Université Hassan II de Casablanca
Ecole Normale Supérieure de l'Enseignement
Technique de Mohammedia

ECOLE NORMALE SUPÉRIEURE DE L'ENSEIGNEMENT
TECHNIQUE DE MOHAMMEDIA
UNIVERSITÉ HASSAN II DE CASABLANCA

Data mining & Deep Learning

Image classification: LeNet5 architecture enhancement

Classification of traffic signs for autonomous cars

Directed by : Abdelhamid LABIHI

Major : GLSID 3

Academic year : 2024-2025

Contents

1. Introduction

This project aims to build a model capable of automatically classifying traffic sign images. This task is essential for advanced driver assistance systems and autonomous vehicles. We use a dataset containing several panel types, pre-process the data, then train a CNN to perform the classification.

2. Dataset overview

```
Dataset extracted to: /content/dataset
Train Keys: dict_keys(['coords', 'labels', 'features', 'sizes'])
Number of training examples: 34799
Feature shape: (34799, 32, 32, 3)
Labels shape: (34799,)
```

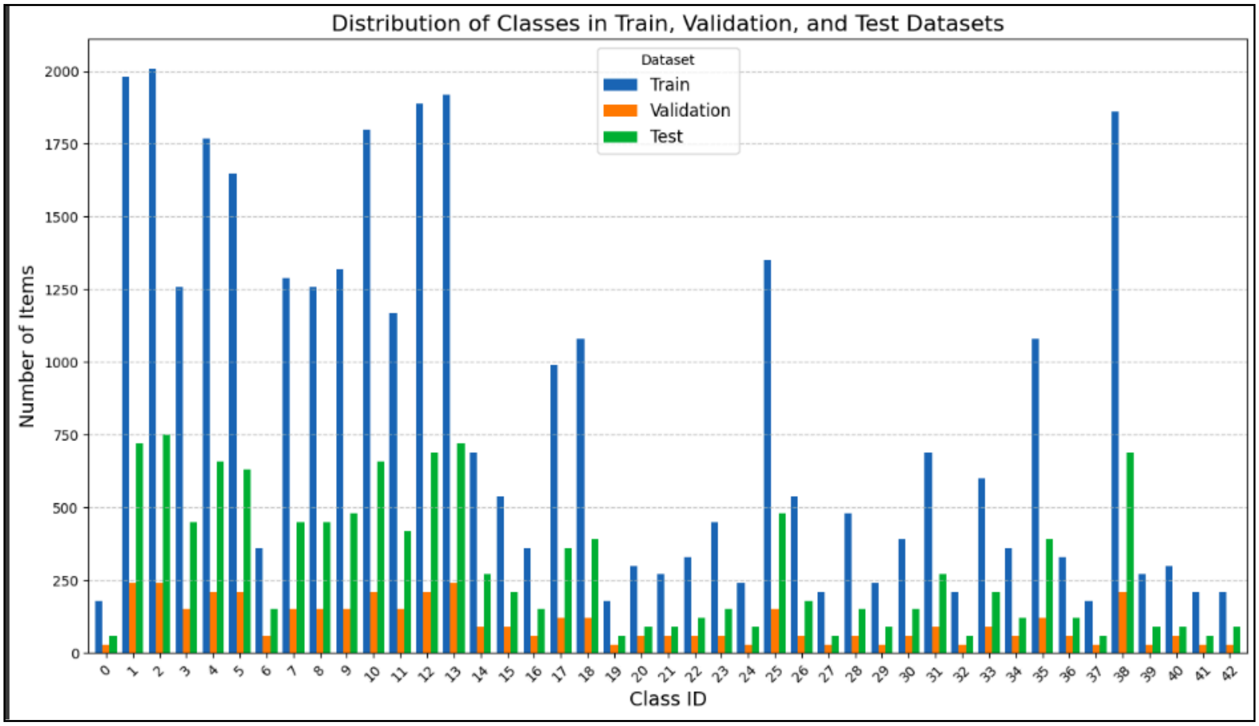
The stripped data is a dictionary composed of 4 key/value pairs:

- ***features*** is a 4D array containing the raw pixel data of the traffic sign images (example numbers, width, height, channels).
- ***labels*** is a 1D array containing the sign label/class identifier. The signnames.csv file contains the identifier -> name correspondences for each identifier.
- ***sizes*** is a list containing tuples, (width, height) representing the original width and height of the image.
- ***coords*** is a list containing tuples (x1, y1, x2, y2) representing the coordinates of a bounding box around the sign in the image.

Dataset link in *kaggle : [dataset]*

Images Examples :

Class distribution in training, validation and test data sets:



3. Implementation of a convolutional neural network based on the architecture of the LeNet5 model

```
modell = Sequential([
    # C1: Convolutional layer with 6 filters 5x5, ReLU activation, input 32x32x3
    Conv2D(6, (5, 5), activation='relu', input_shape=(32, 32, 3)),

    # S2: SubSampling (AveragePooling) 2x2
    AveragePooling2D(pool_size=(2, 2)),

    # C3: Second convolutional layer with 16 filters 5x5, ReLU activation
    Conv2D(16, (5, 5), activation='relu'),

    # S4: SubSampling (AveragePooling) 2x2
    AveragePooling2D(pool_size=(2, 2)),

    # Flatten to transition to dense layers
    Flatten(),

    # C5: Fully connected layer with 120 neurons
    Dense(120, activation='relu'),

    # F6: Fully connected layer with 84 neurons
    Dense(84, activation='relu'),

    # Output layer with 43 neurons (for 43 classification classes)
    Dense(43, activation='softmax')
])

# Display the model summary
modell.summary()

# Compile the model
modell.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
```

Model: "sequential_3"

Layer (type)	Output Shape	Param #
conv2d_6 (Conv2D)	(None, 28, 28, 6)	456
average_pooling2d_6 (AveragePooling2D)	(None, 14, 14, 6)	0
conv2d_7 (Conv2D)	(None, 10, 10, 16)	2,416
average_pooling2d_7 (AveragePooling2D)	(None, 5, 5, 16)	0
flatten_3 (Flatten)	(None, 400)	0
dense_9 (Dense)	(None, 120)	48,120
dense_10 (Dense)	(None, 84)	10,164
dense_11 (Dense)	(None, 43)	3,655

Total params: 64,811 (253.17 KB)
Trainable params: 64,811 (253.17 KB)
Non-trainable params: 0 (0.00 B)

- Model training and evaluation

```
Epoch 10/10
1088/1088 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.9842 - loss: 0.0653 - val_accuracy: 0.9234 - val_loss: 0.7217
```

Evaluate on test and validation data

```
[ ] test_loss_model1, test_accuracy_model1 = model1.evaluate(X_test, y_test)
    print(f"Test Accuracy: {test_accuracy_model1 * 100:.2f}%")
```

```
395/395 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9162 - loss: 0.7239
Test Accuracy: 91.55%
```

```
[ ] val_loss_model1, val_accuracy_model1 = model1.evaluate(X_valid, y_valid)

    print(f"Validation Loss: {val_loss_model1 * 100:.2f}%")
    print(f"Validation Accuracy:{ val_accuracy_model1 * 100:.2f}%")
```

```
138/138 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9074 - loss: 0.7842
Validation Loss: 72.17%
Validation Accuracy:92.34%
```

- Observations

Initial performance (epoch 1):

Training accuracy: 0.5742 (57.42%)

Training loss: 1.9987

The model starts with relatively low accuracy, indicating that it is still learning the basic features.

Validation accuracy: 0.8311 (83.11%)

Validation loss: 0.7008

Validation accuracy is higher than training accuracy, which may suggest that the model generalizes well initially.

Improvement over time:

Training accuracy increases significantly, reaching 98.42% in epoch 10.

Training loss decreases steadily from 1.9987 in epoch 1 to 0.0653 in epoch 10.

Validation accuracy fluctuates slightly but remains high, reaching 92.34% at epoch 10.

Validation loss behavior:

Validation loss initially decreases, indicating better generalization.

However, it begins to fluctuate and increase in later epochs (e.g. 0.8695 in epoch 6, 0.7217 in epoch 10), suggesting overfitting.

Overfitting occurs when the model performs very well on training data, but struggles to generalize to unseen validation data.

Final performance:

Training accuracy: 98.42

Validation accuracy: 92.34

A large discrepancy (~6%) between training and validation accuracy suggests that the model works well, but may be slightly overfitting.

4. Implementing the LeNet5 model with droupout

```
# Define the model
model2 = Sequential([
    # C1: Convolutional layer with 6 filters 5x5, ReLU activation, input 32x32x3
    Conv2D(6, (5, 5), activation='relu', input_shape=(32, 32, 3)),
    # Dropout after the first convolutional layer
    Dropout(0.2),  # Drop 20% of neurons randomly
    # Explanation: Helps prevent reliance on specific features in the initial convolution.

    # S2: SubSampling (AveragePooling) 2x2
    AveragePooling2D(pool_size=(2, 2)),

    # C3: Second convolutional layer with 16 filters 5x5, ReLU activation
    Conv2D(16, (5, 5), activation='relu'),
    # Dropout after the second convolutional layer
    Dropout(0.3),  # Drop 30% of neurons randomly
    # Explanation: Further reduces reliance on specific filters and promotes generalization.

    # S4: SubSampling (AveragePooling) 2x2
    AveragePooling2D(pool_size=(2, 2)),

    # Flatten to transition to dense layers
    Flatten(),

    # C5: Fully connected layer with 120 neurons
    Dense(120, activation='relu'),
    # Dropout after the first dense layer
    Dropout(0.5),  # Drop 50% of neurons randomly
    # Explanation: Dense layers have a high number of parameters, making them prone to overfitting.

    # F6: Fully connected layer with 84 neurons
    Dense(84, activation='relu'),
    # Dropout after the second dense layer
    Dropout(0.5),  # Drop 50% of neurons randomly
    # Explanation: Helps further reduce overfitting in fully connected layers.

    # Output layer with 43 neurons (for 43 classification classes)
    Dense(43, activation='softmax')
])

# Display the model summary
model2.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 28, 28, 6)	456
dropout (Dropout)	(None, 28, 28, 6)	0
average_pooling2d (AveragePooling2D)	(None, 14, 14, 6)	0
conv2d_1 (Conv2D)	(None, 10, 10, 16)	2,416
dropout_1 (Dropout)	(None, 10, 10, 16)	0
average_pooling2d_1 (AveragePooling2D)	(None, 5, 5, 16)	0
flatten (Flatten)	(None, 400)	0
dense (Dense)	(None, 120)	48,120
dropout_2 (Dropout)	(None, 120)	0
dense_1 (Dense)	(None, 84)	10,164
dropout_3 (Dropout)	(None, 84)	0
dense_2 (Dense)	(None, 43)	3,655

Total params: 64,811 (253.17 KB)
Trainable params: 64,811 (253.17 KB)
Non-trainable params: 0 (0.00 B)

- Model training with epochs = 10

```
training2 = model2.fit(X_train, y_train, validation_data=(X_valid, y_valid), epochs=10, batch_size=32)
```

```
Epoch 1/10
1088/1088 ━━━━━━━━━━━━━━━━━━━━ 33s 28ms/step - accuracy: 0.1204 - loss: 5.8561 - val_accuracy: 0.4712 - val_loss: 1.7793
Epoch 2/10
1088/1088 ━━━━━━━━━━━━━━━━━━━━ 30s 27ms/step - accuracy: 0.4481 - loss: 1.8764 - val_accuracy: 0.6918 - val_loss: 1.0218
Epoch 3/10
1088/1088 ━━━━━━━━━━━━━━━━━━━━ 48s 34ms/step - accuracy: 0.5962 - loss: 1.3002 - val_accuracy: 0.7730 - val_loss: 0.7915
Epoch 4/10
1088/1088 ━━━━━━━━━━━━━━━━━━━━ 36s 33ms/step - accuracy: 0.6680 - loss: 1.0674 - val_accuracy: 0.7900 - val_loss: 0.5923
Epoch 5/10
1088/1088 ━━━━━━━━━━━━━━━━━━━━ 40s 32ms/step - accuracy: 0.7282 - loss: 0.8893 - val_accuracy: 0.8109 - val_loss: 0.5652
Epoch 6/10
1088/1088 ━━━━━━━━━━━━━━━━━━━━ 38s 30ms/step - accuracy: 0.7497 - loss: 0.7941 - val_accuracy: 0.8728 - val_loss: 0.4540
Epoch 7/10
1088/1088 ━━━━━━━━━━━━━━━━━━━━ 35s 32ms/step - accuracy: 0.7699 - loss: 0.7464 - val_accuracy: 0.8844 - val_loss: 0.4079
Epoch 8/10
1088/1088 ━━━━━━━━━━━━━━━━━━━━ 36s 27ms/step - accuracy: 0.7937 - loss: 0.6688 - val_accuracy: 0.9020 - val_loss: 0.3396
Epoch 9/10
1088/1088 ━━━━━━━━━━━━━━━━━━━━ 42s 29ms/step - accuracy: 0.8182 - loss: 0.6032 - val_accuracy: 0.8728 - val_loss: 0.4305
Epoch 10/10
1088/1088 ━━━━━━━━━━━━━━━━━━━━ 41s 28ms/step - accuracy: 0.8128 - loss: 0.6216 - val_accuracy: 0.9014 - val_loss: 0.3603
```

- Increasing epochs to 25

```
Epoch 25/25
1088/1088 ━━━━━━━━━━━━━━━━━━━━ 43s 32ms/step - accuracy: 0.8602 - loss: 0.5008 - val_accuracy: 0.9383 - val_loss: 0.2417
```

Evaluate on test and validation data

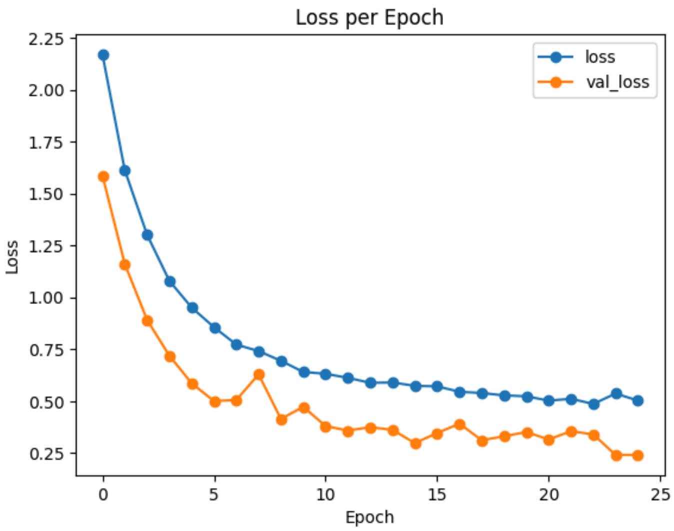
```
test_loss_model2, test_accuracy_model2 = model2.evaluate(X_test, y_test)
print(f"Test Accuracy: {test_accuracy_model2 * 100:.2f}%")
```

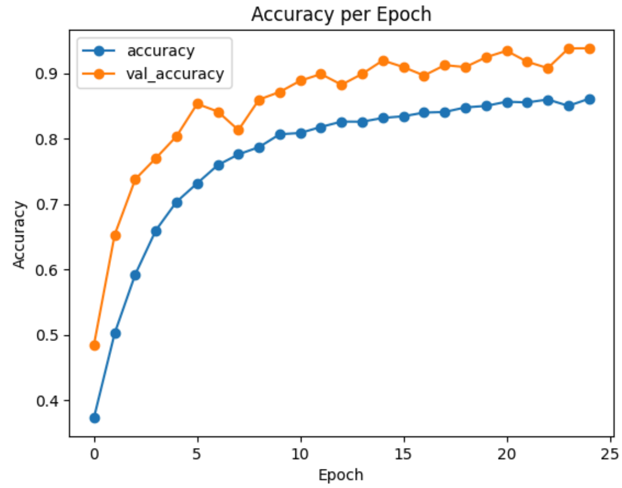
```
395/395 ━━━━━━━━━━━━━━━━━━━━ 3s 9ms/step - accuracy: 0.9125 - loss: 0.3257
Test Accuracy: 91.60%
```

```
val_loss_model2, val_accuracy_model2 = model2.evaluate(X_valid, y_valid)

print(f"Validation Loss: {val_loss_model2 * 100:.2f}%")
print(f"Validation Accuracy:{ val_accuracy_model2 * 100:.2f}%")
```

```
138/138 ━━━━━━━━━━━━━━━━━━━━ 2s 15ms/step - accuracy: 0.9336 - loss: 0.2654
Validation Loss: 24.17%
Validation Accuracy:93.83%
```





5. Design and test an alternative architecture model

- **Data normalization:**

As a minimum, image data should be normalized so that the mean of the data is zero and the variance is equal. For image data, (pixel - 128)/ 128 is a quick way of approximately normalizing the data and can be used in this project.

Here, I use a simple normalization technique - (x-min)/(max-min) for all pixels in the image.

This technique normalizes pixel values between 0 and 1, which improves contrast. Images before and after normalization are shown below.

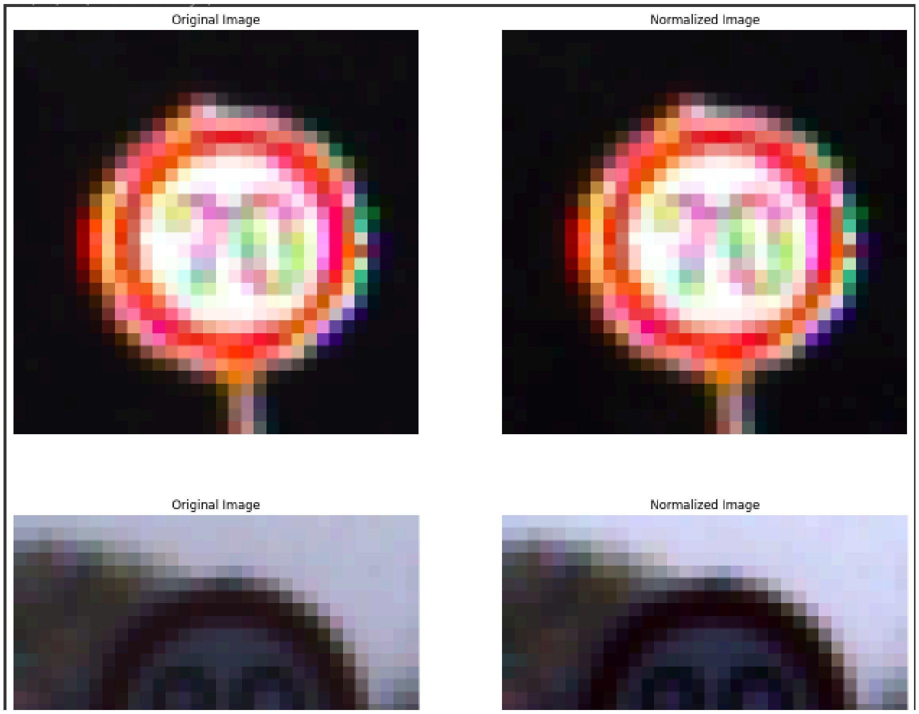
– Normalization function

```
[ ] def normalize(x):
        """
            argument
                - x: input image data in numpy array [length, width, color_depth]
            return
                - normalized x
        """
        min_val = np.min(x)
        max_val = np.max(x)
        x = (x-min_val) / (max_val-min_val)
        return x
```

```
[ ] normalized_train = np.zeros([n_train,32,32,3])
    normalized_validation = np.zeros([n_validation,32,32,3])
    normalized_test = np.zeros([n_test,32,32,3])
    for i in range(0,n_train):
        normalized_train[i] = normalize(X_train[i])
    for i in range(0,n_validation):
        normalized_validation[i] = normalize(X_valid[i])
    for i in range(0,n_test):
        normalized_test[i] = normalize(X_test[i])

    assert(X_train.shape==normalized_train.shape)
```

the difference standardization makes:



- Model architecture :

```
[ ] from tensorflow.keras.models import Sequential
    from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout

    model3 = Sequential([
        # C1: Convolutional layer with 6 filters 5x5, ReLU activation, input 32x32x3
        Conv2D(6, (5, 5), activation='relu', input_shape=(32, 32, 3), padding='valid'),
        MaxPooling2D(pool_size=(2, 2)),

        # C3: Second convolutional layer with 16 filters 5x5, ReLU activation
        Conv2D(16, (5, 5), activation='relu', padding='valid'),
        MaxPooling2D(pool_size=(2, 2)),

        # Extra convolutional layer (Added Improvement)
        Conv2D(412, (5, 5), activation='relu', padding='valid'),

        # Flatten for fully connected layers
        Flatten(),

        # Fully connected layers with dropout (Added Improvement)
        Dense(122, activation='relu'),
        Dropout(0.5),  # Dropout with 50% rate
        Dense(84, activation='relu'),
        Dropout(0.5),  # Dropout with 50% rate

        # Output layer for 43 classes
        Dense(43, activation='softmax')
    ])

    # Compile the model
    model3.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])

    # Display the model summary
    model3.summary()
```

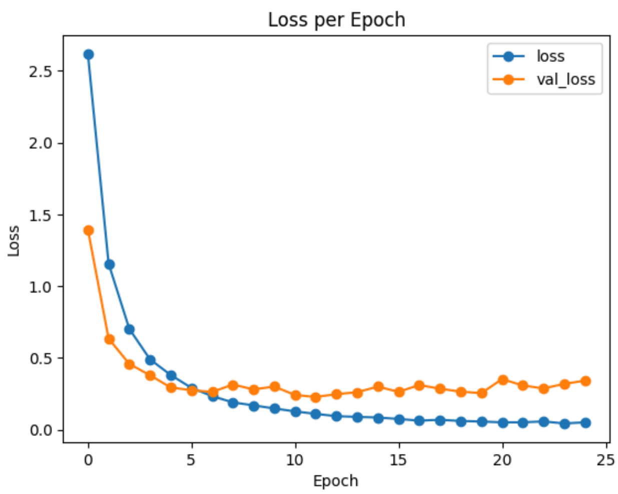
Model: "sequential_2"

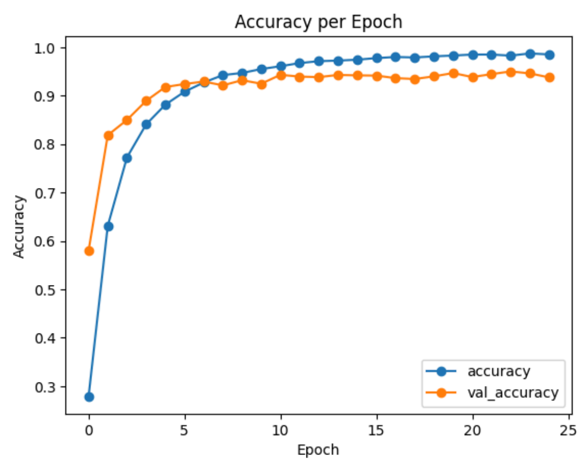
Layer (type)	Output Shape	Param #
conv2d_5 (Conv2D)	(None, 28, 28, 6)	456
max_pooling2d_2 (MaxPooling2D)	(None, 14, 14, 6)	0
conv2d_6 (Conv2D)	(None, 10, 10, 16)	2,416
max_pooling2d_3 (MaxPooling2D)	(None, 5, 5, 16)	0
conv2d_7 (Conv2D)	(None, 1, 1, 412)	165,212
flatten_2 (Flatten)	(None, 412)	0
dense_6 (Dense)	(None, 122)	50,386
dropout_6 (Dropout)	(None, 122)	0
dense_7 (Dense)	(None, 84)	10,332
dropout_7 (Dropout)	(None, 84)	0
dense_8 (Dense)	(None, 43)	3,655

Total params: 232,457 (908.04 KB)
Trainable params: 232,457 (908.04 KB)
Non-trainable params: 0 (0.00 B)

```
Epoch 25/25
224/224 ━━━━━━━━━━━━━━━━━━━━ 31s 136ms/step - accuracy: 0.9852 - loss: 0.0539 - val_accuracy: 0.9379 - val_loss: 0.3419
Validation Accuracy: 0.9378684759140015
```

- Visualisation:





6. Test the model on new images

Test images before normalization

After normalization

- Predict the type of panel for each image

```python
from tensorflow.keras.models import load_model
import numpy as np

# Load the trained Keras model
model = load_model('/content/lenet_best_model.keras')

# Ensure images are in the correct shape and normalized
# Assume `my_signs_normalized` is already normalized and in (6, 32, 32, 3) shape
print("Shape of my_signs_normalized:", my_signs_normalized.shape)

# Predict the sign types
predictions = model.predict(my_signs_normalized)

# Decode predictions
predicted_classes = np.argmax(predictions, axis=1)
print("Predicted classes:", predicted_classes)

# Map predicted classes to labels
for idx, pred_class in enumerate(predicted_classes):
    print(f"Image {idx + 1}: Predicted Class = {pred_class}, True Label = {my_labels[idx]}")
```

```
Shape of my_signs_normalized: (6, 32, 32, 3)
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 105ms/step
Predicted classes: [18  1 38 34 25  3]
Image 1: Predicted Class = 18, True Label = 18
Image 2: Predicted Class = 1, True Label = 1
Image 3: Predicted Class = 38, True Label = 38
Image 4: Predicted Class = 34, True Label = 34
Image 5: Predicted Class = 25, True Label = 25
Image 6: Predicted Class = 3, True Label = 3
```

- Performance analysis

```
My Data Set Accuracy = 100.00%
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 151ms/step
Image 1: Predicted Class = 18, True Label = 18
Correct Prediction!
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 22ms/step
Image 2: Predicted Class = 1, True Label = 1
Correct Prediction!
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 23ms/step
Image 3: Predicted Class = 38, True Label = 38
Correct Prediction!
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 22ms/step
Image 4: Predicted Class = 34, True Label = 34
Correct Prediction!
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 22ms/step
Image 5: Predicted Class = 25, True Label = 25
Correct Prediction!
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 26ms/step
Image 6: Predicted Class = 3, True Label = 3
Correct Prediction!
```

Overview of models :

```
     Model  Train Accuracy  Train Loss  Validation Accuracy  Validation Loss  \
0  Model 1          98.42%       6.53%               90.74%           78.42%
1  Model 2          86.02%      50.08%               93.83%           24.17%
2  Model 3          98.52%       5.39%               93.79%           34.19%

  Test Accuracy  Test Loss
0        91.62%     72.39%
1        91.60%     32.57%
2        94.76%     20.14%

Formatted Summary Table:

+---+---------+----------------+--------------------+---------------------+--------------------+--------------------+-----------+
|   |  Model  | Train Accuracy |     Train Loss     | Validation Accuracy |  Validation Loss   |   Test Accuracy    | Test Loss |
+---+---------+----------------+--------------------+---------------------+--------------------+--------------------+-----------+
| 0 | Model 1 |     98.42      | 6.529999999999999  |        90.74        |       78.42        |       91.62        |   72.39   |
| 1 | Model 2 |     86.02      | 50.080000000000005 |        93.83        | 24.169999999999998 | 91.600000000000001 |   32.57   |
| 2 | Model 3 |     98.52      | 5.390000000000001  |  93.78999999999999  |       34.19        |       94.76        |   20.14   |
+---+---------+----------------+--------------------+---------------------+--------------------+--------------------+-----------+
```

7. Conclusion

This project shows that CNNs are powerful tools for image classification, particularly in the field of traffic signs. Next steps could include integrating this model into an embedded system for real-time application.