Image Recognition Math Solver

Jennifer Chiang, Alyssa Morada, Marcos Herrera, Enrique Hernandez
California Polytechnic University of Pomona
Computer Science Artificial Intelligence: CS4800.01-02
Professor Adam Summerville
jchiang@cpp.edu, ajmorada@cpp.edu, mherrera@cpp.edu, eghernandez@cpp.edu

Abstract

We created an AI web application that can solve simple, handwritten math problems. Our implementation recognizes numbers as well as mathematical syntax using the open-source learning platform Tensorflow. The AI accepts the image of the equation as a whole and solves it, providing the user with a calculated output. (As for now, the application only supports basic math equations.)

Introduction

When you think of AI, what first comes to mind? Could it possibly be video game intelligence; maybe self-driving cars; roomba path tracking? Let's take it down a notch to a much simpler implementation that can commonly be overlooked but still is impressive in actuality: image recognition. Now, let us combine it with a common struggle for students and overall world applications, the subject of mathematics. What do we get in combination? We get an image-recognizing math solver.

We recognize math as the universal subject that many students commonly have trouble comprehending. The goal that this project aims to reach is to provide assistance where understanding is lacking. Our interface, although simple, supports handwritten equations. Our math solver, once recognizing the values of the input, will then be able to solve simple mathematical equations like a handwriting-recognizing calculator.

This project provided us with newfound knowledge and deeper understanding of Machine Learning and Artificial Intelligence. Our build trains a neural network model for classification of mathematical syntax and integer recognition. This is similar to what a human brain classifies as "what you see is what you get" with effortless ability to distinguish the differences of everyday objects. It is not as easy for machines to understand. Throughout this paper we explain our methods, applications, and the troubles that followed in pursuit of our goal.

Related Work

There proves to be no shortage of open-source "math-solvers" and definitely no shortage of the premium-subscription solvers with all sorts of other benefits.

Microsoft Math provides a complex math solving tool that takes in a variety of advanced mathematical functions (See Figure 1.1). As shown in its interface, it requires a text-input submission, supporting other characters by providing a click-on special character keyboard.

Another easily-accessible math solving web application includes Symbolab (Figure 1.2) which works similarly to Microsoft Math in prompting the user for a text input equation and a provided math character fill-in.

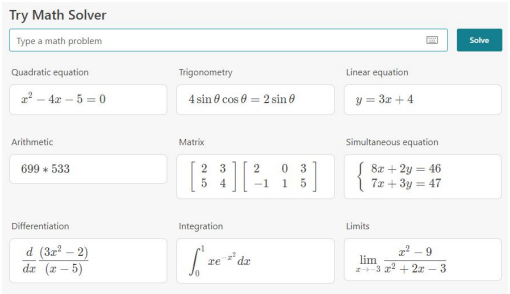

Figure 1.1 - Microsoft Math interface.

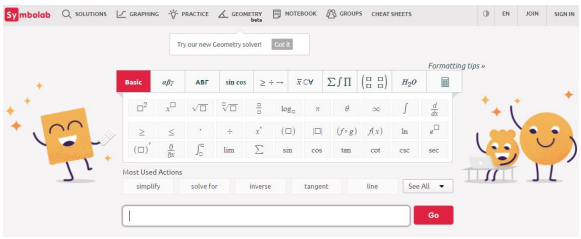

Figure 1.2 - Symbolab Math Solver interface.

What these applications share in common is a tedious process of formatting the wanted equation in the input bar with the particular syntax of the program and checking if it is being read properly. What makes our project differ is being able to eliminate that entire process and make available the option to draw out the equation themselves.

Take Chegg for example. With Chegg, a student in need of assistance is either capable of posting an image of their problem, or submitting a typed-out version of their problem for the Chegg teachers to read. Outside the step-by-step learning that the teachers do post also, after a few days-- sometimes sooner--an answer is given to the student. Now imagine something similar to that, but the answer is accessible on your own time, with a program that is capable of reading and recognizing your own handwriting. That is what we aim to fulfill.

Our Method

When breaking the problem up, we discovered that splitting the project into 2 sub-projects--front end and back end--works best and most efficiently by allowing team members to work simultaneously and independently.

For the back end we chose to implement Machine Learning because it would provide the necessary requirement for character recognition, including digits and symbols for operations such as subtraction, addition, multiplication and dubiously, the character "y" as a free variable that could be solved for, as in basic algebra. Using a Convolutional Neural Network (CNN)--with a setup guided by *Machine Learning Mastery* (Brownlee 2019)--would account for multi-class labeling since we intend to submit an image containing the whole mathematical problem. Therefore, labeling the different characters and symbols would be necessary in order to train our model. Tensorflow provides a 'Sequential' model (Figure 2.1) that allows us to build up layers for our CNN; the last layer being a "dense" layer with 17 unique classes representing the characters needed. The activation for the dense layer is the keyword 'softmax' which approximates the probability of multi-class labeling determining which class the image parsing belongs to.

```
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3),
activation='relu', input_shape=(32, 32, 1)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3),
activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3),
activation='relu'))
model.add(layers.Flatten())
model.add(layers.Dense(64,
activation='relu'))
model.add(layers.Dense(17,
activation='softmax'))
```

Figure 2.1 - The initial set up of our Sequential model (python).

Our data comes from author Xai Nano on kaggle.com (Nano 2016), containing all the images where we split the data into training: '750-count', testing: '150-count', and validation '100-count' as our respective ratios. We locally saved and pruned the data from kaggle because we did not need every character or symbol for a functioning math solver to work. This presented itself to be a very long process with a 3-hour long upload time to a shared Google Drive folder. It was after then that we normalized the images to preprocess and center our images for the model and feed it into the Machine Learning algorithm. Our loss function is a categorical cross entropy function as it best distinguishes differences between two probabilities. The images are fed to the Machine Learning model, which if you thought the upload time was long, then this will give that a run for its money. And then once trained, we can make our predictions with the newly testable, processed images that are set to work with the model, which currently takes in a 32x32 image grayscale to remove any color (See Figure 2.2 for example image). The output should return an array of floating point numbers between the ranges of 0 - 1; 1 indicating that the model is highly confident that this is the label, and 0 being not-at-all likely of the image matching the label. With indexing and getting the max relative values, we can receive our best guess from the

Machine Learning model and then apply it to the math-solving portion of our project.

Figure 2.2 - images from our pruned dataset (5)

For the frontend, we started off in attempt to create a UI on Google Colab, using just the Dash.py library, but instead we decided to adopt React, a JavaScript library for UI, because it was the cleanest way of implementing the canvas HTML and JavaScript in an orderly file. We had moved on from Google Script and Dash.py because they caused a lot of trouble setting up the ML models. The most difficult part of setting up the project with React was getting the model to load from a fetch source (Figure 2.3), since tensorflow.js does not allow local file loading for models. So it was decided to host the model online with Google Buckets, because loading from Google Drive would have caused more code to be written for it to work. From the GCP(Google Cloud Platform) a project was set up in order to fetch the data. A lot of configuration needed to be done in order to avoid CORS (Cross Origin Resources Sharing) errors from popping up, as of now it is set up that anyone who has the link to the model will be able to use it for machine learning predictions for personal use.

```
async componentDidMount(){
    console.log(tf);
    const model = await
tf.loadLayersModel('https://storage.goog
leapis.com/mathsolvermodel/model.json');
}
```

Figure 2.3 - Code snippet fetch source (JavaScript)

Overall, we anticipated our project to follow the general layout in Figure 2.4. We ran into multiple issues--some minor, some extreme--and took our time addressing all. In our results below, we reported our major issues and our attempts at their solutions in order for our project to run smoothly.

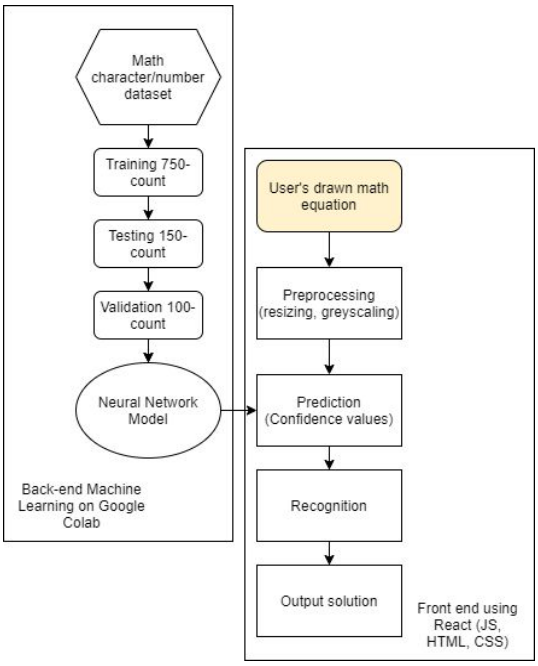

Figure 2.4 General Layout of Project Architecture

Our Results

We were able to get our program to run and return confidence values for a single digit input. A problem we kept referencing back to was how would we split the image so that our program could read an entire equation and single out its characters from left to right. We decided to attempt at splitting the image as a whole then resizing to the dimensions that our machine can analyze. Inspiration on how to split the image came from a tutorial by the "NathanJeanShow" on Youtube (Jean 2017).

Just before implementing the image splitting, we wanted to test for our model's accuracy at detecting and recognizing single characters. We set up our epochs (the number of passes the entire training dataset the machine learning algorithm has completed) to a strong number of 10 assuming an accuracy value of 0.9569 would benefit our machine to recognize characters better. To the team's dismay, Marcos warned us the larger the epoch value, the stricter our machine will be at recognizing an image to look entirely like that of our training dataset. We then get the following observation of predictions:

character	Confidence Values (5 trials)				
0	1.5266 e-8	0.00049 87	0.00020 2	1.15 e-8	3.61 e-16
1	0.00001 289	0.0036	0.9825	0.00049 7	0.00075 4
2	1	1	1	1	1
3	0.02894	0.99969	0.9862	0.00275	0.02432
4	0.00028 8	0.00002 0499	0.02286	0.00018 45	1.19 e-8
5	0.00002 69	6.032 e-14	0.99999 9	3.94 e-8	2.24 e-12
6	1.01 e-8	8.04 e-12	0.00000 25	1	0.00000 2
7	5.43 e-7	9.26 e-16	0.00452 1	0.00000 532	0.00000 7506
8	5.31 e-15	3.95 e-15	6.09 e-14	5.12 e-10	0.00294 5
9	6.373 e-33	2.57 e-21	2.55 e-31	0.01033	5.669 e-24
subtract	1	1	0.73828	0.99999	1
addition	2.59 e-8	3.57 e-10	4.96 e-13	1.687 e-12	3.96 e-22
multiplicatio n	3.545 e-13	3.94 e-7	1.69 e-13	2.96 e-11	3.35 e-12
open parenthesis	0.88574 6	0.00180 1	6.17 e-7	2.73 e-17	0.99997 9
close parenthesis	8.44 e-34	5.86 e-17	3.58867 e-15	1.59 e-17	2.12 e-19
y-variable	1.84 e-16	3.77 e-16	0.00008 6632	2.64 e-11	5.802 e-22
equal sign	0.99998 8	1	0.99993 8	0.00127	0.99999 99

Figure 3.1 - Prediction Table

The highlighted green cells indicate the correct predictions. Very often, many of the mispredicted characters are being read as the characters that show to have the best results. For example, 3 and close parenthesis were commonly being mistaken as 2.

After changing the epoch count to 5, our Google Colab began returning an accuracy value of 0.0587 and that raised major concern.The confidence values were returning the same value for each character every single time it ran (See Figure 3.2).

Figure 3.2 - Snippet of the console after running 3 tests: 0, 1, 4. Array goes from 0-9 as integers and 10-17 as special math characters.

Although fixing the accuracy value in Colab, our machine persisted in only returning the above values and not predicting characters. It is becoming more and more difficult to pinpoint the root of the issue. Marcos and Alyssa messed a bit more with the JavaScript and new values were predicted; however, it did not change after the second trial. A new problem surfaced with the concern of saving the image in the browser. To test, we were saving the files locally, but some tests turned out with empty photos.

We submitted to returning to an older version of our model that was observed earlier. We were receiving

different confidence values, but same as the last goal, we want to improve the scores and also be able to compensate for multiple characters written on the canvas rather than just one.

With more time and effort paid attention to image splitting, our program is now capable of recognizing multiple characters on a single canvas and resizing the individually saved characters to feed to our model. The image splitting method is far from perfect, however it was one of the more feasible ways of getting our recognition model to detect multiple objects drawn on a single canvas. Ideally, implementing an object-edge-detecting system would have allowed us to more accurately segment a larger equation, however this procedure did not appear viable within the deadline. With that being said, we noticed that our model consistently encountered mispredictions particularly when splitting certain digits placed in specific canvas locations. For example, a zero could be split into 3 sections, where the first (left) portion is interpreted as a left parenthesis, the second (middle) portion is interpreted as an equal sign, and the third (right) portion is interpreted as a right parenthesis.

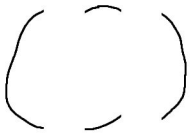

Figure 3.3 - Diagram of a 3-way split for a zero drawn on canvas.

Due to the fact that our recognition model contains a logic structure which tests all four (1-way, 2-way, 3-way, and 4-way) split methods prior to choosing which particular method might have the most "accurate" prediction values, a 3-way split of a centered zero would technically have a greater count of closely "accurate" predictions when compared to a 1-way split – 2 parentheses and 1 equals sign versus 1 zero. In order to mitigate consistent misinterpretations, we decided to further prune our dataset by omitting a few more symbols from our model including the parenthesis, equals sign, and the variable *y* (which was initially included with hopes of implementing algebraic solutions). The evaulatory JavaScript function we used to facilitate the calculation of the implied expression did not recognize the equals sign as a syntactically valid symbol, nor would the function be capable of solving an algebraic problem, hence the exclusion of some of these mathematical symbols actually improved the potential results provided by the *eval()* function by helping reduce both syntax errors and misinterpretations by our prediction model.

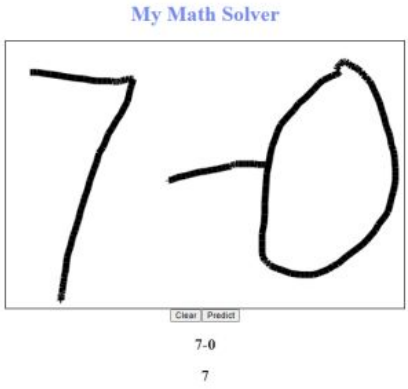

Figure 3.4 - Successful subtraction read.

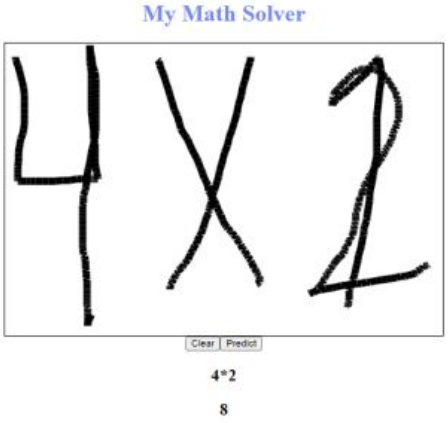

Figure 3.5 - Successful multiplication read.

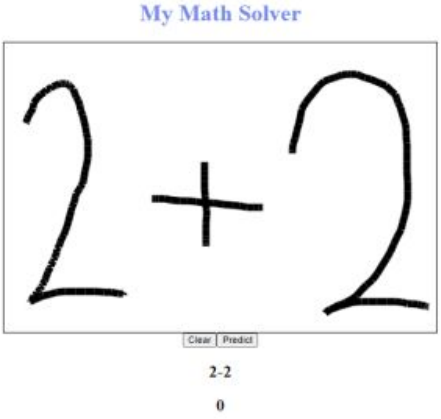

Figure 3.6 - Not-so-successful addition read.

Conclusion

Initially, our goal was to provide users with a web application that not only hits this project's requirements, but also provides guidance with how an equation is solved with step-by-step guidance. Because of a limited availability and short-coming deadline, we were incapable of doing something of that complexity.

Due to the out-of-reach scope that we had pitched in our project proposal, we simplified the project to something more in our grasp. Nonetheless, our inexperience showed to hinder us with long sessions of research and trial-and-error tests when trying to tie everything together.

As shown in our results above, our model is not so successful, and this was with cherry picking some of the data and conforming to the canvas structure so the images would be parsed correctly in order for it to produce a proper result. Many of the issues lie within the model structure on the backend involving preprocessing the images, not using validation or other techniques for improving character recognition. A major proponent in misreads for characters is that we had to conform the data to 32 x 32 dimensional images, thus shrinking/cropping images to a scale where there was significant data loss, say drawing an = sign and reducing its scale it may appear to to the model as a - instead. The image splitting was not perfect but a primitive way of detecting characters individually and independently of each other, but a solution for the time constraint and limiting library resources to demonstrate the illusion of intelligence to the user.

In the end, we are able to achieve more or less mathematical computational intelligence given by character recognition to a certain extent, with achievable examples such as 4 * 2 = 8 producing expected and desired results.

If given the opportunity to expand on this project, all of us would agree that we would want to build and implement our original idea of supporting higher level math and provide our users with a step by step guide on how to solve the given problem versus just posting an answer.

References

Public Data Set

Nano, X. 2016. kaggle. https://www.kaggle.com/xainano/handwrittenmathsymbols

Youtube Tutorials

NathanJeanShow. 2017. “JAVASCRIPT: Splitting Images with Canvas”. Youtube. https://www.youtube.com/watch?v=x06MfsJFdAo

Grayscaling Tutorial

Firdaus, T. October 26, 2017. “3 Ways to Turn Web Images to Grayscale”. *Hongkiat*. https://www.hongkiat.com/blog/grayscale-image-web/

Use of StackOverflow in General

Various forums-

https://stackoverflow.com/

Tensorflow Tutorials

Various pages-

https://www.tensorflow.org/tutorials

CNN Guide

Brownlee, J. 2019. “How to Develop a CNN for MNIST Handwritten Digit Classification”. *Machine Learning Mastery*. https://machinelearningmastery.com/how-to-develop-a-convolutional-neural-network-from-scratch-for-mnist-handwritten-digit-classification/