SCE301 Bio-computation Week 3 Lab , 2016

Further Practice with Perceptrons

Prepared by Bailing Zhang

Part 1:

Analyze the given program `PerceptronExample, which consists of following parts:`

(1) prepare some kind of toy data by generating 2 dimensional linearly separable data. After the following lines

```
mydata = rand(500,2);
% Separate the data into two classes
acceptindex = abs(mydata(:,1)-mydata(:,2))>0.012;
acceptindex = abs(mydata(:,1)-mydata(:,2))>0.012;
mydata = mydata(acceptindex,:); % data
[m n]=size(mydata);
```

you can take a look of the data distribution by using

```
scatter(mydata(:,1),mydata(:,2))
```

Then the next two lines divide the data into training and testing parts.

(2) Train the perceptron by calling the function `PerceptronTrn` with the prepared training data (x,y), which will return the connection weights, the bias, and the number of iteration;

(3) Test the trained Perceptron model with the testing data xt and yt, by calling another function `PerceptronTst`, which will return the testing error

(4) Display the two classes of data points with a separating line.

Exercises:

(1) Revise the program to calculate the Root-Mean-Square (RMS) error for every input data points and display the error curve (i.e., RMS vs. iteration);

(2) The Perceptron training function uses a learning rate 0.5 and a threshold 0.5. Change these two parameters, e.g., learning rate 0.1 and threshold 0. See the different results;

(3) There is a given data LS, which can be directly invoked by typing

```
load LS.mat
```

in the command window. Divide this data into two parts of training and testing, for example,

```
xtr=x(1:30,:);ytr=y(1:30);

xt=x(31:end,:);yt=y(31:end);
```

and then apply the PerecptronTrn and PerecptronTst. Discuss the obtained results and show the RMS. Is this data linearly separable?

Part 2

Another Perceptron network demo - classifying 3D patterns into 2 classes
The following matlab program sets up a Perceptron network having 3 input units and 1 output unit, to perform a 2-way classification. The m-file script initPerceptron.m initializes the weights and the training patterns. The m-file script runPerceptron.m calls the function trainPerceptron for each training pattern, and after 1 learning epoch (1 pass through the training pattern set) it plots the input patterns, the weight vector, and the classification boundary defined by the weight vector. The patterns displayed as asterisks have a target output of +1, so the output unit's weight vector should be within 90 degrees of these points to produce a positive weighted summed input; in other words, all of these points should be above the plane orthogonal to the weight vector. The patterns displayed as circles have a target output of -1, so the weight vector should be more than 90 degrees away from each of those points to produce a negative weighted summed input to the unit (negative dot product between each input vector and the weight vector).

Exercises:

Download the following Matlab code: Perceptron demo.
initPerceptron.m (initialization module)
plotPerceptronInput.m (plots the input patterns)
plotDecisionSurf.m (plot the weight vector, and the classification boundary defined by the weight vector)
trainPerceptron.m (function for implementing the learning)
runPerceptron.m (m-file; calls the plotting m-files and trainPerceptron)

1. Run the example: Look at the code in 'initPerceptron.m', 'runPerceptron.m' and 'trainPerceptron.m' to see how the weights and states are initialized, and how the weights are updated. Now do the following:

1. Execute the script 'initPerceptron, which initializes the weights and training patterns.
2. Run the script 'runPerceptron'. Re-size and move your matlab interpreter window if necessary so that the matlab figure window and matlab interpreter windows are both visible.
3. Repeatedly run the script 'runPerceptron' until the network has converged (weights no longer change). You can view the weight vector, and the associated classification boundary (desicion surface perpendicular to the weight vector) on the figure, and you can also see the numerical values of the weights printed out after each call to runPerceptron. It will probably take between 10 and 30 learning epochs (10-20 calls to runPerceptron) before the learning converges.

Repeat this entire process for several runs, using different random patterns and different random initial weights. For each run, re-initialize the Perceptron (initPerceptron), then re-train (runPerceptron) until convergence. Why does the convergence time differ from one run to the next?

2. Increase the difficulty of the classification problem by changing the variance of the random Gaussian function used to generate the training patterns. Specifically, in initPerceptron.m, change var from 0.1 to 0.3. Rerun your Perceptron a few more times on these data.

(1) Does the Perceptron converge more quickly or more slowly on these data?
(2) Does it always converge, each time you re-initialize the perceptron and data set, or does it sometimes fail to converge?
(3) How can you explain the convergence or lack of convergence by looking at the training patterns?

3.Train using general error correction learning (ECL) rather than the previous simplified Perceptron learning:

Make a copy of the function `trainPerceptron.m` and rename it `trainECL.m`, and modify it to implement general error correcting learning. The learning rule should now be:

```
weight_change_for_ith_output_jth_input = learning_rate *
(desired_output_i - actual_output_i) * input_j.
```

Or in vector notation, for one output unit's weight vector:

```
weight_change_for_ith_output = learning_rate *
(desired_output_i - actual_output_i) * input
```

Replace the call to 'trainPerceptron' in runPerceptron.m with a call to your own function. In trainECL, use a linear activation function (output equals weighted summed input) rather than the binary threshold activation. Test the error correction-trained neural network on the same patterns, with both low and high variance. How does the error correction learning model differ in terms of speed and smoothness/directness (i.e. does the weight vector wobble around as much) of learning on the easy version of the problem? How does the convergence compare (i.e. does it converge at all) on the hard version of the problem?

Use the toy artificial data introduced in Part 1, experiment with the **`trainPerceptron`** and **`trainECL`** functions for creating the models.