Recognizing Hebrew alphabet of specific handwriting

Final Project by

Rotem Dvir and Osnat Stein

rotemdvi@bgu.ac.il  steinos@bgu.ac.il

Introduction:

Motivation:

Let's say, we have a scanned notebook of all the notes in Computational Vision course. Let's say, it has 100 pages. Now, we want information about a specific topic, for example chain code. We would like to have the option to search the scanned file, but obviously we can't.

Our only option is to browse manually the entire notebook, and search the wanted topic until we find it.

We could have been saving a lot of time and effort if we had software that translates the scanned text into machine-encoded text. Then, by using an existing search functions, we could easily find what we looked for in a short time.

This is only one example of the benefits in translating scanned images of handwritten text into machine-encoded text. Other usages can be:

- Reprocessing and rearranging the text.
- Combining the text with another document.
- Redesign the text.
- Using other applications that require text files.

Nowadays, this kind of translation can only be applied on images with printed text and not handwritten text [3]. In addition to analyzing the pattern of the letters in the text, this software uses other methods for improvement, like checking words in
dictionary and probability analysis. Even then, there is no 100% guarantee for success.

**Background**

In this project, we chose to use the chain code method in order to characterize the handwritten letters, and analyze them.

Chain code is encoding of each connected component in an image. For each component, a point on the boundary is selected. The encoder then moves along the boundary of the image and at each step, transmits a symbol representing the direction of its movement.

We learned about two different types of chain code:

- **Absolute chain code**: Using the encoder described in pic1 to encode the direction of movement from pixel X. The encoder doesn't change.
- **Relative chain code**: Using the encoder described in pic1 to encode the direction of movement from pixel X. When moving to some direction the encoder rotates to a state where number '2' points towards the moving direction.

![Pic1](image)

**Goal:**

Recognize handwritten Hebrew alphabet of specific handwriting, known in advance. When given a scanned image of an alphabetic letter, we would like to recognize this letter. This is the first step in the process of translating a full scanned text into machine-encoded text.

We decided to focus on a few letters: פ ג י ג ה.
We selected this group of letters, since it contains simple letters, more complex letters, letter with two disconnected parts, and also similarities between some of them.

Assumptions:

- Prior input: scanned images of all letters in the group, written by specific person.
- Input: scanned image of one letter, which we will recognize.
- All the letters should be written black on white, in a specific size and in specific position on the image.
- The images should be noise-free.

Approach and methods:

Generic Algorithm:

Input: scanned letter image.

Output: Numerical chain code.

1. Find black pixel to start the encoding from.
2. While there is black pixel who wasn’t visited:
   2.1. Find a direction to continue.
   2.2. Update chain code and mark the pixel as visited.
   2.3. Move forward to the chosen pixel.
3. Return chain code.

The algorithm is generic, and there are few question marks, that can cause very different results:

- How to choose the beginning pixel?
- How to choose the next direction if there is more than one option?
- Which encoder to use?
- What to do if we reached dead-end, but there are still black pixels left?
After getting the chain code for the input, we can determine which letter from our alphabet it is, by comparing its chain code to the chain code of each one of the prior letters we received. Another question is how to describe similar chain codes?

**Choosing the beginning pixel:**

Since we don't know in advance what letter is in the image, we have to decide on a generic method to find the pixel to start from. We would prefer this pixel to be an edge pixel. The problem is we can't find a generic method that will guarantee that the first pixel will be an edge pixel.

Specifically, in the Hebrew alphabet there are some locations for the letter beginning point: it can be the most top right pixel of the letter or the most bottom left pixel, like in the letter ֔ but in different letters it can be exactly the opposite, like for example in the letter ֐. For other letters, there are different locations.

After exploring the Hebrew alphabet, we came to conclusion that it might be best for most letters to start at the top left pixel.

**Choosing the next direction:**

Now that we know where to begin, we need to decide how to choose the next direction. If there is one black pixel adjacent to the current pixel there is no dilemma. But if there is more than one option, how can we choose the best? The situation of having more than one option can be caused by two different reasons:

- If the beginning pixel we chose is not an edge one.
- The letter line is not thin enough, and therefore there is more than one black pixel around the current pixel. It's important to notice that in most cases, a real handwriting is not a thin line. This is one of the main problems we had to deal with.

**Choosing the encoder:**

A relative encoder takes in account the direction of progressing because the encoder rotates when moving. This is an advantage because the letters are continuous and
we need to know the direction we came from, in order to know how to move forward. On the other hand this is a disadvantage, because the chain code will contain mostly the numbers 1, 2, 3 and the purpose is to get different chains for different letters. Another advantage of the relative encoder is that it enables recognition of letters written in different angles (like \( \mathcal{N} \) and \( \mathcal{Z} \)).

Choosing an absolute encoder means ignoring previous directions. We will get variety of numbers in the chain code.

**Dead – end:**

We need to distinguish between the different types of dead-ends:

- Dead-end caused by past wrong choice for continuation, which leaded to a pixel we can't go on from although there is a continuation in the letter.
- Dead-end caused by reaching the end of the letter.
- Dead-end caused by a sharp curve of the line.

In each case we need to take different actions.

**First approach – Absolute naive chain code:**

In this approach, we chose to use absolute encoder. To determine the beginning pixel, we chose the first black pixel discovered when scanning the image by rows, starting from the top.

To choose the direction to continue, we scanned the pixels around the current pixel, from direction 8 (described in pic1) against the clock direction, looking for black pixel. This is a naive algorithm, since it doesn’t necessarily fit the letter's shape. This way, moving right has the highest priority, with no practical reason.

If reaching dead-end, again it uses the naive algorithm: search a new beginning pixel at the same way as before – top and most left – and continue.

**Second approach – Relative chain code:**

In this approach, we chose to use relative encoder. The beginning pixel was chosen in the same way as the first approach.
To choose the direction to continue, we give priority to direction 2, since the encoder rotates according to direction of progress. If the pixel in direction 2 isn't black, we will check the pixels in direction 1 and 3, and so on, until we find a black pixel. If we didn't find black pixel, we are in a dead-end. If both directions we examine are black (e.g. 1 and 3) we need to decide between them. For now, we don't have a good solution for this problem, and we simply gave priority to the lower directions.

When reaching dead-end, first we scan the close environment of the pixel, to find continuation. If a pixel was found, we will continue creating the chain code from this pixel. If a pixel was not found, we mark all the pixels in this part of the letter as visited, and add a special character (9) to the chain code. Now, we look for a different part of the letter, if exists, and continue from it.

The motivation for this behavior is to try to ignore pixels we already visited their environment, but not miss pixels of different part of the letter (like in p).

Third approach – Absolute chain code consider previous direction:

In this approach, we chose to use the absolute encoder, to get variety of numbers in the chain code. We also wanted to use the advantage of the relative encoder - consideration of past direction.

The beginning pixel was chosen in the same way as the other approaches.

To choose the direction to continue, we gave priority to the previous direction we chose, but without rotating the encoder.

When reaching dead-end, we act the same way as the second approach.

Method for recognizing input chain code as a letter:

After getting the chain code, we want to determine which letter from our alphabet it is. For this purpose we developed a state machine. The state machine receives as an input the chain code produced by the third approach algorithm, and return as output the letter the chain code describes (from our alphabet). If the input chain code
doesn’t describe any of the letters, the state machine will return a proper message. This state machine is designed for chain code that represents a general shape of the letter. Therefore, it is suitable only for thin letters.

**Results:**

In order to test the output of the first approach, we use a scanned notebook. We took different instances of the same letter, and produce chain codes. Then we tried to find similarity between them.

- For simple letters like I ("Vav") we got pretty good results. It is possible to see the resemblance between their chain codes:
  
  \[
  I : 85858585366858585858585666
  \]
  
  \[
  \bar{I} : 585858585658585666
  \]

- For more complex letters like N ("Mem") the results wasn’t similar:

  \[
  N : 658585665333335858854685468585667768686
  \]
  
  \[
  \bar{N} : 5858558545885858858858858686685854685858547
  \]

  The main difference is that in the first chain code we can see sequence of 3’s that represents the middle line of the letter (from bottom up). This sequence does not appear in the second chain. This is only because of a small change in the arrangement of the pixels, that led in the second chain to dead-end and therefore starting again from the top.

  We can also notice a problem: In the second chain code we have mainly 5’s and 8’s, while it describes lines in different directions (the three lines of "Mem").

  From those results we came to a conclusion that we need to find more elegant way to deal with dead-ends, and take into account direction of progress.

We developed the second approach, and tested it the same way:
- All the chains we got, also chains of different letters were consisting from mainly 2, 1, and 3.
- The main difference became the length of the chain code, and it wasn’t enough for determine what letter this is.

The next step was to combine between the approaches in order to obtain both: characterize chain codes and resemblance when expected. So we improved the algorithm and developed the third approach.

- The results were better, but not clear enough to distinguish between the letters.
- Example of output chain code:
  \[ N : 6656655723333346666656622213777529 \]
  \[ N : 665555788222125333177677823566666563122269 \]
- We can see some resemblance between the chains.
- A small change in the letter leads to big change in the chain code.
- In order to avoid this problem, the chain code should describe the general shape of the letter.
- As long as the letters are not thin enough, we will have big differences in the output chain code.

We realized that in order to distinguish between letters, it is necessary to use "thin" letters. This way the chain will not contain interruptions, and describe the general shape of the letter. We tested the third approach algorithm on thin letters.

- The results were excellent. We got characterize chain code for each letter.
- Example of output chain code:
  \[ N : 6565665533332256669 \]
  \[ N : 6656656333432256669 \]
  \[ \mathcal{O} : 88787766555587766554545439 \]
  \[ \mathcal{O} : 88877676555567777655454449 \]
  \[ \mathcal{O} : 8777667656696666666656656669 \]
  \[ \mathcal{O} : 877766666966666566566669 \]
• It is possible to notice the resemblance between two instances of the same letter, and the differences between two different letters.

We tested the state machine with several thin letters, some letters belongs to our alphabet, and some doesn’t. We created the chain code for this letters by the third algorithm, and used it as an input.

• Many instances of the same latter from the alphabet were tested.
• Letters outside the alphabet were tested: we chose similar letters to the alphabet letters (for example ד ,ה) and different letters (for example ש ,ב).
• The state machine had 100% success in recognizing the letters.

An attempt to solve easier problem

The original problem was to recognize hand-written Hebrew alphabet of specific hand-writing. We managed to get good results only for thin letters, but most hand-writings are not thin enough. An easier problem is to determine if an input letter is a specific target letter or not. This way, we can analyze the input letter according to the guide lines of the target letter, and focus only on this target. Therefore, we wanted to develop an algorithm that solves this decision problem.

Since we couldn't produce characterizing chain codes for (not thin) letters, chain codes of two instances of the same letter can be very different. In addition, the chain code of letters can be misleading, a fact that can cause similarity between chain codes of two different letters.

Therefore, we came to a conclusion that such an algorithm cannot produce good results, as long as using chain codes as described above. It is possible that using different algorithm for creating the chain codes, would allow us to solve this decision problem.

Conclusions:

The goal is not achieved yet. When comparing the results to the goal, we can see there is a lot of work to do and many directions to keep investigating. Here are the conclusions from what we done this far:
• The chain code must describe the shape of the letter. Otherwise, a very minor change can lead to very different output.

• For thin letters we had good results, but for regular letters it is much harder. To identify the shape of the letter in regular letters it is necessary to perform intensive and complex analyze of the image while creating the chain code.

• Improvement: In all the approaches, the staring pixel was the top left black pixel. We can use other method for selecting the starting pixel, such as finding an edge pixel.

• Reduction: Instead of trying to develop an algorithm for regular letters, we can focus on preprocessing the letter by making it thin.

• This project was about off-line recognition - performed after the writing is complete. A different approach is on-line recognition - recognizes the writing while the person is writing. This approach might have better results (since it is easier to know the direction of progress) but the motivation for using it is different.

References:


