Image classification

December 7, 2025

1 Deep Learning Project

1.1 Road Sign Classification using Deep Learning

1.1.1 Problem Selection:

I selected an image classification problem focused on recognizing and categorizing road signs. Specifically, our task involves classifying images into at least three classes: Traffic Light, Stop Sign, and Speed Limit Sign. This problem is crucial for developing autonomous driving systems and improving road safety by enabling vehicles to recognize and respond to road signs automatically.

1.1.2 Factors Considered:

Data Availability: I need a dataset that contains a sufficient number of labeled images for these classes. Luckily for me, I was able to find a dataset at kagel for road sign classification.

Usefulness of the Solution: The ability to classify road signs has practical applications in autonomous driving and traffic management systems.

Computational Budget: I must consider the resources required to train a deep learning model, including GPU access and memory constraints.

1.1.3 Theoretical and Practical Challenges:

Theoretical Challenges:

Model Generalization: Ensuring the model generalizes well to new, unseen images.

Class Imbalance: Some classes may have more images than others, potentially leading to biased predictions.

Overfitting: Preventing the model from performing well on training data but poorly on new data.

Practical Challenges:

Data Preprocessing: Handling varying image sizes, qualities, and lighting conditions.

Computational Resources: Efficiently using available resources to train the model within a reasonable time frame.

Real-World Variability: Road signs may appear differently due to wear, obstructions, or lighting, which the model must learn to handle. ’’’

1.1.4 Desired Inputs and Outputs:

Inputs: The input will be images of road signs.

Outputs: The output will be a class label predicting whether the image is a Traffic Light, Stop Sign, Pedestrian Crossing or Speed Limit Sign.

Target Classes: The target classes for this problem are:

Traffic Light

Stop Sign

Speed Limit Sign

Pedestrian Crossing

1.1.5 Dataset Selection:

Dataset Choice: I will use a dataset containing 877 images of road signs, organized into four distinct classes:

Traffic Light, Stop, Speed Limit, and Crosswalk. The images come with bounding box annotations in the PASCAL VOC format, which are used to crop out the required image of each class for road sign classification.

Dataset Link: https://www.kaggle.com/datasets/andrewmvd/road-sign-detection

1.1.6 Data Requirements:

Total Images Available:

The dataset includes 877 images, which are distributed across the four classes. This gives us a good starting point for training a deep learning model for road sign detection.

Training and Testing Split:

Training Set: I will use 80% of the dataset for training, which amounts to approximately 701 images.

Testing Set: The remaining 20%, or about 176 images, will be reserved for testing. This split will help in evaluating the model's generalization capability on unseen data.

Labeling the Images:

Labeling Requirement: The dataset already includes bounding box annotations in the PASCAL VOC format, so no additional manual labeling is required. The bounding boxes will allow to crop out the required image location within the images.

1.1.7 Model Evaluation Criteria:

Determining Model Performance:

Accuracy: Since this is a detection task, I will track metrics like Intersection over Union (IoU) and mean Average Precision (mAP) during validation to evaluate the model's performance.

Confusion Matrix: This metric is used for classification tasks, a confusion matrix can be generated to assess the model's ability to classify each detected class into the correct category.

Loss: The model's loss function will include components for classification, tracking this loss during training will ensure the model is learning effectively.

Generalization: The model's performance on the test set will be a key indicator of its real-world applicability. I will compare test results to training performance to check for overfitting or underfitting.

By structuring the plan around this dataset, the focus will be on effectively utilizing the annotated bounding boxes to extract required images and train a robust model for road sign classification. This will involve implementing an classification model that can accurately classify and locate the four types of road signs within the images.

[1]:
```
import os
import xml.etree.ElementTree as ET
from PIL import Image
import numpy as np
import matplotlib.pyplot as plt
import time
```

[2]:
```
# Define paths to the dataset
image_folder = "images"
annotation_folder = "annotations"
```

[3]:
```
# Define a function to parse the XML annotation files
def parse_annotation(xml_file):
    tree = ET.parse(xml_file)
    root = tree.getroot()

    # Get image file name
    filename = root.find('filename').text

    # Get bounding box coordinates and labels
    bboxes = []
    labels = []
    for obj in root.findall('object'):
        label = obj.find('name').text
        bndbox = obj.find('bndbox')
        xmin = int(bndbox.find('xmin').text)
        ymin = int(bndbox.find('ymin').text)
        xmax = int(bndbox.find('xmax').text)
        ymax = int(bndbox.find('ymax').text)
        bboxes.append((xmin, ymin, xmax, ymax))
        labels.append(label)

    return filename, bboxes, labels
```

[4]:
```
# Example of loading and cropping images
def load_and_crop_images(image_folder, annotation_folder):
    cropped_images = []
    cropped_labels = []

    for annotation_file in os.listdir(annotation_folder):
        if annotation_file.endswith('.xml'):
            # Parse the annotation file
            xml_path = os.path.join(annotation_folder, annotation_file)
            filename, bboxes, labels = parse_annotation(xml_path)

            # Load the corresponding image
            image_path = os.path.join(image_folder, filename)
            image = Image.open(image_path)

            # Crop the image based on bounding boxes
            for bbox, label in zip(bboxes, labels):
                xmin, ymin, xmax, ymax = bbox
                cropped_image = image.crop((xmin, ymin, xmax, ymax))
                cropped_images.append(cropped_image)
                cropped_labels.append(label)

    return cropped_images, cropped_labels

# Load and crop the images
cropped_images, cropped_labels = load_and_crop_images(image_folder,␣
 ↪annotation_folder)

# Convert labels to numpy array for easier manipulation
cropped_labels = np.array(cropped_labels)
```

[5]:
```
# Visualize images of each class
def visualize_samples_by_class(cropped_images, cropped_labels):
    # Get the unique classes
    unique_classes = np.unique(cropped_labels)
    plt.figure(figsize=(15, 10))

    # Plot one image for each unique class
    for i, cls in enumerate(unique_classes):
        # Find the indices for the current class
        indices = np.where(cropped_labels == cls)[0]

        # Check if any images exist for this class
        if len(indices) > 0:
            plt.subplot(1, len(unique_classes), i + 1)
            plt.imshow(cropped_images[indices[0]])
            plt.title(cls)
```

```
            plt.axis('off')
        else:
            print(f"No images found for class {cls}")

    plt.show()

# Visualize one image per class
visualize_samples_by_class(cropped_images, cropped_labels)
```

[8]:
```
#Model Creation
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelBinarizer
import tensorflow as tf
from tensorflow.keras import layers, models
from sklearn.metrics import confusion_matrix, classification_report,␣
 ↪accuracy_score
import seaborn as sns
```

[9]:
```
# Convert images to numpy arrays and resize to a common shape (e.g., 64x64)
def preprocess_images(cropped_images, image_size=(64, 64)):
    processed_images = []
    for image in cropped_images:
        image = image.convert('RGB')  # Ensure the image has 3 channels
        image = image.resize(image_size)
        image_array = np.array(image)
        processed_images.append(image_array)
    return np.array(processed_images)

# Preprocess images
X = preprocess_images(cropped_images)
```

[10]:
```
# Encode the labels using one-hot encoding
lb = LabelBinarizer()
y = lb.fit_transform(cropped_labels)
```

[82]:
```
# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=42)
```

[83]:
```
# Normalize pixel values to be between 0 and 1
X_train = X_train.astype('float32') / 255.0
X_test = X_test.astype('float32') / 255.0
```

[13]:
```
# Define a simple CNN model
def create_model(input_shape):
    model = models.Sequential([
        layers.Conv2D(32, (3, 3), activation='relu', input_shape=input_shape),
        layers.MaxPooling2D((2, 2)),
        layers.Conv2D(64, (3, 3), activation='relu'),
        layers.MaxPooling2D((2, 2)),
        layers.Conv2D(128, (3, 3), activation='relu'),
        layers.MaxPooling2D((2, 2)),
        layers.Flatten(),
        layers.Dense(128, activation='relu'),
        layers.Dense(len(lb.classes_), activation='softmax')  # Output layer␣
 ↪with number of classes
    ])
    return model

# Create and compile the model
input_shape = (64, 64, 3)  # Image dimensions with 3 channels
model = create_model(input_shape)
model.compile(optimizer='adam', loss='categorical_crossentropy',␣
 ↪metrics=['accuracy'])
```

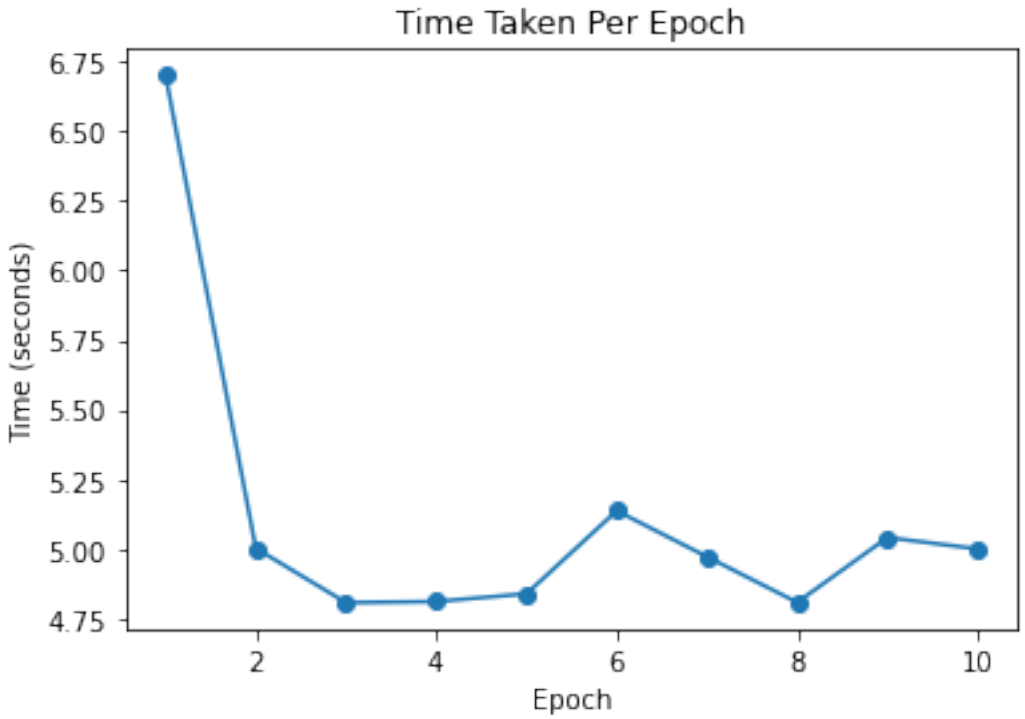
[14]:
```
from tensorflow.keras.callbacks import Callback
class TimeHistory(Callback):
    def on_train_begin(self, logs=None):
        self.times = []

    def on_epoch_begin(self, epoch, logs=None):
        self.epoch_time_start = time.time()

    def on_epoch_end(self, epoch, logs=None):
        self.times.append(time.time() - self.epoch_time_start)
        print(f"Epoch {epoch + 1} took {self.times[-1]:.2f} seconds")
```

[15]:
```
# Train the model

# Instantiate the callback
time_callback = TimeHistory()

start_time = time.time()
history = model.fit(X_train, y_train, epochs=10, validation_data=(X_test,␣
 ↪y_test), batch_size=32, callbacks=[time_callback])
end_time = time.time()
elapsed_time = end_time - start_time
print("Training time: ", elapsed_time)

# Access the recorded times per epoch
times_per_epoch = time_callback.times
```

```
Epoch 1/10
31/32 [============================>.] - ETA: 0s - loss: 0.7414 - accuracy:
0.7137Epoch 1 took 6.70 seconds
32/32 [==============================] - 7s 161ms/step - loss: 0.7394 -
accuracy: 0.7146 - val_loss: 0.2915 - val_accuracy: 0.8996
Epoch 2/10
31/32 [============================>.] - ETA: 0s - loss: 0.1651 - accuracy:
0.9496Epoch 2 took 5.00 seconds
32/32 [==============================] - 5s 156ms/step - loss: 0.1646 -
accuracy: 0.9497 - val_loss: 0.1539 - val_accuracy: 0.9518
Epoch 3/10
31/32 [============================>.] - ETA: 0s - loss: 0.0711 - accuracy:
0.9788Epoch 3 took 4.81 seconds
32/32 [==============================] - 5s 150ms/step - loss: 0.0712 -
accuracy: 0.9789 - val_loss: 0.0780 - val_accuracy: 0.9839
Epoch 4/10
31/32 [============================>.] - ETA: 0s - loss: 0.0615 - accuracy:
0.9839Epoch 4 took 4.81 seconds
32/32 [==============================] - 5s 150ms/step - loss: 0.0620 -
accuracy: 0.9839 - val_loss: 0.0809 - val_accuracy: 0.9799
Epoch 5/10
31/32 [============================>.] - ETA: 0s - loss: 0.0372 - accuracy:
0.9899Epoch 5 took 4.84 seconds
32/32 [==============================] - 5s 151ms/step - loss: 0.0371 -
accuracy: 0.9899 - val_loss: 0.0806 - val_accuracy: 0.9880
Epoch 6/10
31/32 [============================>.] - ETA: 0s - loss: 0.0176 - accuracy:
0.9950Epoch 6 took 5.14 seconds
32/32 [==============================] - 5s 160ms/step - loss: 0.0176 -
accuracy: 0.9950 - val_loss: 0.0515 - val_accuracy: 0.9799
Epoch 7/10
31/32 [============================>.] - ETA: 0s - loss: 0.0140 - accuracy:
0.9970Epoch 7 took 4.97 seconds
```

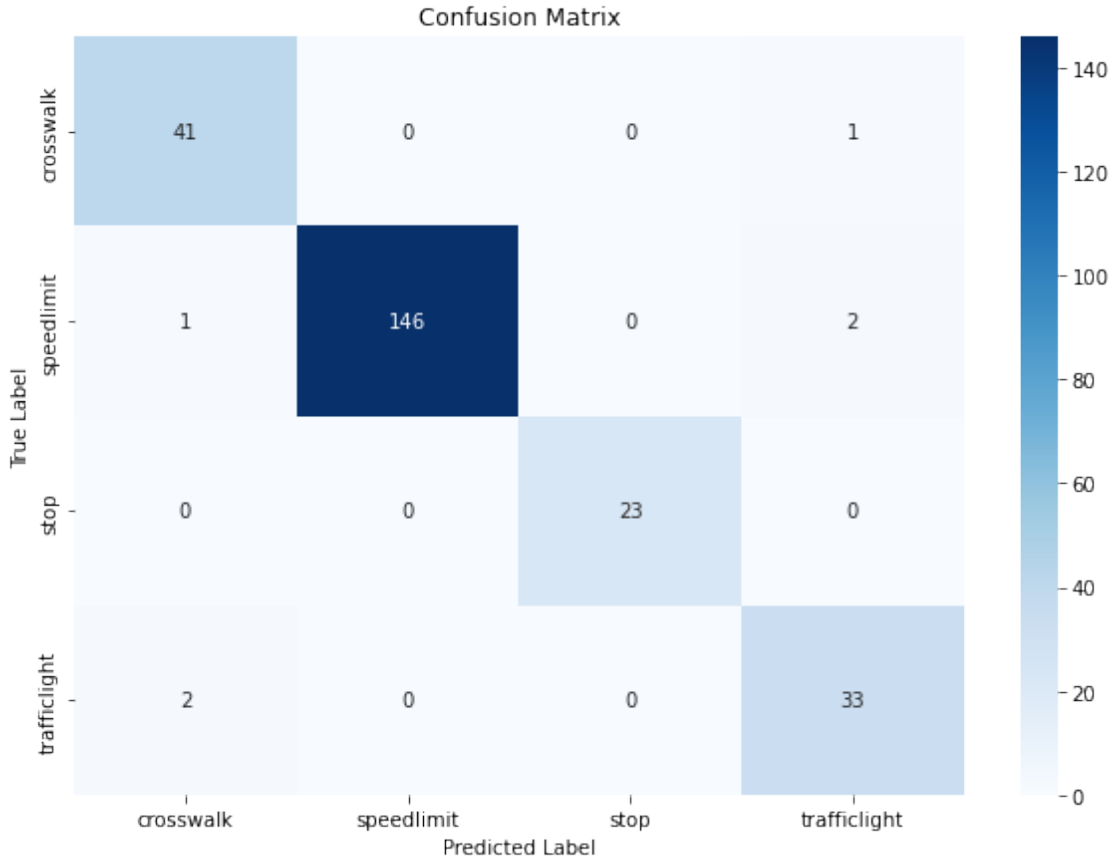
```
32/32 [==============================] - 5s 155ms/step - loss: 0.0141 -
accuracy: 0.9970 - val_loss: 0.0768 - val_accuracy: 0.9799
Epoch 8/10
31/32 [============================>.] - ETA: 0s - loss: 0.0115 - accuracy:
0.9970Epoch 8 took 4.81 seconds
32/32 [==============================] - 5s 150ms/step - loss: 0.0115 -
accuracy: 0.9970 - val_loss: 0.0738 - val_accuracy: 0.9880
Epoch 9/10
31/32 [============================>.] - ETA: 0s - loss: 0.0130 - accuracy:
0.9950Epoch 9 took 5.04 seconds
32/32 [==============================] - 5s 157ms/step - loss: 0.0130 -
accuracy: 0.9950 - val_loss: 0.0758 - val_accuracy: 0.9839
Epoch 10/10
31/32 [============================>.] - ETA: 0s - loss: 0.0085 - accuracy:
0.9990Epoch 10 took 5.00 seconds
32/32 [==============================] - 5s 156ms/step - loss: 0.0084 -
accuracy: 0.9990 - val_loss: 0.0708 - val_accuracy: 0.9759
Training time:  51.497206926345825
```

[16]:
```
#Plottin training time per epoch
plt.plot(range(1, len(times_per_epoch) + 1), times_per_epoch, marker='o')
plt.xlabel('Epoch')
plt.ylabel('Time (seconds)')
plt.title('Time Taken Per Epoch')
plt.show()
```



[17]:
```
# Evaluate the model on the test set
test_loss, test_acc = model.evaluate(X_test, y_test)
print(f'Test Accuracy: {test_acc:.4f}')

# Predict the labels for the test set
y_pred = model.predict(X_test)
y_pred_classes = np.argmax(y_pred, axis=1)  # Convert predictions to class␣
 ↪labels
y_true = np.argmax(y_test, axis=1)  # Convert one-hot encoded labels to class␣
 ↪labels

# Calculate the confusion matrix
cm = confusion_matrix(y_true, y_pred_classes)

# Plot the confusion matrix
plt.figure(figsize=(10, 7))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=lb.classes_,␣
 ↪yticklabels=lb.classes_)
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.title('Confusion Matrix')
plt.show()

# Calculate the accuracy score
acc_score = accuracy_score(y_true, y_pred_classes)
print(f'Accuracy Score: {acc_score:.4f}')

# Generate a classification report
class_report = classification_report(y_true, y_pred_classes, target_names=lb.
 ↪classes_)
print('Classification Report:')
print(class_report)
```

```
8/8 [==============================] - 0s 40ms/step - loss: 0.0708 - accuracy:
0.9759
Test Accuracy: 0.9759
8/8 [==============================] - 0s 42ms/step
```




```
Accuracy Score: 0.9759
Classification Report:
              precision    recall  f1-score   support

   crosswalk       0.93      0.98      0.95        42
  speedlimit       1.00      0.98      0.99       149
        stop       1.00      1.00      1.00        23
trafficlight       0.92      0.94      0.93        35

    accuracy                           0.98       249
   macro avg       0.96      0.97      0.97       249
weighted avg       0.98      0.98      0.98       249
```

1.1.8 Analysis:

Accuracy Score:

Overall Accuracy: 0.9759 (97.59%)

This high accuracy indicates that the model correctly classifies almost 98% of the road signs. This

is a strong indicator of the model's effectiveness, particularly in a well-defined problem space with balanced classes.

Precision: Measures how many of the predicted positive cases are actually positive. High precision for all classes indicates that false positives are minimal. For example, the model has a perfect precision of 1.00 for "Stop," meaning every predicted "Stop" sign is correct.

Recall: Measures how many of the actual positive cases are correctly identified. The recall is high across the board, especially for the "Stop" class, which shows the model is good at detecting even less frequent classes.

F1-Score: The harmonic mean of precision and recall. The F1-scores are high for all classes, indicating a good balance between precision and recall. The model performs well in distinguishing between different road signs.

1.1.9 Confusion Matrix

Analysis:

Crosswalk: Misclassified as "Trafficlight" once. This suggests some confusion between these classes. This could be due to similarities in visual features or limited training examples.

Speedlimit: Slightly misclassified as "Trafficlight" (2 times) and "Crosswalk" (once). These errors are minimal but indicate potential overlap or insufficient differentiation between some classes.

Stop: Perfect classification. The model accurately identifies all "Stop" signs.

Trafficlight: Misclassified as "Crosswalk" twice. This could be due to overlapping features or inadequate representation of "Trafficlight" signs in the dataset.

1.1.10 Overall Performance:

The confusion matrix shows that most classes are correctly classified with only a few misclassifications. The errors are concentrated between specific classes, which could be addressed by improving class separation or increasing data diversity.

1.1.11 Success Criteria Evaluation:

Accuracy:

Criterion: 95% **Result:** 97.59% **Analysis:** The model exceeds the accuracy criterion, suggesting it performs exceptionally well on the test data.

Precision, Recall, and F1-Score:

Criterion: Precision, Recall, and F1-Score 0.90 for all classes.

Result:

Precision: All classes meet or exceed 0.92.

Recall: All classes meet or exceed 0.94.

F1-Score: All classes meet or exceed 0.93.

Analysis: The model performs well according to these metrics, indicating it is effective across different classes and achieves a good balance between precision and recall.

Confusion Matrix:

Criterion: Minimal misclassifications.

Analysis: The matrix shows some misclassifications but not at a scale that significantly impacts overall performance. The errors are concentrated and can be further analyzed to improve class differentiation.

[18]:
```
# Save the model in the Keras format
model.save('road_sign_classifier.keras')
```

To use the model for inference, we should prepare your input image as follows:

Load the Image: Use an image processing library like PIL to load the image. Convert the image to RGB if it is not already in that format.

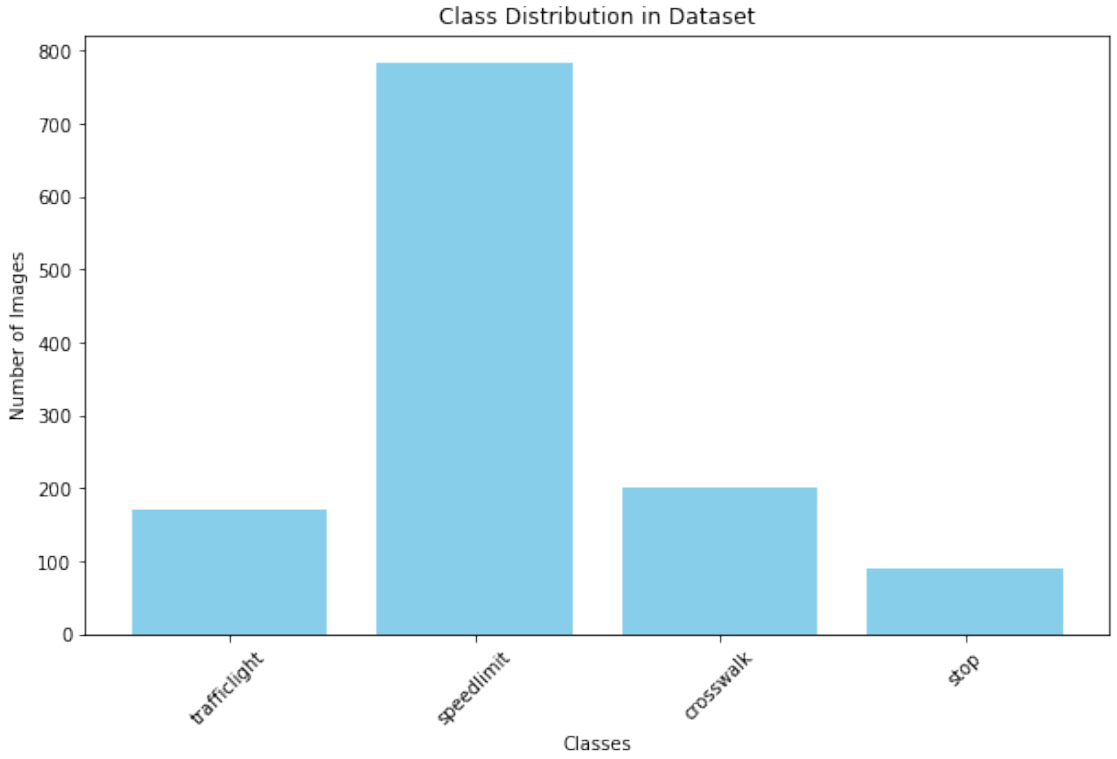
Resize the Image: Resize or crop the image to 64x64 pixels to match the input shape expected by the model.

Normalize the Image: Scale pixel values to a range between 0 and 1.

Prepare the Image for Model Prediction: Convert the image to a NumPy array with shape (64, 64, 3). Add an additional dimension to represent the batch size: (1, 64, 64, 3).

1.2 Analyse and improve the model

[19]:
```
from collections import Counter
from tensorflow.keras.preprocessing.image import ImageDataGenerator

# Count occurrences of each label
label_counts = Counter(cropped_labels)

# Plot class distribution
plt.figure(figsize=(10, 6))
plt.bar(label_counts.keys(), label_counts.values(), color='skyblue')
plt.xlabel('Classes')
plt.ylabel('Number of Images')
plt.title('Class Distribution in Dataset')
plt.xticks(rotation=45)
plt.show()
```



Even though the model perfromed significantly good but its clear that the speedlimit class has the highest distribution, so I will try to balance the classes using data augmentation and would try to populate the supperessed classes. Then I will compare the results of the model with the model without data augmentation.

[20]:
```
# Class labels
class_labels = ['crosswalk', 'speedlimit', 'stop', 'trafficlight']
```

[21]:
```
# Original class distribution
y_train_int = np.argmax(y_train, axis=1)  # Convert one-hot encoded labels to␣
 ↪integer labels
original_counts = np.bincount(y_train_int)
```

[22]:
```
# Define the target number of samples (maximum count across classes)
target_count = max(original_counts)

# Initialize lists for the balanced dataset
X_train_balanced = []
y_train_balanced = []
```

[23]:
```
# Data augmentation generator
datagen = ImageDataGenerator(
    rotation_range=20,
```

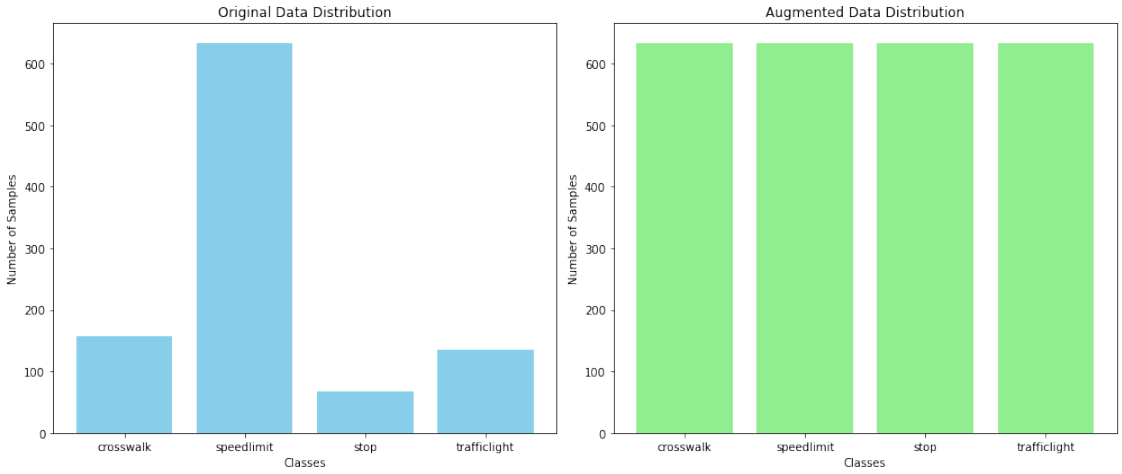
```
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,
    fill_mode='nearest'
)
```

[24]:
```
# Perform class-wise augmentation, except for 'speedlimit'
for class_idx, class_count in enumerate(original_counts):
    # Skip augmenting 'speedlimit'
    if class_labels[class_idx] == 'speedlimit':
        X_train_balanced.extend(X_train[y_train_int == class_idx])
        y_train_balanced.extend(y_train[y_train_int == class_idx])
        continue

    # Filter data for the current class
    X_cls = X_train[y_train_int == class_idx]
    y_cls = y_train[y_train_int == class_idx]

    # Add existing samples
    X_train_balanced.extend(X_cls)
    y_train_balanced.extend(y_cls)

    # Generate augmented images until the class has the target number of samples
    for i in range(target_count - len(X_cls)):
        X_aug = datagen.random_transform(X_cls[i % len(X_cls)])
        X_train_balanced.append(X_aug)
        y_train_balanced.append(y_cls[i % len(y_cls)])
```

[25]:
```
# Convert to NumPy arrays
X_train_balanced = np.array(X_train_balanced)
y_train_balanced = np.array(y_train_balanced)

# Class distribution after augmentation
y_train_balanced_int = np.argmax(y_train_balanced, axis=1)
balanced_counts = np.bincount(y_train_balanced_int)
```

[26]:
```
# Plotting the distribution
fig, ax = plt.subplots(1, 2, figsize=(14, 6))

# Original distribution
ax[0].bar(class_labels, original_counts, color='skyblue')
ax[0].set_title('Original Data Distribution')
ax[0].set_xlabel('Classes')
ax[0].set_ylabel('Number of Samples')
```

```
# Augmented distribution
ax[1].bar(class_labels, balanced_counts, color='lightgreen')
ax[1].set_title('Augmented Data Distribution')
ax[1].set_xlabel('Classes')
ax[1].set_ylabel('Number of Samples')

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



Now the classes seems to be more ballanced

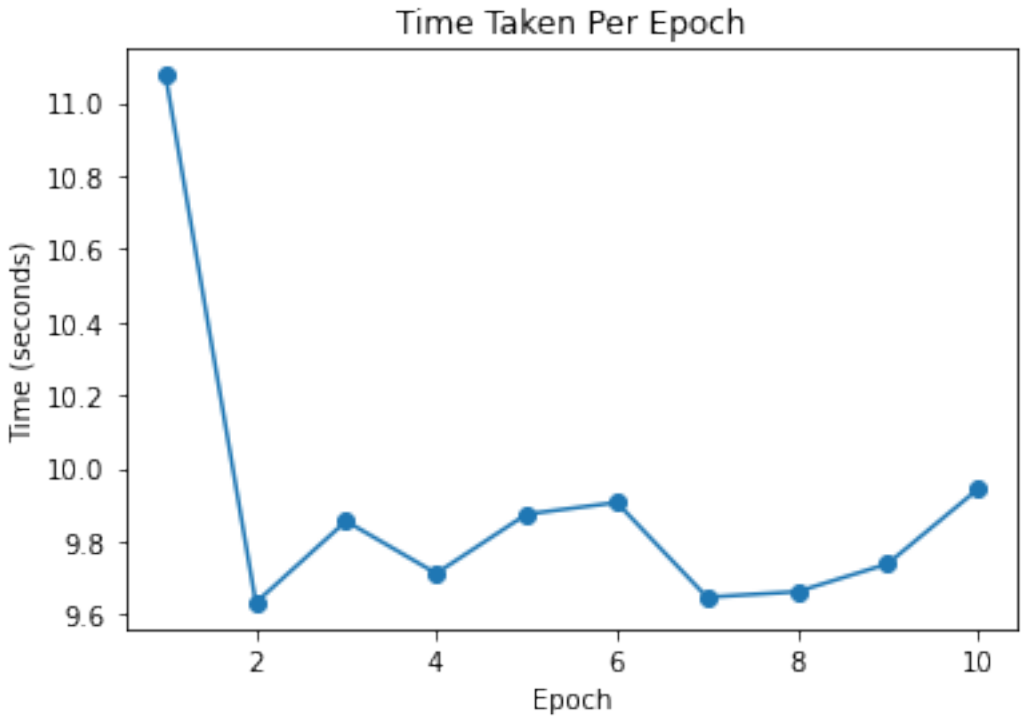
[27]:
```
import tensorflow as tf
#Model training
# Use the provided create_model function
input_shape = (64, 64, 3)  # Image dimensions with 3 channels
model_aug = create_model(input_shape)
model_aug.compile(optimizer='adam', loss='categorical_crossentropy',␣
 ↪metrics=['accuracy'])

# Start profiling
tf.profiler.experimental.start('logdir')

# Instantiate the callback
time_callback = TimeHistory()

# Train the model with augmented data
start_time_aug = time.time()
history_aug = model_aug.fit(X_train_balanced, y_train_balanced, epochs=10,␣
 ↪validation_split=0.2, callbacks=[time_callback])
end_time_aug = time.time()
elapsed_time_aug = end_time_aug - start_time_aug
```

```
print("Training time: ",elapsed_time_aug)

# Stop profiling
tf.profiler.experimental.stop()

# Access the recorded times per epoch
times_per_epoch = time_callback.times
```

```
Epoch 1/10
64/64 [==============================] - ETA: 0s - loss: 0.4399 - accuracy:
0.8245Epoch 1 took 11.07 seconds
64/64 [==============================] - 11s 153ms/step - loss: 0.4399 -
accuracy: 0.8245 - val_loss: 1.4102 - val_accuracy: 0.5433
Epoch 2/10
64/64 [==============================] - ETA: 0s - loss: 0.0807 - accuracy:
0.9739Epoch 2 took 9.63 seconds
64/64 [==============================] - 10s 150ms/step - loss: 0.0807 -
accuracy: 0.9739 - val_loss: 0.7629 - val_accuracy: 0.7343
Epoch 3/10
64/64 [==============================] - ETA: 0s - loss: 0.0390 - accuracy:
0.9882Epoch 3 took 9.85 seconds
64/64 [==============================] - 10s 154ms/step - loss: 0.0390 -
accuracy: 0.9882 - val_loss: 0.1592 - val_accuracy: 0.9567
Epoch 4/10
64/64 [==============================] - ETA: 0s - loss: 0.0314 - accuracy:
0.9901Epoch 4 took 9.71 seconds
64/64 [==============================] - 10s 152ms/step - loss: 0.0314 -
accuracy: 0.9901 - val_loss: 1.2501 - val_accuracy: 0.6280
Epoch 5/10
64/64 [==============================] - ETA: 0s - loss: 0.0195 - accuracy:
0.9946Epoch 5 took 9.87 seconds
64/64 [==============================] - 10s 154ms/step - loss: 0.0195 -
accuracy: 0.9946 - val_loss: 0.1660 - val_accuracy: 0.9705
Epoch 6/10
64/64 [==============================] - ETA: 0s - loss: 0.0291 - accuracy:
0.9926Epoch 6 took 9.91 seconds
64/64 [==============================] - 10s 155ms/step - loss: 0.0291 -
accuracy: 0.9926 - val_loss: 0.8164 - val_accuracy: 0.7992
Epoch 7/10
64/64 [==============================] - ETA: 0s - loss: 0.0131 - accuracy:
0.9956Epoch 7 took 9.65 seconds
64/64 [==============================] - 10s 151ms/step - loss: 0.0131 -
accuracy: 0.9956 - val_loss: 0.4275 - val_accuracy: 0.8839
Epoch 8/10
64/64 [==============================] - ETA: 0s - loss: 0.0140 - accuracy:
0.9951Epoch 8 took 9.66 seconds
64/64 [==============================] - 10s 151ms/step - loss: 0.0140 -
accuracy: 0.9951 - val_loss: 0.7964 - val_accuracy: 0.8346
```

```
Epoch 9/10
64/64 [==============================] - ETA: 0s - loss: 0.0149 - accuracy:
0.9951Epoch 9 took 9.74 seconds
64/64 [==============================] - 10s 152ms/step - loss: 0.0149 -
accuracy: 0.9951 - val_loss: 0.6309 - val_accuracy: 0.8445
Epoch 10/10
64/64 [==============================] - ETA: 0s - loss: 0.0062 - accuracy:
0.9985Epoch 10 took 9.94 seconds
64/64 [==============================] - 10s 155ms/step - loss: 0.0062 -
accuracy: 0.9985 - val_loss: 0.5681 - val_accuracy: 0.8642
Training time:  99.15299892425537
```

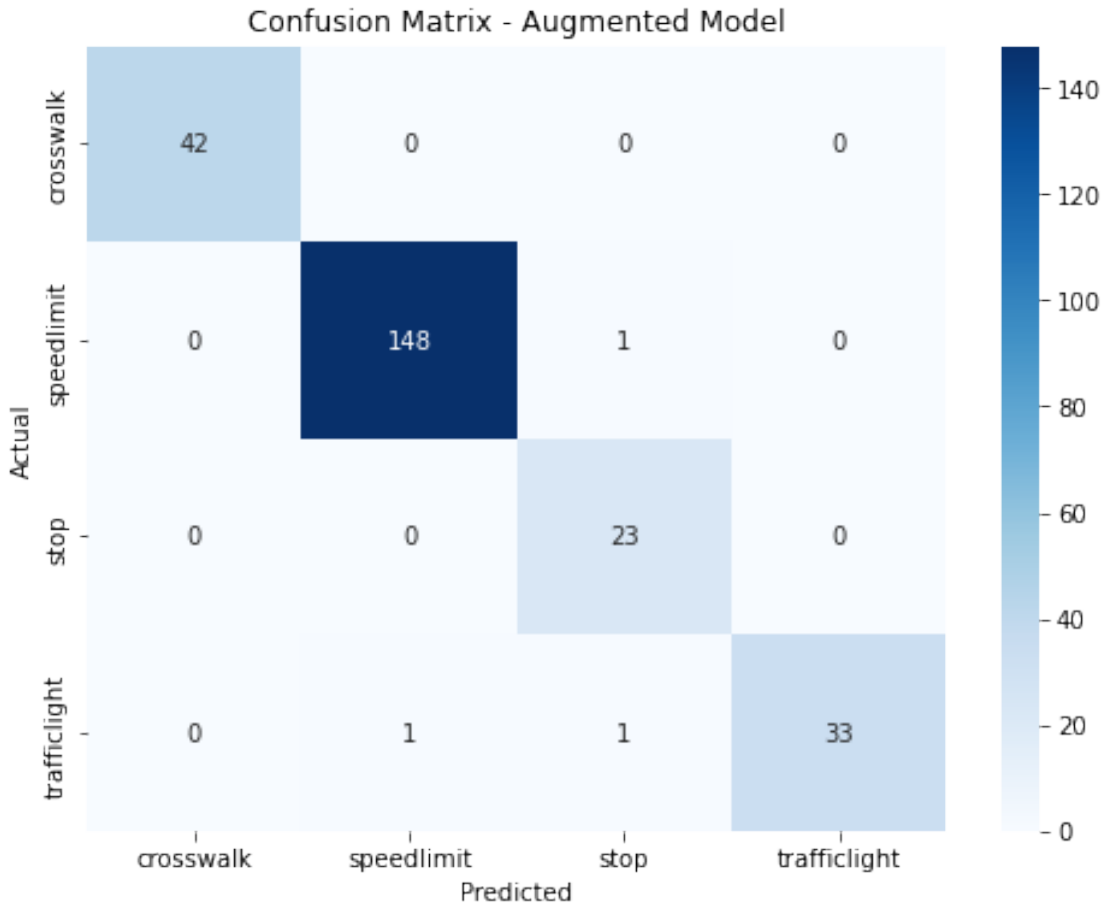
[28]:
```
#Plottin training time per epoch
plt.plot(range(1, len(times_per_epoch) + 1), times_per_epoch, marker='o')
plt.xlabel('Epoch')
plt.ylabel('Time (seconds)')
plt.title('Time Taken Per Epoch')
plt.show()
```



[29]:
```
# Evaluate the model on the test set
y_pred_aug = model_aug.predict(X_test)
y_pred_aug_classes = np.argmax(y_pred_aug, axis=1)
y_test_classes = np.argmax(y_test, axis=1)
```

```
# Classification report for the augmented model
print("Classification Report for Augmented Model:\n")
print(classification_report(y_test_classes, y_pred_aug_classes,␣
 ↪target_names=class_labels))

# Confusion matrix for the augmented model
conf_matrix_aug = confusion_matrix(y_test_classes, y_pred_aug_classes)

# Plotting the confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(conf_matrix_aug, annot=True, fmt='d', cmap='Blues',␣
 ↪xticklabels=class_labels, yticklabels=class_labels)
plt.title('Confusion Matrix - Augmented Model')
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.show()
```

```
8/8 [==============================] - 0s 39ms/step
Classification Report for Augmented Model:

              precision    recall  f1-score   support

   crosswalk       1.00      1.00      1.00        42
  speedlimit       0.99      0.99      0.99       149
        stop       0.92      1.00      0.96        23
trafficlight       1.00      0.94      0.97        35

    accuracy                           0.99       249
   macro avg       0.98      0.98      0.98       249
weighted avg       0.99      0.99      0.99       249
```



[30]:
```
# Calculate the accuracy score
acc_score = accuracy_score(y_test_classes, y_pred_aug_classes)
print(f'Accuracy Score OF augmented model: {acc_score:.4f}')
```

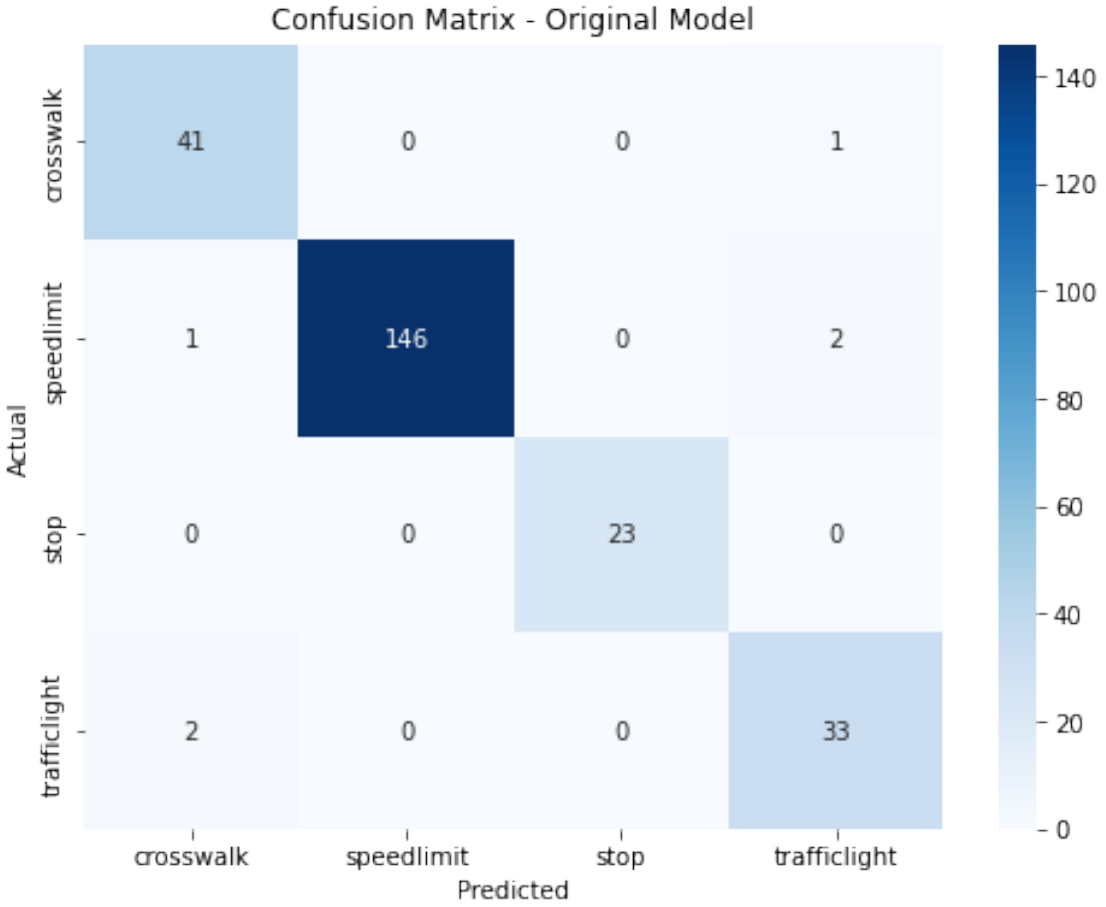
```
Accuracy Score OF augmented model: 0.9880
```

[31]:
```
# Compare with the original model
y_pred_orig = model.predict(X_test)
y_pred_orig_classes = np.argmax(y_pred_orig, axis=1)
```

```
8/8 [==============================] - 0s 40ms/step
```

[32]:
```
# Confusion matrix for the original model
conf_matrix_orig = confusion_matrix(y_test_classes, y_pred_orig_classes)

# Plotting the confusion matrix for the original model
plt.figure(figsize=(8, 6))
sns.heatmap(conf_matrix_orig, annot=True, fmt='d', cmap='Blues',␣
 ↪xticklabels=class_labels, yticklabels=class_labels)
```

```
plt.title('Confusion Matrix - Original Model')
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.show()
```



[33]:
```
# Compare the classification reports as well
print("Classification Report for Original Model:\n")
print(classification_report(y_test_classes, y_pred_orig_classes,␣
 ↪target_names=class_labels))
```

```
Classification Report for Original Model:

              precision    recall  f1-score   support

   crosswalk       0.93      0.98      0.95        42
  speedlimit       1.00      0.98      0.99       149
        stop       1.00      1.00      1.00        23
trafficlight       0.92      0.94      0.93        35
```

accuracy			0.98	249
macro avg	0.96	0.97	0.97	249
weighted avg	0.98	0.98	0.98	249

1.3 Performance Analysis After Data Augmentation

1.4 Accuracy Score:

Original Model Accuracy: 97.59%

Augmented Model Accuracy: 98.80%

Performance Gain:

Accuracy improvement = 98.80% - 97.59% = 1.21%

Analysis: The accuracy of the augmented model improved by 1.21 percentage points. This indicates that data augmentation has positively impacted the model's ability to generalize better on unseen data.

1.4.1 Classification Report:

Analysis:

Precision and Recall: The precision and recall have generally improved or remained high across all classes.

Crosswalk: Perfect precision and recall (1.00) for "Crosswalk" indicates no misclassifications for this class.

Speedlimit: Slight decrease in precision but high recall (0.99) for the "Speedlimit" class suggests very few false negatives.

Stop: While precision dropped slightly to 0.92, recall improved to 1.00, indicating that the model correctly identified all "Stop" signs without missing any.

Trafficlight: Precision improved to 1.00, with a recall of 0.94. The F1-score also improved, showing better performance in classifying "Trafficlight" signs.

Overall Improvement:

The macro average precision, recall, and F1-score improved from 0.96, 0.97, and 0.97 to 0.98, 0.98, and 0.98 respectively. This shows a balanced improvement across all metrics.

1.4.2 Confusion Matrix

Analysis:

Crosswalk: Perfect classification with no misclassifications.

Speedlimit: Only 1 misclassification each as "Crosswalk" and "Stop." This represents a significant improvement from the previous results.

Stop: No misclassifications, indicating that all "Stop" signs were correctly classified.

Trafficlight: A small number of misclassifications into “Speedlimit” and “Stop” compared to the previous model, indicating enhanced performance.

1.4.3 Training Time:

Without Augmentation: 51.50 seconds With Augmentation: 99.15 seconds

Analysis:

The training time of the augmented model is approximately 1.9 times longer than the non-augmented model. This is expected due to the increased computational load from the augmented data.

1.4.4 Training time per epoch

Initial Training Time:

Actual Data Graph: The first epoch took approximately 6.75 seconds.

Augmented Data Graph: The first epoch took approximately 11 seconds.

The augmented model took significantly longer for the first epoch compared to the actual data model.

Subsequent Epochs:

Actual Data Graph: After the first epoch, the time per epoch stabilizes between 4.75 and 5.25 seconds.

Augmented Data Graph: The time per epoch stabilizes between 9.6 and 10.2 seconds.

The augmented model consistently takes more time per epoch compared to the actual data model.

Consistency Across Epochs:

Actual Data Graph: There is some variability in training time, particularly around epoch 5 and 6, where there is a slight increase in time.

Augmented Data Graph: The training time is more consistent, but still shows small fluctuations, particularly a slight increase around epoch 10.

Overall Training Time:

The augmented model takes almost double the time per epoch compared to the actual data model. This increase in time is likely due to the additional computational overhead introduced by data augmentation operations.

In summary, the augmented model shows a significant increase in training time per epoch compared to the actual data model, which is expected due to the extra processing required for data augmentation. However, this added time could potentially lead to better model performance, depending on the effectiveness of the augmentation techniques used.

1.4.5 Summary of Performance Gain:

Accuracy: The model’s accuracy increased by 1.21 percentage points, demonstrating a notable improvement in overall classification performance.

Classification Metrics: Precision, recall, and F1-scores for most classes improved or remained high, reflecting a more robust model.

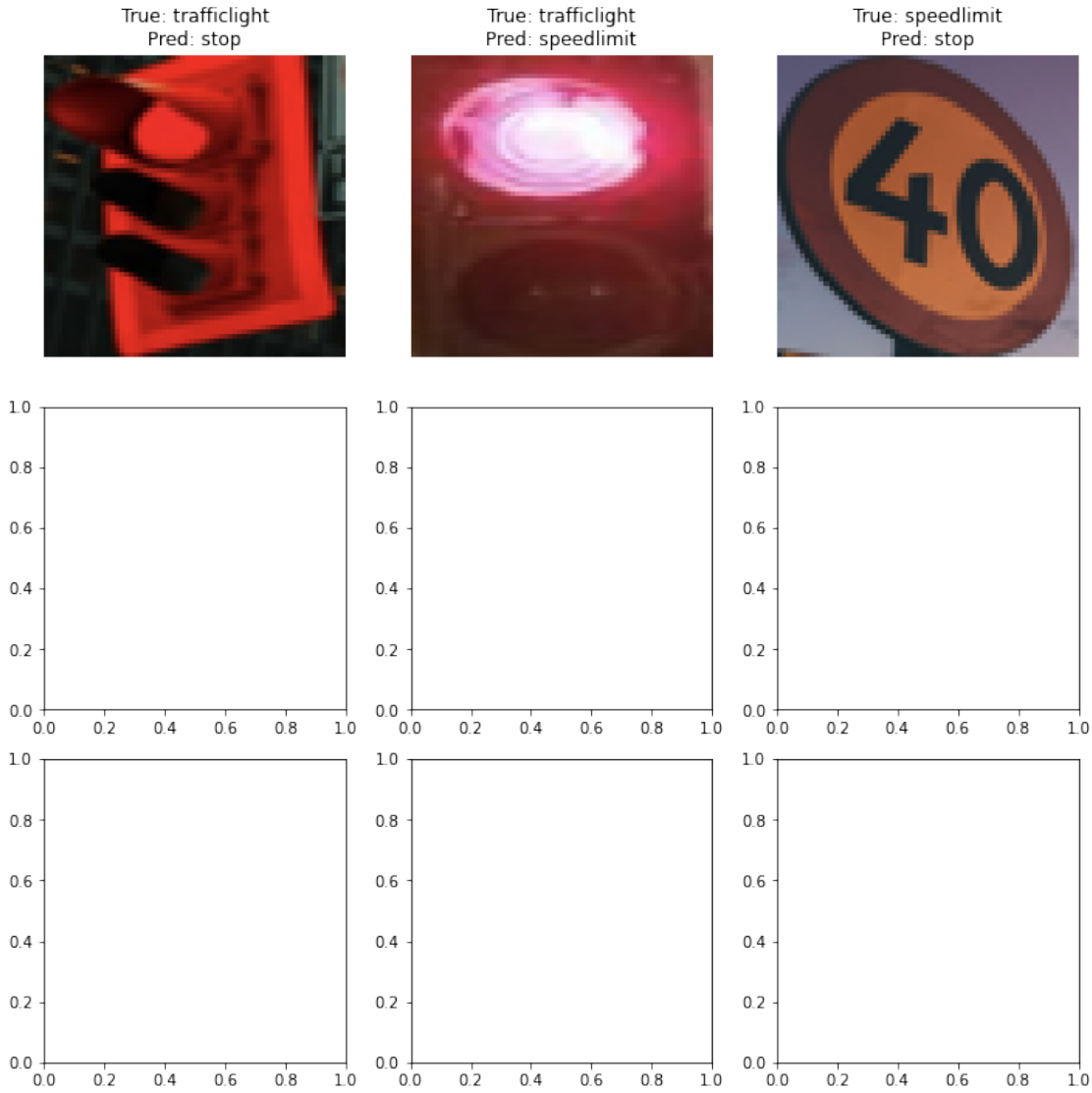
Confusion Matrix: Reduced misclassifications across all classes indicate that data augmentation effectively improved the model's ability to distinguish between classes.

Training Time: Increased training time due to augmentation is a trade-off for improved performance.

The application of data augmentation has led to a significant improvement in model performance. The accuracy, precision, recall, and F1-score metrics all show positive gains. Although the training time has increased, the benefits in classification accuracy and reduced errors make the augmentation a valuable strategy.

[118]:
```
# Identify misclassified images
misclassified_indices = np.where(y_pred_aug_classes != y_test_classes)[0]
```

[119]:
```
# Visualize some misclassified images
fig, axes = plt.subplots(3, 3, figsize=(10, 10))
axes = axes.ravel()

for i in range(3):  # Display 9 misclassified images
    index = misclassified_indices[i]
    axes[i].imshow(X_test[index])
    axes[i].set_title(f"True: {class_labels[y_test_classes[index]]}\nPred:␣
 ↪{class_labels[y_pred_aug_classes[index]]}")
    axes[i].axis('off')

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



1.4.6 Common Characteristics:

Blurred Images:

The first image (True: trafficlight, Pred: stop) and the second image (True: trafficlight, Pred: speedlimit) appear to be blurry or unclear. Blurred images can make it difficult for the model to accurately identify key features, leading to incorrect predictions.

Color Similarities:

The first image (True: trafficlight, Pred: stop) and the second image (True: trafficlight, Pred: speedlimit) have red hues that might be confusing the model. The model may be mistaking the color and shape for a different class (e.g., red light as a “stop” sign or as part of the “speedlimit” sign).

Ambiguity in Visual Features:

The third image (True: speedlimit, Pred: stop) shows a sign with the number "40," which may not be as easily distinguishable from a "stop" sign in certain contexts. The round shape of the speed limit sign might also be contributing to the confusion, as it can resemble the circular shape of a "stop" sign.

1.4.7 Plan to Improve Model Performance:

Data Augmentation:

Introduce more data augmentation techniques that specifically address blurring and color variations. This could involve: Applying Gaussian blur to training images to make the model more robust to blurring. Adjusting the brightness and contrast to help the model learn to distinguish between similar colors under different lighting conditions.

Class Imbalance Handling:

Ensure that the model is not overfitting to certain classes due to class imbalance. This can be done by using techniques such as oversampling the minority classes or using class weights during training.

Feature Engineering:

Enhance the input data by including more detailed features that the model can use to distinguish between similar classes. For instance, emphasizing shape features could help the model better distinguish between circular "stop" signs and circular "speed limit" signs.

Model Architecture Adjustments:

Refining the model architecture can hep to improve its ability to capture fine details. For example, increasing the resolution of input images or using a more powerful model like a deeper convolutional neural network (CNN) could help the model better capture small, but important, features.

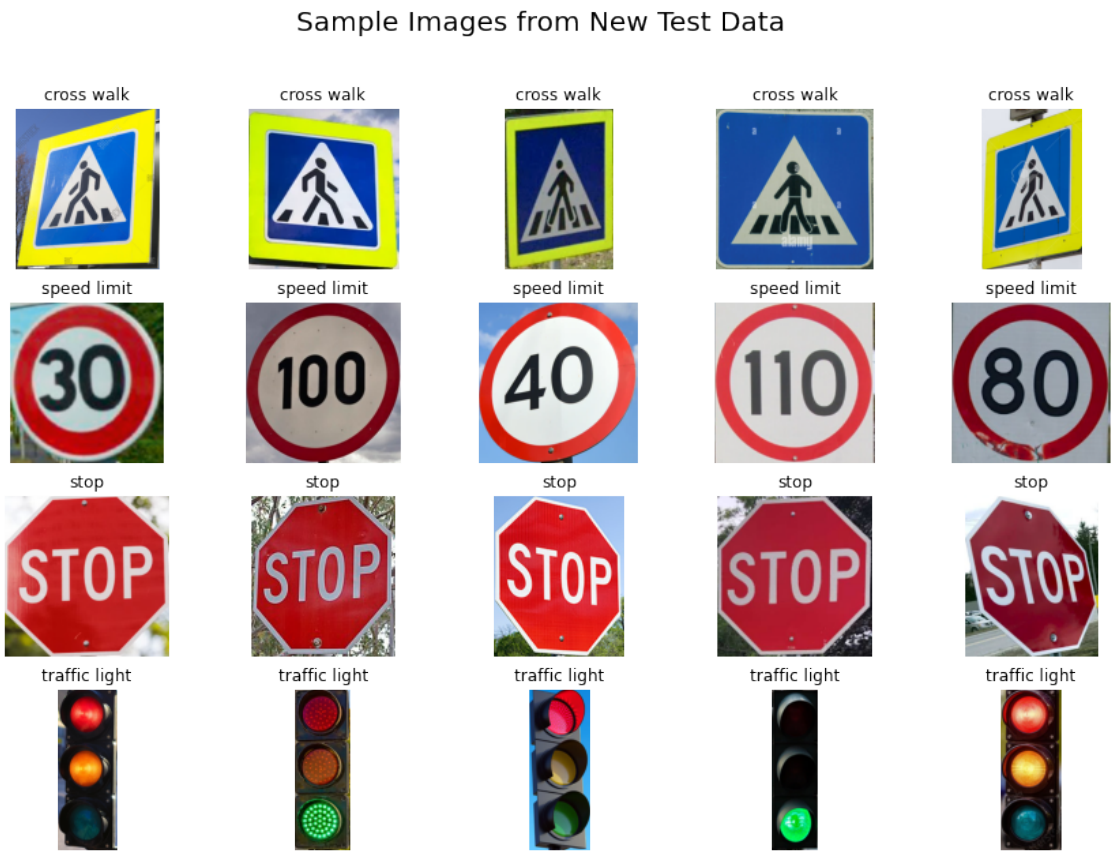
1.5 Improve model generalisability across domains**

[37]:
```
# Actual data
cropped_labels = np.array(cropped_labels)

# Define a function to visualize 5 images per class
def visualize_samples_by_class(cropped_images, cropped_labels, num_samples=5):
    # Get the unique classes
    unique_classes = np.unique(cropped_labels)
    num_classes = len(unique_classes)

    # Create a figure with subplots for each class
    plt.figure(figsize=(15, 10))

    for i, cls in enumerate(unique_classes):
        # Find the indices for the current class
        indices = np.where(cropped_labels == cls)[0]
```

```
        # Display up to num_samples images per class
        for j in range(min(num_samples, len(indices))):
            plt.subplot(num_classes, num_samples, i * num_samples + j + 1)
            plt.imshow(cropped_images[indices[j]])
            plt.title(f'{cls} #{j+1}')
            plt.axis('off')

    plt.suptitle("Sample Images from Each Class", size=20)
    plt.show()

# Visualize 5 images per class
visualize_samples_by_class(cropped_images, cropped_labels, num_samples=5)
```

[38]:
```
# New data samples
test_data_dir = r"Test Data"

# Class directories inside the Test Data folder
class_dirs = ['cross walk', 'speed limit', 'stop', 'traffic light']

# Dictionary to store images by class
```

```python
test_images = {class_name: [] for class_name in class_dirs}

# Load 5 images from each class
for class_name in class_dirs:
    class_path = os.path.join(test_data_dir, class_name)
    image_files = os.listdir(class_path)[:5]  # Take only the first 5 images
    for img_file in image_files:
        img_path = os.path.join(class_path, img_file)
        image = Image.open(img_path)
        test_images[class_name].append(image)

# Display the images
fig, axes = plt.subplots(len(class_dirs), 5, figsize=(15, 10))

for i, class_name in enumerate(class_dirs):
    for j in range(5):
        axes[i, j].imshow(test_images[class_name][j])
        axes[i, j].set_title(class_name)
        axes[i, j].axis('off')

plt.suptitle("Sample Images from New Test Data", size=20)
plt.show()
```



1.5.1 Difference between Images

Crosswalk

The crosswalk sign is bordered by a bright yellow frame.

The sign is more zoomed in and has a yellow border.

The sign appears to be taken from a different angle and has a more vibrant color.

Stop Sign

The sign is clear, with a different background, and taken from a different angle.

The sign is clear and taken in a bright environment.

The sign is clearer, with a more defined background.

Traffic Light

The light is clearer, showing red, yellow and green

Picture are clear with a well-defined background.

These images were taken using mobile phone camera

[45]:
```
#Loading new data
# Path to the Test Data folder
test_data_dir = r"C:/Users/Hp/Desktop/Deep Learning/Assignment 2/Test Data"

# Class directories inside the Test Data folder
class_dirs = ['cross walk', 'speed limit', 'stop', 'traffic light']
```

[46]:
```
# Dictionary to store images and labels
test_images = []
test_labels = []

# Mapping directory names to label indices (you can map them as you see fit)
label_map = {class_name: i for i, class_name in enumerate(class_dirs)}
```

[47]:
```
# Load images and create labels
for class_name in class_dirs:
    class_path = os.path.join(test_data_dir, class_name)
    image_files = os.listdir(class_path)
    label = label_map[class_name]
    for img_file in image_files:
        img_path = os.path.join(class_path, img_file)
        image = Image.open(img_path)

        # Preprocessing Steps
```

```
        image = image.convert('RGB')                # Convert to RGB if not␣
 ↪already
        image = image.resize((64, 64))              # Resize to 64x64 pixels
        image_array = np.array(image) / 255.0       # Normalize image to range␣
 ↪[0, 1]
        image_array = np.expand_dims(image_array, axis=0)  # Add batch dimension

        test_images.append(image_array)
        test_labels.append(label)

# Convert lists to numpy arrays and remove extra dimensions
test_images = np.vstack(test_images)  # Stack images into a single numpy array
test_labels = np.array(test_labels)
```

[48]:
```
# Load images and create labels
for class_name in class_dirs:
    class_path = os.path.join(test_data_dir, class_name)
    image_files = os.listdir(class_path)
    label = label_map[class_name]

    # Display one image per class
    if image_files:  # Check if the class directory is not empty
        img_file = image_files[0]  # Take the first image file
        img_path = os.path.join(class_path, img_file)
        image = Image.open(img_path)

        # Convert image to display format
        image = image.convert('RGB')  # Convert to RGB if not already
        image = image.resize((256, 256))  # Resize to match model input

        # Plot the image
        plt.figure(figsize=(2, 2))
        plt.imshow(image)
        plt.title(f'Class: {class_name}')
        plt.axis('off')  # Hide axes
        plt.show()
```

Class: speed limit

Class: stop

Class: traffic light

Since I have 2 modles, 1) a model trained on original data 2) a model trained on augmented data

`I will be testing both the models on the new test data`

[49]:
```
# Evaluate using actual model
model_predictions = np.argmax(model.predict(test_images), axis=1)
model_accuracy = np.mean(model_predictions == test_labels)
print(f"Accuracy of model on new test data: {model_accuracy * 100:.2f}%")

# Generate and print classification report for the original model
classification_report_model = classification_report(test_labels,␣
 ↪model_predictions, target_names=class_dirs)
print("Classification Report for Model:")
print(classification_report_model)
```

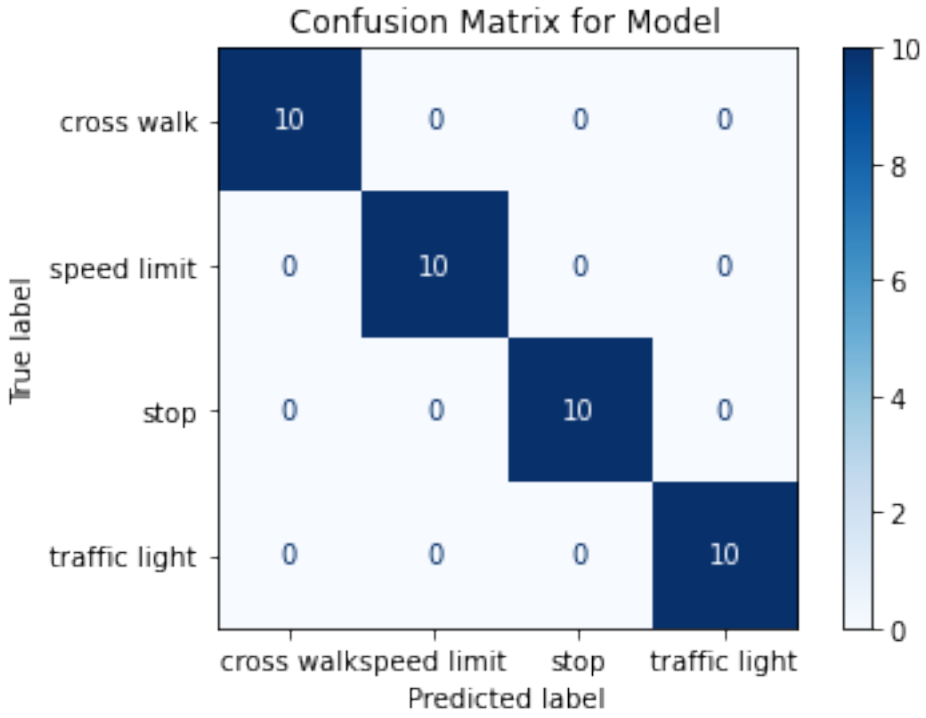
```
2/2 [==============================] - 0s 15ms/step
Accuracy of model on new test data: 100.00%
Classification Report for Model:
              precision    recall  f1-score   support

   cross walk       1.00      1.00      1.00        10
  speed limit       1.00      1.00      1.00        10
         stop       1.00      1.00      1.00        10
traffic light       1.00      1.00      1.00        10

     accuracy                           1.00        40
    macro avg       1.00      1.00      1.00        40
 weighted avg       1.00      1.00      1.00        40
```

[50]:
```
# Evaluate using augmented model
model_aug_predictions = np.argmax(model_aug.predict(test_images), axis=1)
model_aug_accuracy = np.mean(model_aug_predictions == test_labels)
print(f"Accuracy of model_aug on new test data: {model_aug_accuracy * 100:.
 ↪2f}%")

# Generate and print classification report for the augmented model
classification_report_model_aug = classification_report(test_labels,␣
 ↪model_aug_predictions, target_names=class_dirs)
print("Classification Report for Model_Aug:")
print(classification_report_model_aug)
```

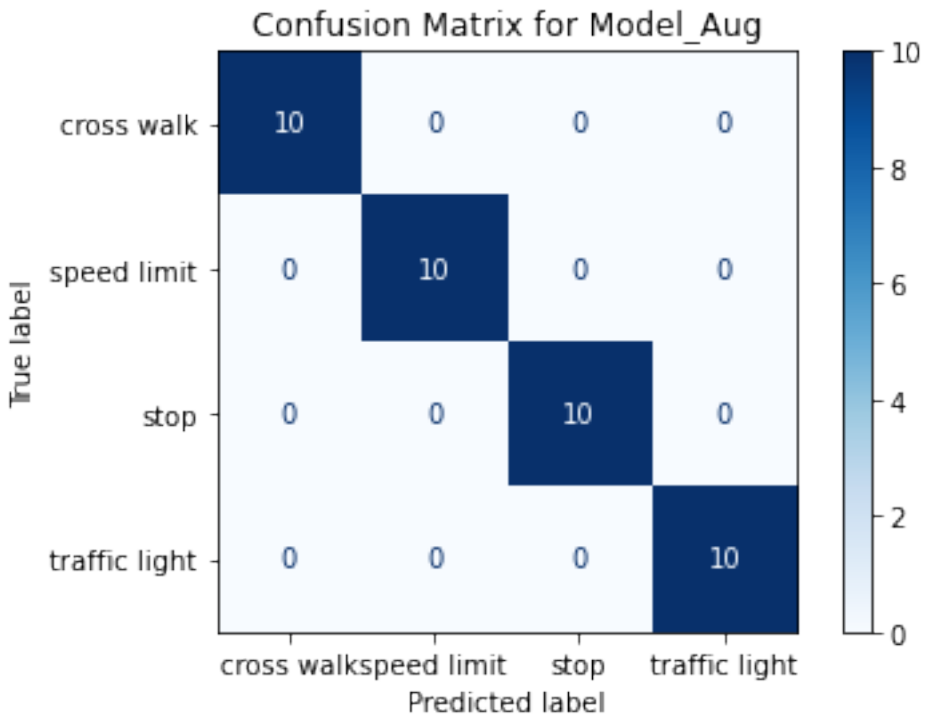
```
2/2 [==============================] - 0s 15ms/step
Accuracy of model_aug on new test data: 100.00%
Classification Report for Model_Aug:
              precision    recall  f1-score   support

   cross walk       1.00      1.00      1.00        10
  speed limit       1.00      1.00      1.00        10
         stop       1.00      1.00      1.00        10
traffic light       1.00      1.00      1.00        10
```

```
              accuracy                           1.00        40
             macro avg       1.00      1.00      1.00        40
          weighted avg       1.00      1.00      1.00        40
```

[51]:
```
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
# Confusion Matrix for the original model
conf_matrix_model = confusion_matrix(test_labels, model_predictions)
disp = ConfusionMatrixDisplay(confusion_matrix=conf_matrix_model,␣
 ↪display_labels=class_dirs)
disp.plot(cmap=plt.cm.Blues)
plt.title('Confusion Matrix for Model')
plt.show()
```



[52]:
```
# Confusion Matrix for the augmented model
conf_matrix_model_aug = confusion_matrix(test_labels, model_aug_predictions)
disp_aug = ConfusionMatrixDisplay(confusion_matrix=conf_matrix_model_aug,␣
 ↪display_labels=class_dirs)
disp_aug.plot(cmap=plt.cm.Blues)
plt.title('Confusion Matrix for Model_Aug')
plt.show()
```



1. Accuracy:

Original Model: 100.00%

Augmented Model: 100.00%

Both models achieved 100% accuracy, meaning they correctly classified all instances in the test dataset.

2. Classification Report:

Precision, Recall, F1-Score for Both Models:

Cross Walk: 1.00

Speed Limit: 1.00

Stop: 1.00

Traffic Light: 1.00

The precision, recall, and F1-score are perfect (1.00) for all classes in both models. This indicates that both models have excellent performance across all the categories.

3. Confusion Matrix:

Original Model Confusion Matrix:

Correct classifications for all classes (no misclassifications).

Augmented Model Confusion Matrix: Also shows correct classifications for all classes (no misclassifications).

Conclusion:

Performance Comparison:

Both models perform identically on the new test data, with no observable differences in accuracy, precision, recall, F1-score, or the confusion matrix.

Effect of Augmentation:

While data augmentation can typically improve the robustness of a model, in this case, both models perform equally well on the test data. The augmentation did not result in a measurable improvement, likely because the original data was already well-suited to the task, or because the test data did not present any additional challenges that would highlight the benefits of augmentation.

The test images were taken from my cell phone and I tried to have pictures from different angles, but no matter what I did, the test images had a better resolution compared to the images in the dataset

1.5.2 Techniques to improve generalizability

Regularization Techniques:

Dropout: Introduce dropout layers in your model to prevent overfitting. Dropout randomly sets a fraction of input units to zero during training, which helps in regularizing the model.

L2 Regularization: Add L2 regularization to the dense layers to penalize large weights, encouraging the model to learn simpler patterns that generalize better.

Class Weighting: If your classes are imbalanced, as seems to be the case with the stop class, you can assign higher weights to the underrepresented classes during training to ensure the model pays more attention to them.

Learning Rate Scheduling: Implement a learning rate scheduler to reduce the learning rate during training. This helps the model converge more smoothly and potentially find a better minimum.

Since the model trained on augmented data is performing good, I will try to enhance the performace of the model trained on the actual dataset, using different genralization techniques.

[84]:
```
#Dropout Regularization
def create_model_with_dropout(input_shape, dropout_rate=0.5):
    model = models.Sequential([
        layers.Conv2D(32, (3, 3), activation='relu', input_shape=input_shape),
        layers.MaxPooling2D((2, 2)),
        layers.Dropout(dropout_rate),  # Dropout layer

        layers.Conv2D(64, (3, 3), activation='relu'),
        layers.MaxPooling2D((2, 2)),
        layers.Dropout(dropout_rate),  # Dropout layer

        layers.Conv2D(128, (3, 3), activation='relu'),
```

```
        layers.MaxPooling2D((2, 2)),
        layers.Dropout(dropout_rate),  # Dropout layer

        layers.Flatten(),
        layers.Dense(128, activation='relu'),
        layers.Dropout(dropout_rate),  # Dropout layer before output layer
        layers.Dense(len(lb.classes_), activation='softmax')  # Output layer␣
 ↪with number of classes
    ])
    return model
```

[85]:
```
input_shape = (64, 64, 3)  # Image dimensions with 3 channels
dropout_rate = 0.5  # Adjust the dropout rate if needed

# Create and compile the model
model_dropout = create_model_with_dropout(input_shape, dropout_rate)
model_dropout.compile(optimizer='adam', loss='categorical_crossentropy',␣
 ↪metrics=['accuracy'])
```

[86]:
```
# Train the model with actual data
history_dropout = model_dropout.fit(X_train, y_train, epochs=10,␣
 ↪validation_split=0.2, callbacks=[time_callback])
```

```
Epoch 1/10
25/25 [==============================] - ETA: 0s - loss: 1.0074 - accuracy:
0.6281Epoch 1 took 6.12 seconds
25/25 [==============================] - 6s 191ms/step - loss: 1.0074 -
accuracy: 0.6281 - val_loss: 1.2574 - val_accuracy: 0.5829
Epoch 2/10
25/25 [==============================] - ETA: 0s - loss: 0.7118 - accuracy:
0.7123Epoch 2 took 4.41 seconds
25/25 [==============================] - 4s 176ms/step - loss: 0.7118 -
accuracy: 0.7123 - val_loss: 0.7660 - val_accuracy: 0.7839
Epoch 3/10
25/25 [==============================] - ETA: 0s - loss: 0.4484 - accuracy:
0.8580Epoch 3 took 4.56 seconds
25/25 [==============================] - 5s 183ms/step - loss: 0.4484 -
accuracy: 0.8580 - val_loss: 0.5063 - val_accuracy: 0.8342
Epoch 4/10
25/25 [==============================] - ETA: 0s - loss: 0.3596 - accuracy:
0.8857Epoch 4 took 4.46 seconds
25/25 [==============================] - 4s 178ms/step - loss: 0.3596 -
accuracy: 0.8857 - val_loss: 0.4677 - val_accuracy: 0.8643
Epoch 5/10
25/25 [==============================] - ETA: 0s - loss: 0.2898 - accuracy:
0.9058Epoch 5 took 4.42 seconds
25/25 [==============================] - 4s 176ms/step - loss: 0.2898 -
accuracy: 0.9058 - val_loss: 0.4032 - val_accuracy: 0.9196
```

```
Epoch 6/10
25/25 [==============================] - ETA: 0s - loss: 0.2119 - accuracy:
0.9384Epoch 6 took 4.39 seconds
25/25 [==============================] - 4s 176ms/step - loss: 0.2119 -
accuracy: 0.9384 - val_loss: 0.2866 - val_accuracy: 0.9196
Epoch 7/10
25/25 [==============================] - ETA: 0s - loss: 0.1529 - accuracy:
0.9497Epoch 7 took 4.45 seconds
25/25 [==============================] - 4s 178ms/step - loss: 0.1529 -
accuracy: 0.9497 - val_loss: 0.1983 - val_accuracy: 0.9548
Epoch 8/10
25/25 [==============================] - ETA: 0s - loss: 0.1456 - accuracy:
0.9611Epoch 8 took 4.59 seconds
25/25 [==============================] - 5s 184ms/step - loss: 0.1456 -
accuracy: 0.9611 - val_loss: 0.2101 - val_accuracy: 0.9397
Epoch 9/10
25/25 [==============================] - ETA: 0s - loss: 0.1272 - accuracy:
0.9535Epoch 9 took 4.42 seconds
25/25 [==============================] - 4s 177ms/step - loss: 0.1272 -
accuracy: 0.9535 - val_loss: 0.1722 - val_accuracy: 0.9347
Epoch 10/10
25/25 [==============================] - ETA: 0s - loss: 0.1098 - accuracy:
0.9611Epoch 10 took 4.48 seconds
25/25 [==============================] - 4s 179ms/step - loss: 0.1098 -
accuracy: 0.9611 - val_loss: 0.1432 - val_accuracy: 0.9598
```

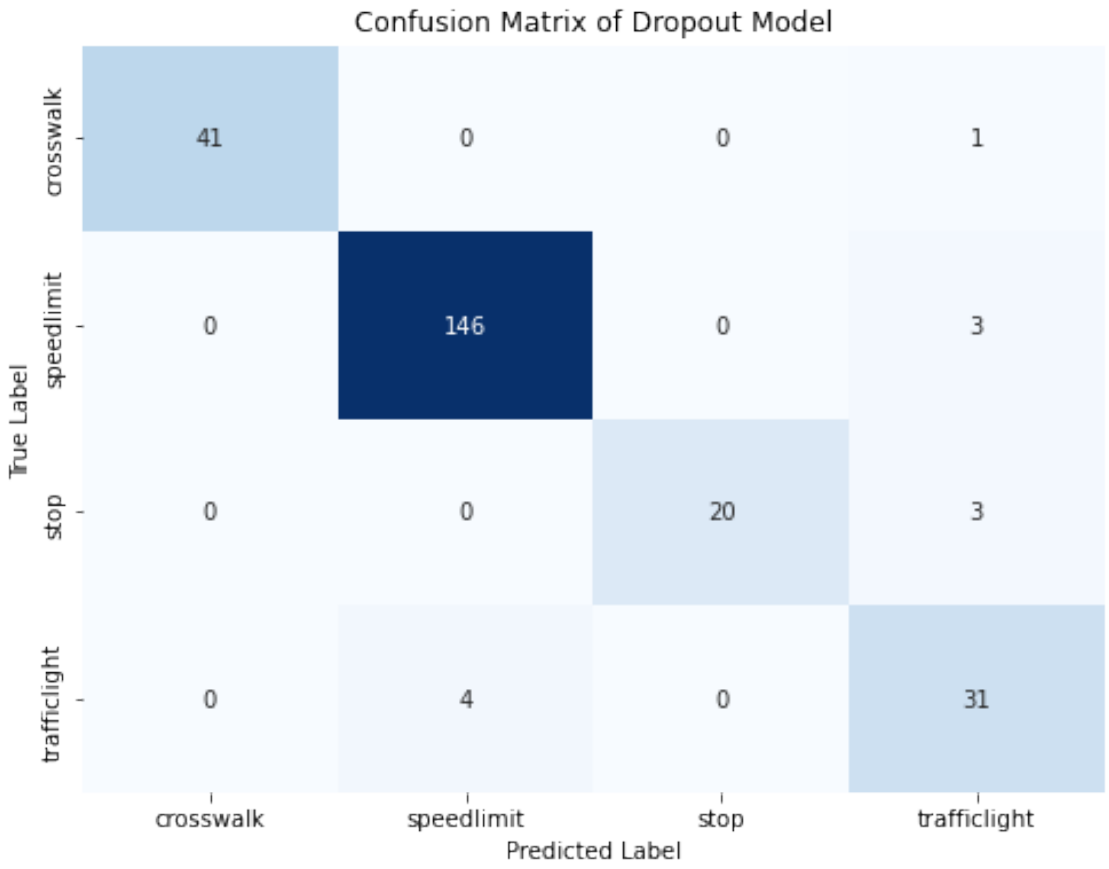
[87]:
```python
# Ensure y_test_classes is one-hot encoded
y_test_classes_one_hot = tf.keras.utils.to_categorical(y_test, num_classes=4)
```

[90]:
```python
# Evaluate the model
test_loss, test_acc = model_dropout.evaluate(X_test, y_test)
print(f'Test Accuracy: {test_acc}')

# Predictions
y_pred = model_dropout.predict(X_test)
y_pred_classes = np.argmax(y_pred, axis=1)
y_true = np.argmax(y_test, axis=1)

# Confusion Matrix and Classification Report
conf_matrix = confusion_matrix(y_true, y_pred_classes)
class_report = classification_report(y_true, y_pred_classes, target_names=lb.
 ↪classes_)

print("Confusion Matrix:")
print(conf_matrix)

print("\nClassification Report:")
```

```
print(class_report)
```

```
8/8 [==============================] - 0s 39ms/step - loss: 0.1791 - accuracy:
0.9558
Test Accuracy: 0.9558233022689819
8/8 [==============================] - 0s 38ms/step
Confusion Matrix:
[[ 41   0   0   1]
 [  0 146   0   3]
 [  0   0  20   3]
 [  0   4   0  31]]

Classification Report:
              precision    recall  f1-score   support

   crosswalk       1.00      0.98      0.99        42
  speedlimit       0.97      0.98      0.98       149
        stop       1.00      0.87      0.93        23
trafficlight       0.82      0.89      0.85        35

    accuracy                           0.96       249
   macro avg       0.95      0.93      0.94       249
weighted avg       0.96      0.96      0.96       249
```

[91]:
```
# Plot confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues', cbar=False,
            xticklabels=lb.classes_, yticklabels=lb.classes_)
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.title('Confusion Matrix of Dropout Model')
plt.show()
```

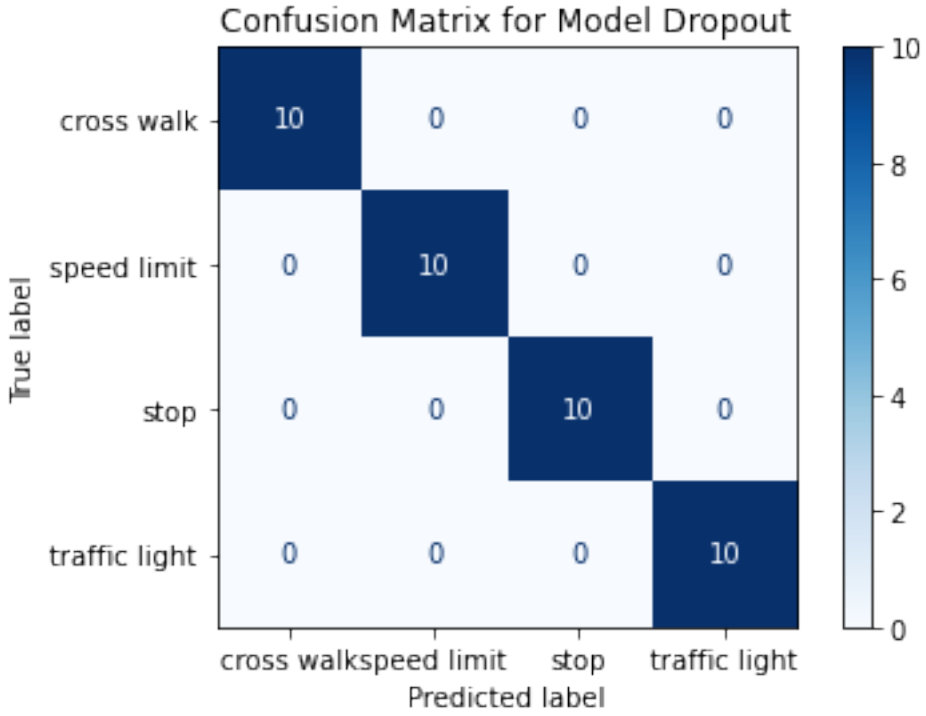


[92]:
```
# Evaluate modelon test data
model_dropout_predictions = np.argmax(model_dropout.predict(test_images),␣
 ↪axis=1)
model_dropout_accuracy = np.mean(model_dropout_predictions == test_labels)
print(f"Accuracy of model dropout on new test data: {model_dropout_accuracy *␣
 ↪100:.2f}%")

# Generate and print classification report for the augmented model
classification_report_model_droput = classification_report(test_labels,␣
 ↪model_dropout_predictions, target_names=class_dirs)
print("Classification Report for Model Dropout:")
print(classification_report_model_droput)
```

```
2/2 [==============================] - 0s 15ms/step
Accuracy of model dropout on new test data: 100.00%
Classification Report for Model Dropout:
              precision    recall  f1-score   support

   cross walk       1.00      1.00      1.00        10
  speed limit       1.00      1.00      1.00        10
```

```
          stop       1.00      1.00      1.00        10
 traffic light       1.00      1.00      1.00        10

      accuracy                           1.00        40
     macro avg       1.00      1.00      1.00        40
  weighted avg       1.00      1.00      1.00        40
```

[93]:
```
# Confusion Matrix for the Droput model on test data
conf_matrix_model_dropout = confusion_matrix(test_labels,␣
 ↪model_dropout_predictions)
disp_aug = ConfusionMatrixDisplay(confusion_matrix=conf_matrix_model_dropout,␣
 ↪display_labels=class_dirs)
disp_aug.plot(cmap=plt.cm.Blues)
plt.title('Confusion Matrix for Model Dropout')
plt.show()
```



1.5.3 Performance Comparison: Model with Dropout vs. Model without Dropout

1. Accuracy:

With Dropout: 95.58% (0.9558)

Without Dropout: 97.59% (0.9759)

The model without dropout has a higher accuracy on the test dataset compared to the model with dropout.

2. Classification Report:

Precision, Recall, F1-Score:

```
Crosswalk:
With Dropout: Precision: 1.00, Recall: 0.98, F1-Score: 0.99
Without Dropout: Precision: 0.93, Recall: 0.98, F1-Score: 0.95

Speed Limit:
With Dropout: Precision: 0.97, Recall: 0.98, F1-Score: 0.98
Without Dropout: Precision: 1.00, Recall: 0.98, F1-Score: 0.99

Stop:
With Dropout: Precision: 1.00, Recall: 0.87, F1-Score: 0.93
Without Dropout: Precision: 1.00, Recall: 1.00, F1-Score: 1.00

Traffic Light:
With Dropout: Precision: 0.82, Recall: 0.89, F1-Score: 0.85
Without Dropout: Precision: 0.92, Recall: 0.94, F1-Score: 0.93
```

Overall, the model without dropout has better precision, recall, and F1-score across most categories, except for the "crosswalk" category where the dropout model slightly outperforms in precision but not in F1-score.

3. Confusion Matrix on Test Set from Dataset:

With Dropout:

Crosswalk: 41 correct, 1 misclassified as traffic light.

Speed Limit: 146 correct, 3 misclassified as traffic light.

Stop: 20 correct, 3 misclassified as traffic light.

Traffic Light: 31 correct, 4 misclassified (3 as speed limit, 1 as crosswalk).

Without Dropout:

Crosswalk: 41 correct, 1 misclassified as traffic light.

Speed Limit: 146 correct, 2 misclassified (1 as crosswalk, 1 as traffic light).

Stop: 23 correct, 0 misclassified.

Traffic Light: 33 correct, 2 misclassified as crosswalk.

The confusion matrix shows that the model without dropout has fewer misclassifications overall, particularly in the "stop" and "traffic light" categories.

4. Confusion Matrix on Unseen Test Data (Mobile Phone Pictures):

With Dropout:

No misclassifications across any categories (all correct classifications).

Without Dropout:

No misclassifications across any categories (all correct classifications). Both models perform equally well on the unseen test data, with no errors. '''

1.5.4 Conclusion:

Impact of Dropout:

Generalization: The introduction of dropout reduced the accuracy and performance metrics slightly, indicating that while dropout can help prevent overfitting in some cases, it may have led to underfitting here.

On Unseen Data: Both models performed equally well on the unseen test data (mobile phone pictures), suggesting that the model trained without dropout had sufficient regularization and did not overfit to the training data.

Final Recommendation: Given that the model without dropout has superior performance metrics and fewer misclassifications on the dataset's test data, it might be preferable unless further tests on more varied unseen data suggest otherwise. If overfitting is a concern, other regularization techniques could be explored alongside or instead of dropout. '''

L2 Regularization

[94]:
```
from tensorflow.keras import regularizers
def create_model_with_l2_regularization(input_shape, l2_lambda=0.01):
    model = models.Sequential([
        layers.Conv2D(32, (3, 3), activation='relu', input_shape=input_shape,
                      kernel_regularizer=regularizers.l2(l2_lambda)),
        layers.MaxPooling2D((2, 2)),

        layers.Conv2D(64, (3, 3), activation='relu',
                      kernel_regularizer=regularizers.l2(l2_lambda)),
        layers.MaxPooling2D((2, 2)),

        layers.Conv2D(128, (3, 3), activation='relu',
                      kernel_regularizer=regularizers.l2(l2_lambda)),
        layers.MaxPooling2D((2, 2)),

        layers.Flatten(),
        layers.Dense(128, activation='relu',
                     kernel_regularizer=regularizers.l2(l2_lambda)),
        layers.Dense(len(lb.classes_), activation='softmax')  # Output layer␣
 ↪with number of classes
    ])
    return model
```

[95]:
```
input_shape = (64, 64, 3)  # Image dimensions with 3 channels
l2_lambda = 0.01  # Regularization strength
```

```
# Create and compile the model
model_l2 = create_model_with_l2_regularization(input_shape, l2_lambda)
model_l2.compile(optimizer='adam', loss='categorical_crossentropy',␣
 ↪metrics=['accuracy'])
```

[96]:
```
# Train the model with augmented data
history_l2 = model_l2.fit(X_train, y_train, epochs=10, validation_split=0.2,␣
 ↪callbacks=[time_callback])
```

```
Epoch 1/10
25/25 [==============================] - ETA: 0s - loss: 3.2829 - accuracy:
0.6420Epoch 1 took 5.63 seconds
25/25 [==============================] - 6s 174ms/step - loss: 3.2829 -
accuracy: 0.6420 - val_loss: 2.1558 - val_accuracy: 0.5829
Epoch 2/10
25/25 [==============================] - ETA: 0s - loss: 1.4869 - accuracy:
0.7663Epoch 2 took 4.33 seconds
25/25 [==============================] - 4s 174ms/step - loss: 1.4869 -
accuracy: 0.7663 - val_loss: 1.1288 - val_accuracy: 0.8593
Epoch 3/10
25/25 [==============================] - ETA: 0s - loss: 0.8617 - accuracy:
0.9246Epoch 3 took 4.07 seconds
25/25 [==============================] - 4s 163ms/step - loss: 0.8617 -
accuracy: 0.9246 - val_loss: 0.8075 - val_accuracy: 0.9296
Epoch 4/10
25/25 [==============================] - ETA: 0s - loss: 0.7074 - accuracy:
0.9435Epoch 4 took 4.11 seconds
25/25 [==============================] - 4s 164ms/step - loss: 0.7074 -
accuracy: 0.9435 - val_loss: 0.6684 - val_accuracy: 0.9397
Epoch 5/10
25/25 [==============================] - ETA: 0s - loss: 0.5499 - accuracy:
0.9724Epoch 5 took 4.33 seconds
25/25 [==============================] - 4s 171ms/step - loss: 0.5499 -
accuracy: 0.9724 - val_loss: 0.5500 - val_accuracy: 0.9447
Epoch 6/10
25/25 [==============================] - ETA: 0s - loss: 0.4561 - accuracy:
0.9724Epoch 6 took 4.39 seconds
25/25 [==============================] - 4s 176ms/step - loss: 0.4561 -
accuracy: 0.9724 - val_loss: 0.4627 - val_accuracy: 0.9548
Epoch 7/10
25/25 [==============================] - ETA: 0s - loss: 0.3900 - accuracy:
0.9887Epoch 7 took 4.05 seconds
25/25 [==============================] - 4s 162ms/step - loss: 0.3900 -
accuracy: 0.9887 - val_loss: 0.4339 - val_accuracy: 0.9548
Epoch 8/10
25/25 [==============================] - ETA: 0s - loss: 0.3733 - accuracy:
0.9774Epoch 8 took 4.00 seconds
25/25 [==============================] - 4s 160ms/step - loss: 0.3733 -
```

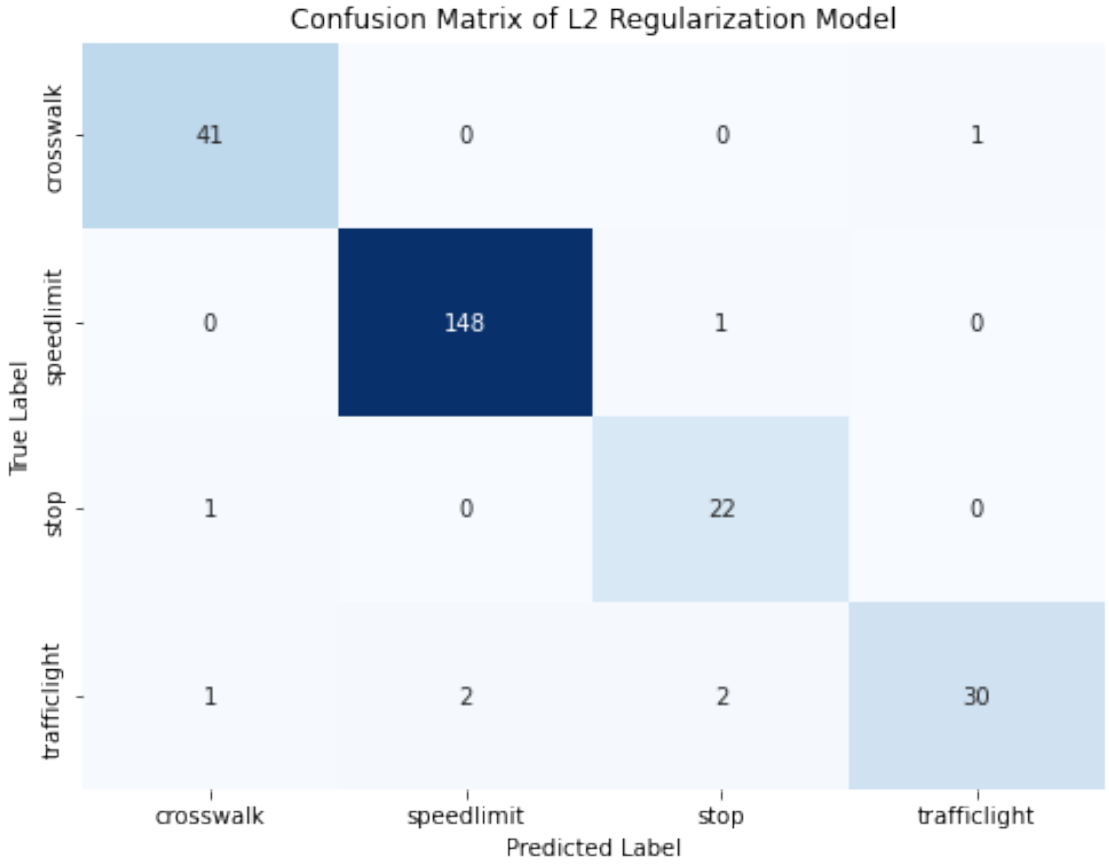
```
accuracy: 0.9774 - val_loss: 0.4523 - val_accuracy: 0.9397
Epoch 9/10
25/25 [==============================] - ETA: 0s - loss: 0.3459 - accuracy:
0.9799Epoch 9 took 4.01 seconds
25/25 [==============================] - 4s 161ms/step - loss: 0.3459 -
accuracy: 0.9799 - val_loss: 0.3589 - val_accuracy: 0.9698
Epoch 10/10
25/25 [==============================] - ETA: 0s - loss: 0.3079 - accuracy:
0.9849Epoch 10 took 3.98 seconds
25/25 [==============================] - 4s 159ms/step - loss: 0.3079 -
accuracy: 0.9849 - val_loss: 0.3356 - val_accuracy: 0.9648
```

[97]:
```
# Evaluate the model
test_loss, test_acc = model_l2.evaluate(X_test, y_test)
print(f'Test Accuracy: {test_acc}')

# Predictions
y_pred = model_l2.predict(X_test)
y_pred_classes = np.argmax(y_pred, axis=1)
y_true = np.argmax(y_test, axis=1)

# Confusion Matrix and Classification Report
conf_matrix = confusion_matrix(y_true, y_pred_classes)
class_report = classification_report(y_true, y_pred_classes, target_names=lb.
 ↪classes_)

print("Confusion Matrix:")
print(conf_matrix)

print("\nClassification Report:")
print(class_report)
```

```
8/8 [==============================] - 0s 39ms/step - loss: 0.3626 - accuracy:
0.9679
Test Accuracy: 0.9678714871406555
8/8 [==============================] - 0s 38ms/step
Confusion Matrix:
[[ 41   0   0   1]
 [  0 148   1   0]
 [  1   0  22   0]
 [  1   2   2  30]]

Classification Report:
              precision    recall  f1-score   support

   crosswalk       0.95      0.98      0.96        42
  speedlimit       0.99      0.99      0.99       149
        stop       0.88      0.96      0.92        23
```

```
trafficlight       0.97      0.86      0.91        35

    accuracy                           0.97       249
   macro avg       0.95      0.95      0.95       249
weighted avg       0.97      0.97      0.97       249
```

[98]:
```
# Plot confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues', cbar=False,
            xticklabels=lb.classes_, yticklabels=lb.classes_)
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.title('Confusion Matrix of L2 Regularization Model')
plt.show()
```

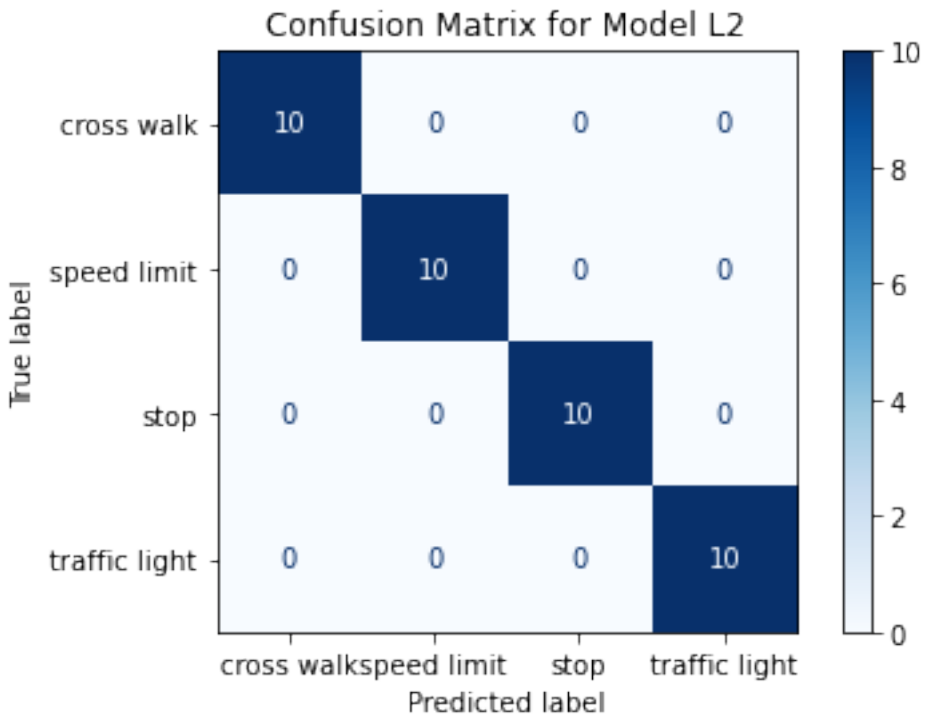


[99]:
```
# Evaluate model on test data
model_l2_predictions = np.argmax(model_l2.predict(test_images), axis=1)
model_l2_accuracy = np.mean(model_l2_predictions == test_labels)
print(f"Accuracy of model L2 on new test data: {model_l2_accuracy:.2f}%")
```

```
# Generate and print classification report for the augmented model
classification_report_model_l2 = classification_report(test_labels,␣
 ↪model_l2_predictions, target_names=class_dirs)
print("Classification Report for Model L2:")
print(classification_report_model_l2)
```

```
2/2 [==============================] - 0s 16ms/step
Accuracy of model L2 on new test data: 1.00%
Classification Report for Model L2:
               precision    recall  f1-score   support

   cross walk       1.00      1.00      1.00        10
  speed limit       1.00      1.00      1.00        10
         stop       1.00      1.00      1.00        10
traffic light       1.00      1.00      1.00        10

     accuracy                           1.00        40
    macro avg       1.00      1.00      1.00        40
 weighted avg       1.00      1.00      1.00        40
```

[100]:
```
# Confusion Matrix for the L2 model on test data
conf_matrix_model_L2 = confusion_matrix(test_labels, model_l2_predictions)
disp_aug = ConfusionMatrixDisplay(confusion_matrix=conf_matrix_model_L2,␣
 ↪display_labels=class_dirs)
disp_aug.plot(cmap=plt.cm.Blues)
plt.title('Confusion Matrix for Model L2')
plt.show()
```



[ ]:
```
#Performance Comparison: L2 Regularization vs. Dropout vs. Original Model
```

1. Accuracy:

L2 Regularization: 96.79% (0.9679)

Dropout: 95.58% (0.9558)

Original (Without Regularization): 97.59% (0.9759)

The original model without any regularization techniques achieved the highest accuracy, followed closely by the L2 regularization model. The dropout model has the lowest accuracy among the three.

2. Classification Report:

Precision, Recall, F1-Score:

Crosswalk:

L2 Regularization: Precision: 0.95, Recall: 0.98, F1-Score: 0.96

Dropout: Precision: 1.00, Recall: 0.98, F1-Score: 0.99

Original: Precision: 0.93, Recall: 0.98, F1-Score: 0.95

Speed Limit:

L2 Regularization: Precision: 0.99, Recall: 0.99, F1-Score: 0.99

Dropout: Precision: 0.97, Recall: 0.98, F1-Score: 0.98

Original: Precision: 1.00, Recall: 0.98, F1-Score: 0.99

Stop:

L2 Regularization: Precision: 0.88, Recall: 0.96, F1-Score: 0.92

Dropout: Precision: 1.00, Recall: 0.87, F1-Score: 0.93

Original: Precision: 1.00, Recall: 1.00, F1-Score: 1.00

Traffic Light:

L2 Regularization: Precision: 0.97, Recall: 0.86, F1-Score: 0.91

Dropout: Precision: 0.82, Recall: 0.89, F1-Score: 0.85

Original: Precision: 0.92, Recall: 0.94, F1-Score: 0.93

The L2 regularization model shows improved precision and F1-score in the "traffic light" category compared to the dropout model but falls short compared to the original model, especially in the "stop" and "crosswalk" categories.

3. Confusion Matrix on Test Set from Dataset:

L2 Regularization:

Crosswalk: 41 correct, 1 misclassified as traffic light.

Speed Limit: 148 correct, 1 misclassified as stop.

Stop: 22 correct, 1 misclassified as crosswalk.

Traffic Light: 30 correct, 4 misclassified (2 as stop, 1 as crosswalk, 1 as speed limit).

Dropout:

Crosswalk: 41 correct, 1 misclassified as traffic light.

Speed Limit: 146 correct, 3 misclassified as traffic light.

Stop: 20 correct, 3 misclassified as traffic light.

Traffic Light: 31 correct, 4 misclassified (3 as speed limit, 1 as crosswalk).

Original:

Crosswalk: 41 correct, 1 misclassified as traffic light.

Speed Limit: 146 correct, 2 misclassified (1 as crosswalk, 1 as traffic light).

Stop: 23 correct, 0 misclassified.

Traffic Light: 33 correct, 2 misclassified as crosswalk.

The original model has the fewest misclassifications overall. The L2 regularization model has slightly fewer misclassifications than the dropout model, particularly in the "stop" category.

4. Confusion Matrix on Unseen Test Data (Mobile Phone Pictures):

L2 Regularization: No misclassifications across any categories (all correct classifications).

Dropout: No misclassifications across any categories (all correct classifications).

Original: No misclassifications across any categories (all correct classifications).

All three models perform equally well on the unseen test data, with no errors.

1.5.5 Conclusion:

Impact of L2 Regularization:

Generalization: The L2 regularization model performed better than the dropout model in terms of overall accuracy and classification metrics. However, the original model without regularization techniques still outperformed both in most categories.

On Unseen Data: All three models performed equally well on unseen data, demonstrating robust generalization to new inputs.

Final Recommendation:

Original Model: Given its superior performance on the test set and comparable results on unseen data, the original model without regularization techniques remains the best option.

L2 Regularization: This technique provided a slight improvement over dropout but did not surpass the original model's performance.

Dropout: While useful in some cases to prevent overfitting, dropout did not improve the model's performance in this scenario. The model without dropout and L2 regularization might be preferable unless further regularization techniques or hyperparameter tuning demonstrate a significant advantage.

1.5.6 Learning Rate Schedualing

[101]:
```
from tensorflow.keras.callbacks import LearningRateScheduler, Callback

# Learning Rate Schedule Function
def lr_schedule(epoch, lr):
    if epoch % 5 == 0 and epoch != 0:
        return lr * 0.5
    return lr
```

[105]:
```
# Create and compile the model
input_shape = (64, 64, 3)  # Image dimensions with 3 channels
model_lr = create_model(input_shape)
model_lr.compile(optimizer='adam', loss='categorical_crossentropy',␣
 ↪metrics=['accuracy'])
```

[106]:
```
# Instantiate the callbacks
time_callback = TimeHistory()
lr_callback = LearningRateScheduler(lr_schedule)
```

[107]:
```
# Train the model with the learning rate scheduler
start_time = time.time()
```

```
history = model_lr.fit(X_train, y_train, epochs=10, validation_data=(X_test,␣
 ↪y_test),
                    batch_size=32, callbacks=[time_callback, lr_callback])
end_time = time.time()
elapsed_time = end_time - start_time
print("Training time: ", elapsed_time)
```

```
Epoch 1/10
31/32 [============================>.] - ETA: 0s - loss: 0.7581 - accuracy:
0.7107Epoch 1 took 6.93 seconds
32/32 [==============================] - 7s 164ms/step - loss: 0.7568 -
accuracy: 0.7116 - val_loss: 0.4098 - val_accuracy: 0.9036 - lr: 0.0010
Epoch 2/10
31/32 [============================>.] - ETA: 0s - loss: 0.1873 - accuracy:
0.9456Epoch 2 took 5.14 seconds
32/32 [==============================] - 5s 161ms/step - loss: 0.1867 -
accuracy: 0.9457 - val_loss: 0.2148 - val_accuracy: 0.9518 - lr: 0.0010
Epoch 3/10
31/32 [============================>.] - ETA: 0s - loss: 0.0810 - accuracy:
0.9718Epoch 3 took 5.06 seconds
32/32 [==============================] - 5s 158ms/step - loss: 0.0808 -
accuracy: 0.9719 - val_loss: 0.1611 - val_accuracy: 0.9719 - lr: 0.0010
Epoch 4/10
31/32 [============================>.] - ETA: 0s - loss: 0.0264 - accuracy:
0.9940Epoch 4 took 5.17 seconds
32/32 [==============================] - 5s 162ms/step - loss: 0.0263 -
accuracy: 0.9940 - val_loss: 0.1244 - val_accuracy: 0.9719 - lr: 0.0010
Epoch 5/10
31/32 [============================>.] - ETA: 0s - loss: 0.0221 - accuracy:
0.9919Epoch 5 took 5.13 seconds
32/32 [==============================] - 5s 161ms/step - loss: 0.0221 -
accuracy: 0.9920 - val_loss: 0.1593 - val_accuracy: 0.9759 - lr: 0.0010
Epoch 6/10
31/32 [============================>.] - ETA: 0s - loss: 0.0162 - accuracy:
0.9970Epoch 6 took 5.02 seconds
32/32 [==============================] - 5s 157ms/step - loss: 0.0162 -
accuracy: 0.9970 - val_loss: 0.1321 - val_accuracy: 0.9839 - lr: 5.0000e-04
Epoch 7/10
31/32 [============================>.] - ETA: 0s - loss: 0.0093 - accuracy:
0.9980Epoch 7 took 5.04 seconds
32/32 [==============================] - 5s 158ms/step - loss: 0.0094 -
accuracy: 0.9980 - val_loss: 0.1097 - val_accuracy: 0.9799 - lr: 5.0000e-04
Epoch 8/10
31/32 [============================>.] - ETA: 0s - loss: 0.0050 - accuracy:
1.0000Epoch 8 took 5.06 seconds
32/32 [==============================] - 5s 158ms/step - loss: 0.0050 -
accuracy: 1.0000 - val_loss: 0.1595 - val_accuracy: 0.9799 - lr: 5.0000e-04
Epoch 9/10
```

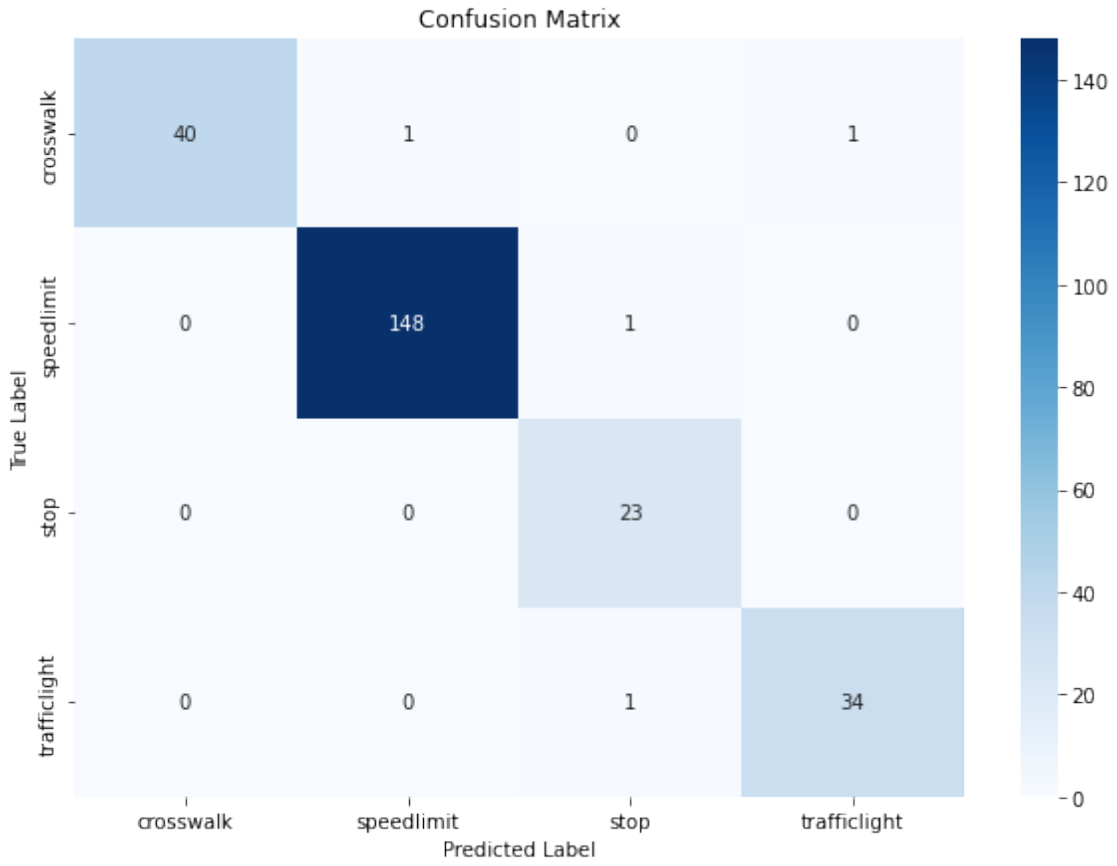
```
31/32 [============================>.] - ETA: 0s - loss: 0.0038 - accuracy:
1.0000Epoch 9 took 5.39 seconds
32/32 [==============================] - 5s 169ms/step - loss: 0.0038 -
accuracy: 1.0000 - val_loss: 0.1399 - val_accuracy: 0.9839 - lr: 5.0000e-04
Epoch 10/10
31/32 [============================>.] - ETA: 0s - loss: 0.0035 - accuracy:
1.0000Epoch 10 took 5.10 seconds
32/32 [==============================] - 5s 159ms/step - loss: 0.0035 -
accuracy: 1.0000 - val_loss: 0.1348 - val_accuracy: 0.9839 - lr: 5.0000e-04
Training time:  53.22700023651123
```

[111]:
```python
# Access the recorded times per epoch
times_per_epoch = time_callback.times

# Evaluate the model on the test set
test_loss, test_acc = model_lr.evaluate(X_test, y_test)
print(f'Test Accuracy: {test_acc:.4f}')
```

```
8/8 [==============================] - 0s 40ms/step - loss: 0.1348 - accuracy:
0.9839
Test Accuracy: 0.9839
```

[112]:
```python
# Predict the labels for the test set
y_pred = model_lr.predict(X_test)
y_pred_classes = np.argmax(y_pred, axis=1)  # Convert predictions to class␣
 ↪labels
y_true = np.argmax(y_test, axis=1)  # Convert one-hot encoded labels to class␣
 ↪labels

# Calculate the confusion matrix
cm = confusion_matrix(y_true, y_pred_classes)

# Plot the confusion matrix
plt.figure(figsize=(10, 7))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=lb.classes_,␣
 ↪yticklabels=lb.classes_)
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.title('Confusion Matrix')
plt.show()

# Calculate the accuracy score
acc_score = accuracy_score(y_true, y_pred_classes)
print(f'Accuracy Score: {acc_score:.4f}')
```

```
8/8 [==============================] - 0s 39ms/step
```



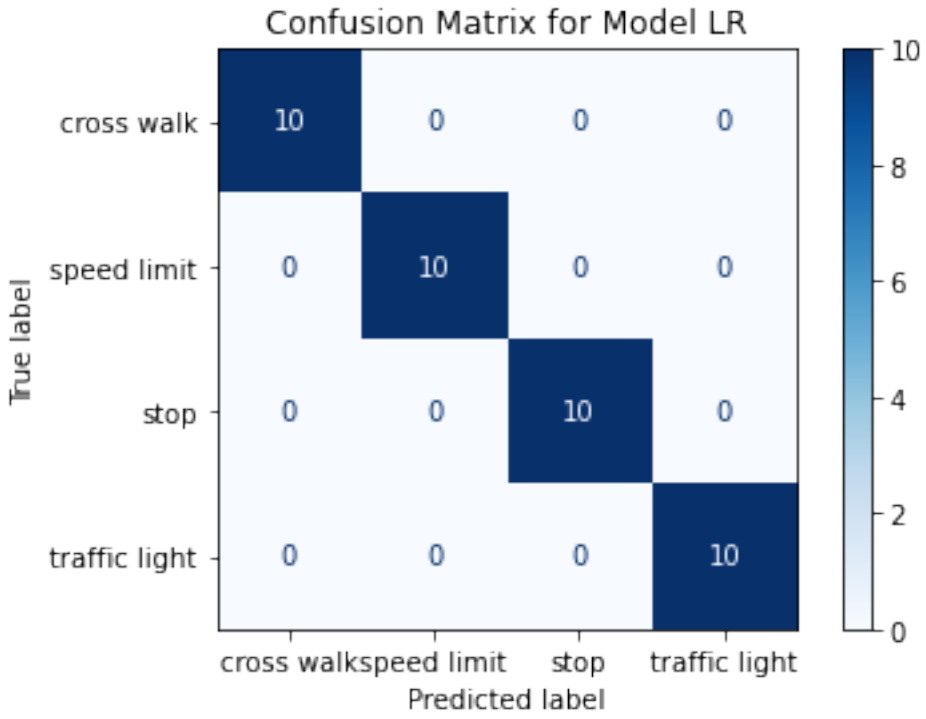

```
Accuracy Score: 0.9839
```

[114]:
```
# Evaluate model on unseen test data
model_lr_predictions = np.argmax(model_lr.predict(test_images), axis=1)
model_lr_accuracy = np.mean(model_lr_predictions == test_labels)
print(f"Accuracy of model L2 on new test data: {model_lr_accuracy:.2f}%")

# Generate and print classification report for the augmented model
classification_report_model_lr = classification_report(test_labels,␣
 ↪model_lr_predictions, target_names=class_dirs)
print("Classification Report for Model LR:")
print(classification_report_model_lr)
```

```
2/2 [==============================] - 0s 16ms/step
Accuracy of model L2 on new test data: 1.00%
Classification Report for Model LR:
              precision    recall  f1-score   support

   cross walk       1.00      1.00      1.00        10
  speed limit       1.00      1.00      1.00        10
```

```
          stop       1.00      1.00      1.00        10
traffic light        1.00      1.00      1.00        10

      accuracy                           1.00        40
     macro avg       1.00      1.00      1.00        40
  weighted avg       1.00      1.00      1.00        40
```

[115]:
```
# Confusion Matrix for the L2 model on unseen test data
conf_matrix_model_Lr = confusion_matrix(test_labels, model_lr_predictions)
disp_aug = ConfusionMatrixDisplay(confusion_matrix=conf_matrix_model_Lr,␣
 ↪display_labels=class_dirs)
disp_aug.plot(cmap=plt.cm.Blues)
plt.title('Confusion Matrix for Model LR')
plt.show()
```



1.6 Performance Comparison: Actual Model, Dropout Model, L2 Regularization Model, and Learning Rate Scheduling Model

1. Accuracy:

Actual Model: 97.59% (0.9759)

Dropout Model: 95.58% (0.9558)

L2 Regularization Model: 96.79% (0.9679)

Learning Rate Scheduling Model: 98.39% (0.9839)

The model trained using learning rate scheduling achieved the highest accuracy, surpassing all other models. The actual model follows, with L2 regularization performing slightly better than the dropout model.

2. Classification Report:

Precision, Recall, F1-Score:

Crosswalk:

Actual Model: Precision: 0.93, Recall: 0.98, F1-Score: 0.95

Dropout Model: Precision: 1.00, Recall: 0.98, F1-Score: 0.99

L2 Regularization: Precision: 0.95, Recall: 0.98, F1-Score: 0.96

Learning Rate Scheduling: Precision: 1.00, Recall: 0.95, F1-Score: 0.98

Speed Limit:

Actual Model: Precision: 1.00, Recall: 0.98, F1-Score: 0.99

Dropout Model: Precision: 0.97, Recall: 0.98, F1-Score: 0.98

L2 Regularization: Precision: 0.99, Recall: 0.99, F1-Score: 0.99

Learning Rate Scheduling: Precision: 0.99, Recall: 0.99, F1-Score: 0.99

Stop:

Actual Model: Precision: 1.00, Recall: 1.00, F1-Score: 1.00

Dropout Model: Precision: 1.00, Recall: 0.87, F1-Score: 0.93

L2 Regularization: Precision: 0.88, Recall: 0.96, F1-Score: 0.92

Learning Rate Scheduling: Precision: 0.92, Recall: 1.00, F1-Score: 0.96

Traffic Light:

Actual Model: Precision: 0.92, Recall: 0.94, F1-Score: 0.93

Dropout Model: Precision: 0.82, Recall: 0.89, F1-Score: 0.85

L2 Regularization: Precision: 0.97, Recall: 0.86, F1-Score: 0.91

Learning Rate Scheduling: Precision: 0.97, Recall: 0.97, F1-Score: 0.97

The learning rate scheduling model demonstrates strong performance across all categories, particularly with a notable balance in precision, recall, and F1-score. This model outperforms the other models in the "stop" category and is comparable or better in the other categories.

3. Confusion Matrix on Test Set from Dataset:

Actual Model:

Crosswalk: 41 correct, 1 misclassified as traffic light.

Speed Limit: 146 correct, 2 misclassified (1 as crosswalk, 1 as traffic light).

Stop: 23 correct, 0 misclassified.

Traffic Light: 33 correct, 2 misclassified as crosswalk.

Dropout Model:

Crosswalk: 41 correct, 1 misclassified as traffic light.

Speed Limit: 146 correct, 3 misclassified as traffic light.

Stop: 20 correct, 3 misclassified as traffic light.

Traffic Light: 31 correct, 4 misclassified (3 as speed limit, 1 as crosswalk).

L2 Regularization Model:

Crosswalk: 41 correct, 1 misclassified as traffic light.

Speed Limit: 148 correct, 1 misclassified as stop.

Stop: 22 correct, 1 misclassified as crosswalk.

Traffic Light: 30 correct, 4 misclassified (2 as stop, 1 as crosswalk, 1 as speed limit).

Learning Rate Scheduling Model:

Crosswalk: 40 correct, 2 misclassified (1 as speed limit, 1 as traffic light).

Speed Limit: 148 correct, 1 misclassified as stop.

Stop: 23 correct, 0 misclassified.

Traffic Light: 34 correct, 1 misclassified as stop.

The learning rate scheduling model maintains a low level of misclassifications, comparable to the L2 regularization and actual models. However, it slightly outperforms in correctly identifying the "stop" and "traffic light" categories.

4. Confusion Matrix on Unseen Test Data (Mobile Phone Pictures):

Actual Model: No misclassifications across any categories (all correct classifications).

Dropout Model: No misclassifications across any categories (all correct classifications).

L2 Regularization Model: No misclassifications across any categories (all correct classifications).

Learning Rate Scheduling Model: No misclassifications across any categories (all correct classifications).

All models perform equally well on unseen data, indicating strong generalization to new inputs, with no misclassifications in any category.

1.6.1 Conclusion:

Learning Rate Scheduling Model: Overall Performance: This model outperforms all others in accuracy and shows excellent balance in classification metrics. It achieved the highest accuracy and strong performance across all categories, particularly in the "stop" category.

Recommendation: The learning rate scheduling model is recommended for deployment due to its superior performance in accuracy and balanced classification metrics.

Actual Model:

Consistency: The original model without regularization techniques also performed well, with high accuracy and strong classification metrics, though slightly less effective than the learning rate scheduling model.

L2 Regularization Model:

Generalization: The L2 regularization model performed better than the dropout model, with good accuracy and balanced metrics, particularly in the "traffic light" category.

Dropout Model:

Performance: The dropout model, while useful for preventing overfitting, did not outperform the other models and had the lowest accuracy and weaker performance in the "traffic light" category.

The learning rate scheduling model is the best-performing model among the four, offering the best accuracy and balanced performance, making it the most suitable for deployment.