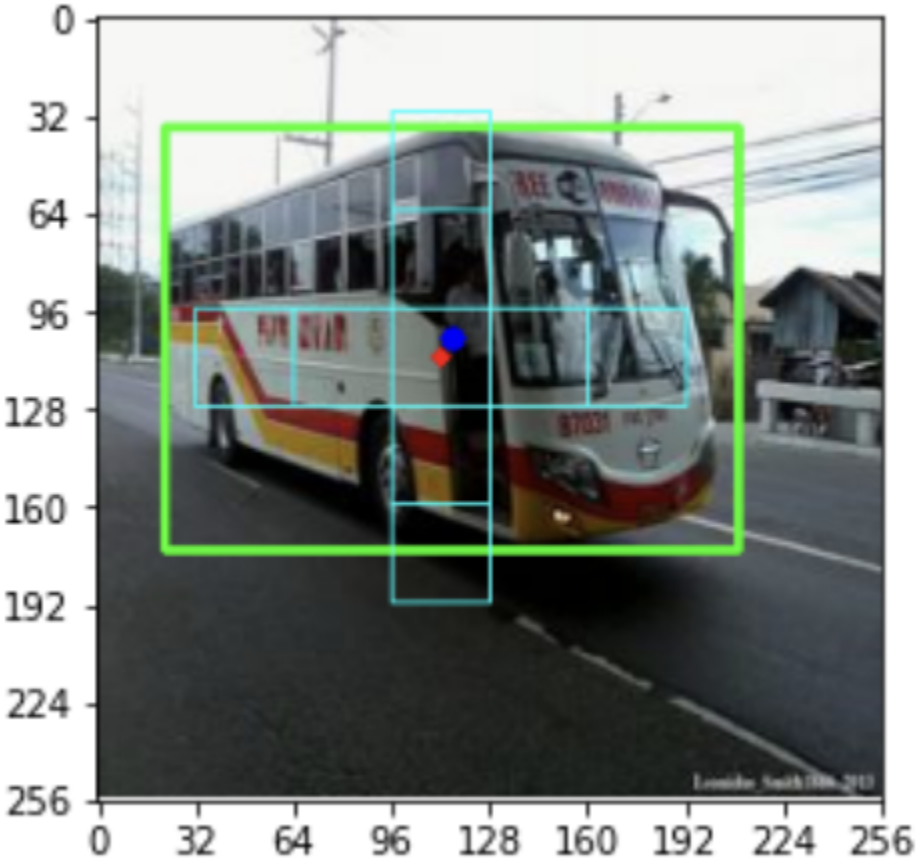
In this lab, we are given images and need to predict multiple instances of an object class and the bounding boxes of those objects. The COCO dataset was split into training and validation. Below is an output of 3 images from each of the three classes. The plot purposefully chose images in the dataset that had multiple object instances in an image.

Building the Network:

This network used a ground truth yolo tensor of the shape 8 x 8 x 5 x 8. Specifically, that means that there were 8 yolo grid cells in the horizontal direction, 8 yolo grid cells in the vertical direction. 8 grid cells in each direction was chosen because it was easily divisible by a 256 x 256 image. It would make the size of each grid cell 32 x 32 pixels. Furthermore, for each yolo grid cell, there were 5 possible anchor boxes, which is what the 5 in the shape above

represents. Anchor boxes have ratios of 5 x 1, 3 x 1, 1 x 1, 1 x 3, and 1 x 5. If the 0th anchor box was chosen in the tensor above, it would mean the 5x1 anchor box was chosen. A 1 x 1 anchor box has a height and width of the yolo interval, which was set to 32 in this lab. To make this more concrete, below is an example of a ground truth image with the possible anchor box choices:



Note that the red dot in the middle of the anchor boxes is the center of the yolo grid cell. The blue dot is the center of the bounding box. In this case, we can visually see that the object instance's bounding box center belongs to the 4th yolo grid cell in the horizontal direction and the 4th yolo grid cell in the vertical direction.

The final dimension, 8, is meant to represent the yolo vector within the yolo tensor. The first element represents "objectness" - whether an object's center is present in that yolo grid and anchor box combination. The second and third elements are the displacement of the center of the yolo grid (in pixel terms) compared to the center of the bounding box in the x and y directions, respectively. The fourth element is the log of the bounding box width minus the log of the anchor box width. The fifth element is the log of the bounding box height minus the log of

the anchor box height. The sixth, seventh, and eighth elements are a one-hot encoded vector of the class of the object represented in that specific yolo vector.
The network architecture mainly consisted of downsampled skip blocks which were then put through linear operators to a final output vector size of 2,560 (which is the size of the yolo tensor). Using this output, we could compute the loss of the output yolo tensor against the ground truth yolo tensor.

In the following pages are the code blocks for the network. Note that due to screenshot dimensions, the code blocks are going across multiple pages.

Below is a code block for the network architecture:

```
class ResBlock(nn.Module):
    def __init__(self, in_ch, out_ch):
        super(ResBlock, self).__init__() #make sure inherited classes are instantiated
        self.in_ch = in_ch
        self.out_ch = out_ch
        self.convo1 = nn.Conv2d(in_ch, out_ch, kernel_size = 3, stride = 1, padding = 1) #creates instance of
        #Conv2d that preserves shape of input!
        self.convo2 = nn.Conv2d(in_ch, out_ch, kernel_size = 3, stride = 1, padding = 1) #preserve shape
        self.convo3 = nn.Conv2d(in_ch, out_ch, kernel_size = 3, stride = 1, padding = 1) #preserve shape
        self.bn1 = nn.BatchNorm2d(out_ch) #batch norm to improve performance
        self.bn2 = nn.BatchNorm2d(out_ch) #batch norm to improve performance
        self.bn3 = nn.BatchNorm2d(out_ch) #batch norm to improve performance

    def forward(self, x):
        identity = x #save input so we can add it to the output
        #run three convolutions that preserve the shape, does BN, and applies an activation function
        out = F.relu(self.bn1(self.convo1(x)))
        out = F.relu(self.bn2(self.convo2(x)))
        out = F.relu(self.bn3(self.convo3(x)))
        # add input to output to give the network more of the original signal
        out = out + identity
        return out



class HW6Net(nn.Module):
    def __init__(self, input_nc, output_nc, ngf=8, n_blocks = 4): #ngf = num of conv filters in first convo layer
    #n_blocks is number of ResNet blocks
        assert(n_blocks >=0)
        super(HW6Net, self).__init__()
        # 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 = model + [nn.Conv2d(ngf * mult, ngf * mult * 2, kernel_size = 3, stride = 2, padding = 1),
                             nn.BatchNorm2d(ngf * mult * 2),
                             nn.ReLU(True)]
        #add own Skip blocks
```

```
                            nn.ReLU(True)]
        #add own Skip blocks
        mult = 2 ** n_downsampling
        for i in range(n_blocks):
            model = model + [ResBlock(64, 64)] # do 4 skip connections with 64 in_ch and 64 out_ch
        self.model = nn.Sequential(*model)
        ### YOLO head ###

        yolo_head = [nn.Linear(16384, 10000), nn.ReLU(inplace=True), nn.Linear(10000, 2560)]
        self.yolo_head = nn.Sequential(*yolo_head)

#         class_head = [nn.Linear(16384, 3)] #classification task just takes the skip connection's output and
#         #applies linear function
#         self.class_head = nn.Sequential(*class_head)
#         ### bounding box regression

#         bbox_head_conv = [nn.Conv2d(64, 64, kernel_size = 3, padding = 1), nn.BatchNorm2d(64),
#                           nn.ReLU(inplace = True), nn.Conv2d(64, 64, 3 , 1),
#                           nn.BatchNorm2d(64), nn.ReLU(inplace = True)]
#         self.bbox_head_conv = nn.Sequential(*bbox_head_conv)
#         bbox_head_fc = [nn.Linear(12544, 1024), nn.ReLU(inplace = True), nn.Linear(1024,512),
#                         nn.ReLU(inplace = True),
#                        nn.Linear(512,4), nn.Sigmoid()] #sigmoid at the end to keep values between 0-1
#         self.bbox_head_fc = nn.Sequential(*bbox_head_fc)
# #         for i in range(n_blocks):
    def forward(self, x):
        #skip blocks:
        ft = self.model(x)
        #classification task:
        ft_r = ft.view(ft.shape[0], -1) #reshape output of skipblock to vector, then convert it to
        # a vector (from original shape)
        yolo_tensor = self.yolo_head(ft_r) #run classification task, return the 3 node output
#         #bbox regression task
#         bbox_temp = self.bbox_head_conv(ft)
#         bbox_temp = bbox_temp.view(bbox_temp.shape[0], -1)
#         bbox = self.bbox_head_fc(bbox_temp) #return the bounding box.
        return yolo_tensor


```

Logic behind dataloading, training, and evaluation:

Dataloader:
In order to write the dataloader, a dictionary was created that stores each image's bounding box annotations, original image height and original image width. Within the dataloader, the bounding box is rescaled to account for using a 256 x 256 image. The dataloader also implements creation of the yolo tensor. Specifically, 5 anchor boxes were used along with a yolo interval of 32 pixels in each direction. Therefore, there are a total of 64 x 5, or 320 possible yolo vectors in each image. Since the dataloader calls for only passing an image and the ground truth, the ground truth yolo vector was created by finding each bounding box's center and the yolo cell that that specific bounding box center belongs to. Then, the IOU for all the possible anchor box configurations was computed, and the anchor box that gave the largest IOU was chosen as the anchor box for that specific yolo cell. Now that the yolo cell and anchor box was chosen, it is now possible to fill out the 8 elements of the yolo vector (as described earlier in the report).

Training:
In order to run training, the image and ground truth yolo tensor was returned from the dataloader in a batch of 20. The images were fed into the network, and the predicted yolo tensor was returned. Since the output is a 20 x 2,560 matrix, it was reshaped back to 20 x 8 x 8 x 5 x 8, which represents a yolo tensor for each of the 20 images. In order to find the correct parts of the

yolo tensor to compute loss without looping through all the images, I used the function torch.nonzeros() on the ground truth yolo tensor's objectness portion of the yolo vector. Specifically, this means passing in the 0th element of every yolo vector into the nonzero function. This can be done by indexing into [:, :, :, :, 0] of the ground truth batch of 20. Passing that into torch.nonzeros() gets us the indices of the ground truth yolo tensor that contain objects.

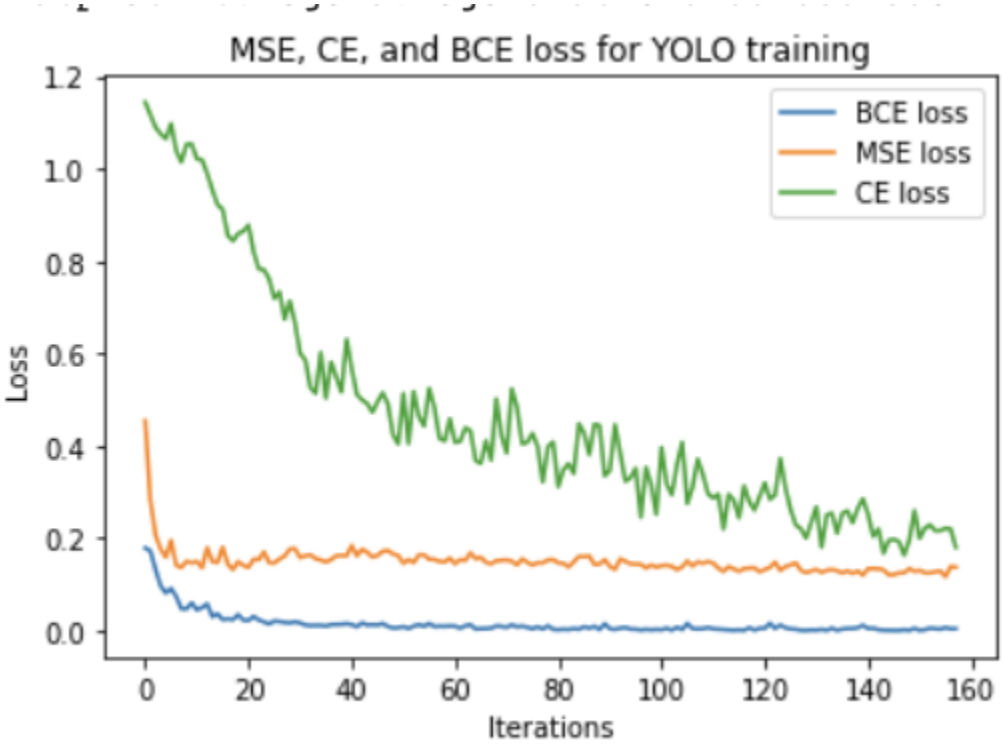
I then passed each index returned from the torch.nonzeros function (about 20 - 30 per batch of 20) and was easily able to access the "objectness", predicted bounding box parameters, and class probabilities vector.

In order to compute the loss, I found the length of the array returned from the nonzeros function, and called that the number of objects in the batch. For the BCE loss, since we are just comparing 2 distinct values, the predicted objectness was a vector of number of object x 1. Using the indices from the torch.nonzero() function, I was able to get the first element of every ground truth and predicted yolo vector and computed the BCE loss.

Similarly, MSE loss was computed except it required creating a tensor of number of objects x 4, since there are 4 values for each bounding box tensor. The prediction and ground truth were then passed into the MSE loss function.

CE loss was also computed in a similar way (the tensor holding all of the predicted classes was number of objects in batch x 3). The 2 number_of_objects x 3 tensors (one representing predictions, one representing ground truth), were passed into the CE loss function.

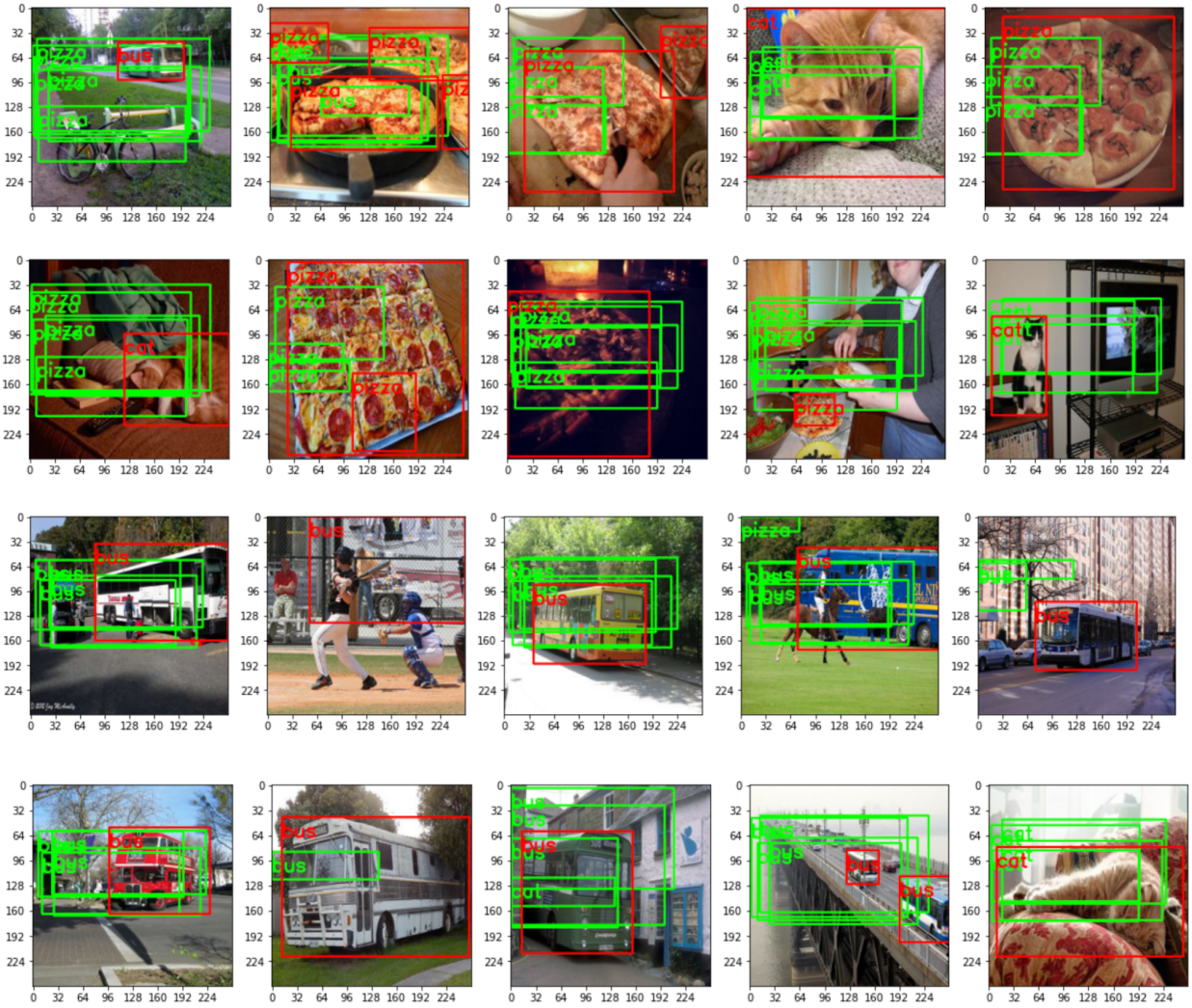
Training was run for 10 epochs total. Below is the output of the loss curves for BCE, MSE, and CE loss.



Evaluation:

To run evaluation, the batch size was updated to 1. I capped the number of total possible object instances to 5. I then captured the values of the 5 highest “objectness” probabilities, assuming that they were above a threshold of 0.99 in the first place, and stored the indices of the resized yolo tensor that those probabilities belonged to (as part of an 8 x 8 x 5 x 8 tensor). I then accessed each of those stored indices, checked if the predicted object was actually larger than 4,096 pixels (since that was the minimum size in the dataset), and returned the corresponding bounding boxes represented in the correct yolo vectors.

Results



Results Discussion:

The detector returns many instances of the correct classification. For example, pizzas, cats, and buses all seemed to classify correctly but with multiple bounding boxes over a single object instance. This inability to return one single instance of a bounding box seems to be a pitfall of the detector - the cap of objects was set to 5 and the threshold was also quite high, but, there were still more instances of objects returned than actual objects in the image. The bounding boxes also seem a bit inaccurate - at an eyeball's glance there seems to be roughly a 50% IOU of the predicted bounding box with the ground truth.

Source Code:

Script to prepare data from COCO:

```python
#Write a script that filters through images and annotations to generate your training and testing dataset
# such that ay image meets the following criteria: 1) contains at least one foreground object from bus, cat, pizza.
# must be at least 64 x 64 = 4096 pixels. There can be multiple foreground objects, but, if there are none, we
# can discard it. Once we have the images downloaded, resize them to 256 x 256 and also rescale the bbox params
# accordingly.
class_list = ['bus', 'cat', 'pizza']

### PREPROCESSING TO GENERATE IMAGE UNIQUE IDS ###
catIds = coco.getCatIds(catNms = class_list)
imgIds = coco.getImgIds(catIds = catIds)
imgIds_list = []
for catId in catIds:
    imgIds_list.append(coco.getImgIds(catIds = catId))
concatenated_list = []
for x in imgIds_list:
    for y in x:
        concatenated_list.append(y)

np_li = np.array(concatenated_list)
unique_imgIds = np.unique(np_li) #make the list unique - there might be duplicate image ids

unique_imgIds = list(unique_imgIds) #convert numpy array back to list

for n, i in enumerate(unique_imgIds):
    unique_imgIds[n] = int(i) #convert each element of list to integer, so it can easily be interpretted by
    #coco api
print(len(unique_imgIds)) #7799
```

Create Training Data Dictionary for Dataloader:

```python
data_dict_train = {}

#os.chdir("/Users/alim/Documents/ECE60146/hw6/")
for n, imgId in enumerate(unique_imgIds):
        annId = coco.getAnnIds(imgId, catIds=catIds, iscrowd=False)
        #now that we have the annotation id, we want to check if there is a dominant object. In order to get
        #the details of that annotation id, we need to load the annotation using that id:
        annId_metadata = coco.loadAnns(annId)
        key = -1 #temp value for key that doesn't exist
        for ann in annId_metadata:
            [x, y, w, h] = ann['bbox']
            #print(ann)
            if ((ann['area'] <= 4096) or (w <= 64) or (h <= 64)):
                continue #do not consider looking at annotation details for that annId
            else:
                #store this annotation:
                imgId_metadata = coco.loadImgs(imgId)
                key = imgId_metadata[0]['file_name']
                if (key in data_dict_train):
                    #we need to append to existing dictionary
                    data_dict_train[key].append({'ann_id' : ann['id'], 'bbox' : ann['bbox'], 'cat_id': ann['category_id'], 'url':imgId_metadata[0]['coco_url'],
                                'img_id' : imgId_metadata[0]['id'], 'img_h' : imgId_metadata[0]['height'],
                                'img_w' : imgId_metadata[0]['width']})

                else:
                    #we need to create new dictionary key
                    data_dict_train[key] = [{'ann_id' : ann['id'], 'bbox' : ann['bbox'], 'cat_id': ann['category_id'], 'url':imgId_metadata[0]['coco_url'],
                                'img_id' : imgId_metadata[0]['id'], 'img_h' : imgId_metadata[0]['height'],
                                'img_w' : imgId_metadata[0]['width']}]
                #download this image:
        #if (key != -1):
        #    coco.download(tarDir = 'train_orig/', imgIds = [data_dict_train[key][0]['img_id']])

#print('this is the data_dict:', data_dict_train)

###ORGANIZATION OF data_dict_train:
#key is file_name
#value is a list of elements.
#element 0: annotation id
#element 1: bbox
#element 2: category id
#element 3: img_id coco_url
#element 4: img_id
#element 5: img height
#element 6: img width
#print("imgId", imgId)
full_len = 0
print(len(data_dict_train)) #6712 images in training dataset
for k in data_dict_train:
    temp_len = len(data_dict_train[k])
    full_len += temp_len
print(full_len) #8343 annotations total
```

Create Validation Data Dictionary for Dataloader:

```
data_dict_val = {}

#os.chdir("/Users/alim/Documents/ECE60146/hw6/")
for n, imgId in enumerate(unique_imgIds_test):
        annId = coco.getAnnIds(imgId, catIds=catIds, iscrowd=False)
        #now that we have the annotation id, we want to check if there is a dominant object. In order to get
        #the details of that annotation id, we need to load the annotation using that id:
        annId_metadata = coco.loadAnns(annId)
        key = -1 #temp value for key that doesn't exist
        for ann in annId_metadata:
            [x, y, w, h] = ann['bbox']
            #print(ann)
            if ((ann['area'] <= 4096) or (w <= 64) or (h <= 64)):
                continue #do not consider looking at annotation details for that annId
            else:
                #store this annotation:
                imgId_metadata = coco.loadImgs(imgId)
                key = imgId_metadata[0]['file_name']
                if (key in data_dict_val):
                    #we need to append to existing dictionary
                    data_dict_val[key].append({'ann_id' : ann['id'], 'bbox' : ann['bbox'],'cat_id': ann['category_id'],
                                               'url':imgId_metadata[0]['coco_url'],
                                    'img_id' : imgId_metadata[0]['id'], 'img_h' : imgId_metadata[0]['height'],
                                   'img_w' : imgId_metadata[0]['width']})

                else:
                    #we need to create new dictionary key
                    data_dict_val[key] = [{'ann_id' : ann['id'], 'bbox' : ann['bbox'], 'cat_id': ann['category_id'], 'url':imgId_metadata[0]['coco_url'],
                                     'img_id' : imgId_metadata[0]['id'], 'img_h' : imgId_metadata[0]['height'],
                                    'img_w' : imgId_metadata[0]['width']}]
                #download this image:
        #if (key != -1):
        #    coco.download(tarDir = 'val_orig/', imgIds = [data_dict_val[key][0]['img_id']])

full_len = 0
print(len(data_dict_val)) #3427 images in val dataset
for k in data_dict_val:
    temp_len = len(data_dict_val[k])
    full_len += temp_len
print(full_len) #4198 annotations total in val dataset
```

Create Dataloader:

```
##### build the dataloader.

#root = '/Users/alim/Documents/ECE60146/hw6/'
folders = ['train_resized/', 'val_resized/']

class MyDataset(torch.utils.data.Dataset):
    def __init__(self, folder):
        super(MyDataset).__init__()
        #self.root = root
        self.folder = folder
        if (self.folder == 'train_resized/'):
            self.data_dict = data_dict_train
        if (self.folder == 'val_resized/'):
            self.data_dict = data_dict_val

        self.mapping = {6: 0, 17: 1, 59: 2}
        self.one_hot_encoding = {0: torch.tensor(np.array([1, 0, 0])),
                                 1: torch.tensor(np.array([0, 1, 0])),
                                 2: torch.tensor(np.array([0, 0, 1]))}
        self.images = os.listdir(folder) #create list of files in the train or val directory. We will use
        # this list to get bbox params, read image files, etc.

        for img in self.images:
            if (img == ".DS_Store"):
                self.images.remove(".DS_Store") #handle case when image isn't an image. Just remove it from the
                #image list.

        self.to_Tensor_and_Norm = tvt.Compose([tvt.ToTensor(), tvt.Normalize([0], [1])])

        #find the image with the most objects, get that number of objects so we can create an empty list
        # with that number of elements:
        self.max_obj_instances = 0 #this becomes 14 for train, 13 for val
        for key in self.data_dict:
            if len(self.data_dict[key]) > self.max_obj_instances:
                self.max_obj_instances = len(self.data_dict[key])
        self.cell_and_anchors_tracker = np.zeros((8, self.max_obj_instances)) -99 #instantiate to -99
        #tracker is a 3 x 14 matrix. Each col will hold data of an obj instance
        #row 1 and row 2 will be x,y idx of assigned cell
        #row 3 will be the assigned anchor box. The assignments will be made in the __getitem__ function
        #rows 4 to 7 will be the x,y,w,h (normalized to a 256 to 256 image)
        #row 8 is the class in terms of 0, 1, 2



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

    def __getitem__(self, index):
        #prepare image:
        PIL_img = Image.open(self.folder + self.images[index])
        torch_img = self.to_Tensor_and_Norm(PIL_img)

        #prepare class labels
        li_class_labels = []
```

```
        li_class_labels = []
        for i in range(len(self.data_dict[self.images[index]])):
            #print(self.data_dict[self.images[index]])
            class_label = self.data_dict[self.images[index]][i]['cat_id']
            class_label = self.mapping[class_label]
            class_label = torch.tensor(class_label)
            li_class_labels.append(class_label)
        #for each object in the image, figure out which grid cell it belongs (i.e, cell which is closest)
        #to the center of the bbox.
        #will have a 8 by 8 grid cell, so, the yolo interval will be 256 / 8 = 32
        #get bbox center:
        yolo_interval = 32
        bbox_anchor_list = []
        yolo_tensor = torch.zeros(8,8,5,8)
        dict_assigned_yolo_vectors = {}

        for i in range(len(self.data_dict[self.images[index]])): #iterate through each object instance
            if i == 0: #start to print image for sanity check
                np_img = np.uint8(PIL_img)
                ticks = np.linspace(start = 0, stop = 256, num=9)


            [x, y, w, h] = self.data_dict[self.images[index]][i]['bbox']
            img_w = self.data_dict[self.images[index]][i]['img_w']
            img_h = self.data_dict[self.images[index]][i]['img_h']
            #scale x y w h in terms of a 256 x 256 image.
            scaling_factor_x = 256.0 / img_w
            scaling_factor_y = 256.0 / img_h
            x = scaling_factor_x * x
            y = scaling_factor_y * y
            w = scaling_factor_x * w
            h = scaling_factor_y * h
            #done scaling original x, y, w, h parameters

            #find center of bbox
            cx_bbox = (x + x + w) / 2 #center of bbox (x coord)
            cy_bbox = (y + y + h) / 2 #center of bbox  (y coord)

            #find the yolo cell that the bbox belongs to:
            yolo_grid_x_idx = cx_bbox // yolo_interval
            yolo_grid_y_idx = cy_bbox // yolo_interval
            #find the exact yolo cell center coords. We will need this to compute dx, dy, the difference
            #between center of bbox and center of yolo grid
            yolo_grid_x_center = yolo_grid_x_idx * yolo_interval + (yolo_interval / 2)
            yolo_grid_y_center = yolo_grid_y_idx * yolo_interval + (yolo_interval / 2)

            #compute displacements between the center of the cell and the center of the bounding box in a
            #training image for a given object instance (dx and dy)
            dx =  cx_bbox - yolo_grid_x_center
            dy =  cy_bbox -  yolo_grid_y_center

            #plot bbox, center of yolo grid cell that was assigned, and center of bbox
            np_img = cv2.rectangle(np_img, (int(x), int(y)),
                                   (int((x+w)), int((y+h))), (0,255,0), 2)
            #image, center_coordinates, radius, color, thickness
```

```
            #image, center_coordinates, radius, color, thickness
            np_img = cv2.circle(np_img, (int(cx_bbox), int(cy_bbox)), 1,
                                (0,0,255),  5)
            np_img = cv2.circle(np_img, (int(yolo_grid_x_center), int(yolo_grid_y_center)), 1, (255, 0, 0), 4)

            #pick the correct anchor box. There are 5 anchor boxes with different aspect ratio:
            # 5 x 1, 3 x 1, 1 x 1, 1 x 3, 1 x 5
            #iou_box function in pytorch expects that coords are in [x1 y1 x2 y2] format.
            #we already know the bbox:
            GT_bbox_for_iou = torch.tensor([x, y, x+w, y+h]).view(1,4) #view reshapes to 1,4
            IOU_li = []
            anchor_box_options = [[5,1], [3,1], [1,1], [1,3], [1, 5]]
            for n, ab in enumerate(anchor_box_options):

                bbox_anchor_w = yolo_interval * ab[0]
                bbox_anchor_h = yolo_interval * ab[1]
                bbox_anchor = torch.tensor([yolo_grid_x_center, yolo_grid_y_center, bbox_anchor_w, bbox_anchor_h]).view(1,4)
                #above bbox_anchor is in cx,cy,w,h fmt
                bbox_anchor = ops.box_convert(bbox_anchor, in_fmt = 'cxcywh', out_fmt = 'xyxy')
                bbox_anchor = torch.clamp(bbox_anchor, 0, 256)
                bbox_anchor_list.append(bbox_anchor)

                np_img = cv2.rectangle(np_img, (int(bbox_anchor_list[n+(5*i)][0][0]), int(bbox_anchor_list[n+(5*i)][0][1])), (int(bbox_anchor_list[n+(5*i)][0][2]),
                                    int(bbox_anchor_list[n+(5*i)][0][3])), (0, 255, 255), 1)

                IOU = ops.box_iou(GT_bbox_for_iou, bbox_anchor)
                IOU_li.append(IOU)
                if n == 4:
                    temp_arr = np.empty(5)
                    for k, iou in enumerate(IOU_li):
                        temp_arr[k] = iou.view(-1).item() #put computed IOUs into nparray format so we can perform argmax
                    max_IOU_idx = np.argmax(temp_arr)
                    anchor_mapping = {0: [5,1], 1: [3, 1], 2: [1, 1], 3: [1, 3], 4: [1,5]} #gets from index to h/w of anchor box
                    #create yolo vector's first 3 elements:
                    yolo_vector = torch.empty((1,8))
                    yolo_vector[0,0] = 1
                    yolo_vector[0,1] = dx / yolo_interval
                    yolo_vector[0,2] = dy / yolo_interval
                    #compute sigma_w, sigma_h, the ratio between GT and anchor w and ratio between GT and anchor h
                    best_anchor_w = anchor_mapping[max_IOU_idx][0]
                    best_anchor_h = anchor_mapping[max_IOU_idx][1]
                    w_GT = torch.log(torch.tensor(w)) - torch.log(torch.tensor(best_anchor_w * yolo_interval))
                    h_GT = torch.log(torch.tensor(h)) - torch.log(torch.tensor(best_anchor_h * yolo_interval))


                    yolo_vector[0,3] = w_GT
                    yolo_vector[0,4] = h_GT
                    #assign class label as one hot vector:
                    one_hot_vector =  self.one_hot_encoding[int(li_class_labels[i])]
                    numerical_class = torch.argmax(one_hot_vector)
                    #print("ONE HOT VEC", one_hot_vector)
                    yolo_vector[0,5:] = one_hot_vector
                    #print("FINAL YOLO VECTOR", yolo_vector.view(-1))
                    #print("anchor box chosen was:", anchor_mapping[max_IOU_idx], "where anchbox is W, H")
                    #print("yolo cell was:", yolo_grid_x_idx, yolo_grid_y_idx)
```

```
                    h_GT = torch.log(torch.tensor(h)) - torch.log(torch.tensor(best_anchor_h * yolo_interval))


                    yolo_vector[0,3] = w_GT
                    yolo_vector[0,4] = h_GT
                    #assign class label as one hot vector:
                    one_hot_vector =  self.one_hot_encoding[int(li_class_labels[i])]
                    numerical_class = torch.argmax(one_hot_vector)
                    #print("ONE HOT VEC", one_hot_vector)
                    yolo_vector[0,5:] = one_hot_vector
                    #print("FINAL YOLO VECTOR", yolo_vector.view(-1))
                    #print("anchor box chosen was:", anchor_mapping[max_IOU_idx], "where anchbox is W, H")
                    #print("yolo cell was:", yolo_grid_x_idx, yolo_grid_y_idx)
                    #assign yolo_vector to the correct index of yolo_tensor:
                    #print('assigned to:',int(yolo_grid_x_idx), int(yolo_grid_y_idx), max_IOU_idx)

                    yolo_tensor[int(yolo_grid_x_idx), int(yolo_grid_y_idx),max_IOU_idx,:] = yolo_vector
                    dict_assigned_yolo_vectors[(int(yolo_grid_x_idx), int(yolo_grid_y_idx),max_IOU_idx)] = \
                    {'grid_x' : int(yolo_grid_x_idx), 'grid_y' : int(yolo_grid_y_idx), 'anchor_box_idx' : max_IOU_idx}
                    self.cell_and_anchors_tracker[:,i] = [yolo_grid_x_idx, yolo_grid_y_idx, max_IOU_idx, x, y, w, h, numerical_class]
                    #print("yolo_vector", yolo_vector)
            # if i == (len(self.data_dict[self.images[index]])  - 1):
            #     plt.imshow(np_img)
            #     plt.xticks(ticks)
            #     plt.yticks(ticks)

        return torch_img, yolo_tensor, self.cell_and_anchors_tracker

train_dataset = MyDataset( folders[0])
index = 6
train_dataset[index][1].shape
batch_size = 1
#create the dataloader:
train_dataloader = torch.utils.data.DataLoader(train_dataset, batch_size=batch_size, num_workers = 0, drop_last=True, shuffle=True)
# for n, i in enumerate(train_dataloader):
#     print(n)
#     print(i[2])

val_dataset = MyDataset(folders[1])
val_dataloader = torch.utils.data.DataLoader(val_dataset, batch_size = 1, num_workers = 0, drop_last = True, shuffle = True)
# for n, i in enumerate(val_dataloader):
#     print(n)

```

Code for Model was given above

Code for Training Loop:

```
#create training loop:
torch.autograd.set_detect_anomaly(True)

test_net = HW6Net(input_nc = 3, output_nc = 3, ngf = 4, n_blocks=4)
test_net = test_net.to(torch.device("cuda:0"))
total_loss_list = []
bce_loss_list = []
ce_loss_list = []
mse_loss_list = []
criterion_CE = nn.CrossEntropyLoss()
criterion_MSE = nn.MSELoss()
criterion_BCE = nn.BCELoss()
optimizer = torch.optim.Adam(test_net.parameters(), lr = 1e-3, betas = (0.9, 0.99))
epochs = 10
num_layers = len(list(test_net.parameters()))
print(num_layers)

for epoch in range(epochs):
    running_loss_CE = 0.0
    running_loss_MSE = 0.0
    running_loss_BCE = 0.0
    print('in epoch', epoch)
    for n, data in enumerate(train_dataloader):
        imgs, GT_yolo_tensor, grid_info = data
        imgs = imgs.to(torch.device("cuda:0"))
        GT_yolo_tensor = GT_yolo_tensor.to(torch.device("cuda:0"))
        optimizer.zero_grad()
        output_yolo_tensor = test_net(imgs) #output is 20 x 2560

        #compute BCE loss for whether there is an object present or not in specific anchor box:
        GT_yolo_tensor = torch.reshape(GT_yolo_tensor, (batch_size, 8, 8, 5, 8))
        output_yolo_tensor = torch.reshape(output_yolo_tensor, (batch_size, 8, 8, 5, 8))
        #print(GT_yolo_tensor.shape)
        #find idx of non-zeros:
        non_zero_idx = torch.nonzero(GT_yolo_tensor[:,:,:,:,0])


        #compute BCE loss for object vs not object:
        num_objects = non_zero_idx.shape[0] #find the total number of objects in the GT.
        #for each yolo vector, we will return 6 non-zero values (2 of the 8 are 0 because of the 3 class
        #one hot encoding). We want to store the predictions of every 6th index returned from the non-zero
        #indices as those will contain the "predicted objectness", or the first element of the yolo vector.
        #Since we want to store every 6th index, we need to divide the total length of the non-zero indices
        #by 6 so we have the correct length vector of predictions to feed into BCE loss.
        #prepare arrays for pred/GT objectness:
        pred_objectness = torch.zeros(num_objects)
        GT_objectness = torch.ones(num_objects) #note that 3 is the batch size

        #prepare arrays for pred/GT bbox:
        pred_bbox = torch.zeros(num_objects , 4)
        GT_bbox = torch.zeros(num_objects, 4)

        #prepare arrays for pred/GT classification:
```

```
        #prepare arrays for pred/GT classification:
        pred_class_prob = torch.zeros(num_objects, 3)
        GT_classes = torch.zeros(num_objects).type(torch.LongTensor) #torch expects GT categories to be
        #"long tensors", which are really just integers in torch. So, we convert this to long tensor.
        #compute CE loss for classification:

        for idx, i in enumerate(non_zero_idx):
            #the following line gets pred objectness for one of the inputs into BCE loss.
            pred_objectness[idx] = nn.Sigmoid()(output_yolo_tensor[i[0], i[1], i[2], i[3],0])

            #these following 2 lines will be dx, dy, w ratio, h ratio for pred and GT for input into MSE loss
            pred_bbox[idx, :]   = output_yolo_tensor[i[0], i[1], i[2], i[3], 1:5]
            GT_bbox[idx, :]     = GT_yolo_tensor[i[0], i[1], i[2], i[3], 1:5]

            #the following 2 lines get the pred/GT classifications for input into CE loss
            pred_class_prob[idx,:] = output_yolo_tensor[i[0], i[1], i[2], i[3], 5:]
            GT_classes[idx] = torch.argmax(GT_yolo_tensor[i[0], i[1], i[2], i[3], 5:])


        loss_BCE = criterion_BCE(pred_objectness, GT_objectness) ##remember - BCE is not symmetric! Need
        #to have the first arg be the prediction and the second arg be the target/GT.
        loss_BCE.backward(retain_graph = True)
        #total_loss = total_loss + loss_BCE

        #convert bbox to xyxy fmt:
        pred_bbox_MSE = ops.box_convert(pred_bbox, in_fmt = 'cxcywh', out_fmt = 'xyxy')
        GT_bbox_MSE = ops.box_convert(GT_bbox, in_fmt = 'cxcywh', out_fmt = 'xyxy')

        #compute MSE loss for bbox.
        loss_MSE = criterion_MSE(pred_bbox_MSE, GT_bbox_MSE)
        loss_MSE.backward(retain_graph = True)

        #compute CE loss for classification:
        loss_CE = criterion_CE(pred_class_prob, GT_classes )
        loss_CE.backward()
        running_loss_BCE = running_loss_BCE + loss_BCE.item()
        running_loss_CE = running_loss_CE + loss_CE.item()
        running_loss_MSE = running_loss_MSE + loss_MSE.item()

        optimizer.step()
        if (n+1) % 20 == 0:
            print("[epoch: %d, batch: %5d] loss: %3f" % (epoch + 1, n + 1, running_loss_CE + running_loss_MSE +
                                                         running_loss_BCE / 20))
            print("running_loss_BCE:", running_loss_BCE)
            print("running_loss_CE:", running_loss_CE)
            print("running_loss_MSE:", running_loss_MSE)
            bce_loss_list.append(running_loss_BCE / 20)
            running_loss_BCE = 0.0
            ce_loss_list.append(running_loss_CE / 20)
            running_loss_CE = 0.0
            mse_loss_list.append(running_loss_MSE / 20)
            running_loss_MSE = 0.0
            total_loss_list.append(running_loss_CE + running_loss_MSE +
                                                         running_loss_BCE / 20)
```

Code for Evaluation:

```
#eval code:
#iterate over validation dataset, get predictions with "objectness" and plot the corresponding pred classification
#and bbox and the GT pred/bbox.
count_objects = 0
class_mapping = {0: 'bus', 1: 'cat', 2: 'pizza'}
fig, ax = plt.subplots(2,5, figsize=(20,20))
with torch.no_grad():
    for n, data in enumerate(val_dataloader):
        if n < 10:
            max_obj_count = 0
            print("in val dataloader iteration", n)
            #print("STARTING EVAL CODE")
            images, GT_yolo_tensor, grid_info = data
            images = images.to(torch.device("cuda:0"))
            GT_yolo_tensor = GT_yolo_tensor.to(torch.device("cuda:0"))

            #bbox_label = ops.box_convert(bbox_label, in_fmt = 'xyxy', out_fmt = 'cxcywh')
            pred_yolo_tensor = test_net(images)
            pred_yolo_tensor = torch.reshape(pred_yolo_tensor, (8,8, 5, 8))
            GT_yolo_tensor = torch.reshape(GT_yolo_tensor, (8, 8, 5, 8))


            #open image, put into numpy fmt
            img_np = np.array(images.squeeze(dim = 0).cpu())
            img_np = img_np.transpose(1,2,0)
            img_PIL = Image.fromarray(np.uint8(img_np * 255)) #this was originally a tensor, so, need to unnormalize!!
            img_np = np.uint8(img_PIL)


            #get maximum probabilities of "objectness of prediction" and once we have found those indices, plot their corresponding bboxes:

            max_objectness = np.zeros((5))
            max_objectness_pos = np.zeros((5,4))
            for i in range(8):
                for j in range(8):
                    for k in range(5):
                        temp_objectness = np.array(nn.Sigmoid()(pred_yolo_tensor[i,j,k,0].cpu()))
                        if (temp_objectness > 0.999):
                            if np.min(max_objectness) < temp_objectness:
                              idx_temp = np.argmin(max_objectness)
                              max_objectness[idx_temp] = temp_objectness #store probability of object if current probability is higher than the lowest probability in the top 5 objects
                              max_objectness_pos[idx_temp] = [i,j,k,0]

                        if (i == 7) and (j == 7) and (k == 4):
                          for num, x in enumerate(max_objectness_pos):
                            #print("MAX objectness is", max_objectness)
                            if np.array_equal(np.zeros((5)), max_objectness):
                              continue #didn't find any objects above threshold!
                            #print("X IS ", x)
                            #print("the max objectnes pos is", max_objectness_pos)
                            #print("max_objectness is", max_objectness)
                            #we know this has a high probability of being an object. Compute h/w to make sure it is a reasonable size.
                            pred_bbox = pred_yolo_tensor[int(x[0]), int(x[1]), int(x[2]), 1:5]
```

```
                            #we know this has a high probability of being an object. Compute h/w to make sure it is a reasonable size.
                            pred_bbox = pred_yolo_tensor[int(x[0]), int(x[1]), int(x[2]), 1:5]

                            center_x = x[0] * 32 + pred_bbox[0] * 32 #predicted yolo cell_x + offset_x from predicted yolo center gives us the predicted center of bbox
                            center_y = x[1] * 32 + pred_bbox[1] * 32 #predicted yolo_cell_y + offset_y from predicted yolo center gives us the predicted center of bbox
                            anchor_mapping = {0: [5,1], 1: [3, 1], 2: [1, 1], 3: [1, 3], 4: [1,5]} #gets from index to w/h of anchor box
                            pred_bbox_w = torch.exp(pred_bbox[2]) * anchor_mapping[x[2]][0] *32 #this value represents which anchor box we are looking at in the prediction.
                            #the above line multiplies the ratio of the GT/anch width by the anchor width by the yolo interval to get the predicted w
                            pred_bbox_h = torch.exp(pred_bbox[3]) * anchor_mapping[x[3]][1] *32
                            #print("pred_bbox_h", pred_bbox_h)
                            if ((pred_bbox_h * pred_bbox_w) < 4096):
                              #print('found an obj too small')
                              continue #object is too small for dataset.
                            max_obj_count += 1
                            # we have passed test for object size. Convert it to xyxy fmt so bbox can be displayed on image:
                            pred_bbox_real_coords = torch.tensor([center_x, center_y, pred_bbox_w, pred_bbox_h])
                            pred_bbox_xyxy = ops.box_convert(pred_bbox_real_coords, in_fmt = 'cxcywh', out_fmt = 'xyxy')
                            pred_bbox_xyxy = torch.clamp(pred_bbox_xyxy, 0, 256)
                            x1 = int(pred_bbox_xyxy[0])
                            y1 = int(pred_bbox_xyxy[1])
                            x2 = int(pred_bbox_xyxy[2])
                            y2 = int(pred_bbox_xyxy[3])
                            #print(x1, y1, x2, y2)
                            #add bbox to image:
                            img_np = cv2.rectangle(img_np, (x1, y1), (x2, y2), (0,255,0), 2)

                            # #get object classification:
                            #print(pred_yolo_tensor[i, j, k, 5:])
                            pred_class_vector = int(torch.argmax(pred_yolo_tensor[int(x[0]), int(x[1]), int(x[2]), 5:]).cpu())
                            #print("pred_class_vector", pred_class_vector)
                            class_mapping = {0: 'bus', 1: 'cat', 2: 'pizza'}
                            pred_classification = class_mapping[pred_class_vector]
                            #print(pred_classification)
                            img_np = cv2.putText(img_np, pred_classification, (x1, y1 + 25),
                                     cv2.FONT_HERSHEY_SIMPLEX, 0.8, (0,255, 0),2)


            ## Plot GT bboxes

            GT_yolo_tens  grid_info.squeeze
            non_zeros =
            non_zero_idx  <built-in method squeeze of Tensor object at 0x7f7dfd821ef0>
            grid_info = grid_info.squeeze(dim=0)
            ticks = np.linspace(start = 0, stop = 256, num=9)
            for num, i in enumerate(non_zero_idx):
              #print("grid_info", grid_info[:,num])
              x1, y1 = grid_info[3:5,num]
              w,  h  = grid_info[5:7, num]
              class_num = grid_info[-1, num]
              x2 = int(x1 + w)
              y2 = int(y1 + h)
              x1 = int(x1)
              y1 = int(y1)
              img_np = cv2.rectangle(img_np, (x1, y1), (x2, y2), (255, 0, 0), 2)
```

```
            x1 = int(x1)
            y1 = int(y1)
            img_np = cv2.rectangle(img_np, (x1, y1), (x2, y2), (255, 0, 0), 2)
            GT_bbox = GT_yolo_tensor[i[0], i[1], i[2], 1:5]
            anchor_mapping = {0: [5,1], 1: [3, 1], 2: [1, 1], 3: [1, 3], 4: [1,5]} #gets from index to w/h of anchor box
            GT_bbox_w = GT_bbox[2] * anchor_mapping[int(i[2])][0] * 32  #this value represents which anchor box we are looking at in the prediction.
            #the above line multiplies the ratio of the GT/anch width by the anchor width by the yolo interval to get the predicted w
            GT_bbox_h = GT_bbox[3] * anchor_mapping[int(i[2])][1] * 32



            img_np = cv2.putText(img_np, class_mapping[int(class_num)], (x1, y1 + 25),
                                 cv2.FONT_HERSHEY_SIMPLEX, 0.8, (255,0, 0),2)
            #print("about to plot image!!! when n is", n)
            if n < 5:
              ax[0,n].set_xticks(ticks)
              ax[0,n].set_yticks(ticks)
              ax[0,n].imshow(img_np)
            if n >4:
              ax[1,n-5].set_xticks(ticks)
              ax[1,n-5].set_yticks(ticks)
              ax[1,n-5].imshow(img_np)



        if n > 10:
            break
#print(count)
```