RTML-Midterm-2022

March 4, 2022

1 RTML Midterm 2022

1.1 Question 1 (20 points)

In Labs 04 and 05, you developed your own PyTorch implementations of YOLOv4 and YOLOR. Download the image at http://www.cs.ait.ac.th/~mdailey/ait-orientation.jpg and run it through your YOLOv4 and YOLOR models. Provide your source code to load the model, image, get the result, and display the result here. Display the resulting bounding boxes.

[81]:
```python
from __future__ import division

import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.autograd import Variable
import numpy as np
#import cv2

def unique(tensor):
    #tensor_np = tensor.cpu().numpy()
    tensor_np = tensor.detach().cpu().numpy()
    unique_np = np.unique(tensor_np)
    unique_tensor = torch.from_numpy(unique_np)

    tensor_res = tensor.new(unique_tensor.shape)
    tensor_res.copy_(unique_tensor)
    return tensor_res


def bbox_iou(box1, box2):
    """
    Returns the IoU of two bounding boxes


    """
    #Get the coordinates of bounding boxes
    b1_x1, b1_y1, b1_x2, b1_y2 = box1[:,0], box1[:,1], box1[:,2], box1[:,3]
    b2_x1, b2_y1, b2_x2, b2_y2 = box2[:,0], box2[:,1], box2[:,2], box2[:,3]
```

```python
    #get the corrdinates of the intersection rectangle
    inter_rect_x1 =  torch.max(b1_x1, b2_x1)
    inter_rect_y1 =  torch.max(b1_y1, b2_y1)
    inter_rect_x2 =  torch.min(b1_x2, b2_x2)
    inter_rect_y2 =  torch.min(b1_y2, b2_y2)

    #Intersection area
    inter_area = torch.clamp(inter_rect_x2 - inter_rect_x1 + 1, min=0) * torch.clamp(inter_rect_y2 - inter_rect_y1 + 1, min=0)

    #Union Area
    b1_area = (b1_x2 - b1_x1 + 1)*(b1_y2 - b1_y1 + 1)
    b2_area = (b2_x2 - b2_x1 + 1)*(b2_y2 - b2_y1 + 1)

    iou = inter_area / (b1_area + b2_area - inter_area)

    return iou

def predict_transform(prediction, inp_dim, anchors, num_classes, CUDA = True):


    batch_size = prediction.size(0)
    stride =  inp_dim // prediction.size(2)
    grid_size = inp_dim // stride
    bbox_attrs = 5 + num_classes
    num_anchors = len(anchors)

    prediction = prediction.view(batch_size, bbox_attrs*num_anchors, grid_size*grid_size)
    prediction = prediction.transpose(1,2).contiguous()
    prediction = prediction.view(batch_size, grid_size*grid_size*num_anchors, bbox_attrs)
    anchors = [(a[0]/stride, a[1]/stride) for a in anchors]

    #Sigmoid the  centre_X, centre_Y. and object confidencce
    prediction[:,:,0] = torch.sigmoid(prediction[:,:,0])
    prediction[:,:,1] = torch.sigmoid(prediction[:,:,1])
    prediction[:,:,4] = torch.sigmoid(prediction[:,:,4])

    #Add the center offsets
    grid = np.arange(grid_size)
    a,b = np.meshgrid(grid, grid)

    x_offset = torch.FloatTensor(a).view(-1,1)
    y_offset = torch.FloatTensor(b).view(-1,1)
```

```
    if CUDA:
        x_offset = x_offset.cuda()
        y_offset = y_offset.cuda()

    x_y_offset = torch.cat((x_offset, y_offset), 1).repeat(1,num_anchors).
 ↪view(-1,2).unsqueeze(0)

    prediction[:,:,:2] += x_y_offset

    #log space transform height and the width
    anchors = torch.FloatTensor(anchors)

    if CUDA:
        anchors = anchors.cuda()

    anchors = anchors.repeat(grid_size*grid_size, 1).unsqueeze(0)
    prediction[:,:,2:4] = torch.exp(prediction[:,:,2:4])*anchors

    prediction[:,:,5: 5 + num_classes] = torch.sigmoid((prediction[:,:, 5 : 5 +␣
 ↪num_classes]))

    prediction[:,:,:4] *= stride

    return prediction

def write_results(prediction, confidence, num_classes, nms_conf = 0.4):
    conf_mask = (prediction[:,:,4] > confidence).float().unsqueeze(2)
    prediction = prediction*conf_mask

    box_corner = prediction.new(prediction.shape)
    box_corner[:,:,0] = (prediction[:,:,0] - prediction[:,:,2]/2)
    box_corner[:,:,1] = (prediction[:,:,1] - prediction[:,:,3]/2)
    box_corner[:,:,2] = (prediction[:,:,0] + prediction[:,:,2]/2)
    box_corner[:,:,3] = (prediction[:,:,1] + prediction[:,:,3]/2)
    prediction[:,:,:4] = box_corner[:,:,:4]

    batch_size = prediction.size(0)

    write = False



    for ind in range(batch_size):
        image_pred = prediction[ind]          #image Tensor
       #confidence threshholding
       #NMS
```

```
        max_conf, max_conf_score = torch.max(image_pred[:,5:5+ num_classes], 1)
        max_conf = max_conf.float().unsqueeze(1)
        max_conf_score = max_conf_score.float().unsqueeze(1)
        seq = (image_pred[:,:5], max_conf, max_conf_score)
        image_pred = torch.cat(seq, 1)

        non_zero_ind =  (torch.nonzero(image_pred[:,4]))
        try:
            image_pred_ = image_pred[non_zero_ind.squeeze(),:].view(-1,7)
        except:
            continue

        if image_pred_.shape[0] == 0:
            continue
#

        #Get the various classes detected in the image
        img_classes = unique(image_pred_[:,-1])  # -1 index holds the class␣
↪index



        for cls in img_classes:
            #perform NMS


            #get the detections with one particular class
            cls_mask = image_pred_*(image_pred_[:,-1] == cls).float().
↪unsqueeze(1)
            class_mask_ind = torch.nonzero(cls_mask[:,-2]).squeeze()
            image_pred_class = image_pred_[class_mask_ind].view(-1,7)

            #sort the detections such that the entry with the maximum objectness
            #confidence is at the top
            conf_sort_index = torch.sort(image_pred_class[:,4], descending =␣
↪True )[1]
            image_pred_class = image_pred_class[conf_sort_index]
            idx = image_pred_class.size(0)   #Number of detections

            for i in range(idx):
                #Get the IOUs of all boxes that come after the one we are␣
↪looking at
                #in the loop
                try:
                    ious = bbox_iou(image_pred_class[i].unsqueeze(0),␣
↪image_pred_class[i+1:])
                except ValueError:
                    break
```

```
                except IndexError:
                    break

                #Zero out all the detections that have IoU > treshhold
                iou_mask = (ious < nms_conf).float().unsqueeze(1)
                image_pred_class[i+1:] *= iou_mask

                #Remove the non-zero entries
                non_zero_ind = torch.nonzero(image_pred_class[:,4]).squeeze()
                image_pred_class = image_pred_class[non_zero_ind].view(-1,7)

            batch_ind = image_pred_class.new(image_pred_class.size(0), 1).
↪fill_(ind)      #Repeat the batch_id for as many detections of the class cls␣
↪in the image
            seq = batch_ind, image_pred_class

            if not write:
                output = torch.cat(seq,1)
                write = True
            else:
                out = torch.cat(seq,1)
                output = torch.cat((output,out))

    try:
        return output
    except:
        return 0

def letterbox_image(img, inp_dim):
    '''resize image with unchanged aspect ratio using padding'''
    img_w, img_h = img.shape[1], img.shape[0]
    w, h = inp_dim
    new_w = int(img_w * min(w/img_w, h/img_h))
    new_h = int(img_h * min(w/img_w, h/img_h))
    resized_image = cv2.resize(img, (new_w,new_h), interpolation = cv2.
↪INTER_CUBIC)

    canvas = np.full((inp_dim[1], inp_dim[0], 3), 128)

    canvas[(h-new_h)//2:(h-new_h)//2 + new_h,(w-new_w)//2:(w-new_w)//2 + new_w,␣
↪ :] = resized_image

    return canvas

def prep_image(img, inp_dim):
    """
```

```
    Prepare image for inputting to the neural network.

    Returns a Variable
    """
    img = (letterbox_image(img, (inp_dim, inp_dim)))
    img = img[:,:,::-1].transpose((2,0,1)).copy()
    img = torch.from_numpy(img).float().div(255.0).unsqueeze(0)
    return img

def load_classes(namesfile):
    fp = open(namesfile, "r")
    names = fp.read().split("\n")[:-1]
    return names
```

[82]:
```
import torch
from torch import tanh
import torch.nn as nn
import torch.nn.functional as F

class Mish(nn.Module):
    def __init__(self):
        super().__init__()
    def forward(self, x):
        return x * tanh(F.softplus(x))
```

[83]:
```
from __future__ import division

import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.autograd import Variable
import numpy as np
#from util import *
#from mish import Mish



def get_test_input():
    img = cv2.imread("ait-orientation.jpeg")
    img = cv2.resize(img, (416,416))          #Resize to the input dimension
    img_ =  img[:,:,::-1].transpose((2,0,1))  # BGR -> RGB | H X W C -> C X H X␣
↪W
    img_ = img_[np.newaxis,:,:,:]/255.0       #Add a channel at 0 (for batch) |␣
↪Normalise
    img_ = torch.from_numpy(img_).float()     #Convert to float
    img_ = Variable(img_)                     # Convert to Variable
    return img_
```

```python
def parse_cfg(cfgfile):
    """
    Takes a configuration file

    Returns a list of blocks. Each blocks describes a block in the neural
    network to be built. Block is represented as a dictionary in the list

    """

    file = open(cfgfile, 'r')
    lines = file.read().split('\n')                        # store the lines in␣
↪a list
    lines = [x for x in lines if len(x) > 0]               # get read of the␣
↪empty lines
    lines = [x for x in lines if x[0] != '#']              # get rid of comments
    lines = [x.rstrip().lstrip() for x in lines]           # get rid of fringe␣
↪whitespaces

    block = {}
    blocks = []

    for line in lines:
        if line[0] == "[":                 # This marks the start of a new block
            if len(block) != 0:            # If block is not empty, implies it is␣
↪storing values of previous block.
                blocks.append(block)       # add it the blocks list
                block = {}                 # re-init the block
            block["type"] = line[1:-1].rstrip()
        else:
            key,value = line.split("=")
            block[key.rstrip()] = value.lstrip()
    blocks.append(block)

    return blocks


class EmptyLayer(nn.Module):
    def __init__(self):
        super(EmptyLayer, self).__init__()


class DetectionLayer(nn.Module):
    def __init__(self, anchors):
        super(DetectionLayer, self).__init__()
        self.anchors = anchors
```

```python
def create_modules(blocks):
    net_info = blocks[0]     #Captures the information about the input and␣
↪pre-processing
    module_list = nn.ModuleList()
    prev_filters = 3
    output_filters = []

    for index, x in enumerate(blocks[1:]):
        module = nn.Sequential()

        #check the type of block
        #create a new module for the block
        #append to module_list

        #If it's a convolutional layer
        if (x["type"] == "convolutional"):
            #Get the info about the layer
            activation = x["activation"]
            try:
                batch_normalize = int(x["batch_normalize"])
                bias = False
            except:
                batch_normalize = 0
                bias = True

            filters= int(x["filters"])
            padding = int(x["pad"])
            kernel_size = int(x["size"])
            stride = int(x["stride"])

            if padding:
                pad = (kernel_size - 1) // 2
            else:
                pad = 0

            #Add the convolutional layer
            conv = nn.Conv2d(prev_filters, filters, kernel_size, stride, pad,␣
↪bias = bias)
            module.add_module("conv_{0}".format(index), conv)

            #Add the Batch Norm Layer
            if batch_normalize:
                bn = nn.BatchNorm2d(filters)
                module.add_module("batch_norm_{0}".format(index), bn)

            #Check the activation.
```

```
            #It is either Linear or a Leaky ReLU for YOLO
            if activation == "leaky":
                activn = nn.LeakyReLU(0.1, inplace = True)
                module.add_module("leaky_{0}".format(index), activn)

            #''' Mish activation modification here '''
            elif activation == "mish":
                activn = Mish()
                module.add_module("mish_{0}".format(index), activn)


        #If it's an upsampling layer
        #We use Bilinear2dUpsampling
        elif (x["type"] == "upsample"):
            stride = int(x["stride"])
            upsample = nn.Upsample(scale_factor = 2, mode = "nearest")
            module.add_module("upsample_{}".format(index), upsample)

        #''' route layermodification here '''
        elif (x["type"] == "route"):
            x["layers"] = x["layers"].split(',')
            filters = 0

            for i in range(len(x["layers"])):
                pointer = int(x["layers"][i])
                if  pointer > 0:
                    filters += output_filters[pointer]
                else:
                    filters += output_filters[index + pointer]

            route = EmptyLayer()
            module.add_module("route_{0}".format(index), route)

        #shortcut corresponds to skip connection
        elif x["type"] == "shortcut":
            shortcut = EmptyLayer()
            module.add_module("shortcut_{}".format(index), shortcut)

        #Yolo is the detection layer
        elif x["type"] == "yolo":
            mask = x["mask"].split(",")
            mask = [int(x) for x in mask]

            anchors = x["anchors"].split(",")
            anchors = [int(a) for a in anchors]
            anchors = [(anchors[i], anchors[i+1]) for i in range(0,␣
↪len(anchors),2)]
```

```
            anchors = [anchors[i] for i in mask]

            detection = DetectionLayer(anchors)
            module.add_module("Detection_{}".format(index), detection)

       # ''' Max pooling layer modification here '''
        elif x["type"] == "maxpool":
            stride = int(x["stride"])
            size = int(x["size"])
            max_pool = nn.MaxPool2d(size, stride, padding=size // 2)
            module.add_module("maxpool_{}".format(index), max_pool)

        module_list.append(module)
        prev_filters = filters
        output_filters.append(filters)

    return (net_info, module_list)




class MyDarknet(nn.Module):
    def __init__(self, cfgfile):
        super(MyDarknet, self).__init__()
        # load the config file and create our model
        self.blocks = parse_cfg(cfgfile)
        self.net_info, self.module_list = create_modules(self.blocks)

    def forward(self, x, CUDA:bool):
        modules = self.blocks[1:]
        outputs = {}   #We cache the outputs for the route layer

        write = 0
        # run forward propagation. Follow the instruction from dictionary␣
 ↪modules
        for i, module in enumerate(modules):
            module_type = (module["type"])

            if module_type == "convolutional" or module_type == "upsample":
                # do convolutional network
                x = self.module_list[i](x)

            elif module_type == "route":
                # concat layers
                layers = module["layers"]
                layers = [int(a) for a in layers]
```

```
                if (layers[0]) > 0:
                    layers[0] = layers[0] - i

                if len(layers) == 1:
                    x = outputs[i + (layers[0])]

                else:
                    if (layers[1]) > 0:
                        layers[1] = layers[1] - i

                    map1 = outputs[i + layers[0]]
                    map2 = outputs[i + layers[1]]
                    x = torch.cat((map1, map2), 1)


            elif  module_type == "shortcut":
                from_ = int(module["from"])
                # residual network
                x = outputs[i-1] + outputs[i+from_]

            elif module_type == 'yolo':
                anchors = self.module_list[i][0].anchors
                #Get the input dimensions
                inp_dim = int (self.net_info["height"])

                #Get the number of classes
                num_classes = int (module["classes"])

                #Transform
                #x = x.data
                # predict_transform is in util.py
                x = predict_transform(x, inp_dim, anchors, num_classes, CUDA)
                if not write:              #if no collector has been intialised.

                    detections = x
                    write = 1

                else:
                    detections = torch.cat((detections, x), 1)

            outputs[i] = x

        return detections


    def load_weights(self, weightfile, backbone_only = False):
        '''
```

```
        Load pretrained weight
        '''
        #Open the weights file
        fp = open(weightfile, "rb")

        #The first 5 values are header information
        # 1. Major version number
        # 2. Minor Version Number
        # 3. Subversion number
        # 4,5. Images seen by the network (during training)
        header = np.fromfile(fp, dtype = np.int32, count = 5)
        self.header = torch.from_numpy(header)
        self.seen = self.header[3]

        weights = np.fromfile(fp, dtype = np.float32)

        ptr = 0
        for i in range(len(self.module_list)):

            if i>104 and backbone_only:
                break

            module_type = self.blocks[i + 1]["type"]

            #If module_type is convolutional load weights
            #Otherwise ignore.

            if module_type == "convolutional":
                model = self.module_list[i]
                try:
                    batch_normalize = int(self.blocks[i+1]["batch_normalize"])
                except:
                    batch_normalize = 0

                conv = model[0]


                if (batch_normalize):
                    bn = model[1]

                    #Get the number of weights of Batch Norm Layer
                    num_bn_biases = bn.bias.numel()

                    #Load the weights
                    bn_biases = torch.from_numpy(weights[ptr:ptr +␣
↪num_bn_biases])
                    ptr += num_bn_biases
```

```
                bn_weights = torch.from_numpy(weights[ptr: ptr +␣
↪num_bn_biases])
                ptr  += num_bn_biases

                bn_running_mean = torch.from_numpy(weights[ptr: ptr +␣
↪num_bn_biases])
                ptr  += num_bn_biases

                bn_running_var = torch.from_numpy(weights[ptr: ptr +␣
↪num_bn_biases])
                ptr  += num_bn_biases

                #Cast the loaded weights into dims of model weights.
                bn_biases = bn_biases.view_as(bn.bias.data)
                bn_weights = bn_weights.view_as(bn.weight.data)
                bn_running_mean = bn_running_mean.view_as(bn.running_mean)
                bn_running_var = bn_running_var.view_as(bn.running_var)

                #Copy the data to model
                bn.bias.data.copy_(bn_biases)
                bn.weight.data.copy_(bn_weights)
                bn.running_mean.copy_(bn_running_mean)
                bn.running_var.copy_(bn_running_var)

            else:
                #Number of biases
                num_biases = conv.bias.numel()

                #Load the weights
                conv_biases = torch.from_numpy(weights[ptr: ptr +␣
↪num_biases])
                ptr = ptr + num_biases

                #reshape the loaded weights according to the dims of the␣
↪model weights

                conv_biases = conv_biases.view_as(conv.bias.data)

                #Finally copy the data
                conv.bias.data.copy_(conv_biases)

            #Let us load the weights for the Convolutional layers
            num_weights = conv.weight.numel()

            #Do the same as above for weights
            conv_weights = torch.from_numpy(weights[ptr:ptr+num_weights])
            ptr = ptr + num_weights
```

```
                conv_weights = conv_weights.view_as(conv.weight.data)
                conv.weight.data.copy_(conv_weights)
```

[84]:
```
#!wget https://pjreddie.com/media/files/yolov4.weights
```

[85]:
```
#!wget https://raw.githubusercontent.com/ayooshkathuria/
 ↪YOLO_v3_tutorial_from_scratch/master/cfg/yolov4.cfg
```

[ ]:
```
### YOLOv4
```

[86]:
```
# Code to load model, image, and display result here



import torch
import torchvision
from PIL import Image



#import cv2
import torch
#from util import *
#from darknet import MyDarknet

def get_test_input():
    img = cv2.imread("ait-orientation.jpeg")
    img = cv2.resize(img, (416,416))          #Resize to the input dimension
    img_ =  img[:,:,::-1].transpose((2,0,1))  # BGR -> RGB | H X W C -> C X H X␣
 ↪W
    img_ = img_[np.newaxis,:,:,:]/255.0       #Add a channel at 0 (for batch) |␣
 ↪Normalise
    img_ = torch.from_numpy(img_).float()     #Convert to float
    img_ = Variable(img_)                     # Convert to Variable
    return img_

#Create YOLOv3 model
model = MyDarknet("yolov4.cfg")
model.load_weights("yolov4.weights")

#model = MyDarknet("cfg/yolov3.cfg")
inp = get_test_input()
pred = model(inp, False)
print (pred)
print('Output tensor size :', pred.shape)



result = write_results(pred, 0.5, 80, nms_conf = 0.4)
```

```
print(result)


def load_classes(namesfile):
    fp = open(namesfile, "r")
    names = fp.read().split("\n")[:-1]
    return names

num_classes = 91
coco_names = [
    '__background__', 'person', 'bicycle', 'car', 'motorcycle', 'airplane',␣
↪'bus',
    'train', 'truck', 'boat', 'traffic light', 'fire hydrant', 'N/A', 'stop␣
↪sign',
    'parking meter', 'bench', 'bird', 'cat', 'dog', 'horse', 'sheep', 'cow',
    'elephant', 'bear', 'zebra', 'giraffe', 'N/A', 'backpack', 'umbrella', 'N/
↪A', 'N/A',
    'handbag', 'tie', 'suitcase', 'frisbee', 'skis', 'snowboard', 'sports ball',
    'kite', 'baseball bat', 'baseball glove', 'skateboard', 'surfboard',␣
↪'tennis racket',
    'bottle', 'N/A', 'wine glass', 'cup', 'fork', 'knife', 'spoon', 'bowl',
    'banana', 'apple', 'sandwich', 'orange', 'broccoli', 'carrot', 'hot dog',␣
↪'pizza',
    'donut', 'cake', 'chair', 'couch', 'potted plant', 'bed', 'N/A', 'dining␣
↪table',
    'N/A', 'N/A', 'toilet', 'N/A', 'tv', 'laptop', 'mouse', 'remote',␣
↪'keyboard', 'cell phone',
    'microwave', 'oven', 'toaster', 'sink', 'refrigerator', 'N/A', 'book',
    'clock', 'vase', 'scissors', 'teddy bear', 'hair drier', 'toothbrush'
]
print(classes)
```

```
        ␣
↪---------------------------------------------------------------------------

        RuntimeError                              Traceback (most recent call␣
↪last)

        /tmp/ipykernel_114/173419776.py in <module>
         23 #Create YOLOv3 model
         24 model = MyDarknet("yolov4.cfg")
    ---> 25 model.load_weights("yolov4.weights")
         26
         27 #model = MyDarknet("cfg/yolov3.cfg")
```

```
    /tmp/ipykernel_114/2710790522.py in load_weights(self, weightfile,␣
↪backbone_only)
    334                     ptr = ptr + num_weights
    335
--> 336                     conv_weights = conv_weights.view_as(conv.weight.data)
    337                     conv.weight.data.copy_(conv_weights)


    RuntimeError: shape '[1024, 512, 3, 3]' is invalid for input of size␣
↪2552319
```

[ ]:
```
### YOLOR
```

[87]:
```
# Code to load model, image, and display result here


import torch
import torchvision
from PIL import Image


#import cv2
import torch
#from util import *
#from darknet import MyDarknet

def get_test_input():
    img = cv2.imread("ait-orientation.jpeg")
    img = cv2.resize(img, (416,416))          #Resize to the input dimension
    img_ =  img[:,:,::-1].transpose((2,0,1))  # BGR -> RGB | H X W C -> C X H X␣
↪W
    img_ = img_[np.newaxis,:,:,:]/255.0       #Add a channel at 0 (for batch) |␣
↪Normalise
    img_ = torch.from_numpy(img_).float()     #Convert to float
    img_ = Variable(img_)                     # Convert to Variable
    return img_

#Create YOLOv3 model
model = MyDarknet("yolo.cfg")
model.load_weights("yolor_p6.pt")

#model = MyDarknet("cfg/yolov3.cfg")
inp = get_test_input()
pred = model(inp, False)
print (pred)
print('Output tensor size :', pred.shape)
```

```python
result = write_results(pred, 0.5, 80, nms_conf = 0.4)
print(result)


def load_classes(namesfile):
    fp = open(namesfile, "r")
    names = fp.read().split("\n")[:-1]
    return names

num_classes = 91
coco_names = [
    '__background__', 'person', 'bicycle', 'car', 'motorcycle', 'airplane',␣
↪'bus',
    'train', 'truck', 'boat', 'traffic light', 'fire hydrant', 'N/A', 'stop␣
↪sign',
    'parking meter', 'bench', 'bird', 'cat', 'dog', 'horse', 'sheep', 'cow',
    'elephant', 'bear', 'zebra', 'giraffe', 'N/A', 'backpack', 'umbrella', 'N/
↪A', 'N/A',
    'handbag', 'tie', 'suitcase', 'frisbee', 'skis', 'snowboard', 'sports ball',
    'kite', 'baseball bat', 'baseball glove', 'skateboard', 'surfboard',␣
↪'tennis racket',
    'bottle', 'N/A', 'wine glass', 'cup', 'fork', 'knife', 'spoon', 'bowl',
    'banana', 'apple', 'sandwich', 'orange', 'broccoli', 'carrot', 'hot dog',␣
↪'pizza',
    'donut', 'cake', 'chair', 'couch', 'potted plant', 'bed', 'N/A', 'dining␣
↪table',
    'N/A', 'N/A', 'toilet', 'N/A', 'tv', 'laptop', 'mouse', 'remote',␣
↪'keyboard', 'cell phone',
    'microwave', 'oven', 'toaster', 'sink', 'refrigerator', 'N/A', 'book',
    'clock', 'vase', 'scissors', 'teddy bear', 'hair drier', 'toothbrush'
]
print(classes)
```

```
       ␣
↪---------------------------------------------------------------------------

       RuntimeError                              Traceback (most recent call␣
↪last)

       /tmp/ipykernel_114/604548207.py in <module>
        23 #Create YOLOv3 model
        24 model = MyDarknet("yolo.cfg")
   ---> 25 model.load_weights("yolor_p6.pt")
        26
```

```
     27 #model = MyDarknet("cfg/yolov3.cfg")


    /tmp/ipykernel_114/2710790522.py in load_weights(self, weightfile,␣
↪backbone_only)
    334                 ptr = ptr + num_weights
    335
--> 336                 conv_weights = conv_weights.view_as(conv.weight.data)
    337                 conv.weight.data.copy_(conv_weights)


    RuntimeError: shape '[1024, 512, 3, 3]' is invalid for input of size␣
↪3925786
```

[43]:
```
#!wget https://raw.githubusercontent.com/AlexeyAB/darknet/master/cfg/yolov4.cfg
```

```
--2022-03-04 03:00:41--
https://raw.githubusercontent.com/AlexeyAB/darknet/master/cfg/yolov4.cfg
Connecting to 192.41.170.23:3128… connected.
Proxy request sent, awaiting response… 200 OK
Length: 12231 (12K) [text/plain]
Saving to: 'yolov4.cfg'

yolov4.cfg          100%[===================>]  11.94K  --.-KB/s    in 0.03s

2022-03-04 03:00:41 (387 KB/s) - 'yolov4.cfg' saved [12231/12231]
```

[ ]:
```
#!wget https://pjreddie.com/media/files/yolov4.weights
```

[55]:
```
from __future__ import division
import time
import torch
import torch.nn as nn
from torch.autograd import Variable
import numpy as np
#import cv2
#from util import *
import argparse
import os
import os.path as osp
#from darknet import Darknet
import pickle as pkl
import pandas as pd
import random
```

```
images = "cocoimages"
batch_size = 4
confidence = 0.5
nms_thesh = 0.4
start = 0
CUDA = torch.cuda.is_available()

num_classes = 91
classes = [
    '__background__', 'person', 'bicycle', 'car', 'motorcycle', 'airplane',␣
 ↪'bus',
    'train', 'truck', 'boat', 'traffic light', 'fire hydrant', 'N/A', 'stop␣
 ↪sign',
    'parking meter', 'bench', 'bird', 'cat', 'dog', 'horse', 'sheep', 'cow',
    'elephant', 'bear', 'zebra', 'giraffe', 'N/A', 'backpack', 'umbrella', 'N/
 ↪A', 'N/A',
    'handbag', 'tie', 'suitcase', 'frisbee', 'skis', 'snowboard', 'sports ball',
    'kite', 'baseball bat', 'baseball glove', 'skateboard', 'surfboard',␣
 ↪'tennis racket',
    'bottle', 'N/A', 'wine glass', 'cup', 'fork', 'knife', 'spoon', 'bowl',
    'banana', 'apple', 'sandwich', 'orange', 'broccoli', 'carrot', 'hot dog',␣
 ↪'pizza',
    'donut', 'cake', 'chair', 'couch', 'potted plant', 'bed', 'N/A', 'dining␣
 ↪table',
    'N/A', 'N/A', 'toilet', 'N/A', 'tv', 'laptop', 'mouse', 'remote',␣
 ↪'keyboard', 'cell phone',
    'microwave', 'oven', 'toaster', 'sink', 'refrigerator', 'N/A', 'book',
    'clock', 'vase', 'scissors', 'teddy bear', 'hair drier', 'toothbrush'
]

#Set up the neural network

print("Loading network.....")
model = Darknet("yolov4.cfg")

# Edit Convo Layer 114
# Here we need to edit this layer because previously the input channel to this␣
 ↪was set as 1024 but actually this layer needs to accept the input from the␣
 ↪concatenation of four 512-channel layers so I need to modify this layer to␣
 ↪have input channel of 2048
model.module_list[114].conv_114 = nn.Conv2d(2048, 512, kernel_size=(1, 1),␣
 ↪stride=(1, 1), bias=False)

model.load_weights("yolov4.weights")
print("Network successfully loaded")
```

```python
model.net_info["height"] = 416
inp_dim = int(model.net_info["height"])
assert inp_dim % 32 == 0
assert inp_dim > 32

#If there's a GPU availible, put the model on GPU

if CUDA:
    model.cuda()

# Set the model in evaluation mode

model.eval()

read_dir = time.time()

# Detection phase

try:
    imlist = [osp.join(osp.realpath('.'), images, img) for img in os.
↪listdir(images)]
except NotADirectoryError:
    imlist = []
    imlist.append(osp.join(osp.realpath('.'), images))
except FileNotFoundError:
    print ("No file or directory with the name {}".format(images))
    exit()

if not os.path.exists("des"):
    os.makedirs("des")

load_batch = time.time()
loaded_ims = [cv2.imread(x) for x in imlist]

im_batches = list(map(prep_image, loaded_ims, [inp_dim for x in␣
↪range(len(imlist))]))
im_dim_list = [(x.shape[1], x.shape[0]) for x in loaded_ims]
im_dim_list = torch.FloatTensor(im_dim_list).repeat(1,2)


leftover = 0
if (len(im_dim_list) % batch_size):
    leftover = 1

if batch_size != 1:
    num_batches = len(imlist) // batch_size + leftover
    im_batches = [torch.cat((im_batches[i*batch_size : min((i +  1)*batch_size,
```

```
                      len(im_batches))]]))  for i in range(num_batches)]

write = 0

if CUDA:
    im_dim_list = im_dim_list.cuda()

start_det_loop = time.time()
for i, batch in enumerate(im_batches):
    # Load the image
    start = time.time()
    if CUDA:
        batch = batch.cuda()
    with torch.no_grad():
        prediction = model(Variable(batch), CUDA)

    prediction = write_results(prediction, confidence, num_classes, nms_conf =␣
 ↪nms_thesh)

    end = time.time()

    if type(prediction) == int:

        for im_num, image in enumerate(imlist[i*batch_size: min((i + ␣
 ↪1)*batch_size, len(imlist))]):
            im_id = i*batch_size + im_num
            print("{0:20s} predicted in {1:6.3f} seconds".format(image.split("/
 ↪")[-1], (end - start)/batch_size))
            print("{0:20s} {1:s}".format("Objects Detected:", ""))
            print("----------------------------------------------------------")
        continue

    prediction[:,0] += i*batch_size    #transform the atribute from index in␣
 ↪batch to index in imlist

    if not write:                      #If we have't initialised output
        output = prediction
        write = 1
    else:
        output = torch.cat((output,prediction))

    for im_num, image in enumerate(imlist[i*batch_size: min((i + ␣
 ↪1)*batch_size, len(imlist))]):
        im_id = i*batch_size + im_num
        objs = [classes[int(x[-1])] for x in output if int(x[0]) == im_id]
        print("{0:20s} predicted in {1:6.3f} seconds".format(image.split("/
 ↪")[-1], (end - start)/batch_size))
```

```
        print("{0:20s} {1:s}".format("Objects Detected:", " ".join(objs)))
        print("----------------------------------------------------------")

    if CUDA:
        torch.cuda.synchronize()
try:
    output
except NameError:
    print ("No detections were made")
    exit()

im_dim_list = torch.index_select(im_dim_list, 0, output[:,0].long())

scaling_factor = torch.min(416/im_dim_list,1)[0].view(-1,1)

output[:,[1,3]] -= (inp_dim - scaling_factor*im_dim_list[:,0].view(-1,1))/2
output[:,[2,4]] -= (inp_dim - scaling_factor*im_dim_list[:,1].view(-1,1))/2

output[:,1:5] /= scaling_factor

for i in range(output.shape[0]):
    output[i, [1,3]] = torch.clamp(output[i, [1,3]], 0.0, im_dim_list[i,0])
    output[i, [2,4]] = torch.clamp(output[i, [2,4]], 0.0, im_dim_list[i,1])

output_recast = time.time()
class_load = time.time()
colors = [[255, 0, 0], [255, 0, 0], [255, 255, 0], [0, 255, 0], [0, 255, 255],
 ↪[0, 0, 255], [255, 0, 255]]

draw = time.time()

def write(x, results):
    c1 = tuple(x[1:3].int())
    c2 = tuple(x[3:5].int())
    img = results[int(x[0])]
    cls = int(x[-1])
    color = random.choice(colors)
    label = "{0}".format(classes[cls])
    cv2.rectangle(img, c1, c2,color, 1)
    t_size = cv2.getTextSize(label, cv2.FONT_HERSHEY_PLAIN, 1 , 1)[0]
    c2 = c1[0] + t_size[0] + 3, c1[1] + t_size[1] + 4
    cv2.rectangle(img, c1, c2,color, -1)
    cv2.putText(img, label, (c1[0], c1[1] + t_size[1] + 4), cv2.
 ↪FONT_HERSHEY_PLAIN, 1, [225,255,255], 1);
    return img
```

```
list(map(lambda x: write(x, loaded_ims), output))

det_names = pd.Series(imlist).apply(lambda x: "{}/det_{}".format("des",x.
 ↪split("/")[-1]))

list(map(cv2.imwrite, det_names, loaded_ims))

end = time.time()

print("SUMMARY")
print("----------------------------------------------------------")
print("{:25s}: {}".format("Task", "Time Taken (in seconds)"))
print()
print("{:25s}: {:2.3f}".format("Reading addresses", load_batch - read_dir))
print("{:25s}: {:2.3f}".format("Loading batch", start_det_loop - load_batch))
print("{:25s}: {:2.3f}".format("Detection (" + str(len(imlist)) +  " images)",␣
 ↪output_recast - start_det_loop))
print("{:25s}: {:2.3f}".format("Output Processing", class_load - output_recast))
print("{:25s}: {:2.3f}".format("Drawing Boxes", end - draw))
print("{:25s}: {:2.3f}".format("Average time_per_img", (end - load_batch)/
 ↪len(imlist)))
print("----------------------------------------------------------")
```

```
Loading network…

       ␣
 ↪---------------------------------------------------------------------------

        RuntimeError                              Traceback (most recent call␣
 ↪last)

        /tmp/ipykernel_114/631565818.py in <module>
         47 model.module_list[114].conv_114 = nn.Conv2d(2048, 512,␣
 ↪kernel_size=(1, 1), stride=(1, 1), bias=False)
         48
    ---> 49 model.load_weights("yolov4.weights")
         50 print("Network successfully loaded")
         51


        /tmp/ipykernel_114/7441142.py in load_weights(self, weightfile,␣
 ↪backbone_only)
        319                 ptr = ptr + num_weights
        320
    --> 321                 conv_weights = conv_weights.view_as(conv.weight.data)
        322                 conv.weight.data.copy_(conv_weights)
```

```
323
```

```
RuntimeError: shape '[1024, 512, 3, 3]' is invalid for input of size
↪2552319
```

1.2 Question 2 (10 points)

In Labs 02-03, you became familiar with different image classification models and the technique of retraining/fine-tuning a pre-trained model on a new dataset. Let's create a ResNet model for classifying images in the CIFAR100 dataset.

First, create dataset objects for the CIFAR100 training and test sets. You'll find documentation at the torchvision datasets page. To use the already-downloaded dataset on puffer/gourami/guppy, use the following dataset location:

```
train_dataset = torchvision.datasets.CIFAR100('/home/fidji/mdailey/Datasets/CIFAR100', train=T
```

Write some code to get one of the samples from the dataset object. Show your code here, and display the image print its attributes here.

[88]:
```python
# Code to extract a sample from the dataset
import torch
import torchvision
from torchvision import datasets, models, transforms
import torch.nn as nn
import torch.optim as optim
import time
import os
from copy import copy
from copy import deepcopy
import torch.nn.functional as F


# import os

# os.environ['http_proxy'] = 'http://192.41.170.23:3128'
# os.environ['https_proxy'] = 'http://192.41.170.23:3128'
# Set device to GPU or CPU

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

# Allow augmentation transform for training set, no augementation for val/test
↪set

train_preprocess = transforms.Compose([
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
```

```
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

eval_preprocess = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])


#full_train_dataset = torchvision.datasets.CIFAR10(root='./data',␣
↪train=True,download=True)

full_train_dataset = torchvision.datasets.CIFAR100('/home/fidji/mdailey/
↪Datasets/CIFAR100', download=True)

print(len(full_train_dataset))

# train_dataset, val_dataset = torch.utils.data.
↪random_split(full_train_dataset, [40000, 10000])
# train_dataset.dataset = copy(full_train_dataset)
# train_dataset.dataset.transform = train_preprocess
# val_dataset.dataset.transform = eval_preprocess

#test_dataset = torchvision.datasets.CIFAR100('/home/fidji/mdailey/Datasets/
↪CIFAR100', download=True)
```

```
Files already downloaded and verified
50000
```

Show your sample image and its attributes here.

1.3 Question 3 (20 points)

Next, create data loaders for the training dataset and validation dataset (no need to use the test set). Use a batch size of 4 and appropriate transforms for the training and validation sets.

Put your code to create the data loaders, sample one minibatch from the training set, and output the shapes of the tensors comprising the minibatch here.

[106]:
```
# Code to create dataloaders, sample a minibatch, and print out tensor shape␣
↪here

# DataLoaders for the three datasets

train_dataset, val_dataset = torch.utils.data.random_split(full_train_dataset,␣
↪[40000, 10000])
train_dataset.dataset = copy(full_train_dataset)
train_dataset.dataset.transform = train_preprocess
val_dataset.dataset.transform = eval_preprocess
```

```
BATCH_SIZE=4
NUM_WORKERS=4

train_dataloader = torch.utils.data.DataLoader(train_dataset,␣
↪batch_size=BATCH_SIZE,
                                              shuffle=True,␣
↪num_workers=NUM_WORKERS)
val_dataloader = torch.utils.data.DataLoader(val_dataset, batch_size=BATCH_SIZE,
                                              shuffle=False,␣
↪num_workers=NUM_WORKERS)


dataloaders = {'train': train_dataloader, 'val': val_dataloader}



for inputs, labels in train_dataloader:

    inputs = inputs.to(device)
    labels = labels.to(device)

    print(inputs.shape)
    print(labels.shape)


    # outputs = resnet(inputs)
    # print('outputs', outputs)
    break
```

```
torch.Size([4, 3, 32, 32])
torch.Size([4])
```

1.4 Question 4 (20 points)

Next, create a ResNet-50 model with pretrained weights from ImageNet using the torchvision ResNet class. Remove the classification layer and replace it with a layer appropriate for identification in CIFAR100. Show that your resulting model can process a minibatch from your validation dataloader and output (incorrect) identities.

[90]:
```
# Code to create a ResNet 50 model, remove classification layer, replace with a␣
↪CIFAR100 identity layer, and run in evaluation model on a validation␣
↪minibatch
class BasicBlock(nn.Module):
    '''
    BasicBlock: Simple residual block with two conv layers
    '''
    EXPANSION = 1
```

```
    def __init__(self, in_planes, out_planes, stride=1):
        super().__init__()
        self.conv1 = nn.Conv2d(in_planes, out_planes, kernel_size=3,␣
↪stride=stride, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(out_planes)
        self.conv2 = nn.Conv2d(out_planes, out_planes, kernel_size=3, stride=1,␣
↪padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(out_planes)
        self.shortcut = nn.Sequential()
        # If output size is not equal to input size, reshape it with 1x1␣
↪convolution
        if stride != 1 or in_planes != out_planes:
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_planes, out_planes, kernel_size=1, stride=stride,␣
↪bias=False),
                nn.BatchNorm2d(out_planes)
            )

    def forward(self, x):
        out = F.relu(self.bn1(self.conv1(x)))
        out = self.bn2(self.conv2(out))
        out += self.shortcut(x)
        out = F.relu(out)
        return out
```

[91]:
```
class BottleneckBlock(nn.Module):
    '''
    BottleneckBlock: More powerful residual block with three convs, used for␣
↪Resnet50 and up
    '''
    EXPANSION = 4
    def __init__(self, in_planes, planes, stride=1):
        super().__init__()
        self.conv1 = nn.Conv2d(in_planes, planes, kernel_size=1, bias=False)
        self.bn1 = nn.BatchNorm2d(planes)
        self.conv2 = nn.Conv2d(planes, planes, kernel_size=3, stride=stride,␣
↪padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(planes)
        self.conv3 = nn.Conv2d(planes, self.EXPANSION * planes, kernel_size=1,␣
↪bias=False)
        self.bn3 = nn.BatchNorm2d(self.EXPANSION * planes)

        self.shortcut = nn.Sequential()
        # If the output size is not equal to input size, reshape it with 1x1␣
↪convolution
        if stride != 1 or in_planes != self.EXPANSION * planes:
```

```
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_planes, self.EXPANSION * planes,
                          kernel_size=1, stride=stride, bias=False),
                nn.BatchNorm2d(self.EXPANSION * planes)
            )

    def forward(self, x):
        out = F.relu(self.bn1(self.conv1(x)))
        out = F.relu(self.bn2(self.conv2(out)))
        out = self.bn3(self.conv3(out))
        out += self.shortcut(x)
        out = F.relu(out)
        return out
```

[92]:
```
class ResNet(nn.Module):
    def __init__(self, block, num_blocks, num_classes=10):
        super().__init__()
        self.in_planes = 64
        # Initial convolution
        self.conv1 = nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1,␣
↪bias=False)
        self.bn1 = nn.BatchNorm2d(64)
        # Residual blocks
        self.layer1 = self._make_layer(block, 64, num_blocks[0], stride=1)
        self.layer2 = self._make_layer(block, 128, num_blocks[1], stride=2)
        self.layer3 = self._make_layer(block, 256, num_blocks[2], stride=2)
        self.layer4 = self._make_layer(block, 512, num_blocks[3], stride=2)
        # FC layer = 1 layer
        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
        self.linear = nn.Linear(512 * block.EXPANSION, num_classes)

    def _make_layer(self, block, planes, num_blocks, stride):
        strides = [stride] + [1] * (num_blocks-1)
        layers = []
        for stride in strides:
            layers.append(block(self.in_planes, planes, stride))
            self.in_planes = planes * block.EXPANSION
        return nn.Sequential(*layers)

    def forward(self, x):
        out = F.relu(self.bn1(self.conv1(x)))
        out = self.layer1(out)
        out = self.layer2(out)
        out = self.layer3(out)
        out = self.layer4(out)
        out = self.avgpool(out)
        out = out.view(out.size(0), -1)
```

```
        out = self.linear(out)
        return out
```

[93]:
```
def ResNet50(num_classes = 100):
    '''
    First conv layer: 1
    4 residual blocks with [3, 4, 6, 3] sets of three convolutions each: 3*3 +␣
↪4*3 + 6*3 + 3*3 = 48
    last FC layer: 1
    Total layers: 1+48+1 = 50
    '''
    return ResNet(BottleneckBlock, [3, 4, 6, 3], num_classes)
```

[94]:
```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print('Using device', device)

resnet = ResNet50().to(device)

def count_parameters(model):
    return sum(p.numel() for p in model.parameters() if p.requires_grad)

print(f'number of trainable parameters: {count_parameters(resnet)}')

print(resnet)
```

```
Using device cuda:0
number of trainable parameters: 23705252
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (layer1): Sequential(
    (0): BottleneckBlock(
      (conv1): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (shortcut): Sequential(
        (0): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
```

```
track_running_stats=True)
      )
    )
    (1): BottleneckBlock(
      (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (shortcut): Sequential()
    )
    (2): BottleneckBlock(
      (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (shortcut): Sequential()
    )
  )
  (layer2): Sequential(
    (0): BottleneckBlock(
      (conv1): Conv2d(256, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (shortcut): Sequential(
        (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
```

```
    (1): BottleneckBlock(
      (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (shortcut): Sequential()
    )
    (2): BottleneckBlock(
      (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (shortcut): Sequential()
    )
    (3): BottleneckBlock(
      (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (shortcut): Sequential()
    )
  )
  (layer3): Sequential(
    (0): BottleneckBlock(
      (conv1): Conv2d(512, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
```

```
track_running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (shortcut): Sequential(
        (0): Conv2d(512, 1024, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (1): BottleneckBlock(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (shortcut): Sequential()
    )
    (2): BottleneckBlock(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (shortcut): Sequential()
    )
    (3): BottleneckBlock(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (shortcut): Sequential()
```

```
    )
    (4): BottleneckBlock(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (shortcut): Sequential()
    )
    (5): BottleneckBlock(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (shortcut): Sequential()
    )
  )
  (layer4): Sequential(
    (0): BottleneckBlock(
      (conv1): Conv2d(1024, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (shortcut): Sequential(
        (0): Conv2d(1024, 2048, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (1): BottleneckBlock(
      (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
```

```
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (shortcut): Sequential()
    )
    (2): BottleneckBlock(
      (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (shortcut): Sequential()
    )
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (linear): Linear(in_features=2048, out_features=100, bias=True)
)
```

[101]:
```python
for inputs, labels in val_dataloader:

    inputs = inputs.to(device)
    labels = labels.to(device)

    outputs = resnet(inputs)
    print('outputs', outputs)
    print(outputs.shape)
    break
```

```
outputs tensor([[ 8.9998e-01,  1.6702e+00, -1.0239e-01,  3.9116e-01,
3.2572e-01,
         -4.0929e-01,  1.5373e+00,  1.3315e-01, -4.5107e-02,  1.8990e-01,
         -8.3797e-01,  1.1464e+00,  6.7381e-01,  1.1757e-01,  1.2707e+00,
         -1.0679e+00,  1.2887e-01, -3.1658e-01,  6.1735e-01, -6.1088e-01,
         -1.1022e-01,  8.3536e-01, -3.2672e-01,  1.3450e+00,  6.0671e-01,
         -8.8988e-01,  9.9034e-01, -8.2813e-01,  2.5868e-01,  1.9961e-01,
         -2.8454e-01,  4.8063e-02,  4.4529e-01,  9.3352e-01,  1.9787e+00,
          4.3030e-01, -5.2796e-01,  9.5990e-01, -2.3635e+00,  1.2462e+00,
```

```
 5.8215e-01, -3.0574e-01, -5.4765e-01, -1.2662e+00,  1.4668e+00,
-7.2958e-01, -2.5948e-01,  7.2205e-01, -1.1518e+00, -3.7021e-01,
 6.0187e-01,  1.9724e-01, -6.2571e-01,  8.3927e-01, -9.8660e-01,
-5.7788e-01,  4.1529e-01,  2.9495e-01, -2.9577e-02, -1.2082e+00,
-1.1462e+00, -8.8534e-01, -2.2149e-01,  1.5619e-01,  8.9053e-01,
-5.3656e-02, -1.1022e+00,  5.6012e-01, -6.9934e-01,  4.9124e-01,
-8.7174e-01, -3.1645e-01, -1.0918e+00, -9.2082e-01, -4.0551e-01,
-9.1574e-01, -3.5136e-01,  4.2576e-01, -5.9874e-01, -2.9588e-01,
-1.2578e-01,  4.7166e-01,  1.5194e-01,  1.2196e+00, -1.2163e-01,
-6.4394e-01,  4.3781e-01,  8.9811e-01, -3.9048e-01,  4.6303e-02,
 9.5334e-02, -7.2972e-01, -1.0542e+00,  3.6290e-01,  6.0306e-01,
-4.1401e-02,  1.6764e-01, -9.0179e-01,  7.9255e-01,  2.4306e-01],
[ 3.3652e-01,  9.7995e-01, -7.6092e-02,  2.4922e-01,  6.5185e-02,
-4.2356e-01,  7.5814e-01,  2.1694e-01,  4.0049e-02,  3.6240e-01,
-3.2533e-01,  1.2483e+00,  4.3076e-01,  4.2888e-01,  6.1150e-01,
-1.1189e+00,  7.7292e-04, -1.7045e-01,  1.7668e-01, -3.0646e-01,
-2.0876e-01,  1.0809e+00, -2.8076e-01,  1.1685e+00,  5.5551e-01,
-6.7568e-01,  8.9358e-01, -6.6817e-01,  5.4225e-01, -2.7501e-01,
-7.1430e-01,  2.4459e-01,  2.7625e-01,  7.2105e-01,  1.1253e+00,
 1.2990e-01, -9.9642e-01,  8.0834e-01, -1.0307e+00,  9.6251e-01,
-1.4622e-01,  4.9757e-02, -4.6263e-01, -5.3749e-01,  7.2816e-01,
-1.0260e-01, -8.9366e-03,  5.6065e-01, -2.6952e-01, -2.7664e-01,
 2.6980e-01,  7.1925e-01, -6.5756e-01,  2.8432e-01, -1.1414e+00,
-7.7135e-01,  2.1849e-01,  1.3906e-01, -3.1426e-01, -8.0672e-01,
-4.1695e-01, -7.6113e-01, -6.5145e-02, -6.4082e-02,  5.7508e-01,
 2.0735e-01, -3.9553e-01,  5.7266e-01, -4.6811e-01,  7.7756e-01,
-1.7605e-01,  2.5759e-02, -4.5971e-01, -1.0344e+00,  1.4466e-01,
-8.2333e-01, -2.0018e-01,  5.7777e-01, -1.4065e-01, -1.6009e-01,
-5.3490e-01,  2.3186e-01, -2.0280e-05,  3.2391e-01, -5.1936e-01,
-4.8170e-01,  2.7292e-01,  5.3359e-01, -5.8469e-02,  3.4338e-01,
 1.1075e-01, -3.7595e-01, -6.7051e-01, -6.2576e-02,  4.0793e-01,
 7.9966e-03, -1.9656e-01, -4.7401e-01,  1.4209e-01, -2.8892e-01],
[ 4.6785e-01,  1.1490e+00, -5.5182e-01,  3.3921e-01,  4.9347e-02,
-4.4134e-01,  8.6528e-01,  2.0787e-01,  2.5063e-01,  5.2560e-01,
-3.5657e-01,  1.2278e+00,  4.1537e-01,  6.9597e-01,  1.0183e+00,
-1.1772e+00, -2.4501e-01, -1.0089e-01,  2.2724e-01, -1.2840e-01,
-1.1995e-01,  9.2416e-01, -1.2651e-01,  1.1671e+00,  2.7198e-01,
-6.4583e-01,  8.3780e-01, -9.0966e-01,  4.0444e-01,  9.9777e-02,
-3.1198e-01,  4.0287e-01,  1.8299e-01,  6.7662e-01,  1.4482e+00,
 2.6842e-01, -9.1646e-01,  7.3840e-01, -1.3286e+00,  7.9004e-01,
 5.3317e-03,  1.1500e-02, -6.6670e-01, -8.5721e-01,  7.7029e-01,
-4.7755e-01,  1.0883e-01,  5.8415e-01, -2.9199e-01, -1.3133e-01,
 5.3171e-01,  5.3652e-01, -1.0282e+00,  3.2304e-01, -1.0194e+00,
-5.1697e-01,  3.9382e-01,  4.0261e-02, -3.9823e-02, -8.5380e-01,
-7.6522e-01, -8.1315e-01, -1.5117e-01, -1.3390e-01,  8.3196e-01,
 7.7147e-02, -3.4003e-01,  5.4326e-01, -5.6658e-01,  7.2667e-01,
-7.3736e-01, -6.7738e-02, -6.7341e-01, -7.0380e-01,  3.3970e-01,
-7.0079e-01, -6.6631e-01,  4.7850e-01, -2.0972e-01, -4.2343e-01,
```

```
         -3.7236e-01,  1.1773e-01,  1.0293e-02,  8.8695e-01, -2.1974e-01,
         -6.4167e-01,  2.8313e-01,  3.4416e-01, -1.5859e-01,  1.0596e-01,
          3.0805e-01, -6.7648e-01, -9.6337e-01,  4.2950e-01,  5.7700e-01,
         -1.4313e-01, -4.3527e-01, -6.4001e-01,  3.5543e-01,  5.4640e-02],
        [ 1.1878e-01,  1.2880e+00, -1.8852e-01,  3.4545e-01,  4.0316e-01,
         -4.8628e-01,  9.1431e-01, -2.1486e-01, -5.8963e-02,  9.1988e-02,
         -3.3946e-01,  1.3948e+00,  3.4573e-01,  6.1785e-01,  1.1081e+00,
         -1.5921e+00, -2.6362e-01, -2.7745e-01,  4.1687e-01, -3.2295e-01,
         -1.6892e-01,  9.9446e-01, -2.4276e-01,  1.3043e+00,  4.6045e-01,
         -1.0582e+00,  7.6477e-01, -8.1304e-01,  4.5012e-01, -2.1552e-01,
         -6.2382e-01,  3.3891e-01,  5.7551e-01,  8.1190e-01,  1.6309e+00,
          6.8185e-01, -9.9550e-01,  1.0205e+00, -1.6347e+00,  1.2368e+00,
          4.4837e-01,  6.8465e-02, -6.3306e-01, -1.0017e+00,  1.0282e+00,
         -3.7839e-01,  3.5995e-02,  9.0508e-01, -7.3571e-01, -3.2504e-01,
          3.2560e-01,  4.3751e-01, -6.8331e-01,  5.8928e-01, -1.0285e+00,
         -5.4440e-01, -1.3308e-02,  3.1297e-01, -4.2229e-01, -1.0400e+00,
         -5.7200e-01, -9.1167e-01,  1.4999e-01,  1.1832e-01,  1.7518e-01,
          3.9479e-01, -6.4971e-01,  4.6504e-01, -2.2483e-01,  6.6389e-01,
         -9.9112e-01, -1.6883e-01, -8.0644e-01, -8.9429e-01, -7.7068e-02,
         -9.3185e-01, -4.1487e-01,  5.4179e-01, -3.2033e-01, -5.3375e-02,
         -1.6528e-01,  5.9681e-01,  2.9799e-01,  9.8440e-01, -4.4777e-01,
         -7.0076e-01,  5.3666e-01,  6.3056e-01, -1.0645e-01,  4.2828e-01,
          7.5125e-01, -6.5072e-01, -1.1454e+00,  5.4178e-01,  2.4802e-01,
         -5.6690e-02, -2.1093e-01, -7.8539e-01,  6.7639e-01, -6.5312e-02]],
       device='cuda:0', grad_fn=<AddmmBackward0>)
torch.Size([4, 100])
```

1.5 Question 5 (20 points)

Next, write a training function, create an optimizer and loss function, and show training loss and validation loss for one epoch.

Show the new ouptut identities for the validation minibatch used in Question 4.

[96]:
```python
def train_model(model, dataloaders, criterion, optimizer, num_epochs=25,␣
 ↪weights_name='weight_save', is_inception=False):
    since = time.time()

    val_acc_history = []
    loss_acc_history = []

    best_model_wts = deepcopy(model.state_dict())
    best_acc = 0.0

    for epoch in range(num_epochs):
        epoch_start = time.time()

        print('Epoch {}/{}'.format(epoch, num_epochs - 1))
```

```
        print('-' * 10)

        for phase in ['train', 'val']:
            if phase == 'train':
                model.train()
            else:
                model.eval()

            running_loss = 0.0
            running_corrects = 0


            for inputs, labels in dataloaders[phase]:

                inputs = inputs.to(device)
                labels = labels.to(device)

                optimizer.zero_grad()


                with torch.set_grad_enabled(phase == 'train'):

                    if is_inception and phase == 'train':

                        outputs, aux_outputs = model(inputs)

                        loss1 = criterion(outputs, labels)
                        loss2 = criterion(aux_outputs, labels)
                        loss = loss1 + 0.4*loss2
                    else:
                        outputs = model(inputs)
                        loss = criterion(outputs, labels)

                    _, preds = torch.max(outputs, 1)

                    # backward + optimize only if in training phase
                    if phase == 'train':
                        loss.backward()
                        optimizer.step()

                # statistics
                running_loss += loss.item() * inputs.size(0)
                running_corrects += torch.sum(preds == labels.data)

            epoch_loss = running_loss / len(dataloaders[phase].dataset)
            epoch_acc = running_corrects.double() / len(dataloaders[phase].
↪dataset)
```

```
            epoch_end = time.time()

            elapsed_epoch = epoch_end - epoch_start

            print('{} Loss: {:.4f} Acc: {:.4f}'.format(phase, epoch_loss,␣
↪epoch_acc))
            print("Epoch time taken: ", elapsed_epoch)

            # deep copy the model
            if phase == 'val' and epoch_acc > best_acc:
                best_acc = epoch_acc
                best_model_wts = deepcopy(model.state_dict())
                torch.save(model.state_dict(), weights_name + ".pth")
            if phase == 'val':
                val_acc_history.append(epoch_acc)
            if phase == 'train':
                loss_acc_history.append(epoch_loss)

        print()

    time_elapsed = time.time() - since
    print('Training complete in {:.0f}m {:.0f}s'.format(time_elapsed // 60,␣
↪time_elapsed % 60))
    print('Best val Acc: {:4f}'.format(best_acc))

    # load best model weights
    model.load_state_dict(best_model_wts)
    return model, val_acc_history, loss_acc_history
```

[65]:
```
# Code for training, optimizer, loss here, training one epoch, and new result␣
↪on validation minibatch
# Optimizer and loss function
criterion = nn.CrossEntropyLoss()
params_to_update = resnet.parameters()
# Now we'll use Adam optimization
optimizer = optim.Adam(params_to_update, lr=0.01)

best_model, val_acc_history, loss_acc_history = train_model(resnet,␣
↪dataloaders, criterion, optimizer, 1, 'resnet18_bestsofar')
```

```
Epoch 0/0
----------
train Loss: 2.0762 Acc: 0.2142
Epoch time taken:  616.2770862579346
val Loss: 2.2004 Acc: 0.2828
Epoch time taken:  665.5300993919373
```

```
Training complete in 11m 7s
Best val Acc: 0.282800
```

1.6 Question 6 (10 points)

Explain how you could use the model you just created as a classifier model in a Control GAN.

Put your explanation here.

Several studies have been conducted for using a classifier to address the problem. Auxiliary Classifier GAN (AC-GAN)uses a classifier as the discriminator of GAN structure. Triple-GAN uses the classification results as an input for discriminator. However, such methods commonly use a classifier that is attached to a discriminator.

ControlGAN is composed of three neural network structures, which are a generator/decoder, a discriminator and a classifier/encoder. Three- player game is conducted in ControlGAN where the generator tries to deceive the discriminator, which is the same as vanilla GAN, and simultaneously aim to be classified corresponding class by the classifier. The generator and the classifier can be interpreted as a decoder-encoder structure because labels are commonly used for inputs for the generator and outputs for the classifier.

D =argmin{ · LD(tD,D(x; D))+(1−) · LD((1−tD),D(G(z,l; G); D))}, G = argmin{ t · LC(l,G(z,l; G))+LD(tD,D(G(z,l; G); D))}, C =argmin{LC(l,x; C)},

where l is the binary representation of labels of sample x and input data for the generator, tD is the label for discriminator which we set to one in this work, and denotes a parameter for the discriminator. ControlGAN forces features to be mapped onto corresponding l inputted into the generator. The parameter t decides how much the generator focus on the input labels for the generator.

[ ]: