CS230

TexNet: The First Stage of a Handwriting to LaTeX Pipeline

Zen A. Simone
Stanford University
zsimone@stanford.edu

Cameron Cruz
Stanford University
camcruz@stanford.edu

Kofi Adu
Stanford University
kadu25@stanford.edu

Abstract

Converting handwritten equations into an electronic format like LaTeXis a relatively simple, yet time consuming, process for humans, so we propose that this task can be accomplished by implementing a pipeline to recognize and interpret mathematical symbols. In this paper, we describe the process of preprocessing and segmenting images for recognition using a custom CNN architecture, and structuring data into a format that can be processed more at a later time. We find that our model trained on Xai Nano's "Handwritten Math Symbols" dataset achieves 99% accuracy on individual characters in our test distribution.

1 Introduction

As students in STEM we are tasked with frequently writing up problem sets in LaTeX. It is often easier and more natural to compose these by hand instead of performing the painstaking process of transcribing the math into LaTeX. This isn't a mentally taxing or difficult task, however it takes more time to type out multiple formatting commands than drawing the expressions on paper. Thus it would be ideal to automate with machine learning.

We envision a three-step pipeline for this task: first, a segmentation task: a CNN would separate out the parts of an image that correspond to individual equations from a problem set page. Next, a detection task: a second CNN would recognize the symbols present in the equation and identify their position data. Lastly, a machine translation task: an RNN would take this character and position data and determine their relationship to each other, constructing the corresponding LaTeXexpression. We decided it was most logical to start with the second step, as it would be difficult to build the RNN without predictable input from the second step, and we see little utility from the first step without being able to derive meaning from the equations themselves.

The input to this second step of the pipeline is an image of a handwritten equation (we assume we are able to find equations in a page already). This image is preprocessed and segmented into individual characters, finding bounding boxes for each. Crops of these characters are then passed one at a time into our classification CNN, which outputs a predicted math symbol. Once we've done this for each character, we create a list of lists containing character labels and bounding box positions which we return in JSON format. This will be passed into the eventual RNN to construct the equivalent expression in LaTeX.

CS230: Deep Learning, Spring 2018, Stanford University, CA. (LateX template borrowed from NIPS 2017.)

2 Related work

There are essentially three camps of approaches when it comes to converting handwritten mathematical expressions into LaTeX. The first is apps that focus solely on single character recognition like Detexify and MDig which only recognizes digits. [8]

MDig uses very similar methods to us, particularly for preprocessing the data. Instead of using YOLO (You Only Look Once), we both use segmentation to define bounding boxes around the symbols. However, one major difference between our approaches is the fact that in an effort to reduce the computational power that the mobile device must do, MDig has a very shallow CNN network. On the other hand we use a 13 layer network to identify and classify objects. Even though MDig uses a shallow network they are able to achieve good error rates. This probably due to the fact that their network is only recognizing 10 classes/digits as opposed to 82 classes.

The second approach people are attempting to use is an end to end learning approach. This causes accuracy to suffer because the task is too complex for a CNN to resolve. [9]. In general, humans naturally do the task in a two-step process: identifying what the components of the equations are and then converting the pieces to LaTeX. Which takes us to the next group of approaches which separate the pipeline into detection, classification, then structural analysis steps. [6]

Offline Chinese character recognition is one example of this that has been applied to many areas, including transcription of handwritten notes, so it may be possible for us to utilize transfer learning somewhere down the line to help our algorithm. The paper highlights an enhancement of the typical CNN by adding network pruning using adaptive drop-weight technique to reduce redundancies in the CNN which is something we did not think about in the creation of our own CNN. Considering our net works pretty well for the data we have tested it on, if we do get more data and our accuracy goes down, it would be interesting to see how this algorithm affects our application. However, even though they reported high accuracy and speed numbers, the algorithm looks very complex relative to our task. Perhaps the analysis of Chinese characters requires this level of robustness due to the complexity of the language, but it seems unnecessary for the task we are focusing on.

3 Dataset and Features

Our dataset came from Xai Nano's Handwritten Math Symbol Dataset from Kaggle. We had 147234 training examples for 82 classes, which we perform cross validation on, and 37161 test examples. Each example has a resolution of 32 x 32. Given a training image, we preprocess the image by first converting the image to greyscale. Next we apply a Gaussian blur to smooth out the image and reduce noise. Then we use Otsu thresholding to binarize the image converting all the pixels above the threshold to white and everything else to black and we invert the colors. This creates a white expression/equation on a black background.

Here are some examples from our training set (after preprocessing):

Figure 1: 'beta'

Figure 2: '7'

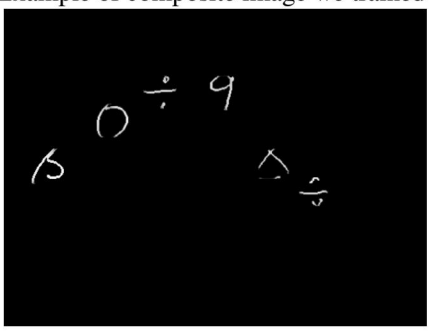
Figure 3: 'div'

4 Methods

Since we wanted to obtain positional information in addition to recognizing math symbols, we initially thought YOLO would be a great approach to this task.

To use YOLO, we would superimpose images in our dataset onto a 500×375 black canvas, saving the min/max x-y coordinates to an annotation file in the Pascal VOC format.

Figure 4: Example of composite image we trained YOLO on



We used darkflow, an open-source Tensorflow implementation of darknet, to train a model on our dataset. We tried both retraining Tiny YOLOv2 weights and training the model from scratch. However, due to poor performance, we had to design a new architecture. We discuss the issues we encountered with YOLO later in this paper.

We decided that a better approach would be to split the task into two stages: segmentation and classification. During the segmentation step, we would find bounding boxes for each character in the image, then use those boxes to create image crops containing individual characters. These crops would then be sent one at a time through our classifier to determine the correct label.

We first preprocess real-world images using similar thresholding techniques. However, we also perform background equalization and a Hough transform to reduce the paper texture and lines on ruled paper, since that could create extra unwanted edges that get thresholded.

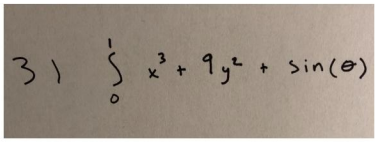


Figure 5: 'Before preprocessing'

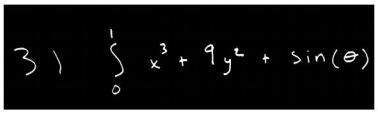


Figure 6: 'After preprocessing'

For segmentation, we utilized OpenCV's contour function to detect and bound the contours of the individual symbols in our expression. If there is any overlap in the bounding boxes we assume the algorithm is attempting to identify the same item so we get rid of the smaller box and expand the bigger bounding box to capture the entire symbol. We extract these bounding boxes from of our original image and feed it into our classification algorithm.

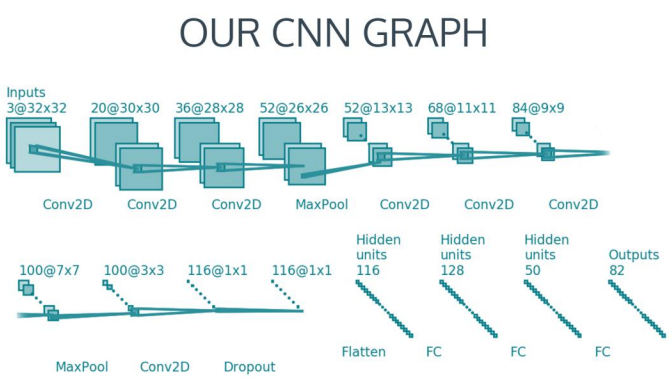
For classification, we built a 13 layer convolutional neural network (shown below) using Keras. We trained our CNN on the binarized 32×32 images from the Kaggle dataset. We have a total of 7 convolutional (conv.) layers which use filters to learn differentiating features between all the classes and 2 max pooling layers, interspersed every 3 conv. layers, that downsample the data extracted by the convolutional layers. This is followed by 3 dense layers that learn the classification function for the downsampled features from the earlier part of the network.

We measure our error using the binary cross entropy loss function: - $\sum_i y_i \log \hat{y}_i$. The loss function works by measuring the difference between predicted probabilities $\hat{y}$ and ground-truth probabilities y and minimizing the negative log likelihood of our prediction. We also tried categorical cross entropy, but found it gave us poorer results.

5 Experiments/Results/Discussion

We tried retraining Tiny YOLOv2 weights using multiple combinations of hyperparameters, with learning rates from 0.001 to 0.00001, batch sizes from 16 to 64. In all cases, the model struggled to detect images at all, providing very few bounding box predictions. Training YOLO from scratch performed marginally better, outputting more bounding box predictions. However, the class prediction

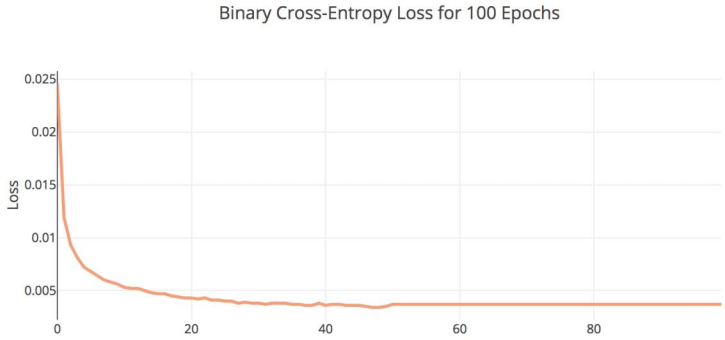
Figure 7:



for each bounding box was wildly inaccurate. After a great deal of experimentation and research, we believe that the resolution of the characters in images wasn't high enough for YOLO since YOLO struggles to detect small objects.

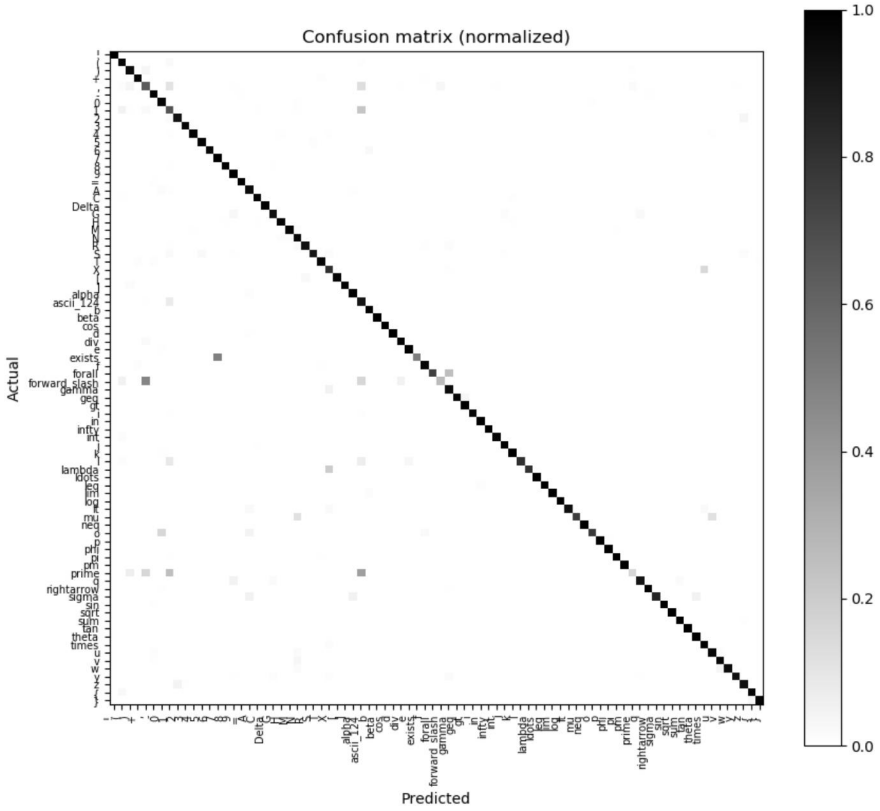
Building a segmentation algorithm partnered with our own classification algorithm resulted in significantly better performance. We chose a learning rate of 0.001, which was large enough such that our model converged relatively quickly, but not so large that it overshot the local minimum. We chose a relatively small batch size of 64 so that our model would generalize better while still taking sufficient advantage of computational resources for training speed. The model converged after around 60 - 80 epochs with a loss of around 0.0043. A full graph of the loss is provided below:

Figure 8:



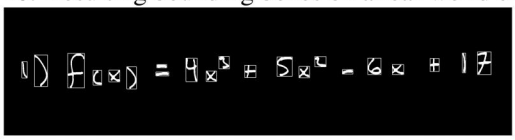
Our model achieved 0.9988 accuracy on the training set and 0.9985 accuracy on the test set. Since we had similar accuracy on both the training and test distributions, we do not seem to have overfit to the training data. When we used categorical cross entropy as the loss function, the model performed worse, with the loss converging at around 0.185 and yielding a training accuracy 0.947 and test accuracy 0.936. Thus, we stuck with the model trained using binary cross entropy. Here is a normalized confusion matrix for each class when running that model on the test set:

Figure 9:



We created some real world examples to test the full segmentation/classification process on. The following example was taken using an iPhone 8 and run through our network:

Figure 10: Resulting bounding boxes on a real-world example



The classes predicted for this image are as follows: ’1’, ‘)’, ‘f’, ‘(’, ‘x’, ‘)’, ‘=’, ‘=’, ‘4’, ‘x’, ‘y’, ‘+’, ‘5’, ‘x’, ‘i’, ‘=’, ‘6’, ‘x’, ‘+’, ‘|’, ‘7’. The model has fairly good performance, but has more difficulty classifying superscript. Additionally, we need to account for symbols with disconnected strokes in the segmentation step, like =.

6 Conclusion/Future Work

Overall, we sought to tackle the task of converting handwritten mathematical equations to LaTeXby separating it into two main tasks: segmenting and classifying. Given an image with an equation our model preprocesses the image by denoising and binarizing the image. Then we segment the equation into individual characters with position data and classify the characters with 99% accuracy. We are pleased with these results given the time period. However, the long term objective is to be able to build an application that takes in an image of a sheet of homework with embedded equations and transform the entire page into LaTeX. As such, we plan on continuing the project over the coming months to complete the pipeline. The next step is building an RNN that can take the structured data we obtain from our segmentation/classification step and output the corresponding LaTeXexpression.

7 Contributions

Cameron Cruz: Researched the task and possible approaches, build data generation and preprocessing scripts, handled devops, designed the poster.
Zen Simone: Researched the task and possible approaches, built segmentation script and Keras CNN, built prediction script for use on real-world images.
Kofi Adu: Researched the task and possible approaches, tested the YOLO and custom CNN implementations of the task, wrote most of the final report.

References

[1] Amit Schechter, Norah Borus, William Bakst. (2017) *Converting Handwritten Mathematical Expressions in LaTeX*

[2] Catherine Lu, Karanveer Mohan. (2015) *Recognition of Online Handwritten Mathematical Expressions Using Convolutional Neural Networks*

[3] Chang, Joseph, Gupta, Shrey, Zhang, Andrew. (2016) *Painfree LaTeX with Optical Character Recognition and Machine Learning*

[4] Lisa Yan. (2016)*Converting Handwritten Mathematical Expressions in LaTeX*

[4] thtrieu, *thtrieu/darkflow* [Online]. Available: https://github.com/thtrieu/darkflow.

[5] Xuan Yang, Jing Pu. (2015) *MDig: Multi-digit Recognition using Convolutional Nerual Network on Mobile*

[6] Xuefeng Xiaoa, Lianwen Jina, Yafeng Yanga, Weixin Yanga, Jun Sunb, Tianhai Changa. (2017) *Building Fast and Compact Convolutional Neural Networks for Offline Handwritten Chinese Character Recognition*