Painfree LaTeX with Optical Character Recognition and Machine Learning

Chang, Joseph `chang100`
Gupta, Shrey `shreyg19`
Zhang, Andrew `azhang97`

December 17, 2016

1 Introduction

Recent years have seen an increasing interest in harnessing advancements machine learning (ML) and optical character recognition (OCR) to convert physical and handwritten documents into digital versions. The increasing adoption of digital documents in academia, however, has provided a new layer of complexity to automatic digitization of physical documents. Compared to typical texts written in natural language, academic papers often contain many elements like equations that are notably tougher to recognize with many current OCR techniques. Furthermore, creating digital documents from scratch tends to be difficult too: while typesetting systems like LaTeX help streamline equation formatting and document generation, the process of typesetting complex equations is a arduous and error-prone process. Ultimately, academics aspire to find a solution where they can get the simplicity of generating documents by handwriting while still obtaining the convenience of using digital documents. In this paper, we aim to explore various techniques for converting images of mathematical equations into LaTeX source code.

2 Related Work

The problem of generating LaTeX from equations is nothing new, with solutions focusing on various different aspects and subsets of the problem. For example, the popular tool DeTeXify [9] is used to identify LaTeX code for specific handwritten characters, while Mathbrush allows conversion of entire images [8]. As mentioned by Kam-Fai Chan, current solutions involve either using online data in which stroke order of handwriting is available, or offline data, where the input is simply the rendered image [5]. In either form, the main problem reduces into a few specific subtasks (we will provide more details about existing methods in sections below):

- *Character Segmentation (CSeg)*: Segmenting an equation image into images of individual mathematical characters
- *Character Recognition (OCR)*: Identifying individual mathematical characters.
- *Structural Analysis (SA)*: Clustering semantically related symbols and forming an expression tree to represent the equation.
- *Lexing and Parsing*: Converting an expressing tree to LaTeX source.

3 Project Scope and Methods Overview

While the problem of converting equation images to LaTeX is fascinating, it is also an incredibly large problem with wide scope. For this project, we will focus on working with already typeset LaTeX equations as inputs and outputting the LaTeX markup to generate it. To focus the scope of this project, we will not work with handwritten equations, and we will develop based on a specific subset of LaTeX expressions that include commonly used symbols as found in the InftyCDB-3B [2] dataset. These simplifications allow us to focus on three of the interesting tasks at hand, CSeg, OCR, and SA (see Related Work). (We don't mention lexing and parsing here, as this is less interesting from an OCR or ML standpoint, so we omit the details.)

4 Datasets and Tools

For this project, we use two main training sets. The first dataset, the *Wikipedia Equation Dataset*, contains 6,111 typeset equations extracted by spidering the "List of Equations" article on *Wikipedia* using a custom Python script. The dataset provides a variety of equations used across various disciplines. Each training example is an SVG of the equation, and is annotated with its ground-truth LaTeX source. When using this data, we preprocessed the images by converting the SVG into a PNG image, applying a binarization filter with brightness threshold 0.1, removing any alpha channel transparency, and cropping the image by removing padding. This dataset was used for development of the entire pipeline.

We also used a secondary dataset for training the OCR module specifically. This dataset, the *InftyCDB-3B* dataset [2], contains 70,637 PNG images of 275 unique individual characters of roughly size 36 by 48 pixels. In order to preprocess these images, we binarize each image with brightness threshold 0.1 and resize the images to 20 by 15 pixels. This resizing is done by converting the image into an array, breaking the original image into 300 blocks of roughly equal size and setting the value for the block to the number of black pixels present in the block.

For this project, we also make extensive use of Python libraries for image manipulation, matrix operations, and classification, namely numpy [3] and scikit-learn [4]

5 Character Segmentation

5.1 K-means

Our initial experimentation with K-means for segmenting revealed many issues that would have to be addressed by our segmentation strategy. For some simple inputs with characters far from one another, it works quite well. However, we must provide the number of centroids, which is unfeasible for larger datasets. With k-means, some parts of characters leak into other clusters, and small characters are not clustered on properly.

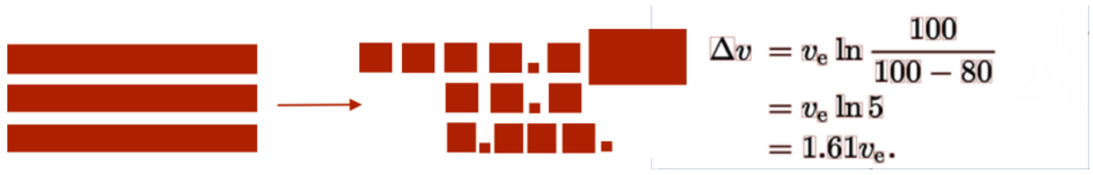
5.2 Floodfill

We can instead extract connected pieces individually from the image, by picking an arbitrary point and floodfilling points connected to it. This ensures each set of connected points is always grouped together properly.

A new challenge that arises is combining pieces of a character with separated parts. However, doing so essentially requires comparing the relative spatial positioning of every pair of pieces to a known set of split characters. Furthermore, afterwards we also have to establish spatial relationships between each character, and then infer a more detailed structure from them, which proves extremely difficult.

5.3 Recursive Projection Profile Cuttings

Ultimately, we settled with Recursive Projection Profile Cuttings (RPPC), a method that is perfectly suited for our segmentation purposes. [10]



While we need to segment each equation we are given into characters, our ultimate goal is to not only have pieces that can be classified into symbols by OCR, but also to have a structural mapping of all the characters.

RPPC takes an image and erodes it vertically or horizontally, to split it into multiple smaller pieces. However, when we do this, we know how the individual sub-pieces are spatially related. We can recursively do this on each smaller piece to create a structural mapping of the individual symbols.

Doing so in this way creates a mapping of each piece of the equation to its internal structures, which comprises of smaller expressions. The characters are identified through OCR and can be used to . For example, a '-' sign as the middle of 3 vertically separated expressions is actually a division.

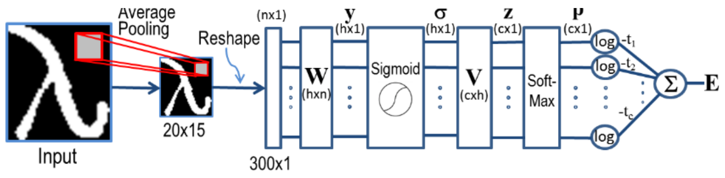
An additional problem arises from skewed (italicized) characters and external symbols (such as square roots). We learned a range of slopes to check for a modified horizontal separation, which supports a combination of serif and italics that breaks the usual algorithm for horizontal separation. We also support square roots (and similar characters) by extracting characters that span the entire dimension of the subexpression.

6 Character Recognition

6.1 Neural Network

Once the mathematical context image is segmented into individual symbol images, the next step is to classify these segmented symbol images into the actual symbols they represent. The candidate classes will be the possible LaTeX symbols.

Inspired by the success of LeNet-5, a multi-level convolutional neural network for recognizing handwritten numerical digits by Yann LeCun [1], we first implement a simple version of Convolutional Neural Network (CNN) from scratch. The architecture of our simple CNN is shown in the diagram below.

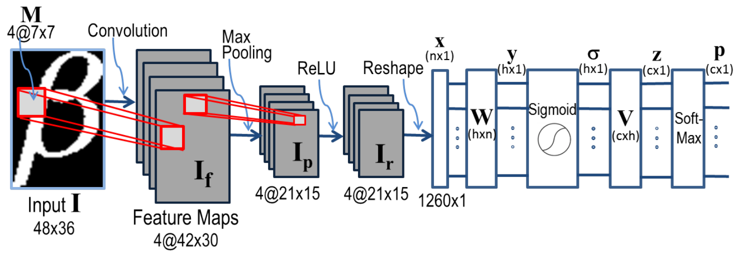


$$\nabla W\, E = \nabla y \qquad E \cdot xT + \lambda\, W$$

We utilize images from the InftyCDB-3B database (which are variably sized) as training and test data for our Neural Net. To overcome this variable-sized input issue, we use an average pooling layer as the first stage in our simple CNN. This average pooling layer adapts the variable-sized images into fixed-sized images by dividing an original image into rectangular pooling regions and computing the average of each region. We experimented a few fixed sizes and chose 20x15 as the fixed size for subsequent steps. The output images of the pooling layer are reshaped into 300x1 vectors x and fed into the input layer of the neural network. Thus, there are 300 input units. Because we are trying to classify the image as one of 275 different symbols, there are 275 output units which correspond to the probabilities that the image is one specific character. Four our hidden layer, we have 288 units, the approximate mean of the number of input and output units.

6.2 Convolution Neural Network

Inspired by class notes notes of CS231N, we investigate if we can further improve our result using the architecture [INPUT - CONV - RELU - POOL - FC], where CONV denotes a layer of feature-extracting convoluters, RELU denotes Rectified Linear Unit, POOL denotes maximum pooling, and FC denotes fully-connected neural network. The architecture of the CNN is shown below.



6.3 Support Vector Machine

We also applied a Support Vector Machine classifer to the training data. Previous papers have found success with SVM for mathematical expressions [6]; we aimed to try using modifications on previously used feature sets. Our feature extractor extracted two types of features: 1) density based features based on how many black pixels were present in a block after resizing and 2) directional, shape based features [6] where we consider each block on the contour and set 4 features: 1 for the horizontal, 1 for the vertical axis, and two for the diagonal axes, and give a value of 1 when the next pixel on the contour is in a given direction of an axis. These features were then applied to the RBF, linear, and poly kernels with C=1.2.

7 Structural Analysis

The final step is Structural Analysis, which involves using the data provided by the CSeg and OCR modules to build an expression tree for the equation. This involves two key forms of analysis: 1) Layout-based analysis and 2) Grammar-based analysis.

Layout-based analysis focuses on using cues from the layout to determine implicit relationships between characters. The CSeg module provides this layout information by describing the bounding boxes that enclose each character, and using this, we can find various features of interest, such as centroids (the center of mass for a character) and character size. This layout data can then be used to identify specific implicit relationships of interest, namely "horizontal", "above" and "below" (useful for fractions or operators like summation), and "sub" and "sup" (for subscripts and superscripts). For example, superscripted characters tend to have centroids higher than the a character on the baseline, and tend to be smaller.

The second form of analysis focuses on using grammars to verify the validity of an expression tree. In this process, we consider that some operators have a specific grammar: `frac{numerator}{denominator}`. Hence, when traversing the expression tree, we can use a set of these hand written grammars to distinguish among easily confusable characters like the fraction bar and the minus sign – if the character has an above and below relationship with two subexpressions in the tree, it is likely a fraction bar.

8 Results and Analysis

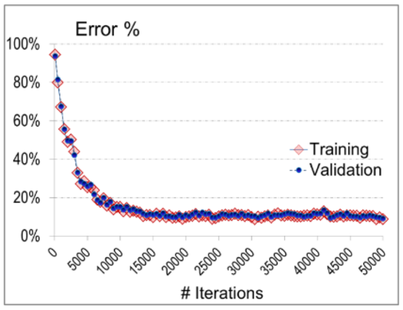
8.1 CSeg Results

In its current configuration, CSeg provides us with 92% accuracy, as measured by running it on subsets of our taining set containing 50 examples each.

8.2 OCR Results

We randomly partition the 70470 images in our input database into training, validation, and testing sets, which account for 60%, 20% and 20% of the images from the input database, respectively. We train our simple CNN in minibatch of size 64. We experiment a few hyperparameters and choose the learning rate 0.3 and the weight penalty $\lambda = 0.001$. The run-time prediction error percentage vs. the number of iterations are shown below.

We freeze the CNN model and use the 14018 test images to test the performance of our simple CNN.The



	Accuracy	F1 Score
Linear	98.46%	98.44%
RBF	97.22%	96.66%
Poly	98.88%	98.88%

CNN makes 1236 prediction errors out of the 14018 test images, which represents an error percentage of 8.817%. The top few misclasifications are lower case letter l vs 1, letter O number 0, uppercase S vs lowercase s, uppercase W vs lowercase w, Λ vs A, and ρ vs p. The confusion between l vs 1 are hard for any classifiers, even humans. Uppercase vs lowercase letters are confused because original image size is lost. We believe that the confusion between Λ and A and ρ vs p is due to the relative few number of Greek symbol training examples which may bias the training of the model. For the support vector machine, we have trained extensively on a small set of unique classes. As such, when we test the SVM, there is low distinction between the training and test set which causes a high accuracy percentage; it is unclear if this is due to the relative strength of SVM for OCR or if this is a result of overfitting.

8.3 Convolutional Neural Net

We also implement this CNN in Python. Then we train our CNN in minibatch of size 64 with the same 60-20-20 partitioned training/validation/test sets. Because this proposed CNN has a much larger number of trainable parameters, there are only 2000 minibatch training iterations per hour. We experiment a few sets of hyperparameters (learning rate, weight penalty, etc.). However, at the time of writing this report, this CNN model has yet converged.

8.4 Conclusion and Future Directions

Overall, the system provides operational levels of accuracy by using a RPPC, SVM, and grammar focused approach to the three key tasks of mathematical equation recognition. However, there are various ways to extend and improve on the existing system. For one, OCR can be improved by considering the context in which a character appear by incorporating layout information from the CSeg module. This would improve classification error for similar characters like "l" and "1"; while such pairs are difficult to distinguish betwen out of context, if we know the character is surrounded by other numbers, we can provide a much more accurate guess. Also, the SA module could be improved by using a neural net approach to take small subexpressions and map them directly to code snippets, which would also make the SA module much more robust. Other approaches to SA traditionally involve classifying the spatial relationship (i.e. horizontal, sup, etc.) using shape and layout based features [11] and using SVM classifiers; this was not possible in this project at the moment due to the lack of easily and publicly accessible data.

References

[1] “Gradient-Based Learning Applied to Document Recognition”, Proc. of the IEEE, November 1998, Y. LeCun, et al.

[2] M. Suzuki, et al. “Infty: an integrated ocr system for mathematical documents,” *ACM symposium on Document engineering*, 2003.

[3] Numpy. http://numpy.org

[4] Scikit-Learn. http://scikit-learn.org/stable/

[5] Kam-Fai Chan, Dit-Yan Yeung. "Mathematical expression recognition: a survey". International Journal on Document Analysis and Recognition. 2000.

[6] Christopher Malon. "Support Vector Machines for Mathematical Symbol Recognition" Kyushu University.

[7] Seiichi Toyota. "Structural Analysis of Mathematical Formulae with Verification Based on Formula Description Grammar" Kyushu University.

[8] University of Waterloo. "MathBrush." https://www.scg.uwaterloo.ca/mathbrush/

[9] Kirelabs. "DeTeXify." http://detexify.com

[10] Masayuki Okamoto. "Recognition of Mathematical Expressions by Using the Layout Structure of Symbols". IAPR-TC11

[11] Francisco Álvaro. "A Shape-Based Layout Descriptor for Classifying Spatial Relationships in Handwritten Math". Rochester Institute of Technology.