Cairo University
Faculty of Computers and Artificial Intelligence

Machine Learning

Project

Section IS S1&S2

Team Members

Name	ID
Salma Mamdoh Sabry	**20210162**
Roaa Talat Mohamed	**20210138**
Youssef Ehab Mohamed	**20210466**
Zeyad Ehab Maamoun	**20211043**
Youssef Mohamed Salah Eldin Anwar	**20210483**

About the Dataset: A-Z Handwritten Alphabets

This dataset provides handwritten representations of English alphabets (A-Z), captured in grayscale images stored in a structured .csv format. It is designed to support machine learning projects, particularly in training models for handwritten character recognition. The dataset includes over **370,000 samples** of alphabets.

Key Information

Dataset Overview:

- **Size**: 370,000+ images of handwritten English alphabets.
- **Format**: Each image is represented as a row in a .csv file, with pixel values and corresponding labels.
- **Pixel Data**: Grayscale intensity values ranging from 0 (black) to 255 (white).
- **Labels**: Indicate the corresponding alphabet (A-Z).

Image Details:

- **Resolution**: Each image is resized to 28x28 pixels.
- **Center-Fitting**: Alphabets are centered in a 20x20 pixel bounding box.
- **Noisy Samples**: Some noisy images may be present in the dataset.

Column Descriptions

1. **Pixel Values**: 784 columns (28x28 pixels) representing the grayscale intensity of each pixel.
2. **Label**: Indicates the alphabet associated with the image, represented numerically (e.g., 0 = A, 1 = B, ..., 25 = Z).

1- Data Processing

1- Load Data

Preprocessing

[2]:
```
df = pd.read_csv('A_Z Handwritten Data.csv')
df.head()
```

[2]:

	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	...	0.639	0.640	0.641	0.642	0.643	0.644	0.645	0.646	0.647	0.648
0	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0

5 rows × 785 columns

2- Explore Dataset

[5]:
```
print("Dataset Information: \n")
df.info()
```

```
Dataset Information:


<class 'pandas.core.frame.DataFrame'>

RangeIndex: 372450 entries, 0 to 372449

Columns: 785 entries, 0 to 0.648

dtypes: int64(785)

memory usage: 2.2 GB
```

3- Rename Label Column

[6]:
```
print("Dataset Shape: \n")
df.shape
```

```
Dataset Shape:
```

[6]:
```
(372450, 785)
```

[3]:
```
# Rename columns (first column is the label, the rest are features)
df.rename(columns={'0':'label'}, inplace=True)
print(df.columns)
```

```
Index(['label', '0.1', '0.2', '0.3', '0.4', '0.5', '0.6', '0.7', '0.8', '0.9',
       ...
       '0.639', '0.640', '0.641', '0.642', '0.643', '0.644', '0.645', '0.646',
       '0.647', '0.648'],
      dtype='object', length=785)
```

4- Explore Unique Classes

Identify the number of unique classes

[9]:
```
# the first column contains the labels, let's check for unique values
unique_classes = df['label'].nunique()

print(f"Number of unique classes: {unique_classes}")
```

```
Number of unique classes: 26
```

[16]:
```
# Map labels to alphabets
y_mapped = y.map(lambda x: chr(x + 65))
```

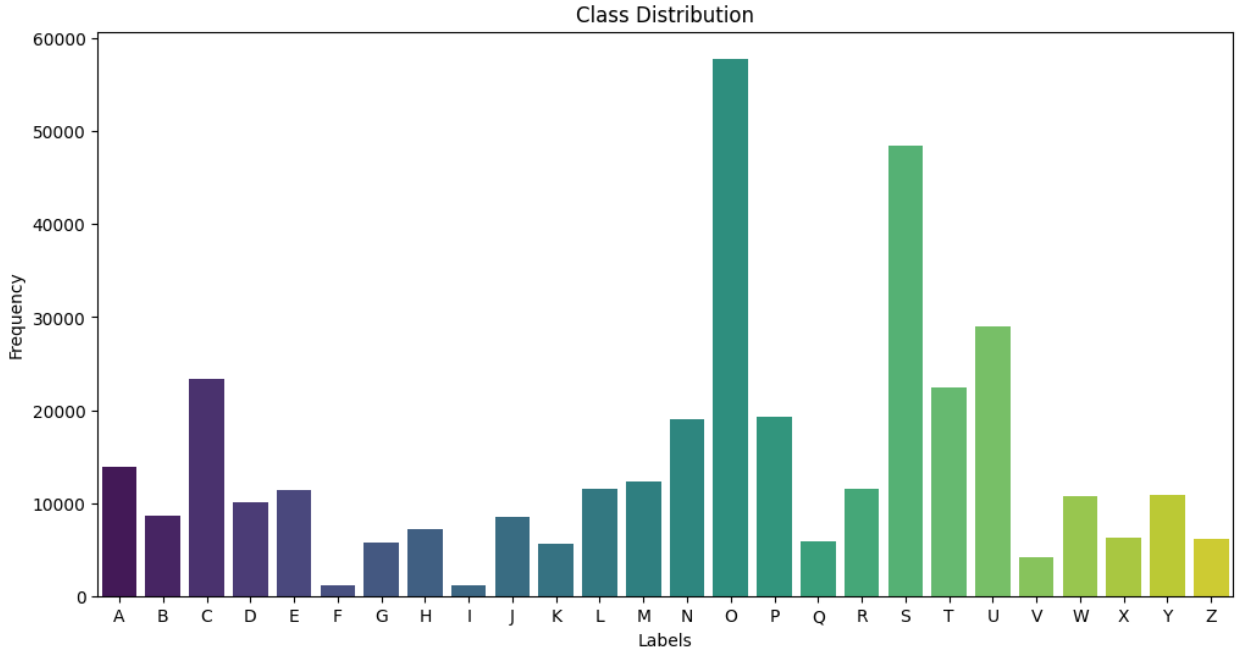
[17]:
```
# Show the distribution of labels (count of each class)
class_counts = df['label'].value_counts().sort_index()

# Print the sorted distribution with corresponding characters
print("\nClass distribution (sorted with corresponding characters):")
for label, count in class_counts.items():
    print(f"{chr(label + 65)}: {count}")
```

```
Class distribution (sorted with corresponding characters):

A: 13869

B: 8668

C: 23409

D: 10134

E: 11440

F: 1163

G: 5762

H: 7218

I: 1120

J: 8493

K: 5603

L: 11586

M: 12336

N: 19010

O: 57825

P: 19341

Q: 5812
```

5- Plot class Distribution

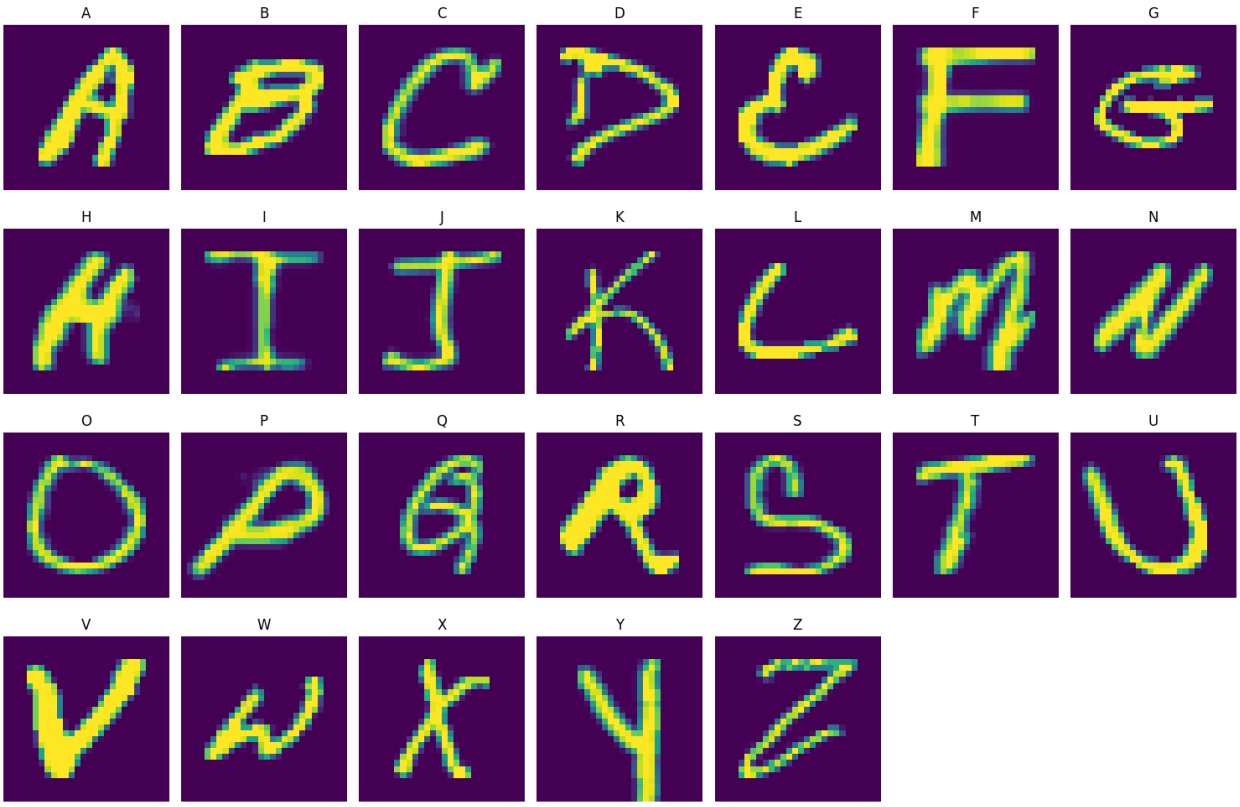
[11]:

```
# Plot class distribution
plt.figure(figsize=(12, 6))
sns.barplot(x=class_counts.index, y=class_counts.values, palette="viridis")
plt.title("Class Distribution")
plt.xlabel("Labels")
plt.ylabel("Frequency")
plt.xticks(range(26), [chr(i + 65) for i in range(26)])  # Convert numeric labels to alphabets
plt.show()
```



6- Visualize Samples From Data

[12]:
```
# Visualize samples for each class
plt.figure(figsize=(15, 10))
for label in range(26):
    sample = df[df['label'] == label].iloc[0, 1:].values.reshape(28, 28)
    plt.subplot(4, 7, label + 1)
    plt.imshow(sample, cmap='viridis')
    plt.axis('off')
    plt.title(chr(label + 65))
plt.tight_layout()
plt.show()
```



7- Split Data into Train and Test

[5]:
```
# Split features and labels
X = df.iloc[:, 1:]
y = df['label']
```

[6]:
```
# Split the dataset into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42, stratify=y)

print(f"Training set shape: {X_train.shape}, {y_train.shape}")
print(f"Test set shape: {X_test.shape}, {y_test.shape}")
```

```
Training set shape: (297960, 784), (297960,)

Test set shape: (74490, 784), (74490,)
```

8- Normalize Each Image

Normalize each image

[7]:
```
X_train = X_train.astype('float32') / 255.0  # Normalize to 0-1 range
X_test = X_test.astype('float32') / 255.0
```

[19]:
```
# 8. Normalization Validation: Verify that the images are properly normalized
print(f"Minimum value in normalized train data: {np.min(X_train)}")
print(f"Maximum value in normalized train data: {np.max(X_train)}")
```

```
Minimum value in normalized train data: 0.0

Maximum value in normalized train data: 1.0
```

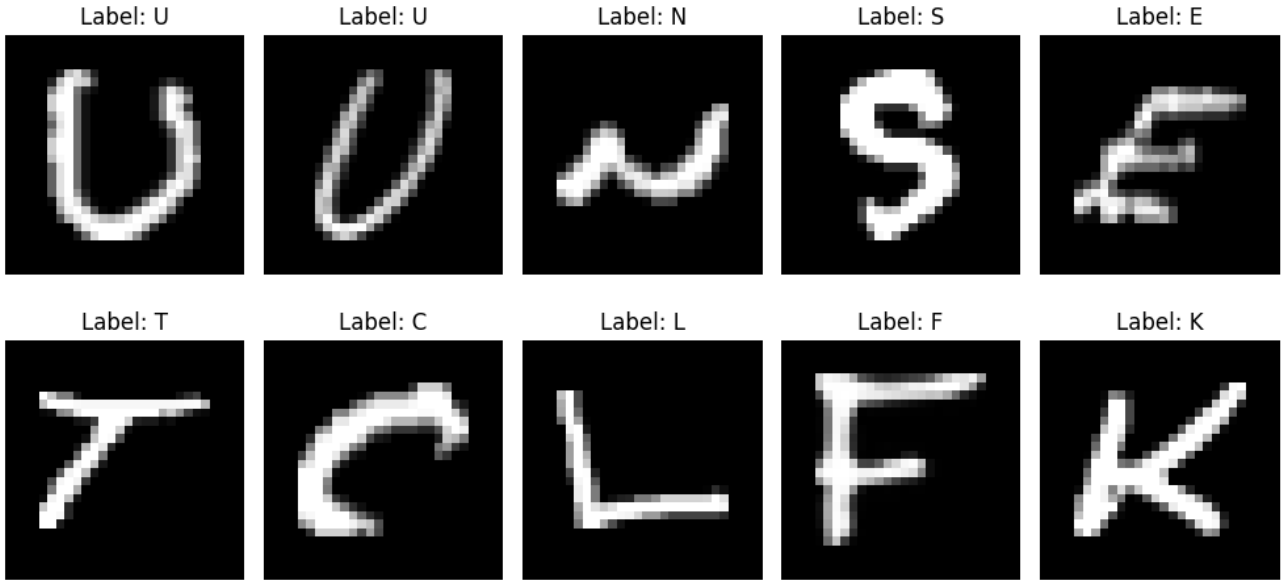
[20]:
```
# 8. Normalization Validation: Verify that the images are properly normalized
print(f"Minimum value in normalized test data: {np.min(X_test)}")
print(f"Maximum value in normalized test data: {np.max(X_test)}")
```

```
Minimum value in normalized test data: 0.0

Maximum value in normalized test data: 1.0
```

9- Reshape the flattened vectors to reconstruct and display the corresponding images while testing the models.

[8]:
```
X_train = np.array(X_train)
X_test = np.array(X_test)

# 5. Reshape the flattened vectors back to 28x28 images
X_train_reshaped = X_train.reshape(-1, 28, 28)  # Reshape to original image dimensions
X_test_reshaped = X_test.reshape(-1, 28, 28)
```

[22]:
```
# Function to plot images with their labels
def plot_images(images, labels, num_images=10):
    plt.figure(figsize=(10, 5))
    for i in range(num_images):
        plt.subplot(2, 5, i + 1)  # Create a grid of 2 rows and 5 columns
        plt.imshow(images[i], cmap='gray')  # Display image
        plt.title(f"Label: {labels[i]}")  # Display corresponding label
        plt.axis('off')  # Hide axes
    plt.tight_layout()
    plt.show()


labels_alphabet = [chr(label + 65) for label in y_test]  # Convert encoded labels to A-Z letters

# Plot the first 10 images with their labels
plot_images(X_test_reshaped, labels_alphabet, num_images=10)
```

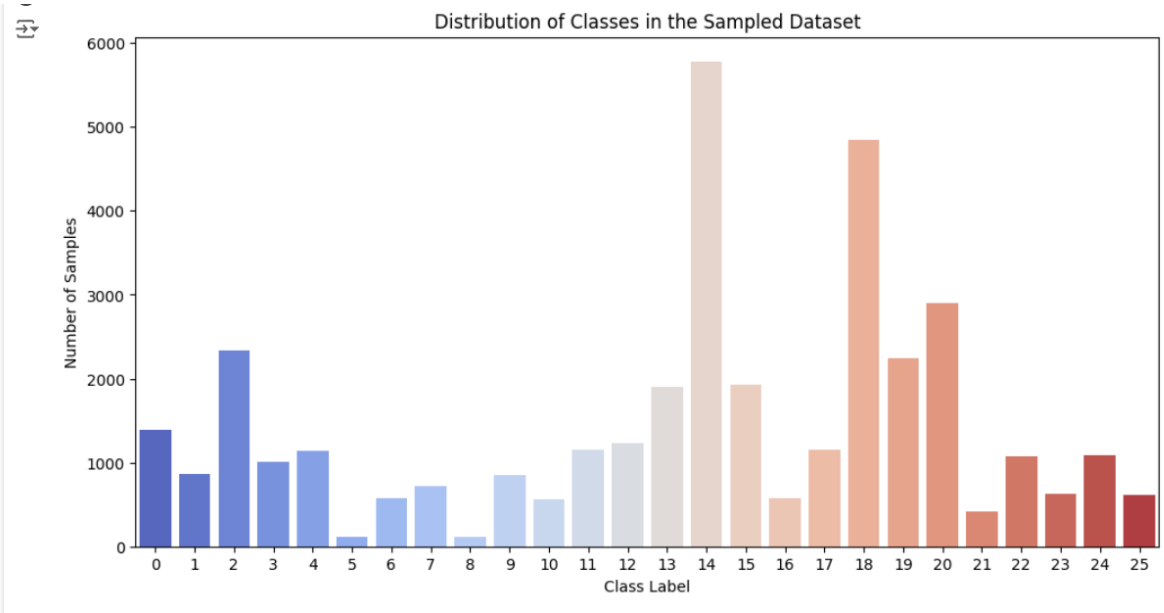


SVM Linear & nonlinear

- Sampled 10% of each class in Dataset (to lessen training time)

```
# Sample 10% of data from each class
portion = 0.10
data_sampled = data.groupby('label', group_keys=False).apply(
    lambda x: x.sample(frac=portion, random_state=42)
).reset_index(drop=True)

# Verify the distribution of classes in the sampled data
sampled_class_distribution = data_sampled['label'].value_counts().sort_index()
print('\nSampled dataset class distribution:')
print(sampled_class_distribution)

plt.figure(figsize=(12,6))
sns.barplot(x=sampled_class_distribution.index, y=sampled_class_distribution.values, palette='coolwarm')
plt.title('Distribution of Classes in the Sampled Dataset')
plt.xlabel('Class Label')
plt.ylabel('Number of Samples')
plt.show()

# Separate features and labels
X = data_sampled.drop('label', axis=1).values
y = data_sampled['label'].values

X_normalized = X / 255.0
```

```
Sampled dataset class distribution:
label
0     1387
1      867
2     2341
3     1013
4     1144
5      116
6      576
7      722
8      112
9      849
10     560
11    1159
12    1234
13    1901
14    5782
15    1934
16     581
17    1157
18    4842
19    2250
20    2901
21     418
22    1078
23     627
24    1086
25     608
Name: count, dtype: int64
```



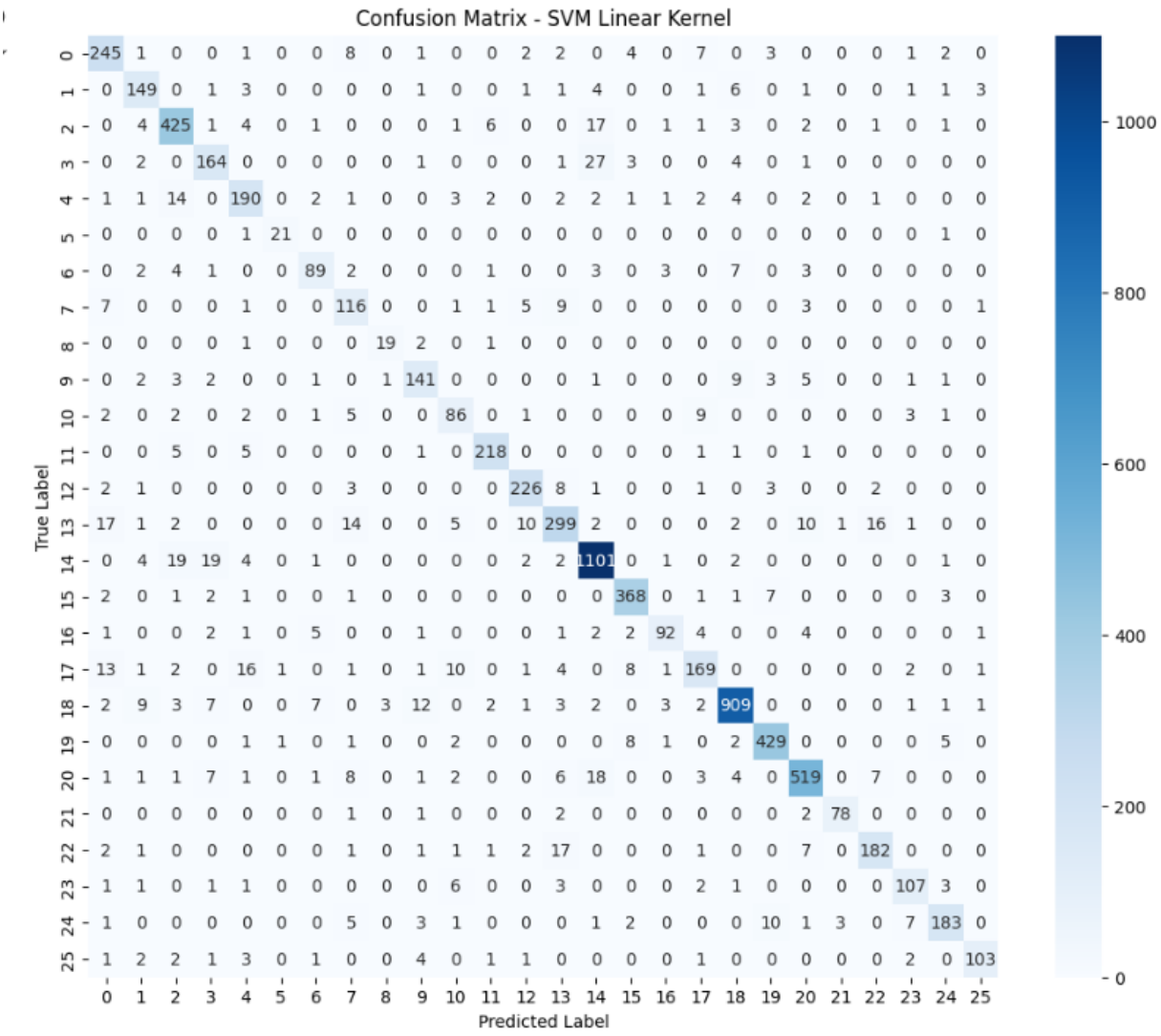
- Linear Kernel accuracy, f1 score, classification report and confusion matrix

```
# Train SVM models
print('\nTraining SVM with Linear Kernel...')
svm_linear = SVC(kernel='linear', random_state=42)
svm_linear.fit(X_train, y_train)

print('Training SVM with RBF Kernel...')
svm_rbf = SVC(kernel='rbf', random_state=42)
svm_rbf.fit(X_train, y_train)

# Test and evaluate the linear SVM
print('\nEvaluating SVM with Linear Kernel...')
y_pred_linear = svm_linear.predict(X_test)

# Compute Accuracy
accuracy_linear = accuracy_score(y_test, y_pred_linear)
print('Accuracy (Linear Kernel): {:.2f}%'.format(accuracy_linear * 100))

# Compute F1 Score
f1_linear = f1_score(y_test, y_pred_linear, average='weighted')
print('Average F1 Score (Linear Kernel): {:.2f}'.format(f1_linear))

# Classification Report
report_linear = classification_report(y_test, y_pred_linear)
print('\nClassification Report for SVM with Linear Kernel:\n')
print(report_linear)
```

```
# Confusion Matrix
conf_matrix_linear = confusion_matrix(y_test, y_pred_linear)

# Visualize Confusion Matrix
plt.figure(figsize=(12,10))
sns.heatmap(conf_matrix_linear, annot=True, fmt='d', cmap='Blues')
plt.title('Confusion Matrix - SVM Linear Kernel')
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.show()
```

```
Training SVM with Linear Kernel...
Training SVM with RBF Kernel...

Evaluating SVM with Linear Kernel...
Accuracy (Linear Kernel): 88.98%
Average F1 Score (Linear Kernel): 0.89

Classification Report for SVM with Linear Kernel:
```

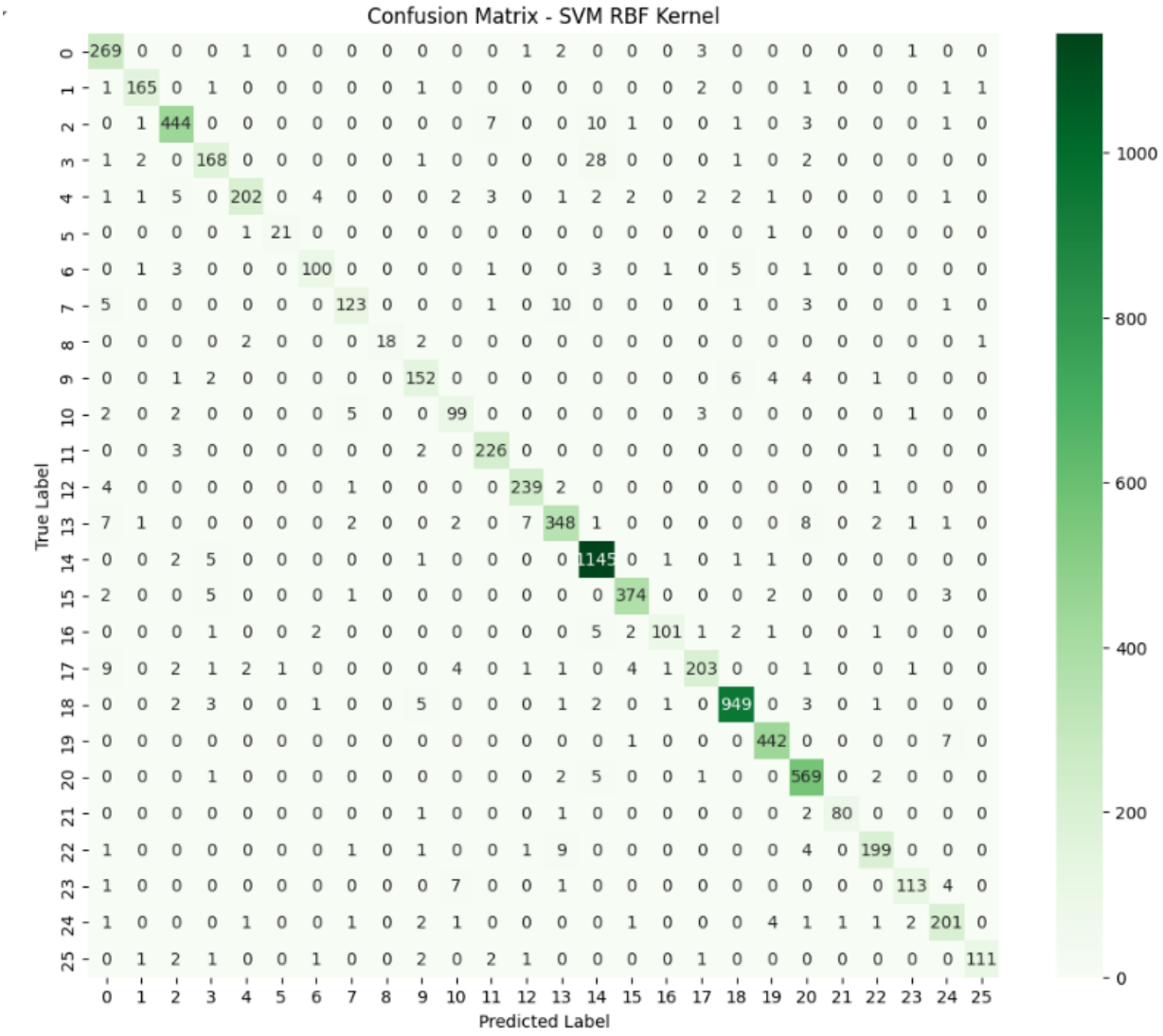
	precision	recall	f1-score	support
0	0.82	0.88	0.85	277
1	0.82	0.86	0.84	173
2	0.88	0.91	0.89	468
3	0.79	0.81	0.80	203
4	0.81	0.83	0.82	229
5	0.91	0.91	0.91	23
6	0.82	0.77	0.79	115
7	0.69	0.81	0.75	144
8	0.83	0.83	0.83	23
9	0.82	0.83	0.83	170
10	0.73	0.77	0.75	112
11	0.94	0.94	0.94	232
12	0.90	0.91	0.91	247
13	0.83	0.79	0.81	380
14	0.93	0.95	0.94	1156
15	0.93	0.95	0.94	387
16	0.89	0.79	0.84	116
17	0.82	0.73	0.78	231
18	0.95	0.94	0.95	968
19	0.94	0.95	0.95	450
20	0.93	0.89	0.91	580
21	0.95	0.93	0.94	84
22	0.87	0.84	0.86	216
23	0.85	0.85	0.85	126
24	0.90	0.84	0.87	217
25	0.94	0.84	0.89	122
accuracy			0.89	7449
macro avg	0.86	0.86	0.86	7449
weighted avg	0.89	0.89	0.89	7449


Confusion Matrix - SVM Linear Kernel
True Label
Predicted Label
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
1000
800
600
400
200
0

- Non-Linear (RBF) Kernel accuracy, f1 score, classification report and confusion matrix

```
# Test and evaluate the RBF SVM
print('Evaluating SVM with RBF Kernel...')
y_pred_rbf = svm_rbf.predict(X_test)

# Compute Accuracy
accuracy_rbf = accuracy_score(y_test, y_pred_rbf)
print('Accuracy (RBF Kernel): {:.2f}%'.format(accuracy_rbf * 100))

# Compute F1 Score
f1_rbf = f1_score(y_test, y_pred_rbf, average='weighted')
print('Average F1 Score (RBF Kernel): {:.2f}'.format(f1_rbf))

# Classification Report
report_rbf = classification_report(y_test, y_pred_rbf)
print('\nClassification Report for SVM with RBF Kernel:\n')
print(report_rbf)

# Confusion Matrix
conf_matrix_rbf = confusion_matrix(y_test, y_pred_rbf)

# Visualize Confusion Matrix
plt.figure(figsize=(12,10))
sns.heatmap(conf_matrix_rbf, annot=True, fmt='d', cmap='Greens')
plt.title('Confusion Matrix - SVM RBF Kernel')
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.show()
```

```
Evaluating SVM with RBF Kernel...
Accuracy (RBF Kernel): 94.79%
Average F1 Score (RBF Kernel): 0.95

Classification Report for SVM with RBF Kernel:

              precision    recall  f1-score   support

           0       0.88      0.97      0.93       277
           1       0.96      0.95      0.96       173
           2       0.95      0.95      0.95       468
           3       0.89      0.83      0.86       203
           4       0.97      0.88      0.92       229
           5       0.95      0.91      0.93        23
           6       0.93      0.87      0.90       115
           7       0.92      0.85      0.88       144
           8       1.00      0.78      0.88        23
           9       0.89      0.89      0.89       170
          10       0.86      0.88      0.87       112
          11       0.94      0.97      0.96       232
          12       0.96      0.97      0.96       247
          13       0.92      0.92      0.92       380
          14       0.95      0.99      0.97      1156
          15       0.97      0.97      0.97       387
          16       0.96      0.87      0.91       116
          17       0.94      0.88      0.91       231
          18       0.98      0.98      0.98       968
          19       0.97      0.98      0.98       450
          20       0.95      0.98      0.96       580
          21       0.99      0.95      0.97        84
          22       0.95      0.92      0.94       216
          23       0.95      0.90      0.92       126
          24       0.91      0.93      0.92       217
          25       0.98      0.91      0.94       122

    accuracy                           0.95      7449
   macro avg       0.94      0.92      0.93      7449
weighted avg       0.95      0.95      0.95      7449
```


Confusion Matrix - SVM RBF Kernel
True Label
Predicted Label
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
1000
800
600
400
200
0

Second experiment (Build from scratch) Logistic Regression:

Split the training dataset into training and validation datasets

Experimnet 2 -- Logistic Regression From Scratch

```
from sklearn.model_selection import train_test_split

X_train_split, X_val, y_train_split, y_val = train_test_split(X_train, y_train, test_size=0.2, random_state=42)

print("Training data shape:", X_train_split.shape)
print("Validation data shape:", X_val.shape)
print(f"Test set shape: {X_test.shape}, {y_test.shape}")
```

Python

```
Training data shape: (238368, 784)
Validation data shape: (59592, 784)
Test set shape: (74490, 784), (74490,)
```

Implement logistic regression for one-versus-all multi-class classification.

+ Code + Markdown

Sigmoid Function Explanation

The sigmoid function maps any input (z) to a value between (0) and (1). It is used to model the probability that a given input belongs to a particular class. The formula is:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

Where:

- (z) is a linear combination of inputs and weights:

$$z = X \cdot \theta$$

- (e) is the base of the natural logarithm.

```
def sigmoid(z):
    return 1 / (1 + np.exp(-z))
```

Cost Function Explanation

The cost function for logistic regression measures how well the model's predictions match the true labels.

The formula is:

$$J(\theta) = -\frac{1}{m} \sum_{i=1}^{m} \left[y^{(i)} \log(h_\theta(x^{(i)})) + (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)})) \right]$$

where

- $h_\theta(x) = \sigma(\theta^T x)$ is the predicted probability.
- m is the number of training examples.
- $y^{(i)}$ is the true label (0 or 1) for the (i)-th example.
- $x^{(i)}$ is the input features for the (i)-th example.

The cost penalizes incorrect predictions, particularly those with high confidence.

```
def compute_cost(X, y, theta):
    m = len(y)  # No.examples
    h = sigmoid(np.dot(X, theta))  # Predicted probabilities
    cost = (-1 / m) * (np.dot(y.T, np.log(h)) + np.dot((1 - y).T, np.log(1 - h)))
    return cost
```

Gradient Descent

Gradient descent is an optimization algorithm used to minimize the cost function by iteratively updating the parameters θ.

At each iteration:

$$theta_j := \theta_j - \alpha \frac{\partial J(\theta)}{\partial \theta_j}$$

Where:

- $alpha$ is the learning rate, controlling the step size.
- $\frac{\partial J(\theta)}{\partial \theta_j}$ is the gradient of the cost function with respect to θ_j.

The gradient is given by:

$$frac\partial J(\theta)\partial \theta_j = \frac{1}{m} \sum_{i=1}^{m} \left[h_\theta(x^{(i)}) - y^{(i)} \right] x_j^{(i)}$$

The algorithm stops after a fixed number of iterations $\mathrm{num\_iters}$ or when the cost converges.

```python
def gradient_descent(X, y, theta, alpha, num_iters):
    m = len(y)  # No.examples
    cost_history = []

    for _ in range(num_iters):
        h = sigmoid(np.dot(X, theta))

        gradient = np.dot(X.T, (h - y)) / m

        theta -= alpha * gradient

        cost_history.append(compute_cost(X, y, theta))

    return theta, cost_history
```

Prediction Explanation

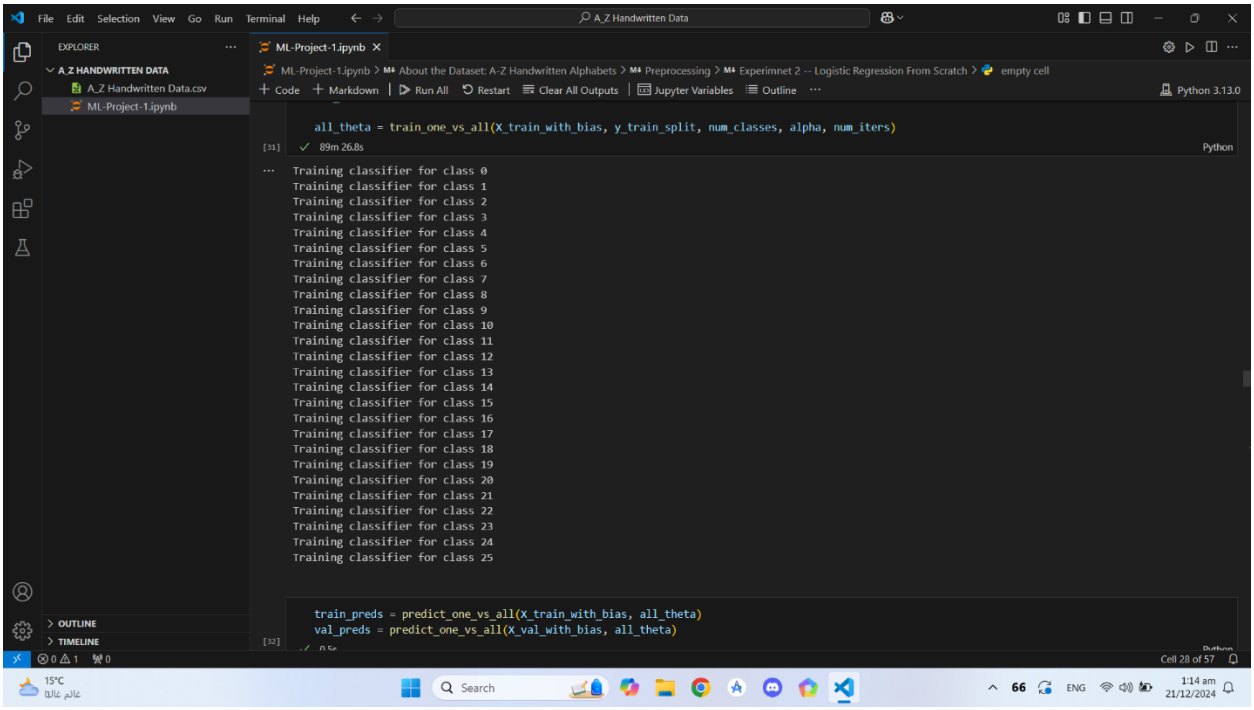
For each input (X), we compute the probability of belonging to each class using the sigmoid function and the learned parameters $\theta $. The class with the highest probability is chosen as the prediction:

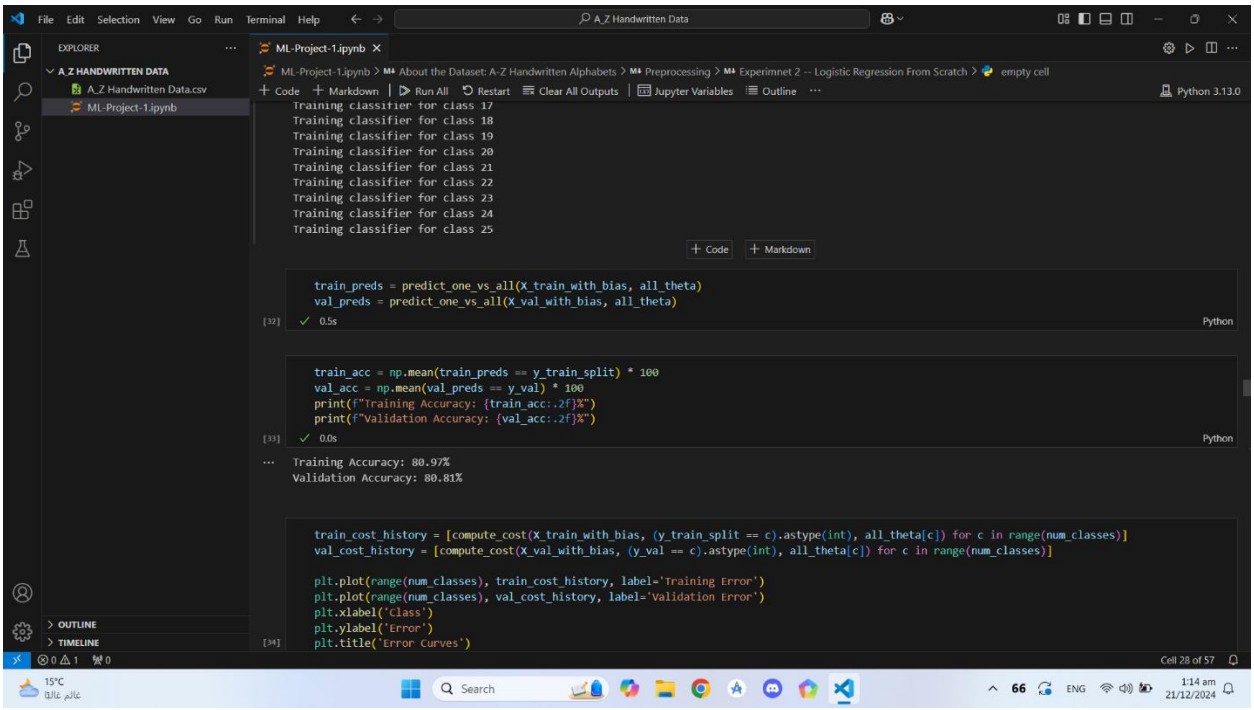
$$\text{Prediction} = \arg\max_c \sigma(\theta_c^T X)$$

Where (c) is the class index, and θ_c is the parameter vector for class (c).

```python
def train_one_vs_all(X, y, num_classes, alpha=0.01, num_iters=1000):
    m, n = X.shape  # m = examples, n = features
    all_theta = np.zeros((num_classes, n))

    for c in range(num_classes):
        print(f"Training classifier for class {c}")
        binary_y = (y == c).astype(int)  # Convert to binary classification for class c
        theta = np.zeros(n)
        theta, _ = gradient_descent(X, binary_y, theta, alpha, num_iters)
        all_theta[c] = theta

    return all_theta
```

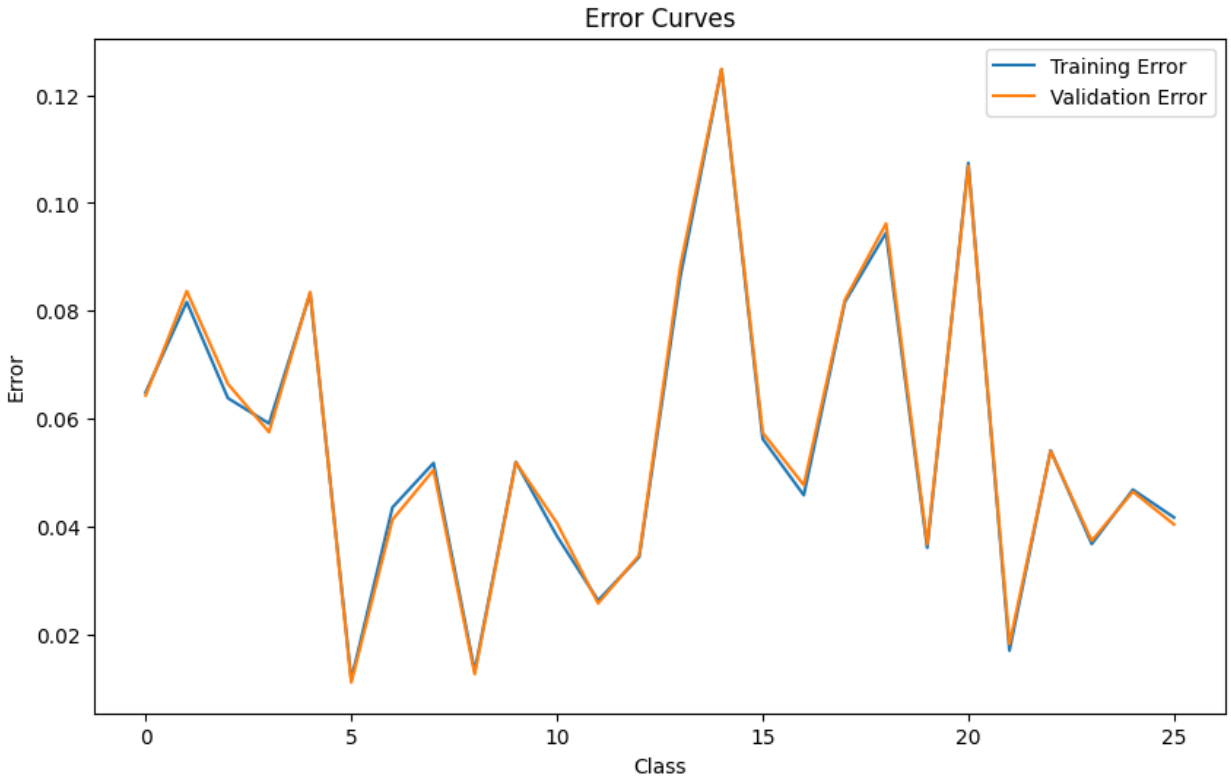
With Learning Rate 0.1





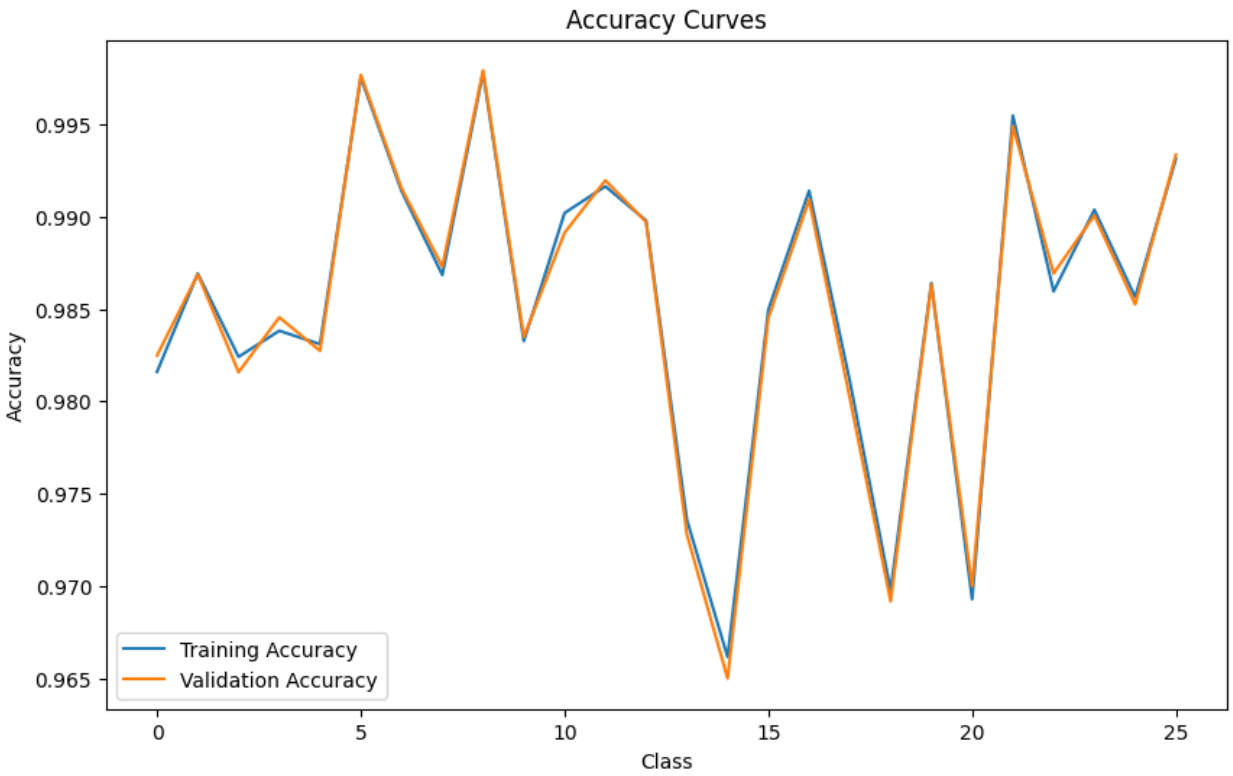
Error Curve

```
train_cost_history = [compute_cost(X_train_with_bias, (y_train_split == c).astype(int), all_theta[c]) for c in range(num_classes)]
val_cost_history = [compute_cost(X_val_with_bias, (y_val == c).astype(int), all_theta[c]) for c in range(num_classes)]

plt.plot(range(num_classes), train_cost_history, label='Training Error')
plt.plot(range(num_classes), val_cost_history, label='Validation Error')
plt.xlabel('Class')
plt.ylabel('Error')
plt.title('Error Curves')
plt.legend()
plt.show()
```



Accuracy Curve

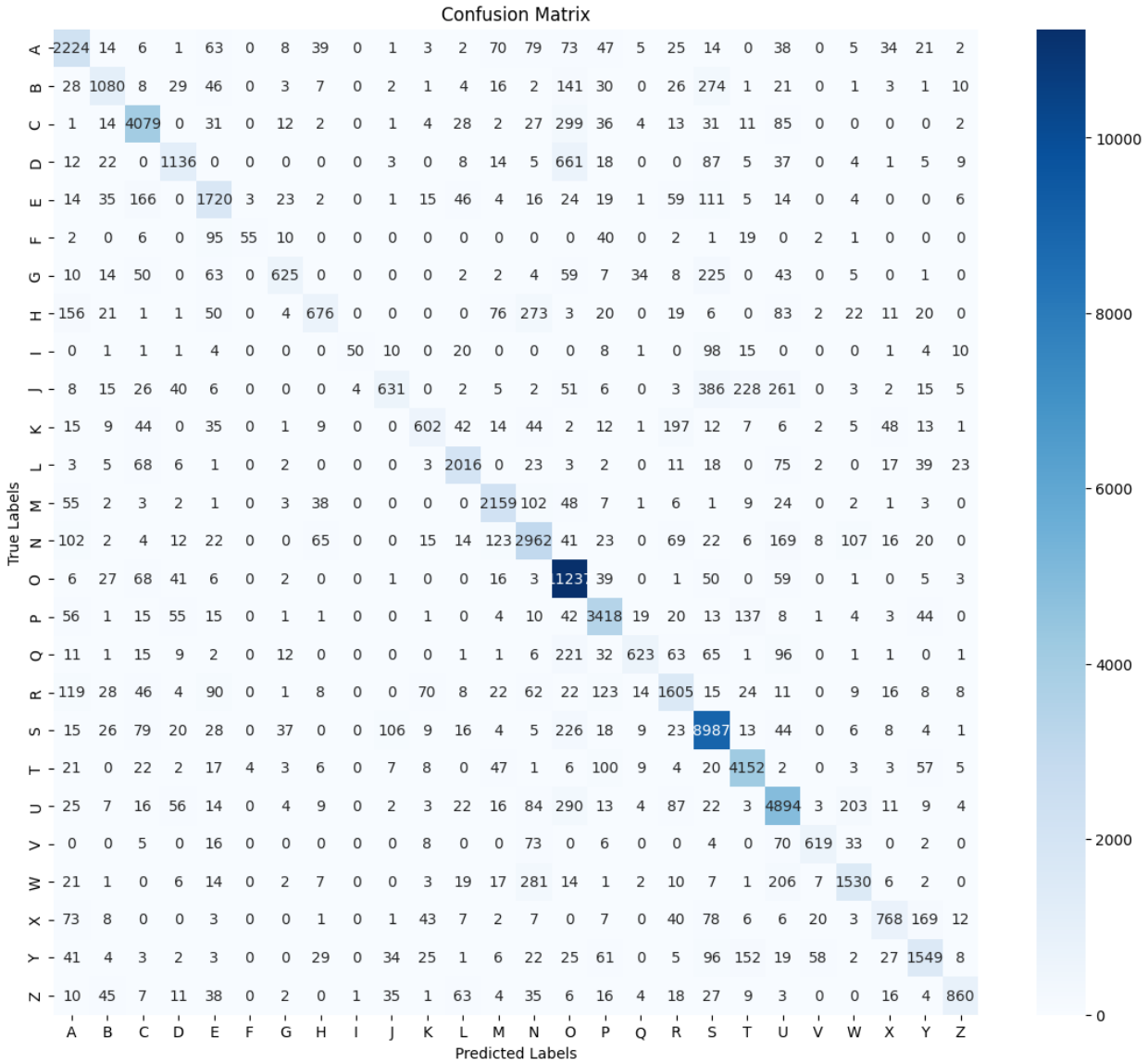
```
from sklearn.metrics import accuracy_score

train_accuracy = [accuracy_score((y_train_split == c).astype(int), predict_one_vs_all(X_train_with_bias, all_theta) == c) for c in range(num_classes)]
val_accuracy = [accuracy_score((y_val == c).astype(int), predict_one_vs_all(X_val_with_bias, all_theta) == c) for c in range(num_classes)]

plt.plot(range(num_classes), train_accuracy, label='Training Accuracy')
plt.plot(range(num_classes), val_accuracy, label='Validation Accuracy')
plt.xlabel('Class')
plt.ylabel('Accuracy')
plt.title('Accuracy Curves')
plt.legend()
plt.show()
```



Confusion Matrix , Classification Report

```python
def plot_confusion_matrix(y_true, y_pred, target_names):
    cm = confusion_matrix(y_true, y_pred)
    plt.figure(figsize=(14, 12))
    sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=target_names, yticklabels=target_names)
    plt.title('Confusion Matrix')
    plt.xlabel('Predicted Labels')
    plt.ylabel('True Labels')
    plt.show()
```

```python
from sklearn.metrics import classification_report, confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np

# Print Classification Report
print(classification_report(y_test, test_preds, target_names=[chr(i) for i in range(65, 91)]))

# Plot Confusion Matrix
conf_matrix = confusion_matrix(y_test, test_preds)
plt.figure(figsize=(14, 12))
sns.heatmap(conf_matrix, annot=True, fmt="d", cmap="Blues",
            xticklabels=[chr(i) for i in range(65, 91)],
            yticklabels=[chr(i) for i in range(65, 91)])
plt.title('Confusion Matrix')
plt.xlabel('Predicted Labels')
plt.ylabel('True Labels')
plt.show()
```


Confusion Matrix
True Labels
Predicted Labels
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
10000
8000
6000
4000
2000
0

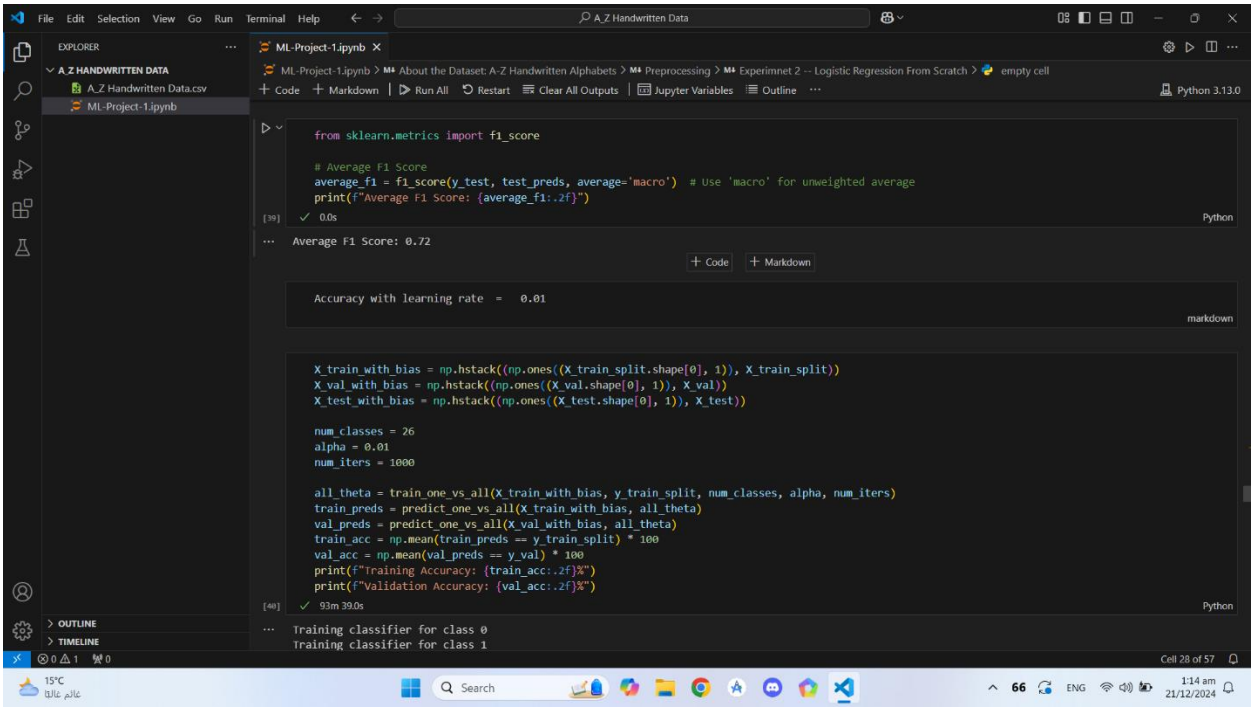
	precision	recall	f1-score	support
A	0.73	0.80	0.77	2774
B	0.78	0.62	0.69	1734
C	0.86	0.87	0.87	4682
D	0.79	0.56	0.66	2027
E	0.72	0.75	0.74	2288
F	0.89	0.24	0.37	233
G	0.83	0.54	0.66	1152
H	0.75	0.47	0.58	1444
I	0.91	0.22	0.36	224
J	0.76	0.37	0.50	1699
K	0.74	0.54	0.62	1121
L	0.87	0.87	0.87	2317
M	0.82	0.88	0.85	2467
N	0.72	0.78	0.75	3802
O	0.83	0.97	0.90	11565
P	0.83	0.88	0.86	3868
Q	0.85	0.54	0.66	1162
R	0.69	0.69	0.69	2313
S	0.84	0.93	0.88	9684
T	0.86	0.92	0.89	4499
U	0.78	0.84	0.81	5801
V	0.85	0.74	0.79	836
W	0.78	0.71	0.74	2157
X	0.77	0.61	0.68	1254
Y	0.78	0.71	0.74	2172
Z	0.89	0.71	0.79	1215

F1_score

```
from sklearn.metrics import f1_score

# Average F1 Score
average_f1 = f1_score(y_test, test_preds, average='macro')  # Use 'macro' for unweighted average
print(f"Average F1 Score: {average_f1:.2f}")
```

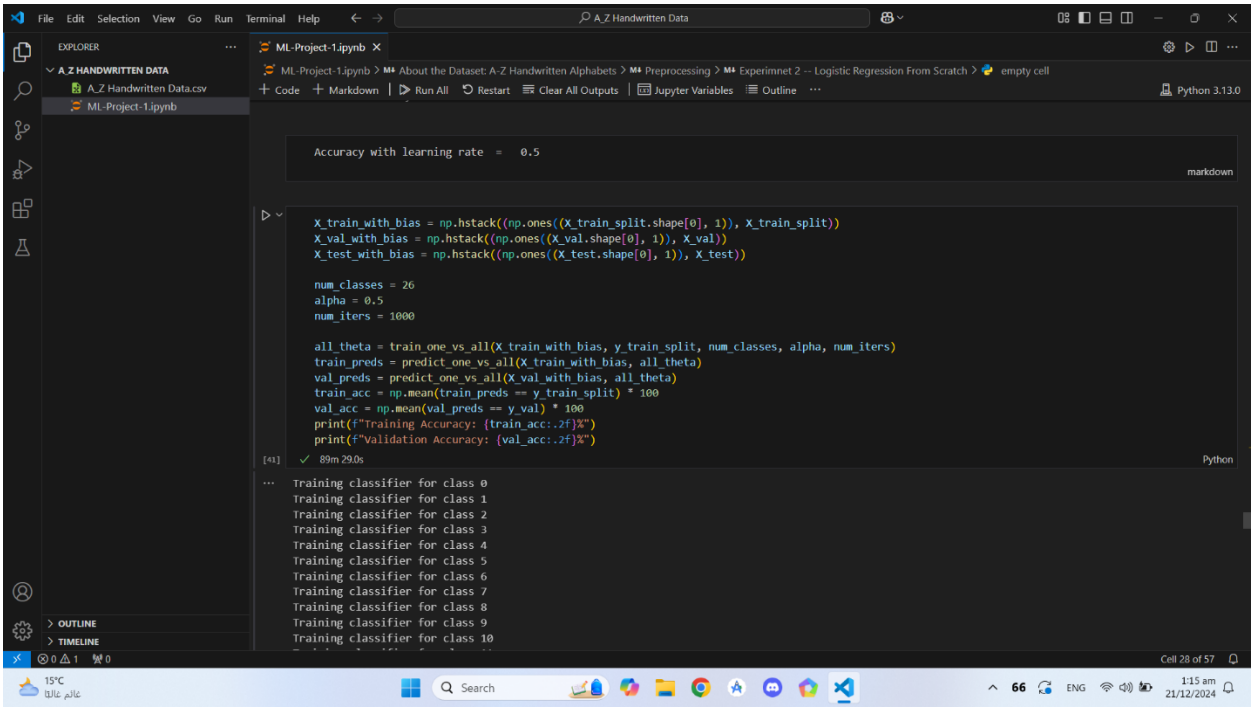
```
Average F1 Score: 0.72
```

Try with learning rate 0.01

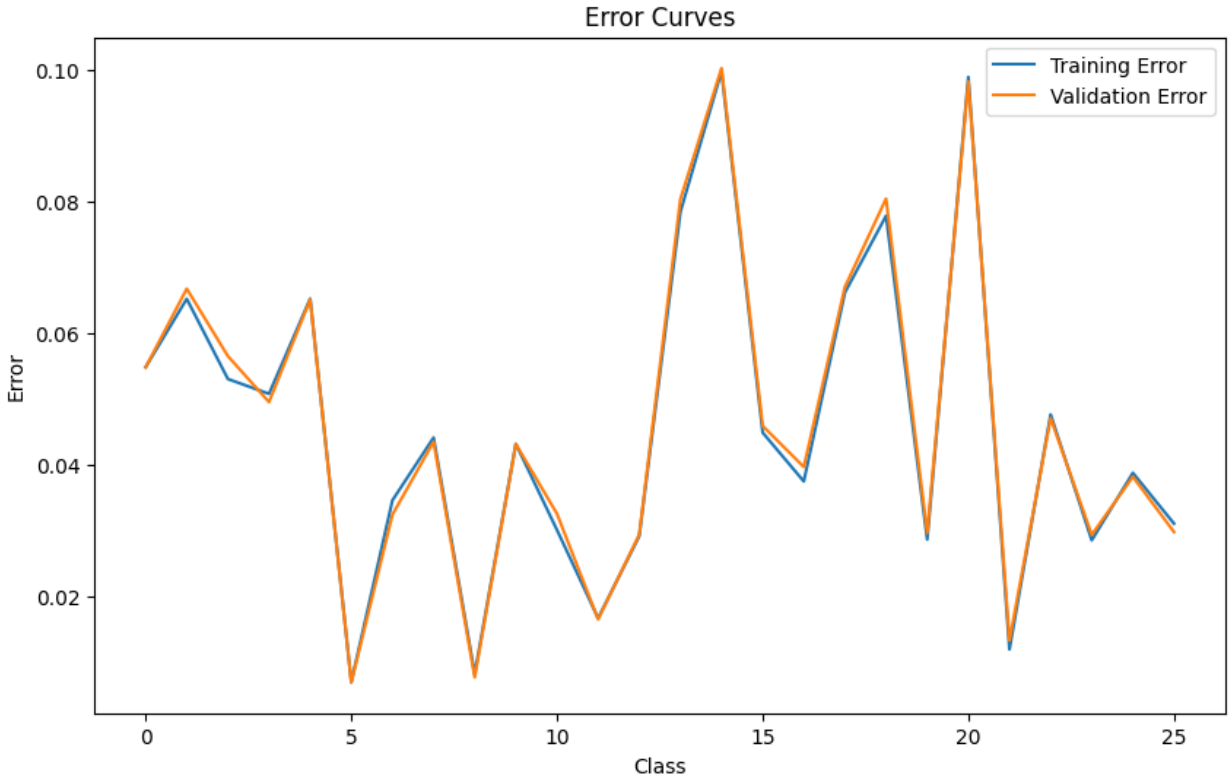


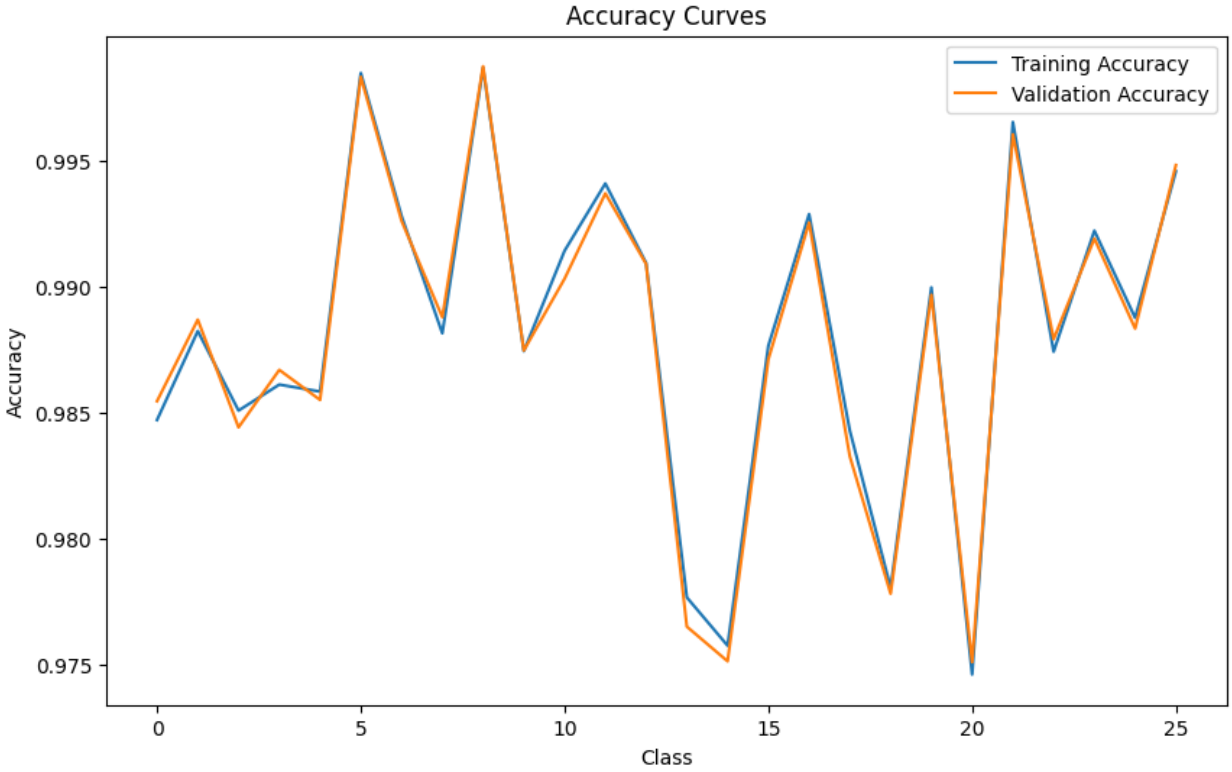

```
Training classifier for class 0
Training classifier for class 1
Training classifier for class 2
Training classifier for class 3
Training classifier for class 4
Training classifier for class 5
Training classifier for class 6
Training classifier for class 7
Training classifier for class 8
Training classifier for class 9
Training classifier for class 10
Training classifier for class 11
Training classifier for class 12
Training classifier for class 13
Training classifier for class 14
Training classifier for class 15
Training classifier for class 16
Training classifier for class 17
Training classifier for class 18
Training classifier for class 19
Training classifier for class 20
Training classifier for class 21
Training classifier for class 22
Training classifier for class 23
Training classifier for class 24
Training classifier for class 25
Training Accuracy: 64.49%
Validation Accuracy: 64.54%
```

Trying with learning rate 0.5 – The Best

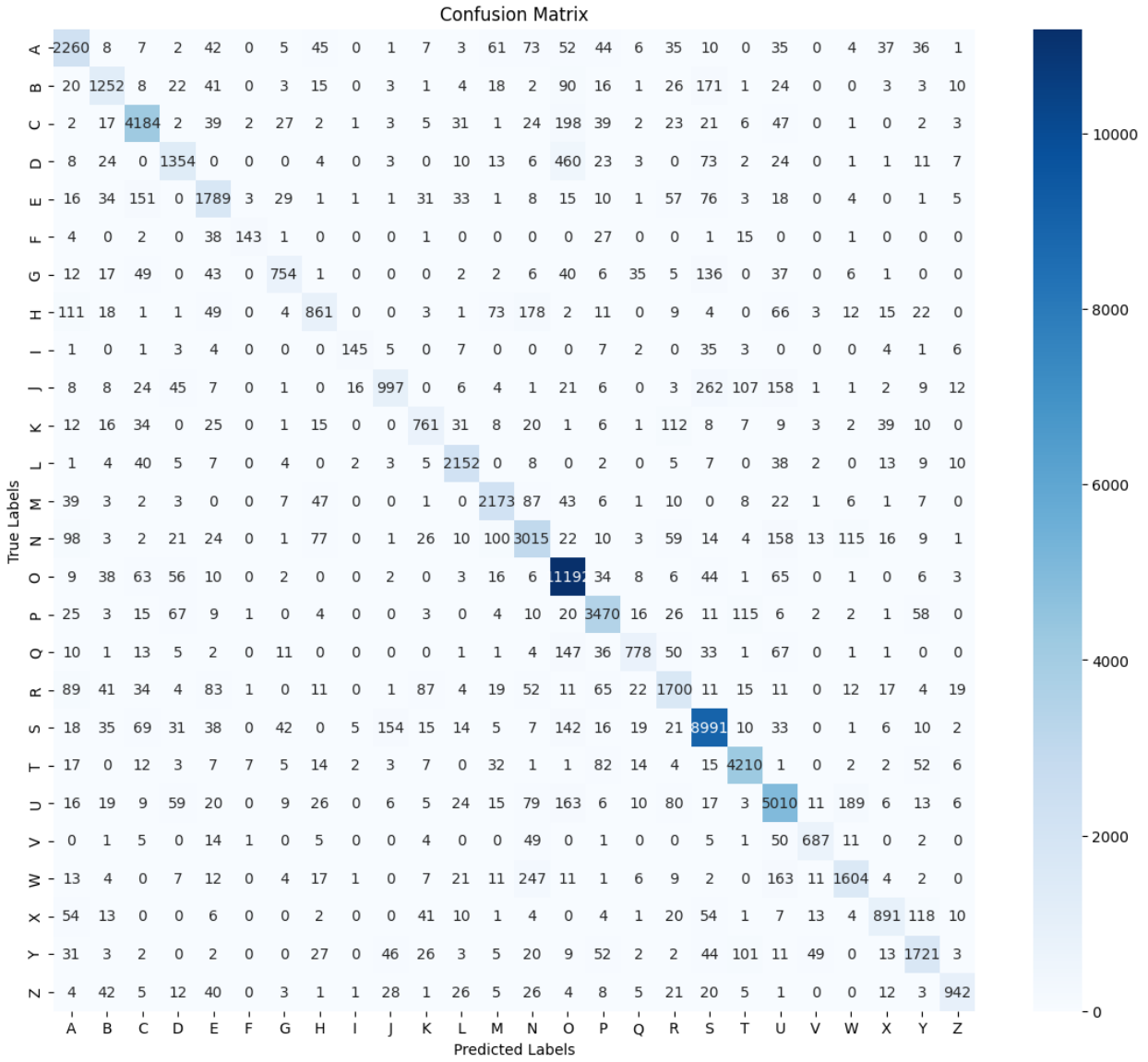



```
Training classifier for class 0
Training classifier for class 1
Training classifier for class 2
Training classifier for class 3
Training classifier for class 4
Training classifier for class 5
Training classifier for class 6
Training classifier for class 7
Training classifier for class 8
Training classifier for class 9
Training classifier for class 10
Training classifier for class 11
Training classifier for class 12
Training classifier for class 13
Training classifier for class 14
Training classifier for class 15
Training classifier for class 16
Training classifier for class 17
Training classifier for class 18
Training classifier for class 19
Training classifier for class 20
Training classifier for class 21
Training classifier for class 22
Training classifier for class 23
Training classifier for class 24
Training classifier for class 25
Training Accuracy: 84.62%
Validation Accuracy: 84.39%
```


Error Curves
Training Error
Validation Error
0.10
0.08
0.06
0.04
0.02
Error
0
5
10
15
20
25
Class


Accuracy Curves
Training Accuracy
Validation Accuracy
0.995
0.990
0.985
0.980
0.975
Accuracy
0
5
10
15
20
25
Class

A	0.79	0.81	0.80	2774
B	0.78	0.72	0.75	1734
C	0.88	0.89	0.89	4682
D	0.80	0.67	0.73	2027
E	0.76	0.78	0.77	2288
F	0.91	0.61	0.73	233
G	0.83	0.65	0.73	1152
H	0.73	0.60	0.66	1444
I	0.83	0.65	0.73	224
J	0.79	0.59	0.67	1699
K	0.73	0.68	0.71	1121
L	0.90	0.93	0.91	2317
M	0.85	0.88	0.86	2467
N	0.77	0.79	0.78	3802
O	0.89	0.97	0.92	11565
P	0.87	0.90	0.88	3868
Q	0.83	0.67	0.74	1162
R	0.74	0.73	0.74	2313
S	0.89	0.93	0.91	9684
T	0.91	0.94	0.92	4499
U	0.83	0.86	0.84	5801
V	0.86	0.82	0.84	836
W	0.81	0.74	0.78	2157
X	0.82	0.71	0.76	1254
Y	0.82	0.79	0.80	2172
Z	0.90	0.78	0.83	1215
accuracy			0.85	74490
macro avg	0.83	0.77	0.80	74490
weighted avg	0.84	0.85	0.84	74490




```
from sklearn.metrics import f1_score

average_f1 = f1_score(y_test, test_preds, average='macro')  # Use 'macro' for unweighted average
print(f"Average F1 Score: {average_f1:.2f}")
```

```
Average F1 Score: 0.80
```

Third experiment (You can use TensorFlow) Neural Networks:

Model 1

A simple neural network with one hidden layer of 128 units and a dropout rate of 0.3 to prevent overfitting. The output layer has 26 neurons (one for each letter).

```python
def create_model_1(input_shape):
    model = Sequential([
        Flatten(input_shape=input_shape),
        Dense(128, activation='relu'),
        Dropout(0.3),
        Dense(26, activation='softmax')  # 26 output classes
    ])
    model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
    return model
```

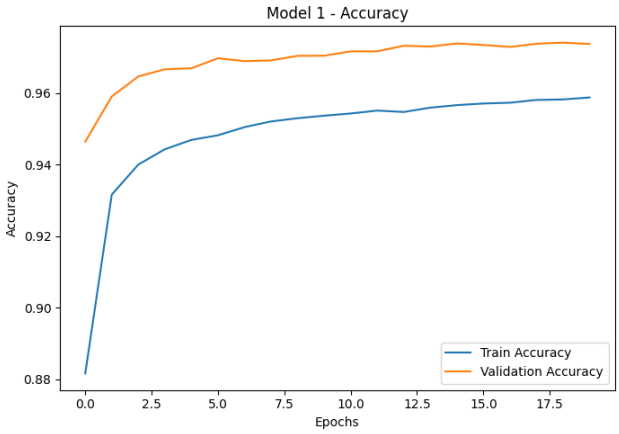
Python

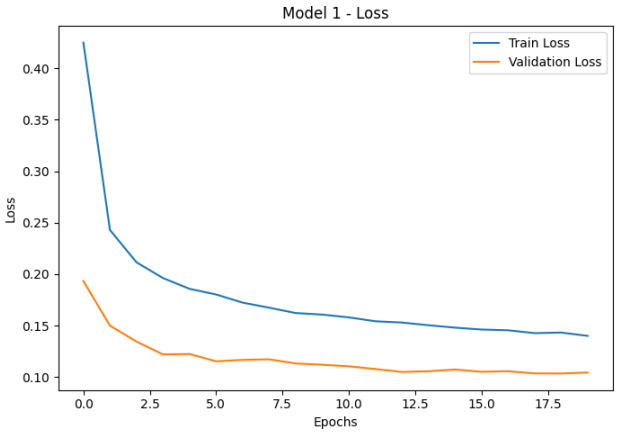
```python
def train_and_plot(model, X_train, y_train, X_val, y_val, model_name):
    history = model.fit(
        X_train, y_train,
        validation_data=(X_val, y_val),
        epochs=20
    )

    # Plot accuracy and loss
    plt.figure(figsize=(14, 5))

    # Plot accuracy
    plt.subplot(1, 2, 1)
    plt.plot(history.history['accuracy'], label='Train Accuracy')
    plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
    plt.title(f'{model_name} - Accuracy')
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy')
    plt.legend()

    # Plot loss
    plt.subplot(1, 2, 2)
    plt.plot(history.history['loss'], label='Train Loss')
    plt.plot(history.history['val_loss'], label='Validation Loss')
    plt.title(f'{model_name} - Loss')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.legend()

    plt.tight_layout()
    plt.show()
```

```python
# Train Model 1
model_1 = create_model_1(input_shape=X_train_split[0].shape)
model_1 = train_and_plot(model_1, X_train_split, y_train_split, X_val, y_val, model_name="Model 1")
```

```
Epoch 1/20
WARNING: All log messages before absl::InitializeLog() is called are written to STDERR
I0000 00:00:1734636356.742054      68 service.cc:145] XLA service 0x7ea960005260 initialized for platform CUDA (this does not guarantee that XLA will be used).
I0000 00:00:1734636356.742108      68 service.cc:153]   StreamExecutor device (0): Tesla P100-PCIE-16GB, Compute Capability 6.0
 120/7449 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.3636 - loss: 2.3619
I0000 00:00:1734636358.659228      68 device_compiler.h:188] Compiled cluster using XLA!  This line is logged at most once for the lifetime of the process.
7449/7449 ━━━━━━━━━━━━━━━━━━━━ 14s 2ms/step - accuracy: 0.8174 - loss: 0.6598 - val_accuracy: 0.9465 - val_loss: 0.1932
Epoch 2/20
7449/7449 ━━━━━━━━━━━━━━━━━━━━ 11s 2ms/step - accuracy: 0.9285 - loss: 0.2539 - val_accuracy: 0.9591 - val_loss: 0.1500
Epoch 3/20
7449/7449 ━━━━━━━━━━━━━━━━━━━━ 11s 1ms/step - accuracy: 0.9392 - loss: 0.2158 - val_accuracy: 0.9647 - val_loss: 0.1345
Epoch 4/20
7449/7449 ━━━━━━━━━━━━━━━━━━━━ 11s 1ms/step - accuracy: 0.9426 - loss: 0.1993 - val_accuracy: 0.9667 - val_loss: 0.1220
Epoch 5/20
7449/7449 ━━━━━━━━━━━━━━━━━━━━ 11s 1ms/step - accuracy: 0.9468 - loss: 0.1866 - val_accuracy: 0.9670 - val_loss: 0.1224
Epoch 6/20
7449/7449 ━━━━━━━━━━━━━━━━━━━━ 11s 1ms/step - accuracy: 0.9484 - loss: 0.1806 - val_accuracy: 0.9698 - val_loss: 0.1153
Epoch 7/20
7449/7449 ━━━━━━━━━━━━━━━━━━━━ 11s 1ms/step - accuracy: 0.9504 - loss: 0.1718 - val_accuracy: 0.9690 - val_loss: 0.1167
Epoch 8/20
7449/7449 ━━━━━━━━━━━━━━━━━━━━ 11s 2ms/step - accuracy: 0.9515 - loss: 0.1676 - val_accuracy: 0.9692 - val_loss: 0.1173
Epoch 9/20
7449/7449 ━━━━━━━━━━━━━━━━━━━━ 11s 1ms/step - accuracy: 0.9535 - loss: 0.1605 - val_accuracy: 0.9704 - val_loss: 0.1132
```

```
Epoch 10/20
7449/7449 ━━━━━━━━━━━━━━━━━━━━ 11s 1ms/step - accuracy: 0.9535 - loss: 0.1605 - val_accuracy: 0.9705 - val_loss: 0.1120
Epoch 11/20
7449/7449 ━━━━━━━━━━━━━━━━━━━━ 11s 2ms/step - accuracy: 0.9543 - loss: 0.1576 - val_accuracy: 0.9717 - val_loss: 0.1105
Epoch 12/20
7449/7449 ━━━━━━━━━━━━━━━━━━━━ 11s 1ms/step - accuracy: 0.9556 - loss: 0.1533 - val_accuracy: 0.9717 - val_loss: 0.1078
Epoch 13/20
7449/7449 ━━━━━━━━━━━━━━━━━━━━ 11s 1ms/step - accuracy: 0.9550 - loss: 0.1514 - val_accuracy: 0.9733 - val_loss: 0.1050
Epoch 14/20
7449/7449 ━━━━━━━━━━━━━━━━━━━━ 11s 2ms/step - accuracy: 0.9564 - loss: 0.1487 - val_accuracy: 0.9731 - val_loss: 0.1057
Epoch 15/20
7449/7449 ━━━━━━━━━━━━━━━━━━━━ 11s 1ms/step - accuracy: 0.9568 - loss: 0.1494 - val_accuracy: 0.9739 - val_loss: 0.1074
Epoch 16/20
7449/7449 ━━━━━━━━━━━━━━━━━━━━ 11s 1ms/step - accuracy: 0.9576 - loss: 0.1451 - val_accuracy: 0.9735 - val_loss: 0.1052
Epoch 17/20
7449/7449 ━━━━━━━━━━━━━━━━━━━━ 11s 2ms/step - accuracy: 0.9586 - loss: 0.1434 - val_accuracy: 0.9729 - val_loss: 0.1057
Epoch 18/20
7449/7449 ━━━━━━━━━━━━━━━━━━━━ 11s 2ms/step - accuracy: 0.9587 - loss: 0.1414 - val_accuracy: 0.9738 - val_loss: 0.1036
Epoch 19/20
7449/7449 ━━━━━━━━━━━━━━━━━━━━ 11s 1ms/step - accuracy: 0.9577 - loss: 0.1428 - val_accuracy: 0.9741 - val_loss: 0.1035
Epoch 20/20
7449/7449 ━━━━━━━━━━━━━━━━━━━━ 11s 2ms/step - accuracy: 0.9594 - loss: 0.1380 - val_accuracy: 0.9738 - val_loss: 0.1044
```





Model 1 - Training Insights

Overview:

- **Input Layer**: A Flatten layer to preprocess input data.

- **Hidden Layer**: A dense layer with 128 neurons, ReLU activation, and 30% dropout to prevent overfitting.

- **Output Layer**: A dense layer with 26 neurons (one for each class) using softmax activation for classification probabilities.

The model was trained using:

- **Optimizer**: Adam

- **Loss Function**: Categorical Crossentropy

- **Evaluation Metric**: Accuracy

Training and Validation Metrics:

1. **Accuracy**:

  - Training accuracy started at **81.97%** by the end of the first epoch and progressively increased to **95.70%** by epoch 20.

  - Validation accuracy began at **94.63%**in epoch 1 and improved steadily to **97.45%** by epoch 20.

  - The gap between training and validation accuracy remained small, indicating good generalization.

2. **Loss**:

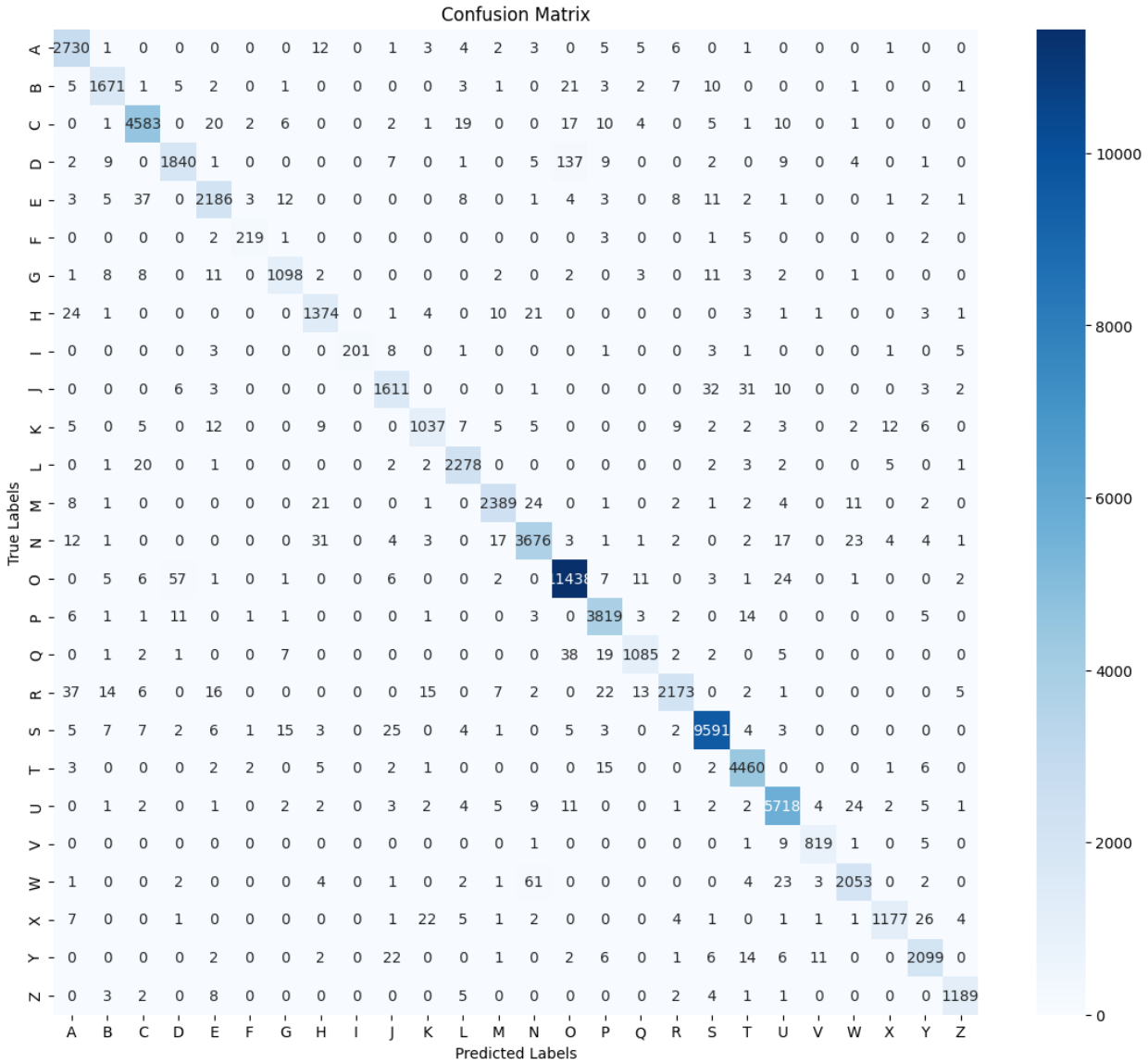
  - Training loss began at **0.6481** in epoch 1 and decreased to **0.1427** by epoch 20.

  - Validation loss started at **0.1937** and plateaued at around **0.1044** after several epochs, showing consistent improvements.

```python
def plot_confusion_matrix(y_true, y_pred, target_names):
    cm = confusion_matrix(y_true, y_pred)
    plt.figure(figsize=(14, 12))
    sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=target_names, yticklabels=target_names)
    plt.title('Confusion Matrix')
    plt.xlabel('Predicted Labels')
    plt.ylabel('True Labels')
    plt.show()
```

```python
from sklearn.metrics import classification_report, confusion_matrix,f1_score
y_test_pred = model_1.predict(X_test)
y_test_pred_classes = np.argmax(y_test_pred, axis=1)
y_test_true_classes = np.argmax(y_test_categorical, axis=1)

# Print Classification Report
print(classification_report(y_test_true_classes, y_test_pred_classes, target_names=[chr(i) for i in range(65, 91)]))

# Plot Confusion Matrix
plot_confusion_matrix(y_test_true_classes, y_test_pred_classes, target_names=[chr(i) for i in range(65, 91)])

# Calculate Average F1 Score
average_f1 = f1_score(y_test_true_classes, y_test_pred_classes, average='macro')
print(f"Average F1 Score: {average_f1:.2f}")
```

```
2328/2328 ━━━━━━━━━━━━━━━━━━━━ 3s 1ms/step
              precision    recall  f1-score   support

           A       0.96      0.98      0.97      2774
           B       0.97      0.96      0.96      1734
           C       0.98      0.98      0.98      4682
           D       0.96      0.91      0.93      2027
           E       0.96      0.96      0.96      2288
           F       0.96      0.94      0.95       233
           G       0.96      0.95      0.96      1152
           H       0.94      0.95      0.94      1444
           I       1.00      0.90      0.95       224
           J       0.95      0.95      0.95      1699
           K       0.95      0.93      0.94      1121
           L       0.97      0.98      0.98      2317
           M       0.98      0.97      0.97      2467
           N       0.96      0.97      0.97      3802
           O       0.98      0.99      0.98     11565
           P       0.97      0.99      0.98      3868
           Q       0.96      0.93      0.95      1162
           R       0.98      0.94      0.96      2313
           S       0.99      0.99      0.99      9684
           T       0.98      0.99      0.98      4499
           U       0.98      0.99      0.98      5801
           V       0.98      0.98      0.98       836
           W       0.97      0.95      0.96      2157
           X       0.98      0.94      0.96      1254
           Y       0.97      0.97      0.97      2172
           Z       0.98      0.98      0.98      1215
```




```
Average F1 Score: 0.96
```

Optimized Neural Network Design

1. Layer Architecture:

- Use multiple hidden layers with different number of neurons per layer.

- Reduce the number of trainable parameters to prevent overfitting.

2. Regularization:

- Apply Dropout to avoid overfitting.

- Add L2 regularization to the dense layers.

3. Activation Functions:

- Use LeakyReLU for hidden layers and Softmax for the output layer.

4. Early Stopping:

- Monitor validation loss to avoid training too long and overfitting.

5. Batch Normalization:

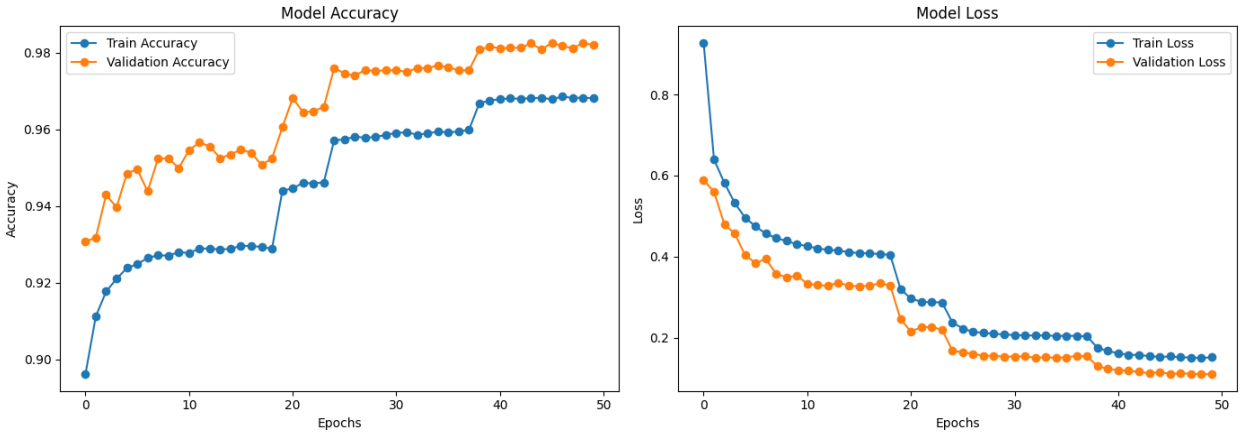
- Add after each dense layer to stabilize learning.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, BatchNormalization, LeakyReLU
from tensorflow.keras.regularizers import l2
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import ReduceLROnPlateau, EarlyStopping

def create_optimized_model():
    model = Sequential([
        Dense(1024, kernel_regularizer=l2(0.001)),
        LeakyReLU(alpha=0.1),
        BatchNormalization(),
        Dropout(0.2),

        Dense(512, kernel_regularizer=l2(0.001)),
        LeakyReLU(alpha=0.1),
        BatchNormalization(),
        Dropout(0.2),

        Dense(256, kernel_regularizer=l2(0.001)),
        LeakyReLU(alpha=0.1),
        BatchNormalization(),
        Dropout(0.2),

        Dense(26, activation='softmax')  # Output for 26 classes
    ])
```

```
    optimizer = Adam(learning_rate=0.001)
    model.compile(
        optimizer=optimizer,
        loss='categorical_crossentropy',
        metrics=['accuracy']
    )
    return model

# Callbacks for training
lr_scheduler = ReduceLROnPlateau(monitor='val_loss', factor=0.5, patience=3)
early_stopping = EarlyStopping(monitor='val_loss', patience=5, restore_best_weights=True)

# Fit model (assuming x_train, y_train, x_val, y_val are defined)
model = create_optimized_model()
history = model.fit(
    X_train_split, y_train_split,
    validation_data=(X_val, y_val),
    epochs=50,
    batch_size=64,
    callbacks=[lr_scheduler, early_stopping]
)
```

```
Epoch 16/50
3725/3725 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - accuracy: 0.9295 - loss: 0.4084 - val_accuracy: 0.9547 - val_loss: 0.3269 - learning_rate: 0.0010
Epoch 17/50
3725/3725 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - accuracy: 0.9296 - loss: 0.4087 - val_accuracy: 0.9539 - val_loss: 0.3280 - learning_rate: 0.0010
Epoch 18/50
3725/3725 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - accuracy: 0.9293 - loss: 0.4055 - val_accuracy: 0.9507 - val_loss: 0.3338 - learning_rate: 0.0010
Epoch 19/50
3725/3725 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - accuracy: 0.9290 - loss: 0.4043 - val_accuracy: 0.9524 - val_loss: 0.3287 - learning_rate: 0.0010
Epoch 20/50
3725/3725 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - accuracy: 0.9432 - loss: 0.3364 - val_accuracy: 0.9605 - val_loss: 0.2450 - learning_rate: 5.0000e-04
Epoch 21/50
3725/3725 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - accuracy: 0.9441 - loss: 0.2990 - val_accuracy: 0.9681 - val_loss: 0.2151 - learning_rate: 5.0000e-04
Epoch 22/50
3725/3725 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - accuracy: 0.9459 - loss: 0.2881 - val_accuracy: 0.9644 - val_loss: 0.2258 - learning_rate: 5.0000e-04
```

```
3725/3725 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - accuracy: 0.9687 - loss: 0.1518 - val_accuracy: 0.9808 - val_loss: 0.1149 - learning_rate: 1.2500e-04
Epoch 46/50
3725/3725 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - accuracy: 0.9677 - loss: 0.1531 - val_accuracy: 0.9825 - val_loss: 0.1108 - learning_rate: 1.2500e-04
Epoch 47/50
3725/3725 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - accuracy: 0.9695 - loss: 0.1505 - val_accuracy: 0.9817 - val_loss: 0.1114 - learning_rate: 1.2500e-04
Epoch 48/50
3725/3725 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - accuracy: 0.9688 - loss: 0.1492 - val_accuracy: 0.9811 - val_loss: 0.1111 - learning_rate: 1.2500e-04
Epoch 49/50
3725/3725 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - accuracy: 0.9680 - loss: 0.1497 - val_accuracy: 0.9824 - val_loss: 0.1093 - learning_rate: 1.2500e-04
Epoch 50/50
3725/3725 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - accuracy: 0.9685 - loss: 0.1505 - val_accuracy: 0.9820 - val_loss: 0.1102 - learning_rate: 1.2500e-04
```



[43]:

```
y_test_pred = best_model.predict(X_test)
y_test_pred_classes = np.argmax(y_test_pred, axis=1)
y_test_true_classes = np.argmax(y_test_categorical, axis=1)

# Print Classification Report
print(classification_report(y_test_true_classes, y_test_pred_classes, target_names=[chr(i) for i in range(65, 91)]))

# Plot Confusion Matrix
plot_confusion_matrix(y_test_true_classes, y_test_pred_classes, target_names=[chr(i) for i in range(65, 91)])

# Calculate Average F1 Score
average_f1 = f1_score(y_test_true_classes, y_test_pred_classes, average='macro')
print(f"Average F1 Score: {average_f1:.2f}")
```

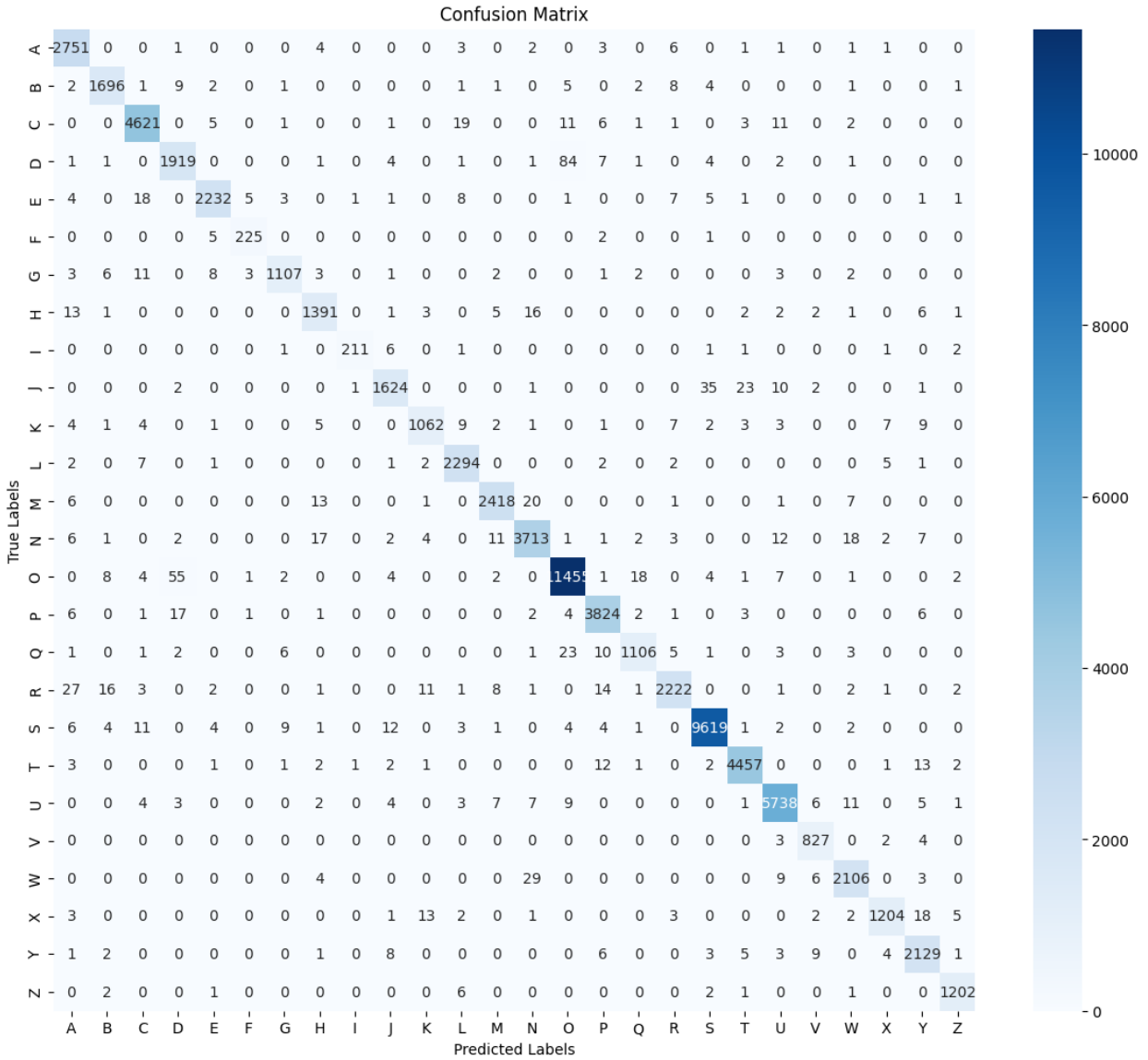
```
2328/2328 ━━━━━━━━━━━━━━━━━━━━ 3s 1ms/step
```

	precision	recall	f1-score	support
A	0.97	0.99	0.98	2774
B	0.98	0.98	0.98	1734
C	0.99	0.99	0.99	4682
D	0.95	0.95	0.95	2027

H	0.96	0.96	0.96	1444
I	0.99	0.94	0.96	224
J	0.97	0.96	0.96	1699
K	0.97	0.95	0.96	1121
L	0.98	0.99	0.98	2317
M	0.98	0.98	0.98	2467
N	0.98	0.98	0.98	3802
O	0.99	0.99	0.99	11565
P	0.98	0.99	0.99	3868
Q	0.97	0.95	0.96	1162
R	0.98	0.96	0.97	2313
S	0.99	0.99	0.99	9684
T	0.99	0.99	0.99	4499
U	0.99	0.99	0.99	5801
V	0.97	0.99	0.98	836
W	0.97	0.98	0.98	2157
X	0.98	0.96	0.97	1254

Predicted Labels

Average F1 Score: 0.98



Save Best Model and reuse it

[41]:
```
model_1.save("First_Try_model.h5")  # Assume Model 2 performed better
First_Try_model = tf.keras.models.load_model("First_Try_model.h5")
```

[42]:
```
model.save("best_model.h5")  # Assume Model 2 performed better
best_model = tf.keras.models.load_model("best_model.h5")
```

Test best Model with images

[44]:
```python
# Function to load images from the given paths
def load_images(image_paths):
    images = []
    for path in image_paths:
        img = Image.open(path).convert('L')  # Convert to grayscale
        img = img.resize((28, 28))  # Resize to match model input size (28x28 for MNIST-like models)
        images.append(np.array(img))  # Convert to numpy array
    return np.array(images)
```

[45]:
```python
# Plot function for images and predictions
def plot_image(index, predictions_array, true_labels, images):
    plt.imshow(images[index], cmap='gray')
    predicted_label = np.argmax(predictions_array)
    true_label = ord(true_labels[index]) - 65  # Convert letter to index (A=0, B=1, ...)
    color = 'blue' if predicted_label == true_label else 'red'
    plt.title(f"True: {true_labels[index]}\nPred: {chr(predicted_label + 65)}", color=color)
    plt.axis('off')
```

[46]:
```python
def plot_value_array(index, predictions_array, true_labels):
    predicted_label = np.argmax(predictions_array)
    true_label = ord(true_labels[index]) - 65
    plt.bar(range(26), predictions_array, color="#777777")
    plt.xticks(range(26), [chr(i + 65) for i in range(26)], rotation=90)
    plt.yticks([])
    plt.ylim([0, 1])
    plt.bar(predicted_label, predictions_array[predicted_label], color='red')
    plt.bar(true_label, predictions_array[true_label], color='blue')
```

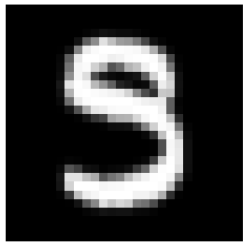
```python
# Load the trained model
model = load_model('best_model.h5')  # Ensure the correct path to the model

# Define the paths to the images to be tested
image_paths = [
    "/kaggle/input/test-images/S_3.png",
    "/kaggle/input/test-images/R_2.png",
    "/kaggle/input/test-images/Y_1.png",
    "/kaggle/input/test-images/Z_2.png"
]

# Load the images
test_images = load_images(image_paths)

# Preprocess the images (normalize and reshape as required by the model)
test_images = test_images.astype('float32') / 255.0  # Normalize pixel values to [0, 1]
test_images_flattened = test_images.reshape(-1, 28 * 28)  # Flatten images for the model

# Get model predictions
predictions = model.predict(test_images_flattened)
```

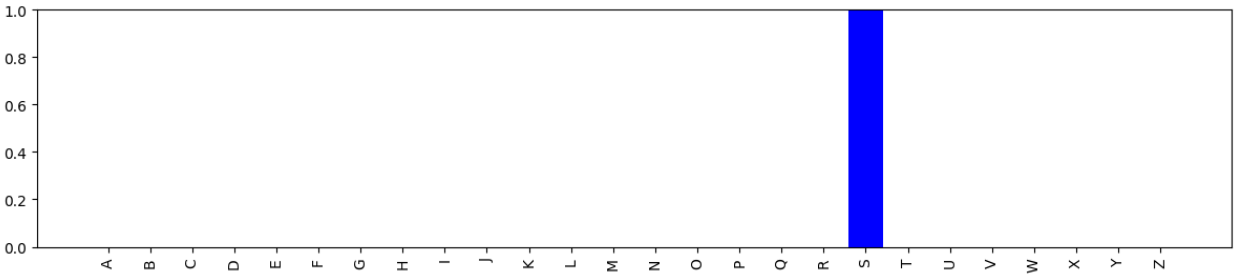
```python
# Define the actual labels for the test images
test_labels = ['S', 'R', 'Y', 'Z']  # Corresponding true labels for the images

# Display predictions and confidence levels for each character
for label in ['S', 'R', 'Y', 'Z']:
    label_indices = [i for i, x in enumerate(test_labels) if x == label]
    num_images = len(label_indices)

    # Create subplots for images and prediction value arrays
    fig, axes = plt.subplots(2, num_images, figsize=(12, 6))

    # Ensure axes is always 2D (even when num_images = 1)
    if num_images == 1:
        axes = np.expand_dims(axes, axis=1)

    for i, index in enumerate(label_indices):
        # Plot the test image
        axes[0, i].imshow(test_images[index], cmap='gray')
        predicted_label = np.argmax(predictions[index])
        true_label = ord(test_labels[index]) - 65  # Convert letter to index (A=0, B=1, ...)
        color = 'blue' if predicted_label == true_label else 'red'
        axes[0, i].set_title(f"True: {test_labels[index]}\nPred: {chr(predicted_label + 65)}", color=color)
        axes[0, i].axis('off')
```

```python
  for i, index in enumerate(label_indices):
      # Plot the test image
      axes[0, i].imshow(test_images[index], cmap='gray')
      predicted_label = np.argmax(predictions[index])
      true_label = ord(test_labels[index]) - 65  # Convert letter to index (A=0, B=1, ...)
      color = 'blue' if predicted_label == true_label else 'red'
      axes[0, i].set_title(f"True: {test_labels[index]}\nPred: {chr(predicted_label + 65)}", color=color)
      axes[0, i].axis('off')

      # Plot the prediction value array
      axes[1, i].bar(range(26), predictions[index], color="#777777")
      axes[1, i].set_xticks(range(26))
      axes[1, i].set_xticklabels([chr(i + 65) for i in range(26)], rotation=90)
      axes[1, i].set_ylim([0, 1])
      axes[1, i].bar(predicted_label, predictions[index][predicted_label], color='red')
      axes[1, i].bar(true_label, predictions[index][true_label], color='blue')

  plt.suptitle(f"Character: {label}", fontsize=16)
  plt.tight_layout()
  plt.subplots_adjust(top=0.85)  # Adjust the title position
  plt.show()
```

Character: S

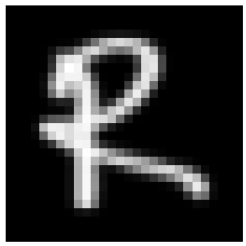
True: S
Pred: S

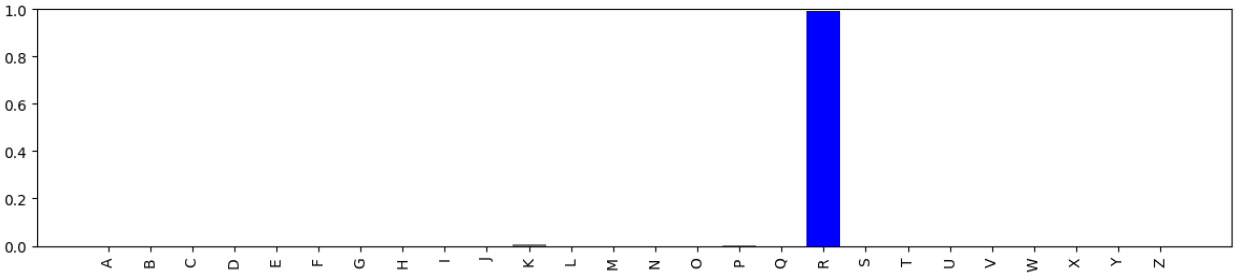


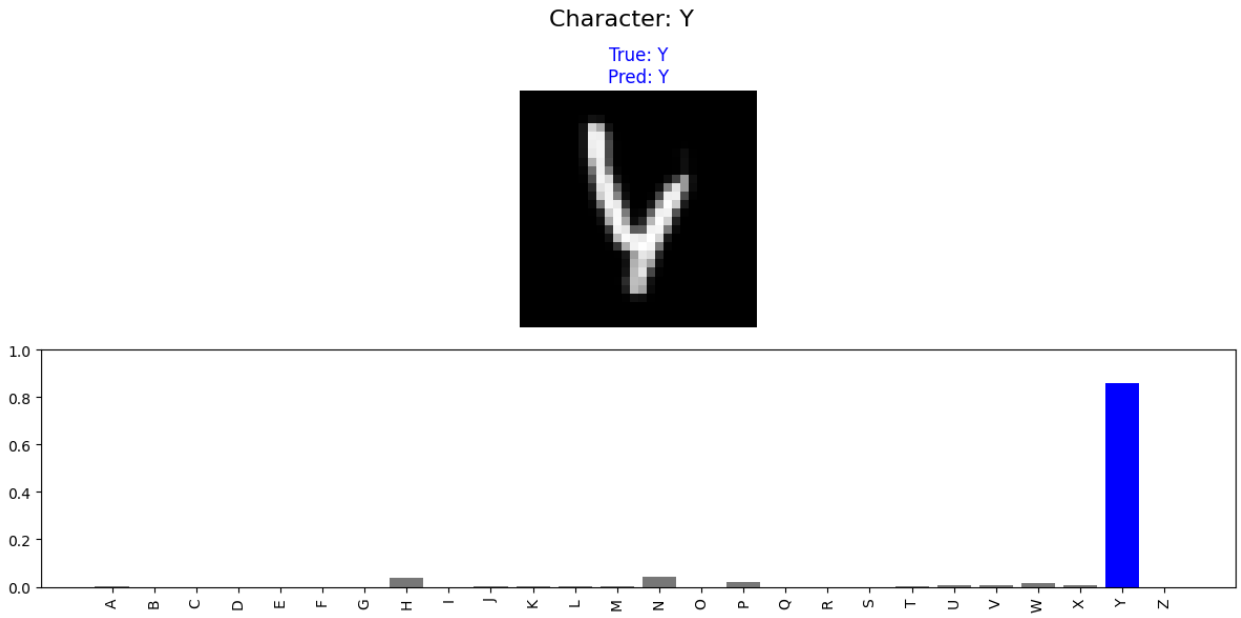


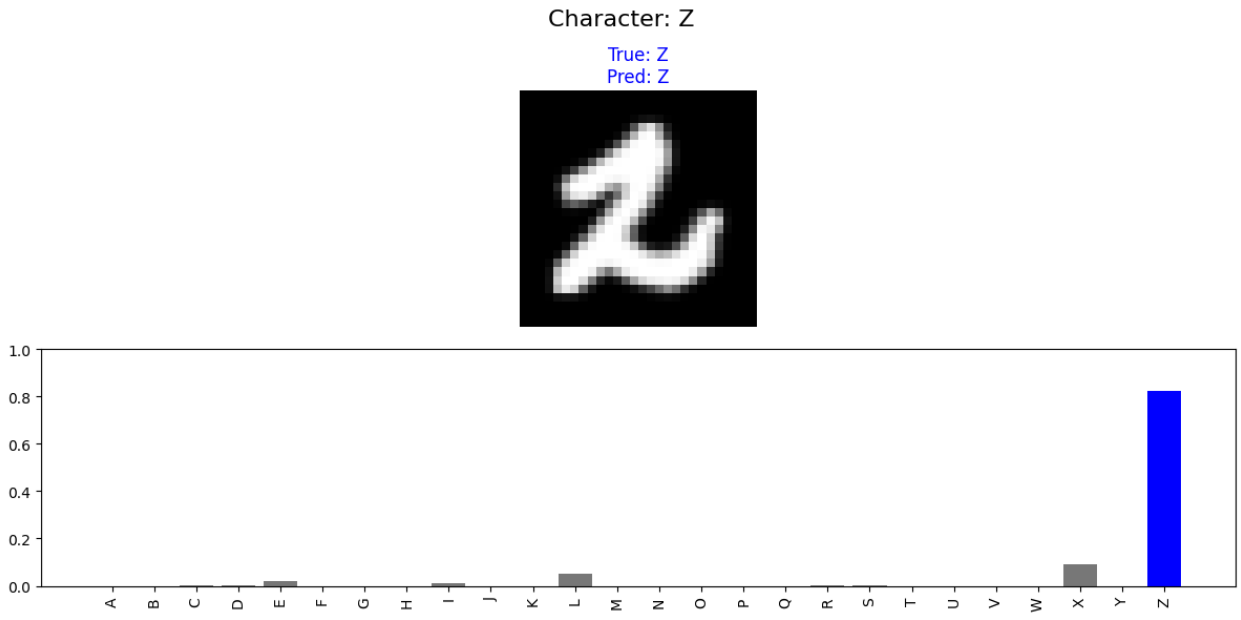
Character: R

True: R
Pred: R



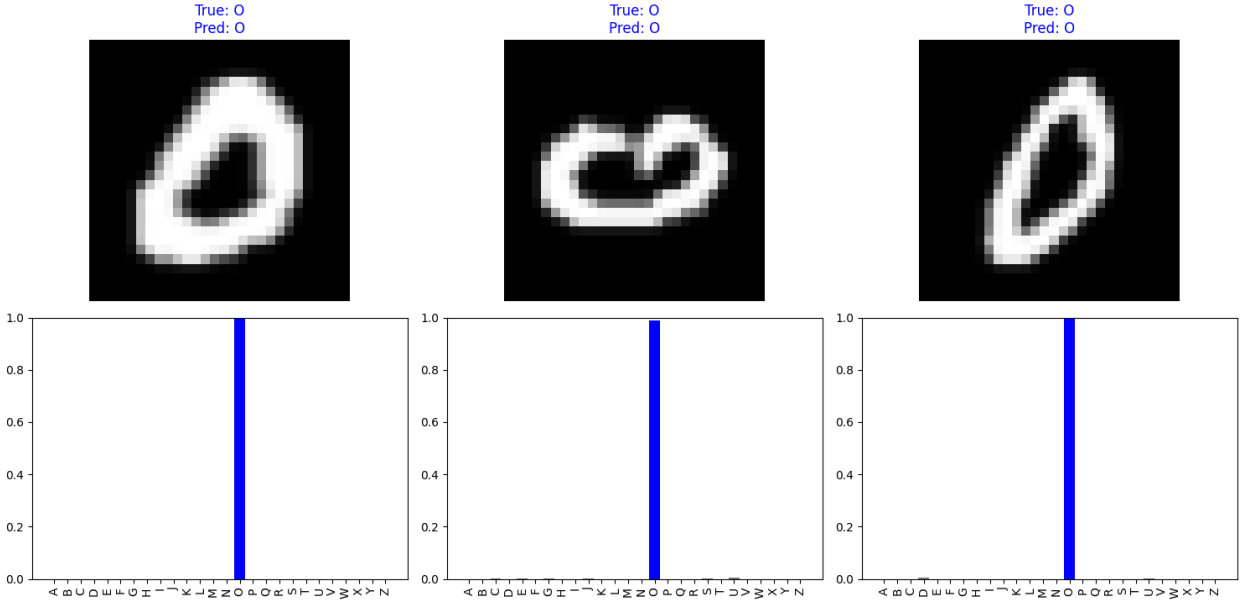




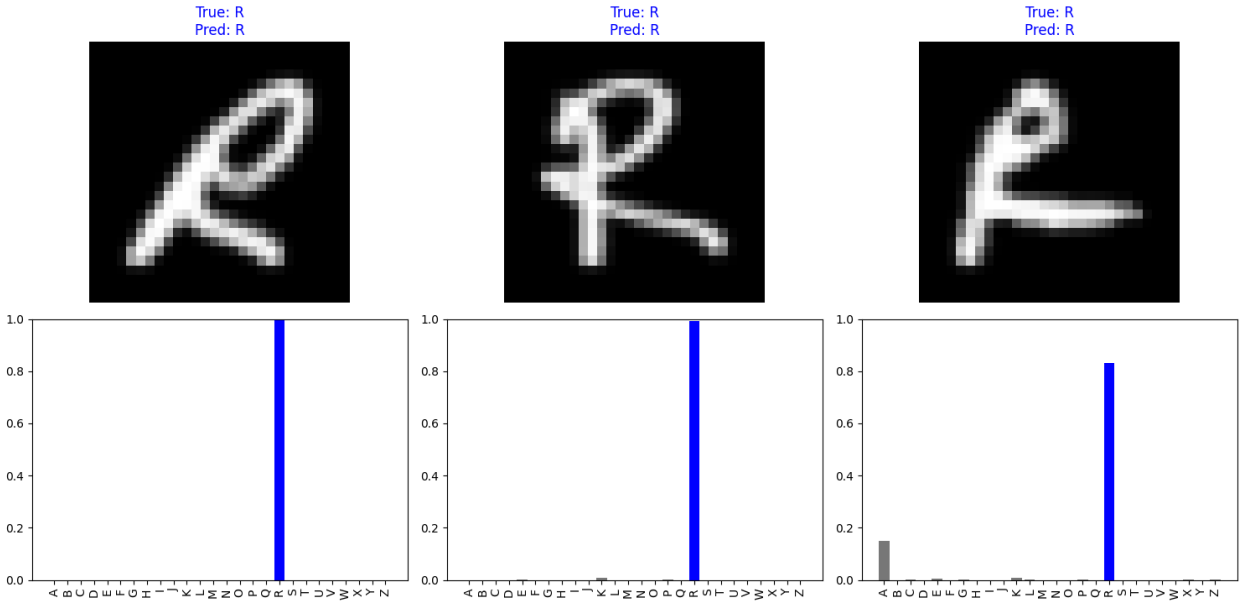


Test with different images for the same char

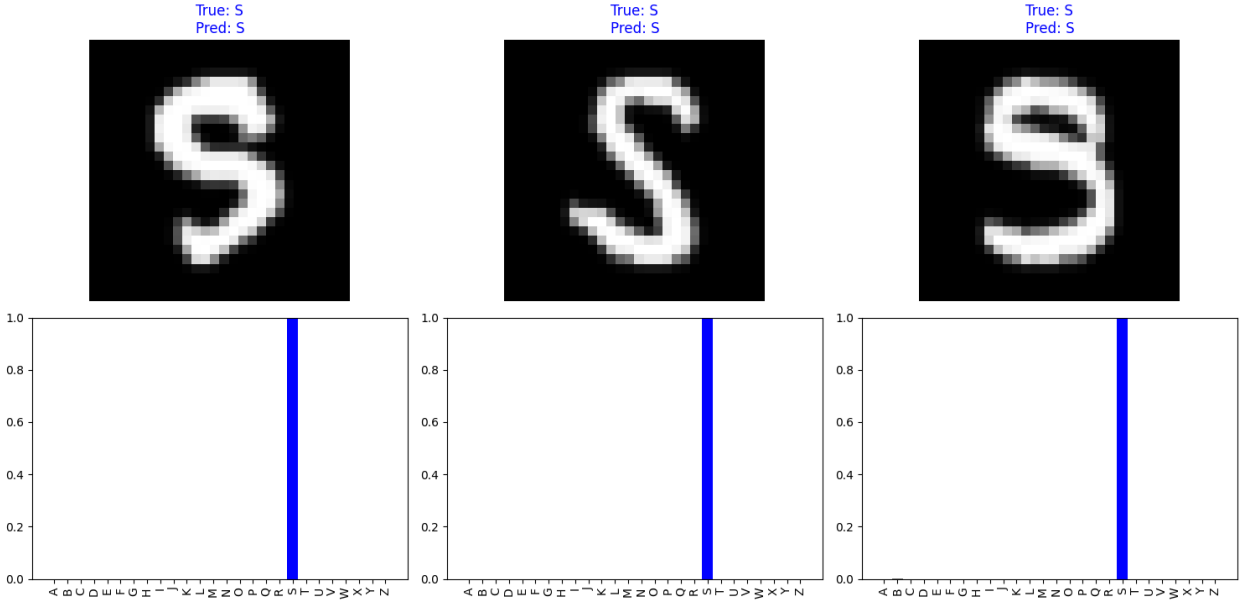
Character: O



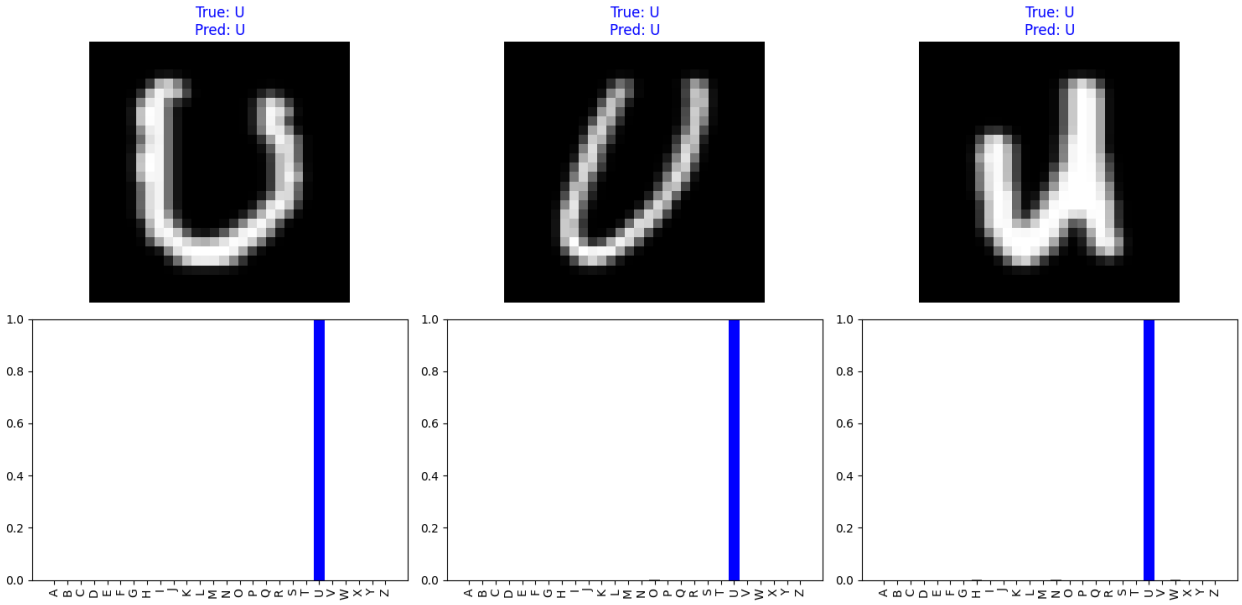
Character: R

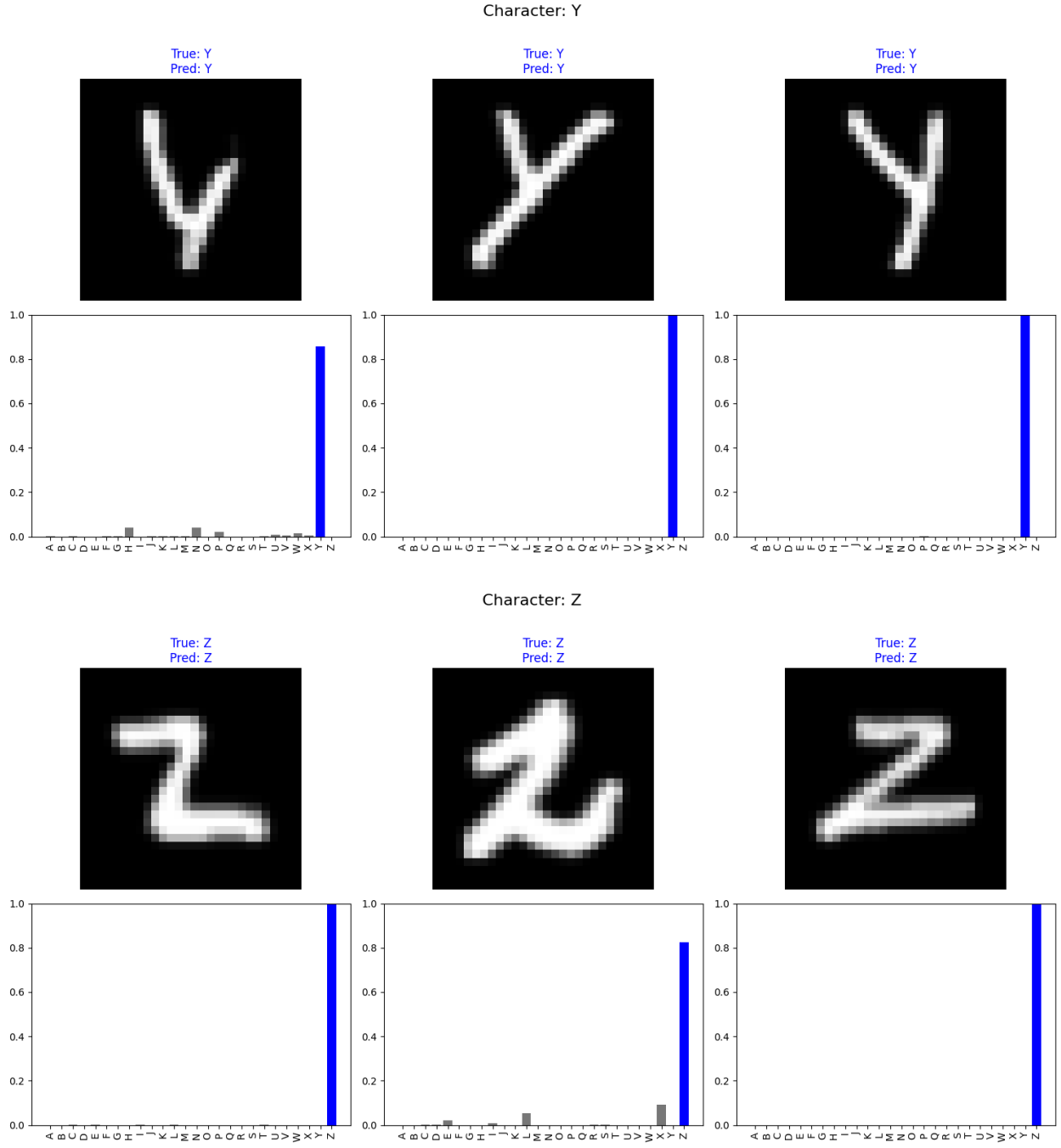


Character: S



Character: U

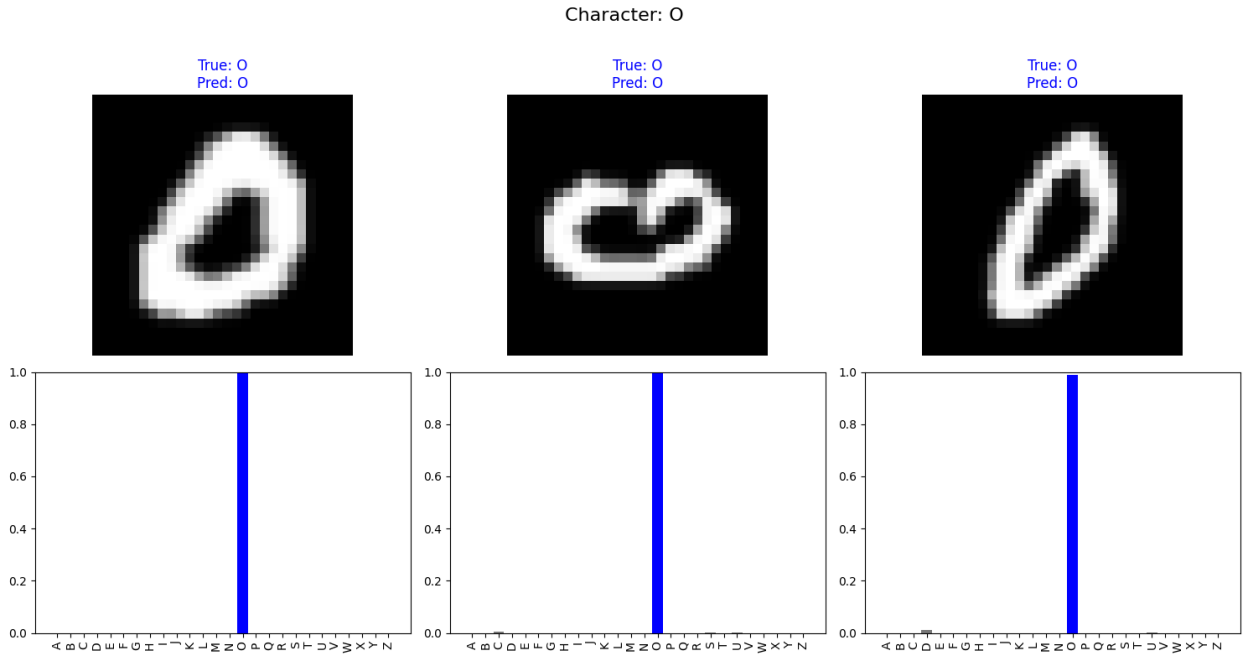




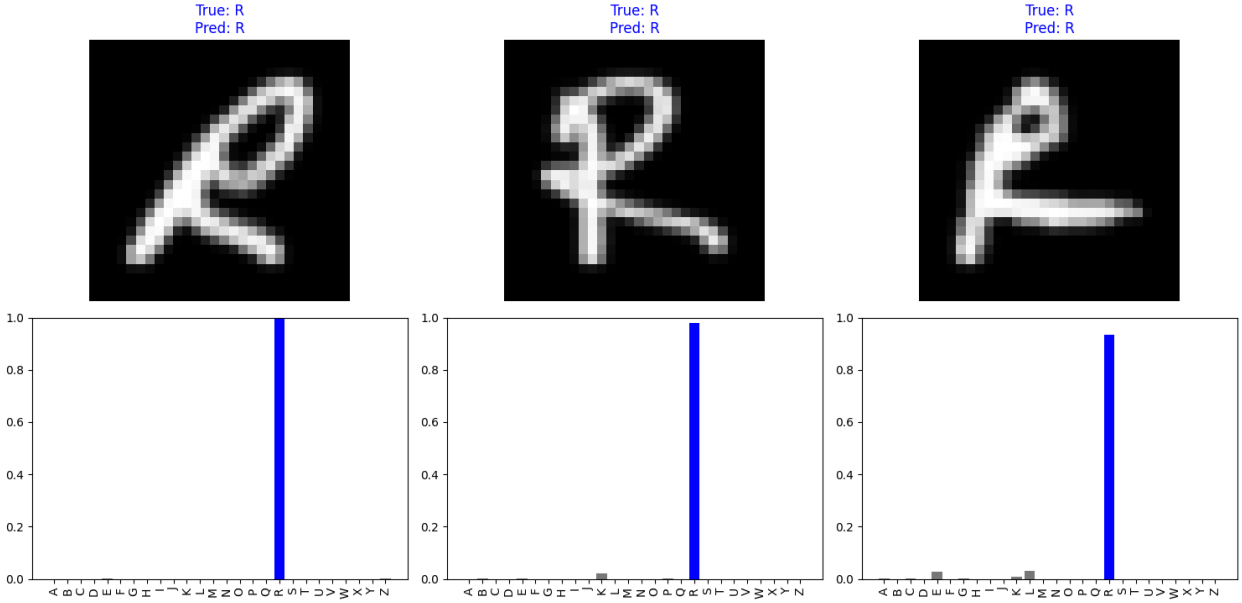
Test Model 1 in Neural in Test images to see the difference

```python
# Load the trained model
model = load_model('First_Try_model.h5')  # Ensure the correct path to the model

# Define the paths to the images for all test characters (3 images each)
image_paths = [
    "/kaggle/input/test-images/S_1.png", "/kaggle/input/test-images/S_2.png", "/kaggle/input/test-images/S_3.png",
    "/kaggle/input/test-images/R_1.png", "/kaggle/input/test-images/R_2.png", "/kaggle/input/test-images/R_3.png",
    "/kaggle/input/test-images/Y_1.png", "/kaggle/input/test-images/Y_2.png", "/kaggle/input/test-images/Y_3.png",
    "/kaggle/input/test-images/Z_1.png", "/kaggle/input/test-images/Z_2.png", "/kaggle/input/test-images/Z_3.png",
    "/kaggle/input/test-images/O_1.png", "/kaggle/input/test-images/O_2.png", "/kaggle/input/test-images/O_3.png",
    "/kaggle/input/test-images/U_1.png", "/kaggle/input/test-images/U_2.png", "/kaggle/input/test-images/U_3.png"
]

# Corresponding true labels for the images
test_labels = [
    'S', 'S', 'S',
    'R', 'R', 'R',
    'Y', 'Y', 'Y',
    'Z', 'Z', 'Z',
    'O', 'O', 'O',
    'U', 'U', 'U'
]
```

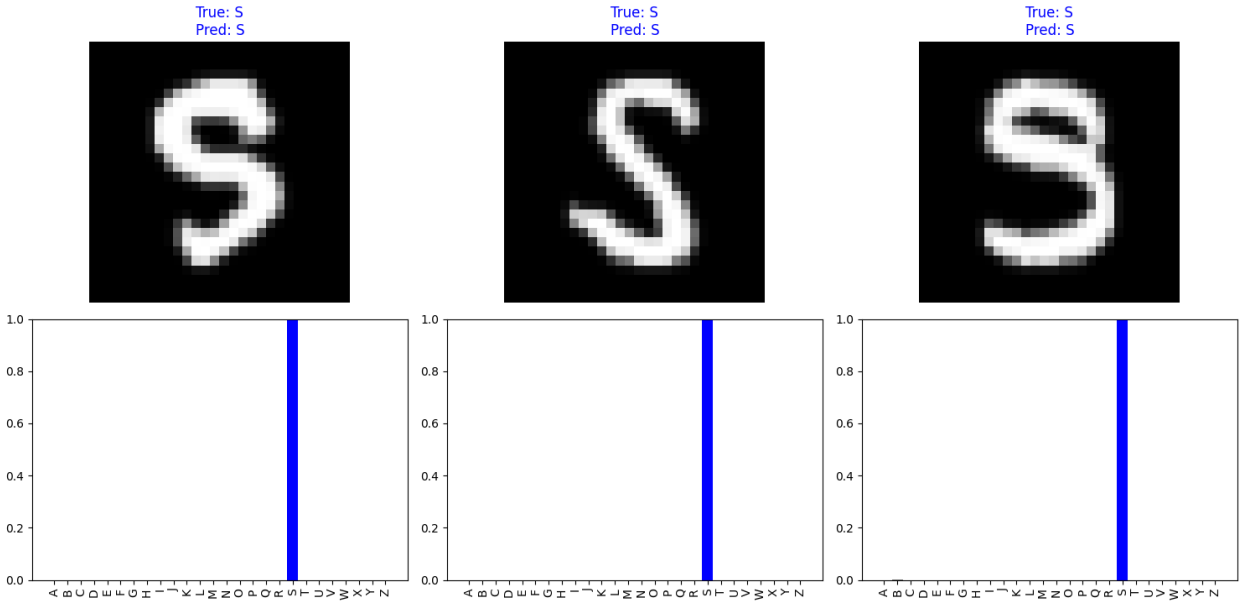
The same Remaining code



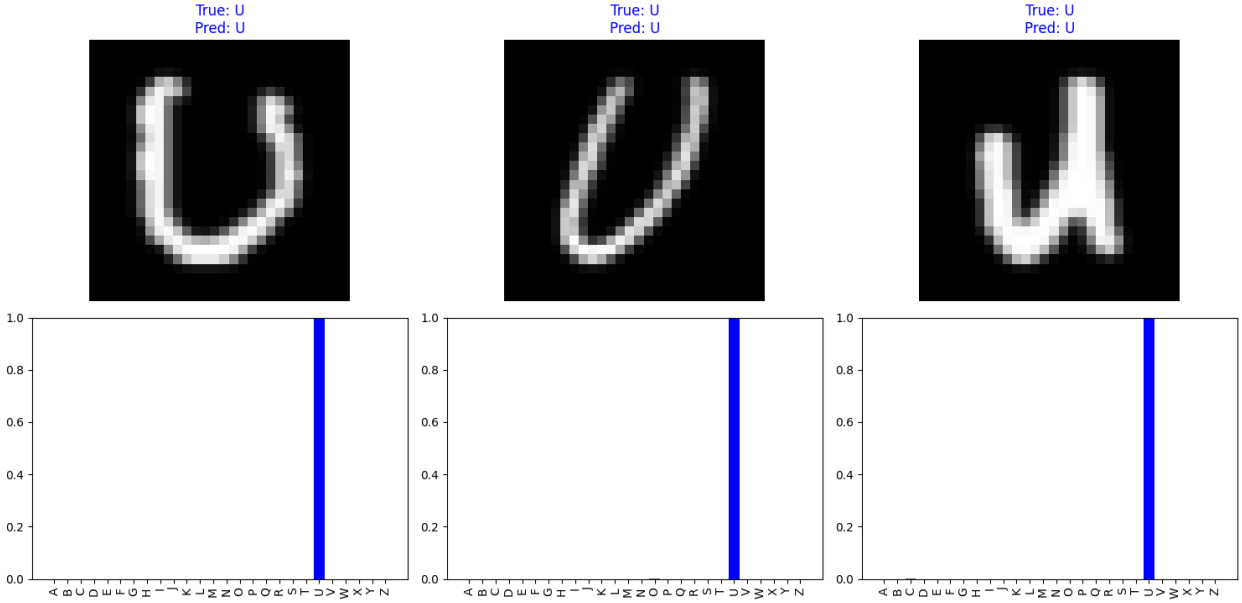
Character: R



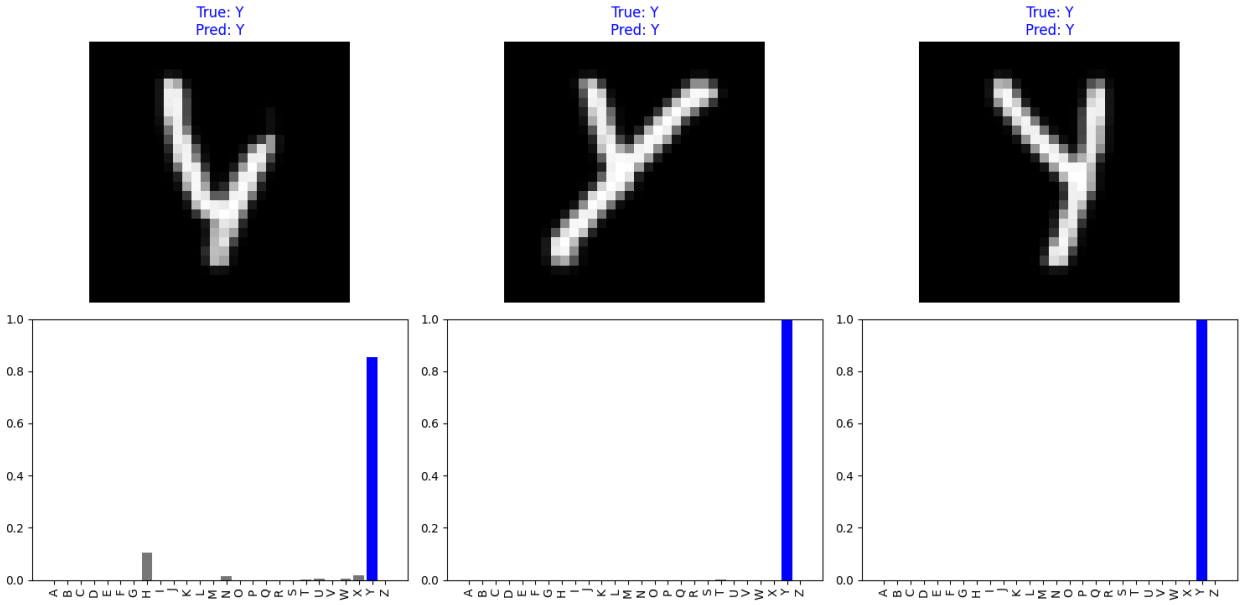
Character: S

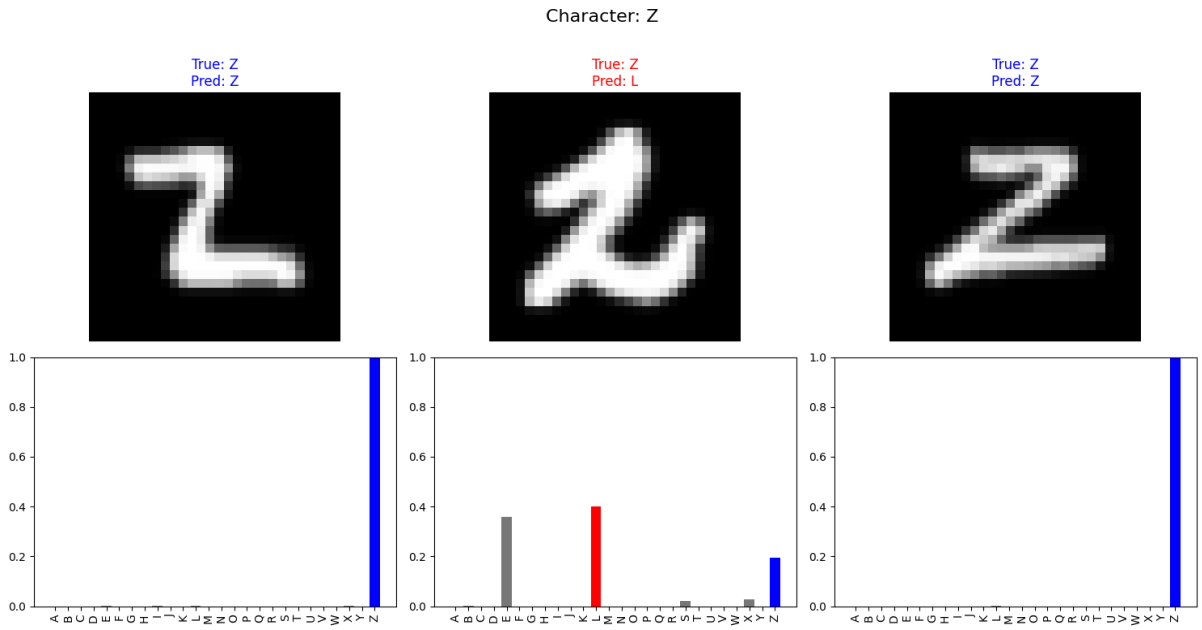


Character: U



Character: Y





Like we see here model 1 fail to predict one sample of letter Z while optimized model got it .

Compare the results of the models and suggest the best model.

Attribute	SVM Linear	SVM Non-Linear	Logistic Regression From Scratch	Neural Network Model 1	Neural Network optimized Model
Test Accuracy	88.98%	94.79%	85 %	97 %	98 %
F1- Score	0.89	0.95	0.80	0.96	0.98
Time for Training	Working on sample 10% of data – 10 minutes	Working on sample 10% of data – 10 minutes	From 80 Minutes to 150 minutes Depend on Number of iterations and Learning Rate	Around 10 minutes on Kaggle using Accelerator 20 epoch	Around 20 minutes on Kaggle using Accelerator – 50 epoch
Training Accuracy			84.62%	0.9594	0.9685
Training Loss				0.1380	0.1505
Validation Accuracy			84.39%	0.9738	0.9820
Validation Loss				0.9738	0.1102

Model	Advantages	Disadvantages
SVM (Linear)	- Simple and interpretable kernel. - Fast on small or moderately sized datasets.	- Poor performance on non-linear data. - May not scale well to very large datasets. -very slow in large dataset
SVM (Non-Linear)	- Handles complex, non-linear data relationships well. - Higher accuracy than Linear SVM.	- Computationally expensive on larger datasets. - Kernel tuning can be challenging.
Logistic Regression	- Easy to interpret and explain results. - Probabilistic outputs are useful in decision-making.	- Poor performance on non-linear data. - Training is time-intensive (due to scratch implementation).
Neural Network Model 1	- High accuracy and F1-score (97%, 0.96). - Able to handle highly complex data effectively.	- Requires careful tuning of hyperparameters. - Needs more computational resources than SVM.
Neural Network (Optimized)	- Best accuracy and F1-score (98%, 0.98). - Optimized for higher efficiency with reduced losses.	- Requires more computational resources and training time compared to simpler models. - Less interpretable.

Best one is Neural Network Optimized