SPRING 2023 ECE 60146 – Homework 6

Sahithi Kodali – 34789866
kodali1@purdue.edu

3.1 Creating own Multi-Instance Object Localization dataset:

A dataset with a total of 6282 training images and 3480 validation images has been created which includes at least one object from the categories ['bus', 'cat', 'pizza']. The bounding box criteria of 4096 pixels area for a foreground object and resizing to 256x256 size images has been applied to attain this custom dataset.

A sample of three images from each of the class with its scaled bounding box have been plotted as shown in Figure 1.

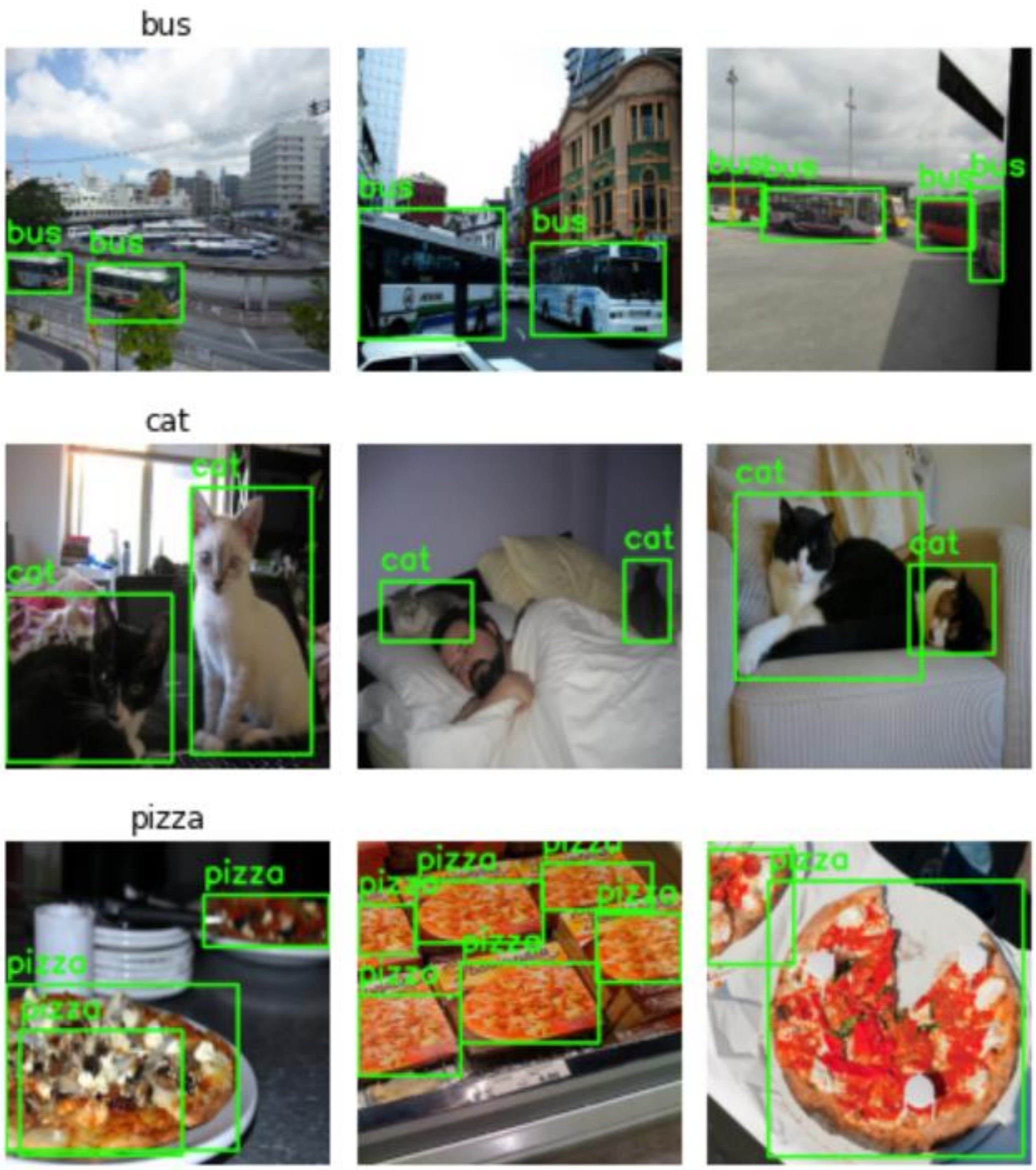


Figure 1: Plot of three images from each class with a boundary box for each instance

4.2 Building Deep Neural Network

To implement a deep CNN for multi-instance object detection and location, a Skip-connection block/ResBlock is used inside the network with the total learnable layers in the model being 64. This model is similar to my previous HW5 network and Prof.Kak network, with change in a few parameters and layers. A snippet of the YOLO network using ResBlock can be seen in Figure 2.

The key parameters to be observed in the YOLO network that are specific to this implementation are in the self.fc_seqn layer. Here the parameters for the first linear layer are 128*32*32 which represents the (output_channels*image_height*image_width). The initial image size of 256*256 is changed to (32*32) after a series of convolution, maxpooling and downsampling as shown in the forward function of network. Further, in the last linear layer of sel.fc_seqn, we can observe the output size to be (6*6*5*9) which is the shape of the yolo tensor that needs to be obtained for each image. Here, the yolo tensors shape represents (grid_width * grid_height * num_anchor_boxes * yolovector_size) and the chosen grid size is (6*6), the number of anchor boxes are 5 and the yolo vector shape is 9 (where the first element represents is object presence in anchor box, second and third the displacement between the boundary box center and the cell center, fourth and fifth are the weight and height of boundary box w.r.t cell dimension and the last but one three elements are the one hot encoding form of the label, and the last element is to drop all the instance probability mass that are beyond maximum).

```
#ResNet Block (Inspired from Prof Kak's RPG SkipBlock)

class ResBlock(nn.Module):
  def __init__(self, in_ch, out_ch, downsample=False, skip_connections=True):
    super(ResBlock, self).__init__()
    self.downsample = downsample
    self.skip_connections = skip_connections
    self.in_ch = in_ch
    self.out_ch = out_ch
    #convolution layer and batch normalization
    self.conv1 = nn.Conv2d(in_ch, out_ch, 3, stride=1, padding=1)
    norm_layer1 = nn.BatchNorm2d
    self.bn1 = norm_layer1(out_ch)
    #downsampler - convolution layer
    if downsample:
        self.downsampler = nn.Conv2d(in_ch, out_ch, 1, stride=2)

  def forward(self, x):
    #residual input
    identity = x.clone()
    #convolution and batch normalization, relu
    out = self.conv1(x)
    out = self.bn1(out)
    out = torch.nn.functional.relu6(out)
```

```python
        #check for input and output channels
        if self.in_ch == self.out_ch:
            out = torch.nn.functional.relu6(out)
        #check downsampling
        if self.downsample:
            out = self.downsampler(out)
            identity = self.downsampler(identity)
        #check for skip connections
        if self.skip_connections:
            if self.in_ch == self.out_ch:
                out = out + identity
            else:
                out[:,:self.in_ch,:,:] = out[:,:self.in_ch,:,:] + identity
                out[:,self.in_ch:,:,:] = out[:,self.in_ch:,:,:] + identity
        return out
```

```python
#Yolo Network (Inspired from Prof Kak's RPG)

class YOLO(nn.Module):
  def __init__(self, depth = 8, skip_connections = True):
    super(YOLO, self).__init__()

    if depth not in [8,10,12,14,16]:
        sys.exit("This network has only been tested for 'depth' values 8, 10, 12, 14, and 16")

    self.depth = depth // 2
    self.conv1 = nn.Conv2d(3, 64, 3, padding=1)
    self.pool = nn.MaxPool2d(2, 2)
    self.bn1  = nn.BatchNorm2d(64)
    self.bn2  = nn.BatchNorm2d(128)

    #layers for downsampling and changing to 128 channels
    self.skip64_arr = nn.ModuleList()
    for i in range(self.depth):
      self.skip64_arr.append(ResBlock(64, 64, skip_connections = skip_connections))
    self.skip64ds = ResBlock(64, 64, downsample=True, skip_connections=skip_connections)
    self.skip64to128 = ResBlock(64, 128, skip_connections=skip_connections )

    #layers for downsampling
    self.skip128_arr = nn.ModuleList()
    for i in range(self.depth):
      self.skip128_arr.append(ResBlock(128,128, skip_connections=skip_connections))
    self.skip128ds = ResBlock(128,128, downsample=True, skip_connections=skip_connections)
    # self.skip128to256 = ResBlock(128, 256, skip_connections=skip_connections )

    self.fc_seqn = nn.Sequential(
        nn.Linear(128*32*32, 4096),
        nn.ReLU(inplace=True),
        nn.Linear(4096, 2048),
        nn.ReLU(inplace=True),
        nn.Linear(2048, 6*6*5*9) #grid_size*grid_size*num_anchorboxes*yolovector_size
        )
```

```python
def forward(self, x):
  #applying the initialized layers in sequence
  x = self.pool(torch.nn.functional.relu6(self.conv1(x)))

  for i,skip64 in enumerate(self.skip64_arr[:self.depth//4]):
      x = skip64(x)
  x = self.skip64ds(x)

  for i,skip64 in enumerate(self.skip64_arr[self.depth//4:]):
      x = skip64(x)
  x = self.bn1(x)
  x = self.skip64to128(x)

  for i,skip128 in enumerate(self.skip128_arr[:self.depth//4]):
      x = skip128(x)
  # x = self.bn2(x)
  x = self.skip128ds(x)
  # x = self.skip128to256(x)

  x = x.view(-1, 128*32*32)

  x = self.fc_seqn(x)
  return x
```

Figure 2: The ResNet block and the Yolo network block using ResNet

4.3. Training and Evaluating the Trained Network

An own torch.utils.data.Dataset and DataLoader are implemented based on the own dataset requirements. The network in now trained using the training data and evaluated on validation data. The hyperparameters used for training are 10 epochs, 8 batch size and 2 number of workers.

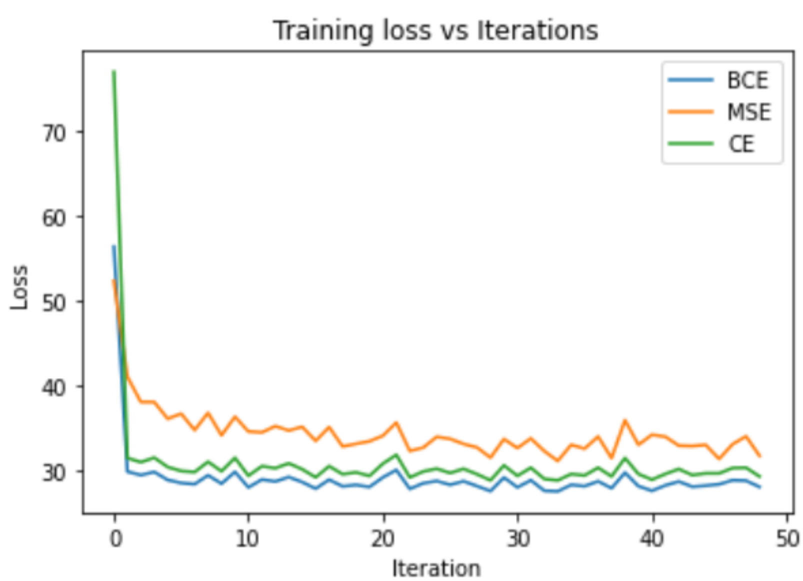
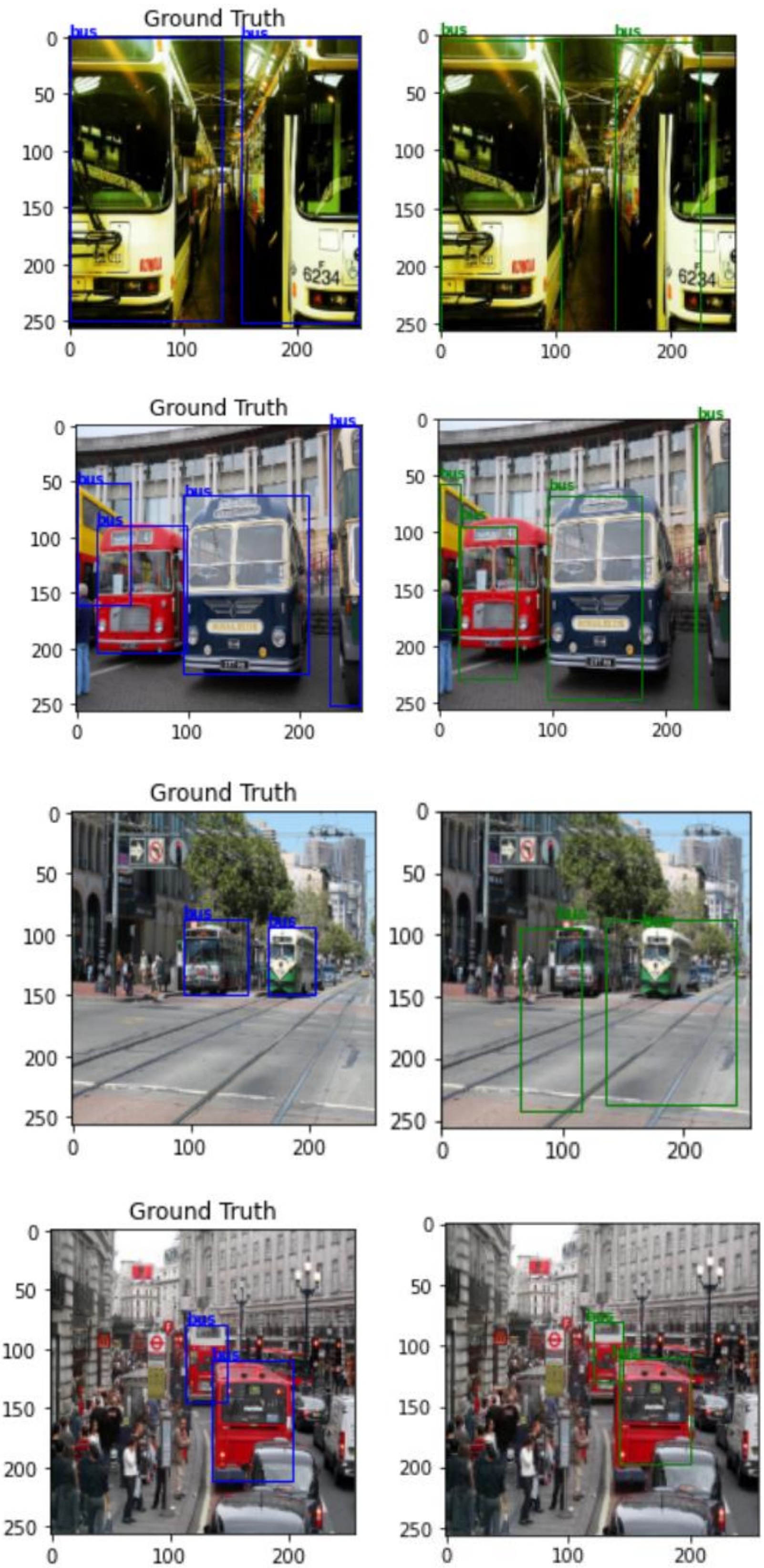


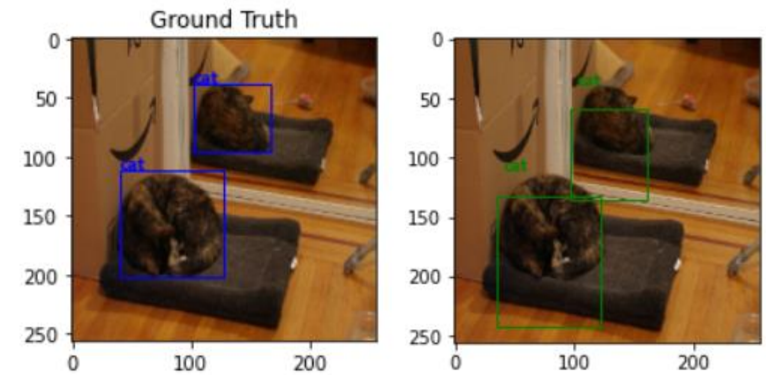
Figure 3: Plot of the three losses vs iterations

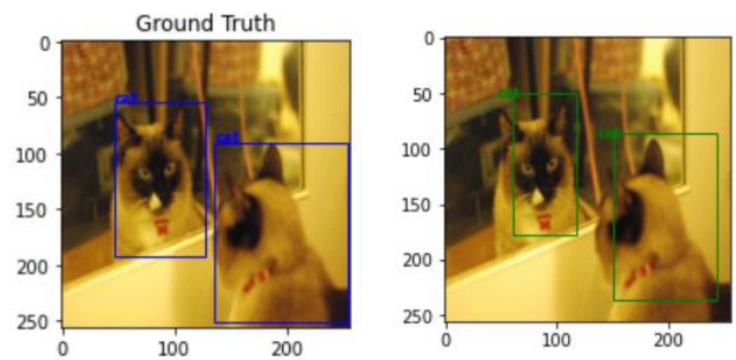
Class – Bus

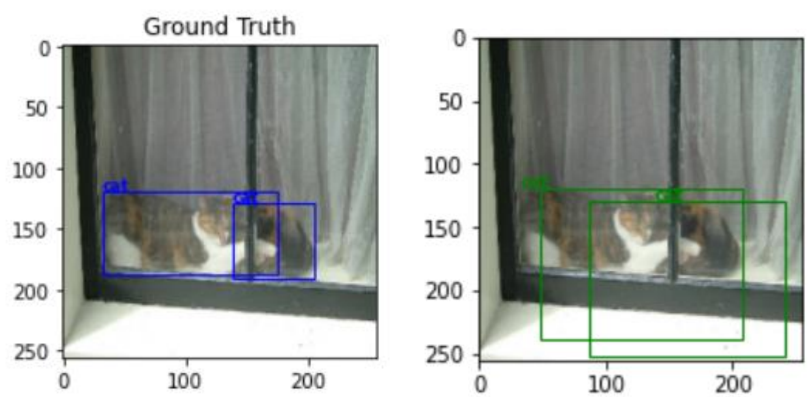

Ground Truth
bus
Ground Truth
bus
Ground Truth
bus
Ground Truth
bus

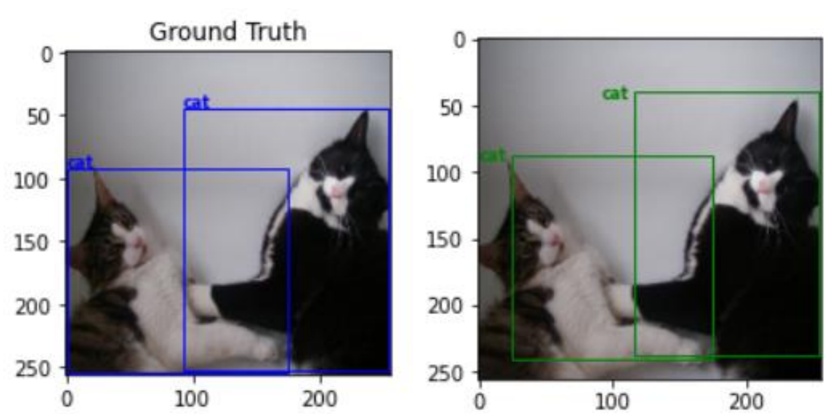

Ground Truth
bus
bus
Ground Truth
bus
bus
Ground Truth
bus
bus
bus
Ground Truth
bus
bus
0
50
100
150
200
250

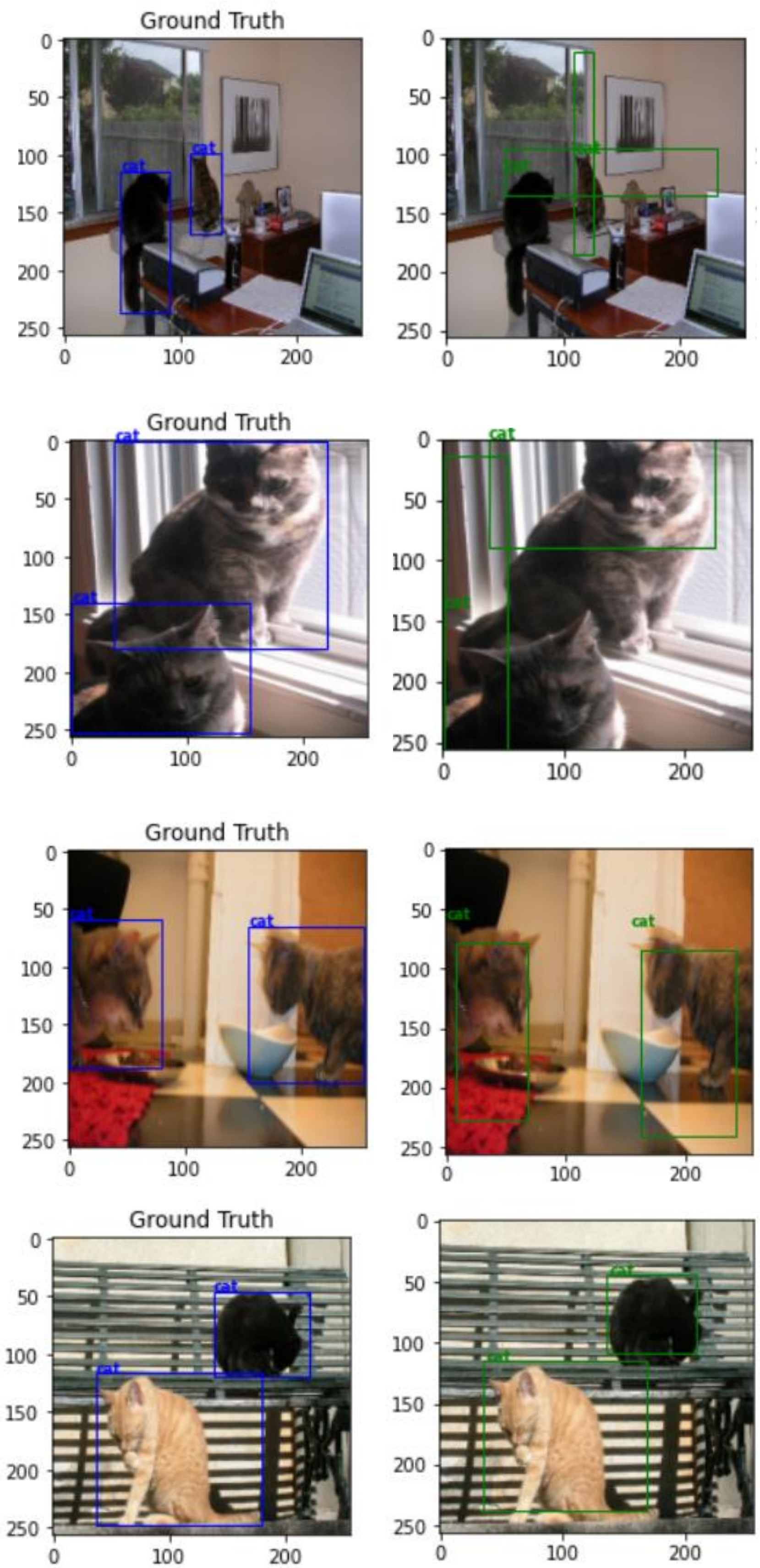
Class – Cats


Ground Truth
cat
cat

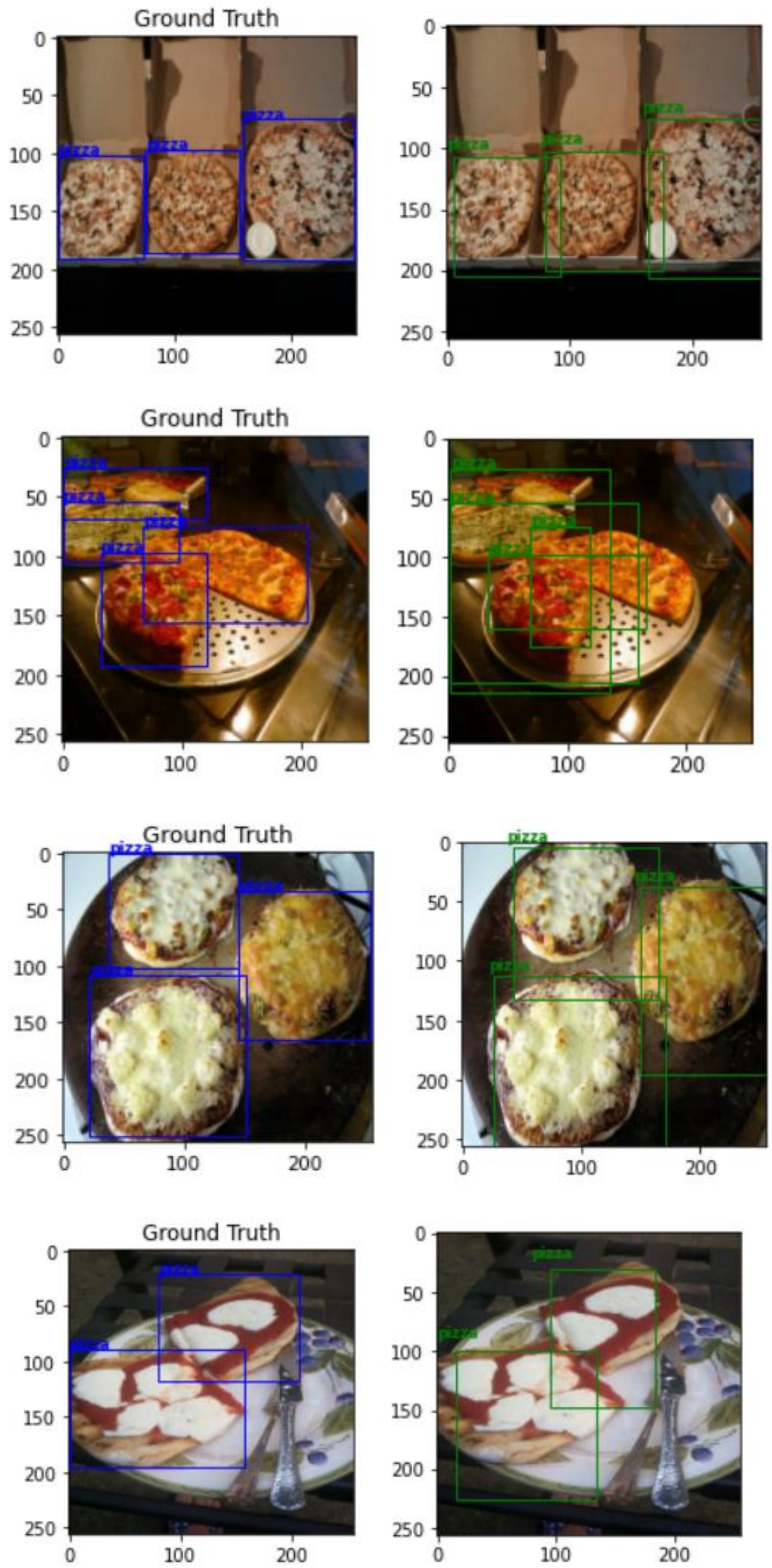

Ground Truth
cat
cat


Ground Truth
cat
cat


Ground Truth
cat
cat
cat
cat


Ground Truth
cat
Ground Truth
cat
Ground Truth
cat
Ground Truth
cat
0
50
100
150
200
250
0
100
200

Class – Pizza



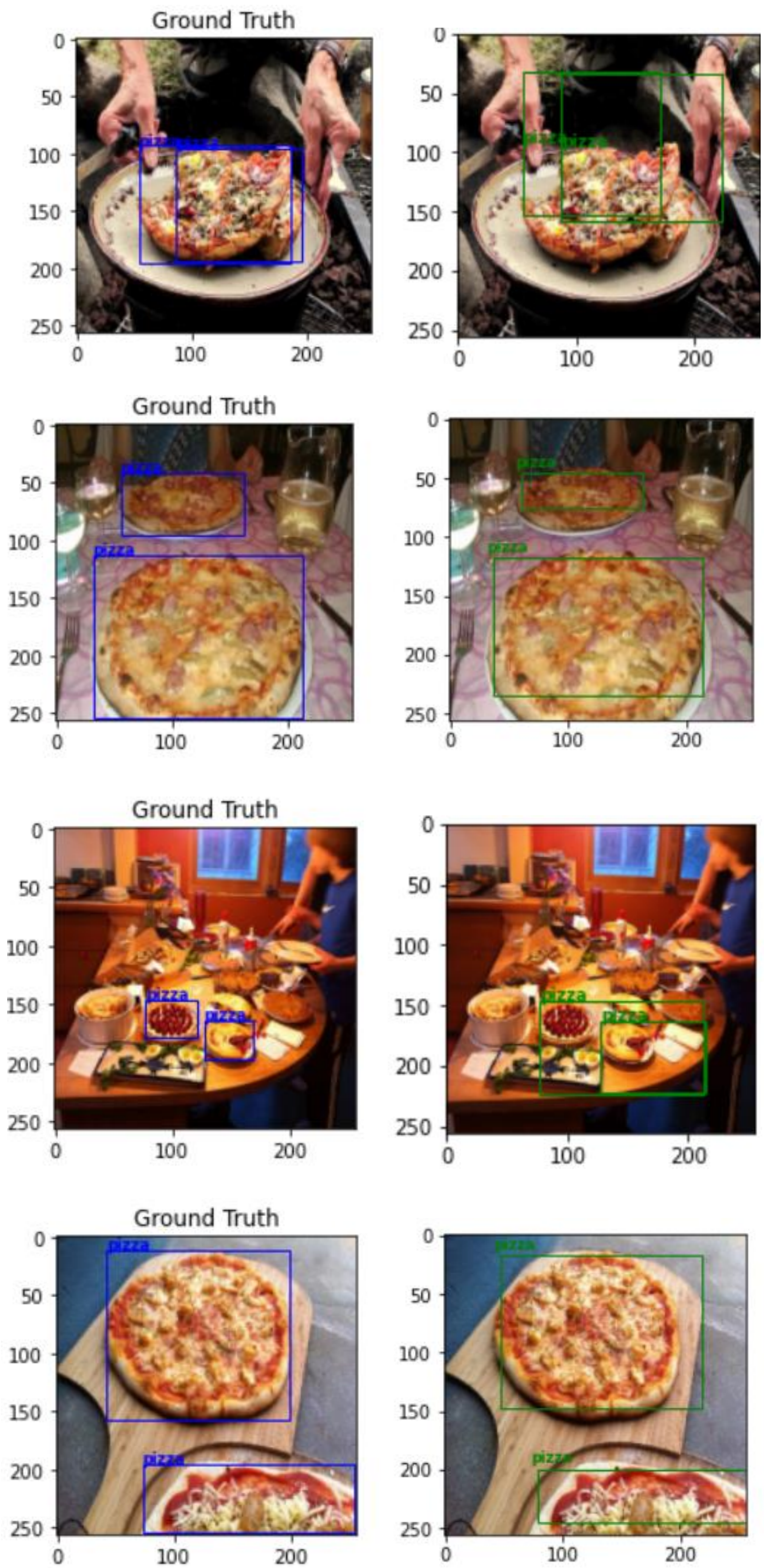


Figure 4: Ground truth (blue) and predicted (green) outputs for 8 images of each class

Details of Implementation of Data loader, Training and Evaluation:

The data loader, training evaluation process followed are not batch processing based but are similar to the method Prof. Kak explained. However, attempts to perform batch processing (i.e., not looping through each image in batch) have been performed but due to the weird losses attained, I have proceeded with the regular batch looping method. The code for batch processing attempted can be seen in the end of source code in this report which involves usage of tensor slicing for processing all images are once.

Data Loader:

The dataloader loads the images and respective annotation file and returns the boundary box, labels tensor for the number of objects instances in each image based on index of the image given. The respective image, num of instances, boundary box tensor and label tensor is returned by the data loader.

Training:

This task of training was more complex for the yolo model. The steps followed for training are:

i. Each image is divided into a 6X6 grid of cells and 5 anchor boxes of the aspect ratio 1/3, 1/5, 1/1, 5/1, 3/1 for each cell are initialized. Each cell will be of size 40X40 since the image is of size 256X256.

ii. This is followed by assigning boundary box to each object instance in image to the anchor box most suitable in the best cell. This is given by a yolo vector for each instance in an image and thus a yolo tensor of yolo vectors (each for an object instance in image) needs to be trained. As explained in the previous 3.2 section, the yolo vector has 9 elements each having its own role for predicting losses.

iii. The first element is the objectiveness which is 1 if object present else 0. This is used for computing Binary cross entropy loss. The second to fifth elements are used to compute the mean squared error loss of the boundary box and the remaining elements are the label representation used to compute cross Entropy loss.

iv. This way three losses are used to train the yolo model. The source code includes more detailed comments for a step by step explanation.

Evaluation:

Finally in the evaluation step, the respective prediction in the form of yolo tensor is made for each image. For the predictions, the cells pertaining to the best anchor box are retained and the boundary box and labels are estimated from these retained cells by following the center displacement and object centers-based method as in training. These predicted boundary boxes and labels are used to plot the boundary box and their respective annotations. Again, these details are clearly commented in the source code.

Performance observations:

From the images shown in Figure 4 we can observe that the predictions are well versed with the ground truth. However, there are some predictions where multiple instances are together, and the predicted boundary box is larger in such case than the desired aspect ratio in ground truth. Also, sometimes due to the noise in the image the image goes completely undetected (such worst cases were not shown in the figures but were observed during evaluation). Finally, the performance can be improved by making the model more complex and checking the issues for cases when the predicted boundary box is larger than the actual ground truth size i.e., more accurate localization is required.

SOURCE CODE:

```
# -*- coding: utf-8 -*-
"""HW6_ECE60146.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1TtwHAVOxaiTLsxhgJHUlDy-y354wB0sc
"""

# Commented out IPython magic to ensure Python compatibility.
#Change directories, unzip RPG and install
# %cd /content/drive/MyDrive/Purdue/HW6

# !tar -xzf RegionProposalGenerator-2.0.8.tar.gz

# Commented out IPython magic to ensure Python compatibility.
# %cd /content/drive/MyDrive/Purdue/HW6/RegionProposalGenerator-2.0.8

!pip install .

# Commented out IPython magic to ensure Python compatibility.
# %cd ExamplesObjectDetection

!tar -xzf datasets_for_RPG.tar.gz

!tar -xzf /content/drive/MyDrive/Purdue/HW6/RegionProposalGenerator-
2.0.8/ExamplesObjectDetection/data/Purdue_Dr_Eval_Multi_Dataset-clutter-10-noise-
20-size-10000-valid.gz
```

```
!tar -xzf /content/drive/MyDrive/Purdue/HW6/RegionProposalGenerator-
2.0.8/ExamplesObjectDetection/data/Purdue_Dr_Eval_Multi_Dataset-clutter-10-noise-
20-size-1000-test.gz

# Commented out IPython magic to ensure Python compatibility.
# %run 'multi_instance_object_detection.py'

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

# Commented out IPython magic to ensure Python compatibility.
#import libraries required
# %matplotlib inline
from pycocotools.coco import COCO
import numpy as np
import matplotlib.pyplot as plt
import skimage.io as io
import random
import os
import skimage
from shutil import copyfile
import cv2

#get the annotation file and call an instance of it
train_annot_path =
'/content/drive/MyDrive/Purdue/annotations_trainval2014/annotations/instances_tra
in2014.json'
train_annot = COCO(train_annot_path)

valid_annot_path =
'/content/drive/MyDrive/Purdue/annotations_trainval2014/annotations/instances_val
2014.json'
valid_annot = COCO(valid_annot_path)

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

#check the categories for train data
catgs_ids = train_annot.getCatIds(catNms = classes)
catgs = train_annot.loadCats(catgs_ids)
catgs.sort(key=lambda x:x['id'])
print(catgs)

#get image ids for train data
train_ids = []
```

```
for c in classes:
  ids = train_annot.getImgIds(catIds = train_annot.getCatIds([c]))
  for i in ids:
    train_ids.append(i)

len(train_ids)

#load train data images
train_imgs_load = train_annot.loadImgs(train_ids)
len(train_imgs_load)

#check the categories for validation data
catgs_ids = valid_annot.getCatIds(catNms = classes)
catgs = valid_annot.loadCats(catgs_ids)
catgs.sort(key=lambda x:x['id'])
print(catgs)

#get image ids for validation data
valid_ids = []

for c in classes:
  ids = valid_annot.getImgIds(catIds = valid_annot.getCatIds([c]))
  for i in ids:
    valid_ids.append(i)

len(valid_ids)

#load validation data images
valid_imgs_load = valid_annot.loadImgs(valid_ids)
len(valid_imgs_load)

catgs

#set labels for categories
train_labels = {}
for i,cls in enumerate(classes):
  for c in catgs:
    if c['name'] == cls:
      train_labels[c['id']] = i

print(train_labels)

train_imgs_load[4000]

#plotting an image to check for correctness of classes (from demo doc)
```

```
trial_img =
train_annot.loadImgs(train_ids[np.random.randint(0,len(train_ids))])[0]

I = io.imread(trial_img['coco_url'])
plt.axis('off')
plt.imshow(I)
plt.show()

# load and display instance annotations (from demo doc)
# plt.imshow(I); plt.axis('off')
annIds = train_annot.getAnnIds(imgIds=trial_img['id'], catIds=catgs_ids,
iscrowd=None)
anns = train_annot.loadAnns(annIds)
print(anns)

fig, ax = plt.subplots(1,1)
image = np.uint8(I)

for ann in anns:
  print(ann)
  [x, y, w, h] = ann['bbox']
  label = train_labels[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, classes[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()

# load and display instance annotations (from demo doc)
annIds = train_annot.getAnnIds(imgIds=trial_img['id'], catIds=catgs_ids,
iscrowd=None)
anns = train_annot.loadAnns(annIds)
print(anns)

min_area = 4096
all_annots = []

filename, headers = urllib.request.urlretrieve(trial_img['coco_url'])

for ann in anns:
  print(ann)
```

```
  if ann['area'] > 4096:
    I = cv2.imread(filename)
    I_resized = cv2.resize(I, (256, 256))

    top_x, top_y, w, h = ann['bbox']
    top_x_scaled = int(top_x * 256 / I.shape[1])
    top_y_scaled = int(top_y * 256 / I.shape[0])
    w_scaled = int(w * 256 / I.shape[1])
    h_scaled = int(h * 256 / I.shape[0])
    bbox_scaled = [top_x_scaled, top_y_scaled, w_scaled, h_scaled]

    label_num = train_labels[ann['category_id']]
    category = classes[label_num]

    all_annots.append({'id': ann['id'],
    'category_name': category,
    'label_num' : label_num,
    'bbox': [top_x_scaled, top_y_scaled, w_scaled, h_scaled]})

#save resized image to train path with its annotations
cv2.imwrite(os.path.join(valid_path, '{}.jpg'.format(ann['image_id'])),
I_resized)
with open(os.path.join(valid_path, '{}.json'.format(ann['image_id'])), 'w') as f:
    json.dump({
        'filename': '{}.jpg'.format(ann['image_id']),
        'width': 256,
        'height': 256,
        'anns': all_annots
        }, f)


# fig, ax = plt.subplots(1,1)
# image = np.uint8(I)

# for ann in anns:
#   print(ann)
#   [x, y, w, h] = ann['bbox']
#   label = train_labels[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, classes[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()

#install libraries
from PIL import Image
import urllib.request
import json

#give directories path
train_path = '/content/drive/MyDrive/Purdue/HW6/train_data'
valid_path = '/content/drive/MyDrive/Purdue/HW6/valid_data'

#Saving training data
for i,img in enumerate(train_imgs_load):
  # print(i)
  #extract image from source path
  filename, headers = urllib.request.urlretrieve(img['coco_url'])

  #get annotations and filter those that do not belong to our categories
  ann_ids = train_annot.getAnnIds(imgIds=img['id'], catIds=catgs_ids)
  anns = train_annot.loadAnns(ann_ids)
  # print(anns)

  min_area = 4096
  train_annots = []

  for ann in anns:
    # print(ann)
    if ann['area'] > min_area:
      I = cv2.imread(filename)
      I_resized = cv2.resize(I, (256, 256))

      top_x, top_y, w, h = ann['bbox']
      top_x_scaled = int(top_x * 256 / I.shape[1])
      top_y_scaled = int(top_y * 256 / I.shape[0])
      w_scaled = int(w * 256 / I.shape[1])
      h_scaled = int(h * 256 / I.shape[0])
      bbox_scaled = [top_x_scaled, top_y_scaled, w_scaled, h_scaled]

      label_num = train_labels[ann['category_id']]
      category = classes[label_num]

      train_annots.append({'id': ann['id'],
                           'category_name': category,
                           'label_num' : label_num,
```

```
                          'bbox': [top_x_scaled, top_y_scaled, w_scaled,
h_scaled]})

  if len(train_annots) != 0:
    # print('done')
    #save resized image to train path with its annotations
    cv2.imwrite(os.path.join(train_path, '{}.jpg'.format(ann['image_id'])),
I_resized)
    with open(os.path.join(train_path, '{}.json'.format(ann['image_id'])), 'w')
as f:
      json.dump({
          'filename': '{}.jpg'.format(ann['image_id']),
          'width': 256,
          'height': 256,
          'anns': train_annots
          }, f)

#Saving validation data
for i, img in enumerate(valid_imgs_load):
  print(i)
  #extract image from source path
  filename, headers = urllib.request.urlretrieve(img['coco_url'])

  #get annotations and filter those that do not belong to our categories
  ann_ids = valid_annot.getAnnIds(imgIds=img['id'], catIds=catgs_ids)
  anns = valid_annot.loadAnns(ann_ids)
  # print(anns)

  min_area = 4096
  valid_annots = []

  for ann in anns:
    if ann['area'] > min_area:
      I = cv2.imread(filename)
      I_resized = cv2.resize(I, (256, 256))

      top_x, top_y, w, h = ann['bbox']
      top_x_scaled = int(top_x * 256 / I.shape[1])
      top_y_scaled = int(top_y * 256 / I.shape[0])
      w_scaled = int(w * 256 / I.shape[1])
      h_scaled = int(h * 256 / I.shape[0])
      bbox_scaled = [top_x_scaled, top_y_scaled, w_scaled, h_scaled]

      label_num = train_labels[ann['category_id']]
      category = classes[label_num]
```

```
      valid_annots.append({'id': ann['id'],
                    'category_name': category,
                    'label_num' : label_num,
                    'bbox': [top_x_scaled, top_y_scaled, w_scaled, h_scaled]})

  if len(valid_annots) != 0:
    print('done')
    #save resized image to valid path with its annotations
    cv2.imwrite(os.path.join(valid_path, '{}.jpg'.format(ann['image_id'])),
I_resized)
    with open(os.path.join(valid_path, '{}.json'.format(ann['image_id'])), 'w')
as f:
      json.dump({
          'filename': '{}.jpg'.format(ann['image_id']),
          'width': 256,
          'height': 256,
          'anns': valid_annots
          }, f)

#checking number of files

import os

num_files = len([f for f in os.listdir(train_path) if
os.path.isfile(os.path.join(train_path, f))])
print("Number of files in folder:", num_files)

num_files = len([f for f in os.listdir(valid_path) if
os.path.isfile(os.path.join(valid_path, f))])
print("Number of files in folder:", num_files)

#plot image and check annotations

def plot_imgs(dir_path, cls, imgs, imgs_annots):
  fig, axs = plt.subplots(1,1, figsize = (3,3))

  for i, img_name in enumerate(imgs):
    I = os.path.join(dir_path, img_name)
    image = Image.open(I)
    image = np.uint8(image)

    annots_path = os.path.join(dir_path, imgs_annots[i])
    with open(annots_path) as f:
      annots = json.load(f)
```

```
    for ann in annots['anns']:
      [x, y, w, h] = ann['bbox']
      image = cv2.rectangle(image, (int(x), int(y)), (int(x+w), int(y+h)), (36, 
255, 12), 2)
      image = cv2.putText(image, ann['category_name'], (int(x), int(y-10)), 
cv2.FONT_HERSHEY_SIMPLEX, 0.8, (36, 255, 12), 2)

    axs[i].imshow(image)
    axs[i].set_axis_off()
    if i == 0:
      axs[i].set_title('ground')

  plt.tight_layout()
  fig.savefig('/content/drive/MyDrive/Purdue/cls_{}_imgs.png'.format(cls))

plot_imgs(valid_path, 'bus', ['15517.jpg', '18366.jpg', '21644.jpg'], 
['15517.json', '18366.json', '21644.json'])
plot_imgs(valid_path, 'cat', ['26768.jpg', '12085.jpg', '21396.jpg'], 
['26768.json', '12085.json', '21396.json'])
plot_imgs(valid_path, 'pizza', ['7787.jpg', '271986.jpg', '118739.jpg'], 
['7787.json', '271986.json', '118739.json'])

#import libraries
import torch
from torch.utils.data import Dataset
from torch.utils.data import DataLoader
from PIL import Image
import pandas as pd
import torchvision.transforms as tvt
import torch.nn as nn
import torch.nn.functional as F

#check for GPU
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
device

#create dataset and dataloader
class MyDataset(Dataset):

    #intializations
    def __init__(self, root_dir, transform = None):
        super().__init__()
        self.root_dir = root_dir
        self.transform = transform
```

```
      self.image_paths = []
      self.num_objects = []
      self.labels = []
      self.bbox = []


      #add image paths and corresponding class as a label
      for file in os.listdir(root_dir):
        if file.endswith('.json'):
          with open(os.path.join(self.root_dir, file), 'r') as f:
            img_data = json.load(f)
            # print(len(img_data['anns']))

            if len(img_data['anns']) > 0 and len(img_data['anns']) < 6:
              path = os.path.join(self.root_dir, img_data['filename'])
              if os.path.exists(path):
                self.image_paths.append(path)
                bbox = []
                bbox_label = []

                for i, ann in enumerate(img_data['anns']):
                  bbox_label.append(ann['label_num'])
                  bbox.append(ann['bbox'])

                self.num_objects.append(len(img_data['anns']))
                self.labels.append(bbox_label)
                self.bbox.append(bbox)

      # print(self.num_objects[:20])
      # print(self.labels[:20])
      # print(self.bbox[:20])


  #compute length of dataset
  def __len__(self):
    return len(self.image_paths)

  #apply transformations for the image chosen by index
  def __getitem__(self, index):
    img_path = self.image_paths[index]
    num_objects = self.num_objects[index]
    bboxes = self.bbox[index]
    labels = self.labels[index]

    #load image and normalize pixel values
```

```python
      img = Image.open(img_path).convert('RGB')
      img = np.array(img).astype(np.uint8)
      # img = img/255


      #reshape bbox parameters and scale to (0,1) range
      scaled_bboxes = []
      for bbox in bboxes:
        # print(bbox)
        x1, y1, w, h = bbox
        x2 = x1+w
        y2 = y1+h
        new_bbox = [x1, y1, x2, y2]

        scaled_bboxes.append(new_bbox)

      #   new_bbox[0] /= img.shape[0]
      #   new_bbox[1] /= img.shape[1]
      #   new_bbox[2] /= img.shape[0]
      #   new_bbox[3] /= img.shape[1]

      #   print(new_bbox)

      #perform transformations if any
      if self.transform:
          img = self.transform(img)


      tensor_labels = torch.zeros(5, dtype=torch.uint8) + 13
      tensor_bbox = torch.zeros(5, 4, dtype=torch.uint8)

      for i in range(num_objects):
        bbox = scaled_bboxes[i]
        label = labels[i]

        tensor_bbox[i] = torch.LongTensor(bbox)
        tensor_labels[i] = label

      return img, num_objects, tensor_labels, tensor_bbox

#initialize the dataset and dataloader and apply transformations as required

transform = tvt.Compose([tvt.ToTensor(), tvt.Normalize([0.5, 0.5, 0.5],[0.5, 0.5,
0.5])])
```

```
train_dataset = MyDataset(train_path, transform = transform)
val_dataset = MyDataset(valid_path, transform = transform)

#check for the data length
print(len(train_dataset))
print(len(val_dataset))

#choosing an image
index = 15
print(train_dataset[index])
print(val_dataset[index])

import sys

#initialize batch and num workers
batch_size = 8
num_workers = 2

#create dataloader
train_data_loader = DataLoader(train_dataset, batch_size = batch_size, shuffle =
True, num_workers=num_workers)
val_data_loader = DataLoader(val_dataset, batch_size = batch_size, shuffle =
True, num_workers=num_workers)

#ResNet Block (Inspired from Prof Kak's RPG SkipBlock)

class ResBlock(nn.Module):
  def __init__(self, in_ch, out_ch, downsample=False, skip_connections=True):
    super(ResBlock, self).__init__()
    self.downsample = downsample
    self.skip_connections = skip_connections
    self.in_ch = in_ch
    self.out_ch = out_ch
    #convolution layer and batch normalization
    self.conv1 = nn.Conv2d(in_ch, out_ch, 3, stride=1, padding=1)
    norm_layer1 = nn.BatchNorm2d
    self.bn1 = norm_layer1(out_ch)
    #downsampler - convolution layer
    if downsample:
        self.downsampler = nn.Conv2d(in_ch, out_ch, 1, stride=2)

  def forward(self, x):
    #residual input
    identity = x.clone()
```

```
    #convolution and batch normalization,
relu
    out = self.conv1(x)
    out = self.bn1(out)
    out = torch.nn.functional.relu6(out)
    #check for input and output channels
    if self.in_ch == self.out_ch:
        out = torch.nn.functional.relu6(out)
    #check downsampling
    if self.downsample:
        out = self.downsampler(out)
        identity = self.downsampler(identity)
    #check for skip connections
    if self.skip_connections:
        if self.in_ch == self.out_ch:
            out = out + identity
        else:
            out[:,:self.in_ch,:,:] = out[:,:self.in_ch,:,:] + identity
            out[:,self.in_ch:,:,:] = out[:,self.in_ch:,:,:] +
identity
    return out

#Yolo Network (Inspired from Prof Kak's RPG)

class YOLO(nn.Module):
  def __init__(self, depth = 8, skip_connections = True):
    super(YOLO, self).__init__()

    if depth not in [8,10,12,14,16]:
        sys.exit("This network has only been tested for 'depth' values 8, 10, 12,
14, and 16")

    self.depth = depth // 2
    self.conv1 = nn.Conv2d(3, 64, 3, padding=1)
    self.pool = nn.MaxPool2d(2, 2)
    self.bn1  = nn.BatchNorm2d(64)
    self.bn2  = nn.BatchNorm2d(128)

    #layers for downsampling and changing to 128 channels
    self.skip64_arr = nn.ModuleList()
    for i in range(self.depth):
      self.skip64_arr.append(ResBlock(64, 64, skip_connections =
skip_connections))
    self.skip64ds = ResBlock(64, 64, downsample=True,
skip_connections=skip_connections)
```

```
    self.skip64to128 = ResBlock(64, 128, skip_connections=skip_connections )

    #layers for downsampling
    self.skip128_arr = nn.ModuleList()
    for i in range(self.depth):
      self.skip128_arr.append(ResBlock(128,128,
skip_connections=skip_connections))
    self.skip128ds = ResBlock(128,128, downsample=True,
skip_connections=skip_connections)
    # self.skip128to256 = ResBlock(128, 256, skip_connections=skip_connections )

    self.fc_seqn = nn.Sequential(
        nn.Linear(128*32*32, 4096),
        nn.ReLU(inplace=True),
        nn.Linear(4096, 2048),
        nn.ReLU(inplace=True),
        nn.Linear(2048, 6*6*5*9)
#grid_size*grid_size*num_anchorboxes*yolovector_size
        )

  def forward(self, x):
    #applying the initialized layers in sequence
    x = self.pool(torch.nn.functional.relu6(self.conv1(x)))

    for i,skip64 in enumerate(self.skip64_arr[:self.depth//4]):
        x = skip64(x)
    x = self.skip64ds(x)

    for i,skip64 in enumerate(self.skip64_arr[self.depth//4:]):
        x = skip64(x)
    x = self.bn1(x)
    x = self.skip64to128(x)

    for i,skip128 in enumerate(self.skip128_arr[:self.depth//4]):
        x = skip128(x)
    # x = self.bn2(x)
    x = self.skip128ds(x)
    # x = self.skip128to256(x)

    x = x.view(-1, 128*32*32)

    x = self.fc_seqn(x)
    return x

#initialize model
```

```
torch.cuda.empty_cache()
model = YOLO(depth=8, skip_connections = True)
model = model.to(device)

#list total number of layers
num_layers = len(list(model.parameters()))
num_layers

#check model summary
from torchsummary import summary

summary(model,(3, 256, 256))

#give image size and epochs
image_size = [256, 256]
epochs = 10

#training

#initialize the three losses
criterion1 = nn.BCELoss()
criterion2 = nn.MSELoss()
criterion3 = nn.CrossEntropyLoss()

#store running loss average
loss_avg = []
bce_train_loss = []
mse_train_loss = []
ce_train_loss = []

#intialize optimizer
optimizer = torch.optim.SGD(model.parameters(), lr = 1e-5, momentum = 0.9)

#give the yolo interval, grid size, anchor boxes
yolo_interval = 40
num_yolo_cells = (image_size[0]//yolo_interval) * (image_size[1]//yolo_interval)
#6*6
num_anchor_boxes = 5
max_num_objs = 5

#start training
model.train()
for epoch in range(epochs):
  print(epoch, 'done')
  running_loss = 0.0
```

```
running_bce_loss = 0.0
running_mse_loss = 0.0
running_ce_loss = 0.0

for i, data in enumerate(train_data_loader):
  yolo_tensor = torch.zeros(batch_size, num_yolo_cells, num_anchor_boxes, 8)

  img, num_objects, tensor_labels, tensor_bbox = data

  imgs = img.to(device)
  bboxes = tensor_bbox.to(device)
  labels = tensor_labels.to(device)

  # print(bboxes.shape)
  # print(labels.shape)

  num_cells_image_width = image_size[0] // yolo_interval
  num_cells_image_height = image_size[1] // yolo_interval

  height_center_bb = torch.zeros(imgs.shape[0], 1).float().to(device)
  width_center_bb = torch.zeros(imgs.shape[0], 1).float().to(device)
  obj_bb_height = torch.zeros(imgs.shape[0], 1).float().to(device)
  obj_bb_width = torch.zeros(imgs.shape[0], 1).float().to(device)


  #loop through images in batch
  for ibx in range(imgs.shape[0]):
    for idx in range(max_num_objs):

      #compute center and height,width of boundary box
      height_center_bb = (bboxes[ibx, idx, 1] + bboxes[ibx, idx, 3]) //2
      width_center_bb = (bboxes[ibx, idx, 0] + bboxes[ibx, idx, 2]) //2

      obj_bb_height = bboxes[ibx, idx, 3] - bboxes[ibx, idx, 1]
      obj_bb_width = bboxes[ibx, idx, 2] - bboxes[ibx, idx, 0]

      if(obj_bb_height < 4.0) or (obj_bb_width < 4.0): continue

      #computing for the cell postion that has object i.e i and j coordinates
      cell_row_idx = (height_center_bb / yolo_interval).int()
      cell_col_idx = (width_center_bb / yolo_interval).int()
      cell_row_idx = torch.clamp(cell_row_idx, max=num_cells_image_height - 1)
      cell_col_idx = torch.clamp(cell_col_idx, max=num_cells_image_width - 1)

      #get boundaries in terms of cell height and width
```

```
        bh = obj_bb_height.float() / yolo_interval
        bw = obj_bb_width.float() / yolo_interval

        #calculating the center of object
        obj_center_x = (bboxes[ibx,idx][0].float() + bboxes[ibx,idx][2].float())
/ 2.0
        obj_center_y = (bboxes[ibx,idx][1].float() + bboxes[ibx,idx][3].float())
/ 2.0

        #switching back from (x,y) to (i,j) coordinate format
        yolocell_center_i = cell_row_idx*yolo_interval + float(yolo_interval) /
2.0
        yolocell_center_j = cell_col_idx*yolo_interval + float(yolo_interval) /
2.0
        del_x = (obj_center_x.float() - yolocell_center_j.float()) /
yolo_interval
        del_y = (obj_center_y.float() - yolocell_center_i.float()) /
yolo_interval

        class_label_of_object = labels[ibx,idx].item()

        if class_label_of_object == 13: continue

        #check for aspect ratio to assign anchor box
        AR = obj_bb_height.float() / obj_bb_width.float()
        if AR <= 0.2:
          anch_box_index = 0
        if 0.2 < AR <= 0.5:
          anch_box_index = 1
        if 0.5 < AR <= 1.5:
          anch_box_index = 2
        if 1.5 < AR <= 4.0:
          anch_box_index = 3
        if AR > 4.0:
          anch_box_index = 4

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

        yolo_cell_index = cell_row_idx.item() * num_cells_image_width +
cell_col_idx.item()
        yolo_tensor[0, yolo_cell_index, anch_box_index] = yolo_vector
```

```
        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
element of yolo_vector
      for icx in range(num_yolo_cells):
        for iax in range(num_anchor_boxes):
          if yolo_tensor_aug[ibx,icx,iax,0] == 0:
            yolo_tensor_aug[ibx,icx,iax,-1] = 1

      optimizer.zero_grad()
      output = model(imgs)

      if(output.shape[0] == 8):

        predictions_aug = output.view(batch_size, num_yolo_cells,
num_anchor_boxes,9)
        loss = torch.tensor(0.0, requires_grad=True).float().to(device)
        bce_loss = torch.tensor(0.0, requires_grad=True).float().to(device)
        mse_loss = torch.tensor(0.0, requires_grad=True).float().to(device)
        ce_loss = 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[ibx,icx,iax]
              target_yolo_vector = yolo_tensor_aug[ibx,icx,iax]

              ##  Estimating presence/absence of object and the Binary Cross
Entropy section:
              object_presence = nn.Sigmoid()(torch.unsqueeze(pred_yolo_vector[0],
dim=0))
              target_for_prediction = torch.unsqueeze(target_yolo_vector[0],
dim=0)
              bceloss = criterion1(object_presence, target_for_prediction)
              bce_loss += bceloss
              loss += bceloss

              ## MSE section for regression params:
              pred_regression_vec =
pred_yolo_vector[1:5]
```

```
                pred_regression_vec = torch.unsqueeze(pred_regression_vec,
dim=0)
                target_regression_vec = torch.unsqueeze(target_yolo_vector[1:5],
dim=0)
                regression_loss = criterion2(pred_regression_vec,
target_regression_vec)
                mse_loss += regression_loss
                loss += regression_loss

                ##  CrossEntropy section for object class label:
                probs_vector =
pred_yolo_vector[5:]
                probs_vector = torch.unsqueeze( probs_vector, dim=0
)
                target =
torch.argmax(target_yolo_vector[5:])

                target = torch.unsqueeze( target, dim=0
)
                class_labeling_loss = criterion3(probs_vector, target)
                ce_loss += class_labeling_loss
                loss += class_labeling_loss

        loss.backward()
        optimizer.step()
        running_loss += loss.item()
        running_bce_loss+= bce_loss.item()
        running_mse_loss += mse_loss.item()
        running_ce_loss += ce_loss.item()

    if(i+1)%100 == 0:
      avg_loss = running_loss / 100
      print("[ epoch : %d, batch : %5d] mean_loss : %.3f" %(epoch + 1, i + 1,
avg_loss))
      loss_avg.append(avg_loss)

      bce_train_loss.append(running_bce_loss/100)
      mse_train_loss.append(running_mse_loss/100)
      ce_train_loss.append(running_ce_loss/100)

      running_bce_loss = 0.0
      running_mse_loss = 0.0
      running_ce_loss = 0.0
      running_loss = 0.0
```

```
#plotting the loss vs iterations
plt.plot(bce_train_loss, label = 'BCE')
plt.plot(mse_train_loss, label = 'MSE')
plt.plot(ce_train_loss, label = 'CE')

plt.title('Training loss vs Iterations')
plt.xlabel('Iteration')
plt.ylabel('Loss')
plt.legend()
plt.show()

from matplotlib.patches import Rectangle

#evaluation
class_list = ['bus', 'cat', 'pizza']
model.eval()

with torch.no_grad():
  for i, data in enumerate(val_data_loader):
    img, num_objects, tensor_labels, tensor_bbox = data

    imgs = img.to(device)
    bboxes = tensor_bbox.to(device)
    labels = tensor_labels.to(device)

    preds = model(imgs)
    preds = preds.reshape((imgs.shape[0], 36, 5, 9)).detach()

    #finding the best anchor box to retain the corresponding cells
    for ibx in range(preds.shape[0]):

      best_cell_anchBox = {ic: None for ic in range(36)}

      for icx in range(preds.shape[1]):
        pred_cell = preds[ibx, icx]
        curr_best = 0

        for anc_box in range(pred_cell.shape[0]):
          if pred_cell[anc_box][0] > pred_cell[curr_best][0]:
            curr_best = anc_box
        best_anchor_box = curr_best
        best_cell_anchBox[icx] = best_anchor_box

      sorted_icx = sorted(best_cell_anchBox, key = lambda x: preds[ibx, x,
best_cell_anchBox[x]][0].item(), reverse = True)
```

```
      retained_cells = sorted_icx[:5]

      objects_detected = []
      predicted_label_index_vals = []

      predicted_boxes = []
      predicted_classes_for_boxes = []


      for k, icx in enumerate(retained_cells):
        pred_vec = preds[ibx, icx, best_cell_anchBox[icx]]

        #for labels
        pred_cls_label = pred_vec[-4:]
        pred_cls_labels_probs = torch.nn.Softmax(dim=0)(pred_cls_label)[:-1]

        if torch.all(pred_cls_labels_probs < 0.2):
          predicted_cls_label = None
        else:
          best_predicted_cls_index = (pred_cls_labels_probs ==
pred_cls_labels_probs.max())
          best_predicted_cls_index = torch.nonzero(best_predicted_cls_index,
as_tuple=True)
          predicted_label_index_vals.append(best_predicted_cls_index[0].item())

          predicted_cls_label = class_list[best_predicted_cls_index[0].item()]
          predicted_classes_for_boxes.append(predicted_cls_label)

          #for bbox
          pred_bbox_vec = pred_vec[1:5].cpu()

          del_x, del_y, h, w = pred_bbox_vec[0], pred_bbox_vec[1],
pred_bbox_vec[2], pred_bbox_vec[3]
          h *= yolo_interval
          w *= yolo_interval

          #find cell_indexes
          cell_row_idx, cell_col_idx = icx //6, icx % 6

          #bbox centers
          bb_cx = cell_col_idx * yolo_interval + yolo_interval/2 +
del_x*yolo_interval
          bb_cy = cell_row_idx * yolo_interval + yolo_interval/2 +
del_y*yolo_interval
```

```
          bb_xpred = int(bb_cx - w/2.0)
          bb_ypred = int(bb_cy - h/2.0)

          bbox_pred = [bb_xpred, bb_ypred, int(w), int(h)]

          predicted_boxes.append(bbox_pred)

      if(i+1)%20 == 0:
        img_plot = img[ibx].cpu().numpy().transpose((1,2,0))
        img_plot = (img_plot+1)/2
        fig, axs = plt.subplots(1,2)

        #plot GT image
        axs[0].imshow(img_plot)
        for idx, bbox in enumerate(tensor_bbox[ibx]):
          x, y, w, h = np.array(bbox)
          rect = Rectangle((x,y), w, h, edgecolor = 'b', fill = False)
          axs[0].add_patch(rect)
          axs[0].set_title('Ground Truth')

          if tensor_labels[ibx][idx] != 13:
            cls = class_list[tensor_labels[ibx][idx]]
            axs[0].annotate(cls, (x,y-1), color = 'blue', weight = 'bold',
fontsize = 8)

        #plot pred image
        axs[1].imshow(img_plot)
        for idx, bbox in enumerate(predicted_boxes):
          x, y, w, h = np.array(bbox)
          rect = Rectangle((x,y), w, h, edgecolor = 'g', fill = False)
          axs[1].add_patch(rect)
          axs[1].set_title('Prediction')

          cls = predicted_classes_for_boxes[idx]
          axs[1].annotate(cls, (x,y-1), color = 'green', weight = 'bold',
fontsize = 8)


  plt.show()

## training using batch processing (trial)

#criterion1 = nn.BCELoss()
# criterion2 = nn.MSELoss()
# criterion3 = nn.CrossEntropyLoss()
```

```
# loss_avg = []
# bce_train_loss = []
# mse_train_loss = []
# ce_train_loss = []

# optimizer = torch.optim.SGD(model.parameters(), lr = 1e-5, momentum = 0.9)

# yolo_interval = 40
# num_yolo_cells = (image_size[0]//yolo_interval) *
(image_size[1]//yolo_interval) #6*6
# num_anchor_boxes = 5
# max_num_objs = 5

# model.train()

# for epoch in range(epochs):
#   print(epoch, 'done')
#   running_loss = 0.0
#   running_bce_loss = 0.0
#   running_mse_loss = 0.0
#   running_ce_loss = 0.0
#   num_batches = 0

#   for i, data in enumerate(train_data_loader):
#     yolo_tensor = torch.zeros(batch_size, num_yolo_cells, num_anchor_boxes, 8)
#     img, num_objects, tensor_labels, tensor_bbox = data

#     imgs = img.to(device)
#     num_objects = num_objects.to(device)
#     bboxes = tensor_bbox.to(device)
#     labels = tensor_labels.to(device)

#     print(num_objects.shape)
#     print(bboxes.shape)
#     print(labels.shape)

#     num_cells_image_width = image_size[0] // yolo_interval
#     num_cells_image_height = image_size[1] // yolo_interval

#     height_center_bb = torch.zeros(imgs.shape[0], 1).float().to(device)
#     width_center_bb = torch.zeros(imgs.shape[0], 1).float().to(device)
#     obj_bb_height = torch.zeros(imgs.shape[0], 1).float().to(device)
#     obj_bb_width = torch.zeros(imgs.shape[0], 1).float().to(device)
```

```
#     #compute center and height, width of boundary box
#     height_center_bb = (bboxes[:, :, 1] + bboxes[:, :, 3]) //2
#     width_center_bb = (bboxes[:, :, 0] + bboxes[:, :, 2]) //2

#     obj_bb_height = bboxes[:, :, 3] - bboxes[:, :, 1]
#     obj_bb_width = bboxes[:, :, 2] - bboxes[:, :, 0]

#     # if torch.any(obj_bb_height < 4.0) or torch.any(obj_bb_width < 4.0):
continue

#     #computing for the cell postion that has object i.e i and j coordinates
#     cell_row_idx = (height_center_bb / yolo_interval).int()
#     cell_col_idx = (width_center_bb / yolo_interval).int()
#     cell_row_idx = torch.clamp(cell_row_idx, max=num_cells_image_height - 1)
#     cell_col_idx = torch.clamp(cell_col_idx, max=num_cells_image_width - 1)

#     #get boundaries in terms of cell height and width
#     bh = obj_bb_height.float() / yolo_interval
#     bw = obj_bb_width.float() / yolo_interval

#     #calculating the center of object
#     obj_center_x = (bboxes[:,:,0].float() + bboxes[:,:,2].float()) / 2.0
#     obj_center_y = (bboxes[:,:,1].float() + bboxes[:,:,3].float()) / 2.0

#     #switching back from (x,y) to (i,j) coordinate format
#     yolocell_center_i = cell_row_idx*yolo_interval + float(yolo_interval) / 2.0
#     yolocell_center_j = cell_col_idx*yolo_interval + float(yolo_interval) / 2.0
#     del_x = (obj_center_x.float() - yolocell_center_j.float()) / yolo_interval
#     del_y = (obj_center_y.float() - yolocell_center_i.float()) / yolo_interval

#     print(labels)
#     class_label_of_object = labels[:,0].squeeze().long()
#     print(class_label_of_object)

#     if class_label_of_object == 13: continue

#     #check for aspect ratio to assign anchor box
#     AR = obj_bb_height.float() / obj_bb_width.float()
#     if AR <= 0.2:
#       anch_box_index = 0
#     if 0.2 < AR <= 0.5:
#       anch_box_index = 1
#     if 0.5 < AR <= 1.5:
#       anch_box_index = 2
#     if 1.5 < AR <= 4.0:
```

```
#       anch_box_index = 3
#     if AR > 4.0:
#       anch_box_index = 4

#     yolo_vector = torch.FloatTensor([0, del_x.item(), del_y.item(), bh.item(),
bw.item(), 0, 0, 0])
#     yolo_vector[0] = 1
#     yolo_vector[5 + class_label_of_object] = 1

#     print(yolo_vector)

#     yolo_cell_index = cell_row_idx.item() * num_cells_image_width +
cell_col_idx.item()
#     yolo_tensor[0, yolo_cell_index, anch_box_index] = yolo_vector

#     yolo_tensor_aug = torch.zeros(batch_size, num_yolo_cells, num_anchor_boxes,
9).float().to(device)
#     yolo_tensor_aug[:,:,:,:-1] = yolo_tensor

#     print(yolo_tensor_aug)

#     ## If no object is present, throw all the prob mass into the extra 9th
element of yolo_vector
#     for icx in range(num_yolo_cells):
#       for iax in range(num_anchor_boxes):
#         if yolo_tensor_aug[:,icx,iax,0] == 0:
#           yolo_tensor_aug[:,icx,iax,-1] = 1

#     optimizer.zero_grad()
#     output = model(imgs)

#     print(output.shape)

#     predictions_aug = output.view(batch_size, num_yolo_cells,
num_anchor_boxes,9)

#     loss = torch.tensor(0.0, requires_grad=True).float().to(device)
#     bce_loss = torch.tensor(0.0, requires_grad=True).float().to(device)
#     mse_loss = torch.tensor(0.0, requires_grad=True).float().to(device)
#     ce_loss = torch.tensor(0.0, requires_grad=True).float().to(device)

#     pred_yolo_vector = predictions_aug[:, :, :, :]
#     target_yolo_vector = yolo_tensor_aug[:, :, :, :]
```

```
#     ##  Estimating presence/absence of object and the Binary Cross Entropy
section:
#     object_presence = nn.Sigmoid()(torch.unsqueeze(pred_yolo_vector[:, :, :,
0], dim=3))
#     target_for_prediction = torch.unsqueeze(target_yolo_vector[:, :, :, 0],
dim=3)
#     bceloss = criterion1(object_presence, target_for_prediction)
#     bce_loss += bceloss
#     loss += bceloss
#     print(loss)

#     ## MSE section for regression params:
#     pred_regression_vec = pred_yolo_vector[:, :, :,
1:5]
#     pred_regression_vec = torch.unsqueeze(pred_regression_vec,
dim=3)
#     target_regression_vec = torch.unsqueeze(target_yolo_vector[:, :, :, 1:5],
dim=3)
#     regression_loss = criterion2(pred_regression_vec, target_regression_vec)
#     mse_loss += regression_loss
#     loss += regression_loss

#     ##  CrossEntropy section for object class label:
#     probs_vector = pred_yolo_vector[:, :, :,
5:]
#     probs_vector = torch.unsqueeze(probs_vector,
dim=3)
#     target = torch.argmax(target_yolo_vector[:, :, :,
5:])
#     target = torch.unsqueeze( target, dim=0
)
#     class_labeling_loss = criterion3(probs_vector, target)
#     ce_loss += class_labeling_loss
#     loss += class_labeling_loss

#     loss.backward()
#     optimizer.step()

#     running_loss += loss.item()
#     running_loss_bce += bce_loss.item()
#     running_loss_mse += mse_loss.item()
#     running_loss_ce += ce_loss.item()

#     if(i+1)%100 == 0:
#       avg_loss = running_loss / 100
```

```
#       print("[ epoch : %d, batch : %5d] mean_loss : %.3f" %(epoch + 1, i + 1, 
avg_loss))
#       loss_avg.append(avg_loss)

#       bce_train_loss.append(running_loss_bce/100)
#       mse_train_loss.append(running_loss_mse/100)
#       ce_train_loss.append(running_loss_ce/100)

#       running_loss_bce = 0.0
#       running_loss_mse = 0.0
#       running_loss_ce = 0.0
#       running_loss = 0.0
```

****