t_4 with the corresponding labels from the training set and train a softmax classifier n_4 with 200 input units and 10 output units (see figure 3.5).

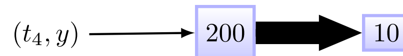

Figure 3.5: Training the softmax classifier.

After training the last stage, the networks n_1 through n_4 are stacked to form a deep neural network. It turns out that, although training deep networks is very hard and prone to be stuck in local optima, the network that has been pre-tranied by the greedy layerwise training algorithm doesn't suffer as much from the non-convexity. We use the full training set (images and labels) to train the deep neural network – we call this final step **fine-tuning** (see figure 3.6).

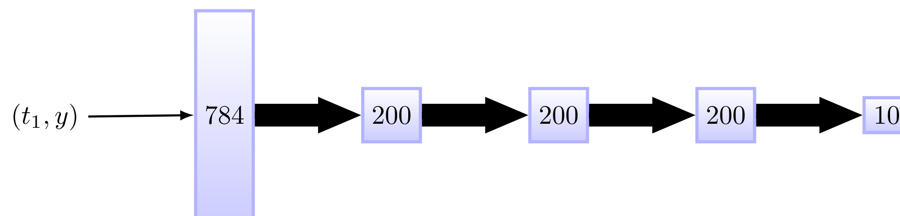

Figure 3.6: Fine-tuning the deep network.

3.2.3 Training Parameters and Methods

All autoencoders were trained using the sparsity parameter $\rho = 0.1$, sparsity weight $\beta = 3.0$, and the weight decay parameter $\lambda = 0.003$. Sparse backpropagation algorithm was implemented to find the network gradients and verified with numerical results obtained using the finite difference method[1].
Optimization of autoencoders was done with L-BFGS [Noc80] which is a quasi-Newton method of optimization with good memory characteristics. The networks were improved iteratively, and in

[1] A numeric method for finding the derivative of a function using the n-dimensional generalization of the approximation: $f'(x) \approx \frac{f(x+h)+f(x-h)}{2h}$

each iteration i the representation error e_i was measured. The optimization was stopped if criterion of $\Delta = |e_i - e_{i-1}| < 0.0001$ was satisfied.
The softmax layer, as well as the final deep network were optimized with the help of L-BFGS algorithm and with the stopping criterion of $\Delta < 0.0001$. As previously with the autoencoders, the gradients obtained by the optimization algorithms were verified against the numerical gradient using the finite differences method.
In addition to training the 784-200-200-200-10 (or three stage) model, simpler models were also trained for comparison - a two stage 784-200-200-10 model and a one stage 784-200-10 model.

3.3 Architecture-II – Stacked Convolutional Autoencoders

Architecture-I presented in the previous section suffers heavily when the size of the image becomes large e.g. when training on images of size 196×196, an autoencoder with 200 features has approximately $7.5 \cdot 10^6$ trainable parameters. This is almost 30 times more than the full 3 stage network from Architecture-I.
To alleviate this problem we propose an architecture that can scale well with the image size. Instead of training the network on the full image we can exploit local connectivity via convolutional networks, and additionally restrict the number of trainable parameters with the use of pooling.

3.3.1 Training Protocol

Note: We will reuse symbols for networks $n_i, i = 1, 2$ and training sets t_1. Additionally we discern between the 'full' training sets i.e. t_1 and autoencoder training sets i.e. $\hat{t}_1$.
To create the training set $\hat{t}_1$ (see Fig 3.7) we sample randomly from the MNIST training set and take a random 7×7 sub-images from the selected image. We draw 10000 of such sub-images and ensure that no duplicates were drawn.

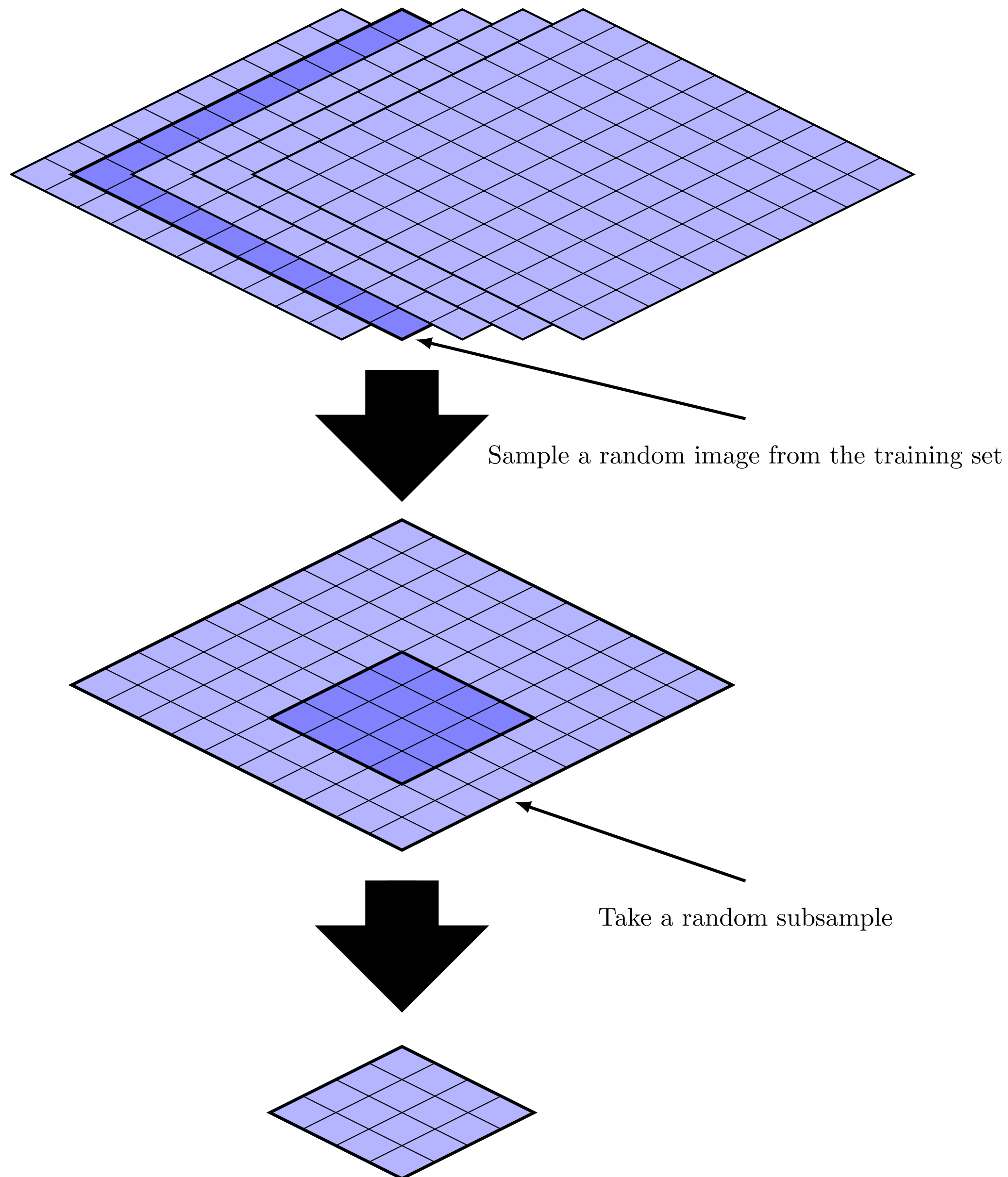

Figure 3.7: Sampling random sub-images for the first training set $\hat{t}^1$.

We then train the neural network on the t_1 set similarly like we did in architecture-I using 200 hidden neurons (see figure 3.8)

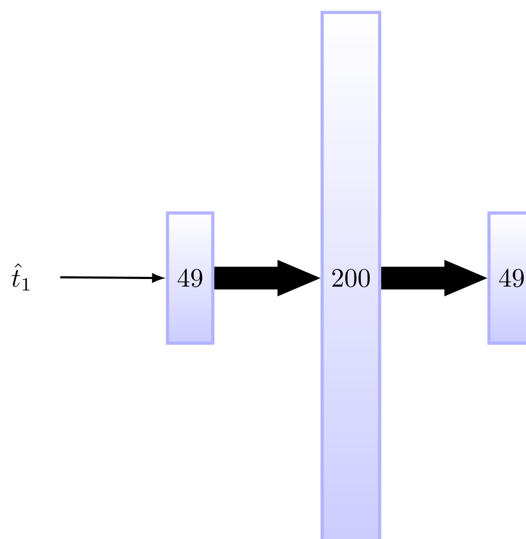

Figure 3.8: Training the first stage autoencoder. The network n_1 is built from the first two layers of the autoencoder (omitted for brevity).

To create the t_2 data we convolve the training data from the MNIST t_1 with the first stage autoencoder obtaining a set of 22×22 matrices. The data is then max-pooled with pooling-size 2 creating a set of 11×11 matrices (see figure 3.9). To create the autoencoder training data $\hat{t}_2$ random 5×5 sub-matrices are selected from the t_2. As before we sample 10000 such data points and avoid duplicates.

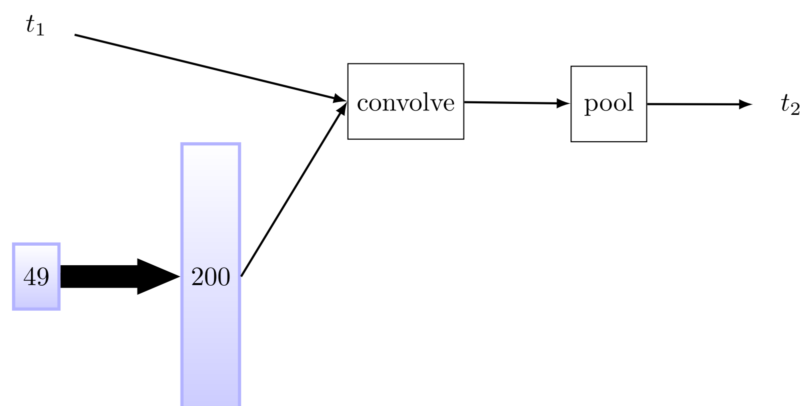

Figure 3.9: Obtaining the traning data t_2.

The next stage autoencoder is then trained with 100 hidden units. The data then passes through a max-pooling layer with pooling-size 2, shrinking it down to matrices with size 7×7. The final stage of the network consists of a softmax classifier similarly as in Architecture-I (see figure 3.10). The final matrix of activation is flattened to a single vector.

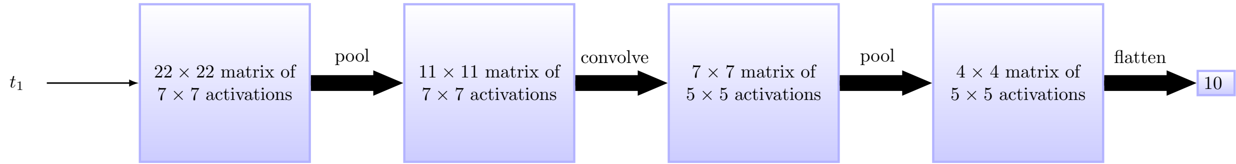

Figure 3.10: The full Architecture-II model.

As before the network needs to be fine-tuned after pre-training. **Unlike convolutional networks we keep a matrix of layer weights in the convolutional layer** i.e. the weight parameters aren't shared at different locations in the image. This is similar to the training method proposed by [LRM+12, p. 4].

3.3.2 Training parameters and Methods

To train the autoencoders for Architecture-II the same parameters and stopping criteria were used as in Architecture-I. Where possible, L-BFGS was used for optimization, but due to memory constraints the full networks were optimized using the momentum method[2] with batch-learning (the momentum coefficient was set to a standard 0.9).
In addition to training the two stage model, a one stage model was also trained for comparison.

3.4 Visualizing features

To gain a better understanding of the features learned by the first stage autoencoder we can visualize them by finding what features cause the neuron to activate. Activation of the hidden unit i is calculated as follows:

$$a_i(x) = f(\sum_j^{s_2} W_{ji}^{(1)} x_j + b_i^{(1)})$$

To visualize the neuron of an autoencoder we find an input x that maximizes the function a. To obtain a nontrivial answer to the above equation we constrain the input x by its norm $\|x\| \leq 1$. The input x that maximizes the activation of hidden unit i is given by setting x_j to [vis]:

$$x_j = \frac{W_{ij}^{(1)}}{\sqrt{\sum_{j=1}^{N} (W_{ij}^{(1)})^2}} \quad \forall j = 1, \ldots, N$$

[2]We use an axillary momentum vector which continually adds a fraction of the previous gradient to the current one. This improves convergence by diminishing random walks – see [IM02] for the improved version of the algorithm.

Chapter 4

Results

The experiments were performed on a machine with a Intel® Core™ i7-3632QM @ 2.2GHz CPU with 4 cores (8 threads due to hyper-threading), and 16 GB of RAM. The results for Architecture-I were calculated in a 24 hour period, but fully optimizing Architecture-II took about 3 weeks of real time work.

4.1 Unsupervised Learning

After training the autoencoders with full images we can visually verify the training quality. White regions of the visualization indicate positive correlation between the activation and input, while black regions indicate negative correlation. The gray regions can be thought of as "don't care" values. As seen in figure 4.1 when trained on full images the autoencoder learned to recognize shapes that are similar to pen strokes (which shows that the autoencoder decomposed the handwritten digits down to a very characteristic fragments). In contrast in figure 4.2 when trained on sub-images, the resulting autoencoder recognizes various edges of an image.

4.2 Supervised Learning

The results for both architectures were collected in table 4.1.

	Stages	Accuracy
Architecture I	1	95.16% (93.37%)
	2	97.05% (86.72%)
	3	98.35% (11.03%)
Architecture II	1	93.11% (91.08%)
	2	94.36% (89.99%)

Table 4.1: Classification accuracy for both architectures as measured on the MNIST test set. The number in parentheses indicate classification accuracy before fine-tuning.

Classifier type	Accuracy
Linear classifier	88.00%
Softmax classifier	91.57%
2-layer neural network 300 hidden units mean square error	95.30%
Committee of 35 ConvNet 1-20-P-40-P-150-10	99.77%

Table 4.2: Results for the state-of-the art and common learning methods.

We can compare the results to simple classifiers like a linear classifier (88% accuracy [LC]), or softmax classifier (91.57% accuracy) trained on raw pixel values (see table 4.2). The hypothesis from 1.1 that decomposition is key to better classification seems fully justified - the worst results from each architecture are still better then using raw data.
The state-of-the-art result for the MNIST data set has 99.77% accuracy [LC]. The results for Architecture-I are close to state-of-the-art performance and could probably be improved with careful choice of network parameters (the amount of layers and number of neurons in each layer, the sparsity parameter and sparsity weight, as well as the regularization parameter). As expected the classification accuracy increased with the number of stages of the network.
When comparing to other neural networks the results for Architecture-I are comparable to state-of-the-art training methods, and much better then simple vanilla neural networks (95.3% accuracy [LC]).
Classification accuracy of Architecture-II is comparable with vanilla neural networks. While it seems that Architecture-II only introduced complexity with little accuracy gain we must remember that the model was employed to alleviate problems with large scale images. Therefore, the model may fare better when used on larger images. Like in case of Architecture-I and as suspected - the results are better with more stages.

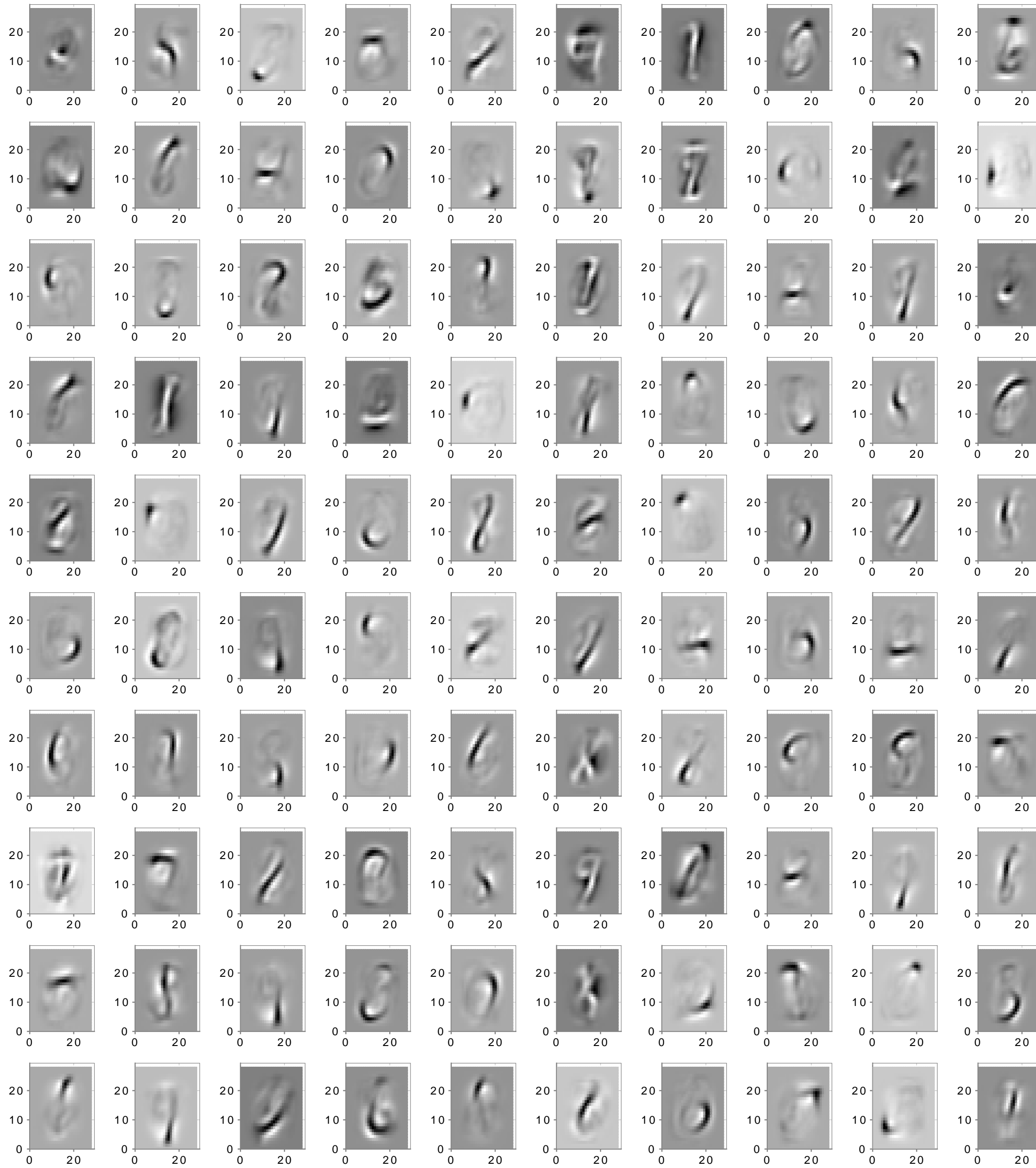

Figure 4.1: Visualization of the first 100 neurons of the autoencoder trained on 28×28 images from the MNIST training set. The numbers on the axes correspond to coordinates in the pixel domain.

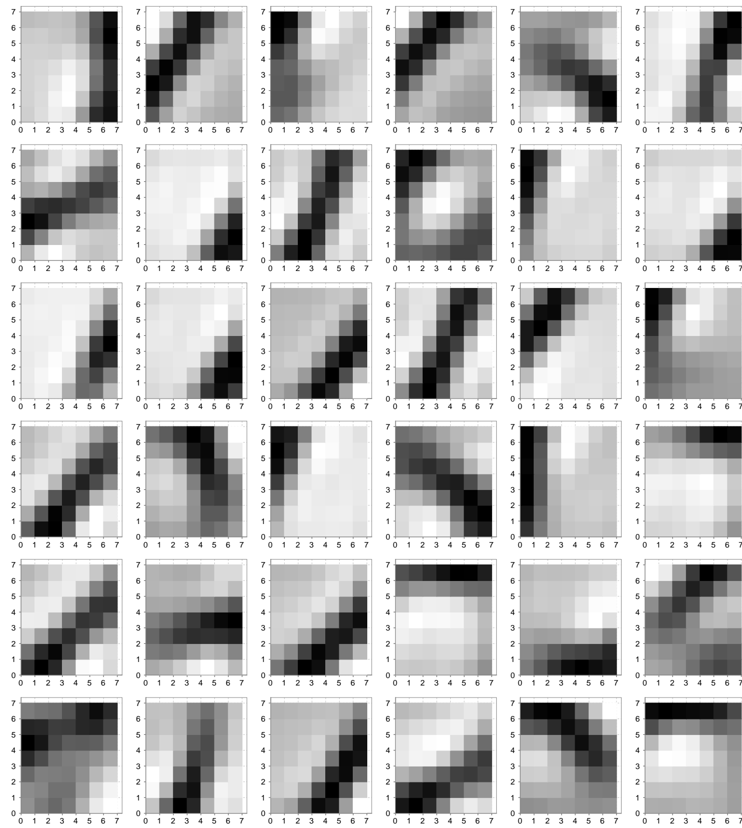

Figure 4.2: Visualization of the first 36 neurons of the autoencoder trained on 7×7 sub-images.The numbers on the axes correspond to coordinates in the pixel domain.

Chapter 5

Conclusions and Future Work

In this work we explored the possibility of using deep learning on commodity grade hardware. The results for Architecture-I are close to state-of-the art performance, and that suggests that deep learning is a viable option even without clusters of (super-)computers.
The results of Architecture-II are close to results obtained with vanilla neural networks but in theory could be used for images much larger in scale.

Future work could be put to verify that the Architecture-II model and methods used therein are applicable for large scale images. While the model works properly on small scale images and exhibits moderate accuracy characteristics it is still an open question if it performs equally well for images bigger in resolution.

Parameters for the networks in this work were chosen somewhat haphazardly – we have not explored fully the connection between model performance and the amount of stages, the amount of features in each stage, and the sparsity parameter. Ideally those parameters should be chosen with respect to their performance on a validation set (for example using cross validation). Due to expensive computations needed to train each model, and the time constraints the parameters were chosen heuristically. This aspect could be explored more thoroughly in future work.

Finally, apart from weight decay on the softmax classifier, no regularization methods were investigated which could further improve classification accuracy.

Bibliography

[Ben09] Yoshua Bengio. Learning deep architectures for AI. *Foundations and Trends in Machine Learning*, 2(1):1–127, 2009. Also published as a book. Now Publishers, 2009.

[CMGS11] Dan Claudiu Ciresan, Ueli Meier, Luca Maria Gambardella, and Jürgen Schmidhuber. Convolutional neural network committees for handwritten character classification. In *ICDAR*, pages 1135–1139. IEEE, 2011.

[DT05] Navneet Dalal and Bill Triggs. Histograms of oriented gradients for human detection. In *In CVPR*, pages 886–893, 2005.

[Hin10] Geoffrey E. Hinton. Learning to represent visual input. *Philosophical Transactions of the Royal Society B: Biological Sciences*, 365(1537):177–184, January 2010.

[HOT06] Geoffrey E. Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural Comput.*, 18(7):1527–1554, July 2006.

[HTF09] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning, Second Edition: Data Mining, Inference, and Prediction.* Springer Series in Statistics. Springer, 0002-2009. corr. 3rd edition, February 2009.

[IM02] Ernest Istook and Tony R. Martinez. Improved Backpropagation Learning in Neural Networks with Windowed Momentum. *International Journal of Neural Systems*, 12:303–318, 2002.

[LC] Yann Lecun and Corinna Cortes. The MNIST database of handwritten digits.

[Low99] David Lowe. Object recognition from local scale-invariant features. pages 1150–1157, 1999.

[LRM+12] Quoc Le, Marc'Aurelio Ranzato, Rajat Monga, Matthieu Devin, Kai Chen, Greg Corrado, Jeff Dean, and Andrew Ng. Building high-level features using large scale unsupervised learning. In *International Conference in Machine Learning*, 2012.

[mni] The MNIST database of handwritten digits website. `http://yann.lecun.com/exdb/mnist/`. Accessed: 2013-09-02.

[Noc80] Jorge Nocedal. Updating Quasi-Newton Matrices with Limited Storage. *Mathematics of Computation*, 35(151):773–782, 1980.

[SCL12] Pierre Sermanet, Soumith Chintala, and Yann LeCun. Convolutional neural networks applied to house numbers digit classification. *CoRR*, abs/1204.3968, 2012.

[vis] Visualizing a trained autoencoder. `http://ufldl.stanford.edu/wiki/index.php/Visualizing_a_Trained_Autoencoder`. Accessed: 2013-07-30.