AML Final Project: Creating an Image Input Calculator

Timothy Jay Herbst, Fenja Kollasch, Duc Anh Phi

September 28, 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`. While similar projects exist, this project goes into detail, how this can be done. And, being open source, demonstrates, how such a problem may be tackled and gives a basis for many other projects on a variety of topics.

Contents

1 Motivation

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. In this paper, we explain how such a calculator can be designed.
We hope that this serves as an inspiration to solve similar issues, and that the code provided proves useful.

2 Introduction

The creation of an automatic image input calculator for handwritten formulae brings together two common computer-scientific problems.

For once, several operations from visual computing need to be applied to the image. The application has to segment the input image into one segment per character. It is important that the ordering of these symbols is preserved during this step. Otherwise, it would not be possible to reconstruct the original formula.

The second issue our program has to address, is the recognition of each segmented character. To do so, we can apply methods from machine learning. Luckily, the recognition of handwritten symbols is a well-known problem that has been approached by various researchers before. LeCun et al [**lecun1998**] 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 [**mnist**] 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 two major steps are complete, there is only one task left. Reconstructing the formula with the recognized symbols, and aligning them in the order in which they appeared on the input image to finally evaluate the term and present the result.

3 Read and Interpret Visual Information

3.1 Data Collection

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.

3.2 Preprocessing

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) and white (255).
- Connectedness of the symbols: For later steps, it is useful to not segment any symbols. In the same manner, we do not wish for two different symbols to be connected.
- 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.)

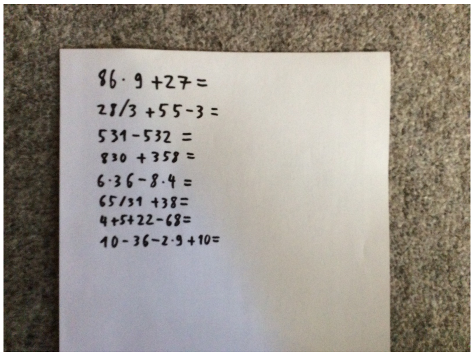
- Background contour removal: Oftentimes when taking an image of a contour there will be other objects in the image (e.g. pencils, the border of the paper, etc.).
- Correction for reflected bright spots: Especially, when using images from white boards, the program has to contend with reflected light sources.

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.

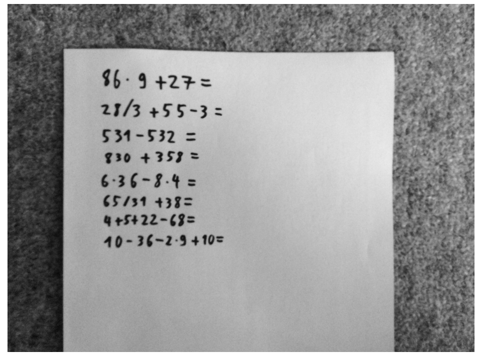
3.2.1 Simple Preprocessing

The image our program receives as input is likely not easily analyzed as is. We therefore pre-process 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 well for our purpose.
It operates in five steps (see fig: 1):

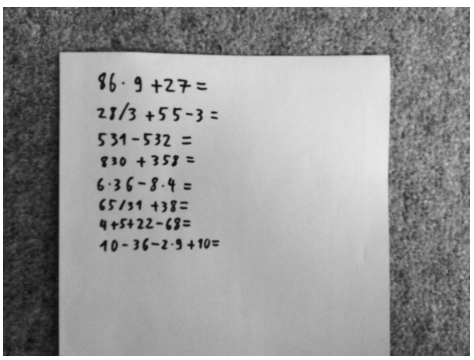
- Grayscaling: At first, we grayscale our image. Here, we reduce the BGR (blue, green, red) image into one with a single dimension per pixel. This is useful, because it simplifies the image and allows us to more easily analyse it.
- Gaussian Blur: Here, we apply a Gaussian blur kernel on the image to 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. We will later explain in more detail how this filter works.
- Dilation: After the previous steps, it is possible that some of the symbols have decomposed into several parts. This allows for easier contour



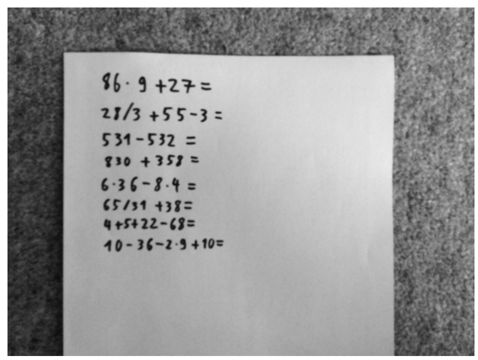
(a) Original Frame



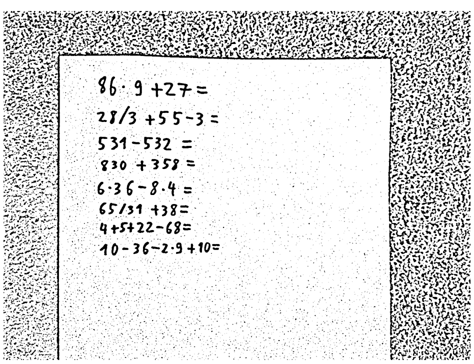
(b) Grayscaled



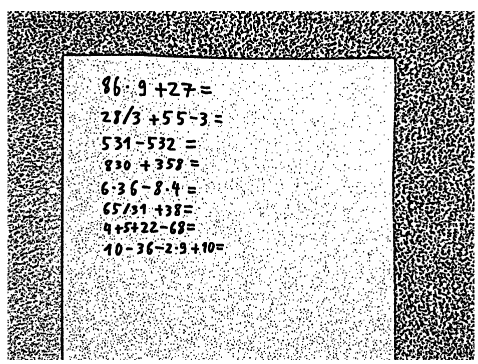
(c) Gaussian Blur



(d) Morphological Opening



(e) Adaptive Gaussian Thresholding



(f) Dilation

Figure 1: Various steps performed during the simple Pre-Processing Process

detection. However, this makes the contour more difficult for the classifier to recognise, which is why this step will be undone later. Therefore, it is useful at this point to thicken the contours.

3.2.2 Border Removal

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 be flaky and can be roughly traced to the border of an image, see fig .Using this knowledge we can create a mask, which removes the background contours.
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.

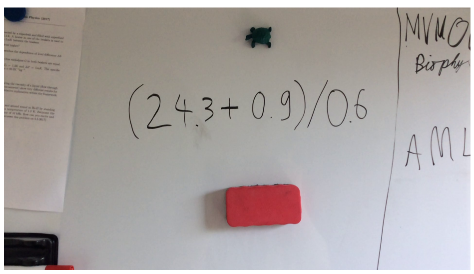
- Adding a Border: For the next step we add a black border to the image. Our aim is to remove all contours which connect with this border.

- 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.

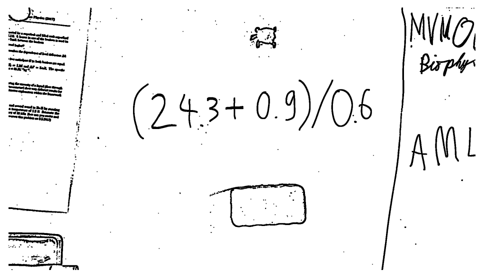
- Floodfill Border: We then floodfill the border. This removes any contour which connects to the border.

- Remove Border: Since we changed the image size by adding a border, we now have to remove it.

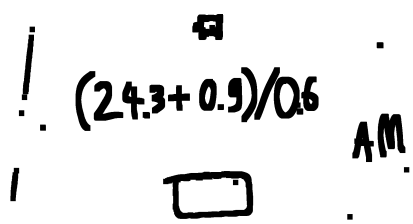
In this manner we determine the mask. Our resulting image is determined, by having all pixels white, except those, that are black both in the mask and in the simple preprocessing. This removes background contours, without making our image unreadable.



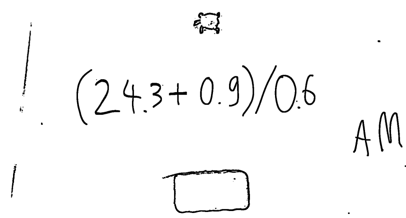
(a) Original Frame



(b) After Simple Preprocessing



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

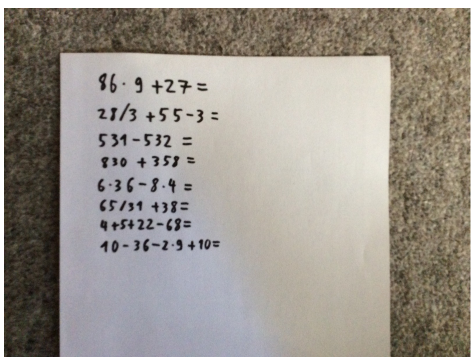


(d) Resulting Image

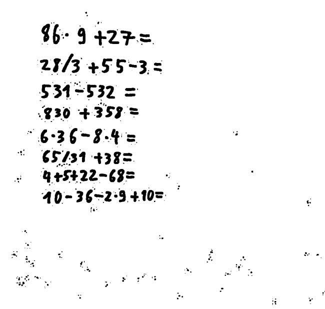
Figure 2: Performance of the Border Removal Mask.

3.2.3 Reflection Removal

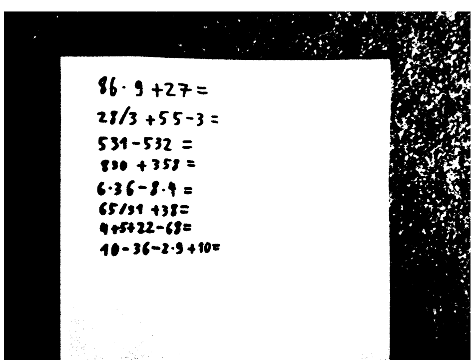
Bright light sources may reflect off the surface on which the equation is written. This occurs especially often when using a white board, but also on paper (see fig. 3). Unfortunately, the adaptive threshold function cannot differentiate between the writing and the bright reflections. Fortunately, the reflected light sources all have a commontrait: 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.



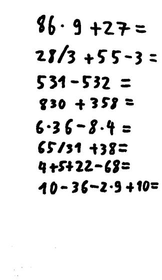
(a) Original Frame



(b) After Simple Preprocessing and Background Removal



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



(d) Resulting Image

Figure 3: Performance of the Reflection Removal Mask.

3.2.4 Adaptive Threshold

A problem we face with the images, is that we do not know if a pixel is part of the writing or not. The "Adaptive Threshold" function [**adaThresh**], found in the `OpenCV` library is very useful in this case. 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 [**cvAdaThresh**] 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

3.3 Segmentation

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.

3.3.1 Contour Detection

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

The underlying algorithm we used for contour detection is from the paper Suzuki et. al. (1985). 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.

3.3.2 Removing Small Contours

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 boundary 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 boundary line thickness.

Compute Line Thickness The algorithm for computing the line thickness of a contour was taken from `https://cs.stackexchange.com/questions/59475/width-of-a-string-line-in-an-image`

1. We start with a binary image (see fig. 4), so the pixel values are going to be 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.

3. After skeletonizing the image, the white line becomes one pixel thick (see fig. 4). The resulting skeleton is a thin approximation of the original image.
   Summing up the skeleton is going to give us the approximate length of the line.

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

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

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

(a) Original Contour (b) After Skeletonizing

Figure 4: Computing the Line Thickness

3.3.3 Group Contours

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

Handle 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 most outer contour stored in its "contour" property. The contours representing holes are stored in the "holes" property.

Fractions Fractions are problematic as they have an inherit order to them – especially nested fractions. These orders have to 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 nominator and 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 fractions first. We find fraction bars, which are contours with the following criteria:

- their width is more than twice as big as their height
- they do not contain any holes
- their ratio of contour area to minimum bounding box area exceeds 0.7
- their ratio of contour area to their convex hull area exceeds 0.7
- they have a horizontal orientation
- they contain contours above and below them inside a defined acceptance area

We chose these criteria based on experimental observations.

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 "nominator" or "denominator" property.

- sort each contour in the “nominator” or “denominator” property by their x-coordinates – this way we ensure a correct order
- create a bounding box around the whole fraction
- create a contour object with the bounding box and reference to the Fraction object

Equal Sign A written ”=” sign consists of two vertically stacked horizontal bars. We group both contours in a single contour object. The bars of an equal sign will be detected in a similar manner as the fraction bars. They can, however, be distinguished by the contours around them. While a fraction bar will always have symbols above and below it, equal sign bars always appear in pairs and either have a single contour above it or below it, but never both. 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.

3.4 Line Assigning and Ordering

At this stage we have a list of (grouped) contour objects. We now face the problem of bringing them in a 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 jump a lot between lines. With this observation we developed the following algorithm to separate lines:

1. Initially contours are sorted by their y-coordinate. 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 boundary value to determine whether a new line starts.

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 both. If the calculated value is within the y-deviation boundary, both belong to the same line. However, if the calculated value is greater than our 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 Assigning we have developed several algorithms, of which we will be discussing the most interesting, the ones able to tackle a particular problem and the most efficient ones.

3.4.1 Determining the Direction of Writing

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:
Firstly, unlike a computer a human will not precisely write their symbols in a line. Oftentimes a symbol will be significantly above or below said line. It 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 is very useful to be able to determine the direction of such an "invisible line". In the following we will be discussing a method how to determine this line. It should be noted at this point however, that in most images we took this "invisible line" deviates by less than 10°. All of the following algorithms are capable of dealing with such small deviations.

3.4.2 Assigning Lines to Proposed Line Positions

Our aim is to accurately order the contours into lines. A possible method for doing this is by proposing the position of lines. Then we check for each proposed line, whether or not a given contour is a good fit (if it is within a certain radius of the line).
For this procedure we at first have to find a rough estimate of the diameter 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 minimal 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 every one 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 lines, 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 smallest 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 hazy patches in an image, which cause a large amount of wrong contours close to each other. The algorithm will always attempt to place a line around these contours.
Another issue is caused, if the lines are written too close and there is a large contour. If this is the case, then the line might be placed in between both lines and use contours from both lines.

3.5 Recognition

3.5.1 Preclassification

Given the size, shape and positional information along with the order in which the symbols 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 less classes to differentiate from. Additionally it saves computational time, as less contours have to be passed through the network. We have already performed some classification before, in the segmentation step. Contours were classified as fraction bars, based on local rules. The same goes for equal signs. In the following we will be discussing some of the possible preclassifications that performed well.

Dots, Commas and Multiplication Symbols A difficulty with detecting handwritten equations is the common usage of points or point-like symbols. 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 partial number (e.g. "$3,14$" or "3.14").
Other important times where a dot appears in an equation, which unfortunately we were unable to cover, include points included 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 "=:").
To find determine whether or not a contour is such a symbol, we at first can determine it's size. If it has a significantly larger width or height than the line thickness (for instance twice as large), we may rule out that this symbol is a dot of any kind. The size however, does not give a good indication as to what type of dot this may be. The different types of dots can only be distinguished by their position in relation to other symbols. As the line has already been ordered, we are able to take a look at the preceding and following contour. If the contour in question is either the first or last contour in a line, we can rule out, that it is a multiplication dot or a comma, as these never appear at the beginning or end of a line. Next we compare its position relative to its neighbouring contours.
If the contour is closer to the lower end of the contour than the mid point for both of its neighbours, then the contour is classified as a comma.

If the contour is closer to the mid point than the lower end of the contour for both of its neighbours, then the contour is classified as a multiplication dot.

3.5.2 Extract Subimages for the Neural Network

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 that mask to the binarized original image. The result is a proper representation of what was written without the noise.

3.5.3 Symbol Classification

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 *)*, as well as the letters x and y. All kinds of dashes like the ones used in an equal sign or a minus symbol will not be recognized by machine learning techniques but will be handled by manual classification.

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.

Dash- or dotlike symbols however are extremely hard to recognize automatically, since they cause major overfitting within the network. A highly accurate classifier is crucial for our program to work. For this reason, we perform a manual classification of difficult symbols and leave the automatic classification for symbols with distinctive features.

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[**hasy**] 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.

Custom training images Furthermore, we generated some training images ourselves by writing down symbols and capture them the same way in which our application will capture the mathematical formula during usage.

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*[**adam**] optimizer. Since it is claimed to work best on classification issues with multiple classes, we use cross entropy loss as a loss function.

We decided to leave 11 different symbols to the automatical classification: The digits from 0 to 9, and $+$. Therefore, the classification problem we are trying to solve is very close to MNIST what makes the chances high to achieve a good accuracy. Other operators will be classified in an earlier step in a manual way. Thus, our classifier needs to map the images to one of 15 labels.

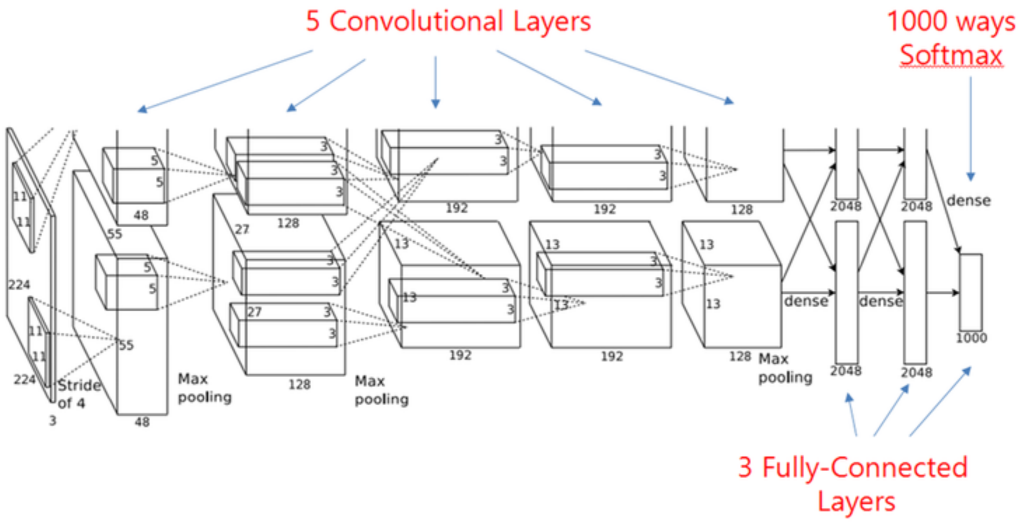


Figure 5: The AlexNet architecture

Neural network The network architecture we decided on is *AlexNet*[**alexnet**]. 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 5). 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 1 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 1: 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 class simply contains an *AlexNet* instance in which we load the information from the trained model. To predict what each of our segmented images shows, the class provides a function `classify`. The function takes a `NumPy` array with images as an input, creates a tensor out of this array and leads the data through the network. Afterwards the predicted label is returned as a string and can be further processed by the symbolic math solver `SymPy`.

Manual Classification Before the extracted symbols are given as an input for the automatic classifier, we check manually if they picture a symbol that our classifier can not recognize. This applies for opening and closing brackets, minus symbols, equal signs, commas, and multiplication dots.

3.6 Calculations

4 Discussion

In this section, we will discuss the progression of this project regarding the result.

4.1 Pre-Processing

Transforming the original image into a usable binary image worked relatively smoothly. The main issues which arise during this part are caused by the large variety of images. These vary in brightness, background color and medium of writing to name a few. However, it was possible to develop a method which consistently gives good results.
This step had to have a very low false negative rate. In other words, we had to allow for a small amount of noise rather than accidently have symbols or parts of symbols disappear. The simple pre-processing step performed well on this task.
Images were already usable after preprocessing. However, often contours could be found which did not belong to the equation itself. These contours can almost always be classified as either objects which are not text (for instance the border of a piece of paper) or as simple noise (caused for instance by the texture of the paper). They can be countered for separately.

Unwanted contours caused by background objects are often close to or connect to the border of the image. They could often be removed with background removal mask. While this often worked, this failed when the colour of the background object does not differ strongly from the background colour. This causes the border of the object to be inconsistent which may leave some part of the border in the image.

Contours created by noise tend to be small and appear either statistically isolated or in large clusters.
Isolated pixels of noise are often caused by small aberrations in the texture of the writing medium like paper.
Noisy Pixels which appear in larger groups can often be attributed to reflections of light sources. This happens more often on white boards.
Both types offer only minor variations of the brightness of the writing medium

and can therefore be countered by using a mask which removes all pixels which are of higher than average brightness in the original image.
While this method is in no way sufficient to detect text, it gives us a fair indicator as to which pixels are not part of the equation. This doesn't work if the equation is brighter than the medium it is written on (e.g. a chalk board). A simple inversion of the brightness of the original image might be able to solve this problem. However, reflected light sources (whose brightness may well be bright than the text) would present a problem.

4.2 Classification problems

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.

5 Conclusion

6 Installation and Usage

6.1 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
   ```

6.2 Usage

In the project directory, run the program with:

```
$ python3 app.py
```