Homework 3: Tic tac toe

The goal of this exercise to write a computer program that trains two players to play tic-tac-toe by playing against each other.

Two players learn how to play tic tac toe with Q-learning. Train the two players simultaneously, playing against each other. The goal is to find a Q-table ensuring that the player never loses, see Section 11.5 in the Lecture notes. Use the \varepsilon$\varepsilon$-greedy policy, start with a suitable value of ε , and let ε tend to zero as training progresses.

Each player has his/her own Q-table. For an efficient implementation, do not initialise your Q-tables for every possible state of the tic tac toe board. Instead only initialise Q-entries when they are enountered for the first time during training. To this end, write a function that checks whether a board configuration occurred previously, or not. If not, add it to your Q-table and initialise it.

You need to upload two csv files. One should be named "player1.csv" and describe the Q-table of the player going first. The other one should be called "player2.csv" for the player going second. The csv files should have the following structure: The first three lines should consist of all the board states known to your Q-table, horizontally appended. A board position is implented as a 3-by-3 block, where an "X" is represented by a "1", an "O" by a "-1" and an empty field by a "0". The next three lines mimic the structure of the boards in the first three lines, except here an entry represents the expected value for a move in that location. Move values for occupied locations should be represented by NaN entries.

The following image visualises the structure on the file "examplecsv.csv", which is also available to download. Note that the specific numerical values in bottom three lines are not meaningful, they are only meant to demonstrate the structure of the file.

Hints: To easily obtain the desired .csv structure in Matlab, maintain the Q-table as a cell array with 2 rows and as many columns as there are known board states. In the first row, save a board state in every cell as a 3x3 matrix and in the second row the corresponding expected future rewards also as a 3x3 matrix with NaN entries on occupied fields. You can then use the function cell2mat followed by csvwrite to generate a .csv file with the right structure. The matlab function isnan can be very useful when debugging errors with the NaN entries. Note that Matlab evaluates arithmetic operations where one component is NaN entirely as NaN.

```
clc
clear
global Q1
global Q2
Q1 = cell(0);
Q2 = cell(0);
alpha = 0.1;
gamma = 1;
epsi = 1;
p1_wins = 0;
p2_wins = 0;
draws = 0;
p1_wins_list = zeros([1,300]);
p2_wins_list= zeros([1,300]);
```

```
draws_list = zeros([1,300]);
```

```
tic
for itr = 1:30000
    if ~mod(itr,1000)
        epsi = epsi * 0.95;
        if fix(itr/3000)<1
            epsi = 0.95;
        end
        p1_wins_list(fix(itr/100)) = p1_wins/100;
        p2_wins_list(fix(itr/100)) = p2_wins/100;
        draws_list(fix(itr/100)) = draws/100;
        p1_wins = 0;
        p2_wins = 0;
        draws = 0;
    end

    if ~mod(itr,500)
        fprintf("iteration: %d", itr)
        disp(size(Q1,2))
        disp(size(Q2,2))
        toc
        tic
    end
    % borad init
    board = zeros(3);
    isend = 0;
    player = 1;

    ifseen(board, 1);
    [action_board, action_coord_1] = choose_action(board, epsi, player, 1);
    board_prime = board + action_board;
    player = ~player;

    ifseen(board_prime, 2);
    [action_board, action_coord_2] = choose_action(board_prime, epsi, player, 2);
    board_next = board_prime + action_board;

    ifseen(board_next, 1);
    update_Q(board, board_next, action_coord_1, 1, alpha, gamma, 0);
    player = ~player;
    board = board_next;
    while true
        %% P1 round
        board_prime_old = board_prime;
        [action_board, action_coord_1] = choose_action(board, epsi, player, 1);
        board_prime = board + action_board;
        isend = boardcheck(board_prime);
        if isend
```

```
            [R_1, R_2] = reward_assign(isend, player);
            fupdate_Q(board, action_coord_1, 1, alpha, gamma, R_1);
            fupdate_Q(board_prime_old, action_coord_2, 2, alpha, gamma, R_2);
            if R_1 == 1
                p1_wins = p1_wins + 1;
            elseif R_1 == 0
                draws = draws + 1;
            end
            break
        end
        ifseen(board_prime, 2);
        update_Q(board_prime_old, board_prime, action_coord_2, 2, alpha, gamma, 0);
        player = ~player;

        %% P2 round
        [action_board, action_coord_2] = choose_action(board_prime, epsi, player, 2);
        board_next = board_prime + action_board;
        isend = boardcheck(board_next);
        if isend
            [R_1, R_2] = reward_assign(isend, player);
            fupdate_Q(board, action_coord_1, 1, alpha, gamma, R_1);
            fupdate_Q(board_prime, action_coord_2, 2, alpha, gamma, R_2);
            if R_2 == 1
                p2_wins = p2_wins + 1;
            elseif R_2 == 0
                draws = draws + 1;
            end
            break
        end
        ifseen(board_next, 1);
        update_Q(board, board_next, action_coord_1, 1, alpha, gamma, 0);
        player = ~player;
        board = board_next;
    end
end
```

```
figure
plot(p1_wins_list)
hold on
plot(p2_wins_list)
plot(draws_list)
legend("P1 wins", "P2 wins", "Draws")
```

```
Q1_mat = cell2mat(Q1);
Q2_mat = cell2mat(Q2);
csvwrite("player1.csv", Q1_mat)
csvwrite("player2.csv", Q2_mat)
```

```matlab
function Q = ifseen(board, Q)
    global Q1
    global Q2

    if Q == 1
        if isempty(Q1)
            Q1{1,1} = board;
            board(board~=0)=NaN;
            Q1{2,1} = board;
        else
            index = find(cellfun(@(x) isequal(x, board), Q1(1,:)), 1);
            if isempty(index)
                Q1{1,end+1} = board;
                board(board~=0)=NaN;
                Q1{2,end} = board;
            end
        end
    else
        if isempty(Q2)
            Q2{1,1} = board;
            board(board~=0)=NaN;
            Q2{2,1} = board;
        else
            index = find(cellfun(@(x) isequal(x, board), Q2(1,:)), 1);
            if isempty(index)
                Q2{1,end+1} = board;
                board(board~=0)=NaN;
                Q2{2,end} = board;
            end
        end
    end
end

%% Distribute reward to players
function [R_1, R_2] = reward_assign(isend, player)
    R_1 = 0;
    R_2 = 0;
    if isend == 1 && player == 1
        R_1 = 1;
        R_2 = -1;
    elseif isend == 1 && player == 0
        R_1 = -1;
        R_2 = 1;
    elseif isend == 0.5
        R_1 = 0;
        R_2 = 0;
    end
end

% choose A from S using epsilon-greedy
```

```
function [action_board, action_coord] = choose_action(board, epsi, which_player, Q)
    global Q1
    global Q2
    action_board = zeros(3);
    if binornd(1, epsi) == 1
%         disp("randomly choose")
        [row, col] = find(board==0);
        randchoice = randi(length(row));
        action_coord = [row(randchoice); col(randchoice)];
    else
        if Q==1
            index = find(cellfun(@(x) isequal(x, board), Q1(1,:)), 1);
            Q_values = Q1{2, index};
            max_qvalue = max(Q_values(:));
            [x,y] = find(Q_values == max_qvalue);
            if length(x) > 1
                randchoice = randi(length(x));
                action_coord = [x(randchoice); y(randchoice)];
            else
                action_coord = [x;y];
            end
        else
            index = find(cellfun(@(x) isequal(x, board), Q2(1,:)), 1);
            Q_values = Q2{2, index};
            max_qvalue = max(Q_values(:));
            [x,y] = find(Q_values == max_qvalue);
            if length(x) > 1
                randchoice = randi(length(x));
                action_coord = [x(randchoice); y(randchoice)];
            else
                action_coord = [x;y];
            end
        end
    end
    if which_player == 1
        action_board(action_coord(1),action_coord(2)) = 1;
    else
        action_board(action_coord(1),action_coord(2)) = -1;
    end
end

%% Q update when game continues
function update_Q(board_old, board_new, action_coord, Q, alpha, gamma, reward)
    global Q1
    global Q2
    if Q == 1
        index_old = find(cellfun(@(x) isequal(x, board_old), Q1(1,:)), 1);
        index_new = find(cellfun(@(x) isequal(x, board_new), Q1(1,:)), 1);
        max_Q = max(Q1{2, index_new}(:));
        Q_value = Q1{2, index_old}(action_coord(1), action_coord(2));
        Q1{2, index_old}(action_coord(1), action_coord(2)) = Q_value + alpha * (reward + gamma
    else
        index_old = find(cellfun(@(x) isequal(x, board_old), Q2(1,:)), 1);
        index_new = find(cellfun(@(x) isequal(x, board_new), Q2(1,:)), 1);
        max_Q = max(Q2{2, index_new}(:));
        Q_value = Q2{2, index_old}(action_coord(1), action_coord(2));
        Q2{2, index_old}(action_coord(1), action_coord(2)) = Q_value + alpha * (reward + gamma
    end
end

%% Q update when game over
function fupdate_Q(board_old, action_coord, Q, alpha, gamma, reward)
    global Q1
    global Q2
    if Q == 1
        index_old = find(cellfun(@(x) isequal(x, board_old), Q1(1,:)), 1);
        Q_value = Q1{2, index_old}(action_coord(1), action_coord(2));
        Q1{2, index_old}(action_coord(1), action_coord(2)) = Q_value + alpha * (reward - Q_valu
    else
        index_old = find(cellfun(@(x) isequal(x, board_old), Q2(1,:)), 1);
        Q_value = Q2{2, index_old}(action_coord(1), action_coord(2));
        Q2{2, index_old}(action_coord(1), action_coord(2)) = Q_value + alpha * (reward - Q_valu
    end
end

%% Check if game over
function isend = boardcheck(board)
    isend = 0;
    board_flip = fliplr(board);
    row_sum = abs(sum(board,1));
    col_sum = abs(sum(board,2));
    if abs(sum(diag(board)))==3 || abs(sum(diag(board_flip))) == 3 || any(col_sum(:)==3) || any
        isend = 1;
        return
    end
    if all(board,"all")
        isend = 0.5;
        return
    end
end
```

Homework 3 Tic-Tac-Toe

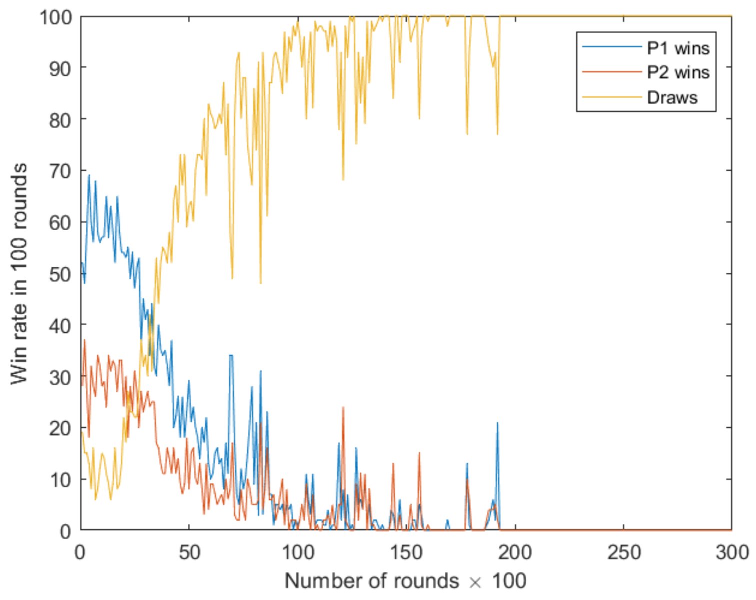


Figure 1: Winning rate respect to each 100 rounds

Since player 1 always start first, it's natural he will have a larger win rate at the beginning. The ϵ-greedy policy with $\epsilon = 0.95$ in the first 3000 rounds makes the plot slightly different from other's result, that non of the players plays with best solution before 3000 rounds. However it encourages the players to explore as much as possible to avoid keeping exploiting a local optimal strategy. Otherwise the final Q-table will not cover enough states, which will result in a failure.

As the training continues, both players gradually choose optimal action from the Q-table. Thus the wining rate decays and the draw rate increases until it's no longer possible for any player to win.

With the final Q-table, the agent will always find an optimal action to win, or at least draw, even when playing against human.

Homework 3: Chaotic time series prediction

The files training_set.csv and test_set.csv contain 3-dimensional time series generated from the chaotic Lorenz dynamics (see link). The time series was generated using a time step size of $dt = 0.02$ seconds. Use the training set to train a reservoir computer to predict the Lorenz dynamics.

The reservoir has $N = 3$ input neurons and 500 reservoir neurons. The input weights $w_{ij}^{(in)}$ and reservoir weights w_{ij} are independent Gaussian random numbers with mean zero and variances 0.002 and 2/500 respectively.

Use the following update rule for the reservoir dynamics

$$r_i(t+1) = \tanh\left(\sum_j w_{ij} r_j(t) + \sum_{k=1}^{N} w_{ik}^{(in)} x_k(t)\right)$$

The output of the reservoir computer is given by

$$O_i(t) = \sum_{j=1} w_{ij}^{(out)} r_j(t)$$

Train the output weights $w_{ij}^{(out)}$ using ridge regression with ridge parameter $k = 0.01$ (see link). When the test data has been fed through the network, use the following update rules to predict the continuation of the test data:

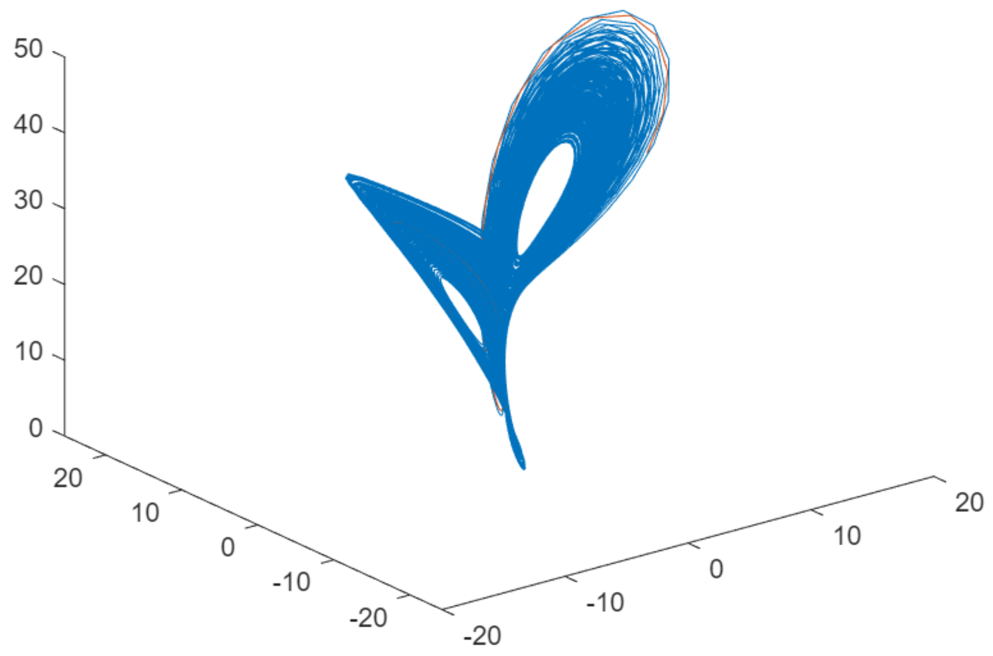
$$r_i(t+1) = \tanh\Big(\sum_{j=1}^{N} w_{ij} r_j(t) + \sum_{k=1}^{n} w_{ik}^{(in)} O_k(t)\Big)$$

Use the reservoir to predict 500 time steps and save the *y*-component of the output $O_2(t)$ for every time step. After training, feed the test data through the network using the same update rules. Note that the output should be saved starting from $O_2(T+1)$, where T is the length of the test set. Thus, the saved data will range from $O_2(T+1)$ to $O_2(T+501)$.

```
clc
clear
w_in = normrnd(0,0.002,[500,3]); % 500x3
w_reservoir = normrnd(0,2/500, [500,500]); % 500x500
w_reservoir = w_reservoir -  diag(w_reservoir);
w_out = zeros([3,500]); % 3x500
input = csvread("training-set.csv");
test_data = csvread("test-set-6.csv");
input_length = size(input,2);
test_length = size(test_data, 2);
k = 0.01;
```

Plotting

```
figure
plot3(input(1,:), input(2,:),input(3,:))
hold on
plot3(test_data(1,:), test_data(2,:),test_data(3,:))
```



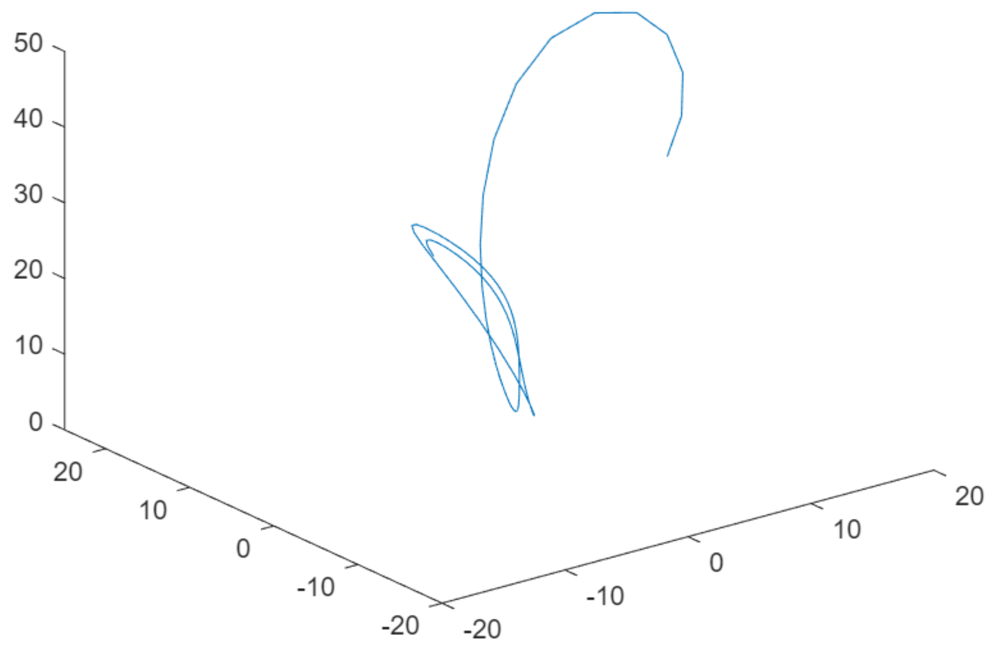
```
R = zeros([500,1]); % 500x1
R_old = R;
for epoch=1:5
    for idx=1:input_length
        R(:,idx) = tanh(w_reservoir*R_old + w_in * input(:,idx));
        R_old = R(:,idx);
    end
    w_out = input*R'*inv((R*R'+k*eye(500)));
end
```

```
R_old = zeros([500,1]);
R = R_old;
for idx=1:test_length
    R(:,idx) = tanh(w_reservoir*R_old + w_in * test_data(:,idx));
    R_old = R(:,idx);
end
O = w_out *R;
R_old = zeros([500,1]);
neuron_R = R_old;
```

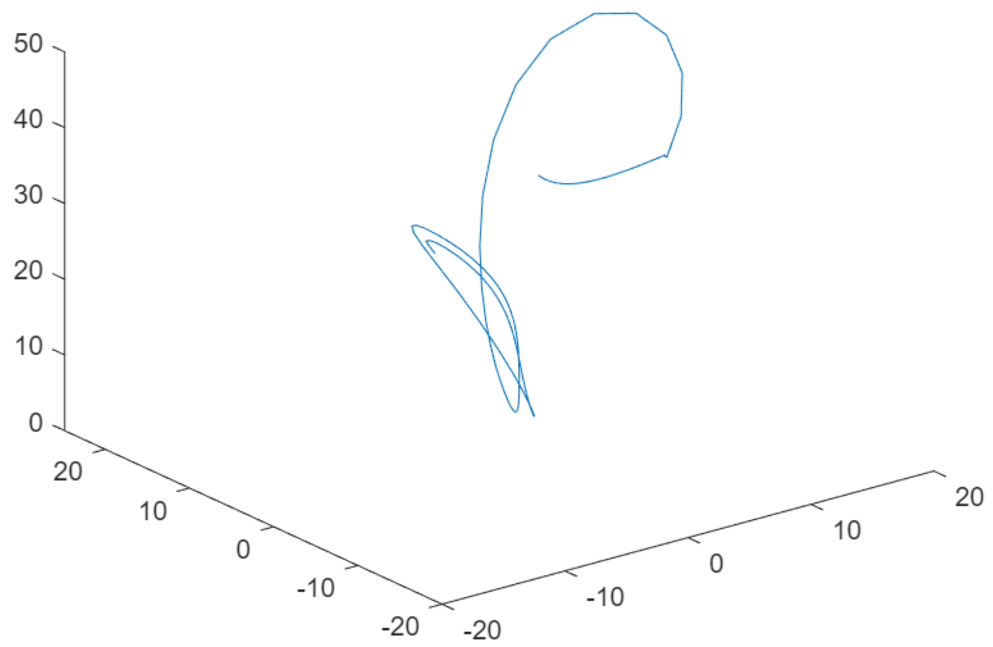
```
for step=1:500
    neuron_R = tanh(w_reservoir*R_old + w_in * O(:,test_length+step-1));
    O(:,test_length+step) = w_out*neuron_R;
    R_old = neuron_R;
end
```

```
y = O(2,101:end)';
csvwrite("prediction.csv",y);
```

```
figure
plot3(test_data(1,:), test_data(2,:),test_data(3,:))
```



```
% hold on
plot3(O(1,:), O(2,:),O(3,:))
```



50
40
30
20
10
0
20
10
0
-10
-20
-20
-10
0
10
20

Homework 3: Self organising map

The files iris-data.csv and iris-labels.csv contain input and targets from the Iris-flower data set (see Ref. [70] in the course book), with data from three species of Iris flowers. You can read more about the data set here: link. Train a self-organising map (p. 185 in the course book) to cluster the different flower species in the Iris data set. To do this, use the learning rule Eq. (10.17)

$$\delta \mathbf{w}_i = \eta h(i, i_0)(\mathbf{x} - \mathbf{w}_i)$$

where the neighbourhood function $h(i, i_0)$ is defined as (Eq. 10.18)

$$h(i, i_0) = \exp\left(-\frac{1}{2\sigma^2}|\mathbf{r}_i - \mathbf{r}_{i_0}|^2\right)$$

where $\mathbf{r}_i$ is the position of neuron i in the output array.

Initialize the weight array by sampling W_{ij} from a uniform distribution in the interval [0,1]. Use a 40×40 output array -- since the input dimension is $n = 4$, the weight array has dimensions (40,40,4). The learning rate η and the width σ of the neighbourhood function depend on the iteration number:

$$\eta = \eta_0 \exp(-d_\eta \times \text{epoch})$$

and

$$\sigma = \sigma_0 \exp(-d_\sigma \times \text{epoch})$$

Here the initial learning rate is $\eta_0 = 0.1$ with a decay rate of $d_\eta = 0.01$, and the initial width of the neighbourhood function is $\sigma_0 = 10$ with a decay rate $d_\sigma = 0.05$.

Standardise the input data by dividing all values in the dataset with the maximal value of the dataset. Train the self-organising map for 10 epochs using a batch size of 1. One epoch corresponds to making p stochastic weight updates, where p is the number of patterns in the iris data set.

Make a figure with two panels: one panel shows, for each input, the location of the winning neuron in the output array, colour-coded according to class, using the randomly chosen initial weights. The second panel shows the same, but for the final weights obtained after iterating the learning rule. You will find that the samples are organised into three separate clusters, one for each class. Include a legend for the three classes in the figure caption.

```
clc
clear
iris_raw = csvread("iris-data.csv");
input = iris_raw/(max(iris_raw,[],'all')); % normalization
sz = size(input);
noise = normrnd(0, 0.01, sz);
input_n = input+noise;
labels = csvread("iris-labels.csv");
length = size(input,1);
```

```
epoches = 20;
init_eta = 0.1;
eta = init_eta;
eta_drate= 0.01;

init_sigma = 10;
sigma_drate = 0.05;
sigma = init_sigma;
```

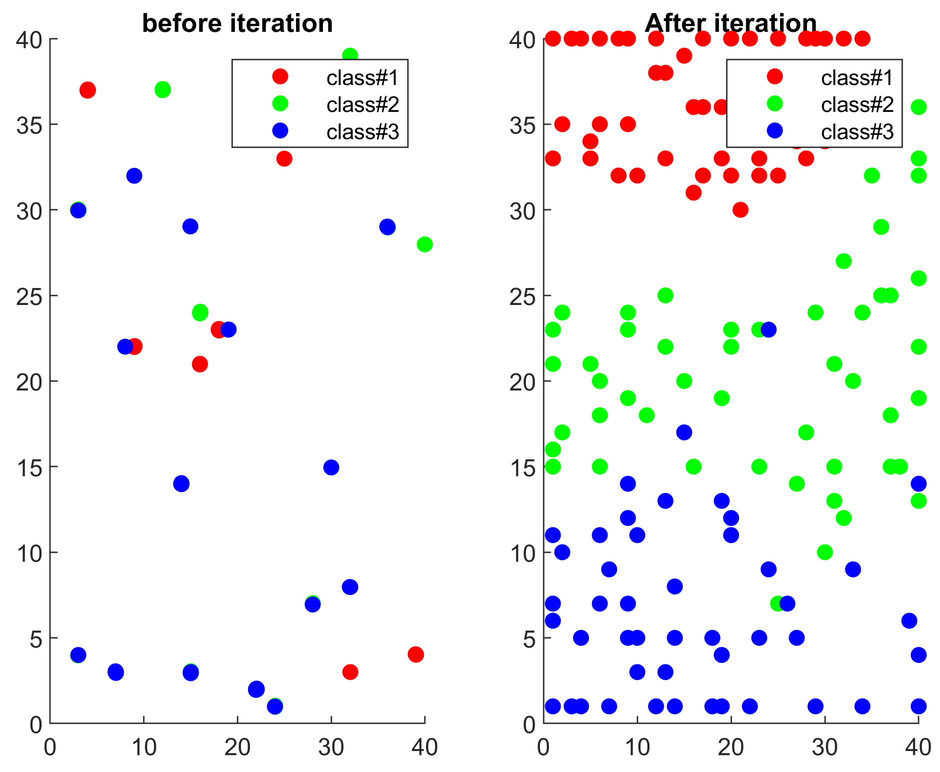
```
W_init = rand([40,40,4]);
W = W_init;
for epo = 1:epoches
    eta = init_eta*exp(-eta_drate*epo);
    sigma = init_sigma*exp(-sigma_drate*epo);
    for idx = 1:length
        X = input(randi(length),:)';
        i0 = find_minimal_distance(W,X);
        for i=1:40
            for j = 1:40
                neuron_i = [i;j];
                v = neuron_i-i0;
                if norm(v) < 3*sigma
                    h = neightbour_hood(v,sigma);
                    delta_w = eta * h * (X-squeeze(W(i,j,:)));
                    for k=1:4
                        W(i,j,k) = W(i,j,k) + delta_w(k);
                    end
                end
            end
        end
    end
end
```

```
% Plot the classification with randomlized weight
figure
subplot(1,2,1)
hold on
coord1 = zeros([2,length]);
for idx = 1:length
    X = input(idx,:)';
    i0 = find_minimal_distance(W_init, X);
    coord1(:,idx) = i0+normrnd(0, 0.02, [2,1]);
end

scatter(coord1(1,1:50), coord1(2,1:50),[], "r", "filled")
scatter(coord1(1,51:100), coord1(2,51:100),[], "g", "filled")
scatter(coord1(1,100:150), coord1(2,100:150),[], "b", "filled")
legend("class#1","class#2","class#3")
```

```
title("before iteration")

% Plot the classification with final weight
subplot(1,2,2)
coord = zeros([2,length]);
for idx = 1:length
    X = input_n(idx,:)';
    i0 = find_minimal_distance(W, X);
    coord(:,idx) = i0;
end
scatter(coord(1,1:50), coord(2,1:50),[], "r", "filled")
hold on
scatter(coord(1,51:100), coord(2,51:100),[], "g", "filled")
scatter(coord(1,100:150), coord(2,100:150),[], "b", "filled")
legend("class#1","class#2","class#3")
title("After iteration")
```



```
function i0 = find_minimal_distance(W, X)
    i0 = zeros([2,1]);
    minimal_d = 100;
    for i =1:40
        for j = 1:40
            current_weight = squeeze(W(i,j,:));
            temp_d = sqrt(dot(current_weight-X, current_weight-X));
            if temp_d < minimal_d
                i0 = [i;j];
                minimal_d = temp_d;
                d = X-current_weight;
            end
        end
    end
end

function h = neightbour_hood(v,sigma)
    h = exp((-1/(2*sigma^2)) *  norm(v)^2);
end

% function w = weight_update()
```

Homework 3 Self-Organising Map

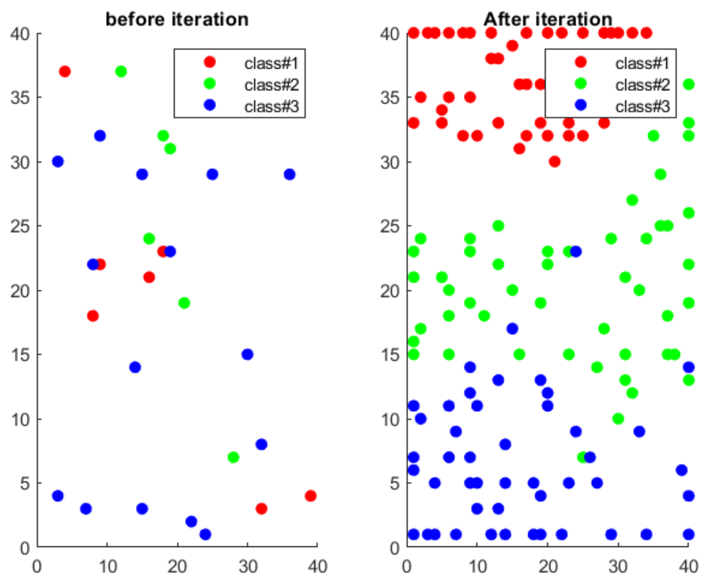


Figure 1: The location of the winning neuron in the output array

The plot on the left shows the closest location of the winning neuron before learning. Since the initial weights are randomly generated, the winning neurons are therefore locates randomly.

However, from the plot on the right, the neurons are clearly separated into three parts, corresponding to three types of iris. It illustrates that if the patterns are close to each other, they should activate neurons also close to each other. The patterns belong to the first class are strictly separated(red dots). While some of the green dots are overlapped with blue ones, which means some of the irises are missed classified or perhaps these patterns are quite similar to each other.

Due to the random initial weights, the figure varies from each experiment. But it should give three clear clusters in general.