AML Final Project: Creating an Image Input Calculator

Timothy Jay Herbst, Fenja Kollasch, Duc Anh Phi

September 22, 2019

Abstract

This paper deals with the development of a calculator, which can infer, from an image of an equation, its solution. To do this, the image will, after going through some preprocessing, be segmented into the separate symbols that make up the equation. These symbols will then be ordered and classified, which allows solving of the equation. 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 sophisticated 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, uses less time to calculate the solution of most simple mathematical problems than it does for the user to enter the equation. 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 of equations, when given an image of the equation. In the following paper we will 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

3 Preprocessing

Our program receives at first a simple unaltered image. From this image it is difficult to detect significant features, such as the symbols that make up the equation. Therefore, it is necessary to do some preprocessing, before further steps make sense.
We are facing several issues in this. Some improvements we could make include:

- Correction for different Quality of Camera, Lighting, etc.: The incoming images vary to a great extent due to changes of lighting and may come from different cameras. Correcting for this, is not a simple task. A future improvement of this project would improve greatly by having an adaptive preprocessing step.
- Simplification of the input dimensions: Explicitly we want our image to be a black-white image, with the only two colours being black (0) and white (255).

- Connectedness of the symbols: For later steps it is usefull 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 on the image. (The human eye is very good at filtering these out.)
- Background Contour removal: Often times when taking an image of a contour there will be other objects in the image (e.g. pencils, the border of the paper, etc.).

We correct for (most of) these issues in two separate steps:
At 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.

3.1 Simple Preprocessing

The image our programm receives as input is likely not easily analysed. We therefore manipulate 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 be the most usefull for our purpose.
It operates in five steps:

- Grayscaling: At first we grayscale our image. Here we reduce the BGR (blue, green, red) image into an image with a single dimension per pixel. This is usefull, because this simplifies our image and allows us to more easily analyse the image.
- 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 Threshold: We add a filter that applies a threshold. Any pixel below this threshold gets set to 0 any pixel above it gets set to 255. This threshold differentiates the symbols from the background. We will later explain in more precision how this filter works.

- Dilation: After the previous steps it is possible that some of the symbols have deteriorated into several parts. (Because our image is inverted, the code uses an erosion function.) Therefore, it is useful at this point to thicken the contours.

3.2 Border Removal

After the steps above, we determine a fairly good image. However, very often the shape of background objects register as contours. Our mission in the following section is to be able 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.

3.3 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)):

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

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

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

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

Here α is the normalisation factor and σ is the recommended standard deviation for a box size of 11.

4 Segmentation

5 Line Assigning and Ordering

After finding the Contours we now face the problem, of bringing them into an order, where they then can be classified and calculated. We order them before classifying them, because this allows us to remove certain contours, based on criteria, we will go into later. This saves the program from having to classify contours caused by noise. Thereby our program becomes more efficient. It also becomes possible to classify some symbols, based on their size and position. Thereby, we can reduce the amount of classes we need to identify with the classifier.
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.

5.1 Problems with Determining Line Order

5.2 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.

5.3 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.

6 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.

6.1 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 applied a padding of 2 pixels to the images and 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.

6.1.1 MNIST

As a basic training routine, we used the already mentioned MNIST database of handwritten digits[**mnist**]. `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.

6.1.2 HASY

Mathematical symbols such as brackets or plus signs 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. We remain with roughly 6000 images after the filtering. Since MNIST contains 60000 images of digits, we have a critical overflow of digit characters in comparison to mathematical operators and letters. To reduce this gap, we 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.

6.1.3 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.

6.2 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 15 different symbols to the automatical classification: The digits from 0 to 9, the letters x and y, as well as the symbols *(*, *)* and $+$. Other characters such as - or $=$ will be classified in an earlier step in a manual way. Thus, our classifier needs to map the images to one of 15 labels.

6.2.1 Neural network

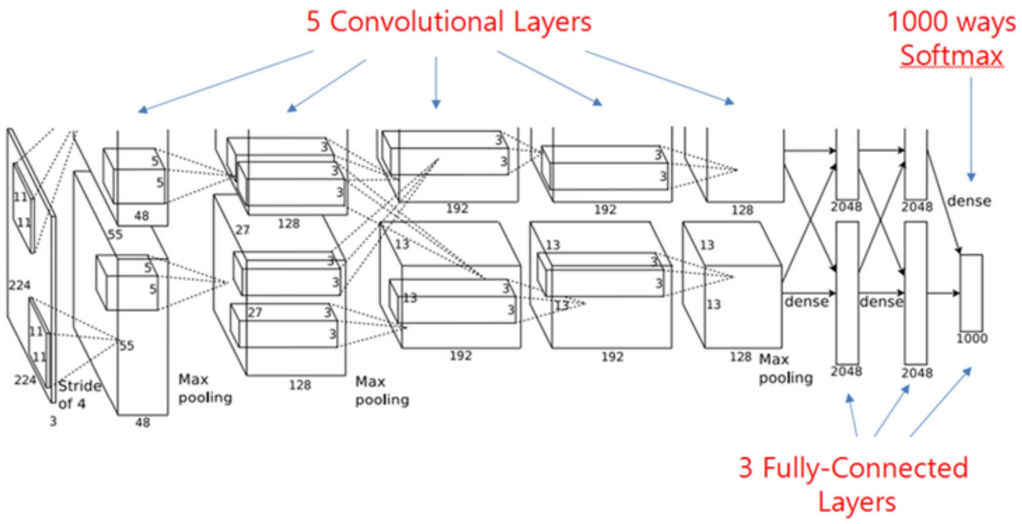

Figure 1: The AlexNet architecture

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 1 and 2). 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.

6.2.2 Hyperparameter training

`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

6.3 Classifier

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 furtherly processed by the symbolic math solver.

7 Calculations

8 Discussion

9 Conclusion

```
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)
  )
)
```

Figure 2: AlexNet layers in PyTorch