ECE695DL: Homework 5

Pranab Dash

28 March 2021

```
#training
import argparse
from pycocotools.coco import COCO
import cv2
import torch
import torch.nn as nn
from torch.utils.data import DataLoader, Dataset
import torchvision.transforms as tvt
import torch.nn.functional as F
from torchsummary import summary
import numpy as np
import os
from PIL import Image
import matplotlib.pyplot as plt
from itertools import combinations
import time

parser = argparse.ArgumentParser ( description ='HW05 COCO
↪  training')
parser.add_argument ( '--training_dataset'                    ,
↪  required =True , type=str)
parser.add_argument ( '--training_annotation'                 ,
↪  required =True , type=str)
parser.add_argument ( '--validation_dataset'                  ,
↪  required =True , type=str)
parser.add_argument ( '--validation_annotation'               ,
↪  required =True , type=str)
args, args_other = parser.parse_known_args ()

dtype = torch.float32
```

```
def get_image_ids ( coco, class_index_list, num_objects,
↪  strict=True):
        catagory    = { catagory_id:coco.loadCats (
        ↪  catagory_id)[0]['name'] for catagory_id in
        ↪  coco.getCatIds ()}
        rcatagory   = { coco.loadCats (
        ↪  catagory_id)[0]['name']:catagory_id for catagory_id
        ↪  in coco.getCatIds ()}

        image_ids = []
        for num_object in range ( 1, num_objects+1):

                all_image_ids = []
                comb = combinations( class_index_list,
                ↪  num_object)
                for i in list(comb):
                        cat_id = np.array (i)
                        all_image_ids += coco.getImgIds(
                        ↪  catIds=cat_id)
                image_datas = coco.loadImgs  ( ids=all_image_ids)
#                print ( '===> ', num_object, ' : ', len (
↪  image_datas))
                count = 0
                for image_data in image_datas:
                        image_id = image_data['id']
                        image_annotations = coco.loadAnns (
                        ↪  coco.getAnnIds ( imgIds=image_id))

                        include = True
                        for image_annotation in
                        ↪  image_annotations:
                                area =
                                ↪  image_annotation['bbox'][2]*image_annotation['bbox'][3]
#                                print ( area)
                                if
                                ↪  image_annotation['category_id']
                                ↪  not in class_index_list or
                                ↪  area < 0.2*0.2*640*640:
                                        include = False
                                        break
                        if not include:
                                continue
                        if strict:
                            if len ( image_annotations) ==
                            ↪  num_objects:
                                image_ids.append ( image_id)
                        else:
                            if len ( image_annotations) <=
                            ↪  num_objects and len (
                            ↪  image_annotations) > 0:
                                image_ids.append ( image_id)
```

```
#                print ( '===> ', number of images, ' : ', len (
↪  image_ids))
        return image_ids

class block ( nn.Module):
        def __init__ ( self, in_channels, out_channels,
        ↪  kernel_size, stride):
                super ( block, self).__init__()

                self.conv = torch.nn.Conv2d (
                ↪  in_channels=in_channels,
                ↪  out_channels=out_channels,
                ↪  kernel_size=kernel_size,
                                stride=stride, padding=1,
                                ↪  dilation=1, groups=1,
                                ↪  bias=True,
                                ↪  padding_mode='zeros')
                self.convi = torch.nn.Conv2d (
                ↪  in_channels=out_channels,
                ↪  out_channels=out_channels,
                ↪  kernel_size=kernel_size,
                                stride=1, padding=1, dilation=1,
                                ↪  groups=1, bias=True,
                                ↪  padding_mode='zeros')
                self.bnorm = torch.nn.BatchNorm2d (
                ↪  num_features=out_channels,
                                eps=1e-05, momentum=0.1,
                                ↪  affine=True,
                                ↪  track_running_stats=True)
                self.conv1x1 = torch.nn.Conv2d (
                ↪  in_channels=in_channels,
                ↪  out_channels=out_channels, kernel_size=1,
                                stride=1, padding=0, dilation=1,
                                ↪  groups=1, bias=True,
                                ↪  padding_mode='zeros')
                self.relu = torch.nn.LeakyReLU (
                ↪  negative_slope=0.01, inplace=False)
                self.maxp = torch.nn.MaxPool2d ( kernel_size=2,
                ↪  stride=None, padding=0, dilation=1,
                ↪  return_indices=False, ceil_mode=False)
                self.in_channels = in_channels
                self.out_channels = out_channels

        def forward ( self, x):
                x_identity = x

                x = self.conv (x)
                x = self.bnorm (x)
                x = self.relu (x)

                x = self.convi (x)
```

```
                x = self.bnorm (x)
                if self.in_channels != self.out_channels:
                        x_identity = self.maxp ( x_identity)
                        x_identity = self.conv1x1 ( x_identity)
                        x += x_identity
                else:
                        x += x_identity
                x = self.relu (x)

                return x

class network ( nn.Module):
        def __init__ ( self, num_classes, num_objects):
                super ( network, self).__init__()
                self.conv = torch.nn.Conv2d ( in_channels=3,
                ↪  out_channels=64, kernel_size=7, stride=2,
                ↪  padding=3)
                self.bloc64    = block (   in_channels=64,
                ↪  out_channels=64, kernel_size=3, stride=1)
                self.bloc128i  = block (   in_channels=64,
                ↪  out_channels=128, kernel_size=3, stride=2)
                self.bloc128   = block (  in_channels=128,
                ↪  out_channels=128, kernel_size=3, stride=1)
                self.bloc256i  = block (  in_channels=128,
                ↪  out_channels=256, kernel_size=3, stride=2)
                self.bloc256   = block (  in_channels=256,
                ↪  out_channels=256, kernel_size=3, stride=1)
                self.bloc512i  = block (  in_channels=256,
                ↪  out_channels=512, kernel_size=3, stride=2)
                self.bloc512   = block (  in_channels=512,
                ↪  out_channels=512, kernel_size=3, stride=1)
                self.bloc1024i = block (  in_channels=512,
                ↪  out_channels=1024, kernel_size=3, stride=2)
                self.bloc1024  = block ( in_channels=1024,
                ↪  out_channels=1024, kernel_size=3, stride=1)
                self.num_classes = num_classes
                self.num_objects = num_objects
                fc_output_size = 65536 # int ( 131072/4) # 262144
                fc_output_size = int ( 32768/4*4) # int (
                ↪  131072/4) # 262144
                self.fc = torch.nn.Linear ( fc_output_size, 1000)
                self.fc_exists = torch.nn.Linear ( 1000,
                ↪  self.num_objects)
                self.fc_bbox   = torch.nn.Linear ( 1000,
                ↪  4*self.num_objects)

                self.fc_cat_up   = torch.nn.Linear ( 1000,
                ↪  self.num_objects*self.num_classes)
                self.fc_cat_down = torch.nn.Linear (
                ↪  self.num_objects*self.num_classes,
                ↪  self.num_classes)
```

```
#                self.fc_cat    = torch.nn.Linear ( 1000,
↪  self.num_objects)

                self.relu = torch.nn.LeakyReLU (
                ↪  negative_slope=0.01, inplace=False)
                self.softmax = torch.nn.Softmax ()
                self.sig = nn.Sigmoid()
                self.maxp = torch.nn.MaxPool2d ( kernel_size=2,
                ↪  stride=None, padding=0, dilation=1,
                ↪  return_indices=False, ceil_mode=False)

                self.conv15 = torch.nn.Conv2d ( in_channels=512,
                ↪  out_channels=128, kernel_size=15, stride=1,
                ↪  padding=0)
                self.conv13 = torch.nn.Conv2d ( in_channels=512,
                ↪  out_channels=128, kernel_size=13, stride=1,
                ↪  padding=0)

                self.conv7 = torch.nn.Conv2d ( in_channels=512,
                ↪  out_channels=128, kernel_size=7, stride=1,
                ↪  padding=0)
                self.conv5 = torch.nn.Conv2d ( in_channels=512,
                ↪  out_channels=128, kernel_size=5, stride=1,
                ↪  padding=0)
                self.conv3 = torch.nn.Conv2d ( in_channels=512,
                ↪  out_channels=128, kernel_size=3, stride=1,
                ↪  padding=0)
                self.conv1 = torch.nn.Conv2d ( in_channels=512,
                ↪  out_channels=128, kernel_size=1, stride=1,
                ↪  padding=0)

                self.fc7 = torch.nn.Linear ( 512, 1000)
                self.fc5 = torch.nn.Linear ( 2048, 1000)
                self.fc3 = torch.nn.Linear ( 4608, 1000)
                self.fc1 = torch.nn.Linear ( 8192, 1000)

        def forward ( self, x):
                x = self.conv (x)
                ##
                x = self.maxp ( x)
                ##
                m = 0
                for i in range ( 5-m):
                        x = self.bloc64 (x)
                x = self.bloc128i (x)
                for i in range ( 4-m):
                        x = self.bloc128 (x)
                x = self.bloc256i (x)
                for i in range ( 4-m):
                        x = self.bloc256 (x)
                x = self.bloc512i (x)
```

```
                for i in range ( 4-m):
                        x = self.bloc512 (x)

##                print ( 'x =', x.size())
                x7 = self.conv7 (x)
                x5 = self.conv5 (x)
                x3 = self.conv3 (x)
                x1 = self.conv1 (x)

                fc_output_size = 1
                for index in range ( 1, len ( x7.size())):
                        fc_output_size *= x7.size()[index]
                x7 = x7.view ( -1, fc_output_size)
                x7 = self.fc7 (x7)
                fc_output_size = 1
                for index in range ( 1, len ( x5.size())):
                        fc_output_size *= x5.size()[index]
                x5 = x5.view ( -1, fc_output_size)
                x5 = self.fc5 (x5)
                fc_output_size = 1
                for index in range ( 1, len ( x3.size())):
                        fc_output_size *= x3.size()[index]
                x3 = x3.view ( -1, fc_output_size)
                x3 = self.fc3 (x3)
                fc_output_size = 1
                for index in range ( 1, len ( x1.size())):
                        fc_output_size *= x1.size()[index]
                x1 = x1.view ( -1, fc_output_size)
                x1 = self.fc1 (x1)

                '''
                print ( 'x7 =', x7.size())
                print ( 'x5 =', x5.size())
                print ( 'x3 =', x3.size())
                print ( 'x1 =', x1.size())
                print ( 'x1 =', x_res.size())
                '''

                x_res = x7 + x5 + x3 +x1

####                x = x.view ( -1, fc_output_size)

                ##
#                x = self.bloc1024i (x)
#                for i in range ( 4):
#                        x = self.bloc1024 (x)
                ##

#                print ( 'x1 = ', x.size())
#                x    = self.maxp (x)
#                print ( 'x2 = ', x.size())
```

```
                fc_output_size = 1
                for index in range ( 1, len ( x.size())):
                        fc_output_size *= x.size()[index]
#                print ( x.size())
#                print ( fc_output_size)
                '''
                x = x.view ( -1, fc_output_size)
                x_fwd  = self.fc (x)
                '''
###                x = x.view ( -1, fc_output_size)

                x_fwd  = x_res

                x_exists = self.fc_exists (x_fwd)
                x_exists = self.sig ( x_exists)

                x_bbox   = self.fc_bbox (x_fwd)
                x_bbox   = self.relu ( x_bbox)

####                x_cat_up = self.fc (x)
####                x_cat_up = self.fc_cat_up (x_cat_up)
                x_cat_up = self.fc_cat_up (x_res)
                x_cats = []
                for index in range ( self.num_objects):
                        x_cat    =
                        ↪  x_cat_up[:,self.num_classes*index:self.num_classes*index+self.num_classes]
                        x_cat    = self.softmax (x_cat)
                        x_cats.append ( x_cat)

                return x_exists, x_bbox, x_cats

scale = 0.4
class dataset_class ( Dataset):
        def __init__ ( self,  imagenet_root, coco, image_ids,
        ↪  class_index_list, num_objects, transform):
                self.transform = transform

                self.imagenet_root = imagenet_root
                self.filename = []
                self.exist    = []
                self.label    = []
                self.bbox     = []

                image_datas = coco.loadImgs  ( ids=image_ids)
                for image_data in image_datas:
                        image_id = image_data['id']
                        filename = image_data['file_name']
#                        print ( self.imagenet_root + '/' +
↪  filename)
                        image_annotations = coco.loadAnns (
                        ↪  coco.getAnnIds ( imgIds=image_id))
```

```
#                        if len ( image_annotations) !=
↪  num_objects:
#                                continue

                       labels = np.zeros ( num_objects) # ,
                       ↪  dtype=int)
                       bboxs = np.zeros ( 4*num_objects)
                       exists = np.zeros ( num_objects)
                       for index in range ( len (
                       ↪  image_annotations)):
                               image_ann =
                               ↪  image_annotations[index]

                               category_id =
                               ↪  image_ann['category_id']
                               class_index = np.where ( np.array
                               ↪  ( class_index_list) ==
                               ↪  category_id)
#                               print ( category_id, ' ',
↪  class_index_list, ' ', class_index)

                               bbox = scale*np.array (
                               ↪  image_ann['bbox'])

                               exists[index] = 1
                               bboxs[4*index:4*index+4] = bbox
                               labels[index] = class_index[0] #
                               ↪  category_id

                       self.filename.append ( filename)
                       self.exist.append    ( exists)
                       self.label.append    ( labels)
                       self.bbox.append     ( bboxs)

       def __len__ ( self) :
               return ( len ( self.filename))
       def __getitem__ ( self, idx) :
               # Load the image
               image = Image.open( self.imagenet_root + '/' +
               ↪  self.filename[idx])
               data = self.transform ( image)
               image.close ()

               exist = self.exist[idx]
               label = self.label[idx]
               bbox = self.bbox[idx]

               return data, exist, label, bbox,
               ↪  self.imagenet_root + '/' + self.filename[idx]
```

```
device = torch.device("cuda" if torch.cuda.is_available() else
↪  "cpu") # PyTorch v0.4.0
# device = torch.device("cpu")

coco_training   = COCO ( args.training_annotation)
coco_validation = COCO ( args.validation_annotation)
catagory    = { index:coco_training.loadCats ( index)[0]['name']
↪  for index in coco_training.getCatIds ()}
rcatagory   = { coco_training.loadCats ( index)[0]['name']:index
↪  for index in coco_training.getCatIds ()}
# print ( catagory)

## class_list = list ( rcatagory.keys()) # [ 'cat', 'dog']
## class_list = [ 'bird', 'cat', 'dog', 'horse', 'sheep', 'cow',
↪  'elephant', 'bear', 'zebra', 'giraffe']
class_list = [ 'cat', 'dog', 'cow', 'zebra', 'giraffe']
## class_list = [ 'cat', 'dog']
class_index_list = [ rcatagory[class_name] for class_name in
↪  class_list]

num_objects = 3
## num_objects = 1
image_ids = get_image_ids ( coco_training, class_index_list,
↪  num_objects, False) # True)

transform = tvt.Compose ( [ tvt.ToTensor (), tvt.Normalize ( (
↪  0.5, 0.5, 0.5), ( 0.5, 0.5, 0.5))])

train_dataset = dataset_class ( args.training_dataset,
↪  coco_training, image_ids, class_index_list, num_objects,
↪  transform)
train_data_loader = torch.utils.data.DataLoader (
↪  dataset=train_dataset,
        batch_size=10, shuffle=True, num_workers=4)

val_image_ids = get_image_ids ( coco_validation,
↪  class_index_list, num_objects, False)
val_dataset = dataset_class ( args.validation_dataset,
↪  coco_validation, val_image_ids, class_index_list,
↪  num_objects, transform)
val_data_loader = torch.utils.data.DataLoader (
↪  dataset=val_dataset,
        batch_size=10, shuffle=True, num_workers=4)

def calculate_jaccard_distance ( bbox1, bbox2):
        bbox1_x_min, bbox1_y_min, bbox1_width, bbox1_height =
        ↪  bbox1
        bbox2_x_min, bbox2_y_min, bbox2_width, bbox2_height =
        ↪  bbox2

        bbox1_x_max = bbox1_x_min + bbox1_width
```

```
        bbox1_y_max = bbox1_y_min + bbox1_height
        bbox2_x_max = bbox2_x_min + bbox2_width
        bbox2_y_max = bbox2_y_min + bbox2_height

        bbox1_area = bbox1_width*bbox1_height
        bbox2_area = bbox2_width*bbox2_height

        i_x1        = max ( bbox1_x_min, bbox2_x_min)
        i_x2        = min ( bbox1_x_max, bbox2_x_max)
        i_y1        = max ( bbox1_y_min, bbox2_y_min)
        i_y2        = min ( bbox1_y_max, bbox2_y_max)
        intersection_area        = max ( i_x2 - i_x1, 0) * max (
        ↪  i_y2 - i_y1, 0)
        union_area                = bbox1_area + bbox2_area -
        ↪  intersection_area

        jaccard_distance        = float (
        ↪  intersection_area)/float ( union_area)

        return jaccard_distance

def annotate_bound_box ( image, bbox, label, color,
↪  label_pos='top'):
        font                = cv2.FONT_HERSHEY_SIMPLEX
        fontScale        = 0.5
        thickness        = 1

        x_min, y_min, width, height = bbox
        x_max         = x_min + width
        y_max         = y_min + height

        x_min        = max ( x_min, 0)
        y_min        = max ( y_min, 0)
        x_max        = min ( x_max, 256)
        y_max        = min ( y_max, 256)

        start_point        = ( int ( x_min), int ( y_min))
        end_point        = ( int ( x_max), int ( y_max))
        image = cv2.rectangle ( image, start_point, end_point,
        ↪  color, thickness)

        if label_pos=='top':
                pos = ( int ( x_min), int ( y_min+10))
        elif label_pos=='bottom':
                pos = ( int ( x_min), int ( y_max))
        image = cv2.putText   ( image, label, pos,
                                font, fontScale, color,
                                ↪  thickness, cv2.LINE_AA)

        return image
```

```
def dump_train_output ( filename, exists, bboxs, labels,
↪  pred_exists, pred_bboxs, pred_labels, dumppath):
        n        = 5
        n_index        = 0
        plt.figure ( figsize=(20, 4))
        green        = [ 0, 255, 0]
        red        = [ 255, 0, 0]
        bboxs                = bboxs.cpu().detach().numpy()
        exists                = exists.cpu().detach().numpy()
        labels                = labels.cpu().detach().numpy()
        pred_exists        = pred_exists.cpu().detach().numpy()
        pred_bboxs        = pred_bboxs.cpu().detach().numpy()
        for bbox_index in range ( len ( exists[0])):
                pred_labels[bbox_index]        =
                ↪  pred_labels[bbox_index].cpu().detach().numpy()
        threshold        = 0.5
        for index in range ( len ( filename)):
                if n_index < n:
                        image = cv2.imread ( filename[index])
                        image = cv2.cvtColor ( image,
                        ↪  cv2.COLOR_BGR2RGB)
                for bbox_index in range ( len ( exists[index])):
                        if n_index < n:
                                if exists[index][bbox_index] >
                                ↪  threshold:
                                        l =
                                        ↪  class_list[labels[bbox_index][index]]
                                        image =
                                        ↪  annotate_bound_box (
                                        ↪  image,
                                        ↪  bboxs[index][4*bbox_index:4*(bbox_index+1)],
                                        ↪  l, green, 'top')
                                if pred_exists[index][bbox_index]
                                ↪  > threshold:
                                        l =
                                        ↪  class_list[np.argmax(pred_labels[bbox_index][index])]
                                        image =
                                        ↪  annotate_bound_box (
                                        ↪  image,
                                        ↪  pred_bboxs[index][4*bbox_index:4*(bbox_index+1)],
                                        ↪  l, red, 'bottom')
                if n_index < n:
                        plt.subplot ( 1, n, n_index + 1)
                        plt.imshow ( image)
                        n_index += 1
                        if not n_index < n:
                                plt.savefig ( dumppath,
                                ↪  bbox_inches='tight')
                                plt.close ()
def get_loss ( network, data_loader, dumppath):
        l1s = []
```

```
l2s = []
l3s = []
cat_criterion        = torch.nn.CrossEntropyLoss()
reg_criterion        = torch.nn.MSELoss()
y_true = np.array ( [])
y_pred = np.array ( [])
first=True
threshold=0.5
iou = []
with torch.no_grad():
        for x, y_exists, y_labels, y_bboxs, y_filename in
        ↪  data_loader:

                y_exists               = y_exists.to (
                ↪  device, dtype=dtype)
                y_labels = y_labels.T
                y_labels = y_labels.to ( device,
                ↪  dtype=torch.long)
                y_bboxs                = y_bboxs.to (
                ↪  device, dtype=dtype)

                x = x.to ( device=device)
                predicted_exists, predicted_bboxs,
                ↪  predicted_labels  = network (x)

                total_l1 = 0.0
                for index in range ( num_objects):
                        l1 = cat_criterion (
                        ↪  predicted_labels[index],
                        ↪  y_labels[index])
                        total_l1 += l1.item()
                l1s.append ( total_l1)
                for index in range ( len ( y_filename)):
                        for bbox_index in range ( len (
                        ↪  y_exists[index])):
                                if
                                ↪  y_exists[index][bbox_index]
                                ↪  > threshold:
                                        l =
                                        ↪  y_labels[bbox_index][index].cpu().numpy()
                                        y_true =
                                        ↪  np.concatenate
                                        ↪  ( [ y_true,
                                        ↪  [l]])
                                        l =
                                        ↪  np.argmax(predicted_labels[bbox_index]
                                        [index].cpu().numpy())
                                        y_pred =
                                        ↪  np.concatenate
                                        ↪  ( [ y_pred,
                                        ↪  [l]])
```

```
                        for bbox_index in range ( len (
                        ↪  y_exists[index])):
                                if y_exists[index][bbox_index] >
                                ↪  threshold:
                                        u =
                                        ↪  calculate_jaccard_distance(
                                        ↪  y_bboxs[index][4*bbox_index:4*(bbox_index+1)],
                                        predicted_bboxs[index][4*bbox_index:4*(bbox_index+1)])
                                        iou.append ( u)

                        l2 = reg_criterion ( predicted_bboxs,
                        ↪  y_bboxs)
                        l2s.append ( l2.item())

                        l3 = reg_criterion ( predicted_exists,
                        ↪  y_exists)
                        l3s.append ( l3.item())
                        accuracy = (y_true ==
                        ↪  y_pred).sum()/y_true.shape[0]
                        if first:
                                dump_train_output ( y_filename,
                                ↪  y_exists, y_bboxs, y_labels,
                                ↪  predicted_exists,
                                ↪  predicted_bboxs,
                                ↪  predicted_labels, dumppath)
                                first=False
                return np.mean ( l1s), np.mean ( l2s), np.mean (
                ↪  l3s), accuracy, np.mean ( iou)

# print ( len ( train_dataset))

num_classes = len ( class_list)
model = network ( num_classes=num_classes,
↪  num_objects=num_objects)

epochs = 50
def run_code_for_training ( net):
        net                = net.to ( device)
        cat_criterion        = torch.nn.CrossEntropyLoss()
        reg_criterion        = torch.nn.MSELoss()
#        optimizer        = torch.optim.SGD ( net.parameters(),
↪  lr=1e-6, momentum=0.9, weight_decay=0.5)
        optimizer        = torch.optim.SGD ( net.parameters(),
        ↪  lr=1e-6, momentum=0.9)
        l1s = []
        l2s = []
        l3s = []
        for epoch in range ( epochs):
                start = time.process_time()
                running_loss = 0.0
                for i, data in enumerate ( train_data_loader):
```

```
            inputs, exists, labels, bboxs,
            ↪  filename        = data
            inputs                = inputs.to (
            ↪  device, dtype=dtype)
            exists                = exists.to (
            ↪  device, dtype=dtype)
            labels = labels.T
            labels = labels.to ( device,
            ↪  dtype=torch.long)
            bboxs                 = bboxs.to ( device,
            ↪  dtype=dtype)

            optimizer.zero_grad()
            outputs                = net ( inputs)
            predicted_exists = outputs[0]
            predicted_labels = outputs[2]
            predicted_bboxs  = outputs[1]

            if (
            ↪  torch.isnan(predicted_exists).any()):
                    print ( "exists Nan Error")
            for index in range ( num_objects):
                    if (
                    ↪  torch.isnan(predicted_labels[index]).any()):
                            print ( "labels Nan
                            ↪  Error")
            if ( torch.isnan(predicted_bboxs).any()):
                    print ( "bboxs Nan Error")

            total_l1 = 0.0
            for index in range ( num_objects):
                    l1 = cat_criterion (
                    ↪  predicted_labels[index],
                    ↪  labels[index])
                    l1.backward( retain_graph=True)
                    total_l1 += l1.item()
            l1s.append ( total_l1)

            l2 = reg_criterion ( predicted_bboxs,
            ↪  bboxs)
            l2.backward( retain_graph=True)
            l2s.append ( l2.item())

            l3 = reg_criterion ( predicted_exists,
            ↪  exists)
            l3.backward()
            l3s.append ( l3.item())

            optimizer.step()

    if (epoch+1) % 25 == 0:
```

```
                        torch.save ( net,
                        ↪  'model/network_epoch_%03d.pth' %
                        ↪  (epoch))

                l1t, l2t, l3t, acct, iout = get_loss ( net,
                ↪  train_data_loader,
                ↪  './epoch/Epoch_%03d__Train.png' % (epoch))
                l1v, l2v, l3v, accv, iouv = get_loss ( net,
                ↪  val_data_loader,
                ↪  './epoch/Epoch_%03d__Val.png' % (epoch))
                print ( 'Epoch %3d : Train [ %.6f %.6f %.6f
                ↪  (%.2f) %.6f] Val [ %.6f %.6f %.6f (%.2f)
                ↪  %.6f]' % ( epoch, l1t, l2t, l3t, 100*acct,
                ↪  iout, l1v, l2v, l3v, 100*accv, iouv))
                print ( 'Epoch %3d : Traing Loss' % ( epoch),
                ↪  end='')
                print ( ' [Cat : %.6f ] ' % ( np.mean ( l1s)),
                ↪  end='')
                print ( ' [Reg : %.6f, %.6f] ' % ( np.mean (
                ↪  l2s), np.mean ( l3s)), end='')
                print ( ' Time %.3f s' % ( time.process_time() -
                ↪  start), end='')
                print ( '')
run_code_for_training ( model)
torch.save ( model, 'net.pth')

## model = network( 2, 3).to(device)
## summary ( model, ( 3, 256, 256))
```

```
#validation
import argparse
from pycocotools.coco import COCO
import cv2
import torch
import torch.nn as nn
from torch.utils.data import DataLoader, Dataset
import torchvision.transforms as tvt
import torch.nn.functional as F
from torchsummary import summary
import numpy as np
import os
from PIL import Image
import matplotlib.pyplot as plt
from itertools import combinations
import time

parser = argparse.ArgumentParser ( description ='HW05 COCO
↪  training')
parser.add_argument ( '--training_dataset'                    ,
↪  required =True , type=str)
```

```
parser.add_argument ( '--training_annotation'                ,
↪  required =True , type=str)
parser.add_argument ( '--validation_dataset'                 ,
↪  required =True , type=str)
parser.add_argument ( '--validation_annotation'              ,
↪  required =True , type=str)
parser.add_argument ( '--network_model_path'              ,
↪  required =True , type=str)
args, args_other = parser.parse_known_args ()

dtype = torch.float32

def get_image_ids ( coco, class_index_list, num_objects,
↪  strict=True):
        catagory    = { catagory_id:coco.loadCats (
        ↪  catagory_id)[0]['name'] for catagory_id in
        ↪  coco.getCatIds ()}
        rcatagory   = { coco.loadCats (
        ↪  catagory_id)[0]['name']:catagory_id for catagory_id
        ↪  in coco.getCatIds ()}

        image_ids = []
        for num_object in range ( 1, num_objects+1):

                all_image_ids = []
                comb = combinations( class_index_list,
                ↪  num_object)
                for i in list(comb):
                        cat_id = np.array (i)
                        all_image_ids += coco.getImgIds(
                        ↪  catIds=cat_id)
                image_datas = coco.loadImgs  ( ids=all_image_ids)
#                print ( '===> ', num_object, ' : ', len (
↪  image_datas))
                count = 0
                for image_data in image_datas:
                        image_id = image_data['id']
                        image_annotations = coco.loadAnns (
                        ↪  coco.getAnnIds ( imgIds=image_id))

                        include = True
                        for image_annotation in
                        ↪  image_annotations:
                                area =
                                ↪  image_annotation['bbox'][2]*
                                ↪  image_annotation['bbox'][3]
#                                print ( area)
                                if image_annotation['category_id
                                ↪  '] not in class_index_list
                                ↪  or area < 0.2*0.2*640*640:
                                        include = False
```

```
                                    break
                        if not include:
                                continue
                        if strict:
                            if len ( image_annotations) ==
                            ↪  num_objects:
                                image_ids.append ( image_id)
                        else:
                            if len ( image_annotations) <=
                            ↪  num_objects and len (
                            ↪  image_annotations) > 0:
                                image_ids.append ( image_id)
#                print ( '===> ', number of images, ' : ', len (
↪  image_ids))
        return image_ids

class block ( nn.Module):
        def __init__ ( self, in_channels, out_channels,
        ↪  kernel_size, stride):
                super ( block, self).__init__()

                self.conv = torch.nn.Conv2d (
                ↪  in_channels=in_channels,
                ↪  out_channels=out_channels,
                ↪  kernel_size=kernel_size,
                                stride=stride, padding=1,
                                ↪  dilation=1, groups=1,
                                ↪  bias=True,
                                ↪  padding_mode='zeros')
                self.convi = torch.nn.Conv2d (
                ↪  in_channels=out_channels,
                ↪  out_channels=out_channels,
                ↪  kernel_size=kernel_size,
                                stride=1, padding=1, dilation=1,
                                ↪  groups=1, bias=True,
                                ↪  padding_mode='zeros')
                self.bnorm = torch.nn.BatchNorm2d (
                ↪  num_features=out_channels,
                                eps=1e-05, momentum=0.1,
                                ↪  affine=True,
                                ↪  track_running_stats=True)
                self.conv1x1 = torch.nn.Conv2d (
                ↪  in_channels=in_channels,
                ↪  out_channels=out_channels, kernel_size=1,
                                stride=1, padding=0, dilation=1,
                                ↪  groups=1, bias=True,
                                ↪  padding_mode='zeros')
                self.relu = torch.nn.LeakyReLU (
                ↪  negative_slope=0.01, inplace=False)
```

```
                self.maxp = torch.nn.MaxPool2d ( kernel_size=2,
                ↪  stride=None, padding=0, dilation=1,
                ↪  return_indices=False, ceil_mode=False)
                self.in_channels = in_channels
                self.out_channels = out_channels

        def forward ( self, x):
                x_identity = x

                x = self.conv (x)
                x = self.bnorm (x)
                x = self.relu (x)

                x = self.convi (x)
                x = self.bnorm (x)
                if self.in_channels != self.out_channels:
                        x_identity = self.maxp ( x_identity)
                        x_identity = self.conv1x1 ( x_identity)
                        x += x_identity
                else:
                        x += x_identity
                x = self.relu (x)

                return x

class network ( nn.Module):
        def __init__ ( self, num_classes, num_objects):
                super ( network, self).__init__()
                self.conv = torch.nn.Conv2d ( in_channels=3,
                ↪  out_channels=64, kernel_size=7, stride=2,
                ↪  padding=3)
                self.bloc64    = block (   in_channels=64,
                ↪  out_channels=64, kernel_size=3, stride=1)
                self.bloc128i  = block (   in_channels=64,
                ↪  out_channels=128, kernel_size=3, stride=2)
                self.bloc128   = block (  in_channels=128,
                ↪  out_channels=128, kernel_size=3, stride=1)
                self.bloc256i  = block (  in_channels=128,
                ↪  out_channels=256, kernel_size=3, stride=2)
                self.bloc256   = block (  in_channels=256,
                ↪  out_channels=256, kernel_size=3, stride=1)
                self.bloc512i  = block (  in_channels=256,
                ↪  out_channels=512, kernel_size=3, stride=2)
                self.bloc512   = block (  in_channels=512,
                ↪  out_channels=512, kernel_size=3, stride=1)
                self.bloc1024i = block (  in_channels=512,
                ↪  out_channels=1024, kernel_size=3, stride=2)
                self.bloc1024  = block ( in_channels=1024,
                ↪  out_channels=1024, kernel_size=3, stride=1)
                self.num_classes = num_classes
                self.num_objects = num_objects
```

```
                fc_output_size = 65536 # int ( 131072/4) # 262144
                fc_output_size = int ( 32768/4*4) # int (
                ↪  131072/4) # 262144
                self.fc = torch.nn.Linear ( fc_output_size, 1000)
                self.fc_exists = torch.nn.Linear ( 1000,
                ↪  self.num_objects)
                self.fc_bbox   = torch.nn.Linear ( 1000,
                ↪  4*self.num_objects)

                self.fc_cat_up   = torch.nn.Linear ( 1000,
                ↪  self.num_objects*self.num_classes)
                self.fc_cat_down = torch.nn.Linear (
                ↪  self.num_objects*self.num_classes,
                ↪  self.num_classes)
#                self.fc_cat    = torch.nn.Linear ( 1000,
↪  self.num_objects)

                self.relu = torch.nn.LeakyReLU (
                ↪  negative_slope=0.01, inplace=False)
                self.softmax = torch.nn.Softmax ()
                self.sig = nn.Sigmoid()
                self.maxp = torch.nn.MaxPool2d ( kernel_size=2,
                ↪  stride=None, padding=0, dilation=1,
                ↪  return_indices=False, ceil_mode=False)

                self.conv15 = torch.nn.Conv2d ( in_channels=512,
                ↪  out_channels=128, kernel_size=15, stride=1,
                ↪  padding=0)
                self.conv13 = torch.nn.Conv2d ( in_channels=512,
                ↪  out_channels=128, kernel_size=13, stride=1,
                ↪  padding=0)

                self.conv7 = torch.nn.Conv2d ( in_channels=512,
                ↪  out_channels=128, kernel_size=7, stride=1,
                ↪  padding=0)
                self.conv5 = torch.nn.Conv2d ( in_channels=512,
                ↪  out_channels=128, kernel_size=5, stride=1,
                ↪  padding=0)
                self.conv3 = torch.nn.Conv2d ( in_channels=512,
                ↪  out_channels=128, kernel_size=3, stride=1,
                ↪  padding=0)
                self.conv1 = torch.nn.Conv2d ( in_channels=512,
                ↪  out_channels=128, kernel_size=1, stride=1,
                ↪  padding=0)

                self.fc7 = torch.nn.Linear ( 512, 1000)
                self.fc5 = torch.nn.Linear ( 2048, 1000)
                self.fc3 = torch.nn.Linear ( 4608, 1000)
                self.fc1 = torch.nn.Linear ( 8192, 1000)

        def forward ( self, x):
```

```
                x = self.conv (x)
                ##
                x = self.maxp ( x)
                ##
                m = 0
                for i in range ( 5-m):
                        x = self.bloc64 (x)
                x = self.bloc128i (x)
                for i in range ( 4-m):
                        x = self.bloc128 (x)
                x = self.bloc256i (x)
                for i in range ( 4-m):
                        x = self.bloc256 (x)
                x = self.bloc512i (x)
                for i in range ( 4-m):
                        x = self.bloc512 (x)

##                print ( 'x =', x.size())
                x7 = self.conv7 (x)
                x5 = self.conv5 (x)
                x3 = self.conv3 (x)
                x1 = self.conv1 (x)

                fc_output_size = 1
                for index in range ( 1, len ( x7.size())):
                        fc_output_size *= x7.size()[index]
                x7 = x7.view ( -1, fc_output_size)
                x7 = self.fc7 (x7)
                fc_output_size = 1
                for index in range ( 1, len ( x5.size())):
                        fc_output_size *= x5.size()[index]
                x5 = x5.view ( -1, fc_output_size)
                x5 = self.fc5 (x5)
                fc_output_size = 1
                for index in range ( 1, len ( x3.size())):
                        fc_output_size *= x3.size()[index]
                x3 = x3.view ( -1, fc_output_size)
                x3 = self.fc3 (x3)
                fc_output_size = 1
                for index in range ( 1, len ( x1.size())):
                        fc_output_size *= x1.size()[index]
                x1 = x1.view ( -1, fc_output_size)
                x1 = self.fc1 (x1)

                '''
                print ( 'x7 =', x7.size())
                print ( 'x5 =', x5.size())
                print ( 'x3 =', x3.size())
                print ( 'x1 =', x1.size())
                print ( 'x1 =', x_res.size())
                '''
```

```
                x_res = x7 + x5 + x3 +x1

####                x = x.view ( -1, fc_output_size)

                ##
#                x = self.bloc1024i (x)
#                for i in range ( 4):
#                        x = self.bloc1024 (x)
                ##

#                print ( 'x1 = ', x.size())
#                x    = self.maxp (x)
#                print ( 'x2 = ', x.size())
                fc_output_size = 1
                for index in range ( 1, len ( x.size())):
                        fc_output_size *= x.size()[index]
#                print ( x.size())
#                print ( fc_output_size)
                '''
                x = x.view ( -1, fc_output_size)
                x_fwd  = self.fc (x)
                '''
###                x = x.view ( -1, fc_output_size)

                x_fwd  = x_res

                x_exists = self.fc_exists (x_fwd)
                x_exists = self.sig ( x_exists)

                x_bbox   = self.fc_bbox (x_fwd)
                x_bbox   = self.relu ( x_bbox)

####                x_cat_up = self.fc (x)
####                x_cat_up = self.fc_cat_up (x_cat_up)
                x_cat_up = self.fc_cat_up (x_res)
                x_cats = []
                for index in range ( self.num_objects):
                        x_cat    = x_cat_up[:,self.num_classes*i
                        ↪  ndex:self.num_classes*index+self.num
                        ↪  _classes]
                        x_cat    = self.softmax (x_cat)
                        x_cats.append ( x_cat)

                return x_exists, x_bbox, x_cats

scale = 0.4
class dataset_class ( Dataset):
        def __init__ ( self,  imagenet_root, coco, image_ids,
        ↪  class_index_list, num_objects, transform):
                self.transform = transform
```

```
                self.imagenet_root = imagenet_root
                self.filename = []
                self.exist    = []
                self.label    = []
                self.bbox     = []

                image_datas = coco.loadImgs  ( ids=image_ids)
                for image_data in image_datas:
                        image_id = image_data['id']
                        filename = image_data['file_name']
#                        print ( self.imagenet_root + '/' +
↪  filename)
                        image_annotations = coco.loadAnns (
                        ↪  coco.getAnnIds ( imgIds=image_id))

#                        if len ( image_annotations) !=
↪  num_objects:
#                                continue

                        labels = np.zeros ( num_objects) # ,
                        ↪  dtype=int)
                        bboxs = np.zeros ( 4*num_objects)
                        exists = np.zeros ( num_objects)
                        for index in range ( len (
                        ↪  image_annotations)):
                                image_ann =
                                ↪  image_annotations[index]

                                category_id =
                                ↪  image_ann['category_id']
                                class_index = np.where (
                                ↪  np.array ( class_index_list)
                                ↪  == category_id)
#                                print ( category_id, ' ',
↪  class_index_list, ' ', class_index)

                                bbox = scale*np.array (
                                ↪  image_ann['bbox'])

                                exists[index] = 1
                                bboxs[4*index:4*index+4] = bbox
                                labels[index] = class_index[0] #
                                ↪  category_id

                        self.filename.append ( filename)
                        self.exist.append    ( exists)
                        self.label.append    ( labels)
                        self.bbox.append     ( bboxs)

        def __len__ ( self) :
```

```
                return ( len ( self.filename))
        def __getitem__ ( self, idx) :
                # Load the image
                image = Image.open( self.imagenet_root + '/' +
                ↪  self.filename[idx])
                data = self.transform ( image)
                image.close ()

                exist = self.exist[idx]
                label = self.label[idx]
                bbox = self.bbox[idx]

                return data, exist, label, bbox,
                ↪  self.imagenet_root + '/' + self.filename[idx]

device = torch.device("cuda" if torch.cuda.is_available() else
↪  "cpu") # PyTorch v0.4.0
# device = torch.device("cpu")

coco_training   = COCO ( args.training_annotation)
coco_validation = COCO ( args.validation_annotation)
catagory    = { index:coco_training.loadCats ( index)[0]['name']
↪  for index in coco_training.getCatIds ()}
rcatagory   = { coco_training.loadCats ( index)[0]['name']:index
↪  for index in coco_training.getCatIds ()}
# print ( catagory)

## class_list = list ( rcatagory.keys()) # [ 'cat', 'dog']
## class_list = [ 'bird', 'cat', 'dog', 'horse', 'sheep', 'cow',
↪  'elephant', 'bear', 'zebra', 'giraffe']
class_list = [ 'cat', 'dog', 'cow', 'zebra', 'giraffe']
## class_list = [ 'cat', 'dog']
class_index_list = [ rcatagory[class_name] for class_name in
↪  class_list]

num_objects = 3
## num_objects = 1
image_ids = get_image_ids ( coco_training, class_index_list,
↪  num_objects, False) # True)

transform = tvt.Compose ( [ tvt.ToTensor (), tvt.Normalize ( (
↪  0.5, 0.5, 0.5), ( 0.5, 0.5, 0.5))])

train_dataset = dataset_class ( args.training_dataset,
↪  coco_training, image_ids, class_index_list, num_objects,
↪  transform)
train_data_loader = torch.utils.data.DataLoader (
↪  dataset=train_dataset,
        batch_size=10, shuffle=True, num_workers=4)
```

```
val_image_ids = get_image_ids ( coco_validation,
↪  class_index_list, num_objects, False)
val_dataset = dataset_class ( args.validation_dataset,
↪  coco_validation, val_image_ids, class_index_list,
↪  num_objects, transform)
val_data_loader = torch.utils.data.DataLoader (
↪  dataset=val_dataset,
        batch_size=10, shuffle=True, num_workers=4)

def calculate_jaccard_distance ( bbox1, bbox2):
        bbox1_x_min, bbox1_y_min, bbox1_width, bbox1_height =
        ↪  bbox1
        bbox2_x_min, bbox2_y_min, bbox2_width, bbox2_height =
        ↪  bbox2

        bbox1_x_max = bbox1_x_min + bbox1_width
        bbox1_y_max = bbox1_y_min + bbox1_height
        bbox2_x_max = bbox2_x_min + bbox2_width
        bbox2_y_max = bbox2_y_min + bbox2_height

        bbox1_area = bbox1_width*bbox1_height
        bbox2_area = bbox2_width*bbox2_height

        i_x1        = max ( bbox1_x_min, bbox2_x_min)
        i_x2        = min ( bbox1_x_max, bbox2_x_max)
        i_y1        = max ( bbox1_y_min, bbox2_y_min)
        i_y2        = min ( bbox1_y_max, bbox2_y_max)
        intersection_area        = max ( i_x2 - i_x1, 0) * max (
        ↪  i_y2 - i_y1, 0)
        union_area                = bbox1_area + bbox2_area -
        ↪  intersection_area

        jaccard_distance        = float (
        ↪  intersection_area)/float ( union_area)

        return jaccard_distance

def annotate_bound_box ( image, bbox, label, color,
↪  label_pos='top'):
        font                = cv2.FONT_HERSHEY_SIMPLEX
        fontScale        = 0.5
        thickness        = 1

        x_min, y_min, width, height = bbox
        x_max         = x_min + width
        y_max         = y_min + height

        x_min        = max ( x_min, 0)
        y_min        = max ( y_min, 0)
        x_max        = min ( x_max, 256)
        y_max        = min ( y_max, 256)
```

```
        start_point        = ( int ( x_min), int ( y_min))
        end_point        = ( int ( x_max), int ( y_max))
        image = cv2.rectangle ( image, start_point, end_point,
        ↪  color, thickness)

        if label_pos=='top':
                pos = ( int ( x_min), int ( y_min+10))
        elif label_pos=='bottom':
                pos = ( int ( x_min), int ( y_max))
        image = cv2.putText   ( image, label, pos,
                                font, fontScale, color,
                                ↪  thickness, cv2.LINE_AA)

        return image

def dump_train_output ( filename, exists, bboxs, labels,
↪  pred_exists, pred_bboxs, pred_labels, dumppath):
        n        = 5
        n_index        = 0
        plt.figure ( figsize=(20, 4))
        green        = [ 0, 255, 0]
        red        = [ 255, 0, 0]
        bboxs                = bboxs.cpu().detach().numpy()
        exists                = exists.cpu().detach().numpy()
        labels                = labels.cpu().detach().numpy()
        pred_exists        = pred_exists.cpu().detach().numpy()
        pred_bboxs        = pred_bboxs.cpu().detach().numpy()
        for bbox_index in range ( len ( exists[0])):
                pred_labels[bbox_index]        = pred_labels[bbo
                ↪  x_index].cpu().detach().numpy()
        threshold        = 0.5
        for index in range ( len ( filename)):
                if n_index < n:
                        image = cv2.imread ( filename[index])
                        image = cv2.cvtColor ( image,
                        ↪  cv2.COLOR_BGR2RGB)
                for bbox_index in range ( len ( exists[index])):
                        if n_index < n:
                                if exists[index][bbox_index] >
                                ↪  threshold:
                                        l = class_list[labels[bb
                                        ↪  ox_index][index]]
                                        image =
                                        ↪  annotate_bound_box (
                                        ↪  image, bboxs[index][
                                        ↪  4*bbox_index:4*(bbox
                                        ↪  _index+1)], l,
                                        ↪  green, 'top')
```

```
                                if pred_exists[index][bbox_index
                                ↪  ] >
                                ↪  threshold:
                                        l = class_list[np.argmax
                                        ↪  (pred_labels[bbox_in
                                        ↪  dex][index])]
                                        image =
                                        ↪  annotate_bound_box (
                                        ↪  image, pred_bboxs[in
                                        ↪  dex][4*bbox_index:4*
                                        ↪  (bbox_index+1)], l,
                                        ↪  red, 'bottom')
                        if n_index < n:
                                plt.subplot ( 1, n, n_index + 1)
                                plt.imshow ( image)
                                n_index += 1
                                if not n_index < n:
                                        plt.savefig ( dumppath,
                                        ↪  bbox_inches='tight')
                                        plt.close ()
def get_loss ( network, data_loader, dumppath):
        l1s = []
        l2s = []
        l3s = []
        cat_criterion        = torch.nn.CrossEntropyLoss()
        reg_criterion        = torch.nn.MSELoss()
        y_true = np.array ( [])
        y_pred = np.array ( [])
        first=True
        threshold=0.5
        iou = []
        with torch.no_grad():
                for x, y_exists, y_labels, y_bboxs, y_filename
                ↪  in data_loader:

                        y_exists               = y_exists.to (
                        ↪  device, dtype=dtype)
                        y_labels = y_labels.T
                        y_labels = y_labels.to ( device,
                        ↪  dtype=torch.long)
                        y_bboxs               = y_bboxs.to (
                        ↪  device, dtype=dtype)
                        x = x.to ( device=device)
                        predicted_exists, predicted_bboxs,
                        ↪  predicted_labels  = network (x)
                        total_l1 = 0.0
                        for index in range ( num_objects):
                                l1 = cat_criterion (
                                ↪  predicted_labels[index],
                                ↪  y_labels[index])
                                total_l1 += l1.item()
```

```
l1s.append ( total_l1)
for index in range ( len ( y_filename)):
        for bbox_index in range ( len (
        ↪  y_exists[index])):
                if y_exists[index][bbox_
                ↪  index] >
                ↪  threshold:
                        l = y_labels[bbo
                        ↪  x_index][ind
                        ↪  ex].cpu().nu
                        ↪  mpy()
                        y_true = np.conc
                        ↪  atenate ( [
                        ↪  y_true, [l]])
                        l = np.argmax(pr
                        ↪  edicted_labe
                        ↪  ls[bbox_inde
                        ↪  x][index].cp
                        ↪  u().numpy())
                        y_pred = np.conc
                        ↪  atenate ( [
                        ↪  y_pred, [l]])
for bbox_index in range ( len (
↪  y_exists[index])):
        if y_exists[index][bbox_index] >
        ↪  threshold:
                u = calculate_jaccard_di
                ↪  stance (
                ↪  y_bboxs[index][4*bbo
                ↪  x_index:4*(bbox_inde
                ↪  x+1)],
                ↪  predicted_bboxs[inde
                ↪  x][4*bbox_index:4*(b
                ↪  box_index+1)])
                iou.append ( u)

l2 = reg_criterion ( predicted_bboxs,
↪  y_bboxs)
l2s.append ( l2.item())

l3 = reg_criterion ( predicted_exists,
↪  y_exists)
l3s.append ( l3.item())
accuracy = (y_true ==
↪  y_pred).sum()/y_true.shape[0]
if first:
        dump_train_output ( y_filename,
        ↪  y_exists, y_bboxs, y_labels,
        ↪  predicted_exists,
        ↪  predicted_bboxs,
        ↪  predicted_labels, dumppath)
```

```
                        first=False
                return np.mean ( l1s), np.mean ( l2s), np.mean (
                ↪  l3s), accuracy, np.mean ( iou)

# print ( len ( train_dataset))

'''
num_classes = len ( class_list)
model = network ( num_classes=num_classes,
↪  num_objects=num_objects)

epochs = 50
def run_code_for_training ( net):
        net                = net.to ( device)
        cat_criterion        = torch.nn.CrossEntropyLoss()
        reg_criterion        = torch.nn.MSELoss()
#        optimizer        = torch.optim.SGD ( net.parameters(),
↪  lr=1e-6, momentum=0.9, weight_decay=0.5)
        optimizer        = torch.optim.SGD ( net.parameters(),
        ↪  lr=1e-6, momentum=0.9)
        l1s = []
        l2s = []
        l3s = []
        for epoch in range ( epochs):
                start = time.process_time()
                running_loss = 0.0
                for i, data in enumerate ( train_data_loader):
                        inputs, exists, labels, bboxs,
                        ↪  filename        = data
                        inputs                = inputs.to (
                        ↪  device, dtype=dtype)
                        exists                = exists.to (
                        ↪  device, dtype=dtype)
                        labels = labels.T
                        labels = labels.to ( device,
                        ↪  dtype=torch.long)
                        bboxs                = bboxs.to (
                        ↪  device, dtype=dtype)

                        optimizer.zero_grad()
                        outputs                = net ( inputs)
                        predicted_exists = outputs[0]
                        predicted_labels = outputs[2]
                        predicted_bboxs  = outputs[1]

                        if (
                        ↪  torch.isnan(predicted_exists).any()):
                                print ( "exists Nan Error")
                        for index in range ( num_objects):
                                if ( torch.isnan(predicted_label
                                ↪  s[index]).any()):
```

```
                                print ( "labels Nan
                                ↪  Error")
                        if ( torch.isnan(predicted_bboxs).any()):
                                print ( "bboxs Nan Error")

                        total_l1 = 0.0
                        for index in range ( num_objects):
                                l1 = cat_criterion (
                                ↪  predicted_labels[index],
                                ↪  labels[index])
                                l1.backward( retain_graph=True)
                                total_l1 += l1.item()
                        l1s.append ( total_l1)

                        l2 = reg_criterion ( predicted_bboxs,
                        ↪  bboxs)
                        l2.backward( retain_graph=True)
                        l2s.append ( l2.item())

                        l3 = reg_criterion ( predicted_exists,
                        ↪  exists)
                        l3.backward()
                        l3s.append ( l3.item())

                        optimizer.step()

                if (epoch+1) % 25 == 0:
                        torch.save ( net,
                        ↪  'model/network_epoch_%03d.pth' %
                        ↪  (epoch))

                l1t, l2t, l3t, acct, iout = get_loss ( net,
                ↪  train_data_loader,
                ↪  './epoch/Epoch_%03d__Train.png' % (epoch))
                l1v, l2v, l3v, accv, iouv = get_loss ( net,
                ↪  val_data_loader,
                ↪  './epoch/Epoch_%03d__Val.png' % (epoch))
                print ( 'Epoch %3d : Train [ %.6f %.6f %.6f
                ↪  (%.2f) %.6f] Val [ %.6f %.6f %.6f (%.2f)
                ↪  %.6f]' % ( epoch, l1t, l2t, l3t, 100*acct,
                ↪  iout, l1v, l2v, l3v, 100*accv, iouv))
                print ( 'Epoch %3d : Traing Loss' % ( epoch),
                ↪  end='')
                print ( ' [Cat : %.6f ] ' % ( np.mean ( l1s)),
                ↪  end='')
                print ( ' [Reg : %.6f, %.6f] ' % ( np.mean (
                ↪  l2s), np.mean ( l3s)), end='')
                print ( ' Time %.3f s' % ( time.process_time() -
                ↪  start), end='')
                print ( '')
# run_code_for_training ( model)
```

```
# torch.save ( model, 'network.pth')

## model = network( 2, 3).to(device)
## summary ( model, ( 3, 256, 256))
'''

network_model = torch.load ( args.network_model_path)
network_model                = network_model.to ( device)

l1t, l2t, l3t, acct, iout = get_loss ( network_model,
↪  train_data_loader, './epoch_val/Train.png')
l1v, l2v, l3v, accv, iouv = get_loss ( network_model,
↪  val_data_loader, './epoch_val/Val.png')

print ( l1t, l2t, l3t, acct, iout)
```

Author: Pranab Dash

Homework 5: Image localization and classification

To solve this problem two approaches were used. I will briefly explain the two approaches I tried for multi-object detection, localization, and classification. The aim of this endeavor was to detect variable number of objects and label the object with maximum accuracy.

Note:
The entire COCO dataset was got by using the COCO website link (https://cocodataset.org/#download) and untarred.
1. Training Dataset: http://images.cocodataset.org/zips/train2017.zip
2. Validation Dataset: http://images.cocodataset.org/zips/val2017.zip
3. Annotations: http://images.cocodataset.org/annotations/annotations_trainval2017.zip

The images were preprocessed to add padding to make them of equal height and width and rescaled.

I.

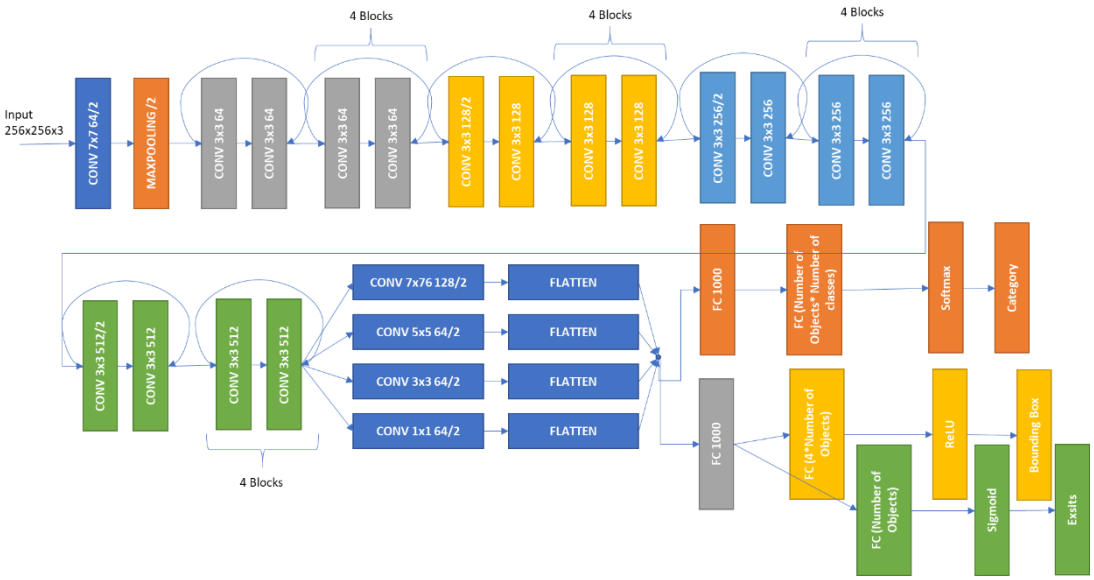
Approach 1

Using a Skip block Network Type model to detect 1 to 3 objects from 5 animal classes.

Design consideration and explanation:

1. The architecture should be capable of detecting the objects and bounding boxes also used for classification. The assumption was that the Skip blocks in the architecture could be used in extracting image features from the inputs. The 4 convolution layers with different kernel sizes gives the network capabilities to search different sizes of bounding boxes across the images.
2. There are two types of output which are outputted by the network.
    a. Exists: This indicated if the detected bounding boxes contain an object or not. This is used to get variable number of object detection.
    b. Bounding Box: This gives a 4-element tuple, i.e., x_min, y_min, width, and height, for each object detected.
    c. Category: This provides a one-hot vector for each object detected. This used to get the labels.

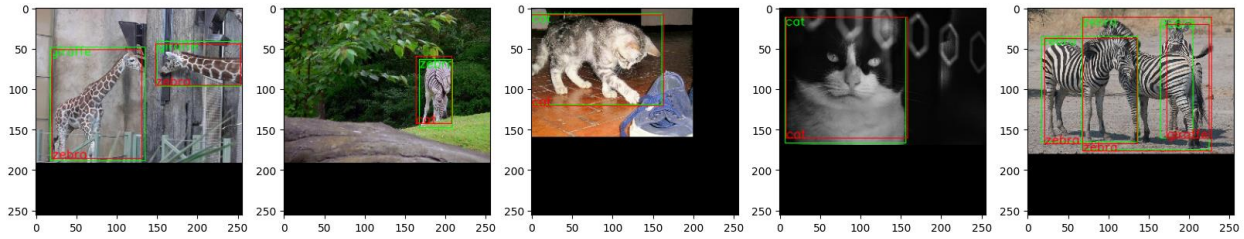
Final Architecture:



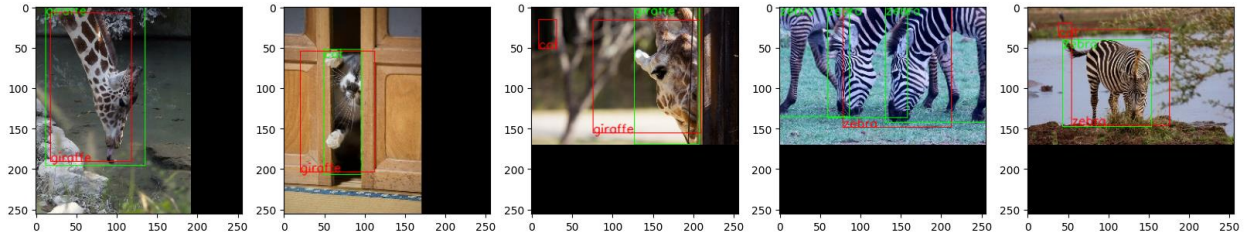
Evaluation:

1. Epoch 30:

   Training Set:

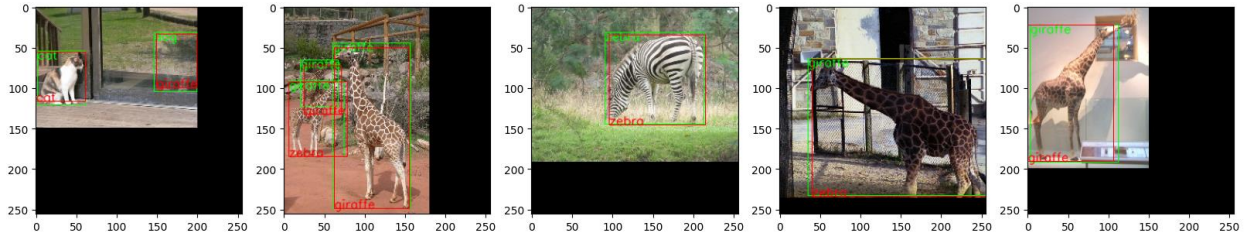


   Validation Set:

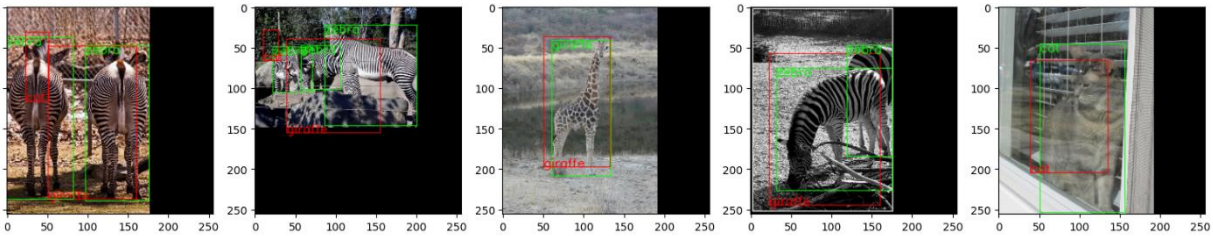


2. Epoch 40:

   Training Set:

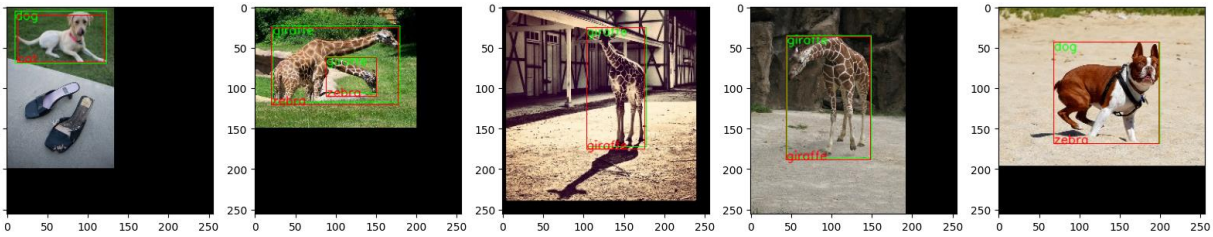


   Validation Set:

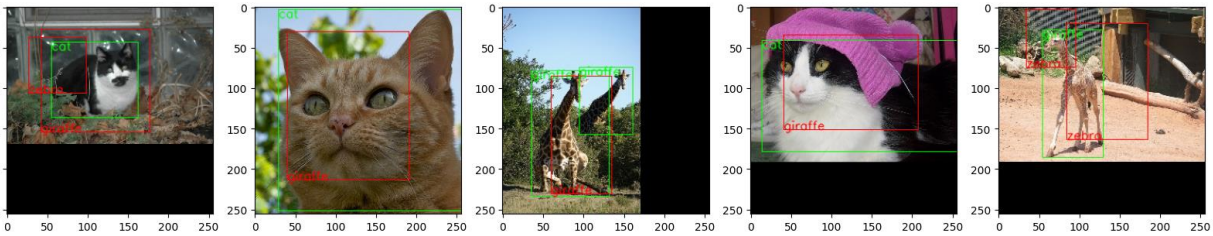


3. Epoch 50:

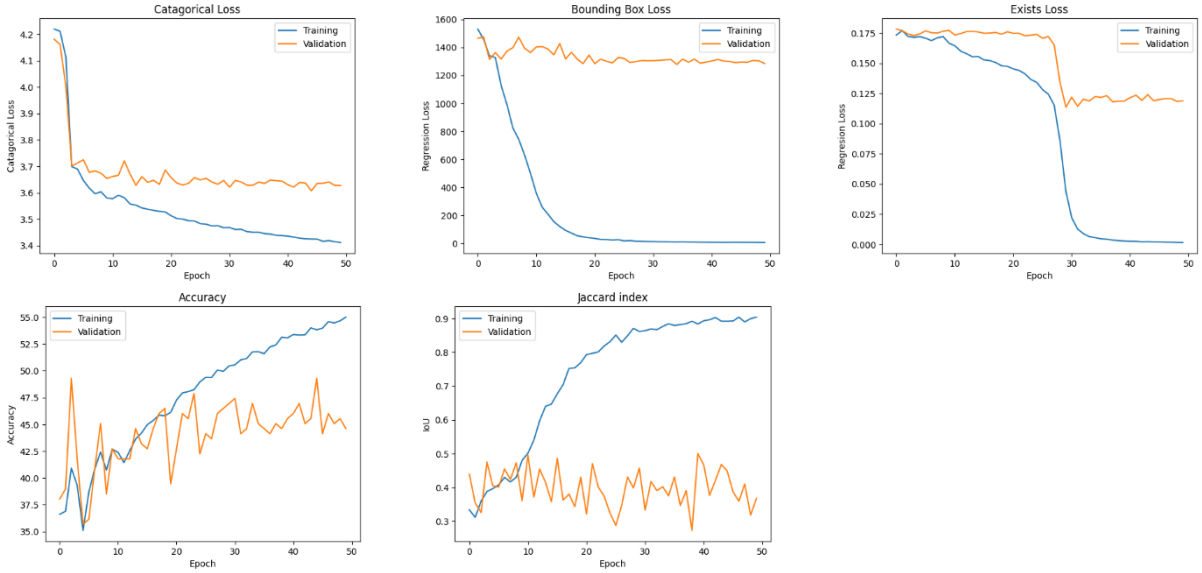
   Training Set



   Validation Set:



Losses and Accuracy:



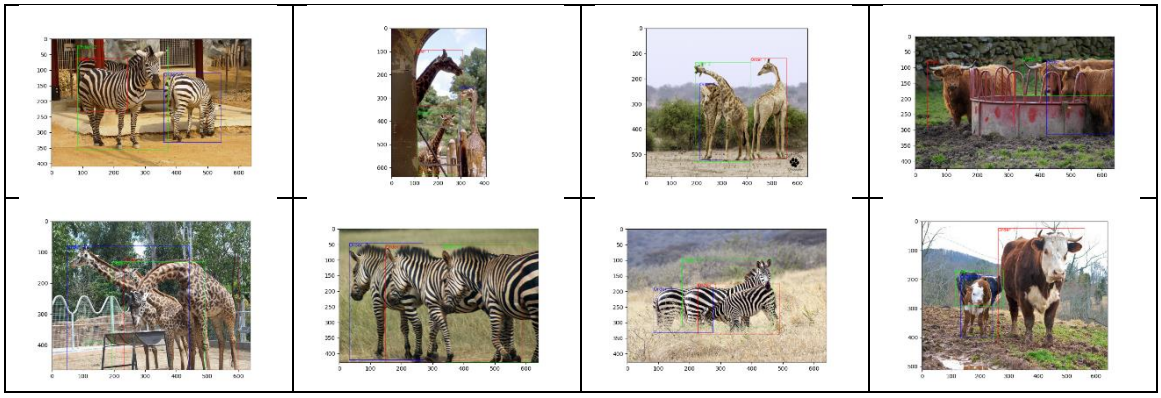
Things tried to improve generalization:

1. Training the classification using a single object, then trained for the bounding boxes using one object. After this trained with 1 to 3 objects for classification, bounding boxes and objects exists. This was done to boosts the classification accuracy.

2. Added dropout layer in the skip blocks for preventing overfitting. This caused deterioration in both the training as well as validation loss and accuracy.
3. Tried different number of layers and fitter sizes.

Other things could have been tried to improve generalization:

1. Move the categorical layer to tap out from different layer in the network and add extra skip block to the branch. Possibly the training of the classification is getting interfered by the training of the bounding box and exits.
2. The annotation in the COCO dataset is in no particular order causes the model training tricky. Use of Jaccrad index to find out the correlation of the predicting bounding boxes and the ground truth.
   Some examples are as following showing lack of order. Red is the first annotation, green next and blue the last.



3. A two-layer approach. Training by first scaling the region ground truth patches with the object. Using this classifier network, then add a detector network to find the bounding boxes.

Conclusion:

The model did not generalize properly and there is a high classification error associated.
This may be due to the following reasons:

1. After going through the Single Shot Detection [3] paper and Week 9 slides it became clear that a region proposal is required for accurate detection. This gives way to the Approach 2.

II.

<u>Approach 2</u>

Using a ResNet Type network as backbone with Single Shot detector [3] model

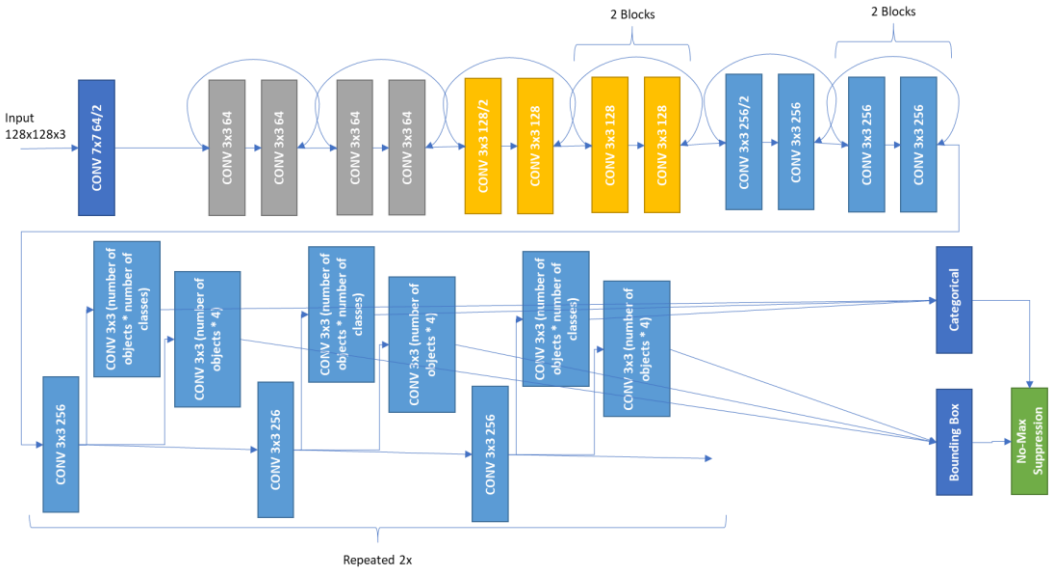
Design consideration and explanation:

1. The architecture should be capable of detecting the object in various region proposals in the images like the Single Shot detector.

Training Challenges:

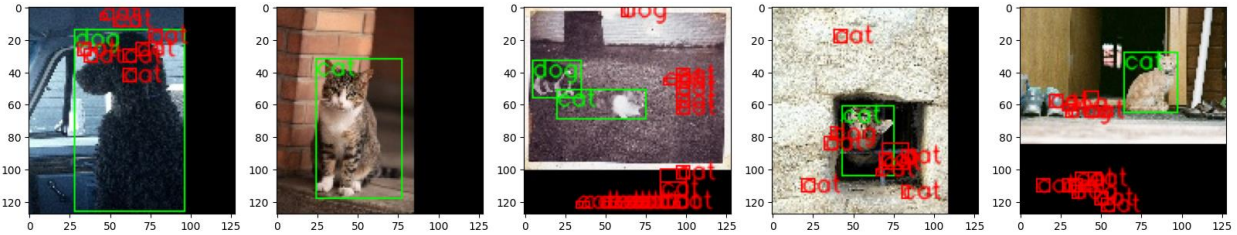
1. Each training epoch takes > 750 seconds to complete and >1000 seconds for non-max suppression layer. Due to resource and time constraints the model was ran only up to 20 epochs hence could not provide the expected accuracy.

Final Architecture:

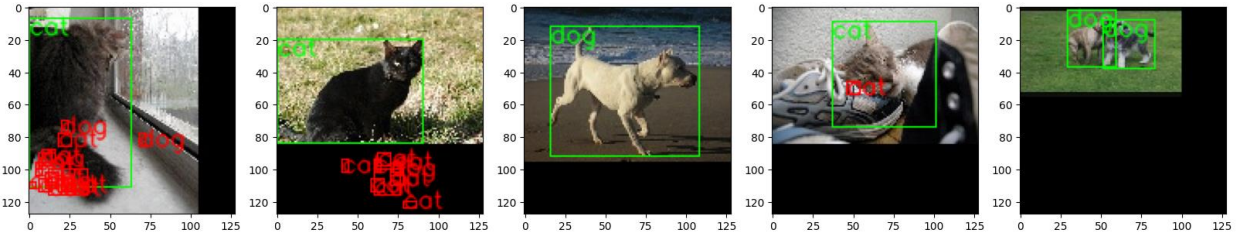


Evaluation:

1. Epoch 20:
   Training Set:



   Validation Set:



Conclusion:

Approach 2 requires much longer training and possible modification in the loss backpropagation and non-max suppression layer to provide the anticipated accuracy.

Code and Theory References:

1. https://github.com/FrancescoSaverioZuppichini/Pytorch-how-and-when-to-use-Module-Sequential-ModuleList-and-ModuleDict

2. https://github.com/MrParosk/ml_playground/blob/master/computer_vision/object_detection/ssd.ipynb
3. https://arxiv.org/pdf/1512.02325.pdf
4. https://github.com/sgrvinod/a-PyTorch-Tutorial-to-Object-Detection
5. https://teaching.pages.centralesupelec.fr/deeplearning-lectures-build/01-pytorch-object-detection.html
6. https://towardsdatascience.com/learning-note-single-shot-multibox-detector-with-pytorch-part-1-38185e84bd79
7. https://github.com/amdegroot/ssd.pytorch
8. https://stackoverflow.com/questions/61545482/ssdsingle-shot-detectors-default-box-implementation
9. https://nms.readthedocs.io/en/latest/_modules/nms/felzenszwalb.html
10. https://www.pyimagesearch.com/2014/11/17/non-maximum-suppression-object-detection-python/
11. https://github.com/jeremyfix/deeplearning-lectures/tree/master/LabsSolutions/01-pytorch-object-detection
12. https://pytorch.org/assets/deep-learning/Deep-Learning-with-PyTorch.pdf
13. https://stackoverflow.com/questions/57323023/pytorch-loss-backward-and-optimizer-step-in-eval-mode-with-batch-norm-laye
14. https://discuss.pytorch.org/t/if-my-model-has-dropout-do-i-have-to-alternate-between-model-eval-and-model-train-during-training/83007/8
15. http://seba1511.net/tutorials/beginner/blitz/neural_networks_tutorial.html