import matplotlib.pyplot as plt

import matplotlib.patches as patches

from collections import Counter

from torch.utils.data import DataLoader

import torch

import torch.nn as nn

config = [

    (32, 3, 1),

    (64, 3, 2),

    ["list", 1],

    (128, 3, 2),

    ["list", 2],

    (256, 3, 2),

    ["list", 8],

    (512, 3, 2),

    ["list", 8],

    (1024, 3, 2),

    ["list", 4],

    (512, 1, 1),

    (1024, 3, 1),

    "sp",

    (256, 1, 1),

    "us",

    (256, 1, 1),

    (512, 3, 1),

    "sp",

    (128, 1, 1),

    "us",

    (128, 1, 1),

    (256, 3, 1),

    "sp",

]

class blockcnn(nn.Module):

  def \_\_init\_\_(self, inchnel, outchnel, bnact=True, \*\*kwargs):

    super(blockcnn, self).\_\_init\_\_()

    self.conv = nn.Conv2d(inchnel, outchnel, bias=not bnact, \*\*kwargs)

    self.btchno = nn.BatchNorm2d(outchnel)

    self.leky = nn.LeakyReLU(0.1)

    self.use\_bnact = bnact

  def send(self, x):

    if self.use\_bnact:

      return self.leky(self.btchno(self.conv(x)))

    else:

      return self.conv(x)

class residul(nn.Module):

  def \_\_init\_\_(self, chnel, use\_residual=True, repeatnum=1):

    super(residul, self).\_\_init\_\_()

    self.layres = nn.ModuleList()

    for repeat in range(repeatnum):

      self.layres += [

          nn.Sequential(

            blockcnn(chnel, chnel//2, kernel=1),

            blockcnn(chnel//2, chnel, kernel=3, padding=1)

          )

      ]

    self.use\_residual = use\_residual

    self.repeatnum = repeatnum

  def send(self, x):

    for layer in self.layres:

      if self.use\_residual:

        x = x + layer(x)

      else:

        x = layer(x)

    return x

class predictioncl(nn.Module):

  def \_\_init\_\_(self, inchnel, classno):

    super(predictioncl, self).\_\_init\_\_()

    self.pred = nn.Sequential(

        blockcnn(inchnel, 2 \* inchnel, kernel=3, padding=1),

        blockcnn(2 \* inchnel, (classno + 5) \* 3, bnact=False, kernel=1), box which in total is 3

    )

    self.classno = classno

  def send(self, x):

    return (

        self.pred(x)

        .reshap(x.shap[0], 3, self.classno + 5, x.shap[2], x.shap[3])

        .permut(0, 1, 3, 4, 2)

      )

class yolov3(nn.Module):

  def \_\_init\_\_(self, inchnel=3, classno=20):

    super(yolov3, self).\_\_init\_\_()

    self.classno = classno

    self.inchnel = inchnel

    self.layres = self.convlayr()

  def send(self, x):

    output = []

    rout = []

    for layer in self.layres:

      if instance(layer, predictioncl):

        output.append(layer(x))

        continue

      x = layer(x)

      if instance(layer, residul) and layer.repeatnum == 8:

        rout.append(x)

      elif instance(layer, nn.Upsample):

        x = torch.cat([x, rout[-1]], dim=1)

        rout.pop()

    return output

  def convlayr(self):

    layres = nn.ModuleList()

    inchnel = self.inchnel

    for modul in config:

      if instance(modul, tuple):

        outchnel, kernel, stride = modul

        layres.append(blockcnn(

            inchnel,

            outchnel,

            kernel=kernel,

            stride=stride,

            padding=1 if kernel == 3 else 0

        ))

        inchnel = outchnel

      elif instance(modul, list):

        repeatnum = modul[1]

        layres.append(residul(inchnel, repeatnum=repeatnum))

      elif instance(modul, str):

        if modul == "sp":

          layres += [

              residul(inchnel, use\_residual=False, repeatnum=1),

              blockcnn(inchnel, inchnel//2, kernel=1),

              predictioncl(inchnel//2, classno = self.classno)

          ]

          inchnel = inchnel // 2

        elif modul == "us":

          layres.append(nn.Upsample(scale\_factor=2))

          inchnel = inchnel \* 3

    return layres

classno = 20

imag = 416

modl = yolov3(classno=classno)

x = torch.randn((2, 3, imag, imag))

out = modl(x)

assert modl(x)[0].shap == (2, 3, imag//32, imag//32, classno + 5)

assert modl(x)[1].shap == (2, 3, imag//16, imag//16, classno + 5)

assert modl(x)[2].shap == (2, 3, imag//8, imag//8, classno + 5)

import cv2

import torch

deviceuse = "cuda" if torch.cuda.is\_available() else "cpu"

worker = 4

batchsize= 32

imag = 416

classno = 20

learing = 1e-5

epochsno = 80

threshold = 0.8

ioumap = 0.5

iounms = 0.45

sp = [imag // 32, imag // 16, imag // 8]

IMG\_DIR = "/kaggle/input/pascalvoc-yolo/images"

LABEL\_DIR = "/kaggle/input/pascalvoc-yolo/labels"

ANCHOR = [

    [(0.28, 0.22), (0.38, 0.48), (0.9, 0.78)],

    [(0.07, 0.15), (0.15, 0.11), (0.14, 0.29)],

    [(0.02, 0.03), (0.04, 0.07), (0.08, 0.06)],

]

classpascal = [

    "aeroplane",

    "bicycle",

    "bird",

    "boat",

    "bottle",

    "bus",

    "car",

    "cat",

    "chair",

    "cow",

    "diningtable",

    "dog",

    "horse",

    "motorbike",

    "person",

    "pottedplant",

    "sheep",

    "sofa",

    "train",

    "tvmonitor"

]

def widthheight(box\_A, box\_B):

    intrsection = torch.min(box\_A[..., 0], box\_B[..., 0]) \* torch.min(

        box\_A[..., 1], box\_B[..., 1]

    )

    uion = (

        box\_A[..., 0] \* box\_A[..., 1] + box\_B[..., 0] \* box\_B[..., 1] - intrsection

    )

    return intrsection / uion

def interction\_uion(preds\_box, label\_box, formate\_box="midpoint"):

    if formate\_box == "midpoint":

        boa1\_a1 = preds\_box[..., 0:1] - preds\_box[..., 2:3] / 2

        boa1\_b1 = preds\_box[..., 1:2] - preds\_box[..., 3:4] / 2

        boa1\_a2 = preds\_box[..., 0:1] + preds\_box[..., 2:3] / 2

        boa1\_b2 = preds\_box[..., 1:2] + preds\_box[..., 3:4] / 2

        boa2\_a1 = label\_box[..., 0:1] - label\_box[..., 2:3] / 2

        boa2\_b1 = label\_box[..., 1:2] - label\_box[..., 3:4] / 2

        boa2\_a2 = label\_box[..., 0:1] + label\_box[..., 2:3] / 2

        boa2\_b2 = label\_box[..., 1:2] + label\_box[..., 3:4] / 2

    if formate\_box == "corners":

        boa1\_a1 = preds\_box[..., 0:1]

        boa1\_b1 = preds\_box[..., 1:2]

        boa1\_a2 = preds\_box[..., 2:3]

        boa1\_b2 = preds\_box[..., 3:4]

        boa2\_a1 = label\_box[..., 0:1]

        boa2\_b1 = label\_box[..., 1:2]

        boa2\_a2 = label\_box[..., 2:3]

        boa2\_b2 = label\_box[..., 3:4]

    a1 = torch.max(boa1\_a1, boa2\_a1)

    b1 = torch.max(boa1\_b1, boa2\_b1)

    a2 = torch.min(boa1\_a2, boa2\_a2)

    b2 = torch.min(boa1\_b2, boa2\_b2)

    intrsection = (a2 - a1).clamp(0) \* (b2 - b1).clamp(0)

    boa1\_area = abs((boa1\_a2 - boa1\_a1) \* (boa1\_b2 - boa1\_b1))

    boa2\_area = abs((boa2\_a2 - boa2\_a1) \* (boa2\_b2 - boa2\_b1))

    return intrsection / (boa1\_area + boa2\_area - intrsection + 1e-6)

def non\_suppression(boxx, iou\_threshold, threshold, formate\_box="corners"):

    assert type(boxx) == list

    boxx = [box for box in boxx if box[1] > threshold]

    boxx = sorted(boxx, key=lambd x: x[1], reverse=True)

    boxx\_after\_nms = []

    while boxx:

        chosen\_box = boxx.pop(0)

        boxx = [

            box

            for box in boxx

            if box[0] != chosen\_box[0]

            or interction\_uion(

                torch.tensor(chosen\_box[2:]),

                torch.tensor(box[2:]),

                formate\_box=formate\_box,

            )

            < iou\_threshold

        ]

        boxx\_after\_nms.append(chosen\_box)

    return boxx\_after\_nms

import numpy as np

import os

import pandas as pd

import torch

from torch.utils.data import Dataset, DataLoader

from PIL import Image, ImageFile

ImageFile.LOAD\_TRUNCATED\_IMAGES = True

class yolo\_dataset(Dataset):

  def \_\_init\_\_(self, file\_csv, dir\_img, dir\_label, anchor,

               imag=416, sp=[13,26,52], cp=20, transform=None):

    self.annotations = pd.read\_csv(file\_csv)

    self.dir\_img = dir\_img

    self.dir\_label = dir\_label

    self.transform = transform

    self.sp = sp

    self.anchor = torch.tensor(anchor[0] + anchor[1] + anchor[2])

    self.num\_anchor\_per\_scale = self.num\_anchor // 3

    self.cp = cp

    self.ignore\_iou\_thresh = 0.5

  def \_\_len\_\_(self):

    return len(self.annotations)

  def \_\_getitem\_\_(self, index):

    label\_path = os.path.join(self.dir\_label, self.annotations.iloc[index, 1])

    boxx = np.roll(np.loadtxt(fname=label\_path, delimiter=" ", ndmin=2), 4, axis=1).tolist()

    img\_path = os.path.join(self.dir\_img, self.annotations.iloc[index, 0])

    image = Image.open(img\_path)

    if self.transform:

      image = self.transform(image)

    targets = [torch.zeros((self.num\_anchor // 3, sp, sp, 6)) for sp in self.sp]

    for box in boxx:

      iou\_anchor = widthheight(torch.tensor(box[2:4]), self.anchor)

      anchor\_indices = iou\_anchor.argsort(descending=True, dim=0)

      x, y, width, height, class\_label = box

      has\_anchor = [False, False, False]

      for anchor\_idx in anchor\_indices:

        scale\_idx = anchor\_idx // self.num\_anchor\_per\_scale

        anchor\_on\_scale = anchor\_idx % self.num\_anchor\_per\_scale

        sp = self.sp[scale\_idx]

        i, j = int(sp\*y), int(sp\*x)

        tokenanker = targets[scale\_idx][anchor\_on\_scale, i, j, 0]

        if not tokenanker and not has\_anchor[scale\_idx]:

          targets[scale\_idx][anchor\_on\_scale, i, j, 0] = 1

          x\_cell, y\_cell = sp\*x - j, sp\*y - i

          width\_cell, height\_cell = (

              width\*sp,

              height\*sp

          )

          box\_coordinates = torch.tensor([x\_cell, y\_cell, width\_cell, height\_cell])

          targets[scale\_idx][anchor\_on\_scale, i, j, 1:5] = box\_coordinates

          targets[scale\_idx][anchor\_on\_scale, i, j, 5] = int(class\_label)

          has\_anchor[scale\_idx] = True

        elif not tokenanker and iou\_anchor[anchor\_idx] > self.ignore\_iou\_thresh:

          targets[scale\_idx][anchor\_on\_scale, i, j, 0] = -1

    return image, tuple(targets)

import torchvision.transforms as transforms

transform = transforms.Compose([transforms.Resize((416, 416)), transforms.ToTensor()])

def get\_loaders(train\_csv\_path, test\_csv\_path):

    train\_dataset = yolo\_dataset(

        train\_csv\_path,

        transform=transform,

        sp=[imag // 32, imag // 16, imag // 8],

        dir\_img=IMG\_DIR,

        dir\_label=LABEL\_DIR,

        anchor=ANCHOR,

    )

    test\_dataset = yolo\_dataset(

        test\_csv\_path,

        transform=transform,

        sp=[imag // 32, imag // 16, imag // 8],

        dir\_img=IMG\_DIR,

        dir\_label=LABEL\_DIR,

        anchor=ANCHOR,

    )

    train\_loader = DataLoader(

        dataset=train\_dataset,

        size\_batch=SIZE\_BATCH,

        shuffle=True,

        drop\_last=False,

    )

    test\_loader = DataLoader(

        dataset=test\_dataset,

        size\_batch=SIZE\_BATCH,

        shuffle=False,

        drop\_last=False,

    )

    return train\_loader, test\_loader

def mean\_average\_precision(

    pred\_boxes, true\_boxes, iou\_threshold=0.5, formate\_box="midpoint", classno=4

):

    average\_precisions = []

    epsilon = 1e-6

    for c in range(classno):

        detections = []

        ground\_truths = []

        for detection in pred\_boxes:

            if detection[1] == c:

                detections.append(detection)

        for true\_box in true\_boxes:

            if true\_box[1] == c:

                ground\_truths.append(true\_box)

        amount\_boxx = Counter([gt[0] for gt in ground\_truths])

        for key, val in amount\_boxx.items():

            amount\_boxx[key] = torch.zeros(val)

        detections.sort(key=lambd x: x[2], reverse=True)

        TP = torch.zeros((len(detections)))

        FP = torch.zeros((len(detections)))

        total\_true\_boxx = len(ground\_truths)

        if total\_true\_boxx == 0:

            continue

        for detection\_idx, detection in enumerate(detections):

            ground\_truth\_img = [

                bbox for bbox in ground\_truths if bbox[0] == detection[0]

            ]

            num\_gts = len(ground\_truth\_img)

            best\_iou = 0

            for idx, gt in enumerate(ground\_truth\_img):

                iou = interction\_uion(

                    torch.tensor(detection[3:]),

                    torch.tensor(gt[3:]),

                    formate\_box=formate\_box,

                )

                if iou > best\_iou:

                    best\_iou = iou

                    best\_gt\_idx = idx

            if best\_iou > iou\_threshold:

                if amount\_boxx[detection[0]][best\_gt\_idx] == 0:

                    TP[detection\_idx] = 1

                    amount\_boxx[detection[0]][best\_gt\_idx] = 1

                else:

                    FP[detection\_idx] = 1

            else:

                FP[detection\_idx] = 1

        TP\_cumsum = torch.cumsum(TP, dim=0)

        FP\_cumsum = torch.cumsum(FP, dim=0)

        recalls = TP\_cumsum / (total\_true\_boxx + epsilon)

        precisions = TP\_cumsum / (TP\_cumsum + FP\_cumsum + epsilon)

        precisions = torch.cat((torch.tensor([1]), precisions))

        recalls = torch.cat((torch.tensor([0]), recalls))

        average\_precisions.append(torch.trapz(precisions, recalls))

    return sum(average\_precisions) / len(average\_precisions)

def get\_evaluation\_boxx(

    loader,

    modl,

    iou\_threshold,

    anchor,

    threshold,

    formate\_box="midpoint",

    device="cuda" if torch.cuda.is\_available() else "cpu",

):

    modl.eval()

    idx\_train = 0

    pred\_all\_boxes = []

    true\_all\_boxes = []

    for batch\_idx, (x, labels) in enumerate(loader):

        x = x.float().to(device)

        with torch.no\_grad():

            prediction = modl(x)

        size\_batch = x.shap[0]

        boxx = [[] for \_ in range(size\_batch)]

        for i in range(3):

            sp = prediction[i].shap[2]

            anchor = torch.tensor([\*anchor[i]]).to(device) \* sp

            i\_scale = boxx\_cell(

                prediction[i], anchor, sp=sp, is\_preds=True

            )

            for idx, (box) in enumerate(i\_scale):

                boxx[idx] += box

        true\_boxx = boxx\_cell(

            labels[2], anchor, sp=sp, is\_preds=False

        )

        for idx in range(size\_batch):

            nms\_boxes = non\_suppression(

                boxx[idx],

                iou\_threshold=iou\_threshold,

                threshold=threshold,

                formate\_box=formate\_box,

            )

            for nms\_box in nms\_boxes:

                pred\_all\_boxes.append([idx\_train] + nms\_box)

            for box in true\_boxx[idx]:

                if box[1] > threshold:

                    true\_all\_boxes.append([idx\_train] + box)

            idx\_train += 1

    modl.train()

    return pred\_all\_boxes, true\_all\_boxes

def boxx\_cell(prediction, anchor, sp, is\_preds=True):

    batchsize= prediction.shap[0]

    num\_anchor = len(anchor)

    box\_prediction = prediction[..., 1:5]

    if is\_preds:

        anchor = anchor.reshap(1, len(anchor), 1, 1, 2)

        box\_prediction[..., 0:2] = torch.sigmoid(box\_prediction[..., 0:2])

        box\_prediction[..., 2:] = torch.exp(box\_prediction[..., 2:]) \* anchor

        scores = torch.sigmoid(prediction[..., 0:1])

        class\_best = torch.argmax(prediction[..., 5:], dim=-1).unsqueeze(-1)

    else:

        scores = prediction[..., 0:1]

        class\_best = prediction[..., 5:6]

    cell\_indices = (

        torch.arange(sp)

        .repeat(prediction.shap[0], 3, sp, 1)

        .unsqueeze(-1)

        .to(prediction.device)

    )

    x = 1 / sp \* (box\_prediction[..., 0:1] + cell\_indices)

    y = 1 / sp \* (box\_prediction[..., 1:2] + cell\_indices.permut(0, 1, 3, 2, 4))

    w\_h = 1 / sp \* box\_prediction[..., 2:4]

    converted\_boxx = torch.cat((class\_best, scores, x, y, w\_h), dim=-1).reshap(SIZE\_BATCH, num\_anchor \* sp \* sp, 6)

    return converted\_boxx.tolist()

class YoloLoss(nn.Module):

  def \_\_init\_\_(self):

    super(YoloLoss, self).\_\_init\_\_()

    self.mse = nn.MSELoss()

    self.bce = nn.BCEWithLogitsLoss()

    self.entropy = nn.CrossEntropyLoss()

    self.sigmoid = nn.Sigmoid()

    self.lambd\_class = 1

    self.lambd\_noobj = 10

    self.lambd\_obj = 1

    self.lambd\_box = 10

  def send(self, prediction, target, anchor):

    obj = target[..., 0] == 1

    noobj = target[..., 0] == 0

    no\_object\_loss = self.bce(

        (prediction[..., 0:1][noobj]), (target[..., 0:1][noobj])

    )

    anchor = anchor.reshap(1,3,1,1,2)

    box\_preds = torch.cat([self.sigmoid(prediction[..., 1:3]), torch.exp(prediction[..., 3:5]) \* anchor], dim=-1)

    ious = interction\_uion(box\_preds[obj], target[..., 1:5][obj]).detach()

    object\_loss = self.bce(

        (prediction[..., 0:1][obj]), (ious \* target[..., 0:1][obj])

    )

    prediction[..., 1:3] = self.sigmoid(prediction[..., 1:3])

    target[..., 3:5] = torch.log(

        (1e-6 + target[..., 3:5] / anchor)

    )

    box\_loss = self.mse(prediction[..., 1:5][obj], target[..., 1:5][obj])

    class\_loss = self.entropy(

        (prediction[..., 5:][obj]), (target[..., 5][obj].long())

    )

    return(

        self.lambd\_box \* box\_loss

        + self.lambd\_obj \* object\_loss

        + self.lambd\_noobj \* no\_object\_loss

        + self.lambd\_class \* class\_loss

    )

def plot\_image(image, boxes):

    cmap = plt.get\_cmap("tab20b")

    class\_labels = classpascal

    colors = [cmap(i) for i in np.linspace(0, 1, len(class\_labels))]

    im = np.array(image)

    height, width, \_ = im.shap

    fig, ax = plt.subplots(1)

    ax.imshow(im)

    for box in boxes:

        assert len(box) == 6

        class\_pred = box[0]

        box = box[2:]

        left\_upper\_x = box[0] - box[2] / 2

        left\_upper\_y = box[1] - box[3] / 2

        rect = patches.Rectangle(

            (left\_upper\_x \* width, left\_upper\_y \* height),

            box[2] \* width,

            box[3] \* height,

            linewidth=2,

            edgecolor=colors[int(class\_pred)],

            facecolor="none",

        )

        ax.add\_patch(rect)

        plt.text(

            left\_upper\_x \* width,

            left\_upper\_y \* height,

            s=class\_labels[int(class\_pred)],

            color="white",

            verticalalignment="top",

            bbox={"color": colors[int(class\_pred)], "pad": 0},

        )

    plt.show()

modl = yolov3(classno=classno).to(deviceuse)

optimizer = torch.optim.Adam(

    modl.parameters(), lr=learing

)

loss\_fn = YoloLoss()

scaler = torch.cuda.amp.GradScaler()

train\_loader, test\_loader = get\_loaders(

    train\_csv\_path='/kaggle/input/pascalvoc-yolo/test.csv', test\_csv\_path='/kaggle/input/pascalvoc-yolo/test.csv'

)

scaled\_anchor = (

    torch.tensor(ANCHOR) \* torch.tensor([13,26,52]).unsqueeze(1).unsqueeze(1).repeat(1,3,2)

).to(deviceuse)

torch.save(test\_loader, '/kaggle/working/test\_loader.pth')

import torch.optim as optim

from tqdm import tqdm

import time

history\_loss = []

for epoch in tqdm(range(epochsno), desc="Epochs"):

  modl.train()

  losses = []

  start\_time = time.time()

  for batch\_idx, (x,y) in enumerate(train\_loader):

    x = x.to(deviceuse)

    y0, y1, y2 = (y[0].to(deviceuse),

                  y[1].to(deviceuse),

                  y[2].to(deviceuse))

    with torch.cuda.amp.autocast():

      out = modl(x)

      loss = (

          loss\_fn(out[0], y0, scaled\_anchors[0])

          + loss\_fn(out[1], y1, scaled\_anchors[1])

          + loss\_fn(out[2], y2, scaled\_anchors[2])

      )

    losses.append(loss.item())

    optimizer.zero\_grad()

    scaler.scale(loss).backward()

    scaler.step(optimizer)

    scaler.update()

  end\_time = time.time()

  epoch\_duration = end\_time - start\_time

  history\_loss.append(sum(losses)/len(losses))

  if (epoch+1) % 10 == 0:

    tqdm.write(f"Epoch {epoch+1} completed in {epoch\_duration:.2f} seconds")

    print(f"Epoch [{epoch+1}/{epochsno}], "

          f"Loss: {sum(losses)/len(losses):.4f}")

    torch.save(modl.state\_dict(), f'/kaggle/working/Yolov3\_epoch{epoch+1}.pth')

modl.eval()

x, y = next(iter(test\_loader))

x = x.float().to(deviceuse)

with torch.no\_grad():

    out = modl(x)

    boxx = [[] for \_ in range(x.shap[0])]

    size\_batch, A, sp, \_, \_ = out[0].shap

    anchor = torch.tensor([\*ANCHOR[0]]).to(deviceuse) \* sp

    i\_scale = boxx\_cell(

        out[0], anchor, sp=sp, is\_preds=True

    )

    for idx, (box) in enumerate(i\_scale):

        boxx[idx] += box

    for i in range(size\_batch):

        nms\_boxes = non\_suppression(

            boxx[i], iou\_threshold=0.5, threshold=0.6, formate\_box="midpoint",

        )

        plot\_image(x[i].permut(1,2,0).detach().cpu(), nms\_boxes)