#!/usr/bin/env python

# coding: utf-8

# # 从零开始实现经典网络之—ViT图像分类任务

#

# 使用paddle复现ViT，并完成图像分类训练

# ## 1、数据准备

# In[1]:

# \*\*数据说明：\*\*

# \* fruits: banana, apple, pear, grapes, orange, kiwi, watermelon, pomegranate, pineapple, mango

# \* vegetables: cucumber, carrot, capsicum, onion, potato, lemon, tomato, raddish, beetroot, cabbage, lettuce, spinach, soy bean, cauliflower, bell pepper, chilli pepper, turnip, corn, sweetcorn, sweet potato, paprika, jalepeño, ginger, garlic, peas, eggplant

#

# 三个文件夹train、test、validation

# \* train: 每类100张图片

# \* test: 每类10张图片

# \* validation: 每类10张图片

# ## 2、数据处理

# In[3]:

# 读取数据集

from visualdl import LogWriter

import matplotlib.pyplot as plt

import paddle

import paddle.nn as nn

import os

from PIL import Image

import numpy as np

import paddle.vision.transforms as T

train\_trans = T.Compose([

    T.Resize(300),

    T.RandomResizedCrop(size=224),

    T.RandomHorizontalFlip(),

    # T.ColorJitter(0.4, 0.4, 0.4, 0.4),

    T.ToTensor()

])

test\_trans = T.Compose([

    T.Resize(300),

    T.RandomResizedCrop(size=224),

    T.ToTensor()

])

# In[4]:

# 定义数据集接口

def preprocess(img, is\_train):

    img = img.convert("RGB")

    if is\_train:

        img = train\_trans(img)

    else:

        img = test\_trans(img)

    return img

class MyDataset(paddle.io.Dataset):

    def \_\_init\_\_(self, path\_dir, root\_dir='work', train=True):

        super().\_\_init\_\_()

        self.train = train

        self.path\_dir = os.path.join(root\_dir, path\_dir)

        self.imgcls = os.listdir(self.path\_dir)

        self.img\_path = []

        for path in self.imgcls:

            img\_dir = os.path.join(self.path\_dir, path)

            img\_name = os.listdir(img\_dir)

            img\_name = [os.path.join(img\_dir, x) for x in img\_name]

            self.img\_path.extend(img\_name)

        # print(self.img\_path)

        self.label\_map = {v: k for k, v in dict(

            enumerate(self.imgcls)).items()}

        # print(self.label\_map)

    def \_\_getitem\_\_(self, idx):

        img\_path = self.img\_path[idx]

        img = Image.open(img\_path)

        img = preprocess(img, self.train)

        label = self.label\_map[img\_path.split('/')[-2]]

        label = paddle.to\_tensor(label).astype('int64')

        return img, label

    def \_\_len\_\_(self):

        return len(self.img\_path)

train\_set = MyDataset('train')

val\_set = MyDataset('validation')

test\_set = MyDataset('test', train=False)

# In[6]:

# 可视化

get\_ipython().run\_line\_magic('matplotlib', 'inline')

plt.figure(figsize=(8, 8))

for i in range(6):

    plt.subplot(2, 3, i+1)

    img, label = train\_set[i\*101]

    plt.imshow(img.transpose((1, 2, 0)))

plt.show()

# In[7]:

# 定义数据读取器

train\_loader = paddle.io.DataLoader(

    dataset=train\_set, batch\_size=32, shuffle=True)

val\_loader = paddle.io.DataLoader(dataset=val\_set, batch\_size=32, shuffle=True)

test\_loader = paddle.io.DataLoader(

    dataset=test\_set, batch\_size=32, shuffle=True)

# ## 3、模型准备

#

# ViT结构如下图所示：

# <div>

#   <img src="https://ai-studio-static-online.cdn.bcebos.com/a1a742457f284073abaeff1871502ca10ad86c9fecb6416eac50dbb46b801db1" width=700 height=500/>

#   </div>

#

# 将图像打成patch，然后每个patch作为一个token送入基本的transformer block即可。

#

# 该项目默认设置的参数是ViT-Base的参数，如果需要使用其他的版本，仅需要按照下图配置相关参数即可。

# <div>

#   <img src="https://ai-studio-static-online.cdn.bcebos.com/27d0d46b29f545768c909649cbf15ea36c7dc194709142a79d4d60564d922388" width=700 height=500/>

#   </div>

# In[8]:

# 实现Multihead self-Attention模块

class Attention(nn.Layer):

    def \_\_init\_\_(self, embed\_dim, num\_heads, qkv\_bias=False, qk\_scale=None, droupout=0.):

        super().\_\_init\_\_()

        self.embed\_dim = embed\_dim  # 每个token的维度，对应于Q的宽度

        self.num\_heads = num\_heads  # 头

        self.head\_dim = int(self.embed\_dim / num\_heads)  # 每个头需要将token投射到什么长度

        self.all\_head\_dim = self.head\_dim \* num\_heads  # 这个参数的目的只是为了方便编写待会的fc层

        self.qkv = nn.Linear(embed\_dim,

                             self.all\_head\_dim \* 3,

                             bias\_attr=False if qkv\_bias is False else None,  # bias\_attr默认为None，此时框架提供初始全零的偏置

                             )

        # 为了对计算的注意力进行缩放的根号dk，这里提供了可选比例

        self.scale = self.embed\_dim \*\* -0.5 if qk\_scale is None else qk\_scale

        self.softmax = nn.Softmax(-1)  # 沿最后一维做softmax，也就是将每个q与所有k的内积进行概率化

        self.proj = nn.Linear(self.all\_head\_dim, embed\_dim)  # 投射层

    def transpose\_multi\_head(self, x):

        # 这个函数就是对输入的所有头的x分离一下，输出每个头的 k/q/v

        # x:[batch, num\_patches, all\_head\_dim]

        new\_shape = x.shape[:-1] + [self.num\_heads, self.head\_dim]

        x = x.reshape(new\_shape)  # x:[batch, num\_patches, num\_heads, head\_dim]

        # x:[batch, num\_heads, num\_patches, head\_dim]

        x = x.transpose([0, 2, 1, 3])

        # 这个转置吧head提前，目的是保证每个头单独做自注意力计算

        return x

    def forward(self, x):

        # 输入x的形状为[batch, num\_patches, embed\_dim]

        batch, num\_patches, \_ = x.shape

        # 线性层默认只对最后一个维度进行全连接计算，经过qkv后的最后一维维度是all\_head\_dim \* 3, 切分返回列表含三个元素

        qkv = self.qkv(x).chunk(3, -1)

        # 列表每个元素分别经过前面函数句柄的处理返回三个值

        q, k, v = map(self.transpose\_multi\_head, qkv)

        # q, k, v : [batch, num\_heads, num\_patches, head\_dim]

        attn = paddle.matmul(q, k, transpose\_y=True)  # 计算Q \* K'

        attn = self.softmax(attn \* self.scale)  # 计算softmax

        # attn : [batch, num\_heads, num\_patches, num\_patches]

        out = paddle.matmul(attn, v)  # attn为n\*n, v为n\*dk, 直接做矩阵乘法

        # out:[batch, num\_heads, num\_patches, embed\_dim]

        out = out.transpose([0, 2, 1, 3])

        # out:[batch, num\_patches, num\_heads, embed\_dim]

        # 多头结果合并

        # 输出x的形状为[batch, num\_patches, embed\_dim]

        out = out.reshape([batch, num\_patches, -1])

        # 投射到原始维度，接下来准备进行MLP操作

        out = self.proj(out)

        return out

# In[11]:

# 测试注意力模块:输入和输出形状一样

MSA = Attention(embed\_dim=96, num\_heads=12)

x = paddle.randn([8, 16, 96])

out = MSA(x)

print(out.shape)

# \*\*小插曲：\*\*

# - 纠正一个错误：很多地方都说LayerNorm是对每一句话进行归一化，但是源码中这一层就传入了一个embed\_dim，所以实际上是对每一个token进行的归一化

# - 测试结果如下

# In[12]:

# 测试LayerNorm

np.random.seed(123)

x\_data = np.random.random(size=(2, 2, 3)).astype('float32')

x = paddle.to\_tensor(x\_data)

print("x:", x)

layer\_norm\_1 = paddle.nn.LayerNorm(3)

layer\_norm\_2 = paddle.nn.LayerNorm((2, 3))

layer\_norm\_out\_1 = layer\_norm\_1(x)

layer\_norm\_out\_2 = layer\_norm\_2(x)

print("layer\_norm\_out\_1:", layer\_norm\_out\_1)

print("layer\_norm\_out\_2:", layer\_norm\_out\_2)

# 取出一维我们来自己计算一下

x = np.array([0.69646919, 0.28613934, 0.22685145])

t = np.mean(x)

v = np.std(x)

x = (x-t) / v

print(x)

y = [[0.69646919, 0.28613934, 0.22685145],

     [0.55131477, 0.71946895, 0.42310646]]

t = np.mean(y)

v = np.std(y)

y = (y-t) / v

print(y)

# In[14]:

# 实现EncoderLayer

class EncoderLayer(nn.Layer):

    def \_\_init\_\_(self, embed\_dim=768, num\_heads=12, mlp\_ratio=4.0):

        super().\_\_init\_\_()

        self.attn\_norm = nn.LayerNorm(embed\_dim)

        self.attn = Attention(embed\_dim, num\_heads)

        self.mlp\_norm = nn.LayerNorm(embed\_dim)

        self.mlp = Mlp(embed\_dim, mlp\_ratio)

    def forward(self, x):

        # 采用prenorm，也就是说layernorm层在MSA和MLP之前实现

        # 下面就是一个encoder的逻辑

        h = x

        x = self.attn\_norm(x)

        x = self.attn(x)

        x = x + h

        h = x

        x = self.mlp\_norm(x)

        x = self.mlp(x)

        x = x + h

        return x

# 实现Encoder

class Encoder(nn.Layer):

    def \_\_init\_\_(self, embed\_dim, depth):

        super().\_\_init\_\_()

        layer\_list = []

        # depth就是堆叠多少个encoder

        for i in range(depth):

            encoder\_layer = EncoderLayer()

            layer\_list.append(encoder\_layer)

        self.layers = nn.LayerList(layer\_list)

        self.norm = nn.LayerNorm(embed\_dim)

    def forward(self, x):

        for layer in self.layers:

            x = layer(x)

        x = self.norm(x)

        return x

# In[13]:

# 实现Positionembedding

class PatchEmbedding(nn.Layer):

    def \_\_init\_\_(self, image\_size=224, patch\_size=16, in\_channels=3, embed\_dim=768, dropout=0.):

        super().\_\_init\_\_()

        # 计算patches的数量，默认是14x14

        num\_patches = (image\_size // patch\_size) \* (image\_size // patch\_size)

        self.patch\_embedding = nn.Conv2D(

            in\_channels=in\_channels, out\_channels=embed\_dim, kernel\_size=patch\_size, stride=patch\_size)

        self.dropout = nn.Dropout(dropout)

        self.embed\_dim = embed\_dim

        # 添加cls\_token

        self.class\_token = paddle.create\_parameter(

            shape=[1, 1, embed\_dim],

            dtype='float32',

            default\_initializer=nn.initializer.Constant(0.))

        # 添加位置编码

        self.position\_embedding = paddle.create\_parameter(

            shape=[1, num\_patches + 1, embed\_dim],

            dtype='float32',

            default\_initializer=nn.initializer.TruncatedNormal(std=(0.2)))

    def forward(self, x):

        # x:[N,C,H,W]

        cls\_token = self.class\_token.expand([x.shape[0], 1, self.embed\_dim])

        x = self.patch\_embedding(x)  # x:[N, embed\_dim, h', w']

        x = x.flatten(2)

        x = x.transpose([0, 2, 1])  # x:[N, h'\*w', embed\_dim]

        x = paddle.concat([cls\_token, x], axis=1)

        x = x + self.position\_embedding

        return x

# In[15]:

# 实现MLP模块

class Mlp(nn.Layer):

    def \_\_init\_\_(self, embed\_dim, mlp\_ratio, dropout=0.):

        super().\_\_init\_\_()

        self.fc1 = nn.Linear(embed\_dim, int(embed\_dim\*mlp\_ratio))

        self.fc2 = nn.Linear(int(embed\_dim\*mlp\_ratio), embed\_dim)

        self.act = nn.GELU()

        self.dropout = nn.Dropout(dropout)

    def forward(self, x):

        x = self.fc1(x)

        x = self.act(x)

        x = self.dropout(x)

        x = self.fc2(x)

        x = self.dropout(x)

        return x

# In[16]:

# 实现ViT

class ViT(nn.Layer):

    def \_\_init\_\_(self,

                 image\_size=224,  # 图片大小

                 patch\_size=16,  # 图片的patches数量

                 in\_channels=3,  # 原始图片通道

                 num\_classes=1000,  # 目标类别

                 embed\_dim=768,  # 编码长度

                 depth=3,  # encoder堆叠的数量

                 num\_heads=12,  # 头

                 mlp\_ratio=4,  # mlp维度缩放比例

                 qkv\_bias=True,  # qkv投射层的偏置

                 dropout=0.,  # dropout

                 attention\_dropout=0.,  # attn的dropout，其实就是qkv投射层之后是否加入dropout层

                 droppath=0.):

        super().\_\_init\_\_()

        # 将图像使用一个卷积编码为patches，然后加入clstoken和位置编码

        self.patch\_embedding = PatchEmbedding(

            image\_size, patch\_size, in\_channels, embed\_dim)

        # 这个堆叠了depth个encoder的编码器，输入等于输出，最后加一个norm

        self.encoder = Encoder(embed\_dim, depth)

        self.classfier = nn.Linear(embed\_dim, num\_classes)  # 分类头

    def forward(self, x):

        x = self.patch\_embedding(x)  # x:[N, num\_patches, embed\_dim]

        x = self.encoder(x)  # x:[N, num\_patches, embed\_dim]

        x = self.classfier(x[:, 0])  # 相当于只对所有的cls\_token做映射

        return x

# In[17]:

# 测试

vit = ViT()

paddle.summary(vit, (4, 3, 224, 224))

# In[18]:

# 模型准备

EPOCH = 100

lr = 8e-3

lr\_schedual = paddle.optimizer.lr.CosineAnnealingDecay(

    learning\_rate=lr, T\_max=EPOCH, verbose=True)

loss\_fn = nn.CrossEntropyLoss()

metric = paddle.metric.Accuracy()

vit = ViT(num\_classes=36)

opt = paddle.optimizer.Adam(parameters=vit.parameters(

), learning\_rate=lr\_schedual, beta1=0.9, beta2=0.999)

# ## 4、模型训练

# In[17]:

# 模型训练

writer = LogWriter('work/logs')

total\_train\_step = 0

for epoch in range(EPOCH):

    vit.train()

    for batch\_id, data in enumerate(train\_loader):

        img = data[0]

        label = data[1]

        # print(img.shape, label.shape)

        # y = paddle.argmax(vit(img), axis=1)

        # y = y.reshape(shape=[y.shape[0], -1])

        y = vit(img)

        loss = loss\_fn(y, label)

        corr = metric.compute(y, label)

        acc = metric.update(corr)

        # 更新

        opt.clear\_grad()

        loss.backward()

        opt.step()

        total\_train\_step += 1

        if batch\_id % 10 == 0:

            print('epoch:{}, batch\_id:{}, loss:{}, acc:{}'.format(

                epoch, batch\_id, loss.item(), acc.item()))

            writer.add\_scalar(tag="train\_loss",

                              value=loss.item(), step=total\_train\_step)

    # 模型验证

    vit.eval()

    with paddle.no\_grad():

        for batch\_id, data in enumerate(val\_loader):

            img = data[0]

            label = data[1]

            y = vit(img)

            loss = loss\_fn(y, label)

            corr = metric.compute(y, label)

            acc = metric.update(corr)

            if batch\_id % 10 == 0:

                print('----- val: ----- batch\_id:{}, loss:{}, acc:{}'.format(batch\_id,

                      loss.item(), acc.item()))

                writer.add\_scalar(

                    tag="val\_loss", value=loss.item(), step=total\_train\_step)

    lr\_schedual.step()

    if epoch % 20 == 0:

        paddle.save(vit.state\_dict(), "./output/vit\_{}.pdparams".format(epoch))

writer.close()