Data Structure and Algorithms: Final Project

Research on Handwriting Character Recognition using Machine Learning

Joon Kang, Jackie Zeng, Mari Kang

Introduction

With the recent improvement in technology in stylus and handwriting in digital formats, there has been a significant increase in handwriting in electronic formats. Proportional to the increase of these technologies, there has also been some new methods and improvements in recognizing these digitally written characters and converting/detecting them as well. In this project, we will discuss the methodology and implementation of handwriting character recognition in python using Convolution Neural Networks. After the explanation of the background information and research on this topic, we will implement the system in python using a dataset in the form of a csv that contains 372450 images of handwritten alphabets and analyze the results. We will also perform different experiments with the created model to find new insights and present potential extensions to the project.

Background Information

Handwriting character recognition can be done in a variety of ways, even with machine learning. There are two types of image recognition - offline and online. Offline recognition uses optical scanning from a camera or an image. Online recognition tracks the movements of a pen tip from a computer screen surface. These types of systems handle formatting, and perform segmentation of characters to find plausible characters. In this paper, we are focusing on the offline recognition method.

The offline recognition method has three parts to detecting characters. First, the characters have to be extracted from the text that the user has scanned. The character extraction step is followed by the character recognition. This is done by machine learning techniques by training the engine and testing for individual characters. Lastly an addition to the offline recognition method is the feature extraction. With the characters that have been recognized from training the user can choose the properties that are important, which allows the recognizer more control over the properties that are used in training and recognition.

After training, we compile and fit the model using optimizing functions. Optimizing functions are indispensable because machine learning algorithms are to solve mathematical problems like curve fittings in the most efficient manner. For example, the optimization algorithm that is utilized in this project can be boiled down to the RMSprop and Adagram algorithms. Both RMSprop and Adagram are first order differential algorithms, where they can find the quickest predicted path to the next step.

Before we move into understanding neural networks, we first need to understand multilayer perceptrons. MLPs are a class of feedforward neural networks. They consist of at least 3 layers of neural networks, which are the input layer, hidden layer and an output layer. Each of the neurons that are in the MLPs

are a node that uses a nonlinear activation function. This type of neural network can distinguish data that is not linearly separable, like images and complex data structures.

The theory behind multilayered perceptrons can be divided into 3 parts. First, since the perceptron was modeled after the biological firing of the neurons, the activation function was developed to model the frequency of action potentials, which defines the output of that node given an input. Second, MLPs have multiple layers as the name suggests. Since all nodes of MLPs are connected to all nodes in the other layers, every node in the perceptron has a weight assigned for a connection to the next layer. This weight is changed consistently as the neurons learn in training, based on the amount of error that the machine has outputted. This is a form of backpropagation, in a generalization of the least mean squared algorithm. The change of weight in the neural network can be presented like so:

$$\Delta w_{ji}(n) = -\eta \frac{\partial \mathcal{E}(n)}{\partial v_j(n)} y_i(n)$$

Figure 1. The change of weight in the neural network

Where y is the output of the previous epoch and n is the learning rate.

In this program, we will use neural networks that are based off of this multilayered perceptron as a part of the character recognition step in offline recognition. The most common neural networks used in these types of training include the CNN (Convolutional Neural Network).

Convolutional Neural Network

A Convolution Neural Network is an algorithm that works with images and assigns importance (weights) to different types of objects/aspects in the images that it is processing. Because an image may have spatial and temporal dependencies that the neural network can capture, there is a reduction in the number of parameters - in short, the network is able to analyze images in a more sophisticated way. With images that has color channels, heights and width, the role of convnet which reduces the images to a form that is easier to process without lossy data is very important for a sophisticated analysis.

A CNN is a type of multilayer perceptron. This means that each neuron in a layer is connected to all the neurons in the next layer. This tends to make the algorithm overfitting in various situations, so the algorithm implements ways to penalize the training of the algorithm by trimming or weight decay. Because CNNs are capable of optimizing using patterns of increasing complexity within the data, they are on the lower extreme in terms of connectivity and complexity.

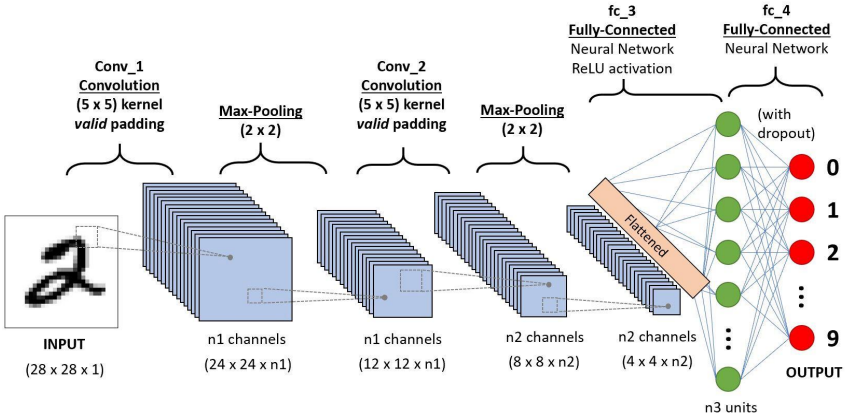

Figure 2. A diagram of multilayer CNN

Like most multilayer perceptrons, the network functions with layers. The input layer, the hidden and the output layers are the general forms of these layers. The input layer takes a tensor as its input that has a shape of number of inputs by height by width by the channels. The convolution layer, which is a part of the hidden layers, is responsible for carrying out the convolution operation which creates a convolved feature by multiplying a defined matrix to parts of the image. This defined matrix is called a Kernel/Filter, and it strides throughout the entire image, performing a matrix multiplication every stride. This process is used because it can extract high level features like edges from the image it is processing. A more detailed process of the CNN is presented below, along with the direct implementation to systematically solve this problem.

Model/Algorithm Overview

To create our handwriting recognition model, we synthesised what we had learned from research into various kinds of CNN models. The following figure gives an overview of the layers in our handwriting recognition model.

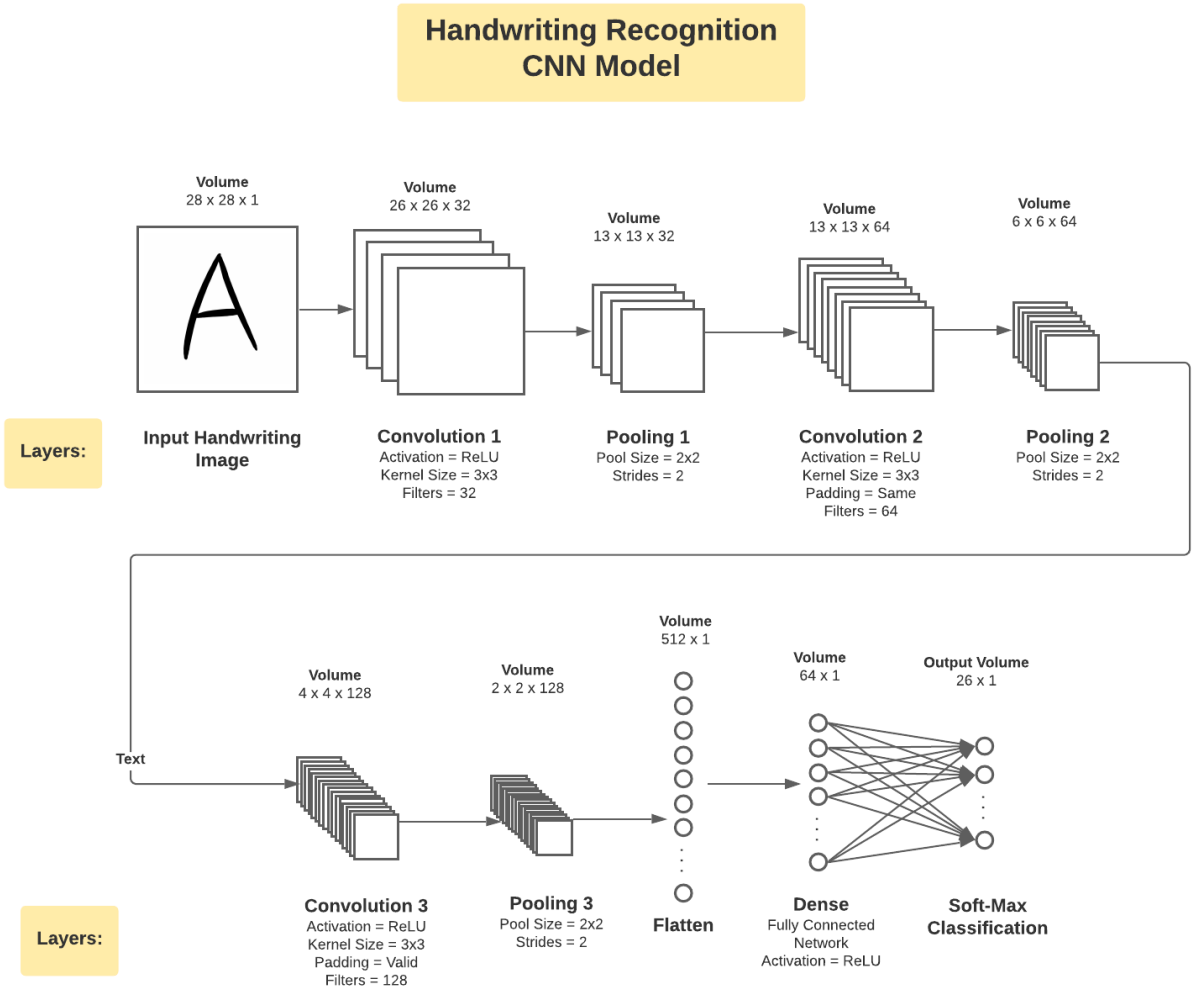

Figure 3. A diagram of our CNN model

Like explained in the background information of both MLPs and CNN, this model has three types of layers: the input layer that takes raw input as an image of a handwritten uppercase letter, hidden layers that take input from another layer and pass output to another layer, and an output layer that predicts the letter. The hidden layers consist of three cycles of convolution and pooling that extract key identification features. The filters are flattened into a column matrix at the and key features used to identify the letter using the soft-max function for classification.

A more detailed explanation of each layer of the model can be seen In the model implementation walkthrough below.

Model Implementation

The below section walks through the part of our code that creates the model and explains the code as well as any design decisions we made.

Model Initiation

```
model = Sequential()
```

We use a sequential model where there is a stack of layers and each layer has exactly one input tensor and one output tensor.

Feature Extraction

Convolution and Pooling Round 1

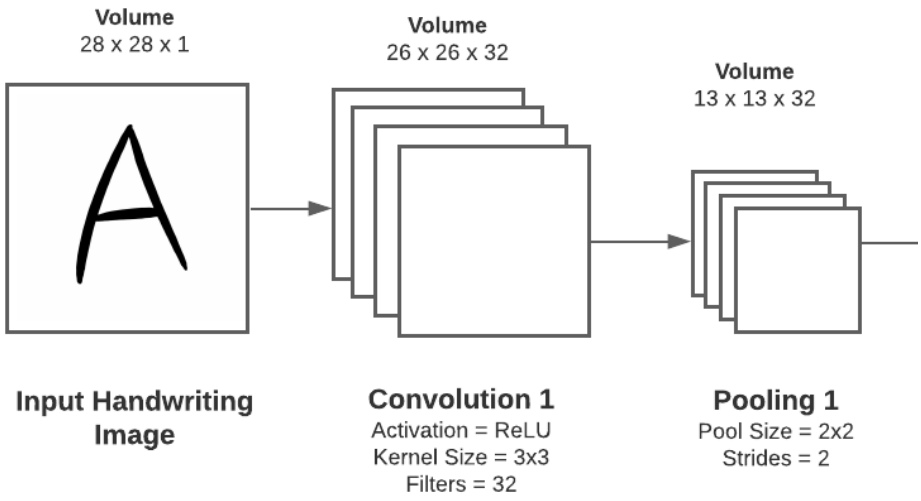

Figure 4. A diagram that depicts the first round of convolution and pooling.

```
model.add(Conv2D(filters=32, kernel_size=(3, 3), activation='relu', input_shape=(28,28,1)))
```

The initial convolution layer takes the 28x28 handwriting image as an input. It then creates 32 filters/kernels, which filters the image section by section by multiplying each 3x3 section by the following matrix and saving the sum.

$$K = \begin{matrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{matrix}$$

This operation extracts the high-level features such as edges from the input image. Passing the kernels over the initial image reduces the size of the image into 26x26. In our case, using a total of 32 kernels results in a output volume of 26x26x32. The number of filters used is typically chosen based on complexity of tasks where more difficult tasks require more filters. Additionally, more filters are used as layers get deeper.

After the kernel, we process the result using an activation function. An activation function in a neural network defines how the weighted sum of the input is transformed into an output from a node or nodes in a layer of the network. For our model, we decided to use the rectified linear unit (ReLU) activation function.

```
ReLU = max(x, 0)
```

This function ensures that all values in the tensor are positive.

```
model.add(MaxPool2D(pool_size=(2, 2), strides=2))
```

After convolution, we condense the data using a pooling layer. We chose Max Pooling, which returns the maximum value from the portion of the image covered by the 2x2 pool kernel, because it acts as a noise suppressant by removing noisy activations and de-noises with dimensionality reduction. The stride length specifies how much the pooling window needs to move for each step. A kernel of 2x2 with a stride length of 2 will reduce the output volume to 13x13x32.

Convolution and Pooling Rounds 2 & 3

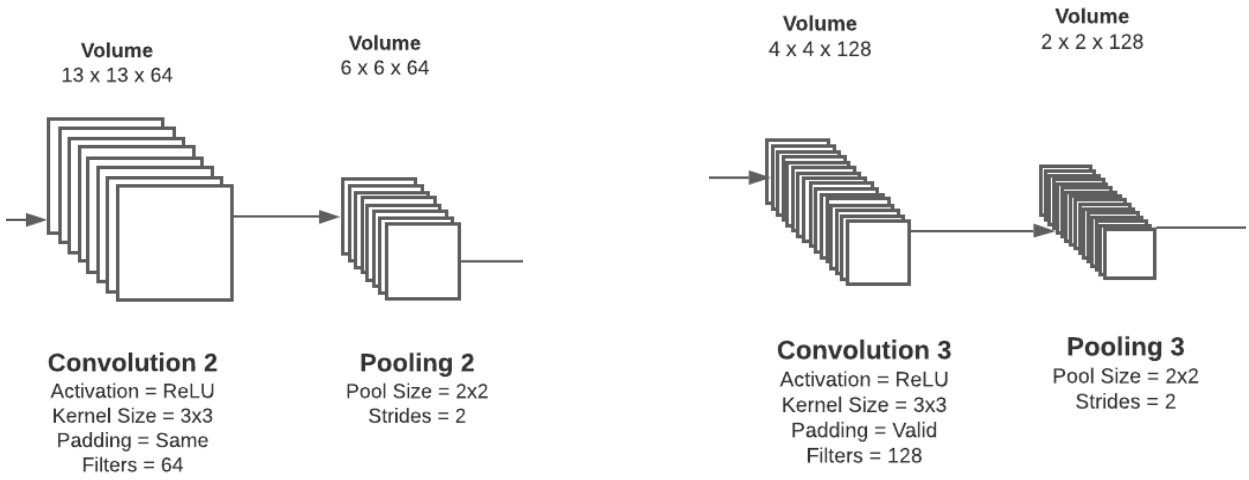

Figure 5. A diagram that depicts the second and third rounds of convolution and pooling.

```
model.add(Conv2D(filters=64, kernel_size=(3, 3), activation='relu', padding = 'same'))
model.add(MaxPool2D(pool_size=(2, 2), strides=2))

model.add(Conv2D(filters=128, kernel_size=(3, 3), activation='relu', padding = 'valid'))
model.add(MaxPool2D(pool_size=(2, 2), strides=2))
```

We repeat the convolution and pooling process two more times to capture more low level detail. These two rounds are the same as the first round except for the padding parameters for the convolution layers. By examining the volume of the layers above, you can see that `padding = valid` reduces the spatial dimensions and `padding = same` preserves the spatial dimensions by padding the edges of the image with zeros.

Classification

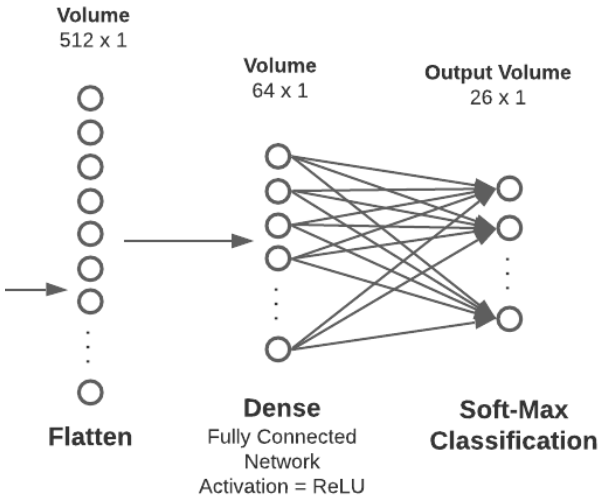

Figure 6. A diagram that depicts the classification part of CNN.

```
model.add(Flatten())
model.add(Dense(64,activation ="relu"))
model.add(Dense(2,activation ="softmax"))
```

To build the classification section, we first flattened the volume into a one dimensional vector of size 512x1. Then, we used the dense layer to further reduce the spatial dimensions and create a fully connected network. Each neuron in a dense layer receives input from all neurons of a previous layer. Finally, for the output layer, we used another dense layer to reduce the dimensions to 26x1, matching that of the number of letters. We chose to use the softmax activation function because of its high performance with multiclass classification.

Algorithmic Complexity of the CNN

The algorithmic complexity of the convolution neural network can be roughly estimated by dividing up processes to the number of neurons. Our model has 26 images with size 28 by 28 by 1, which is considered

the input. This goes through layers and is converted into a 512 by 1 flattened input nodes, then moves to 64 hidden nodes, which then outputs into 26 outputs. Since the input nodes do not have any computational value to them, the hidden nodes and the outputs are the only things we have to consider.

Each hidden neurons perform a combination of its inputs and an application of the function, which can be represented with the equation $\sigma(\sum_{i}^{512} w_{ij}x_i)$, where the w is the weight of the connection from the input neuron to the hidden neuron. There are 64 hidden neurons, so we perform 512*64, or 32,768 matrix multiplications in the hidden layer. The linear combination to the output neuron then can be represented as $\tau(\sum_{i}^{64} w_{ij}x_i)$ Similar to the operation above, the linear combination can be calculated by 64*26, which is the number of output neurons, resulting in 1,664 multiplications. In total, there will be 32,768+1,664 multiplications involved with a Neural Network with 512 inputs, 64 hidden neurons and 26 outputs.

Like so, the time complexity of a neural network is based on the size of the input - however, since the time complexity relies on the number of multiplications involved in linear combinations of the neurons, the complexity depends heavily on the number of layers and the size of the layer. Hence, in general, this process can be written out as the equation below:

$$nm_1 + m_1m_2 + m_2m_3 + \cdots + m_{M-1}m_M + m_Mk = nm_1 + m_Mk + \sum_{i=1}^{M-1} m_im_{i+1}$$

Where n is the number of inputs, M being the number of hidden layers, m being the specific neuron being accessed, and k being the number of output neurons. This equation can then be simplified into:

$$\Theta\left(nm_1 + m_Mk + \sum_{i=1}^{M-1} m_im_{i+1}\right)$$

Results

We used a dataset in a form of csv file that contains 372450 images of handwritten alphabets in 28 x 28 pixels to train and test the model. We splitted the data into train and test sets using train_test_split() function from sklearn library and created a dictionary that matches the label of the data to the actual alphabet. Then, we reshaped the csv data into 28x28 images and built the model using the model implementation explained above. We fitted the model with the train data with epochs of 1, which indicates the number of passes of the entire training dataset, and validated using the test data.

```
history = model.fit(train_x_reshaped, train_y, epochs=1,  validation_data =
(test_x_reshaped,test_y))
```

```
9312/9312 [==============================] - 171s 18ms/step - loss: 0.3011 - accuracy: 0.9135 - val_loss: 0.0724 -
val_accuracy: 0.9806
```

With a single epoch, the accuracy of the training was 0.9135 and the percentage of the training loss was 0.3011. With the validation data, the accuracy was 0.9806 and the loss was 0.0724. After training the mode, we used the existing test data again to visualize the result. The prediction is made using the predict() function.

```
preds = model.predict(test_x_reshaped)
```

Below is the visualization of the testing dataset with real alphabets and the predicted alphabets.

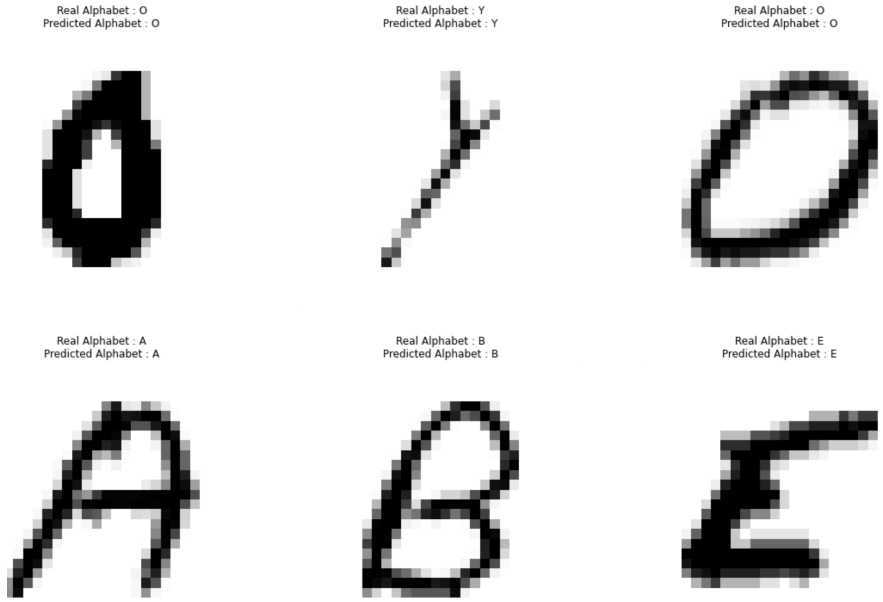

Figure 7. Prediction using the test data

Then we used our own handwriting data to test the algorithm. We created png images of handwritten alphabet from A to Z, each converted into 28 x 28 pixels. The figure below is the result of our own handwriting and we can see that the result using our handwriting also correctly predicts the alphabet.

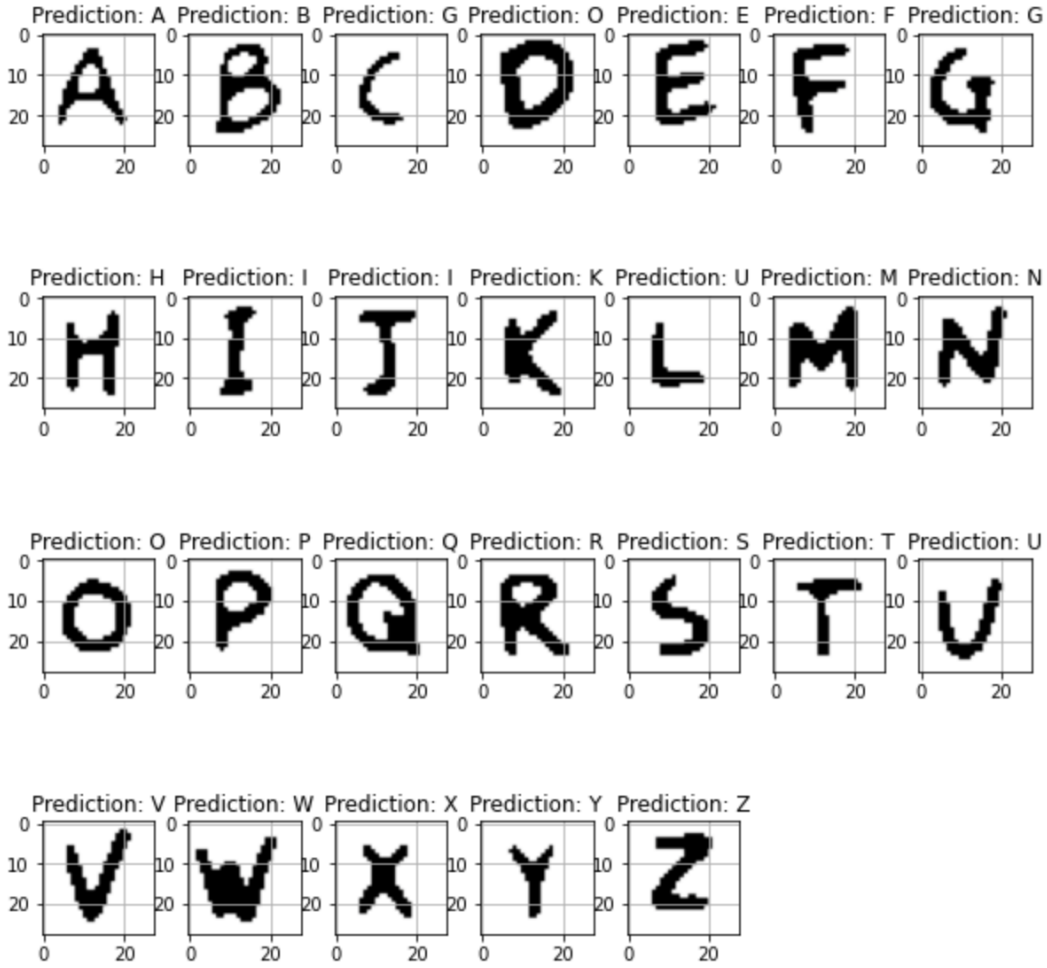

Figure 8. Prediction using our own handwriting.

Experiment / Interpretation

In order to decide the epoch value, we had to know the change in accuracy and loss of the data and its runtime. Passing a higher value of epoch can increase the accuracy of the model, but it takes more time for the code to run. On the other hand, fitting the model with lower epoch value decreases the accuracy but the runtime also decreases. Thus, we conducted an experiment to find out how the runtime and accuracy changes if the epoch value increases.

We fitted the model with an epoch value of 10, and checked the time of each epoch. Then, we plotted the loss and accuracy of the training data and validation data to figure out the relationship between the accuracy/loss and epochs.

```
Epoch 1/10
2328/2328 - 103s - loss: 0.0313 - accuracy: 0.9909 - val_loss: 0.0401 - val_accuracy: 0.9897
Epoch 2/10
2328/2328 - 111s - loss: 0.0250 - accuracy: 0.9927 - val_loss: 0.0368 - val_accuracy: 0.9907
Epoch 3/10
2328/2328 - 108s - loss: 0.0198 - accuracy: 0.9940 - val_loss: 0.0345 - val_accuracy: 0.9917
Epoch 4/10
2328/2328 - 110s - loss: 0.0163 - accuracy: 0.9951 - val_loss: 0.0313 - val_accuracy: 0.9919
Epoch 5/10
2328/2328 - 106s - loss: 0.0139 - accuracy: 0.9957 - val_loss: 0.0345 - val_accuracy: 0.9924
Epoch 6/10
2328/2328 - 107s - loss: 0.0123 - accuracy: 0.9961 - val_loss: 0.0334 - val_accuracy: 0.9924
Epoch 7/10
2328/2328 - 105s - loss: 0.0112 - accuracy: 0.9962 - val_loss: 0.0338 - val_accuracy: 0.9926
Epoch 8/10
2328/2328 - 106s - loss: 0.0097 - accuracy: 0.9968 - val_loss: 0.0322 - val_accuracy: 0.9932
Epoch 9/10
2328/2328 - 108s - loss: 0.0088 - accuracy: 0.9972 - val_loss: 0.0340 - val_accuracy: 0.9932
Epoch 10/10
2328/2328 - 111s - loss: 0.0090 - accuracy: 0.9970 - val_loss: 0.0336 - val_accuracy: 0.9937


Execution Time :17.934 minutes
```

Figure 8. A result of fitting the model with epoch of 10

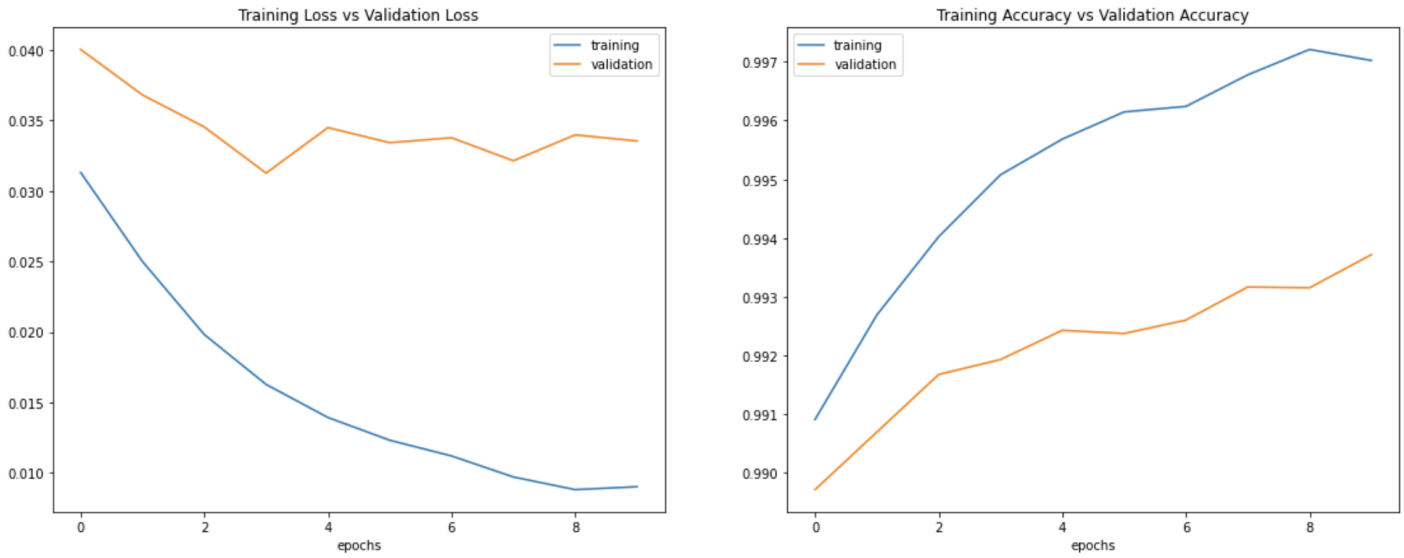

Figure 9. Correlation between epoch and loss/accuracy of data

As shown in figure 8, each pass(epoch) of the entire train data takes approximately the same time of around 110 seconds. In other words, the runtime increases linearly with the increase of the value of epoch. Also, looking at the graph, we figured out that the loss of the data decreases and the accuracy of the data increases when epoch increases. Validation data has less impact on loss and accuracy compared to the training dataset, but still has a decrease in loss and increase in accuracy with the number of epochs.

Through this experiment, we found out that the runtime of the fitting increases linearly when the value of epoch increases. If we decide on a high number of epochs, the model runs for a long time, which is

inefficient. Moreover, the accuracy of the model is already high with a single epoch because the dataset is large enough. Thus, we decided to train the model for only a single epoch. If higher accuracy is needed, we can easily change epoch to higher value based on the accuracy and loss in figure 8.

Conclusion/Extension

In this project, we implemented a machine learning model in python using Convolution Neural Networks (CNN). Based on the research we conducted on CNN, we created our own model that has multi-layers and implemented the model to train the handwriting data. Our handwritten character recognition algorithm can recognize both the existing test data and our own handwritten alphabet images in 28 x 28 pixels with high accuracy. Throughout this project, we learned how CNN works in machine learning and what layers do and how they are created and chosen for different purposes.

The next step of our project is to create our own handwritten fonts. Since the machine is capable of detecting fonts and characters, it is able to detect and categorize our own handwriting. Using this categorization, we can analyze what each character's characteristics are and normalize each character using machine learning. After each 26 characters have been normalized by ML, we can create a vector image that will be the font that we create out of our own handwriting.

Citations

- Pooling
  - https://keras.io/api/layers/pooling_layers/max_pooling2d/
- Flatten Layer
  - https://keras.io/api/layers/reshaping_layers/flatten/
- Layers Activation
  - *https://machinelearningmastery.com/choose-an-activation-function-for-deep-learning/#:~:text=for%20Output%20Layers-,Activation%20Functions,a%20layer%20of%20the%20network*.
  - *https://keras.io/api/layers/activations/*
- https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53
- *https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/*
- Dataset
  - https://www.kaggle.com/sachinpatel21/az-handwritten-alphabets-in-csv-format