CSCI 1430 Final Project Report: Image to LaTeX

LaTeX Heroes: Ian Gordon, Jaja Sothanaphan, Yutong Liu, Beatrice Hoang.

Abstract

In this report we propose a character-by-character convolutional neural network (CNN) model that translates images of math equations into LaTeX markup language. The input image goes through 4 steps: pre-processing, character separation, character identification using a CNN, and LaTeX assembly.

1. Introduction

We set out to create a model to convert handwritten Math equations/symbols into Latex formulas. Our model takes in an image that contains one or more handwritten equations on separate lines, with the constraint that each equation can be read purely from left to right– this does not include characters that contain a symbol above or below, such as fractions or summation notation.

2. Related Work

We looked at papers such as this Harvard NLP paper by Deng et al, this subsequent paper from Texas A&M by Wang et al., and some standard approaches to Object Character Recognition like those outlined in this series of articles.

Deng et al. [1] outline a data-driven approach, with a model that ”does not require any knowledge of the underlying markup language.” Their model first extracts image features using a CNN and arranges them on a grid. Then, an RNN is used to encode each row, and these encoded features are used by an RNN decoder.

Wang et al. [2] also outline a model with an encoder-decoder structure. Their model's encoder is a CNN that ”transforms images into a group of feature maps” and is trained in two steps– token-level training and sequence-level training.

3. Method

We decided to begin with a more modular apporach that implements standard OCR techniques and then modify the model for the particular challenges encountered in Latex. Our project structure was broken down into four main parts: (1) pre-processing the image, (2) identifying the individual characters, (3) training a convolutional neural network, and (4) assembling the latex string.

3.1. Pre-processing

The image received by the user is first gray-scaled and sent to a pre-processor that thresholds it and removes skew. The image is thesholded using adaptive gaussian thresholding, and the skew remover adjusts the image so that the lines are horizontal.

3.2. Finding characters using Bounding Boxes

After pre-processing the image is sent to the character finder, which first separates the multiple equations into their own lines. It does this by scanning along the y-axis and recording the start of a line when a black pixel is first recognized, and recording the end of the line when there are no more black pixels in the vertical scan.

Once the lines have been distinguished, the character finder then uses the same method horizontally on each line to distinguish each character. Now with these character locations stored, the last step is cropping each image so that the character fills up the image-space and then resizing each image onto a 28x28 pixel square. Here, the character images are sent to the CNN, and their locations are directly sent to the assembler.

3.3. Training a CNN

We used this handwritten math symbol dataset from Kaggle to train and test our CNN initially. This dataset composes of 16 common symbol classes. This model is in the model folder. Later on, we tried to move on using another handwritten math symbols dataset from Kaggle which contains 82 character classes. However, We did not have enough time and resources to find tune, train, and test our new CNN. This model is in the modelbig folder. The labels for each character are their Latex math strings. Our CNN's architecture is:

```
self.model = tf.keras.models.Sequential([
    Conv2D(32, 9, 1, padding="same",
        activation="relu", name="conv1"),
    MaxPool2D(3, 2, name="maxpool1"),
    Conv2D(64, 5, 1, padding="same",
        activation="relu", name="conv2"),
    MaxPool2D(2, 2, name="maxpool2"),
    Conv2D(128, 5, 1, padding="same",
        activation="relu", name="conv3"),
    MaxPool2D(2, 2, name="maxpool3"),
    Flatten(),
    Dense(100, name="dense1"),
    Dropout(0.3, name="dropout1"),
    Dense(50, name="dense2"),
    Dropout(0.3, name="dropout2"),
    Dense(num_classes,
        activation="softmax", name="dense3")
])
```

3.4. Assembling the Latex String

The assembler takes in the characters' latex labels from the CNN and their location information from the identifier to assemble the complete latex strings for each equation. For each line the assembler first orders the characters by their x-locations.

The assembler implements a naive approach for identifying superscripts. First, the assembler groups together adjacent characters with similar heights. From here to identify superscripts and subscripts, the assembler was implemented with a few assumptions: (1) the first character will not be a superscript or subscript, (2) every group above the previous normal group's middle y-value is a superscript, and (3) every group below the previous normal group's lower y-value is a subscript. After finding which groups are super-scripted and which are sub-scripted, the assembler adds carets and underscores where necessary.

Lastly, the assembler combines the latex string code for each equation and outputs them all to a text file and saves a rendered image of it.

4. Results

4.1. Bounding Boxes

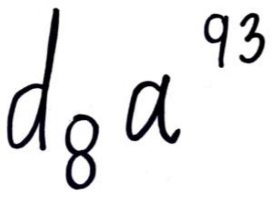


Figure 1. Example Image 1

Using the example image in Figure 1 as input, the bounding box was able to output each character resized to 28x28:

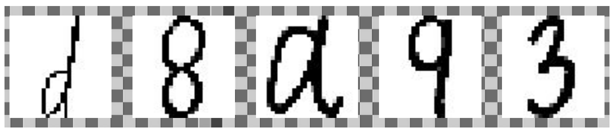
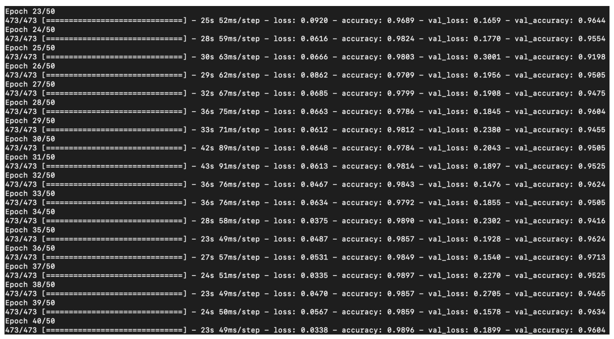


Figure 2. Caption

4.2. CNN



```
Epoch 23/50
473/473 [==============================] - 25s 52ms/step - loss: 0.0920 - accuracy: 0.9689 - val_loss: 0.1659 - val_accuracy: 0.9644
Epoch 24/50
473/473 [==============================] - 28s 59ms/step - loss: 0.0616 - accuracy: 0.9824 - val_loss: 0.1770 - val_accuracy: 0.9554
Epoch 25/50
473/473 [==============================] - 30s 63ms/step - loss: 0.0666 - accuracy: 0.9803 - val_loss: 0.3001 - val_accuracy: 0.9198
Epoch 26/50
473/473 [==============================] - 29s 62ms/step - loss: 0.0862 - accuracy: 0.9709 - val_loss: 0.1956 - val_accuracy: 0.9505
Epoch 27/50
473/473 [==============================] - 32s 67ms/step - loss: 0.0685 - accuracy: 0.9799 - val_loss: 0.1908 - val_accuracy: 0.9475
Epoch 28/50
473/473 [==============================] - 36s 75ms/step - loss: 0.0663 - accuracy: 0.9786 - val_loss: 0.1845 - val_accuracy: 0.9604
Epoch 29/50
473/473 [==============================] - 33s 71ms/step - loss: 0.0612 - accuracy: 0.9812 - val_loss: 0.2380 - val_accuracy: 0.9455
Epoch 30/50
473/473 [==============================] - 42s 89ms/step - loss: 0.0648 - accuracy: 0.9784 - val_loss: 0.2043 - val_accuracy: 0.9505
Epoch 31/50
473/473 [==============================] - 43s 91ms/step - loss: 0.0613 - accuracy: 0.9814 - val_loss: 0.1897 - val_accuracy: 0.9525
Epoch 32/50
473/473 [==============================] - 36s 76ms/step - loss: 0.0467 - accuracy: 0.9843 - val_loss: 0.1476 - val_accuracy: 0.9624
Epoch 33/50
473/473 [==============================] - 36s 76ms/step - loss: 0.0634 - accuracy: 0.9792 - val_loss: 0.1855 - val_accuracy: 0.9505
Epoch 34/50
473/473 [==============================] - 28s 58ms/step - loss: 0.0375 - accuracy: 0.9890 - val_loss: 0.2302 - val_accuracy: 0.9416
Epoch 35/50
473/473 [==============================] - 23s 49ms/step - loss: 0.0487 - accuracy: 0.9857 - val_loss: 0.1928 - val_accuracy: 0.9624
Epoch 36/50
473/473 [==============================] - 27s 57ms/step - loss: 0.0531 - accuracy: 0.9849 - val_loss: 0.1540 - val_accuracy: 0.9713
Epoch 37/50
473/473 [==============================] - 24s 51ms/step - loss: 0.0335 - accuracy: 0.9897 - val_loss: 0.2270 - val_accuracy: 0.9525
Epoch 38/50
473/473 [==============================] - 23s 49ms/step - loss: 0.0470 - accuracy: 0.9857 - val_loss: 0.2705 - val_accuracy: 0.9465
Epoch 39/50
473/473 [==============================] - 24s 50ms/step - loss: 0.0567 - accuracy: 0.9859 - val_loss: 0.1578 - val_accuracy: 0.9634
Epoch 40/50
473/473 [==============================] - 23s 49ms/step - loss: 0.0338 - accuracy: 0.9896 - val_loss: 0.1899 - val_accuracy: 0.9604
```

Figure 3. CNN Training Accuracy

As shown in Figure 3, our CNN was able to reach 97.13% accuracy, hovering around 93-96% after 20 epochs.

4.3. Assembler

Using the bounding-box location information found for Figure 1 and manual labels, the assembler was tested and able to produce the latex string

```
$d_{8}a^{9 3} \\ $
```

which renders as d_8a^{93}

4.4. Overall Results: All together

We were not able to extensively test our whole model on handwritten examples, but here is one example, testing using Figure 4 and flagging off the superscript/subscript calculator:

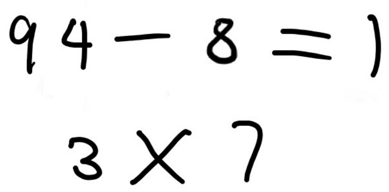


Figure 4. Example Image 2

that outputs latex string

```
$9 9 - 8 - 1  \\ 2 \times 7  \\ $
```

and renders as:
$99 - 8 - 1$
2×7

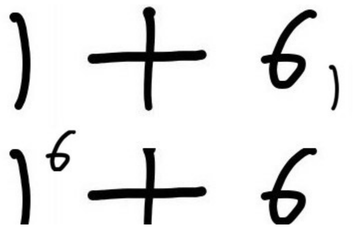


Figure 5. Example Image 3

Another example we tested on was Figure 5, which output latex string

```
$1 +61 \\ 9^{6}+ 6 \\ $
```

and renders as:
$1 + 61$
$9^6 + 6$

4.5. Technical Discussion

While our method works in identifying left-to-right equations, because our CNN only trains on individual characters our current model is unable to identify more complex equations with characters under grouping (such as a square root), fractions, and stacked notations such as summations and integrals. Ideally we would have implemented an RNN that would be able to process inputs of any length, but because of our time constraint we chose to implement a CNN from our knowledge on past projects.

Another downfall of this approach is that this requires a post-processing assembler that manually identifies special formatting, such as superscripts and subscripts, given the CNN label output and character location information. This requires each special formatting case to be outlined, so as the model becomes more complex so will the assembler's number of rules. The assembler currently has a naive implementation to identify superscripts and subscripts, but only works if the strict rules outlined by it are followed. There are many exceptions that could not be caught by this method previously outlined, for example if someone draws a still-small subscript but slightly above its previous number's median y-value. As seen in the overall results, the superscript in Figure 5 was not caught because of this.

Since the character identifier uses bounding boxes to find and separate characters next to each other it cannot distinguish overlapping characters. At present this method would also group a fraction as one character. One way we considered solving stacked-notation identification (like fractions, integrals, and summations) was to recursively run the character identifier again over each "character" until only one line is identified for each resulting character.

4.6. Social Discussion

Latex has become an important system to learn for students in Math and CS departments. Students that take notes in handwritten formats may find it difficult or simply time consuming to translate what they write into Latex. Students that want to follow Math equations in class may not know which Latex syntax to use or may . Our project could be useful for these students who struggle in translating Math formulas/symbols to Latex syntax.

Our algorithm recognizes handwritten characters. There has been a lot of research done on handwriting recognition. To transform a handwritten document into its digital form, as a part of AI application, handwriting recognition technology improves the efficiency of jobs, and at the meantime it reduces the need for human labor.

We would like to think that our product could help Latex learners or Latex users to identify Latex syntax they need in order to write their intended symbols/equations in Latex. Having such technology will greatly reduce time and energy spent on searching for the correct Latex syntax. Additionally, while Latex might look professional, writing latex notes is not practical for large equations, and our tool would allow note-takers to quickly scribble and store equations in their digital notes.

Also, for users to submit their handwritten document to software and get it processed to be recognize, there is a legit concern of leaking personal information. Will the software storage the personal information on the handwritten documents? Will the private handwritten document be leaked to the public? A math document can be important or classified, such as Nuclear Launch codes. For a handwritten recognition software to be implement by all types of users, it need to protect the user's information.

We would like to think that our idea is relatively benign. But if our software were used on important or classified equations, we would want to make sure that our software is secure and does not store user information. Additionally, the software need to be secure in way that someone could not simply break our software. However, these threats need to be handle within the app or program implementing our tool as well as to reduce the vulnerability with the tool itself.

Interestingly enough, mathematical symbols are largely consistent across most of the globe, albeit with a few variations. With any text-processing system, it is important to consider the variety of languages and scripts across cultures and countries. In mathematics however, there is a relative consistency in using Arabic numbers with the Latin and Greek alphabets. However, here are a few discrepancies that we found:

- In Israeli primary schools, the + plus sign is sometimes written as a small $\perp$ (an inverted T; U+FB29). This is a Jewish tradition that dates from at least the 19th century; the usual explanation is that it avoids writing a symbol that looks like a Christian cross.
- In Arabic schools, algebra is sometimes presented

using Eastern Arabic numerals and right-to-left variants of common symbols: see for example Arabic mathematical symbols in Unicode.

- In Japan, Taiwan and Korea, the approximately equal symbol is usually written $\fallingdotseq$ rather than $\approx$.
- The use of both A (a) and α (alpha) in the same equation could be problematic in Greece, since α in Modern Greek is often not distinguished from A.
- abbreviations are sometimes different: for example, Russia and some other countries use tg and ctg rather than tan and cot.

In mathematics, even though the majority of collected data is applied in western countries, there is no system or process that is used to collect the data biased against any groups in our knowledge. The main reason could be that there is a relative consistency in using Arabic numbers with the Latin and Greek alphabets across cultures and countries, and the discrepancies is not obvious.

We used datasets from Kaggle. Kaggle supports a variety of datasets. With public datasets, it is hard to determinate how the data is collected and whether the data is used in a manner agreed to by the individuals who provided the data. However, in our project, the dataset is trained to recognize other user's handwritten characters. Any personal information may come with the handwritten character is not be used in our project.

In handwritten character recognition, computer vision relies on training on large datasets of handwritten text that is collected from a broad group. If privacy laws changes, such as how public datasets could be posted and implemented, it have a huge affect on our project. Also, if someone's handwritten text is considered as a privacy, it could be difficult to collect data and test our recognition to a broad range of users.

5. Conclusion

We create a model to convert handwritten Math equations/symbols into Latex formulas. Latex has become an important system for professional writing. HoweverLatex math syntax could be confusing for beginners. For anyone who has no knowledge of Latex syntax but want to present their math equations in a professional formula, our model would be helpful. A user can hand written the math equations on a paper, then the model can output the Latex equivalent. The model can make Latex, a professional system with its own syntax, easier to implement for naive user.

References

[1] Yuntian Deng, Anssi Kanervisto, and Alexander M. Rush. What you get is what you see: A visual markup decompiler, 2016. 1

[2] Jyh-Charn Liu Zelun Wang. Translating math formula images to latex sequences using deep neural networks with sequence-level training, 2019. 1

Appendix

Team contributions

Ian Gordon I did most of the research to find the tools and techniques used in this project. After consolidating the datasets used to train our character recognition neural network, Jaja and I worked together to build the CNN. I got the model up-and-running with our datasets and model-saving capabilities, and she worked to restructure and optimize the model. I also set up the image-processing class and adjusted the parameters on those functions for speed and accuracy as well as setting up the live view of our project with the webcam.

Jaja Sothanaphan I worked on constructing CNN, the model, and connecting all separate parts we each worked on to function in a single flow. I trained and tested with different configurations, and I decided on Adam optimizer and sparse categorical cross entropy as a loss function with learning rate of $1e^{-4}$, to avoid reaching local maximum (val_accuracy = 0.0792). Additionally, I trained the model on a small dataset of 16 classes and a larger dataset of 82 classes. In addition, I worked on connecting other parts for the end-to-end workflow, general debugging for the entire flow, fine tuning the model, and creating constant.py.

Yutong Liu I worked on finding each input characters using bounding boxes. Two method are tested, I decided to use the histogram separation that is implemented in ORC because it has better result with simpler network. With a prepossessed image, I used horizontal histograms to separate each line and vertical histograms to separate each characters in that lines. Then by combining this two method, I found the each character in the image. I also worked with other team members to connect the code.

Beatrice Hoang I worked on the assembler and character finder. Working with Yutong, we established the parameters between the finder and assembler. I implemented the main assembler function that called the grouping function, sub-scripting/super-scripting function, and display. For the character finder I wrote the final_characters function that calls Yutong's functions to help consolidate the parameters and return the character images and locations.