import matplotlib.pyplot as plt

import matplotlib.patches as patches

import numpy as np

import os

import torch

from collections import Counter

from torch.utils.data import DataLoader

from tqdm import tqdm

import torch.nn as nn

# Configuration for YOLO layers

yolo\_config = [

(32, 3, 1),

(64, 3, 2),

["ResBlk", 1],

(128, 3, 2),

["ResBlk", 2],

(256, 3, 2),

["ResBlk", 8],

(512, 3, 2),

["ResBlk", 8],

(1024, 3, 2),

["ResBlk", 4],

(512, 1, 1),

(1024, 3, 1),

"ScalePred",

(256, 1, 1),

"Upsample",

(256, 1, 1),

(512, 3, 1),

"ScalePred",

(128, 1, 1),

"Upsample",

(128, 1, 1),

(256, 3, 1),

"ScalePred",

]

# Basic convolutional block with optional batch normalization and activation

class ConvBlock(nn.Module):

def \_\_init\_\_(self, in\_channels, out\_channels, use\_bn\_act=True, \*\*kwargs):

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

self.conv = nn.Conv2d(in\_channels, out\_channels, bias=not use\_bn\_act, \*\*kwargs)

self.bn = nn.BatchNorm2d(out\_channels)

self.leaky\_relu = nn.LeakyReLU(0.1)

self.use\_bn\_act = use\_bn\_act

def forward(self, x):

if self.use\_bn\_act:

return self.leaky\_relu(self.bn(self.conv(x)))

else:

return self.conv(x)

# Residual block for YOLO layers

class ResidualBlock(nn.Module):

def \_\_init\_\_(self, channels, use\_residual=True, num\_repeats=1):

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

self.layers = nn.ModuleList()

for \_ in range(num\_repeats):

self.layers.append(

nn.Sequential(

ConvBlock(channels, channels // 2, kernel\_size=1),

ConvBlock(channels // 2, channels, kernel\_size=3, padding=1)

)

)

self.use\_residual = use\_residual

def forward(self, x):

for layer in self.layers:

if self.use\_residual:

x = x + layer(x)

else:

x = layer(x)

return x

# Scale prediction block for YOLO output layers

class PredictionBlock(nn.Module):

def \_\_init\_\_(self, in\_channels, num\_classes):

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

self.pred = nn.Sequential(

ConvBlock(in\_channels, 2 \* in\_channels, kernel\_size=3, padding=1),

ConvBlock(2 \* in\_channels, (num\_classes + 5) \* 3, use\_bn\_act=False, kernel\_size=1),

)

self.num\_classes = num\_classes

def forward(self, x):

return (

self.pred(x)

.reshape(x.shape[0], 3, self.num\_classes + 5, x.shape[2], x.shape[3])

.permute(0, 1, 3, 4, 2)

)

# YOLOv3 model

class YOLOv3Model(nn.Module):

def \_\_init\_\_(self, in\_channels=3, num\_classes=20):

super(YOLOv3Model, self).\_\_init\_\_()

self.num\_classes = num\_classes

self.in\_channels = in\_channels

self.layers = self.\_create\_conv\_layers()

def forward(self, x):

outputs = []

route\_connections = []

for layer in self.layers:

if isinstance(layer, PredictionBlock):

outputs.append(layer(x))

continue

x = layer(x)

if isinstance(layer, ResidualBlock) and layer.layers[0][1].out\_channels == 256:

route\_connections.append(x)

elif isinstance(layer, nn.Upsample):

x = torch.cat([x, route\_connections.pop()], dim=1)

return outputs

def \_create\_conv\_layers(self):

layers = nn.ModuleList()

in\_channels = self.in\_channels

for module in yolo\_config:

if isinstance(module, tuple):

out\_channels, kernel\_size, stride = module

layers.append(

ConvBlock(

in\_channels,

out\_channels,

kernel\_size=kernel\_size,

stride=stride,

padding=1 if kernel\_size == 3 else 0,

)

)

in\_channels = out\_channels

elif isinstance(module, list):

num\_repeats = module[1]

layers.append(ResidualBlock(in\_channels, num\_repeats=num\_repeats))

elif isinstance(module, str):

if module == "ScalePred":

layers.append(

nn.Sequential(

ResidualBlock(in\_channels, use\_residual=False, num\_repeats=1),

ConvBlock(in\_channels, in\_channels // 2, kernel\_size=1),

PredictionBlock(in\_channels // 2, num\_classes=self.num\_classes),

)

)

in\_channels = in\_channels // 2

elif module == "Upsample":

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

in\_channels = in\_channels \* 3

return layers

# Instantiate and test the model

num\_classes = 20

IMAGE\_SIZE = 416

model = YOLOv3Model(num\_classes=num\_classes)

x = torch.randn((2, 3, IMAGE\_SIZE, IMAGE\_SIZE))

out = model(x)

assert model(x)[0].shape == (2, 3, IMAGE\_SIZE // 32, IMAGE\_SIZE // 32, num\_classes + 5)

assert model(x)[1].shape == (2, 3, IMAGE\_SIZE // 16, IMAGE\_SIZE // 16, num\_classes + 5)

assert model(x)[2].shape == (2, 3, IMAGE\_SIZE // 8, IMAGE\_SIZE // 8, num\_classes + 5)

# Data processing and helper functions

def compute\_iou(box1, box2):

inter = torch.min(box1[..., 0], box2[..., 0]) \* torch.min(box1[..., 1], box2[..., 1])

union = box1[..., 0] \* box1[..., 1] + box2[..., 0] \* box2[..., 1] - inter

return inter / union

def intersection\_over\_union(box\_preds, box\_labels, box\_format="midpoint"):

if box\_format == "midpoint":

box1\_x1 = box\_preds[..., 0:1] - box\_preds[..., 2:3] / 2

box1\_y1 = box\_preds[..., 1:2] - box\_preds[..., 3:4] / 2

box1\_x2 = box\_preds[..., 0:1] + box\_preds[..., 2:3] / 2

box1\_y2 = box\_preds[..., 1:2] + box\_preds[..., 3:4] / 2

box2\_x1 = box\_labels[..., 0:1] - box\_labels[..., 2:3] / 2

box2\_y1 = box\_labels[..., 1:2] - box\_labels[..., 3:4] / 2

box2\_x2 = box\_labels[..., 0:1] + box\_labels[..., 2:3] / 2

box2\_y2 = box\_labels[..., 1:2] + box\_labels[..., 3:4] / 2

elif box\_format == "corners":

box1\_x1 = box\_preds[..., 0:1]

box1\_y1 = box\_preds[..., 1:2]

box1\_x2 = box\_preds[..., 2:3]

box1\_y2 = box\_preds[..., 3:4]

box2\_x1 = box\_labels[..., 0:1]

box2\_y1 = box\_labels[..., 1:2]

box2\_x2 = box\_labels[..., 2:3]

box2\_y2 = box\_labels[..., 3:4]

x1 = torch.max(box1\_x1, box2\_x1)

y1 = torch.max(box1\_y1, box2\_y1)

x2 = torch.min(box1\_x2, box2\_x2)

y2 = torch.min(box1\_y2, box2\_y2)

intersection = (x2 - x1).clamp(0) \* (y2 - y1).clamp(0)

box1\_area = abs((box1\_x2 - box1\_x1) \* (box1\_y2 - box1\_y1))

box2\_area = abs((box2\_x2 - box2\_x1) \* (box2\_y2 - box2\_y1))

return intersection / (box1\_area + box2\_area - intersection + 1e-6)

def non\_max\_suppression(bboxes, iou\_threshold, conf\_threshold, box\_format="corners"):

assert type(bboxes) == list

bboxes = [box for box in bboxes if box[1] > conf\_threshold]

bboxes = sorted(bboxes, key=lambda x: x[1], reverse=True)

bboxes\_after\_nms = []

while bboxes:

chosen\_box = bboxes.pop(0)

bboxes = [

box

for box in bboxes

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

or intersection\_over\_union(

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

torch.tensor(box[2:]),

box\_format=box\_format,

)

< iou\_threshold

]

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

return bboxes\_after\_nms

def mean\_avg\_precision(pred\_boxes, true\_boxes, iou\_threshold=0.5, box\_format="midpoint", num\_classes=20):

average\_precisions = []

epsilon = 1e-6

for c in range(num\_classes):

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

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

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

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

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

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

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

if total\_true\_boxes == 0:

continue

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

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

best\_iou = 0

best\_gt\_idx = None

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

iou = intersection\_over\_union(

torch.tensor(detection[3:]),

torch.tensor(gt[3:]),

box\_format=box\_format,

)

if iou > best\_iou:

best\_iou = iou

best\_gt\_idx = idx

if best\_iou > iou\_threshold:

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

TP[detection\_idx] = 1

amount\_bboxes[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\_boxes + 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\_dataloader(train\_csv\_path, test\_csv\_path):

train\_dataset = YOLODataset(

train\_csv\_path,

transform=transform,

S=[IMAGE\_SIZE // 32, IMAGE\_SIZE // 16, IMAGE\_SIZE // 8],

anchors=ANCHORS,

image\_size=IMAGE\_SIZE,

num\_classes=NUM\_CLASSES,

)

test\_dataset = YOLODataset(

test\_csv\_path,

transform=transform,

S=[IMAGE\_SIZE // 32, IMAGE\_SIZE // 16, IMAGE\_SIZE // 8],

anchors=ANCHORS,

image\_size=IMAGE\_SIZE,

num\_classes=NUM\_CLASSES,

)

train\_loader = DataLoader(

dataset=train\_dataset,

batch\_size=BATCH\_SIZE,

num\_workers=NUM\_WORKERS,

pin\_memory=True,

shuffle=True,

drop\_last=False,

)

test\_loader = DataLoader(

dataset=test\_dataset,

batch\_size=BATCH\_SIZE,

num\_workers=NUM\_WORKERS,

pin\_memory=True,

shuffle=False,

drop\_last=False,

)

return train\_loader, test\_loader

# Training function

def train\_fn(train\_loader, model, optimizer, loss\_fn, scaler):

loop = tqdm(train\_loader, leave=True)

avg\_loss = []

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

x = x.to(DEVICE)

y0, y1, y2 = (

y[0].to(DEVICE),

y[1].to(DEVICE),

y[2].to(DEVICE),

)

with torch.cuda.amp.autocast():

out = model(x)

loss = (

loss\_fn(out[0], y0, ANCHORS[0])

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

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

)

avg\_loss.append(loss.item())

optimizer.zero\_grad()

scaler.scale(loss).backward()

scaler.step(optimizer)

scaler.update()

loop.set\_postfix(loss=loss.item())

print(f"Average loss: {sum(avg\_loss)/len(avg\_loss)}")

# Main function for training

def main():

model = YOLOv3Model(num\_classes=NUM\_CLASSES).to(DEVICE)

optimizer = optim.Adam(model.parameters(), lr=LEARNING\_RATE, weight\_decay=1e-4)

loss\_fn = YOLOLoss()

scaler = torch.cuda.amp.GradScaler()

train\_loader, test\_loader = get\_dataloader("train.csv", "test.csv")

for epoch in range(NUM\_EPOCHS):

train\_fn(train\_loader, model, optimizer, loss\_fn, scaler)

if epoch > 0 and epoch % 10 == 0:

print(f"Epoch {epoch}")

pred\_boxes, true\_boxes = get\_evaluation\_boxx(

test\_loader, model, iou\_threshold=NMS\_IOU\_THRESH, threshold=CONF\_THRESHOLD

)

mAP = mean\_avg\_precision(pred\_boxes, true\_boxes, iou\_threshold=MAP\_IOU\_THRESH, box\_format="midpoint", num\_classes=NUM\_CLASSES)

print(f"mAP: {mAP.item()}")

torch.save(model.state\_dict(), "model.pth")

if \_\_name\_\_ == "\_\_main\_\_":

main()