Algorithm

A restricted Boltzmann machine (RBM) was trained to recognize four of the XOR boolean patterns with a probability of 1/4 for each pattern. The RBM had three inputs and 1, 2, 3, 4 and 8 hidden neurons for different runs. It was trained for 100 trials, each trial had 40 minibatches. Weights and thresholds were updated every trial using gradient descent by back propagating in each mini batch. The learning rate was 0.005. Learning was based on Markov Chain Monte Carlo simulation by Gibbs-sampling the probability distribution of the RBM using the CD-k algorithm where k = 200. The Kullback-Leibler divergence (D_{KL}) was then computed to measure the difference between the data and the model distribution (P_{Data}, P_{Model})after training for the different numbers of hidden neurons. P_{Model} after training was sampled using an inner and outer CD-k loop using 300 respectively 200 iterations. This was sufficient enough for P_{Model} to converge to P_{Data} for 2^{N-1} hidden neurons. For more info about the RBM see chapter 4 in the book *Machine Learning for Neural Networks* by Bernard Mehlig.

Results

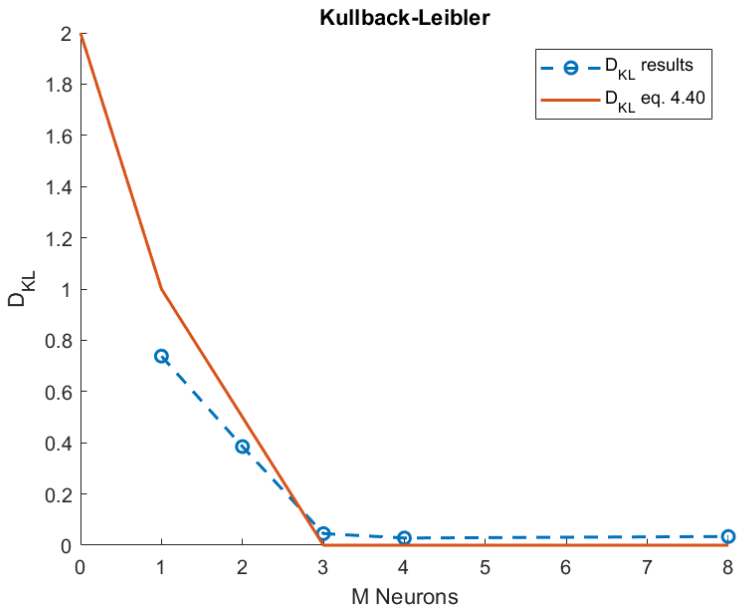


Figure 1: The estimated D_{KL} of the trained RBM for m hidden neurons vs. the hypothetical D_{KL} (eq. 4.40 in the named book).

Discussion and conclusion

It's said on p.69 in the named book that the plausible number of hidden neurons for an RBM to be sufficient to match P_{Model} to P_{Data} appropriately is 2^N neurons (N = no. of input neurons), but that 2^{N-1} number of neurons can be shown to be sufficient as well. This is what the results imply (see figure 1). P_{Model} converges to P_{Data} appropriately for 8 hidden neurons (2^N) as the rule says, but also for 4 and 3 hidden neurons. With 3 hidden neurons being sufficient enough to converge to P_{Data} shows that 2^{N-1} number of hidden neurons is sufficient for an RBM as stated in the book. P_{Model} doesn't converge entirely because that would require precision that wouldn't be realistic to implement.

Restricted Boltzmann machine

Erik Norlin

All equations are taken from the course book *Machine Learning With Neural Networks.*

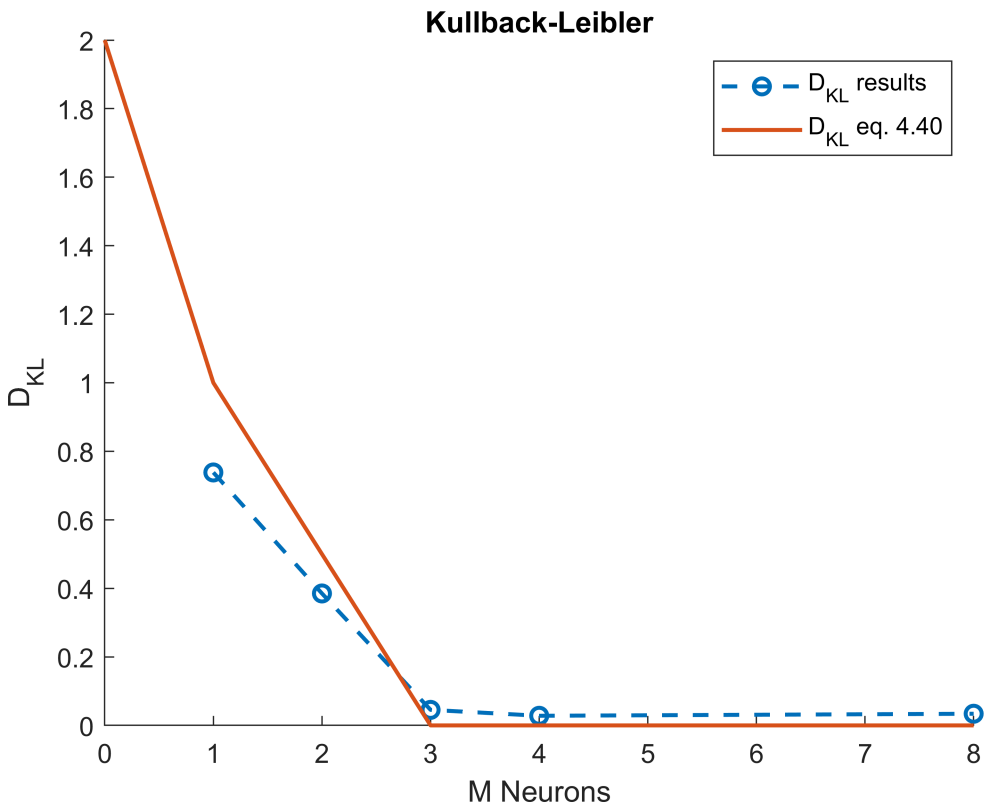
```
mHiddenNeurons = [1,2,3,4,8];

k = 200;
eta = 0.005;
nTrials = 500;
nMinibatches = 40;
nTrialsOuter = 300;
nTrialsInner = 200;

data =    [-1,-1,-1;-1,1,1;1,-1,1;1,1,-1;1,1,1;-1,1,-1;-1,-1,1;1,-1,-1];
xorData = [-1,-1,-1;-1,1,1;1,-1,1;1,1,-1];

mSetsHiddenNeurons = length(mHiddenNeurons);
nPatterns = height(data);
dataProbs = zeros(nPatterns,mSetsHiddenNeurons);
kullback = zeros(1,mSetsHiddenNeurons);

% CD-k, algorithm 3 in the course book
for mNeurons = 1:mSetsHiddenNeurons

    % Averaging the Kullback-Leibler divergence to get a more precise estimate of the RBM's dis
    nKullbackAvg = 1;
    for iKullbackAvg = 1:nKullbackAvg

        nHiddenNeurons = mHiddenNeurons(mNeurons);
        nVisibleNeurons = 3;

        w_ij = randn(nHiddenNeurons,nVisibleNeurons);
        theta_j = zeros(nVisibleNeurons,1);
        theta_i = zeros(nHiddenNeurons,1);

        h_i = zeros(nHiddenNeurons,1);
        b_i = zeros(nHiddenNeurons,1);
        v_j = zeros(nVisibleNeurons,1);
        b_j = zeros(nVisibleNeurons,1);

        probB_i = zeros(nHiddenNeurons,1);
        probB_j = zeros(nVisibleNeurons,1);

        % Training the Boltzmann machine for a number of trials
        for iTrial = 1:nTrials

            deltaWeight_ij = zeros(nHiddenNeurons,nVisibleNeurons);
            deltaTheta_j = zeros(nVisibleNeurons,1);
            deltaTheta_i = zeros(nHiddenNeurons,1);

            % Training the Boltzmann machine over minibatches
            for iMiniBatch = 1:nMinibatches
```

```
        nXorPatterns = height(xorData);
        randPattern = randi(nXorPatterns);
        v_0 = xorData(randPattern,:)';

        % Updating hidden neurons
        for iNeuron = 1:nHiddenNeurons
            b_i(iNeuron) = w_ij(iNeuron,:)*v_0 - theta_i(iNeuron);
            probB_i(iNeuron) = 1/(1 + exp(-2*b_i(iNeuron)));
            r = rand;
            if r < probB_i(iNeuron)
                h_i(iNeuron) = 1;
            else
                h_i(iNeuron) = -1;
            end
        end

        % MCMC
        v_j = v_0;
        b_i0 = b_i;
        for t = 1:k

            % Updating visible neurons
            for jNeuron = 1:nVisibleNeurons
                b_j(jNeuron) = w_ij(:,jNeuron)'*h_i - theta_j(jNeuron);
                probB_j(jNeuron) = 1/(1 + exp(-2*b_j(jNeuron)));
                r = rand;
                if r < probB_j(jNeuron)
                    v_j(jNeuron) = 1;
                else
                    v_j(jNeuron) = -1;
                end
            end

            % Updating hidden neurons
            for iNeuron = 1:nHiddenNeurons
                b_i(iNeuron) = w_ij(iNeuron,:)*v_j - theta_i(iNeuron);
                probB_i(iNeuron) = 1/(1 + exp(-2*b_i(iNeuron)));
                r = rand;
                if r < probB_i(iNeuron)
                    h_i(iNeuron) = 1;
                else
                    h_i(iNeuron) = -1;
                end
            end
        end

        % Computing weight and threshold increments
        for iNeuron = 1:nHiddenNeurons
            deltaWeight_ij(iNeuron,:) = deltaWeight_ij(iNeuron,:) + eta*(tanh(b_i0(iNeu
        end
        deltaTheta_j = deltaTheta_j - eta*(v_0 - v_j);
        deltaTheta_i = deltaTheta_i - eta*(tanh(b_i0) - tanh(b_i));
    end
```

```
        % Updating weights and thresholds
        w_ij = w_ij + deltaWeight_ij;
        theta_j = theta_j + deltaTheta_j;
        theta_i = theta_i + deltaTheta_i;
    end

    % Computing the distribution of the trained Boltzmann machine
    counter = zeros(nPatterns,1);

    for iTrialOuter = 1:nTrialsOuter

        randPattern = randi(nPatterns);
        v_j = data(randPattern,:)';

        % Updating hidden neurons
        for iNeuron = 1:nHiddenNeurons
            b_i(iNeuron) = w_ij(iNeuron,:)*v_j - theta_i(iNeuron);
            probB_i(iNeuron) = 1/(1 + exp(-2*b_i(iNeuron)));
            r = rand;
            if r < probB_i(iNeuron)
                h_i(iNeuron) = 1;
            else
                h_i(iNeuron) = -1;
            end
        end

        for iTrialInner = 1:nTrialsInner

            % Updating visible neurons
            for jNeuron = 1:nVisibleNeurons
                b_j(jNeuron) = w_ij(:,jNeuron)'*h_i - theta_j(jNeuron);
                probB_j(jNeuron) = 1/(1 + exp(-2*b_j(jNeuron)));
                r = rand;
                if r < probB_j(jNeuron)
                    v_j(jNeuron) = 1;
                else
                    v_j(jNeuron) = -1;
                end
            end

            % Updating hidden neurons
            for iNeuron = 1:nHiddenNeurons
                b_i(iNeuron) = w_ij(iNeuron,:)*v_j - theta_i(iNeuron);
                probB_i(iNeuron) = 1/(1 + exp(-2*b_i(iNeuron)));
                r = rand;
                if r < probB_i(iNeuron)
                    h_i(iNeuron) = 1;
                else
                    h_i(iNeuron) = -1;
                end
            end

            for muPattern = 1:nPatterns
```

```
                    target = data(muPattern,:)';
                    if isequal(v_j,target)
                        counter(muPattern) = counter(muPattern) + 1;
                        break
                    end
                end
            end
        end

        % Avergaing the distribution due to inner and outer trial loops
        for muPattern = 1:nPatterns
            dataProbs(muPattern,mNeurons) = counter(muPattern)/(nTrialsOuter*nTrialsInner);
        end

        % Computing the Kullback-Leibler divergence with the numerical results of the distribut
        pB = dataProbs(:,mNeurons);
        logPb = zeros(nPatterns,1);

        pData = 1/nXorPatterns;
        logPdata = log(pData);

        kullbackTemp = 0;
        for muPattern = 1:4
            if pB(muPattern) == 0
                logPb(muPattern) = 0;
            else
                logPb(muPattern) = log(pB(muPattern));
            end
            if muPattern > nXorPatterns
                pData = 0;
                logPdata = 0;
            end
            kullback(mNeurons) = kullback(mNeurons) + pData*(logPdata - logPb(muPattern));
        end
    end
    % Avergaing the Kullback-Leibler divergence
    kullback(mNeurons) = kullback(mNeurons) / nKullbackAvg;
end

% Computing the Kullback-Leibler divergence theory
kullbackTheoryX = 0:1:8;
kullbackTheoryY = kullbackTheoryX;
kullbackTheoryY = nVisibleNeurons - floor(log2(kullbackTheoryY' + 1)) - (kullbackTheoryY' + 1)
kullbackTheoryY(4:9) = 0;

% Plotting the Kullback-Leibler divergence theory and numerical results as a function of M hidd
figure
hold on
plot(mHiddenNeurons,kullback,"--o","Linewidth",1.5)
plot(kullbackTheoryX,kullbackTheoryY,"-","Linewidth",1.5)
title("Kullback-Leibler")
xlabel("M Neurons")
ylabel("D_{KL}")
legend("D_{KL} results","D_{KL} eq. 4.40")
```

```
hold off
```




```
dataProbs
```

```
dataProbs = 8×5
    0.1297    0.1033    0.1975    0.1952    0.2681
    0.1038    0.1231    0.2719    0.2586    0.2994
    0.0800    0.2920    0.2925    0.3136    0.2388
    0.1892    0.2254    0.2074    0.2205    0.1778
    0.0916    0.0075    0.0044    0.0024    0.0054
    0.1688    0.1365    0.0072    0.0030    0.0040
    0.0905    0.1041    0.0102    0.0037    0.0036
    0.1464    0.0081    0.0089    0.0029    0.0028
```

Classification challenge

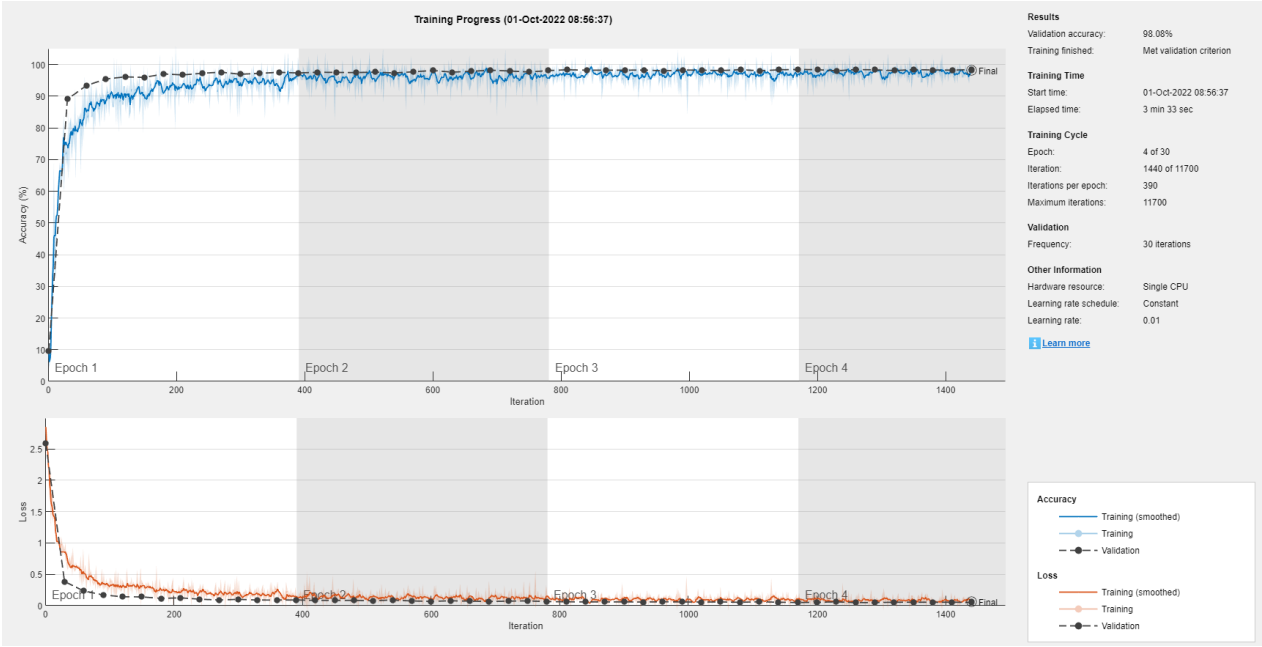
Erik Norlin

```
% Using Matlabs deep learning toolbox to classify digits.
% Code is essentially only settings for the package to run.
% Most of the settings used can be found here:
% https://se.mathworks.com/help/deeplearning/ug/create-simple-deep-learning-network-for-classif
% https://se.mathworks.com/help/deeplearning/ref/trainnetwork.html

% Loading training, validation and test set
[xTrain, tTrain, xValid, tValid, xTest, tTest] = LoadMNIST(3);
```

```
Preparing MNIST data...
MNIST data preparation complete.
```

```
xTest2 = loadmnist2();

% Processing training set to prevent overfitting during training
imageAugmenter = imageDataAugmenter( ...
    'RandRotation',[-20,20], ...
    'RandXTranslation',[-3 3], ...
    'RandYTranslation',[-3 3]);
imageSize = [28 28 1];
augTrain = augmentedImageDatastore(imageSize,xTrain,tTrain,'DataAugmentation',imageAugmenter);

layers = [
    imageInputLayer([28 28 1])

    convolution2dLayer(3,8,'Padding','same')
    batchNormalizationLayer
    reluLayer

    maxPooling2dLayer(2,'Stride',2)

    convolution2dLayer(3,16,'Padding','same')
    batchNormalizationLayer
    reluLayer

    maxPooling2dLayer(2,'Stride',2)

    convolution2dLayer(3,32,'Padding','same')
    batchNormalizationLayer
    reluLayer

    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer];

options = trainingOptions('sgdm', ...
    'InitialLearnRate',0.01, ...
    'Momentum',0.9, ...
    'MaxEpochs',30, ...
    'ValidationPatience',5, ...
```

```
    'ValidationData',{xValid,tValid}, ...
    'Shuffle','every-epoch', ...
    'ValidationFrequency',30, ...
    'Verbose',false, ...
    'Plots','training-progress');

net = trainNetwork(augTrain,layers,options);
```




```
tPredict = classify(net,xTest2);
writematrix(tPredict,"classifications.csv")
```

One-layer perceptron

Erik Norlin

```
trainingSet = readmatrix("training_set.csv");
validationSet = readmatrix("validation_set.csv");

% Normalizing the training and validation data set
nTrainingSetPatterns = height(trainingSet);
meanTrainingSet1 = 0;
meanTrainingSet2 = 0;
for i = 1:nTrainingSetPatterns
    meanTrainingSet1 = meanTrainingSet1 + trainingSet(i,1);
    meanTrainingSet2 = meanTrainingSet2 + trainingSet(i,2);
end
meanTrainingSet1 = meanTrainingSet1 / nTrainingSetPatterns;
meanTrainingSet2 = meanTrainingSet2 / nTrainingSetPatterns;

varianceTrainingSet1 = 0;
varianceTrainingSet2 = 0;
for i = 1:nTrainingSetPatterns
    varianceTrainingSet1 = varianceTrainingSet1 + (trainingSet(i,1) - meanTrainingSet1)^2;
    varianceTrainingSet2 = varianceTrainingSet2 + (trainingSet(i,2) - meanTrainingSet2)^2;
end
standardDeviationTrainingSet1 = sqrt(varianceTrainingSet1 / nTrainingSetPatterns);
standardDeviationTrainingSet2 = sqrt(varianceTrainingSet2 / nTrainingSetPatterns);

normTrainingSet = zeros(nTrainingSetPatterns,3);
for i = 1:nTrainingSetPatterns
    normTrainingSet(i,1) = (trainingSet(i,1) - meanTrainingSet1) / standardDeviationTrainingSet
    normTrainingSet(i,2) = (trainingSet(i,2) - meanTrainingSet2) / standardDeviationTrainingSet
    normTrainingSet(i,3) = trainingSet(i,3);
end

nValidationSetPatterns = height(validationSet);
normValidationSet = zeros(nValidationSetPatterns,3);
for i = 1:nValidationSetPatterns
    normValidationSet(i,1) = (validationSet(i,1) - meanTrainingSet1) / standardDeviationTrainin
    normValidationSet(i,2) = (validationSet(i,2) - meanTrainingSet2) / standardDeviationTrainin
    normValidationSet(i,3) = validationSet(i,3);
end

nHiddenNeurons = 12;
nEpochs = 150;
eta = 0.011;
classError = zeros(nEpochs,1);

nInputNeurons = 2;
nOutputNeurons = 1;

v_j = zeros(nHiddenNeurons,1);
w_ij = randn(nOutputNeurons,nHiddenNeurons);
w_jk = randn(nHiddenNeurons,nInputNeurons);
theta_j = zeros(nHiddenNeurons,1);
```

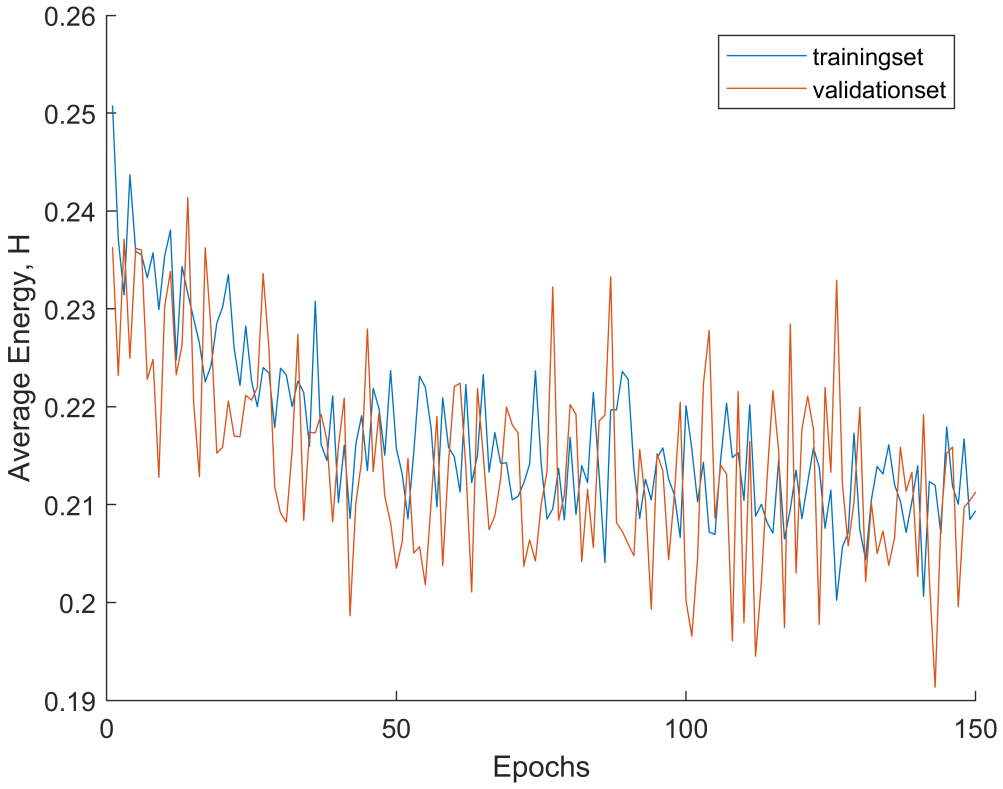
```
theta_i = 0;

b_j = zeros(nHiddenNeurons,1);
b_i = zeros(nOutputNeurons,1);

deltaError1 = zeros(nOutputNeurons,1);
deltaError2 = zeros(nHiddenNeurons,1);

deltaWeight_ij = zeros(nOutputNeurons,nHiddenNeurons);
deltaWeight_jk = zeros(nHiddenNeurons,nInputNeurons);
deltaTheta_j = zeros(nHiddenNeurons,1);
deltaTheta_i = zeros(nOutputNeurons,1);

energyTrainingSet = zeros(nEpochs,1);
energyValidationSet = zeros(nEpochs,1);
minClassError = 1;

% Training the network
for iEpoch  = 1:nEpochs

    nPatterns = nTrainingSetPatterns;
    for muPattern = 1:nPatterns

        randomPattern = randi(nPatterns);
        inputPattern = normTrainingSet(randomPattern,:);
        x_k = inputPattern(1:2)';
        target = inputPattern(3);

        % Forward propagating
        for jNeuron = 1:nHiddenNeurons
            b_j(jNeuron) = w_jk(jNeuron,:)*x_k - theta_j(jNeuron);
            v_j(jNeuron) = tanh(b_j(jNeuron));
        end

        b_i = w_ij*v_j - theta_i;
        output = tanh(b_i);

        % Calculating the energy of the output
        energyTrainingSet(iEpoch) = energyTrainingSet(iEpoch) + ((target - output)^2)/2;

        % Computing errors and backpropagating
        deltaError1 = (target - output)*((sech(b_i))^2);
        for jNeuron = 1:nHiddenNeurons
            deltaError2(jNeuron) = deltaError1*(w_ij(jNeuron)*((sech(b_j(jNeuron)))^2));
        end

        for jNeuron = 1:nHiddenNeurons
            deltaWeight_ij(jNeuron) = eta*deltaError1*v_j(jNeuron);
        end
        for kInput = 1:nInputNeurons
            for jNeuron = 1:nHiddenNeurons
               deltaWeight_jk(jNeuron,kInput) = eta*deltaError2(jNeuron)*x_k(kInput);
            end
        end
```

```
        deltaTheta_i = -eta*deltaError1;
        for jNeuron = 1:nHiddenNeurons
            deltaTheta_j(jNeuron) = -eta*deltaError2(jNeuron);
        end

        % Updating weights and thresholds
        w_ij = w_ij + deltaWeight_ij;
        theta_i = theta_i + deltaTheta_i;

        w_jk = w_jk + deltaWeight_jk;
        theta_j = theta_j + deltaTheta_j;
    end

    % Validating the trained network with the validation data set
    pVal = nValidationSetPatterns;
    for muPattern = 1:pVal

        % Forward propagating a random pattern from the valiadation set
        randomPattern = randi(muPattern);
        inputPattern = normValidationSet(randomPattern,:);
        x_k = inputPattern(1:2)';
        target = inputPattern(3);

        for jNeuron = 1:nHiddenNeurons
            b_j(jNeuron) = w_jk(jNeuron,:)*x_k - theta_j(jNeuron);
            v_j(jNeuron) = tanh(b_j(jNeuron));
        end

        b_i = w_ij*v_j - theta_i;
        output = tanh(b_i);
        if output == 0
            output = 1;
        end

        % Calculating energy and classification error
        classError(iEpoch)= classError(iEpoch) + abs(sign(output) - target);
        energyValidationSet(iEpoch) = energyValidationSet(iEpoch) + ((target - output)^2)/2;
    end

    classError(iEpoch) = classError(iEpoch)/(2*pVal);
    if classError(iEpoch) < minClassError
        minClassError = classError(iEpoch);
    end
end

writematrix(w_jk,"w1.csv");
writematrix(w_ij','w2.csv");
writematrix(theta_j,"t1.csv");
writematrix(theta_i,"t2.csv");

% Averaging the energy from respective data set to be able to compare them
energyTrainingSet = energyTrainingSet ./ height(normTrainingSet);
energyValidationSet = energyValidationSet ./ height(normValidationSet);
```

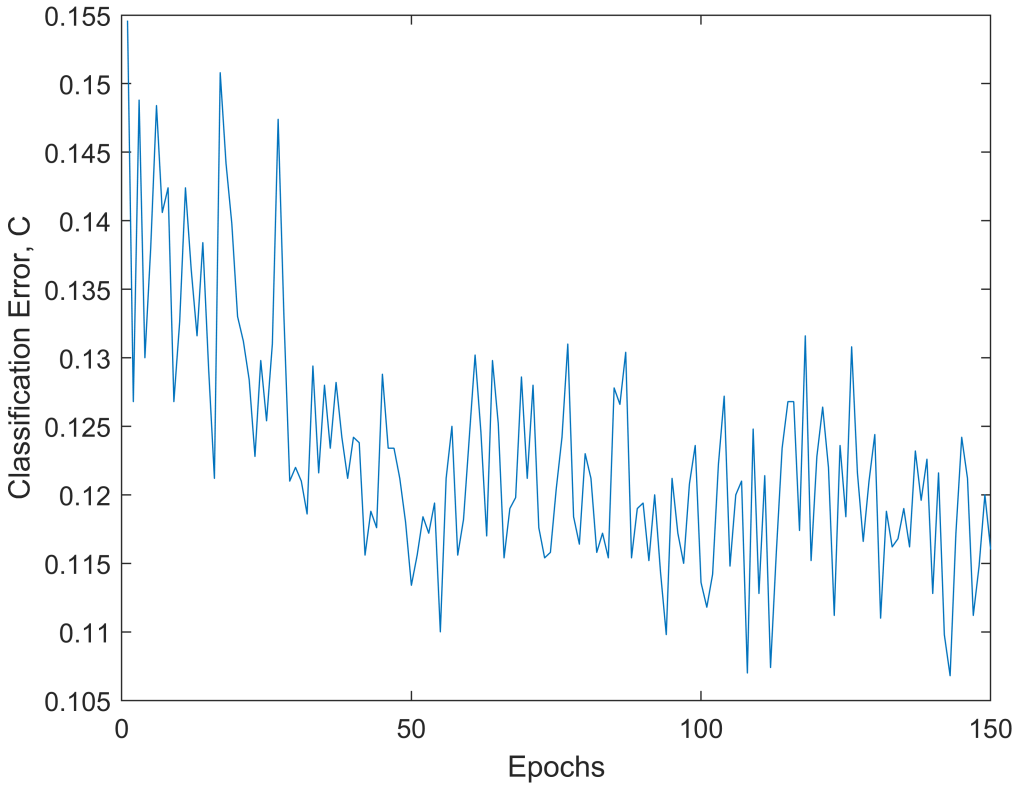
```
figure
hold on
plot(1:nEpochs,energyTrainingSet,"-");
plot(1:nEpochs,energyValidationSet,"-");
xlabel("Epochs")
ylabel("Average Energy, H")
legend({'trainingset','validationset'});
minClassErrorProcent = minClassError*100;
fprintf("Number of hidden neurons: %d\nClassification error: %.1f%%", nHiddenNeurons,minClassEr
```

```
Number of hidden neurons: 12
Classification error: 10.7%
```

```
hold off
```




```
figure
plot(1:nEpochs,classError,"-");
xlabel("Epochs")
ylabel("Classification Error, C")
```


0.155
0.15
0.145
0.14
0.135
0.13
0.125
0.12
0.115
0.11
0.105
Classification Error, C
0
50
100
150
Epochs