**Intro:**

Our objective was to create a neural network which could recognize letters presented in a captcha image. Limiting ourselves to captchas following this format:



All images had 6 letters, and no alterations beyond the above blurring, noise, and rotations. In addition, we limited the alphabet to not include: f, i, j, l, m, r, and t. This leaves a truncated alphabet of 19 letters, and no Capitals. So, these models are highly specialized and not built to beat captchas in general, or even any real world applications of the above format.

**Image processing:**

Prior to building any neural networks, we found that the images required a good deal of pre-processing. Using a few different python image libraries, we found OpenCV to be the most useful, and modified the images in this way:

Given



We found that it was actually relatively difficult to remove noise. Most of the built-in methods seemed to use contour detection and functions which tend to treat black noise as actual image, and negative space as the distortions. So, we inverted the image



From there, it was relatively easy to find all the empty space and fill it in, provided the total area was below a set threshold.



Crop edges off, re-invert, and the built-in contour detection was able to find where the letters are.

Further processing was needed to handle when letters overlap and read as one contour, but those shapes were exceptionally long and easy to detect.

(Maybe explain here that all of the letters used were approximately the same width for exactly this situation. Including i’s and j’s made it too difficult to address every possible combination of widths)

About 16,000 captcha images were prepared, and even though our code wasn’t able to parse all of them we were able to produce nearly 95,000 individual letters.

**Building the neural network:**

As these letter images



were very easy to convert into small, black/white images, we decided to use a convolutional neural network, which not only learns based on the locations of each pixel, but also by the relationship between the pixels. This allows it to learn shapes, which helps fight against the rotations of each letter.

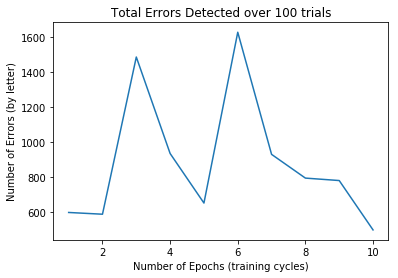
We started with looking at how others tackled similar problems with character recognition, and we found some promising resources that seemed to work with only two convolutional layers. The problem then was determining some particulars:

* How many training cycles (epochs)
* How many output filters per convolution
* If including a hidden, dense layer, how many nodes are appropriate

To test these, we simply made a bunch of different models while varying one of the above parameters and keeping the others constant. All epoch models were built on the same training data, all filter models were built on a separate set of data. We then produced 100 sets of 10 random test images, ran them through every model built to determine the ideal parameters.

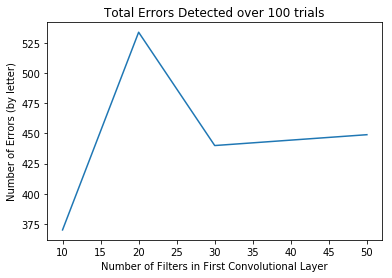
**Epoch Test Results:**

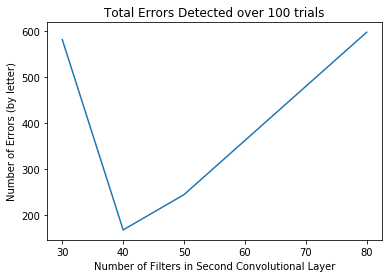
Curiously, more training cycles did not translate to a higher degree of accuracy. From research, it seems that a common problem with CNNs is a tendency to overfit the training data, and a greater number of epochs possibly gives the model too much opportunity to see patterns which don’t exist/are not relevant. After creating ten models with different numbers of training cycles, with all other parameters the same, we ran them through the random tests and found these results.



**Filters per Layer Results:**

After settling on 2 training cycles as sufficient, we repeated this test method to determine the ideal number of filters for each convolutional layer. Like in the epoch test, more did not mean better, it just meant a longer runtime.





**Number of nodes in hidden layer:**

Choosing the optimal parameters from the previous tests, we found the best results with

* An initial convolutional layer with 10 filters
* A secondary convolutional layer with 40 filters
* Minimal advantages in training cycles greater than 2

We last wanted to figure out how many nodes we needed in the hidden layer where the magic happens. Repeating the process above, we found this

