Homework 1: One-step error probability

Problem description:

Write a computer program implementing asynchronous deterministic dynamics [Eq. (1.9) in the course book] for a Hopfield network. Use Hebb's rule with $w_{ii} = 0$ [Eq. (2.26) in the course book]. Generate and store $p = [12, 24, 48, 70, 100, 120]$ random patterns with $N = 120$ bits. Each bit is either +1 or -1 with probability $\frac{1}{2}$.
For each value of p, estimate the one-step error probability $P_{\text{error}}^{t=1}$ based on 10^5 independent trials. Here, one trial means that you generate and store a set of p random patterns, feed one of them, and perform one asynchronous update of a single randomly chosen neuron. If in some trials you encounter $\text{sgn}(0)$, simply set $\text{sgn}(0) = 1$.
List below the values of $P_{\text{error}}^{t=1}$ that you obtained in the following form: $[p_1, p_2, \ldots, p_6]$, where p_n is the value of $P_{\text{error}}^{t=1}$ for the n-th value of p from the list above. Give four decimal places for each p_n.

Hebb's rule

$$w_{ij} = \frac{1}{N} \sum_{\mu=1}^{p} x_i^{(\mu)} x_j^{(\mu)} \quad \text{for} \quad i \neq j, \quad w_{ii} = 0, \quad \text{and} \quad \theta_i = 0. \qquad (2.26)$$

Asynchronous deterministic updates

$$s_i(t+1) = \begin{cases} g\left(\sum_j w_{mj} s_j(t) - \theta_m\right) & \text{for } i = m, \\ s_i(t) & \text{otherwise.} \end{cases} \qquad (1.9)$$

Initial conditions:

```
clear
p = [12 24 48 70 100 120]; % 24 48 70 100 120 amount of stored patterns
N = 120; % amount of neurons
nTrials = 10e5; % numver of trials
```

Q1: Hebb's rule setting the diagonal weights to zero

```
% loop through all given numbers
diagonal_zero = true;
P_error_list = [];
for i = 1:length(p)
    P_error_list(end+1) = P_err_cal(p(i), N, nTrials, diagonal_zero);
end
```

```
disp("P_error with diagonal elements are 0: ")
```

```
P_error with diagonal elements are 0:
```

```
out=['[' sprintf(' %.4f, ', P_error_list(1:end-1) ) num2str(P_error_list(end)) ']'];
```

```
disp(out)
```

```
[ 0.0005,  0.0114,  0.0560,  0.0947,  0.1363, 0.15848]
```

Q2: Hebb's rule without setting the diagonal weights to zero

```
diagonal_zero = false;
P_error_list_dz = [];
for i = 1:length(p)
    P_error_list_dz(end+1) = P_err_cal(p(i), N, nTrials, diagonal_zero);
end
```

```
disp("P_error with diagonal elements not set to 0: ")
```

```
P_error with diagonal elements not set to 0:
```

```
out=['[' sprintf(' %.4f, ', P_error_list_dz(1:end-1) ) num2str(P_error_list_dz(end)) ']'];
disp(out)
```

```
[ 0.0001,  0.0031,  0.0124,  0.0179,  0.0222, 0.022872]
```

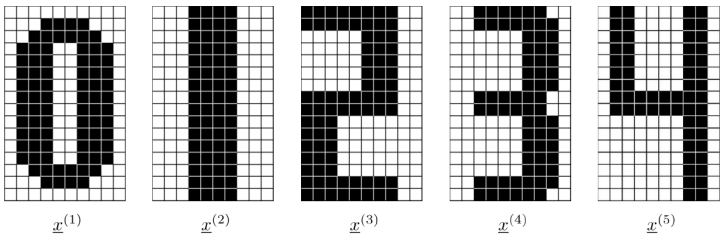
One-step error calculation function

```
function P_error = P_err_cal(p, N, nTrials, diagonal_zero)
    error = 0;
    for trial = 1:nTrials
        % Generate random patterns 120*p
        patterns_generated = 2 * randi([0, 1], N, p) - 1;
        % Initialize weight matrix NxN with Hebb's rule
        W = zeros(N);
        for i = 1:p
            W = W + patterns_generated(:,i)*patterns_generated(:,i)'/N;
        end

        % set diagonal elements to 0
        if diagonal_zero == true
            for i = 1:N
                W(i,i) = 0;
            end
        end

        % choose a random pattern in the stored patterns 120*1
        pr = patterns_generated(:,randi(p));
        % choose a bit
        nr = randi(N);

        b_nr = W(nr,:)*pr;
        if b_nr == 0
            x_nr = 1;
        else
```

```
            x_nr = sign(b_nr);
        end

        % check if the updated bit equals the origin bit
        if x_nr ~= pr(nr)
            error = error + 1;
        end
    end
    P_error = error/nTrials;
end
```

Homework 1: Recognising digits

Five patterns



The Figure shows five patterns $\underline{x}^{(1)}, \underline{x}^{(2)}, \underline{x}^{(3)}, \underline{x}^{(4)}, \underline{x}^{(5)}$ representing the digits 0 to 4. The patterns are binary, white bits correspond to -1, black bits to 1.

Represent these patterns as vectors to store them in a Hopfield network using Hebb's rule with $w_{ii} = 0$, Eq. (2.26) in the course book. Use a typewriter scheme to index the bits, starting from the top row and going from left to right in each row.

Below you are given three different patterns to feed into your network. For each of these patterns, iterate asynchronous determistic updates of the network using the typewriter scheme. If, at some update, you encounter $\mathrm{sgn}(0)$, then simply set $\mathrm{sgn}(0) = 1$. Stop iterating when you reach a steady state, and answer the following questions:

(A) To which pattern does your network converge?

(B) Classify this pattern using the following scheme: if the pattern you obtain correspods to any of the stored patterns $\underline{x}^{(\mu)}$, enter the pattern index μ (for example, you enter 1 if you retrieve digit 0). If your network retrieves an inverted stored pattern, then enter $-\mu$ (for example, you enter -1 if you get the inverted digit 0). If you get anything else, enter 6.

Read in the patterns to store and classify

```
clear
load("patterns.mat")
```

Initial conditions

```
p = size(digit_patterns,2)
```

```
p = 5
```

```
N = size(digit_patterns,1)
```

```
N = 160
```

Initalize Hopfield network

```
W = zeros(N);
```

```
for mu = 1:p
    W = W + digit_patterns(:,mu)*digit_patterns(:,mu)';
end
W = W./N;
% set diagonal elements to 0
for i = 1:N
    W(i,i) = 0;
end
W
```

```
W = 160×160
         0    0.0187    0.0063    0.0063    0.0063    0.0063    0.0063    0.0063 ···
    0.0187         0    0.0187   -0.0063   -0.0063   -0.0063   -0.0063    0.0187
    0.0063    0.0187         0    0.0063    0.0063    0.0063    0.0063    0.0312
    0.0063   -0.0063    0.0063         0    0.0312    0.0312    0.0312    0.0063
    0.0063   -0.0063    0.0063    0.0312         0    0.0312    0.0312    0.0063
    0.0063   -0.0063    0.0063    0.0312    0.0312         0    0.0312    0.0063
    0.0063   -0.0063    0.0063    0.0312    0.0312    0.0312         0    0.0063
    0.0063    0.0187    0.0312    0.0063    0.0063    0.0063    0.0063         0
    0.0063    0.0187    0.0063   -0.0187   -0.0187   -0.0187   -0.0187    0.0063
    0.0187    0.0063   -0.0063   -0.0063   -0.0063   -0.0063   -0.0063   -0.0063
     ⋮
```

```
steady_patterns = [];
steady_pattern_strs = [];
classify_indexs = [];
for question_index = 1:size(test_patterns,2)
    test_pattern = test_patterns(:,question_index);
    steady_patterns = [steady_patterns update_iterate(N, W, test_pattern)];
    steady_pattern_strs = [steady_pattern_strs pattern_to_str(steady_patterns(:,question_index)
    classify_indexs = [classify_indexs classify_index(p, digit_patterns, steady_patterns(:,ques
end

for question_index = 1:size(test_patterns,2)
    Q = fprintf("Question: %d", question_index);
    disp("The input pattern: ")
    show_digit(test_patterns(:,question_index));
    disp("The converged pattern is: ")
    show_digit(steady_patterns(:,question_index))
    fprintf(steady_pattern_strs(question_index))
    disp("The pattern index: ")
    disp(classify_indexs(question_index))

end
```

```
Question: 1
The input pattern:
```

The converged pattern is:

```
[[-1, -1, -1, -1, -1, -1, -1, -1, 1, 1],[-1, -1, -1, -1, -1, -1, -1, -1, 1, 1],[1, 1, 1, 1, 1, -1, -1, -1, 1, 1],[1,
The pattern index:
    -3
Question: 2
The input pattern:
```

The converged pattern is:

```
[[-1, -1, -1, -1, -1, -1, -1, -1, -1, -1],[-1, -1, -1, 1, 1, 1, 1, -1, -1, -1],[-1, -1, 1, 1, 1, 1, 1, 1, -1, -1],[-
The pattern index:
     1
Question: 3
The input pattern:
```

The converged pattern is:

```
[[1, 1, -1, -1, -1, -1, -1, -1, 1, 1],[1, 1, -1, -1, -1, -1, -1, -1, -1, 1],[1, 1, 1, 1, 1, 1, -1, -1, -1, 1],[1, 1,
The pattern index:
    -4
```

```
function steady_pattern = update_iterate(neuron_n, weight_matrix, pattern_fed)
    s = pattern_fed;
    s_old = zeros(neuron_n,1);
    while ~isequal(s,s_old)
        s_old = s;
        for i = 1:neuron_n
            b = weight_matrix(i,:)*s;
            if b == 0
                s(i) = 1;
            else
                s(i) = sign(b);
            end
        end
    end
    steady_pattern = s;
end

function classify_idx = classify_index(patterns_n, patterns, pattern_fed)
    digit = 6;
    for index = 1:patterns_n
        if pattern_fed == patterns(:,index)
            digit = index;
        elseif pattern_fed == -patterns(:,index)
            digit = -index;
        end
    end
    classify_idx = digit;
end

function pattern_str = pattern_to_str(pattern)
    pattern = pattern';
    pattern = round(reshape(pattern,[10,16])');
    char="";
    for i = 2:16
        out=['[' sprintf('%d, ', pattern(i,1:end-1)) num2str(pattern(i,end)) ']'];
        char = char + "," + out;
    end
    out=['[' sprintf('%d, ', pattern(1,1:end-1)) num2str(pattern(1,end)) ']'];
    pattern_str = "[" + out + char + "]";
end
function show_digit(state_in_vector)
    figure
    states_reshaped = -reshape(state_in_vector, 10, 16)';
    imshow(states_reshaped)
end
```

Homework1: Boolean functions

A perceptron with a single output neuron can be used to determine whether an *n*-dimensional Boolean function is linearly separable. In this task, you implement a computer program that determines whether a Boolean function is linearly separable or not, using a perceptron with activation function $g(b) = \mathrm{sgn}(b)$, where $b = \sum_{j=1}^{n} w_j x_j - \theta$ [See Eq. (5.9) in the course book for the case *n*=2]. Using this program, sample an *n*-dimensional Boolean function randomly and determine whether it is linearly separable. Do this 10^4 times for *n*=2,3,4 and 5 dimensions, and save the number of linearly separable Boolean functions found. Be sure to not count the same Boolean function twice. This can, for example, be done by adding every sampled Boolean function to a list and excluding the function if it comes up a second time.

The learning rules for the weights w_j and threshold θ are

$$\delta w_j^{(\mu)} = \eta(t^{(\mu)} - O^{(\mu)})x_j^{(\mu)}$$

and

$$\delta\theta^{(\mu)} = -\eta(t^{(\mu)} - O^{(\mu)})$$

See also Eq. (5.18) in the course book. For each Boolean function, train the perceptron for 20 training epochs (one epoch amounts to updating the parameters once for every input-output pair) using a learning rate η=0.05. Initialize the weights from a normal distribution with mean zero and variance 1/*n*, and initialize the thresholds to zero.

Task description

for trail = 1:10^4

- Sample Boolean

- Train percepton

- Check accuracy

- if 100% correct, counter+=1

- else, pass

shape of weights

$$w = \begin{bmatrix} w_1 \\ w_2 \\ \dots \\ w_n \end{bmatrix}$$

number of Boolean functions

$$2^{2^N}$$

number of outputs

2^N

```
clc
clear
n = 2; % N binary inputs 3,4,5
nTrials = 10e4;     % number of trails
nEpoches = 20;      % number of epoches
eta = 0.05;         % learning rate
counter = 0;        % number of linearly separable Boolean found
```

Setup boolean inputs and used bool list

```
boolean_inputs_2 = ff2n(2)';
boolean_inputs_3 = ff2n(3)';
boolean_inputs_4 = ff2n(4)';
boolean_inputs_5 = ff2n(5)';
```

```
n = 2;
boolean_inputs = boolean_inputs_2;
separable_counter_2D = learning_iter(n, nTrials, nEpoches, eta, boolean_inputs);
disp(separable_counter_2D)
```

```
    14
```

```
n = 3;
boolean_inputs = boolean_inputs_3;
separable_counter_3D = learning_iter(n, nTrials, nEpoches, eta, boolean_inputs);
disp(separable_counter_3D)
```

```
   104
```

```
n = 4;
boolean_inputs = boolean_inputs_4;
separable_counter_4D = learning_iter(n, nTrials, nEpoches, eta, boolean_inputs);
disp(separable_counter_4D)
```

```
        1304
```

```
n = 5;
boolean_inputs = boolean_inputs_5;
separable_counter_5D = learning_iter(n, nTrials, nEpoches, eta, boolean_inputs);
disp(separable_counter_5D)
```

```
     2
```

```
function separable_counter = learning_iter(n, nTrials, nEpoches, eta, boolean_inputs)
```

```
    used_bool=[];
    separable_counter = 0;
    for trail = 1:nTrials
        boolean_outputs = 2 * randi([0, 1], 2^n, 1) - 1;
        if isempty(used_bool) || ~ismember(boolean_outputs',used_bool','rows')
            w = normrnd(0, 1/n, [n,1]);       % initialize weight with normal distribution
            theta = 0;                        % initialize theta with 0
            for epoch = 1:nEpoches
                total_error = 0;
                for mu = 1:2^n                % 1 epoch is updating the parameters once for every
                    Out = sign(dot(w,boolean_inputs(:,mu))-theta);
                    if Out == 0
                        Out = 1;
                    end
                    err = boolean_outputs(mu) - Out;
                    % delta_w, delta_theta calculation
                    delta_w = cal_delta_w(n, eta, err, boolean_inputs(:,mu));
                    delta_theta = -eta * err;
                    % update w and theta
                    w = w + delta_w;
                    theta = theta + delta_theta;

                    total_error = total_error + abs(err);
                end
                if total_error == 0
                    separable_counter = separable_counter + 1;   % find one linear sepration fu
                    break
                end
            end
            used_bool(:,end+1) = boolean_outputs;
        end
    end
end

function delta_w = cal_delta_w(n, eta, err, input)
    delta_w = zeros([n,1]);
    for j=1:n
        delta_w(j) = eta * err * input(j);
    end
end
```

Homework 1 Boolean functions

To find all possible linearly separable cases for n-dimensional Boolean functions, one has to do sampling in the possible combinations of binary digits, especially in high-dimension cases.

When the trial starts, create a random Boolean output, a $2^n \times 1$ vector, containing only -1 or 1. Using this output with the learning rule during each epoch to iteratively update the weight and threshold. The output that has been generated before should be skipped so that the same Boolean function will not be counted twice. Also, a total error variable accumulates the differences between generated output and the one from McCulloch-Pitts dynamics. For the weight and threshold that has zero total error in one epoch, they can separate the n-dimension points and thus one linearly separable Boolean function is found.

n	Boolean functions 2^{2^n}	Linearly separable Boolean functions	My result
2	16	14	14
3	256	104	104
4	65536	1882	1304
5	4294967296	94572	2

Table 1: Result from Matlab experiment

For the n-dimensional Boolean function, the total number of possible combinations is 2^{2^n}. This gives the data in the second column.

The Matlab experiment always gives the correct result when the dimension $n = 2, 3$. In these two circumstances, the number of trials, 10^4, is much larger than the number of their possible Boolean functions, so that the trials can cover all the linearly separable functions.

For $n = 4$, the average amount is 1304 which is 30% smaller than the correct number. As the number of Boolean functions is now 6 times larger than trials, it is highly possible the program cannot find all the functions needed.

When the dimension goes up to $n = 5$, the final result mainly outputs 0 but may sometimes output a single digit like 2 or 3. It means the program hardly finds any accepted functions. This is because the number of trials is too small compared with 94572 possible functions in 10^9 possibilities.