ECE695DL: Homework 6

Pranab Dash

14 April 2021

hw06_training.py

```
# Downloader, dataloader, network, training code, validation
↪  code, code to verify and visualize the validation, anchor
↪  boxes, custom loss function, IoU...

from pycocotools.coco import COCO
import torch
import torchvision.transforms as tvt
import sys
import os
import cv2
import numpy as np
from PIL import Image
from torchsummary import summary
import time

import argparse
import os
import json
from pycocotools.coco import COCO
import requests
from PIL import Image
from requests.exceptions import ConnectionError, ReadTimeout,
↪  TooManyRedirects, MissingSchema, InvalidURL
import argparse
import pickle
from pycocotools.coco import COCO
from itertools import combinations
import numpy as np
import cv2
import matplotlib.pyplot as plt
```

```
# annIds=coco.getAnnIds(imgIds=im['id'],catIds=catIds,iscrowd=Fa⌋
↪  lse)

cat_list = [ 'bus', 'truck', 'car']
cat_list = [ 'cat', 'dog', 'giraffe']
## cat_list = [ 'cat', 'dog', 'person']

train_anno_path = sys.argv[1]
val_anno_path = sys.argv[2]

train_coco = COCO ( train_anno_path)
val_coco = COCO ( val_anno_path)


## print ( 'Train')
## for index in train_coco.getCatIds ():
##         cat_id = index
##         cat_image_ids = train_coco.getImgIds (
↪  catIds=[cat_id])
##         cat = train_coco.loadCats ( index)[0]['name']
##         print ( cat , ' : ', len ( cat_image_ids))
##
## print ( 'Val')
## for index in val_coco.getCatIds ():
##         cat_id = index
##         cat_image_ids = val_coco.getImgIds ( catIds=[cat_id])
##         cat = val_coco.loadCats ( index)[0]['name']
##         print ( cat , ' : ', len ( cat_image_ids))

image_size        = 128
grid_size        = 8
yolo_interval        = image_size // grid_size
num_archor_box        = 5
archor_aspect_ratio        = [ 1/5, 1/3, 1/1, 3/1, 5/1]

device = torch.device ( "cuda" if torch.cuda.is_available() else
↪  "cpu")

def get_image_d ( coco_image, root_path):
        image_url = coco_image['coco_url']
        image_name = coco_image['file_name']

        if len ( image_url) <= 1:
                #url is useless Do something
                raise RuntimeError
        try:
                image_response = requests.get( image_url,
                 ↪  timeout=1)
        except ConnectionError:
                # Handle this exception
                raise RuntimeError
        except ReadTimeout:
                # Handle this exception
                raise RuntimeError
        except TooManyRedirects:
                # handle exception
                raise RuntimeError
        except MissingSchema:
                # handle exception
                raise RuntimeError
        except InvalidURL:
                # handle exception
                raise RuntimeError

        if not 'content-type' in image_response.headers :
                # Missing content. Do something
                raise RuntimeError
        if not 'image' in image_response.headers['content-type']:
                # The url doesn't have any image. Do something.
                raise RuntimeError
        if ( len ( image_response.content) < 1000):
                # ignore images < 1kb
                raise RuntimeError

#        image_file_path = os.path.join ( root_path, image_name)
#        if os.path.exists ( image_file_path):
#                raise RuntimeError

        image_file_path = root_path
        with open ( image_file_path, 'wb') as img_f:
                img_f.write ( image_response.content)

def get_image ( image_data, orig_image_path):
        image_id = image_data['id']
        filename = image_data['file_name']
        get_image_d ( image_data, orig_image_path) # root_path)
#        os.system ( '')
        return None

def process_image ( image_data, scale, orig_image_path,
 ↪  process_image_path):
        max_height = 640
        max_width = 640

        image = cv2.imread ( orig_image_path)

        black = ( 0, 0, 0)
        height = image_data['height']
        width  = image_data['width']
        blank_image = np.zeros ( ( max_height, max_width, 3),
        ↪  np.uint8)
        blank_image[:height,:width,:] = image
        blank_image = cv2.resize ( blank_image, (0,0), fx=scale,
        ↪  fy=scale)

        cv2.imwrite ( process_image_path, blank_image)

        return None

class dataset ( torch.utils.data.Dataset):
        def __init__ ( self, coco, cat_list, max_instance,
        ↪  isTrain=True):
                self.filename = []
                self.exist    = []
                self.label    = []
                self.bbox     = []
                self.yolo_vectors     = []
                self.transform = tvt.Compose ( [ tvt.ToTensor
                ↪  (), tvt.Normalize ( ( 0.5, 0.5, 0.5), ( 0.5,
                ↪  0.5, 0.5))])

                scale = 0.2 # 640 -> 128

                image_ids = []
                catagory    = { index:coco.loadCats (
                ↪  index)[0]['name'] for index in
                ↪  coco.getCatIds ()}
                rcatagory   = { coco.loadCats (
                ↪  index)[0]['name']:index for index in
                ↪  coco.getCatIds ()}

                for cat in cat_list:
                        cat_id = rcatagory[cat]
                        cat_image_ids = coco.getImgIds (
                        ↪  catIds=[cat_id])
                        image_ids = image_ids + cat_image_ids
                image_ids = list ( set ( image_ids))
                print ( len ( image_ids))
                image_datas = coco.loadImgs  ( ids=image_ids)
                for image_data in image_datas:
                        train = 'train'
                        if not isTrain:
                                train = 'val'

                        image_id = image_data['id']
                        filename = image_data['file_name']
                        orig_image_path = 'dataset/%s/%s' % (
                        ↪  train, filename)
##                        print ( 'cp $SRC/%s $DST/%s/%s' % (
↪  image_data['file_name'], train, image_data['file_name'])) #
↪  Locate repo
                        if not os.path.isfile ( orig_image_path):
                                get_image ( image_data,
                                ↪  orig_image_path)
                        process_image_path =
                        ↪  'dataset/%s_process/%s' % ( train,
                        ↪  filename)
                        if not os.path.isfile (
                        ↪  process_image_path):
                                process_image ( image_data,
                                ↪  scale, orig_image_path,
                                ↪  process_image_path)

##                        ann_ids = []
##                        for cat in cat_list:
##                                for ann_id  in coco.getAnnIds
↪  ( imgIds=image_id, catIds=rcatagory[cat], iscrowd=False):
##                                        ann_ids.append (
↪  ann_id)
##                        ann_ids1 = [ ann_id for cat in
↪  cat_list for ann_id  in coco.getAnnIds ( imgIds=image_id,
↪  catIds=rcatagory[cat], iscrowd=False)]
##                        print ( len ( ann_ids), ' ', len (
↪  ann_ids1), ' ',  len ( ann_ids1) == len ( ann_ids))
                        ann_ids = [ ann_id for cat in cat_list
                        ↪  for ann_id  in coco.getAnnIds (
                        ↪  imgIds=image_id,
                        ↪  catIds=rcatagory[cat],
                        ↪  iscrowd=False)]
                        if not ( 0 < len ( ann_ids) <=
                        ↪  max_instance):
                                continue
                        image_anns = coco.loadAnns ( ann_ids)
##                        print ( len ( ann_ids), ' ', len (
↪   ann_ids), ' ',  len ( ann_ids) == len ( ann_ids))

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

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

                                exists[index] = 1
                                bboxs[4*index:4*index+4] = bbox
                                labels[index] = category_id

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

                        # 1 Anchor Box => [ e, c_x, c_y, b_h,
                        ↪   b_w, c1, c2, c3]
                        archor_length = 1+4+3
                        yolo_vector = np.zeros ( grid_size *
                        ↪   grid_size * num_archor_box *
                        ↪   archor_length)
                        for index in range ( len ( image_anns)):
                                archor_vector = np.zeros ( 1+4+3)
                                image_ann = image_anns[index]

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

                                tl_x, tl_y, w, h = bbox
                                c_x = tl_x + w/2
                                c_y = tl_y + h/2

                                aspect_ratio = w/h
```

```
                              id_x = c_x // yolo_interval
                              id_y = c_y // yolo_interval

                              del_x = id_x + 0.5 - c_x /
                               ↪  yolo_interval
                              del_y = id_y + 0.5 - c_y /
                               ↪  yolo_interval

                              yolo_w = float ( w) / float (
                               ↪  yolo_interval)
                              yolo_h = float ( h) / float (
                               ↪  yolo_interval)

                              archor_vector[0] = 1
                              archor_vector[1] = del_x
                              archor_vector[2] = del_y
                              archor_vector[3] = yolo_w
                              archor_vector[4] = yolo_h
                              sub_cat_id = cat_list.index (
                               ↪  catagory[category_id])
                              archor_vector[5+sub_cat_id] = 1

                              archor_index = -1
                              delta = 0.01
##                              for ar in archor_aspect_ratio:
##                                      if ar < aspect_ratio +
 ↪  delta:
##                                              archor_index
 ↪  += 1
##                              print ( len (
 ↪  archor_aspect_ratio)-1)
                              for index in range ( len (
                               ↪  archor_aspect_ratio)):
                                      if index == 0:
                                              hi_limit  = ( ar⌋
                                               ↪  chor_aspect_⌋
                                               ↪  ratio[index]
                                               ↪  + archor_asp⌋
                                               ↪  ect_ratio[in⌋
                                               ↪  dex+1])/2
##                                              print ( index,
 ↪  ' : ', hi_limit)

                                              if aspect_ratio
                                               ↪  < hi_limit:
```

```
                                                                archor_i⌋
                                                                ↪  ndex
                                                                ↪  =
                                                                ↪  index
                                                elif 0 < index < len (
                                                ↪  archor_aspect_ratio)⌋
                                                ↪  -1:
                                                        low_limit = (
                                                        ↪  archor_aspec⌋
                                                        ↪  t_ratio[inde⌋
                                                        ↪  x-1] +
                                                        ↪  archor_aspec⌋
                                                        ↪  t_ratio[inde⌋
                                                        ↪  x])/2
                                                        hi_limit  = ( ar⌋
                                                        ↪  chor_aspect_⌋
                                                        ↪  ratio[index]
                                                        ↪  + archor_asp⌋
                                                        ↪  ect_ratio[in⌋
                                                        ↪  dex+1])/2
##                                                        print ( index,
↪  ' : ', low_limit, ' ', hi_limit)
                                                        if low_limit <=
                                                        ↪  aspect_ratio
                                                        ↪  <= hi_limit:
                                                                archor_i⌋
                                                                ↪  ndex
                                                                ↪  =
                                                                ↪  index
                                                else:
                                                        low_limit = (
                                                        ↪  archor_aspec⌋
                                                        ↪  t_ratio[inde⌋
                                                        ↪  x-1] +
                                                        ↪  archor_aspec⌋
                                                        ↪  t_ratio[inde⌋
                                                        ↪  x])/2
##                                                        print ( index,
↪  ' : ', low_limit)
                                                        if low_limit <
                                                        ↪  aspect_ratio:
                                                                archor_i⌋
                                                                ↪  ndex
                                                                ↪  =
                                                                ↪  index
```

```
                                yolo_archor_index = int ( (
                                ↪  archor_index + (
                                ↪  id_y*grid_size + id_x)*num_a⌋
                                ↪  rchor_box)*archor_length)

##                                print ( '%f %f %f %f' % (
↪  tl_x, tl_y, w, h))
##                                print ( '%f %f %f %f' % (
↪  id_x, id_y, yolo_w, yolo_h))
##                                print ( '%f %f %f' % ( c_x,
↪  c_y, aspect_ratio))
##                                print ( '%f %f' % (
↪  (id_x+0.5)*yolo_interval, (id_y+0.5)*yolo_interval))
##                                print ( '%f %f' % ( del_x,
↪  del_y))
##                                print ( archor_vector)
##                                print ( archor_aspect_ratio)
##                                print ( archor_index)
##                                print ( yolo_archor_index, '
↪  ', archor_vector)
##                                print ( '')


                                yolo_vector[yolo_archor_index:yo⌋
                                ↪  lo_archor_index+archor_lengt⌋
                                ↪  h] =
                                ↪  archor_vector


##                                archor_index = 4
##                                id_x = id_y = 7
##                                yolo_archor_index = int ( (
↪  archor_index + ( id_y*grid_size +
↪  id_x)*num_archor_box)*archor_length)
##                                print ( yolo_archor_index)

                                '''
                                print ( '%f %f %f %f' % ( tl_x,
                                ↪  tl_y, w, h))
                                print ( '%f %f %f %f' % ( id_x,
                                ↪  id_y, yolo_w, yolo_h))
                                print ( '%f %f %f' % ( c_x, c_y,
                                ↪  aspect_ratio))
                                print ( '%f %f' % (
                                ↪  (id_x+0.5)*yolo_interval,
                                ↪  (id_y+0.5)*yolo_interval))
                                print ( '%f %f' % ( del_x,
                                ↪  del_y))
                                print ( archor_vector)
                                print ( '')
                                '''
                        self.yolo_vectors.append ( yolo_vector)
##                        exit ()


                print ( 'Length = ', len ( self.filename))

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

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

                return data, exist, label, bbox, yolo_vectors


train_dataset = dataset ( train_coco, cat_list, 5, True)
val_dataset = dataset ( val_coco, cat_list, 5, False)

batch_size=32
train_data_loader = torch.utils.data.DataLoader (
↪  dataset=train_dataset,
        batch_size=batch_size, shuffle=True, num_workers=4)
val_data_loader = torch.utils.data.DataLoader (
↪  dataset=val_dataset,
        batch_size=batch_size, shuffle=True, num_workers=4)

class block ( torch.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 ( torch.nn.Module):
        def __init__ ( self):
                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.maxp = torch.nn.MaxPool2d ( kernel_size=2,
                ↪  stride=None, padding=0, dilation=1,
                ↪  return_indices=False, ceil_mode=False)
#                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.fc = torch.nn.Linear ( 256*8*8, 40*8*8)
                self.relu = torch.nn.LeakyReLU (
                ↪  negative_slope=0.01, inplace=False)

                self.drop = torch.nn.Dropout ( p=0.5)
        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.drop ( x)
                x = self.bloc128i (x)
                for i in range ( 4-m):
                        x = self.bloc128 (x)
                        x = self.drop ( x)
                x = self.bloc256i (x)
                for i in range ( 4-m):
                        x = self.bloc256 (x)
                        x = self.drop ( x)
#                x = self.bloc512i (x)
#                for i in range ( 4-m):
#                        x = self.bloc512 (x)
##                print ( x.size())

                x = x.view ( x.size (0), -1)
                x = self.fc (x)
                x = self.relu (x)
##                x = x.view ( x.size (0), 40, 8, 8)
###                print ( x.size())

#                exit ()
                return x

### class AnchorBox:
###         #               aspect_ratio top_left_corner
↪  anchor_box height & width   anchor_box index
###         def __init__(self,   AR,     tlc,
↪  ab_height,   ab_width,      adx):
###                 self.AR = AR
###                 self.tlc = tlc
###                 self.ab_height = ab_height
###                 self.ab_width = ab_width
###                 self.adx = adx
###         def __str__(self):
###                 return "AnchorBox type (h/w): %s    tlc for
↪  yolo cell: %s    anchor-box height: %d     \
###                         anchor-box width: %d   adx: %d" %
↪  (self.AR, str(self.tlc), self.ab_height, self.ab_width,
↪  self.adx)
```

```
### yolo_interval = 128 // 8
### num_cells_image_width = 128 // yolo_interval
### num_cells_image_height = 128 // yolo_interval
###
### anchor_boxes_1_1 = [[AnchorBox("1/1",
↪  (i*yolo_interval,j*yolo_interval), yolo_interval,
↪  yolo_interval,  0)
###                                                 for i in
↪  range(0,num_cells_image_height)]
###                                                         for
↪  j in range(0,num_cells_image_width)]
### anchor_boxes_1_3 =
↪  [[AnchorBox("1/3",(i*yolo_interval,j*yolo_interval),
↪  yolo_interval, 3*yolo_interval, 1)
###                                                 for i in
↪  range(0,num_cells_image_height)]
###                                                         for
↪  j in range(0,num_cells_image_width)]
### anchor_boxes_3_1 =
↪  [[AnchorBox("3/1",(i*yolo_interval,j*yolo_interval),
↪  3*yolo_interval, yolo_interval, 2)
###                                                 for i in
↪  range(0,num_cells_image_height)]
###                                                         for
↪  j in range(0,num_cells_image_width)]
### anchor_boxes_1_5 =
↪  [[AnchorBox("1/5",(i*yolo_interval,j*yolo_interval),
↪  yolo_interval, 5*yolo_interval, 3)
###                                                 for i in
↪  range(0,num_cells_image_height)]
###                                                         for
↪  j in range(0,num_cells_image_width)]
### anchor_boxes_5_1 =
↪  [[AnchorBox("5/1",(i*yolo_interval,j*yolo_interval),
↪  5*yolo_interval, yolo_interval, 4)
###                                                 for i in
↪  range(0,num_cells_image_height)]
###                                                         for
↪  j in range(0,num_cells_image_width)]

epochs = 100
net = network ().to ( device)
if False:
        summary ( net, ( 3, 128, 128))
        exit ()
```

```
threshold = 0.5
class yolo_loss ( torch.nn.Module):
        def __init__ ( self):
                super().__init__()

                self.mse = torch.nn.MSELoss()

                self.lambda_class = 1
                self.lambda_noobj = 10
                self.lambda_obj = 1
                self.lambda_box = 10

        def forward ( self, pred, ground):
                loss_init = False

                obj = ground[:,0::8] == 1
                nobj = ground[:,0::8] == 0

                '''
                loss_e = torch.square( pred[:,0::8][obj] -
                ↪  ground[:,0::8][obj])

                loss_x = torch.square( pred[:,1::8][obj] -
                ↪  ground[:,1::8][obj])
                loss_y = torch.square( pred[:,2::8][obj] -
                ↪  ground[:,2::8][obj])

                loss_w = torch.square( torch.sqrt (
                ↪  pred[:,3::8][obj]) - torch.sqrt (
                ↪  ground[:,3::8][obj]))
                loss_h = torch.square( torch.sqrt (
                ↪  pred[:,4::8][obj]) - torch.sqrt (
                ↪  ground[:,4::8][obj]))
                '''


                loss_e = self.mse( pred[:,0::8][obj],
                ↪  ground[:,0::8][obj])
                loss_ne = self.mse( pred[:,0::8][nobj],
                ↪  ground[:,0::8][nobj])
##                print ( loss_e)

                loss_x = self.mse( pred[:,1::8][obj],
                ↪  ground[:,1::8][obj])
                loss_y = self.mse( pred[:,2::8][obj],
                ↪  ground[:,2::8][obj])
```

```
##              print ( loss_x)
##              print ( loss_y)

##              loss_w = self.mse( torch.sqrt (
↪  pred[:,3::8])[obj], torch.sqrt ( ground[:,3::8])[obj])
##              loss_h = self.mse( torch.sqrt (
↪  pred[:,4::8])[obj], torch.sqrt ( ground[:,4::8])[obj])
            loss_w = self.mse( pred[:,3::8][obj],
            ↪  ground[:,3::8][obj])
            loss_h = self.mse( pred[:,4::8][obj],
            ↪  ground[:,4::8][obj])
##              print ( loss_w)
##              print ( loss_h)

##              loss_obj = 0
##              loss_nobj = 0
##              for c_idx in [ 5, 6, 7]:
##                      '''
##                      loss_obj += torch.square(
↪  pred[:,c_idx::8][obj] - ground[:,c_idx::8][obj])
##                      loss_nobj += torch.square(
↪  pred[:,c_idx::8][nobj] - ground[:,c_idx::8][nobj])
##                      '''

            c_idx = 5
            loss_obj = self.mse( pred[:,c_idx::8][obj],
            ↪  ground[:,c_idx::8][obj])
            loss_nobj = self.mse( pred[:,c_idx::8][nobj],
            ↪  ground[:,c_idx::8][nobj])

            c_idx = 6
            loss_obj += self.mse( pred[:,c_idx::8][obj],
            ↪  ground[:,c_idx::8][obj])
            loss_nobj += self.mse( pred[:,c_idx::8][nobj],
            ↪  ground[:,c_idx::8][nobj])

            c_idx = 7
            loss_obj += self.mse( pred[:,c_idx::8][obj],
            ↪  ground[:,c_idx::8][obj])
            loss_nobj += self.mse( pred[:,c_idx::8][nobj],
            ↪  ground[:,c_idx::8][nobj])

##              print ( loss_obj)
##              print ( loss_nobj)
##               loss = self.lambda_box ( loss_x + loss_y +
↪  loss_w + loss_h) + loss_obj + self.lambda_noobj*loss_nobj +
↪  loss_e
#                loss = self.lambda_box ( loss_x + loss_y +
↪  loss_w + loss_h) # + loss_obj + self.lambda_noobj*loss_nobj
↪  + loss_e
###              loss = 10 * ( loss_x + loss_y + loss_w +
↪  loss_h) + loss_obj + 10*loss_nobj + loss_e
                loss = 10 * ( loss_x + loss_y + loss_w + loss_h)
                 ↪  + loss_obj + 10*loss_ne + loss_obj
                return loss

#                ground = ground.detach().cpu().numpy()
#                pred   = pred.detach().cpu().numpy()
#                for box_idx in range (
↪  grid_size*grid_size*num_archor_box):
#                        if ground[box_idx*8] > threshold:

def calculate_jaccard_distance ( bbox1, bbox2):
        bbox1_cx_min, bbox1_cy_min, bbox1_width, bbox1_height =
        ↪  bbox1
        bbox2_cx_min, bbox2_cy_min, bbox2_width, bbox2_height =
        ↪  bbox2

        bbox1_x_min = bbox1_cx_min - bbox1_width/2
        bbox1_y_min = bbox1_cy_min - bbox1_height/2
        bbox1_x_max = bbox1_cx_min + bbox1_width/2
        bbox1_y_max = bbox1_cy_min + bbox1_height/2

        bbox2_x_min = bbox2_cx_min - bbox2_width/2
        bbox2_y_min = bbox2_cy_min - bbox2_height/2
        bbox2_x_max = bbox2_cx_min + bbox2_width/2
        bbox2_y_max = bbox2_cy_min + bbox2_height/2

        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)/(
        ↪  1e-6 + float ( union_area))

        return jaccard_distance

def max_suspression ( pred_yolo_vectors):
        pred_e = pred_yolo_vectors[0::8].detach().cpu().numpy()
        pred_e_idx = list( reversed ( np.argsort ( pred_e)))

        found_indexes = []
        for i in range ( 5):
                found_index = int ( pred_e_idx[0])

                bbox1 = pred_yolo_vectors[found_index*8+1:found_⌋
                ↪  index*8+5]
                del_x, del_y, yolo_w, yolo_h = bbox1
                w = yolo_w * float ( yolo_interval)
                h = yolo_h * float ( yolo_interval)
                y = found_index // 8
                x = found_index - found_index%5 - 8*y
                c_x = yolo_interval * ( x + 0.05 - del_x)
                c_y = yolo_interval * ( y + 0.05 - del_y)
                bbox1 = c_x, c_y, w, h
                print ( 'bb1 ', bbox1)

                list_pred_index = pred_e_idx[1:]
                sub_pred_e_idx = []
                for index in list_pred_index:
                        index = int ( index)
                        bbox2 = pred_yolo_vectors[index*8+1:inde⌋
                        ↪  x*8+5]

                        del_x, del_y, yolo_w, yolo_h = bbox2
                        w = yolo_w * float ( yolo_interval)
                        h = yolo_h * float ( yolo_interval)
                        y = index // 8
                        x = index - index%5 - 8*y
                        c_x = yolo_interval * ( x + 0.05 - del_x)
                        c_y = yolo_interval * ( y + 0.05 - del_y)
                        bbox2 = c_x, c_y, w, h

                        jc = calculate_jaccard_distance ( bbox1,
                        ↪  bbox2)
                        print ( jc)
                        if not jc > 0.45:
                                sub_pred_e_idx.append ( index)
                found_indexes.append ( found_index)
                pred_e_idx = sub_pred_e_idx
                if not len ( pred_e_idx) > 2:
                        break
        return found_indexes

def dump_train_output ( filename, prediction, ground, 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()
#        print ( labels)
        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,
⌋
↪  b⌋
↪  b⌋
↪  o⌋
↪  x⌋
↪  s⌋
↪  [⌋
↪  i⌋
↪  n⌋
↪  d⌋
↪  e⌋
↪  x⌋
↪  ]⌋
↪  [⌋
↪  4⌋
↪  *⌋
↪  b⌋
↪  b⌋
↪  o⌋
↪  x⌋
↪  _⌋
↪  i⌋
↪  n⌋
↪  d⌋
↪  e⌋
↪  x⌋
↪  :⌋
↪  4⌋
↪  *⌋
↪  (⌋
↪  b⌋
↪  b⌋
↪  o⌋
↪  x⌋
↪  _⌋
↪  i⌋
↪  n⌋
↪  d⌋
↪  e⌋
↪  x⌋
↪  +⌋
↪  1⌋
↪  )⌋
↪  ]⌋
↪  ,⌋
↪
↪   ⌋
↪  l⌋
↪  ,⌋
↪
↪   ⌋
↪  g⌋
↪  r⌋
↪  e⌋
↪  e⌋
↪  n⌋
```

```
if pred_exists[index][bbox_index⌋
↪  ] >
↪  threshold:
        l = class_list[np.argmax⌋
        ↪  (pred_labels[bbox_in⌋
        ↪  dex][index])]
        image =
        ↪  annotate_bound_box (
        ↪  image,
⌟
↪  p⌟
↪  r⌟
↪  e⌟
↪  d⌟
↪  _⌟
↪  b⌟
↪  b⌟
↪  o⌟
↪  x⌟
↪  s⌟
↪  [⌟
↪  i⌟
↪  n⌟
↪  d⌟
↪  e⌟
↪  x⌟
↪  ]⌟
↪  [⌟
↪  4⌟
↪  *⌟
↪  b⌟
↪  b⌟
↪  o⌟
↪  x⌟
↪  _⌟
↪  i⌟
↪  n⌟
↪  d⌟
↪  e⌟
↪  x⌟
↪  :⌟
↪  4⌟
↪  *⌟
↪  (⌟
↪  b⌟
↪  b⌟
↪  o⌟
↪  x⌟
↪  _⌟
↪  i⌟
↪  n⌟
↪  d⌟
↪  e⌟
↪  x⌟
↪  +⌟
↪  1⌟
↪  )⌟
↪  ]⌟
↪  ,⌟
↪
↪   ⌟
↪  1⌟
↪  ,⌟
↪
↪   ⌟
```

```
                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):
        losses = []
        with torch.no_grad():
##               network.eval()
                start = time.process_time()
                for iter, data in enumerate ( data_loader):
                        inputs, exist, label, bbox, yolo_vectors
                        ↪  = data

                        inputs = inputs.to ( device, dtype=dtype)
                        yolo_vectors = yolo_vectors.to ( device,
                        ↪  dtype=dtype)

                        outputs = net ( inputs)
#                        loss = reg_criterion ( outputs,
↪  yolo_vectors)
                        loss = criterion ( outputs, yolo_vectors)
                        losses.append ( loss.item())

#                        start1 = time.process_time()
#                        found_indexes = max_suspression (
↪  outputs[0])
#                        end1 = time.process_time()
#                        print ( found_indexes, ' : ',
↪  end1-start1)

                end = time.process_time()
##               network.train()
        return np.mean ( losses), end-start


dtype = torch.float32
optimizer       = torch.optim.SGD ( net.parameters(), lr=1e-6,
↪  momentum=0.9, weight_decay=0.0005)
reg_criterion       = torch.nn.MSELoss()
criterion       = yolo_loss()
```

```
for epoch in range ( epochs):
        losses = []
        p_yolo_vectors_num_tot = 0
        p_outputs_num_tot = 0
        start = time.process_time()
        for iter, data in enumerate ( train_data_loader):
                inputs, exist, label, bbox, yolo_vectors = data

                inputs = inputs.to ( device, dtype=dtype)
                yolo_vectors = yolo_vectors.to ( device,
                ↪  dtype=dtype)

                optimizer.zero_grad ()
                outputs = net ( inputs)
#                loss1 = reg_criterion ( outputs, yolo_vectors)
                loss = criterion ( outputs, yolo_vectors)
                loss.backward ()
                optimizer.step ()

                yolo_vectors        =
                ↪  yolo_vectors.detach().cpu().numpy()
                outputs                =
                ↪  outputs.detach().cpu().numpy()
                p_yolo_vectors        = yolo_vectors[:,::8]
                p_outputs        = outputs[:,::8]

                losses.append ( loss.item())
        end = time.process_time()
        val_loss, val_time = get_loss ( net, val_data_loader)
        print ( 'Epoch %3d : Loss %.6f / %.6f [ %7.3f] ' % (
        ↪  epoch, np.mean ( losses),
                        val_loss, end-start,
                        ))
        if epoch % 10 == 0 and epoch != 0:
                torch.save ( net, 'model/network_epoch_%03d.pth'
                ↪  % (epoch))
```

hw06_validation.py

```
from pycocotools.coco import COCO
import torch
import torchvision.transforms as tvt
import sys
import os
import cv2
import numpy as np
from PIL import Image
from torchsummary import summary
import time
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix
import seaborn as sns

cat_list = [ 'bus', 'truck', 'car']
cat_list = [ 'cat', 'dog', 'giraffe']
## cat_list = [ 'cat', 'dog', 'person']

train_anno_path = sys.argv[1]
val_anno_path = sys.argv[2]

train_coco = COCO ( train_anno_path)
val_coco = COCO ( val_anno_path)

image_size        = 128
grid_size        = 8
yolo_interval        = image_size // grid_size
num_archor_box        = 5
archor_aspect_ratio        = [ 1/5, 1/3, 1/1, 3/1, 5/1]

device = torch.device ( "cuda" if torch.cuda.is_available() else
↪  "cpu")

def get_image ( image_data, orig_image_path):
        image_id = image_data['id']
        filename = image_data['file_name']
        print ( filename)
#        os.system ( '')
        return None

def process_image ( image_data, scale, orig_image_path,
↪  process_image_path):
        max_height = 640
        max_width = 640

        image = cv2.imread ( orig_image_path)

        black = ( 0, 0, 0)
        height = image_data['height']
        width  = image_data['width']
        blank_image = np.zeros ( ( max_height, max_width, 3),
        ↪  np.uint8)
        blank_image[:height,:width,:] = image
        blank_image = cv2.resize ( blank_image, (0,0), fx=scale,
        ↪  fy=scale)

        cv2.imwrite ( process_image_path, blank_image)

        return None

class dataset ( torch.utils.data.Dataset):
        def __init__ ( self, coco, cat_list, max_instance,
        ↪  isTrain=True):
                self.filename = []
                self.exist    = []
                self.label    = []
                self.bbox     = []
                self.yolo_vectors     = []
                self.transform = tvt.Compose ( [ tvt.ToTensor
                ↪  (), tvt.Normalize ( ( 0.5, 0.5, 0.5), ( 0.5,
                ↪  0.5, 0.5))])

                scale = 0.2 # 640 -> 128

                image_ids = []
                catagory    = { index:coco.loadCats (
                ↪  index)[0]['name'] for index in
                ↪  coco.getCatIds ()}
                rcatagory   = { coco.loadCats (
                ↪  index)[0]['name']:index for index in
                ↪  coco.getCatIds ()}

                for cat in cat_list:
                        cat_id = rcatagory[cat]
                        cat_image_ids = coco.getImgIds (
                        ↪  catIds=[cat_id])
                        image_ids = image_ids + cat_image_ids

                image_ids = list ( set ( image_ids))
                print ( len ( image_ids))
                image_datas = coco.loadImgs  ( ids=image_ids)
                for image_data in image_datas:
                        train = 'train'
                        if not isTrain:
                                train = 'val'

                        image_id = image_data['id']
                        filename = image_data['file_name']
```

```
orig_image_path = 'dataset/%s/%s' % (
↪  train, filename)
if not os.path.isfile ( orig_image_path):
        get_image ( image_data,
         ↪  orig_image_path)
process_image_path =
↪  'dataset/%s_process/%s' % ( train,
↪  filename)
if not os.path.isfile (
↪  process_image_path):
        process_image ( image_data,
         ↪  scale, orig_image_path,
         ↪  process_image_path)

ann_ids = [ ann_id for cat in cat_list
↪  for ann_id  in coco.getAnnIds (
↪  imgIds=image_id,
↪  catIds=rcatagory[cat],
↪  iscrowd=False)]
if not ( 0 < len ( ann_ids) <=
↪  max_instance):
        continue
image_anns = coco.loadAnns ( ann_ids)

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

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

        exists[index] = 1
        bboxs[4*index:4*index+4] = bbox
        labels[index] = category_id

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

```
##                        print ( '>>>  ', process_image_path)
                        # 1 Anchor Box => [ e, c_x, c_y, b_h,
                        ↪  b_w, c1, c2, c3]
                        archor_length = 1+4+3
                        yolo_vector = np.zeros ( grid_size *
                        ↪  grid_size * num_archor_box *
                        ↪  archor_length)
                        for index in range ( len ( image_anns)):
                                archor_vector = np.zeros ( 1+4+3)
                                image_ann = image_anns[index]

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

                                tl_x, tl_y, w, h = bbox
##                                c_x = ( tl_x + w)/2
##                                c_y = ( tl_y + h)/2
                                c_x = tl_x + w/2
                                c_y = tl_y + h/2

                                aspect_ratio = w/h
                                id_x = c_x // yolo_interval
                                id_y = c_y // yolo_interval

                                del_x = id_x + 0.5 - c_x /
                                ↪  yolo_interval
                                del_y = id_y + 0.5 - c_y /
                                ↪  yolo_interval

                                yolo_w = float ( w) / float (
                                ↪  yolo_interval)
                                yolo_h = float ( h) / float (
                                ↪  yolo_interval)

                                archor_vector[0] = 1
                                archor_vector[1] = del_x
                                archor_vector[2] = del_y
                                archor_vector[3] = yolo_w
                                archor_vector[4] = yolo_h
                                sub_cat_id = cat_list.index (
                                ↪  catagory[category_id])
                                archor_vector[5+sub_cat_id] = 1

                                archor_index = -1
```

```
delta = 0.01
for index in range ( len (
↪  archor_aspect_ratio)):
        if index == 0:
                hi_limit  = ( ar⌋
                ↪  chor_aspect_⌋
                ↪  ratio[index]
                ↪  + archor_asp⌋
                ↪  ect_ratio[in⌋
                ↪  dex+1])/2
                if aspect_ratio
                ↪  < hi_limit:
                        archor_i⌋
                        ↪  ndex
                        ↪  =
                        ↪  index
        elif 0 < index < len (
        ↪  archor_aspect_ratio)⌋
        ↪  -1:
                low_limit = (
                ↪  archor_aspec⌋
                ↪  t_ratio[inde⌋
                ↪  x-1] +
                ↪  archor_aspec⌋
                ↪  t_ratio[inde⌋
                ↪  x])/2
                hi_limit  = ( ar⌋
                ↪  chor_aspect_⌋
                ↪  ratio[index]
                ↪  + archor_asp⌋
                ↪  ect_ratio[in⌋
                ↪  dex+1])/2
                if low_limit <=
                ↪  aspect_ratio
                ↪  <= hi_limit:
                        archor_i⌋
                        ↪  ndex
                        ↪  =
                        ↪  index
        else:
```

```
                                                low_limit = (
                                                ↪  archor_aspec⌋
                                                ↪  t_ratio[inde⌋
                                                ↪  x-1] +
                                                ↪  archor_aspec⌋
                                                ↪  t_ratio[inde⌋
                                                ↪  x])/2
                                                if low_limit <
                                                ↪  aspect_ratio:
                                                        archor_i⌋
                                                        ↪  ndex
                                                        ↪  =
                                                        ↪  index
                                yolo_archor_index = int ( (
                                ↪  archor_index + (
                                ↪  id_y*grid_size + id_x)*num_a⌋
                                ↪  rchor_box)*archor_length)
                                yolo_vector[yolo_archor_index:yo⌋
                                ↪  lo_archor_index+archor_lengt⌋
                                ↪  h] =
                                ↪  archor_vector
##                                print ( '\t\t', bbox, ' / ',
↪  image_ann['bbox'], ' : ', category_id, ' :: ',
↪  yolo_archor_index, ' [ ', id_x, ' , ', id_y, ' ]')
                        self.yolo_vectors.append ( yolo_vector)


                print ( 'Length = ', len ( self.filename))

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

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

                return data, exist, label, bbox, yolo_vectors,
                ↪  self.filename[idx]
```

```
train_dataset = dataset ( train_coco, cat_list, 5, True)
val_dataset = dataset ( val_coco, cat_list, 5, False)

batch_size=32
train_data_loader = torch.utils.data.DataLoader (
↪  dataset=train_dataset,
        batch_size=batch_size, shuffle=True, num_workers=4)
val_data_loader = torch.utils.data.DataLoader (
↪  dataset=val_dataset,
        batch_size=batch_size, shuffle=True, num_workers=4)

class block ( torch.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 ( torch.nn.Module):
        def __init__ ( self):
                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.maxp = torch.nn.MaxPool2d ( kernel_size=2,
                ↪  stride=None, padding=0, dilation=1,
                ↪  return_indices=False, ceil_mode=False)
#                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.fc = torch.nn.Linear ( 256*8*8, 40*8*8)
                self.relu = torch.nn.LeakyReLU (
                ↪  negative_slope=0.01, inplace=False)

                self.drop = torch.nn.Dropout ( p=0.5)

        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.drop ( x)
                x = self.bloc128i (x)
                for i in range ( 4-m):
                        x = self.bloc128 (x)
                        x = self.drop ( x)
                x = self.bloc256i (x)
                for i in range ( 4-m):
                        x = self.bloc256 (x)
                        x = self.drop ( x)
#                x = self.bloc512i (x)
#                for i in range ( 4-m):
#                        x = self.bloc512 (x)
##                print ( x.size())

                x = x.view ( x.size (0), -1)
                x = self.fc (x)
                x = self.relu (x)
##                x = x.view ( x.size (0), 40, 8, 8)
###                print ( x.size())

#                exit ()
```

```
                return x


threshold = 0.5
class yolo_loss ( torch.nn.Module):
        def __init__ ( self):
                super().__init__()

                self.mse = torch.nn.MSELoss()

                self.lambda_class = 1
                self.lambda_noobj = 10
                self.lambda_obj = 1
                self.lambda_box = 10

        def forward ( self, pred, ground):
                loss_init = False

                obj = ground[:,0::8] == 1
                nobj = ground[:,0::8] == 0

                '''
                loss_e = torch.square( pred[:,0::8][obj] -
                ↪  ground[:,0::8][obj])

                loss_x = torch.square( pred[:,1::8][obj] -
                ↪  ground[:,1::8][obj])
                loss_y = torch.square( pred[:,2::8][obj] -
                ↪  ground[:,2::8][obj])

                loss_w = torch.square( torch.sqrt (
                ↪  pred[:,3::8][obj]) - torch.sqrt (
                ↪  ground[:,3::8][obj]))
                loss_h = torch.square( torch.sqrt (
                ↪  pred[:,4::8][obj]) - torch.sqrt (
                ↪  ground[:,4::8][obj]))
                '''


                loss_e = self.mse( pred[:,0::8][obj],
                ↪  ground[:,0::8][obj])
                loss_ne = self.mse( pred[:,0::8][nobj],
                ↪  ground[:,0::8][nobj])
##                print ( loss_e)
```

```
            loss_x = self.mse( pred[:,1::8][obj],
            ↪  ground[:,1::8][obj])
            loss_y = self.mse( pred[:,2::8][obj],
            ↪  ground[:,2::8][obj])
##              print ( loss_x)
##              print ( loss_y)

##              loss_w = self.mse( torch.sqrt (
↪  pred[:,3::8])[obj], torch.sqrt ( ground[:,3::8])[obj])
##              loss_h = self.mse( torch.sqrt (
↪  pred[:,4::8])[obj], torch.sqrt ( ground[:,4::8])[obj])
            loss_w = self.mse( pred[:,3::8][obj],
            ↪  ground[:,3::8][obj])
            loss_h = self.mse( pred[:,4::8][obj],
            ↪  ground[:,4::8][obj])
##              print ( loss_w)
##              print ( loss_h)

##              loss_obj = 0
##              loss_nobj = 0
##              for c_idx in [ 5, 6, 7]:
##                      '''
##                      loss_obj += torch.square(
↪  pred[:,c_idx::8][obj] - ground[:,c_idx::8][obj])
##                      loss_nobj += torch.square(
↪  pred[:,c_idx::8][nobj] - ground[:,c_idx::8][nobj])
##                      '''

            c_idx = 5
            loss_obj = self.mse( pred[:,c_idx::8][obj],
            ↪  ground[:,c_idx::8][obj])
            loss_nobj = self.mse( pred[:,c_idx::8][nobj],
            ↪  ground[:,c_idx::8][nobj])

            c_idx = 6
            loss_obj += self.mse( pred[:,c_idx::8][obj],
            ↪  ground[:,c_idx::8][obj])
            loss_nobj += self.mse( pred[:,c_idx::8][nobj],
            ↪  ground[:,c_idx::8][nobj])

            c_idx = 7
            loss_obj += self.mse( pred[:,c_idx::8][obj],
            ↪  ground[:,c_idx::8][obj])
            loss_nobj += self.mse( pred[:,c_idx::8][nobj],
            ↪  ground[:,c_idx::8][nobj])
```

```
##               print ( loss_obj)
##               print ( loss_nobj)

##               loss = self.lambda_box ( loss_x + loss_y +
 ↪  loss_w + loss_h) + loss_obj + self.lambda_noobj*loss_nobj +
 ↪  loss_e
#               loss = self.lambda_box ( loss_x + loss_y +
 ↪  loss_w + loss_h) # + loss_obj + self.lambda_noobj*loss_nobj
 ↪  + loss_e
###               loss = 10 * ( loss_x + loss_y + loss_w +
 ↪  loss_h) + loss_obj + 10*loss_nobj + loss_e
                loss = 10 * ( loss_x + loss_y + loss_w + loss_h)
                 ↪  + loss_obj + 10*loss_ne + loss_obj
                return loss

#               ground = ground.detach().cpu().numpy()
#               pred   = pred.detach().cpu().numpy()
#               for box_idx in range (
 ↪  grid_size*grid_size*num_archor_box):
#                       if ground[box_idx*8] > threshold:

def calculate_jaccard_distance ( bbox1, bbox2):
        bbox1_cx_min, bbox1_cy_min, bbox1_width, bbox1_height =
         ↪  bbox1
        bbox2_cx_min, bbox2_cy_min, bbox2_width, bbox2_height =
         ↪  bbox2

        bbox1_x_min = bbox1_cx_min - bbox1_width/2
        bbox1_y_min = bbox1_cy_min - bbox1_height/2
        bbox1_x_max = bbox1_cx_min + bbox1_width/2
        bbox1_y_max = bbox1_cy_min + bbox1_height/2

        bbox2_x_min = bbox2_cx_min - bbox2_width/2
        bbox2_y_min = bbox2_cy_min - bbox2_height/2
        bbox2_x_max = bbox2_cx_min + bbox2_width/2
        bbox2_y_max = bbox2_cy_min + bbox2_height/2

        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)/(
        ↪  1e-6 + float ( union_area))

        return jaccard_distance

def max_suspression ( pred_yolo_vectors):
        pred_e = pred_yolo_vectors[0::8].detach().cpu().numpy()
        pred_e_idx = list( reversed ( np.argsort ( pred_e)))

        found_indexes = []
        for i in range ( 5):
                found_index = int ( pred_e_idx[0])

                del_x, del_y, yolo_w, yolo_h = pred_yolo_vectors⌋
                ↪  [found_index*8+1:found_index*8+5]
                w = yolo_w * float ( yolo_interval)
                h = yolo_h * float ( yolo_interval)
                _found_index = found_index - found_index%5
                _found_index = _found_index // 5
                x = _found_index % 8
                y = _found_index // 8
                c_x = yolo_interval * ( x + 0.5 - del_x)
                c_y = yolo_interval * ( y + 0.5 - del_y)
                bbox1 = c_x, c_y, w, h


                list_pred_index = pred_e_idx[1:]
                sub_pred_e_idx = []
                for index in list_pred_index:
                        index = int ( index)
                        bbox2 = pred_yolo_vectors[index*8+1:inde⌋
                        ↪  x*8+5]

                        del_x, del_y, yolo_w, yolo_h = pred_yolo⌋
                        ↪  _vectors[index*8+1:index*8+5]
                        w = yolo_w * float ( yolo_interval)
                        h = yolo_h * float ( yolo_interval)
                        _found_index = found_index -
                        ↪  found_index%5
                        _found_index = _found_index // 5
                        x = _found_index % 8
                        y = _found_index // 8
                        c_x = yolo_interval * ( x + 0.5 - del_x)
                        c_y = yolo_interval * ( y + 0.5 - del_y)
                        bbox2 = c_x, c_y, w, h


                        jc = calculate_jaccard_distance ( bbox1,
                        ↪  bbox2)
                        if not jc > 0.45:
                                sub_pred_e_idx.append ( index)
                found_indexes.append ( found_index)
                pred_e_idx = sub_pred_e_idx
                if not len ( pred_e_idx) > 2:
                        break
        return found_indexes

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
        c_x, c_y, w, h = bbox

        width = w
        height = h
        x_min = c_x - width/2
        y_min = c_y - height/2

        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, 128)
        y_max        = min ( y_max, 128)

##        print ( '\t\t\t', x_min, ' ', y_min , ' : ', x_max, '
↪  ', y_max)

        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, prediction, ground,
↪  found_indexes_vec, gt_found_indexes_vec, dumppath):
        n        = 5
        n_index        = 0
        plt.figure ( figsize=(20, 4))
        green        = [ 0, 255, 0]
        red        = [ 255, 0, 0]

        ground = ground.detach().cpu().numpy()
        prediction = prediction.detach().cpu().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)

##                       print ( filename[index])
                        for found_index in
                        ↪  gt_found_indexes_vec[index]:
                                del_x, del_y, yolo_w, yolo_h =
                                ↪  ground[index][found_index*8+⌋
                                ↪  1:found_index*8+5]
                                w = yolo_w * float (
                                ↪  yolo_interval)
                                h = yolo_h * float (
                                ↪  yolo_interval)
                                _found_index = found_index -
                                ↪  found_index%5
                                _found_index = _found_index // 5
                                x = _found_index % 8
                                y = _found_index // 8
                                c_x = yolo_interval * ( x + 0.5
                                ↪  - del_x)
                                c_y = yolo_interval * ( y + 0.5
                                ↪  - del_y)
                                bbox = c_x, c_y, w, h
                                c_id = np.argmax (
                                ↪  ground[index][found_index*8+⌋
                                ↪  5:found_index*8+8])
##                                print ( '\t\t%f %f %f %f : %d
↪  [ %d , %d ]' % (c_x, c_y, w, h, found_index, x, y))
                                image = annotate_bound_box (
                                ↪  image, bbox, cat_list[c_id],
                                ↪  green, 'top')


                        for found_index in
                        ↪  found_indexes_vec[index]:
                                del_x, del_y, yolo_w, yolo_h =
                                ↪  prediction[index][found_inde⌋
                                ↪  x*8+1:found_index*8+5]
                                w = yolo_w * float (
                                ↪  yolo_interval)
                                h = yolo_h * float (
                                ↪  yolo_interval)
                                _found_index = found_index -
                                ↪  found_index%5
                                _found_index = _found_index // 5
                                x = _found_index % 8
                                y = _found_index // 8
                                c_x = yolo_interval * ( x + 0.5
                                ↪  - del_x)
                                c_y = yolo_interval * ( y + 0.5
                                ↪  - del_y)
                                bbox = c_x, c_y, w, h
                                c_id = np.argmax (
                                ↪  prediction[index][found_inde⌋
                                ↪  x*8+5:found_index*8+8])
##                                print ( '\t\t%f %f %f %f : %d
↪  [ %d , %d ]' % (c_x, c_y, w, h, found_index, x, y))
                                image = annotate_bound_box (
                                ↪  image, bbox, cat_list[c_id],
                                ↪  red, 'bottom')

#                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.show ()
#                                exit ()
                                plt.savefig ( dumppath,
                                 ↪  bbox_inches='tight')
                                plt.close ()

def get_loss ( network, data_loader):
        losses = []
        network.eval()
        first = True
        gt_cat = []
        pred_cat = []
        ji = []
        with torch.no_grad():
                start = time.process_time()
                for iter, data in enumerate ( data_loader):
                        inputs, exist, label, bbox,
                         ↪  yolo_vectors, filename = data

                        inputs = inputs.to ( device, dtype=dtype)
                        yolo_vectors = yolo_vectors.to ( device,
                         ↪  dtype=dtype)

                        outputs = network ( inputs)
                        loss = criterion ( outputs, yolo_vectors)
                        losses.append ( loss.item())

                        found_indexes_vec = []
                        for index in range ( len ( outputs)):
```

```
        found_indexes = max_suspression
         ↪  ( outputs[index])
        found_indexes_vec.append (
         ↪  found_indexes)

obj = yolo_vectors[:,0::8] == 1
obj = obj.cpu().detach().numpy()
gt_found_indexes_vec = []
for index in range ( len ( obj)):
        gt_indexes = []
        for sub_index in range ( len (
         ↪  obj[index])):
                if obj[index][sub_index]:
                        gt_indexes.appen⌋
                         ↪  d (
                         ↪  sub_index)
        gt_found_indexes_vec.append (
         ↪  gt_indexes)
if first:
        dumppath = './Val.png'
        dump_train_output ( filename,
         ↪  outputs, yolo_vectors,
         ↪  found_indexes_vec,
         ↪  gt_found_indexes_vec,
         ↪  dumppath)
        first = False

yolo_vectors_pred =
 ↪  outputs.detach().cpu().numpy()
yolo_vectors =
 ↪  yolo_vectors.detach().cpu().numpy()
for batch_index in range ( len (
 ↪  yolo_vectors)):
        for found_index_idx in range (
         ↪  len ( gt_found_indexes_vec[b⌋
         ↪  atch_index])):

                found_index = gt_found_i⌋
                 ↪  ndexes_vec[batch_ind⌋
                 ↪  ex][found_index_idx]
                del_x, del_y, yolo_w,
                 ↪  yolo_h =
                 ↪  yolo_vectors[batch_i⌋
                 ↪  ndex][found_index*8+⌋
                 ↪  1:found_index*8+5]
```

```
w = yolo_w * float (
↪  yolo_interval)
h = yolo_h * float (
↪  yolo_interval)
_found_index =
↪  found_index -
↪  found_index%5
_found_index =
↪  _found_index // 5
x = _found_index % 8
y = _found_index // 8
c_x = yolo_interval * (
↪  x + 0.5 - del_x)
c_y = yolo_interval * (
↪  y + 0.5 - del_y)
bbox1 = c_x, c_y, w, h
c_id = np.argmax (
↪  yolo_vectors[batch_i⌋
↪  ndex][found_index*8+⌋
↪  5:found_index*8+8])
gt_cat.append ( int (
↪  c_id))

del_x, del_y, yolo_w,
↪  yolo_h = yolo_vector⌋
↪  s_pred[batch_index][⌋
↪  found_index*8+1:foun⌋
↪  d_index*8+5]
w = yolo_w * float (
↪  yolo_interval)
h = yolo_h * float (
↪  yolo_interval)
_found_index =
↪  found_index -
↪  found_index%5
_found_index =
↪  _found_index // 5
x = _found_index % 8
y = _found_index // 8
c_x = yolo_interval * (
↪  x + 0.5 - del_x)
c_y = yolo_interval * (
↪  y + 0.5 - del_y)
bbox2 = c_x, c_y, w, h
```

```
                                        c_id = np.argmax ( yolo_⌋
                                        ↪  vectors_pred[batch_i⌋
                                        ↪  ndex][found_index*8+⌋
                                        ↪  5:found_index*8+8])
                                        pred_cat.append ( int (
                                        ↪  c_id))

                                        jc = calculate_jaccard_d⌋
                                        ↪  istance ( bbox1,
                                        ↪  bbox2)
                                        ji.append ( jc)


                end = time.process_time()
##               network.train()
        print ( 'Average Jaccard Index = ', np.mean ( ji))
        confusion_matrix_model = confusion_matrix ( gt_cat,
        ↪  pred_cat)
        sns.heatmap ( confusion_matrix_model,
                xticklabels=cat_list, yticklabels=cat_list,
                annot=True, fmt='d', cmap='Blues', square=True,
                ↪  cbar=False)

        gt_cat = np.array ( gt_cat)
        pred_cat = np.array ( pred_cat)
        accuracy = sum ( gt_cat == pred_cat)/len ( gt_cat)

#        accuracy = 0
        plt.ylabel ( 'True label')
        plt.xlabel ( 'Predicted label\nAccuracy=%.3f' % (
        ↪  accuracy))
#        plt.xlabel ( 'Predicted label')
        plt.show ()

        return np.mean ( losses), end-start

## net = network ().to ( device)

print ( sys.argv[3])
network_model = torch.load ( sys.argv[3])
network_model                = network_model.to ( device)

dtype = torch.float32
## optimizer       = torch.optim.SGD ( net.parameters(),
↪  lr=1e-6, momentum=0.9, weight_decay=0.0005)
reg_criterion       = torch.nn.MSELoss()
```

```
criterion        = yolo_loss()
get_loss ( network_model, val_data_loader)
```

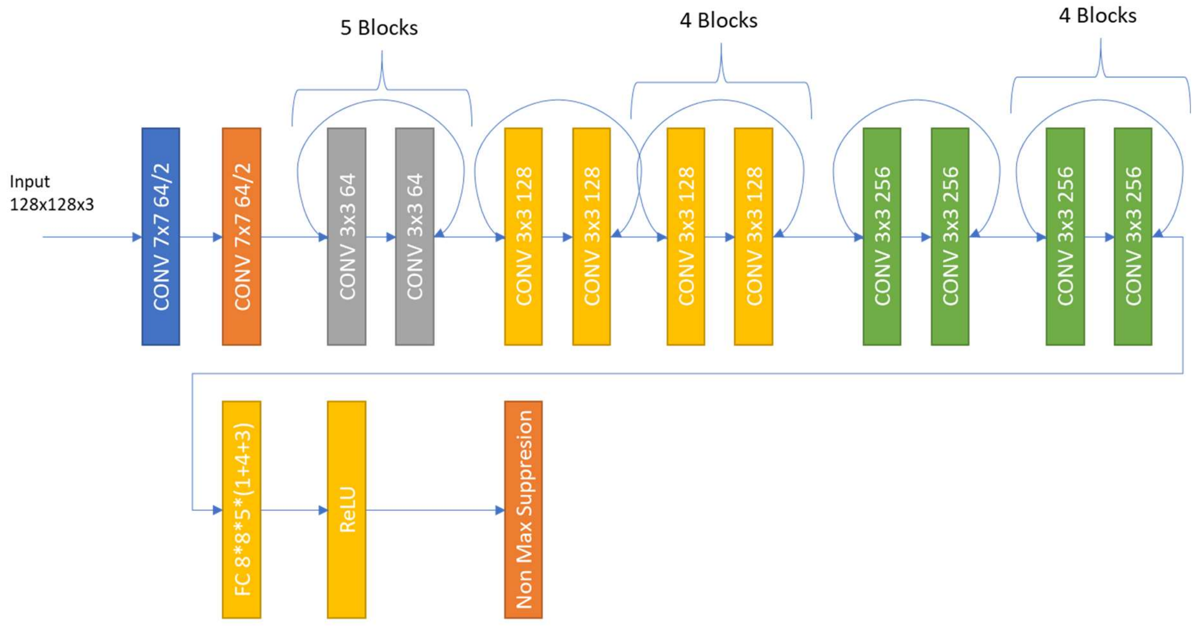
Author: Pranab Dash

Homework 6: Multi-Object detection and localization

To detect multi-object in the image I have tried to implement a version on YOLO (You Only Look Once).

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.

Final Architecture:



Loss Function used:
A modified form of the loss function from the YOLO v1 and v3 was used. I was not able to get the square root of the tensor without getting a nan value. So, I have modified the loss function to.

$$loss = \alpha \sum_{i=}^{S\text{^}2} \sum_{j=0}^{B} I_{ij}^{obj} \left[(x_i - x\_p_i)^2 + (y_i - y\_p_i)^2\right]$$
$$+ \alpha \sum_{i=}^{S\text{^}2} \sum_{j=0}^{B} I_{ij}^{obj} \left[(w_i - w\_p_i)^2 + (h_i - h\_p_i)^2\right] + \sum_{i=}^{S\text{^}2} \sum_{j=0}^{B} I_{ij}^{obj} \left[(c_i - c\_p_i)^2\right]$$
$$+ \beta \sum_{i=}^{S\text{^}2} \sum_{j=0}^{B} I_{ij}^{obj} \left[(c_i - c\_p_i)^2\right] + \sum_{i=}^{S\text{^}2} \sum_{All\ Classes} I_{ij}^{obj} \left[(p_i - p\_p_i)^2\right]$$

Where:

$$\alpha = \beta = 10$$
$$x \text{ and } y \text{ are the delta from the center of the anchor box}$$
$$w \text{ and } h \text{ are the height and width of the bounding box nornalized with yolo interval}$$
$$p \text{ is } 1 \text{ if the object exsits in the anchor box otherwise } 0$$

Anchor box definition:
I tried various aspect ratios for anchor boxes with no better results. Some combination tried are: (width/height)

1. [2/3, 1/3, 1/1, 3/1, 3/2]
2. [2/7, 3/5, 1/1, 5/3, 7/2]
3. [3/7, 2/5, 1/1, 5/2, 7/3]

The final the design has 5 anchor boxes with the following aspect ratios. (width/height)
[1/5, 1/3, 1/1, 3/1, 5/1]

The number of anchor boxes was also varied but there was not much change in accuracy. So, in final design 5 anchor boxes were chosen according to the lecture notes:

Grid Size:
I have tried grid of 6x6, 7x7 and 8x8. The loss seems to decrease as the grid becomes tighter, but the accuracy remains bad.

Use of Dropout:
The model seems not getting generalized properly so a dropout layer was introduced after every skip block of dropout rate of 0.5.

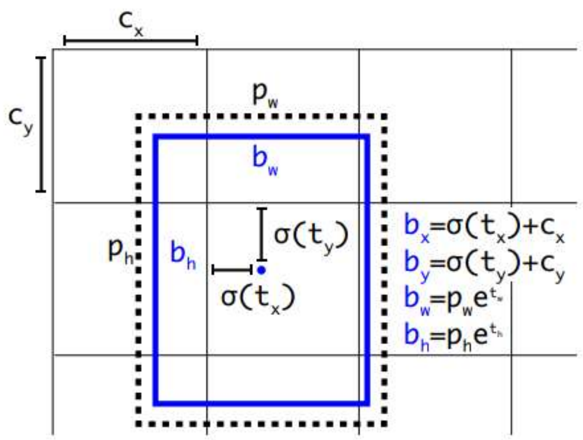
Choice of Images:
Three categories were chosen for the design with the maximum of 5 object instances in an image.
The following combination of categories was tried upon.

1. Bus, Truck, Person
2. Bus, Truck, Car
3. Cat, Dog, Person
4. Cat, Dog, Giraffe

The final choice for the evaluation was made for the categories [Cat, Dog, Giraffe]. But the accuracy did not change much compared with other combinations.

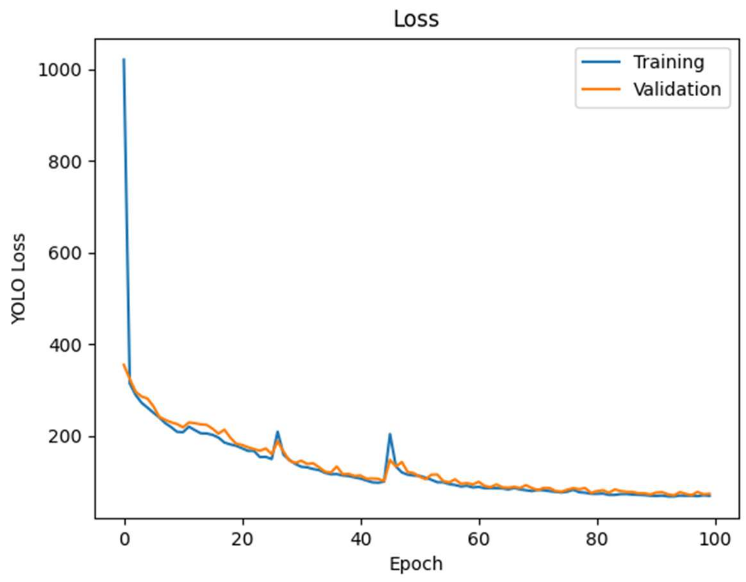
For any of the above 3 categories with having 2 instance and more caused the training set to shrink to be in 1000's of images. This has caused me to take all the images which contain any of these 3 categories with one instance or more.

Anchor Box description:



The calculation for delta on x and y from the center point on the anchor box was done according to YOLO v3. Whereas the height and width were just normalized w.r.t. YOLO interval length (i.e. b_h = height/yolo_interval)

Evaluation:
Loss Plot

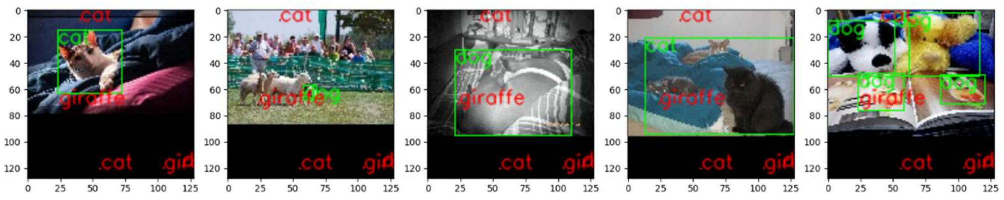


The loss shows that there are two major problems with implemented model:

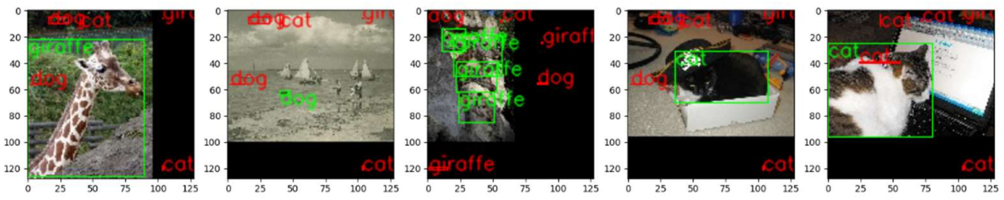
1. The validation loss is flowing too close to the training loss signifying that model is not able to capture the adequate features.
2. The training curve shows that it is not able to overfit. This implies that the model needs to more complicated to capture the features. As I tried to apply more complicated (i.e., more number parameters by adding skip blocks) model the training time increased, and I could not reach an acceptable loss point.

Average Jaccard Index (IoU) = 0.16797059891282887

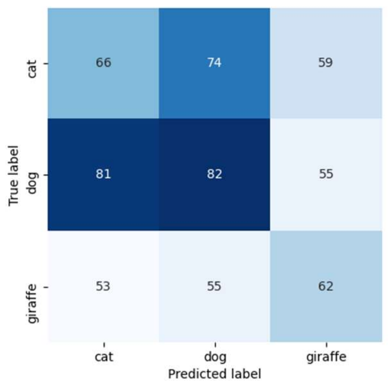
Epoch 50:



Epoch 90:



Confusion Matrix:



The classification accuracy = 35.77 %

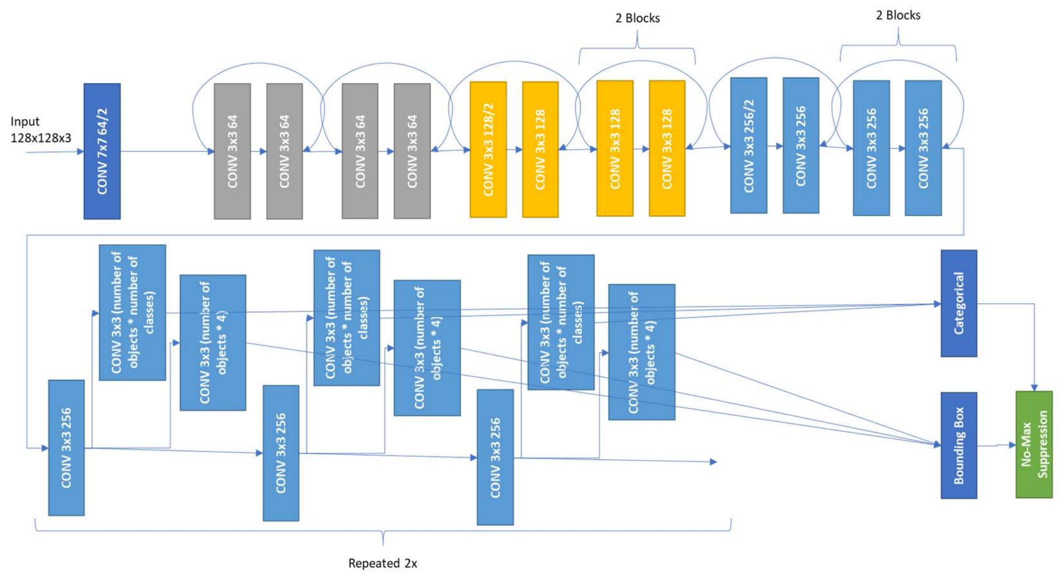
Other approaches tied to increase accuracy:

1. Pretrained the skip net block network for only classification using image patch extracted of the object instances from the training set. This only have an accurate which did not exceed 60%.

Comparison with Homework 5, approach 2:

I had implemented a version on Single Shot Detector for Homework 5.

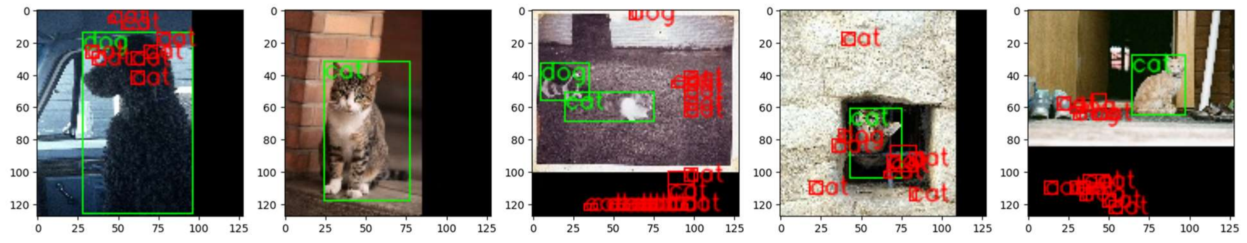
Architecture was:



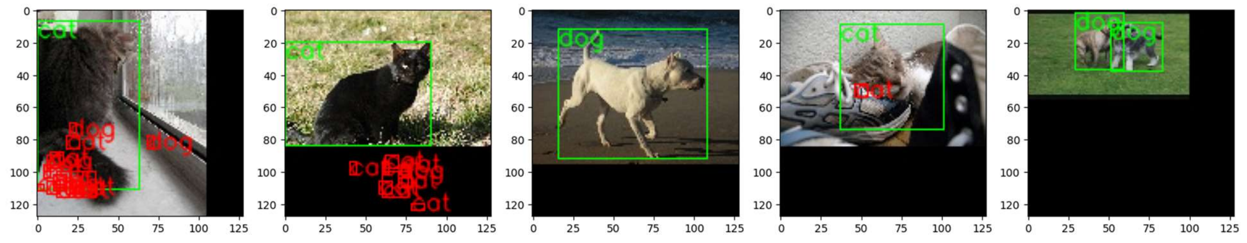
Evaluation:

1. Epoch 20:

   Training Set:



   Validation Set:



The homework 6 and 5 have similar associated problems. The model required has to much more complex and needs more training time.

Conclusion:

1. The number of parameters needs to be much higher to capture the complexity of the task and improve accuracy.
2. The input image size must increase for better accuracy. In the input size used in the paper is 224x244 or 448x448.
3. The time needed to train must be in days.

References:

[1] https://pytorch.org/docs/stable/optim.html
[2] https://pytorch.org/docs/stable/generated/torch.nn.Dropout.html
[3] https://www.youtube.com/watch?v=Grir6TZbc1M
[4] https://www.youtube.com/watch?v=n9_XyCGr-MI
[5] https://arxiv.org/pdf/1804.02767.pdf
[6] https://arxiv.org/pdf/1506.02640.pdf
[7] https://pytorch.org/docs/stable/generated/torch.sqrt.html
[8] https://pytorch.org/docs/stable/generated/torch.square.html
[9] https://github.com/aladdinpersson/Machine-Learning-Collection
[10] https://github.com/aladdinpersson/Machine-Learning-Collection/blob/master/ML/Pytorch/object_detection/YOLO/loss.py