Chapter 1

1.1 Introduction

Digitization of offline work is on the rise to increase the longevity of documents most of which consist of mathematical expressions because of the ubiquitous nature of mathematics. Since mathematics is almost entirely subsumed in its expressions, it is imperative to properly digitize these expressions to maintain the consistency in the digital documents [11]. But recognition of handwritten mathematical expressions is a difficult task and topic of many ongoing and concluded research works [10].

Handwritten mathematical expression recognition is of two types: online and offline. The former form consists of recognizing the characters by their strokes while they are being written on a tablet or smartphone. While the latter form consists of recognizing the characters from an image of a handwritten document. This research paper will be entirely dedicated to the digitization and evaluation of offline handwritten mathematical expressions.

Since mathematics itself is a very wide field, digitizing and evaluating all of the mathematical symbols becomes a very complex and tedious task. Therefore, only a subset of these mathematical symbols is considered in this paper which are digits (0-9), arithmetic operators (+, -, *, ÷), characters (a, b, c, d, u, v, w, x, y, z) and parenthesis. All of these will be referred to as symbols in the rest of this research paper.

The focus in this research has been on segmentation and recognition of multiple arithmetic expressions and linear equations from a single image and then evaluating these successfully recognized expressions. The original contributions and approach followed in this research paper are outlined in the following paragraphs.

Image comprising of single or multiple mathematical expressions containing either entirely arithmetic or linear expressions are considered as input. A new approach of pre-contour filtration is considered in this paper where expressions are tightly cropped to remove any noise that might obfuscate the segmentation of symbols. The segmented symbols are arranged in their original manner using a novel algorithmic technique. These segmented symbols are recognized using a Convolutional Neural Network (CNN), because of its state-of-the-art performance in classification of images [12], which result in a digitized expression. The evaluation of these digitized expressions is performed by ingeniously built string manipulation algorithms.

The segmentation of '=' and '÷' characters results in the detection of separate components instead of a single symbol. This problem is solved by vertically combining the components for the height of the image resulting in a single segmented image of the symbol. Another problem faced is the ambiguity between 'x' and '*' characters because of the similarity in their handwritten versions. This problem is solved by considering the succeeding symbol which should be a digit or an open parenthesis in case of '*' and any other symbol for 'x'. Finally, the recognition of offline handwritten symbols was made easier by using a shallow CNN which was able to understand the complex relationship of strokes constituting a symbol.

1.2 Motivation

A lot of work has been done in the field of Handwritten Mathematical Expression recognition. Some of these have been studied before the implementation of the proposed system in this paper.

The proposed system used in [1, 2] is to normalize the image of HME. The threshold value of 50px was used. Then edge detection followed by morphological transformations are applied and separation of components of the image has been considered. Features like skew, entropy and standard deviation were extracted to improve the accuracy of the neural network. The recognition was done with a backpropagation neural network with adaptive learning. The neural network had 10 input nodes, two hidden layers and one output layer with 10 nodes. The proposed network achieved an accuracy of 99.91% on the training dataset of 5x7 pixels.

In [3] the proposed method was to classify handwritten mathematical symbols. Convolutional Neural Network (CNN) model was used with a 5x5 kernel for convolutional layer and 2x2 kernel for max pooling layer. Sigmoid function for non-linearity was used at every layer of the network. The log-likelihood cost function was used to check the performance. The CROHME 2014 dataset was used for training and images were resized to 32x32 pixels. The accuracy achieved was 87.72%. A crucial point identified in [3] was that some symbols were misclassified by CNN because they had a similar structure with the other symbols, a problem that was also faced in this paper.

In [4], the main objective was symbol detection from images of HME. 3 modified versions SSD model were used along with the original SSD model for the detection and classification of mathematical symbols from HME. There were 52353 Gray images of 32X32 size belonging to 106 classes of symbols. Dataset of HME contained 2256 images of 552 expressions of 300X300 resolution divided into 3 sets. The precision for each class was calculated and class weight for each symbol was calculated. The maximum mAP gain (0.65) was observed in the version of SSD where 1 convolution layer was modified and 2 new layers were added.

In [5], the main focus was to recognize and digitize HME. Convolutional Neural network with input shape of 45X45, 3 convolution and max pooling layers, a fully connected layer and output shape of 83 was used for classification of HMS extracted from HME. Preprocessing of HME images included grayscale conversion, noise reduction using median blur, binarization using adaptive threshold and thinning to make the thickness of foreground 1 pixel. Then segmentation was done using projection profiling algorithm and using connected component labeling.CNN achieved an accuracy of about 87.72% on HMS.

The approach taken in [6] is different from other proposed systems, it mainly focuses on Chinese HME. For symbol segmentation, A decomposition on strokes is operated, then dynamic programming to find the paths corresponding to the best segmentation manner and to reduce the stroke searching complexity is used. For symbol recognition, Spatial geometry and directional element features are classified by a Gaussian Mixture Model learned through the Expectation-Maximization algorithm. For semantic relationship analysis, A ternary tree is utilized to store the ranked symbols through calculating their priorities. The system was tested on a dataset consisting of 30 model expressions with a total of about 15000 symbols. The system performs well at symbol level but recognition of full expression shows 17 % accuracy.

The system proposed in [7] is for recognition and evaluation for single or a group of handwritten quadratic equations. NIST dataset and self-prepared symbols are used for training after preprocessing techniques such as grayscale, binarization and low pass filtering Horizontal compact projection analysis and combined connected component analysis methods are used for segmentation. For the classification of specific characters, CNN is applied. The system was able to fully recognize 39.11% equation correctly in the set of 1000 images whereas character segmentation accuracy was 91.08%.

The methodology proposed in [8] uses a CNN for feature extraction, a bidirectional LSTM for encoding extracted features and an LSTM and an attention model for generating target LaTex.
The dataset used is CROHME with augmentation techniques such as local distortion and global distortion. Recognition neural network consists of 5 convolution layers, 4 max-pooling layers with no dropout layer. The accuracy obtained on CHROME was 35.19%.

1.3 Objective

Writing mathematical expressions in digital form in a difficult task because of the combination of strokes making up a mathematical expression. Therefore, the objective of this project is to digitize and evaluate the handwritten mathematical expressions from images. But since the domain of mathematical expressions is too wide, this project is focused on arithmetic and linear expressions.

This objective is achieved by implementing the following steps:

1. Segmenting the digits, operators and variables from the input image of a mathematical expression. These characters will be sorted and stored in the same order as they appear in the image of the expression.
2. The segmented characters need to be recognized in order to be digitized. This will be achieved using Convolutional Neural Network because of their state-of-the-art performance is the classification of images.
3. The digitized expression needs to be evaluated. Evaluation of the expressions will be done by using an algorithmic process designed in the project.

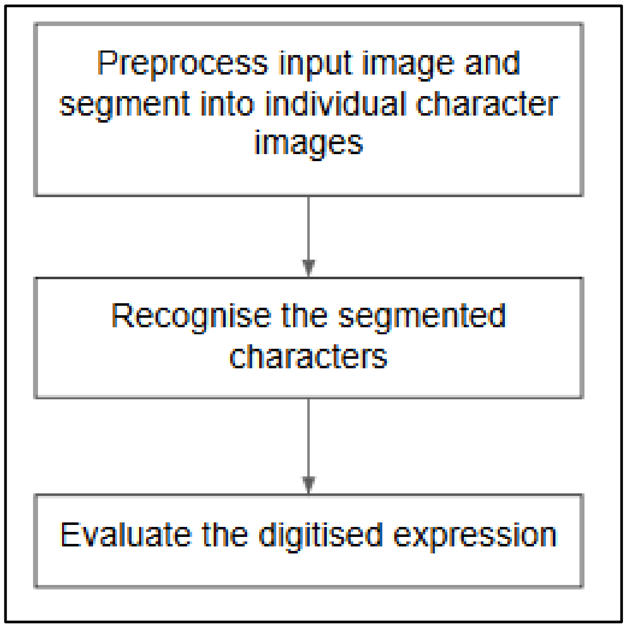

Preprocess input image and segment into individual character images
Recognise the segmented characters
Evaluate the digitised expression

Fig 1.1 Objective

1.4 Summary of the work done

Chapters 2-6 describe the whole project in detail. They elaborate on the processes that have been talked about in brief in the objective of this project. The project can be divided into three separate parts: segmentation, recognition and evaluation.

For segmentation, the foremost step is preprocessing of the image which prepares it for further processing and extraction of features. This involves reducing the number of colour channels of the image and blurring the image to remove any noise that might interfere during feature extraction. Following this step, the characters are extracted from the image using OpenCV tool. A novel technique of pre-contour filtration is used to achieve better results with the extraction of characters from the image. Finally, these extracted characters images are sorted in the order present in the original input image. An innovative technique for sorting of contours has been designed so that even an image containing multiple expressions can be sorted and segregated according to their expressions, making it easier to digitize and evaluate multiple expressions in a single image.

For recognition, Convolutional Neural Network (CNN) is used. The key to an accurate CNN is preparing a good dataset. To improve the images present in the existing EMNIST [9] dataset and balance the imbalanced class of image, data augmentation technique has been used. This data augmentation and preparation of CNN architecture has been implemented using the Keras library with Tensorflow backend. The mathematical expression is finally digitised after recognition of characters achieved by CNN.

Finally, the digitized mathematical expressions are evaluated using the ingeniously designed algorithmic process. First, the digitized expressions are tokenised to extract the digits and operators from the expression and append them into an empty string according to their precedence order. The tokenised expression can now be manipulated based on whether it is an arithmetic expression or system of linear equations. Separate algorithms have been designed to solve arithmetic and linear expressions.

A diagrammatic representation of the steps involved in the development of the project and the steps that will be followed in solving a mathematical expression once an image is input is represented in the figure 1.2.

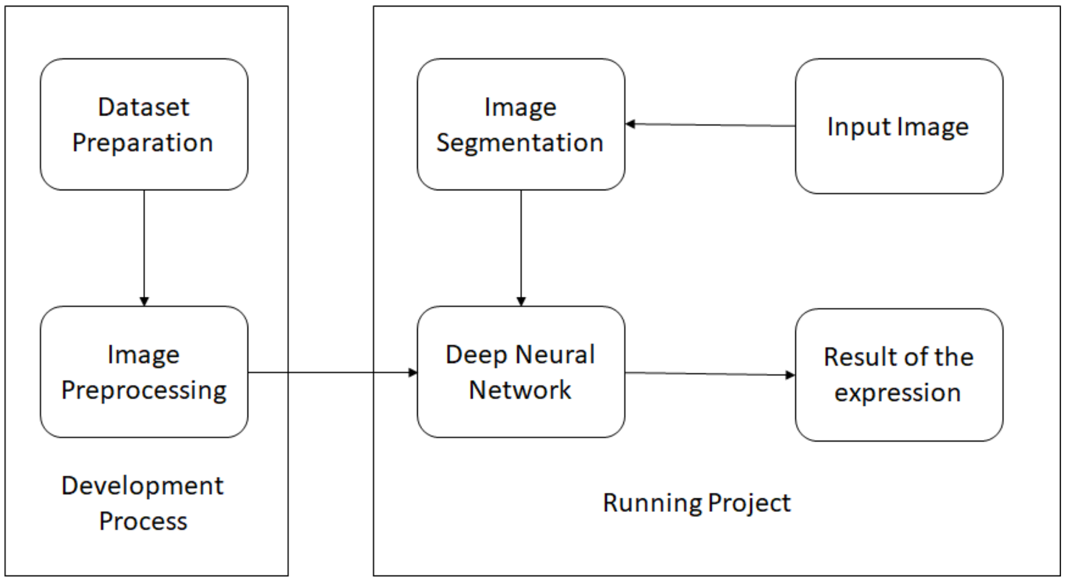


Fig 1.2 Represents the summary of the work in a diagrammatic manner

The experimental results are discussed in chapter 7. The result of the achieved accuracy in recognition of characters using CNN has been presented in a comprehensive manner. The resultant images from different steps in the project have also been presented to comprehend the project in its entirety.

The problems faced in the project has been discussed in chapter 8, why they occur and how they were overcome. Also discussed is the future scope of this project to further extend its capabilities to other domains of mathematical expressions.

Chapter 2

2.1 Introduction

The first part of implementing the project is to preprocess the input image of the mathematical expression. This preprocessing step is important as it will remove any noise from the image and prepare it for further processing and extraction of segmented symbols from the image.

The steps involved in preprocessing of the input image that are employed in this project are: illumination, grayscaling, gaussian blurring and thresholding as shown in **Figure 2.1**.

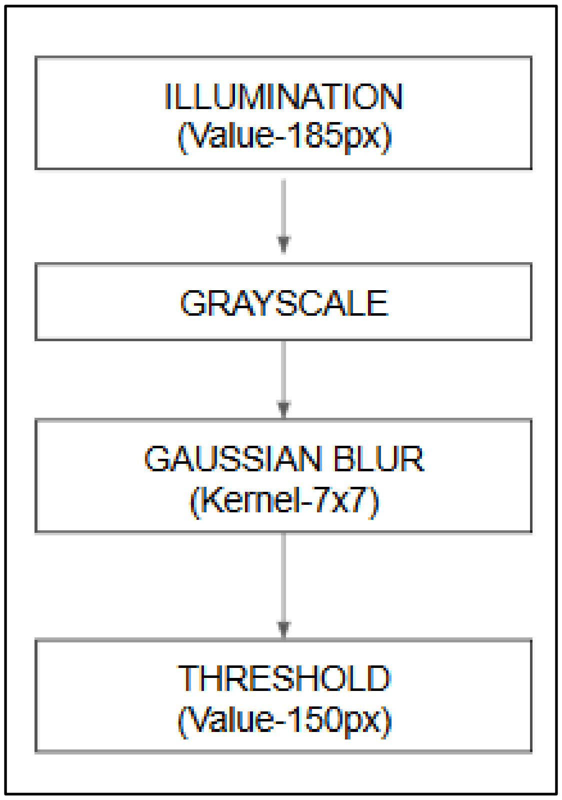


Fig 2.1 Steps involved in the preprocessing of the input image

2.2 Implementation

1. Illumination

A coloured image is made of three colour channels: Red, Green, Blue. Together they are known as the **RGB** colour scheme. RGB describes colour as a tuple of three colour components. Each component can take a value between 0 and 255, where the tuple (0, 0, 0) represents black and (255, 255, 255) represents white.

RGB is one of the five major colour space models, each of which has many offshoots. There are so many colour spaces because different colour spaces are useful for different purposes.

HSV colour scheme is the description of hue, saturation, and value (brightness), which are particularly useful for identifying contrast in images. These colour spaces are frequently used in colour selection tools in software and web design.

The image is read using OpenCV function **cv2.imread()** which takes the address of the image in the local machine as an input argument and read it in **BGR** format.

The image is converted to **HSV** colour scheme using OpenCV function **cv2.cvtColor()** which takes two arguments as input: image and the color conversion scheme. **cv2.COLOR_BGR2HSV** conversion scheme has been used here.

The images that are provided as input differ in brightness values throughout the image which causes even some noise ridges to be detected as contours and hence affecting the final evaluation of the expression. By making the brightness constant throughout (185) the image, which adjusts the images well for the succeeding algorithms, irrelevant contours can be eliminated to a great extent thereby improving the accuracy of evaluation.

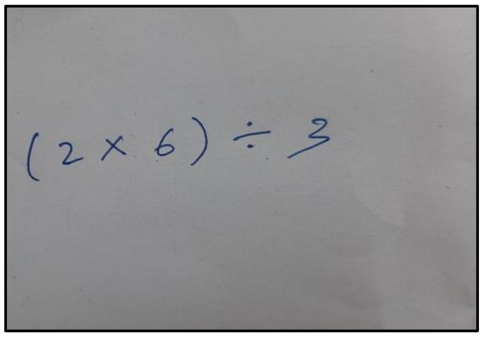


Fig 2.2 (a) Image captured by the camera.

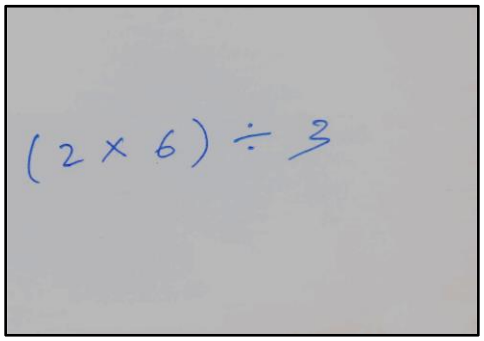


Fig 2.2 (b) Image after illumination.

2. Grayscale

An important part of preprocessing image for computer vision is converting it from colour image to grayscale image. This is important because many computer vision algorithms, like threshold and canny, require the images in grayscale. These tasks involve edge detection and so the colour information is not useful. Also, grayscale processing is faster than that of colour image processing. This is because the grayscale image has only one colour channel as opposed to three in a colour image (BGR: Blue, Green, Red).

In OpenCV this is achieved using **cv2.cvtColor()** function in which the first parameter is the image and the second parameter is the colour scheme conversion method, here it is **cv2.COLOR_BGR2GRAY**.

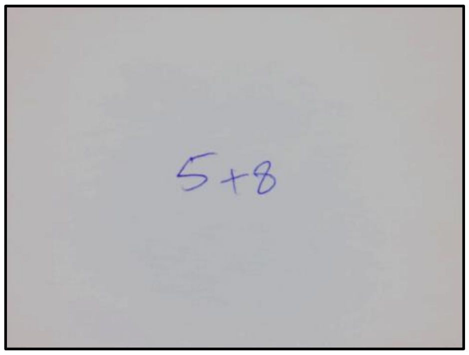


Fig 2.3 (a) Illuminated image

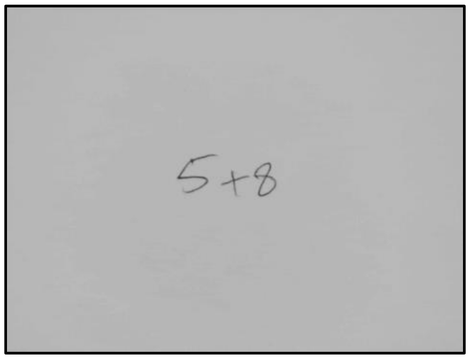


Fig 2.3 (b) Image converted to grayscale

3. Image Blur

Image Blurring refers to making the image less clear or distinct. It is done with the help of various low pass filter kernels. It has several advantages like it helps in noise removal (noise is considered as high pass signal so by the application of low pass filter kernel we restrict noise), it helps in smoothing the image and it removes low intensity edges. All this helps in smoothing and simplifying the image so that processing it becomes easy and we can easily extract the relevant features from the image.

Image blurring has many types like Median blurring, Gaussian blurring and Bilateral blurring. In the project Gaussian blurring is used because it smooths any sharp edges in the image while minimizing too much blurring which is why it showed better end results than other blurring techniques when applied in the project.

The Gaussian blurring algorithm scans over each pixel of the image and recalculates the pixel value based on the pixel values that surround it. The area that is scanned around each pixel is called the kernel. A larger kernel scans a larger amount of pixels that surround the centre pixel.

Gaussian blurring does not weigh each pixel equally. The closer a pixel is to the centre, the greater it affects the weighted average used to calculate the new centre pixel value. This way the pixels closest to the centre pixel will be closest to the true value of that pixel, so they will influence the averaged value of the centre pixel greater than pixels further away.

The OpenCV **cv2.GaussianBlur()** function takes in 3 parameters: the image, the kernel size, and the sigma values for X and Y. The kernel is the matrix that the algorithm uses to scan over the image, **7x7 kernel** has been used in the project, where the centre pixel is the pixel that will be changed with respect to the surrounding 24 pixels. The sigma dictates the standard deviation in X and Y direction. The value 1 has been used for sigma in the project. If the value 1 is provided to the function, it will serve as sigma for both X and Y. If the sigma is 0, it will be auto-calculated from the kernel size.

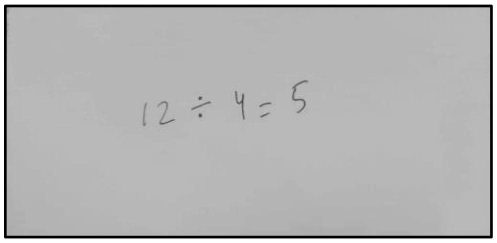


Fig 2.4 (a) Grayscale image

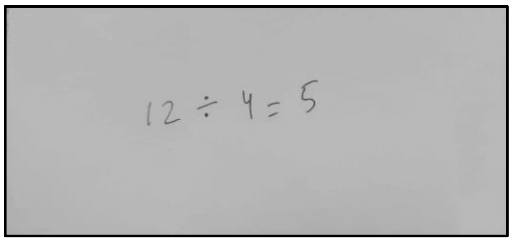


Fig 2.4 (b) Image after Gaussian blurring

4. Threshold

Thresholding simplifies an image for further analysis. Even after converting an image to grayscale, we are still left with a lot of information which is not relevant to our requirements in the project. By using threshold we can eliminate some pixels below a certain threshold value by converting them to white or black, and whatever is left is the relevant information we will be using for analysis.

There are many threshold techniques like binary, adaptive, Otsu etc. In binary threshold, if pixel intensity is greater than the set threshold, value set to 255, else set to 0 (black).

Threshold function in OpenCV is implemented using **cv2.threhsold()**. It has 4 parameters: image, threshold value, maximum value to convert pixels above threshold to and the type of threshold.

The threshold value used in preprocessing of the image was **150px**, so all the pixels having values below 150px will be turned black. This helps in separating the digits/foreground (higher pixel values) from the background (lower pixel values). The type of threshold used was **cv2.THRESH_BINARY.**

After applying all the mentioned preprocessing steps we get the image on which the segmentation operations can be performed.

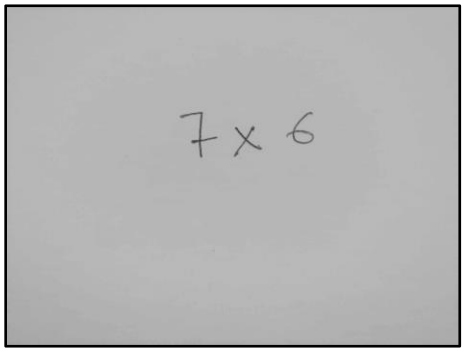


Fig 2.5 (a) Blurred image

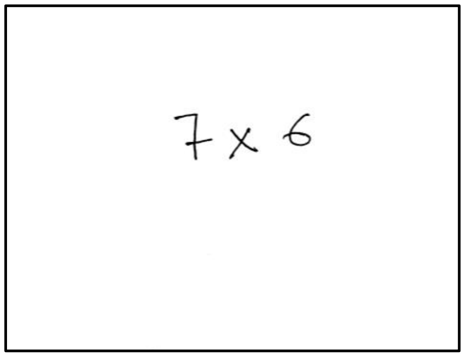


Fig 2.5 (b) Image after applying the threshold

The complete process of image preprocessing is shown in **Figure 2.6**.

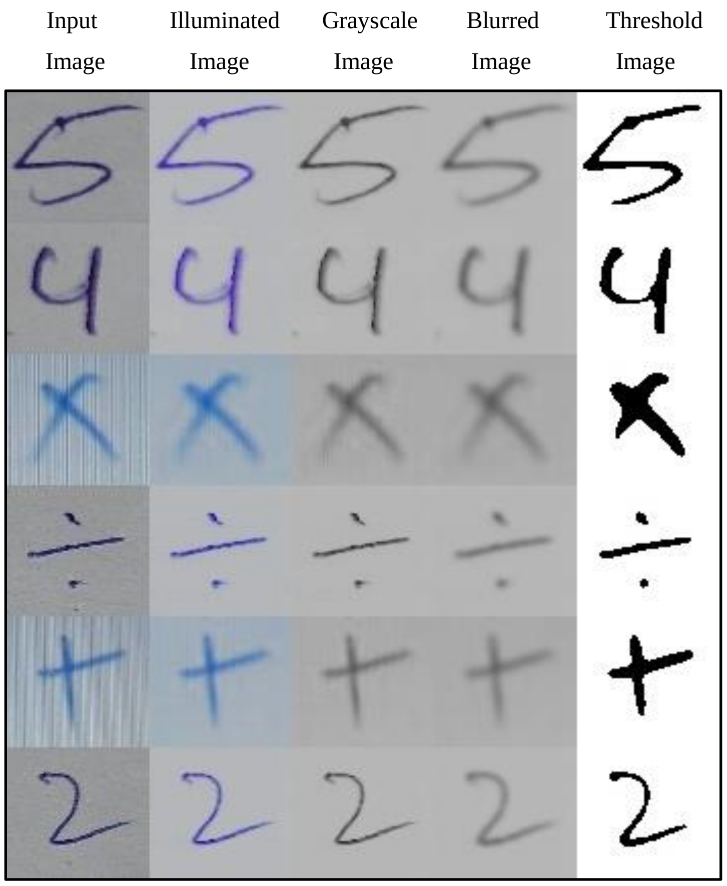


Fig 2.6 The transition of the images during various preprocessing steps.

Chapter 3

3.1 Introduction

After preprocessing of the input image of the mathematical expression, the next step is to segment and extract the symbols the process of which is explained in the **Figure 3.1**.

The preprocessed image might still contain some noise which will get detected as small contours during segmentation of the symbols. To overcome this problem, a pre-filtration technique is applied which crops the image along the maximum and minimum coordinates of the expression itself. Contours of symbols are detected using OpenCV function and their images are padded to increase the boundary space between the actual text and the edges of the image. Contours for some symbols like '=' and '÷' are detected as separate images due to the limitations of OpenCV. To overcome this, the contours were extended and combined as a single image. Finally, since the symbols are detected in random order, their proper sorting is done according to their actual order in the image.

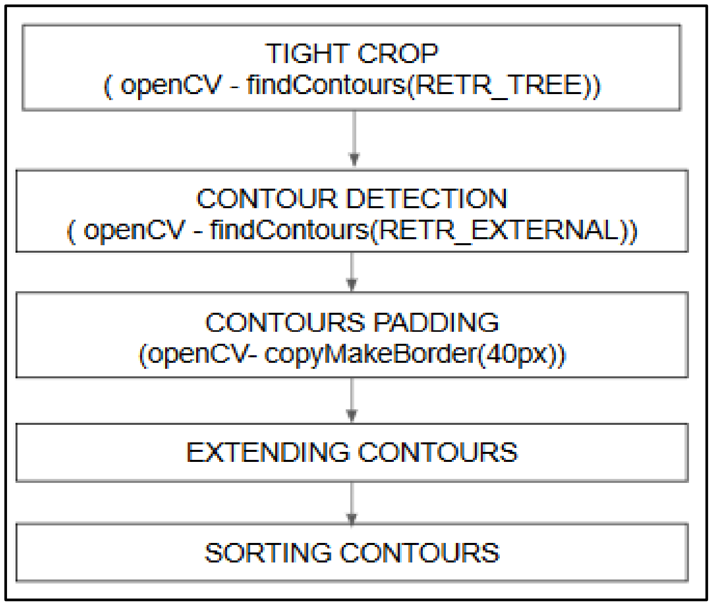

TIGHT CROP
(openCV - findContours(RETR_TREE))
CONTOUR DETECTION
(openCV - findContours(RETR_EXTERNAL))
CONTOURS PADDING
(openCV- copyMakeBorder(40px))
EXTENDING CONTOURS
SORTING CONTOURS

Fig 3.1 Steps in the segmentation process.

3.2 Implementation

1. Tight Crop

Contours are defined as the line joining all the points along the boundary of an image that are having the same intensity. Contours come handy in shape analysis, finding the size of the object of interest, and object detection.

OpenCV has **cv2.findContour()** function that helps in extracting the contours from the image. It works best on binary images, so we should first apply thresholding techniques, Sobel edges, etc. This function takes 3 parameters: image, the hierarchy of contours method and method for finding the coordinates of the contour.

In this project, contours were used to segment the operators, operands and variables from the mathematical expression present in the image. But segmenting these symbols also sometimes results in the extraction of noise present in the image present in the image which may be hard to remove all the time. Therefore, a clever method was used to get rid of the noise from the image to a great extent.

To remove contours which are very small and meaningless as they can be marks on the paper or just noise, a pre-contour filtration technique is used.

The contours are initially detected using OpenCV library function **cv2.findContours()** using **RETR_TREE** hierarchy of contours because it provides all the contours in the expression image. **cv2.CHAIN_APPROX_SIMPLE** is used which removes all the redundant points detected and compresses the contour, thereby saving memory.

From the contours that are now detected, those with **(contour area) < 0.002*(total area of the image)** are removed thereby eliminating the small and irrelevant contours which would have otherwise affected the overall accuracy of the expression evaluation. The **0.002** value was determined on a trial and error basis.

After the removal of small contours, the expression image is tightly cropped within the minimum and maximum values of x and y coordinates achieved from all of the detected contours.

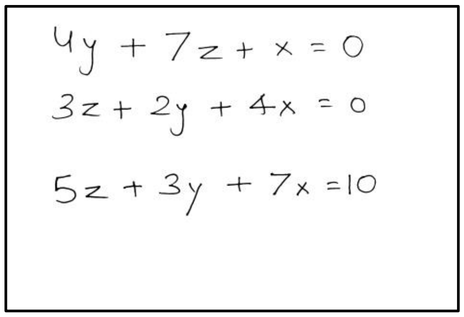


Fig 3.2 (a) Preprocessed Image

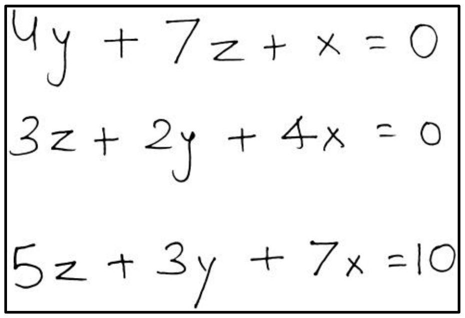


Fig 3.2 (b) Tightly cropped image

2. Contour Detection

A threshold technique with 120px value is applied to remove any remaining noise from the tightly cropped expression image.

For extracting the digits and operators from the resultant image, OpenCV **cv2.findContours()** function is used along with the **RETR_EXTERNAL** hierarchy of contours because it provides only the extreme outer contour containing the complete digit and operator. This way the complete digit or operator is enclosed within the same contour. **cv2.CHAIN_APPROX_SIMPLE** is used which removes all the redundant points detected and compresses the contour, thereby saving memory.

To filter out the small contours that might be sub-parts of operands and operators, all the contours with **(contour area) < 0.002*(total area of the image)** are removed (0.002 value achieved on trial and error basis).

Each contour containing an operand or an operator is separately stored as an image along with its x and y coordinates.

3. Padding Contours

The contours obtained from the input image are tightly cropped, so by using **cv2.copyMakeBorder()** function of OpenCV, the contours are padded with **40 pixels** on all sides thereby making sure that the value within the contour is centrally aligned for easy detection with the neural network.

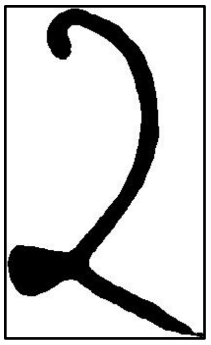


Fig 3.3 (a) Extracted contour

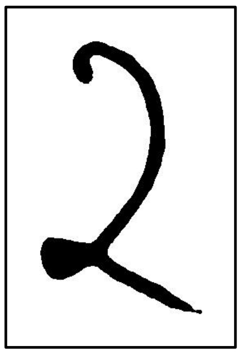


Fig 3.3 (b) Padded contour

4. Extending Contours

The contours which have x-coordinate length twice or more than twice the y-coordinate length, are extended vertically upwards and downwards. These contours are extended half of the difference in x and y-direction, each side.

Fig 3.4 (a) Extracted contour

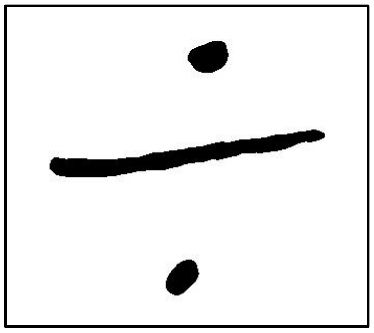


Fig 3.4 (b) Extended contour

5. Sorting Contours

This is the most important step as operators and operands should be in the correct order so that the expression can be solved correctly.

The expression image can contain multiple lines of mathematical expressions sorting which can be a complicated task.

The contours are initially sorted according to their minimum and maximum y coordinate values segregating each row of segmented contours together.

```
1. For i=0 till (number of contours)-1 do step 2.
2. For j=0 till (number of contours)-i-1 do step 3.
3. If (present contour) y_min > (next contour) y_min
   swap contours.
4. End
```

Contours from each row of a mathematical expression from the image that were clubbed together, are stored in separate arrays.

```
1. Create empty lists exps and img. Initialize prev to 0.
2. For i=0 to till (number of image contours)-1 do step 3.
3. If (current contour) y_max < (next contour) y_min
```

```
    Then create a list of contours from prev till ith contour and append to exps
    list. Create a list of images from prev till ith image and append to img list.
    Increment prev to i+1.
4.  Append left-out contours to exps list.
5.  Append left-out images to img list.
6.  End
```

The contours for separate mathematical expressions are sorted according to their x coordinate values thereby organising the contours in their original order present in the input image.

```
1.  For i=0 till (number of rows of expressions) do step 2.
2.  For k=0 till (number of images in ith row)-1 do step 3.
3.  For j=0 till (number of images in ith row)-k-1 do step 4.
4.  If (present image) x_min > (next image) x_min
    swap images.
5.  End
```

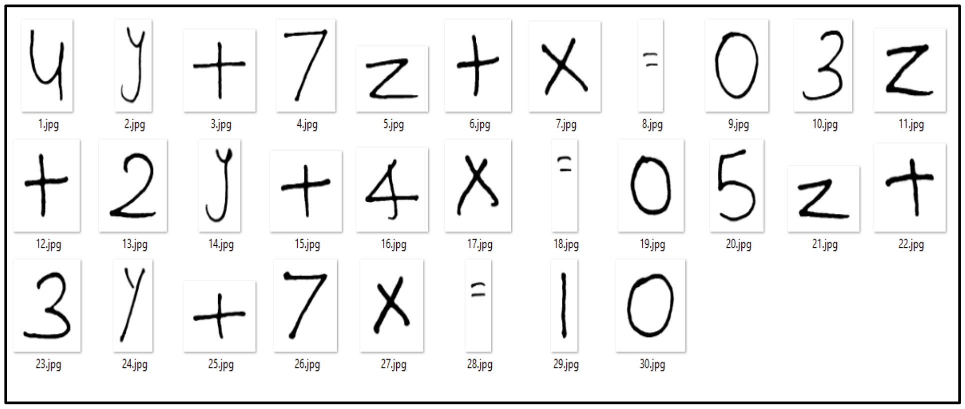


Fig 3.5 Contours in the sorted manner

(original equations: 4y+7x+x=0; 3z+2y+4x=0; 5z+3y+7x=10)

Chapter 4

4.1 Introduction

Recognition of the segmented symbols is done using neural networks. However, neural networks work only as good as their learning data. Therefore, proper data needs to be prepared to recognise symbols with high accuracy.

The Extended MNIST (EMNIST) [9] dataset is a set of handwritten character digits derived from the NIST Special Database 19, which contains digits, uppercase and lowercase handwritten letters having a size of **32x32x3**. This dataset was used with a combination of another dataset [15] which contains the images of digits, characters and mathematical operators with a size of **45x45x3**.

The images of the dataset we improved using two techniques; augmentation and preprocessing. Augmentation using simple techniques was done to increase the size of the dataset. Preprocessing of images of some symbols was done so that the text in the images could be better understood without any ambiguity.

4.2 Implementation

4.2.1 Augmenting the images

The dataset [9] used in this project had a few classes with not sufficient number of images. To increase the number of images, augmentation of the images is done by using various processes such as rotating the images upside down (in the case of vertical symmetry) and laterally inverting the images (in case of horizontal symmetry).

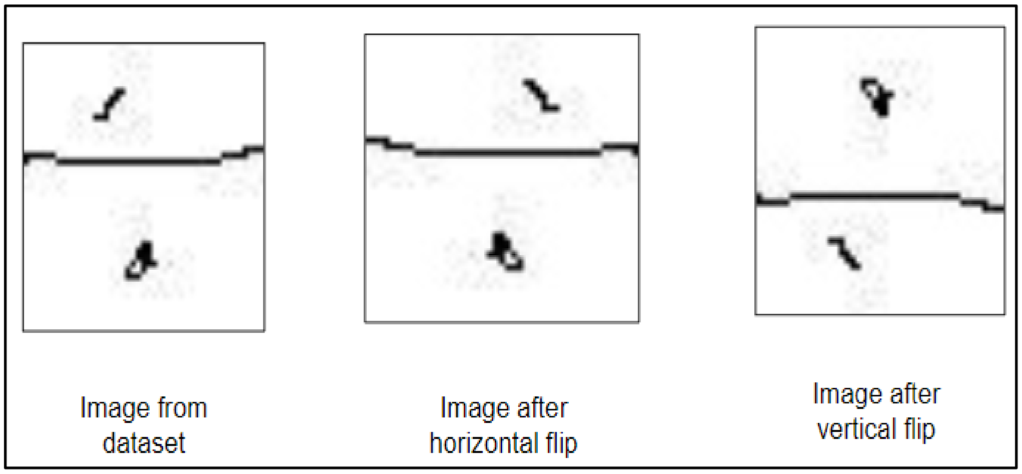


Fig 4.1 Augmented images of the dataset

4.2.2 Preprocessing the dataset images

1. **Preprocessing Operators**

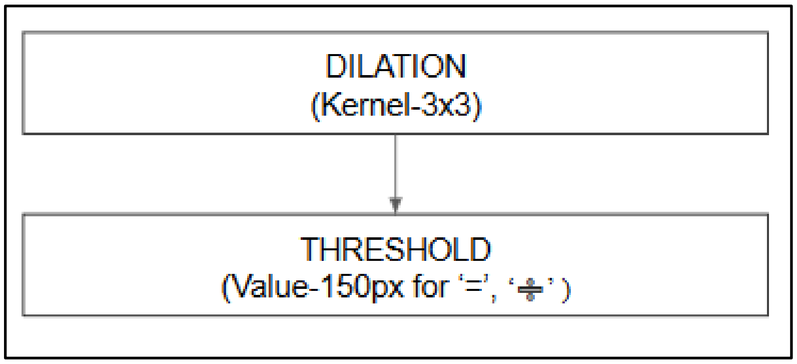


Fig 4.2 Steps involved in preprocessing of operator images

The dataset of operator images had **2471** images of **45x45x3** of each class [9]. The writing strokes in the images were thin partially visible. To overcome this each image underwent through the following preprocessing steps.

1. Dilation

   Dilation is a morphological transformation which is used to increase the size of the foreground object that has been reduced during the preprocessing. Removing noise during the preprocessing stage shrinks our object. After noise removal, we enlarge the object that is left to make it more relevant in the image.

It is implemented in OpenCV using **cv2.dilate(**) method which takes 3 parameters: image, kernel size, number of iterations for which to apply dilation.

Dilation was performed on each image with a matrix of ones having a size **3x3** as the kernel. This process smoothed the image which was pixelated due to increased size.

2. Threshold

The threshold value of **150px** has been used for the **'='** operator and **235px** for **'÷'** operator. By application of threshold, all the pixels having values below threshold value will be turned black. This helps in separating the foreground (higher pixel values) from the background (lower pixel values).

2. Preprocessing Operands and Variables

The EMNIST[9] dataset used for operands(digits) and variables(alphabets). All the digits and 10 alphabets were selected. Each class contained a few thousand images to tens of thousands of images with each image of size **32x32x3**. A total of **2471** images of each lower case alphabet were selected, to match the number of images in the operand dataset. The selected alphabets were **'a', 'b', 'c', 'd', 'u', 'v', 'w', 'x', 'y' and 'z'**. These letters were selected on the basis of the quality of the text written in the image, the possible similarities with other necessary symbols used in the classifier and the frequency of occurrence of different alphabets in different types of equations.

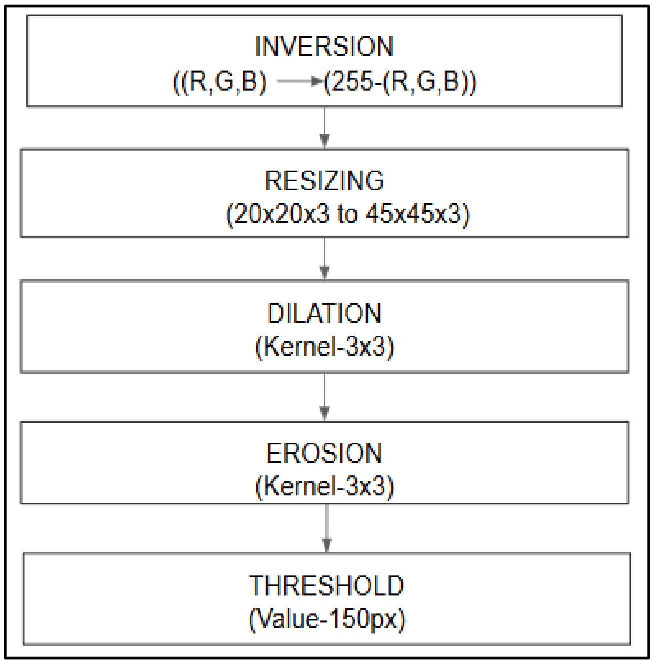


Fig 4.3 Steps involved in preprocessing operand and variable images

The following preprocessing steps were taken to ensure the quality of images that were used to train the deep neural network.

1. Inverting the RGB values

   The RGB values of each image are inverted (i.e. Subtracted from 255) since the images in the EMNIST dataset have white text on a black background. After the colour inversion, the images have black text on a white background.

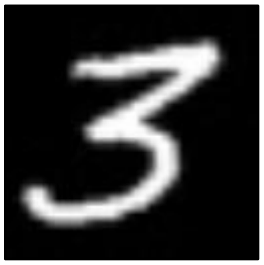


Fig 4.4 (a) Original image

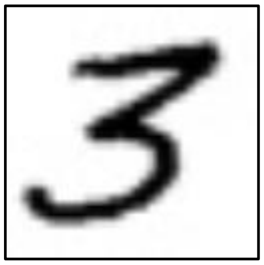


Fig 4.4 (b) Inverted color image

2. Resizing the images

Each image was resized from **32x32x3** to **45x45x3**, to match the size of images of the operators. OpenCV function **cv2.resize()** was used.

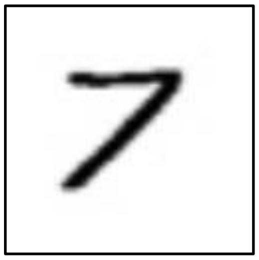


Fig 4.5 (a) Inverted image

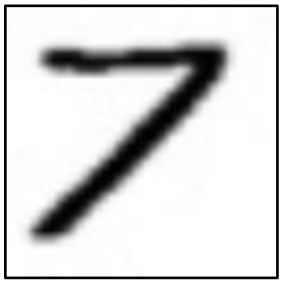


Fig 4.5 (b) Resized image

3. Dilating the image

Since the image colours have been inverted, dilation now acts as a morphological erosion scheme. In this, it erodes away the boundaries of the foreground object. The kernel slides through the image (as in 2D convolution). A pixel in the original image (either 1 or 0) will be considered 1 only if all the pixels under the kernel is 1, otherwise it is eroded (made to zero).

So what happens is that all the pixels near the boundary will be discarded depending upon the size of the kernel. So the thickness or size of the foreground object decreases or simply white region decreases in the image. It is useful for removing small white noises.

Dilation (erosion because of the inverted colour scheme) was performed on using OpenCV function **cv2.erode()** on each image with a matrix of ones having a size **3x3** as the kernel. This process smoothed the image which was pixelated due to increased size.

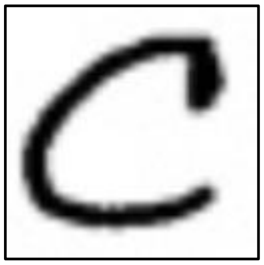


Fig 4.6 (a) Resized image

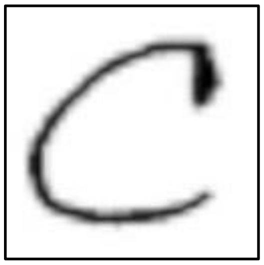


Fig 4.6 (b) Dilated (Eroded) image

4. Eroding the image

   Since the image colours have been inverted, erosion acts as a morphological dilation scheme. Dilation is a morphological transformation which is used to increase the size of the foreground object that has been reduced during the preprocessing.

   OpenCV function **cv2.dilate()** with kernel size **3x3** has been used to dilate the edges of the text that had become thin after application on morphological erosion scheme applied on it.

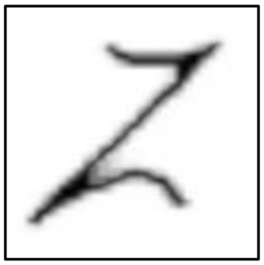


Fig 4.7 (a) Dilated (Eroded) image

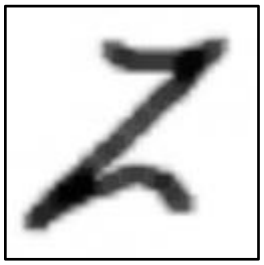


Fig 4.7 (b) Eroded (Dilated) image

5. Threshold

   The threshold value of **150px** is applied to the images of the alphabets. By application of threshold, all the pixels having values below 150px will be turned black. This helps in separating the foreground (higher pixel values) from the background (lower pixel values).

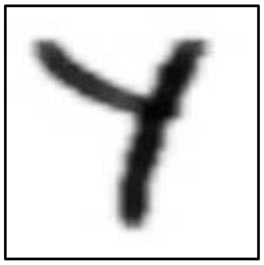


Fig 4.8 (a) Eroded (Dilated) image

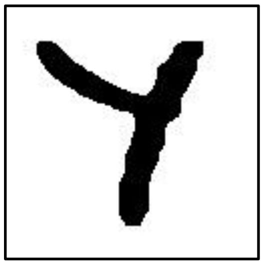


Fig 4.8 (b) Threshold image

3. Preprocessing Parenthesis

The dataset [9] had a few thousand images of both left and right parenthesis of size **45x45x3**, out of which **2471** images were used from each. Both the parenthesis were appreciably similar to the digit '1'. To make these images look like parentheses, the following preprocessing steps were applied.

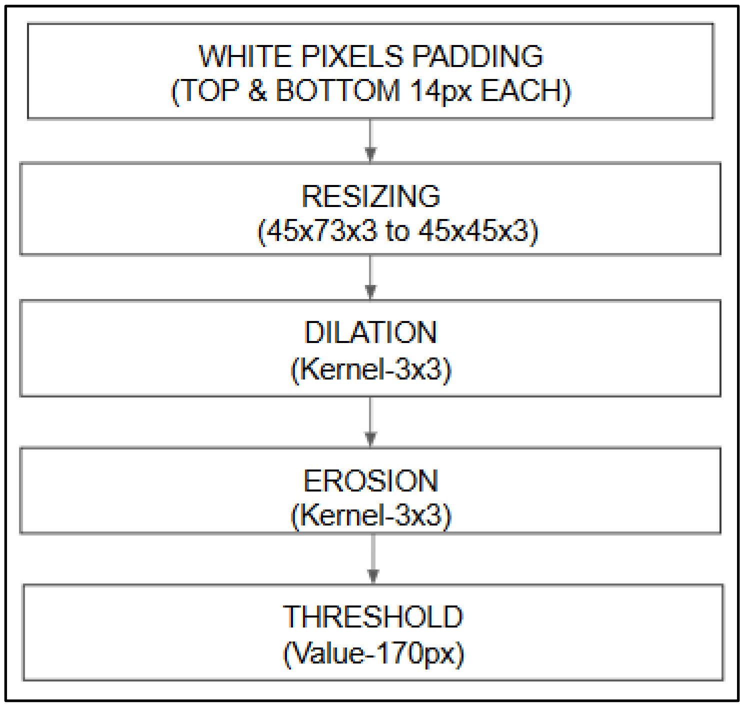


Fig 4.9 Steps involved in preprocessing the parenthesis dataset images

1. Padding the image with white pixels and resizing

   Each image was padded with **14 white pixels** on the top and the bottom of the image. After which the images resized back to 45x45.

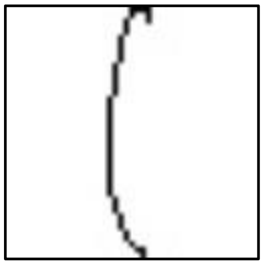


Fig 4.10 (a) Original Image

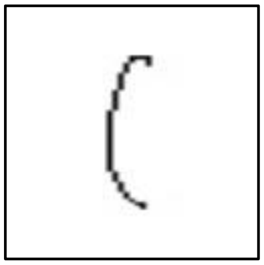


Fig 4.10 (b) After padding and resizing.

2. Eroding the image

   Each image was eroded with a matrix of ones having a size **3x3** as the kernel.

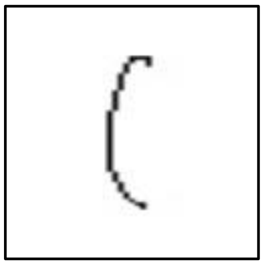


Fig 4.11 (a) Resized image

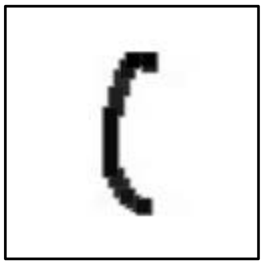


Fig 4.11 (b) Eroded image

3. Threshold

   The threshold value of **170px** is applied to the images of parenthesis. By application of threshold, all the pixels having values below 170px will be turned black. This helps in separating the foreground (higher pixel values) from the background (lower pixel values).

Fig 4.12 (a) Eroded image

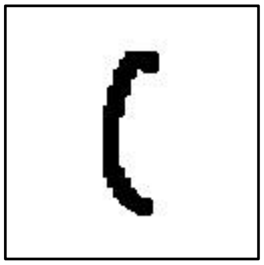


Fig 4.12 (b) After applying threshold.

Chapter 5

5.1 Introduction

The recognition of the segmented symbols from images is done using Convolutional Neural Network (CNN) because of their state-of-the-art performance in classification of images [12]. This will result in a completely digitised mathematical expression present in the image.

5.2 Implementation

Neural Networks receive an input (a single vector) and transform it through a series of hidden layers. Each hidden layer is made up of a set of neurons, where each neuron is fully connected to all neurons in the previous layer, and where neurons in a single layer function completely independently and do not share any connections. The last fully-connected layer is called the "output layer" and in classification settings, it represents the class scores.

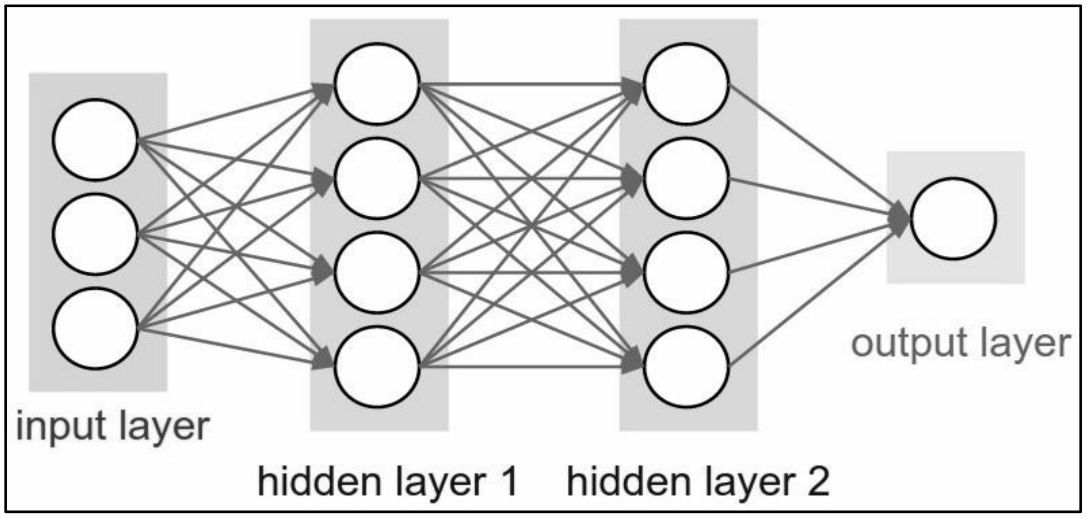


Fig 5.1 Neural Network architecture example

Neural networks learn the problem using **Backpropagation algorithm**. By backpropagation, the neurons learn how much error they did and correct themselves, i.e, correct their “weights” and “biases”. By this, they learn the problem to produce correct outputs given the inputs. Backpropagation involves computing gradients for each layer and propagating it backwards, hence the name.

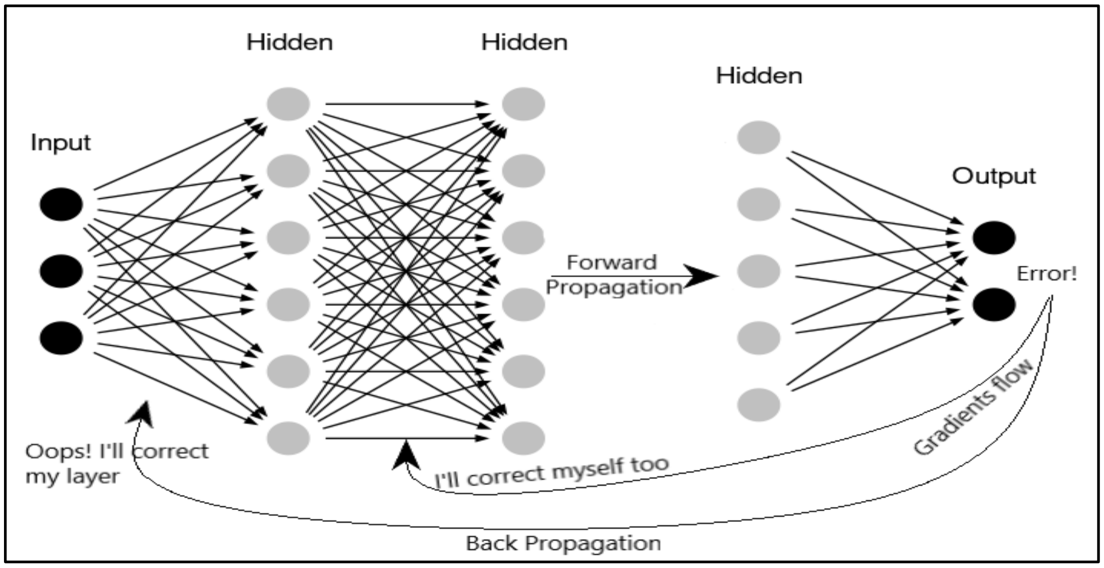


Fig 5.2 Backpropagation Algorithm

Convolutional Neural Networks (CNN/ConvNet) are very similar to ordinary Neural Networks. they are made up of neurons that have learnable weights and biases. Each neuron receives some inputs, performs a dot product and optionally follows it with a non-linearity. The whole network still expresses a single differentiable score function: from the raw image pixels on one end to class scores at the other. And they still have a loss function (e.g. SVM/Softmax) on the last (fully-connected) layer and all the tips/tricks we developed for learning regular Neural Networks still apply.

ConvNet architectures make the explicit assumption that the inputs are images, which allows us to encode certain properties into the architecture. These then make the forward function more efficient to implement and vastly reduce the number of parameters in the network.

ConvNet is a sequence of layers, and every layer of a ConvNet transforms one volume of activations to another through a differentiable function. The main types of layers used to build ConvNet architectures:

- **Convolutional Layer** compute the output of neurons that are connected to local regions in the input, each computing a dot product between their weights and a small region they are connected to the input volume.
- **Pooling Layer** performs a downsampling operation along the spatial dimensions (width, height).
- **Fully-Connected Layer** layer will compute the class scores, resulting in a volume of size [1x1xN], where each of the N numbers corresponds to a class score.
- **Flatten layer** converts the data into a 1-dimensional array for inputting it to the next layer. Flattening the output of the convolutional layers creates a single long feature vector. And it is connected to the final classification model, which is called a *fully-connected* layer.
- **Dropout** is a technique where randomly selected neurons are ignored during training. They are “dropped-out” randomly. This means that their contribution to the activation of downstream neurons is temporarily removed on the forward pass and any weight updates are not applied to the neuron on the backward pass.

RELU layer apply an elementwise activation function, such as the $\max(0,x)$max(0,x) thresholding at zero. This leaves the size of the volume unchanged.

During backpropagation of errors to the weights and biases, an undesired property of **Internal Covariate Shift** which makes the network too long to train.

During training, each layer is trying to correct itself for the error made up during the forward propagation. But every single layer act separately, trying to correct itself for the error made up.

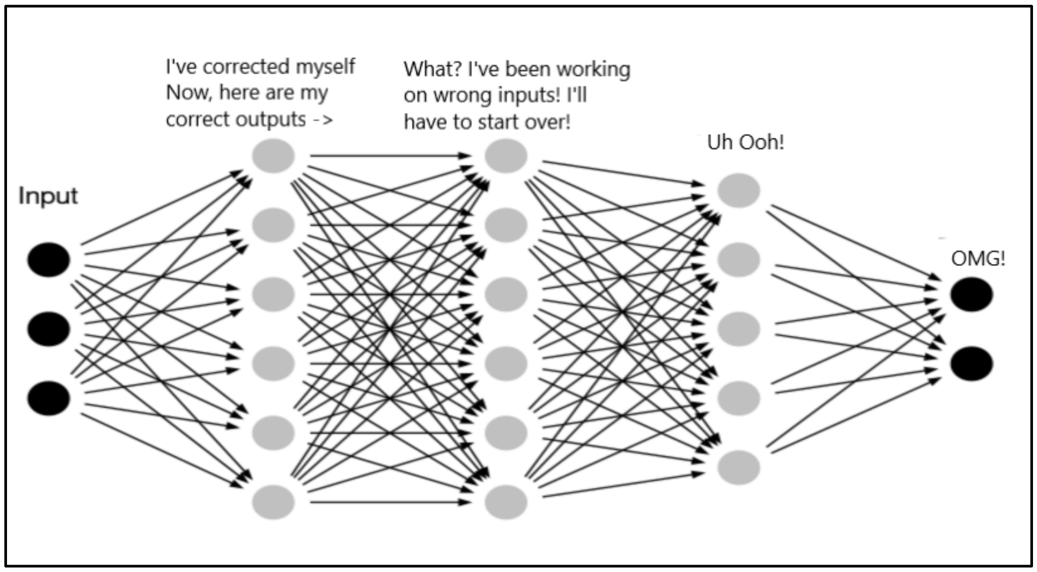

I've corrected myself Now, here are my correct outputs ->
What? I've been working on wrong inputs! I'll have to start over!
Uh Ooh!
Input
OMG!

Fig 5.3 Internal Covariate Shift

For example, in the network given above, the 2nd layer adjusts its weights and biases to correct for the output. But due to this readjustment, the output of the 2nd layer, i.e, the input of the 3rd layer is changed for the same initial input. So, the third layer has to learn from scratch to produce the correct outputs for the same data.

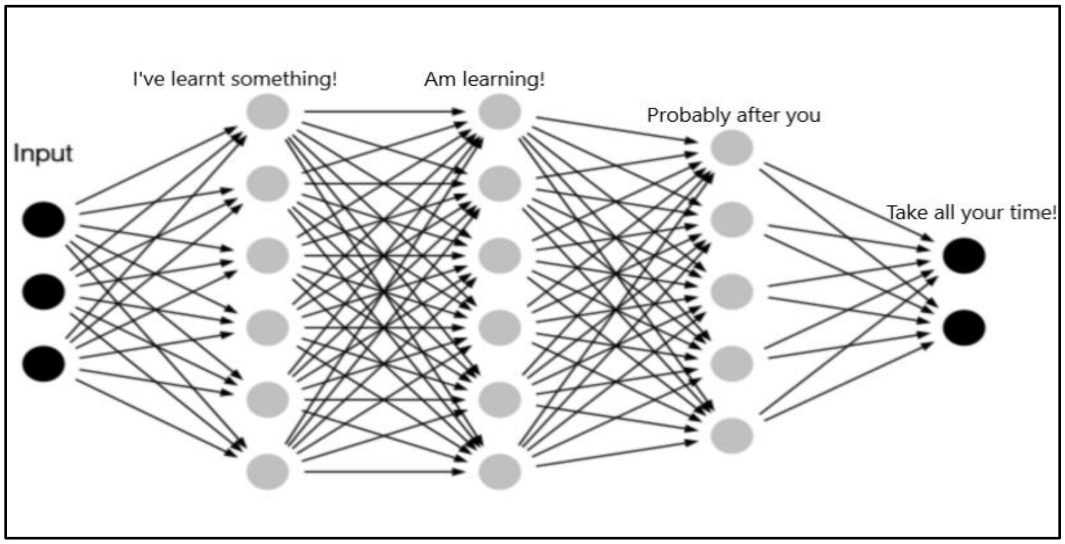


Fig 5.4 Order of training

This presents the problem of a layer starting to learn after its previous layer, i.e, 3rd layer learns after 2nd finished, 4th starts learning after 3rd, etc. So, in deep neural networks that are about 100 to 1000 layers deep. It would really take more no. of epochs to train them. More specifically, due to changes in weights of previous layers, the distribution of input values for current layer changes, forcing it to learn from new “input distribution”. To overcome this problem of **Internal Covariate Shift, Batch Normalisation** is used.

Usually, in simpler ML algorithms like linear regression, the input is “normalized” before training to make them into single distribution. Normalization is to convert the distribution of all inputs to have **mean=0** and **standard deviation=1**. So, most of the values lie between -1 and 1.

A similar technique of normalisation is also implemented in deep learning in the form of batch normalisation in-between consecutive layers of neural network. It reduces this problem of internal covariate shift, Batch Normalization adds Normalization “layer” between each layer. An important thing to note here is that normalization has to be done separately for each dimension (input neuron), over the ‘mini-batches’, and not altogether with all dimensions. Hence the name ‘batch’ normalization.

Due to this normalization “layers” between each fully connected layers, the range of input distribution of each layer stays the same, no matter the changes in the previous layer.

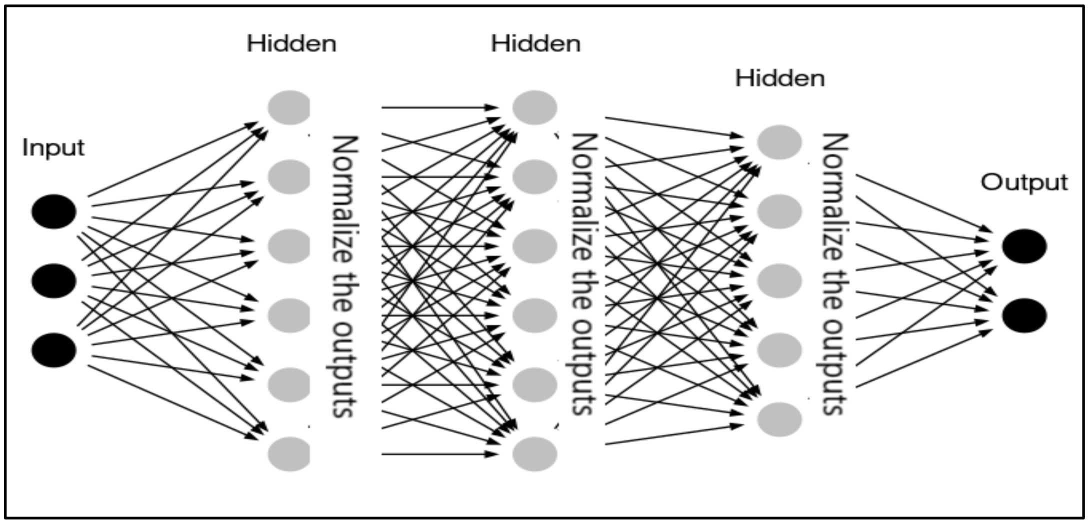
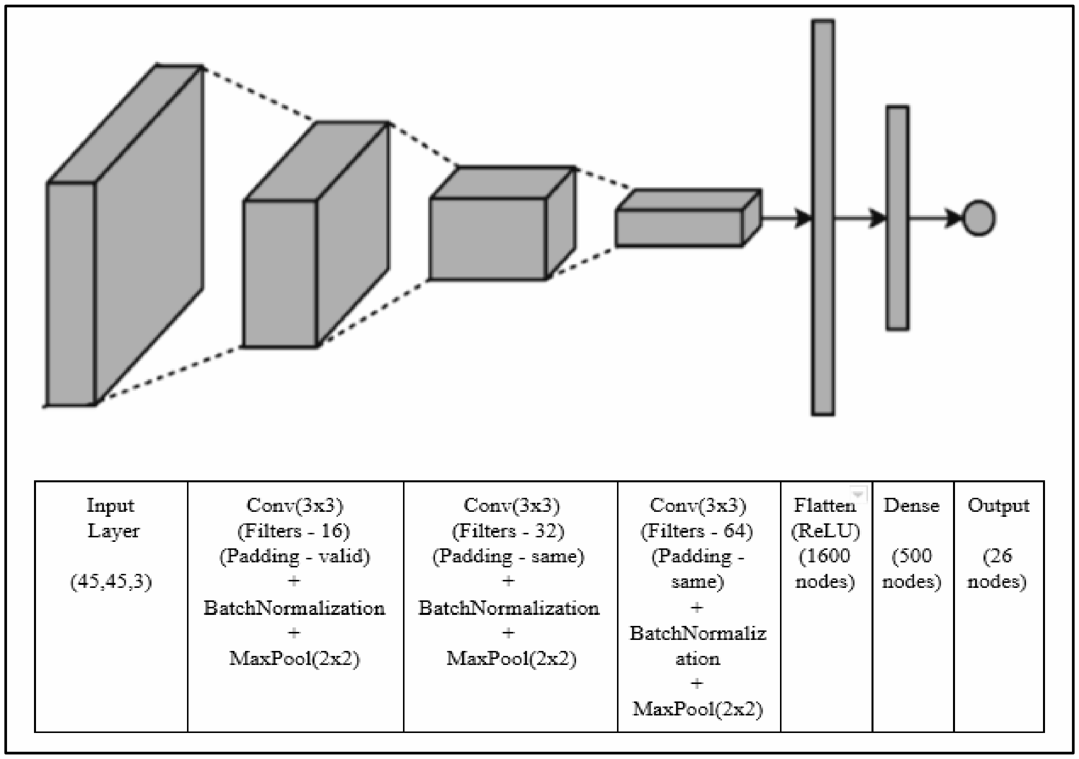


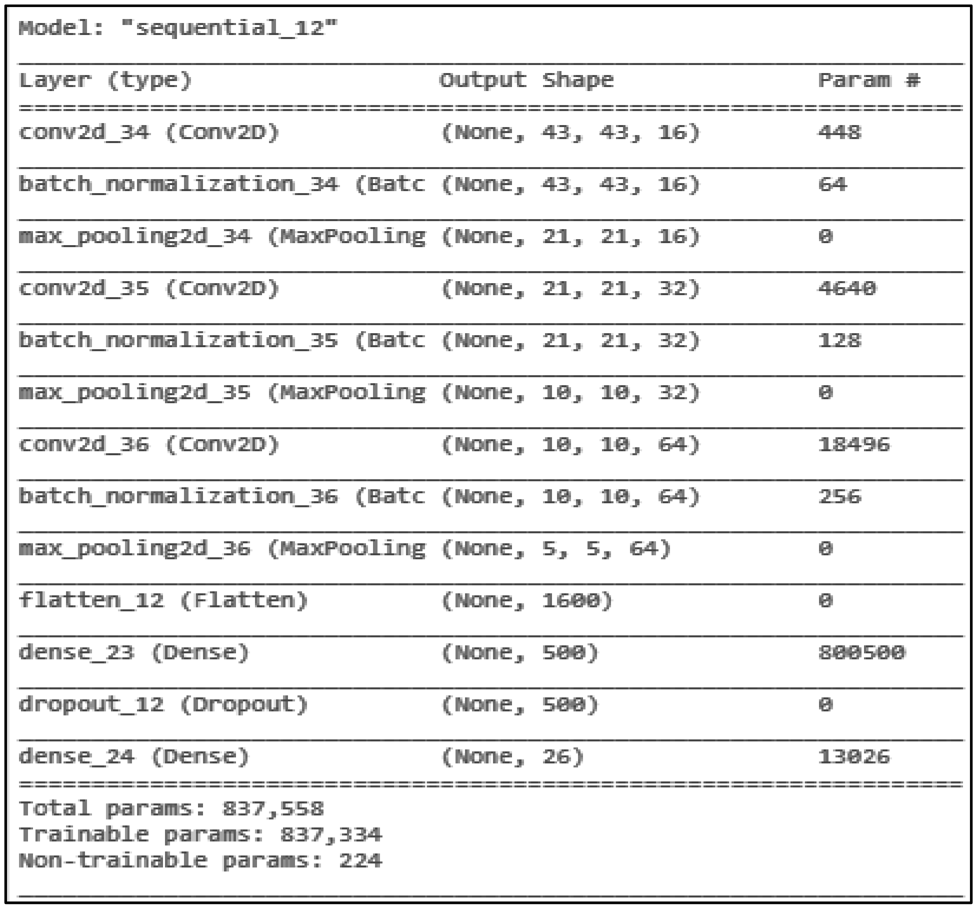
Fig 5.5 Batch Normalization



Input Layer (45,45,3)	Conv(3x3) (Filters - 16) (Padding - valid) + BatchNormalization + MaxPool(2x2)	Conv(3x3) (Filters - 32) (Padding - same) + BatchNormalization + MaxPool(2x2)	Conv(3x3) (Filters - 64) (Padding - same) + BatchNormalization + MaxPool(2x2)	Flatten (ReLU) (1600 nodes)	Dense (500 nodes)	Output (26 nodes)

Fig 5.6 Convolutional neural network architecture used in the project

The CNN model implemented to recognize handwritten characters in this project is made up of three **convolutional layers with batch normalisation and max pooling, two fully connected layers and two dropout layers** (to reduce overfitting). The deep neural network model was trained on about **61149** images with **26** classes with about **2471** images of each class.



```
Model: "sequential_12"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_34 (Conv2D)           (None, 43, 43, 16)        448
_________________________________________________________________
batch_normalization_34 (Batc (None, 43, 43, 16)        64
_________________________________________________________________
max_pooling2d_34 (MaxPooling (None, 21, 21, 16)        0
_________________________________________________________________
conv2d_35 (Conv2D)           (None, 21, 21, 32)        4640
_________________________________________________________________
batch_normalization_35 (Batc (None, 21, 21, 32)        128
_________________________________________________________________
max_pooling2d_35 (MaxPooling (None, 10, 10, 32)        0
_________________________________________________________________
conv2d_36 (Conv2D)           (None, 10, 10, 64)        18496
_________________________________________________________________
batch_normalization_36 (Batc (None, 10, 10, 64)        256
_________________________________________________________________
max_pooling2d_36 (MaxPooling (None, 5, 5, 64)          0
_________________________________________________________________
flatten_12 (Flatten)         (None, 1600)              0
_________________________________________________________________
dense_23 (Dense)             (None, 500)               800500
_________________________________________________________________
dropout_12 (Dropout)         (None, 500)               0
_________________________________________________________________
dense_24 (Dense)             (None, 26)                13026
=================================================================
Total params: 837,558
Trainable params: 837,334
Non-trainable params: 224
_________________________________________________________________
```

Fig 5.7 Summary of the CNN used to indicate the number of parameters achieved after each layer

Chapter 6

6.1 Introduction

The expression obtained after the classification is a stream of characters stored in a string. Different operations are performed for arithmetic expressions and linear equations. The first step is to extract tokens i.e, operators and operands from the stream of characters. The extracted tokens are stored as individual strings in a python list data structure. This process is done for both arithmetic as well as linear equations. After the tokens are extracted, the list is passed through a parser that uses stacks to solve arithmetic equations and a numpy function numpy.linalg.solve, which takes coefficients of linear equations and return the values of variables.

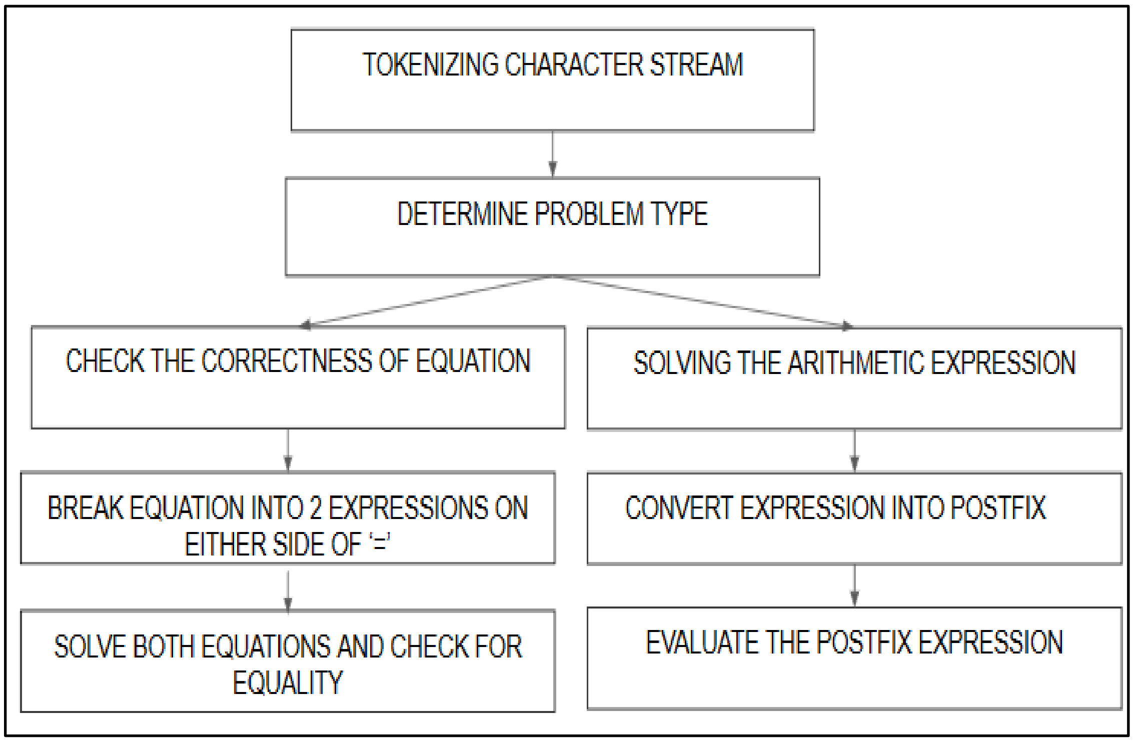

TOKENIZING CHARACTER STREAM
DETERMINE PROBLEM TYPE
CHECK THE CORRECTNESS OF EQUATION
SOLVING THE ARITHMETIC EXPRESSION
BREAK EQUATION INTO 2 EXPRESSIONS ON EITHER SIDE OF '='
CONVERT EXPRESSION INTO POSTFIX
SOLVE BOTH EQUATIONS AND CHECK FOR EQUALITY
EVALUATE THE POSTFIX EXPRESSION

Fig 6.1 Steps involved in solving the digitised expressions.

6.2 Implementation

1. Tokenizing the string

   The stream of characters are converted into a list of tokens in the order they appear in the expression.

   1. For e=0 till (length of string)-1 do step 2.
   2. If (character at position e) is not an operator do step 4.
   3. Add e to the buffer (initially empty string).
   4. Append buffer in exp_list, then append e in exp_list and then make buffer empty.
   5. If exp_list[0] == '-' do step 6.
   6. Remove exp_list[0].
   7. While ')' or an operator is not encountered do step 8.
   8. Multiply number/token with -1 and move to next token.
   9. Remove empty strings from exp_list

2. Creating a parser to solve the arithmetic equations

   First, a function that converts the string detected by the neural network into a list of strings (each string containing an operator or an operand) is created. Then a function to solve the arithmetic equations is created which first determines whether to check the correctness of the expression or to solve the expression according to BODMAS rule by first converting the expression into a postfix expression and then evaluating the postfix expression.

   1. Add '(' to the start and ')' to the end of exp_list.
   2. Flag = 0 // solve the arithmetic equation
   3. If '=' sign is present in exp_list and is not preceded by '(' nor succeeded by ')' goto step 3.
   4. Flag = 1 // check for correctness of equation

```
5.  If flag == 1 goto step 23.
6.  For j in exp_list goto step 7.  // Converting the expression into postfix
    notation from step 6 to 17.
7.  If j != ‘(‘ goto step 9.
8.  push j in the stack.
9.  if  j is not a number goto step 11
10. Append j at the end of p_exp
11. If j is not an operator goto step 14.
12. while (priority(j) <= priority(s.top)) repeat step 13 else goto step 14.
13.  a = s.pop() and pexp.append(a)
14.  if j != ‘)’ goto step 18.
15.  a = stack.pop()
16.  while a != ‘(‘) repeat step 17.
17.  Append a at end of pexp and a=stack.pop()
18. For j in pexp repeat step 19 to 21.  // Evaluating the postfix expression
    from steps 18 to 21.
19. If j is not a number goto step 21.
20. Push j on top of stack.
21. d1=stack.pop()
    d2.stack.pop()
    Push (d2 (operator) d1) on top of stack.
22. Print top of stack.  // Solution of expression
    EXIT.
23. Store expression before ‘=’ sign in exp1 and expression after ‘=’ sign in
    exp2.
24. Repeat steps 6 to 21 to solve exp1 and exp2.
25. If solution of exp != solution of exp2 goto step 27.  // Checking for
    correctness of expression
26. Print “TRUE”. EXIT.
27. Print “FALSE”. EXIT.
```

3. Creating a parser to solve the linear equation

If the expression is a set of linear equations, they are passed through a function to extract the coefficients of variables used in the system of linear equations (in the form of a stream of characters) and then passing the coefficients to a numpy function numpy.linalg.solve to solve them and return the values of variables used in that system of equations.

```
1.  Sign = 1  // After = sign the sign of coefficients should be negative
    Buffer = ‘ ‘
    Coeff = {‘zz’ : 0}  // Using zz as the key for constant
2.  For e in equation repeat steps 3 to 21.
3.  If e != ‘=’ goto step 8.
4.  Sign = -1  // If a constant is just before the '=' we need to store it in the coefficient sum
5.  If buffer == ' '  goto step 7.
6.  coeff['zz'] += int(buffer)
    Empty buffer.
7.  Skip steps 8 to 21.
8.  If e is not a variable goto step 18.
9.  If buffer is not empty goto step 11.  //If buffer is empty, it means its coefficient is 1
10. value=1
11. If buffer does not contain an operator goto step 13.  //If buffer is '-' or '+', it means the coefficient is 1
12.  Add ‘1’ at end of buffer.
13. Value = sign * value in buffer.
14. If e has encountered before goto step 16.
15. Coeff[e] = value and goto step 17.
16. Coeff[e] += value
17. Make buffer empty and goto step 2.
```

```
18. If (e is an operator and buffer is empty) or e is not an operator goto step 19
    else goto 20.
19. Add e to buffer.
20. If e is an operator and buffer is not empty goto step 21 else goto 22.
21. coeff['zz'] += sign* numeric value of buffer
    Buffer = e  // We need the constant term on the right side of '='
22. Change the sign of coeff['zz']
    coeff['zz'] += numeric value buffer
23. End
```

Chapter 7

7.1 Experimental Result

The experimental results are shown in table 7.1, which shows the result of k-fold cross-validated approach used to train the convolutional neural network for better results and lower overfitting. The accuracy of the customised convolutional neural network on the training and validation dataset improves with the increasing number of epochs as shown in figure 7.1.

Table 7.1 Accuracy and loss of the model on the training and validation sets on each fold

Fold No.	**Training accuracy (max)**	**Training loss (min)**	**Validation accuracy (max)**	**Validation loss (min)**
Fold 1	0.9639	0.1115	0.9786	0.0635
Fold 2	0.9787	0.0686	0.9894	0.0260
Fold 3	0.9882	0.0371	0.9955	0.0144
Fold 4	0.9908	0.0287	0.9968	0.0108
Fold 5	0.9904	0.0291	0.9982	0.0064
Fold 6	0.9946	0.0162	0.9989	0.0028
Fold 7	0.9955	0.0150	0.9993	0.0019
Fold 8	0.9948	0.0177	0.9991	0.0042
Fold 9	0.9962	0.0137	0.9993	0.0015
Fold 10	0.9966	0.0111	0.9998	0.0006
AVG.	0.9889	0.032	0.9954	0.01321

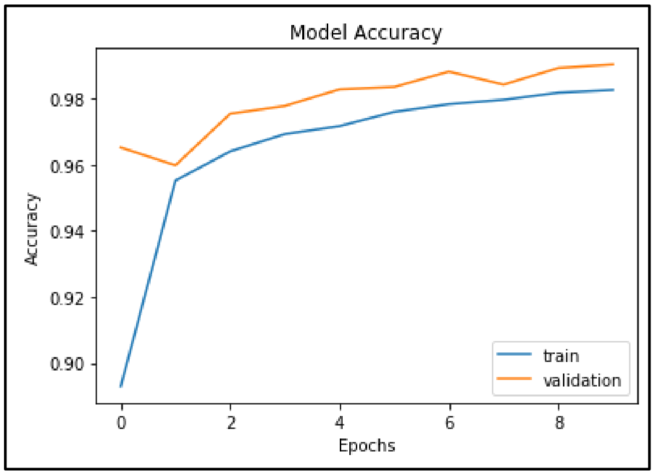
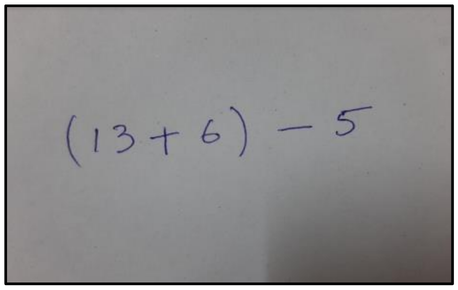


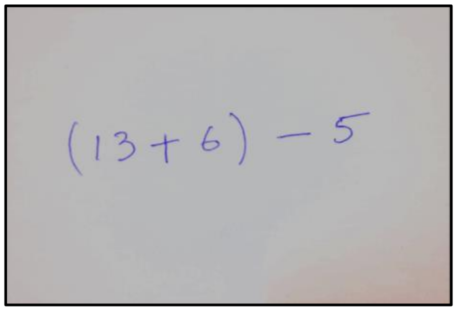
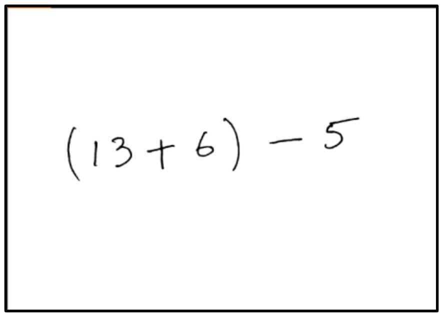
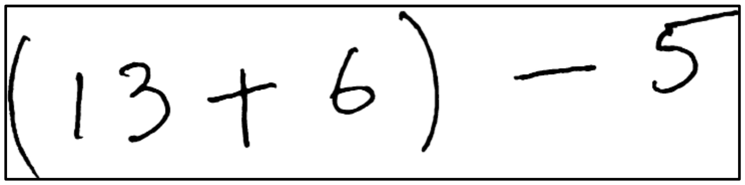
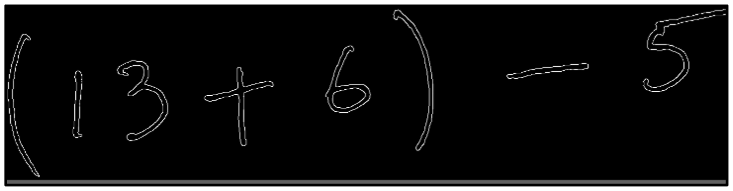
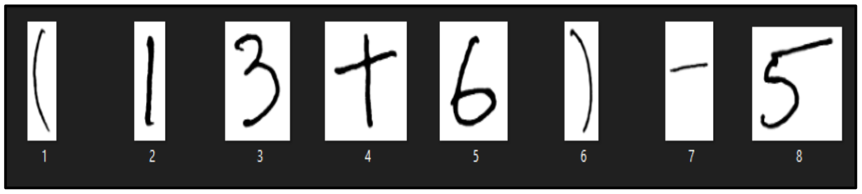
Fig 7.1 Model accuracy on training and validation set.

The problem of detecting '=' and '÷' was overcome by our proposed approach of sorting the contours according to x coordinates which was similar to the approach used in [7]. The problem of ambiguity between '*' and 'x' symbols was solved by checking whether the succeeding token in the string is a digit or '(' in the case of '*' and any other symbol otherwise. The augmentation and preprocessing of training data images used for the customised convolutional neural network helped in improving the classification of alphabets and parentheses. The solution to these problems is shown in table 7.2.

The proposed system was also tested on various other self-shot images having various kinds of expressions and equations. The proposed system was able to recognize and evaluate them successfully.

Table 7.2 Different operations being performed on the input image

Original image	

Illuminated image	
Preprocessed image	
Tightly cropped image	
Canny edge detection	
Sorted contours	 (1 1 2 3 3 + 4 6 5) 6 − 7 5 8

Chapter 8

8.1 Conclusion and Future Work

This paper focused on segmentation, recognition and evaluation of offline handwritten mathematical expressions. Only arithmetic and linear mathematical expressions were considered in this paper. A pre-contour filtration technique was suggested to remove distortions from segmented symbols which was able to reduce the noise from images to a great extent. The sorting technique designed was able to preserve the equation order for any number of expressions in the image. The customised convolutional neural network designed gave a convincing result with an accuracy of 97% in recognising the segmented symbols. Finally, the correct evaluation was achieved for the tested expressions.

In future work, the segmentation technique needs to be further improved because the sub-parts of symbols are being detected as a separate contour leading to erroneous detection. The proposed method can be extended to quadratic equations by employing the same technique of segmentation and recognition. Also, the dataset can be expanded to include the remaining alphabets, thus increasing the domain of the system.

References

[1] S. Shinde, R. B. Waghulade, D. S. Bormane, “A new neural network based algorithm for identifying handwritten mathematical equations” in *International Conference on Trends in Electronics and Informatics ICEI,* pp. 204-209, 2017.

[2] S. Shinde, R. B. Waghulade, “An Improved Algorithm for Recognizing Mathematical Equations by Using Machine Learning Approach and Hybrid Feature Extraction Technique” in *International Conference on Electrical, Instrumentation and Communication Engineering (ICEICE2017),* pp. 1-7, Dec 2017.

[3] I. Ramadhan, B. Purnama, S. A. Faraby, “Convolutional Neural Networks Applied to Handwritten Mathematical Symbols Classification” in *Fourth International Conference on Information and Communication Technologies (ICoICT),* pp. 1-4, 2016.

[4] G. S. Tran, C. K. Huynh, T. S. Le, T. P. Phan, “Handwritten Mathematical Expression Recognition Using Convolutional Neural Network” in *3rd International Conference on Control, Robotics and Cybernetics (CRC),* pp. 15-19, 2018.

[5] L. D’ Souza, M. Mascarenhas, “Offline Handwritten Mathematical Expression Recognition using Convolutional Neural Network” in *International Conference on Information, Communication, Engineering and Technology (ICICET),* Zeal College of Engineering and Research, Narhe, Pune, India, pp. 1-3, 2018.

[6] Y. Hu, L. Peng, Y. Tang, “On-line Handwritten Mathematical Expression Recognition Method based on Statistical and Semantic Analysis” in *11th IAPR International Workshop on Document Analysis Systems*, pp. 171-175, 2014.

[7] M. B. Hossain, F. Naznin, Y. A. Joarder, M. Z. Islam, M. J. Uddin, “Recognition and Solution for Handwritten Equation Using Convolutional Neural Network” in *Joint 7th International Conference on Informatics, Electronics & Vision (ICIEV),* pp. 250-255, 2018.

[8] A. D. Le, M. Nakagawa, “Training an End-to-End System for Handwritten Mathematical Expression Recognition by Generated Patterns” in *14th IAPR International Conference on Document Analysis and Recognition,* pp. 15-19, 2017.

[9] G. Cohen, S. Afshar, J. Tapson, & A. van Schaik, “EMNIST: an extension of MNIST to handwritten letters”, 2017 [Online]. Available: http://arxiv.org/abs/1702.05373 [Accessed Aug. 08, 2019].

[10] C. Lu, K. Mohan., “Recognition of Online Handwritten Mathematical Expressions Using Convolutional Neural Networks”, Dept. Comput. Sci., Stanford Uni., California, cs231n project report, 2015.

[11] A. M. Awal, H. Mouchère, C. V. Gaudin., “Towards Handwritten Mathematical Expression Recognition” in *10th International Conference on Document Analysis and Recognition*, pp. 1046-1050, 2009.

[12] A. Krizhevsky, I. Sutskever, G. E. Hinton, "ImageNet classification with deep convolutional neural networks", *Neural Information Processing Systems,* vol. 1, 25, pp. 1097-1105, Dec 2012.

[13] Y. Chajri, A. Maarir, B. Bouikhalene, “A comparative study of Handwritten Mathematical Symbols recognition”, *International Conference Computer Graphics, Imaging and Visualization (CGIV),* vol. 1, 13, pp. 448-451, 2016.

[14] A. M. Hambal, Z. Pei, F. L. Ishabailu, “Image Noise Reduction and Filtering Techniques”, *International Journal of Science and Research (IJSR),* vol. 6, 3, pp. 2033-2038, March 2017.

[15] Handwritten math symbols dataset [Online]. Available: https://www.kaggle.com/ xainano/handwrittenmathsymbols