Final Project
Neural Networks

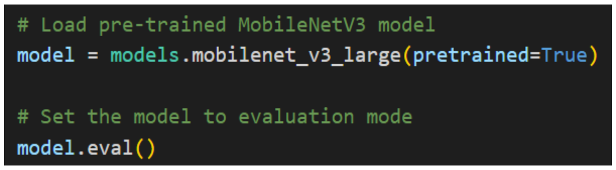
Ramez Mannaa - 209074491
Denis Kharenko - 324464536

Part 1

Sum of ID numbers = 209074491 + 324464536 = 533539027
The last digit is 7, therefore we will work with the MobileNet V3 backbone architecture.
Here's an example for how MobileNet V3 can help us classify images:
We used a pre-trained mobilenet V3 model to classify images from its own known object and classes We will go through the process step by step and see how we use this model in order to classify images.

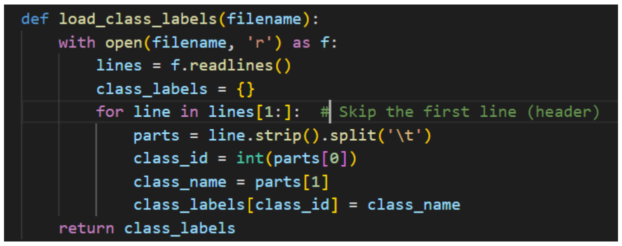
- Load Pre-trained MobileNetV3 Model - The code loads a pre-trained MobileNetV3 Large model using models.mobilenet_v3_large(pretrained=True).

- Set Model to Evaluation Mode: model.eval() sets the model to evaluation mode, which disables dropout and batch normalization layers.



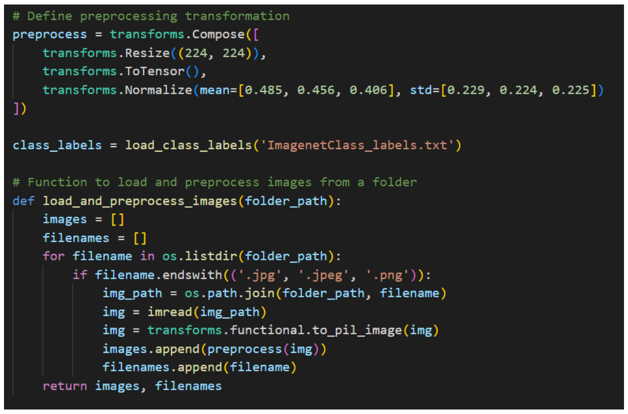
```
# Load pre-trained MobileNetV3 model
model = models.mobilenet_v3_large(pretrained=True)

# Set the model to evaluation mode
model.eval()
```

- Load class labels - load_class_labels function reads class labels from a text file (ImagenetClass_labels.txt) and returns them as a dictionary.

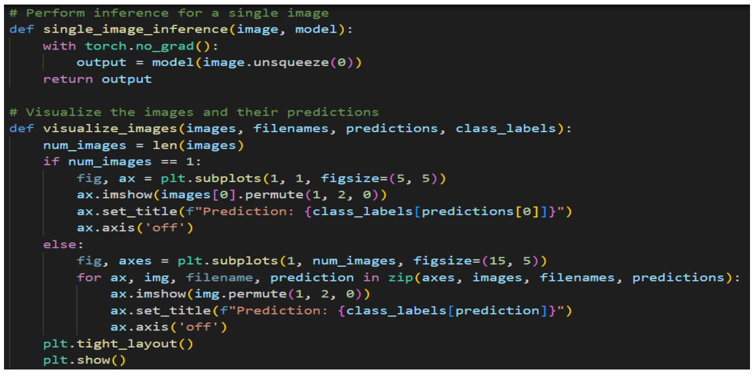


```
def load_class_labels(filename):
    with open(filename, 'r') as f:
        lines = f.readlines()
        class_labels = {}
        for line in lines[1:]:  # Skip the first line (header)
            parts = line.strip().split('\t')
            class_id = int(parts[0])
            class_name = parts[1]
            class_labels[class_id] = class_name
    return class_labels
```

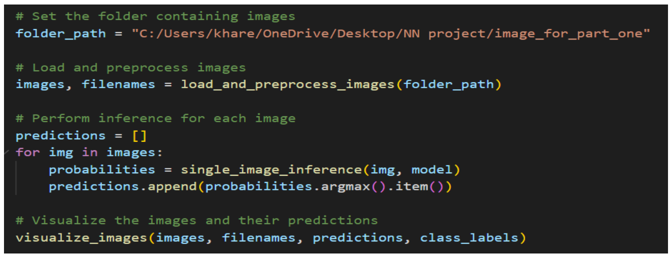
- Define preprocessing transformation - preprocess is a composition of image transformations (resizing, converting to tensor, and normalizing) using transforms.Compose.
- Load and preprocess image - load_and_preprocess_images function loads images from a folder (folder_path), preprocesses them using the defined transformation, and returns a list of preprocessed images and their filenames.


```python
# Define preprocessing transformation
preprocess = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])

class_labels = load_class_labels('ImagenetClass_labels.txt')

# Function to load and preprocess images from a folder
def load_and_preprocess_images(folder_path):
    images = []
    filenames = []
    for filename in os.listdir(folder_path):
        if filename.endswith(('.jpg', '.jpeg', '.png')):
            img_path = os.path.join(folder_path, filename)
            img = imread(img_path)
            img = transforms.functional.to_pil_image(img)
            images.append(preprocess(img))
            filenames.append(filename)
    return images, filenames
```

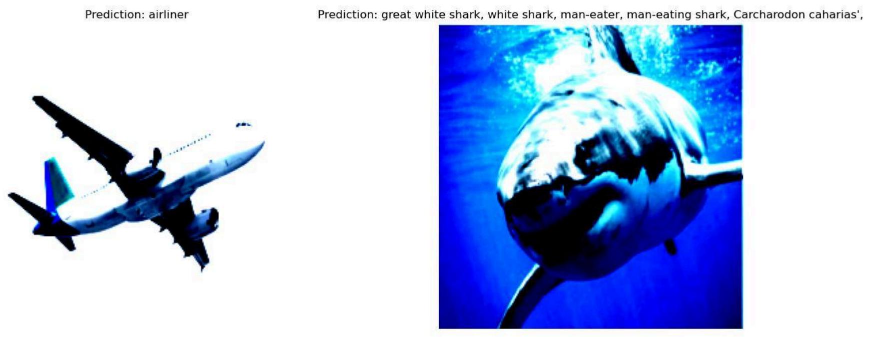
- Inference for a single image - single_image_inference function takes a preprocessed image and the model, performs inference, and returns the output probabilities.
- Visualize images and predictions - visualize_images function visualizes the preprocessed images along with their predicted labels. If there is only one image, it plots it in a single subplot. Otherwise, it plots multiple images in a row.


```python
# Perform inference for a single image
def single_image_inference(image, model):
    with torch.no_grad():
        output = model(image.unsqueeze(0))
    return output

# Visualize the images and their predictions
def visualize_images(images, filenames, predictions, class_labels):
    num_images = len(images)
    if num_images == 1:
        fig, ax = plt.subplots(1, 1, figsize=(5, 5))
        ax.imshow(images[0].permute(1, 2, 0))
        ax.set_title(f"Prediction: {class_labels[predictions[0]]}")
        ax.axis('off')
    else:
        fig, axes = plt.subplots(1, num_images, figsize=(15, 5))
        for ax, img, filename, prediction in zip(axes, images, filenames, predictions):
            ax.imshow(img.permute(1, 2, 0))
            ax.set_title(f"Prediction: {class_labels[prediction]}")
            ax.axis('off')
    plt.tight_layout()
    plt.show()
```

- Load and preprocess the images - after obtaining the folder containing the images, load_and_preprocess_images is called to load and preprocess the images from the specified folder.
- Inference for each image - For each preprocessed image, inference is performed using the single_image_inference function, and the predicted class index is appended to the predictions list.
- Visualize images and predictions - visualize_images is called to plot the preprocessed images along with their predicted labels.


```python
# Set the folder containing images
folder_path = "C:/Users/khare/OneDrive/Desktop/NN project/image_for_part_one"

# Load and preprocess images
images, filenames = load_and_preprocess_images(folder_path)

# Perform inference for each image
predictions = []
for img in images:
    probabilities = single_image_inference(img, model)
    predictions.append(probabilities.argmax().item())

# Visualize the images and their predictions
visualize_images(images, filenames, predictions, class_labels)
```

This is the output of the classification:


Prediction: airliner
Prediction: great white shark, white shark, man-eater, man-eating shark, Carcharodon caharias',

<u>About MobileNet V3</u>
MobileNet V3 is a convolutional neural network architecture specifically designed for efficient inference on mobile and embedded devices. It expands on the success of its predecessors, MobileNet and MobileNet V2, by introducing better architectural elements and optimization techniques to develop and improve the performance, efficiency, and versatility. Developed by Google researchers, MobileNet V3 targets a wide range of applications, from image classification to object detection and semantic segmentation, making it a valuable tool for many different real-world scenarios.

MobileNet V3 architecture consists of several key components which add to its efficiency and effectiveness:

- Backbone network - MobileNet V3 utilizes a streamlined backbone network based on efficient (in terms of depth) separable convolutions. This architecture reduces the number of parameters and computational complexity while maintaining competitive performance.
- Inverted residual blocks - MobileNet V3 incorporates inverted residual blocks with linear bottleneck and shortcut connections. These blocks allow for efficient information flow through the network while minimizing computational cost.
- Efficient last layer - MobileNet V3 introduces efficient last layers, including a squeeze-and-excitation module and hard-swish activation function. These layers enhance feature representation and promote better model generalization.
- Network design choices - MobileNet V3 fulfills various design choices, such as efficient up-sampling methods, network width and depth scaling, and improved kernel size selection. These choices optimize model performance across different resource constraints and target applications.

Overall, the architecture of MobileNet V3 is very similar to V2 but with different and major improvements.

MobileNet V3 offers several notable features and advantages:

- High efficiency - With its lightweight architecture and efficient operations, MobileNet V3 enables fast and low-power inference on resource-constrained devices, including smartphones, IoT devices, and embedded systems.
- Versatility - MobileNet V3 is versatile and can be customized for various computer vision tasks, including image classification, object detection, semantic segmentation, and more. Its modular design allows for easy adaptation to different application requirements.
- Excellent performance - Despite its compact size, MobileNet V3 achieves state-of-the-art performance on standard benchmark datasets, demonstrating competitive accuracy compared to larger and more computationally intensive models.
- Scalability - MobileNet V3 is designed to scale efficiently across different hardware platforms and deployment scenarios. It supports model compression techniques such as quantization, pruning, and knowledge distillation, enabling deployment on a wide range of devices with varying computational resources.

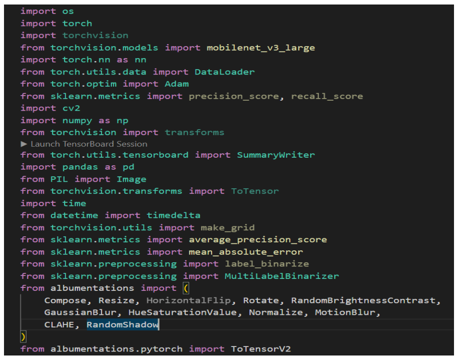
MobileNet V3 finds applications across various domains and industries:

- Mobile and edge detection - MobileNet V3 is widely used for on-device AI applications, including image recognition, object detection, and scene understanding on smartphones, tablets, and other mobile devices.
- IoT and embedded systems - In IoT and embedded systems, MobileNet V3 enables intelligent edge computing for tasks such as smart surveillance, environmental monitoring, and industrial automation.
- Autonomous vehicles - MobileNet V3 contributes to the development of autonomous vehicles by providing efficient solutions for real-time object detection, road scene analysis, and driver assistance systems.
- Healthcare and biomedical imaging - In healthcare, MobileNet V3 facilitates medical image analysis, disease diagnosis, and telemedicine applications by running lightweight models directly on medical devices or edge servers.
- Retail and Ecommerce - MobileNet V3 powers visual search, product recommendation, and augmented reality experiences in retail and e-commerce applications, enhancing user engagement and shopping experiences.

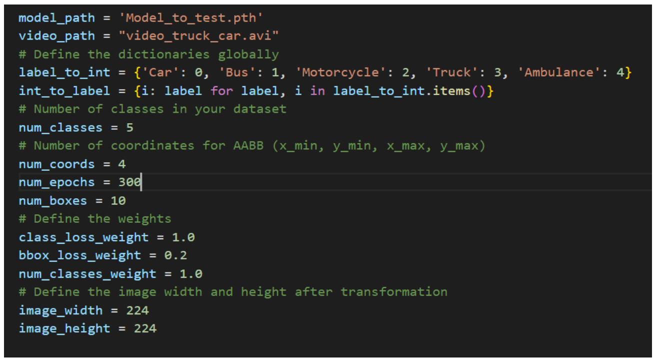
MobileNet V3 represents a significant advancement in deep learning architecture for mobile and embedded deployment. Its efficient design, versatile capabilities, and wide range of applications make it a valuable tool for developers, researchers, and practitioners seeking to deploy AI solutions on resource-constrained devices. With ongoing advancements and optimizations, MobileNet V3 continues to push the boundaries of performance, efficiency, and accessibility in the field of computer vision and beyond.

Part2

We implemented a code for object detection using a custom dataset with bounding box annotations. Let's break down the key components and functionalities of the code.
We start by importing the relevant libraries, and defining global and constant variables.
We selected the Vehicles dataset from roboflow, it contains images from 5 different classes, each one represents a different vehicle.

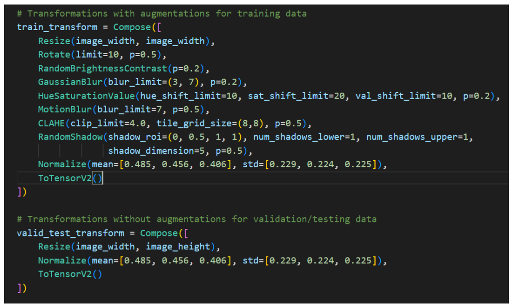


```python
import os
import torch
import torchvision
from torchvision.models import mobilenet_v3_large
import torch.nn as nn
from torch.utils.data import DataLoader
from torch.optim import Adam
from sklearn.metrics import precision_score, recall_score
import cv2
import numpy as np
from torchvision import transforms
▶ Launch TensorBoard Session
from torch.utils.tensorboard import SummaryWriter
import pandas as pd
from PIL import Image
from torchvision.transforms import ToTensor
import time
from datetime import timedelta
from torchvision.utils import make_grid
from sklearn.metrics import average_precision_score
from sklearn.metrics import mean_absolute_error
from sklearn.preprocessing import label_binarize
from sklearn.preprocessing import MultiLabelBinarizer
from albumentations import (
    Compose, Resize, HorizontalFlip, Rotate, RandomBrightnessContrast,
    GaussianBlur, HueSaturationValue, Normalize, MotionBlur,
    CLAHE, RandomShadow
)
from albumentations.pytorch import ToTensorV2
```

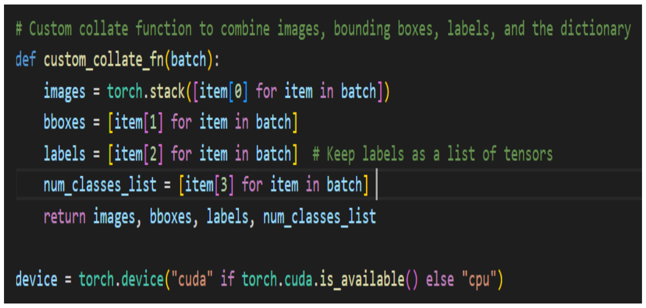


```python
model_path = 'Model_to_test.pth'
video_path = "video_truck_car.avi"
# Define the dictionaries globally
label_to_int = {'Car': 0, 'Bus': 1, 'Motorcycle': 2, 'Truck': 3, 'Ambulance': 4}
int_to_label = {i: label for label, i in label_to_int.items()}
# Number of classes in your dataset
num_classes = 5
# Number of coordinates for AABB (x_min, y_min, x_max, y_max)
num_coords = 4
num_epochs = 300
num_boxes = 10
# Define the weights
class_loss_weight = 1.0
bbox_loss_weight = 0.2
num_classes_weight = 1.0
# Define the image width and height after transformation
image_width = 224
image_height = 224
```

Now we define the train and valid transform augmentations, and the custom collate function:



```python
# Transformations with augmentations for training data
train_transform = Compose([
    Resize(image_width, image_width),
    Rotate(limit=10, p=0.5),
    RandomBrightnessContrast(p=0.2),
    GaussianBlur(blur_limit=(3, 7), p=0.2),
    HueSaturationValue(hue_shift_limit=10, sat_shift_limit=20, val_shift_limit=10, p=0.2),
    MotionBlur(blur_limit=7, p=0.5),
    CLAHE(clip_limit=4.0, tile_grid_size=(8,8), p=0.5),
    RandomShadow(shadow_roi=(0, 0.5, 1, 1), num_shadows_lower=1, num_shadows_upper=1,
                 shadow_dimension=5, p=0.5),
    Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
    ToTensorV2()
])

# Transformations without augmentations for validation/testing data
valid_test_transform = Compose([
    Resize(image_width, image_height),
    Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
    ToTensorV2()
])
```



```python
# Custom collate function to combine images, bounding boxes, labels, and the dictionary
def custom_collate_fn(batch):
    images = torch.stack([item[0] for item in batch])
    bboxes = [item[1] for item in batch]
    labels = [item[2] for item in batch]  # Keep labels as a list of tensors
    num_classes_list = [item[3] for item in batch]
    return images, bboxes, labels, num_classes_list

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

Train transform - Augments training images with various transformations like resizing, rotation, brightness/contrast adjustments, blur, color shifts, motion blur, CLAHE (Contrast Limited Adaptive Histogram Equalization), random shadows, normalization, and conversion to PyTorch tensors.
Valid test transform - Prepares validation/testing images with resizing, normalization, and conversion to PyTorch tensors.

The augmentations we used are:

- Resize: Resizes the image to a specified size(specifically The size 224X224 that our backbone accepts)
- Rotate: Randomly rotates the image within a specified limit.
- RandomBrightnessContrast - Randomly adjusts the brightness and contrast of the image.
- GaussianBlur: Applies Gaussian blur to the image with a random kernel size within a specified range.
- HueSaturationValue: Randomly adjusts the hue, saturation, and value of the image.
- MotionBlur: Applies motion blur to the image with a random kernel size within a specified range.
- CLAHE (Contrast Limited Adaptive Histogram Equalization): Enhances the local contrast of the image.
- RandomShadow: Adds random shadows to the image within a specified region.
- Normalize: Normalizes the image pixel values.
- ToTensorV2: Converts the image to a PyTorch tensor.

These augmentations help in increasing the diversity of the training data, improving the model's ability to generalize to unseen images and conditions commonly encountered in real-world vehicle detection scenarios.

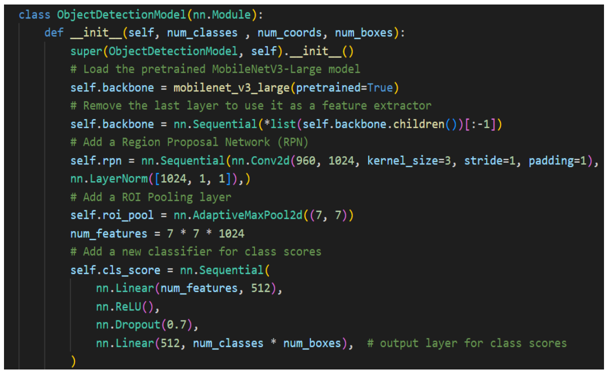
Custom collate - Defines a custom function to organize batches of data. It stacks images into a single tensor, keeps bounding boxes and labels as lists of tensors, and includes a list of the number of classes.

Device - Determines the computing device (CPU or GPU) available for running the model, preferring GPU if available.
Now we will go over the ObjectDetectionModel class:
__init__ method: This method is the constructor of the class where the model architecture is defined. Here's a breakdown of its components:

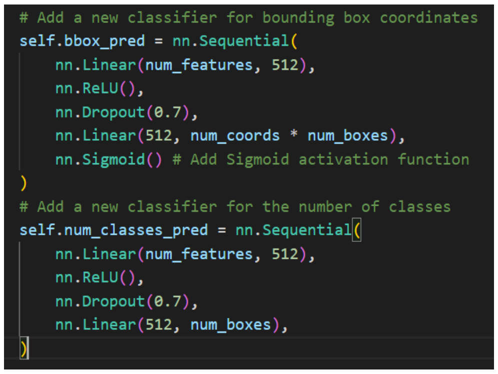
- super(ObjectDetectionModel, self).__init__(): This line calls the constructor of the parent class (nn.Module) to initialize the object detection model.
- backbone: This attribute holds the backbone feature extractor of the model. In this implementation, it loads the pretrained MobileNetV3-Large model and removes its last layer to use it as a feature extractor.
- rpn: This attribute represents the Region Proposal Network (RPN) module. It applies a convolutional layer followed by layer normalization.
- roi_pool: This attribute is an adaptive max pooling layer used for Region of Interest (ROI) pooling.

- cls_score: This attribute is a classifier for predicting class scores. It consists of fully connected layers followed by ReLU activation and dropout, and ends with a linear layer outputting scores for each class and bounding box combination.
- bbox_pred: This attribute is a regressor for predicting bounding box coordinates. Similar to cls_score, it consists of fully connected layers, ReLU activation, dropout, and a linear layer with a sigmoid activation function.
- num_classes_pred: This attribute predicts the number of classes present in the image. It follows a similar structure to cls_score and bbox_pred.

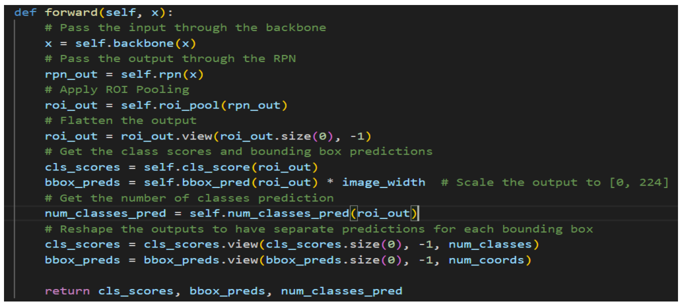
Forward method - This method defines the forward pass of the model. It takes input data x and passes it through the defined layers of the model. The output consists of class scores (cls_scores), bounding box predictions (bbox_preds), and the number of class predictions (num_classes_pred). These outputs are returned by the method.



```python
class ObjectDetectionModel(nn.Module):
    def __init__(self, num_classes , num_coords, num_boxes):
        super(ObjectDetectionModel, self).__init__()
        # Load the pretrained MobileNetV3-Large model
        self.backbone = mobilenet_v3_large(pretrained=True)
        # Remove the last layer to use it as a feature extractor
        self.backbone = nn.Sequential(*list(self.backbone.children())[:-1])
        # Add a Region Proposal Network (RPN)
        self.rpn = nn.Sequential(nn.Conv2d(960, 1024, kernel_size=3, stride=1, padding=1),
        nn.LayerNorm([1024, 1, 1]),)
        # Add a ROI Pooling layer
        self.roi_pool = nn.AdaptiveMaxPool2d((7, 7))
        num_features = 7 * 7 * 1024
        # Add a new classifier for class scores
        self.cls_score = nn.Sequential(
            nn.Linear(num_features, 512),
            nn.ReLU(),
            nn.Dropout(0.7),
            nn.Linear(512, num_classes * num_boxes),  # output layer for class scores
        )
```



```python
# Add a new classifier for bounding box coordinates
self.bbox_pred = nn.Sequential(
    nn.Linear(num_features, 512),
    nn.ReLU(),
    nn.Dropout(0.7),
    nn.Linear(512, num_coords * num_boxes),
    nn.Sigmoid() # Add Sigmoid activation function
)
# Add a new classifier for the number of classes
self.num_classes_pred = nn.Sequential(
    nn.Linear(num_features, 512),
    nn.ReLU(),
    nn.Dropout(0.7),
    nn.Linear(512, num_boxes),
)
```

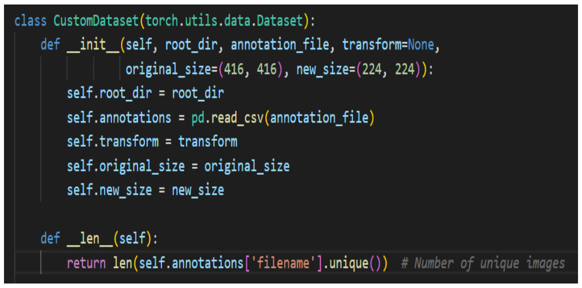


```python
def forward(self, x):
    # Pass the input through the backbone
    x = self.backbone(x)
    # Pass the output through the RPN
    rpn_out = self.rpn(x)
    # Apply ROI Pooling
    roi_out = self.roi_pool(rpn_out)
    # Flatten the output
    roi_out = roi_out.view(roi_out.size(0), -1)
    # Get the class scores and bounding box predictions
    cls_scores = self.cls_score(roi_out)
    bbox_preds = self.bbox_pred(roi_out) * image_width  # Scale the output to [0, 224]
    # Get the number of classes prediction
    num_classes_pred = self.num_classes_pred(roi_out)
    # Reshape the outputs to have separate predictions for each bounding box
    cls_scores = cls_scores.view(cls_scores.size(0), -1, num_classes)
    bbox_preds = bbox_preds.view(bbox_preds.size(0), -1, num_coords)

    return cls_scores, bbox_preds, num_classes_pred
```

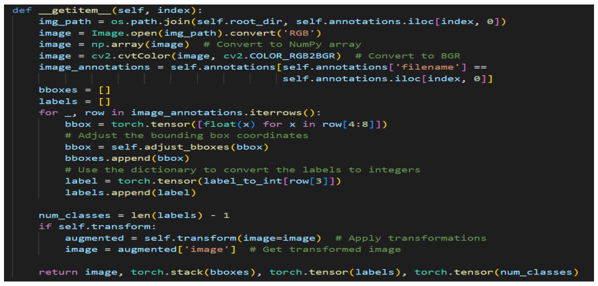
Now we will go over the CustomDatasetclass:

- **__init__ method:** Initializes the dataset object with the root directory, annotation file, transformation functions (optional), original and new image sizes.
- **__len__ method:** Returns the total number of unique images in the dataset.
- **__getitem__ method:** Retrieves an item (image and its corresponding annotations) from the dataset at a given index. It reads the image, converts it to the RGB format, retrieves annotations, adjusts bounding box coordinates, applies transformations (if specified), and returns the image, bounding boxes, labels, and the number of classes.

This adjust_bboxes function rescales the coordinates of a bounding box (bbox) from the original image size to a new size. It calculates scale factors for both the x and y dimensions based on the original and new image sizes. Then, it multiplies the coordinates of the bounding box by these scale factors to obtain the adjusted bounding box coordinates (adjusted_bbox). Finally, it returns the adjusted bounding box.



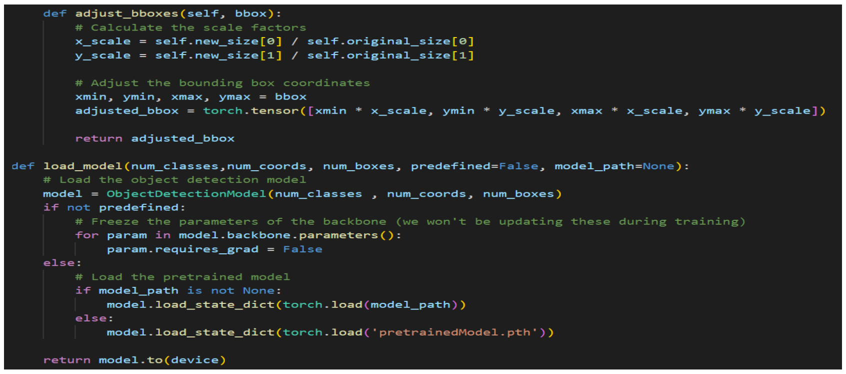
```python
class CustomDataset(torch.utils.data.Dataset):
    def __init__(self, root_dir, annotation_file, transform=None,
                 original_size=(416, 416), new_size=(224, 224)):
        self.root_dir = root_dir
        self.annotations = pd.read_csv(annotation_file)
        self.transform = transform
        self.original_size = original_size
        self.new_size = new_size

    def __len__(self):
        return len(self.annotations['filename'].unique()) # Number of unique images
```



```python
def __getitem__(self, index):
    img_path = os.path.join(self.root_dir, self.annotations.iloc[index, 0])
    image = Image.open(img_path).convert('RGB')
    image = np.array(image)  # Convert to NumPy array
    image = cv2.cvtColor(image, cv2.COLOR_RGB2BGR)  # Convert to BGR
    image_annotations = self.annotations[self.annotations['filename'] ==
                                         self.annotations.iloc[index, 0]]
    bboxes = []
    labels = []
    for _, row in image_annotations.iterrows():
        bbox = torch.tensor([float(x) for x in row[4:8]])
        # Adjust the bounding box coordinates
        bbox = self.adjust_bboxes(bbox)
        bboxes.append(bbox)
        # Use the dictionary to convert the labels to integers
        label = torch.tensor(label_to_int[row[3]])
        labels.append(label)

    num_classes = len(labels) - 1
    if self.transform:
        augmented = self.transform(image=image)  # Apply transformations
        image = augmented['image']  # Get transformed image

    return image, torch.stack(bboxes), torch.tensor(labels), torch.tensor(num_classes)
```

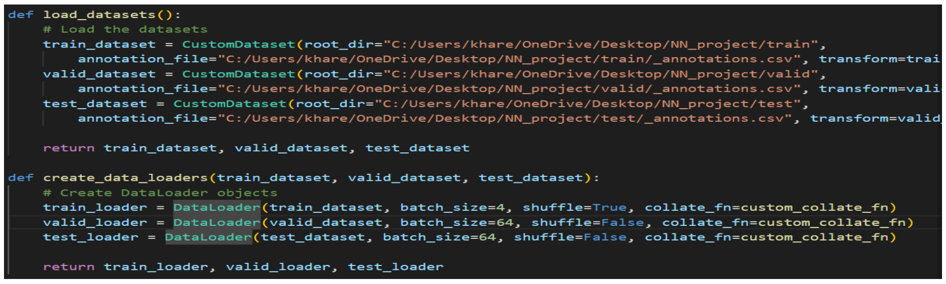
Now let's go over the functions relevant to loading the dataset.

load_model function: It loads an object detection model with a specified number of classes, coordinates, and boxes.
If predefined is False, it freezes the parameters of the backbone (feature extractor) to prevent them from being updated during training.
If predefined is True and a model_path is provided, it loads a pretrained model from the specified path or from a default path ('pretrainedModel.pth').
Finally, it returns the model moved to the specified device (CPU or GPU).



```python
    def adjust_bboxes(self, bbox):
        # Calculate the scale factors
        x_scale = self.new_size[0] / self.original_size[0]
        y_scale = self.new_size[1] / self.original_size[1]

        # Adjust the bounding box coordinates
        xmin, ymin, xmax, ymax = bbox
        adjusted_bbox = torch.tensor([xmin * x_scale, ymin * y_scale, xmax * x_scale, ymax * y_scale])

        return adjusted_bbox

def load_model(num_classes,num_coords, num_boxes, predefined=False, model_path=None):
    # Load the object detection model
    model = ObjectDetectionModel(num_classes , num_coords, num_boxes)
    if not predefined:
        # Freeze the parameters of the backbone (we won't be updating these during training)
        for param in model.backbone.parameters():
            param.requires_grad = False
    else:
        # Load the pretrained model
        if model_path is not None:
            model.load_state_dict(torch.load(model_path))
        else:
            model.load_state_dict(torch.load('pretrainedModel.pth'))

    return model.to(device)
```

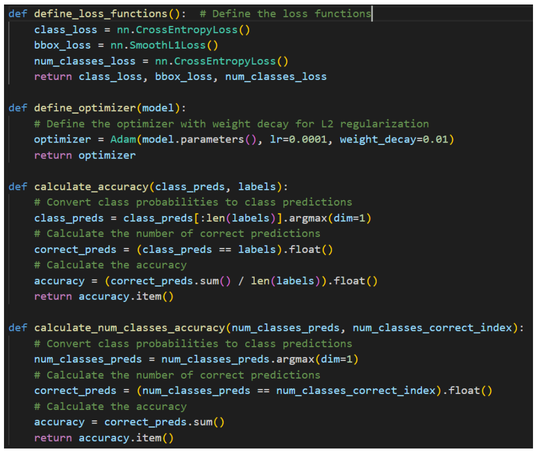
load_datasets(): This function loads three datasets—train, validation, and test. It creates instances of the CustomDataset class for each dataset, specifying the root directory where images are stored, the annotation file containing bounding box annotations, and the transformation to be applied to the images.
create_data_loaders(train_dataset, valid_dataset, test_dataset): This function creates DataLoader objects for the train, validation, and test datasets. It specifies parameters such as batch size, shuffle (whether to shuffle the data), and collate function (how to collate individual samples into batches). It returns DataLoader objects for each dataset.



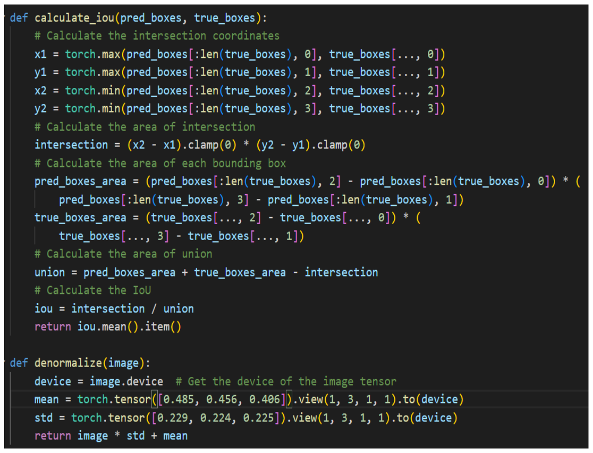
```
def load_datasets():
    # Load the datasets
    train_dataset = CustomDataset(root_dir="C:/Users/khare/OneDrive/Desktop/NN_project/train",
        annotation_file="C:/Users/khare/OneDrive/Desktop/NN_project/train/_annotations.csv", transform=trai
    valid_dataset = CustomDataset(root_dir="C:/Users/khare/OneDrive/Desktop/NN_project/valid",
        annotation_file="C:/Users/khare/OneDrive/Desktop/NN_project/valid/_annotations.csv", transform=vali
    test_dataset = CustomDataset(root_dir="C:/Users/khare/OneDrive/Desktop/NN_project/test",
        annotation_file="C:/Users/khare/OneDrive/Desktop/NN_project/test/_annotations.csv", transform=valid

    return train_dataset, valid_dataset, test_dataset

def create_data_loaders(train_dataset, valid_dataset, test_dataset):
    # Create DataLoader objects
    train_loader = DataLoader(train_dataset, batch_size=4, shuffle=True, collate_fn=custom_collate_fn)
    valid_loader = DataLoader(valid_dataset, batch_size=64, shuffle=False, collate_fn=custom_collate_fn)
    test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False, collate_fn=custom_collate_fn)

    return train_loader, valid_loader, test_loader
```

Now we will go over calculating functions and talk about their use and the math

- **define_loss_functions():** This function defines three loss functions—classification loss (class_loss), bounding box regression loss (bbox_loss), and loss for predicting the number of classes (num_classes_loss). nn.CrossEntropyLoss() is used for classification loss, while nn.SmoothL1Loss() is used for bounding box regression loss. num_classes_loss is also a classification loss.
- **define_optimizer(model):** This function defines an optimizer for training the model. It uses the Adam optimizer with a learning rate of 0.0001 and weight decay for L2 regularization.
- **calculate_accuracy(class_preds, labels):** This function calculates the accuracy of classification predictions. It compares the predicted class probabilities (class_preds) with the ground truth labels (labels) and computes the accuracy as the proportion of correct predictions.
- **calculate_num_classes_accuracy(num_classes_preds, num_classes_correct_index):** This function calculates the accuracy of predicted number of classes. It compares the predicted number of classes (num_classes_preds) with the correct index of the number of classes (num_classes_correct_index) and computes the accuracy as the proportion of correct predictions.
- **calculate_iou(pred_boxes, true_boxes):** This function calculates the Intersection over Union (IoU) between predicted bounding boxes (pred_boxes) and ground truth bounding boxes (true_boxes). IoU is computed as the ratio of the area of overlap between the predicted and ground truth bounding boxes to the area of union between them. It is a measure of the spatial overlap between two bounding boxes and is commonly used in object detection tasks.
- **The denormalize function:** Reverses the normalization process applied to an image by scaling each pixel value. It takes a normalized image tensor and returns a denormalized image tensor by multiplying the normalized image tensor by the standard deviation tensor and adding the mean tensor. This effectively brings the image back to its original scale.

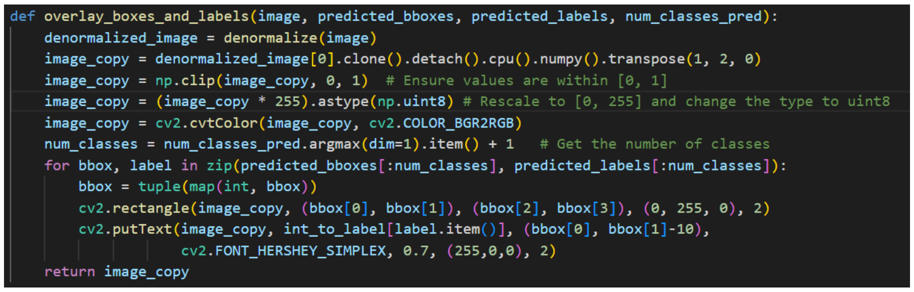


```python
def define_loss_functions():  # Define the loss functions
    class_loss = nn.CrossEntropyLoss()
    bbox_loss = nn.SmoothL1Loss()
    num_classes_loss = nn.CrossEntropyLoss()
    return class_loss, bbox_loss, num_classes_loss

def define_optimizer(model):
    # Define the optimizer with weight decay for L2 regularization
    optimizer = Adam(model.parameters(), lr=0.0001, weight_decay=0.01)
    return optimizer

def calculate_accuracy(class_preds, labels):
    # Convert class probabilities to class predictions
    class_preds = class_preds[:len(labels)].argmax(dim=1)
    # Calculate the number of correct predictions
    correct_preds = (class_preds == labels).float()
    # Calculate the accuracy
    accuracy = (correct_preds.sum() / len(labels)).float()
    return accuracy.item()

def calculate_num_classes_accuracy(num_classes_preds, num_classes_correct_index):
    # Convert class probabilities to class predictions
    num_classes_preds = num_classes_preds.argmax(dim=1)
    # Calculate the number of correct predictions
    correct_preds = (num_classes_preds == num_classes_correct_index).float()
    # Calculate the accuracy
    accuracy = correct_preds.sum()
    return accuracy.item()
```



```python
def calculate_iou(pred_boxes, true_boxes):
    # Calculate the intersection coordinates
    x1 = torch.max(pred_boxes[:len(true_boxes), 0], true_boxes[..., 0])
    y1 = torch.max(pred_boxes[:len(true_boxes), 1], true_boxes[..., 1])
    x2 = torch.min(pred_boxes[:len(true_boxes), 2], true_boxes[..., 2])
    y2 = torch.min(pred_boxes[:len(true_boxes), 3], true_boxes[..., 3])
    # Calculate the area of intersection
    intersection = (x2 - x1).clamp(0) * (y2 - y1).clamp(0)
    # Calculate the area of each bounding box
    pred_boxes_area = (pred_boxes[:len(true_boxes), 2] - pred_boxes[:len(true_boxes), 0]) * (
        pred_boxes[:len(true_boxes), 3] - pred_boxes[:len(true_boxes), 1])
    true_boxes_area = (true_boxes[..., 2] - true_boxes[..., 0]) * (
        true_boxes[..., 3] - true_boxes[..., 1])
    # Calculate the area of union
    union = pred_boxes_area + true_boxes_area - intersection
    # Calculate the IoU
    iou = intersection / union
    return iou.mean().item()

def denormalize(image):
    device = image.device  # Get the device of the image tensor
    mean = torch.tensor([0.485, 0.456, 0.406]).view(1, 3, 1, 1).to(device)
    std = torch.tensor([0.229, 0.224, 0.225]).view(1, 3, 1, 1).to(device)
    return image * std + mean
```

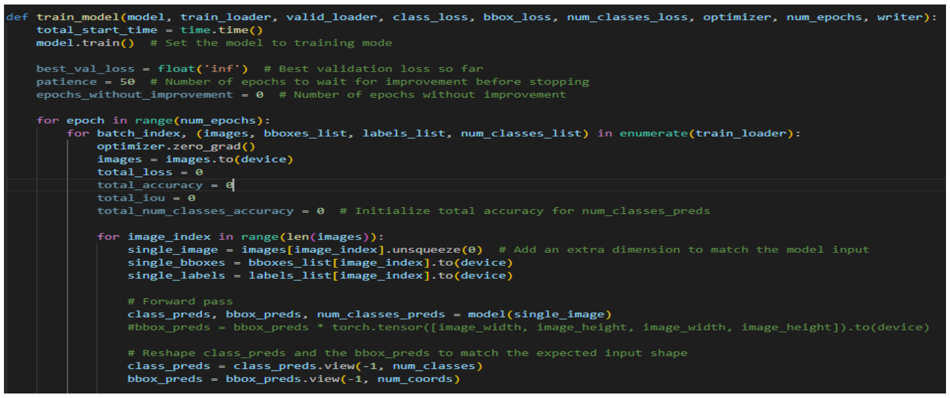
The overlay_boxes_and_labels function overlays predicted bounding boxes and labels on the denormalized image. It first denormalizes the image using the ‘denormalize’ function, converts it to a NumPy array, and rescales it. Then, it draws bounding boxes and adds labels using OpenCV (cv2). The number of classes is determined from num_classes_pred. The function utilizes int_to_label to convert integer labels to class names.



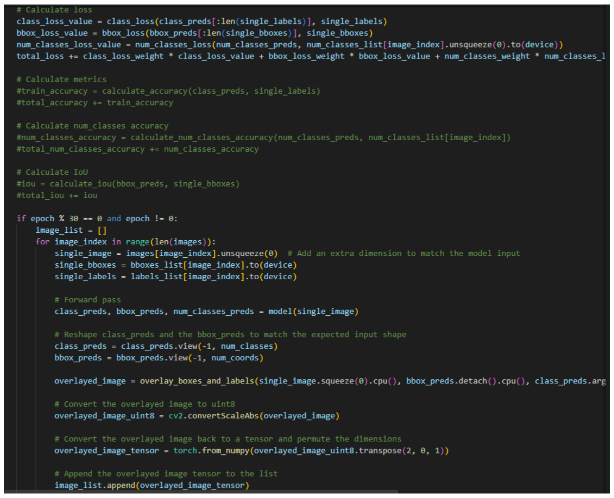
```python
def overlay_boxes_and_labels(image, predicted_bboxes, predicted_labels, num_classes_pred):
    denormalized_image = denormalize(image)
    image_copy = denormalized_image[0].clone().detach().cpu().numpy().transpose(1, 2, 0)
    image_copy = np.clip(image_copy, 0, 1)  # Ensure values are within [0, 1]
    image_copy = (image_copy * 255).astype(np.uint8) # Rescale to [0, 255] and change the type to uint8
    image_copy = cv2.cvtColor(image_copy, cv2.COLOR_BGR2RGB)
    num_classes = num_classes_pred.argmax(dim=1).item() + 1   # Get the number of classes
    for bbox, label in zip(predicted_bboxes[:num_classes], predicted_labels[:num_classes]):
        bbox = tuple(map(int, bbox))
        cv2.rectangle(image_copy, (bbox[0], bbox[1]), (bbox[2], bbox[3]), (0, 255, 0), 2)
        cv2.putText(image_copy, int_to_label[label.item()], (bbox[0], bbox[1]-10),
                    cv2.FONT_HERSHEY_SIMPLEX, 0.7, (255,0,0), 2)
    return image_copy
```

Using Axis-Aligned Bounding Boxes (AABBs) in object detection offers mathematical simplicity and practical efficiency. Mathematically, AABBs are straightforward to represent and compute, making them efficient for collision detection and containment tests. They provide a conservative bounding volume and are computationally efficient for intersection and union operations. In the context of the model, AABBs are suitable for handling multiple labels due to their ability to encapsulate objects of various shapes and orientations efficiently. They contribute to faster training and inference, scalability across diverse datasets, and adaptability to objects of different sizes and aspect ratios. Overall, AABBs provide a robust and efficient solution for bounding box representation in object detection models.

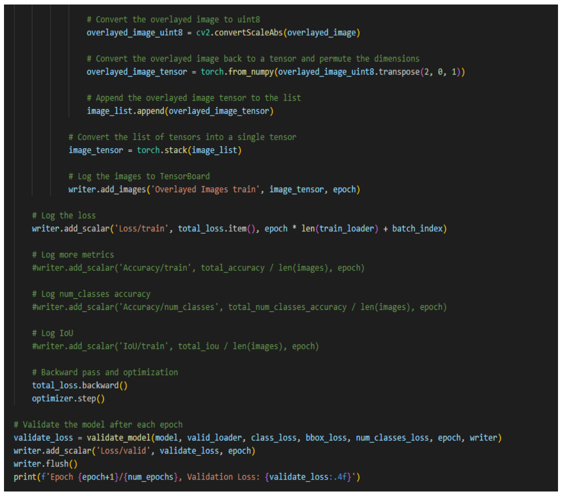
Train model: The train function iterates through epochs, optimizing the object detection model using training data. It calculates losses for classification, bounding box prediction, and number of classes prediction which are the three things that our model tries to predict. It also logs training loss and optionally overlayed images to TensorBoard. After each epoch, it validates the model's performance on a separate validation set and logs validation loss. The function implements early stopping based on validation loss to prevent overfitting. Finally, it saves the best-performing model and records total training time, also we added to the option to calculate the IOU and the accuracy for the number of classes predicted. IoU measures the overlap between predicted bounding boxes and ground truth boxes, providing insights into how accurately the model localizes objects. Accuracy for the number of classes predicted evaluates how accurately the model predicts the number of objects in an image, which is crucial for object detection tasks. We chose a batch size of 4 because we noticed a better generalization that way, also we chose a learning rate of 0.0001 because it produced the best results on the train set. We used dropout, normalization layers in the models architecture to avoid overfitting and also augmentations from the Albumentations library.



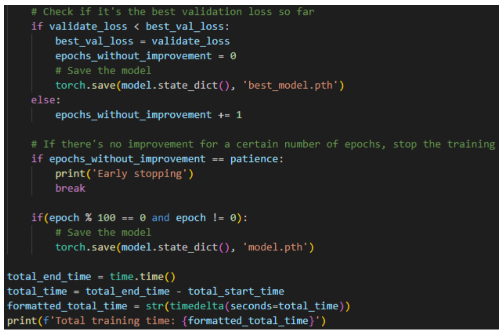
```python
def train_model(model, train_loader, valid_loader, class_loss, bbox_loss, num_classes_loss, optimizer, num_epochs, writer):
    total_start_time = time.time()
    model.train()  # Set the model to training mode

    best_val_loss = float('inf')  # Best validation loss so far
    patience = 50  # Number of epochs to wait for improvement before stopping
    epochs_without_improvement = 0  # Number of epochs without improvement

    for epoch in range(num_epochs):
        for batch_index, (images, bboxes_list, labels_list, num_classes_list) in enumerate(train_loader):
            optimizer.zero_grad()
            images = images.to(device)
            total_loss = 0
            total_accuracy = 0
            total_iou = 0
            total_num_classes_accuracy = 0  # Initialize total accuracy for num_classes_preds

            for image_index in range(len(images)):
                single_image = images[image_index].unsqueeze(0)  # Add an extra dimension to match the model input
                single_bboxes = bboxes_list[image_index].to(device)
                single_labels = labels_list[image_index].to(device)

                # Forward pass
                class_preds, bbox_preds, num_classes_preds = model(single_image)
                #bbox_preds = bbox_preds * torch.tensor([image_width, image_height, image_width, image_height]).to(device)

                # Reshape class_preds and the bbox_preds to match the expected input shape
                class_preds = class_preds.view(-1, num_classes)
                bbox_preds = bbox_preds.view(-1, num_coords)
```



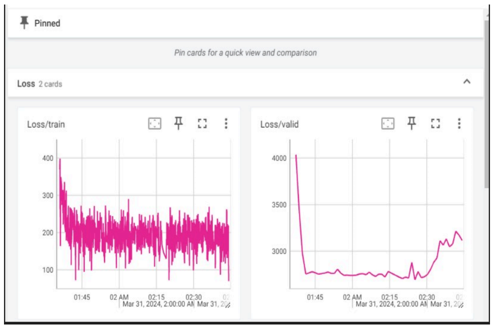
```python
# Calculate loss
class_loss_value = class_loss(class_preds[:len(single_labels)], single_labels)
bbox_loss_value = bbox_loss(bbox_preds[:len(single_bboxes)], single_bboxes)
num_classes_loss_value = num_classes_loss(num_classes_preds, num_classes_list[image_index].unsqueeze(0).to(device))
total_loss += class_loss_weight * class_loss_value + bbox_loss_weight * bbox_loss_value + num_classes_weight * num_classes_l

# Calculate metrics
#train_accuracy = calculate_accuracy(class_preds, single_labels)
#total_accuracy += train_accuracy

# Calculate num_classes accuracy
#num_classes_accuracy = calculate_num_classes_accuracy(num_classes_preds, num_classes_list[image_index])
#total_num_classes_accuracy += num_classes_accuracy

# Calculate IoU
#iou = calculate_iou(bbox_preds, single_bboxes)
#total_iou += iou

if epoch % 30 == 0 and epoch != 0:
    image_list = []
    for image_index in range(len(images)):
        single_image = images[image_index].unsqueeze(0)  # Add an extra dimension to match the model input
        single_bboxes = bboxes_list[image_index].to(device)
        single_labels = labels_list[image_index].to(device)

        # Forward pass
        class_preds, bbox_preds, num_classes_preds = model(single_image)

        # Reshape class_preds and the bbox_preds to match the expected input shape
        class_preds = class_preds.view(-1, num_classes)
        bbox_preds = bbox_preds.view(-1, num_coords)

        overlayed_image = overlay_boxes_and_labels(single_image.squeeze(0).cpu(), bbox_preds.detach().cpu(), class_preds.arg

        # Convert the overlayed image to uint8
        overlayed_image_uint8 = cv2.convertScaleAbs(overlayed_image)

        # Convert the overlayed image back to a tensor and permute the dimensions
        overlayed_image_tensor = torch.from_numpy(overlayed_image_uint8.transpose(2, 0, 1))

        # Append the overlayed image tensor to the list
        image_list.append(overlayed_image_tensor)
```



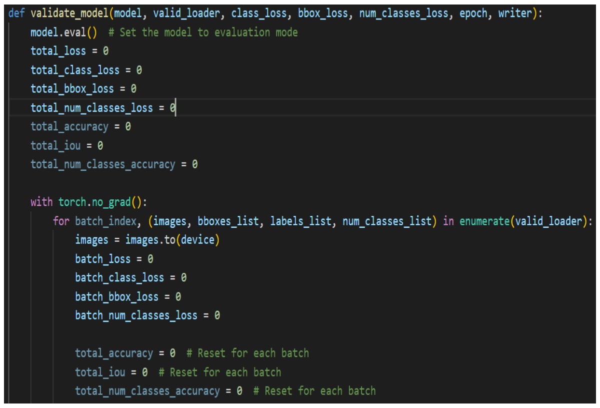
```python
            # Convert the overlayed image to uint8
            overlayed_image_uint8 = cv2.convertScaleAbs(overlayed_image)

            # Convert the overlayed image back to a tensor and permute the dimensions
            overlayed_image_tensor = torch.from_numpy(overlayed_image_uint8.transpose(2, 0, 1))

            # Append the overlayed image tensor to the list
            image_list.append(overlayed_image_tensor)

        # Convert the list of tensors into a single tensor
        image_tensor = torch.stack(image_list)

        # Log the images to TensorBoard
        writer.add_images('Overlayed Images train', image_tensor, epoch)

    # Log the loss
    writer.add_scalar('Loss/train', total_loss.item(), epoch * len(train_loader) + batch_index)

    # Log more metrics
    #writer.add_scalar('Accuracy/train', total_accuracy / len(images), epoch)

    # Log num_classes accuracy
    #writer.add_scalar('Accuracy/num_classes', total_num_classes_accuracy / len(images), epoch)

    # Log IoU
    #writer.add_scalar('IoU/train', total_iou / len(images), epoch)

    # Backward pass and optimization
    total_loss.backward()
    optimizer.step()

# Validate the model after each epoch
validate_loss = validate_model(model, valid_loader, class_loss, bbox_loss, num_classes_loss, epoch, writer)
writer.add_scalar('Loss/valid', validate_loss, epoch)
writer.flush()
print(f'Epoch {epoch+1}/{num_epochs}, Validation Loss: {validate_loss:.4f}')
```



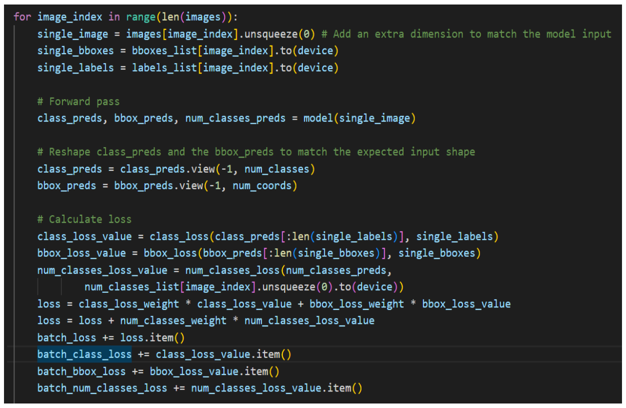
```
    # Check if it's the best validation loss so far
    if validate_loss < best_val_loss:
        best_val_loss = validate_loss
        epochs_without_improvement = 0
        # Save the model
        torch.save(model.state_dict(), 'best_model.pth')
    else:
        epochs_without_improvement += 1

    # If there's no improvement for a certain number of epochs, stop the training
    if epochs_without_improvement == patience:
        print('Early stopping')
        break

    if(epoch % 100 == 0 and epoch != 0):
        # Save the model
        torch.save(model.state_dict(), 'model.pth')

total_end_time = time.time()
total_time = total_end_time - total_start_time
formatted_total_time = str(timedelta(seconds=total_time))
print(f'Total training time: {formatted_total_time}')
```



Valid function: This function validates the model's performance on the validation dataset. It iterates through batches of images from the validation loader, calculates loss, and optionally visualizes predictions on a subset of images every 30 epochs. During each epoch, it computes the total loss, class loss, bounding box loss, and number of classes loss. It returns the average loss over all batches in the validation dataset. The function is designed to be used within the training loop to monitor model performance during training. As in the train model, there is an option to calculate Iou and accuracy for the number of classes predicted.

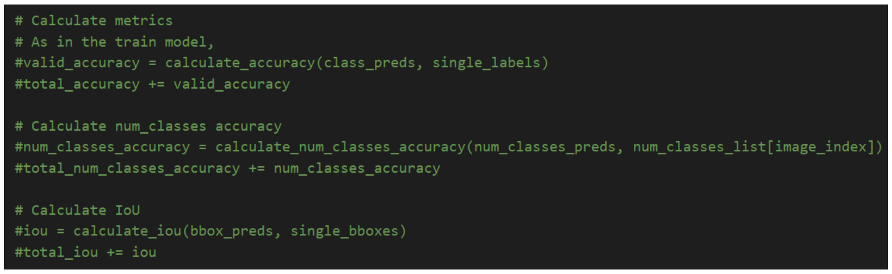


```
def validate_model(model, valid_loader, class_loss, bbox_loss, num_classes_loss, epoch, writer):
    model.eval()  # Set the model to evaluation mode
    total_loss = 0
    total_class_loss = 0
    total_bbox_loss = 0
    total_num_classes_loss = 0
    total_accuracy = 0
    total_iou = 0
    total_num_classes_accuracy = 0

    with torch.no_grad():
        for batch_index, (images, bboxes_list, labels_list, num_classes_list) in enumerate(valid_loader):
            images = images.to(device)
            batch_loss = 0
            batch_class_loss = 0
            batch_bbox_loss = 0
            batch_num_classes_loss = 0

            total_accuracy = 0  # Reset for each batch
            total_iou = 0  # Reset for each batch
            total_num_classes_accuracy = 0  # Reset for each batch
```

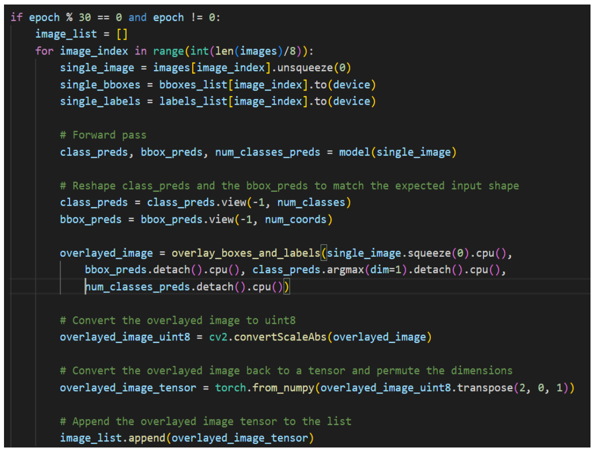


```
for image_index in range(len(images)):
    single_image = images[image_index].unsqueeze(0) # Add an extra dimension to match the model input
    single_bboxes = bboxes_list[image_index].to(device)
    single_labels = labels_list[image_index].to(device)

    # Forward pass
    class_preds, bbox_preds, num_classes_preds = model(single_image)

    # Reshape class_preds and the bbox_preds to match the expected input shape
    class_preds = class_preds.view(-1, num_classes)
    bbox_preds = bbox_preds.view(-1, num_coords)

    # Calculate loss
    class_loss_value = class_loss(class_preds[:len(single_labels)], single_labels)
    bbox_loss_value = bbox_loss(bbox_preds[:len(single_bboxes)], single_bboxes)
    num_classes_loss_value = num_classes_loss(num_classes_preds,
            num_classes_list[image_index].unsqueeze(0).to(device))
    loss = class_loss_weight * class_loss_value + bbox_loss_weight * bbox_loss_value
    loss = loss + num_classes_weight * num_classes_loss_value
    batch_loss += loss.item()
    batch_class_loss += class_loss_value.item()
    batch_bbox_loss += bbox_loss_value.item()
    batch_num_classes_loss += num_classes_loss_value.item()
```

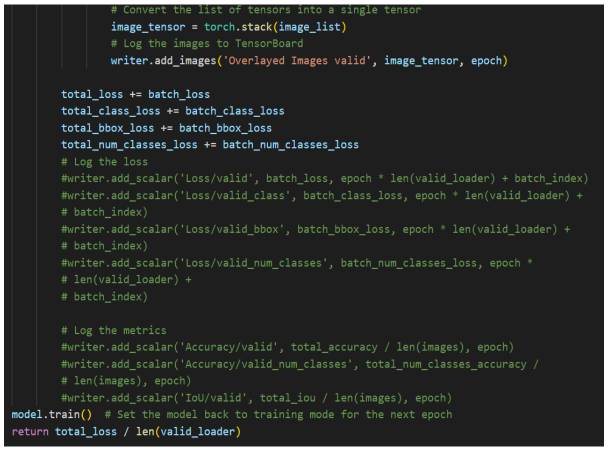
The optional calculation -



```
# Calculate metrics
# As in the train model,
#valid_accuracy = calculate_accuracy(class_preds, single_labels)
#total_accuracy += valid_accuracy

# Calculate num_classes accuracy
#num_classes_accuracy = calculate_num_classes_accuracy(num_classes_preds, num_classes_list[image_index])
#total_num_classes_accuracy += num_classes_accuracy

# Calculate IoU
#iou = calculate_iou(bbox_preds, single_bboxes)
#total_iou += iou
```

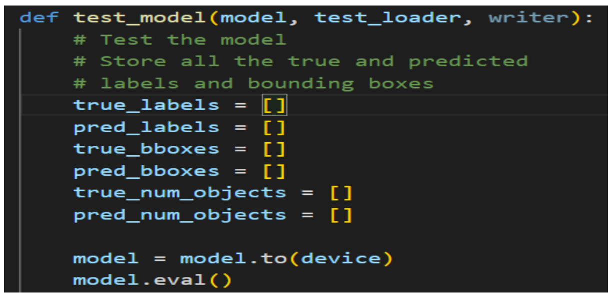


```python
if epoch % 30 == 0 and epoch != 0:
    image_list = []
    for image_index in range(int(len(images)/8)):
        single_image = images[image_index].unsqueeze(0)
        single_bboxes = bboxes_list[image_index].to(device)
        single_labels = labels_list[image_index].to(device)

        # Forward pass
        class_preds, bbox_preds, num_classes_preds = model(single_image)

        # Reshape class_preds and the bbox_preds to match the expected input shape
        class_preds = class_preds.view(-1, num_classes)
        bbox_preds = bbox_preds.view(-1, num_coords)

        overlayed_image = overlay_boxes_and_labels(single_image.squeeze(0).cpu(),
            bbox_preds.detach().cpu(), class_preds.argmax(dim=1).detach().cpu(),
            num_classes_preds.detach().cpu())

        # Convert the overlayed image to uint8
        overlayed_image_uint8 = cv2.convertScaleAbs(overlayed_image)

        # Convert the overlayed image back to a tensor and permute the dimensions
        overlayed_image_tensor = torch.from_numpy(overlayed_image_uint8.transpose(2, 0, 1))

        # Append the overlayed image tensor to the list
        image_list.append(overlayed_image_tensor)
```

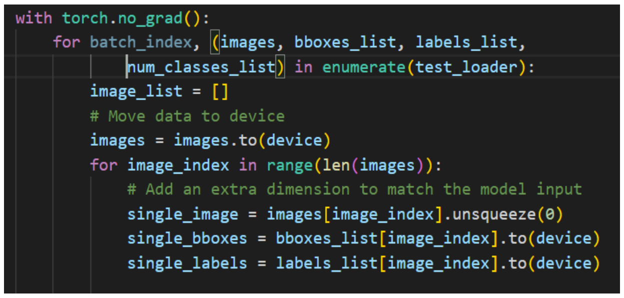


```python
                # Convert the list of tensors into a single tensor
                image_tensor = torch.stack(image_list)
                # Log the images to TensorBoard
                writer.add_images('Overlayed Images valid', image_tensor, epoch)

        total_loss += batch_loss
        total_class_loss += batch_class_loss
        total_bbox_loss += batch_bbox_loss
        total_num_classes_loss += batch_num_classes_loss
        # Log the loss
        #writer.add_scalar('Loss/valid', batch_loss, epoch * len(valid_loader) + batch_index)
        #writer.add_scalar('Loss/valid_class', batch_class_loss, epoch * len(valid_loader) +
        # batch_index)
        #writer.add_scalar('Loss/valid_bbox', batch_bbox_loss, epoch * len(valid_loader) +
        # batch_index)
        #writer.add_scalar('Loss/valid_num_classes', batch_num_classes_loss, epoch *
        # len(valid_loader) +
        # batch_index)

        # Log the metrics
        #writer.add_scalar('Accuracy/valid', total_accuracy / len(images), epoch)
        #writer.add_scalar('Accuracy/valid_num_classes', total_num_classes_accuracy /
        # len(images), epoch)
        #writer.add_scalar('IoU/valid', total_iou / len(images), epoch)
model.train()  # Set the model back to training mode for the next epoch
return total_loss / len(valid_loader)
```

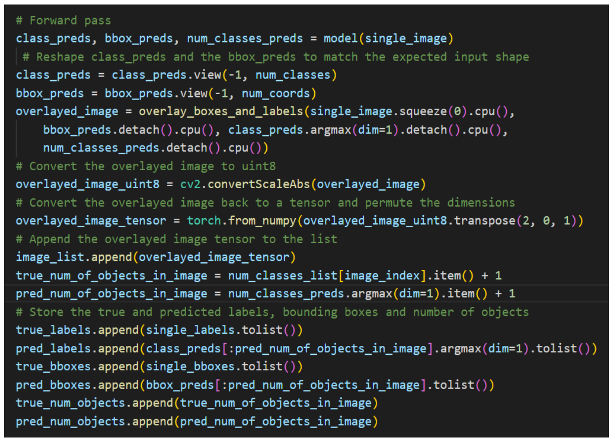
Test function: This function tests the trained model's performance on the test dataset. It iterates through batches of images from the test loader, performs inference using the model, and stores the true and predicted labels, bounding boxes, and number of objects. It also calculates the mean average precision (mAP) for each class and the mean absolute error (MAE) for the number of objects. Finally, it prints the total actual vs predicted number of objects, as well as the calculated mAP and MAE. The function utilizes the overlay_boxes_and_labels function for visualizing predictions and evaluates the model's performance using metrics such as mAP and MAE.



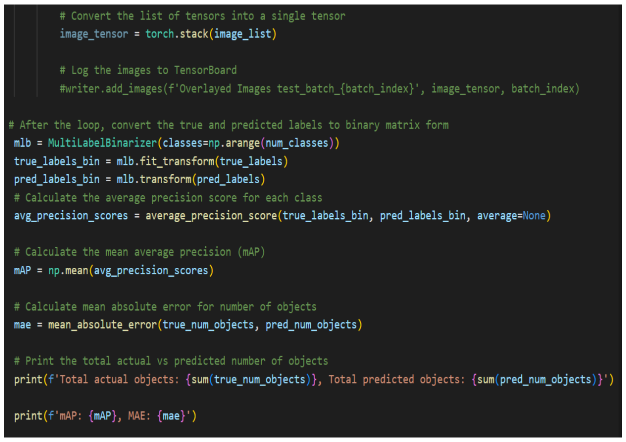
```python
def test_model(model, test_loader, writer):
    # Test the model
    # Store all the true and predicted
    # labels and bounding boxes
    true_labels = []
    pred_labels = []
    true_bboxes = []
    pred_bboxes = []
    true_num_objects = []
    pred_num_objects = []

    model = model.to(device)
    model.eval()
```



```python
with torch.no_grad():
    for batch_index, (images, bboxes_list, labels_list,
            num_classes_list) in enumerate(test_loader):
        image_list = []
        # Move data to device
        images = images.to(device)
        for image_index in range(len(images)):
            # Add an extra dimension to match the model input
            single_image = images[image_index].unsqueeze(0)
            single_bboxes = bboxes_list[image_index].to(device)
            single_labels = labels_list[image_index].to(device)
```

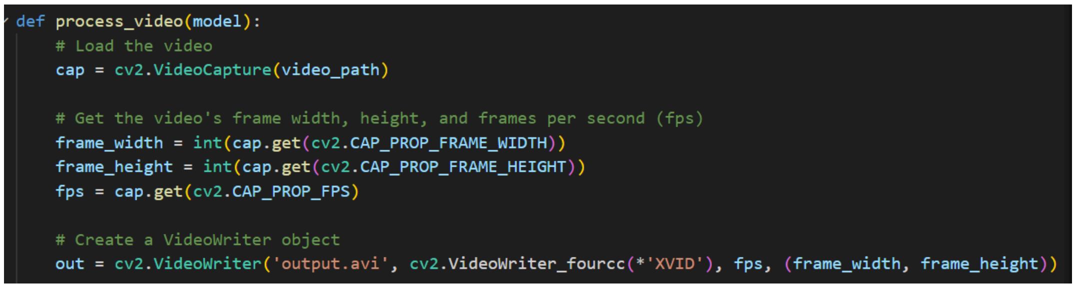


```python
# Forward pass
class_preds, bbox_preds, num_classes_preds = model(single_image)
 # Reshape class_preds and the bbox_preds to match the expected input shape
class_preds = class_preds.view(-1, num_classes)
bbox_preds = bbox_preds.view(-1, num_coords)
overlayed_image = overlay_boxes_and_labels(single_image.squeeze(0).cpu(),
    bbox_preds.detach().cpu(), class_preds.argmax(dim=1).detach().cpu(),
    num_classes_preds.detach().cpu())
# Convert the overlayed image to uint8
overlayed_image_uint8 = cv2.convertScaleAbs(overlayed_image)
# Convert the overlayed image back to a tensor and permute the dimensions
overlayed_image_tensor = torch.from_numpy(overlayed_image_uint8.transpose(2, 0, 1))
# Append the overlayed image tensor to the list
image_list.append(overlayed_image_tensor)
true_num_of_objects_in_image = num_classes_list[image_index].item() + 1
pred_num_of_objects_in_image = num_classes_preds.argmax(dim=1).item() + 1
# Store the true and predicted labels, bounding boxes and number of objects
true_labels.append(single_labels.tolist())
pred_labels.append(class_preds[:pred_num_of_objects_in_image].argmax(dim=1).tolist())
true_bboxes.append(single_bboxes.tolist())
pred_bboxes.append(bbox_preds[:pred_num_of_objects_in_image].tolist())
true_num_objects.append(true_num_of_objects_in_image)
pred_num_objects.append(pred_num_of_objects_in_image)
```

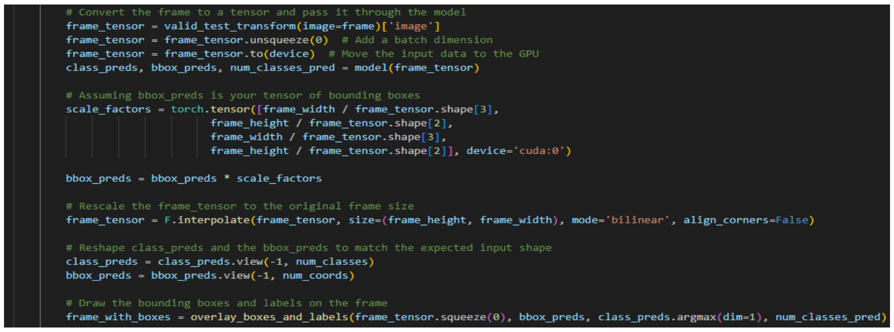


```python
        # Convert the list of tensors into a single tensor
        image_tensor = torch.stack(image_list)

        # Log the images to TensorBoard
        #writer.add_images(f'Overlayed Images test_batch_{batch_index}', image_tensor, batch_index)

# After the loop, convert the true and predicted labels to binary matrix form
mlb = MultiLabelBinarizer(classes=np.arange(num_classes))
true_labels_bin = mlb.fit_transform(true_labels)
pred_labels_bin = mlb.transform(pred_labels)
# Calculate the average precision score for each class
avg_precision_scores = average_precision_score(true_labels_bin, pred_labels_bin, average=None)

# Calculate the mean average precision (mAP)
mAP = np.mean(avg_precision_scores)

# Calculate mean absolute error for number of objects
mae = mean_absolute_error(true_num_objects, pred_num_objects)

# Print the total actual vs predicted number of objects
print(f'Total actual objects: {sum(true_num_objects)}, Total predicted objects: {sum(pred_num_objects)}')

print(f'mAP: {mAP}, MAE: {mae}')
```

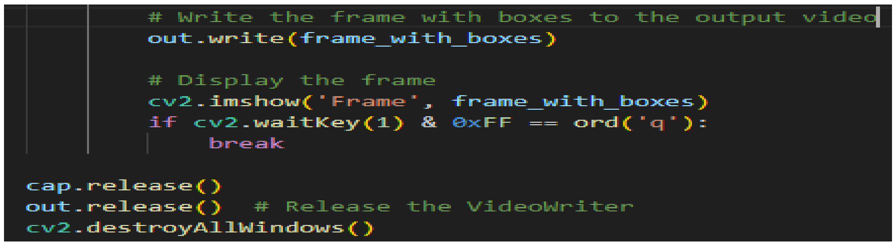
Video Inference function: The process_video function takes a pre-trained object detection model and a video as input. It iterates through the frames of the video, performing object detection on each frame using the model. Bounding boxes and labels are overlaid on the frames based on the model's predictions. The annotated frames are then written to an output video file. Finally, the function cleans up by releasing resources



```python
def process_video(model):
    # Load the video
    cap = cv2.VideoCapture(video_path)

    # Get the video's frame width, height, and frames per second (fps)
    frame_width = int(cap.get(cv2.CAP_PROP_FRAME_WIDTH))
    frame_height = int(cap.get(cv2.CAP_PROP_FRAME_HEIGHT))
    fps = cap.get(cv2.CAP_PROP_FPS)

    # Create a VideoWriter object
    out = cv2.VideoWriter('output.avi', cv2.VideoWriter_fourcc(*'XVID'), fps, (frame_width, frame_height))
```

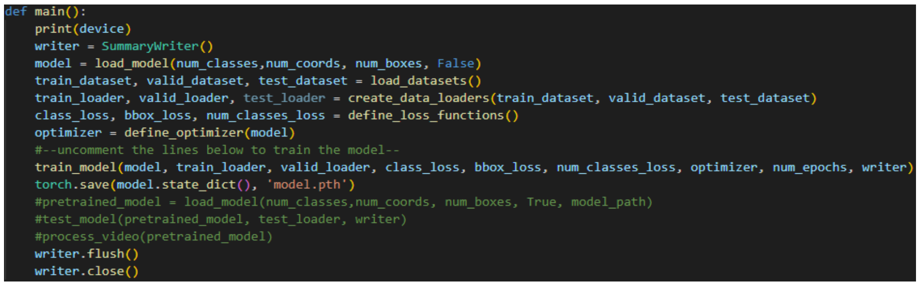


```python
            # Convert the frame to a tensor and pass it through the model
            frame_tensor = valid_test_transform(image=frame)['image']
            frame_tensor = frame_tensor.unsqueeze(0)  # Add a batch dimension
            frame_tensor = frame_tensor.to(device)  # Move the input data to the GPU
            class_preds, bbox_preds, num_classes_pred = model(frame_tensor)

            # Assuming bbox_preds is your tensor of bounding boxes
            scale_factors = torch.tensor([frame_width / frame_tensor.shape[3],
                                          frame_height / frame_tensor.shape[2],
                                          frame_width / frame_tensor.shape[3],
                                          frame_height / frame_tensor.shape[2]], device='cuda:0')

            bbox_preds = bbox_preds * scale_factors

            # Rescale the frame_tensor to the original frame size
            frame_tensor = F.interpolate(frame_tensor, size=(frame_height, frame_width), mode='bilinear', align_corners=False)

            # Reshape class_preds and the bbox_preds to match the expected input shape
            class_preds = class_preds.view(-1, num_classes)
            bbox_preds = bbox_preds.view(-1, num_coords)

            # Draw the bounding boxes and labels on the frame
            frame_with_boxes = overlay_boxes_and_labels(frame_tensor.squeeze(0), bbox_preds, class_preds.argmax(dim=1), num_classes_pred)
```



```python
            # Write the frame with boxes to the output video
            out.write(frame_with_boxes)

            # Display the frame
            cv2.imshow('Frame', frame_with_boxes)
            if cv2.waitKey(1) & 0xFF == ord('q'):
                break

    cap.release()
    out.release()  # Release the VideoWriter
    cv2.destroyAllWindows()
```

Main: In the main function, we initialize the writer with which we log to TensorBoard throughout the program. We load the model, whether it's a pretrained one or a model we want to train. Additionally, here we create the train, validation, and test datasets and create their DataLoader objects. Moreover, we define the variables for loss functions and the optimizer we are going to use during training. There is an option to load a pretrained model and perform inference on a video with it.



```python
def main():
    print(device)
    writer = SummaryWriter()
    model = load_model(num_classes,num_coords, num_boxes, False)
    train_dataset, valid_dataset, test_dataset = load_datasets()
    train_loader, valid_loader, test_loader = create_data_loaders(train_dataset, valid_dataset, test_dataset)
    class_loss, bbox_loss, num_classes_loss = define_loss_functions()
    optimizer = define_optimizer(model)
    #--uncomment the lines below to train the model--
    train_model(model, train_loader, valid_loader, class_loss, bbox_loss, num_classes_loss, optimizer, num_epochs, writer)
    torch.save(model.state_dict(), 'model.pth')
    #pretrained_model = load_model(num_classes,num_coords, num_boxes, True, model_path)
    #test_model(pretrained_model, test_loader, writer)
    #process_video(pretrained_model)
    writer.flush()
    writer.close()
```

Struggles and Limitations:

- The model succeeded in detecting objects in the train set but struggles to generalize on unseen data as shown in the video that we performed inference on, even though we applied a relatively high dropout rate of 0.7 used L2 regularization and also a number of appropriate augmentations from the Albumentations library.
- Our model assumed that there will be no more then 10 objects in an image and in our entire dataset the max amount of objects per image was 9.