Chapter 1

Method

1.1 Description

present our methods. Bellow we present each component individually and the corresponding methods for them. A short description on what challenges have to be solved for each component will be addressed first.

1.2 Component: Text segmentation

1.2.1 Description

In this part of the program, we want to be able segment out parts of text in the image. We want to then later segment out lines and letter for further classification. In this part wee assume that the image will mainly black text on white paper.

1.2.2 Tried

We where uncertain on how to do this part when we initial started the project. We ended up looking for a lot of different ways to do this part. We ended up trying 3 different approaches with mix result.

1. Simple image analysis technics, using Otsu thresholding and Morphology.(Used)
2. Stroke Width Transform(SWT) to detect text in natural images.(Discarded)
3. OpenCv implementation of Scene Text Detection.(Discarded)

Approach 1: Simple Image Analysis Technics

We was inspired by the 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.(so long the text and background is close to black and white)

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

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

4. **Extract Regions** OpenCv FindContours was used to find the different text regions. We then exclude any regions that is smaller than a selected threshold. The different region are return as coordinate of the different rectangular boxes.

Approach 2: Stroke Width Transform

The second approach we tried Stroke Width Transform to do Text Segmentation, it was original propose by Epstein et al 2010 [**epshtein_stroke_2010**]. Since OpenCv do 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 was not able to finish this, but think we should mention it since we spend some time on it. The steps of Stroke Width Transform is 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 have 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 was able to implement this part, but was some uncertainties. It only work on black text with white background. That is because the orientation(Sobel filtering) are dependent on it. The paper talk 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 are to connect component with the same Stride length together. We used One component at a time algorithm to find all the different components. We was able to finish it.

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 are suppose to exclude this noise and at the same time exclude anything that is not a letter.The theory is, since letter and text all usually have the same stroke width, we can use this information do estimate what is letter and what is not. We was not able to finish this part.

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

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

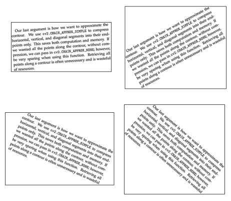
Approach 3: OpenCv Scene Text Detection

OpenCV have it 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 simple approach, we decided not use it. We had some problem to get propel result as well.

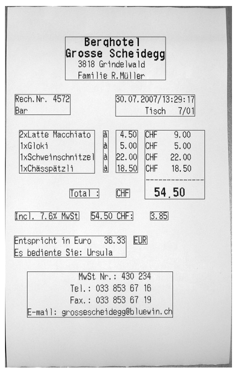
1.2.3 Used in end result

Since we had problem on getting with both approach 2 and 3 we decided on approach 1. It gave us ideal result on most images, but have some problem in images with non text objects.

fill the figure with good examples



(a) Example 1, Good result on just text



Berghotel
Grosse Scheidegg
3818 Grindelwald
Familie R.Müller

Rech.Nr. 4572
Bar

30.07.2007/13:29:17
Tisch 7/01

2xLatte Macchiato à 4.50 CHF 9.00
1xGloki à 5.00 CHF 5.00
1xSchweinschnitzel à 22.00 CHF 22.00
1xChässpätzli à 18.50 CHF 18.50

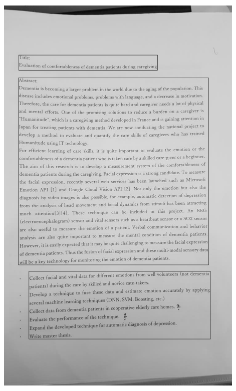
Total : CHF 54.50

Incl. 7.6% MwSt 54.50 CHF: 3.85

Entspricht in Euro 36.33 EUR
Es bediente Sie: Ursula

MwSt Nr.: 430 234
Tel.: 033 853 67 16
Fax.: 033 853 67 19
E-mail: grossescheidegg@bluewin.ch

(b) Example 2, Good result on paper



(c) text

Figure 1.1: Example 3, Part of the text was not included

1.3 Component: Preprocessing

1.3.1 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 it to the right data type for the classifier. Hence the challenges we will have to solve here are:

Challenges

- 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. No need for a specific approach.
- 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.

1.3.2 Find rotation

Approach: OpenCV minAreaRect() + CNN solution
This approach uses convex hull to find the convex hull of the text segments, and then rotating calipers to find the minimum Area rectangle.
One important note here is that methods above operate on binary images, therefor we need to convert our image to binary, possibly using some threshold algorithm. Because we want the convex hull algorithm to find the convex hull of the text segment.
This method is really good at finding the angles, however it does not solve the problem of rotating the text correctly, Figure **??** illustrates this problem. It will only allow us to rotate it along one of the text segments edges. To rotate the text correctly from the result of cv.minAreaRect we decided to try and use another Convolutional neural networks to find if the text was rotated 0, 90, 180 or 270 degrees. For this we had to feed our new network new dataset with rotated images with their corresponding angles. After a quick search on Internet no such dataset was found, so we decided to create it ourself (We describe our approach later in this document).

1. **Binary image** For the Convex hull algorithm to work, the text segment and the background needs to be distinguishable. Convert image to a binary image using OpenCV threshold function and possibly bitwise_not to flip foreground and background colors.
2. **cv.minAreaRect()** Feed our newly generated image to cv.minAreaRect with some additional parameters.
3. **Custom CNN solution - find correct rotation** Now that we have text rotated in one of [0°, 90°, 180°, 270°], we have several options of finding final rotation angle of current text segment.
    (a) Most straightforward would be to find 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 (somewhat slower (depends on N) but gives us bit more confidence than approach above)
    (c) Finally we can find angle for each individual character before we try to classify it, gives us the most accuracy but the price is time/computation power. This is the only approach able to successfully recognize this type of text: (find better example)LaTeX

1.3.3 Find line

1.3.4 Find Symbol/Letter Segmentation

Letter segmentation was similar to Text Segmentation. we added a few steps and removed the morphology part. The additional step was to fill holes in the image, example like 8 and O can give multiple wrong contour. cv2.floodFill was used to solve this problem.

1. Do threshold.
2. Inverse the image since we work on black text on white background.
3. Fill any holes, cv2.floodFills
4. FindContours

1.3.5 Approach: find contour after Filled holes

test text alksjdfh alsjkdbfkas

1.4 Component: Classification

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

Convolutional Neural Network
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.
The basic idea of a CNN is somewhat understood, but the algorithms and how to implement it is still not 100% grasped. Hence because of lack of knowledge, trying to solve the classification problem with a CNN will mostly be done by researching and trying to solve it as it unfolds.

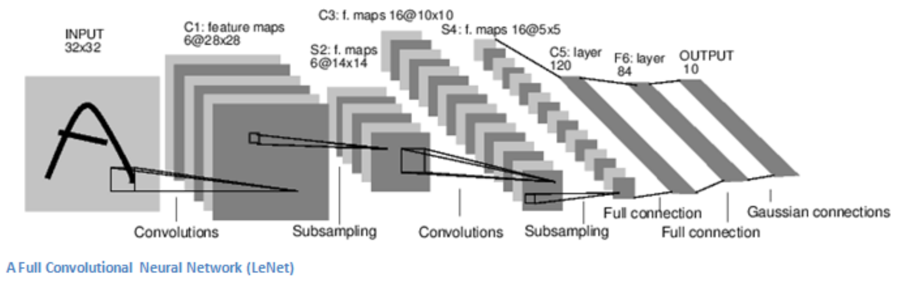


Figure 1.2: Convolutional Neural network Source

CNN - Basics

- Convolutional layer
  - This layer consists of several filters/kernels that are convolved over the image and based on how many filters one uses, the same amount of feature images are made. The purpose of the filters are to grasp some spatial characteristics of the image, and make it available for further processing. The filters here correspond to the weights in the MLP, these are first initialized at random (or with a smart initialization method), then these will be updated as the network is trained. Some features that the filters might find could be; vertical or horizontal lines, they could also find more complex features like faces or animals at later layers.

    In a CNN we can have arbitrary many convolutional layer and arbitrary many filters at each layer. However just like a regular MLP, increasing the number of layers and or filters; can allow the network to fit the training data arbitrarily well, unfortunately at the cost of processing time.
- Pooling
  - The pooling layer allows us to downsize the data. This makes it possible to start with data which are relatively big, and as the data propagates through the network we downsize it. As for the convolution layer the pooling layer can be used arbitrarily many times in a network.
- Fully connected layers (FC)
  - This layer works basically the same way as the hidden layers in an MLP. The pixels of the image is sent as data for each node and they propagate through these layers (usually 2 layers of FC are used) just as they would in an MLP.

1.5 Component: Datasets

1.5.1 Description

Description
Labeled data is needed because our classifiers need to be trained to understand the difference between the characters. This is usually done by training a classifier with a set of training data, labels are needed in our case, since it is a supervised machine learning algorithm we want to use. As the training data is used to train the software, we will need data to test our software as well, hence the need for test data. The test data is used to get a measure of what the error rate of our software is, based on the results we can then tune the hyper-parameters to get a better/smaller error rate. Lastly we will need validation dataset. This is an independent dataset that the software is not familiar with. The accuracy of the software on the validation set will then be a measure of how good the software can classify the characters.

Limitation - proof-of-concept
As we have limited us to the English alphabet and numbers ranging from [0-9], we will need labeled data for each of these 36 characters; 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. Thereafter we will search for a dataset containing all the characters we need.

Dataset
MNIST
This is a dataset containing handwritten numbers [0-9]. It has a training set of 60.000 examples and a test set of 10.000 examples. (ref. reader to http://yann.lecun.com/exdb/mnist/).

1.6 Classification

Deep Neural Network
Multilayer perceptron neural networks are relatively straight forward to code, however the challenging part is to decide on good hyper-parameters and to not overfit our network.
Research has shown that the choices of parameters can have huge effects on the error rate through empirical testing. As empirical testing has shown that some combinations of parameters are better than others, we will also use the same method to find decent values on several of the hyper-parameters. More on this bellow, where a short description of the hyper-parameters follow.
One obvious disadvantage we might face by using MLP is that slight spatial change on where in the image the characters are located, might lead to characters classified differently. This is because there is no spatial connections on a MLP.

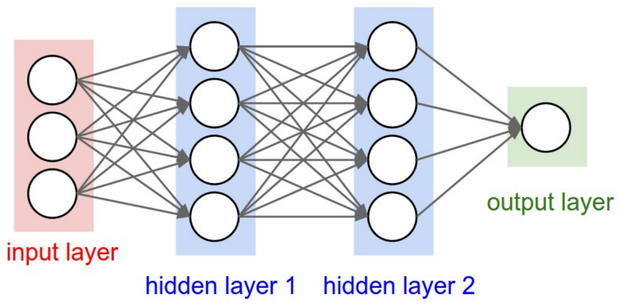


Figure 1.3: MLP Neural network Source

Hyper-parameters

- Number of hidden layers
  - Layers decide how well the software can define the decision borders. Hence increase in layers can have a positive effect, there are also cons with the amount of layers. The more layers, the greater the computational power needed to train the system. We will be using the empirical method to decide how many layers we need
- Number of nodes in each hidden layer
  - Nodes in each hidden layer has the same effect as the number of hidden layers, hence the same applies for this hyper-parameter.
- Activation functions
  - The activation function decides which combination of nodes, with their signals, are allowed to propagate through the network. Here we will be using the *rectified linear unit* (RELU) activation function. This is an activation function that allows propagation if the signal is positive, otherwise it will forward a zero. The reason for choosing this activation function is because this function handles the *vanishing gradient problem* better than sigmoid and a tanh activation functions. More on vanishing gradient problem under "optimization function".

- Loss function
    - The loss function describes how far off the predicted class of the character is from the real class. In our case since we have multiple classes and we are going to use *softmax regression* as the output layer, we also will be using the *cross-entropy loss function.*
- Optimization function
    - The backpropogation will train the weights by Gradient Decent Optimization. However as training with several thousand examples, and then optimizing the weights and run the training process, is too costly resource wise, we will have to implement the *mini-batch gradient decent optimizations*. Same principle as gradient decent optimization, but this way we will find the gradient decent for each batch. As long as these batches are randomly chosen, and the sizes are large enough, (we will be using 100 as batch size), these will represent the entire dataset well enough.
- Learning rate
    - Learning rate is a scalar that decides how large the steps towards the gradient minimum will be, for the weights. Choosing too small of a learning-rate we might risk not reaching the bottom of the graph, we also might get stuck in a local minimum. Choosing too large of a learning rate we might risk never settle down on a minimum.

      For the learning rate we will be using the empirical method too.
- Initialization of the weights and biases
    - Initialization of the weights also seems to be of importance, researchers have found out. This is obvious, as for example setting all the weights to zero, would of course lead to a network with very few active nodes.

      We will be using the initialization of zeros for the biases, not any apparent reason. Based on our research, it seems people have gotten decent results when using this initialization. For the weights we will be using a gaussian distribution, mean=0, standard deviation=1. Again this is also something that we have read should be a good initialization for the weights, no other reason.
- Number of epochs
    - Number of epochs are only relevant when we have a small number of dataset. When we have a small dataset we might want to run the software on the same dataset several times. This might result in overfitting the software to the dataset, therefore it is really important to be careful of the number of epochs, in cases with small datasets.

1.7 Datasets