Code for One-step Error Probability task

```
import numpy as np
import random

# The nr of patterns
p_vector = [12, 24, 48, 70, 100, 120]
# the number of bits
N = 120
diagonal_zero = True  # set true if you want w_ii = 0 otherwise false

probability_error = []
for p in p_vector:
    # Creating a errors variable that will grow when errors occur( when updated_bit != bit)
    errors = 0
    for _ in range(10 ** 5):

        # creating p random patterns with 120 bits in a 120xp matrix
        patterns = np.random.randint(0, 2, (N, p))
        patterns[patterns == 0] = -1

        # weight_matrix = sum([np.dot(patterns[:, i:i + 1], patterns[:, i:i + 1].T) for i in rang
        weight_matrix = np.dot(patterns, patterns.T)
        if diagonal_zero:
            np.fill_diagonal(weight_matrix, 0)

        # Drawing a random number between 0 and 11 to choose randomly one pattern to feed
        random_pattern_index = random.randint(1, 11)
        # Taking out one pattern to be feed randomly
        feed_pattern = patterns[:, random_pattern_index]
        # feed_pattern = patterns[:, random_pattern_index:random_pattern_index + 1]

        # Taking one bit/ neuron to update randomly
        random_bit_index = random.randint(0, 119)
        random_bit = feed_pattern[random_bit_index]

        # weight = weight_matrix[random_bit_index:random_bit_index + 1, :]
        weight = weight_matrix[random_bit_index, :]
        b = np.dot(weight, feed_pattern) / N
        if b == 0:
            b = 1

        new_bit = np.sign(b)

        if new_bit != random_bit:
            errors += 1

    probability_error.append(errors / 10 ** 5)
```

```
print(probability_error)
```

Code for Recognising digits task

```
clear all
close all
clc

% Storing all the patterns
x1=[ [ -1, -1, -1, -1, -1, -1, -1, -1, -1, -1],[ -1, -1, -1, 1, 1, 1, 1, -1, -1, -1],[ -1, -1, 1
x2=[ [ -1, -1, -1, 1, 1, 1, 1, -1, -1, -1],[ -1, -1, -1, 1, 1, 1, 1, -1, -1, -1],[ -1, -1, -1, 1
x3=[ [ 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
x4=[ [ -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
x5=[ [ -1, 1, 1, -1, -1, -1, -1, 1, 1, -1],[ -1, 1, 1, -1, -1, -1, -1, 1, 1, -1],[ -1, 1, 1, -1,
% Storing the pattern that is going to be feeded
feedPattern = [[1, -1, -1, 1, 1, 1, 1, -1, -1, 1], [1, -1, -1, 1, 1, 1, 1, -1, -1, 1], [1, -1, -1
% Getting all the stored patterns into one matrix
storedPatterns = [x1; x2; x3; x4; x5]';
N = length(storedPatterns);

% Creating the weight matrix by dotproduct of the stored patterns
weightMatrix = storedPatterns * storedPatterns';
weightMatrix = weightMatrix./N;
n = size(weightMatrix,1);
for i = 1:size(weightMatrix,1)
    weightMatrix(i,i) = 0;      % making the diagonal weights to zero
end

updatedPattern = feedPattern;   % setting the updated pattern to feed pattern because
                                % The first iteration will be with the feed
                                % pattern
while true
    % Updating the pattern asynchronous determistic
    for i = 1:N
        weight = weightMatrix(i,:);
        localField = weight * updatedPattern;
        if localField == 0
            localField = 1;
        end
        newBit = sign(localField);
        updatedPattern(i,1) = newBit;

    end
    if isequal(updatedPattern,feedPattern)     % If updated pattern gets
                                               % to steady state, break
        break
    else
        feedPattern = updatedPattern;
    end
end
```

```
% Looping through all stored patterns to see if the updated pattern
% has converged to one of those (and also their inverted form)
for i = 1:5
    if isequal(updatedPattern,storedPatterns(:,i))
        fprintf('The updated pattern corresponds to the following pattern index %.0f\n', i)
    end
    if isequal(updatedPattern,-storedPatterns(:,i)) % Checks for inverted
        fprintf('The updated pattern corresponds to the following pattern index %.0f\n', -i)
    end
end

% To print the result. In other words the updated pattern.
fprintf(['[' strjoin(repmat({'%.17g'},1,size(updatedPattern',2)), ', ') '],\n'], updatedPattern.
```

Code for boolean functions task

```
%% Task to find linearly separable boolean functions for different dimensions
% Written by: Axel Qvarnström
clear all
close all
clc
n = 2; % Number of dimensions, just change to obtain result for different dimensions.
nTrials = 10^4;
nEpochs = 20;
eta = 0.05; % Learning rate
counter = 0;

booleanInputs = dec2bin(0:2^n-1)' - '0';
booleanInputs(booleanInputs == 0) = -1;     % Getting all zeros to -1
usedBooleanFuns = [];                       % A list for appending all already
                                            % tried boolean functions
k = 0;     % Index variabel used to store used boolean functions (see bottom of code)
for trial = 1:nTrials
    % Sample boolean functions
    booleanOutputs = randi([0,1],2^n,1);
    booleanOutputs(booleanOutputs == 0) = -1;


    boolFunInUsedBool = false;
    for i = 1:size(usedBooleanFuns,2)
        % To check if a boolean function is already used
        if isequal(usedBooleanFuns(:,i), booleanOutputs)
            boolFunInUsedBool = true;
            break
        end
    end
    if boolFunInUsedBool == true    % if so the case, skip to next iteration
        continue;
    end

    % Initializing weights and threshold
    weights = randn(n,1);
    threshold = 0;

    for epoch = 1:nEpochs
        totalError = 0;
        for mu = 1:2^n
            % Calculating the local field by summing (w(j)*x(j,mu) -
            % threshold) over j
            localField = 0;
            for j = 1:length(weights)
                localField = localField + (weights(j) * booleanInputs(j,mu) - threshold);
            end
```

```
            % Computing the output and checks if its equal to target or not
            y = sign(localField);
            error = booleanOutputs(mu) - y;

            % updating the weights and threshold
            deltaW = eta*(booleanOutputs(mu) - y).*booleanInputs(:,mu);
            deltaThreshold = -eta*(booleanOutputs(mu) - y);
            weights = weights + deltaW;
            threshold = threshold + deltaThreshold;

            totalError = totalError + abs(error);   % Calculating the total error

        end

        if totalError == 0
            counter = counter + 1;  % append everytime a boolean fun
                                    % is linearly separable
            break
        end
    end

    k = k+1;    % Index variable to append for every new boolean fun
                % into list for used boolean functions
    % Adding the boolean output to used boolean functions
    usedBooleanFuns(:,k) = booleanOutputs;

end

% Creating a variable for the nr of linearly separable functions
% and printing the result
nrOfLinearlySeparableFun = counter;
fprintf('Number of linearly separable functions for n = %0.f dimensions is %0.f'...
,n,nrOfLinearlySeparableFun);
```

1 Discussion of result

From the code I have written, the number of linearly separable boolean functions for dimensions n = 2, 3, 4, 5 is shown in the following table:

Number of dimensions n	Number of linearly separable boolean functions
2	14
3	104
4	206
5	0

Table 1: Number of linearly separable boolean functions for different number of dimensions

First I will explain the code briefly(will not go in detail), and from that I will discuss the result that is given in table 1. The code creates boolean inputs, that is all possible combination of 1 and -1 for n-dimensions. For instance if the dimension is n=2 then the boolean inputs is a vector containing all possible combination of 2-value pairs of 1 and -1 which is $2^n = 4$ possible combinations. For n=3 we have a input vector containing all 3-values pair of 1 and -1 which is $2^3 = 8$ possible combinations. after that 10^4 trials is made. For every trial random boolean outputs with 1 and -1 is generated. Than a loop for 20 epochs is made where for every epoch we are calculating the output using equation (5.9) from the course book. If the output differs from the target then we have an error and we have to update the weights and threshold according to the learning rules stated in the task description at OpenTa. When the output for a tested boolean function meet all the targets then we have a linearly separable boolean function and we count it in a counter. When a function is tested it is stored in a list. The list is then used to prevent testing the same function. The result for n = 2 and n = 3 matches exaclty, it's exactly correct. Comparing with n = 4 and n = 5 we do not get valid results because we only have 10^4 trials and the number of boolean functions is 2^{2^n} so it's 65536 and 4294967296 boolean functions for n = 4 and n = 5 respectively. We also for every trial generate randomly a boolean function which means that if we want to randomly generate all functions for n = 4 and n = 5 we have to increase the number of trials tremendously. For example consider n = 4, we need at least as many trials, in other words trials = 65536. But we are generating these function randomly which means that we with very high probability will generate a already generated function. Therefore we need the number of trials to be much much bigger than the number of boolean functions, $trials >> 65536$ to be sure that we are testing for all possible boolean functions, but I don't now how much bigger we need, I just know that for n = 4 and n = 5, we will get different number of linearly separable boolean functions for many of the runs of the code because of the number of trials and that we generate the boolean function randomly for every trial.