ECE60146: Homework 6

Manish Kumar Krishne Gowda, 0033682812

(Spring 2024)

1 Introduction

In this assignment,our primary goal was to construct a object detection model for identifying specific classes, namely cakes, dogs, and motorcycles. I studied both single-object detection and multi-instance object detection methods using available code in YoLo Logic Module [1].

For single-object detection, I ran the object_detection_and_localization.py script during the checkpoint submission. Meanwhile, multi-instance object detection was done through the implementation of a custom code derived from YOLO logic, using the instances of images having multiple object from the CoCo Dataset [2] with the three classes.

I started by identifying where the objects are located in the images and what they are labeled as, using annotation information from the COCO dataset. I created a custom dataloader specifically designed for our own dataset and created a Resnet based architecture using skipblock studied in HW4 and HW5. Training was done executed using bothMSE and CIOU loss functions.

Next sections of the report explains the approach of dataset creation, dataloader implementation, model architecture, and the other specific details of the training and testing processes.

2 Methodology

2.1 Using the COCO Annotations

The following code shows how to access the required COCO annotation entries and display a randomly chosen cake image with desired annotations for visual verification. This is referenced from the HW task instructions. The sample output is shown in Figure 1.

```
    import skimage
import skimage.io as io
import cv2

input_json = '/content/drive/MyDrive/hw6_data/coco/annotations/instances_train2017
                                                .json'
class_list = ['cake']
# ##########################
# Mapping from COCO label to Class indices
coco_labels_inverse = {}
coco = COCO(input_json)
catIds = coco.getCatIds(catNms = class_list)
categories = coco.loadCats(catIds)

categories.sort(key = lambda x: x['id'])
print(categories)

for idx, in_class in enumerate(class_list):
    for c in categories:
        if c['name'] == in_class:
            coco_labels_inverse[c['id']] = idx
print(coco_labels_inverse)

# ##########################
```

Figure 1: Sample display of a randomly chosen image with desired annotations

```
# Retrieve Image list
imgIds = coco.getImgIds(catIds = catIds)

# ############################
# Display one random image with annotation
idx = np.random.randint(0, len(imgIds))
img = coco.loadImgs(imgIds[idx])[0]
I = io.imread(img['coco_url'])
if len(I.shape) == 2:
    I = skimage.color.gray2rgb(I)
annIds = coco.getAnnIds(imgIds = img['id'], catIds = catIds, iscrowd = False)
anns = coco.loadAnns(annIds)
fig, ax = plt.subplots(1, 1)
image = np.uint8(I)
for ann in anns :
    [x, y, w, h] = ann['bbox']
    label = coco_labels_inverse[ann['category_id']]
    image = cv2.rectangle(image, (int(x), int(y)), (int(x + w), int (y + h)), (36,
                                                    255, 12), 2)
    image = cv2.putText(image, class_list[label], (int(x), int(y - 10)), cv2.
                                                   FONT_HERSHEY_SIMPLEX,0.8, (36, 255,
                                                   12), 2)
ax.imshow(image)
ax.set_axis_off()
plt.axis('tight')
plt.show()
```

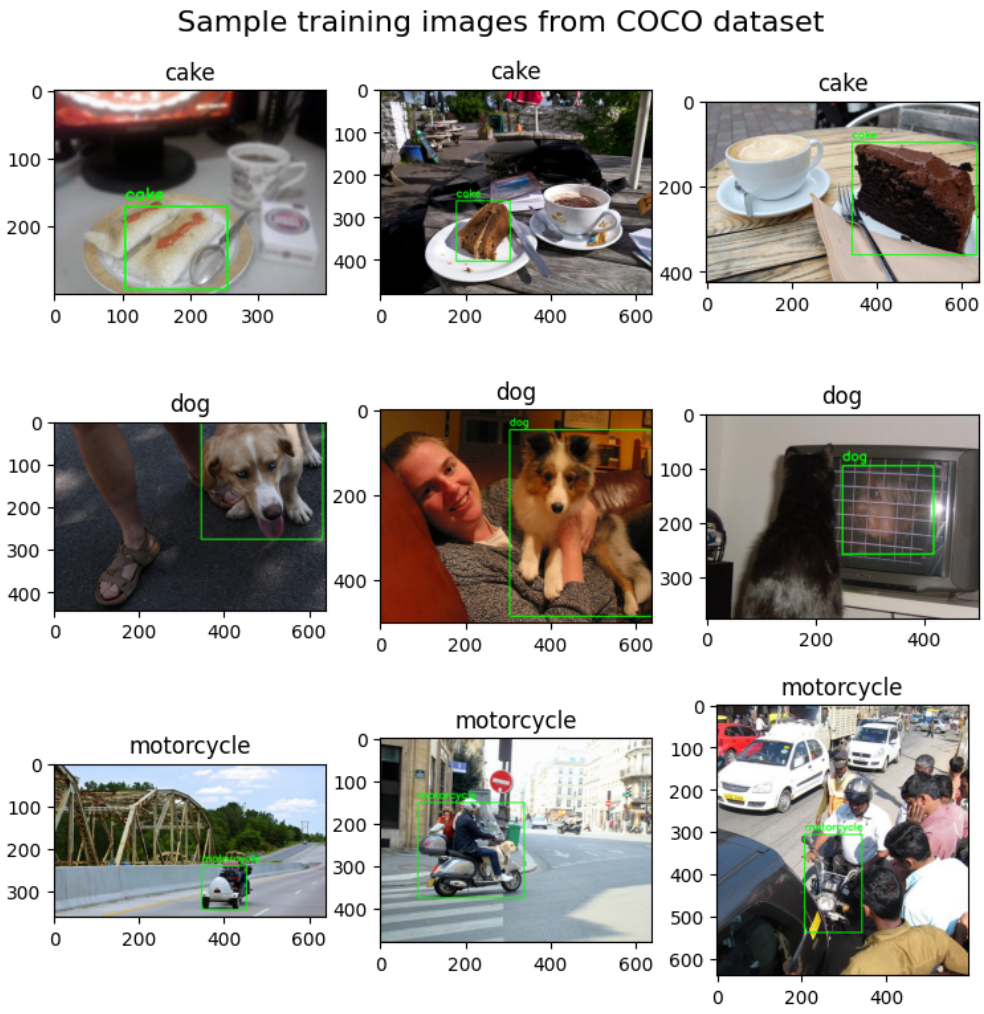

Figure 2: 3x3 sample training images

2.2 Creating Your Own Multi-Instance Object Localization Dataset

Own dataset was created for Multi-Instance Object Localization. Following script similar to HW4 that filters through the images and annotations to generate training and testing dataset. The image contains atleast 1 forground image containing 'objects' cake, dog and motorcycle. The parameter "min_annotation_area = 64 * 64" ensures that the image is of atleast 64x64 pixel. ALthough HW instructions said "When saving your images to disk, resize them to 256 ×256," this was avoided as changing the images to different size might lead to loss of info of certain images, say for example images which are of smaller size. So, instead, the image was resized while training and testing using reshape attribute of torchvision.transforms. Number of images in the training dataset = 8523 and Number of images in the validation dataset = 360.

Sample training and validation images with their annotations are shown in Figure 2 and 3

```
import os
import tqdm
from PIL import Image
import requests
from pycocotools.coco import COCO  # Assuming this is the COCO library in use
```

Figure 3: 3x3 sample validation images

```
class ImageDownloader():
    def __init__(
        self, root_dir, annotation_path, classes, min_annotation_area=64 * 64
    ):
        self.root_dir = root_dir
        self.annotation_path = annotation_path
        self.classes = classes
        self.min_annotation_area = min_annotation_area

        self.coco = COCO(annotation_path)
        self.catIds = self.coco.getCatIds(catNms=classes)
        self.categories = self.coco.loadCats(self.catIds)
        self.categories.sort(key=lambda x: x["id"])
        self.class_dir = {}

        self.coco_labels_inverse = {
            c["id"]: idx for idx, c in enumerate(self.categories)
        }

    def create_directories(self):
        for c in self.classes:
            dir_path = os.path.join(self.root_dir, c)
            self.class_dir[c] = dir_path
            os.makedirs(dir_path, exist_ok=True)

    def download_images(self, download=True, val=False):
        img_paths = {c: [] for c in self.classes}
        img_anns = {c: [] for c in self.classes}

        for c in tqdm.tqdm(self.classes):
            class_id = self.coco.getCatIds(c)
            img_ids = self.coco.getImgIds(catIds=class_id)
            images = self.coco.loadImgs(img_ids)

            for image in images:
                annIds = self.coco.getAnnIds(
                    imgIds=image["id"], catIds=class_id, iscrowd=False
                )
                annotations = self.coco.loadAnns(annIds)

                valid_annotations = [
                    ann
                    for ann in annotations
                    if ann["area"] >= self.min_annotation_area
                ]

                if valid_annotations:
                    boxes = [ann["bbox"] for ann in valid_annotations]
                    labels = [
                        self.coco_labels_inverse[ann["category_id"]]
                        for ann in valid_annotations
                    ]

                    img_path = os.path.join(
                        self.root_dir, c, image["file_name"]
                    )
                    if download:
                        if self.download_image(img_path, image["coco_url"]):
```

```
                        self.convert_image(img_path)
                        img_paths[c].append(img_path)
                        img_anns[c].append({"boxes": boxes, "labels": labels})
                else:
                    img_paths[c].append(img_path)
                    img_anns[c].append({"boxes": boxes, "labels": labels})

        return img_paths, img_anns

    # Download image from URL using requests
    def download_image(self, path, url):
        try:
            img_data = requests.get(url).content
            with open(path, 'wb') as f:
                f.write(img_data)
            return True
        except Exception as e:
            print(f"Caught exception: {e}")
        return False

    # Convert image
    def convert_image(self, path):
        im = Image.open(path)
        if im.mode != "RGB":
            im = im.convert(mode="RGB")
        im.save(path)
```

2.3 Building Your Deep Neural Network

Like HW instructions briefly mentioned, ResNet has two different kinds of skip blocks, named BasicBlock and BottleNeck. BasicBlock is used as a building-block in ResNet-18 and ResNet-34. In light of this, the SkipBlock studied in last HWs was used as a BasicBlock and using this basic block a custom Resnet called HW6Net was constructed. The Network was based on [3]. HW6Net consists of an Encoder Backbone and Prediction Layers.

2.3.1 Encoder Backbone

1. The architecture starts with a ReflectionPad2d layer followed by a convolutional layer, batch normalization, and ReLU activation.

2. It then includes a series of downsampling layers, each consisting of a convolutional layer, batch normalization, and ReLU activation, which helps in reducing the spatial dimensions of the feature maps.

3. Next, a series of Basic blocks are added to the model. The number of blocks is determined by the n_blocks parameter.

2.3.2 Prediction Layer

1. Following the encoder backbone, the architecture includes prediction layers to generate the final output.

2. These layers consist of convolutional, pooling, batch normalization, ReLU, flattening, and fully connected (linear) layers.

3. The final layer outputs a tensor of shape (batch_size, 2880)

In total, the network contains 62 layers and a whopping 46935872 (i.e. 46 Million) Trainable Parameters (I know it is less compared to many industry models, but the highest across all the HWs we have worked on so far :)

```
    # Define HW6Net architecture
class HW6Net(nn.Module):
    """ Resnet - based encoder that consists of a few
    downsampling + several Resnet blocks as the backbone
    and two prediction heads .
    """

    def __init__ (self, input_nc, ngf = 8, n_blocks = 4):
        """
        Parameters :
        input_nc (int) -- the number of channels input images
        output_nc (int) -- the number of channels output images
        ngf (int ) -- the number of filters in the first conv layer
        n_blocks (int) -- the number of ResNet blocks
        """
        assert(n_blocks >= 0)
        super(HW6Net, self). __init__ ()
        # The first conv layer
        model = [
            nn.ReflectionPad2d(3),
            nn.Conv2d(input_nc, ngf, kernel_size = 7, padding = 0),
            nn.BatchNorm2d(ngf),
            nn.ReLU(True)
        ]
        # Add downsampling layers
        n_downsampling = 4
        for i in range(n_downsampling):
            mult = 2 ** i
            model += [
                nn.Conv2d(ngf * mult, ngf * mult * 2, kernel_size = 3, stride = 2,
                                                        padding = 1),
                nn.BatchNorm2d(ngf * mult * 2),
                nn.ReLU(True)
            ]
        # Add your own ResNet blocks
        mult = 2 ** n_downsampling
        for i in range(n_blocks):
            model += [BasicBlock(ngf * mult, ngf * mult, downsample = False)]
        self.model = nn.Sequential(*model)

        # Prediction Layers
        pred_layers = [
            nn.Conv2d(ngf * mult, ngf * mult, kernel_size = 3, padding = 1),
            nn.MaxPool2d(2, 2),
            nn.ReLU(inplace=True),
            nn.Conv2d(ngf * mult, ngf * mult, kernel_size = 3, padding = 1),
            nn.BatchNorm2d(ngf * mult),
            nn.ReLU(inplace=True),
            nn.Flatten(),
            nn.Linear(128*8*8, 4096),
            nn.ReLU(inplace=True),
            nn.Linear(4096, 2880)
        ]
```

```
        self.pred_layer = nn.Sequential(*pred_layers)

    def forward (self, input):
        ft = self.model(input)
        x = self.pred_layer(ft)
        return x
```

2.4 Yolo Tensor

The YOLO (You Only Look Once) logic for object detection involves dividing the input image into a grid, associating anchor boxes with each grid cell, and predicting bounding boxes and class probabilities for objects within those cells. The following info (combining info from prof Kak's slides and my specific code implementation)

- **Grid Division** : The input image (of size 256x256) is divided into a grid of cells (32x32), and each cell is responsible for predicting objects within its boundaries.
- **Anchor Boxes**: Anchor boxes are predefined bounding box shapes of different aspect ratios. Each grid cell is associated with multiple anchor boxes to detect objects of varying sizes.
- **Output Tensor Dimension:** The output (of the network) tensor's dimensions depend on the size of the grid cell for the anchor boxes. In this case, the YOLO interval is selected as 32, and the image tensor has dimensions of 256x256, giving $\frac{256\times256}{32\times32} = 64$ cells per image.
- **Anchor Boxes and YOLO Vector:** Each cell is associated with 5 anchor boxes. Each anchor box produces a YOLO vector of size 9, including the background class(1x1). This vector contains information about the bounding box coordinates (1x4), objectness score(1x1), and class probabilities(one hot vector of size 1x3).
- **Calculation of Total Nodes in Output:** The total number of nodes in the output of the last linear layer is calculated by multiplying the number of cells, the number of anchor boxes per cell, and the size of the YOLO vector. Thus total Nodes $= 64 \times 5 \times 9 = 2880$
- **Batch Size Adjustment:** The output tensor has dimensions (B,2880), where B is the batch size (64 in my case).

2.5 Training and Evaluating The Network

The code for training and evaluating the network was adopted from previous year solutions [3].

2.5.1 data loader

For each image annotation, ground box coordinates are extracted and converted into a YOLO vector. This conversion involves creating a grid of size 32x32 and placing 5 anchor boxes with specific aspect ratios. By evaluating the Complete Box Intersection over Union (IoU) values of ground truth boxes with these anchors, each ground truth box is assigned to a particular grid cell. The YOLO vector is then computed and returned, following the format [1, δx, δy, σ_x, σ_y, 0, 0, 0] $\in 1 \times 9$. The data loader not only includes the YOLO tensor and the transformed image tensor but also returns the index of the assigned grid cell and the corresponding anchor box index. This data loader is configured to process images in batches of 64 for both training and validation phases.

2.5.2 trainnet

Three different loss functions are used Binary Cross Entropy (BCE) loss for object presence. Cross-entropy loss for class labeling and MSE or CIOU loss for bounding box regression. Following are some of the key variables in the logic

- **yolo_interval**: The interval for the YOLO grid cells.
- **num_yolo_cells**:The total number of YOLO grid cells generated in the image.
- **num_anchor_boxes**:The number of anchor boxes associated with each grid cell.
- **max_obj_num**: Maximum number of objects considered in a single image.

A tensor yolo_tensor is created to store YOLO vectors for each anchor box in each YOLO grid cell. An additional class is introduced in the YOLO vector to represent the absence of an object. As the forward pass is performed using the neural network (net) on input images (im_tensor) predictions are reshaped to match the YOLO tensor shape. Loss is calculated in a nested loop over YOLO cells and anchor boxes. Losses are accumulated separately for BCE, regression, and class labeling for reporting purposes.

2.5.3 validate_net

If a model path is provided, the pre-trained weights are loaded. The model is moved to the specified device (gpu/cpu) and set to evaluation mode. For each batch in the validation data loader: Images (im_tensor), ground truth bounding boxes (bbox_tensor), and ground truth labels (bbox_label_tensor) are extracted. The model processes the input images to produce predictions. The YOLO vectors are analyzed to determine the best anchor box for each YOLO cell. Retained cells are sorted based on the objectness score, and the top 5 cells are selected. Predicted bounding boxes and labels are extracted based on the retained cells. Softmax is applied to predict class labels. There is an early stop mechanism (break statement) after processing the first batch. This is useful for quick validation during development.

2.6 MSE Loss results

Network was trained for 60 epochs with bounding box regression loss as the Mean Square Error(MSE) loss and optimizer as Adam having parameters $\beta_1 = 0.5$ $\beta_2 = 0.999$. The learning rate was kept high (0.01) for initial epochs, while it was reduced for finetuning in later epochs. Ground truth and predicted boxes for sample validation images is shown in Figure 4.

Image in top left and bottom right provide the best results, while that in the bottom left, middle left and middle centre relatively worse. The true ground truth box is reported in green, while the predicted box is in red. The validation reported a object detection accuracy of about 36%, which explain why the class labels corresponding to the predictions (green labels) do not match with their true labels (green)

The training loss is shown in Figure 5.

2.7 CIOU Loss results

Network was trained for 10 epochs with bounding box regression loss as the Complete IOU (CIOU) loss and optimizer as Adam having parameters $\beta_1 = 0.9$ $\beta_2 = 0.999$ and learning rate 0.001. Ground truth and predicted boxes for sample validation images is shown in Figure 6.

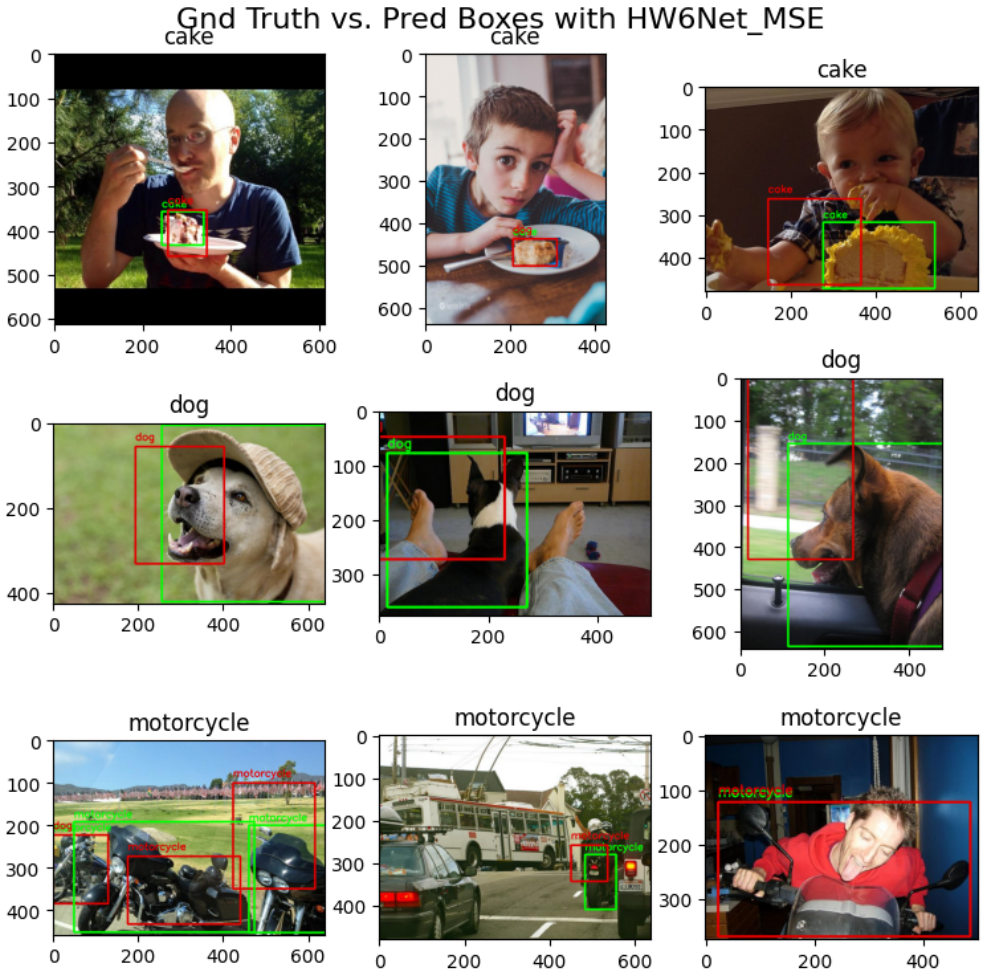

Figure 4: MSE regression sample images

Figure 5: MSE training loss

Overall CIOU performs poorly comaared to the MSE regression results as we can see the bounding boxe predictions are way off compared to the ground truth boxes. This can be attributed to the lesser amount of training (only 10 epochs) as opposed to over 50 epochs for MSE regression.

3 Conclusion

In conclusion, this report documents the development and training of a custom object detection model using the YOLO logic for detecting cakes, dogs, and motorcycles in images. The methodology involved creating a dedicated dataset and data loader, designing a custom neural network architecture (HW6Net), and implementing a training routine with multiple loss functions. The YOLO logic, involving anchor boxes and grid cells, was used for object localization. The validation routine provided insights into the model's performance. The custom data processing pipeline, combined with the designed neural network, showcases the effectiveness of the approach for multi-object detection tasks. The report contributes to the understanding of YOLO-based object detection and demonstrates the model's capability in identifying and localizing diverse objects in images.

4 Code

```
# -*- coding: utf-8 -*-
"""HW6_2.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1YZAOl9GG6vVj3v1PGZSqpIi-hQUPzNoA
"""

from google.colab import drive
drive.mount('/content/drive')
```

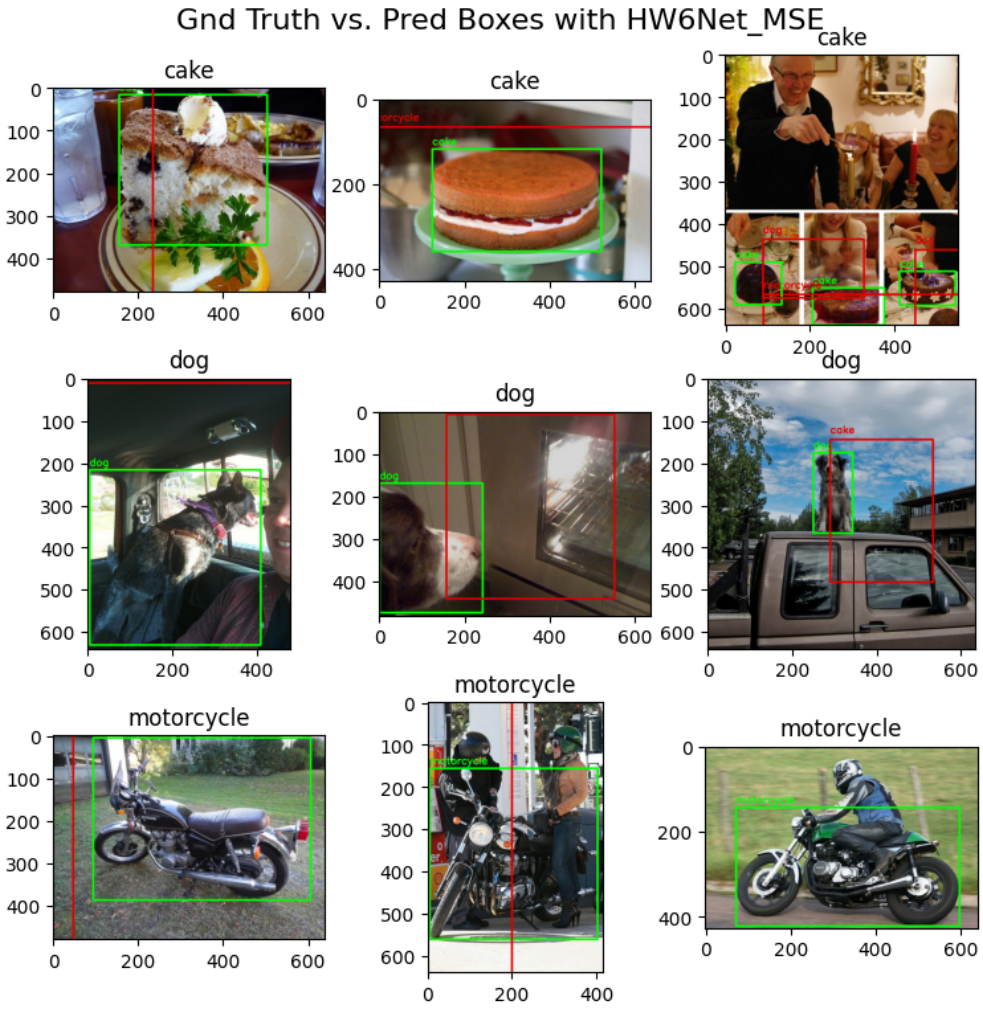

Figure 6: CIOU regression sample images

```
# Commented out IPython magic to ensure Python compatibility.
# %cd /content/drive/My Drive/hw6_data/

!wget -O RegionProposalGenerator-2.1.0.tar.gz \
    https://engineering.purdue.edu/kak/distRPG/RegionProposalGenerator-2.1.0.tar.
                                                gz?download

!tar -xvf RegionProposalGenerator-2.1.0.tar.gz

!wget -O /content/datasets_for_RPG.tar.gz \
    https://engineering.purdue.edu/kak/distRPG/datasets_for_RPG.tar.gz

!tar -xvf /content/datasets_for_RPG.tar.gz -C /content/drive/MyDrive/hw6_data/
                                            RegionProposalGenerator-2.1.0/
                                            ExamplesObjectDetection

!tar -xvf /content/drive/MyDrive/hw6_data/RegionProposalGenerator-2.1.0/
                                            ExamplesObjectDetection/data/
                                            Purdue_Dr_Eval_Multi_Dataset-clutter-10-
                                            noise-20-size-10000-train.gz -C /content/
                                            drive/MyDrive/hw6_data/
                                            RegionProposalGenerator-2.1.0/
                                            ExamplesObjectDetection/data/
!tar -xvf /content/drive/MyDrive/hw6_data/RegionProposalGenerator-2.1.0/
                                            ExamplesObjectDetection/data/
                                            Purdue_Dr_Eval_Multi_Dataset-clutter-10-
                                            noise-20-size-1000-test.gz -C /content/
                                            drive/MyDrive/hw6_data/
                                            RegionProposalGenerator-2.1.0/
                                            ExamplesObjectDetection/data/

# Commented out IPython magic to ensure Python compatibility.
# %cd /content/drive/MyDrive/hw6_data/RegionProposalGenerator-2.1.0

!pip install pymsgbox
!python setup.py install

import os
import torch
import random
import numpy as np
import requests
import matplotlib.pyplot as plt
from tqdm import tqdm
from PIL import Image
from pycocotools.coco import COCO

seed = 0
random.seed(seed)
np.random.seed(seed)

# Commented out IPython magic to ensure Python compatibility.
from RegionProposalGenerator import *
# %cd /content/drive/MyDrive/hw6_data/RegionProposalGenerator-2.1.0/
                                            ExamplesObjectDetection/

!python multi_instance_object_detection.py
```

```
# Commented out IPython magic to ensure Python compatibility.
# %cd /content/drive/My Drive/hw6_data//DLStudio-2.3.4
!python setup.py install
from DLStudio import *
# %cd /content/drive/My Drive/hw6_data/DLStudio-2.3.4/Examples
!python object_detection_and_localization.py

import skimage
import skimage.io as io
import cv2

input_json = '/content/drive/MyDrive/hw6_data/coco/annotations/instances_train2017
                                                .json'
class_list = ['cake']
# ##########################
# Mapping from COCO label to Class indices
coco_labels_inverse = {}
coco = COCO(input_json)
catIds = coco.getCatIds(catNms = class_list)
categories = coco.loadCats(catIds)

categories.sort(key = lambda x: x['id'])
print(categories)

for idx, in_class in enumerate(class_list):
    for c in categories:
        if c['name'] == in_class:
            coco_labels_inverse[c['id']] = idx
print(coco_labels_inverse)

# ############################
# Retrieve Image list
imgIds = coco.getImgIds(catIds = catIds)

# ############################
# Display one random image with annotation
idx = np.random.randint(0, len(imgIds))
img = coco.loadImgs(imgIds[idx])[0]
I = io.imread(img['coco_url'])
if len(I.shape) == 2:
    I = skimage.color.gray2rgb(I)
annIds = coco.getAnnIds(imgIds = img['id'], catIds = catIds, iscrowd = False)
anns = coco.loadAnns(annIds)
fig, ax = plt.subplots(1, 1)
image = np.uint8(I)
for ann in anns :
    [x, y, w, h] = ann['bbox']
    label = coco_labels_inverse[ann['category_id']]
    image = cv2.rectangle(image, (int(x), int(y)), (int(x + w), int (y + h)), (36,
                                                255, 12), 2)
    image = cv2.putText(image, class_list[label], (int(x), int(y - 10)), cv2.
                                                FONT_HERSHEY_SIMPLEX,0.8, (36, 255,
                                                12), 2)
ax.imshow(image)
ax.set_axis_off()
plt.axis('tight')
plt.show()

!mkdir /content/drive/MyDrive/hw6_data/coco
```

```
!wget --no-check-certificate http://images.cocodataset.org/annotations/
                                        annotations_trainval2017.zip \
    -O /content/drive/MyDrive/hw6_data/coco/annotations_trainval2017.zip

!unzip /content/drive/MyDrive/hw6_data/coco/annotations_trainval2017.zip -d /
                                        content/drive/MyDrive/hw6_data/coco/

import os
import tqdm
from PIL import Image
import requests
from pycocotools.coco import COCO  # Assuming this is the COCO library in use

class ImageDownloader():
    def __init__(
        self, root_dir, annotation_path, classes, min_annotation_area=64 * 64
    ):
        self.root_dir = root_dir
        self.annotation_path = annotation_path
        self.classes = classes
        self.min_annotation_area = min_annotation_area

        self.coco = COCO(annotation_path)
        self.catIds = self.coco.getCatIds(catNms=classes)
        self.categories = self.coco.loadCats(self.catIds)
        self.categories.sort(key=lambda x: x["id"])
        self.class_dir = {}

        self.coco_labels_inverse = {
            c["id"]: idx for idx, c in enumerate(self.categories)
        }

    def create_directories(self):
        for c in self.classes:
            dir_path = os.path.join(self.root_dir, c)
            self.class_dir[c] = dir_path
            os.makedirs(dir_path, exist_ok=True)

    def download_images(self, download=True, val=False):
        img_paths = {c: [] for c in self.classes}
        img_anns = {c: [] for c in self.classes}

        for c in tqdm.tqdm(self.classes):
            class_id = self.coco.getCatIds(c)
            img_ids = self.coco.getImgIds(catIds=class_id)
            images = self.coco.loadImgs(img_ids)

            for image in images:
                annIds = self.coco.getAnnIds(
                    imgIds=image["id"], catIds=class_id, iscrowd=False
                )
                annotations = self.coco.loadAnns(annIds)

                valid_annotations = [
                    ann
                    for ann in annotations
                    if ann["area"] >= self.min_annotation_area
                ]
```

```
                if valid_annotations:
                    boxes = [ann["bbox"] for ann in valid_annotations]
                    labels = [
                        self.coco_labels_inverse[ann["category_id"]]
                        for ann in valid_annotations
                    ]

                    img_path = os.path.join(
                        self.root_dir, c, image["file_name"]
                    )
                    if download:
                        if self.download_image(img_path, image["coco_url"]):
                            self.convert_image(img_path)
                            img_paths[c].append(img_path)
                            img_anns[c].append({"boxes": boxes, "labels": labels})
                    else:
                        img_paths[c].append(img_path)
                        img_anns[c].append({"boxes": boxes, "labels": labels})

        return img_paths, img_anns

    # Download image from URL using requests
    def download_image(self, path, url):
        try:
            img_data = requests.get(url).content
            with open(path, 'wb') as f:
                f.write(img_data)
            return True
        except Exception as e:
            print(f"Caught exception: {e}")
        return False

    # Convert image
    def convert_image(self, path):
        im = Image.open(path)
        if im.mode != "RGB":
            im = im.convert(mode="RGB")
        im.save(path)

from typing import Dict, List
classes = ['cake', 'dog', 'motorcycle']
try:
    # Download training images
    train_downloader = ImageDownloader('/content/drive/MyDrive/hw6_data/coco/
                                        train2017',
                '/content/drive/MyDrive/hw6_data/coco/annotations/
                                                    instances_train2017.json'
                                                    ,
                classes)

    train_downloader.create_directories()
    train_img_paths, train_img_anns = train_downloader.download_images(download=
                                        False)

    # Access and process downloaded data
    num_cake_images = len(train_img_paths["cake"])
    num_dog_images = len(train_img_paths["dog"])
    num_mc_images = len(train_img_paths["motorcycle"])
```

```
    print()
    print(f"Number of downloaded 'cake' images: {num_cake_images}")
    print(f"Number of downloaded 'dog' images: {num_dog_images}")
    print(f"Number of downloaded 'motorcycle' images: {num_mc_images}")

    # Assuming there is at least one annotation for "cake"
    if num_cake_images > 0:
        cake_annotations = train_img_anns["cake"][0]  # Assuming valid index
        boxes = cake_annotations["boxes"]
        labels = cake_annotations["labels"]

        for annotation_idx in range(len(boxes)):
            print("Box:", boxes[annotation_idx])
            print("Label:", labels[annotation_idx])
            print("-" * 20)  # Optional separator

except Exception as e:
    print(f"An error occurred: {e}")

classes = ['cake', 'dog', 'motorcycle']
try:
    # Download valing images
    val_downloader = ImageDownloader('/content/drive/MyDrive/hw6_data/coco/val2017
                                     ',
                '/content/drive/MyDrive/hw6_data/coco/annotations/
                                                instances_val2017.json',
                classes)

    val_downloader.create_directories()
    val_img_paths, val_img_anns = val_downloader.download_images(download=False)

    # Access and process downloaded data
    num_cake_images = len(val_img_paths["cake"])
    num_dog_images = len(val_img_paths["dog"])
    num_mc_images = len(val_img_paths["motorcycle"])

    print()
    print(f"Number of downloaded 'cake' images: {num_cake_images}")
    print(f"Number of downloaded 'dog' images: {num_dog_images}")
    print(f"Number of downloaded 'motorcycle' images: {num_mc_images}")

    # Assuming there is at least one annotation for "cake"
    if num_cake_images > 0:
        cake_annotations = val_img_anns["cake"][0]  # Assuming valid index
        boxes = cake_annotations["boxes"]
        labels = cake_annotations["labels"]

        for annotation_idx in range(len(boxes)):
            print("Box:", boxes[annotation_idx])
            print("Label:", labels[annotation_idx])
            print("-" * 20)  # Optional separator
except Exception as e:
    print(f"An error occurred: {e}")

import matplotlib.pyplot as plt
import numpy as np
import cv2

def plot_sample_images(paths, annotations, title, figsize=(9, 9)):
```

```
    fig, axes = plt.subplots(3, 3, figsize=figsize)
    indices = list(range(50, 53))

    for i, cls in enumerate(classes):
        for j, ind in enumerate(indices):
            path = paths[cls][ind]
            im = cv2.imread(path)  # Load directly as RGB
            for box in annotations[cls][ind]["boxes"]:
                x, y, w, h = box
                cv2.rectangle(im, (int(x), int(y)), (int(x + w), int(y + h)), (0,
                                                        255, 0), 2)
                cv2.putText(
                    im, cls, (int(x), int(y - 10)), cv2.FONT_HERSHEY_SIMPLEX, 0.8,
                                                            (0, 255, 0), 2
                )
            axes[i][j].imshow(cv2.cvtColor(im, cv2.COLOR_BGR2RGB))  # Convert BGR
                                                    to RGB for display
            axes[i][j].set_title(cls)

    fig.suptitle(title, fontsize=16, y=0.92)
    plt.show()

# Plot sample training images
plot_sample_images(train_img_paths, train_img_anns, "Sample training images from
                                        COCO dataset")

# Plot sample validation images
plot_sample_images(val_img_paths, val_img_anns, "Sample validation images from
                                        COCO dataset")

import os
import torch
from torchvision.ops import box_iou, distance_box_iou, complete_box_iou

# Custom dataset class for COCO
# class CocoMultiObjectDetectionDataset is a custom dataset class designed for
                                        handling object detection task
class CocoMultiObjectDetectionDataset(torch.utils.data.Dataset):
    def __init__(self, root, paths, anns, max_objects = 5, transforms=None, mode =
                                            'train'):
        super().__init__()
        self.mode = mode
        self.root_dir = root
        self.classes = os.listdir(self.root_dir)
        self.transforms = transforms
        self.max_objects = max_objects

        self.class_to_idx = {'cake':0, 'dog':1, 'motorcycle':2}
        self.idx_to_class = {i:c for c, i in self.class_to_idx.items()}

        self.img_paths = []
        self.img_labels = []
        self.img_bboxes = []
        # Populate lists with data from provided paths and annotations
        for cls in self.classes:
            self.img_paths += paths[cls]

            boxes = [valid_anns['boxes'] for valid_anns in anns[cls]]
            labels = [valid_anns['labels'] for valid_anns in anns[cls]]
```

```
            self.img_labels += labels
            self.img_bboxes += boxes

        # YOLO-specific configurations and calculations
        #creatng grids and assigning their indices
        self.yolo_interval = 32
        self.num_yolo_cells = (256 // self.yolo_interval) * (256 // self.
                                                yolo_interval)
        self.cell_height = self.yolo_interval
        self.cell_width = self.yolo_interval
        self.num_cells_image_width = 256 // self.yolo_interval
        self.num_cells_image_height = 256 // self.yolo_interval

        cell_row_indx = list(range(self.num_cells_image_width))
        cell_col_indx = list(range(self.num_cells_image_height))
        self.yolocell_centers_w = torch.FloatTensor(cell_col_indx)*self.
                                                yolo_interval + float(self.
                                                yolo_interval) / 2.0
        self.yolocell_centers_h = torch.FloatTensor(cell_row_indx)*self.
                                                yolo_interval + float(self.
                                                yolo_interval) / 2.0

        self.aspect_ratios = [1/5.0, 1/3.0, 1.0, 3.0, 5.0]
        self.anchor_box_shapes = [[self.cell_width*np.sqrt(r), self.cell_height/np
                                                .sqrt(r)] for r in self.
                                                aspect_ratios]
        self.anchor_boxes = []

        for c_h in self.yolocell_centers_h:
            for c_w in self.yolocell_centers_w:
                for w, h in self.anchor_box_shapes:
                    x = c_w - w / 2.0 #Calculates the x-coordinate of the top-left
                                                            corner of the anchor
                                                            box.
                    y = c_h - h / 2.0 #Calculates the y-coordinate of the top-left
                                                            corner of the anchor
                                                            box.
                    self.anchor_boxes.append([x, y, x+w, y+h]) #self.
                                                        anchor_box_shapes is
                                                        a list of pairs [
                                                        width, height]
                                                        representing
                                                        different aspect
                                                        ratios for the anchor
                                                        boxes.

    def __len__(self):
        # Return the total number of images
        return len(self.img_paths)

    def __getitem__(self, index):
        #Obtains the image path, opens the image, converts it to RGB, and applies
                                                any specified transformations.
        index = index % len(self.img_paths)
        img_path = self.img_paths[index]
        im = Image.open(img_path).convert('RGB')
        W, H = im.size
        im_transformed = self.transforms(im)
```

```
        #Initializes tensors to store bounding boxes and labels for each object.
        bbox_tensor = torch.zeros(self.max_objects, 4, dtype=torch.float32)
        bbox_label_tensor = torch.zeros(self.max_objects, dtype=torch.uint8) + 4

        boxes = self.img_bboxes[index]
        labels = self.img_labels[index]

        num_boxes = len(boxes)
        num_objects_in_image = min(self.max_objects, num_boxes)

        for i in range(num_objects_in_image):
            box = self.get_bbox(boxes[i], H, W)
            bbox_label_tensor[i] = labels[i]
            bbox_tensor[i] = torch.FloatTensor(box)

        #Calculates the Intersection over Union (IOU) between the ground truth
                                                bounding boxes and anchor boxes.
        #Determines the index of the anchor box with the maximum IOU for each
                                                object.
        anchor_boxes_tensor = torch.FloatTensor(self.anchor_boxes)
        iou = complete_box_iou(bbox_tensor, anchor_boxes_tensor)
        max_ind = torch.argmax(iou, dim = 1)

        yolo_cell_index = torch.zeros(self.max_objects)
        anch_box_index = torch.zeros(self.max_objects)
        yolo_vectors = torch.zeros((self.max_objects, 8))

        anc_boxes_width = anchor_boxes_tensor[:,2] - anchor_boxes_tensor[:,0]
        anc_boxes_height = anchor_boxes_tensor[:,3] - anchor_boxes_tensor[:,1]
        anc_boxes_center_x = (anchor_boxes_tensor[:,2] + anchor_boxes_tensor[:,0])
                                                /2.0
        anc_boxes_center_y = (anchor_boxes_tensor[:,3] + anchor_boxes_tensor[:,1])
                                                /2.0

        obj_bb_width = bbox_tensor[:,2] - bbox_tensor[:,0]
        obj_bb_height = bbox_tensor[:,3] - bbox_tensor[:,1]
        obj_center_x = (bbox_tensor[:,2].float() + bbox_tensor[:,0].float()) / 2.0
        obj_center_y = (bbox_tensor[:,3].float() + bbox_tensor[:,1].float()) / 2.0

        for i in range(num_objects_in_image):
            if bbox_label_tensor[i].item() == 4:
                continue
            yolo_cell_index[i] = max_ind[i] // 5 #Determine the YOLO cell index
                                                    for each object based on the
                                                    maximum intersection index.
            ind = max_ind[i]

            #Calculate Relative Position and Size (Delta) between Object and
                                                    Anchor Box:
            del_x = (obj_center_x[i].float() - anc_boxes_center_x[ind].float()) /
                                                    self.yolo_interval
            del_y = (obj_center_y[i].float() - anc_boxes_center_y[ind].float()) /
                                                    self.yolo_interval
            # Objects in images can vary significantly in size.
            #Using the logarithm allows the model to handle both small and large
                                                    objects effectively,
            #as the relative size information becomes more prominent.
            bw = torch.log(obj_bb_width[i] / anc_boxes_width[ind])
            bh = torch.log(obj_bb_height[i] / anc_boxes_height[ind])
```

```
            yolo_vector = torch.FloatTensor([1, del_x.item(), del_y.item(), bw.
                                                    item(), bh.item(), 0, 0, 0])
            yolo_vector[5 + bbox_label_tensor[i].item()] = 1

            #Determine Anchor Box Index based on Aspect Ratio (AR):
            AR = float(anc_boxes_width[ind]) / float(anc_boxes_height[ind])
            if AR <= 0.2:
                anch_box_index[i] = 0
            if 0.2 < AR <= 0.5:
                anch_box_index[i] = 1
            if 0.5 < AR <= 1.5:
                anch_box_index[i] = 2
            if 1.5 < AR <= 4.0:
                anch_box_index[i] = 3
            if AR > 4.0:
                anch_box_index[i] = 4

            yolo_vectors[i] = yolo_vector
        if self.mode == 'test':
            return im_transformed, bbox_tensor, bbox_label_tensor
        return im_transformed, yolo_cell_index, anch_box_index, yolo_vectors

    def get_bbox(self, box, h, w):
        x_scale = 256.0/w
        y_scale = 256.0/h
        return [box[0]*x_scale, box[1]*y_scale, (box[0]+box[2])*x_scale, (box[1]+
                                                box[3])*y_scale]

import torchvision.transforms as tvt

# Set the desired image size for resizing
reshape_size = 256

# Define the data transformations
transforms = tvt.Compose([
    tvt.ToTensor(),  # Convert to PyTorch tensor
    tvt.Resize((reshape_size, reshape_size)),  # Resize to desired size
    tvt.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))  # Normalize pixel values
])

# Specify the COCO dataset paths (modify according to your dataset location)
train_data_path = "/content/drive/MyDrive/hw6_data/coco/train2017"  # Replace with
                                            your train data path
val_data_path = "/content/drive/MyDrive/hw6_data/coco/val2017"  # Replace with
                                          your val data path

# Create the training dataset
train_dataset = CocoMultiObjectDetectionDataset(
    train_data_path, train_img_paths, train_img_anns, transforms=transforms
)

# Get the number of images in the training dataset
train_dataset_size = len(train_dataset)
print(f"Number of images in the training dataset: {train_dataset_size}")

# Create the validation dataset
val_dataset = CocoMultiObjectDetectionDataset(
    val_data_path, val_img_paths, val_img_anns, transforms=transforms, mode="test"
```

```
)

# Get the number of images in the validation dataset
val_dataset_size = len(val_dataset)
print(f"Number of images in the validation dataset: {val_dataset_size}")

# Create custom training/validation dataloaders
train_data_loader = torch.utils.data.DataLoader(train_dataset, batch_size=64,
                                                shuffle=True, num_workers=2)
val_data_loader = torch.utils.data.DataLoader(val_dataset, batch_size=9, shuffle=
                                                False, num_workers=2)

# Check dataloaders
train_loader_iter = iter(train_data_loader)
img, yolo_cell_index, anch_box_index, yolo_vectors = next(train_loader_iter)

# Print information about the loaded batch
print('img has length:', len(img))  # Assuming img is a list of images
print('yolo_cell_index has length:', len(yolo_cell_index))
print('anch_box_index has length:', len(anch_box_index))
print('yolo_vectors has shape:', yolo_vectors.shape)

'''
The first dimension (64) corresponds to the batch size, meaning you have
                                                predictions for 64 different images in a
                                                single batch.
The second dimension (5) corresponds to the number of anchor boxes per image,
                                                indicating that the model predicts
                                                bounding boxes for 5 different anchor
                                                boxes.
The third dimension (8) corresponds to the elements of the prediction vector for
                                                each anchor box :
*   4 : bbox dimensions
*   1 : objectness score
*   3 : class one hot encoding
'''

import torch.nn as nn
# Routine to train a neural network
def train_net(device, net, optimizer, data_loader, criterion_bbox,
              model_name, epochs = 10, display_interval = 100):
    net = net.to(device)
    net.train()

    criterion1 = nn.BCELoss()
    criterion2 = criterion_bbox
    criterion3 = nn.CrossEntropyLoss()

    loss_running_record = []

    loss_1_running_record = []
    loss_2_running_record = []
    loss_3_running_record = []

    yolo_interval = 32
    num_yolo_cells = (256 // yolo_interval) * (256 // yolo_interval)
    num_anchor_boxes = 5
    max_obj_num = 5
```

```
for epoch in range(epochs):
    running_loss = 0.0
    running_loss_1 = 0.0
    running_loss_2 = 0.0
    running_loss_3 = 0.0
    for i, data in enumerate(data_loader):
        im_tensor, yolo_cell_index, anch_box_index, yolo_vectors = data
        batch_size = im_tensor.shape[0]
        yolo_tensor = torch.zeros(batch_size, num_yolo_cells, num_anchor_boxes
                                                            , 8)
        im_tensor = im_tensor.to(device)

        yolo_cell_index = yolo_cell_index.to(device)
        anch_box_index = anch_box_index.to(device)
        yolo_vectors = yolo_vectors.to(device)

        ## idx is for object index
        for idx in range(max_obj_num):
            for bx in range(batch_size):
                if yolo_vectors[bx][idx][0] == 0:
                    continue
                yolo_tensor[bx, int(yolo_cell_index[bx][idx]), int(
                                                        anch_box_index[bx][
                                                        idx])] = yolo_vectors
                                                        [bx][idx]
            yolo_tensor_aug = torch.zeros(batch_size, num_yolo_cells,
                                                    num_anchor_boxes, 9).
                                                    float().to(device)
            yolo_tensor_aug[:,:,:,:-1] = yolo_tensor

            ## If no object is present, throw all the prob mass into the extra
                                                     9th ele of yolo_vector
            c = (yolo_tensor_aug[:, :, :, 0] == 0)
            y = torch.zeros([yolo_tensor_aug[c].shape[0], 9]).to(device)
            y[:, -1] = 1
            yolo_tensor_aug[c] = y

        optimizer.zero_grad()
        output = net(im_tensor)
        predictions_aug = output.view(batch_size, num_yolo_cells,
                                                num_anchor_boxes, 9)
        loss = torch.tensor(0.0, requires_grad=True).float().to(device)
        loss_bce = torch.tensor(0.0, requires_grad=True).float().to(device)
        loss_reg = torch.tensor(0.0, requires_grad=True).float().to(device)
        loss_cls = torch.tensor(0.0, requires_grad=True).float().to(device)
        for icx in range(num_yolo_cells):
            for iax in range(num_anchor_boxes):
                pred_yolo_vector = predictions_aug[:,icx,iax]
                target_yolo_vector = yolo_tensor_aug[:,icx,iax]

                object_presence = nn.Sigmoid()(pred_yolo_vector[:, 0])
                target_for_prediction = target_yolo_vector[:, 0]
                bceloss = criterion1(object_presence, target_for_prediction)
                loss += bceloss
                loss_bce += bceloss.item()

                pred_regression_vec = pred_yolo_vector[:, 1:5]
                target_regression_vec = target_yolo_vector[:, 1:5]
```

```
                    regression_loss = criterion2(pred_regression_vec,
                                                 target_regression_vec
                                                 )
                    loss += regression_loss
                    loss_reg += regression_loss.item()


                    probs_vector = pred_yolo_vector[:, 5:]
                    target = torch.argmax(target_yolo_vector[:, 5:], dim = 1)
                    class_labeling_loss = criterion3(probs_vector, target)
                    loss += class_labeling_loss
                    loss_cls += class_labeling_loss.item()

            running_loss += loss.item()
            running_loss_1 += loss_bce.item()
            running_loss_2 += loss_reg.item()
            running_loss_3 += loss_cls.item()

            loss.backward()
            optimizer.step()

            if (i+1) % display_interval == 0:
                avg_loss = running_loss / display_interval
                avg_loss_1 = running_loss_1 / display_interval
                avg_loss_2 = running_loss_2 / display_interval
                avg_loss_3 = running_loss_3 / display_interval
                print(f"[epoch: {epoch + 1}, batch: {i + 1}] loss: {avg_loss:.3f},
                                                          loss_1: {avg_loss_1:.3f}
                                                         , loss_2: {avg_loss_2:.3f
                                                         }, loss_3: {avg_loss_3:.
                                                         3f}")
                loss_running_record.append(avg_loss)
                loss_1_running_record.append(avg_loss_1)
                loss_2_running_record.append(avg_loss_2)
                loss_3_running_record.append(avg_loss_3)

                running_loss = 0.0
                running_loss_1 = 0.0
                running_loss_2 = 0.0
                running_loss_3 = 0.0

    checkpoint_path = os.path.join('/content/drive/MyDrive/hw6_data/saved_models/'
                                   ,
                                   f'{model_name}.pt')
    torch.save(net.state_dict(), checkpoint_path)

    return loss_running_record, loss_1_running_record, loss_2_running_record,
                                            loss_3_running_record

import matplotlib.pyplot as plt
import numpy as np

def plot_loss(loss_record, presence_loss_record, regression_loss_record,
              class_loss_record, display_interval, model_name):

    plt.figure(figsize=(10, 6))

    # Find the largest loss value among all records
    largest_loss = max(np.max(loss_record), np.max(presence_loss_record),
```

```
                        np.max(regression_loss_record), np.max(class_loss_record)
                                                                )

    # Plot all loss curves
    plt.plot(np.arange(len(loss_record)) * display_interval, loss_record,
             label="Total Loss", color='blue', linewidth=2)
    plt.plot(np.arange(len(presence_loss_record)) * display_interval,
                                                presence_loss_record,
             label="Object Presence Loss", color='green', linewidth=1.5)
    plt.plot(np.arange(len(regression_loss_record)) * display_interval,
                                                regression_loss_record,
             label="Bounding Box Regression Loss", color='orange', linewidth=1.5)
    plt.plot(np.arange(len(class_loss_record)) * display_interval,
                                                class_loss_record,
             label="Class Probability Loss", color='red', linewidth=1.5)

    # Set title, labels, and limits, scaling y-axis based on largest_loss
    plt.title(f'Training Loss for {model_name}', fontsize=16)
    plt.xlabel('Iterations', fontsize=14)
    plt.ylabel('Loss', fontsize=14)
    plt.xlim(0, len(loss_record) * display_interval)
    plt.ylim(0, largest_loss * 1.1)  # Add a 10% margin for visual clarity

    plt.legend(fontsize=12)
    plt.grid(True)
    plt.tight_layout()
    plt.show()

# Routine to validate a neural network
def validate_net(device, net, data_loader, model_path = None):
    if model_path is not None:
        net.load_state_dict(torch.load(model_path))
    net = net.to(device)
    net.eval()

    device_cpu = torch.device('cpu')

    class_labels = ['cake', 'dog', 'motorcycle']

    imgs = []
    all_labels = []
    all_bboxes = []
    all_labels_pred = []
    all_bboxes_pred = []

    yolo_interval = 32
    num_yolo_cells = (256 // yolo_interval) * (256 // yolo_interval)
    num_anchor_boxes =  5

    cell_row_indx = list(range(8))
    cell_col_indx = list(range(8))
    yolocell_centers_w = torch.FloatTensor(cell_col_indx)*yolo_interval + float(
                                                yolo_interval) / 2.0
    yolocell_centers_h = torch.FloatTensor(cell_row_indx)*yolo_interval + float(
                                                yolo_interval) / 2.0
    ar = [1/5.0, 1/3.0, 1.0, 3.0, 5.0]
    anchor_box_shapes = [[yolo_interval*np.sqrt(r), yolo_interval/np.sqrt(r)] for
                                                r in ar]
    anchor_boxes = []
```

```
    for c_h in yolocell_centers_h:
        for c_w in yolocell_centers_w:
            for w, h in anchor_box_shapes:
                x = c_w - w / 2.0
                y = c_h - h / 2.0
                anchor_boxes.append([x, y, x+w, y+h])

    with torch.no_grad():
        for iter, data in enumerate(data_loader):
            im_tensor, bbox_tensor, bbox_label_tensor = data
            batch_size = im_tensor.shape[0]

            im_tensor = im_tensor.to(device)
            bbox_tensor = bbox_tensor.to(device_cpu)
            bbox_label_tensor = bbox_label_tensor.to(device_cpu)

            imgs += [im_tensor.to(device_cpu).numpy()]
            all_labels += [bbox_label_tensor.numpy()]
            all_bboxes += [bbox_tensor.numpy()]
            output = net(im_tensor)

            predictions = output.view(batch_size, num_yolo_cells, num_anchor_boxes
                                                    , 9)

            instance_bboxes_pred = []
            instance_bboxes_labels_pred = []
            for ibx in range(predictions.shape[0]):
                icx_2_best_anchor_box = {ic : None for ic in range(64)}
                for icx in range(predictions.shape[1]):
                    cell_predi = predictions[ibx, icx]
                    prev_best = 0
                    for anchor_bdx in range(cell_predi.shape[0]):
                        if cell_predi[anchor_bdx][0] > cell_predi[prev_best][0]:
                            prev_best = anchor_bdx
                    best_anchor_box_icx = prev_best
                    icx_2_best_anchor_box[icx] = best_anchor_box_icx
                sorted_icx_to_box = sorted(icx_2_best_anchor_box,
                            key=lambda x: predictions[ibx,x,icx_2_best_anchor_box[
                                                                    x]][0].item()
                                                                    , reverse=
                                                                    True)
                retained_cells = sorted_icx_to_box[:5]

                objects_detected = []
                predicted_bboxes  = []
                predicted_labels_for_bboxes = []
                predicted_label_index_vals = []
                for icx in retained_cells:
                    pred_vec = predictions[ibx,icx, icx_2_best_anchor_box[icx]]
                    class_labels_predi  = pred_vec[-4:]
                    class_labels_probs = torch.nn.Softmax(dim=0)(
                                                            class_labels_predi)
                    class_labels_probs = class_labels_probs[:-1]
                    if torch.all(class_labels_probs < 0.2):
                        predicted_class_label = None
                    else:
                        # Get the predicted class label:
                        best_predicted_class_index = (class_labels_probs ==
                                                        class_labels_probs
```

```
                                                        .max())
                        best_predicted_class_index = torch.nonzero(
                                                        best_predicted_class_index
                                                        , as_tuple=True)
                        predicted_label_index_vals.append(
                                                        best_predicted_class_index
                                                        [0].item())
                        predicted_class_label = class_labels[
                                                        best_predicted_class_index
                                                        [0].item()]
                        predicted_labels_for_bboxes.append(predicted_class_label)

                        w_anchor = yolo_interval * np.sqrt(ar[
                                                        icx_2_best_anchor_box
                                                        [icx]])
                        h_anchor = yolo_interval / np.sqrt(ar[
                                                        icx_2_best_anchor_box
                                                        [icx]])
                        ## Analyze the predicted regression elements:
                        pred_regression_vec = pred_vec[1:5].cpu()
                        del_x,del_y = pred_regression_vec[0], pred_regression_vec[
                                                        1]
                        h,w = torch.exp(pred_regression_vec[2]), torch.exp(
                                                        pred_regression_vec
                                                        [3])
                        h *= h_anchor
                        w *= w_anchor
                        cell_row_index =  icx // 8
                        cell_col_index =  icx % 8
                        bb_center_x = cell_col_index * yolo_interval  +
                                                        yolo_interval/2
                                                        +  del_x *
                                                        yolo_interval
                        bb_center_y = cell_row_index * yolo_interval  +
                                                        yolo_interval/2
                                                        +  del_y *
                                                        yolo_interval
                        bb_top_left_x =  int(bb_center_x - w / 2.0)
                        bb_top_left_y =  int(bb_center_y - h / 2.0)
                        predicted_bboxes.append([bb_top_left_x, bb_top_left_y, int
                                                        (w), int(h)])

                for pred_bbox in predicted_bboxes:
                    w,h = pred_bbox[2], pred_bbox[3]
                    pred_bbox[2] = pred_bbox[0] + w
                    pred_bbox[3] = pred_bbox[1] + h

                instance_bboxes_pred.append(predicted_bboxes)
                instance_bboxes_labels_pred.append(predicted_labels_for_bboxes)

            all_bboxes_pred+=instance_bboxes_pred
            all_labels_pred+=instance_bboxes_labels_pred
            break

    return imgs, all_bboxes, all_labels, all_bboxes_pred, all_labels_pred

# Defining BasicBlock
#the skipblock studied in class is used as a ResBlock
import torch.nn as nn
```

```
import torch.nn.functional as F
class BasicBlock(nn.Module):
    def __init__(self, in_ch, out_ch, downsample=False):
        super(BasicBlock, self).__init__()
        self.downsample = downsample
        self.in_ch = in_ch
        self.out_ch = out_ch
        self.conv1 = nn.Conv2d(in_ch, out_ch, 3, stride=1, padding=1)
        self.conv2 = nn.Conv2d(in_ch, out_ch, 3, stride=1, padding=1)
        self.bn1 = nn.BatchNorm2d(out_ch)
        self.bn2 = nn.BatchNorm2d(out_ch)
        self.relu = nn.LeakyReLU()
        if downsample:
            self.downsampler = nn.Conv2d(in_ch, out_ch, 1, stride=2)

    def forward(self, x):
        identity = x
        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)
        if self.in_ch == self.out_ch:
            out = self.conv2(out)
            out = self.bn2(out)
            out = self.relu(out)

        if self.downsample:
            out = self.downsampler(out)
            identity = self.downsampler(identity)

        if self.in_ch == self.out_ch:
            out = out + identity
        else:
            out[:,:self.in_ch,:,:] += identity
            out[:,self.in_ch:,:,:] += identity
        return out

# Define HW6Net architecture
class HW6Net(nn.Module):
    """ Resnet - based encoder that consists of a few
    downsampling + several Resnet blocks as the backbone
    and two prediction heads .
    """

    def __init__ (self, input_nc, ngf = 8, n_blocks = 4):
        """
        Parameters :
        input_nc (int) -- the number of channels input images
        output_nc (int) -- the number of channels output images
        ngf (int ) -- the number of filters in the first conv layer
        n_blocks (int) -- the number of ResNet blocks
        """
        assert(n_blocks >= 0)
        super(HW6Net, self). __init__ ()
        # The first conv layer
        model = [
            nn.ReflectionPad2d(3),
            nn.Conv2d(input_nc, ngf, kernel_size = 7, padding = 0),
            nn.BatchNorm2d(ngf),
            nn.ReLU(True)
```

```
        ]
        # Add downsampling layers
        n_downsampling = 4
        for i in range(n_downsampling):
            mult = 2 ** i
            model += [
                nn.Conv2d(ngf * mult, ngf * mult * 2, kernel_size = 3, stride = 2,
                                                                padding = 1),
                nn.BatchNorm2d(ngf * mult * 2),
                nn.ReLU(True)
            ]
        # Add your own ResNet blocks
        mult = 2 ** n_downsampling
        for i in range(n_blocks):
            model += [BasicBlock(ngf * mult, ngf * mult, downsample = False)]
        self.model = nn.Sequential(*model)

        # Prediction Layers
        pred_layers = [
            nn.Conv2d(ngf * mult, ngf * mult, kernel_size = 3, padding = 1),
            nn.MaxPool2d(2, 2),
            nn.ReLU(inplace=True),
            nn.Conv2d(ngf * mult, ngf * mult, kernel_size = 3, padding = 1),
            nn.BatchNorm2d(ngf * mult),
            nn.ReLU(inplace=True),
            nn.Flatten(),
            nn.Linear(128*8*8, 4096),
            nn.ReLU(inplace=True),
            nn.Linear(4096, 2880)
        ]

        self.pred_layer = nn.Sequential(*pred_layers)

    def forward (self, input):
        ft = self.model(input)
        x = self.pred_layer(ft)
        return x

def plot_validation_results(imgs, gt_bboxes, gt_labels, pred_bboxes, pred_labels,
                                        plot_title="Ground Truth vs. Predicted
                                        Boxes"):
    """
    Function to plot the validation results.

    Args:
        imgs (list): List of images (numpy arrays)
        gt_bboxes (list): List of ground truth bounding boxes (numpy arrays)
        gt_labels (list): List of ground truth labels (numpy arrays)
        pred_bboxes (list): List of predicted bounding boxes (numpy arrays)
        pred_labels (list): List of predicted labels (numpy arrays)
        plot_title (str, optional): Title for the visualization plot. Defaults to
                                                "Ground Truth vs. Predicted Boxes
                                                ".
    """

    num_images_to_plot = 9  # Adjust as needed
    fig, axes = plt.subplots(3, 3, figsize=(12, 12))  # Increase figure size for
                                            better visualization
```

```
    for i in range(num_images_to_plot):
        image_idx = i % len(imgs)
        image = imgs[image_idx]  # Assuming the first element in the list is the
                                                image data
        image = (image * 0.5 + 0.5) * 255  # Normalize and convert to uint8
        image = np.ascontiguousarray(image.transpose(1, 2, 0), dtype=np.uint8)  #
                                                Convert to OpenCV format

        for box_idx in range(len(gt_bboxes[image_idx])):
            # Draw ground truth boxes
            x1, y1, x2, y2 = gt_bboxes[image_idx][box_idx]
            image = cv2.rectangle(image, (int(x1), int(y1)), (int(x2), int(y2)), (
                                                    36, 255, 12), 2)

            # Draw predicted boxes with label if valid
            print(pred_bboxes[image_idx])
            if len(pred_bboxes[image_idx])>0 and box_idx < len(pred_bboxes[
                                                    image_idx]) and pred_bboxes[
                                                    image_idx][box_idx][0] > 0
                                                    and pred_bboxes[image_idx][
                                                    box_idx][1] > 0:
                X1, Y1, X2, Y2 = pred_bboxes[image_idx][box_idx]
                image = cv2.rectangle(image, (int(X1), int(Y1)), (int(X2), int(Y2)
                                                        ), (255, 0, 0), 2)
                image = cv2.putText(image, pred_labels[image_idx][box_idx], (int(
                                                        X1), int(Y1 - 10)), cv2.
                                                        FONT_HERSHEY_SIMPLEX, 0.8
                                                        , (255, 0, 0), 1)

        axes[i // 3, i % 3].imshow(image)
        axes[i // 3, i % 3].set_axis_off()

    fig.suptitle(plot_title, fontsize=16, y=0.95)
    plt.tight_layout()
    plt.show()

import os

# Define a function to load loss records if they exist
def load_previous_losses(model_name):
    loss_file = f'/content/drive/MyDrive/hw6_data/saved_models/{model_name}_losses
                                                .pkl'
    if os.path.exists(loss_file):
        import pickle
        with open(loss_file, 'rb') as f:
            prev_loss_running_record, prev_loss_1_running_record, \
                prev_loss_2_running_record, prev_loss_3_running_record = pickle.
                                                        load(f)
        return prev_loss_running_record, prev_loss_1_running_record, \
               prev_loss_2_running_record, prev_loss_3_running_record
    else:
        print(f"Previous loss records not found for {model_name}. Starting fresh."
                                                )
        return None, None, None, None

# Save the updated loss records (optional)
def save_losses(net_mse_losses, model_name):
    loss_file = f'/content/drive/MyDrive/hw6_data/saved_models/{model_name}_losses
                                                .pkl'
```

```
    import pickle
    with open(loss_file, 'wb') as f:
        pickle.dump(net_mse_losses, f)

# Initialize device
device = torch.device('cuda') if torch.cuda.is_available() else torch.device('cpu'
                                                )
print(device)

# Initialize HW6Net
net_mse = HW6Net(3) #HW6Net takes 3 input channels

#Load saved model weights if desired (uncomment and adjust path)
if os.path.exists('/content/drive/MyDrive/hw6_data/saved_models/HW6Net_MSE2.pt'):
    net_mse.load_state_dict(torch.load('/content/drive/MyDrive/hw6_data/
                                            saved_models/HW6Net_MSE2.pt',
                                            map_location=device))

criterion_bbox = nn.MSELoss()  # MSE for bounding box regression

# Initialize optimizer
optimizer = torch.optim.Adam(net_mse.parameters(), lr=0.5*1e-2, betas=(0.5, 0.99))

# Set training parameters
epochs = 1
display_interval = 5

# Display Number of Layers
num_layers = len(list(net_mse.parameters()))
print("Number of Layers:", num_layers)

# Display Number of Trainable Parameters
num_params = sum(p.numel() for p in net_mse.parameters() if p.requires_grad)
print("Number of Trainable Parameters:", num_params)

# Train HW6Net with MSE Loss
net_mse_losses = train_net(device, net_mse, optimizer=optimizer, data_loader=
                                        train_data_loader,criterion_bbox=
                                        criterion_bbox,
                        model_name='HW6Net_MSE2', epochs=epochs, display_interval=
                                                            display_interval)

net_mse_losses = list(net_mse_losses)
# Load previous loss records for 'HW6Net_MSE2' (replace with your actual model
                                        name)
prev_loss_running_record, prev_loss_1_running_record, prev_loss_2_running_record,
                                        prev_loss_3_running_record =
                                        load_previous_losses('HW6Net_MSE2')

# Append the previous losses if they exist
if prev_loss_running_record is not None:
    net_mse_losses[0] = prev_loss_running_record + net_mse_losses[0]
    net_mse_losses[1] = prev_loss_1_running_record + net_mse_losses[1]
    net_mse_losses[2] = prev_loss_2_running_record + net_mse_losses[2]
    net_mse_losses[3] = prev_loss_3_running_record + net_mse_losses[3]

# Save the loss records for future use (replace with your actual model name)
save_losses(net_mse_losses, 'HW6Net_MSE2')
```

```
# Plotting HW6Net_MSE training loss
plot_loss(net_mse_losses[0][:400], net_mse_losses[1][:400], net_mse_losses[2][:400
                                        ], net_mse_losses[3][:400],
                                        display_interval, 'HW6Net_CIOU')
#plot_loss(net_mse_losses[0][50:], net_mse_losses[1][50:], net_mse_losses[2][50:],
                                         net_mse_losses[3][50:], display_interval
                                        , 'HW6Net_MSE2')

plot_loss(net_mse_losses[0][:100], net_mse_losses[1][:100], net_mse_losses[2][:100
                                        ], net_mse_losses[3][:100],
                                        display_interval, 'HW6Net_CIOU')



device = torch.device('cuda') if torch.cuda.is_available() else torch.device('cpu'
                                        )
print(device)
save_path = '/content/drive/MyDrive/hw6_data/saved_models/HW6Net_MSE2.pt'

# Initialize HW6Net
net_mse = HW6Net(3) #HW6Net takes 3 input channels

# Load saved model weights if desired (uncomment and adjust path)
net_mse.load_state_dict(torch.load('/content/drive/MyDrive/hw6_data/saved_models/
                                        HW6Net_MSE2.pt', map_location=device))
imgs, gt_bboxes, gt_labels, pred_bboxes, pred_labels = validate_net(device,
                                        net_mse, val_data_loader, model_path =
                                        save_path)
img_list = [imgs[0][i] for i in range(imgs[0].shape[0]) ]
print(img_list[0].shape)
plot_validation_results(img_list, gt_bboxes[0], gt_labels, pred_bboxes,
                                        pred_labels, plot_title="Gnd Truth vs.
                                        Pred Boxes with HW6Net_MSE")

# Define CIoU Loss
from torchvision.ops import complete_box_iou_loss
class CIoULoss(nn.Module):
    def __init__(self):
        super(CIoULoss, self).__init__()

    def forward(self, outputs, targets):
        loss = complete_box_iou_loss(outputs, targets, reduction = 'mean')
        return loss

# Initialize device
device = torch.device('cuda') if torch.cuda.is_available() else torch.device('cpu'
                                        )
device = torch.device('cpu')
print(device)

# Initialize HW6Net
net_ciou = HW6Net(3) #HW6Net takes 3 input channels

#Load saved model weights if desired (uncomment and adjust path)
if os.path.exists('/content/drive/MyDrive/hw6_data/saved_models/HW6Net_CIOU.pt'):
    net_ciou.load_state_dict(torch.load('/content/drive/MyDrive/hw6_data/
                                            saved_models/HW6Net_CIOU.pt',
                                            map_location=device))
```

```
criterion_bbox = CIoULoss()  # MSE for bounding box regression

# Initialize optimizer
optimizer = torch.optim.Adam(net_ciou.parameters(), lr=5*1e-3, betas=(0.5, 0.999))

# Set training parameters
epochs = 2
display_interval = 5

# Display Number of Layers
num_layers = len(list(net_ciou.parameters()))
print("Number of Layers:", num_layers)

# Display Number of Trainable Parameters
num_params = sum(p.numel() for p in net_ciou.parameters() if p.requires_grad)
print("Number of Trainable Parameters:", num_params)

# Train HW6Net with CIOU Loss
net_ciou_losses = train_net(device, net_ciou, optimizer=optimizer, data_loader=
                                        train_data_loader,criterion_bbox=
                                        criterion_bbox,
                        model_name='HW6Net_CIOU', epochs=epochs, display_interval=
                                                                display_interval)

net_ciou_losses = list(net_ciou_losses)
# Load previous loss records for 'HW6Net_MSE2' (replace with your actual model
                                        name)
prev_loss_running_record, prev_loss_1_running_record, prev_loss_2_running_record,
                                        prev_loss_3_running_record =
                                        load_previous_losses('HW6Net_CIOU')

# Append the previous losses if they exist
if prev_loss_running_record is not None:
    net_ciou_losses[0] = prev_loss_running_record + net_ciou_losses[0]
    net_ciou_losses[1] = prev_loss_1_running_record + net_ciou_losses[1]
    net_ciou_losses[2] = prev_loss_2_running_record + net_ciou_losses[2]
    net_ciou_losses[3] = prev_loss_3_running_record + net_ciou_losses[3]

# Save the loss records for future use (replace with your actual model name)
save_losses(net_ciou_losses, 'HW6Net_CIOU')

# Plotting HW6Net_MSE training loss
plot_loss(net_ciou_losses[0][50:], net_ciou_losses[1][50:], net_ciou_losses[2][50:
                                        ], net_ciou_losses[3][50:],
                                        display_interval, 'HW6Net_CIOU')

device = torch.device('cuda') if torch.cuda.is_available() else torch.device('cpu'
                                        )
print(device)
save_path = '/content/drive/MyDrive/hw6_data/saved_models/HW6Net_CIOU.pt'

# Initialize HW6Net
net_ciou = HW6Net(3) #HW6Net takes 3 input channels

# Load saved model weights if desired (uncomment and adjust path)
net_ciou.load_state_dict(torch.load('/content/drive/MyDrive/hw6_data/saved_models/
                                        HW6Net_CIOU.pt', map_location=device))
imgs, gt_bboxes, gt_labels, pred_bboxes, pred_labels = validate_net(device,
                                        net_ciou, val_data_loader, model_path =
```

```
                                                save_path)
img_list = [imgs[0][i] for i in range(imgs[0].shape[0]) ]
print(img_list[0].shape)
plot_validation_results(img_list, gt_bboxes[0], gt_labels, pred_bboxes,
                                                pred_labels, plot_title="Gnd Truth vs.
                                                Pred Boxes with HW6Net_MSE")
```

References

[1] URL https://engineering.purdue.edu/kak/distYOLO/

[2] URL https://cocodataset.org/#download

[3] URL http://tinyurl.com/34k7jt5n