UiO : Department of Informatics
University of Oslo

UNIK4690 Project

Text Recognition

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

Report

Contents

1 Introduction **2**
- 1.1 Project Description . . . 2
- 1.2 Project Limitations . . . 2
- 1.3 Project Components . . . 2
- 1.4 Report Layout . . . 3

2 Method **4**
- 2.1 Description . . . 4
- 2.2 Component: Text segmentation . . . 4
- 2.3 Component: Preprocessing . . . 5
- 2.4 Component: Classification . . . 7
- 2.5 Component: Datasets . . . 7

3 Discarded Method **9**
- 3.1 Description . . . 9
- 3.2 Text Segmentation . . . 9
- 3.3 Preprocessing . . . 10
- 3.4 Classification . . . 11
- 3.5 Datasets . . . 11

4 Result - Discussion **12**
- 4.1 Result: Text Segmentation . . . 12
- 4.2 Result: Preprocessing . . . 13
- 4.3 Result: Classification . . . 13
- 4.4 Result: Datasets . . . 14

5 Conclusion **15**
- 5.1 Ambition vs reality? . . . 15
- 5.2 Future work? . . . 15
- 5.3 Ambition vs reality? . . . 15
- 5.4 Future work? . . . 15

6 Recognition **16**
- 6.1 Background Knowledges . . . 16
- 6.2 Datasets . . . 19
- 6.3 Code used . . . 19
- 6.4 articles . . . 19

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 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 cover 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 actuals 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 for them. 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 letter 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 was inspired by this online blog Source[4]. 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 (as long as 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 return as coordinate 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 it to the right data type for the classifier. Hence we have 4 additional steps:

Mini-steps

- 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. No need for a specific approach.

2.3.1 Find rotation

Approach: OpenCV minAreaRect() + CNN solution
This approach uses convex hull to find the convex hull of the text, and then rotating calipers to find the minimum area rectangle.
One important note here is that the method above operates on binary images, therefor we need to convert our image to binary, possibly using some threshold algorithm. cv.minAreaRect returns text rotated in one of following angles [0°, 90°, 180°, 270°], see Figure 4.2 for illustration. We will use 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 ourself.

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/or bitwise_not to flip foreground and background colors.
2. **cv.minAreaRect()** Feed our newly generated image to cv.minAreaRect.
3. **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

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 will mean there are some data (most likely text) between 5th and 10th row, and 15th up to another line break. see fig 2.1

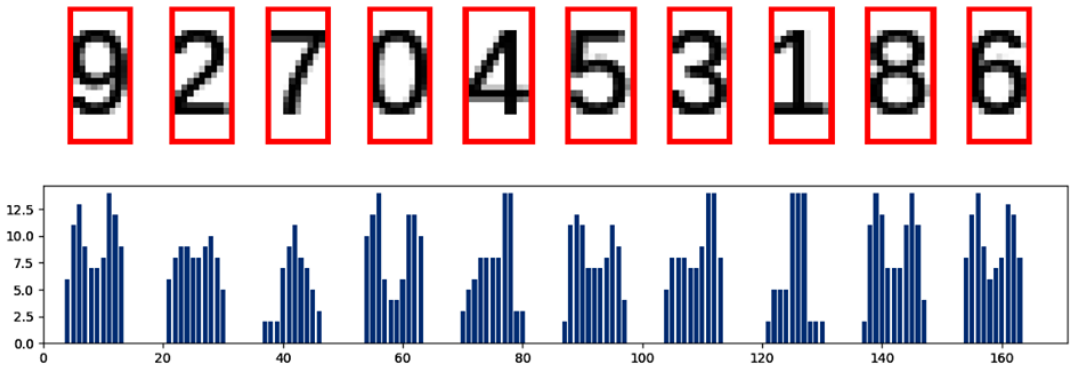


Figure 2.1: Vertical project histogram, same applies to horizontal

2.3.3 Find Symbol/Letter Segmentation

Letter segmentation was similar to Text Segmentation, section 2.2. We added a few steps and kernel shape in the morphologic 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. Do morphological dilation with long vertical kernel, This is to include the dot in 'i'.
4. Add a bolder around the line-image, that is separate elements from edge after previous step. This is to make cv2.floodFills work.
5. Fill any 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

missing method:Data formatting/casting

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 2.2, and a Multilayer Perceptron (MLP or Deep neural network (DNN)), Figure 6.1. 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.
A short description of key points of a CNN is presented in chapter 6. Below are the architecture and variable choices we have made for the CNN,

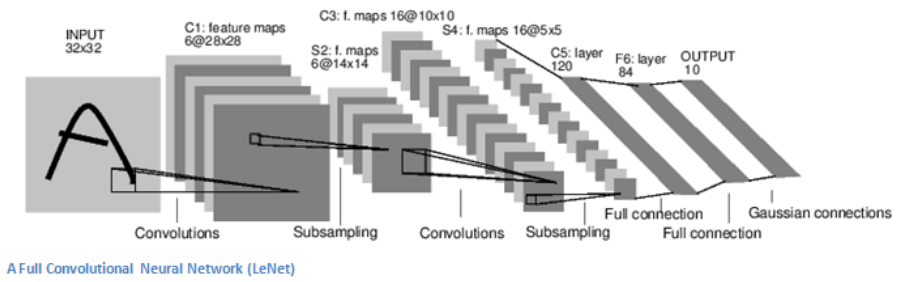


Figure 2.2: Convolutional Neural network Source

Architecture description should follow, choices of variables too
.
.
.
.

2.5 Component: Datasets

include only datasets we have used for the final software
Remove MNIST from this place, and moove it to chapter 3 discardMethod

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.

Limitation - proof-of-concept
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
MNIST
This is a dataset containing images of handwritten digits [0-9]. It has a training set of 60.000 examples and a test set of 10.000 examples. Training set is further divided into 55000 examples of actual training set, while 5000 images are treated as validation set (this number can be changed depending which interface we use to access MNIST data). MNIST dataset is compressed into 4 binary *.gz files, but we do not need to implement any readers/parsers ourself as tensorflow already has two we can choose between. (ref. reader to http://yann.lecun.com/exdb/mnist/).

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 part of the project first.

Other Method 1: Stroke Width Transform

We tried Stroke Width Transform to do Text Segmentation when original approach gave a decent result. It was original propose by Epstein et al 2010 [3]. Since OpenCv do not have this implemented we tried to implement it ourself. Additional sources was used in our attempt to implement it[1, 2, 6]. 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 find connect component. The caveat here is 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 was able to finish.

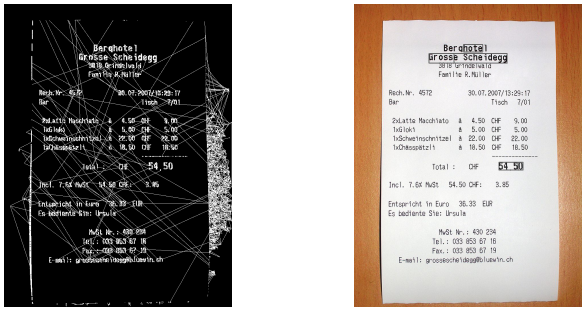
4. **Exclude noise and find letters(not Done)** Since Stroke Width Transform tend to make a lot of noise. The obvious one is making single lines. This part are suppose to exclude this noise and at the same time exclude anything that is not a letter.The theory is, since letter and text all usually have 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.

Other Method 2: 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 [5]. 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.



(a) Unfinished SWT

(b) OpenCv Scene text detection

Figure 3.1: Result of Discarded Segment Text Method

3.3 Preprocessing

3.3.1 Find rotation

Approach: Hough Transform
Hough transform is a well known algorithm to find lines, its approach is to see if it can align 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.

3.3.2 Character Segmentation

Since we used Projection Histogram to fine line, we can use it to fine character segment as well. We started initially to do that, but later switch to the new approach. The approach is similar to Line detection in section 2.3.2. It weakness is that we may not get the right height of each character, meaning we get more background. See figure 2.1

3.4 Classification

As mention in 2.4, we also tried to use MLP to do classification of letter. We gave a bigger overview of the different Hyper-parameters, strength and weakness of MLP in section 6.1.2. We decided to avoid using MLP in our project as it required lots of training data, is not able to detect spacial 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.

3.5 Datasets

MNIST was discarded as were 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) but it was time to move on.
ROT* - dataset containing rotated examples of another set (*) but labels corespont to angle of rotation. In our case we only had 4 classes [0-3], one for each angle in [0, 90, 180, 270]. This set was discarded as out network showed very little learning progress as the accuracy of predictions was constantly around 65%. Now that is not datasets fault, most likely it was due to bad network topology or poor choice of hyper parameters but we didn't had enough time to debug this part of project and decided to stick to OpenCV's minAreaRect() results.

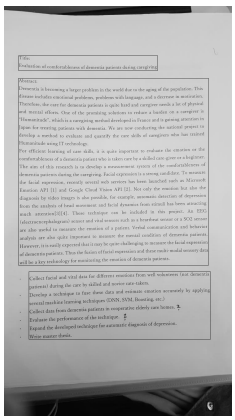
Chapter 4

Result - Discussion

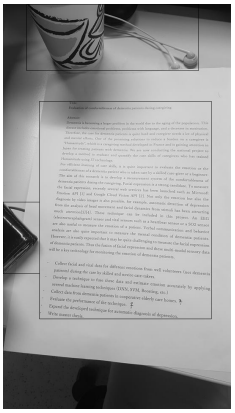
4.1 Result: Text Segmentation

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

Figure 4.2: cv.minAreaRect cannot differentiate between 0°and 180°, and 90°and 270°

4.2.2 Line segmentation

result: 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

Figure 4.3: 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

5.1 Ambition vs reality?

5.2 Future work?

Sadegh's thoughts, noe å gå utifra

5.3 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 machinelearning 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 machinlearning algorithms with the TensorFlow API. This too was a challange, as it takes some time to understand how everything in the TensorFlow environment works.

5.4 Future work?

- LeNet vs Our solution
- prepo

Chapter 6

Recognition

6.1 Background Knowledges

6.1.1 CNN - Basics

AS mention i section 2.4 CNN have a few component to tune. Here we list up them and discuss the different component of a CNN.

- 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. First layer filters are able to spot simple geometry like straight or curved lines, but the deeper we dive into networks topology the more complex shapes the filters can recognize. It is not unusual that last layers on large networks can differentiate between faces, animals, objects.

    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, mostly during training, but also during prediction phase.
- Pooling
  - The pooling layer allows us to downsize the data. This makes it possible to start with images which are relatively big, and as it's 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. Pooling makes our network rotation invariant to minor changes in angle, as outputs max/min value of a (pooling-size) block regardless of where in the block this value is.
- Fully connected layers (FC)
  - This layer works basically the same way as the hidden layers in an MLP. The output of last convolution layer is then flattened and send to N fully

connected layers (also sometimes referenced as dense layer). At the last layer we have same number of outputs as we have classes, just like in regular MLP.

6.1.2 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, or rather MLP can't connect same object on the image but slightly shifted to right/left as it appears like completely different objects due to the fact MLP is build up.

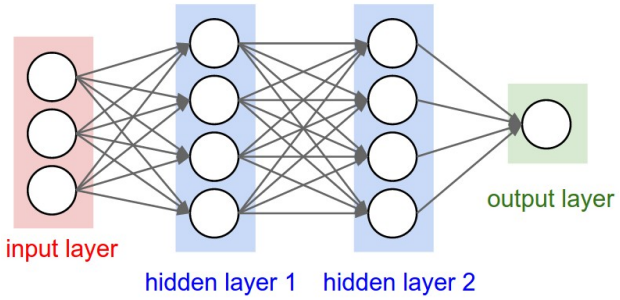


Figure 6.1: MLP Neural network Source

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 backpropagation 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 optimization.* 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.

6.2 Datasets

6.3 Code used

6.4 articles