CPS 170, Spring 2002
Programming Project #2 — Neural Nets

Due date for the programming assignment is Tuesday, April 23.
For this problem, your programming team (2 people, in different teams from the Othello project) will use Neural Networks to build a handwritten digit recognizer, which learns how to classify a digit using a neural network. The set of handwritten characters are supplied for you. They come from the post office in Buffalo, NY, where a classifier like yours would greatly speed up delivery. The goal of the network is to get 100% of the characters correctly classified as the appropriate digit, but the problem with real-world data is that they are noisy. Some of these digits could not even be pre-classified by people, so getting as close as possible to 100% is a more feasible goal. We will test two things: whether your program implements the right functionality, and the performance of your best network when applied to some unseen test data.

1 Simple Neural Network Digit Classifier

The input of your neural network consists of an image of a centered hand-written digit. Each image is a $14 \times 14$ matrix where each pixel's intensity is in the range $(0, 255)$, which we normalize to be in the range $(0, 1)$. Each image is fed as input to the neural network, which classifies it into one of the categories $0, \ldots, 9$.

You will use a neural network consisting of an input layer, a hidden layer, and an output layer. The input layer should have an input for each pixel in the digit image. The image data are read in as a vector of $14^2$ elements, in row-major format. The data set is further described in section 3.1. The number of units in the hidden layer is something that you are going to have to select experimentally. The number of output units should be 10, with the value of the $j$ output corresponding to the digit $j$. Most simply, we can hope that if the digit really is $j$, then the $j$th output of the neural network will be 1 and the others 0. Thus, for the digit 0, we would expect the output $[1, 0, 0, 0, 0, 0, 0, 0, 0, 0]$, for the digit 1 the output $[0, 1, 0, 0, 0, 0, 0, 0, 0, 0]$, and so on.

However, this approach turns out to be flawed because sigmoid units cannot produce the output values 0 or 1 given finite weights. If we attempt to train the neural network to fit the target values of exactly 0 and 1, our weight update function will force the weights to grow without bound. Thus, instead of 0 and 1 values, we will expect our neural network to produce the values of 0.1 and 0.9, which are achievable using a sigmoid function with finite weights. Thus, for example, $[0.1, 0.1, 0.9, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1]$ is the target vector for the digit 2.¹ Thus, we will train our neural network to try to achieve these output values for the corresponding digits.

To run your network on a sample image, the network should take the pixels of the image, feed each pixel into the corresponding input node, and compute the network output on this image (i.e., the values of the 10 output units). The actual classification is the index of the output unit with the largest value.

The mean-squared error, required for training the network, can be calculated based on the signals of all of the output nodes compared to the target output signals. More precisely, consider an image $x$ whose true class is $j$, and let $o_i$ be the value of the $i$th output unit when the network is run on $x$. Let $t_{j,i}$ be the target value of the $i$th output unit when the digit is actually $j$ (0.9

¹[Mitchell, p.115]
if \( j = i \) and 0.1 otherwise. Then the mean-squared error is
\[
\frac{1}{2} \sum_i (t_{j,i} - o_i)^2
\]

Training this structure consists of multiple iterations of the following cycle.

- Step through the training data set one instance at a time.
- For each instance, use your current network to classify it.
- Use the back-propagation algorithm to change the weights in the network appropriately, based on the actual output and the target class.

Repeat this cycle until you have reached some predetermined stopping point. Ways to choose this stopping point will be discussed later. Every few iterations of this cycle, you should stop and output the network into a file. This will allow you to test the accuracy of your network as the algorithm proceeds, and will allow you to restart your program from the middle if it crashes.

Note that we are also providing you with your own independent test set, which you will use to evaluate your algorithm. You must not use your test data to train the network.

2  Tasks and Deliverables

1. [20 points] **Forward Propagation.** Implement basic forward propagation. This requires modifying the ForwardProp routine in netmanager.c (see below). This should not involved a lot of coding if you read through the supporting functions and read the text carefully. Use the example outputs to test your program.

2. [25 points] **Backward Propagation.** Implement basic backward propagation. This requires modifying the BackProp routine in netmanager.c (see below). This should not involved a lot of coding if you read through the supporting functions and read the text carefully. Use the example outputs to test your program.

3. [25 points] **Varying Number of Hidden Units and Iterations.** Compare the accuracy obtained by neural network learning for different numbers of hidden units and iterations of training. You should experiment with 5 different network sizes between 10 and 300 hidden units: 10, 50, 100, 200, 300. For each network size, use five settings for the number of iterations: 5, 10, 15, 25, 50. Note that you don’t need to restart training when you’re comparing accuracy for different number of iterations for the same number of hidden units. Just output the network, say at 5 iterations, and then keep on training until 10, output the network, and so on.

Graph accuracy (percentage classified correctly) on both the training and test set. Use a separate graph for each of the five network sizes. Answer the following questions:

(a) How does changing the number of units in the hidden layer change the space of possible solutions?

(b) As we increase network size, do we always get monotonic improvement in accuracy for the training set? For the test set? Explain.

(c) As we increase the number of iterations, do we always get monotonic improvement in accuracy for the training set? For the test set? Explain.
(d) How does the “optimal” number of iterations change with the network size?

4. [25 points] Varying Amount of Training Data. Compare the accuracy of networks trained with different amounts of training data. Choose the settings for the number of hidden units and iterations that performs best from the previous question and use them as you vary the size of the training dataset. You can use the `save_digit_data` and `save_digit_labels` routines to generate subsets of training data to files or simply focus on a subset of the entire training set as you train. Graph accuracies (both training and test) as you vary the training set from 1000 to 10000: 1000, 2000, 4000, 8000, 10000. Does the accuracy on the training set always grow as we increase the amount of data? How about the accuracy on the test set? Explain.

5. [5 points] Extensions Implement one or more extensions to the basic neural network algorithm; we provide a couple of suggestions in Section 4.

Here is the list of items we expect from each team:

1. An implementation of a learning algorithm for a sigmoid neural network with a single hidden layer. You must turn in the executables `nn_train` and `nn_test`, and these must pass our tests (see below).

2. A single neural Network file `best.net` that you’ve selected as your best. We will test accuracy on a test set that you haven’t seen. Remember you are allowed to use both the training set (10k) and the test set (1k) to train your neural net that you submit to us. **This is the only purpose for which you are allowed to use the given test set as training data!**

3. The source for the code that you have written. Your code should be well written and readable.

4. A writeup which details your experiments and answers the questions above. Your writeup what is the network you used in `best.net`. Finally, if you’ve implemented extensions, the writeup should specify which extensions you’ve implemented and present experiments analyzing whether the extensions improve the performance.

Note: The test set that we will use to determine highest accuracy comes from the same publicly available database as your training and test sets, and is disjoint from both. It is possible for an enterprising team to download the entire database and train on that, improving their chances of having the best accuracy. Using any training data except for what we supply will be considered a violation of the Honor Code.

3 Data and Code

3.1 Data set

The data set comes from a large database (over 60k) of handwritten digits known as MNIST. There is a relevant webpage on the data at [http://yann.lecun.com](http://yann.lecun.com). We have created our own subset of this data by randomly sampling 10k of the cleaner training data and 1k of the cleaner testing data. The data sets are in the directory: [http://www.cs.duke.edu/education/courses/spring02/cps170/nn/](http://www.cs.duke.edu/education/courses/spring02/cps170/nn/). There should be 4 files:


- **training-10k-images.idx3** is your training set which contains the image data.
- **training-10k-labels.idx1** is your training set which contains the image labels (digits 0-9).
- **test-1k-images.idx3** is your test set images.
- **test-1k-labels.idx1** is your test set labels.

Don’t worry about having to learn the .idx file format. A friendly TA has already gone about and done this for you. All you need to know is that each digit is represented as an 14x14 array of pixel values which range from 0 to 1. You will access this array as a vector (in row-major format) with $14^2$ elements.

### 3.2 Code Infrastructure

We have provided most of the infrastructure code you will need. There is only one file, `netmanager.c`, that you will need to modify to complete the assignment. `reader.c` contains basic functions for reading and saving data from/to the .idx files described above. You are going to use these functions to read all digits into a local data structures, to allow for faster lookup. The only procedures you should ever have to use from `reader.c` are in the .h file: The code is found at [http://www.cs.duke.edu/education/courses/spring02/cps170/nn/](http://www.cs.duke.edu/education/courses/spring02/cps170/nn/). `netmanager.c` contains routines for reading and writing neural networks as well as placeholders for functionality you will implement.

The functions that you will need to modify are:

- **ForwardProp**: forward propagation — given an input to the neural network, compute the values of the output units (using the current weights). This routine should store any intermediate results required for the backward propagation phase.
- **BackProp**: backward propagation — after completing a forward propagation step, use the target outputs to adapt the weights in the network using the backprop algorithm.\(^2\)
- **Test**: Iterate through an entire dataset of test images and print out the output produced by the network.
- **Train**: Iterate through an entire dataset of training images and adapt the network weights to each image, one at a time.

In addition, there are two files, `main.test.c` and `main.train.c` which contain “main” functions. The `Makefile` provided links two executables, `nn_test` and `nn_train` from these files, respectively. We will test your programs using these two executables.

For testing basic forward propagation, we’ll use:

```
% nn_test neural_net_filename test_images_filename test_images_labels
```

The output should be tab delimited pairs, one per line, of predicted and actual labels for each image. A sample output would look like this:

```
0 0
5 2
1 1
3 3
```

\(^2\)**Note**: One standard bug in backprop is to use the new weights of one layer to adapt the weights for the previous one. This is incorrect, and will lead your code to have different behavior from ours.
An example output is in the code directory, in file
initial_weights_1.70.1.test.out.
We’ve provided a simple perl script to compute accuracy using the above output:
accuracy.pl.

For testing basic backward propagation, we’ll use:
nn_train hidden_units training_iterations training_images training_labels initial_weights
The initial weights you should use will be specified in the initial_weights file. The file will
have the following format:
Output node 0:
0.01
0.005
.
.
.
Output node 1:
.
.
.
.
The first line after each label indicates the threshold value for that output node, and the weights
from the hidden units to that output node follow. We use the same ordering for the weights for
each hidden unit, i.e., the kth weight in the lists for the different output units should all belong
to the same hidden unit. The output nodes are numbered based on the digit they are classifying,
0...9.
Your executable should load the weights specified, and initialize the weights from the input to
the hidden layer to 0. The program then trains for the specified number of iterations on the
provided training set and outputs the network in the initial_weights.saved.net file.
We’ve provided a simple perl script to generate random initial weight settings:
weights_generator.pl.
You can run it as follows:
% initial_weights_generator.pl # hidden units
The format of the net files is fairly simple. The first line contains three space delimited numbers:
the number of weights in each hidden node (which is the number of input nodes +1 for the
threshold 196 + 1 = 197), number of weights per output node (which is the number of hidden
nodes +1) and the number of output units. The weights follow. For example:
197 71 10
Hidden node 0:
0.01
0.005
.
.
.
Output node 0:
.
We’ve also provided you with example parameters and expected result to allow you to debug your code. We ran our training program, nn_train, with 70 hidden units for 1 iteration using the training data and initial_weights_1 in the code directory:

```
nn_train 70 1 training-10k-images.idx3 training-10k-labels.idx1 initial_weights_1
```

The output is in the file weights_1.70.1.saved.net.

We then ran our testing program, nn_test, using the resulting network on the test set:

```
nn_test initial_weights_1.70.1.saved.net test-1k-images.idx3 test-1k-labels.idx1
```

The resulting example output is in the code directory, in file 1.70.1.test.out.

### 3.3 Etiquette and Unix Tips

If you are running a long experiment, you should nice the processes, like this:

```
% nice +10 myprog parameters
```

Processor hogging will be considered a violation of class rules. If you have access to computing resources outside of acpub, we encourage you to use them, as long as your final code compiles and runs on Ultrasparc machines as well.

Some other Unix tips:

- Use the `keepticket` command to allow your processes to run for longer than several hours.
- Use the `nohup myprog parameters` command to allow your processes to run after you log out.
- Use `fflush(0)` to flush files to disk during long training runs. This ensures that your accuracies after each iteration are maintained even if the process dies prematurely.

### 4 Extensions

To get a top score, you will need to implement at least on extension that goes beyond the basics. This section contains a couple of possible additional features you can add to your neural network. This list is by no means exhaustive. Please feel free to explore the literature on neural networks (some pointers are in the bibliography) to enhance your understanding of these concepts or generate more additions that might be just as interesting as the features listed here. If you feel unsure about some feature you want to implement, please ask us.

If you want to try out more than one extension, we recommend that you experiment with each one in isolation at first. This reduces the risk of error and allows you to better evaluate the benefit of each technique by itself.

### Features

The performance of a neural network can be greatly influenced by the representation used for the inputs. The assignment calls for a simple grayscale bitmap, but this may not be the best
representatin to use for learning. Note, for example, that our eyes do a good bit of image processing before the signal is even sent to our brains. Experiment with adding new inputs or modifying the existing inputs to your neural network. For example, you may want to do some processing to detect or enhance edges. You might also try enhancing the contrast. Achieving better performance isn’t necessary to get full credit. However, a good explanation of your approach and thorough testing is.

**Error-Correcting Output Coding**

The problem of classifying inputs into 10 classes is much more difficult than binary (2 category) classification for most machine learning methods, including neural networks. Suppose we wanted to use binary classifiers for a K-class learning problem. We can partition our K classes into two disjoint subsets A and B (randomly or using prior knowledge) and train our classifier on data labeled using our new classes A and B. (We use 0 to denote a classification of A and 1 to denote a classification of B.) We do this L times, splitting K into different subsets A_l and B_l each time and obtaining a classifier h_l. For example, imagine classifying instances into four classes. We can, for example, define the following five partitions:

<table>
<thead>
<tr>
<th>A_l</th>
<th>B_l</th>
</tr>
</thead>
<tbody>
<tr>
<td>{1, 3}</td>
<td>{2, 4}</td>
</tr>
<tr>
<td>{2, 3}</td>
<td>{1, 4}</td>
</tr>
<tr>
<td>{1}</td>
<td>{2, 3, 4}</td>
</tr>
<tr>
<td>{1, 3, 4}</td>
<td>{2}</td>
</tr>
<tr>
<td>{3}</td>
<td>{1, 2, 4}</td>
</tr>
</tbody>
</table>

We now take each new point x and run it through every classifier. If all of them work perfectly, we can predict exactly what the output should be. For example, if the true class of x is 2, the output vector we get from the five classifiers should be (1, 0, 1, 1, 1); if the true class is 3, the output vector should (0, 0, 1, 0, 0). In practice, of course, our classifiers will not work perfectly. So, given a new data point x, how do we classify it?

The simplest way is to think of it by voting. We let each classifier h_l predict its class. Suppose h_l(x) = 0, then each class in A_l gets a vote, if h_l(x) = 1, then each class in B_l gets a vote. The class with largest number of votes is selected as the prediction of this collection of classifiers.

A more powerful approach is to view each class j as encoding a codeword C_j of length L. Thus, the code word for class 2 above is the vector we showed. Each classifier is trying to predict one bit of the codeword, i.e., classifier h_l attempts to predict the lth bit, but it might be wrong. The predictions of classifiers for a new datapoint x are combined into a L-bit string C, which usually will not be exactly like any of the code words C_j. However, we can find the codeword C_j which is closest (in Hamming distance, i.e., edit distance) to the C we get. We then classify x as being in class j.

Some experiments you can do for error-correcting output coding:

- Experiment with different ways of partitioning our 10 digits — by similarity, randomly, or using error-correcting algorithms.
- Vary the number of classifiers/bits in your encoding.

In any case, you should compare the accuracy you get to the accuracy of the naïve 10-way classifier in the standard neural net implementation.
Smart Network Structure

The basic neural network you have constructed does not take into account the *a priori* insights you have about digit recognition. For example, should some areas of pixels be weighted more than others because they are more important in classification? Should some clusters of pixels not be included at all? Is a fully-connected network appropriate for this type of problem, or can some of the connections between units be eliminated? Intuitions about these questions can lead you to hand-constructing a network which takes into account your *a priori* understanding of the problem.

One approach to constructing domain-specific network structures is to take advantage of similar features replicated through the input vector. We would like to train certain hidden units to correspond to certain features in the images. For example, in our digit training example being able to recognize specific curves and lines in certain orientations might be useful. Furthermore, it would be useful to be able to recognize the same feature in different parts of the image matrix. We can do this via weight-sharing, where we apply the same subnet to different parts of the image to look for the same types of specific features.

For example, in the figure below, we are trying to recognize an "8". We could structure the network so that we assign one hidden node for each 7x7 input matrix as shown below. Then we could train the network on the data, and share weights of the hidden units that we have set up to correspond to the same features. Note that we typically want to detect more than one type of feature over the image, so that we would have several sets of "feature nodes", where each set is trying to detect one particular feature (e.g., a vertical edge) using shared weights. Each hidden unit in the set is trying to detect the feature in a different part of the image. Note that the actual features detected by each set is not determined by you (the designer); rather, the training algorithm gradually homes in on a useful feature for this set of nodes.

![Figure 1: Weight Sharing.](image-url)

Note that in order to implement weight-sharing, you need to structure your network so that the inputs units correspond to the appropriate areas on the image and are connected to the proper hidden unit. Your network will no longer be fully connected and you might need to change your implementation of neural network so that you can modify the structure of the network.
CPS170 Programming Project 2

For an exciting example of weight sharing and feature detection refer to the papers on digit recognition by LeCun et al.

Some experiments you can do for weight-sharing and constructing problem-specific network structures:

- Construct more than one network which uses weight sharing and check each network's performance against a fully-connected network with the same number of hidden nodes.

- Look at the images of the digits that your network misclassified. If they lead you to hypothesize about the problems with your choices about which groups of pixels to group or which hidden unit layers to share, then restructure your network and compare the results to the previous network’s performance.

- Hand-construct more than one network based on some a priori understandings you have of the data, but do not implement weight-sharing. Describe your a priori reasoning so that the changes don’t seem arbitrary.

- Compare the weight-sharing and problem-specific networks to networks with the same number of hidden nodes using some of the testing ideas in section 2.

- What are the limitations of the hand-crafted models? What are the limitations of the original models? Use experiments to test some of your hypotheses.

Bibliography


Presents a wide variety of current techniques for generic machine learning.


Techniques on constructing neural networks for digit recognition. Includes testing and experimentation ideas.