

Artificial Neural Networks

Reports on Programming Assignments

Jannes Nys

Master of Artificial Intelligence: Engineering and Computer Science

r0639870

Contents

1	REPORT 1: backpropagation in feedforward multi-layer nets	2
1.1	Function approximation	2
1.2	Generalization: noisy data and overfitting	3
2	REPORT 2: Bayesian learning in neural networks	5
2.1	‘Pure’ Bayesian training versus ‘trainbr’	5
2.2	Bayesian regularization using ‘trainbr’	6
3	Report 3: Recurrent neural networks	7
3.1	Hopfield network for digit reconstruction	7
3.2	Elman recurrent network approximation for the Hammerstein function	8
4	Report 4: Unsupervised learning: PCA and SOM	9
4.1	Principal Component Analysis on Handwritten Digits	9
4.2	Self-organizing maps: concentric cylinders	10
4.3	Self-organizing maps: unsupervised clustering of the Iris data set	10
5	PROJECT 1: Regression of function data	11
6	PROJECT 2: Classification of wine data	12
7	PROJECT 3: Character recognition	14
7.1	Hopfield network	14
7.2	Alternative solution to character recognition	15
A	Matlab code that implements the solutions	17
A.1	Regression (see Section 5)	17
A.1.1	Main code	17
A.2	Classification (see Section 6)	20
A.2.1	Main code	20
A.2.2	PCA related	22
A.3	Character recognition (see Section 7)	23
A.3.1	Main code	23
A.3.2	Image distortion	25
A.3.3	Name generation	25
A.3.4	Alternative to Hopfield	26

1 REPORT 1: backpropagation in feedforward multi-layer nets

1.1 Function approximation

In this section, the goal is to approximate the function

$$f(x) = \sin(x), \quad (1)$$

over the interval $[0, 3\pi]$ using a linearly spaced vector as input, which covers the full interval with 50 points. The standard number of neurons in the hidden layer is 5. We train the network using a number of learning algorithms¹ provided by `Matlab`. Since the ability of a network to approximate a function depends largely on its architecture, we consider one function and network architecture for all learning algorithms.

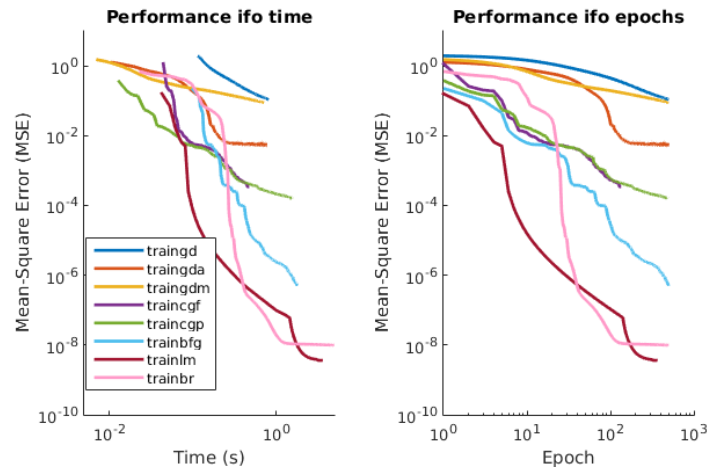


Figure 1: Performance (MSE) of the tested training rules as a function of (left) time and (right) epochs where the maximum number of epochs is limited to 1000. The log-log scale is opted to visualize the initial drop in MSE.

Below we compare the speed of various learning schemes, compared to the most basic gradient descent algorithm. We use the abbreviations of the algorithms, as used in the assignment.

[traingd] The gradient descent algorithm does not result in a proper approximation for most of the tested functions. The speed of each iteration evaluation is fast, but convergence is slow, and mostly never reached within 1000 epochs. It is clear from the broad shoulder in the performance graph in Fig. 1, that an adaptive learning rate could significantly reduce the required time for convergence.

[traingda] Fixed step-size gradient descent has the following features: if the learning rate is set too high, the algorithm can oscillate and become unstable. If the learning rate is too small, the algorithm takes too long to converge. An adaptive algorithm is an attempt to provide the best of both worlds by keeping the step size as large as possible, while allowing for smaller step sizes for locally complex error surfaces. This change is based on the performance of the network in the preceding epoch. In the first few (~ 10) epochs, the performance is very similar as for traingd, yet, due to the adaptive learning rate, the MSE is reduced significantly after 100 epochs, compared to traingd. This is reflected in an improved function approximation. Hence, approximate convergence is reached after fewer epochs. The time of an iteration is approximately the same as for the non-adaptive gradient descent.

[traingdm] This algorithm includes a momentum term in the step. Momentum softens the learning algorithms' dependence on the small local variations of the error surface. It allows to jump out of a local minimum, while other methods get stuck. Except for this feature, the performance appears very similar in Fig. 1.

¹In this report, we discuss batch learning algorithms. Online algorithms are not a part of the assignment. We only briefly mention that the online learning algorithms were found to converge faster, while the remaining MSE is larger compared to batch learning. This observation is in-line with the expectations.

[trainbfg] This quasi-Newton method beats `traingda` by orders of magnitude (even after 10 epochs, a clear difference is observed). Quasi-Newton methods build up second-order derivative information, based upon gradient information during the iteration process, allowing for faster convergence, while limiting the computational overhead.

[traincgf] and **[traincgp]** The conjugate gradient algorithms are usually much faster than variable learning rate algorithms. The considered conjugate-gradient methods perform very similar on the considered example. They only differ in the factor that determines the amount of influence of the momentum term in determining the next search direction. Then the next search direction is determined so that it is conjugate to previous search directions. These methods converge after just a few epochs, and combine this with a fast iteration step.

[trainlm] The Levenberg - Marquardt algorithm can be viewed as an adaptive mixture of a Newton-like and steepest-descent algorithm. By varying the λ parameter in the step size $\Delta x = -(H + \lambda I)^{-1}g$, one can change which algorithm is dominant. The λ is decreased after each successful step, converging towards Newton's method which allows for fast and accurate convergence close to an error minimum. This is clearly visible in the the performance plot in Fig. 1. When for a given λ the step would increase the error value, the λ is increased such that the error is always reduced. For the MSE function that is used in the network discussed here, an efficient approximation for the Hessian H can be computed from the Jacobian matrix, which can be calculated much more efficiently. Altogether, the algorithm's design leads to a significant overhead in the speed of each epoch (factor of ~ 10 compared to `traingd`). However, the computational cost is fully justified, since for a minimum number of epochs, function (1) is properly approximated. From the above analysis, the Levenberg - Marquardt algorithm shows to be the most suitable training rule to approximate function (1), given the neural network architecture and the MSE error function. It is also the default algorithm of the `feedforwardnet` object.

1.2 Generalization: noisy data and overfitting

To simulate noisy data, we add a normally distributed noise to the original dataset, with standard deviation $\sigma = 0.3$. We map the bias and variance of the various methods considered in the previous section. This is done by generating 100 data sets, and training a network with a given learning rule with at most 1000 epochs.

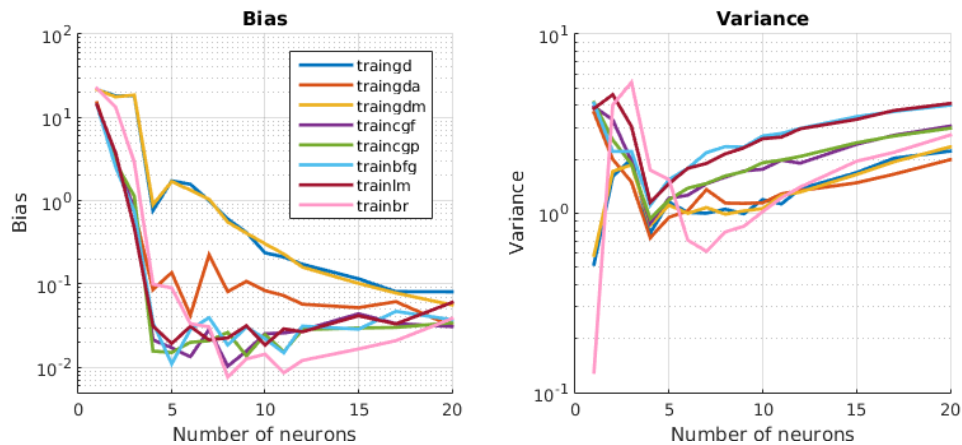


Figure 2: Bias and variance as a function of the complexity of the neural network. In this exercise, `trainbr` can be ignored, since it is discussed in the next report.

Due to the fast learning property of the Levenberg - Marquardt algorithm, it now also requires more care to avoid overfitting. This is indicated by the large variance in Fig. 2. The more the network overfits the data, the larger the variation within the ensemble of networks, and the less generalizable it is. This can be avoided by e.g. early stopping, regularization or cross validation. More general, it appears that for different learning rules, with the same architecture and cost-function, the chance of overfitting is positively correlated with the speed of convergence (which was

the topic of the previous section). Note that the bias of simple algorithms as `traingd`, `traingdm` and `trainгда` result in a high bias. This is the result of the fact that these algorithms do not converge within 1000 epochs. This is also reflected in the low variance. Since ‘a strategy to avoid overfitting’ is the topic of the next report, we refer the reader to that report for more details.

2 REPORT 2: Bayesian learning in neural networks

In this report, we discuss how Bayesian learning can be used as a method to avoid overfitting. Overfitting the noisy data in the data set of Exercise session 1 occurs when the resulted fit shows large gradients in order to reproduce every single data points (which minimizes the cost function). However, the principle of Occam's razor dictates that overcomplex systems should be avoided for simple problems. When complexity rises, errors on the validation set might increase, indicating overfitting.

2.1 'Pure' Bayesian training versus 'trainbr'

In the `perbayes` script, the weight values correspond to the MAP, i.e. the maximum of the posterior

$$\log P(\vec{w}|\mathcal{D}) = \log P(\mathcal{D}|\vec{w}) + \log P(\vec{w}) - \log P(\mathcal{D}). \quad (2)$$

for the data \mathcal{D} . Since the last term does not depend on the weight vector \vec{w} , we do not consider it in the following. Note that, if one compares this with the cost function M in the assignment, we note that no hyper-parameters α and β are included here. Therefore, we refer to this as the 'pure' Bayesian algorithm. This is a restriction compared to the more general `trainbr` algorithm discussed below, where hyperparameters determine the effect of the prior and likelihood. These hyperparameters are also obtained from the data in a Bayesian way.

This first step of the exercise is mainly to see that regularization can be intuitively interpreted in a Bayesian framework. The reason for this is was illustrated above. However, without the inclusion of hyperparameters, we face a number of problems in the given simulation, which are to be avoided.

In order to better understand the need for the hyper parameters, we vary the number of data points in the data set. For a high number of data points, the influence of the prior on the posterior is negligible. However, the likelihood calculates the probability of generating a set of discrete values (0 and 1), from a continuous distribution (a sigmoid in this case). Naturally, the optimization converges to a sigmoid function which is as sharp as possible, corresponding to a weight vector in the outskirts of its domain. In the limit of an infinite data set, the modulus of the weight vector is over estimated, and the direction is incorrect. The latter is due to the fact that the modulus of the weight vector is largest when $|\vec{w}| = \sqrt{2}$. Hence, a sharp boundary is favoured over a correctly aligned boundary. This is of course an effect of a limited range for the weight vector size. However, the main point is that the weights are over estimated. Overfitting of the network in principle corresponds to an overestimated $|\vec{w}|$. This effect we wish to counteract by introduction of the prior, which gives a higher prior probability of obtaining smaller values of $|\vec{w}|$.

For similar reasons, the MAP does not always result in a weight vector which is close to the weight vector of the original network. In most cases, the weight vector has an over estimated modulus. An example is shown in Fig 3.

For a low number of data points, the prior has too much effect, leading to a misclassification of a significant percentage of the samples.

Adjusting the effect of the prior properly can reduce this effect. Hence, introducing hyper parameters to scale the effect of the data and prior can improve the fitting. This is the case for the Bayesian regularization algorithm `trainbr`.

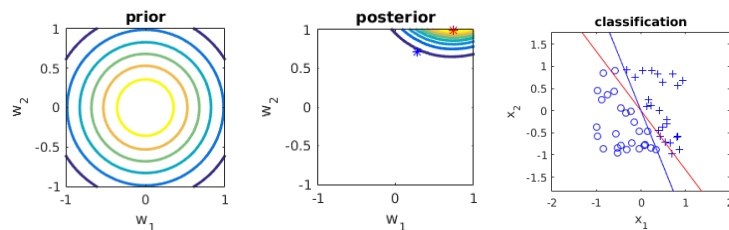


Figure 3: Prior, posterior and classification with corresponding separation line of the original (blue) and MAP network (red). Some (but not many) samples are misclassified.

2.2 Bayesian regularization using ‘trainbr’

We now focus on the data set used in the previous report to discuss overfitting, and how Bayesian regularization may help reduce it. An example of the effect of a Bayesian regularization term is illustrated in Fig. 4 where we compare the Maximum-a-posteriori result with a Maximum-Likelihood result. In the case at hand, in essence, we wish to approximate the underlying function $\sin(x)$ of the

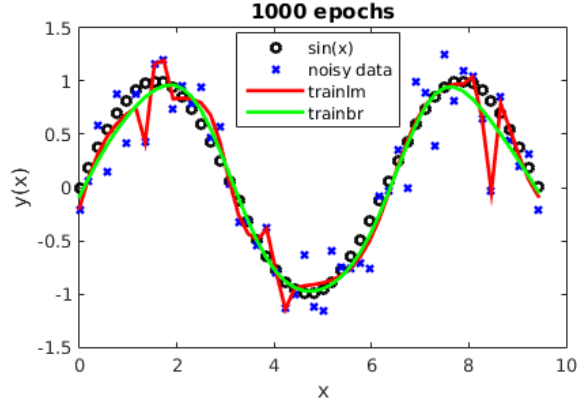


Figure 4: Illustration of an overfitted network trained with the Levenberg-Marquadt algorithm (red), compared to the results of the Bayesian learning (green) for a data set with $h(x) = \sin(x) + \mathcal{N}(\sigma)$ where $\sigma = 0.3$. Both networks have the same architecture, which includes 10 neurons in the hidden layer.

data, rather than the data set itself. In the Bayesian framework, regularization can be implemented in a natural way by opting for a parameter prior which has the bulk of its probability close to small values of the network weights (like a gaussian).

In the last report we characterized generalization and overfitting via the network’s variance and bias². We can therefore use the figure used in the previous report, now shown in Fig. 5, and compare the **trainbr** algorithm with the other training algorithms. Figure 5 compares the bias and variance as a function of the complexity of the network for different learning algorithms.

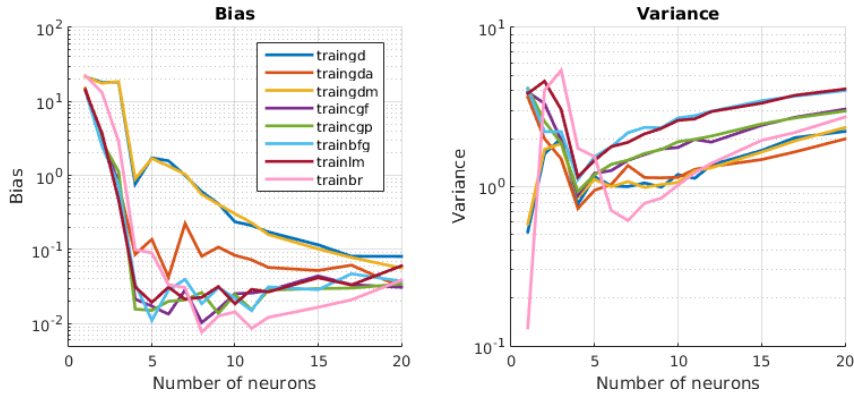


Figure 5: Bias and variance plot as a function of the complexity of the neural network.

An overfitted network corresponds to a low bias but high variance in Fig. 5. Assuming convergence has been reached, a large variance indicates that the resulting network depends largely on the data set to which it is fitted, and hence, to the noise it includes. The effect of the Bayesian regularization term is most clear in the interval of medium complexity (5-15 hidden neurons). Both the bias and variance remain low, indicating that the **trainbr** algorithm allows for a more accurate and precise function reconstruction, while avoiding noise fitting (see again Fig. 4 for an illustration). Interestingly enough however, Fig. 5 also clearly shows the limited range of application of the Bayesian regularization. When the complexity is high enough, the regulation term has a negligible effect, and also **trainbr** will start overfitting.

²Naturally, we could have also done this using a test set.

3 Report 3: Recurrent neural networks

3.1 Hopfield network for digit reconstruction

In this section, we use a Hopfield network to reconstruct 15×16 pixel digits. Assume p denotes the number of patterns stored and N the dimensions of those pattern vectors. Since $p/N = 10/240$ is reasonably small, the probability of instabilities is small.

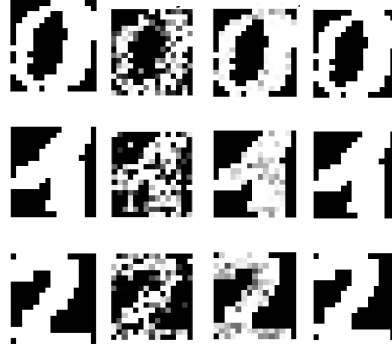


Figure 6: From left to right: attractor states, noisy digits, reconstructed digits after 1 iteration, reconstructed digits.

The retrieval states are the attractors of the system. They form the local minima of the associated energy function of the system. By distorting the original image (which is located at a local minimum), one increases the energy of the system (the image), allowing for a jump to a state (image) away from the original minimum. The higher the noise level, the less the distorted image resembles the original image. The Hopfield network is used to ‘cool the system down’, lowering the energy along the geodesics defined by the weight matrix, such that it reaches a (local) minimum (or saddle points if one is unlucky). It can be shown that each step in a Hopfield iteration corresponds to a lowering of the energy of the matrix. The final minimum can of course be different from the original one if distorting the image results in a state located in the attraction basin of another mode of the energy surface. In the following, we use the term ‘distortion energy’ as the average energy that is added to the image during the distortion at a given distortion level.

When the noise level is low, the distorted image ends up within a small perimeter on the energy surface around the original image. Hence, in the majority of the cases, the image is properly restored. When the distortion increases, the contour of the domain of distorted images increases in energy and hence in perimeter. The first false image reconstructions occur when the distortion energy reaches the same energy as the ridge separating its original minimum from another local minimum. In this case, when the distorted image is cooled down and energy is drained, it may end up in the wrong minimum. Hence, if this local-minima-separating ridge has a small energy (meaning that the two minima have a lot of resemblance), then this phenomenon will occur at low distortion. For a noise fraction between 0 and 1, I observed no false reconstructions. However, if I increase the noise level above 1, more interesting cases occur. The digits for which we first observe a false reconstruction when increasing the noise level, are at distortion level³ 5, for the ‘2’ and ‘9’. These respectively end up in the ‘7’ and ‘2’ minimum in a large number of cases. This is expected, since these digits have similar features.

A second factor that needs to be taken into account is the number of iterations in the cooling process. If the number of iterations is too low, the system will not be able to fully cool down and will not reach a minimum. The maximum number of iterations should be large enough (such that the image results in an attractor state) and depends on the distortion level: the higher the distortion, the more steps are needed to guide the system to a local minimum. For the case at hand, 100 iterations is generally sufficient.

A problem of Hopfield networks is that spurious states may occur. These are states which are

³The noise distortion level of the `hopdigit` function is supposed to be a noise fraction. However, nothing prevents us from taking values above 1.

not put in explicitly, but are generated from a linear combination of retrieval states. In the digit reconstruction, no such states were found in any simulation (even when the noise level is high). The reason for this is that it is unlikely to reach such a state, since it forms a perfect symmetry. We expect that clipping the initial noisy input would result in a higher probability to end up in spurious states. When the number of iterations is kept low, one can recognize combinations of digits in the reconstructed images. These states are close to the expected spurious states.

As a final test, we use other handwritten versions of the digits. These are also included in the `digits` workspace. Even with a low noise input, the Hopfield network can propagate to the wrong attractor state.

3.2 Elman recurrent network approximation for the Hammerstein function

First, the `elmants2` script was adapted. We generate time series set of 300 subsequent time steps. These are divided in two blocks with training:validation ratio 85:15. Note that, since we are working with a time series, the `newelm` uses blocks of subsequent time data instead of randomized indices. The test set is a sequence of 200 datapoints, separated from the initial set by 500 time steps.

In the documentation of `newelm`, Matlab makes the following interesting warning: “*Algorithms which take large step sizes, such as TRAINLM, and TRAINRP, etc., are not recommended for Elman networks. Because of the delays in Elman networks the gradient of performance used by these algorithms is only approximated making learning difficult for large step algorithms.*”

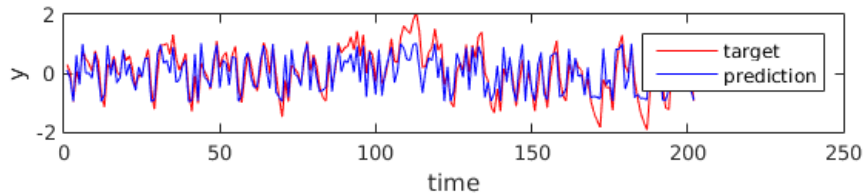


Figure 7: Output of an Elman network with 10 neurons in the hidden layer and the `tansig` and `purelin` transfer function in the hidden and output layer respectively. Also shown is the target test set. The correlation between both data sets is 0.795. One should be aware that there is a time gap between the train and and test set.

In contrast to the non-dynamical case, where `trainlm` is often the default learning algorithm, `newelm` uses `traingdx` by default. This algorithm uses a gradient descent with momentum and adaptive learning rate backpropagation.

The use of the transfer functions as `logsig` is to be avoided, since its output domain is strictly positive. Especially when used in the output layer, the performance is low.

Even with the most simple architecture with `purelin` in the hidden and output layer, strong correlations (> 0.75) between the prediction and target test set are obtained⁴. Using the `tansig` transfer function in the hidden layer efficiently yields large correlations in the test set. However, it does not properly reproduce the outliers. This is illustrated in Fig. 7. However, we choose this configuration as the optimal one, since the test set correlation is maximal compared to all tested configurations.

In Fig. 8 we map the test-set correlation and MSE in function of the number of neurons in the hidden layer. From this plot, it is clear that the number of neurons must be limited to < 20 in order for the network to be generalizable.

⁴In the following, we will refer to this correlation as ‘the correlation of the test set’. We opt to quantify the performance of a network on the test set by this correlation, since the underlying Hammerstein function models a stochastic process.

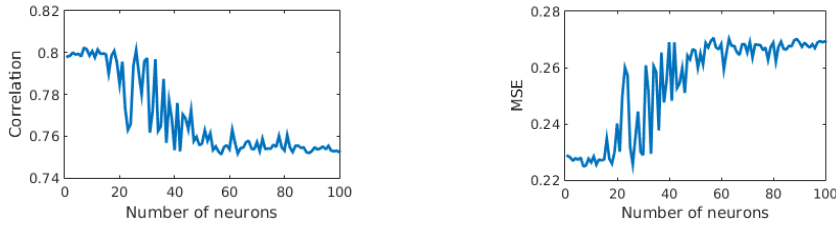


Figure 8: Correlation (left) and mean-square error (right) of the Elman network tested on the test set in function of the number of neurons in the hidden layer.

4 Report 4: Unsupervised learning: PCA and SOM

4.1 Principal Component Analysis on Handwritten Digits

In the Principle Component Analysis, one decomposes the data entries into the principal eigenvectors of the covariance matrix. Each figure is then approximated by a linear combination (for linear PCA) of the limited set of eigenvectors.

Two types of data were considered: non-standardized and standardized data. The former corresponds to the original data set, while in the latter, the data is preprocessed by putting the standard deviation of all pixels to 1. However, the data preprocessing was found to be less stable, resulting in higher reconstruction errors. Hence, we continue with the raw data below.

Figure 9 shows the total reconstruction error for the data set of threes as a function of the number of eigenvectors used in a PCA projection. Figure 9 also depicts the total sum of neglected eigenvalues. The figure illustrates that both quantities are proportional⁵. Hence, by setting the required maximum reconstruction error, one can choose a proper number of eigenvectors of the covariance matrix to span the compressed space based on the cumulative sum of the eigenvalues.

When k is equal to 256, which is the dimension of the original basis, the eigenbasis of the PCA spans the full space of the original figures. Hence, the reconstruction error is approximately⁶ zero, since the PCA is now a rotation of the figures, which is inverted exactly upon reconstruction.

As an example, we show a reconstructed image for a number of dimensions of the PCA basis in Fig. 10. As the basis increases in dimensions, the reconstructed image converges towards the original image.

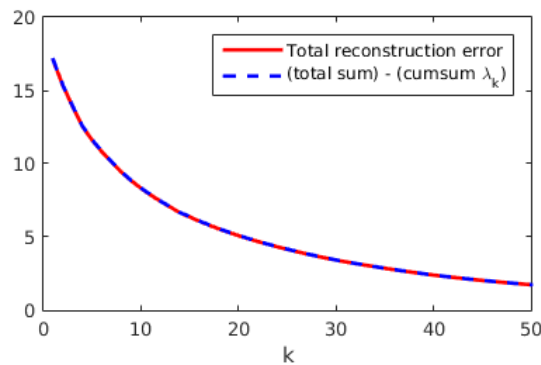


Figure 9: Total sum of all eigenvalues minus the cumulative sum of the reconstruction k largest eigenvalues of the covariance matrix (dashed blue) and mean-squared reconstruction error (solid red) as a function of the dimensionality of the PC basis.

⁵Interestingly, the same simulation was also carried out without a standardized data set. In this case, the two curves in Fig. 9 overlap exactly.

⁶Due to computational round-off errors, during the eigenvector decomposition and reconstruction, the obtained is not exactly 0, but rather 6.7150×10^{-29} .



Figure 10: Example of a reconstruction of an image using a basis of (f.l.t.r.) 1, 2, 3 and 4 principal eigenvectors. The outer-rightmost figure is the original, uncompressed image. It was observed that for this data item, a basis of ~ 50 eigenvectors is required to restore the image properly.

4.2 Self-organizing maps: concentric cylinders

Initially, a grid with a fixed (gridtop or hextop) or random (randtop) topology are generated. Using competitive learning, the weight vectors vary in such a way that they form a support for the concentric cylinder. The advantage of the fixed topologies is that these are easy to project onto 2D space, which makes the final result presentable. In this exercise, however, the purpose of the SOM is to represent the data density and topology using a lower number of vectors than the input data set.

It was found that fixed topologies generally have a number of weight vectors that remain in the center of the cylinder, while for ‘randtop’ combined with ‘dist’, this is not the case. The various distance measures in the representation space were found to have little effect on the general trends discussed above.

4.3 Self-organizing maps: unsupervised clustering of the Iris data set

The given example script `example_SOM_Iris.m` allows one to cluster the Iris data set given the prior knowledge that 3 different classes can be distinguished. More interesting is to run a SOM, without implementing such knowledge and using the SOM to retrieve structural information about the data set. This study is discussed below. In this exercise, the benefits of a fixed topology are clear: it allows one to map a 4D data set onto a representable 2D figure. Hence, we opt for a 10×10 ‘hextop’ grid and use the ‘dist’ measure (the typical SOM rule of thumb suggests $5\sqrt{N} = 61$ units for $N = 150$). If too many weight vectors are included, many will never win and receive a hit. When the number of weight vectors is too low, the structure is washed out. The SOM adjusts its weights 1000 times.

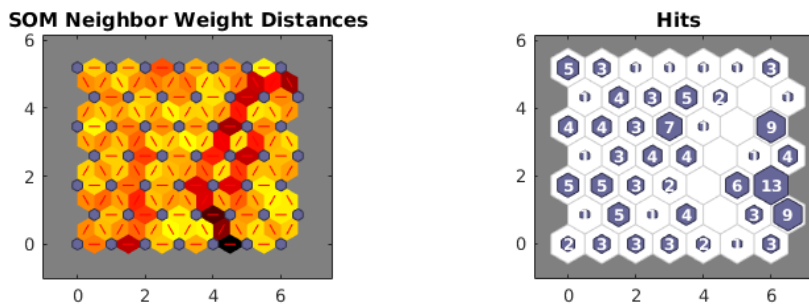


Figure 11: From left to right: neighbor weight distance (darker is further) and number of hits per weight vector.

In Fig. 11, we show the distances between the weight vectors, and the number of hits that corresponds to a certain weight vector. These visually indicate that there are two clearly distinguishable data types which are far apart in feature space. In between this region, there are a number of weight vectors that receive little to no hits. This also shows that, since we cannot clearly distinguish a third cluster, two clusters are located in each other’s neighbourhood in feature space.

5 PROJECT 1: Regression of function data

In this project, we use neural networks to approximate an unknown real scalar function $f : [0, 1] \times [0, 1] \rightarrow \mathbb{R}$, given a data set that was generated from this function. Since the underlying function is non-linear, a non-linear transfer function is required. Bearing in mind the results of Leshno *et al.*⁷, we find that the `tansig` (non-polynomial) function for the hidden layers must allow a function approximation if the number of neurons is sufficient. For the output layer, we use a `purelin` transfer function, which does not restrict the output values to a particular range of values. In most cases, a single layer neural network is sufficient to reproduce a function (in theory, it is sufficient for *all* functions if the number of neurons is large enough). Hence, we start with a single layer. Since the typical Euclidian distance between data points is small compared to the typical dimension of the spatial structure, in addition to the fact that the data has not been jittered, the number of neurons can be taken relatively high without overfitting. To get an estimate of the number of neurons, we refer to Fig. 12 where we map the MSE of the training and validation set as a function of the number of neurons in the hidden layer⁸. Note that at this stage, we may not consider the test set performance. We opt for 40 neurons, since this architecture forms a minimum for the combined validation and training MSE. In principle, the MSE plot suggest that 60 neurons might yield an even lower MSE. However, the bump between 40 en 60 neurons suggests that the final result might depend on the seed used to initialize the optimization process. This can in principle be resolved by using multiple initializations, which is very computationally intensive. However, it is safer and sufficient to take the first minimum at 40 neurons. Note that to generate Fig. 12, we used the validation set in a stopping criterion.

First, we discuss the purpose of the individual data sets. Training is done on the training set, meaning that the network-parameters are varied such that it reproduces the training set with an overall error which is as small as possible. When the network starts overfitting the training data, the error on the validation set typically begins to rise. This indicates that the network is adjusted in such a way that it minimizes the error with respect to the training set, while having bad generalization properties in the regions where no training data is available. Therefore, when the validation error does not decrease below the lowest validation MSE for a specified number of iterations (`net.trainParam.max_fail` = 1000 or less if the number of epochs exceeds 5000), the training is stopped and the weights and biases at the minimum of the validation error are returned. The test set only serves as a final check and cannot be used in the training and validation process.

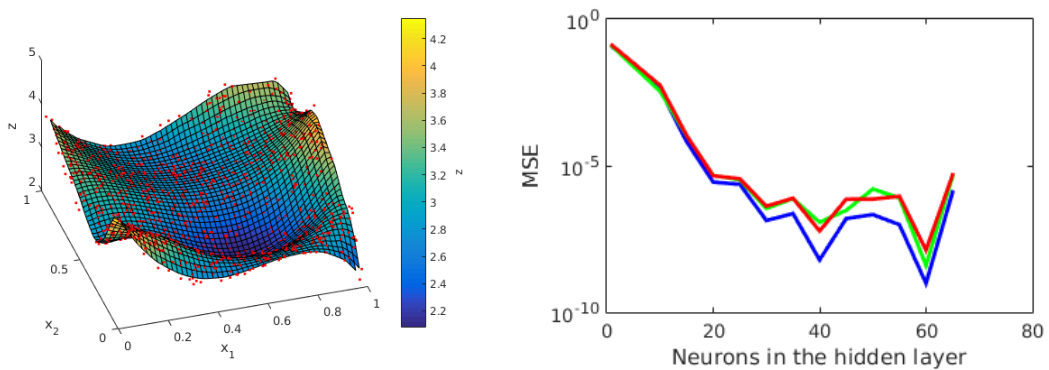


Figure 12: Left: surface and data points in the training set. Right: Performance on the training, validation and test set as a function of the number of neurons in the hidden layer using 5000 epochs.

The MSE on the test set of the resulting network is 2.2×10^{-8} . It clearly shows that the number of neurons is more than sufficient. We observe the pattern that the network has slightly higher values when the test-set interpolation is large, and smaller values when the interpolation function

⁷Leshno, M., Lin, V. Y., Pinkus, A., & Schocken, S. (1993). Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. *Neural Networks*, 6 (6), 861867

⁸This strategy allows us to get the optimal number of neurons, given a certain activation architecture and maximum number of epochs (5000).

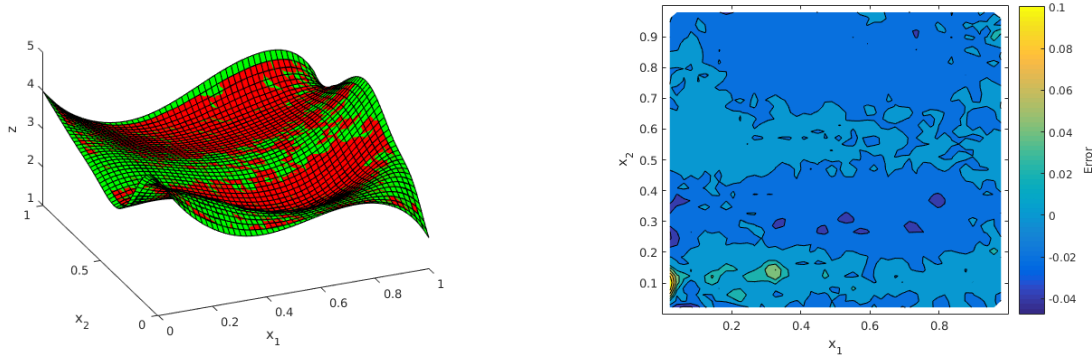


Figure 13: Left: Neural network (green) and test data (red) surface. The surfaces are almost indistinguishable. Right: Contour plot of the test set error.

is small. Hence, more extremal values are obtained. In the error surface plot in Fig. 13, the effect of a large number of neurons is clear: a small irregular oscillatory behaviour is observed.

To improve the results, we can use all of the provided data, instead of a subset of 3000 data points. Since the number of training data entries is significantly larger, we also allow for 10 additional neurons, resulting in 50 neurons in the hidden layer. Also, the data division does not need to be equal for all sets. As a rule of thumb, one generally assumes 70:15:15 (or 60:20:20) for the train:validation:test set ratios. This changes the MSE test error to 1.46×10^{-9} .

Note that one could even improve these results by e.g. using cross validation, which allows one to use an even larger size training set. In 10-fold cross validation for example, 90 percent of the data can be used for training in each step. In the training-validation-test set strategy that we have applied above, 30-40 percent of the data is used for checking generalization properties of the network.

Also, training multiple networks to approximate the function can in principle result in a better approximation. However, a certain bias was observed above. Since the current MSE is already low enough, we do not go into further details.

6 PROJECT 2: Classification of wine data

According to my student number, I classify data sets 5 and 6. Since it is not given whether or not the wine data is linearly separable, we use a non-linear model. We now use a `tansig` transfer function in the output layer, since this restricts the output to the domain $[-1, 1]$ and the target output is in the format of ± 1 . 15 neurons in the hidden layer is optimal to classify the validation data. Using this architecture, a CCR of 0.737 and 0.724 is obtained for the validation and test set respectively. This might indicate that the two wine classes have a significant overlap in their attribute space. The effect of standardization of the data in this case appeared to have a rather negative effect.

Next, the data is projected onto its lower dimensional principle component basis and reconstructed afterwards. The eigenvalue spectrum of the covariance matrix is depicted in Fig. 14. Prior to computing the covariance matrix, we rescale the data such that all attributes have zero mean and unit standard deviation. The effect of this rescaling results in a CCR difference of 3 – 6% in the final results (without standardization, we obtain 0.704 and 68.4 for the CCR of the validation and test set respectively). The scaled eigenvalue spectrum does not show a clear dominant principle component. We also show the non-standardized eigenvalue spectrum, which shows that only a single component dominates the spectrum. By rescaling the data, other features are emphasized. Note that the dominant eigenvalue might very well be related to the choice of attribute units. Hence, it makes sense to rescale the data before performing PCA.

There is no clear indication that the standardized data lives on a low-dimensional subspace. We take a reconstruction error of 10%, which corresponds to an 8 dimensional basis. 11 hidden

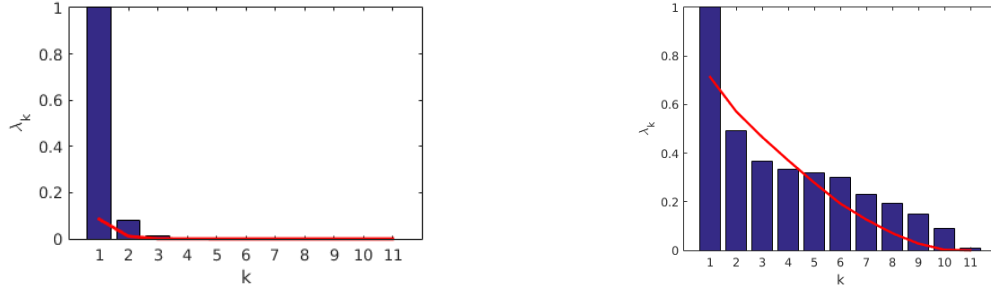


Figure 14: Eigenvalues of the 11×11 covariance matrix. The left and right panel show the result for the original and standardized data respectively. The red line indicates the cumulative sum of the remaining $(k+1)$:end eigenvalues, which is proportional to the reconstruction error.

neurons minimize the CCR of the validation set, which is less than before, as expected. Using this approach, a validation and test set CCR of 0.754 and 0.717 is obtained. Hence, there is a CCR increase of 5% for the validation set, while test set CCR is almost constant. Hence, by linearly projecting the data onto a lower dimensional space, an insignificant classification improvement is realized. By rescaling the data, the PCA analysis in principle corresponds to the original PCA, where the covariance matrix is substituted by the correlation matrix, which is dimensionless.

We remark that the possibility of overfitting is very real in this classification assignment. One can obtain an almost perfect CCR on the training set, while the test CCR is very low.

We have tested the effect of an additional hidden layer, since these are known to be beneficial in complex classification problems. The architectures [10 10], [15 15] and [20 20] and even architectures with more than 2 hidden layers did not improve the results and resulted in at most the CCR ratios of the single-layer network. Hence, the additional non-linearity of the hidden layers does not improve the classification.

In principle, one of the reasons the PCA has a small effect can be due to the fact that it is organised in input space in such a way that linear PCA does not work. Hence, in principle other methods for PCA should be tested.

7 PROJECT 3: Character recognition

7.1 Hopfield network

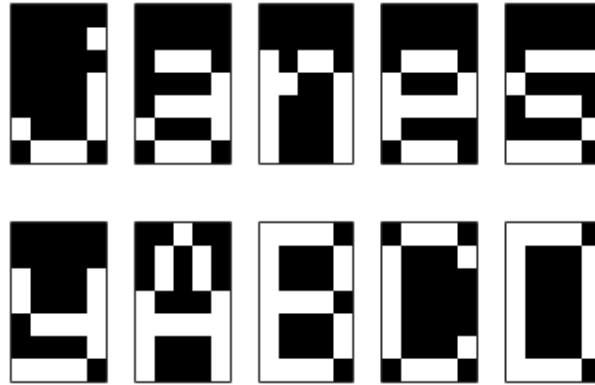


Figure 15: First 10 of the 32 letters that are used in the character recognition exercise.

In this section we train a Hopfield network to reconstruct the states in Fig. 15 from a distorted version of these characters. Hopfield networks are generated by providing them the undistorted images.

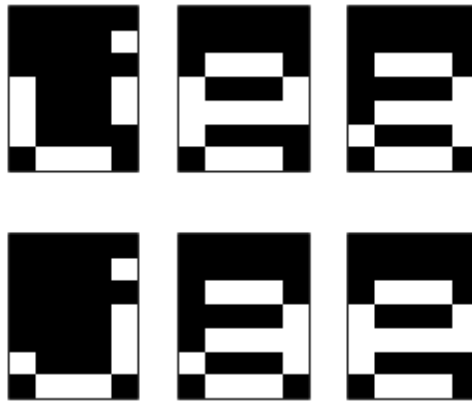


Figure 16: Incorrectly reconstructed characters (top row) with corresponding input (bottom row).

Figure 16 illustrates a number of incorrectly reconstructed states with corresponding undistorted input. The ‘e’ and ‘a’ characters are often interchanged in the reconstruction. This is related to the fact that these characters have many pixels with the same value, and hence a similar shape. This means that the ridge in the energy function between these two attractor states is lower than the 3-pixel distortion. The incorrectly returned ‘j’ character shows a spurious state which is a linear combination of multiple characters. These cases are obtained when allowing for a sufficiently large number of time steps (1000). This allows one to observe the spurious states that are inherent to the system, rather than states that have not yet converged.

We now determine how the number of patterns stored influence the number of erroneously restored characters. For each case of P stored patterns, we create a Hopfield network and generate a sufficient number of 3-pixel distorted images. These are then reconstructed, after which we calculate the number of pixels that do not correspond to the original image. The results are shown in Fig. 17. One can observe a steep rise in the reconstructed pixel error after $P = 17$. Hence, 17 stored patterns is the critical loading capacity of the network. Naturally, when the number of

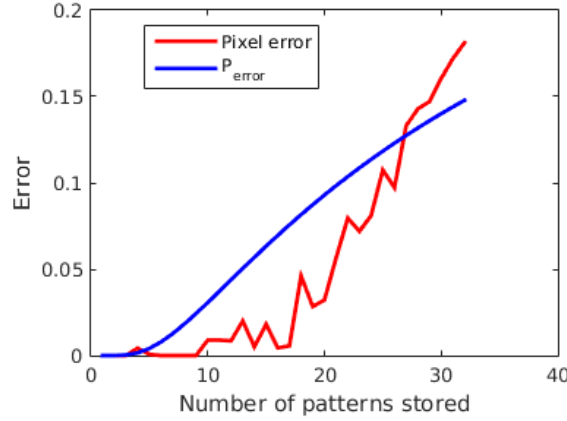


Figure 17: Total pixel error (red), normalized over the number of states generated and total pixels in an image (35), and the theoretical prediction curve based on the Hebb rule (blue).

distorted pixels is increased, this number is expected to decrease. For P larger than the critical loading capacity, the large number of spurious states and the small basins of attraction of attractor state do not allow for a reliable character restoration.

One can theoretically predict the dependence of the reconstruction error as a function of the number of patterns stored, using Hebb's rule for uncorrelated patterns. The prediction curve is also shown in Fig. 17. If we assume a large P and large N , we obtain an estimated P_{error}

$$P_{\text{error}} = \frac{1}{\sqrt{2\pi}\sigma} \int_1^{+\infty} e^{-\frac{x^2}{2\sigma^2}} dx, \quad (3)$$

where $\sigma = \sqrt{P/N}$. Hence, a first approximation is that P and N are large, which is not actually the case here. However, both the prediction and simulation show the same (expected) behaviour: first a flat P dependence, followed by a steep rise. One can estimate the storage capacity from

$$P_{\text{max}} = N/(4 \log N) = 2.46. \quad (4)$$

Hence, for 2 or less stored patterns, the Hopfield should be able to perfectly reconstruct the distorted images. It can be shown⁹ theoretically that the storage capacity is $P = 0.138N = 4.83$, for which if P is larger, the system might get stuck in spurious local minima.

One way to resolve the issue of incorrectly reconstructed images, is to increase the number of pixels in each image. In the case at hand, each image is determined by 35 pixels. Hence, it was expected that the critical loading capacity would be relatively low. By increasing the number of pixels in an image, each character has more attributes by which it is characterized. This lowers the P/N ratio. Using Eq. (4), one can estimate the number of patterns that can be stored.

This is the obvious alternative to the current Hopfield network. In the next subsection we discuss another alternative which is more general and can easily be extended to more than 25 characters. Since in principle, two-layered neural networks can reproduce any function, we can in principle form a perfect classifier (since we know *all* 3 distorted images and hence can evaluate the underlying function everywhere) if and only if there are no overlapping distorted images. It requires, however, much more computing power compared to the Hopfield solution, since the Hopfield solution's training process is almost instantaneous. Note that many more solutions can be put forward, which all have their pros and cons. Naturally, also SVM's can be used in this case.

7.2 Alternative solution to character recognition

The Hopfield network for character recognition is “only implemented in MATLAB for historical reasons”. Nowadays, many more, and better ANNs are available on the market. Not only can one

⁹D. J. Amit, H. Gutfreund, and H. Sompolinsky. Statistical Mechanics of Neural Networks near Saturation. Annals. of Physics 173, pp30-67, 1985

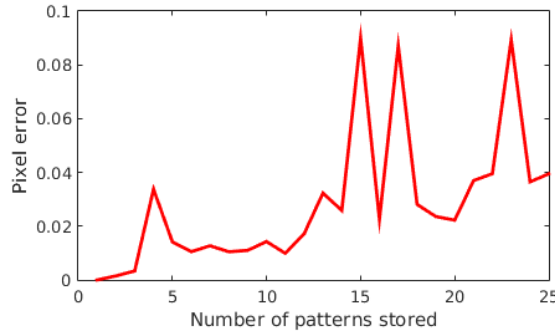


Figure 18: Percentage of false reconstructed images (not pixels as before) as a function of the number of patterns stored.

choose the architecture of the network, but also the error determination can be chosen to optimize the problem. Another choice is which input is used to train the network. It has been shown that (as expected), a network performs better in character restoration if it is trained with distorted input images. This is clearly illustrated in an example by Matlab ¹⁰.

The opted network architecture is as follows: we generate a feedforward neural network with one hidden layer of 25 neurons and an output layer of dimension equal to the number of patterns P that are need to be stored. In the hidden layer, we use a `tansig` transfer function, while in the output layer, we use the `softmax` function. The final output is such that we put all output components to 0, except for the one with the maximum value, which is 1. Hence, we use one-hot encoding of the stored characters. This means that every character out of the P stored characters is determined by a P -dimensional vector of zeros, except for a one in a unique place. Hence, also the actual characters must be stored such that one can return a character instead of this vector. Note that the input still takes the 35 pixels as in the original Hopfield network. In the syllabus a similar approach is discussed where the Hamming distance is used to select the final character when the image output vector results in an undetermined result. However, using the softmax function, we almost never find inconclusive results.

One-hot encoding is feasible if the number of patterns stored is not too large, since the size of the output layer is equal to the size of the alphabet to be stored.

In the previous subsection we showed that the “e” and “a” are difficult for the Hopfield network to restore correctly, since these characters have very similar features. Hence, if one would create a network that uses an $N = 35$ dimensional output, we know a priori that similar problems would be faced, since these characters are only separated by a small distance in output space. In one hot-encoding, the distance between two patterns in output space is $\sqrt{2}$, independent of whether they have similar patterns.

We train our network with 100 distorted images of each letter to be stored. We vary the number of patterns and determine the number of incorrectly restored patterns as a function of the number of patterns stored. We use a constant 25 hidden neurons for practical reasons. The results are shown in Fig. 18. A steady rise is observed as expected.

Naturally, the network’s performance can easily be improved by increasing the number of neurons and/or hidden layers. In addition, approximately 6500 possible 3-pixel distortions of each letter. Hence, in theory all possibilities can be generated to form the training batch. However, substantial computing resources are required if batch-training is required. We will not go into further detail, since these calculations are hard to perform on a simple laptop.

¹⁰<http://nl.mathworks.com/help/nnet/examples/character-recognition.html>

A Matlab code that implements the solutions

A.1 Regression (see Section 5)

A.1.1 Main code

```

1  clear all; close all; clc;
2
3  %% LOAD THE DATA FROM TOLEDO
4
5  % load the data from Toledo
6  load 'data/Data_Problem1_regression.mat'
7
8  % generate my data from studentnr 0639870
9  d1 = 9;
10 d2 = 8;
11 d3 = 7;
12 d4 = 6;
13 d5 = 3;
14 T = (d1*T1 + d2*T2 + d3*T3 + d4*T4 + d5*T5)/(d1 + d2 + d3 + d4 + d5);
15 X = [X1 X2]; % useful for running sim
16 N = size(T,1);
17
18 % a check
19 assert((size(X1,1) == N) && (size(X2,1) == N));
20
21 %% DIVIDE THE DATA IN SUBSETS
22
23 % randomize the data, shuffle it, also transpose it
24 shuffledInd = randperm(size(T,1));
25 X1 = X1(shuffledInd);
26 X2 = X2(shuffledInd);
27 T = T(shuffledInd);
28 X = [ X1'; X2' ]';
29
30 % divide the data in subsets (use the first 3000, they are already shuffled)
31 rng(1);
32 useN = 3000;
33 [trainInd, valInd, testInd] = dividerand(useN,1/3,1/3,1/3);
34
35 % easier variables for later
36 xtrain = [X1(trainInd)'; X2(trainInd)']; % input
37 ytrain = T(trainInd,:)'; % target
38 xval = [X1(valInd)'; X2(valInd)'];
39 yval = T(valInd)';
40 xtest = [X1(testInd)'; X2(testInd)'];
41 ytest = T(testInd)';
42
43 %% VIZUALISE THE DATA
44 % create a regular grid for the interpolant (which is not the net)
45 ndim = 50;
46 [Xmesh, Ymesh, ~, ~] = meshinterpolate(X1, X2, T, ndim);
47 % now interpolate
48 trainInterpolate = TriScatteredInterp(X1(trainInd), X2(trainInd), T(trainInd))
    ; % plot the interpolator
49 Zmesh = trainInterpolate(Xmesh, Ymesh);
50 % show the surface
51 figure;
52 surface(Xmesh, Ymesh, Zmesh); hold on;
53 % also show the data

```

```

54 scatter3(X1(trainInd), X2(trainInd), T(trainInd), '.', 'MarkerEdgeColor','red'
    , 'MarkerFaceColor', 'red');
55 xlabel('x_1'); ylabel('x_2'); zlabel('z');
56 colorbar; h = colorbar; ylabel(h, 'z');
57 savefig('train_surface.fig'); hold off;
58
59 % show that the data is well distributed and divided randomly
60 figure;
61 surface(Xmesh, Ymesh, Zmesh); hold on; % plot the interpolator
62 scatter3(X1(trainInd), X2(trainInd), T(trainInd), 15, 'MarkerFaceColor','g');
    hold on;
63 scatter3(X1(valInd), X2(valInd), T(valInd), 15, 'MarkerFaceColor',[1 .5 0]);
    hold on;
64 scatter3(X1(testInd), X2(testInd), T(testInd), 15, 'MarkerFaceColor','r');
    hold off;
65
66 %% CREATE THE VALIDATION PLOT
67
68 % create some vectors to plot
69 nhvals = [1,10,15,20,25,30,35,40,45,50,55,60,65];
70 mseVal = zeros(length(nhvals),1);
71 mseTrain = zeros(length(nhvals),1);
72 mseTest = zeros(length(nhvals),1);
73
74 for nhIt = 1:length(nhvals)
75
76     % change the number of neurons in the hidden layer
77     nh = nhvals(nhIt);
78     disp(['Training with ' num2str(nh) ' neurons in hidden layer']);
79
80     rng(1); % same seed for all
81
82     % train the network
83     net = feedforwardnet(nh, 'trainlm');
84     net.divideFcn = 'divideind';
85     net.divideParam.trainInd = trainInd;
86     net.divideParam.valInd = valInd;
87     net.divideParam.testInd = testInd; % we won't need it in the final plot,
        so it's just for monitoring
88     net.layers{1}.transferFcn = 'tansig';
89     net.layers{2}.transferFcn = 'purelin';
90     net.trainParam.showWindow=0;
91     net.trainParam.epochs = 5000; % maximum number, take it large, since I don
        't know how long it will take
92     net.trainParam.max_fail = 100; % set really high, so it can decide itself
93
94     % train the network
95     [net,tr] = train(net, X', T', 'UseParallel', 'yes');
96     %nntraintool('close');
97
98     % show number of epochs used
99     disp(['Used ' num2str(tr.num_epochs) ' epochs']);
100
101     % make predictions for the data sets
102     ytrainPred = sim(net, xtrain);
103     yvalPred = sim(net, xval);
104     ytestPred = sim(net, xtest); % plot this for completeness
105
106     % calculate the mse for these points
107     perfTrain = perform(net, ytrain, ytrainPred);

```

```

108     perfVal = perform(net,yval,yvalPred);
109     perfTest = perform(net,ytest,ytestPred);
110
111     % store it
112     mseTrain(nhIt) = perfTrain;
113     mseVal(nhIt) = perfVal;
114     mseTest(nhIt) = perfTest;
115 end
116
117 %% SHOW PERFORMANCE ON VALIDATION SET
118 figure
119 semilogy(nhvals, mseTrain, 'Color','b', 'LineWidth',2); hold on;
120 semilogy(nhvals, mseVal, 'Color','g', 'LineWidth',2); hold on;
121 semilogy(nhvals, mseTest, 'Color','r', 'LineWidth',2);
122 xlabel('Neurons in the hidden layer'); ylabel('MSE');
123 savefig('performance_val.fig');
124
125 %% SELECT A NETWORK ARCHITECTURE FROM THE RESULTS
126 nhfinal = 40;
127
128 %% PLOT THE RESULT OF THE OPTED ARCHITECTURE
129 % train the network
130 net = feedforwardnet(nhfinal,'trainlm');
131 net.divideFcn = 'divideind'; % also give it access to the other sets for
    plotting
132 net.divideParam.trainInd = trainInd;
133 net.divideParam.valInd = valInd;
134 net.divideParam.testInd = testInd;
135 net.layers{1}.transferFcn = 'tansig'; % hidden layer
136 net.layers{2}.transferFcn = 'purelin'; % output layer
137 net.trainParam.epochs = 20000; % set really high, so it can decide itself
138 net.trainParam.max_fail = 1000; % set really high, so it can decide itself
139 [net,tr] = train(net, X', T', 'UseParallel', 'yes');
140 plotperform(tr)
141
142
143 % calculate its output on the meshgrid
144 ZmeshNN = net([Xmesh(:) Ymesh(:)]');
145 ZmeshNN = reshape(ZmeshNN,size(Xmesh,1),size(Xmesh,2));
146 % map the interpolator of the test set
147 testInterpolant = TriScatteredInterp(X1(testInd),X2(testInd),T(testInd));
148 ZmeshTest = testInterpolant(Xmesh, Ymesh);
149 % plot it together with the interpolant of the TEST set
150 figure
151 surface(Xmesh, Ymesh, ZmeshNN, 'FaceColor', 'g'); hold on; % plot the NN
152 surface(Xmesh, Ymesh, ZmeshTest, 'FaceColor', 'r'); % plot the interpolator
153 xlabel('x_1'); ylabel('x_2'); zlabel('z');
154 savefig('NN_and_testsurf.fig');
155
156 % make the error plot
157 ZmeshError = ZmeshNN - ZmeshTest;
158 figure
159 contourf(Xmesh, Ymesh, ZmeshError);
160 xlabel('x_1'); ylabel('x_2'); colorbar;
161 h = colorbar; ylabel(h, 'Error');
162 savefig('NN_test_error.fig');
163
164 % compute MSE on the test set
165 mseTest = sum((net(xtest)-ytest).^2)/size(ytest,2);
166 fprintf('MSE on the test set: %.2E\n', mseTest);

```

```

167
168 % plot the regression for the test set
169 figure;
170 plotregression(sim(net, xtest), ytest);
171
172
173
174 %% USE ALL THE DATA TO SEE WHAT HAPPENS, ALSO USE BETTER RATIOS
175 nhfinal = 50;
176 useN = size(T,1);
177 [trainInd, valInd, testInd] = dividerand(useN);
178 % easier variables for training the network
179 % xtrain = [X1(trainInd)'; X2(trainInd)']; % input
180 % ytrain = T(trainInd,:)'; % target
181 % xval = [X1(valInd)'; X2(valInd)'];
182 % yval = T(valInd)';
183 xtest = [X1(testInd)'; X2(testInd)'];
184 ytest = T(testInd)';
185 % train the network
186 net = feedforwardnet(nhfinal, 'trainlm');
187 net.divideFcn = 'divideind'; % also give it access to the other sets for
    plotting
188 net.divideParam.trainInd = trainInd;
189 net.divideParam.valInd = valInd;
190 net.divideParam.testInd = testInd;
191 net.layers{1}.transferFcn = 'tansig'; % hidden layer
192 net.layers{2}.transferFcn = 'purelin'; % output layer
193 net.trainParam.epochs = 20000; % set really high, so it can decide itself
194 net.trainParam.max_fail = 1000; % set really high, so it can decide itself
195 [net, tr] = train(net, X', T', 'UseParallel', 'yes');
196 plotperform(tr)
197 % calculate its output on the meshgrid
198 ZmeshNN = net([Xmesh(:) Ymesh(:)]');
199 ZmeshNN = reshape(ZmeshNN, size(Xmesh,1), size(Xmesh,2));
200 % map the interpolator of the test set
201 testInterpolant = TriScatteredInterp(X1(testInd), X2(testInd), T(testInd));
202 ZmeshTest = testInterpolant(Xmesh, Ymesh);
203 % plot it together with the interpolant of the TEST set
204 figure
205 surface(Xmesh, Ymesh, ZmeshNN, 'FaceColor', 'g'); hold on; % plot the NN
206 surface(Xmesh, Ymesh, ZmeshTest, 'FaceColor', 'r'); % plot the interpolator
207 xlabel('x_1'); ylabel('x_2'); zlabel('z');
208 % make the error plot
209 ZmeshError = ZmeshNN - ZmeshTest;
210 figure
211 contourf(Xmesh, Ymesh, ZmeshError);
212 xlabel('x_1'); ylabel('x_2'); colorbar;
213 h = colorbar; ylabel(h, 'Error');
214 % compute MSE on the test set
215 mseTest = sum((net(xtest)-ytest).^2)/size(ytest,2);
216 fprintf('MSE on the test set: %.2E\n', mseTest);

```

A.2 Classification (see Section 6)

A.2.1 Main code

```

1 clear all; close all; clc;
2
3 %% LOAD THE DATA FROM TOLEDO
4

```

```

5 % generate my data from studentnr 0639870
6 % digit 0 gives me (C+,C-) = (5,6) of the white wine
7 % these classes are in the last column
8 datatable = importdata('data/winequality-white.csv');
9 data = datatable.data;
10 pos = data(data(:,end) == 5,:); % 5 is positive
11 neg = data(data(:,end) == 6,:); % 6 is negative
12 Npos = size(pos,1);
13 Nneg = size(neg,1);
14
15 % put these in more useful training formats (all in one)
16 % input are all columns, except for the last one, which is target
17 % we set the target values to +/-1 instead of 5 and 6
18 X = [pos(:,1:(end-1)) ; neg(:,1:(end-1))]';
19 T = [ones(Npos,1) ; -ones(Nneg,1)]';
20 N = Npos + Nneg;
21
22 % shuffle and divide the data
23 rng(1);
24 [trainInd, valInd, testInd] = dividerand(N); % default is 0.7, 0.15, 0.15
25 %stdX = mapstd(X);
26 stdX=X;
27
28 %% CREATING A NEURAL NETWORK TO CLASSIFY THE DATA
29 % create a network and train it
30 rng(1);
31 net = feedforwardnet(15, 'trainlm');
32 % classification values are between -1 and 1, hence, we can use the tangent
33 % sigmoid function in the output layer as well
34 net.layers{1}.transferFcn = 'tansig'; % hidden layer
35 net.layers{2}.transferFcn = 'tansig'; % output layer
36 net.divideFcn = 'divideind';
37 net.divideParam.trainInd = trainInd;
38 net.divideParam.valInd = valInd;
39 net.divideParam.testInd = testInd;
40 net.trainParam.max_fail = 50; % may vary this
41 net.trainParam.min_grad = 10^-15; % may vary this
42 net = train(net,stdX,T);
43
44 %% PERFORMANCE CHECKS
45 predVal = sim(net, X(:,valInd));
46 CCRval = sum(sign(predVal) == T(valInd))*100/length(valInd);
47 predTest = sim(net, stdX(:,testInd));
48 CCRtest = sum(sign(predTest) == T(testInd))*100/length(testInd);
49 fprintf('CCRval= %f and CCRtest %f .\n', CCRval, CCRtest);
50
51 %% PCA
52 % preprocess the data to get zero mean = 0 and stddev = 1 for all
53 % properties
54 Ntrain = size(X(:,trainInd),2); % number of training data points
55 [~,~,eigvals,~,~,~] = doPCA(X(:,trainInd)',11); % use all 11
56
57 % plot the eigenvalues
58 figure;
59 bar(eigvals/max(eigvals)); hold on; % shows that one needs 'only' 10 basis
    vectors
60 plot(1:11, 1-cumsum(eigvals/sum(eigvals)), 'r-', 'LineWidth', 2)
61 ylabel('\lambda_k'); xlabel('k');
62 axis([0 12 0 1]);
63 savefig('eigenvalues.fig');

```

```

64
65 % we project the vectors onto the restricted eigenbasis (columns of eigvecs)
66 numBasisVecs=8; % choose the number of eigenvectors
67 [PCABasis,redXTrain,eigvals,~,~,stdXTrain] = doPCA(X(:,trainInd)',numBasisVecs
    );
68
69 % project also validation and test set, but first standardize them, use
70 % part of my doPCA function for this
71 % note that we project with the PCA basis of the training set, as required
72 % by the assignment
73 [~,~,~,meanVal,stddevVal,stdXVal] = doPCA(X(:,valInd)',numBasisVecs);
74 [~,~,~,meanTest,stddevTest,stdXTest] = doPCA(X(:,testInd)',numBasisVecs);
75 redXVal = stdXVal*PCABasis;
76 redXTest = stdXTest*PCABasis;
77
78 % now reconstruct
79 redXTrain = redXTrain*PCABasis';
80 redXVal = redXVal*PCABasis';
81 redXTest = redXTest*PCABasis';
82
83 % create a new input matrix X from these components
84 %X = zeros(numBasisVecs,N);
85 X(:,trainInd) = redXTrain';
86 X(:,valInd) = redXVal';
87 X(:,testInd) = redXTest';
88
89 % we can now use them for training
90 %% CREATING A NEURAL NETWORK TO CLASSIFY THE DATA
91 % create a network and train it
92 rng(1);
93 net = feedforwardnet(11, 'trainlm');
94 net.layers{1}.transferFcn = 'tansig'; % hidden layer
95 net.layers{2}.transferFcn = 'tansig'; % output layer
96 net.divideFcn = 'divideind';
97 net.divideParam.trainInd = trainInd;
98 net.divideParam.valInd = valInd;
99 net.divideParam.testInd = testInd;
100 net.trainParam.max_fail = 50; % may vary this
101 net.trainParam.min_grad = 10^-15; % may vary this
102 [net, tr] = train(net,X,T);
103
104 %% PERFORMANCE CHECKS
105 predVal = sim(net, X(:,valInd));
106 CCRval = sum(sign(predVal) == T(valInd))*100/length(valInd);
107 predTest = sim(net, X(:,testInd));
108 CCRtest = sum(sign(predTest) == T(testInd))*100/length(testInd);
109 fprintf('CCRval= %f and CCRtest %f .\n',CCRval,CCRtest);

```

A.2.2 PCA related

```

1 function [E,z,d,meanVec,stddevVec,stdX] = doPCA(x, q)
2 % DOPCA Do a Principle Component Analysis on a data set x
3 % [E,z,mean,stddev] = DOPCA(x,q) with x a dataset with each row a data entry
4 % ,
5 % performs PCA with result z and matrix E.
6 % this means that z is x in a reduced basis
7 % q is the reduced dimension (dimension of z)
8 % d is the eigenvalue vector
9 % meanVec stddevVec contain the mean and standard deviation vector of the
10 % original data

```

```

10 % stdX contains the standardized set in unreduced space
11
12 % get the dimension p of x and the number of datapoints N
13 p = size(x,2);
14 assert( q <= p );
15 N = size(x,1);
16
17 % calculate the mean for all p data properties
18 meanVec = mean(x);
19 stddevVec = std(x);
20 %stddevVec = ones(1,p);
21
22 % rescale and shift with these vectors
23 stdX = zeros(N, 11);
24 for i = 1:N
25     stdX(i,:) = x(i,:) - meanVec; % shift
26     stdX(i,:) = stdX(i,:) ./ stddevVec; % rescale
27 end
28
29 % calculate the covariance matrix of the standardized data set
30 V = cov(stdX);
31
32 % calculate the eigenvectors and eigenvalues
33 % E is the matrix with columns the eigenvectors
34 [E,d] = eig(V);
35 d = diag(d);
36
37 % now sort them in descending order and take only q of them
38 [d, indices] = sort(d, 'descend');
39 d = d(1:q);
40 E = E(:,indices(1:q));
41
42 % reduce the data set by multiplying with this matrix
43 z=stdX*E;

```

A.3 Character recognition (see Section 7)

A.3.1 Main code

```

1 close all; clc; clear all;
2
3 %% CREATE THE ATTRACTOR STATES
4 % get all the letters of the alphabet in CAPITALS
5 [ALPHABET, ~]=prprob();
6
7 % now the unique lower case letters of my name
8 name=GenerateName;
9
10 % add the ALPHABET and name
11 allLetters=[name';ALPHABET']';
12
13 % rescale from -1 to 1 instead of 0 and 1
14 allLetters = 2*allLetters - 1;
15
16 % plot the first 10 letters of my data set
17 figure;
18 colormap(gray);
19 for letterNr=1:10
20     subplot(2,5,letterNr);
21     imagesc(reshape(allLetters(:,letterNr),5,7) ', 'CDataMapping', 'scaled'); % 5

```



```

        x7 bit maps transposed to a 7x5
22     set(gca,'xtick',[]); set(gca,'xticklabel',[]); set(gca,'ytick',[]); set(
        gca,'yticklabel',[]);
23 end
24 savefig('letters.fig');
25 hold off; close all;
26
27 %% CREATE A HOPFIELD RECURRENT NETWORK, TYPE 1
28 % type 1 retrieves 5 first letters
29 T = allLetters(:,1:5);
30 net = newhop(T);
31
32 %% DISTORT AND RETRIEVE IMAGES
33 % check the correct retrieval rate, output states that are spurious
34 % even though we can do the following exactly, we do a
35 % simulation. We create 1000 distorted images of each of the 5 letters and
36 % use the Hopfield network to retrieve the original states. If the number
37 % of attempts is large enough, we should find all spurious states. Note
38 % however that there are  $35!/(3!32!)=6545$  possible distorted images of
39 % each of the letters
40 % Compared to the assignment on Hopfields, the distorted image now has
41 % discrete values for the pixels. Hence, it's now more feasible to end up
42 % in a spurious state
43
44 Nwrong = 0;
45 timesteps = 1000;
46 for letterNr = 1:5
47     fprintf('Start with letter nr : %i\n', letterNr)
48     letter = T(:,letterNr);
49     for it = 1:1000
50         distImage = DistortImage(letter);
51         [Y,~,~] = net({1 timesteps},{},{distImage});
52         if ~isequal(Y{end},letter)
53             if (Nwrong ~= 0 && sum(ismember(Y{end}', wrongStates', 'rows'))
                    ==1) % avoid doubles
54                 fprintf('Same state.\n')
55             else
56                 Nwrong = Nwrong + 1;
57                 fprintf(' Wrong state at iteration: %i\n',it);
58                 wrongStates(:,Nwrong) = Y{end}; % add the state
59                 originalStates(:,Nwrong) = letter;
60             end
61         end
62     end
63 end
64
65 figure;
66 colormap(gray)
67 for wrongNr = 1:Nwrong
68     subplot(2,Nwrong, wrongNr);
69     imagesc(reshape(wrongStates(:,wrongNr),5,7)', 'CDataMapping', 'scaled');
70     hold on;
71     subplot(2,Nwrong,Nwrong+wrongNr);
72     imagesc(reshape(originalStates(:,wrongNr),5,7)', 'CDataMapping', 'scaled');
73     hold on;
74 end
75 hold all;
76 savefig('wrong_states.fig'); hold off;
77
78 %% MAPPING ERROR IFO P

```

```

77 Nit = 100;
78 % store number of wrong results
79 Nwrong = zeros(1,size(allLetters,2));
80 % loop over the number of stored patterns P
81 for P = 1:size(allLetters,2)
82     fprintf('Simulating with P = %i patterns stored.\n',P);
83     % take P attractors
84     T = allLetters(:,1:P);
85     % create a hopfield net
86     net = newhop(T);
87     % loop over Nit distorted images per letter and calculate the error
88     for letterNr = 1:P
89         letter = T(:,P);
90         for it = 1:Nit
91             distImage = DistortImage(letter);
92             % use the net to retrieve
93             [Y,~,~] = net({1 timesteps},{},{distImage});
94
95             % clip
96             Y{end} = sign(Y{end});
97             % check if it's the correct one
98             Nwrong(P) = Nwrong(P) + sum(abs(Y{end} - letter));
99         end
100     end
101     Nwrong(P) = Nwrong(P) / (Nit*P*size(allLetters,1)); % normalize over
        number of states that we generated and nr of pixels
102 end
103
104 % estimate also with Hebb rule
105 figure;
106 sigmas = sqrt((1:size(allLetters,2))/size(allLetters,1));
107 mus = zeros(1,size(allLetters,2));
108 Perr = ones(1,size(allLetters,2))-normcdf(ones(1,size(allLetters,2)),mus,
        sigmas);
109 %plot it
110 plot(1:size(allLetters,2),Nwrong,'r-','LineWidth',2); hold on;
111 plot(1:size(allLetters,2),Perr,'b-','LineWidth',2);
112 legend('Pixel error','P_{error}')
113 ylabel('Error');
114 xlabel('Number of patterns stored');
115 savefig('Error_ifo_P.fig');

```

A.3.2 Image distortion

```

1 function distImage = DistortImage(image)
2
3 % take three random numbers between 1 and 35
4 indices = randsample(35,3); % unique, no replacement
5
6 % now get those pixels and switch them
7 distImage = image;
8 distImage(indices) = -distImage(indices);

```

A.3.3 Name generation

```

1 function lowercasename = GenerateName()
2
3
4 j = [ ...
5     0 0 0 0 0 ...
6     0 0 0 0 1 ...

```

```

7      0 0 0 0 0 ...
8      0 0 0 0 1 ...
9      0 0 0 0 1 ...
10     1 0 0 0 1 ...
11     0 1 1 1 0 ]';
12
13  a = [ ...
14       0 0 0 0 0 ...
15       0 0 0 0 0 ...
16       0 1 1 1 0 ...
17       0 0 0 0 1 ...
18       0 1 1 1 1 ...
19       1 0 0 0 1 ...
20       0 1 1 1 0 ]';
21
22  n = [ ...
23       0 0 0 0 0 ...
24       0 0 0 0 0 ...
25       1 0 1 1 0 ...
26       1 1 0 0 1 ...
27       1 0 0 0 1 ...
28       1 0 0 0 1 ...
29       1 0 0 0 1 ]';
30
31  e = [ ...
32       0 0 0 0 0 ...
33       0 0 0 0 0 ...
34       0 1 1 1 0 ...
35       1 0 0 0 1 ...
36       1 1 1 1 1 ...
37       1 0 0 0 0 ...
38       0 1 1 1 0 ]';
39
40  s = [ ...
41       0 0 0 0 0 ...
42       0 0 0 0 0 ...
43       0 1 1 1 1 ...
44       1 0 0 0 0 ...
45       0 1 1 1 0 ...
46       0 0 0 0 1 ...
47       1 1 1 1 0 ]';
48
49  y = [ ...
50       0 0 0 0 0 ...
51       0 0 0 0 0 ...
52       1 0 0 0 1 ...
53       1 0 0 0 1 ...
54       0 1 1 1 1 ...
55       0 0 0 0 1 ...
56       1 1 1 1 0 ]';
57
58  lowercasename = [j,a,n,e,s,y];

```

A.3.4 Alternative to Hopfield

```

1  close all; clc; clear all;
2
3  %% CREATE THE ATTRACTOR STATES
4  % get all the letters of the alphabet in CAPITALS
5  [ALPHABET, ~]=prprob();

```

```

6 % now the unique lower case letters of my name
7 name=GenerateName;
8 % add the ALPHABET and name
9 allLetters=[name';ALPHABET']';
10 % rescale from -1 to 1 instead of 0 and 1
11 allLetters = 2*allLetters - 1;
12
13 % loop over number of patterns stored
14 MaxNpatt=25;
15 icf = zeros(1,MaxNpatt);
16 %storage for some wrong letters, take one correct and one incorrect per
17 for P=1:size(icf,2)
18     %% CREATE THE FULL SET OF POSSIBILITIES
19     % make the set of the first 25 letters
20     letters = allLetters(:,1:P);
21
22     % make some possibilities of distortion of 3 pixels
23     % this is used for training
24     numDist = 100; % number of distorted images per letter
25     X = zeros(size(letters,1),size(letters,2)*numDist);
26     T = zeros(size(letters,2),size(letters,2)*numDist);
27     for il = 1:size(letters,2)
28         letter = letters(:,il);
29         for id = 1:numDist
30             X(:,(il-1)*numDist+id) = DistortImage(letter);
31             T(il,(il-1)*numDist+id) = 1; % only that one is 1
32         end
33     end
34
35     %% TRAIN A NEURAL NETWORK
36     % create network with 25 neurons
37     net = feedforwardnet(25);
38     net.layers{1}.transferFcn = 'tansig';
39     net.layers{2}.transferFcn = 'softmax';
40     net.trainParam.showWindow=0;
41     net = train(net, X, T, 'useParallel', 'yes'); % 25 dimensional output
42     % all letters have same distance of sqrt2
43
44     %% NOW CHECK ITS CAPABILITIES
45     Nit = 1000;
46     % store number of wrong results
47     Nwrong = zeros(1,size(letters,2));
48
49     % loop over Nit distorted images per letter and calculate the error
50     parfor letterNr = 1:size(letters,2)
51         letter = letters(:,letterNr);
52         fprintf('Starting with letter %i out of %i...\n',letterNr,size(letters,2))
53         for it = 1:Nit
54             distImage = DistortImage(letter);
55             % use the net to retrieve
56             [Y,~,~] = net(distImage);
57
58             % one-hot representation
59             [~,ind] = max(Y);% one-hot encoding
60             ind = ind(1); % just in case there are multiple maxima
61
62             % keep the labels
63             trueLabels(letterNr,it) = letterNr;
64             simLabels(letterNr,it) = ind;

```

```

65
66         % check if it's the correct one
67         Nwrong(letterNr) = Nwrong(letterNr) + (ind ~= letterNr);
68     end
69 end
70 %icf(P) = sum(Nwrong) / (Nit*size(letters,2)*size(letters,1)); % normalize
71 icf(P) = sum(Nwrong) / (Nit*P); % normalize
72 fprintf('%i patterns give a normalized image error of %f\n',P,icf(P));
73 end
74
75 figure;
76 plot(1:P,icf,'r-','LineWidth',2);
77 xlabel('Number of patterns stored'); ylabel('Error');
78 savefig('alternative_error.fig');
79
80 % THIS PART IS TO INTENSIVE FOR MY PC
81 % %% CASE OF 10 PATTERNS
82 % letters = allLetters(:,1:10);
83 % numDist = 100; % number of distorted images per letter
84 % X = zeros(size(letters,1),size(letters,2)*numDist);
85 % T = zeros(size(letters,2),size(letters,2)*numDist);
86 % for il = 1:size(letters,2)
87 %     letter = letters(:,il);
88 %     for id = 1:numDist
89 %         X(:,(il-1)*numDist+id) = DistortImage(letter);
90 %         T(il,(il-1)*numDist+id) = 1; % only that one is 1
91 %     end
92 % end
93 %
94 % % create network with lots of neurons
95 % net = feedforwardnet(50);
96 % net.layers{1}.transferFcn = 'tansig';
97 % net.layers{2}.transferFcn = 'softmax';
98 % net = train(net, X, T, 'UseParallel', 'yes');
99 %
100 % Nit = 1000;
101 % % store number of wrong results
102 % Nwrong = zeros(1,size(letters,2));
103 % % we will make the confusion matrix later
104 % trueLabels = zeros(size(letters,2),Nit);
105 % simLabels = zeros(size(letters,2),Nit);
106 %
107 % % loop over Nit distorted images per letter and calculate the error
108 % parfor letterNr = 1:size(letters,2)
109 %     letter = letters(:,letterNr);
110 %     fprintf('Starting with letter %i out of %i...\n',letterNr,size(letters
111 % ,2))
112 %     for it = 1:Nit
113 %         distImage = DistortImage(letter);
114 %         % use the net to retrieve
115 %         [Y,~,~] = net(distImage);
116 %         % one-hot representation
117 %         [~,ind] = max(Y); % one-hot encoding
118 %         ind = ind(1); % just in case there are multiple maxima
119 %
120 %         % keep the labels
121 %         trueLabels(letterNr,it) = letterNr;
122 %         simLabels(letterNr,it) = ind;
123 %

```

```
124 %           % check if it's the correct one
125 %
126 %           Nwrong(letterNr) = Nwrong(letterNr) + (ind ~= letterNr);
127 %       end
128 % end
129 %
130 %
131 % fprintf('%i patterns give a normalized image error of %f\n',25,sum(Nwrong) /
132 %       (Nit*25));
133 % % confusion matrix
134 % trueLabels = reshape(trueLabels,[1 size(trueLabels,1)*size(trueLabels,2)]);
135 % simLabels = reshape(simLabels,[1 size(simLabels,1)*size(simLabels,2)]);
136 % plotconfusion(ind2vec(trueLabels),ind2vec(simLabels))
```