Untexify - a bad clone of Detexify

Robert Grimley

May 21, 2023

Contents

1 What this is

Untexify is handwriting-to- LaTeX dictionary. LaTeX is the most popular markup language for technical documents in mathematics, and is widely used throughout computer science and industry.

However, its mnemonic conventions for symbols are old and debatably antiquated. Many beginning researchers and students of the sciences find difficulty remembering the LaTeX name for a certain symbol. Untexify is an interactive canvas which will return the closest LaTeX code for any symbol drawn on it.

This is accomplished using an OCR machine learning model written in Python using TensorFlow, which is then ran in a webapp created with Django hosted online with the help of fly.io.

The training dataset was not hand-drawn, like in the case of many other MNIST-like models. Rather, it was programmatically generated using an

image-transformation pipeline created with the Albumentations library. This way, I didn't have to find or write myself thousands of hand-drawn LaTeX.

This entire project was planned and conceived in the lovely Org-mode, as Emacs is my primary development tool. The website was itself written as a series of HTML and Javascript codeblocks which were tangled together using Org-mode's exporting functionality.

This README is also my planning document, and contains annotations and citations for the various steps of the project. This is not only to lower the barrier of entry to someone trying something similar, but also for my future reference.

2 Road-map

2.1 Backend

2.1.1 Create the dataset

1. Pull a large list of symbols from OEIS

   I simply copied a table's symbols and formatted them into a file such that each piece of was on its own line.

2. Convert them into .png files

   I used latex2image to convert the list of commands into small, square images of each symbol. The program is a bit finnicky, so for future reference, I placed the file containing my equations called `equations.txt` the root of the git repository, then ran from the root:

   ```
   cd src/
   . set.sh $absolute_path_to_equations_txt
   cd ..
   cd equations.txt_aux
   python generate_latex.py
   ```

   I numbered the resulting files using Dired.

3. Sort them into classes based on their code

   I created my images folder, and used this bit of bash magic to sort them into subfolders sharing their same names:

   ```
   for i in $(seq 0 $IMAGE_COUNT); do mkdir $i; mv $i.png $i/; done
   ```

Which resulted in:

4. Simulate handwriting

   To do this I need a series of "transforms" which will piecewise randomly affect an aspect of a given image. This prevents overfitting, and in the first phase makes the model functional at all. Here are the aspects of the image I chose to transform:

Writing aspect	Transform name
"wiggliness" or poor handwriting	`A.ElasticTransform()`
Sharpening	`A.Sharpen()`
Uniform color	`A.Equalize()`
Orientation/rotation	
Scale	

   (a) Translation and scale Although a textbook cited at the keras docs mentions that convolution layers *should* be translation invariant, a cursory test of my model indicates they are definitely not. So, I need to alter the transformation stack accordingly. The model is also not resistant to the scale of the input, so I need to fix that as well.

   (b) Stroke The model is not resistant to different strokes. Depending on the way I implement the frontend, there may be no reason to train the model to recognize this.

   (c) Choose a list of symbols Initially, I chose a sample of 50 symbols picked mostly arbitrarily. The initial sample includes multiple sets of symbols which would be similarly drawn ($\prec$ and $<$, for example), and also made liberal use of ""/'s ($\neg$'s). Because no large public facing database of small symbols in the model's format exists, and the transform stack is prohibitively computationally expensive, I had to decide what my relatively small data set will contain. I decided on a set of symbols composed mostly of some of the most popular mathematical symbols.

   This might be a bit paradoxical, because those symbols which are most popular surely are the most remembered. This may be true, but it is also true that there are probably more beginning researchers and students in need of a reference for basic symbols than there are people who need to look up the more esoteric symbols. Since detexify exists and has a more comprehensive database, I choose for my tool to be more of a quick reference.

2.1.2 Train the model

2.2 Frontend

2.2.1 Hosting

To host this project I used fly.io for its excellent integration with Django, which was used to construct the frontend. Fly.io extremely simple installation instructions for a number of web-app libraries for popular languages, and it was overall very simple to use for someone not experienced in website hosting like myself.

2.2.2 Website structure

The frontend's structure was made entirely using Django, which was excellent for me as someone with lots of python experience, and little HTML or CSS experience. Most of the interface between the model (which was made using another python library, Tensorflow) and the page was handled in a single views.py file. Python acted as the glue between Django and Tensorflow, which was extremely helpful and satisfying to work with. Those parts of the website I needed to actually delve into HTML for, were done almost entirely using org-mode's helpful HTML export. I could export large swaths of org-mode documents to a nice-looking CSS "frame", while embedding HTML within the plain org text for seamless integration into the final product.

3 Testing exporting with blocks

This block is the javascript code for the HTML canvas responsible for accepting user input, in the form of hand-drawn approximations of the symbol they are trying to look up. Now, we render embed the user-facing HTML elements onto the page.