EE7207 Neural & Fuzzy Systems Assignment 1

Wu Tianwei

Matric No. G2101446F

e-mail: WU0008EI@e.ntu.edu.sg

RBF neural network classifier

To train the network, the provieded training set need to be preprocessed. Fisrtly, the data will be shuffled to make sure the sequence of the data has no influence on the training. Then, the data will be divided into training set, validation set and test set. As the test set is provided, the ratio of test samples will be set to 0. Here, the ratio of validation samples is set to 20%. The data for each subset is picked randomly.

① Data preprocessing

```
% Data preparation
load('data_train.mat');
load('label_train.mat');

data_and_label = [data_train,label_train];
data_and_label = shuffling(data_and_label);

[train_set, valid_set, test_set] = data_divider(data_and_label, 20, 0);

% Seperate data and labels
train_labels = train_set(:, end);
valid_labels = valid_set(:, end);
test_label = test_set(:, end);

train_data = train_set(:, 1 : end - 1);
valid_data =  valid_set(:, 1 : end - 1);
test_data = test_set(:, 1 : end - 1);
```

② Find the centers for RBF neural network

```
% Using SOM to find centers for RBF neural network
data = train_data;
rows = 8;
cols = 8;
learningRate1 = 0.1;
learningRate2 = 0.01;
iterLimit = 1000;
c = SOM(data, rows, cols, learningRate1, learningRate2, iterLimit)
```

```
c = 64×33
    1.0000    0.9515    0.0481    0.9313    0.0174    0.9361    0.0125    0.9488 ···
    0.4717    0.8748    0.0636    0.9566    0.3069    0.9306    0.9042    0.9434
    0.9995    0.9203    0.4375    0.9753   -0.2880    0.9409   -0.0586    0.8600
    1.0000    0.9464    0.0971    0.9530    0.2258    0.9031    0.1874    0.8667
    0.9991    0.5937   -0.0472    0.9107    0.0520    0.7715    0.6503    0.5996
    0.8987    0.7300    0.0360    0.6794    0.1318    0.6249    0.1345    0.6164
    0.9641    0.7976    0.1107    0.6988    0.0858    0.6368    0.1039    0.3455
    0.4305    0.9487   -0.8528    0.9992    0.4383    0.9999   -0.9492    0.9709
    0.9144    0.9593    0.0155    0.8664   -0.0293    0.9669   -0.3297    0.9569
    1.0000    0.7812    0.0438    0.8300    0.0336    0.8198    0.0729    0.8021
       ⋮
```

The learning rates of self-organizing phase and convergence phase are set to 0.1 and 0.01. The number of hidden layer neurons is computed by multiplying parameter rows and cols. As it's not privided, the suitable number is determined by trail and error. After trying different number of neurons and calculating the average accuracy, it's finally set to 64 (8 * 8). The average accuracy of training is about 98.04%. The average accuracy of validation is about 94.09%.

③ Calculate Phi of f = Phi * W where f is the network's output

```
% Calculate Phi
[rows, cols] = size(c);
d = zeros(rows, rows);
for i = 1 : rows
    for j = 1 : rows
        if j < i
            d(i, j) = sqrt((c(j, :) - c(i, :)) * (c(j, :) - c(i, :))') / 2;
        end
    end
end
sigma = max(max(d));
Phi = calculate_Phi(train_data, c, sigma);
[rows, ~] = size(Phi);
Phi = [Phi, ones(rows, 1)];
```

Here, Gaussian basis function is used. The width of the basis function sigma is determined by maximum distance of any two centers.

④ Train the weigths of RBF neural network

```
W = weights_regression(Phi, train_labels);
W'
```

```
ans = 1×65
10^3 ×
   -0.0056    0.0094    0.0137   -0.0159    0.0171    0.3716   -0.0438    0.0064 ⋯
```

The weight is calculated by using LSM.

⑤ Cassification accuracy of the training data

```
% Test the RBF network on train set:
output_train = Phi * W;
index_minus = find(output_train < 0);
index_plus = find(output_train >= 0);
```

```
output_train(index_minus) = -1;
output_train(index_plus) = 1;
e = output_train - train_labels;
accuracy_train = 1 - length(nonzeros(e)) / length(output_train)
```

```
accuracy_train = 0.9628
```

⑥ Cassification accuracy of the validation data

```
% Test the RBF network on valid set:
Phi_valid = calculate_Phi(valid_data, c, sigma);
[rows, ~] = size(Phi_valid);
Phi_valid = [Phi_valid, ones(rows, 1)];
output_valid = Phi_valid * W;
index_minus_valid = find(output_valid < 0);
indexes_plus_valid = find(output_valid >= 0);
output_valid(index_minus_valid) = -1;
output_valid(indexes_plus_valid) = 1;
e_valid = output_valid - valid_labels;
accuracy_valid = 1 - length(nonzeros(e_valid)) / length(output_valid)
```

```
accuracy_valid = 0.9697
```

⑦ Classification result on provided test data

```
% Classification on test data
load('data_test.mat');
Phi_test = calculate_Phi(data_test, c, sigma);
[rows, ~] = size(Phi_test);
Phi_test = [Phi_test, ones(rows, 1)];
output_test = Phi_test * W;

output_test_modified = output_test;
index_minus_test = find(output_test_modified < 0);
index_plus_test = find(output_test_modified >= 0);
output_test_modified(index_minus_test) = -1;
output_test_modified(index_plus_test) = 1;

output_test_modified'
```

```
ans = 1×21
     1    -1     1     1     1    -1     1    -1     1    -1     1    -1     1···
```

Kernel SVM classifier

The data processing is the same as RBF neural network and Gaussian kernel function is used. Here, the SVM model is constructed and trained by using the built-in function 'fitcsvm' in Matlab.

```
% Train the kernel SVM classifier
SVMModel = fitcsvm(train_data, train_labels, ...
    'KernelFunction', 'gaussian', 'ClassNames', [-1, 1]);

% Test the SVM on train set
[output_train_1, score_train] = predict(SVMModel, train_data);
index_minus_1 = find(output_train_1 < 0);
index_plus_1 = find(output_train_1 >= 0);
output_train_1(index_minus_1) = -1;
output_train_1(index_plus_1) = 1;
e_1 = output_train_1 - train_labels;
accuracy_train_1 = 1 - length(nonzeros(e_1)) / length(output_train_1)
```

```
accuracy_train_1 = 0.9926
```

```

% Test the RBF network on valid set:
[output_valid_1, score_valid] = predict(SVMModel, valid_data);
index_minus_valid_1 = find(output_valid_1 < 0);
indexes_plus_valid_1 = find(output_valid_1 >= 0);
output_valid_1(index_minus_valid_1) = -1;
output_valid_1(indexes_plus_valid_1) = 1;
e_valid_1 = output_valid_1 - valid_labels;
accuracy_valid_1 = 1 - length(nonzeros(e_valid_1)) / length(output_valid_1)
```

```
accuracy_valid_1 = 0.9242
```

```

% Classification on test data
load('data_test.mat');
[output_test_1, score_test] = predict(SVMModel, data_test);

output_test_1'
```

```
ans = 1×21
     1    -1     1    -1     1    -1     1    -1     1    -1     1    -1     1⋯
```

Comparison of two classifier

As shown above, the accuracy of RBF neural network classifier on training data is 96.28% and the accuracy of kernel SVM classifier on training data is 99.26%. Abviously, the performances of two classifier are good. The kernel SVM classifier performs a little bit better than RBF neural network classifier.

The accuracy of RBF neural network classifier on validation data is 96.97% and the accuracy of kernel SVM classifier on training data is 92.42%. So we can see the performances of RBF neural network classifier on training and validation set are almost the same. This means it has achieved a good generalization and especially no overfitting occurred. So it should also perform equally well on the provided test data. But kernel SVM classifier performs not so well on the validation set. The accuracy is much lower than that on training set. Maybe overfitting occurred during training phase. Comparing predictions of these two classifiers, there are only two different results. The difference ratio is about 9.52%. Because there are only 21 test samples, the result may not be so precise. At least, one conclusion can be roughly made by the result that the RBF neural network classifier is more suitable than kernel SVM classifier to do the classification on the provided data.

Appendix

All the functions used above:

```
% Shuffle the data
```

```
function data = shuffling(data)
    [rows,~] = size(data);
    data = data(randperm(rows), :);
end

% Divide data into training set, validation set and test set
function [train_data,valid_data,test_data]
    = data_divider(data,validation_ratio,test_ratio)
    [rows, ~] = size(data);
    validation_amount = round(rows * validation_ratio / 100);
    test_amount = round(rows * test_ratio / 100);

    data_index = 1 : rows;
    valid_index = randi([1, rows], 1, validation_amount);
    valid_data = data(valid_index, :);
    data_index = setdiff(data_index, valid_index);
    data = data(data_index, :);
    [new_rows, ~] = size(data);

    data_index = 1 : new_rows;
    test_index = randi([1, new_rows], 1 ,test_amount);
    test_data = data(test_index, :);
    data_index = setdiff(data_index, test_index);
    train_data = data(data_index, :);
end

% Generate distance matrix
function M = distance_matrix(n1, n2)
    n = n1 * n2;
    M = zeros(n, n);
    for i = 1 : n
        xi = floor(i / n2);
        yi = mod(i, n2);
        if yi > 0
            xi = xi + 1;
        end
        for j = 1 : n
            xj = floor(j / n2);
            yj = mod(j, n2);
            if yj > 0
                xj = xj + 1;
            end
            M(i, j) = sqrt((xj - xi)^2 + (yj - yi)^2);
        end
    end
end

% SOM
function w = SOM(data, rows, cols, learningRate1, learningRate2, iterLimit)
    % data ---- input matrix
    % rows, cols ---- the dimension of the 2D lattice
    % learningRate ---- the learning rate in two phases
    % iterLimit ---- the times of iteration
    [nSample, nDim] = size(data);
    nNeuron = rows * cols;

    % Generate the initial weight vectors
    w = randn(nNeuron, nDim);

    % Define the initial values for the time constant and  learning rate
    eta0 = learningRate1;
    sigma0 = sqrt((rows - 1)^2 + (cols - 1)^2) / 2;
    tau1 = 1000 / log(sigma0);

    % Generate the lateral distance matrix
    dist = distance_matrix(rows, cols);

    % Self-orgnizing phase
    for k = 1 : iterLimit
        % Calculate learning rate and width of the neighbourhood function
        eta1 = eta0 * exp(-k / 1000);
        sigma = sigma0 * exp(-k / tau1);

        % Randomly select a training sample
        row1 = randperm(nSample, 1);
        x1 = data(row1, :);

        % Compete and find the winning neuron
        d1 = zeros(nNeuron, 1);
        for i = 1 : nNeuron
            d1(i, 1) = (w(i, :) - x1) * (w(i, :) - x1)';
        end

        [~, index_win_1] = min(d1);

        % Update weight vectors of all neurons
        for i = 1 : nNeuron
            h1 = exp(-dist(i, index_win_1) / 2 / sigma^2);
            w(i, :) = w(i, :) + eta1 * h1 * (x1 - w(i, :));
        end
    end

    % Convergence phase
    % Set the learning rate to a small constant
    eta2 = learningRate2;

    % Repeat 500 * nNeuron times
    for k = 1 : 500 * nNeuron
```

```
        % Randomly select a sample
        row2 = randperm(nSample, 1);
        x2 = data(row2, :);

        % Compete and find the winning neuron
        d2 = zeros(nNeuron, 1);
        for i = 1 : nNeuron
            d2(i, 1) = (w(i, :) - x2) * (w(i, :) - x2)';
        end

        [~, index_win_2] = min(d2);

        % Update the weight vector of the wining neuron only
        h2 = 1;
        w(index_win_2, :) = w(index_win_2, :) + eta2 * h2 * (x2 - w(index_win_2, :));
    end
end

% To calculate Phi
function phi = calculate_Phi(data, center, sigma)
    [nSample, ~] = size(data);
    [nCenter, ~] = size(center);
    phi = zeros(nSample, nCenter);
    for i = 1 : nCenter
        c_mat = repmat(center(i, :), nSample, 1);
        column_i_of_phi = exp((1 / (2 * sigma^2)) * sqrt(sum((data - c_mat).^2, 2)));
        phi(:, i) = column_i_of_phi;
    end
end

% Using LSM to adjust / train the weights
function W = weights_regression(phi, label)
 W = (phi' * phi)^-1 * (phi' * label);
end
```