Building A Digit Recognition System

Jack Cohen, 12035750
CP467, Physics and Computer Science Department, Wilfrid Laurier University

1. Abstract:

An initial implementation for a digit recognizing OCR system is introduced. The system uses a machine learning algorithm whereby it is fed a collection of sample images which are compiled into digit data that is used to compare to the digit image to be recognized. The system uses a collection of features (including multiple zoning grids, pixel density histograms, and aspect ratio) as well as weighted probabilities representing the effectiveness of the feature. The systems allows for the user to either load an image of a digit or draw their own on screen. The system then attempts to recognize the digit and then saves the image to the database. Therefore the program is always learning. The final overall recognition rate was 81.2%.

---

2. Introduction:

Optical Character Recognition (OCR) systems (specifically digit recognition in this case) have been a key element in the rise of the information age since the first OCR system was created.

They are a very important aspect of image processing and will continue to be important especially in certain areas like computer security and UX for the foreseeable future. Therefore it is imperative to have OCR systems with very good recognition rates.

However, implementing an OCR system is no easy task and obtaining a high recognition rate can be very difficult.

In this paper I will discuss a program that includes a digit recognizing OCR system as well as other image processing methods.

The program is written in Python and is mainly interacted with the command-line although GUI draw functionality is included to allow for user inputted digits.

I decided to write the program in Python due to the easy readability of the language and the strong community support that it has.

To read images I used a

library called the PIL (Python Imaging Library) which allowed me to easily obtain important information about images such as the colour depth, dimensions, and the image matrix representation.

---

3. Image Processing (Phase 1):

(a) Introduction: Techniques such as convolution and thinning are also important aspects of image processing. Not only do they allow for users such as photographers to transform images to their liking, but they also allow for developers to improve their OCR systems.

For example, a digit may be easier to recognize if the image of the digit is first stripped of noise, sharpened, and/or thinned. Although, I did not add thinning to my OCR system (the reason of which will be discussed shortly), it can be an especially useful method to normalize images of digits and other characters.

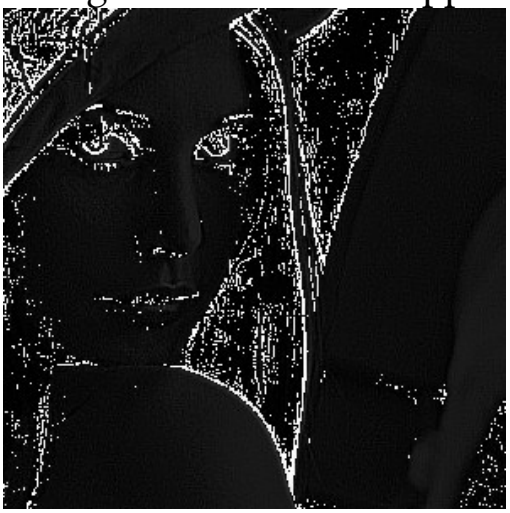
(b) Filters: The first feature I implemented was a convolution algorithm. Due to the facts that the program was written in Python and that a convolution algorithm requires nested for loops means that this function ran very slow. Although it worked perfectly, I decided to test my filter kernels using PIL's built in convolution function which was much faster.

In the program I only included 7 filters, however more can always be added very easily by adding the filters to the kernels file. I chose to include low-pass filters, high-pass filters, edge-detection and emboss filters, as well as an identity matrix to use for testing purposes (some of which are shown below).

Original Image:

With slight blurring filter applied:

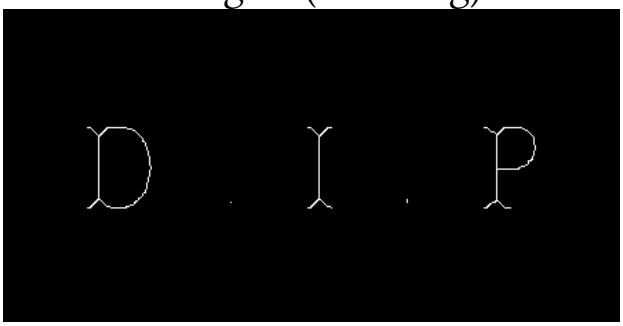
With edge detection filter applied:



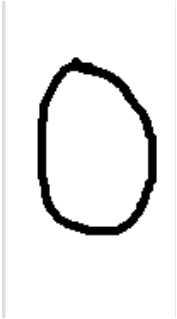
(c) Thinning: The next feature I implemented was a ZS thinning algorithm. Using C code of the algorithm, I converted it line by line into Python. Although this implementation was fast and worked perfectly for some images, it did not work at all for other images and the thinned image would look nothing like the original (both are shown below).

Original Image 1:

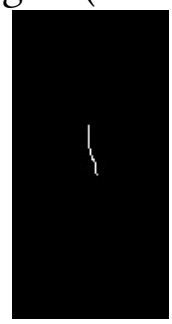
Thinned Image 1 (Working):



Original Image 2:



Thinned Image 2 (Not Working):



As you can see, the thinning worked perfectly on the first image but did not seem to work at all on the image where the colours are inverted. I believe this error was due to a fault in the way I was converting the images to matrices and so I intend to look into that in the future.

The inconsistency in this thinning functionality is the main reason that I decided to leave it out of my OCR program (as I mentioned above). I didn't want a non-perfect thinning algorithm potentially affecting my OCR recognition rate.

In the future I intend to fix my version of the ZS thinning algorithm so that I can include it into my OCR system and use it to help with stroke extraction (a feature I wanted to implement but was unable to without fixing my thinning algorithm.

---

4. OCR System (Phases 2 and 3):

(a) Introduction: The final feature of this program also happens to be the most important and difficult one to implement. As I mentioned in the introduction for this paper, building a good OCR system is no easy task so I decided to focus only on digit recognition. This way, I am working with less characters so the program has a higher chance of recognition.

This makes implementing the system easier and we can still easily convert it to a character recognition system by adding images of each character to the database and compiling those images during the machine-learning function as well.

I will now go into detail about the features I extracted to actually perform the digit recognition. However before I do, I would like to note that I started testing each feature using only 10 sample images for each digit (so 100 images total) and then grew that number to around 50 sample images for each digit (500 images total). I would also like to note that I included images of all the different ways the people usually draw each number (1 vs |. 7 vs 7̶, etc.). The recognition rates that I discuss below are obtained for each number by taking the average of 25 tests for each number.

(b) Aspect Ratio: The first feature I extracted was a simple one, although I found it to be a very good place to start my program.

Extracting the digit's aspect ratio was very easy. I made function to find the left, top, right, and bottom endpoints of the actual digit (not empty space) and then I simply subtracted the values to obtain the width and height. To compare the average aspect ratio of the images in the database to the inputted image to be recognized I simply chose the recognized digit using this method to be the one with the smallest difference in aspect ratio.

This feature proved to be the least effective (as expected) with an initial overall recognition rate of 26% (using 10 sample images for each digit)

Digit	Initial Recognition % (Avg of 25 trials)
0	24% (6/25)
1	52% (13/25)
2	8% (2/25)
3	28% (7/25)
4	12% (3/25)
5	28% (7/25)
6	20% (5/25)
7	36% (9/25)
8	28% (7/25)
9	24% (6/25)

and a final overall recognition rate of 36.8% (using 50 sample images for each digit).

Digit	Final Recognition % (Avg of 25 trials)
0	36% (9/25)
1	56% (14/25)
2	28% (7/25)
3	32% (8/25)
4	28% (7/25)
5	20% (5/25)
6	32% (8/25)
7	56% (14/25)
8	44% (11/25)
9	36% (9/25)

(c) Pixel Density Histograms: The second and third features I implemented were vertical and horizontal pixel density histograms.

Obtaining the actual histograms was also easy. It only required looping through the image and keeping track of the number of pixels in each row and column as arrays.

Actually comparing the histograms of the sample images to the inputted image proved to be a much more difficult task then it was for the aspect ratio. For the

database I decided to find the average number of pixels in each row and column for all the sample images of each digit and then compare these to the number of pixels in each row and column for the input image. I then chose the digit recognized using this method to be the one with the smallest difference in both vertical and horizontal pixel densities.

This feature proved to be very effective when combined with the aspect ratio feature, raising the initial overall recognition rate to 50.4%

Digit	Initial Recognition % (Avg of 25 trials)
0	40% (10/25)
1	64% (16/25)
2	52% (13/25)
3	60% (15/25)
4	32% (8/25)
5	52% (13/25)
6	44% (11/25)
7	68% (17/25)
8	26% (9/25)
9	56% (14/25)

and the final overall recognition rate to 64%.

Digit	Final Recognition % (Avg of 25 trials)
0	60% (15/25)
1	84% (21/25)
2	60% (15/25)
3	64% (16/25)
4	44% (11/25)
5	52% (13/25)
6	56% (14/25)
7	88% (22/25)
8	76% (19/25)
9	56% (14/25)

Adding the vertical and horizontal histograms thus almost doubled the final recognition rate using just aspect ratio.

(d) Zoning: The final features I extracted were 1x5, 5x1, 20x10, 40x20, and 50x25 zoning grids.

Obtaining the grids for each image was a bit more difficult than for the histograms because I added the ability to add any MxN grid to the image data. It was similar to the method of obtaining the histograms, however it included an extra for loop which made it much slower. My method to compare the data from the sample images to the inputted image was also similar to the method I used for histograms.

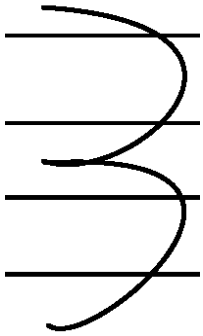
I chose the five grid configurations that I did for two main reasons.

The larger grids with more zones I used for the overall precision of the digits in general.

More zones in each grid allow for more precise zoning.

The 5x1 and 1x5 grids were used to increase the chance of recognizing specific numbers with distinct horizontal and vertical strokes. For example, most "3"s have 3 distinct horizontal lines and so the grid with 5 rows would help to zone that digit.

5x1 grid helping to recognize a "3":



Again, as expected, the zoning grids proved to increase my recognition rate quite drastically. My initial overall recognition rate jumped to 76.8%

Digit	Initial Recognition % (Avg of 25 trials)
0	60% (15/25)
1	84% (21/25)
2	80% (20/25)
3	88% (22/25)
4	68% (17/25)
5	76% (19/25)
6	72% (18/25)
7	88% (22/25)
8	88% (22/25)
9	64% (16/25)

and my final overall recognition rate jumped to 81.2%.

Digit	Final Recognition % (Avg of 25 trials)
0	80% (20/25)
1	96% (24/25)
2	84% (21/25)
3	88% (22/25)
4	76% (19/25)
5	76% (19/25)
6	68% (17/25)
7	96% (24/25)
8	88% (22/25)
9	60% (15/25)

As I predicted, the digits that have more distinct horizontal and vertical lines such as "1"s, "3"s, and "7"s proved to have very high recognition rates due to the specific 1x5 and 5x1 grids that I added.

(e) Notes on my OCR system: I used the above recognition rates to determine the effectiveness of each feature. Based off of the recognition rates above, I weighted the more effective methods (zoning and histograms) higher than the

others (aspect ratio).

So to recognize digits, each feature returns a digit that was the best guess for that feature and image and I put that digit into a vector. Then I insert the digit for each feature into the same vector a specific number of times based on the weight of that feature. The digit recognized will be the mode of this new vector.

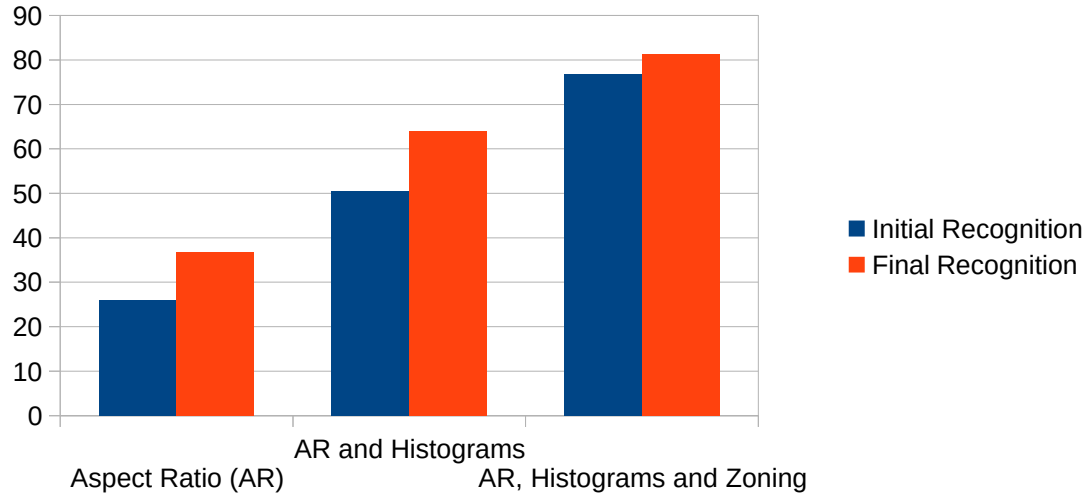
Another note: I tested the OCR system using only my handwriting, therefore the recognition rate is much better for my handwriting than it would be for others. I am sure that if other people were to input images of digits in their own handwriting then the recognition rates would initially fall. This is why I added the function to save any inputted digit images to the database. This way the program is always learning regardless of if it correctly recognizes a digit or fails to do so.

A final note: As I mentioned above, although Python is easy to read and quick to write, it is not nearly as fast as a language like C would have been and therefore extracting all the features from the images in the database takes a very long time. In the future I plan to convert this program from Python into a faster language.

---

4. Results:

(a) Recognition Rate vs # of Features:



As you can see, adding more and more features will increase the recognition rate like a logarithmic function; it starts easy to increase but the higher it gets, the harder it is to raise.

---

5. Conclusion: I believe that for an initial implementation of an OCR system, this was a very successful one. I consider a final recognition rate of above 80% to be very good for only a few months work and with more time I think that I definitely could have increased my recognition rate even more. Two features I would have included with more time are stroke detection and adding thinning to my OCR system.

Some Program Screenshots: