CS156: Second pipeline

NEWLY ADDED:

EXECUTIVE SUMMARY:

This project creates an optical character reader for Urdu, which is my native language. I do so by making use of existing data on the internet and feeding it to a Convuluted Neural Network.

In section one I explained the data and imported all neccessary libraries. Section two focused on converting this data to python readable format and section three involved pre-processing the data. I added a new part to section three where I use data augmentation to create new data which is then used for the confusion matrix in section seven. I added a psedocode and a mathematical explaination to section 4 with equations and I also explained what each variable meant, I then defined the model. In section six I trained the model and in section seven I generated predictions from it. Then I proceeded to visualize the results from my model.

Section 1:

- Explained the data: The need for data was extensive and collection was very time and labour extensive. Thus this model leverages some already built data-sets trained on languages that are written in the same script, like Arabic or Persian.(Urdu, Persian and Arabic are three completely different languages but they are written in the same script and thus make no difference for an OCR). The data sets were found on kaggle or other open-source platforms. Using such datasets was approved by the professor. I included the extra alphabets that are not already found in Arabic myself. The data includes pictures of alphabets in Urdu (My native language and one of the official languages spoken in Pakistan). I sampled the data to first incorporate some easier alphabets in my model and then slowly as my model got more

robust, I made use of more and more elements that were similar to each other or could be confusing for the model.

خ	ح	چ	ج	ث	ٹ	ت	پ	ب	ا
xē ख़े	baṙī ḥē बड़ी हे	čē चे	jīm जीम	s̱ē से	ṫē टे	tē ते	pē पे	bē बे	alif अलिफ़
ص	ش	س	ژ	ز	ڑ	ر	ذ	ڈ	د
ṣuād सुआद	šīn शीन	sīn सीन	žē झ़े	zē ज़े	ṙē ड़े	rē रे	ẕāl ज़ाल	ḋāl डाल	dāl दाल
ل	گ	ک	ق	ف	غ	ع	ظ	ط	ض
lām लाम	gāf गाफ़	kāf काफ़	qāf क़ाफ़	fē फ़े	ġǣn ग़ैन	ʻǣn ऐन	ẓōē ज़ोए	ṭōē तोए	źuād ज़ुआद
		ے	ی	ء	ھ	ہ	و	ن	م
		baṙī yē बड़ी ये	čẖōṫī yē छोटी ये	hamzah हमज़ा	dō-čašmī ḥē दोचश्मी हे	čẖōṫī hē छोटी हे	wāō वाओ	nūn नून	mīm मीम
۰	۱	۲	۳	۴	۵	۶	۷	۸	۹
० 0 ṣifar सिफ़र	१ 1 ēk एक	२ 2 dō दो	३ 3 tīn तीन	४ 4 čār चार	५ 5 pāⁿč पांच	६ 6 čẖē छे	७ 7 sāt सात	८ 8 āṫẖ आठ	९ 9 nō̤ नौ

- Imported all the neccessary libraries.

```
#libraries
import numpy as np
import pandas as pd
import tensorflow as tf
import seaborn as sns
#from tensorflow.keras.utils import to_categorica
import os
from zipfile import ZipFile
from io import BytesIO
from google.colab import files
from sklearn.metrics import confusion_matrix

# Virsualization for data
import matplotlib.pyplot as plt
from tensorflow.keras.utils import to_categorical

# import Callbacks for EarlySttoping
from tensorflow.keras.callbacks import ReduceLROnPlateau, EarlyStopping


# Data Augmentation (THis is bonus :) )
from tensorflow.keras.preprocessing.image import ImageDataGenerator
```

Section 2:

- Contains well-commented code for converting this data to python readable format loading this data into an appropriate data structure which in this case was a pandas dataframe.

```
local_ref  = os.path.join('letters.zip')
root_dir = os.path.join('tmp')



# Upload the letters.zip file using the file uploader in Colab
uploaded = files.upload()

# Specify the zip file name
# zip_file_name = '/content/letters.zip'

# Read the CSV files from the zip archive
# with ZipFile(BytesIO(uploaded[zip_file_name]), 'r') as zip_file:
uploaded_file_content = list(uploaded.values())[0]
# Get the content of the first (and only) uploaded file

with ZipFile(BytesIO(uploaded_file_content), 'r') as zip_file:
    # Your code for reading CSV files from the zip archive
    # Assuming your CSV files are directly in the root of the zip file
    with zip_file.open('csvTrainImages 13440x1024.csv') as file:
        df_train = pd.read_csv(file, header=None)

    with zip_file.open('csvTestImages 3360x1024.csv') as file:
        df_test = pd.read_csv(file, header=None)

    with zip_file.open('csvTrainLabel 13440x1.csv') as file:
        train_label = pd.read_csv(file, header=None)

    #  line for reading the test label file
    with zip_file.open('csvTestLabel 3360x1.csv') as file:
        test_label = pd.read_csv(file, header=None)
```

```
Choose Files No file chosen    Upload widget is only available when the cell has been executed in the current browser session. Please rerun this cell to enable.
Saving letters.zip to letters.zip
```

Section 3:

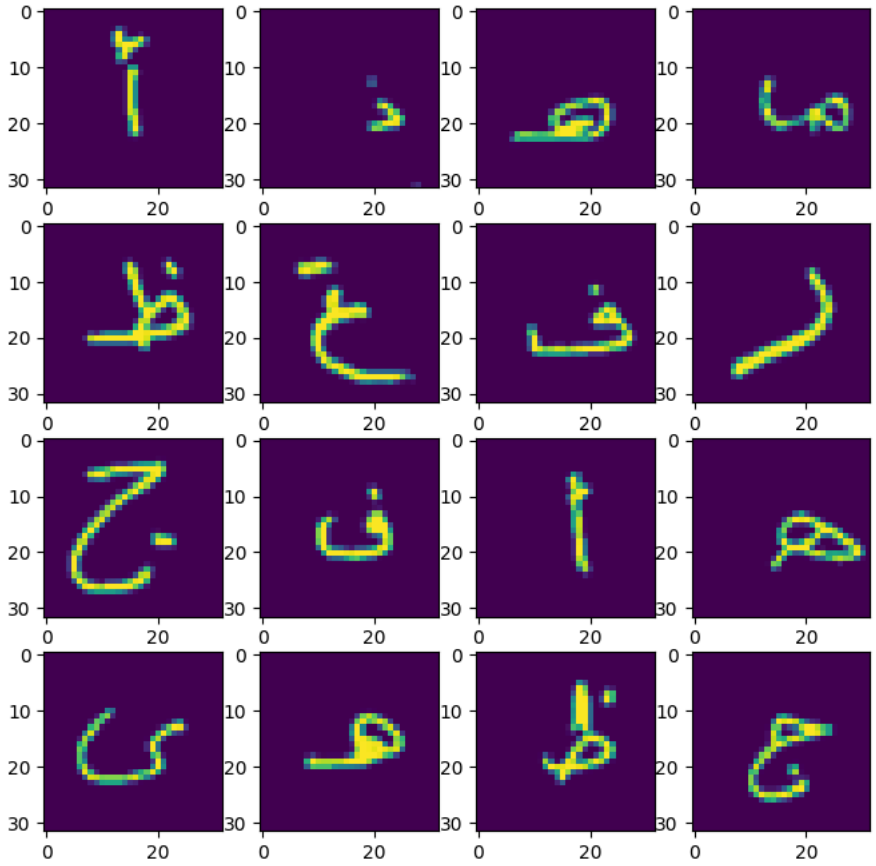
- Explaining any necessary cleaning, pre-processing, and feature engineering the data required: Not a lot of pre-processing was required. Data Augmentation was done by setting up data generators for training and validation sets. The pixel values of the images were scaled to be in the range [0, 1]. Which is a common practice to make training more stable. The code also randomly rotates the images by up to 40 degrees. Shear mapping is also applied in which one coordinate is held fixed and the other coordinates are shifted. The code also augmented the data by randomly shifting the width and height of the images by up to 20% and fliped images horizontally. The CNN will be trained on a combination of the original data and the augmented data. The data generator generates batches of augmented images on-the-fly during each training iteration.
- Code block doing these steps included below.
- Basic exploratory data analysis performed along with visualizing some samples and appropriate descriptive statistics.

```
x_train = np.array(df_train).reshape(-1 , 32 , 32 , 1)
y_train = to_categorical(np.array(train_label))
x_test = np.array(df_test).reshape(-1 , 32 , 32 , 1)
y_test = to_categorical(np.array(test_label))
y_train = y_train[: , 1:]
y_test = y_test[: , 1:]
```

Newly Added:

Data Augmentation

```
train_gen = ImageDataGenerator(rescale = 1./255.,
                               rotation_range= 40,
                               shear_range=0.2,
                               width_shift_range=0.2,
                               height_shift_range=0.2,
                               fill_mode='nearest',
                               horizontal_flip=True)
train_data_gen = train_gen.flow(x_train  , y_train , batch_size = 500)
validation_gen = ImageDataGenerator(rescale = 1./255.,
                               rotation_range= 40,
                               shear_range=0.2,
                               width_shift_range=0.2,
                               height_shift_range=0.2,
                               fill_mode='nearest',
                               horizontal_flip=True)
test_data_gen = validation_gen.flow(x_test , y_test , batch_size = 500)


random_letters = df_train.sample(16)
fig = plt.gcf()
fig.set_size_inches(8 , 8)

for letter in range(16):
    sub = plt.subplot(4,4 , letter + 1)
    img = np.array(random_letters.iloc[letter])
    img = img.reshape(32 , 32) # 1024 = 32 * 32
    plt.imshow(img.T)

plt.show()
```



```
print("df_train:")
print(df_train.head())

print("\ndf_test:")
print(df_test.head())
```

```
df_train:
   0     1     2     3     4     5     6     7     8     9     ...  1014  \
0     0     0     0     0     0     0     0     0     0     0  ...     0
1     0     0     0     0     0     0     0     0     0     0  ...     0
```

```
2     0     0     0     0     0     0     0     0     0     0  ...     0
3     0     0     0     0     0     0     0     0     0     0  ...     0
4     0     0     0     0     0     0     0     0     0     0  ...     0

   1015  1016  1017  1018  1019  1020  1021  1022  1023
0     0     0     0     0     0     0     0     0     0
1     0     0     0     0     0     0     0     0     0
2     0     0     0     0     0     0     0     0     0
3     0     0     0     0     0     0     0     0     0
4     0     0     0     0     0     0     0     0     0

[5 rows x 1024 columns]

df_test:
   0     1     2     3     4     5     6     7     8     9     ...  1014  \
0     0     0     0     0     0     0     0     0     0     0  ...     0
1     0     0     0     0     0     0     0     0     0     0  ...     0
2     0     0     0     0     0     0     0     0     0     0  ...     0
3     0     0     0     0     0     0     0     0     0     0  ...     0
4     0     0     0     0     0     0     0     0     0     0  ...     0

   1015  1016  1017  1018  1019  1020  1021  1022  1023
0     0     0     0     0     0     0     0     0     0
1     0     0     0     0     0     0     0     0     0
2     0     0     0     0     0     0     0     0     0
3     0     0     0     0     0     0     0     0     0
4     0     0     0     0     0     0     0     0     0

[5 rows x 1024 columns]
```

```
print("df_train shape:", df_train.shape)
print("df_test shape:", df_test.shape)
```

```
df_train shape: (13440, 1024)
df_test shape: (3360, 1024)
```

```
print("df_train summary:")
print(df_train.describe())

print("\ndf_test summary:")
print(df_test.describe())
```

```
50%        0.000000      0.000000      0.000000      0.000000      0.000000
75%        0.000000      0.000000      0.000000      0.000000      0.000000
```

```
mean      0.355655     0.375595     0.421726     0.274107     0.232440
std       6.143127     6.910821     7.868210     5.439443     4.497343
min       0.000000     0.000000     0.000000     0.000000     0.000000
25%       0.000000     0.000000     0.000000     0.000000     0.000000
50%       0.000000     0.000000     0.000000     0.000000     0.000000
75%       0.000000     0.000000     0.000000     0.000000     0.000000
max     255.000000   237.000000   244.000000   245.000000   177.000000

              1019         1020         1021         1022         1023
count  3360.000000  3360.000000  3360.000000  3360.000000  3360.000000
mean      0.301786     0.499405     0.298512     0.392857     1.942262
std       5.679072     8.817794     5.657528     4.687604    11.008141
min       0.000000     0.000000     0.000000     0.000000     0.000000
25%       0.000000     0.000000     0.000000     0.000000     0.000000
50%       0.000000     0.000000     0.000000     0.000000     0.000000
75%       0.000000     0.000000     0.000000     0.000000     0.000000
max     224.000000   249.000000   193.000000   112.000000   146.000000

[8 rows x 1024 columns]
```

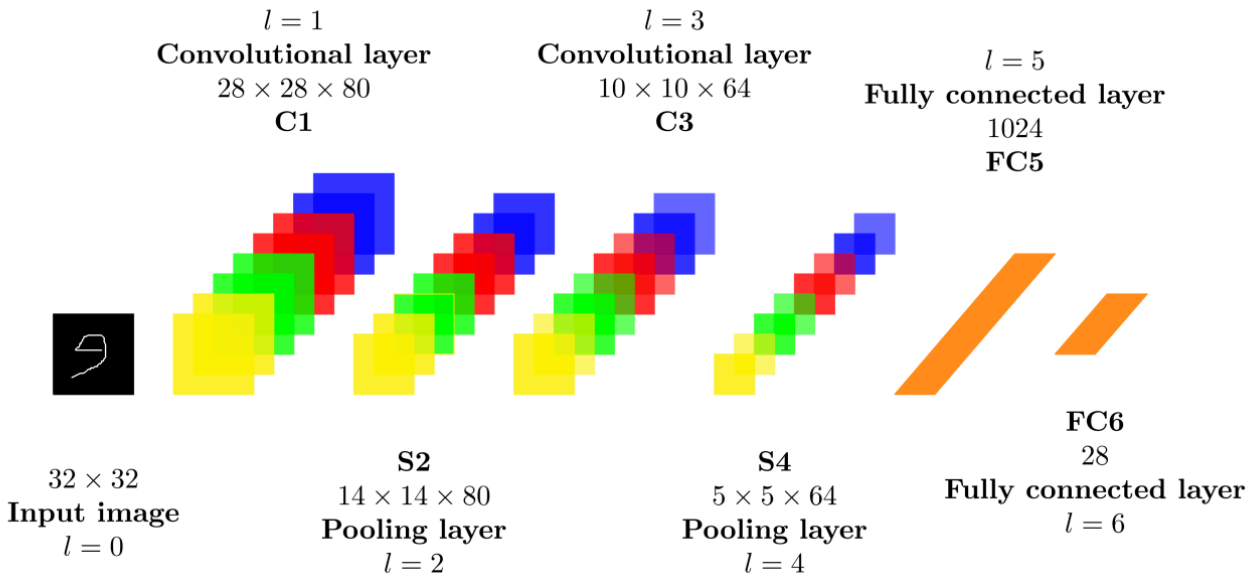
Section 4:

- Discussing the task: Classification will be conducted on the data using a Convoluted Neural Network-CNN. OCR is primarily used to recognize and extract text from images or documents. The classification aspect of OCR involves determining whether a given region or element in an image contains text or not. This initial classification step is crucial to identify areas of interest that need further processing for text extraction.
- Well commented code block that splits data into training and test sets included below.

```
# Convert the training data (images) to a NumPy array and reshape it
# -1 in the reshape function infers the size of one of the dimensions automatically
x_train = np.array(df_train).reshape(-1, 32, 32, 1)

# Convert the training labels to categorical format
# For example, if there are 10 classes, each label becomes a one-hot encoded vector
y_train = to_categorical(np.array(train_label))

# Convert the testing data (images) to a NumPy array and reshape it
x_test = np.array(df_test).reshape(-1, 32, 32, 1)

# Convert the testing labels to categorical format
y_test = to_categorical(np.array(test_label))
```

Section 5:

- Discussed model selection: CNNs were used because they make use of hierarchical feature learning which allowes them to capture complex patterns and variations in characters. They are also capable of recognizing patterns regardless of their position in the image, providing a degree of translation invariance. A well-trained CNN can also generalize well to recognize characters in different fonts, styles, and orientations. Another major advantage of CNN is the use of shared weight in convolutional layers, which means that the same filter is used for each input in the layer. The share weight reduces parameter number and improves performance
- Model initialization and construction in a well-commented code block included below.
- Discussion of the model's mathematical underpinnings: I included details of the mathematical underpinnings below, along with a well explained psuedocode.



NEWLY ADDED

EXPLAINED MATH:

A convoluted network is made up of different kinds of layers and functions that we need for different purposes. Below are short discriptions of them.

- Input Layer: The input layer receives the raw data, such as an image or a sequence of data, and passes it to the subsequent layers for processing.The size and shape of the input layer depend on the nature of the input data.
- Convolutional Layers:Convolutional layers apply convolution operations to the input data using filters or kernels. This helps the network learn local patterns and spatial hierarchies in the input.The depth of the filters increases as we go deeper into the network, allowing the model to capture increasingly complex features.
- Activation Function:Activation functions introduce non-linearity to the model. ReLU (Rectified Linear Unit) is commonly used to add non-linearity, allowing the network to learn complex patterns and relationships. Non-linear activation functions enable the model to approximate more complex functions.
- Pooling Layers: Pooling layers reduce the spatial dimensions of the feature maps generated by convolutional layers. MaxPooling or AveragePooling is commonly used to downsample the data and retain the most important information. Pooling helps reduce computational complexity and makes the model more robust to variations in position and scale.
- Batch Normalization: Batch Normalization normalizes the input of a layer, improving training stability and convergence. It can act as a form of regularization and accelerates training. It is often applied after the activation function and before subsequent layers.
- Flattening: Flattening converts the multi-dimensional output from the previous layers into a one-dimensional vector. This vector serves as the input for the fully connected layers. Flattening retains the hierarchical features learned by the convolutional layers in a format suitable for dense layers.
- Fully Connected (Dense) Layers: Fully connected layers capture global patterns and relationships in the data. They transform the high-level features extracted by the convolutional layers into predictions for the target task. The number of neurons in the output layer corresponds to the number of classes in classification tasks.
- Output Layer: The output layer produces the final predictions for the task at hand. The number of neurons in this layer corresponds to the number of classes in a classification task. For multi-class classification, a softmax activation function is commonly used to convert the raw output into class probabilities.

Here is the math behind all the layers that I used in my model.

1. **Convolutional Layer 1:**
    - This layer performs a convolution operation on the input image (X) using 16 filters of size (3, 3).
    - The convolution operation is followed by the ReLU activation function.

$$\text{Conv1}(X)_{i,j,k} = \sigma\left(\sum_{m=0}^{2}\sum_{n=0}^{2} X_{i+m,j+n,0} \cdot W_{m,n,k} + b_k\right)$$

- $(X_{i+m,j+n,0})$ represents the pixel values of the input image at position $((i + m, j + n, 0))$.

- ($W_{m,n,k}$) represents the weights of the convolutional filter at position ((m, n, k)).
- (b_k) is the bias term for the (k)-th filter.
- (σ) is the ReLU activation function.

2. **Max-Pooling Layer 1:**
    - This layer performs max-pooling on the output of the first convolutional layer with a pool size of (2, 2).

$$\text{MaxPool1}(X)_{i,j,k} = \max\left(X_{2i,2j,k}, X_{2i,2j+1,k}, X_{2i+1,2j,k}, X_{2i+1,2j+1,k}\right)$$

- ($X_{2i,2j,k}$) represents the pixel values at position (($2i$, $2j$, k)) in the output of Conv1.

3. **Batch Normalization Layer:**
    - Batch normalization normalizes the output of the previous layer by subtracting the mean ((\mu_k)) and dividing by the standard deviation (($sigma_k$)).
    - It then scales and shifts the normalized values using learnable parameters ($gamma_k$) and ($beta_k$).

$$\text{BatchNorm}(X)_{i,j,k} = \frac{X_{i,j,k} - \mu_k}{\sqrt{\sigma_k^2 + \epsilon}} \cdot \gamma_k + \beta_k$$

- $epsilon$ is a small constant for numerical stability.

4. **Convolutional Layer 2:**
    - This layer performs a second convolution on the output of the batch normalization layer using 32 filters of size (3, 3).
    - The convolution operation is followed by the ReLU activation function.

$$\text{Conv2}(X)_{i,j,k} = \sigma\left(\sum_{m=0}^{2}\sum_{n=0}^{2}\{\text{MaxPool1}\}(X)_{i+m,j+n,k'} \cdot W'_{m,n,k} + b'_k\right)$$

- (k') represents the channels from the output of the first convolutional layer.

5. **Max-Pooling Layer 2:**
    - This layer performs max-pooling on the output of the second convolutional layer with a pool size of (2, 2).

$$\text{MaxPool2}(X)_{i,j,k} = \max\left(\{\text{Conv2}\}(X)_{2i,2j,k}, \{\text{Conv2}\}(X)_{2i,2j+1,k}, \{\text{Conv2}\}(X)_{2i+1,2j,k}, \{\text{Conv2}\}(X)_{2i+1,2j+1,k}\right)$$

6. **Convolutional Layer 3:**
    - This layer performs a third convolution on the output of the second max-pooling layer using 64 filters of size (3, 3).
    - The convolution operation is followed by the ReLU activation function.

$$\text{Conv3}(X)_{i,j,k} = \sigma\left(\sum_{m=0}^{2}\sum_{n=0}^{2}\{\text{MaxPool2}\}(X)_{i+m,j+n,k'} \cdot W''_{m,n,k} + b''_k\right)$$

-(k'') represents the channels from the output of the second convolutional layer.

7. **Max-Pooling Layer 3:**
    - This layer performs max-pooling on the output of the third convolutional layer with a pool size of (2, 2).

$$\text{MaxPool3}(X)_{i,j,k} = \max\left(\{\text{Conv3}\}(X)_{2i,2j,k}, \{\text{Conv3}\}(X)_{2i,2j+1,k}, \{\text{Conv3}\}(X)_{2i+1,2j,k}, \{\text{Conv3}\}(X)_{2i+1,2j+1,k}\right)$$

8. **Flatten Layer:**
    - This layer flattens the output of the third max-pooling layer into a 1D tensor.

$$\text{Flatten}(X)_i = \{\text{MaxPool3}\}(X)_{\lfloor i/(32\times32)\rfloor,\lfloor(i \mod (32\times32))/32\rfloor,i \mod 32}$$

- It converts the 3D tensor to a 1D tensor by mapping the indices (i, j, k) to a single index (i).

9. **Dropout Layer:**
    - This layer randomly drops a fraction of neurons with a probability of 0.2 during training.

$$\text{Dropout}(X)_i = \begin{cases} X_i & \{\text{with probability }\}1 - 0.2 \\ 0 & \{\text{with probability }\}0.2 \end{cases}$$

10. **Dense Layer 1:**
    - This layer is a fully connected (dense) layer with 512 neurons and ReLU activation.

$$\{\text{Dense1}\}(X)_i = \sigma\left(\sum_{j=0}^{N}\{\text{Dropout}\}(X)_j \cdot W'''_{j,i} + b'''_i\right)$$

- N is the total number of neurons in the flattened layer.

11. **Dense Layer 2:**
    - This is the final dense layer with 28 neurons (for 28 classes in the Urdu alphabet)

NEWLY ADDED

EXPLAINED PSUDOCODE:

Create CNN Model Function: Define a function create_cnn_model to encapsulate the creation of the CNN model.

```
# Defined the CNN architecture
function create_cnn_model():
```

Initialize Sequential Model: Create an empty Sequential model and assign it to the variable model.

```
model = Sequential()
```

Convolutional Layer 1:

- Add a 2D convolutional layer with 16 filters, a kernel size of (3, 3), ReLU activation, and an input shape of (32, 32, 1).
- Add a 2D max pooling layer with a pool size of (2, 2).
- Add a batch normalization layer.

```
# Added Convolutional Layer 1
model.add(Conv2D(16, kernel_size=(3, 3), activation='relu', input_shape=(32, 32, 1)))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(BatchNormalization())
```

Convolutional Layer 2:

- Add another 2D convolutional layer with 32 filters and a kernel size of (3, 3).
- Add another 2D max pooling layer with a pool size of (2, 2).

```
# Added Convolutional Layer 2
model.add(Conv2D(32, kernel_size=(3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
```

Convolutional Layer 3:

- Add a third 2D convolutional layer with 64 filters and a kernel size of (3, 3).
- Add a third 2D max pooling layer with a pool size of (2, 2).

```
# Added Convolutional Layer 3
model.add(Conv2D(64, kernel_size=(3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
```

Flatten Layer: Add a flatten layer to transform the 3D output to a 1D vector.

```
# Added Flatten Layer
model.add(Flatten())
```

Dropout Layer: Add a dropout layer with a dropout rate of 0.2 for regularization.

```
# Added Dropout Layer
model.add(Dropout(0.2))
```

Dense Layer 1: Add a dense layer with 512 neurons and ReLU activation.

```
# Added Dense Layer 1
model.add(Dense(512, activation='relu'))
```

Dense Layer 2:

Add a dense layer with 28 neurons (representing the Urdu alphabet) and softmax activation.

```
 # Added Dense Layer 2
 model.add(Dense(28, activation='softmax'))

 return model
```

Compile the Model:

Compile the model using the Adam optimizer, categorical cross-entropy loss, and accuracy as the metric.

```
 # Cleared any existing TensorFlow session
 tf.keras.backend.clear_session()

 # Created the CNN model
 cnn_model = create_cnn_model()

 # Compiled the model
 cnn_model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
```

Display Model Summary: Print a summary of the model architecture, including the type of layers, output shapes, and the number of parameters.

```
 # Displayed the model summary
 cnn_model.summary()
```

```
tf.keras.backend.clear_session()
model = tf.keras.models.Sequential([
    tf.keras.layers.Conv2D(16 , (3,3) , activation = 'relu'  , input_shape = (32 , 32 , 1)),
    tf.keras.layers.MaxPooling2D(2,2),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Conv2D(32 , (3,3) , activation = 'relu'),
    tf.keras.layers.MaxPooling2D(2,2),
    tf.keras.layers.Conv2D(64 , (3,3) , activation = 'relu'),
    tf.keras.layers.MaxPooling2D(2,2),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(512, activation = 'relu'),
    tf.keras.layers.Dense(28 , activation = 'softmax') #only 28 letters in the arabic alphabet
])
```

```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 30, 30, 16)        160

 max_pooling2d (MaxPooling2  (None, 15, 15, 16)        0
 D)

 batch_normalization (Batch  (None, 15, 15, 16)        64
 Normalization)

 conv2d_1 (Conv2D)           (None, 13, 13, 32)        4640

 max_pooling2d_1 (MaxPoolin  (None, 6, 6, 32)          0
 g2D)

 conv2d_2 (Conv2D)           (None, 4, 4, 64)          18496

 max_pooling2d_2 (MaxPoolin  (None, 2, 2, 64)          0
 g2D)

 flatten (Flatten)           (None, 256)               0

 dropout (Dropout)           (None, 256)               0

 dense (Dense)               (None, 512)               131584

 dense_1 (Dense)             (None, 28)                14364

=================================================================
Total params: 169308 (661.36 KB)
Trainable params: 169276 (661.23 KB)
```

```
Non-trainable params: 32 (128.00 Byte)
_________________________________________________________________
```

Section 6:

Trained the model, including any necessary cross validation and hyperparameter tuning.

```
model.compile(loss = tf.keras.losses.categorical_crossentropy, optimizer= tf.keras.optimizers.Adam() , metrics = ['accuracy'])
```

```
es_callback = tf.keras.callbacks.EarlyStopping(monitor = 'accuracy' , patience=6)
history = model.fit(train_data_gen , validation_data= validation_data_gen , epochs = 100 , callbacks=[es_callback] )
```

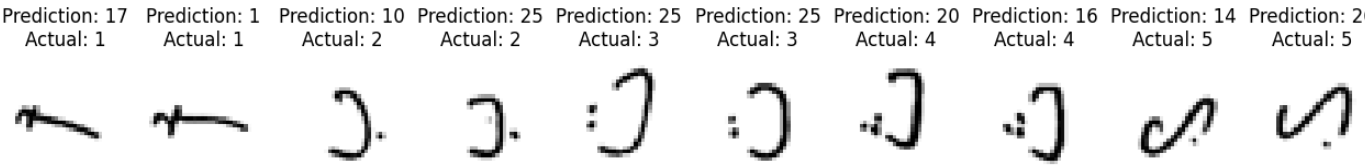
```
Epoch 51/100
269/269 [==============================] - 16s 58ms/step - loss: 0.5240 - accuracy: 0.8268 - val_loss: 0.3102 - val_accuracy
Epoch 52/100
269/269 [==============================] - 15s 57ms/step - loss: 0.5092 - accuracy: 0.8291 - val_loss: 0.3022 - val_accuracy
Epoch 53/100
269/269 [==============================] - 17s 64ms/step - loss: 0.5073 - accuracy: 0.8295 - val_loss: 0.3040 - val_accuracy
Epoch 54/100
269/269 [==============================] - 16s 58ms/step - loss: 0.5282 - accuracy: 0.8230 - val_loss: 0.4012 - val_accuracy
Epoch 55/100
269/269 [==============================] - 15s 55ms/step - loss: 0.5130 - accuracy: 0.8304 - val_loss: 0.3065 - val_accuracy
Epoch 56/100
269/269 [==============================] - 15s 55ms/step - loss: 0.5029 - accuracy: 0.8286 - val_loss: 0.3592 - val_accuracy
Epoch 57/100
269/269 [==============================] - 16s 58ms/step - loss: 0.5193 - accuracy: 0.8279 - val_loss: 0.3644 - val_accuracy
Epoch 58/100
269/269 [==============================] - 16s 58ms/step - loss: 0.5044 - accuracy: 0.8319 - val_loss: 0.2841 - val_accuracy
Epoch 59/100
269/269 [==============================] - 18s 68ms/step - loss: 0.4956 - accuracy: 0.8368 - val_loss: 0.2654 - val_accuracy
Epoch 60/100
269/269 [==============================] - 16s 58ms/step - loss: 0.5008 - accuracy: 0.8368 - val_loss: 0.3659 - val_accuracy
Epoch 61/100
269/269 [==============================] - 16s 58ms/step - loss: 0.5108 - accuracy: 0.8282 - val_loss: 0.2635 - val_accuracy
Epoch 62/100
269/269 [==============================] - 18s 67ms/step - loss: 0.5017 - accuracy: 0.8328 - val_loss: 0.4119 - val_accuracy
Epoch 63/100
269/269 [==============================] - 18s 66ms/step - loss: 0.5059 - accuracy: 0.8339 - val_loss: 0.2360 - val_accuracy
Epoch 64/100
269/269 [==============================] - 15s 56ms/step - loss: 0.4842 - accuracy: 0.8381 - val_loss: 2.4612 - val_accuracy
Epoch 65/100
269/269 [==============================] - 16s 58ms/step - loss: 0.4907 - accuracy: 0.8368 - val_loss: 0.6800 - val_accuracy
Epoch 66/100
269/269 [==============================] - 15s 56ms/step - loss: 0.4982 - accuracy: 0.8333 - val_loss: 0.2929 - val_accuracy
Epoch 67/100
269/269 [==============================] - 19s 72ms/step - loss: 0.4975 - accuracy: 0.8350 - val_loss: 0.2509 - val_accuracy
Epoch 68/100
269/269 [==============================] - 18s 66ms/step - loss: 0.4907 - accuracy: 0.8353 - val_loss: 0.2256 - val_accuracy
Epoch 69/100
269/269 [==============================] - 16s 58ms/step - loss: 0.4835 - accuracy: 0.8392 - val_loss: 0.4295 - val_accuracy
Epoch 70/100
269/269 [==============================] - 16s 58ms/step - loss: 0.4736 - accuracy: 0.8382 - val_loss: 0.2532 - val_accuracy
Epoch 71/100
269/269 [==============================] - 16s 58ms/step - loss: 0.4902 - accuracy: 0.8365 - val_loss: 0.5487 - val_accuracy
Epoch 72/100
269/269 [==============================] - 18s 67ms/step - loss: 0.4732 - accuracy: 0.8401 - val_loss: 0.3818 - val_accuracy
Epoch 73/100
269/269 [==============================] - 16s 58ms/step - loss: 0.4710 - accuracy: 0.8421 - val_loss: 0.6222 - val_accuracy
Epoch 74/100
269/269 [==============================] - 16s 59ms/step - loss: 0.4807 - accuracy: 0.8387 - val_loss: 0.2020 - val_accuracy
Epoch 75/100
269/269 [==============================] - 15s 57ms/step - loss: 0.4831 - accuracy: 0.8363 - val_loss: 0.4052 - val_accuracy
Epoch 76/100
269/269 [==============================] - 15s 56ms/step - loss: 0.4723 - accuracy: 0.8408 - val_loss: 0.4629 - val_accuracy
Epoch 77/100
269/269 [==============================] - 15s 56ms/step - loss: 0.4616 - accuracy: 0.8459 - val_loss: 0.2129 - val_accuracy
Epoch 78/100
269/269 [==============================] - 15s 56ms/step - loss: 0.4765 - accuracy: 0.8417 - val_loss: 0.2275 - val_accuracy
Epoch 79/100
121/269 [============>.................] - ETA: 8s - loss: 0.4776 - accuracy: 0.8422
```

Section 7:

Generated predictions for out of sample data, and computed appropriate performance metrics.

```
import numpy as np
import matplotlib.pyplot as plt


# Assuming you have already trained your model and obtained predictions
predict = model.predict(test_data_gen)
pred = [np.argmax(i) for i in predict]

# Plotting the images
_, axes = plt.subplots(nrows=1, ncols=10, figsize=(16, 4))
for ax, image, actual, prediction in zip(axes, x_test, y_test, pred):
    ax.set_axis_off()
    ax.imshow(image, cmap=plt.cm.gray_r)
    ax.set_title(f'Prediction: {prediction}\nActual: {np.argmax(actual)}')

plt.show()
```

```
7/7 [==============================] - 2s 221ms/step
```

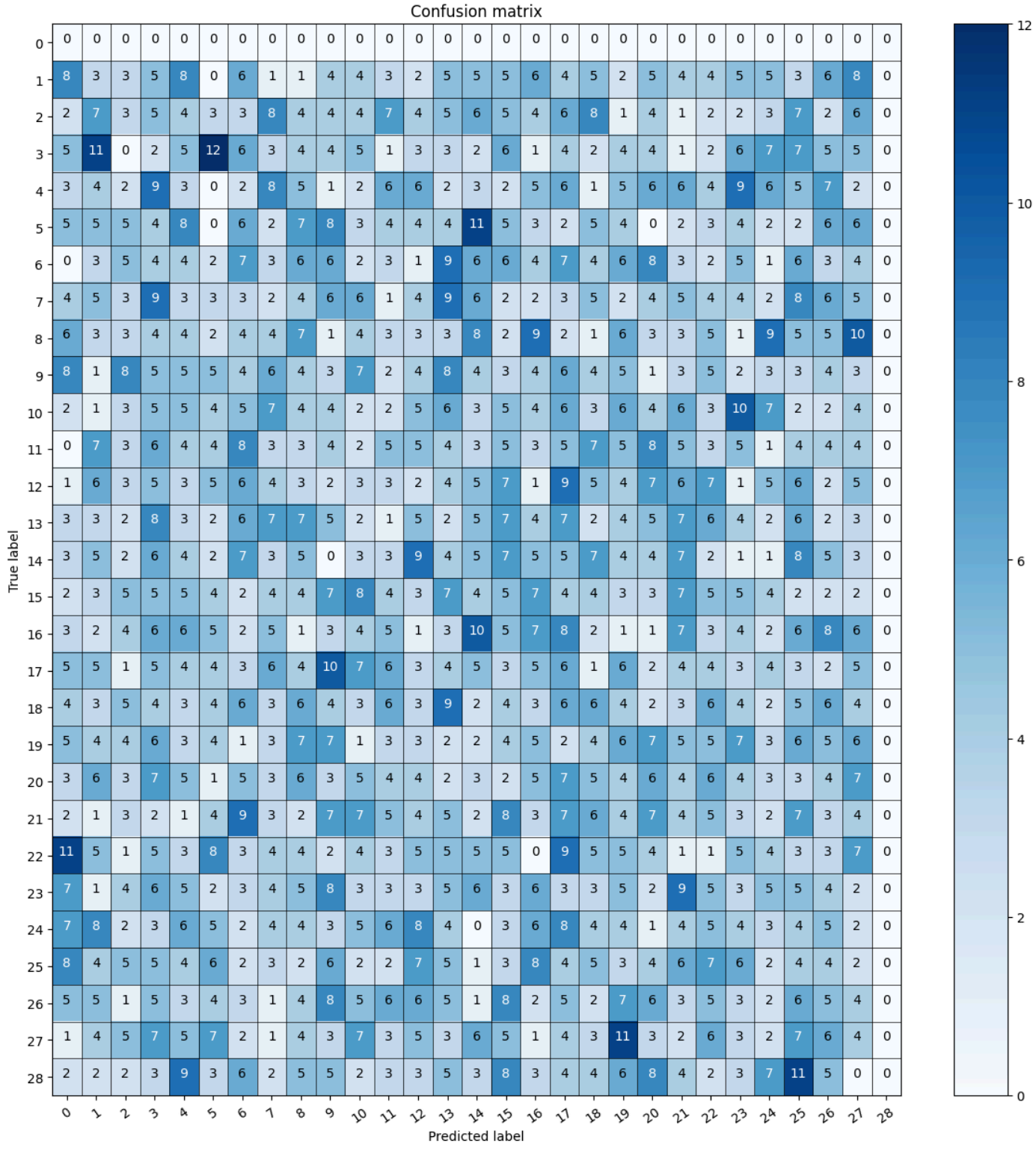


NEWLY ADDED

I used the Augmented data to test my model so I get less zeros across the diagonal of my confusion matrix.

```
from sklearn.metrics import confusion_matrix
import itertools
def plot_confusion_matrix(cm, classes,
                          normalize=False,
                          title='Confusion matrix',
                          cmap=plt.cm.Blues,figsize=(12, 8)):
    plt.figure(figsize=figsize)
    plt.imshow(cm, interpolation='nearest', cmap=cmap, aspect='auto')
    plt.title(title)
    plt.colorbar()

    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=35)
    plt.yticks(tick_marks, classes)

    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]

    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, f"{cm[i, j]:.2f}" if normalize else f"{cm[i, j]}",
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")

    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')
    for i in range(len(classes) - 1):
        plt.axhline(i + 0.5, color='black', linewidth=0.5)
        plt.axvline(i + 0.5, color='black', linewidth=0.5)
    plt.xticks(np.arange(len(classes)), classes)  # Set the x-axis ticks to represent values from 0 to 28
    plt.yticks(np.arange(len(classes)), classes)  # Set the y-axis ticks to represent values from 0 to 28
    plt.subplots_adjust(top=1.5)  # Adjust the top margin
# compute the confusion matrix
conf_matrix = confusion_matrix(np.argmax(y_test, axis=1), pred)
# plot the confusion matrix
plot_confusion_matrix(conf_matrix, classes = range(29))
```

```
672/672 [==============================] - 3s 4ms/step - loss: 0.3505 - accuracy: 0.8792
[0.3505166471004486, 0.8791666626930237]
```

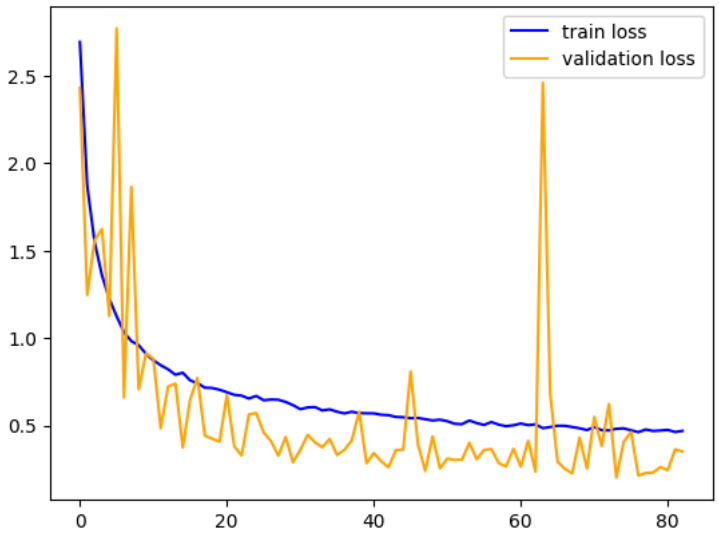


Section 8:

- Visualized the results.
- Discussing the results: The results show us that the model is not overfitting as much anymore, since the training accuracy continues to improve with the validation accuracy. Similarly, decrease in training loss is also in-line with validation loss. This is something I fine-tuned

the model to avoid. We do see that the accuracy curves are not consistent across epochs. Sudden jumps or drops in accuracy might indicate issues such as learning rate problems, data preprocessing errors, or model architecture issues. Which means the model is still overfitting the data but a lot less than how it did initially. I tried to minimize these by fine-tuning the parameters but it is still not ideal.

```
loss = history.history["loss"]
val_loss= history.history["val_loss"]

acc = history.history["accuracy"]
val_acc  = history.history["val_accuracy"]

plt.plot(loss , 'b' , label = 'train loss')
plt.plot(val_loss , 'orange' , label = 'validation loss')
plt.legend()
plt.show()
```




```
plt.plot(acc , 'b' , label = 'train accuracy')
plt.plot(val_acc , 'orange' , label = 'validation accuracy')
plt.legend()
plt.show()
```