卷积神经网络实验报告

- 姓名：陈睿颖
- 学号：2013544
- 专业：计算机科学与技术

1. 实验要求

- 掌握卷积的基本原理
- 学会使用PyTorch搭建简单的CNN实现Cifar10数据集分类
- 学会使用PyTorch搭建简单的ResNet实现Cifar10数据集分类
- 学会使用PyTorch搭建简单的DenseNet实现Cifar10数据集分类
- 学会使用PyTorch搭建简单的SE-ResNet实现Cifar10数据集分类

2. 实验

2.1 原始CNN

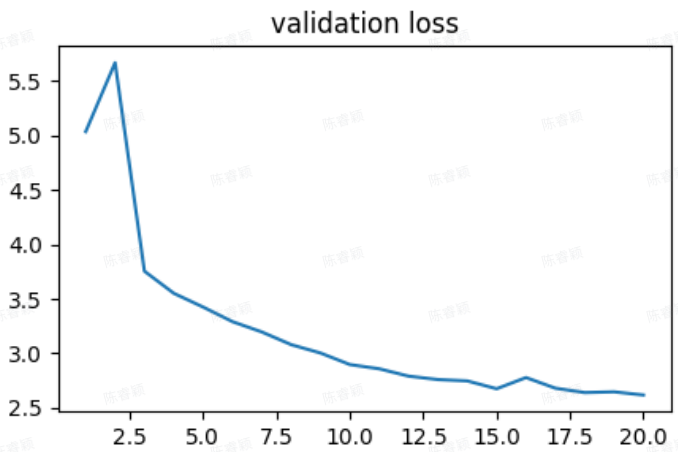
网络结构

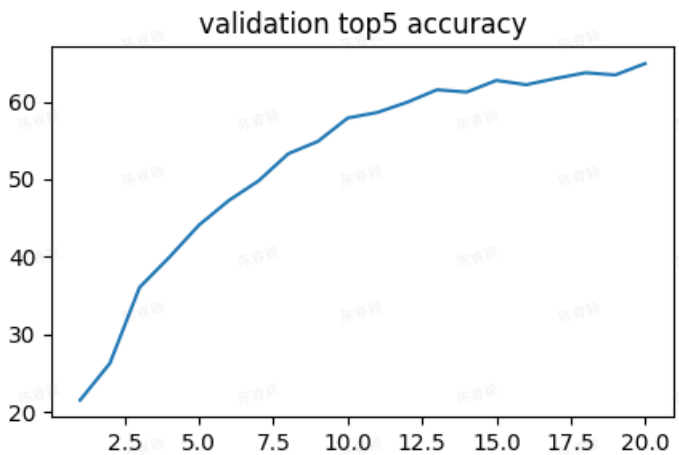
```
Net(
  (conv1): Conv2d(3, 6, kernel_size=(5, 5), stride=(1, 1))
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv2): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
  (fc1): Linear(in_features=400, out_features=120, bias=True)
  (fc2): Linear(in_features=120, out_features=84, bias=True)
  (fc3): Linear(in_features=84, out_features=10, bias=True)
)
```

如图所示，打印的net结构为

1. `conv1`：输入通道数为3，输出通道数为6，使用5x5的卷积核，步幅为1，没有填充。这一层的作用是提取图像的特征。
2. `pool`：使用2x2的最大池化操作，步幅为2。这一层的作用是进行下采样，减小特征图的尺寸，并保留主要的特征。
3. `conv2`：输入通道数为6，输出通道数为16，使用5x5的卷积核，步幅为1，没有填充。这一层继续提取更高级的特征。
4. `fc1`：全连接层，输入特征数为400，输出特征数为120。这一层的作用是将卷积层提取的特征映射到更高维度的特征空间。
5. `fc2`：全连接层，输入特征数为120，输出特征数为84。这一层继续将特征映射到更高维度的特征空间。
6. `fc3`：全连接层，输入特征数为84，输出特征数为10。这一层最终将特征映射到对应类别的分数或概率。

训练loss曲线、准确度曲线图





可以看到，经过20轮训练，准确率在65%左右。

2.2 ResNet网络

ResNet是一种深度卷积神经网络架构，由Kaiming He等人在2015年的论文《Deep Residual Learning for Image Recognition》中提出。ResNet通过使用残差模块（residual blocks）来解决深度网络训练过程中的梯度消失和退化问题，使得可以训练非常深的网络。

在传统的卷积神经网络中，增加网络的深度会导致网络性能的下降，因为加深网络会增加训练过程中的梯度消失问题。ResNet通过引入残差连接（residual connection）来解决这个问题。残差连接将输入直接与输出相加，使得网络可以学习残差（即输入与输出之间的差异），而不是学习完整的映射。这种残差学习的方式使得网络更容易优化，并且允许更深的网络能够获得更好的性能。

ResNet的核心思想是引入了残差块（residual block），它由两个或三个卷积层组成。其中，第一个卷积层用于降低维度，第二个卷积层用于恢复维度，并将其与输入相加。这种残差块的设计使得信息可以在网络中更直接地传播，避免了梯度消失问题。

ResNet还使用了批量归一化（batch normalization）和全局平均池化（global average pooling）等技术来加速训练和提高模型性能。

代码实现

layer name	output size	18-layer
conv1	112×112	
conv2_x	56×56	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$

ResNet18各层结构[1]

由于CIFAR10数据集较小，我们可以选择ResNet18来进行实现：

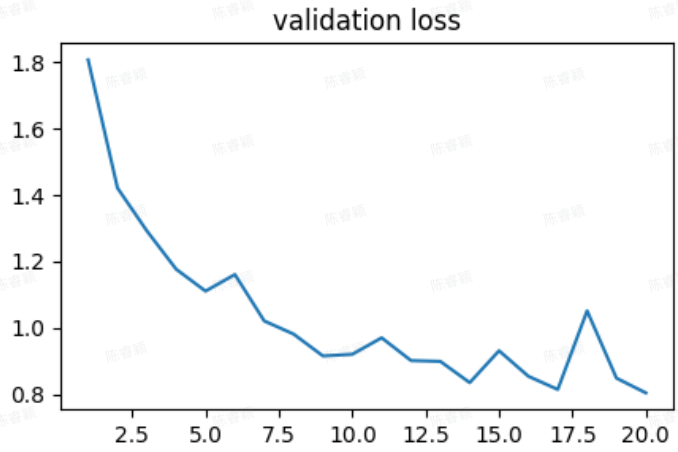
```python
import torch.nn as nn
import torch.nn.functional as F

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

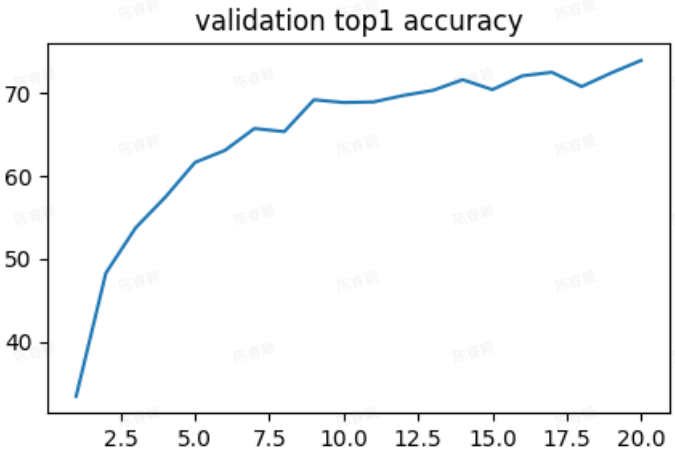
```
6
7       def __init__(self, in_channel, out_channel, stride=1, downsample=None, **kwa
8           super(BasicBlock, self).__init__()
9           self.conv1 = nn.Conv2d(in_channels=in_channel, out_channels=out_channel,
10                                 kernel_size=3, stride=stride, padding=1, bias=Fal
11          self.bn1 = nn.BatchNorm2d(out_channel)
12          self.relu = nn.ReLU()
13          self.conv2 = nn.Conv2d(in_channels=out_channel, out_channels=out_channel
14                                 kernel_size=3, stride=1, padding=1, bias=False)
15          self.bn2 = nn.BatchNorm2d(out_channel)
16          self.downsample = downsample
17
18      def forward(self, x):
19          identity = x
20          if self.downsample is not None:
21              identity = self.downsample(x)
22
23          out = self.conv1(x)
24          out = self.bn1(out)
25          out = self.relu(out)
26
27          out = self.conv2(out)
28          out = self.bn2(out)
29
30          out += identity
31          out = self.relu(out)
32
33          return out
34
35
36  class Bottleneck(nn.Module):
37
38      expansion = 4
39
40      def __init__(self, in_channel, out_channel, stride=1, downsample=None, group
41          super(Bottleneck, self).__init__()
42
43          width = int(out_channel * (width_per_group / 64.)) * groups
44
45          self.conv1 = nn.Conv2d(in_channels=in_channel, out_channels=width,
46                                 kernel_size=1, stride=1, bias=False)  # squeeze c
47          self.bn1 = nn.BatchNorm2d(width)
48          # -----------------------------------------
49          self.conv2 = nn.Conv2d(in_channels=width, out_channels=width, groups=gro
50                                 kernel_size=3, stride=stride, bias=False, padding
51          self.bn2 = nn.BatchNorm2d(width)
52          # -----------------------------------------
```

```
53          self.conv3 = nn.Conv2d(in_channels=width, out_channels=out_channel*self.
54                                 kernel_size=1, stride=1, bias=False)  # unsqueeze
55          self.bn3 = nn.BatchNorm2d(out_channel*self.expansion)
56          self.relu = nn.ReLU(inplace=True)
57          self.downsample = downsample
58
59      def forward(self, x):
60          identity = x
61          if self.downsample is not None:
62              identity = self.downsample(x)
63
64          out = self.conv1(x)
65          out = self.bn1(out)
66          out = self.relu(out)
67
68          out = self.conv2(out)
69          out = self.bn2(out)
70          out = self.relu(out)
71
72          out = self.conv3(out)
73          out = self.bn3(out)
74
75          out += identity
76          out = self.relu(out)
77
78          return out
79
80
81  class ResNet(nn.Module):
82
83      def __init__(self, block, blocks_num, num_classes=1000):
84          super(ResNet, self).__init__()
85          self.in_channel = 64
86
87          self.conv1 = nn.Conv2d(3, self.in_channel, kernel_size=7, stride=2, padd
88          self.bn1 = nn.BatchNorm2d(self.in_channel)
89          self.relu = nn.ReLU(inplace=True)
90          self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
91          self.layer1 = self._make_layer(block, 64, blocks_num[0])
92          self.layer2 = self._make_layer(block, 128, blocks_num[1], stride=2)
93          self.layer3 = self._make_layer(block, 256, blocks_num[2], stride=2)
94          self.layer4 = self._make_layer(block, 512, blocks_num[3], stride=2)
95          self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
96          self.fc = nn.Linear(512 * block.expansion, num_classes)
97
98      def _make_layer(self, block, channel, block_num, stride=1):
99          downsample = None
```

```
        if stride != 1 or self.in_channel != channel * block.expansion:
            downsample = nn.Sequential(
                nn.Conv2d(self.in_channel, channel * block.expansion, kernel_siz
                nn.BatchNorm2d(channel * block.expansion))

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

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

        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)

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

        return x

def resnet18(num_classes=10):
    return ResNet(BasicBlock, [2, 2, 2, 2], num_classes=num_classes)

net = resnet18().to(device)
print(net)
```

运行结果





可以看到，在经过20轮的训练后，ResNet18在CIFAR10上的准确率达到97%左右。

2.3 SE-ResNet

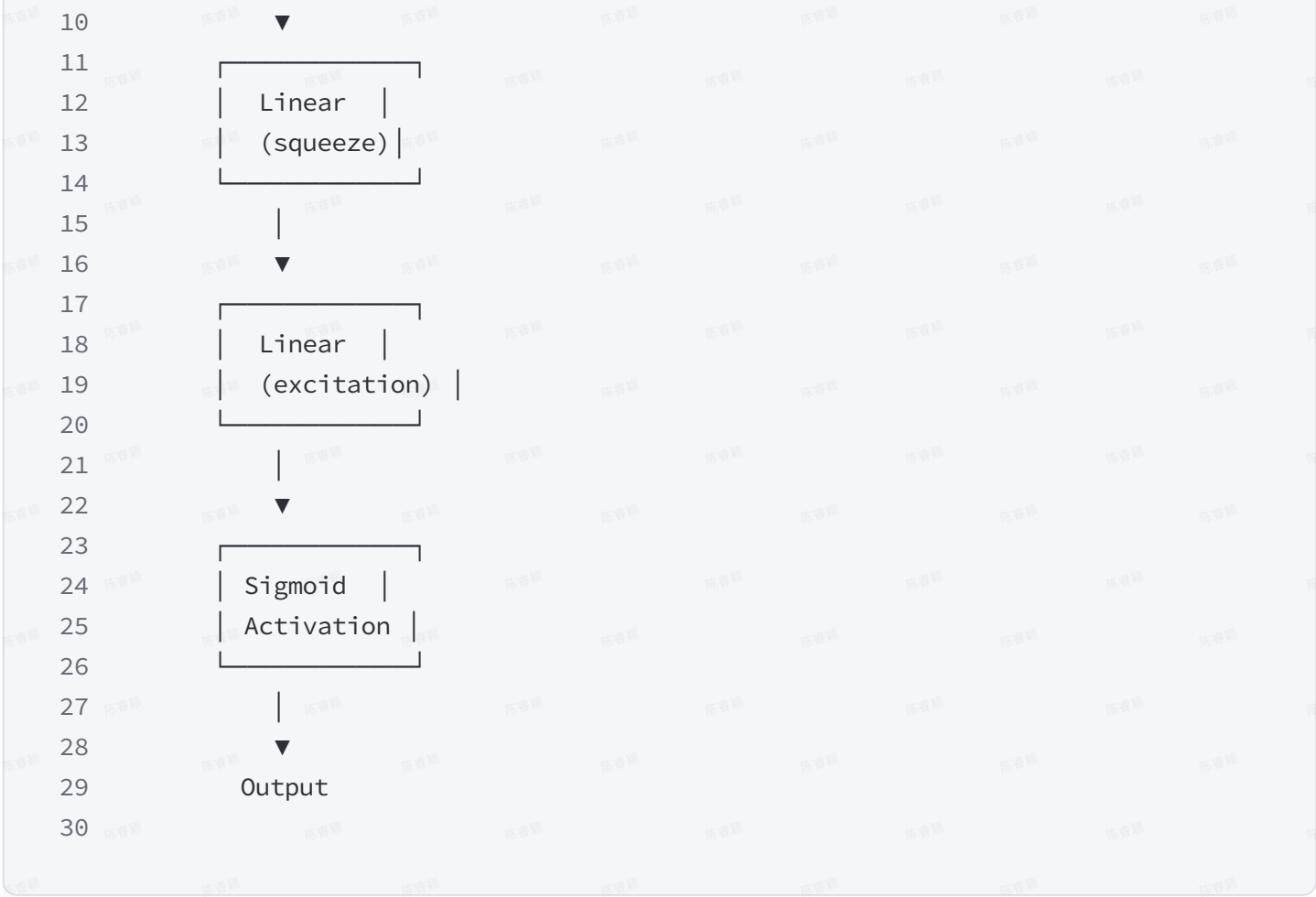
SE模块（Squeeze-and-Excitation Module）是一种用于增强深度神经网络性能的模块，特别是在卷积神经网络（CNN）中应用广泛。SE模块通过自适应地学习通道间的依赖关系来增强特征表示能力，从而提升模型的性能。

SE模块主要由两个步骤组成：压缩（squeeze）和激励（excitation）。

1. 压缩（Squeeze）：
   在这一步骤中，SE模块通过全局平均池化（global average pooling）操作来压缩输入特征图在通道维度上的信息。这意味着对于每个通道，SE模块会计算该通道上所有空间位置的平均值，从而得到一个通道数相同的全局特征描述符。
2. 激励（Excitation）：
   在这一步骤中，SE模块利用一个小型的全连接（FC）神经网络来学习通道间的依赖关系。该网络通常由两个连续的全连接层组成，这些层通过非线性激活函数（如ReLU）进行连接。这些层的输出表示了每个通道对于重要性的度量，即该通道的激活值在整个特征图中的重要程度。

最后，通过使用学习到的通道权重，SE模块通过将输入特征图与通道权重相乘来重新加权输入特征，以增强对于重要特征的响应，从而提升模型性能。

```
      Input
        |
        ▼
  ┌──────────┐
  │  Global  │
  │  Average │
  │  Pooling │
  └──────────┘
        |
```



SE模块的基本结构和信息流动。示意图中的"Linear"表示全连接层，"Sigmoid Activation"表示sigmoid激活函数。

代码实现

```python
import torch
import torch.nn as nn
import torch.nn.functional as F

class ResidualBlock(nn.Module):
    def __init__(self, inchannel, outchannel, stride=1): #需要确定输入维度和输出维度，步长默认为1
        super(ResidualBlock, self).__init__()
        ##################
        if outchannel == 64:
            self.globalAvgPool = nn.AvgPool2d(32, stride=1)
        elif outchannel == 128:
            self.globalAvgPool = nn.AvgPool2d(16, stride=1)
        elif outchannel == 256:
            self.globalAvgPool = nn.AvgPool2d(8, stride=1)
        elif outchannel == 512:
```

```
16              self.globalAvgPool = nn.AvgPool2d(4, stride=1)
17
18          self.fc1 = nn.Linear(in_features=outchannel,
    out_features=round(outchannel / 16))
19          self.fc2 = nn.Linear(in_features=round(outchannel / 16),
    out_features=outchannel)
20          self.sigmoid = nn.Sigmoid()
21          ####################
22
23          ####################
24          # self.seg = nn.Sequential(
25          #
26          # )
27          ####################
28          self.left = nn.Sequential(  #残差主体部分
29              nn.Conv2d(inchannel, outchannel, kernel_size=3, stride=stride,
    padding=1, bias=False), #第一个卷积把输入维度转成输出维度，步长可变
30              nn.BatchNorm2d(outchannel),
31              nn.ReLU(inplace=True),
32              nn.Conv2d(outchannel, outchannel, kernel_size=3, stride=1,
    padding=1, bias=False),#第二个卷积输入和输出的维度都是一样的，都为输出维度，步长为1
33              nn.BatchNorm2d(outchannel)
34          )
35          self.shortcut = nn.Sequential()  #一般而言，shortcut不做操作，输入x就输出x
36          if stride != 1 or inchannel != outchannel: #但是block的步长万一不是1 或者
    block的输入维度与输出维度不一致
37              self.shortcut = nn.Sequential(
38                  nn.Conv2d(inchannel, outchannel, kernel_size=1, stride=stride,
    bias=False), #shortcut的卷积，若步长不为1或者输入维度与输出维度不一致，就会触发
39                  nn.BatchNorm2d(outchannel)
40              )
41
42      def forward(self, x):  #block的前向反馈，包含主体的2个卷积和shortcut可能触发的一
    个卷积
43          out = self.left(x)      #Residual主体
44          original_out = out
45          ###seg
46          out = self.globalAvgPool(out)
47          out = out.view(out.size(0), -1)
48          out = self.fc1(out)
49          out = F.relu(out)
50          out = self.fc2(out)
51          out = self.sigmoid(out)
52          out = out.view(out.size(0), out.size(1), 1, 1)
53          out = out * original_out
54          ###
55
```

```
56          out += self.shortcut(x)#shortcut
57          out = F.relu(out)
58          return out
59
60  class ResNet(nn.Module):
61      def __init__(self, ResidualBlock, num_classes=10):
62          super(ResNet, self).__init__()
63          self.inchannel = 64  #输入网络后，遇到的第一个卷积是3x3，64层的
64          self.conv1 = nn.Sequential(
65              nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1, bias=False), #
    输入维度是3，因为一开始输入是RGB，卷积是3x3，64
66              nn.BatchNorm2d(64),
67              nn.ReLU(),
68          )
69          self.layer1 = self.make_layer(ResidualBlock, 64,  2, stride=1) #4个卷积
    是 3x3，64层的，其中有两个block，每个block2个卷积
70          self.layer2 = self.make_layer(ResidualBlock, 128, 2, stride=2)#4个卷积
    是 3x3，128层的，其中有两个block，每个block2个卷积
71          self.layer3 = self.make_layer(ResidualBlock, 256, 2, stride=2)#4个卷积
    是 3x3，256层的，其中有两个block，每个block2个卷积
72          self.layer4 = self.make_layer(ResidualBlock, 512, 2, stride=2)#4个卷积
    是 3x3，512层的，其中有两个block，每个block2个卷积
73          self.fc = nn.Linear(512, num_classes)   #最后的全连接层，512层要转成10层，
    因为cifar数据集是10分类的
74
75      def make_layer(self, block, channels, num_blocks, stride):#制造layer，一个
    layer 两个block
76          strides = [stride] + [1] * (num_blocks - 1)   #strides=[1,1],或[2,1]
77          layers = []
78          for stride in strides:#制造不同步长(1或者2)的block，一个block两个卷积，可能
    触发shortcut的第三个卷积（1不同layer之间的卷积会触发）
79              layers.append(block(self.inchannel, channels, stride))
80              self.inchannel = channels
81          return nn.Sequential(*layers)
82
83      def forward(self, x):
84          out = self.conv1(x)              #只有最开始的一个卷积
85          out = self.layer1(out)           #4个64层3x3卷积
86          out = self.layer2(out)           #4个128层3x3卷积
87          out = self.layer3(out)           #4个256层3x3卷积
88          out = self.layer4(out)           #4个512层3x3卷积
89          out = F.avg_pool2d(out, 4)       #最后的输出是4x4的，所以这里第二个参数为4
90          out = out.view(out.size(0), -1)
91          out = self.fc(out)
92          return out
93
94  def SE_ResNet18():
```

```
95
96      return ResNet(ResidualBlock)
97
98  net=SE_ResNet18().to(device)
```

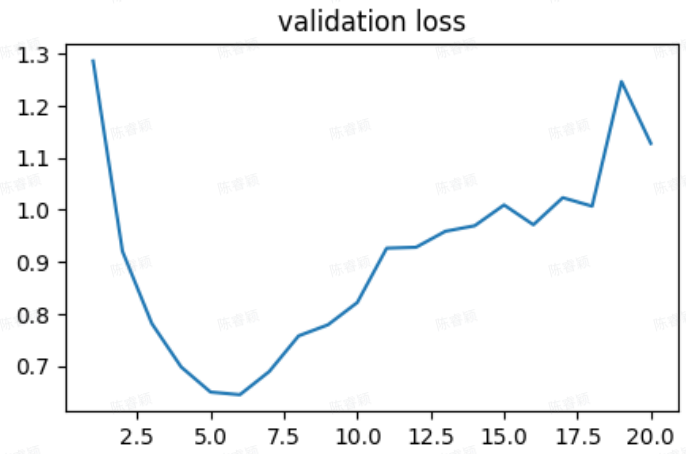
在 `ResidualBlock` 类中，可以看到以下部分代码实现了SE模块：

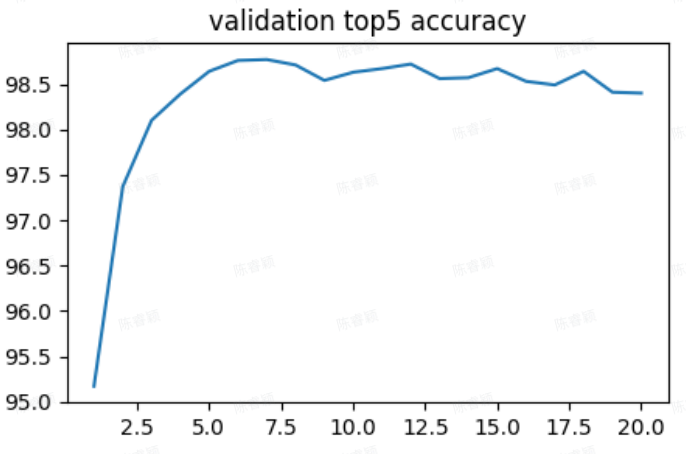
```
1  pythonCopy code
2  if outchannel == 64:
3      self.globalAvgPool = nn.AvgPool2d(32, stride=1)
4  elif outchannel == 128:
5      self.globalAvgPool = nn.AvgPool2d(16, stride=1)
6  elif outchannel == 256:
7      self.globalAvgPool = nn.AvgPool2d(8, stride=1)
8  elif outchannel == 512:
9      self.globalAvgPool = nn.AvgPool2d(4, stride=1)
10
11 self.fc1 = nn.Linear(in_features=outchannel, out_features=round(outchannel / 16)
12 self.fc2 = nn.Linear(in_features=round(outchannel / 16), out_features=outchannel
13 self.sigmoid = nn.Sigmoid()
```

这些代码定义了SE模块的主要组成部分。其中，`self.globalAvgPool` 使用了平均池化层，`self.fc1` 和 `self.fc2` 是两个全连接层，而 `self.sigmoid` 是一个sigmoid激活函数。这些组件用于计算SE模块的权重，并将其应用于输入数据。在 `forward` 方法中，可以看到具体的计算流程，包括计算SE模块的输出，与原始输入进行相乘并返回结果。

因此，该代码确实实现了SE模块。

运行结果





由图可以看到，在经过20轮的训练之后，带有SE模块的ResNet-18在CIFAR10数据集上取得了97%左右的正确率。

2.4 DenseNet

DenseNet通过引入密集连接（Dense Connection）的方式来促进信息流动和特征重用。相比于传统的卷积神经网络，DenseNet具有更强的特征表达能力和梯度传播效益，可以有效解决深层网络中的梯度消失问题。

DenseNet的核心思想是通过密集连接将前面层的特征图与后面层的特征图进行连接。与传统网络中将特征图逐层串联不同，DenseNet中每一层的特征图都与后续所有层的特征图直接连接。这种密集连接的方式使得每一层的输入包含了前面所有层的特征信息，同时也能够利用后续层的特征信息。这种密集连接的方式有效地提高了信息传递的效率，并且使得网络更加稠密和紧凑。

DenseNet的基本单元是称为"Dense Block"的模块。Dense Block由多个卷积层组成，每个卷积层的输入都是前面所有层的特征图的堆叠。在每个卷积层之后，特征图会被拼接在一起，作为下一层的输入。这种密集连接的方式使得特征图在网络中的传递更加直接和高效。

除了Dense Block，DenseNet还引入了"Transition Layer"来控制特征图的维度和尺寸。Transition Layer通过使用1x1卷积层和池化层来减小特征图的通道数和空间尺寸，从而减少网络的计算复杂度。这样的设计使得DenseNet具有较少的参数量和计算量，同时在保持较高特征表达能力的同时提高了网络的效率。

```
 Input
   |
Conv2d
   |
Dense Block 1
   |
Transition 1
   |
Dense Block 2
   |
Transition 2
   |
Dense Block 3
   |
Transition 3
   |
Dense Block 4
   |
```

```
                    BatchNorm2d
                        |
                      ReLU
                        |
                AdaptiveAvgPool2d
                        |
                   Flatten
                        |
                    Linear
                        |
                    Output
```

输入图像首先经过一个卷积层(Conv2d)。然后，通过四个密集块(Dense Block)和三个过渡层(Transition)进行特征提取。最后，通过批标准化(BatchNorm2d)、ReLU激活函数和自适应平均池化层(AdaptiveAvgPool2d)进行全局特征池化。最后，通过展平(Flatten)操作和线性层(Linear)将特征映射转换为最终的输出。

代码实现

```python
class Bottleneck(nn.Module):
    def __init__(self, in_channels, growth_rate):
        super().__init__()
        #"""In  our experiments, we let each 1×1 convolution
        #produce 4k feature-maps."""
        inner_channel = 4 * growth_rate

        #"""We find this design especially effective for DenseNet and
        #we refer to our network with such a bottleneck layer, i.e.,
        #to the BN-ReLU-Conv(1×1)-BN-ReLU-Conv(3×3) version of H ` ,
        #as DenseNet-B."""
        self.bottle_neck = nn.Sequential(
            nn.BatchNorm2d(in_channels),
            nn.ReLU(inplace=True),
            nn.Conv2d(in_channels, inner_channel, kernel_size=1, bias=False),
            nn.BatchNorm2d(inner_channel),
            nn.ReLU(inplace=True),
            nn.Conv2d(inner_channel, growth_rate, kernel_size=3, padding=1,
bias=False)
        )

```

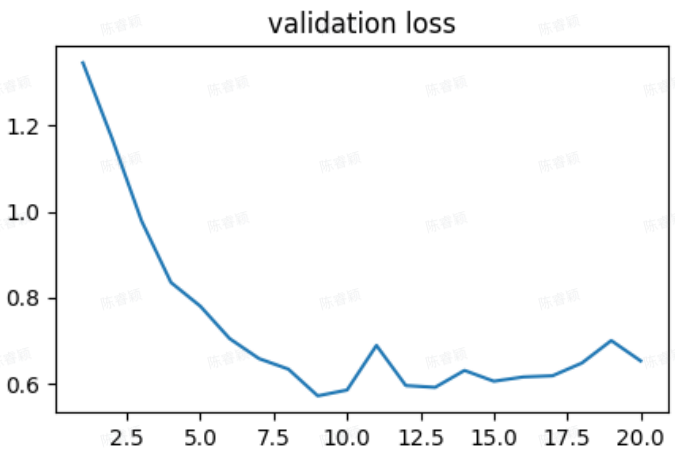
```
    def forward(self, x):
        return torch.cat([x, self.bottle_neck(x)], 1)

#"""We refer to layers between blocks as transition
#layers, which do convolution and pooling."""
class Transition(nn.Module):
    def __init__(self, in_channels, out_channels):
        super().__init__()
        #"""The transition layers used in our experiments
        #consist of a batch normalization layer and an 1×1
        #convolutional layer followed by a 2×2 average pooling
        #layer""".
        self.down_sample = nn.Sequential(
            nn.BatchNorm2d(in_channels),
            nn.Conv2d(in_channels, out_channels, 1, bias=False),
            nn.AvgPool2d(2, stride=2)
        )

    def forward(self, x):
        return self.down_sample(x)

#DesneNet-BC
#B stands for bottleneck layer(BN-RELU-CONV(1x1)-BN-RELU-CONV(3x3))
#C stands for compression factor(0<=theta<=1)
class DenseNet(nn.Module):
    def __init__(self, block, nblocks, growth_rate=12, reduction=0.5,
num_class=100):
        super().__init__()
        self.growth_rate = growth_rate

        #"""Before entering the first dense block, a convolution
        #with 16 (or twice the growth rate for DenseNet-BC)
        #output channels is performed on the input images."""
        inner_channels = 2 * growth_rate

        #For convolutional layers with kernel size 3×3, each
        #side of the inputs is zero-padded by one pixel to keep
        #the feature-map size fixed.
        self.conv1 = nn.Conv2d(3, inner_channels, kernel_size=3, padding=1,
bias=False)

        self.features = nn.Sequential()

        for index in range(len(nblocks) - 1):
            self.features.add_module("dense_block_layer_{}".format(index),
self._make_dense_layers(block, inner_channels, nblocks[index]))
            inner_channels += growth_rate * nblocks[index]
```

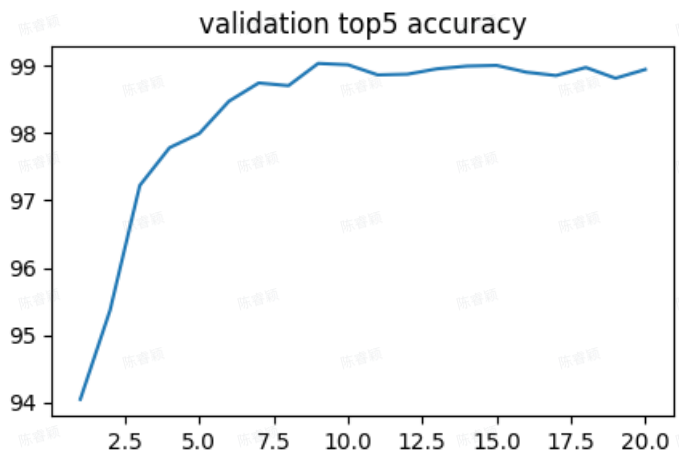
```
65
66              #"""If a dense block contains m feature-maps, we let the
67              #following transition layer generate θm output feature-
68              #maps, where 0 < θ ≤ 1 is referred to as the compression
69              #fac-tor.
70              out_channels = int(reduction * inner_channels) # int() will
    automatic floor the value
71              self.features.add_module("transition_layer_{}".format(index),
    Transition(inner_channels, out_channels))
72              inner_channels = out_channels
73
74          self.features.add_module("dense_block{}".format(len(nblocks) - 1),
    self._make_dense_layers(block, inner_channels, nblocks[len(nblocks)-1]))
75          inner_channels += growth_rate * nblocks[len(nblocks) - 1]
76          self.features.add_module('bn', nn.BatchNorm2d(inner_channels))
77          self.features.add_module('relu', nn.ReLU(inplace=True))
78
79          self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
80
81          self.linear = nn.Linear(inner_channels, num_class)
82
83      def forward(self, x):
84          output = self.conv1(x)
85          output = self.features(output)
86          output = self.avgpool(output)
87          output = output.view(output.size()[0], -1)
88          output = self.linear(output)
89          return output
90
91      def _make_dense_layers(self, block, in_channels, nblocks):
92          dense_block = nn.Sequential()
93          for index in range(nblocks):
94              dense_block.add_module('bottle_neck_layer_{}'.format(index),
    block(in_channels, self.growth_rate))
95              in_channels += self.growth_rate
96          return dense_block
97
98  def densenet121():
99      return DenseNet(Bottleneck, [6,12,24,16], growth_rate=32)
100
101 def densenet169():
102     return DenseNet(Bottleneck, [6,12,32,32], growth_rate=32)
103
104 def densenet201():
105     return DenseNet(Bottleneck, [6,12,48,32], growth_rate=32)
106
107 def densenet161():
```

```
108     return DenseNet(Bottleneck, [6,12,36,24], growth_rate=48)
```

- Bottleneck类定义了DenseNet中的Bottleneck单元，它由一系列层组成，包括BatchNormalization、ReLU激活函数、1x1卷积、BatchNormalization、ReLU激活函数和3x3卷积。在前向传播中，输入x通过Bottleneck单元后与原始输入x进行拼接，形成密集连接的输出。
- Transition类定义了DenseNet中的Transition层，它由一系列层组成，包括BatchNormalization、1x1卷积和2x2平均池化。Transition层用于在不同密集块之间进行卷积和池化操作，从而控制特征图的通道数和尺寸。
- DenseNet类定义了完整的DenseNet模型。它接受一个Bottleneck类作为基本单元，以及一个表示每个密集块中Bottleneck单元数量的列表nblocks。在模型的初始化过程中，通过添加一系列密集块和Transition层来构建DenseNet的特征提取部分。最后，通过全局平均池化和线性层将特征映射转换为最终的输出。

运行结果





20轮epoch后同样在CIFAR10上达到了98%的准确率。

3. 思考题

解释没有跳跃连接的卷积网络、ResNet、DenseNet在训练过程中有什么不同

1. 没有跳跃连接的卷积网络（如普通的卷积神经网络）：这种网络通过堆叠卷积层和池化层来逐渐提取特征，然后通过全连接层进行分类。在训练过程中，每一层的输出直接传递给下一层，没有额外的路径或连接。

2. ResNet（残差网络）：ResNet引入了跳跃连接或称为“残差连接”，这样可以在网络中建立直接的捷径。每个残差块由两个卷积层组成，并通过跳跃连接将输入直接添加到输出上。这种设计使得网络可以更容易地学习恒等映射，从而避免了梯度消失的问题。在训练过程中，通过残差连接，梯度可以更轻松地在网络中传播，有助于更深层次的训练。
3. DenseNet：DenseNet引入了密集连接的概念，即每一层的输出都与后续层的输入连接在一起。这种设计使得网络中的每一层都可以直接访问来自前面所有层的特征图。在训练过程中，通过密集连接，特征可以更充分地传递和重用，促进了信息流动和特征重用。这有助于减轻梯度消失问题，增强了特征的多样性和网络的表达能力。

总的来说，ResNet通过残差连接解决了梯度消失问题，并支持更深的网络训练。而DenseNet通过密集连接促进了信息流动和特征重用，使得网络更加紧凑和表达能力更强。这些不同的连接方式在训练过程中影响了梯度传播、特征传递和网络的学习能力，从而导致了不同的训练效果和模型性能。

没有跳跃连接的卷积网络：

```
1
2 输入 -> 卷积层 -> 池化层 -> 卷积层 -> 池化层 -> 全连接层 -> 输出
```

在这种网络中，每一层的输出直接传递给下一层，没有额外的路径或连接。

ResNet：

```
1
2 输入 -> 卷积层 -> Residual连接 -> 卷积层 -> Residual连接 -> 池化层 -> 全连接层 -> 输出
```

在ResNet中，每个残差块由两个卷积层组成，并通过跳跃连接将输入直接添加到输出上。这样的跳跃连接建立了一条直接的捷径，使得梯度可以更轻松地在网络中传播。

DenseNet：

```
1
2 输入 -> 卷积层 -> 密集连接 -> 卷积层 -> 密集连接 -> 池化层 -> 全连接层 -> 输出
```

在DenseNet中，每一层的输出与后续层的输入连接在一起，形成了密集连接。这意味着每一层都可以直接访问来自前面所有层的特征图，使得特征可以更充分地传递和重用。

通过图例的比较，可以看出ResNet和DenseNet在网络结构上引入了额外的连接，这些连接使得信息能够更好地传播和利用。ResNet的跳跃连接解决了梯度消失问题，而DenseNet的密集连接促进了信息流

动和特征重用。这些连接方式的不同在训练过程中影响了梯度传播、特征传递和网络的学习能力，从而导致了不同的训练效果和模型性能。

4. 参考文献

[1]He, K., Zhang, X., Ren, S., & Sun, J. (2015). Deep Residual Learning for Image Recognition. [Online]. Available at: 1512.03385 (arXiv: cs.CV).