**6. Appendix: Matlab Scripts**

**6.1. Exercise: Supervise learning and generalization**

%% creating the data set for a sinus 2-period wave

% training size data\_size = 100;

% the jump size

jump\_size = 1/(data\_size) \* (4\*pi);

% creating the X array to store the data size in

X = zeros(1,data\_size);

% creating x values for i = 1:data\_size

X(i) = i\*jump\_size;

end

% creating the y-values of the sinus wave

Y = sin(X);

% number of neurons

number\_neurons = [5, 10, 20, 40, 100];

% different training algorithms

training\_algs = { 'traingd', 'traingda', 'traincgf', 'traincgp', 'trainbfg', 'trainlm', 'trainbr'};

% matrix to store speed values in -> speed\_values

speed\_values = zeros(length(training\_algs), length(number\_neurons));

RMSE\_training\_average\_error = zeros(length(training\_algs), length(number\_neurons)); RMSE\_test\_average\_error = zeros(length(training\_algs), length(number\_neurons));

% number of iterarions for each execution n\_iterations = 40;

%% Number Neurons

% index for speed\_values matrix

i = 0;

j= 0;

for train\_alg = training\_algs

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

for n\_number = number\_neurons

% creating the feedforward neural network train\_alg = char(train\_alg);

net = feedforwardnet(n\_number, train\_alg);

% array to store the calculation times over the 20 iterations calc\_time\_values = zeros(1,n\_iterations);

% array to store the RMSE training errors over the 20 iterations

RMSE\_training\_values = zeros(1,n\_iterations);

% array to store the RMSE test errors over the 20 iterations

RMSE\_test\_values = zeros(1,n\_iterations);

for it = 1:n\_iterations

% proportion of rows to select for training p = 0.8;

% create logical index vector training\_vector = false(1, data\_size); training\_vector(1:round(p\*data\_size)) = true;

% randomise order

training\_vector = training\_vector(randperm(data\_size));

% training set

X\_train = X(training\_vector);

% test set

X\_test = X(~training\_vector);

% Y-values of the training and test set

Y\_train = sin(X\_train); Y\_test = sin(X\_test);

% training the neural network and measuring average calculation time tic;

net = train(net, X\_train, Y\_train);

calc\_time = toc;

% the calculation time values over 20 iterations on average calc\_time\_values(it) = calc\_time;

% RMSE training set

RMSE\_training\_values(it) = sqrt(sum((Y\_train - sim(net, X\_train)).^2));

% RMSE test set

RMSE\_test\_values(it) = sqrt(sum((Y\_test - sim(net, X\_test)).^2));

end

% average calculation time

average\_calc\_time = mean(calc\_time\_values);

% average RMSE training error

average\_RMSE\_training = mean(RMSE\_training\_values);

% average RMSE test error

average\_RMSE\_test = mean(RMSE\_test\_values);

% updating the column index that corresponds to the amount of noise

% added

j = j + 1;

% adding the calculation time values to the matrix speed\_values(i,j) = average\_calc\_time;

% adding RMSE training error values to the matrix

RMSE\_training\_average\_error(i,j) = average\_RMSE\_training;

% adding the RMSE test error to the matrix

RMSE\_test\_average\_error(i,j) = average\_RMSE\_test;

end

end

j = 0;

%% Bar Plot Speed

% returning a barplot for the Speed names = categorical(training\_algs); barplot = bar(names, speed\_values);

title("Effect of Amount Hidden Neurons on Calculation Time");

ylabel("Calculation Time")

legend\_bar = legend(barplot,{"5", "10", "20", "40", "100"});

title(legend\_bar, "Hidden Neurons");

number\_neurons = [1, 5, 10, 20, 40, 100];

ylim([0 5]);

%% Bar Plot Training RMSE

% returning a barplot for the RMSE Training Error

names = categorical(training\_algs);

barplot = bar(names, RMSE\_training\_average\_error);

title("Effect of Amount Hidden Neurons on Training Error");

ylabel("RMSE Training Error")

legend\_bar = legend(barplot,{"5", "10", "20", "40", "100"});

title(legend\_bar, "Hidden Neurons");

%% Bar Plot Test RMSE

% returning a barplot for the RMSE Test Error names = categorical(training\_algs);

barplot = bar(names, RMSE\_test\_average\_error); title("Effect of Amount Hidden Neurons on Test Error"); ylabel("RMSE Test Error")

legend\_bar = legend(barplot,{"5", "10", "20", "40", "100"});

title(legend\_bar, "Hidden Neurons");

%% creating the data set for a sinus 2-period wave

% training size data\_size = 100;

% the jump size

jump\_size = 1/(data\_size) \* (4\*pi);

% creating the X array to store the data size in

X = zeros(1,data\_size);

% creating x values for i = 1:data\_size

X(i) = i\*jump\_size;

end

% creating the y-values of the sinus wave

Y = sin(X);

% different training algorithms

training\_algs = { 'traingd', 'traingda', 'traincgf', 'traincgp', 'trainbfg', 'trainlm', 'trainbr'};

% the increasing noise steps

noise\_steps = [0.1, 0.2, 0.3, 0.4, 0.5];

% matrices to store the speed and error values in

speed\_values = zeros(length(training\_algs), length(noise\_steps));

RMSE\_training\_average\_error = zeros(length(training\_algs), length(noise\_steps)); RMSE\_test\_average\_error = zeros(length(training\_algs), length(noise\_steps));

% number of iterarions for each execution n\_iterations = 40;

%% Noise

% index for speed\_values matrix

i = 0;

j= 0;

for train\_alg = training\_algs

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

for noise\_step = noise\_steps

% creating the feedforward neural network train\_alg = char(train\_alg);

net = feedforwardnet(10, train\_alg);

% array to store the calculation times over the 20 iterations calc\_time\_values = zeros(1,n\_iterations);

% array to store the RMSE training errors over the 20 iterations

RMSE\_training\_values = zeros(1,n\_iterations);

% array to store the RMSE test errors over the 20 iterations

RMSE\_test\_values = zeros(1,n\_iterations);

for it = 1:n\_iterations

% proportion of rows to select for training p = 0.8;

% create logical index vector training\_vector = false(1, data\_size); training\_vector(1:round(p\*data\_size)) = true;

% randomise order

training\_vector = training\_vector(randperm(data\_size));

% training set

X\_train = X(training\_vector);

% test set

X\_test = X(~training\_vector);

% Y-values of the training and test set

Y\_train = sin(X\_train); Y\_test = sin(X\_test);

Y\_train\_noise = zeros(1, length(Y\_train));

% adding random noise to the training set for n = 1:length(data\_size)

Y\_train\_noise(n) = Y\_train(n) + noise\_step \* normrnd(0,1);

end

% training the neural network and measuring average calculation time tic;

net = train(net, X\_train, Y\_train\_noise);

calc\_time = toc;

% the calculation time values over 20 iterations on average calc\_time\_values(it) = calc\_time;

% RMSE training set

RMSE\_training\_values(it) = sqrt(sum((Y\_train\_noise - sim(net, X\_train)).^2));

% RMSE test set

RMSE\_test\_values(it) = sqrt(sum((Y\_test - sim(net, X\_test)).^2));

end

% average calculation time

average\_calc\_time = mean(calc\_time\_values);

% average RMSE training error

average\_RMSE\_training = mean(RMSE\_training\_values);

% average RMSE test error

average\_RMSE\_test = mean(RMSE\_test\_values);

% updating the column index that corresponds to the amount of noise

% added

j = j + 1;

% adding the calculation time values to the matrix speed\_values(i,j) = average\_calc\_time;

% adding RMSE training error values to the matrix

RMSE\_training\_average\_error(i,j) = average\_RMSE\_training;

% adding the RMSE test error to the matrix

RMSE\_test\_average\_error(i,j) = average\_RMSE\_test;

end

end

j = 0;

%% Bar Plot Speed

% returning a barplot for the Speed

names = categorical(training\_algs); barplot = bar(names,speed\_values); title("Effect of Noise on Calculation Time"); ylabel("Calculation Time")

legend\_bar = legend(barplot,{"O.1", "O.2", "0.3", "0.4", "0.5"});

title(legend\_bar, "Noise Step");

%% Bar Plot Training RMSE

% returning a barplot for the RMSE Training Error names = categorical(training\_algs);

barplot = bar(names, RMSE\_training\_average\_error); title("Effect of Noise on Training Error"); ylabel("RMSE Training Error")

legend\_bar = legend(barplot,{"O.1", "O.2", "0.3", "0.4", "0.5"});

title(legend\_bar, "Noise Step");

%% Bar Plot Test RMSE

% returning a barplot for the RMSE Test Error

names = categorical(training\_algs);

barplot = bar(names, RMSE\_test\_average\_error);

title("Effect of Noise on Test Error");

ylabel("RMSE Test Error")

legend\_bar = legend(barplot,{"O.1", "O.2", "0.3", "0.4", "0.5"});

title(legend\_bar, "Noise Step");

**6.2. Exercise: Recurrent neural networks**

**6.2.1. Hopfield net**

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

number = number + 1;

%loading digit data and preparation

% loading the digits dataset load digits

% getting the numbr of observations and number of variables

[n\_observations, n\_variables] = size(X);

% Values must be +1 or -1 in order for the Hopfield networks to function

X(X==0)=-1;

%Attractors of the Hopfield network -> the correct digits of the Hopfield

%network

% index 1 - 20 -> digit 0

zero = X(1,:);

% index 21 - 40 -> digit 1

one = X(21,:);

% index 41 - 60 -> digit 2

two = X(41,:);

% index 61 - 80 -> digit 3

three = X(61,:);

% index 81 - 100 -> digit 4

four = X(81,:);

% index 101 - 120 -> digit 5 five = X(101,:);

% ...

six = X(121,:);

seven = X(141,:); eight = X(161,:); nine = X(181,:);

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

% Intializing Hopfield net

% defining the right attractors

Attractor\_T = [zero;one;two;three;four;five;six;seven;eight;nine]';

% the number of different digits num\_dig = size(Attractor\_T,2);

% creating the Hopfield network with initial starting digits as attractor

% states

net = newhop(Attractor\_T);

% Check if initial digits are attractors [Y,~,~] = sim(net,num\_dig,[],Attractor\_T); Y = Y';

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

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

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

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

% 10 different digit observations from the orginal digits data set

X\_10 = X(index\_dig,:);

%%

for l = 1:4

noiselevel = noises(l);

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

for a = 1:10

% choosing certain observation that needs to be checked observations = X\_10(a,:);

for n = 1:100

% adding noise

observations\_noise = observations;

for k=1:size(observations,1)

% adding noise taking from the Gaussian standard distribution observations\_noise(k,:) = observations(k,:) + noiselevel\*normrnd(0,1);

end

for i = 1:size(observations\_noise,1)

% certain observed digit

digit\_input = observations\_noise(i,:);

digit\_input = digit\_input';

% getting the best classification output fromt he Hopfield network

T = {digit\_input};

[digit\_output,~,~] = sim(net,{1 ,iterations},{},T); digit\_output = digit\_output{1, iterations}; digit\_output = digit\_output';

% the array to store the digits to check in digits\_to\_check = Attractor\_T';

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

for j = 1:10

if(isequal(digit\_output, digits\_to\_check(j,:)) == true)

digit\_classifications(a,j) = digit\_classifications(a,j) + 1;

t = 1;

end

end

if(t == 0)

digit\_classifications(a,11) = digit\_classifications(a,11) + 1;

% plotting the spurious state hold on

digit = reshape(digit\_output,15,16)';

imshow(digit)

figure;

end

end

end

% Amounts Missclassified

n\_classified = digit\_classifications(a,a);

assigned\_spurious = digit\_classifications(a,11); digit\_classified(a,l) = n\_classified; digit\_spurious(a,l) = assigned\_spurious;

end end

digit\_classified digit\_spurious

%% Barplot of the (mis)classifications

categories = {'0', '1', '2', '3', '4', '5', '6', '7', '8', '9'};

names = categorical(categories);

barplot = bar(names,digit\_classified);

title("Effect of Noise on Number of Correct Classifications ");

ylabel("Number Correct Classifications")

legend\_bar = legend(barplot,{"1", "2", "4", "10"});

title(legend\_bar, "Noise Step");

%% barplot of the amount of digits of certain type assigned to spurious vectors categories = {'0', '1', '2', '3', '4', '5', '6', '7', '8', '9'};

names = categorical(categories);

barplot = bar(names,digit\_spurious);

title("Effect of Noise on Spurious States Assignment");

ylabel("Number Correct Classifications")

legend\_bar = legend(barplot,{"1", "2", "4", "10"});

title(legend\_bar, "Noise Step");

ylim([0 6])

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

number = number + 1;

%loading digit data and preparation

% loading the digits dataset load digits

% getting the numbr of observations and number of variables

[n\_observations, n\_variables] = size(X);

% Values must be +1 or -1 in order for the Hopfield networks to function

X(X==0)=-1;

%Attractors of the Hopfield network -> the correct digits of the Hopfield

%network

% index 1 - 20 -> digit 0

zero = X(1,:);

% index 21 - 40 -> digit 1

one = X(21,:);

% index 41 - 60 -> digit 2

two = X(41,:);

% index 61 - 80 -> digit 3

three = X(61,:);

% index 81 - 100 -> digit 4 four = X(81,:);

% index 101 - 120 -> digit 5 five = X(101,:);

% ...

six = X(121,:); seven = X(141,:); eight = X(161,:); nine = X(181,:);

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

% Intializing Hopfield net

% defining the right attractors

Attractor\_T = [zero;one;two;three;four;five;six;seven;eight;nine]';

% the number of different digits num\_dig = size(Attractor\_T,2);

% creating the Hopfield network with initial starting digits as attractor

% states

net = newhop(Attractor\_T);

% Check if initial digits are attractors [Y,~,~] = sim(net,num\_dig,[],Attractor\_T); Y = Y';

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

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

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

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

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

% 10 different digit observations from the orginal digits data set

X\_10 = X(index\_dig,:);

%%

for l = 1:4

iterations = time\_steps(l);

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

for a = 1:10

% choosing certain observation that needs to be checked observations = X\_10(a,:);

for n = 1:100

% adding noise

observations\_noise = observations;

for k=1:size(observations,1)

% adding noise taking from the Gaussian standard distribution observations\_noise(k,:) = observations(k,:) + noiselevel\*normrnd(0,1);

end

for i = 1:size(observations\_noise,1)

% certain observed digit

digit\_input = observations\_noise(i,:);

digit\_input = digit\_input';

% getting the best classification output from the Hopfield network

T = {digit\_input};

[digit\_output,~,~] = sim(net,{1 ,iterations},{},T); digit\_output = digit\_output{1, iterations}; digit\_output = digit\_output';

% the array to store the digits to check in digits\_to\_check = Attractor\_T';

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

for j = 1:10

if(isequal(digit\_output, digits\_to\_check(j,:)) == true)

digit\_classifications(a,j) = digit\_classifications(a,j) + 1;

t = 1;

end

end

if(t == 0)

digit\_classifications(a,11) = digit\_classifications(a,11) + 1;

% plotting the spurious state

% hold on

% digit = reshape(digit\_output,15,16)';

% imshow(digit)

% figure;

end

end

end

% Amounts Missclassified

n\_classified = digit\_classifications(a,a);

assigned\_spurious = digit\_classifications(a,11); digit\_classified(a,l) = n\_classified; digit\_spurious(a,l) = assigned\_spurious;

end end

digit\_classified digit\_spurious

%% Barplot of the (mis)classifications

categories = {'0', '1', '2', '3', '4', '5', '6', '7', '8', '9'};

names = categorical(categories);

barplot = bar(names,digit\_classified);

title("Effect of Time Steps on Number of Correct Classifications ");

ylabel("Number Correct Classifications")

legend\_bar = legend(barplot,{"10", "50", "100", "1000"});

title(legend\_bar, "Time Steps");

%% barplot of the amount of digits of certain type assigned to spurious vectors categories = {'0', '1', '2', '3', '4', '5', '6', '7', '8', '9'};

names = categorical(categories);

barplot = bar(names,digit\_spurious);

title("Effect of Time Steps on Spurious States Assignment");

ylabel("Number Correct Classifications")

legend\_bar = legend(barplot,{"10", "50", "100", "1000"});

title(legend\_bar, "Time Steps");

**6.2.2. Elman network to model Hammerstein time series**

%% Set the parameters of the run

n = 2000; % Total number of samples

ne = 1000; % Number of epochs

perc\_training = 0.7; % Number between 0 and 1. The validation set will be 1-perc\_training.

if perc\_training >= 1 || perc\_training <= 0

error('The training set is ill defined. The variable perc\_training should be between 0 and 1')

end

%% Create the samples

% Allocate memory u = zeros(1, n);

x = zeros(1, n);

y = zeros(1, n);

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

% Calculate the samples for i=2:n

u(i)=randn;

end

x(i)=.8\*x(i-1)+sin(u(i));

y(i)=x(i);

%% matrix to store values in

% first amount of hidden neurons hidden\_neurons = [1 5 10 20 40];

% the matrix to store the results in

% each row represents a certain metric -> first row has the speed, second

% RSME test error

results = zeros(2, length(hidden\_neurons));

% number of iterations to get a stable metric n\_iterations = 40;

%% Results for different number of hidden neurons j =0;

for number\_neurons = 1:5

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

% array to store the calculation times over the 20 iterations calc\_time\_values = zeros(1,n\_iterations);

% array to store the RMSE test errors over the 20 iterations

R2\_values = zeros(1,n\_iterations);

for it = 1:n\_iterations

% proportion of rows to select for training-validation p = 0.8;

% create logical index vector training\_vector = false(1, n); training\_vector(1:round(p\*n)) = true;

% randomise order

training\_vector = training\_vector(randperm(n));

% training set

X = u(training\_vector);

% test set

X\_test = u(~training\_vector);

% Y-values of the training and test set

T = y(training\_vector);

T\_test = y(~training\_vector);

% changing the number of neurons

net = newelm(X, T, hidden\_neurons(number\_neurons));

net.trainParam.epochs = ne; % Number of epochs net.divideParam.testRatio = 0;

net.divideParam.valRatio = 1-perc\_training; % validation set ratio net.divideParam.trainRatio = perc\_training; % training set ratio

tic

net = train(net,X,T); % Training

calc\_time = toc;

T\_test\_sim = sim(net,X\_test); % Testing

% the Rsquared values T\_test = T\_test'; T\_test\_sim = T\_test\_sim';

% training predictions T\_sim = sim(net,X); T\_sim = T\_sim';

% correlation

% corr = corrcoef(T\_test, T\_test\_sim);

corr = corrcoef(T, T\_sim);

% R squared

Rsquared = (corr(1,2))^2;

% storing the calculation over 40 iterations calc\_time\_values(it) = calc\_time;

% storing test RMSE values over 40 iterations

R2\_values(it) = Rsquared;

end

% adding speed values to results matrixvalues to the matrix results(1,j) = mean(calc\_time\_values);

% adding the RMSE test error to the matrix results(2,j) = mean(R2\_values);

end

%% Barplot for Calculation for the different hidden neurons

% returning a barplots for different amounts of hidden neurons

names = categorical({'1 neuron', '5 neurons', '10 neurons', '20 neurons', '40 neurons'});

names = reordercats(names,{'1 neuron', '5 neurons', '10 neurons', '20 neurons', '40 neurons'});

time\_values = results(1,:);

bar(names,time\_values);

title("Effect of Hidden Neurons on Calculation Time");

%% Barplot for the R squared for the different hidden neurons

% returning a barplots for different amounts of hidden neurons

names = categorical({'1 neuron', '5 neurons', '10 neurons', '20 neurons', '40 neurons'});

names = reordercats(names,{'1 neuron', '5 neurons', '10 neurons', '20 neurons', '40 neurons'});

r\_values = results(2,:);

bar(names,r\_values);

title("Effect of Hidden Neurons on R-squared");

%% Set the parameters of the run

ne = 1000; % Number of epochs

perc\_training = 0.7; % Number between 0 and 1. The validation set will be 1-perc\_training.

number\_neurons = 5; % Keeping number of hidden neurons constant

if perc\_training >= 1 || perc\_training <= 0

error('The training set is ill defined. The variable perc\_training should be between 0 and 1')

end

% number of samples

samples = [50 100 500 1000 10000];

% matrix to store values in

% the matrix to store the results in

% each row represents a certain metric -> first row has the speed, second

% R squared

results = zeros(2, length(samples));

%%

for z = 1:5

n = samples(z);

% Create the samples

% Allocate memory u = zeros(1, n);

x = zeros(1, n);

y = zeros(1, n);

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

% Calculate the samples for i=2:n

u(i)=randn;

x(i)=.8\*x(i-1)+sin(u(i));

y(i)=x(i);

end

% number of iterations to get a stable metric n\_iterations = 40;

% array to store the calculation times over the 20 iterations calc\_time\_values = zeros(1,n\_iterations);

% array to store the RMSE test errors over the 20 iterations

R2\_values = zeros(1,n\_iterations);

for it = 1:n\_iterations

% proportion of rows to select for training-validation p = 0.8;

% create logical index vector training\_vector = false(1, n); training\_vector(1:round(p\*n)) = true;

% randomise order

training\_vector = training\_vector(randperm(n));

% training set

X = u(training\_vector);

% test set

X\_test = u(~training\_vector);

% Y-values of the training and test set

T = y(training\_vector);

T\_test = y(~training\_vector);

% changing the number of neurons

net = newelm(X, T, hidden\_neurons(number\_neurons));

net.trainParam.epochs = ne; % Number of epochs net.divideParam.testRatio = 0;

net.divideParam.valRatio = 1-perc\_training; % validation set ratio net.divideParam.trainRatio = perc\_training; % training set ratio

tic

net = train(net,X,T); % Training

calc\_time = toc;

T\_test\_sim = sim(net,X\_test); % Testing

% the Rsquared values T\_test = T\_test'; T\_test\_sim = T\_test\_sim';

% training predictions T\_sim = sim(net,X); T\_sim = T\_sim';

% correlation

% corr = corrcoef(T\_test, T\_test\_sim);

corr = corrcoef(T\_test, T\_test\_sim);

% R squared

Rsquared = (corr(1,2))^2;

% storing the calculation over 40 iterations calc\_time\_values(it) = calc\_time;

% storing test RMSE values over 40 iterations

R2\_values(it) = Rsquared;

end

% adding speed values to results matrix values to the matrix results(1,z) = mean(calc\_time\_values);

% adding the RMSE test error to the matrix results(2,z) = mean(R2\_values);

end

%% Barplot for Calculation for different numbers of samples

% returning a barplots for different numbers of samples

names = categorical({'50 samples', '100 samples', '500 samples', '1000 samples', '10000 samples'});

names = reordercats(names,{'50 samples', '100 samples', '500 samples', '1000 samples', '10000 samples'});

time\_values = results(1,:);

bar(names,time\_values);

title("Effect of Number of Samples on Calculation Time");

%% Barplot for the R squared for different numbers of samples

% returning a barplots for different number of samples

names = categorical({'50 samples', '100 samples', '500 samples', '1000 samples', '10000 samples'}); names = reordercats(names,{'50 samples', '100 samples', '500 samples', '1000 samples', '10000 samples'});

r\_values = results(2,:);

bar(names,r\_values);

title("Effect of Number of Samples on R-squared");

%% Set the parameters of the run

perc\_training = 0.7; % Number between 0 and 1. The validation set will be 1-perc\_training.

number\_neurons = 5; % Keeping number of hidden neurons constant n = 2000;

if perc\_training >= 1 || perc\_training <= 0

error('The training set is ill defined. The variable perc\_training should be between 0 and 1')

end

% number of samples

epochs = [10 100 1000 2000 5000];

% matrix to store values in

% the matrix to store the results in

% each row represents a certain metric -> first row has the speed, second

% R squared

results = zeros(2, length(samples));

% Create the samples

% Allocate memory

u = zeros(1, n); x = zeros(1, n); y = zeros(1, n);

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

% Calculate the samples for i=2:n

u(i)=randn;

x(i)=.8\*x(i-1)+sin(u(i));

y(i)=x(i);

end

%%

for z = 1:5

ne = epochs(z);

% number of iterations to get a stable metric n\_iterations = 40;

% array to store the calculation times over the 20 iterations calc\_time\_values = zeros(1,n\_iterations);

% array to store the RMSE test errors over the 20 iterations

R2\_values = zeros(1,n\_iterations);

for it = 1:n\_iterations

% proportion of rows to select for training-validation p = 0.8;

% create logical index vector training\_vector = false(1, n); training\_vector(1:round(p\*n)) = true;

% randomise order

training\_vector = training\_vector(randperm(n));

% training set

X = u(training\_vector);

% test set

X\_test = u(~training\_vector);

% Y-values of the training and test set

T = y(training\_vector);

T\_test = y(~training\_vector);

% changing the number of neurons

net = newelm(X, T, hidden\_neurons(number\_neurons));

net.trainParam.epochs = ne; % Number of epochs net.divideParam.testRatio = 0;

net.divideParam.valRatio = 1-perc\_training; % validation set ratio net.divideParam.trainRatio = perc\_training; % training set ratio

tic

net = train(net,X,T); % Training calc\_time = toc;

T\_test\_sim = sim(net,X\_test); % Testing

% the Rsquared values T\_test = T\_test'; T\_test\_sim = T\_test\_sim';

% training predictions T\_sim = sim(net,X); T\_sim = T\_sim';

% correlation

% corr = corrcoef(T\_test, T\_test\_sim);

corr = corrcoef(T\_test, T\_test\_sim);

% R squared

Rsquared = (corr(1,2))^2;

% storing the calculation over 40 iterations calc\_time\_values(it) = calc\_time;

% storing test RMSE values over 40 iterations

R2\_values(it) = Rsquared;

end

% adding speed values to results matrixvalues to the matrix results(1,z) = mean(calc\_time\_values);

% adding the RMSE test error to the matrix results(2,z) = mean(R2\_values);

end

%% Barplot for Calculation for the different hidden neurons

% returning a barplots for different amounts of hidden neurons

names = categorical({'10 epochs', '100 epochs', '1000 epochs', '2000 epochs', '5000 epochs'});

names = reordercats(names,{'10 epochs', '100 epochs', '1000 epochs', '2000 epochs', '5000 epochs'});

time\_values = results(1,:);

bar(names,time\_values);

title("Effect of Number of Epochs on Calculation Time");

%% Barplot for the R squared for the different hidden neurons

% returning a barplots for different amounts of hidden neurons

names = categorical({'10 epochs', '100 epochs', '1000 epochs', '2000 epochs', '5000 epochs'});

names = reordercats(names,{'10 epochs', '100 epochs', '1000 epochs', '2000 epochs', '5000 epochs'});

r\_values = results(2,:);

bar(names,r\_values);

title("Effect of Number of Epochs on R-squared");

**6.3. Exercise: Unsupervised learning: PCA and SOM**

**6.3.1. PCA on handwritten digits**

%% load

load threes -ascii

%% Total sum of all eigenvalues

% creating observations set x = threes;

% rows are the observations, columns the variables cov\_matrix = cov(x);

% amount of dimensions

% determining sum of all eigenvalues n\_dimensions = size(cov\_matrix,1);

[all\_v, all\_d] = eigs(cov\_matrix,n\_dimensions);

total\_eigen\_values = sum(diag(all\_d));

%% Eigenvalue decomposition x\_plot = 1:256;

proportion\_values = zeros(1,256);

error\_values = zeros(1,256);

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

% returning the eigenvectors of k largest eigenvalues

[v,d] = eigs(cov\_matrix,k);

% PROPORTION EXPLAINED

% proportion explained by k principal components -> eigenvalues

proportion\_eigen\_values = sum(diag(d));

% proportion of variance explained

proportion\_value = proportion\_eigen\_values / total\_eigen\_values;

% RMSE ERROR OF RECONSTRUCTION MATRIX

transformation = v;

% transforming the original data with k principal components as dimensions x\_transformed = x \* transformation;

% try to reconstruct the original data\_matrix x\_estimated = x\_transformed \* (transformation'); RMSE\_error = sqrt(mean(mean((x - x\_estimated).^2)));

% adding values to arrays proportion\_values(k) = proportion\_value;

error\_values(k) = RMSE\_error;

end

%% Plot proportion variance explained and RMSE errors on one plot graph\_plot = plot(x\_plot, proportion\_values, '--', x\_plot, error\_values); xlabel("Number Principal Components");

legend\_bar = legend(graph\_plot,{"Proportion Variance Explained", "RMSE"});

%% Additional proportion explained going from 1-50 principal components and from 50-100 principal components

proption\_1 = proportion\_values(50) - proportion\_values(1)

proportion\_2 = proportion\_values(100) - proportion\_values(50)

%% creating reconstruction image for the first observation for 1, 50, 100 and 256 principal components

% returning the eigenvectors of k largest eigenvalues array\_pc = [1, 50, 100, 256];

for i = 1:4

k = array\_pc(i);

% constructing eigenvectors and eigenvalues of the original covariance matrix

[v,d] = eigs(cov\_matrix,k);

% RMSE ERROR OF RECONSTRUCTION MATRIX

transformation = v;

% transforming the original data with k principal components as dimensions

x\_transformed = x \* transformation;

% try to reconstruct the original data\_matrix

x\_estimated = x\_transformed \* (transformation');

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

end

clear clc

**6.3.2. SOM on Iris datasets**

close all

%% looking at different topologies

topologies = {'hextop', 'gridtop', 'randtop'};

distance\_functions = {'linkdist', 'dist', 'mandist'};

results = zeros(3,3);

%%

for t = 1:3

top = topologies(t);

for d = 1:3

dis = distance\_functions(d);

top =char(top);

dis = char(dis);

% Load data load iris

X = iris(:,1:end-1);

true\_labels = iris(:,end);

% Training the SOM x\_length = 3; y\_length = 1;

gridsize=[y\_length x\_length];

net = newsom(X',gridsize,top,dis);

net.trainParam.epochs = 1000;

net = train(net,X');

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

%Compare clusters with true labels

ARI = RandIndex(assignment,true\_labels);

results(t,d) = ARI;

end

end

%% returning barplot for Rand index for topolgy and distance function

% returning a barplot for the RMSE Test Error

names = categorical({'hextop', 'gridtop', 'randtop'});

barplot = bar(names, results);

title("Impact of Topology and Distance Function");

ylabel("Rand Index")

legend\_bar = legend(barplot,{'linkdist', 'dist', 'mandist'});

title(legend\_bar, "Distance Function");

%% Different number epochs

epochs = [10, 50, 100, 200, 500, 1000, 10000];

results = zeros(1,7);

%% Rand index for different number of epochs for i = 1:7

e = epochs(i);

% Load data load iris

X = iris(:,1:end-1);

true\_labels = iris(:,end);

% Training the SOM x\_length = 3; y\_length = 1;

gridsize=[y\_length x\_length];

net = newsom(X',gridsize,'hextop', 'mandist');

net.trainParam.epochs = e;

net = train(net,X');

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

%Compare clusters with true labels

ARI = RandIndex(assignment,true\_labels);

results(i) = ARI;

end

%% barplot

names = categorical({'10 epochs', '50 epochs', '100 epochs','200 epochs', '500 epochs', '1000 epochs', '10000 epochs'});

names = reordercats(names,{'10 epochs', '50 epochs', '100 epochs','200 epochs', '500 epochs', '1000 epochs', '10000 epochs'});

barplot = bar(names, results); title("Impact of Epochs") ylabel("Rand Index");

**6.4. Exercise: Deep Learning: Stacked Autoencoders and**

**Convolutional Neural Networks**

**6.4.1. Digit classification with Stacked Autoencoders**

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

% Load the training data into memory

%[xTrainImages, tTrain] = digittrain\_dataset;

load('digittrain\_dataset');

hidden\_neurons\_1 = [20, 50, 100, 150, 200];

hidden\_neurons\_2 = [10, 20, 50, 100];

%% Multiple loops to store the results speed\_results = zeros(4,5); accuracy\_results = zeros(4,5);

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

speed\_values\_nofinetuning = zeros(1,5); speed\_values\_finetuned = zeros(1,5); speed\_values\_pattern1 = zeros(1,5); speed\_values\_pattern2 = zeros(1,5);

accuracy\_values\_nofinetuning = zeros(1,5); accuracy\_values\_finetuned = zeros(1,5); accuracy\_values\_pattern1 = zeros(1,5); accuracy\_values\_pattern2 = zeros(1,5);

for j = 1:5

% certain amount of hidden neurons n1 = hidden\_neurons\_1(n);

% Neural networks have weights randomly initialized before training.

% Therefore the results from training are different each time. To avoid

% this behavior, explicitly set the random number generator seed.

rng('default')

% Layer 1 hiddenSize1 = n1; tic

autoenc1 = trainAutoencoder(xTrainImages,hiddenSize1, ...

'MaxEpochs',200, ...

'L2WeightRegularization',0.004, ...

'SparsityRegularization',4, ...

'SparsityProportion',0.15, ...

'ScaleData', false);

toc\_1 = toc;

feat1 = encode(autoenc1,xTrainImages);

% Layer 2 hiddenSize2 = 50; tic

autoenc2 = trainAutoencoder(feat1,hiddenSize2, ...

'MaxEpochs',200, ...

'L2WeightRegularization',0.002, ...

'SparsityRegularization',4, ...

'SparsityProportion',0.1, ...

'ScaleData', false);

toc\_2 = toc;

feat2 = encode(autoenc2,feat1);

% Layer 3 tic

softnet = trainSoftmaxLayer(feat2,tTrain,'MaxEpochs',200);

toc\_3 = toc;

% Deep Net

deepnet = stack(autoenc1,autoenc2,softnet);

% Test deep net imageWidth = 28; imageHeight = 28;

inputSize = imageWidth\*imageHeight;

%[xTestImages, tTest] = digittest\_dataset;

load('digittest\_dataset');

xTest = zeros(inputSize,numel(xTestImages));

for i = 1:numel(xTestImages)

xTest(:,i) = xTestImages{i}(:);

end

y = deepnet(xTest);

%

accuracy\_values\_nofinetuning(1,j) = 100\*(1-confusion(tTest,y));

%

% Speed of the deep net without fine tuning

speed\_values\_nofinetuning(1,j) = toc\_1 + toc\_2 + toc\_3;

% Test fine-tuned deep net

xTrain = zeros(inputSize,numel(xTrainImages));

for i = 1:numel(xTrainImages)

xTrain(:,i) = xTrainImages{i}(:);

end tic

deepnet = train(deepnet,xTrain,tTrain);

toc\_4 = toc;

y = deepnet(xTest);

%

accuracy\_values\_finetuned(1,j) = 100\*(1-confusion(tTest,y));

%

speed\_values\_finetuned(1,j) = toc\_1 + toc\_2 + toc\_3 + toc\_4;

%Compare with normal neural network (1 hidden layers)

net = patternnet(n1);

net.trainParam.epochs = 200; tic net=train(net,xTrain,tTrain);

speed\_values\_pattern1(1,j) = toc;

y=net(xTest);

plotconfusion(tTest,y);

accuracy\_values\_pattern1(1,j) = 100\*(1-confusion(tTest,y));

% Compare with normal neural network (2 hidden layers)

net = patternnet([n1 50]);

net.trainParam.epochs = 200; tic net=train(net,xTrain,tTrain);

speed\_values\_pattern2(1,j) = toc;

y=net(xTest);

plotconfusion(tTest,y);

accuracy\_values\_pattern2(1,j) = 100\*(1-confusion(tTest,y));

end

speed\_results(1,n) = mean(speed\_values\_nofinetuning); speed\_results(2,n) = mean(speed\_values\_finetuned); speed\_results(3,n) = mean(speed\_values\_pattern1); speed\_results(4,n) = mean(speed\_values\_pattern2);

accuracy\_results(1,n) = mean(accuracy\_values\_nofinetuning); accuracy\_results(2,n) = mean(accuracy\_values\_finetuned); accuracy\_results(3,n) = mean(accuracy\_values\_pattern1); accuracy\_results(4,n) = mean(accuracy\_values\_pattern2);

end

%% Plot of the speed

plot(hidden\_neurons\_1, speed\_results(1,:) ,hidden\_neurons\_1, speed\_results(2,:),hidden\_neurons\_1,

speed\_results(3,:),hidden\_neurons\_1, speed\_results(4,:))

xlabel('Number of Hidden Neurons');

ylabel('Calculation Time');

legend('y = Stacked Autoencoders without fine tuning','y = Stacked Autoencoders with fine tuning',

'y = patternnet 1 layer', 'y = patternnet 2 layers', 'Location','southwest')

%% Plot of the accuracy

plot(hidden\_neurons\_1, accuracy\_results(1,:) ,hidden\_neurons\_1,

accuracy\_results(2,:),hidden\_neurons\_1, accuracy\_results(3,:),hidden\_neurons\_1, accuracy\_results(4,:))

xlabel('Number of Hidden Neurons');

ylabel('Accuracy');

legend('y = Stacked Autoencoders without fine tuning','y = Stacked Autoencoders with fine tuning',

'y = patternnet 1 layer', 'y = patternnet 2 layers', 'Location','southwest')

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

% Load the training data into memory

%[xTrainImages, tTrain] = digittrain\_dataset;

load('digittrain\_dataset');

hidden\_neurons\_1 = [20, 50, 100, 150, 200];

hidden\_neurons\_2 = [10, 20, 50, 100, 150];

%% Multiple loops to store the results speed\_results = zeros(4,5); accuracy\_results = zeros(4,5);

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

speed\_values\_nofinetuning = zeros(1,5); speed\_values\_finetuned = zeros(1,5); speed\_values\_pattern1 = zeros(1,5); speed\_values\_pattern2 = zeros(1,5);

accuracy\_values\_nofinetuning = zeros(1,5); accuracy\_values\_finetuned = zeros(1,5); accuracy\_values\_pattern1 = zeros(1,5); accuracy\_values\_pattern2 = zeros(1,5);

for j = 1:5

% certain amount of hidden neurons n2 = hidden\_neurons\_2(n);

% Neural networks have weights randomly initialized before training.

% Therefore the results from training are different each time. To avoid

% this behavior, explicitly set the random number generator seed. rng('default')

% Layer 1 hiddenSize1 = 100; tic

autoenc1 = trainAutoencoder(xTrainImages,hiddenSize1, ...

'MaxEpochs',200, ...

'L2WeightRegularization',0.004, ...

'SparsityRegularization',4, ...

'SparsityProportion',0.15, ...

'ScaleData', false);

toc\_1 = toc;

feat1 = encode(autoenc1,xTrainImages);

% Layer 2 hiddenSize2 = n2; tic

autoenc2 = trainAutoencoder(feat1,hiddenSize2, ...

'MaxEpochs',200, ...

'L2WeightRegularization',0.002, ...

'SparsityRegularization',4, ...

'SparsityProportion',0.1, ...

'ScaleData', false);

toc\_2 = toc;

feat2 = encode(autoenc2,feat1);

% Layer 3 tic

softnet = trainSoftmaxLayer(feat2,tTrain,'MaxEpochs',200);

toc\_3 = toc;

% Deep Net

deepnet = stack(autoenc1,autoenc2,softnet);

% Test deep net imageWidth = 28; imageHeight = 28;

inputSize = imageWidth\*imageHeight;

%[xTestImages, tTest] = digittest\_dataset;

load('digittest\_dataset');

xTest = zeros(inputSize,numel(xTestImages));

for i = 1:numel(xTestImages) xTest(:,i) = xTestImages{i}(:); end

y = deepnet(xTest);

%

accuracy\_values\_nofinetuning(1,j) = 100\*(1-confusion(tTest,y));

%

% Speed of the deep net without fine tuning speed\_values\_nofinetuning(1,j) = toc\_1 + toc\_2 + toc\_3;

% Test fine-tuned deep net

xTrain = zeros(inputSize,numel(xTrainImages));

for i = 1:numel(xTrainImages)

xTrain(:,i) = xTrainImages{i}(:);

end tic

deepnet = train(deepnet,xTrain,tTrain);

toc\_4 = toc;

y = deepnet(xTest);

%

accuracy\_values\_finetuned(1,j) = 100\*(1-confusion(tTest,y));

%

speed\_values\_finetuned(1,j) = toc\_1 + toc\_2 + toc\_3 + toc\_4;

%Compare with normal neural network (1 hidden layers)

net = patternnet(100); net.trainParam.epochs = 200; tic net=train(net,xTrain,tTrain);

speed\_values\_pattern1(1,j) = toc; y=net(xTest); plotconfusion(tTest,y);

accuracy\_values\_pattern1(1,j) = 100\*(1-confusion(tTest,y));

% Compare with normal neural network (2 hidden layers)

net = patternnet([100 n2]); net.trainParam.epochs = 200; tic net=train(net,xTrain,tTrain);

speed\_values\_pattern2(1,j) = toc; y=net(xTest); plotconfusion(tTest,y);

accuracy\_values\_pattern2(1,j) = 100\*(1-confusion(tTest,y));

end

speed\_results(1,n) = mean(speed\_values\_nofinetuning); speed\_results(2,n) = mean(speed\_values\_finetuned); speed\_results(3,n) = mean(speed\_values\_pattern1); speed\_results(4,n) = mean(speed\_values\_pattern2);

accuracy\_results(1,n) = mean(accuracy\_values\_nofinetuning); accuracy\_results(2,n) = mean(accuracy\_values\_finetuned); accuracy\_results(3,n) = mean(accuracy\_values\_pattern1); accuracy\_results(4,n) = mean(accuracy\_values\_pattern2);

end

%% Plot of the speed

plot(hidden\_neurons\_1, speed\_results(1,:) ,hidden\_neurons\_1, speed\_results(2,:),hidden\_neurons\_1, speed\_results(3,:),hidden\_neurons\_1, speed\_results(4,:))

xlabel('Number of Hidden Neurons');

ylabel('Calculation Time');

legend('y = Stacked Autoencoders without fine tuning','y = Stacked Autoencoders with fine tuning',

'y = patternnet 1 layer', 'y = patternnet 2 layers', 'Location','southwest')

%% Plot of the accuracy

plot(hidden\_neurons\_1, accuracy\_results(1,:) ,hidden\_neurons\_1, accuracy\_results(2,:),hidden\_neurons\_1, accuracy\_results(3,:),hidden\_neurons\_1, accuracy\_results(4,:))

xlabel('Number of Hidden Neurons');

ylabel('Accuracy');

legend('y = Stacked Autoencoders without fine tuning','y = Stacked Autoencoders with fine tuning',

'y = patternnet 1 layer', 'y = patternnet 2 layers', 'Location','southwest')

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

% Load the training data into memory

%[xTrainImages, tTrain] = digittrain\_dataset;

load('digittrain\_dataset');

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

%% Multiple loops to store the results speed\_results = zeros(4,5); accuracy\_results = zeros(4,5);

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

speed\_values\_nofinetuning = zeros(1,5); speed\_values\_finetuned = zeros(1,5); speed\_values\_pattern1 = zeros(1,5); speed\_values\_pattern2 = zeros(1,5);

accuracy\_values\_nofinetuning = zeros(1,5); accuracy\_values\_finetuned = zeros(1,5); accuracy\_values\_pattern1 = zeros(1,5); accuracy\_values\_pattern2 = zeros(1,5);

for j = 1:5

% certain amount epochs e = epochs(n);

% Neural networks have weights randomly initialized before training.

% Therefore the results from training are different each time. To avoid

% this behavior, explicitly set the random number generator seed.

rng('default')

% Layer 1 hiddenSize1 = 100; tic

autoenc1 = trainAutoencoder(xTrainImages,hiddenSize1, ...

'MaxEpochs',e, ...

'L2WeightRegularization',0.004, ...

'SparsityRegularization',4, ...

'SparsityProportion',0.15, ...

'ScaleData', false);

toc\_1 = toc;

feat1 = encode(autoenc1,xTrainImages);

% Layer 2 hiddenSize2 = 50; tic

autoenc2 = trainAutoencoder(feat1,hiddenSize2, ...

'MaxEpochs',e, ...

'L2WeightRegularization',0.002, ...

'SparsityRegularization',4, ...

'SparsityProportion',0.1, ...

'ScaleData', false);

toc\_2 = toc;

feat2 = encode(autoenc2,feat1);

% Layer 3 tic

softnet = trainSoftmaxLayer(feat2,tTrain,'MaxEpochs',e);

toc\_3 = toc;

% Deep Net

deepnet = stack(autoenc1,autoenc2,softnet);

% Test deep net imageWidth = 28; imageHeight = 28;

inputSize = imageWidth\*imageHeight;

%[xTestImages, tTest] = digittest\_dataset;

load('digittest\_dataset');

xTest = zeros(inputSize,numel(xTestImages));

for i = 1:numel(xTestImages)

xTest(:,i) = xTestImages{i}(:);

end

y = deepnet(xTest);

%

accuracy\_values\_nofinetuning(1,j) = 100\*(1-confusion(tTest,y));

%

% Speed of the deep net without fine tuning

speed\_values\_nofinetuning(1,j) = toc\_1 + toc\_2 + toc\_3;

% Test fine-tuned deep net

xTrain = zeros(inputSize,numel(xTrainImages));

for i = 1:numel(xTrainImages)

xTrain(:,i) = xTrainImages{i}(:);

end tic

deepnet = train(deepnet,xTrain,tTrain);

toc\_4 = toc;

y = deepnet(xTest);

%

accuracy\_values\_finetuned(1,j) = 100\*(1-confusion(tTest,y));

%

speed\_values\_finetuned(1,j) = toc\_1 + toc\_2 + toc\_3 + toc\_4;

%Compare with normal neural network (1 hidden layers)

net = patternnet(100);

net.trainParam.epochs = e; tic net=train(net,xTrain,tTrain);

speed\_values\_pattern1(1,j) = toc;

y=net(xTest);

plotconfusion(tTest,y);

accuracy\_values\_pattern1(1,j) = 100\*(1-confusion(tTest,y));

% Compare with normal neural network (2 hidden layers)

net = patternnet([100 50]);

net.trainParam.epochs = e; tic net=train(net,xTrain,tTrain);

speed\_values\_pattern2(1,j) = toc;

y=net(xTest);

plotconfusion(tTest,y);

accuracy\_values\_pattern2(1,j) = 100\*(1-confusion(tTest,y));

end

speed\_results(1,n) = mean(speed\_values\_nofinetuning); speed\_results(2,n) = mean(speed\_values\_finetuned); speed\_results(3,n) = mean(speed\_values\_pattern1); speed\_results(4,n) = mean(speed\_values\_pattern2);

accuracy\_results(1,n) = mean(accuracy\_values\_nofinetuning); accuracy\_results(2,n) = mean(accuracy\_values\_finetuned); accuracy\_results(3,n) = mean(accuracy\_values\_pattern1); accuracy\_results(4,n) = mean(accuracy\_values\_pattern2);

end

%% Plot of the speed

plot(epochs, speed\_results(1,:) ,epochs, speed\_results(2,:),epochs, speed\_results(3,:),epochs,

speed\_results(4,:)) xlabel('Epochs'); ylabel('Calculation Time');

legend('y = Stacked Autoencoders without fine tuning','y = Stacked Autoencoders with fine tuning',

'y = patternnet 1 layer', 'y = patternnet 2 layers', 'Location','northwest')

%% Plot of the accuracy

plot(epochs, accuracy\_results(1,:) ,epochs, accuracy\_results(2,:),epochs,

accuracy\_results(3,:),epochs, accuracy\_results(4,:))

xlabel('Epochs');

ylabel('Accuracy');

legend('y = Stacked Autoencoders without fine tuning','y = Stacked Autoencoders with fine tuning',

'y = patternnet 1 layer', 'y = patternnet 2 layers', 'Location','southwest')

**6.4.2.Answers to questions in section 2.2 of the exercise**

%% Image Category Classification Using Deep Learning

% This example shows how to use a pre-trained Convolutional Neural Network

% (CNN) as a feature extractor for training an image category classifier.

%

% Copyright 2016 The MathWorks, Inc.

%% Overview

% A Convolutional Neural Network (CNN) is a powerful machine learning

% technique from the field of deep learning. CNNs are trained using large

% collections of diverse images. From these large collections, CNNs can

% learn rich feature representations for a wide range of images. These

% feature representations often outperform hand-crafted features such as

% HOG, LBP, or SURF. An easy way to leverage the power of CNNs, without

% investing time and effort into training, is to use a pre-trained CNN as a

% feature extractor.

%

% In this example, images from Caltech 101 are classified into categories

% using a multiclass linear SVM trained with CNN features extracted from

% the images. This approach to image category classification follows the

% standard practice of training an off-the-shelf classifier using features

% extracted from images. For example, the

% <matlab:showdemo('ImageCategoryClassificationExample') Image Category

% Classification Using Bag Of Features> example uses SURF features within a

% bag of features framework to train a multiclass SVM. The difference here

% is that instead of using image features such as HOG or SURF, features are

% extracted using a CNN. And, as this example will show, the classifier

% trained using CNN features provides close to 100% accuracy, which

% is higher than the accuracy achieved using bag of features and SURF.

%

% Note: This example requires Neural Network Toolbox(TM), Parallel

% Computing Toolbox(TM), Statistics and Machine Learning Toolbox(TM), and a

% CUDA-capable GPU card.

%% Download Image Data

% The category classifier will be trained on images from

% <http://www.vision.caltech.edu/Image\_Datasets/Caltech101 Caltech 101>.

% Caltech 101 is one of the most widely cited and used image data sets,

% collected by Fei-Fei Li, Marco Andreetto, and Marc 'Aurelio Ranzato.

% Download the compressed data set from the following location

url = 'http://www.vision.caltech.edu/Image\_Datasets/Caltech101/101\_ObjectCategories.tar.gz';

% Store the output in a temporary folder

outputFolder = fullfile(tempdir, 'caltech101'); % define output folder

%%

% Note: Download time of the data depends on your internet connection. The

% next set of commands use MATLAB to download the data and will block

% MATLAB. Alternatively, you can use your web browser to first download the

% dataset to your local disk. To use the file you downloaded from the web,

% change the 'outputFolder' variable above to the location of the

% downloaded file.

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

end

disp('Downloading 126MB Caltech101 data set...');

untar(url, outputFolder);

%% Load Images

% Instead of operating on all of Caltech 101, which is time consuming, use

% three of the categories: airplanes, ferry, and laptop. The image category

% classifier will be trained to distinguish amongst these six categories.

rootFolder = fullfile(outputFolder, '101\_ObjectCategories');

categories = {'airplanes', 'ferry', 'laptop'};

%%

% Create an |ImageDatastore| to help you manage the data. Because

% |ImageDatastore| operates on image file locations, images are not loaded

% into memory until read, making it efficient for use with large image

% collections.

imds = imageDatastore(fullfile(rootFolder, categories), 'LabelSource', 'foldernames');

%%

% The |imds| variable now contains the images and the category labels

% associated with each image. The labels are automatically assigned from

% the folder names of the image files. Use |countEachLabel| to summarize

% the number of images per category. tbl = countEachLabel(imds)

%%

% Because |imds| above contains an unequal number of images per category,

% let's first adjust it, so that the number of images in the training set

% is balanced.

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

% Use splitEachLabel method to trim the set.

imds = splitEachLabel(imds, minSetCount, 'randomize');

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

%%

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

% in the dataset.

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

ferry = find(imds.Labels == 'ferry', 1);

laptop = find(imds.Labels == 'laptop', 1);

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

%% Download Pre-trained Convolutional Neural Network (CNN)

% Now that the images are prepared, you will need to download a pre-trained

% CNN model for this example. There are several pre-trained networks that

% have gained popularity. Most of these have been trained on the ImageNet

% dataset, which has 1000 object categories and 1.2 million training

% images[1]. "AlexNet" is one such model and can be downloaded from

% MatConvNet[2,3]:

% Location of pre-trained "AlexNet"

cnnURL = 'http://www.vlfeat.org/matconvnet/models/beta16/imagenet-caffe-alex.mat';

% Store CNN model in a temporary folder

cnnMatFile = fullfile(tempdir, 'imagenet-caffe-alex.mat');

%%

% Note: Download time of the data depends on your internet connection. The

% next set of commands use MATLAB to download the data and will block

% MATLAB. Alternatively, you can use your web browser to first download the

% dataset to your local disk. To use the file you downloaded from the web,

% change the 'cnnMatFile' variable above to the location of the downloaded

% file.

if ~exist(cnnMatFile, 'file') % download only once disp('Downloading pre-trained CNN model...'); websave(cnnMatFile, cnnURL);

end

%% Load Pre-trained CNN

% The CNN model is saved in MatConvNet's format [3]. Load the MatConvNet

% network data into |convnet|, a |SeriesNetwork| object from Neural Network

% Toolbox(TM), using the helper function |helperImportMatConvNet|. A

% SeriesNetwork object can be used to inspect the network architecture,

% classify new data, and extract network activations from specific layers.

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

%%

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

% View the CNN architecture convnet.Layers

%%

% The first layer defines the input dimensions. Each CNN has a different

% input size requirements. The one used in this example requires image

% input that is 227-by-227-by-3.

% Inspect the first layer convnet.Layers(1)

%%

% The intermediate layers make up the bulk of the CNN. These are a series

% of convolutional layers, interspersed with rectified linear units (ReLU)

% and max-pooling layers [2]. Following the these layers are 3

% fully-connected layers.

%

% The final layer is the classification layer and its properties depend on

% the classification task. In this example, the CNN model that was loaded

% was trained to solve a 1000-way classification problem. Thus the

% classification layer has 1000 classes from the ImageNet dataset.

% Inspect the last layer convnet.Layers(end)

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

%%

% Note that the CNN model is not going to be used for the original

% classification task. It is going to be re-purposed to solve a different

% classification task on the Caltech 101 dataset.

%% Pre-process Images For CNN

% As mentioned above, |convnet| can only process RGB images that are

% 227-by-227. To avoid re-saving all the images in Caltech 101 to this

% format, setup the |imds| read function, |imds.ReadFcn|, to pre-process

% images on-the-fly. The |imds.ReadFcn| is called every time an image is

% read from the |ImageDatastore|.

% Set the ImageDatastore ReadFcn

imds.ReadFcn = @(filename)readAndPreprocessImage(filename);

%% Prepare Training and Test Image Sets

% Split the sets into training and validation data. Pick 30% of images

% from each set for the training data and the remainder, 70%, for the

% validation data. Randomize the split to avoid biasing the results. The

% training and test sets will be processed by the CNN model.

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

%% Extract Training Features Using CNN

% Each layer of a CNN produces a response, or activation, to an input

% image. However, there are only a few layers within a CNN that are

% suitable for image feature extraction. The layers at the beginning of the

% network capture basic image features, such as edges and blobs. To see

% this, visualize the network filter weights from the first convolutional

% layer. This can help build up an intuition as to why the features

% extracted from CNNs work so well for image recognition tasks. Note that

% visualizing deeper layer weights is beyond the scope of this example. You

% can read more about that in the work of Zeiler and Fergus [4].

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

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

w1 = imresize(w1,5);

% Display a montage of network weights. There are 96 individual sets of

% weights in the first layer. figure

montage(w1)

title('First convolutional layer weights')

% %%

% % Notice how the first layer of the network has learned filters for

% % capturing blob and edge features. These "primitive" features are then

% % processed by deeper network layers, which combine the early features to

% % form higher level image features. These higher level features are better

% % suited for recognition tasks because they combine all the primitive

% % features into a richer image representation [5].

% %

% % You can easily extract features from one of the deeper layers using the

% % |activations| method. Selecting which of the deep layers to choose is a

% % design choice, but typically starting with the layer right before the

% % classification layer is a good place to start. In |convnet|, the this

% % layer is named 'fc7'. Let's extract training features using that layer.

% featureLayer = 'fc7';

% trainingFeatures = activations(convnet, trainingSet, featureLayer, ...

% 'MiniBatchSize', 32, 'OutputAs', 'columns');

% %%

% % Note that the activations are computed on the GPU and the 'MiniBatchSize'

% % is set 32 to ensure that the CNN and image data fit into GPU memory.

% % You may need to lower the 'MiniBatchSize' if your GPU runs out of memory.

% %

% % Also, the activations output is arranged as columns. This helps speed-up

% % the multiclass linear SVM training that follows.

%

% %% Train A Multiclass SVM Classifier Using CNN Features

% % Next, use the CNN image features to train a multiclass SVM classifier. A

% % fast Stochastic Gradient Descent solver is used for training by setting

% % the |fitcecoc| function's 'Learners' parameter to 'Linear'. This helps

% % speed-up the training when working with high-dimensional CNN feature

% % vectors, which each have a length of 4096.

%

% % Get training labels from the trainingSet

% trainingLabels = trainingSet.Labels;

%

% % Train multiclass SVM classifier using a fast linear solver, and set

% % 'ObservationsIn' to 'columns' to match the arrangement used for training

% % features.

% classifier = fitcecoc(trainingFeatures, trainingLabels, ...

% 'Learners', 'Linear', 'Coding', 'onevsall', 'ObservationsIn', 'columns');

%

% %% Evaluate Classifier

% % Repeat the procedure used earlier to extract image features from

% % |testSet|. The test features can then be passed to the classifier to

% % measure the accuracy of the trained classifier.

%

% % Extract test features using the CNN

% testFeatures = activations(convnet, testSet, featureLayer, 'MiniBatchSize',32);

%

% % Pass CNN image features to trained classifier

% predictedLabels = predict(classifier, testFeatures);

%

% % Get the known labels

% testLabels = testSet.Labels;

%

% % Tabulate the results using a confusion matrix.

% confMat = confusionmat(testLabels, predictedLabels);

%

% % Convert confusion matrix into percentage form

% confMat = [bsxfun(@rdivide,confMat,sum(confMat,2))](mailto:@rdivide)

% %%

%

% % Display the mean accuracy

% mean(diag(confMat))

%

% %% Try the Newly Trained Classifier on Test Images

% % You can now apply the newly trained classifier to categorize new images.

% newImage = fullfile(rootFolder, 'airplanes', 'image\_0690.jpg');

%

% % Pre-process the images as required for the CNN

% img = readAndPreprocessImage(newImage);

%

% % Extract image features using the CNN

% imageFeatures = activations(convnet, img, featureLayer);

% %%

%

% % Make a prediction using the classifier

% label = predict(classifier, imageFeatures)

%

%

% %% References

% % [1] Deng, Jia, et al. "Imagenet: A large-scale hierarchical image

% % database." Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE

% % Conference on. IEEE, 2009.

% %

% % [2] Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet

% % classification with deep convolutional neural networks." Advances in

% % neural information processing systems. 2012.

% %

% % [3] Vedaldi, Andrea, and Karel Lenc. "MatConvNet-convolutional neural

% % networks for MATLAB." arXiv preprint arXiv:1412.4564 (2014).

% %

% % [4] Zeiler, Matthew D., and Rob Fergus. "Visualizing and understanding

% % convolutional networks." Computer Vision-ECCV 2014. Springer

% % International Publishing, 2014. 818-833.

% %

% % [5] Donahue, Jeff, et al. "Decaf: A deep convolutional activation feature

% % for generic visual recognition." arXiv preprint arXiv:1310.1531 (2013).

%

% displayEndOfDemoMessage(mfilename)

**6.5. Final Project**

**6.5.1. Problem 1: nonlinear regression with MLP’s**

%% loading the data load('Data\_Problem1\_regression.mat')

% the new target variable

Y = (8\*T1 + 5\*T2 + 4\*T3 + 3\*T3 + 3\*T5) / (8+5+4+3+3);

%% creating the training, validation and test sets rng('default');

index = randperm(size(X1, 1),3000);

% deriving 3000 points, which can be divided between the training,

% validation and test set

X = [X1 X2]'; Y = Y';

% creating the right 3000 X and Y values

X = X(:,index); Y = Y(index);

%%

% number of epochs -> 1000 epochs = 1000;

% transfer function for the hidden layer -> ("logsig" , "tansig")

transfer\_functions = {'logsig', 'tansig'};

% number of neurons

number\_neurons = [5, 10, 20, 50, 100];

% different training algorithms

training\_algs = { 'traingd', 'traingda', 'traincgf', 'traincgp', 'trainbfg', 'trainlm', 'trainbr'};

% number of iterarions for each execution n\_iterations = 10;

%% creating the barplots figure

subplot\_index = 0;

for t = transfer\_functions

subplot\_index = subplot\_index + 1;

transfer\_function = char(t);

% matrix to store speed values in -> speed\_values

speed\_values = zeros(length(training\_algs), length(number\_neurons));

RMSE\_val\_average\_error = zeros(length(training\_algs), length(number\_neurons));

% index for speed\_values matrix i = 0;

j= 0;

for train\_alg = training\_algs

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

for n\_number = number\_neurons

% training algorithm train\_alg = char(train\_alg);

% array to store the calculation times over the 20 iterations calc\_time\_values = zeros(1,n\_iterations);

% array to store the RMSE test errors over the 20 iterations

RMSE\_val\_values = zeros(1,n\_iterations);

for it = 1:n\_iterations

% the forwardfeed neural network with 1/3 training, 1/3

% validation, 1/3 testset

net = feedforwardnet(n\_number, train\_alg);

net.trainParam.epochs = epochs; net.divideParam.trainRatio = 1/3; net.divideParam.valRatio = 1/3; net.divideParam.testRatio = 1/3;

% the transfer function

net.layers{1}.transferFcn = char(transfer\_function);

% !! THE OUTPUT LAYER WILL ALWAYS HAVE THE LINEAR TRANSFER

% FUNCTION AS IS NECASSARY FOR REGRESSION

% training the neural network and measuring average calculation time tic;

[net, tr] = train(net, X, Y);

calc\_time = toc;

X\_train = X(:, tr.trainInd); Y\_train = Y(:, tr.trainInd);

X\_val = X(:, tr.testInd); Y\_val = Y(:, tr.testInd);

% the calculation time values over 10 iterations calc\_time\_values(it) = calc\_time;

% RMSE validation set

RMSE\_val\_values(it) = sqrt(sum((Y\_val - sim(net, X\_val)).^2));

end

% average calculation time

average\_calc\_time = mean(calc\_time\_values);

% average RMSE val error

average\_RMSE\_val = mean(RMSE\_val\_values);

% updating the column index that corresponds to certain amount of

% hidden neurons j = j + 1;

% adding the calculation time values to the matrix speed\_values(i,j) = average\_calc\_time;

% adding the RMSE test error to the matrix

RMSE\_val\_average\_error(i,j) = average\_RMSE\_val;

end

j = 0;

end

% Bar Plot Speed

% returning a barplot for the Speed

subplot(1,4,subplot\_index)

names = categorical(training\_algs);

barplot = bar(names, speed\_values);

ylabel("Calculation Time")

legend\_bar = legend(barplot,{"5", "10", "20", "50", "100"}, 'Location', 'northwest');

title(legend\_bar, "Hidden Neurons");

subplot\_index = subplot\_index + 1;

subplot(1,4,subplot\_index)

% Bar Plot Validation RMSE

% returning a barplot for the RMSE Test Error

names = categorical(training\_algs);

barplot = bar(names, RMSE\_val\_average\_error);

ylabel("RMSE Validation Error")

legend\_bar = legend(barplot,{"5", "10", "20", "50", "100"}, 'Location', 'northwest');

title(legend\_bar, "Hidden Neurons");

end

%% loading the data load('Data\_Problem1\_regression.mat')

% the new target variable

Y = (8\*T1 + 5\*T2 + 4\*T3 + 3\*T3 + 3\*T5) / (8+5+4+3+3);

%% Data set

% test and training set together should contain 2000 points rng('default');

index = randperm(size(X1, 1),3000); X = [X1 X2]';

Y = Y';

X = X(:,index); Y = Y(index);

%% Feedforward neural network with trainbr as training algorithm and 50 hidden neurons rng('default');

% number of epochs n\_epochs = 1000;

% the forwardfeed neural network with 1/3 training, 1/3 validation, 1/3 testset net = feedforwardnet(50, 'trainbr');

net.trainParam.epochs = n\_epochs; net.divideParam.trainRatio = 1/3; net.divideParam.valRatio = 1/3; net.divideParam.testRatio = 1/3;

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

X\_train = X(:, tr.trainInd); Y\_train = Y(:, tr.trainInd);

X\_test = X(:, tr.testInd); Y\_test = Y(:, tr.testInd);

% MSE training error

MSE\_training\_error = mean((Y\_train - sim(net, X\_train)).^2)

% MSE test error

MSE\_test\_error = mean((Y\_test - sim(net, X\_test)).^2)

% corr = corrcoef(T\_test, T\_test\_sim);

corr = corrcoef(Y\_test, sim(net, X\_test));

% R squared

Rsquared = (corr(1,2))^2

%% scatter plot for the real and predicted Y values of the test set

% first putting predicted y values in variables Y\_test\_predicted

Y\_test\_predicted = sim(net, X\_test);

X\_1\_test = X\_test(1,:); X\_2\_test = X\_test(2,:);

%% 3D scatter predicted figure(1)

scatter3(X\_1\_test, X\_2\_test, Y\_test, 'filled', 'MarkerEdgeColor','k')

xlabel('X1'), ylabel('X2'), zlabel('F(X1, X2)')

%% 3D scatter target, real values

figure(2)

scatter3(X\_1\_test, X\_2\_test, Y\_test\_predicted,'filled', 'r', 'MarkerEdgeColor','k')

xlabel('X1'), ylabel('X2'), zlabel('Predicted Values')

**6.5.2. Problem 1: classification with MLP’s**

%% loading the data

data = readtable('winequality-white.csv');

wines\_1 = data(data.quality == 5,:); wines\_2 = data(data.quality == 6,:); wines\_3 = data(data.quality == 7,:);

% creating the right frame in order to be able to perform the patternnet wines = [wines\_1; wines\_2; wines\_3];

wines.class1 = wines.quality == 5;

wines.class2 = (wines.quality == 6 | wines.quality == 7);

wines = table2array(wines); X = wines(:, 1:end-3)';

Y = wines(:, end-1:end)';

%% PCA reduction of the input variables

% rows are the observations, columns the variables

X = X';

cov\_matrix = cov(X);

% amount of dimensions

% determining sum of all eigenvalues n\_dimensions = size(cov\_matrix,1);

[all\_v, all\_d] = eigs(cov\_matrix,n\_dimensions);

total\_eigen\_values = sum(diag(all\_d));

%% Eigenvalue decomposition x\_plot = 1:11;

proportion\_values = zeros(1,11);

error\_values = zeros(1,11);

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

% returning the eigenvectors of k largest eigenvalues

[v,d] = eigs(cov\_matrix,k);

% PROPORTION EXPLAINED

% proportion explained by k principal components -> eigenvalues

proportion\_eigen\_values = sum(diag(d));

% proportion of variance explained

proportion\_value = proportion\_eigen\_values / total\_eigen\_values;

% RMSE ERROR OF RECONSTRUCTION MATRIX

transformation = v;

% transforming the original data with k principal components as dimensions x\_transformed = X \* transformation;

% try to reconstruct the original data\_matrix x\_estimated = x\_transformed \* (transformation'); RMSE\_error = sqrt(mean(mean((X - x\_estimated).^2)));

% adding values to arrays proportion\_values(k) = proportion\_value; error\_values(k) = RMSE\_error;

end

%% Plot proportion variance explained graph\_plot = plot(x\_plot, proportion\_values); graph\_plot.LineWidth = 2;

xlabel("Number Principal Components");

ylabel("Proportion Variance Explained");

%% Reduced dataset with three principal components

% 3 is a good number of principal components

% returning the eigenvectors of 3 largest eigenvalues

[v,d] = eigs(cov\_matrix,3);

% the reduced data set with the three principal components transformation = v;

% reduced data set with three principal components

% transforming the original data with k principal components as dimensions

X\_transformed = X \* transformation;

X\_transformed = X\_transformed';

%% Network architecture for three principal components

% number of epochs -> 1000

epochs = 1000;

% transfer function for the hidden layer -> ("logsig" , "tansig")

transfer\_functions = {'logsig', 'tansig'};

% number of neurons

number\_neurons = [5, 10, 20, 50, 100];

% different training algorithms

training\_algs = { 'traingd', 'traingda', 'traincgf', 'traincgp', 'trainbfg', 'trainlm', 'trainbr'};

% number of iterarions for each execution n\_iterations = 1;

% creating the barplots subplot\_index = 0;

%%

figure;

for t = transfer\_functions

subplot\_index = subplot\_index + 1;

transfer\_function = char(t);

% matrix to store speed values in -> speed\_values

speed\_values = zeros(length(training\_algs), length(number\_neurons)); CCR\_val\_average\_error = zeros(length(training\_algs), length(number\_neurons));

% index for speed\_values matrix i = 0;

j= 0;

for train\_alg = training\_algs

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

for n\_number = number\_neurons

% training algorithm train\_alg = char(train\_alg);

% array to store the calculation times over the 10 iterations calc\_time\_values = zeros(1,n\_iterations);

% array to store the RMSE test errors over the 10 iterations

CCR\_val\_values = zeros(1,n\_iterations);

for it = 1:n\_iterations

net = patternnet(n\_number, train\_alg);

net.trainParam.epochs = epochs;

% the transfer function net.layers{1}.transferFcn = char(t);

% output layer will be equal to softmax transfer

% function

% training the neural network and measuring average calculation time tic;

[net, tr] = train(net, X\_transformed, Y);

calc\_time = toc;

X\_transformed\_val = X\_transformed(:, tr.testInd); Y\_val = Y(:, tr.testInd);

% the calculation time values over 10 iterations calc\_time\_values(it) = calc\_time;

% correct classification ratio

Y\_val\_sim = sim(net, X\_transformed\_val); [conf,~,~,~] = confusion(Y\_val, Y\_val\_sim); CCR\_val\_values(it) = 100\*(1-conf);

end

% average calculation time

average\_calc\_time = mean(calc\_time\_values);

% average RMSE val error

average\_CCR\_val = mean(CCR\_val\_values);

% updating the column index that corresponds to certain amount of

% hidden neurons j = j + 1;

% adding the calculation time values to the matrix speed\_values(i,j) = average\_calc\_time;

% adding the Validation CCR to the matrix

CCR\_val\_average\_error(i,j) = average\_CCR\_val;

end

j = 0;

end

% Bar Plot Speed

% returning a barplot for the Speed

subplot(1,4,subplot\_index)

names = categorical(training\_algs);

barplot = bar(names, speed\_values);

ylabel("Calculation Time")

legend\_bar = legend(barplot,{"5", "10", "20", "50", "100"}, 'Location', 'northeast');

title(legend\_bar, "Hidden Neurons");

subplot\_index = subplot\_index + 1;

subplot(1,4,subplot\_index)

% Bar Plot Validation CCR

% returning a barplot for the CCR

names = categorical(training\_algs);

barplot = bar(names, CCR\_val\_average\_error);

ylabel("CCR Validation Error")

legend\_bar = legend(barplot,{"5", "10", "20", "50", "100"}, 'Location', 'southwest');

title(legend\_bar, "Hidden Neurons");

end

%% loading the data

data = readtable('winequality-white.csv'); wines\_1 = data(data.quality == 5,:); wines\_2 = data(data.quality == 6,:); wines\_3 = data(data.quality == 7,:);

% creating the right frame in order to be able to perform the patternnet wines = [wines\_1; wines\_2; wines\_3];

wines.class1 = wines.quality == 5;

wines.class2 = (wines.quality == 6 | wines.quality == 7);

wines = table2array(wines);

X = wines(:, 1:end-3)';

Y = wines(:, end-1:end)';

%%

% number of epochs -> 1000 epochs = 1000;

% transfer function for the hidden layer -> ("logsig" , "tansig")

transfer\_functions = {'logsig', 'tansig'};

% number of neurons

number\_neurons = [5, 10, 20, 50, 100];

% different training algorithms

training\_algs = { 'traingd', 'traingda', 'traincgf', 'traincgp', 'trainbfg', 'trainlm', 'trainbr'};

% number of iterarions for each execution n\_iterations = 10;

%% creating the barplots figure

subplot\_index = 0;

for t = transfer\_functions

subplot\_index = subplot\_index + 1;

transfer\_function = char(t);

% matrix to store speed values in -> speed\_values

speed\_values = zeros(length(training\_algs), length(number\_neurons)); CCR\_val\_average\_error = zeros(length(training\_algs), length(number\_neurons));

% index for speed\_values matrix i = 0;

j= 0;

for train\_alg = training\_algs

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

for n\_number = number\_neurons

% training algorithm train\_alg = char(train\_alg);

% array to store the calculation times over the 10 iterations calc\_time\_values = zeros(1,n\_iterations);

% array to store the RMSE test errors over the 10 iterations

CCR\_val\_values = zeros(1,n\_iterations);

for it = 1:n\_iterations

net = patternnet(n\_number, train\_alg);

net.trainParam.epochs = epochs;

% the transfer function

net.layers{1}.transferFcn = char(t);

% output layer will be equal to softmax transfer

% function

% training the neural network and measuring average calculation time tic;

[net, tr] = train(net, X, Y);

calc\_time = toc;

X\_val = X(:, tr.testInd); Y\_val = Y(:, tr.testInd);

% the calculation time values over 10 iterations calc\_time\_values(it) = calc\_time;

% correct classification ratio

Y\_val\_sim = sim(net, X\_val);

[conf,~,~,~] = confusion(Y\_val, Y\_val\_sim); CCR\_val\_values(it) = 100\*(1-conf);

end

% average calculation time

average\_calc\_time = mean(calc\_time\_values);

% average RMSE val error

average\_CCR\_val = mean(CCR\_val\_values);

% updating the column index that corresponds to certain amount of

% hidden neurons

j = j + 1;

% adding the calculation time values to the matrix speed\_values(i,j) = average\_calc\_time;

% adding the Validation CCR to the matrix

CCR\_val\_average\_error(i,j) = average\_CCR\_val;

end

j = 0;

end

% Bar Plot Speed

% returning a barplot for the Speed subplot(1,4,subplot\_index)

names = categorical(training\_algs); barplot = bar(names, speed\_values); ylabel("Calculation Time")

legend\_bar = legend(barplot,{"5", "10", "20", "50", "100"}, 'Location', 'northeast');

title(legend\_bar, "Hidden Neurons");

subplot\_index = subplot\_index + 1;

subplot(1,4,subplot\_index)

% Bar Plot Validation CCR

% returning a barplot for the CCR

names = categorical(training\_algs);

barplot = bar(names, CCR\_val\_average\_error);

ylabel("CCR Validation Error")

legend\_bar = legend(barplot,{"5", "10", "20", "50", "100"}, 'Location', 'southwest');

title(legend\_bar, "Hidden Neurons");

end

**6.5.3.Problem 2: character recognition with Hopfield networks**

% creating dataset of letters capital\_letters = prprob();

% creating the lowercase letters of 'simondelac' letter\_s = [0 0 0 0 0 ...

0 0 0 0 0 ...

0 1 1 1 0 ...

1 0 0 0 0 ...

0 1 1 1 0 ...

0 0 0 0 1 ...

0 1 1 1 0 ]';

letter\_i = [0 0 0 0 0 ...

0 0 0 0 0 ...

0 0 1 0 0 ...

0 0 0 0 0 ...

0 0 1 0 0 ...

0 0 1 0 0 ...

0 0 1 0 0 ]';

letter\_m = [0 0 0 0 0 ...

0 0 0 0 0 ...

1 1 0 1 1 ...

1 0 1 0 1 ...

1 0 1 0 1 ...

1 0 1 0 1 ...

1 0 1 0 1 ]';

letter\_o = [0 0 0 0 0 ...

0 0 0 0 0 ...

0 1 1 1 0 ...

1 0 0 0 1 ...

1 0 0 0 1 ...

1 0 0 0 1 ...

0 1 1 1 0 ]';

letter\_n = [0 0 0 0 0 ...

0 0 0 0 0 ...

1 1 1 0 0 ...

1 0 0 1 0 ...

1 0 0 1 0 ...

1 0 0 1 0 ...

1 0 0 1 0 ]';

letter\_d = [0 0 0 0 0 ...

0 0 0 0 0 ...

0 0 0 1 0 ...

0 1 1 1 0 ...

1 0 0 1 0 ...

1 0 0 1 0 ...

0 1 1 1 0 ]';

letter\_e = [0 0 0 0 0 ...

0 0 0 0 0 ...

0 1 1 0 0 ...

1 0 0 1 0 ...

1 1 1 1 0 ...

1 0 0 0 0 ...

0 1 1 1 0 ]';

letter\_l = [0 0 0 0 0 ...

0 0 0 0 0 ...

0 1 0 0 0 ...

0 1 0 0 0 ...

0 1 0 0 0 ...

0 1 0 0 0 ...

0 1 1 0 0 ]';

letter\_a = [0 0 0 0 0 ...

0 0 0 0 0 ...

0 1 1 0 0 ...

1 0 0 1 0 ...

1 0 0 1 0 ...

1 0 0 1 0 ...

0 1 1 0 1 ]';

letter\_c = [0 0 0 0 0 ...

0 0 0 0 0 ...

0 1 1 1 0 ...

1 0 0 0 0 ...

1 0 0 0 0 ...

1 0 0 0 0 ...

0 1 1 1 0 ]';

% creating the dataset with all letters

letters = [letter\_s, letter\_i, letter\_m, letter\_o, letter\_n, letter\_d, letter\_e, letter\_l, letter\_a, letter\_c, capital\_letters];

%% image of the ten first letters subplot\_index = 0;

for i = 1:10

letter = letters(:,i);

colormap('gray');

subplot\_index = subplot\_index + 1;

end

subplot(1,10,subplot\_index); imagesc(reshape(letter',5,7)') axis off

%% Setting the zeros equal to -1, in order for the Hopfield network to work letters(letters == 0) = -1;

%% Hopfield network to retrieve the first five characters

% the data of first five letters

first\_5 = letters(:, 1:5);

% creating the Hopfield network net = newhop(first\_5);

%% randomly distorting 3 pixels of the first five characters as input first\_5\_distorted = first\_5;

% array to store the amount of correct classifications right\_classifications = zeros(1,5); wrong\_classifications = zeros(1,5);

time\_steps = 100;

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

% determine index of the three random components that we will change first\_index = round(randi([1 35],1));

second\_index = round(randi([1 35],1));

third\_index = round(randi([1 35],1));

indices = [first\_index, second\_index, third\_index];

for in = indices

% inversing the pixel value if(first\_5\_distorted(in,i) == 1)

first\_5\_distorted(in,i) = -1;

else

first\_5\_distorted(in,i) = 1;

end

end

end

for k = 1:5

letter\_input = first\_5\_distorted(:,k);

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

% letter

T = {letter\_input};

[letter\_output,~,~] = sim(net,{1 ,time\_steps},{},T);

letter\_output = letter\_output{1, time\_steps};

% the letters to check letters\_to\_check = first\_5;

if(isequal(letter\_output, letters\_to\_check(:,k)) == true)

right\_classifications(k) = right\_classifications(k) + 1;

end

% plotting if spurious state m = 0;

for j = 1:5

if(isequal(letter\_output, letters\_to\_check(:,j)) == true)

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

end

end

if(m == 0)

% plotting the spurious state

% hold on

% figure;

% colormap('gray');

% printing the spurious state

% imagesc(reshape(letter\_output',5,7)')

% axis off

% title(k);

end

end

end

%% Barplot of the classifications

names = categorical({'s', 'i', 'm', 'o', 'n'});

names = reordercats(names,{'s', 'i', 'm', 'o', 'n'});

frequency\_classifications = right\_classifications / 100;

bar(names, frequency\_classifications);

title("Correct Classifications"); ylabel("Number of Correct Classifications") xlabel('Distorted letters')

% creating dataset of letters capital\_letters = prprob();

% creating the lowercase letters of 'simondelac' letter\_s = [0 0 0 0 0 ...

0 0 0 0 0 ...

0 1 1 1 0 ...

1 0 0 0 0 ...

0 1 1 1 0 ...

0 0 0 0 1 ...

0 1 1 1 0 ]';

letter\_i = [0 0 0 0 0 ...

0 0 0 0 0 ...

0 0 1 0 0 ...

0 0 0 0 0 ...

0 0 1 0 0 ...

0 0 1 0 0 ...

0 0 1 0 0 ]';

letter\_m = [0 0 0 0 0 ...

0 0 0 0 0 ...

1 1 0 1 1 ...

1 0 1 0 1 ...

1 0 1 0 1 ...

1 0 1 0 1 ...

1 0 1 0 1 ]';

letter\_o = [0 0 0 0 0 ...

0 0 0 0 0 ...

0 1 1 1 0 ...

1 0 0 0 1 ...

1 0 0 0 1 ...

1 0 0 0 1 ...

0 1 1 1 0 ]';

letter\_n = [0 0 0 0 0 ...

0 0 0 0 0 ...

1 1 1 0 0 ...

1 0 0 1 0 ...

1 0 0 1 0 ...

1 0 0 1 0 ...

1 0 0 1 0 ]';

letter\_d = [0 0 0 0 0 ...

0 0 0 0 0 ...

0 0 0 1 0 ...

0 1 1 1 0 ...

1 0 0 1 0 ...

1 0 0 1 0 ...

0 1 1 1 0 ]';

letter\_e = [0 0 0 0 0 ...

0 0 0 0 0 ...

0 1 1 0 0 ...

1 0 0 1 0 ...

1 1 1 1 0 ...

1 0 0 0 0 ...

0 1 1 1 0 ]';

letter\_l = [0 0 0 0 0 ...

0 0 0 0 0 ...

0 1 0 0 0 ...

0 1 0 0 0 ...

0 1 0 0 0 ...

0 1 0 0 0 ...

0 1 1 0 0 ]';

letter\_a = [0 0 0 0 0 ...

0 0 0 0 0 ...

0 1 1 0 0 ...

1 0 0 1 0 ...

1 0 0 1 0 ...

1 0 0 1 0 ...

0 1 1 0 1 ]';

letter\_c = [0 0 0 0 0 ...

0 0 0 0 0 ...

0 1 1 1 0 ...

1 0 0 0 0 ...

1 0 0 0 0 ...

1 0 0 0 0 ...

0 1 1 1 0 ]';

% creating the dataset with all letters

letters = [letter\_s, letter\_i, letter\_m, letter\_o, letter\_n, letter\_d, letter\_e, letter\_l, letter\_a, letter\_c, capital\_letters];

%% image of the ten first letters subplot\_index = 0;

for i = 1:10

letter = letters(:,i);

colormap('gray');

subplot\_index = subplot\_index + 1;

end

subplot(1,10,subplot\_index); imagesc(reshape(letter',5,7)') axis off

%% Setting the zeros equal to -1, in order for the Hopfield network to work letters(letters == 0) = -1;

%% Error and storing different numbers of P P = 36;

final\_error\_values = zeros(1,P); final\_correct\_classifications = zeros(1,P); time\_steps = 2000;

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

first\_letters = letters(:, 1:P);

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

% storing the distorted letters first\_letters\_distorted = first\_letters;

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

for iter = 1:20 for i = 1:h

% determine index of the three random components that we will change first\_index = round(randi([1 35],1));

second\_index = round(randi([1 35],1));

third\_index = round(randi([1 35],1));

indices = [first\_index, second\_index, third\_index];

for in = indices

% inversing the pixel value if(first\_letters\_distorted(in,i) == 1)

first\_letters\_distorted(in,i) = -1;

else

first\_letters\_distorted(in,i) = 1;

end

end

end

error\_P\_values = zeros(1,P);

for k = 1:h

letter\_input = first\_letters\_distorted(:,k);

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

% letter

T = {letter\_input};

[letter\_output,~,~] = sim(net,{1 ,time\_steps},{},T);

letter\_output = letter\_output{1, time\_steps};

% rounding the values certainly to a 1 or -1 for output\_rounding = 1:35

if(letter\_output(output\_rounding) > 0)

letter\_output(output\_rounding) = 1;

else

letter\_output(output\_rounding) = -1;

end

end

% the letters to check letters\_to\_check = first\_letters;

% calculating error

if(isequal(letter\_output, letters\_to\_check(:,k)) == true)

error = 0;

else

error = sum(abs(letters\_to\_check(:,1) - letter\_output))/2;

end

error\_P\_values(1,k) = error;

end

storing\_error\_values = storing\_error\_values + error\_P\_values;

end

final\_error = median(storing\_error\_values/20);

final\_error\_values(1,h) = final\_error;

end

%% plotting error in function of P

p = plot(1:P, final\_error\_values,'-o');

p.LineWidth = 1.2;

xlabel('Number of characters P stored');

ylabel('"median of averages" error')

hold on; % hold the plot for other curves plot(17,final\_error\_values(17),'o','MarkerSize',15); txt = ('critical loading capacity');

t = text(17, final\_error\_values(17)+0.7, txt);

t.FontWeight = 'bold';

% creating dataset of letters capital\_letters = prprob();

% creating the lowercase letters of 'simondelac' letter\_s = [0 0 0 0 0 ...

0 0 0 0 0 ...

0 1 1 1 0 ...

1 0 0 0 0 ...

0 1 1 1 0 ...

0 0 0 0 1 ...

0 1 1 1 0 ]';

letter\_i = [0 0 0 0 0 ...

0 0 0 0 0 ...

0 0 1 0 0 ...

0 0 0 0 0 ...

0 0 1 0 0 ...

0 0 1 0 0 ...

0 0 1 0 0 ]';

letter\_m = [0 0 0 0 0 ...

0 0 0 0 0 ...

1 1 0 1 1 ...

1 0 1 0 1 ...

1 0 1 0 1 ...

1 0 1 0 1 ...

1 0 1 0 1 ]';

letter\_o = [0 0 0 0 0 ...

0 0 0 0 0 ...

0 1 1 1 0 ...

1 0 0 0 1 ...

1 0 0 0 1 ...

1 0 0 0 1 ...

0 1 1 1 0 ]';

letter\_n = [0 0 0 0 0 ...

0 0 0 0 0 ...

1 1 1 0 0 ...

1 0 0 1 0 ...

1 0 0 1 0 ...

1 0 0 1 0 ...

1 0 0 1 0 ]';

letter\_d = [0 0 0 0 0 ...

0 0 0 0 0 ...

0 0 0 1 0 ...

0 1 1 1 0 ...

1 0 0 1 0 ...

1 0 0 1 0 ...

0 1 1 1 0 ]';

letter\_e = [0 0 0 0 0 ...

0 0 0 0 0 ...

0 1 1 0 0 ...

1 0 0 1 0 ...

1 1 1 1 0 ...

1 0 0 0 0 ...

0 1 1 1 0 ]';

letter\_l = [0 0 0 0 0 ...

0 0 0 0 0 ...

0 1 0 0 0 ...

0 1 0 0 0 ...

0 1 0 0 0 ...

0 1 0 0 0 ...

0 1 1 0 0 ]';

letter\_a = [0 0 0 0 0 ...

0 0 0 0 0 ...

0 1 1 0 0 ...

1 0 0 1 0 ...

1 0 0 1 0 ...

1 0 0 1 0 ...

0 1 1 0 1 ]';

letter\_c = [0 0 0 0 0 ...

0 0 0 0 0 ...

0 1 1 1 0 ...

1 0 0 0 0 ...

1 0 0 0 0 ...

1 0 0 0 0 ...

0 1 1 1 0 ]';

% creating the dataset with all letters

letters = [letter\_s, letter\_i, letter\_m, letter\_o, letter\_n, letter\_d, letter\_e, letter\_l, letter\_a, letter\_c, capital\_letters];

%% image of the ten first letters subplot\_index = 0;

for i = 1:10

letter = letters(:,i);

colormap('gray');

subplot\_index = subplot\_index + 1;

end

subplot(1,10,subplot\_index); imagesc(reshape(letter',5,7)') axis off

%% Setting the zeros equal to -1, in order for the Hopfield network to work letters(letters == 0) = -1;

%% !adding the same vector again as additional dimension letters(37:70, :) = 0;

for r = 1:36

vector = letters(1:35,r);

letters(36:70,r) = vector;

end

%% Error and storing different numbers of P P = 36;

final\_error\_values = zeros(1,P); final\_correct\_classifications = zeros(1,P); time\_steps = 1000;

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

first\_letters = letters(:, 1:P);

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

% storing the distorted letters first\_letters\_distorted = first\_letters;

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

for iter = 1:20 for i = 1:h

% determine index of the three random components that we will change first\_index = round(randi([1 35],1));

second\_index = round(randi([1 35],1));

third\_index = round(randi([1 35],1));

indices = [first\_index, second\_index, third\_index];

for in = indices

% inversing the pixel value if(first\_letters\_distorted(in,i) == 1)

first\_letters\_distorted(in,i) = -1;

else

first\_letters\_distorted(in,i) = 1;

end

end

end

error\_P\_values = zeros(1,P);

for k = 1:h

letter\_input = first\_letters\_distorted(:,k);

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

% letter

T = {letter\_input};

[letter\_output,~,~] = sim(net,{1 ,time\_steps},{},T);

letter\_output = letter\_output{1, time\_steps};

% rounding the values certainly to a 1 or -1 for output\_rounding = 1:70

if(letter\_output(output\_rounding) > 0)

letter\_output(output\_rounding) = 1;

else

letter\_output(output\_rounding) = -1;

end

end

% the letters to check letters\_to\_check = first\_letters;

% calculating error

if(isequal(letter\_output, letters\_to\_check(:,k)) == true)

error = 0;

else

error = sum(abs(letters\_to\_check(:,1) - letter\_output))/2;

end

error\_P\_values(1,k) = error;

end

storing\_error\_values = storing\_error\_values + error\_P\_values;

end

final\_error = median(storing\_error\_values/20);

final\_error\_values(1,h) = final\_error;

end

%% plotting error in function of P

p = plot(1:P, final\_error\_values,'-o');

p.LineWidth = 1.2;

xlabel('Number of characters P stored');

ylabel('"median of averages" error')

hold on; % hold the plot for other curves plot(22,final\_error\_values(22),'o','MarkerSize',15); txt = ('critical loading capacity');

t = text(17, final\_error\_values(17)+1.5, txt);

t.FontWeight = 'bold';