Chalmers University of Technology

FFR135 - Artificial Neural Networks

Examples sheet 3

Jacopo Credi
(910216-T396)

October 7, 2015

Problem 1

Classes	Subclasses	LS	n^o configurations
1) All of the same colour	(none)	yes	2
2) 1 of a colour, 7 of the other	(none)	yes	16
3) 2 of a colour, 6 of the other	3a) Two vertices sharing an edge	yes	24
	3b) Two vertices on square diagonal	no	24
	3c) Two vertices on cube diagonal	no	8
4) 3 of a colour, 5 of the other	4a) Three vertices forming a V	yes	48
	4b) Two vertices adjacent, other not	no	48
	4c) Three vertices not adjacent	no	16
5) 4 of a colour, 4 of the other	5a) Four vertices on a face	yes	6
	5b) Four vertices forming a star	yes	8
	5c) Vertices adjacent two by two, opposite edges	no	6
	5d) Four vertices forming two connected V's	no	24
	5e) Three vertices forming a V, one not adjacent	no	24
	5f) Four vertices non-adjacent	no	2
		Subtotal LS	104
		Subtotal non-LS	152
		Total	256

Table 1: Classification of Boolean functions with three inputs in linearly separable (LS) or not, seen as the problem of colouring the vertices of a cube with two colours.

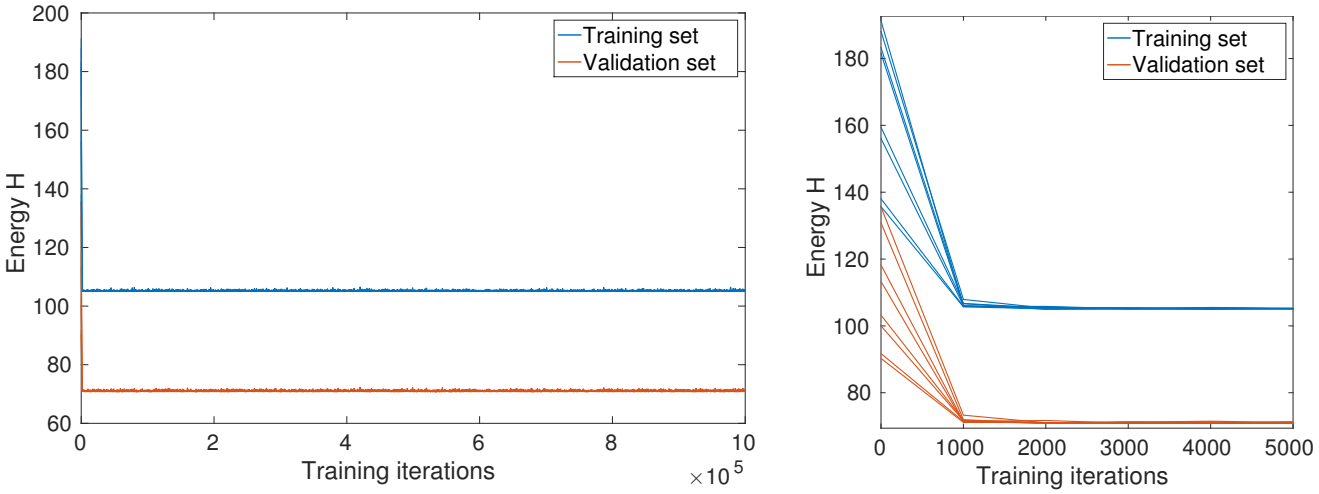
Discussion

The problem of classifying three-input Boolean functions as linearly separable (LS) or not can be visualised as the problem of colouring the vertices of a $(2 \times 2 \times 2)$-cube with two colours and then finding a plane separating vertices of different colour.

Table 1 describes the different classes and subclasses of colouring configurations, the number of such configurations in each subclass and whether they are linearly separable or not. The subclasses were found by visualising the $(2 \times 2 \times 2)$-cube and considering the possible ways of colouring n vertices with one colours and $8 - n$ with the other colour, for $n \in \{0, 1, 2, 3, 4\}$ (other cases are symmetrical). The number of configurations in each subclass was found by considering all possible rotations and reflections of a configuration of the subclass.

Out of the $2^{2^3} = 256$ Boolean functions with three inputs, 104 are found to be linearly separable and 152 non-linearly separable.

Problem 2(a)



(a) (b)

Figure 1: Left panel: energy H vs training iterations for both the training set (blue) and the validation set (red). Energy was evaluated every 1000 iterations. Ten independently initialised runs are displayed, with the same colour-coding. Right: zoom of the first 5000 iterations.

Discussion

A simple perceptron with 2 input neurons, 1 output neuron and no hidden layer was trained with asynchronous backpropagation algorithm, using the provided training data. Data was first preprocessed, by subtracting the sample mean and dividing by the sample standard deviation.

Network weights were initialised randomly with uniform distribution in $[0.2, 0.2]$ and biases with uniform distribution in $[1, 1]$. The activation function was taken to be $g(z) = \tanh(\beta z)$, with parameter $\beta = 1/2$. The learning rate was set to $\eta = 0.01$. Ten independently initialised training runs were performed, with 10^6 training iterations in each run.

At each iteration, a random pattern $\boldsymbol{\xi}^{(\mu)}$ is used as input and the output value is computed using McCulloch-Pitts dynamics, i.e. $O^{(\mu)} = g(b)$, where $b = \boldsymbol{w}\ \boldsymbol{\xi}^{(\mu)}$ and the threshold/bias is incorporated in the weight vector $\boldsymbol{w}$, using an extra (dummy) neuron with fixed state -1. The weights are then updated as $\boldsymbol{w} \to \boldsymbol{w} + \boldsymbol{\delta w}$, with $\boldsymbol{\delta w} = \eta\ \Delta\ (\boldsymbol{\xi}^{(\mu)})^T$, where $\Delta = (\zeta^{(\mu)} - O^{(\mu)})g'(b)$.

As shown in Fig. 1, energy initially drops very quickly (see panel b), but it then stabilises around 105 for the training set and 71 for the validation set. The difference in the energy value between training and validation sets can be explained by considering that the data sets have different size (energy is indeed an "extensive" property of the system) and therefore different values are not indicative of different performance (in fact $105/71 \simeq 1.5$, which is the ratio of the data sets size).

Since the backpropagation algorithm is based on gradient descent, the energy H is expected not to increase if using batch mode. However, in asynchronous backpropagation, energy in general exhibits oscillations (and therefore can increase), depending on the learning rate η. The effect η was investigated by performing some additional training runs with $\eta \in [0.001, 0.1]$. As one can expect, a very small η slows down the training, but yields smaller energy oscillations, whereas a larger η increases the learning speed, but leads to stronger oscillations.

Reference code

See MATLAB script `MultilayerPerceptronBackpropagation.m` and custom functions used in the script itself, in particular `RunPerceptron.m`, `Backpropagation.m` and `ComputeErrorAndEnergy.m`.

Problem 2(b)

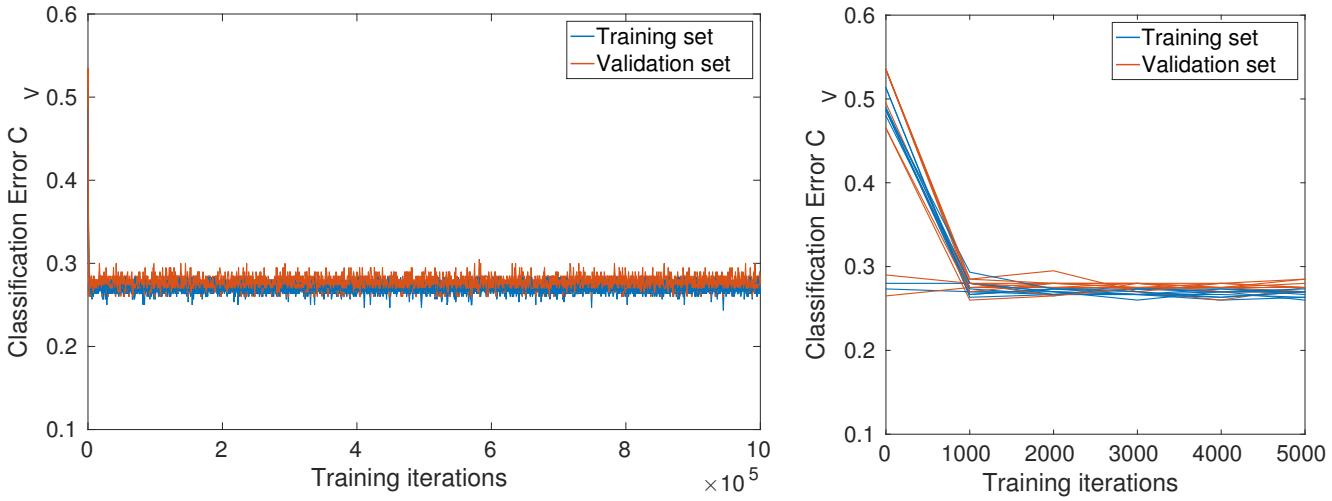


Figure 2: Left panel: classification error C_V vs training iterations for both the training set (blue) and the validation set (red). One point every 1000 iterations is shown. Ten independently initialised runs are displayed, with the same colour-coding. Right: zoom of the first 5000 iterations.

Discussion

The classification error of a data set with one output is defined as

$$C_V = \frac{1}{2p} \sum_{\mu=1}^{p} \mid \zeta^{(\mu)} - \mathrm{sgn}(O^{(\mu)}) \mid , \tag{1}$$

where p is the number of patterns in the set and the factor $1/2$ was added to have $C_V \in [0, 1]$. The classification error was computed every 1000 training iterations for both the training and the validation set (Fig. 2). This plots clearly show that the classification performance of this simple perceptron is quite poor: even though the error quickly drops from around 0.5 (i.e. random classification) to around 0.27 for both training and validation sets, then it does not really decrease any further.

In other terms, the perceptron is incorrectly classifying patterns more than 25% of the time, suggesting that this simple network architecture is unsuitable for the function that we are trying to approximate. The reason for this is that such function is not linearly separable, as one can notice by a quick plot of the patterns.

In the geometric interpretation of straight lines (or, in general, hyperplanes) separating points in the input space, the above results mean that the best straight line found by the backpropagation algorithm is such that around 27% of the points classified as "+1" actually lie below the line itself, and around 27% of the points classified as "-1" actually lie above it.

The difference between errors obtained in different runs is negligible, as there are no "local optima" to get stuck in. All training runs are equally good, or more precisely equally bad.

Reference code

See MATLAB script `MultilayerPerceptronBackpropagation.m` and custom functions used in the script itself, in particular `RunPerceptron.m`, `Backpropagation.m` and `ComputeErrorAndEnergy.m`.

Problem 2(c)

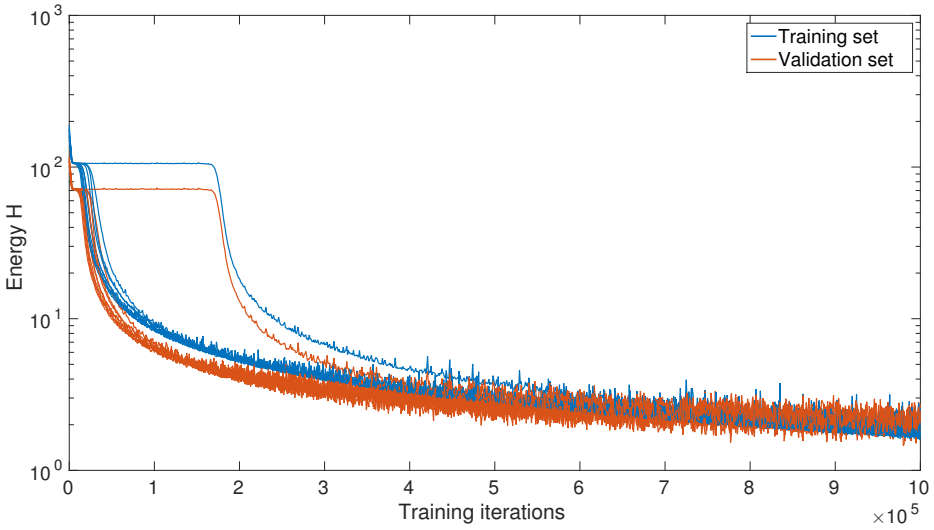


Figure 3: Energy H vs training iterations for both the training set (blue) and the validation set (red). One point every 1000 iterations is shown. Ten independently initialised runs are displayed (with the same colour-coding), y-axis is in log scale, for visual clarity.

Discussion

The network architecture was then modified, by adding a hidden layer with three neurons between input and output layers. With a slight change of notation, let us denote by $\boldsymbol{V}^{(l)}$ the vector of neuron states in layer l, with $l = 1, \dots, L$. Then, the network dynamics (proceeding forward) is governed by $\boldsymbol{V}^{(l)} = g(\boldsymbol{b}^{(l)})$, where $\boldsymbol{b}^{(l)} = W^{(l)}\, \boldsymbol{V}^{(l-1)}$ and where $W^{(l)}$ is the matrix of connections between neuron layers $l-1$ and l, and obviously $\boldsymbol{V}^{(0)} = \boldsymbol{\xi}^{(\mu)}$.

The backpropagation training rule (proceeding backwards) can be expressed in vectorial form as $W^{(l)} \to W^{(l)} + \delta W^{(l)}$, with

$$\delta W^{(l)} = \eta\, \boldsymbol{\delta}^{(l)}\, (\boldsymbol{V}^{(l-1)})^T\,, \quad \text{where} \quad \boldsymbol{\delta}^{(l)} = \begin{cases} g'(\boldsymbol{b}^{(l)}) \circ (\boldsymbol{\zeta}^{(\mu)} - \boldsymbol{V}^{(l)\mu}) & \text{for } l = L \\ g'(\boldsymbol{b}^{(l)}) \circ (W^{(l+1)})^T \boldsymbol{\delta}^{(l+1)} & \text{for } l < L\,. \end{cases} \tag{2}$$

The symbol $\circ$ in previous equations denotes the Hadamard product (or entrywise product) between vectors of the same length. The above rules were implemented in MATLAB functions able to handle a perceptron of any size. Again, thresholds/biases were encoded inside the weight matrices $W^{(l)}$ (and thus a dummy neuron with fixed state -1 was added in each layer $l < L$). The program was then run with the same parameters as in part **2a**, i.e. $\eta = 0.01$, $\beta = 1/2$, ten independent runs with 10^6 training iterations per run.

Plots in Fig. 3 show that the energy decreases, albeit non monotonically (due to the stochastic update), reaching very low values at the end of the training. In one case, a local minimum was found, but the algorithm was eventually able to escape from it. Performance is clearly much better than in the case of a simple perceptron, since the hidden layer allows a good approximation of the non linearly separable objective function producing the data.

Reference code

See MATLAB script `MultilayerPerceptronBackpropagation.m` and custom functions used in the script itself, in particular `RunPerceptron.m`, `Backpropagation.m` and `ComputeErrorAndEnergy.m`.

Problem 2(d)

	Classification error C_V	
Run	**Training set**	**Validation set**
1	0.0028 ± 0.0013	0.006 ± 0.002
2	0.0024 ± 0.0015	0.0054 ± 0.0014
3	0.0024 ± 0.0016	0.0052 ± 0.0009
4	0.0027 ± 0.0014	0.0052 ± 0.0010
5	0.0026 ± 0.0016	0.0052 ± 0.0010
6	0.0030 ± 0.0012	0.006 ± 0.002
7	0.0026 ± 0.0015	0.0053 ± 0.0011
8	0.0026 ± 0.0014	0.0053 ± 0.0012
9	0.0025 ± 0.0016	0.0055 ± 0.0016
10	0.0025 ± 0.0016	0.0054 ± 0.0013

Table 2: Classification error for both the training and the validation set in ten independent training runs of a multilayer perceptron with 3 hidden neurons. Values are averages of the last 100 values computed every 1000 iterations, i.e. they refer to the last 100000 iterations.

Discussion

With this network architecture (one hidden layer with three neurons), the perceptron performance in classifying input patterns increases dramatically if compared to the simple perceptron with no hidden layer.

As shown in Table 2, the average classification error computed over the last 250000 iterations is consistently below 0.01 for both the training and validation data sets, meaning that the perceptron correctly classifies more than 99% of the input patterns.

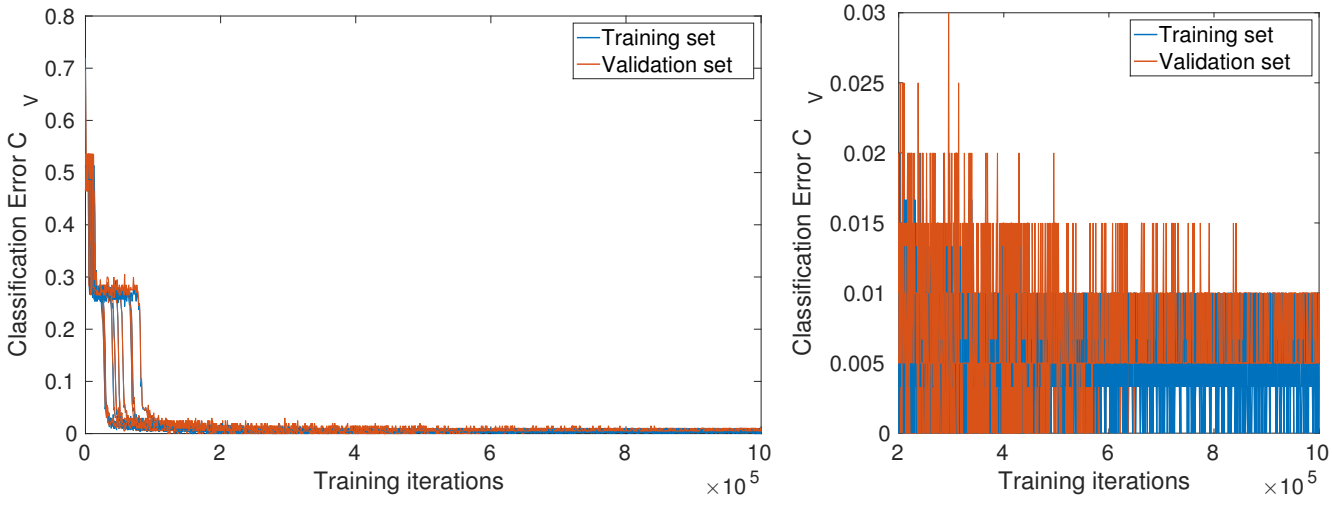
The classification error of the validation data set was found to be slightly larger than that of the training set. This can be due to the fact that training is performed using the training set, as the name says, and the noise in the data, albeit very low, can have a little effect on the perceptron performance on the validation set.

Zero classification error for both the training and validation set can be actually obtained if one performs training runs in which C_V is computed much more often than once every 1000 training iterations. Some experimental runs with error evaluation every 50 iterations, with the same parameters as in **2c**, showed that zero error can be obtained in almost all runs ($\sim 90\%$) after less than $2 \cdot 10^5$ iterations, on average.

Reference code

See MATLAB script `MultilayerPerceptronBackpropagation.m` and custom functions used in the script itself, in particular `RunPerceptron.m`, `Backpropagation.m` and `ComputeErrorAndEnergy.m`.

Problem 2(e)



(a) (b)

Figure 4: Left panel: classification error C_V vs training iterations for both the training set (blue) and the validation set (red). One point every 1000 iterations is shown. Ten independently initialised runs are displayed (with the same colour-coding). right axis: detail of the plot in the left panel.

Discussion

The network architecture was further modified by adding a second hidden layer with two neurons. As explained in **2c**, implemented MATLAB functions `RunPerceptron.m` and `Backpropagation.m` can handle multilayer perceptrons of any size.

Ten independent training runs were performed, with 10^6 iterations per run and same parameters as before. The obtained classification error, computed every 1000 iterations, is plotted in Fig 4 (left panel). Although most runs encounter local minima, in all cases the algorithm is eventually able to escape from them, reaching very small classification error. This network architecture seems therefore to give results as good as the previous one, with a single hidden layer.

However, consider the right panel of Fig 4, showing a zoom-in of the left panel, without the initial transient phase. Although quite chaotic, these plots show the emergence of overfitting. In fact, after about 600000 iterations, the average classification error of the validation set increases slightly and never drops below 0.005 afterwards, whereas in the first part of the run the error frequently dropped to zero, and if the training had been stopped there, zero classification error could have been achieved.

This suggests that a single hidden layer is sufficient to capture the features of the provided data sets (as can be intuitively seen by visualising the data). Adding an extra hidden layer does not improve the perceptron performance as a classifier and may instead cause overfitting problems (fitting the noise in the data).

Reference code

See MATLAB script `MultilayerPerceptronBackpropagation.m` and custom functions used in the script itself, in particular `RunPerceptron.m`, `Backpropagation.m` and `ComputeErrorAndEnergy.m`.

Appendix: MATLAB code

MultilayerPerceptronBackpropagation.m

```
%MultilayerPerceptronBackpropagation
clear
close all
tic

% ================================================================== %
% PARAMETERS
% ================================================================== %
% can specify a perceptron of L+1 layers (any L), but use 1 output
    neuron for safety (not tested with multiple outputs, though it should
     work)
networkArchitecture = [2 3 2 1];
beta = 0.5;
learningRate = 0.01;
weightRange = 0.2;
biasRange = 1.0;
nIterations = 1e6;
iterationsBetweenPlots = 1000;
nRuns = 10;
trainingDataPath = '../Data/train_data_new.txt';
validationDataPath = '../Data/valid_data_new.txt';
% ================================================================== %

% Read and normalize training and validation data sets
[trainingPatterns, trainingOutputs] = ReadData(trainingDataPath);
[validationPatterns, validationOutputs] = ReadData(validationDataPath);
nTraining = size(trainingPatterns,1);
nValidation = size(validationPatterns,1);
[trainingPatterns, validationPatterns] = NormalizeData(trainingPatterns,
    validationPatterns);

% Add dummy neuron with fixed value -1 (for including biases in weights)
trainingPatterns = [trainingPatterns, -ones(nTraining,1)];
validationPatterns = [validationPatterns, -ones(nValidation,1)];

% Initialise output data structure
x = 0:iterationsBetweenPlots:nIterations;
nPlotPoints = length(x);
emptyData = nan(1,length(x));
outputData = repmat(struct('errorTraining',emptyData,'errorValidation',
    emptyData,'energyTraining',emptyData,'energyValidation',emptyData),
    nRuns, 1 );

%% Loop over runs
```

```
40 parfor run = 1:nRuns
41
42   % Initialize random weight matrix (or, well, cell of matrices)
43   weightsCell = InitialiseWeightsBiases(networkArchitecture, weightRange
        , biasRange);
44
45   % Initialize plots (only use if nRuns = 1)
46 %     [errorHandle, trainingErrorHandle, validationErrorHandle, ...
47 %       energyHandle, trainingEnergyHandle, validationEnergyHandle] =
      InitialisePlots(x, nPlotPoints);
48
49   tPlot = 1;
50
51   % Compute initial classification error and energy for training set
52   [trainingError, trainingEnergy] = ComputeErrorAndEnergy(weightsCell,
        beta, trainingPatterns, trainingOutputs);
53
54   % Compute initial classification error and energy for validation set
55   [validationError, validationEnergy] = ComputeErrorAndEnergy(...
56     weightsCell, beta, validationPatterns, validationOutputs);
57
58   % Store values
59   outputData(run).errorTraining(tPlot) = trainingError;
60   outputData(run).energyTraining(tPlot) = trainingEnergy;
61   outputData(run).errorValidation(tPlot) = validationError;
62   outputData(run).energyValidation(tPlot) = validationEnergy;
63
64   for t = 1:nIterations
65
66     % Pick pattern to feed (and corresponding output)
67     patternID = randi(nTraining);
68     inputPattern = trainingPatterns(patternID,:)';
69     desiredOutput = trainingOutputs(patternID,:)';
70
71     % Run perceptron
72     perceptronState = RunPerceptron(inputPattern, weightsCell, beta);
73
74     % Backpropagation
75     weightsCell = Backpropagation(perceptronState, weightsCell, ...
76       desiredOutput, beta, learningRate);
77
78     if mod(t,iterationsBetweenPlots) == 0
79       tPlot = tPlot+1;
80
81       % Compute and store classification error and energy for training
            set
82       [trainingError, trainingEnergy] = ComputeErrorAndEnergy(...
83           weightsCell, beta, trainingPatterns, trainingOutputs);
84
```

```
85        % Compute and store classification error and energy for validation
              set
86        [validationError, validationEnergy] = ComputeErrorAndEnergy(...
87            weightsCell, beta, validationPatterns, validationOutputs);
88
89        % Store values
90        outputData(run).errorTraining(tPlot) = trainingError;
91        outputData(run).energyTraining(tPlot) = trainingEnergy;
92        outputData(run).errorValidation(tPlot) = validationError;
93        outputData(run).energyValidation(tPlot) = validationEnergy;
94
95        % Update plots (only use if nRuns = 1)
96  %       UpdatePlots(tPlot, trainingErrorHandle, trainingError, ...
97  %         validationErrorHandle, validationError, trainingEnergyHandle,
      trainingEnergy, validationEnergyHandle, validationEnergy);
98
99  %       if trainingError == 0 && validationError == 0
100 % %         load handel
101 % %         sound(y,Fs)
102 %         fprintf('\nRUN %i: zero classification error reached. Training
      stopped after %i iterations.\n.', ...
103 %           run, t);
104 %         break
105 %       end
106     end
107
108   end
109
110 end
111 [errorHandle, energyHandle] = PlotErrorEnergy(outputData,x);
112
113 toc
```

ReadData.m

```
1  function [ patterns, outputs ] = ReadData( dataFilePath )
2  %READDATA
3
4  fileID = fopen(dataFilePath);
5  data = textscan(fileID, '%f %f %f');
6  fclose(fileID);
7
8  patterns = [data{1}, data{2}];
9  outputs = data{3};
10
11 end
```

NormaliseData.m

```
function [normalisedTrainingPatterns, normalisedValidationPatterns] =
    ...
     NormaliseData(trainingPatterns, validationPatterns)
%NormaliseData

n = size(trainingPatterns,1);

tmpArray = [trainingPatterns; validationPatterns];
tmpArray(:,1) = (tmpArray(:,1) - mean(tmpArray(:,1))) ...
    ./std(tmpArray(:,1));
tmpArray(:,2) = (tmpArray(:,2) - mean(tmpArray(:,2)))...
    ./std(tmpArray(:,2));

normalizedTrainingPatterns = tmpArray(1:n,:);
normalizedValidationPatterns = tmpArray(n+1:end,:);

end
```

InitialiseWeightsBiases.m

```
function weightsCell = InitialiseWeightsBiases(neuronsInLayers, wR, bR)
%InitialiseWeightsBiases

% wR = weightRange = 0.2;
% bR = biasRange = 1.0;

L = size(neuronsInLayers,2)-1;
weightsCell = cell(1,L);

for l = 1:L
    N = neuronsInLayers(l);
    M = neuronsInLayers(l+1);

    W = [-wR + 2*wR.*rand(M,N), -bR + 2*bR.*rand(M,1)];
    weightsCell{l} = W;
end

end
```

RunPerceptron.m

```
function perceptronState = RunPerceptron(inputPattern, weightsCell,beta)
%RUNPERCEPTRON

L = size(weightsCell,2); % L+1 layers
perceptronState = cell(1,L+1);
perceptronState{1,1} = inputPattern;

for l = 1:L
    b = beta*weightsCell{1,l}*perceptronState{1,l};
    perceptronState{1,l+1} = [tanh(b); -1]; % dummy -1 added
end

% remove dummy neuron from output layer
perceptronState{1,L+1}(end) = [];
end
```

Backpropagation.m

```
function updatedWeightsCell = Backpropagation(perceptronState, ...
    weightsCell, desiredOutput, beta, learningRate)
%Backpropagation

L = length(weightsCell);
updatedWeightsCell = weightsCell;

% Update connections to output neuron(s)
gPrime = beta*(1-perceptronState{1,end}.^2); % because g = tanh
delta = (desiredOutput - perceptronState{1,end}).*gPrime;
dW = learningRate*(delta*perceptronState{1,end-1}'); % note transpose!
updatedWeightsCell{1,end} = weightsCell{1,end} + dW;

% And now backpropagate to the rest of the network (if any)
if L > 1
    for l = L-1:-1:1
        gPrime = beta*(1-(perceptronState{1,l+1}(1:end-1)).^2);
        % this is a trick to make computation easier to write and run
        tempW = weightsCell{1,l+1}';
        tempW(end,:) = [];
        delta = (tempW*delta).*gPrime;
        dw = learningRate*(delta*perceptronState{1,l}'); % transpose!
        updatedWeightsCell{1,l} = weightsCell{1,l} + dw;
    end
end

end
```

ComputeErrorAndEnergy.m

```
function [classificationError, energy] = ComputeErrorAndEnergy(...
    weightsCell, beta, inputPatterns, trueOutputs)
%COMPUTEERRORANDENERGY

nPatterns = size(inputPatterns, 1);
nOutputs = size(trueOutputs, 2);

perceptronOutputs = zeros(nPatterns, nOutputs);

for mu = 1:nPatterns
  perceptronState = RunPerceptron(inputPatterns(mu,:)', weightsCell,
     beta);
  perceptronOutputs(mu,:) = perceptronState{1,end};
end

tmpArrayError = abs(trueOutputs - sign(perceptronOutputs));
tmpArrayEnergy = (trueOutputs - perceptronOutputs).^2;

classificationError = 0.5/nPatterns*sum(tmpArrayError(:));
energy = 0.5*sum(tmpArrayEnergy(:));

end
```

InitialisePlots.m

```
function [errorHandle, trainingErrorHandle, validationErrorHandle, ...
    energyHandle, trainingEnergyHandle, validationEnergyHandle] = ...
    InitialisePlots(x, nPlotPoints)
%INITIALISEPLOTS

errorHandle = figure;
set(gcf,'color','w');
hold on;
set(errorHandle, 'Position', [100,100,800,400]);
set(errorHandle, 'DoubleBuffer','on');
xlim([0 x(end)]);
xlabel('Training iterations');
ylabel('Classification Error C_V');
set(gca, 'FontSize', 16);
trainingErrorHandle = plot(x, nan(1,nPlotPoints), 'LineWidth',1.5);
validationErrorHandle = plot(x, nan(1,nPlotPoints), 'LineWidth',1.5);
legend([trainingErrorHandle, validationErrorHandle], 'Training set', ...
    'Validation set');
hold off;
drawnow;
```

```
21
22 energyHandle = figure;
23 set(gcf,'color','w');
24 hold on;
25 set(energyHandle, 'Position', [1000,100,800,400]);
26 set(energyHandle, 'DoubleBuffer','on');
27 xlim([0 x(end)]);
28 xlabel('Training iterations');
29 ylabel('Energy H');
30 set(gca, 'FontSize', 16);
31 trainingEnergyHandle = plot(x, nan(1,nPlotPoints), 'LineWidth',1.5);
32 validationEnergyHandle = plot(x, nan(1,nPlotPoints), 'LineWidth',1.5);
33 legend([trainingEnergyHandle, validationEnergyHandle], 'Training set',
      ...
34     'Validation set');
35 hold off;
36 drawnow;
37
38 end
```

UpdatePlots.m

```
1  function [] = UpdatePlots(tPlot, trainingErrorHandle, trainingError, ...
2      validationErrorHandle, validationError, trainingEnergyHandle, ...
3      trainingEnergy, validationEnergyHandle, validationEnergy)
4  %UPDATEPLOTS
5
6  plotTrainingError = get(trainingErrorHandle,'YData');
7  plotTrainingError(tPlot) = trainingError;
8  set(trainingErrorHandle,'YData', plotTrainingError);
9
10 plotValidationError = get(validationErrorHandle,'YData');
11 plotValidationError(tPlot) = validationError;
12 set(validationErrorHandle,'YData', plotValidationError);
13
14 plotTrainingEnergy = get(trainingEnergyHandle,'YData');
15 plotTrainingEnergy(tPlot) = trainingEnergy;
16 set(trainingEnergyHandle,'YData', plotTrainingEnergy);
17
18 plotValidationEnergy = get(validationEnergyHandle,'YData');
19 plotValidationEnergy(tPlot) = validationEnergy;
20 set(validationEnergyHandle,'YData', plotValidationEnergy);
21
22 drawnow;
23
24 end
```