Recognising Handwritten Math Equations using Machine Learning

Saksham Goyal
Virginia Tech
vt.edu
sakgoy2001@vt.edu

Abstract

In this project, I used a Deep Convolutional Neural Network (CNN) to identify handwritten math equations, and verify if the solution is correct, incorrect or invalid. This can help teachers grade work faster and more accurately, giving them more free time. After training my model, I was able to get an accuracy of 99% although it was overfitting and underfitting many of the symbols and digits at the same time. Overall, my project largely completed what I sought out to do even though there are a couple significant bugs and issues. I plan on continuing to improve this project even after I publish it.

1. Introduction

By using the MNIST dataset that is widely used, I can create an application that allows the user to draw numbers and mathematical symbols onto the screen and form an equation in the given space. This allows an arbitrary length statement to be sent to the Machine Learning model to identify the characters. It will extract the individual numbers and symbols via a bounding box algorithm and normalize that box so that it can be sent to the CNN in a standard 28x28 image. The objective of this project is to have the CNN be able to accurately identify the correct numbers and symbols and give the correct solution to the equation. If given in the drawing, it may also check if the drawn equation is valid or not. For example, check if "x*y=z".

1.1. Background

Currently other similar projects are either perform only digit classification or have low accuracies. My goal is to improve this project beyond what others have achieved and have full math symbol and digit recognition and high accuracy while being able to have quick turnaround time. Another limitation of other similar projects is that they are usually just number classifiers. I am trying to classify numbers as well as math symbols which is not what others are doing. By creating a program and model that can take in more than just a number for classification, I hope to create a program that can in the future take in arbitrary symbols that are drawn and then classify them as needed.

My program also improves other projects in that I have 2 different modes. I can give the answer to an expression, and I can check if a given statement is correct, incorrect, or give an error if the statement does not make sense.

To make my application efficient in terms of being able to classify images quickly from start to finish, I have developed a GUI based application to take in the "images" as drawings on a Tkinter canvas. This allows the user to input the expressions in a streamlined manner. Other projects I have seen use tedious methods would be slower than just grading it manually. [2]

1.2. Motivation

The reason I created this project is so it can help speed up a big headache that teachers and professors have. The speed of grading. Often times they are busy with grading homework which reduces how much time they have for other things. If grading can be sped up or automated, they will reduce their headache and dramatically reduce their workload.

From what I have heard, a lot of time that educators spend is grading work. By streamlining this process and potentially speeding it up, the time spend grading will be reduced and feedback to students can be improved. As a student, I have experienced that many times when I get my work back, all I have received is a score. I do not get an explanation of why its wrong, how to improve, or any resources to explain the reason why I got something wrong. By speeding up grading, I hope that students can get better quality feedback so they can actually learn, instead of just memorizing things for the exam; a failure of the American education system.

In addition to just giving professors more time, the other project I found requires the user to take an image, upload it to the computer running the program, manually set the path to the image and then run the predict function to get the classification. By creating a GUI application, it speeds up the process by orders of magnitude and makes it easier for a user to interact with the program without

Figure 1: The GUI Application developed in Python Tkinter

requiring knowledge of python code or how the internals of the program work.

Approach

The goal of this project is ultimately a classification problem. I want my project to be able to look at an expression or a statement, evaluate it, and output the result. There are 4 parts of this project. The machine learning model itself, creating my dataset for the model to train on as well as normalizing that set, getting the user input, and outputting the result after classifying the input.

I did not want to get the user data from a picture taken from a phone as that negates the point of this project. I needed a streamlined process to be able to input things quickly into the model to classify to save time.

I used TensorFlow to complete this project. Since the only part of this project that I needed the Machine Learning model for was the actual classification of the image, the model was only a small part of the overall application. I chose TensorFlow because it is what I have used before in the homework's for my machine learning class at VT. In homework 4, we used TF to model what would happen by changing the architecture of the model and observe its effects. So, I had a pretty decent understanding of how to use TF. There are many other options I could have used, but I chose TF because it was very efficient to implement and it is what I have experience in.

My model is a sequential model since I am building a CNN. Sequential means that the input data flows only 1 in direction to the output layer i.e., a stack of layers. I used several different types of layers to create my CNN, namely, Convolution layers, Pooling layers, and Dense layers. I chose these layers because my project is an issue of image classification. CNNs are ideal for images.

The input layer consists of a convolution layer which takes in the 28x28 sized image. I then have a set of pooling and additional convolution layers which are then flattened and passed down to a dense layer. The dense layer classifies the image as 1 of 13 classes.

To train this model, I had to configure several hyperparameters. The simplest hyperparameter was the number of classes. Since I am classifying digits and symbols, I just had to specify how many of those I am looking for. There are 9 digits (0-9) and 4 symbols ("x", "+", "-", "/"). However, I was encountering significant issues with the division classification. Because the division symbol looks almost identical to the number 1, my model would rarely classify these correctly. So, I chose to omit the division symbol from the dataset. I ended up with 13 total symbols - The numbers and symbols as above, except division.

For the batch size, I set that to 10. My dataset is not very balanced and so the dataset of symbols is much smaller than the numbers. So, to try and counteract this issue, I made the batch size relatively small. This ensured that backpropagation happened frequently hoping that symbols would not underfit as much given sparse data. Choosing the value as 10 was slightly problematic however. Because this number is relatively small, it means that backpropagation happens more frequently which slowed down training. Thankfully, running it on my local machine allowed me to run this overnight so it became less of an issue.

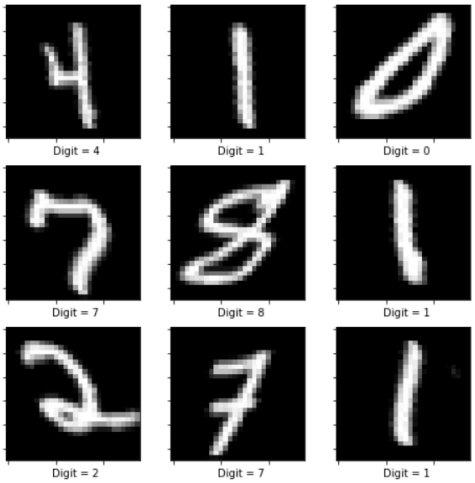

Figure 2: Example of the corrupted shear set

Lastly, I chose the number of epochs based on the accuracies I observed. I set this to 20 epochs. I initially let the training go for much longer as I set it to 50. After training was finished, I observed how when the validation accuracies gradient started to taper off and only get marginal improvement. I observed that the accuracies were still going up even after 30 epochs but because the gain was much lower for each epoch, I did not want the model to overfit any class. The taper started around 20 epochs which is where I decided to stop the training.

For my optimizer, I chose the Adam optimizer. From homework 4, it showed that the RMSprop version was better, but that was after changing the learning rate and other parameters. I was not sure what parameters to choose for my specific scenario so I chose the Adam optimizer since it was the best one by default. Several YouTube videos showing examples of MNIST classification also chose to use the Adam optimizer with default parameters.

After training my model with the dataset I created, I started testing if the trained model was overfitting or not. I found out that my model was overfitting some things and strongly underfitting others. The reason for this high imbalance is because I was using a custom dataset. This caused all sorts of different types of bias. The primary issue is that symbols are not being recognized properly. In the dataset, the 3 symbols only account for approximately 9000 images. My total dataset is 200,000 images. This means that even though the 3 symbols are 3/13 of the classes (~23%), the proportion of the images I have come out to be only ~4.5% which is less than a quarter of the required number of images as symbols to avoid a bias towards numbers. This resulted in my model classifying symbols as the number 4 almost all the time since the plus and multiply symbol are quite similar to the number 4. This issue has been exacerbated by the fact that when I created the model, I also used the "corrupted" versions of the MNIST dataset. This derived dataset has "corrupted" the original set by performing a few different operations on the images to make them different such as rotating the image, changing the brightness, and shearing the numbers, along with many more that I did chose not to use in my set. Since the rotate was applied to the number 4, if it rotates clockwise 45 degrees, it becomes very similar to the multiply symbol. Without rotation, it is similar to the plus symbol. This is why the symbols were underfitting. The combination of significantly sparser data for the symbols combined with the number 4 looking similar to the symbols, it caused a big problem with my model.

To counteract this problem, I designed a solution that tried to mitigate this issue. It does improve the situation, but it still persists. First, I needed to understand what the predicted values were for each class when a digit was provided. So, the predict function of the model allows you to get the probability of each class the model believes the digit to be in. When I passed in symbols, I saw that the class for 4 was the highest (which is why it was underfitting), and the actual symbol was the next highest. So, this means that I could get the 2nd highest predicted value for symbols. But then how would I know when to take the 2nd highest class or take the highest class. This led me to design a simple algorithm before doing the final evaluation of the expression that was input to the canvas. What this algorithm does is first check if the highest-class probability is the number 4 since that is what digits are being misclassified as. If, and only if the probability from the model is 1.000 for the number 4, will it proceed to classify it as actually the number 4. If the model things that there is even a slight chance that the number may not be 4, it will pick the 2nd highest. I chose this algorithm because when symbols were being misclassified, almost every single time, the output was a 4 and the probability for 4 was not 1.000. However, when I did actually draw a 4, the model always gave a prediction of 1.000 for 4. Using that algorithm improved the probability of symbols being classified correctly by a significant margin, but it still did not work as well as I hoped.

This approach does not generalize well because much of the tuning and parameters set are specific to this dataset. Changing the hyperparameters may significantly reduce the accuracy of this application. However, apart from the model itself, the project as a whole I believe is very generalizable. Because I am using a bounding box algorithm to find the positions of the numbers it allows any arbitrary input to the GUI. If one chooses, they could import their own model to the application so they can classify symbols and numbers say in a different language/script. Because the bounding box algorithm depends on each feature that needs to be classified to not touch each other, as long as that is the case, any model will work. If the classifications of something requires small features in the image, some parameters will certainly need to be changed. Because the input of the model is a mere 28x28 pixels, any small feature in the canvas will be eliminated when the image is compressed. Fixing this would require a restructuring of the model completely since many parameters of the filters and depth would need to be changed.

The model visualization is provided in the GitHub page for this project as the image is too large to display in the report.

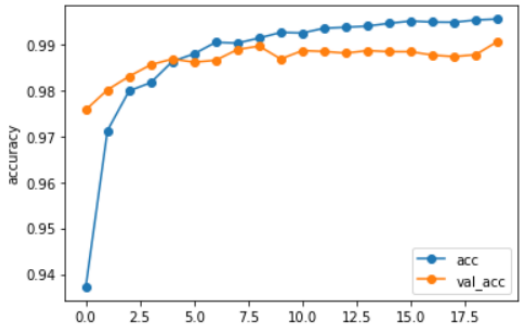

Figure 3: Training accuracies over the validation set

Originally, my project looked quite different. I was going to make the project so that the user can only input 2digit numbers and put them in set positions on the canvas. I did this so that I could train the model on 2-digit numbers and output 2-digit labels for the classifications. This however caused lots of problems because if the user can only place the 2-digit numbers in set positions on the canvas, if the user did not adhere to that rule, the classification would fail. Also, creating the dataset to support 2-digit numbers would be much more complicated as the combination of possible ways the user can input a number drastically increases with 2 digits. This slowed my progress significantly until I found that there is the OpenCV library that has functions to get bounding boxes from an image.

This revelation made my project easier to complete and significantly cut down on the amount of code I would need to write. The limitations to my original project opened up and it became much more generalizable. I could now give arbitrary length numbers on the canvas and get accurate results with a model that would perform better.

I also wanted to create a GUI for my application. This also caused problems because Google Colab does not support Tkinter GUI windows (or any GUI window for the matter). To continue making the GUI. I had to switch to running the program locally. But that posses another issue. My computer is not powerful enough to run the training without it taking significantly longer and slowing my progress even more. So, I had to find a way to train the model on Colab, and then run it on my machine. That is when I found that TF has a function to save the model after training it and load it back in later. I used this functionality to let the training happen in the background while I was developing the Tkinter GUI app.

Originally, I was also going to have to create my own function to evaluate the statement/expression that the user entered. That would have been extremely tedious and time consuming. However, python came to the rescue and provided the *eval()* function. It takes in a string, evaluates the expression, and outputs the result. That is exactly what I needed. Instead of creating a long and complicated function, python build it right in.

2. Experiments

To judge how well the model performed, the accuracy was used to measure how well each image was classified. However, the main function that needed to be minimized was the loss of the classification. A model that misclassifies an image slightly is obviously better than a model that gets it completely wrong when it is wrong. That is what the loss function calculates. So, accuracy is used to see a general idea of how accuracy it is but the loss function sees the precision of the model. They are mutually exclusive so one cannot be derived from the other. After training my model, I believe the model to be accurate given the accuracy is >99%, however the precision of the model is very poor.

The accuracy is calculated by dividing the number of correct classifications to the total number of classifications made. This number is calculated at every step (once before backpropagation). The loss function is calculated in TensorFlow with the *categortial_crossentropy* loss function. This function is used when the model is classifying data into more than 2 classes. The labels are provided in one-hot encoding. This lets TF clearly understand what the labels is in reference to what class it goes to. The calculation of this categorical cross-entropy function simply multiplies the correct label of the image to the log of the output of the model. The output is then scaled according to the rest of the probabilities for each class so they sum to one [1]. This provides an intuitive way to calculate how bad the classification is when misclassified rather than just using accuracy which just tells you if it was right or wrong. This loss function is perfect for a multi-class classification model. This loss function allows the backpropagation algorithm to see how much to adjust the weights of the model in proportion to how close it was to getting the correct answer.

The training, validation, and test batches have been created using TensorFlow's *image_dataset_from_directory* library. This function allows you to give it a directory containing the images correctly labeled and give it the seed parameter that tells TensorFlow how to split the data into the 3 batches. Since they are created at runtime, and the seed for random shuffling the data is a set value, anyone who wishes to create the same splits can do so without changing any parameters. The random seed is set to *134* which was a number I picked arbitrarily.

I set the training split to 90%, validation split to 10%, and the test split to 99%. I did this because I wanted to evaluate the performance of the model over all of the images but not to use all of them during training. That's why the test split is basically all of the images. I wanted to avoid using such a low split for the validation, but because my data for the symbols was sparse, I needed to use more training images

to reduce the underfitting that was already happening with symbols.

2.1. Results

After training my model and finishing the Tkinter GUI application, I started to test the model by inputting different expressions and statements into the app. This is when I found 2 major issues with my model and application. Even though the model accuracy was very high (>99.8%) and the loss was very low (~0.005), the plus symbol and the multiply symbol were being misclassified as the number 4 about 90% of the time. The 2nd major issue was my GUI application not working as intended. After the user has drawn something to the canvas and evaluated the function, the canvas clears visually, but for some reason, drawing something again will send a combined version of all previous drawings in addition to the current one. *So, the only way to do multiple classifications is to re-run the program every time you want to draw something.*

The fix for the misclassification is because my dataset I used was not conducive to classifying the symbols and the number 4 correctly. Because I used the "rotated" subset of the MNIST corrupted dataset, it made the number 4, the multiple symbols and the plus symbol too similar to each other for the CNN to work. I also did not have nearly enough data for the symbols for it to work correctly.

For the GUI bug to clear the canvas, I have narrowed down the issue to me not resetting some variable when I click the clear button. That is because re-running the GUI application will work as expected which means that some variable that runs in the GUI is being used in multiple runs without being cleared of the previous data.

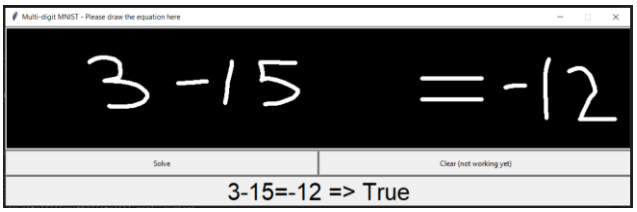

Figure 4 How the GUI will validate the input when it works

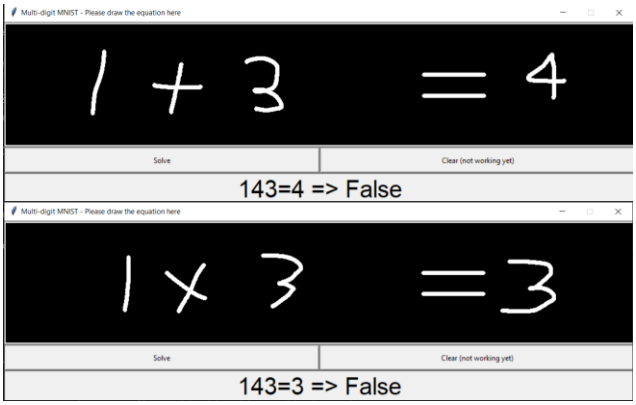

Figure 5 How the model misclassifies symbols

3. Availability

My code is available here: *https://github.com/TheVaccin3/HandWritten-Calculator*. I made this code available on GitHub and added the GNU GPL License to my project so people can freely use my project and improve it if they choose to do so. I chose to use this particular license as its widely regarded as the best copy left, most accessible, and open-source friendly license. I do not want to prevent people from using my code although I need derivatives of my work to also continue to use GPL. With that in mind, GPL was the best choice. I created the license file that was auto generated by GitHub when I created the repository for this project.

I will also be uploading a compressed version of the dataset I am using. Since I had used a collection of different datasets and combined them together, I want others to have the same access to the dataset I created. But, since others will have to spent time and resources to train this model, I have also opted to save the model weights and parameters to a file that can be read by TensorFlow. Since my trained model is published with the dataset, it lets others more easily reproduce my results. By uploading all of the work and progress I have made so far, along with code snippets of how I created the dataset, others can use what I did to understand how to create their own dataset that will still work with the model I have created and any arbitrary weights model can be applied to this TensorFlow CNN.

4. Reproducibility

I have used the GPL license for my project meaning that the code is freely available. However, I have also included my dataset in the repository meaning that the dataset is also covered under that license. I did this to ensure that everyone has equal access to the dataset I created so that others can recreate my exact project.

TensorFlow allows you to save the model into a "checkpoint" of the model. This saves all of the architecture features, the weights of each of the layers, and all of the

parameters that TensorFlow used to train the model. This checkpoint is then saved into a folder. There would be no point in being able to save it without pulling it back, to TF also includes a function to load an existing model so you can directly classify images without having to retrain the model every time you run the application.

The model that is saved can be chosen by code. You can choose to either save the best model, or the most recent model during training. I chose to save the best model since the validation accuracy can always go down and we want the best accuracy possible without overfitting.

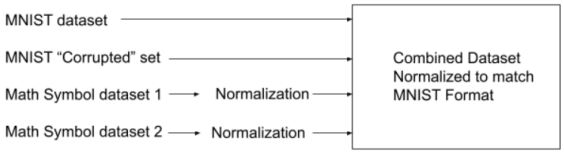

Figure 6 showing how the dataset was created

References

[1] Raúl Gómez, May 23 2018, *Understanding Categorical Cross-Entropy Loss, Binary Cross-Entropy Loss, Softmax Loss, Logistic Loss, and all those confusing names*, www.gombru.github.io/2018/05/23/cross_entropy_loss

[2] Rohan Kurdekar, 2021, *Kaggle Handwritten basic math equation solver*, https://www.kaggle.com/code/rohankurdekar/handwritten-basic-math-equation-solver/notebook

[3] nathancy, Sept 20, 2021. *Extract all bounding boxes using OpenCV Python*, https://stackoverflow.com/a/60068297/7230903

[4] Deng, L. (2012). *The mnist database of handwritten digit images for machine learning research*. IEEE Signal Processing Magazine, 29(6), 141–142.

[5] CodersHubb, March 1 2021, *Create the simple paint app using python*, https://www.codershubb.com/create-the-simple-paint-app-using-python/