ECE695DL: Homework 5

Chohsin Tsai

28 March 2021

hw05_training.py

```
'''
The designed structure of the code is similar to
object_detection_and_localization.py (written by Prof. Kak).

However, the modifications are all written by myself.

The program will download COCO images of 5 classes ('cat',
'train', 'elephant', 'airplane', 'giraffe') using
instances_train2017.json.


Each class has 1250 images, and each image will be resized to
64 x 64.

Each image's pixel values and its annotations will be extracted
and saved to the JSON file dict_train2017.json.

The program will train two neural networks: net1 and net2.

net1 is designed without skip connections, whereas net2 uses
skip connections.

The training results (loss values and images with bboxes) will
be saved to the directory 'Train Results.'

net1.pth and net2.pth will be saved to the current directory.

'''

import os, sys
import random
import json
import numpy as np
import torch
import torchvision
import torch.nn as nn
```

```
import torch.nn.functional as F
import torch.optim as optim
from torch.utils.data import DataLoader, Dataset
import torchvision.transforms as tvt
from PIL import Image
import requests
from requests.exceptions import
ConnectionError,ReadTimeout,TooManyRedirects,MissingSchema,
InvalidURL
from pycocotools.coco import COCO
import copy
import pickle
import gzip
import matplotlib.pyplot as plt

# ensure reproducibility
seed = 0
random.seed(seed)
torch.manual_seed(seed)
torch.cuda.manual_seed(seed)
np.random.seed(seed)
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmarks = False
os.environ['PYTHONHASHSEED'] = str(seed)

class MyDLStudio(object):
    def __init__(self, *args, **kwargs ):
        if args:
            raise ValueError('''Only keyword arguments are
            allowed''')
        learning_rate = epochs = batch_size = momentum = None
        image_size = dataroot =  path_saved_model = classes =
        use_gpu = None
        if 'dataroot' in kwargs                        :
        dataroot = kwargs.pop('dataroot')
        if 'learning_rate' in kwargs                   :
        learning_rate = kwargs.pop('learning_rate')
        if 'momentum' in kwargs                        :
        momentum = kwargs.pop('momentum')
        if 'epochs' in kwargs                          :
        epochs = kwargs.pop('epochs')
        if 'batch_size' in kwargs                      :
        batch_size = kwargs.pop('batch_size')
        if 'path_saved_model' in kwargs                :
        path_saved_model = kwargs.pop('path_saved_model')
        if 'classes' in kwargs                         :
        classes = kwargs.pop('classes')
        if 'use_gpu' in kwargs                         :
        use_gpu = kwargs.pop('use_gpu')
        if 'image_size' in kwargs                      :
        image_size = kwargs.pop('image_size')
```

```
    if 'use_gpu' in kwargs                         :
    use_gpu = kwargs.pop('use_gpu')
    if 'debug_train' in kwargs                     :
    debug_train = kwargs.pop('debug_train')
    if 'debug_test' in kwargs                      :
    debug_test = kwargs.pop('debug_test')
    if dataroot:
        self.dataroot = dataroot
    if image_size:
        self.image_size = image_size
    if  path_saved_model:
        self.path_saved_model = path_saved_model
    if classes:
        self.class_labels = classes
        self.num_classes = len(classes)
    if learning_rate:
        self.learning_rate = learning_rate
    else:
        self.learning_rate = 1e-6
    if momentum:
        self.momentum = momentum
    if epochs:
        self.epochs = epochs
    if batch_size:
        self.batch_size = batch_size
    if use_gpu is not None:
        self.use_gpu = use_gpu
        if use_gpu is True:
            if torch.cuda.is_available():
                self.device = torch.device("cuda:0")
            else:
                raise Exception("No GPU on this machine")
        else:
            self.device = torch.device("cpu")
    if debug_train:
        self.debug_train = debug_train
    else:
        self.debug_train = 0
    if debug_test:
        self.debug_test = debug_test
    else:
        self.debug_test = 0

def display_tensor_as_image(self, tensor, title=""):
    tensor_range = (torch.min(tensor).item(),
    torch.max(tensor).item())
    if tensor_range == (-1.0,1.0):
        ##  The tensors must be between 0.0 and 1.0
        for the display:
        print("\n\n\nimage un-normalization called")
        tensor = tensor/2.0 + 0.5     # unnormalize
```

```
        font_size = 20
        font = {'size': font_size}
        plt.rc('font', **font)
        plt.figure(figsize = (20,200))
        img_title = title.replace('_', ' ')
        plt.title(img_title)

        ###  The call to plt.imshow() shown below needs a
        numpy array. We must also
        ###  transpose the array so that the number of
        channels (the same thing as the
        ###  number of color planes) is in the last element.
        For a tensor, it would be in
        ###  the first element.
        if tensor.shape[0] == 3 and len(tensor.shape) == 3:
#            plt.imshow( tensor.numpy().transpose(1,2,0) )
            plt.imshow( tensor.numpy().transpose(1,2,0) )
        ###  If the grayscale image was produced by calling
        torchvision.transform's
        ###  ".ToPILImage()", and the result converted to a
        tensor, the tensor shape will
        ###  again have three elements in it, however the first
        element that stands for
        ###  the number of channels will now be 1
        elif tensor.shape[0] == 1 and len(tensor.shape) == 3:
            tensor = tensor[0,:,:]
            plt.imshow( tensor.numpy(), cmap = 'gray' )
        ###  For any one color channel extracted from the
        tensor representation of a color
        ###  image, the shape of the tensor will be (W,H):
        elif len(tensor.shape) == 2:
            plt.imshow( tensor.numpy(), cmap = 'gray' )
        else:
            sys.exit("\n\n\nfrom 'display_tensor_as_image()':
            tensor for image is ill formed -- aborting")

        path_saved_image = './Train Results/'
        if not os.path.exists(path_saved_image):
            os.makedirs(path_saved_image)

        img_title_save = path_saved_image +
        title.lower().replace('[', "").replace(']',
        "").replace(',', "")

        plt.savefig(img_title_save + '.png', bbox_inches =
        "tight")
        im = Image.open(img_title_save + '.png')
        if im.mode in ("RGBA", "P"):
            im = im.convert("RGB")
        im.save(img_title_save + '.jpg','JPEG', quality = 95)
```

```
    os.remove(img_title_save + '.png')

    plt.show()

class DetectAndLocalize(nn.Module):
    def __init__(self, dl_studio, dataserver_train=None,
    dataserver_test=None, dataset_file_train=None,
    dataset_file_test=None):
        super().__init__()
        self.dl_studio = dl_studio
        self.dataserver_train = dataserver_train
        self.dataserver_test = dataserver_test
        self.debug = False

    class COCO5Dataset(torch.utils.data.Dataset):
        def __init__(self, dl_studio, train_or_test,
        transform=None):
            super().__init__()
            if train_or_test == 'train':
                if os.path.exists("../dict_train2017.json"):
                    print("\nLoading training data from the
                    JSON file")
                    dict_train2017 = json.load( open(
                    "../dict_train2017.json" ) )
                    self.label_list = []
                    self.bbox_list = []
                    self.image_r_list = []
                    self.image_g_list = []
                    self.image_b_list = []
                    for img_id, img_content in
                    dict_train2017.items():

                        self.label_list.append
                        (img_content['class_id'])
                        self.bbox_list.append
                        (img_content['bbox'])
                        self.image_r_list.append
                        (img_content['image_r'])
                        self.image_g_list.append
                        (img_content['image_g'])
                        self.image_b_list.append
                        (img_content['image_b'])
                    self.resized_width =
                    dl_studio.image_size[0]
                    self.resized_height =
                    dl_studio.image_size[1]
                    self.transform = transform
                    self.class_labels = dl_studio.class_labels

                else:
                    print("""\n\n\nLooks like this is the
```

```
first time you will be loading in """
      """the training data for this
      script.\nFirst time loading could
      take """
      """2-3 hours or so to download COCO
      images.\nAny subsequent attempts
      will only take """
      """a few minutes.\n\n\n""")
self.class_list = dl_studio.class_labels

root_path_train = '../Train/'

coco_json_path_train2017 =
'../instances_train2017.json'

img_per_class_train2017 = 1250

self.resized_width =
dl_studio.image_size[0]
self.resized_height =
dl_studio.image_size[1]
resize_size = (self.resized_width,
self.resized_height)

# create training image folders, download
images and save data to a JSON file
dict_train2017 =
download_img(root_path_train,
coco_json_path_train2017, self.class_list,
img_per_class_train2017, resize_size)
print('\nSaving training data to the JSON
file\n')
json.dump(dict_train2017, open(
"../dict_train2017.json", 'w' ), indent =
4 )

self.label_list = []
self.bbox_list = []
self.image_r_list = []
self.image_g_list = []
self.image_b_list = []
for img_id, img_content in
dict_train2017.items():

    self.label_list.append(
    img_content['class_id'])
    self.bbox_list.append(
    img_content['bbox'])
    self.image_r_list.append(
    img_content['image_r'])
    self.image_g_list.append(
```

```
            img_content['image_g'])
            self.image_b_list.append(
            img_content['image_b'])
        self.transform = transform
        self.class_labels = dl_studio.class_labels

else: # train_or_test == 'test'
    if os.path.exists("../dict_val2017.json"):
        print("\nLoading validation data from
        the JSON file")
        dict_val2017 = json.load(
        open( "../dict_val2017.json" ) )
        self.label_list = []
        self.bbox_list = []
        self.image_r_list = []
        self.image_g_list = []
        self.image_b_list = []
        for img_id, img_content in
        dict_val2017.items():
            self.label_list.append
            (img_content['class_id'])
            self.bbox_list.append
            (img_content['bbox'])
            self.image_r_list.append
            (img_content['image_r'])
            self.image_g_list.append
            (img_content['image_g'])
            self.image_b_list.append
            (img_content['image_b'])
        self.resized_width =
        dl_studio.image_size[0]
        self.resized_height =
        dl_studio.image_size[1]
        self.transform = transform
        self.class_labels = dl_studio.class_labels
    else:

        print("""\n\n\nLooks like this is the
        first time you will be loading in """
              """the validation data for this
              script.\nFirst time loading could
              take """
              """10-20 minutes or so to download
              COCO images.\nAny subsequent
              attempts will only take """
              """a few minutes.\n\n\n""")
        self.class_list = dl_studio.class_labels

        root_path_val = '../Val/'

        coco_json_path_val2017 =
```

```
            '../instances_val2017.json'

            img_per_class_val2017 = 125

            self.resized_width =
            dl_studio.image_size[0]
            self.resized_height =
            dl_studio.image_size[1]
            resize_size = (self.resized_width,
            self.resized_height)

            # create testing image folders, download
            images and save data to a JSON file
            dict_val2017 = download_img(root_path_val,
            coco_json_path_val2017, self.class_list,
            img_per_class_val2017, resize_size)
            print('\nSaving validation data to the
            JSON file\n')
            json.dump(dict_val2017, open(
            "../dict_val2017.json", 'w' ),
            indent = 4 )

            self.label_list = []
            self.bbox_list = []
            self.image_r_list = []
            self.image_g_list = []
            self.image_b_list = []
            for img_id, img_content in
            dict_val2017.items():
                self.label_list.append
                (img_content['class_id'])
                self.bbox_list.append
                (img_content['bbox'])
                self.image_r_list.append
                (img_content['image_r'])
                self.image_g_list.append
                (img_content['image_g'])
                self.image_b_list.append
                (img_content['image_b'])
            self.class_labels = dl_studio.
            class_labels
            self.transform = transform

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

def __getitem__(self, idx):
    r = np.array(self.image_r_list[idx])
    g = np.array(self.image_g_list[idx])
    b = np.array(self.image_b_list[idx])
```

```
        im_tensor = torch.zeros(3,self.resized_height,
        self.resized_width, dtype=torch.float)
        im_tensor[0,:,:] = torch.from_numpy(r)
        im_tensor[1,:,:] = torch.from_numpy(g)
        im_tensor[2,:,:] = torch.from_numpy(b)

        bb_tensor = torch.tensor(self.bbox_list[idx],
        dtype=torch.float)
        sample = {'image' : im_tensor,
                  'bbox' : bb_tensor,
                  'label' : self.label_list[idx]}
        return sample

def load_COCO5Dataset_training_set(self,
dataserver_train):
    self.train_dataloader =
    torch.utils.data.DataLoader(dataserver_train,
    batch_size=self.dl_studio.batch_size,shuffle=True,
    num_workers=4)

class MySkipBlock(nn.Module):
    def __init__(self, in_ch, out_ch,
    downsample=False, skip_connections=True):
        super().__init__()
        self.downsample = downsample
        self.skip_connections = skip_connections
        self.in_ch = in_ch
        self.out_ch = out_ch
        self.convo1 = nn.Conv2d(in_ch, out_ch, 3,
        stride=1, padding=1)
        self.convo2 = nn.Conv2d(out_ch, out_ch, 3,
        stride=1, padding=1)
        norm_layer1 = nn.BatchNorm2d
        norm_layer2 = nn.BatchNorm2d
        self.bn1 = norm_layer1(out_ch)
        self.bn2 = norm_layer2(out_ch)
        if downsample:
            self.downsampler = nn.Conv2d(in_ch, out_ch,
            1, stride=2)
    def forward(self, x):
        identity = x
        out = self.convo1(x)
        out = self.bn1(out)
        out = torch.nn.functional.relu(out)
        if self.skip_connections:
            if self.in_ch == self.out_ch:
                out += identity

            else:
                out[:,:self.in_ch,:,:] += identity
                out[:,self.in_ch:,:,:] += identity
```

```
        out_1 = out.clone()

        out = self.convo2(out)

        out = self.bn2(out)
        out = torch.nn.functional.relu(out)
        if self.skip_connections:
            out += out_1
            if self.in_ch == self.out_ch:
                out += identity

            else:
                out[:,:self.in_ch,:,:] += identity
                out[:,self.in_ch:,:,:] += identity

        if self.downsample:
            out = self.downsampler(out)
            identity = self.downsampler(identity)
        return out


class MyLOADnet2(nn.Module):
    def __init__(self, skip_connections=True, depth=8):
        super().__init__()
        self.skip_connections = skip_connections
        if depth not in [8,10,12,14,16]:
            sys.exit("LOADnet2 has only been tested for
            'depth' values 8, 10, 12, 14, and 16")
        self.depth = depth // 2
        self.conv = nn.Conv2d(3, 64, 3, padding=1)
        self.bn1  = nn.BatchNorm2d(64)
        self.bn2  = nn.BatchNorm2d(128)
        self.skip64_arr = nn.ModuleList()
        for i in range(self.depth):
            self.skip64_arr.
            append(MyDLStudio.DetectAndLocalize.
            MySkipBlock(64, 64,


                                                skip_connections=
                                                  skip_connections))
        self.skip64ds = MyDLStudio.DetectAndLocalize.MySkipBlock
        (64, 64,
                                    downsample=True,
                                    skip_connections=
                                    skip_connections)
        self.skip64to128 = MyDLStudio.DetectAndLocalize.
        MySkipBlock(64, 128,
                                                    skip_connections=
                                                    skip_connections )
        self.skip128_arr = nn.ModuleList()
```

```
for i in range(self.depth):
    self.skip128_arr.append
    (MyDLStudio.DetectAndLocalize.
    MySkipBlock(128, 128,skip_connections=
              skip_connections))
self.skip128ds = MyDLStudio.
DetectAndLocalize.MySkipBlock(128,128,
                              downsample=True,
                              skip_connections=
                              skip_connections)
self.fc1 =  nn.Linear(8 * 8 * 128, 1000)
self.fc2 =  nn.Linear(1000, 5)

##  for regression
self.conv_seqn_init = nn.Sequential(
    nn.Conv2d(in_channels=64, out_channels=64,
    kernel_size=3, padding=1),
    nn.ReLU(inplace=True),
    nn.Conv2d(in_channels=64, out_channels=64,
    kernel_size=3, padding=1),
    nn.ReLU(inplace=True),
    nn.Conv2d(in_channels=64, out_channels=64,
    kernel_size=3, padding=1),
    nn.ReLU(inplace=True),
    nn.Conv2d(in_channels=64, out_channels=64,
    kernel_size=3, padding=1),
    nn.ReLU(inplace=True),
    nn.Conv2d(in_channels=64, out_channels=64,
    kernel_size=3, padding=1),
    nn.ReLU(inplace=True),
    nn.Conv2d(in_channels=64, out_channels=64,
    kernel_size=3, padding=1),
    nn.ReLU(inplace=True)
)

self.conv_seqn = nn.Sequential(
    nn.Conv2d(in_channels=64, out_channels=64,
    kernel_size=3, padding=1),
    nn.BatchNorm2d(64),
    nn.ReLU(inplace=True),
    nn.Conv2d(in_channels=64, out_channels=64,
    kernel_size=3, padding=1),
    nn.ReLU(inplace=True)
)
self.fc_seqn = nn.Sequential(
    nn.Linear(32 * 32 * 64, 1024),
    nn.ReLU(inplace=True),
    nn.Linear(1024, 512),
    nn.ReLU(inplace=True),
    nn.Linear(512, 4)
)
```

```
    regression_list = ['conv_seqn_init.weight',
    'conv_seqn_init.bias',\
                       'conv_seqn.weight',
                       'conv_seqn.bias', \
                           'fc_seqn.weight',
                           'fc_seqn.bias']
    regression_params_list_tuples = list(
    filter(lambda kv: kv[0] in regression_list,
    self.named_parameters() ) )
    classifier_params_list_tuples = list(
    filter(lambda kv: kv[0] not in regression_list,
    self.named_parameters() ) )

    self.regression_params = [ param[1] for param in
    regression_params_list_tuples  ]
    self.classifier_params = [ param[1] for param in
    classifier_params_list_tuples  ]

def forward(self, x):
    x = nn.MaxPool2d(2,2)(torch.nn.functional.relu(
    self.conv(x)))


    ## The labeling section:
    x1 = x.clone()
    for i,skip64 in
    enumerate(self.skip64_arr[:self.depth//4]):
        x1 = skip64(x1)
    x1 = self.skip64ds(x1)
    for i,skip64 in
    enumerate(self.skip64_arr[self.depth//4:]):
        x1 = skip64(x1)
    x1 = self.bn1(x1)
    x1 = self.skip64to128(x1)
    for i,skip128 in
    enumerate(self.skip128_arr[:self.depth//4]):
        x1 = skip128(x1)
    x1 = self.bn2(x1)

    x1 = self.skip128ds(x1)

    for i,skip128 in
    enumerate(self.skip128_arr[self.depth//4:]):
        x1 = skip128(x1)
    x1 = x1.view(-1, 8 * 8 * 128  )
    x1 = torch.nn.functional.relu(self.fc1(x1))
    x1 = self.fc2(x1)
```

```
        ## The Bounding Box regression:
        x_regre = x.clone()
        x2 = self.conv_seqn_init(x_regre)
        x2 = self.conv_seqn(x2)
        x2 = self.conv_seqn(x2)
        x2 = self.conv_seqn(x2)
        # flatten
        x2 = x2.view(x.size(0), -1)
        x2 = self.fc_seqn(x2)
        return x1,x2



def run_code_for_training_with_CrossEntropy_and_MSE_Losse
s(self, net):
    print("\nStart Training\n")
    loss_record_labeling = []
    loss_record_regression = []
    if net.skip_connections == True:
        filename_for_out1 = "training_loss_" +
        str(self.dl_studio.epochs) +
        "_epochs_with_skip_connections_label.txt"
        filename_for_out2 = "training_loss_" +
        str(self.dl_studio.epochs) +
        "_epochs_with_skip_connections_regres.txt"

    else:
        filename_for_out1 = "training_loss_" +
        str(self.dl_studio.epochs) +
        "_epochs_without_skip_connections_label.txt"
        filename_for_out2 = "training_loss_" +
        str(self.dl_studio.epochs) +
        "_epochs_without_skip_connections_regres.txt"
    path_to_save = './Train Results/'
    if not os.path.exists(path_to_save):
        os.makedirs(path_to_save)
    FILE1 = open(path_to_save + filename_for_out1, 'w')
    FILE2 = open(path_to_save + filename_for_out2, 'w')
    net = copy.deepcopy(net)
    net = net.to(self.dl_studio.device)
    criterion1 = nn.CrossEntropyLoss()
    criterion2 = nn.MSELoss()
    optimizer = optim.SGD([
                          {'params':
                          net.classifier_params, 'lr':
                          2 * 1e-5}
                        ],
                        lr=self.dl_studio.learning_rate,
                        momentum=self.dl_studio.momentum)
    for epoch in range(self.dl_studio.epochs):
        running_loss_labeling = 0.0
```

```
running_loss_regression = 0.0
for i, data in enumerate(self.train_dataloader):

    inputs, bbox_gt, labels = data['image'],
    data['bbox'], data['label']

    if self.dl_studio.debug_train and i % 500 ==
    499:
        print("\n\n[epoch=%d iter=%d:] Ground
        Truth:     " % (epoch+1, i+1) +
        ' '.join('%15s' %
        self.dataserver_train.class_labels
        [labels[j].item()] for j in
        range(self.dl_studio.batch_size)))
    inputs = inputs.to(self.dl_studio.device)
    labels = labels.to(self.dl_studio.device)
    bbox_gt = bbox_gt.to(self.dl_studio.device)
    optimizer.zero_grad()
    if self.debug:
        self.dl_studio.display_tensor_as_image(

          torchvision.utils.make_grid
          (inputs.cpu(), nrow=4,
          normalize=True, padding=2,
          pad_value=10))
    outputs = net(inputs)
    outputs_label = outputs[0]
    bbox_pred = outputs[1]
    if self.dl_studio.debug_train and
    i % 500 == 499:
        inputs_copy = inputs.detach().clone()
        inputs_copy = inputs_copy.cpu()
        bbox_pc = bbox_pred.detach().clone()
        bbox_pc[bbox_pc<0] = 0
        bbox_pc[bbox_pc>63] = 63
        bbox_pc[torch.isnan(bbox_pc)] =
        0
        _, predicted =
        torch.max(outputs_label.data, 1)
        print("[epoch=%d iter=%d:] Predicted
        Labels: " % (epoch+1, i+1) +
         ' '.join('%15s' %
         self.dataserver_train.class_labels
         [predicted[j].item()]

                          for j in
                          range(len(labels))))
        for idx in range(len(labels)):
            i1 = int(bbox_gt[idx][1])
            i2 = int(bbox_gt[idx][3])
            j1 = int(bbox_gt[idx][0])
```

```
            j2 = int(bbox_gt[idx][2])
            k1 = int(bbox_pc[idx][1])
            k2 = int(bbox_pc[idx][3])
            l1 = int(bbox_pc[idx][0])
            l2 = int(bbox_pc[idx][2])
            print("
            gt_bb:  [%d,%d,%d,%d]"%(j1,i1,j2,i2))
            print("
            pred_bb:
            [%d,%d,%d,%d]"%(l1,k1,l2,k2))


            # circle the ground truth in red

            inputs_copy[idx,0,i1:i2+1,j1] = 255
            inputs_copy[idx,0,i1:i2+1,j2] = 255
            inputs_copy[idx,0,i1,j1:j2+1] = 255
            inputs_copy[idx,0,i2,j1:j2+1] = 255
            inputs_copy[idx,1,i1:i2+1,j1] = 0
            inputs_copy[idx,1,i1:i2+1,j2] = 0
            inputs_copy[idx,1,i1,j1:j2+1] = 0
            inputs_copy[idx,1,i2,j1:j2+1] = 0
            inputs_copy[idx,2,i1:i2+1,j1] = 0
            inputs_copy[idx,2,i1:i2+1,j2] = 0
            inputs_copy[idx,2,i1,j1:j2+1] = 0
            inputs_copy[idx,2,i2,j1:j2+1] = 0

            # circle the predicted bbox in blue

            inputs_copy[idx,2,k1:k2+1,l1] = 255

            inputs_copy[idx,2,k1:k2+1,l2] = 255
            inputs_copy[idx,2,k1,l1:l2+1] = 255
            inputs_copy[idx,2,k2,l1:l2+1] = 255
            inputs_copy[idx,0,k1:k2+1,l1] = 0

            inputs_copy[idx,0,k1:k2+1,l2] = 0
            inputs_copy[idx,0,k1,l1:l2+1] = 0
            inputs_copy[idx,0,k2,l1:l2+1] = 0

            inputs_copy[idx,1,k1:k2+1,l1] = 0

            inputs_copy[idx,1,k1:k2+1,l2] = 0
            inputs_copy[idx,1,k1,l1:l2+1] = 0
            inputs_copy[idx,1,k2,l1:l2+1] = 0


    loss_labeling = criterion1(outputs_label,
    labels)
    loss_labeling.backward(retain_graph=True)
```

```
            loss_regression = criterion2(bbox_pred,
            bbox_gt)
            loss_regression.backward()
            optimizer.step()
            running_loss_labeling += loss_labeling.item()

            running_loss_regression +=
            loss_regression.item()
            if i % 500 == 499:
                avg_loss_labeling = running_loss_labeling
                / float(500)
                avg_loss_regression =
                running_loss_regression / float(500)

                loss_record_labeling.append
                (avg_loss_labeling)
                loss_record_regression.append
                (avg_loss_regression)
                print("\n[epoch:%d, iteration:%5d]
                loss_labeling: %.3f  loss_regression:
                %.3f  " % (epoch + 1, i + 1,
                avg_loss_labeling, avg_loss_regression))
                FILE1.write("%.3f\n" % avg_loss_labeling)
                FILE1.flush()
                FILE2.write("%.3f\n" %
                avg_loss_regression)
                FILE2.flush()
                running_loss_labeling = 0.0
                running_loss_regression = 0.0
            if self.dl_studio.debug_train and i%500==499:

                if net.skip_connections == True:
                    self.dl_studio.display_tensor_as_imag
                    e(
                      torchvision.utils.make_grid
                      (inputs_copy, normalize=True),

                     "Training_Results_With_Skip
                     _Connections_[Epoch=%d,_Iter=%d]" %
                     (epoch+1, i+1))
                else:
                     self.dl_studio.display_tensor_
                     as_image(
                      torchvision.utils.make_grid
                      (inputs_copy, normalize=True),
                     "Training_Results_Without_
                     Skip_Connections_[Epoch=%d,
                     _Iter=%d]" %  (epoch+1, i+1))

    print("\nFinished Training\n")
    self.save_model(net)
```

```
            return loss_record_labeling, loss_record_regression


        def save_model(self, model):
            torch.save(model.state_dict(),
            self.dl_studio.path_saved_model)


def get_image(img_url, img_class, path_img_class, cat_id, anns,
resize_size):
    if len(img_url) <= 1:
        print("Invalid URL.")
        return {}
    try:
        img_resp = requests.get(img_url, timeout = 1)
    except ConnectionError:
        print("Connection Error.")
        return {}
    except ReadTimeout:
        print("Read timeout.")
        return {}
    except TooManyRedirects:
        print("Too may redirects.")
        return {}
    except MissingSchema:
        print("Missing schema.")
        return {}
    except InvalidURL:
        print("Invalid URL.")
        return {}
    if not 'content-type' in img_resp.headers:
        print("Missing content.")
        return {}
    if not 'image' in img_resp.headers['content-type']:
        print("The URL doesn't have any image.")
        return {}

    img_name = img_url.split('/')[-1]
    img_name = img_name.split("?")[0]
    img_name_no_jpg = img_name.split('.')[0]

    if (len(img_name) <= 1):
        print("Missing image name.")
        return {}

    output_dict = {}
    img_dict = {}

    # find the class_id
    if img_class == 'cat':
        class_id = 0
```

```
elif img_class == 'train':
    class_id = 1
elif img_class == 'elephant':
    class_id = 2
elif img_class == 'airplane':
    class_id = 3
elif img_class == 'giraffe':
    class_id = 4
img_dict['class_id'] = class_id



# check if the dominant object in the image is our target
(i.e., equal to cat_id)
cat_id_list = []
bbox_value_list = []
bbox_area_list = []
for ann in anns:
    cat_id_list.append(ann['category_id'])
    bbox_value_list.append(ann['bbox'])
    bbox_area_list.append(ann['bbox'][2] * ann['bbox'][3])

dominant_ann_id = np.argmax(bbox_area_list)

if cat_id_list[dominant_ann_id] != cat_id:
    return {}

img_path = os.path.join(path_img_class, img_name)
with open(img_path, 'wb') as img_f:
    img_f.write(img_resp.content)

# save the original bbox
ori_bbox = np.zeros(4)
ori_bbox[0] = bbox_value_list[dominant_ann_id][0]
ori_bbox[1] = bbox_value_list[dominant_ann_id][1]
ori_bbox[2] = ori_bbox[0] +
bbox_value_list[dominant_ann_id][2]
ori_bbox[3] = ori_bbox[1] +
bbox_value_list[dominant_ann_id][3]
img_dict['ori_bbox'] = ori_bbox.tolist()

# find the original image size
im = Image.open(img_path)
width, height = im.size
img_dict['ori_size'] = [width, height]


# check if the dominant bbox is too small
dominant_bbox_size = bbox_value_list[dominant_ann_id][2] *
bbox_value_list[dominant_ann_id][3]
if ( dominant_bbox_size / (width * height) ) < 0.15:
```

```
            img_f.close()
            im.close()
            os.remove(img_path)
            return {}

        # save the adjusted bbox
        resized_width, resized_height = resize_size
        bbox = np.zeros(4)
        bbox[0] = resized_width * (ori_bbox[0] + 1) / width - 1
        bbox[1] = resized_height * (ori_bbox[1] + 1) / height - 1
        bbox[2] = resized_width * (ori_bbox[2] + 1)/ width - 1
        bbox[3] = resized_height * (ori_bbox[3] + 1)/ height - 1


        # make sure the adjusted bbox does not exceed the border
        of the resized image
        bbox[0] = min(max(bbox[0], 0), resized_width - 1)
        bbox[1] = min(max(bbox[1], 0), resized_height - 1)
        bbox[2] = min(max(bbox[2], 0), resized_width - 1)
        bbox[3] = min(max(bbox[3], 0), resized_height - 1)

        img_dict['bbox'] = bbox.tolist()

        if im.mode != "RGB":
            im = im.convert(mode="RGB")

        # save the resized image (64x64)
        im_resized = im.resize((resized_width, resized_height),
        Image.BOX)
        np_im_resized = np.array(im_resized)
        img_dict['image_r'] = np_im_resized[:,:,0].tolist()
        img_dict['image_g'] = np_im_resized[:,:,1].tolist()
        img_dict['image_b'] = np_im_resized[:,:,2].tolist()

        # overwrite original image with downsampled image
        im_resized.save(img_path, quality = 95)

        output_dict[img_name_no_jpg] = img_dict
        return output_dict


def download_img(root_path, coco_json_path, class_list,
images_per_class, resize_size):
    # create image folders if they were not existed
    for img_class in class_list:
        path_img_class = os.path.join(root_path, img_class)
        if not os.path.exists(path_img_class):
            os.makedirs(path_img_class)

    # initialize COCO API and get image IDs
    coco = COCO(coco_json_path)
```

```
    dict_to_return = {}

    for img_class in class_list:
        path_img_class = os.path.join(root_path, img_class)
        cat_id = coco.getCatIds(catNms = img_class);
        imgs_ids = coco.getImgIds(catIds = cat_id);
        imgs = coco.loadImgs(imgs_ids)
        num_img_downloaded = 0
        current_id = 0
        num_img_to_download = min(images_per_class, len(imgs_ids))

        print('\nGoal: Download %d images of %s\n' %
        (num_img_to_download, img_class))
        while( (num_img_downloaded < num_img_to_download) and
        (current_id <  len(imgs_ids)) ):
            annId = coco.getAnnIds(imgIds=imgs_ids[current_id])
            anns = coco.loadAnns(annId)
            img_url = imgs[current_id]['coco_url']
            output_dict = get_image(img_url, img_class,
            path_img_class, cat_id[0], anns, resize_size)
            if (len(output_dict) > 0):
                num_img_downloaded += 1
                dict_to_return.update(output_dict)
                if num_img_downloaded % 10 == 0:
                    print('Downloaded %d images' %
                    num_img_downloaded)
            current_id += 1
        print('Finish downloading')
        print('\nResult: Downloaded a total of %d images of %s\n'
        % (num_img_downloaded, img_class))
    print('\nFinish downloading all the images\n')
    return dict_to_return

def plot_loss(loss_val_list, loss_property):
    line_wid = 5
    font_size = 25
    font = {'size': font_size}
    plt.rc('font', **font)
    plt.figure(figsize = (12,9))
    for loss in loss_val_list:
        plt.plot(loss, linewidth = line_wid)
    plt.xlabel("iteration")
    plt.ylabel("loss")
    if loss_property == 'labeling':
        title = 'Training Loss (Dectection)'
    else:
        title = 'Training Loss (Localization)'
    plt.title(title)
    plt.grid()
    plt.legend(['Without Skip Connections', 'With Skip
    Connections'])
```

```
    path_saved_image = './Train Results/'
    if not os.path.exists(path_saved_image):
        os.makedirs(path_saved_image)

    img_title = path_saved_image + title.lower().replace('(',
    "").replace(')',"").replace(' ','_')
    plt.savefig(img_title + '.png', bbox_inches = "tight")
    im = Image.open(img_title + '.png')
    if im.mode in ("RGBA", "P"):
        im = im.convert("RGB")

    im.save(img_title + '.jpg','JPEG', quality = 95)
    os.remove(img_title + '.png')
    plt.show()



#_______________________________  Main program starts here
_______________________________
if __name__ == '__main__':


    #_______________________________  Part 1: Download COCO images
    and extract useful annotations
    _______________________________
    # Part 1 will be done in the beginning of Part 2. See the
    class COCO5Dataset.
    # The program will download COCO images of 5 classes ('cat',
    'train', 'elephant', 'airplane', 'giraffe') using
    instances_train2017.json.

    # Each class has 1250 images.
    # Each image will be resized to 64 x 64.
    # Each image's pixel values and its annotations will be
    extracted and saved into the JSON file dict_train2017.json.


    #_______________________________  Part 2: Run my designed
    network (without skip connections)
    _______________________________

    loss_labeling_list = []
    loss_regression_list = []

    my_dls = MyDLStudio(
                      image_size = [64,64],
                      # resized image width, resized image height
                      path_saved_model = "./net1.pth",
                      momentum = 0.95,
                      learning_rate = 1 * 1e-5,
```

```
                epochs = 10,
                batch_size = 8,
                classes = ['cat', 'train', 'elephant',
                'airplane', 'giraffe'],
                debug_train = 1,
                debug_test = 1,
                use_gpu = True,
            )


my_detector = MyDLStudio.DetectAndLocalize(
dl_studio = my_dls )
my_dataserver_train =
MyDLStudio.DetectAndLocalize.COCO5Dataset(
                                    train_or_test = 'train',
                                    dl_studio = my_dls,
                                    )

my_detector.dataserver_train = my_dataserver_train

my_detector.load_COCO5Dataset_training_set
(my_dataserver_train)


# My designed network (without skip connections)
my_model = my_detector.MyLOADnet2
(skip_connections=False, depth=8)

number_of_learnable_params = sum(p.numel()
for p in my_model.parameters() if p.requires_grad)
print("\n\nThe number of learnable parameters in the my_model:
%d" % number_of_learnable_params)

num_layers = len(list(my_model.parameters()))
print("\n\nThe number of layers in the model: %d\n\n" %
num_layers)

loss_labeling, loss_regression =
my_detector.run_code_for_training_with_CrossEntropy_and_MSE_
Losses(my_model)
loss_labeling_list.append(loss_labeling)
loss_regression_list.append(loss_regression)


#_______________________________  Part 3: Run my designed
network (with skip connections)
--------------------------------
my_dls_2 = MyDLStudio(
                image_size = [64,64],
                # resized image width, resized image height
                path_saved_model = "./net2.pth",
```

```
                momentum = 0.95,
                learning_rate = 1 * 1e-5,
                epochs = 10,
                batch_size = 8,
                classes = ['cat', 'train', 'elephant',
                'airplane', 'giraffe'],
                debug_train = 1,
                debug_test = 1,
                use_gpu = True,
            )


my_detector_2 = MyDLStudio.DetectAndLocalize( dl_studio =
my_dls_2 )
my_dataserver_train_2 =
MyDLStudio.DetectAndLocalize.COCO5Dataset(
                                    train_or_test = 'train',
                                    dl_studio = my_dls_2,

                                    )

my_detector_2.dataserver_train = my_dataserver_train_2

my_detector_2.load_COCO5Dataset_training_set
(my_dataserver_train_2)

# My designed network (with skip connections)
my_model_2 = my_detector_2.MyLOADnet2(skip_connections=True,
depth=8)

number_of_learnable_params = sum(p.numel() for p in
my_model_2.parameters() if p.requires_grad)
print("\n\nThe number of learnable parameters in the
my_model_2: %d" % number_of_learnable_params)

num_layers = len(list(my_model_2.parameters()))
print("\n\nThe number of layers in the model: %d\n\n" %
num_layers)

loss_labeling, loss_regression =
my_detector_2.run_code_for_training_with_CrossEntropy_
and_MSE_Losses(my_model_2)
loss_labeling_list.append(loss_labeling)
loss_regression_list.append(loss_regression)

# plot loss values
plot_loss(loss_labeling_list, 'labeling')
plot_loss(loss_regression_list, 'regression')
```

hw05_validation.py

```
'''
The designed structure of the code is similar to object_detection_and_localization.py (written by
↪  Prof. Kak).

However, the modifications are all written by myself.

The program will download COCO images of 5 classes ('cat', 'train', 'elephant', 'airplane', 'giraffe')
↪  using instances_val2017.json.

Each class has 125 images, and each image will be resized to 64 x 64.

Each image's pixel values and its annotations will be extracted and saved to the JSON file
↪  dict_val2017.json.

The program will load the weights of two neural networks form the current directory: net1 and net2.

net1 is designed without skip connections, whereas net2 uses skip connections.

The validation results (loss values, normalized confusion matrices and images with bboxes) will be
↪  saved to the directory 'Val Results'

'''
import os, sys
import random
import json
import numpy as np
import torch
import torchvision
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.utils.data import DataLoader, Dataset
import torchvision.transforms as tvt
from PIL import Image
import requests
from requests.exceptions import ConnectionError,ReadTimeout,TooManyRedirects,MissingSchema, InvalidURL
from pycocotools.coco import COCO
import copy
import pickle
import gzip
import matplotlib.pyplot as plt
import seaborn as sns

# ensure reproducibility
seed = 0
```

```
random.seed(seed)
torch.manual_seed(seed)
torch.cuda.manual_seed(seed)
np.random.seed(seed)
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmarks = False
os.environ['PYTHONHASHSEED'] = str(seed)

class MyDLStudio(object):
    def __init__(self, *args, **kwargs ):
        if args:
            raise ValueError('''Only keyword arguments are allowed''')
        learning_rate = epochs = batch_size = momentum = None
        image_size = dataroot =  path_saved_model = classes = use_gpu = None
        if 'dataroot' in kwargs                       :   dataroot = kwargs.pop('dataroot')
        if 'learning_rate' in kwargs                  :   learning_rate = kwargs.pop('learning_rate')
        if 'momentum' in kwargs                       :   momentum = kwargs.pop('momentum')
        if 'epochs' in kwargs                         :   epochs = kwargs.pop('epochs')
        if 'batch_size' in kwargs                     :   batch_size = kwargs.pop('batch_size')
        if 'path_saved_model' in kwargs               :   path_saved_model =
        ↪  kwargs.pop('path_saved_model')
        if 'classes' in kwargs                        :   classes = kwargs.pop('classes')
        if 'use_gpu' in kwargs                        :   use_gpu = kwargs.pop('use_gpu')
        if 'image_size' in kwargs                     :   image_size = kwargs.pop('image_size')
        if 'use_gpu' in kwargs                        :   use_gpu = kwargs.pop('use_gpu')
        if 'debug_train' in kwargs                    :   debug_train = kwargs.pop('debug_train')
        if 'debug_test' in kwargs                     :   debug_test = kwargs.pop('debug_test')
        if dataroot:
            self.dataroot = dataroot
        if image_size:
            self.image_size = image_size
        if  path_saved_model:
            self.path_saved_model = path_saved_model
        if classes:
            self.class_labels = classes
            self.num_classes = len(classes)
        if learning_rate:
            self.learning_rate = learning_rate
        else:
            self.learning_rate = 1e-6
        if momentum:
            self.momentum = momentum
        if epochs:
            self.epochs = epochs
        if batch_size:
            self.batch_size = batch_size
        if use_gpu is not None:
            self.use_gpu = use_gpu
            if use_gpu is True:
                if torch.cuda.is_available():
                    self.device = torch.device("cuda:0")
```

```
                else:
                    raise Exception("No GPU on this machine")
            else:
                self.device = torch.device("cpu")
        if debug_train:
            self.debug_train = debug_train
        else:
            self.debug_train = 0
        if debug_test:
            self.debug_test = debug_test
        else:
            self.debug_test = 0

    def display_tensor_as_image(self, tensor, title=""):
        tensor_range = (torch.min(tensor).item(), torch.max(tensor).item())
        if tensor_range == (-1.0,1.0):
            ##  The tensors must be between 0.0 and 1.0 for the display:
            print("\n\n\nimage un-normalization called")
            tensor = tensor/2.0 + 0.5     # unnormalize

        font_size = 20
        font = {'size': font_size}
        plt.rc('font', **font)
        plt.figure(figsize = (20,200))
        img_title = title.replace('_', ' ')
        plt.title(img_title)

        ###  The call to plt.imshow() shown below needs a numpy array. We must also
        ###  transpose the array so that the number of channels (the same thing as the
        ###  number of color planes) is in the last element.  For a tensor, it would be in
        ###  the first element.
        if tensor.shape[0] == 3 and len(tensor.shape) == 3:
#            plt.imshow( tensor.numpy().transpose(1,2,0) )
            plt.imshow( tensor.numpy().transpose(1,2,0) )
        ###  If the grayscale image was produced by calling torchvision.transform's
        ###  ".ToPILImage()", and the result converted to a tensor, the tensor shape will
        ###  again have three elements in it, however the first element that stands for
        ###  the number of channels will now be 1
        elif tensor.shape[0] == 1 and len(tensor.shape) == 3:
            tensor = tensor[0,:,:]
            plt.imshow( tensor.numpy(), cmap = 'gray' )
        ###  For any one color channel extracted from the tensor representation of a color
        ###  image, the shape of the tensor will be (W,H):
        elif len(tensor.shape) == 2:
            plt.imshow( tensor.numpy(), cmap = 'gray' )
        else:
            sys.exit("\n\n\nfrom 'display_tensor_as_image()': tensor for image is ill formed --
            ↪  aborting")

        path_saved_image = './Val Results/'
        if not os.path.exists(path_saved_image):
```

```
            os.makedirs(path_saved_image)

        img_title_save = path_saved_image + title.lower().replace('[', "").replace(']',
        ↪  "").replace(',', "")

        plt.savefig(img_title_save + '.png', bbox_inches = "tight")
        im = Image.open(img_title_save + '.png')
        if im.mode in ("RGBA", "P"):
            im = im.convert("RGB")
        im.save(img_title_save + '.jpg','JPEG', quality = 95)
        os.remove(img_title_save + '.png')

        plt.show()

    class DetectAndLocalize(nn.Module):
        def __init__(self, dl_studio, dataserver_train=None, dataserver_test=None,
        ↪  dataset_file_train=None, dataset_file_test=None):
            super().__init__()
            self.dl_studio = dl_studio
            self.dataserver_train = dataserver_train
            self.dataserver_test = dataserver_test
            self.debug = False

        class COCO5Dataset(torch.utils.data.Dataset):
            def __init__(self, dl_studio, train_or_test, transform=None):
                super().__init__()
                if train_or_test == 'train':
                    if os.path.exists("../dict_train2017.json"):
                        print("\nLoading training data from the JSON file")
                        dict_train2017 = json.load( open( "../dict_train2017.json" ) )
                        self.label_list = []
                        self.bbox_list = []
                        self.image_r_list = []
                        self.image_g_list = []
                        self.image_b_list = []
                        for img_id, img_content in dict_train2017.items():
                            self.label_list.append(img_content['class_id'])
                            self.bbox_list.append(img_content['bbox'])
                            self.image_r_list.append(img_content['image_r'])
                            self.image_g_list.append(img_content['image_g'])
                            self.image_b_list.append(img_content['image_b'])
                        self.resized_width = dl_studio.image_size[0]
                        self.resized_height = dl_studio.image_size[1]
                        self.transform = transform
                        self.class_labels = dl_studio.class_labels

                    else:
                        print("""\n\n\nLooks like this is the first time you will be loading in """
                              """the training data for this script.\nFirst time loading could take """
                              """2-3 hours or so to download COCO images.\nAny subsequent attempts
                              ↪  will only take """
```

```
                """a few minutes.\n\n\n""")
            self.class_list = dl_studio.class_labels

            root_path_train = '../Train/'

            coco_json_path_train2017 = '../instances_train2017.json'

            img_per_class_train2017 = 1250

            self.resized_width = dl_studio.image_size[0]
            self.resized_height = dl_studio.image_size[1]
            resize_size = (self.resized_width, self.resized_height)

            # create training image folders, download images and save data to a JSON file
            dict_train2017 = download_img(root_path_train, coco_json_path_train2017,
            ↪  self.class_list, img_per_class_train2017, resize_size)
            print('\nSaving training data to the JSON file\n')
            json.dump(dict_train2017, open( "../dict_train2017.json", 'w' ), indent = 4 )

            self.label_list = []
            self.bbox_list = []
            self.image_r_list = []
            self.image_g_list = []
            self.image_b_list = []
            for img_id, img_content in dict_train2017.items():
                self.label_list.append(img_content['class_id'])
                self.bbox_list.append(img_content['bbox'])
                self.image_r_list.append(img_content['image_r'])
                self.image_g_list.append(img_content['image_g'])
                self.image_b_list.append(img_content['image_b'])

            self.transform = transform
            self.class_labels = dl_studio.class_labels


    else: # train_or_test == 'test'
        if os.path.exists("../dict_val2017.json"):
            print("\nLoading validation data from the JSON file")
            dict_val2017 = json.load( open( "../dict_val2017.json" ) )
            self.label_list = []
            self.bbox_list = []
            self.image_r_list = []
            self.image_g_list = []
            self.image_b_list = []
            for img_id, img_content in dict_val2017.items():
                self.label_list.append(img_content['class_id'])
                self.bbox_list.append(img_content['bbox'])
                self.image_r_list.append(img_content['image_r'])
                self.image_g_list.append(img_content['image_g'])
                self.image_b_list.append(img_content['image_b'])
            self.resized_width = dl_studio.image_size[0]
```

```
            self.resized_height = dl_studio.image_size[1]
            self.transform = transform
            self.class_labels = dl_studio.class_labels
        else:

            print("""\n\n\nLooks like this is the first time you will be loading in """
                  """the validation data for this script.\nFirst time loading could take
                  ↪  """
                  """10-20 minutes or so to download COCO images.\nAny subsequent attempts
                  ↪  will only take """
                  """a few minutes.\n\n\n""")
            self.class_list = dl_studio.class_labels

            root_path_val = '../Val/'

            coco_json_path_val2017 = '../instances_val2017.json'

            img_per_class_val2017 = 125

            self.resized_width = dl_studio.image_size[0]
            self.resized_height = dl_studio.image_size[1]
            resize_size = (self.resized_width, self.resized_height)

            # create testing image folders, download images and save data to a JSON file
            dict_val2017 = download_img(root_path_val, coco_json_path_val2017,
            ↪  self.class_list, img_per_class_val2017, resize_size)
            print('\nSaving validation data to the JSON file\n')
            json.dump(dict_val2017, open( "../dict_val2017.json", 'w' ), indent = 4 )

            self.label_list = []
            self.bbox_list = []
            self.image_r_list = []
            self.image_g_list = []
            self.image_b_list = []
            for img_id, img_content in dict_val2017.items():
                self.label_list.append(img_content['class_id'])
                self.bbox_list.append(img_content['bbox'])
                self.image_r_list.append(img_content['image_r'])
                self.image_g_list.append(img_content['image_g'])
                self.image_b_list.append(img_content['image_b'])
            self.class_labels = dl_studio.class_labels
            self.transform = transform

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

    def __getitem__(self, idx):
        r = np.array(self.image_r_list[idx])
        g = np.array(self.image_g_list[idx])
        b = np.array(self.image_b_list[idx])
```

```
        im_tensor = torch.zeros(3,self.resized_height,self.resized_width, dtype=torch.float)
        im_tensor[0,:,:] = torch.from_numpy(r)
        im_tensor[1,:,:] = torch.from_numpy(g)
        im_tensor[2,:,:] = torch.from_numpy(b)

        bb_tensor = torch.tensor(self.bbox_list[idx], dtype=torch.float)
        sample = {'image' : im_tensor,
                  'bbox' : bb_tensor,
                  'label' : self.label_list[idx]}
        return sample

def load_COCO5Dataset_training_set(self, dataserver_train):
    self.train_dataloader = torch.utils.data.DataLoader(dataserver_train,
                        batch_size=self.dl_studio.batch_size,shuffle=True, num_workers=4)

def load_COCO5Dataset_validation_set(self, dataserver_test):
    self.test_dataloader = torch.utils.data.DataLoader(dataserver_test,
                        batch_size=self.dl_studio.batch_size,shuffle=True, num_workers=4)

class MySkipBlock(nn.Module):
    def __init__(self, in_ch, out_ch, downsample=False, skip_connections=True):
        super().__init__()
        self.downsample = downsample
        self.skip_connections = skip_connections
        self.in_ch = in_ch
        self.out_ch = out_ch
        self.convo1 = nn.Conv2d(in_ch, out_ch, 3, stride=1, padding=1)
        self.convo2 = nn.Conv2d(out_ch, out_ch, 3, stride=1, padding=1)
        norm_layer1 = nn.BatchNorm2d
        norm_layer2 = nn.BatchNorm2d
        self.bn1 = norm_layer1(out_ch)
        self.bn2 = norm_layer2(out_ch)
        if downsample:
            self.downsampler = nn.Conv2d(in_ch, out_ch, 1, stride=2)
    def forward(self, x):
        identity = x
        out = self.convo1(x)
        out = self.bn1(out)
        out = torch.nn.functional.relu(out)
        if self.skip_connections:
            if self.in_ch == self.out_ch:
                out += identity
            else:
                out[:,:self.in_ch,:,:] += identity
                out[:,self.in_ch:,:,:] += identity
        out_1 = out.clone()

        out = self.convo2(out)
        out = self.bn2(out)
        out = torch.nn.functional.relu(out)
        if self.skip_connections:
```

```
                out += out_1
                if self.in_ch == self.out_ch:
                    out += identity
                else:
                    out[:,:self.in_ch,:,:] += identity
                    out[:,self.in_ch:,:,:] += identity
            if self.downsample:
                out = self.downsampler(out)
                identity = self.downsampler(identity)
            return out

    class MyLOADnet2(nn.Module):
        def __init__(self, skip_connections=True, depth=8):
            super().__init__()
            self.skip_connections = skip_connections
            if depth not in [8,10,12,14,16]:
                sys.exit("LOADnet2 has only been tested for 'depth' values 8, 10, 12, 14, and 16")
            self.depth = depth // 2
            self.conv = nn.Conv2d(3, 64, 3, padding=1)
#               self.pool = nn.MaxPool2d(2, 2)
            self.bn1  = nn.BatchNorm2d(64)
            self.bn2  = nn.BatchNorm2d(128)
            self.skip64_arr = nn.ModuleList()
            for i in range(self.depth):
                self.skip64_arr.append(MyDLStudio.DetectAndLocalize.MySkipBlock(64, 64,
                                                    skip_connections=skip_connections))
            self.skip64ds = MyDLStudio.DetectAndLocalize.MySkipBlock(64, 64,
                                        downsample=True, skip_connections=skip_connections)
            self.skip64to128 = MyDLStudio.DetectAndLocalize.MySkipBlock(64, 128,
                                                      skip_connections=skip_connections )
            self.skip128_arr = nn.ModuleList()
            for i in range(self.depth):
                self.skip128_arr.append(MyDLStudio.DetectAndLocalize.MySkipBlock(128, 128,
                                                    skip_connections=skip_connections))
            self.skip128ds = MyDLStudio.DetectAndLocalize.MySkipBlock(128,128,
                                        downsample=True, skip_connections=skip_connections)
            self.fc1 =  nn.Linear(8 * 8 * 128, 1000)
            self.fc2 =  nn.Linear(1000, 5)

            ##  for regression
            self.conv_seqn_init = nn.Sequential(
                nn.Conv2d(in_channels=64, out_channels=64, kernel_size=3, padding=1),
                nn.ReLU(inplace=True),
                nn.Conv2d(in_channels=64, out_channels=64, kernel_size=3, padding=1),
                nn.ReLU(inplace=True),
                nn.Conv2d(in_channels=64, out_channels=64, kernel_size=3, padding=1),
                nn.ReLU(inplace=True),
                nn.Conv2d(in_channels=64, out_channels=64, kernel_size=3, padding=1),
                nn.ReLU(inplace=True),
                nn.Conv2d(in_channels=64, out_channels=64, kernel_size=3, padding=1),
                nn.ReLU(inplace=True),
```

```
            nn.Conv2d(in_channels=64, out_channels=64, kernel_size=3, padding=1),
            nn.ReLU(inplace=True)
        )

        self.conv_seqn = nn.Sequential(
            nn.Conv2d(in_channels=64, out_channels=64, kernel_size=3, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True),
            nn.Conv2d(in_channels=64, out_channels=64, kernel_size=3, padding=1),
            nn.ReLU(inplace=True)
        )
        self.fc_seqn = nn.Sequential(
            nn.Linear(32 * 32 * 64, 1024),
            nn.ReLU(inplace=True),
            nn.Linear(1024, 512),
            nn.ReLU(inplace=True),
            nn.Linear(512, 4)
        )


        regression_list = ['conv_seqn_init.weight', 'conv_seqn_init.bias',\
                           'conv_seqn.weight', 'conv_seqn.bias', \
                               'fc_seqn.weight', 'fc_seqn.bias']
        regression_params_list_tuples = list( filter(lambda kv: kv[0] in regression_list,
        ↪  self.named_parameters() ) )
        classifier_params_list_tuples = list( filter(lambda kv: kv[0] not in regression_list,
        ↪  self.named_parameters() ) )
        self.regression_params = [ param[1] for param in regression_params_list_tuples  ]
        self.classifier_params = [ param[1] for param in classifier_params_list_tuples  ]

    def forward(self, x):
        x = nn.MaxPool2d(2,2)(torch.nn.functional.relu(self.conv(x)))
        x_regre = x.clone()

        ## The labeling section:
        x1 = x.clone()
        for i,skip64 in enumerate(self.skip64_arr[:self.depth//4]):
            x1 = skip64(x1)
        x1 = self.skip64ds(x1)
        for i,skip64 in enumerate(self.skip64_arr[self.depth//4:]):
            x1 = skip64(x1)
        x1 = self.bn1(x1)
        x1 = self.skip64to128(x1)
        for i,skip128 in enumerate(self.skip128_arr[:self.depth//4]):
            x1 = skip128(x1)
        x1 = self.bn2(x1)

        x1 = self.skip128ds(x1)

        for i,skip128 in enumerate(self.skip128_arr[self.depth//4:]):
            x1 = skip128(x1)
```

```
            x1 = x1.view(-1, 8 * 8 * 128  )
            x1 = torch.nn.functional.relu(self.fc1(x1))
            x1 = self.fc2(x1)


            ## The Bounding Box regression:
            x2 = self.conv_seqn_init(x_regre)
            x2 = self.conv_seqn(x2)
            x2 = self.conv_seqn(x2)
            x2 = self.conv_seqn(x2)
            # flatten
            x2 = x2.view(x.size(0), -1)
            x2 = self.fc_seqn(x2)
            return x1,x2



    def save_model(self, model):
        torch.save(model.state_dict(), self.dl_studio.path_saved_model)


    def run_code_for_testing_detection_and_localization(self, net):
        print("\nStart Testing\n")
        criterion1 = nn.CrossEntropyLoss()
        criterion2 = nn.MSELoss()
        net.load_state_dict(torch.load(self.dl_studio.path_saved_model))
        correct = 0
        total = 0
        confusion_matrix = torch.zeros(len(self.dataserver_test.class_labels),
                                       len(self.dataserver_test.class_labels))
        class_correct = [0] * len(self.dataserver_test.class_labels)
        class_total = [0] * len(self.dataserver_test.class_labels)
        with torch.no_grad():
            running_loss_labeling = 0.0
            running_loss_regression = 0.0
            for i, data in enumerate(self.test_dataloader):
                images, bounding_box, labels = data['image'], data['bbox'], data['label']

                if self.dl_studio.debug_test and i % 50 == 0:
                    print("\n\n[i=%d:] Ground Truth:     " %i + ' '.join('%15s' %
self.dataserver_test.class_labels[labels[j]] for j in range(self.dl_studio.batch_size)))

                outputs = net(images)
                outputs_label = outputs[0]
                outputs_regression = outputs[1]
                outputs_regression[outputs_regression < 0] = 0
                outputs_regression[outputs_regression > 63] = 63
                outputs_regression[torch.isnan(outputs_regression)] = 0
                output_bb = outputs_regression.tolist()
                _, predicted = torch.max(outputs_label.data, 1)
                loss_labeling = criterion1(outputs_label, labels)
```

```
                loss_regression = criterion2(outputs_regression, bounding_box)
                running_loss_labeling += loss_labeling.item()
                running_loss_regression += loss_regression.item()

                predicted = predicted.tolist()
                labels = labels.tolist()
                if self.dl_studio.debug_test and i % 50 == 0:
                    print("[i=%d:] Predicted Labels: " %i + ' '.join('%15s' %
self.dataserver_test.class_labels[predicted[j]] for j in range(self.dl_studio.batch_size)))
                    for idx in range(self.dl_studio.batch_size):
                        i1 = int(bounding_box[idx][1])
                        i2 = int(bounding_box[idx][3])
                        j1 = int(bounding_box[idx][0])
                        j2 = int(bounding_box[idx][2])
                        k1 = int(output_bb[idx][1])
                        k2 = int(output_bb[idx][3])
                        l1 = int(output_bb[idx][0])
                        l2 = int(output_bb[idx][2])
                        print("                    gt_bb:  [%d,%d,%d,%d]"%(j1,i1,j2,i2))
                        print("                  pred_bb:  [%d,%d,%d,%d]"%(l1,k1,l2,k2))

                        # circle the ground truth in red
                        images[idx,0,i1:i2+1,j1] = 255
                        images[idx,0,i1:i2+1,j2] = 255
                        images[idx,0,i1,j1:j2+1] = 255
                        images[idx,0,i2,j1:j2+1] = 255
                        images[idx,1,i1:i2+1,j1] = 0
                        images[idx,1,i1:i2+1,j2] = 0
                        images[idx,1,i1,j1:j2+1] = 0
                        images[idx,1,i2,j1:j2+1] = 0
                        images[idx,2,i1:i2+1,j1] = 0
                        images[idx,2,i1:i2+1,j2] = 0
                        images[idx,2,i1,j1:j2+1] = 0
                        images[idx,2,i2,j1:j2+1] = 0

                        # circle the predicted bbox in blue
                        images[idx,2,k1:k2+1,l1] = 255
                        images[idx,2,k1:k2+1,l2] = 255
                        images[idx,2,k1,l1:l2+1] = 255
                        images[idx,2,k2,l1:l2+1] = 255
                        images[idx,0,k1:k2+1,l1] = 0
                        images[idx,0,k1:k2+1,l2] = 0
                        images[idx,0,k1,l1:l2+1] = 0
                        images[idx,0,k2,l1:l2+1] = 0
                        images[idx,1,k1:k2+1,l1] = 0
                        images[idx,1,k1:k2+1,l2] = 0
                        images[idx,1,k1,l1:l2+1] = 0
                        images[idx,1,k2,l1:l2+1] = 0
                    if net.skip_connections == True:
                        self.dl_studio.display_tensor_as_image(
                          torchvision.utils.make_grid(images, normalize=True),
```

```
                    "Validation_Results_With_Skip_Connections_[Iter=%d]" %  (i+1) )
                else:
                    self.dl_studio.display_tensor_as_image(
                     torchvision.utils.make_grid(images, normalize=True),
                    "Validation_Results_Without_Skip_Connections_[Iter=%d]" % (i+1) )
            for label,prediction in zip(labels,predicted):
                confusion_matrix[label][prediction] += 1
            total += len(labels)
            correct +=  [predicted[ele] == labels[ele] for ele in
            ↪  range(len(predicted))].count(True)
            comp = [predicted[ele] == labels[ele] for ele in range(len(predicted))]
            for j in range(len(labels)):
                label = labels[j]
                class_correct[label] += comp[j]
                class_total[label] += 1
    avg_loss_labeling = running_loss_labeling / float(total)
    avg_loss_regression = running_loss_regression / float(total)
    print("\n")
    for j in range(len(self.dataserver_test.class_labels)):
        print('Prediction accuracy for %5s : %2d %%' % (
      self.dataserver_test.class_labels[j], 100 * class_correct[j] / class_total[j]))
    print("\n\n\nOverall accuracy of the network on the test images: %d %%" %
                                                          (100 * correct / float(total)))
    print("\n\nDisplaying the normalized confusion matrix:\n")
    out_str = "                   "
    for j in range(len(self.dataserver_test.class_labels)):
                        out_str +=  "%15s" % self.dataserver_test.class_labels[j]
    print(out_str + "\n")
    confusion_mat = []
    for i,label in enumerate(self.dataserver_test.class_labels):
        out_percents = [100 * confusion_matrix[i,j] / float(class_total[i])
                        for j in range(len(self.dataserver_test.class_labels))]
        confusion_mat.append(out_percents)
        out_percents = ["%.2f" % item.item() for item in out_percents]
        out_str = "%15s:  " % self.dataserver_test.class_labels[i]
        for j in range(len(self.dataserver_test.class_labels)):
                                              out_str +=  "%15s" % out_percents[j]
        print(out_str)
    acc =  100 * correct / float(total)
    print("\nFinished Testing\n")
    return [acc, np.array(confusion_mat), avg_loss_labeling, avg_loss_regression]

def plot_confusion_mat(self, acc, confusion_mat, net):
    font_size = 30
    font = {'size': font_size}
    plt.rc('font', **font)
    plt.figure(figsize = (20,20))
    ax = plt.gca()

    sns.heatmap(confusion_mat, annot = True, ax = ax, fmt = '.2f', cmap = "YlGnBu");
```

```
            ax.set_xticklabels(
                ax.set_xticklabels(self.dl_studio.class_labels),
                rotation = 35,
                horizontalalignment='right'
                );
            ax.set_yticklabels(
                ax.set_xticklabels(self.dl_studio.class_labels),
                rotation = 'horizontal'
                );

            if net.skip_connections == False:
                title = 'Normalized Confusion Matrix (Without Skip Connections)'
                net_name = 'without_skip_connections'
            else:
                title = 'Normalized Confusion Matrix (With Skip Connections)'
                net_name = 'with_skip_connections'


            plt.xlabel('Predicted label\n\nOverall accuracy=%.3f' % acc)
            plt.ylabel('True label')
            plt.title(title)

            path_saved_image = './Val Results/'
            if not os.path.exists(path_saved_image):
                os.makedirs(path_saved_image)

            filename = path_saved_image + "confusion_matrix_" + net_name
            plt.savefig(filename + '.png', bbox_inches = "tight")
            im = Image.open(filename + '.png')
            if im.mode in ("RGBA", "P"):
                im = im.convert("RGB")
            im.save(filename + '.jpg','JPEG', quality = 95)
            os.remove(filename + '.png')
            plt.show()

def get_image(img_url, img_class, path_img_class, cat_id, anns, resize_size):
    if len(img_url) <= 1:
        print("Invalid URL.")
        return {}
    try:
        img_resp = requests.get(img_url, timeout = 1)
    except ConnectionError:
        print("Connection Error.")
        return {}
    except ReadTimeout:
        print("Read timeout.")
        return {}
    except TooManyRedirects:
        print("Too may redirects.")
        return {}
    except MissingSchema:
```

```
    print("Missing schema.")
    return {}
except InvalidURL:
    print("Invalid URL.")
    return {}
if not 'content-type' in img_resp.headers:
    print("Missing content.")
    return {}
if not 'image' in img_resp.headers['content-type']:
    print("The URL doesn't have any image.")
    return {}

img_name = img_url.split('/')[-1]
img_name = img_name.split("?")[0]
img_name_no_jpg = img_name.split('.')[0]

if (len(img_name) <= 1):
    print("Missing image name.")
    return {}

output_dict = {}
img_dict = {}

# find the class_id
if img_class == 'cat':
    class_id = 0
elif img_class == 'train':
    class_id = 1
elif img_class == 'elephant':
    class_id = 2
elif img_class == 'airplane':
    class_id = 3
elif img_class == 'giraffe':
    class_id = 4
img_dict['class_id'] = class_id



# check if the dominant object in the image is our target (i.e., equal to cat_id)
cat_id_list = []
bbox_value_list = []
bbox_area_list = []
for ann in anns:
    cat_id_list.append(ann['category_id'])
    bbox_value_list.append(ann['bbox'])
    bbox_area_list.append(ann['bbox'][2] * ann['bbox'][3])

dominant_ann_id = np.argmax(bbox_area_list)

if cat_id_list[dominant_ann_id] != cat_id:
    return {}
```

```
img_path = os.path.join(path_img_class, img_name)
with open(img_path, 'wb') as img_f:
    img_f.write(img_resp.content)

# save the original bbox
ori_bbox = np.zeros(4)
ori_bbox[0] = bbox_value_list[dominant_ann_id][0]
ori_bbox[1] = bbox_value_list[dominant_ann_id][1]
ori_bbox[2] = ori_bbox[0] + bbox_value_list[dominant_ann_id][2]
ori_bbox[3] = ori_bbox[1] + bbox_value_list[dominant_ann_id][3]
img_dict['ori_bbox'] = ori_bbox.tolist()

# find the original image size
im = Image.open(img_path)
width, height = im.size
img_dict['ori_size'] = [width, height]


# check if the dominant bbox is too small
dominant_bbox_size = bbox_value_list[dominant_ann_id][2] * bbox_value_list[dominant_ann_id][3]
if ( dominant_bbox_size / (width * height) ) < 0.15:
    img_f.close()
    im.close()
    os.remove(img_path)
    return {}

# save the adjusted bbox
resized_width, resized_height = resize_size
bbox = np.zeros(4)
bbox[0] = resized_width * (ori_bbox[0] + 1) / width - 1
bbox[1] = resized_height * (ori_bbox[1] + 1) / height - 1
bbox[2] = resized_width * (ori_bbox[2] + 1)/ width - 1
bbox[3] = resized_height * (ori_bbox[3] + 1)/ height - 1


# make sure the adjusted bbox does not exceed the border of the resized image
bbox[0] = min(max(bbox[0], 0), resized_width - 1)
bbox[1] = min(max(bbox[1], 0), resized_height - 1)
bbox[2] = min(max(bbox[2], 0), resized_width - 1)
bbox[3] = min(max(bbox[3], 0), resized_height - 1)

img_dict['bbox'] = bbox.tolist()

if im.mode != "RGB":
    im = im.convert(mode="RGB")

# save the resized image (64x64)
im_resized = im.resize((resized_width, resized_height), Image.BOX)
np_im_resized = np.array(im_resized)
img_dict['image_r'] = np_im_resized[:,:,0].tolist()
```

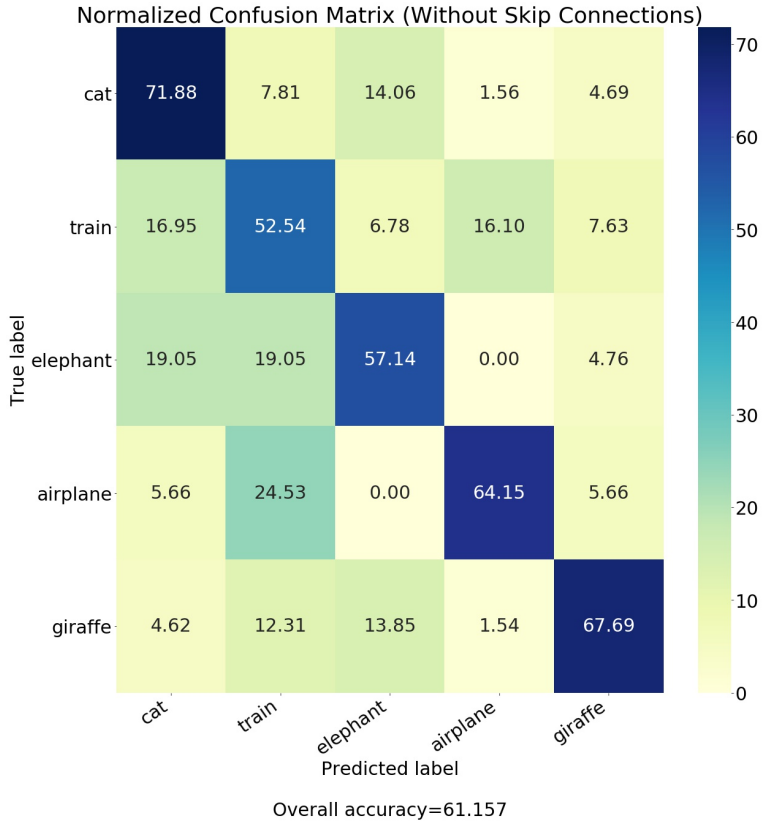
```
    img_dict['image_g'] = np_im_resized[:,:,1].tolist()
    img_dict['image_b'] = np_im_resized[:,:,2].tolist()

    # overwrite original image with downsampled image
    im_resized.save(img_path, quality = 95)

    output_dict[img_name_no_jpg] = img_dict
    return output_dict



def download_img(root_path, coco_json_path, class_list, images_per_class, resize_size):
    # create image folders if they were not existed
    for img_class in class_list:
        path_img_class = os.path.join(root_path, img_class)
        if not os.path.exists(path_img_class):
            os.makedirs(path_img_class)

    # initialize COCO API and get image IDs
    coco = COCO(coco_json_path)
    dict_to_return = {}

    for img_class in class_list:
        path_img_class = os.path.join(root_path, img_class)
        cat_id = coco.getCatIds(catNms = img_class);
        imgs_ids = coco.getImgIds(catIds = cat_id);
        imgs = coco.loadImgs(imgs_ids)
        num_img_downloaded = 0
        current_id = 0
        num_img_to_download = min(images_per_class, len(imgs_ids))
        print('\nGoal: Download %d images of %s\n' % (num_img_to_download, img_class))
        while( (num_img_downloaded < num_img_to_download) and (current_id <  len(imgs_ids)) ):
            annId = coco.getAnnIds(imgIds=imgs_ids[current_id])
            anns = coco.loadAnns(annId)
            img_url = imgs[current_id]['coco_url']
            output_dict = get_image(img_url, img_class, path_img_class, cat_id[0], anns, resize_size)
            if (len(output_dict) > 0):
                num_img_downloaded += 1
                dict_to_return.update(output_dict)
                if num_img_downloaded % 10 == 0:
                    print('Downloaded %d images' % num_img_downloaded)
            current_id += 1
        print('Finish downloading')
        print('\nResult: Downloaded a total of %d images of %s\n' % (num_img_downloaded, img_class))
    print('\nFinish downloading all the images\n')
    return dict_to_return

def plot_val_loss(loss_labeling_list, loss_regression_list):
    bar_wid = 0.25
    font_size = 20
    color_label = 'g'
```

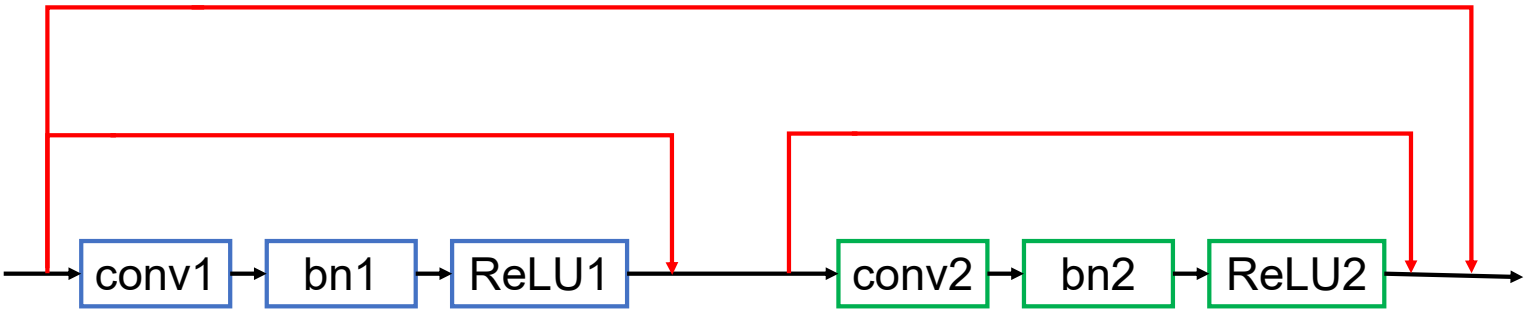
```
color_regre = 'b'
font = {'size': font_size}
plt.rc('font', **font)

# plot labeling loss
plt.figure(figsize = (16,9))
x_axis = ['Without Skip Connections', 'With Skip Connections']
plt.bar(x_axis, loss_labeling_list, color = color_label, width = bar_wid, label ='Loss
↪  (Detection)')

plt.ylabel('loss', fontweight ='bold', fontsize = 30)

title = 'Validation Loss (Dectection)'
plt.title(title)
path_saved_image = './Val Results/'
if not os.path.exists(path_saved_image):
    os.makedirs(path_saved_image)
img_title = path_saved_image + title.lower().replace('(', "").replace(')',"").replace(' ','_')
plt.savefig(img_title + '.png', bbox_inches = "tight")
im = Image.open(img_title + '.png')
if im.mode in ("RGBA", "P"):
    im = im.convert("RGB")
im.save(img_title + '.jpg','JPEG', quality = 95)
os.remove(img_title + '.png')
plt.show()

# plot regression loss
plt.figure(figsize = (16,9))
x_axis = ['Without Skip Connections', 'With Skip Connections']
plt.bar(x_axis, loss_regression_list, color = color_regre, width = bar_wid, label ='Loss
↪  (Localization)')

plt.ylabel('loss', fontweight ='bold', fontsize = 30)

title = 'Validation Loss (Localization)'
plt.title(title)
path_saved_image = './Val Results/'
if not os.path.exists(path_saved_image):
    os.makedirs(path_saved_image)
img_title = path_saved_image + title.lower().replace('(', "").replace(')',"").replace(' ','_')
plt.savefig(img_title + '.png', bbox_inches = "tight")
im = Image.open(img_title + '.png')
if im.mode in ("RGBA", "P"):
    im = im.convert("RGB")
im.save(img_title + '.jpg','JPEG', quality = 95)
os.remove(img_title + '.png')
plt.show()
```

```
#_______________________________  Main program starts here  _________________________________
if __name__ == '__main__':


    #_______________________________  Part 1: Download COCO images and extract useful annotations
    ↪  _______________________________
    # Part 1 will be done in the beginning of Part 2. See the class COCO5Dataset.
    # The program will download COCO images of 5 classes ('cat', 'train', 'elephant', 'airplane',
    ↪  'giraffe') using instances_val2017.json.
    # Each class has 125 images.
    # Each image will be resized to 64 x 64.
    # Each image's pixel values and its annotations will be extracted and saved into the JSON file
    ↪  dict_val2017.json.

    #_______________________________  Part 2: Run my designed network (without skip connections)
    ↪  _______________________________

    loss_labeling_list = []
    loss_regression_list = []

    my_dls = MyDLStudio(
                      image_size = [64,64], # resized image width, resized image height
                      path_saved_model = "./net1.pth",
                      momentum = 0.95,
                      learning_rate = 1 * 1e-5,
                      epochs = 10,
                      batch_size = 8,
                      classes = ['cat', 'train', 'elephant', 'airplane', 'giraffe'],
                      debug_train = 1,
                      debug_test = 1,
                      use_gpu = True,
                  )


    my_detector = MyDLStudio.DetectAndLocalize( dl_studio = my_dls )

    my_dataserver_test = MyDLStudio.DetectAndLocalize.COCO5Dataset(
                                        train_or_test = 'test',
                                        dl_studio = my_dls,
                                                                )
    my_detector.dataserver_test = my_dataserver_test

    my_detector.load_COCO5Dataset_validation_set(my_dataserver_test)


    # My designed network (without skip connections)
    my_model = my_detector.MyLOADnet2(skip_connections=False, depth=8)
```

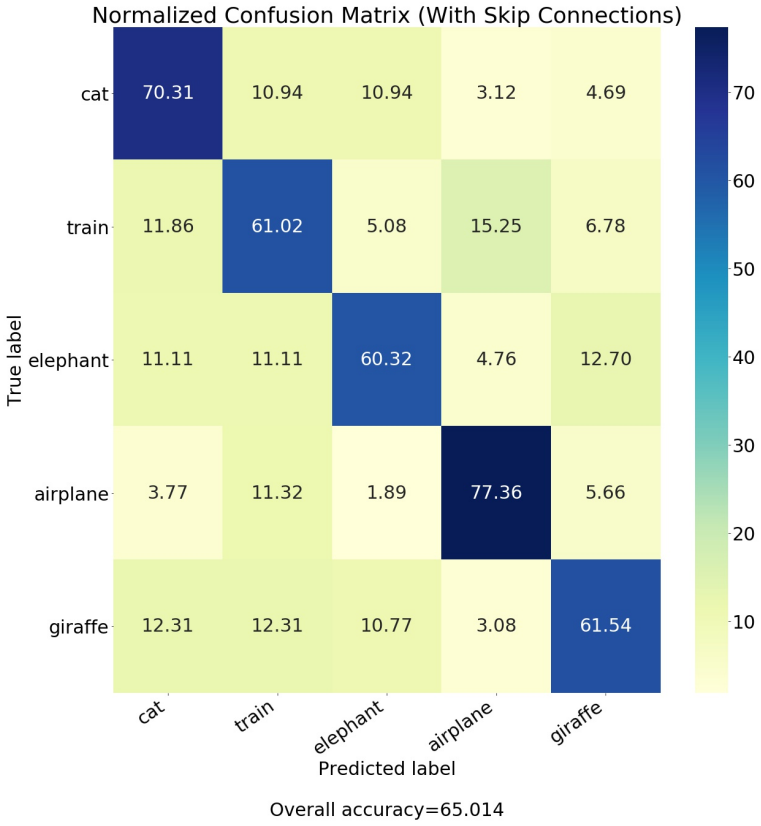
```
number_of_learnable_params = sum(p.numel() for p in my_model.parameters() if p.requires_grad)
print("\n\nThe number of learnable parameters in the my_model: %d" % number_of_learnable_params)

num_layers = len(list(my_model.parameters()))
print("\n\nThe number of layers in the model: %d\n\n" % num_layers)

[acc, confusion_mat, avg_loss_label, avg_loss_regre] =
↪  my_detector.run_code_for_testing_detection_and_localization(my_model)
loss_labeling_list.append(avg_loss_label)
loss_regression_list.append(avg_loss_regre)
my_detector.plot_confusion_mat(acc, confusion_mat, my_model)



#_______________________________  Part 3: Run my designed network (with skip connections)
↪  _______________________________
my_dls_2 = MyDLStudio(
                  image_size = [64,64], # resized image width, resized image height
                  path_saved_model = "./net2.pth",
                  momentum = 0.95,
                  learning_rate = 1 * 1e-5,
                  epochs = 10,
                  batch_size = 8,
                  classes = ['cat', 'train', 'elephant', 'airplane', 'giraffe'],
                  debug_train = 1,
                  debug_test = 1,
                  use_gpu = True,
              )


my_detector_2 = MyDLStudio.DetectAndLocalize( dl_studio = my_dls_2 )

my_dataserver_test_2 = MyDLStudio.DetectAndLocalize.COCO5Dataset(
                                    train_or_test = 'test',
                                    dl_studio = my_dls_2,
                                                          )

my_detector_2.dataserver_test = my_dataserver_test_2

my_detector_2.load_COCO5Dataset_validation_set(my_dataserver_test_2)

# My designed network (with skip connections)
my_model_2 = my_detector_2.MyLOADnet2(skip_connections=True, depth=8)

number_of_learnable_params = sum(p.numel() for p in my_model_2.parameters() if p.requires_grad)
print("\n\nThe number of learnable parameters in the my_model_2: %d" % number_of_learnable_params)

num_layers = len(list(my_model_2.parameters()))
print("\n\nThe number of layers in the model: %d\n\n" % num_layers)

[acc_2, confusion_mat_2, avg_loss_label_2, avg_loss_regre_2] \
```

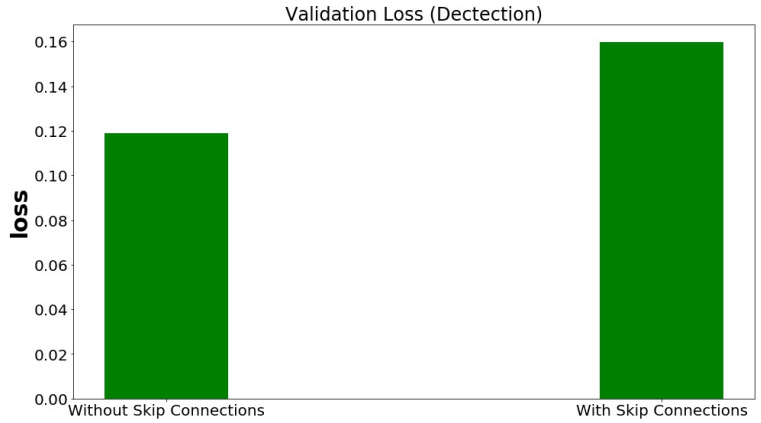
```
    = my_detector_2.run_code_for_testing_detection_and_localization(my_model_2)
loss_labeling_list.append(avg_loss_label_2)
loss_regression_list.append(avg_loss_regre_2)
my_detector_2.plot_confusion_mat(acc_2, confusion_mat_2, my_model_2)


# plot loss values
plot_val_loss(loss_labeling_list, loss_regression_list)
```

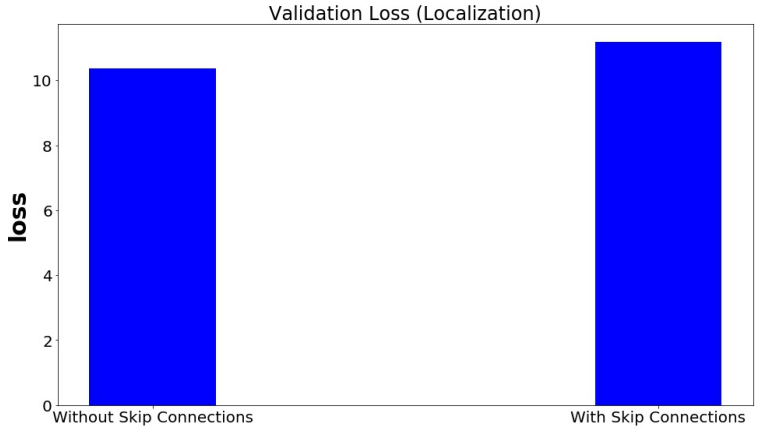


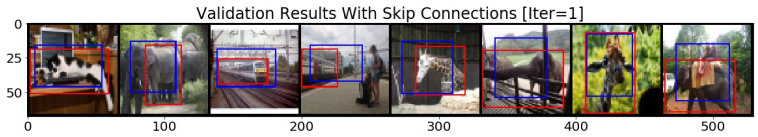


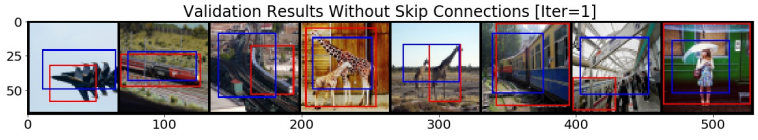
Red line: exclusive for the neural network with skip connections.
There are a total of 3 shortcuts.


Normalized Confusion Matrix (With Skip Connections)
cat
train
elephant
airplane
giraffe
True label
70.31
10.94
10.94
3.12
4.69
11.86
61.02
5.08
15.25
6.78
11.11
11.11
60.32
4.76
12.70
3.77
11.32
1.89
77.36
5.66
12.31
12.31
10.77
3.08
61.54
70
60
50
40
30
20
10
cat
train
elephant
airplane
giraffe
Predicted label
Overall accuracy=65.014


Validation Loss (Dectection)
loss
0.16
0.14
0.12
0.10
0.08
0.06
0.04
0.02
0.00
Without Skip Connections
With Skip Connections


Validation Loss (Localization)
loss
0
2
4
6
8
10
Without Skip Connections
With Skip Connections


Validation Results With Skip Connections [Iter=1]
0
25
50
0
100
200
300
400
500


Validation Results Without Skip Connections [Iter=1]
0
25
50
0
100
200
300
400
500