AI can recognize

Handwriting

Aayesha Islam

Introduction

It is crucial for us to have technology to recognize handwriting and be able to digitize it. This could lead to preserving old handwritten books, letters, parchments, and other important documents. Although this can be done by humans reading and typing out a digitized version of the particular document, using AI to accomplish this will make the process more efficient. In this project, an existing labeled dataset was used to train an AI Neural Network model so that it could recognize words written by hand.

Code

This code was written in the language Python on the Google Colab platform, which allows code to be shared online.

It starts off by importing a dataset. This code is performing several tasks related to the EMNIST (Extended MNIST) dataset, which is an extension of the popular MNIST dataset. The MNIST dataset consists of images of handwritten alphabets, commonly used for training and testing machine learning models. It clones a GitHub repository, downloads the MNIST dataset, installs the emnist library, and imports the required function for extracting training samples from the EMNIST dataset.

```
Downloaded: 6 files, 11M in 0.1s (103 MB/s)
/content/data /content
/content
Collecting emnist
  Downloading emnist-0.0-py3-none-any.whl (7.3 kB)
Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-packages (from emnist) (1.23.5)
Requirement already satisfied: requests in /usr/local/lib/python3.10/dist-packages (from emnist) (2.31.0)
Requirement already satisfied: tqdm in /usr/local/lib/python3.10/dist-packages (from emnist) (4.66.1)
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.10/dist-packages (from requests->emnist) (3.3.2)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-packages (from requests->emnist) (3.4)
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.10/dist-packages (from requests->emnist) (2.0.7)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.10/dist-packages (from requests->emnist) (2023.7.22)
Installing collected packages: emnist
Successfully installed emnist-0.0
Imported the EMNIST libraries we need!
```

In the above image we can see the results after running the code section.

Next it fetches the EMNIST "letters" dataset from the OpenML website, containing 145,600 28x28 pixel images of letters. It normalizes the pixel values to a range of 0 to 1, splits the dataset into training and testing sets (60,000 for training and 10,000 for testing), and reshapes the data to align with the neural network architecture. The code concludes by confirming the successful extraction and division of the dataset.

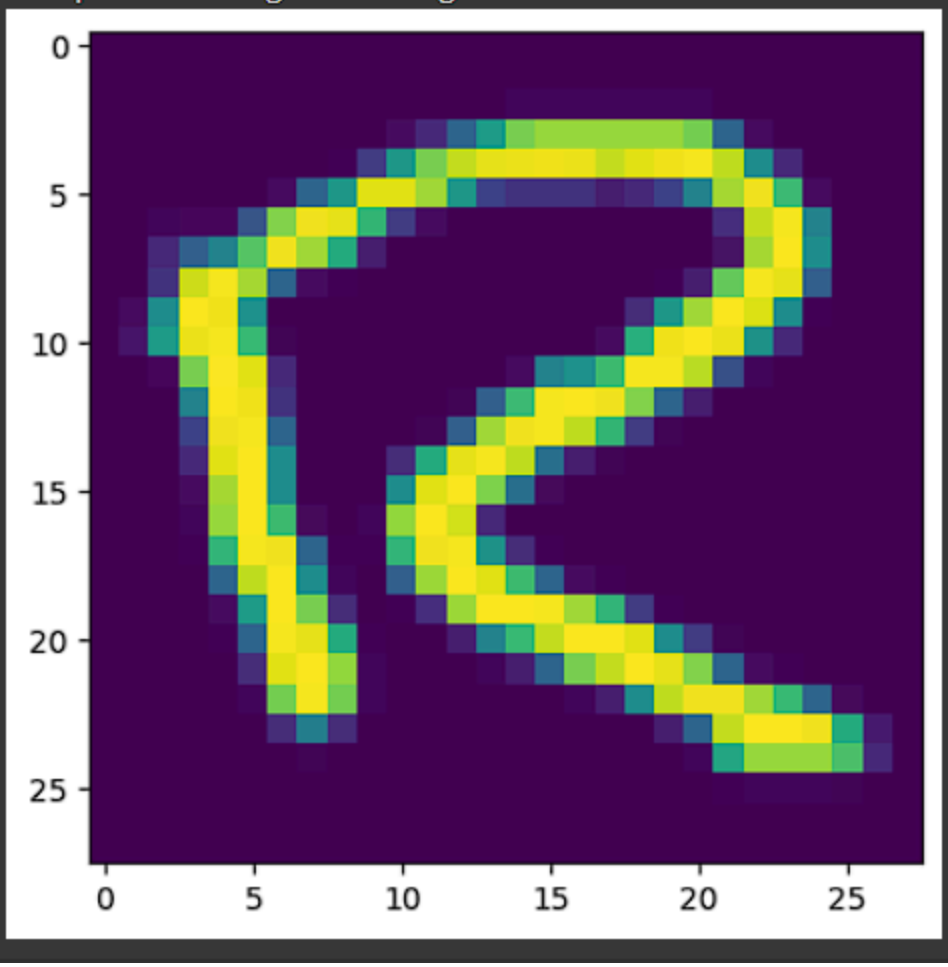
To confirm whether the dataset was properly accessed, the code accesses a particular index and displays the results. From this, we can see what the dataset looks like and we can infer that our test data has to look similar.

```
# STEP 1.3

import matplotlib.pyplot as plt

img_index = 12000 # <<<<<  You can update this value to look at o
img = X_train[img_index]
print("Image Label: " + str(chr(y_train[img_index]+96)))
plt.imshow(img.reshape((28,28)))
```

```
Image Label: r
<matplotlib.image.AxesImage at 0x7b3cadb3c250>
```

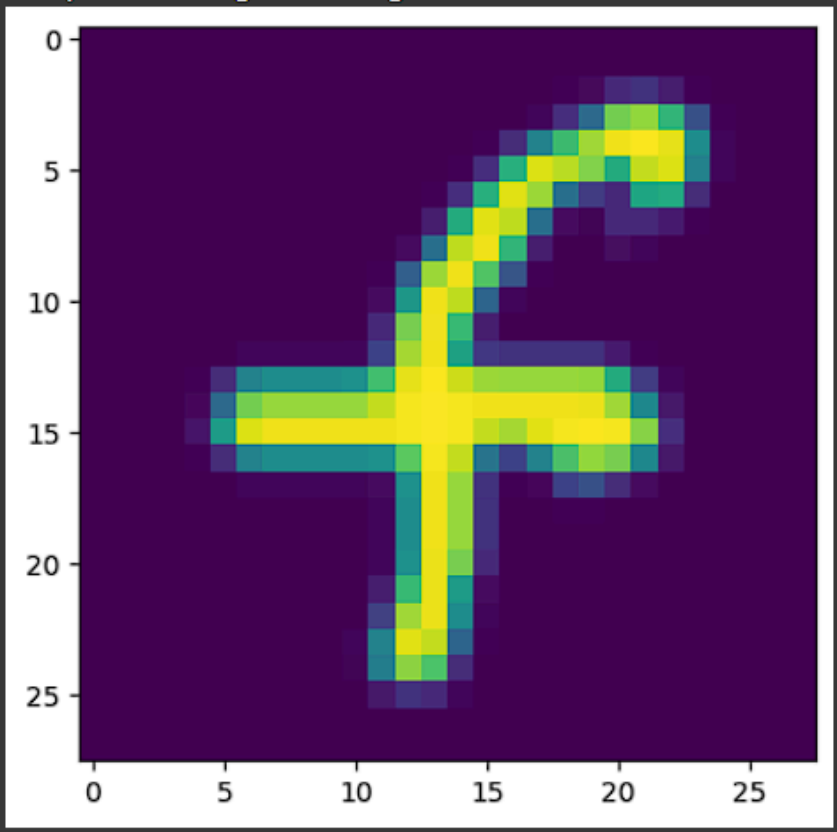


```python
# STEP 1.3

import matplotlib.pyplot as plt

img_index = 397 # <<<<<  You can update this value to look at other images
img = X_train[img_index]
print("Image Label: " + str(chr(y_train[img_index]+96)))
plt.imshow(img.reshape((28,28)))
```

```
Image Label: f
<matplotlib.image.AxesImage at 0x7b3c4039c280>
```

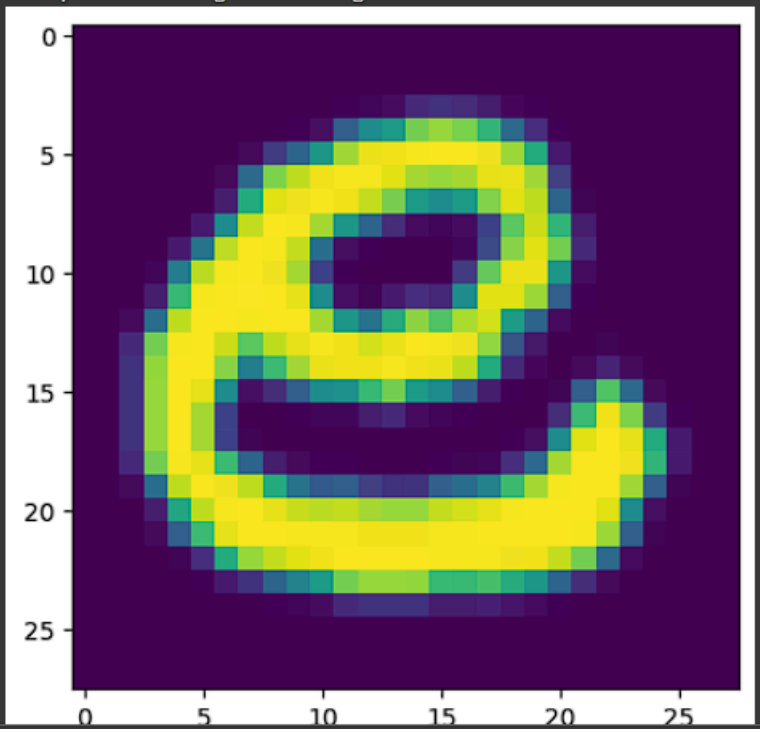


```
# STEP 1.3

import matplotlib.pyplot as plt

img_index = 50000 # <<<<<  You can update this value to look at other images
img = X_train[img_index]
print("Image Label: " + str(chr(y_train[img_index]+96)))
plt.imshow(img.reshape((28,28)))
```

```
Image Label: e
<matplotlib.image.AxesImage at 0x7b3c3fa05090>
```



If you put different indexes, different letters that are stored and labeled can be discovered. These images are able to be clearly interpreted by the AI even though to us they are a bit hazy.

Now that we've examined the data, next is to construct a neural network to take an image as input and predict the corresponding letter. A multi-layer

perceptron (MLP) is perfect for this task. Instead of coding everything from scratch, an existing library that offers MLP tools is used.

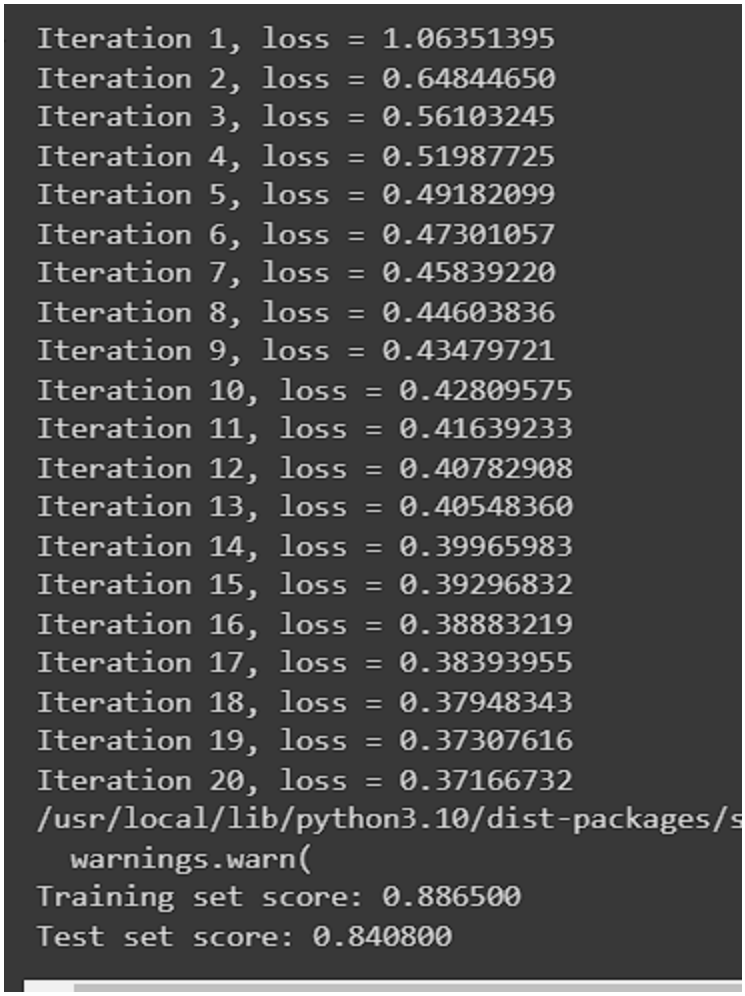
```
# These two lines import the ML libraries we need
from sklearn.datasets import fetch_openml
from sklearn.neural_network import MLPClassifier
```

Initially, an MLP classifier with a single hidden layer containing 50 neurons is made, and trained on the data for 20 iterations. Various learning parameters, such as the learning rate, are set to configure the MLP. Adjusting these parameters can influence the performance of the MLP during training and testing.

Next is testing the algorithm using the training data and seeing how well it performs. There are a few values we have to look out for.
The following code is used for testing.

```
mlp1.fit(X_train, y_train)
print("Training set score: %f" % mlp1.score(X_train, y_train))
print("Test set score: %f" % mlp1.score(X_test, y_test))
```

The output of this code is as below.



Some terms that are seen are:

Iteration: The number of times the training data is run through the algorithm.

Loss: A mathematical function that computes the difference between the predicted output and the actual target. The model adjusts its internal parameters during training to minimize this loss, making the predicted output closer to the true values. The loss value ideally decreases as the number of iterations increases.

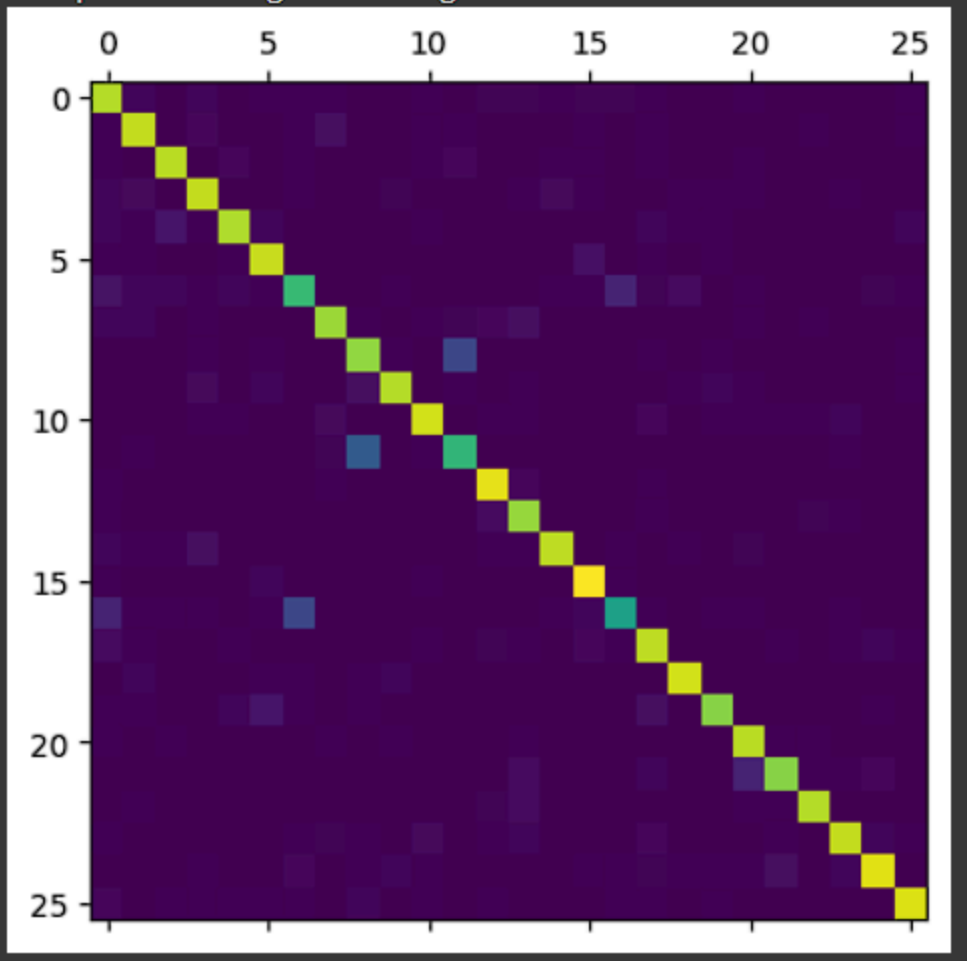
If this code is run more times, the results are similar but slightly different each time, and this is because the neutrons are weighted differently each time.

```
Iteration 1, loss = 1.06351395
Iteration 2, loss = 0.64844650
Iteration 3, loss = 0.56103245
Iteration 4, loss = 0.51987725
Iteration 5, loss = 0.49182099
Iteration 6, loss = 0.47301057
Iteration 7, loss = 0.45839220
Iteration 8, loss = 0.44603836
Iteration 9, loss = 0.43479721
Iteration 10, loss = 0.42809575
Iteration 11, loss = 0.41639233
Iteration 12, loss = 0.40782908
Iteration 13, loss = 0.40548360
Iteration 14, loss = 0.39965983
Iteration 15, loss = 0.39296832
Iteration 16, loss = 0.38883219
Iteration 17, loss = 0.38393955
Iteration 18, loss = 0.37948343
Iteration 19, loss = 0.37307616
Iteration 20, loss = 0.37166732
/usr/local/lib/python3.10/dist-packages/sklearn/
  warnings.warn(
Training set score: 0.886500
Test set score: 0.840800
```

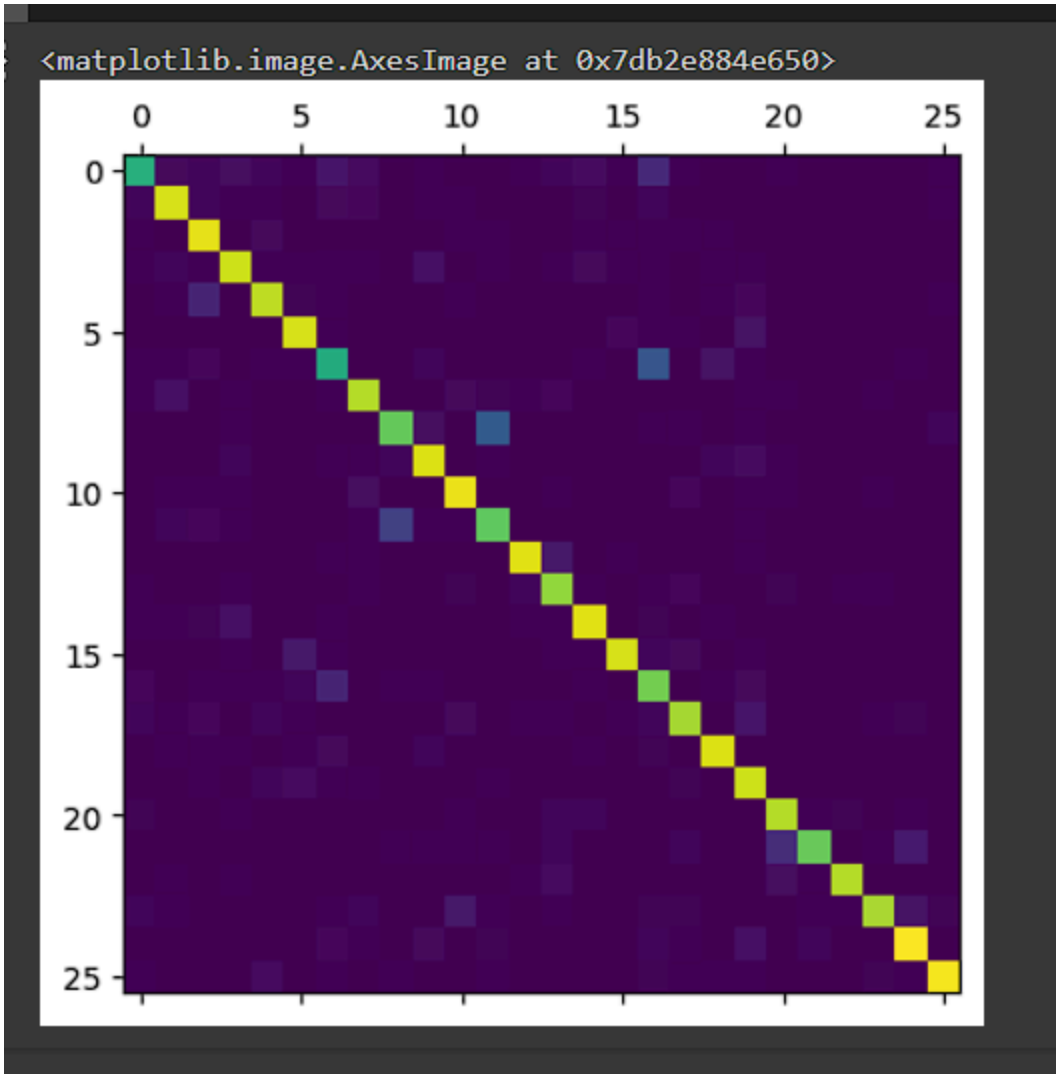
A confusion matrix is the next step as it helps in figuring out where the accuracy is lowering.

```
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_test, y_pred)
plt.matshow(cm)
```

```
<matplotlib.image.AxesImage at 0x7db2f431eec0>
```



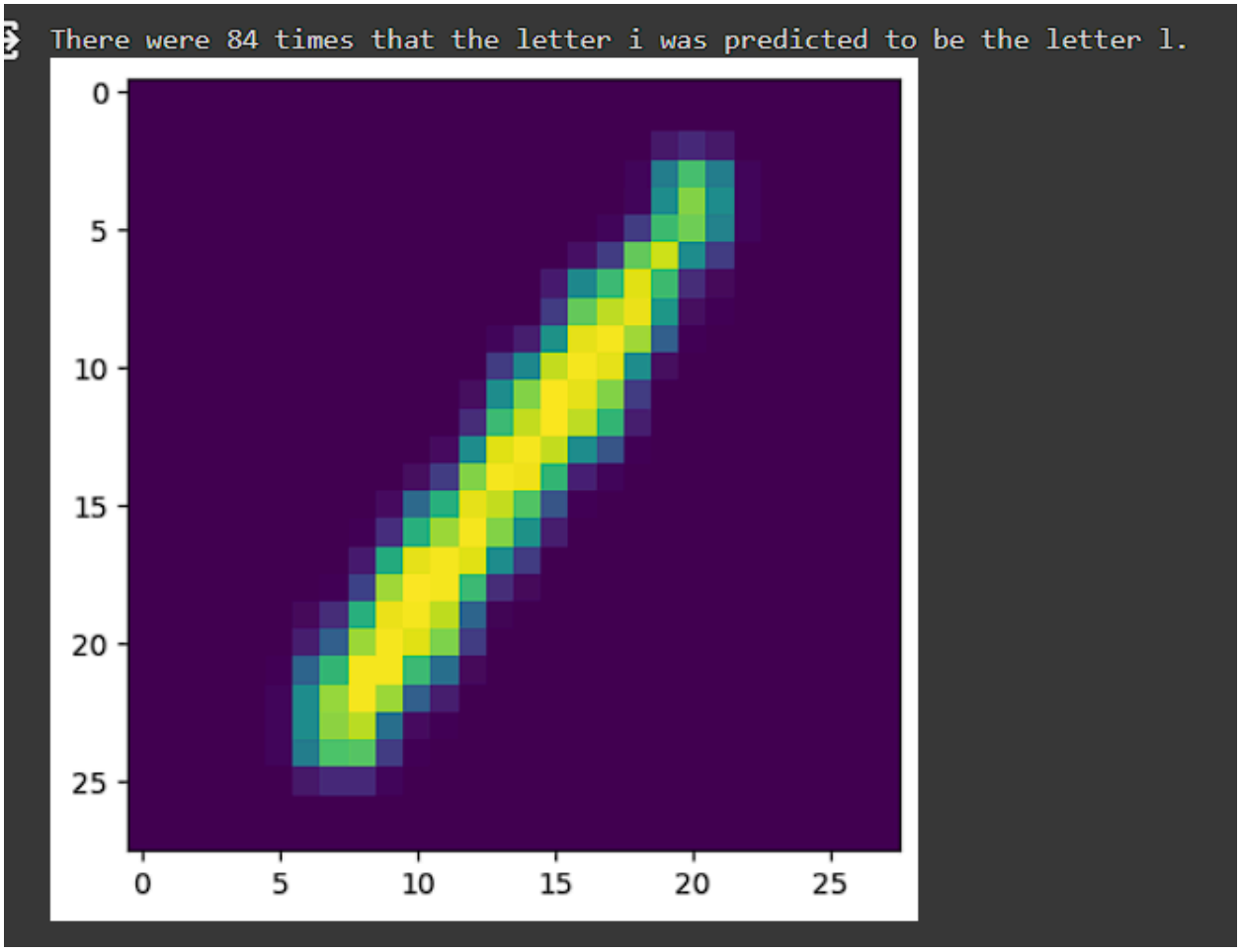
The below is an example of a change in the brightness of the confusion if the hidden layers values are changed from 50 to 30.



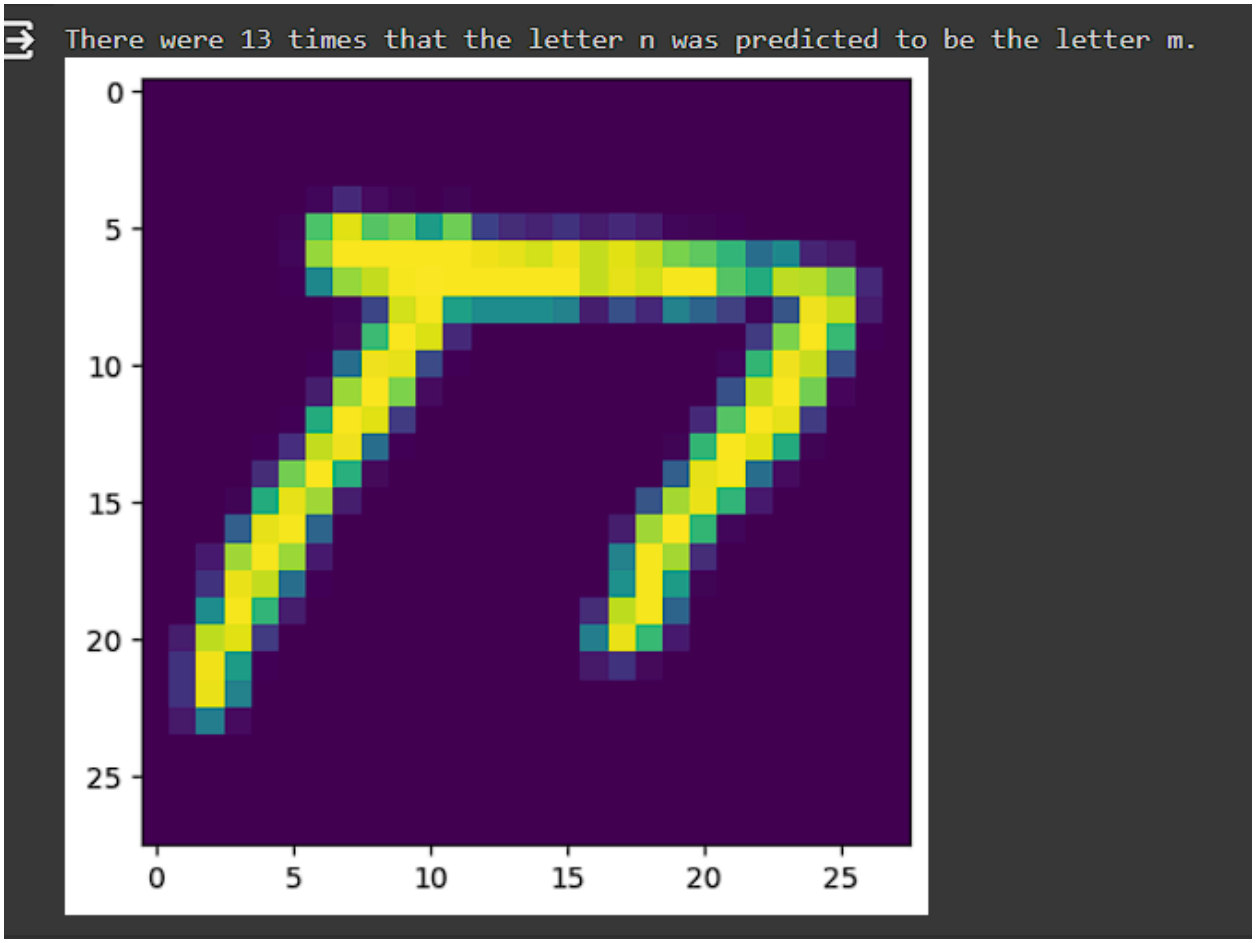
Inaccurate results can happen when the algorithm mixes up different letters which are similar, such as “I” and “l”.

The brightness of each cell in the confusion matrix indicates the quantity of elements within that cell, with a higher brightness denoting more elements. The matrix is structured with rows representing correct values and columns representing predicted values. The numerical labels on the axes correspond to the 26 letters of the alphabet (in the case of EMNIST, letters

are represented by numbers, where 0 corresponds to "A," 1 to "B," and so on). The intensity of cell (0,0) signifies the frequency of correct predictions for the letter "A." The diagonal cells portray instances where the predicted value aligns with the actual value, forming a conspicuous bright line. Any cells outside this diagonal line that are brighter need further investigation.



This is another example of confusing letters. This means that it is confusing these letters less and the number of errors are less.



The accuracy of this can be increased. One approach involves increasing the complexity of the network by adding more hidden layers and neurons within those layers. For instance, to incorporate an additional hidden layer containing 50 neurons, the adjustment can be made as follows: hidden_layer_sizes=(50,50,).

Another strategy is extending the training duration by increasing the number of epochs (or iterations). For example, modifying the parameter max_iter to max_iter=30 would achieve this.

```
# Change some of the values in the below statement and re-run to see how they
# affect performance!
mlp2 = MLPClassifier(hidden_layer_sizes=(100,100,100,100,100,), max_iter=50, alpha=1e-4,
                     solver='sgd', verbose=10, tol=1e-4, random_state=1,
                     learning_rate_init=.1)
mlp2.fit(X_train, y_train)
print("Training set score: %f" % mlp2.score(X_train, y_train))
print("Test set score: %f" % mlp2.score(X_test, y_test))
```

```
Iteration 41, loss = 0.18179144
Iteration 42, loss = 0.17865350
Iteration 43, loss = 0.17174149
Iteration 44, loss = 0.17433637
Iteration 45, loss = 0.17649701
Iteration 46, loss = 0.17965740
Iteration 47, loss = 0.16644906
Iteration 48, loss = 0.16974640
Iteration 49, loss = 0.16569554
Iteration 50, loss = 0.16854811
/usr/local/lib/python3.10/dist-packages/
  warnings.warn(
Training set score: 0.952167
Test set score: 0.890700
```

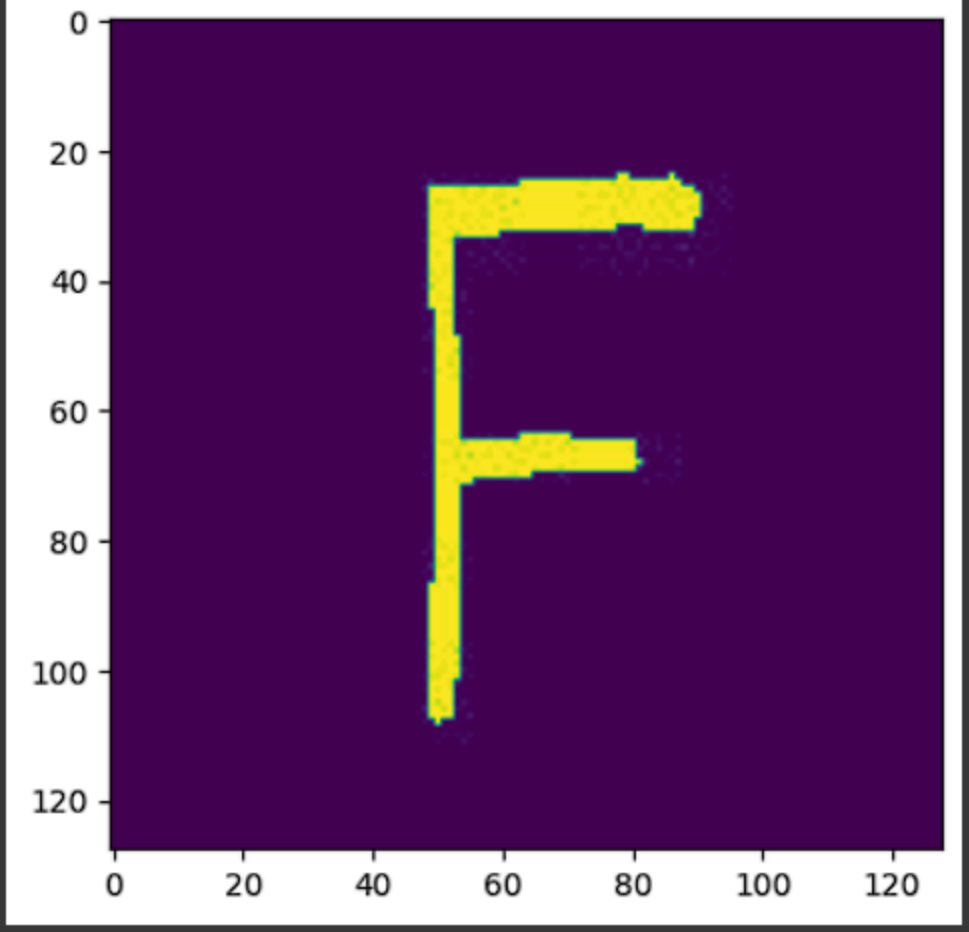
This considerably increased the score. It can also be changed as follows. Here, I changed the number of iterations and hidden layer values to see whether I would get a higher score. The score was increased from 95 to 96.

```
# Change some of the values in the below statement and re-run to see how they
# affect performance!
mlp2 = MLPClassifier(hidden_layer_sizes=(200,200,200,200,200,), max_iter=30, alpha=1e-4,
                     solver='sgd', verbose=10, tol=1e-4, random_state=1,
                     learning_rate_init=.1)
mlp2.fit(X_train, y_train)
print("Training set score: %f" % mlp2.score(X_train, y_train))
print("Test set score: %f" % mlp2.score(X_test, y_test))
```

```
Iteration 23, loss = 0.12740098
Iteration 24, loss = 0.11985547
Iteration 25, loss = 0.12294470
Iteration 26, loss = 0.11664134
Iteration 27, loss = 0.12189476
Iteration 28, loss = 0.11680624
Iteration 29, loss = 0.12311795
Iteration 30, loss = 0.11334774
/usr/local/lib/python3.10/dist-packages/sklearn/
  warnings.warn(
Training set score: 0.960617
Test set score: 0.896000
```

Next is to get the AI to read our own handwriting. Initially, letters are written on a paper and scanned, but there is a size issue as these images are much larger than the AI can read. After the modifications, check a certain index and see how it looks.

```
Imported the scanned images.
<matplotlib.image.AxesImage at 0x7db2e9127040>
```



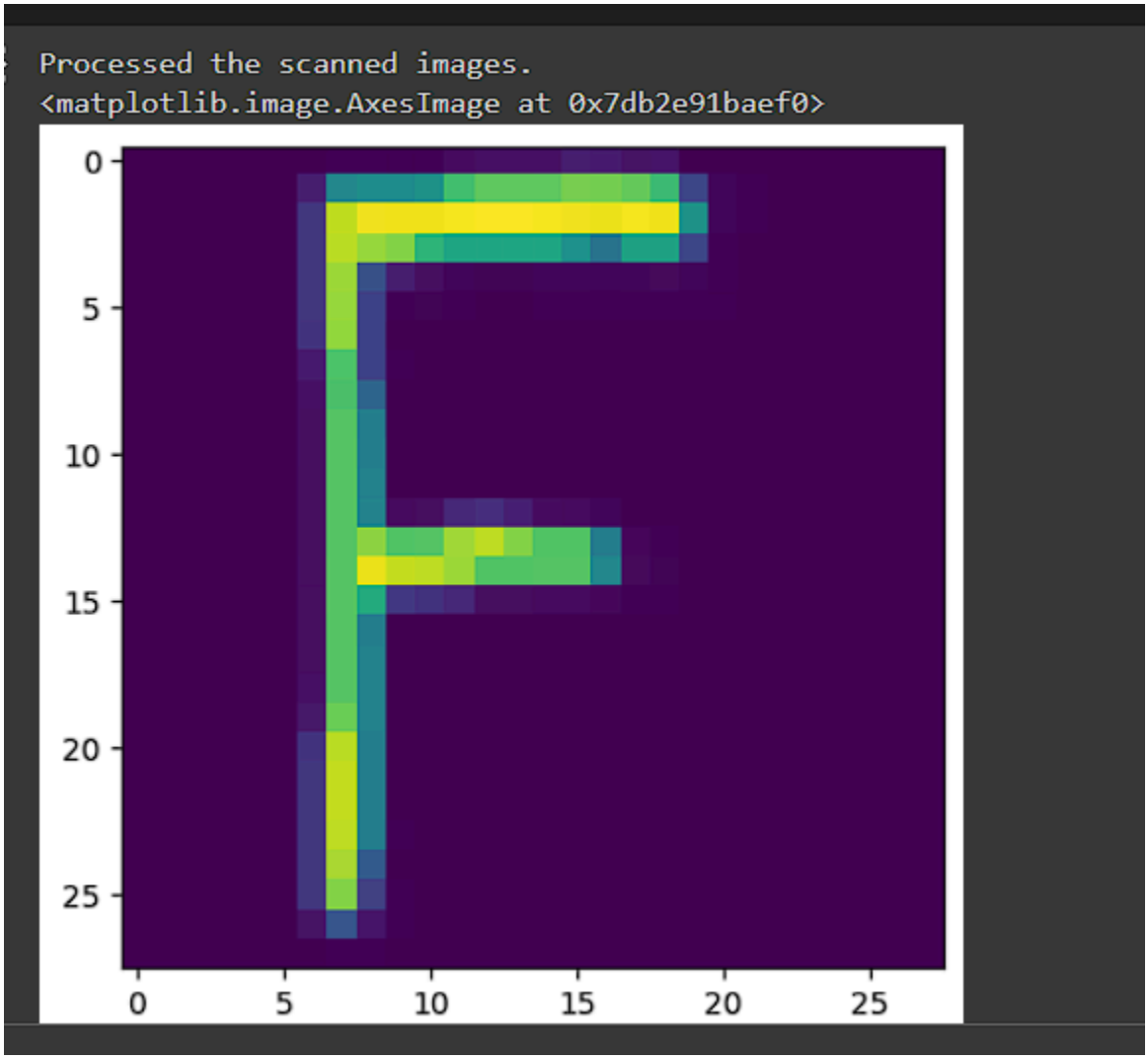
This looks very clean for a human but the AI has issues interpreting it.

If tested now, this type of wrong output can be seen due to how different this image is from the EMNIST training data.

```
the fault tn our rgwfk sypplies f ffll fn luye rhe way yguk batteky oies slowlr aao then all at oncf
```

Essentially, the strokes need to be more blurry, the letter has to be at the very center of the image and the rest has to be cropped, and resized to be

28x28 pixels. This below is the processed image with the modifications. This should be easier for the AI to read.



More words can certainly be identified now. Despite the fact that our neural network didn't achieve 100% accuracy, it is anticipated to exhibit a comparable error rate in this context, possibly even slightly higher, given the distinct size at which these letters were originally generated. However,

by considering the context and being aware of the letters that are prone to be confused with each other, the narrative remains comprehensible.