hw6_ChengjunGuo

Chengjun Guo

March 2023

1 Programming Logic

The dataset code for this YOLO-like detector is the mostly the same as the previous homework. The only difference is that for each image, if it has multiple objects, the database would store a list of object information.

The training code for this homework is a little bit different than other network. For each image, there will be 1620 outputs to form a 6 by 6 cell, 5 anchor box for each cell, size-9 vector for each anchor box. In each batch, I calculate the object bounding boxes, object centers and the object class label. Then the vector will store presence, offset of x and y, height and width of bounding box, and the class label. Then I reshape the output of network to shape of batch size, cell number, anchor box number, size of vector. Looping through the tensor in unit of vector: for each vector, I calculate loss with the vector achieved previously as the ground truth. The loss for first digit is the binary cross-entropy loss. This is to predict the presence. The loss for the next 4 digits is mean square error. This is for the bounding box. The next 3 items are for the class labeling. Loss will be calculated with cross-entropy loss. After back processing the loss is trivial. We can add the loss together and keep the gradient attribute then back propagate the sum.

The logic for decoding the output of a yolo network is also inside the training file. It is for validation. The outputs shape of the network will be batch size by 1620. In case one object is repeatedly detected in one cell, I select one anchor box from each cell. Then for all cells, the maximum 5 presence anchor box cells would be selected where the presence is the first item. For these cells, if the probability of each label is less than 0.2 is dropped. The last fourth item to the last second item are the probability of each labels. If probability any label is larger than 0.2, the label with largest probability is considered as the prediction of the item. The rest is parsing the trivial process of parsing the anchor box. From the second item to the fifth, they are the delta x and y that is the displacement from the center and the fourth and fifth are the height and width that indicate the shape of the bounding box.

2 network structure

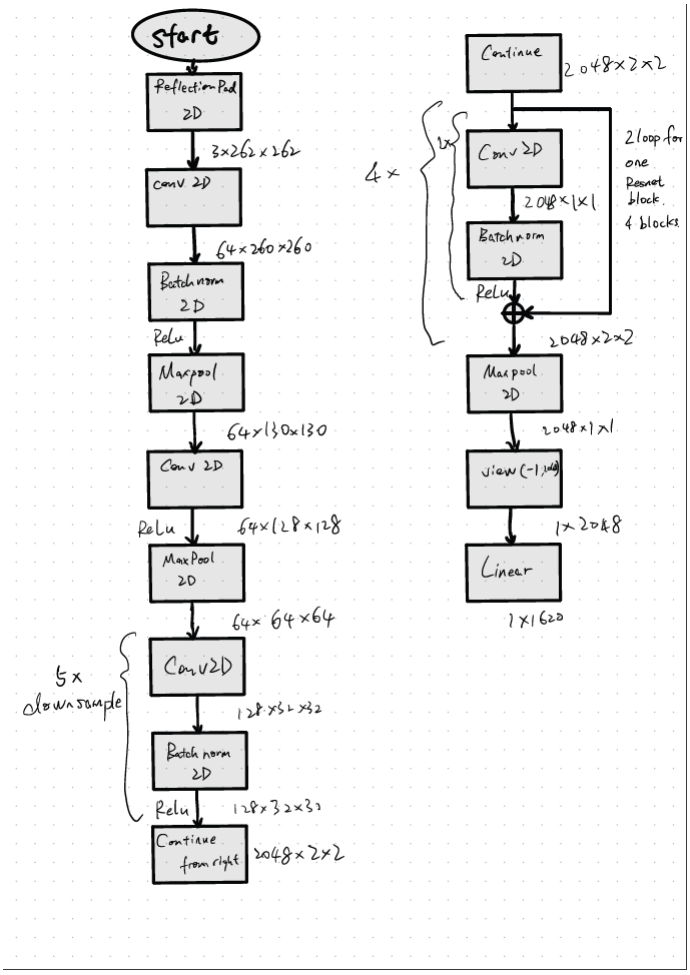

Figure 1: network

3 Inputs

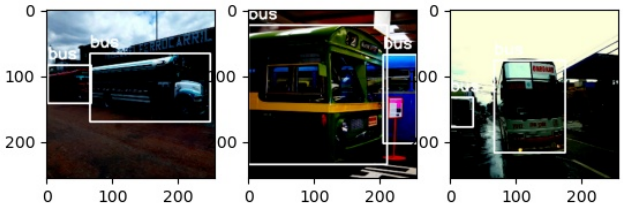

Figure 2: bus

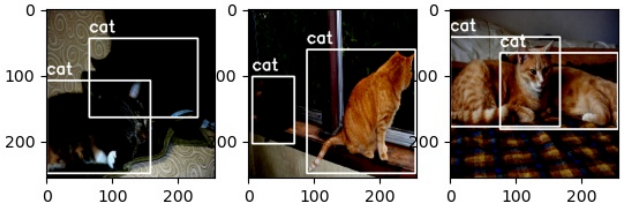

Figure 3: cat

Figure 4: pizza

4 Training loss

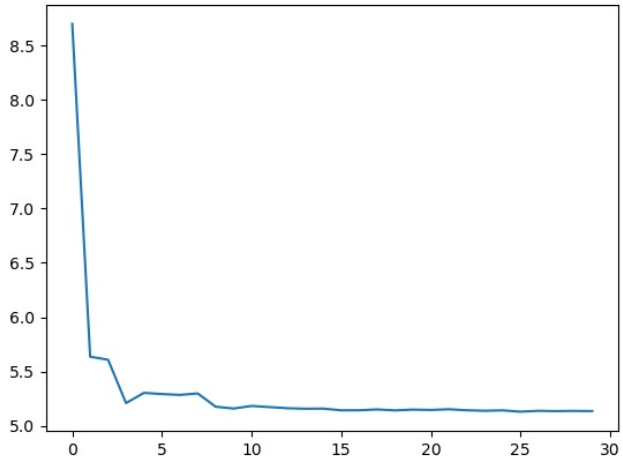

Figure 5: BCE

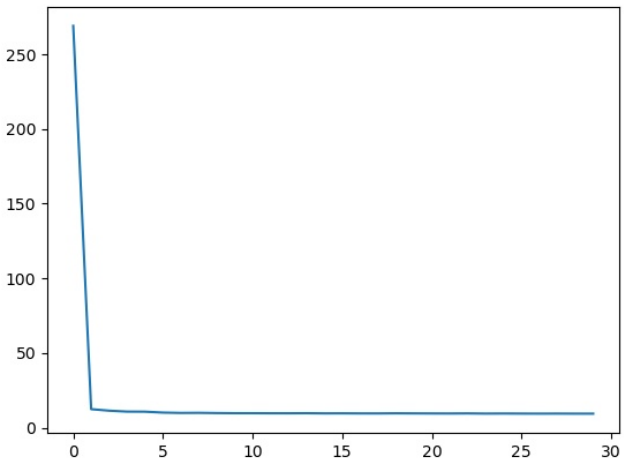

Figure 6: MSE

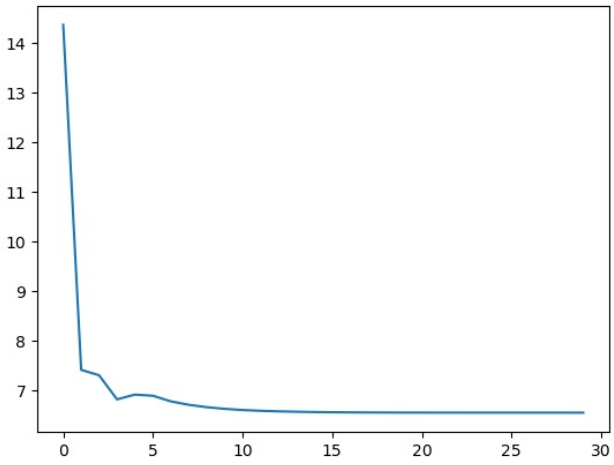

Figure 7: CE

5 Outputs

Figure 8: outputs

6 Discussion

The detector take much longer than expected to train. For 10 epoch, it takes about 40 hours training with cuda on rtx 3090. The training result it not as good as expected, when probability is set to larger than 0.2, most of images cannot be detected anything. The reason behind this should be the structure of network is too deep. I think the vanishing gradient is happening.

7 Code

coco dataset:

```
from hw6_COCO_dataset import MyDataset
import os
os.environ['KMP_DUPLICATE_LIB_OK'] = 'True'
import torch
from PIL import Image
import torchvision
import torchvision.transforms as tvt
from torchvision.io import read_image
from torch.utils.data import DataLoader, Dataset
```

```
import copy
from pycocotools.coco import COCO
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import confusion_matrix
from hw6_net import HW6Net
import time
import pickle
from statistics import mean
import cv2
import numpy as np

## reference: Avi kak's tutorial https://engineering.
    purdue.edu/kak/distRPG/RegionProposalGenerator-2.0.8
    _CodeOnly.html

def run_training(net,train_data_loader,net_save_path):
    net = copy.deepcopy(net)
    net = net.to(device)
    criterion3 = torch.nn.CrossEntropyLoss()
    criterion2 = torch.nn.MSELoss()
    criterion1 = torch.nn.BCELoss()
    optimizer = torch.optim.Adam(net.parameters(), lr=1e
        -3, betas=(0.9, 0.99))
    epochs = 10
    num_anchor_boxes = 5
    num_yolo_cells = 36
    max_obj_num = 5
    batch_size = 4
    Loss_presence = []
    Loss_labeling = []
    Loss_regression = []
    for epoch in range(epochs):
        start_time = time.time()
        running_loss = 0.0
        running_loss_presence = 0.0
        running_loss_labeling = 0.0
        running_loss_regression = 0.0
        for i,data in enumerate(train_data_loader):
            yolo_tensor = torch.zeros(batch_size,
                num_yolo_cells, num_anchor_boxes, 8)
            im_tensor, bbox_label_tensor, bbox_tensor,
                num_objects_in_image = data
            im_tensor = im_tensor.to(device)
            bbox_tensor = bbox_tensor.to(device)
```

```
bbox_label_tensor = bbox_label_tensor.to(
    device)
yolo_tensor = yolo_tensor.to(device)
num_cells_image_width = 6
num_cells_image_height = 6
for idx in range(max_obj_num):
    height_center_bb = (bbox_tensor[0, idx,
        1] + bbox_tensor[0, idx, 3]) // 2
    width_center_bb = (bbox_tensor[0, idx, 0]
         + bbox_tensor[0, idx, 2]) // 2
    obj_bb_height = bbox_tensor[0, idx, 3] -
        bbox_tensor[0, idx, 1]
    obj_bb_width = bbox_tensor[0, idx, 2] -
        bbox_tensor[0, idx, 0]
    if (obj_bb_height < 4.0) or (obj_bb_width
         < 4.0): continue     #skip too small
    cell_row_indx = (height_center_bb / 40).
        int()       # 40 as interval for 256 x
         256
    cell_col_indx = (width_center_bb / 40).
        int()
    cell_row_indx = torch.clamp(cell_row_indx
        , max=num_cells_image_height - 1)
    cell_col_indx = torch.clamp(cell_col_indx
        , max=num_cells_image_width - 1)
    bh = obj_bb_height.float() / 40
    bw = obj_bb_width.float() / 40
    obj_center_x = (bbox_tensor[0, idx][2].
        float() + bbox_tensor[0, idx][0].float
        ()) / 2.0
    obj_center_y = (bbox_tensor[0, idx][3].
        float() + bbox_tensor[0, idx][1].float
        ()) / 2.0
    yolocell_center_i = cell_row_indx * 40 +
        float(40) / 2.0
    yolocell_center_j = cell_col_indx * 40 +
        float(40) / 2.0
    del_x = (obj_center_x.float() -
        yolocell_center_j.float()) / 40
    del_y = (obj_center_y.float() -
        yolocell_center_i.float()) / 40
    class_label_of_object = bbox_label_tensor
        [0, idx].item()
    #shape
    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
        yolo_vector = torch.FloatTensor([0, del_x
            .item(), del_y.item(), bh.item(), bw.
            item(), 0, 0, 0])
        yolo_vector[0] = 1
        yolo_vector[5 + class_label_of_object] =
            1
        yolo_cell_index = cell_row_indx.item() *
            num_cells_image_width + cell_col_indx.
            item()
        yolo_tensor[0, yolo_cell_index,
            anch_box_index] = yolo_vector
        yolo_tensor_aug = torch.zeros(batch_size,
             num_yolo_cells, num_anchor_boxes, 9).
            float().to(device)
        yolo_tensor_aug[:, :, :, :-1] =
            yolo_tensor

    for icx in range(num_yolo_cells):
        for iax in range(num_anchor_boxes):
            if yolo_tensor_aug[0, icx, iax, 0] ==
                0:
                yolo_tensor_aug[0, icx, iax, -1]
                    = 1

    optimizer.zero_grad()
    outputs = net(im_tensor)
    predictions_aug = outputs.view(batch_size,
        num_yolo_cells, num_anchor_boxes, 9)
    loss = torch.tensor(0.0, requires_grad=True).
        float().to(device)
    for icx in range(num_yolo_cells):
        for iax in range(num_anchor_boxes):
            pred_yolo_vector = predictions_aug[0,
                icx, iax]
            target_yolo_vector = yolo_tensor_aug
                [0, icx, iax]
```

```
            #BCE for presence
            object_presence = torch.nn.Sigmoid()(
                torch.unsqueeze(pred_yolo_vector
                [0], dim=0))
            target_for_prediction = torch.
                unsqueeze(target_yolo_vector[0],
                dim=0)
            bceloss = criterion1(object_presence,
                 target_for_prediction)
            running_loss_presence += bceloss.item
                ()
            loss = loss + bceloss
            #MSE for regression
            pred_regression_vec =
                pred_yolo_vector[1:5]
            pred_regression_vec = torch.unsqueeze
                (pred_regression_vec, dim=0)
            target_regression_vec = torch.
                unsqueeze(target_yolo_vector[1:5],
                 dim=0)
            regression_loss = criterion2(
                pred_regression_vec,
                target_regression_vec)
            running_loss_regression+=
                regression_loss.item()
            loss += regression_loss
            #CrossEntropy 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)
            running_loss_labeling +=
                class_labeling_loss.item()
            loss += class_labeling_loss
    loss.backward()
    optimizer.step()
    running_loss += loss.item()
    if (i+1) % 500 == 0:
        print("[epoch: %d, batch: %5d] loss: %.3f
            " % (epoch+1, i+1, running_loss/500))
```

```
                    Loss_presence.append(
                        running_loss_presence/500)
                    Loss_regression.append(
                        running_loss_regression/500)
                    Loss_labeling.append(
                        running_loss_labeling/500)
                    print(running_loss_presence/500,
                        running_loss_regression/500,
                        running_loss_labeling/500)
                    running_loss_presence = 0.0
                    running_loss_labeling = 0.0
                    running_loss_regression = 0.0
                    print("----------time executing for epoch
                        %s : %s seconds----------" % (epoch +
                        1, (time.time() - start_time)))
        torch.save(net, net_save_path)

        return Loss_presence, Loss_regression, Loss_labeling



def decoder_for_pbc(category_id):
    # dict = {59:0,6:1,17:2}
    dict = {0: 'pizza', 1: 'bus', 2: 'cat'}
    return dict[category_id]



if __name__ == '__main__':
    if torch.cuda.is_available() == True:
        device = torch.device("cuda:0")
    else:
        device = torch.device("cpu")
    # train_dataset = MyDataset(True)

    # train_data_loader = torch.utils.data.DataLoader(
        dataset=train_dataset, batch_size=4, shuffle=False
        , num_workers=4)
    # train_data_loader = torch.utils.data.DataLoader(
        dataset=train_dataset, batch_size=4, shuffle=True,
         num_workers=4)
    # net = HW6Net()
    net = torch.load('net1.pth')
    # loss_BC, loss_MSE,loss_Crossentropy = run_training(
        net,train_data_loader,"net1.pth")
    # num_layers = len(list(net.parameters()))
```

```
# print(num_layers)
# print(loss_BC,loss_MSE,loss_Crossentropy)

# plt.figure()
# plt.plot(loss_BC, label = "Presence Training Loss")
# plt.savefig('loss_BCE_labeling.jpg')
# plt.figure()
# plt.plot(loss_MSE, label = "Regression Training
    Loss")
# plt.savefig('loss_MSE_regression.jpg')
# plt.figure()
# plt.plot(loss_Crossentropy, label = "Labeling
    Training Loss")
# plt.savefig('loss_CE_Labeling.jpg')


val_dataset = MyDataset(False)
val_data_loader = torch.utils.data.DataLoader(dataset
   =val_dataset, batch_size=1, shuffle=False,
    num_workers=4)
num_anchor_boxes = 5
num_yolo_cells = 36
max_obj_num = 5
batch_size = 1
count = 0
for i, data in enumerate(val_data_loader):
    yolo_tensor = torch.zeros(batch_size,
        num_yolo_cells, num_anchor_boxes, 8)
    im_tensor, bbox_label_tensor, bbox_tensor,
        num_objects_in_image = data
    im_tensor = im_tensor.to(device)
    bbox_tensor = bbox_tensor.to(device)
    bbox_label_tensor = bbox_label_tensor.to(device)
    yolo_tensor = yolo_tensor.to(device)
    output = net(im_tensor)
    outputs = output.view(num_yolo_cells,
        num_anchor_boxes, 9)
    print(outputs.shape)
    icx_2_best_anchor_box = {ic: None for ic in range
        (36)}
    for icx in range(outputs.shape[0]):
        cell_predi = outputs[icx]
        prev_best = 0
        for anchor_bdx in range(cell_predi.shape[0]):
            if cell_predi[anchor_bdx][0] > cell_predi
                [prev_best][0]:
```

```
                prev_best = anchor_bdx
        best_anchor_box_icx = prev_best
        icx_2_best_anchor_box[icx] =
            best_anchor_box_icx
    sorted_icx_to_box = sorted(icx_2_best_anchor_box,
        key=lambda x: outputs[x,
       icx_2_best_anchor_box[x]][0].item(), reverse=
       True)
    retained_cells = sorted_icx_to_box[:5]
    objects_detected = []
    predicted_bboxes = []
    predicted_labels_for_bboxes = []
    predicted_label_index_vals = []
    for icx in retained_cells:
        pred_vec = outputs[icx, icx_2_best_anchor_box
            [icx]]
        class_labels_predi = pred_vec[-4:]
        class_labels_probs = torch.nn.Softmax(dim=0)(
            class_labels_predi)
        class_labels_probs = class_labels_probs[:-1]
        if torch.all(class_labels_probs < 0.2):
            predicted_class_label = None
            # print('less')
        else:
            count+=1
            best_predicted_class_index = (
                class_labels_probs ==
                class_labels_probs.max())
            best_predicted_class_index = torch.
                nonzero(best_predicted_class_index,
                as_tuple=True)
            predicted_label_index_vals.append(
                best_predicted_class_index[0].item())
            predicted_class_label = decoder_for_pbc(
                best_predicted_class_index[0].item())
            predicted_labels_for_bboxes.append(
                predicted_class_label)
            print(predicted_bboxes)
            pred_regression_vec = pred_vec[1:5].cpu()
            del_x, del_y = pred_regression_vec[0],
                pred_regression_vec[1]
            h, w = pred_regression_vec[2],
                pred_regression_vec[3]
            h *= 40
            w *= 40
            cell_row_index = icx // 6
```

```
                cell_col_index = icx % 6
                bb_center_x = cell_col_index * 40 + 40 /
                    2 + del_x * 40
                bb_center_y = cell_row_index * 40 + 40 /
                    2 + del_y * 40
                bb_top_left_x = int(bb_center_x - w /
                    2.0)
                bb_top_left_y = int(bb_center_y - h /
                    2.0)
                predicted_bboxes.append([bb_top_left_x,
                    bb_top_left_y, int(w), int(h)])
        print(predicted_bboxes)
        print(count)
        fig = plt.figure(figsize=[12, 12])
        ax = fig.add_subplot(111)
        display_scale = 2
        new_im_tensor = torch.nn.functional.interpolate(
            im_tensor, scale_factor=display_scale, mode='
            bilinear', align_corners=False)
        ax.imshow(np.transpose(torchvision.utils.
            make_grid(new_im_tensor, normalize=True,
            padding=3, pad_value=255).cpu(), (1, 2, 0)))
        for i, bbox_pred in enumerate(predicted_bboxes):
            x, y, w, h = np.array(bbox_pred)
            x, y, w, h = [item * display_scale for item
                in (x, y, w, h)]
            rect = cv2.rectangle((x, y), w, h, (36, 255,
                12), 2)
            ax.annotate(predicted_labels_for_bboxes[i], (
                x, y - 1), color='red', weight='bold',
                fontsize=10 * display_scale)
        plt.savefig(str(i)+'.jpg')
        if i==7:
            break
```

net:

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

class HW6Net(nn.Module):
```

```
    def __init__(self, n_blocks=4):
        super(HW6Net, self).__init__()
        model = [nn.ReflectionPad2d(3),
                 nn.Conv2d(3, 64, kernel_size=3,padding
                     =0),
                 nn.BatchNorm2d(64),
                 nn.ReLU(True),
                 nn.MaxPool2d(2, 2),
                 nn.Conv2d(64, 64, kernel_size=3, padding
                     =1),
                 nn.ReLU(True),
                 nn.MaxPool2d(2, 2)
                 ]
        n_downsampling = 5
        for i in range(n_downsampling):
            mult = 2 ** i
            model += [nn.Conv2d(64 * mult, 64 * mult * 2,
                 kernel_size=3, stride=2, padding=1),
                       nn.BatchNorm2d(64 * mult * 2),
                       nn.ReLU(True)]
        mult = 2 ** n_downsampling
        for i in range(n_blocks):
            model += [ResnetBlock(64 * mult,64 * mult *
                2,64 * mult),
                       nn.ReLU(True)
                       ]
        model += [nn.MaxPool2d(2, 2)]
        self.model = nn.Sequential(*model)
        head = [
            nn.Linear(64 * mult, 1620), # 6*6*5*9
        ]
        self.head = nn.Sequential(*head)



    def forward(self, input):
        ft = self.model(input)
        ft = ft.view(-1,2048)
        # print(ft.shape)
        ft = self.head(ft)
        return ft

class ResnetBlock(nn.Module):

    def __init__(self, in_size, mid_size, out_size):
        super(ResnetBlock,self).__init__()
```

```
        self.conv1 = nn.Conv2d(in_size, mid_size, 3,
            padding=1)
        self.conv2 = nn.Conv2d(mid_size, out_size, 3,
            padding=1)
        self.batchnorm1 = nn.BatchNorm2d(mid_size)
        self.batchnorm2 = nn.BatchNorm2d(out_size)


    def forward(self, x):
        y = x
        x = F.relu(self.batchnorm1(self.conv1(x)))
        x = F.relu(self.batchnorm2(self.conv2(x)))
        return y+x
```

train:

```
from hw6_COCO_dataset import MyDataset
import os
os.environ['KMP_DUPLICATE_LIB_OK'] = 'True'
import torch
from PIL import Image
import torchvision
import torchvision.transforms as tvt
from torchvision.io import read_image
from torch.utils.data import DataLoader, Dataset
import copy
from pycocotools.coco import COCO
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import confusion_matrix
from hw6_net import HW6Net
import time
import pickle
from statistics import mean
import cv2
import numpy as np

## reference: Avi kak's tutorial https://engineering.
    purdue.edu/kak/distRPG/RegionProposalGenerator-2.0.8
    _CodeOnly.html

def run_training(net,train_data_loader,net_save_path):
```

```
net = copy.deepcopy(net)
net = net.to(device)
criterion3 = torch.nn.CrossEntropyLoss()
criterion2 = torch.nn.MSELoss()
criterion1 = torch.nn.BCELoss()
optimizer = torch.optim.Adam(net.parameters(), lr=1e
    -3, betas=(0.9, 0.99))
epochs = 10
num_anchor_boxes = 5
num_yolo_cells = 36
max_obj_num = 5
batch_size = 4
Loss_presence = []
Loss_labeling = []
Loss_regression = []
for epoch in range(epochs):
    start_time = time.time()
    running_loss = 0.0
    running_loss_presence = 0.0
    running_loss_labeling = 0.0
    running_loss_regression = 0.0
    for i,data in enumerate(train_data_loader):
        yolo_tensor = torch.zeros(batch_size,
            num_yolo_cells, num_anchor_boxes, 8)
        im_tensor, bbox_label_tensor, bbox_tensor,
            num_objects_in_image = data
        im_tensor = im_tensor.to(device)
        bbox_tensor = bbox_tensor.to(device)
        bbox_label_tensor = bbox_label_tensor.to(
            device)
        yolo_tensor = yolo_tensor.to(device)
        num_cells_image_width = 6
        num_cells_image_height = 6
        for idx in range(max_obj_num):
            height_center_bb = (bbox_tensor[0, idx,
                1] + bbox_tensor[0, idx, 3]) // 2
            width_center_bb = (bbox_tensor[0, idx, 0]
                + bbox_tensor[0, idx, 2]) // 2
            obj_bb_height = bbox_tensor[0, idx, 3] -
                bbox_tensor[0, idx, 1]
            obj_bb_width = bbox_tensor[0, idx, 2] -
                bbox_tensor[0, idx, 0]
            if (obj_bb_height < 4.0) or (obj_bb_width
                < 4.0): continue     #skip too small
            cell_row_indx = (height_center_bb / 40).
                int()       # 40 as interval for 256 x
```

```
     256
cell_col_indx = (width_center_bb / 40).
    int()
cell_row_indx = torch.clamp(cell_row_indx
    , max=num_cells_image_height - 1)
cell_col_indx = torch.clamp(cell_col_indx
    , max=num_cells_image_width - 1)
bh = obj_bb_height.float() / 40
bw = obj_bb_width.float() / 40
obj_center_x = (bbox_tensor[0, idx][2].
    float() + bbox_tensor[0, idx][0].float
    ()) / 2.0
obj_center_y = (bbox_tensor[0, idx][3].
    float() + bbox_tensor[0, idx][1].float
    ()) / 2.0
yolocell_center_i = cell_row_indx * 40 +
    float(40) / 2.0
yolocell_center_j = cell_col_indx * 40 +
    float(40) / 2.0
del_x = (obj_center_x.float() -
    yolocell_center_j.float()) / 40
del_y = (obj_center_y.float() -
    yolocell_center_i.float()) / 40
class_label_of_object = bbox_label_tensor
    [0, idx].item()
#shape
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
yolo_vector = torch.FloatTensor([0, del_x
    .item(), del_y.item(), bh.item(), bw.
    item(), 0, 0, 0])
yolo_vector[0] = 1
yolo_vector[5 + class_label_of_object] =
    1
yolo_cell_index = cell_row_indx.item() *
    num_cells_image_width + cell_col_indx.
```

```
            item()
        yolo_tensor[0, yolo_cell_index,
            anch_box_index] = yolo_vector
        yolo_tensor_aug = torch.zeros(batch_size,
             num_yolo_cells, num_anchor_boxes, 9).
            float().to(device)
        yolo_tensor_aug[:, :, :, :-1] =
            yolo_tensor

    for icx in range(num_yolo_cells):
        for iax in range(num_anchor_boxes):
            if yolo_tensor_aug[0, icx, iax, 0] ==
                 0:
                yolo_tensor_aug[0, icx, iax, -1]
                    = 1

    optimizer.zero_grad()
    outputs = net(im_tensor)
    predictions_aug = outputs.view(batch_size,
        num_yolo_cells, num_anchor_boxes, 9)
    loss = torch.tensor(0.0, requires_grad=True).
        float().to(device)
    for icx in range(num_yolo_cells):
        for iax in range(num_anchor_boxes):
            pred_yolo_vector = predictions_aug[0,
                 icx, iax]
            target_yolo_vector = yolo_tensor_aug
                [0, icx, iax]
            #BCE for presence
            object_presence = torch.nn.Sigmoid()(
                torch.unsqueeze(pred_yolo_vector
                [0], dim=0))
            target_for_prediction = torch.
                unsqueeze(target_yolo_vector[0],
                dim=0)
            bceloss = criterion1(object_presence,
                 target_for_prediction)
            running_loss_presence += bceloss.item
                ()
            loss = loss + bceloss
            #MSE for regression
            pred_regression_vec =
                pred_yolo_vector[1:5]
            pred_regression_vec = torch.unsqueeze
                (pred_regression_vec, dim=0)
            target_regression_vec = torch.
```

```
                        unsqueeze(target_yolo_vector[1:5],
                         dim=0)
                    regression_loss = criterion2(
                        pred_regression_vec,
                        target_regression_vec)
                    running_loss_regression+=
                        regression_loss.item()
                    loss += regression_loss
                    #CrossEntropy 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)
                    running_loss_labeling +=
                        class_labeling_loss.item()
                    loss += class_labeling_loss
            loss.backward()
            optimizer.step()
            running_loss += loss.item()
            if (i+1) % 500 == 0:
                print("[epoch: %d, batch: %5d] loss: %.3f
                    " % (epoch+1, i+1, running_loss/500))
                Loss_presence.append(
                    running_loss_presence/500)
                Loss_regression.append(
                    running_loss_regression/500)
                Loss_labeling.append(
                    running_loss_labeling/500)
                print(running_loss_presence/500,
                    running_loss_regression/500,
                    running_loss_labeling/500)
                running_loss_presence = 0.0
                running_loss_labeling = 0.0
                running_loss_regression = 0.0
                print("----------time executing for epoch
                    %s : %s seconds----------" % (epoch +
                    1, (time.time() - start_time)))
    torch.save(net, net_save_path)

    return Loss_presence, Loss_regression, Loss_labeling
```

```
def decoder_for_pbc(category_id):
    # dict = {59:0,6:1,17:2}
    dict = {0: 'pizza', 1: 'bus', 2: 'cat'}
    return dict[category_id]



if __name__ == '__main__':
    if torch.cuda.is_available() == True:
        device = torch.device("cuda:0")
    else:
        device = torch.device("cpu")
    # train_dataset = MyDataset(True)

    # train_data_loader = torch.utils.data.DataLoader(
        dataset=train_dataset, batch_size=4, shuffle=False
        , num_workers=4)
    # train_data_loader = torch.utils.data.DataLoader(
        dataset=train_dataset, batch_size=4, shuffle=True,
         num_workers=4)
    # net = HW6Net()
    net = torch.load('net1.pth')
    # loss_BC, loss_MSE,loss_Crossentropy = run_training(
        net,train_data_loader,"net1.pth")
    # num_layers = len(list(net.parameters()))
    # print(num_layers)
    # print(loss_BC,loss_MSE,loss_Crossentropy)

    # plt.figure()
    # plt.plot(loss_BC, label = "Presence Training Loss")
    # plt.savefig('loss_BCE_labeling.jpg')
    # plt.figure()
    # plt.plot(loss_MSE, label = "Regression Training
        Loss")
    # plt.savefig('loss_MSE_regression.jpg')
    # plt.figure()
    # plt.plot(loss_Crossentropy, label = "Labeling
        Training Loss")
    # plt.savefig('loss_CE_Labeling.jpg')


    val_dataset = MyDataset(False)
    val_data_loader = torch.utils.data.DataLoader(dataset
       =val_dataset, batch_size=1, shuffle=False,
```

```
    num_workers=4)
num_anchor_boxes = 5
num_yolo_cells = 36
max_obj_num = 5
batch_size = 1
count = 0
for i, data in enumerate(val_data_loader):
    yolo_tensor = torch.zeros(batch_size,
        num_yolo_cells, num_anchor_boxes, 8)
    im_tensor, bbox_label_tensor, bbox_tensor,
        num_objects_in_image = data
    im_tensor = im_tensor.to(device)
    bbox_tensor = bbox_tensor.to(device)
    bbox_label_tensor = bbox_label_tensor.to(device)
    yolo_tensor = yolo_tensor.to(device)
    output = net(im_tensor)
    outputs = output.view(num_yolo_cells,
        num_anchor_boxes, 9)
    print(outputs.shape)
    icx_2_best_anchor_box = {ic: None for ic in range
        (36)}
    for icx in range(outputs.shape[0]):
        cell_predi = outputs[icx]
        prev_best = 0
        for anchor_bdx in range(cell_predi.shape[0]):
            if cell_predi[anchor_bdx][0] > cell_predi
                [prev_best][0]:
                prev_best = anchor_bdx
        best_anchor_box_icx = prev_best
        icx_2_best_anchor_box[icx] =
            best_anchor_box_icx
    sorted_icx_to_box = sorted(icx_2_best_anchor_box,
        key=lambda x: outputs[x,
       icx_2_best_anchor_box[x]][0].item(), reverse=
       True)
    retained_cells = sorted_icx_to_box[:5]
    objects_detected = []
    predicted_bboxes = []
    predicted_labels_for_bboxes = []
    predicted_label_index_vals = []
    for icx in retained_cells:
        pred_vec = outputs[icx, icx_2_best_anchor_box
            [icx]]
        class_labels_predi = pred_vec[-4:]
        class_labels_probs = torch.nn.Softmax(dim=0)(
            class_labels_predi)
```

```
        class_labels_probs = class_labels_probs[:-1]
        if torch.all(class_labels_probs < 0.2):
            predicted_class_label = None
            # print('less')
        else:
            count+=1
            best_predicted_class_index = (
                class_labels_probs ==
                class_labels_probs.max())
            best_predicted_class_index = torch.
                nonzero(best_predicted_class_index,
                as_tuple=True)
            predicted_label_index_vals.append(
                best_predicted_class_index[0].item())
            predicted_class_label = decoder_for_pbc(
                best_predicted_class_index[0].item())
            predicted_labels_for_bboxes.append(
                predicted_class_label)
            print(predicted_bboxes)
            pred_regression_vec = pred_vec[1:5].cpu()
            del_x, del_y = pred_regression_vec[0],
                pred_regression_vec[1]
            h, w = pred_regression_vec[2],
                pred_regression_vec[3]
            h *= 40
            w *= 40
            cell_row_index = icx // 6
            cell_col_index = icx % 6
            bb_center_x = cell_col_index * 40 + 40 /
                2 + del_x * 40
            bb_center_y = cell_row_index * 40 + 40 /
                2 + del_y * 40
            bb_top_left_x = int(bb_center_x - w /
                2.0)
            bb_top_left_y = int(bb_center_y - h /
                2.0)
            predicted_bboxes.append([bb_top_left_x,
                bb_top_left_y, int(w), int(h)])
print(predicted_bboxes)
print(count)
fig = plt.figure(figsize=[12, 12])
ax = fig.add_subplot(111)
display_scale = 2
new_im_tensor = torch.nn.functional.interpolate(
    im_tensor, scale_factor=display_scale, mode='
    bilinear', align_corners=False)
```

```
ax.imshow(np.transpose(torchvision.utils.
    make_grid(new_im_tensor, normalize=True,
    padding=3, pad_value=255).cpu(), (1, 2, 0)))
for i, bbox_pred in enumerate(predicted_bboxes):
    x, y, w, h = np.array(bbox_pred)
    x, y, w, h = [item * display_scale for item
        in (x, y, w, h)]
    rect = cv2.rectangle((x, y), w, h, (36, 255,
        12), 2)
    ax.annotate(predicted_labels_for_bboxes[i], (
        x, y - 1), color='red', weight='bold',
        fontsize=10 * display_scale)
plt.savefig(str(i)+'.jpg')
if i==7:
    break
```