18-551, Spring 2003
Group 5, Final Report

Write-On!
Character Recognition

Jihoon Choi (jihoon@andrew.cmu.edu)
Peter Schmidt (pos@andrew.cmu.edu)
Isaac Shum (iqs@andrew.cmu.edu)

TABLE OF CONTENTS

1.0 INTRODUCTION

Most commercial Optical Character Recognition products claim to have exceptional accuracy while retaining the format of the document. They also have many other options such as spell checking and PDF conversion. However, these OCR systems require a great deal of memory and processing power and as such are not suitable for embedded applications, which have minimal resources. Embedded OCR systems currently on the market such as those found in PDAs only work in real-time, often by implementing stroke-based or velocity-based recognition. The purpose of our project was to create a low memory non-real-time OCR system that could be embedded in a scanner or other device.

2.0 PRIOR 551 WORK

In Spring 2002, Group 8 did a similar passive OCR project called "Automatic Machine Reading of Text". Their primary recognition algorithm used Fourier Descriptors. The system had several problems, including inability to distinguish between rotationally similar letters and letters with a vertical axis of symmetry. They also had significant problems segmenting and detecting spaces in the input. Our segmentation, space detection and recognition algorithms are all different. In addition, our system also implements capital letter detection and context recognition, neither of which was present in the project in Spring 2002.

3. ALGORITHMS

There are five commands that can be given to the system:

- Reset system and delete current training set
- Take a bitmap and add it to the lowercase training set
- Take a bitmap and add it to the uppercase training set
- Take a bitmap and do OCR without space detection and context processing (this is referred to as simple OCR).
- Take a bitmap and do OCR with space detection and context processing (this is referred to as full OCR).

The algorithm the system uses varies with the command, but in most cases the algorithms are very similar. The bitmaps need to be formatted a specific way for each of the commands:

- For a lowercase training set, the bitmap must be the letters 'a'-'z' in alphabetical order
- For an uppercase training set, the bitmap must be the letters 'A'-'Z' in alphabetical order
- For simple OCR, the bitmap can be any string of letters
- For full OCR, the bitmap can be a word or several words separated by spaces
- For all commands, the bitmap needs to:
  - Be 8-bit grayscale
  - Consist of black letters on a white background
  - Consist of one line of text
  - Have one-pixel (minimum) spacing in-between all letters
  - Have all letters in a generally horizontal fashion (no letters on their sides)

See figures 3.1, 3.2 and 3.3 for details.

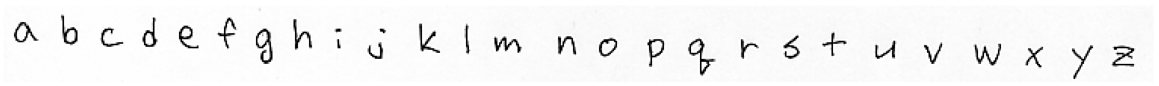


Figure 3.1 – A valid lowercase training set

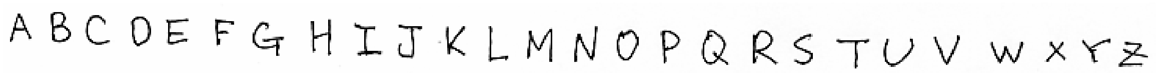


Figure 3.2 – A valid uppercase training set

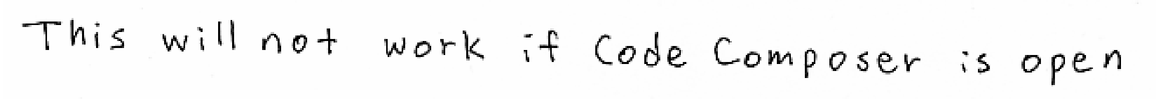


Figure 3.3 – A valid bitmap for OCR

Note figures 3.1, 3.2 and 3.3 all represent valid inputs for simple OCR. Only figure 3.3 is a valid input for OCR – the others would produce bizarre results if space detection and context processing were done on them since “abcdefghijklmnopqrstuvwxyz” is not a valid English word.

3.1 PC Algorithms

The PC does very little in our implementation. The PC is in an endless loop as follows:

1. Wait for a command
2. Transmit command
3. Transmit bitmap x, y sizes if command is not “reset”
4. Transmit bitmap (bitmap header is stripped off so only pixel data is transmitted)
5. Get OCR result if command is “full OCR” or “simple OCR”
6. Repeat…

The bitmap is loaded and the pixel data and x, y sizes obtained using the Windows library. All data is sent to the EVM using synchronous FIFO transmissions. The transmission of the data takes very little time relative to the other algorithms so asynchronous transmissions provided little speed enhancement at the cost of extra code.

The pixel data is sent in 20 byte packets (bigger packet sizes resulted in transmission errors). This also assures us that all transmissions are 4-byte aligned (a requirement of the FIFO). The OCR result is limited to 100 characters (100 characters are always sent regardless of the actual length of the result). This simplifies the programming a great deal and removes the need for an additional transmission from the EVM to the PC telling the PC the length of the result.

3.2 EVM Algorithms

The EVM does almost all of the work in our implementation. We did this since we wanted to design a system that could be embedded in a scanner or other device.

The EVM waits for a command and then takes appropriate action. If the command is “Reset”, the EVM simply deletes the training set it has stored in memory (52 32x32 images consisting of the averages of all the training sets submitted). For all other commands, the EVM receives a bitmap and processes it as follows:

3.2.1 Segmentation

The segmentation algorithm splits the bitmap into smaller bitmaps consisting of one letter each. It also detects spaces if the command is full OCR. The algorithm loops over all pixels in the image. If the pixel isn’t “white” and hasn’t yet been assigned to something referred to as a group, the algorithm recursively assigns the pixel and all adjacent non-"white" pixels to a so-called group. All pixels in the same group are considered to be part of the same letter. Groups with fewer pixels than a predetermined threshold are thrown away (they are usually due to scanner noise). We determined this threshold to be 4 pixels.

"White" for purposes of this algorithm is any pixel with luminance above a predetermined threshold. We determined the threshold to be 200 (out of 255).

At this point vertically overlapping groups are merged into one group. This prevents the bottom of an 'i' or 'j' and the dot on top from being considered separate letters. This is also why a one-pixel (minimum) spacing in-between all letters is required – this prevents the letters from being merged by this part of the algorithm.

Figure 3.4 shows two letters that do not have one pixel of white space in-between them (there is a one-pixel gray bridge between them). They will be treated as a single letter by the segmenter. The sentence these appear in is #2 in section 5.0. Refer to that section to see how the OCR algorithm misidentified them.

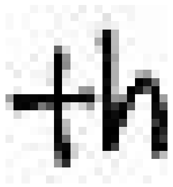


Figure 3.4 – These letters do not have one pixel of white space in-between them

Next, the algorithm determines the horizontal spacing between the smallest bounding rectangles of each pair of adjacent groups. The average horizontal spacing between all pairs of adjacent groups is computed and adjacent groups with horizontal spacing greater than a predetermined threshold times this number are considered to have a space in-between them. This step is skipped unless full OCR is being performed.

Lastly, the algorithm copies the pixels in every group (hereafter referred to as letters) into their own 32x32 buffers. As the pixels are copied a bilinear resize is applied to guarantee a 32x32 result. The resize allows us to guarantee that all comparisons are done between letters of equal size. The result is zero-padded so the aspect ratio of the original letter is maintained. In the process of being zero-padded the letter is centered as well so slightly taller and slightly wider letters have identical horizontal and vertical centers with respect to each other.

What happens next depends on the command the EVM is currently executing.

3.2.2 Training Set Processing

If the bitmap represents a training set, the EVM checks to ensure that there are 26 letters (groups) in the segmented input. If this requirement isn't met, the input is thrown away and the EVM waits for the next command.

The EVM's "comparison set" consists of 52 32x32 images (one 32z32 image for every uppercase letter followed by one for every lowercase letter, in alphabetical order). Each letter in the comparison set represents the pixel-wise average of all instances of that letter in all training sets submitted so far.

The EVM assumes that the segmented input of a training set is in alphabetical order, and so it just begins at the appropriate image in the comparison set (‘A’ or ‘a’ depending on if the training set sent is uppercase or lowercase) and pixel-wise averages in the first group of the training set with the first item in the comparison set, followed by the second group of the training set with the second item in the comparison set and so on.

The average is done with the comparison set’s pixels weighted by the number of training sets previously averaged into it. This is done separately for uppercase and lowercase so that one need not submit the same number of uppercase and lowercase training sets.

3.2.3 OCR Processing

For OCR, each group in the segmented input is compared to each letter in the comparison set. For each letter in the comparison set, an overall score is computed. The group is considered to be the letter in the comparison set that received the lowest overall score (lower is better). If one or more letters receives an overall score within a predetermined threshold times the lowest overall score, all such letters whose overall scores are within this threshold are sent to context processing. We determined this threshold to be 1.1.

A so-called “sub-score” is determined by summing the difference squared between the luminance value of each pixel in one image and the luminance value of the pixel in the same position in the other image. Sub-scores are computed for a pair of images for all possible offsets of up to two pixels in the x and y directions of one image with respect to another. This results in 5x5=25 sub-scores computed per pair of input images. This is illustrated in the pseudo-code below:

```
FOR X OFFSET OF A = -2 TO 2
        FOR Y OFFSET OF A = -2 TO 2
                COMPARE A AND B
```

The overall score referred to above is the best (lowest) sub-score computed for a pair of input images. This is important because parts of letters can be slightly shifted with respect to one another (the bottom of a “T” in one could be shifted slightly to the right of the bottom of the “T” in the other resulting in only the top part matching if we didn’t try computing the difference with one offset slightly from the other).

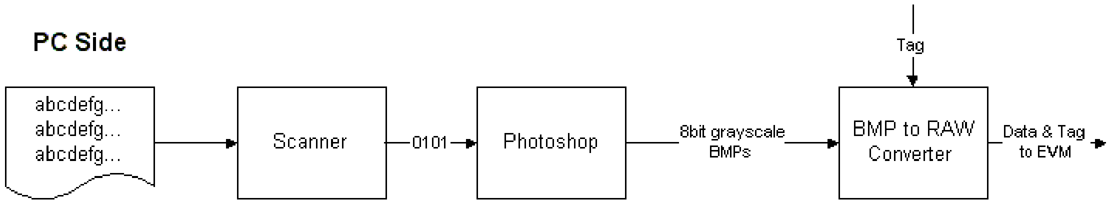
3.2.4 Context Processing

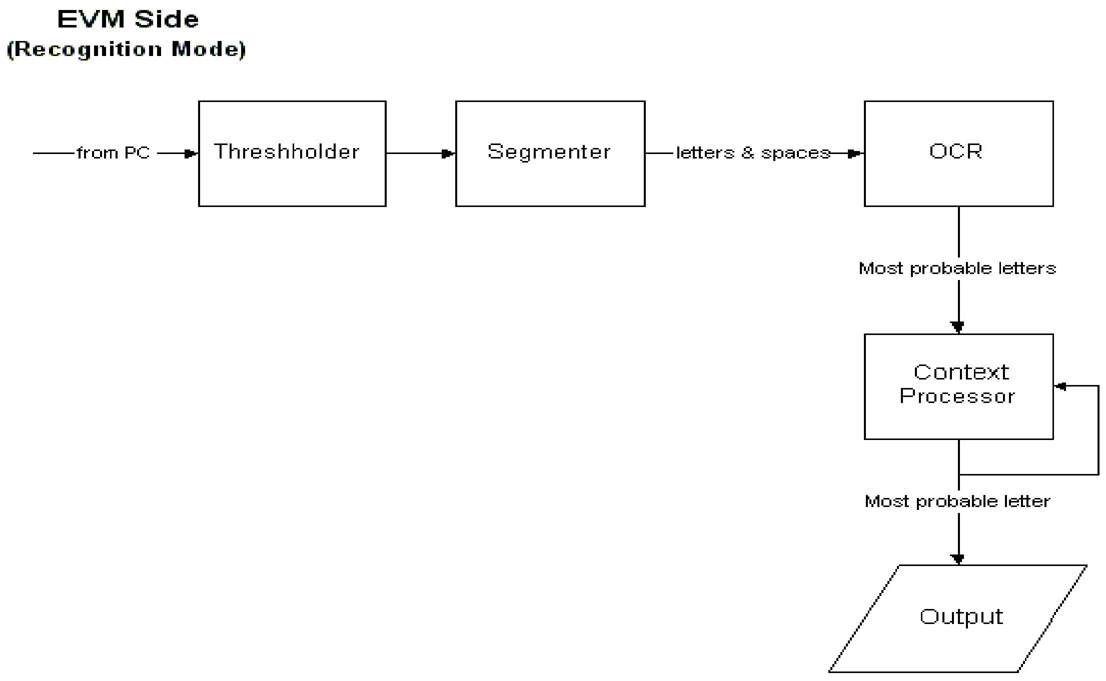
Refer to section 3.2.3 for reasons letters may be referred to context processing. Context processing is ignored in the following cases:

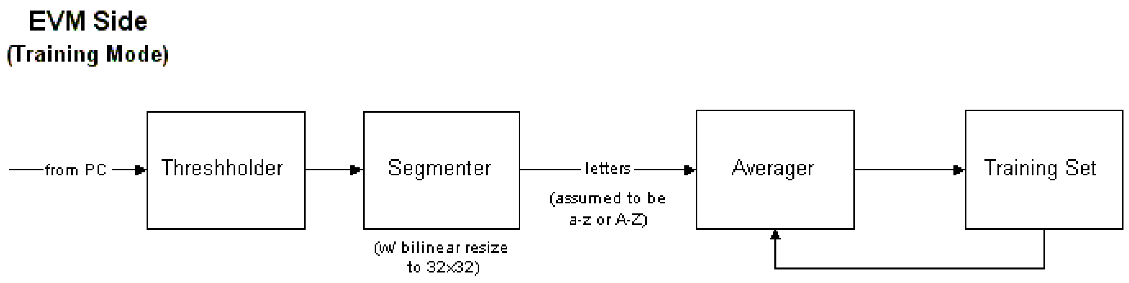
- If simple OCR is being performed
- If the previous letter output was a space
- If the previous letter output was uppercase

In these cases, context processing outputs whatever letter has the best (lowest) overall score. Otherwise, context processing ignores the scores entirely and outputs whatever letter is most likely in a dictionary, of those submitted (recall that only letters with overall scores within 10% of the best (lowest) overall score are submitted) to follow the last letter output.

4.0 SIGNAL FLOW







5.0 RESULTS

Group 8 of 2002 used a benchmark in which they trained the system on 50 lowercase alphabets and then did OCR on a different 50 lowercase alphabets. They achieved an accuracy of 75%.

We did this benchmark on our system as well so we could directly compare our performance with their performance. To do this on our system, we submitted 50 lowercase training sets to build a comparison set and then did simple OCR on a different 50 lowercase training sets. Two things are of note in this benchmark: First, simple OCR does not implement context recognition, and so these are raw results. Second, as only lowercase training sets are present in the comparison set, letters could only be recognized as lowercase (no letters could be mistakenly detected as being capital letters). Since Spring 2002 group 8 did not implement context recognition or capital letter detection, this allows our results to be directly compared.

Our recognition rate was 96.8%. Table 5.1 shows what letters were detected as what. The first element in a row indicates the input character and the following columns list how many times the character was recognized as the character at the top of the column.

Most errors occurred with characters that have similarities. Looking at the Table 5.1, we can see that ‘z’ gave lowest accuracy out of all the lowercase alphabets. Out of 50 z’s, OCR detected 45 correctly but misinterpreted 5 of them as ‘x’. There were also confusion between ‘c’ and ‘e’, ‘g’ and ‘q’, and ‘r’ and ‘h’. Some of this character confusion is obvious, for example a lowercase ‘e’ is very similar to a ‘c,’ except for the middle line down the center, and a printed lowercase ‘g’ and ‘q’ are also very similar, differing only in the direction of their respective tails. Other cases are less obvious and occur in part due to the nuances of our user’s handwriting. Due to the fact that most of our printed z’s were written with a middle line, they were easily confused with ‘x’s. Similarly, the h’s could easily be viewed as an ‘r’ with an extended line.

- z - versus x -
- r - versus h -

	a	**b**	**c**	**d**	**e**	**f**	**g**	**h**	**i**	**j**	**k**	**l**	**m**	**n**	**o**	**p**	**q**	**r**	**s**	**t**	**u**	**v**	**w**	**x**	**y**	**Z**
a	**50**																									
b		**49**									**1**															
c			**50**																							
d				**50**																						
e			**2**		**48**																					
f						**48**						**1**								**1**						
g							**49**										**1**									
h								**49**										**1**								
I									**50**																	
j										**50**																
k											**50**															
l												**50**														
m													**50**													

n													**49**				**1**									
o	**1**													**49**												
p											**1**				**49**											
q																**50**										
r								**2**							**1**		**47**									
s																		**50**								
t																			**50**							
u																				**50**						
v										**1**											**48**	**1**				
w																						**50**				
x																							**49**	**1**		
y																								**50**		
z																							**5**		**45**	

Table 5.1 – Output data based on recognition only (no context processing)

We also wanted to find out how our recognition algorithm's performance varied with the number of training sets in the comparison set. We built different comparison sets from different numbers of training sets ranging from 10 to 100 training sets and tested 5 sentences using full OCR. Figure 5.2 shows the results.

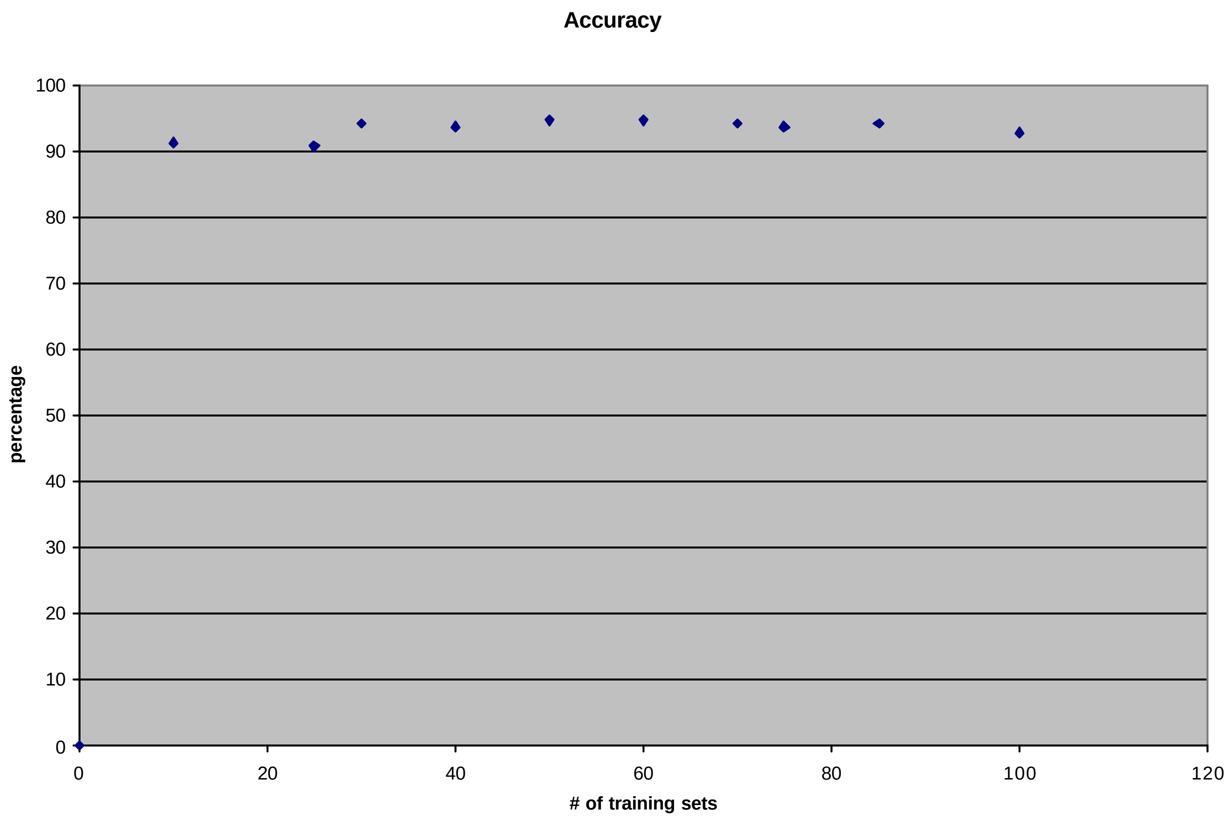
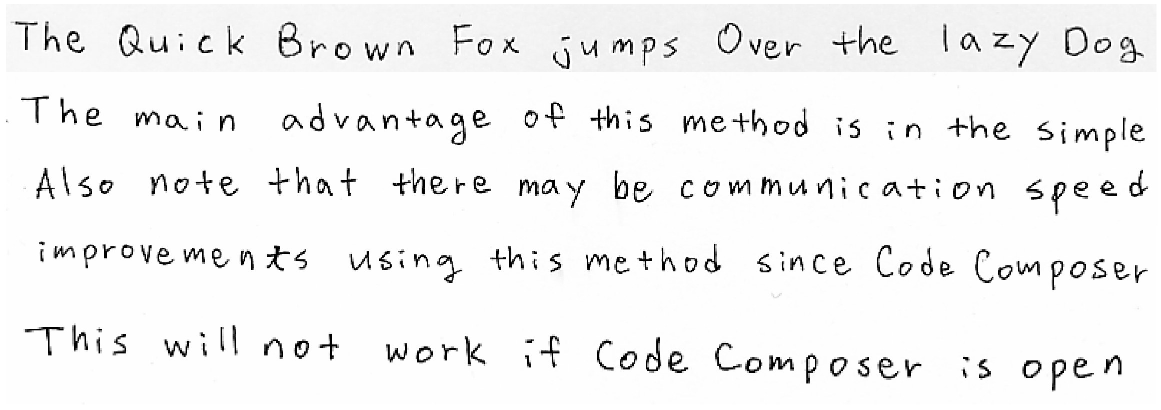


Figure 5.2 – Recognition rate vs. # of training sets

The best result was 96.4%. Interestingly enough, this occurred for 20, 50 and 60 training sets. Also interesting was that even with 10 training sets, the recognition rate was above 90 percent. Accuracy seemed to decrease slightly at higher numbers of training sets submitted, probably because the letters become very blurry. Note that, theoretically, as the number of training sets submitted goes to infinity, recognition rate goes to zero because everything in the comparison set approaches being a 32x32 solid gray square.

To analyze the effect of context recognition, we did OCR on 25 training sets **without** context recognition, 25 training sets **with** context processing, 50 training sets **without** context recognition, and 50 training sets **with** context processing. The results are shown below; letters in red indicate a letter that was misidentified:

Original Sentences



1. The Quick Brown Fox jumps Over the lazy Dog
2. The main advantage of this method is in the simple
3. Also note that there may be communication speed
4. improvements using this method since Code Composer
5. This will not work if Code Composer is open

Using 25 training sets without context processing

1. The Quick BroWn Pox jumps Over the laky Doq
2. The main adVantage of tis metbod iS in tHe Slmple
3. AlSo Hote that there maY bO comMuHication sfeGd
4. improYemeAts usinq this mothOd since polk Composer
5. This Wlll not Work if code Composer iS opcn

174/205 = 84.878 % accuracy

Using 25 training sets with context processing

1. The Quich Brown Pox jumps Over the laky Doq
2. The main advantage of tis method is in tre Slmple
3. Also Hote that there may bo communication spead
4. improvemerts using this mothed since pole Composer
5. This Wlll net Work if Code Composer is opon

186/205 = 90.732 % accuracy

Using 50 training sets without context processing

1. The Duick BroWn Fox jumps Over the laky Dog
2. The main advantage Df diS method is in tHe Slmple
3. Also note that thehe may bO commuHication sfeed
4. improYemeAts using this mcthOd since Cole CompoSer
5. This will not Work if Code Composer is open

186/205 = 90.732 % accuracy

Using 50 training sets with context processing

1. The Duick Brown Fox jumps Over the laky Dog

2. The main advantage of dis method is in the Slmple
3. Also note that there may be communication sfead
4. impropemerts using this mothod since Cole Composer
5. This will not Work if Code Composer is open.

194/205 = 94.634 % accuracy

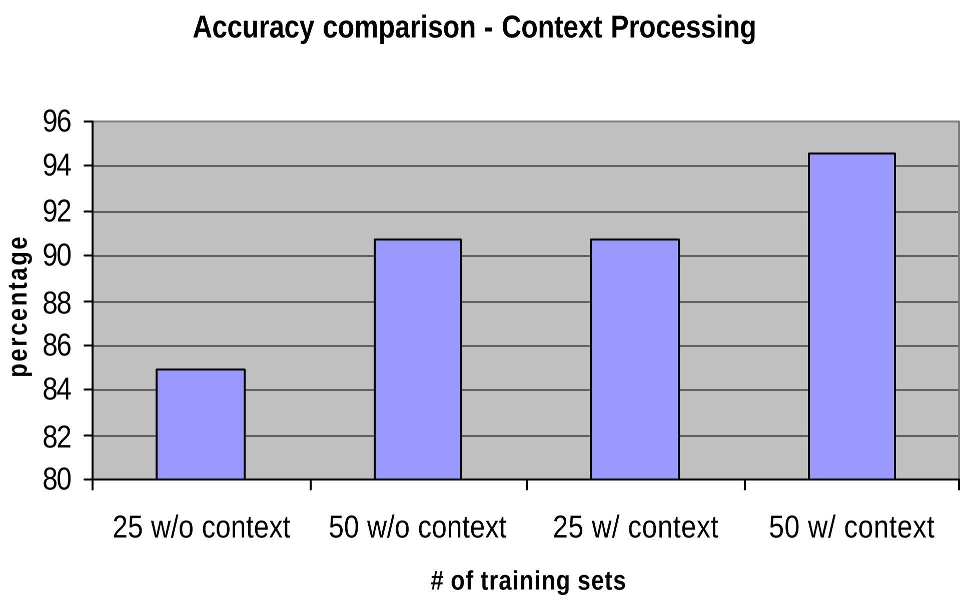


Figure 5.3 – Plot of results of context processing study

The results show that context processing improves results by a few percentage points. With a small amount of training sets, the OCR algorithm often confuses some lowercase letters with their uppercase equivalents. The most prevalent example of this occurs in sentence 2 with 25 training sets without context processing. Capital letters are common, often occurring in the middle of sentences, though the desired result only has one capital at the beginning of the sentence. The reason for this confusion is that our resize algorithm treats each group segment equally, resizing them to the same size each time. For letters in which lower and uppercase letters are very similar, this can be a problem (a lowercase ‘w’ looks exactly like an uppercase ‘W’ when both are resized to 32x32).

Fortunately, our context post-processing can fix many of these errors, since it eliminates any chance of an uppercase letter following a lowercase one. To further improve the base algorithm however, it may be wise to implement some sort of baseline extraction, knowing that lowercase letters tend to appear only on the lower half of the centerline while uppercase letters will cross the center into both halves. This may also help distinguish letters that actually fall below the baseline such as ‘j’ and ‘g.’

6.0 MEMORY ALLOCATION AND SPEED

Our memory map for the EVM is shown in table 6.1:

```
         name            origin    length      used    attr    fill
----------------------  --------  ---------  --------  ----  --------
ONCHIP_PROG             00000000   00010000  0000f480  R  X
SBSRAM_PROG             00400000   00004000  00000000  R  X
SBSRAM_DATA             00404000   0003c000  00000400  RW
SDRAM0                  02000000   00400000  00000000  RW
SDRAM1                  03000000   00400000  00100000  RW
ONCHIP_DATA             80000000   00010000  0000ee1b  RW
```

Table 6.1 – EVM memory map

Note that the program memory is almost completely full – our program just fits in the EVM’s memory. Almost everything is stored in ONCHIP_DATA; however, the great majority of ONCHIP_DATA is taken up by the comparison set, which uses 52x32x32 bytes (0xd000 out of 0xee1b). Since the majority of our accesses are into the comparison set, this greatly speeds up our algorithm. This is due to the 25 accesses per pixel per item in the comparison set per group in the input during OCR.

Unfortunately, this doesn’t leave any room for the stack in ONCHIP_DATA, so we had to move it to SBSRAM_DATA. This is not too much of an issue since our algorithm uses very few local variables, however, this slows down the segmentation which uses a lot of recursive function calls. We tried moving the stack to ONCHIP_DATA and the comparison set to SBSRAM_DATA, but this was far slower.

The only other things not stored on ONCHIP_DATA are those buffers that are allocated with malloc and thus stored on SDRAM0:

- The receive buffer
  - Receives images from the PC
- The map buffer
  - Stores what pixels in the input image are in what groups
- The groups array
  - Segmented 32x32 groups to be put in or compared to the comparison set
- The spaces array
  - Stores the location of spaces in the input sentence (for full OCR)

The sizes of all of these vary greatly with the input data size and thus cannot effectively be pre-allocated.

Another major memory saving technique we used was to apply the bilinear resize filter as the data was copied from the receive buffer to the groups array (see section 3.2.1). Without this another buffer of approximately the same size as the groups array is needed and all the data in that buffer has to be copied one more time.

Speed enhancements were difficult to make because the actual mathematical operations are very simple. We put the compiler on –o3 for the final result and saw some improvements in our overall speed. The profiler produced the following results:

- Training: Approx. 68,000,000 cycles/26 letters

    - Approx. 2,600,000 cycles/letter
- OCR: Approx 114,000,000 cycles/35 letters
    - Approx 3,300,000 cycles/letter

However, the profiler seemed to have a lot of difficulty profiling our algorithm. The above results are the minimum inclusive results; all other results turned out to be garbage (numbers on the order of 10^20!!). A better metric is that it takes about :15 to process a training set of 26 characters at 96 dpi and about :30 to process a 35 letter sentence for OCR. The majority of time in our algorithm is spent in segmentation, probably due to a lot of recursive calls and the stack being on SBSRAM_DATA.

Note that in an actual application the training sets only need to be submitted once per user and after that are just stored on the chip in the comparison set (probably in EEPROM in a real-world implementation).

7.0 REFERENCES

Although our algorithm was entirely original, we used some basic ideas and dos and don'ts from several sources:

- http://www.computerworld.com/softwaretopics/software/apps/story/0,10801,73023,00.html
- http://www.acadjournal.com/2001/V5/part5/p2/
- http://www.cs.byuh.edu/research/wilmott/491/paper.html

We also took the bilinear resize algorithm from:

- http://www.codeproject.com/cs/media/imageprocessing4.asp

The algorithm was modified to work in-place as data was copied from one buffer to another as detailed above (see sections 3.2.1 and 6.0). The PC-side communications code was taken from 18-551 Spring 2003 Lab #2 and modified to fit our needs.