Cos 429 Final Report

Austin Mejia, Matthew Hetrick, and Preeti Iyer

January 14, 2020

https://github.com/austinlmejia/COS429/tree/Final_Files

1 Introduction

Handwriting recognition has been booming in recent years. With new tablets and smart pens coming out every year, programs that interpret writing are becoming ubiquitous in modern technology. Moreover, predictive and assistive handwriting interpreters hold much promise for a variety of applications including disability assistance and IoT notebook incorporation. Therefore, we were seeking to design a product that would be able to fit into this burgeoning milieu.

We realized that there are many handwriting recognizers, such as MyScript and WIRIS, that recognize mathematical handwriting but none can solve equations on the level of a WolframAlpha or SymboLab. We thought it would be exciting to try and fit into that space in which a program can combine mathematical logic with handwriting recognition to solve an equation.

2 Related Work

Though in it's infancy, incredible applications have already been found for the usage of handwriting analysis. Most of the application come in one of two forms: digitization of previous archival information that was written by hand, and the digitization of current handwritten documents.

Applications thus far appear nearly limitless: in November of 2019, the National Institute for Water and Atmospheric Research partnered with Microsoft to develop a network capable of reading old weather logs. Access to this information could yield important insights to climate trends in the face of global warming. Doxper, a start-up from India, is developing a record system that digitizes patient's records as doctors write them. Better record keeping can make document sharing between hospitals dramatically easier and benefit both patients and practitioners. We think mathematics could be another such application of handwriting recognition.

Utilities such as WolframAlpha have been crucial developments in making the fields of science and mathematics more accessible to the general population. Since it's development in the late 2000's, WolframAlpha has shattered the paradigm for search and how people seek answers. It popularized the use of mathematical expressions in search through language that best represents the problem. Further, knowing the solutions to problems enables a deeper understanding of concepts at play. It's one thing to memorize that $2 + 2 = 4$, but a grasp of why addition performs this way is a far more valuable insight. Utilities that provide the answer are not easy outs, but tools that facilitate the connect different concepts in a single challenge. However, typing an equation into online tools is not nearly as natural as writing them by hand is. Without this ease of use, these WolframAlpha continue to fall short of their promise to make knowledge accessible. This is where we see potencial for our group to enter.

Some mathematical handwriting recognition tools exist. WIRIS and MyScript are two such tools that are capable of reading and solving problems. However, both use a real-time virtual environment to capture handwriting, either my writing on a screen or via a mouse. These tools are able to track the motion of handwriting and form additional insights based off of the velocity, angle, and curvature of strokes. Further, they their real-time capability means that that can be less noise tolerant as unrecognized symbols will be removed from screen, inviting users to attempt again. Photomath, a popular app, allows users to take photos of equations and solves them real time. However, Photomath limits itself to a 1x3 dimension window, meaning all equations must fit into that space. Additionally, there is no functionality to upload an existing picture or file of handwritten text.

Our team aims to create the first mathematical equation recognition and solving program designed for uploaded images of all sizes. Other systems require some real-time input, such as writing in a contained environment or taking a live photo. Our project aims to add the additional utility of uploading any major image format at any time. We believe there is unmet need in this respect that our project aims to address.

3 Data sets

While we knew the system would involve a complex network of some sorts, which would require thousands of images to train and test on, it was also apparent that a database with images of equations did not exist. Since we wanted the ultimate design of our system to take in a picture of a full equation and output an answer it was important that we used fill equation pictures in constructing our system. We ended up using a combination of existing large databases and our own data construction to build and evaluate our system.

3.0.1 MNIST Dataset

Upon initial research we came across the MNIST database. This is a database of handwriting images which includes 60K training images and 10K testing images from American Census Bureau employees and American high school students. This provided us with a critical mass of images with which to train a network/classifier for single digit images of handwritting. Some examples from the dataset are seen below:

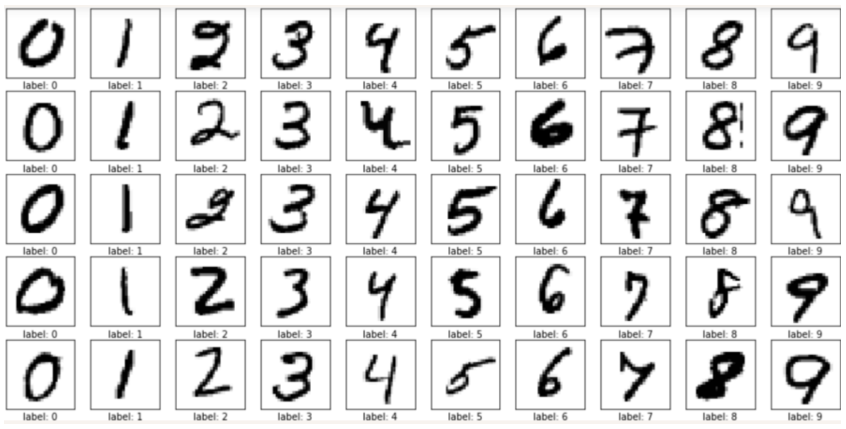


Figure 1: Sample Images from MNIST

3.0.2 Our Equation Database

To evaluate our overall system we needed full images of equations, and thus we constructed a handwritten equation data set comprising of 56 different equations. We understood that each individuals handwriting style is different so in order to build and train a robust system, we gathered handwriting styles of our groups members and of other individuals not in our group using both dominant and non-dominant hands. We made sure we had a diverse range of style and the set was ultimately tagged with a label descriptor of thin, medium, thick v1 (dominant hands), and thick v2 (non-dominant hand) for our analysis.
We also tried a few data capture methods including capturing phone images of paper written equations and capturing images of digitally written images; we ended up going with the later as our source of data since taking images of a paper often resulted in difficult lighting and shadows which impacted the accuracy of our system and through some research we found that similar technical projects in the field like apps in development right now also have chosen to capture digital data (though it

would be an interesting follow up to add the image processing to handle both types of data). Some example images from our dataset, of varying handwriting styles, are listed below.

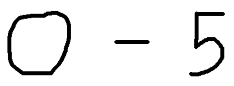


Figure 2: sample 'thick' image from our constructed data set

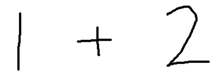


Figure 3: sample 'medium' image from our constructed data set

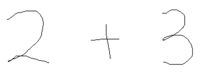


Figure 4: sample 'thin' image from our constructed data set

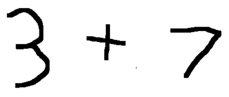


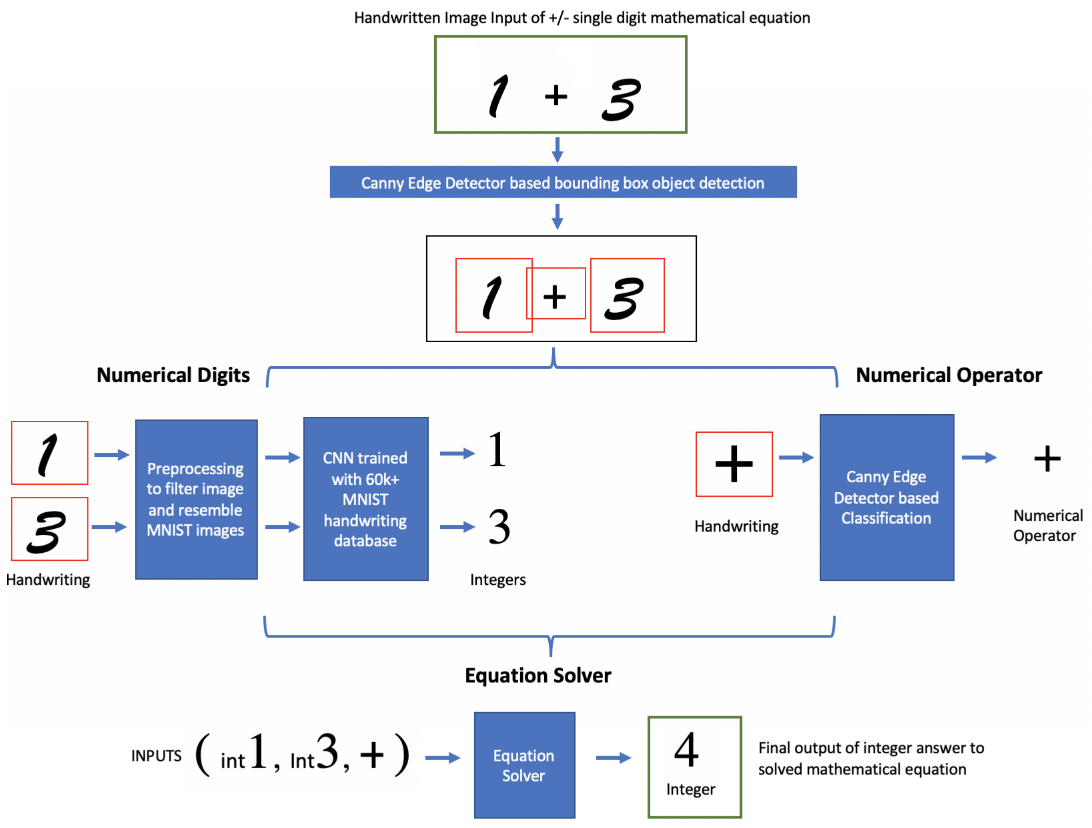
Figure 5: sample 'thick' image from our constructed data set

We hoped that using two different sources of images (our own image captures and MNIST images) would make our system more robust since single digit classification would have to work for different images (and thus we'd hopefully be able to pre-process any image input to be similar enough to MNIST data where a classification built with MNIST would work).

4 System Design and Implementation

4.1 High level system overview

We started this project intending to build an equation solving system that was fast, easy to use, and could be used by a child (from our initial understanding of cool applications for this type of app it followed these were the scenarios under which would make sense), and thus we built our system with this in mind. Opting for as much accuracy as possible, but also leaning towards more simple design architecture in situations where complicating the architecture would not greatly improve the accuracy. For example, we opted for Canny Edge Detection based bounding boxes and classification instead of using full scale object detection and an additional CNN for these components, because we found that we could leverage the simplicity and speed of the simpler system, that didn't require training/running an additional object detector or classifier, while still being accurate. It was also important to keep in mind the structure of our data - 60K training/10K test MNIST data set 56 hand generated images of the full equations; since we ultimately generated our own data we had to keep in mind our small sample size of final input data for our system. A schematic of the system architecture is included in the figure below and outlined in the following subsections:



Handwritten Image Input of +/- single digit mathematical equation
Canny Edge Detector based bounding box object detection
Numerical Digits
Numerical Operator
Handwriting
Preprocessing to filter image and resemble MNIST images
CNN trained with 60k+ MNIST handwriting database
Integers
Handwriting
Canny Edge Detector based Classification
Numerical Operator
Equation Solver
INPUTS (int 1, Int 3, +)
Equation Solver
4
Integer
Final output of integer answer to solved mathematical equation

Figure 6: Overall System Architecture

4.2 Canny Edge detection based Bounding Box extraction

Detecting and drawing the bounding boxes around elements was a crucial first step in implementation. Without it, our network optimized for individual symbols would fall short.

Our original direction was to utilize a object detection tool set in order to identify individual elements. However, we quickly realized that such an approach would be far more than necessary. Object detection is typically applied to scenes with multiple elements–a pedestrian crossing a street or flower blooming in the woods–where a network needs to separate objects from a background. However, our inputs were relatively sterile in comparison, being black text on a white background. The elements we want to identify already clearly stand out. Developing and training an object detection network would dramatically slow performance and add unnecessary weight to the program.

Instead, we realized that a modified canny edge detector could create accurate bounding boxes. Given the contrast of elements against the background, a Canny Edge detector will capture the entire object as one continuous outline. Using this outline, we can draw padded boxes around each number and center them based on the edge map. As each element is comprised of a continuous line and each line is disjoint, we can draw high accuracy bounding boxes in a fraction of what object detection would require.

4.3 Pre-processing to resemble MNIST images

One of the biggest challenges we found when testing our system was formatting our handwritten data to match the formatting of the MNIST data set. We needed to do this because the model trained on the MNIST set expects features to be formatted a certain way, which we learned after uploading our handwritten images and having our predictions operate at around 20%, which is not much better than random selection. Therefore, we needed to perform a more involved image processing step before having the CNN model classify our image. This proved much more difficult than simply thresholding black pixels, setting those as black, and setting everything else as white. The MNIST images are tightly fit to a 20x20 grid, and then this 20x20 grid is centered and resized in a 28x28 image. We had to figure out a way to do this with our handwritten digits. Luckily, we found a great article that stepped through how to process handwritten pictures and prepare them like the MNIST data set, and it even had some open-sourced code examples. We based our logic off of this example for our pre-processing before passing the processed image into the Keras predict API to output the predicted class. The logic was as follows:

1. Convert the color to grayscale to go from 3 to 1 channels
2. Invert the color to match the black background/white number format of the MNIST set and resize the image
3. Threshold using the cv2 threshold function with a binary threshold argument so that there is no system interpretation of the threshold
4. The next sections include tightly wrapping the image and then adding the necessary padding to fit the image to the 28x28 format. We used the open sourced code from Github for this, just tweaking the parameters to suit our needs. We used this because there were a lot of data type conversions and intricate rounding that we needed to make sure worked. This section is commented in our code.
5. We next needed to center the image, which after some googling seemed to be best accomplished by using the Scipy method that measures the center of mass of the image. We then calculated the differential of how much the image needed to move up/down and left/right.
6. To actually shift the image, we implemented the warpAffine function. This implements an affine transformation, which was recommended to keep our

key points, lines, and planes preserved when we actually shifted the image. We used the translation matrix as the argument in this function, where

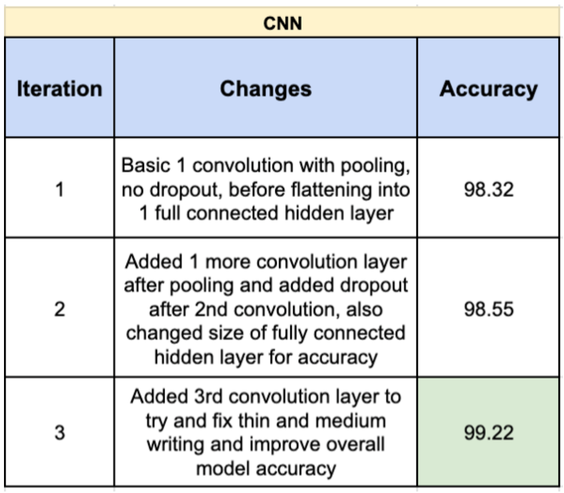
$$M = \begin{bmatrix} 1 & 0 & sx \\ 0 & 1 & sy \end{bmatrix}$$

At this point, the handwritten image was ready to upload into the model for prediction. We normalized and reshaped the image to fit the Keras API.

4.4 CNN Trained on MNIST

We decided to use the Keras Sequential model with the ”adam” optimzer and the sparse categorical crossentropy error based on the recommendation of most google searches for building a CNN for classifying MNIST images. The sparse categorical error was chosen because our sources suggested that categorical cross-entropy is a great loss measure for classification, and making it sparse allows us to keep the simple numeric/class label relationship without hot endcoding. We used Keras to download the MNIST dataset.

For this model, the general format of the CNN is as follows: a convolutional layer followed by a max pooling layer, followed by a fully connected layer, whose output is fed into the softmax function, the maximum class of which is the predicted class. This was pretty consistent for all of the articles and tutorials we found online regarding the Sequential model, specifically ”Image Classification in 10 Minutes with MNIST Dataset” and ”Keras Conv2d and Convolutional Layers”. We experimented with various architectures by modifying this general format, mixing and matching to learn more features with other convolutional layers, adding and modifying dropout to decrease overfitting, and added a pooling layer after a convolutional layer to prevent overfitting. Below is a summary of the main changes between models and the associated accuracy:



CNN		
Iteration	**Changes**	**Accuracy**
1	Basic 1 convolution with pooling, no dropout, before flattening into 1 full connected hidden layer	98.32
2	Added 1 more convolution layer after pooling and added dropout after 2nd convolution, also changed size of fully connected hidden layer for accuracy	98.55
3	Added 3rd convolution layer to try and fix thin and medium writing and improve overall model accuracy	99.22

Figure 7: CNN Architecture Changes Chart

When we were ”mixing and matching”, the goal was to visualize misclassified images and figure out what was wrong with them. After the first architecture, the mis-classified images were ones with more gray pixels within the number. Therefore, we added a second convolutional layer to try and have the network learn more features about each input image, even though this did significantly increase training time. We also added dropout and max pooling after the second convolutional layer in order to reduce overfitting given that we were increasing the filters our network was learning. We chose max pooling due to the fact that we wanted to keep learning important features whereas if we had used average pooling our images would have been smoothed.

After further analysis on the images that were misclassified, we still had some issues with gray pixels as well as images that appeared slightly disconnected, like a 7 where the top was not continuous. Most of the other misclassified images were images that even we couldn't necessarily classify with our eyes or were very ambiguous. Therefore, we added another convolutional layer and dropout after this layer to continue to try and learn these gray pixels. After a long training period for 20 epochs, we were able to acquire a model that had a 99.22% accuracy on the MNIST data set, displayed below.

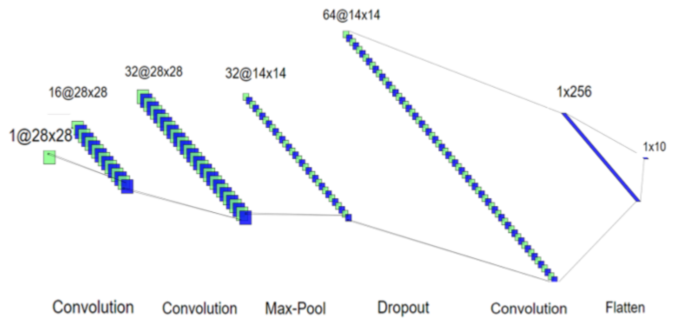
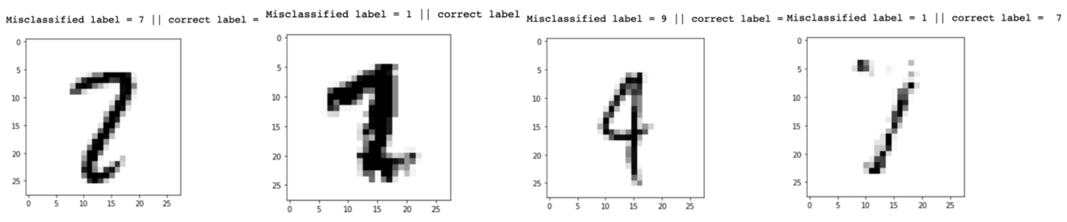


Figure 8: Final CNN Architecture (produced with NN-SVG sketch)

We do note that, out of the misclassified images, the disconnected images and light-gray pixel images were still the main ones misclassified, such as a 2 that had a short bottom line and resembled a 7, a 4 which had a seemingly circular top that looked more like a 9, and so on, examples of which are displayed below.



(a) Misclassified 2 (b) Misclassified 2 (c) Misclassified 4 (d) Misclassified 7

Figure 9: CNN Misclassified Analysis

In general, images that had a lot of disjoint features were the ones misclassified, which helped us figure out the issues with our system later on in the project. We ended our CNN modification here, as we believed the 99.22% accuracy was robust enough to be adapted to our own data. We then saved the model using the Keras Sequential API save option so that we didn't have to train the model each time we ran the program.

4.5 Canny Edge detection based +/- classification

Our base system just dealt with two mathematical operators - plus(+) or minus(-) - which are structurally very different. The plus sign has roughly equal dimensions in the x and y directions, versus the minus sign's dimensions in the x direction is far greater than the that in the y direction (see Figure 5 below). Therefore, we decided to using a Canny Edge Detector to detect the edges and understand what the size in each dimension was for the numerical operator, which could be used to infer which operator it is. In our implementation, we use data gathered from the Canny Edge Detector based bounding box while drawing the second box on our input data (since we are only dealing with single digit +/- equations we relied on a known 1 number,1 mathematical operator,1 number data structure). From this, we extracted the pixel values of the most left and right columns and the highest and lowest rows that belonged to the edges (mathematical symbols). We used the ratio between thee understood column and row dimensions to discern between plus and minus operators.

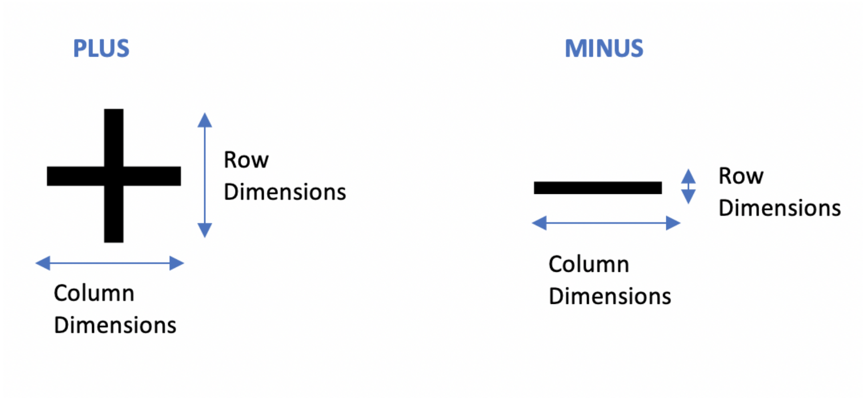


Figure 10: +/- Dimension example

While different styles and types of handwriting will result in different dimensions, in general the ratio between the column and row dimensions for the plus sign is much closer to 1, whereas in a series of 10 tests on minus sign data the ratio (column Dimension /Row Dimensions) ended up being closer to 15-16. With more testing, we decided a threshold of 2.5 for a minus sign and found that if the prepossessing was successful then with this threshold the mathematical operator was always classified correctly.

4.6 Equation Solving and combining the system

The final portion of our system read in the outputted integers and symbol from the previous classifier and network, and incorporated the logic to solve the equation. Once again we relied on a simple design for this portion since the structure of our data (1 number,1 mathematical operator,1 number) was simple and known to be constant. We strove to develop our system in a modular way, thus for this final step we were able to simply call the various functions/modules of our implementation and synthesize them into a final equation structure. This final function outputted an integer answer which was the final output of the entire system.

5 Evaluation Methods

Since we developed this system in a very modular way, each part of our system was tested and iterated on, evaluated both quantitatively (accuracy of CNN, percentage of correct classification of +/- and images to improve Canny Edge Detector) and qualitatively (visualizing bounding box detection and incrementally seeing if there was enough padding, visually testing if our pre-processing resulted in data similar to the MNIST set, visualizing miss-classifications for the CNN to see which pixels got picked up and which did not and adapting the CNN architecture based on visual understanding). However, we also wanted to test our overall system in a systematic way. We evaluated our system on a variety of different tiers of accuracy to understand the performance of all sub-parts of our system, keeping the goals of our exploration in mind: to build an accurate and easy to use equation solver.
Understanding the different steps of our system and the various stages of technical processing, we came up with a following list of metrics to better understand for our system's performance:

1. Whether the system was able to correct detect bounding boxes and was correctly pre-processed (registered after pre-processing on legend)
2. Whether the mathematical operator was correctly classified
3. Whether at least one number was correctly classified (understood as partial numerical classification success of the system)
4. Whether the total system was accurate (the outputted answer was the correct integer response for the inputted picture - interpreted as a total pipeline success)

While all the metrics gave a good understanding of the performance and challenges of various elements of our system, ultimately we considered our system to fail in some way if it did not produce a total pipeline success (since there is no room for error for a real life application of this technology). A complete system failure was if it failed on the first metric (since the rest of the pipeline could not continue).

6 Initial Results

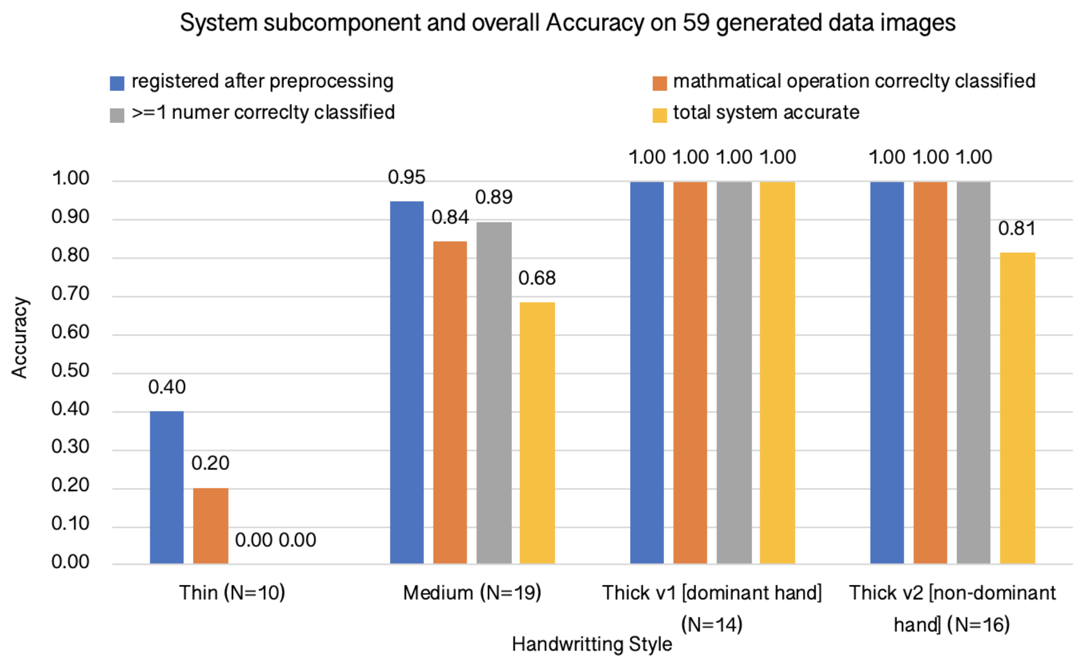


Figure 11: Component-based System Accuracies

We were limited by our small data set, but using our data set of N=59 images, we saw the following accuracies for the total system: 100% for thick equations written with dominant hands (thick v1), 81% for thick equations written with non-dominant hands (thick v2), 68% accuracy for medium-thickness-written equations, and 0% accuracy for thinly-written equations. A list of all accuracies we measured can be be seen in Figure 11.

It was seen that the thicker handwriting was best detected by our system, as we found the thin handwriting was often unable to be picked up / processed correctly - only 40% of the images were even able to be registered with bounding boxes drawn around the post processed images (more information on this in 7.3). 0 of the 10 thin handwriting images resulted in even a partial numerical classification success. Overall, for the handwriting styles that did not have 100% success in registering data (thin and medium) the accuracies were much lower, since that results in a complete system failure.

Between the two thick variants, as expected, the neater handwriting (dominant hands) had higher accuracy due to a more precise structure.

7 Challenges

7.1 CNN Challenges

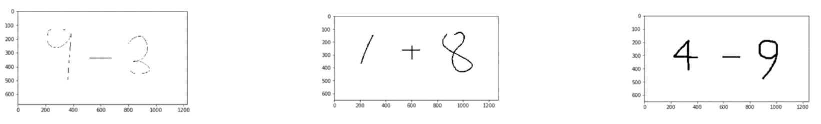
The first challenge we came across was how to actually construct a CNN to learn to predict classes from the MNIST data set. When we found out that the recommended way to do this was to use TensorFlow with the Keras API, We had to read numerous tutorials and descriptions of the various parameters to pass with the convolutional layers, pooling layers, etc. in order to make our CNN actually learn the features from the images. Once we got that figured out, the main challenge with the CNN was just the training time of imlementing changes.

7.2 Image Processing Challenges

One of the hardest parts of this project was figuring out why our CNN was not accurately classifying our uploaded, handwritten digits. We realized that the MNIST data set was particularly formatted, and once we determined the processing steps we needed to take to format the output of the bounded boxes, we were able to drastically improve our performance.

7.3 Image Thickness

As one can see in the results, the thickness of the digits in the handwritten image is of drastic importance with our system. Thinly-drawn images, once processed and resized, will often not register in the CNN and therefore the CNN will randomly predict a number. Moreover, some thin images did not output any dark pixels after the drawing of the bounding boxes. This caused error messages in the processing step because there was no data to be resized or processed. This also happened with medium-thickness handwritten digits, but the performance was more variable, with the system often misclassifying images rather than throwing errors. By training our CNN to recognize more features, we were able to have better performance on the medium images and also have a few thin images perform better. But overall, the thickness of the handwriting still basically determined how well the system would perform. Examples are included below to demonstrate thin, medium, and thick images.



(a) Thin Digits (b) Medium Thick Digits (c) Thick Digits

Figure 12: Image Thickness Examples

The main reason for this thickness issue is that in the processing step, we thresholded with a value of 115, about half of the max of 255. For our images that had white numbers and black backgrounds, we started at a threshold of 180, but this had very low performance and often output only certain pixels of the number. We kept lowering the threshold and the performance kept improving, up until around 115. Much lower than this, and we began to get noise in the non-digit portion of our images, which lowered our performance. So, we settled on the value of 115. One can see below that the 8 here has many more light gray pixels than the 9. The 8 is from the ”medium” thickness sample above, and the 9 from the ”thick” example. This is a visualization of how the thickness impacts the thresholding.

(a) Lighter Digit 8 (b) Darker Digit 9

Figure 13: Pixel Intensity Examples

This leads to one potential extension of this project, in which would try and find a way to thicken the thin numbers. We could potentially use some sort of nonmaximum suppression and hysteresis thresholding on the thin images to try and trace out the number by padding it with darker pixels, which could potentially yield better results.

8 Conclusion

Overall, we set out to create a handwritten equation solver in which one can upload a handwritten equation and the system would provide an answer. We initially had ambitious goals with trying to be able to solve advanced math problems, but the more we got into the nitty gritty of the implementation, we realized that a more viable product would be to start with 3-item equations that followed the same pattern of: (Number 1) (+ or -) (Number 2). We then designed a system architecture that would allow us to write programs that took in the equation, found the relevant numbers, and output the problem’s solution. We did this through first using canny edge detection to draw bounding boxes around the elements of the equation, processed these new images to be formatted the same way as the MNIST data set, then input the formatted images into a CNN trained on the MNIST data set, and finally input the classified numbers to our mathematical logic program. Our system operated at 90.5% accuracy for thickly-written equations (v1 and v2), 68% accuracy for medium-thickly-written equations, and 0% accuracy for thinly-written equations. This leads us to believe that future iterations of this system could implement a program that recognizes, processes, and thickens more thinly-drawn images. Additionally, future iterations could implement a more sophisticated object recognition technique to recognize a larger array of digits and mathematical problems. Ultimately, by building the full pipeline for a handwriting equation solver we were able to understand many of the challenges involved with this type of image detection / classification. While the MNIST database provided a great dataset for us to build a very accurate CNN, preprocessing our own images to resemble that of the MNIST data was very difficult and with more field research we found papers that question how replicable MNIST handwriting data truly is. By iterating over our preprocessing methods and CNN we were able to make an accurate system for thick and stable handwriting, but found that our system was not ultimately robust to detect all kinds of handwriting.

9 References + Tutorials

[1] Allibhai, Eijaz. “Building A Deep Learning Model Using Keras.” Medium, 21 Nov. 2018.

[2] Brownlee, Jason. “How to Choose Loss Functions When Training Deep Learning Neural Networks.” Machine Learning Mastery, 29 Jan. 2019.

[3] “How to Save and Load Your Keras Deep Learning Model.” Machine Learning Mastery, 12 May 2019.

[4] Geometric Transformations of Images — OpenCV-Python Tutorials 1 Documentation. Accessed 13 Jan. 2020.

[5] Kröger, Ole. “Tensorflow, MNIST and Your Own Handwritten Digits.” Medium, 15 Sept. 2018.

[6] Optimizers - Keras Documentation.Accessed 13 Jan. 2020.

[7] Rosebrock, Adrian. “Keras Conv2D and Convolutional Layers.” PyImageSearch, 31 Dec. 2018.

[8] Sequential - Keras Documentation. Accessed 13 Jan. 2020.

[9] Yalçın, Orhan Gazi. “Image Classification in 10 Minutes with MNIST Dataset.” Medium, 5 Sept. 2019.