# 1 PROJECT 1: Regression of function data

The data stems from a non linear function. Hence, a non-linear transfer function is required. Bearing in mind the results of Lehno *et al.*, I 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, I 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 spacial structure dimension in addition to the fact that the data has not been jittered, the number of neurons can be taken to be relatively high without overfitting. To get an estimate of the number of neurons, we refer to Fig. 1. We note that by considering the validation set performance, the optimal number of neurons is between $25 - 40$. Note that at this stage, we may not consider the test set performance. We opt for 35 neurons. Note that to generate Fig. 1, we did not use the validation set in a stopping criterion. This is because we want to *detect* overfitting, rather than *avoiding* it. The latter is the subject of the next paragraph.

First, we discuss the purpose of the 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 to overfit 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 increases for a specified number of iterations (`net.trainParam.max_fail` = 20), 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.

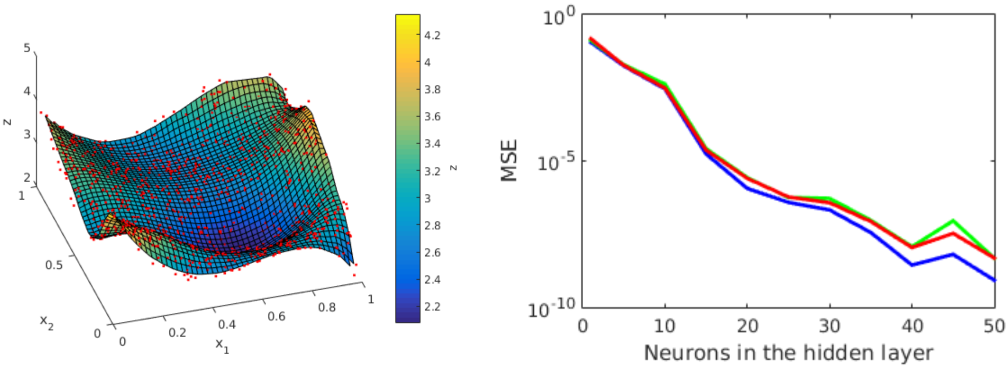


Figure 1: Left: surface and data points in the training set. Right: Performance on the training, validation and test set as a function of the number of neurons in the hidden layer.

The MSE on the test set of the resulting NN is $5.5 \times 10^{-5}$. We observe the pattern that the network has slightly higher values when the test set interpolation is large, and smaller values when the interpolations function is small. Hence, more extremal values are obtained.

To improve the results, we can use all of the provided data, instead of a subset of 3000. Also, the data division does not need to be equal for all sets. As a rule of thumb,

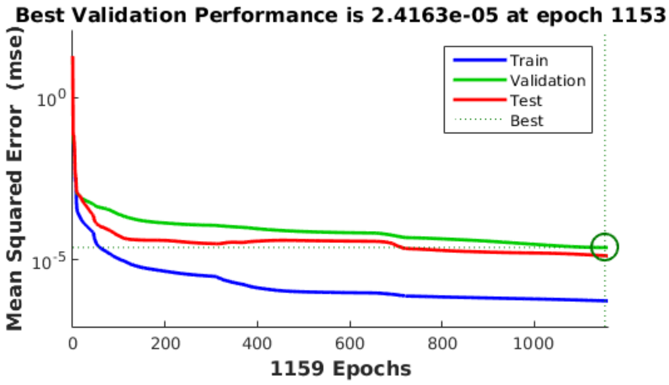


Figure 2: Performance on the training, validation and test set as a function of the number of epochs in the training process.

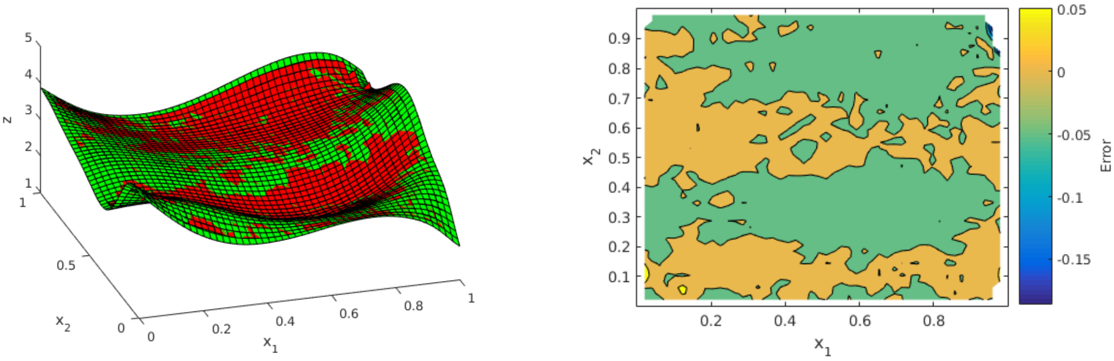


Figure 3: Left: Neural network (green) and test data (red) surface. The surfaces are almost indistinguishable. Right: Contour plot of the test set error.

one generally assumes 70:15:15 (or 60:20:20) for the train:validation:test set ratios. This changes the MSE test error to $4.1 \times 10^{-8}$.

# 2 PROJECT 2: Classification of wine data

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 $\pm 1$. We rescale the attributes to standardized values in order to make it independent of the scale of the data attributes. 15 neurons in the hidden layer is optimal to classify the validation data. Using this architecture, a CCR of 0.60 and 0.72 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.

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. 4. Prior to computing the covariance matrix, we again rescale the data such that all attributes have zero mean and unit standard deviation. The eigenvalue spectrum does not show a clear dominant principle component. Hence, there is no clear indication that the data lives on a low-dimensional subspace. We take a reconstruction error of 10%, which corresponds with an 8 dimensional basis. 11 hidden neurons minimize the CCR of the validation set, which is less than before, as expected. Using this approach, a validation

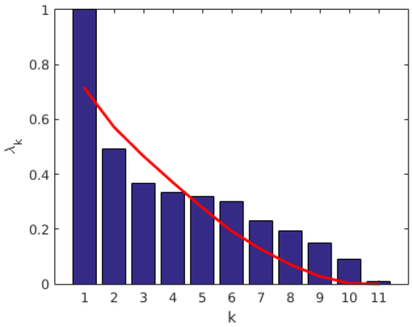


Figure 4: Eigenvalues of the $11 \times 11$ covariance matrix. The red line indicates the cumulative sum of remaining (k+1):end eigenvalues, which is proportional to the reconstruction error.

and test set CCR of 0.75 and 0.72 is obtained. Hence, there is a significant CCR increase for the validation set, while test set CCR is almost constant.

Hence, by projecting the data onto a lower dimensional space, a significant classification improvement of approximately 14% of the validation set is realized. However, the result is not true in general, since such a large increase is not found for the test set. This is related to the fact that the principle component basis is that of the training set, and is not updated if new data is included.

We remark that the possibility of overfitting is very real in this classification assignment.

# 3 PROJECT 3: Character recognition

## 3.1 Hopfield network

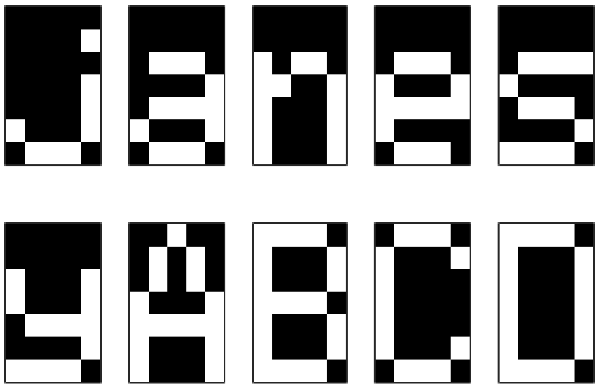

Figure 5: First 10 of the 32 letters that are used in the character recognition exercise.

In this section we train a Hopfield network to reconstruct the states in Fig. 5 from a distorted version of these characters. Hopfield networks are generated by providing it the undistorted images.

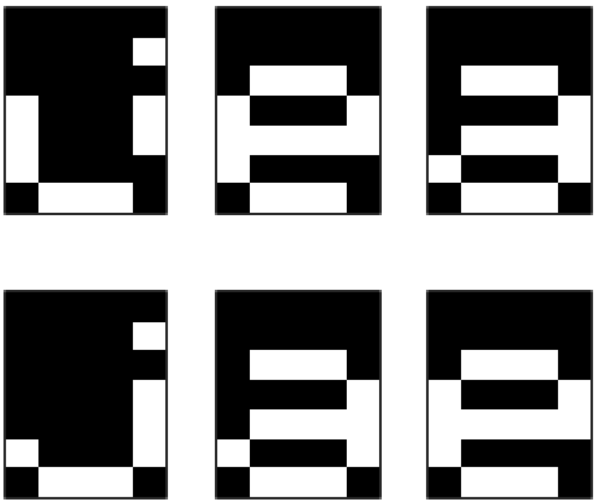

Figure 6: Incorrectly reconstructed characters (top row) with corresponding input (bottom row).

Figure 6 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.

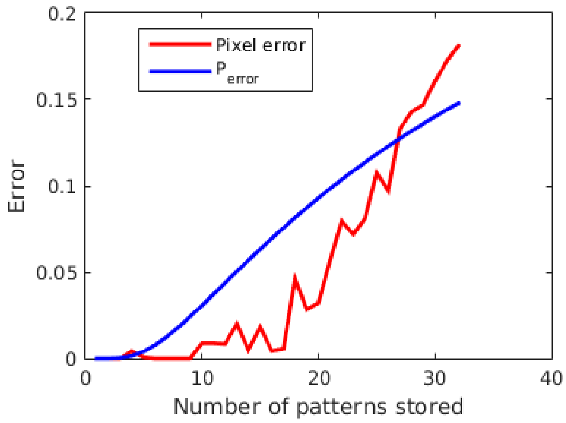


Figure 7: Total pixel error (red),normalized over the number of states generated and total pixels in an image (35), and the theoretical prediction curve based on the Hebb rule (blue).

We now determine how number of patterns that are 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. 7. On 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 distorted pixels is increased, this number is expected to go down. 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 Hebbs' rule for uncorrelated patters. The prediction curve is also shown in Fig. 7. If we assume a large $P$ and large $N$, we obtain an estimated $P_{\text{error}}$

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

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

$$P_{\max} = N/(4 \log N) = 2.46 \,. \tag{2}$$

Hence, for 2 or less stored patterns, the Hopfield should be able to perfectly reconstruct the distorted images.

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 is the obvious alternative to the current Hopfield network. In the next subsection we discuss another alternative.

## 3.2 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 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 [1].

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 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.

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, 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. 8. A steady rise is observed as expected.

Naturally, the networks' 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.

[1] `http://nl.mathworks.com/help/nnet/examples/character-recognition.html`

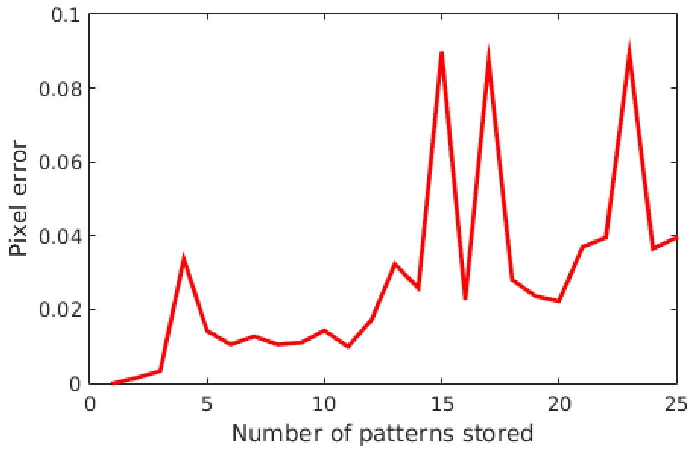


Figure 8: Percentage of false reconstructed images (not pixels as before) as a function of the number of patters stored.

# A Matlab code that implements the solutions

## A.1 Regression (see Section 0.1)

### A.1.1 Main code

```
clear all; close all; clc;

%% LOAD THE DATA FROM TOLEDO

% load the data from Toledo
load 'data/Data_Problem1_regression.mat'

% generate my data from studentnr 0639870
d1 = 9;
d2 = 8;
d3 = 7;
d4 = 6;
d5 = 3;
T = (d1*T1 + d2*T2 + d3*T3 + d4*T4 + d5*T5)/(d1 + d2 + d3 + d4 + d5);
X = [X1 X2]; % useful for running sim
N = size(T,1);

% a check
assert((size(X1,1) == N) && (size(X2,1) == N));

%% DIVIDE THE DATA IN SUBSETS

% randomize the data, shuffle it, also transpose it
shuffledInd = randperm(size(T,1));
X1 = X1(shuffledInd);
X2 = X2(shuffledInd);
T = T(shuffledInd);

% divide the data in subsets (use the first 3000, they are already
```

```
    shuffled)
rng(1);
useN = 3000;
[trainInd,valInd,testInd] = dividerand(useN,1/3,1/3,1/3);

% easier variables for training the network
xtrain = [X1(trainInd)'; X2(trainInd)'];% input
ytrain = T(trainInd,:)'; % target
xval = [X1(valInd)'; X2(valInd)'];
yval = T(valInd)';
xtest = [X1(testInd)'; X2(testInd)'];
ytest = T(testInd)';

%% VIZUALISE THE DATA
% create a regular grid for the interpolant (which is not the net)
ndim = 50;
[Xmesh, Ymesh, ~, ~] = meshinterpolate(X1, X2, T, ndim);
% now interpolate
trainInterpolate = TriScatteredInterp(X1(trainInd), X2(trainInd), T(
    trainInd)); % plot the interpolator
Zmesh = trainInterpolate(Xmesh, Ymesh);
% show the surface
figure;
surface(Xmesh, Ymesh, Zmesh); hold on;
% also show the data
scatter3(X1(trainInd), X2(trainInd), T(trainInd), '.', '
    MarkerEdgeColor','red', 'MarkerFaceColor', 'red');
xlabel('x_1'); ylabel('x_2'); zlabel('z');
colorbar; h = colorbar; ylabel(h, 'z');
savefig('train_surface.fig'); hold off;

% show that the data is well distributed and divided randomly
figure;
surface(Xmesh, Ymesh, Zmesh); hold on; % plot the interpolator
scatter3(X1(trainInd), X2(trainInd), T(trainInd), 15, '
    MarkerFaceColor','g'); hold on;
scatter3(X1(valInd), X2(valInd), T(valInd), 15, 'MarkerFaceColor',[1
    .5 0]); hold on;
scatter3(X1(testInd), X2(testInd), T(testInd), 15, 'MarkerFaceColor',
    'r'); hold off;

%% CREATE THE VALIDATION PLOT

% create some vectors to plot
nhvals = [1,5,10,15,20,25,30,35,40,45,50];
mseVal = zeros(length(nhvals),1);
mseTrain = zeros(length(nhvals),1);
mseTest = zeros(length(nhvals),1);

for nhIt = 1:length(nhvals)

```

```
    % change the number of neurons in the hidden layer
    nh = nhvals(nhIt);
    disp(['Training with ' num2str(nh) ' neurons in hidden layer']);

    rng(1); % same seed for all

    % train the network
    net = feedforwardnet(nh,'trainlm');
    net.divideFcn = 'dividetrain'; % Use the whole training set for
       training
    net.layers{1}.transferFcn = 'tansig';
    net.layers{2}.transferFcn = 'purelin';
    net.trainParam.showWindow=0;
    net.trainParam.epochs = 5000;

    % train the network
    [net,tr] = train(net, xtrain, ytrain, 'UseParallel', 'yes');
    nntraintool('close');

    % show number of epochs used
    disp(['Used ' num2str(tr.num_epochs) ' epochs']);

    % make predictions for the data sets
    ytrainPred = sim(net, xtrain);
    yvalPred = sim(net, xval);
    ytestPred = sim(net, xtest); % plot this for completeness

    % calculate the mse for these points
    perfTrain = perform(net,ytrain,ytrainPred);
    perfVal   = perform(net,yval,yvalPred);
    perfTest = perform(net,ytest,ytestPred);

    % store it
    mseTrain(nhIt) = perfTrain;
    mseVal(nhIt) = perfVal;
    mseTest(nhIt) = perfTest;
end

%% SELECT A NETWORK ARCHITECTURE FROM THE RESULTS
nhfinal = 35;

%% SHOW PERFORMANCE ON VALIDATION SET
figure
semilogy(nhvals, mseTrain, 'Color','b', 'LineWidth',2); hold on;
semilogy(nhvals, mseVal, 'Color','g', 'LineWidth',2); hold on;
semilogy(nhvals, mseTest, 'Color','r', 'LineWidth',2);
xlabel('Neurons in the hidden layer'); ylabel('MSE');
savefig('performance_val.fig');

%% PLOT THE RESULT OF THE OPTED ARCHITECTURE
% train the network
```

```
rng(1);
net = feedforwardnet(nhfinal,'trainlm');
net.divideFcn = 'divideind'; % also give it access to the other sets for plotting
net.divideParam.trainInd = trainInd;
net.divideParam.valInd = valInd;
net.divideParam.testInd = testInd;
net.layers{1}.transferFcn = 'tansig'; % hidden layer
net.layers{2}.transferFcn = 'purelin'; % output layer
net.trainParam.epochs = 5000; % set really high, so it can decide itself
net.trainParam.max_fail = 50; % set really high, so it can decide itself
[net,tr] = train(net, xtrain, ytrain);

% calculate its output on the meshgrid
ZmeshNN = net([Xmesh(:) Ymesh(:)]');
ZmeshNN = reshape(ZmeshNN,size(Xmesh,1),size(Xmesh,2));

% map the interpolator of the test set
testInterpolant = TriScatteredInterp(X1(testInd),X2(testInd),T(testInd));
ZmeshTest = testInterpolant(Xmesh, Ymesh);

% plot it together with the interpolant of the TEST set
figure
surface(Xmesh, Ymesh, ZmeshNN,'FaceColor', 'g'); hold on; % plot the NN
surface(Xmesh, Ymesh, ZmeshTest, 'FaceColor','r'); % plot the interpolator
xlabel('x_1'); ylabel('x_2'); zlabel('z');
savefig('NN_and_testsurf.fig');

% make the error plot
ZmeshError = ZmeshNN - ZmeshTest;
figure
contourf(Xmesh, Ymesh, ZmeshError);
xlabel('x_1'); ylabel('x_2'); colorbar;
h = colorbar; ylabel(h, 'Error');
savefig('NN_test_error.fig');

% compute MSE on the test set
mseTest = sum((net(xtest)-ytest).^2)/size(ytest,2);
fprintf('MSE on the test set: %.2E\n', mseTest);

% plot the regression for the test set
figure;
plotregression(sim(net, xtest),ytest);
```

```
%% USE ALL THE DATA TO SEE WHAT HAPPENS, ALSO USE BETTER RATIOS
rng(1);
useN = size(T,1);
[trainInd,valInd,testInd] = dividerand(useN);
% easier variables for training the network
xtrain = [X1(trainInd)'; X2(trainInd)'];% input
ytrain = T(trainInd,:)'; % target
xval = [X1(valInd)'; X2(valInd)'];
yval = T(valInd)';
xtest = [X1(testInd)'; X2(testInd)'];
ytest = T(testInd)';
% train the network
rng(1);
net = feedforwardnet(nhfinal,'trainlm');
net.divideFcn = 'divideind'; % also give it access to the other sets
    for plotting
net.divideParam.trainInd = trainInd;
net.divideParam.valInd = valInd;
net.divideParam.testInd = testInd;
net.layers{1}.transferFcn = 'tansig'; % hidden layer
net.layers{2}.transferFcn = 'purelin'; % output layer
net.trainParam.epochs = 5000; % set really high, so it can decide
    itself
net.trainParam.max_fail = 50; % set really high, so it can decide
    itself
[net,tr] = train(net, xtrain, ytrain, 'UseParallel','yes');
% calculate its output on the meshgrid
ZmeshNN = net([Xmesh(:) Ymesh(:)]');
ZmeshNN = reshape(ZmeshNN,size(Xmesh,1),size(Xmesh,2));
% map the interpolator of the test set
testInterpolant = TriScatteredInterp(X1(testInd),X2(testInd),T(
    testInd));
ZmeshTest = testInterpolant(Xmesh, Ymesh);
% plot it together with the interpolant of the TEST set
figure
surface(Xmesh, Ymesh, ZmeshNN,'FaceColor', 'g'); hold on; % plot the
    NN
surface(Xmesh, Ymesh, ZmeshTest, 'FaceColor','r'); % plot the
    interpolator
xlabel('x_1'); ylabel('x_2'); zlabel('z');
% make the error plot
ZmeshError = ZmeshNN - ZmeshTest;
figure
contourf(Xmesh, Ymesh, ZmeshError);
xlabel('x_1'); ylabel('x_2'); colorbar;
h = colorbar; ylabel(h, 'Error');
% compute MSE on the test set
mseTest = sum((net(xtest)-ytest).^2)/size(ytest,2);
fprintf('MSE on the test set: %.2E\n', mseTest);
```

## A.2 Classification (see Section 0.2)

### A.2.1 Main code

```
clear all; close all; clc;

%% LOAD THE DATA FROM TOLEDO

% generate my data from studentnr 0639870
% digit 0 gives me (C+,C-) = (5,6) of the white wine
% these classes are in the last column
datatable = importdata('data/winequality-white.csv');
data = datatable.data;
pos = data(data(:,end) == 5,:); % 5 is positive
neg = data(data(:,end) == 6,:); % 6 is negative
Npos = size(pos,1);
Nneg = size(neg,1);

% put these in more useful training formats (all in one)
% input are all columns, except for the last one, which is target
% we set the target values to +/-1 instead of 5 and 6
X = [pos(:,1:(end-1)) ; neg(:,1:(end-1))]';
T = [ones(Npos,1) ; -ones(Nneg,1)]';
N = Npos + Nneg;

% shuffle and divide the data
rng(1);
[trainInd, valInd, testInd] = dividerand(N); % default is 0.7, 0.15, 0.15
stdX = mapstd(X);


%% CREATING A NEURAL NETWORK TO CLASSIFY THE DATA
% create a network and train it
rng(1);
net = feedforwardnet(15, 'trainlm');
% classification values are between -1 and 1, hence, we can use the tangent
% sigmoid function in the output layer as well
net.layers{1}.transferFcn = 'tansig'; % hidden layer
net.layers{2}.transferFcn = 'tansig'; % output layer
net.divideFcn = 'divideind';
net.divideParam.trainInd = trainInd;
net.divideParam.valInd = valInd;
net.divideParam.testInd = testInd;
net.trainParam.max_fail = 50; % may vary this
net.trainParam.min_grad = 10^-15; % may vary this
net = train(net,stdX,T);

%% PERFORMANCE CHECKS
predVal = sim(net, X(:,valInd));
CCRval = sum(sign(predVal) == T(valInd))*100/length(valInd);
```

```
predTest = sim(net, stdX(:,testInd));
CCRtest = sum(sign(predTest) == T(testInd))*100/length(testInd);
fprintf('CCRval= %f and CCRtest %f .\n',CCRval,CCRtest);

%% PCA
% preprocess the data to get zero mean = 0 and stddev = 1 for all
% properties
Ntrain = size(X(:,trainInd),2); % number of training data points
[~,~,eigvals,~,~,~] = doPCA(X(:,trainInd)',11); % use all 11

% plot the eigenvalues
figure;
bar(eigvals/max(eigvals)); hold on; % shows that one needs 'only' 10 basis vectors
plot(1:11, 1-cumsum(eigvals/sum(eigvals)), 'r-', 'LineWidth', 2)
ylabel('\lambda_k'); xlabel('k');
axis([0 12 0 1]);
savefig('eigenvalues.fig');

% we project the vectors onto the restricted eigenbasis (columns of eigvecs)
numBasisVecs=8; % choose the number of eigenvectors
[PCABasis,redXTrain,eigvals,meanTrain,stddevTrain,stdXTrain] = doPCA(X(:,trainInd)',numBasisVecs);

% project also validation and test set, but first standardize them, use
% part of my doPCA function for this
% note that we project with the PCA basis of the training set, as required
% by the assignment
[~,~,~,meanVal,stddevVal,stdXVal] = doPCA(X(:,valInd)',numBasisVecs);
[~,~,~,meanTest,stddevTest,stdXTest] = doPCA(X(:,testInd)',numBasisVecs);
redXVal = stdXVal*PCABasis;
redXTest = stdXTest*PCABasis;

% now reconstruct
redXTrain = redXTrain*PCABasis';
redXVal = redXVal*PCABasis';
redXTest = redXTest*PCABasis';

% create a new input matrix X from these components
%X = zeros(numBasisVecs,N);
X(:,trainInd) = redXTrain';
X(:,valInd) = redXVal';
X(:,testInd) = redXTest';

% we can now use them for training
%% CREATING A NEURAL NETWORK TO CLASSIFY THE DATA
% create a network and train it
```

```
rng(1);
net = feedforwardnet(11, 'trainlm');
net.layers{1}.transferFcn = 'tansig'; % hidden layer
net.layers{2}.transferFcn = 'tansig'; % output layer
net.divideFcn = 'divideind';
net.divideParam.trainInd = trainInd;
net.divideParam.valInd = valInd;
net.divideParam.testInd = testInd;
net.trainParam.max_fail = 50; % may vary this
net.trainParam.min_grad = 10^-15; % may vary this
[net, tr] = train(net,X,T);

%% PERFORMANCE CHECKS
predVal = sim(net, X(:,valInd));
CCRval = sum(sign(predVal) == T(valInd))*100/length(valInd);
predTest = sim(net, X(:,testInd));
CCRtest = sum(sign(predTest) == T(testInd))*100/length(testInd);
fprintf('CCRval= %f and CCRtest %f .\n',CCRval,CCRtest);
```

### A.2.2 PCA related

```
function [E,z,d,meanVec,stddevVec,stdX] = doPCA(x, q)
% DOPCA Do a Principle Component Analysis on a data set x
%   [E,z,mean,stddev] = DOPCA(x,q) with x a dataset with each row a
    data entry,
%   performs PCA with result z and matrix E.
%   this means that z is x in a reduced basis
%   q is the reduced dimension (dimension of z)
%   d is the eigenvalue vector
%   meanVec stddevVec contain the mean and standard deviation vector
    of the
%   original data
%   stdX contains the standardized set in unreduced space

% get the dimension p of x and the number of datapoints N
p = size(x,2);
assert( q <= p );
N = size(x,1);

% calculate the mean for all p data properties
meanVec = mean(x);
stddevVec = std(x);
%stddevVec = ones(1,p);

% rescale and shift with these vectors
stdX = zeros(N, 11);
for i = 1:N
    stdX(i,:) = x(i,:) - meanVec;         % shift
    stdX(i,:) = stdX(i,:) ./ stddevVec;   % rescale
end

% calculate the covariance matrix of the standardized data set
```

```
V = cov(stdX);

% calculate the eigenvectors and eigenvalues
% E is the matrix with columns the eigenvectors
[E,d] = eig(V);
d = diag(d);

% now sort them in descending order and take only q of them
[d, indices] = sort(d, 'descend');
d = d(1:q);
E = E(:,indices(1:q));

% reduce the data set my multiplying with this matrix
z=stdX*E;
```

### A.3 Classification (see Section 0.2)

#### A.3.1 Main code

```
close all; clc; clear all;

%% CREATE THE ATTRACTOR STATES
% get all the letters of the alphabet in CAPITALS
[ALPHABET, ~]=prprob();

% now the unique lower case letters of my name
name=GenerateName;

% add the ALPHABET and name
allLetters=[name';ALPHABET']';

% rescale from -1 to 1 instead of 0 and 1
allLetters = 2*allLetters - 1;

% plot the first 10 letters of my data set
figure;
colormap(gray);
for letterNr=1:10
    subplot(2,5,letterNr);
    imagesc(reshape(allLetters(:,letterNr),5,7)','CDataMapping','
        scaled'); % 5x7 bit maps transposed to a 7x5
    set(gca,'xtick',[]); set(gca,'xticklabel',[]); set(gca,'ytick'
        ,[]); set(gca,'yticklabel',[]);
end
savefig('letters.fig');
hold off; close all;

%% CREATE A HOPFIELD RECURRENT NETWORK, TYPE 1
% type 1 retrieves 5 first letters
T = allLetters(:,1:5);
net = newhop(T);

```

```matlab
%% DISTORT AND RETRIEVE IMAGES
% check the correct retrieval rate, output states that are spurious
% even though we can do the following exactly, we do a
% simulation. We create 1000 distorted images of each of the 5 letters and
% use the Hopfield network to retrieve the original states.  If the number
% of attempts is large enoug, we should find all spurious states. Note
% however that there are 35!/(3!32!)= 6545 possible distorted images of
% each of the letters
% Compared to the assignment on Hopfields, the distorted image now has
% discrete values for the pixels. Hence, it's now more feasible to end up
% in a spurious state

Nwrong = 0;
timesteps = 1000;
for letterNr = 1:5
    fprintf('Start with letter nr : %i\n', letterNr)
    letter = T(:,letterNr);
    for it = 1:1000
        distImage = DistortImage(letter);
        [Y,~,~] = net({1 timesteps},{},{distImage});
        if ~isequal(Y{end},letter)
            if (Nwrong ~= 0 && sum(ismember(Y{end}', wrongStates', 'rows'))==1) % avoid doubles
                fprintf('Same state.\n')
            else
                Nwrong = Nwrong + 1;
                fprintf(' Wrong state at iteration: %i\n',it);
                wrongStates(:,Nwrong) = Y{end}; % add the state
                originalStates(:,Nwrong) = letter;
            end
        end
    end
end

figure;
colormap(gray)
for wrongNr = 1:Nwrong
    subplot(2,Nwrong,wrongNr);
    imagesc(reshape(wrongStates(:,wrongNr),5,7)','CDataMapping','scaled'); hold on;
    subplot(2,Nwrong,Nwrong+wrongNr);
    imagesc(reshape(originalStates(:,wrongNr),5,7)','CDataMapping','scaled'); hold on;
end
hold all;
```

```
savefig('wrong_states.fig'); hold off;

%% MAPPING ERROR IFO P
Nit = 100;
% store number of wrong results
Nwrong = zeros(1,size(allLetters,2));
% loop over the number of stored patterns P
for P = 1:size(allLetters,2)
    fprintf('Simulating with P = %i patterns stored.\n',P);
    % take P attractors
    T = allLetters(:,1:P);
    % create a hopfield net
    net = newhop(T);
    % loop over Nit distored images per letter and calculate the error
    for letterNr = 1:P
        letter = T(:,P);
        for it = 1:Nit
            distImage = DistortImage(letter);
            % use the net to retrieve
            [Y,~,~] = net({1 timesteps},{},{distImage});

            % clip
            Y{end} = sign(Y{end});
            % check if it's the correct one
            Nwrong(P) = Nwrong(P) + sum(abs(Y{end} - letter));
        end
    end
    Nwrong(P) = Nwrong(P) / (Nit*P*size(allLetters,1)); % normalize over number of states that we generated and nr of pixels
end

% estimate also with Hebb rule
figure;
sigmas = sqrt((1:size(allLetters,2))/size(allLetters,1));
mus = zeros(1,size(allLetters,2));
Perr = ones(1,size(allLetters,2))-normcdf(ones(1,size(allLetters,2)), mus,sigmas);
%plot it
plot(1:size(allLetters,2),Nwrong,'r-','LineWidth',2); hold on;
plot(1:size(allLetters,2),Perr,'b-','LineWidth',2);
legend('Pixel error','P_{error}')
ylabel('Error');
xlabel('Number of patterns stored');
savefig('Error_ifo_P.fig');
```

### A.3.2 Image distortion

```
function distImage = DistortImage(image)

% take three random numbers between 1 and 35
indices = randsample(35,3); % unique, no replacement
```

```

% now get those pixels and switch them
distImage = image;
distImage(indices) = -distImage(indices);
```

### A.3.3 Name generation

```
function lowercasename = GenerateName()


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

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

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

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

s =  [...
     0 0 0 0 0 ...
     0 0 0 0 0 ...
     0 1 1 1 1 ...
     1 0 0 0 0 ...
     0 1 1 1 0 ...
```

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

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

lowercasename = [j,a,n,e,s,y];
```

### A.3.4 Alternative to Hopfield

```
close all; clc; clear all;

%% CREATE THE ATTRACTOR STATES
% get all the letters of the alphabet in CAPITALS
[ALPHABET, ~]=prprob();
% now the unique lower case letters of my name
name=GenerateName;
% add the ALPHABET and name
allLetters=[name';ALPHABET']';
% rescale from -1 to 1 instead of 0 and 1
allLetters = 2*allLetters - 1;

% loop over number of patterns stored
MaxNpatt=25;
icf = zeros(1,MaxNpatt);
%storage for some wrong letters, take one correct and one incorrect
    per
for P=1:size(icf,2)
    %% CREATE THE FULL SET OF POSSIBILITIES
    % make the set of the first 25 letters
    letters = allLetters(:,1:P);

    % make some possibilities of distortion of 3 pixels
    % this is used for training
    numDist = 100; % number of distorted images per letter
    X = zeros(size(letters,1),size(letters,2)*numDist);
    T = zeros(size(letters,2),size(letters,2)*numDist);
    for il = 1:size(letters,2)
        letter = letters(:,il);
        for id = 1:numDist
            X(:,(il-1)*numDist+id) = DistortImage(letter);
            T(il,(il-1)*numDist+id) = 1; % only that one is 1
        end
    end

    %% TRAIN A NEURAL NETWORK
```

```
    % create network with 25 neurons
    net = feedforwardnet(25);
    net.layers{1}.transferFcn = 'tansig';
    net.layers{2}.transferFcn = 'softmax';
    net.trainParam.showWindow=0;
    net = train(net, X, T,'useParallel','yes'); % 25 dimensional output
    % all letters have same distance of sqrt2

    %% NOW CHECK ITS CAPABILIIES
    Nit = 1000;
    % store number of wrong results
    Nwrong = zeros(1,size(letters,2));

    % loop over Nit distored images per letter and calculate the error
    parfor letterNr = 1:size(letters,2)
        letter = letters(:,letterNr);
        fprintf('Starting with letter %i out of %i...\n',letterNr, size(letters,2))
        for it = 1:Nit
            distImage = DistortImage(letter);
            % use the net to retrieve
            [Y,~,~] = net(distImage);

            % one-hot representation
            [~,ind] = max(Y);% one-hot encoding
            ind = ind(1); % just in case there are multiple maxima

            % keep the labels
            trueLabels(letterNr,it) = letterNr;
            simLabels(letterNr,it) = ind;

            % check if it's the correct one
            Nwrong(letterNr) = Nwrong(letterNr) + (ind ~= letterNr);
        end
    end
    %icf(P) = sum(Nwrong) / (Nit*size(letters,2)*size(letters,1)); % normalize
    icf(P) = sum(Nwrong) / (Nit*P); % normalize
    fprintf('%i patterns give a normalized image error of %f\n',P,icf(P));
end

figure;
plot(1:P,icf,'r-','LineWidth',2);
xlabel('Number of patterns stored'); ylabel('Pixel error');
savefig('alternative_error.fig');

% THIS PART IS TO INTENSIVE FOR MY PC
% %% CASE OF 10 PATTERNS
```

```
% letters = allLetters(:,1:10);
% numDist = 100; % number of distorted images per letter
% X = zeros(size(letters,1),size(letters,2)*numDist);
% T = zeros(size(letters,2),size(letters,2)*numDist);
% for il = 1:size(letters,2)
%     letter = letters(:,il);
%     for id = 1:numDist
%         X(:,(il-1)*numDist+id) = DistortImage(letter);
%         T(il,(il-1)*numDist+id) = 1; % only that one is 1
%     end
% end
%
% % create network with lots of neurons
% net = feedforwardnet(50);
% net.layers{1}.transferFcn = 'tansig';
% net.layers{2}.transferFcn = 'softmax';
% net = train(net, X, T, 'UseParallel', 'yes');
%
% Nit = 1000;
% % store number of wrong results
% Nwrong = zeros(1,size(letters,2));
% % we will make the confusion matrix later
% trueLabels = zeros(size(letters,2),Nit);
% simLabels = zeros(size(letters,2),Nit);
%
% % loop over Nit distored images per letter and calculate the error
% parfor letterNr = 1:size(letters,2)
%     letter = letters(:,letterNr);
%     fprintf('Starting with letter %i out of %i...\n',letterNr,size(
    letters,2))
%     for it = 1:Nit
%         distImage = DistortImage(letter);
%         % use the net to retrieve
%         [Y,~,~] = net(distImage);
%
%         % one-hot representation
%         [~,ind] = max(Y);% one-hot encoding
%         ind = ind(1); % just in case there are multiple maxima
%
%         % keep the labels
%         trueLabels(letterNr,it) = letterNr;
%         simLabels(letterNr,it) = ind;
%
%         % check if it's the correct one
%
%         Nwrong(letterNr) = Nwrong(letterNr) + (ind ~= letterNr);
%     end
% end
%
%
% fprintf('%i patterns give a normalized image error of %f\n',25,sum(
```

```
Nwrong) / (Nit*25));
%
%% confusion matrix
% trueLabels = reshape(trueLabels,[1 size(trueLabels,1)*size(trueLabels,2)]);
% simLabels = reshape(simLabels,[1 size(simLabels,1)*size(simLabels,2)]);
% plotconfusion(ind2vec(trueLabels),ind2vec(simLabels))
```