Homework 5: Neural Networks

Robert Grant
2009-10-16

1 The Assignment

In this homework, we were to implement both a linear classifier and a backpropagation-based classifier to classify handwritten digit images.

2 Submission Information

My implementation is contained in four main function files: `hw5TrainL.m`, `hw5TestL.m`, `hw5TrainB.m`, and `hw5TestB.m`, which are written to the specifications of the assignment. Additionally, the function files `hw5ReshapeImages.m`, `hw5VectorizeLabels.m`, `hw5UnvectorizeLabelsL.m`, and `hw5UnvectorizeLabelsB.m` are called by the main files to condition the input data. Finally, the script file `hw5Main.m` runs experiments and makes plots, calling the function file `hw5ErrorRate.m` to compute the final absolute error rate of classification. Please see the detailed comments in these files for usage, or query the functions using the standard MATLAB `help` command.

3 Implementation Details

All of my neural network implementations leverage the MATLAB neural network toolbox.

3.1 Linear Classifier

I implemented a linear classifier using a single layer of 10 perceptrons, each using a simple hard threshold transfer function (`hardlim`). 10 perceptrons were used to correspond to the 10 different digits we are to classify. Ideally, each perceptron learns to linearly separate one particular digit from the rest, and that perceptron outputs a 1 when this digit is presented while the rest output 0. To train this network correctly, I first had to vectorize the input target vectors. For example, the target 5 is represented as [0000010000], where each binary digit in that vector represents the output of a neuron (starting with a neuron looking for 0 and ending with a neuron looking for 9).

Before translating from the vector output back to an integer, two error conditions must be handled. First, multiple neurons can output a 1. In this case I choose one of those neurons randomly. Second, it may occur that no neurons output a 1. In this case I simply choose a random neuron (classify the digit randomly). The network is trained using the `learnp` function and `adapt` in MATLAB, which simply applies the Perceptron Learning Rule.

3.2 Backpropagation-Based Classifier

The backpropagation-based classifier required more experimentation and more input processing. First, I tried a single hidden layer of 20 neurons with `tansig` transfer functions, and a single-neuron output layer with a linear transfer function (`purelin`). This network took the integer training targets as inputs and output a real number, which was simply rounded to produce the desired integer output. The results weren't spectacular, only reaching an error rate of about 60%.

Next, I tried vectorizing the targets as in my linear classifier, and changing the output layer to a layer of 10 neurons with `tansig` transfer functions. This produces a vector of real numbers (each between 0 and 1) for its output, and I simply select the neuron with the maximum output as the one who has correctly classified the digit. The output is then transformed from a vector to an integer as in the linear classifier (but without the need for error handling, as there is always a maximum to select). The problem with this approach turned out to be execution time and memory consumption.

The default Levenberg-Marquardt (`trainlm`) training algorithm would not complete due to memory limitations (despite my adjustment of its `mem_reduc` parameter). Other provided training algorithms would not run out of memory, but they would not complete in a reasonable amount of time. These issues were due to the high dimensionality of the input vectors ($D = 28 \times 28 = 784$). I first tried some naive schemes for dimensionality reduction (using only a couple representative rows of the image), and though these would at least run, they did not produce reasonable error rates (error rates were greater than 70%).

Finally, I tried using Principle Component Analysis (PCA) on the input to reduce its dimensionality, using the MATLAB function `processpca`. After experimenting with the `maxfrac` parameter (which tunes how many principal components to eliminate) I was able to produce a reasonable result, which I explore in the next section. With this dimensionality reduction I was able to go back to using the Levenberg-Marquardt algorithm (which is the default and balances speed with efficacy, though it can possibly use a lot of memory).

4 Experiment

Using the perceptron-based linear classifier and backpropagation-based classifier (with PCA input dimensionality reduction) that I outlined in the previous section, I profiled the error rate of each classification algorithm versus the number of training passes used. In Figure 1, I show the backpropagation algorithm with `maxfrac` set to 5%, and in Figure 2, I show it set to 2%. This means that the principal components that contribute less that 5% or 2% of the variance, respectively, are eliminated. You can see that this makes a large difference in the both best error rate achieved, and in convergence behavior.

This experiment required that I add another parameter to both `hw5TrainL.m` and `hw5TrainB.m` to specify the number of training passes, but a default value is provided if the functions are called with two arguments as specified in the assignment.

5 Conclusions

This assignment was very instructive. My experiment indicates that though my linear classifier doesn't achieve quite as good best-classification-accuracy as my backpropagation-based classifier, it ran much faster (even without reducing input dimensionality), was less prone to overfit, and was much simpler to get working. The backpropagation algorithm was also very sensitive to the principle components used, as can be seen by comparing Figures 1 and 2. Additionally, this shows how well a simple perceptron-based linear classifier can

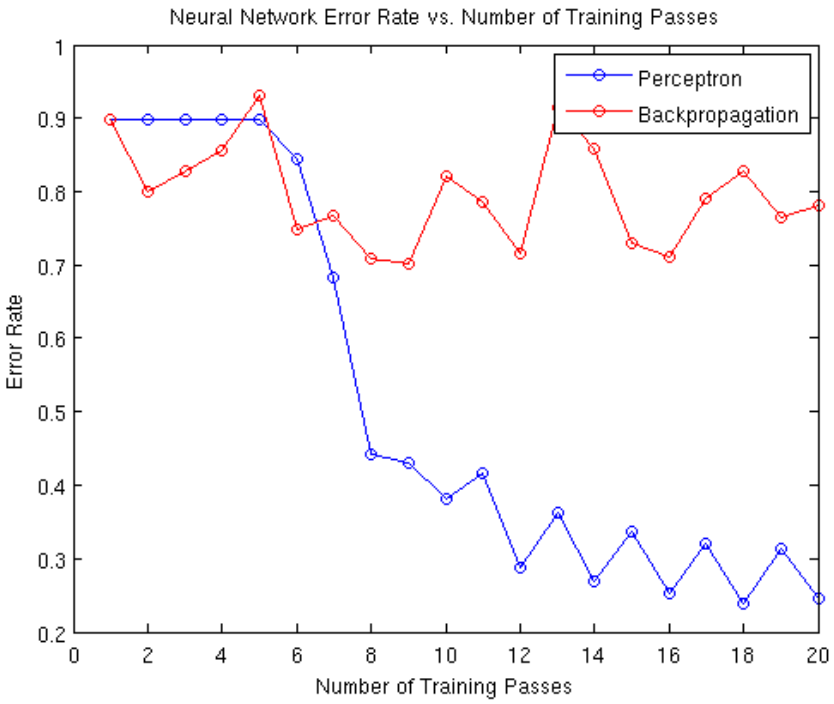

Figure 1: maxfrac = 5%

do on a very practical problem.

6 Acknowlegements

I discussed the efficacy of various neural net architectures with Birigi Tamersoy and Alex Loh.

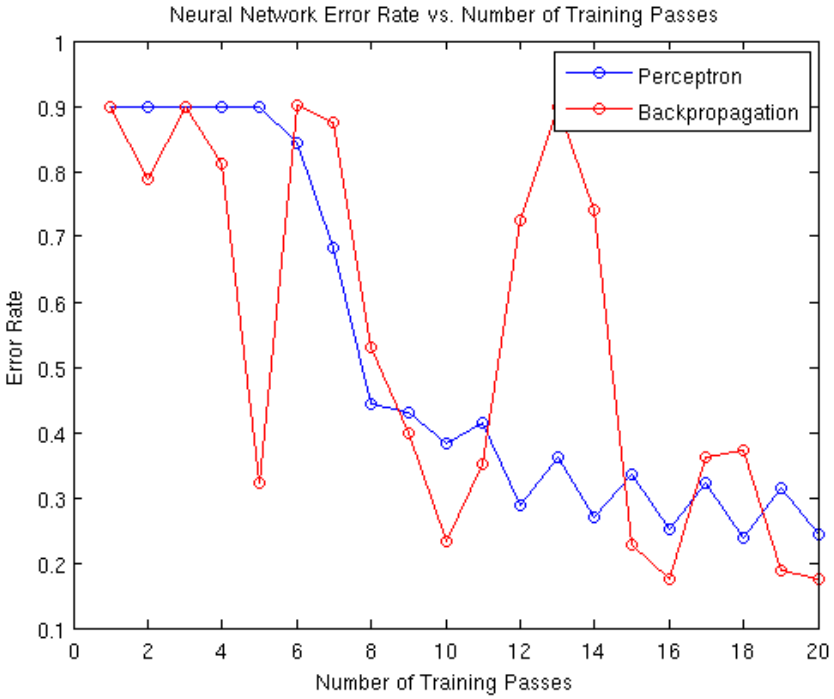

Figure 2: maxfrac = 2%