ECE60146: Homework 6

Submission By: Joseph Wang
wang3450@purdue.edu

Prepared for Dr. Avinash Kak and Dr. Charles Bouman
GTA: Fangda Li and Qiuchen Zhai
Purdue University
March 22, 2023

Contents

List of Figures

1 Introduction

The main goal of this homework assignment is to create a multi-instance object detector. In order to accomplish this, we needed to perform the following tasks:

1. Understand the YOLO logic – how multi-instance object detection can be done with just a single forward pass of the network
2. Implement custom YOLO training and evaluation logic.

2 Building the Dataset

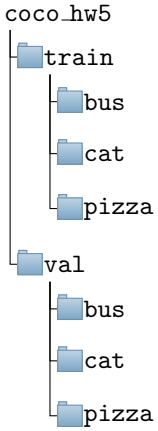
The images we trained and evaluated our deep neural network on are a subset derived from the MS-COCO 2014 Train/Val dataset. The images in our dataset meet the following criteria:

1. Contain at least one object from any of the following three categories: ['bus', 'cat', 'pizza']
2. Contains one or more foreground objects whose bounding box areas exceed $64 \times 64 = 4090$ pixels.
3. The images that meet the two above requirements are accepted into the dataset and resized to 256×256. The bounding box parameters are then adjusted appropriately.

Numerically, our training dataset consists of about 6,000 images and our validation dataset consists of about 3,000 images. To make the job of loading the data during training and validation easier, we stored all the bounding bound parameters and paths to the respective images in a list of dictionaries written to a pickle file.

More specifically, every entry was formatted as such: {'filename': path_to_file, 'bbox': [[x1, y1, x2, y2], [x1, y1, x2, y2], ..., [x1, y1, x2, y2]].

Visually, our training and validation data was organized as such. Each class directory contained the images for that class, as well as a pickle file containing bounding box parameters.



Shown below in Figure 1 is sample of selected images from our dataset with their ground truth bounding boxes. The first row corresponds to images from the bus class, the second row corresponds to images from the cat class, and the last row corresponds to images from the pizza class.

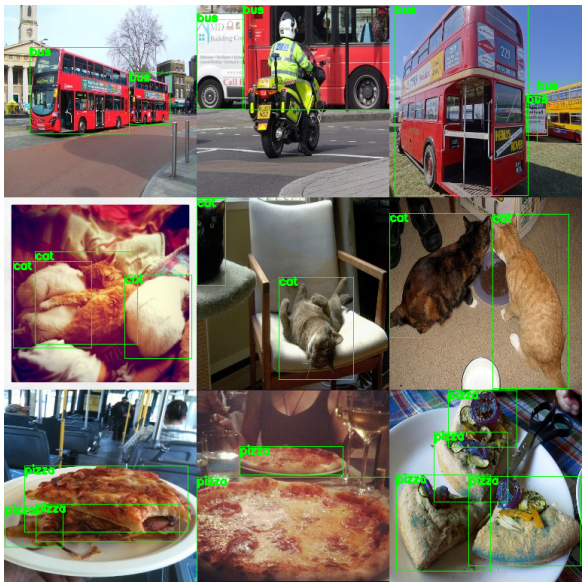


Figure 1: Sample Images from Custom COCO Dataset

3 Building Our Deep Neural Network

The first step in building our deep neural network was to first construct a Skip-Connection Block that will become a foundational building block in our network. The design of the Skip block allows us to construct three different instances of the block that accomplish three unique tasks:

1. Blocks that have the same number of in and out channels. These blocks help add depth to the overall network, but still provide a shortcut between input and output to mitigate the vanishing gradient phenomenon.
2. Blocks that can be used as down samplers, which halve the image size, which doubling the input channels at the output.
3. Blocks that can chained together to form a resolution hierarchy.

Shown blow in Figure 2 is a high level block diagram depicting the logic of our skip block.

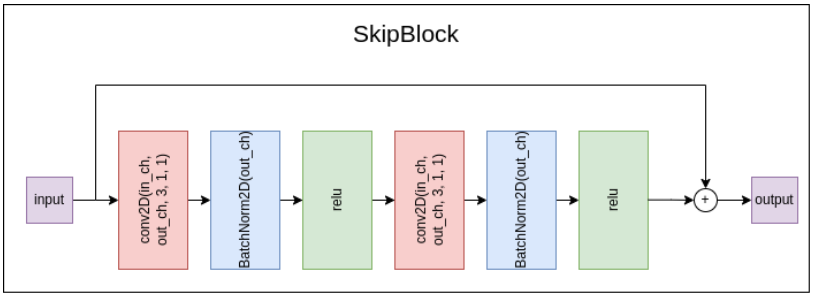


Figure 2: Skip-Connection Block Diagram

After constructing the Skip Block, we synthesised them together, along with other `torch.nn` callable classes, to build our single headed network. Shown below in Figure 3 is a block diagram depicting the logic of our `YoloNet()`.

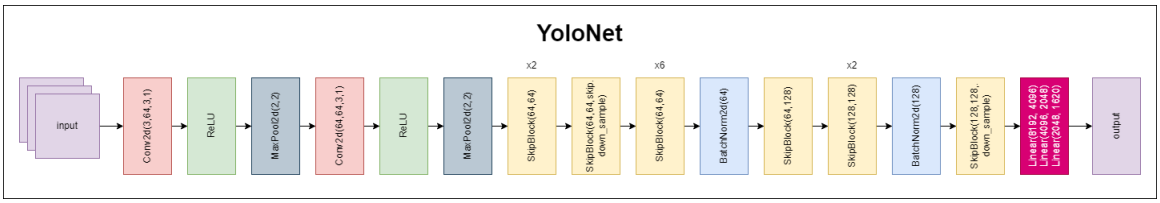


Figure 3: YoloNet Block Diagram

The python implementation for both block diagrams are listed in the appendix under `yolo.pdf`. The number of yolo cells and anchor boxes we used were 144 and 5 respectively. Since each yolo cell has 5 9-element yolo vectors, the output of our network is of shape $B \times 144 \times 5 \times 9$ where B is the batch size. **Furthermore, the number of learnable layers in this network is 122.**

4 Dataloading

To load the training and validation images, we implemented a custom Dataset class and overwrote the `__getitem()__` method to return the image tensor and ground truth yolo tensor. The steps we followed are as follows:

1. We constructed a yolo tensor of shape [144, 5, 9]. Where 144 is the number of yolo cells in a given image, 5 is the number of anchor boxes per cell, and 9 is the length of a particular yolo vector.
2. For each of the bounding boxes in the image, we computed a yolo vector and stored it in the appropriate location based on the cell index and the anchor box index. Any particular yolo vector consists of the following:
    - The first element is a single bit that indicated whether or not the yolo cell and anchor box pair contain an object instance.
    - The second and third elements are denoted as δx and δy. The offset between the center of the assigned yolo cell and the center of the bounding box.
    - The fourth and fifth elements are denoted as bh and bw, the scaled height and width dimensions of the bounding box.
    - The remaining elements are used to indicate the label of object instance assigned to the yolo cell anchor box pair.
3. Since the yolo tensor was initialized to zero, the result after step 2 became our ground truth yolo tensor.

5 Training the Network

To train out method, we constructed a function that iterates over our dataloader that minimizes the three following loss functions:

- Binary Cross Entropy Loss (BCE): Used on the first element of any given yolo vector since it is an an indication of object presence or absense.
- Minimum Least Squared Error Loss (MSE): Used for regressing the four bounding box parameters δx, δy, bh, bw.
- Cross Entropy Loss: Used for the remaining elements in the yolo vector since we are solving a classification problem.

For step size optimization, our model uses the `ADAM` optimizer with $\beta_1 = 0.9$ and $\beta_2 = 0.999$ and an initial learning rate of 1×10^{-3}. We trained our model for 10 epochs.

To accommodate arbitrary batch sizes without having to iterate across the instances in each batch, we performed the following steps:

1. Splice the yolo tensor for all instances where the first element in every yolo vector is non-zero. We use the `torch.nonzero()` method to perfom this step.
2. After, we used these indices to extract the ground truth and predicted bounding box regression parameters, class labels, and object presence.
3. Having all pairs of ground truth v. predicted parameters, we applied the respective criterion on them before summing all of the losses and performing back propagation
4. The result is a training method that only requires two for loops. One to iterate across the epochs, and one to iterate across the dataloader. The ground truth and predicted parameters are extracted in one shot and there is no need to iterate across the batch axis in the yolo tensor.

6 Evaluation of the Network

To evaluate our model's performance on unseen data, we constructed a function that plots both the ground truth and predicted bounding boxes and labels. The ground truth bounding boxes and labels were easy to plot since we simply needed to iterate through the ground truth yolo tensor and plot any yolo vector where the first element was 1. Plotting the predicted bounding boxes and labels required the following steps:

1. First we fed the image tensor to our trained neural network to get the predicted yolo tensor.
2. After, we iterated through all the yolo cells and anchor boxes and examined the contents of each yolo vector. If a particular yolo vector indicated that there was a car, bus, or pizza in that specific cell/anchor box pair, we stored that information into a dictionary where the cell/anchor box pair was the key and the yolo vector was the value.
3. For each of the key/value pairs in the dictionary, we extracted the bounding box and label information and plotted it on the same image for visual comparison.

An important aspect to note about our validation function is that it only accommodates a batch size of 1, since there were no constraints for this task.

7 Results

Shown below in Figures 4, 5, and 6 are the training losses plotted against iterations. Figure 4 shows the cross entropy loss for classification. Figure 5 shows the mean squared error loss for regression of bounding box parameters. Figure 6 shows binary cross entropy loss for object detection in a given cell.

Figure 4: Cross Entropy Loss

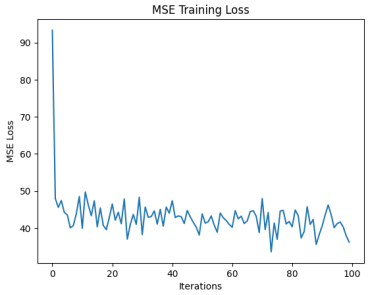


Figure 5: Reg Loss (MSE)

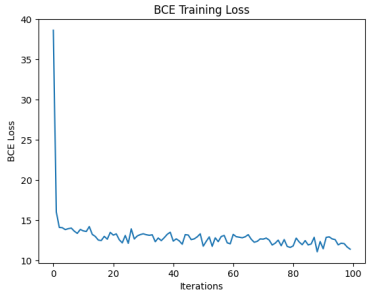


Figure 6: BCE Loss

Shown below in Figure 7 is a 3×2 subplot depicting our model's decent performance at inference time on unseen validation data. The first row corresponds to inferences made on images with buses, the second row corresponds to inferences made on images with cats, and the third row corresponds to inferences made on pizzas. The blue bounding boxes and text are inferred while the green ones are the ground truth. Comparatively, Figure 8 depicts shortcomings of our model. The most common of which were mis-detections of instances in the image. These mis-detections are most likely attributed to our model's lack of robustness to changes in illumination, scale, rotation, etc.

Figure 7: Decent Validation Results

Figure 8: Poor Validation Results

8 Discussion of Results

For the most part, object detection results on the validation set were very positive. Our model performed the best on images where object instances were clearly segregated and were exposed to 'normal' illumination. This is clearly indicated as seen in Figure 7, as all but one of the images shown, depict a clear separation between object instance under relatively normal illumination. Another positive observation noticed was that the anchor boxes did their job in controlling the aspect ratios of the predicted bounding boxes. Although some predicted bounding boxes had a low IOU score compared to their ground truth counterpart, most predicted bounding boxes took the shape of their assigned anchor boxes.

Comparatively, our model struggled very hard on images where either the object instances were not clearly segregated, were occluded by other objects, or if the object was seen under non-normal illumination settings. Misdetection can be clearly seen in the first pizza image from Figure 8. The model does a decent job of identifying the first pizza, but completely misses the second pizza. In the second and third pizzas, there isn't much segregation between the two instances, and thus, our model sees the two instances as a single one. The bus image in Figure 8 shows yet another shortcoming of our model. When the object instances in question are occluded by other objects closer in the foreground, our model has a really hard time distinguishing them.

9 Appendix

What follows on the next page is the source code for replicating the results found in the writing. The same source code can also be found on github here.

Create Custome Multi-Instance Dataset From MS-COCO 2014 Train/Val

Import Statements

In [1]:
```
%matplotlib inline
from pycocotools.coco import COCO
import numpy as np
import matplotlib.pyplot as plt
import pylab
import cv2
from skimage.transform import resize
pylab.rcParams['figure.figsize'] = (8.0, 10.0)
import os
import pickle
```

Initialize COCO API for Instance Annotations

In [2]:
```
train_json = "/mnt/cloudNAS4/jo/coco/annotations/instances_train2014.json"
val_json = "/mnt/cloudNAS4/jo/coco/annotations/instances_val2014.json"
class_list = ['pizza', 'bus', 'cat']

coco_train = COCO(train_json)
coco_val = COCO(val_json)
```

```
loading annotations into memory...
Done (t=12.96s)
creating index...
index created!
loading annotations into memory...
Done (t=6.66s)
creating index...
index created!
```

Get Train and Val IDS For Categories in {'bus', 'cat', 'pizza'}

In [3]:
```
catIDS_train = coco_train.getCatIds(catNms=class_list)
categories_train = coco_train.loadCats(catIDS_train)

catIDS_val = coco_val.getCatIds(catNms=class_list)
categories_val = coco_val.loadCats(catIDS_val)

catID_dict_train = {}
catID_dict_train['bus'] = [6]
catID_dict_train['cat'] = [17]
catID_dict_train['pizza'] = [59]

catID_dict_val = {}
catID_dict_val['bus'] = [6]
```

```
catID_dict_val['cat'] = [17]
catID_dict_val['pizza'] = [59]

print(f'Train Cat Ids: {catID_dict_train}')
print(f'Val Cat Ids: {catID_dict_val}')
```

```
Train Cat Ids: {'bus': [6], 'cat': [17], 'pizza': [59]}
Val Cat Ids: {'bus': [6], 'cat': [17], 'pizza': [59]}
```

Get Image Ids for All Classes in both Train/Val

In [ ]:
```
# image ids for all classes in train set
bus_imgIds_train = coco_train.getImgIds(catIds=catID_dict_train['bus'])
cat_imgIds_train = coco_train.getImgIds(catIds=catID_dict_train['cat'])
pizza_imgIds_train = coco_train.getImgIds(catIds=catID_dict_train['pizza'])

# image ids for all classes in val set
bus_imgIds_val = coco_val.getImgIds(catIds=catID_dict_val['bus'])
cat_imgIds_val = coco_val.getImgIds(catIds=catID_dict_val['cat'])
pizza_imgIds_val = coco_val.getImgIds(catIds=catID_dict_val['pizza'])

print(bus_imgIds_train)
```

def get_img_anns(coco, imgIds, catID, train, cat)

- coco (pycocotools.coco.COCO): specify which coco instance to use (train/val)
- imgIds (list): list of image ids corresponding to a certain class
- catID (list): list of one id, specifying the class
- train (bool): whether data is train or val
- cat (str): category name, used for saving image to correct path

In [5]:
```
train_src_dir = "/mnt/cloudNAS4/jo/coco/train/"
val_src_dir = "/mnt/cloudNAS4/jo/coco/val"

train_dest_dir = "/mnt/cloudNAS4/jo/coco_hw6/train/"
val_dest_dir = "/mnt/cloudNAS4/jo/coco_hw6/val"

def get_images_annotations(coco: COCO, imgIds:list, catID:list, train:bool,
    all_annotations = list()
    for iid in imgIds:

        # img_dict is a dict that stores all info related to img with id iid
        img_dict = coco.loadImgs(iid)[0]

        # Determine the app. src directory
        if train:
            img_path = os.path.join(train_src_dir, img_dict["file_name"])
        else:
            img_path = os.path.join(val_src_dir, img_dict["file_name"])

        # read in the image as a np array
        I = cv2.imread(img_path)
```

```
        # check to make sure image has 3 channels
        if I.shape[2] != 3:
                I = cv2.cvtColor(I,cv2.COLOR_GRAY2RGB)

        # get the annotation ids
        annIDs = coco.getAnnIds(imgIds=img_dict['id'], catIds=catID, iscrowd

        # load the annotations
        anns = coco.loadAnns(annIDs)

        # check for dominant object in image
        if len(anns) == 1:
            continue

        # reshape the image to 256 x 256
        img_h, img_w, _ = I.shape
        I = cv2.resize(I, (256,256), cv2.INTER_AREA)

        # specify the save path for the augmented image and the pickle file
        if train:
            save_path = os.path.join(train_dest_dir, cat, img_dict["file_nam
            pickle_path = os.path.join(train_dest_dir, cat, cat+".pkl")
        else:
            save_path = os.path.join(val_dest_dir, cat, img_dict["file_name"
            pickle_path = os.path.join(val_dest_dir, cat, cat+".pkl")


        single_annotation = {'filename': save_path, 'bbox': list()}

        skip_this_image = False
        for ann in anns:
            if ann["area"] < 64 * 64:
                skip_this_image = True
                single_annotations = None
                break
            [x, y, w, h] = ann['bbox']
            x = x * 256 / img_w
            y = y * 256 / img_h
            w = w * 256 / img_w
            h = h * 256 / img_h

            single_bbox = [x, y, x+w, y+h]
            single_annotation['bbox'].append(single_bbox)

            # I = cv2.rectangle(I, (int(x), int(y)), (int(x+w), int(y+h)), c

        if not skip_this_image:
            all_annotations.append(single_annotation)
            cv2.imwrite(save_path, I)

    with open(pickle_path, 'wb') as f:
        pickle.dump(all_annotations, f)
    return all_annotations
```

Get and Store Train Annotations

In [6]:
```
bus_annotations_train = get_images_annotations(coco=coco_train, imgIds=bus_i
cat_annotations_train = get_images_annotations(coco=coco_train, imgIds=cat_i
pizza_annotations_train = get_images_annotations(coco=coco_train, imgIds=piz
```

Get and Store Val Annotations

In [7]:
```
bus_annotations_val = get_images_annotations(coco=coco_val, imgIds=bus_imgId
cat_annotations_val = get_images_annotations(coco=coco_val, imgIds=cat_imgId
pizza_annotations_val = get_images_annotations(coco=coco_val, imgIds=pizza_i
```

Define a Function For Plotting 3 images w/ anns

- file_dict (list): list of dictionaries where each dict has filename and bbox parameters
- num_images (int): 3 by default

In [1]:
```
import random
import cv2
import numpy as np
def plt_3_imgs(file_dict:list, cat:str, num_images=3)->None:
    all_images = list()
    for _ in range(num_images):
        idx = random.randint(0, len(file_dict)-1)
        entry = file_dict[idx]
        I = cv2.imread(entry['filename'])
        for bbox in entry['bbox']:
            x1 = bbox[0]
            y1 = bbox[1]
            x2 = bbox[2]
            y2 = bbox[3]
            I = cv2.rectangle(I, (int(x1), int(y1)), (int(x2), int(y2)), col
            I = cv2.putText(I, cat, (int(x1), int(y1)+10), cv2.FONT_HERSHEY_
        all_images.append(I)

    return np.hstack((all_images[0], all_images[1], all_images[2]))
```

In [5]:
```
import pickle

bus_annotations_train = pickle.load(open("/mnt/cloudNAS4/jo/coco_hw6/train/b
cat_annotations_train = pickle.load(open("/mnt/cloudNAS4/jo/coco_hw6/train/c
pizza_annotations_train = pickle.load(open("/mnt/cloudNAS4/jo/coco_hw6/train



bus_3_plot = plt_3_imgs(bus_annotations_train, "bus" )
cat_3_plot = plt_3_imgs(cat_annotations_train, "cat")
pizza_3_plot = plt_3_imgs(pizza_annotations_train, "pizza")
```

```
all_class_images = np.vstack((bus_3_plot, cat_3_plot, pizza_3_plot))
cv2.imwrite('/mnt/cloudNAS4/jo/ECE60146/HW06/results/3plot.png', all_class_i
```

Out[5]: True

In [ ]:

Multi-Instance Object Detection via. YOLO

Import Statements

In [1]:
```
import os,os.path
import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision
import torchvision.transforms as tvt
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
import pandas as pd
import seaborn as sns
import cv2
import pickle
from torch.utils.data import DataLoader
device = 'cuda' if torch.cuda.is_available() == True else 'cpu'
print(device)
```

```
cuda
```

Define a Dataset Class

In [2]:
```
class MyDataset(torch.utils.data.Dataset):
    def __init__(self, root):
        super().__init__()
        self.root = root
        self.classes = ['bus', 'cat', 'pizza']
        self.all_images = list()
        self.all_labels = list()
        self.all_bbox = list()
        self.transforms = tvt.Compose([
            tvt.PILToTensor(),
            tvt.ConvertImageDtype(torch.float),
            tvt.Normalize([0.5,0.5,0.5], [0.5,0.5,0.5])
        ])
        self.encoder = {'bus':0, 'cat': 1, 'pizza': 2}
        self.yolo_interval = 20

        self.build_image_label_lists()
        self.num_anchor_boxes = 5 # (height/width) 1/5 1/3 1/1 3/1 5/1

    def build_image_label_lists(self):
        for cat in self.classes:
            pickle_path = os.path.join(self.root, cat, cat+".pkl")
            with open(pickle_path, 'rb') as f:
                all_entries = pickle.load(f)
                for entry in all_entries:
                    self.all_images.append(entry['filename'])
```

```
                    self.all_bbox.append(torch.tensor(entry['bbox'], dtype=t
                    self.all_labels.append(self.encoder[cat])

    def getYoloTensor(self, img, label, bbox):
        num_yolo_cells = (img.shape[1] // self.yolo_interval) * (img.shape[2
        yolo_tensor = torch.zeros(num_yolo_cells, self.num_anchor_boxes, 8)

        # cell shape (20 x 20)
        cell_height = self.yolo_interval
        cell_width = self.yolo_interval

        # num cells in x and y direction
        num_cells_image_width = img.shape[1] // self.yolo_interval
        num_cells_image_height = img.shape[2] // self.yolo_interval

        height_center_bb = torch.zeros(img.shape[1], 1).float()
        width_center_bb = torch.zeros(img.shape[1], 1).float()
        obj_bb_height = torch.zeros(img.shape[1], 1).float()
        obj_bb_width = torch.zeros(img.shape[1], 1).float()

        all_anchor_box_idx = list()
        all_cell_box_idx = list()

        ### idx for object index
        for idx in range(len(bbox)):
            # location of center of gt bbox
            height_center_bb = (bbox[idx, 1] + bbox[idx, 3]) // 2
            width_center_bb = (bbox[idx, 0] + bbox[idx, 2]) // 2

            # gt bbox height and width
            obj_bb_height = bbox[idx, 3] - bbox[idx, 1]
            obj_bb_width = bbox[idx, 2] - bbox[idx, 0]

            # gt bbox belongs to yolo cell (cell_row_idx, cell_col_idx)
            cell_row_idx = (height_center_bb / self.yolo_interval).int()
            cell_col_idx = (width_center_bb / self.yolo_interval).int()
            cell_row_idx = torch.clamp(cell_row_idx, max=num_cells_image_hei
            cell_col_idx = torch.clamp(cell_col_idx, max=num_cells_image_wid

            # compute bh and bw for yolo_vector
            bh = obj_bb_height.float() / self.yolo_interval
            bw = obj_bb_width.float() / self.yolo_interval

            # compute object center
            obj_center_x = (bbox[idx][2].float() + bbox[idx][0].float()) / 2
            obj_center_y = (bbox[idx][3].float() + bbox[idx][1].float()) / 2

            # compute yolo cell center (yolocell_center_j, yolocell_center_i
            yolocell_center_i = cell_row_idx * self.yolo_interval + float(se
            yolocell_center_j = cell_col_idx * self.yolo_interval + float(se

            # compute del_x and del_y for yolo_vector
            del_x = (obj_center_x.float() - yolocell_center_j.float()) / sel
            del_y = (obj_center_y.float() - yolocell_center_i.float()) / sel

            # compute anchor box index
```

```
            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

            # construct yolo_vector
            yolo_vector = torch.FloatTensor([0, del_x.item(), del_y.item(),
            yolo_vector[0] = 1
            yolo_vector[5 + label] = 1

            yolo_cell_index = cell_row_idx.item() * num_cells_image_width +

            # place yolo_vector in correct place, based on yolo_cell_index a
            yolo_tensor[yolo_cell_index, anch_box_index] = yolo_vector

            yolo_tensor_aug = torch.zeros(num_yolo_cells, self.num_anchor_bo
            yolo_tensor_aug[:,:,:-1] = yolo_tensor

            # save anchor box and yolo cell idx
            all_anchor_box_idx.append(anch_box_index)
            all_cell_box_idx.append(yolo_cell_index)

            # print(f'label: {label}')
            # print(f'cell_row_idx: {cell_row_idx}')
            # print(f'cell_col_idx: {cell_col_idx}')
            # print(f'yolocell_center_i: {yolocell_center_i}')
            # print(f'yolocell_center_j: {yolocell_center_j}')
            # print(f'AR: {AR}')
            # print(f'anchor_box_index: {anch_box_index}')
            # print(f'yolo_vector: {yolo_vector}')
            # print(f'yolo_cell_index: {yolo_cell_index}')
            # print("")



        # if not object is present, throw all prob mass into extra 9th eleme
        # located at yolo_tensor[yolo_cell_idx, anchor_box_idx]
        for icx in range(num_yolo_cells):
            for iax in range(self.num_anchor_boxes):
                if yolo_tensor_aug[icx, iax, 0] == 0:
                    yolo_tensor_aug[icx, iax, -1] = 1

        # print(f'yolo_tensor_aug: {yolo_tensor_aug}')
        # print(f'yolo_tensor shape: {yolo_tensor_aug.shape}')

        return yolo_tensor_aug, all_anchor_box_idx, all_cell_box_idx

    def __len__(self):
        assert(len(self.all_images) == len(self.all_labels))
        assert(len(self.all_images) == len(self.all_bbox))
        return len(self.all_images)

    def __getitem__(self,idx):
        img = Image.open(self.all_images[idx])
```

```
        img = self.transforms(img)
        yolo_tensor, anchor_box_idx, cell_box_idx = self.getYoloTensor(img,
        return img, self.all_labels[idx], self.all_bbox[idx], yolo_tensor, a
```

Define a Skip-Connection Block

In [3]:
```
class SkipBlock(nn.Module):
    def __init__(self, in_ch, out_ch, downsample=False, skip_connections=Tru
        super(SkipBlock, self).__init__()
        self.downsample = downsample
        self.skip_connections = skip_connections
        self.in_ch = in_ch
        self.out_ch = out_ch
        self.convo1 = nn.Conv2d(in_ch, out_ch, 3, stride=1, padding=1)
        self.convo2 = nn.Conv2d(in_ch, out_ch, 3, stride=1, padding=1)
        self.bn1 = nn.BatchNorm2d(out_ch)
        self.bn2 = nn.BatchNorm2d(out_ch)
        if downsample:
            self.downsampler = nn.Conv2d(in_ch, out_ch, 1, stride=2)

    def forward(self, x):
        identity = x
        out = self.convo1(x)
        out = self.bn1(out)
        out = torch.nn.functional.relu(out)
        if self.in_ch == self.out_ch:
            out = self.convo2(out)
            out = self.bn2(out)
            out = torch.nn.functional.relu(out)
        if self.downsample:
            out = self.downsampler(out)
            identity = self.downsampler(identity)
        if self.skip_connections:
            if self.in_ch == self.out_ch:
                out = out + identity
            else:
                out = torch.cat((out[:,:self.in_ch,:,:] + identity, out[:, s
        return out
```

Define a Yolo Network

In [4]:
```
class YoloNet(nn.Module):
    def __init__(self, skip_connections=True, depth=8):
        super(YoloNet, self).__init__()
        self.depth = depth // 2
        self.conv1 = nn.Conv2d(3, 64, 3, padding=1)
        self.conv2 = nn.Conv2d(64, 64, 3, padding=1)
        self.pool = nn.MaxPool2d(2, 2)
        self.bn1  = nn.BatchNorm2d(64)
        self.bn2  = nn.BatchNorm2d(128)
        self.bn3  = nn.BatchNorm2d(256)
        self.skip64_arr = nn.ModuleList()
        for i in range(self.depth):
```

```
            self.skip64_arr.append(SkipBlock(64, 64, skip_connections=skip_c
        self.skip64ds = SkipBlock(64, 64, downsample=True, skip_connections=
        self.skip64to128 = SkipBlock(64, 128, skip_connections=skip_connecti
        self.skip128_arr = nn.ModuleList()
        for i in range(self.depth):
            self.skip128_arr.append(SkipBlock(128, 128, skip_connections=ski
        self.skip128ds = SkipBlock(128, 128,downsample=True, skip_connectior
        self.skip128to256 = SkipBlock(128, 256, skip_connections=skip_connec
        self.skip256_arr = nn.ModuleList()
        for i in range(self.depth):
            self.skip256_arr.append(SkipBlock(256, 256, skip_connections=ski
        self.skip256ds = SkipBlock(256,256, downsample=True, skip_connectior
        self.fc_seqn = nn.Sequential(
            nn.Linear(8192, 4096),
            nn.ReLU(inplace=True),
            nn.Linear(4096, 2048),
            nn.ReLU(inplace=True),
            nn.Linear(2048, 1620)
        )

    def forward(self, x):
        x = self.pool(torch.nn.functional.relu(self.conv1(x)))
        x = nn.MaxPool2d(2,2)(torch.nn.functional.relu(self.conv2(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 = x.view(-1, 8192 )
        x = self.fc_seqn(x)
        return x
```

Construct Instances of MyDataSet for Dataloading

In [3]:
```
train_set = MyDataset("/mnt/cloudNAS4/jo/coco_hw6/train/")
val_set = MyDataset("/mnt/cloudNAS4/jo/coco_hw6/val/")

train_loader = DataLoader(train_set, batch_size=4, num_workers=2, shuffle=Tr
val_loader = DataLoader(val_set, batch_size=1, num_workers=2, shuffle=True)
```

Define a Method for Training the Network

In [6]:
```
def train(net, dataloader, epochs, model_save_path, display_when):
    # move network onto device
    net = net.to(device)

    # specify the loss functions
```

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

    # specify ADAM step size optimizer
    optimizer = torch.optim.Adam(net.parameters(), lr=1e-3, betas=(0.9, 0.99

    # store the loss for plotting
    loss_tally_bce = list()
    loss_tally_cross = list()
    loss_tally_mse = list()

    for epoch in range(epochs):
        running_loss_bce = 0.0
        running_loss_mse = 0.0
        running_loss_cross = 0.0

        for iteration, data in enumerate(dataloader):
            img, label, bbox, yolo_tensor, anchor_box, cell_box = data
            img = img.to(device)
            yolo_tensor = yolo_tensor.to(device)
            optimizer.zero_grad()
            output = net(img)

            prediction_aug = output.view(1, 144, 5, 9)
            loss_bce = torch.tensor(0.0, requires_grad=True).float().to(devi
            loss_mse = torch.tensor(0.0, requires_grad=True).float().to(devi
            loss_cross = torch.tensor(0.0, requires_grad=True).float().to(de

            # get all the yolo_vectors where object presence is not zero
            s = nn.Sigmoid()
            object_presence = torch.nonzero(prediction_aug[:,:,:,0])
            bce_loss = criterion1(s(prediction_aug[:,:,:,0]), target_for_pre
            loss_bce = loss_bce + bce_loss

            # reg for bbox parameters
            pred_reg_vec = prediction_aug[object_presence[:,0], object_prese
            target_reg_vec = yolo_tensor[object_presence[:,0], object_presen
            reg_loss = criterion2(pred_reg_vec, target_reg_vec)
            loss_mse = loss_mse + reg_loss

            # cross entropy loss for class labels
            probs_vector = prediction_aug[object_presence[:,0], object_prese
            target = torch.argmax(yolo_tensor[object_presence[:,0], object_p
            class_labeling_loss = criterion3(probs_vector, target)
            loss_cross = loss_cross + class_labeling_loss

            # For batch size = 1
            # for icx in range(144):
            #     for iax in range(5):
            #         pred_yolo_vector = prediction_aug[0, icx, iax]
            #         target_yolo_vector = yolo_tensor[0, icx, iax]

            #         # estimate the presence/absence of object using BCE Lo
            #         object_presence = nn.Sigmoid()(torch.unsqueeze(pred_yo
            #         target_for_prediction = torch.unsqueeze(target_yolo_ve
```

```
            #         bce_loss = criterion1(object_presence, target_for_pred
            #         loss_bce = loss_bce + bce_loss

            #         # regression for bbox parameters
            #         pred_reg_vec = torch.unsqueeze(pred_yolo_vector[1:5],
            #         target_reg_vec = torch.unsqueeze(target_yolo_vector[1:
            #         reg_loss = criterion2(pred_reg_vec, target_reg_vec)
            #         loss_mse = loss_mse + reg_loss

            #         # cross entropy for classification
            #         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)
            #         loss_cross = loss_cross + class_labeling_loss

            loss_cross.backward(retain_graph=True)
            loss_mse.backward(retain_graph=True)
            loss_bce.backward(retain_graph=True)

            optimizer.step()

            running_loss_bce += loss_bce.item()
            running_loss_mse += loss_mse.item()
            running_loss_cross += loss_cross.item()


            if iteration % display_when == display_when - 1:
                avg_loss_bce = running_loss_bce / float(display_when)
                avg_loss_mse = running_loss_mse / float(display_when)
                avg_loss_cross = running_loss_cross / float(display_when)

                print(f'Epoch: {epoch+1}, Iteration: {iteration+1}, BCE Loss

                loss_tally_bce.append(avg_loss_bce)
                loss_tally_cross.append(avg_loss_cross)
                loss_tally_mse.append(avg_loss_mse)

                running_loss_bce = 0.0
                running_loss_mse = 0.0
                running_loss_cross = 0.0
    torch.save(net.state_dict(), model_save_path + "Epoch" + str(epoch+1) +
    return loss_tally_bce, loss_tally_cross, loss_tally_mse
```

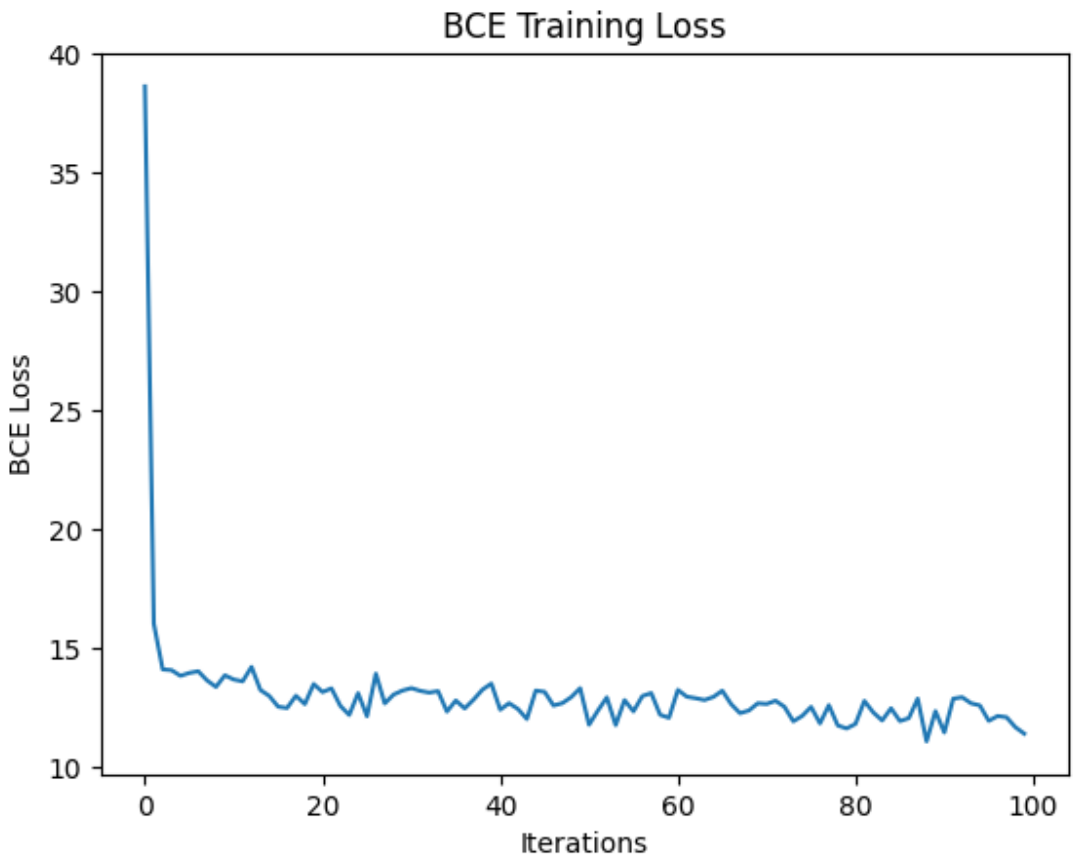
In [ ]:
```
net = YoloNet()
bce_loss, cross_loss, mse_loss = train(net, train_loader, 10, "/mnt/cloudNAS
```

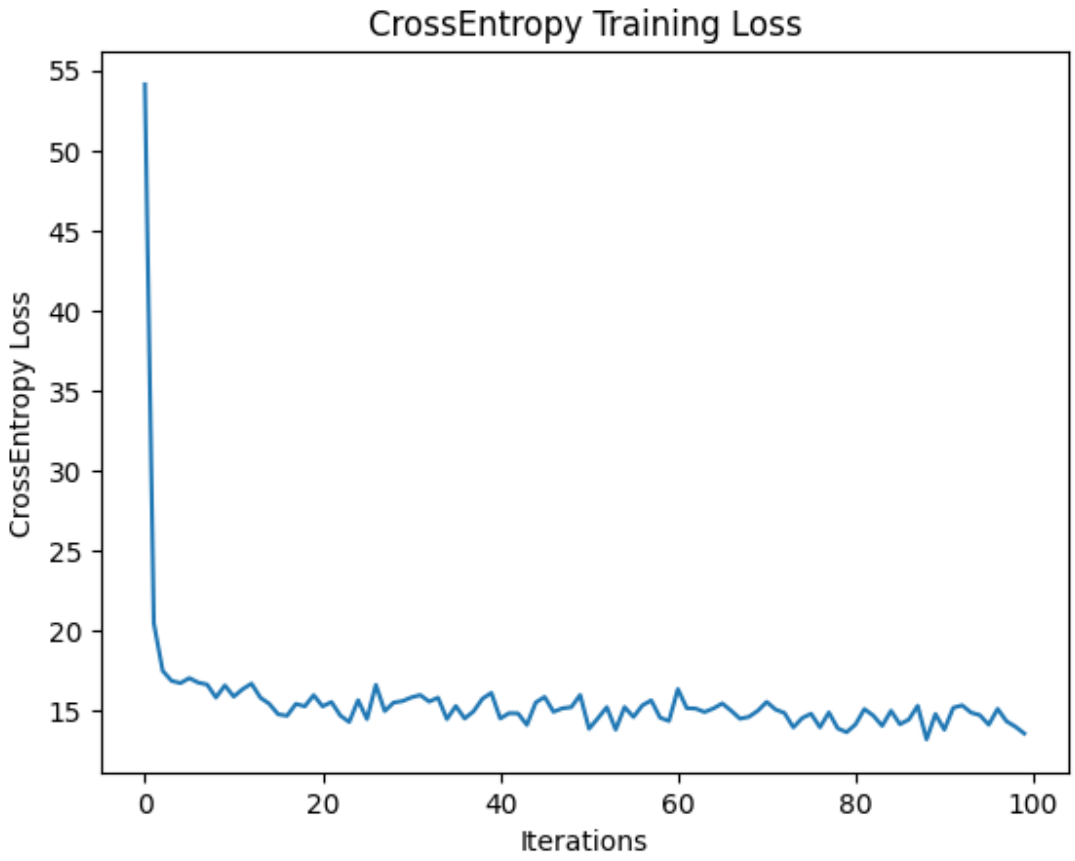
Plot the Training Loss

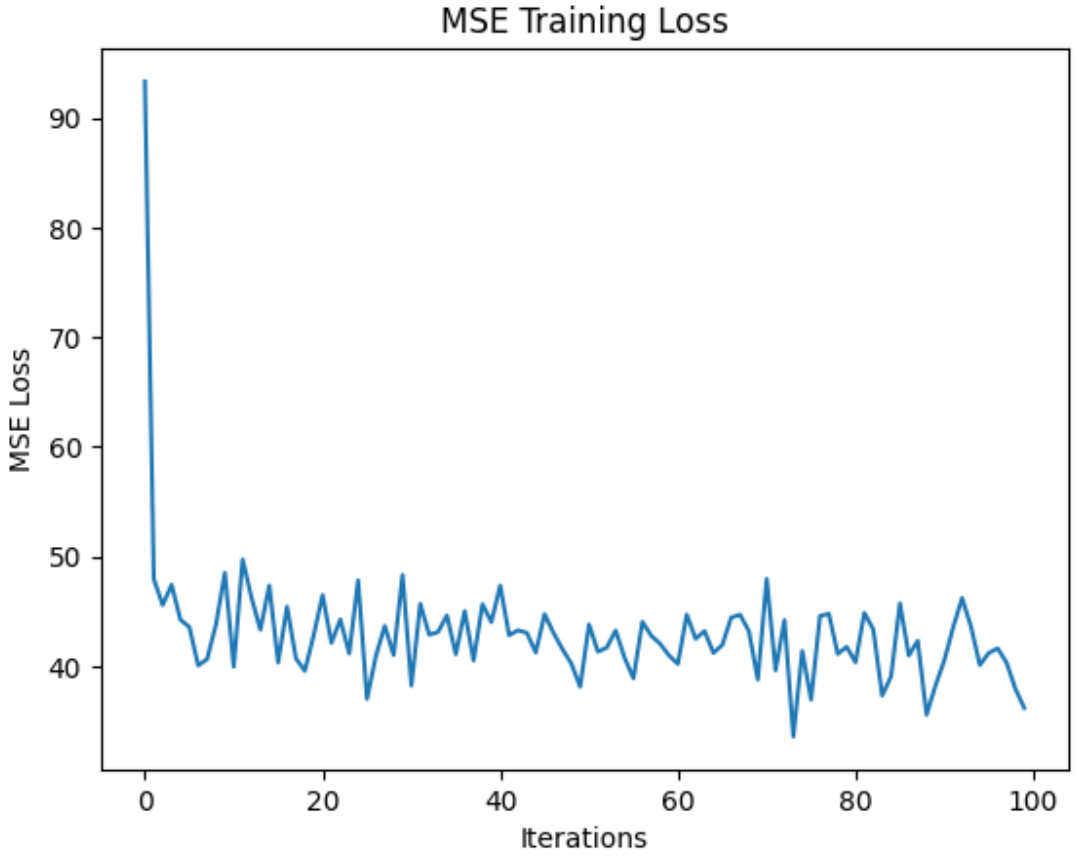
In [8]:
```
plt.figure()
plt.plot(bce_loss)
plt.title('BCE Training Loss')
plt.xlabel('Iterations')
```

```
plt.ylabel('BCE Loss')

plt.figure()
plt.plot(cross_loss)
plt.title('CrossEntropy Training Loss')
plt.xlabel('Iterations')
plt.ylabel('CrossEntropy Loss')

plt.figure()
plt.plot(mse_loss)
plt.title('MSE Training Loss')
plt.xlabel('Iterations')
plt.ylabel('MSE Loss')
```

Out[8]: Text(0, 0.5, 'MSE Loss')




CrossEntropy Training Loss
CrossEntropy Loss
55
50
45
40
35
30
25
20
15
0
20
40
60
80
100
Iterations


MSE Training Loss
MSE Loss
90
80
70
60
50
40
0
20
40
60
80
100
Iterations

In [4]:
```
from copy import deepcopy
def single_instance_validation(net, data, model_save_path):
    net.load_state_dict(torch.load(model_save_path))
    net.eval()
    net.to(device)


    img, label, bbox, yolo_tensor, anchor_box, cell_box = data

    # unsqueeze img tensor to be correct size for model eval
    img_2_net = torch.unsqueeze(deepcopy(img), dim=0).to(device)

    # send img tensor to net to get pred yolo vector
    pred_yolo_tensor = net(img_2_net)
    pred_yolo_tensor = pred_yolo_tensor.view(1, 144, 5, 9)
    yolo_tensor = torch.unsqueeze(yolo_tensor, dim=0)

    pred_dict = {}

    for icx in range(144):
        for iax in range(5):
            pred_yolo_vector = pred_yolo_tensor[0, icx, iax]
            target_yolo_vector = yolo_tensor[0, icx, iax]

            # check object presense
            # object_presence = nn.Sigmoid()(torch.unsqueeze(pred_yolo_vecto
            # if object_presence.item() > 0.05:
            #     pred_dict[(icx, iax)] = pred_yolo_vector

            s = nn.Softmax(dim=0)
            target = torch.argmax(s(pred_yolo_vector[5:]))


            if target.item() != 3:
                pred_dict[(icx, iax)] = pred_yolo_vector

    img = img / 2 + 0.5
    img = img.numpy()
    img = np.transpose(img, (1,2,0))


    # plotting detected instances
    for key in pred_dict:
        cell_idx, anchor_box_idx = key
        yolo_vector = pred_dict[key]
        object_presence, del_x, del_y, bh, bw, bus, cat, pizza, nothing = yo

        label = torch.argmax(s([bus, cat, pizza, nothing]))

        if label == 0: label = 'bus'
        if label == 1: label = 'cat'
        if label == 2: label = 'pizza'

        cell_col_idx = cell_idx % 12
        cell_row_idx = cell_idx // 12
```

```
        yolocell_center_i = cell_row_idx * 20 + float(20) / 2.0
        yolocell_center_j = cell_col_idx * 20 + float(20) / 2.0

        obj_center_x = del_x * 20 + yolocell_center_j
        obj_center_y = del_y * 20 + yolo_cell_center_i

        bh = bh * 20
        bw = bw * 20

        x1 = int(obj_center_x - (0.5 * bw))
        y1 = int(obj_center_y - (0.5 * bh))

        x2 = int(obj_center_x + (0.5 * bw))
        y2 = int(obj_center_y + (0.5 * bh))

        img = cv2.rectangle(img,  (x1, y1), (x2, y2), color=(0, 0, 255), thi
        img = cv2.putText(img, label, (x1, y1-10), cv2.FONT_HERSHEY_SIMPLEX,

    # plot gt bbox
    for i in range(len(bbox)):
        x1 = int(bbox[i,0])
        y1 = int(bbox[i,1])
        x2 = int(bbox[i,2])
        y2 = int(bbox[i,3])

        img = cv2.rectangle(img,  (x1, y1), (x2, y2), color=(0, 255, 0), thi
    return img
```

In [ ]:

```
net = YoloNet()
data = train_set[60]
single_instance_validation(net, data,"/mnt/cloudNAS4/jo/ECE60146/HW06/models
```