```python
import os  # Importing the os module to interact with the operating system
import numpy as np  # Importing NumPy for numerical operations
import tensorflow as tf  # Importing TensorFlow for deep learning
from tensorflow.keras import layers, models  # Importing Keras layers and models
from sklearn.model_selection import train_test_split  # Importing function to split data into training and validation sets
from sklearn.metrics import f1_score  # Importing F1 score metric for evaluation
import cv2  # Importing OpenCV for image processing
from tensorflow.keras.applications import MobileNetV2  # Importing MobileNetV2 model for transfer learning
from tensorflow.keras.layers import Dense, GlobalAveragePooling2D  # Importing Dense and Global Average Pooling layers
from tensorflow.keras.preprocessing.image import ImageDataGenerator  # Importing class for image data augmentation
from sklearn.utils import class_weight  # Importing function to compute class weights
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau, ModelCheckpoint  # Importing callbacks for training

# Check and configure GPU usage
if tf.config.list_physical_devices('GPU'):  # Check if a GPU is available
    print("GPU is available. Using CUDA for training.")  # If GPU is available, notify that CUDA will be used
else:
    print("GPU is not available. Training will be performed on the CPU.")  # If not, notify that CPU will be used


# Dataset loading and Preprocessing Phase
# Constants
img_dir = '/content/drive/My Drive/processed_images/'  # Directory where processed images are stored
lbl_dir = '/content/drive/My Drive/processed_labels/'  # Directory where corresponding labels are stored
TARGET_CLASSES = ['Car', 'Cyclist', 'Pedestrian', 'Van', 'Truck']  # List of target classes for classification
image_size = (224, 224)  # Target image size for model input

# Load images and labels
def load_data(img_dir, lbl_dir):  # Define function to load images and labels
    images, labels = [], []  # Initialize empty lists for images and labels
    for image_file in sorted(os.listdir(img_dir)):  # Loop through sorted image files in the image directory
        img = cv2.imread(os.path.join(img_dir, image_file))  # Read the image file
        img = cv2.resize(img, image_size) / 255.0  # Resize the image and normalize pixel values to [0, 1]
```

```
        images.append(img)  # Append the processed image to the images list

        label_file = os.path.splitext(image_file)[0] + '.txt'  # Construct the label file name based on the image file name
        label_path = os.path.join(lbl_dir, label_file)  # Create the full path for the label file
        if os.path.exists(label_path):  # Check if the label file exists
            with open(label_path, 'r') as f:  # Open the label file
                for line in f:  # Loop through each line in the label file
                    class_label = line.strip().split()[0]  # Extract the class label from the line
                    if class_label in TARGET_CLASSES:  # Check if the class label is one of the target classes
                        labels.append(TARGET_CLASSES.index(class_label))  # Append the corresponding index of the class label to labels
        else:
            print(f"Warning: Missing label file for {image_file}")  # Warn if the label file is missing

    return np.array(images), np.array(labels)  # Return the images and labels as NumPy arrays

# Load data
X, y = load_data(img_dir, lbl_dir)  # Call the load_data function to load images and labels
print(f"Initial shapes → X: {X.shape}, y: {y.shape}")  # Print the shapes of the loaded images and labels

# Ensure that y is a valid length corresponding to X
if len(X) < len(y):  # Check if the number of images is less than the number of labels
    print("Warning: The length of y exceeds the length of X. Adjusting y to match X.")  # Warn if y exceeds X
    y = y[:len(X)]  # Trim y to match the length of X

# Filter out invalid labels
valid_indices = [i for i in range(len(X)) if y[i] is not None]  # Create a list of valid indices where y is not None
X_filtered = X[valid_indices]  # Filter images based on valid indices
y_filtered = y[valid_indices]  # Filter labels based on valid indices

# Convert labels to categorical
y_filtered = tf.keras.utils.to_categorical(y_filtered, num_classes=len(TARGET_CLASSES))  # Convert labels to one-hot encoded format

print(f"Filtered shapes → X_filtered: {X_filtered.shape}, y_filtered: {y_filtered.shape}")  # Print the shapes of the filtered images and labels

# Split data into training and validation sets
X_train, X_val, y_train, y_val = train_test_split(X_filtered, y_filtered, test_size=0.2, random_state=42)  # Split data into training and validation sets
```

```
with a 80-20 ratio

# Augment Data
datagen = ImageDataGenerator(  # Initialize ImageDataGenerator for data augmentation
    rotation_range=20,  # Randomly rotate images by up to 20 degrees
    width_shift_range=0.2,  # Randomly shift images horizontally by up to 20%
    height_shift_range=0.2,  # Randomly shift images vertically by up to 20%
    horizontal_flip=True,  # Randomly flip images horizontally
    zoom_range=0.2,  # Randomly zoom images by up to 20%
    fill_mode='nearest'  # Fill in new pixels after transformations
)



# Model define and Training
# Define MobileNetV2 model
base_model = MobileNetV2(weights='imagenet', include_top=False, input_shape=(224, 224, 3))  # Load MobileNetV2 without the top layer for transfer learning

for layer in base_model.layers[:-20]:  # Freeze most layers in the base model for transfer learning
    layer.trainable = False  # Set the trainable property to False to prevent weight updates during training

# Add classifier layers
model = tf.keras.Sequential([  # Initialize a Sequential model
    base_model,  # Add the base model (MobileNetV2)
    GlobalAveragePooling2D(),  # Add global average pooling to reduce dimensionality
    Dense(256, activation='relu'),  # Add a fully connected layer with ReLU activation
    Dense(len(TARGET_CLASSES), activation='softmax')  # Add output layer with softmax activation for multi-class classification
])

# Compile model
optimizer = tf.keras.optimizers.Adam(learning_rate=1e-5)  # Initialize Adam optimizer with a low learning rate
model.compile(optimizer=optimizer, loss='categorical_crossentropy', metrics=['accuracy'])  # Compile the model with the optimizer, loss function, and metrics

# Class weights
class_weights = class_weight.compute_class_weight('balanced', classes=np.unique(np.argmax(y_train, axis=1)), y=np.argmax(y_train, axis=1))  # Compute class weights to handle class imbalance
class_weights_dict = dict(enumerate(class_weights))  # Convert class weights to a dictionary format for use in training

# Callbacks
```

```
early_stopping = EarlyStopping(monitor='val_loss', patience=10, restore_best_weights=True)  # Early stopping to prevent overfitting
reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.5, patience=5, min_lr=1e-6)  # Reduce learning rate when validation loss plateaus
model_checkpoint = ModelCheckpoint('best_model.keras', save_best_only=True, monitor='val_loss')  # Save the best model based on validation loss

# Training setup
batch_size = 8  # Set the batch size for training
train_gen = datagen.flow(X_train, y_train, batch_size=batch_size)  # Create a generator for training data augmentation

# Train the model
history = model.fit(  # Fit the model to the training data
    train_gen,  # Use the augmented training generator
    steps_per_epoch=len(X_train) // batch_size,  # Calculate steps per epoch based on batch size
    epochs=100,  # Set the number of training epochs
    validation_data=(X_val, y_val),  # Validate the model on the validation set
    class_weight=class_weights_dict,  # Use computed class weights during training
    callbacks=[early_stopping, reduce_lr, model_checkpoint]  # Include callbacks for training
)




# Evaluate the model
y_pred = np.argmax(model.predict(X_val), axis=1)  # Use the model to predict classes on the validation set and get the class indices with the highest probabilities
y_true = np.argmax(y_val, axis=1)  # Convert one-hot encoded true labels back to class indices

# Compute F1 scores for all classes
f1_per_class = f1_score(y_true, y_pred, average=None, labels=np.arange(len(TARGET_CLASSES)))  # Compute F1 scores for each class separately, specifying the target class labels
f1_weighted = f1_score(y_true, y_pred, average='weighted')  # Compute the weighted F1 score, considering the support of each class

# Print F1 scores
for i, class_name in enumerate(TARGET_CLASSES):  # Loop through each target class
    if i < len(f1_per_class):  # Check if F1 score is available for the class
        print(f"F1 Score for {class_name}: {f1_per_class[i]:.4f}")  # Print the F1 score for the class formatted to four decimal places
    else:
        print(f"F1 Score for {class_name}: Not applicable (no instances predicted)")  # Notify if there are no predicted instances for the class

```

```
print("Weighted F1 Score:", f1_weighted)  # Print the overall weighted F1 score



#Model testing phase.
import os
import numpy as np
import cv2
import tensorflow as tf
from tensorflow.keras.applications import MobileNetV2
from tensorflow.keras.layers import Dense, GlobalAveragePooling2D
from tensorflow.keras.models import Model
import matplotlib.pyplot as plt

# Constants
TARGET_CLASSES = ['Car', 'Cyclist', 'Pedestrian', 'Van', 'Truck']  # List of target classes for classification
image_size = (224, 224)  # Target size for images

# Function to build the model
def build_model():
    # Load the MobileNetV2 model without the top layer, using pretrained ImageNet weights
    base_model = MobileNetV2(weights='imagenet', include_top=False, input_shape=(224, 224, 3))
    for layer in base_model.layers[:-20]:  # Freeze most layers to retain pre-trained weights
        layer.trainable = False

    # Add a global average pooling layer and a dense layer for classification
    x = GlobalAveragePooling2D()(base_model.output)
    x = Dense(256, activation='relu')(x)  # Fully connected layer with ReLU activation
    outputs = Dense(len(TARGET_CLASSES), activation='softmax')(x)  # Output layer for multi-class classification

    model = Model(inputs=base_model.input, outputs=outputs)  # Create the model with specified inputs and outputs
    return model

# Load the model architecture
model = build_model()  # Instantiate the model

# Load the saved weights
try:
    model.load_weights('best_model.keras')  # Load the weights from the saved file
    print("Weights loaded successfully.")  # Confirmation message
except Exception as e:
    print("Error loading weights:", e)  # Error message if loading fails
```

```
185
186 # Print model summary to check its structure
187 model.summary()  # Display the architecture of the model
188
189 # Function to load and preprocess test images
190 def load_test_data(test_dir):
191     images = []  # Initialize list for images
192     image_files = []  # Initialize list for image filenames
193     for image_file in sorted(os.listdir(test_dir)):  # Iterate over files in
    the test directory
194         img = cv2.imread(os.path.join(test_dir, image_file))  # Read the image
    file
195         if img is None:  # Check if the image was loaded successfully
196             print(f"Warning: {image_file} could not be read. Skipping.")  #
    Warning for unreadable images
197             continue  # Skip to the next file
198         img = cv2.resize(img, image_size) / 255.0  # Resize and normalize the
    image
199         images.append(img)  # Add image to list
200         image_files.append(image_file)  # Store image filename for later
201     return np.array(images), image_files  # Return numpy array of images and
    list of filenames
202
203 # Load test data
204 test_dir = '/content/drive/My Drive/test_images/'  # Directory containing test
    images
205 X_test, image_files = load_test_data(test_dir)  # Load images and filenames
206
207 # Ensure the shape is correct
208 print(f"Test data shape: {X_test.shape}")  # Print the shape of the loaded test
    data
209
210 # Define bounding box scaling factors for each class
211 bounding_box_scale_factors = {
212     'Car': (0.4, 0.3),  # Scale factors for bounding box dimensions for each
    class
213     'Cyclist': (0.2, 0.4),
214     'Pedestrian': (0.1, 0.5),
215     'Van': (0.5, 0.3),
216     'Truck': (0.5, 0.4),
217 }
218
219 # Run predictions only if there are images to predict
220 if X_test.shape[0] > 0:  # Check if there are any test images
221     y_pred_probs = model.predict(X_test)  # Get prediction probabilities for
    the test images
222     y_pred = np.argmax(y_pred_probs, axis=1)  # Determine the predicted class
    indices
223
224     # Map predictions to class labels
```

```
    predicted_labels = [TARGET_CLASSES[i] for i in y_pred]  # Convert indices to class names

    # Display predicted labels for each test image
    for filename, label in zip(image_files, predicted_labels):  # Iterate through filenames and predicted labels
        print(f"{filename}: {label}")  # Print the filename and its predicted label

        # Load original image to draw the bounding box
        img = cv2.imread(os.path.join(test_dir, filename))  # Load the original image
        height, width, _ = img.shape  # Get image dimensions

        # Get bounding box scale based on the predicted class
        scale_x, scale_y = bounding_box_scale_factors[label]  # Retrieve scale factors for the predicted class

        # Calculate bounding box coordinates
        box_width = int(width * scale_x)  # Calculate the bounding box width
        box_height = int(height * scale_y)  # Calculate the bounding box height

        # Center the bounding box in the image
        x_min = (width - box_width) // 2  # Calculate the minimum x-coordinate
        y_min = (height - box_height) // 2  # Calculate the minimum y-coordinate
        x_max = x_min + box_width  # Calculate the maximum x-coordinate
        y_max = y_min + box_height  # Calculate the maximum y-coordinate

        # Draw bounding box and label on the image
        cv2.rectangle(img, (x_min, y_min), (x_max, y_max), (0, 0, 255), 2)  # Draw a red rectangle for the bounding box
        cv2.putText(img, label, (x_min, y_min - 10), cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 0, 255), 2)  # Add the label text above the bounding box

        # Display the image with bounding box
        plt.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))  # Convert BGR to RGB for displaying
        plt.axis('off')  # Turn off the axis
        plt.title(f"Predicted: {label}")  # Set the title to show the predicted label
        plt.show()  # Display the image
else:
    print("No valid test images to predict.")  # Message if no valid test images were found


```