#### 我打算在VGG16模型的特定层之间添加自注意力模块，以使模型能够自动学习图像中的关键区域。这样做需要对VGG16模型的结构进行修改，并添加自注意力模块的定义和连接，使用注意力机制替代VGG16模型中的卷积层。

import torch

import torch.nn as nn

# 定义自注意力层

class SelfAttention(nn.Module):

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

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

self.query\_conv = nn.Conv2d(in\_channels, in\_channels // 8, kernel\_size=1)

self.key\_conv = nn.Conv2d(in\_channels, in\_channels // 8, kernel\_size=1)

self.value\_conv = nn.Conv2d(in\_channels, in\_channels, kernel\_size=1)

self.gamma = nn.Parameter(torch.zeros(1))

def forward(self, x):

batch\_size, channels, height, width = x.size()

query = self.query\_conv(x).view(batch\_size, -1, height \* width).permute(0, 2, 1)

key = self.key\_conv(x).view(batch\_size, -1, height \* width)

energy = torch.bmm(query, key)

attention = torch.softmax(energy, dim=-1)

value = self.value\_conv(x).view(batch\_size, -1, height \* width)

out = torch.bmm(value, attention.permute(0, 2, 1))

out = out.view(batch\_size, channels, height, width)

out = self.gamma \* out + x

return out

# 定义VGG16模型

class VGG16Attention(nn.Module):

def \_\_init\_\_(self, num\_classes=1000):

super(VGG16Attention, self).\_\_init\_\_()

self.features = nn.Sequential(

nn.Conv2d(3, 64, kernel\_size=3, padding=1),

nn.ReLU(inplace=True),

SelfAttention(64),

nn.Conv2d(64, 64, kernel\_size=3, padding=1),

nn.ReLU(inplace=True),

nn.MaxPool2d(kernel\_size=2, stride=2),

nn.Conv2d(64, 128, kernel\_size=3, padding=1),

nn.ReLU(inplace=True),

SelfAttention(128),

nn.Conv2d(128, 128, kernel\_size=3, padding=1),

nn.ReLU(inplace=True),

nn.MaxPool2d(kernel\_size=2, stride=2),

nn.Conv2d(128, 256, kernel\_size=3, padding=1),

nn.ReLU(inplace=True),

nn.Conv2d(256, 256, kernel\_size=3, padding=1),

nn.ReLU(inplace=True),

nn.Conv2d(256, 256, kernel\_size=3, padding=1),

nn.ReLU(inplace=True),

nn.MaxPool2d(kernel\_size=2, stride=2),

nn.Conv2d(256, 512, kernel\_size=3, padding=1),

nn.ReLU(inplace=True),

nn.Conv2d(512, 512, kernel\_size=3, padding=1),

nn.ReLU(inplace=True),

nn.Conv2d(512, 512, kernel\_size=3, padding=1),

nn.ReLU(inplace=True),

nn.MaxPool2d(kernel\_size=2, stride=2),

nn.Conv2d(512, 512, kernel\_size=3, padding=1),

nn.ReLU(inplace=True),

nn.Conv2d(512, 512, kernel\_size=3, padding=1),

nn.ReLU(inplace=True),

nn.Conv2d(512, 512, kernel\_size=3, padding=1),

nn.ReLU(inplace=True),

nn.MaxPool2d(kernel\_size=2, stride=2)

)

self.classifier = nn.Sequential(

nn.Linear(512 \* 7 \* 7, 4096),

nn.ReLU(inplace=True),

nn.Dropout(),

nn.Linear(4096, 4096),

nn.ReLU(inplace=True),

nn.Dropout(),

nn.Linear(4096, num\_classes)

)

def forward(self, x):

x = self.features(x)

x = torch.flatten(x, start\_dim=1)

x = self.classifier(x)

return x

# 创建VGG16Attention模型实例

model = VGG16Attention()

# 打印模型结构

print(model)

##### 它的结构图大概如下：

​

Input (224x224x3)

Convolutional Layer: 64 filters, 3x3 kernel, padding=1

ReLU Activation

Self-Attention Layer

Convolutional Layer: 64 filters, 3x3 kernel, padding=1

ReLU Activation

Max Pooling Layer: 2x2 pool size, stride=2

Convolutional Layer: 128 filters, 3x3 kernel, padding=1

ReLU Activation

Self-Attention Layer

Convolutional Layer: 128 filters, 3x3 kernel, padding=1

ReLU Activation

Max Pooling Layer: 2x2 pool size, stride=2

Convolutional Layer: 256 filters, 3x3 kernel, padding=1

ReLU Activation

Convolutional Layer: 256 filters, 3x3 kernel, padding=1

ReLU Activation

Convolutional Layer: 256 filters, 3x3 kernel, padding=1

ReLU Activation

Max Pooling Layer: 2x2 pool size, stride=2

Convolutional Layer: 512 filters, 3x3 kernel, padding=1

ReLU Activation

Convolutional Layer: 512 filters, 3x3 kernel, padding=1

ReLU Activation

Convolutional Layer: 512 filters, 3x3 kernel, padding=1

ReLU Activation

Max Pooling Layer: 2x2 pool size, stride=2

Convolutional Layer: 512 filters, 3x3 kernel, padding=1

ReLU Activation

Convolutional Layer: 512 filters, 3x3 kernel, padding=1

ReLU Activation

Convolutional Layer: 512 filters, 3x3 kernel, padding=1

ReLU Activation

Max Pooling Layer: 2x2 pool size, stride=2

Flatten

Fully Connected Layer: 4096 units

ReLU Activation

Dropout

Fully Connected Layer: 4096 units

ReLU Activation

Dropout

Fully Connected Layer: num\_classes units

Output

##### ​接着我们用添加注意力机制的VGG16做图像分类训练：

​

import torch.optim as optim

import torch

import torch.nn as nn

import torch.nn.parallel

import torch.utils.data

import torch.utils.data.distributed

import torchvision.transforms as transforms

from dataset.dataset import SeedlingData

from torch.autograd import Variable

from torchvision.models import vgg16

# 设置全局参数

modellr = 1e-4

BATCH\_SIZE = 32

EPOCHS = 100

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

# 数据预处理

transform = transforms.Compose([

transforms.Resize((224, 224)),

transforms.ToTensor(),

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

])

transform\_test = transforms.Compose([

transforms.Resize((224, 224)),

transforms.ToTensor(),

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

])

dataset\_train = SeedlingData('data/train', transforms=transform, train=True)

dataset\_test = SeedlingData("data/train", transforms=transform\_test, train=False)

# 读取数据

print(dataset\_train.imgs)

# 导入数据

train\_loader = torch.utils.data.DataLoader(dataset\_train, batch\_size=BATCH\_SIZE, shuffle=True)

test\_loader = torch.utils.data.DataLoader(dataset\_test, batch\_size=BATCH\_SIZE, shuffle=False)

# 实例化模型并且移动到GPU

criterion = nn.CrossEntropyLoss()

model\_ft = VGG16Attention()

model\_ft.classifier = classifier = nn.Sequential(

nn.Linear(512 \* 7 \* 7, 4096),

nn.ReLU(True),

nn.Dropout(),

nn.Linear(4096, 4096),

nn.ReLU(True),

nn.Dropout(),

nn.Linear(4096, 3),

)

model\_ft.to(DEVICE)

optimizer = optim.Adam(model\_ft.parameters(), lr=modellr)

def adjust\_learning\_rate(optimizer, epoch):

"""Sets the learning rate to the initial LR decayed by 10 every 30 epochs"""

modellrnew = modellr \* (0.1 \*\* (epoch // 50))

print("lr:", modellrnew)

for param\_group in optimizer.param\_groups:

param\_group['lr'] = modellrnew

# 定义训练过程

def train(model, device, train\_loader, optimizer, epoch):

model.train()

sum\_loss = 0

total\_num = len(train\_loader.dataset)

print(total\_num, len(train\_loader))

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

data, target = Variable(data).to(device), Variable(target).to(device)

output = model(data)

loss = criterion(output, target)

optimizer.zero\_grad()

loss.backward()

optimizer.step()

print\_loss = loss.data.item()

sum\_loss += print\_loss

if (batch\_idx + 1) % 10 == 0:

print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(

epoch, (batch\_idx + 1) \* len(data), len(train\_loader.dataset),

100. \* (batch\_idx + 1) / len(train\_loader), loss.item()))

ave\_loss = sum\_loss / len(train\_loader)

print('epoch:{},loss:{}'.format(epoch, ave\_loss))

# 验证过程

def val(model, device, test\_loader):

model.eval()

test\_loss = 0

correct = 0

total\_num = len(test\_loader.dataset)

print(total\_num, len(test\_loader))

with torch.no\_grad():

for data, target in test\_loader:

data, target = Variable(data).to(device), Variable(target).to(device)

output = model(data)

loss = criterion(output, target)

\_, pred = torch.max(output.data, 1)

correct += torch.sum(pred == target)

print\_loss = loss.data.item()

test\_loss += print\_loss

correct = correct.data.item()

acc = correct / total\_num

avgloss = test\_loss / len(test\_loader)

print('\nVal set: Average loss: {:.4f}, Accuracy: {}/{} ({:.0f}%)\n'.format(

avgloss, correct, len(test\_loader.dataset), 100 \* acc))

# 训练

for epoch in range(1, EPOCHS + 1):

adjust\_learning\_rate(optimizer, epoch)

train(model\_ft, DEVICE, train\_loader, optimizer, epoch)

val(model\_ft, DEVICE, test\_loader)

torch.save(model\_ft, 'model.pth')

​