# Homework # 6

## 15-496/782: Artificial Neural Networks

## Dave Touretzky, Spring 2004

- Due April 28, 2004.
- You do not have to do this homework if you are doing a course project.
- The data file you need is in `/afs/cs/academic/class/15782-s04/matlab/digits`. This is binary data, so if you need to FTP it to another machine, be sure to do the transfer in binary mode.

## Overview

In this problem you will experiment with several architectures for recognizing handwritten digits. There are 10,000 images in the data set, stored in the file `t10k.mat`. See the README file for a description of where this data came from. The file contains two variables, `Images` and `Labels`. `Images` is a $10000 \times 28 \times 28$ matrix of pixel values between 0 and 255. `Labels` is a 10000 element vector containing integers between 0 and 9, indicating the correct class of each character.
**Warning:** this exercise uses a large dataset and requires many iterations for training. It is imperative that you write your code in good Matlab style, meaning most of the work is done using matrix operations, in order to achieve a reasonably fast training time. If you code your solution by writing lots of nested **for** loops as you would in C, you may find your simulation runs too slowly to be usable.

## Instructions

- Load the data file `t10k.mat` and display the first digit by calling `show_char(1,Images,Labels)`. Browse through the dataset a bit to get a feel for the variations in digit appearance.
- The arrays named Images and Labels are of type **uint8**, not **double**, so they require only 1 byte per element instead of the normal 8 bytes. However, it is not possible to do arithmetic on **uint8** values, so it is necessary to convert them to **double** after loading the file. Also, the $(28 \times 28)$ patterns must be reshaped into 784-element vectors in order to process multiple patterns in parallel in the normal way, using matrix multiplication. These functions have been taken care of for you, in the procedure named `setup_problem`, which you should call to initialize the classifiers you write. By setting the values of NTRAIN and NTEST, you'll be able to experiment with different size training and test sets.
- The procedure `getmeans` has been provided to compute the mean and standard deviation of each class for you. Use `setup_problem` to create training and test sets, then do `show_weights(getmeans(traindata,trainlabels))`. Now try setting NTRAIN to a small value, like 100, call `setup_problem` again, and see what the means look like.
- For this assignment, use the first 2000 examples in the dataset provided to you as your training set, and the next 1000 examples in the dataset as your test set.

1. Write a function `nearestMean` which implements a 1-nearest-neighbor classifier whose search space consists of the means of the 10 digit classes. Use your training set only to compute

the means. Measure the time it takes to "train" this classifier (compute the means), and the time it takes to run it on the test set. To time the execution of your code use the matlab command `cpuinfo` (type `help cpuinfo` at the matlab prompt).

**Hand in:** source listing, training set accuracy, test set accuracy, the training time and the testing time.

2. Write a DSM (decision surface mapping) classifier for the same problem. Recall that DSM is a variant of Kohonen's LVQ; the details of the algorithm are given in the class notes from the competitive learning lecture. Initialize your network with one prototype for each class. The initial weight vectors can be computed as the means of all the input patterns of that class, as in the first part of this homework, but you should normalize both your input patterns and your weight vectors so that they have unit magnitudes.

   DSM can add new prototypes if there is no sufficiently "close" prototype for a given input. As a criterion for closeness, use the magnitude of the dot product of the input pattern with the weight vector of the most active unit of the correct class. Set your closeness criterion so that the algorithm adds a "reasonable" number of prototypes; a value of 0.5 worked well for me, but feel free to experiment. At each time step your classifier should output a line like the following:

   ```
   Epoch  58, Train=94.2, Units= 17, Test=80.2
   ```

   The value of Units above is the current number of prototype units. Train your network for up to 300 epochs, you may create no more than 40 prototypes total. Display the weight vectors for your initial prototypes, and some of the additional prototypes your algorithm created. Also, at the conclusion of training, display a table of each prototype's number, its class, and the number of training patterns it captured.

   Many of your automatically-added prototypes will capture only a single pattern; they're handling noisy cases in the training set. But a few new prototypes will capture multiple patterns. Pick one of these and describe what that prototype is looking for.

   Measure the time it takes to train this classifier. You will need to also measure the time it takes to test it once, then multiply it by the number of training epochs and subtract that from the training time (since you are evaluating the classifier on the test set at every epoch).

   **Hand in:** source listing, sample training run, images of a few of your prototypes, table of prototype number/class/number of patterns captured, analysis of one learned prototype, the training time and the testing time.

3. Write a function `nearestNeighbor` which implements a 1-nearest-neighbor classifier whose search space consists of the whole training set. Measure the time it takes to "train" this classifier (hint: zero), and the time it takes to run it on the test set.

   **Hand in:** source listing, test set accuracy (the training set accuracy is 100%), the training time and the testing time.

4. List (1) the training set accuracy, (2) the test set accuracy, (3) the training time and (4) the testing time for all three algorithms in a table. Compare their performance. Also compare their time complexity in training and testing, and relative space requirements, and comment on any trade-offs that you notice.

   **Hand in:** your analysis.