ECE 60146
Homework 5

March 6, 2023

Alex Rogers
roger299@purdue.edu

1 Introduction

This assignment is an introduction into using skip connections and creating a network for single object detection as well as creating and utilizing a custom dataset for training. It also gives experience with a well known and used dataset such as COCO. It also gives a brief introduction into performance metrics for classifiers and bounding box applications.

2 Custom COCO Image Classification Dataset

The COCO dataset was downloaded locally and then was processed using the code in section 5.1. As described by the assignment, only the three classes *bus, cat, pizza* were kept with the condition that a single dominant object of the class greater than 40,000 pixels was present. The code was designed to ensure there was only one of these dominant objects in the image and to ensure there were no duplicate images. Separate datasets were created for the train and validation sets using the corresponding COCO versions. In total, there were 3788 training images and 1971 validation images. A figure showing examples of the training set with the dominant image highlighted via the bounding box can be found in 1. Note that the pizza image in the bottom right of the figure has a bounding box which takes up the entire image and thus it can't be seen in the outputted image.

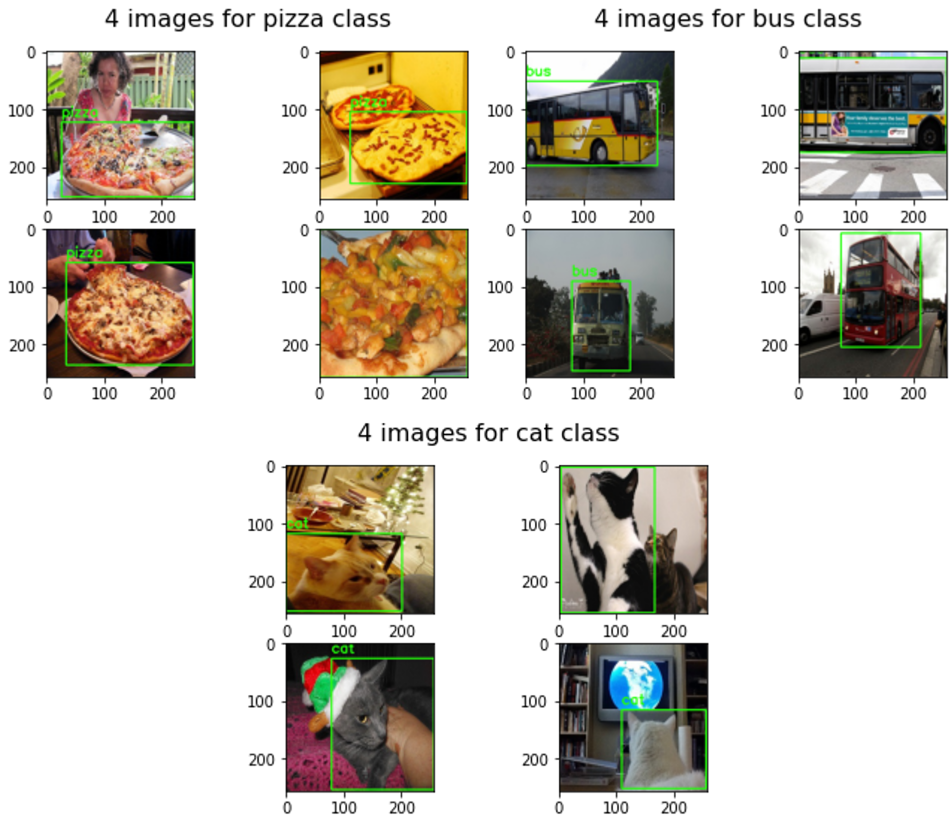

Figure 1: Four Images for Each Class in Custom Dataset

3 Image Classification Using CNNs

A custom dataloader was first created in order to easily load in the custom dataset. This along with the code for the network architecture, training, testing, and data visualization can be found in section 5.2. The `SkipBlock` class is in this section as well which outlines the details of the created skip connection.

3.1 Mean Squared Error Regression Network

The first network created and trained was one using mean squared error for the bounding box regression task. This network had a total of **144 layers** as determined by the `net.parameters()` logic we were given in the assignment. The loss of both the regression and classification tasks can be found in figure 2 where 20 epochs with a learning rate of $1e-4$ was used. The confusion matrix can be found in figure 3. The average classification accuracy for all three classes is **86.10%** and the mean IOU is **0.576**. An example of the ground truth and predicted bounding boxes for each image can be found in figure 4.

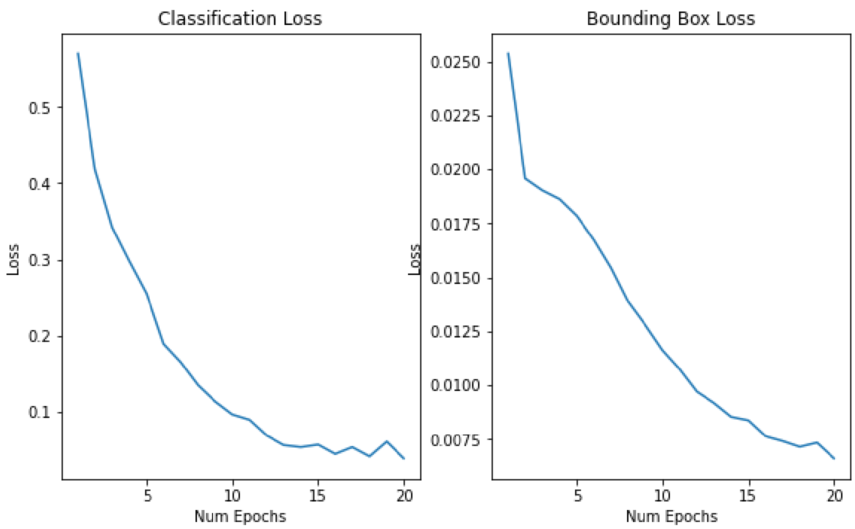

Figure 2: Average Epoch Loss For MSE Network

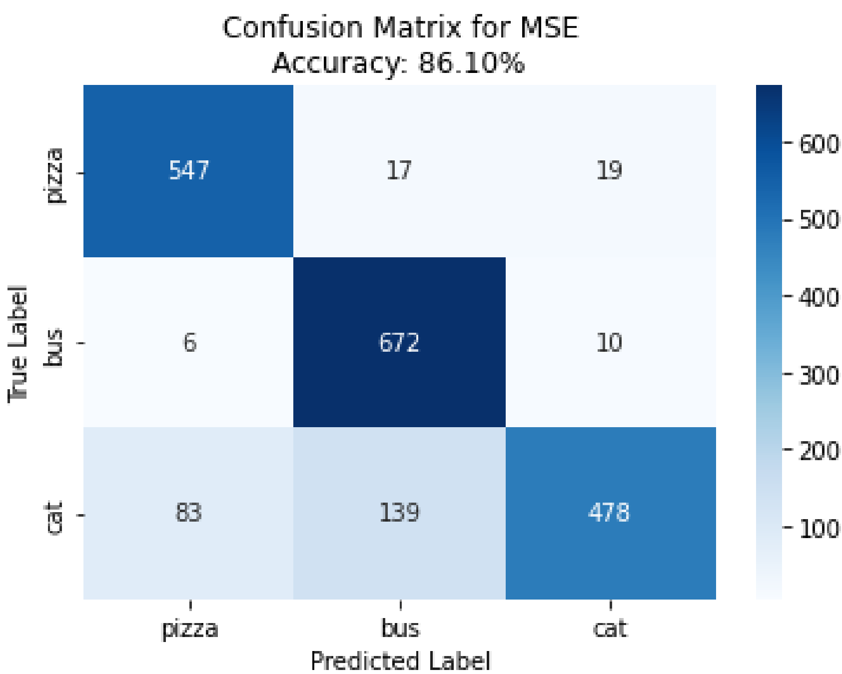

Figure 3: Confusion Matrix for MSE Network

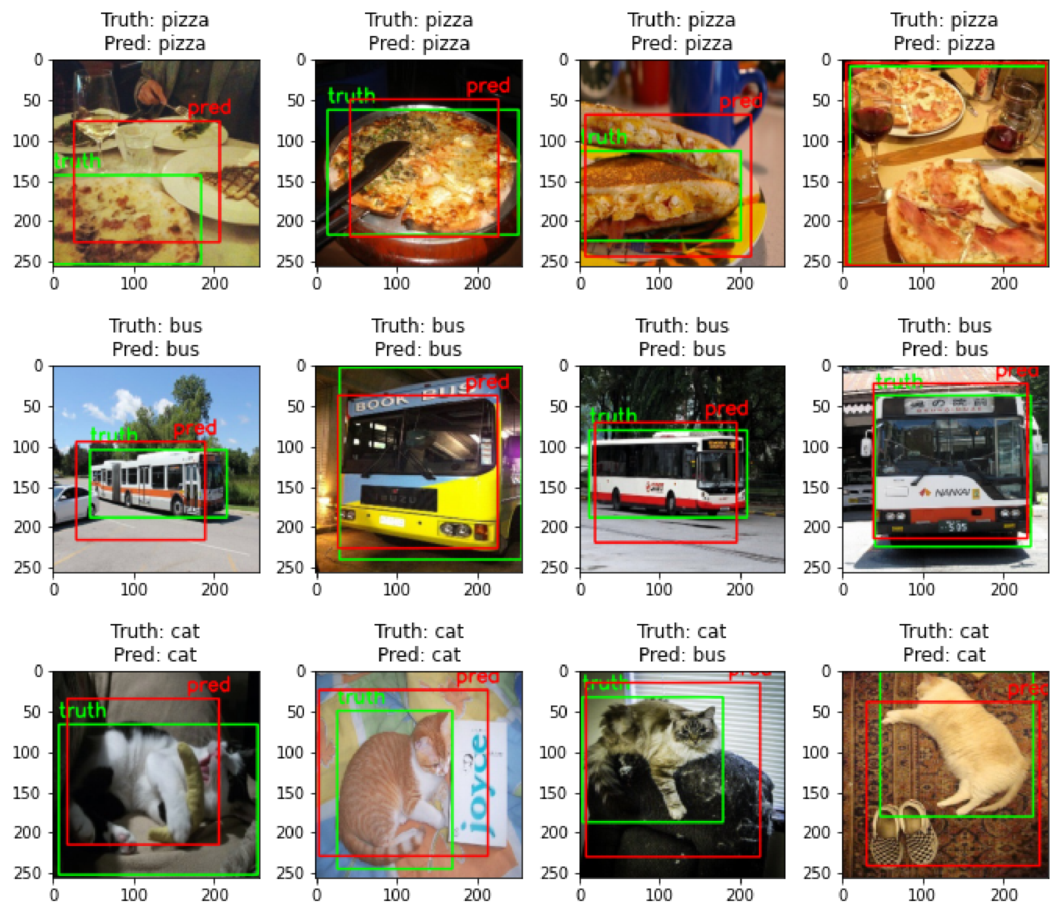

Figure 4: Comparison of Ground Truth and Predicted BBox for MSE Network

3.2 Complete IoU Regression Network

The second network created and trained was one using complete IoU (CIOU) for the bounding box regression task. This network also had a total of **144 layers** as determined by the `net.parameters()` logic we were given in the assignment. The loss of both the regression and classification tasks can be found in figure 5 where 20 epochs with a learning rate of $1e-4$ was used. The confusion matrix can be found in figure 6. The average classification accuracy for all three classes is **90.26%** and the mean IOU is **0.645**. An example of the ground truth and predicted bounding boxes for each image can be found in figure 7.

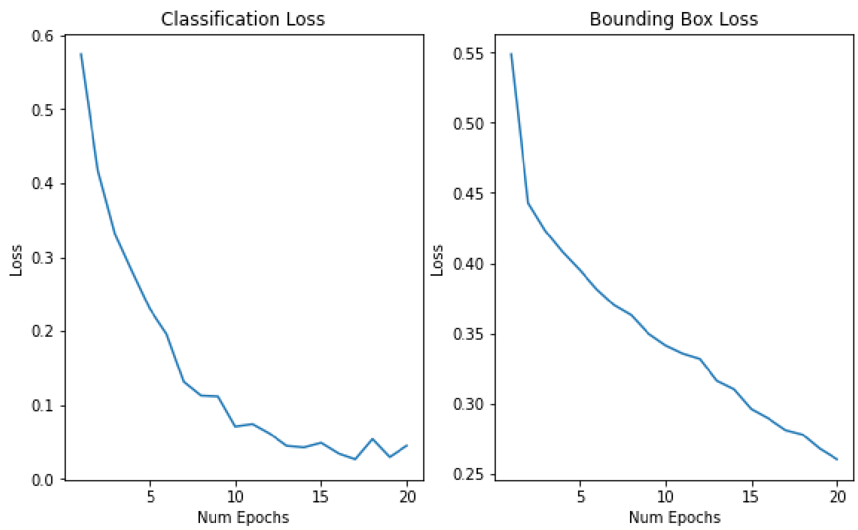

Figure 5: Average Epoch Loss For CIOU Network

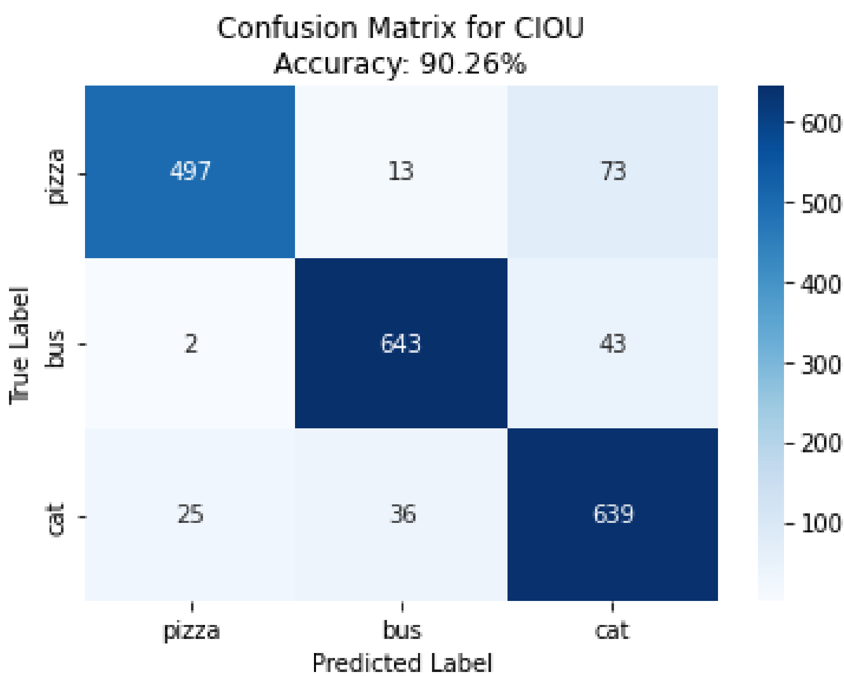

Figure 6: Confusion Matrix for CIOU Network

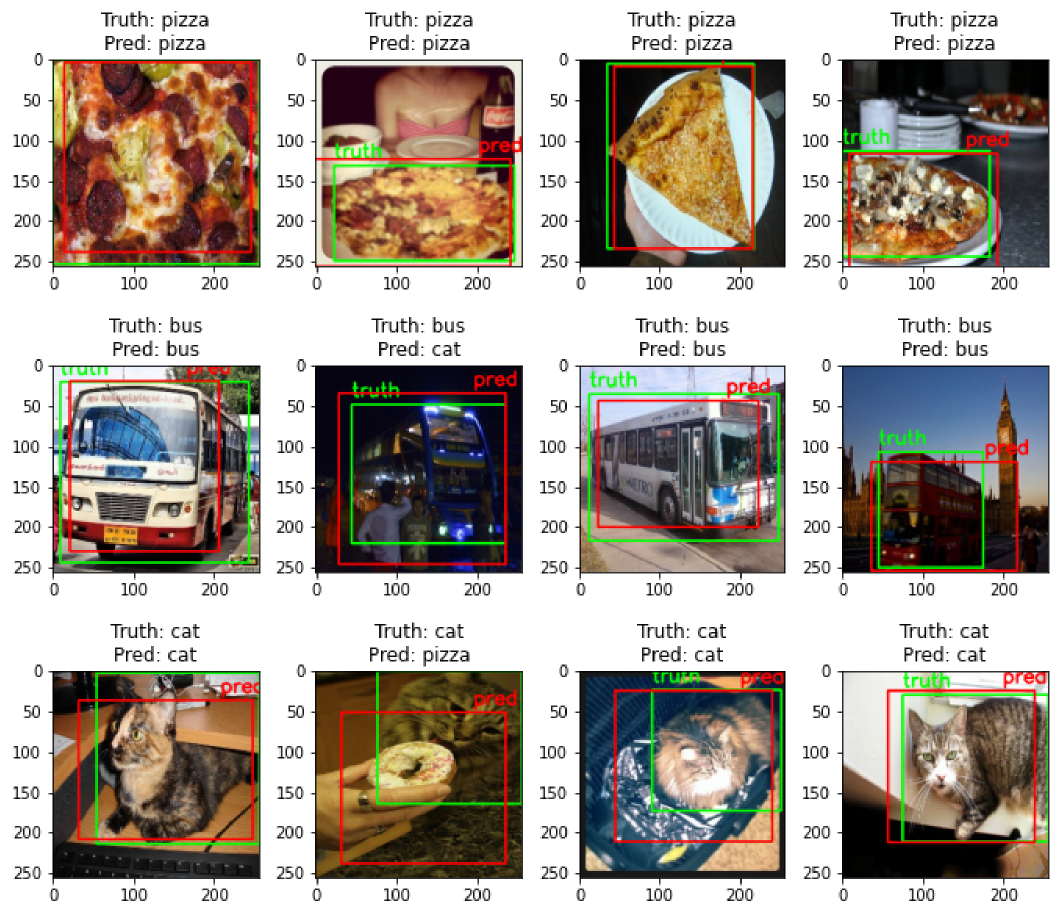

Figure 7: Comparison of Ground Truth and Predicted BBox for CIOU Network

4 Concluding Discussion

In all, the classifier seems to work relatively well on the given dataset. All of the losses decrease during training and appear to approach some form of a minimum which is a good sign that the network is learning. The classification of the classes performs at a pretty high rate only missing a few of the weirder images. Looking at some of the images, it is obvious why the classifier performs poorly since it only has a portion of the object in view or there are multiple objects in the image. The multiple objects seemed to mess up the bounding box the most as it would identify a box covering most of the multiple objects image instead of just the dominant one. Despite this, it seems to draw the bounding boxes well for both loss functions. The CIOU loss function performs decently better than the MSE loss which makes sense since the MSE loss isn't specific to this task and thus has a very very small loss. There is certainly lots of room for improvement. Given more time, hyper parameter tuning could be performed in order to find the optimal learning rate, momentum, batch size, etc. I did a little bit of this, but did not have the time or resources to get the most out of it. Furthermore, changing the dataset to include images with only a single annotation regardless of size would likely help the bounding box success. Finally, the network layers could certainly be designed better should I have more time and knowledge. Someone with more experience could better determine which types of hidden layers to use and what the size of each should be.

5 Source Code

The source code was broken across two different files: `custom_dataset.py` and `hw5_network.py`.

5.1 `custom_dataset.py`

```python
'''
The code in this file is used to generate the custom COCO dataset
'''
from pycocotools.coco import COCO
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
import cv2
import json
import skimage
import json
from skimage import data, io, filters

def get_images_with_dominant_obj(coco, catIds):
  """Find all the images that have a dominant object in them
  """
  return_images = {}

  # get the unique images for the categories
  uniqueImgIds = set()
  for catId in catIds:
    imgIds = coco.getImgIds(catIds=catId)
    uniqueImgIds |= set(imgIds)
```

```python

  # loop through the categories
  for i, img in enumerate(coco.loadImgs(list(uniqueImgIds))):
    annIds = coco.getAnnIds(imgIds=img['id'], catIds=catIds, iscrowd=False)
    anns = coco.loadAnns(annIds)

    curr_ann = []
    # find the images with one dominant object
    for ann in anns:
      if ann['category_id'] in catIds and ann['area'] > 40000:
        curr_ann.append(ann)
    # make sure there is only one dominant object
    if len(curr_ann) == 1:
      assert img['id'] not in return_images, img['id']
      return_images[img['id']] = {'coco_url': img['coco_url'], 'annotations': curr_ann[0]}

  return return_images

def save_images(images, folder_name, coco_labels_inverse):
  """Save images into custom dataset folder
  """
  labels = {}
  # loop through all the images
  for i, (k, img) in enumerate(images.items()):
    if i % 100 == 0:
      print(i)

    # read image from coco API and convert
    I = io.imread(img['coco_url'])
    if len(I.shape) == 2:
      I = skimage.color.gray2rgb(I)
    image = np.uint8(I)
    h, w, c = image.shape

    # resize image and bounding boxes
    xFactor = 256 / w
    yFactor = 256 / h
    image = cv2.resize(image, (256, 256))
    curr_bbox = img['annotations']['bbox']
    new_bbox = [curr_bbox[0]*xFactor, curr_bbox[1]*yFactor,
                curr_bbox[2]*xFactor, curr_bbox[3]*yFactor]
```

```
    label = coco_labels_inverse[img['annotations']['category_id']]

    # save image and label for image
    image_name = f'{i}_{label}.jpg'
    assert image_name not in labels
    labels[image_name] = {'label': label, 'bbox': new_bbox}
    image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    cv2.imwrite(f'custom_dataset/{folder_name}/{i}_{label}.jpg', image)

  return labels

def display_class_images(imageBoxes, class_num, class_list):
  """Display class_num number of images for a given class
  """
  imgs = []
  for k, v in imageBoxes.items():
    if len(imgs) == 4:
      break
    if v['label'] == class_num:
      image = np.uint8(Image.open(f'custom_dataset/train/{k}'))
      [x, y, w, h] = v['bbox']
      # draw bounding boxes
      image = cv2.rectangle(image, (int(x), int(y)), (int(x + w), int(y + h)), (36,255,12), 2)
      image = cv2.putText(image, class_list[v['label']], (int(x), int(y - 10)),
                          cv2.FONT_HERSHEY_SIMPLEX, 0.8 , (36,255,12), 2)
      imgs.append(image)

  fig, ax = plt.subplots(2,2)
  ax[0,0].imshow(imgs[0])
  ax[0,1].imshow(imgs[1])
  ax[1,0].imshow(imgs[2])
  ax[1,1].imshow(imgs[3])
  fig.suptitle(f"4 images for {class_list[class_num]} class", fontsize=16)

if __name__ == '__main__':
  class_list = ['pizza', 'bus', 'cat']

  cocoTrain = COCO('annotations/instances_train2014.json')
  cocoVal = COCO('annotations/instances_val2014.json')

  catIdsTrain = cocoTrain.getCatIds(catNms=class_list)
```

```python
catIdsVal = cocoVal.getCatIds(catNms=class_list)

# create train and validation sets
trainImages = get_images_with_dominant_obj(cocoTrain, catIdsTrain)
valImages = get_images_with_dominant_obj(cocoVal, catIdsVal)

# pulled from the homework assignment example code
categories = cocoTrain.loadCats(catIdsTrain)
coco_labels_inverse_train = {}
for idx, in_class in enumerate(class_list):
  for c in categories:
    if c['name'] == in_class:
      coco_labels_inverse_train[c['id']] = id

# pulled from the homework assignment example code
categories = cocoVal.loadCats(catIdsVal)
coco_labels_inverse_val = {}
for idx, in_class in enumerate(class_list):
  for c in categories:
    if c['name'] == in_class:
      coco_labels_inverse_val[c['id']] = id

# save images and get annotations
trainImageBoxes = save_images(trainImages, 'train', coco_labels_inverse_train)
valImageBoxes = save_images(valImages, 'val', coco_labels_inverse_val)

# save annotations
with open("custom_dataset/train_labels.json", "w") as f:
  json.dump(trainImageBoxes, f)
with open("custom_dataset/val_labels.json", "w") as f:
  json.dump(valImageBoxes, f)

print(len(trainImages))
print(len(valImages))

display_class_images(trainImageBoxes, 0, class_list)
display_class_images(trainImageBoxes, 1, class_list)
display_class_images(trainImageBoxes, 2, class_list)
```

5.2 hw5_network.py

```
'''
The code in this file lays out the model architecture and the training routine
'''
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
import os
import glob
from sklearn.metrics import confusion_matrix, accuracy_score
import seaborn as sns
import torchvision
import torchvision.transforms as tvt
import torch
import torch.nn as nn
import torch.nn.functional as F
import cv2
import json
from tqdm.notebook import trange, tqdm

class MyDataset(torch.utils.data.Dataset):
  """Class to load in custom COCO dataset
  """
  def __init__(self, root, label_file):
    super().__init__()
    self.root = root
    self.label_dict = {0:'pizza', 1:'bus', 2: 'cat'}

    with open(label_file, 'r') as f:
      self.labels = json.load(f)

    # referenced https://stackoverflow.com/questions/26392336/
    # importing-images-from-a-directory-python-to-list-or-dictionary
    # to determine how to find all image file names
    self.image_files = glob.glob(os.path.join(self.root, '*.jpg'))

  def __len__(self):
    return len(self.image_files)

  def __getitem__(self, index):
```

```python
    # open image and get label
    pil_img = Image.open(self.image_files[index])
    curr_label_info = self.labels[os.path.basename(self.image_files[index])]
    label = torch.tensor(curr_label_info['label'])

    # get bbox info
    bbox = curr_label_info['bbox']
    [x, y, w, h] = bbox
    bbox = torch.tensor([x,y,x+w,y+h])

    # perform RGB and tensor transforms
    if pil_img.mode != "RGB":
      pil_img = pil_img.convert(mode="RGB")
    transforms = tvt.Compose([
        tvt.ToTensor()
    ])
    transformed_img = transforms(pil_img)
    assert transformed_img.shape == torch.Size([3,256,256])

    # normalize bbox to be in range [0,1]
    bbox = torch.div(bbox, transformed_img.shape[1])
    assert torch.max(bbox) <= 1

    return transformed_img, label.squeeze(), bbox.squeeze()

class SkipBlock(nn.Module):
  """Skip Connection Layer with option to downsample tensor
  """
  def __init__(self, in_ch, out_ch, downsample=False):
    super(SkipBlock, self).__init__()
    self.downsample = downsample
    if self.downsample:
      self.downsampler = nn.Conv2d(in_ch, out_ch, (3,3), stride=2)

    self.in_ch = in_ch
    self.out_ch = out_ch
    self.conv1 = nn.Conv2d(in_ch, out_ch, (3,3), stride=1, padding=1)
    self.bn1 = nn.BatchNorm2d(out_ch)

    # different logic for when in_ch == out_ch
    if self.in_ch == self.out_ch:
```

```
      self.conv2 = nn.Conv2d(in_ch, out_ch, (3,3), stride=1, padding=1)
      self.bn2 = nn.BatchNorm2d(out_ch)
      # downsample layer sizes are different too
      if self.downsample:
        self.downsampler_out = nn.Conv2d(in_ch, out_ch, (3,3), stride=2)
        self.downsampler_identity = nn.Conv2d(in_ch, out_ch, (3,3), stride=2)
    else:
      self.conv2 = nn.Conv2d(out_ch, out_ch, (3,3), stride=1, padding=1)
      self.bn2 = nn.BatchNorm2d(out_ch)
      # downsample layer sizes are different too
      if self.downsample:
        self.downsampler_out = nn.Conv2d(out_ch, out_ch, (3,3), stride=2)
        self.downsampler_identity = nn.Conv2d(in_ch, out_ch, (3,3), stride=2)

    self.relu = nn.ReLU(True)
    return

  def forward(self, x):
    # store input for skip connection
    identity = x

    # run input through two conv and bn layers
    out = self.conv1(x)
    out = self.bn1(out)
    out = self.relu(out)
    out = self.conv2(out)
    out = self.bn2(out)

    # downsample output and identity if necessary
    if self.downsample:
      out = self.downsampler_out(out)
      identity = self.downsampler_identity(identity)

    # combine input with output
    out += identity
    out = self.relu(out)

    return out

class FlattenForLinear(nn.Module):
  """Custom module to flatten tensor for input into fc layers
```

```
  """
  def __init__(self):
    super(FlattenForLinear, self).__init__()
  def forward(self, x):
    return x.view(x.shape[0], -1)

class HW5Net(nn.Module):
  """Resnet-based encoder that consists of a few downsampling + several Resnet
  blocks as the backbone and two prediction heads.
  NOTE: THIS FUNCTION IS DERIVED FROM THE ONE PROVIDED IN THE HOMEWORK.
  MODIFICATIONS WERE MADE IN THE NECESSARY PARTS
  """
  def __init__(self, input_nc, output_nc, ngf=8, n_blocks=4):
    assert (n_blocks >= 0)
    super(HW5Net, 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
    ## most of code below is all my own (code above is from assignment) ##
    mult = 2 ** n_downsampling
    for i in range(n_blocks):
      model += [SkipBlock(ngf * mult, ngf * mult, downsample=False)]
    self.model = nn.Sequential(*model)

    #### The classification head ####
    n_downsampling = 3
    skip_out = ngf * mult
    class_head = []
```

```python

    # use skip blocks to downsample tensor before linear layers
    for i in range(n_downsampling):
      mult = 2 ** i
      class_head += [SkipBlock(skip_out * mult, skip_out * mult * 2,
                               downsample=True)]
    # linear layers to get output classes
    class_head += [FlattenForLinear(),
                   nn.Linear(skip_out * mult * 2, int(skip_out * mult / 16)),
                   nn.ReLU(True),
                   nn.Linear(int(skip_out * mult / 16), output_nc)]

    self.class_head = nn.Sequential(*class_head)

    #### The bounding box regression head ####
    bbox_head = []
    n_downsampling = 3

    # use skip blocks to downsample tensor before linear layers
    for i in range(n_downsampling):
      mult = 2 ** i
      bbox_head += [SkipBlock(skip_out * mult, skip_out * mult * 2,
                              downsample=True)]
    # linear layers to get output classes
    bbox_head += [FlattenForLinear(),
                  nn.Linear(skip_out * mult * 2, int(skip_out * mult / 16)),
                  nn.ReLU(True),
                  nn.Linear(int(skip_out * mult / 16), 4)]
    self.bbox_head = nn.Sequential(*bbox_head)

  def forward(self, input):
    ft = self.model(input)
    cls = self.class_head(ft.clone())
    bbox = self.bbox_head(ft.clone())
    return cls, bbox

def train(net, data_loader, criterion_bbox_type, device, num_epochs=20):
  all_epoch_loss_cls = []
  all_epoch_loss_bbox = []

  net = net.to(device)
```

```
net.train()
criterion_cls = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(
    net.parameters(), lr=1e-4, betas=(0.9, 0.99))

# run training for all epochs (tqdm displays progress bar)
for epoch in tqdm(range(num_epochs), desc=" epochs", position=0):
  # keep track of different running losses
  running_loss_cls = 0.0
  running_loss_bbox = 0.0
  epoch_loss_cls = 0.0
  epoch_loss_bbox = 0.0

  # loop through data in train dataset
  pbar = tqdm(data_loader, desc=" data loader", position=1)
  for i, data in enumerate(pbar):
    inputs, labels, bboxes = data
    inputs = inputs.to(device)
    labels = labels.to(device)
    bboxes = bboxes.to(device)

    optimizer.zero_grad()
    outputs_cls, outputs_bbox = net(inputs)

    # compute loss for classification
    loss_cls = criterion_cls(outputs_cls, labels)
    loss_cls.backward(retain_graph=True)

    # choose correct loss function for regression
    if criterion_bbox_type == 'MSE':
      criterion_bbox = torch.nn.MSELoss()
      loss_bbox = criterion_bbox(outputs_bbox, bboxes)
    elif criterion_bbox_type == 'CIOU':
      loss_bbox = torchvision.ops.complete_box_iou_loss(outputs_bbox, bboxes,
                                                        reduction='mean')
    else:
      assert False, 'Invalid Criterion Type for bbox'

    loss_bbox.backward()
    optimizer.step()

```

```
      # update running losses
      running_loss_cls += loss_cls.item()
      running_loss_bbox += loss_bbox.item()
      epoch_loss_cls += loss_cls.item()
      epoch_loss_bbox += loss_bbox.item()

      # display progress as it trains
      pbar.set_description(f'loss_cls: {epoch_loss_cls/(i+1):.3f} \
                           loss_bbox: {epoch_loss_bbox/(i+1):.3f}')
      if (i+1) % 100 == 0:
        print("[epoch: %d, batch: %5d] loss_cls: %.3f loss_bbox: %.3f" \
              % (epoch + 1, i + 1, running_loss_cls / 100, running_loss_bbox / 100))
        running_loss_cls = 0.0
        running_loss_bbox = 0.0
    all_epoch_loss_cls.append(epoch_loss_cls / (i+1))
    all_epoch_loss_bbox.append(epoch_loss_bbox / (i+1))

  return all_epoch_loss_cls, all_epoch_loss_bbox

def graph_loss(epoch_loss_cls, epoch_loss_bbox):
  """This function graphs the loss for the classification and regression
  """
  num_epochs = len(epoch_loss_cls)
  fig, ax = plt.subplots(1,2)
  fig.set_size_inches(9.5, 5.5)

  # plot classification loss
  ax[0].plot(list(range(1,num_epochs+1)), epoch_loss_cls)
  ax[0].set_xlabel('Num Epochs')
  ax[0].set_ylabel('Loss')
  ax[0].set_title('Classification Loss')

  # plot regression loss
  ax[1].plot(list(range(1,num_epochs+1)), epoch_loss_bbox)
  ax[1].set_xlabel('Num Epochs')
  ax[1].set_ylabel('Loss')
  ax[1].set_title('Bounding Box Loss')
  plt.show()

class EvaluateModel():
  """Class to perform various evaluations on the validation set
```

```
  """
  def __init__(self, net, name, data_loader, epoch_loss_cls, epoch_loss_bbox, num_comp=4):
    self.net = net
    self.name = name
    self.data_loader = data_loader
    self.labels = ['pizza','bus','cat']
    self.epoch_loss_cls = epoch_loss_cls
    self.epoch_loss_bbox = epoch_loss_bbox
    self.num_epochs = len(self.epoch_loss_cls)
    self.num_comp = num_comp

  def perform_inference(self):
    self.net.eval()

    self.y_true = []
    self.y_pred = []
    self.bbox_true = []
    self.bbox_pred = []

    self.compare_dict = {'pizza': [], 'bus': [], 'cat': []}

    for i, data in enumerate(self.data_loader):
      inputs, labels, bboxes = data
      inputs = inputs.to(device)

      # put data through model
      outputs_cls, outputs_bbox = self.net(inputs)
      outputs_cls = torch.argmax(outputs_cls, dim=1)

      # move outputs to numpy on cpu
      labels = labels.numpy()
      bboxes = bboxes.numpy()
      outputs_cls = outputs_cls.detach().cpu().numpy()
      outputs_bbox = outputs_bbox.detach().cpu().numpy()
      inputs = inputs.detach().cpu().numpy()

      # track image, label, bbox groupings for later display
      for label, image, true_bbox, pred_bbox, pred_label in \
                  zip(labels, inputs, bboxes, outputs_bbox, outputs_cls):
        curr_label = self.labels[label]
        # only save the number desired by the user
```

```
      if len(self.compare_dict[curr_label]) < self.num_comp:
        self.compare_dict[curr_label].append({'img': image.transpose(1,2,0).copy(),
                                              'pred_label': self.labels[pred_label],
                                              'true_bbox': true_bbox*image.shape[1],
                                              'pred_bbox': pred_bbox*image.shape[1]})

    # track data for confusion matrix
    self.y_true.extend(labels)
    self.y_pred.extend(outputs_cls)

    # track data for IOU metrics
    self.bbox_true.extend(bboxes)
    self.bbox_pred.extend(outputs_bbox)

  def confusion_matrix(self):
    # plot confusion matrix and accuracy score
    conf_mat = confusion_matrix(self.y_true, self.y_pred)
    acc_score = accuracy_score(self.y_true, self.y_pred)

    sns.heatmap(conf_mat, cmap='Blues', annot=True, fmt='g',
                xticklabels=self.labels, yticklabels=self.labels)
    plt.xlabel('Predicted Label')
    plt.ylabel('True Label')
    plt.title(f'Confusion Matrix for {self.name}\nAccuracy: {acc_score*100:.2f}%')
    plt.show()

  def IOU_metrics(self):
    # calculate mean iou
    running_iou = 0
    for true, pred in zip(self.bbox_true, self.bbox_pred):
      true = torch.tensor(true)
      pred = torch.tensor(pred)
      running_iou += torchvision.ops.box_iou(true.unsqueeze(0), pred.unsqueeze(0))

    mean_iou = running_iou / len(self.bbox_true)
    print(f'Mean IOU: {float(mean_iou.squeeze())}')

  def show_bbox_inference(self):
    # loop through saved images from validation loop
    for label, images in self.compare_dict.items():
      # fig, ax = plt.subplots(int(self.num_comp / 2),int(self.num_comp / 2))
```

```
      fig, ax = plt.subplots(1,self.num_comp)
      fig.set_size_inches(9.5, 5.5)
      ax = ax.flatten()

      # loop through image for a given class
      for i, img_dict in enumerate(images):
        img = img_dict['img']

        # draw ground truth bbox
        [x1_gt,y1_gt,x2_gt,y2_gt] = img_dict['true_bbox']
        img = cv2.rectangle(img, (int(x1_gt), int(y1_gt)),
                            (int(x2_gt), int(y2_gt)), (0,1,0), 2)
        img = cv2.putText(img, f'truth', (int(x1_gt), int(y1_gt - 10)),
                          cv2.FONT_HERSHEY_SIMPLEX, 0.8 , (0,1,0), 2)

        # draw prediction bbox
        [x1_pred,y1_pred,x2_pred,y2_pred] = img_dict['pred_bbox']
        img = cv2.rectangle(img, (int(x1_pred), int(y1_pred)),
                            (int(x2_pred), int(y2_pred)), (1,0,0), 2)
        img = cv2.putText(img, f'pred',
                                        (int(x2_pred-40), int(y1_pred - 10)),
                           cv2.FONT_HERSHEY_SIMPLEX, 0.8 , (1,0,0), 2)
        img[img > 1] = 1
        img[img < 0] = 1
        ax[i].imshow(img)
        ax[i].set_title(f'Truth: {label}\nPred: {img_dict["pred_label"]}')

      plt.tight_layout()
      plt.show()

if __name__ == '__main__':
  device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
  print(device)

  #### MSE NETWORK ####
  train_dataset = MyDataset('custom_dataset/train', 'custom_dataset/train_labels.json')
  val_dataset = MyDataset('custom_dataset/val', 'custom_dataset/val_labels.json')

  train_batch_size = 16
  train_data_loader = torch.utils.data.DataLoader(train_dataset, batch_size=train_batch_size, shuffle=Tr

```

```
net_mse = HW5Net(3, 3)
num_layers = len(list(net_mse.parameters()))
print(f'Num layers: {num_layers}')

# train network
num_epochs = 20
all_epoch_loss_cls, all_epoch_loss_bbox = train(net_mse, train_data_loader,
                                                'MSE', device, num_epochs=num_epochs)

# perform validation
val_batch_size = 16
val_data_loader = torch.utils.data.DataLoader(val_dataset,
                                              batch_size=val_batch_size,
                                              shuffle=True)

eval = EvaluateModel(net_mse, 'MSE', val_data_loader,
                     all_epoch_loss_cls, all_epoch_loss_bbox)
graph_loss(all_epoch_loss_cls, all_epoch_loss_bbox)
eval.perform_inference()
eval.confusion_matrix()
eval.show_bbox_inference()
eval.IOU_metrics()

#### CIOU NETWORK ####
net_ciou = HW5Net(3, 3)
num_layers = len(list(net_ciou.parameters()))
print(f'Num layers: {num_layers}')

# train network
num_epochs = 20
all_epoch_loss_cls, all_epoch_loss_bbox = train(net_ciou, train_data_loader,
                                                'CIOU', device, num_epochs=num_epochs)

# perform validation
eval = EvaluateModel(net_ciou, 'CIOU', val_data_loader,
                     all_epoch_loss_cls, all_epoch_loss_bbox)
graph_loss(all_epoch_loss_cls, all_epoch_loss_bbox)
eval.perform_inference()
eval.confusion_matrix()
eval.show_bbox_inference()
eval.IOU_metrics()
```