HW 6: YOLO

David E. Farache, Email ID: dfarache@purdue.edu

March 28, 2023

1 Introduction

There are different methods for multi-instance processing of images and there boundry boxes, including R-CNN, SSD, and YOLO. This work focuses on "You Only Look Once" or the YOLO method, into a convolutional neural network (CNN). This work should implement skip-blocks created from the previous homework and using Binary Cross-Entropy (BCE) loss , Mean Squared Error (MSE) loss, and Cross-Entropy (CE) loss for training.

2 Methodology

COCO was utilized to create an initial training set via the pycocotools.coco function, which would reduce to 6890 for training and 3494 for validation. The data-loader was based on HW4 with the addition of encoding based on the category and transform the image from a PIL image to a normalized tensor.

YOLO method functions by dividing the image into a grid system and the prediction of the boundary box and the class-label is designated to the grid-cell whereby the center of the image is. This enables for single network to be trained for multiple labels as the seperation is made with the grid-cells and anchor boxes, instead of multiple networks as R-CNN would require. For loss calculation, the existence of the image within the cell is done using BCE, classification is measured using CE, and refinement and decision of the bounding box is based upon the MSE regression loss, using the equation seen below.

$$BCE = -(y \log(p) + (1-y)\log(1-p)) \tag{1}$$

$$MSE = \sum_{i=1}^{D} (x_i - y_i)^2 \tag{2}$$

$$Cross - Entropy = -\sum_{c=1}^{M} y_{o,c} \log(p_{o,c}) \tag{3}$$

3 Task 1: COCO Data-set Preparation

For task 1, the COCO data-set was used from train2014 and val2014 in order to reduce and create a training and validation for categories: pizza, cat, and bus. Images were selected based on whether there was a at least one dominant label with area greater than 64 x 64. Each image was turned in RGB and resized from their original height to 256 x 256. All bounding boxes of the images were

re-scaled and there accompanying category saved to a list. A data-frame of filename, category list, and boundry box list, were saved for efficient processing.

```
# %% [markdown]
# # Library

# %%
%matplotlib inline
from pycocotools.coco import COCO

import numpy as np
import skimage.io as io
import skimage
import cv2
import pandas as pd

import matplotlib.pyplot as plt
import pylab
import random
from PIL import Image
pylab.rcParams['figure.figsize'] = (8.0, 10.0)

# %% [markdown]
# # Display Images

# %%
inverse_categories = {6: "bus", 17: "cat", 59: "pizza"}

# %%
# Based from skeleton code given
def display_random_image_with_bbox(image_path, bboxs, cat_set):
    file = image_path
    image = Image.open(file)

    print(bboxs)
    print(cat_set)
    for bbox, cat in zip(bboxs, cat_set):
        [x, y, w, h] = bbox
        fig, ax = plt.subplots(1,1)
        image = np.uint8(image)
        image = cv2.rectangle(image, (int(x), int(y)), (int(x + w), int(y + h)),
    color=(36, 255, 12), thickness=2)
        image = cv2.putText(image, inverse_categories[cat], (int(x), int(y - 10)
   ), fontFace=cv2.FONT_HERSHEY_SIMPLEX, fontScale=0.8, color=(36, 255, 12),
   thickness=2)

        ax.imshow(image)
        ax.set_axis_off()
    plt.axis("tight")
    plt.show()

    fig, axs = plt.subplots(nrows=3, ncols=3, figsize=(10,10), dpi = 150)
    axs = axs.flatten()
    axs_count = 0

# %%
def display_images(saveForPlotting):
```

```
    fig, ax = plt.subplots(3, 3)
    row, col = 0, 0

    #print(len(saveForPlotting))
    for arr in saveForPlotting:
        file = arr[0]
        image = Image.open(file)
        print(arr)
        for cat, bbox in zip(arr[2], arr[1]):
            [x, y, w, h] = bbox
            image = np.uint8(image)
            image = cv2.rectangle(image, (int(x), int(y)), (int(x + w), int(y + h)), color=(36, 255, 12), thickness=2)
            image = cv2.putText(image, inverse_categories[cat], (int(x), int(y - 10)), fontFace=cv2.FONT_HERSHEY_SIMPLEX, fontScale=0.8, color=(36, 255, 12), thickness=2)
            ax[row, col].imshow(image)

        # Increment through row
        col += 1

        if col == 3:
            col = 0
            row += 1

    fig, axs = plt.subplots(nrows=3, ncols=3, figsize=(10,10), dpi = 150)
    plt.show()
    axs = axs.flatten()

# %% [markdown]
# # Adjust and Change Images

# %%
def resizeBBOX(bbox, startWidth, startHeight, resize):
        x_scale = resize / startWidth
        y_scale = resize / startHeight
        bboxResize = np.zeros(4)

        bboxResize[0] = int(x_scale * bbox[0])
        bboxResize[1] = int(y_scale * bbox[1])
        bboxResize[2] = int(x_scale * bbox[2])
        bboxResize[3] = int(y_scale * bbox[3])

        return bboxResize.tolist()

# %%
def resizeAndRepairImage(start_image_path, new_image_path, filename, resize):
        img = Image.open(start_image_path + '/' + filename)
        width, height = img.size

        image = img.convert(mode="RGB")

        img = image.resize((resize, resize), Image.BOX)
        img.save(new_image_path + '/' + filename)

# %%
def saveAsDataFrame(ids, cats, filepaths, x1, y1, width, height, type):
```

```
    # Containers for data to go in training label csv
    columns = ["id", "category", "filepath", "x1", "y1", "width", "height"]
    dataFrame = pd.DataFrame(columns=columns)
    dataFrame = dataFrame.astype('object')

    dataFrame["id"] = np.array(ids)
    dataFrame["category"] = cats
    dataFrame["filepath"] = filepaths
    dataFrame["x1"]  = np.array(x1)
    dataFrame["y1"]  = y1
    dataFrame["width"]  = width
    dataFrame["height"]  = height

    dataFrame.to_csv("{0}_labels.csv".format(type))

    return dataFrame


# %% [markdown]
# # Choose Images Within Parameters

# %%
# Get Random Images from set
def ImageSelection(start_image_path, new_image_path, cocoObj, class_list, type, spotSave):
    # Save File Location List
    saveImportant = []
    saveForPlotting = []

    # Save image info
    ids = []
    cats = []
    filepaths = []
    x1 = []
    y1 = []
    widthFrame = []
    heightFrame = []
    for cat in class_list:
        # get all images containing given categories
        catIds = cocoObj.getCatIds(catNms=[cat]) # Get ids from annotations
        imgIds = cocoObj.getImgIds(catIds=catIds ) # Load images ids of chosen annotations ids
        img = cocoObj.loadImgs(ids=imgIds) # Get images
        numPlots = 0 # Make X number plots

        #Loop per image
        for idx, images  in enumerate(img):
            annIds = cocoObj.getAnnIds(imgIds=images['id'], catIds=catIds, iscrowd=False) # Get dictionary value
            anns = cocoObj.loadAnns(annIds) # Get annotations

            # Saving List
            id_set = []
            cat_set = []
            x1_set = []
            y1_set = []
            widthFrame_set = []
```

```
            heightFrame_set = []
            bbox_set = []

            forGroundImge = 0
            for jdx, ann in enumerate(anns):

                width, height = images['width'], images['height']

                if ann['area'] > 64 * 64: #check if means parameter of dominate obj
                    # Resize Images
                    bboxResize = resizeBBOX(ann['bbox'], int(width), int(height), resize=256) #Adjust Box

                    # Append info
                    id_set.append(ann['id'])
                    cat_set.append(ann['category_id'])
                    x1_set.append(bboxResize[0])
                    y1_set.append(bboxResize[1])
                    widthFrame_set.append(bboxResize[2])
                    heightFrame_set.append(bboxResize[3])
                    bbox_set.append(bboxResize)

                    forGroundImge = 1

            if forGroundImge == 1:
                resizeAndRepairImage(start_image_path, new_image_path, images['file_name'], resize=256) # Save Adjust Image

                cats.append(cat_set)
                filepaths.append("{0}/{1}".format(spotSave,images['file_name']))
                ids.append(id_set)
                x1.append(x1_set)
                y1.append(y1_set)
                widthFrame.append(widthFrame_set)
                heightFrame.append(heightFrame_set)

                if numPlots < 3 and len(cat_set) > 1:
                    #display_random_image_with_bbox("{0}/{1}".format(spotSave, images['file_name']), bbox_set, cat_set)
                    saveForPlotting.append(["{0}/{1}".format(spotSave,images['file_name']), bbox_set, cat_set])
                    numPlots += 1

    display_images(saveForPlotting)
    dataFrame = saveAsDataFrame(ids, cats, filepaths, x1, y1, widthFrame, heightFrame, type)
    return dataFrame

# %%
# Input
train_json = '/Users/davidfarache/Documents/ECE60146/HW6/annotations/instances_train2014.json'
val_json = '/Users/davidfarache/Documents/ECE60146/HW6/annotations/instances_val2014.json'

train_path = '/Users/davidfarache/Documents/ECE60146/HW6/train2014'
```

```
train_data_path = '/Users/davidfarache/Documents/ECE60146/HW6/trainingData'

val_path = '/Users/davidfarache/Documents/ECE60146/HW6/val2014'
val_data_path = '/Users/davidfarache/Documents/ECE60146/HW6/valData'

trainSaveSpot = 'trainingData'
valSaveSpot = 'valData'

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

# %%
cocoTrain = COCO(train_json)
cocoVal = COCO(val_json)

trainDataFrame = ImageSelection(train_path, train_data_path, cocoTrain,
    class_list, 'train', trainSaveSpot)
valDataFrame = ImageSelection(val_path, val_data_path, cocoVal, class_list, 'val
    ', valSaveSpot)

print(len(trainDataFrame))
print(len(valDataFrame))

# %%
print(trainDataFrame.columns)
print(valDataFrame.columns)

# %%
#print(trainDataFrame['category'])
saving_set = []
for i, val in enumerate(trainDataFrame['category']):
    if val[0] not in saving_set:
        print(trainDataFrame['filepath'].iloc[i])
        saving_set.append(val[0])
        print(val[0])
```

Listing 1: *Creating COCO data-set and plotting*

A sample of the images select has been placed below, which can bee see to have shrunk and had their boundry box rescaled within them:

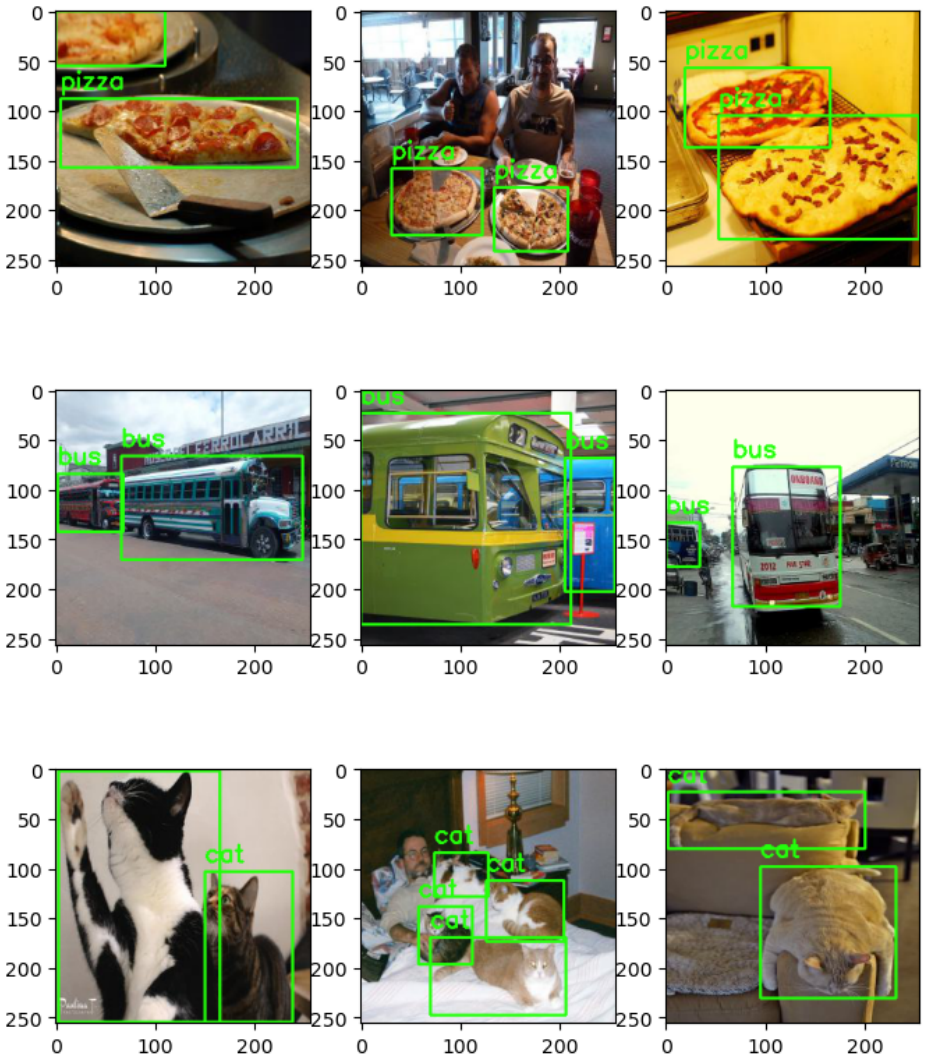


Figure 1: *Selection of Training Set*

4 Task 2: Create YOLO Vector

The following code has the data set that stores the boundary boxes, normalized images, and image labels. The code below is what is required to create the data-loader which includes the YOLO tensor. The YOLO tensor code takes in the boundary boxes, finds the center of the x,y cord for the boundary box image, gets the height and width of the box, and the difference of distance between a YOLO cell center and its own to establish which is responsible. The aspect ratio is taken to set which anchor box is saved and all information is saved as a tensor of [1, δ_x, δ_y, h, w, n-categories for one hot encoding (in this case 3)]. In this case the image grid was selected to be 6x6 meaning each must have 42x42 pixels. The aspect ratios calculated are those given in the lecture: 1/5, 1/3, 1/1, 3/1, and 5/1.

```
# Import Libraries
import numpy as np
import torch
import torchvision.transforms as tvt
import torch.utils.data
import torch.nn as nn
```

```
import torch.nn.functional as F
import matplotlib.pyplot as plt
from PIL import Image
import os
import seaborn as sns
from torchvision.ops import box_convert
import pandas as pd
import cv2
import json
from torchinfo import summary
from copy import deepcopy

# GLOBAL VARIABLES
device = 'cuda'
device = torch.device(device)

# For DataLoader
root = '/scratch/gilbreth/dfarache/ece60146/David/HW6/'
class_list = ["bus", "cat", "pizza"]
inverse_categories = {6: "bus", 17: "cat", 59: "pizza"}

# Global YOLO Values (had to seperate due to no longer sharing training and yolo vecotr in same function)
num_anchor_boxes = 5 # (height/width)   1/5  1/3  1/1  3/1  5/1  aspect ratios
max_num_objects = len(class_list)

yolo_vector_size = 8 # []
yolo_interval = 42 # Each cell is 42x42 pixels

num_cells_image_width = 256 // yolo_interval
num_cells_image_height = 256 // yolo_interval

num_yolo_cells = num_cells_image_width * num_cells_image_height # Create a grid of cells overlaying the image

# YOLO Tensor

# Based on DLStudio run_code_for_training_multi_instance_detection function and homework example
def createYoloTensor(bboxs, labels, num_images_in_batch=1):
    yolo_tensor = torch.zeros(num_yolo_cells, num_anchor_boxes, yolo_vector_size)

    # Create empty torch tensors
    height_center_bb = torch.zeros(num_images_in_batch, 1).float()
    width_center_bb = torch.zeros(num_images_in_batch, 1).float()
    object_bb_height = torch.zeros(num_images_in_batch, 1).float()
    object_bb_width = torch.zeros(num_images_in_batch, 1).float()

    numericDict = {6: 0, 17: 1, 59: 2} # Swap id to index for one-hot encoding

    # i is index of object in the foreground
    for i in range(max_num_objects):
        # remove .float
        y_cord_center = (bboxs[i, 1] + bboxs[i, 3] / 2.0).int() # Get y-coordinate object center from y1
```

```
58      x_cord_center = (bboxs[i, 0] + bboxs[i, 2] / 2.0).int() # Get x-
    coordinate object center from x1
59
60      object_bb_height = (bboxs[i, 3] - bboxs[i, 1]).float() # Height bounding
     box
61      object_bb_width = (bboxs[i, 2] - bboxs[i, 0]).float() # Width bounding
    box
62
63      if (object_bb_height < 4.0) or (object_bb_width < 4.0): continue
64
65      # Get the cell row and column index that corresponds to the center of
    the bounding box
66      cell_row_i = torch.clamp((y_cord_center / yolo_interval).int(), max=
    num_cells_image_height - 1)
67      cell_col_i = torch.clamp((x_cord_center / yolo_interval).int(), max=
    num_cells_image_width - 1)
68
69      # Get the height of the bounding box divided by the actual height of the
     cell
70      bheight = y_cord_center / yolo_interval
71      bwidth = x_cord_center / yolo_interval
72
73      # Swap from x,y system to i,j coordinate
74      cell_center_i = cell_row_i * yolo_interval + float(yolo_interval) / 2.0
75      cell_center_j = cell_col_i * yolo_interval + float(yolo_interval) / 2.0
76
77      # Compute del_x and del_y
78      del_x = (x_cord_center.float() - cell_center_j.float()) / yolo_interval
79      del_y = (y_cord_center.float() - cell_center_i.float()) / yolo_interval
80
81      # Get the class label
82      class_label_of_object = int(labels[i].item())
83      if class_label_of_object == 31: continue  # Disregard labels with class
    label 31
84
85      # Get Aspect Ratio
86      aspect_ratio = object_bb_height / object_bb_width
87      anchor_box_idx = 0
88      if aspect_ratio <= 0.2:                 anchor_box_idx = 0
                                        ## (45)
89      if 0.2 < aspect_ratio <= 0.5:           anchor_box_idx = 1
                                        ## (46)
90      if 0.5 < aspect_ratio <= 1.5:           anchor_box_idx = 2
                                        ## (47)
91      if 1.5 < aspect_ratio <= 4.0:           anchor_box_idx = 3
                                        ## (48)
92      if aspect_ratio > 4.0:                  anchor_box_idx = 4
93
94      # Create the yolo vector
95      # Vector [exsit, x1, y1, height, width, n-encoding]
96      yolo_vector = torch.FloatTensor([1, del_x.item(), del_y.item(), bheight.
    item(), bwidth.item(), 0, 0, 0])
97
98      yolo_vector[5 + numericDict[class_label_of_object]] = 1 # One-hot
    encoding
99
100     # Assign to yolo tensor
```

```
        yolo_cell_index = cell_row_i.item() * num_cells_image_width + cell_col_i.item()

        yolo_tensor[yolo_cell_index, anchor_box_idx] = yolo_vector  # place into proper index

    # Create an augmented yolo tensor
    yolo_tensor_aug = torch.zeros(num_yolo_cells, num_anchor_boxes, yolo_vector_size + 1).float()
    yolo_tensor_aug[:,:,:-1] = yolo_tensor

    # If not exist throw
    for icx in range(num_yolo_cells):
        for iax in range(num_anchor_boxes):
            if(yolo_tensor_aug[icx, iax, 0] == 0):
                yolo_tensor_aug[icx, iax, -1] = 1

    return yolo_tensor_aug


# Create DataLoader Code

def ImageProcessing(images, root):
    # Get dir
    image_dir = images["filepath"]

    # Get inputs
    image = Image.open(root + image_dir)

    # Normalize image
    toTensor = tvt.ToTensor()(image)
    toNormalize = tvt.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))(toTensor)
    return toNormalize


# Generate Datasets
class MyDataset(torch.utils.data.Dataset):
    def __init__(self, imagesDataFrame, root, transform=None):
        super().__init__()
        self.imagesDataFrame = imagesDataFrame
        self.root = root # Directory for folder images

    def bboxAndLabels(self, imageInfo):
        # Split bbox into parts
        x1 = imageInfo["x1"]
        y1 = imageInfo["y1"]
        width = imageInfo["width"]
        height = imageInfo["height"]

        # Get Category
        category = imageInfo["category"]

        labels = torch.zeros(max_num_objects, dtype=torch.uint8) + 31
        bboxs = torch.zeros(max_num_objects, 4, dtype=torch.uint8)

        # Exract info from dataFrame
        x1_cat = x1[1:-1].split(',')
```

```
        y1_cat = y1[1:-1].split(',')
        width_cat = width[1:-1].split(',')
        height_cat = height[1:-1].split(',')
        labelList = category[1:-1].split(',')

        #print(x1_cat, y1_cat, width_cat, height_cat, labelList)
        for j, cat in enumerate(labelList):
            create_box = [float(x1_cat[j]), float(y1_cat[j]), float(width_cat[j
    ]), float(height_cat[j])]

            if (j < max_num_objects):
                bboxs[j] = torch.tensor(create_box, dtype=torch.float)
                labels[j] = int(cat)

        return labels, bboxs

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

    def __getitem__(self, i):
        imageInfo = self.imagesDataFrame
        imageInfo = imageInfo.iloc[i]
        normalImage = ImageProcessing(imageInfo, self.root)

        # Make YOLO Tensor for processing
        labels, bboxs = self.bboxAndLabels(imageInfo)
        yoloTensor = createYoloTensor(bboxs, labels)

        return normalImage, bboxs, labels, yoloTensor
```

Listing 2: *Setup YOLO Tensor and DatLoader*

5 Task 3: Create Network and Train

The code below is based on that from hw 5, which grants 102 layers within the network.

```
# Based on DLStudio SkipBlock Code
# Based off following https://blog.paperspace.com/writing-resnet-from-scratch-in
    -pytorch/
class Block(nn.Module):
    def __init__(self, in_ch, out_ch, downsample=False, skip_connections=True):
        super(Block, self).__init__()
        self.downsample = downsample
        self.in_ch = in_ch
        self.out_ch = out_ch

        self.skip_connections = skip_connections

        self.conv1 = nn.Sequential(
                     nn.Conv2d(in_ch, out_ch, kernel_size = 3, stride = 1,
    padding = 1),
                     nn.BatchNorm2d(out_ch),
                     nn.ReLU())
        self.conv2 = nn.Sequential(
                         nn.Conv2d(out_ch, out_ch, kernel_size = 3, stride = 1,
    padding = 1),
```

```
                        nn.BatchNorm2d(out_ch))
        self.relu = nn.ReLU()

        if downsample:
            self.downsampler = nn.Conv2d(in_ch, out_ch, 1, stride=2)

    def forward(self, x):
        residual = x
        out = self.conv1(x)

        if self.in_ch == self.out_ch:
            out = self.conv1(out)

        if self.downsample:
            out = self.downsampler(out)
            residual = self.downsampler(residual)

        if self.skip_connections:
            if self.in_ch == self.out_ch:
                out = out + residual
            else:
                # Assuming equivalent dimensions which this dataset fits
                firstSection = out[:,:self.in_ch,:,:]
                secondSection = out[:,self.in_ch:,:,:]

                out = torch.cat((firstSection + residual, secondSection +
    residual), dim=1)

        return out

# CNN model
# Based on DLStudio LOADnet2 && notes from class
class NetForYolo(nn.Module):
    # Inspired by Professor Kak's NetForYolo class
    def __init__(self, skip_connections=True, depth=8):
        super(NetForYolo, self).__init__()
        self.skip_connections = skip_connections
        self.depth = depth // 2

        self.conv1 = nn.Conv2d(in_channels=3, out_channels=64, kernel_size=3,
    padding=1)
        self.conv2 = nn.Conv2d(in_channels=64, out_channels=64, kernel_size=3,
    padding=1)

        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)

        self.bn1 = nn.BatchNorm2d(num_features=64)
        self.bn2 = nn.BatchNorm2d(num_features=128)

        self.skip64_arr = nn.ModuleList()
        for idx in range(self.depth):
            self.skip64_arr.append(Block(in_ch=64, out_ch=64, skip_connections=
    self.skip_connections))

        self.skip64ds = Block(in_ch=64, out_ch=64, downsample=True,
    skip_connections=self.skip_connections)
```

```
        self.skip64to128 = Block(in_ch=64, out_ch=128, skip_connections=self.
    skip_connections)

        self.skip128_arr = nn.ModuleList()
        for idx in range(self.depth):
            self.skip128_arr.append(Block(in_ch=128, out_ch=128,
    skip_connections=self.skip_connections))

        self.skip128ds = Block(in_ch=128, out_ch=128, downsample=True,
    skip_connections=self.skip_connections)
        self.skip128to256 = Block(in_ch=128, out_ch=256, skip_connections=self.
    skip_connections)
        self.skip256_arr = nn.ModuleList()
        for idx in range(self.depth):
            self.skip256_arr.append(Block(in_ch=256, out_ch=256,
    skip_connections=self.skip_connections))

        self.skip256ds = Block(in_ch=256, out_ch=256, downsample=True,
    skip_connections=self.skip_connections)

        self.fc_seqn = nn.Sequential(
            nn.Linear(in_features=128*16*16, out_features=6*6*5*9),
                        nn.ReLU(inplace=True),
#                         nn.Linear(in_features=4096, out_features=2048),
#                         nn.ReLU(inplace=True),
                        nn.Linear(in_features=6*6*5*9, out_features=6*6*5*9) # 6
    x6 grid
                        )
     def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.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, 128*16*16)
        x = self.fc_seqn(x)
        return x
```

Listing 3: *Make Network*

For training, the criterion of BCE, MSE, and CE was utilized. The inputs for the model were batch_size 64, learning rate 1e-5, and trained for 8 epochs. Each YOLO vector is iterated through with values extracted for training and a running average of the losses taken every 25 values in the batch.

```
# Based on DLStudio run_code_for_training_multi_instance_detection function and
    notes in class
def train(net, trainLoader, epochs, lr, betas):
    print("Training")
    optimizer = torch.optim.Adam(net.parameters(), lr=lr, betas=betas)
```

```
5
6       # Criteria
7       criterion1 = nn.BCELoss() # If object in image present
8       criterion2 = nn.MSELoss() # Regression for bounding box
9       criterion3 = nn.CrossEntropyLoss() # One hot encoding of label
10
11      net = net.to(device)
12      BCELossIter = []
13      MSELossIter = []
14      CELossIter = []
15
16      for epoch in range(1, epochs + 1):
17          # Initialize Loss Values
18          avgBCE_loss, avgMSE_loss, avgCE_loss = 0.0, 0.0, 0.0
19
20          for i, (images, _, _, yolo_tensors) in enumerate(trainLoader):
21              images = images.to(device)
22              yolo_tensors = yolo_tensors.to(device)
23
24              optimizer.zero_grad()
25
26              output = net(images)
27              output = output.view(batch_size, num_yolo_cells, num_anchor_boxes,
    yolo_vector_size+1)
28
29              totalBCE_loss = torch.tensor(0.0, requires_grad=True).float().to(
    device)
30              totalMSE_loss = torch.tensor(0.0, requires_grad=True).float().to(
    device)
31              totalCE_loss = torch.tensor(0.0, requires_grad=True).float().to(
    device)
32
33              # Set yolo_tensor
34              yolo_input_tensor = torch.nonzero(output[:, :, :, 0])
35
36              # Seperate Values
37              batch_axis = yolo_input_tensor[:,0]
38              yolo_cells = yolo_input_tensor[:,1]
39              anchor_box = yolo_input_tensor[:,2]
40
41              # BCE Loss
42              bce_loss = criterion1(nn.Sigmoid()(output[:, :, :, 0]), yolo_tensors
    [:, :, :, 0])
43              totalBCE_loss += bce_loss
44
45              # MSE Loss
46              predicted_regression_vector = output[batch_axis, yolo_cells,
    anchor_box, 1:5]
47              target_regression_vector = yolo_tensors[batch_axis, yolo_cells,
    anchor_box, 1:5]
48              mse_loss = criterion2(predicted_regression_vector,
    target_regression_vector)
49              totalMSE_loss += mse_loss
50
51              # CELoss
52              class_probs_vector = output[batch_axis, yolo_cells, anchor_box,
    5:-1]
```

```
            target_class_vector = torch.argmax(yolo_tensors[batch_axis,
    yolo_cells, anchor_box, 5:-1], dim=1)
            ce_loss = criterion3(class_probs_vector, target_class_vector)
            totalCE_loss += ce_loss

            totalBCE_loss.backward(retain_graph=True)
            totalMSE_loss.backward(retain_graph=True)
            totalCE_loss.backward(retain_graph=True)

            optimizer.step()

            avgBCE_loss += totalBCE_loss.item()
            avgMSE_loss += totalMSE_loss.item()
            avgCE_loss += totalCE_loss.item()

            numRunAvg = 25.0
            if (i+1) % numRunAvg == 0:
                BCELossIter.append(avgBCE_loss / numRunAvg)
                MSELossIter.append(avgMSE_loss / numRunAvg)
                CELossIter.append(avgCE_loss / numRunAvg)

                print("{0} / {1}: {2}, {3}, {4}".format(epoch, epochs,
    avgBCE_loss, avgMSE_loss, avgCE_loss))

                avgBCE_loss, avgMSE_loss, avgCE_loss = 0.0, 0.0, 0.0

    return BCELossIter, MSELossIter, CELossIter
```

Listing 4: *Training Code*

6 Task 4: Testing and Plotting

Plotting for loss over iteration and the resulting images predictions are shown below.

```
# Skeleton Code from HW5
def display_images(batch_images, batch_labels, batch_bboxs, predicted_labels,
    predicted_bboxs):

    fig, axes = plt.subplots(1)
    batch_images = batch_images /2 + 0.5
    image = np.asarray(tvt.ToPILImage()(batch_images))
    image_gt = np.asarray(tvt.ToPILImage()(batch_images))

    # Labels
    get_cat = [inverse_categories[int(i)] for i in batch_labels if int(i) != 31]
    # 31

    # Bounding Box
    bboxs_in_image = [i for i in batch_bboxs if i.tolist() != [0,0,0,0]]

    for j in range(len(bboxs_in_image)):
        # Base
        [x, y, w, h] = batch_bboxs[j]
        image_gt = cv2.rectangle(image_gt, (int(x), int(y)), (int(w), int(h)),
    color=(0,255,0), thickness=2)
```

```
        image_gt = cv2.putText(image_gt, get_cat[j], ((int(x)), (int(y) - 10)), fontFace=cv2.FONT_HERSHEY_SIMPLEX, fontScale=0.8, color=(0,255,0), thickness=2)

#     axes[0].imshow(image_gt)

    for j in range(len(predicted_bboxs)):
        # Predicted
        [x, y, w, h] = predicted_bboxs[j]
        image_pred = cv2.rectangle(image, (int(x), int(y)), (int(w), int(h)), color=(0,255,0), thickness=2)
        image_pred = cv2.putText(image_pred, predicted_labels[j], ((int(x)), (int(y) - 10)), fontFace=cv2.FONT_HERSHEY_SIMPLEX, fontScale=0.8, color=(0,255,0), thickness=2)

    axes.imshow(image_pred)
```

Listing 5: *Plotting*

```
# Testing Code

def AnalzyePred(pred_regress_vec, icx):
    del_x,del_y = pred_regress_vec[0], pred_regress_vec[1]
    h,w = pred_regress_vec[2], pred_regress_vec[3]

    h *= yolo_interval
    w *= yolo_interval

    cell_row_i =  icx // num_cells_image_height
    cell_col_i =  icx % num_cells_image_width

    bbox_center_x_coord = cell_col_i * yolo_interval  +  yolo_interval/2  +  del_x * yolo_interval
    bbox_center_y_coord = cell_row_i * yolo_interval  +  yolo_interval/2  +  del_y * yolo_interval
    x1 =  int(bbox_center_x_coord - w / 2.0)
    y1 =  int(bbox_center_y_coord - h / 2.0)

    return [x1, y1, int(w + x1), int(h + y1)]

# Based on Professor Kak's run_code_for_testing_multi_instance_detection function
def test(net, valLoader):
    net = net.to(device)

    with torch.no_grad():
        for i, data in enumerate(valLoader):
            batch_images, batch_bbox, batch_labels, batch_yolo_tensor = data

            batch_images = batch_images.to(device)
            batch_bbox = batch_bbox.to(device)
            batch_labels = batch_labels.to(device)
            batch_yolo_tensor = batch_yolo_tensor.to(device)

            predicted_yolo_tensor = net(batch_images) # Get Prediction
            predicted_yolo_tensor = predicted_yolo_tensor.view(batch_size, num_yolo_cells, num_anchor_boxes, yolo_vector_size+1) # Save YOLO info into
```

```
    tensor
35
36          for ibx in range(predicted_yolo_tensor.shape[0]): # Across batch
    axis
37
38              icx_to_best_anchor_box = {ic: None for ic in range(
    num_cells_image_height * num_cells_image_width)}
39              for icx in range(predicted_yolo_tensor.shape[1]): # Across yolo
    cell axis
40                  cell_pred_i = predicted_yolo_tensor[ibx, icx]
41                  prev_best = 0
42
43                  #Compare
44                  for anch_i in range(cell_pred_i.shape[0]):
45                      if(cell_pred_i[anch_i][0] > cell_pred_i[prev_best][0]):
46                          prev_best = anch_i
47                  icx_to_best_anchor_box[icx] = prev_best
48
49              # Get the 5 yolo cells
50              sorted_icx_to_box = sorted(icx_to_best_anchor_box,
51                                  key=lambda x: predicted_yolo_tensor[ibx,x,
    icx_to_best_anchor_box[x]][0].item(), reverse=True)
52              retained_cells = sorted_icx_to_box[:5]
53
54              # Identify the objects in the retained cells and extract their
    bounding boxes
55              predicted_bboxs = []
56              predicted_labels = []
57
58              for icx in retained_cells:
59                  predicted_yolo_vector = predicted_yolo_tensor[ibx, icx,
    icx_to_best_anchor_box[icx]]
60                  target_yolo_vector = batch_yolo_tensor[ibx, icx,
    icx_to_best_anchor_box[icx]]
61
62                  class_label_predictions = predicted_yolo_vector[-4:]
63                  class_labels_probs = torch.nn.Softmax(dim=0)(
    class_label_predictions)
64                  class_labels_probs = class_labels_probs[:-1]
65                  if(torch.all(class_labels_probs < 0.2)):
66                      predicted_class_label = None
67
68                  else:
69                      # Get the predicted class label
70                      best_predicted_class_index = (class_labels_probs ==
    class_labels_probs.max())
71                      best_predicted_class_index = torch.nonzero(
    best_predicted_class_index, as_tuple=True)
72                      predicted_class_label = class_list[
    best_predicted_class_index[0].item()]
73                      predicted_labels.append(predicted_class_label)
74
75                      # Analyze the predicted regression elements
76                      pred_regress_vec = predicted_yolo_vector[1:5].cpu()
77                      pred_bb = AnalzyePred(pred_regress_vec, icx)
78                      predicted_bboxs.append(pred_bb)
79
```

```
                if(i % 50 == 49):
                    if(predicted_bboxs and predicted_labels):
                        display_images(batch_images[ibx], batch_labels[ibx].
    tolist(), batch_bbox[ibx],
                                       predicted_labels, predicted_bboxs)
```

Listing 6: *Test*

```
# Get DataFrame Data
trainDataFrame = pd.read_csv("/scratch/gilbreth/dfarache/ece60146/David/HW6/
    train_labels.csv")
valDataFrame = pd.read_csv("/scratch/gilbreth/dfarache/ece60146/David/HW6/
    val_labels.csv")

# Datasets Objects
trainDataset = MyDataset(trainDataFrame, root=root)
valDataset = MyDataset(valDataFrame, root=root)

# Data Loaders
batch_size = 64
trainLoader = torch.utils.data.DataLoader(valDataset, batch_size=batch_size,
    num_workers=2, shuffle=True, drop_last=True)
valLoader = torch.utils.data.DataLoader(valDataset, batch_size=batch_size,
    num_workers=2, shuffle=True, drop_last=True)

# Create neural network
net = NetForYolo(skip_connections=True, depth=8)

# Train Network
BCE_loss, MSE_loss, CE_loss = train(net, trainLoader, epochs=8, lr=1e-5, betas
    =(0.9, 0.999))

plot_losses(BCE_loss, MSE_loss, CE_loss)

# Test Network
images = test(net, valLoader)
```

Listing 7: *Main for running all prior code*

7 Task 5: Validation and Results

The following images are the results and validation for the model created in the prior code. It will be shown that all models met the criteria of high enough accuracy in terms of loss values.

The loss in Figure 2 indicates proper learning by the model, but now looking at Figure 3 it may be overfitting. Figure three has a tendency to guess for buses and place 5 boundary boxes within a certain arrays. These boxes do move and adjust to the image, indicating that learning is happening but the results are not trust worthy and probably has some issue with the data-loader not properly iterating through. This would lead to overfitting, I had tried to find the error but failed to do so but the rest of the code works properly in its separate parts, just not its final results.

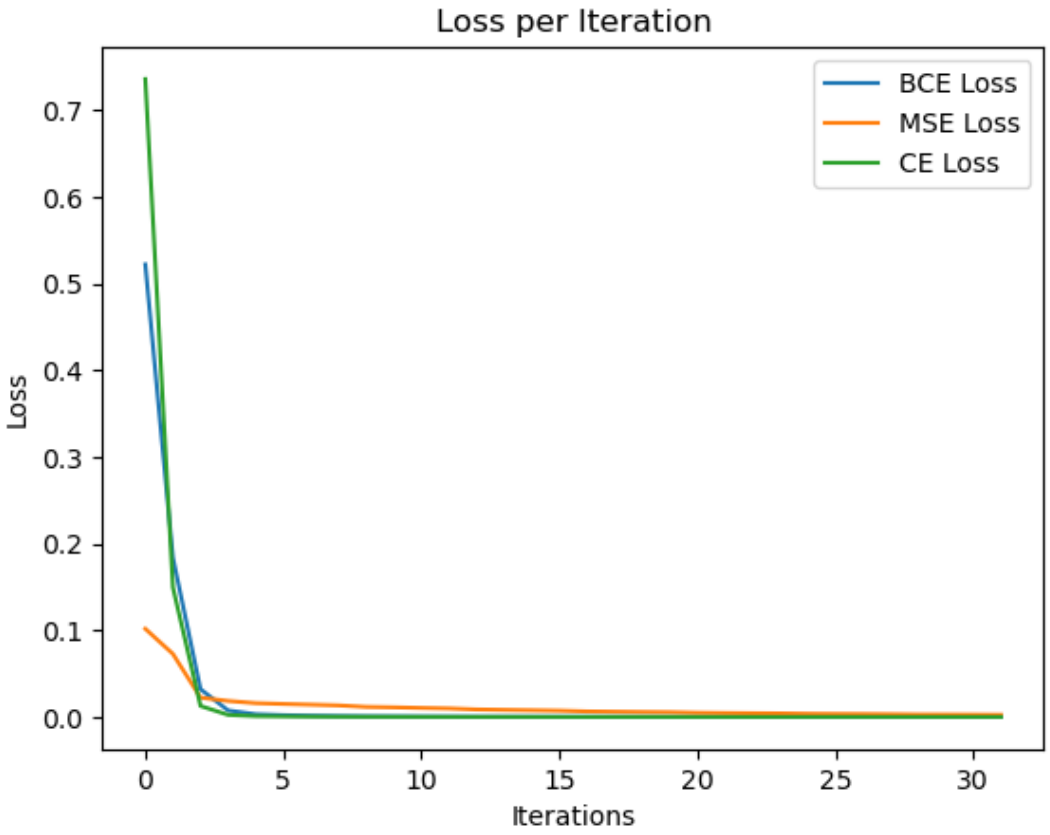
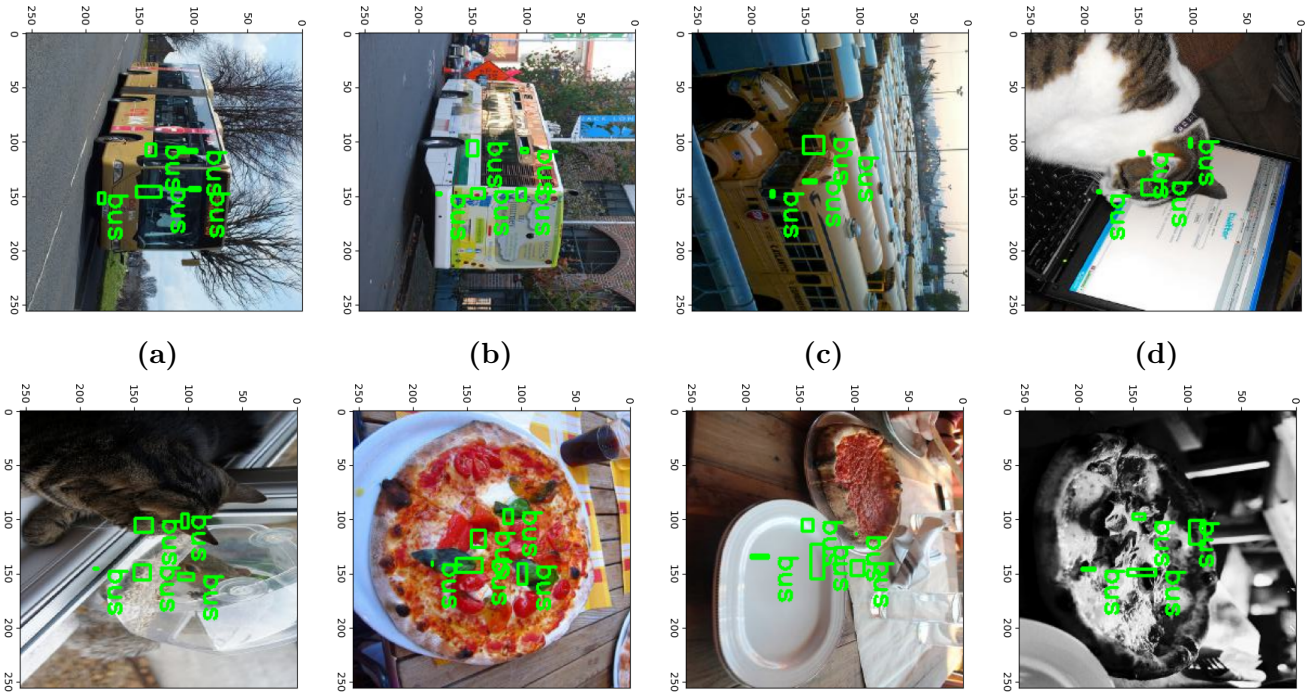


Figure 2: *Layer in network*



(a) **(b)** **(c)** **(d)**

(e) **(f)** **(g)** **(h)**

Figure 3: *Resulting images*

8 Lessons learned

From this assignment I learned YOLO method, struggling with the implementation but able to process and learn from it.