Data Mining & Neural Networks

Report

Vincent Buekers
r0754046

Master of Statistics
G9X29A
2019-2020

Contents

1 Training Algorithms and Generalization **1**
1.1 The Perceptron and Beyond 1
1.2 Backpropagation in Feedforward Multi-layer networks 3
1.3 Bayesian Inference 5
1.4 Curse of Dimensionality 6

2 Applications: Time-series Prediction and Classification **7**
2.1 Time-Series Prediction 7
2.2 Classification 10
2.3 Automatic Relevance Determination 11

3 Unsupervised Learning and Data Visualization **12**
3.1 Principal Component Analysis 12
3.2 K-Means Clustering 14
3.3 Density Based Methods 14
3.4 Prototype Vectors and Data Visualization 16

4 Autoencoders and Convolutional Neural Networks **18**
4.1 Stacked Autoencoders 18
4.2 Variational Autoencoders 20
4.3 Convolutional Neural Networks 21

Chapter 1

Training Algorithms and Generalization

1.1 The Perceptron and Beyond

Exercise 1

As the name implies, linear regression is used to model a linear relationship between one or several input variables (X_i) and an output variable (y_i). This is in fact equivalent to the McCulloch-Pitts model of a neuron. The input variables all have their own associated weights, just like the β_j coefficients in linear regression. Moreover, a bias term is also included, similar to the the intercept β_0 in linear regression.

However, the exact equivalence between the McCulloch-Pitts model and linear regression only holds when the activation function is chosen to be linear. The equivalence is apparent when comparing the mathematical specification of linear regression (1.1) with the visual representation of a neuron as developed by McCulloch-Pitts (Fig. 1.1). Moreover, these models can also be optimized by the same objective function. That is, through a minimization of the squared errors, which is referred to as least squares in linear regression.

$$y_i = \beta_0 + \beta_1 x_{i,1} + \beta_2 x_{i,2} + ... + \beta_n x_{i,n} \tag{1.1}$$

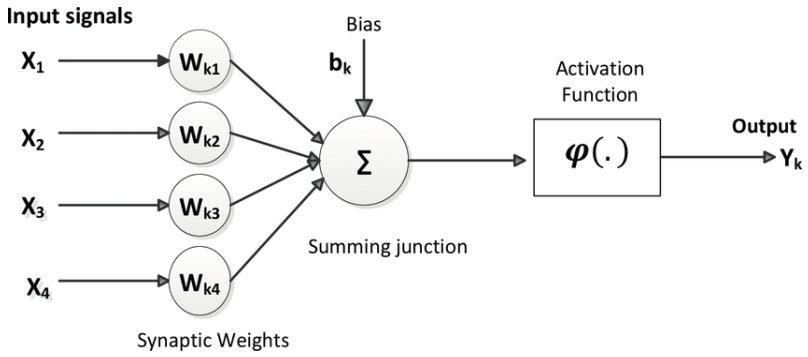


Figure 1.1: McCulloch-Pitts model of a neuron

Exercise 2

In this first example, input data $x_i \sim U[0,1]$ and output data $y_i = -sin(0.8\pi x_i)$ are considered, where $i = 1, ..., 75$. This function is visualized below (Fig 1.2). It is clear that a linear model would fail to adequately capture this relationship. In that sense, one would be underfitting this problem by simply using linear models. This is not surprising given the non-linear transformation used on the input data. Consequently, non-linear models are likely to result in a more appropriate fit for the data, which will be dealt with in the next exercise.

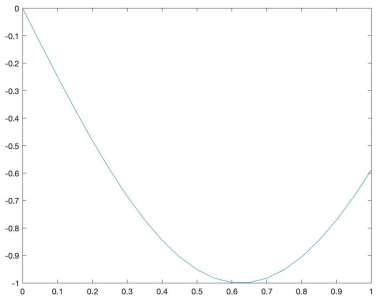


Figure 1.2: Plot of x vs $-sin(0.8\pi x)$

Exercise 4

In order to approximate the function outlined above, a neural network with one hidden layer consisting of two neurons is trained. The activation values obtained from this network are x_i^1 and x_i^2. Plotting these vectors and the corresponding y_i results in an insightful depiction of the network functioning (Fig. 1.3). Modelling y with (x_i^1, x_i^2) seems rather straightforward. That is, the weights should be set such that x_1 is emphasized in the upward sloping part of y and x_2 in the downward sloping part of y.

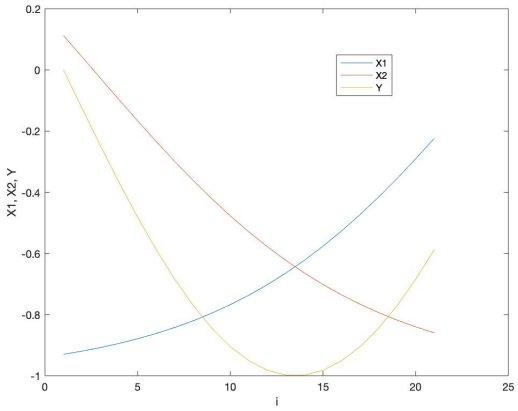


Figure 1.3: Activation values

Exercise 6

Lastly, the output obtained from the neural network results in an excellent approximation of y (Fig. 1.4). In fact, the values are identical to those of the labeled data. Since neural networks are universal approximators, this is a natural result. However, such perfect interpolation might be considered overfitting in applications where generalization on unseen data is of interest.

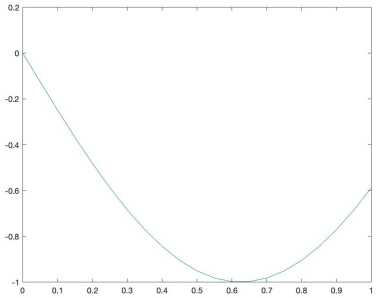


Figure 1.4: Output of Neural Network

1.2 Backpropagation in Feedforward Multi-layer networks

Exercise 1: Function approximation

Consider $x_i \sim U[0, 3\pi]$ and their corresponding $y_i = sin(x_i^2)$ for $i = 1, ..., 75$. A neural network with one hidden layer is trained to approximate this function. It is of interest to compare several training algorithms in terms of their efficiency by means of evaluating the regression between output and targets. All else equal (i.e. equal epochs and neurons), the Levenberg-Marquardt algorithm gives the best approximation of the function (Fig 1.5). In that regard it outperforms the other algorithms such as (adaptive) gradient descent or Quasi-Newton. Training performance aside, it remains to be seen how it is able to generalize on noisy data.

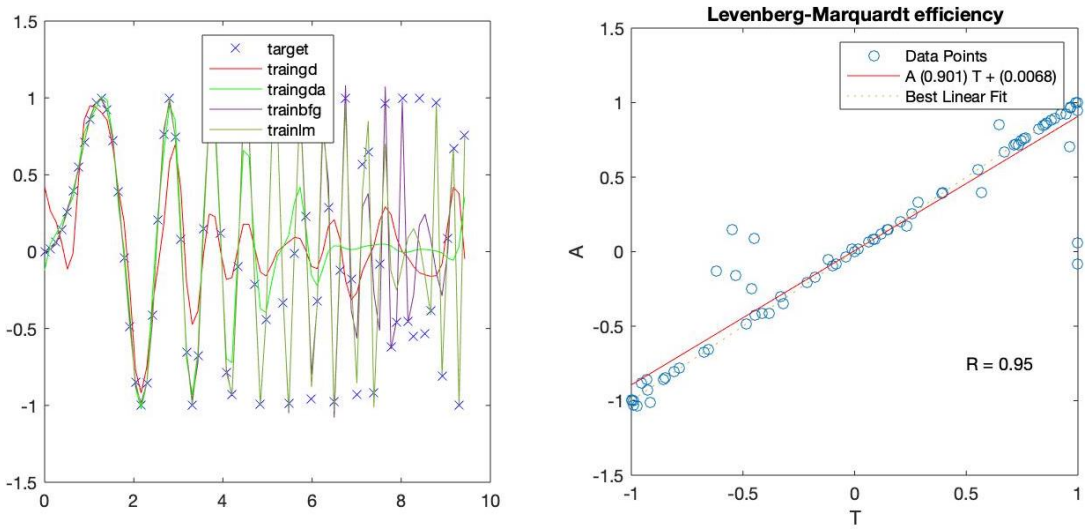


Figure 1.5: Training algorithms

Exercise 2: Generalization with noisy data

After having added noise $e_i \sim \mathcal{N}(0, 1)$ to the input data x_i, the efficiency decreased slightly from 0.95 to 0.907 (Fig. 1.5, left). Next, the noise level has also been doubled for twice the amount of data points, i.e. $e_i \sim \mathcal{N}(0, 2)$ for $i = 1, ..., 150$. Once again, performance decreases, seemingly proportional to the amount of noise that is added (Fig 1.5, right). This could indicate that the algorithm has a tendency to overfit on the training data, as it seems to substantially suffer from noise contributions, increasingly so for larger noise levels.

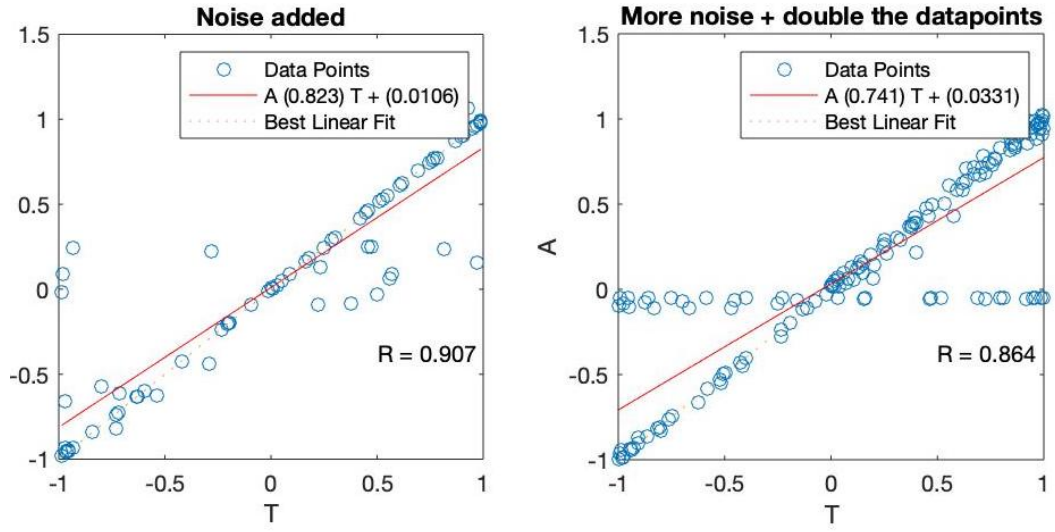


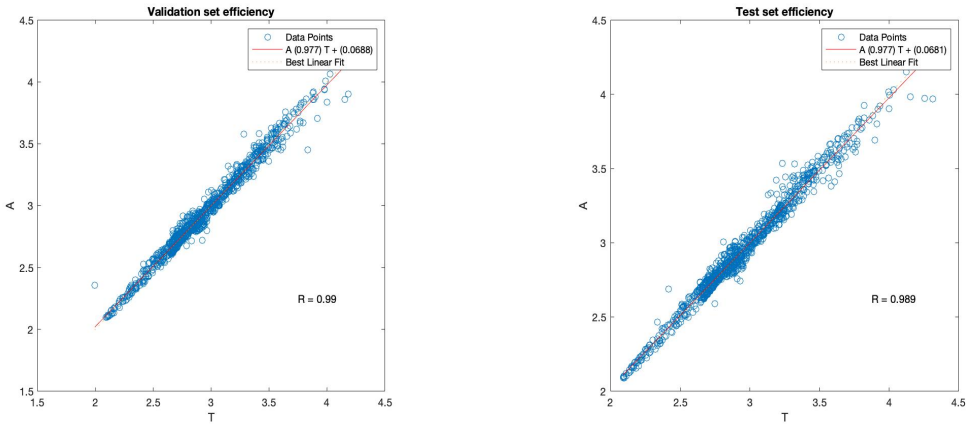
Figure 1.6: Efficiency Comparison

Exercise 3: Personal Regression Problem

In this example, an unknown nonlinear function needs to be approximated using 13600 values sampled from it. This includes inputs X_1 and X_2 and nonlinear functions $\{T_1, ..., T_5\}$ of (X_1, X_2). With my student number being **r0754046**, the target T_{new} has been constructed as follows:

$$T_{new} = \frac{(7 * T1 + 6 * T2 + 5 * T3 + 4 * T4 + 4 * T5)}{(7 + 6 + 5 + 4 + 4)}$$

From this data set, 3 independent samples of 1000 observations have been drawn, yielding a training, validation and test set as such. In an attempt to approximate the unknown function T_{new}, a neural network is trained, consisting of one hidden layer with 8 neurons. A single hidden layer should suffice as this already yields a universal function approximator. Furthermore, the Levenberg-Marquardt algorithm has been used for training since it gave the best results on the validation set (Fig 1.6). Also, the hyperbolic tangent function is used for the hidden layer, whereas the output layer will have a linear activation function. Such a specification is commonly adapted in regression settings.



(a) Validation set

(b) Test set

Figure 1.7: Performance

The performance on the validation set is very high using the model outlined above ($R = 0.99$). Fortunately, the generalization to the unseen test data turned out to yield a very similar efficiency ($R = 0.989$). As such, the neural network specifications based on the validation set would seem justified towards its generalization capacity. The model predictions are visualized below (Fig. 1.7, red circles), for which the corresponding Mean Squared Error (MSE) on the set is 0.0029.

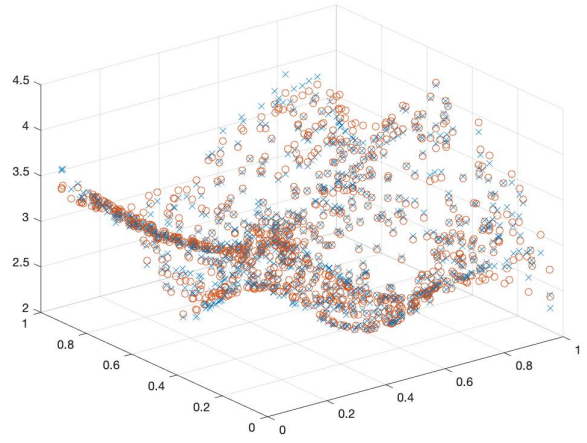


Figure 1.8: 3D Scatter plot of test data and model approximations

1.3 Bayesian Inference

Once more the data $x_i \sim [0, 3\pi]$ and $y_i = sin(x_i^2)$ are considered. Again, a noise contribution $\epsilon_i \sim \mathcal{N}(0, 1)$ to the the input data x_i is also used. This time, Bayesian regulation is implemented as the training algorithm, in which the training function updates the weight and bias values according to Levenberg-Marquardt optimization. The objective function (1.2) is optimized with respect to w for which the level 2 hyperparameters (α, β) are determined such that the network generalizes well. More specifically, the suppression of the weights is controlled by the regularization term α in level 1, whereas in level 2 it also determines effective number of parameters.

$$\min_w M(w) = \beta E_D(w) + \alpha_W(w) \tag{1.2}$$

To illustrate the regularisation mechanism, networks with a moderate amount of neurons (10) and overparametrized networks (100 neurons) are trained using both the noiseless and the noisy data. Furthermore, a comparison is made with the regular Levenberg-Marquardt algorithm, which does not implement weight suppression through regularization. The networks are evaluated based on the MSE obtained on the test set. To that end, the training data ($n = 75$) is divided into training and test sets with respective ratios 2/3 and 1/3.

Results for the MSE on train and test are tabulated below (Table 1.1, Table 1.2). Both algorithms obtain respectable results on the noiseless data. As expected, there are slight performance decreases on noisy data in both cases. In the case of overparametrized networks, the regularization mechanism is well demonstrated. While the Levenberg-Marquardt algorithm quite heavily overfits on the training set, Bayesian regulation prevents itself to do so. As opposed to a practically zero training error, the weight suppression results in a reasonable training error. This, in turn, leads to a better generalization capacity since training and testing errors are more in line. Clearly, this does not hold for the LM algorithm, or any non-regularized training algorithm for that matter. That is, the test performance is inferior to that of Bayesian regulation due to the overfitting on the training data.

(a) Noiseless data

	Regular	Overparametrized
train MSE	0.4022	0.0263
test MSE	0.5831	0.4239

(b) Noisy data

	Regular	Overparametrized
train MSE	0.3555	0.1189
test MSE	0.5342	0.8849

Table 1.1: Bayesian Regulation

(a) Noiseless data

	Regular	Overparametrized
train MSE	0.4022	3.8670e-22
test MSE	0.5831	0.6007

(b) Noisy data

	Regular	Overparametrized
train MSE	0.1682	7.5326e-22
test MSE	0.6270	1.0612

Table 1.2: Levenberg-Marquardt

For a network consisting of 100 neurons with 1 input and 1 output, the interconnection weight matrices are the following:

$$W \in \mathbb{R}^{1\times 100}, V \in \mathbb{R}^{100\times 1}, \beta \in \mathbb{R}^{100}$$

Thus yielding 300 parameters, which is clearly excessive for the task at hand. During training of the networks implementing Bayesian regulation, this amount was reduced to 47 and 28 for the noiseless data and the noisy data respectively. As such, the effective number of parameters was significantly less than the actual model specification. Hence, the regularization constant α demonstrates its influence in level 2 inference based on the evidence provided by level 1, in which it already controlled the weight sizes.

1.4 Curse of Dimensionality

In this last part of assignment 1, it is of interest to evaluate the approximation of a sinus cardinal (1.3) of domain radius 4 in different dimensions via the provided program `curse.mlx`.

$$f(x) = sinc(||x||_2), x \in \mathbb{R}^d \tag{1.3}$$

In this analysis, both a neural network and a polynomial are evaluated in different dimensions. To that end, a fifth-degree polynomial and a neural network with two hidden layers (4 and 2 neurons) are considered. In 2-dimensional space, these model specifications yield a similar amount of parameters (Fig. 1.9, a) All else equal, it can be noted that the amount of parameters substantially increase in higher dimensions, especially in the case of the polynomial approximation (Fig. 1.9, b). While a fifth-degree polynomial is not parsimonious in higher dimensions, a neural network is better able to cope with this increased dimensionality. Moreover, it is able to achieve a a very similar performance (MSE) on both training and test set with only a small fraction of the parameters.

Dimension: 2
Domain: radius=4.0 volume=5.03e+01
Training set size: 5000
Test set size: 500

	POLYNOMIAL	NEURAL NETWORK
Number of parameters:	21	24
Datapoints/parameter:	238.1	208.3
RMSE (Train):	2.12e-03	3.71e-03
RMSE (Test):	7.78e-03	1.18e-03
Training time [s]:	8.86e-02	4.87e+00

(a) d = 2

Dimension: 7
Domain: radius=4.0 volume=7.74e+04
Training set size: 5000
Test set size: 500

	POLYNOMIAL	NEURAL NETWORK
Number of parameters:	792	44
Datapoints/parameter:	6.3	113.6
RMSE (Train):	7.68e-04	7.33e-03
RMSE (Test):	2.62e-03	2.26e-03
Training time [s]:	2.17e+00	9.03e-01

(b) d = 7

Figure 1.9: Dimensionality

Once more, the approximators are trained in a higher dimension (5). Yet now additional model parameters are introduced. The polynomial has been expanded into seventh order, whereas the amount of neurons in the two hidden layers have been doubled (8 and 4). It can be noted that the polynomial approximation heavily suffers from the increased parametrization (Fig. 1.10, a). Even though this leads to a higher training and test performance, it is not proportional to what the neural network can attain with much less parameters.

Lastly, the sample size has been doubled with increased noise level (Fig 1.10, b). With its relatively parsimonious parametrization, the neural network is able to generalize very well to the test data, even with increased noise variance. Although the polynomial is able to do so as well, once again the amount of parameters is problematic in higher dimensions.

Dimension: 5
Domain: radius=4.0 volume=5.39e+03
Training set size: 5000
Test set size: 500

	POLYNOMIAL	NEURAL NETWORK
Number of parameters:	792	88
Datapoints/parameter:	6.3	56.8
RMSE (Train):	7.14e-04	2.82e-03
RMSE (Test):	2.48e-03	6.87e-04
Training time [s]:	2.11e+00	2.71e+00

(a) Increased parametrization

Dimension: 5
Domain: radius=4.0 volume=5.39e+03
Training set size: 10000
Test set size: 1000

	POLYNOMIAL	NEURAL NETWORK
Number of parameters:	252	60
Datapoints/parameter:	39.7	166.7
RMSE (Train):	7.47e-04	2.21e-03
RMSE (Test):	2.40e-03	7.23e-04
Training time [s]:	7.62e-01	5.30e+00

(b) Increased noise level and sample size

Figure 1.10: Parametrization, sample size and noise level

Chapter 2

Applications: Time-series Prediction and Classification

2.1 Time-Series Prediction

Santa Fe

Consider the Santa Fe data, consisting of 1000 training data points (Fig. 2.1, left) and 100 test measurements (Fig. 2.1, right). The objective is to train a feedforward neural network that performs well on the test data. That is, an appropriate amount of lags and neurons (hyperparameters) has to be determined. To that end, varying amount of lags and neurons are considered, yet all consisting of one hidden layer since it boils down to a nonlinear function estimation problem.

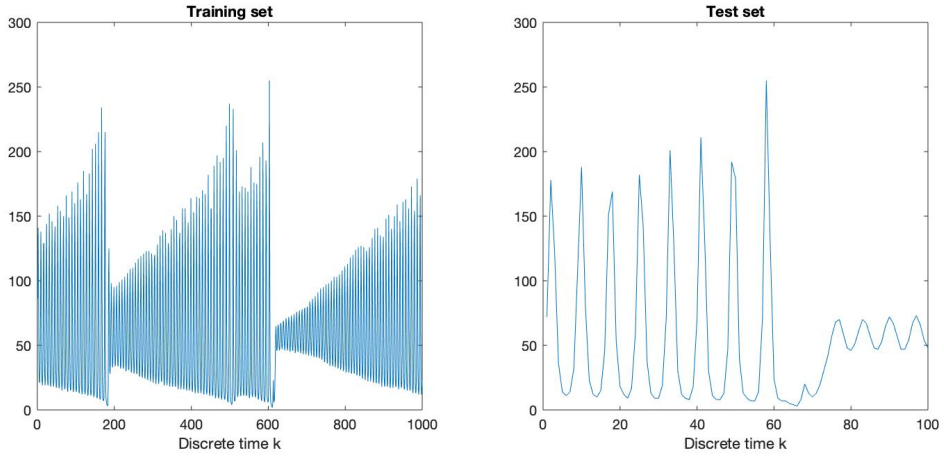
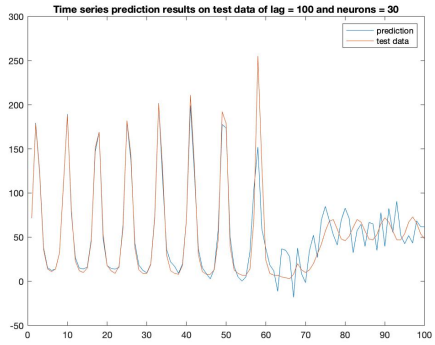


Figure 2.1: Santa Fe data

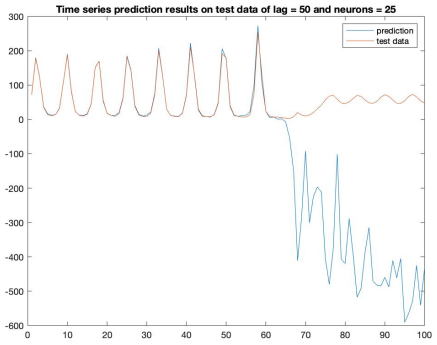
The impact of both the input pattern (i.e. lag length) and the parametrization on the performance of the network is quite substantial. Indeed, the quality of the test data predictions is drastically different for two example models (a) and (b) (Fig. 2.2). For instance, model (a) seems to perform quite well in comparison to model (b). Note that here appear to be two distinct time dynamics in the test data. Although model (b) fails terribly on the larger forecast horizon, it seems to outperform model (a) on the shorter forecast horizon. Overall, (a) should still be preferred over (b).

However, to make matters less heuristic, models are evaluated by means of performances metrics such as (root) mean square error. Note that the RMSE reverts the error back to the scale of the measurements, thus allowing for more convenient interpretation. For several models, these metrics are computed (Table 2.1). As I'm only equipped with commodity hardware, I haven't tried all possible combinations of lags and neurons. In that sense, these options are merely illustrative.

Also note that, to make training somewhat time-efficient, a maximum of 50 epochs is set for all models. Among the listed models, a specification of 10 neurons and 100 lags seems to achieve the best relative performance.



(a)



(b)

Figure 2.2: Example test data predictions

(a) MSE

	Neurons	
Lags	10	20
25	138866.46	18689.18
50	33081.64	4822.19
75	8568.95	31916.70
100	3195.49	3904.52

(b) RMSE

	Neurons	
Lags	10	20
25	372.64	136.70
50	181.88	69.44
75	92.56	178.65
100	56.52	62.48

Table 2.1: Performance Metrics test set

Note that these metrics are indeed calculated with respect to the test data. However, with generalization in mind, it is not entirely sensible to adjust model hyperparameters such that the this specific test set is optimally predicted. Rather, a particular model should be able to generalize to any form of unseen data. As such, it would be more sensible to hold out a validation set to fine tune the hyperparameters. Moreover, it is also important to note that, in the context of time series, predictions are made in a recurrent manner in the following sense:

$$\hat{y}_{k+1} = \mathbf{w}^T tanh(\mathbf{V}[\hat{y_k}; \hat{y}_{k-1}; ...; \hat{y}_{k-p}] + \beta) \tag{2.1}$$

In this way, up to p lags are considered in the prediction of a certain y_{k+1}. Without reserving a validation set, this would also cause training data to be involved in the prediction process of the test set. Hence the need for a validation set. Furthermore, I have noticed great variability between models of the same specification. Therefore, several iterations of model training have been conducted, making it possible to average performance results over multiple runs. Among the considered hyperparameter specifications, the validation set error is relatively optimal for a model consisting of 10 neurons, taking into account 100 lags (Table 2.2). After all, this seems to be in line with what was indicated earlier (Table 2.1).

(a) MSE

	Neurons	
Lags	10	20
25	97127	96049
50	14973	6084
75	5952	15662
100	3304	5036

(b) RMSE

	Neurons	
Lags	10	20
25	311.65	309.91
50	122.36	78
75	77.14	125.14
100	57.48	70.96

Table 2.2: Performance Metrics validation set (average over multiple runs)

PM 2.5

In this section, data about Shanghai Particulate Matter (PM) is analyzed in an attempt to select a suitable model to predict the PM 2.5 concentration. The data set contains 4344 observations, yet only the first 1000 are taken into account in this example, of which the first 700 serve as training set and the last 300 serve as test set. Note that there are missing values among these data points. As opposed to leaving these untreated, they will be replaced by first order interpolation values. Using this method, the data set visualized in Fig 2.3 is obtained.

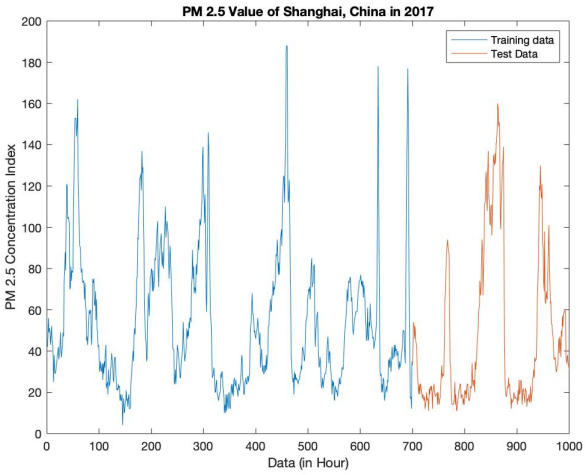


Figure 2.3: Shanghai Data

The objective is to identify a suitable model which, like in the previous example, will depend on the selection of the lags and neurons in the hidden layer. Once again, only one hidden layer should suffice for nonlinear function estimation. However, for illustrative purposes, several models will also be extended to a double hidden layer architecture. To make training somewhat time-efficient, a maximum of 50 epochs is set. Once more, a validation set is held out to evaluate the best hyperparameters for the problem, because of reasons mentioned in the preceding section. Also, several runs have again been conducted to obtain average performance results for a certain model specification (Table 2.3).

(a) One hidden layer

	Neurons	
Lags	10	20
50	11018	8821
60	16075	9124
70	8886	11132
80	3005	4542
90	4041	2543
100	5879	4082
AVG	9780.8	8048.8

(b) Two hidden layers

	Neurons	
Lags	(10, 10)	(20, 20)
50	15123	12918
60	2733	5390
70	6067	3378
80	15000	3775
90	4743	5700
100	3135	5203
AVG	7800	6060.6

Table 2.3: Performance Metrics validation set (average over multiple runs)

When only one hidden layers is considered, it would seem a model consisting of 20 neurons, taking into consideration 90 lags, results in the best relative performance. In contrast, 20 neurons and 60 lags would seem relatively best when two hidden layers are considered.

Note that, on average, a two hidden layer does outperform a one hidden layer network. As such, this increased parametrization doesn't yet seem to lead to overfitting. On average, a two hidden layer model with 20 neurons in each layer performs best, regardless of the amount of lags that are taken into consideration.

2.2 Classification

In this section, it is of interest to model the Breast Cancer Data Set (binary classification). Considering these are very high dimensional data (30 input variables), standard visualization techniques become troublesome. To that end, I considered a nonlinear dimensionality reduction technique known as t-SNE. This technique aims to learn a low dimensional map that reflects the (dis-)similarities between datapoints in higher dimensions (Fig. 2.4). Using this method, both the test and training can be represented in terms of a 2-dimensional embedding. Although these embeddings are a measure of (dis-)similarity, their interpretation is much less evident. Nonetheless, they provide insights in the shared characteristics between cancer patients and the healthy control group on the other hand.

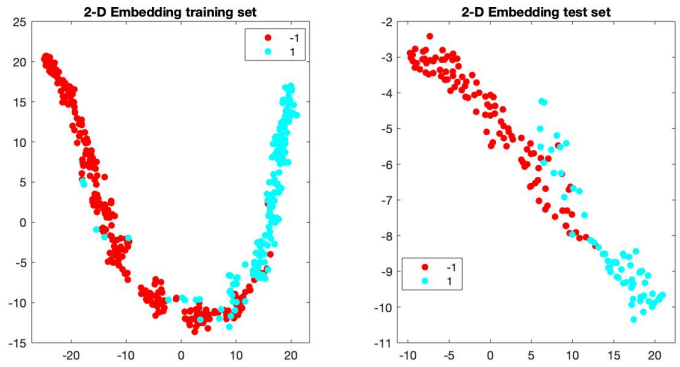


Figure 2.4: t-SNE: Graphical Representation of train and test set

Next, the objective is to train a multilayer perceptron in the form of a classifier. To that end, several training algorithms are considered, namely Levenberg-Marquardt, Bayesian regulation and gradient descent. Moreover, two hidden layers are considered for the realization of non-convex regions, each consisting of 10 neurons. The number of epochs have been set to 50. Furthermore, 10-fold cross validation has been implemented during training. With this approach, the classifications performance is satisfactory for all training algorithms (Fig. 2.5). Moreover, the average performance over all 10 folds is very close to the test set performance for all algorithms (Table 2.4). In that sense, the networks generalize well.

	AUC test set	10-fold AUC average
Levenberg-Marquardt	0.9852	0.9653
Bayesian regualtion	0.9897	0.9547
Gradient descent	0.9694	0.9597

Table 2.4: Classification Performance

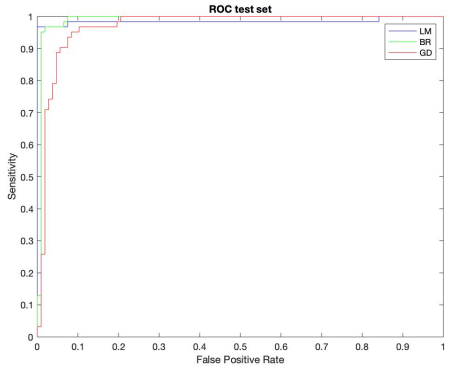


Figure 2.5: ROC test set for different training algorithms

2.3 Automatic Relevance Determination

In this last part of assignment 2, the breast cancer data set from the previous section is revisited. This time, Automatic Relevance Determination (ARD) is implemented to determine the most relevant inputs. After all, identifying the most important variables from such high-dimensional data could be very informative in medical applications regarding breast cancer for instance, while also reducing training time. The implementation of ARD is done using the provided program `demard.m`

With ARD, regularization constants $\alpha_1, ..., \alpha_{30}$ are assigned to the corresponding 30 input variables of the breast cancer data. The prior over these weights is of a Gaussian type for the group of weights associated with each input. The re-estimation of these parameters is based on the level 1 evidence for the corresponding weights. As such, the α_i ultimately reflect the importance of certain input variables with respect to the targets (i.e. breast cancer group). As it turns out, variable 4 and 24 have the lowest associated α and the highest associated weights in both layers. Since, the α_i correspond to the inverse posterior variance, it makes sense that the weights are placed such that the inputs containing the most information (i.e. highest posterior variance) are emphasized by higher weights.

Next, the networks from the previous section are trained once again using only the aforementioned variables, presumed to be most important according to ARD. For comparative purposes, the hyperparameters from the previous section are used again. Hence cross-validation is no longer implemented for hyperparameter tuning. Also, previously considered training algorithms are again used to see how each of them are affected by the downsized input data. The obtained results are given below (Table 2.5) and are once more visualized by their corresponding ROC curves (Fig. 2.5).

	AUC test set
Levenberg-Marquardt	0.9483
Bayesian regualtion	0.9864
Gradient descent	0.8190

Table 2.5: Performance comparison (ARD variables)

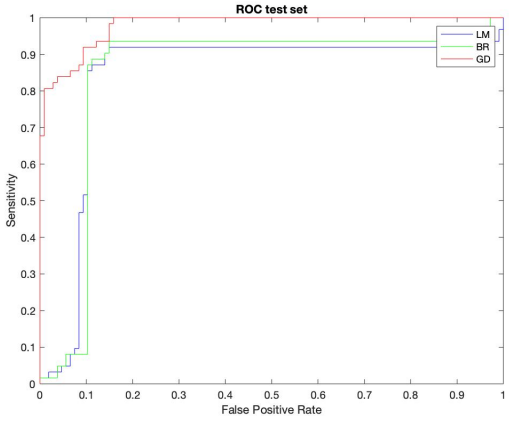


Figure 2.6: ROC test set for different training algorithms after ARD

Even though only 2 out of the original 30 input variables are retained, the networks are still able to attain quite a respectable performance on the test data. Bayesian regulation is doing particularly well, attaining only a slightly lower performance than it did on the full data. This could hint at the appropriate functioning of ARD since Bayesian regulation could have already been supressing the weights corresponding to the less important input variables. Gradient descent seems to suffer most from the retention of information.

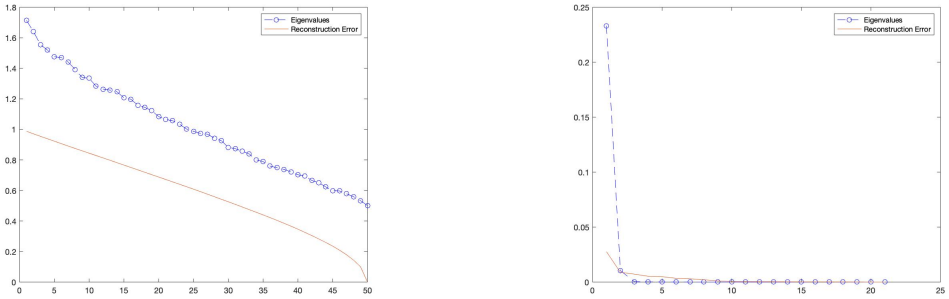
Chapter 3

Unsupervised Learning and Data Visualization

3.1 Principal Component Analysis

Redundancy and Random Data

In this section, random data of dimension $d = 50$ are generated. Furthermore, the highly correlated `choles` data are also considered ($d = 21$). It is of interest to reconstruct these matrices using Principal Component Analysis (PCA). When looking at the eigenvalues of the covariance matrix of the centered data, it is troublesome to identity the appropriate amount of components to retain for the random data (Fig 3.1, a). In contrast, it is quite easy to see that for the redundancy data, 1 component should be sufficient (Fig 3.1, b). Note that the decrease in reconstruction error is proportional to the amount of components that are considered.



(a) Random Data

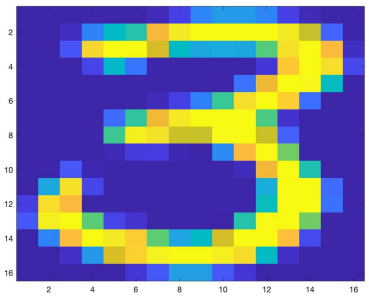
(b) Redundancy Data

Figure 3.1: Scree plots

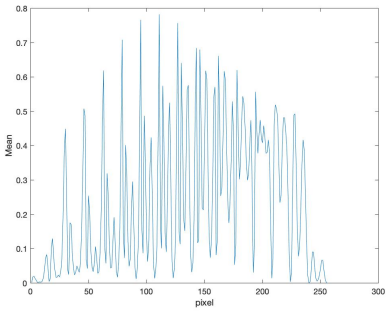
Note that, as opposed to the random data, the redundancy data are very highly correlated (correlation coefficient attains values between 0.8623 and 1). As such, it is not surprising that the identification of principal components becomes much clearer. That is, a single eigenvalue can represent a lot of the variance in the data along this one dimension. Unlike the manual method used above, it is also possible to conduct a PCA analysis based on the amount of variance contribution to the total variation. With this approach, the choice of the amount of components follows indirectly. Depending on the application, one could a priori set a desired variance contribution.

Handwritten Digits

In this section, PCA is conducted on the digit 3 from the US postal Service database. Before delving into the image compression, the mean vector of the $16 \times 16 = 256$ pixels is evaluated for exploratory purposes (Fig 3.2, b). As the digits are brightly coloured, it would be logical if the pixels with higher mean values correspond to the pixels that comprise the handwritten digits themselves.



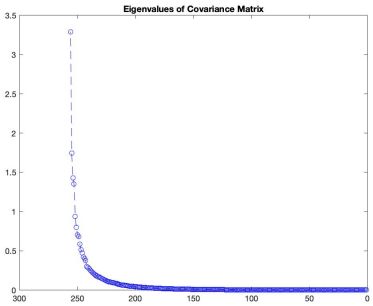
(a) Random 3 from the data set



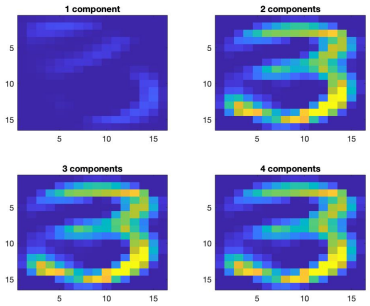
(b) Mean Vector of pixels

Figure 3.2: Handwritten Digits Data

Next, it is desired to compress the images using $k = 1, 2, 3, 4$ principal components. As it turns out, 1 principal component seems to result in quite a poor reconstruction (Fig 3.3, b), whereas the additional second component adds a lot of sharpness to the reconstructed image (Fig 3.3, b). There is some minor improvement for 3 and 4 components, yet not as substantial. If these were not image data, it would harder to determine a good choice of k based on only on the eigenvalues of the covariance matrix (Fig 3.3, a).



(a) Eigenvalues of Covariance Matrix



(b) Reconstructed 3's

Figure 3.3: Handwritten Digits Data

Lastly, the reconstruction error is evaluated for $k = 1, ..., 50$ components. In line with the previous finding, it decreases drastically from 1 to 2 components, whereas the decline becomes less steep for additional components. For $k = 50$, the error already seems to be very close to 0. For $k = 256$ (i.e. complete information retention), it is still not exactly 0 due to the reconstruction process. Note that the decrease in the squared deconstruction error is proportional to its eigenvalue since it tells us how much we distort the original space by our deconstruction.

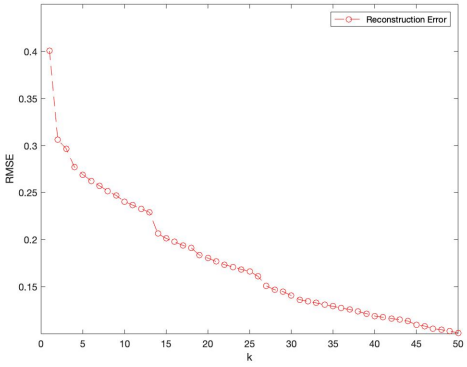
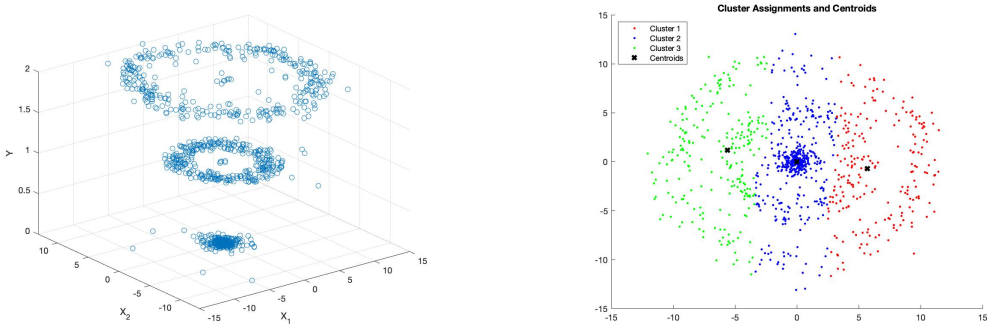


Figure 3.4: Reconstruction error for $k = 1, ..., 50$

3.2 K-Means Clustering

In this section, the K-means algortihm is briefly described w.r.t. the clustering of the `rings` data set. Based on the associated labels, the data is clearly divisible into 3 clusters a priori (Fig. 3.5, a). However, an unsupervised learning algorithm such as K-means struggles to identify these 3 ring-shaped clusters. This is mainly due to the fact that it tries to establish spheroidal patterns. For inputs (X_1, X_2) the rings are contained within each other and thus cause problems for K-means. Moreover, centroids don't necessarily make sense for the outer rings.



(a) Data set (b) Clustering

Figure 3.5: K-Means: rings data

3.3 Density Based Methods

Gaussian Mixture Model

After applying a Gaussian Mixture Model (GMM) to the rings data set, the results below (Fig. 3.6) are obtained. While it is appropriate to consider different distributions for the 3 ring-shaped clusters, a Gaussian model inherently assumes corresponding mean and variance parameters. The mean parameter is likely causing problems similar to the centroids in K-means. Furthermore, assuming a shared covariance structure results in the worst fit. Indeed, the outer most rings should be assumed to have a larger variance. While a full covariance matrix independently adopts any position and shape, a diagonal covariance uses contour axes are oriented along the coordinate axes. The latter assumption would make more sense for this data set.

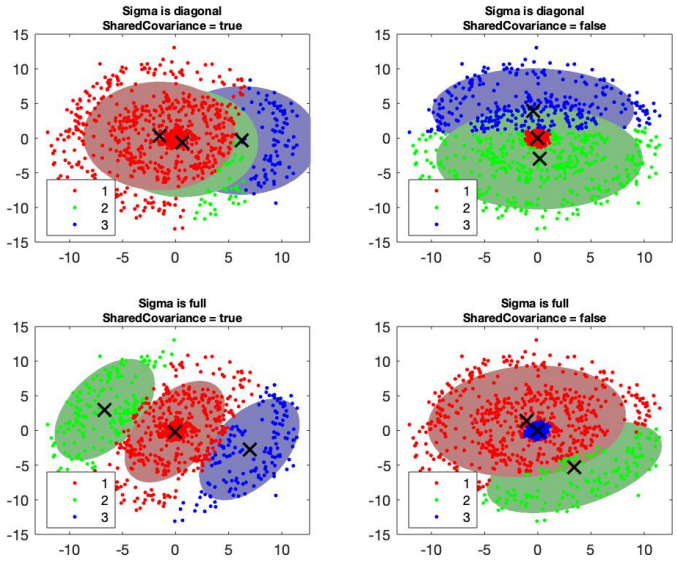
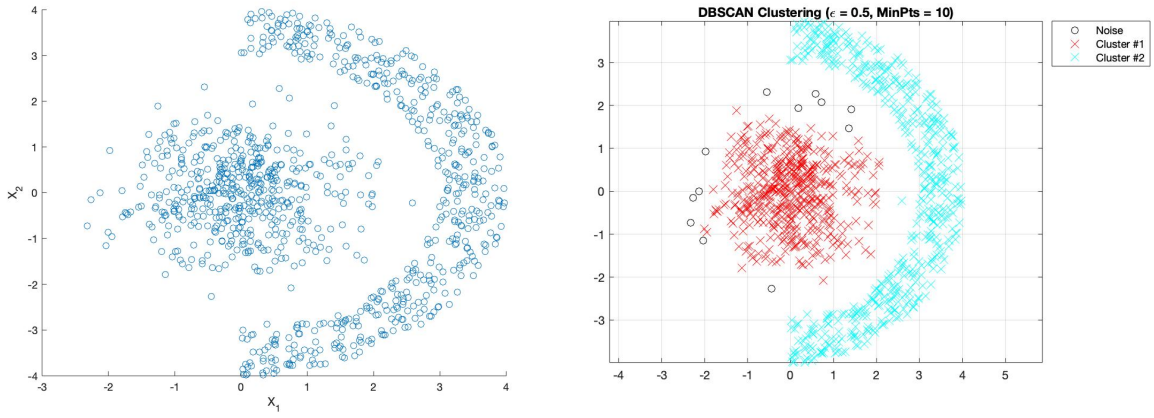


Figure 3.6: GMM: rings data

DBSCAN

Before applying DBSCAN to the rings data, the `DBSCAN.m` demo is used to illustrate its working. As can be seen, the example data are characterized by two distinct shapes (Fig 3.6, a). K-means and GMM would probably be able to identify the spheroidal cluster, yet they would struggle to create a cluster for the moon-shaped pattern. Yet DBSCAN is able to correctly identify these patterns without prior knowledge on the amount of clusters. Moreover, it is also able to identify outliers (Fig 3.6, b).



(a) Data set

(b) Clustering

Figure 3.7: DBSCAN illustration

Once again, the `rings` data set is considered. As it turns out, the DBSCAN algorithm is able to attain a very good clustering performance. In that sense, clearly outperforms both K-mean and GMM on these data. Moreover, it demonstrates its capacity to identify arbitrarily shaped patterns. In this example, the absence of cluster centroids algorithm makes sense for the two outer most shapes, whereas K-means and GMM inappropriately assumed a center for each cluster. Note that the hyperparameters $\epsilon = 0.5$ and the minimal amount of points per cluster were adopted from the toy example in `DBSCAN.m`.

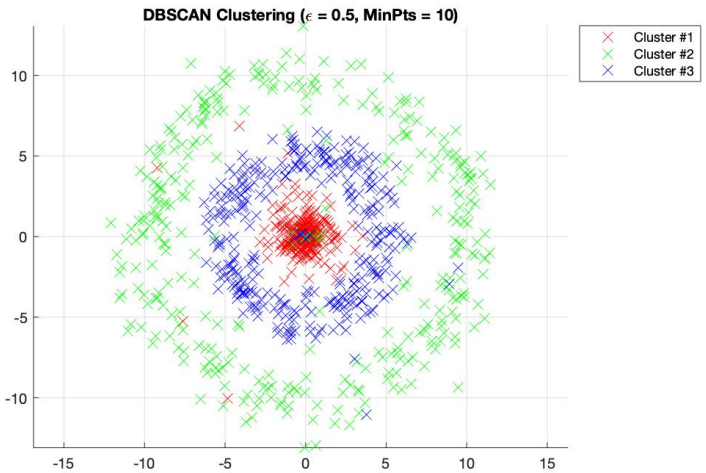


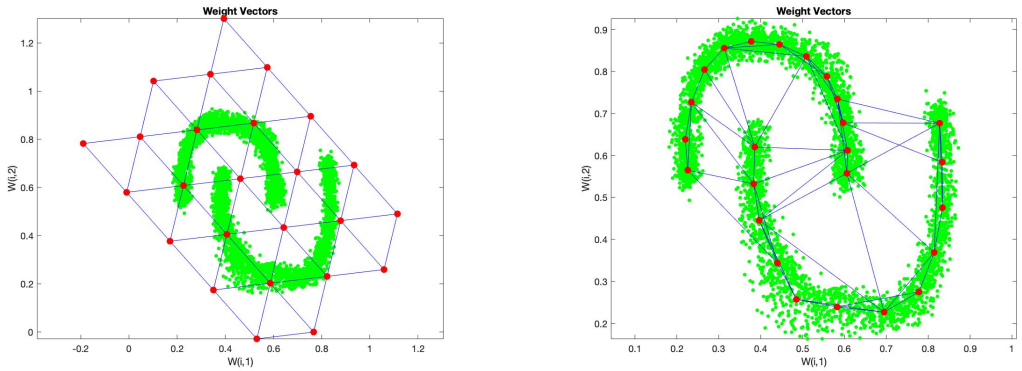
Figure 3.8: DBSCAN: rings data

3.4 Prototype Vectors and Data Visualization

This last section of assignment 3 explores Self-Organizing maps (SOMs) based on vector quantization methods for representing input vectors by representative prototypes. This is mainly geared towards the visualization of large-scale data, both in terms of observations and features.

Banana Data

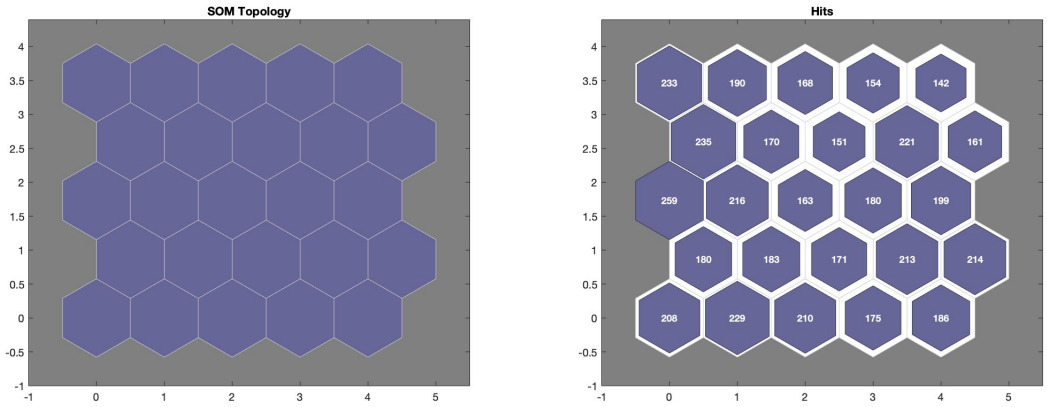
The `banana` data are considered to demonstrate the working of vector quantization by means of a toy example. The purpose is to characterize the banana data distribution, by a set of prototype vectors. To that end, a self-organizing map is initialized, yielding the prototypes in the form of untrained weight vectors (Fig. 3.9, a). Note that the hexagonal topology function has been used, along with the Euclidean distance function. Clearly, these initialized weight vectors don't correctly represent the data distribution yet. After training, the positions of these prototypes should change to a more appropriate representation. Indeed, the trained weights end up accurately characterizing the two banana shaped data clouds (Fig. 3.9, b). Moreover, they are also nicely dispersed throughout each of the bananas.



(a) Untrained weights (b) Trained weights

Figure 3.9: Prototype Vectors before and after training

The specified topology consists of 5×5 neurons (Fig. 3.10, left). The winning units (Fig. 3.10, right) seem to be quite evenly dispersed among the neurons of the trained network (i.e. the prototype vector). The underlying density of the data is indeed well contained within the the banana shapes. The topology succesfully reflects this data dispersion by means of the prototype vectors.

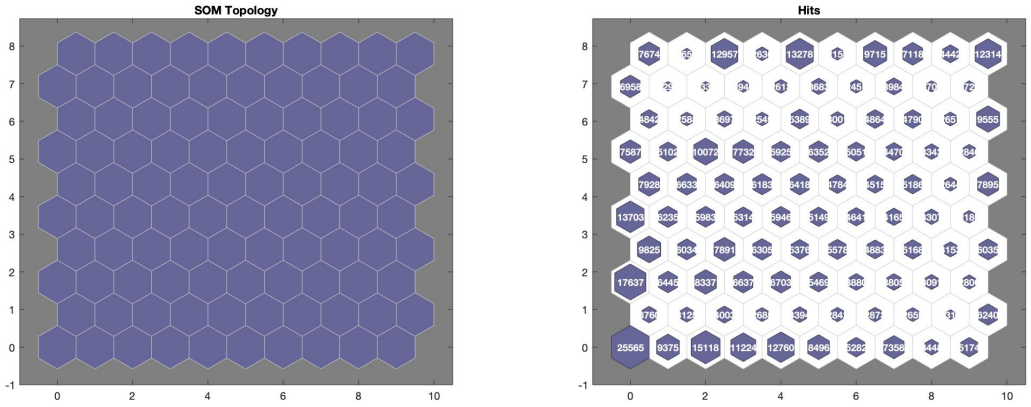


(a) hexagonal topology: 5×5 neurons (b) Winning units

Figure 3.10: Self-organizing map: banana data

Covertype Data

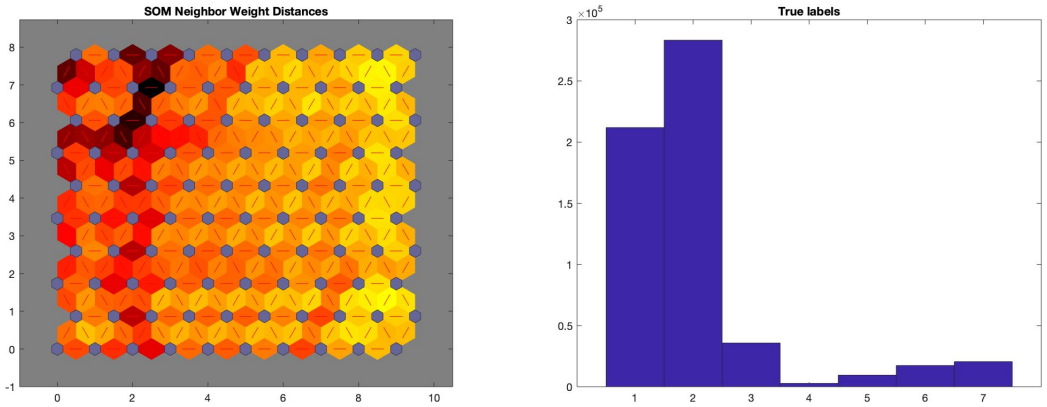
Lastly, self-organizing maps (SOM) are applied to the covertype data. Once more, a hexagonal topology (10×10 neurons) is used as it results in the most satisfactory visualization, as opposed to the grid or random topology. Furthermore, the link distance function is used since Euclidean distance becomes troublesome in higher dimensions. Moreover, it tries to find the distances between the layer's neurons. This model specification results in the SOM displayed below (Fig 3.11, a), with corresponding sample hits (Fig. 3.11, b) after training for 50 epochs.



(a) Hexagonal topology (b) Winning units

Figure 3.11: Self-Organizing Map: Covertype Data

According to the neighbor distances, clustering can be performed in the sense that it reveals how far apart prototype vectors are in term of their link distance. As such, a similar color palette resembles dense regions characterized by closely located prototype vectors, thus yielding clusters of sorts. Note that the labels of the data contain 7 unique forest types, i.e. classes. It seems there are about 3 to 4 distinct clusters based on the color variability (Fig. 3.12, a). As the label distribution is highly unbalanced (Fig. 3.12, b), it makes sense that identifying the 7 clusters in this manner is not feasible.



(a) (b)

Figure 3.12: Neighbor Distances and label distribution

The performance of the trained SOM can be evaluated by a measure known as the Adjusted Rand Index (ARI). It compares the agreements between two partitions and can thus be used to compare the labels of the data points to the cluster assignments based on the model predictions. Unfortunately, the provided function `ARI.m` repeatedly resulted in an undefined function, no matter the attempts I made to resolve this issue. As such, this analysis will remain shy of said index.

Chapter 4

Autoencoders and Convolutional Neural Networks

4.1 Stacked Autoencoders

In this first part of the last assignment, it is of interest to compare the performance of a stacked autoencoder to that of a normal multilayer network w.r.t. the classification of digits. To that end, parameters such as epochs, hidden layers and their respective amount of hidden units are tuned for both types of networks. To address the problems arising in deep networks (many local minima, diffusion of gradient), greedy layer-wise training has been used rather than training the full network. This, along with the fact that an autoencoder attempts to reconstruct the input pattern, is what distinguishes it from the regular MLP.

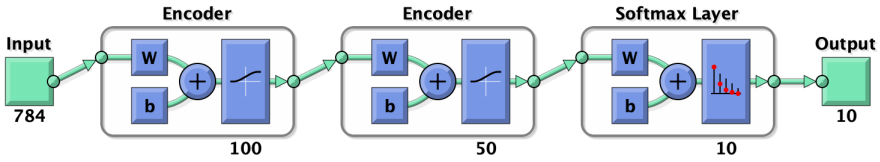
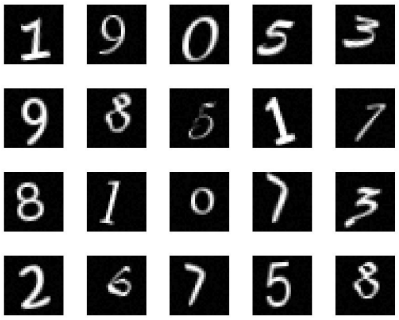
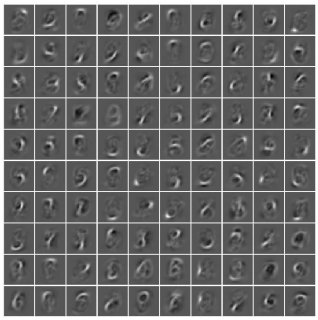


Figure 4.1: two layer Stacked Autoencoder

Using the default hyperparameters from the provided program (Fig. 4.1), the stacked autoencoder already attains a very high classification accuracy. That is, after finetuning based on the labeled data. In the pretraining phase, the overall classification accuracy measured 82.84%, while after finetuning this increased to no less than 99.64%. One can see that the primary feature still bears somewhat of a resemblance to the actual input digits, yet the representation is already more abstract. As such, the autoencoders ability to learn a sparse representation of the input pattern is well demonstrated.



(a) Input



(b) Primary feature

Figure 4.2: MNIST data

Next, a regular nueral network with one hidden layer consisting of 100 neurons was trained. It was able to attain a classification accuracy of 95.84%, which is still respectable. However, an additional hidden layer (Fig. 4.3) only increased the accuracy to 96.6%, which is considerably lower than the stacked autoencoder with 2 hidden layers (Fig. 4.1). A third hidden layer even led to a small decrease, 96.22%. As such, it is clear that the depth of a regular neural network is not quite able to attain the performance of the stacked autoencoder.

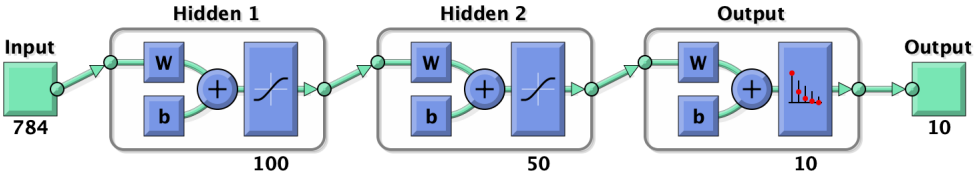


Figure 4.3: Regular two-hidden layer neural network

Next, the objective is to enhance the performance of the autoencoder by tuning the hyperparameters. However, there is little improvement to be made with respect to the accuracy on the test set (99.64%). As such, the hyperparameter tuning mainly serves to examine the behaviour of the autoencoder for varying specifications.

Adding an additional autoencoder layer (Table 4.1, Autoencoder 2) did not lead to performance improvements. That is, all else equal, it resulted in a classification accuracy of 98.80%, which is lower than its two layer variant. Next, the number of hidden units have been doubled with respect to the default specification (Fig. 4.1). All else equal, this resulted in an accuracy of 99.90% (Table 4.1). Lastly, the maximum amount of epochs have been increased by 100 for all layers. All else equal, the obtained accuracy on the test set after finetuning was 99.12% (Table 4.1). As expected, attaining a better performance was quite unlikely in the first place.

	Characteristics	Test Accuracy
Autoencoder 1	2 Autoencoders	99.64%
Autoencoder 2	3 Autoencoders	98.80%
Autoencoder 3	Double hidden units of (1)	99.90%
Autoencoder 4	Double Epochs of (1)	99.12%

Table 4.1: Stacked autoencoders: Test accuracy

Note that for all autoencoder specifications, finetuning led to very significant performance improvements as the pretraining phase accuracy was often unsatisfactory. This is not surprising since the separate training of the autoencoder layers is not optimized with respect to correct classification of the labeled data. Instead, it is learning compressed representation of the input through sparse training in an unsupervised manner. By stacking the autoencoders and considering the labeled data, the network is finetuned by means of supervised learning. Regularization was achieved by introducing an L2 weight penalty and a sparsity penalty which limits the amount of hidden units.

4.2 Variational Autoencoders

In this section, the autoencoder is revisited in the context of latent variables rather than observed variables. In an information bottleneck, the bottleneck vectors can in fact be regarded as latent variables. That is, the features a regular sparse autoencoder derives from learning the identity map are not only compressed representations of the input space, they are also an unobserved variable of sorts (i.e. latent variable). As opposed to mapping the input space to a vector through the encoder mapping, it is now mapping it to a distribution which can be characterized by its mean vector and standard deviation vector. As such, the decoder mapping is slightly less trivial than the identity map in the case of regular autoencoders. It can be represented as the product of on one hand, the distribution of the latent given the observed variables $p(x|z)$ and the distribution of the latent variables $p(z)$ on the other hand. When the latent variables z are integrated out, one ends up with the marginal distribution of the reconstructed observed variables $p(x)$. However, based on the derivations provided in the assignment document, it can be shown to equal the Evidence Lower Bound, yet in terms of the approximate posterior distribution for the latent variables, $q(z|x)$. Hence, the autoencoder is estimating this variational bound (4.1).

$$\begin{aligned} \log p(x) &= \log \int p(x|z)p(z)dz \\ &= \log \int \frac{q(z|x)}{q(z|x)} p(x|z)p(z)dz \\ &= -D_{KL}[q(z|x)|p(z)] + E_{q(z|x)}[\log p(x|z)] \end{aligned} \tag{4.1}$$

Similar to regular autoencoders, this reconstruction is not of immediate interest. The compressed features the autoencoder derives are of particular interest. In the context Bayesian analysis, the class of the posterior distribution depends on the prior distribution $p(z)$, provided it is a conjugate prior. This follows from the conjugacy between prior and posterior. For instance, in the provided demo for the `MNIST` data, a multivariate normal prior is chosen for the latent variables. This is a conjugate prior, resulting in a posterior that is also a multivariate normal. This in turn, makes it possible to sample from that distribution using the re-parametrization trick used in the exercise demo.

Performance metric

The variation lower bound involves the divergence between the approximated posterior and the prior and the expected negative reconstruction error. On one hand, the divergence term acts as a regularizer, ensuring the sparsity of the variational autoencoder. On the other hand, the negative expected reconstruction error is more closely related to the stacked autoencoder. This, in fact, corresponds to the Cross-entropy loss that was used for the stacked autoencoder, be it not in terms of the latent variables.

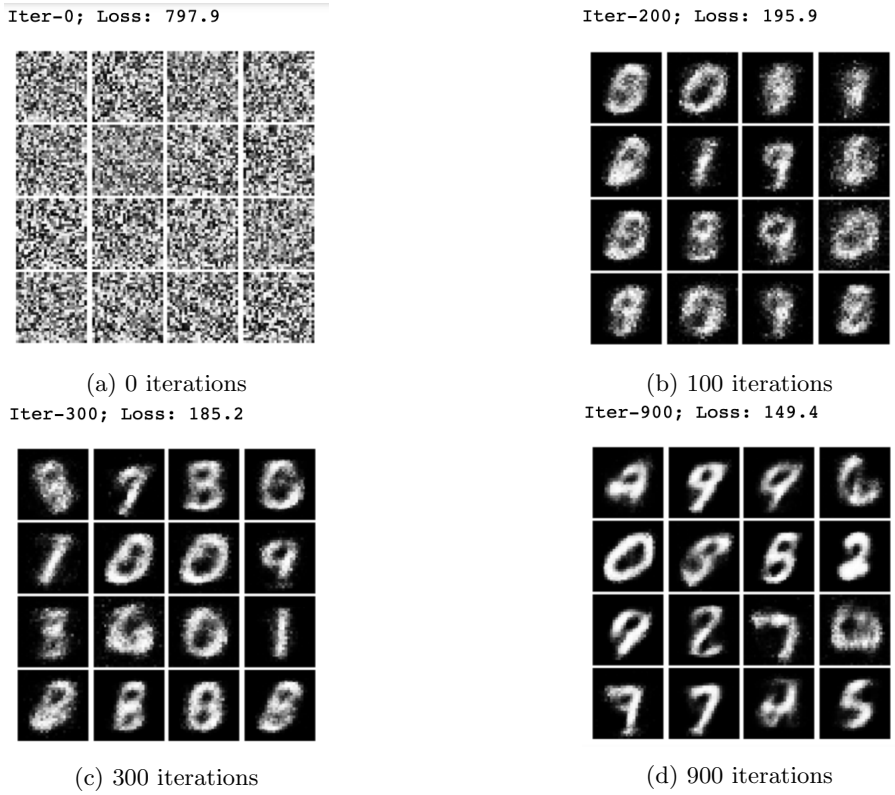
Optimization algorithm

As opposed to the greedy layer-wise training used for the stacked autoencoder, the variational autoencoder demo implements an adaptive learning rate optimization algorithm known as Adam. More specifically, it computes individual learning rates for different parameters. There are several advantages to using Adam, such as superior training time and the ability to work with sparse gradients. The latter is of particular interest since autoencoders suffer from the diffusion of gradient which causes the gradient to rapidly decrease in magnitude as the depth of the network increases, thus resulting in sparse gradients. However, this comes at a cost. Adam has not been shown to converge to an optimal solution. This in turn, results in poor generalization of the algorithm.

Greedy layer-wise training on the other hand, suffers from poor training time since it trains shallow networks separately. In that regard, it is less favorable compared to Adam. In the previous section, scaled conjugate gradient descent was implemented at each training stage. It is known that conjugate gradient algorithms do converge to an optimal solution. Hence, the trade-off essentially boils down to speed versus generalization capacity.

MNIST Data

When applied to the data, the variational autoencoder essentially starts from random noise (Fig. 4.3, a). After each iteration, the representation becomes better and better, i.e. lower loss value (Fig. 4.3; b, c). Ultimately, it is able to attain a good representation of the input space at 900 iterations (Fig. 4.3, d)



(a) 0 iterations

(b) 100 iterations

(c) 300 iterations

(d) 900 iterations

Figure 4.4: Variational Autoencoder: MNIST data (1)

4.3 Convolutional Neural Networks

This last section explores convolutional neural networks (CNNs) as a tool for feature extraction in order to train an image category classifier. To that end, a pre-trained CNN ("AlexNet") is applied to the `Caltech101` image data. The classifier takes the form of a linear support vector machine (SVM), applied on the features as extracted by the CNN. Although this data set has a large variety of image categories, a small subsample of classes is considered, namely airplanes, ferries and laptops (Fig. 4.5). Note that due to class imbalance, the class count has been adjusted to a balanced class distribution.

Figure 4.5: Caption

The architecture of AlexNet consist of 23 layers (including input/output), among which there are 5 convolutional layers and 3 fully connected layers (Fig. 4.6). Furthermore, it also includes

normalization (2), max pooling (5) and ReLu (7) layers. Finally, it passes through a softmax and classification layer.

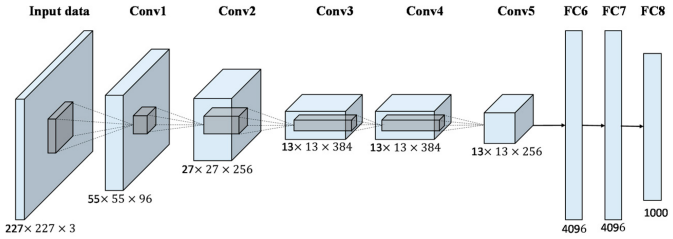


Figure 4.6: AlexNet

When looking at the first convolutional layer weights, it is known that the low-level features, i.e. weights, generally represent edges, even though this is hard to detect by the human eye (Fig. 4.7). Note that this corresponds to quite a large amount of weights, namely $11 \times 11 \times 3 \times 96 = 34848$ weights. The amount of convolutions performed on the natural image corresponds to the value 96. Note that when such convolutions are performed, it is often implemented using matrix multiplication via the Toeplitz matrix. Simply inverting this matrix to convert back to the natural image does not work since the inverse of a Toeplitz is no longer Toeplitz. The inverse has to be obtained otherwise.

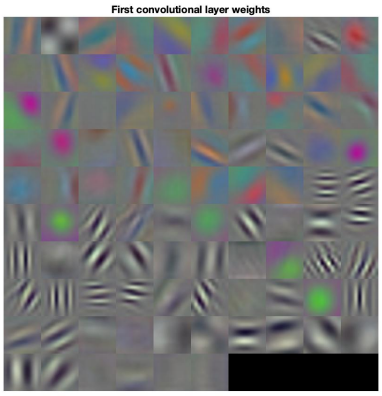


Figure 4.7: Convolutional layer weights

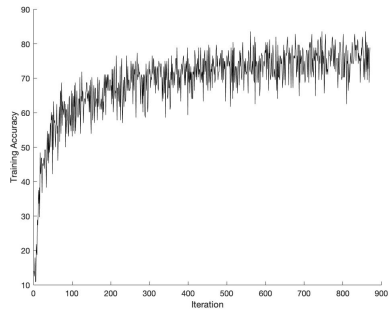
Layers 1 to 5 consecutively consist of an input layer, a convolutional layer, a ReLu layer, a normalization layer and a Max pooling layer.

- The input is of very high dimensionality. That is, $227 \times 227 \times 3 = 154587$, or rather width of 227, length of 227 and 3 the color channel RGB.
- After convolving the input, the dimensionality is substantially reduced to $11 \times 11 \times 3 \times 96 = 34848$.
- The ReLu layer outputs the summed weights if positive, and 0 otherwise.
- Since the transformation results in an unbounded output, normalization is applied in layer 4 across the color channels. Note that the ReLu layer and the normalization layer do not affect the dimension of the preceding layers.
- Layer 5 performs a Max pooling operation on the 11×11 feature with a stride of 2×2, hence resulting in a 5×5. The number of pixels in each direction is more than halved, resulting in a size of less than a quarter of the convolved features. More specifically, the dimension of the problem is now only $5 \times 5 \times 3 \times 96 = 7200$.

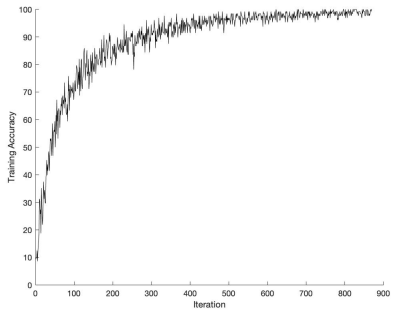
Finally, the output layer consists of 1000 neurons, each representing one of the 1000 classes AlexNet is able to distinguish. A dimensionality of 1000 is very small compared to the input dimension of 154587.

MNIST data

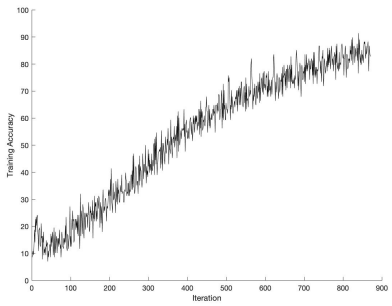
Once more the `MNIST` data set is revisited in the context of CNN's. The networks provided in the `CNNdigits.m` script respectively consist of 7 and 9 layers for the first and second specification. In terms of training, the 7 layer CNN was significantly faster in training and it also reaches a satisfactory accuracy much quicker (Fig. 4.9, b) as opposed to its 9 layer counterpart (Fig. 4.9, c). During testing, the 7 layer network was able to attain a test accuracy of 93.52%, whereas the 9 layer network only reached 82.32% (Table 4.1). Although a deeper network, its performance is inferior possibly because an image size 28×28 might not need many convolutional layers. In fact, it would even be feasible to work with the images in their original representation for this specific example. To that end, a fully connected network has been trained (Fig. 4.9, a), yet it did not perform that well (Table 4.1).



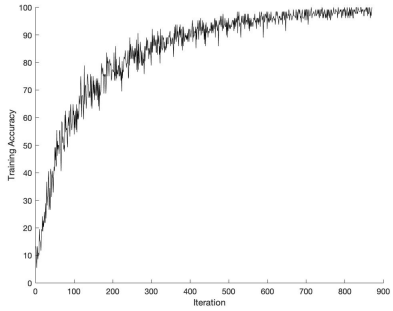
(a) Fully connected, no convolutions



(b) 7 layers



(c) 9 layers



(d) AlexNet replica

Figure 4.8: CNN training

Next, an alternative CNN specification is considered to see if any performance improvements can be obtained. To that end, I drew inspiration from the AlexNet architecture outlined above. However, this replica does not have cross channel normalization, and it only consists of 4 convolutional layers and only 2 fully connected layers. During training it was able to reach an accuracy of 99.22% (Fig. 4.9, d). Its test accuracy came out at 96.79% (Table 4.1). Note that these performance improvements led to a substantial increase in training time.

	Test Accuracy
Fully Connected, No Convolutions	69.08%
7 layer CNN	93.52%
9 layer CNN	82.32%
AlexNet inspired CNN	96.79%

Table 4.2: CNN test accuracy