Handwriting Recognition

Ashley Jacob and Brendan Barnett

Introduction

This project focuses on using machine learning to recognize and interpret handwritten sentences of capital letters. This proved to be a challenging task with many practical applications - digitizing documents, improving accessibility, and automating data entry.

This approach to handwriting recognition can be split into two main tasks:

1. Identifying individual letters
    a. Classifying an image of a single capital letter into one of 26 classes. Random guessing has a 3.8% accuracy.
    b. The initial aim was for 80% accuracy, a realistic goal given the variability in handwriting styles and large number of classes.
2. Reading entire sentences
    a. Breaking a handwritten sentence into words and letters, classifying each letter, and reconstructing the text in the correct order.

The ultimate goal is to analyze an image of handwriting, restricted to capital letters, and return the correct text as a string. To achieve this, the following were used:

1. Convolutional Neural Networks (CNNs) to classify an image of any given capital letter.
2. DBSCAN clustering to segment images of sentences into words, then words into letters.

This paper details the approach, results, and insights.

The Dataset and Its Influences

The original dataset chosen was very small. It only had 55 images per letter which led to a lot of overfitting. Attempts were made to combat this through data augmentation, but the issue persisted. It was essential to find a better dataset [1].

The final dataset used for this project was unbalanced, with varying numbers of samples for each letter class. Each image was 28x28 pixels and centered, which influenced the approach to processing. To handle images from the "reading" portion of the pipeline, all inputs were converted to the same 28x28 centered format [2].

Given the dataset's size and limitations on Colab, an emphasis was placed on building a robust processing pipeline and a specialized model focused on classifying centered letters. While dataset augmentation - such as shifting or rotating images - could have improved generalization, the code is structured to work directly with the raw dataset due to limited RAM.

The dataset contained a wide range of sample counts per class:

- Highest count: 57,825. Letter is O.
- Lowest count: 1,120. Letter is I.
- Mean count: 14,325.

Down-sampling more common classes was not chosen, as every letter's representation contributes valuable information to the classifier. Instead, the model trained on all available data to maximize the model's exposure to variations. The class distributions are shown below.

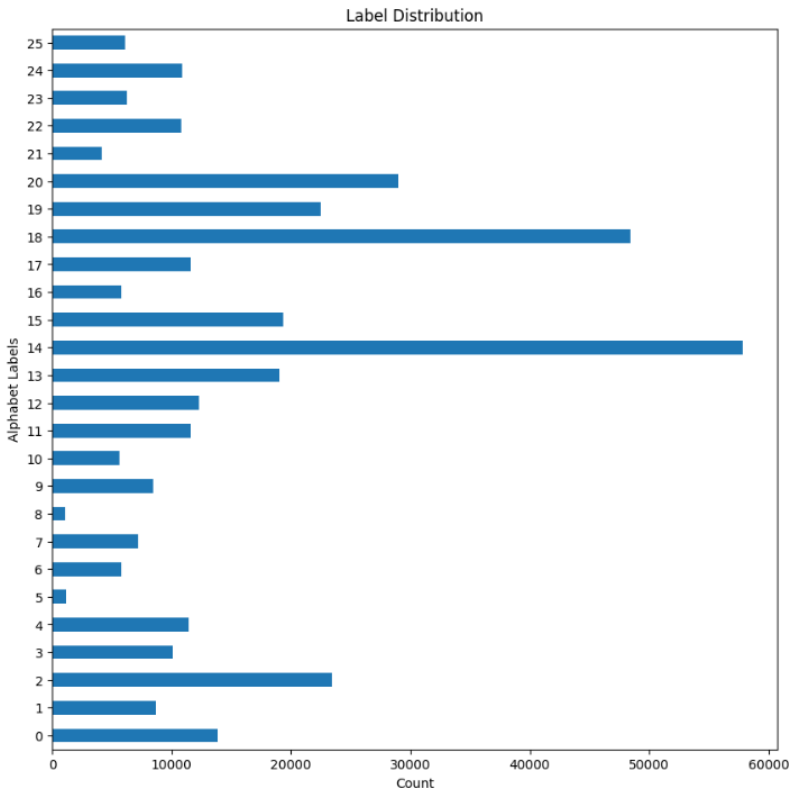


Image 1: Class Distributions

Literature Review for Classifier

This approach to handwriting recognition leveraged Convolutional Neural Networks (CNNs) for multi-class image classification. As noted in An Introduction to Convolutional Neural Networks [3],

> "CNNs are primarily used in the field of pattern recognition within images, allowing for the encoding of image-specific features into the architecture, making the network more suited for image-focused tasks."

Ultimately, a CNN was chosen for the following reasons:

1. Effective for image recognition tasks by leveraging spatial hierarchies.

2. Ability to capture more nuanced and abstract details while offering flexibility to avoid overfitting through layered convolution and pooling layers.
3. Easy to fine-tune hyperparameters for improved learning.

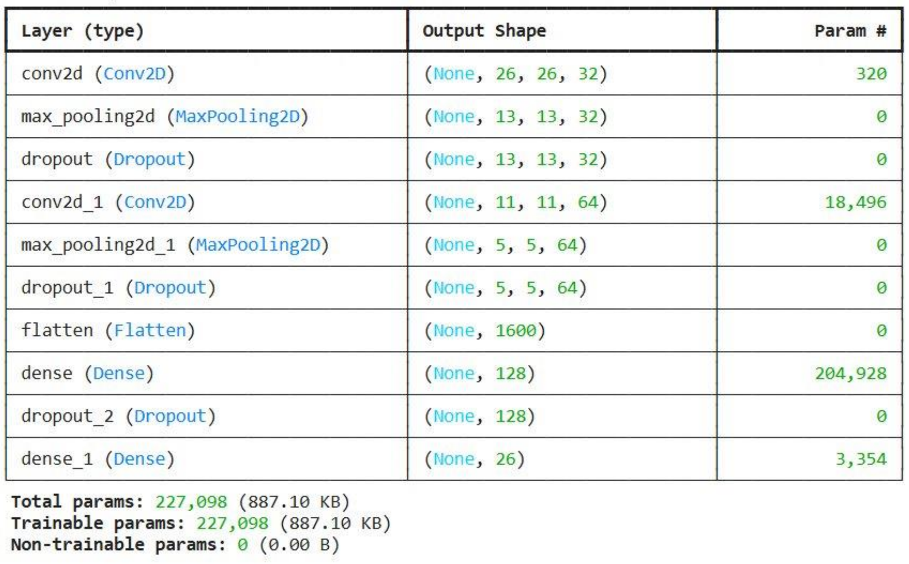
To enhance the performance of the CNN, the architecture was bolstered by incorporating additional features such as pooling layers. According to Pooling Methods in Deep Neural Networks [4],

> “Pooling is a key-step in convolutional based systems that reduces the dimensionality of the feature maps. It combines a set of values into a smaller number of values, i.e., the reduction in the dimensionality of the feature map.”

The final CNN architecture had the following layers:

- Conv2D: Applies convolutional filters to learn spatial hierarchies.
- MaxPooling2D: Reduces the dimensionality of feature maps by retaining the most key information.
- Dropout: Regularization technique to prevent overfitting by randomly dropping neurons during training.
- Flatten: Converts the multi-dimensional feature maps into a 1D array for input into dense layers.
- Dense: Fully connected layers to map learned features to output classes.

The Conv2D and MaxPooling2D layers were repeated to gain increasingly abstract and nuanced understanding of the input data, with some Dropout layers included to prevent the model from just memorizing the data. This refined the model’s ability to classify handwritten letters accurately. The final model architecture is shown below.



Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 26, 26, 32)	320
max_pooling2d (MaxPooling2D)	(None, 13, 13, 32)	0
dropout (Dropout)	(None, 13, 13, 32)	0
conv2d_1 (Conv2D)	(None, 11, 11, 64)	18,496
max_pooling2d_1 (MaxPooling2D)	(None, 5, 5, 64)	0
dropout_1 (Dropout)	(None, 5, 5, 64)	0
flatten (Flatten)	(None, 1600)	0
dense (Dense)	(None, 128)	204,928
dropout_2 (Dropout)	(None, 128)	0
dense_1 (Dense)	(None, 26)	3,354

```
Total params: 227,098 (887.10 KB)
Trainable params: 227,098 (887.10 KB)
Non-trainable params: 0 (0.00 B)
```

Image 2: Classifier Architecture

After solidifying the architecture, hyperparameter tuning was essential to balance overfitting and learning. There was a specific focus on experimenting with number of epochs and batch size to prevent overfitting and learn efficiently.

Grounding the approach in research and what was learned in class allowed for a robust model design that could be freely tweaked and optimized with a solid understanding of CNNs and how they work.

Classifier Experiments and Results

To evaluate the handwriting recognition model, it was trained on 80% of the dataset while withholding 20% for testing. The model was trained for 8 epochs – training more showed a marginal improvement in performance while increasing the risk of overfitting. The convergence is seen in the validation curve below. The training took approximately 50 minutes.

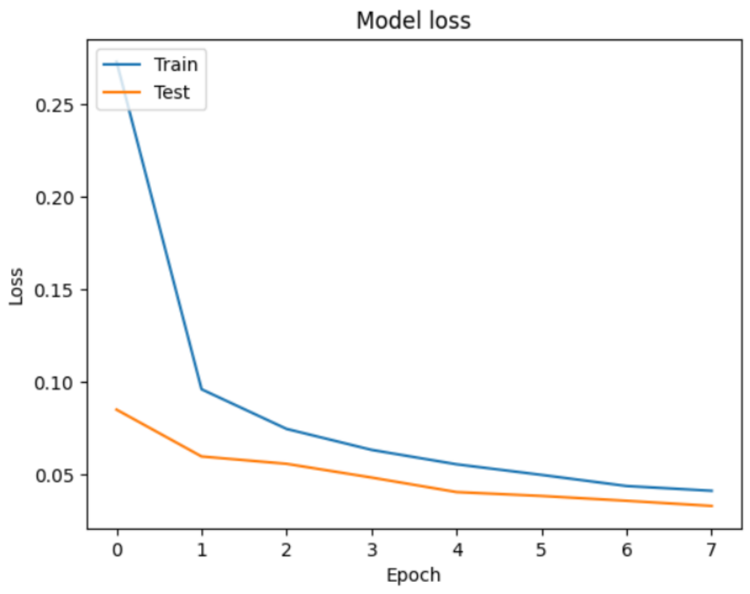


Image 3: Loss While Training

The model demonstrated strong performance on the test set. It achieved the following metrics:

- Accuracy: 0.991193
- Precision: 0.991197
- Recall: 0.991193
- F1-Score: 0.991176

The confusion matrix, shown below, provides insights into the model's errors. The x-axis represents predicted labels, and the y-axis represents true labels. The strong diagonal from the top-left to the bottom-right indicates accurate predictions, with relatively few misclassifications.

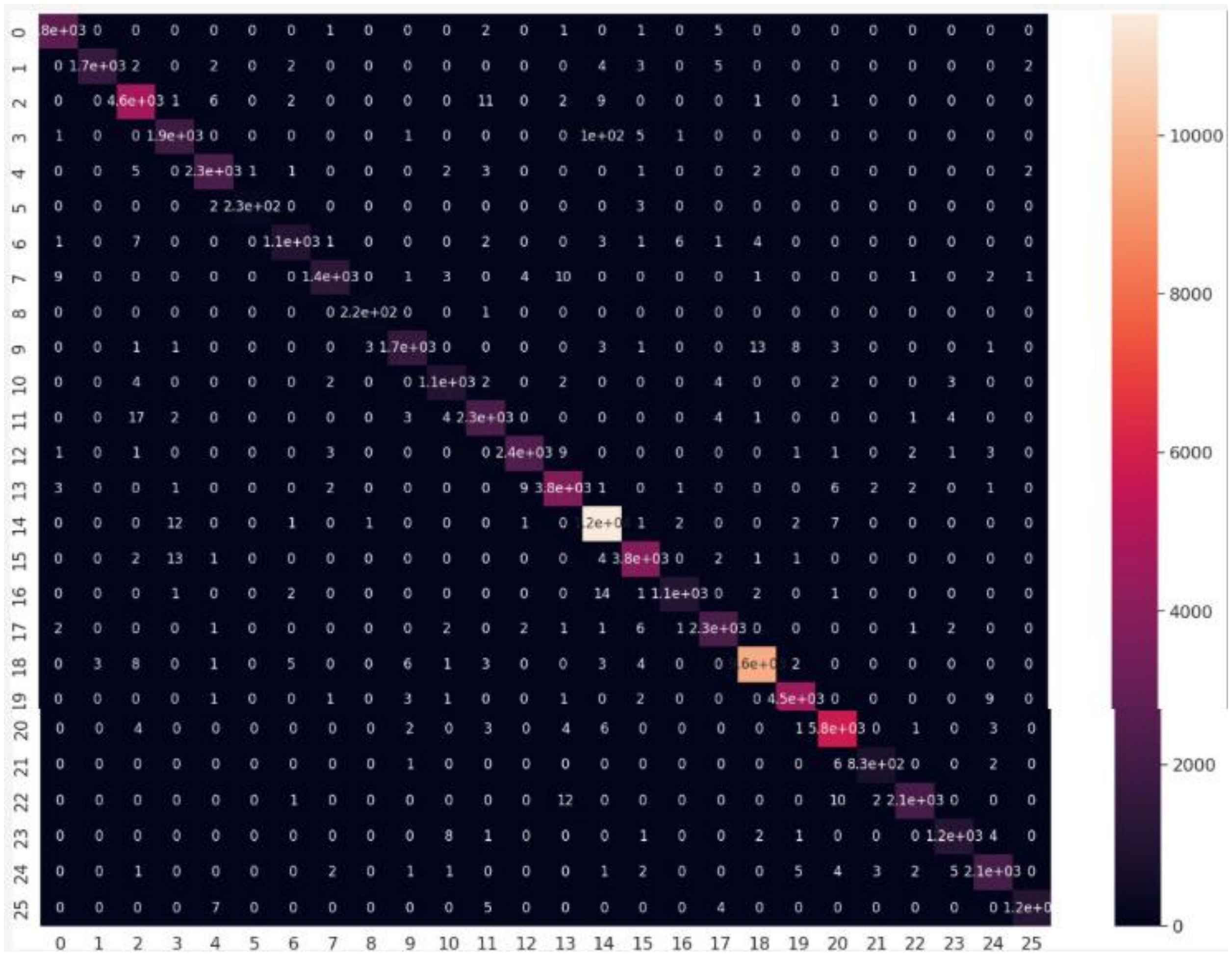
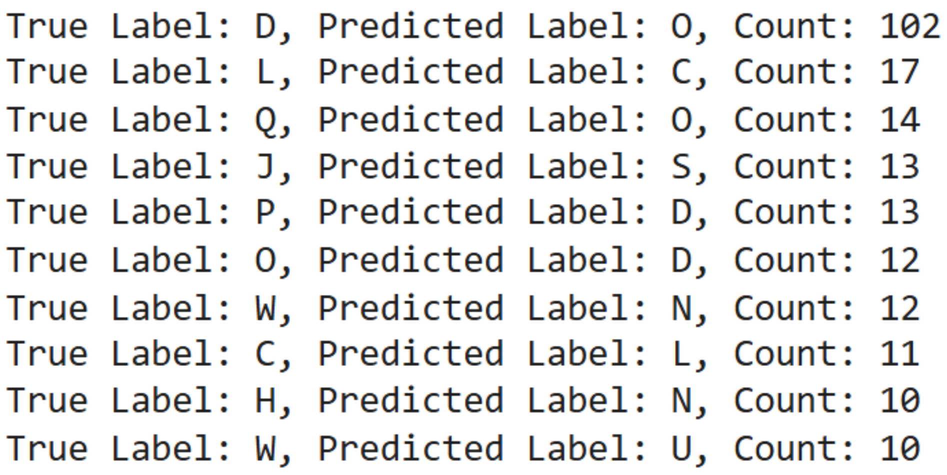


Image 4: Confusion Matrix

The most common misclassifications are shown in the image below. Notable misclassifications included letters with similar shapes, which was expected. For example, D and O as well as Q and O were commonly confused due to their visual similarity.



```
True Label: D, Predicted Label: O, Count: 102
True Label: L, Predicted Label: C, Count: 17
True Label: Q, Predicted Label: O, Count: 14
True Label: J, Predicted Label: S, Count: 13
True Label: P, Predicted Label: D, Count: 13
True Label: O, Predicted Label: D, Count: 12
True Label: W, Predicted Label: N, Count: 12
True Label: C, Predicted Label: L, Count: 11
True Label: H, Predicted Label: N, Count: 10
True Label: W, Predicted Label: U, Count: 10
```

Image 5: Most Common Incorrect Predictions

For instance, the test image below of a “Y” was misclassified as an “X.” This classification, though incorrect, aligns with human perception. When reviewing this image, both experimenters thought it was an “X”.

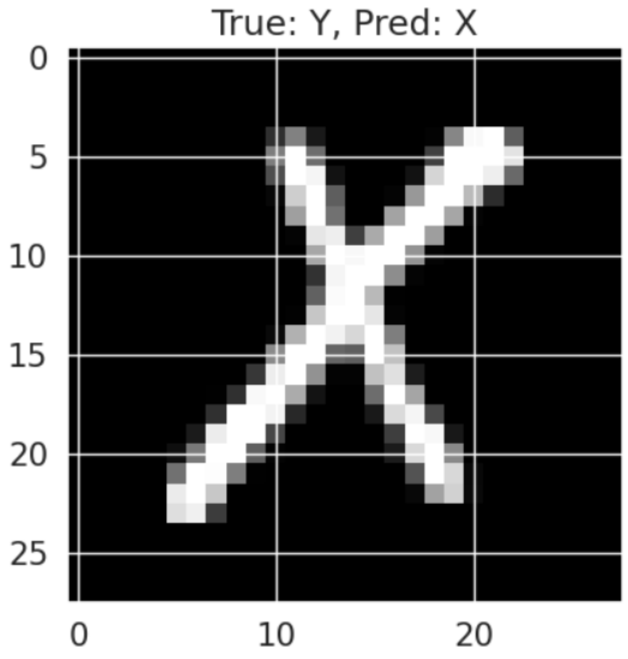
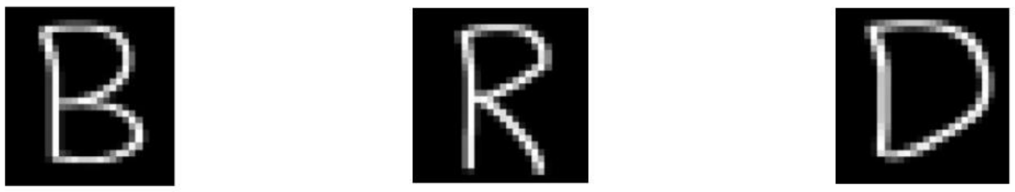


Image 6: Incorrect Classification

These errors highlight the edge cases that are present in the dataset, where even humans might struggle to classify certain samples correctly. To ensure the model works well on different handwriting styles, some small mistakes are willingly conceded, like the one above, to improve its ability to handle new handwriting. This also helps prevent overfitting.

To further validate generalizability, the model was tested using custom samples of handwritten letters, including commonly misclassified letters. The custom handwriting used below, which is more rigid and structured compared to the dataset, served as a useful first test. Some of the test images and respective predicted classes are shown below.



```
Rank 1: B with probability 1.0000
Rank 2: R with probability 0.0000
Rank 3: G with probability 0.0000
```

```
Rank 1: R with probability 0.9841
Rank 2: K with probability 0.0159
Rank 3: B with probability 0.0000
```

```
Rank 1: D with probability 0.9999
Rank 2: U with probability 0.0001
Rank 3: N with probability 0.0000
```

Image 7: Test Letters and Predictions

As shown above, the model performed well, with high confidence scores for correct predictions across all “sanity-check” tests. Since Softmax was used as output, it was possible to observe the highest probabilities assigned to each class. This demonstrated the model’s ability to identify the correct letter with high certainty.

The classifier excelled at recognizing centered, capital letters, achieving high accuracy and demonstrating robustness to variability in handwriting. Processing these letters well before applying the model is essential. While some misclassifications occurred, especially for visually similar letters, the model strikes a good balance between dataset specificity and generalizability to unseen handwriting.

Clustering Experiments

The end goal of the project was to be able to have the model read an entire sentence from an image. However, the model could only make accurate predictions for images that resembled the samples in its training set. For example, one difference that needed to be rectified was that the training images white text on a black background, while the images that would be run through the pipeline would be black text on a while background — the inverse. That meant that images needed to have some sort of preprocessing before being fed into the model.

The main issue that needed to be solved was isolating the letters in the image. Since the model could only predict one letter at a time, the image needed to be divided up by character. The simplest way to do this seemed to be through clustering. The initial approach taken was to use the k-means algorithm to cluster the pixels into images.

Assuming that the text in the image is written onto a white background, all the dark pixels are converted into points. The algorithm then uses their relative proximity to group them together. Ideally, the pixels that make up a single letter will be close enough to each other that they should all be put into the same cluster. However, this turned out to not be the case.

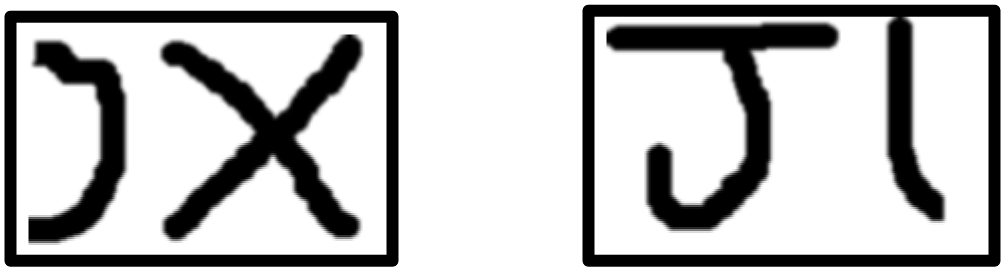


Image 8: Examples of k-means clustering

As can be seen from the examples shown above, while it was able to capture some letters in their entirety, it would often also cluster segments from adjacent letters. As a result, the overall clusters would not accurately represent all of the letters. Using k-means also came with other issues. K-means Clustering Algorithms: A Comprehensive Review, Variants Analysis, and Advances in the Era of Big Data [3] outlines many of these shortcomings:

> “In the standard K-means algorithm, the cluster number is required as a user parameter and is used in the arbitrary cluster center selection from the dataset. However, the K-means algorithm may converge to a local minimum because of its greedy nature. Therefore, it requires several runs for a given value with different initial cluster center selections to obtain the optimal cluster result. In addition, the standard algorithm detects ball-shaped or spherical clusters only because of the use of the Euclidean metric as its distance measure.”

Since letters are made up of irregular shapes, it is difficult for the algorithm to cluster just the letters themself. It is always possible that sections from one letter will actually be closer to the center of another. Additionally, k-means is pretty inconsistent. It depends so heavily on the initial centroid values that the end clusters vary between each iteration of the algorithm. On top of this, the number of clusters, or letters in this case, must be specified prior to running k-means. This would make the pipeline a lot less flexible since it limits the length of sentences that can be read. With all these downsides, it was clear that a new clustering method was necessary.

Thankfully, DBSCAN seemed to fit the bill. According to A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise [6],

> “...DBSCAN is significantly more effective in discovering clusters of arbitrary shape than the well-known algorithm...”

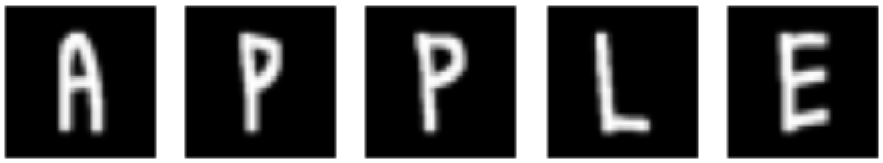


Image 9: DBSCAN cluster

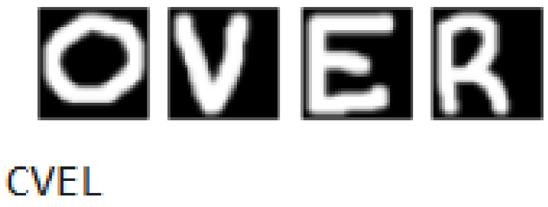
DBSCAN solves pretty much all of the problems that k-means had. Since it clusters based on density, the algorithm will return the same clusters every time. It also avoids the issues of multiple letters showing up in the same cluster so long as they are not overlapping. This means that an area of low-density separates them, indicating that they belong to different shapes.

Having finally solved the letter isolation issue, there remained one other minor obstacle with the clusters. The clusters themselves were not guaranteed to be in the correct order and there was no way to tell where the separation between words laid. Fortunately, the fix for this was fairly simple.

Two layers of DBSCAN can be run to first separate the sentence into words and then into letters. This implementation runs off the assumption that the gaps between words are generally larger than the gaps between letters within the same word. The centers can then be sorted along the x-axis to determine the correct order.

Image Preprocessing

Before the model can make a prediction, the clusters must be converted into an image format that it can read. The initial approach was to use the range of x and y values of each cluster to determine the dimensions of the image and paste the letter in with some padding. This way, each image would clearly display just one letter. Then, it could be resized to match the expected input of the model. However, the model seemed to perform pretty poorly on these images. While it was able to get over 99% accuracy on the test set, it was failing to identify real handwriting. A lot of these images were also quite clear and legible to the human eye, so the misclassification was confusing.



```
CVEL
```

Image 10: Prediction without proper padding

After taking a closer look at the images in the training dataset, the issues became clear.

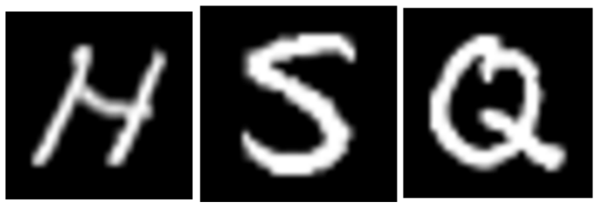
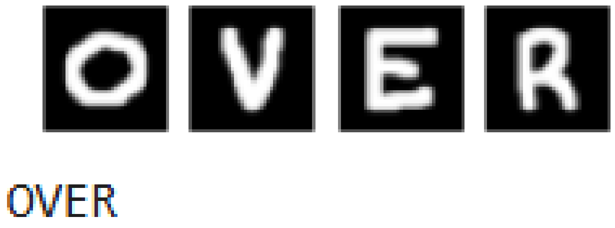


Image 11: Samples from training dataset

While the difference may seem subtle, the dataset images are much more centered with greater padding around each letter. Since the custom handwriting samples did not match the data the model was trained on, it could not make correct predictions. After the preprocessing was updated to account for these changes, the model's performance improved dramatically.



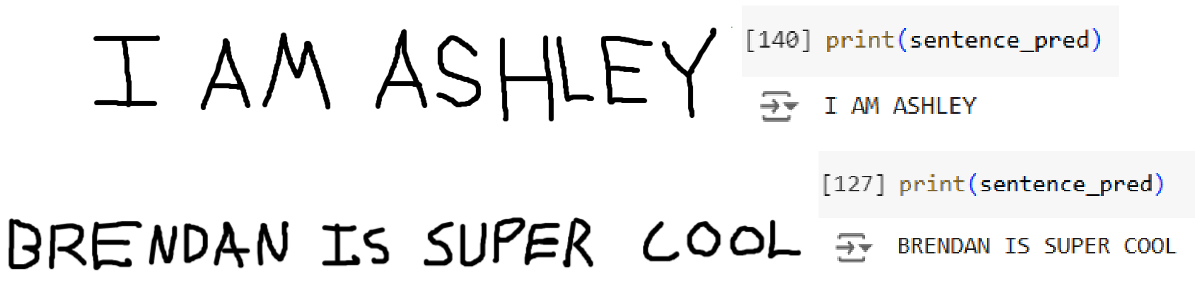
```
OVER
```

Image 12: Prediction after proper preprocessing

As the picture above shows, the model was able to properly read the image once the proper preprocessing was applied. Compared to the images after the original preprocessing, these images are a lot closer to the dataset.

Final Pipeline Analysis

Once the pipeline was completed, custom handwriting samples were run through it to test if it could accurately read them. It was found that the pipeline performed well, especially if the handwriting was neat.



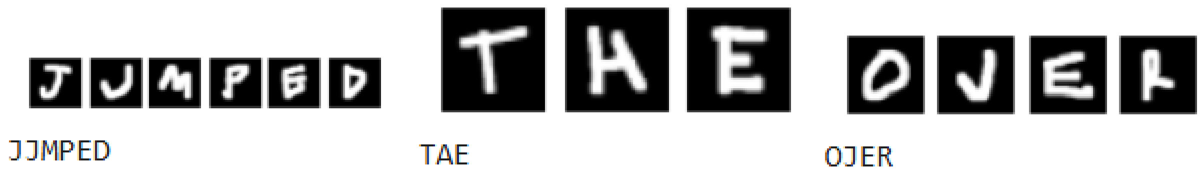
```
[140] print(sentence_pred)
I AM ASHLEY
```

```
[127] print(sentence_pred)
BRENDAN IS SUPER COOL
```

Image 13: Examples of correct pipeline output

Both of the sentences above were clearly read by the pipeline. However, that does not mean that the pipeline is without its flaws. There were quite a few concessions that had to be made regarding the types of handwriting that could be read. Since DBSCAN was used for letter isolation, the individual letters could not touch each other, even though that happens quite often in actual handwriting. The clustering algorithm groups the pixels into letters based on what is touching each other. If letters overlap, they will get placed into the same cluster, interfering with the final prediction. The pipeline also makes some assumptions with how words and letters are spaced out. If words or letters are closer or further apart than anticipated, the final predicted sentence may not make sense since the clusters were messed up. On top of this, only single line sentences can be read for now. The cluster centers from DBSCAN are sorted by x-value to determine the order of the letters. If the words were all located on multiple lines, their order in the final prediction would not be accurate. The model can only read from left to right — not top to bottom.

In addition to these fundamental flaws of the system chosen for this pipeline, the model itself has its own errors. While better preprocessing went a long way in improving its ability to read the custom images, it still is not perfect. Its performance tends to suffer quite a bit when presented with messy handwriting.



```
JJMPED
```

```
TAE
```

```
OJER
```

Image 14: Examples of prediction errors

While the above words were read incorrectly by the model, it is not too hard to understand why. These errors are understandable when the images were viewed in isolation. Humans can mitigate this because they have context — they can see the whole picture. Since the model is not predicting the letters within their wider context, it is not able to correctly discriminate between similar-looking letters. When looking at the words that these letters were placed in, it seems obvious that a mistake has been made.

Additional Experiments

For the custom tests, the images were created in Microsoft Paint by using a black brush on a white background. This gave the highest contrast between the test and background which made it easier to isolate the letters. However, a couple of additional experiments to see if the model could read other types of images.

Images with handwriting on paper have a lot more variables that can make it hard to separate the words from the rest of the picture. The pipeline uses a filter based on how dark a pixel's gray level is to determine if it is part of a letter or not. However, shadows and notebook lines can add additional noise to the image. Especially if gray pencil lead is used to write the sentence, an area under shadow may be just as dark as text.

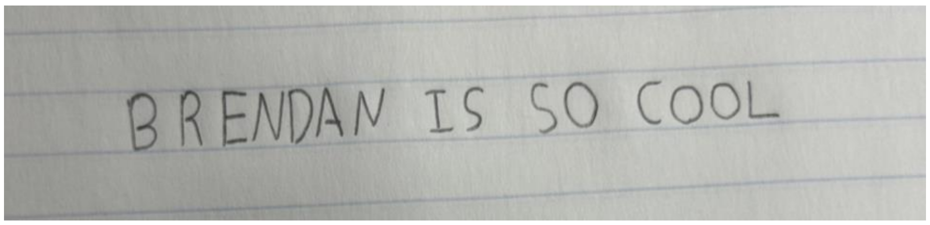
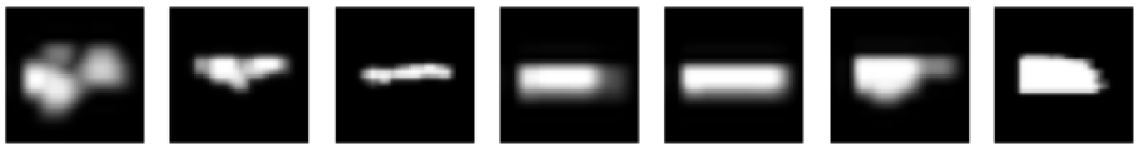


Image 15: Handwriting on paper

When the picture was run through the pipeline, it had a hard time isolating and clustering the letters. In quite a few "word" clusters, there were letters that just looked like horizontal lines. These "letters" were likely notebook lines that managed to be dark enough to also get picked up as potential parts of the sentence.



```
OTTJJEWPCAAPECJ
```

Image 16: Incorrect clusters including notebook lines

On the flip side, parts of the actual letters were not dark enough to all get picked up by the filter at once. The image below shows how some letters got split up. Pencils write in gray, so variations in pressure and lighting may mean that it does not show up all that dark in images.

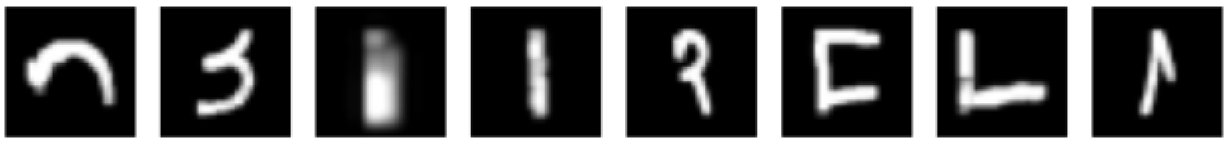
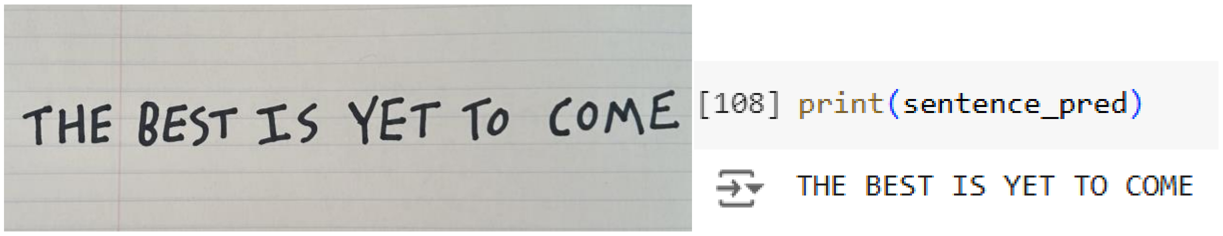


Image 17: Incorrect clustering of "BRENDAN"

On the bright side, that does not mean that the pipeline never works for images taken in real life. If certain aspects are controlled, like lighting, the model will perform just as well as it does for images made in Paint. Since pencil sometimes shows up as too light, using dark ink, like a Sharpie, improves the accuracy of the pipeline.

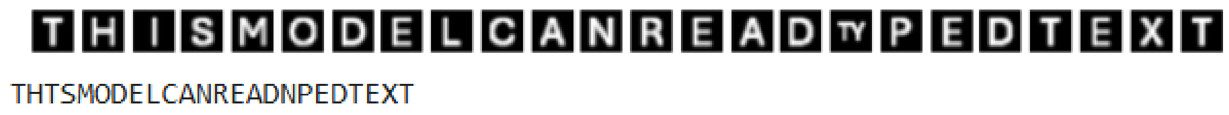


```
[108] print(sentence_pred)

THE BEST IS YET TO COME
```

Image 18: Handwritten text using Sharpie

One final test was to see if the model could read typed text. Since it was trained on handwritten images, this seemed unlikely, but it was worth a shot.



```
THTSMODELCANREADNPEDTEXT
```

Image 19: Pipeline output for typed text

While the pipeline struggled to separate the sentence into words, it almost perfectly divided it into letters. Only two letters were close enough to accidentally get grouped together. This likely happened since the space between typed characters is a lot smaller than gaps between handwritten letters. Adding a few more spaces between words or adjusting the DBSCAN criteria would likely fix this issue completely. For the letters that were isolated properly, the prediction accuracy was quite high. Only one letter was misclassified. The differences between typed and written letters likely disappeared once the image was shrunk down to 28x28 pixels. Only the basic shapes of the letters remained which allowed them to be properly classified.

Conclusion

Despite some errors, the model exceeded all expectations. The initial expectation was for the model to only be accurate about 80% of the time due to the number of classification classes and potential variability in handwriting. However, it was able to predict the right letter in over 99% of the samples in the test dataset. In nearly all of the custom tests, the model was able to correctly predict the right sentence after the image was processed correctly.

Division of Work

- Model Creation: Brendan, Ashley
- Dataset Analysis: Brendan

- Image Preprocessing: Ashley, Brendan
- Letter Isolation: Ashley
- Paper
    - Written by Brendan:
        - Introduction
        - Literature Review for Classifiers
        - Classifier Experiments and Results
    - Written by Ashley:
        - Clustering Experiments
        - Images Preprocessing
        - Final Pipeline Analysis
        - Conclusion

Links

- Original Dataset
    - https://www.kaggle.com/datasets/dhruvildave/english-handwritten-characters-dataset
- Final Dataset
    - https://www.kaggle.com/datasets/sachinpatel21/az-handwritten-alphabets-in-csv-format
- GitHub Repository
    - https://github.com/Ashley-Jacob/CSE5819_HandwritingRecognition/tree/main

Works Cited

[1] “English handwritten characters,” *Kaggle*, Feb. 24, 2021. https://www.kaggle.com/datasets/dhruvildave/english-handwritten-characters-dataset.

[2] “A-Z Handwritten Alphabets in .csv format,” *Kaggle*, Feb. 16, 2018. https://www.kaggle.com/datasets/sachinpatel21/az-handwritten-alphabets-in-csv-format.

[3] K. O’Shea and R. Nash, “An introduction to convolutional neural networks,” *arXiv (Cornell University)*, Jan. 2015, doi: 10.48550/arxiv.1511.08458.

[4] H. Gholamalinezhad and H. Khosravi, “Pooling methods in deep neural networks, a review,” *arXiv (Cornell University)*, Jan. 2020, doi: 10.48550/arxiv.2009.07485.

[5] A. M. Ikotun, A. E. Ezugwu, L. Abualigah, B. Abuhaija, and J. Heming, “K-means clustering algorithms: A comprehensive review, variants analysis, and advances in the era of big data,” *Information Sciences*, vol. 622, pp. 178–210, Dec. 2022, doi: 10.1016/j.ins.2022.11.139.

[6] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, “A density-based algorithm for discovering clusters in large spatial Databases with Noise,” *Knowledge Discovery and Data Mining*, pp. 226–231, Jan. 1996, [Online]. Available: https://www.aaai.org/Papers/KDD/1996/KDD96-037.pdf.