Content recognition of business cards

Christian Theil Have, cth at itu.dk
Supervisor: David Gustavsson

September 5, 2007

Summer Project, IT University of Copenhagen
Contents

1 Introduction ........................................... 3
  1.1 Properties of business cards ..................... 3
  1.2 Overview of the prototype ........................ 3
  1.3 Background of the author ........................ 4

2 Image preprocessing .................................. 5
  2.1 Grayscaling ....................................... 5

3 Segmentation ......................................... 5
  3.1 Registration and alignment of the business card within the image 5
    3.1.1 Sobel edge detection .......................... 5
    3.1.2 Fitting the lines .............................. 6
    3.1.3 Business card object alignment ............... 8
  3.2 Thresholding ...................................... 11
    3.2.1 ISODATA algorithm ......................... 12
    3.2.2 Triangle threshold algorithm .............. 12
  3.3 Segmentation of words and characters ............ 13
    3.3.1 Connected Component labeling algorithm .. 13
  3.4 Reasoning about the components .................. 14
    3.4.1 Component clustering ....................... 15
    3.4.2 The touching characters problem .......... 16
  3.5 Slant detection and correction ................... 17
    3.5.1 Slant correction approaches ............. 17

4 Classification ........................................ 18
  4.1 Neural networks .................................. 18
    4.1.1 Perceptrons .................................. 19
    4.1.2 Neural network architecture ............... 20
    4.1.3 Training the Neural Network ............. 21
  4.2 Generating training and test data ................. 23
  4.3 Feature selection and extraction ................ 24
    4.3.1 Properties of a feature ..................... 24
    4.3.2 Downsampling and size normalization ...... 24
    4.3.3 Histogram based features .................. 25
    4.3.4 Evaluation of the feature extractor .... 25
  4.4 Optimizing learning ................................ 27
    4.4.1 Fitting the network ....................... 28
    4.4.2 The optimal learning rate .............. 29

5 Conclusion .......................................... 31
  5.1 Review of the process ............................ 31
  5.2 Future work ...................................... 32
1 Introduction

Business cards are small cards used to communicate contact details in the corporate world. Exchanging contact details using this method seems rather old-fashioned considering the state of modern communication technology where contact details are stored electronically, for instance in mobile phones. However, it's seems unlikely that business cards will go away any time soon, since they are an integral part of business tradition and etiquette. Besides communicating contact details, business cards also sends an expensive status signal and brands the company.

The purpose of this project and the subject of the paper is to study aspects of how OCR techniques can be applied to extracting the information on a business card, so it can be represented electronically.

One particular usage scenario could be to integrate business cards with existing mobile camera phones. Mobile phones equipped with a camera, micro-processor and internet access has become widespread, opening up to possibility to use optical character recognition to read contact details from business cards and save them on the mobile phone.

1.1 Properties of business cards

The content of a business card is normally printed only one side of the card. The paper type varies in quality from high-gloss photo paper to matte paper perhaps with a textured surface.

The standard size of business cards is defined in ISO 7810, ID-1. The ID-1 format specifies a size of 85.60x53.98 mm, where the proportion of the sides corresponds to a golden ratio. Most business cards actually have this size within millimeter precision, but of course there are exceptions.

The layout used on business cards is very varied. Company logos and graphics are often interleaved with the text, different and non-standard fonts can be used and so on.

However, the contents of a business card is rather uniform. Usually a company name/logo and the contact details of the owner of the card. The contact details might include title, name, address, phone numbers, fax number, email address, website etc.

1.2 Overview of the prototype

I have experimented with implementing various parts of a prototype for business card content recognition. The result is not a complete system, but rather an hands-on study of the various parts involved in such a system. In figure 1 an overview of such a system is shown. I have only implemented selected parts of the system. I would have like to see the whole thing pieced together, but this was not possible in limited time of a 7.5 ECTS project.

However, to properly illustrate the process I will include an overview of how the different components should be pieced together. The offset is a picture of...
a business card taken with a mobile phone with a built-in camera. This image is then gray scaled. The first task is to locate the business card within the image. Once, the business card has been located, it is extracted and aligned using rotation and scaling to compensate for the angle of picture. The business card image is then thresholded in order to separate objects from the background. Connected component analysis is used to extract the characters in the thresholded image. The characters resulting from this process are classified using a neural network. The character components in the image are grouped together with a clustering algorithm, in order to identify words and sentences. The resulting clusters are classified to form words and sentences and graphics components. The classifications of the characters can be improved by considering the context in which they were found using linguistic features. Finally, the recognized characters grouped into words using clustering must be classified according to the type of information the word represents; phone number, address, name, etc. Figure 1 illustrates the process and also shows what parts have been implemented as part of this project. Some of the parts, that have not been implemented, are briefly discussed in section 5.2.

1.3 Background of the author

I'm a student at ITU on the software development line. I have a lot of programming experience, but I have no prior experience with image processing. I have some experience with artificial intelligence and neural networks obtained from the course Ai Game programming.
2 Image preprocessing

2.1 Grayscaling

Since most of the photos we consider are 24/32 bit RGB color photos, we need to convert them to a gray-scale representation. It is easier to work with gray-scaled images, since such images only have one intensity parameter. In the color representation considered, the most significant byte represents red, the second most significant byte represents green and the least significant byte represents blue. The process of converting an RGB image to a gray-scaled image is simple. For each pixel color, the value of each color-band byte is extracted by using a bitmask and the value is normalized to by shifting the bits accordingly, e.g. \( \text{red} = (\text{color} \& 0xFF0000) >> 16 \). The gray value of a pixel is the average of it’s normalized values for red, green and blue.

3 Segmentation

Segmentation is the process of locating the objects of interest within the image. In the case of business cards, this means locating the characters in image. Before normal OCR character recognition can occur, the input image must be split up into the characters that can be recognized individually.

The segmentation process presents quite a few challenges. The first one is to locate the business card within the image. Second the object representing the business card must be aligned so that we do not have to work with it from a skewed angle. Next, we have to find the objects within the image. The important objects, the text on the business card, must be separated from the noise objects such as irrelevant graphics etc. Having located the the text object we need to segment these into words and characters. The characters the are final result, which is fed to the classification process.

3.1 Registration and alignment of the business card within the image

The approach taken to finding the business card within the image is based on edge based segmentation techniques. A Sobel edge detector is used to find the edges between the business card and the background. Then these edges must be fitted to lines represented parametrically. The intersections between the lines found are calculated, and the final result of this process is the four coordinates of the corners of the business card.

3.1.1 Sobel edge detection

Edge detection is a segmentation technique which highlights edges in a image. An edge can be seen as a discontinuity in the gray-level of the image, a jump or fall in intensity. I’ve implemented the commonly used Sobel edge detector (described in [Gre02], [SHB98] and [YGv04]), which exploits the fact the derivative
of a sigmoid is a bell curve. If we consider a "one-dimensional image" with a jump in intensity it will look like a sigmoid, cf. figure 3. If we replace the pixel values with the magnitude of the gradient, it will cause the place of the jump to get a high-intensity value while the values on each side of the jump will get low intensity values.

The Sobel edge detection algorithms use convolution filters across the image to highlight the edges. A convolution filter is a "weight-mask" which is applied for each pixel and assigns that pixel a new value based on the a weighted average the surrounding pixels. The particular masks of the Sobel algorithm are 3x3 masks derived from the idea about the gradient of a sigmoid. The Sobel algorithm applies two convolution masks to be able to detect both horizontal and vertical lines. The convolution filters used is shown below:

| -1 0 1 | +1 +2 +1 |
| -2 0 +2 | 0 0 0 |
| -1 0 +1 | -1 -2 -1 |

$G_x$ $G_y$

Figure 2: Sobel convolution masks

The value of a pixel is the the magnitude of the gradient, given by

$$|G| = \sqrt{G_x^2 + G_y^2},$$

where $G_x$ and $G_y$ is the the result of of applying the horizontal and vertical filter respectively, for a particular pixel $(x, y)$.

Since almost no pixels are zero-valued, a certain amount of noise from non-significant gray-level changes sneaks into the result. Thresholding can be used to remove this noise, but as described in section 3.2, selection of a threshold value can be very difficult. In many cases encountered, the triangle threshold algorithm (see section 3.2.2) removes the noise and preserves the important edges pretty well, but in some cases I have been forced to manually fix a threshold for a reasonable result.

The result of applying the algorithm to a business card is shown in figure 4.

### 3.1.2 Fitting the lines

The lines in the image have been highlighted using the Sobel edge detector, but the output of edge detection is not directly representable and usable by

Figure 3: A sigmoid (symbolizing a jump in intensity) and it’s gradient.
geometric operations. We need to map the lines in the edge detected image to a parametric form, which we can work with geometrically.

Normally something called a Hough transform [DH72] is used to perform this step. The Hough transform considers lines of various angles through each point. Instead of representing the lines on slope-intercept form, the Hough transform uses a parametric form $r = x\sin\theta + y\cos\theta$, where $r$ is the distance to the closest point on the line and $\theta$ is the angle from the origin, between the x-axis and that point. For all lines going through a point $(r, \theta)$ is constant and is called the hough plane. Points in the image plane, thus corresponds to sinusoids in the hough plane and points lying on the same straight line in the in image plane corresponds to curves going through the same point in the hough plane. Intersection of sinusoids in the hough plane, means that a straight line can be drawn between the corresponding points in the image plane. By finding the most intersecting sinusoids, the most likely lines in the image plane are determined.

The Hough transform is a bit complex and quite computationally expensive, so I decided to use a simpler heuristic based technique. The particular nature of the problem does not require all lines to be found, only the four sides of the business card. By exploiting the fact that these lines will most likely be the outermost lines in the image (with the possible exception of noise from the background), the lines can be detected more easily.

The method builds on the assumption that the entire business card is contained in the image and that it occupies a reasonable percentage of the image. Unlike the hough transform, the method traces lines directly in the image plane.

The method works by tracing lines in the relevant part of the image and counting object pixels for each line. For instance, if we are looking for the left-most edge, lines are traced from every pixel in the top of the left-most third of the image to every pixel in the bottom of the left-most third of the image. The lines are traced using the digital differential analyzer (DDA) algorithm, described in [FvDFH90], section 3.2. For each line traced, the pixels visited by the DDA is compared to the actual pixels in the image and black (object) pixels are counted and stored for each line. The traced line intersecting most object pixels is assumed to be the left-most edge. The algorithm proceeds similarly for
the other three edges.

The lines traced are longer than the edges in the image. Essentially we are only interested in the segment of the traced line that corresponds to the edge. We can find these segments by finding out where the lines intersect, which is a trivial geometric operation, now that we have the lines slope intercept formula. Actually, by finding the intersections of the lines, we have found the four corners in the business card. With the four corner points it is also trivial to derive the diagonals and to compute their intersection. The intersection of the diagonals represents the center of the business card.

3.1.3 Business card object alignment

A photograph of a business card can be taken from different angles. This really complicates segmentation and classification process. The algorithms used for this usually requires that the input image is aligned. Several approaches [SK06, KFK99, WTL94] exist for aligning the input image when it is two-dimensional, as is the case for scanned documents. The process is also known as skew estimation and correction. However, the business card will be rotated a bit depending on the angle camera when the picture was taken and none of approaches seems to be suitable. I couldn’t find an directly applicable approach described in the examined OCR related papers, which solves the problem in three dimensions. However the problem can be seen as problem of registering a 3D object.

A registration matches the object with a given template, using an euclidean transformation (translation, rotation and scaling) that maps the points of the template to the points of the object. If such a transformation can be found, it is possible to apply the inverse transformation to every point in the business card object, which will cause it to be aligned.

We already know the format and size of a business card, which is described in section 1.1. A template with the same shape and proportions is used. We assume that the object is a rigid shape with no local deformations. For instance, a case where the business card has been bent is not considered. We can calculate
the center of the template using the intersection of the diagonals as we did
with the business card object. If we align the center of the template and the
object, design a 3D coordinate system such that the center of the 3D version
of template and object is the origin of coordinate system and place the camera
along the z-axis, we can match the objects using a perspective projection of the
rotated/scaled template with no need for object or camera translation. I have
written a small 3D engine, which does exactly this.

The photograph of the business card can be seen as a perspective projection
of the "real world" 3d environment. Consider that the business card is a plane
aligned with the X and Y coordinate axis (that is, every point in the business
card has the same Z coordinate). If we also assume that the camera was per-
fectedly aligned with the Z-axis (a point 0,0,z) then the problem becomes much
simpler. We just need to find the scale-factor (which depends on the cameras
z-coordinate position) and the rotation around the Z-axis. However, the as-
sumption of perfect camera alignment of course does not hold, all though it is
probably safe to assume that the rotation will be fairly small. The pictures we
consider will be taken from various angles, which mean that we will have to find
three rotation angles (x, y, z). The solution cannot be directly calculated, since
it is an equation with three unknowns.

Search based methods   Instead, search based methods can be used. The
search space can be considered infinite, although, really it is not since the tar-
get image is discrete. But this is an irrelevant detail. Brute-force search is
still not feasible and we need an intelligent search method that incorporates
heuristics. To reformulate the template matching problem, we need to find the
transformation which minimizes the error according to some heuristic which
indicates how close we are to a perfect match. Also, the rotation and scaling
problems can be solved independently. If the rotation problem have been solved,
then the scaling problem can be solved using the rotated template.

Two search algorithms for determining rotation have been implemented, a
hill-climbing algorithm and genetic algorithm. Also, a couple of heuristics to
use with the algorithms have been implemented.

Heuristics   A good heuristic should provide a measure of how good given
template transformation is with regards to the original object.

The heuristics designed compares the 2D coordinates of the template with
the 2D coordinates of the business card using some function, after a given ro-
tation and perspective projection of the template has been performed. The
purpose of the heuristic is to evaluate whether the rotation aligns the template
with the business card object.

Minimum distance heuristic   The first heuristic I attempted to use was the
sum of squared distances from the templates corner points to the objects corner
points. When used with rotation optimization search, this heuristic frequently
caused the search algorithms to converge to a local minima, to my great despair.
Figure 6: Left: Template corner point heuristic didn’t work out that well for figuring out the rotation. This is the rotation that the algorithm converges to. Note, that the image contains the diagonals, the template before rotation and the template after rotation. Right: The result of using the second heuristic for rotation and the first for scaling.

Usually, the rotated template will approach the object wrt. only one of the corners. The situation is illustrated in figure 6.

**Parallel Distance Heuristic** The second heuristic does a better job estimating the rotation error. It measures the angles between each corresponding line in the template and the object. An zero angle means that the template is aligned wrt. one of the sides of the business card object.

Actually, it turned out that both of the heuristics was needed. The first heuristic is works well when optimizing the scale of the of template and the second heuristic works well for optimizing the rotation of the template.

**Hill-Climbing algorithm** The hill-climbing algorithm works by taking small steps in the direction of the correct rotation. It starts with a rotation configuration where the template has not been rotated yet, e.g. a rotation of \((x = 0, y = 0, z = 0)\) degrees. A step size defines the amount change in rotation per iteration, given in radians, and is initially initialized to 0.1.

Each iteration, the algorithm considers all combinations of rotations along each axis with step size added to or substracted from the current rotation configuration. This amounts to twelve possible rotations. Each of these rotations are evaluated using the parallel distance heuristic and the one with the lesser error is selected. This rotation configuration is then the starting point of the next iteration. If none of the rotations improves the rotation error, we can assume that the algorithm has converged as much as the step size allows it to. The step size is then decreased to by an order of ten before the next iteration.

When the step size reaches a certain error margin, the maximum allowed rotation error, the algorithm terminates and the optimal rotation has been found. The algorithm can be compared with gradient descent, but with the difference that this algorithm does not directly utilize the gradient to determine the step size. The algorithm has the weakness that it might converge to a local minima, however, with the right heuristic, this does not seem to happen.
**Genetic algorithm**  The genetic algorithm was implemented because the hill-climbing algorithm seemed to converge to a local minima, but was abandoned when this problem was solved by using the parallel distance heuristic. Even though it’s seems possible to avoid local minima with the parallel distance heuristic and hill climbing, a description of the genetic algorithm is included for completeness.

The genetic algorithm can be used to optimize the rotation angle. At first it generates a generation of rotations. Then for each iteration, the best fifth of rotations are chosen using tournament selection. Five rotations are selected at random from the generation and only the one with the least error will survive to the next generation. After selection, new rotations are added to the generation by cloning survivors (asexual breeding) and applying a few mutations to them with a certain probability. The mutations are random but applied in a relative controlled manner and are quite small. There is a tradeoff between generating small rotation mutation and exploring the search space. If we allowed larger mutations we might find an other area of the search space which is more interesting, than what we can achieve with more local optimizations. The algorithm terminates when the error is small enough (according to some threshold) or when the algorithm has produced 200 generations. The genetic algorithm is quite slow to converge and sometimes it doesn’t converge at all. However the result produces when it did converge, was similar to the results of the hill-climbing algorithm.

**The Inverse transform**  If we apply the inverse transform to every point in the business card object, it will be aligned. However, we will also need to interpolate the points. This has not been implemented.

### 3.2 Thresholding

Thresholding is a simple segmentation technique that converts an image to an 1-bit (black-and-white, no shades of gray) image. To do this a parameter called the **brightness threshold** is chosen. Assuming dark objects on a light background, pixels that exceed the threshold value are identified as objects, and other pixels are identified as background.

The difficult part of thresholding is to choose a good threshold value. The simplest approach would be just to pick a given value in the middle of the spectrum and use that for all images. This approach doesn’t consider the particular features of an image and is sub-optimal in almost all cases. Fortunately, a number of good algorithms exist for doing this. The most prevalent approaches derives the threshold value from a pixel intensity histogram. Threshold selection algorithms each have weaknesses and strengths and they are usually based upon some assumption about distribution of the histogram.

In the case of business cards photos we can also make a few assumptions. We can probably assume that there will be a large peak in histogram signifying the background (of the business card) and one or more smaller peaks for the objects. The image will usually also have some space around the business card,
which may produce a relatively large peak in the histogram. We can, however, eliminate the problem with the background by performing a registration of the business card object cf. section 3.1.

However the number and size of these peaks may vary quite a bit. Second, the quality of the image may degrade because of bad lighting conditions, which seems to be the rule rather than the exception, since mobile camera phones have small optics. Some areas of the picture will be quite dark while other will be bright because of the light. This is especially true if the business card has a glossy surface that reflects the light better.

The rest of this section contains a description two algorithms I implemented and considered using for this tasks. The end of the sections concludes this by discussing their usefulness to application photos of business cards.

3.2.1 ISODATA algorithm

The ISODATA algorithm (meaning Iterative Self Organizing Data Analysis Techniques) works by choosing a threshold in the middle of the intensity range. It then iteratively adapts the threshold value. In each iteration the average intensity value of each part of the histogram divided by the threshold is computed. A new threshold is computed as the average of the two averages. The algorithm is finished when the threshold doesn’t change between iterations.

I’ve implemented the ISODATA algorithm as described in [YGv04], but the results wasn’t very good. It seems to consistently converges to a very low threshold value and the result is unusable. It may be a bug or result of the particular features of the image. I suspect that I need to normalize the histogram so that the average values makes more sense.

3.2.2 Triangle threshold algorithm

The triangle algorithm is ”particularly effective when the object pixels produce a weak peak in the histogram” [YGv04]. This will often be the case with business cards where the objects only amount to small percentage of pixels. So this seems like a fairly good choice for business cards.

The triangle algorithm constructs a line between the dominant peak (background) and the lowest value in the histogram. It then finds the intensity value where the euclidean line distance to the constructed line is longest. This intensity value is chosen as the threshold value. This is illustrated in figure 7.

In most cases the triangle threshold algorithm does quite well, but it seems to include poorly lighted background regions of image as objects, as it can be seen in figure 8. The problem is that local regions in the image differ in intensity and one threshold must be selected for the entire image. I haven’t implemented a region based thresholding algorithm (also known as adaptive thresholding), but I experimented with manually applying the triangle algorithm on particular troublesome regions and the result was promising. However, it’s non-trivial to decide on a way to segment the image wrt. such regions.
3.3 Segmentation of words and characters

Segmentation of sentences, words and characters is often seen as two different types segmentation. According to [HM71], External Segmentation is the isolation of various writing units such as paragraphs, sentences and words, while internal segmentation deals with the isolation of letters.

3.3.1 Connected Component labeling algorithm

Connected component analysis is a method used for grouping pixels which are connected to each other. In a word, for instance, each character will roughly corresponds to one component.

The approach used is described in [Fis00]. The input is the aligned, thresholded business card image. A black pixel is assumed to be part of a component and white pixels are non-components. The product of the process is an image where each pixel in a coherent component is labeled with the same integer label.

The algorithm works by processing each pixel in the the image from left to right, top-down. If the examined pixels is an object pixel, it’s previously processed neighbors are examined (there are no need to process neighbors which will be processed later). We handle three distinct cases:

- No neighboring pixels have labels assigned (no object pixels), in which case the pixel being processed is assigned a new unique component label.
If just one pixel in the neighborhood is an object pixel, the current pixel must belong to that component and is given the same label as the neighboring pixel.

Should two or more pixels in the neighborhood have different labels assigned, we need to make a decision wrt. which label to assign. The pixel being processed actually connects the neighbor components and should be merged into one component. We postpone connecting the components, but make a note of the equivalence of the labels. For the current pixel, we just select the label value of one of the neighbor components. The equivalence is noted in a graph, such that there are edges between equivalent labels. As result, it will usually not be a fully connected graph.

When all the pixels have been labeled, there will usually be a number of equivalent labels in the equivalence graph. The graph has an entry node for each label and the equivalent labels are all the connected labels, which are found by traversal. When an equivalence class has been resolved, all the involved label nodes are removed from the graph. The process continues until there are no more nodes in the graph. Each resulting equivalence class is assigned a new unique label. The equivalent labels are stored in a table, mapping from the original labels, to the equivalence labels. For convenience, the resulting labels are normalized so that they occur in sequential order. Finally, in a second pass though the label matrix (the output image), all labels are reassigned to the corresponding equivalence class label.

The result of applying the algorithm to a portion of a business card image is shown in figure 9.

### 3.4 Reasoning about the components

Connected component labeling roughly divides the characters into components. Even though it seems to do okay with the example in figure 9, there are a
number of problems:

1. Some characters blend together and are detected as one component, which is problematic wrt. classification. This problem is known as the *Touching characters problem* [HM71]. A method is needed for detecting this and separating touching characters.

2. *Graphics components* The example in figure 9 is rather contrived, since most inputs would also contain graphics components. We need a way to distinguish text components from graphics components.

3. *grouping components* The labeling does not give us any hints as to how words can be comprised of the detected character components. In a larger perspective we would like to know which words are grouped together.

3.4.1 Component clustering

The third problem can be addressed using clustering. In this context, clustering means grouping together components (representing characters) into words and words into sentences. Letters that appear close to each other are likely to be part of the same word and words appearing together are likely to be part of a "sentence". For instance, a name will usually consist of a few words. To be able to decide what components are close together we need a distance measure. In this case it makes sense to use the minimal distance between components; that is the smallest euclidean distance between object pixels in the components.

We can also exploit the fact that text is written left-to-right and normally horizontally aligned. This results in the following distance formula, for two component clusters $\alpha$ and $\beta$. $\delta = \min(\text{dist}(\alpha, \beta) \ast \text{abs}(\text{slope} + 1))$. We find the distance from each pixel in cluster $\alpha$ to every pixel in cluster $\beta$. Each distance can be considered a straight line between the two pixels. If the line is near-horizontal, the slope approaches zero, and distance is multiplied by a small factor. If the slope is near vertical, it’s multiplied by a very large factor. Of course, since a vertical line has an infinite slope, we will cheat by multiplying only with a very large value. This approach penalizes vertical distances, and thus clusters characters in the same line closer together.

Since we wish to find different kinds of clusters (words and sentences), it seems natural to organize the clusters in a hierarchical fashion. Common hierarchical clustering methods use a tree-representation. Cutting the tree at a given height threshold will yield a specific kind of cluster.

Many simpler clustering algorithms such as ISODATA, "k-means clustering" and self organizing maps require a fixed number of clusters in order to automatically divide a data set into clusters. We have no way of predetermining the number of clusters, so these algorithms cannot be directly applied. Instead, I have chosen to use a agglomerative, hierarchical approach. The algorithm is outlined below.

Still remains difficult to break up the tree in a sensible fashion. How do we determine if some sub-cluster is a word for instance. A heuristic to determine
Input: The set of clusters found by the connected component labelling
Output: One cluster, consisting of subclusters, organized as a tree of clusters

\[ \text{shortestdistance} \leftarrow \infty \]
\[ \text{mergeclusters} \leftarrow \text{nil,nil} \]

while There is more than one cluster do
  foreach Cluster \( \alpha \) do
    foreach Cluster \( \beta \) do
      if \( \min(\text{dist}(\alpha, \beta)) \leq \text{shortestdistance} \) then
        \[ \text{shortestdistance} \leftarrow \min(\text{dist}(\alpha, \beta)) \]
        \[ \text{mergeclusters} \leftarrow \alpha, \beta \]
      end
    end
  end
  MergeClusters(mergeclusters)
end

Algorithm 1: Agglomerative clustering algorithm

this is difficult to come up with. Instead of designing such a heuristic a neural network classifier can be used. To do this some sort of feature calculation is required. Due to the limited time given for the project, I have not implemented a neural network classifier for this problem.

3.4.2 The touching characters problem

In handwriting touching characters is very common. However, the problem is a bit different with printed characters. In handwriting, the cause of touching characters is that the author writes without removing the pen from the paper. Thus, characters are usually only connected with a thin line. In our scenario, touching characters occur because of bad focus or lighting conditions in the image and not because they are touching in the first place. However, some of the methods for separating characters in handwriting can also be applied to our problem.

If we can detect that a component consists of two characters we need to separate the component using a cut; a particular division of the component into two components. Deciding how to make such a cut can be very difficult. In our case, it’s seems fair to assume that the cut is a vertical line through the image (if the characters are not slanted, see section 3.5).

Since the neural network classifier (see section 4) gives sort of a confidence score for each character, we can use the classifier to gives an idea of how good a cut is. The character classifier could be applied for multiple possible cuts, and we could use classification with the highest confidence score for the detected characters. The neural network classifier can also help us detecting components with touching characters, since the result of classifying such a component is likely to give a low confidence score for the classification. However, it can
be difficult to detect if a component is two touching characters or just one character. For instance, if a sequence nn touch, it might look a conspicuous lot like an m. The context in which the character is found might give some hints about this. For instance, if we expect that the character is part of a name, and the surrounding characters allows the following interpretations Kenny and Kemy. We might be able to infer that Kenny is the more likely interpretation and thus the component must be two touching n’s.

I haven’t implemented any of the described mechanisms for dealing with touching characters.

### 3.5 Slant detection and correction

Slant in text, as encountered with italic fonts, complicates segmentation and classification. Slant is generally a bigger problem with handwritten text, where the slant might vary even within a word. My focus has been slant in printed text, and I have assumed that the slant is constant across a word.

Slant in segmented characters can also create problems in classification. For instance, a slanted K may have similar features to non-slanted X in certain fonts, and can easily be classified incorrectly as such by the neural net classifier.

#### 3.5.1 Slant correction approaches

I have considered approaches which corrects slant uniformly. [RPW04] suggest rotating the the word through a series of slanted positions and claims that the minimally slanted configuration will have the smallest width. An other approach [BS89] works by drawing slanted strokes of various angles across the text and measures the amount of pixels that intersects with these strokes. [KFK00, KFK01] and [dZ06] approach the problem using a vertical histogram. The basic idea is that the minimally slanted text will have higher peaks and lower drops in the histogram. I’ve decided to use a histogram based approach, since it seems like the most robust of the examined approaches.

The basic operation of my approach works the same as the other examined histogram approaches, but the feature calculation score is done a little different. [KFK00, KFK01] uses something called a Wigner-Ville distribution. [dZ06] uses compares average height of the three highest peaks to a fixed threshold. When the the peeks reach the threshold, the text is considered deslanted.

I assume that the slant is at most 45 degrees in either direction. This is a threshold limit implemented in the code, because I think this is a reasonable assumption, and is also used by other approaches. Since we are dealing with printed text, the slant will almost always be in a forward direction.

The approach works by shearing the bitmap of the text using various angles between -45 and 45 degrees. The sheared image outcome is scored according to a vertical histogram based feature calculation. Note, this is not an intensity histogram as the one used in section 3.2. The vertical histogram records the number of object pixels in each pixel column.
The feature score is calculated based on the peaks and valleys in the histogram. It rates the histogram with the highest peaks and the lowest valleys highest. The algorithm is shown below:

**Algorithm 2: Feature score algorithm**

The shearing operation with the highest score is assumed to be the least slanted configuration. The algorithm works reasonably well, but the shearing operation seems to leave noise artifacts. The result for a given sample is shown in 10.

### 4 Classification

#### 4.1 Neural networks

The classification scheme that I have implemented and trained is an artificial neural network with a histogram based feature extractor. Artificial neural networks are models inspired by the the function of the human brain, or in particular the neocortex. I will not give an explanation of the biological principles neural networks, but the following sections provide an overview of the principles of ANN’s and how those principles are applied in my implementation. In the following, the term neural network refers to an artificial neural network.
4.1.1 Perceptrons

The fundamental unit in a neural network is the neuron, also called a perceptron. A perceptron can only be trained to classify linearly separable problems, but with a connected network of perceptrons, non-linearly separable problems can be classified. A perceptron is basically a function on a set of weighted inputs. The sum of the weighted inputs, known as the net sum, can be expressed mathematically as:

\[ \zeta = w_0 + \sum_{i=1}^{n} (x_i w_i) \]

where \( x_i \) is an input and \( w_i \) is the corresponding weight. \( w_0 \) is called the bias and is used as an offset value.

The core of the perceptron is the activation function, which squashes net sum. Usually the activation function is a step function, but when used in neural networks it is usually a sigmoid (or at least, some differentiable function). The activation function implements a sort of threshold; a hard threshold with the step function and a soft threshold with the sigmoid. The result of a simulation of a perceptron is thus: \( y = g(\zeta) \). The structure of the perceptron is illustrated in figure 11.

\[ \sum \rightarrow g \rightarrow y \]

Figure 11: Perceptron (image borrowed from [Nis03])

Simulating a perceptron will produce an erroneous value unless the weights have been set properly. Fortunately, perceptrons can be trained to reduce the error by optimizing the weights. Perceptrons are trained using an algorithm known as the delta rule. The idea behind the delta rule is that the scale of the error depends on the inputs and the error in the output can be blamed on the weights. Each weight is to blame for a part of the error. Optimization is done using gradient descent along the error-surface. The gradient of the error wrt. each weight is calculated and the weight is adjusted a little bit according to this. More precisely, the delta rule makes an adjustment of each weight \( w_i \),

\[ \Delta w_i = -\eta \frac{\partial E}{\partial w_i} \]

The fraction \( \frac{\partial E}{\partial w_i} \) is the partial derivative wrt. to the i’th weight, \( w_i \), and \( \eta \) is called the learning rate and is used to scale the adjustment. The delta rule simplifies to

\[ \eta x_i (t - y) \]
That is, the difference between the desired output \( t \) and the actual output \( y \) scaled by the learning rate \( \eta \). I will not go through the proof of this here, but it can be found in [Cha04] and [RN95]. The delta rule is applied repeatedly for all the samples until it converges toward the global minimum of the error surface. This can be be determined using simulation with the samples. If all the samples are classified correctly, within an allowed margin of error, we are sufficiently close to the global minimum and the algorithm terminates.

I do not use perceptron classifiers in my implementation, since they are too simple apply to the problem of character classification. Instead I use a fully connected network of perceptrons.

### 4.1.2 Neural network architecture

In a neural network, the perceptrons are connected to each other. This is usually done by creating layers where each of the perceptrons are connected to the next layers. This architecture is called a *feed-forward neural network*. It is also possible to create neural networks with back-connections, but feed-forward networks are by far the most commonly used architecture. Since each neuron a one layer has a connection to each neuron in the next layer, the network is called *fully connected*. Inversely, a network where this is not the case is called *sparse*.

A feed-forward ANN usually consist of an input layer, one or more hidden layers and one output layer. It’s common with only one hidden layer, since this is enough to approximate continuous functions (an ANN with more than one hidden layer can even approximate non-continuous problems (piecewise continuous)). I use this architecture although my implementation allows any number of hidden layers. A multilayer ANN in illustrated in figure 12.

In a simulation of the ANN, an input sample is fed to the input perceptrons and each of the those perceptrons are activated using the given input. The output of the input perceptrons are fed to the first hidden layer neurons and so on. The process is repeated until the output neurons are activated and gives the classification of the sample.

In my implementation each neuron in the output layer represents a character class. For instance, in a network to recognize digits, there will be 10 neurons in the output layer, one for each digit. When the network is given a sample, one of the output neurons should be (close to) 1 and all the other output neurons should be zero. It would also be possible to use only one neuron and then have ranges representing each character class. Even though it’s desirable to have a network with as few neurons as possible, I have chosen to to use several to get a *confidence score* for each classification. For instance, with my scheme, the network can predict that the sample input characters is a \( K \) with an 80 percent likelihood and an \( X \) with a 20 percent likelihood.
4.1.3 Training the Neural Network

Training the neural network can be seen as an optimization problem. Given a set of samples (the training set), find the set of weights which best generalizes the samples. There are many algorithms for optimization (such as gradient descent, genetic algorithms, simulated annealing, etc.). The purpose of the optimization is to find the global minimum in the error surface. There may be many basins in the error surface and while it may be easy to find the minimum of such a basin, it proves difficult to determine if it is the global minimum.

The most common algorithm for training neural networks is a technique called back propagation, henceforth referred to as backprop. Backprop is essentially gradient descent along the error surface. Backprop is said the be converged when the the descent reaches the global minimum. Backprop is not guaranteed to converge to the global minimum per se, but in practice this is usually not that big of a problem. Other algorithms (such as Quickprop and RProp) exists for dealing with the inherit problems of a gradient descent technique, but they haven’t been implemented or considered in this project. However, advantages and disadvantages of the approach should be mentioned.

Two classes of training algorithms, stochastic and batch, are common with neural networks. Stochastic approaches adjust the network for each sample seen where as batch training optimizes the network based on the whole training set at once. The incremental approach is well-suited for online learning, but can also be used in a batch learning scenario. There are different opinions wrt. what approach to use for various problems. Champandard [Cha04] argues that the batch training should always be preferred, whereas Lecun [LBOM98] argues that a stochastic learning approach have several advantages, in particular that it is faster and results in better solutions than batch training. However, batch training allows the incorporation of advanced techniques such as locally adaptive learning rate and higher order derivatives to better predict slope. Also, it is easier to theoretically analyze batch learning algorithms, since the conditions
of convergence is more well understood. Quickprop and RProp, which will not be described further (descriptions can be found in [Cha04]), are both examples of batch learning algorithms.

Training using back propagation  

Training is done by running a simulation of each sample in the training set and then adjusting for the error in the weights in the neural network. I train my neural with a batch version of back propagation which is illustrated in algorithm 3.

Input: A set of sample inputs, known as the training set: Samples
Input: A set of desired outcomes for each sample: DesiredOutcomes
Input: Maximal average error: MaximalError
AverageError ← ∞
while AverageError < MaximalError do
    foreach s ∈ Samples, d ∈ DesiredOutcomes do
        BackPropagate(s,d)
    end
    TotalError ← 0
    foreach Input sample i, desired outcome d do
        outputs ← Simulate(i)
        TotalError ← TotalError + SampleError(i, d)
    end
    AverageError ← \frac{TotalError}{\text{length(inputs)} \times \text{NumberOfOutputNeurons}}
end

Algorithm 3: Batch training with backprop.

The algorithm terminates when the average error reaches an acceptable level. The average error is calculated by running a simulation with all the samples in the training set, and the summing and averaging the results.

The error $e_n$ of an output neuron $n$ is the difference between the desired output $d_n$ and the actual output $y_n$:

\[ e_n = d_n - y_n. \]

The error of a neuron might be negative, so we use the Squared Error measure,

\[ E_n = \frac{1}{2} \sqrt{e_n^2}. \]

The function SampleError calculates this for each output neuron and sums the errors.

Theory of back propagation  
The idea of backprop is a generalization of the delta rule described in section 4.1.1. The error of the output can be blamed in part on each connection leading to a particular output neuron. For each incoming connection, it is determined how much this connection contributes to the error. The weight of the incoming connection is then adjusted accordingly. This procedure continues recursively all the way down to the input neurons.
The perceptrons in my implementation use the Sigmoid function (also known as the logistic function) as activation function. The sigmoid is the function,

\[ g(x) = \frac{1}{1+e^{-x}} \]

The error \( e_n \) of an output neuron \( n \) is the difference between the desired output \( d_n \) and the actual output \( y_n \):

\[ e_n = d_n - y_n \]

The error is calculated for each output neuron. The weights of each output neuron is adjusted based on the error signal:

\[ \delta_n = e_n g'(\zeta_n) \]

where \( \zeta_n \) is the net sum and \( g' \) is the derivative of the activation function. The derivative of the Sigmoid function is simply \( g'(x) = x(x-1) \). After an update, the weight between a neuron in the previous layer \( l \) and the output neuron \( n \) becomes

\[ w_{ln} \leftarrow w_{ln} + \eta \delta_n o_n \]

where \( \eta \) is the learning rate. After weights have been updated in output layer, we proceed to the hidden layer. The formula for calculating the error signal for a hidden unit \( i \) is:

\[ \delta_i = g'(\zeta_j) \sum_n (\delta_n w_{ln}) \]

The sum of errors in the neuron next layer scaled by the weight between the actual hidden unit and the next layer neuron multiplied by derivative of the activation function. The weight update procedure is the same for a hidden unit \( n \) connected to a previous layer neuron \( l \):

\[ w_{ln} \leftarrow w_{ln} + \eta \delta_n o_l \]

The procedure continues until the weights connecting the inputs to the input neurons have been updated. This way, the errors observed in the output layer are propagated all the way back through the network, thus optimizing the network wrt. the the sample.

### 4.2 Generating training and test data

To train the network we need a considerable amount samples. In our case a sample is simply an image of a character. One approach to doing this would be to collect a large stack of business cards and extract all the images of characters on those business cards. However, this approach would require a lot of work and a lot of business cards.

Instead, I have written a tool for generating sample images. The tool, which I simply dub Font Generator is configurable via XML and can generate samples using various fonts and distortions. It generates size normalized and centered characters images. There are three types of supported distortions:

- A percentage of random noise (percentage is configurable).
- Stochastic rotations between configurable degrees.
• Stochastic shearing operations.

I use the tool to generate both training and test sets. The tool allows me to generate as big sets and experiment with the various settings. Of course, the training sets generated by such a tool will not be the same as if I had extracted the characters from the business card images. Various other distortions will appear on the characters in a business card image. So it can be discussed how well a network trained on samples generated by the font generator would generalize the character images on the business card, and I provide no direct measurements of this.

4.3 Feature selection and extraction

Traditionally OCR systems have used handcrafted feature detectors using image processing techniques. Such feature detectors can be very time-consuming to design. More recent approaches to OCR have applied "learning feature extractors", where the features are built in to so called convolutional neural networks, and are hence trainable.

I have experimented with a hand-crafted feature extractor, which have been applied to character recognition.

4.3.1 Properties of a feature

Since a given neural network always has a fixed number of inputs, a feature calculation should also result in a fixed number of input values. The simplest feature would probably be the values of the pixels in the input image. However, if the input image is large, this is usually not feasible. For instance in a 24x24 pixel image, there will be 576 inputs. Features should extract the relevant information and disregard noise and distortions. This property is referred to as invariance. There are a host of different ways of doing feature extraction, but given the limited timeframe of this project I have chosen only to use a simple histogram based feature extractor.

4.3.2 Downsampling and size normalization

One way of reducing the number of inputs is simply by down-sampling the image. Down-sampling simply scales the image down to a given smaller resolution. By scaling the image down to a reasonably small size, it can be directly as a feature to the network. For instance, an 8x8 picture will have a feature map of 64 inputs. Several authors [Hea05] propose to use such a down-sampled size-normalized image directly as feature feature vector.

The character images extracted during previous steps have different size and proportions. Since we require the feature vector to be a specific size, the images must be size normalized. This is done by cropping away white space and scaling the images to a particular size while preserving the proportions of the characters. The cropping algorithm trace horizontal and vertical lines from the borders and inwards until object pixels are encountered. The cropped image
is the minimal image rectangular image containing all the object pixels. The cropped image is scaled to fit an fixed size using a uniform scaling factor (e.g. it is scaled equally horizontally and vertically). This scaling factor is determined based on the maximum of either width or height of the cropped image. If the image is wider than it is high, the width determines the scaling factor and vice versa. Next, the image scaled image is centered with an image of a (larger) particular size.

4.3.3 Histogram based features

The histogram feature extractor exploit the statistical distribution of object pixels in the image. The histogram feature extractor works on the size-normalized image version of the character image. A horizontal and vertical pixel histogram are created and combined into a feature vector. A 24x24 pixel image will thus have a feature vector of size 48, which is a considerable reduction compared to the 576 size feature vector of the raw image. The feature extractor is illustrated in figure 13.

4.3.4 Evaluation of the feature extractor

The histogram based feature extractor significantly reduces the number of inputs to the network, but how well does it perform with regards to classification performance. Trained on a training set with 26 samples, one for each letter in the alphabet, it classifies them all correctly. However, we are more interested in how general the classifier is. That is, how well does a network trained using the feature extractor adapt to various scenarios. We would like the feature extractor to be as invariant as possible wrt. to noise and distortions.

Adapting to other fonts In particular, we cannot predict the font used on a given business card. However, we can test how well a network trained on a particular set of fonts, classifies samples of an other font. Humans can read letters, almost no matter what font is used because letters in the various fonts seems similar to us. The question is if the feature extractor can reach such a level of invariance to the font used.

To test this we train a neural network classifier on samples in select set of fonts and test it on sample in using a disjunct set of fonts. The set of fonts used is:
Training set: Helvetica, Marker Felt, American Typewriter, Andale Mono, Arial, Baskerville

Test set: Century, Chalkbord, Courier, Futura, Garamond, Verdana.

<table>
<thead>
<tr>
<th>Fonts in training set</th>
<th>correct classifications on test set</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>35.9%</td>
</tr>
<tr>
<td>2</td>
<td>41.0%</td>
</tr>
<tr>
<td>3</td>
<td>45.5%</td>
</tr>
<tr>
<td>4</td>
<td>40.4%</td>
</tr>
<tr>
<td>5</td>
<td>58.3%</td>
</tr>
<tr>
<td>6</td>
<td>66.0%</td>
</tr>
</tbody>
</table>

The invariance wrt. to different fonts isn’t quite as good as I hoped. However, the number of correct classifications seem to increase when the trained on more font samples, so there is hope. We cannot hope to be able to classify characters in any font, but would be nice with a better accuracy on common fonts. If the neural network classifier should generalize to large number of fonts, the required size of the network would probably increase drastically.

Distortions An effect of poor camera focus and light conditions distortions may appear in the characters in the business card image. The font generator tool allows to generate samples with various distortions, with which we can perform some tests of how invariant the feature extractor is with regard to such distortions. A have tested this for two kinds of distortions, namely random noise and rotations.

Invariance to random noise To test the invariance to random noise, I conducted an experiment where I trained a neural network classifier using non-noisy samples in one font, 26 classes, one for each letter. Of course, 100% correctness is achieved on the training set. I tested this network on a sets samples in the same font, but with different levels of random noise added. Each test set contained 10 samples for each letter. The classifier has a rather impressive invariance to random noise. The results are shown in figure 14. The performance of the classifier wrt. to noisy samples can be improved even more by jittering the test data, as described in section 4.4.1.

Invariance to rotations I did a similar test, to evaluate invariance to rotations. The neural network was trained on different 26 samples using the same font. In the test the classifier is evaluated on rotated samples with various rotation degrees. Classification results drops fast when the samples are rotated. When the rotation degree reaches 30, only about eight percent of the samples are recognized. I think that part of the reason that the classification rate is not closer to zero for this kind of rotation is that some characters are rather symmetric around their origins (for instance o).
Figure 14: Invariance to noise. As long as the noise levels are low, the feature extractor remains relatively invariant to random noise.

Figure 15: Invariance to rotation. Rotation significantly degrades the performance of the classifier.

The results are shown in 15. The conclusion is that the classifier is not very invariant to rotation. However, in section 3.5, methods for dealing with slant are discussed. These techniques could be applied to alleviate the problem.

4.4 Optimizing learning

To obtain the optimal configuration of the neural network, it is necessary to experiment different parameters such as the learning rate, size of the hidden layer etc.
4.4.1 Fitting the network

A good neural network should adapt well to unseen examples. After the training converges it will naturally adapt well to the samples in the training set. However, the degree of fit can only be evaluated using a disjunct test set.

If the network has been optimized to much to fit the training set, it might lack the generality to correctly classify the test set. This situation is called over-fitting. Under-fitting, which is the opposite of over-fitting, happens when the network hasn’t learned the training set well enough. The trick is to find the correct balance between over-fitting and under-fitting. There are many parameters affecting the fitness of the network and are they are highly interdependent, so we have to take a holistic view of fitness.

When has the network converged enough, so that the learning process should be terminated? We use an error margin to decide this, but this parameter can be adjusted to suit our needs. Other parameters such as the number of hidden neurons in the network and the size and generality of the training set also have a great influence on fitness. A network with too few hidden neurons will result in under-fitting and large number of hidden neurons has more potential for over-fitting. According to [Pre95] the best way of fitting is to use plenty of training samples. With a large training set which incorporate many realistic examples, it is more likely that the resulting network will adapt to unseen examples. If the training only covers a small percentage of examples, learning will generalize to this small set of examples and not be able to generalize to different examples. We can generate some noisy samples in the training set, in order to achieve a larger training set.

The deliberate introduction of noise in the training samples is known as jitter. I tried to test effect of jitter by training an classifier with and without a jitted training set. The regular training set, was 26 samples, one for each character. The jittered training set consisted of 5 samples of each character with 10 percent random noise. I have generated a large test set with 5330 samples, 205 samples per character. Each sample in the test set has a random noise rate between 0 and 40 percent. Characters with 40 percent noise are almost unrecognizable even for the human eyes. The results were:

<table>
<thead>
<tr>
<th>Classifier</th>
<th>Correct classifications</th>
<th>Errorous classifications</th>
</tr>
</thead>
<tbody>
<tr>
<td>No jitter</td>
<td>4080 (76.5%)</td>
<td>1250 (23.5%)</td>
</tr>
<tr>
<td>Jittered 10 percent noise</td>
<td>4854 (91%)</td>
<td>476 (9%)</td>
</tr>
</tbody>
</table>

The result of introducing is jitted training data is very impressive, a 14.5 percent improvement in correct classifications!

It’s difficult to predict how large the hidden layer in the network has to be, so it has to be found using trial and error. Heaton [Hea05] describes a few rules of thumb for selecting the correct number of hidden neurons, but he does not provide any evidence or explanation of his recommendations. The rules for selecting the number of hidden neurons are:

- should be in the range between the size of the input layer and the size of the output layer.
• should be 2/3 of the input layer size, plus the size of the output layer.

• should be less than twice the input layer size.

Of course, depending on the size of the input and output layer, these rules may actually conflict. It is also easy to construct an example where the rules of thumb definitely doesn’t hold. Consider a network for approximating the sine function. We have one input and one output, but the number of hidden neurons required is considerably larger.

I have found that for this particular problem, an hidden layer which is half the size of the input layer seems to work well. However, this depends on many things. If we want the classifier to generalize to many different fonts and various distortions, we would probably need a bigger hidden layer. [Hea05] doesn’t mention about the size and variety of training set in his rules of thumb. I think this is a very relevant parameter.

4.4.2 The optimal learning rate

The learning rate controls how fast a neural network learns with back propagation. It is the parameter which is used to scale the adjustment of the weights.

Normal back propagation use a fixed learning rate during the whole training process, although different learning rates at different points in the learning process can be beneficial. My implementation, the learning rate is fixed during the whole learning process. The optimal learning rate depends on the problem, so it must be found through experimentation. Since the weights of the network is randomly initialized, the number of epochs required for back propagation to converge can vary for a given learning rate. However, it is possible to observe some general trends.

When the learning rate is too small, the learning takes a long time to converge. When the learning rate is too large, the adjustments overshoots the minimums it is supposed to converge to and may as a result also take to long time to converge. This may observed as zigzagging in the average error instead of a relatively steady decrease. The situation is illustrated in figure 16.

![Figure 16: Zigzag effect because of a too large learning rate. In this case the average error still decreases, but it might actually increase in between epochs.](image)

I’ve conducted an experiment where a three layer network is trained to recognize all lower-case letters in the alphabet (26 classes). The training-set consist of exactly 26 samples, one for each character, in the same font. All weights
in the network are initialized randomly before training. Then the network is trained with different error rates from 0.001 to 1.5. The goal is to see which learning rate provide for the fastest convergence. The random initialization of the weights have the effect that training the network with the same learning rate multiple times will lead to different results. We are interested in the learning rate which provide the fastest convergence on average, so convergence is measured as an average of five training sessions. It would not help to just initialize the weights to the same values each time before training, because that would bias the results.

The average number of epochs required for converging with various learning rates, is shown in 17. It’s interesting to observe the peak around 1.2. Sometimes, for a given learning rate above one, the back propagation seems to get stuck and don’t converge further because of zigzagging. One hour and 300000 epochs later I had to stop it manually.

Very small error rates, such as 0.01, consistently takes a long time to converge, so that is apparently too small. Large error rates, around and above one, seems to converge very fast toward a minimum in the beginning. However, when the error rate becomes to large, problems begin to surface. Sometimes, they get stuck zigzagging (as with the peak in figure 17) causing the error rate to go up and down instead of steadily decreasing. This is illustrated in figure 18.

From the results of this experiment it seems that 0.4 is a reasonably good learning rate for this problem. No zigzagging occurred with this learning rate and it converges almost as fast on average as the larger error rates when no zigzagging occur.
5 Conclusion

5.1 Review of the process

I have worked with the process of recognizing the content of business cards on many levels, but there is still a long way to a working prototype. I produced quite a bit of code and implemented quite a few interesting algorithms and learned a lot during the process.

Starting out with almost no image processing experience, I have begun to realize how mind-numbingly big and complex this field is. I thought it would be relatively straight-forward to get a simple prototype working. I hadn’t really anticipated the complexity before I started.

It turned out that the task of creating simple prototype, is quite a huge task. I didn’t manage to produce a working prototype, but I suppose I did not really expect to do this either.

I did, however, manage to study almost every step along the way and implement some proof-of-concept algorithms for various tasks involved in the process.

I lot of my time was spent on basic image processing. I could have used libraries as a shortcut to some of the functionality, but everything was done from scratch. The higher purpose of the project is learning and I did it from scratch to get a deeper understanding and a personal satisfaction from seeing my understanding put to work.

I intended to spend a lot more of the effort on the parts where machine learning techniques can applied, but working with image processing techniques has certainly also been very rewarding.
5.2 Future work

Working with this project has increased my understanding of image processing and neural networks and opened a lot of doors to interesting problems. It is indeed interesting, but complex fields, which I would like to explore further.

I have developed some parts of a prototype for recognizing the content of the business card. The parts haven’t been put together to form an entire system and vital parts are still missing if the system should be applicable in the real world. It would be interesting to complete the system, but this is no small task. To fill in some of the gaps, I will briefly describe the parts which I think would be vital in a real system for this task.

I don’t think the pipeline approach, where each layer in the recognition process depends solely on the output of the previous layer, is good enough. This sort of one-way flow discards a lot of useful information. In many cases, information extracted at some point could be use increase the accuracy or performance of a previous layer.

For instance, the information discovered by the classifier could be used to do better segmentation. Consider for instance an example where the classifier recognizes a sequence of characters erroneously (e.g. we get a non-sensical output). We could use this information to conclude that the segmentation was probably wrong and try another one. When we get a sensible sequence out of it, we know that the segmentation is right.

Improving classification using linguistic features I haven’t implemented any error correction using linguistic features, but doing so would be essential for real-world system. For that, reason I will briefly outline how I imagined that this could be done.

The neural network classifier might not always give the right classification for a character. For instance, the digit 1 could easily be misclassified as the letter l. However, given the surrounding characters we should be able to make a better guess at what the character should be. For instance, in the context of business cards, if the surrounding characters of the misclassified letter l is all digits (e.g. a phone number), it would be fair to assume that the l should have been the digit one. Such a relation could be represented using a markov model.

The neural network is designed such that each output neuron represents a character. As a result, a given simulation gives a confidence score for each character. To apply models like markov models, we would like this confidence score to be the probability that a sample belongs to some character class. However, using the normal logistic function in the output neurons, we are not guaranteed that outputs lies between zero and one. To handle this, an other activation function such as the softmax can be used in the output layer.

It would be interesting to study if and how hidden markov models [Rab89] could be used to improve classification, for instance by inferring the most likely interpretations of a sequence of classifications.
Classification of content  After words in the business cards have been recognized and extracted, they should be classified according to their type. In particular, we are interested in different kinds of contact details. The contact details may be broken down into different categories such as: Person name, company name, title, address information, Telephone and fax numbers, mail address, website etc.

This could be done using feature patterns based on regular expression. For instance we know that an email address is likely to contain the @ character. Dictionaries with names and street names could also be used.

Also, I think the layout of the business card could be of relevance. For instance, the person name is likely be placed before the address. Such things could also be modeled using Markov models; eg. If we have observed the title and name, what is the most likely next element on the business card.

The classification of content is itself a very complicated tasks with a lot challenges. There are a lot of observable features, but a model of this is bound be rather complex. However, it would be a very interesting topic to explore.

State of the Art - Convolutional Neural Networks  Convolutional neural networks [LB95, LBBH98] are the state of the art within handwritten character recognition. In a convolutional neural network, the feature extraction is learned as part of the neural network. This provides for much better adaptability but also better accuracy than the traditional approach with hand-crafted feature extractors. In my histogram based feature extractor, I cut away a lot information in order to get the feature vector as small as possible. The reason that I want it to be small is as explained in section 4.3, that in a fully connected neural network, the number of connections increase drastically with the number of inputs. Each input must be connected to every hidden unit, so depending on the size of the hidden layer this is going to be a lot of trainable parameters.

Convolutional neural networks are inspired the the locally sensitive nerve cells in the visual cortex of a cat. A convolutional neural network implicitly extracts features by limiting the weights of the first layer to a local receptive field of the input image to a small convolution. The output of each neuron goes into creating part of a feature map of a lower resolution. A convolutional neural network has a number of convolution layers where this happen. The same set of weights are used in the feature map, which drastically reduces the number of trainable parameters. Because the features are generated on the basis of local receptive fields, the convolutional neural network becomes invariant to noise and distortions, to a very impressive degree. Training is done using normal back propagation, but as mentioned, there are less trainable parameters.

It would definitely also be interesting to implement and experiment with such a neural network.
References


<table>
<thead>
<tr>
<th>Reference</th>
<th>Title</th>
</tr>
</thead>
</table>