Camera Calculator App

An investigation into the implementations of neural networks for handwritten mathematical equation recognition.

Jerry Liu

Table of Contents

Analysis

Introduction

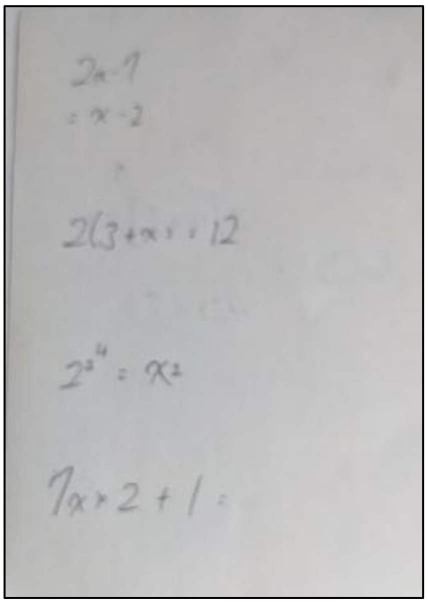
As a student, when tasked with completing math assignments and homework, seeking help from calculators is a common habit of many. However, as we progress through education the mathematical equations we face grow in complexity, where pre-installed calculators on phones and commonplace household calculators often offer little assistance. Online calculators although providing more functionality, transcribing mathematical expressions to text can often be a tedious task; some online calculators use the LaTeX language to express mathematical notation and symbols, but this requires users to learn its keywords. Other online calculators simply let users select symbols from a keyboard, which can also be time-consuming when dealing with longer expressions.

To improve the efficiency of inputting mathematical expressions and calculating them, I intend to create a mobile phone app which utilizes the camera functionality to recognize handwritten mathematical expressions and perform the desired calculations. I aim to be able to solve simple, one-variable linear equations and expressions, and gradually implement more functionality as the project progresses. The app will be designed for those who are looking for a quicker way to enter equations into their calculators.

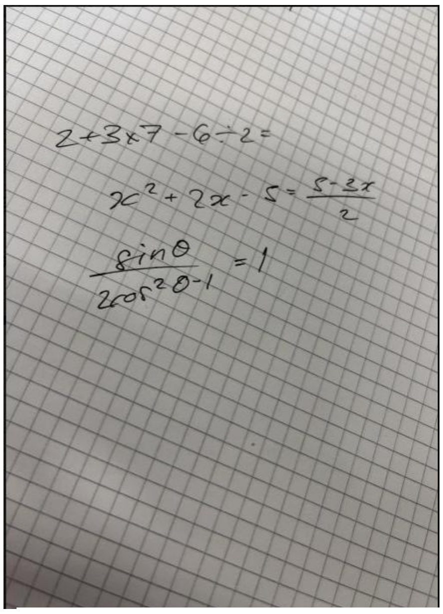
Investigation of Existing Solutions

To better understand the functionality and uses of current solutions available to users, I've identified 3 calculator/equation solver apps from Google play with the features of photographic input to investigate in and collected reviews from the play store and math students within the college. The apps will be evaluated by their user interfaces relevant to the camera feature, the accuracy of results, error handling and overall functionality.

To test the accuracy of input and calculation. I've collected handwritten equations from different fields of math from varying handwriting on different backgrounds to test the recognition of the software.

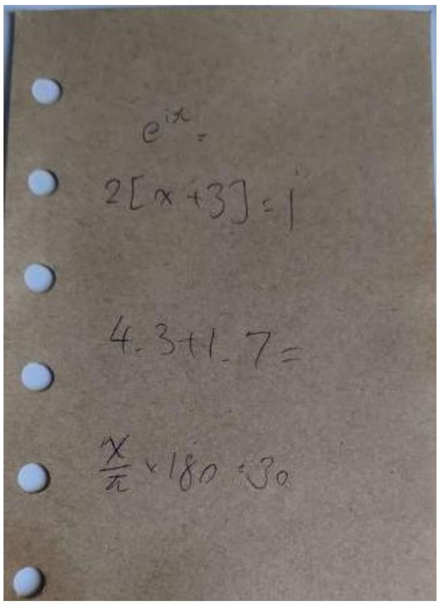


▲ *Blurred equations. Tests for recognition of equations written on multiple lines, indices and similar symbols (x and multiply sign)*



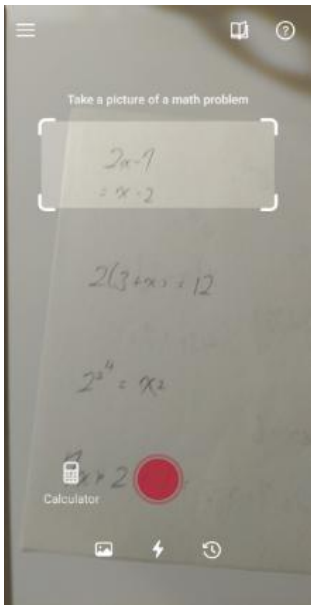
$2+3\times7-6\div2=$
$x^2+2x-5=\frac{5-2x}{2}$
$\frac{\sin\theta}{2\cos^2\theta-1}=1$

▲*Quadratics and trigonometry, test for recognition of theta and factors*



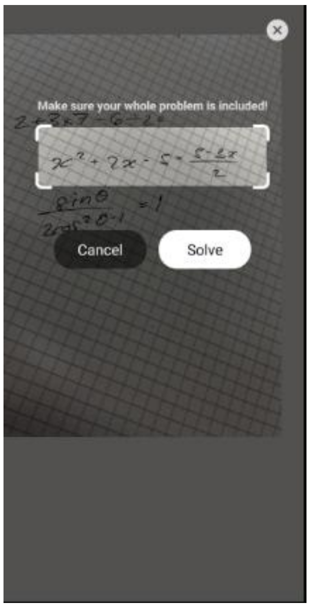
▲*Other symbols and brackets, including decimal points, pi and imaginary numbers.*

Product 1 - Photomath



Take a picture of a math problem

▲*Camera mode GUI*



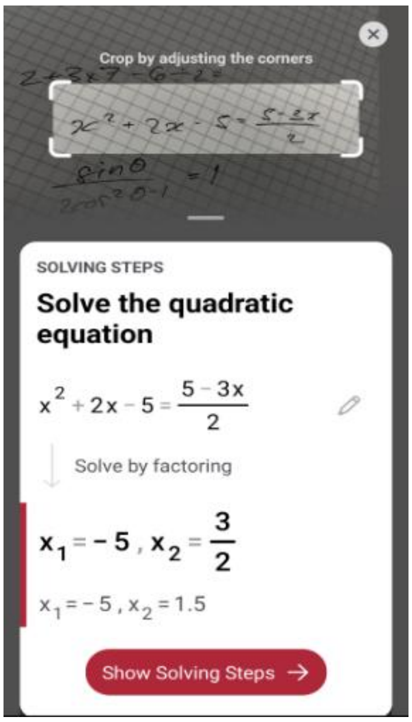
Make sure your whole problem is included!

▲*Photo import GUI*

Photomath was able to recognize and solve most of the equations set including indices and variables such as theta; the only failure was when tasked to recognize equations written on different lines, to which an error message was outputted. I suspect that is due to the nature of how the images are broken down into individual symbols for recognition and how they are reassembled and is not due to the inability to compute the equation as the program was able to compute the result when I rewrote it on the same line.

The GUI is simple; the view of the camera as the background with text prompts at the top of the screen and an adjustable white box to target the desired question. The straightforward design with minimal icons along with text prompts on the top allowed users to know what to do immediately; adjusting the size of the box as needed and capturing the image with the bright red button. The ability to adjust the white box and crop the image was particularly helpful in aligning the equations after capturing an image, along with the ability to cancel the action if an image is unclear or does not capture the entire question, reducing time wasted processing the image. Additional functionality such as allowing users to access their history using the icon in the bottom left and import images from their camera roll have proven to be very useful during testing as they allowed me to use blurry images such as the ones above.

The calculator displays the recognized equation and the type of operation performed, in this case, "Solve the quadratic equation", along with other features such as the graphical calculator below which plots the equation if applicable. The little pen icon on the right side of the recognized equation allows users to modify the input if there has been a recognition error. The solution is then clearly written below, with a red box allowing users access to the steps used to acquire the result.



SOLVING STEPS

Solve the quadratic equation

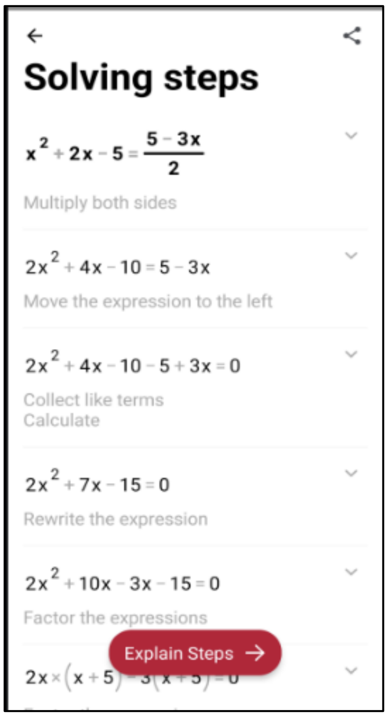
$x^2 + 2x - 5 = \frac{5 - 3x}{2}$

Solve by factoring

$x_1 = -5, x_2 = \frac{3}{2}$

$x_1 = -5, x_2 = 1.5$

▲*Recognition of equation and calculated solution*


Solving steps

$x^2 + 2x - 5 = \frac{5 - 3x}{2}$

Multiply both sides

$2x^2 + 4x - 10 = 5 - 3x$

Move the expression to the left

$2x^2 + 4x - 10 - 5 + 3x = 0$

Collect like terms
Calculate

$2x^2 + 7x - 15 = 0$

Rewrite the expression

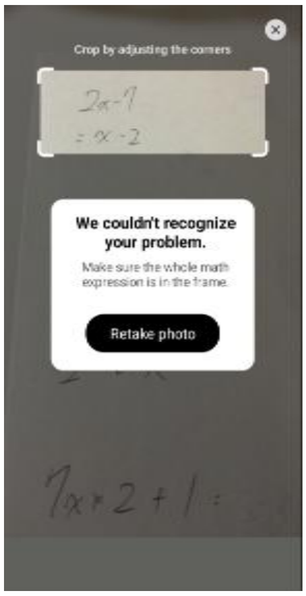
$2x^2 + 10x - 3x - 15 = 0$

Factor the expressions

$2x \times (x + 5) - 3(x + 5) = 0$

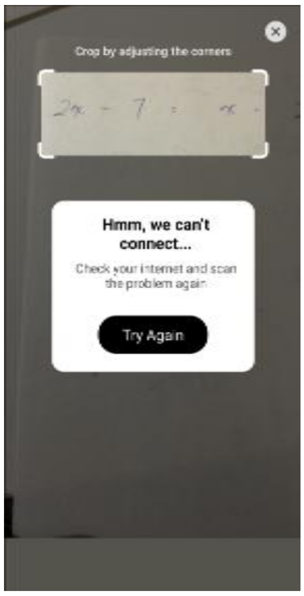
▲*Step-by-step display*

The error messages are large and clear, the prompt and proposed solution is user friendly and easy to understand:



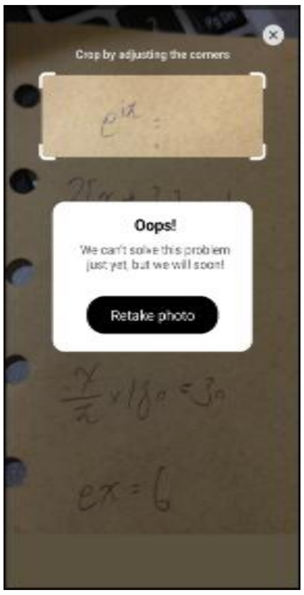
We couldn't recognize your problem.
Make sure the whole math expression is in the frame.

▲ *Unrecognized problem error*



Hmm, we can't connect...
Check your internet and scan the problem again

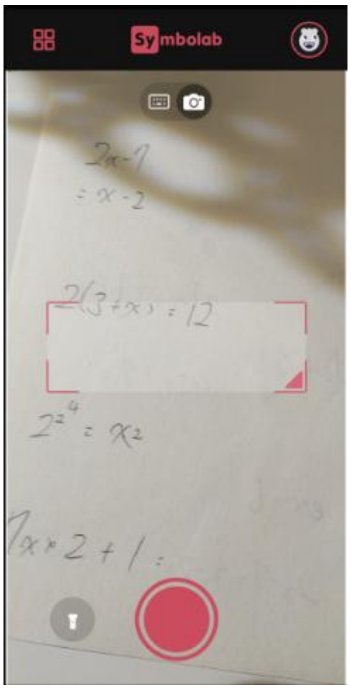
▲ *Offline error*



Oops!
We can't solve this problem just yet, but we will soon!

▲ *Unsupported equation error*

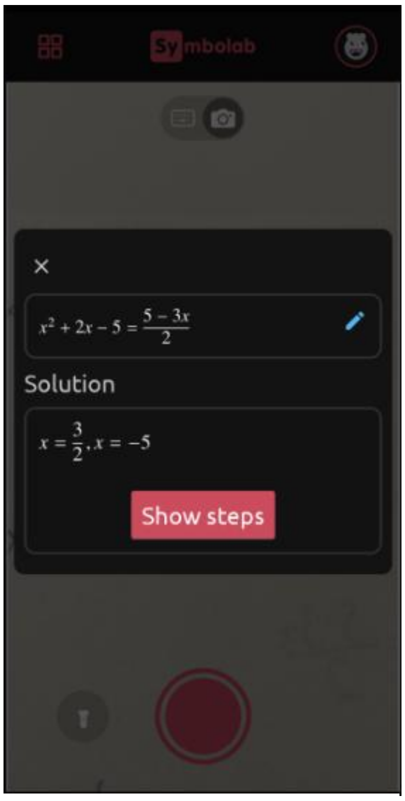
Product 2 – Symbolab



▲ *Camera mode GUI*

Symbolab is well-known for its powerful math engine and online calculators, which let you execute a variety of mathematical operations. Their smartphone app combines those into one and adds new features such as the camera input. This allowed *Symbolab* to perform more complex operations than the other calculators tested, such as imaginary numbers and limits, and was quite difficult to find exceptions or trigger error messages. It was also able to recognize equations composed of multiple lines such as simultaneous equations by registering the two lines as two separate equations, which I suspect was solved by implementing matrix calculations.

Similar to *Photomath*, *Symbolab*'s user interface features a translucent background with a target red box in the middle for alignment. It does not, however, offer access to previously typed equations or the ability to directly import photographs, unlike *Photomath*. Through interviewing maths students, it seemed that the ability to adjust the size and position of the red target box was not as clear, as many seemed to have struggled to align the box to the desired equations. It is also worth noting that *Symbolab* does not show the captured image and immediately starts to translate the targeted equation, leading to error messages if not targeted properly. After recognition, the recognized input is displayed along with the solution below it, and the option to display the steps taken. It allows the user to edit the inputted expression through the pen icon on the right of the expression, however Even after pressing capture the camera still runs in the background, which may cause distraction and take up display space that may have been used to show more information about the inputted equation.



Symbolab

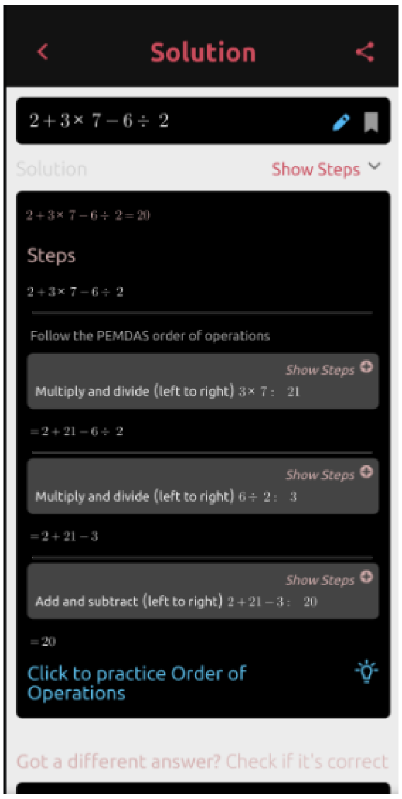
$x^2 + 2x - 5 = \frac{5-3x}{2}$

Solution

$x = \frac{3}{2}, x = -5$

Show steps

▲*Recognition of equation and calculated solution*



Solution

$2+3\times 7-6\div 2$

Solution Show Steps

$2+3\times 7-6\div 2=20$

Steps

$2+3\times 7-6\div 2$

Follow the PEMDAS order of operations

Show Steps

Multiply and divide (left to right) 3×7: 21

$=2+21-6\div 2$

Show Steps

Multiply and divide (left to right) $6\div 2$: 3

$=2+21-3$

Show Steps

Add and subtract (left to right) $2+21-3$: 20

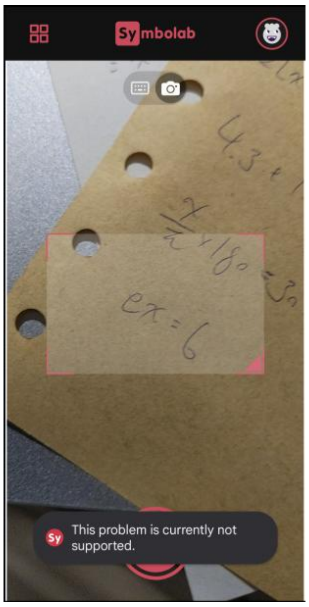
$=20$

Click to practice Order of Operations

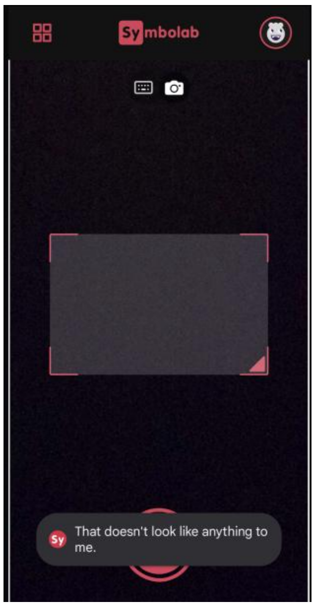
Got a different answer? Check if it's correct

▲*Step-by-step display*

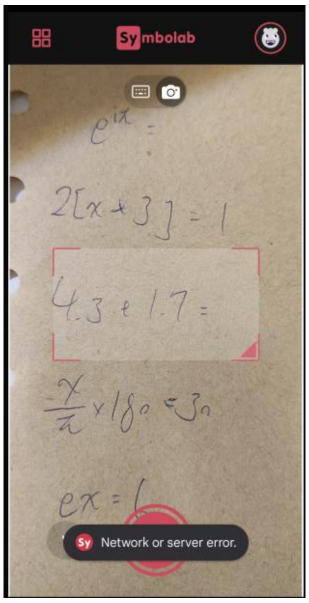
The error messages are small and often have a demanding tone and do not provide a clear solution to the errors shown, not user-friendly.



▲*Unsupported equation error*

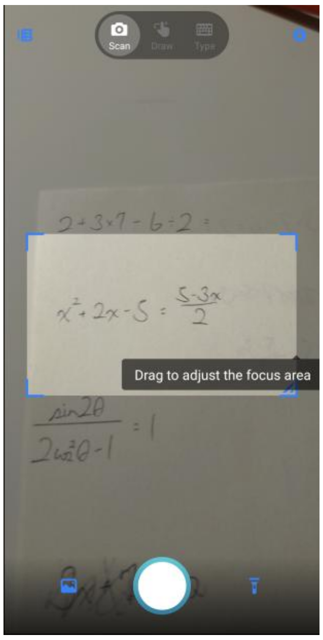


▲*Unrecognized problem error*

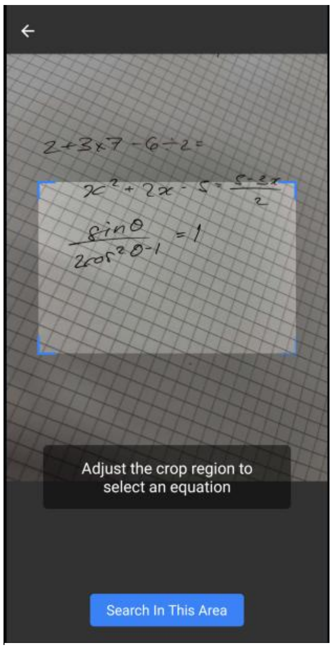


▲*Offline error*

Product 3 – Math



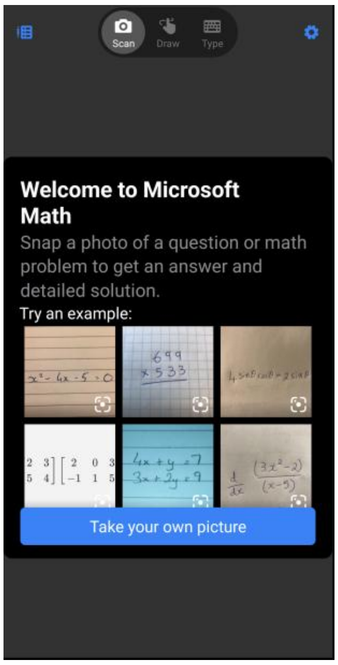
▲*Camera mode GUI*



▲*Photo import GUI*

Math by Microsoft is the middle ground between the two previous calculators, with a GUI and the ability to solve a variety of mathematical equations similar to that of *Symbolab*, with functionality such as importing images and history review like *Photomath*. After scanning the equations, the app was able to recognize all equations and even those the previous products failed to process. However, I did find slight errors in recognizing decimal points, which the previous two did not exhibit the same problem. Although this could be easily fixed with the editing tool accessible through the pen icon.

Upon first use, the app shows text prompts showing how to navigate the interface, along with example pictures to demonstrate the functionality of the software. With a similar "capture and crop" system, *Math* prompts the user to focus on specific questions on the captured image using the blue box, sharing the advantages of *Photomath.*

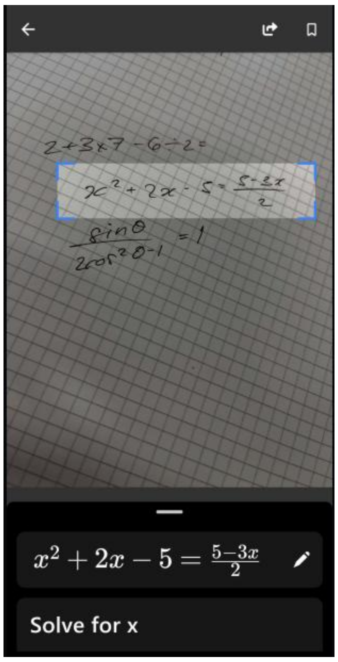


Welcome to Microsoft Math

Snap a photo of a question or math problem to get an answer and detailed solution.

Try an example:

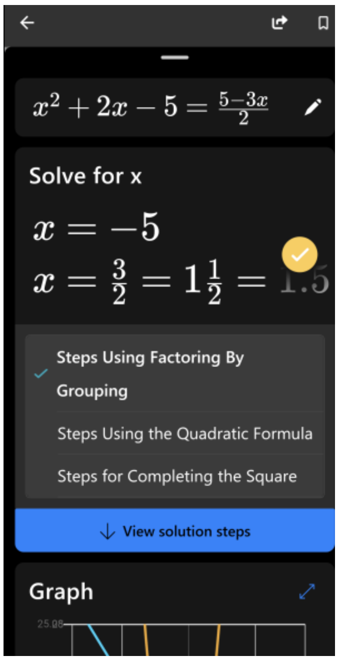
▲*Example pictures for demonstration*



$x^2 + 2x - 5 = \frac{5-3x}{2}$

Solve for x

▲*Recognition of equation and calculated solution*



$x^2 + 2x - 5 = \frac{5-3x}{2}$

Solve for x

$x = -5$

$x = \frac{3}{2} = 1\frac{1}{2} = 1.5$

Steps Using Factoring By Grouping

Steps Using the Quadratic Formula

Steps for Completing the Square

▲*Step-by-step display*

Proposed Solution and Limitations

My solution will be a phone application software that uses the phone camera to take pictures of handwritten mathematical expressions, which a neural network then uses to recognize the expressions and carry out the desired operations. The application will be aimed at those who have just started learning algebra or simply needs a faster way to enter expressions into the calculator. The challenge in this project will be the recognition of handwritten mathematical expressions, for which I will be implementing a convolutional neural network to perform the task of classifying the different symbols and numbers to be inputted into the calculator. From gathering information from existing software, I have targeted some key functions and limitations of my project:

1 The application will be able to be used offline to allow users to access it anywhere
2 The initial stages of development will be aimed toward solving linear expressions and expressions; however, the implementation of trigonometric identities and quadratics will be investigated.
3 The target box will be adjustable in size, cropping the desired expression after capturing the image.

Before listing more detailed objectives, further research will be done to gain a more detailed understanding of the methods used.

Further Research

As aforementioned, this project will heavily rely on the ability of the software to recognize handwritten digits and mathematical equations, which I will be implementing in convolutional neural networks to perform optical character recognition. To research further into the topic, I have referenced "Neural Networks" by Michael Nelson which describes a similar problem in recognizing handwritten digits through training a neural network using the MNIST dataset, as well as SciML's GitHub repository on CNNs. All resources referenced will be cited in the References section.

Modelling

Here I will lay out the basic flow of the application to provide a basis for the objectives which need to be set and will be further investigated in the Design section of the report.

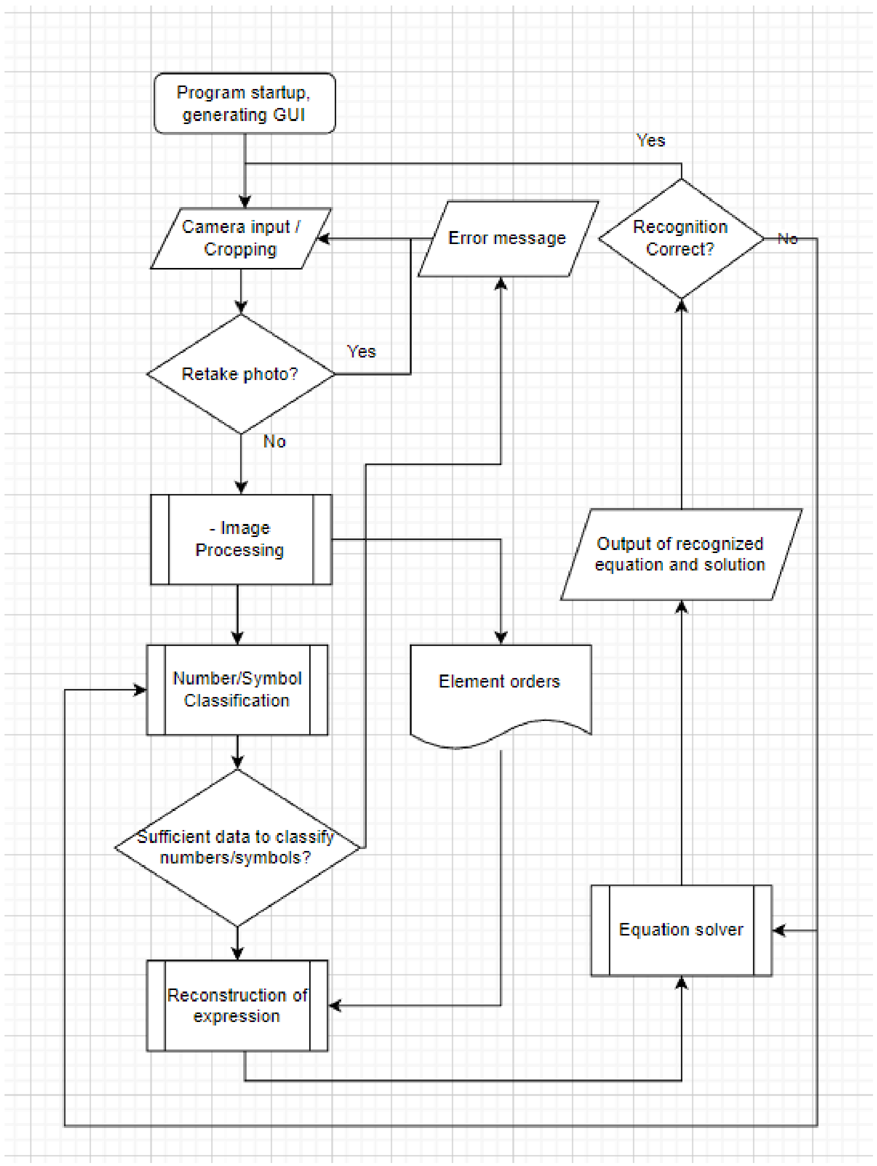
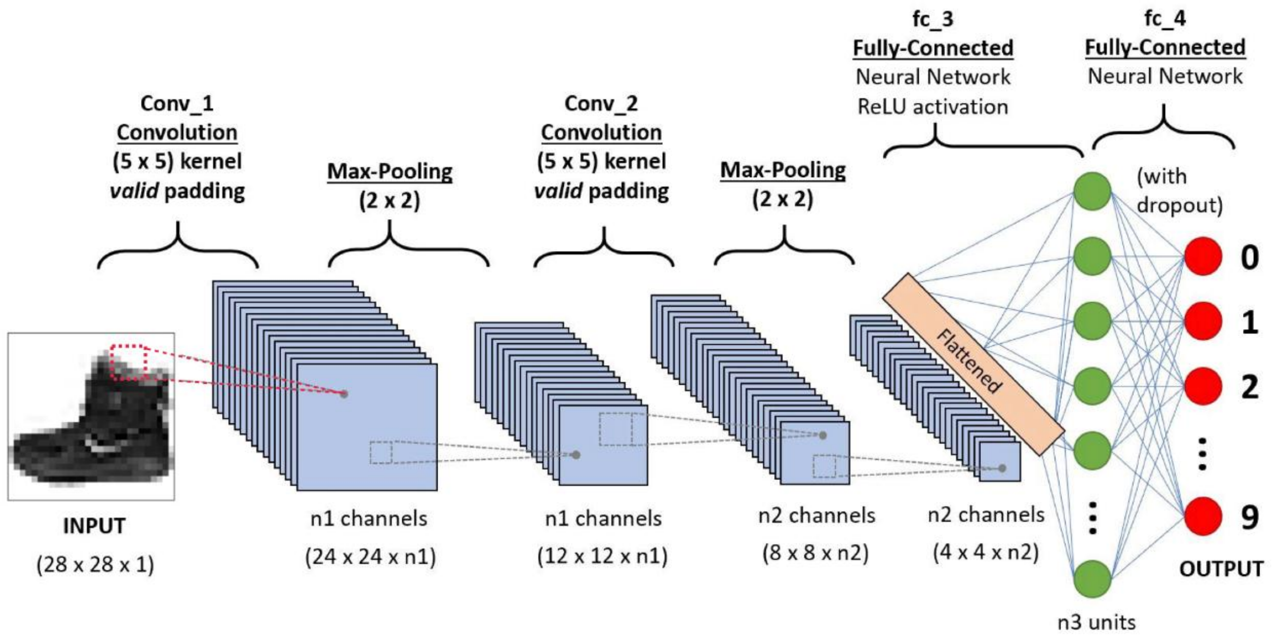

Program startup, generating GUI
Yes
Camera input / Cropping
Error message
Recognition Correct?
No
Retake photo?
Yes
No
- Image Processing
Output of recognized equation and solution
Number/Symbol Classification
Element orders
Sufficient data to classify numbers/symbols?
Equation solver
Reconstruction of expression

Image Processing

Before the classification of numbers and symbols can be performed, the image needs to be preprocessed to allow uniform inputs into the neural network. The image will be denoised and reshaped using OpenCV, and segmented into its symbols. During this process, the segmented parts will be labelled to allow for the reconstruction of the data after recognition, in which the order may be stored as an attribute to the symbols or numbers.

Neural Networks

I have chosen to use Convolutional Neural Networks to perform optical character recognition due to their efficiency in image recognition as they require less preprocessing of the input images and the abundance of resources available. One of the biggest advantages of Convolutional neural networks is the ability to capture spatial dependencies; where a pixel's value is influenced by the value of nearby pixels; by applying "filters" to the image.



▲*An example of a CNN architecture used to classify clothing. I intend to implement a similar architecture.*

These filters, often referred to as "kernels," are typically 3 x 3 or 5 x 5 matrices that are multiplied by the image's pixel values to identify desired characteristics by sampling the same amount of pixels as their size (aka 3 x 3 kernels sample pixels in a 3 x 3 region); repeating the process across the image with a fixed "stride", which can be thought of the distance between each sampled region. The values within the kernels are the weights which may be optimized through training where machine learning algorithms such as backpropagation may be used. The application of the kernels will be done through layers, each layer in the CNN having a different purpose. In the typical CNN, the architecture consists of a combination of convolutional layers to filter the image, a max-pooling layer to standardize and group convoluted features for ease of computing, and a fully connected layer which is responsible for classifying the collected features into their respective classes.

Libraries such as TensorFlow will be used for the training and development of the neural network, whereas training data will be sourced from Kaggle, which will be cited in the References section.

Equation solving

After the recognition of numbers and symbols, the next step would be representing them in a malleable manner for it to be solved by translating them to Python's syntax such as "÷" to "/" for example. This may be performed by setting the output parameters of the neural network to be already in Python syntax, however, this may require modification of the training data as the labels on the data may differ. We can also employ libraries to perform the translation, by setting the recognized symbols as the key and their python equivalents as the element. To reconstruct the inputted expression, the order will be arranged into order by referencing the "elements order array" defined during preprocessing.

Objectives

1 Allow image capturing through the app's user interface.

- 1.1 The app will be able to request access to the phone's camera.
- 1.2 The captured image will be able to be collected by the software for processing.
- 1.3 The captured image will be able to be converted into a malleable format (NumPy array or otherwise).
  - 1.3.1 The converted data will be able to be stored in the system for later use.

2 The app must have a graphical user interface, allowing the targeting of required math expressions.

- 2.1 A target box will be presented to the user with the ability to be adjusted by dragging the corners, however, the centre of the box will be fixed in place to avoid awkward positioning of the target box if moved.
- 2.2 There will be a text prompt near the target box explaining how to use the target box and how to adjust it.
- 2.3 The target box will allow for cropping after the capture of the image.
- 2.4 There will be options to allow users to retake the image if unsatisfied.

3 The app will be able to segment the individual symbols with sufficient accuracy for them to be distinguished.

- 3.1 The app will be able to de-noise and filter the image for easier processing during recognition
- 3.2 The app will be able to store the order of the segmented elements in the form of an attribute of the segmented image or otherwise to allow for later access.

4 The neural network must be able to apply changes to the converted data (as mentioned in 1.3) to allow for filters to be applicable.

- 4.1 Matrix multiplication of kernels and the respective pixel values on the image must be correct.
  - 4.1.1 Kernel parameters including size and stride must be defined and changeable.
    - 4.1.1.1 The Kernel will be defined as a function with size and stride as parameters.

4.1.2 The values within the kernel will be able to be modified during the training process through forms of backwards propagation.

4.1.3 The amount of padding applied through each layer must be definable

4.1.3.1 The padding process is to be defined as a function with the level of padding as a parameter.

5 The layers within the CNN must be able to connect with each other.

5.1 Data must be able to be passed to the next layer while maintaining the integrity of the original data. There should be no unwanted changes to the ordering of the values.

6 Activation functions of the layers must be correctly implemented.

6.1 Activation functions must give correct values when tested individually.

7 Output of the classification must be stored in some form in relation to the original ordering it was in before segmentation (as a reference to 3.2)

8 If insufficient evidence is gathered to deduce the equation entered, an error message will be displayed telling the user, providing the option to reactivate the camera to take another input.

9 The identified numbers and symbols must be able to be translated into a usable form (python syntax)

9.1 Relationships between identified numbers and symbols and their respective mathematical notation in python must be defined.

10 The translated form of the full equation must have the same ordering or be mathematically equivalent to the original equation.

11 The translated form of the equation will be presented to the user and allow for feedback on the accuracy of the recognition

11.1 It will allow users to edit the outputted equation and coefficients.

11.1.1 Implementation of a textbox or other form of text input.

12 The equation solver will be able to correctly output the calculated solution

12.1 If the equation has a variable, the solution should have "x=A", where A is the answer.

12.2 If the equation is an expression, only the value should be outputted.

13 The output should be able to be closed and the camera will reactivate to allow for input from the user, the process repeats.

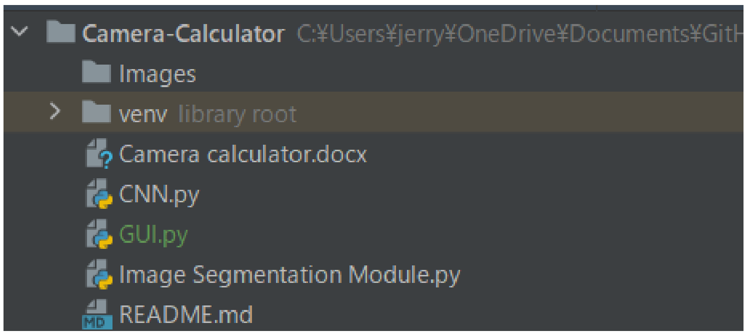
13.1 An implementation of a flag to check if the user is using the output box.

Design

Overview

As aforementioned, I will be splitting the process of my app into 4 sections :

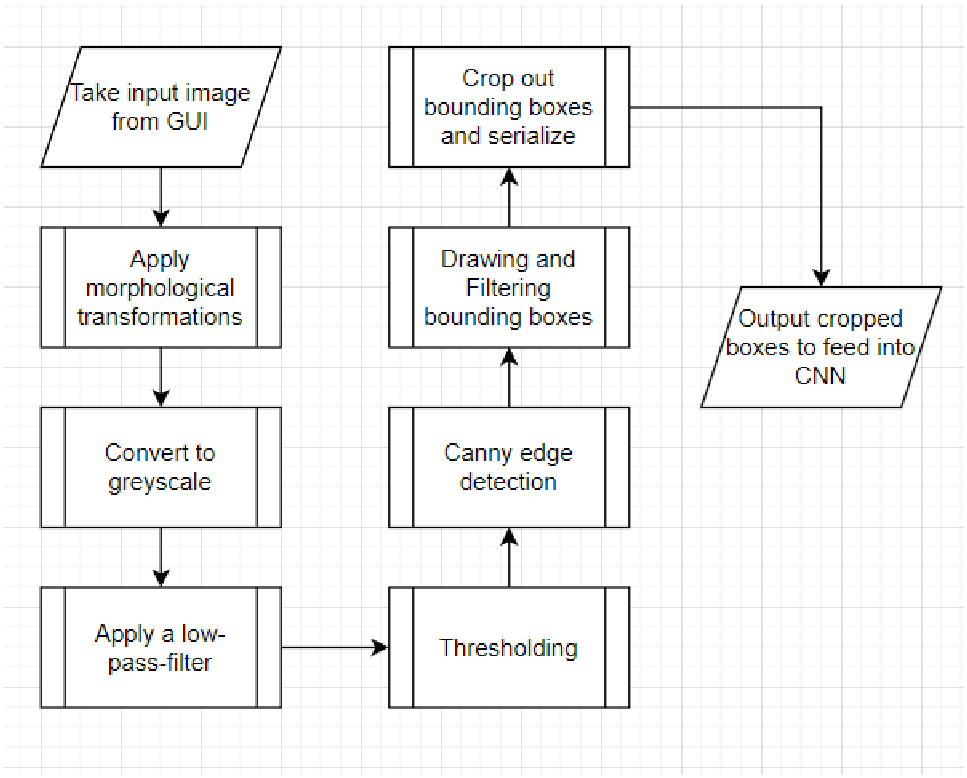
- Image pre-processing and segmentation
- UI (Kivy and Obtaining camera images)
- Convolutional neural network and recognition of symbols
- Post-processing (reconstruction and solving of equations)



```
Camera-Calculator C:¥Users¥jerry¥OneDrive¥Documents¥GitH
    Images
  > venv library root
    Camera calculator.docx
    CNN.py
    GUI.py
    Image Segmentation Module.py
    README.md
```

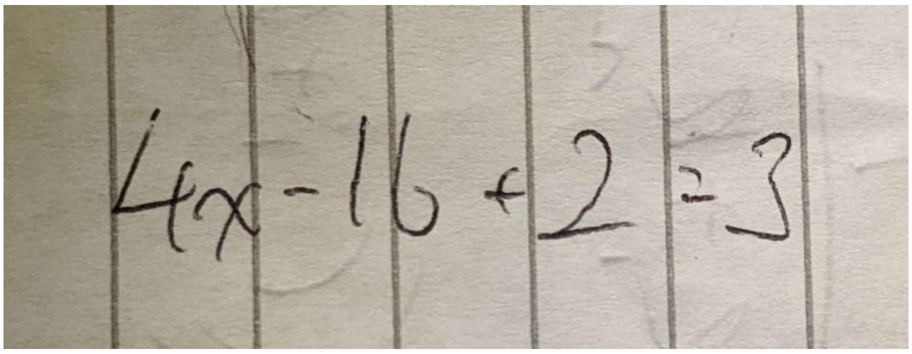
Image Segmentation and Pre-processing:

This is the first process after the image has been retrieved from the camera, which consists of removing background features and denoising, followed by thresholding and edge-detection to draw bounding boxes around the numbers and symbols for them to be later cropped out and inputted into the neural network to be recognized. The process will follow the flowchart below:



Take input image from GUI
Apply morphological transformations
Convert to greyscale
Apply a low-pass-filter
Thresholding
Canny edge detection
Drawing and Filtering bounding boxes
Crop out bounding boxes and serialize
Output cropped boxes to feed into CNN

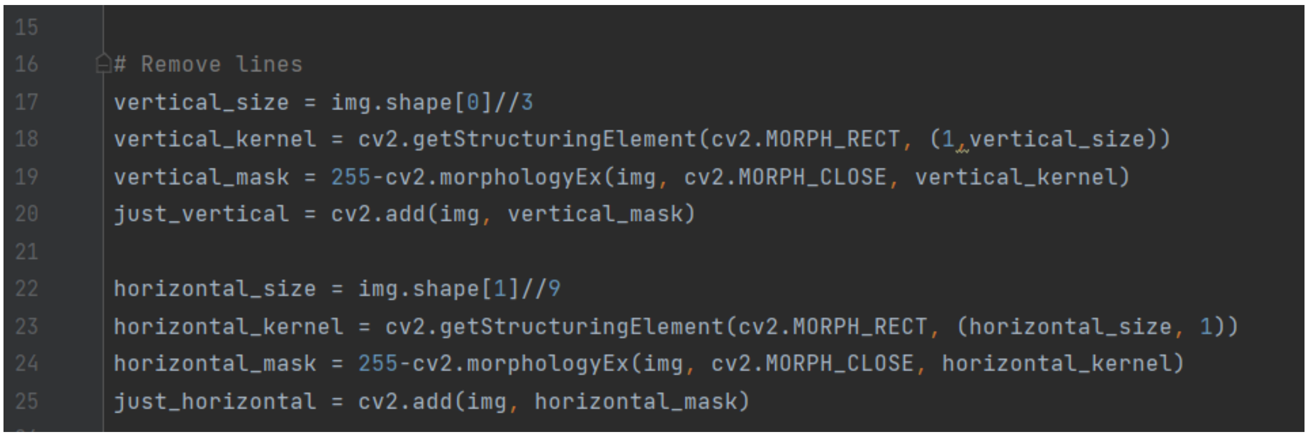
For this process, I have incorporated the OpenCV library, a library widely used for image processing and allows for quick implementation of my ideas.



▲*Original image with a colour gradient in the background as well as lines.*

Removing Background Features

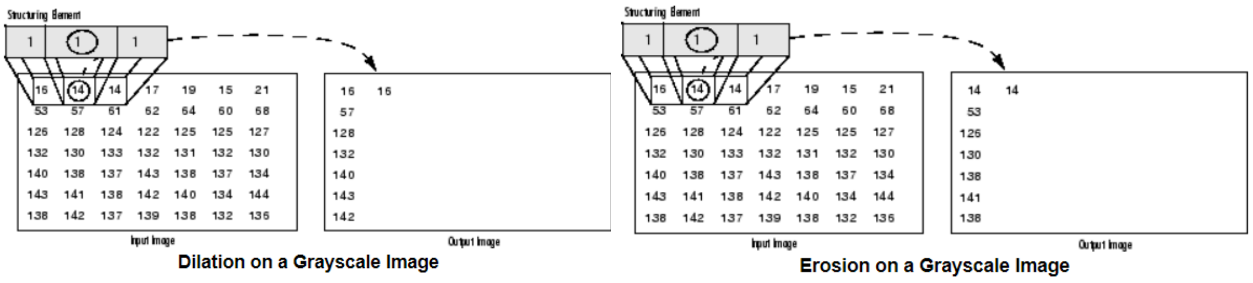
Many of the images inputted will consist of notebook lines and other background features which may affect later processes to correctly filter out wanted features. I have experimented with many methods, from changing contrast and brightness to blurring the image to reduce background features, however, through research the most effective method seems to be through implementing morphological transformations. This is done through a 2D-convolutional layer, by applying a kernel in which the shape is defined to identify a certain structure (also known as structuring element, the two will be used interchangeably), we can extract and remove unwanted lines from our image.



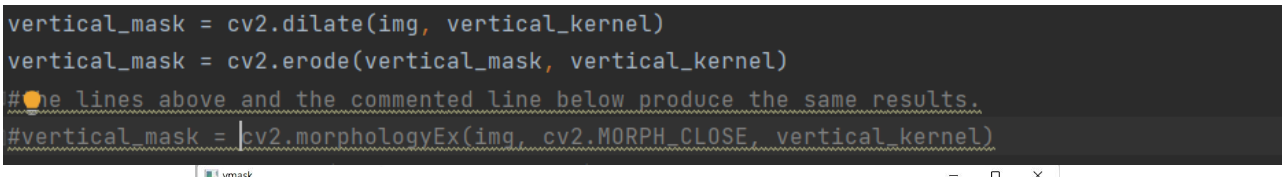
```

# Remove lines
vertical_size = img.shape[0]//3
vertical_kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (1,vertical_size))
vertical_mask = 255-cv2.morphologyEx(img, cv2.MORPH_CLOSE, vertical_kernel)
just_vertical = cv2.add(img, vertical_mask)

horizontal_size = img.shape[1]//9
horizontal_kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (horizontal_size, 1))
horizontal_mask = 255-cv2.morphologyEx(img, cv2.MORPH_CLOSE, horizontal_kernel)
just_horizontal = cv2.add(img, horizontal_mask)
```

▲*Background line removal prototype*

cv2.getStructuringElement allows us to easily create a kernel which fits the shape we need from the default selection, which is through cv2.MORPH_RECT we can create a rectangular matrix of size defined in the second parameter. Each matrix consists of ones and zeroes, which are compared to the input image to produce an output in a process known as morphological transformation. The morphological transformation I have implemented is known as a closing operation, which consists of two consecutive basic transformations: erosion, where the output Is the minimum of all the pixels when compared to the kernel; and dilation, where the output is the maximum of the comparison. Performing these two transformations sequentially allows us to extract the vertical or horizontal features of the image as it removes the structures which don't fit the kernel shape (in this case, it's our numbers and symbols), and dilates the remaining features back to their original thickness.

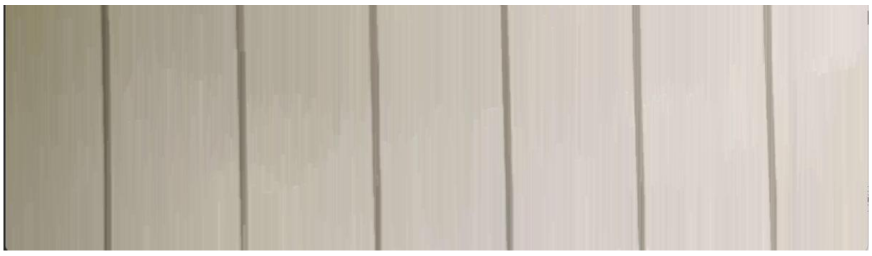


▲ *Visual explanation of a morphological transformation using a horizontal kernel*
https://docs.opencv.org/3.4/dd/dd7/tutorial_morph_lines_detection.html



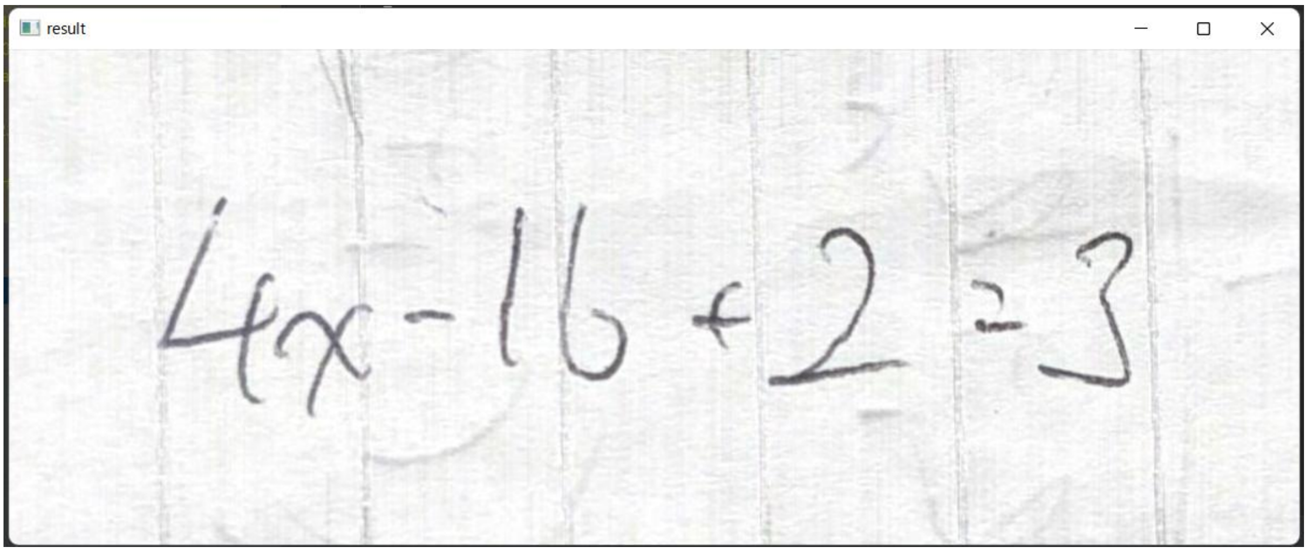
```
vertical_mask = cv2.dilate(img, vertical_kernel)
vertical_mask = cv2.erode(vertical_mask, vertical_kernel)
#The lines above and the commented line below produce the same results.
#vertical_mask = cv2.morphologyEx(img, cv2.MORPH_CLOSE, vertical_kernel)
```

▲ *Deconstructing the closing operation to its basic transformations*



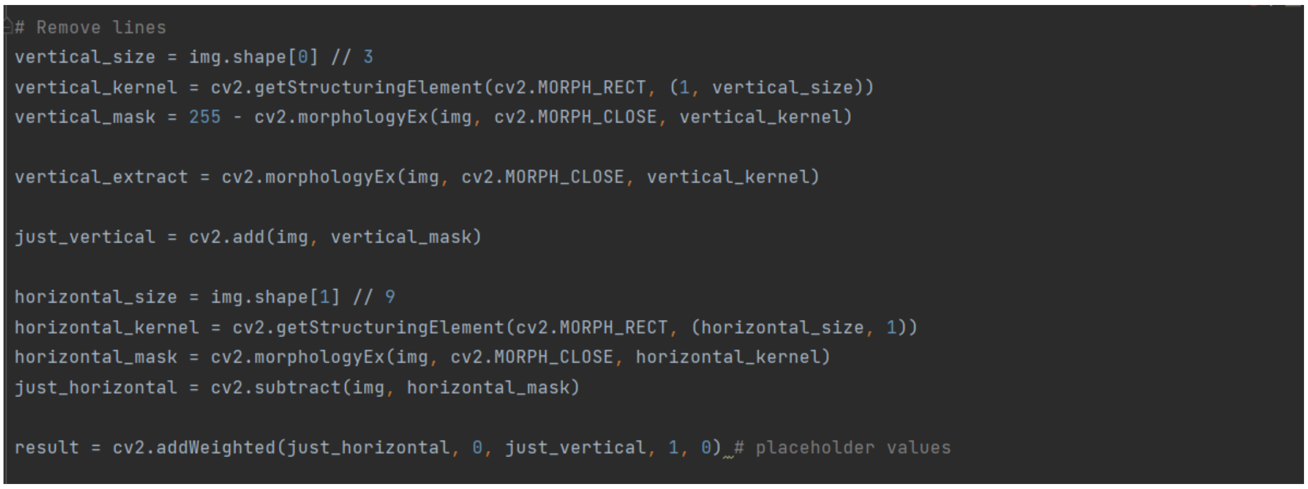
▲ *Extracted vertical lines from the original image*

The extracted image is then inverted and added onto the original image to produce an output without the vertical lines obstructing it. Conveniently, this also increases the brightness of the image, creating a more levelled background for thresholding.



▲ *Resultant image after adding the negative extracted image.*

To account for the different line directions the input image may have in the background, I have applied morphological transformations in different directions and merged the resultant images with weights to produce a result free of background obstructions. However, at the current moment of development, the weights introduced are tailored for this specific example. I aim to implement a function to generate weights for the weighted sum of the images by how many features are extracted.

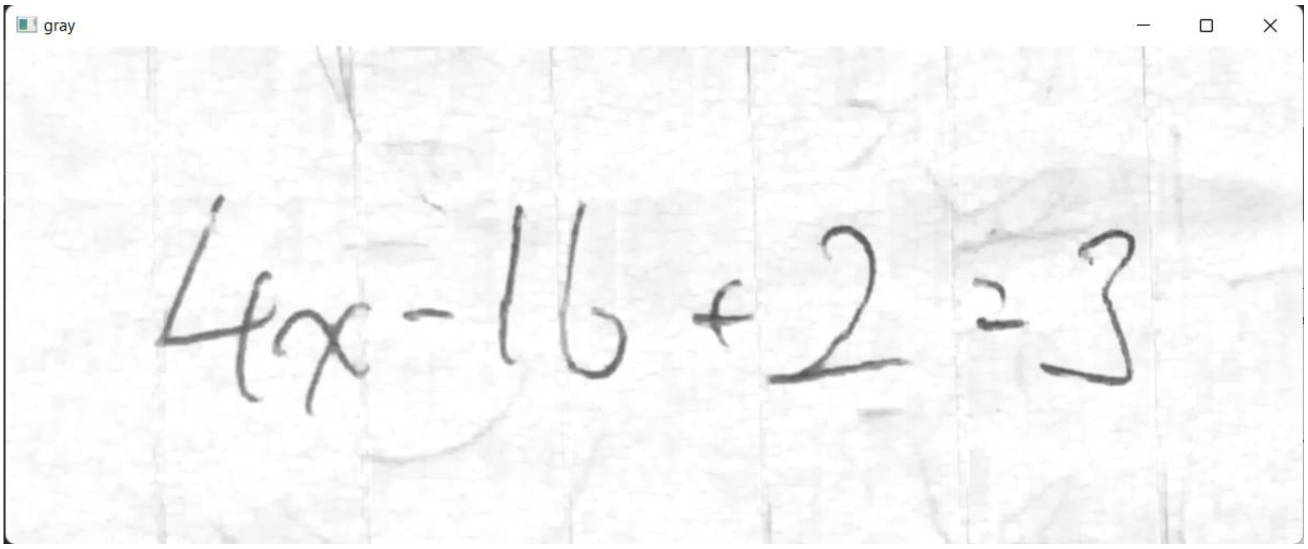


```
# Remove lines
vertical_size = img.shape[0] // 3
vertical_kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (1, vertical_size))
vertical_mask = 255 - cv2.morphologyEx(img, cv2.MORPH_CLOSE, vertical_kernel)

vertical_extract = cv2.morphologyEx(img, cv2.MORPH_CLOSE, vertical_kernel)

just_vertical = cv2.add(img, vertical_mask)

horizontal_size = img.shape[1] // 9
horizontal_kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (horizontal_size, 1))
horizontal_mask = cv2.morphologyEx(img, cv2.MORPH_CLOSE, horizontal_kernel)
just_horizontal = cv2.subtract(img, horizontal_mask)

result = cv2.addWeighted(just_horizontal, 0, just_vertical, 1, 0) # placeholder values
```

▲ *Consecutive morphological transformations with weighted sum of results*

Converting to Grey Scale and Thresholding

For ease of edge-detection and further processing, the processed BGR image from the last step is converted to a grey scale image through the cv2.cvtColor function into a 2-dimensional array by stating the colour of the output image.



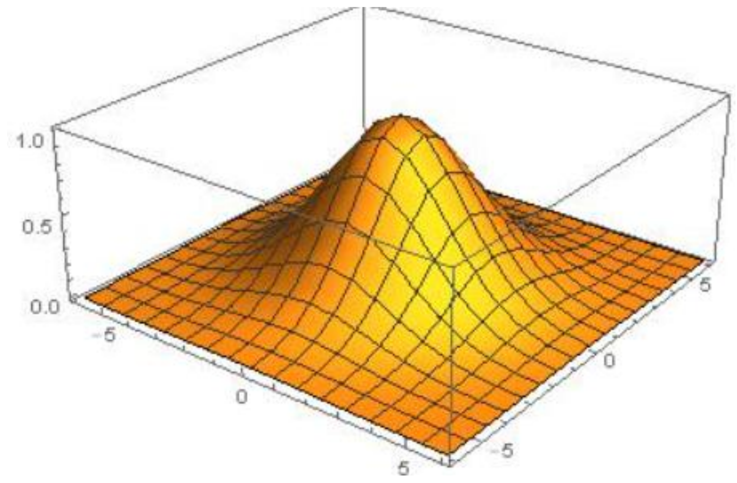
▲ *Image converted to greyscale (enhanced for visibility)*

A blurring function is applied to the greyscale image to remove noise and smoothen the edges from line removal with cv2.GaussianBlur. The algorithm used is a Gaussian blurring function, which again uses a 2D-convolutional layer to calculate a gaussian distribution of neighbouring pixels and re-assigns each pixel a value based on the weighted average using the Gaussian distribution, effectively reducing the intensity gradient in neighbouring pixels. This is done by applying the Gaussian function in 2 dimensions for the x and y components (in other words the rows and columns of pixels) and using it to create a kernel of a defined size where the values within follow the distribution.

$$\begin{bmatrix}
0.00000067 & 0.00002292 & \mathbf{0.00019117} & 0.00038771 & \mathbf{0.00019117} & 0.00002292 & 0.00000067 \\
0.00002292 & 0.00078633 & 0.00655965 & 0.01330373 & 0.00655965 & 0.00078633 & 0.00002292 \\
\mathbf{0.00019117} & 0.00655965 & 0.05472157 & 0.11098164 & 0.05472157 & 0.00655965 & \mathbf{0.00019117} \\
0.00038771 & 0.01330373 & 0.11098164 & \mathbf{0.22508352} & 0.11098164 & 0.01330373 & 0.00038771 \\
\mathbf{0.00019117} & 0.00655965 & 0.05472157 & 0.11098164 & 0.05472157 & 0.00655965 & \mathbf{0.00019117} \\
0.00002292 & 0.00078633 & 0.00655965 & 0.01330373 & 0.00655965 & 0.00078633 & 0.00002292 \\
0.00000067 & 0.00002292 & \mathbf{0.00019117} & 0.00038771 & \mathbf{0.00019117} & 0.00002292 & 0.00000067
\end{bmatrix}$$

▲ *Sample Gaussian filter with $\sigma = 0.84089642$.* https://en.wikipedia.org/wiki/Gaussian_blur

This helps us later in edge detections, as we'll come to see in the processed image the detected "blobs" are often discontinuous due to the unevenness of ink flow when writing as well as sensitive to any noise in the images, and therefore by smoothening out the image before applying an edge-detecting kernel we can keep the continuous features of the symbols we wish to focus on.

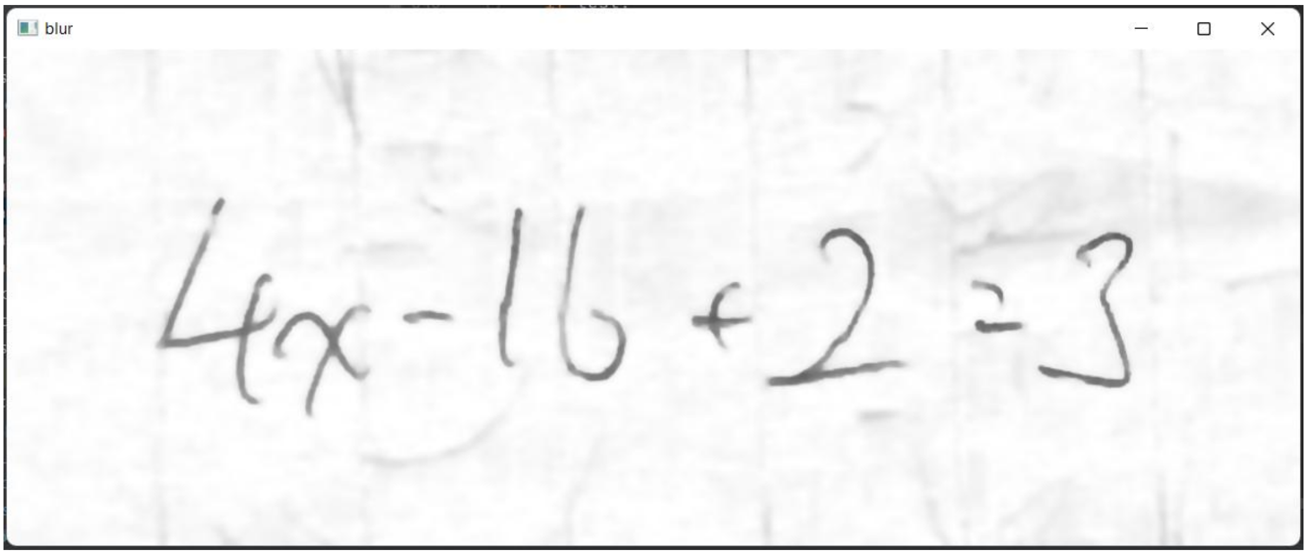


▲ *A 3d representation of the Gaussian kernel. As you can see, the pixels near the centre of the kernel are given more weight than the outer perimeters.* *https://codegolf.stackexchange.com/questions/123039/plot-the-gaussian-distribution-in-3d*

However, this proved to remove some features of the handwritten numbers within the image later on in the process as it blurs each pixel neighbourhood with the same weight. In order to preserve details of the written symbols, a gaussian blur but with pixel intensity sensitivity which adjusts the parameters of the gaussian blur depending on the difference in intensity between pixels within the kernel such that the larger the intensity difference, the less blur. This can be done with a bilateral filter such as Open CV's cv2.bilateralFilter(), which takes a the kernel shape, colour range and space parameters as input.

$$BF[I]_{\mathbf{p}} = \frac{1}{W_{\mathbf{p}}} \sum_{\mathbf{q} \in \mathcal{S}} G_{\sigma_s}(\|\mathbf{p} - \mathbf{q}\|)\, G_{\sigma_r}(I_{\mathbf{p}} - I_{\mathbf{q}})\, I_{\mathbf{q}}$$

▲ *$G_{\sigma s}$ is a spatial Gaussian that decreases the influence of distant pixels, where σ_s is the standard deviation in space (same as the normal gaussian blur) and σ_r is the standard deviation in intensity difference/colour range.*

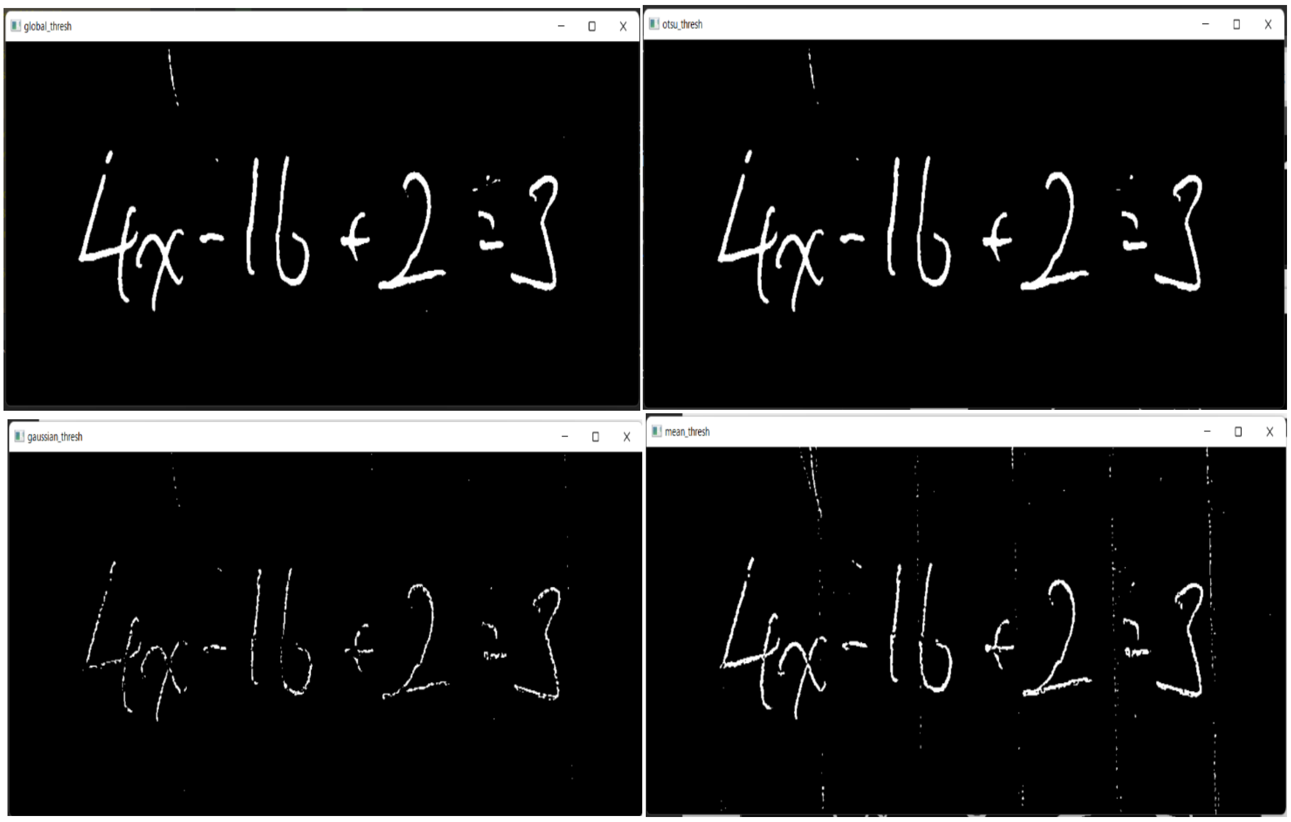


▲ *Applying bilateral filtering*

After blurring the resulting image is binarized through thresholding, by defining minimum and maximum values (thresholds) which if the pixel intensity is above the threshold value the pixel will be assigned as 255 (white) and 0 (black) if below.

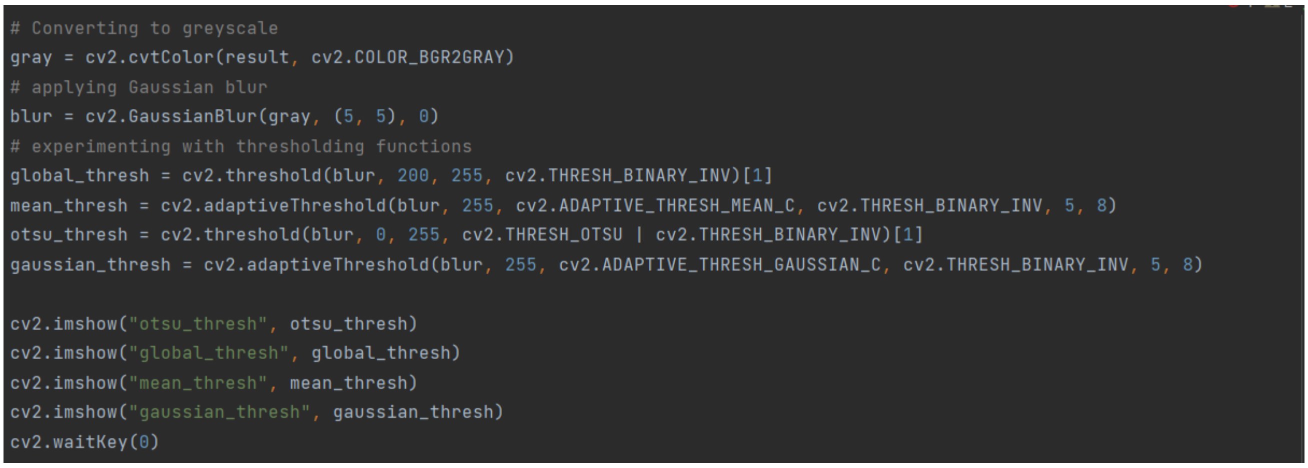
$$\texttt{dst}(x,y) = \begin{cases} \texttt{maxVal} & \text{if } \texttt{src}(x,y) > \texttt{thresh} \\ 0 & \text{otherwise} \end{cases}$$

Thresholding can be performed in OpenCV with a variety of functions, here I have experimented with different thresholding methods to find the most optimal one to work with my images. Notice that I have used inverse binarization (if the pixel value is larger than the threshold it is assigned 0 and vice versa) instead of the aforementioned binarization method, to allow for edge-detection algorithms which by convention we give wanted sections of the image higher intensity (in this case our numbers and symbols) and unwanted portions 0 when image processing. Details will be expanded further in the following section. For now, we can see the side-by-side comparisons of the different thresholding methods OpenCV offers:



▲ *Different thresholding methods*

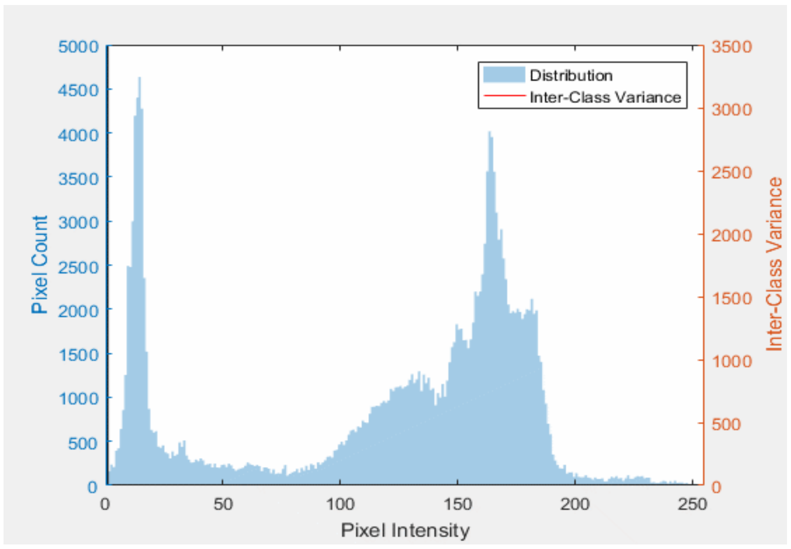
cv2.threshold applies the threshold function to every pixel in the image passed and returns the threshold value used and outputs the image in the format defined in the fourth parameter cv2.THRESH_BINARY_INV. In cases such as global thresholding, a fixed thresholding value is defined in the second parameter. Although this may produce good results as we'll see below, this threshold may not work for other images where the average intensity of the pixels is not known and therefore produce noisy results if the average intensity is higher than the predefined threshold, or it simply removes features we'd like to preserve if the average intensity is below it.



```
# Converting to greyscale
gray = cv2.cvtColor(result, cv2.COLOR_BGR2GRAY)
# applying Gaussian blur
blur = cv2.GaussianBlur(gray, (5, 5), 0)
# experimenting with thresholding functions
global_thresh = cv2.threshold(blur, 200, 255, cv2.THRESH_BINARY_INV)[1]
mean_thresh = cv2.adaptiveThreshold(blur, 255, cv2.ADAPTIVE_THRESH_MEAN_C, cv2.THRESH_BINARY_INV, 5, 8)
otsu_thresh = cv2.threshold(blur, 0, 255, cv2.THRESH_OTSU | cv2.THRESH_BINARY_INV)[1]
gaussian_thresh = cv2.adaptiveThreshold(blur, 255, cv2.ADAPTIVE_THRESH_GAUSSIAN_C, cv2.THRESH_BINARY_INV, 5, 8)

cv2.imshow("otsu_thresh", otsu_thresh)
cv2.imshow("global_thresh", global_thresh)
cv2.imshow("mean_thresh", mean_thresh)
cv2.imshow("gaussian_thresh", gaussian_thresh)
cv2.waitKey(0)
```

▲ *Code used to compare the different thresholding methods.*

We can improve this by passing Otsu's algorithm into the threshold function using cv2.THRESH_OTSU through the third parameter along with the output format. Otsu's algorithm computes a threshold value based on the maximum inter-class variance of the two maxima in a histogram and is ideal for bimodal images where there are 2 distinct groups of colour, hence why we converted the image to greyscale beforehand.



▲ *A visualization of Otsu's algorithm where the black line is the threshold, and the orange line the inter-class variance*

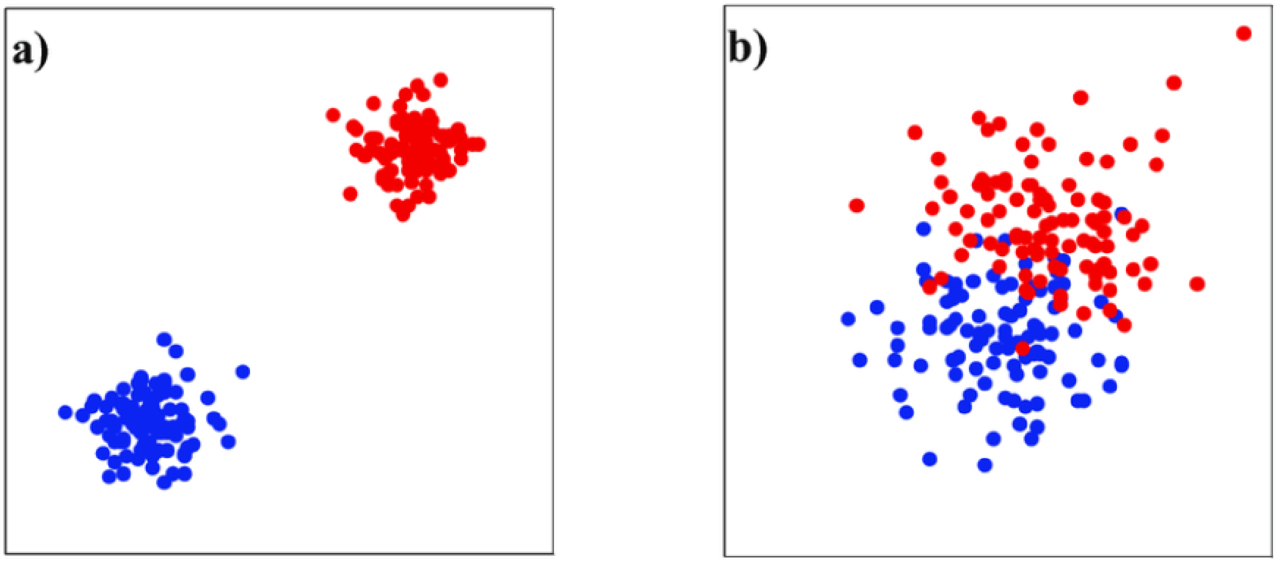
As seen from the image histogram above, a "peak" suggests a high frequency of a pixel value on the x-axis, which we can see there are two distinct groups in the image represented. Otsu's algorithm finds the optimal threshold value by finding the threshold which gives the minimum variance within the two classes separated by the threshold, and hence the maximum variance between the two classes. The mathematical representation can be seen by

$$\sigma_w^2(t) = \omega_0(t)\sigma_0^2(t) + \omega_1(t)\sigma_1^2(t)$$

▲ *Variance within the classes (intra-class variance)*

$$\begin{aligned}\sigma_b^2(t) = \sigma^2 - \sigma_w^2(t) &= \omega_0(t)(\mu_0 - \mu_T)^2 + \omega_1(t)(\mu_1 - \mu_T)^2 \\ &= \omega_0(t)\omega_1(t)[\mu_0(t) - \mu_1(t)]^2\end{aligned}$$

▲ *Variance between the classes (inter-class variance)*



▲ *A visual representation of variance between classes. A) shows a case of high inter-class variance and low intra-class variance, whereas b) has low inter-class variance but high intra-class variance.*

Where sigma squared is the respective variance using threshold t, defined as a weighted sum of the variances in each class. The weights are determined by the sum of the probability of each intensity in the class (t being the threshold intensity, L being the maximum intensity of the image, 255), and the mean used to determine the variance is defined as the expected value of the class divided by the weight of respective class.

$$\omega_0(t) = \sum_{i=0}^{t-1} p(i)$$

$$\omega_1(t) = \sum_{i=t}^{L-1} p(i)$$

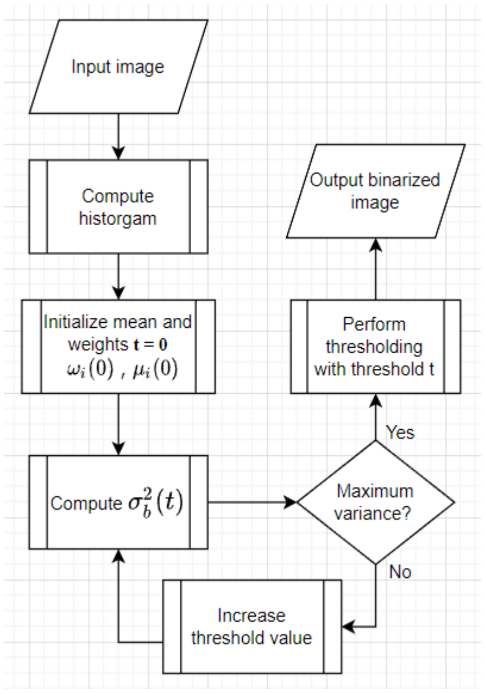
▲ *Weights of each respective class*

$$\mu_0(t) = \frac{\sum_{i=0}^{t-1} ip(i)}{\omega_0(t)}$$

$$\mu_1(t) = \frac{\sum_{i=t}^{L-1} ip(i)}{\omega_1(t)}$$

▲ *Mean of each respective class*

With all the components in place, the algorithm searches through the whole intensity range of the image to find the optimal threshold value iteratively.

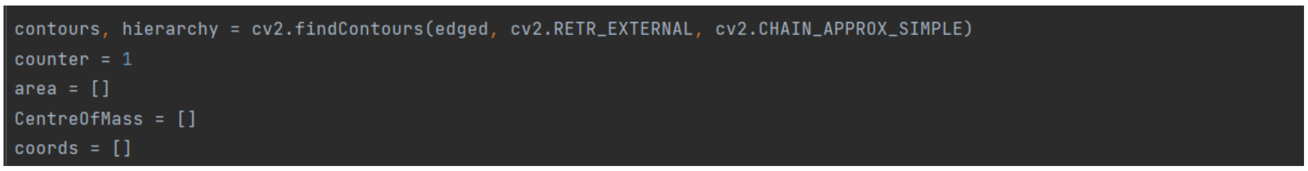


▲ *Flowchart of Otsu's algorithm*

Other thresholding methods experimented are cv2.adaptiveThresholding, which instead of applying a threshold globally, computes a threshold for a specific region and applies it to the region using a kernel. The methodology of computing the threshold can be determined using cv2.ADAPTIVE_THRESH_MEAN_C and cv2.ADAPTIVE_THRESH_GAUSSIAN_C, which computes the threshold value of each kernel using the arithmetic mean and gaussian mean respectively, with a constant c added to the value computed (bias). This is ideal for images which have different lighting conditions in different regions, however as with global thresholding the size of the kernel and bias needs to be specific for the image, and therefore produces unclean results as seen above.

Finding Contours and Drawing Bounding Boxes

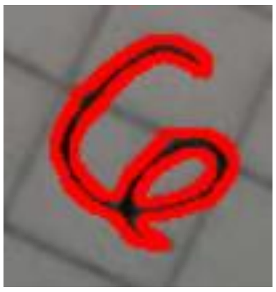
Having binarized the image, we have successfully segmented out the wanted symbols from our original image. However, for symbols such as ÷ and = where the symbol consists of more than one component, we need to be able to combine their components into one box for them to be fed into the CNN together.



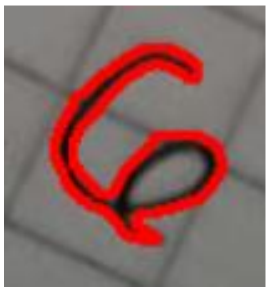
```
contours, hierarchy = cv2.findContours(edged, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
counter = 1
area = []
CentreOfMass = []
coords = []
```

▲ *Implementing cv2.findContours. I have also defined arrays for area and coordinates of contours to be used for later filtering.*

cv2.findContours provides a solution to retrieve the contours of the objects within the image, returning coordinates which trace around the desired objects marked by their higher intensity values (white), and the hierarchy of the contours. Contour hierarchy is a representation of the parent-child relationship between contours, and by defining the contour retrieval mode in the second parameter we can effectively remove layers of contours we don't want. cv2.RETR_EXTERNAL retrieves only the outermost edge of each object (where hierarchy is 0) and ignores all child contours (contours which are inside of another contour). An example can be seen from the 6 using a different input, the inner tracing of the number is ignored when we use cv2.RETR_EXTERNAL.

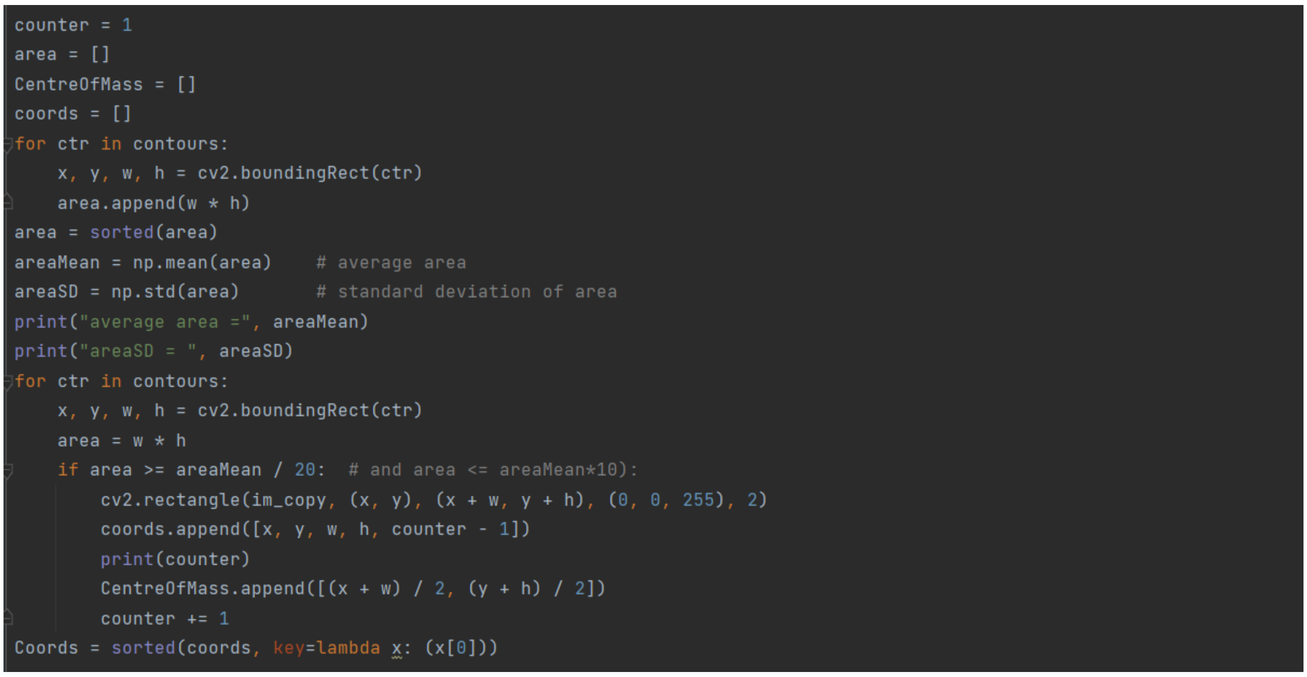


▲ *Using RETR_TREE to return all contours*



▲ *Returning only the outer-most contours*

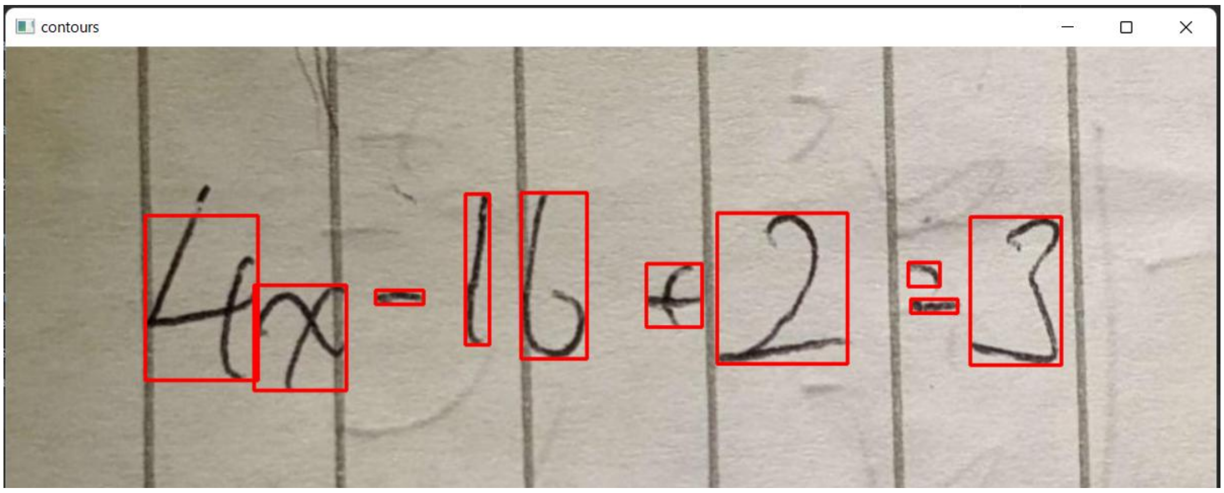
After retrieving the wanted contours, we then use cv2.boundingRect to calculate the outermost points of the contour and store them as coordinates to be drawn out with bounding boxes. However, as seen from previous stages we cannot always completely remove noises from the image, therefore I have calculated the average area of bounding boxes to use as a filter to remove the contours around the noise. The resulting contours will be plotted as a rectangle onto the original image, where it will then be cropped and fed into the CNN in the next process.



```
counter = 1
area = []
CentreOfMass = []
coords = []
for ctr in contours:
    x, y, w, h = cv2.boundingRect(ctr)
    area.append(w * h)
area = sorted(area)
areaMean = np.mean(area)    # average area
areaSD = np.std(area)       # standard deviation of area
print("average area =", areaMean)
print("areaSD = ", areaSD)
for ctr in contours:
    x, y, w, h = cv2.boundingRect(ctr)
    area = w * h
    if area >= areaMean / 20:  # and area <= areaMean*10):
        cv2.rectangle(im_copy, (x, y), (x + w, y + h), (0, 0, 255), 2)
        coords.append([x, y, w, h, counter - 1])
        print(counter)
        CentreOfMass.append([(x + w) / 2, (y + h) / 2])
        counter += 1
Coords = sorted(coords, key=lambda x: (x[0]))
```

▲ *Code to prototype the bounding box tracing process*

At the moment of prototyping, the size which separates the noise contours from the wanted contours is defined through trial and error, however, I intend to improve this by using the standard deviation to find outliers in the contours and effectively remove them.

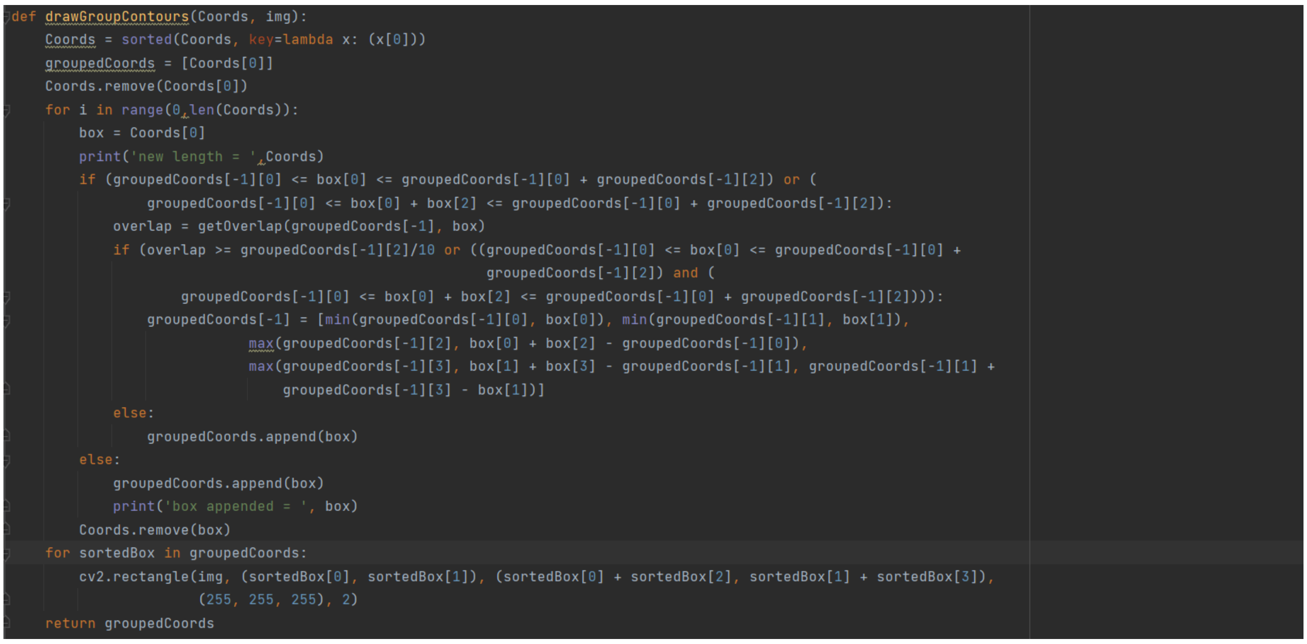


▲ *Resulting image with bounding boxes*

Group Contour Boxes

As one can observe from the bounding boxes above, symbols which contain multiple components such as the equal sign or division symbol will be separated into different bounding boxes, which is unideal for the input for the neural network as it will be unable to classify, say, an equal sign from a subtraction symbol.

To overcome this issue, I created a function which loops through the boxes within the 'Coords' list and append them to a new list 'groupedCoords' if two or more boxes overlap each other vertically.

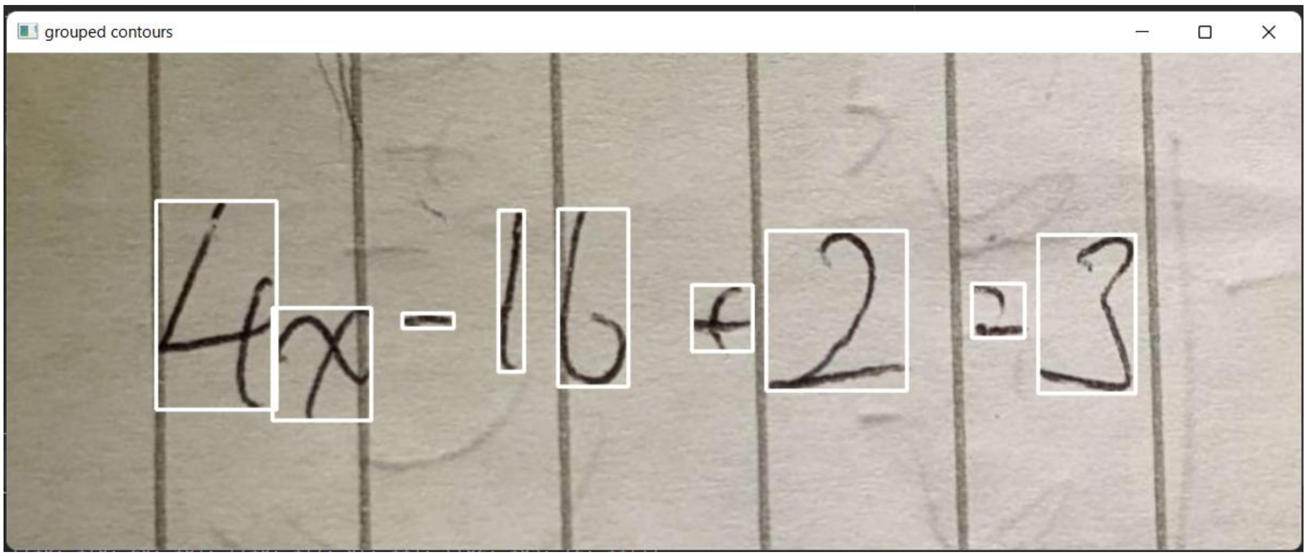


```python
def drawGroupContours(Coords, img):
    Coords = sorted(Coords, key=lambda x: (x[0]))
    groupedCoords = [Coords[0]]
    Coords.remove(Coords[0])
    for i in range(0,len(Coords)):
        box = Coords[0]
        print('new length = ',Coords)
        if (groupedCoords[-1][0] <= box[0] <= groupedCoords[-1][0] + groupedCoords[-1][2]) or (
                groupedCoords[-1][0] <= box[0] + box[2] <= groupedCoords[-1][0] + groupedCoords[-1][2]):
            overlap = getOverlap(groupedCoords[-1], box)
            if (overlap >= groupedCoords[-1][2]/10 or ((groupedCoords[-1][0] <= box[0] <= groupedCoords[-1][0] +
                                                      groupedCoords[-1][2]) and (
                    groupedCoords[-1][0] <= box[0] + box[2] <= groupedCoords[-1][0] + groupedCoords[-1][2]))):
                groupedCoords[-1] = [min(groupedCoords[-1][0], box[0]), min(groupedCoords[-1][1], box[1]),
                                      max(groupedCoords[-1][2], box[0] + box[2] - groupedCoords[-1][0]),
                                      max(groupedCoords[-1][3], box[1] + box[3] - groupedCoords[-1][1], groupedCoords[-1][1] +
                                          groupedCoords[-1][3] - box[1]))]
            else:
                groupedCoords.append(box)
        else:
            groupedCoords.append(box)
            print('box appended = ', box)
        Coords.remove(box)
    for sortedBox in groupedCoords:
        cv2.rectangle(img, (sortedBox[0], sortedBox[1]), (sortedBox[0] + sortedBox[2], sortedBox[1] + sortedBox[3]),
                      (255, 255, 255), 2)
    return groupedCoords
```

▲ *Groups boxes by rewriting the reference (as stored in the groupedCoords list) using min and max functions to create a bounding box which bounds both the reference box and target box in one larger bounding box.*

The grouped boxes can then be plotted onto the image provided as a parameter 'img', and can be outputted later when called to visualize the grouped boxes.

I have also introduced a threshold of overlap between the boxes to remove boxes which have a small overlap, to prevent numbers close together being grouped together.



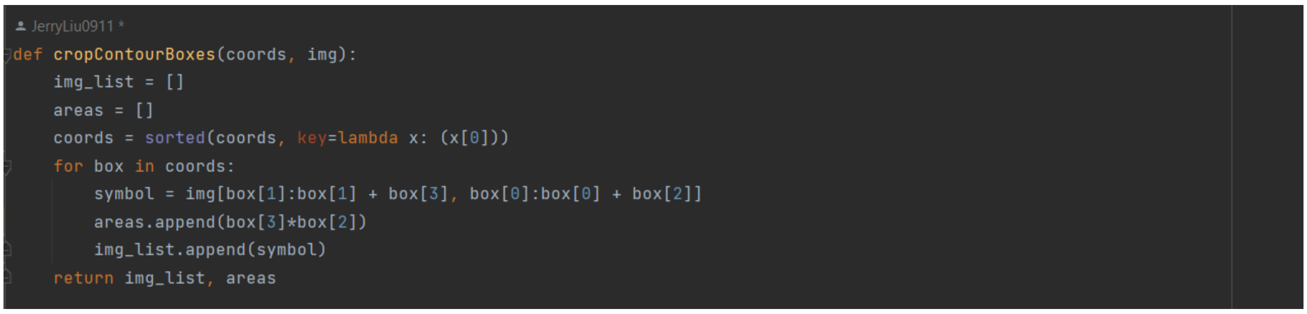
▲ *Grouped boxes. The threshold prevents the '4' and the 'x' in this example to be grouped together.*

My solution to this although less sophisticated, for the simple symbols and linear equations that I will be analyzing this should be sufficient for most test cases.

Cropping and Resizing for input

As previously mentioned, the CNN requires the input data to be uniform, as each neuron in the input layer corresponds to a pixel in the image the input images must be in the same shape as the input layer of the CNN.

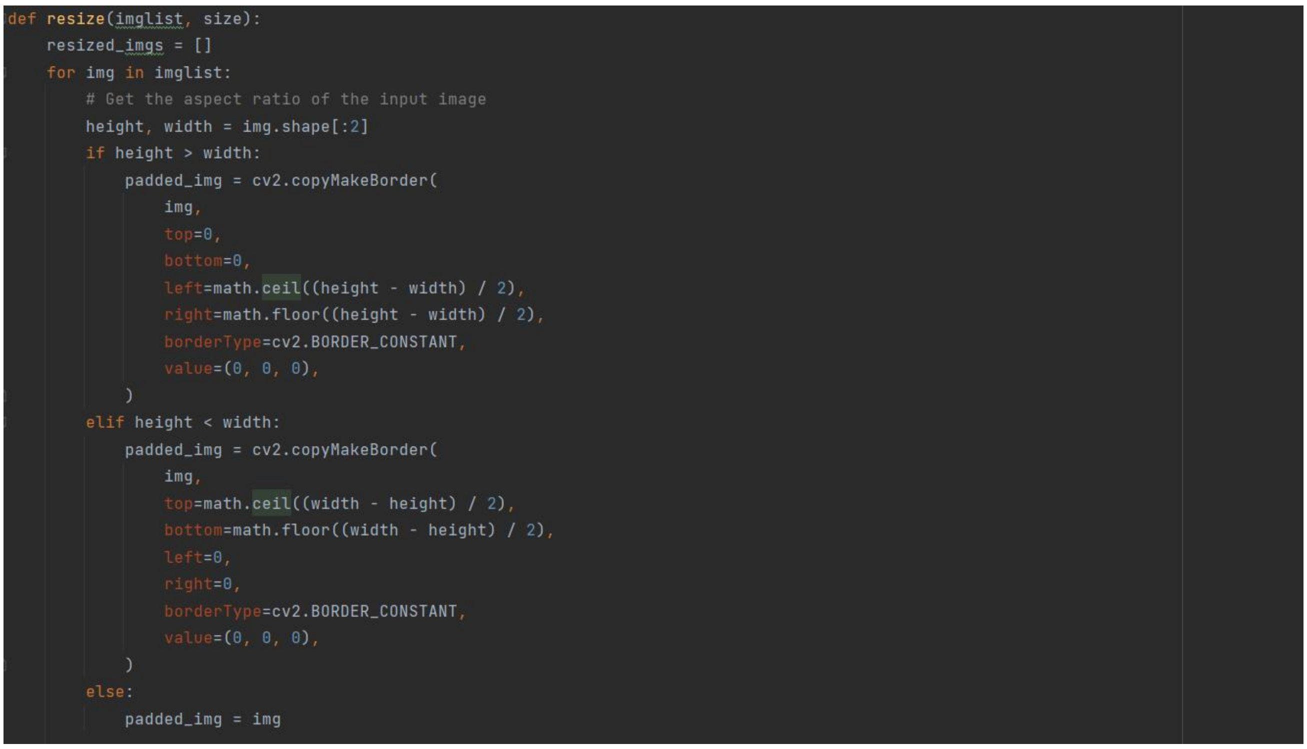
To crop the image into its individual symbols, the coordinates of the grouped bounding boxes are mapped onto the binarized image, and a list of the cropped symbols is outputted. Binarizing the image simplifies and highlights the key features of the image allowing for the neural network to focus on the most relevant features for recognition, as well as decreasing the size of each image, reducing training time and improving accuracy.



```
JerryLiu0911 *
def cropContourBoxes(coords, img):
    img_list = []
    areas = []
    coords = sorted(coords, key=lambda x: (x[0]))
    for box in coords:
        symbol = img[box[1]:box[1] + box[3], box[0]:box[0] + box[2]]
        areas.append(box[3]*box[2])
        img_list.append(symbol)
    return img_list, areas
```

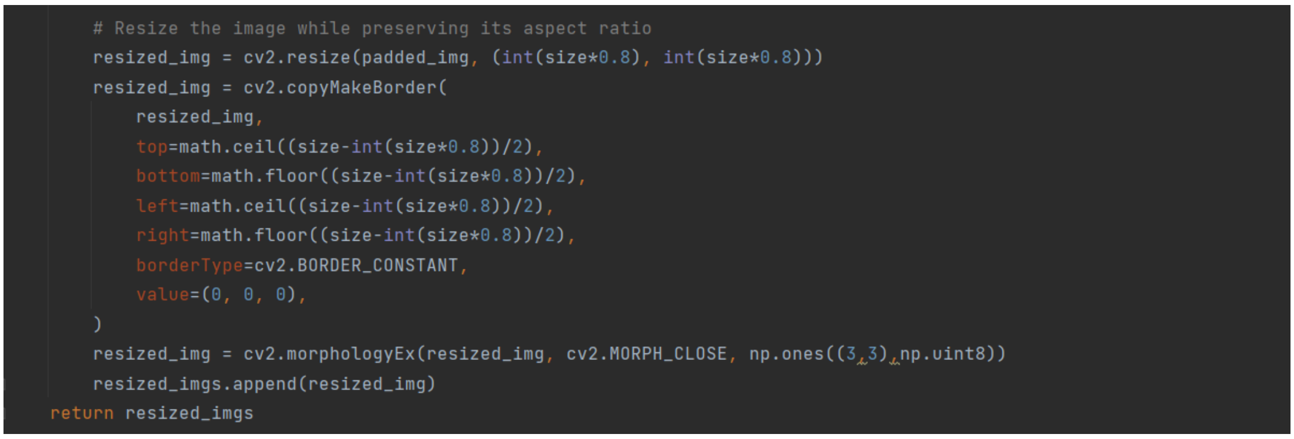
▲ *cropContourBoxes function*

The cropped images are resized to fit the input size according to the input layer of the neural network, by padding the original image into a square, and resizing it to the desired size.



```
def resize(imglist, size):
    resized_imgs = []
    for img in imglist:
        # Get the aspect ratio of the input image
        height, width = img.shape[:2]
        if height > width:
            padded_img = cv2.copyMakeBorder(
                img,
                top=0,
                bottom=0,
                left=math.ceil((height - width) / 2),
                right=math.floor((height - width) / 2),
                borderType=cv2.BORDER_CONSTANT,
                value=(0, 0, 0),
            )
        elif height < width:
            padded_img = cv2.copyMakeBorder(
                img,
                top=math.ceil((width - height) / 2),
                bottom=math.floor((width - height) / 2),
                left=0,
                right=0,
                borderType=cv2.BORDER_CONSTANT,
                value=(0, 0, 0),
            )
        else:
            padded_img = img
```

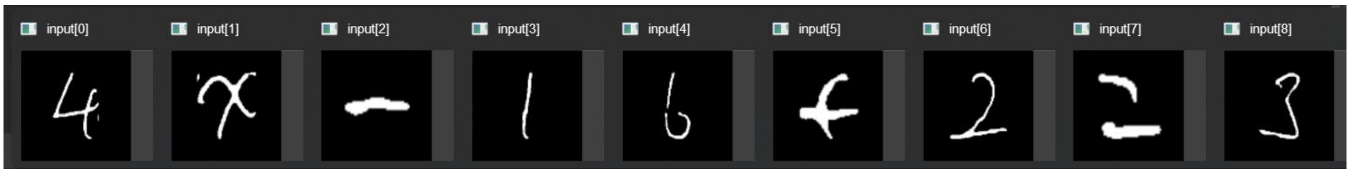
▲ *Rectangular images are padded around the shorter side to create a square shape.*



```
        # Resize the image while preserving its aspect ratio
        resized_img = cv2.resize(padded_img, (int(size*0.8), int(size*0.8)))
        resized_img = cv2.copyMakeBorder(
            resized_img,
            top=math.ceil((size-int(size*0.8))/2),
            bottom=math.floor((size-int(size*0.8))/2),
            left=math.ceil((size-int(size*0.8))/2),
            right=math.floor((size-int(size*0.8))/2),
            borderType=cv2.BORDER_CONSTANT,
            value=(0, 0, 0),
        )
        resized_img = cv2.morphologyEx(resized_img, cv2.MORPH_CLOSE, np.ones((3,3),np.uint8))
        resized_imgs.append(resized_img)
    return resized_imgs
```

▲ *The adjusted images are resized to 80% of the desired size and then padded around the edges into the desired size. Padding around the edges preserves the features on the edges of the image and makes the input data similar to the training data used for the neural network to increase accuracy.*

A morphological closing transformation is applied to the image to reduce the effects of disconnected lines in the image from inconsistent ink flow in the pen or disconnected due to the previous transformations, especially to retain continuous lines after the interpolation from resizing the images.



▲ *The cropped images resized to 50,50.*

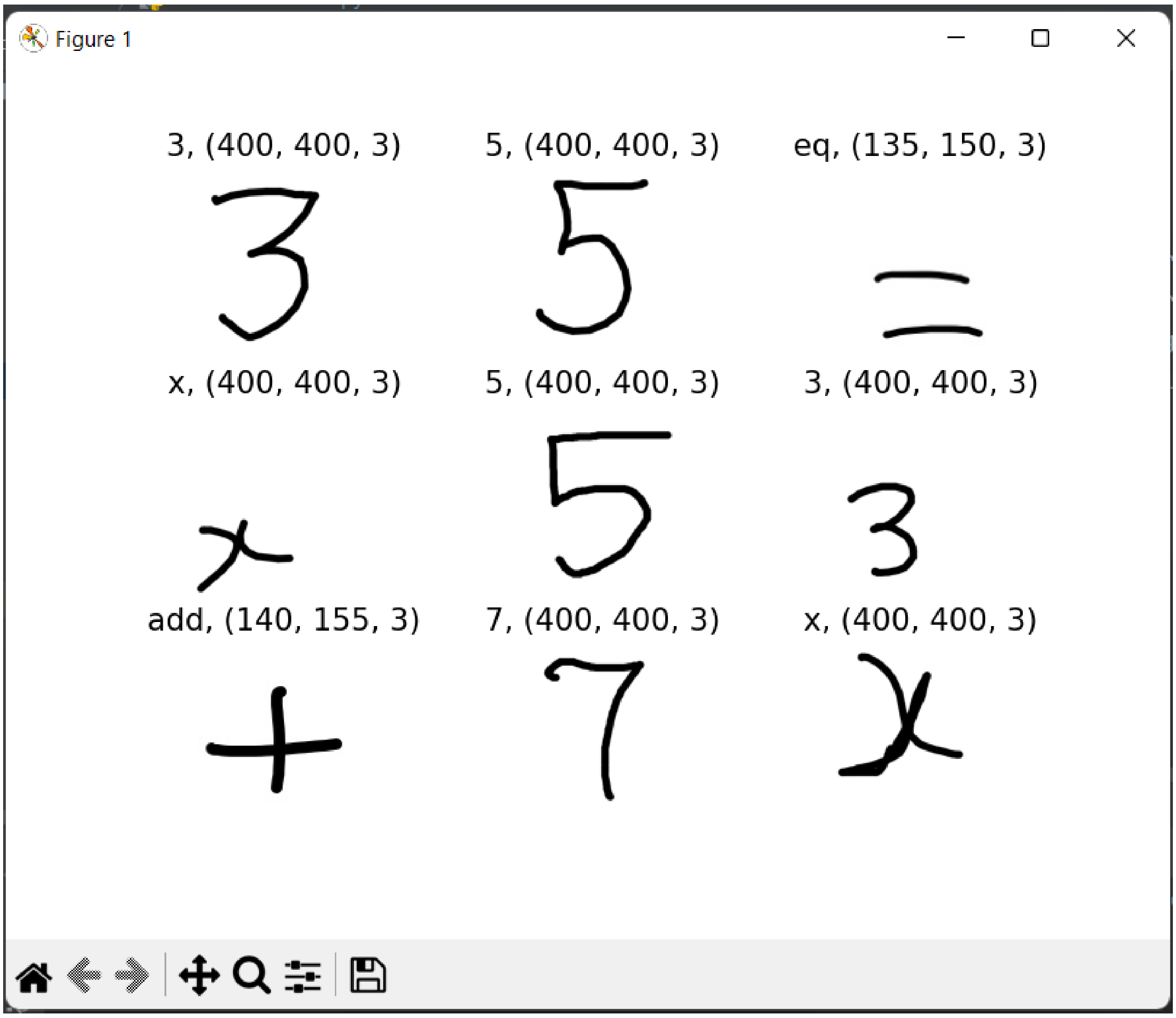
The cropped images will be stored as a NumPy array and later on to be inputted into the neural network.

Convolutional Neural Network

Dataset Preprocessing

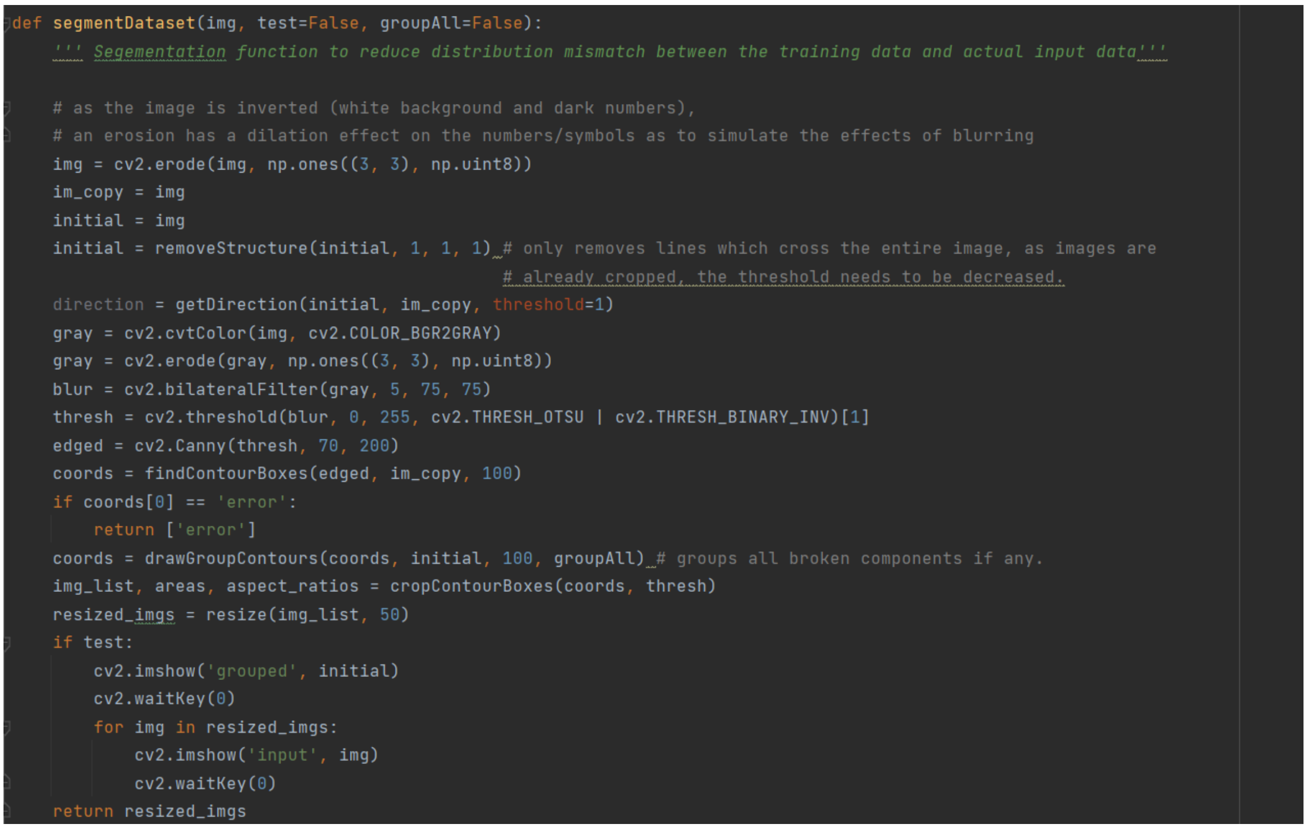
Before building a neural network, a suitable dataset needs to be constructed to be used as training data to test and train the weights and biases of the neural network. For this task in particular, I found a dataset on Kaggle (sited in the References section) which consisted of 19 classes with an average of 500 images per class, summing up to approximately 10,000 images of handwritten digits and mathematical operators, notably the addition, subtraction, multiplication and division symbols, as well as the equal sign. However for the simplicity of development, I chose to use 16 out of the 19 classes provided (0-9, the five operators and x).

From observing the dataset, we can see that the data is clearly different from what our segmented images look like, being uncentered and having different colour channels:



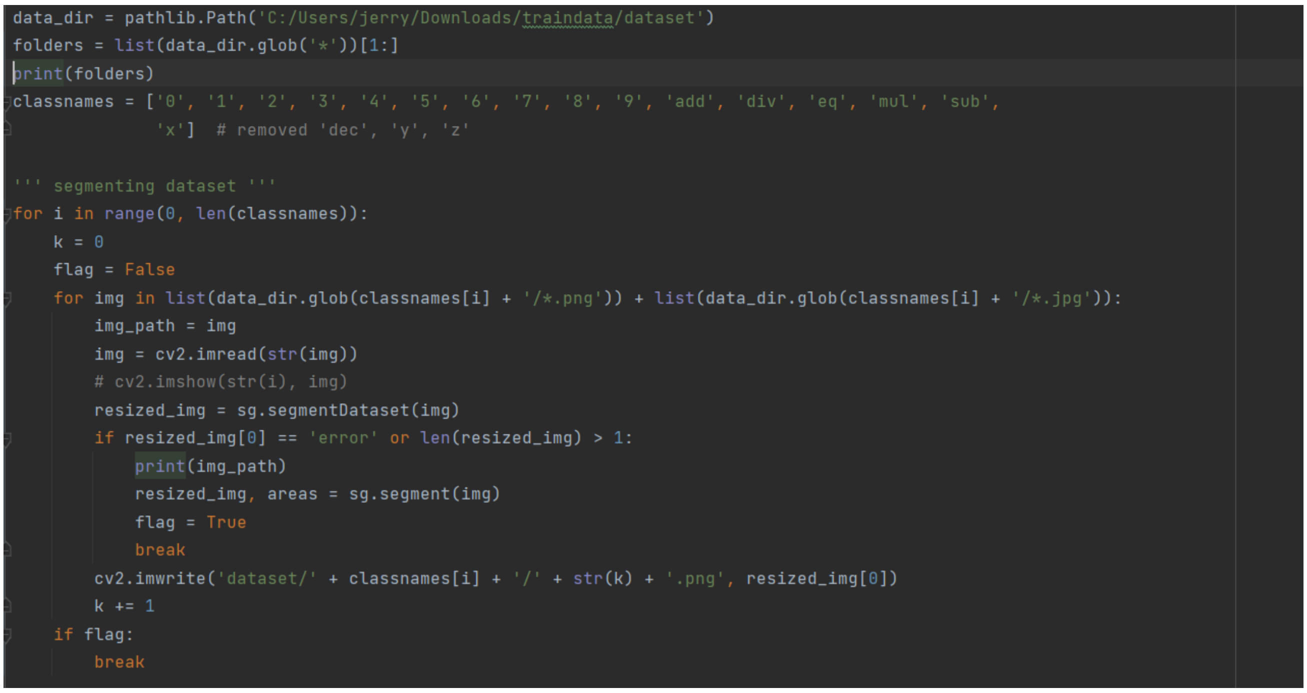
▲ *Randomly selected 9 images from random classes. As observed, the sizes of each are different.*

Therefore, to prevent dataset mismatching, each image within the dataset is transformed in a similar fashion to the actual input image through the segmentDataset function:



```
def segmentDataset(img, test=False, groupAll=False):
    ''' Segementation function to reduce distribution mismatch between the training data and actual input data'''

    # as the image is inverted (white background and dark numbers),
    # an erosion has a dilation effect on the numbers/symbols as to simulate the effects of blurring
    img = cv2.erode(img, np.ones((3, 3), np.uint8))
    im_copy = img
    initial = img
    initial = removeStructure(initial, 1, 1, 1) # only removes lines which cross the entire image, as images are
                                                # already cropped, the threshold needs to be decreased.
    direction = getDirection(initial, im_copy, threshold=1)
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    gray = cv2.erode(gray, np.ones((3, 3), np.uint8))
    blur = cv2.bilateralFilter(gray, 5, 75, 75)
    thresh = cv2.threshold(blur, 0, 255, cv2.THRESH_OTSU | cv2.THRESH_BINARY_INV)[1]
    edged = cv2.Canny(thresh, 70, 200)
    coords = findContourBoxes(edged, im_copy, 100)
    if coords[0] == 'error':
        return ['error']
    coords = drawGroupContours(coords, initial, 100, groupAll) # groups all broken components if any.
    img_list, areas, aspect_ratios = cropContourBoxes(coords, thresh)
    resized_imgs = resize(img_list, 50)
    if test:
        cv2.imshow('grouped', initial)
        cv2.waitKey(0)
        for img in resized_imgs:
            cv2.imshow('input', img)
            cv2.waitKey(0)
    return resized_imgs
```

▲ *segmentDataset function from ImageSegmentationModule*



```
data_dir = pathlib.Path('C:/Users/jerry/Downloads/traindata/dataset')
folders = list(data_dir.glob('*'))[1:]
print(folders)
classnames = ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'add', 'div', 'eq', 'mul', 'sub',
              'x']  # removed 'dec', 'y', 'z'

''' segmenting dataset '''
for i in range(0, len(classnames)):
    k = 0
    flag = False
    for img in list(data_dir.glob(classnames[i] + '/*.png')) + list(data_dir.glob(classnames[i] + '/*.jpg')):
        img_path = img
        img = cv2.imread(str(img))
        # cv2.imshow(str(i), img)
        resized_img = sg.segmentDataset(img)
        if resized_img[0] == 'error' or len(resized_img) > 1:
            print(img_path)
            resized_img, areas = sg.segment(img)
            flag = True
            break
        cv2.imwrite('dataset/' + classnames[i] + '/' + str(k) + '.png', resized_img[0])
        k += 1
    if flag:
        break
```

▲ *Each image within the dataset is processed through the segmentation function, and stored as a png file in the project dataset.*

Neural Network Architecture

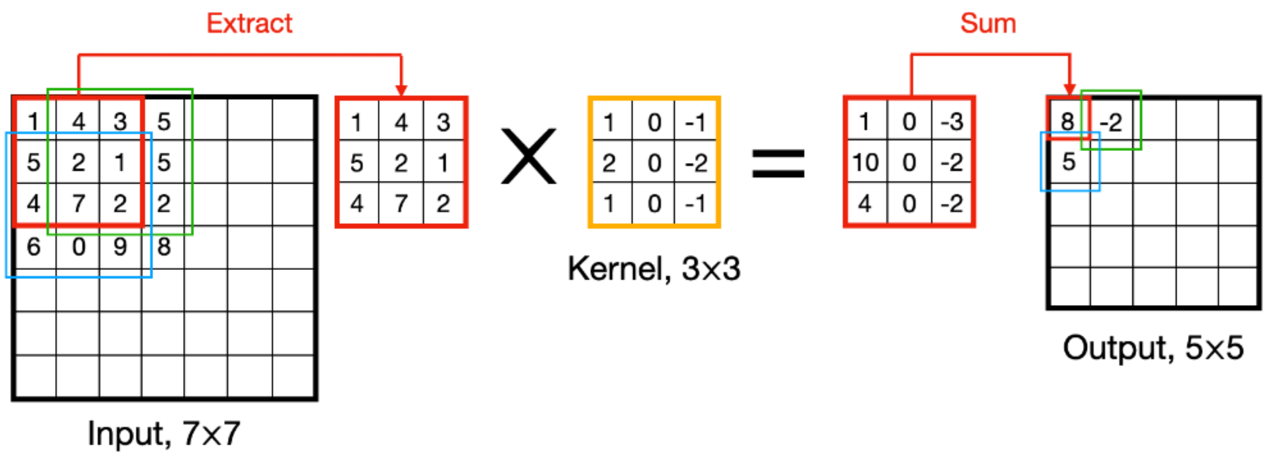
The neural network architecture I have chosen to implement a modified version of the LeNet architecture, one of the first convolutional neural networks developed for handwritten digit recognition, consisting of 7 layers, 2 sets of a convolutional layer followed by a pooling payer, and 3 fully connected neural networks responsible for the classification of the images according to their features.

Convolutional Layer	Input Shape: 50x50, Activation: ReLu, Kernel Shape: 5x5, Features: 6
Max-pooling layer	Shape: 2x2
Convolutional Layer	Activation: ReLu, Kernel Shape: 5x5, Features: 16
Max-pooling Layer	Shape: 2x2
Dropout Layer	Dropout rate: 0.5
Fully-connected Layer	Nodes: 200, Activation: ReLu
Dropout Layer	Dropout rate: 0.3
Fully-connected Layer	Nodes: 100, Activation: ReLu
Dropout Layer	Dropout rate: 0.2
Fully-connected Layer	Nodes: 16, Activation: Softmax

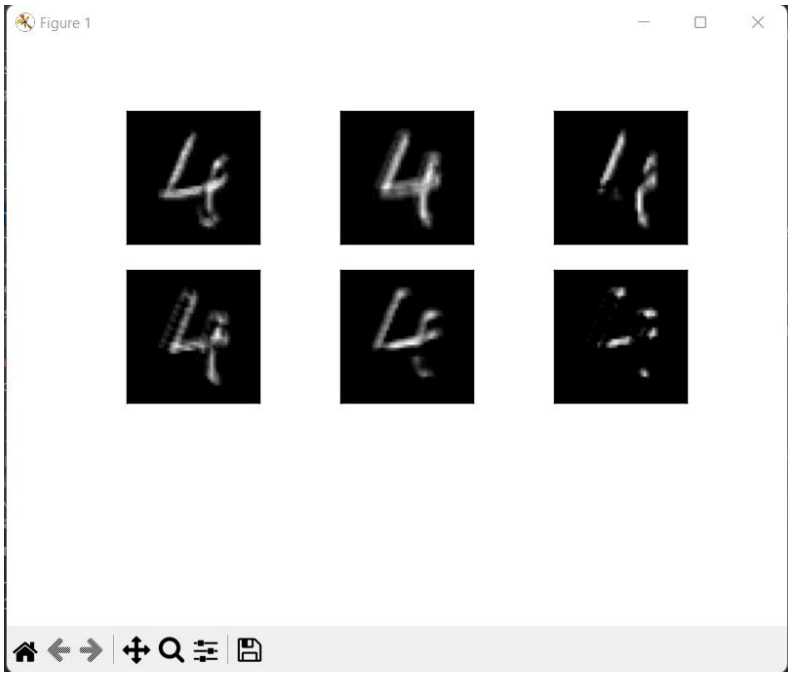
Convolutional Layer

As previously mentioned, a convolution can be thought of as passing a set of filters which highlight specific features of an image. Depending on the contents of the filters, different features may be highlighted and allows for the neural network to pick up on important details for recognition in consequent layers. The contents of the filters are known as “weights” and “biases”, which can be thought of as a linear transformation on the images to perform extraction on desired features, which can be visualized through a feature map.

Tensorflow’s Keras library allows for ease of implementation of a convolutional layer by using its built in keras.layers.Conv2D function, which allows the user to implement a convolutional layer with custom activation functions, number of filters as well as filter sizes.

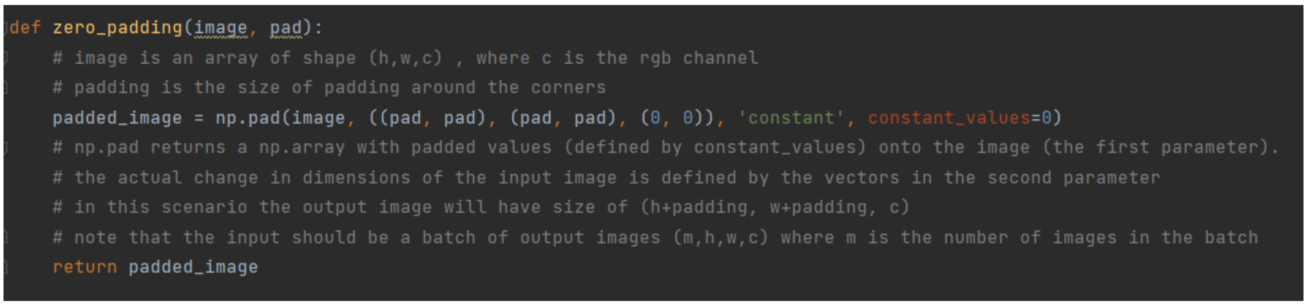


▲ *A visualization of the convoluting process with a 3*3 matrix of stride 1.*



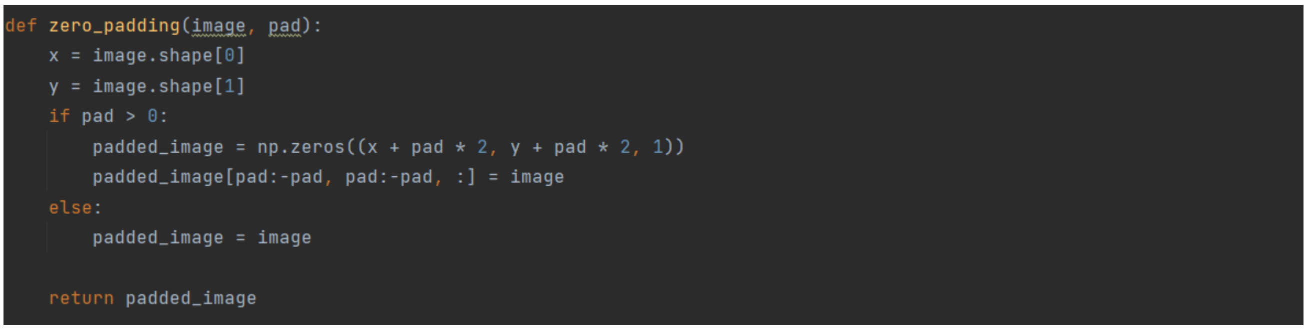
Padding Function

A padding function creates a wrapping of zeroes around the input image to preserve or increase image size, allowing more space for the kernel to cover the image, and therefore increasing the accuracy of the feature analysis. Here I have simply used np. pad to create an array of zeroes around the image, which could also be done by defining an array of zeroes of size (original array + padding) and substituting the values within the original pixel values of the image to create the same effect. To demonstrate understanding of these concepts, I have implemented the padding functions in Python as seen:



```python
def zero_padding(image, pad):
    # image is an array of shape (h,w,c) , where c is the rgb channel
    # padding is the size of padding around the corners
    padded_image = np.pad(image, ((pad, pad), (pad, pad), (0, 0)), 'constant', constant_values=0)
    # np.pad returns a np.array with padded values (defined by constant_values) onto the image (the first parameter).
    # the actual change in dimensions of the input image is defined by the vectors in the second parameter
    # in this scenario the output image will have size of (h+padding, w+padding, c)
    # note that the input should be a batch of output images (m,h,w,c) where m is the number of images in the batch
    return padded_image
```

▲ *Using np.pad*

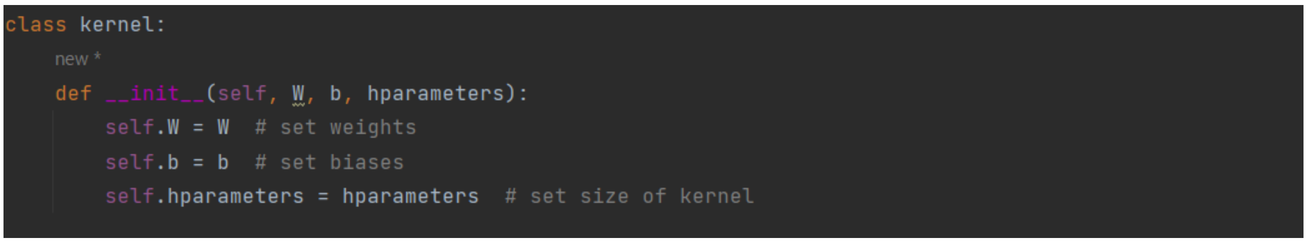


```python
def zero_padding(image, pad):
    x = image.shape[0]
    y = image.shape[1]
    if pad > 0:
        padded_image = np.zeros((x + pad * 2, y + pad * 2, 1))
        padded_image[pad:-pad, pad:-pad, :] = image
    else:
        padded_image = image

    return padded_image
```

▲ *Alternative method*

Initialising Kernels for Back Propagations

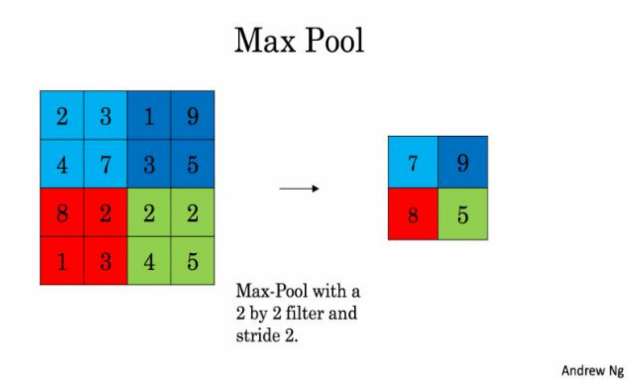
A convolution can be expressed as the dot product of an input vector to a vector of weights and biases contained within the kernel. An example of an implementation of the kernel is shown, defining an object 'kernel' allows us to access and modify specific kernels for backpropagation, as well as encapsulating the data to prevent unwanted modifications.

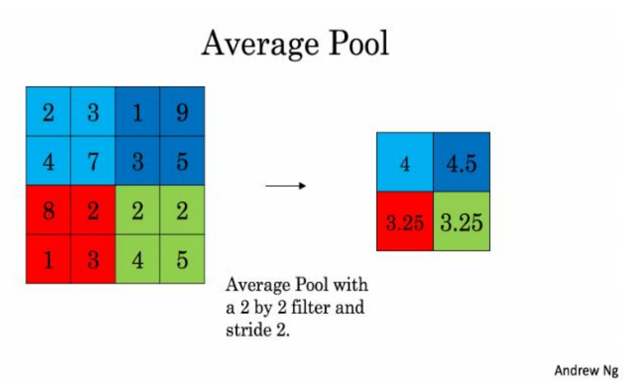


```python
class kernel:
    new *
    def __init__(self, W, b, hparameters):
        self.W = W  # set weights
        self.b = b  # set biases
        self.hparameters = hparameters  # set size of kernel
```

Pooling Layer

The pooling layer condenses the size of the input to reduce computation after we've extracted the desired features from a convolutional layer. This can be done through two forms of pooling :





Activation Functions

A singular neuron with weights and biases is nothing but a simple linear regression model. Activation functions introduce non-linearity into the model to allow it to capture more complex features. An example of this is the popular Sigmoid or Logarithmic function, defined by :

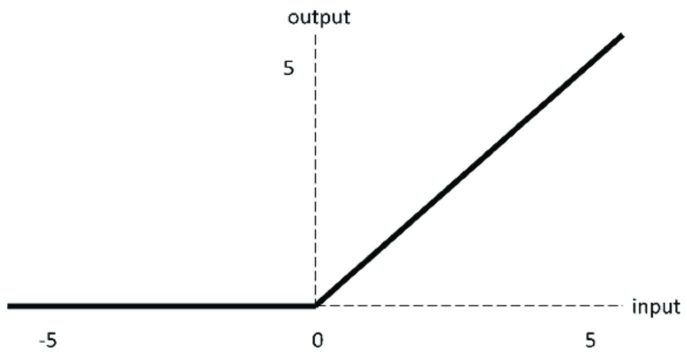
$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

Where z is the input vector. The result is a smooth curve which tends to 1 for large values of z, and 0 for large negative values, with a 0 corresponding to 0.5. This can be interpreted as the "confidence" of how relevant the output of our linear model is to the truth labels. To take this analogy further, we look at the output layer of our model. We would like to translate the output from the weights and biases into representable data, for example, if we were to output a percentage on how close the input image is to a number "5", we may choose to have an output which ranges from 0-1 to represent the likelihood of it being a "5". This process of translating the weighted sum of pixels into a vector of probabilities which sum to 1 (suitable for mutually exclusive classes) can be done through a SoftMax function, which is defined by

$$\sigma(\mathbf{z})_i = \frac{e^{\beta z_i}}{\sum_{j=1}^{K} e^{\beta z_j}}$$

The Softmax function exponentiates each element of the input vector and normalizes it by dividing by the sum of all exponentiated values, ensuring the outputs sum to 1.

For other layers however, we are not restricted to representing the output of the layer as probabilities. To reduce computation, we may choose another non-linear function such as the Rectified Linear Unit or ReLU, which is a simple linear function which crosses through 0, restricted to only positive inputs.



We therefore choose the ReLU as the activation function for most of the hidden layers, and the Softmax function for the activation function of the output layer.

References

- http://neuralnetworksanddeeplearning.com
- https://github.com/stfc-sciml/sciml-workshop
-https://towardsdatascience.com/a-guide-to-an-efficient-way-to-build-neural-network-architectures-part-ii-hyper-parameter-42efca01e5d7
https://en.wikipedia.org/wiki/Gaussian_blur
- https://en.wikipedia.org/wiki/Otsu%27s_method

- http://www.labbookpages.co.uk/software/imgProc/otsuThreshold.html
- https://docs.opencv.org/4.x/d9/d8b/tutorial_py_contours_hierarchy.html
-https://opencv24-python-tutorials.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_morphological_ops/py_morphological_ops.html
https://en.wikipedia.org/wiki/Softmax_function
-https://www.tensorflow.org/api_docs/python/tf
-https://keras.io/api/
https://docs.python.org/3/library/itertools.html
-https://github.com/enggen/Deep-Learning-Coursera/tree/master/Convolutional%20Neural%20Networks
-https://kivy.org/doc/stable/