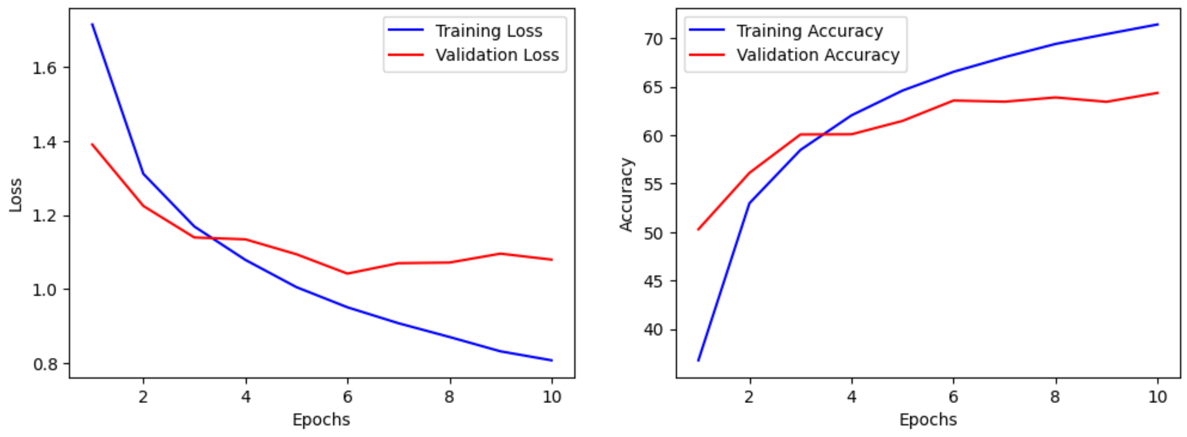
深度学习报告w3：卷积神经网络

1 原模型复现

原模型是一个两层卷积加一个多层感知机，结构如下：

```
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)
)
```

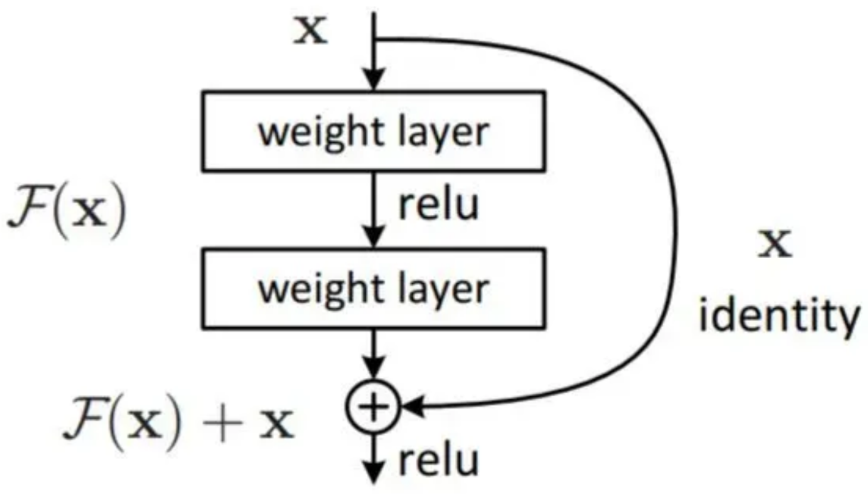
复现结果如下，epoch=10，最终在10000张测试集上取得56%的acc。



可以看出最后一个阶段有轻微的过拟合嫌疑

2 ResNet

我认为理解ResNet的第一步是理解为何残差块能够work，结构如下：


x
weight layer
$\mathcal{F}(\mathbf{x})$
relu
weight layer
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

梯度消失：我们知道神经网络在进行反向传播(BP算法)的时候会对参数W进行更新，梯度消失就是靠后面网络层(如layer3)能够正常的得到一个合理的偏导数，但是靠近输入层的网络层，计算的到的偏导数近乎零，W几乎无法得到更新。

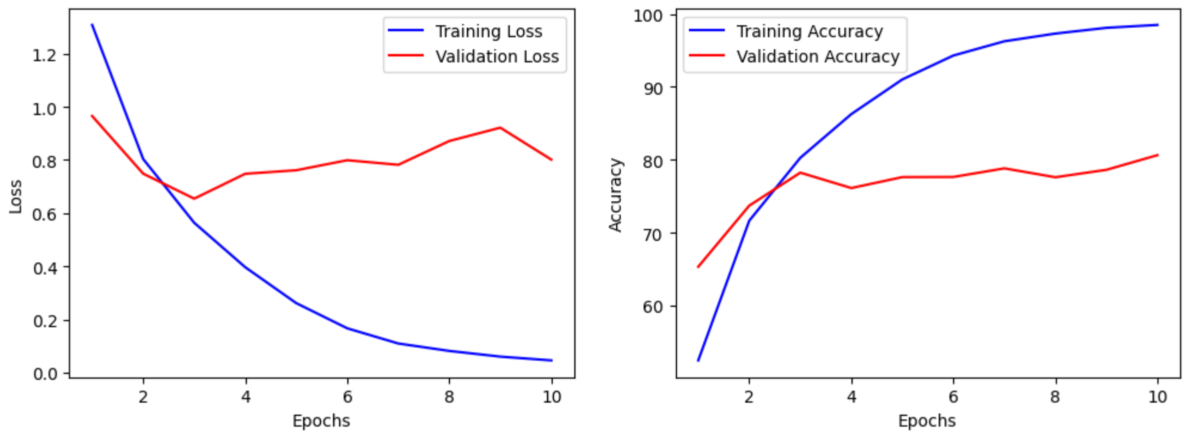
原因：反向传播的时候的链式法则，越是浅层的网络，其梯度表达式可以展现出来连乘的形式，而这样如果都是小于1的，这样的话，浅层网络参数值的更新就会变得很慢，这就导致了深层网络的学习就等价于了只有后几层的浅层网络的学习了。

梯度爆炸：靠近输入层的网络层，计算的到的偏导数极其大，更新后W变成一个很大的数(爆炸)。

原因：类似于上述的原因，假如都是大于1的时候，那么浅层的网络的梯度过大，更新的参数变量也过大，所以无论是梯度消失还是爆炸都是训练过程会十分曲折的，都应该尽可能避免。

```
# 定义残差块
class ResidualBlock(nn.Module):
    def __init__(self, in_channels, out_channels, stride=1, downsample=None):
        super(ResidualBlock, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3,
stride=stride, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3,
stride=1, padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(out_channels)
        self.downsample = downsample

    def forward(self, x):
        residual = x
        out = F.relu(self.bn1(self.conv1(x)))
        out = self.bn2(self.conv2(out))
        if self.downsample:
            residual = self.downsample(x)
        out += residual
        out = F.relu(out)
        return out

# 定义ResNet
class ResNet(nn.Module):
    def __init__(self, block, layers, num_classes=10):
        super(ResNet, self).__init__()
        self.in_channels = 64
        self.conv = nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1,
bias=False)
        self.bn = nn.BatchNorm2d(64)
        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.fc = nn.Linear(512, num_classes)

    def make_layer(self, block, out_channels, blocks, stride=1):
        downsample = None
        if stride != 1 or self.in_channels != out_channels:
            downsample = nn.Sequential(
                nn.Conv2d(self.in_channels, out_channels, kernel_size=1,
stride=stride, bias=False),
                nn.BatchNorm2d(out_channels))
        layers = []
        layers.append(block(self.in_channels, out_channels, stride, downsample))
        self.in_channels = out_channels
        for _ in range(1, blocks):
            layers.append(block(out_channels, out_channels))
        return nn.Sequential(*layers)

    def forward(self, x):
        out = F.relu(self.bn(self.conv(x)))
        out = self.layer1(out)
        out = self.layer2(out)
        out = self.layer3(out)
        out = self.layer4(out)
        out = F.avg_pool2d(out, 4)
        out = out.view(out.size(0), -1)
        out = self.fc(out)
        return out

def ResNet18():
    return ResNet(ResidualBlock, [2, 2, 2, 2])
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
net = ResNet18().to(device)
print(net)
```

训练模型为ResNet18，训练同样为10轮，结果如下：



可以看出准确率有大幅度上升，最终在10000张测试集上取得80%的acc。

3 Res2Net

ResNet+多尺度，结构图如下：https://arxiv.org/pdf/1904.01169

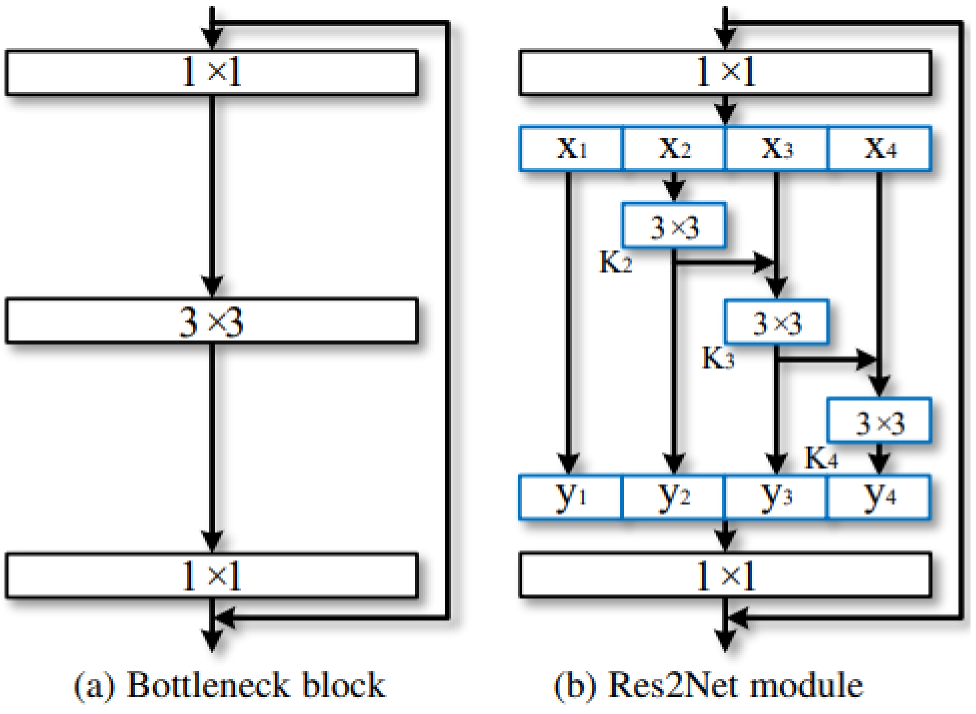
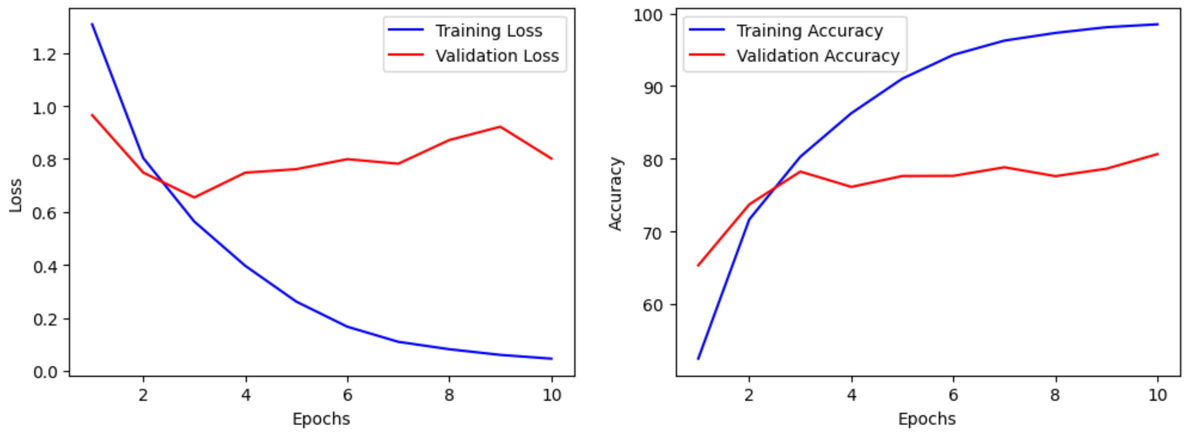


Fig. 2. Comparison between the bottleneck block and the proposed Res2Net module (the scale dimension $s = 4$).

在残差块中加入了多尺度结构，我的实现如下：

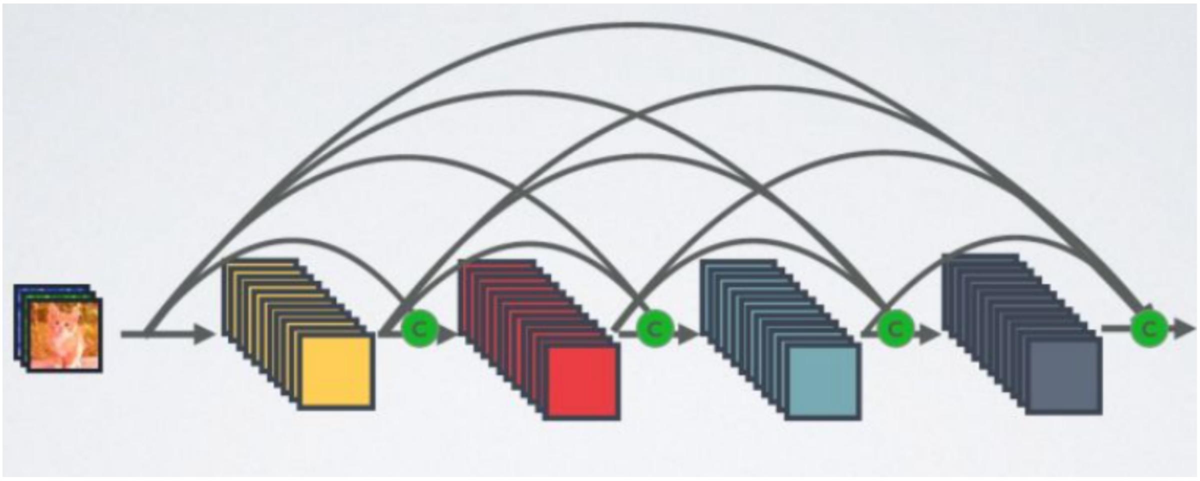
```
# 定义Res2Net残差块
class Res2NetBlock(nn.Module):
    def __init__(self, in_channels, out_channels, stride=1, scales=4,
base_width=26):
        super(Res2NetBlock, self).__init__()
        self.scales = scales
        width = out_channels // scales

        self.conv1 = nn.Conv2d(in_channels, width * scales, kernel_size=1,
stride=1, bias=False)
        self.bn1 = nn.BatchNorm2d(width * scales)

        self.convs = nn.ModuleList([
            nn.Conv2d(width, width, kernel_size=3, stride=stride, padding=1,
bias=False) if i == 0
            else nn.Conv2d(width, width, kernel_size=3, stride=1, padding=1,
bias=False)
            for i in range(scales - 1)
        ])
        self.bns = nn.ModuleList([nn.BatchNorm2d(width) for _ in range(scales -
1)])

        self.conv3 = nn.Conv2d(width * scales, out_channels, kernel_size=1,
stride=1, bias=False)
        self.bn3 = nn.BatchNorm2d(out_channels)

        self.relu = nn.ReLU(inplace=True)

        self.downsample = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, kernel_size=1, stride=stride,
bias=False),
            nn.BatchNorm2d(out_channels)
        ) if stride != 1 or in_channels != out_channels else None

    def forward(self, x):
        residual = x

        out = self.relu(self.bn1(self.conv1(x)))
        spx = torch.split(out, out.size(1) // self.scales, 1)
        out = spx[0]
        for i in range(1, self.scales):
            sp = spx[i]
            if i > 1:
                sp = sp.clone() + spx[i - 1]
            sp = self.relu(self.bns[i-1](self.convs[i-1](sp)))
            out = torch.cat((out, sp), 1)

        out = self.bn3(self.conv3(out))
        if self.downsample is not None:
            residual = self.downsample(x)
        out += residual
        out = self.relu(out)
        return out

# 定义Res2Net
class Res2Net(nn.Module):
    def __init__(self, block, layers, num_classes=10):
        super(Res2Net, self).__init__()
        self.in_channels = 64
        self.conv = nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1,
bias=False)
        self.bn = nn.BatchNorm2d(64)
        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.fc = nn.Linear(512, num_classes)

    def make_layer(self, block, out_channels, blocks, stride=1):
        layers = []
        layers.append(block(self.in_channels, out_channels, stride))
        self.in_channels = out_channels
        for _ in range(1, blocks):
            layers.append(block(out_channels, out_channels))
        return nn.Sequential(*layers)

    def forward(self, x):
        out = F.relu(self.bn(self.conv(x)))
        out = self.layer1(out)
        out = self.layer2(out)
        out = self.layer3(out)
        out = self.layer4(out)
        out = F.avg_pool2d(out, 4)
        out = out.view(out.size(0), -1)
        out = self.fc(out)
        return out

def Res2Net18():
    return Res2Net(Res2NetBlock, [2, 2, 2, 2])

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
net = Res2Net18().to(device)
print(net)
```



训练结果略高于resnet

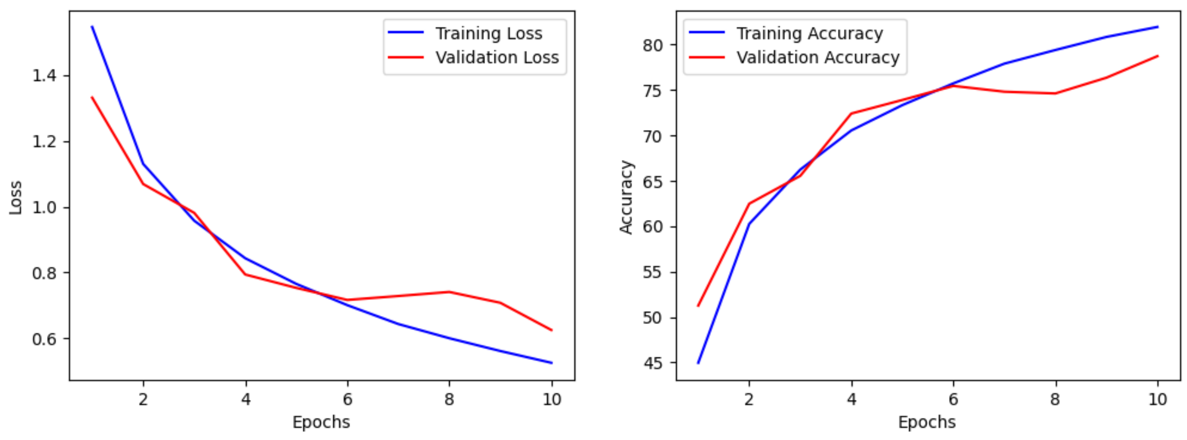
4 DenseNet

DenseNethttps://arxiv.org/pdf/1608.06993.pdf，不同于resnet，densenet采用稠密连接。DenseNet由多个Dense Block结构组成，它的核心思想就是后面每个层都会接受其前面所有层作为其额外的输入，可以看到最后Transition Layer处，所有颜色的线都有，也就是前面所有层都作为输入。



我的实现如下：

```
import math
# 定义DenseNet块
class Bottleneck(nn.Module):
    def __init__(self, in_channels, growth_rate):
        super(Bottleneck, self).__init__()
        self.bn1 = nn.BatchNorm2d(in_channels)
        self.conv1 = nn.Conv2d(in_channels, 4 * growth_rate, kernel_size=1,
bias=False)
        self.bn2 = nn.BatchNorm2d(4 * growth_rate)
        self.conv2 = nn.Conv2d(4 * growth_rate, growth_rate, kernel_size=3,
padding=1, bias=False)

    def forward(self, x):
        out = self.conv1(F.relu(self.bn1(x)))
        out = self.conv2(F.relu(self.bn2(out)))
        out = torch.cat([out, x], 1)
        return out

class Transition(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(Transition, self).__init__()
        self.bn = nn.BatchNorm2d(in_channels)
        self.conv = nn.Conv2d(in_channels, out_channels, kernel_size=1,
bias=False)
        self.avg_pool = nn.AvgPool2d(2)

    def forward(self, x):
        out = self.conv(F.relu(self.bn(x)))
        out = self.avg_pool(out)
        return out

class DenseNet(nn.Module):
    def __init__(self, block, nblocks, growth_rate=12, reduction=0.5,
num_classes=10):
        super(DenseNet, self).__init__()
        self.growth_rate = growth_rate
        num_planes = 2 * growth_rate

        self.conv1 = nn.Conv2d(3, num_planes, kernel_size=3, padding=1,
bias=False)

        self.dense1 = self._make_dense_layers(block, num_planes, nblocks[0])
        num_planes += nblocks[0] * growth_rate
        out_planes = int(math.floor(num_planes * reduction))
        self.trans1 = Transition(num_planes, out_planes)
        num_planes = out_planes

        self.dense2 = self._make_dense_layers(block, num_planes, nblocks[1])
        num_planes += nblocks[1] * growth_rate
        out_planes = int(math.floor(num_planes * reduction))
        self.trans2 = Transition(num_planes, out_planes)
        num_planes = out_planes

        self.dense3 = self._make_dense_layers(block, num_planes, nblocks[2])
        num_planes += nblocks[2] * growth_rate
        out_planes = int(math.floor(num_planes * reduction))
        self.trans3 = Transition(num_planes, out_planes)
        num_planes = out_planes

        self.dense4 = self._make_dense_layers(block, num_planes, nblocks[3])
        num_planes += nblocks[3] * growth_rate

        self.bn = nn.BatchNorm2d(num_planes)
        self.linear = nn.Linear(num_planes, num_classes)
    def _make_dense_layers(self, block, in_channels, nblock):
        layers = []
        for _ in range(nblock):
            layers.append(block(in_channels, self.growth_rate))
            in_channels += self.growth_rate
        return nn.Sequential(*layers)

    def forward(self, x):
        out = self.conv1(x)
        out = self.trans1(self.dense1(out))
        out = self.trans2(self.dense2(out))
        out = self.trans3(self.dense3(out))
        out = self.dense4(out)
        out = F.avg_pool2d(F.relu(self.bn(out)), 4)
        out = out.view(out.size(0), -1)
        out = self.linear(out)
        return out

def DenseNet4():
    return DenseNet(Bottleneck, [1, 1, 1, 1], growth_rate=32)

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
net = DenseNet4().to(device)  # 将模型移动到GPU
print(net)
```



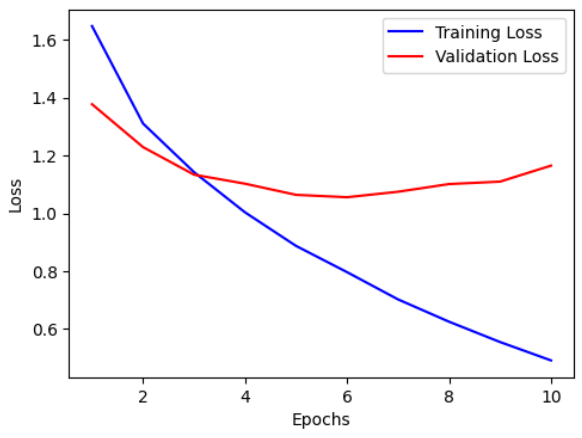
训练设置不变，最终达到78%的acc。

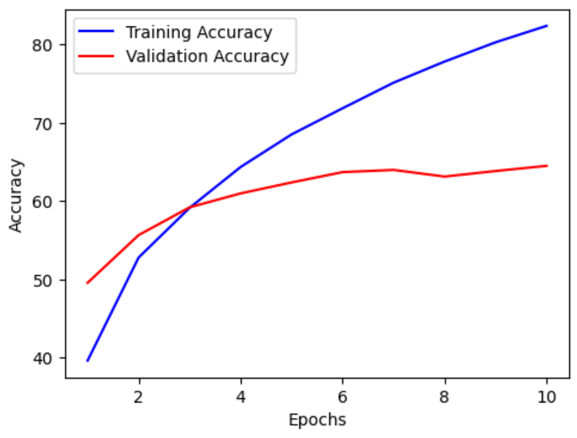
5 MobileNet with denseconv

我的实现如下：

```
# 定义深度可分离卷积
class DepthwiseSeparableConv(nn.Module):
    def __init__(self, in_channels, out_channels, stride=1):
        super(DepthwiseSeparableConv, self).__init__()
        self.depthwise = nn.Conv2d(in_channels, in_channels, kernel_size=3,
stride=stride, padding=1, groups=in_channels, bias=False)
        self.pointwise = nn.Conv2d(in_channels, out_channels, kernel_size=1,
stride=1, bias=False)
        self.bn = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU(inplace=True)

    def forward(self, x):
        out = self.depthwise(x)
        out = self.pointwise(out)
        out = self.bn(out)
        out = self.relu(out)
        return out

# 定义MobileNet
class MobileNet(nn.Module):
    def __init__(self, num_classes=10):
        super(MobileNet, self).__init__()
        self.conv1 = nn.Conv2d(3, 32, kernel_size=3, stride=2, padding=1,
bias=False)
        self.bn1 = nn.BatchNorm2d(32)
        self.relu = nn.ReLU(inplace=True)

        self.layers = self._make_layers(in_channels=32)

        self.avg_pool = nn.AdaptiveAvgPool2d((1, 1))
        self.fc = nn.Linear(1024, num_classes)

    def _make_layers(self, in_channels):
        layers = []
        cfg = [
            # (out_channels, stride)
            (64, 1),
            (128, 2),
            (128, 1),
            (256, 2),
            (256, 1),
            (512, 2),
            (512, 1),
            (512, 1),
            (512, 1),
            (512, 1),
            (512, 1),
            (1024, 2),
            (1024, 1),
        ]

        for out_channels, stride in cfg:
            layers.append(DepthwiseSeparableConv(in_channels, out_channels,
stride))
            in_channels = out_channels

        return nn.Sequential(*layers)

    def forward(self, x):
        out = self.relu(self.bn1(self.conv1(x)))
        out = self.layers(out)
        out = self.avg_pool(out)
        out = out.view(out.size(0), -1)
        out = self.fc(out)
        return out

# 检查是否可以使用GPU
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

```
print('Using device:', device)

# 实例化MobileNet
net = MobileNet().to(device)
print(net)
```


Training Loss
Validation Loss
Loss
Epochs


Training Accuracy
Validation Accuracy
Accuracy
Epochs