KU LEUVEN

Report Artificial Neural Networks

name: Simon De Lamonica

student number: r0380453

study program: credit contract

1. Exercise: Supervised learning and generalisation

This part of the report tries to examine the difference between different training algorithms that are used for training a feedforward neural network. The feedforward neural networks that are being used here use one input layer, a specified number of hidden neurons and one output layer. The training algorithms that are being used are: gradient descent ("traingd"), gradient descent with adaptive learning rate ('traingda"), Fletcher-Reeves conjugate gradient descent ("traincgf"), Polak-Ribiere conjugate gradient algorithm ("traincgp"), BFGS quasi Newton algorithm ("trainbfg"), Levenberg-Marquardt algorithm ("trainlm") and Bayesian regularization backpropagation ("trainbr"). The last training algorithm, Bayesian regularization backpropagation uses a regularization term in order to prevent overfitting by including the amounts of the weights in the cost function and thus favoring small values for these weight vectors. The hyperparameters of these training algorithms will be set to their default values in MATLAB.

In this exercise, a feedforward neural network will be used to estimate a 2-period sinus wave. The total data set will consist of 100 data points. These points will be evenly drawn from the 2-period sinus wave, meaning that the x-value of the ith point will be equal to i × (1/100) × (4π) with i being a discrete number from 1 till 100. The corresponding y-values are then equal to the sinus of those x-values.

In order to examine the speed of the different training algorithms the tic-toc functions in MATLAB are used. By inserting the tic function before the training and saving the value of the toc function that is placed after the training, it is possible to save the time that is necessary to train the feedforward neural networks until convergence. Model performance of the neural network can be measured by calculating the error in the form of the RMSE (root-mean-square-error).

A larger number of hidden neurons increases the complexity of the neural network, but increases also the chance of overfitting. Because of the random initialization of the network's weights, it also becomes important to execute many different iterations.
In order to correctly estimate the performance of the network, each iteration will create a different training and test sample before the training. Which of the 100 data points will be added to the training set, will be determined completely at random. A 80-20 rule is used, in order obtain that 80 data points are assigned to the training set and remaining 20 to the test set. After each iteration, a training RMSE and test RMSE are calculated. The speed, training RMSE and test RMSE can then be averaged across all iterations in order to obtain a stable metric. Figure 1 represents bar plots of the averaged speed, training RMSE and test RMSE for feedforward neural networks with a certain training algorithm and a certain number of neurons in the hidden layer. The number of neurons in the hidden layer can be equal to 5, 10, 20, 40 and 100. The amount of iterations that was used for each type of feedforward neural network is equal to 40. No random noise was added here.

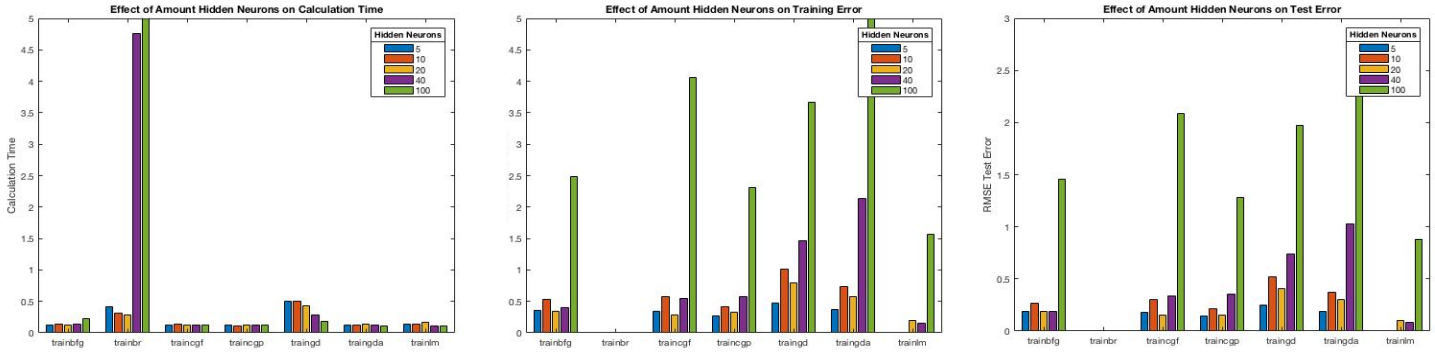

Figure 1: Impact of the number of hidden neurons.

Figure 2 shows bar plots of the averaged speed, training RMSE, test RMSE for feedforward neural networks with different types of training algorithms when noise is added to the y-values of the data. During each iteration with a different training and test set a random value from the standard Gaussian distribution, multiplied with a constant 'noise step' will be added as 'noise' to the y-values of the training set. The higher this 'noise step', the higher the amount of random noise that is added. This noise step will be able to take values 0.1, 0.2, 0.3, 0.4 and 0.5. Overfitting is the problem when the network starts to model this random noise, which is not part of the actual sinus function. All feedforward neural networks in Figure 2 used a fixed amount of 10 neurons in the hidden layer.

Figure 2: Impact of noise.

The training RMSE error and test RMSE error for the Bayesian regularization algorithm ("trainbr") seem remain very low in Figure 1, even if the number of hidden neurons increases. As expected, the regularization term in this algorithm makes it possible for the values of the parameter weights to remain low, which in turn makes the feedforward neural network fit the data better, even if the neural networks is over parametrized. The barplots of the training RMSE error and test RMSE error for the Bayesian regularization algorithm ("trainbr") are missing in Figure 1. This is because the training and test error values are lower than 0,001 which is much lower that the errors for the other training algorithms. The training and test errors of the other training algorithms seem increase when the number of hidden neuron is becomes higher. This means that performance of the training algorithms, except for the Bayesian regularization algorithm ("trainbr"), decreases whenever the number of hidden neurons becomes too high and thus the model becomes too complex. Between those types of algorithms, excluding the Bayesian regularization algorithm ("trainbr"), Levenberg-Marquardt algorithm ("trainlm") seems to have significant lower training and test errors for all number of hidden neurons in comparison to the other training algorithms. The barplots of the training and test errors for the Levenberg-Marquardt algorithm with number of hidden neurons 5 and 10 are not shown, as these values are lower than 0,001.
While it is clear that the Bayesian regularization algorithm leads to the best result, it does have a significantly higher calculation time, when looking on the left plot of Figure 1. When a higher number of hidden neurons is being used, the Bayesian regularization algorithm seems to be a lot slower at estimating the sinus wave.

The gradient descent ("traingd") algorithm seems to have a higher average calculation time than the other training algorithms (also see graphs on the left of Figure 1 and 2). Only the Bayesian regularization algorithm seems to becoming a lot slower when the number of hidden neurons becomes very large (e.g. 40-100). Since the gradient descent algorithm only incorporates first-order information (gradient) in the updating of the parameter weights, the convergence can take longer than the other training methods.

When looking at Figure 2 it is not clear whether the calculation time increases for the different training algorithms when the number of hidden neurons increases. As expected, adding noise seems to negatively impact the training error when looking at the middle graph in Figure 2. When increasing the amount of noise all training algorithms perform worse and model less well the actual relationship. The Bayesian regularization algorithm handles noise much better, as the training errors for the different degrees of noise seem to be much lower in comparison to the other training algorithms. The test error is an indication how well the neural networks performs on new data and thus generalization. It is surprising that the performance of all training algorithms is very comparable when it comes to predicting new data, as the test errors of the different training algorithms seem very similar for all degrees of noise.

The Bayesian regularization ("trainbr") algorithm seems to handle overfitting, due to a large amount of parameters, and noise much better, at a cost of a higher calculation time. Incorporating regularization seems to be good at handling overfitting, at a cost of longer calculation. The gradient descent algorithm often performs worse than other training algorithms and seems to have a relatively high calculation time. The other algorithms perform similarly and are often much faster than gradient descent ("traingd") and Bayesian regularization ("trainbr").

2. Exercise: Recurrent neural networks

2.1. Hopfield net

A Hopfield network consisting of 240 neurons will be used to perform digit recognition. Since every digit can presented by a total number of 240 elements, the same number of neurons needs to be used in the Hopfield network. Initially the correct attractor vectors, each one corresponding to one of the digits 0-9, are assigned to the network. Each attractor vector corresponds to a minimum in the energy function. When a new input, of which the correct digit needs to be identified, is given to the Hopfield net, it may converge to the right attractor vector. However, a spurious attractor vector may exist as well. These are attractor vectors that can also be identified as

minima in the energy function, but do not correspond with one of the 'correct' digit attractor vectors that were initially added. When a new input converges to one of these spurious states then the right digit cannot be identified.

To test the impact of noise of the Hopfield net, random noise originating from the standard Gaussian distribution, multiplied with a constant noise step, was added to the orginal attractor vectors in order to corrupt the original digit images. For each type of digit (0-9), 100 new corrupted input vectors are used to test the classification of the network. These 100 corrupted images for each type of digit are then given as input for the Hopfield net. The impact of noise on the classification results is shown in Figure 3.

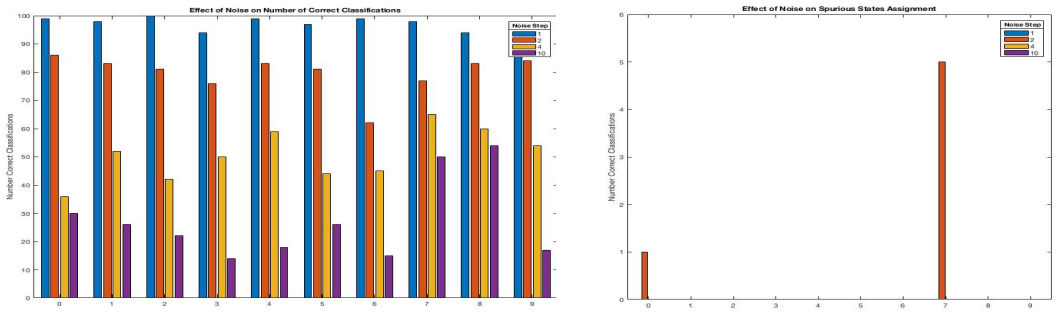

Figure 3: Impact of noise. The amount of time steps was kept constant at 100.

The barplots on the left of Figure 3 show the amount of correct classifications for each type of digit on the 100 corrupted images of that type that were given as input to the network model. Figure 3 shows that a higher level of noise decreases the performance and the amount of correct classifications for each digit type. High levels of noise seem to have a lower impact on the performance for digit types 7 and 8. The barplots on the right on Figure 3 show when the corrupted images of a certain digit type where classified in the spurious states that exist in the Hopfield net. For a noise step of 2, a corrupted image from the original digit type 0 was classified in a spurious vector state and five corrupted images from the original digit type 7 were classified in a spurious state. When these spurious states are examined, two different types were found. The decoding of these spurious states can be found in Figure 4. The image of the left shows the spurious state to which the corrupted digit image of original type 0 was classified to. The image on the right shows the spurious state for the 5 corrupted images of digit type 7. The image on the right shows an image comparable to a digit "7". This is probably an indication why corrupted images of digit type 7 are sometimes misclassified as the spurious state on the right of Figure 4.

Figure 4: Spurious states.

Figure 5 shows the classifications of corrupted images for different number of time steps. Increasing the number of time steps seems to increase the number of correct classifications for each type of corrupted digit. A remarkable exception is corrupted images of digit 8. The graph on the right shows how many corrupted digit images of certain type converged to a spurious state or did not converge at all. It shows that a low number of time steps increases the chance of not converging or that digits of a certain type are wrongfully assigned to a spurious digit state.

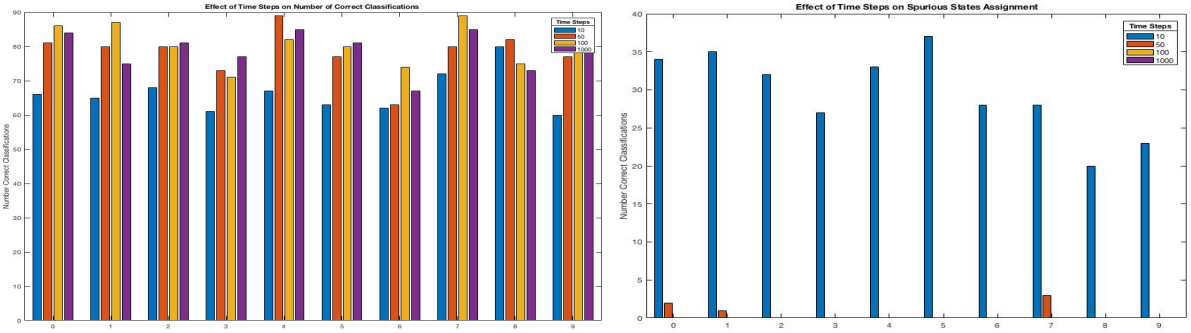

Figure 5: Impact of the number of time steps. The noise level was kept constant at 2.

2.2. Elman network to model Hammerstein time series

The performance of an Elman network in modelling the Hammerstein system data is tested here. The effect of a different number of neurons in the hidden layer, number of total samples used, epochs, the percentage division between training-validation set and test set and the percentage division between the training and validation set are all tested here. The effect of these different parameter values is tested on the speed of the training (measured with the tic-toc functions) and on the R squared. Where the latter reflects the performance of the model. Because the

initialization of the weight vectors of the Elman network and the creation of the test and training set happens at random, a total of 40 iterations will be executed for each configuration, for which the average R squared and calculation time can be calculated. This way coincidence is excluded as much as possible.

Figure 6 shows that one neuron leads to the best performance in estimating the time series. The calculation time increases when a higher number of neurons is used. There seems to be a phenomenon of diminishing returns, whereby increasing the number of hidden neurons to a large number (e.g. 20 and 40) does not seem to significantly improve performance but does increases calculation time. The R-squared does not seem to significantly increase when the number of hidden neurons is systematically raised. For an Elman network with one neuron in the hidden layer the best performance is observed (R^2 = 0,3245). It is however notable that the R-squared values remain low for all executions. When a network architecture with a number of hidden neurons equal to 5 or higher is used, the model performance does not seem to significantly change.

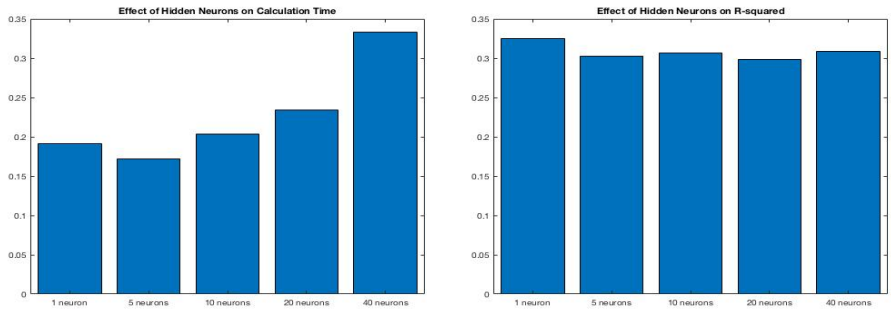

Figure 6: Impact of the number of hidden neurons on the calculation time and R-squared. For each execution the other parameter values were kept constant. The total number of samples used here is equal to 2000, the number of epochs to 1000, the percentage division between the training-validation set and test set equal to 80%-20%. The training-validation set is afterwards split in 70% training set and 30% validation set.

Figure 7 shows the impact of the number of samples on the calculation time on the left and on the performance (R-squared) on the right. As expected, an increasing number of samples improves the R-squared and thus the prediction performance of the model. It is however important to note that a trade off exists between the number of samples used and the calculation time. Increasing the number of samples increases the performance of the model at the cost of an increasing calculation time.

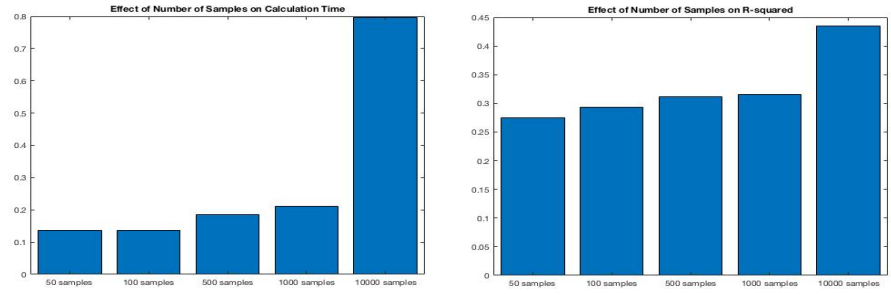

Figure 7: Impact of the number samples on the calculation time and R-squared. For each execution the other parameter values were kept constant. The number of hidden neurons here is equal to 5, the number of epochs equal to 1000, the percentage division between the training-validation set and test set equal to 80%-20%. The training-validation set is afterwards split in 70% training set and 30% validation set.

Figure 8 shows the impact of increasing the number of maximum epochs on the calculation time and performance. Increasing the number of epochs does not seem to have a significant impact on the performance. It is not clear whether increasing the maximum number of epochs above 10 could improve the performance.

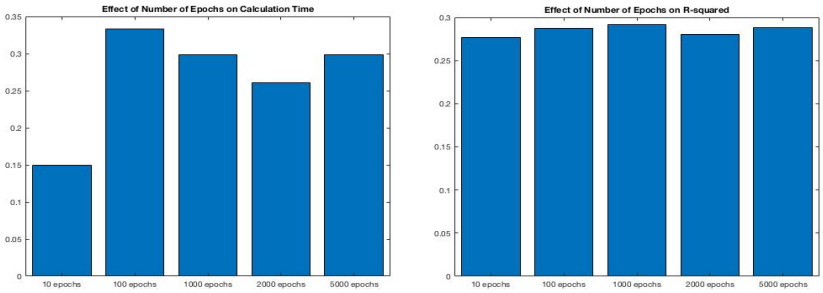

Figure 8: Impact of the number of epochs on the calculation time and R-squared. For each execution the other parameter values were kept constant. The number of hidden neurons here is equal to 5, the number of samples equal to 2000, the percentage division between the training-validation set and test set equal to 80%-20%. The training-validation set is afterwards split in 70% training set and 30% validation set.

The impact of increasing the percentage of samples that is assigned to the training and validation seems to be less clear. Increasing the percentage of samples in the training and validation set seems to have the biggest impact when the initial number of samples used is low e.g. 50, 100. Further investigation showed that changing the percentage split between the training, validation and or the test set does not seem to have a significant impact on the performance of the model.

Within the ranges of the testing, the number of hidden neurons and the number of samples seem to be the most important parameters for improving the network performance. Since one hidden neuron and a high number of 10000 samples seem to perform best, when looking at Figure 6 and 7, a graphical representation of such an execution can now be made. Figure 9 represents the output of the Elman network on predicting the Hammerstein time series with one hidden neuron, 10000 samples as input, the maximum number of epochs equal to 1000, the percentage division between training-validation set equal to 80%-20% and the training–validation split in 70% training and 30% validation set.

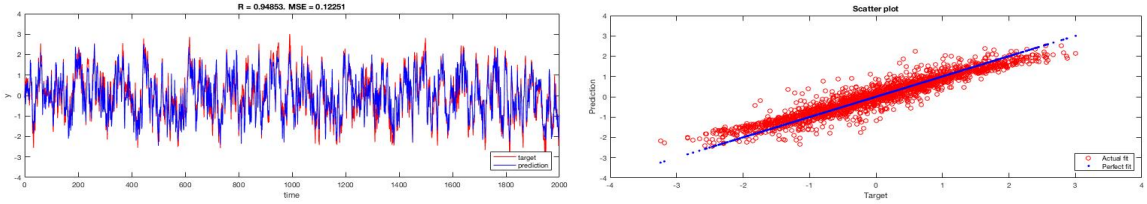

Figure 9: Output with optimal parameter values.

Figure 9 shows that one hidden neuron and a high number of samples leads to a network model with a high performance (R^2 = 0,95). The graphs on the right and left of Figure 6 show that the final network model seems to fit the Hammerstein time series very well.

3. Exercise: Unsupervised learning: PCA and SOM

3.1. PCA on handwritten digits

Principal Components Analysis is a dimension reduction technique that lower the dimensions in the dataset, while keeping the most important information. An eigenvalue decomposition of the covariance matrix makes it possible to calculate eigenvectors and rank them according to their eigenvalues. PCA projects the data on a smaller space with lower dimensions using q eigenvectors with the highest eigenvalues. These q new dimensions are called the principal components. The eigenvalues indicate how much variance can be explained in the original dataset by transforming original data points using the corresponding principal components as new dimensions. When the number of principal components used is equal to the number of original dimensions then all variance in the data set is used and no dimensions reduction happens. Summing up the eigenvalues of the chosen number q of principal components and dividing them by the total number of eigenvalues of all possible principal components, gives the proportion of the amount of variance that can be explained by keeping q principal components. Figure 10 shows a plot of the proportion of variance explained and the reconstruction error in function of the number of principal components for a dataset consisting of handwritten threes with dimensions 16 × 16 = 256. The reconstruction error is equal to the root mean square (RMSE) difference between the original and the reconstructed data set. The reconstructed dataset originates from projecting the original dataset onto q principal components and then reconstruct back the higher dimensional dataset using this projection.

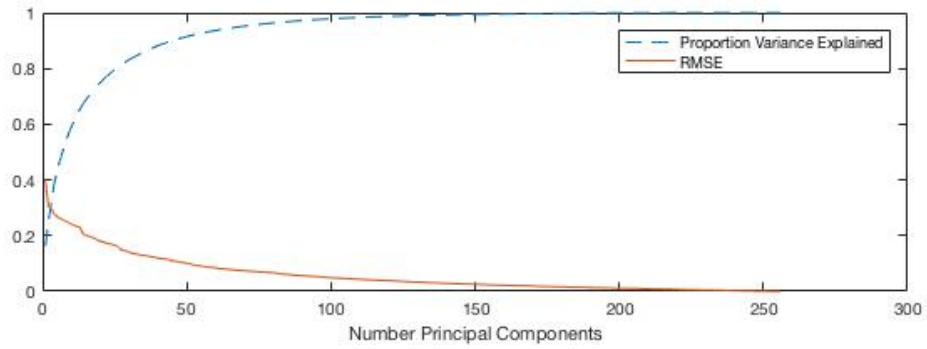

Figure 10: Proportion of variance explained and RMSE reconstruction error in function of the number of principal components.

As expected, the proportion of variance explained increases sharply for low numbers of principal components. This is due to the fact that the first number of principal components that are being added, have significant higher eigenvalues. This means that the first number of principal components explain a large portion of the variance in the original dataset, while higher number principal components do not seem to contribute significant, additional information. Increasing the number of principal components from 1 to 50 increases the proportion of variance explained by 0.7530 (75%), while going from 50 to 100 principal components increases the proportion only with a marginal 0.0641 (6,4%). As expected, the same trend can be found with the RMSE reconstruction error. The low numbers of principal components seem to reduce the RMSE reconstruction error by a significant amount, while higher principal components no longer seem to reduce the RMSE significantly.

Below on Figure 11 a reconstructed image is given for the first observation of the dataset of handwritten trees when the dimensions of original dataset were reduced to respectively 1, 50, 100 and 256 principal components. It is clear that a number of 50 principal components is an adequate number of reduced dimensions, in order to be able to visually represent the image of the handwritten “3” of the first observation very well.

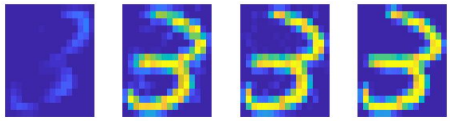

Figure 11: Reconstructed image of the first observed handwritten “3” for a certain number of principal components. Going from left to right: 1, 50, 100 and 256 principal components.

3.2. SOM on Iris datasets

SOM (Self-Organizing Map) like PCA, can be seen as a dimension reduction technique. Data with a high number of dimensions can be projected on a certain number of neurons. These neurons can be organized in a certain way (hexagonal grid, square grid etc.) in a space of dimensions. This dimension is often set equal to 2, in order to be able to make a graphical representation. The training updates the weights on the neurons such that their location changes in order to fit the training data well.

Here SOM’s are used to cluster the Iris sample set. By projecting the 4-dimensional iris data onto a set of three neurons, clusters can be identified. The clustering performed by SOM can then be compared to the real Iris flowers species with the Rand index. The Rand index takes on values between 0 and 1, where an index closer to one means that two clustering results resemble each other better. A higher Rand index value between the clustering result of the Iris data obtained by SOM and the original species classification indicates a better result. Because the number of species in the iris dataset is equal to 3, the same number of neurons will be included in the grid of SOM.

The graphs in Figure 12 show the impact of changing the topology of how the neurons are organized and distance functions on the clustering result. The grid topologies that are tested here are **gridtop** (square grid), **hextop** (hexagonal grid) and **randtop** (random grid topology). The different distance functions include **linkdist** (link distance function), **dist** (Euclidean distance weight function), **mandist** (Manhattan distance weight function). The maximum number of epochs was set constant to 1000. A higher Rand index indicates a better clustering result for the SOM.

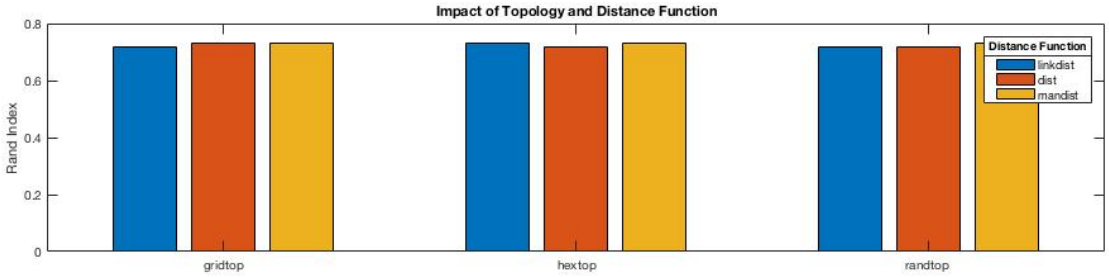

Figure 12: Impact of different topologies and distance functions on the Rand index.

Figure 12 shows that changing the neuron grid topology or the distance function does not seem to have a clear impact on the Rand index and thus the clustering performance. There seems to be a small preference for the Manhattan distance weight function.

Figure 13 shows the impact of the number of epochs on the clustering result. A SOM with a default topology of **hextop** (hexagonal grid), **mandist** (Manhattan distance weight function) distance were used here. It is clear that there is a relationship of diminishing returns between the number of epochs and the value of the Rand index. The greatest improvement occurs when going from 50 epochs (Rand = 0,5529) to 100 epochs (Rand = 0,7163). When the number of epochs is raised above 200, increasing the number of epochs no longer seems to significant improve the Rand index.

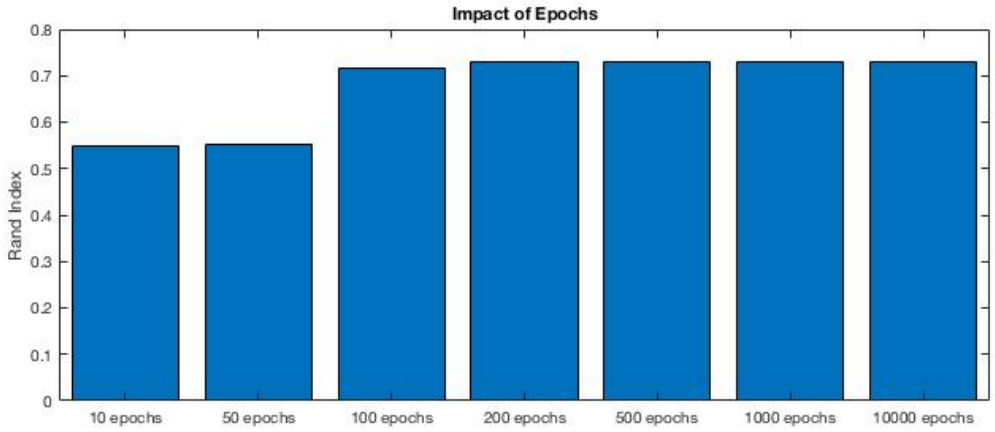

Figure 13: Impact of the number of epochs on the Rand index.

A fairly good solution for clustering the Iris dataset with SOM seems be a SOM with a **hexatop** (hexagonal grid) topology, **mandist** (Manhattan distance weight function) and 200 training epochs. Over a total of 20 different iterations such a SOM clustering results in average Rand index of 0.7261.

4. Exercise: Deep Learning: Stacked Autoencoders and Convolutional Neural Networks

4.1. Digit classification with Stacked Autoencoders

In this exercise, Stacked Autoencoders are used to classify handwritten digits. A stacked autoencoder is a multilayer neural network that consists of sparse autoencoders. The outputs of each layer are connected as input to the successive layer. Greedy layer-wise learning makes it possible to train each layer individually using an autoencoder, while keeping the parameters of the other layers constant. Fine tuning with backpropagation on all layers at the same time is also an option to improve the results. The use of sparse autoencoders in multiple layers makes it possible to reduce the dimensions (comparable to PCA reduction). The feature output from the last autoencoder can be used as input for the training of a supervised softmax layer in order to classify the digit images.

Important parameters of a stacked autoencoders network model for classification of images are the number of hidden neurons in each layer, the number of hidden layers (each one can be represented by an autoencoder) and

the maximum number of training epochs. Varying these can result in different training speeds and accuracy (number of correct classifications). Figure 14 shows the speed and accuracy for a different number of hidden neurons in the first hidden layer for 4 different network architecture models[1]: a stacked autoencoders network without the finetuning, with the fine tuning, a standard network model with one hidden layer created with the default parameters of **patternnet** network in matlab and a standard patternnet with two hidden layers. Figure 15 shows the effect of changing the number of neurons in the second hidden layer. Figure 16 shows the effect of changing the number of maximum epochs. The speed and accuracy for each configuration were averaged over a total of 5 iterations (this is less iterations then in the previous exercises due to longer calculation times). When a certain parameter was tested, the values of the other parameters were kept constant at their default values (100 neurons first hidden layer, 50 neurons second hidden layers and maximum 200 epochs).

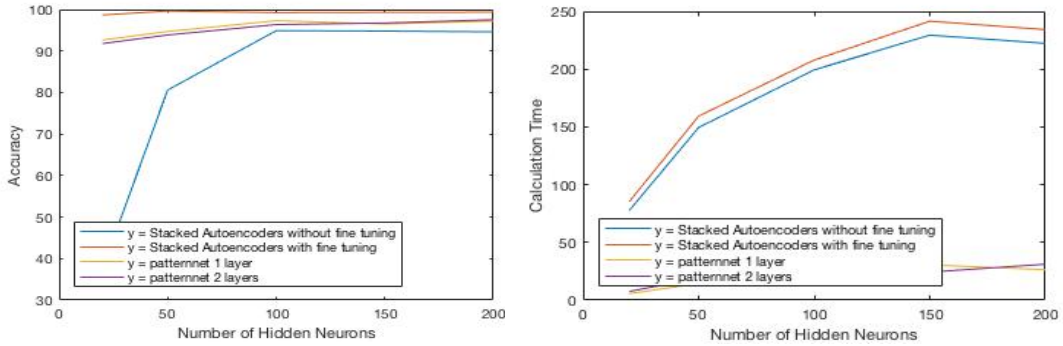

Figure 14: Impact of number of hidden neurons of the first hidden layer on accuracy and calculation time. For each configuration, the number of neurons in the second hidden layer was set constant to 50 and the maximum number of epochs equal to 200.

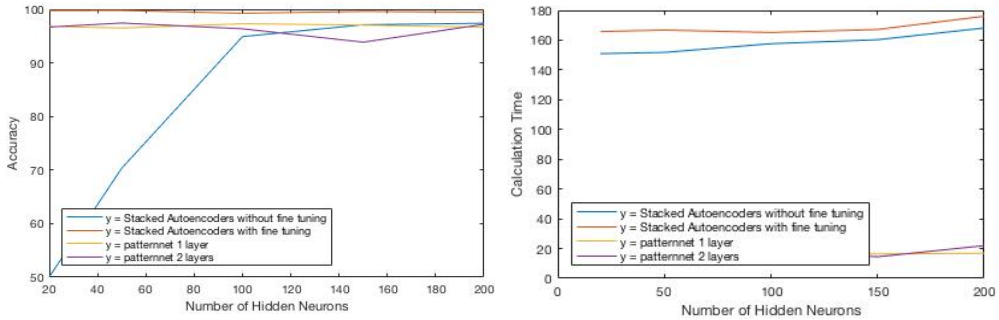

Figure 15: Impact of number of hidden neurons of the second hidden layer on accuracy and calculation time. For each configuration, the number of neurons in the first hidden layer was set constant to 100 and the maximum number of epochs equal to 200.

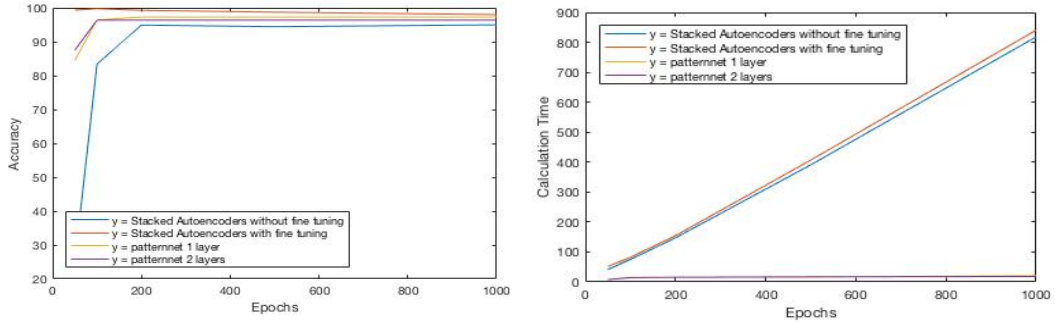

Figure 16: Impact of the number of epochs on the accuracy and calculation time. For each configuration, the number of neurons in the first hidden layer was set to 100 and in the second hidden layer to 50.

One too low number of hidden neurons seems to greatly degrade the performance of a stacked autoencoder model without fine tuning. A lower number of epochs also seems to degrade the performance of a stacked autoencoder without fine tuning. The accuracy performance of a stacked autoencoder model with fine tuning seems to be always better than any other network model, whatever their architecture. When fine tuning is not applied, the performance seems to be almost always worse than the standard **patternnet** neural networks. It can be concluded that fine

[1] In Matlab the rng('default') command was used in order to obtain the same initialization in such a way that the results of different configurations can be compared.

tuning the stacked autoencoder seems to be very important in order to obtain an optimal performance. A stacked autoencoder network with fine tuning always results in an accuracy prediction above 99%. The prediction accuracy of the stacked autoencoder model seems to be significantly higher, even for a low number of hidden neurons in both layers.

Despite the higher performance, the calculation time for the stacked autoencoder model (with and without finetuning) seems to be always significantly higher than for the other two standard **patternnet** neural networks. These latter are far faster to train when looking at the calculation times in Figure 14, 15 and 16. Results seem to suggest that the calculation time of the stacked autoenocder model is at least three times higher than a **patternnet** neural network with the same configurations. This indicates a tradeoff between performance and calculation time.

When looking at the number of epochs it becomes clear that 100 epochs is optimal for training a stacked autoencoder with fine tuning with 100 neurons in the first hidden layer and 50 in the second hidden layer. This configuration seems to lead to the highest prediction accuracy (99,74%) across all different architectures for the 2-hidden-layer stacked autoencoder with fine tuning. Increasing the number of epochs beyond that scope seems to degrade the accuracy performance due to overfitting. The calculation time seems to increase much more for the training of stacked autoencoders than for the 'standard' patternnet neural networks when the number of epochs increases (see Figure 16).

A stacked autoencoder with more hidden layers was also tested and showed to have comparable results with the one-hidden-layer version when applying finetuning. Because the prediction accuracy doesn't seem to increase significantly, a stacked autoencoder with one hidden layer seem to be adequate for correctly classifying digits.

4.2. Answers to questions in section 2.2 of the exercise

The script **CNNex.m** is run to use a pre-trained CNN that is able to classify images between three classes: laptop, ferry or airplane. Below an example image is given for each.



The images have dimensions 227 × 227 × 3. The three indicates the three different colour dimensions: red, green and blue.

- The second, convolutional layer seems to have weights with dimensions 11 × 11 × 3 × 96. The dimensions of these weights show that that 96 convolution masks (receptive fields) will be formed with dimensions 11 × 11 × 3 (filter).

- The masks of the first convolutional layer will be convolved with each image using a stride (4,4). The stride can be seen a step size along the convolution. The second layer will give an output of 96 images (because there are 96 masks) with the new dimensions being approximately 56 × 56 (since the 227 / 4 ≈ 56, rounded downwards). Layer three is a ReLU and layer four a Cross Channel Normalization meaning that these layers do not affect the dimensions. Layer five is a max pooling layer with dimensions 3 × 3 and stride (2,2). This means that the last max pooling layer will cut the of the 96 images from the convolutional layer again in half to dimensions 28 × 28 (56 / 2 = 28).

- Because the CNN was pre-trained to divide images in to 1000 different classes, the dimension of the final layer consists of 1000 neurons. Since the original pixel dimensions were 227 × 227 × 3 = 154587, using a total number of 1000 neurons in the end is still a significant dimension reduction. In the classification here, only three neurons / classes will actually be used to classify images as either a laptop, ferry or airplane.

5. Final project:

5.1. Problem 1: nonlinear regression with MLP's

In this exercise, a nonlinear function $f(X_1, X_2)$ with inputs X_1 and X_2 will estimated using an MLP. The function $f(X_1, X_2)$ to be estimated with student number r0380453 can be represented as follows:

$$f(X_1, X_2) = (8T1 + 5T2 + 4T3 + 3T4 + 3T5)/(8 + 5 + 4 + 3 + 3) \quad (1)$$

The vectors $T1$, $T2$, $T3$, $T4$ and $T5$ can be found in the Data_Problem1_regression.mat file in order to create the target variables for the function described in (1). The final file contains 13 600 data points in total. In order to find the best MLP architecture a training, validation and test set each containing 1000 different data points will be created. The samples for the training, validation and test set will be drawn completely at random from the original

dataset. The training set will be used for the training the MLP, the validation set to select the correct (architectural) hyper parameters and the test set to test the final model. Figure 17 shows an example of the scattered surface plot of 1000 randomly selected training datapoints.

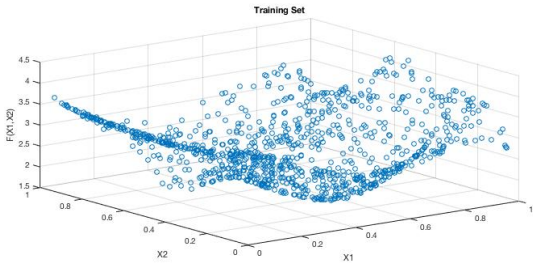

Figure 17: Scattered surface plot of 1000 samples.

Since there are 2 inputs and 1 output to be estimated the number of neurons in the first layer be equal to 2 and in the final layer to 1. The transfer function in the last output layer should be set to the linear transfer function ('**purelin**') in order to obtain the continuous output that the regression requires.

The different parameters of a feedforward neural network that will be tested:

- Training algorithms = ("traingd", "traingda", "traincgf", "traincgp", "trainbfg", "trainlm", "trainbr")
- Transfer functions = ("logsig[2]", "tansig[3]")
- Number of neurons in each hidden layer = (5, 10, 20, 50, 100)
- Number of hidden layers
- Other parameters will be set to default

First a simple feedforward net with one hidden layer will be tested. Figure 18 shows the RMSE of the validation set and training calculation time for different types of feedforward neural networks with one hidden layer and a maximum number of 1000 epochs. Each type of network was iterated 10 ten times in order to obtain the more stable, averaged results for the calculation time and RMSE. The two graphs on the left resulted from using the "logsig" as transfer function for the hidden layer, the two on the right used the "tansig" as transfer function.

log-sigmoid transfer function ("logsig") **hyperbolic tangent sigmoid transfer function ("tansig")**

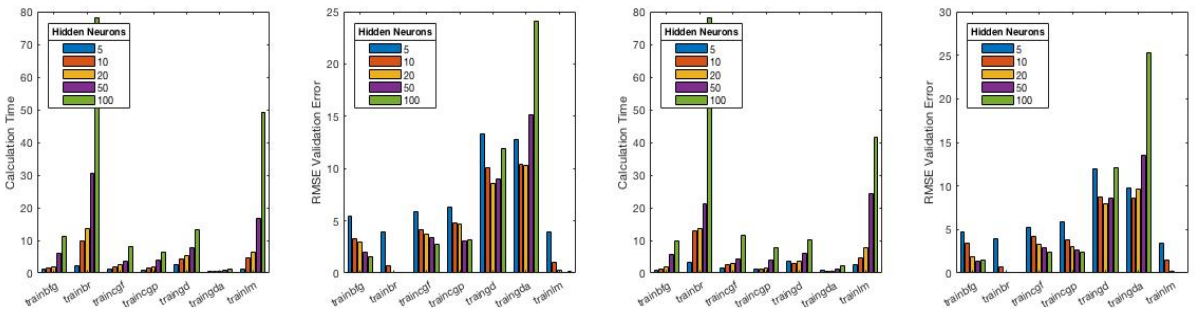

Figure 18: RMSE and calculation times for feedforward neural networks with one hidden layer.

The performance results (RMSE errors) are quite similar whether the network uses the "logsig" or "tansig" as transfer function in the first layer. Since the calculation time is a bit lower on average for the "tansig" function, this one is preferred. Results seem to suggest that the training algorithm Bayesian regularization backpropagation ("trainbr") leads to the best performance for all numbers of hidden neurons. When the RMSE error falls below 0.1, the respective barplot is no longer given in Figure 18. This is the case for the "trainbr" training algorithm when the number of neurons in the hidden layer is equal to 20, 50 and 100. When the "tansig" transfer function with a "trainbr" training algorithm is used, the RMSE error of the validation set is equal to 0,0962, 0,0033 and 0,0008 for respective 20, 50 and 100 neurons in the hidden layer. 50 neurons in the hidden layer seems to an adequate amount since a fairly low validation error (0,0033) can be achieved. Increasing the number to 100 requires a lot of additional computation time when looking at Figure 18 and does not seem to significantly improve the performance.

The best architecture for a one-hidden-layer feedforward neural network seems to be using a Bayesian regularization backpropagation ("trainbr") training algorithm with a hyperbolic tangent sigmoid ("tansig") as transfer function and 50 neurons in the hidden layer. The graph on the left of Figure 19 shows the training and test MSE

[2] "logsig" stands for the log sigmoid transfer function
[3] "tansig" stands for the hyperbolic tangent sigmoid transfer function

error for this network in function of the number of epochs. It is clear that the training and test error show a similar downwards path. In the end the MSE of the training set is equal to 1.2251e-06 and converges almost to zero. The MSE of the test is a little bit higher with a value of 1.6846e-06 and also comes very close to zero. When calculating the R-squared value on the test set, Matlab gives a rounded value of 1. The MSE error and R-squared values indicate that this feedforward neural network is able to model the data almost perfectly. The graph in the middle of Figure 19 indicates the data points of a test set with the real target values from the function $f(X_1, X_2)$. The graph on the right of Figure 19 shows the corresponding predicted values of the test set. The scatter plots are visually almost indistinguishable from each other. Based on the test set, the final network model is able to predict the function near perfection.

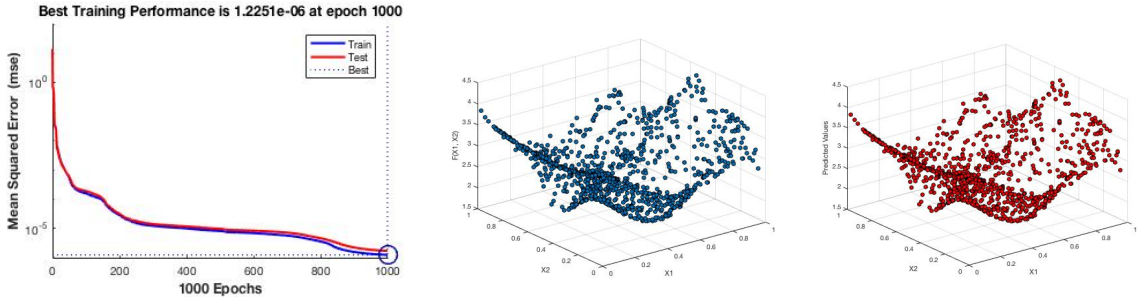

Figure 19: Results of the best feedforward neural net with one hidden layer, 50 neurons in the hidden layer, "tansig" as transfer function in the first layer, Bayesian regularization backpropagation as training algorithm and a maximum number of 1000 epochs.

Since a one-hidden-layer neural network is able to predict the function very well, it is not necessary to examine neural networks with higher number of hidden layers. The architectures of networks with multiple hidden layers are far more complex and often more computationally expensive. It however possible that a network with an even higher performance can be found by using multiple hidden layers. Changing the size ratios between the training-validation-test sets can also be a parameter to improving the model performance.

5.2. Problem 1: classification with MLP's

A feedforward neural network will be used here to classify wines. First the best feedforward neural network architecture needs to be found that distinguishes between two classes of wine: white wines of quality 6 and white wines of quality 7-8 otherwise. White wines with the quality level 6 are assigned the to the first class and white wines with quality 7 and 8 to the second class. The only difference with the feedforward neural network in the previous question is the number of neurons in the input layer, neurons in the output layer and the transfer function. The number of neurons in the input layer will be equal to 11, corresponding to the number of predictor variables (obviously without the target variable 'quality') in the wine quality data. The number of neurons in the output layer will be equal to two, since two wine classes need to be distinguished. The transfer function in the last output layer will be set to the softmax function, which is appropriate for a classification task with a feedforward neural network. A wine will be assigned to class corresponding with the output neuron with highest value. In Matlab this type of 'classification' feedforward net can be instituted using the **patternnet** command. The same network architectures as in 'Problem 1: nonlinear regression with MLP's' will be tested in order to identify the best training algorithm, best transfer function for the hidden layer and the best number of hidden neurons. Results for the different network architectures can be found in Figure 20. These results were found by using a maximum number of 1000 epochs and 10 iterations for each network configuration.

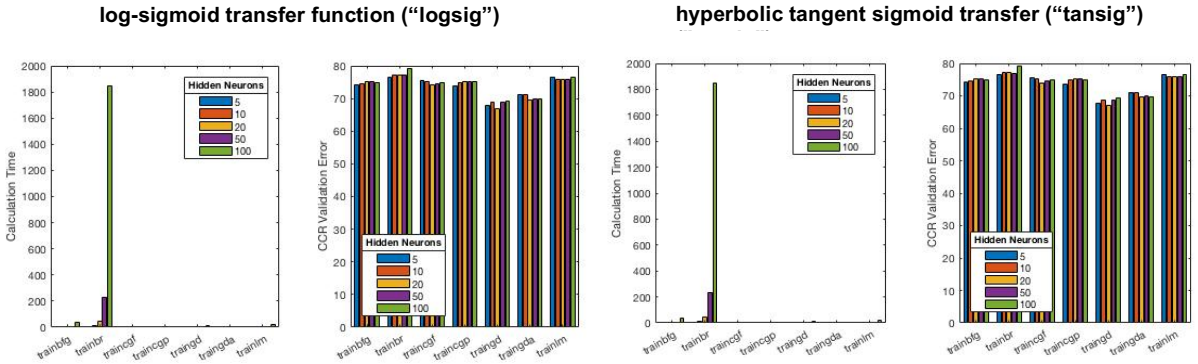

Figure 20: CCR and calculation times for patternnet neural network with one hidden layer, based on all 11 input variables

Figure 20 shows that gradient descent ('traingd') and gradient descent with adaptive learning rate ('traingda') lead to the worst performances. The other training algorithms give similar CCR (correct classification ratio's) values between 70% - 80% for all the different number of hidden neurons and both transfer functions. The best performance is achieved using the Bayesian regularization backpropagation ('trainbr') training algorithm. However, using this training algorithm seems to have a much higher calculation rate than the other training algorithms, especially for a high number of hidden neurons (e.g. 50 and 100). The second-best training algorithm here is Levenberg-Marquardt ('trainlm'), which shows comparable results, but has a much lower calculation time. The Levenberg-Marquadt training algorithm seems to be the preferred choice. Changing the number of hidden neurons above 5 doesn't seem to significantly improve the classification performance, but does increase complexity and calculation time. Since the transfer function in the hidden layer does not seem to affect performance, it is probably best to choose the 'tansig', which is default used as transfer function in the hidden layers of the **patternnet** neural network. When increasing the number of hidden layers, similar performances were found. Increasing the number of hidden layers doesn't seem to significantly increase the performance.

An adequate network architecture seems to be one hidden layer with 5 hidden neurons, the default 'tansig' as transfer function in the hidden layer and Levenberg-Marquadt as training algorithm. Over a total number of 10 iterations an average CCR of a little over 76% was achieved.

PCA can be used to reduce the 11 predictor variables to a lower number of independent dimensions. The graph on Figure 21 shows the proportion of variance that can be explained in the original input data of 11 predictor variables in function of the number of best principal components. Almost all of the variance in the 11-dimensional input data can be explained by using 3 principal components. Using additional components seems to only add dimensionality while not actually explaining any additional variance.

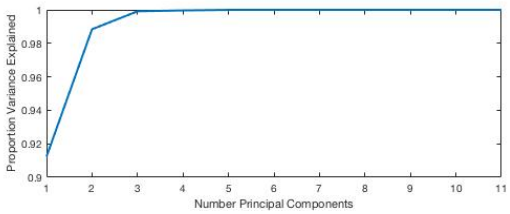

Figure 21: Proportion of variance explained by the number of best principal components.

After creating the dataset for the best three principal components, a similar analysis can be conducted as before to find the best **patternnet** architecture for classifying the wines. Figure 22 gives the CCR ratio's and calculation times for different network architectures.

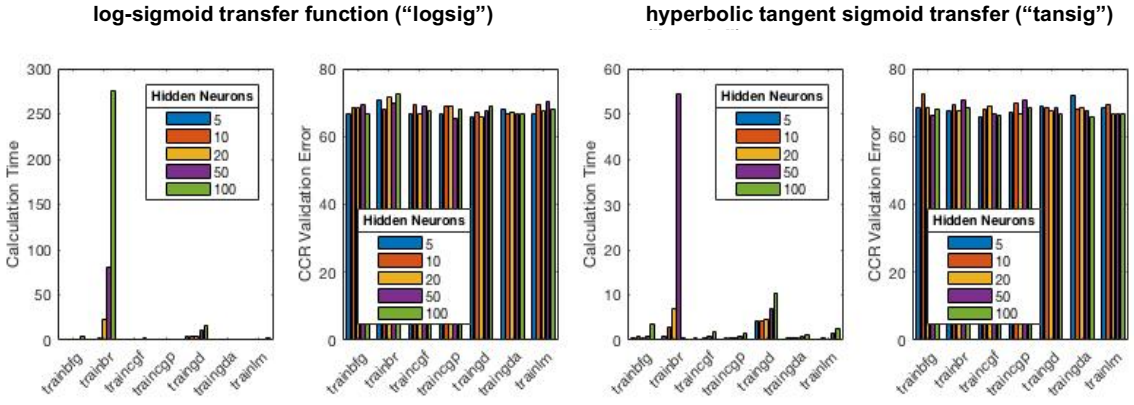

Figure 22: CCR and calculation times for patternnet neural network with one hidden layer, based on the three best principal components.

Results show that all network architectures for a one-hidden-layer patternnet neural network using the best three principal components as input lead to a similar performance. With the CCR ratio's laying on average between 65% - 70% for all network architectures, the performance seems to be worse than networks based on all 11 input variables. It is however remarkable that the calculation times are much lower, in comparison to networks estimated with all 11 input variables. The calculation time is in particularly a lot lower for the Bayesian regularization backpropagation training algorithm with a higher number of hidden neurons. The calculation time of the Bayesian

regularization backpropagation training algorithm remains however much higher for a larger number of hidden neurons, in comparison to the other training algorithms.

Since all network architectures seem to have a similar performance, it is not clear which architecture leads to the best output. The best training algorithm is probably any other than the calculation-intensive Bayesian regularization backpropagation. Since a high number of hidden neurons doesn't result in a better performance, a low number of hidden neurons (e.g. 5 or 10) is recommended in order to keep the complexity of the network model low. Log-sigmoid transfer function is preferable because its use reduces the calculation times of most of the training algorithms (with the exception of "trainbr"), while having the same performance as the "tansig" transfer function.

5.3. Problem 2: character recognition with Hopfield networks

A Hopfield network can be used to recognize uppercase and lowercase letters of the alphabet, which can be represented by 7x5 black-white pixel images. The input letters that will be stored in the Hopfield network and should be retrieved are 'simondelacABCDEFGHIJKLMNOPQRSTUVWXYZ'. Each letter can be represented by a 7x5 = 35 length vector with only the values 1 or -1 for each component in the vector. Consequently, this means that a Hopfield network consisting of 35 neurons will be used. Figure 23 below shows the binary images of 10 first letters ('simondelac').

Figure 23: binary images of the first 10 letters 'simondelac'.

First a Hopfield network that stores the first five letters of the series ('s', 'i', 'm', 'o', 'n') explicitly in its minima of the error function (attractor states) needs to be trained. Using the right input of these five letters, the interconnection weights of the network will be set according to the Hebb rule. To test the retrieval capability of the network, a new dataset containing distorted patterns for these five letters will be given as input, to see if the network is able to recall these distorted patterns. The distorted pattern for each character ('s', 'i', 'm', 'o', 'n') contains the original character pattern, where 3 randomly chosen pixels are inverted (-1 to 1 or 1 to -1). Because the three changed pixels are chosen at random, a total of 1000 distortion iterations were executed. This means that each of the five first characters were distorted 1000 times, each with 3 other pixels randomly inverted. The results of classifying these 5 x 1000 = 5000 different distorted characters with the Hopfield net are shown in Figure 24. Both graphs show the proportion of correct classifications for a distorted certain type of letter. The graph on the left shows the amount of correct classifications for 100 time steps/iterations in the Hopfield net, on the right for 1000 time steps/iterations.

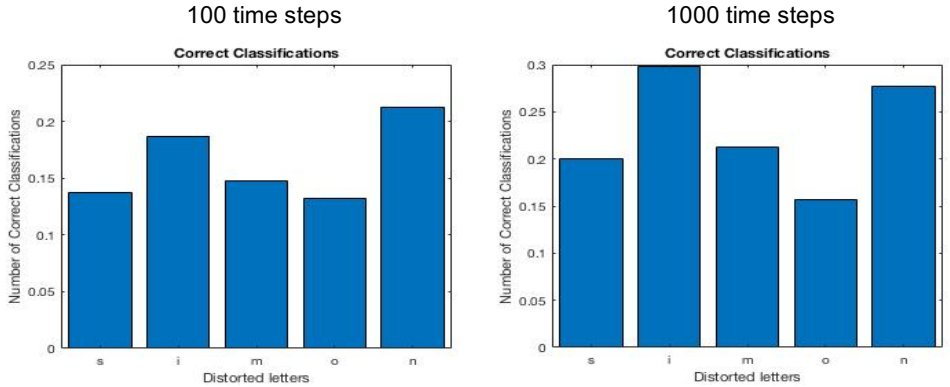

Figure 24: from left to right: Proportion of correct classifications for Hopfield network with a respective maximum of 100 and 1000 iterations to reach convergence.

The graphs on Figure 24 seems to suggest that the Hopfield network is not good at correctly classifying the distorted versions of the letters 's', 'i', 'm', 'o' and 'n'. The proportion of correct classifications remains low for both number of time steps/iterations. Increasing the number of iterations from 100 to 1000 seems to increase the proportion of correct classifications only marginally. The performance does not seem to improve significantly when increasing the number of iterations. Distorted letters can converge to spurious states in the Hopfield net. These are minima in the error functions that do not correspond to any of the letters that were 'stored' in the net during the initial training. The reconstructed images of the three most recurring spurious states are presented in Figure 25.

Figure 25: Three most recurring spurious states.

It is not clear whether distorted letters of a certain type converge more often to a certain spurious state. When distorted 'i' characters converge to a spurious state, this seems to correspond often with the one in the middle of Figure 25.

When the right characters are not retrieved this mainly depends on converging to spurious patterns or not converging at all. The theoretical, maximal number of vectors that can be stored in the minima of the error function of a Hopfield network is approximately equal to $N/(4 * \log(N))$, with N being the number of neurons. Since the number of neurons is equal to the number of components in each character (number of 1/-1 values to define each character), the theoretical loading capacity of the Hopfield network here is approximately equal to 5,67. We could say that the theoretical CLC (critical loading capacity) appears to be equal to ~5. In theory, a maximum number of five characters can be stored within the Hopfield network and recalled without error.
It is now possible to plot the number of character patterns P, stored in the Hopfield network (with 35 neurons), in function of the error. The retrieval capability will be tested by randomly distorting 3 pixels in the P different characters that need to be retrieved and giving them as input to the Hopfield network. The pixels of the final output for each distorted character will be rounded to a final value of 1 or -1. The final error for a Hopfield network with P characters stored is then equal to the total number of wrong pixel values over all retrieved, distorted P characters. Since this distorting happens randomly, for each number of P characters chosen to be stored, the error will be averaged over a number of 20 iterations with P slightly other distorted characters, in order to get a stable and representative evaluation. The P characters that will be stored correspond to the first number of P characters in the letter sequence 'simondelacABCDEFGHIJKLMNOPQRSTUVWXYZ'.
Some characters are more likely to be correctly to be recognized then others. These characters converge much easier to an attractor state, than others who converge more often to spurious states or do not converge at all. Figure 24 illustrates this: some characters are more likely to recognized than others.
When calculating the error across all the types of distorted characters, calculating the average error can perhaps lead to the wrong conclusions. Here, the error is not just calculated as the average across all P distorted characters and 20 iterations. The errors calculated over 20 iterations are averaged for each type of the P distorted characters independently. After calculating these averages, the median of these averages is chosen as a representative error. This gives a more robust representation against the spurious states and different character types.

The left graph of Figure 26 presents the 'median of averages' error in function of the number of P characters for a total of 100 time steps / iterations to achieve convergence when inserting a new input character. The graph in the middle right shows the error results for a total of 1000 time steps and on the right for 2000 time steps. It is clear that the critical loading capacity remains the same, namely 17, when increasing the number of time steps above 100. Increasing the number of time steps could make it possible to increase the chance of convergence and thus perhaps lower the error for each P number of characters. This is not really the case when increasing the number of time steps from 100 to 1000 and 2000. This indicates that convergence is almost certain for 100 time steps when distorted characters were given to the Hopfield network as new inputs. Because convergence is the most likely outcome, the error probably not comes from a lack of it. The error probably mainly results from the existence of spurious states. New inputs can converge to these spurious states, which results in pixel error(s). When P increases above the critical loading capacity of 17, it is clear that the error increases. According to the theory a higher P number of initially stored patterns is most likely to increase the number of existing spurious states. Since the error increases for a higher P, even when the number of time steps is very large (1000, 2000), it is probable to assume that the higher error results from the existence of a higher number of spurious states. A higher number of spurious states makes it more likely for distorted character inputs to converge to one of these and result in a wrong output and thus increasing the final error rate.

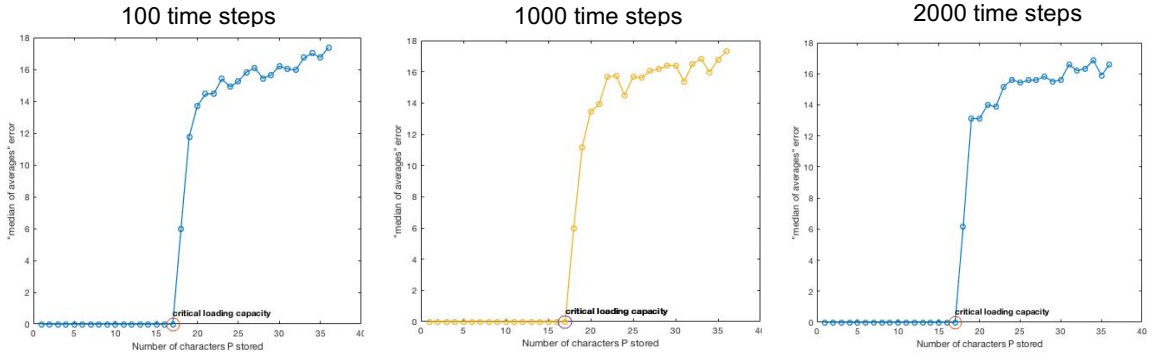

Figure 26: 'median of averages' error in function of the number of P characters stored.

It hard is to increase the number of initially stored characters, above the critical load capacity of 17, without increasing the error. The critical load capacity and thus the amount of patterns that can be stored are positively related with the number of neurons in the Hopfield network. In order to increase the storage capacity, some suggestion must be made in order to increase the number of neurons in the Hopfield network.
In this exercise, a Hopfield network with 35 neurons was used, each one corresponding with a certain pixel. Increasing the numbers of neurons and storage must be achieved by finding a way to increase the dimensionality of the characters that are stored (and thus also the distorted characters that are fed to the Hopfield network for retrieval) without losing the information of the different characters. A solution may be to add a few dimensions to each character consisting of the same values (e.g. 1 or -1) for all the different characters. This doesn't really change the data of the original characters, but does increase the dimensionality. As a consequence, this increases the number of neurons in the Hopfield network and thus also the critical storage capacity.
The dimensions of each character type are increased to 70, such that the number of neurons in the Hopfield network also increases to 70. This should be able to increase the storage capacity. This means that the number of dimensions for each (distorted) character was increased from the initial 35 to 70. Each character vector was added an extra number of 35 dimensions. For a certain character, the values of these additional 35 dimensions are equal to the values of the first 35 dimensions. In other words, each character is represented by two times the vector of the original 35 dimensions. This means that the value of 36th dimension will equal the value of the 1st dimension. The value of the 37th dimension will equal the value of the 2nd dimension and so on.
By doing this the size of the dimensions is doubled, but the information that the data represents remains kind of the same. The distortion of the characters will only be applied to three of first 35 dimensions that represent the initial character information.
Figure 27 shows the 'median of averages' error in function of a P number of characters stored, whereby each character is represented by 70 dimensions (2 times the initial 35-dimensional vector). A Hopfield network with a maximum of 100 time steps to converge was used here. The graph shows that the errors do not start to increase until the number of patterns stored is equal to 22. By doubling the dimensions of the characters, the critical load capacity has increased from 17 to 22. Increasing the dimensions appears to be having a diminishing returns impact on the critical storage capacity of the Hopfield net.

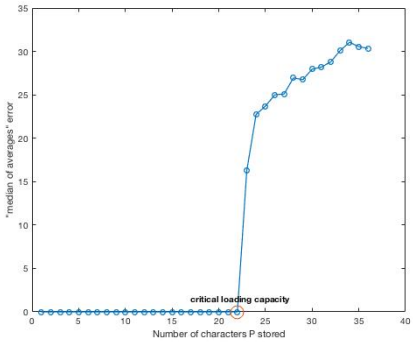

Figure 27: 'median of averages' error in function of the number of P characters stored, whereby each character is represented by 70 dimensions. The amount of time steps was set constant to 100.

6. Appendix: Matlab Scripts

6.1. Exercise: Supervise learning and generalization

```
%% creating the data set for a sinus 2-period wave
% training size
data_size = 100;
% the jump size
jump_size = 1/(data_size) * (4*pi);

% creating the X array to store the data size in
X = zeros(1,data_size);

% creating x values
for i = 1:data_size
    X(i) = i*jump_size;
end

% creating the y-values of the sinus wave
Y = sin(X);

% number of neurons
number_neurons = [5, 10, 20, 40, 100];

% different training algorithms
training_algs = { 'traingd', 'traingda', 'traincgf', 'traincgp', 'trainbfg', 'trainlm', 'trainbr'};

% matrix to store speed values in -> speed_values
speed_values = zeros(length(training_algs), length(number_neurons));
RMSE_training_average_error = zeros(length(training_algs), length(number_neurons));
RMSE_test_average_error = zeros(length(training_algs), length(number_neurons));

% number of iterarions for each execution
n_iterations = 40;

%% Number Neurons
% index for speed_values matrix
i = 0;
j= 0;

for train_alg = training_algs

    % updating row index that corresponds to the training algorithm
    i  = i + 1;

    for n_number = number_neurons

        % creating the feedforward neural network
        train_alg = char(train_alg);
        net = feedforwardnet(n_number, train_alg);

        % array to store the calculation times over the 20 iterations
        calc_time_values = zeros(1,n_iterations);

        % array to store the RMSE training errors over the 20 iterations
        RMSE_training_values  = zeros(1,n_iterations);

        % array to store the RMSE test errors over the 20 iterations
        RMSE_test_values = zeros(1,n_iterations);

        for it = 1:n_iterations

            % proportion of rows to select for training
            p = 0.8;
            % create logical index vector
            training_vector = false(1, data_size);
            training_vector(1:round(p*data_size)) = true;
            % randomise order
            training_vector = training_vector(randperm(data_size));
            % training set
            X_train = X(training_vector);
            % test set
            X_test = X(~training_vector);

            % Y-values of the training and test set
            Y_train = sin(X_train);
            Y_test = sin(X_test);

            % training the neural network and measuring average calculation time
            tic;
            net = train(net, X_train, Y_train);
            calc_time = toc;
```

```
            % the calculation time values over 20 iterations on average
            calc_time_values(it) = calc_time;

            % RMSE training set
            RMSE_training_values(it) = sqrt(sum((Y_train - sim(net, X_train)).^2));

            % RMSE test set
            RMSE_test_values(it) = sqrt(sum((Y_test - sim(net, X_test)).^2));

        end

        % average calculation time
        average_calc_time = mean(calc_time_values);

        % average RMSE training error
        average_RMSE_training = mean(RMSE_training_values);

        % average RMSE test error
        average_RMSE_test = mean(RMSE_test_values);

        % updating the column index that corresponds to the amount of noise
        % added
        j = j + 1;

        % adding the calculation time values to the matrix
        speed_values(i,j) = average_calc_time;

        % adding RMSE training error values to the matrix
        RMSE_training_average_error(i,j) = average_RMSE_training;

        % adding the RMSE test error to the matrix
        RMSE_test_average_error(i,j) = average_RMSE_test;

    end

    j = 0;
end

%% Bar Plot Speed
% returning a barplot for the Speed
names = categorical(training_algs);
barplot = bar(names, speed_values);
title("Effect of Amount Hidden Neurons on Calculation Time");
ylabel("Calculation Time")
legend_bar = legend(barplot,{"5", "10", "20", "40", "100"});
title(legend_bar, "Hidden Neurons");

number_neurons = [1, 5, 10, 20, 40, 100];
ylim([0 5]);


%% Bar Plot Training RMSE
% returning a barplot for the RMSE Training Error
names = categorical(training_algs);
barplot = bar(names, RMSE_training_average_error);
title("Effect of Amount Hidden Neurons on Training Error");
ylabel("RMSE Training Error")
legend_bar = legend(barplot,{"5", "10", "20", "40", "100"});
title(legend_bar, "Hidden Neurons");

%% Bar Plot Test RMSE
% returning a barplot for the RMSE Test Error
names = categorical(training_algs);
barplot = bar(names, RMSE_test_average_error);
title("Effect of Amount Hidden Neurons on Test Error");
ylabel("RMSE Test Error")
legend_bar = legend(barplot,{"5", "10", "20", "40", "100"});
title(legend_bar, "Hidden Neurons");
```

```
%% creating the data set for a sinus 2-period wave
% training size
data_size = 100;
% the jump size
jump_size = 1/(data_size) * (4*pi);

% creating the X array to store the data size in
X = zeros(1,data_size);

% creating x values
for i = 1:data_size
    X(i) = i*jump_size;
end
```

```
% creating the y-values of the sinus wave
Y = sin(X);

% different training algorithms
training_algs = { 'traingd', 'traingda', 'traincgf', 'traincgp', 'trainbfg', 'trainlm', 'trainbr'};

% the increasing noise steps
noise_steps = [0.1, 0.2, 0.3, 0.4, 0.5];

% matrices to store the speed and error values in
speed_values = zeros(length(training_algs), length(noise_steps));
RMSE_training_average_error = zeros(length(training_algs), length(noise_steps));
RMSE_test_average_error = zeros(length(training_algs), length(noise_steps));

% number of iterarions for each execution
n_iterations = 40;

%% Noise
% index for speed_values matrix
i = 0;
j= 0;

for train_alg = training_algs

    % updating row index that corresponds to the training algorithm
    i  = i + 1;

    for noise_step = noise_steps

        % creating the feedforward neural network
        train_alg = char(train_alg);
        net = feedforwardnet(10, train_alg);

        % array to store the calculation times over the 20 iterations
        calc_time_values = zeros(1,n_iterations);

        % array to store the RMSE training errors over the 20 iterations
        RMSE_training_values  = zeros(1,n_iterations);

        % array to store the RMSE test errors over the 20 iterations
        RMSE_test_values = zeros(1,n_iterations);

        for it = 1:n_iterations

            % proportion of rows to select for training
            p = 0.8;
            % create logical index vector
            training_vector = false(1, data_size);
            training_vector(1:round(p*data_size)) = true;
            % randomise order
            training_vector = training_vector(randperm(data_size));
            % training set
            X_train = X(training_vector);
            % test set
            X_test = X(~training_vector);

            % Y-values of the training and test set
            Y_train = sin(X_train);
            Y_test = sin(X_test);

            Y_train_noise = zeros(1, length(Y_train));
            % adding random noise to the training set
            for n = 1:length(data_size)
                Y_train_noise(n) = Y_train(n) + noise_step * normrnd(0,1);
            end

            % training the neural network and measuring average calculation time
            tic;
            net = train(net, X_train, Y_train_noise);
            calc_time = toc;

            % the calculation time values over 20 iterations on average
            calc_time_values(it) = calc_time;

            % RMSE training set
            RMSE_training_values(it) = sqrt(sum((Y_train_noise - sim(net, X_train)).^2));

            % RMSE test set
            RMSE_test_values(it) = sqrt(sum((Y_test - sim(net, X_test)).^2));

        end

        % average calculation time
        average_calc_time = mean(calc_time_values);
```

```
        % average RMSE training error
        average_RMSE_training = mean(RMSE_training_values);

        % average RMSE test error
        average_RMSE_test = mean(RMSE_test_values);

        % updating the column index that corresponds to the amount of noise
        % added
        j = j + 1;

        % adding the calculation time values to the matrix
        speed_values(i,j) = average_calc_time;

        % adding RMSE training error values to the matrix
        RMSE_training_average_error(i,j) = average_RMSE_training;

        % adding the RMSE test error to the matrix
        RMSE_test_average_error(i,j) = average_RMSE_test;

    end

    j = 0;
end

%% Bar Plot Speed
% returning a barplot for the Speed
names = categorical(training_algs);
barplot = bar(names,speed_values);
title("Effect of Noise on Calculation Time");
ylabel("Calculation Time")
legend_bar = legend(barplot,{"0.1", "0.2", "0.3", "0.4", "0.5"});
title(legend_bar, "Noise Step");

%% Bar Plot Training RMSE
% returning a barplot for the RMSE Training Error
names = categorical(training_algs);
barplot = bar(names, RMSE_training_average_error);
title("Effect of Noise on Training Error");
ylabel("RMSE Training Error")
legend_bar = legend(barplot,{"0.1", "0.2", "0.3", "0.4", "0.5"});
title(legend_bar, "Noise Step");

%% Bar Plot Test RMSE
% returning a barplot for the RMSE Test Error
names = categorical(training_algs);
barplot = bar(names, RMSE_test_average_error);
title("Effect of Noise on Test Error");
ylabel("RMSE Test Error")
legend_bar = legend(barplot,{"0.1", "0.2", "0.3", "0.4", "0.5"});
title(legend_bar, "Noise Step");
```

6.2. Exercise: Recurrent neural networks

6.2.1. Hopfield net

```
% parameters
noiselevel = 1;
iterations = 100;
number = 9;

number = number + 1;

%loading digit data and preparation
% loading the digits dataset
load digits

% getting the numbr of observations and number of variables
[n_observations, n_variables] = size(X);

% Values must be +1 or -1 in order for the Hopfield networks to function
X(X==0)=-1;

%Attractors of the Hopfield network -> the correct digits of the Hopfield
%network
% index 1 - 20 -> digit 0
zero = X(1,:);
% index 21 - 40 -> digit 1
one = X(21,:);
% index 41 - 60 -> digit 2
two = X(41,:);
% index 61 - 80 -> digit 3
three = X(61,:);
% index 81 - 100 -> digit 4
```

```
four = X(81,:);
% index 101 - 120 -> digit 5
five = X(101,:);
% ...
six = X(121,:);
seven = X(141,:);
eight = X(161,:);
nine = X(181,:);

% index indicating the different digit types
index_dig = [1,21,41,61,81,101,121,141,161,181];

% Intializing Hopfield net
% defining the right attractors
Attractor_T = [zero;one;two;three;four;five;six;seven;eight;nine]';

% the number of different digits
num_dig = size(Attractor_T,2);

% creating the Hopfield network with initial starting digits as attractor
% states
net = newhop(Attractor_T);

% Check if initial digits are attractors
[Y,~,~] = sim(net,num_dig,[],Attractor_T);
Y = Y';

% the array to store the rigth and wrong classifications in
digit_classifications = zeros(10,11);

% the different noise levels
noises = [1, 2, 4, 10];

% the array to store the number of misclassified digits
digit_classified = zeros(10,4);

% the array that stores the amount that are assigned to spurious states
digit_spurious = zeros(10,4);

% 10 different digit observations from the orginal digits data set
X_10 = X(index_dig,:);

%%
for l = 1:4

    noiselevel = noises(l);

    % the array to store the rigth and wrong classifications in
    digit_classifications = zeros(10,11);


for a = 1:10

% choosing certain observation that needs to be checked
observations = X_10(a,:);

for n = 1:100

    % adding noise
    observations_noise = observations;
    for k=1:size(observations,1)
        % adding noise taking from the Gaussian standard distribution
        observations_noise(k,:) = observations(k,:) + noiselevel*normrnd(0,1);
    end

    for i = 1:size(observations_noise,1)

        % certain observed digit
        digit_input = observations_noise(i,:);
        digit_input = digit_input';

        % getting the best classification output fromt he Hopfield network
        T = {digit_input};
        [digit_output,~,~] = sim(net,{1 ,iterations},{},T);
        digit_output = digit_output{1, iterations};
        digit_output = digit_output';

        % the array to store the digits to check in
        digits_to_check = Attractor_T';

        % t variable in order to check if a right digit was recognised
        t = 0;

        for j = 1:10
```

```
            if(isequal(digit_output, digits_to_check(j,:)) == true)

                digit_classifications(a,j) = digit_classifications(a,j) + 1;

                t = 1;

            end
        end

        if(t == 0)
            digit_classifications(a,11) = digit_classifications(a,11) + 1;

            % plotting the spurious state
            hold on
            digit = reshape(digit_output,15,16)';
            imshow(digit)
            figure;

        end
    end
end

% Amounts Missclassified
n_classified = digit_classifications(a,a);

assigned_spurious = digit_classifications(a,11);

digit_classified(a,l) = n_classified;

digit_spurious(a,l) = assigned_spurious;

end
end

digit_classified
digit_spurious

%% Barplot of the (mis)classifications
categories = {'0', '1', '2', '3', '4', '5', '6', '7', '8', '9'};
names = categorical(categories);
barplot = bar(names,digit_classified);
title("Effect of Noise on Number of Correct Classifications ");
ylabel("Number Correct Classifications")
legend_bar = legend(barplot,{"1", "2", "4", "10"});
title(legend_bar, "Noise Step");

%% barplot of the amount of digits of certain type assigned to spurious vectors
categories = {'0', '1', '2', '3', '4', '5', '6', '7', '8', '9'};
names = categorical(categories);
barplot = bar(names,digit_spurious);
title("Effect of Noise on Spurious States Assignment");
ylabel("Number Correct Classifications")
legend_bar = legend(barplot,{"1", "2", "4", "10"});
title(legend_bar, "Noise Step");
ylim([0 6])
```

```
% parameters
noiselevel = 2;
iterations = 100;
number = 9;

number = number + 1;

%loading digit data and preparation
% loading the digits dataset
load digits

% getting the numbr of observations and number of variables
[n_observations, n_variables] = size(X);

% Values must be +1 or -1 in order for the Hopfield networks to function
X(X==0)=-1;

%Attractors of the Hopfield network -> the correct digits of the Hopfield
%network
% index 1 - 20 -> digit 0
zero = X(1,:);
% index 21 - 40 -> digit 1
one = X(21,:);
% index 41 - 60 -> digit 2
two = X(41,:);
% index 61 - 80 -> digit 3
three = X(61,:);
```

```
% index 81 - 100 -> digit 4
four = X(81,:);
% index 101 - 120 -> digit 5
five = X(101,:);
% ...
six = X(121,:);
seven = X(141,:);
eight = X(161,:);
nine = X(181,:);

% index indicating the different digit types
index_dig = [1,21,41,61,81,101,121,141,161,181];

% Intializing Hopfield net
% defining the right attractors
Attractor_T = [zero;one;two;three;four;five;six;seven;eight;nine]';

% the number of different digits
num_dig = size(Attractor_T,2);

% creating the Hopfield network with initial starting digits as attractor
% states
net = newhop(Attractor_T);

% Check if initial digits are attractors
[Y,~,~] = sim(net,num_dig,[],Attractor_T);
Y = Y';

% the array to store the rigth and wrong classifications in
digit_classifications = zeros(10,11);

% the different noise levels
noises = [1, 2, 4, 10];

% different time steps
time_steps = [10 50 100 1000];

% the array to store the number of misclassified digits
digit_classified = zeros(10,4);

% the array that stores the amount that are assigned to spurious states
digit_spurious = zeros(10,4);

% 10 different digit observations from the orginal digits data set
X_10 = X(index_dig,:);

%%
for l = 1:4

    iterations = time_steps(l);

    % the array to store the rigth and wrong classifications in
    digit_classifications = zeros(10,11);


for a = 1:10

% choosing certain observation that needs to be checked
observations = X_10(a,:);

for n = 1:100

    % adding noise
    observations_noise = observations;
    for k=1:size(observations,1)
        % adding noise taking from the Gaussian standard distribution
        observations_noise(k,:) = observations(k,:) + noiselevel*normrnd(0,1);
    end

    for i = 1:size(observations_noise,1)

        % certain observed digit
        digit_input = observations_noise(i,:);
        digit_input = digit_input';

        % getting the best classification output from the Hopfield network
        T = {digit_input};
        [digit_output,~,~] = sim(net,{1 ,iterations},{},T);
        digit_output = digit_output{1, iterations};
        digit_output = digit_output';

        % the array to store the digits to check in
        digits_to_check = Attractor_T';
```

```
        % t variable in order to check if a right digit was recognised
        t = 0;

        for j = 1:10
            if(isequal(digit_output, digits_to_check(j,:)) == true)

                digit_classifications(a,j) = digit_classifications(a,j) + 1;

                t = 1;

            end
        end

        if(t == 0)
            digit_classifications(a,11) = digit_classifications(a,11) + 1;

            % plotting the spurious state
            % hold on
            % digit = reshape(digit_output,15,16)';
            % imshow(digit)
            % figure;

        end
    end
end

% Amounts Missclassified
n_classified = digit_classifications(a,a);

assigned_spurious = digit_classifications(a,11);

digit_classified(a,l) = n_classified;

digit_spurious(a,l) = assigned_spurious;

end
end

digit_classified
digit_spurious

%% Barplot of the (mis)classifications
categories = {'0', '1', '2', '3', '4', '5', '6', '7', '8', '9'};
names = categorical(categories);
barplot = bar(names,digit_classified);
title("Effect of Time Steps on Number of Correct Classifications ");
ylabel("Number Correct Classifications")
legend_bar = legend(barplot,{"10", "50", "100", "1000"});
title(legend_bar, "Time Steps");

%% barplot of the amount of digits of certain type assigned to spurious vectors
categories = {'0', '1', '2', '3', '4', '5', '6', '7', '8', '9'};
names = categorical(categories);
barplot = bar(names,digit_spurious);
title("Effect of Time Steps on Spurious States Assignment");
ylabel("Number Correct Classifications")
legend_bar = legend(barplot,{"10", "50", "100", "1000"});
title(legend_bar, "Time Steps");
```

6.2.2. Elman network to model Hammerstein time series

```
%% Set the parameters of the run
n = 2000;                % Total number of samples
ne = 1000;               % Number of epochs
perc_training = 0.7;     % Number between 0 and 1. The validation set will be 1-perc_training.

if perc_training >= 1 || perc_training <= 0
    error('The training set is ill defined. The variable perc_training should be between 0 and 1')
end

%% Create the samples
% Allocate memory
u = zeros(1, n);
x = zeros(1, n);
y = zeros(1, n);

% Initialize u, x and y
u(1)=randn;
x(1)=rand+sin(u(1));
y(1)=x(1);

% Calculate the samples
for i=2:n
    u(i)=randn;
```

```
    x(i)=.8*x(i-1)+sin(u(i));
    y(i)=x(i);
end

%% matrix to store values in
% first amount of hidden neurons
hidden_neurons = [1 5 10 20 40];

% the matrix to store the results in
% each row represents a certain metric -> first row has the speed, second
% RSME test error
results = zeros(2, length(hidden_neurons));

% number of iterations to get a stable metric
n_iterations = 40;

%% Results for different number of hidden neurons

j =0;

for number_neurons = 1:5

    % index to store the values for different hidden neurons
    j = j + 1;

    % array to store the calculation times over the 20 iterations
    calc_time_values = zeros(1,n_iterations);

    % array to store the RMSE test errors over the 20 iterations
    R2_values = zeros(1,n_iterations);

    for it = 1:n_iterations

        % proportion of rows to select for training-validation
        p = 0.8;
        % create logical index vector
        training_vector = false(1, n);
        training_vector(1:round(p*n)) = true;
        % randomise order
        training_vector = training_vector(randperm(n));
        % training set
        X = u(training_vector);
        % test set
        X_test = u(~training_vector);

        % Y-values of the training and test set
        T = y(training_vector);
        T_test = y(~training_vector);

        % changing the number of neurons
        net = newelm(X, T, hidden_neurons(number_neurons));

        net.trainParam.epochs = ne;         % Number of epochs
        net.divideParam.testRatio = 0;
        net.divideParam.valRatio = 1-perc_training; % validation set ratio
        net.divideParam.trainRatio = perc_training; % training set ratio

        tic
        net = train(net,X,T); % Training
        calc_time = toc;

        T_test_sim = sim(net,X_test); % Testing

        % the Rsquared values
        T_test = T_test';
        T_test_sim = T_test_sim';

        % training predictions
        T_sim = sim(net,X);
        T_sim = T_sim';

        % correlation
        % corr = corrcoef(T_test, T_test_sim);

        corr = corrcoef(T, T_sim);

        % R squared
        Rsquared = (corr(1,2))^2;

        % storing the calculation over 40 iterations
        calc_time_values(it) = calc_time;

        % storing test RMSE values over 40 iterations
        R2_values(it) = Rsquared;
```

```
    end
        % adding speed values to results matrixvalues to the matrix
        results(1,j) = mean(calc_time_values);

        % adding the RMSE test error to the matrix
        results(2,j) = mean(R2_values);

end


%% Barplot for Calculation for the different hidden neurons
% returning a barplots for different amounts of hidden neurons
names = categorical({'1 neuron', '5 neurons', '10 neurons', '20 neurons', '40 neurons'});
names = reordercats(names,{'1 neuron', '5 neurons', '10 neurons', '20 neurons', '40 neurons'});
time_values = results(1,:);
bar(names,time_values);
title("Effect of Hidden Neurons on Calculation Time");

%% Barplot for the R squared for the different hidden neurons
% returning a barplots for different amounts of hidden neurons
names = categorical({'1 neuron', '5 neurons', '10 neurons', '20 neurons', '40 neurons'});
names = reordercats(names,{'1 neuron', '5 neurons', '10 neurons', '20 neurons', '40 neurons'});
r_values = results(2,:);
bar(names,r_values);
title("Effect of Hidden Neurons on R-squared");
```

```
%% Set the parameters of the run
ne = 1000;               % Number of epochs
perc_training = 0.7;     % Number between 0 and 1. The validation set will be 1-perc_training.
number_neurons = 5; % Keeping number of hidden neurons constant

if perc_training >= 1 || perc_training <= 0
    error('The training set is ill defined. The variable perc_training should be between 0 and 1')
end

% number of samples
samples = [50 100 500 1000 10000];

% matrix to store values in
% the matrix to store the results in
% each row represents a certain metric -> first row has the speed, second
% R squared
results = zeros(2, length(samples));
%%
for z = 1:5

    n = samples(z);

% Create the samples
% Allocate memory
u = zeros(1, n);
x = zeros(1, n);
y = zeros(1, n);

% Initialize u, x and y
u(1)=randn;
x(1)=rand+sin(u(1));
y(1)=x(1);

% Calculate the samples
for i=2:n
    u(i)=randn;
    x(i)=.8*x(i-1)+sin(u(i));
    y(i)=x(i);
end

% number of iterations to get a stable metric
n_iterations = 40;

    % array to store the calculation times over the 20 iterations
    calc_time_values = zeros(1,n_iterations);

    % array to store the RMSE test errors over the 20 iterations
    R2_values = zeros(1,n_iterations);

    for it = 1:n_iterations

        % proportion of rows to select for training-validation
        p = 0.8;
        % create logical index vector
        training_vector = false(1, n);
        training_vector(1:round(p*n)) = true;
```

```
        % randomise order
        training_vector = training_vector(randperm(n));
        % training set
        X = u(training_vector);
        % test set
        X_test = u(~training_vector);

        % Y-values of the training and test set
        T = y(training_vector);
        T_test = y(~training_vector);

        % changing the number of neurons
        net = newelm(X, T, hidden_neurons(number_neurons));

        net.trainParam.epochs = ne;         % Number of epochs
        net.divideParam.testRatio = 0;
        net.divideParam.valRatio = 1-perc_training; % validation set ratio
        net.divideParam.trainRatio = perc_training; % training set ratio

        tic
        net = train(net,X,T); % Training
        calc_time = toc;

        T_test_sim = sim(net,X_test); % Testing

        % the Rsquared values
        T_test = T_test';
        T_test_sim = T_test_sim';

        % training predictions
        T_sim = sim(net,X);
        T_sim = T_sim';

        % correlation
        % corr = corrcoef(T_test, T_test_sim);
        corr = corrcoef(T_test, T_test_sim);

        % R squared
        Rsquared = (corr(1,2))^2;

        % storing the calculation over 40 iterations
        calc_time_values(it) = calc_time;

        % storing test RMSE values over 40 iterations
        R2_values(it) = Rsquared;

    end
        % adding speed values to results matrix values to the matrix
        results(1,z) = mean(calc_time_values);

        % adding the RMSE test error to the matrix
        results(2,z) = mean(R2_values);

end


%% Barplot for Calculation for different numbers of samples
% returning a barplots for different numbers of samples
names = categorical({'50 samples', '100 samples', '500 samples', '1000 samples', '10000 samples'});
names = reordercats(names,{'50 samples', '100 samples', '500 samples', '1000 samples', '10000
samples'});
time_values = results(1,:);
bar(names,time_values);
title("Effect of Number of Samples on Calculation Time");

%% Barplot for the R squared for different numbers of samples
% returning a barplots for different number of samples
names = categorical({'50 samples', '100 samples', '500 samples', '1000 samples', '10000 samples'});
names = reordercats(names,{'50 samples', '100 samples', '500 samples', '1000 samples', '10000
samples'});
r_values = results(2,:);
bar(names,r_values);
title("Effect of Number of Samples on R-squared");
```

```
%% Set the parameters of the run
perc_training = 0.7;    % Number between 0 and 1. The validation set will be 1-perc_training.
number_neurons = 5; % Keeping number of hidden neurons constant
n = 2000;

if perc_training >= 1 || perc_training <= 0
    error('The training set is ill defined. The variable perc_training should be between 0 and 1')
end
```

```
% number of samples
epochs = [10 100 1000 2000 5000];

% matrix to store values in
% the matrix to store the results in
% each row represents a certain metric -> first row has the speed, second
% R squared
results = zeros(2, length(samples));

% Create the samples
% Allocate memory
u = zeros(1, n);
x = zeros(1, n);
y = zeros(1, n);

% Initialize u, x and y
u(1)=randn;
x(1)=rand+sin(u(1));
y(1)=x(1);

% Calculate the samples
for i=2:n
    u(i)=randn;
    x(i)=.8*x(i-1)+sin(u(i));
    y(i)=x(i);
end
%%
for z = 1:5

ne = epochs(z);

% number of iterations to get a stable metric
n_iterations = 40;

    % array to store the calculation times over the 20 iterations
    calc_time_values = zeros(1,n_iterations);

    % array to store the RMSE test errors over the 20 iterations
    R2_values = zeros(1,n_iterations);

    for it = 1:n_iterations

        % proportion of rows to select for training-validation
        p = 0.8;
        % create logical index vector
        training_vector = false(1, n);
        training_vector(1:round(p*n)) = true;
        % randomise order
        training_vector = training_vector(randperm(n));
        % training set
        X = u(training_vector);
        % test set
        X_test = u(~training_vector);

        % Y-values of the training and test set
        T = y(training_vector);
        T_test = y(~training_vector);

        % changing the number of neurons
        net = newelm(X, T, hidden_neurons(number_neurons));

        net.trainParam.epochs = ne;          % Number of epochs
        net.divideParam.testRatio = 0;
        net.divideParam.valRatio = 1-perc_training; % validation set ratio
        net.divideParam.trainRatio = perc_training; % training set ratio

        tic
        net = train(net,X,T); % Training
        calc_time = toc;

        T_test_sim = sim(net,X_test); % Testing

        % the Rsquared values
        T_test = T_test';
        T_test_sim = T_test_sim';

        % training predictions
        T_sim = sim(net,X);
        T_sim = T_sim';

        % correlation
        % corr = corrcoef(T_test, T_test_sim);
        corr = corrcoef(T_test, T_test_sim);
```

```
        % R squared
        Rsquared = (corr(1,2))^2;

        % storing the calculation over 40 iterations
        calc_time_values(it) = calc_time;

        % storing test RMSE values over 40 iterations
        R2_values(it) = Rsquared;

    end
        % adding speed values to results matrixvalues to the matrix
        results(1,z) = mean(calc_time_values);

        % adding the RMSE test error to the matrix
        results(2,z) = mean(R2_values);

end


%% Barplot for Calculation for the different hidden neurons
% returning a barplots for different amounts of hidden neurons

names = categorical({'10 epochs', '100 epochs', '1000 epochs', '2000 epochs', '5000 epochs'});
names = reordercats(names,{'10 epochs', '100 epochs', '1000 epochs', '2000 epochs', '5000 epochs'});
time_values = results(1,:);
bar(names,time_values);
title("Effect of Number of Epochs on Calculation Time");

%% Barplot for the R squared for the different hidden neurons
% returning a barplots for different amounts of hidden neurons
names = categorical({'10 epochs', '100 epochs', '1000 epochs', '2000 epochs', '5000 epochs'});
names = reordercats(names,{'10 epochs', '100 epochs', '1000 epochs', '2000 epochs', '5000 epochs'});
r_values = results(2,:);
bar(names,r_values);
title("Effect of Number of Epochs on R-squared");
```

6.3. Exercise: Unsupervised learning: PCA and SOM

6.3.1. PCA on handwritten digits

```
%% load
load threes -ascii

%% Total sum of all eigenvalues
% creating observations set
x = threes;
% rows are the observations, columns the variables
cov_matrix = cov(x);
% amount of dimensions

% determining sum of all eigenvalues
n_dimensions = size(cov_matrix,1);
[all_v, all_d] = eigs(cov_matrix,n_dimensions);
total_eigen_values = sum(diag(all_d));


%% Eigenvalue decomposition

x_plot = 1:256;
proportion_values = zeros(1,256);
error_values = zeros(1,256);

% for all values k = 1 - 256
for k = 1:256

    % returning the eigenvectors of k largest eigenvalues
    [v,d] = eigs(cov_matrix,k);

    % PROPORTION EXPLAINED
    % proportion explained by k principal components -> eigenvalues
    proportion_eigen_values = sum(diag(d));
    % proportion of variance explained
    proportion_value = proportion_eigen_values / total_eigen_values;

    % RMSE ERROR OF RECONSTRUCTION MATRIX
    transformation = v;
    % transforming the original data with k principal components as dimensions
    x_transformed = x * transformation;
    % try to reconstruct the original data_matrix
    x_estimated = x_transformed * (transformation');
    RMSE_error = sqrt(mean(mean((x - x_estimated).^2)));

    % adding values to arrays
    proportion_values(k) = proportion_value;
```

```
    error_values(k) = RMSE_error;

end



%% Plot proportion variance explained and RMSE errors on one plot
graph_plot = plot(x_plot, proportion_values, '--', x_plot, error_values);
xlabel("Number Principal Components");
legend_bar = legend(graph_plot,{"Proportion Variance Explained", "RMSE"});

%% Additional proportion explained going from 1-50 principal components and from 50-100 principal
components
proption_1 = proportion_values(50) - proportion_values(1)
proportion_2 = proportion_values(100) - proportion_values(50)

%% creating reconstruction image for the first observation for 1, 50, 100 and 256 principal
components
% returning the eigenvectors of k largest eigenvalues
array_pc = [1, 50, 100, 256];
for i = 1:4
    k = array_pc(i);

    % constructing eigenvectors and eigenvalues of the original covariance matrix
    [v,d] = eigs(cov_matrix,k);

    % RMSE ERROR OF RECONSTRUCTION MATRIX
    transformation = v;
    % transforming the original data with k principal components as dimensions
    x_transformed = x * transformation;
    % try to reconstruct the original data_matrix
    x_estimated = x_transformed * (transformation');

    subplot(1,4,i)
    imagesc(reshape(x_estimated(1,:),16,16),[0,1])
    axis off

end
```

6.3.2. SOM on Iris datasets

```
clear
clc
close all
%% looking at different topologies

topologies = {'hextop', 'gridtop', 'randtop'};
distance_functions = {'linkdist', 'dist', 'mandist'};

results = zeros(3,3);

%%

for t = 1:3

    top = topologies(t);

    for d = 1:3

        dis = distance_functions(d);

        top =char(top);
        dis = char(dis);

        % Load data
        load iris
        X = iris(:,1:end-1);
        true_labels = iris(:,end);

        % Training the SOM
        x_length = 3;
        y_length = 1;
        gridsize=[y_length x_length];
        net = newsom(X',gridsize,top,dis);
        net.trainParam.epochs = 1000;
        net = train(net,X');

        % Assigning examples to clusters
        outputs = sim(net,X');
        [~,assignment]  =  max(outputs);

        %Compare clusters with true labels
        ARI = RandIndex(assignment,true_labels);
```

```
        results(t,d) = ARI;

    end
end

%% returning barplot for Rand index for topolgy and distance function
% returning a barplot for the RMSE Test Error
names = categorical({'hextop', 'gridtop', 'randtop'});
barplot = bar(names, results);
title("Impact of Topology and Distance Function");
ylabel("Rand Index")
legend_bar = legend(barplot,{'linkdist', 'dist', 'mandist'});
title(legend_bar, "Distance Function");
```

```
%% Different number epochs
epochs = [10, 50, 100, 200, 500, 1000, 10000];

results = zeros(1,7);

%% Rand index for different number of epochs

for i = 1:7

        e = epochs(i);

        % Load data
        load iris
        X = iris(:,1:end-1);
        true_labels = iris(:,end);

        % Training the SOM
        x_length = 3;
        y_length = 1;
        gridsize=[y_length x_length];
        net = newsom(X',gridsize,'hextop', 'mandist');
        net.trainParam.epochs = e;
        net = train(net,X');

        % Assigning examples to clusters
        outputs = sim(net,X');
        [~,assignment]  =  max(outputs);

        %Compare clusters with true labels
        ARI = RandIndex(assignment,true_labels);

        results(i) = ARI;

end

%% barplot
names = categorical({'10 epochs', '50 epochs', '100 epochs','200 epochs', '500 epochs', '1000
epochs', '10000 epochs'});
names = reordercats(names,{'10 epochs', '50 epochs', '100 epochs','200 epochs', '500 epochs', '1000
epochs', '10000 epochs'});
barplot = bar(names, results);
title("Impact of Epochs")
ylabel("Rand Index");
```

6.4. Exercise: Deep Learning: Stacked Autoencoders and Convolutional Neural Networks

6.4.1. Digit classification with Stacked Autoencoders

```
clear all
close all
nntraintool('close');
nnet.guis.closeAllViews();

% Load the training data into memory
%[xTrainImages, tTrain] = digittrain_dataset;
load('digittrain_dataset');

hidden_neurons_1 = [20, 50, 100, 150, 200];
hidden_neurons_2 = [10, 20, 50, 100];

%% Multiple loops to store the results
speed_results = zeros(4,5);
accuracy_results = zeros(4,5);

%% executing deep net and normal neural networks
for n = 1:5
```

```
speed_values_nofinetuning = zeros(1,5);
speed_values_finetuned = zeros(1,5);
speed_values_pattern1 = zeros(1,5);
speed_values_pattern2 = zeros(1,5);

accuracy_values_nofinetuning = zeros(1,5);
accuracy_values_finetuned = zeros(1,5);
accuracy_values_pattern1 = zeros(1,5);
accuracy_values_pattern2 = zeros(1,5);

for j = 1:5

    % certain amount of hidden neurons
    n1 = hidden_neurons_1(n);

    % Neural networks have weights randomly initialized before training.
    % Therefore the results from training are different each time. To avoid
    % this behavior, explicitly set the random number generator seed.
    rng('default')

    % Layer 1
    hiddenSize1 = n1;
    tic
    autoenc1 = trainAutoencoder(xTrainImages,hiddenSize1, ...
    'MaxEpochs',200, ...
    'L2WeightRegularization',0.004, ...
    'SparsityRegularization',4, ...
    'SparsityProportion',0.15, ...
    'ScaleData', false);
    toc_1 = toc;
    feat1 = encode(autoenc1,xTrainImages);


    % Layer 2
    hiddenSize2 = 50;
    tic
    autoenc2 = trainAutoencoder(feat1,hiddenSize2, ...
    'MaxEpochs',200, ...
    'L2WeightRegularization',0.002, ...
    'SparsityRegularization',4, ...
    'SparsityProportion',0.1, ...
    'ScaleData', false);
    toc_2 = toc;

    feat2 = encode(autoenc2,feat1);

    % Layer 3
    tic
    softnet = trainSoftmaxLayer(feat2,tTrain,'MaxEpochs',200);
    toc_3 = toc;

    % Deep Net
    deepnet = stack(autoenc1,autoenc2,softnet);

    % Test deep net
    imageWidth = 28;
    imageHeight = 28;
    inputSize = imageWidth*imageHeight;
    %[xTestImages, tTest] = digittest_dataset;
    load('digittest_dataset');
    xTest = zeros(inputSize,numel(xTestImages));
    for i = 1:numel(xTestImages)
    xTest(:,i) = xTestImages{i}(:);
    end
    y = deepnet(xTest);

    %
    accuracy_values_nofinetuning(1,j) = 100*(1-confusion(tTest,y));
    %
    % Speed of the deep net without fine tuning
    speed_values_nofinetuning(1,j) = toc_1 + toc_2 + toc_3;

    % Test fine-tuned deep net
    xTrain = zeros(inputSize,numel(xTrainImages));
    for i = 1:numel(xTrainImages)
        xTrain(:,i) = xTrainImages{i}(:);
    end
    tic
    deepnet = train(deepnet,xTrain,tTrain);
    toc_4 = toc;
    y = deepnet(xTest);
    %
    accuracy_values_finetuned(1,j) = 100*(1-confusion(tTest,y));
```

```
        %
        speed_values_finetuned(1,j) = toc_1 + toc_2 + toc_3 + toc_4;

        %Compare with normal neural network (1 hidden layers)
        net = patternnet(n1);
        net.trainParam.epochs = 200;
        tic
        net=train(net,xTrain,tTrain);
        speed_values_pattern1(1,j) = toc;
        y=net(xTest);
        plotconfusion(tTest,y);
        accuracy_values_pattern1(1,j) = 100*(1-confusion(tTest,y));

        % Compare with normal neural network (2 hidden layers)
        net = patternnet([n1 50]);
        net.trainParam.epochs = 200;
        tic
        net=train(net,xTrain,tTrain);
        speed_values_pattern2(1,j) = toc;
        y=net(xTest);
        plotconfusion(tTest,y);
        accuracy_values_pattern2(1,j) = 100*(1-confusion(tTest,y));

    end

    speed_results(1,n) = mean(speed_values_nofinetuning);
    speed_results(2,n) = mean(speed_values_finetuned);
    speed_results(3,n) = mean(speed_values_pattern1);
    speed_results(4,n) = mean(speed_values_pattern2);

    accuracy_results(1,n) = mean(accuracy_values_nofinetuning);
    accuracy_results(2,n) = mean(accuracy_values_finetuned);
    accuracy_results(3,n) = mean(accuracy_values_pattern1);
    accuracy_results(4,n) = mean(accuracy_values_pattern2);

end

%% Plot of the speed
plot(hidden_neurons_1, speed_results(1,:) ,hidden_neurons_1, speed_results(2,:),hidden_neurons_1,
speed_results(3,:),hidden_neurons_1, speed_results(4,:))
xlabel('Number of Hidden Neurons');
ylabel('Calculation Time');
legend('y = Stacked Autoencoders without fine tuning','y = Stacked Autoencoders with fine tuning',
'y = patternnet 1 layer', 'y = patternnet 2 layers', 'Location','southwest')

%% Plot of the accuracy
plot(hidden_neurons_1, accuracy_results(1,:) ,hidden_neurons_1,
accuracy_results(2,:),hidden_neurons_1, accuracy_results(3,:),hidden_neurons_1,
accuracy_results(4,:))
xlabel('Number of Hidden Neurons');
ylabel('Accuracy');
legend('y = Stacked Autoencoders without fine tuning','y = Stacked Autoencoders with fine tuning',
'y = patternnet 1 layer', 'y = patternnet 2 layers', 'Location','southwest')
```

```
clear all
close all
nntraintool('close');
nnet.guis.closeAllViews();

% Load the training data into memory
%[xTrainImages, tTrain] = digittrain_dataset;
load('digittrain_dataset');

hidden_neurons_1 = [20, 50, 100, 150, 200];
hidden_neurons_2 = [10, 20, 50, 100, 150];

%% Multiple loops to store the results
speed_results = zeros(4,5);
accuracy_results = zeros(4,5);

%% executing deep net and normal neural networks
for n = 1:5

    speed_values_nofinetuning = zeros(1,5);
    speed_values_finetuned = zeros(1,5);
    speed_values_pattern1 = zeros(1,5);
    speed_values_pattern2 = zeros(1,5);

    accuracy_values_nofinetuning = zeros(1,5);
    accuracy_values_finetuned = zeros(1,5);
    accuracy_values_pattern1 = zeros(1,5);
    accuracy_values_pattern2 = zeros(1,5);
```

```
    for j = 1:5

        % certain amount of hidden neurons
        n2 = hidden_neurons_2(n);

        % Neural networks have weights randomly initialized before training.
        % Therefore the results from training are different each time. To avoid
        % this behavior, explicitly set the random number generator seed.
        rng('default')

        % Layer 1
        hiddenSize1 = 100;
        tic
        autoenc1 = trainAutoencoder(xTrainImages,hiddenSize1, ...
        'MaxEpochs',200, ...
        'L2WeightRegularization',0.004, ...
        'SparsityRegularization',4, ...
        'SparsityProportion',0.15, ...
        'ScaleData', false);
        toc_1 = toc;
        feat1 = encode(autoenc1,xTrainImages);


        % Layer 2
        hiddenSize2 = n2;
        tic
        autoenc2 = trainAutoencoder(feat1,hiddenSize2, ...
        'MaxEpochs',200, ...
        'L2WeightRegularization',0.002, ...
        'SparsityRegularization',4, ...
        'SparsityProportion',0.1, ...
        'ScaleData', false);
        toc_2 = toc;

        feat2 = encode(autoenc2,feat1);

        % Layer 3
        tic
        softnet = trainSoftmaxLayer(feat2,tTrain,'MaxEpochs',200);
        toc_3 = toc;

        % Deep Net
        deepnet = stack(autoenc1,autoenc2,softnet);

        % Test deep net
        imageWidth = 28;
        imageHeight = 28;
        inputSize = imageWidth*imageHeight;
        %[xTestImages, tTest] = digittest_dataset;
        load('digittest_dataset');
        xTest = zeros(inputSize,numel(xTestImages));
        for i = 1:numel(xTestImages)
        xTest(:,i) = xTestImages{i}(:);
        end
        y = deepnet(xTest);

        %
        accuracy_values_nofinetuning(1,j) = 100*(1-confusion(tTest,y));
        %
        % Speed of the deep net without fine tuning
        speed_values_nofinetuning(1,j) = toc_1 + toc_2 + toc_3;

        % Test fine-tuned deep net
        xTrain = zeros(inputSize,numel(xTrainImages));
        for i = 1:numel(xTrainImages)
            xTrain(:,i) = xTrainImages{i}(:);
        end
        tic
        deepnet = train(deepnet,xTrain,tTrain);
        toc_4 = toc;
        y = deepnet(xTest);
        %
        accuracy_values_finetuned(1,j) = 100*(1-confusion(tTest,y));
        %
        speed_values_finetuned(1,j) = toc_1 + toc_2 + toc_3 + toc_4;

        %Compare with normal neural network (1 hidden layers)
        net = patternnet(100);
        net.trainParam.epochs = 200;
        tic
        net=train(net,xTrain,tTrain);
        speed_values_pattern1(1,j) = toc;
        y=net(xTest);
        plotconfusion(tTest,y);
```

```
        accuracy_values_pattern1(1,j) = 100*(1-confusion(tTest,y));

        % Compare with normal neural network (2 hidden layers)
        net = patternnet([100 n2]);
        net.trainParam.epochs = 200;
        tic
        net=train(net,xTrain,tTrain);
        speed_values_pattern2(1,j) = toc;
        y=net(xTest);
        plotconfusion(tTest,y);
        accuracy_values_pattern2(1,j) = 100*(1-confusion(tTest,y));

    end

    speed_results(1,n) = mean(speed_values_nofinetuning);
    speed_results(2,n) = mean(speed_values_finetuned);
    speed_results(3,n) = mean(speed_values_pattern1);
    speed_results(4,n) = mean(speed_values_pattern2);

    accuracy_results(1,n) = mean(accuracy_values_nofinetuning);
    accuracy_results(2,n) = mean(accuracy_values_finetuned);
    accuracy_results(3,n) = mean(accuracy_values_pattern1);
    accuracy_results(4,n) = mean(accuracy_values_pattern2);

end

%% Plot of the speed
plot(hidden_neurons_1, speed_results(1,:) ,hidden_neurons_1, speed_results(2,:),hidden_neurons_1,
speed_results(3,:),hidden_neurons_1, speed_results(4,:))
xlabel('Number of Hidden Neurons');
ylabel('Calculation Time');
legend('y = Stacked Autoencoders without fine tuning','y = Stacked Autoencoders with fine tuning',
'y = patternnet 1 layer', 'y = patternnet 2 layers', 'Location','southwest')

%% Plot of the accuracy
plot(hidden_neurons_1, accuracy_results(1,:) ,hidden_neurons_1,
accuracy_results(2,:),hidden_neurons_1, accuracy_results(3,:),hidden_neurons_1,
accuracy_results(4,:))
xlabel('Number of Hidden Neurons');
ylabel('Accuracy');
legend('y = Stacked Autoencoders without fine tuning','y = Stacked Autoencoders with fine tuning',
'y = patternnet 1 layer', 'y = patternnet 2 layers', 'Location','southwest')
```

```
clear all
close all
nntraintool('close');
nnet.guis.closeAllViews();

% Load the training data into memory
%[xTrainImages, tTrain] = digittrain_dataset;
load('digittrain_dataset');

epochs = [50, 100, 200, 500, 1000];

%% Multiple loops to store the results
speed_results = zeros(4,5);
accuracy_results = zeros(4,5);

%% executing deep net and normal neural networks
for n = 1:5

    speed_values_nofinetuning = zeros(1,5);
    speed_values_finetuned = zeros(1,5);
    speed_values_pattern1 = zeros(1,5);
    speed_values_pattern2 = zeros(1,5);

    accuracy_values_nofinetuning = zeros(1,5);
    accuracy_values_finetuned = zeros(1,5);
    accuracy_values_pattern1 = zeros(1,5);
    accuracy_values_pattern2 = zeros(1,5);

    for j = 1:5

        % certain amount epochs
        e = epochs(n);
        % Neural networks have weights randomly initialized before training.
        % Therefore the results from training are different each time. To avoid
        % this behavior, explicitly set the random number generator seed.
        rng('default')

        % Layer 1
        hiddenSize1 = 100;
        tic
```

```
        autoenc1 = trainAutoencoder(xTrainImages,hiddenSize1, ...
        'MaxEpochs',e, ...
        'L2WeightRegularization',0.004, ...
        'SparsityRegularization',4, ...
        'SparsityProportion',0.15, ...
        'ScaleData', false);
        toc_1 = toc;
        feat1 = encode(autoenc1,xTrainImages);


        % Layer 2
        hiddenSize2 = 50;
        tic
        autoenc2 = trainAutoencoder(feat1,hiddenSize2, ...
        'MaxEpochs',e, ...
        'L2WeightRegularization',0.002, ...
        'SparsityRegularization',4, ...
        'SparsityProportion',0.1, ...
        'ScaleData', false);
        toc_2 = toc;

        feat2 = encode(autoenc2,feat1);

        % Layer 3
        tic
        softnet = trainSoftmaxLayer(feat2,tTrain,'MaxEpochs',e);
        toc_3 = toc;

        % Deep Net
        deepnet = stack(autoenc1,autoenc2,softnet);

        % Test deep net
        imageWidth = 28;
        imageHeight = 28;
        inputSize = imageWidth*imageHeight;
        %[xTestImages, tTest] = digittest_dataset;
        load('digittest_dataset');
        xTest = zeros(inputSize,numel(xTestImages));
        for i = 1:numel(xTestImages)
        xTest(:,i) = xTestImages{i}(:);
        end
        y = deepnet(xTest);

        %
        accuracy_values_nofinetuning(1,j) = 100*(1-confusion(tTest,y));
        %
        % Speed of the deep net without fine tuning
        speed_values_nofinetuning(1,j) = toc_1 + toc_2 + toc_3;

        % Test fine-tuned deep net
        xTrain = zeros(inputSize,numel(xTrainImages));
        for i = 1:numel(xTrainImages)
            xTrain(:,i) = xTrainImages{i}(:);
        end
        tic
        deepnet = train(deepnet,xTrain,tTrain);
        toc_4 = toc;
        y = deepnet(xTest);
        %
        accuracy_values_finetuned(1,j) = 100*(1-confusion(tTest,y));
        %
        speed_values_finetuned(1,j) = toc_1 + toc_2 + toc_3 + toc_4;

        %Compare with normal neural network (1 hidden layers)
        net = patternnet(100);
        net.trainParam.epochs = e;
        tic
        net=train(net,xTrain,tTrain);
        speed_values_pattern1(1,j) = toc;
        y=net(xTest);
        plotconfusion(tTest,y);
        accuracy_values_pattern1(1,j) = 100*(1-confusion(tTest,y));

        % Compare with normal neural network (2 hidden layers)
        net = patternnet([100 50]);
        net.trainParam.epochs = e;
        tic
        net=train(net,xTrain,tTrain);
        speed_values_pattern2(1,j) = toc;
        y=net(xTest);
        plotconfusion(tTest,y);
        accuracy_values_pattern2(1,j) = 100*(1-confusion(tTest,y));

    end
```

```
    speed_results(1,n) = mean(speed_values_nofinetuning);
    speed_results(2,n) = mean(speed_values_finetuned);
    speed_results(3,n) = mean(speed_values_pattern1);
    speed_results(4,n) = mean(speed_values_pattern2);

    accuracy_results(1,n) = mean(accuracy_values_nofinetuning);
    accuracy_results(2,n) = mean(accuracy_values_finetuned);
    accuracy_results(3,n) = mean(accuracy_values_pattern1);
    accuracy_results(4,n) = mean(accuracy_values_pattern2);

end

%% Plot of the speed
plot(epochs, speed_results(1,:) ,epochs, speed_results(2,:),epochs, speed_results(3,:),epochs,
speed_results(4,:))
xlabel('Epochs');
ylabel('Calculation Time');
legend('y = Stacked Autoencoders without fine tuning','y = Stacked Autoencoders with fine tuning',
'y = patternnet 1 layer', 'y = patternnet 2 layers', 'Location','northwest')

%% Plot of the accuracy
plot(epochs, accuracy_results(1,:) ,epochs, accuracy_results(2,:),epochs,
accuracy_results(3,:),epochs, accuracy_results(4,:))
xlabel('Epochs');
ylabel('Accuracy');
legend('y = Stacked Autoencoders without fine tuning','y = Stacked Autoencoders with fine tuning',
'y = patternnet 1 layer', 'y = patternnet 2 layers', 'Location','southwest')
```

6.4.2. Answers to questions in section 2.2 of the exercise

```
%% Image Category Classification Using Deep Learning
% This example shows how to use a pre-trained Convolutional Neural Network
% (CNN) as a feature extractor for training an image category classifier.
%
% Copyright 2016 The MathWorks, Inc.

%% Overview
% A Convolutional Neural Network (CNN) is a powerful machine learning
% technique from the field of deep learning. CNNs are trained using large
% collections of diverse images. From these large collections, CNNs can
% learn rich feature representations for a wide range of images. These
% feature representations often outperform hand-crafted features such as
% HOG, LBP, or SURF. An easy way to leverage the power of CNNs, without
% investing time and effort into training, is to use a pre-trained CNN as a
% feature extractor.
%
% In this example, images from Caltech 101 are classified into categories
% using a multiclass linear SVM trained with CNN features extracted from
% the images. This approach to image category classification follows the
% standard practice of training an off-the-shelf classifier using features
% extracted from images. For example, the
% <matlab:showdemo('ImageCategoryClassificationExample') Image Category
% Classification Using Bag Of Features> example uses SURF features within a
% bag of features framework to train a multiclass SVM. The difference here
% is that instead of using image features such as HOG or SURF, features are
% extracted using a CNN. And, as this example will show, the classifier
% trained using CNN features provides close to 100% accuracy, which
% is higher than the accuracy achieved using bag of features and SURF.
%
% Note: This example requires Neural Network Toolbox(TM), Parallel
% Computing Toolbox(TM), Statistics and Machine Learning Toolbox(TM), and a
% CUDA-capable GPU card.

%% Download Image Data
% The category classifier will be trained on images from
% <http://www.vision.caltech.edu/Image_Datasets/Caltech101 Caltech 101>.
% Caltech 101 is one of the most widely cited and used image data sets,
% collected by Fei-Fei Li, Marco Andreetto, and Marc 'Aurelio Ranzato.

% Download the compressed data set from the following location
url = 'http://www.vision.caltech.edu/Image_Datasets/Caltech101/101_ObjectCategories.tar.gz';
% Store the output in a temporary folder
outputFolder = fullfile(tempdir, 'caltech101'); % define output folder

%%
% Note: Download time of the data depends on your internet connection. The
% next set of commands use MATLAB to download the data and will block
% MATLAB. Alternatively, you can use your web browser to first download the
% dataset to your local disk. To use the file you downloaded from the web,
% change the 'outputFolder' variable above to the location of the
% downloaded file.

if ~exist(outputFolder, 'dir') % download only once
```

```
    disp('Downloading 126MB Caltech101 data set...');
    untar(url, outputFolder);
end

%% Load Images
% Instead of operating on all of Caltech 101, which is time consuming, use
% three of the categories: airplanes, ferry, and laptop. The image category
% classifier will be trained to distinguish amongst these six categories.

rootFolder = fullfile(outputFolder, '101_ObjectCategories');
categories = {'airplanes', 'ferry', 'laptop'};

%%
% Create an |ImageDatastore| to help you manage the data. Because
% |ImageDatastore| operates on image file locations, images are not loaded
% into memory until read, making it efficient for use with large image
% collections.
imds = imageDatastore(fullfile(rootFolder, categories), 'LabelSource', 'foldernames');

%%
% The |imds| variable now contains the images and the category labels
% associated with each image. The labels are automatically assigned from
% the folder names of the image files. Use |countEachLabel| to summarize
% the number of images per category.
tbl = countEachLabel(imds)
%%
% Because |imds| above contains an unequal number of images per category,
% let's first adjust it, so that the number of images in the training set
% is balanced.

minSetCount = min(tbl{:,2}); % determine the smallest amount of images in a category

% Use splitEachLabel method to trim the set.
imds = splitEachLabel(imds, minSetCount, 'randomize');

% Notice that each set now has exactly the same number of images.
countEachLabel(imds)

%%
% Below, you can see example images from three of the categories included
% in the dataset.

% Find the first instance of an image for each category
airplanes = find(imds.Labels == 'airplanes', 1);
ferry = find(imds.Labels == 'ferry', 1);
laptop = find(imds.Labels == 'laptop', 1);

figure
subplot(1,3,1);
imshow(imds.Files{airplanes})
subplot(1,3,2);
imshow(imds.Files{ferry})
subplot(1,3,3);
imshow(imds.Files{laptop})

%% Download Pre-trained Convolutional Neural Network (CNN)
% Now that the images are prepared, you will need to download a pre-trained
% CNN model for this example. There are several pre-trained networks that
% have gained popularity. Most of these have been trained on the ImageNet
% dataset, which has 1000 object categories and 1.2 million training
% images[1]. "AlexNet" is one such model and can be downloaded from
% MatConvNet[2,3]:

% Location of pre-trained "AlexNet"
cnnURL = 'http://www.vlfeat.org/matconvnet/models/beta16/imagenet-caffe-alex.mat';
% Store CNN model in a temporary folder
cnnMatFile = fullfile(tempdir, 'imagenet-caffe-alex.mat');

%%
% Note: Download time of the data depends on your internet connection.  The
% next set of commands use MATLAB to download the data and will block
% MATLAB. Alternatively, you can use your web browser to first download the
% dataset to your local disk. To use the file you downloaded from the web,
% change the 'cnnMatFile' variable above to the location of the downloaded
% file.
if ~exist(cnnMatFile, 'file') % download only once
    disp('Downloading pre-trained CNN model...');
    websave(cnnMatFile, cnnURL);
end

%% Load Pre-trained CNN
% The CNN model is saved in MatConvNet's format [3]. Load the MatConvNet
% network data into |convnet|, a |SeriesNetwork| object from Neural Network
% Toolbox(TM), using the helper function |helperImportMatConvNet|. A
```

```
% SeriesNetwork object can be used to inspect the network architecture,
% classify new data, and extract network activations from specific layers.

% Load MatConvNet network into a SeriesNetwork
convnet = helperImportMatConvNet(cnnMatFile)

%%
% |convnet.Layers| defines the architecture of the CNN.

% View the CNN architecture
convnet.Layers

%%
% The first layer defines the input dimensions. Each CNN has a different
% input size requirements. The one used in this example requires image
% input that is 227-by-227-by-3.

% Inspect the first layer
convnet.Layers(1)

%%
% The intermediate layers make up the bulk of the CNN. These are a series
% of convolutional layers, interspersed with rectified linear units (ReLU)
% and max-pooling layers [2]. Following the these layers are 3
% fully-connected layers.
%
% The final layer is the classification layer and its properties depend on
% the classification task. In this example, the CNN model that was loaded
% was trained to solve a 1000-way classification problem. Thus the
% classification layer has 1000 classes from the ImageNet dataset.

% Inspect the last layer
convnet.Layers(end)

% Number of class names for ImageNet classification task
numel(convnet.Layers(end).ClassNames)

%%
% Note that the CNN model is not going to be used for the original
% classification task. It is going to be re-purposed to solve a different
% classification task on the Caltech 101 dataset.

%% Pre-process Images For CNN
% As mentioned above, |convnet| can only process RGB images that are
% 227-by-227. To avoid re-saving all the images in Caltech 101 to this
% format, setup the |imds| read function, |imds.ReadFcn|, to pre-process
% images on-the-fly. The |imds.ReadFcn| is called every time an image is
% read from the |ImageDatastore|.

% Set the ImageDatastore ReadFcn
imds.ReadFcn = @(filename)readAndPreprocessImage(filename);

%% Prepare Training and Test Image Sets
% Split the sets into training and validation data. Pick 30% of images
% from each set for the training data and the remainder, 70%, for the
% validation data. Randomize the split to avoid biasing the results. The
% training and test sets will be processed by the CNN model.

[trainingSet, testSet] = splitEachLabel(imds, 0.3, 'randomize');

%% Extract Training Features Using CNN
% Each layer of a CNN produces a response, or activation, to an input
% image. However, there are only a few layers within a CNN that are
% suitable for image feature extraction. The layers at the beginning of the
% network capture basic image features, such as edges and blobs. To see
% this, visualize the network filter weights from the first convolutional
% layer. This can help build up an intuition as to why the features
% extracted from CNNs work so well for image recognition tasks. Note that
% visualizing deeper layer weights is beyond the scope of this example. You
% can read more about that in the work of Zeiler and Fergus [4].

% Get the network weights for the second convolutional layer
w1 = convnet.Layers(2).Weights;

% Scale and resize the weights for visualization
w1 = mat2gray(w1);
w1 = imresize(w1,5);

% Display a montage of network weights. There are 96 individual sets of
% weights in the first layer.
figure
montage(w1)
title('First convolutional layer weights')
```

```
% %%
% % Notice how the first layer of the network has learned filters for
% % capturing blob and edge features. These "primitive" features are then
% % processed by deeper network layers, which combine the early features to
% % form higher level image features. These higher level features are better
% % suited for recognition tasks because they combine all the primitive
% % features into a richer image representation [5].
% %
% % You can easily extract features from one of the deeper layers using the
% % |activations| method. Selecting which of the deep layers to choose is a
% % design choice, but typically starting with the layer right before the
% % classification layer is a good place to start. In |convnet|, the this
% % layer is named 'fc7'. Let's extract training features using that layer.
% featureLayer = 'fc7';
% trainingFeatures = activations(convnet, trainingSet, featureLayer, ...
%     'MiniBatchSize', 32, 'OutputAs', 'columns');
% %%
% % Note that the activations are computed on the GPU and the 'MiniBatchSize'
% % is set 32 to ensure that the CNN and image data fit into GPU memory.
% % You may need to lower the 'MiniBatchSize' if your GPU runs out of memory.
% %
% % Also, the activations output is arranged as columns. This helps speed-up
% % the multiclass linear SVM training that follows.
%
% %% Train A Multiclass SVM Classifier Using CNN Features
% % Next, use the CNN image features to train a multiclass SVM classifier. A
% % fast Stochastic Gradient Descent solver is used for training by setting
% % the |fitcecoc| function's 'Learners' parameter to 'Linear'. This helps
% % speed-up the training when working with high-dimensional CNN feature
% % vectors, which each have a length of 4096.
%
% % Get training labels from the trainingSet
% trainingLabels = trainingSet.Labels;
%
% % Train multiclass SVM classifier using a fast linear solver, and set
% % 'ObservationsIn' to 'columns' to match the arrangement used for training
% % features.
% classifier = fitcecoc(trainingFeatures, trainingLabels, ...
%     'Learners', 'Linear', 'Coding', 'onevsall', 'ObservationsIn', 'columns');
%
% %% Evaluate Classifier
% % Repeat the procedure used earlier to extract image features from
% % |testSet|. The test features can then be passed to the classifier to
% % measure the accuracy of the trained classifier.
%
% % Extract test features using the CNN
% testFeatures = activations(convnet, testSet, featureLayer, 'MiniBatchSize',32);
%
% % Pass CNN image features to trained classifier
% predictedLabels = predict(classifier, testFeatures);
%
% % Get the known labels
% testLabels = testSet.Labels;
%
% % Tabulate the results using a confusion matrix.
% confMat = confusionmat(testLabels, predictedLabels);
%
% % Convert confusion matrix into percentage form
% confMat = bsxfun(@rdivide,confMat,sum(confMat,2))
% %%
%
% % Display the mean accuracy
% mean(diag(confMat))
%
% %% Try the Newly Trained Classifier on Test Images
% % You can now apply the newly trained classifier to categorize new images.
% newImage = fullfile(rootFolder, 'airplanes', 'image_0690.jpg');
%
% % Pre-process the images as required for the CNN
% img = readAndPreprocessImage(newImage);
%
% % Extract image features using the CNN
% imageFeatures = activations(convnet, img, featureLayer);
% %%
%
% % Make a prediction using the classifier
% label = predict(classifier, imageFeatures)
%
%
% %% References
% % [1] Deng, Jia, et al. "Imagenet: A large-scale hierarchical image
% % database." Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE
% % Conference on. IEEE, 2009.
% %
```

```
% % [2] Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet
% % classification with deep convolutional neural networks." Advances in
% % neural information processing systems. 2012.
% %
% % [3] Vedaldi, Andrea, and Karel Lenc. "MatConvNet-convolutional neural
% % networks for MATLAB." arXiv preprint arXiv:1412.4564 (2014).
% %
% % [4] Zeiler, Matthew D., and Rob Fergus. "Visualizing and understanding
% % convolutional networks." Computer Vision-ECCV 2014. Springer
% % International Publishing, 2014. 818-833.
% %
% % [5] Donahue, Jeff, et al. "Decaf: A deep convolutional activation feature
% % for generic visual recognition." arXiv preprint arXiv:1310.1531 (2013).
%
% displayEndOfDemoMessage(mfilename)
```

6.5. Final Project

6.5.1. Problem 1: nonlinear regression with MLP's

```
%% loading the data
load('Data_Problem1_regression.mat')

% the new target variable
Y = (8*T1 + 5*T2 + 4*T3 + 3*T3 + 3*T5) / (8+5+4+3+3);

%% creating the training, validation and test sets
rng('default');
index = randperm(size(X1, 1),3000);
% deriving 3000 points, which can be divided between the training,
% validation and test set
X = [X1 X2]';
Y = Y';

% creating the right 3000 X and Y values
X = X(:,index);
Y = Y(index);

%%
% number of epochs -> 1000
epochs = 1000;

% transfer function for the hidden layer -> ("logsig" , "tansig")
transfer_functions = {'logsig', 'tansig'};

% number of neurons
number_neurons = [5, 10, 20, 50, 100];

% different training algorithms
training_algs = { 'traingd', 'traingda', 'traincgf', 'traincgp', 'trainbfg', 'trainlm', 'trainbr'};

% number of iterarions for each execution
n_iterations = 10;

%% creating the barplots

figure
subplot_index = 0;

for t = transfer_functions

        subplot_index = subplot_index + 1;

        transfer_function = char(t);

        % matrix to store speed values in -> speed_values
        speed_values = zeros(length(training_algs), length(number_neurons));
        RMSE_val_average_error = zeros(length(training_algs), length(number_neurons));

        % index for speed_values matrix
        i = 0;
        j= 0;

        for train_alg = training_algs

            % updating row index that corresponds to the training algorithm
            i  = i + 1;

            for n_number = number_neurons

                % training algorithm
                train_alg = char(train_alg);
```

```
            % array to store the calculation times over the 20 iterations
            calc_time_values = zeros(1,n_iterations);

            % array to store the RMSE test errors over the 20 iterations
            RMSE_val_values = zeros(1,n_iterations);

            for it = 1:n_iterations

                % the forwardfeed neural network with 1/3 training, 1/3
                % validation, 1/3 testset
                net = feedforwardnet(n_number, train_alg);
                net.trainParam.epochs = epochs;
                net.divideParam.trainRatio = 1/3;
                net.divideParam.valRatio = 1/3;
                net.divideParam.testRatio = 1/3;

                % the transfer function
                net.layers{1}.transferFcn = char(transfer_function);
                % !! THE OUTPUT LAYER WILL ALWAYS HAVE THE LINEAR TRANSFER
                % FUNCTION AS IS NECASSARY FOR REGRESSION

                % training the neural network and measuring average calculation time
                tic;
                [net, tr] = train(net, X, Y);
                calc_time = toc;

                X_train = X(:, tr.trainInd);
                Y_train = Y(:, tr.trainInd);

                X_val = X(:, tr.testInd);
                Y_val = Y(:, tr.testInd);

                % the calculation time values over 10 iterations
                calc_time_values(it) = calc_time;

                % RMSE validation set
                RMSE_val_values(it) = sqrt(sum((Y_val - sim(net, X_val)).^2));

            end

            % average calculation time
            average_calc_time = mean(calc_time_values);

            % average RMSE val error
            average_RMSE_val = mean(RMSE_val_values);

            % updating the column index that corresponds to certain amount of
            % hidden neurons
            j = j + 1;

            % adding the calculation time values to the matrix
            speed_values(i,j) = average_calc_time;

            % adding the RMSE test error to the matrix
            RMSE_val_average_error(i,j) = average_RMSE_val;

        end

        j = 0;


    end

    % Bar Plot Speed
    % returning a barplot for the Speed
    subplot(1,4,subplot_index)
    names = categorical(training_algs);
    barplot = bar(names, speed_values);
    ylabel("Calculation Time")
    legend_bar = legend(barplot,{"5", "10", "20", "50", "100"}, 'Location', 'northwest');
    title(legend_bar, "Hidden Neurons");

    subplot_index = subplot_index + 1;

    subplot(1,4,subplot_index)
    % Bar Plot Validation RMSE
    % returning a barplot for the RMSE Test Error
    names = categorical(training_algs);
    barplot = bar(names, RMSE_val_average_error);
    ylabel("RMSE Validation Error")
    legend_bar = legend(barplot,{"5", "10", "20", "50", "100"}, 'Location', 'northwest');
    title(legend_bar, "Hidden Neurons");
```

```
end
```

```
%% loading the data
load('Data_Problem1_regression.mat')

% the new target variable
Y = (8*T1 + 5*T2 + 4*T3 + 3*T3 + 3*T5) / (8+5+4+3+3);

%% Data set
% test and training set together should contain 2000 points
rng('default');
index = randperm(size(X1, 1),3000);
X = [X1 X2]';
Y = Y';

X = X(:,index);
Y = Y(index);
%% Feedforward neural network with trainbr as training algorithm and 50 hidden neurons
rng('default');
% number of epochs
n_epochs = 1000;

% the forwardfeed neural network with 1/3 training, 1/3 validation, 1/3 testset
net = feedforwardnet(50, 'trainbr');
net.trainParam.epochs = n_epochs;
net.divideParam.trainRatio = 1/3;
net.divideParam.valRatio = 1/3;
net.divideParam.testRatio = 1/3;

% tansig transfer function
net.layers{1}.transferFcn = 'tansig';
[net, tr] = train(net, X, Y);

X_train = X(:, tr.trainInd);
Y_train = Y(:, tr.trainInd);

X_test = X(:, tr.testInd);
Y_test = Y(:, tr.testInd);

% MSE training error
MSE_training_error = mean((Y_train - sim(net, X_train)).^2)

% MSE test error
MSE_test_error = mean((Y_test - sim(net, X_test)).^2)

% corr = corrcoef(T_test, T_test_sim);
corr = corrcoef(Y_test, sim(net, X_test));
% R squared
Rsquared = (corr(1,2))^2

%% scatter plot for the real and predicted Y values of the test set
% first putting predicted y values in variables Y_test_predicted
Y_test_predicted = sim(net, X_test);
X_1_test = X_test(1,:);
X_2_test = X_test(2,:);

%% 3D scatter predicted
figure(1)
scatter3(X_1_test, X_2_test, Y_test, 'filled', 'MarkerEdgeColor','k')
xlabel('X1'), ylabel('X2'), zlabel('F(X1, X2)')
%% 3D scatter target, real values
figure(2)
scatter3(X_1_test, X_2_test, Y_test_predicted,'filled', 'r', 'MarkerEdgeColor','k')
xlabel('X1'), ylabel('X2'), zlabel('Predicted Values')
```

6.5.2. Problem 1: classification with MLP's

```
%% loading the data
data = readtable('winequality-white.csv');
wines_1 = data(data.quality == 5,:);
wines_2 = data(data.quality == 6,:);
wines_3 = data(data.quality == 7,:);

% creating the right frame in order to be able to perform the patternnet
wines = [wines_1; wines_2; wines_3];
wines.class1 = wines.quality == 5;
wines.class2 = (wines.quality == 6 | wines.quality == 7);
wines = table2array(wines);
X = wines(:, 1:end-3)';
Y = wines(:, end-1:end)';

%% PCA reduction of the input variables
% rows are the observations, columns the variables
X = X';
```

```
cov_matrix = cov(X);
% amount of dimensions

% determining sum of all eigenvalues
n_dimensions = size(cov_matrix,1);
[all_v, all_d] = eigs(cov_matrix,n_dimensions);
total_eigen_values = sum(diag(all_d));

%% Eigenvalue decomposition
x_plot = 1:11;
proportion_values = zeros(1,11);
error_values = zeros(1,11);

% for all values k = 1 - 11
for k = 1:11

    % returning the eigenvectors of k largest eigenvalues
    [v,d] = eigs(cov_matrix,k);

    % PROPORTION EXPLAINED
    % proportion explained by k principal components -> eigenvalues
    proportion_eigen_values = sum(diag(d));
    % proportion of variance explained
    proportion_value = proportion_eigen_values / total_eigen_values;

    % RMSE ERROR OF RECONSTRUCTION MATRIX
    transformation = v;
    % transforming the original data with k principal components as dimensions
    x_transformed = X * transformation;
    % try to reconstruct the original data_matrix
    x_estimated = x_transformed * (transformation');
    RMSE_error = sqrt(mean(mean((X - x_estimated).^2)));

    % adding values to arrays
    proportion_values(k) = proportion_value;
    error_values(k) = RMSE_error;

end

%% Plot proportion variance explained
graph_plot = plot(x_plot, proportion_values);
graph_plot.LineWidth = 2;
xlabel("Number Principal Components");
ylabel("Proportion Variance Explained");

%% Reduced dataset with three principal components
% 3 is a good number of principal components
% returning the eigenvectors of 3 largest eigenvalues
[v,d] = eigs(cov_matrix,3);

% the reduced data set with the three principal components
transformation = v;
% reduced data set with three principal components
% transforming the original data with k principal components as dimensions
X_transformed = X * transformation;
X_transformed = X_transformed';

%% Network architecture for three principal components
% number of epochs -> 1000
epochs = 1000;

% transfer function for the hidden layer -> ("logsig" , "tansig")
transfer_functions = {'logsig', 'tansig'};

% number of neurons
number_neurons = [5, 10, 20, 50, 100];

% different training algorithms
training_algs = { 'traingd', 'traingda', 'traincgf', 'traincgp', 'trainbfg', 'trainlm', 'trainbr'};

% number of iterarions for each execution
n_iterations = 1;

% creating the barplots
subplot_index = 0;
%%
figure;
for t = transfer_functions

        subplot_index = subplot_index + 1;

        transfer_function = char(t);

        % matrix to store speed values in -> speed_values
```

```
speed_values = zeros(length(training_algs), length(number_neurons));
CCR_val_average_error = zeros(length(training_algs), length(number_neurons));

% index for speed_values matrix
i = 0;
j= 0;

for train_alg = training_algs

    % updating row index that corresponds to the training algorithm
    i  = i + 1;

    for n_number = number_neurons

        % training algorithm
        train_alg = char(train_alg);

        % array to store the calculation times over the 10 iterations
        calc_time_values = zeros(1,n_iterations);

        % array to store the RMSE test errors over the 10 iterations
        CCR_val_values = zeros(1,n_iterations);

        for it = 1:n_iterations

            net = patternnet(n_number, train_alg);
            net.trainParam.epochs = epochs;

            % the transfer function
            net.layers{1}.transferFcn = char(t);
            % output layer will be equal to softmax transfer
            % function

            % training the neural network and measuring average calculation time
            tic;
            [net, tr] = train(net, X_transformed, Y);
            calc_time = toc;

            X_transformed_val = X_transformed(:, tr.testInd);
            Y_val = Y(:, tr.testInd);

            % the calculation time values over 10 iterations
            calc_time_values(it) = calc_time;

            % correct classification ratio
            Y_val_sim = sim(net, X_transformed_val);
            [conf,~,~,~] = confusion(Y_val, Y_val_sim);
            CCR_val_values(it) = 100*(1-conf);


        end

        % average calculation time
        average_calc_time = mean(calc_time_values);

        % average RMSE val error
        average_CCR_val = mean(CCR_val_values);

        % updating the column index that corresponds to certain amount of
        % hidden neurons
        j = j + 1;

        % adding the calculation time values to the matrix
        speed_values(i,j) = average_calc_time;

        % adding the Validation CCR to the matrix
        CCR_val_average_error(i,j) = average_CCR_val;

    end

    j = 0;


end

% Bar Plot Speed
% returning a barplot for the Speed
subplot(1,4,subplot_index)
names = categorical(training_algs);
barplot = bar(names, speed_values);
ylabel("Calculation Time")
legend_bar = legend(barplot,{"5", "10", "20", "50", "100"}, 'Location', 'northeast');
title(legend_bar, "Hidden Neurons");
```

```
        subplot_index = subplot_index + 1;

        subplot(1,4,subplot_index)
        % Bar Plot Validation CCR
        % returning a barplot for the CCR
        names = categorical(training_algs);
        barplot = bar(names, CCR_val_average_error);
        ylabel("CCR Validation Error")
        legend_bar = legend(barplot,{"5", "10", "20", "50", "100"}, 'Location', 'southwest');
        title(legend_bar, "Hidden Neurons");

end
```

```
%% loading the data
data = readtable('winequality-white.csv');
wines_1 = data(data.quality == 5,:);
wines_2 = data(data.quality == 6,:);
wines_3 = data(data.quality == 7,:);

% creating the right frame in order to be able to perform the patternnet
wines = [wines_1; wines_2; wines_3];
wines.class1 = wines.quality == 5;
wines.class2 = (wines.quality == 6 | wines.quality == 7);
wines = table2array(wines);
X = wines(:, 1:end-3)';
Y = wines(:, end-1:end)';

%%
% number of epochs -> 1000
epochs = 1000;

% transfer function for the hidden layer -> ("logsig" , "tansig")
transfer_functions = {'logsig', 'tansig'};

% number of neurons
number_neurons = [5, 10, 20, 50, 100];

% different training algorithms
training_algs = { 'traingd', 'traingda', 'traincgf', 'traincgp', 'trainbfg', 'trainlm', 'trainbr'};

% number of iterarions for each execution
n_iterations = 10;

%% creating the barplots
figure
subplot_index = 0;

for t = transfer_functions

        subplot_index = subplot_index + 1;

        transfer_function = char(t);

        % matrix to store speed values in -> speed_values
        speed_values = zeros(length(training_algs), length(number_neurons));
        CCR_val_average_error = zeros(length(training_algs), length(number_neurons));

        % index for speed_values matrix
        i = 0;
        j= 0;

        for train_alg = training_algs

            % updating row index that corresponds to the training algorithm
            i  = i + 1;

            for n_number = number_neurons

                % training algorithm
                train_alg = char(train_alg);

                % array to store the calculation times over the 10 iterations
                calc_time_values = zeros(1,n_iterations);

                % array to store the RMSE test errors over the 10 iterations
                CCR_val_values = zeros(1,n_iterations);

                for it = 1:n_iterations

                    net = patternnet(n_number, train_alg);
                    net.trainParam.epochs = epochs;

                    % the transfer function
```

```
                    net.layers{1}.transferFcn = char(t);
                    % output layer will be equal to softmax transfer
                    % function

                    % training the neural network and measuring average calculation time
                    tic;
                    [net, tr] = train(net, X, Y);
                    calc_time = toc;

                    X_val = X(:, tr.testInd);
                    Y_val = Y(:, tr.testInd);

                    % the calculation time values over 10 iterations
                    calc_time_values(it) = calc_time;

                    % correct classification ratio
                    Y_val_sim = sim(net, X_val);
                    [conf,~,~,~] = confusion(Y_val, Y_val_sim);
                    CCR_val_values(it) = 100*(1-conf);


                end

                % average calculation time
                average_calc_time = mean(calc_time_values);

                % average RMSE val error
                average_CCR_val = mean(CCR_val_values);

                % updating the column index that corresponds to certain amount of
                % hidden neurons
                j = j + 1;

                % adding the calculation time values to the matrix
                speed_values(i,j) = average_calc_time;

                % adding the Validation CCR to the matrix
                CCR_val_average_error(i,j) = average_CCR_val;

            end

            j = 0;


        end

        % Bar Plot Speed
        % returning a barplot for the Speed
        subplot(1,4,subplot_index)
        names = categorical(training_algs);
        barplot = bar(names, speed_values);
        ylabel("Calculation Time")
        legend_bar = legend(barplot,{"5", "10", "20", "50", "100"}, 'Location', 'northeast');
        title(legend_bar, "Hidden Neurons");

        subplot_index = subplot_index + 1;

        subplot(1,4,subplot_index)
        % Bar Plot Validation CCR
        % returning a barplot for the CCR
        names = categorical(training_algs);
        barplot = bar(names, CCR_val_average_error);
        ylabel("CCR Validation Error")
        legend_bar = legend(barplot,{"5", "10", "20", "50", "100"}, 'Location', 'southwest');
        title(legend_bar, "Hidden Neurons");


end
```

6.5.3. Problem 2: character recognition with Hopfield networks

```
% creating dataset of letters
capital_letters = prprob();

% creating the lowercase letters of 'simondelac'
letter_s =  [0 0 0 0 0 ...
             0 0 0 0 0 ...
             0 1 1 1 0 ...
             1 0 0 0 0 ...
             0 1 1 1 0 ...
             0 0 0 0 1 ...
             0 1 1 1 0 ]';
```

```
letter_i =  [0 0 0 0 0 ...
             0 0 0 0 0 ...
             0 0 1 0 0 ...
             0 0 0 0 0 ...
             0 0 1 0 0 ...
             0 0 1 0 0 ...
             0 0 1 0 0 ]';

letter_m =  [0 0 0 0 0 ...
             0 0 0 0 0 ...
             1 1 0 1 1 ...
             1 0 1 0 1 ...
             1 0 1 0 1 ...
             1 0 1 0 1 ...
             1 0 1 0 1 ]';

letter_o =  [0 0 0 0 0 ...
             0 0 0 0 0 ...
             0 1 1 1 0 ...
             1 0 0 0 1 ...
             1 0 0 0 1 ...
             1 0 0 0 1 ...
             0 1 1 1 0 ]';

letter_n =  [0 0 0 0 0 ...
             0 0 0 0 0 ...
             1 1 1 0 0 ...
             1 0 0 1 0 ...
             1 0 0 1 0 ...
             1 0 0 1 0 ...
             1 0 0 1 0 ]';

letter_d =  [0 0 0 0 0 ...
             0 0 0 0 0 ...
             0 0 0 1 0 ...
             0 1 1 1 0 ...
             1 0 0 1 0 ...
             1 0 0 1 0 ...
             0 1 1 1 0 ]';

letter_e =  [0 0 0 0 0 ...
             0 0 0 0 0 ...
             0 1 1 0 0 ...
             1 0 0 1 0 ...
             1 1 1 1 0 ...
             1 0 0 0 0 ...
             0 1 1 1 0 ]';

letter_l =  [0 0 0 0 0 ...
             0 0 0 0 0 ...
             0 1 0 0 0 ...
             0 1 0 0 0 ...
             0 1 0 0 0 ...
             0 1 0 0 0 ...
             0 1 1 0 0 ]';

letter_a =  [0 0 0 0 0 ...
             0 0 0 0 0 ...
             0 1 1 0 0 ...
             1 0 0 1 0 ...
             1 0 0 1 0 ...
             1 0 0 1 0 ...
             0 1 1 0 1 ]';

letter_c =  [0 0 0 0 0 ...
             0 0 0 0 0 ...
             0 1 1 1 0 ...
             1 0 0 0 0 ...
             1 0 0 0 0 ...
             1 0 0 0 0 ...
             0 1 1 1 0 ]';

% creating the dataset with all letters
letters = [letter_s, letter_i, letter_m, letter_o, letter_n, letter_d, letter_e, letter_l, letter_a,
letter_c, capital_letters];

%% image of the ten first letters
subplot_index = 0;
for i = 1:10
    letter = letters(:,i);

    colormap('gray');
    subplot_index = subplot_index + 1;
```

```
    subplot(1,10,subplot_index);
    imagesc(reshape(letter',5,7)')
    axis off
end
%% Setting the zeros equal to -1, in order for the Hopfield network to work
letters(letters == 0) = -1;

%% Hopfield network to retrieve the first five characters
% the data of first five letters
first_5 = letters(:, 1:5);

% creating the Hopfield network
net = newhop(first_5);

%% randomly distorting 3 pixels of the first five characters as input
first_5_distorted = first_5;
% array to store the amount of correct classifications
right_classifications = zeros(1,5);
wrong_classifications = zeros(1,5);

time_steps = 100;

for iter = 1:100
    for i = 1:5

        % determine index of the three random components that we will change
        first_index = round(randi([1 35],1));
        second_index = round(randi([1 35],1));
        third_index = round(randi([1 35],1));
        indices = [first_index, second_index, third_index];

        for in = indices

            % inversing the pixel value
            if(first_5_distorted(in,i) == 1)
                first_5_distorted(in,i) = -1;
            else
                first_5_distorted(in,i) = 1;
            end
        end
    end

    for k = 1:5

        letter_input = first_5_distorted(:,k);

        % getting the best solution from the Hopfield network for the certain
        % letter
        T = {letter_input};
        [letter_output,~,~] = sim(net,{1 ,time_steps},{},T);
        letter_output = letter_output{1, time_steps};

        % the letters to check
        letters_to_check = first_5;

        if(isequal(letter_output, letters_to_check(:,k)) == true)

            right_classifications(k) = right_classifications(k) + 1;

        end

        % plotting if spurious state
        m = 0;

        for j = 1:5

            if(isequal(letter_output, letters_to_check(:,j)) == true)

                % m be set to 1, if number is classified as spurious state
                m = 1;

            end
        end

        if(m == 0)

            % plotting the spurious state
            % hold on
            % figure;
            % colormap('gray');
            % printing the spurious state
            % imagesc(reshape(letter_output',5,7)')
            % axis off
            % title(k);
```

```
        end

    end
end

%% Barplot of the classifications
names = categorical({'s', 'i', 'm', 'o', 'n'});
names = reordercats(names,{'s', 'i', 'm', 'o', 'n'});
frequency_classifications = right_classifications / 100;
bar(names, frequency_classifications);
title("Correct Classifications");
ylabel("Number of Correct Classifications")
xlabel('Distorted letters')
```

```
% creating dataset of letters
capital_letters = prprob();

% creating the lowercase letters of 'simondelac'
letter_s =  [0 0 0 0 0 ...
             0 0 0 0 0 ...
             0 1 1 1 0 ...
             1 0 0 0 0 ...
             0 1 1 1 0 ...
             0 0 0 0 1 ...
             0 1 1 1 0 ]';

letter_i =  [0 0 0 0 0 ...
             0 0 0 0 0 ...
             0 0 1 0 0 ...
             0 0 0 0 0 ...
             0 0 1 0 0 ...
             0 0 1 0 0 ...
             0 0 1 0 0 ]';

letter_m =  [0 0 0 0 0 ...
             0 0 0 0 0 ...
             1 1 0 1 1 ...
             1 0 1 0 1 ...
             1 0 1 0 1 ...
             1 0 1 0 1 ...
             1 0 1 0 1 ]';

letter_o =  [0 0 0 0 0 ...
             0 0 0 0 0 ...
             0 1 1 1 0 ...
             1 0 0 0 1 ...
             1 0 0 0 1 ...
             1 0 0 0 1 ...
             0 1 1 1 0 ]';

letter_n =  [0 0 0 0 0 ...
             0 0 0 0 0 ...
             1 1 1 0 0 ...
             1 0 0 1 0 ...
             1 0 0 1 0 ...
             1 0 0 1 0 ...
             1 0 0 1 0 ]';

letter_d =  [0 0 0 0 0 ...
             0 0 0 0 0 ...
             0 0 0 1 0 ...
             0 1 1 1 0 ...
             1 0 0 1 0 ...
             1 0 0 1 0 ...
             0 1 1 1 0 ]';

letter_e =  [0 0 0 0 0 ...
             0 0 0 0 0 ...
             0 1 1 0 0 ...
             1 0 0 1 0 ...
             1 1 1 1 0 ...
             1 0 0 0 0 ...
             0 1 1 1 0 ]';

letter_l =  [0 0 0 0 0 ...
             0 0 0 0 0 ...
             0 1 0 0 0 ...
             0 1 0 0 0 ...
             0 1 0 0 0 ...
             0 1 0 0 0 ...
             0 1 1 0 0 ]';
```

```
letter_a =  [0 0 0 0 0 ...
             0 0 0 0 0 ...
             0 1 1 0 0 ...
             1 0 0 1 0 ...
             1 0 0 1 0 ...
             1 0 0 1 0 ...
             0 1 1 0 1 ]';

letter_c =  [0 0 0 0 0 ...
             0 0 0 0 0 ...
             0 1 1 1 0 ...
             1 0 0 0 0 ...
             1 0 0 0 0 ...
             1 0 0 0 0 ...
             0 1 1 1 0 ]';

% creating the dataset with all letters
letters = [letter_s, letter_i, letter_m, letter_o, letter_n, letter_d, letter_e, letter_l, letter_a,
letter_c, capital_letters];

%% image of the ten first letters
subplot_index = 0;
for i = 1:10
    letter = letters(:,i);

    colormap('gray');
    subplot_index = subplot_index + 1;

    subplot(1,10,subplot_index);
    imagesc(reshape(letter',5,7)')
    axis off
end
%% Setting the zeros equal to -1, in order for the Hopfield network to work
letters(letters == 0) = -1;

%% Error and storing different numbers of P
P = 36;
final_error_values = zeros(1,P);
final_correct_classifications = zeros(1,P);
time_steps = 2000;

% parameter to indicate the amount of correct classifications
for h = 1:P

    first_letters = letters(:, 1:P);

    % creating the Hopfield network for P number of first characters
    net = newhop(first_letters);

    % storing the distorted letters
    first_letters_distorted = first_letters;

    % storing error values of a number of 20 iterations
    storing_error_values = zeros(1,P);

    for iter = 1:20

        for i = 1:h

            % determine index of the three random components that we will change
            first_index = round(randi([1 35],1));
            second_index = round(randi([1 35],1));
            third_index = round(randi([1 35],1));
            indices = [first_index, second_index, third_index];

            for in = indices

                % inversing the pixel value
                if(first_letters_distorted(in,i) == 1)
                    first_letters_distorted(in,i) = -1;
                else
                    first_letters_distorted(in,i) = 1;
                end
            end
        end

        error_P_values = zeros(1,P);

        for k = 1:h

            letter_input = first_letters_distorted(:,k);

            % getting the best solution from the Hopfield network for the certain
            % letter
```

```
            T = {letter_input};
            [letter_output,~,~] = sim(net,{1 ,time_steps},{},T);
            letter_output = letter_output{1, time_steps};

            % rounding the values certainly to a 1 or -1
            for output_rounding = 1:35

                if(letter_output(output_rounding) > 0)
                    letter_output(output_rounding) = 1;
                else
                    letter_output(output_rounding) = -1;
                end
            end

            % the letters to check
            letters_to_check = first_letters;

            % calculating error
            if(isequal(letter_output, letters_to_check(:,k)) == true)

                error = 0;

            else

                error = sum(abs(letters_to_check(:,1) - letter_output))/2;

            end

            error_P_values(1,k) = error;

        end

            storing_error_values = storing_error_values + error_P_values;

    end

    final_error = median(storing_error_values/20);

    final_error_values(1,h) = final_error;

end

%% plotting error in function of P
p = plot(1:P, final_error_values,'-o');
p.LineWidth = 1.2;
xlabel('Number of characters P stored');
ylabel('"median of averages" error')
hold on; % hold the plot for other curves
plot(17,final_error_values(17),'o','MarkerSize',15);
txt = ('critical loading capacity');
t = text(17, final_error_values(17)+0.7, txt);
t.FontWeight = 'bold';
```

```
% creating dataset of letters
capital_letters = prprob();

% creating the lowercase letters of 'simondelac'
letter_s =  [0 0 0 0 0 ...
             0 0 0 0 0 ...
             0 1 1 1 0 ...
             1 0 0 0 0 ...
             0 1 1 1 0 ...
             0 0 0 0 1 ...
             0 1 1 1 0 ]';

letter_i =  [0 0 0 0 0 ...
             0 0 0 0 0 ...
             0 0 1 0 0 ...
             0 0 0 0 0 ...
             0 0 1 0 0 ...
             0 0 1 0 0 ...
             0 0 1 0 0 ]';

letter_m =  [0 0 0 0 0 ...
             0 0 0 0 0 ...
             1 1 0 1 1 ...
             1 0 1 0 1 ...
             1 0 1 0 1 ...
             1 0 1 0 1 ...
             1 0 1 0 1 ]';

letter_o =  [0 0 0 0 0 ...
             0 0 0 0 0 ...
```

```
            0 1 1 1 0 ...
            1 0 0 0 1 ...
            1 0 0 0 1 ...
            1 0 0 0 1 ...
            0 1 1 1 0 ]';

letter_n =  [0 0 0 0 0 ...
            0 0 0 0 0 ...
            1 1 1 0 0 ...
            1 0 0 1 0 ...
            1 0 0 1 0 ...
            1 0 0 1 0 ...
            1 0 0 1 0 ]';

letter_d =  [0 0 0 0 0 ...
            0 0 0 0 0 ...
            0 0 0 1 0 ...
            0 1 1 1 0 ...
            1 0 0 1 0 ...
            1 0 0 1 0 ...
            0 1 1 1 0 ]';

letter_e =  [0 0 0 0 0 ...
            0 0 0 0 0 ...
            0 1 1 0 0 ...
            1 0 0 1 0 ...
            1 1 1 1 0 ...
            1 0 0 0 0 ...
            0 1 1 1 0 ]';

letter_l =  [0 0 0 0 0 ...
            0 0 0 0 0 ...
            0 1 0 0 0 ...
            0 1 0 0 0 ...
            0 1 0 0 0 ...
            0 1 0 0 0 ...
            0 1 1 0 0 ]';

letter_a =  [0 0 0 0 0 ...
            0 0 0 0 0 ...
            0 1 1 0 0 ...
            1 0 0 1 0 ...
            1 0 0 1 0 ...
            1 0 0 1 0 ...
            0 1 1 0 1 ]';

letter_c =  [0 0 0 0 0 ...
            0 0 0 0 0 ...
            0 1 1 1 0 ...
            1 0 0 0 0 ...
            1 0 0 0 0 ...
            1 0 0 0 0 ...
            0 1 1 1 0 ]';

% creating the dataset with all letters
letters = [letter_s, letter_i, letter_m, letter_o, letter_n, letter_d, letter_e, letter_l, letter_a,
letter_c, capital_letters];

%% image of the ten first letters
subplot_index = 0;
for i = 1:10
    letter = letters(:,i);

    colormap('gray');
    subplot_index = subplot_index + 1;

    subplot(1,10,subplot_index);
    imagesc(reshape(letter',5,7)')
    axis off
end
%% Setting the zeros equal to -1, in order for the Hopfield network to work
letters(letters == 0) = -1;

%% !adding the same vector again as additional dimension
letters(37:70, :) = 0;
for r = 1:36
    vector = letters(1:35,r);
    letters(36:70,r) = vector;
end
%% Error and storing different numbers of P
P = 36;
final_error_values = zeros(1,P);
final_correct_classifications = zeros(1,P);
time_steps = 1000;
```

```
% parameter to indicate the amount of correct classifications
for h = 1:P

    first_letters = letters(:, 1:P);

    % creating the Hopfield network for P number of first characters
    net = newhop(first_letters);

    % storing the distorted letters
    first_letters_distorted = first_letters;

    % storing error values of a number of 20 iterations
    storing_error_values = zeros(1,P);

    for iter = 1:20

        for i = 1:h

            % determine index of the three random components that we will change
            first_index = round(randi([1 35],1));
            second_index = round(randi([1 35],1));
            third_index = round(randi([1 35],1));
            indices = [first_index, second_index, third_index];

            for in = indices

                % inversing the pixel value
                if(first_letters_distorted(in,i) == 1)
                    first_letters_distorted(in,i) = -1;
                else
                    first_letters_distorted(in,i) = 1;
                end
            end
        end

        error_P_values = zeros(1,P);

        for k = 1:h

            letter_input = first_letters_distorted(:,k);

            % getting the best solution from the Hopfield network for the certain
            % letter
            T = {letter_input};
            [letter_output,~,~] = sim(net,{1 ,time_steps},{},T);
            letter_output = letter_output{1, time_steps};

            % rounding the values certainly to a 1 or -1
            for output_rounding = 1:70

                if(letter_output(output_rounding) > 0)
                    letter_output(output_rounding) = 1;
                else
                    letter_output(output_rounding) = -1;
                end
            end

            % the letters to check
            letters_to_check = first_letters;

            % calculating error
            if(isequal(letter_output, letters_to_check(:,k)) == true)

                error = 0;

            else

                error = sum(abs(letters_to_check(:,1) - letter_output))/2;

            end

            error_P_values(1,k) = error;

        end

            storing_error_values = storing_error_values + error_P_values;

    end

    final_error = median(storing_error_values/20);

    final_error_values(1,h) = final_error;
```

```
end

%% plotting error in function of P
p = plot(1:P, final_error_values,'-o');
p.LineWidth = 1.2;
xlabel('Number of characters P stored');
ylabel('"median of averages" error')
hold on; % hold the plot for other curves
plot(22,final_error_values(22),'o','MarkerSize',15);
txt = ('critical loading capacity');
t = text(17, final_error_values(17)+1.5, txt);
t.FontWeight = 'bold';
```