人工神经网络模型与算法

2025-4-1

姓名：　　李丰克
专业：　自动化（控制）
年级：　　大二
学号：　　3230105182

Chapter 2 利用深度学习框架对比微调和从头训练

2.1 问题简介

以ResNeXt和DenseNet为例，自行选择数据集，按照从头训练和模型预处理后微调两种方式去训练模型，要求训练时间不低于两小时。

2.2 模型与算法介绍

2.2.1 ResNeXt

ResNeXt是在残差网络模型ResNet的基础上发展起来的，引入了分组卷积的概念，将输入层的不同特征图先进行分组，然后分别采用不同的卷积核进行卷积，这样虽然卷积核的总数量没变，但是总参数量减少了，减小了卷积的运算量。

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

上图为ResNet与ResNeXt的各层结构。可以发现两者的输入层、最大池化层、输出层均相同。中间的主体卷积层不同，不同点在于分组卷积。

输入层：7×7卷积，步长为2，输入为三通道RGB图象，输出为64通道。

最大池化层：3×3卷积，步长为2。

中间四层卷积：layer1：有3个ResNeXt模块，每个模块的输出通道数为256，每个分组卷积通道数64，基数为32。layer2：4个ResNeXt模块，每个模块输出通道数512，每个分组卷积通道数128，基数32。layer3：6个ResNeXt模块，每个模块输出通道数1024，每个分组卷积通道数256，基数32。layer4：3个ResNeXt模块，每个模块输出通道数2048，每个分组卷积通道数512，基数32。

输出层：全局平均池化层，全连接层，softmax层。

基于分组卷积大大减小的计算量，而且每层模块有相同的拓扑结构，提高了模型的通用性。

本次使用的模型为resnext50_32x4d。

2.2.2 ResNeXt的从头训练与预处理调参

首先要定义主体的每个resnext模块，因为使用的resnext50模型较为复杂，选用Bottleneck。

```
class Bottleneck(nn.Module):
    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
        #第一层卷积层
        self.conv1 = nn.Conv2d(in_channels=in_channel, out_channels=width,kernel_size=1, stride=1, bias=False)
        self.bn1 = nn.BatchNorm2d(width)
        #第二层中间层
        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)
        #第三层卷积层
        self.conv3 = nn.Conv2d(in_channels=width, out_channels=out_channel*self.expansion,kernel_size=1, stride=1, bias=False)
        self.bn3 = nn.BatchNorm2d(out_channel * self.expansion)
        self.relu=nn.ReLU(inplace=True)
        self.downsample=downsample
```

然后定义resnext整体结构，包括输入层、池化层、主题层和输出层。

```
class ResNeXt(nn.Module):
    def __init__(self,block,block_num,num_classes=1000,include_top=True,groups=1,width_per_group=64):
        super(ResNeXt,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,block_num[0])
        self.layer2=self._make_layer(block,128,block_num[1],stride=2)
        self.layer3=self._make_layer(block,256,block_num[2],stride=2)
        self.layer4=self._make_layer(block,512,block_num[3],stride=2)
```

对于resnext50_32×4d，分组卷积的分组数为32，每组的通道数为4。

从此基础上开始训练，即对resnext50从头训练。

在torchvision库中的models里，有预训练好的resnext50_32×4d模型，但是其默认的输出特征数为1000，即分类得到的类别数，可以根据数据集来调整输出特征数。

```
from torchvision import models
resnext50 = models.resnext50_32x4d(pretrained=True)
num_ftrs = resnext50.fc.in_features
for param in resnext50.parameters():
    param.requires_grad = False #False：冻结模型的参数，

#保持in_features不变，修改out_features=10
resnext50.fc = nn.Sequential(nn.Linear(num_ftrs,2),
                             nn.LogSoftmax(dim=1))
```

这样得到的模型即为预训练好的模型。

2.2.3 DenseNet

DenseNet(稠密神经网络),其与ResNet相比，将所有层连接，每个层都会接受其前面所有层作为其额外的输入，采用密集连接的方式来加快训练速度。

Layers	Output Size	DenseNet-121	DenseNet-169	DenseNet-201	DenseNet-264
Convolution	112 × 112	7 × 7 conv, stride 2			
Pooling	56 × 56	3 × 3 max pool, stride 2			
Dense Block (1)	56 × 56	[1 × 1 conv, 3 × 3 conv] × 6	[1 × 1 conv, 3 × 3 conv] × 6	[1 × 1 conv, 3 × 3 conv] × 6	[1 × 1 conv, 3 × 3 conv] × 6
Transition Layer (1)	56 × 56	1 × 1 conv			
	28 × 28	2 × 2 average pool, stride 2			
Dense Block (2)	28 × 28	[1 × 1 conv, 3 × 3 conv] × 12	[1 × 1 conv, 3 × 3 conv] × 12	[1 × 1 conv, 3 × 3 conv] × 12	[1 × 1 conv, 3 × 3 conv] × 12
Transition Layer (2)	28 × 28	1 × 1 conv			
	14 × 14	2 × 2 average pool, stride 2			
Dense Block (3)	14 × 14	[1 × 1 conv, 3 × 3 conv] × 24	[1 × 1 conv, 3 × 3 conv] × 32	[1 × 1 conv, 3 × 3 conv] × 48	[1 × 1 conv, 3 × 3 conv] × 64
Transition Layer (3)	14 × 14	1 × 1 conv			
	7 × 7	2 × 2 average pool, stride 2			
Dense Block (4)	7 × 7	[1 × 1 conv, 3 × 3 conv] × 16	[1 × 1 conv, 3 × 3 conv] × 32	[1 × 1 conv, 3 × 3 conv] × 32	[1 × 1 conv, 3 × 3 conv] × 48
Classification Layer	1 × 1	7 × 7 global average pool			
		1000D fully-connected, softmax			

如图为DenseNet的结构，首先经过了7×7的卷积层和3×3的最大池化层，然后进入了网络的主体结构，主体结构由DenseBlock和Transition层构成，然后通过全局平均池化层、全连接层、softmax层输出。

DenseBlock中，各个层的特征图大小一致，可以在channel维度上连接。DenseBlock中的非线性组合函数采用的是BN+ReLU+3x3 Conv的结构。由于后面层的输入会非常大，DenseBlock内部可以采用bottleneck层来减少计算量，即在原有的结构中增加1x1 Conv，提升计算效率。

Transition层，主要作用是是连接两个相邻的DenseBlock，并且降低特征图大小。Transition层包括一个1x1的卷积层和2x2的平均池化层，结构为BN+ReLU+1x1Conv+2x2AvgPooling。此外其也可以起到压缩模型的作用。

本次用到的模型为densenet121模型。

2.2.4 DenseNet的从头训练与预处理调参

首先定义denselayer，即为后续结果的单元模块。

```
class _DenseLayer(nn.Sequential):
    def __init__(self, num_input_features, growth_rate, bn_size, drop_rate):
        super(_DenseLayer, self).__init__()
        self.add_module("norm1", nn.BatchNorm2d(num_input_features))
        self.add_module("relu1", nn.ReLU(inplace=True))
        self.add_module("conv1", nn.Conv2d(num_input_features, bn_size*growth_rate,
                                           kernel_size=1, stride=1, bias=False))
        self.add_module("norm2", nn.BatchNorm2d(bn_size*growth_rate))
        self.add_module("relu2", nn.ReLU(inplace=True))
        self.add_module("conv2", nn.Conv2d(bn_size*growth_rate, growth_rate,
                                           kernel_size=3, stride=1, padding=1, bias=False))
        self.drop_rate = drop_rate

    def forward(self, x):
        new_features = super(_DenseLayer, self).forward(x)
        if self.drop_rate > 0:
            new_features = F.dropout(new_features, p=self.drop_rate, training=self.training)
        return torch.cat([x, new_features], 1)
```

然后定义DenseBlock和Transition层。

```
class _DenseBlock(nn.Sequential):
    def __init__(self, num_layers, num_input_features, bn_size, growth_rate, drop_rate):
        super(_DenseBlock, self).__init__()
        for i in range(num_layers):
            layer = _DenseLayer(num_input_features+i*growth_rate, growth_rate, bn_size,
                                drop_rate)
            self.add_module("denselayer%d" % (i+1,), layer)
```

```
class _Transition(nn.Sequential):
    def __init__(self, num_input_feature, num_output_features):
        super(_Transition, self).__init__()
        self.add_module("norm", nn.BatchNorm2d(num_input_feature))
        self.add_module("relu", nn.ReLU(inplace=True))
        self.add_module("conv", nn.Conv2d(num_input_feature, num_output_features,
                                          kernel_size=1, stride=1, bias=False))
        self.add_module("pool", nn.AvgPool2d(2, stride=2))
```

最后定义整体结构，加上输入层的卷积层和最大池化层与输出层。

```
from collections import OrderedDict
class DenseNet(nn.Module):
    def __init__(self, growth_rate=32, block_config=(6, 12, 24, 16), num_init_features=64,
                 bn_size=4, compression_rate=0.5, drop_rate=0, num_classes=1000):
        super(DenseNet, self).__init__()
        # first Conv2d
        self.features = nn.Sequential(OrderedDict([
            ("conv0", nn.Conv2d(3, num_init_features, kernel_size=7, stride=2, padding=3, bias=False)),
            ("norm0", nn.BatchNorm2d(num_init_features)),
            ("relu0", nn.ReLU(inplace=True)),
            ("pool0", nn.MaxPool2d(3, stride=2, padding=1))
        ]))

        # DenseBlock
        num_features = num_init_features
        for i, num_layers in enumerate(block_config):
            block = _DenseBlock(num_layers, num_features, bn_size, growth_rate, drop_rate)
            self.features.add_module("denseblock%d" % (i + 1), block)
            num_features += num_layers*growth_rate
            if i != len(block_config) - 1:
                transition = _Transition(num_features, int(num_features*compression_rate))
                self.features.add_module("transition%d" % (i + 1), transition)
                num_features = int(num_features * compression_rate)

        # final bn+ReLU
        self.features.add_module("norm5", nn.BatchNorm2d(num_features))
        self.features.add_module("relu5", nn.ReLU(inplace=True))

        # classification layer
        self.classifier = nn.Linear(num_features, num_classes)

        # params initialization
        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                nn.init.kaiming_normal_(m.weight)
```

从此基础上开始训练，即对resnext50从头训练。

而torchvision库中的models里，也有预训练好的DenseNet121模型，修改其输出特征数即可得到预处理好的模型。

```
from torchvision import models
densenet121 = models.densenet121(pretrained=True)
num_ftrs = densenet121.classifier.in_features
for param in densenet121.parameters():
    param.requires_grad = False #False：冻结模型的

#保持in_features不变，修改out_features=2
densenet121.classifier = nn.Linear(num_ftrs, 2)
```

2.3 数据集

本次选用的数据集为2013年kaggle里的经典竞赛题cats vs dogs。其中共包含25000张图片，猫狗图片比例1:1，分类任务为分辨出猫狗。

将图片文件保存在本地，用shutil库按照4:1的比例分出训练集和测试集分别存在两个文件夹，用torchvision库中的datasets读取数据，得到训练集与测试集的dataset类型变量，其中cat的标签为0，dog的标签为1。再用torch库中的dataloader方法加载数据集。其中batch_size(每次加载的样本数量)设置为64，num_worker(数据加载的子进程数量)根据cpu情况自行调整。

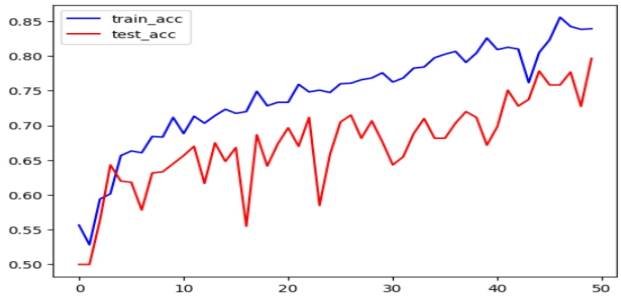
2.4 模型训练与结果

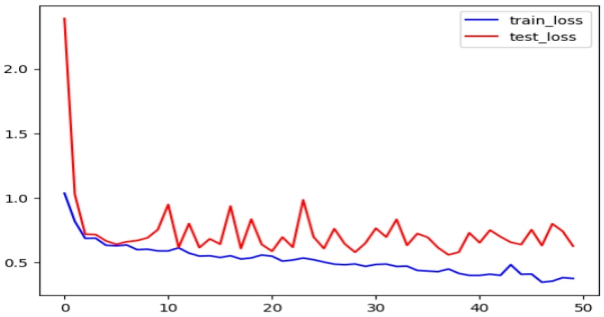
2.4.1 ResNeXt

对resnext50_32×4d分别进行从头训练和预处理训练，学习率为0.001，优化算法为Adam，损失函数为交叉熵损失函数。

对从头训练的模型，训练50轮，用时大约4h，结果如下。

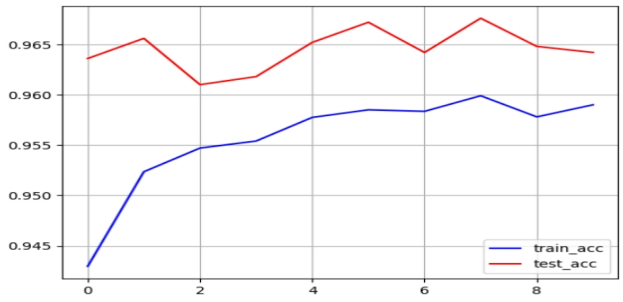
```
Epoch:45, Train_acc:80.5%, Train_loss:0.407, Test_acc:77.8%, Test_loss:0.637, Lr:1.00E-03
train over
Epoch:46, Train_acc:82.3%, Train_loss:0.409, Test_acc:75.8%, Test_loss:0.753, Lr:1.00E-03
train over
Epoch:47, Train_acc:85.6%, Train_loss:0.345, Test_acc:75.8%, Test_loss:0.630, Lr:1.00E-03
train over
Epoch:48, Train_acc:84.2%, Train_loss:0.354, Test_acc:77.7%, Test_loss:0.798, Lr:1.00E-03
train over
Epoch:49, Train_acc:83.8%, Train_loss:0.381, Test_acc:72.8%, Test_loss:0.740, Lr:1.00E-03
train over
Epoch:50, Train_acc:83.9%, Train_loss:0.375, Test_acc:79.6%, Test_loss:0.625, Lr:1.00E-03
Done
```

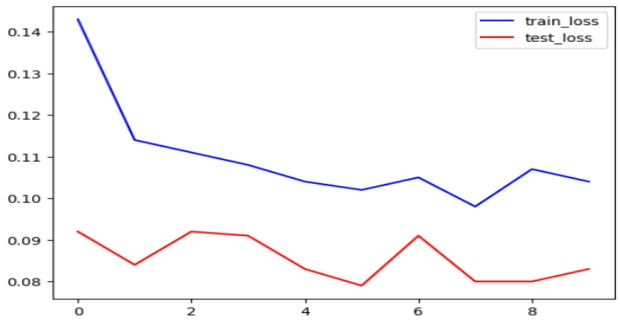




对预处理的模型，训练10轮，用时大约0.5h，结果如下。

```
train over
Epoch: 1, Train_acc:94.3%, Train_loss:0.143, Test_acc:96.4%, Test_loss:0.092, Lr:1.00E-03
train over
Epoch: 2, Train_acc:95.2%, Train_loss:0.114, Test_acc:96.6%, Test_loss:0.084, Lr:1.00E-03
train over
Epoch: 3, Train_acc:95.5%, Train_loss:0.111, Test_acc:96.1%, Test_loss:0.092, Lr:1.00E-03
train over
Epoch: 4, Train_acc:95.5%, Train_loss:0.108, Test_acc:96.2%, Test_loss:0.091, Lr:1.00E-03
train over
Epoch: 5, Train_acc:95.8%, Train_loss:0.104, Test_acc:96.5%, Test_loss:0.083, Lr:1.00E-03
train over
Epoch: 6, Train_acc:95.9%, Train_loss:0.102, Test_acc:96.7%, Test_loss:0.079, Lr:1.00E-03
train over
Epoch: 7, Train_acc:95.8%, Train_loss:0.105, Test_acc:96.4%, Test_loss:0.091, Lr:1.00E-03
train over
Epoch: 8, Train_acc:96.0%, Train_loss:0.098, Test_acc:96.8%, Test_loss:0.080, Lr:1.00E-03
train over
Epoch: 9, Train_acc:95.8%, Train_loss:0.107, Test_acc:96.5%, Test_loss:0.080, Lr:1.00E-03
train over
Epoch:10, Train_acc:95.9%, Train_loss:0.104, Test_acc:96.4%, Test_loss:0.083, Lr:1.00E-03
Done
```





可以看出从头训练时，五十轮后在测试集上有大约80%的准确率。而预处理的模型初始便有96%的准确率，所以这里只训练了十轮，受限于电脑性能，如果将训练轮数提高到100轮或以上，可能能够达到90%以上的准确率。

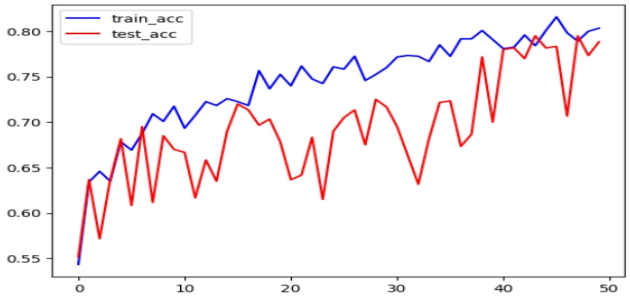
虽然没有在图中表示出来，但是在训练过程中可以看出，使用预处理的模型每轮的训练时间显著少于从头训练的。这可能与结构也有关系。

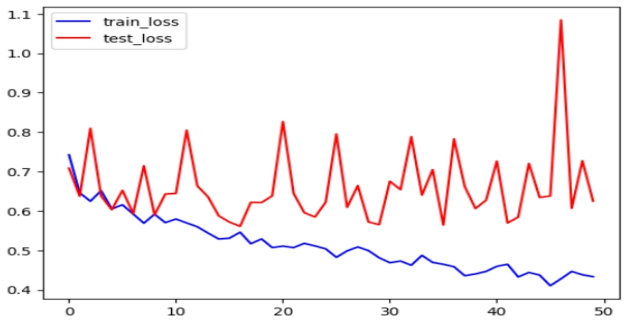
2.4.2 DenseNet

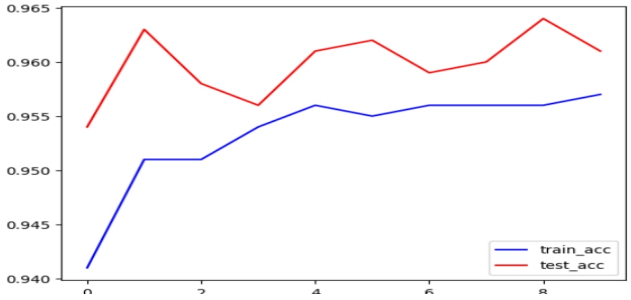
对DenseNet121模型，分别进行从头训练和预处理训练，学习率为0.001，优化算法为Adam，损失函数为交叉熵损失函数。

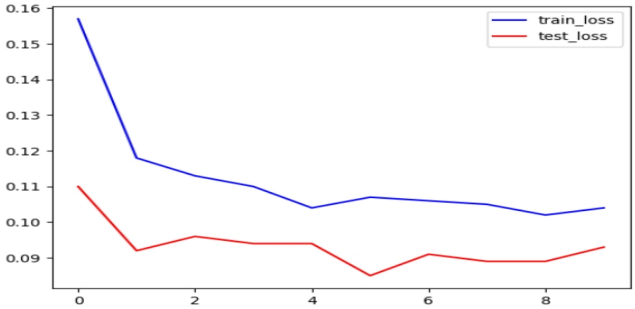
对从头训练的模型，训练50轮，用时大约5h，结果如下。

```
Epoch:45, Train_acc:80.1%, Train_loss:0.437, Test_acc:78.2%, Test_loss:0.634, Lr:1.00E-03
train over
Epoch:46, Train_acc:81.6%, Train_loss:0.410, Test_acc:78.3%, Test_loss:0.638, Lr:1.00E-03
train over
Epoch:47, Train_acc:79.8%, Train_loss:0.428, Test_acc:70.7%, Test_loss:1.085, Lr:1.00E-03
train over
Epoch:48, Train_acc:78.9%, Train_loss:0.446, Test_acc:79.5%, Test_loss:0.607, Lr:1.00E-03
train over
Epoch:49, Train_acc:80.0%, Train_loss:0.438, Test_acc:77.3%, Test_loss:0.727, Lr:1.00E-03
train over
Epoch:50, Train_acc:80.3%, Train_loss:0.433, Test_acc:78.8%, Test_loss:0.625, Lr:1.00E-03
Done
```









对于预处理的模型，训练10轮，用时大约0.5h，结果如下。

```
train over
Epoch: 1, Train_acc:94.1%, Train_loss:0.157, Test_acc:95.4%, Test_loss:0.110, Lr:1.00E-03
train over
Epoch: 2, Train_acc:95.1%, Train_loss:0.118, Test_acc:96.3%, Test_loss:0.092, Lr:1.00E-03
train over
Epoch: 3, Train_acc:95.1%, Train_loss:0.113, Test_acc:95.8%, Test_loss:0.096, Lr:1.00E-03
train over
Epoch: 4, Train_acc:95.4%, Train_loss:0.110, Test_acc:95.6%, Test_loss:0.094, Lr:1.00E-03
train over
Epoch: 5, Train_acc:95.6%, Train_loss:0.104, Test_acc:96.1%, Test_loss:0.094, Lr:1.00E-03
train over
Epoch: 6, Train_acc:95.5%, Train_loss:0.107, Test_acc:96.2%, Test_loss:0.085, Lr:1.00E-03
train over
Epoch: 7, Train_acc:95.6%, Train_loss:0.106, Test_acc:95.9%, Test_loss:0.091, Lr:1.00E-03
train over
Epoch: 8, Train_acc:95.6%, Train_loss:0.105, Test_acc:96.0%, Test_loss:0.089, Lr:1.00E-03
train over
Epoch: 9, Train_acc:95.7%, Train_loss:0.102, Test_acc:96.4%, Test_loss:0.089, Lr:1.00E-03
train over
Epoch:10, Train_acc:95.6%, Train_loss:0.104, Test_acc:96.1%, Test_loss:0.093, Lr:1.00E-03
Done
```

整体结果与resnext类似，如果训练轮数能够更多，准确率可以更高。而且预训练每轮所需的时间更短。

2.5 代码附录

代码的zjugit网址：https://git.zju.edu.cn/3230105182/ResnextandDensenet