AI Final Project Report

Christopher Lackowski
Isabelle Byrne

Link to Project Repo - https://github.com/Clackowski/SymbolSenseAI

Problem and Motivation:

As STEM students, we frequently find ourselves taking handwritten paper notes in our mathematical courses, while opting for typed digital notes in all other subjects, such as computer science and biology. This choice is typically due to the difficulty of formatting digital mathematical equations and symbols in a lecture setting, and students typically also don't have time outside of class to convert their notes into a clean and readable digital format, such as LaTeX. The problem with having notes located in different places (paper versus digital) is that it can cause organizational problems, and paper notes are at risk of being damaged and lost forever. To solve the problems that un-digitized notes cause, we propose a software that will take handwritten math notes as an input, classify individual numerical and mathematical symbols, and output a digitized version of the notes in a file with embedded LaTeX for clean and consistent notes.

Dataset Used:

In order to properly train our model, we researched many different websites to find a dataset that included images of handwritten letters, numbers and mathematical symbols. We were able to find a dataset called HazyV2, which contains images of handwritten LaTeX symbols which include all of what we need. There are images of letters, digits (0 through 9), math symbols and also many other symbols that LaTeX supports. This dataset was found on kaggle.com and we believe it provides the right amount of data to use for training our model to be able to predict every type of character.

Initially, when learning about Tensorflow and all of its capabilities for creating and training a model, we developed a proof of concept model using its built in "MNIST" dataset which includes 70,000 28x28 images of handwritten numbers on a white background. We used this to create a model that is able to predict a digit (0 through 9) at a very high accuracy rate. Tensorflow provided a very simple way to download this dataset and load it into the program whenever we wanted to train a model. We performed research to see if Tensorflow provides other datasets that we could use for training our math equation predictor and saw that there were other sets that could be used. For example, there is one dataset called "EMNIST" which includes 28x28 images of all the digits as well as uppercase and lowercase letters. We ended up using both the "MNIST" and "EMNIST" datasets for training our model allowing it to predict handwritten digits and letters.

The HazyV2 dataset that we had initially planned to use for training our model, was much more complicated to use than we had initially planned for it to be, so we were unable to use it. However, being able to classify both letters and numbers was a big task in itself, so we felt it was understandable to lessen the scope of the project.

Problem Definition:

Our initial project plan was to develop and train a model to be able to predict the letters, numbers and mathematical symbols in images and output them to a latex document which is formatted to match what was on the input image. Below is an example of an image that would be inputted and an output image according to our initial plan.

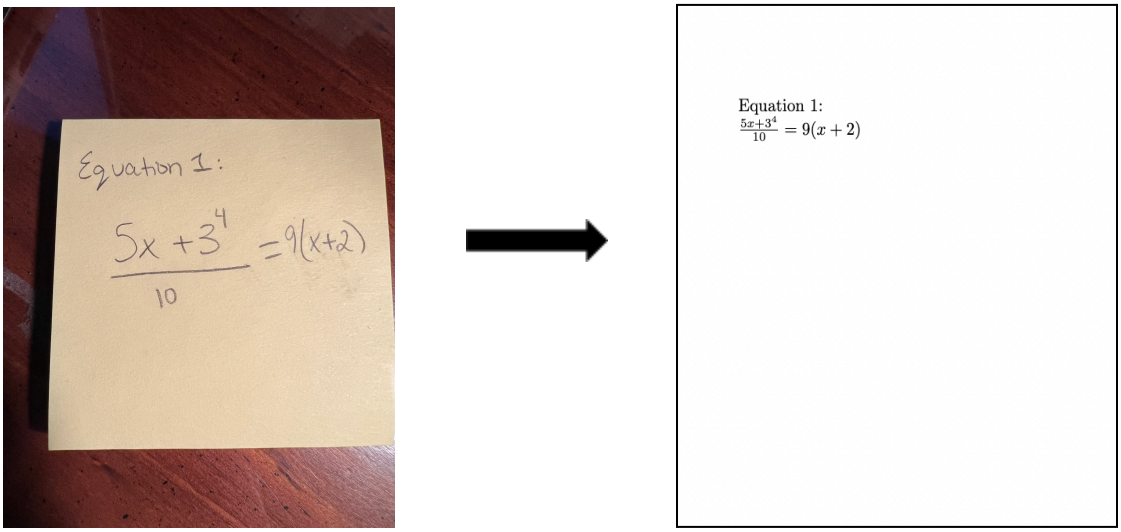

Figure 1: Example of initial plan for input and output

However, after having difficulties using the HazyV2 dataset and being limited to using the "MNIST" and "EMNIST" datasets, we decided to focus on training a model to just classify letters and numbers. Also, being able to convert the symbols into a LaTeX format was a lot more difficult than we initially thought, so we reduced the scope to only classification of text and numbers in a given input image.

The inputs into the program are images in a png format that contain text and numbers. These images can be passed into the model and a prediction will be outputted describing the letters or numbers in the image. So, the actual output of the program that we were able to achieve is text in the console describing the letters or numbers that were found to be in the input image. An example of this can be found below.

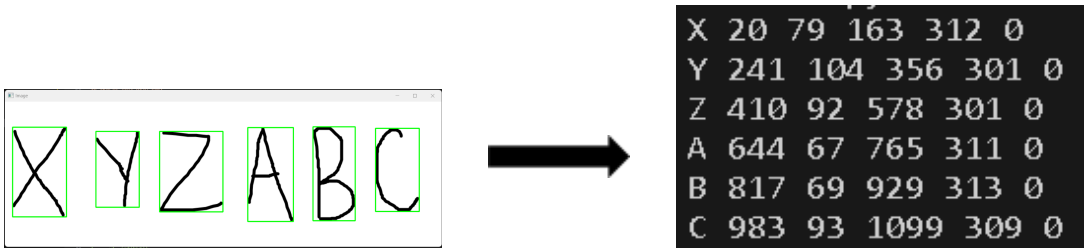

Figure 2: Example of actual input and output

These predictions are able to be made because of the model that we have trained. We used the "MNIST" and "EMNIST" datasets to train the model to be able to recognize letters and numbers in images, so that when an image is provided, it can relatively accurately predict what

characters are shown. The output shown relates each individual character to its corresponding bounding box location in the original image.

Algorithms Used:

When developing our model, we utilized Convolutional Neural Networks (CNNs) to be able to predict letters and numbers in a given image. CNNs are instrumental in image detection of characters due to their ability to automatically learn hierarchical representations of visual features from input images. In the context of optical character recognition (OCR), CNNs excel at capturing patterns, edges, and textures within images, making them highly effective for detecting and classifying letters and digits. The convolutional layers in a CNN use filters or kernels to convolve across the input image, allowing the network to detect features like curves, corners, and strokes that are crucial for recognizing characters. The subsequent pooling layers help reduce the spatial dimensions while retaining essential information, enabling the network to focus on increasingly complex and abstract features as it progresses through the layers.

Also, CNNs enable the creation of translation-invariant features, making them robust to variations in position and orientation of characters within an image. This property is particularly beneficial for handling different fonts, styles, and alignments of letters and digits. The fully connected layers at the end of the CNN architecture then integrate these learned features to make predictions about the presence and identity of characters in the input image. Through training on labeled datasets, CNNs adaptively adjust their weights to recognize diverse patterns associated with letters and digits, making them super important in the field of image-based character detection and recognition.

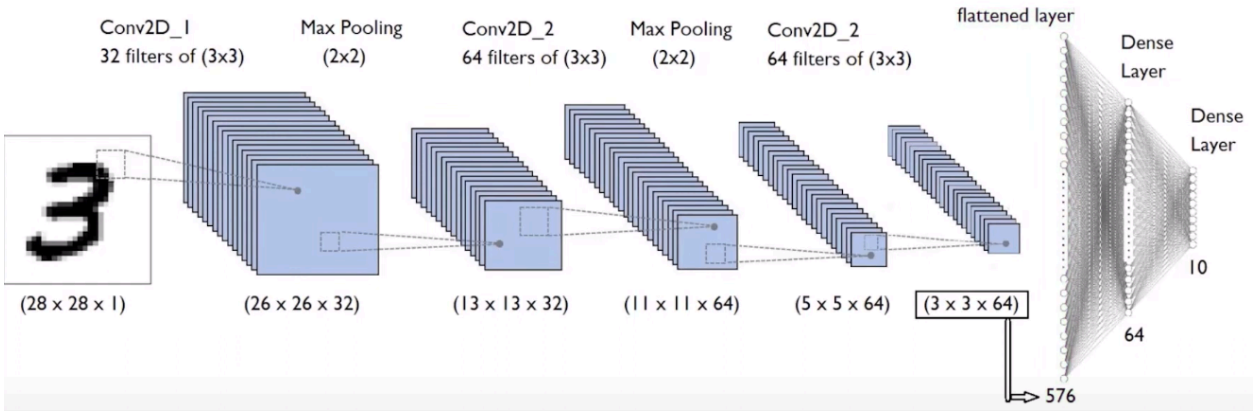

Figure 3. CNN Diagram

After extensive research, a CNN that follows the above structure was found to be the most promising.

Experimental Setup:

In order to set up our CNN model to be trained, there were 6 steps that we created. These steps are loading the dataset, creating the model, compiling the model, training the model, evaluating the model and saving the model.

The first step consists of properly downloading the necessary dataset that we want to train the model with and parsing all of the values that it contains. The dataset that we chose, provides many images for training and the associated label for each image which describes the character in the image. Therefore, when loading all of the data from the dataset, we are taking these image and label pairs and splitting them into large arrays, so that the data can be used when training the model and also testing the accuracy of the model. However, in this step there is some preprocessing that is done on the images. Since the images need to be normalized and made to be equal sizes, the images are reshaped and then able to be stored in the proper lists.

In the second setup stage, we had to create the model. This was done by using Tensorflow's built in library for creating models. The model was instantiated using the built in constructor and then various layers were added to it. The first layer is a 2D convolutional layer with 32 filters, a 3x3 kernel size and an input shape of 28x28. A max pooling layer was then added with a 2x2 pool size. This was followed by another convolutional layer with 64 filters and 3x3 kernel size, another max pooling layer and then a third convolutional layer. Lastly, multiple dense layers were added to process the flattened features from the previous layers and make the final predictions.

Next, we had to compile the model which basically configures the model for training. It sets up the optimizer, loss function and metrics used in the training process, so that the model knows how it should learn. The optimizer we used is "Adam" which combines the benefits of two different extensions of stochastic gradient descent. Our loss parameter was chosen to be "categorical_crossentropy" since it is generally suitable for scenarios where each sample can be assigned to exactly one class out of multiple classes which is perfect for our project. Lastly, the metric we chose was "accuracy" which simply allows it to measure how many predictions match the true labels, expressed as a percentage. We decided that these 3 compilation parameters were the best choices for our model.

Training the model is the next step in setting up the program. We utilized Tensorflow's built in fit function which is used for training a model. When this function is called, the model undergoes training for the specified number of epochs, updating its weights to minimize the specified loss function. Once training is complete, we decided to set a flag indicating that the model being trained has been trained, so that the process can not be repeated on an already trained model.

In order to test the accuracy of the model, it goes through a testing process. We used Tensorflow's built in evaluate function which is perfect for figuring out how accurate the model is. In order to do this, we passed in all of the testing images and labels that were downloaded, parsed and normalized in the loading data stage. This allows us to then print out to the console a proper accuracy percentage. We were able to get a consistent accuracy of 99.02% for the model.

Lastly, the model needs to be saved in order to use it. So, our code uses Tensorflow's built in save function to create a file for the model and save it to the project's main directory. Once all of these steps have been completed, the model is ready to be used to make predictions on input images.

After running through all of the steps to develop and train a model, we expected that 99.02% of the symbols in a single image would be predicted correctly. Since, the accuracy of the model was calculated using a large number of images from the datasets, the sample size should be large enough to provide a relatively correct estimate for accuracy. So, the accuracy given should be the same accuracy that we expect in our symbol prediction rates. To accompany our MNIST model, we found a pretrained EMNIST model (zmandyhe on GitHub) to use for alphabetic characters.

For text recognition from an image, we decided to use pyTesseract to create an OCR model for our project. We created a class to extract individual characters from a given input image, which then is sent to our CNN model to predict the character. By using Tesseract to identify individual characters, we are able to recognize and classify handwritten characters/text from an image, bringing us closer to our goal for the project.

Oversights:

Initially, we had completed a large amount of background research to understand the scope and difficulty of this project. We learned a lot about Neural Networks and dove deep into many of Tensorflow's available libraries and functions. We even started out by developing a proof of concept to test our idea before fully implementing it. This proof of concept involved developing and training a model to be able to predict a single digit 0 through 9 in an image. We developed this project and were able to get it working very quickly, so we assumed it would be relatively simple to expand the dataset to include images of letters and mathematical symbols. However, we found this to be a lot more difficult than we initially thought it would be.

One major issue came from using Tesseract, which is an open-source character recognition engine. Currently, there is no support for images containing handwritten characters, as the model only detects computer generated text. After completing additional research, we found the option to integrate a LSTM, or Long Short-Term Memory. Doing this, however, was not possible as integrating a LSTM model is not straightforward and is extremely labor intensive. Due to our lack of resources and limited time constraints, we could not have access to a trained LSTM model which generalizes human handwritten text.

Despite these issues, we were still able to use Tesseract to extract bounding box information for individual characters. So even though we could not classify them right away, we were able to split up all the characters in an image and then utilize the Convolutional Neural Network model to classify the handwritten characters.

Results:

The results of our program can be split into 2 parts. The first is the results of the Tesseract-OCR which was used for splitting the characters in an image into individual characters and showing the bounding boxes for each character. As shown below in Figure 6, the Tesseract was able to split up characters relatively well.

The second part of the results of our program was to classify the characters that were split up by the Tesseract using our Convolutional Neural Network model. The model that we created and trained was able to achieve 83.19% accuracy from original image input to the final output. This value was calculated by using a large set of images and labels and calculating a ratio of correct to total predictions. In Figure 4 and Figure 5 below, cases where the model correctly and incorrectly classified the characters in the images are shown.

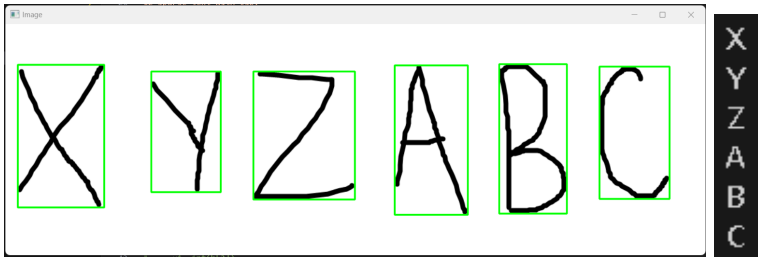

Figure 4: Example of identifying correctly

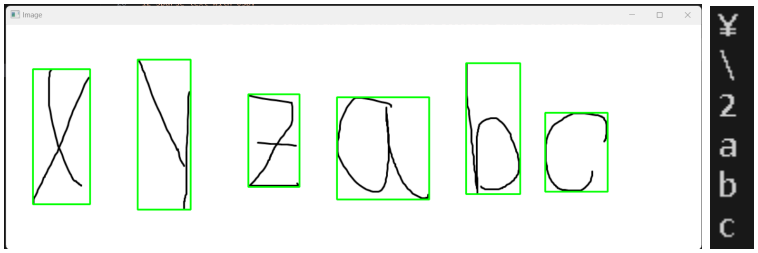

Figure 5: Example of identifying incorrectly

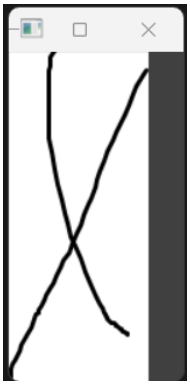

Figure 6: Example of a character being extracted from the original text

Analysis of results:

In order to properly analyze our final results, we can utilize the accuracy score that was provided when testing the trained model. The final accuracy rate that our model was able to reach was 83.19% which is relatively reliable. In order to improve the accuracy of our model, the first step would be to use a larger dataset. Having a larger dataset to train a neural network generally leads to increased accuracy for several reasons. A larger dataset provides the model with a more comprehensive and diverse set of examples, allowing it to learn a wider range of

patterns and variations present in the data. This increased variety helps the model generalize better to unseen instances, enhancing its ability to recognize and classify features in new images accurately. Additionally, a larger dataset helps mitigate overfitting, a common challenge in machine learning, where a model may become too specialized in capturing noise present in a smaller training set. With more data, the model is less likely to memorize specific instances and can instead focus on learning representative features, resulting in improved performance on unseen data.

Implementing regularization techniques would have been another effective way to enhance the accuracy of the CNN. Regularization methods prevent the model from becoming overly complex and overfitting the training data, which can occur when a model learns noise or outliers in the dataset rather than the underlying patterns. Common regularization techniques include dropout and weight regularization. Dropout involves randomly deactivating a fraction of neurons during training, forcing the model to rely on different pathways and reducing the likelihood of co-adaptation among neurons. Weight regularization, such as L1 or L2 regularization, adds penalty terms to the loss function based on the magnitude of the weights, discouraging overly large weights and promoting a simpler model. By incorporating regularization, the CNN would have been able to better generalize to unseen data, resulting in overall improved accuracy for the letter recognition.

Additionally, we were unable to achieve being able to format text into a LaTeX file because symbol classification took a lot longer than expected. Symbol classification is a computationally intensive task, and the intricacies of text formatting require super high levels of accuracy and precision. As a result, we were unable to train the model to format the symbols it recognized into a LaTeX document.

Future Directions:

As described above, we were unable to fully complete our initial goal of allowing users to input images of math notes and have it be converted into a LaTeX document, however a lot of useful work was completed. Even though our work did not completely follow our initial plan, we learned a large amount about some popular libraries used for machine learning coding and gained a pretty deep understanding of Convolutional Neural Networks. We spent around 50-60 hours each on this project and spent a large amount of that time researching machine learning libraries and ways that other people have implemented similar problems. There is still a lot more work that can be done on this project and we plan to continue looking into this concept in the future after completing this course.

If we were given more time to work on this project, we would like to look into classification of digits, letters and also mathematical symbols. As of right now, our model is only able to categorize letters and numbers in images which is useful, but does not completely satisfy our initial overarching goal that we proposed. Developing a program that is able to accurately predict multiple characters in an image took longer than expected, so given more time it would make sense to work on also allowing the model to predict mathematical symbols. This could be

done, by integrating the HazyV2 dataset or any other dataset that contains handwritten images of these symbols into the model, however we were unable to get this working due to the more complex structure of the data files. Then, the next step after that point would be to work on formatting. Formatting takes into account the position of the characters and symbols in the image, so we would need to provide our model with a lot more data in order to get this close to working. This is something that we are very interested in further exploring in the future.

Additionally, if we were able to get to this point of being able to properly classify and format numbers, letters and symbols, it would be beneficial to make this program into a website or app that other people can use. Once this product is fully developed and has high enough accuracy, we believe that there would be a lot of people interested in using it for the purposes that we described above. As college students, being able to translate handwritten math notes into a well formatted document on our computer would save us a lot of time.

Citations:

Thoma, M. (n.d.). *The HASYv2 Datase*.

Tesseract documentation. Tesseract OCR. (n.d.). https://tesseract-ocr.github.io/

Nielsen, M. A. (1970, January 1). Neural networks and deep learning. http://neuralnetworksanddeeplearning.com/

EMNIST model:

https://github.com/zmandyhe/image-classification-mnist-emnist-letters?tab=readme-ov-file