UiO : Department of Informatics
University of Oslo

UNIK4690 Project

Text Recognition

Akhsarbek Gozoev - akhsarbg
Sadegh Hoseinpoor - sadeghh
Key Lung Wong - keylw

Report

Contents

Chapter 1

Introduction

1.1 Project Description

The purpose of our project is to recognize text from image, and be able to do string manipulation on it. Hence this falls under the "Optical character recognition" (OCR) problem

1.2 Project Limitations

As OCRs are still a challenging task even for companies like Google, ref. reader to Googles OCR translator application on smartphones; "Translate", drawbacks such as; difficulty finding all the text on the photo because of lighting, noise etc., therefore we take it for granted that we should limit our text recognition problem.

Initial limitations

- English alphabet [upper and lower case] + numbers [0-9] + space
- Homogeneous background, white
- Skew free text
- Computer printed text
- Even lighting

1.3 Project Components

We have come to the conclusion that the OCR software has 3 main parts. Each part is essential, for the OCR software to be able to perform its purpose.

1. *Text segmentation*
    - Finding text segments on an image and returning the text segments
2. *Preprocessing*
    - Do preprocessing on the segmented text, such as rotation and line and symbol segmentation. Preprocessing from definition, should be done first, however because of simplification we assume we manage to segment out the text first.

3. *Classification*

    - Classification of the symbols

Additionally there is one more very important component for this OCR software to work, ***labeled data***. Even though one might not need to code for this part, a good pool of labeled data is needed to be able to classify symbols. More on this in section 2.5
4. *Data classification - Gathering labeled data to train a classification algorithm*

1.4 Report Layout

This report is divided into 6 sections. In Chapter 1, we introduce the project and how the report is organized. Chapter 2 is where we present methods we have used in our final result. Discarded methods tried will be covered in Chapter 3. In Chapter 4 we review our approach and its results, we present the confidence in our solution and what vulnerability exist. In Chapter 5 we present our thoughts on the project; our ambition vs the actual result, and potential future improvements on the software. Chapter 6 includes reference to third party material we have used, such as; code, articles and datasets.

Chapter 2

Method

2.1 Description

The following section, as mentioned in Section 1.4, will present our methods. Bellow we present each component individually and the corresponding methods. A short description on what challenges have to be solved for each component will be addressed first.

2.2 Component: Text segmentation

Description
In this part of the program, we want to be able to segment out text segments in the image. Afterwards we want to segment out lines and then letters for further classification, see section 2.3.2 and 2.3.3 for reference. In this part we assume that the images have black text on white background, and it mostly consists of text.

Approach: Simple Image Analysis Technics

Our approach here is inspired by this online blog Source[**_finding_????**]. We simplified the original approach to the following steps:

1. **Find Edges/Outliners of the image.** Initial idea is to use Canny, but we found Morphological Gradient to perform better. It indicates the contrast of the edges, so we can get better differences in some natural images (granted the text and background are close to black and white respectively).

2. **Otsu Thresholding.** We need our image to be a binary image. We simply use OpenCV Otsu algorithm to achieve it that.

3. **Morphological Closing.** Since we want line segments we use Morphological Closing with large horizontal filter to merge as many horizontal letters together as possible.

4. **Extract Regions.** OpenCv FindContours was used to find the different text regions. We then exclude regions which are smaller than a selected threshold. The different region are returned as coordinates of the different rectangular boxes.

2.3 Component: Preprocessing

Description
Definition of preprocessing; the act of preparing the data for further use, in our case for classification.
After the text segmentation we assume we have an image consisting of white text on black background. What remains for us to do is to segment out each character and format them to the right data type for the classifier. Hence we have 4 problems to solve:

Problems

- Rotated text
  - Our approach for character segmentation needs text rotated horizontally.
- Line segmentation
  - Our approach for character segmentation needs lines as input, as a sequence of lines on top of each other breaks the algorithm.
- Character segmentation
  - We need to segment each character because the classifier cannot distinguish several characters from one image.
- Data formatting/casting
  - The data we want to test a classifier on needs to match the data we trained our classifier with. Hence we need to format our data to the same format as the datasets. Done using simple array manipulation, described in 2.3.4

2.3.1 Find rotation

Approach: OpenCV minAreaRect()
This approach uses convex hull to find the convex hull of the text, and then rotating calipers to find the minimum area rectangle.

cv.minAreaRect returns text rotated in one of following angles [0°, 90°, 180°, 270°], see Figure 3.2 for illustration. Hence we have a limitation on what angles the text can have, [0, 90] degrees, for us to be able to rotate and classify the text correctly. An approach that covers this vulnerability is mentioned in Chapter 3.

1. **Binary image.** For the Convex hull algorithm to work, the text segments and the background needs to be distinguishable. Convert image to a binary image using OpenCV threshold function and/or bitwise_not to flip foreground and background colors.
2. **cv.minAreaRect().** Feed our newly generated image to cv.minAreaRect.
3. **Calculate angle differences.** From cv.minAreaRect() we can find the angle of the rectangle. From that we can calculate the angle of the potential rotated rectangle.
4. **Affine transform to rotate image.** We use Affine transform to rotate the rectangle.

2.3.2 Find line

After all the preprocessing done up to this point, we assume we have a segment of correctly rotated text. At this point it is enough to just use projection histogram. We basically sum number of active pixels in each row and end up with one dimensional array with same size as image height, one value for each row in input image. E.g. [0 0 0 0 0 5 12 18 20 15 11 0 0 0 0 5 7 8..], this means there are some data (most likely text) between 5th and 10th row, and 15th up to another line break. see fig 2.1

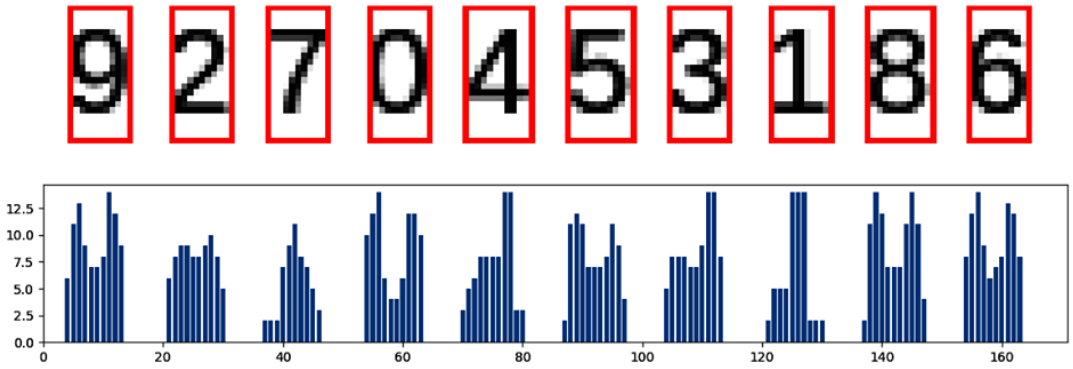


Figure 2.1: Vertical project histogram, same applies to horizontal

2.3.3 Find Symbol/Letter Segmentation

Letter segmentation was similar to Text Segmentation, section 2.2. We added a few steps and kernel shapes in the morphological part. The additional step was to fill holes in the image, for example; 8 and O, can give multiple wrong contours. cv2.floodFill was used to solve this problem.

1. Do threshold.
   Why do we do this? threshold the letters, for what reason?
2. Inverse the image since we work on black text on white background.
3. Do morphological dilation with a long vertical kernel, This is to include the dot in 'i'.
4. Add a border around the line-image, that separates elements from the edge after the previous step. This is to make cv2.floodFills work.
5. Fill all holes, cv2.floodFills.
6. Use cv2.FindContours
7. Ignore element where contour size smaller than a threshold(half the letter size), to ignore ',',',' and other non letter element.

2.3.4 Data formatting/casting

After we extracted single symbol from the line we have to feed it to the classificator. As we know our classificator accepts 28x28 grayscale images so we could simple resize the image to feed it, but this will more often than not yield poor results as the image will

not follow the same convention images used to train the network did, execpt for the image size ofc. So we have to do some manipulations in order to make our image look more like images our CNN is good at recognizing. Lets take a look at some images from training sets:

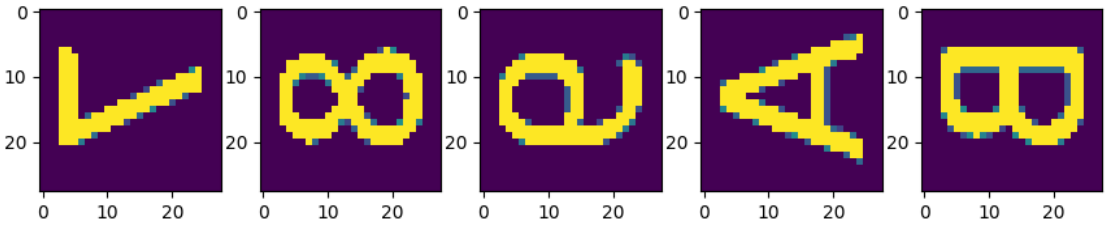


Figure 2.2: Dataset samples

As we can see the characters are centered and has about 4 pixels boarder around it. Furthermore it is not binary image with sharp edges, but rather smoothed out. One may have noticed that x and y axis are swaped, making image look transposed, but that's how EMNIST set stored its data, so we just decided to stick with it.
The algorithm we found working best is described below: First we find the dominant axis of the image using n = max(x, y), then we create new image with both axis equal $n + n//k$ where $k = 7$ as we found $\frac{28_{total-pixel}}{4_{border-pixel}} = 7$, then we paste our original character image in the center of newly created background image and only now resize it to 28x28 pixels. The resize operation will smooth the edges even though original digit may have been a binary image. Next we decided remove unnecessary noise resize operation may have introduced by setting all pixels with intensity value under 50 to zero, and bring out the main shape of our character by setting all pixels with intensity value over 127 to 255 (this only affects the core of the character, edges are still smoothed). And this gives us an image most likely to be correctly classified by our CNN. The final image is then transposed to match EMNIST data.

2.4 Component: Classification

Description
For the classification component there where only 2 approaches we considered, Convolutional Neural Network (CNN), the architecture is illustration in Figure 2.3, and a Multilayer Perceptron (MLP or Deep neural network (DNN)), Figure 3.3 illustrates the architecture. However the method we ended up choosing for the final result is CNN. In Section 3.4 we explain why we discarded the MLP approach.

Convolutional Neural Network
The basic idea of a CNN is; train some number of filters and then after the filters are trained, run the image which is now "filtered", through a fully connected network. The fully connected network will then try and classify based on the feature images from the convolution layers.
Below we present the architecture and variable choices we have made for the CNN.

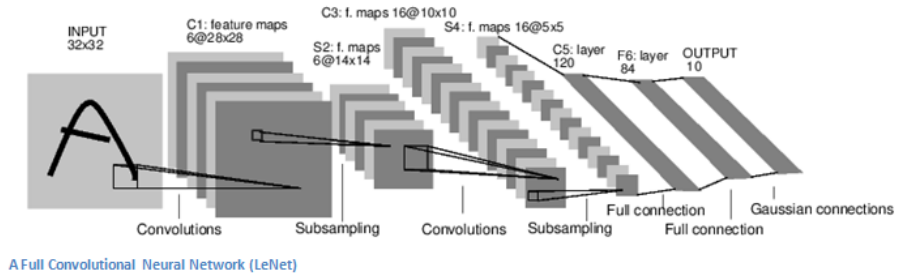


Figure 2.3: Convolutional Neural network Source

Our final model/topology looks like this:
conv(32×5×5) → pool(2×2, 2) → conv(64×5×5) → pool(2×2, 2) → conv(64×5×5) → dense(1024) → dropout(0.4) → dense(48)

1. **conv(X×Y×Z)**: X-filters, Y×Z filter size, All conv layers have ReLu as an activation function.
2. **pool(2×2, 2)**: 2×2 pooling size, 2 stride. Standart size in smaller networks.
3. **dense(Z)**: Z outputs
4. **dropout(F)**: dropout rate (1-F probability that element will be kept)

Second and third conv-layers have 64 filters as we thought this amount is needed to recognize more complicated geometry in characters like **G, Q**, and maybe for further **Æ,Ø,Å**. We added pooling layers for two reasons: to reduce computation time and to add some rotation invariance. Dropout layer was added to reduce overfitting during training.
Given the time we had left for model testing this topology have us best result.

2.5 Component: Datasets

2.5.1 Description

In order to learn our network to distinguish between the characters it needs training. Training is done by feeding images of known objects to the network (labeled data) and telling it how inaccurate it's prediction so that network can adjust it's weights accordingly. This type of training is called supervised training as we guide our network during training process. For network to get high accuracy of prediction a lot of labeled data is needed. Lucky for us datasets like MNIST exists, more info about it later.

description Limitation
As we have limited us to the digits and English alphabet, we will need labeled data for each of these [0..9] + 36 characters; divided into training, test and validation sets. As the concept of classifying only numbers vs all 36 characters does not differ that much, we will first see if we can solve the OCR problem with just numbers. Therefore we only need a dataset containing numbers at first. After that we can proceed our search for a dataset containing all the characters we need.

Dataset

SANS

Dataset created by ourself. Small dataset containing only 3 most used font in machine typing: **Sans**, **Times New Roman** and **Calibri**. These will be the foundation for our CNN as we for the most part will try to recognize characters from there fonts. Therefore we decided to run multiple training steps on this dataset to learn our network the 'etalon' of each character.

FNIST

Another dataset generated by our scripts. This one contain over 1000 free fonts found on the Internet. Here we can find examples of multiple variations of the same character like **bold**, *italics* and regular.

EMNIST-ByMerge

The EMNIST dataset is a set of handwritten character digits derived from the NIST Special Database 19 and converted to a 28x28 pixel image format and dataset structure that directly matches the MNIST dataset. Further information on the dataset contents and conversion process can be found in the paper available at https://arxiv.org/abs/1702.05373v1

Chapter 3

Discarded Method

3.1 Description

This chapter covers the methods tried but not included in the end result, because they showed dissatisfactory results. This chapter is also organized like Chapter 2, but for component description please refer to the previous chapter.

3.2 Text Segmentation

We tried to expand the original approach to be able to detect text with noise, natural images. We tried Stroke Width Transform first, then OpenCv scene text detection. We abandon both approach because of time constrain and difficulties of implementation. We decided therefore to focus on other parts of the project first.

Discarded Method 1: Stroke Width Transform

We tried Stroke Width Transform to do Text Segmentation when original approach gave a decent result. It was originally proposed by Epstein et al 2010 [**epshtein_stroke_2010**]. Since OpenCv does not have this implemented, we tried to implement it ourself. Additional sources was used in our attempt to implement it[**werner_text_????**, **_c++_????**, **bunn_strokewidthtransform:_2018**]. We were not able to finish this, but we think its worth mentioning, since we spent some time on this. The steps of Stroke Width Transform are as followed:

1. **Edge Detection and edge orientation(Done).** We need to have Edge image and orientation of the gradient image. Canny and Sobel was used in the original paper and other sources. This is simple since OpenCv has both Canny and Sobel implemented.

2. **Stroke Width Transform(Done).** Here we had to do more. We have to find a line from a starting point and the angle. We were able to implement this part, but there were some uncertainties. It only worked on black text with white background. That is because the orientation(Sobel filtering) is dependent on it. The paper talks about doing a second pass with inverse image, but we decided to ignore it in order to come farther in the algorithm.

3. **Find Connected Component(Done).** In this point we want to find connected components. The caveat here is that the components need to be connected with

regards to the Stride Length. We used 'One Component At A Time' algorithm to find all the different components. This part we were able to finish.

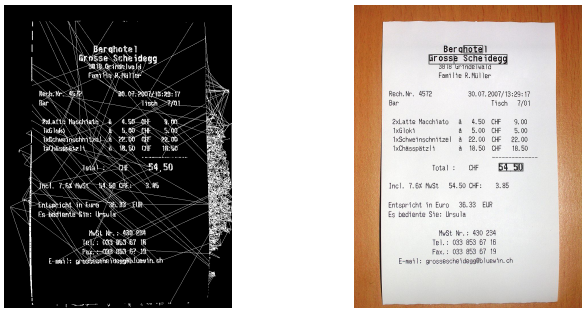
4. **Exclude noise and find letters(not Done).** Since Stroke Width Transform tend to make a lot of noise. The obvious one is making single lines. This part is suppose to exclude this noise and at the same time exclude anything that is not a letter.The theory is, since letters and text in general, usually have the same stroke width, we can use this information do estimate what is a letter and what is not. We were not able to finish this part.

5. **Find lines/words(not Done).** Was not able to get to this part, but idealy it will combine letters to a single line or word.

In cases where the image has a lot of non text objects, it will work fine with it. We ended up discarding this approach since it was too time consuming and decided on working on a simple approach first.

Discarded Method 2: OpenCv Scene Text Detection

OpenCV has its own Text Scene Detection. The approach of this algorithm is to detect text in scene using Classifier and Components Tree, propose by Lukás Neumann & Jiri Matas [**neumann_real-time_2012**]. Since we already discarded Stroke Width Transform to focus on a simple approach, we decided not to use it. We had some problem to get proper result as well.



(a) Unfinished SWT

(b) OpenCv Scene text detection

Figure 3.1: Result of Discarded Segment Text Method

3.3 Preprocessing

We tried different approaches only on find rotation and character segmentation. Line segmentation and dataset matching we didnt try other methods on.

3.3.1 Find rotation

The original thought was to use the Hough Transform to find the lines in a text, and then from that, find the rotation. However because of bad results, we moved to cv.minAreaRect. This too had some limitations on what it could rotate, see Section 2.3.1. These limitation we tried to solve useing a CNN solution, presented below.

Discarded Method 1: Hough Transform
Hough transform is a well known algorithm to find lines, its approach is to see if it can aligne some threshold of pixels on one straight line. To help with this process we will do a simple edge detection algorithm on the image before running it through the Hough Transform.
Bellow, the steps needed to perform this approach are mentioned.

1. **Edge detection** Too make the Hough transform perform better we want to remove unnecessary noise. Canny edge detection algorithm is a robust and fast solution for this.

2. **Line detection** Now perform the actually Hough Transform.

3. **Rotate image** Lastly we need to rotate the image according to the lines from the Hough Transform.

Discarded Method 2: OpenCV minAreaRect() + CNN solution
This approach builds on the approach presented in Section 2.3.1. As cv.minAreaRect returns text rotated in one of following angles [0°, 90°, 180°, 270°], see Figure 3.2 for illustration. We wanted to add a CNN to determine which of the angles is correct and then rotate it accordingly. To train the network we needed a dataset with the corresponding angles as labels, as we could not find one, we decided to create it ourselves, see Section 3.5 for more on this.
We discarded this approach because it gave only a 65% accuracy of the right angle, and we are only working on simple text images with little rotation, hence no need for this addition in our software.
The approach is described below:

1. **OpenCV minAreaRect()** The same approach mention in section 2.3 find rotation

2. **CNN solution - find correct rotation** Now that we have text rotated in one of [0°, 90°, 180°, 270°], we have several options finding the correct angle of the text segment.

   (a) The most straightforward approach would be to find the rotation of first character and rotate the whole segment accordingly (fast and naive)

   (b) Pick N characters and find their rotation, angle with most matches will be our final angle of rotation. This is somewhat slower (depends on N) but gives us bit more confidence than (a)

   (c) Finally we can classify the angle of each individual character before we try to determine the correct angle based on the results. This gives us the highest accuracy but on the cost of time/computation power. This is the only approach able to successfully recognize this type of text: (find better example)LaTeX

Figure 3.2: cv.minAreaRect cannot differentiate between 0°and 180°, and 90°and 270°

3.3.2 Character Segmentation

Since we used Projection Histogram to find lines, we can use it to find character segments as well. We started initially to do that, but later switched to the new approach.
This approach is similar to Line detection in Section 2.3.2. Its weakness is that we may not get the right height of each character, See figure 2.1, meaning we get more background. This is bad becasue later to match the desired format for classification, the resizing/reshaping makes these empty spaces even larger, and the letters even smaller. Consequently it gets more difficult to classify correctly.

3.4 Classification

As mention in Section 2.4, we also tried to use MLP to do classification of characters. We decided to avoid using MLP in our project as it required lots of training data; is not able to detect spatial or minor rotation changes, e.g. a network trained with MNIST with accuracy over 95% was doing poorly in recognizing machine-printed characters, even though they were etalon of a perfect input.

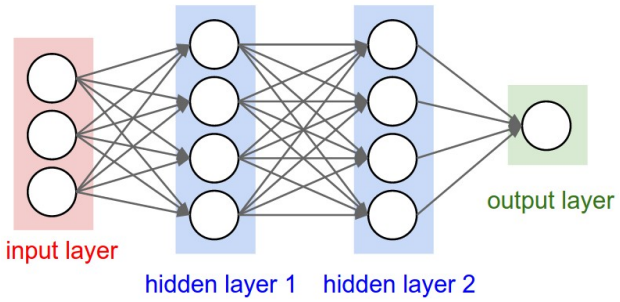


Figure 3.3: MLP Neural network Source

3.5 Datasets

3.5.1 MNIST

MNIST was discarded as we decided to stick with machine printed digits and letters, none of which were included in MNIST. It was a good starting point to test our networks (both MLP and CNN), however to be able to achive a classifier which could classify the english alphabet in addition to numbers ranging from [0-9] we had to go with EMNIST.

3.5.2 ROT*

*ROT** - dataset containing rotated examples of the same dataset, but with labels corresponding to angle of rotation, e.g. [0, 90, 180, 270].

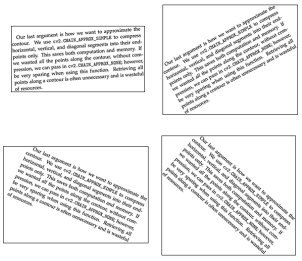
This dataset was discarded as our network showed very little learning progress as the accuracy of predictions was constantly around 65%. This is not the datasets fault, most likely it was due to bad network topology or poor choice of hyper parameters but we didn't have enough time to debug this part of the project and decided to stick to OpenCV's minAreaRect() results.

Chapter 4

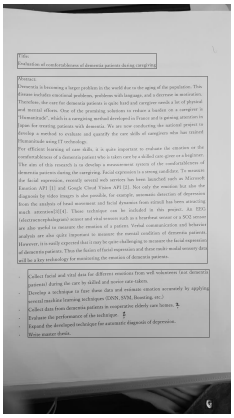
Result - Discussion

4.1 Result: Text Segmentation

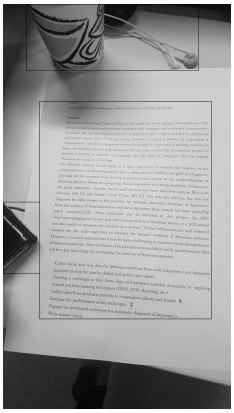
We tried 3 Approaches, Morphological, Stroke Width Transform and OpenCv Scene text detection. We end up with the simple solution Morphological approach. It gave good result, but have a lot of limitations. Only work on black text on white paper, image with little noise and only want text in the image. You can see the result in figure 4.1 We could include a part to see if sections of text or non-text. A strategy to do it can be to compare numbers of connected component, since we know text regions have a lot of letters/component, compare to a coffee cup or a wallet. Canny edge detection can be used to improve it as well.



(a) Good result on pure text



(b) Good result on text on paper with little background noise



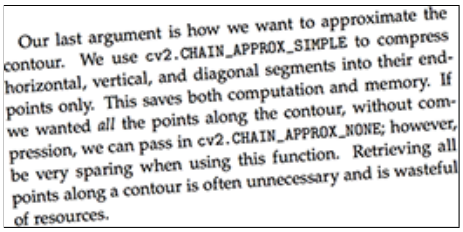
(c) Bad result with many non text object segmented as text

Figure 4.1: Result of the Segment Text

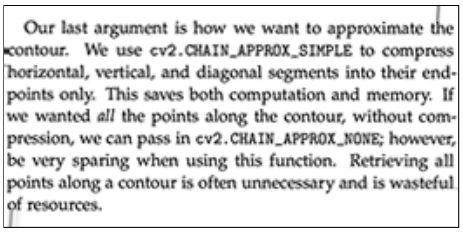
4.2 Result: Preprocessing

4.2.1 Rotation

Rotation only work if the angle of rotation is less that 45 degree. This problem is mention in section 2.3 and an attempt to solve the problem was given in section 3.3.1. Below are a result image of a less skew image.



(a) Region of skew text, after text segmentation

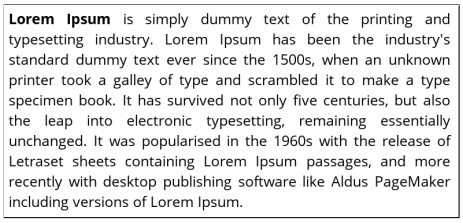


(b) After rotation

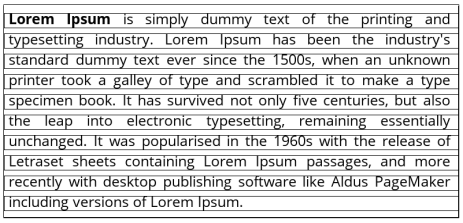
Figure 4.2: Result of rotation

4.2.2 Line segmentation

If the image have no space between lines or is slightly skew, it can cause an error, lines will not be separated. Since our previous steps are to handle these problem our method worked quite well on our test images.



(a) Text Region after rotation

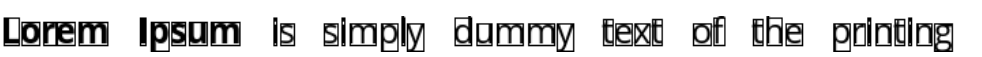


(b) Boundary box of line region

Figure 4.3: Result of line segmentation

4.2.3 Character segmentation

This part worked quiet well, an alternative would be to just using OpenCv findComponent. The steps and result would be much of the same, but we could have dropped cv2.floodFills. A weakness of this approach is if multiple character is merged with one another we would not be able to separate them. A solution for this problem can be to do a morphological opening to separate character better. We did not have test image with that problem, but it came up in latter discussions. There is also another weakness where spaces is not segmented. We tried to solved this in the classification part, by calculate the average space between character



(a) Result of Character Segmentation from extracted line

(b) Result of Character Segmentation from an example line image

Figure 4.4: Result of Character segmentation

4.3 Result: Classification

Result: Classification

4.3.1 Description

Convolutional neural networks are especially good for image classification, because they take local spatial connections into account when they classify. This way it doesn't matter where in the image our object/character is it will be able to recognize it, same yields for rotation, as the CNN classifies based on local spatial connections it doesn't matter if the object is rotated. Hence the classification would be even more robust compared to the MLP.

4.4 Result: Datasets

4.4.1 Description

Chapter 5

Conclusion

Sadegh's thoughts, noe å gå utifra

5.1 Ambition vs reality?

Our ambition was relatively ambitious compared to the end product. We envisioned a software which was rotation invariant, light intensity invariant to some extend; could recognize text with great accuracy. However, it was a far more challenging task then what we expected.

Firstly we started off with a MLP network, which is not the best choice for image classification. Secondly we didn't have much experience with machine learning software, so most of our solutions were a combination of patching existing code and researching what could give a good result for our problems. Additionally, as writing everything in the python environment would be very slow, we decided to write the machine learning algorithms with the TensorFlow API. This too was a challenge, as it takes some time to understand how everything in the TensorFlow environment works.

5.2 Future work?

Since we divided Our approach i 3 part, we can tryimprove them separately.

5.2.1 Future Work: Text Segmentation

On Text Segmentation we would want to be able to detect text in natural images, so a continuation of Stroke Width Transform would be nice.

5.2.2 Future Work: Preprocessing

5.2.3 Future Work: Classification

- LeNet vs Our solution
- prepo

Chapter 6

References

6.1 Datasets

6.2 Code used

6.3 articles

6.4 Online sources

Here is the list of online sources we used for the different part of the project. We probably used more, but these are what we recognize these as heavily influenced online sources:

6.4.1 Text Segmentation

- https://www.danvk.org/2015/01/07/finding-blocks-of-text-in-an-image-using-python-ope
  html
- https://github.com/mypetyak/StrokeWidthTransform
- https://stackoverflow.com/questions/24385714/detect-text-region-in-image-using-opencv
- https://github.com/opencv/opencv_contrib/tree/master/modules/text/samples

6.4.2 Preprocessing

- https://www.pyimagesearch.com/2017/02/20/text-skew-correction-opencv-python/

6.4.3 Classification

- A Guide to TF Layers: Building a Convolutional Neural Network
- The EMNIST Dataset

6.4.4 OpenCv tutorials

We used a OpenCv a lot and found many of the tutorials helpful. Here is a list of few of them, but we may have used more.

- https://docs.opencv.org/3.0-beta/doc/py_tutorials/py_imgproc/py_contours/py_contour_features/py_contour_features.html#contour-features

- https://docs.opencv.org/trunk/d9/d61/tutorial_py_morphological_ops.html