1.2 Methods

1.2.1 Deep Learning Primitives

In the following, the three most prominent deep learning primitives will be described in some detail in their simplest form. These primitives or building blocks are at the foundation of many deep learning methods and understanding their basic form will allow the reader to quickly understand more complex models relying on these building blocks.

1.2.1.1 Deep Belief Networks

Deep Belief Networks (DBNs) consists of a number of layers of Restricted Boltzmann Machines (RBMs) which are trained in a greedy layer wise fashion. A RBM is an generative undirected graphical model.

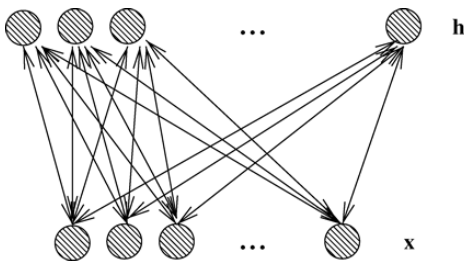

Figure 1.4: Restricted Boltzmann Machine. Taken from [Ben09].

The lower layer x, is defined as the visible layer, and the top layer h as the hidden layer. The visible and hidden layer units x and h are stochastic binary variables. The weights between the visible layer and the hidden layer are undirected and are denoted W. In addition each neuron has a bias. The model defines the probability distribution

$$P(x,h) = \frac{e^{-E(x,h)}}{Z} \tag{1.8}$$

With the energy function, $E(x, h)$ and the partition function Z being defined as

$$E(x, h) = -b'x - c'h - h'Wx \tag{1.9}$$

$$Z(x, h) = \sum_{x,h} e^{-E(x,h)} \tag{1.10}$$

Where b and c are the biases of the visible layer and the hidden layer respectively. The sum over x, h represents all possible states of the model.
The conditional probability of one layer, given the other is

$$P(h|x) = \frac{\exp(b'x + c'h + h'Wx)}{\sum_h \exp(b'x + c'h + h'Wx)} \tag{1.11}$$

$$P(h|x) = \frac{\prod_i \exp(c_i h_i + h_i W_i x)}{\prod_i \sum_h \exp(c_i h_i + h_i W_i x)} \tag{1.12}$$

$$P(h|x) = \prod_i \frac{\exp(h_i(c_i + W_i x))}{\sum_h \exp(h_i(c_i + W_i x))} \tag{1.13}$$

$$P(h|x) = \prod_i P(h_i|x) \tag{1.14}$$

Notice that if one layer is given, the distribution of the other layer is factorial. Since the neurons are binary the probability of a single neuron being on is given by

$$P(h_i = 1|x) = \frac{\exp(c_i + W_i x)}{1 + \exp(c_i + W_i x)} \tag{1.15}$$

$$P(h_i = 1|x) = \mathrm{sigm}(c_i + W_i x) \tag{1.16}$$

Similarly the conditional probability for the visible layer can be found

$$P(x_i = 1|h) = \mathrm{sigm}(b_i + W_i h) \tag{1.17}$$

In other words, it is a probabilistic version of the normal sigmoid neuron activation function. To train the model, the idea is to make the model generate data like the training data. Mathematically speaking we wish to maximize the log probability of the training data or minimize the negative log probability of the training data.
The gradient of the negative log probability of the visible layer with respect to

the model parameters θ is

$$\begin{aligned}
&\frac{\partial}{\partial\theta}(-\log P(x)) = \frac{\partial}{\partial\theta}\left(-\log\sum_h P(x,h)\right)\\
&= \frac{\partial}{\partial\theta}\left(-\log\sum_h \frac{\exp(-E(x,h))}{Z}\right)\\
&= -\frac{Z}{\sum_h \exp(-E(x,h))}\left(\sum_h \frac{1}{Z}\frac{\partial\exp(-E(x,h))}{\partial\theta} - \sum_h \frac{\exp(-E(x,h))}{Z^2}\frac{\partial Z}{\partial\theta}\right)\\
&= \sum_h\left(\frac{\exp(-E(x,h))}{\sum_{\hat{h}}\exp(-E(x,\hat{h}))}\frac{\partial E(x,h)}{\partial\theta}\right) + \frac{1}{Z}\frac{\partial Z}{\partial\theta}\\
&= \sum_h P(h|x)\frac{\partial E(x,h)}{\partial\theta} - \frac{1}{Z}\sum_{x,h}\exp(-E(x,h))\frac{\partial E(x,h)}{\partial\theta}\\
&= \sum_h P(h|x)\frac{\partial E(x,h)}{\partial\theta} - \sum_{x,h}P(x,h)\frac{\partial E(x,h)}{\partial\theta}\\
&= \mu_1\left[\left.\frac{\partial E(x,h)}{\partial\theta}\right|x\right] - \mu_1\left[\frac{\partial E(x,h)}{\partial\theta}\right]\\
&\frac{\partial}{\partial W}(-\log P(x)) = \mu_1\left[-h'x|\,x\right] - \mu_1\left[-h'x\right]\\
&\frac{\partial}{\partial b}(-\log P(x)) = \mu_1\left[-x|\,x\right] - \mu_1\left[-x\right]\\
&\frac{\partial}{\partial c}(-\log P(x)) = \mu_1\left[-h|\,x\right] - \mu_1\left[-h\right]
\end{aligned}$$

where μ_1 is a function returning the first moment or expected value. The first contribution is dubbed the positive phase, and it lowers the energy of the training data, the second contribution is dubbed the negative phase and it raises the energy of all other visible states the model is likely to generate.

The positive phase is easy to compute as the hidden layer is factorial given the visible layer. The negative phase on the other hand is not trivial to compute as it involves summing all possible states of the model.

Instead of computing the exact negative phase, we will sample from the model. Getting samples from the model is easy; given some state of the visible layer, update the hidden layer, given that state, update the visible layer, and so on, i.e.

$$
\begin{aligned}
h^{(0)} &= P(h|x^{(0)}) \\
x^{(1)} &= P(x|h^{(0)}) \\
h^{(1)} &= P(h|x^{(1)}) \\
&... \\
x^{(n)} &= P(x|h^{(n-1)})
\end{aligned}
$$

The superscripts denote the order in which each calculation is made, not the specific neuron of the layer. At each iteration the entire layer is updated. To get unbiased samples, we should initialize the model at some arbitrary state, and sample n times, n being a large number. To make this efficient, we'll do something slightly different. We'll initialize the model at a training sample, iterate one step, and use this as our negative sample. This is the contrastive divergence algorithm as introduced by Hinton [HOT06] with one step (CD-1). The logic is that, as the model distribution approaches the training data distribution, initializing the model to a training sample approximates letting the model converge.
Finally, for computational efficiency, we will use stochastic gradient descent instead of the batch update rule derived. Alternatively one can use mini-batches. The final RBM learning algorithm can be seen below. α is a learning rate and $rand()$ produces random uniform numbers between 0 and 1.

Algorithm 1 Contrastive Divergence 1

```
for all training samples as $t$ do
    $x^{(0)} \leftarrow t$
    $h^{(0)} \leftarrow sigm(x^{(0)}W + c) > rand()$
    $x^{(1)} \leftarrow sigm(h^{(0)}W^T + b) > rand()$
    $h^{(1)} \leftarrow sigm(x^{(1)}W + c) > rand()$
    $W \leftarrow W + \alpha(x^{(0)}h^{(0)} - x^{(1)}h^{(1)})$
    $b \leftarrow b + \alpha(x^{(0)} - x^{(1)})$
    $c \leftarrow c + \alpha(h^{(0)} - h^{(1)})$
end for
```

Being able to train RBMs we now turn to putting them together to form deep belief networks. The idea is to train the first RBM as described above, then train another RBM using the first RBM's hidden layer as the second RBMs visible layer.

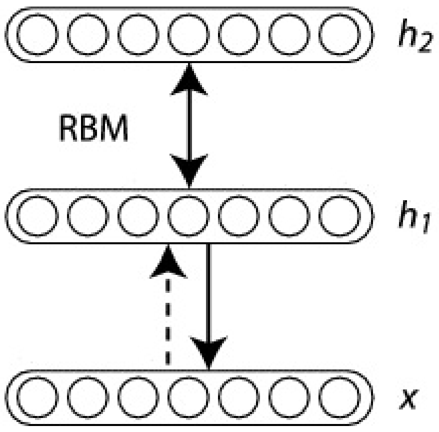

Figure 1.5: Deep Belief Network. Taken from [Ben09].

To train the second RBM, a training sample is clamped to x, transformed to h_1 by the first RBM and then contrastive divergence is used to train the second RBM. As such training the second RBM is exactly equal to training the first RBM, except that the training data is mapped through the first RBM before being used as training samples. The intuition is that if the RBM is a general method for extracting a meaningful representation of data, then it should be indifferent to what data it is applied to. Popularly speaking, the RBM doesn't know whether the visible layer is pixels, or the output of another RBM, or something different altogether. With this intuition it becomes interesting to add a second RBM, to see if it can extract a higher level representation of the data. Hinton et al have shown, that adding a second RBM decreases a variational band on the log likelihood of the training data [HOT06].
Having trained N layers in this unsupervised greedy manner, Hinton et al, adds a last RBM and adds a number of softmax neurons to its visible layer. The softmax neurons are then clamped to the labels of the training data, such that they are 0 for all except the neuron corresponding to the label of the training sample, which is set to 1. In this way, the last RBM learns a joint model of the transformed data, and the labels.

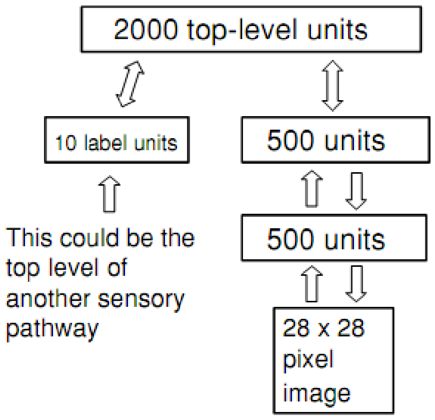

Figure 1.6: Deep Belief Network with softmax label layer. Taken from [HOT06].

To use the DBN for classification, a sample is clamped to the lowest level visible layer, transformed upwards through the DBN until it reaches the last RBMs hidden layer. In these upward passes the probabilities of hidden units being on are used directly instead of sampling, to reduce noise. At the top RBM, a few iterations of gibbs sampling is performed after which the label is read out. Alternatively the exact 'free energy' of each label can be computed and the one with the lowest free energy is chosen [HOT06]. To fine-tune the entire model for classification Hinton et al uses an 'up-down' algorithm [HOT06].
Simpler ways to use the DBN for classification are to simply use the top level RBM hidden layer activation in any standard classifier or to add a last label layer, and train the whole model as a feedforward-backpropagate neural network. If one of the latter methods are used, then there is no need to train the last RBM as a joint model of data and labels.

1.2.1.2 Stacked Autoencoders

Stacked Autoencoders are, as the name suggests, autoencoders stacked on top of each other, and trained in a layerwise greedy fashion.
An autoencoder or auto-associator is a discriminative graphical model that attempts to reconstruct its input signals.

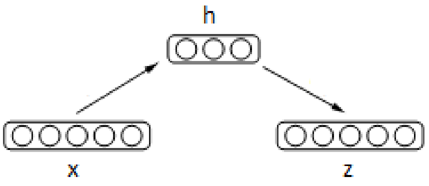

Figure 1.7: Autoencoder. Taken from [Ben09].

There exists a plethora of proposed autoencoder architectures; with/without tied weights, with various activation functions, with deterministic/stochastic variables, etc. This section will use the autoencoder desribed in [BLP$^+$07] as a basic example implementation which have been used successfully as a building block for deep architectures.
Autoencoders take a vector input x, encodes it to a hidden layer h, and decodes it to a reconstruction z.

$$h(x) = \text{sigm}(W_1 x + b_1) \tag{1.18}$$

$$z(x) = \text{sigm}(W_2 h(x) + b_2) \tag{1.19}$$

To train the model, the idea is to minimize the average difference between the input x and the reconstruction z with respect to the parameters, here denoted θ.

$$\theta = \underset{\theta}{\text{argmin}} \frac{1}{N} \sum_{i=1}^{N} L(x^{(i)}, z(x^{(i)})) \tag{1.20}$$

Where N is the number of training samples and L is a function that measures the difference between x and z, such as the traditional squared error $L(x,z) = ||x - z||^2$ or if x and z are bit vectors or bit probabilities, the cross-entropy $L(x,z) = x^T log(z) + (1-x)^T log(1-z)$
Updating the parameters efficiently is achieved with stochastic gradient descent which can be efficiently implementing using the backpropagation algorithm.

There is a serious problem with autoencoders, in that if the hidden layer is the same size or greater than the input and reconstruction layers, then the algorithm could simply learn the identity function. If the hidden layer is smaller than the input layer, or if other restrictions are put on its representation, e.g. sparseness, then this is not an issue.

Having trained the bottom layer autoencoder on data, a second layer autoencoder can be trained on the activities of the first autoencoders hidden layer when exposed to data. In other words the second autoencoder is trained on $h^{(1)}(x)$ and the third autoencoder would be trained on $h^{(2)}(h^{(1)}(x))$, etc, where the superscripts denote the layer of the autoencoder. In this way multiple autoencoders can be stacked on top of each other and trained in a layer-wise greedy fashion, which has been shown to lead to good results [VLBM08].

To use the model for discrimination, the outputs of the last layer can be used in any standard classifier. Alternatively a last supervised layer can be added, and the whole model trained as a feedforward-backpropagate neural network.

1.2.1.3 Convolutional Neural Nets

Convolutional Neural Networks (CNNs) are feedforward, backpropagate neural networks with a special architecture inspired from the visual system. Hubel and Wiesel's early work on cat and monkey visual cortex showed that the visual cortex is composed of cells with high specificity to patterns within a localized area, called their receptive fields [HW68]. These so called simple cells are tiled as to cover the entire visual field and higher level cells recieve input from these simple cells, thus having greater receptive fields and showing greater invariance to translation. To mimick these properties Yan Lecun introduced the Convolutional Neural Network [LCBD+90], which still hold state-of-the art performance on numerous machine vision tasks [CMS12] and acts as inspiration to recent reseach [SWB+07], [LGRN09].

CNNs work on the 2 dimensional data, so called maps, directly, unlike normal neural networks which would concatenate these into vectors. CNNs consists of alternating layers of convolution layers and sub-sampling/pooling layers. The convolution layers compose feature maps by convolving kernels over feature maps in layers below them. The subsampling layers simply downsample the feature maps by a constant factor.

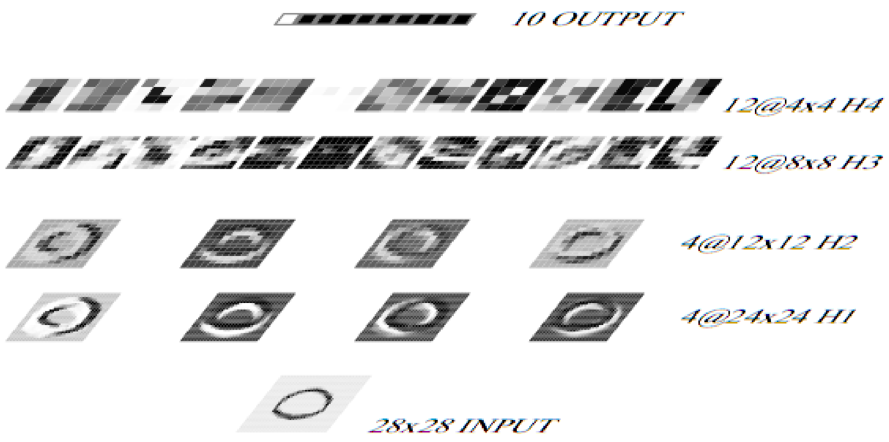

Figure 1.8: Convolutional Neural Net. Taken from [LCBD+90].

The activations a_j^l of a single feature map j in a convolution layer l is

$$a_j^l = f\left(b_j^l + \sum_{i \in M_j^l} a_i^{l-1} * k_{ij}^l\right) \tag{1.21}$$

Where f is a non-linearity, typically tanh() or sigm(), b_j^l is a scalar bias, M_j^l is a vector of indexes of the feature maps i in layer $l-1$ which feature map j in layer l should sum over, $*$ is the 2 dimensional convolution operator and k_{ij}^l is the kernel used on feature map i in layer $l-1$ to produce input to the sum in feature map j in layer l.
For a single feature map j in a subsampling layer l

$$a_j^l = \text{down}(a_j^{l-1}, N^l) \tag{1.22}$$

Where down is a function that downsamples by a factor N^l. A typical choice of down-sampling is mean-sampling in which the mean over non-overlapping regions of size N^lxN^l are calculated.
To discriminate between C classes a fully connected output layer with C neurons are added. The output layer takes as input the concatenated feature maps of the layer below it, denoted the feature vector, fv

$$o = f\left(b^o + W^o fv)\right) \tag{1.23}$$

Where b^o is a bias vector and W^o is a weight matrix. The learnable parameters of the model are k_{ij}^l, b_j^l, b^o and W^o. Learning is done using gradient descent which can be implemented efficiently using a convolutional implementation of the back-propagation algorithm as shown in [Bou06]. It should be clear that because kernels are applied over entire input maps, there are many more connections in the model than weights, i.e. the weights are shared. This makes learning deep models easier, as compared to normal feedforward-backprop neural nets, as there are fewer parameters, and the error gradients goes to zero slower because each weight has greater influence on the final output.

1.2.1.4 Sparsity

Sparse coding is the paradigm that data should be represented by a small subset of available basis functions at any time, and is based on the observation that the brain seems to represent information with a small number of neurons at any given time [OF04]. Sparsity was originally investigated in the context of efficient coding and compressed sensing and was shown to lead to gabor-like filters [OF97]. They are not directly related to deep architectures, but their interesting encoding properties have lead to them being used in deep learning algorithms in various ways.

The most direct use of sparse coding can be seen as formulating a new basis for a dataset which is composed of a feature vector and a set of basis functions, while restricting the feature vector to be sparse.

$$x = Aa \tag{1.24}$$

Where $x \in \Re^n$ is the data, $A \in \Re^{nxm}$ is the m basis functions and $a \in \Re^m$ is the "sparse" vector describing which sum of basis functions represent the data. a is sparse in the sense that it is mostly zero, i.e. few basis functions are used at all times to represent any data. In images this is in contrast to the normal non-sparse representation used, which is pixel intensities. This corresponds to $A = I^{nxn}$, i.e A being a square identity matrix, and a being the normal vector representation of image intensities.

In a deep learning setting, a non-sparsity penalty, i.e. measuring how much the neurons are active on average, can be added to the loss function. If the neurons are binomial, sparse coding could correspond to restricting the mean number of activate hidden neurons to some fraction. If the neurons are continuous valued, this could correspond to restricting the mean of all hidden units to some constant. Sparse variants of RBMs and autoencoders have been proposed

[GLS+09, PCL06]. A sparse autoencoder has a second contribution to its loss function, the non-sparsity measure:

$$L(x, z) = ||x - z||^2 + \beta||h - \rho|| \tag{1.25}$$

Where β is a constant describing how much being non-sparse should be penalized and ρ is a constant close to the lower boundary of the output of the hidden neurons h. In the case of sigmoidal hidden neurons, this could be $\rho = 0.1$. In the case of tanh hidden neurons this could be $\rho = -0.9$

1.2.2 Key Points

1.2.2.1 Training Deep Architectures

A key element to Deep Learning is the ability to learn from unlabelled data as it is available in vast quantities, e.g. video or images of natural scenes, sound, etc. This is also referred to as unsupervised training. This is in contrast to supervised training which is training on labelled data, of which there is relatively little, and which is much harder to generate. To get unlabelled video data one might simply go onto youtube.com and download a million hours of video, but to get 1 hour of labelled video data one would need to painstakingly segment each frame into the objects of interest. Further, being able to learn on the unlabelled data gives one a high-dimensional learning signal, whereas most labelled data is relatively low-dimensional, e.g. a label specifying cat or dog is two bits of information whereas a 100x100 pixels image of a cat or dog in true color is 24*3*100*100 = 720.000 bits.

Machine Learning models are rarely built to achieve good performance on unlabelled data though; usually some kind of classification or regression is required. The idea then is to train the model on the unlabelled data first, called pre-training, to achieve good features or representations of the data. Once this has been achieved the parameters learned are used to initiate a model, which is trained in a supervised fashion to fine-tunes the parameters to the task at hand.

Deep belief nets and stacked auto encoders both use the same method for pre-training: training each layer unsupervised on the activations of the layer below, one after another. Convolutional neural nets stand out in this aspect, as they do not use pre-training. Since CNNs have substantially less parameters, and translation invariance is built into the model, there seems to be less need for pre-training. Pre-training convolutional neural nets in a layer-wise fashion similar to DBNs and SAEs have been shown to be slightly superior to randomly initialized networks though [MMCS11]. Generally the paradigm of greedy layer-wise pre-training followed by global supervised training to fine-tune the parameters seems to give good results.

The theory as to why this works well is that the unsupervised pre-training moves the parameters to a region in parameter space that is closer to a global optimum or at least a region which represents the data more naturally. Numerical studies have shown that pre-trained and randomly initialized networks do indeed end up in very different regions of parameter space after having been trained on a supervised task [EBC+10]. Also, the global supervised learning rarely changes the pre-trained parameters much, what happens instead is a fine-tuning to improve on the supervised task [EBC+10].

After pre-training, instead of training the model globally on the supervised task one can instead use any standard supervised learning model on the output features of the pre-trained model, e.g. pre-training a Deep Belief Network, and then using the activity of the top output neurons as input in a SVM, logistic regression, etc.

Alternatively one can train the model in a supervised and unsupervised setting at the same time, alternating between the two learning modes or having a composite learning rule. This is known as semi-supervised learning.

1.3 Results

The three primitives, DBNs, SAEs and CNNs were implemented and evaluated on the MNIST dataset to illustrate state of the art in Deep Learning. The error rates achieved for the DBN, SAE and CNN were 1.67%, 1.71% and 1.22%, respectively. The error rates compared to state-of-the art with comparable network architectures for the DBN and SDAEs are slightly worse whereas the CNN error rate is slightly better.

DBN (1000-1000-1000)	SAE (1000-1000-1000)	CNN (c6-d2-c72-d2)
1.67 %	1.71 %	1.22 %

Table 1.1: Error rates of three deep learning primitives. The DBN and SAE both had 3 hidden layers each with 1000 neurons. The CNN had the following layers: 6 feature maps using 5x5 kernels, 2x2 mean-pool downsampling, 72 feature maps using 5x5 kernels, 2x2 downsampling and a fully connected output layer.

The MNIST dataset [LBBH98] contains 70.000 labelled images of handwritten digits. The images are 28 by 28 pixels and gray scale. The dataset is divided into a training set of 60.000 images and a test set of 10.000 images. The dataset has been widely used as a benchmark of machine learning algorithms. In the following details of the implementations of the three models on MNIST is described and results on MNIST are shown.

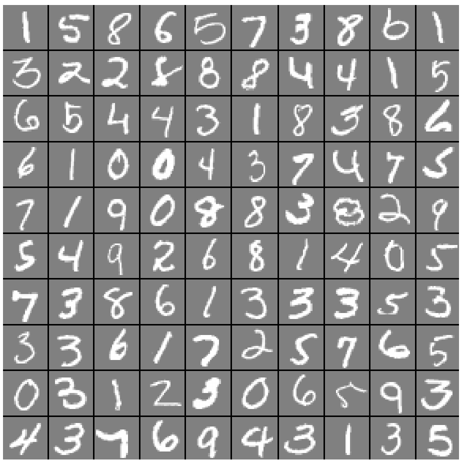

Figure 1.9: A random selection of MNIST training images.

Except otherwise noted all experiments used the sigmoid non-linearity for all neurons, initialized the biases to zero and drew weights from a uniform random distribution with upper and lower bounds $\pm\sqrt{6/(fan_{in} + fan_{out})}$ as recommended in [LBOM98]. All experiments were run on a machine with a 2.66 GHz Intel Xeon Processor and 24 GB of memory.

1.3.1 Deep Belief Network

A three layer DBN were constructed. The net consisted of three RBMs each with 1000 hidden neurons, and each RBM was trained in a layer-wise greedy manner with contrastive divergence. All weights and biases were initialized to be zero. Each RBM was trained on the full 60.000 images training set, using mini-batches of size 10, with a fixed learning rate of 0.01 for 100 epochs. One epoch is one full sweep of the data. The mini-batches were randomly selected each epoch. Having trained the first RBM the entire training dataset was transformed through the first RBM resulting in a new 60.000 x 1000 dataset which the second RBM was trained on and similarly so for the third RBM. Having pre-trained each RBM the weights and biases were used to initialize a feed-forward neural net with 4 layers of sizes 1000-1000-1000-10, the last 10 neurons being the output label units. The FFNN was trained using mini-batches of size 10 for 50 epochs using a fixed learning rate of 0.1 and a small L2 weight-decay of 0.00001 using back-propagation. To evaluate the performance the test set was feed-forwarded and the maximum output unit was chosen as the label for each sample resulting in an error rate of 1.67% or 167 errors out of the 10.000 test samples. The code ran for 28 hours.

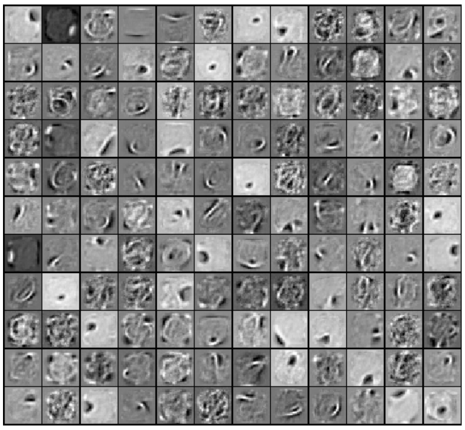

Figure 1.10: Weights of a random subset of the 1000 neurons of the first RBM. Each image is contrast normalized individually to be between minus one and one.

The first RBM has to a large degree learned stroke and blob detectors as can be seen from the weights. Less meaningful detectors are also present either reflecting the higly over-parametrized nature of the RBM or a lack of learning. Given the good performance it is probable that the dataset could be sufficiently represented by the other neurons.

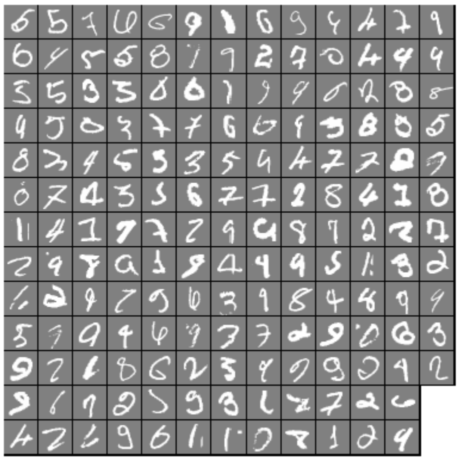

Figure 1.11: The 167 errors using a 3 layer DBN with 1000, 1000 and 1000 neurons respectively.

While some of the images are genuinely difficult to label, a number of them seems easy. Many of the sevens in particular seem fairly easy. The added intra-class variation due to continental sevens and regular sevens might explain this to a degree.

Hinton showed an error rate of 1.25% in his paper introducing the DBN and contrastive divergence [HOT06]. This impressive performance was achieve with a 3 layer DBN with 500, 500 and 2000 hidden units respectively, training a combined model of the representation and the labels in the last layer together and using extensive cross validation to tune the hyper parameters. Additionally Hinton used a novel up-down algorithm to tune the weights on the classification task, running a total of 359 epochs resulting in a learning time of about a week. It has been shown [VLL$^+$10] that pre training a DBN, using its weights to initialize a deep FFNN and training that on a supervised task with stochastic backpropagation can lead to the same error rates as those reported by Hinton. As such it seems that it is not the training regime used resulting in the higher error rate but rather a need for further tuning of the hyper parameters.

1.3.2 Stacked Denoising Autoencoder

A three layer stacked denoising autoencoder (SDAE) with architecture identical to the DBN was created. The denoising autoencoder works just like the normal autoencoder except that the input is corrupted in some way, and the autoencoder is trained to reconstruct the un-corrupted input [VLBM08]. The idea is that the autoencoder cannot simply copy pixels and will have to learn corruption invariant features to reconstruct well. The corruption process used was setting a randomly selected fraction of the pixel in the input image to zero. The SDAE consisted of three denoising autoencoder (DAE) stacked on top of each other each with 1000 hidden neurons, and each trained in a greedy-layer wise fashion. Each DAE was pre-trained with a fixed learning rate of 0.01 and a batchsize of 10 for 30 epochs and with a corruption fraction of 0.25 i.e. a quarter of the pixels set to zero in the input images. The noise level was chosen based on conclusions from [VLL$^+$10]. Having trained the first DAE the training set was feed-forwarded through the DAE and the second DAE was trained on the hidden neuron states of the first DAE, and similarly for the third DAE. After pre-training in this manner the upwards weights and biases were used to initialize a FFNN with 10 output neurons in the same manner as for the DBN. The FFNN was trained with a fixed learning rate of 0.1, with a batchsize of 10 for 30 epochs. The performance was measured as for the DBN resulting in an error rate of 1.71% or 171 errors. The code ran for 41 hours.

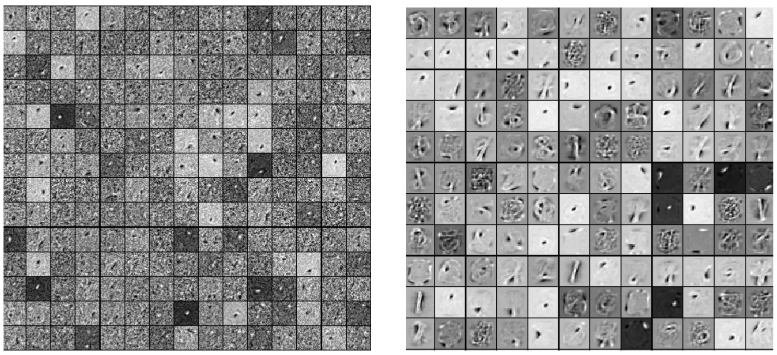

Figure 1.12: Left: Weights of a random subset of the 1000 neurons of a DAE with a corruption level of 0.25. Right: DAE trained in a similar manner with a corruption level of 0.5. The second DAE had worse discriminative performance. Its weights are shown here only to show that DAEs can find good representations. Each image is contrast normalized individually to be between minus one and one.

The DAE seems to find mostly seemingly non-sensical and blob detectors. For reference the weights of another DAE trained in a similar manner with a corruption level of 0.5 is provided. The latter finds stroke detectors and seem less noisy. The latter DAE had worse discriminative performance (not shown).

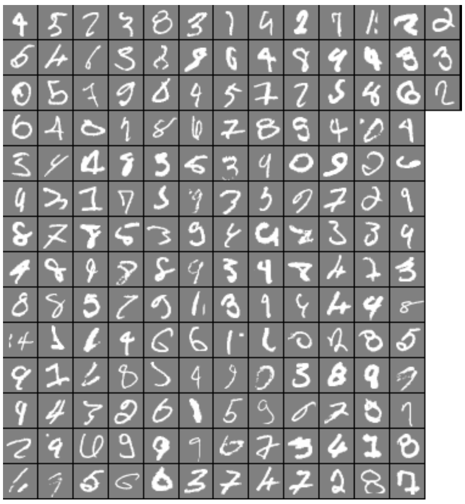

Figure 1.13: The 171 errors using a 3 layer SDAE with 1000,1000 and 1000 neurons respectively.

In [VLBM08] Pascal Vincent introduces the denoising autoencoder and reports superior performance on a number of MNIST like tasks. The basic MNIST test score is not reported though until his 2010 paper [VLL+10] in which Vincent reports an error rate of 1.28% on MNIST with a three layer SDAE using 25% corruption and extensive cross validation to tune the hyperparameters. The 1.71 % error rate here is on the same order of magnitude but compares unfavourably to the 1.28 %. It is evident that further tuning of the hyper parameters would have been beneficial.

1.3.3 Convolutional Neural Network

A Convolutional Neural Network was created following the architecture in [LCBD+90] in which Yann LeCun introduces the CNN. The first layer has 6 feature maps connected to the single input layer through 6 5x5 kernels. The second layer is a a 2x2 mean-pooling layer. The third layer has 12 feature maps which are all connected to all 6 mean-pooling layers below through 72 5x5 kernels. This full

connection between layer 2 and 3 is in contrast to the architecture proposed by LeCun, which used a hand-chosen set of connections. The fourth layer is a 2x2 mean-pooling layer. When training, the feature maps of this fourth layer is concatenated into a feature vector which feeds into the fifth and final layer which consists of 10 output neurons corresponding to the 10 class labels.
The CNN was trained with stochastic gradient descent on the full MNIST training set. A batch size of 50 and a fixed learning rate of 1 was used for 100 epochs resulting in a test score of 1.22% or 122 misclassifications. The code ran for 7 hours.

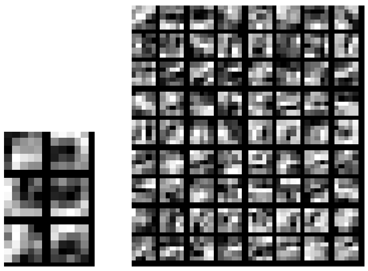

Figure 1.14: Left: The 6 kernels of the first layer. Right: The 72 kernels of the third layer. All kernels are contrast normalized individually to be between minus and plus one.

The CNNs 6 first layer kernels seems to be 4 curvy stroke detectors and two less well defined detectors. The 72 layer three kernels cannot be directly analysed with respect to what detectors they are as they operate on already transformed input. There does seem to be some structure in them though reflecting that the feature maps in layer 2 are still resembling digits.

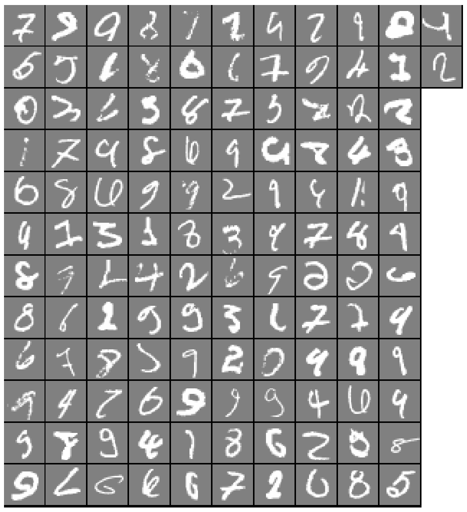

Figure 1.15: The 122 errors with the CNN.

The MNIST dataset did not exist at the time LeCun introduced the CNN. However in his 1998 paper [LBBH98] he reports a 1.7 % error for a network of this architecture, which he names LeNet1. Lecun subsamples the images to 16x16 pixels and uses a second order backprop method to achieve the 1.7%. The 1.22 % error rate compares favourably with this as there were no pre-processing and simple first-order backprop with a fixed learning rate was used. It should be noted that LeCun reports an error rate of 0.95% with LeNet-5, a more advanced net in the same paper. As no cross-validation was used to find the hyper-parameters the performance could probably be increased with further tuning of the hyper-parameters. It is remarkable that such a simple architecture as LeNet-1 is able to achieve such good performance.