VisiSolve

Vision-Based Equation Resolver

Sebastian Ho, Matteo Negro, Andrea Paparella,
Anup Raj & Dwij Shetty

Project Progress Report

December 3, 2023

December 3, 2023

1 Introduction

In this report, we explore the field of image processing and machine learning, specifically focusing on the recognition of handwritten mathematical equations. The ability to recognize and interpret handwritten symbols, especially in the context of mathematics, presents a unique set of challenges. This study aims to address these challenges by implementing and evaluating various image preprocessing techniques and neural network architectures.

The process begins with image preprocessing, an essential step to prepare images for effective analysis by neural networks. We employ several techniques, including noise reduction, brightness adjustment, and histogram equalization, to enhance the clarity and contrast of the images. The primary goal here is to improve the visibility of the symbols while minimizing the interference of background noise.

Next, we delve into the segmentation process, where the focus is on isolating individual mathematical symbols from the images. This step is crucial for the subsequent recognition process. We experiment with different methods, such as contour detection and Mean Shift Clustering, to determine the most effective approach for symbol extraction.

Following segmentation, we standardize the segmented symbols to facilitate easier recognition by the neural network. This involves centering and resizing the symbols against a standard background.

For the recognition of these standardized symbols, we test and compare various neural network architectures, including shallow and deep multi-layer perceptrons (MLPs) and convolutional neural networks (CNNs), to identify the most effective model in terms of accuracy and generalization.

This report details each step of the process, from preprocessing to neural network implementation, highlighting the successes and limitations encountered. Our findings underscore the significance of advanced image preprocessing and the effectiveness of deep CNNs in the field of handwritten equation recognition.

2 Preprocessing

2.1 Image Cleaning

To achieve accurate segmentation of symbols and precise prediction with a CNN, it is essential to ensure that the numbers are clearly displayed in the image. Several different methods of augmenting the image were experimented with to achieve this result.

2.1.1 Noise reduction

Two versions of noise reduction have been developed and implemented. The first version employs basic thresholding and blurring techniques, while the second version utilizes advanced methods like shadow removal, erosion, and adaptive thresholding. Users can select the desired version by passing a parameter via the command line. To run Version 1, execute `pipeline.py -n 1`. For Version 2, use `pipeline.py -n 2`.

Version 1 This version begins by applying the thresholding technique. It then iteratively applies a blur (using the `cv2.blur` function) and re-applies thresholding. This process is repeated for a specified number of iterations. While this method can help reduce noise, it may also blur the details in the image.

Version 2 Version 2 employs a more complex series of steps, including shadow removal, erosion, and adaptive thresholding for noise reduction. It dilates the image and applies a median blur, creating a "shadow" version of the image. This shadow is then subtracted from the original image, effectively reducing the shadows [6]. Then it uses erosion to remove fine lines, such as those in ruled notebooks [1]. Finally, it employs adaptive thresholding techniques (such as Otsu's, mean, or Gaussian) which are more sophisticated compared to the basic thresholding used in Version 1. Otsu's method, for instance, analyzes the histogram of pixel intensities, assumes a bimodal distribution, and selects an optimal threshold between the two peaks [5].

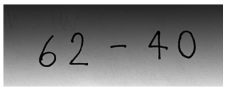
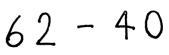
Original	
Version 1	
Version 2	

Table 1: Output comparison between Version 1 and Version 2

2.1.2 Brightness

Altering the brightness of the image can actually impair results in certain scenarios. If the image text is light-colored and the background is sufficiently dim, increasing brightness may make the background more prominent than the text. This complicates the selection of a suitable threshold or output format, whether it be white text on a black background, or vice versa. Thus, this adjustment was not included in the preprocessing.

Original	Brightened
	
	

Table 2: Applying brightness to the image, then thresholding

2.1.3 Histogram Equalization

Adding contrast is helpful for detecting symbols in low-contrast images. This approach is particularly beneficial when the background is poorly lit or when the symbols are faintly written. The function `cv2.equalizeHist` from OpenCV [4], which stretches out the intensity range of the image to increase contrast, was utilized.

Despite its potential benefits, this approach was not optimal for our preprocessing needs. The primary issue encountered was the inadvertent amplification of background features. This enhancement of unwanted elements, or noise, in the image made them stand out more, which was counterproductive.

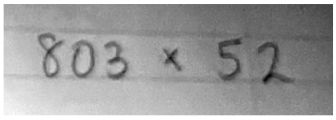
Original	
No Hist-Eq	
Hist-Eq applied	

Table 3: Histogram Equalization effects on low-contrast images

2.2 Segmentation

Photos of equations, after the cleanup process, need to be separated into their constituent symbols so that each symbol can be fed as input to the neural network and recognized. Recognition allows for the digitization of the equation and eventually its computation.

2.2.1 Symbol Extraction

Two versions of symbol extraction have been developed and implemented. The first version employs a Mean Shift Clustering approach, while the second version utilizes Contour Detection for symbol extraction. Users can select the desired version by passing a parameter via the command line. To run Version 1, execute `python pipeline.py -s 1`. For Version 2, use `python pipeline.py -s 2`.

2.2.2 Version 1 - Mean Shift Clustering

This symbol extraction algorithm decomposes an image into its constituent symbols based on the clustering of feature points. It uses Harris corner detection (`cv2.cornerHarris`) to find feature points in the image. It filters out feature points based on the intensity threshold (`par4 * max_value`). Here, `param4` is a hyperparameter that needs to be tuned. It employs the Mean Shift clustering algorithm to cluster the feature points. It filters clusters based on the `threshold_cluster`. This allows to discard any noise in the image that might otherwise be incorrectly classified as a separate symbol.

To then extract the symbols, it iteratively processes each cluster. It determines the cutting coordinates (left, right, upper, and bottom cuts). It cuts the image to extract and pad each symbol.

2.2.3 Version 2 - Contouring

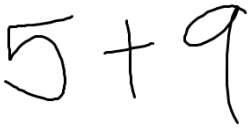
This symbol extraction algorithm identifies the outlines of objects and shapes in an image. The contours are detected using the `cv2.findContours` function from the OpenCV library[2]. This method keeps the external contours and ignores the child ones. Child contours refer to contours that are enclosed within another contour. This step ensures that only the primary shapes in the image are considered, avoiding nested or minor elements that might complicate the analysis.

The contours are then sorted, and a margin is applied to each detected symbol to ensure that the entire symbol is captured without being cut off. The function also calculates the density of each symbol - a measure of how many of the pixels within the symbol's boundary are part of the

symbol itself, compared to the background. Symbols with a density higher than a set threshold (in this case, 80%) are considered significant and are saved for further analysis. This process helps in distinguishing between meaningful symbols and noise.

2.2.4 Results

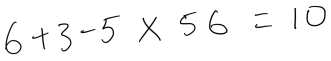
Version 2 proved to be more effective. The method's effectiveness is highlighted by visual comparisons between the original handwritten equations and their segmented versions, demonstrating successful segmentation of individual symbols.



(a) Original handwritten equation

(b) Segmented symbols from the original equation

Figure 1: Comparison of original and segmented symbols for the first equation



(a) Second original handwritten equation

(b) Segmented symbols from the second original equation

Figure 2: Comparison of original and segmented symbols for the second equation

2.3 Standardization

Standardization aims to transform separated number images into a format compatible with the CNN. The images, sourced from the segmentation phase, have variable shapes and sizes. Each symbol is placed on top of a white background, centered, and re-scaled to the CNN's input size of 28x28. While various sub-sampling methods exist, utilizing the same sub-sampling method employed during the CNN's training phase can prevent sub-sampling bias.

3 Image Recognition

3.1 Dataset

The dataset utilized is a collection of handwritten mathematical operators and numbers. It comprises 554 images for each symbol, with each image having dimensions of 155x135 pixels. The dataset is publicly available here [3].

3.2 Experimentation & Analysis

Several neural network architectures have been evaluated, including multi-layer perceptrons (MLPs) and convolutional neural networks (CNNs). For both types, both shallow and deep architectures were tested. The deep convolutional neural network emerged as the superior model.

It demonstrated enhanced generalization capabilities, largely due to the incorporation of pooling techniques. Furthermore, this model adeptly captured the underlying patterns of the dataset while maintaining a controlled variance, thus avoiding overfitting.

Let's analyze and compare all the architectures in detail.

Table 4: Architecture of the Shallow MLP

Layer	**Type**	**Input Shape**	**Output Shape**	**Activation**	**Remarks**
fc1	nn.Linear	1 * 28 * 28	64	ReLU	First fully connected layer
fc2	nn.Linear	64	14	None	Output layer for classification with 14 neurons

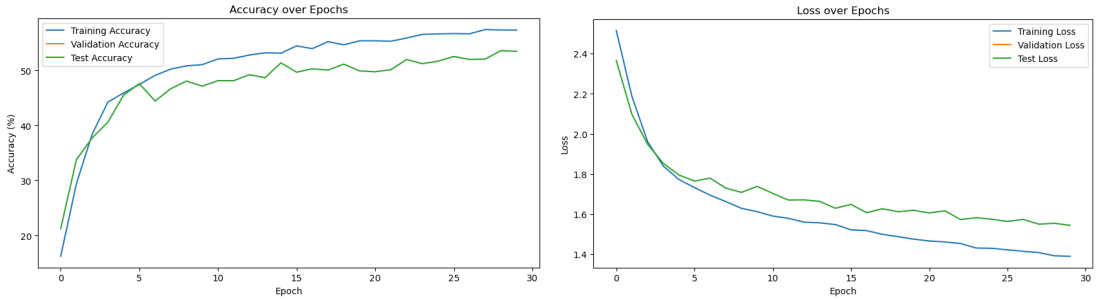


Figure 3: Accuracy and Loss for Shallow MLP

Table 5: Architecture of the Deep MLP

Layer	**Type**	**Input Shape**	**Output Shape**	**Activation**	**Remarks**
fc1	nn.Linear	1 * 28 * 28	128	ReLU	First fully connected layer
fc2	nn.Linear	128	64	ReLU	Second fully connected layer
fc3	nn.Linear	64	14	None	Output layer with 14 neurons for classification

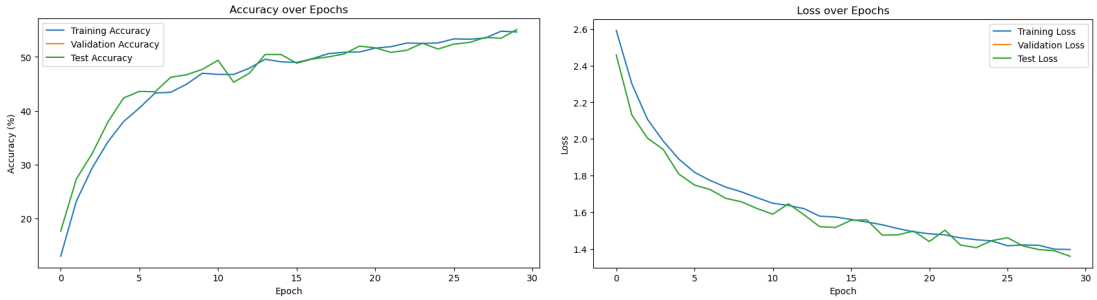


Figure 4: Accuracy and Loss for Deep MLP

Table 6: Architecture of the Shallow CNN

Layer	**Type**	**Input Shape**	**Output Shape**	**Activation**	**Remarks**
conv1	nn.Conv2d	1 x 28 x 28	32 x 28 x 28	ReLU	Convolutional layer with 32 filters, kernel size 3, stride 1, padding 1
pool	nn.MaxPool2d	32 x 28 x 28	32 x 14 x 14	-	Max pooling layer with kernel size 2, stride 2
fc1	nn.Linear	32 * 14 * 14	64	ReLU	First fully connected layer after flattening
fc2	nn.Linear	64	14	None	Output layer with 14 neurons for classification

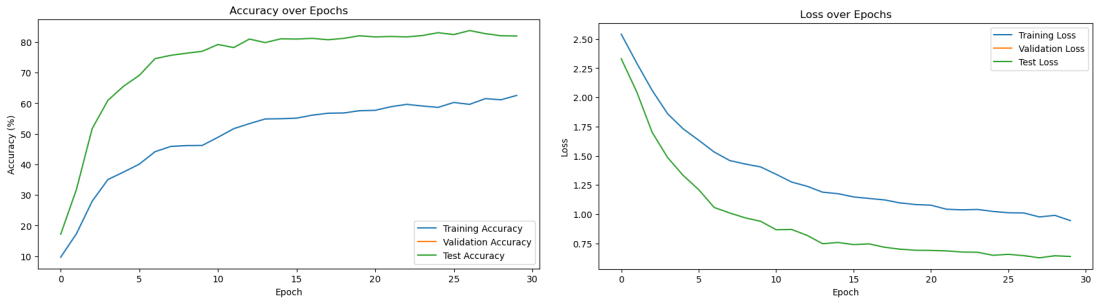


Figure 5: Accuracy and Loss for Shallow CNN

Table 7: Architecture of the Deep CNN

Layer	**Type**	**Input Shape**	**Output Shape**	**Activation**	**Remarks**
conv1	nn.Conv2d	1 x 28 x 28	32 x 28 x 28	ReLU	Convolutional layer with 32 filters, kernel size 3, stride 1, padding 1
pool	nn.MaxPool2d	32 x 28 x 28	32 x 14 x 14	-	Max pooling layer with kernel size 2, stride 2
conv2	nn.Conv2d	32 x 14 x 14	32 x 14 x 14	ReLU	Second convolutional layer with 32 filters, kernel size 3, stride 1, padding 1
fc1	nn.Linear	32 * 14 * 14	64	ReLU	First fully connected layer after flattening
fc2	nn.Linear	64	14	None	Output layer with 14 neurons for classification

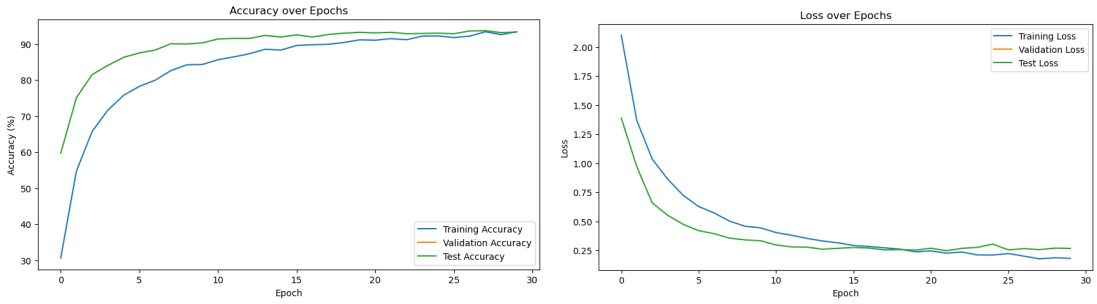


Figure 6: Accuracy and Loss for Deep CNN

Table 8: Performance Comparison of Different Architectures

Architecture	Epochs	Train Loss	Train Accuracy	Test Loss	Test Accuracy
Shallow MLP	30	1.38	57.34%	1.54	53.47%
Deep MLP	30	1.39	54.64%	1.36	55.08%
Shallow CNN	30	0.94	62.56%	0.64	81.97%
Deep CNN	30	0.18	93.37%	0.27	93.3%

3.3 Considerations

Both Multilayer Perceptrons (MLPs) have lower accuracy compared to CNNs. Deep MLPs, however, perform better than shallow MLPs due to their superior generalization capabilities. As illustrated in the graph, the shallow MLP is slightly overfitting the training data. The blue curve (training accuracy) is a little bit higher than the green one (test accuracy), indicating that the model performs slightly better on the training data than on unseen data.

A similar pattern is observed with the loss. The loss for the shallow MLP on test data is higher than that for the training data at this phase, while for the deep MLP, they almost coincide.

CNNs are more effective. However, having a simple CNN architecture is not sufficient. It achieves an 80% accuracy on the test data and only 60% on the training data. The model may not have learned enough from the training data. Its simplicity hinders it from capturing the complexity of the training data, resulting in lower training accuracy. However, it performs better on the test data, potentially due to random factors. Therefore, a more complex architecture that can better grasp the features of the images is needed. To achieve this, an additional convolutional layer is added to learn hierarchical feature representations and gain translation invariance. Lower layers might learn simple features (like edges), and deeper layers combine these to detect more complex patterns. Moreover, due to the way convolutional layers process input (by sliding over the image), they are naturally more invariant to translation than fully connected layers, which lack these properties. As shown in the plot, the accuracy and loss curves of the Deep CNN for both training and test data coincide at this point in the training, thus achieving good final accuracy.

4 Conclusion

This report discusses the importance of image preprocessing and neural networks in handwritten equation recognition. Image preprocessing, particularly for symbol segmentation and prediction in convolutional neural networks (CNNs), is crucial. We tested methods like noise reduction, brightness adjustment, and histogram equalization to improve image clarity and contrast.

We developed two noise reduction techniques: a basic approach using thresholding and blurring (Version 1), and a more advanced approach with shadow removal, erosion, and adaptive thresh-

olding (Version 2). Version 2 proved more effective in reducing noise without losing key image details.

Brightness adjustment sometimes negatively affected the process by making the background more prominent than the text. Histogram equalization, while useful in low-contrast images, often amplified background features, hindering symbol detection.

Segmenting images into symbols for neural network input was critical. We used contour detection and two symbol extraction methods: Mean Shift Clustering and Contour Detection. Contour detection was more successful in isolating individual symbols.

For CNN processing, standardizing segmented symbols was necessary. This included centering and resizing symbols on a white background to a uniform size.

We tested various neural network architectures with a dataset of handwritten mathematical symbols. Deep CNNs outperformed shallow and deep multi-layer perceptrons (MLPs) due to better generalization and pattern recognition capabilities. The deep CNN achieved high accuracy and low loss in both training and test datasets, while both MLPs showed lower accuracy and higher loss.

In conclusion, advanced noise reduction, appropriate image preprocessing, segmentation, and deep CNNs are crucial for effective recognition and processing of handwritten mathematical symbols. The deep CNN demonstrated the best performance in accuracy and loss metrics.

References

[1] et al. abidrahmank et al. 2016. URL: https://opencv24-python-tutorials.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_morphological_ops/py_morphological_ops.html.

[2] "Find Contours function from OpenCV". URL: https://docs.opencv.org/3.4/d4/d73/tutorial_py_contours_begin.html.

[3] "Handwritten math symbol and digit dataset". URL: https://www.kaggle.com/datasets/clarencezhao/handwritten-math-symbol-dataset.

[4] "Histograms - 2: Histogram Equalization". URL: https://docs.opencv.org/4.x/d5/daf/tutorial_py_histogram_equalization.html.

[5] "Image Thresholding". URL: https://docs.opencv.org/3.4/d7/d4d/tutorial_py_thresholding.html.

[6] Dan Mašek. "Increase image brightness without overflow". URL: https://stackoverflow.com/questions/44047819/increase-image-brightness-without-overflow/44054699#44054699.