# Textify: A Neural Network for Optical Character Recognition for Typed Text

**Kim Soffen, Lisa Wang, Diane Yang, Amna Hashmi**

**Demo Video: bit.ly/textify**

## Overview

We implemented a neural network for optical character recognition, which ended up identifying ~95% of our test set characters correctly. At a high level, it takes in a .png file of a scanned page of text, splits it into separate characters, determines what each of those characters are, and outputs a .txt file containing the text that our program has read from the image. We also created a command line search function to be used on this outputted text file.

## Planning

Our functional spec can be found in /report/Draft_Spec.pdf and the our technical spec can be found in /report/Final_Spec.pdf.  We planned to implement the following features:

- Take in a scanned document and split it into letters, spaces, and some punctuation.
- Implement a neural network that recognizes letters and some punctuation
- Output characters into a .txt document.
- Make a command line search function

Our original planning was quite on target; we had a good idea of what goals we had to achieve and they were of a reasonable scope. We were able to meet these goals, with some minor tweaking. We changed our program to go from scanned image to txt to being png to txt, just because of what the Python Imaging Library was best equipped to handle. We managed to pace our work well, so at every milestone we had accomplished something new. By the time the final spec was due, we had started some coding. By the first checkpoint, we had completed most of our pre-processing program as well as the backbone of the neural network functions. By the second checkpoint, we had completed the code for our preprocessing and neural network programs, and we had trained the network. We actually went beyond our originally intended scope of the project by including character recognition for lowercase letters and periods along with uppercase ones. This was achieved by adding these characters to our data set and training/validating them. Additionally, we achieved a high accuracy for fonts that were variable width, including Tahoma and Verdana, a few serif fonts, such as Times, and many sans-serif like Arial.

## Design and implementation

Overall, our experience with design, interfaces, languages, and systems was quite positive and fruitful. However, testing became quite an endeavor because we created our own training set.

In terms of design, it was split up into three major components: preprocessing, the neural network, and then connecting the two, as detailed below. All of the functions are exposed, but we believe we effectively modularized the design to provide a clear logical system. Additionally, we were able to test simple networks before implementing our more complicated and larger scale network. For example, after example_test.png was run through our character_extraction.py program, the outputted characters were stored in the example_text_chars folder and example_text.txt, which demonstrated that we effectively pulled out each character and space in a 20 x 20 pixel box to be given to the neural network.

In terms of language, we chose to use python because it's a high level language that allows us to focus on logic and design rather than getting caught up in syntax errors.

In terms of testing, we tested our neural network via the validation sets and the testing set. erm.py, character_extraction.py, and search.py were all tested via visual inspection of the output versus the input. We did not think assert-style tests like those written for psets would have been appropriate in this context, since our program is aiming for a high success rate, yet will not be 100%, which is what an assert requires.

One major design decision we made was regarding the # of hidden nodes to put into our network. We tested at 30, 60, and 90 hidden nodes. We found, as expected, that more hidden nodes corresponded to a higher performance level (seen on the performance_graph.pdf in the data folder) as well as a longer run time. We decided to balance these two by choosing 60 nodes. We then tried multiple learning rates -- 0.1 and 0.2 -- and found that the higher one was more erratic, as expected, and ended very slightly outperforming 0.1, so we went with that.

We couldn't find a suitable training set of typed letters in any online databases. We created our own training, validation, and testing set and had to figure out ways to make sure each letter was labeled correctly. Originally,we created our own training and validation sets by generating images containing the alphabet typed in a wide variety of Microsoft Word fonts. The training set had 18000 characters and the validation set had 2000, but when training with 100 epochs, we were at ~96% with the training set and 95% with the validation set.

In addition, we had to figure out the fonts the wouldn't create problems such as letters stuck together, finding spaces between letters that weren't there, or were non-standard looking. For example, we avoided script-style fonts. Our biggest challenge in the pre-processing part of the project was figuring out the best way to detect spaces. Essentially, we had difficulty finding a good metric to determine how big a gap had to be before we considered it a space character ( instead of a mere gap between letters). We tested different widths based on the height of the letters, and found that the best number was the height divided by two. When we first tested the program on only capital letters, everything worked very well, so we added in lower case and periods. The accuracy is slightly lower now, but the program is more adaptable to every day use.

Description of each file and class found in README

## Reflection

If we had more time, we would work on improving the accuracy of the network even more. It is currently performing around 95%, but this actually means many fonts operate perfectly and some not at all (mostly serif fonts). We would like to adjust our training such that it works on a wider variety of fonts. We would also like to expand the program so that we can take in more image formats, such as pdf or jpeg. This require a higher level understanding of the Python imaging library, because there are certain compatibility issues there. On the preprocessing side, we would like to improve how we find spaces; currently we're dividing the height of the character by 2 to estimate the length of a space; there's certainly a more reliable way to find the spaces, we just didn't have enough time or knowledge to implement it.

If we were to redo this project from scratch, we would still use the framework from CS181, but we would probably work on improving our training set so that more varied fonts are represented (or just outsource this part, because it is a serious pain to label and check through that many sets of pixel matrices). We would also spend more time really understanding what each function in the neural network does before we started coding (rather than during), because we found ourselves rewriting code that could be found elsewhere in the CS181 framework. It would also have made the purpose of the functions we were writing more clear. We would also change the size of the images we take in from 20x20 to 30x30, because the smaller number of pixels means we lose a lot of distinction between the different letters. (ie a serif E and B look very similar when looking at few pixels). More pixels would make the difference bigger so it would be easier to detect each character correctly.

We were surprised at how difficult it was to detect spaces. A lot of our errors in the final test document are the result of spaces being neglected or extra spaces being detected. We tried many different methods and then scoured the internet for ideas, but this seems to be a widely experienced problem that doesn't have a lot of answers. (One research paper simply said "Detecting spaces can be very difficult," without any specifics on how to face said difficulties). We also found that character recognition becomes very difficult as you add more and more fonts to the training set, as letters look very similar in certain fonts, which screws with the data set.

Our best choice was about a week in when we chose to abandon our own program and go with the CS181 framework. This made *what* a neural network even was a lot more clear to us, so we could focus on how the network fit together and the functions to move data through the network, rather than struggling with how each class should be designed. Our worst choice was our last minute decision to expand beyond a few standard fonts to try to capture all the fonts in Microsoft Word. This made each of our font detections less accurate, and it's not even like most of those fonts would ever be used. After the fact, we realized that the existing programs that do this typically focus on the primary 5 or so fonts in Word. We also lost a lot of accuracy when we decided to include lowercase letters and periods for the same reason.

The most important thing we learned was how neural networks are implemented and how the back propagation algorithm functions. Through reading documentations from other neural network projects and talking with our project TF, Kenny, who took CS181, we were able to understand the many functions that go into making a neural network both effective and functional. In addition to that, another important thing we learned was how to work with the Python Imaging Library in manipulating images for character extraction.

Group member contributions:

- All of us worked on labeling and checking the training, validation, and testing sets.
- Kim and Lisa worked mostly on the implementation of the neural network, making sure the neural network functioned and all the main learning aspects of the neural network behaved correctly, i.e., propagating inputs and back propagating the weights.
- Kim donated her entire CPU to the cause of training the neural network for a total of 36 hours. But actually. She had to walk around holding her computer open since she couldn't close it and let it go to sleep.
- Diane and Amna worked mostly on the pre-processing part which was the character extraction of letters from a png file to a matrix of black and white pixels of each letter in a txt file.

## Advice for future students

The most important thing we learned, that everyone contemplating a project like this should know, is start EARLY, because there will be problems that you won't foresee. If you want to use machine learning, it's very helpful to read through the lecture notes of CS181 on neural networks (or whatever the more relevant notes are for your project). Don't make your own data set; ever. You **will** be up for 36 hours mindlessly labeling your training/validation/testing sets. On the upside you can catch up on Modern Family while you do it. Also of note, future students should ask Professor Morrisett if he prefers Gateway computers because its logo is of a cow cube.