



As in the NIST data set, the test set I've prepared was collected from different people than the training set. The two data sets are available at:

```
http://www.cs.colorado.edu/~grudic/data/digits_train
http://www.cs.colorado.edu/~grudic/data/digits_test
```

As you'll see when you look at the data, each example is preceded by a line with the word "train" or "test" followed by the example number, followed by the target digit class (0-9). The next 14 lines contain the 14 rows of the image. Each file contains 2500 examples, 250 of each digit.

## Methodology

You are to train and test your network on these data. You should evaluate the network's performance for different numbers of hidden units. I recommend you try a range between 5 and 40 hidden units, perhaps evaluating the network's performance for the following values: 5, 10, 15, 20, 30, 40. In measuring the test set performance, you should take the most active output unit to be the network's response. Score if this response is correct, and measure the % correct responses of the network. If you are performing cross validation, compute the average % correct over the different test sets. Your network should have 196 input units, one per pixel in the 14x14 array, and 10 output units, one per possible digit response.

You should normalize the pixel values before you feed the images to your neural network. The simplest sort of normalization would be to divide each pixel value by 10 so that the input units of your neural net lie in the range of 0-1. You can also normalize the inputs so that they have mean zero, and you could even normalize so that each input had a standard deviation of .5 (so that most of the inputs will range from -1 to +1).

Because the initial weights are chosen at random, and the initial weights have an effect on the training of the network as well as the performance on the test set, it would be best if you trained the network multiple times (for a given number of hidden units) using different random initial weights, and reported performance averaged over the different random initial weights. Replicating the experiment with different random initial weights should factor out some of the variability in performance attributable to the choice of initial weights and will give you an estimate of performance that is more directly related to the hypothesis complexity (the number of hidden units).

## Write up

I would like you to hand in a one page summary of your experiments on the hand-printed digits data base. This summary should include details of the procedure, a summary of your results (a graph or table showing average % correct on the test set as a function of the number of hidden units), and a copy of your code.

## Advice

As I will warn you in class, the code to implement a neural network is fairly straightforward. However, debugging the code is tricky; it is difficult to tell whether your implementation of back propagation is correct. You might want to test your code on a simple problem, such as one of the logical functions AND(A,B), OR(A,B), and XOR(A,B).

Setting parameters such as the number of training epochs and the learning rate requires much experimentation. To determine the number of training epochs and the learning rate, you might print out the error on the training set (the squared difference between the actual and target outputs for all output units and all training examples). If you do not see this number decreasing gradually over training epochs, you may have chosen a learning rate that is too large or too small; you should continue training until this error seems to reach asymptote.

As for the code itself, here's one suggestion about what functions you might want.

- `read_data`: read the training or test data from a file, also read the target output values from a file
- `init_weights`: randomly initialize the weights in your network
- `load_input`: copy the pattern from one training or testing example to the array that represents the activities of the input units
- `activate`: propagate activation from the input units to the hidden units, and from the hidden units to the output units
- `calc_error`: compute the squared error between the actual and target output values
- `adjust_weights`: compute the delta values for the output units, perform back propagation to compute the delta values for the hidden units, then adjust the weights from hidden-to-output, and the weights from input-to-hidden.
- `write_weights`: write the network weights to a file
- `read_weights`: read the network weights from a file

As we discussed in class, here is one set of arrays that could be used to represent the network state:

- `out1[i]`: the activity of input unit  $i$
- `out2[j]`: the activity of hidden unit  $j$
- `out3[k]`: the activity of output unit  $k$
- `w21[j][i]`: the connection weight to hidden unit  $j$  from input unit  $i$
- `w32[k][j]`: the connection weight to output unit  $k$  from hidden unit  $j$
- `b2[j]`: the bias weight for hidden unit  $j$
- `b3[k]`: the bias weight for output unit  $k$
- `delta3[k]`: the delta value for output unit  $k$  (used in `adjust_weights`)
- `delta2[j]`: the delta value for hidden unit  $j$  (used in `adjust_weights`)

## Bells and Whistles

If you want to use the test set as a *real* test set, put it aside and use the training set alone to determine the optimal number of hidden units. You can do this by splitting the training set into  $k$  partitions and performing  $k$ -fold cross validation.

You might consider a neural net architecture that includes direct connections from the inputs to the outputs.

If you are feeling really ambitious, you might consider writing a more general neural network simulator that allows you to design an arbitrary feedforward architecture consisting of *layers* of units and *connectivity* between layers. You could assign each layer an index number (starting with 1) and an indication of whether it is an input layer, hidden layer, or output layer. To ensure that the network is feedforward, you should make sure that layer  $n$  connects only to layers  $n+1$ ,  $n+2$ , ... You might read the definition of the architecture from a file. **I do not recommend that you take on this challenge unless you are a proficient C++ programmer.**

As a simpler but interesting alternative, you might try training your 3-layer network simulator on the voting or mushroom data bases, and compare performance of the neural network to the naive Bayes classifier, decision tree, or k-NN classifier. To make the comparison as fair as possible, you should use the same split of the data (training/testing) for each model.