Code for Chaotic time series prediction

```
% Chaotic time series prediction, written by Axel Qvarnström

clear all
close all
clc

N = 500;  % numbers of reservoirs
trainingData = load('training-set.csv');
testData = load('test-set-7.csv');
Ttrain = length(trainingData);
Ttest = length(testData);

% Initializing weights
inputWeights = normrnd(0, 0.002, [500, 3]);
reservoirWeights = normrnd(0, 2/500, [500, 500]);

reservois = zeros(N,1); % initialize reservoir time step zero
reservoirMatrix = zeros(N,Ttrain);

% Updating the dynamics of the reservoir
for t = 1:Ttrain
    reservoirMatrix(:,t) = reservois;
    reservoisUpdated = UpdateReservoir(reservoirWeights, reservois, inputWeights, trainingData(:
    reservois = reservoisUpdated;
end



identityMatrix = eye(N,N);
k = 0.01;

% Training the output weights with ridge regression
outputWeights = RidgeRegression(reservoirMatrix, k, trainingData, identityMatrix);

%% Feed the test data and make predictions

% Looping through the test data for timesteps = 100
for t = 1:Ttest
    reservois = UpdateReservoir(reservoirWeights, reservois, inputWeights, testData(:,t));
    outputNeurons = Output(outputWeights, reservois);
end

% Making the prediction for 500 timesteps
outputNeurons2Matrix = zeros(3,500);
for T = 1:500
    reservois = UpdateReservoir(reservoirWeights, reservois, inputWeights, outputNeurons);
    outputNeurons2Matrix(:,T) = Output(outputWeights,reservois);
end
```

```
StoredOutputNeurons = outputNeurons2Matrix(2,:);
csvwrite('prediction.csv',StoredOutputNeurons)

    function output = Output(outputWeigths, reservoirs)

    output = outputWeigths * reservoirs;

end

    function outputWeights = RidgeRegression(R, k, xTrain, identityMatrix)

    outputWeights = xTrain * R' *(R * R' + k*identityMatrix);

end

    function newReservoirs = UpdateReservoir(weights, reservoirs, inputWeights, inputNeurons)

    newReservoirs = tanh(weights * reservoirs + inputWeights * inputNeurons);

end
```

Code for Self organising map task

```
% Self organising map. Written by: Axel Qvarnström
clear all
close all
clc

% Loading the data and labels
data = load('iris-data.csv');
labels = load('iris-labels.csv');

% Standardise the data
data = data ./ max(data);

% Initializing
initWeightArray = rand([40,40,4]);
weightArray = initWeightArray;
learningRate = 0.1;
learningDecay = 0.01;
width = 10;
widthDecay = 0.05;


nEpochs = 10;
nDataPoints = length(data);



for epoch = 1:nEpochs
    [learningRate, width] = Decay(learningRate, width, learningDecay, widthDecay, epoch);
    for i = 1:150
        randomDataPointIndex = randi(nDataPoints);
        input = data(randomDataPointIndex,:);
        term1Final = (weightArray(:,:,1) - input(1)).^2;
        term2Final = (weightArray(:,:,2) - input(2)).^2;
        term3Final = (weightArray(:,:,3) - input(3)).^2;
        term4Final = (weightArray(:,:,4) - input(4)).^2;
        distanceFinal = sqrt(term1Final + term2Final + term3Final + term4Final);

        % Updating
        minPos = WinningNeuronPos(distanceFinal);
        weights2Update = squeeze(weightArray(minPos(1),minPos(2),:))';
        neighbourhoodfun = NeighbourhoodFun(minPos, minPos, width);
        deltaWeights = DeltaWeights(learningRate, neighbourhoodfun, input, weights2Update);
        weights2Update = weights2Update + deltaWeights;
        weightArray(minPos(1),minPos(2),:) = weights2Update;

        % Updating Weights for the close neurons
        weightArray = UpdateWeightsForCloseNeurons(distanceFinal, learningRate, input,...
```

```
            weightArray, minPos,width);

    end

end

% Create empty lists to append the winning neuron positions using the
% final weight array. The different list correspond to the different
% labels
list1FinalWeight = zeros(50,2);
list2FinalWeight = zeros(50,2);
list3FinalWeight = zeros(50,2);

% Create empty lists to append the winning neuron positions using the
% initializing weight array. The different list correspond to the different
% labels
list1InitWeight = zeros(50,2);
list2InitWeight = zeros(50,2);
list3InitWeight = zeros(50,2);

% Creating indexes to append winning neurons to the right spot in the lists
list1Index = 1;
list2Index = 1;
list3Index = 1;

for j = 1:150
    input = data(j,:);
    label = labels(j);
    % Process for taking out the winning neuron position base on the final
    % updated weight array
    term1Final = (weightArray(:,:,1) - input(1)).^2;
    term2Final = (weightArray(:,:,2) - input(2)).^2;
    term3Final = (weightArray(:,:,3) - input(3)).^2;
    term4Final = (weightArray(:,:,4) - input(4)).^2;
    distanceFinal = sqrt(term1Final + term2Final + term3Final + term4Final);
    winningNeuronPositionFinal = WinningNeuronPos(distanceFinal);

    % Process for taking out the winning neuron position base on the
    % initial weight array
    term1Init = (initWeightArray(:,:,1) - input(1)).^2;
    term2Init = (initWeightArray(:,:,2) - input(2)).^2;
    term3Init = (initWeightArray(:,:,3) - input(3)).^2;
    term4Init = (initWeightArray(:,:,4) - input(4)).^2;
    distanceInit = sqrt(term1Init + term2Init + term3Init + term4Init);
    winningNeuronPositionInit = WinningNeuronPos(distanceInit) + normrnd(0,0.04,[1,2]);

    if label == 0
        list1FinalWeight(list1Index,:) = winningNeuronPositionFinal;
```

```
        list1InitWeight(list1Index,:) = winningNeuronPositionInit;
        list1Index = list1Index + 1;
    end
    if label == 1
        list2FinalWeight(list2Index,:) = winningNeuronPositionFinal;
        list2InitWeight(list2Index,:) = winningNeuronPositionInit;
        list2Index = list2Index + 1;
    end

    if label == 2
        list3FinalWeight(list3Index,:) = winningNeuronPositionFinal;
        list3InitWeight(list3Index,:) = winningNeuronPositionInit;
        list3Index = list3Index + 1;
    end

end

% Panel 1 based on the initial weight array
subplot(1,2,1);
scatter(list1InitWeight(:,1), list1InitWeight(:,2),'green','filled')
hold on
scatter(list2InitWeight(:,1), list2InitWeight(:,2),'red','filled')
scatter(list3InitWeight(:,1), list3InitWeight(:,2),'blue','filled')
legend('Iris Setosa', 'Iris Versicolour', 'Iris Virginica')
title('Panel 1, winning neuron positions using initial weights')

% Panel 2 based on the final weigth array
subplot(1,2,2);
scatter(list1FinalWeight(:,1), list1FinalWeight(:,2),'green','filled')
hold on
scatter(list2FinalWeight(:,1), list2FinalWeight(:,2),'red','filled')
scatter(list3FinalWeight(:,1), list3FinalWeight(:,2),'blue','filled')
legend('Iris Setosa', 'Iris Versicolour', 'Iris Virginica')
title('Panel 2, winning neuron positions using final weights')
```

Function to update learning rate and the neighbourhood width for each epoch:

```
    function [learningRate, width] = Decay(learningRate, width, learningDecay,...
    widthDecay, epoch)

    learningRate = learningRate * exp(-learningDecay * epoch);
    width = width * exp(-widthDecay * epoch);

end
```

Function for calculate the neighbourhood function value:

```
    function funValue = NeighbourhoodFun(pos, minPos, width)
```

```
    nominator = norm(pos - minPos)^2;
    denominator = 2 * width^2;
    funValue = exp(-nominator / denominator);

end
```

Function to calculate the delta weights:

```
    function deltaW = DeltaWeights(learningRate, neighbourhoodFun, inputs, Weight2Update)

    deltaW = learningRate * neighbourhoodFun * (inputs - Weight2Update);

end
```

Function that update the weights based on the neurons that are in a distance $< 3\sigma$:

```
function weightArray = UpdateWeightsForCloseNeurons(distance, learningRate, input,...
    weightArray, minPos,width)

    closeDistances = distance(distance < 3*width);
    nDistances = length(closeDistances);

    for i = 1:nDistances
        [x, y] = find(distance == closeDistances(i));
        closePos = [x, y];
        weights2Update = squeeze(weightArray(x,y,:))';
        neighbourhoodfun = NeighbourhoodFun(closePos, minPos, width);
        deltaWeights = DeltaWeights(learningRate, neighbourhoodfun, input, weights2Update);
        weights2Update = weights2Update + deltaWeights;
        weightArray(x,y,:) = weights2Update;
    end

end
```

Function to calculate the position for the winning neuron:

```
    function winningNeuronPos = WinningNeuronPos(distance)

    minDistance = min(distance, [],"all");
    [x,y] = find(distance == minDistance);
    winningNeuronPos = [x, y];

end
```

We are given a data set called Iris data set, where we get 150 patterns where each pattern has 4 parameters, its in other words 4 dimensional. We are also given the corresponding labels. The data consist of three classes (Iris Setosa, Iris Versicolour, Iris Virginica). What we do in the self organising map is to represent this high dimensional data (4-dimensions) in a lower dimension(2-dimensional map), where the data is clustered into three clusters in this case. The clusters are the three classes, and the reason of doing this self organising map is to make it easier to visualize this high dimensional data.
The algorithm I have written is in the training process looping over 10 epochs, mini-batchsize = 1. Then there is a innerloop looping 150 times where a random data point/pattern is picked. The pattern has 4 input parameters so we have 4 weights between each input and each neuron. We are calcluating the distance between the random choosed datapoint and all the neurons. We take out the position of the winning neuron which is the neuron that is connected with the random choosed datapoint where the distance between them, is the smallest distance. We update the weights corresponding to the winning neuron and we also update the weights corresponding to neurons close to this winning neuron. After these 10 epochs the training is done and the weight array that is initialized (with values in range [0, 1] uniform distributed) in the beginning has changed.
Next Step is to loop through all the data points sequentially and plotting the winning neuron using both the final updated weight array from the training explained above and also the initialized weight array in two different panels, to see how the data gets arrange to different clusters by the training.
The result is visualized in the following plot:

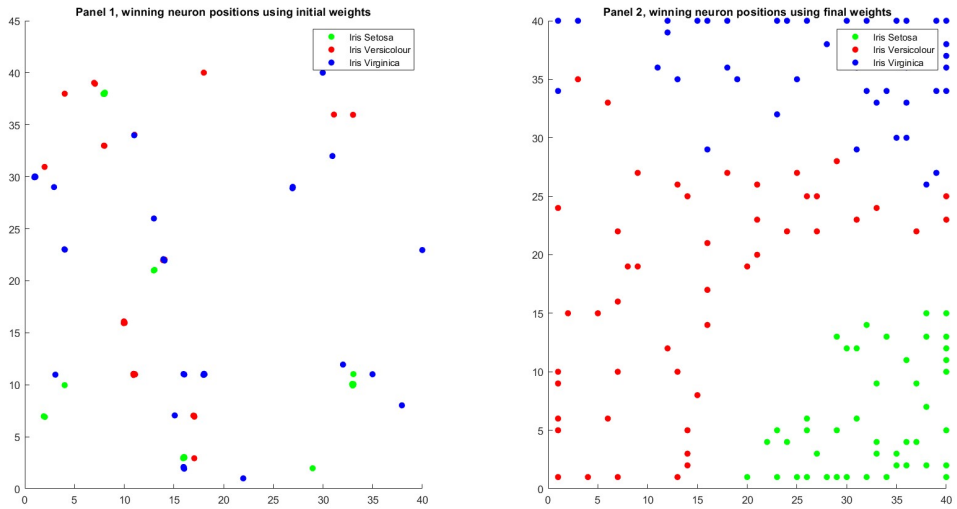

Figure 1: The left panel shows the location of the winning neurons that is calculated using randomly chosen initial weights. Right panel shows the location of the winning neurons that has been calculated using the final weights after iterating the learning rule. The different colors show which class they belongs to

We can see from the plot, the development of updating the weights with the learning rule Eq.(10.17) from the course book, that the winning neurons have grouped into clusters to the class where they belong, using the final update of the weights(compare the left panel to the right panel). The link with the information about the Iris dataset says that

one of the classes are linearly separable from the other two and that the latter are not. We can clearly see that from the above plot, that the green dots(Iris setosa) is linearly separable from the other two, while the red and blue dots seperates clearly but some points differ and gets close to each other.