Deep neural network architectures with restricted Boltzmann machines and autoencoders

Laboratory report 4

Anna Canal Garcia
Magdalini Papaioannou
Jenny Stoby Höglund

DD2437 Artificial neural networks and deep architectures

October 3, 2018

1 Introduction

The main purpose for this lab is to get familiar with some of the key ingredients of deep neural network (DNN) architectures. The focus in this assignment is on restricted Boltzmann machines (RBMs) and autoencoders.

2 Method

The first part of the laboratory consists in training an RBM and an autoencoder for binary-type MNIST images. Unlike typical benchmark applications of DBNs, here the data have already been normalized and converted to pseudobinary distributions, so that we could directly employ a Bernoulli type of RBMs, not Gaussian. In particular, the MNIST dataset consists of four csv files with data for training and the other two for testing. Data in bindigit les represent 28-by-28 matrices organized into 784-dim binary vectors. Data in both training and test sets are relatively balanced with respect to 10 classes.

We have used the sklearn library to build the RBM and the keras API paired with TensorFlow backend to build the auntoencoder.

The second part consists in extending a single-hidden layer network to a deeper architecture by following the idea of greedy layer-wise pretraining.

3 Results

3.1 RBM and autoencoder features for binary-type MNIST images

3.1.1 RBM

In this section the results from training an RBM on the MNIST dataset are presented. We train an RBM on our training data to learn an unsupervised feature representation of the digits. What we are looking for by using an RBM is to express our input data with a better feature representation.

Each image, which consists of 784 pixels, will be expressed with fewer features, which means that it will be expressed in a lower dimension, that corresponds to the number of hidden nodes. We have tried different numbers of hidden nodes: 50, 75, 100 and 150.

The parameters we have used in the BernoulliRBM from sklearn are the following: *learning_rate = 0.05*, *n_iter = [10, 20]*, *n_components = [50, 75, 100, 150]*. First of all, for each image we computed the mean error between the original input and the reconstructed input. Results of the total error on the dataset for the current epoch up to 20 epochs are shown in Figure 1, where we can see that the error decreases as the number of hidden nodes increases.

Finally we show the mean absolute error after 20 epochs in the table 1.

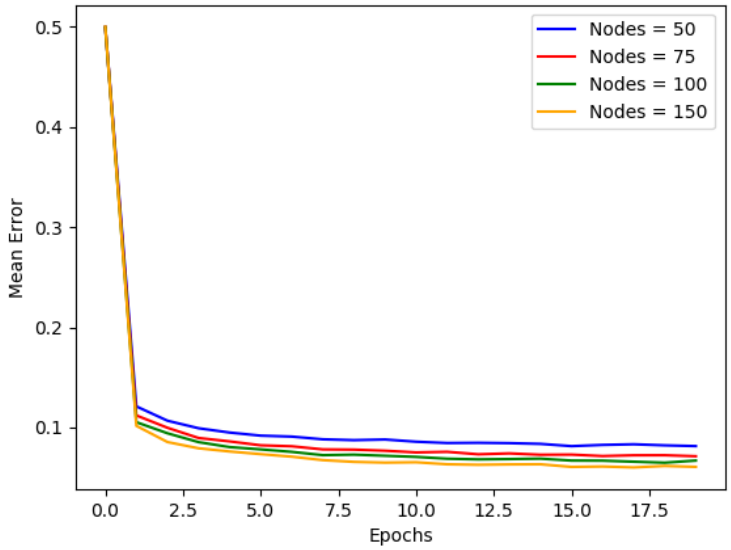

Figure 1: Total mean error on the dataset for the current epoch with 50, 75, 100 and 150 nodes.

#hidden nodes	**Mean error**
50	0.08108
75	0.07046
100	0.06389
150	0.05837

Table 1: Mean error of RBM with different numbers of hidden nodes, after 20 epochs.

In order to check the performance we can also take a look at the image reconstructions. Figure 2 shows the prediction of each digit on the test set after 20 epochs, using different numbers of hidden nodes. As we were expecting from the error results, when using more nodes the reconstruction is better since we have more features which take into consideration more details from images. It can also be seen that the digits that have a unique form are easier to predict (for example "1") than digits that are similar to others (for example 3 can be confused with 8 or 2, or 9 confused with 4).

Secondly, we explore the weights when using different numbers of hidden nodes. Figure 3 shows the 784 bottom-up final weights for each hidden unit for the configuration of 50 nodes and Figure 4 for the configuration of 100 nodes, in order to observe what activates each unit. The weights from both figures are not easy to interpret visually, but we can see that some of them are active when there is a vertical line (like for digits like 1, 9, 7) or when there is a circle (like for digits 8,6,0,9). Moreover, we can see that when using 100 hidden nodes, there are more weights active when there is some diagonal line or diagonal circle that

Figure 2: Reconstruction of digits on the test set. From left to right: input, reconstruction with 50 nodes, reconstruction with 75 nodes, reconstruction with 100 nodes and reconstruction with 150 nodes.

helps to the digits like 2 and 3.

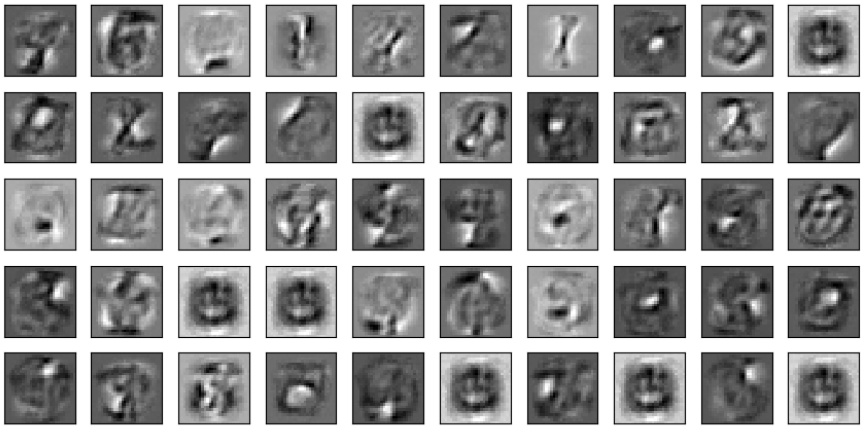

Figure 3: Final weights of RBM with 50 hidden nodes.

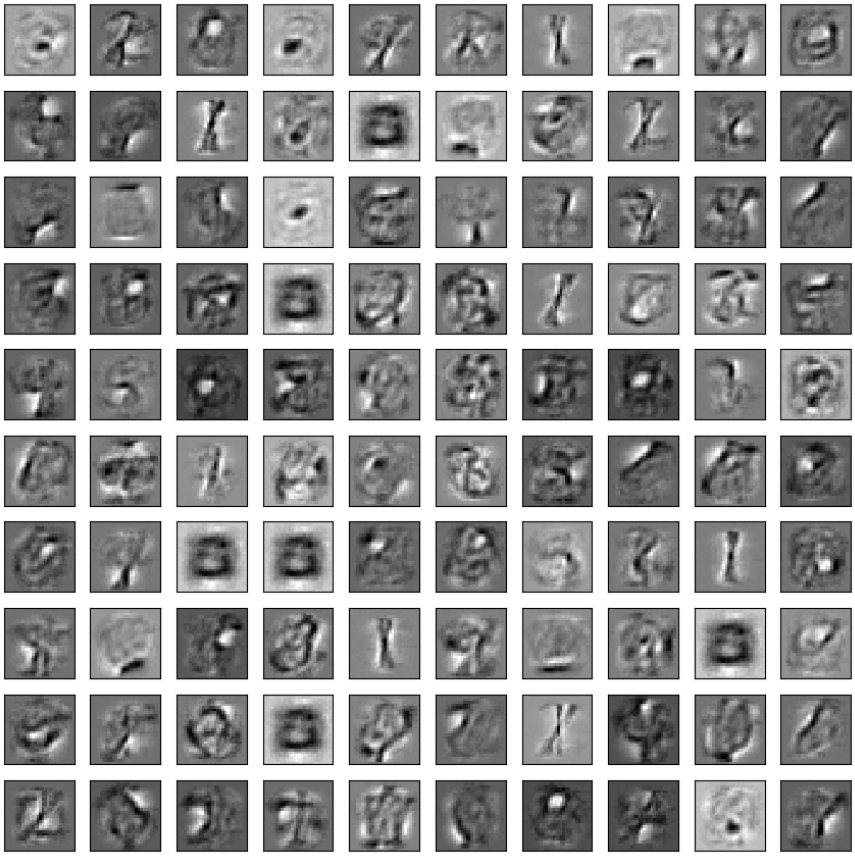

Figure 4: Final weights of RBM with 100 hidden nodes.

3.1.2 Autoencoder

In this section we presenting our results after implementing and experimenting on an autoencoder implementation. In order to do the training and experimenting we followed a similar procedure as before.

We used the Keras API with TensorFlow as backend, as mentioned in the methods section.

We trained an autoencoder with 50, 75, 100, and 150 hidden nodes respectively. The activation function of the encoding nodes was ReLu, while the activation function of the decoding nodes was the sigmoid. We initialized the weights with small, normally distributed random values, and the biases with zeros as requested. We used stochastic gradient descent that tries to minimize the mean squared error between the input and the prediction, because we found that this error minimization gives us more accurate results, but we also monitored the mean error, as this is what we are requested to do in the lab.

We used a learning rate of 0.3. Even though 0.3 might be a considered a quite large learning rate, our autoencoder still needed 75 epochs until convergence.

The parameters used when fitting the Keras autoencoder model were batch size: 1, shuffle: "True" and validation split: 0.3. Setting the shuffle parameter to true makes the training data to be shuffled before each epoch of the algorithm. The validation split represents the fraction of the data that will be used for validation and not for actual training. The batch size controls the number of samples before each gradient update. Bigger batch sizes allowed the training process to converge much faster, but produced far worse results.

After training, we computed the mean error between the original input and the reconstructed input, on the test dataset. Results of the total error per epoch, for 75 epochs (until convergence), are shown in Figure 5, where we can see that the error smoothly decreases throughout the epochs, until it reaches an almost stable state. Furthermore, it is clear that the error of autoencoders with less hidden nodes is constantly higher than the error of autoencoders with more hidden nodes. This is expected as the less the hidden nodes, the more the lossy compression we are forcing on our data.

Finally we show the mean absolute error after 75 epochs in the table 2.

#hidden nodes	**Mean error**
50	0.02464
75	0.01783
100	0.01544
150	0.01313

Table 2: Mean autoencoder error, after 75 epochs

In order to check the performance we can also take a look at the image reconstructions, as we did for RBM. Figure 6 shows the prediction of each digit on

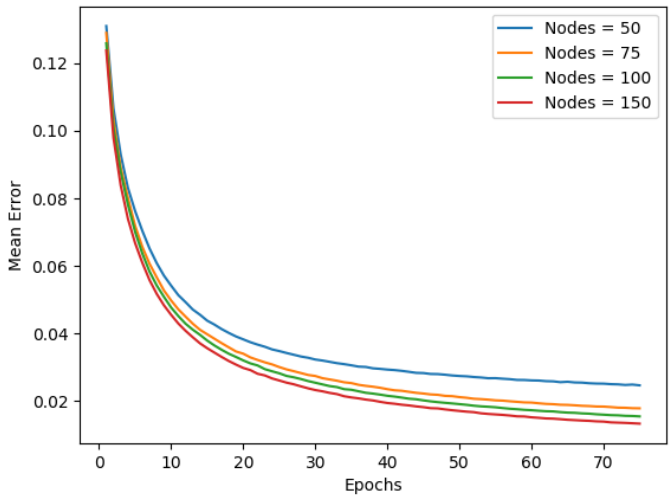

Figure 5: Total mean error on the dataset for the current epoch with 50, 75, 100 and 150 nodes

the test set using different numbers of hidden nodes after 75 epochs. We chose to reconstruct the exact same digits as before, in order to be able to compare the two methods. Autoencoders seem to have smoother and cleaner results after convergence, in comparison to RBMs. As for the difference between different numbers of hidden nodes, we can see that the more the nodes, the better (less blurry) the reconstruction, but the difference is hardly noticeable. This was expected, as the difference on the errors is very small too. Next, we plotted the final (for epoch equal to 75) hidden node weights of the autoencoder as images, in order to visualize their activations. The autoencoder weight figures are more blurry and difficult to interpret than the RBM ones, but most of them do have a digit-like form as well (information gathered at the center, with prominent circles and vertical lines), that seems to spring from a combination of more than one digits. The weights of the 50-hidden-nodes-autoencoder, can be seen in Figure 7, while the weights of the 100-hidden-nodes-autoencoder can be seen in Figure 8. We would also like to note that in Figure 8 there are a few "white noise" weights. As the epochs progress, these weights become fewer and fewer, as they all tend to take some digit-like form.

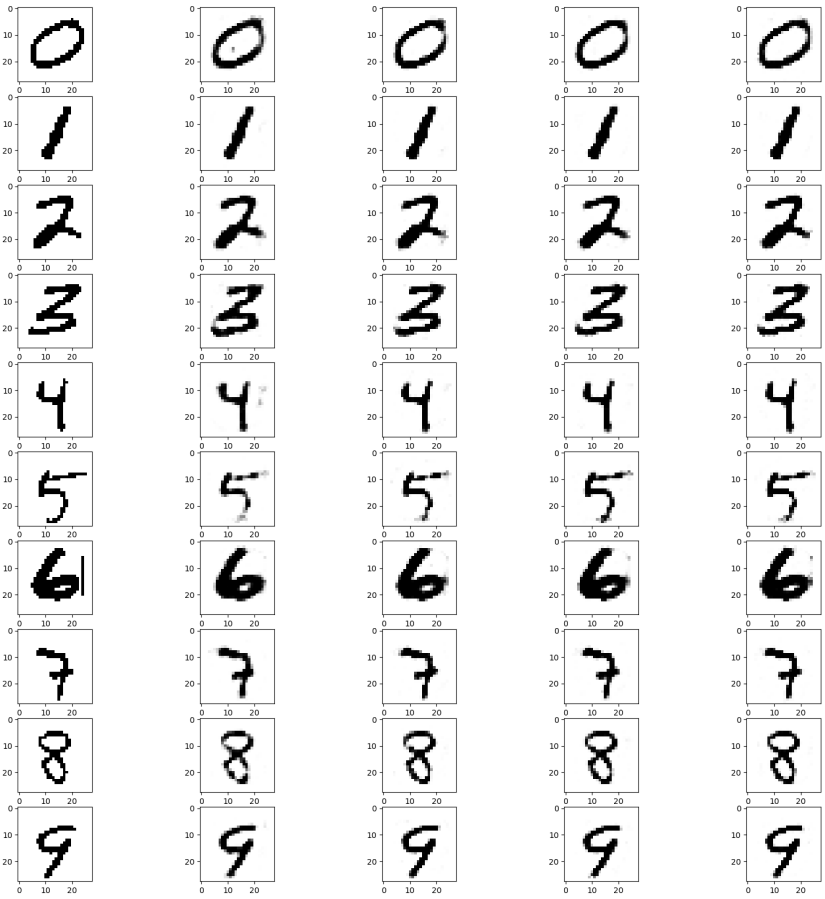

Figure 6: Reconstruction of test set digits, using a simple autoencoder. From left to right: input, reconstruction with 50 nodes,reconstruction with 75 nodes,reconstruction with 100 nodes and reconstruction with 150 nodes.

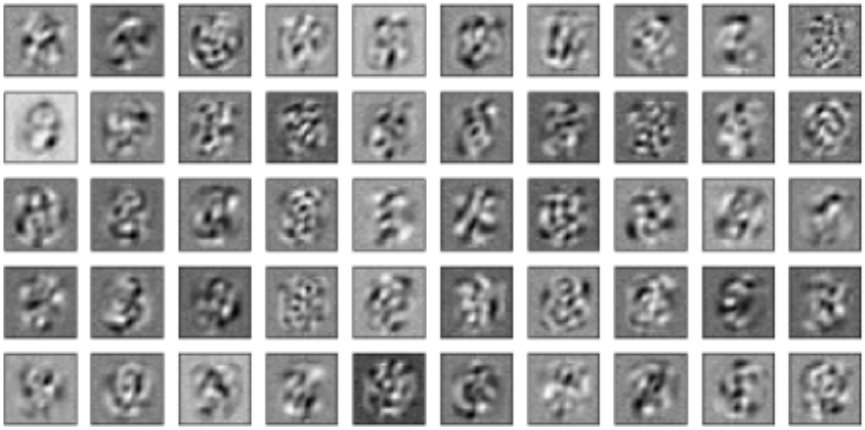

Figure 7: Final weights of autoencoder with 50 hidden nodes.

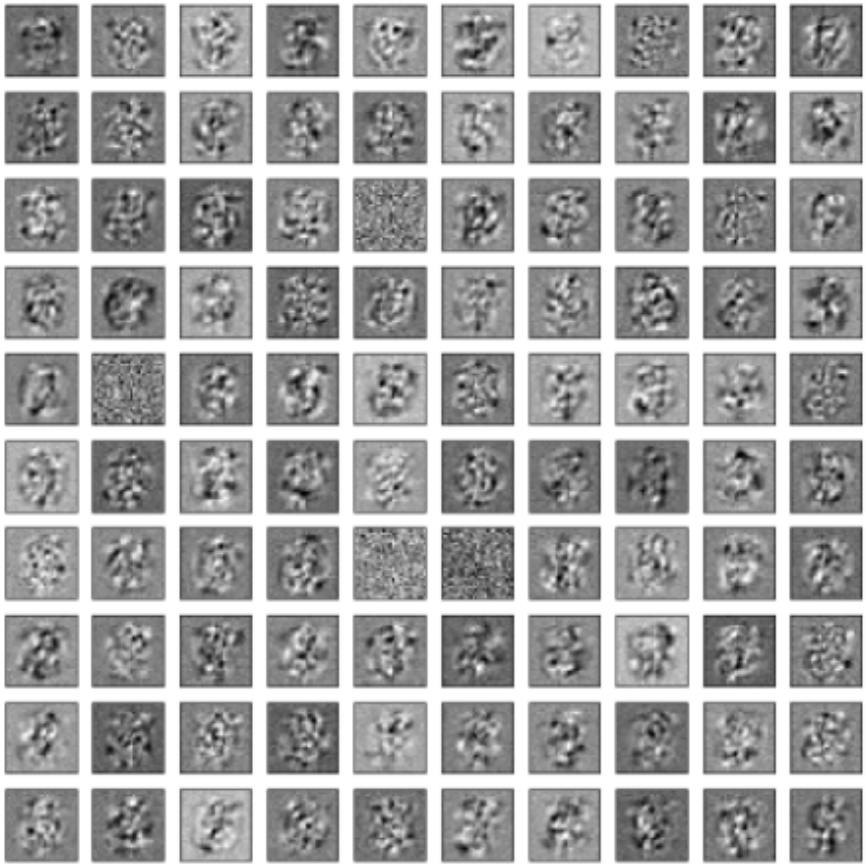

Figure 8: Final weights of autoencoder with 100 hidden nodes.

3.2 DBN and stacked autoencoders for MNIST digit classification

Here we present the results when extending a single-hidden layer network to a deeper architecture by following the idea of greedy layer-wise pretraining.

3.2.1 Classification with deeper architectures

In this section we monitored the classification performance obtained with different numbers of hidden layers (1,2 and 3). We also created a network configuration with a simple classification layer operating directly on the raw inputs as a no-hidden-layer option. We examined the hidden layer representations and observed the effect of images representing different digits on hidden units in the hidden layers. Finally, we compared the deep network configurations of our choice to MLP.

RBM

We trained a deep network of stacked RBMs with a classifier (a logistic regression layer) on top. We fixed the number of hidden nodes for the first layer at 150, because that is the number of nodes that gave us the best results in the exploration done in section 3.1.1. Then we tried the second layer with 30 nodes less. That brings us to 120 nodes. Finally, we decreased the same number of nodes form layer 2 to layer 3. So we used 90 nodes for layer 3. We kept the other parameters as in section 3.1.1 (*learning_rate = 0.05*, *n_iter = 20*). Table 3 shows the comparison of the error using different numbers of hidden layers and without decreasing the dimensionality, so a version without using any RBM.

#hidden layers	Precision	Recall	f1-score
0	0.86	0.86	0.86
1	0.92	0.92	0.92
2	0.93	0.93	0.93
3	0.93	0.93	0.93

Table 3: Performance of the classifier when using different hidden layers for pretraining.

As we can see in table 3, the performance increases a little bit when using 2 layers instead of 1. So, we can achieve now better performance while using the same number of nodes, but doing the dimensionality reduction twice (in two hidden layers) and not once. Moreover, when reducing even more the features extracted from images to 90, we manage to keep the same performance. So, this stacked RBMs network is capable of transforming the 150 features obtained at the first layer to 90 without losing performance.

After the performance comparison, we analyzed the hidden layer representations. Figure 9 shows the weights of the first hidden layer which contains 150 nodes, Figure 10 shows the weights of the second hidden layer(120 nodes) and Figure 11 shows the weights of the third hidden layer (90 nodes).

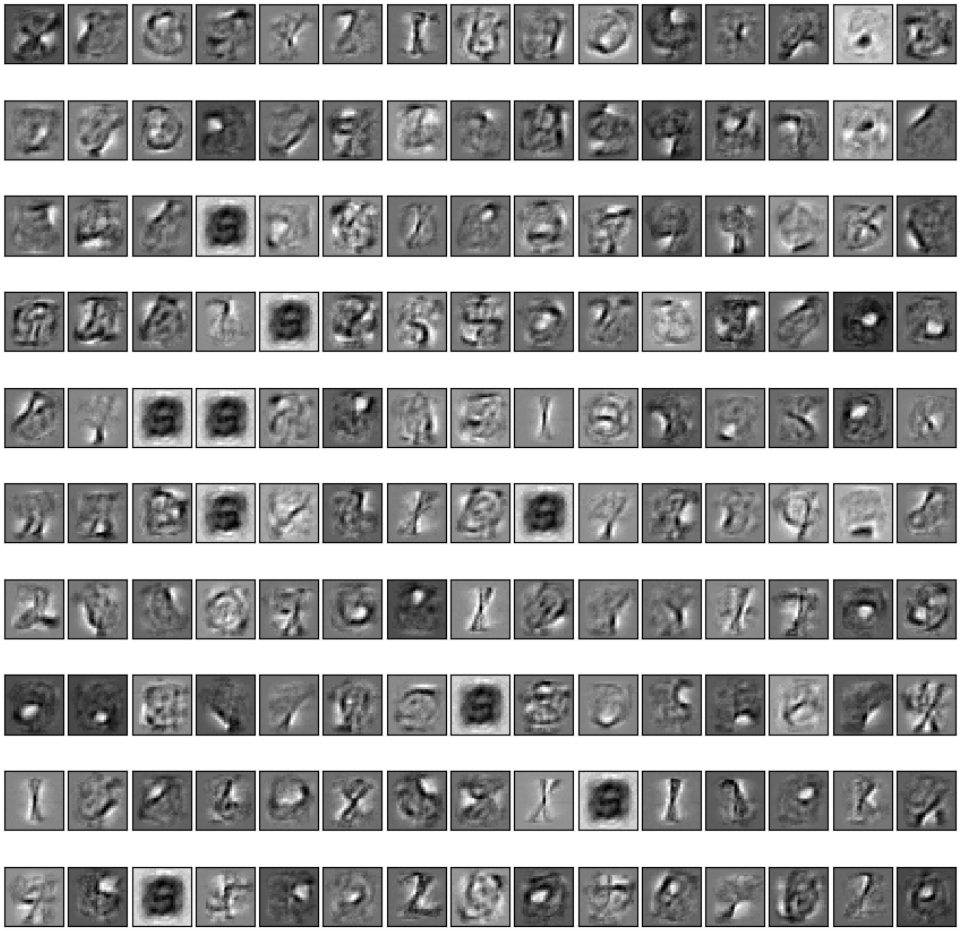

Figure 9: Weights from the 1st hidden layer (150 nodes).

In Figure 9 the weights have shapes similar to the ones obtained in the section 3.1.1, where we only had one layer. However, there are some weights that are almost identical so this tells us that we can reduce the dimensionality even more.

If we look at figures 10 and 11 it is difficult to interpret the weights and find any correspondence between them and the input data. This is because now the input dimensions of the 2nd layer is 150 and for the 3rd layer is 120, so these layers try to encode these, already encoded lower dimension inputs.

Finally, we compare the results of the stacked RBM to an MLP. MLP is implemented using sklearn library and it consists of 3 layers with the same numbers of nodes (150, 120, 90), the same parameters for training, and using stochastic gradient descent as the learning algorithm and ReLu as the activation function. Table 4 shows the performance of both kinds of networks.
We can observe according to the results presented in Table 4 that the MLP performance is a tiny bit higher than the stacked RBM performance, but over-

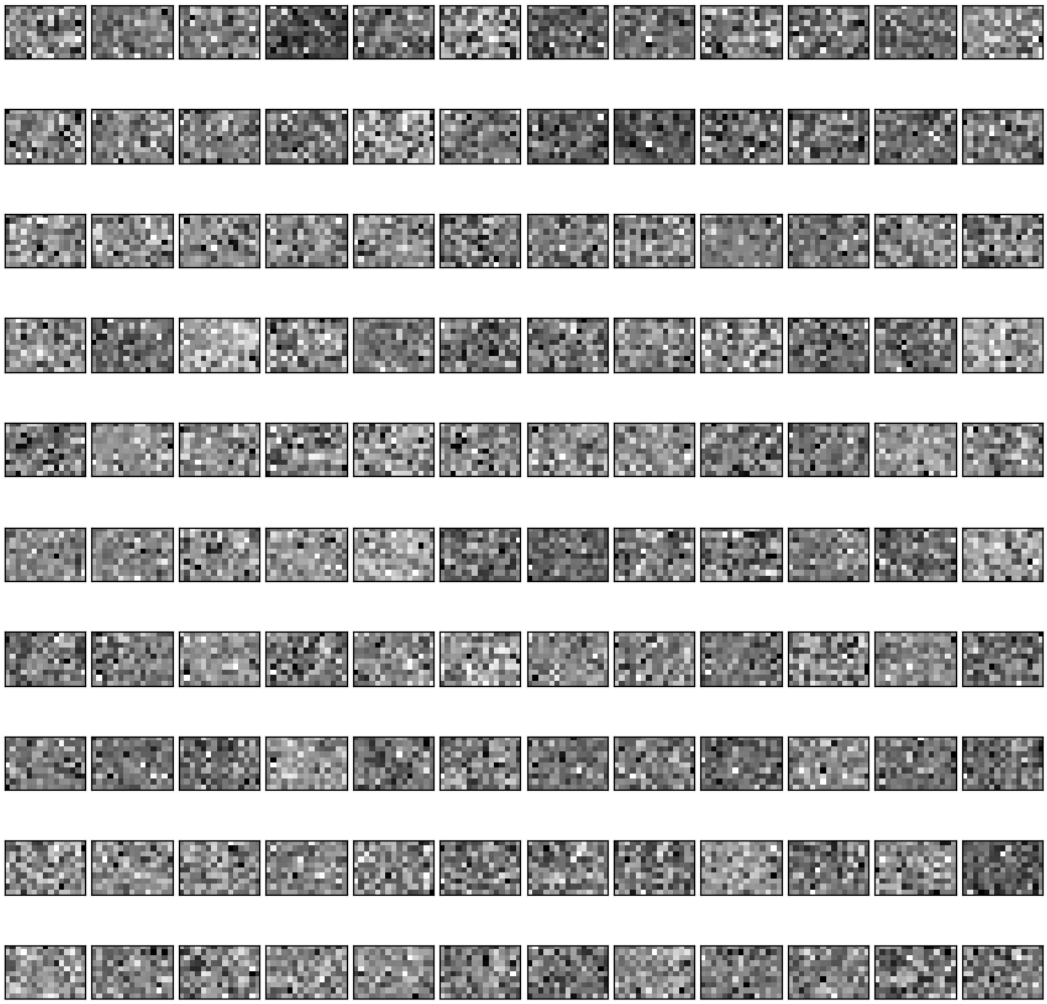

Figure 10: Weights from the 2nd hidden layer (120 nodes).

Network	**Precision**	**Recall**	**f1-score**
Stacked RBM	0.93	0.93	0.93
MLP	0.948	0.948	0.948

Table 4: Performance comparison of stacked RBM and MLP.

all they both perform quite well. However, by using RBM the classification is done using one extra simple layer of just ten nodes, while in the MLP we use 3 hidden layers. So we can see that the dimensionality reduction of data is more successfully performed by the stacked RBMs.

Autoencoder

We created and trained a stacked autoencoder with 1, 2, 3 hidden layers plus a classifier layer. 150 nodes were used for the first hidden layer, 120 nodes for the

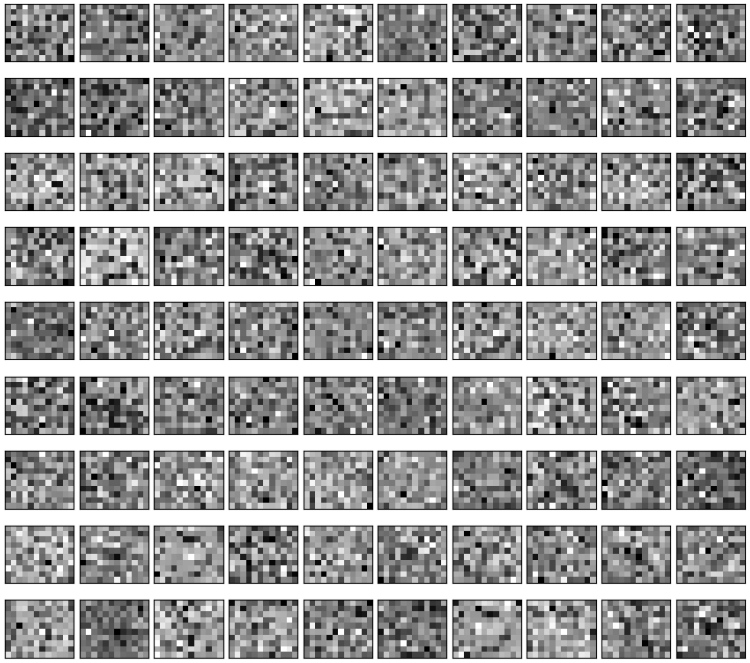

Figure 11: Weights from the 3rd hidden layer (90 nodes).

second and 90 for the third. The stacked autoencoder was pretrained layer by layer and the output of the resulting network was used to train a simple logistic regression classifier layer. We used the sigmoid and ReLu activation functions. We also trained a configuration with no hidden layer (just a simple classifier), in order to be able to have a better perspective onto the performance of our deep networks. We found that the classification performance gets a little bit worse when we use more autoencoder layers, instead of getting better. This is not so strange to happen, as when using more layers we force more and more lossy compression onto our data. But along this tiny loss in accuracy, comes the additional advantage of managing to greatly compress the data (784-dimension data turn into 90-dimension data with similar classification performance). The exact results we managed to obtain are presented in table 5.

#hidden layers	**Precision**	**Recall**	**f1-score**
0	0.88	0.88	0.88
1	0.86	0.86	0.86
2	0.87	0.87	0.87
3	0.85	0.85	0.85

Table 5: Performance of the classifier when using different hidden layers for the pretraining with a deep autoencoder.

The recreation of the digits, using the stacked autoencoders (without the classification layer), was found to be very precise, as shown in figure 12.

Figure 12: Recreation with a three layer autoencoder.

The weights are presented in figure 13, 14 and 15 for the first, second and third hidden layer respectively. We can see that, in the same way as the stacked RBMs, the weights of the first layer are digit-like figures, while the weights of the next layers are non-interpretable. That is expected to happen because the next hidden layers try to encode the already encoded digit figures, that look less and less like actual figures. We should also note that some of the 150-node weights do not look like digit. They look more like having been randomly initialized. This means that they do not encode some specific input, and thus we could omit them - we could force further compression (less nodes) without losing valuable information.

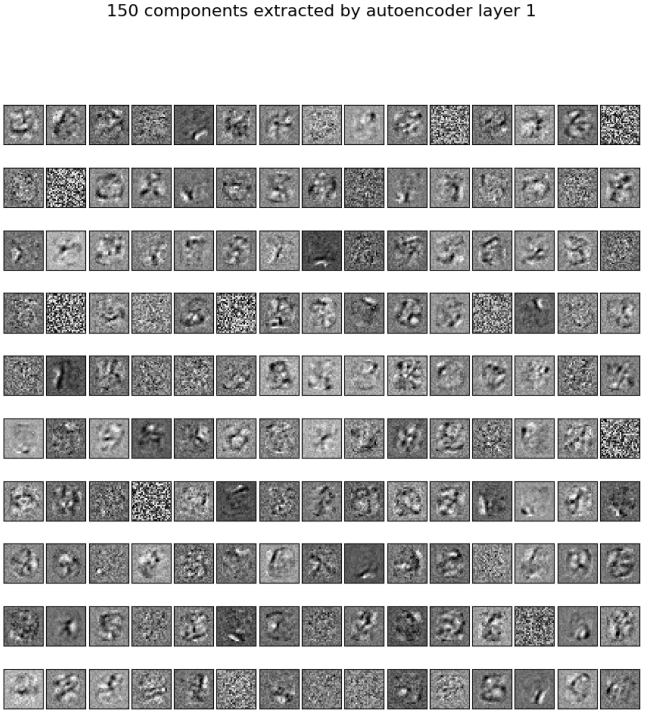

Figure 13: Weights from the first hidden layer (150 nodes).

The stacked autoencoder classifier was finally compared to the method of multilayered perceptron, shown in table 6. We can see that the MLP network has

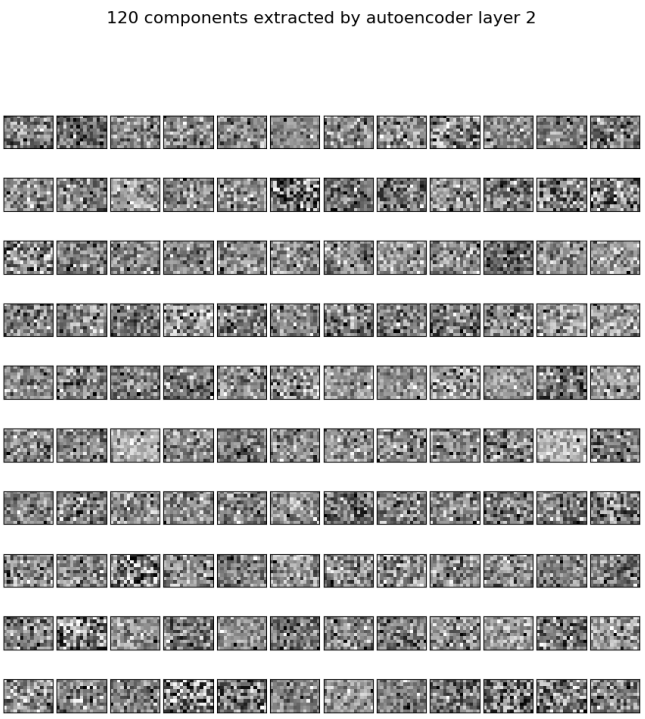

Figure 14: Weights from the second hidden layer (120 nodes).

clearly better performance when classifying the digits. But still, as with stacked RBMs, the stacked autoencoder has the additional advantage of classifying quite accurately highly compressed data, using just ten nodes in the output layer.

Network	**Precision**	**Recall**	**f1-score**
Stacked autoencoder	0.85	0.85	0.85
MLP	0.95	0.95	0.95

Table 6: Performance comparison of stacked autoencoder and MLP.

4 Conclusion

In this lab we explored the advantages of using RBMs and autoencoders as a means of lossy data compression. We built simple single-layer RBMs and autoencoders, as well as deeper architectures. We also got to know how layer-wise pretraining works, and why it is a good way to train deep networks.

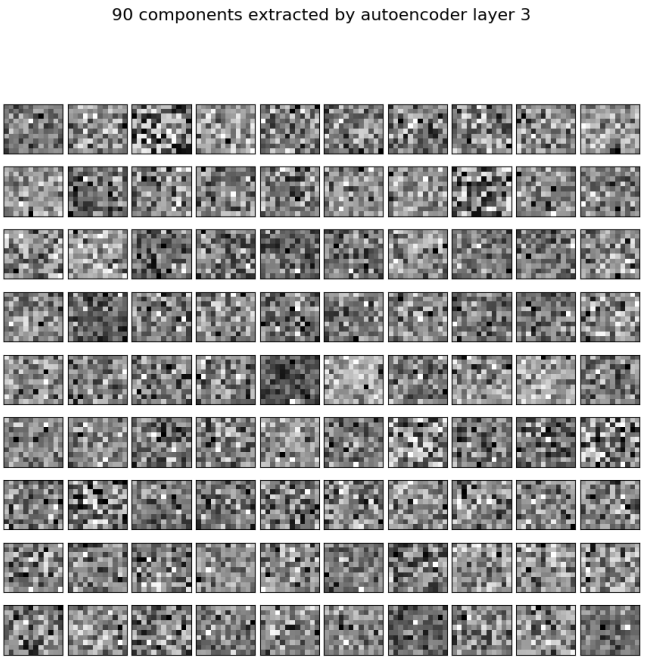

Figure 15: Weights from the third hidden layer (90 nodes).