BME 646/ ECE695DL: Homework 6

Bianjiang Yang

29 Mar 2022

1 Introduction

The main goal for homework6:
1. To create a YOLO-like network architecture for multi-instance object detection and localization in the COCO images.
2. To take your use of the COCO dataset to the next level — by using the bounding boxes for all the object instance in the images.
3. To use COCO images and annotations include the classification labels and the bounding boxes for various types of objects in the images.

2 Methodology

Packages: torch, torch.nn, torch.nn.functional, torchvision.transforms, matplotlib.pyplot, copy, sklearn.metrics, os, sys, glob, seaborn, argparse, requests, PIL, pycocotools.coco, tqdm
Language: Python3
Tools: Anaconda3 virtual environment
System: Ubuntu 20.04.2 LTS (GNU/Linux 5.8.0-43-generic x86_64)

Instructions for running the code: The code I submitted has been modified to be runable, so you can simply run it by excuting the following script:
(1)train.py: CUDA_VISIBLE_DEVICES=1 python train.py
(2)validation.py: CUDA_VISIBLE_DEVICES=1 python -u validation.py

Other information: The dataloader will automatically skip the invalid bounding boxes, resize and save the resized images. You should modify the savefig directory or just use the plt.show() function to show the plots. For faster speed, I resized the images to 128×128

3 Implementation and Results

network.py

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.nn import init as init

class MODL(nn.Module):
    '''
    Multi object detection and localization network
    It contains a deep feature extraction stream with a multi-scale style,
    it is consisted of 3 CNNs, 6 Residual Blocks,
    and 3 skip connections in the same scale;
    and 2 final FC layers;
    a ReLU is utilized at the end of the regression stream
    for non-negative output.
    '''
    def __init__(self):
        super(MODL, self).__init__()
        # main stream
        fc_num = 4 * 4 * 512
        self.conv1_1 = nn.Sequential(nn.Conv2d(in_channels=3,
                                               out_channels=32,
                                               kernel_size=3,
                                               stride=2, padding=1),
                                     nn.BatchNorm2d(32),
                                     nn.LeakyReLU(0.2, True))
        self.conv1_2 = nn.Sequential(ResBlock(num_feat=32),
                                     ResBlock(num_feat=32))
        self.conv2_1 = nn.Sequential(nn.Conv2d(in_channels=32,
                                               out_channels=64,
                                               kernel_size=3,
                                               stride=2, padding=1),
                                       nn.BatchNorm2d(64),
                                       nn.LeakyReLU(0.2, True))
        self.conv2_2 = nn.Sequential(ResBlock(num_feat=64),
                                       ResBlock(num_feat=64))
        self.conv3_1 = nn.Sequential(nn.Conv2d(in_channels=64,
                                               out_channels=128,
                                               kernel_size=3,
                                               stride=2, padding=1),
                                       nn.BatchNorm2d(128),
                                       nn.LeakyReLU(0.2, True))
        self.conv3_2 = nn.Sequential(ResBlock(num_feat=128),
                                       ResBlock(num_feat=128),
                                       )
        self.conv4_1 = nn.Sequential(nn.Conv2d(in_channels=128,
                                               out_channels=256,
```

```
                                            kernel_size=3,
                                            stride=2, padding=1),
                                    nn.BatchNorm2d(256),
                                    nn.LeakyReLU(0.2, True))
            self.conv4_2 = nn.Sequential(ResBlock(num_feat=256),
                                         ResBlock(num_feat=256),
                                         )
            self.conv5 = nn.Sequential(nn.Conv2d(in_channels=256,
                                            out_channels=512,
                                            kernel_size=3,
                                            stride=2, padding=1),
                                    nn.LeakyReLU(0.2, True))

            # fc stream
            self.fc = nn.Sequential(nn.Flatten(),
                                    nn.Linear(fc_num, 4096),
                                    nn.LeakyReLU(0.2, True),
                                    nn.Linear(4096, 2048),
                                    nn.LeakyReLU(0.2, True),
                                    nn.Linear(2048, 1440 + 5*6*6))


        def forward(self, x):
            x = self.conv1_1(x)
            x = x + self.conv1_2(x)
            x = self.conv2_1(x)
            x = x + self.conv2_2(x)
            x = self.conv3_1(x)
            x = x + self.conv3_2(x)
            x = self.conv4_1(x)
            x = x + self.conv4_2(x)
            x = self.conv5(x)
            x = self.fc(x)

            return x

    class ResBlock(nn.Module):
        """Residual block with BN.

        It has a style of:
            ---Conv-BN-ReLU-Conv-BN+-ReLU
             |______________________|

        """

        def __init__(self, num_feat=64, res_scale=1, dilation=1):
```

```
        super().__init__()
        self.res_scale = res_scale
        padding = get_valid_padding(3, dilation)

        self.conv1 = nn.Conv2d(num_feat, num_feat, 3, 1, bias=True, \
                               padding=padding, dilation=dilation)
        self.bn1 = nn.BatchNorm2d(num_feat)
        self.conv2 = nn.Conv2d(num_feat, num_feat, 3, 1, bias=True, \
                               padding=padding, dilation=dilation)
        self.bn2 = nn.BatchNorm2d(num_feat)
        self.act = nn.LeakyReLU(0.2, True)

        default_init_weights([self.conv1, self.conv2], 0.1)

    def forward(self, x):
        identity = x
        out = self.bn2(self.conv2(self.act(self.bn1(self.conv1(x)))))
        return identity + out * self.res_scale

@torch.no_grad()
def default_init_weights(module_list, scale=1, bias_fill=0, **kwargs):
    """Initialize network weights with kaiming_normal.
    """
    if not isinstance(module_list, list):
        module_list = [module_list]
    for module in module_list:
        for m in module.modules():
            if isinstance(m, nn.Conv2d):
                init.kaiming_normal_(m.weight, **kwargs)
                m.weight.data *= scale
                if m.bias is not None:
                    m.bias.data.fill_(bias_fill)
            elif isinstance(m, nn.Linear):
                init.kaiming_normal_(m.weight, **kwargs)
                m.weight.data *= scale
                if m.bias is not None:
                    m.bias.data.fill_(bias_fill)
            elif isinstance(m, _BatchNorm):
                init.constant_(m.weight, 1)
                if m.bias is not None:
                    m.bias.data.fill_(bias_fill)

def get_valid_padding(kernel_size, dilation):
    # get valid padding number
    kernel_size = kernel_size + (kernel_size - 1) * (dilation - 1)
    padding = (kernel_size - 1) // 2
```

```
    return padding
```

dataloader.py

```
import torch
import torchvision
import torchvision.transforms as tvt
from torch.utils.data import DataLoader, Dataset
import glob
from pycocotools.coco import COCO
import os
from PIL import Image
import numpy as np
import cv2
from skimage import io
import matplotlib as plt

class hw6_dataloader(Dataset):
    def __init__(self, datapath, cocodatapath, transform, resize_w, resize_h, resized_path):
        self.tvtNorm = transform
        max_instance = 5
        self.num_archor_box = max_instance
        class_list = ['dog', 'cat', 'horse']
        self.label_list = []
        self.ID_list    = []
        self.bbox_list  = []
        self.exists     = []
        self.filename   = []
        self.skip = 0
        self.n_class = len(class_list)
        coco = COCO(cocodatapath)

        for i in range(self.n_class):
            if not os.path.exists(resized_path + class_list[i]):
                os.makedirs(resized_path + class_list[i])
            catIds = coco.getCatIds(catNms=class_list[i])
            imgIds = coco.getImgIds(catIds=catIds)
            # print(imgIds)
            # file_target = os.path.join(datapath, class_list[i])
            number_of_img = len(imgIds)
            print("class: " + class_list[i])
            print(number_of_img)
            for k in range(number_of_img):
```

```
# print(k)
annIds = coco.getAnnIds(imgIds=imgIds[k], catIds=catIds, iscrowd=False)
anns = coco.loadAnns(annIds)
# Select only images with 2 to max_instance interested bboxs
num_of_bbox_interested = 0
for ann in range(len(anns)):
    if anns[ann]['category_id']==17 or anns[ann]['category_id']==18 or anns[ann]['cat
        num_of_bbox_interested+=1
# print("num_of_bbox_interested: ", num_of_bbox_interested)
if num_of_bbox_interested > max_instance:
    # print("num_of_bbox_interested >5: ", num_of_bbox_interested)
    continue
elif num_of_bbox_interested < 2:
    # print("num_of_bbox_interested <2: ", num_of_bbox_interested)
    continue
else:
    labels = np.zeros(max_instance)
    bboxs = np.zeros((max_instance ,4))
    exists = np.zeros(max_instance)
    image_id_str = str(imgIds[k])
    image_id_str = image_id_str.zfill(12)
    file_list = os.path.join(datapath, class_list[i], image_id_str + ".jpg")
    # print(file_list)
    if not os.path.exists(file_list):
        # print("file not exist: " + image_id_str + ".jpg")
        continue
    image = Image.open(file_list)
    w_i, h_i = image.size
    if image.mode != 'RGB':
        image = image.convert('RGB')
    # Resize image to 128*128
    resized_file_path = resized_path + class_list[i] + "/" + image_id_str + ".jpg"
    if not os.path.exists(resized_file_path):
        img_resize = image.resize((resize_h, resize_w), Image.BOX)
        img_resize.save(resized_file_path)
    for j in range(len(anns)):
        image_ann   = anns[j]
        category_id = image_ann['category_id']
        if category_id!=17 and category_id!=18  and category_id!=19:
            # print("skip!!", category_id)
            continue
        else:
            labels[j] = category_id-17
            exists[j]   = 1
            image_bbox = np.array(image_ann['bbox'])
            # Resize bbox
```

```
                        image_bbox[0] = image_bbox[0] * resize_w / w_i
                        image_bbox[1] = image_bbox[1] * resize_h / h_i
                        image_bbox[2] = image_bbox[2] * resize_w / w_i
                        image_bbox[3] = image_bbox[3] * resize_h / h_i
                        # if image_bbox[0]+image_bbox[2]>128 or image_bbox[1]+image_bbox[3]>128:
                        #     print("image_bbox", image_bbox)
                        bboxs[j,:] = image_bbox

                self.bbox_list.append(bboxs)
                self.label_list.append(labels)
                self.exists.append(exists)
                self.ID_list.append(resized_file_path)
                # print('image ' + str(imgIds[k]) + ' is done!')
                # print(labels)


    def __getitem__ (self, ID) :
        # print(self.ID_list)
        img_path = self.ID_list[ID]
        # Data in tensor
        X = self.tvtNorm(Image.open(img_path).convert('RGB'))
        # Label
        y_label = self.label_list[ID]
        bbox_label = self.bbox_list[ID]
        num_obj_in_img = np.sum(self.exists[ID])

        return X, bbox_label, y_label, num_obj_in_img, img_path

    def __len__ (self) :
        return len(self.ID_list) # Total nb of sample
```

train.py

```
import torch
import torch.nn as nn
import network
import torchvision.transforms as tvt
import matplotlib.pyplot as plt
import dataloader
import copy

# class Anchorbox():
#     def __init__(self, AR, tlc, ab_height, ab_width):
```

```
#         self.AR = AR
#         self.tlc = tlc
#         self.ab_height = ab_height
#         self.ab_width = ab_width

# from torchvision.ops.boxes import _box_inter_union
# def giou_loss(input_boxes, target_boxes, eps=1e-7):
#     """
#     Args:
#         input_boxes: Tensor of shape (N, 4) or (4,).
#         target_boxes: Tensor of shape (N, 4) or (4,).
#         eps (float): small number to prevent division by zero
#     """
#     inter, union = _box_inter_union(input_boxes, target_boxes)
#     iou = inter / union
#
#     # area of the smallest enclosing box
#     min_box = torch.min(input_boxes, target_boxes)
#     max_box = torch.max(input_boxes, target_boxes)
#     area_c = (max_box[:, 2] - min_box[:, 0]) * (max_box[:, 3] - min_box[:, 1])
#
#     giou = iou - ((area_c - union) / (area_c + eps))
#
#     loss = 1 - giou
#
#     return loss.sum()

def train(transform, device, resized_path, lr = 1e-3, momentum = 0.9, epochs = 10,
          batch_size = 10, data_path = "/home/yangbj/695/YANG/hw5/COCO_Download/Train",
          save_path = "/home/yangbj/695/hw6/",
          cocodatapath = "/home/yangbj/695/YANG/hw4/annotations/instances_train2017.json",
          num_anchor_boxes = 5, vector_len = 8, yolo_interval = 20):

    coco_data = dataloader.hw6_dataloader(datapath=data_path, cocodatapath=cocodatapath,
                                      transform=transform, resize_w=128, resize_h=128, resized_path =
    train_data = torch.utils.data.DataLoader(coco_data, batch_size=batch_size,
                                             shuffle=True, num_workers=2)
    net = network.MODL()
    net = copy.deepcopy(net)
    # net.load_state_dict(torch.load("/home/yangbj/695/hw5/net50.pth"))
    net = net.to(device)
    running_loss1 = []
    running_loss2 = []
    running_loss3 = []
    loss_item1 = 0
    loss_item2 = 0
```

```
loss_item3 = 0
criterion1 = nn.BCELoss()
criterion2 = nn.MSELoss()
criterion3 = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(net.parameters(), lr=lr, betas=(0.9, 0.999),
                             eps=1e-08, weight_decay=0, amsgrad=False)

# optimizer = torch.optim.SGD(net.parameters(), lr=lr, momentum=momentum)
print("\n\nStarting training...")
for epoch in range(epochs):
    print("")
    for i, data in enumerate(train_data):
        img, bbox, category_label, num_objects_in_img, _ = data
        img = img.to(device).float()
        bbox = bbox.to(device)
        category_label = category_label.to(device)

        num_cells_w = img.shape[-1] // yolo_interval
        num_cells_h = img.shape[-2] // yolo_interval
        num_cells = num_cells_w * num_cells_h

        # initialize GT
        GT = torch.zeros(batch_size, num_cells, num_anchor_boxes, vector_len + 1).to(device)  # B
        for batch in range(batch_size):
            for cell in range(num_cells):
                for anchor in range(num_anchor_boxes):
                    GT[batch, cell, anchor, 8] = 1
        # bbox = torch.transpose(torch.stack(bbox), 0, 1)
        # bbox = bbox.to(torch.float32).to(device)
        # Construct the anchor boxes
        # anchor_boxes_1_1 = [[AnchorBox("1/1", (i * yolo_interval, j * yolo_interval),yolo_inter
        #                       for i in range(0, num_cells_image_height)] for j in range(0, num_c
        # anchor_boxes_1_3 = [[AnchorBox("1/3", (i * yolo_interval, j * yolo_interval), yolo_inte
        #                       for i in range(0, num_cells_image_height)] for j in range(0, num_c
        # anchor_boxes_1_5 = [[AnchorBox("1/5", (i * yolo_interval, j * yolo_interval), yolo_inte
        #                       for i in range(0, num_cells_image_height)] for j in range(0, num_c
        # anchor_boxes_5_1 = [[AnchorBox("5/1", (i * yolo_interval, j * yolo_interval), 5*yolo_in
        #                       for i in range(0, num_cells_image_height)] for j in range(0, num_c
        # anchor_boxes_3_1 = [[AnchorBox("3/1", (i * yolo_interval, j * yolo_interval), 3*yolo_in
        #                       for i in range(0, num_cells_image_height)] for j in range(0, num_c

        for img_idx in range(img.shape[0]):
            # print(num_objects_in_img)
            # print(img_idx)
            for obj_idx in range(int(num_objects_in_img[img_idx])):
                # print(bbox[img_idx][obj_idx])
```

```
# print(category_label[img_idx][obj_idx])
height_center_bb = bbox[img_idx][obj_idx][1].item() + \
                   (bbox[img_idx][obj_idx][3].item() / 2)  # y + height/2
width_center_bb = bbox[img_idx][obj_idx][0].item() +\
                  (bbox[img_idx][obj_idx][2].item() / 2)  # x _ width/2
obj_bb_height = bbox[img_idx][obj_idx][3].item()
obj_bb_width = bbox[img_idx][obj_idx][2].item()
if (obj_bb_height < 4) or (obj_bb_width < 4):
    continue
AR = float(obj_bb_height) / float(obj_bb_width)
cell_row_idx = width_center_bb // yolo_interval
cell_col_idx = height_center_bb // yolo_interval
cell_row_idx = 5 if cell_row_idx > 5 else cell_row_idx
cell_col_idx = 5 if cell_col_idx > 5 else cell_col_idx
if AR <= 0.2:
    # anchbox = anchor_boxes_1_5[cell_row_idx][cell_col_idx]
    anchor_number = 0
elif AR <= 0.5:
    # anchbox = anchor_boxes_1_3[cell_row_idx][cell_col_idx]
    anchor_number = 1
elif AR <= 1.5:
    # anchbox = anchor_boxes_1_1[cell_row_idx][cell_col_idx]
    anchor_number = 2
elif AR <= 4:
    # anchbox = anchor_boxes_3_1[cell_row_idx][cell_col_idx]
    anchor_number = 3
elif AR > 4:
    # anchbox = anchor_boxes_5_1[cell_row_idx][cell_col_idx]
    anchor_number = 4

bh = float(obj_bb_height) / float(yolo_interval)
bw = float(obj_bb_width) / float(yolo_interval)
# obj_center_x = float(bbox[img_idx][obj_idx][2].item() +
#                      (bbox[img_idx][obj_idx][0].item() / 2.0))
# obj_center_y = float(bbox[img_idx][obj_idx][3].item() +
#                      (bbox[img_idx][obj_idx][1].item() / 2.0))
yolocell_center_i = cell_row_idx * yolo_interval + float(yolo_interval) / 2.0
yolocell_center_j = cell_col_idx * yolo_interval + float(yolo_interval) / 2.0
del_x = float(width_center_bb - yolocell_center_i) / yolo_interval
del_y = float(height_center_bb - yolocell_center_j) / yolo_interval
yolo_vector = [1, del_x, del_y, bw, bh, 0, 0, 0, 0]
yolo_vector[5 + int(category_label[img_idx][obj_idx].item())] = 1
yolo_cell_index = cell_row_idx * num_cells_w + cell_col_idx
# print(yolo_vector)
GT[img_idx, int(yolo_cell_index), anchor_number] = torch.FloatTensor(yolo_vector)
```

```
            GT_flattened = GT.view(img.shape[0], -1)  #B*(1440+5*36)
            optimizer.zero_grad()
            outputs = net(img) #B*(1440+5*36)
            # BCE Loss
            loss1 = 0.0
            loss2 = 0.0
            loss3 = 0.0
            for ind in range(0, outputs.shape[1], 9):
                outputs1 = nn.Sigmoid()(outputs[:,ind])
                GT1 = GT_flattened[:,ind]
                loss1 += criterion1(outputs1, GT1)
                # loss_item1 += loss1
                outputs2 = outputs[:,ind+1:ind+5]
                GT2 = GT_flattened[:,ind+1:ind+5]
                loss2 += criterion2(outputs2, GT2)
                # loss_item2 += loss2
                outputs3 = outputs[:, ind + 5:ind+9]
                GT3 = GT_flattened[:, ind + 5:ind+9]
                # print(img.shape[0])
                for b in range(img.shape[0]):
                    if b==0:
                        class_label = torch.unsqueeze(torch.argmax(GT3[b,:]), 0)
                    else:
                        class_label = torch.cat((class_label, torch.unsqueeze(torch.argmax(GT3[b,:]),
                loss3 += criterion3(outputs3, class_label)

            loss = loss1 + loss2 + loss3
            loss.backward()
            optimizer.step()
            loss_item1 += loss1.item()
            loss_item2 += loss2.item()
            loss_item3 += loss3.item()
            per = 8
            if (i+1) % per == 0:
                print("[epoch:%d, batch:%5d, lr: %.9f] loss1: %.3f" %
                      (epoch + 1, i + 1, lr,  loss_item1 / float(per)))
                print("[epoch:%d, batch:%5d, lr: %.9f] loss2: %.3f" %
                      (epoch + 1, i + 1, lr, loss_item2 / float(per)))
                print("[epoch:%d, batch:%5d, lr: %.9f] loss3: %.3f" %
                      (epoch + 1, i + 1, lr, loss_item3 / float(per)))
                running_loss1.append(loss_item1/float(per))
                running_loss2.append(loss_item2 / float(per))
                running_loss3.append(loss_item3/float(per))
                loss_item1, loss_item2, loss_item3 = 0.0, 0.0, 0.0
    torch.save(net.state_dict(), save_path+'MODL4.pth')
```

```
    return running_loss1, running_loss2, running_loss3


if __name__ == '__main__':
    device  = torch.device('cuda:1')
    transform = tvt.Compose([tvt.ToTensor(), tvt.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])
    trainset_path = "/home/yangbj/695/PEC/hw5/COCO_Download/Train"
    coco_json = "/home/yangbj/695/PEC/hw4/annotations/instances_train2017.json"
    save_path = "/home/yangbj/695/hw6/"
    resized_path = "/home/yangbj/695/hw6/resized/"
    running_loss1, running_loss2, running_loss3 = train(transform = transform,
    device = device, resized_path=resized_path,
    lr = 1e-4, momentum = 0.9, epochs = 50, batch_size = 8,
    data_path = trainset_path, save_path = save_path,
    cocodatapath = "/home/yangbj/695/PEC/hw4/annotations/instances_train2017.json")

    plt.figure()
    plt.title('BCE Loss')
    plt.xlabel('Per 8 Iterations')
    plt.ylabel('Loss')
    plt.plot(running_loss1, label = 'BCE Loss')
    plt.legend(loc='upper right')
    plt.show()
    plt.savefig(save_path + "train3_loss1" + ".jpg")


    plt.figure()
    plt.title('MSE Loss')
    plt.xlabel('Per 8 Iterations')
    plt.ylabel('Loss')
    plt.plot(running_loss2, label='MSE Loss')
    plt.legend(loc='upper right')
    plt.show()
    plt.savefig(save_path + "train3_loss2" + ".jpg")

    plt.figure()
    plt.title('CE Loss')
    plt.xlabel('Per 8 Iterations')
    plt.ylabel('Loss')
    plt.plot(running_loss3, label='CE Loss')
    plt.legend(loc='upper right')
    plt.show()
    plt.savefig(save_path + "train3_loss3" + ".jpg")
```

validation.py

```
import torch
import torch.nn as nn
import network
import torchvision.transforms as tvt
import matplotlib.pyplot as plt
import dataloader
import copy
import sklearn.metrics
import os
import sys
import glob
import seaborn
import cv2

yolo_interval = 20


def test(net, transform, device,
         batch_size, data_path, save_path,
         jsonpath, resized_data_path, model_path):
    coco_data = dataloader.hw6_dataloader(datapath=data_path, cocodatapath=jsonpath,
                                          transform=transform, resize_w=128, resize_h=128, resized_pa

    test_data = torch.utils.data.DataLoader(coco_data,
                     batch_size=batch_size, shuffle=False, num_workers=2)
    print("test data len: ",len(test_data))

    net = copy.deepcopy(net)
    net.load_state_dict(torch.load(model_path))
    net = net.to(device)
    net.eval()

    print("\n\nStarting testing...")
    output_total = []
    label_total = []
    exist = 0
    classify = 0
    misclassify = 0
    notexist = 0
    num_anchor_boxes = 5
    vector_len = 8
    classes = ['dog', 'cat', 'horse', 'none']
```

```
labels_total = []
classify_total = []

for i, data in enumerate(test_data):
    img, bbox, category_label, num_objects_in_img, img_path = data
    img = img.to(device).float()
    bbox = bbox.to(device)
    # print(bbox)
    category_label = category_label.to(device)

    num_cells_w = img.shape[-1] // yolo_interval
    num_cells_h = img.shape[-2] // yolo_interval
    num_cells = num_cells_w * num_cells_h

    # initialize GT
    GT = torch.zeros(batch_size, num_cells, num_anchor_boxes, vector_len + 1).to(device)  # B*36*
    for batch in range(batch_size):
        for cell in range(num_cells):
            for anchor in range(num_anchor_boxes):
                GT[batch, cell, anchor, 8] = 1

    for img_idx in range(img.shape[0]):
        # print(num_objects_in_img)
        # print(img_idx)
        # print(num_objects_in_img[img_idx])
        for obj_idx in range(int(num_objects_in_img[img_idx])):
            height_center_bb = bbox[img_idx][obj_idx][1].item() + \
                               (bbox[img_idx][obj_idx][3].item() / 2)  # y + height/2
            width_center_bb = bbox[img_idx][obj_idx][0].item() + \
                              (bbox[img_idx][obj_idx][2].item() / 2)  # x _ width/2
            obj_bb_height = bbox[img_idx][obj_idx][3].item()
            obj_bb_width = bbox[img_idx][obj_idx][2].item()
            if (obj_bb_height < 4) or (obj_bb_width < 4):
                continue
            AR = float(obj_bb_height) / float(obj_bb_width)
            cell_row_idx = width_center_bb // yolo_interval
            cell_col_idx = height_center_bb // yolo_interval
            cell_row_idx = 5 if cell_row_idx > 5 else cell_row_idx
            cell_col_idx = 5 if cell_col_idx > 5 else cell_col_idx
            if AR <= 0.2:
                # anchbox = anchor_boxes_1_5[cell_row_idx][cell_col_idx]
                anchor_number = 0
            elif AR <= 0.5:
                # anchbox = anchor_boxes_1_3[cell_row_idx][cell_col_idx]
                anchor_number = 1
            elif AR <= 1.5:
```

```
                # anchbox = anchor_boxes_1_1[cell_row_idx][cell_col_idx]
                anchor_number = 2
            elif AR <= 4:
                # anchbox = anchor_boxes_3_1[cell_row_idx][cell_col_idx]
                anchor_number = 3
            elif AR > 4:
                # anchbox = anchor_boxes_5_1[cell_row_idx][cell_col_idx]
                anchor_number = 4

            bh = float(obj_bb_height) / float(yolo_interval)
            bw = float(obj_bb_width) / float(yolo_interval)
            # obj_center_x = float(bbox[img_idx][obj_idx][2].item() +
            #                      (bbox[img_idx][obj_idx][0].item() / 2.0))
            # obj_center_y = float(bbox[img_idx][obj_idx][3].item() +
            #                      (bbox[img_idx][obj_idx][1].item() / 2.0))
            yolocell_center_i = cell_row_idx * yolo_interval + float(yolo_interval) / 2.0
            yolocell_center_j = cell_col_idx * yolo_interval + float(yolo_interval) / 2.0
            del_x = float(width_center_bb - yolocell_center_i) / yolo_interval
            del_y = float(height_center_bb - yolocell_center_j) / yolo_interval
            yolo_vector = [1, del_x, del_y, bw, bh, 0, 0, 0, 0]
            yolo_vector[5 + int(category_label[img_idx][obj_idx].item())] = 1
            yolo_cell_index = cell_row_idx * num_cells_w + cell_col_idx
            # print(yolo_vector)
            GT[img_idx, int(yolo_cell_index), anchor_number] = torch.FloatTensor(yolo_vector)

GT_flattened = GT.view(img.shape[0], -1)  # B*(1440+5*36)
outputs = net(img)  # B*(1440+5*36)
predictions = outputs.view(batch_size, num_cells, num_anchor_boxes, 9)
GT = GT_flattened.view(batch_size, num_cells, num_anchor_boxes, 9)

for b in range(batch_size):
    img = cv2.imread(img_path[0])
    for n in range(num_cells):
        for nu in range(num_anchor_boxes):
            # print(nn.Sigmoid()(predictions[b,n,nu,0]))
            # if nn.Sigmoid()(predictions[b,n,nu,0])>=0.5:
            #     print("predictions", nn.Sigmoid()(predictions[b,n,nu,0]))
            #     print("GT", GT[b,n,nu,0])
            if (nn.Sigmoid()(predictions[b,n,nu,0])>=0.01 and GT[b,n,nu,0]==1) or (nn.Sigmoid
                exist += 1
            else: notexist += 1
            if nn.Sigmoid()(predictions[b,n,nu,0])>=0.01 and GT[b,n,nu,0]==1:
                outputs2 = predictions[b,n,nu,1:5]
                outputs3 = predictions[b,n,nu,5:9]
                text_output = classes[torch.argmax(outputs3)]
                GT2 = GT[b,n,nu,1:5]
```

```
GT3 = GT[b,n,nu,5:9]
text_GT = classes[torch.argmax(GT3)]
row_idx = n//6
col_idx = n%6

yolocell_center_row = row_idx * yolo_interval + float(yolo_interval) / 2.0
yolocell_center_col = col_idx * yolo_interval + float(yolo_interval) / 2.0

pred_obj_center_row = outputs2[0] * yolo_interval + yolocell_center_row
pred_obj_center_col = outputs2[1] * yolo_interval + yolocell_center_col
pred_bh = outputs2[2] * yolo_interval
pred_bw = outputs2[3] * yolo_interval
output_draw = [pred_obj_center_row, pred_obj_center_col, pred_bw, pred_bh]

obj_center_row = GT2[0] * yolo_interval + yolocell_center_row
obj_center_col = GT2[1] * yolo_interval + yolocell_center_col
real_bh = GT2[2] * yolo_interval
real_bw = GT2[3] * yolo_interval
GT_draw = [obj_center_row, obj_center_col, real_bw, real_bh]

cv2.rectangle(img, (int(output_draw[0] - output_draw[2] / 2), int(output_draw
              (int(output_draw[0] + output_draw[2] / 2), int(output_draw[1] +
              (0, 0, 255), 1)
cv2.putText(img, text_output,
            (int(output_draw[0] - output_draw[2] / 2),
                 int(output_draw[1] - output_draw[3] / 2)),
            cv2.FONT_HERSHEY_SIMPLEX, fontScale=0.5,
            color=(0, 0, 255), thickness=1)

cv2.rectangle(img, (int(GT_draw[0] - GT_draw[2] / 2), int(GT_draw[1] - GT_dra
              (int(GT_draw[0] + GT_draw[2] / 2),
               int(GT_draw[1] + GT_draw[3] / 2)),
              (0, 255, 0), 1)
cv2.putText(img, text_GT,
            (int(GT_draw[0] - GT_draw[2] / 2),
             int(GT_draw[1] - GT_draw[3] / 2)),
            cv2.FONT_HERSHEY_SIMPLEX, fontScale=0.5,
            color=(0, 255, 0), thickness=1)

# cv2.rectangle(img, (int(bbox[0][nu][0]), int(bbox[0][nu][1])),
#               (int(bbox[0][nu][0]+bbox[0][nu][2]), int(bbox[0][nu][1]+bbox[
#               1)
# cv2.putText(img, text_GT,
#             (int(bbox[0][nu][0]), int(bbox[0][nu][1])),
#             cv2.FONT_HERSHEY_SIMPLEX, fontScale=0.5,
#             color=(0, 255, 0), thickness=1)
```

```
                    # if torch.argmax(nn.Softmax()(outputs3))<3:
                    #     print(nn.Softmax()(outputs3))
                    # if torch.argmax(GT3)<3:
                    #     print(GT3)
                    # if torch.argmax(nn.Softmax()(outputs3)) == torch.argmax(GT3):
                    #     classify += 1
                    # else:
                    #     misclassify += 1
                    labels_total.append(torch.argmax(GT3).cpu())
                    classify_total.append(torch.argmax(nn.Softmax()(outputs3)).cpu())

        if not os.path.exists(save_path + 'tested_imgs/'):
            os.makedirs(save_path + 'tested_imgs/')
        cv2.imwrite(save_path + 'tested_imgs/' + img_path[0][-16:], img)
        # print("save an img to:" + save_path +
        #       'tested_imgs/' + img_path[0][-16:])

acc_exist = exist / (exist+notexist) * 100
print("acc_exist = "+ str(acc_exist) + "%")
# acc_classify = classify / (classify+misclassify) * 100
# print("acc_classify = " + str(acc_classify) + "%")

# calculate confusion matrix with sklearn module
confus_matrix = sklearn.metrics.confusion_matrix(
    labels_total, classify_total, labels=[0, 1, 2, 3])
print(confus_matrix)

# classification accuracy calculation
acc = 0
for i in range(confus_matrix.shape[0]):
    acc += confus_matrix[i][i]
Accuracy = acc / confus_matrix.sum() * 100
print('Accuracy:'+str(Accuracy)+'%')

# plot confusion matrix
plt_labels = classes
plt.figure(figsize = (10,7))
seaborn.heatmap(confus_matrix, annot=True, fmt= 'd', linewidths = .5,
                xticklabels= plt_labels, yticklabels= plt_labels)
plt.title("Net" + 'Accuracy:'+str(Accuracy)+'%')
plt.savefig(save_path + "net_confusion_matrix.jpg")
```

```
if __name__ == '__main__':

    # settings
    device  = torch.device('cuda:1')
    transform = tvt.Compose([tvt.ToTensor(),
                             tvt.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])
    dataset_path = "/home/yangbj/695/PEC/hw5/COCO_Download/Val"
    save_path = "/home/yangbj/695/hw6/"
    jsonpath = "/home/yangbj/695/PEC/hw4/annotations/instances_val2014.json"
    resized_data_path = "/home/yangbj/695/hw6/resized_Val/"
    batch_size = 1
    resize = [128, 128]
    net = network.MODL()
    model_path = '/home/yangbj/695/hw6/MODL3.pth'

    # Run test
    test(net= net,
         transform = transform, device = device, batch_size = batch_size,
         data_path = dataset_path, save_path = save_path,
         jsonpath = jsonpath,
         resized_data_path = resized_data_path, model_path = model_path)
```

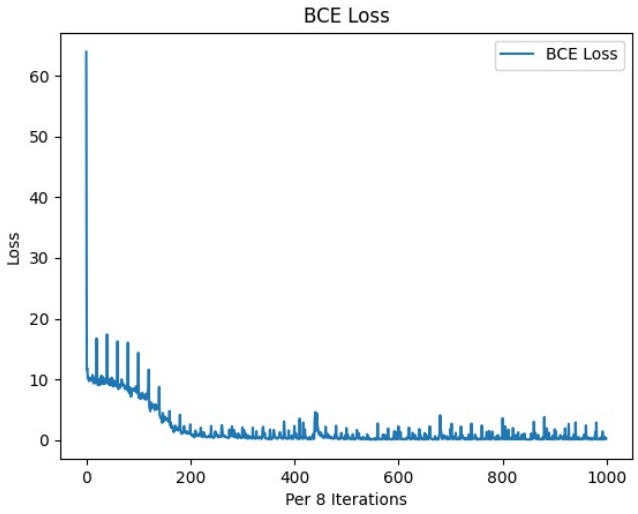

Figure 1: binary crossentropy loss

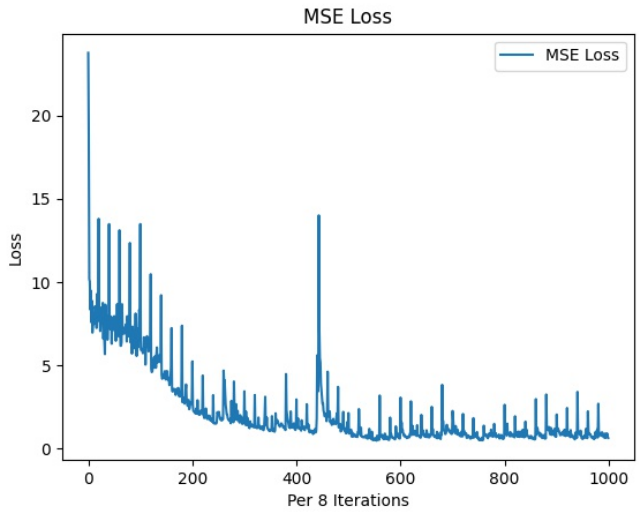

Figure 2: regression loss–MSE loss

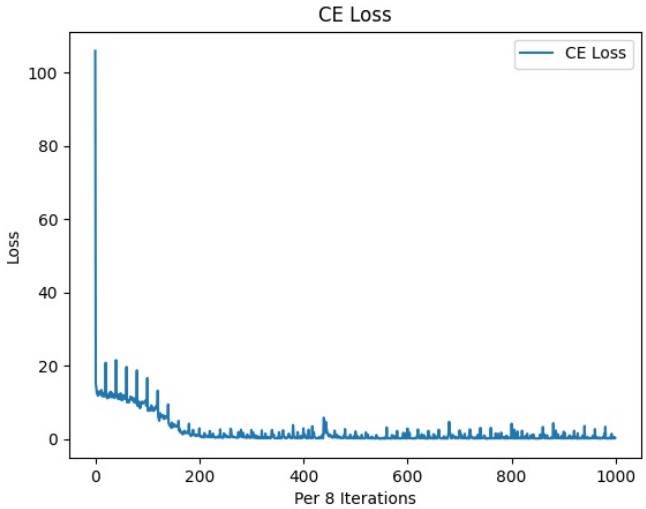

Figure 3: classificaation: crossentropy loss

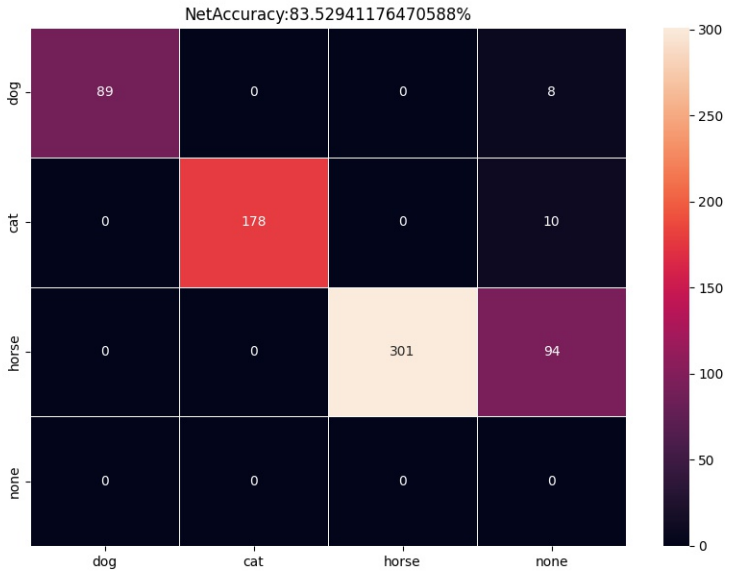

Figure 4: net: confusion matrix

Train Loss

The results in Fig. 1, Fig. 2 and Fig. 3 above show that:
(1) The loss will increase at the beginning of each epoch
(2) After about 600×8 iterations, the network converged.

Confusion Matrix

The results in Fig. 4 above shows that:
(1) The network has a classification accuracy of 83.5%, which indicates most of the prediction is correct.;
(2) The numbers of test samples for each categories are not the same because I choose to skip the image if the bounding box is not satisfactory;
(3) 'horce' class may be misclassified to 'none'. However, there is no misclassification between the three real classes.

Framework

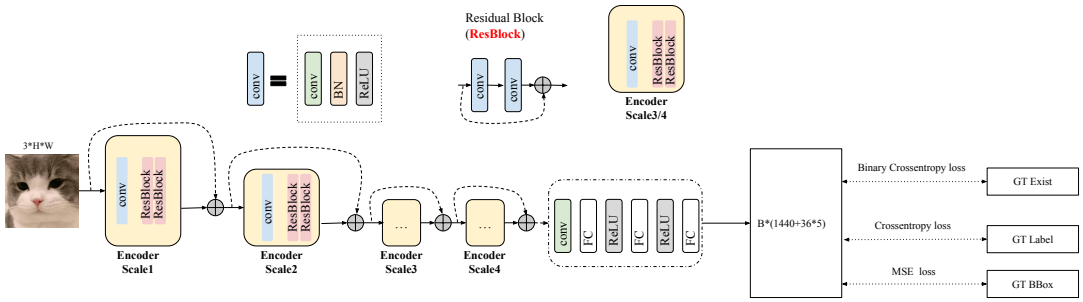

Figure 5: The framework: multiple object detection and localization network (MODL Net). It contains a deep feature extraction stream with a 4 multi-scale style, which is consisted of 4 CNNs, 8 Residual Blocks, and 4 skip connections in the same scale; a final stream with Conv and FC layers; a ReLU is utilized at the part of the regression stream for non-negative output; a softmax is utilized for BCE Loss.

Bounding Box Samples: As shown in Fig. 6, the green bounding box is the prediction; the red bounding box is the true label.

Implementation Details. Batch size=8; I choose Leaky ReLU with negative slop 0.2 except for the last activation function in the regression stream. I choose Adam optimizer with learning rate 10^{-4} for more stable training because SGD sometimes failed for such learning rate with loss nan issue. I initialize the convolution layers of my Residual Blocks with Kaiming Normal.

4 Lessons Learned

The hurdles faced and the techniques employed to overcome them:

(1) For dataloader,

We need to output the label with shape (1,) for one sample, which means the labels are int numbers

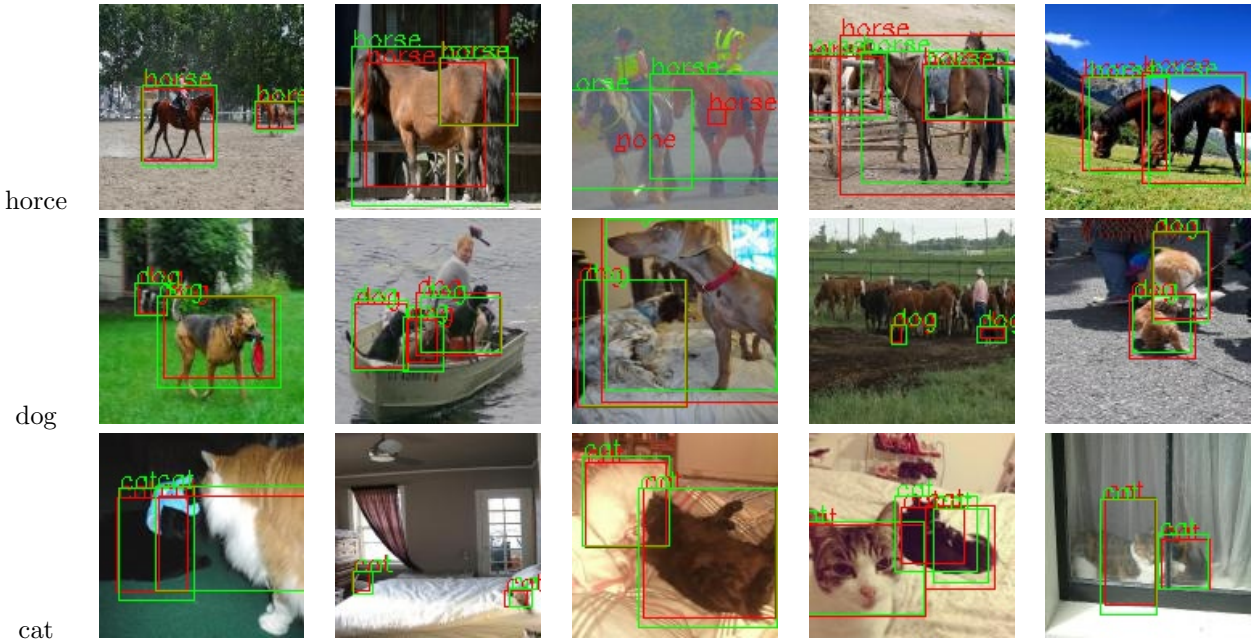

Figure 6: Testing samples

ranging (0,9). However, the outputs of the network are one-hot-vectors with shape(10,). nn.CrossEntropyLoss() can accept such two inputs. We do not need to use softmax to the output.

(2) For training,

yolo_vector is necessary to be correctly implemented so that the loss can be decreasing;

(3) For validation,

We need to eval the model to keep the model parameters static.

seaborn is a better tool than sklearn to plot the heatmap.

Confusion Matrix should be computed on CPU because of sklearn

bounding box should be int value because of cv2.rectangle

5 Suggested Enhancements

To the best of my understanding, this task emphasize more on downloader because it takes more time to accomplish. Also, most of the time was consumed in the ploting and some unimportant part debugging. So I suggest you could give a colab file and let the students fill necessary blanks.