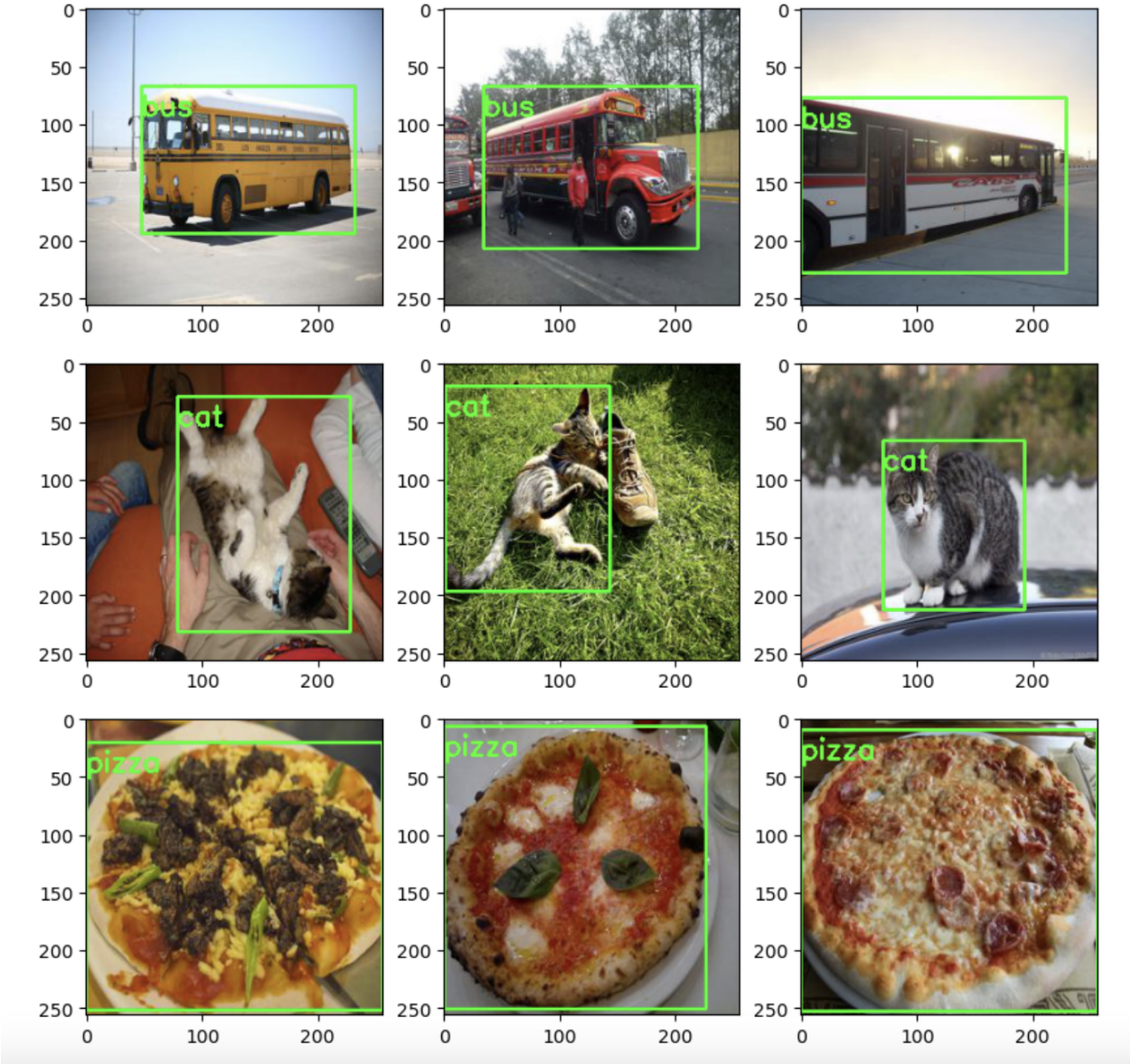
Training Data Generation:

In this lab, we first use the COCO API to generate a dataset of about 4,000 training images and 2,000 validation images with their respective bounding box annotations. The original images were resized to 256 x 256 and their corresponding bounding boxes were resized accordingly. Below is is printout of 3 images from 3 the 3 classes we will aim to classify and localize:



<u>ResBlock and Network Details:</u>

This lab called for a ResBlock. Below is the implementation. It did not use downsampling as the input channel layers equaled the output channel layers:

```python
class ResBlock(nn.Module):
    def __init__(self, in_ch, out_ch):
        super(SkipBlock, 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
```

The network that includes the skip connections is below:

```python
class HW5Net(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(HW5Net, 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
        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)
        ### Classification 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)
        cls = self.class_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 cls, bbox.double()
```

Number of Parameters and Layers:

The total number of parameters and layers of the network were ~13 M and 84, respectively:

The code for these numbers is below. Note that stackoverflow was used to find code for computing total parameters.

```
pytorch_total_params = sum(p.numel() for p in test_net.parameters() if p.requires_grad)
print("Total Parameters:" ,pytorch_total_params)
num_layers = len(list(test_net.parameters()))
print("Total Layers:", num_layers)
```

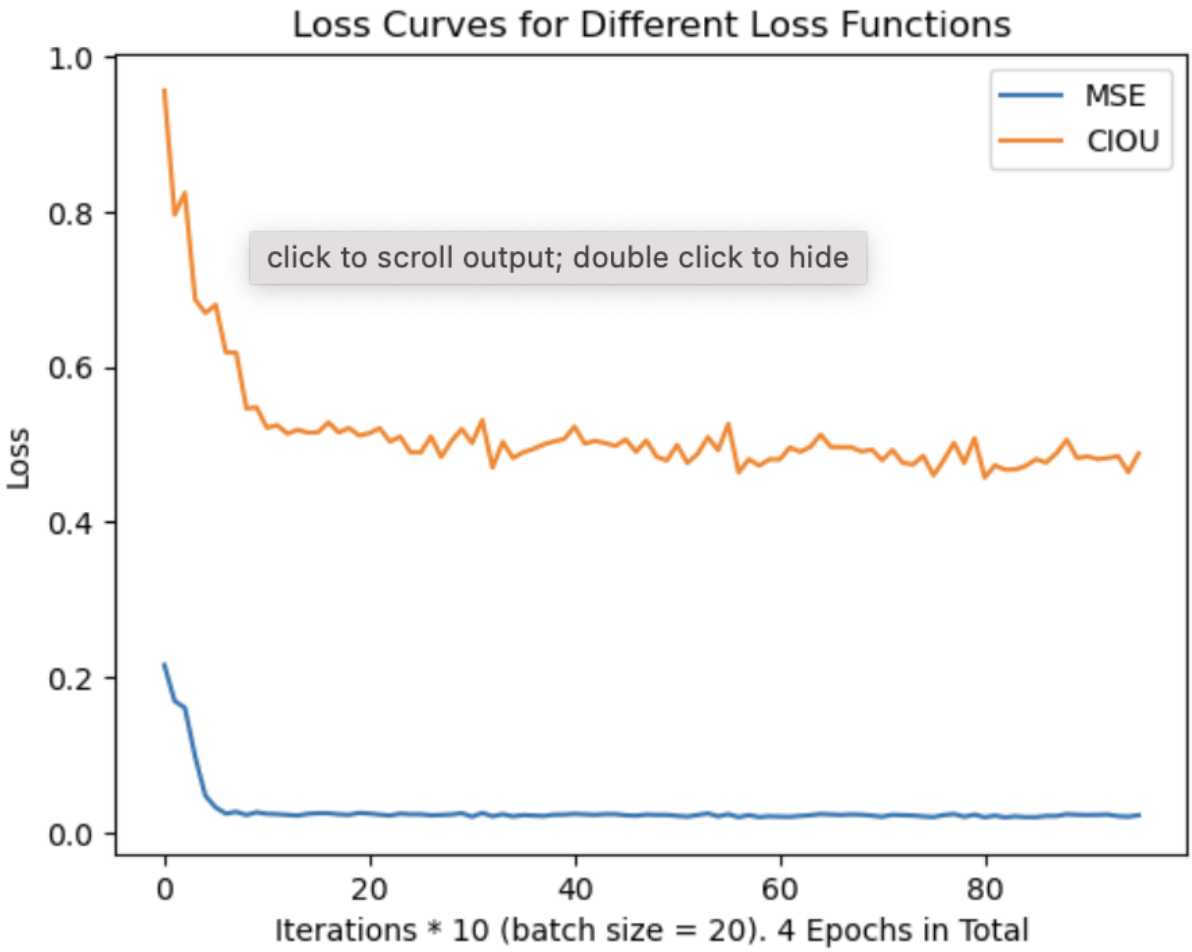
```
Total Parameters: 13966311
Total Layers: 84
```

Results:

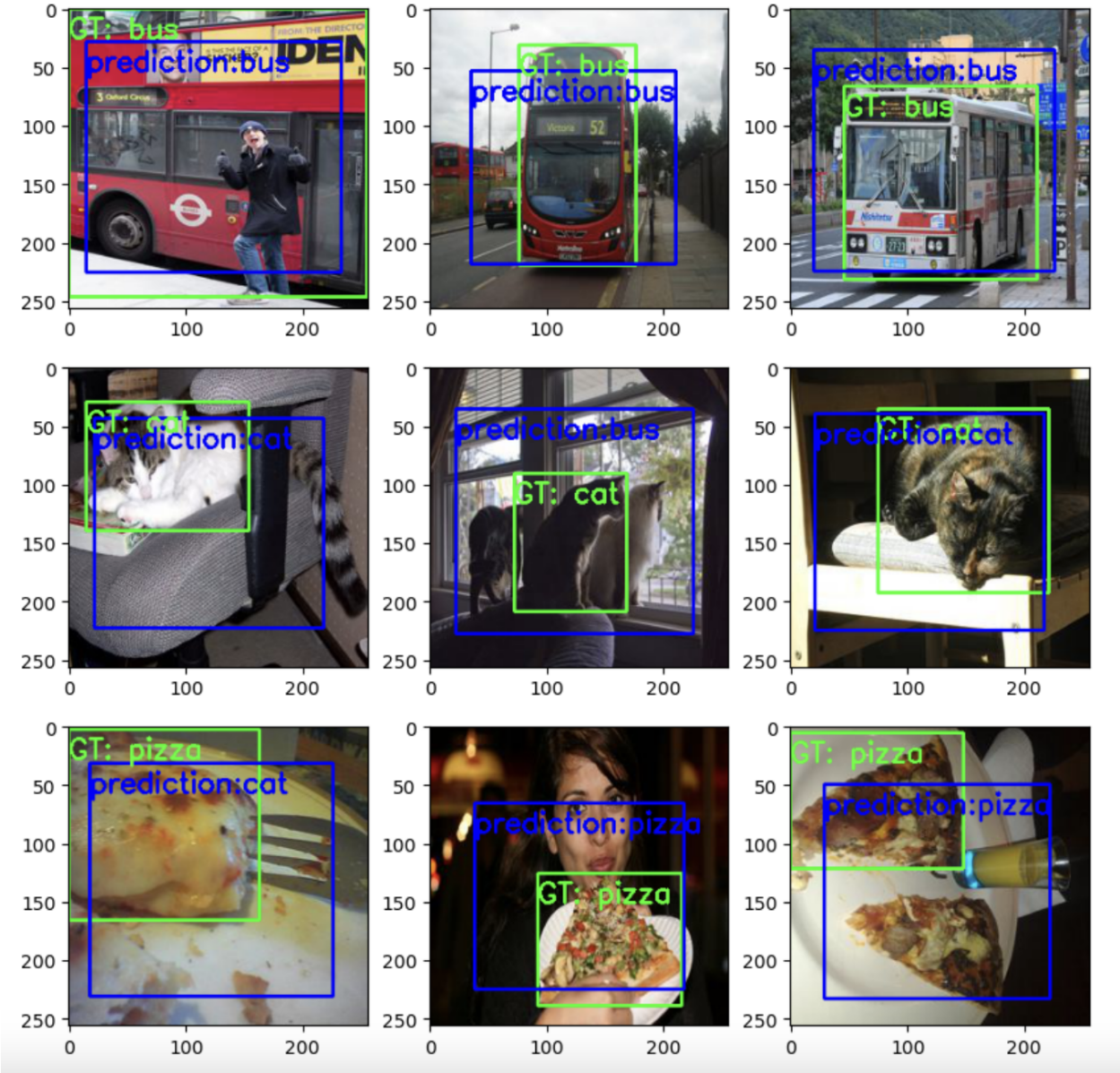
The model was trained and evaluated (the source code for that will be in the following sections). One model was trained with Mean Squared Error Loss while the other was trained with Complete Intersection over Union loss. The mean IoUs were computed for both models on the test dataset. Furthermore, a confusion matrix was computed for each of the models. See below for a summary.

Model - Trained for 4 Epochs	Mean IoU	Overall Classification Accuracy
MSE Loss Model	0.4997	80%
Complete IoU Loss Model	0.5353	80%

Plot of Loss Over Training Iterations:

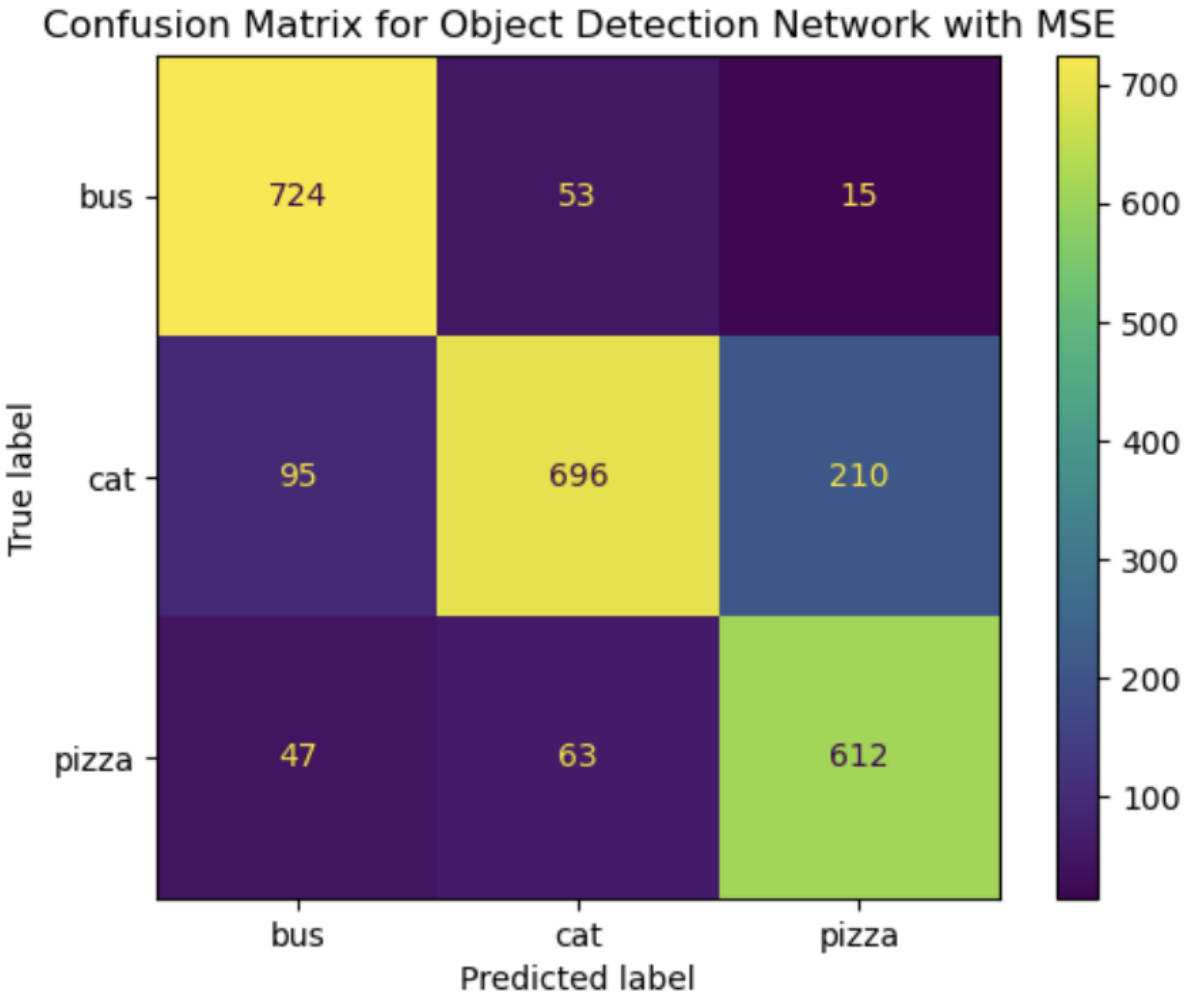

Loss Curves for Different Loss Functions
MSE
CIOU
click to scroll output; double click to hide
Loss
1.0
0.8
0.6
0.4
0.2
0.0
0
20
40
60
80
Iterations * 10 (batch size = 20). 4 Epochs in Total

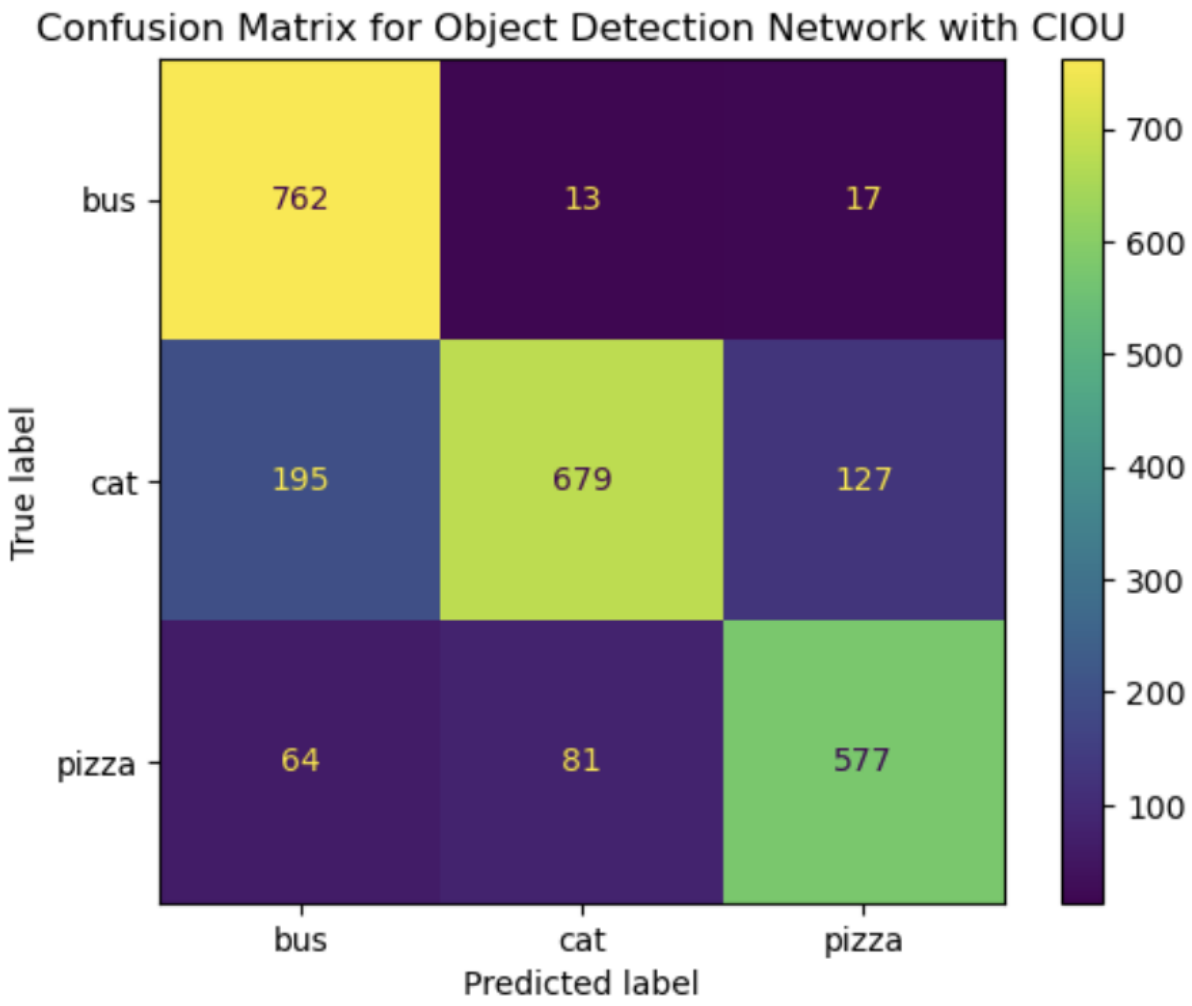
Visualization of Predicted and GT annotation of the dominant object on 3 classes:



Confusion Matrices:

Below are the confusion matrices for the model trained with MSE loss and the model trained with CIoU loss. They both have around 80% accuracy.





Discussion on potential methods to improve pizza detector and results discussion:
One possible method to improve the pizza detector is to use more skip blocks or intersperse the architecture with more of them as the model training loss curves seem to suffer from vanishing gradients fairly early in the training process. I do not believe there is more space for BatchNorms. I trained the models for 4 epochs, but it seems that the optimal parameters were found within 2 epochs. Another method to improve the model is to potentially add downsampling into the skip blocks, as the current skip block only takes in a 64 channel input and outputs a 64 channel feature map.

Overall, the performance of the detector had an accuracy of 80%. For pizza specifically, 577 / 721, or 80 % of predicted pizzas were actually pizzas. For cats, 679 / 773, or 87 % of cats were predicted correctly. For buses, 762 / 1024, or 74% of buses were accurately classified. It is possible that buses have more heterogeneous features in terms of colors, which is why it was harder to predict accurately. Perhaps adding more training data for buses would help improve its accuracy.

Source Code:

Part 1: Code for generating dataset:

This code below shows how dominant objects were found using the COCO API and how the images were resized to 256 x 256. First, we had to find unique image ids, as it is possible that some image ids can be duplicates.

```
### 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


```

The code below shows how a dictionary was created. The dictionary key was the image file name (so it could easily be used by the Dataset class - code for that is below - to provide image metadata as the index ran down the folder structure). For each key, there was a list which contained the bounding box parameters, the annotation id, category id, coco_url, image height and width.

In [4]:

```
### CODE TO DOWNLOAD THE CORRECT IMAGES AND GET THE CORRECT ANNOTATIONS (I.E THE DOMINANT OBJECT)
count = 0
count_true = 0
data_dict = {}
os.chdir("/Users/alim/Documents/ECE60146/hw5/")
for n, imgId in enumerate(unique_imgIds):
        count += 1
        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)
        temp_max_bbox_area = 0 #need a variable to figure out which bbox is the largest, as there can be
        #multiple bboxes per imgId even in a given category!
        max_bbox_area = 0
        found_valid_bbox = False
        key = -1 #temp value for key that doesn't exist
        for ann in annId_metadata:
            len_ann = len(annId_metadata)

            #print(ann['bbox'], "is the bbox for img_id", ann['image_id'])
            [x, y, w, h] = ann['bbox']
            if ((w * h) <= 40000):
                continue #do not consider looking at annotation details for that annId
            else:
                #figure out which annotation for that image id is the largest:
                temp_max_bbox_area = w * h
                if temp_max_bbox_area > max_bbox_area: #if w*h is GT than the current largest bbox area, then
                    #make max_bbox_area the w*h that was just computed. Store the metadata of that ann in
                    #dictionary.
                    found_valid_bbox = True
                    max_bbox_area = temp_max_bbox_area
                    #print("imgId being loaded", imgId)
                    #print(type(imgId))
                    imgId_metadata = coco.loadImgs(imgId)
                    #print("imgId_metadata", imgId_metadata, "for imgId", imgId)
                    #print(imgId_metadata)
                    ###ORGANIZATION OF data_dict:
                    #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)
                    key = imgId_metadata[0]['file_name']
                    data_dict[key] = [ann['id'], ann['bbox'], ann['category_id'], imgId_metadata[0]['coco_url'],
                                      imgId_metadata[0]['id'], imgId_metadata[0]['height'],
                                     imgId_metadata[0]['width']]

        if (key != -1):
            coco.download(tarDir = 'train_orig/', imgIds = [data_dict[key][4]])
        if (found_valid_bbox == True): #check if we found all the expected images
            count_true += 1
```

Part 2: Code to Plot 3 images from each class with their correct bounding box:

```python
# Make a figure of a selection of images from dataset. 3 images from each of the classes, with annotation of
#dominant object.
fig, ax = plt.subplots(3,3 ,figsize = (10,10))
count_bus = 0
count_cat = 0
count_pizza = 0
root = '/Users/alim/Documents/ECE60146/hw5/'

for n, i in enumerate(data_dict):
        cat = data_dict[i][2]
        #print('cat is', cat)
        if (cat == 6) and (count_bus < 3):
            PIL_img = Image.open(root + 'train_resized/' + str(i))
            np_img = np.uint8(PIL_img)
            scaling_factor_x =  256.0 / data_dict[i][6]
            scaling_factor_y =  256.0 / data_dict[i][5]
            [x, y, w, h] = data_dict[i][1]
            np_img = cv2.rectangle(np_img, (int(x * scaling_factor_x), int(y*scaling_factor_y)),
                                   (int((x+w)*scaling_factor_x), int((y+h)*scaling_factor_y)), (0,255,0), 2)
            np_img = cv2.putText(np_img, "bus", (int(x * scaling_factor_x), int(y * scaling_factor_y + 25)),
                  |              cv2.FONT_HERSHEY_SIMPLEX, 0.8, (36,255, 12),2)
            ax[0, count_bus].imshow(np_img)


            count_bus = count_bus + 1

        if (cat == 17) and (count_cat < 3):
            PIL_img = Image.open(root + 'train_resized/' + str(i))
            np_img = np.uint8(PIL_img)
            scaling_factor_x =  256.0 / data_dict[i][6]
            print(data_dict[i][6] )
            scaling_factor_y =  256.0 / data_dict[i][5]
            [x, y, w, h] = data_dict[i][1]

            np_img = cv2.rectangle(np_img, (int(x * scaling_factor_x), int(y*scaling_factor_y)),
                                   (int((x+w)*scaling_factor_x), int((y+h)*scaling_factor_y)), (0,255,0), 2)
            np_img = cv2.putText(np_img, "cat", (int(x * scaling_factor_x), int(y * scaling_factor_y + 25)),
                                   cv2.FONT_HERSHEY_SIMPLEX, 0.8, (36,255, 12),2)
            ax[1, count_cat].imshow(np_img)
            count_cat += 1

        if (cat == 59) and (count_pizza < 3):
            PIL_img = Image.open(root + 'train_resized/' + str(i))
            np_img = np.uint8(PIL_img)

            scaling_factor_x =  256.0 / data_dict[i][6]
            scaling_factor_y =  256.0 / data_dict[i][5]
            [x, y, w, h] = data_dict[i][1]
            np_img = cv2.rectangle(np_img, (int(x * scaling_factor_x), int(y*scaling_factor_y)),
                                   (int((x+w)*scaling_factor_x), int((y+h)*scaling_factor_y)), (0,255,0), 2)
            np_img = cv2.putText(np_img, "pizza", (int(x * scaling_factor_x), int(y * scaling_factor_y + 25)),
                                   cv2.FONT_HERSHEY_SIMPLEX, 0.8, (36,255, 12),2)
            ax[2, count_pizza].imshow(np_img)
            count_pizza += 1
        if ((count_pizza == 3) and (count_bus == 3) and (count_cat == 3)):
            break
```

Part 3: Code for DataLoader:

First, I created a Dataset class which gets each image from the disk. Then, I created the training and validation dataloaders by wrapping the Dataset class in a DataLoader class.

```
root = '/Users/alim/Documents/ECE60146/hw5/'
folders = ['train_resized/', 'val_resized/']

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

        self.mapping = {6: 0, 17: 1, 59: 2}
        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])])

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

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

        #prepare class label
        class_label = self.data_dict[self.images[index]][2]
        class_label = self.mapping[class_label]
        class_label = torch.tensor(class_label)

        #prepare bounding box
        bbox = self.data_dict[self.images[index]][1] #not rescaled. has original values from coco.
        [x, y, w, h] = bbox #bbox is still in terms of original image dims.
        bbox = np.array(bbox)
        img_orig_H = self.data_dict[self.images[index]][5] #H or y
        img_orig_W = self.data_dict[self.images[index]][6] #W or x
        #rescaling bbox so the input is [x1, y1, x2, y2]
        scale_factor_x = 256.0 / img_orig_W
        scale_factor_y = 256.0 / img_orig_H
        bbox[0] = x * scale_factor_x
        bbox[2] = w * scale_factor_x
        bbox[1] = y * scale_factor_y
        bbox[3] = h * scale_factor_y
        bbox[2] = bbox[2] + bbox[0]
        bbox[3] = bbox[3] + bbox[1]

        #normalize bbox to between 0 and 1:
        bbox = bbox / 256.0
        bbox = torch.tensor(bbox) #convert to tensor so it can be used in pytorch for computing loss.

        return torch_img, class_label, bbox
```

Wrapping the DataSet in the DataLoader:

```
my_train_dataset = MyDataset(root = root, folder = folders[0])
index = random.randint(0,200)
print(my_train_dataset[index][0].shape, my_train_dataset[index][1], my_train_dataset[index][2])

my_val_dataset = MyDataset(root = root, folder = folders[1])


my_train_dataloader = torch.utils.data.DataLoader(my_train_dataset, batch_size=20, num_workers = 0, drop_last=False

my_val_dataloader = torch.utils.data.DataLoader(my_val_dataset, batch_size=20, num_workers = 0, drop_last=False)

```

Part 4: Code for Training Loop:
MSE Loss Training Loop:

```python
#TRAINING LOOP FOR MSE LOSS
test_net = HW5Net(input_nc = 3, output_nc = 3, ngf = 4, n_blocks=4)
loss_test_net = []
loss_test_net_MSE = []
criterion_CE = nn.CrossEntropyLoss()
criterion_MSE = nn.MSELoss()
optimizer = torch.optim.Adam(test_net.parameters(), lr = 1e-3, betas = (0.9, 0.99))
epochs = 4
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
    for i, data in enumerate(my_train_dataloader):
        print("in train dataloader iteration", i)
        inputs, class_label, bbox_label = data
        bbox_label = ops.box_convert(bbox_label, in_fmt = 'xyxy', out_fmt = 'cxcywh')
        optimizer.zero_grad() #Sets gradients of all model parameters to zero. We want to compute fresh gradients
        #based on the new forward run.
        outputs = test_net(inputs) #output[0] is cls and output[1] is bbox
        bbox_outputs_converted = ops.box_convert(outputs[1], in_fmt = 'cxcywh', out_fmt = 'xyxy')
        bbox_label = ops.box_convert(bbox_label, in_fmt = 'cxcywh', out_fmt = 'xyxy')
        #print("shape of cls:", outputs[0].shape, "shape of bbox:", outputs[1].shape)
        loss_CE = criterion_CE(outputs[0], class_label) #compute cross-entropy loss
        #print("For computing CE loss, outputs[0] is", outputs[0], "and class_label is", class_label)
        loss_CE.backward(retain_graph = True)
        loss_MSE = criterion_MSE(bbox_outputs_converted, bbox_label)
        #print("bbox output from network:", outputs[1], "label:" ,bbox_label)

        loss_MSE.backward()
        #loss.backward() #compute derivative of loss wrt each gradient.
        optimizer.step() #takes a step on hyperplane based on derivatives
        running_loss_CE += loss_CE.item()
        running_loss_MSE += loss_MSE.item()
        if (i+1) % 10 == 0:
            print("[epoch: %d, batch: %5d] loss: %3f" % (epoch + 1, i + 1, running_loss_CE + running_loss_MSE / 10)
            loss_test_net.append(running_loss_CE + running_loss_MSE/10)
            loss_test_net_MSE.append(running_loss_MSE/10)
            print('running_loss_MSE', running_loss_MSE/10)
            running_loss_CE = 0.0
            running_loss_MSE = 0.0

```

Code for Training Loop (CIOU loss):

```python
#TRAINING LOOP FOR MSE LOSS
test_net = HW5Net(input_nc = 3, output_nc = 3, ngf = 4, n_blocks=4)
loss_test_net = []
loss_test_net_MSE = []
criterion_CE = nn.CrossEntropyLoss()
criterion_MSE = nn.MSELoss()
optimizer = torch.optim.Adam(test_net.parameters(), lr = 1e-3, betas = (0.9, 0.99))
epochs = 4
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
    for i, data in enumerate(my_train_dataloader):
        print("in train dataloader iteration", i)
        inputs, class_label, bbox_label = data
        bbox_label = ops.box_convert(bbox_label, in_fmt = 'xyxy', out_fmt = 'cxcywh')
        optimizer.zero_grad() #Sets gradients of all model parameters to zero. We want to compute fresh gradients
        #based on the new forward run.
        outputs = test_net(inputs) #output[0] is cls and output[1] is bbox
        bbox_outputs_converted = ops.box_convert(outputs[1], in_fmt = 'cxcywh', out_fmt = 'xyxy')
        bbox_label = ops.box_convert(bbox_label, in_fmt = 'cxcywh', out_fmt = 'xyxy')
        #print("shape of cls:", outputs[0].shape, "shape of bbox:", outputs[1].shape)
        loss_CE = criterion_CE(outputs[0], class_label) #compute cross-entropy loss
        #print("For computing CE loss, outputs[0] is", outputs[0], "and class_label is", class_label)
        loss_CE.backward(retain_graph = True)
        loss_MSE = criterion_MSE(bbox_outputs_converted, bbox_label)
        #print("bbox output from network:", outputs[1], "label:" ,bbox_label)

        loss_MSE.backward()
        #loss.backward() #compute derivative of loss wrt each gradient.
        optimizer.step() #takes a step on hyperplane based on derivatives
        running_loss_CE += loss_CE.item()
        running_loss_MSE += loss_MSE.item()
        if (i+1) % 10 == 0:
            print("[epoch: %d, batch: %5d] loss: %3f" % (epoch + 1, i + 1, running_loss_CE + running_loss_MSE / 10)
            loss_test_net.append(running_loss_CE + running_loss_MSE/10)
            loss_test_net_MSE.append(running_loss_MSE/10)
            print('running_loss_MSE', running_loss_MSE/10)
            running_loss_CE = 0.0
            running_loss_MSE = 0.0

```

Part 5: Evaluation Code

Note that the evaluation code below is just for the model trained with CIOU loss. However, the code is basically the same for the evaluation of the model trained with MSE loss - we just need to change one line for the loss function

```python
#EVALUATION LOOP FOR OBJECT DETECTION MODEL TRAINED WITH COMPLETE IOU LOSS
correct = 0
total = 0
y_true = []
y_pred = []
mapping = { 0: 'bus',
            1: 'cat',
            2: 'pizza'}
iou_holder = torch.tensor(0)
with torch.no_grad():
    for n, data in enumerate(my_val_dataloader):
        print("in val dataloader iteration", n)
        #print("STARTING EVAL CODE")
        images, class_labels, bbox_label = data
        bbox_label = ops.box_convert(bbox_label, in_fmt = 'xyxy', out_fmt = 'cxcywh')
        cls_prediction, bbox_prediction = test_net_ciou(images)
        bbox_prediction = ops.box_convert(bbox_prediction, in_fmt = 'cxcywh', out_fmt = 'xyxy')
        bbox_label = ops.box_convert(bbox_label, in_fmt = 'cxcywh', out_fmt = 'xyxy')
        _, predicted = torch.max(cls_prediction.data, 1) # this returns max (_) and max index (predicted)

        total += class_labels.size(0) #add to total's total. This is the denominator for the prediction acc.
        for k, i in enumerate(class_labels): #compute number of correct predictions
            temp = np.array(i) #temp holds the one hot encoded label
            y_true.append(mapping[int(i)])
            y_pred.append(mapping[int(predicted[k])])
            #print("i is ", i)
            idx = np.argmax(temp) #get the argmax of the encoded label - will be a value between 0 and 4.
            if temp == np.array(predicted[k]): #if the predicted value and label match
                correct = correct + 1 #add to correct total

        IOU = ops.box_iou(bbox_label, bbox_prediction)
        print('bbox is', bbox_label)
        print('bbox_pred is', bbox_prediction)
        IOU = np.sum(np.diag(IOU))

        print('IOU for iteration', n, 'is ', IOU)
        iou_holder = iou_holder + IOU



print('Accuracy of the network on the val images: %d %%' % (
    100 * correct / total))
print('mean IOU is ', iou_holder/total)
```

Part 6: Code for Confusion Matrix:

```python
#CREATE CONFUSION MATRIX FOR OBJECT DETECTION MODEL USING COMPLETE IOU LOSS
from sklearn.metrics import confusion_matrix

y_true = y_true
y_pred = y_pred
confusion_matrix=confusion_matrix(y_true, y_pred, labels = [ "bus", "cat", "pizza"])
disp = ConfusionMatrixDisplay(confusion_matrix, display_labels = [  "bus", "cat", "pizza"])
disp.plot()
disp.ax_.set_title("Confusion Matrix for Object Detection Network with CIOU")
plt.show()
```

Part 7: Code for Loss Curves:

```python
saved_loss_test_net_MSE = loss_test_net_MSE #these values were from one full epoch of training.
saved_loss_test_net_CIOU = loss_test_net_ciou
plt.plot(saved_loss_test_net_MSE, label = "MSE")
plt.plot(saved_loss_test_net_CIOU, label = 'CIOU')
plt.title('Loss Curves for Different Loss Functions')
plt.legend()
plt.xlabel('Iterations * 10 (batch size = 20). 4 Epochs in Total')
plt.ylabel('Loss')
```