AML Final Project: Creating an Image Input Calculator

Timothy Jay Herbst, Fenja Kollasch, Duc Anh Phi

September 30, 2019

Abstract

This paper discusses a program that can solve a variety of handwritten mathematical equations, using state-of-the-art computer vision and machine learning methods. Given a single image or a stream of images, the program is able to detect, recognize and compute written equations in real time. We came up with a series of steps to tackle this challenge. Initially, we remove noise with a preprocessing step. Then, each handwritten character is segmented from the image and its size, shape and positional information is used to infer its position in the equation. This also allows simple characters (e.g. points and bars) to be classified. The remaining characters are recognized using an *AlexNet* Artificial Neural Network. Knowing the position and the class of the characters, allows us to reconstruct the equation as a string and pass it to the Symbolic Math Solver `SymPy`. By being open source this project demonstrates, how such a problem may be tackled and could give a basis for many other projects on a variety of topics.

Contents

1 Motivation

Author: Timothy Jay Herbst

Computers are superior to humans in many everyday disciplines. A Calculator, for instance, can produce the solution to most simple mathematical problems faster than the user can enter it. This has inspired us to search for a solution to this flaw in a common household item. If we want to improve upon calculators, we will not have to increase the speed at which computers calculate. Instead the focus should lie on improving the efficiency of entering an equation into the calculator. We decided to make a calculator able to find the solution to an equation, when given an image of it. Ideally the project will some day be usable as a smart-phone app, where the user may scan any equation and instantly get a result. In this paper, we explain how such a calculator can be designed.
Similar projects have been successfully implemented [1]. Our project places its focus more on the complex task of using real and often noisy images. We hope that this serves as an inspiration to solve similar issues, and that the code provided proves useful.

2 Introduction

Author: Fenja Kollasch, Duc Anh Phi

They way our brain solves handwritten formulae is similar to our program. We see the equation `(input)`, focus on only the necessary information we are getting from the view `(preprocessing`, extract each character `(segmentation)`, put them in the right order `(line assignment and ordering)`, and recognize them as what they represent `(recognition)`. The focus is on what is written, so other things like the background are disregarded `(preprocessing)`. We skim over the equations from left, to right, top to bottom `(ordering)` and compute the result in our head along the way `(solving)`. Usually all the steps before finally computing the result, like segmentation and recognition are done effortlessly by our brain - the actual solving takes more time. Interestingly this phenomena is reversed with a machine. Calculations are rapid and easy to implement. However, reconstructing the written formula, by reading and interpreting visual information takes much more effort. It entails challenges in the areas of visual computing and machine learning.

For once, several visual computing methods need to be applied to the image. The application has to segment the input image into one segment per character. These segments have to be put in an order which correctly represents the written formula. This ordering step is not trivial as there can be multiple lines of written equations with varying height and straightness. Each line can also contain fractions. They pose a problem, as their nominators and denominators could be assigned to the wrong line. Making things more complicated, fractions can be nested too.

Machine learning methods can be applied for the recognition of each segmented character. Luckily, the recognition of handwritten symbols is a well-known problem that has been approached by various researchers before. LeCun et al [2] for instance discussed different methods to recognized handwritten digits in 1998. With this work, they made a huge contribution by providing this

MNIST database of handwritten digits [3] that is used for many toy examples in machine learning. In fact, the task that is solved around the MNIST dataset does not differ very much from our own classification task. Symbols that can appear in a calculator input will mainly be digits from 0 to 9. Additionally, mathematical operators, brackets, latin symbols and greek symbols could be found in a mathematical formula. For this work however, we will focus on recognizing only digits and simple mathematical operators like plus, minus, multiplication-, and division symbols, as well as brackets.

After these steps are completed, we end up with a reconstructed formula which we pass to a symbolic math solver, in our case `SymPy`, which is an open source python library. We then present the recognized equations with the computed results.

We implemented the program to harnesses the local webcam and use the stream of frames as the input. The whole process of preprocessing, segmenting and ordering each frame is computed and displayed in near real-time. The whole application flow is summarized in the figure below:

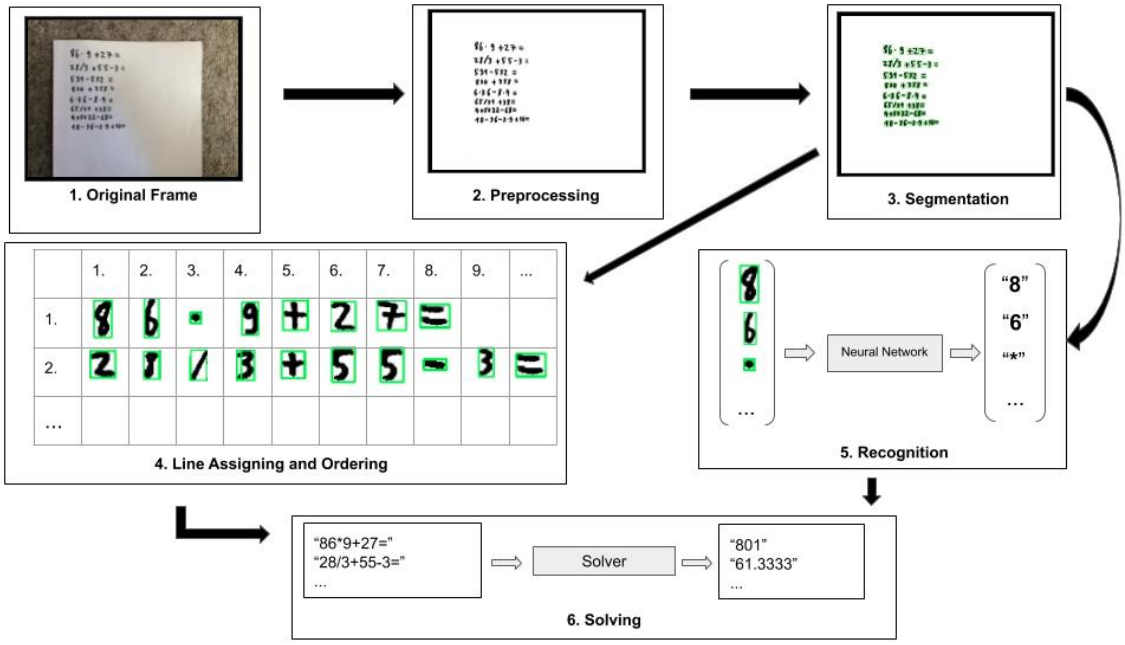


Figure 1: Application Flow

3 Previous work

Author: Fenja Kollasch

As mentioned in the introduction, we are approaching a number of common problems in computer science. Therefore, an amount of related work exists. The motivation of this report declares that we took inspiration of a blog post [1] proposing an application with a similar workflow.

The recognition of handwritten symbols beyond digits has received attention as well. Shortly after the introduction of MNIST, the data set was extended to EMNIST [4] including not only digits, but also handwritten characters. Meanwhile, one can find various data sets with handwritten symbols, even specialized on mathematical symbols [5]. There are existing Kaggle challenges for solving this classification issue [6] which has a rich amount of solutions. So far, there have been many approaches on the topic by using machine learning as we did [7] [8]. Additionally, there are different approaches where the classification of the symbols succeeds over the geometrical properties of the symbols [9].

4 Theory

4.1 Classification with neural networks

Author: Fenja Kollasch

The state-of-the-art for automatically recognizing the content of an image file is to solve this issue with *artificial neural networks*. In theory, this design pattern bases on the neuron structure of the human brain as described by neuroscientists. According to this concept, a neural network consists of different layers with a certain amount of nodes which are also referred to as *neurons*. Each neuron receives an input of different parameters that either come from the previous neuron layer or they are given as initial input. A non-linear function is applied to this input that creates an output value. This value is lead to the next layer where it acts as an input again. The first layer of a neural network is often referred to as the *input layer*, while the last layer is called the *output layer*. Between these layer lies a freely chosen number arbitrarily large *hidden layers* (see figure 2).

Classifying an image means to have a neural network predict how likely the image shows an object from an acquainted catalog, so-called *labels*. As an input, the neural network receives an array including this image's color values. Therefore, the amount of neurons in the input layer needs to be conform with the size and the number of color channels of the input image. The first layer will now produce a certain number of output values that are passed to the next layer. How many output values are generated and to which neurons of the next layer they are passed, depends on the network architecture. When the input meets the last layer of the network, the output layer will calculate for each known label the probability on which the image displays something that can be described with this label. This means, the final output of the network is a vector mapping the probabilities to the known labels. To make a distinct classification of the image content, one would assign the label with the highest probability to the image. Furthermore, it is also possible to design a network that takes not only a single but a set of images as an input and predicts labels for all images. Thus, it will not return a vector with probabilities, but a matrix.

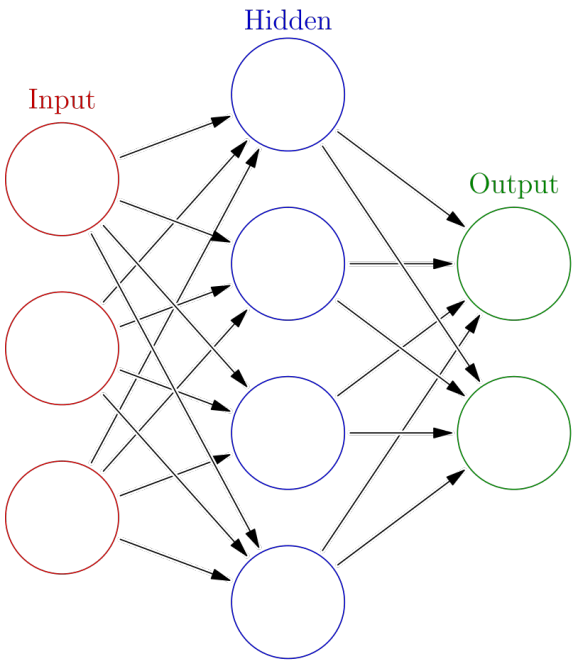


Figure 2: A simple artificial neural network

The just described *forward pass* is used to make a label prediction for an input. However, if data is lead through an untrained network, the prediction is no better then a guess. To make a reliable network, it is necessary to *train* it, usually by passing a high amount of labeled images and comparing the predictions to the labels. The error calculated through the difference between prediction and ground truth now gets *backpropagated* through the network by regarding the gradient at the previous neurons and adapting the weights at the neuron's cost function to minimize the error. This is an iterative process, which is why a training cycle usually has multiple *epochs* where the whole training data set is passed forward and backward. Furthermore it has been proven beneficial to not lead the whole set at once through the network, but to take smaller batches that include randomly chosen samples from the training set. There are various optimization algorithms for the backward pass, as well as different metrics to estimate the error between ground truth and prediction.

4.2 Image processing and segmentation

Author: Fenja Kollasch

The plain image of an equation that our program takes as an input is not suitable for automatic classification. The single written characters need to be extracted while the order and placing of each character must be remembered. Furthermore, a number of image transformations is applied

to clean the image from noise and make it easier to get recognized by the network. To perform the symbol segmentation and preprocess the images correctly, we use concepts from visual computing and image processing.

4.2.1 Adaptive Threshold

Author: Timothy Jay Herbst

The "Adaptive Threshold" function [10], found in the `OpenCV` library is very useful during Preprocessing. This function operates like most threshold functions, in that it compares a pixel to a threshold T. If the pixel is found to have a value higher than the threshold T, we set it to 255. Otherwise the pixel is set to a value of 0 [11] This way it is possible, to easily "classify" a pixel as writing (0) or not (255).
What makes this function work so well, is that for every pixel the threshold $T(x, y)$ is calculated separately. While there are several different ways, how we can calculate this, it was found to work best for our problem, by a simple process [2]:
The threshold $T(x, y)$ is set to the gaussian weighted sum of every pixel within a window with side length 11 (extends 5=(11-1)/2 pixels out from (x,y)):

$$T(x,y) = \sum_{\Delta x=-5}^{5} \sum_{\Delta y=-5}^{5} -2 + G_{(\Delta x,\Delta y)} \cdot img(x+\Delta x, y+\Delta y) \tag{1}$$

$$\text{with} \qquad G_{(\Delta x,\Delta y)} = \alpha \cdot \exp(-(\Delta x^2 + \Delta y^2)/(2*\sigma)^2) \tag{2}$$

$$\text{with} \qquad \alpha = 1/\Big(\sum G_{(\Delta x,\Delta y)}\Big) \tag{3}$$

$$\text{with} \qquad \sigma = 0.3 \cdot ((11-1)/2-1) + 0.8 = 2 \tag{4}$$

Here α is the normalisation factor and σ is the standard deviation (recommended by `OpenCV`) for a box size of 11.
The values mentioned above were found to work well with most of the images tested. If a particularly large or small image would be tested, we would need to adapt these values

4.2.2 Contour detection

Author: Fenja Kollasch

A significant part of the symbol segmentation is the detection of contours. In simple terms, a contour is the boundary of an object in the image. The library `OpenCV` defines a contour as a "curve joining all the continuous points (along the boundary), having the same color or intensity". These contours are essential for later shape analysis, object detection and recognition.

On binary images, contours can be detected by collecting pixels having a different color than their neighbors. Since a pixel can only have one of two color values (black or white), we can use a naive approach. By iterating over all pixels in the image, one could compare the color of pixel $p = (i, j)$ to the color of pixels $(i-1, j-1)$, $(i-1, j)$, $(i-1, j+1)$, $(i, j+1)$, $(i+1, j+1)$, $(i+1, j)$, $(i+1, j-1)$, and $(i, j-1)$. If one of these pixels has a different color than p, p is part of the contour.

Beside this naive approach there exist different algorithms working with the same principle but using more sophisticated methods of regarding the pixels than looping over the image.

By taking the contours of an image fragment, we have successfully segmented a connected symbol. However, there might be symbols in the picture consisting from multiple contours, such as equal signs or the digit `8`. Therefore, it may be necessary to group some contours that display a single symbol. In contrast, some of the extracted contours may not belong to symbols but are results from noise or misplaced strokes on the written equation. These contours should not be regarded. By filtering out all contours with a thickness below a certain threshold, most of misplaced contours will be ignored.

4.2.3 Measuring thickness and symbol length

As one can derive from the previous section, computing the thickness of a symbol is necessary to remove unwanted noise.. The thickness can be computed by dividing the area from the contour image that is filled with white pixels through the length of the symbol. To moreover estimate the length of the symbol, the symbol needs to be *skeletonized*, what means reducing the contour temporally to a thickness of 1 pixel. By counting the pixels in the skeletonized contour, one can receive the symbol length.

Skeletonizing works by iteratively removing the contours of the shape and recalculating the new contours. The contours get removed until there is only one connected contour left. This will be the 1 pixel thick contour of the symbol.

5 Design of the Image Input Calculator

5.1 Data Collection

Author: Timothy Jay Herbst

The user can have the program analyse images from a variety of sources. The library `OpenCV` is capable of reading image files, video files and may also use a webcam, if present. This enables a smooth user experience and allows realtime analysis of an equation.
Sources of images for testing and optimizing this Calculator were created by writing equations on paper or whiteboard with a variety of different writing utensils. These include ballpoint pens, pencils, felt tip pens, fountain pens and more. This was done to ensure a variety of colours, thicknesses and styles of writing.
Images and videos were then taken of these equations, or the image was placed in front of a webcam for live testing.

5.2 Preprocessing

Author: Timothy Jay Herbst, Duc Anh Phi

Our program receives the original unaltered image. It is difficult to detect significant features, such as the symbols that make up the equation from such an image. Therefore, it is necessary to do some preprocessing, before further steps make sense.

This raises several issues. Some improvements we could make include:

- Correction for different quality of camera, lighting, etc.: The incoming images vary significantly due to changes of lighting and may come from different cameras. Correcting for this, is not a simple task.
- Simplification of the input dimensions: Explicitly we want our image to be a binary black-white image, with the colours black (0) for text and white (255) for background.
- Connectedness of the symbols: For later steps, it is useful to not divide any symbols into multiple parts. In the same manner, we do not wish for two different symbols to be connected.
- Background contour removal: Oftentimes when taking an image of an equation there will be other objects in the image (e.g. pencils, the border of the paper, etc.).
- Removal of noise: Often the background (e.g. paper or whiteboard) is not uniform and this can create spots in the image. (The human eye is very good at filtering these out.) Similarly reflected light sources can cause noise. This happens more often on whiteboards than paper, because they tend to reflect more easily.

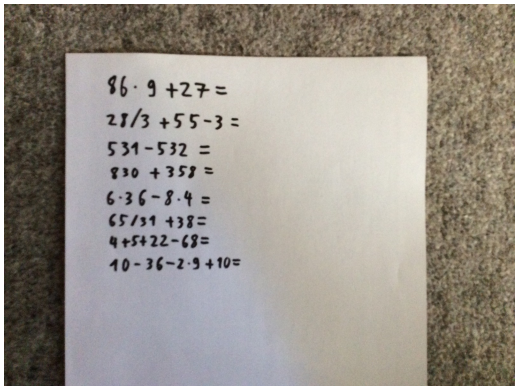
We correct for (most of) these issues in three separate steps:
First we use a few select image processing steps to deal with most of the issues. In a second step, we remove the contours that can be attributed to background objects. Lastly, we correct for reflected light sources and some types of noise.

5.2.1 Simple Preprocessing

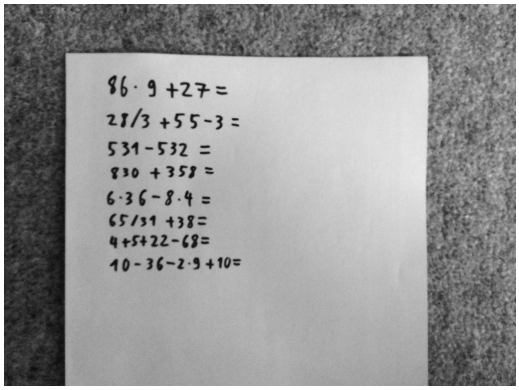
Author: Timothy Jay Herbst

The image our program receives as input is likely not easily analyzed as is. We therefore preprocess the image with a few select operations before continuing with our algorithm. It should be noted here that there is a great variety of different algorithms that work well. The algorithm we will be presenting here is the one we found to work best for our purpose.
It operates in five steps (see fig: 3):

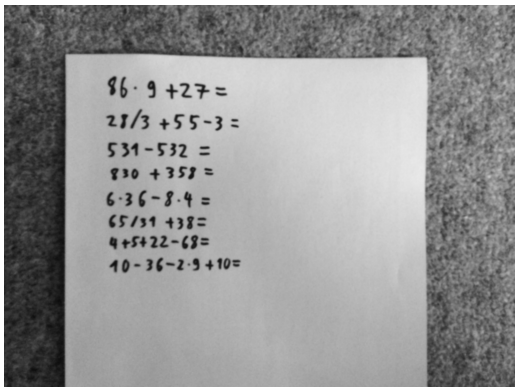
- Grayscaling: At first, we grayscale our image. Here, we reduce the BGR (blue, green, red) image into one with a single dimension per pixel. This is useful, because it simplifies the image and allows us to more easily analyse it.
- Gaussian Blur: Here, we apply a Gaussian blur kernel on the image to smooth it out. This reduces the noise.
- Morphological Opening: We apply a filter that removes a specific type of noise: small dots. This is done by applying an erosion followed by a dilation filter.
- Adaptive Gaussian Threshold: We add a filter that applies a threshold. Any pixel below this threshold gets set to 0 and any pixel above it gets set to 255. This threshold differentiates the symbols from the background well.



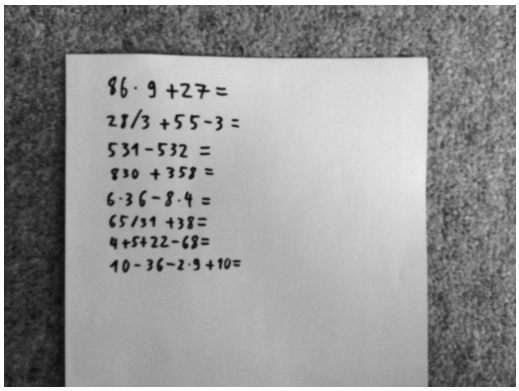
(a) Original Frame



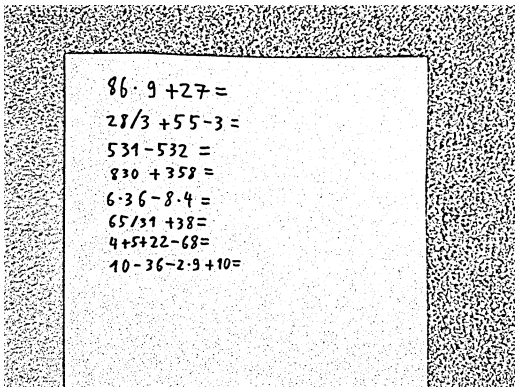
(b) Grayscaled



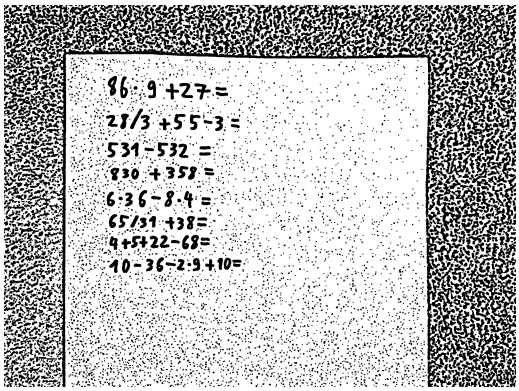
(c) Gaussian Blur



(d) Morphological Opening



(e) Adaptive Gaussian Thresholding



(f) Dilation

Figure 3: Various steps performed during the simple Pre-Processing Process

- Dilation: After the previous steps, it is possible that some of the symbols have decomposed into several parts. This allows for easier contour detection. However, this makes the contour more difficult for the classifier to recognise. Because of this a new image without this step is created for classification. Therefore, it is useful at this point to thicken the contours.

5.2.2 Border Removal

Author: Timothy Jay Herbst

After these pre-processing steps, we have a fairly good image. However, very often the shape of background objects register as contours. THis section explains how to remove the contours of background objects. For this, we make use of a simple observation: symbols tend to be separate from other contours, while background contours tend to connect to the border of the image via other contours, see fig 4. Using this knowledge we can create a mask, which removes most background contour objects.
We determine the mask in the following steps:

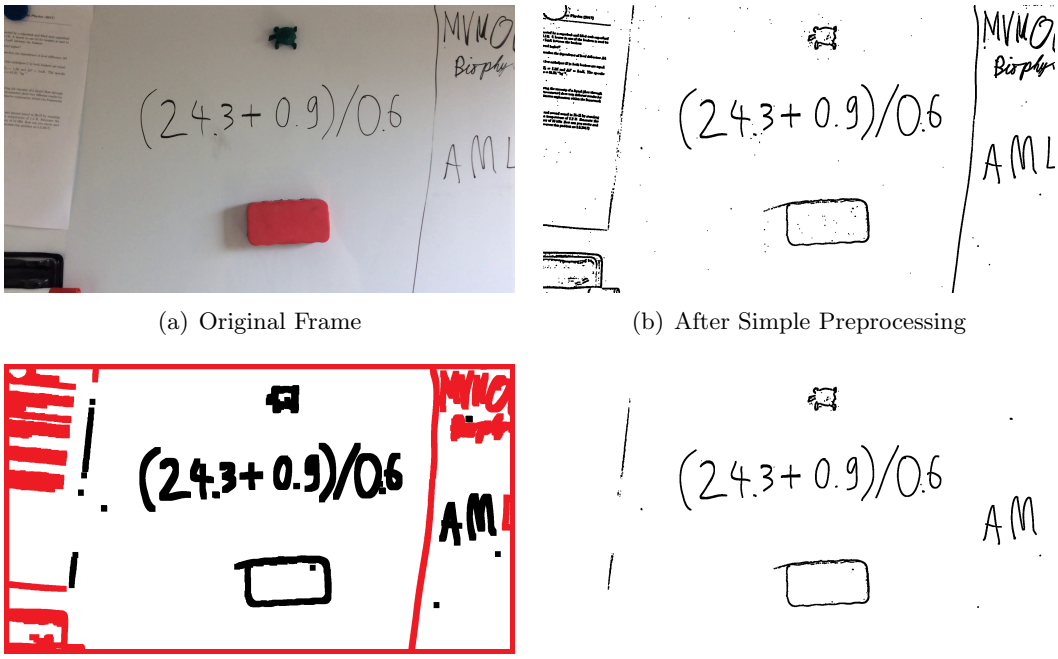
- Simple Preprocessing: At first we need to simplify the image. We use a similar algorithm as used in the section "Simple Preprocessing" for this.
- Adding a Border: For the next step we add a black border to the image. Our aim is to remove all contours which connect with this border.
- Strong Dilation: We now repeatedly dilate the image. Our goal here is to connect all contours which are close to each other. Unfortunately this makes our contours unreadable. This is not a problem since in this step we simply want to determine the contours we want to keep.
- Floodfill Border: All pixels, which connect only over black pixels to the border get removed from the mask. Thereby most contours caused by objects near the border, get removed.
- Remove Border: Since we changed the image size by adding a border, we now have to remove it.

In this manner we determine the mask. Our resulting image is determined, by having all pixels white, except those, that are black both in the mask and in the simple preprocessing. This removes background contours, without making our image unreadable.

5.2.3 Bright Noise Removal

Author: Timothy Jay Herbst

Bright light sources may reflect off the surface on which the equation is written. This occurs especially often when using a white board, but also on paper. Unfortunately, the adaptive threshold function cannot differentiate between the writing and the bright reflections. Fortunately, the reflected light sources all have a common trait: They are significantly brighter than the average image. It is therefore possible to create a fairly simple mask which simply removes all pixels which are significantly brighter than the average pixel of the image. After applying this mask, the image no longer contains contours caused by reflected light sources. A positive side effect is, that noise caused by imperfections on paper get removed because of their brightness (see fig. 5)..



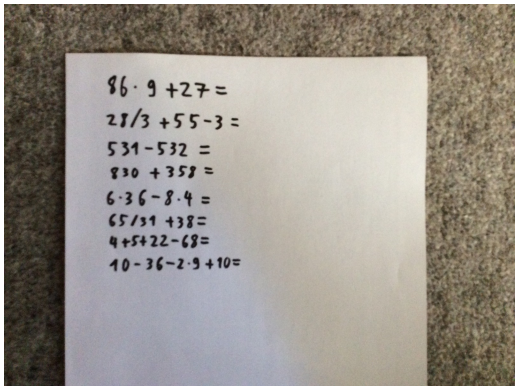
(a) Original Frame

(b) After Simple Preprocessing

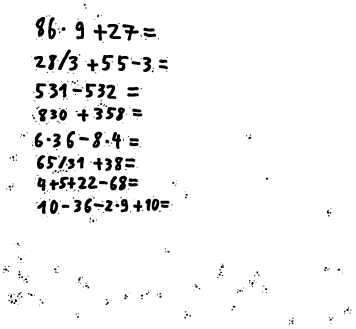
(c) Border Removal Process

(d) Resulting Image

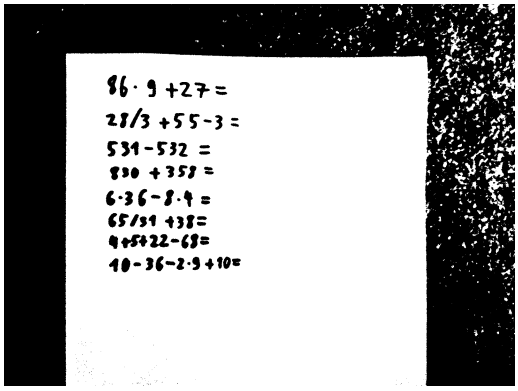
Figure 4: Performance of the Border Removal Mask. The Border Removal Process (as seen in subfigure(c)) allows for the removal of all contours which connect to the border after the strong dilation (red pixels). Any pixels that are black both after the Simple Preprocessing step and after the Border Removal Process remain black. Please note that some contours caused by objects remain, because they don't connect well with the border of the image.



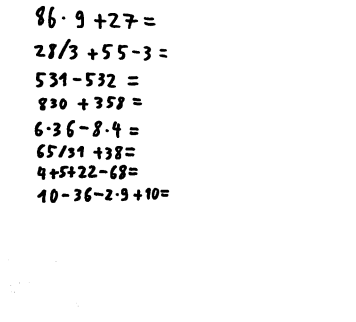
(a) Original Frame



(b) After Simple Preprocessing and Background Removal



(c) Reflection Removal Mask (Black pixels will not be removed)



(d) Resulting Image

Figure 5: Performance of the Reflection Removal Mask.

5.3 Segmentation

Author: Duc Anh Phi

In the segmentation step, we extract written characters from the image. These extracted pixels are then further evaluated - they make up the building blocks of our program. In the following and sections to come, we are going to detect, group, order and classify these segments, in order to reproduce and finally solve the written equations.

5.3.1 Contour Detection

Author: Duc Anh Phi

After the preprocessing is done, we can deploy an algorithm to detect contours from the image.

The underlying algorithm we used for contour detection is from [12]. This paper presents a border following technique for binary images. "It derives a sequence of the coordinates [...] from the border between a connected component of 1-pixels (1-component) and a connected component of 0-pixels (background or hole)." Not only does the algorithm reliably detect contours, but also it gives us hierarchical information about them.

This will be important for making sense of nested (contours completely inside of other contours) contours later.

5.3.2 Removing Small Contours

Author: Duc Anh Phi

Not all segmented contours represent handwritten signs. Some are noise which passed the preprocessing. Usually, this noise is rather small, e.g. points or thin lines. With the assumption that handwritten signs have approximately equal line thickness, we can confidently remove all contours which have a significantly smaller line thickness. For this to work well, we have to find a good threshold line thickness. Computing the average line thickness is costly and might be inaccurate for images with lots of small noise. To make sure not to include the line thickness of noise, contours of median size are selected. Then, we compute the line thickness for each of these selected contours. The median out of these computed values is our threshold line thickness.

Compute Line Thickness (Author: Duc Anh Phi)
The algorithm for computing the line thickness of a contour was taken from [13]

1. We start with a binary image (see fig. 6), with pixel values either zero (black) or 255 (white).
2. Summing up the image gives us the sum of all white pixels, as black pixel are zero. In other words, the sum gives us the area of the whole white line (in pixel times 255).
3. After skeletonizing the image, the white line becomes one pixel thick (see fig. 6). The resulting skeleton is a 1-pixel-thin approximation of the original image.
   Summing up the skeleton gives us the approximate length of the line.

$$Area = Sum/255 \tag{5}$$

4. With the line area and length, we calculate its thickness as:

$$Thickness = Area/Length \tag{6}$$

(a) Original Contour (b) After Skeletonizing

Figure 6: Computing the Line Thickness

5.3.3 Group Contours

Author: Duc Anh Phi

Some contours belong together to form a mathematical symbol or structure. In the following, we replace the individual contours with their grouped versions.

Handling Nested Contours For example, a handwritten '8' usually consists of 3 contours: The outer frame and the inner two holes, which are detected as separate contours. To not confuse these holes as distinct math symbols, we group them together in a single, custom contour object. A contour object has the parent or outermost contour stored in its "contour" property. The contours representing holes are stored in the "holes" property.

Fractions Fractions are problematic, as they have an inherent order to them – especially nested fractions. This order must be taken into account. Additionally, fractions have multiple (at least 2) expressions stacked vertically, which may span several lines. This can cause problems, when evaluating multiple lines of equations, because the numerator or denominator may be accidentally assigned to the wrong line. The solution to these problems is to simplify fractions to single contours

(their bounding boxes), while still preserving positional information about each grouped element. Initially, we have to find the fractions. How we find fraction bars is shown in 5.5.1. After sorting the fraction bars (ascending) by their width we apply the following algorithm to each bar iteratively, starting with the smallest one:

- extract contours above and below the fraction bar (inside the acceptance area) and create a custom fraction object, which references them in the "numerator" or "denominator" property. The acceptance area is inferred from the bar width.
- sort each contour in the "numerator" or "denominator" property by their x-coordinates – in this way, we ensure a correct order
- create a bounding box around the whole fraction
- create a contour object based on the bounding box and the Fraction object

Equal Sign A written "=" sign consists of two vertically stacked horizontal bars. We group both contours in a single contour object. How we find equal bars is shown in 5.5.1. Once an equal bar is found, we create a bounding box around itself and its counter part. A new contour object is created with the bounding box as its "contour" property.

5.4 Line Assignment and Ordering

Author: Duc Anh Phi

At this stage, we have a list of (grouped) contour objects. We now face the problem of bringing them into correct mathematical order, based on their position in the image. This task seems trivial for one equation: simply sort the contours by their x-coordinate. However, we aim to solve multiple equations, which are each written in a separate new line. So, looking at the x-axis is not enough. We have to take the y-axis into account, too.
When observing lines of equations, we noticed that all symbols within the same line have a similar y-coordinate value. However, that value will change particularly between lines. With this observation, we developed the following algorithm to separate lines:

1. Initially, contours are sorted by their y-coordinate. In this way, contours which are in the same horizontal line are closer to each other.
2. Afterwards, the average y-distance between neighboring contours is calculated:

$$AverageNeighborYDeviation = 1/N \sum_{k=2}^{N} |y_k - y_{k-1}| \tag{7}$$

   We use this y-deviation as a threshold value to determine whether a new line has started.
3. In the sorted list of contours, each contour (except for the first) is compared to its predecessor for calculating the y-coordinate deviation between them. If the calculated value is within the y-deviation threshold, both belong to the same line. However, if the calculated value is greater than the reference y-deviation, a new line starts with the current contour.

4. After extracting lines, each contour in a line is then sorted by their x-coordinate.

For the Line Assignment we have developed several algorithms. In the following, we discuss the most interesting, the ones able to tackle a particular problem and the most efficient ones.

5.4.1 Determining the Direction of Writing

Author: Timothy Jay Herbst

When writing an equation, most people tend to write roughly along an "invisible line". Therefore, when taking an image of said equation, we can expect it to adhere roughly to the line. We can use this to determine the order of the symbols.
There are, however, two issues with this:
First, unlike a computer, a human will not write their symbols precisely on a line. Oftentimes, a symbol will be significantly above or below the line. The situation gets even worse, if we take a look at exponents, division bars and similar.
Secondly, the "invisible line" may not be along the horizontal of an image. This may be because of sloppiness of the writer, or because the image was taken at an angle.
Because of this, it may be very useful to determine the direction of such an "invisible line". A good algorithm was found to solve for this problem. However, in most images we found this "invisible line" deviates by less than 10° from the horizontal. Since the code we used, sometimes caused problems when looking at multiple lines of equations, it was removed. The interested reader may find it in the git-repository under the branch `mergeordering2` in the file `ordering2.py`.

5.4.2 Assigning Lines to Proposed Line Positions

Author: Timothy Jay Herbst

Another method how to accurately order the contours into lines can be done by proposing y-coordinates for lines and then assigning contours accordingly. This can be done by checking for each proposed line, whether or not a given contour is within a certain radius of the line.
For this procedure, we at first have to find a rough estimate of the height of a line. This can be done in a variety of different ways. The best method we found at first determines the extension (radius of minimum enclosing circle) of every single found contour. Of these, the biggest radius that isn't part of a fraction is then chosen to be the presumed line radius.
To find the ideal line position, we propose a great variety of possible lines. In practice, we choose a line every $0.4 \cdot \text{lineradius}$. For each of these lines, the appropriate contours are assigned. Because we choose a lot of different lines, it is very likely that contours are present in multiple lines. We can find the best lines by removing all lines that have a neighbouring line which contain more symbols. The longest lines correspond to the written lines.
Afterwards, all lines with less than 3 symbols in them are removed. This can be done with confidence, because even the shortest equation (e.g. "1+1") has at least 3 symbols.

While there are some obvious flaws with this method, it works surprisingly and consistently well. It performs especially well when equations are written along lines with large spaces between symbols. Compared to other methods, it can maintain the position of a line and doesn't wander

into other lines.
It does however, have difficulty if there are large amounts of wrong contours close to each other. The algorithm will always attempt to place a line around these contours.
Another issue appears if the lines are written too close together and there is a large contour. If this is the case, then the proposed line position will lie between both lines and read contours from both lines.
This algorithm performs better, if the image contains many small contours, which are randomly distributed across the image.

5.5 Recognition

5.5.1 Preclassification

Author: Timothy Jay Herbst, Duc Anh Phi

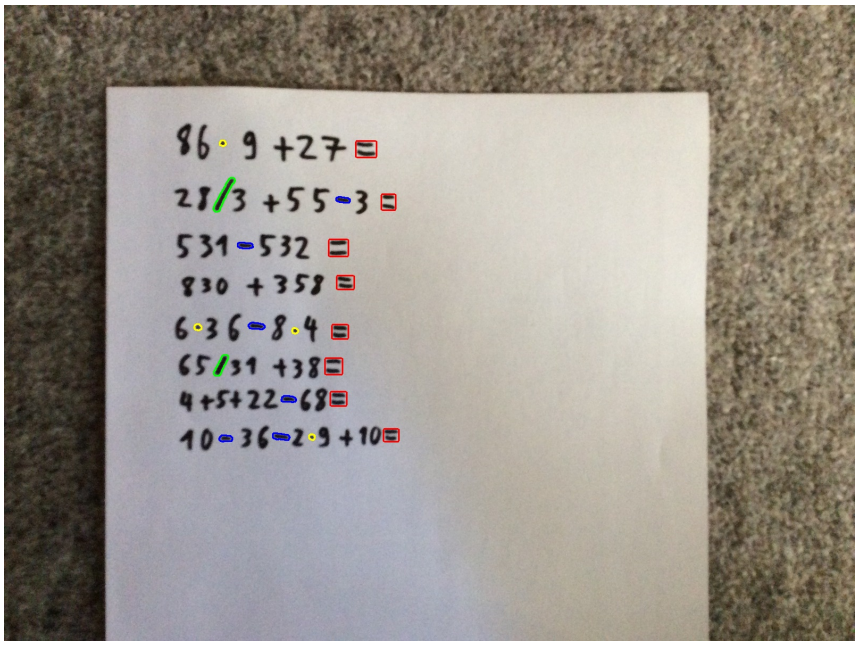


Figure 7: Classification of Math Symbols without a Neural Network. Minuses are marked blue. Multiplications are marked yellow. Division bars are marked green. Decimal points are marked magenta. Equal signs are marked red.

Given the size, shape and positional information of the symbols, along with the order in which they are written, it is possible to classify certain symbols without the need for a neural network. This is very beneficial for two main reasons. The classification problem becomes easier, as there are fewer classes to differentiate from. Additionally, it saves computational time, as fewer contours have to be passed through the network. As a drawback the rule based classification might be less accurate.
We assume that symbols need to have a simple shape while sill being distinct, in order to be accurately classified by certain local rules.
We came up with these rules based on experimentation.

Identify Bars and Points Author: Duc Anh Phi

There are mainly two shapes of interest which are used as a basis for further classification: points and bars. They are significant as many math symbols are based on them, e.g. fraction bars, minus signs, multiply signs, equal signs and commas. After the shape is found, we can distinguish mathematical symbols from their position in relation to other symbols. We defined the following rules:

Bars

- The width of its minimum bounding box is more than twice as big as its corresponding height.
- It does not contain any nested contours (holes)
- The ratio of its contour area to its minimum bounding box area should exceed 0.7. This ratio ensures that the contour is rather "flat". However we add a tolerance for "long" contours, e.g. contours whose ratio of width to height exceeds 4. These long contours do not have to fulfill the "flatness" criterion, to be regarded as bars. This tolerance comes from the observation, that keeping a straight line gets more difficult, as the line increases in length.

Points

- The ratio of its contour area to its minimum bounding circle area exceeds 0.8. This ratio ensures that the contour is rather "round". However, we add a tolerance for "small" contours, e.g. contours whose length or width is smaller than 3 times the threshold line thickness 6. These small contours do not have to fulfill the "roundness" criterion, to be regarded as points. Written points are not always round, especially when writing quickly.

Preclassify bar-shaped Math Symbols Author: Duc Anh Phi

In the following we show how bar-shaped symbols are classified.

Fraction Bars We distinguish between vertical and horizontal fraction bars. Vertical fraction bars are defined as bars with vertical orientation. Analogous, horizontal fraction bars must have a horizontal orientation. Additionally horizontal fraction bars contain contours above and below them, inside a defined acceptance area. This area is inferred from the bar width.

Minus Sign Minus signs are similar to horizontal fraction bars except they do not contain any contours above or below them (inside a defined acceptance area).

Equal Bars While a fraction bar will always have symbols above and below it, equal sign bars always appear in pairs. They either have a single contour (their counter part) above or below them, but never both.

Preclassify point-shaped Math Symbols Author: Timothy Jay Herbst

A difficulty with detecting points or point-like symbols is the abundant use of these. This makes it nearly impossible to account for all possible interpretations of this. Therefore we have focused on the two most common cases: The multiplication dot "$\cdot$") and the comma or dot indicating a non-integer number (e.g. "$3,14$" or "3.14").
There are other important times where a dot appears in an equation, which unfortunately we were unable to cover, including points in letters (e.g. "$i = \sqrt{-1}$"), the time derivative of a variable (e.g. "$\dot{x} = v$") and the dots included in a definition symbol (":=" or "=:").
As the line has already been ordered, we are able to take a look at the preceding and following contour.

Comma If the distance between the contour's center to the lower end of the preceding contour is shorter than the distance between the centers of both, then the contour is classified as a comma.

Multiplication Dot If the distance between the contour's center and the preceding contour's center is shorter than the distance between the contour's center to the lower end of the predecessor, then the contour is classified as a multiplication dot.

5.5.2 Extract Subimages for the Neural Network

Author:, Duc Anh Phi

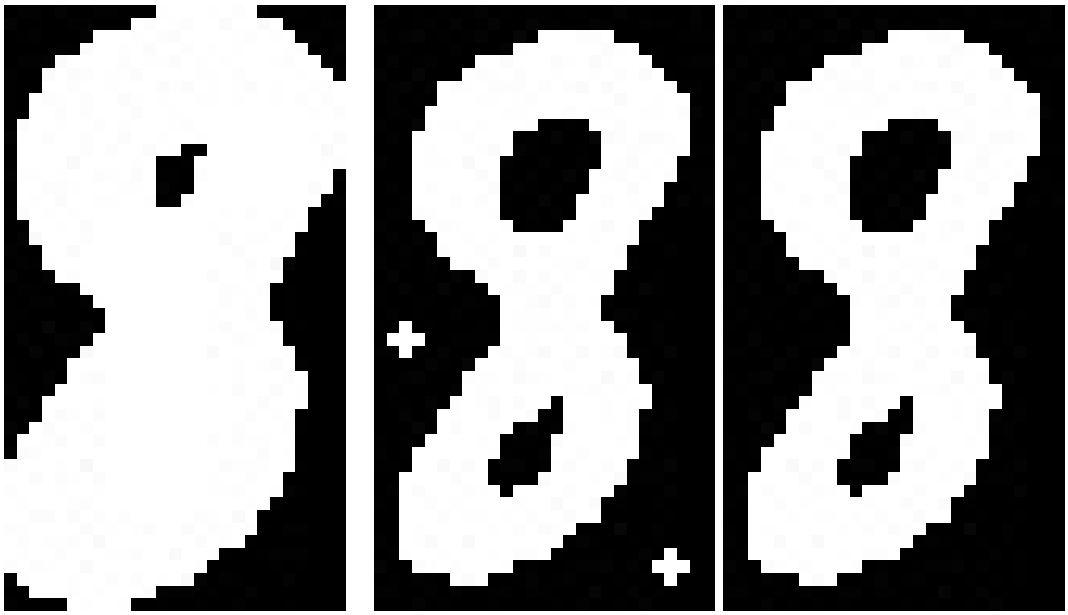
In our preprocessing step, we alter the contours in a way to remove noise. The resulting contours are thicker, due to dilation in the preprocessing. Thus, these contours do not accurately represent what was actually written. Passing these contours to the classifier would yield inaccurate predictions, as the network was not trained on these "thicker" contours. As a solution to this problem, we use the thicker contours as a mask. We apply this mask to the binarized original image, performing a bitwise AND operation. The result is a proper representation of what was written without the noise.

5.5.3 Symbol Classification

Author: Fenja Kollasch

After the preprocessing routine is complete, each handwritten character is extracted from the original image. The correct order in which the characters will appear in the term is also known. Now, the remaining challenge is to recognize the characters and assign the correct labels such that an evaluation of the written term can take place.

Therefore, we need to solve a typical classification problem. It is similar to the classification of the *MNIST* image data set including 60000 images of handwritten digits. Additional to the digits from 0 to 9, our classifier also needs to recognize the symbols *+*, *-*, */ (*, and *)*. Distinguishing



(a) Mask (b) Binary Subimage (c) Resulting Subimage

Figure 8: Extract Subimages

between the mathematical operators is especially a challenge. Brackets, dashes, and minus symbols have a very similar shape.

To classify the image, we use a deep convolutional neural network (CNN). The network will be trained in beforehand. During the use of our main application, the segmented and preprocessed images of the characters are going though the forward pass of the trained model. The labels that were assigned during this process are furthermore send to the symbolic math solver to evaluate the result of the term.

The framework containing most of the machine learning functions we are using, is `PyTorch`. We decided to use `PyTorch` because we familiarized ourselves with this library during the exercises of this class. Furthermore, `PyTorch` offers complete models suitable for classification.

Training data To create a well trained model, a sufficient amount of training data is crucial. For the training routine, we decided to use two public data sets of handwritten symbols. Additionally, we created some training images ourselves that are created the same way as the input of the future classifier in our system. All of these images merge into a combined data set which will be finally used to train the character classificator.

We created our own instance of the `Dataset` class provided by `PyTorch` to capsule the data. It is crucial that the training data resembles the images created by our application as well as possible. Therefore, we apply an individual preprocessing routine to each dataset. Usually, this contains all steps made during the preprocessing of the segmented images. Additionally, the `numpy` arrays

describing the images are transformed into `PIL` images. By doing so, we are able to apply some transformations to the images which are offered by `PyTorch` to create some data augmentation. Furthermore, we resized them to 32*32 pixels. All images are converted to 1-channel grayscale and inverted to have a black background and white content. Finally, the images are transformed to `PyTorch` Tensors to make them suitable for the neural network.

MNIST As a basic training routine, we used the already mentioned MNIST database of handwritten digits. `PyTorch` offers the MNIST data directly. However, the format in which the images are provided by `PyTorch` is not suitable with the preprocessing routine that makes mainly use of the `OpenCV` functions. Therefore, we gather the data not from `TensorFlow`. The images are converted to `NumPy` arrays. Afterwards all necessary operations get applied and the preprocessed images will be saved.

To finally include the preprocessed images in the combined dataset, we provide a function to load the images again, convert them into `PIL` images, apply the necessary transformations and make them into tensors.

HASY Mathematical symbols need to get recognized as well. To include these symbols in our training process we extended our training data with a subset of the HASYv2[14] dataset. This dataset contains 168233 images in total which can be labeled with one of 369 classes including all numbers and latin letters.

Not all symbols in this dataset are relevant for our classifier. Therefore, we filter the images according to their labels and thus simply include the ones that are necessary for our training procedure. Our first approach was to take only the digits and plus symbols from this dataset. Unfortunately, many of the digit images were cropped or of poor quality. In the end, we decided to take only the plus symbols from the dataset what leaves us with roughly 100 images after the filtering. Since MNIST contains 60000 images of digits, we have a critical overflow of digit characters in comparison to the plus symbols. To reduce this gap, we make use of data augmentation and apply a number of transformation to all non-digit characters. By rotating, flipping and combinations of rotation and flipping we generate four extra images per non-digit symbol and remain now with 405 training images.

(Author: Timothy Jay Herbst, Fenja Kollasch)

Custom training images In addition, we generated some training images ourselves by writing down symbols and capturing them the same way in which our application will capture the mathematical formulae during use. All training equations were written on paper. A variety of different writing utensils of different colors and thicknesses were used. Images were then separated using the methods that may be found on the branch `gathertrainingdata2`.

Training A *Jupyter notebook* is provided for the complete training routine. First of all, the combined dataset is loaded for test and training data. This dataset includes images from MNIST, HASY, and our own images as well. Since the class that holds our dataset inherits from `PyTorch`'s class `Dataset`, we are allowed to use a `DataLoader` for training. The `DataLoader` class makes it possible to iterate in a comfortable way over the minibatched dataset. We use minibatches of the size of 16 samples per batch. The training is running over 10 epochs.

Stochastic optimization is done by using an *ADAM*[15] optimizer. Since it is claimed to work best on classification issues with multiple classes, we use cross entropy loss as a loss function.

We trained multiple different models to be able to choose those who work best in practice. All models use data samples from the same combination of data sets but differ in the number of symbols that were included. The models that performed the best in a use case are those trained on digits and plus symbols only. By handling the recognition of $-, /, (,)$ manually, the confusion of the network is reduced and the results are better.

The models were tested on a separate data set including samples from MNIST, HASY, and our own images that were not used during the training of any of the classifiers. Most of our models achieved a quite good accuracy on these testing data (over 90%). Table 1 shows the accuracies of our three best performing models.

Model	**Accuracy**
Trained on 0-9, +	
Trained on 0-9, +, -, /, (,)	
Trained on 0-9, +, (,)	

Table 1: The accuracies on the test set for three of our models

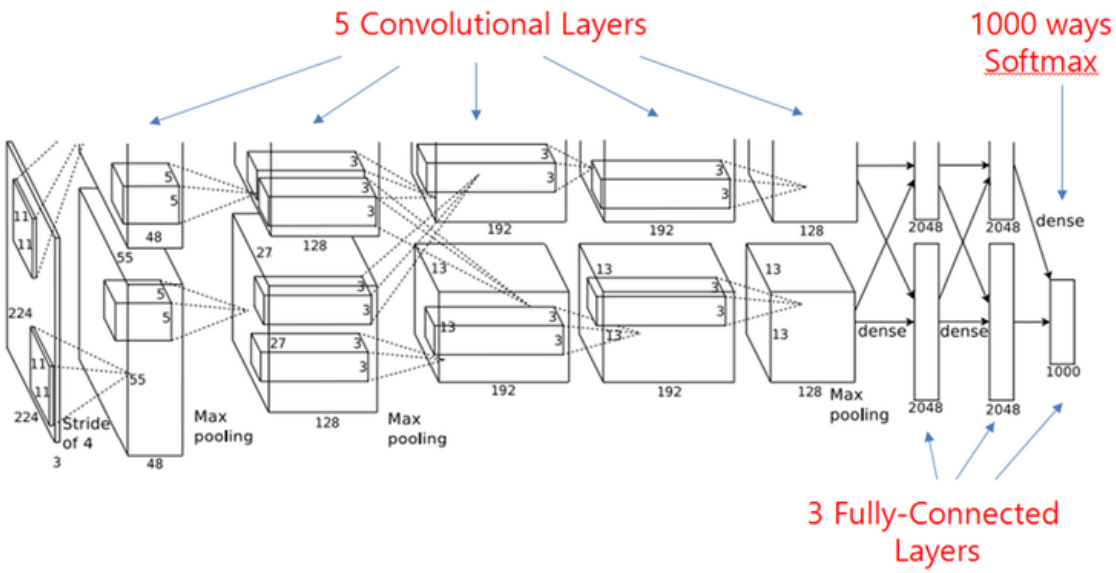


Figure 9: The AlexNet architecture

Neural network The network architecture we decided on is *AlexNet*[16]. Thus, the network contains five convolutional and three fully connected layers. The convolutional layers are followed by max-pooling layers while some dropout layers are placed before the linear layers (see

figures 9). An implementation of *AlexNet* is provided by `PyTorch`. Since the model is suited for images with three color channels we modified the first layer to accept images with only one channel. On the next page, a summary of the network is displayed. It shows, which layers the network contains and how these layers look like.

```
AlexNet(
  (features): Sequential(
    (0): Conv2d(1, 64, kernel_size=(7, 7), stride=(2, 2),
     padding=(3, 3), bias=False)
    (1): ReLU(inplace)
    (2): MaxPool2d(kernel_size=3, stride=2, padding=0,
     dilation=1, ceil_mode=False)
    (3): Conv2d(64, 192, kernel_size=(5, 5), stride=(1, 1),
     padding=(2, 2))
    (4): ReLU(inplace)
    (5): MaxPool2d(kernel_size=3, stride=2, padding=0,
     dilation=1, ceil_mode=False)
    (6): Conv2d(192, 384, kernel_size=(3, 3), stride=(1, 1),
     padding=(1, 1))
    (7): ReLU(inplace)
    (8): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1),
     padding=(1, 1))
    (9): ReLU(inplace)
    (10): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
     padding=(1, 1))
    (11): ReLU(inplace)
    (12): MaxPool2d(kernel_size=3, stride=2, padding=0,
     dilation=1, ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(6, 6))
  (classifier): Sequential(
    (0): Dropout(p=0.5)
    (1): Linear(in_features=9216, out_features=4096, bias=True)
    (2): ReLU(inplace)
    (3): Dropout(p=0.5)
    (4): Linear(in_features=4096, out_features=4096, bias=True)
    (5): ReLU(inplace)
    (6): Linear(in_features=4096, out_features=15, bias=True)
  )
)
```

Hyperparameter training The library `PyTorch` allows to set the training hyper parameters. The ones that are relevant in our routine are the learning rate, the beta values for the *ADAM* optimizer, and the weight decay. To find the optimal values for these parameters, we created a second notebook in which we trained under the same conditions for one epoch with different hyperparameter values and compared the achieved accuracies. Table 2 shows which values we tried out for each parameter. The best results were achieved with a learning rate of 0.001, a beta tuple of (0.95, 0.95), and a weight decay of 0.

Parameter	Values
Learning rate	0.01, 0.001, 0.0001
Beta 1	0.8, 0.85, 0.9, 0.95
Beta 2	0.9, 0.925, 0.95, 0.99
Weight decay	0, 0.01, 0.001, 0.0001

Table 2: The values we evaluated for the different hyperparameters

Automatic Classification After the training is complete, we save the model to a *.ckpt* file. The actual classifier that will be used in our program will be an instance of the class `MathSymbolClassifier`. This classifier allows to use multiple models for classification. As an input, the constructor receives a dictionary containing paths to all models that should be considered. To classify a collection of images, the `MathSymbolClassifier` provides a function `classify`. The function works by executing the following steps:

1. For each model, pass the input data through the forward pass
2. We receive prediction matrices for each model
3. Save the highest probability for each sample by calculating the rowwise maximum of the probability matrix.
4. Concatenate the resulting column with the best probabilities from the previously regarded prediction matrices.
5. Save the index of the highest probability from each prediction matrix since they describe the labels.
6. We receive a matrix where each columns describe the best predictions for the input data. Calculate the rowwise maximum to receive the globally best prediction.
7. Take the labels that belong to the globally best predictions
8. Return either the label indices as they are or map them to the symbols they describe and return the symbol vector.

5.6 Calculations

Author: Fenja Kollasch

After all the previous steps, the application should have been able to extract all written symbols by finding their contours and to remember the correct order and line placement in which they appeared originally. Furthermore, each symbol should have been recognized either from the network or in a manual way. With this information, the original equations can be reconstructed to Python strings containing the same semantic information.

Now the last step is solving these equations. Fortunately, the `SymPy` framework is able to solve an equation that is given as a string. Thus, we give the reconstructed equations as an input to `SymPy`. `SymPy` parses the expression and calculates the result of the expression.

6 Experiments

6.1 Testing the Preprocessing

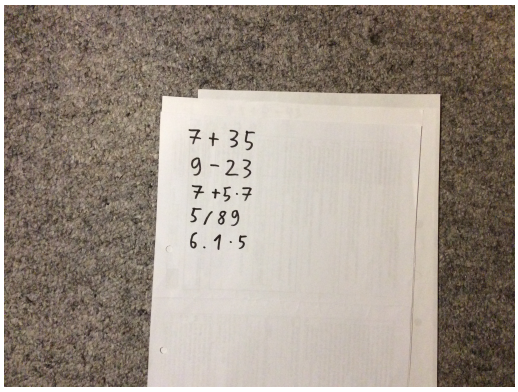
Author: Timothy Jay Herbst

We tested our Preprocessing on a variety of different images (see fig: 10). For this we varied the writing medium (paper or whiteboard) and the pen used. (We varied colour and thickness.)
We found that the Preprocessing worked well under even lighting conditions. It is even able to filter out most background objects. However, it does have difficulties if there is a very strong brightness gradient on the image. Some of the whiteboard images have "hollowed" out text (see fig: 11). We believe this is due to the thickness of the writing. The adaptive threshold filter sees the borders of the equation separately. This could be countered by scaling the image down. We did not implement this in the final version, because it would make thin writing disappear. Additionally this should not cause problems, since the contours are rescaled before classification. It does, however, make the preclassification more difficult, since we can no longer use the amount of nested contours as a reliable reference.

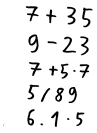
6.2 Comparing Preclassification and Neural Networks

Author: Timothy Jay Herbst, Duc Anh Phi

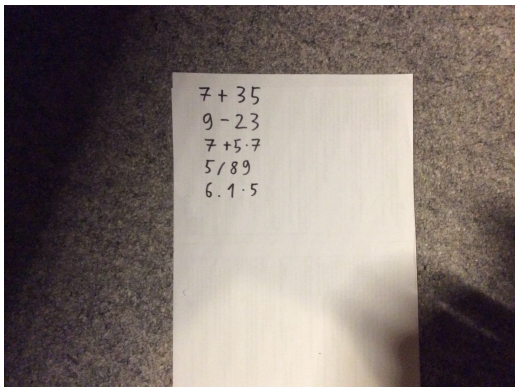
In the following experiment, we test the Image-Input-Calculator and also give some motivation as to why it uses several preclassification functions. For this, we will compare how well the calculator performs using two different types of classifiers:
The first classifier is capable of classifying a contour as one of the symbols that might be in the equation we are analysing. These are the digits 0 through 9 , + , - , / , (and). The only time at which that we use preclassification processes is for deciding whether a dot is a multiplication symbol "·", a comma "," or decimal point "." or if the dot is not part of the equation at all.
The second classifier uses as many preclassifying methods as we were able to develop. This means it is able to classify a contour to any digit 0 through 9 and +. It is not possible for this classifier to correctly identify "(" and ")".



(a) Example of good Preprocessing

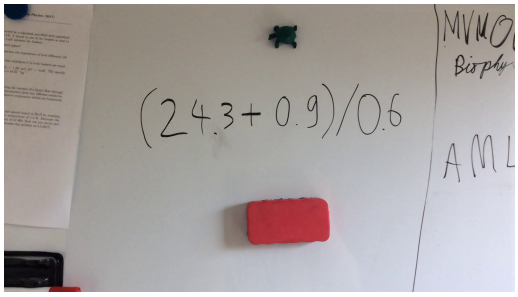


(b) After Preprocessing

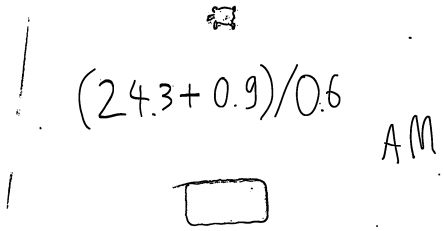


(c) Example of bad Preprocessing

(d) After Preprocessing



(e) Example of good Preprocessing on a whiteboard with many background objects



(f) After Preprocessing

Figure 10: Images before and after Preprocessing. On the left one can see the original image and on the right the image after Preprocessing. Noteworthy is, that even though the image in the first and second row are nearly identical, the image in the second row still has a lot of noise after Preprocessing. This is due to the high brightness gradient and because of the darkness of certain parts of the image.

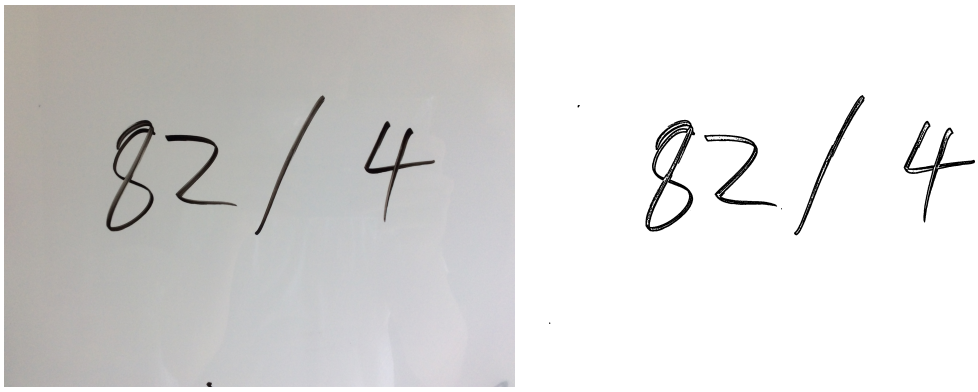


Figure 11: Whiteboard Image before and after Preprocessing. Sometimes the text of a whiteboard image becomes hollow. This is likely because the adaptive threshold filter does not have a large enough kernel size to successfully separate the outside borders of the equation if the text becomes particularly thick.

To compare how well these two approaches classify, we have created several images of equations with varying amounts of digits. Both classifiers were tested on equations of the same style and we tracked, how often contours were correctly or incorrectly classified. These results can be seen in table 6.2.

Classifier	Numbers & "+"	No Preclassification
Correctly Classified Symbols	152	121
Wrongly Classified Symbols	25	34
Wrongly Preclassified Symbols	2	-
Correct Equations	10	2
Wrong Equations	30	20
Symbol Accuracy	85.9%	78.1%
Equation Accuracy	33%	10%

6.3 Confusion Matrix

Author: Fenja Kollasch

The qualitative performance of the automatic classifier can be evaluated by regarding the confusion matrix of the models. During the training process we produced multiple models. We took four approaches that produced the most reliable classifiers and run them with labeled test data that was not used during the training of any of the used models. The test data contains a combination of MNIST, HASY, and our own images, as well as the training data of all models does. It contains 12817 data samples in total. The result of this experiment were the following confusion matrices. The x axis displays the prediction of our classifier while the y axis shows the actual label.

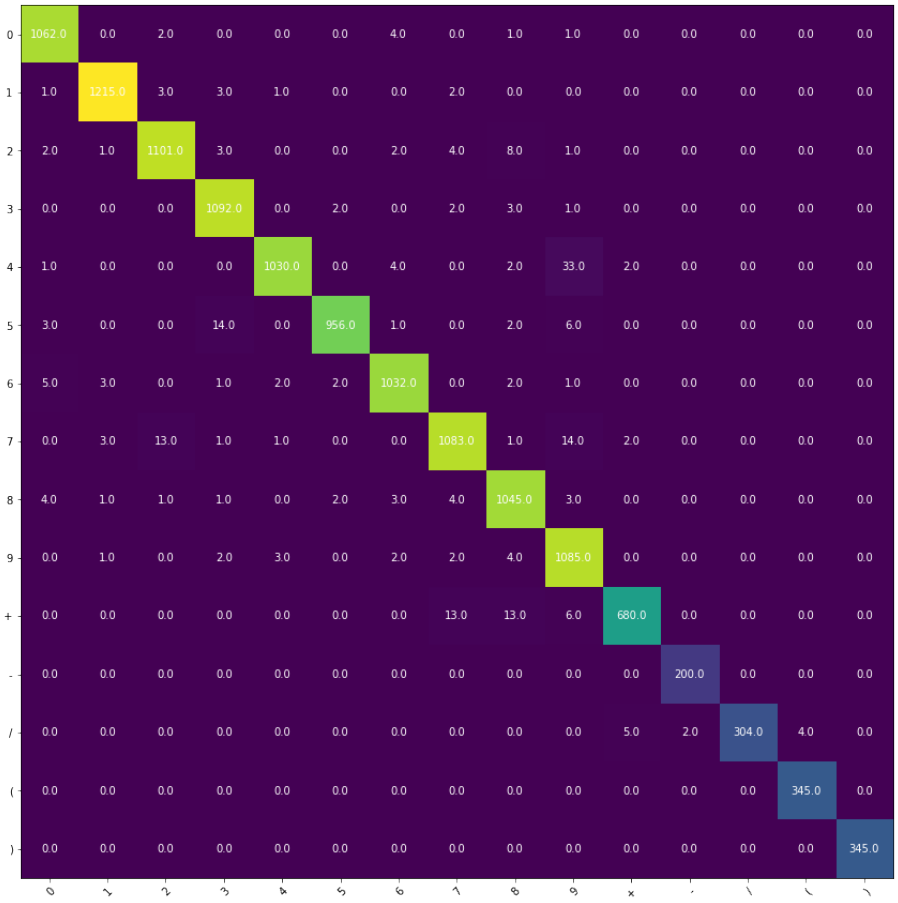


Figure 12: Confusion matrix for a classifier that was trained with the digits from 0-9, +, -, /, (,)

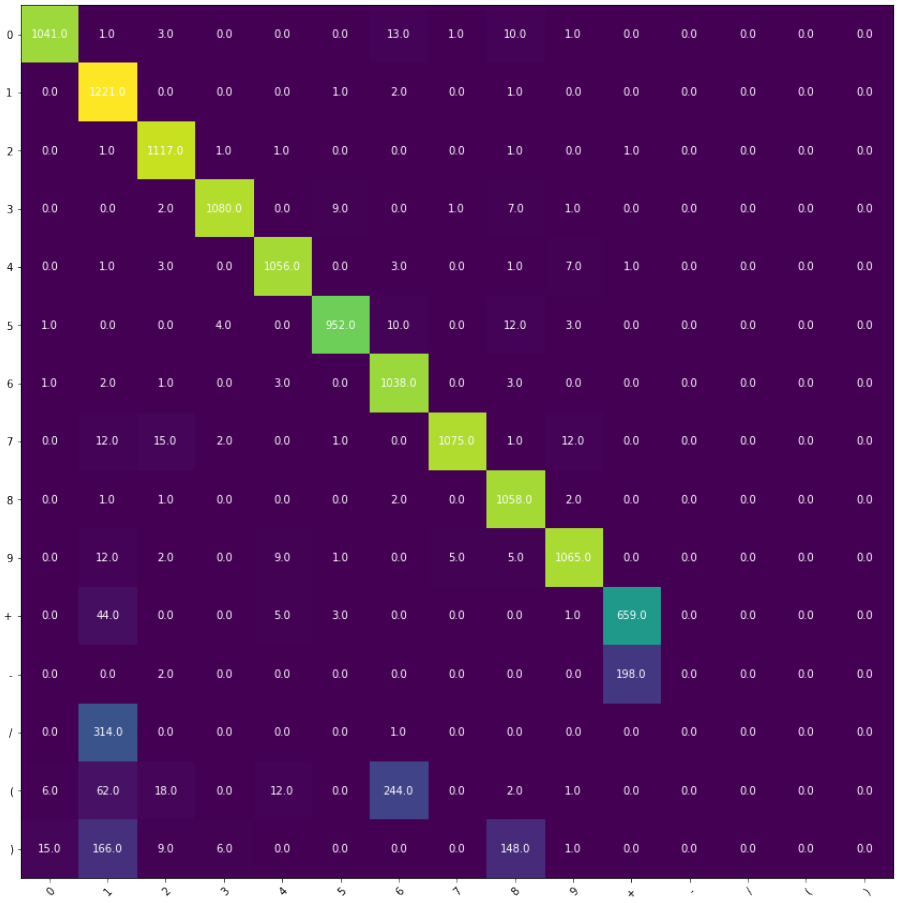


Figure 13: Confusion matrix for a classifier that was trained with the digits from 0-9, +

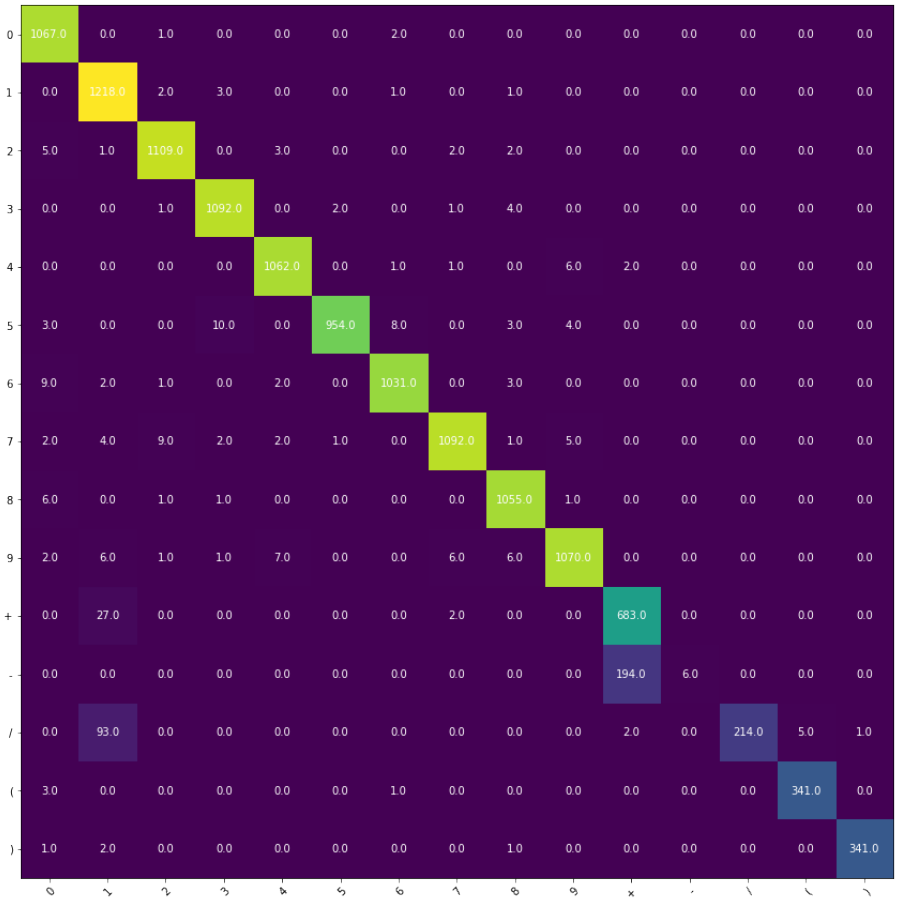


Figure 14: Confusion matrix for a classifier that ensembles three different models

Figure 12 shows the confusion matrix of a classifier using a model that was trained with a data split including images of all 15 labels that our classifier can resolve. On the first sight, it presents an acceptable result. All digits are recognized correctly almost all the time. With the operators however, the classifier makes more wrong predictions than with the numbers. Still we can see that it predicts the symbols most of the time correctly. Unfortunately, in case of application, the predictions are more unreliable than one can assume from this matrix. This is probably the case because the number of MNIST images in training and test data highly outweighs the number of images that were generated from our application.

Figure 13 displays the matrix of a classifier where the models was trained only on digits and the plus symbol. We can see that this classifier performs well on those data samples, but is not able to recognize $-, /, ($ or $)$. Therefore, we receive a lot of false positives for various digits. An interesting fact is however that this classifier performs better in a real use case. The application is able to preclassify the mentioned operators. By not leading them to the classificator, the network

does not get confused with the unknown symbols.

Finally, figure 14 shows the results for a classifier that ensembles three different models: The models that were used in the previously mentioned classifier and an additional third model that was trained on the digits from $0-9, +, (,$ and $)$. As the other classifiers, this one performs quite well on the digits. The operators however cause a high confusion with this classifier. Especially the minus symbol is basically not recognizable but is often mistaken as a plus symbol. The division symbol causes a high confusion with the digit 1.

7 Discussion

7.1 Pre-Processing

Author: Timothy Jay Herbst

Transforming the original image into a usable binary image worked relatively smoothly. The main issues which arise during this step are caused by the large variety of images. These vary in brightness, background color and writing medium, to name a few. However, it was possible to develop a method which consistently gives good results.
The simple pre-processing step performs well on a variety of images. It does, however, experience problems on very large and small images, because it operates with a fixed kernel size. A solution for this would be to have the kernel size adapt to the size of the image or to scale the image. However, when scaling the image we found that it could happen that the equation would dissapear if written less clearly.
The border removal process performs well when removing contours caused by objects that are not the equation. It does fail sometimes, however, if the colour of the background object does not differ strongly from the overall background colour. Then the border of the object may not be a consistent line. It can then not be fully removed and a part of the border may remain (see fig 15). Also if an object is fully within the image, it is not likely to disappear in this step. It will,

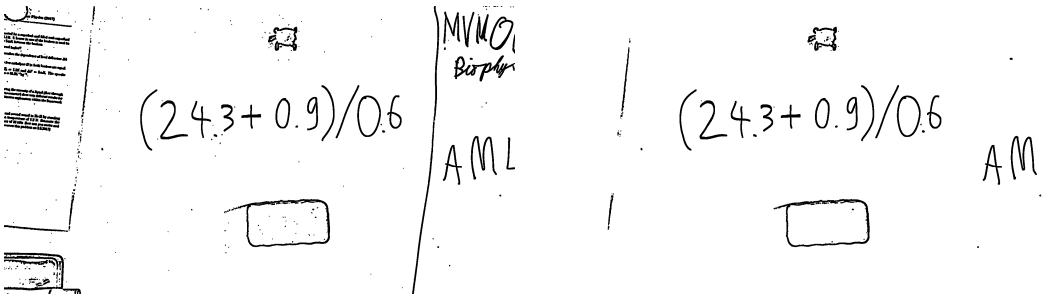


Figure 15: Many of the background contours are removed. However, the paper on the left side of the image is only partially removed and the eraser is too far away from the border to be removed.

however, likely disappear during the line ordering step because it is unlikely to be in the same line as another contour and will be removed because lines that consist of only one line can't be equations.

The brightness removal process allows for the removal of reflected light sources. It also helps remove small sources of noise from the background of an image, if the background is brighter than the image is on average. If the image were on average darker than the equation, however, it could cause the writing to disappear. However, we were never able to observe this on a whiteboard or paper. This step makes it impossible to detect an equation on e.g. a blackboard, because here, the writing is brighter than the background.

7.2 Classification problems

Author: Fenja Kollasch

A major issue appearing during development was the fact that the usability of our application highly depends on the accuracy of the classifier. Our initial ambition was to recognize various more symbols automatically. Unfortunately, we were not able to gather enough training data and therefore we could not create models with the desired accuracy. Any classifier that recognizes less than 95% causes major leaks in the overall performance of our program and makes it unusable. Furthermore, it took a long time to create well-trained models what made the training procedure tedious and decelerated our progress.

7.3 Segmentation

Author: Duc Anh Phi

We segment written characters from the image using visual computing methods. Unfortunately this segmentation process can still be fooled by noise. After the preprocessing we remove noise based on a computed line thickness threshold. For most cases this works well, however it can happen that the threshold is not representative of what was actually written. As seen in this example(!Example with the holes in the paper). This would cause actual written symbols to be regarded as noise and consequently being removed. The reverse case that actual noise is regarded as written symbols is another problem we have not solved yet. Alternatively you could train a network differentiating between noise and written symbols. We have not explored semantic segmentation using deep learning. We assumed it would require too much resources not only in computational power (for training the models) but also in finding sufficient data to train on, as usually these algorithms require pixel-wise labeling of images.

7.4 Comparing Preclassification and Neural Networks

Author: Timothy Jay Herbst

Comparing the preclassification with the Neural Network shows that there must be some difference between the data in our equation and the online datasets that were used. The classifier performed significantly better (around 98% accuracy) on the training data than our data (78.1% and 85.9%). This could be improved upon by creating more training data for the classifier.
The presence of the preclassification drastically improved the quality of the Calculator. Because an equation can only be solved correctly if all symbols are classified, an improvement on the classifier causes an exponential improvement on the Calculator. In this case, the presence of a preclassifier increases the accuracy of the Calculator by a factor of three.
Although the preclassifier is occasionally wrong, it is more reliable than the classifier. It's main advantage, however, lies in reducing the amount of labels the classifier has to learn.

8 Conclusion and Outlook

Author: Timothy Jay Herbst, Fenja Kollasch, Duc Anh Phi

The Image-Input-Calculator is very reliable when it comes to detecting the pixels belonging to the equation. It can consistently retrieve symbols from an image.
It does unfortunately have difficulties reliably classifying symbols. We presume this is due to some difference between our written symbols and the training libraries used. It was possible to improve upon the results by using manual classification criteria for some of the symbols. This produced correct equations far more consistently. Unfortunately with the best trained models, the Calculator cannot detect brackets at the moment.
Future improvements would include increasing the amount of training data, which would make the classifier work reliably with more labels.

9 Installation and Usage

Author: Duc Anh Phi)

9.1 Prerequisites

For the classification, it is important to have a trained model in the same directory as the source code.

A model zoo is available under this link:
`https://drive.google.com/drive/folders/1VQiFUtBg_L8-vnu61NdqksSR-bOV08wq?usp=sharing`

It is possible to train own models by executing the JuPyter notebook *Combined Data Training.ipynb*. To execute this notebook, the HASY data set must be downloaded, which is available under the link above.

9.2 Installation

1. Install Python 3.6 (if not already installed)
2. Download the project from `https://github.com/DucAnhPhi/image-input-calc`
3. (Optional) Set up a Python virtual environment for this project. It keeps the dependencies/libraries required by different projects in separate places (isolates them to avoid conflicts).

   Run the following commands in your terminal:

   ```
   $ pip install virtualenv
   $ cd path/to/image-input-calc
   $ virtualenv -p python3.6 virtual_env
   ```

   To begin using the virtual environment, it needs to be activated.
   If you are on Linux or MacOS, run the following:

   ```
   $ source virtual_env/bin/activate
   ```

   If you are on Windows, run the following:

   ```
   $ virtual_env/Scripts/activate.bat
   ```

4. Install all project dependencies with:

   ```
   $ pip install -r requirements.txt
   ```

9.3 Usage

In the project directory, run the program with:

```
$ python3 app.py
```

Sources

[1] Will Forfang. *Using Artificial Intelligence to evaluate Handwritten Mathematical Expressions*. Apr. 9, 2016. URL: http://www.willforfang.com/computer-vision/2016/4/9/artificial-intelligence-for-handwritten-mathematical-expression-evaluation (visited on 09/29/2019).

[2] Y. Lecun et al. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11 (Nov. 1998), pp. 2278–2324. DOI: 10.1109/5.726791.

[3] Christopher J. C. Burges Yann LeCun Corinna Cortes. *The MNIST database of handwritten digits*. 1998. URL: http://yann.lecun.com/exdb/mnist/ (visited on 09/21/2019).

[4] Gregory Cohen et al. "EMNIST: an extension of MNIST to handwritten letters". In: *CoRR* abs/1702.05373 (2017). arXiv: 1702.05373. URL: http://arxiv.org/abs/1702.05373.

[5] Yassine Chajri and Bouikhalene Belaid. "Handwritten mathematical symbols". In: *Data in Brief* 7 (2016). URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4786757/.

[6] Kaggle Inc. *Handwritten math symbols dataset*. URL: https://www.kaggle.com/xainano/handwrittenmathsymbols (visited on 09/28/2019).

[7] Irwansyah Ramadhan, Bedy Purnama, and Said Al Faraby. "Convolutional neural networks applied to handwritten mathematical symbols classification". In: May 2016, pp. 1–4. DOI: 10.1109/ICoICT.2016.7571941.

[8] S. Lee et al. "Handwritten Music Symbol Classification Using Deep Convolutional Neural Networks". In: *2016 International Conference on Information Science and Security (ICISS)*. Dec. 2016, pp. 1–5. DOI: 10.1109/ICISSEC.2016.7885856.

[9] Oleg Golubitsky and Stephen M. Watt. "Distance-based classification of handwritten symbols". In: *International Journal on Document Analysis and Recognition (IJDAR)* 13.2 (June 2010), pp. 133–146. ISSN: 1433-2825. DOI: 10.1007/s10032-009-0107-7. URL: https://doi.org/10.1007/s10032-009-0107-7.

[10] tutorialspoint. *OpenCV - Adaptive Treshold*. URL: https://www.tutorialspoint.com/opencv/opencv_adaptive_threshold.htm (visited on 09/21/2019).

[11] OpenCV API. *Miscellaneous Image Transformations*. URL: https://docs.opencv.org/2.4/modules/imgproc/doc/miscellaneous_transformations.html?highlight=adaptivethreshold (visited on 09/21/2019).

[12] Keiichi Abe Satoshi Suzuki. "Topological Structural Analysis of Digitized Binary Images by Border Following". In: *Computer Vision, Graphics, and Image Processing* 30 (1985). URL: https://s3.amazonaws.com/academia.edu.documents/38698235/suzuki1985.pdf?response-content-disposition=inline%3B%20filename%3DTopological_Structural_Analysis_of_Digit.pdf&X-Amz-Algorithm=AWS4-HMAC-SHA256&X-Amz-Credential=AKIAIWOWYYGZ2Y53UL3A%2F20190925%2Fus-east-1%2Fs3%2Faws4_request&X-Amz-Date=20190925T141313Z&X-Amz-Expires=3600&X-Amz-SignedHeaders=host&X-Amz-Signature=2cedcb39adec887e8f4d514acb

[13] Alwyn Mathew. URL: https://cs.stackexchange.com/questions/59475/width-of-a-string-line-in-an-image (visited on 09/29/2019).

[14] Martin Thoma. "The HASYv2 dataset". In: *CoRR* abs/1701.08380 (2017). arXiv: 1701.08380. URL: http://arxiv.org/abs/1701.08380.

[15] Jimmy Ba Diederik p. Kingma. "Adam: a method for Stochastic Optimization". In: *CoRR* abs/1412.6980 (2017). arXiv: 1412.6980. URL: https://arxiv.org/abs/1412.6980.

[16] Alex Krizhevsky. "One weird trick for parallelizing convolutional neural networks". In: *CoRR* abs/1404.5997 (2014). arXiv: 1404.5997. URL: http://arxiv.org/abs/1404.5997.