卷积神经网络实验报告

姓名：杨鑫　　学号：2011028

一、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

二、报告内容

- 老师提供的原始版本 CNN 网络结构（可用 print(net)打印，复制文字或截图皆可）、在 Cifar10 验证集上的训练 loss 曲线、准确度曲线图。
- 个人实现的 ResNet 网络结构在上述验证集上的训练 loss 曲线、准确度曲线图。
- 个人实现的 DenseNet 网络结构在上述验证集上的训练 loss 曲线、准确度曲线图。
- 个人实现的带有 SE 模块（Squeeze-and-Excitation Networks）的 ResNet 网络结构在上述验证集上的训练 loss 曲线、准确度曲线图。
- 解释没有跳跃连接的卷积网络、ResNet、DenseNet、SE-ResNet 在训练过程中有什么不同（重点部分）
- 格式不限

三、实验步骤与心得

1. CIFAR10 数据集

CIFAR10 数据集共有 60000 个样本，每个样本都是一张 32*32 像素的 RGB 图像（彩色图像），每个 RGB 图像又必定分为 3 个通道（R 通道、G 通道、B 通道）。这 60000 个样本被分成了 50000 个训练样本和 10000 个测试样本。
CIFAR10 数据集是用来监督学习训练的，那么每个样本就一定都配备了一个标签值（用来区分这个样本是什么），不同类别的物体用不同的标签值，CIFAR10 中有 10 类物体，标签值分别按照 0~9 来区分，他们分别是飞机（airplane）、汽车（automobile）、鸟（bird）、猫（cat）、鹿（deer）、狗（dog）、青蛙（frog）、马（horse）、船（ship）和卡车（truck）。

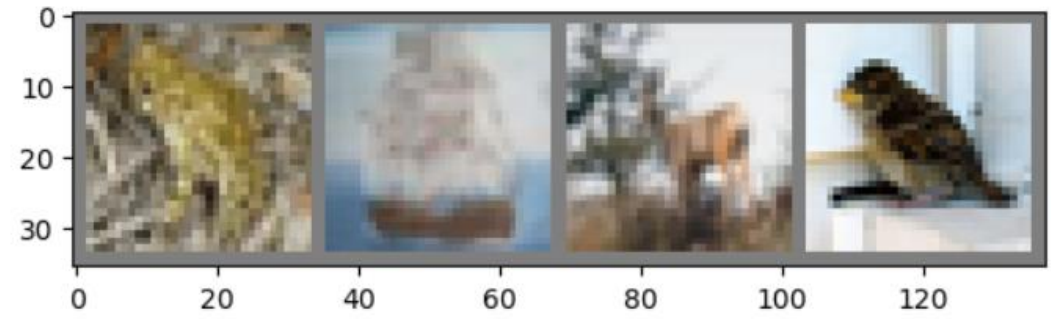
在使用 pytorch 进行 CNN 的学习的时候，可以使用 pytorch 的处理图像视频的 torchvision 工具集直接下载 AIFAR10 的训练和测试图片，torchvision 包含了一些常用的数据集、模型和转换函数等等，比如图片分类、语义切分、目标识别、实例分割、关键点检测、视频分类等工具。

```
transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

batch_size = 4

trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                        download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=batch_size,
                                          shuffle=True, num_workers=2)

testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                       download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=batch_size,
                                         shuffle=False, num_workers=2)

classes = ('plane', 'car', 'bird', 'cat',
           'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
```

✓ 41.3s Python

```
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to ./data\cifar-1
 29%|███       | 49774592/170498071 [00:10<00:30, 4005082.13it/s]
Extracting ./data\cifar-10-python.tar.gz to ./data
Files already downloaded and verified
```

然后可以打印图片和对应的标签值：




```
frog  ship  deer  bird
```

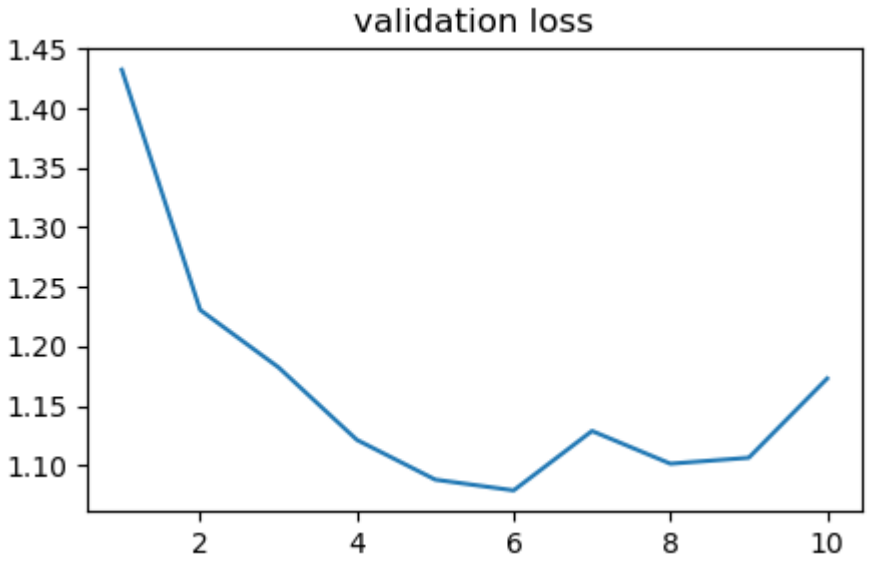
2. 原始版本 CNN 网络结构

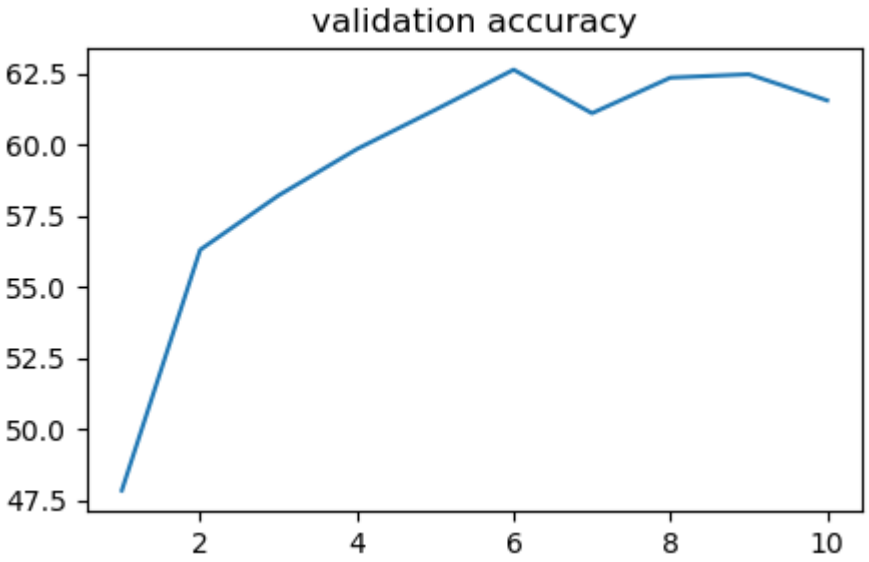
打印提供的原始 CNN 网络结构，可以发现这个简单的 CNN 结构包含以下层次：

- 输入层：输入图像大小为 3 通道，即 RGB，大小为 28*28。
- 卷积层 1：使用 5x5 的卷积核，输出 6 个通道。
- 池化层 1：使用 2x2 的最大池化操作，步长为 2。
- 卷积层 2：使用 5x5 的卷积核，输出 16 个通道。
- 全连接层 1：将输出展平成一维向量，包含 400 个神经元，输出 120 个神经元，设置偏置参数。
- 全连接层 2：输入 120 个神经元，输出 84 个神经元，设置偏置参数。
- 全连接层 3：输入 84 个神经元，输出 10 个神经元，对应 10 个类别的分类结果，设置偏置参数。

```
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)
)
```

训练得到的 loss 曲线、准确度曲线图如下：





可以看到虽然 loss 整体上在减小，但是后面当迭代次数增加后反而会有所上升。将该 CNN 网络模型保存后进行测试，从结果可以看到基本上能够分类正确，模型测试得到的准确率为 61%。

```
Accuracy of the network on the 10000 test images: 61 %
```

得到在各个分类正确率为：

```
Accuracy for class: plane is 69.1 %
Accuracy for class: car   is 80.5 %
Accuracy for class: bird  is 50.5 %
Accuracy for class: cat   is 38.2 %
Accuracy for class: deer  is 58.1 %
Accuracy for class: dog   is 47.9 %
Accuracy for class: frog  is 70.6 %
Accuracy for class: horse is 56.8 %
Accuracy for class: ship  is 70.5 %
Accuracy for class: truck is 71.7 %
```

可以看到该模型在 car、truck、ship、plane、frog 等分类上做得相对较好，而在其他 cat、dog、bird 等分类上做得相对较差，其原因可能是因为后者相比于前者在样本上的形状大小颜色方面变化很大，因此导致上面简单的卷积神经网络做的结果不好。

3. 残差网络 Resnet

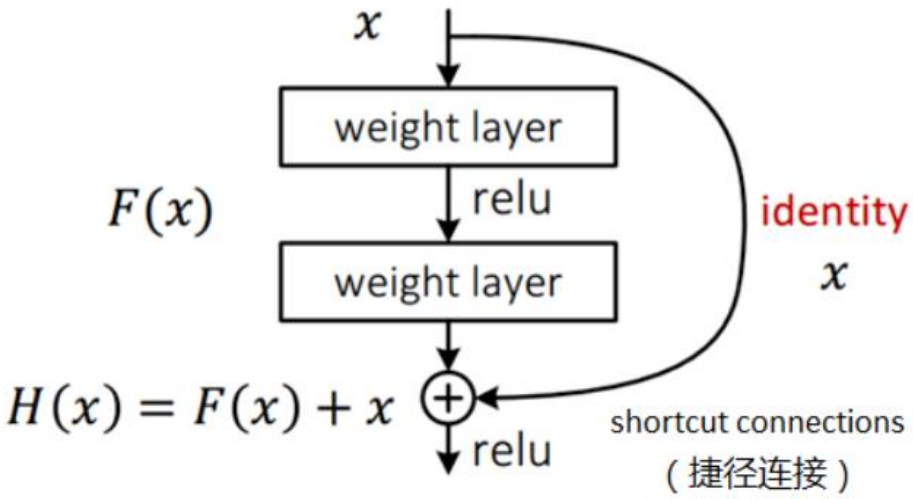
通过正则化初始化和中间的正则化层可以解决因增加网络深度导致的梯度弥散或梯度爆炸问题。但随着网络层数的增加，在训练集上的准确率却饱和甚至下降，这就是退化问题。而通过阅读论文，发现神经网络在反向传播过程中需要不断地传播梯度，而当网络层数加深时，梯度在传播过程中会逐渐消失，导致无法对前面网络层的权重进行有效的调整。因此 Resnet 的提出解决了上述当网络层数加深时候，梯度消失、模型精度下降的问题。

ResNet 的核心是残差结构，通过这种残差网络结构，将上一层的权重信息直接传递给了下一层，然后再进行梯度计算，**因此这样梯度至少保证有一个 1，从而保证了梯度的回传，网络得到更好的训练。**

也就是说：

1、残差结构能够避免普通的卷积层堆叠存在信息丢失问题，保证前向信息流的顺畅。

1、残差结构能够应对梯度反传过程中的梯度消失问题，保证反向梯度流的通顺。



1 图片来源论文《Deep Residual Learning for Image Recognition》

Resnet 的网络结构有常见的 18-layer、34-layer、50-layer 等，具体的如下图：

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
		3×3 max pool, stride 2				
conv2_x	56×56	[3×3, 64; 3×3, 64] ×2	[3×3, 64; 3×3, 64] ×3	[1×1, 64; 3×3, 64; 1×1, 256] ×3	[1×1, 64; 3×3, 64; 1×1, 256] ×3	[1×1, 64; 3×3, 64; 1×1, 256] ×3
conv3_x	28×28	[3×3, 128; 3×3, 128] ×2	[3×3, 128; 3×3, 128] ×4	[1×1, 128; 3×3, 128; 1×1, 512] ×4	[1×1, 128; 3×3, 128; 1×1, 512] ×4	[1×1, 128; 3×3, 128; 1×1, 512] ×8
conv4_x	14×14	[3×3, 256; 3×3, 256] ×2	[3×3, 256; 3×3, 256] ×6	[1×1, 256; 3×3, 256; 1×1, 1024] ×6	[1×1, 256; 3×3, 256; 1×1, 1024] ×23	[1×1, 256; 3×3, 256; 1×1, 1024] ×36
conv5_x	7×7	[3×3, 512; 3×3, 512] ×2	[3×3, 512; 3×3, 512] ×3	[1×1, 512; 3×3, 512; 1×1, 2048] ×3	[1×1, 512; 3×3, 512; 1×1, 2048] ×3	[1×1, 512; 3×3, 512; 1×1, 2048] ×3
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

2 图片来源论文《Deep Residual Learning for Image Recognition》

首先定义一个 3×3 与 1×1 的卷积操作，用于 Resnet 中的卷积操作：

```
def conv3x3(in_planes, out_planes, stride=1):
    """3x3 convolution with padding"""
    return nn.Conv2d(in_planes, out_planes, kernel_size=3,
stride=stride,padding=1, bias=False)
```

```
def conv1x1(in_planes, out_planes, stride=1):
    """1x1 convolution"""
    return nn.Conv2d(in_planes, out_planes, kernel_size=1,
stride=stride, bias=False)
```

然后定义 resnet-18 用到的 block，也就是两个 3x3 卷积：

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

    def __init__(self, inplanes, planes, stride=1,
downsample=None):
        super(BasicBlock, self).__init__()
        self.conv1 = conv3x3(inplanes, planes, stride)
        self.bn1 = nn.BatchNorm2d(planes)
        self.relu = nn.ReLU(inplace=True)  #解决梯度消失
        self.conv2 = conv3x3(planes, planes)
        self.bn2 = nn.BatchNorm2d(planes)
        self.downsample = downsample
        self.stride = stride

    def forward(self, x):
        identity = x

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

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

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

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

        return out
```

然后定义 resnet-50 以上用到的 block，主要是 1x1,3x3,1x1：

```
class Bottleneck(nn.Module):
    expansion = 4

    def __init__(self, inplanes, planes, stride=1,
downsample=None):
```

```
        super(Bottleneck, self).__init__()
        self.conv1 = conv1x1(inplanes, planes)
        self.bn1 = nn.BatchNorm2d(planes)
        self.conv2 = conv3x3(planes, planes, stride)
        self.bn2 = nn.BatchNorm2d(planes)
        self.conv3 = conv1x1(planes, planes * self.expansion)
        self.bn3 = nn.BatchNorm2d(planes * self.expansion)
        self.relu = nn.ReLU(inplace=True)
        self.downsample = downsample
        self.stride = stride

    def forward(self, x):
        identity = 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)

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

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

        return out
```

然后再定义 Resnet 残差网络模型，首先时进行一个 7*7 的卷积操作， 然后进行 3*3 最大池化操作，然后就是定义网络结构中的四个块层，里面会根据具体选择的 Resnet 模型分别填入不同的操作，最后再进行平均池化、全连接操作。

```
class ResNet(nn.Module):

    def __init__(self, block, layers, num_classes=1000):
        super(ResNet, self).__init__()
```

```
        self.inplanes = 64
        self.conv1 = nn.Conv2d(3, 64, kernel_size=7, stride=2,
padding=3,
                               bias=False)
        self.bn1 = nn.BatchNorm2d(64)
        self.relu = nn.ReLU(inplace=True)
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2,
padding=1)
        self.layer1 = self._make_layer(block, 64, layers[0])
        self.layer2 = self._make_layer(block, 128, layers[1],
stride=2)
        self.layer3 = self._make_layer(block, 256, layers[2],
stride=2)
        self.layer4 = self._make_layer(block, 512, layers[3],
stride=2)
        self.avgpool = nn.AdaptiveAvgPool2d((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')
            elif isinstance(m, nn.BatchNorm2d):
                nn.init.constant_(m.weight, 1)
                nn.init.constant_(m.bias, 0)


    def _make_layer(self, block, planes, blocks, stride=1):
        downsample = None
        if stride != 1 or self.inplanes != planes *
block.expansion:
            downsample = nn.Sequential(
                conv1x1(self.inplanes, planes * block.expansion,
stride),
                nn.BatchNorm2d(planes * block.expansion),
            )

        layers = []
        layers.append(block(self.inplanes, planes, stride,
downsample))
        self.inplanes = planes * block.expansion
        for _ in range(1, blocks):
            layers.append(block(self.inplanes, planes))
```

```
        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 = x.view(x.size(0), -1)
        x = self.fc(x)

        return x
```

然后可以根据不同 resnet 的层数选择不同的中间块:

```
def resnet18(pretrained=False, **kwargs):
    model = ResNet(BasicBlock, [2, 2, 2, 2], **kwargs)
    return model

def resnet34(pretrained=False, **kwargs):
    model = ResNet(BasicBlock, [3, 4, 6, 3], **kwargs)
    return model

def resnet50(pretrained=False, **kwargs):
    model = ResNet(Bottleneck, [3, 4, 6, 3], **kwargs)
    return model

def resnet101(pretrained=False, **kwargs):
    model = ResNet(Bottleneck, [3, 4, 23, 3], **kwargs)
    return model

def resnet152(pretrained=False, **kwargs):
    model = ResNet(Bottleneck, [3, 8, 36, 3], **kwargs)
    return model
```

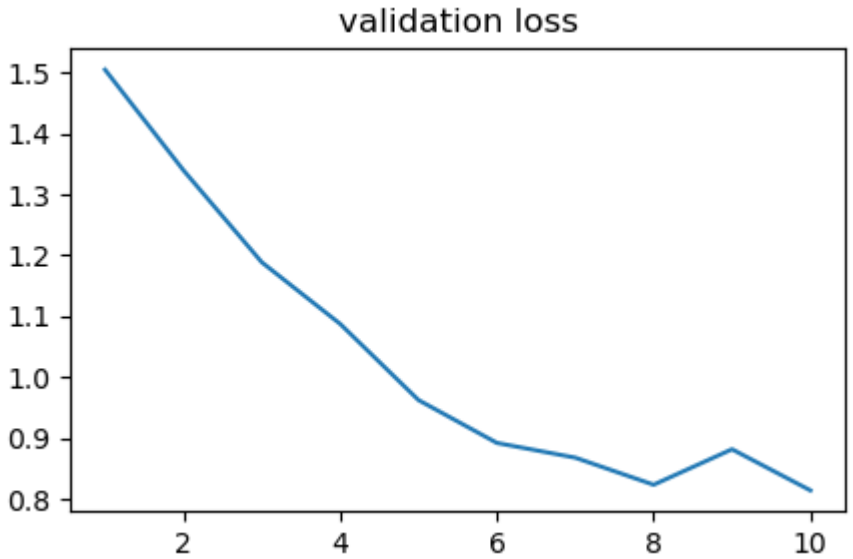
这里只对 resnt18 进行实验，打印 resnet 部分结构(太长了)可以看到，中间的层数主要是两个 basicblock 组成，且 ReLU(inplace=True)，这样就可以将上一层的权重传递到下一层。

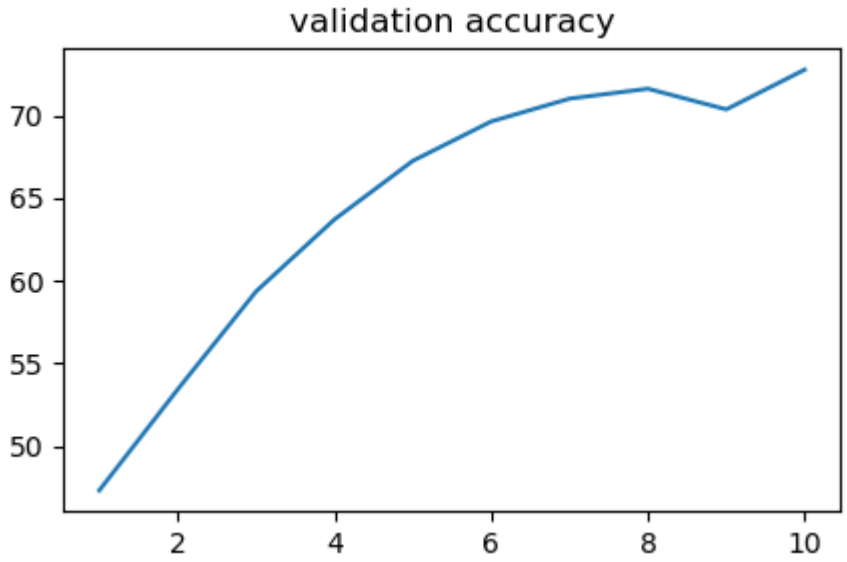
```
(conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
(bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(relu): ReLU(inplace=True)
(maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
(layer1): Sequential(
  (0): BasicBlock(
    (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  )
  (1): BasicBlock(
    (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  )
)
```

然后训练模型，这里因为本地是 CPU，因此使用了一个在线网站的 GPU 进行训练，因此 loss，accv 都是通过打印的方式进行得到的。当迭代次数为 10 次的时候训练时间大概为 51 分钟(包括训练保存的模型)，最后得到的准确率大致为 73%。

```
2023-05-07 01:34:08.399000 Train Epoch: 10 [48000/50000 (96%)]  Loss: 0.674882
2023-05-07 01:34:24.815800
2023-05-07 01:34:24.815800 Validation set: Average loss: 0.8138, Accuracy: 7276/10000 (73%)
2023-05-07 01:34:24.815900
```

得到的 validation loss 和 validation accv 图像如下：





可以看到梯度下降得很平滑，没有像普通 CNN 那种一会上升一会下降的问题。然后将得到的模型进行预测测试可以看到模型在 cifar-10 数据集上的正确率提升到了 73%。

```
Accuracy of the network on the 10000 test images: 73 %
```

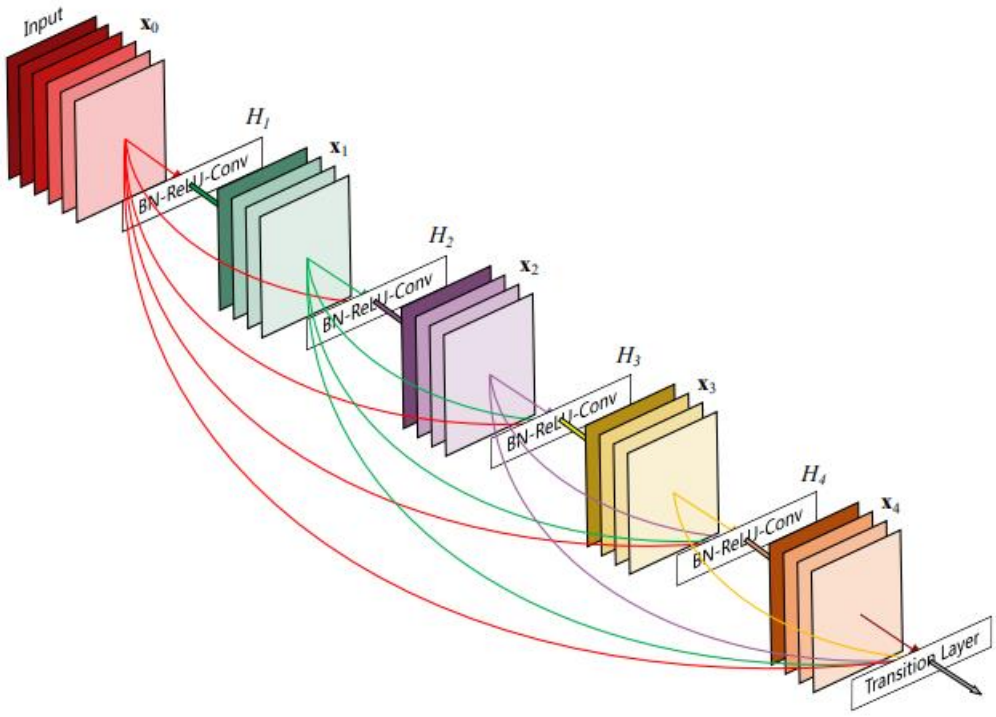
在各个类别上的预测正确率为：

```
Accuracy for class: plane is 76.2 %
Accuracy for class: car   is 89.5 %
Accuracy for class: bird  is 65.1 %
Accuracy for class: cat   is 57.8 %
Accuracy for class: deer  is 74.3 %
Accuracy for class: dog   is 58.2 %
Accuracy for class: frog  is 85.5 %
Accuracy for class: horse is 75.0 %
Accuracy for class: ship  is 81.6 %
Accuracy for class: truck is 76.6 %
over
```

可以看到对于普通 CNN 准确度不高的 cat、dog、bird 等在 Resnet 上均有提升，并且在 car、frog 等准确率达到了 85%以上。

4. DenseNet 卷积神经网络架构

在残差网络 resnet 在合并不同层的时候，用的是向量加法操作，这有可能会阻碍网络中的信息流通。因此在《Deep networks with stochastic depth》的工作中，提出了一个观点，就是 resnet 中多达上百层的卷积层对整体的效果贡献是很少的，有许多层其实是冗余的，可以在训练的过程中随机丢掉（drop），实际上整个网络并不需要那么多参数就可以表现得很好。(类似于 dropout 的做法, 只不过是 dropout 是对一个层里面的神经元输出进行丢弃, 而 stochastic depth 是对整个层进行随机丢弃, 都可以看成是一种正则手段。)作者基于这种观察，将浅层的输出合并起来，作为后面若干层的输入，使得每一层都可以利用到前几个层提取出来的特征，在传统的 CNN 中，因为是串行的连接，L 层网络就有个 L 连接，而在 DenseNet 中，因为采用了密集连结，所以 **L** 层网络有 **L(L-1)/2** 个连接，整个 Dense 连接见下图所示：



3 **图片来源论文**《Densely Connected Convolutional Networks》

DenseNet 的网络架构如下：

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			

4 **图片来源论文**《Densely Connected Convolutional Networks》

DenseNet 的网络架构由 1×1 卷积、平均池化、卷积、稠密模块（Dense Block）、稠密层（全连接层）、DropOut （丢弃层）、全局平均池化、凯明初始化、最大池化 、SoftMax 函数分类等主要部分组成。

1×1 卷积：

1×1 卷积是一种具有一些特殊性质的卷积，它可以用于降维。高效的低维嵌入（以及在卷积后应用非线性层），则会将一个输入像素及其所有通道映射到一个输出像素，该输出像素可以被压缩到所需的输出深度。它可以被视为来查看特定像素位置的多层感知器。

与 Resne 不同的是，DenseNet 在将特征传入一个层之前，不通过求和来组合这些特征。相反，而是选择通过串联来组合这些特征，因此，第 l 层有 l 的输入（由所有之前卷积块的特征图组成）。它自己的特征图被传递到所有的 **L-l 后续层**。

在具体代码中，首先定义一个 DenseLayer 稠密块，主要是由 1*1、3*3 的卷积操作组成，最重要的是定义了 drop_rate，当设置学习率的时候会选择性的

丢失一些层数:

```
class _DenseLayer(nn.Sequential):
    """Basic unit of DenseBlock (using bottleneck layer) """
    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 用来将 DenseLayer 连接起来:

```
class _DenseBlock(nn.Sequential):
    """DenseBlock"""
    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)
```

这里还要定义一个 Transition 函数用于减小特征图的尺寸，同时通过降采样减少模型的计算量，使得模型具有更好的可扩展性和泛化能力。

```
class _Transition(nn.Sequential):
    """Transition layer between two adjacent DenseBlock"""
    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))
        #cifar 的图片为 32*32，大小不用缩减
        self.add_module("pool", nn.AvgPool2d(2, stride=2,
padding=1))
```

然后就可以构造 DenseNet 网络结构了：

```
class DenseNet(nn.Module):
    "DenseNet-BC model"
    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=0.2,
num_classes=10):
        """
        :param growth_rate: (int) number of filters used in
DenseLayer, `k` in the paper
        :param block_config: (list of 4 ints) number of layers in
each DenseBlock
        :param num_init_features: (int) number of filters in the
first Conv2d
        :param bn_size: (int) the factor using in the bottleneck
layer
        :param compression_rate: (float) the compression rate used
in Transition Layer
        :param drop_rate: (float) the drop rate after each
DenseLayer
        :param num_classes: (int) number of classes for
classification
        """
        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)),
```

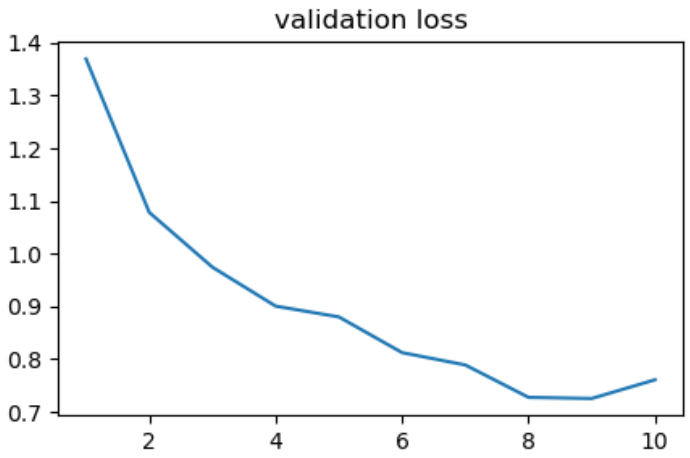
```
            ("pool0", nn.MaxPool2d(3, stride=2, padding=1))
        ]))

        # 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))

        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))

        # 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)
            elif isinstance(m, nn.BatchNorm2d):
                nn.init.constant_(m.bias, 0)
                nn.init.constant_(m.weight, 1)
            elif isinstance(m, nn.Linear):
                nn.init.constant_(m.bias, 0)

    def forward(self, x):
        features = self.features(x)
        out = self.avgpool(features)
        out = out.view(out.size(0), -1)
```

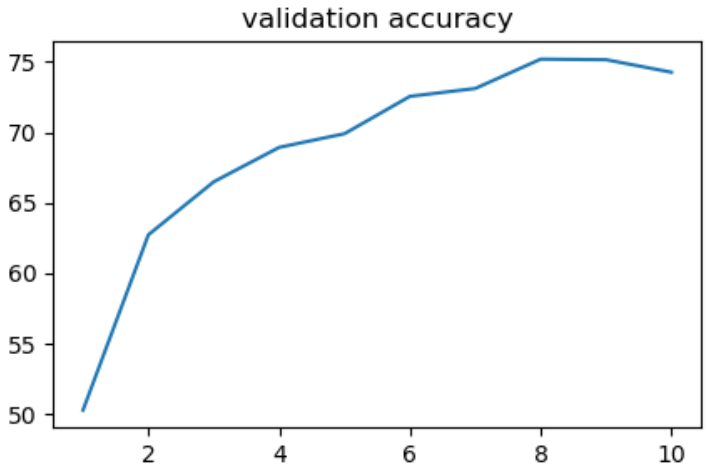
```
#         out = F.avg_pool2d(features, 7,
stride=1).view(features.size(0), -1)
        out = self.classifier(out)
        return out
```

打印网络结构如下(部分):

```
DenseNet(
  (features): Sequential(
    (conv0): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=Fa
    (norm0): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stat
    (relu0): ReLU(inplace=True)
    (pool0): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=Fals
    (denseblock1): _DenseBlock(
      (denselayer1): _DenseLayer(
        (norm1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_
        (relu1): ReLU(inplace=True)
        (conv1): Conv2d(64, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (norm2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running
        (relu2): ReLU(inplace=True)
        (conv2): Conv2d(128, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), b
      )
      (denselayer2): _DenseLayer(
        (norm1): BatchNorm2d(96, eps=1e-05, momentum=0.1, affine=True, track_running_
        (relu1): ReLU(inplace=True)
        (conv1): Conv2d(96, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (norm2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running
        (relu2): ReLU(inplace=True)
        (conv2): Conv2d(128, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), b
      )
      (denselayer3): _DenseLayer(
        (norm1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running
        (relu1): ReLU(inplace=True)
        (conv1): Conv2d(128, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (norm2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running
```

然后对模型进行训练，在 8G8 核的 GPU 上训练花费的时间比较长，大概花费了 4 小时(保存训练模型的时间)，得到的 validation loss 和 validation accv 如下图:





可以看到梯度下降得更加平滑，相比于 Resnet 训练的时间相对来说比较长。

然后将得到的模型进行预测测试可以看到模型在 cifar-10 数据集上的正确率提升到了 76%。

```
[7]  ✓ 2m 20.0s

     Accuracy of the network on the 10000 test images: 76 %
```

在各个图像分类上的准确率为：

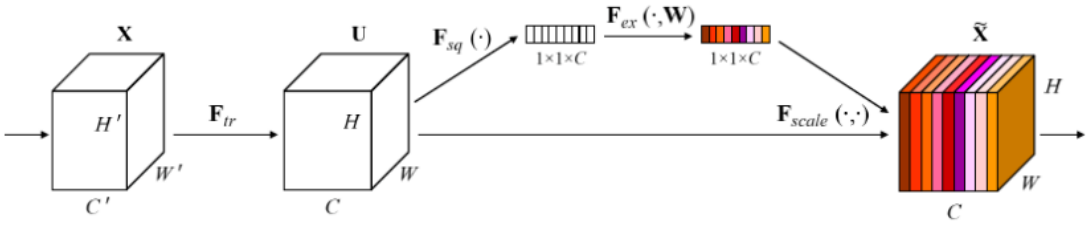
```
[28]  ✓ 2m 9.3s

...  Accuracy for class: plane is 82.5 %
     Accuracy for class: car   is 85.2 %
     Accuracy for class: bird  is 69.7 %
     Accuracy for class: cat   is 45.4 %
     Accuracy for class: deer  is 79.2 %
     Accuracy for class: dog   is 69.8 %
     Accuracy for class: frog  is 78.0 %
     Accuracy for class: horse is 81.0 %
     Accuracy for class: ship  is 86.2 %
     Accuracy for class: truck is 82.3 %
```

可以发现在 cat 上的分类准确率没有提升甚至降低了，但在其他分类准确度上表现得都比较良好基本都在 70%以上。原因可能是学习率、batch_size 以及其他超参数的选择不同导致的。

5. SE-Resnet 卷积神经网络架构

在该架构中提出了一个新的架构单元，称之为挤压和激励块(Squeeze-and-Excitation,SE block)，它通过显式建模通道之间的相互依赖性，自适应地重新校准通道方向的特征响应。这些块可以堆叠在一起，形成 SENet 架构，在不同的数据集之间非常有效地推广。SE 块在略微增加计算成本的情况下，为现有的最先进的 CNN 带来了显著的性能改进。

SE 块的结构如下，先将其先将给定信息 X 经 F 转换映射到 U，然后经过挤压操作，即**对每个通道的整个空间维度** (H×W) 进行特征聚合映射，最后经过激励层，其采用一种简单的**自选门机制**形式，将嵌入作为输入，并产生每通道**调制权值**的集合。这些**权重被应用到特征映射 U** 上，生成 SE 块的输出，该输出可以直接输入到网络的后续层。



A Squeeze-and-Excitation block

5 **图片来源论文**《Squeeze-and-Excitation Networks》

可以通过简单地堆叠 SE 块的集合来构建 SE 网络(SENet)。此外，这些 SE 块也可以作为一个插入式替换原始块。但是模块在整个网络的不同深度上所扮演的角色是不同的。在早期的层中它以一种与类无关的方式激发信息特征，加强共享的低级表示，在后面的层中 SE 块变得越来越专门化，并以一种高度特定于类的方式响应不同的输入。因此，特征重新校准的好处得以实现。

SE 嵌入模型框架如下：

Output size	ResNet-50	SE-ResNet-50	SE-ResNeXt-50 (32 × 4d)
112 × 112	conv, 7 × 7, 64, stride 2		
56 × 56	max pool, 3 × 3, stride 2		
	$\begin{bmatrix} \text{conv}, 1\times1, 64 \\ \text{conv}, 3\times3, 64 \\ \text{conv}, 1\times1, 256 \end{bmatrix} \times 3$	$\begin{bmatrix} \text{conv}, 1\times1, 64 \\ \text{conv}, 3\times3, 64 \\ \text{conv}, 1\times1, 256 \\ fc, [16, 256] \end{bmatrix} \times 3$	$\begin{bmatrix} \text{conv}, 1\times1, 128 \\ \text{conv}, 3\times3, 128 \quad C=32 \\ \text{conv}, 1\times1, 256 \\ fc, [16, 256] \end{bmatrix} \times 3$
28 × 28	$\begin{bmatrix} \text{conv}, 1\times1, 128 \\ \text{conv}, 3\times3, 128 \\ \text{conv}, 1\times1, 512 \end{bmatrix} \times 4$	$\begin{bmatrix} \text{conv}, 1\times1, 128 \\ \text{conv}, 3\times3, 128 \\ \text{conv}, 1\times1, 512 \\ fc, [32, 512] \end{bmatrix} \times 4$	$\begin{bmatrix} \text{conv}, 1\times1, 256 \\ \text{conv}, 3\times3, 256 \quad C=32 \\ \text{conv}, 1\times1, 512 \\ fc, [32, 512] \end{bmatrix} \times 4$
14 × 14	$\begin{bmatrix} \text{conv}, 1\times1, 256 \\ \text{conv}, 3\times3, 256 \\ \text{conv}, 1\times1, 1024 \end{bmatrix} \times 6$	$\begin{bmatrix} \text{conv}, 1\times1, 256 \\ \text{conv}, 3\times3, 256 \\ \text{conv}, 1\times1, 1024 \\ fc, [64, 1024] \end{bmatrix} \times 6$	$\begin{bmatrix} \text{conv}, 1\times1, 512 \\ \text{conv}, 3\times3, 512 \quad C=32 \\ \text{conv}, 1\times1, 1024 \\ fc, [64, 1024] \end{bmatrix} \times 6$
7×7	$\begin{bmatrix} \text{conv}, 1\times1, 512 \\ \text{conv}, 3\times3, 512 \\ \text{conv}, 1\times1, 2048 \end{bmatrix} \times 3$	$\begin{bmatrix} \text{conv}, 1\times1, 512 \\ \text{conv}, 3\times3, 512 \\ \text{conv}, 1\times1, 2048 \\ fc, [128, 2048] \end{bmatrix} \times 3$	$\begin{bmatrix} \text{conv}, 1\times1, 1024 \\ \text{conv}, 3\times3, 1024 \quad C=32 \\ \text{conv}, 1\times1, 2048 \\ fc, [128, 2048] \end{bmatrix} \times 3$
1 × 1	global average pool, 1000-d fc, softmax		

6 图片来源论文《Squeeze-and-Excitation Networks》

在动手实践时，首先要实现一个 Conv 卷积专门用来进行卷积操作：

```
def conv_block(in_channel, out_channel, relu_last=True, **kwargs):
    layers = [nn.Conv2d(in_channel, out_channel, bias=False,
**kwargs),
              nn.BatchNorm2d(out_channel)]
    if relu_last:
        layers.append(nn.ReLU(inplace=True))

    return nn.Sequential(*layers)
```

然后定义 ResidualSEBlock 用来进行中间的挤压与激励处理。

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

    def __init__(self, in_channel, out_channel, stride, r=16):
        super(ResidualSEBlock, self).__init__()
        self.residual = nn.Sequential(
            conv_block(in_channel, out_channel, kernel_size=3,
stride=stride, padding=1),
            conv_block(out_channel, out_channel * self.expansion,
kernel_size=3, padding=1)
        )
        self.shortcut = nn.Sequential()
        if stride != 1 or in_channel != out_channel *
self.expansion:
            self.shortcut = conv_block(in_channel, out_channel *
self.expansion, kernel_size=1, stride=stride,
                                       relu_last=False)
        self.squeeze = nn.AdaptiveAvgPool2d(1)
```

```
        self.excitation = nn.Sequential(
            nn.Linear(out_channel * self.expansion, out_channel *
self.expansion // r),
            nn.ReLU(inplace=True),
            nn.Linear(out_channel * self.expansion // r,
out_channel * self.expansion),
            nn.Sigmoid())

    def forward(self, x):
        r = self.residual(x)
        bs, c, _, _ = r.shape
        s = self.squeeze(r).view(bs, c)
        e = self.excitation(s).view(bs, c, 1, 1)
        return F.relu(self.shortcut(x) + r * e.expand_as(r))
```

最后将模型组装起来，最后进行平均池化、全连接操作。

```
class SEResnet(nn.Module):
    def __init__(self, in_channel, n_classes, num_blocks, block):
        super(SEResnet, self).__init__()
        self.in_channels = 64
        self.feature = nn.Sequential(
            conv_block(in_channel, 64, kernel_size=3, padding=1),
            self._make_stage(64, 1, num_blocks[0], block),
            self._make_stage(128, 2, num_blocks[1], block),
            self._make_stage(256, 2, num_blocks[2], block),
            self._make_stage(512, 2, num_blocks[3], block)
        )
        self.classifier = nn.Sequential(
            nn.AdaptiveAvgPool2d((1, 1)),
            nn.Flatten(),
            nn.Linear(self.in_channels, n_classes),
            nn.LogSoftmax(dim=1)
        )

    def _make_stage(self, out_channel, stride, num_block, block):
        layers = []
        for i in range(num_block):
            stride = stride if i == 0 else 1
            layers.append(block(self.in_channels, out_channel,
stride))
            self.in_channels = out_channel * block.expansion
        return nn.Sequential(*layers)

    def forward(self, x):
```

```
        return self.classifier(self.feature(x))
```

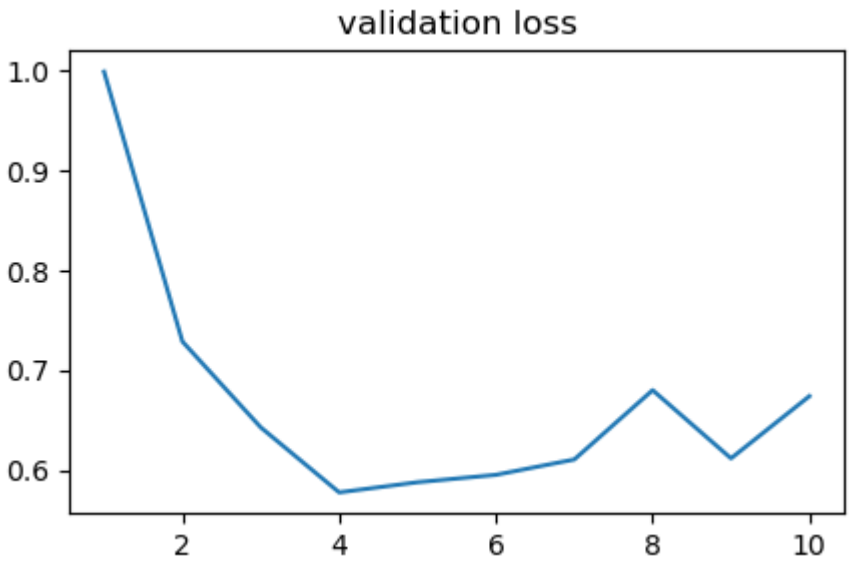
这里为了减少训练时间，因此只定义了 SE-Resnet18 以及 SE-Resnet34 模型：

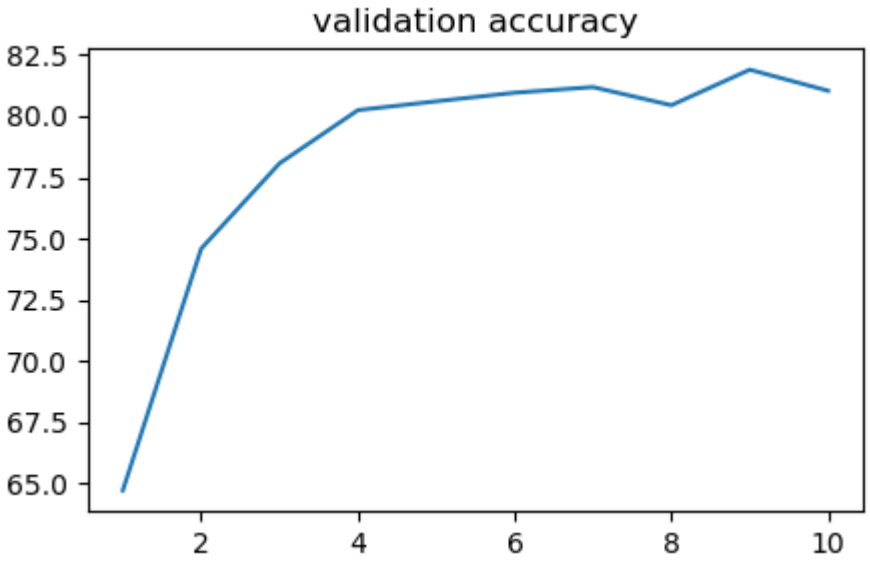
```
def seresnet18():
    return SEResnet(3, 10, [2, 2, 2, 2], ResidualSEBlock)

def seresnet34():
    return SEResnet(3, 10, [3, 4, 6, 3], ResidualSEBlock)
```

对模型进行迭代 10 次训练，这里学习率等超参数采用与普通 CNN 一致的数值，方便进行比较。在 8 核 8G 的 GPU 上训练了大概 80 分钟。可以看到最后得到的 loss 很小，就可以看出该模型应该具有很好的预测结果。

```
2023-05-08 23:59:20.048700 [9,  8000] loss: 0.018
2023-05-08 23:59:59.131700 [9, 10000] loss: 0.020
2023-05-09 00:00:38.202800 [9, 12000] loss: 0.022
2023-05-09 00:01:27.592300 [10,  2000] loss: 0.009
2023-05-09 00:02:10.613100 [10,  4000] loss: 0.013
2023-05-09 00:02:55.703700 [10,  6000] loss: 0.016
2023-05-09 00:03:37.085500 [10,  8000] loss: 0.010
2023-05-09 00:04:18.525900 [10, 10000] loss: 0.011
2023-05-09 00:04:57.826800 [10, 12000] loss: 0.011
2023-05-09 00:05:08.209100 Finished Training
2023-05-09 00:05:09.290300 SYSTEM: Finishing...
```

得到的 validation loss 和 validation accv 图像如下：





可以看到训练过程中 loss 很大，但是 accv 却随着迭代次数的增加逐渐变高。

然后将得到的模型进行预测测试可以看到模型在 cifar-10 数据集上的正确率提升到了 81%。

```
5]  ✓ 2m 46.5s
    Accuracy of the network on the 10000 test images: 81 %
```

在各个分类上的准确率为：

```
    Accuracy for class: plane is 83.3 %
    Accuracy for class: car   is 81.1 %
    Accuracy for class: bird  is 70.5 %
    Accuracy for class: cat   is 67.1 %
    Accuracy for class: deer  is 85.5 %
    Accuracy for class: dog   is 72.0 %
    Accuracy for class: frog  is 84.6 %
    Accuracy for class: horse is 85.2 %
    Accuracy for class: ship  is 89.9 %
    Accuracy for class: truck is 94.2 %
```

可以看到准确率很高，有些都已经达到了 90%以上。

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

正如在上述残差网络结构中提到的一样，残差网络在每两个卷积层之间加入

了短路链接，这种操作避免了梯度消失的问题。于此同时在训练过程中某些卷积层可能已经达到了最优解，此时没有再进行训练的必要。因此通过残差的这种方式我们可以保证再接下来的训练中梯度不会消失的问题，即不会破坏之前训练出来的最优解(这些都已经在上述报告中说明了)。

对于没有跳跃链接的卷积网络，通常是由多个卷积层和池化层组成，中间没有跳跃连接，每个卷积层都将前一层的输出作为输入进行计算。这种网络通常比较浅，当网络深度加大的时候容易出现梯度消失或梯度爆炸等问题。

对于 Resnet 架构中，使用了残差块来解决深度卷积神经网络中的梯度消失和梯度爆炸问题。每个残差块包含两个卷积层和一个跳跃连接，其中跳跃连接将前一层的输出直接传递给后面的卷积层，使得梯度能够更好地传播。ResNet 相比于没有跳跃连接的卷积网络能够更容易地训练深层网络。

DenseNet 使用密集连接的层来构建网络。在每个密集连接的层中，当前层的输出会被传递给后面所有的层作为输入。这种设计使得 DenseNet 具有更高的参数利用率和更好的梯度流动性，能够更好地处理梯度消失和梯度爆炸问题。

SE-ResNet 在 ResNet 的基础上增加了一种称为 Squeeze-and-Excitation（SE）模块的设计。SE 模块能够自适应地调整每个通道的权重，使得网络能够更好地关注重要的特征。这种设计使得 SE-ResNet 相比于 ResNet 在一些图像分类任务上表现更好。

四、 参考文献

[1] Hu, J., Shen, L., & Sun, G. (2018). Squeeze-and-excitation networks. In *Proceedings*

of the IEEE conference on computer vision and pattern recognition (pp. 7132-7141).
[2] Huang, G., Liu, Z., Van Der Maaten, L., & Weinberger, K. Q. (2017). Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 4700-4708).
[3] He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 770-778).