Homework 6 - Deep Learning (ECE 60146)

Souradip Pal
pal43@purdue.edu
PUID 0034772329

March 22, 2023

1 Introduction

In this homework, the programming tasks give an overview of using a deep neural network for multi-objection detection and localization. It introduces the idea of anchor boxes which are widely employed in YOLO-type networks for predicting the multiple objects in an image and their corresponding bounding box parameters. In this assignment, a Convolutional Neural Network is created consisting of Skip Connections to learn image features and an appropriate training subroutine was implemented to train the network for multi-object classification and localization. The network was trained and validated using a subset of images present in the COCO dataset(2014 version) which were downloaded with the help of **COCO API**, a library for managing the dataset. Finally, the performance of the network was compared manually by plotting the predicted and ground truth bounding boxes to understand which images were correctly or incorrectly labeled for each of the classes. Some ideas related to the logic of downloading the COCO dataset and the training and validation routines were taken from the source code of the **RegionProposalGenerator** module and from the previous year's solutions for completing this assignment.

2 Methodology

Initially, to get familiar with the code for training and testing object detection and localization networks, the scripts *multi_instance_object_detection.py* from the **ExamplesObjectDetection** directory in the **RegionProposalGenerator** module were run. This script illustrated how the images are divided into grids and anchor boxes are used to determine the location of multiple object instances in an image. Using a similar approach for the programming tasks, a custom training routine was created and used along with the *HW6Net* network for predicting the multiple objects and their bounding boxes. Next, images with multiple objects from certain categories were filtered based on the area of the object to form a subset of the COCO dataset for this assignment. Training and testing routines were created to train and log the losses in periodic checkpoints. For evaluation, a method was created to plot the predicted and ground truth bounding boxes from the predicted *yolo_tensor*. The following section gives a detailed description of each of the approaches and the results obtained from the experiments performed on the object detection ad localization models.

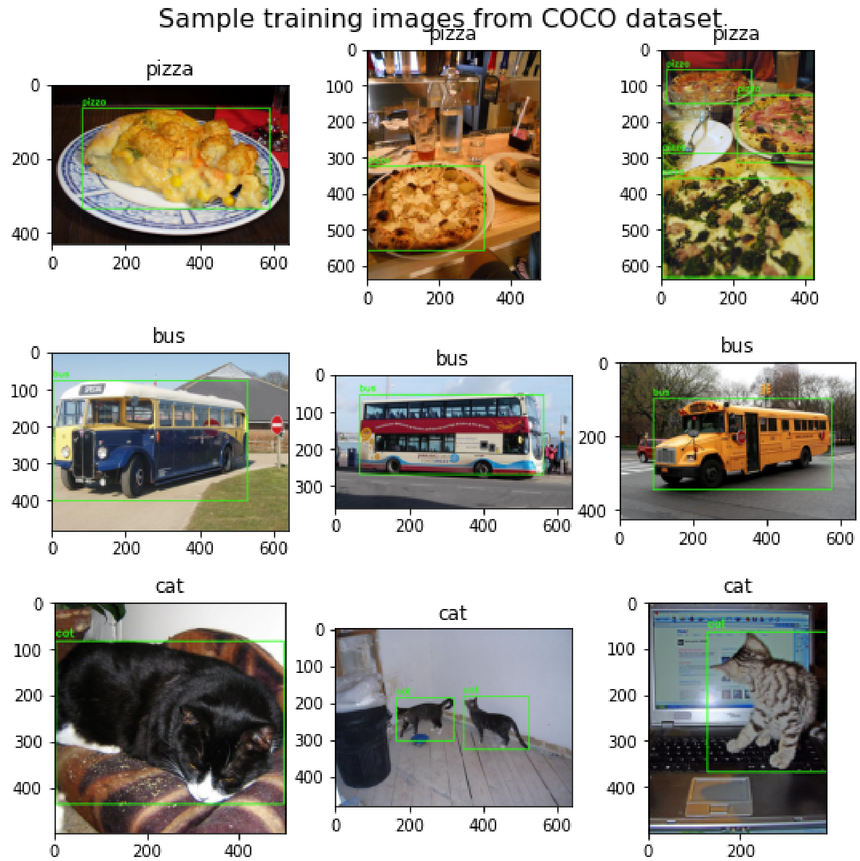

Figure 1: Sample of training images from the COCO dataset

3 Implementation and Results

3.1 Task 3: Programming Tasks

3.1.1 Creating Multi-Instance Object Localization Dataset

- In order to use the **COCO API** for managing the MS-COCO dataset, the python version of the API called *pycocotools* was downloaded.
- Since the 2014 version of the COCO dataset was used for this homework, the 2014 Train/Val annotation zip file was downloaded from the MS-COCO website. It contained a JSON file named *instances_train2014.json* which contained details of the images present in the *train2014.zip* and *val2014.zip* files including the image ids, image URLs, image categories, bounding box annotations, etc.
- Here, only the following three categories : ['pizza', 'bus', 'cat'] were selected. From these categories, the image ids were filtered based on the area of the objects. If an image consists of at least one foreground object and all the objects are from the above-mentioned categories having an area greater than 64×64 i.e. 4096 pixels then the image was picked up. The images are then downloaded using those URLs using the *requests* python library and saved in separate directories named after each of the categories. Moreover, the images were downsampled to a smaller size of 256×256 for training and validation. A total of **6198** training images and **3181** testing images

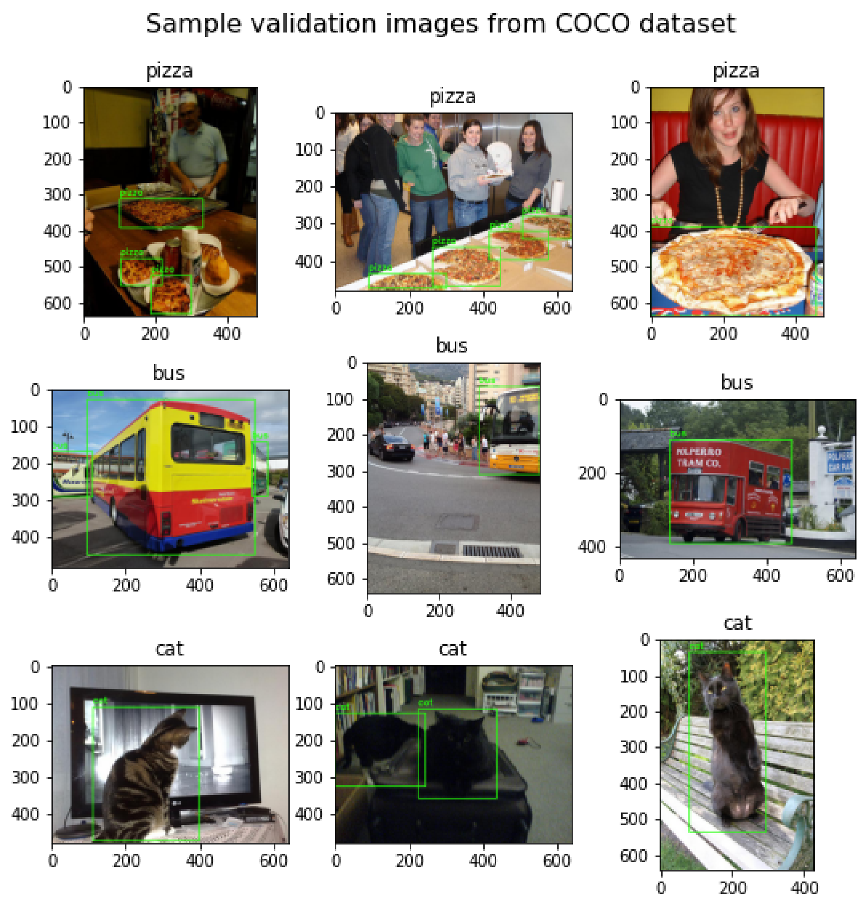

Figure 2: Sample of validation images from the COCO dataset

were obtained after filtering. Shown in Fig. 1 and Fig. 2 are samples of the images from the training and validation sets for each of the three classes with their corresponding ground truth boxes.

3.1.2 Building the Deep Neural Network

- **CNN Architecture**: In this task, a custom skip block network *ResBlock* and the *HW6Net* model were created similar to the one created in the previous homework assignment. The ResBlock consists of two convolutional layers followed by BatchNorm layers. In addition to that, LeakyReLU layers($\alpha = 0.01$) were used as activation functions. Each of the convolutional layers has a kernel size of 3 and padding of 1 to maintain the output shapes without downsampling. The following code block gives a description of the *ResBlock* network.

```
class ResBlock(nn.Module):
    def __init__(self, in_ch, out_ch, downsample=False):
        super(ResBlock, 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
```

The *ResBlock* layers were used sequentially in the network backbone. For the prediction of the *yolo_tensor*, a sequence of convolution followed by linear layers with ReLU activation function was created. The dimension of the output tensor depends on the number of objects to be detected and the size of the grid cell for the anchor boxes. Since the *yolo_interval* was selected as 32 and the height and width of the image tensor were 256 each, the total number of cells generated were $\frac{256\times256}{23\times32} = 64$. There are 5 anchor boxes associated with each of the cells and each of them produces a *yolo_vector* of size 9(including the background class). Thus the total number of nodes in the output of the last linear layer becomes $64 \times 5 \times 9 = 2880$ i.e (B, 2880) where B is the batch size. The total number of learnable layers of the entire network came out to be **62** and the total number of learnable parameters was $\sim$**46M**. The following code snippet shows the HW6Net network.

```
# 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 += [ResBlock(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
```

3.1.3 Training and Evaluating the Deep Neural Network

- **Dataloader**: A custom dataset class called *CocoMultiObjectDetectionDataset* was created from *torch.utils.data.Dataset* class to load the images from each of the class directories after applying the necessary transforms. For each of the image annotations, the ground box coordinates were extracted and converted into a *yolo_vector*. This was achieved by creating grids of size 32×32 and placing 5 anchor boxes with aspect ratios of $[1/5, 1/3, 1/1, 3/1, 5/1]$. Based on the **Complete Box IoU** values of the ground truth boxes with these anchors, the ground truth boxes were assigned a particular grid cell, and the *yolo_vector* having the format $[1, \partial x, \partial y, \sigma_w, \sigma_h, 0, 0, 0]$ was computed and returned. In addition to the *yolo_tensor* and the transformed image tensor, the index of the assigned grid cell and the corresponding anchor box index were also returned. Finally, a *Dataloader* was created to wrap the dataset for processing the images in batches of **64** for training and validation.
- **Training Routine -** Using the training routine created, the network was trained for 20 epochs

with regression loss as the Mean Square Error(MSE) loss and optimizer as Adam having parameters $\beta_1 = 0.9$ and $\beta_2 = 0.999$ and learning rate 0.001. In the training routine, the losses were computed based on the values of the predicted *yolo_vector*. Binary Cross Entropy loss is computed based on the objectness probability obtained from the first element in the *yolo_vector*. The next 4 elements are used for the MSE regression loss and the CrossEntropy loss is calculated based on the classification probabilities from the last 4 elements of the *yolo_vector*. Fig. 3 shows the training loss variation with the number of iterations while training *HW6Net* with MSE Loss.

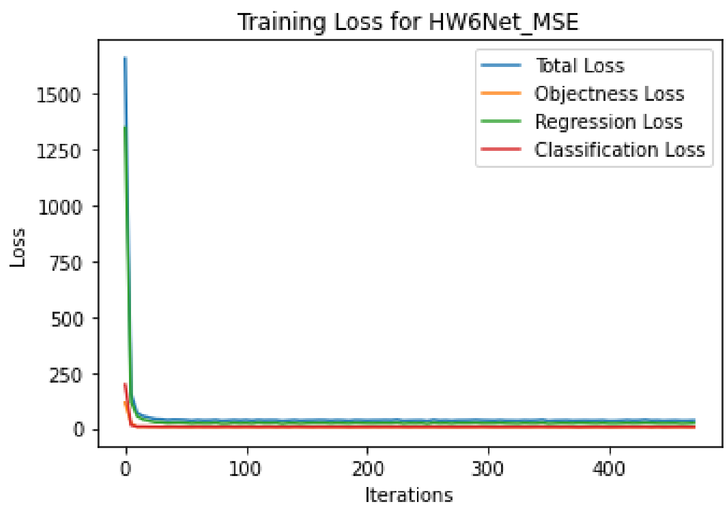

Figure 3: Training loss vs Iterations for *HW6Net* with MSE Loss

- **Evaluation Routine** - In order to validate the network, a validation routine was created where the predicted labels and the predicted bounding box coordinates were obtained via. model evaluation. A smaller subset of the validation dataset was created consisting of 3 images from each category and those images were fed into the network to obtain the prediction. Using those *yolo_vector* predictions, the predicted labels and the box coordinates were extracted and plotted for each category and evaluated qualitatively. Only the top 5 boxes with the highest classification scores were retained and others were discarded. Fig. 4 shows some of the predicted bounding boxes obtained from a subset of the validation image set after removing some of the invalid box predictions.
- **Results & Comparison**: It can be observed that some of the bounding boxes were not as accurate as expected. The main issues encountered in the regression were the overlapping boxes in cases where the objects were close together. For example, in the second cat image, the bounding box for only a single cat was predicted for two very close cats. The same is the case for the first pizza image. Also, in the first bus image, the object was predicted as a pizza which may be attributed to the object having similar image features to a pizza. These issues can be avoided by employing non-maximum suppression. Better results can also be obtained using IoU-type losses for regression instead of MSE loss. Moreover, due to resource and time constraints, the models were run only up to 20 epochs hence it was not able to provide the desired level of accuracy. Different learning rates could also be used to train the box head and class head modules which might lead to faster training and better performance.

Figure 4: Sample multi-object detections with *HW6Net* on COCO validation dataset

4 Conclusion

In conclusion, Yolo-type networks are a powerful way to capture image features and detect multiple objects in one-shot which makes the network perform well in multi-instance object detection and localization. However, sufficient care must be taken in the creation of grids for better flexibility in grid activations and the choice of anchor boxes with different aspect ratios might also be crucial. Hence, well-designed network architectures containing all these elements along with appropriate hyper-parameters are essential in making the training much more efficient and stable and getting higher classification and regression accuracy.

5 Source Code

```
# -*- coding: utf-8 -*-
"""hw6_SouradipPal.ipynb
```

```
Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1ScxuCS_1YOhKpy5EecXQ3mScTuqHY9Dk
"""

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

# Commented out IPython magic to ensure Python compatibility.
# %cd /content/drive/MyDrive/Purdue/ECE60146/

!wget -O RegionProposalGenerator-2.0.8.tar.gz \
    https://engineering.purdue.edu/kak/distRPG/RegionProposalGenerator-2.0.8.tar.gz?download

!tar -xvf RegionProposalGenerator-2.0.8.tar.gz

# Commented out IPython magic to ensure Python compatibility.
# %cd /content/drive/MyDrive/Purdue/ECE60146/RegionProposalGenerator-2.0.8

!pip install pymsgbox

!python setup.py install

!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/Purdue/ECE60146/RegionProposalGenerator-2.0.8/ExamplesObjectDetection/

!tar -xvf /content/drive/MyDrive/Purdue/ECE60146/RegionProposalGenerator-2.0.8/ExamplesObjectDetection/data/Purdue_Dr_Eval_Multi_Dataset-clutte
!tar -xvf /content/drive/MyDrive/Purdue/ECE60146/RegionProposalGenerator-2.0.8/ExamplesObjectDetection/data/Purdue_Dr_Eval_Multi_Dataset-clutte

# %load_ext autoreload
# %autoreload 2

!pip install pycocotools

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)

from RegionProposalGenerator import *

# Commented out IPython magic to ensure Python compatibility.
# %cd /content/drive/MyDrive/Purdue/ECE60146/RegionProposalGenerator-2.0.8/ExamplesObjectDetection/

!python multi_instance_object_detection.py

!mkdir /content/drive/MyDrive/Purdue/ECE60146/coco
!wget --no-check-certificate http://images.cocodataset.org/annotations/annotations_trainval2014.zip \
    -O /content/drive/MyDrive/Purdue/ECE60146/coco/annotations_trainval2014.zip

# !mkdir /content/drive/MyDrive/Purdue/ECE60146/HW6/data/

!rm -rf /content/drive/MyDrive/Purdue/ECE60146/HW6/data/train2014
!mkdir /content/drive/MyDrive/Purdue/ECE60146/HW6/data/train2014

!rm -rf /content/drive/MyDrive/Purdue/ECE60146/HW6/data/val2014
!mkdir /content/drive/MyDrive/Purdue/ECE60146/HW6/data/val2014
!mkdir /content/drive/MyDrive/Purdue/ECE60146/HW6/saved_models

!unzip /content/drive/MyDrive/Purdue/ECE60146/coco/annotations_trainval2014.zip -d /content/drive/MyDrive/Purdue/ECE60146/HW6/data/

import skimage
```

```
import skimage.io as io
import cv2

input_json = '/content/drive/MyDrive/Purdue/ECE60146/HW6/data/annotations/instances_train2014.json'
class_list = ['pizza']
# ##########################
# 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)
print(anns)
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()

# Image Downloader class to download COCO images
class ImageDownloader():
    def __init__(self, root_dir, annotation_path, classes):
        self.root_dir = root_dir
        self.annotation_path = annotation_path
        self.classes = classes

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

        self.coco_labels_inverse = {}
        for idx, in_class in enumerate(self.classes):
            for c in self.categories:
                if c['name'] == in_class:
                    self.coco_labels_inverse[c['id']] = idx
        print(self.coco_labels_inverse)

    # Create directories same as category names to save images
    def create_dir(self):
        for c in self.classes:
            dir = os.path.join(self.root_dir, c)
            self.class_dir[c] = dir
            if not os.path.exists(dir):
                os.makedirs(dir)

    # Download images
```

```
    def download_images(self, download = True, val = False):
        img_paths = {}
        img_anns = {}
        for c in tqdm(self.classes):
            img_paths[c] = []
            img_anns[c] = []
            class_id = self.coco.getCatIds(c)
            img_id = self.coco.getImgIds(catIds=class_id)
            imgs = self.coco.loadImgs(img_id)

            for i, id in enumerate(img_id):
                annIds = self.coco.getAnnIds(imgIds = id, catIds = class_id, iscrowd = False)
                anns = self.coco.loadAnns(annIds)

                invalid_anns = [ann['area'] < 64*64 or not ann['category_id'] in self.coco_labels_inverse.keys() for ann in anns]

                if len(anns) > 0 and not any(invalid_anns):
                    valid_anns = {}
                    boxes = []
                    labels = []
                    for ann in anns:
                        # valid_ann['image_id'] = ann['image_id']
                        # valid_ann['area'] = ann['area']
                        # valid_ann['iscrowd'] = ann['iscrowd']
                        # valid_ann['bbox'] = ann['bbox']
                        # valid_ann['label'] = self.coco_labels_inverse[ann['category_id']]
                        boxes.append(ann['bbox'])
                        labels.append(self.coco_labels_inverse[ann['category_id']])

                    valid_anns['boxes'] = boxes
                    valid_anns['labels'] = labels

                    img_path = os.path.join(self.root_dir, c, imgs[i]['file_name'])
                    if download:
                        done = self.download_image(img_path, imgs[i]['coco_url'])
                        if done:
                            self.convert_image(img_path)
                            img_paths[c].append(img_path)
                            img_anns[c].append(valid_anns)
                    else:
                        img_paths[c].append(img_path)
                        img_anns[c].append(valid_anns)


        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)

classes = ['pizza', 'bus', 'cat']

# Download training images
train_downloader = ImageDownloader('/content/drive/MyDrive/Purdue/ECE60146/HW6/data/train2014',
                '/content/drive/MyDrive/Purdue/ECE60146/HW6/data/annotations/instances_train2014.json',
                classes)
train_downloader.create_dir()
train_img_paths, train_img_anns = train_downloader.download_images(download = False)

len(train_img_paths['bus'])

train_img_anns['cat'][101]
```

```
# Download validation images
val_downloader = ImageDownloader('/content/drive/MyDrive/Purdue/ECE60146/HW6/data/val2014',
                '/content/drive/MyDrive/Purdue/ECE60146/HW6/data/annotations/instances_val2014.json',
                classes)
val_downloader.create_dir()
val_img_paths, val_img_anns = val_downloader.download_images(download = False, val = True)

print(len(val_img_paths['bus']), len(val_img_paths['cat']), len(val_img_paths['pizza']))

# Plotting sample training images
fig, axes = plt.subplots(3, 3, figsize=(9, 9))

indices = list(range(50, 53))

for i, cls in enumerate(classes):
    for j, ind  in enumerate(indices):
        path = train_img_paths[cls][ind]
        im = Image.open(path).convert('RGB')
        im = np.ascontiguousarray(im, dtype=np.uint8)
        for box in train_img_anns[cls][ind]['boxes']:
            [x, y, w, h] = box
            im = cv2.rectangle(im, (int(x), int(y)), (int(x + w), int (y + h)), (36, 255, 12), 2)
            im = cv2.putText(im, cls, (int(x), int(y - 10)), cv2.FONT_HERSHEY_SIMPLEX, 0.8, (36, 255, 12), 2)
        axes[i][j].imshow(im)
        axes[i][j].set_title(cls)

fig.suptitle('Sample training images from COCO dataset', fontsize=16, y=0.92)
plt.show()

# Plotting sample validation images
fig, axes = plt.subplots(3, 3, figsize=(9, 9))

indices = list(range(100, 103))
for i, cls in enumerate(classes):
    for j, ind in enumerate(indices):
        path = val_img_paths[cls][ind]
        im = Image.open(path).convert('RGB')
        im = np.ascontiguousarray(im, dtype=np.uint8)
        for box in val_img_anns[cls][ind]['boxes']:
            [x, y, w, h] = box
            im = cv2.rectangle(im, (int(x), int(y)), (int(x + w), int (y + h)), (36, 255, 12), 2)
            im = cv2.putText(im, cls, (int(x), int(y - 10)), cv2.FONT_HERSHEY_SIMPLEX,0.8, (36, 255, 12), 2)
        axes[i][j].imshow(im)
        axes[i][j].set_title(cls)

fig.suptitle('Sample validation images from COCO dataset', fontsize=16, y=0.95)
plt.show()

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

# Custom dataset class for COCO
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 = {'pizza':0, 'bus':1, 'cat':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 = []
        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
```

```
        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
                    y = c_h - h / 2.0
                    self.anchor_boxes.append([x, y, x+w, y+h])

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

    def __getitem__(self, index):
        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)
        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)

        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
            ind = max_ind[i]

            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
            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

            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

reshape_size = 256
transforms = tvt.Compose([
    tvt.ToTensor(),
    tvt.Resize((reshape_size, reshape_size)),
    tvt.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
    ])

# Create custom training dataset
train_dataset = CocoMultiObjectDetectionDataset('/content/drive/MyDrive/Purdue/ECE60146/HW6/data/train2014/',
                                           train_img_paths, train_img_anns, transforms=transforms)

len(train_dataset)

# Create custom validation dataset
val_dataset = CocoMultiObjectDetectionDataset('/content/drive/MyDrive/Purdue/ECE60146/HW6/data/val2014/',
                                         val_img_paths, val_img_anns, transforms=transforms, mode = 'test')

len(val_dataset)

# Checking yolo cell activation
im, yolo_cell_index, anch_box_index, yolo_vectors = train_dataset[2]

cell_row_id = yolo_cell_index // 8
cell_col_id = yolo_cell_index % 8

yolo_y = cell_row_id.numpy()*32
yolo_x = cell_col_id.numpy()*32

fig, axes = plt.subplots(1,1, figsize=(5, 5))
im = np.ascontiguousarray(im.numpy().transpose(1,2,0))

ar = [1/5.0, 1/3.0, 1.0, 3.0, 5.0]


for i in range(yolo_cell_index.shape[0]):

    if yolo_vectors[i][0] == 0:
        continue

    print(yolo_vectors[i])
    r = ar[int(anch_box_index[i])]
    w, h = 32*np.sqrt(r),32/np.sqrt(r)
    im = cv2.rectangle(im, (int(yolo_x[i]), int(yolo_y[i])), (int(yolo_x[i] + w), int (yolo_y[i] + h)), (36, 255, 12), 2)
    #im = cv2.putText(im, cls, (int(x), int(y - 10)), cv2.FONT_HERSHEY_SIMPLEX,0.8, (36, 255, 12), 2)
    axes.imshow(im)

plt.show()
```

```python
# Create custom training/validation dataloader
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=64, shuffle=False, num_workers=2)

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

print('img has length: ', len(img))
#print('target has length: ', len(label))
print(img[0].shape)

# Commented out IPython magic to ensure Python compatibility.
import torch.nn as nn
# Routine to train a neural network
def train_net(device, net, optimizer, data_loader,
              model_name, epochs = 10, display_interval = 100):
    net = net.to(device)
    net.train()

    criterion1 = nn.BCELoss()
    criterion2 = nn.MSELoss()
    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 ("[epoch : %d, batch : %5d] loss : %.3f loss_1 : %.3f loss_2 : %.3f loss_3 : %.3f" \
#                         % (epoch + 1, i + 1, avg_loss, avg_loss_1, avg_loss_2, avg_loss_3))
                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/Purdue/ECE60146/HW6/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

# Plotting training loss
def plot_loss(loss, loss_1, loss_2, loss_3, display_interval, model_name):
    plt.figure()
    plt.plot(np.arange(len(loss))*display_interval, loss, label="Total Loss");
    plt.plot(np.arange(len(loss_1))*display_interval, loss_1, label="Objectness Loss");
    plt.plot(np.arange(len(loss_2))*display_interval, loss_2, label="Regression Loss");
    plt.plot(np.arange(len(loss_3))*display_interval, loss_3, label="Classification Loss");
    plt.title(f'Training Loss for {model_name}')
    plt.xlabel('Iterations')
    plt.ylabel('Loss')
    plt.legend()
    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 = ['pizza', 'bus', 'cat']

    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 ResBlock
import torch.nn as nn
import torch.nn.functional as F

class ResBlock(nn.Module):
    def __init__(self, in_ch, out_ch, downsample=False):
        super(ResBlock, 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 += [ResBlock(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

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

# Initialize HW6Net
net = HW6Net(3)

net.load_state_dict(torch.load('/content/drive/MyDrive/Purdue/ECE60146/HW6/saved_models/HW6Net_MSE_2.pt',
                               map_location=device))

optimizer = torch.optim.Adam(net.parameters(), lr=5*1e-3, betas=(0.5, 0.999))
epochs = 20
display_interval = 5

# Display Number of Layers
num_layers = len(list(net.parameters()))
print(num_layers)

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

# Train HW6Net with MSE Loss
net1_losses = train_net(device, net, optimizer=optimizer, data_loader = train_data_loader,
```

```
                        model_name = 'HW6Net_MSE_2', epochs=epochs, display_interval = display_interval)

# Plotting HW6Net_MSE training loss
plot_loss(net1_losses[0], net1_losses[1], net1_losses[2], net1_losses[3], display_interval, 'HW6Net_MSE')

import pandas as pd

# list of name, degree, score
losses = net1_losses[0]
losses_1 = net1_losses[1]
losses_2 = net1_losses[2]
losses_3 = net1_losses[3]

# dictionary of lists
dict = {'losses': losses, 'losses_1': losses_1, 'losses_2': losses_2, 'losses_3': losses_3 }

df = pd.DataFrame(dict)

# saving the dataframe
df.to_csv('/content/drive/MyDrive/Purdue/ECE60146/HW6/saved_models/checkpoint.csv')

subset_indices = [100, 101, 104, 1440, 1441, 1443, 2478, 2479, 2480]
small_val_data_set = torch.utils.data.Subset(val_dataset, subset_indices)
len(small_val_data_set)

small_val_data_loader = torch.utils.data.DataLoader(small_val_data_set, batch_size=9, shuffle=False, num_workers=2)

# Validate HW6Net with MSE Loss
save_path = '/content/drive/MyDrive/Purdue/ECE60146/HW6/saved_models/HW6Net_MSE_2.pt'
imgs, gt_bboxes, gt_label, pred_bboxes, pred_label = validate_net(device, net, small_val_data_loader, model_path = save_path)

len(pred_bboxes)

# Plotting ground truth and predicted boxes for validation images with HW6Net_MSE
fig, axes = plt.subplots(3, 3, figsize=(9, 9))

for i in range(3):
    for j in range(3):
        ind = i*3+j
        I = imgs[0][ind]
        I = (I*0.5+0.5)*255
        image = np.ascontiguousarray(I.transpose(1,2,0), dtype=np.uint8)
        for b in range(5):
            [x1, y1, x2, y2] = gt_bboxes[0][ind][b]
            [X1, Y1, X2, Y2] = pred_bboxes[ind][b]
            image = cv2.rectangle(image, (int(x1), int(y1)), (int(x2), int(y2)), (36, 255, 12), 2)
            if X1 > 0 and Y1 > 0:
                image = cv2.rectangle(image, (int(X1), int(Y1)), (int(X2), int(Y2)), (255, 0, 0), 2)
                image = cv2.putText(image, pred_label[ind][b], (int(X1), int(Y1 - 10)), cv2.FONT_HERSHEY_SIMPLEX, 0.8, (255, 0, 0), 1)
            axes[i][j].imshow(image)
            axes[i][j].set_axis_off()

fig.suptitle('Ground truth and predicted boxes for sample validation images', fontsize=16, y=0.95)
plt.axis('tight')
plt.show()
```