南开大学

Nankai University

南　　开　　大　　学

计　算　机　学　院

深度学习及应用实验报告

实验二　卷积神经网络的实现

姓名：苏航

学号：2111039

专业：信息安全

2024 年 5 月 29 日

摘要

本实验在理解基本卷积神经网络的基础上, 实现了 CNN 对 CIFAR10 数据集的识别. 同时我们也尝试了对基本神经网络的改进, 实现了 ResNet,DenseNet, 以及带 SE 结构的 RESNet, 实现了较高正确率的识别模型

关键字：卷积神经网络;ResNet;DenseNet;SE 结构

目录

一、 实验要求

本次实验要求掌握卷积神经网络（CNN）的基本原理并学会使用 PyTorch 搭建简单的 CNN 实现 CIFAR10 数据集分类, 同时掌握如何改进网络结构、调试参数以提升网络识别性能.

二、 实验内容

本实验要求根据原始版本卷积网络结构, 以及自己实现的 ResNet,DenseNet, 以及带 SE 结构的 ResNet 在 CIFAR10 数据集上的训练 LOSS 曲线以及准确率曲线

三、 实验过程

(一) 卷积神经网络的基本原理

卷积神经网络由卷积层, 激活函数, 池化层, 全连接层, 以及损失函数构成。其中卷积是其核心操作, 通过卷积操作, 提取输入数据的特征。卷积操作可以理解为在输入数据上滑动一个称为卷积核的小窗口，然后对窗口中的数据进行加权求和，得到输出特征图。具体的模型结构如下所示:

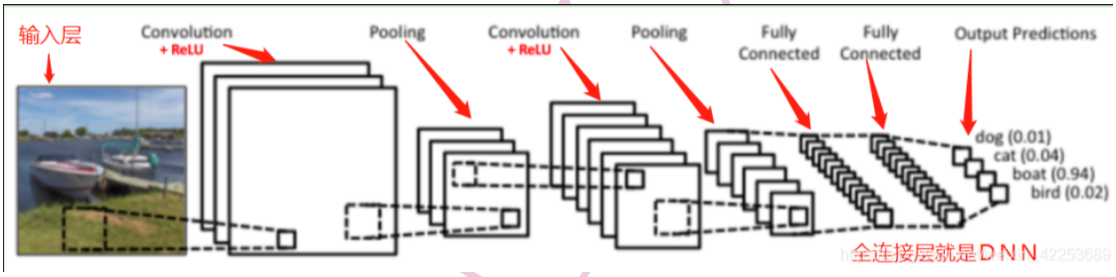


图 1: 卷积神经网络模型图

(二) 原始 CNN 的实现

实验要求我们实现简单的 CNN 网络结构, 如下所示. 对于这个 CNN 我们首先经过两次卷积操作, 后接最大池化层, 然后再接上 3 层全连接层. 具体来讲首先定义了一个卷积层，输入通道数为 3，输出通道数为 6，卷积核大小为 5x5; 然后定义了一个最大池化层，池化窗口大小为 2x2; 定义了另一个卷积层，输入通道数为 6，输出通道数为 16，卷积核大小为 5x5; 再然后就是输入特征数为 400，输出特征数为 120 的全连接层; 输入特征数为 120，输出特征数为 84 的全连接层; 输入特征数为 84，输出特征数为 10 的全连接层

```
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 6, 5) # 4, 6, 28, 28
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

```

```
    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = torch.flatten(x, 1) # flatten all dimensions except batch
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

搭建成功模型后我们可以把它输出出来:

```
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)
)
```

图 2: 原始 CNN 网络结构图

由此可见该网络的结构还是非常简单的

(三) 原始 CNN 的训练与评估

搭建完基本的 MLP 后我们就可以开始对其展开训练与评估, 实验要求我们采用 CIFAR10 数据集, 在进行训练之前我们需要对其进行预处理, 代码如下:

```
transform = transforms.Compose([
        transforms.RandomCrop(32, padding=4),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994,
            0.2010)),
    ])
trainset = torchvision.datasets.CIFAR10(root='./data', train=True, download=
    True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=128, 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=100, shuffle=
    False, num_workers=2)
```

训练函数也很简单, 具体如下:

```
def train(model, criterion, optimizer, trainloader, valloader, epochs):
    train_loss_history = []
    val_loss_history = []
```

```
    val_accuracy_history = []

    for epoch in range(epochs):
        model.train()
        running_loss = 0.0
        for i, data in enumerate(trainloader, 0):
            inputs, labels = data
            optimizer.zero_grad()

            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            running_loss += loss.item()
            if i % 2000 == 1999:
                print('[%d, %5d] training loss: %.3f' %
                      (epoch + 1, i + 1, running_loss / 2000))
                running_loss = 0.0

        # 计算在验证集上的loss和准确度
        model.eval()
        val_running_loss = 0.0
        correct = 0
        total = 0
        with torch.no_grad():
            for data in valloader:
                images, labels = data
                outputs = model(images)
                val_loss = criterion(outputs, labels)
                val_running_loss += val_loss.item()
                _, predicted = torch.max(outputs, 1)
                total += labels.size(0)
                correct += (predicted == labels).sum().item()

        train_loss_history.append(running_loss / len(trainloader))
        val_loss_history.append(val_running_loss / len(valloader))
        val_accuracy_history.append(100 * correct / total)

        print('Epoch %d: Validation accuracy %d %%' % (epoch + 1, 100 *
            correct / total))
    PATH = './cifar_10_net.pth'
    torch.save(net.state_dict(), PATH)

    return train_loss_history, val_loss_history, val_accuracy_history
```

实现该网络的训练和评估后我们得到下面的结果:

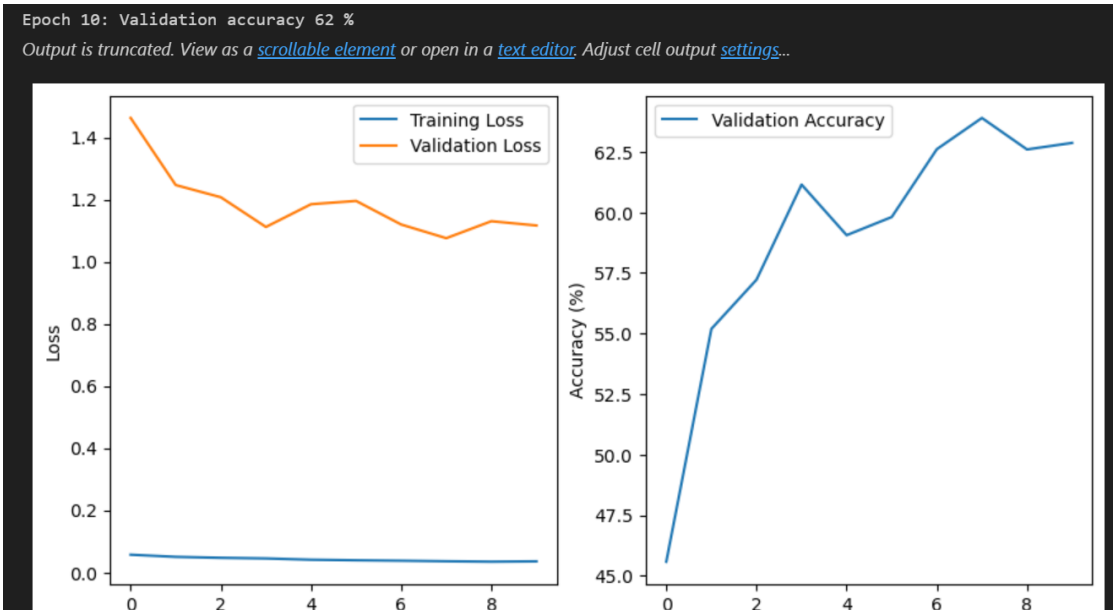


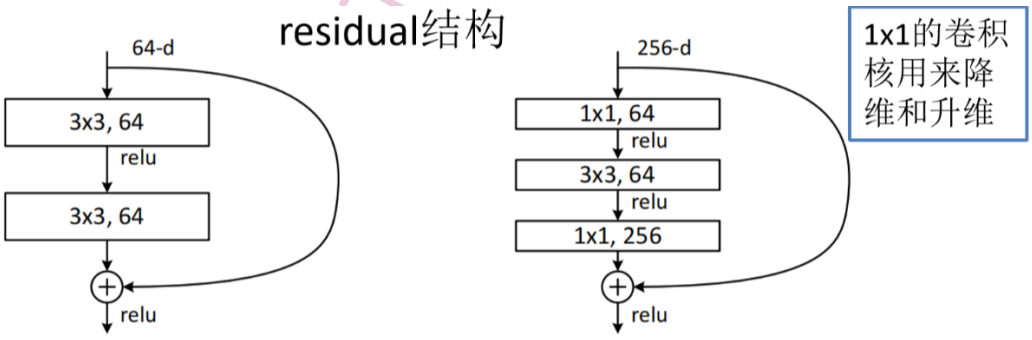
图 3: 模型评估结果

我们可以看到原始网络结构的准确率相对较低, 只有 0.62. 下面我们讲尝试着改进网络结构提升模型的准确率

(四) ResNet 的实现

1. ResNet18 模型

在原始的 ResNet 论文中向我们提供了两类模型, 一种是适用于低层数的网络, 一种是适用高层数的网络. 它们的思想大致相同, 都是在原始的网络结构中增加残差结构, 但是具体的不同:



$$3\times3\times256\times256+3\times3\times256\times256=1{,}179{,}648$$
$$1\times1\times256\times64+3\times3\times64\times64+1\times1\times64\times256=69{,}632$$

图 4: ResNet 的两种残差结构

在本次实验中我们采用了类似 ResNet18 的模型 (具体的卷积参数与原始不同), 显然我们应该选用左侧的残差结构, 具体实现代码如下:

```
# 定义 ResNet 基本块
class BasicBlock(nn.Module):
```

```
    expansion = 1

    def __init__(self, in_planes, planes, stride=1):
        super(BasicBlock, self).__init__()
        self.conv1 = nn.Conv2d(in_planes, planes, kernel_size=3, stride=
            stride, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(planes)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = nn.Conv2d(planes, planes, kernel_size=3, stride=1,
            padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(planes)
        self.shortcut = nn.Sequential()
        if stride != 1 or in_planes != self.expansion * planes:
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_planes, self.expansion * planes, kernel_size=1,
                    stride=stride, bias=False),
                nn.BatchNorm2d(self.expansion * planes)
            )
    def forward(self, x):
        out = self.relu(self.bn1(self.conv1(x)))
        out = self.bn2(self.conv2(out))
        out += self.shortcut(x)
        out = self.relu(out)
        return out
```

上面我们定义了 ResNet 网络结构的基本块, 最引人瞩目的是引入了残差连接结构 shortcut, 其将前一层的输入直接加到后一层的输出上, 这样可以保留前一层的信息并传递到后一层, 从而更好地训练深层网络。 实现完这些后我们就可以实现 Resnet 网络的搭建, 具体实现代码如下:

```
class ResNet(nn.Module):
    def __init__(self, block, num_blocks, num_classes=10):
        super(ResNet, self).__init__()
        self.in_planes = 64

        self.conv1 = nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1,
            bias=False)
        self.bn1 = nn.BatchNorm2d(64)
        self.relu = nn.ReLU(inplace=True)
        self.layer1 = self._make_layer(block, 64, num_blocks[0], stride=1)
        self.layer2 = self._make_layer(block, 128, num_blocks[1], stride=2)
        self.layer3 = self._make_layer(block, 256, num_blocks[2], stride=2)
        self.layer4 = self._make_layer(block, 512, num_blocks[3], stride=2)
        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
        self.fc = nn.Linear(512 * block.expansion, num_classes)

    def _make_layer(self, block, planes, num_blocks, stride):
        strides = [stride] + [1] * (num_blocks - 1)
        layers = []
```

```
        for stride in strides:
            layers.append(block(self.in_planes, planes, stride))
            self.in_planes = planes * block.expansion
        return nn.Sequential(*layers)

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

def ResNet18():
    return ResNet(BasicBlock, [2, 2, 2, 2])
```

实现这些后我们就可以对实验结果进行评估, 首先我们需要打印其网络结构

```
ResNet(
  (conv1): Conv2d(3, 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)
  (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)
      (shortcut): Sequential()
    )
    (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)
      (shortcut): Sequential()
    )
  )
  (layer2): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
...
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Linear(in_features=512, out_features=10, bias=True)
```

图 5: ResNet 的网络结构

然后我们观察其训练评估结果

```
Epoch [1/10], Loss: 1.4524, Train Accuracy: 46.91%, Test Accuracy: 55.40%
Epoch [2/10], Loss: 0.9437, Train Accuracy: 66.34%, Test Accuracy: 67.28%
Epoch [3/10], Loss: 0.7196, Train Accuracy: 74.77%, Test Accuracy: 70.40%
Epoch [4/10], Loss: 0.6025, Train Accuracy: 79.09%, Test Accuracy: 78.64%
Epoch [5/10], Loss: 0.5200, Train Accuracy: 82.14%, Test Accuracy: 80.36%
Epoch [6/10], Loss: 0.4644, Train Accuracy: 84.02%, Test Accuracy: 80.31%
Epoch [7/10], Loss: 0.4243, Train Accuracy: 85.29%, Test Accuracy: 79.74%
Epoch [8/10], Loss: 0.3892, Train Accuracy: 86.54%, Test Accuracy: 84.22%
Epoch [9/10], Loss: 0.3553, Train Accuracy: 87.68%, Test Accuracy: 82.68%
Epoch [10/10], Loss: 0.3262, Train Accuracy: 88.65%, Test Accuracy: 84.87%
```

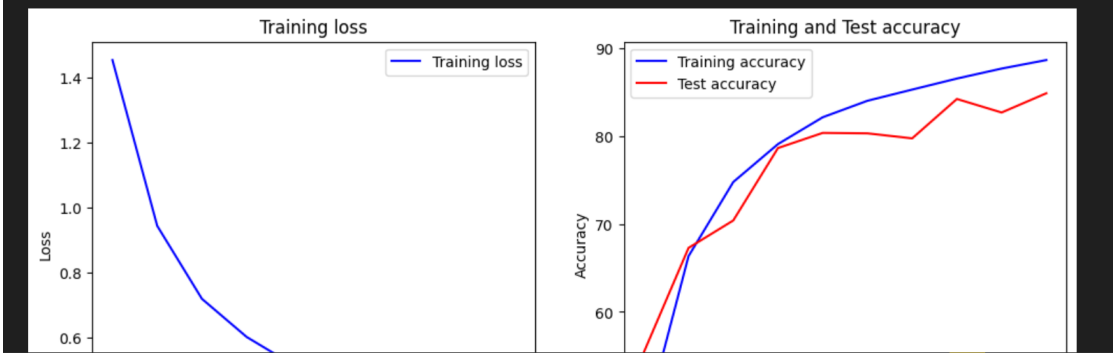


图 6: ResNet 的评估结果

2. 带 SE 结构的 ResNet

接下来我们尝试在 ResNet 的结构上加上 SEblock, 具体实现代码如下:

```python
    # 定义 SE 模块
class SEModule(nn.Module):
    def __init__(self, in_channels, reduction=16):
        super(SEModule, self).__init__()
        self.avg_pool = nn.AdaptiveAvgPool2d(1)
        self.fc = nn.Sequential(
            nn.Linear(in_channels, in_channels // reduction, bias=False),
            nn.ReLU(inplace=True),
            nn.Linear(in_channels // reduction, in_channels, bias=False),
            nn.Sigmoid()
        )

    def forward(self, x):
        b, c, _, _ = x.size()
        y = self.avg_pool(x).view(b, c)
        y = self.fc(y).view(b, c, 1, 1)
        return x * y.expand_as(x)

# 定义 ResNet 基本块 (带 SE 结构)
class SEBasicBlock(nn.Module):
    expansion = 1

    def __init__(self, in_planes, planes, stride=1, reduction=16):
        super(SEBasicBlock, self).__init__()
        self.conv1 = nn.Conv2d(in_planes, planes, kernel_size=3, stride=
            stride, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(planes)
```

```
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = nn.Conv2d(planes, planes, kernel_size=3, stride=1,
            padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(planes)
        self.se = SEModule(planes, reduction)

        self.shortcut = nn.Sequential()
        if stride != 1 or in_planes != self.expansion * planes:
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_planes, self.expansion * planes, kernel_size=1,
                    stride=stride, bias=False),
                nn.BatchNorm2d(self.expansion * planes)
            )

    def forward(self, x):
        out = self.relu(self.bn1(self.conv1(x)))
        out = self.bn2(self.conv2(out))
        out = self.se(out)
        out += self.shortcut(x)
        out = self.relu(out)
        return out
```

然后 ResNet 的其他结构与初始 ResNet 相同, 完成这些后我们就可以实现模型的评估, 评估结果如下:

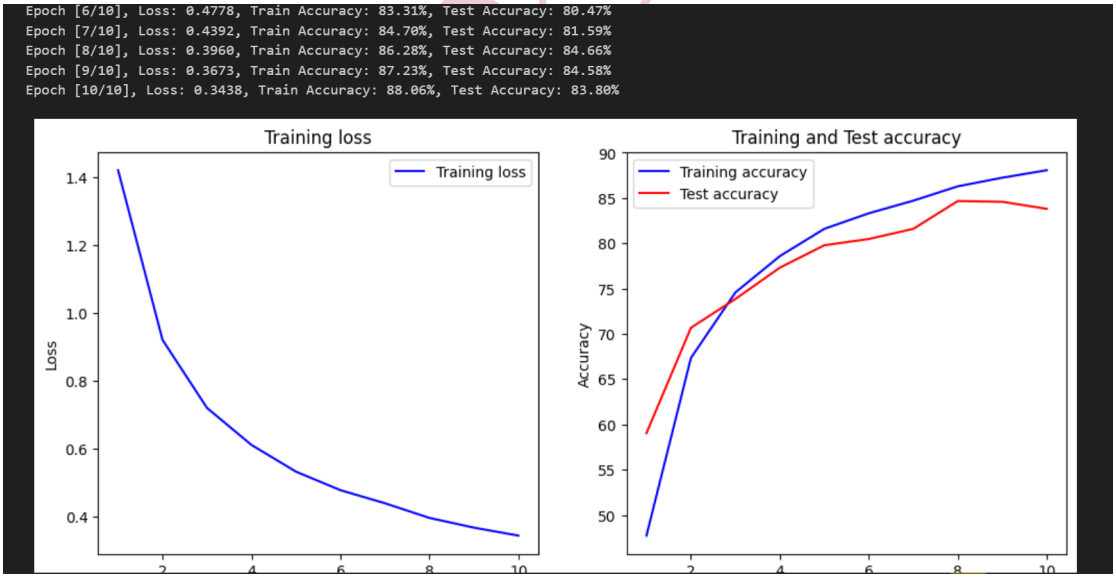


图 7: 带 SE 结构的 Resnet 评估结果

我们观察到相较于原始 ResNet, 改进之后的准确率并没有显著提升, 而且有些许下降. 起初我们认为是 reduction 参数的问题, 但是尝试多个参数后准确率还是没有提升. 推测可能是因为 CIFAR10 数据集较小,ResNet 已经可以很好地反映数据特征, 在这基础上增加参数反而会导致过拟合情况的出现. 要想增加训练的准确度我们可以从实时调整学习率参数入手.

（五） DenseNet 的实现

DenseNet（密集连接网络）是一种高效的卷积神经网络结构，旨在最大化特征的重用并解决深层网络中的梯度消失问题。DenseNet 通过在每一层之间建立密集连接，使得特征图能够更直接地传递和重用。其中 DenseNet 由多个 DenseBlock(密集块) 以及 TransitionLayer(过渡层) 组成, 每一层都接收前面所有层的输出作为其输入.

$$\mathbf{x}_l = H_l([\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_{l-1}])$$

其中 $\mathbf{x}_l$ 是第 l 层的输出，$[\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_{l-1}]$ 表示将所有之前层的输出拼接起来，H_l 是第 l 层的非线性转换（通常是卷积、批归一化和 ReLU 的组合）。 下面我们尝试着构建我们的 DenseNet:

```
class BottleneckLayer(nn.Module):
    def __init__(self, in_channels, growth_rate):
        super(BottleneckLayer, 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([x, out], 1)
        return out

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

    def forward(self, x):
        out = self.conv(F.relu(self.bn(x)))
        out = F.avg_pool2d(out, 2)
        return out

class DenseBlock(nn.Module):
    def __init__(self, num_layers, in_channels, growth_rate):
        super(DenseBlock, self).__init__()
        layers = []
        for i in range(num_layers):
            layers.append(BottleneckLayer(in_channels + i * growth_rate,
                growth_rate))
        self.block = nn.Sequential(*layers)

    def forward(self, x):
```

```
35          return self.block(x)
36
37  class DenseNet(nn.Module):
38      def __init__(self, num_classes=10, growth_rate=12, block_config=(16, 16,
            16)):
39          super(DenseNet, self).__init__()
40          self.conv1 = nn.Conv2d(3, 2 * growth_rate, kernel_size=3, padding=1,
                bias=False)
41          self.dense1 = DenseBlock(block_config[0], 2 * growth_rate,
                growth_rate)
42          self.trans1 = TransitionLayer(2 * growth_rate + block_config[0] *
                growth_rate, growth_rate)
43          self.dense2 = DenseBlock(block_config[1], growth_rate, growth_rate)
44          self.trans2 = TransitionLayer(growth_rate + block_config[1] *
                growth_rate, growth_rate)
45          self.dense3 = DenseBlock(block_config[2], growth_rate, growth_rate)
46          self.bn = nn.BatchNorm2d(growth_rate + block_config[2] * growth_rate)
47          self.linear = nn.Linear(growth_rate + block_config[2] * growth_rate,
                num_classes)
48
49      def forward(self, out):
50          out = self.conv1(out)
51          out = self.dense1(out)
52          out = self.trans1(out)
53          out = self.dense2(out)
54          out = self.trans2(out)
55          out = self.dense3(out)
56          out = torch.squeeze(F.avg_pool2d(F.relu(self.bn(out)), 8))
57          out = F.log_softmax(self.linear(out), dim=1)
58          return out
```

在实验中为了保证运行速度, 同时兼顾准确率我们选取了 growthrate=32, blockconfig=(8, 8, 8) 接下来我们观察其结果:

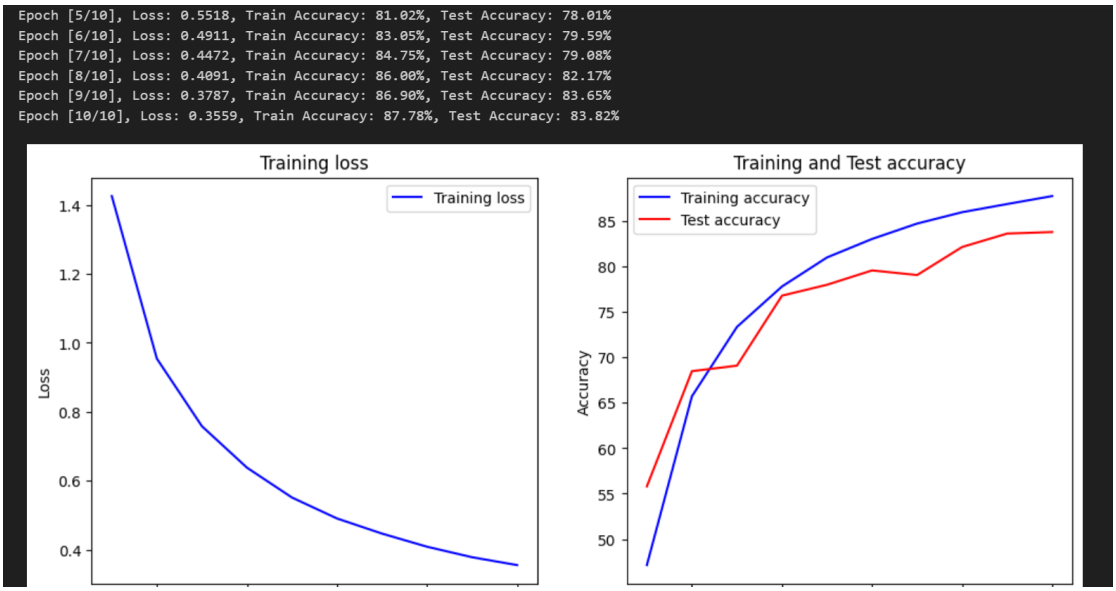


图 8: Densenet 的评估结果

我们观察到 Densenet 的准确率相对来说也是比较高的, 当然我们也可以尝试其他的参数来对该模型进行优化, 限于时间这里不再展开。

四、 总结

本次实验成功实现卷积神经网络模型, 并成功实现了 ResNet 以及带 SE 结构的 ResNet, 以及 DenseNet. 通过这次实验我们对卷积神经网络的结构有了更加深刻的理解.