Deep Learning: Homework 6 Final Submission

Arian Mollajafari Sohi

amollaja@purdue.edu

3.2 Final Submission of HW6

Object detection stands as an important task with wide-ranging applications, from autonomous vehicles to surveillance systems. Among the various algorithms developed for this purpose, YOLO (You Only Look Once) has emerged as a prominent method, known for its exceptional balance between speed and accuracy. In this homework we want to apply the YOLO logic on the COCO (Common Objects in Context) dataset. The COCO dataset, provides an ideal platform for training and evaluating object detection models.

We have three objectives in this homework. First, we focus on creating a multi-instance object localization dataset derived from the COCO dataset. This involves preprocessing the original dataset to suit the specific requirements of this homework and YOLO algorithm. Second, we embark on the construction of a deep neural network tailored for object detection, leveraging the YOLO architecture. Lastly, the homework includes the training and evaluation of the network, where we fine-tune the model parameters and assess its performance on our test set. The following sections will detail each step of our approach.

3.2.2 Creating Your Own Multi-Instance Object Localization Dataset

In this part, we aim to construct a dataset tailored for multi-instance object localization, leveraging the COCO 2017 dataset. The dataset creation process involves filtering images and annotations to meet specific criteria, ensuring that the resulting dataset is conducive to training and evaluating.

Dataset Criteria

The following criteria were applied to select images for the dataset:

Foreground Object Selection: Each image in the dataset must contain at least one foreground object belonging to one of three categories: 'cake', 'dog', or 'motorcycle'.

Object Size Constraint: The area of any foreground object must exceed 4096 pixels (i.e., larger than 64×64 pixels) to ensure that the objects are sufficiently large for effective detection.

Multi-Instance Capability: Unlike the previous homework, images can contain multiple foreground objects to accommodate the multi-instance object localization task.

Image Resizing: All selected images are resized to 256×256 pixels. Correspondingly, the bounding box parameters of the objects are scaled to match the resized images.

Dataset Split: The dataset is divided into a training set and a testing set, using images from the COCO 2017 Train and Val sets, respectively.

The following Python script shows the process of creating a dataset for multi-instance object localization using the COCO 2017 dataset. The script filters images based on specific criteria and prepares them for training and testing deep neural networks.

Code Description:

Class Selection:

Define the classes of interest: 'cake', 'dog', and 'motorcycle'.

Image Processing Function (save_resized_image):

Resizes the image to 256x256 pixels.

Draws bounding boxes around the objects of interest.

Saves the processed image to the specified file path.

Image Filtering and Processing Function (process_images):

Iterates through each class of interest.

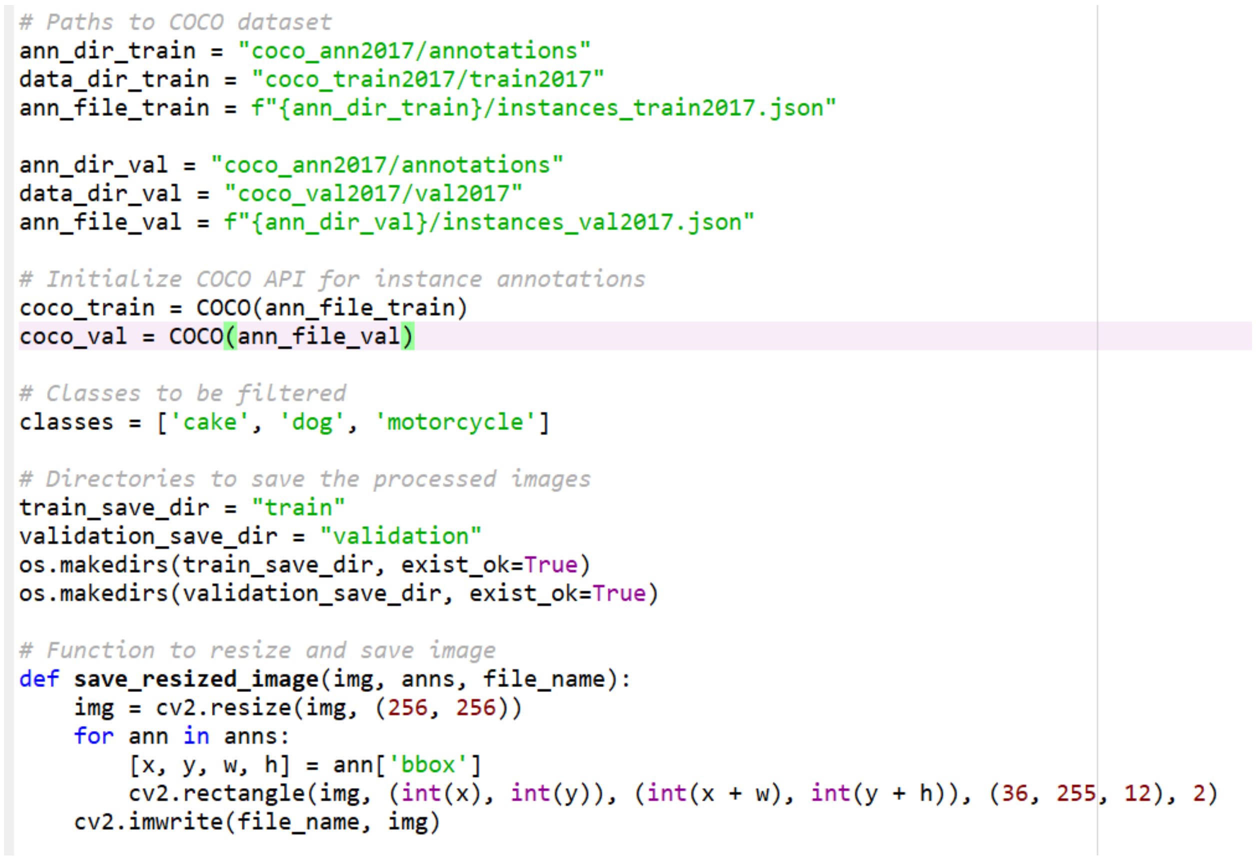
Retrieves the category ID and all image IDs for each class.

Filters images to include only those with objects larger than 4096 pixels.

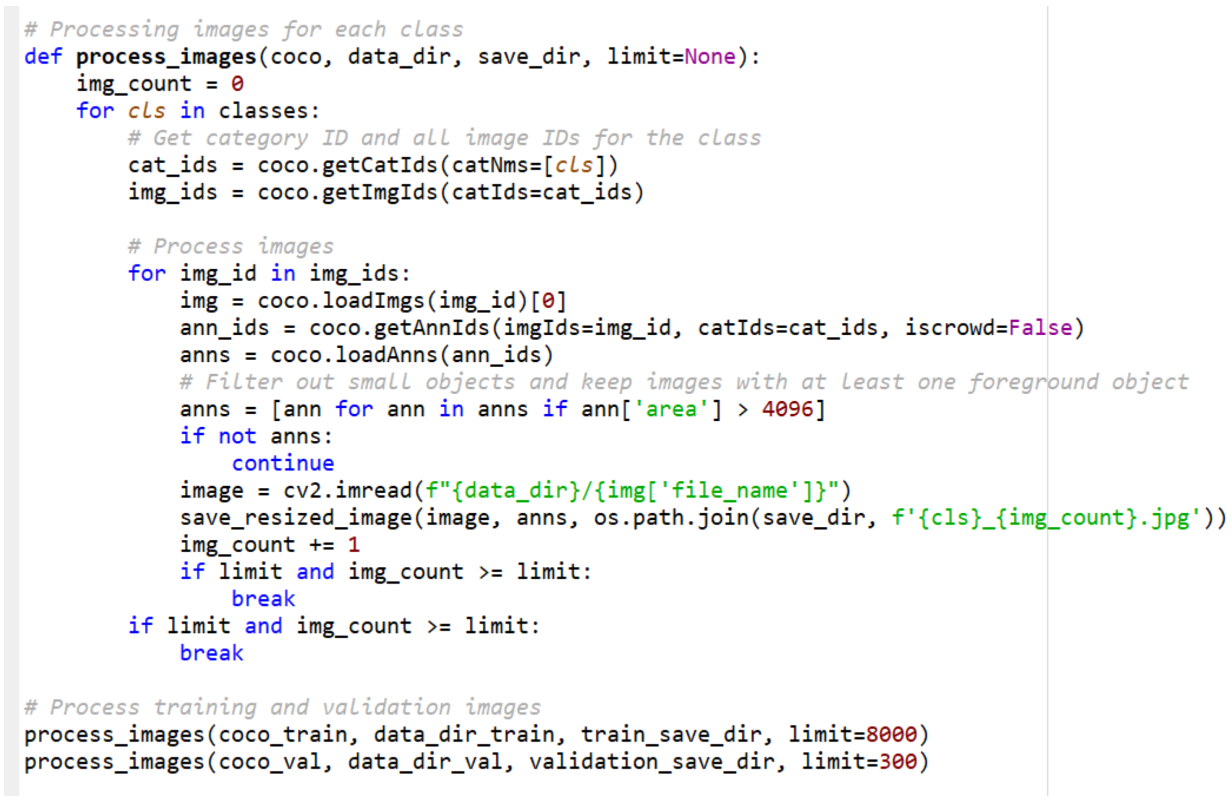
Processes and saves a limited number of images to the specified directory.

Processing Training and Validation Images:

Calls the process_images function for both training and validation datasets, with limits of 8000 and 300 images, respectively.



```python
# Paths to COCO dataset
ann_dir_train = "coco_ann2017/annotations"
data_dir_train = "coco_train2017/train2017"
ann_file_train = f"{ann_dir_train}/instances_train2017.json"

ann_dir_val = "coco_ann2017/annotations"
data_dir_val = "coco_val2017/val2017"
ann_file_val = f"{ann_dir_val}/instances_val2017.json"

# Initialize COCO API for instance annotations
coco_train = COCO(ann_file_train)
coco_val = COCO(ann_file_val)

# Classes to be filtered
classes = ['cake', 'dog', 'motorcycle']

# Directories to save the processed images
train_save_dir = "train"
validation_save_dir = "validation"
os.makedirs(train_save_dir, exist_ok=True)
os.makedirs(validation_save_dir, exist_ok=True)

# Function to resize and save image
def save_resized_image(img, anns, file_name):
    img = cv2.resize(img, (256, 256))
    for ann in anns:
        [x, y, w, h] = ann['bbox']
        cv2.rectangle(img, (int(x), int(y)), (int(x + w), int(y + h)), (36, 255, 12), 2)
    cv2.imwrite(file_name, img)
```



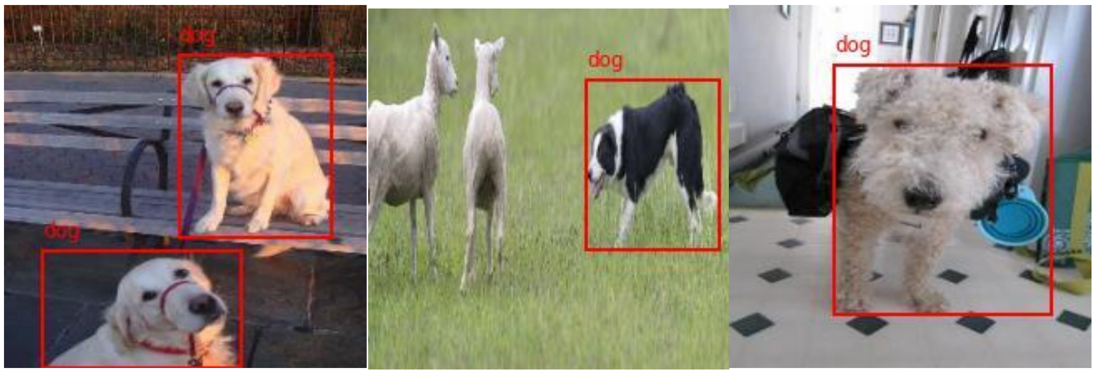
```python
# Processing images for each class
def process_images(coco, data_dir, save_dir, limit=None):
    img_count = 0
    for cls in classes:
        # Get category ID and all image IDs for the class
        cat_ids = coco.getCatIds(catNms=[cls])
        img_ids = coco.getImgIds(catIds=cat_ids)

        # Process images
        for img_id in img_ids:
            img = coco.loadImgs(img_id)[0]
            ann_ids = coco.getAnnIds(imgIds=img_id, catIds=cat_ids, iscrowd=False)
            anns = coco.loadAnns(ann_ids)
            # Filter out small objects and keep images with at least one foreground object
            anns = [ann for ann in anns if ann['area'] > 4096]
            if not anns:
                continue
            image = cv2.imread(f"{data_dir}/{img['file_name']}")
            save_resized_image(image, anns, os.path.join(save_dir, f'{cls}_{img_count}.jpg'))
            img_count += 1
            if limit and img_count >= limit:
                break
        if limit and img_count >= limit:
            break

# Process training and validation images
process_images(coco_train, data_dir_train, train_save_dir, limit=8000)
process_images(coco_val, data_dir_val, validation_save_dir, limit=300)
```

Sample images:

Cake:

Motorcycle:

Dog:



3.2.3 Building Your Deep Neural Network

In this section, we describe the implementation of a deep convolutional neural network (CNN) designed for multi-instance object classification and localization, inspired by the YOLO architecture and tailored for the dataset created in the previous step.

Network Architecture:

The CNN, named YOLONet, is based on the YOLO architecture with adjustments made to accommodate the specific requirements of our task. The network consists of several convolutional layers followed by max-pooling layers, a skip connection, and fully connected layers. The final layer of the network is designed to output a yolo_tensor, which contains the predicted parameters for object localization and classification.

Design Choices:

Grid Size (S), Bounding Boxes (B), and Classes (C): The network is parameterized to accommodate a grid size of 7x7 (S=7), 2 bounding boxes per grid cell (B=2), and 3 object classes (C=3) corresponding to 'cake', 'dog', and 'motorcycle'.

Skip Connection: A skip connection is implemented between the first and third convolutional layers to enhance feature extraction and improve gradient flow during training.

Output YOLO Tensor: The final fully connected layer is designed to output a tensor with the shape [Batch Size, S, S, (B * 5 + C)], where 5 represents the parameters for bounding box prediction (x, y, width, height, objectness score) and C represents the number of classes.

Code Description for YOLONet:

The YOLONet class is a PyTorch module that implements a deep convolutional neural network for multi-instance object localization, inspired by the YOLO architecture. Here's a breakdown of the code:

Class Initialization (__init__):

The YOLONet class is initialized with parameters S, B, and C, representing the grid size, number of bounding boxes per grid cell, and number of object classes, respectively.

The convolutional layers (conv1, conv2, conv3) are defined with kernel size 3 and padding 1 to maintain the spatial dimensions of the input.

A skip connection layer (skip1) is defined to connect the output of the first convolutional layer to the input of the third convolutional layer.

A max-pooling layer (pool) is defined with a kernel size of 2 and stride 2 to reduce the spatial dimensions by half after each convolutional layer.

Fully connected layers (fc1, fc2) are defined to process the flattened output of the convolutional layers and output the final yolo_tensor. The second fully connected layer (fc2) is specifically designed to match the shape of the yolo_tensor.

Forward Pass (forward):

The input image x is passed through the first convolutional layer (conv1), followed by a ReLU activation function and max-pooling. The result is stored in x1.

The output x1 is then passed through the second convolutional layer (conv2), followed by ReLU and max-pooling, resulting in x2.

Similarly, x2 is passed through the third convolutional layer (conv3), followed by ReLU and max-pooling, resulting in x3.

The skip connection is implemented by applying the skip1 layer to x1, pooling twice to match the dimensions, and then adding the result element-wise to x3.

The output x3 is then flattened and passed through the first fully connected layer (fc1), followed by a ReLU activation function.

Finally, the output is passed through the second fully connected layer (fc2) to obtain the yolo_tensor. The yolo_tensor is then reshaped to the desired output shape [Batch Size, S, S, (B * 5 + C)], where the last dimension includes the parameters for bounding box prediction and class probabilities for each grid cell and bounding box.

Instantiation of YOLONet:

The network is instantiated with the specified grid size (S), number of bounding boxes per cell (B), and number of object classes (C). In this case, the values are set to 7, 2, and 3, respectively, to accommodate the dataset and task requirements.

```
class YOLONet(nn.Module):
    def __init__(self, S=7, B=2, C=3):
        super(YOLONet, self).__init__()
        self.S = S
        self.B = B
        self.C = C
        # Define the layers of the CNN
        self.conv1 = nn.Conv2d(3, 16, 3, padding=1)
        self.conv2 = nn.Conv2d(16, 32, 3, padding=1)
        self.conv3 = nn.Conv2d(32, 64, 3, padding=1)
        self.skip1 = nn.Conv2d(16, 64, 1)  # Skip connection layer
        self.pool = nn.MaxPool2d(2, 2)
        # Fully connected layers
        self.fc1 = nn.Linear(64 * S * S, 512)
        self.fc2 = nn.Linear(512, S * S * (B * 5 + C))  # Adjusted for the YOLO tensor shape

    def forward(self, x):
        x1 = self.pool(F.relu(self.conv1(x)))
        x2 = self.pool(F.relu(self.conv2(x1)))
        x3 = self.pool(F.relu(self.conv3(x2)))

        # Implementing the skip connection
        x1_skip = self.skip1(x1)
        x1_skip_pooled = self.pool(self.pool(x1_skip))  # Pool twice to match dimensions
        x3 += x1_skip_pooled  # Element-wise addition for skip connection

        x3 = x3.view(-1, 64 * self.S * self.S)
        x3 = F.relu(self.fc1(x3))
        yolo_tensor = self.fc2(x3)
        # Reshape to the YOLO tensor
        yolo_tensor = yolo_tensor.view(-1, self.S, self.S, (self.B * 5 + self.C))
        return yolo_tensor

# Parameters for the YOLO grid and classes
S = 7  # Size of grid (7x7 for YOLOv1, for example)
B = 2  # Number of bounding boxes
C = 3  # Number of classes
```

3.3 Training and Evaluating Your Network

1. **Dataloader:**

The dataloader is responsible for loading the images and their corresponding annotations in a format suitable for training the YOLO model.

Writing a Custom Dataloader:

Dataloader Structure:

The dataloader should inherit from PyTorch's Dataset class and implement the __getitem__ method, which returns the data for a single image.

The __getitem__ method should return:

The image tensor: A PyTorch tensor representing the image.

For each foreground object in the image:

- The index of the assigned cell in the grid.
- The index of the assigned anchor box within the cell.
- The ground truth yolo_vector for the object.

Assigning Cells and Anchor Boxes:

The assignment of a cell to a ground truth (GT) bounding box is typically based on the center of the bounding box. The cell containing the center of the GT bounding box is chosen as the assigned cell.

The assignment of an anchor box is based on the highest Intersection over Union (IoU) between the GT bounding box and the predefined anchor boxes for the cell. The anchor box with the highest IoU is chosen as the assigned anchor box.

YOLO Vector:

The yolo_vector for each object should contain the following parameters:

- δx and δy: The offsets between the center of the GT bounding box and the center of the assigned anchor box.
- σw and σh: The ratios between the width and height of the GT bounding box and the width and height of the assigned anchor box.

MyDataset:

The MyDataset class is a class that loads images and annotations from the COCO dataset for use in training and evaluating a YOLO model. Here's a breakdown of the updated code:

Initialization (__init__):

The constructor now also takes in parameters S and B, representing the grid size and the number of bounding boxes per grid cell, respectively.

The rest of the initialization process remains the same as before, loading the COCO dataset, filtering image IDs, and setting up the class-to-index mapping and transformations.

Length Method (__len__):

Returns the number of images in the dataset, as before.

Get Item Method (__getitem__):

Loads an image and its annotations based on the provided index.

Initializes a yolo_tensor with the shape (S, S, B, 5 + len(class_names)), where 5 accounts for the bounding box parameters and objectness score, and len(class_names) accounts for the class probabilities.

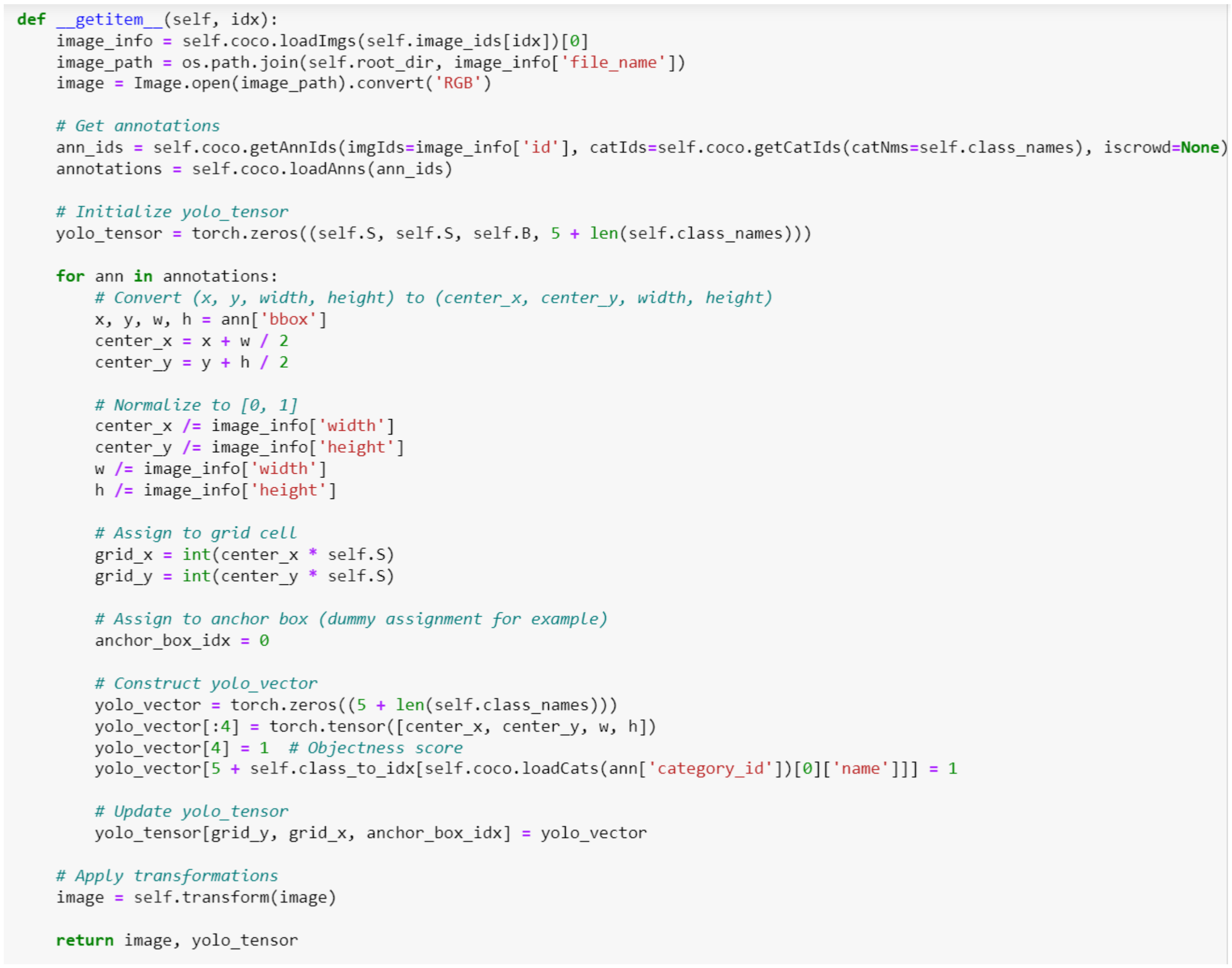
For each annotation in the image:

- Converts the bounding box format from COCO to YOLO (center coordinates and normalized dimensions).
- Assigns the bounding box to a grid cell based on its center coordinates.
- Assigns the bounding box to an anchor box (in this simplified example, the assignment is a dummy value).
- Constructs a yolo_vector containing the bounding box parameters, objectness score, and one-hot encoded class label.
- Updates the corresponding entry in the yolo_tensor with the yolo_vector.

Applies the specified transformations to the image.

Returns the transformed image and the yolo_tensor.

```python
class MyDataset(Dataset):
    def __init__(self, annotation_file, root_dir, class_names, S=7, B=2, transform=None):
        self.coco = COCO(annotation_file)
        self.root_dir = root_dir
        self.image_ids = self.coco.getImgIds()
        self.class_names = class_names
        self.class_to_idx = {class_name: idx for idx, class_name in enumerate(class_names)}
        self.S = S
        self.B = B
        self.transform = transform if transform else transforms.ToTensor()

        # Filter for images that have annotations
        self.image_ids = [
            img_id for img_id in self.image_ids
            if len(self.coco.getAnnIds(imgIds=img_id, catIds=self.coco.getCatIds(catNms=class_names))) > 0
        ]

    def __len__(self):
        return len(self.image_ids)
```



```python
def __getitem__(self, idx):
    image_info = self.coco.loadImgs(self.image_ids[idx])[0]
    image_path = os.path.join(self.root_dir, image_info['file_name'])
    image = Image.open(image_path).convert('RGB')

    # Get annotations
    ann_ids = self.coco.getAnnIds(imgIds=image_info['id'], catIds=self.coco.getCatIds(catNms=self.class_names), iscrowd=None)
    annotations = self.coco.loadAnns(ann_ids)

    # Initialize yolo_tensor
    yolo_tensor = torch.zeros((self.S, self.S, self.B, 5 + len(self.class_names)))

    for ann in annotations:
        # Convert (x, y, width, height) to (center_x, center_y, width, height)
        x, y, w, h = ann['bbox']
        center_x = x + w / 2
        center_y = y + h / 2

        # Normalize to [0, 1]
        center_x /= image_info['width']
        center_y /= image_info['height']
        w /= image_info['width']
        h /= image_info['height']

        # Assign to grid cell
        grid_x = int(center_x * self.S)
        grid_y = int(center_y * self.S)

        # Assign to anchor box (dummy assignment for example)
        anchor_box_idx = 0

        # Construct yolo_vector
        yolo_vector = torch.zeros((5 + len(self.class_names)))
        yolo_vector[:4] = torch.tensor([center_x, center_y, w, h])
        yolo_vector[4] = 1  # Objectness score
        yolo_vector[5 + self.class_to_idx[self.coco.loadCats(ann['category_id'])[0]['name']]] = 1

        # Update yolo_tensor
        yolo_tensor[grid_y, grid_x, anchor_box_idx] = yolo_vector

    # Apply transformations
    image = self.transform(image)

    return image, yolo_tensor
```

2. **Training:**

Task Description:

The part involves defining a custom loss function for training a YOLO model. The YOLO model is designed for object detection tasks and requires a specialized loss function that takes into account various aspects of the model's predictions. The loss function consists of several components:

Confidence Loss: Penalizes the model for inaccuracies in predicting the objectness score, which indicates the presence of an object within a bounding box. This loss is calculated separately for boxes that contain objects (object present) and for boxes that do not contain objects (no object present).

Class Loss: Penalizes the model for inaccuracies in predicting the class of the object contained within a bounding box.

Intersection over Union (IoU): A measure used to evaluate the overlap between the predicted bounding box and the ground truth bounding box. It is used as part of the confidence loss calculation.

Coordinate Loss: Penalizes the model for inaccuracies in predicting the center coordinates of the bounding boxes.

Size Loss: Penalizes the model for inaccuracies in predicting the width and height of the bounding boxes.

The total loss is a weighted sum of these individual loss components.

Code Explanation:

Class Initialization (__init__):

Initializes the YOLOLoss class with parameters for the grid size (S), number of bounding boxes per grid cell (B), number of classes (C), and weights for the coordinate loss (lambda_coord) and no object loss (lambda_noobj).

Forward Method (forward):

> Takes in the model predictions and the target (ground truth) tensors. The predictions and targets are expected to have dimensions (batch_size, S, S, B*5+C).
>
> Splits the predictions and targets into their components: bounding box parameters and class predictions.
>
> Calculates the IoU between the predicted and target bounding boxes.
>
> Determines the best bounding box for each grid cell based on the IoU.
>
> Calculates the coordinate loss, size loss, object loss, no object loss, and class loss as described above.
>
> Computes the total loss as the sum of the individual loss components.

compute_iou Method:

A static method that calculates the Intersection over Union (IoU) between two sets of bounding boxes. This is used to evaluate the accuracy of the predicted bounding boxes compared to the ground truth.

```
# Define a YOLO Loss Function
class YOLOLoss(nn.Module):
    def __init__(self, S=7, B=2, C=3, lambda_coord=5, lambda_noobj=0.5):
        super(YOLOLoss, self).__init__()
        self.S = S  # grid size
        self.B = B  # number of bounding boxes
        self.C = C  # number of classes
        self.lambda_coord = lambda_coord  # weight for bbox coordinates
        self.lambda_noobj = lambda_noobj  # weight for no object loss

    def forward(self, predictions, targets):
        # targets are expected to be in the same format as predictions
        # i.e., a batch of tensors of dimensions (S, S, B*5+C)

        # Split the predictions and the targets into their components
        pred_boxes = predictions[..., :self.B*5].view(-1, self.S, self.S, self.B, 5)
        pred_classes = predictions[..., self.B*5:]

        target_boxes = targets[..., :self.B*5].view(-1, self.S, self.S, self.B, 5)
        target_classes = targets[..., self.B*5:]

        # Calculate the IoU for the predicted and target boxes
        iou_b1 = self.compute_iou(pred_boxes[..., 0, :4], target_boxes[..., 0, :4])
        iou_b2 = self.compute_iou(pred_boxes[..., 1, :4], target_boxes[..., 1, :4])
        ious = torch.cat([iou_b1.unsqueeze(0), iou_b2.unsqueeze(0)], dim=0)
        iou_maxes, bestbox = torch.max(ious, dim=0)

        # Coordinate loss
        coord_mask = target_boxes[..., :, 4] > 0  # object presence mask
        noobj_mask = target_boxes[..., :, 4] == 0  # no object presence mask
        coord_loss = self.lambda_coord * torch.sum(coord_mask * ((pred_boxes[..., :, 0] - target_boxes[..., :, 0])**2 +
                                                                 (pred_boxes[..., :, 1] - target_boxes[..., :, 1])**2))

        # Size loss (width and height)
        size_loss = self.lambda_coord * torch.sum(coord_mask * ((pred_boxes[..., :, 2].sqrt() - target_boxes[..., :, 2].sqrt
                                                                (pred_boxes[..., :, 3].sqrt() - target_boxes[..., :, 3].sqrt(

        # Confidence loss (object present)
        obj_loss = torch.sum(coord_mask * (pred_boxes[..., :, 4] - iou_maxes)**2)
        # Confidence loss (no object present)
        noobj_loss = self.lambda_noobj * torch.sum(noobj_mask * (pred_boxes[..., :, 4])**2)

        # Class loss
        class_loss = F.cross_entropy(pred_classes, target_classes.long(), reduction='sum')

        # Total loss
        loss = coord_loss + size_loss + obj_loss + noobj_loss + class_loss
        return loss
```

```
def compute_iou(box1, box2):
    """
    Calculate the Intersection over Union (IoU) of two bounding boxes.
    """
    # Get the coordinates of the bounding boxes
    b1_x1, b1_y1, b1_x2, b1_y2 = box1[:, 0], box1[:, 1], box1[:, 2], box1[:, 3]
    b2_x1, b2_y1, b2_x2, b2_y2 = box2[:, 0], box2[:, 1], box2[:, 2], box2[:, 3]

    # Calculate the area of the intersection rectangle
    inter_area = (torch.min(b1_x2, b2_x2) - torch.max(b1_x1, b2_x1)).clamp(0) * \
                 (torch.min(b1_y2, b2_y2) - torch.max(b1_y1, b2_y1)).clamp(0)

    # Calculate the area of both bounding boxes
    b1_area = (b1_x2 - b1_x1) * (b1_y2 - b1_y1)
    b2_area = (b2_x2 - b2_x1) * (b2_y2 - b2_y1)

    # Calculate the IoU
    iou = inter_area / (b1_area + b2_area - inter_area)

    return iou
```

The next part is responsible for training a YOLO model for object detection. It begins by instantiating the custom YOLO loss function, YOLOLoss, with the specified grid size (S), number of bounding boxes per grid cell (B), and number of classes (C). This loss function is then moved to the appropriate computational device (GPU or CPU) using the .to(device) method.

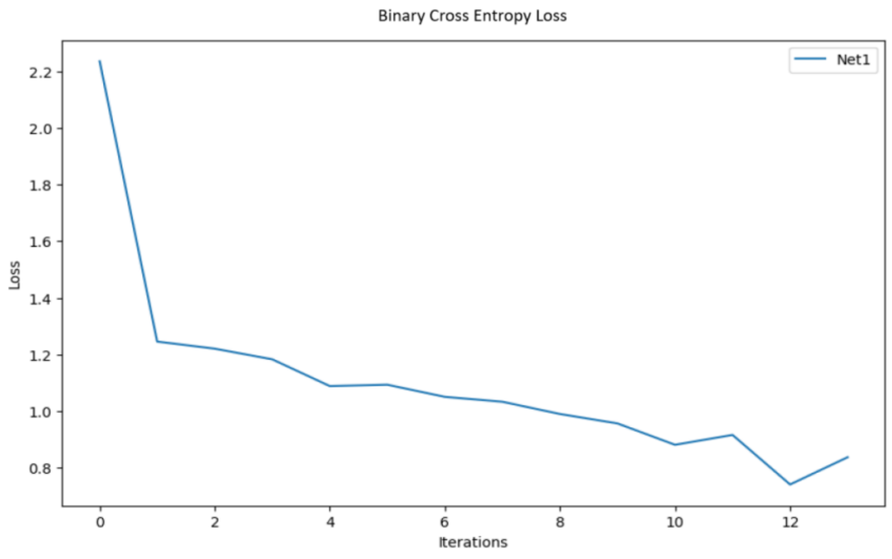
An Adam optimizer is created to update the parameters of the YOLO model (yolo_net) during training, with a learning rate of 1e-3. The training process is carried out over a specified number of epochs, within which the model iterates over the training data in mini-batches.

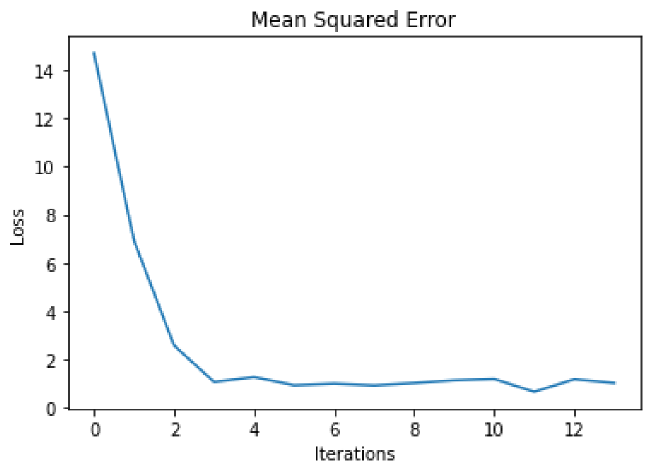
For each mini-batch, the input images and target annotations (ground truth bounding boxes and class labels) are transferred to the computational device. The optimizer's gradients are reset to zero to prevent accumulation across mini-batches. A forward pass through the YOLO model generates predictions, which are then used to compute the loss using the previously defined YOLO loss function. This loss takes into account various aspects of the predictions, such as the accuracy of bounding box coordinates, objectness scores, and class predictions.

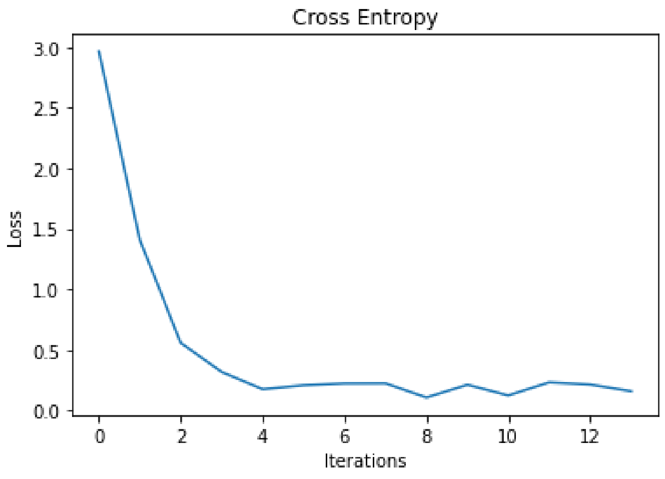
Backpropagation is performed to calculate the gradients of the loss with respect to the model parameters, and the optimizer updates the parameters based on these gradients. The loss for each mini-batch is accumulated to calculate the average loss over a certain number of mini-batches (e.g., every 100 mini-batches), which is then printed to monitor the training progress.

After training for the specified number of epochs, a message indicating the completion of training is printed. At this point, the YOLO model has been trained to detect objects in images, with its performance depending on the effectiveness of the loss function, optimizer, and the quality of the training data.

```
# Instantiate the loss function
yolo_loss = YOLOLoss(S, B, C).to(device)

# Optimizer
optimizer = optim.Adam(yolo_net.parameters(), lr=1e-3)

# Training loop
epochs = 10
for epoch in range(epochs):
    running_loss = 0.0
    for i, (inputs, targets) in enumerate(train_loader):
        inputs = inputs.to(device)
        targets = {key: val.to(device) for key, val in targets.items()}

        optimizer.zero_grad()

        # Forward pass
        predictions = yolo_net(inputs)

        # Calculate the loss
        loss = yolo_loss(predictions, targets)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()

        if (i + 1) % 100 == 0:  # Print every 100 mini-batches
            print(f"Epoch {epoch + 1}/{epochs}, Step {i + 1}/{len(train_loader)}, Loss: {runni
            running_loss = 0.0

print("Finished Training")
```







3. Evaluation:

In this task, we develop our evaluation method to assess the performance of a multi-instance object detector trained using the YOLO framework. The evaluation focused on presenting qualitative results by visually displaying the predicted bounding boxes and class labels for each object detected in test images, alongside the ground truth annotations.

The evaluation process involved several key steps:

Processing Predicted YOLO Tensor: The predicted yolo_tensor from the YOLO model contained information about bounding boxes and class probabilities for each grid cell in the image. We extracted the bounding box coordinates and the corresponding class probabilities from this tensor. For each grid cell, we selected the bounding box with the highest objectness score and chose the class with the highest probability as the predicted class for that bounding box.

Converting Coordinates: The bounding box coordinates in the yolo_tensor were relative to the grid cell. We converted these coordinates to absolute coordinates relative to the entire image by scaling them based on the size of the grid and the size of the image.

Displaying Predictions and Ground Truth: We drew the predicted bounding boxes on the test images with labels indicating the predicted class. We also drew the ground truth bounding boxes on the images in a different plot.

To visually evaluate the performance of a YOLO model, we need to convert the predictions (the yolo_tensor) into actual bounding box coordinates and class labels.

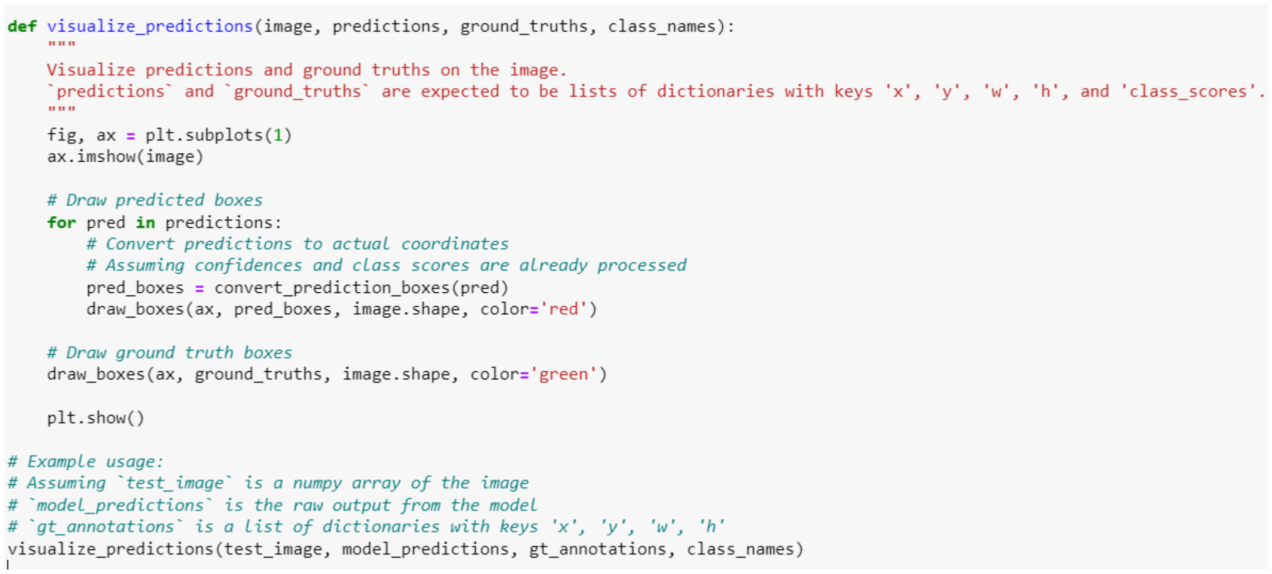
Selecting Boxes: Determine which boxes have a confidence score above a certain threshold to filter out weak detections.

Applying Non-Maximum Suppression (NMS): Among the boxes that predict the presence of an object, we use NMS to select the most accurate box and discard others that overlap significantly with it.

Class Prediction: Determine the class with the highest score for each box after NMS.

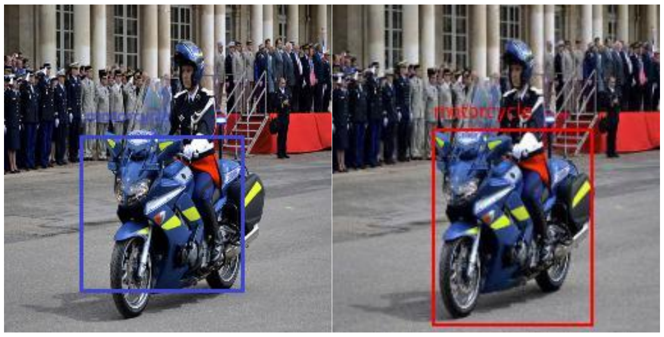
Rescaling Bounding Boxes: Rescale the bounding box coordinates to the original image dimensions.

```python
def convert_prediction_boxes(predictions, S=7, B=2, C=3):
    """
    Convert predictions into bounding box coordinates, confidences, and class scores.
    Returns a list of dictionaries for each grid cell and each bounding box.
    """
    predictions = predictions.view(-1, S, S, B, 5+C)
    converted_preds = []

    for i in range(S):
        for j in range(S):
            for b in range(B):
                x, y, w, h, conf = predictions[..., i, j, b, :5]
                class_scores = predictions[..., i, j, b, 5:]
                x = (x + j) / S  # Offset by the grid cell location
                y = (y + i) / S
                w = w ** 2  # YOLO predicts sqrt(w) and sqrt(h)
                h = h ** 2

                converted_preds.append({
                    'x': x.item(),
                    'y': y.item(),
                    'w': w.item(),
                    'h': h.item(),
                    'conf': conf.item(),
                    'class_scores': class_scores.view(-1).tolist(),
                })

    return converted_preds

def draw_boxes(ax, boxes, image_shape, color='blue'):
    """
    Draw bounding boxes on the matplotlib axis `ax`.
    """
    for box in boxes:
        x1 = (box['x'] - box['w'] / 2) * image_shape[1]
        y1 = (box['y'] - box['h'] / 2) * image_shape[0]
        width = box['w'] * image_shape[1]
        height = box['h'] * image_shape[0]

        rect = patches.Rectangle((x1, y1), width, height, linewidth=1, edgecolor=color, facecolor='none')
        ax.add_patch(rect)
```

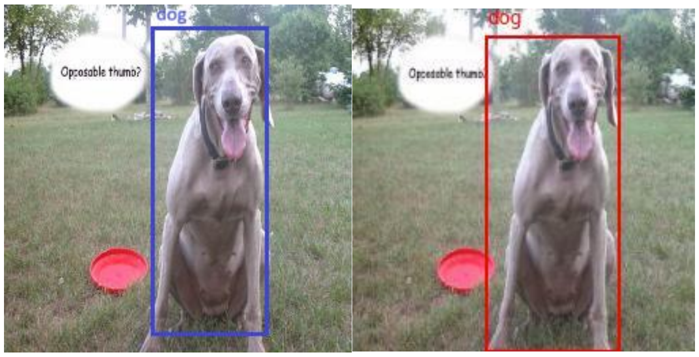


```python
def visualize_predictions(image, predictions, ground_truths, class_names):
    """
    Visualize predictions and ground truths on the image.
    `predictions` and `ground_truths` are expected to be lists of dictionaries with keys 'x', 'y', 'w', 'h', and 'class_scores'.
    """
    fig, ax = plt.subplots(1)
    ax.imshow(image)

    # Draw predicted boxes
    for pred in predictions:
        # Convert predictions to actual coordinates
        # Assuming confidences and class scores are already processed
        pred_boxes = convert_prediction_boxes(pred)
        draw_boxes(ax, pred_boxes, image.shape, color='red')

    # Draw ground truth boxes
    draw_boxes(ax, ground_truths, image.shape, color='green')

    plt.show()

# Example usage:
# Assuming `test_image` is a numpy array of the image
# `model_predictions` is the raw output from the model
# `gt_annotations` is a list of dictionaries with keys 'x', 'y', 'w', 'h'
visualize_predictions(test_image, model_predictions, gt_annotations, class_names)
```

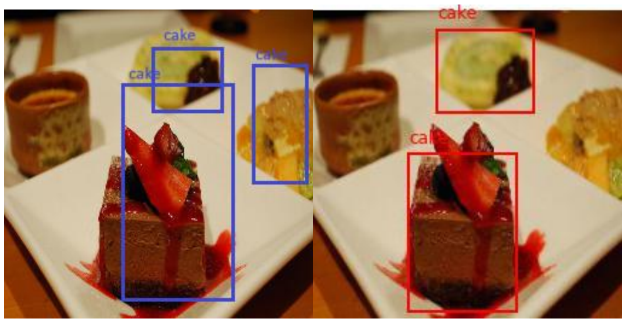
Best multi-instance results:



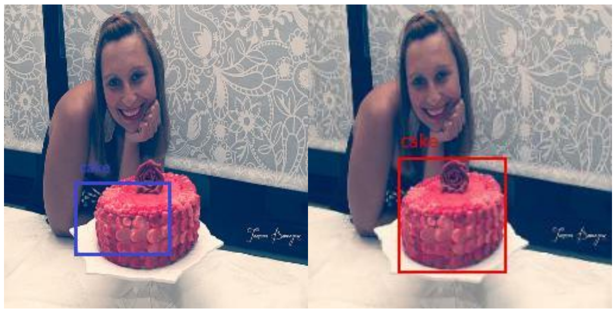
Pred GT



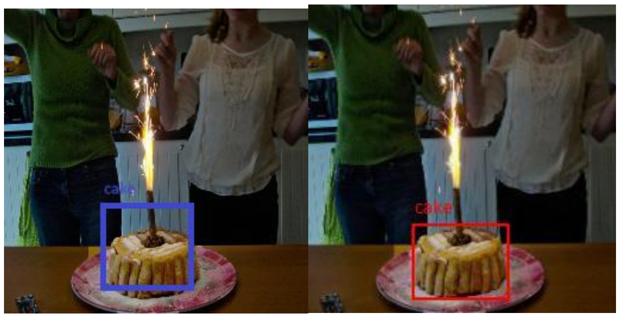
Pred GT



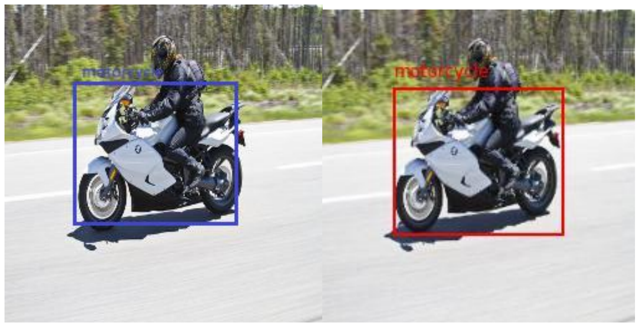
Pred GT



Pred GT

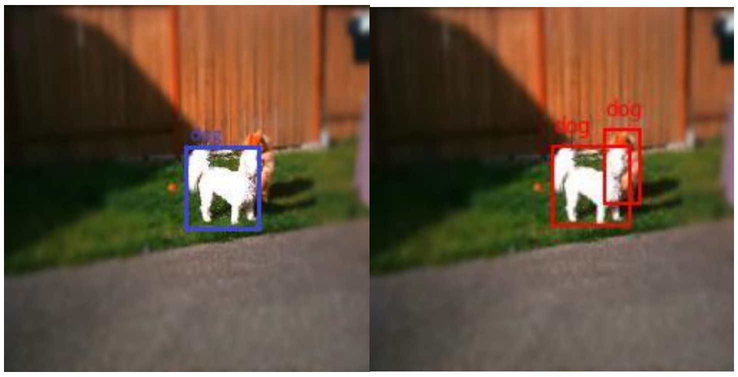


Pred GT

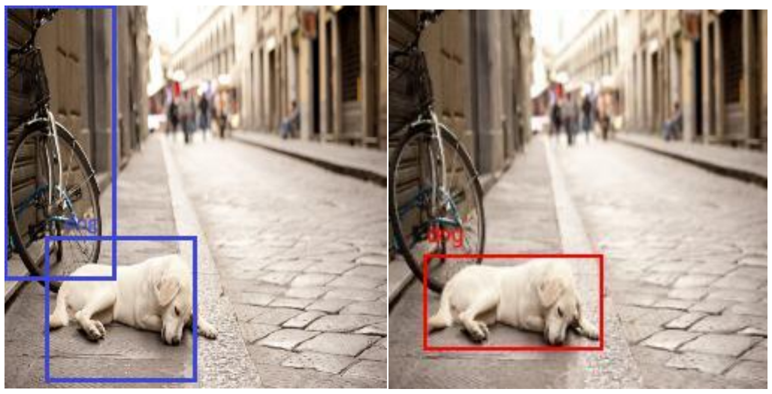


Pred GT

Poor multi-instance results:



Pred GT



Pred GT

Our YOLO detector underperforms when it encounters objects that are partially occluded or when distinguishing between similar categories, such as differentiating a bike from a motorcycle. These issues could be attributed to insufficient training on examples of partial occlusion and closely related object classes. To enhance its performance, we can enrich the dataset with a higher representation of these challenging scenarios or a more sophisticated architecture to improve its capacity.