ResNet与Renxet

笔记本:	深度学习-图像分类篇		
创建时间:	2023/4/5 21:46	**更新时间:**	2023/4/7 22:30
作者:	Kaiyuecui		
URL:	https://zhuanlan.zhihu.com/p/93643523		

1、Resnet创新点

①使网络有了更深的层数
②使用了bn加速训练（丢弃 了dropout）
③使用了residual block残差块

ResNext

更新了block模块

stage	output	ResNet-50	ResNeXt-50 (32×4d)
conv1	112×112	7×7, 64, stride 2	7×7, 64, stride 2
conv2	56×56	3×3 max pool, stride 2 $\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	3×3 max pool, stride 2 $\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128, C{=}32 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3	28×28	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256, C{=}32 \\ 1\times1, 512 \end{bmatrix}\times4$
conv4	14×14	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512, C{=}32 \\ 1\times1, 1024 \end{bmatrix}\times6$
conv5	7×7	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 1024 \\ 3\times3, 1024, C{=}32 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	global average pool 1000-d fc, softmax	global average pool 1000-d fc, softmax
# params.		**25.5**$\times10^6$	**25.0**$\times10^6$
FLOPs		**4.1**$\times10^9$	**4.2**$\times10^9$

2、bn

①bn简介

Batch Normalization是2015年一篇论文中提出的数据归一化方法，往往用在深度神经网络中激活层之前。其作用可以加快模型训练时的收敛速度，使得模型训练过程更加稳定，避免梯度爆炸或者梯度消失。并且起到一定的正则化作用，几乎代替了Dropout。正则化可以防止过拟合

①bn的基础公式

$$Input: B=\{x_{1...m}\}; \gamma, \beta (parameters\ to\ be\ learned)$$
$$Output: \{y_i = BN_{\gamma,\beta}(x_i)\}$$
$$\mu_B \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$$
$$\sigma_B^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i-\mu_B)^2$$
$$\tilde{x}_i \leftarrow \frac{x_i-\mu_B}{\sqrt{\sigma_B^2+\epsilon}}$$
$$y_i \leftarrow \gamma\tilde{x}_i + \beta$$

解释一下上述公式：

1. 输入为数值集合（B），可训练参数 γ、β；
2. BN的具体操作为：先计算 B 的均值和方差，之后将 B 集合的均值、方差变换为0、1（对应上式中 $\tilde{x}_i \leftarrow \frac{x_i-\mu_B}{\sqrt{\sigma_B^2+\epsilon}}$），最后将 B 中每个元素乘以 γ 再加 β，输出。γ、β 是可训练参数，参与整个网络的BP；
3. 归一化的目的：将数据规整到统一区间，减少数据的发散程度，降低网络的学习难度。BN的精髓在于归一之后，使用 γ、β 作为还原参数，在一定程度上保留原数据的分布。

③训练与推理时BN中的均值、方差分别是什么?

此问题是BN争议最大之处，正确答案是：

训练时，均值、方差分别是**该批次**内数据相应维度的均值与方差；

推理时，均值、方差是**基于所有批次**的期望计算所得，

④需要注意的点：

（1）bn用在卷积和激活函数之间，而不是激活函数之后。

（2）用bn时，batch-size最好大一些，因为batch-size越大，其均值和方差就越接近整体样本的均值和方差。

（3）使用bn其卷积层没必要使用bias，使用偏置和不使用偏置最后得到的结果是一致的

3、.eval()与.train（）不同

model.train()和model.eval()的区别主要在于Batch Normalization和Dropout两层。
①在train模式下，Dropout与BN都是起作用的，但是在eval()模式下，其都不起作用。
②如果模型中有BN层(Batch Normalization）和Dropout，在测试时添加model.eval()。model.eval()是保证BN层能够用全部训练数据的均值和方差，即测试过程中要保证BN层的均值和方差不变。对于Dropout，model.eval()是利用到了所有网络连接，即不进行随机舍弃神经元。

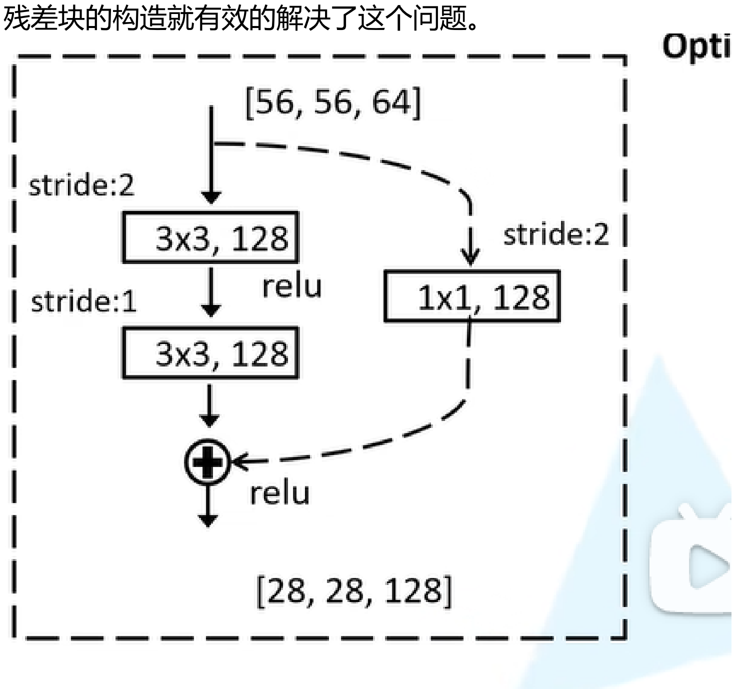
3、迁移学习

比如都是图像问题，我们已经训练好了一个模型，使其能够给不同的图像划分为1000个类别中的一种，但是目前我们又有了一个新的分类问题，这次是五分类问题，且我们这次的样本数据集比较小，没办法支撑来训练一个模型。此时我们就可以借助之前的模型参数来训练一个模型。
迁移学习的优点：①能够快速训练出一个模型②当我们的数据集较小时，也能较好的训练出一个模型。
比如图像问题，其前几层都是类似的，都是特征提取层。
当我们有了一个较好的模型参数时
我们可以选择以下做法：
①载入已有的模型参数，并根据我们的训练集重新去训练所有的参数。
②载入已有的模型参数，只训练后几层模型参数。
③载入已有的模型参数，在最后增加一层，只训练一层
注意：
迁移学习时，我们载入别人训练好的参数，一定要注意别人的预处理，我们的预处理要和别人模型的预处理一致，不然效果会不好。

4、residual block

一般来说，我们预想的情况是，网络层数越多，我们可以学到的信息也越多，待网络达到一定数量时，后边的网络可以是一个恒等学习的状态。但是现实情况并非如此，随着网络层数过多，我们的网络不仅训练慢，而且也很容易出现梯度消失或者梯度爆炸的问题，这就导致深层的网络结构还没有浅层的网络结构训练出来的模型好。

残差块的构造就有效的解决了这个问题。



主意：主分支与shortcut的输出特征矩阵shape必须相同

注意：Inception模块中是长和宽必须相同但是维度可以不同，将维度拼接在一起。但是残差块中的长和宽以及维度都必须要相同，之后相加在一起。

5、transform中的ReSIze()函数：

如果传入的是一个整数n，则会将图片的最短边裁剪到n，对应的长边也会等比例缩放。
比如h>w,w缩放到size，则h 变会缩放到 size*h/w。
如果传入的是一个元组（h,w）则图片将会缩放到我们所输入的大小。

6、迁移学习载入预训练参数方法

如果我们的分类任务是五分类，但是预训练模型是1000分类。那么可以按照下边方式加载与训练参数。

①第一种迁移学习 载入模型参数方式

```
# load pretrain weights
# download url: https://download.pytorch.org/models/resnet34-333f7ec4.pth
model_weight_path = "./resnet34-pre.pth"
assert os.path.exists(model_weight_path), "file {} does not
exist.".format(model_weight_path)
首先由于原始模型的分类是1000，所以我们如果想要将参数载入，我们就不能去指定分类个数
将参数载入到我们的模型之后，我们可以再根据我们的分类个数来修改。
net = resnet34()
载入
net.load_state_dict(torch.load(model_weight_path, map_location=device))
#change fc layer structure
in_channel = net.fc.in_features   #fc的输入层个数
net.fc = nn.Linear(in_channel, 5)
#这里是迁移学习的第一种参数方式，根据已有的参数去重新训练所有参数
#第二种方式，我们可以先指定我们的分类个数
    #之后将参数加载到内存中，但是此时还没有记载到模型中
    #将全连接层的参数删除掉，之后再将模型参数加载到模型中
```

②第一种迁移学习 载入模型参数方式

```
# option2
    net = resnet34(num_classes=5)
    #先加载到内存中
    pre_weights = torch.load(model_weight_path, map_location=device)
    del_key = []
    #找到fc层的参数
    for key, _ in pre_weights.items():
```

```
        if "fc" in key:
            del_key.append(key)
    #删除掉
    for key in del_key:
        del pre_weights[key]
#strict如果为False,则证明我们不需要完全匹配，预训练模型有什么可以匹配上我们就拿什么，预训练模型没有的就会传入missing_key
    #预训练模型有，但是我们不需要的就会传入unexpected_keys
加载到模型中
    missing_keys, unexpected_keys = net.load_state_dict(pre_weights, strict=False)
    print("[missing_keys]:", *missing_keys, sep="\n")
    #因为我们将fc层删除了 就会输出
    # [missing_keys]:
    # fc.weight
    # fc.bias
    #因为 train中有的我们都有 所以为空
    # [unexpected_keys]:
    print("[unexpected_keys]:", *unexpected_keys, sep="\n")
```

7、冻结特征层参数，只更新全连接层

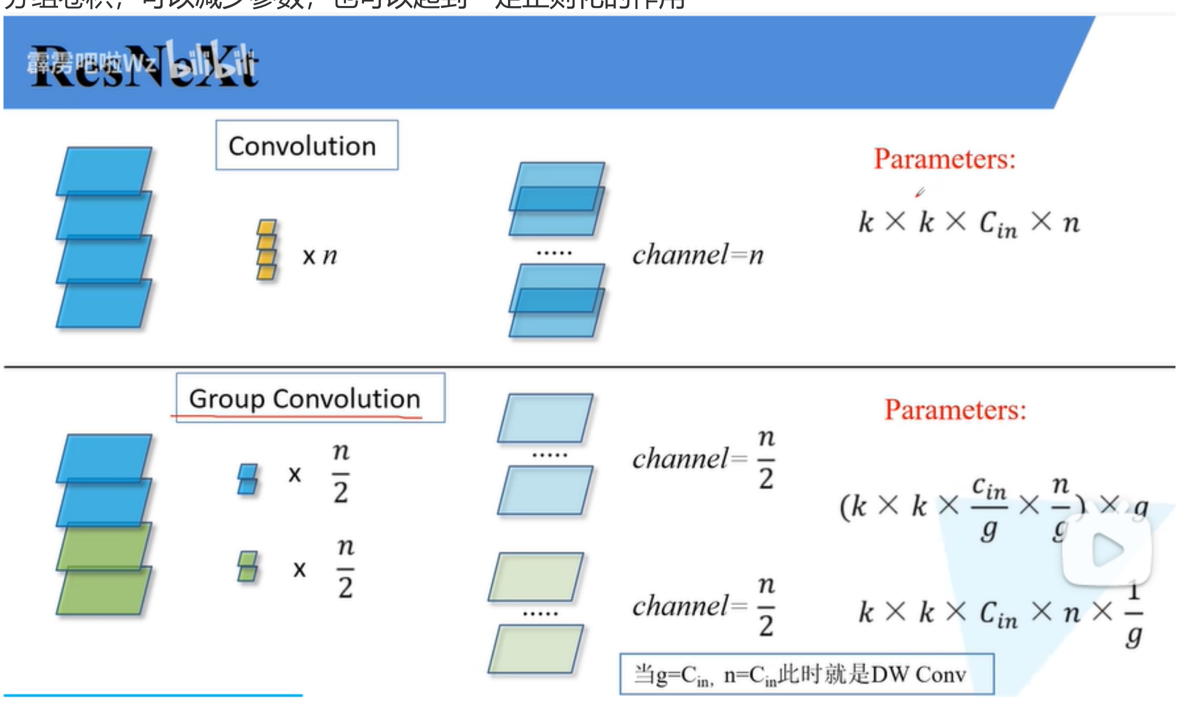
```
# for param in net.parameters():
#     param.requires_grad = False
首先这句代码的作用是：特征层中参数都固定住，不会发生梯度的更新；即它不需要计算梯度，会减少计算量。节省内存。

optimizer=optim.SGD(vgg.classifier.paramters(),lr=0.001)

这句代码的作用是定义一个优化器，这个优化器的作用是优化全连接层中的参数，并没有说要优化特征层中的参数。
```

8、conv2d中的groups参数

分组卷积，可以减少参数，也可以起到一定正则化的作用



https://blog.csdn.net/weixin_43135178/article/details/122426414

9、代码：

①model

```
import torch.nn as nn
import torch
```

```
class BasicBlock(nn.Module):
    expansion = 1


    def __init__(self, in_channel, out_channel, stride=1, downsample=None,
**kwargs):
        super(BasicBlock, self).__init__()
        self.conv1 = nn.Conv2d(in_channels=in_channel, out_channels=out_channel,
                               kernel_size=3, stride=stride, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(out_channel)
        self.relu = nn.ReLU()
        self.conv2 = nn.Conv2d(in_channels=out_channel, out_channels=out_channel,
                               kernel_size=3, stride=1, padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(out_channel)
        self.downsample = downsample


    def forward(self, x):
        identity = x
        if self.downsample is not None:
            identity = self.downsample(x)


        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)


        out = self.conv2(out)
        out = self.bn2(out)


        out += identity
        out = self.relu(out)


        return out




class Bottleneck(nn.Module):
    """
    注意：原论文中，在虚线残差结构的主分支上，第一个1x1卷积层的步距是2，第二个3x3卷积层步
距是1。
    但在pytorch官方实现过程中是第一个1x1卷积层的步距是1，第二个3x3卷积层步距是2，
    这么做的好处是能够在top1上提升大概0.5%的准确率。
    可参考Resnet v1.5 https://ngc.nvidia.com/catalog/model-
scripts/nvidia:resnet_50_v1_5_for_pytorch
    """
    expansion = 4


    def __init__(self, in_channel, out_channel, stride=1, downsample=None,
                 groups=1, width_per_group=64):
        super(Bottleneck, self).__init__()


        width = int(out_channel * (width_per_group / 64.)) * groups


        self.conv1 = nn.Conv2d(in_channels=in_channel, out_channels=width,
```

```
                               kernel_size=1, stride=1, bias=False)  # squeeze
channels
        self.bn1 = nn.BatchNorm2d(width)
        # -----------------------------------------
        self.conv2 = nn.Conv2d(in_channels=width, out_channels=width,
groups=groups,
                               kernel_size=3, stride=stride, bias=False, padding=1)
        self.bn2 = nn.BatchNorm2d(width)
        # -----------------------------------------
        self.conv3 = nn.Conv2d(in_channels=width,
out_channels=out_channel*self.expansion,
                               kernel_size=1, stride=1, bias=False)  # unsqueeze
channels
        self.bn3 = nn.BatchNorm2d(out_channel*self.expansion)
        self.relu = nn.ReLU(inplace=True)
        self.downsample = downsample


    def forward(self, x):
        identity = x
        if self.downsample is not None:
            identity = self.downsample(x)


        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)


        out = self.conv2(out)
        out = self.bn2(out)
        out = self.relu(out)


        out = self.conv3(out)
        out = self.bn3(out)


        out += identity
        out = self.relu(out)


        return out




class ResNet(nn.Module):


    def __init__(self,
                 block,
                 blocks_num,
                 num_classes=1000,
                 include_top=True,
                 groups=1,
                 width_per_group=64):
        super(ResNet, self).__init__()
        self.include_top = include_top
        self.in_channel = 64


        self.groups = groups
        self.width_per_group = width_per_group


        self.conv1 = nn.Conv2d(3, self.in_channel, kernel_size=7, stride=2,
                               padding=3, bias=False)
```

```
        self.bn1 = nn.BatchNorm2d(self.in_channel)
        self.relu = nn.ReLU(inplace=True)
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        self.layer1 = self._make_layer(block, 64, blocks_num[0])
        self.layer2 = self._make_layer(block, 128, blocks_num[1], stride=2)
        self.layer3 = self._make_layer(block, 256, blocks_num[2], stride=2)
        self.layer4 = self._make_layer(block, 512, blocks_num[3], stride=2)
        if self.include_top:
            self.avgpool = nn.AdaptiveAvgPool2d((1, 1))  # output size = (1, 1)
            self.fc = nn.Linear(512 * block.expansion, num_classes)


        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                nn.init.kaiming_normal_(m.weight, mode='fan_out',
nonlinearity='relu')


    def _make_layer(self, block, channel, block_num, stride=1):
        downsample = None
        if stride != 1 or self.in_channel != channel * block.expansion:
            downsample = nn.Sequential(
                nn.Conv2d(self.in_channel, channel * block.expansion,
kernel_size=1, stride=stride, bias=False),
                nn.BatchNorm2d(channel * block.expansion))


        layers = []
        layers.append(block(self.in_channel,
                            channel,
                            downsample=downsample,
                            stride=stride,
                            groups=self.groups,
                            width_per_group=self.width_per_group))
        self.in_channel = channel * block.expansion


        for _ in range(1, block_num):
            layers.append(block(self.in_channel,
                                channel,
                                groups=self.groups,
                                width_per_group=self.width_per_group))


        return nn.Sequential(*layers)


    def forward(self, x):
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu(x)
        x = self.maxpool(x)


        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)


        if self.include_top:
            x = self.avgpool(x)
            x = torch.flatten(x, 1)
            x = self.fc(x)


        return x
```

```
def resnet34(num_classes=1000, include_top=True):
    # https://download.pytorch.org/models/resnet34-333f7ec4.pth
    return ResNet(BasicBlock, [3, 4, 6, 3], num_classes=num_classes,
include_top=include_top)



def resnet50(num_classes=1000, include_top=True):
    # https://download.pytorch.org/models/resnet50-19c8e357.pth
    return ResNet(Bottleneck, [3, 4, 6, 3], num_classes=num_classes,
include_top=include_top)



def resnet101(num_classes=1000, include_top=True):
    # https://download.pytorch.org/models/resnet101-5d3b4d8f.pth
    return ResNet(Bottleneck, [3, 4, 23, 3], num_classes=num_classes,
include_top=include_top)



def resnext50_32x4d(num_classes=1000, include_top=True):
    # https://download.pytorch.org/models/resnext50_32x4d-7cdf4587.pth
    groups = 32
    width_per_group = 4
    return ResNet(Bottleneck, [3, 4, 6, 3],
                  num_classes=num_classes,
                  include_top=include_top,
                  groups=groups,
                  width_per_group=width_per_group)



def resnext101_32x8d(num_classes=1000, include_top=True):
    # https://download.pytorch.org/models/resnext101_32x8d-8ba56ff5.pth
    groups = 32
    width_per_group = 8
    return ResNet(Bottleneck, [3, 4, 23, 3],
                  num_classes=num_classes,
                  include_top=include_top,
                  groups=groups,
                  width_per_group=width_per_group)
```

②train

```
import os
import sys
import json


import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import transforms, datasets
from tqdm import tqdm


from model import resnet34
```

```
def main():
    device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
    print("using {} device.".format(device))


    data_transform = {
        "train": transforms.Compose([transforms.RandomResizedCrop(224),
                                     transforms.RandomHorizontalFlip(),
                                     transforms.ToTensor(),
                                     transforms.Normalize([0.485, 0.456, 0.406],
[0.229, 0.224, 0.225])]),
        "val": transforms.Compose([transforms.Resize(256),
                                   transforms.CenterCrop(224),
                                   transforms.ToTensor(),
                                   transforms.Normalize([0.485, 0.456, 0.406],
[0.229, 0.224, 0.225])])}


    data_root = os.path.abspath(os.path.join(os.getcwd(), "../.."))  # get data
root path
    image_path = os.path.join(data_root, "data_set", "flower_data")  # flower data
set path
    assert os.path.exists(image_path), "{} path does not exist.".format(image_path)
    train_dataset = datasets.ImageFolder(root=os.path.join(image_path, "train"),
                                         transform=data_transform["train"])
    train_num = len(train_dataset)


    # {'daisy':0, 'dandelion':1, 'roses':2, 'sunflower':3, 'tulips':4}
    flower_list = train_dataset.class_to_idx
    cla_dict = dict((val, key) for key, val in flower_list.items())
    # write dict into json file
    json_str = json.dumps(cla_dict, indent=4)
    with open('class_indices.json', 'w') as json_file:
        json_file.write(json_str)


    batch_size = 16
    nw =0  # number of workers
    print('Using {} dataloader workers every process'.format(nw))


    train_loader = torch.utils.data.DataLoader(train_dataset,
                                               batch_size=batch_size, shuffle=True,
                                               num_workers=nw)


    validate_dataset = datasets.ImageFolder(root=os.path.join(image_path, "val"),
                                            transform=data_transform["val"])
    val_num = len(validate_dataset)
    validate_loader = torch.utils.data.DataLoader(validate_dataset,
                                                  batch_size=batch_size,
shuffle=False,
                                                  num_workers=nw)


    print("using {} images for training, {} images for
validation.".format(train_num,
                                                                           val_num))

    net = resnet34()
    # load pretrain weights
    # download url: https://download.pytorch.org/models/resnet34-333f7ec4.pth
    model_weight_path = "./resnet34-pre.pth"
    assert os.path.exists(model_weight_path), "file {} does not
exist.".format(model_weight_path)
```

```
net.load_state_dict(torch.load(model_weight_path, map_location='cpu'))
# for param in net.parameters():
#     param.requires_grad = False


# change fc layer structure
in_channel = net.fc.in_features
net.fc = nn.Linear(in_channel, 5)
net.to(device)


# define loss function
loss_function = nn.CrossEntropyLoss()


# construct an optimizer
params = [p for p in net.parameters() if p.requires_grad]
optimizer = optim.Adam(params, lr=0.0001)


epochs = 3
best_acc = 0.0
save_path = './resNet34.pth'
train_steps = len(train_loader)
for epoch in range(epochs):
    # train
    net.train()
    running_loss = 0.0
    train_bar = tqdm(train_loader, file=sys.stdout)
    for step, data in enumerate(train_bar):
        images, labels = data
        optimizer.zero_grad()
        logits = net(images.to(device))
        loss = loss_function(logits, labels.to(device))
        loss.backward()
        optimizer.step()


        # print statistics
        running_loss += loss.item()


        train_bar.desc = "train epoch[{}/{}] loss:{:.3f}".format(epoch + 1,
                                                                 epochs,
                                                                 loss)


    # validate
    net.eval()
    acc = 0.0  # accumulate accurate number / epoch
    with torch.no_grad():
        val_bar = tqdm(validate_loader, file=sys.stdout)
        for val_data in val_bar:
            val_images, val_labels = val_data
            outputs = net(val_images.to(device))
            # loss = loss_function(outputs, test_labels)
            predict_y = torch.max(outputs, dim=1)[1]
            acc += torch.eq(predict_y, val_labels.to(device)).sum().item()


            val_bar.desc = "valid epoch[{}/{}]".format(epoch + 1,
                                                       epochs)


    val_accurate = acc / val_num
    print('[epoch %d] train_loss: %.3f  val_accuracy: %.3f' %
          (epoch + 1, running_loss / train_steps, val_accurate))
```

```
        if val_accurate > best_acc:
            best_acc = val_accurate
            torch.save(net.state_dict(), save_path)


    print('Finished Training')




if __name__ == '__main__':
    main()
```

③predict

```
import os
import json


import torch
from PIL import Image
from torchvision import transforms
import matplotlib.pyplot as plt


from model import resnet34




def main():
    device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")


    data_transform = transforms.Compose(
        [transforms.Resize(256),
         transforms.CenterCrop(224),
         transforms.ToTensor(),
         transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])])


    # load image
    img_path = "../tulip.jpg"
    assert os.path.exists(img_path), "file: '{}' dose not exist.".format(img_path)
    img = Image.open(img_path)
    plt.imshow(img)
    # [N, C, H, W]
    img = data_transform(img)
    # expand batch dimension
    img = torch.unsqueeze(img, dim=0)


    # read class_indict
    json_path = './class_indices.json'
    assert os.path.exists(json_path), "file: '{}' dose not
exist.".format(json_path)


    with open(json_path, "r") as f:
        class_indict = json.load(f)


    # create model
    model = resnet34(num_classes=5).to(device)
```

```
    # load model weights
    weights_path = "./resNet34.pth"
    assert os.path.exists(weights_path), "file: '{}' dose not
exist.".format(weights_path)
    model.load_state_dict(torch.load(weights_path, map_location=device))


    # prediction
    model.eval()
    with torch.no_grad():
        # predict class
        output = torch.squeeze(model(img.to(device))).cpu()
        predict = torch.softmax(output, dim=0)
        predict_cla = torch.argmax(predict).numpy()


    print_res = "class: {}   prob: {:.3}".format(class_indict[str(predict_cla)],
                                                 predict[predict_cla].numpy())
    plt.title(print_res)
    for i in range(len(predict)):
        print("class: {:10}   prob: {:.3}".format(class_indict[str(i)],
                                                  predict[i].numpy()))
    plt.show()




if __name__ == '__main__':
    main()
```

④ batch-predict

```

import os
import json


import torch
from PIL import Image
from torchvision import transforms


from model import resnet34




def main():
    device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")


    data_transform = transforms.Compose(
        [transforms.Resize(256),
         transforms.CenterCrop(224),
         transforms.ToTensor(),
         transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])])


    # load image
    # 指向需要遍历预测的图像文件夹
    imgs_root = "/data/imgs"
    assert os.path.exists(imgs_root), f"file: '{imgs_root}' dose not exist."
    # 读取指定文件夹下所有jpg图像路径
    img_path_list = [os.path.join(imgs_root, i) for i in os.listdir(imgs_root) if
i.endswith(".jpg")]
```

```
    # read class_indict
    json_path = './class_indices.json'
    assert os.path.exists(json_path), f"file: '{json_path}' dose not exist."


    json_file = open(json_path, "r")
    class_indict = json.load(json_file)


    # create model
    model = resnet34(num_classes=5).to(device)


    # load model weights
    weights_path = "./resNet34.pth"
    assert os.path.exists(weights_path), f"file: '{weights_path}' dose not exist."
    model.load_state_dict(torch.load(weights_path, map_location=device))


    # prediction
    model.eval()
    batch_size = 8  # 每次预测时将多少张图片打包成一个batch
    with torch.no_grad():
        for ids in range(0, len(img_path_list) // batch_size):
            img_list = []
            for img_path in img_path_list[ids * batch_size: (ids + 1) *
batch_size]:
                assert os.path.exists(img_path), f"file: '{img_path}' dose not
exist."
                img = Image.open(img_path)
                img = data_transform(img)
                img_list.append(img)


            # batch img
            # 将img_list列表中的所有图像打包成一个batch
            batch_img = torch.stack(img_list, dim=0)
            # predict class
            output = model(batch_img.to(device)).cpu()
            predict = torch.softmax(output, dim=1)
            probs, classes = torch.max(predict, dim=1)


            for idx, (pro, cla) in enumerate(zip(probs, classes)):
                print("image: {}  class: {}  prob: {:.3}".format(img_path_list[ids
* batch_size + idx],
                                                                 class_indict[str(cla.numpy())],
                                                                 pro.numpy()))




if __name__ == '__main__':
    main()
```