# # 1. 环境设置

import gc

import random

import os

import timm

import torch

import warnings

import torchvision

import numpy as np

import pandas as pd

from PIL import Image

from pathlib import Path

from torch import optim

from torchvision import models

from tqdm.notebook import tqdm

import matplotlib.pyplot as plt

from torchvision import datasets, transforms

from sklearn.model\_selection import train\_test\_split

from torch.utils.data import DataLoader, Dataset, sampler

from sklearn.metrics import accuracy\_score, precision\_recall\_fscore\_support

# 设置随机种子以确保结果可复现

def seed\_everything(seed=42):

os.environ['PYTHONHASHSEED'] = str(seed)

random.seed(seed)

np.random.seed(seed)

torch.manual\_seed(seed)

torch.backends.cudnn.benchmark = False

torch.backends.cudnn.deterministic = True

SEED = 42

seed\_everything(SEED)

warnings.filterwarnings("ignore")

plt.rcParams.update({'axes.titlesize': 20})

# ## 参数设置

# 将超参数放在一起，便于修改

class Args:

def \_\_init\_\_(self) -> None:

# 数据参数

self.num\_classes = 2

self.img\_size = 224

self.num\_train\_data = 20000

self.num\_test\_data = 5000

self.dataset\_path = "./input/mixed/"

# 训练参数

self.learning\_rate = 5e-5

self.epochs = 30

self.scheduler = True

self.sch\_step\_size = 2

self.sch\_gamma = 0.1

# 模型参数

self.drop\_path\_rate = 0.2

self.embed\_dim = 96

self.depths = (2, 2, 6, 2)

self.num\_heads = (3, 6, 12, 24)

self.window\_size = 16

self.save\_model\_path = "./output/mixed\_vit\_base\_patch16\_224.pth"

# 输出参数

self.output\_path = "./output/mixed\_vit\_base16/"

args = Args()

# # 2. 数据处理

# ## 2.1 数据增强

# 对图像进行一些增强操作，以提高训练效果

train\_augmentations = transforms.Compose([

transforms.RandomResizedCrop(args.img\_size, scale=(0.6, 1.0), ratio=(3./ 4., 4. / 3.)),

transforms.RandomHorizontalFlip(0.5),

transforms.RandomPerspective(distortion\_scale=0.2, p=0.2),

transforms.RandomRotation(15),

transforms.ColorJitter(brightness=0.2, contrast=0.1, saturation=0.1, hue=0.1),

transforms.ToTensor(),

transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225]),

])

test\_augmentations = transforms.Compose([

transforms.Resize(args.img\_size),

transforms.CenterCrop(args.img\_size),

transforms.ToTensor(),

transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225]),

])

basic\_augmentations = transforms.Compose([

transforms.Resize(args.img\_size),

transforms.CenterCrop(args.img\_size),

transforms.ToTensor()

])

# ## 2.2 读取和划分数据集

# 读取数据集并将其划分为训练集和测试集。由于原始数据集较大，我们将随机选择一部分数据来完成任务。

# 读取训练和测试数据集

train\_dataset = datasets.ImageFolder(root=args.dataset\_path + "train/", transform=train\_augmentations)

test\_dataset = datasets.ImageFolder(root=args.dataset\_path + "test/", transform=test\_augmentations)

# 选择数据子集

train\_fake\_all\_indices = np.arange(len(train\_dataset) / 2, dtype=np.int32)

train\_fake\_indices = np.random.choice(train\_fake\_all\_indices, size=int(args.num\_train\_data / 2), replace=False)

train\_real\_indices = train\_fake\_indices + int(len(train\_dataset) / 2)

train\_indices = np.append(train\_fake\_indices, train\_real\_indices)

test\_fake\_all\_indices = np.arange(len(test\_dataset) / 2, dtype=np.int32)

test\_fake\_indices = np.random.choice(test\_fake\_all\_indices, size=int(args.num\_test\_data / 2), replace=False)

test\_real\_indices = test\_fake\_indices + int(len(test\_dataset) / 2)

test\_indices = np.append(test\_fake\_indices, test\_real\_indices)

train\_sampler = sampler.SubsetRandomSampler(train\_indices)

train\_loader = torch.utils.data.DataLoader(train\_dataset, batch\_size=32, num\_workers=2, sampler=train\_sampler)

test\_sampler = sampler.SubsetRandomSampler(test\_indices)

test\_loader = torch.utils.data.DataLoader(test\_dataset, batch\_size=8, num\_workers=2, sampler=test\_sampler)

classes = train\_dataset.classes

class\_to\_idx = train\_dataset.class\_to\_idx

idx\_to\_class = dict(zip(class\_to\_idx.values(), class\_to\_idx.keys()))

# ## 2.3 可视化

# 可视化读取的图像以及增强后的数据

raw\_dataset = datasets.ImageFolder(root=args.dataset\_path + "train/", transform=basic\_augmentations)

valid\_dataset = datasets.ImageFolder(root=args.dataset\_path + "valid/", transform=basic\_augmentations)

# 随机选择真实和伪造的面部图像

indices = [random.randint(0, len(train\_dataset)) for i in range(16)]

# # 显示原始训练数据

# figure = plt.figure(figsize=(16, 16))

# for i in range(16):

# index = indices[i]

# img = raw\_dataset[index][0].permute(1, 2, 0)

# label = idx\_to\_class[raw\_dataset[index][1]]

# figure.add\_subplot(4, 4, i + 1)

# plt.title(label)

# plt.axis("off")

# plt.imshow(img)

# # 显示增强后的训练数据

# figure = plt.figure(figsize=(16, 16))

# for i in range(16):

# index = indices[i]

# img = train\_dataset[index][0].permute(1, 2, 0)

# label = idx\_to\_class[train\_dataset[index][1]]

# figure.add\_subplot(4, 4, i + 1)

# plt.title(label)

# plt.axis("off")

# plt.imshow(img)

# # 3. 模型

# ## 3.1 网络结构

# 加载预训练的 ViT-base-patch16 模型

model = timm.create\_model('vit\_base\_patch16\_224', pretrained=True)

# 如果有训练好的模型权重，则加载自定义权重

# state\_dict = torch.load(args.load\_model\_path)

# model.load\_state\_dict(state\_dict["model"])

# 修改最后一层，以适应分类任务

model.head = torch.nn.Linear(model.head.in\_features, args.num\_classes)

# 选择设备 (GPU 或 CPU)

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

model = model.to(device)

# ## 3.2 优化器

optimizer = optim.AdamW(model.parameters(), lr=args.learning\_rate)

if args.scheduler:

scheduler = optim.lr\_scheduler.StepLR(optimizer,

step\_size=args.sch\_step\_size,

gamma=args.sch\_gamma)

# ## 3.3 损失函数

loss\_fn = torch.nn.CrossEntropyLoss()

# # 4. 训练

# 开始训练模型并记录中间结果，包括损失、准确率、精确度、召回率和F1得分

train\_acc, test\_acc = [], []

train\_precision, test\_precision = [], []

train\_recall, test\_recall = [], []

train\_f1, test\_f1 = [], []

train\_loss, test\_loss = [], []

class LossBuffer:

def \_\_init\_\_(self, mean=0, n=0):

self.mean = mean

self.n = n

def add(self, num):

self.mean = (self.mean \* self.n + num) / (self.n + 1)

self.n += 1

def train(model, dataloader, epoch):

model.train()

correct, cursum = 0, 0

loop = tqdm(dataloader, total=len(dataloader))

y\_true, y\_pred = [], []

loss\_buffer = LossBuffer()

for idx, (data, label) in enumerate(loop):

data, label = data.to(device), label.to(device)

output = model(data)

pred = output.argmax(dim=1)

y\_true.extend(label.cpu())

y\_pred.extend(pred.cpu())

acc = accuracy\_score(y\_true, y\_pred)

optimizer.zero\_grad()

loss = loss\_fn(output, label)

loss.backward()

optimizer.step()

loss\_buffer.add(loss.item())

loop.set\_description(f"[Epoch {epoch}/{args.epochs}]")

loop.set\_postfix(LOSS="{:.6f}".format(loss\_buffer.mean), ACC="{:.2f}%".format(100 \* acc))

if args.scheduler:

scheduler.step()

precision, recall, f1, \_ = precision\_recall\_fscore\_support(y\_true, y\_pred, average='macro')

torch.save(model.state\_dict(), args.save\_model\_path)

train\_acc.append(acc)

train\_precision.append(precision)

train\_recall.append(recall)

train\_f1.append(f1)

train\_loss.append(loss\_buffer.mean)

def test(model, dataloader, epoch):

model.eval()

correct = 0

y\_true, y\_pred = [], []

with torch.no\_grad():

total\_len = len(dataloader.dataset)

loss\_buffer = LossBuffer()

for idx, (data, label) in enumerate(dataloader):

data, label = data.to(device), label.to(device)

output = model(data)

pred = output.argmax(dim=1)

y\_true.extend(label.cpu())

y\_pred.extend(pred.cpu())

loss = loss\_fn(output, label)

loss\_buffer.add(loss.item())

acc = accuracy\_score(y\_true, y\_pred)

precision, recall, f1, \_ = precision\_recall\_fscore\_support(y\_true, y\_pred, average='macro')

print("\n" + "-" \* 60)

print("[Epoch {}/{}]: Test -> LOSS: {:.6f} | Accuracy: {:.2f}%".format(epoch, args.epochs, loss\_buffer.mean, 100 \* acc))

print("-" \* 60 + "\n")

test\_acc.append(acc)

test\_precision.append(precision)

test\_recall.append(recall)

test\_f1.append(f1)

test\_loss.append(loss\_buffer.mean)

for epoch in range(1, args.epochs + 1):

train(model, train\_loader, epoch)

test(model, test\_loader, epoch)

# # 5. 可视化

# ## 5.1 打印结果

dic = {

"train\_loss" : train\_loss,

"test\_loss" : test\_loss,

"train\_acc" : train\_acc,

"test\_acc" : test\_acc,

"train\_precision": train\_precision,

"test\_precision" : test\_precision,

"train\_recall" : train\_recall,

"test\_recall" : test\_recall,

"train\_f1" : train\_f1,

"test\_f1" : test\_f1,

}

# 打印并保存结果

for key, value in dic.items():

print(key + ": " + str(value), "\n")

np.savetxt(args.output\_path + key + ".txt", value)

# ## 5.2 绘制图表

# 绘制损失图

plt.plot(np.arange(1, args.epochs + 1), np.array(train\_loss), 'go-')

plt.plot(np.arange(1, args.epochs + 1), np.array(test\_loss), 'ro-')

plt.xticks(np.arange(2, args.epochs + 1, 2))

plt.title("Loss")

plt.grid(True)

plt.legend(["train", "test"], loc="upper right")

plt.savefig("./output/Loss.png", dpi=600)

figure = plt.figure(figsize=(16,16))

# 绘制准确率图

figure.add\_subplot(2, 2, 1)

plt.plot(np.arange(1, args.epochs + 1), np.array(train\_acc), 'go-')

plt.plot(np.arange(1, args.epochs + 1), np.array(test\_acc), 'ro-')

plt.xticks(np.arange(2, args.epochs + 1, 2))

plt.title("Accuracy")

plt.grid(True)

plt.legend(["train", "test"], loc="lower right")

plt.savefig("./output/Accuracy.png", dpi=600)

# 绘制精确度图

figure.add\_subplot(2, 2, 2)

plt.plot(np.arange(1, args.epochs + 1), np.array(train\_precision), 'go-')

plt.plot(np.arange(1, args.epochs + 1), np.array(test\_precision), 'ro-')

plt.xticks(np.arange(2, args.epochs + 1, 2))

plt.title("Precision")

plt.grid(True)

plt.legend(["train", "test"], loc="lower right")

plt.savefig("./output/Precision.png", dpi=600)

# 绘制召回率图

figure.add\_subplot(2, 2, 3)

plt.plot(np.arange(1, args.epochs + 1), np.array(train\_recall), 'go-')

plt.plot(np.arange(1, args.epochs + 1), np.array(test\_recall), 'ro-')

plt.xticks(np.arange(2, args.epochs + 1, 2))

plt.title("Recall")

plt.grid(True)

plt.legend(["train", "test"], loc="lower right")

plt.savefig("./output/Recall.png", dpi=600)

# 绘制F1分数图

figure.add\_subplot(2, 2, 4)

plt.plot(np.arange(1, args.epochs + 1), np.array(train\_f1), 'go-')

plt.plot(np.arange(1, args.epochs + 1), np.array(test\_f1), 'ro-')

plt.xticks(np.arange(2, args.epochs + 1, 2))

plt.title("F1-score")

plt.grid(True)

plt.legend(["train", "test"], loc="lower right")

plt.savefig("./output/F1-score.png", dpi=600)

# 训练结束清除内存

for i in range(10):

torch.cuda.empty\_cache()