Code for Classification challenge

```
% Classification challange written by: Axel Qvarnström
clear all
close all
clc
xTest2 = loadmnist2();
xTest2 = cast(xTest2,'double');
[xTrain, tTrain, xValid, tValid, xTest, tTest] = LoadMNIST(3);

% Casting the data so it becomes 4-D doubles
xTrain = cast(xTrain,'double');
xValid = cast(xValid,'double');
xTest = cast(xTest,'double');

% Preprocessing the data
trainMean = mean(xTrain,4);
trainStd = std(xTrain,1,4);

validMean = mean(xValid,4);
validStd = std(xValid,1,4);

testMean = mean(xTest,4);
testStd = std(xTest,1,4);

xTest2Mean = mean(xTest2,4);
xTest2Std = std(xTest2,1,4);

xTrain = (xTrain - trainMean)./max(max(trainStd));
xValid = (xValid - validMean)./max(max(validStd));
xTest = (xTest - testMean)./max(max(testStd));
xTest2 = (xTest2 - xTest2Mean)./max(max(xTest2Std));


% Creating the network
layers= [
    imageInputLayer([28 28 1])
    convolution2dLayer(3,64,'Padding','same')
    batchNormalizationLayer
    reluLayer

    maxPooling2dLayer(2,'Stride',2)

    convolution2dLayer(3,64,'Padding','same')
    batchNormalizationLayer
    reluLayer

    maxPooling2dLayer(2,'Stride',2)
    fullyConnectedLayer(100)  % Hidden layer 1 with 100 neurons
    batchNormalizationLayer
    reluLayer

    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer];

% Options
options = trainingOptions(...
    'sgdm',...
    'ExecutionEnvironment',...
    'gpu',...
    'InitialLearnRate',0.01,...
    'Momentum',0.9,...
    'MaxEpochs',30,...
    'MiniBatchSize',100,...
    'ValidationFrequency',30,...
    'ValidationPatience',5,...
    'ValidationData',{xValid,tValid});

% Training the network
trainingNetwork = trainNetwork(xTrain,tTrain,layers,options);
save trainingNetwork       % Saving the network, so I don't need to run
                           % it again

%% Making the classification with the saved network written above
load trainingNetwork.mat   % Getting the above saved network
                           % to run this section


% Just to see the accuracy of classify the xTest data
classifyXTest = classify(trainingNetwork,xTest);
classificationAccuracy1 = sum(classifyXTest==tTest)/numel(tTest);

% Classify xTest2 and write the result to a csv file
classifyXTest2 = classify(trainingNetwork,xTest2);
classifyXTest2 = string(classifyXTest2);
classifyXTest2 = double(classifyXTest2);
csvwrite('classifications.csv',classifyXTest2);
```

Code for One-layer perceptron task. Note! Two function files is below the main code

```
% Written by Axel Q
clear all
close all
clc

% Loading the data sets
trainingDataSet = load('training_set.csv');
validationDataSet = load('validation_set.csv');

% Training patterns and targets
inputPatterns = trainingDataSet(:,[1, 2]);
targets = trainingDataSet(:,3);

% Validation patterns and targets
validationPatterns = validationDataSet(:,[1 2]);
validationTargets = validationDataSet(:,3);

% Standardization of the data
meanXTrain = mean(inputPatterns);
stdXTrain = std(inputPatterns);
inputPatterns = inputPatterns - meanXTrain; % scaling
inputPatterns = inputPatterns./stdXTrain;
validationPatterns = validationPatterns - meanXTrain;
validationPatterns = validationPatterns./stdXTrain;



M1 = 10;                                   % Number of neurons in hidden layer
pVal = length(validationDataSet);          % Number of patterns in validation set
nPatterns = length(inputPatterns);         % Number of patterns in training set
eta = 0.01;                                % Learning rate

% initial weights and threshold
weights1 = normrnd(0,1, [M1, 2]);
weights2 = normrnd(0,1,[M1, 1]);
threshold1 = zeros(M1,1);
threshold2 = 0;
nOfepoch = 0;
classificationError = 10;                 % Assign a value higher than the tolerance 0.12

% Run the algorithm until the classification error is below 12%
while classificationError > 0.12

    for i = 1:nPatterns

        % choose random input
        patternIndex = randi(nPatterns);
        inputNeurons = inputPatterns(patternIndex,:)';
        target = targets(patternIndex,:);

        % Calclulating the states of the hidden layer neurons
        b1 = LocalField(weights1, inputNeurons, threshold1);
        hiddenLayerNeurons = tanh(b1);

        % Calculating the states of the output neuron
        b2 = LocalField(weights2', hiddenLayerNeurons, threshold2);
        outputLayerNeuron = tanh(b2);

        % Calculating the output error
        delta2 = activationPrime(b2).*(target - outputLayerNeuron);

        % Error backpropagation
        delta1 = delta2 .* weights2 .* activationPrime(b1);

        % Update weights and thresholds
        deltaWeights2 = eta * delta2 * hiddenLayerNeurons;
        deltaWeights1 = eta * delta1 * inputNeurons';
        deltaThreshold2 = -eta .* delta2;
        deltaThreshold1 = -eta .* delta1;

        weights2 = weights2 + deltaWeights2;
        weights1 = weights1 + deltaWeights1;
        threshold2 = threshold2 + deltaThreshold2;
        threshold1 = threshold1 + deltaThreshold1;
    end

    % Check classification error
    classificationErrorSum = 0;
    for j = 1:pVal

        % Calulating the hidden layer neurons for the validation data
        b1Val = LocalField(weights1,validationPatterns(j,:)',threshold1);
        hiddenLayerNeuronsVal = tanh(b1Val);

        % Calculating the output neuron for the validation data
        b2Val = LocalField(weights2',hiddenLayerNeuronsVal,threshold2);
        outputLayerNeuronVal = tanh(b2Val);

        % Calculating the sum in the classification error formula
        classificationErrorSum = classificationErrorSum +...
            abs(sign(outputLayerNeuronVal) - validationTargets(j));
    end

    % Calculation the classification error and print it for each epoch
```

```
    classificationError = 1/(2*pVal) * classificationErrorSum;
    fprintf('Epoch: %0.f The classification error is: %f\n',nOfepoch,classificationError)
    nOfepoch = nOfepoch + 1;


end

% Writing to csv
csvwrite('w1.csv',weights1);
csvwrite('w2.csv',weights2);
csvwrite('t1.csv',threshold1);
csvwrite('t2.csv',threshold2);
```

Function file for local field:

```
function output = LocalField(weightVector, inputNeurons, threshold)

   output = weightVector * inputNeurons - threshold;
end
```

Function file for the derivative of the activation function:

```
function output = activationPrime(localField)

    output = 1 - tanh(localField).^2;

end
```

1 Discussion of result, restricted Boltzmann machine

I will briefly explain the code/algorithm I have implemented in matlab and then I will discuss the result given from figure 1.
So the algorithm train a restricted Boltzmann machine to learn the XOR data set. I have implement the algorithm 3 from the course book to train the restricted Boltzmann machine. When the training is done the Kullback-Leibler divergence as a function of the number of neurons M is determined. The Kullback-Leibler divergence is determined by iterate the dynamics of the restricted Boltzmann machine, where all possible patterns for 3 bits are considered. Then the probabilities for which the different patterns occur is determined. After getting the probabilities for all the patterns for all runs over the different numbers of neurons M, the Kullback-Leibler divergence is determined by equation (4.18) in the course book.

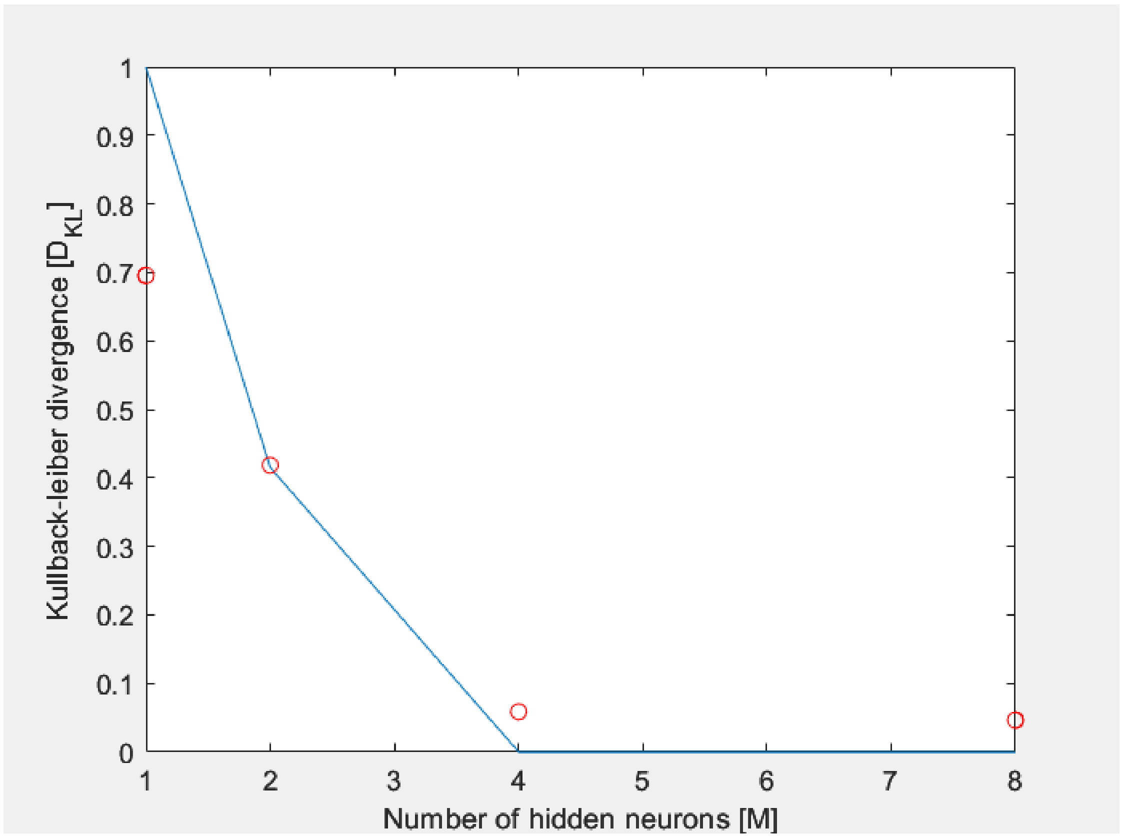

Figure 1: Red dots shows numerical estimates of the Kullback-Leibler divergence D_{KL} versus the number M of hidden neurons while the blue line shows the upper bound for D_{KL} determined as equation (4.40) in the course book

We can see in figure 1 that the restricted Boltzmann machine does approximate quite accurate but not to the optimal solution which corresponds to the blue line in the figure (equation 4.40 in the coursebook). The course book explains that the CD-k algorithm do not guaranteed to converge to the optimal solution and the CD-k algorithm is algorithm 3 that I have used to train the restricted Boltzmann machine so that's one reason why we don't get the optimal result. The result also depends a lot of the parameters, for example the learning rate, number of minibatches and so on. It also depends on how long we run the dynamics when we calculate the Kullback-Leibler divergence. I have an outer loop that runs 3000 times and an inner loop that runs 2000 times where frequencies at which the different patterns occur is determined. I have tried different numbers on the outer and inner loop but, I found kind of good results for the numbers 3000 and 2000. The probabilities for both M = 4 and M = 8 gets kind of close to what we want which is 25% for the patterns in the XOR data set and the rest of the patterns to zero probability. For M = 8 some of the patterns in XOR gets a little bit higher than 25% and some gets a little bit smaller than 25% but not much. I also get that the patterns that are not in the XOR data set occure around 0.2% which is very low, almost to zero probability.

Code for restricted Boltzmann machine task. Notice that I have a fuction at the bottom of the code!

```
% Boltzmann machine written by Axel Qvarnström
clear all
close all
clc

% Inputs
XORInputs = [-1,-1,-1; 1,-1,1; -1,1,1; 1,1,-1]';
allInputs = [-1,-1,-1; 1,-1,1; -1,1,1; 1,1,-1; 1,1,1; 1,-1,-1; -1,-1,1; -1,1,-1]';

% Probability distribution for the data
pData = [0.25, 0.25, 0.25, 0.25, 0, 0, 0, 0];

% Parameters
N = 3;
MValues = [1, 2, 4, 8];
nrOfHiddenNeurons = length(MValues);
nrOfPatterns = length(allInputs);
trials = 1000;
miniBatches = 20;
k = 2000;
eta = 0.005;
nOut = 3000;
nIn = 2000;
dKLSum = zeros(1,nrOfHiddenNeurons);

% Plotting the boundary D_kl
dKL = zeros(1, length(MValues));
for i = 1:length(MValues)
    M = MValues(i);
    if M < 2^(N-1)-1
    dKL(i) = N - log2(M+1) - ((M+1)/(2^(log2(M + 1))));
    else
    dKL(i) = 0;
    end
end
plot(MValues, dKL,'DisplayName','Upper Bound')
hold on

% Running the algorithm for the different M-values
for iHiddenNeuron = 1:nrOfHiddenNeurons
    M = MValues(iHiddenNeuron)

    % Initialize the neurons
    visibleNeurons = zeros(N,1);
    hiddenNeurons = zeros(M,1);
```

```
% Initialize the thresholds
thetaVisible = zeros(N,1);
thetaHidden = zeros(M,1);

% Initialize the weights
weights = normrnd(0, 1, [M N]);
for i = 1:size(weights,1)
        for j = 1:size(weights,2)
            if j == i
                weights(i,i) = 0;      % Making the diagonal weights to zero
            end
        end
end

for itrial = 1:trials
    % Initialize the errors
    deltaWeights = zeros(M,N);
    deltaThetaHidden = zeros(M,1);
    deltaThetaVisible = zeros(N,1);

    for iMiniBatch = 1: miniBatches
        % Pick one pattern randomly from x1-x4
        randomPatternIndex = randi(4);
        feedPattern = XORInputs(:,randomPatternIndex);
        % Initiaize visible neurons as the feed pattern
        visbleNeurons0 = feedPattern;


        % Update hidden neurons,
        localFieldHidden0 = weights * visbleNeurons0 - thetaHidden;
        hiddenNeurons = StochasticUpdate(M,localFieldHidden0);


        for t = 1:k
            % Update visible neurons
            localFieldVisible = weights' * hiddenNeurons - thetaVisible;
            visibleNeurons = StochasticUpdate(N, localFieldVisible);

            % Update hidden neurons
            localFieldHidden = weights * visibleNeurons - thetaHidden;
            hiddenNeurons = StochasticUpdate(M, localFieldHidden);
        end

        % Compute weight and threshold increments
        deltaWeights = deltaWeights + eta*(tanh(localFieldHidden0) * visbleNeurons0' - tanh(l
        deltaThetaHidden = deltaThetaHidden - eta*(tanh(localFieldHidden0) - tanh(localFieldH
        deltaThetaVisible = deltaThetaVisible - eta*(visbleNeurons0 - visibleNeurons);
        end
        % Updating weight and threshold
        weights = weights + deltaWeights;
        thetaHidden = thetaHidden + deltaThetaHidden;
        thetaVisible = thetaVisible + deltaThetaVisible;

    end

    %% Part when calculating the Kullback-Leibler divergence as a function of the number of hidd
    pB = zeros(1,nrOfPatterns);
    for iOuter = 1:nOut
        randomPatternIndex = randi(nrOfPatterns);
        feedPattern = allInputs(:,randomPatternIndex);
        % Initiaize visible neurons as the feed pattern
        visibleNeurons = feedPattern;

        localFieldHidden = weights * visibleNeurons - thetaHidden;
        hiddenNeurons = StochasticUpdate(M, localFieldHidden);


        for iInner = 1:nIn
            % Update visible neurons
            localFieldVisible = weights' * hiddenNeurons - thetaVisible;
            visibleNeurons = StochasticUpdate(N, localFieldVisible);

            % Update hidden neurons
            localFieldHidden = weights * visibleNeurons - thetaHidden;
            hiddenNeurons = StochasticUpdate(M, localFieldHidden);

            for iPattern = 1:nrOfPatterns
                if visibleNeurons == allInputs(:,iPattern)
                    pB(iPattern) = pB(iPattern) + 1/(nIn * nOut);
                end
            end
        end
    end

    % Calculating the kullback-leiber divergence
    dKL = 0;
    for mu = 1:nrOfPatterns
        if (pData(mu)~=0)
            dKL = dKL + pData(mu) * log(pData(mu)/pB(mu));
        end
    end

    dKLSum(iHiddenNeuron) = dKLSum(iHiddenNeuron) + dKL;
end
```

```
% Plotting the kullback-leiber divergence for the different M-values
plot(MValues,dKLSum,'ro','DisplayName','D_{KL}')
xlabel('Number of hidden neurons [M]')
ylabel('Kullback-leiber divergence [D_{KL}]')


% The function for the stochastic update of the neurons
function neuronValues = StochasticUpdate(nrOfNeurons,localField)

    probability = 1 ./ (1+exp(-2.*localField));
    neuronValues = zeros(nrOfNeurons,1);

    for i = 1:nrOfNeurons
        iProbability = probability(i);
        randomNumber = rand;
        if randomNumber < iProbability
            neuronValues(i) = 1;
        else
            neuronValues(i) = -1;
        end
    end
end
```