FFR135 - Assignment 1

Johan Björk jobjork@student.chalmers.se
Fredrik Ring ringf@student.chalmers.se

September 22, 2017

Preface

The code used to solve the tasks is written in Matlab.

1 Deterministic Hopfield Model

1a The one step error proability (P_{err}) is defined as

$$P_{err} = \mathrm{P}(C_i^\nu > 1) = \mathrm{P}(-\frac{1}{N}\sum_{j=1}^{N}\sum_{\substack{\mu=1,\\ \mu\neq\nu}}^{p}\zeta_i^\nu\zeta_i^\mu\zeta_j^\mu\zeta_j^\nu > 1),$$

where C_i^ν represents the modified cross talk term given in the lecture notes p.18 for a pattern ν [1]. N is the number of neurons in the network, and p is the number of stored patterns, both assumed to be large numbers. Since we have not assumed non-zero weights, i.e. $w_{ii} = \frac{1}{N}\sum_{\mu=1}^{p}\zeta_i^\mu\zeta_i^\mu \neq 0$, we will always get contributions from these diagonal elements in the expression for C_i^ν. Therefore, one can elaborate the expression for P_{err} as

$$\begin{aligned}
P_{err} &= \mathrm{P}(-\frac{1}{N}\sum_{j=1}^{N}\sum_{\substack{\mu=1,\\ \mu\neq\nu}}^{p}\zeta_i^\nu\zeta_i^\mu\zeta_j^\mu\zeta_j^\nu > 1)\\
&= \mathrm{P}(\frac{-(p-1)}{N} - \frac{1}{N}\sum_{\substack{j=1,\\ j\neq i}}^{N}\sum_{\substack{\mu=1,\\ \mu\neq\nu}}^{p}\zeta_i^\nu\zeta_i^\mu\zeta_j^\mu\zeta_j^\nu > 1)\\
&\approx \mathrm{P}(-\frac{1}{N}\sum_{\substack{j=1,\\ j\neq i}}^{N}\sum_{\substack{\mu=1,\\ \mu\neq\nu}}^{p}\zeta_i^\nu\zeta_i^\mu\zeta_j^\mu\zeta_j^\nu > \frac{p+N}{N}) \qquad (1)\\
&= \{\text{Central Limit Theorem}\}\\
&= \frac{1}{\sqrt{2\pi}\sigma}\int_{\frac{p+N}{N}}^{\infty}\mathrm{d}z\cdot e^{-\frac{z^2}{2\sigma^2}}\\
&= \frac{1}{2}[1-\mathrm{erf}(\frac{p+N}{\sqrt{2pN}})]
\end{aligned}$$

In the calculations above, we have in accordance with the lecture notes [1] assumed the coupled ζ-terms to be independent random variables drawn from the same distribution, which makes it possible to use the central limit theorem and thereby use the normal distribution. The $(p-1)/N$ on the second row comes from the the $p-1$ cases where $j=i$, each resulting in a term equal to one, since it gives $\zeta_i^\mu\zeta_i^\mu\zeta_i^\nu\zeta_i^\nu = 1\times 1$. This constant term is taken out of the double sum.

One way to check if this theory is correct is to look at the limit of p<<N, since this should give the results derived in the lecture notes. Looking at the final approximation of the one step error probability, $P_{err} = \frac{1}{2}[1-\mathrm{erf}(\frac{p+N}{\sqrt{2pN}})]$ and assuming p<<N we can approximate p+N $\approx$ N, resulting in $P_{err} = \frac{1}{2}[1-\mathrm{erf}(\frac{N}{\sqrt{2pN}})] = \frac{1}{2}[1-\mathrm{erf}(\sqrt{\frac{N}{2p}})]$ which is the exact expression given in the lecture notes, indicating that it is correct.

A curious observation of the obtained approximation of P_{err} is that for a fixed N the error probability will increase with p up to $p=N$, after which it will decrease towards zero. This will be elaborated on in task **1b**.

1b For simulating the one step error probability, a network with $N = 200$ neurons was implemented, and p random patterns were generated, with p ranging between the values given in the assignment description. Once the patterns were initialized, the weights were set according to Hebb's rule

$$w_{ij} := \frac{1}{N} \sum_{\mu=1}^{p} \zeta_i^\mu \zeta_j^\mu. \tag{2}$$

The state elements S_i were then evaluated using the Hopfield model.

$$S_i := \text{sgn}\left(\sum_{j=1}^{N} w_{ij} S_j \right).$$

In Matlab, a zero argument to the signum function will output a zero value. We chose to handle this case by assigning the value 1 or -1 to the state element, both with probability $\frac{1}{2}$.

Once the patterns were evaluated by the network, the Hopfield processed states were matched with the initial states, and potential non-matches (errors) were recorded. We then sampled 10^5 points from the matched states in order to compute an error probability.

The simulated values of P_{err} were recorded for each value of p, and was then plotted in relation to the ratio $\alpha = \frac{p}{N}$, along with the theoretical P_{err} obtained as in equation (1), see Figure 1.

By inspection of Figure 1 the simulated estimates of P_{err} seem to be very much aligned with the theoretical values, which indicates that our results are reasonable.

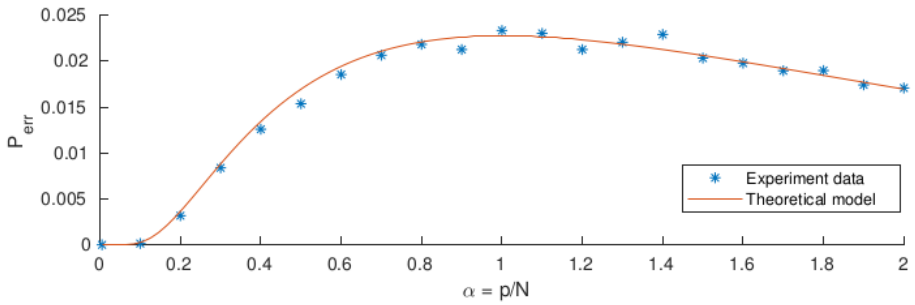

Figure 1: Plot of the analytic estimate of P_{err} (red) along with simulated values (blue).

Studying the slope after $\alpha = 1$ in Figure 1, we see that the error probability decreases with increasing p. This was also noted in **1a** when looking at the formula, but it is not obvious why this happens. Storing more and more patterns will, according to this, result in better pattern recognition. This is due to the fact that the diagonal elements w_{ii} will grow larger and larger. From the definition in Equation 2 we get that the diagonal will always have the value p/N. With larger p, w_{ii} will grow larger and larger, while the non-diagonal elements will stay small. In effect, the weight matrix will become almost a diagonal matrix, essentially outputting whatever is put in to the system. This means that any pattern that is put in to the system will return at the output. A blurred version of a pattern will then not be recognized. Essentially the network loses its memory feature this way and becomes more of a fancy copy machine, outputting whatever was put in regardless if it is stored according to Hebb's rule or not.

2 Stochastic Hopfield Model

2a The stochastic Hopfield model was implemented with the parameter settings according to the assignment description. The wandering mean of the order parameter over the iterations can be seen in Figure 2. Displaying the wandering mean instead of just the order parameter has the advantage of disregarding noise, or fluctuations with equal probability of being above or below the mean, which makes it much easier to spot when the steady state has been reached. The order parameter should reach a steady state at unity for the pattern that was fed into the network and a steady state at zero for every other pattern. This is due to the definition of the order parameter on p.37 in the lecture notes [1]. The sum will simply be a sum of N ones divided by N $(= 1)$ for the pattern that was fed, and a sum of random +1 and -1 for every other pattern with a mean of zero, since we assume the patterns to be random.

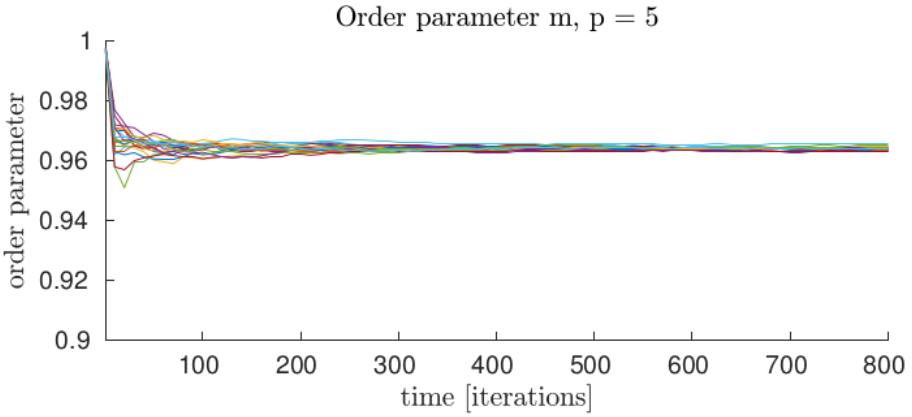

Figure 2: Wandering mean of order parameter for pattern 1 which was fed into the network with $\alpha = 0.025$. The different colors each represent one of the 20 experiments done.

One thing that is immediately obvious is that the the order parameter is not actually unity, but slightly lower. Looking at the phase diagram on p.60 we find a possible reason for why this is the case. The phase diagram shows that for a given β there will be a critical value of $\alpha = p/N$ below which the order parameter is unity, and above which it is zero. The derivation for this diagram is in the limit of $N \to \infty$. In the experiment we have $p = 5$ and $N = 200$ giving an $\alpha = 0.025$ which seems to be below the critical α_c. However, we can suspect that a finite N will mean that the distinct border in the phase diagram actually is a smoother transition between $m = 1$ and $m = 0$. Due to this, the given value of α might cause a small drop in the order parameter, even though it is below the critical α_c. Running an experiment with the same ratio between p and N, but with each of them 40 times larger, so as to have N *closer* to ∞, the steady state value does not change significantly. Something that does change however is that all the trials have a much more uniform curve towards steady state, suggesting that the seemingly stochastic behaviour in the first 200 iterations in Figure 2 is due to the finite N.

Nevertheless, the order parameter for the fed pattern, seen in the figure, is close enough to unity that the network can be considered to perform well. To further strengthen this point, the order parameters for some of the other patterns were calculated when feeding pattern 1. As expected for a functioning network, these order parameters converged close to zero.

A final remark is that the steady state is reached very consistently for all 20 trials in the range 400 - 800 iterations. There is not much difference between the individual trials steady state values, indicating that this system for these specific values of p, N and β is quite stable.

2b Increasing the number of patterns to $p = 40$ changes the outcome of the experiment quite a bit, as can be seen in Figure 3. A well performing system should, as stated in **2a**, have an order parameter close to unity for the fed pattern. This clearly does not happen in the experiments. The different experiments converge to different values and at vastly different convergence times, ranging from around 2,000 iterations to not having converged at 20,000 iterations. Most definitely, this is due to the increase in p making $\alpha = p/N = 0.2$ be in the vicinity of or even larger than α_c. According to the phase diagram on p. 60 this should result in an order parameter of zero, but again, that is in the limit $N \to \infty$. In this case it might cause the order parameter to drop significantly, but not quite like the steep slope in the phase diagram suggests. We can not know if we have really passed α_c or not, but the network certainly is not working well. The different steady state values of the different trials might represent local minima (of the energy function) that the system gets stuck in, but no clear conclusion can be drawn about that.

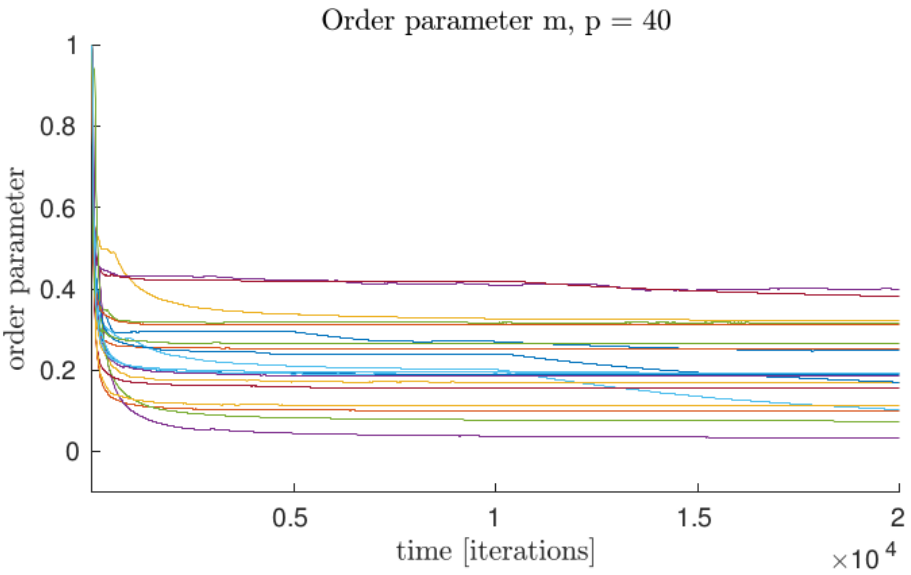

Figure 3: Wandering mean of order parameter for pattern 1 which was fed into the network with $\alpha = 0.2$. The different colors each represent one of the 20 experiments done.

Similarly to the the case in **2a**, increasing both p and N by a factor 40 makes the iterations more aligned with each other, with steady state values in the range of 0.1 and 0.2, instead of 0.1 and 0.5 seen in Figure 3. This indicates, as in **2a**, that the stochastic behaviour is an effect of a finite N.

3 Back Propagation

3a A network without any hidden layers consisting of two input nodes, (ξ_1, ξ_2), and one output node (O_1), was implemented with parameters, weights and thresholds assigned according to the assignment description. The energy function was defined according to common practise as $H = \frac{1}{2}\sum_{\mu=1}^{p}(\zeta_1^\mu - O_1^\mu)^2$, with p being the number of patterns in the data set, and ζ_1^μ being the targets of each corresponding pattern $\underline{\xi}$. The back propagation formulae for updating weights w_{1j} and threshold θ_1, were then explicitly expressed as $x \to x + \delta x$ with

$$\delta w_{1j} = -\eta\frac{\partial H}{\partial w_{1j}} = \eta\beta(1 - \tanh(\beta(w_{1j}\xi_j - \theta_1))^2)\sum_{\mu=1}^{p}(\zeta_1^\mu - O_1^\mu)\xi_j^\mu$$

$$\delta\theta_1 = -\eta\frac{\partial H}{\partial \theta_1} = -\eta\beta(1 - \tanh(\beta(w_{1j}\xi_j - \theta_1))^2)\sum_{\mu=1}^{p}(\zeta_1^\mu - O_1^\mu).$$

The network was trained with the given training data, and the trained network was then used on the validation set. A plot with the average energy values per pattern for both the training set and the validation set, along with a plot of the obtained classification boundary can be found in Figure 4. The obtained average energy for both data sets fluctuates around 0.5, and from inspecting the obtained classification boundary, we can conclude that this network architecture performs poorly on the given data. This is strengthened by the computed mean classification error (0.462 for training set, 0.456 for validation set) and minimum errors (0.424 for training set, 0.432 for validation set). The computed variances of the errors were $1.1 \cdot 10^{-3}$ for the training set, and $5.3 \cdot 10^{-4}$ for the validation set.

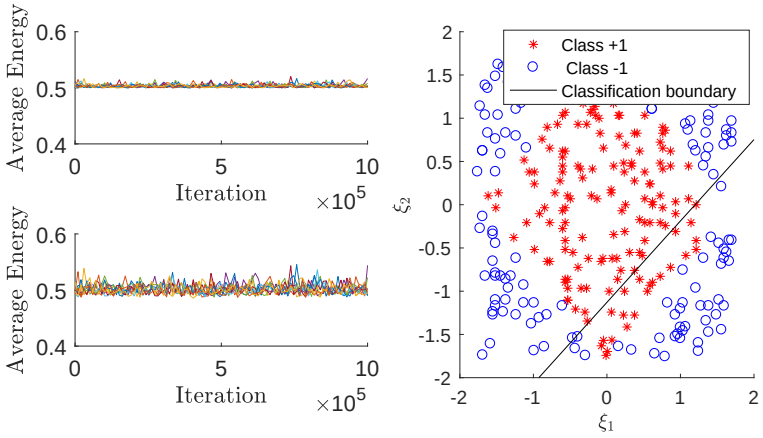

Figure 4: Computed average energy plots for training set (top) and validation set (bottom) in the left panel. The right plot displays the points of both classes in the plane, along with one classification boundary obtained from a sample run of the network.

The geometric interpretation of this problem is that the network weight vector corresponds to a line in the sample space. This line should make an accurate separation of the data into the different classes if the network is working well. The plot to the right in Figure 4 tells us that one line certainly is not enough to create borders between the two classes, which explains why the network fails.

3b A network with one hidden layer consisting of 4 hidden nodes was implemented with the same parameter settings as in the previous network, though because of the hidden layer, the expressions for back propagation formulae becomes (using the notation from the lecture notes [1])

$$\delta W_{ij} = -\eta \frac{\partial H}{\partial W_{ij}} = \eta \sum_{\mu=1}^{p} \delta_i^\mu V_j^\mu$$

$$\delta \Theta_i = -\eta \frac{\partial H}{\partial \Theta_i} = -\eta \sum_{\mu=1}^{p} \delta_i^\mu$$

$$\delta w_{jk} = -\eta \frac{\partial H}{\partial w_{jk}} = \eta \sum_{\mu=1}^{p} \delta_j^\mu \xi_k^\mu$$

$$\delta \theta_j = -\eta \frac{\partial H}{\partial \theta_i} = -\eta \sum_{\mu=1}^{p} \delta_j^\mu$$

$$\text{with } \delta_i^\mu = \beta(\zeta_i^\mu - O_i^\mu)(1 - \tanh^2(\beta b_i^\mu)) \quad \text{and} \quad \delta_j^\mu = \beta \sum_i \delta_i^\mu W_{ij}(1 - \tanh^2(\beta b_j^\mu)),$$

with b_i^μ and b_j^μ as in lecture notes p. 98 and only one output, $i = 1$. Instead of taking the sum over μ in these formulae (= *batch mode*) we randomly select only one pattern for the update.

This network was trained and validated using the given data sets in similar fashion as the previous assignment. The plotted average energy per pattern can be viewed in Figure 5. In comparison to **3a**, the average energy reaches far lower values at the steady state, fluctuating around 0.05 for the training set, and around 0.1 for the validation set, indicating that this network performs well on the given data.

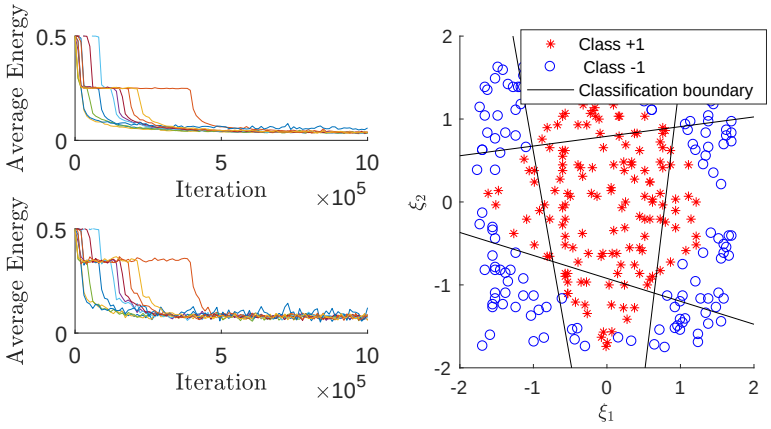

Figure 5: Computed average energy plots for training set (top) and validation set (bottom) in the left panel. The right plot displays the points of both classes in the plane, along with one classification boundary obtained from a sample run of the network.

With one hidden layer with 4 neurons, the weight vectors now correspond to 4 lines in our geometric interpretation of the problem. As can been seen in the right plot in Figure 5, four lines do a much better job of separating the two classes which explains why this network works so much better. Again, this is strengthened by the computed mean classification error (0.0203 for training set, 0.0527 for validation set) and minimum errors (0.0169 for training set, 0.0431 for validation set). The computed variances of the errors were 2.327×10^{-5} for the training set, and 2.574×10^{-5} for the validation set. Since the network was trained on and adapted to the training set it is natural that it will perform slightly worse on the validation set, as seen in the left plot of Figure 5.

References

[1] Bernhard Mehlig. Lecture notes from FFR135, Artificial Neural Networks, September 2017.

A Matlab Code

Task 1

```
%% 1
%1 b)
clf;
p = [1, (20:20:400)]';
N = 200;
num_bits = 100000;

p_err = zeros(length(p),1);

for i=1:length(p)
    tic
    p_err(i) = OSEP(p(i), N, num_bits);
    toc
end

plin = linspace(1,400); % for plotting resolution
erf_vec = 0.5*(1 - erf( (plin+N)./(sqrt(2*plin*N)) ) );

hold on
plot(p/N,p_err,'*')
plot(plin/N, erf_vec, '-')
xlabel('\alpha = p/N')
ylabel('P_{err}')
legend('Experiment data','Theoretical model')
```

```
function p_err = OSEP(p, N , num_bits)

iterations = ceil(num_bits/(p*N));
error_vector = zeros(p*N,iterations);

for i = 1:iterations

    patterns = GeneratePatterns(p, N);
    weights = zeros(N,N);
    for k=1:p
        weights = weights + 1/N*patterns(:,k) * patterns(:,k)';
    end
    for j = 1:p
       state = patterns(:,j);
       new_state = Hopfield(weights, state);
       error_vector(((j-1)*N+1):j*N,i) = new_state~=state;
    end
    %disp(size(error_vector));
end
error_vector = error_vector(:);

index = randperm(length(error_vector));
p_err = sum(error_vector(index(1:num_bits)))/num_bits;

end
```

```
function patterns = GeneratePatterns( p ,N )

patterns = randi([0,1], N ,p); % Generate random patterns
patterns = 2*patterns-1;

end
```

```
function new_state = Hopfield( weights, states )

[R,W] = size(states);
new_state = sign(weights*states);
new_state = new_state + (new_state==0) .* (2*randi([0,1],R,W)-1);

end
```

```
function result = Sigmoid( b, beta )
    result = 1 ./ (1 + exp(-b.*beta));
end
```

Task 2

```
% Stochastic Hopfield
% 2a)
clf; clear all;
N = 200;
p = 5;
beta = 2;
iterations = 20;
tmax = 400;
m = zeros(tmax, iterations);
ts = 10;
Tmax = floor(tmax/ts);
WM = zeros(Tmax,iterations);

for iteration = 1:iterations

patterns = GeneratePatterns(p, N);

weights = zeros(N,N);
for k=1:p
    weights = weights + 1/N*patterns(:,k) * patterns(:,k)';
end
for i=1:N
    weights(i,i)=0;
end

states = zeros(N,tmax);
state_0 = patterns(:,1);
states(:,1)= state_0;

m(1,iteration) = 1/N * (state_0'*state_0);

for t=1:tmax-1
   states(:,t+1) = StochasticUpdate(beta, states(:,t), weights);
   m(t+1,iteration) = 1/N * states(:,t+1)'*state_0;
end

WM(:,iteration) = WanderingMean(m(:,iteration), ts);
end
WM = [m(1,:) ; WM];


plot(ts*(0:Tmax),WM)
title('Order parameter m, p = 5', 'Interpreter', 'LaTeX')
xlabel('time [iterations]', 'Interpreter', 'LaTeX')
ylabel('order parameter', 'Interpreter', 'LaTeX')
```

```
46 set(gca,'fontsize', 14)
47 axis([1 tmax 0.9 1])
48 box off;
49
50 %% 2b)
51 p = 40;
52 tmax = 20000;
53 Tmax = floor(tmax/ts);
54 m = zeros(tmax, iterations);
55 WM = zeros(Tmax,iterations);
56
57 for iteration = 1:iterations
58
59 patterns = GeneratePatterns(p, N);
60
61 weights = zeros(N,N);
62 for k=1:p
63     weights = weights + 1/N*patterns(:,k) * patterns(:,k)';
64 end
65 for i=1:N
66     weights(i,i)=0;
67 end
68
69 state_0 = patterns(:,1);
70 states = zeros(N,tmax);
71 states(:,1)= state_0;
72
73 m(1,iteration) = 1/N * (state_0'*state_0);
74
75 for t=1:tmax-1
76    states(:,t+1) = StochasticUpdate(beta, states(:,t), weights);
77    m(t+1,iteration) = 1/N * states(:,t+1)'*state_0;
78 end
79
80 WM(:,iteration) = WanderingMean(m(:,iteration), ts);
81 end
82 WM = [m(1,:) ; WM];
83
84 plot(ts*(0:Tmax),WM)
85 title('Order parameter m, p = 40', 'Interpreter', 'LaTeX')
86 xlabel('time [iterations]', 'Interpreter', 'LaTeX')
87 ylabel('order parameter', 'Interpreter', 'LaTeX')
88 set(gca,'fontsize', 14)
89 axis([1 tmax -0.1 1])
90 box off;
```

```
function update = StochasticUpdate( beta, state, weights )

b = Hopfield(weights, state);
g = Sigmoid(b,2*beta);
update = 2*(rand(length(state),1)<g)-1;

end
```

```
function [ WM ] = WanderingMean( m, ts )
tmax = length(m);
iterations = floor(tmax/ts);
WM = zeros(iterations,1);

for i=1:iterations
    T = i*ts;
    WM(i) = mean(m(1:T));
end

end
```

Task 3

```
train_data = load('train_data_2017.txt');
val_data = load('valid_data_2017.txt');

for i = 1:2
    train_data(:,i) = (train_data(:,i)-mean(train_data(:,i)))/std(
        train_data(:,i));
    val_data(:,i) = (val_data(:,i)-mean(val_data(:,i)))/std(val_data(:,
        i));

end

train_pat = train_data(:,1:2);
train_ans = train_data(:,3);
val_pat = val_data(:,1:2);
val_ans = val_data(:,3);

lr = 0.02;
beta = 1/2;

class_1 = train_pat(train_ans == 1, :);
class_2 = train_pat(train_ans == -1, :);

%%
hold on
axis equal
plot(class_1(:,1),class_1(:,2), 'r*')
plot(class_2(:,1),class_2(:,2), 'b*')

%% 3a

iterations = 1e6;
w = rand(2,1)*0.4-0.2; % weights
bias = rand(1,1)*2-1; % biases
energy_train = zeros(iterations/1000,1);
energy_val = zeros(iterations/1000,1);
train_end = 1000;
c_err_t = zeros(train_end, 1);
c_err_v = zeros(train_end, 1);
ind_count = 0;

for iter = 1:iterations

% 1: pick random pattern
pat_ind = randperm( length( train_data ) , 1);
xi = train_pat( pat_ind , :)';
```

```
44 zeta = train_ans(pat_ind);
45
46 % 2: feed forward values
47 b = w'*xi - bias;
48 output = tanh(beta*b);
49
50 %%%%%%%%%%%%%%%%INTERLUDE%%%%%%%%%%%%%%%%%%%
51 % Calculating energy and classification errors
52 if mod(iter,1000) == 0
53     for i = 1:length(train_data)
54         out_temp = tanh( beta * (w'*train_pat(i,:)' - bias ));
55         energy_train(iter/1000) = energy_train(iter/1000) + 0.5*(
               train_ans(i) - out_temp)^2;
56     end
57
58     for i = 1:length(val_data)
59         out_val_temp = tanh( beta * (w'*val_pat(i,:)' - bias ));
60         energy_val(iter/1000) = energy_val(iter/1000) + 0.5*(val_ans(i
               ) - out_val_temp)^2;
61     end
62 end
63
64 if iterations - iter < train_end
65     ind_count = ind_count + 1;
66     c_err_t_temp = zeros(train_end,length(train_ans));
67     c_err_v_temp = zeros(train_end,length(val_ans));
68
69     for i = 1:length(train_data)
70         c_err_t_temp(ind_count,i) = tanh( beta * (w'*train_pat(i,:)' -
               bias ));
71     end
72
73     for i = 1:length(val_data)
74         c_err_v_temp(ind_count,i) = tanh( beta * (w'*val_pat(i,:)' -
               bias ));
75     end
76
77     c_err_t(ind_count) = 1/(2*length(train_ans))*sum(abs(train_ans -
           sign(c_err_t_temp(ind_count,:))'));
78     c_err_v(ind_count) = 1/(2*length(val_ans))*sum(abs(val_ans - sign(
           c_err_v_temp(ind_count,:))'));
79 end
80 %%%%%%%%%%%%%%%%INTERLUDE%%%%%%%%%%%%%%%%%%%
81
82 % 3: update the weights
83 w = w + lr*beta*(1 - tanh(beta*b)^2)*(zeta-output).*xi;
84 bias = bias - lr*beta*(1 - tanh(beta*b)^2)*(zeta-output);
```

```
85
86 % 4: and do it all again
87 end
88
89 %%
90 clf;
91 subplot(1,2,1)
92 hold on
93 axis([0 iterations 0.4 0.6])
94 iter_vec = linspace(1,iterations,iterations/(10*1000));
95 plot(iter_vec,energy_train(1:10:end)/length(train_ans), 'r')
96 plot(iter_vec,energy_val(1:10:end)/length(val_ans), 'b')
97 legend('Training set','Validation set')
98 set(gca,'fontsize', 8)
99 xlabel('Iteration', 'Interpreter', 'LaTex')
100 ylabel('Average Energy', 'Interpreter', 'LaTex')
101 set(gca,'fontsize', 14)
102
103 x_vec = linspace(-2,2, 100);
104 line1 = bias(1)/w(2) - w(1)/w(2).*x_vec;
105
106
107 subplot(1,2,2)
108 hold on
109 axis([-2 2 -2 2])
110 plot(class_1(:,1), class_1(:,2), 'r*')
111 plot(class_2(:,1), class_2(:,2), 'bo')
112 plot(x_vec,line1, 'k')
113 legend('Class +1',' Class -1', 'Classification boundary')
114 set(gca,'fontsize', 8)
115 xlabel('$\xi_1$', 'Interpreter', 'LaTex')
116 ylabel('$\xi_2$', 'Interpreter', 'LaTex')
117 set(gca,'fontsize', 12)
118
119
120 %% 3b)
121
122 % weights
123 w_in = rand(4,2)*0.4-0.2;
124 w_out = rand(4,1)*0.4-0.2;
125 % biases
126 bias_in = rand(4,1)*2-1;
127 bias_out = rand(1,1)*2-1;
128
129 iterations = 1e6;
130 energy_train = zeros(iterations/1000,1);
131 energy_val = zeros(iterations/1000,1);
```

```
132 train_end = 1000;
133 c_err_t = zeros(train_end, 1);
134 c_err_v = zeros(train_end, 1);
135 ind_count = 0;
136
137 for iter = 1:iterations
138 % 1: Pick a random pattern
139 pat_ind = randperm( length( train_data ) , 1);
140 xi = train_pat( pat_ind , :)';
141 zeta = train_ans(pat_ind);
142                                                 % k = 1..2, j = 1..4, i =
                                                        1
143 % 2: Feed forward values
144 b_V = w_in*xi - bias_in;
145 V = tanh(beta*b_V);
146
147 b_out = w_out'*V - bias_out;
148 O = tanh(beta*b_out);
149
150 %%%%%%%%%%%%%%%%INTERLUDE%%%%%%%%%%%%%%%%%%%%
151 if mod(iter,1000) == 0
152     for i = 1:length(train_data)
153         xi_temp = train_pat(i,:)';
154         V_temp = tanh(beta*(w_in*xi_temp - bias_in ));
155         b_out_temp = w_out'*V_temp - bias_out;
156         O_temp = tanh(beta*b_out_temp);
157         energy_train(iter/1000) = energy_train(iter/1000) + 0.5*(
                train_ans(i) - O_temp)^2;
158     end
159
160     for i = 1:length(val_data)
161         xi_temp = val_pat(i,:)';
162         V_temp = tanh(beta*(w_in*xi_temp - bias_in ));
163         b_out_temp = w_out'*V_temp - bias_out;
164         O_temp = tanh(beta*b_out_temp);
165
166         energy_val(iter/1000) = energy_val(iter/1000) + 0.5*(val_ans(i
                ) - O_temp)^2;
167     end
168 end
169
170 if iterations - iter < train_end
171     ind_count = ind_count + 1;
172     c_err_t_temp = zeros(train_end,length(train_ans));
173     c_err_v_temp = zeros(train_end,length(val_ans));
174
175     for i = 1:length(train_data)
```

```
176         xi_temp = train_pat(i,:)';
177         V_temp = tanh(beta*(w_in*xi_temp - bias_in ));
178         b_out_temp = w_out'*V_temp - bias_out;
179         O_temp = tanh(beta*b_out_temp);
180         c_err_t_temp(ind_count,i) = O_temp;
181     end
182
183     for i = 1:length(val_data)
184         xi_temp = val_pat(i,:)';
185         V_temp = tanh(beta*(w_in*xi_temp - bias_in ));
186         b_out_temp = w_out'*V_temp - bias_out;
187         O_temp = tanh(beta*b_out_temp);
188         c_err_v_temp(ind_count,i) = O_temp;
189     end
190
191     c_err_t(ind_count) = 1/(2*length(train_ans))*sum(abs(train_ans -
        sign(c_err_t_temp(ind_count,:))'));
192     c_err_v(ind_count) = 1/(2*length(val_ans))*sum(abs(val_ans - sign(
        c_err_v_temp(ind_count,:))'));
193 end
194 %%%%%%%%%%%%%%%%INTERLUDE%%%%%%%%%%%%%%%%%%%
195
196 % 3: Update the weights (and bias)
197 delta_i = beta*(zeta - O)*(1 - O.^2);
198 delta_j = beta*delta_i * w_out .* (1 - V.^2);
199
200 dw_out = lr * delta_i * V;
201 dw_in = lr * delta_j .* xi';
202 dbias_out = -lr * delta_i;
203 dbias_in = -lr * delta_j;
204
205 w_in = w_in + dw_in;
206 w_out = w_out + dw_out;
207 bias_in = bias_in + dbias_in;
208 bias_out = bias_out + dbias_out;
209
210 % 4: And do it all again
211 end
212
213 %%
214 clf;
215 subplot(1,2,1)
216 hold on
217 axis([0 iterations 0 0.5])
218 iter_vec = linspace(1,iterations,iterations/(10*1000));
219 plot(iter_vec,energy_train(1:10:end)/length(train_ans), 'r')
220 plot(iter_vec,energy_val(1:10:end)/length(val_ans), 'b')
```

```
221 legend('Training set','Validation set')
222 set(gca,'fontsize', 8)
223 xlabel('Iteration', 'Interpreter', 'LaTex')
224 ylabel('Average Energy', 'Interpreter', 'LaTex')
225 set(gca,'fontsize', 14)
226
227 x_vec = linspace(-2,2, 100);
228 line1 = bias_in(1)/w_in(1,2) - w_in(1,1)/w_in(1,2).*x_vec;
229 line2 = bias_in(2)/w_in(2,2) - w_in(2,1)/w_in(2,2).*x_vec;
230 line3 = bias_in(3)/w_in(3,2) - w_in(3,1)/w_in(3,2).*x_vec;
231 line4 = bias_in(4)/w_in(4,2) - w_in(4,1)/w_in(4,2).*x_vec;
232
233 subplot(1,2,2)
234 hold on
235 axis([-2 2 -2 2])
236 plot(class_1(:,1), class_1(:,2), 'r*')
237 plot(class_2(:,1), class_2(:,2), 'bo')
238 plot(x_vec, line1, 'k')
239 plot(x_vec, line2, 'k')
240 plot(x_vec, line3, 'k')
241 plot(x_vec, line4, 'k')
242 legend('Class +1',' Class -1', 'Classification boundary')
243 set(gca,'fontsize', 8)
244 xlabel('$\xi_1$', 'Interpreter', 'LaTex')
245 ylabel('$\xi_2$', 'Interpreter', 'LaTex')
246 set(gca,'fontsize', 12)
```