Converting Handwritten Mathematical Expressions into LaTeX Representations

Final Report

Part A: Final Report

Introduction

Handwritten Mathematical Expressions Recognition (HMER) is the field of converting handwritten mathematical expressions (Figure 1) to their LaTeX representation (Figure 2)–this is the goal of our project. This project is useful because scientific documents have been handwritten for centuries (Figure 3), but currently, the main source of communication is the Internet; it is therefore important to have a tool that can convert expressions into a compatible representation [1]. Converting expressions–such as the one seen in Figure 1–manually is tedious, error-prone, and time-consuming [2]. Machine learning can reduce the time and error associated with converting expressions manually. Machine learning is an appropriate tool because we need to classify several characters, which can be done by training a deep network with a large dataset of labelled characters. Further, OpenCV can be used for image segmentation to extract single characters.

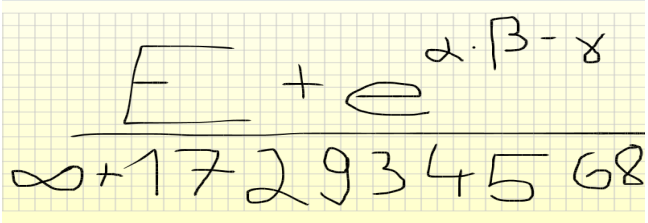


Figure 1: Handwritten mathematical expression.

```
\dfrac {E+e^{\alpha \beta -\gamma }} {\infty +172934568}
```

Figure 2: LaTeX form of the expression in Figure 1.

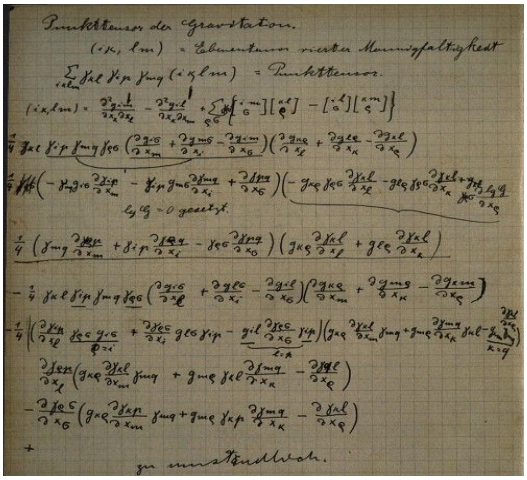


Figure 3: Mathematical document written by Einstein.

Illustration/Figure

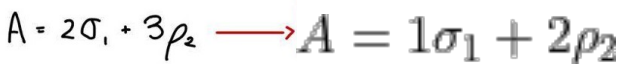


Figure 4: An example of a handwritten expression converted into LaTeX code.

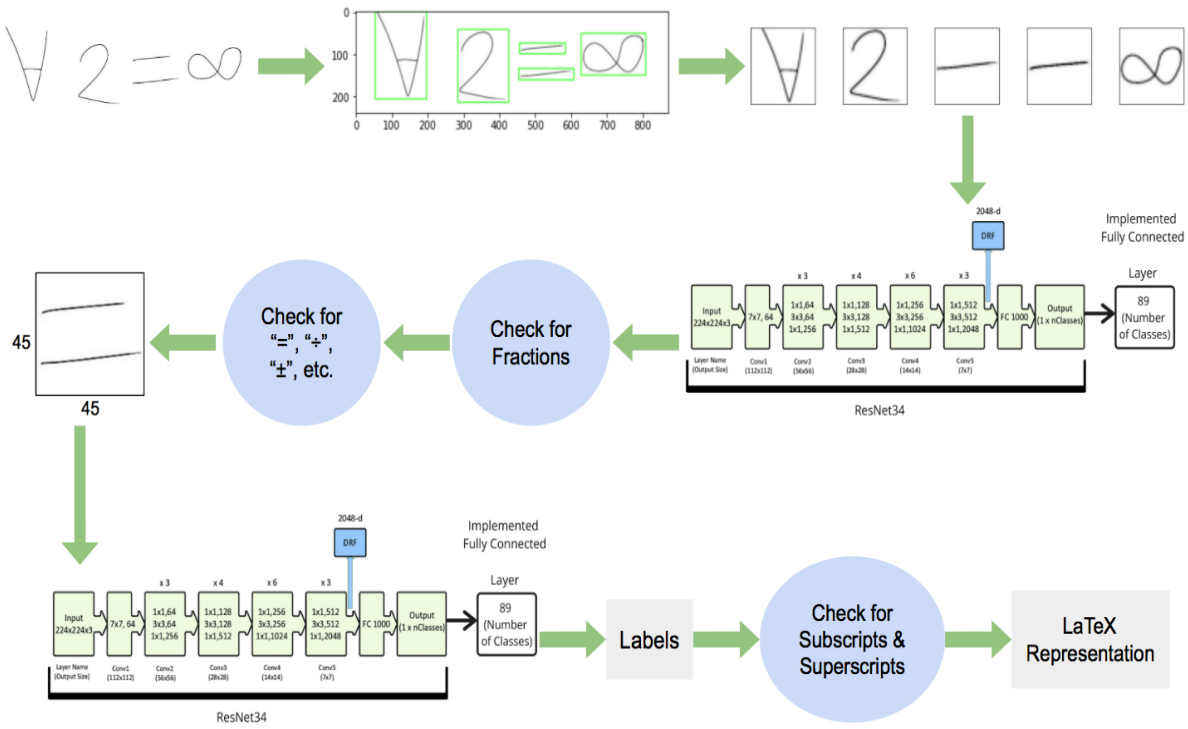


Figure 5: Full overview of model starting from handwritten image to final LaTeX representation.

Background & Related Work

Work 1: HMER Research Paper

The team achieved an accuracy of 94% on test/validation samples [3]. The dataset that this team used was the Handwritten Math Symbols dataset [5]. Below is the methodology the team followed:

Finding Contours–This team started by finding the contours of different numbers in the expression. The team used the OpenCV library–specifically cv2.findContours–to find individual character contours.

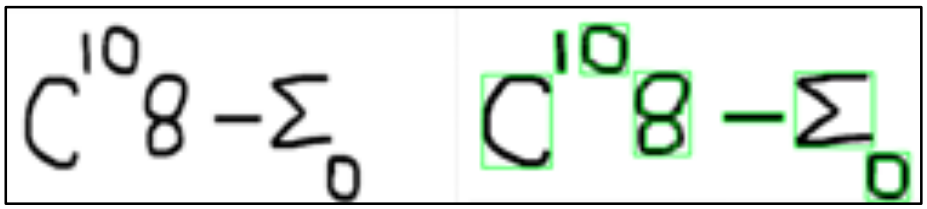


Figure 6: Original image versus the image with bounded boxes [3].

Processing Boxes–All the characters were cropped to 45 by 45 pixels to feed into the model for classification.

Model–The cropped images were inputted into a CNN model individually to classify each character. The architecture is shown in Figure 7.

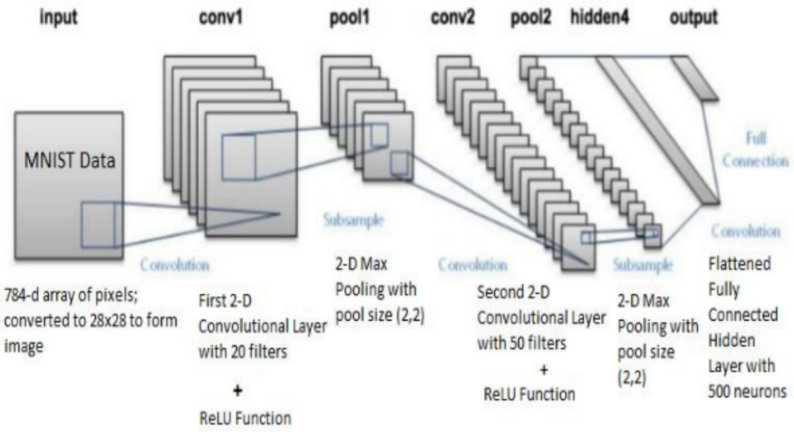


Figure 7: CNN architecture [3].

Latexification–In math expressions, characters can have many positions (e.g, subscript, superscript etc). The team used a recursive approach to determine the location of the next character, and saved the order of the characters in a spanning-tree structure. After all the characters were classified, they were converted into LaTeX.

Work 2: Detexify

Detexify is an online software that allows users to draw characters which are then converted to LaTeX format. This software is limited, as users can only classify a single character at a time.

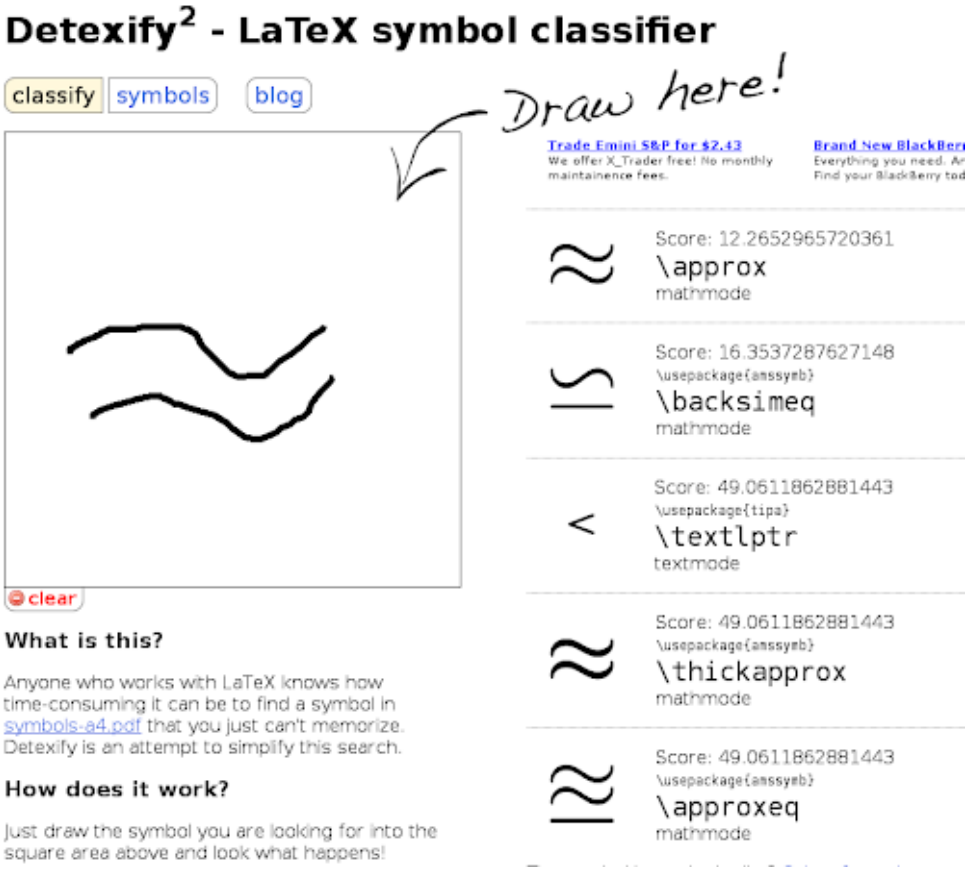


Figure 8: Screenshot of the Detexify application [4].

Data Processing

The 3 datasets that we used are: Handwritten Math Expression Dataset [3], English Alphabet Dataset [4] and Hasyv2 Dataset [5]. Below are the steps we took to clean and process the data.

Table of Data Processing Steps

Step #	Description
1.	The dataset that we used to obtain 50 of our classes is the Handwritten Math Expression Dataset. It had 101 classes but these classes were greatly unbalanced as seen from Figure 9.
2.	To balance the dataset, we parsed the Handwritten Math Expression Dataset and obtained 100 images of each class using a Python script to automate the process of deleting images. This automated process only took approximately 15 seconds per class. Also, we removed cos, sin, tan, log, and lim classes since the model will train on each letter separately, and hence it will detect each letter separately.
3.	The Handwritten Math Expression Dataset had its capital and lowercase English letters mixed together. Due to the inefficiency of separating them, the letter classes were deleted from this dataset. For letters, we utilized the English Alphabet Dataset from Kaggle.
4.	The letters from the English Alphabet Dataset needed to be inverted in color and resized from 36 by 36 pixels to 45 to 45 pixels as seen in Figure 10. To make processes more efficient, we wrote a Python script that automated the process of inverting colours and resizing for us. This automated process only took approximately 45 seconds per class.
5.	Some letters, such as *x*, *y* and *z*, have capital and lowercase letters that look extremely similar. In hopes of increasing the model's accuracy, we decided to eliminate the capital versions of these letters, and keep the lowercase versions.
6.	All the characters in the English Alphabet Dataset were really thick in font. To ensure that the thickness of characters did not affect the model's accuracy, we custom wrote our own dataset of thinner-lined characters. This custom dataset included 20 images for each letter (uppercase and lowercase). We resized these images using step 4.
7.	Some classes look extremely similar. Examples: lowercase "l" and ascii look similar to lowercase "i" (Alphabet Dataset doesn't have dots over any characters), "t" looks very similar to "+." "O" and "o" look very similar to the number 0. Therefore, we deleted "l", ascii, "o", "O", and "t" from our datasets.
8.	The *exists*, *in*, and *forall* symbols in the Handwritten Math Expression Dataset did not have enough samples; therefore, we used the Hasyv2 Dataset from Kaggle to obtain additional samples for these symbols. We used the automation script from step 4 to resize the images from Hasyv2 to 45 by 45 pixels. The distribution of our data is shown in Figure 11.
9.	We have split our data by using 80% for training, 10% for validation and 10% for testing.

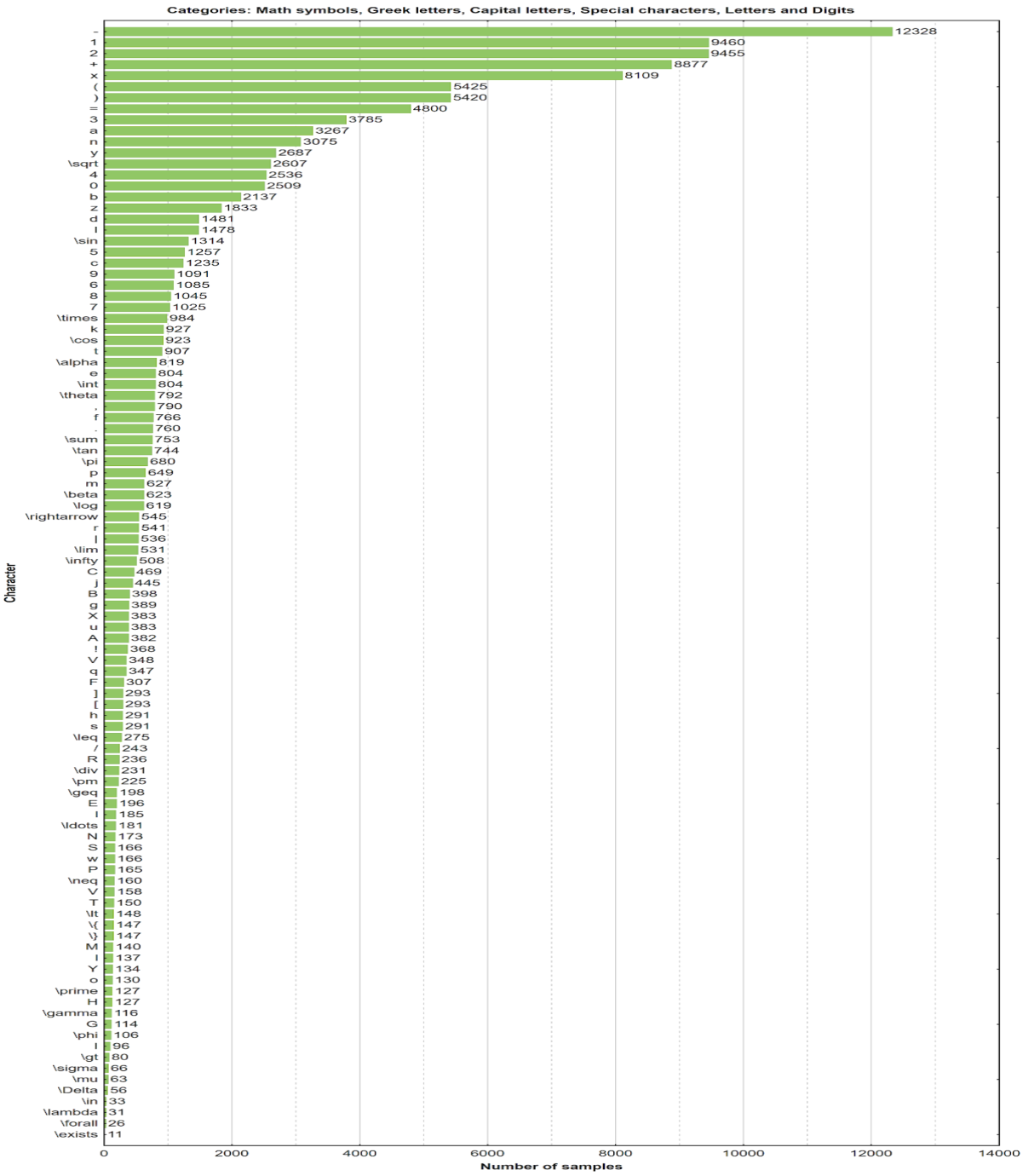


Figure 9: Distribution of classes in the Handwritten Math Expression Dataset [3].

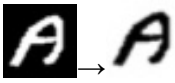


Figure 10: Picture on the left is the original image, and picture on the right is the "cleaned" image after being inverted and resized.

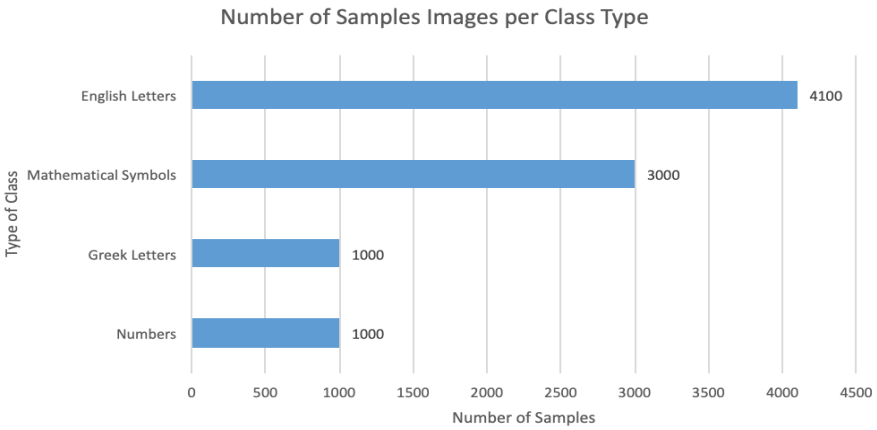


Figure 11: Distribution of *types* of classes in the final dataset. Each specific class contains 100 samples.

English letters: includes all letters (capital and lowercase) from A-Z, excluding l, t, o and O, with only lowercase versions for c, k, p, s, u, v, w, x, y and z (reasons explained in steps 5 and 7 of *Data Processing*).
Mathematical symbols: - ( ) ! [ ] { } + = ÷ ∫ ∈ / > < ≤ ≥ ∀ ∋ … ∞ ± x Σ ' ➡ √
Greek Letters: α β Δ λ μ ϕ π Σ θ
Numbers: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

Architecture
We used Transfer Learning for this project. Specifically, we used ResNet34, as it is one of the best models used for object classification. It is known for solving the vanishing gradients problem. We fine-tuned ResNet34 and re-trained it on our dataset to update the parameters. ResNet34 has 34 layers in total, with 5 types of convolutional layers, followed by a DRF and a fully-connected layer. We added the linear fully-connected layer to account for the 89 output features that we have. Through rigorous hyperparameter tuning, we decided on selecting the following hyperparameters: batch size of 1000, number of epochs to 30, weight decay of 0, and learning rate of 0.001. We utilized the Adam Optimizer, and kept the original ReLU activation function. A detailed visual of ResNet's architecture is shown in Figure 12.

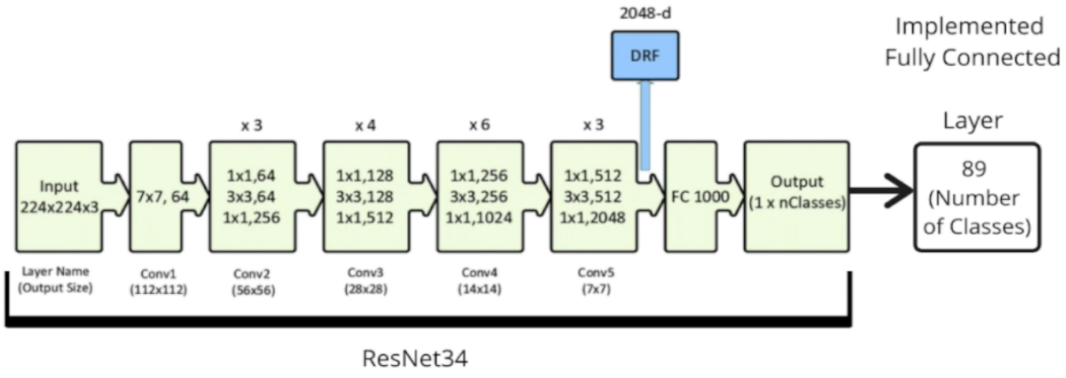


Figure 12: Detailed visual of ResNet34's architecture.

Challenging Examples:

Fractions

To determine whether or not a fraction is present, the program does the following: it determines the average y-value for the expression, and checks to see if a "-" symbol is near this average y-value. It then proceeds to check if the following characters are located above and below the "-" symbol. Upon satisfying all these conditions, the model will consider the expression in a fraction form.

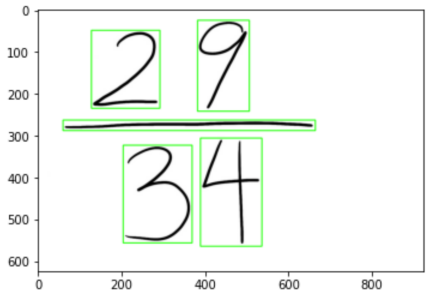


Figure 13: Visual of Fraction Detection.

Subscripts/Superscripts

The midline of the first character's bounding box is used as reference for all succeeding characters (Figure 14). If a succeeding character's entire bounding box is above the midline of the first character's, then the character will be distinguished as a superscript. If a succeeding character's bounding box is below the midline of the first character's, then the character will be distinguished as a subscript.

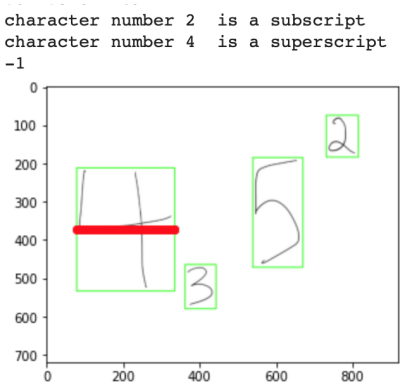


Figure 14: Visual of Subscript/Superscript Detection.

Special Operators

Samples: ≤ ≥ = ± ÷

Example: if the model classifies a character as a >, the bounding box increases to encapsulate the bottom dash (Figure 15). A new bounding box is cropped and fed into the model for reclassification. The same process applies to the rest of the samples above.

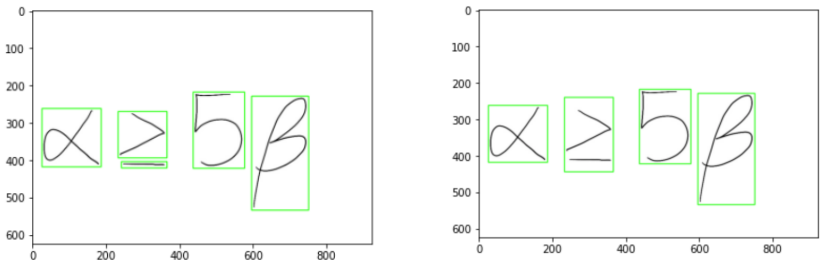


Figure 15: Visual of "Special" Operator Detection.

Trigonometric Operators

Samples: sin(x), tan(x), cos(x)

Example: if the model classifies s, i, and n next to each other, the model will re-label the expression as the appropriate trigonometric operator, in this case sin(x). The same process applies to tan(x) and cos(x).

Baseline Model

We used multi-classification logistic regression for our baseline model. To implement this, we used Scikit-learn–a Python machine learning library–and imported the logistic regression class. The model was trained/fitted using a sample of our training dataset, and validated using a sample of our validation dataset. This was a reasonable choice for a baseline model because our project calls for classifying handwritten mathematical symbols (e.g., digits and expression signs); logistic regression is commonly used for classifying handwritten numbers 0 to 9 in machine learning, as seen in [6,7]. Further, the baseline model is easy to implement, requires minimal tuning, and is quick to test.

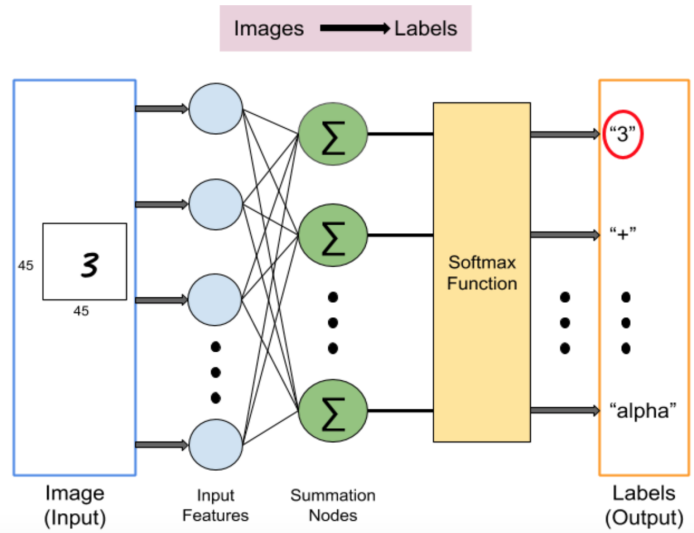


Figure 16: A schematic of the multi-classification logistic regression model. Input features are detected, scaled, summed, then passed through a softmax function to normalize the node output values. These values are then used to predict the output class [8]. Diagram was adapted from [8].

```
# score method gives accuracy of model
# accuracy here is: correct predictions / total number of data points
score = logistic_regression.score(x_test, y_test)
print(score)
```

```
0.3
```

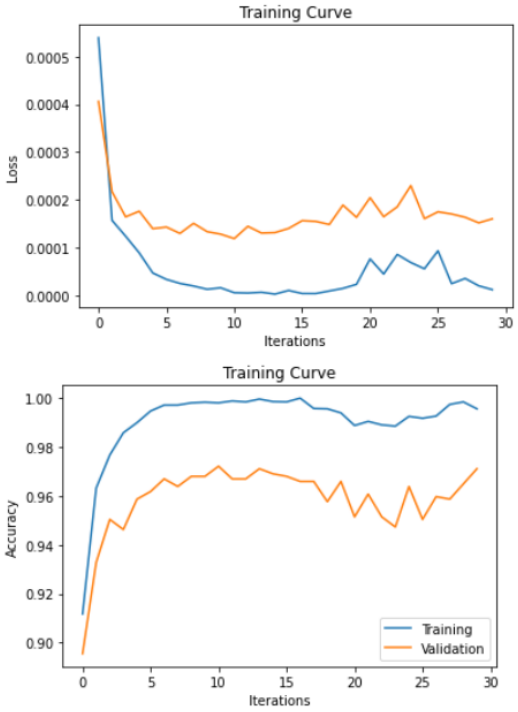
```
from sklearn import metrics
accuracy = metrics.accuracy_score(y_test, predictions)
print(accuracy)
```

```
0.3
```

Figure 17: The validation accuracies for the initial implementation of the baseline model.

Quantitative Results

The model that gave the best results is the Resnet architecture. The highest training accuracy achieved was 99%, validation accuracy is 97% and test accuracy of 95.7% (Figure 18).




```
Final Training Accuracy: 0.9952491011813046
Final Validation Accuracy: 0.9710444674250258
Total time:   857.01 s  Time per Epoch:  28.57 s
```

Figure 18: Validation and test accuracy/ loss graphs for best model.

We also calculated an equation accuracy, where we see how well the model performs on whole equations. Figure 19 shows the 10 equations we used to test our model. Equation accuracy is different compared to test accuracy, since we account for the positioning of elements (e.g., superscript, fractions, etc) as well as the classification of characters. The accuracy that we were able to achieve was 70%.

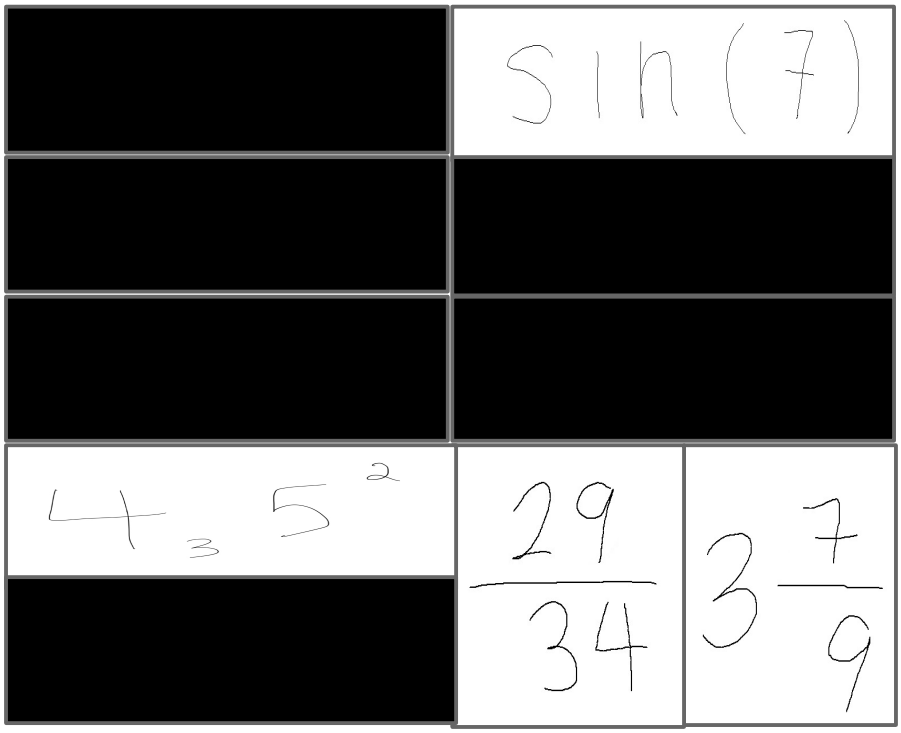


Figure 19: 10 test images used to calculate the equation accuracy.

Qualitative Results

Our model had a very high test accuracy but this did not translate to an equally high equation accuracy. One of the reasons for the difference in accuracies is because some characters looked very similar. This can be seen from Figure 20, which shows the percentages of the characters which the model guessed incorrectly.

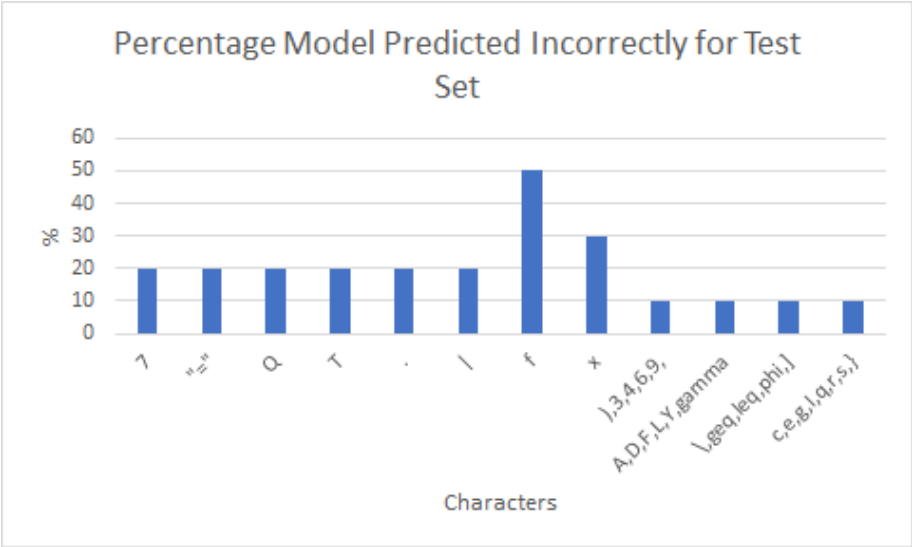


Figure 20: Graph of percentage of characters predicted incorrectly by model in test set.

The reason why the above characters were classified incorrectly is because they look similar to other characters as seen in Figure 21.

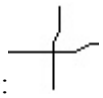
Label	Similar Looking Characters
f: 	+: 
x: 	Times: 
7: 	Greater Than: 
L: 	Less Than: 

Figure 21: Characters that look very similar to each other.

Another reason why the equation accuracy was significantly lower compared to the test accuracy was because of equations with "special" operators. For the equation in Figure 22, the model incorrectly classified the first part of the special character, as seen in Figure 23–the model should have classified the "L" as a "less than" sign. Therefore, the bounding box was not extended as seen in Figure 24. The results of the conversion are shown in Figure 25. This misclassification also resulted in the distortion of the superscript/subscript aspect of the equation.

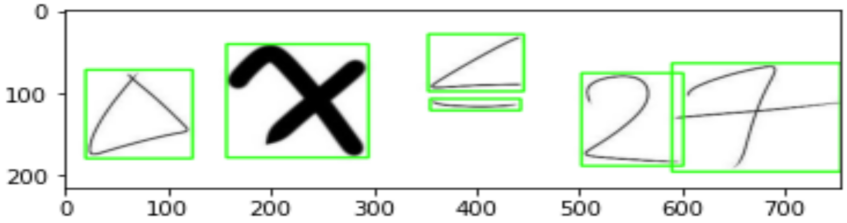


Figure 22: "Special" operator equation.

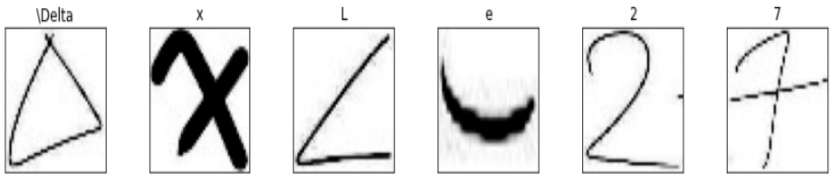


Figure 23: First part of the special operator ("less than" symbol) is predicted incorrectly as an "L".

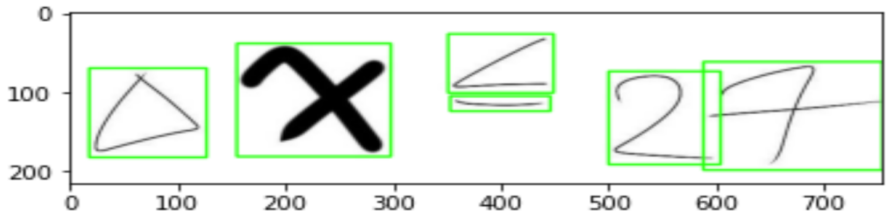


Figure 24: Bounding box does not increase due to the *less than* character being predicted incorrectly by the model.

$$\Delta x^{L} c27$$

Figure 25: "Special" operator equation in LaTeX format.

Evaluate Model on New Data

We obtained new samples that have not been examined or used to influence the tuning of hyperparameters by writing our own math expressions in our own handwriting. Effort was taken to ensure that our results were a good representation of our model's performance on new data. To do this, we evaluated our model's performance on more challenging data that we wrote, rather than simple samples, such as numbers (0 to 9) and elementary arithmetic operators (i.e., +, -, ÷, *) in one line. The challenging data included:

- Superscripts and subscripts.
- Fractions (denominators and numerators).
- Trigonometric equations.
- "Special" operators.

The model's performance met conversion expectations, and is shown in the figures below.

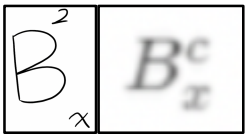


Figure 26: Superscripts and subscripts sample and the model's translation.

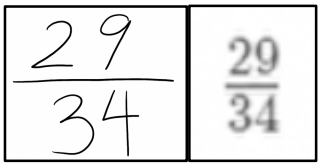


Figure 27: Fraction sample and the model's translation.

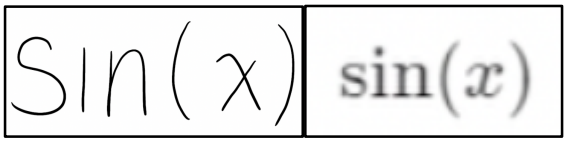


Figure 28: Trigonometric equation sample and the model's translation.

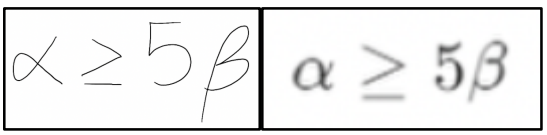


Figure 29: "Special" operators sample and the model's translation.

Discussion

The ResNet 34 model is highly accurate, with a test accuracy of around 95.7%, and a validation accuracy of 97%. However, these results do not properly reflect our model's true accuracy, since our accuracy of translating complete mathematical equations to the LaTeX formula is only about 70%. Although this is much less accurate than our ResNet model accuracy, we are still content with our results, and believe that our model is performing well.

There are multiple reasons for this decreased accuracy. Because we have 89 classes, there is a greater probability of the model picking the wrong class for a symbol. As mentioned in the Qualitative Results, many symbols also look very similar. We believe the most contributing factor of our lower equation accuracy is due to the fact that handwriting is very personalized, and our dataset unfortunately does not have enough variety in data samples to make the model more robust. In addition, due to inconsistencies in object detection, some images passed into the model may not closely resemble what the model was trained upon.

What was interesting was how sometimes, the model classifies a symbol completely incorrectly, and there is a lack of transparency of why this is. For example, sometimes the model classes a "-" symbol as a "i", even when we ensure all the data for "i" can be clearly identified by a human. From Lab 3, we believed that the model would be able to estimate new data relatively well, but this was not the case for our project. Generally, we thought that we would get a higher equation accuracy than 70% due to our high model accuracy, so this was surprising to our team.

Our team gained many insights about artificial intelligence that we will transfer to future projects. We learned a lot about object detection, and how to improve accuracy by adding data to the dataset and padding to images before sending them to the model. We gained a great deal of experience with Python, since we had to be familiar with sorting arrays, saving and loading data, expanding bounding boxes, visualizing the data, and using the location of various symbols. We also achieved much more hands-on experience with transfer learning. We were surprised to discover that there was a 20% increase in accuracy when fine tuning the model as opposed to using feature extraction.

Ethical Considerations

The model is limited by a training dataset of relatively neat handwriting examples. As such, if the handwritten characters are not fairly smooth or straight, the model may produce inaccurate results in LaTeX. This raises ethical issues in terms of accessibility, as a user with a writing disability may not see their desired results. In addition, the training data used are characters common in North American mathematical expressions. This means that the model is limited to recognizing only these symbols, thus, making the model inaccurate in labelling other characters.

Project Difficulty/Quality

Converting handwritten mathematical expressions into LaTeX representation was more challenging than expected. As mentioned above, this project's difficulty was due to complexity in data samples; in addition to identifying and classifying symbols present in an image, the model needed to understand the relationships that the symbols had with each other. For example, in the case of superscripts/subscripts, the model needed to store the positions of the symbols relative to each other. Such further logic/processing of symbols in images was also required for the cases of trigonometric equations, fractions, and "special" operators (e.g., greater than or equal to sign). To successfully implement the project, learning beyond the labs needed to take place when working with bounding boxes, the more challenging examples, and converting to LaTeX code. Ensuring that the model could

deal with more challenging expressions allowed the model to support a larger variety of mathematical expressions.

Works Cited

[1] A.-M. Awal, H. Mouchère, and C. Viard-Gaudin, "Towards Handwritten Mathematical Expression Recognition," *2009 10th International Conference on Document Analysis and Recognition*, Jul. 2009.

[2] A. Schechter, *Pdfs.semanticscholar.org*, 2017. [Online]. Available: https://pdfs.semanticscholar.org/fd25/9c0d70368a50b52755d4e33590566ef5d373.pdf?_ga=2.124156 04.1536964691.1591985627-78053362.1591985627. [Accessed: 12- July- 2020].

[3] X. Nano, "Handwritten math symbols dataset," *Kaggle*, 15-Jan-2017. [Online]. Available: https://www.kaggle.com/xainano/handwrittenmathsymbols. [Accessed: 12-Jul-2020].

[4] Mohneesh_Sreegirisetty, "English Alphabets," *Kaggle*, 18-Jul-2018. [Online]. Available: https://www.kaggle.com/mohneesh7/english-alphabets. [Accessed: 12-Jul-2020].

[5] Solo, "HASYV2," *Kaggle*, 10-Apr-2019. [Online]. Available: https://www.kaggle.com/guru001/hasyv2?fbclid=IwAR3001xLNRRzVPyb7xjYjWYDqzW_RT. [Accessed: 12-Jul-2020].

[6] M. Galarnyk, "Logistic Regression using Python (scikit-learn)," Medium, 29-Apr-2020. [Online]. Available: https://towardsdatascience.com/logistic-regression-using-python-sklearn-numpy-mnist-handwriting-recognition-matplotlib-a6b31e2b166a. [Accessed: 12-Jul-2020].

[7] mGalarnyk, "mGalarnyk/Python_Tutorials," GitHub, 03-Nov-2017. [Online]. Available: https://github.com/mGalarnyk/Python_Tutorials/blob/master/Sklearn/Logistic_Regression/LogisticRe gression_toy_digits_Codementor.ipynb. [Accessed: 12-Jul-2020].

[8] Roland, "Deep Learning Explained: Logistic Regression," *Microsoft Azure Notebooks*. [Online]. Available: https://notebooks.azure.com/rfernand/projects/edxdle/html/Lab2_LogisticRegression.ipynb. [Accessed: 12-Jul-2020].