A Neural Network Implementation of Optical Character Recognition

Introduction

In today’s world of high technology, there is a greater want to convert the analog into digital. Since the advent of digital scanners after the computer came onto the scene, there has been the want to convert books/text into digital media viewable over the internet and/or on a computer. This is where optical character recognition comes in handy.

What is Optical Character Recognition?

Optical Character Recognition (OCR) is the mechanical or electronic translation of images of handwritten, typewritten or printed text (usually captured by a scanner) into a machine-editable text. It is often used to convert paper books and documents into electronic files. “When one scans a paper page into a computer, it produces just an image file, a photo of the page. The computer cannot understand the letters on the page, so you cannot search for words or edit it and have the words re-wrap as you type, or change the font, as in a word processor. You would use OCR software to convert it into a text or word processor file so that you could do those things. The result is much more flexible and compact than the original page photo” (Wikipedia.org [http://en.wikipedia.org/wiki/Optical_character_recognition]).

OCR can get quite complicated. For instance, noise, distortion or skewing in the scanned image could disrupt the OCR algorithm to such an extent so as to make recognition impossible. This is why OCR is generally statistical in nature. This means that generally no OCR algorithm is fully suited to every possible scenario. For type-written books, a simpler OCR algorithm may best suite the digit recognition process while a much more complex OCR algorithm would be required to recognize digits which contain noise or distortion. For some forensic labs which analyze handwriting on, perhaps, broken scraps of paper, a much more complex OCR algorithm would be required than is implemented in this semester project.

Many Forms and Uses of OCR

OCR can take many different forms and has many different uses. For instance, a whole page of text could be scanned into a computer, and the OCR software could take as input this whole page of text and output a corresponding machine-editable font. The software could also take as input single characters or digits and output the corresponding character or digit. OCR also has uses in forgery detection. For instance, forgery detection could be used in the banking industry to detect forgeries on checks. Obviously this could save banks time and money because the bank could not waste time giving credit for forged checks. This could give banks the opportunity to contact the supposed signee to verify the owner of the account actually signed the check.

Final Project Write-Up

Digit Organization
For this project, I implemented a very simple nearest neighbor OCR algorithm. The training and test image sets are used to train and test the neural network, respectively. The neural network is trained on the training set and tested on the test set. There are 60,000 training images and 10,000 test images. The test and training sets are organized in the IDX file format, a simple format for vectors and multidimensional matrices of numerical types. Each image is 28 pixels wide by 28 pixels high, and each pixel is represented by a byte. A value of 0 represents white, and a value of 255 represents black. Values between 0 and 255 represent grayscale shades.

The files are organized at the byte level and are organized as follows:

The following is the organization of the training set label file. In my program, I begin reading from offset 8 in the training and test set label files.

```
TRAINING SET LABEL FILE (train-labels.idx1-ubyte):
```

<table>
<thead>
<tr>
<th>offset</th>
<th>type</th>
<th>value</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>32 bit integer</td>
<td>0000000000000000000000000000</td>
<td>magic number (must be first)</td>
</tr>
<tr>
<td>0004</td>
<td>32 bit integer</td>
<td>0000000000000000000000000000</td>
<td>number of items</td>
</tr>
<tr>
<td>0008</td>
<td>unsigned byte</td>
<td>??</td>
<td>label</td>
</tr>
<tr>
<td>0009</td>
<td>unsigned byte</td>
<td>??</td>
<td>label</td>
</tr>
<tr>
<td>.......</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>xxxx</td>
<td>unsigned byte</td>
<td>??</td>
<td>label</td>
</tr>
</tbody>
</table>

The labels values are 0 to 9.

The following is the organization of the training set file. In my program, I begin reading from offset 16 in the training and test set files.

```
TRAINING SET IMAGE FILE (train-images.idx3-ubyte):
```

<table>
<thead>
<tr>
<th>offset</th>
<th>type</th>
<th>value</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>32 bit integer</td>
<td>0000000000000000000000000000</td>
<td>magic number (must be first)</td>
</tr>
<tr>
<td>0004</td>
<td>32 bit integer</td>
<td>0000000000000000000000000000</td>
<td>number of images</td>
</tr>
<tr>
<td>0008</td>
<td>32 bit integer</td>
<td>28</td>
<td>number of rows</td>
</tr>
<tr>
<td>0012</td>
<td>32 bit integer</td>
<td>28</td>
<td>number of columns</td>
</tr>
<tr>
<td>0016</td>
<td>unsigned byte</td>
<td>??</td>
<td>pixel</td>
</tr>
<tr>
<td>0017</td>
<td>unsigned byte</td>
<td>??</td>
<td>pixel</td>
</tr>
<tr>
<td>.......</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>xxxx</td>
<td>unsigned byte</td>
<td>??</td>
<td>pixel</td>
</tr>
</tbody>
</table>

Pixels are organized row-wise. Pixel values are 0 to 255. 0 means background (white), 255 means foreground (black).

The following is the organization of the test set label file.

```
TEST SET LABEL FILE (t10k-labels.idx1-ubyte):
```

<table>
<thead>
<tr>
<th>offset</th>
<th>type</th>
<th>value</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>32 bit integer</td>
<td>0000000000000000000000000000</td>
<td>magic number (must be first)</td>
</tr>
<tr>
<td>0004</td>
<td>32 bit integer</td>
<td>10000</td>
<td>number of items</td>
</tr>
<tr>
<td>0008</td>
<td>unsigned byte</td>
<td>??</td>
<td>label</td>
</tr>
<tr>
<td>0009</td>
<td>unsigned byte</td>
<td>??</td>
<td>label</td>
</tr>
<tr>
<td>.......</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>xxxx</td>
<td>unsigned byte</td>
<td>??</td>
<td>label</td>
</tr>
</tbody>
</table>

The labels values are 0 to 9.

The following is the organization of the test set file.

```
TEST SET IMAGE FILE (t10k-images.idx3-ubyte):
```

<table>
<thead>
<tr>
<th>offset</th>
<th>type</th>
<th>value</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>32 bit integer</td>
<td>0000000000000000000000000000</td>
<td>magic number (must be first)</td>
</tr>
<tr>
<td>0004</td>
<td>32 bit integer</td>
<td>10000</td>
<td>number of images</td>
</tr>
<tr>
<td>0008</td>
<td>32 bit integer</td>
<td>28</td>
<td>number of rows</td>
</tr>
<tr>
<td>0012</td>
<td>32 bit integer</td>
<td>28</td>
<td>number of columns</td>
</tr>
<tr>
<td>0016</td>
<td>unsigned byte</td>
<td>??</td>
<td>pixel</td>
</tr>
<tr>
<td>0017</td>
<td>unsigned byte</td>
<td>??</td>
<td>pixel</td>
</tr>
<tr>
<td>.......</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>xxxx</td>
<td>unsigned byte</td>
<td>??</td>
<td>pixel</td>
</tr>
</tbody>
</table>

Pixels are organized row-wise. Pixel values are 0 to 255. 0 means background (white), 255 means foreground (black).
Here is a look at the first few training digits.

This image is representative of the digit 5.

This image is representative of the digit 0.

This image is representative of the digit 4.

This image is representative of the digit 1.

This image is representative of the digit 9.

This image is representative of the digit 2.
Training the Network

To train the network, I created a simple 10 x 28 x 28 matrix to hold the digits [0, 9] in a grid of 28 pixels by 28 pixels. Here's the C code I used to train the network:

```c
for (k = 0; k < 60000; k++) {
    if (k % 5000 == 0) {
        printf("%d images remaining to train\n", 60000 - k);
        fprintf(outputFP, "\n", 60000 - k);
    }
    imgToTrain = (int)extract('imgFP', imgOffset, 1);
    digitCounts[imgToTrain]++;
    imgOffset++;
    //      printf("\n\n");
    //      fprintf(outputFP, "\n\n");
}
for (i = 0; i < 28; i++) {
    for (j = 0; j < 28; j++) {
        if (val >= 10 && val <= 99) {
            fprintf(outputFP, "00%d ", val);
        }
        else if (val >= 100 && val <= 999) {
            fprintf(outputFP, "%03d ", val);
        }
        else {
            fprintf(outputFP, "%d ", val);
        }
    }
    fprintf(outputFP, "\n");
}
```

Lines 1 – 31 walk through the training set images and training set labels. I print out a message to the user in lines 2-4 every 5000 trained images to let the user know the program is actually doing work. In line 6, I extract the label of the image to be trained and increment the offset in the training set label file in line 9. I also increment `digitCounts` in line 11 for the corresponding image to keep track of how many images I’ve trained the network on. Next, in lines 10 – 28 I grab the next 28 x 28 = 784 values from the training set file and add them to the previous values of the training image in line 13. The if-else-if block in lines 15 – 24 take care of formatting and lines 26 & 27 and 29 & 30 take care of line breaks in output.txt. This essentially takes each new training digit `k` and directly overlays its image on top of all the previous training data for digit `k` and adds the current value to the previous values. As an example, consider the following toy example:

Training image “1”:

```
0 0 0 0 0 0 0 3 0 0 0 0 0 0 6 0 0
0 0 2 8 0 0 0 8 5 0 0 0 0 6 8 0
0 0 7 7 0 0 0 9 3 0 0 0 7 8 0 0
0 0 3 6 0 0 0 2 6 0 0 3 8 0 0 0
0 0 8 9 0 0 0 3 9 0 0 0 7 0 0 0
```
after adding the corresponding digits in each of the training images.

After the for loop ends at line 31, the next set of for loops takes care of “squashing” the data in the neural network, essentially averaging all the values in the network by dividing each digit’s neural network representation by the total number of training digits encountered for that digit. Lines 32 – 38, specifically line 35, takes care of dividing each pixel position in the neural network by the total number of digits encountered.

Continuing with the example above, the neural network would become the following:

```
0.0 0.0 1.0 2.0 0.0 0.0
0.0 0.0 3.3 6.6 2.6 0.0
0.0 0.0 7.6 9.0 0.0 0.0
0.0 1.0 6.5 4.0 0.0 0.0
0.0 0.0 9.0 9.0 0.0 0.0
0.0 0.0 0.0 2.0 0.0 0.0
```

After this process is complete, training of the neural network is complete.

**Analyzing Test Images Against the Neural Network**

The code for analyzing a test image against the trained neural network is as follows:
1. for (k = 0; k < 10000; k++) {
2.     if (& k % 1000 == 0) {
3.         printf("%d images remaining to test on", 10000 - k);
4.         fprintf(outputFP, "%d images remaining to test on", 10000 - k);
5.     }
6.     trueVal = (int)extractlblFP, lblOffset, 1);
7.     //printf("Analyzing the image:
8.     fprintf(outputFP, "Analyzing the image:
9.     lblOffset++;
10. for (i = 0; i < 28; i++) {
11.     for (j = 0; j < 28; j++) {
12.         val = (int)extract(imgFP, imgOffset, 28, 1);
13.         testImage[i][j] = val;
14.         if (val >= 0 && val <= 9) {
15.             //printf("00%d ", val);
16.             fprintf(outputFP, "00%d ", val);
17.         } else if (val >= 10 && val <= 99) {
18.             //printf("0%d ", val);
19.             fprintf(outputFP, "0%d ", val);
20.         } else {
21.             //printf("%d ", val);
22.             fprintf(outputFP, "%d ", val);
23.         }
24.         imgOffset++;
25.     } //printf("n");
26.     fprintf(outputFP, "n");
27. }
28. // compare the test image against the training images
29. for (i = 0, i < 10; i++) {
30.     for (m = 0; m < 28; m++) {
31.         for (n = 0; n < 28, n++) {
32.             digitStats[i] += pow(imageNetwork[i][m][n] - (double)testImage[m][n], 2.0);
33.         }
34.     }
35. }
36. // make a guess
37. int firstGuessCorrect = 0, secondGuessCorrect = 0;
38. int minIndex = 0;
39. double minVal = DBL_MAX;
40. for (i = 0, i < 10, i++) {
41.     if (minVal > digitStats[i]) {
42.         minIndex = i;
43.         minVal = digitStats[i];
44.     }
45. }
46. firstGuess = minIndex;
47. //printf("First guess: %d", firstGuess);
48. fprintf(outputFP, "First guess: %d", firstGuess);
49. if (firstGuess == trueVal) {
50.     numCorrectFirstGuesses++;
51. } else {
52.     numWrongFirstGuesses++;
53. }
54. minVal = DBL_MAX;
55. if (firstGuessCorrect == 0) {
56.     for (i = 0, i < 10; i++) {
57.         if (i == firstGuess) {
58.             continue;
59.         }
60.     } if (minVal > digitStats[i]) {
61.         minIndex = i;
62.         minVal = digitStats[i];
63.     }
64.     }
65. }
66. secondGuess = minIndex;
67. //printf("Second guess: %d", secondGuess);
68. fprintf(outputFP, "Second guess: %d", secondGuess);
69. if (secondGuess == trueVal) {
70.     numCorrectSecondGuesses++;
71. } else {
72.     numWrongSecondGuesses++;
73. }
74. minVal = DBL_MAX;
75. if (secondGuessCorrect == 0) {
76.     for (i = 0, i < 10; i++) {
77.         if (i == secondGuess) {
78.             continue;
79.         }
80.         if (minVal > digitStats[i]) {
81.             minIndex = i;
82.             minVal = digitStats[i];
83.         }
84.     }
85.     }
86. }
87. thirdGuess = minIndex;
88. //printf("Third guess: %d", thirdGuess);
89. }
90. }
Lines 1 – 105 analyze the 10,000 test images and compare each test image to each of the trained neural network digits. Lines 2 – 5 print out a message after every 1000 images have been tested to indicate to the user the program is actually performing some work. Line 6 extracts from the training set label file the actual value of the test image this OCR algorithm is trying to guess. Lines 10 – 28 actually extract the test image from the test set image file and save it in testImage in line 13. Lines 30 – 36, specifically line 33, calculates the standard deviation of the test image values from the trained neural network values and keeps track of the value in the variable digitStats. After line 36, digitStats[k] contains the standard deviation of the pixel values of digit k from the test image. My algorithm assumes the minimum value held in the whole array digitStats after line 36 corresponds to the actual value of the digit. Therefore, if digitStats[k] contains the minimum value in digitStats, then the test image is guessed to be digit k. Lines 37 – 55 make a first guess. If this first guess is incorrect, lines 56 – 77 make a second guess. If the first guess and the second guess are incorrect, lines 78 – 98 make a third guess. The program then prints the actual digit. Lines 102 – 104 reset digitStats[i] to zero to hold the statistics for the next test image.

OCR Results

Surprisingly the program was able to correctly determine the digit displayed in 82% of the images.

Results:
First guess identified 8203 of 10000, that is 82.030000%.
Second guess identified 967 of 1797, that is 53.811909%.
Third guess identified 383 of 830, that is 46.144578%.
Overall identified 9553 of 10000, that is 95.530000%.

Conclusion

While the nearest neighbor approach to digit recognition is not fool-proof, it is easy to see that this very simple and naïve implementation could be used as the basis for other OCR work. Extensions of this work could be to add the capability to feed the program pages of text so that the program must do preprocessing work to single out characters.

References