人工智能实验——经典 CNN 架构 实验报告

姓名：	郑逸潇	学号：	10192100406

实验目标：

实现并理解几个 CNN 神经网络用于 MNIST 图像分类：LeNet、AlexNet、VGGNet、GoogLeNet、ResNet。

简介：

本次实验通过实现了五种经典的 CNN 架构，加强了对 CNN 的理解，其中在每个网络实现前都有一部分自己的理解，以及灰底部分的代码实现问题，每个模型后均有参数调整的过程及结果。在模型对比部分分别对不同模型的效果及其原因作了适当的分析。

实验环境

VSCode、Python、Pytorch

实验过程：

LeNet：

LeNet 的整体网络结构可以用一张图来概括，也就是下面这张 Yann LeCun 在 1998 年的论文中展示的图片：

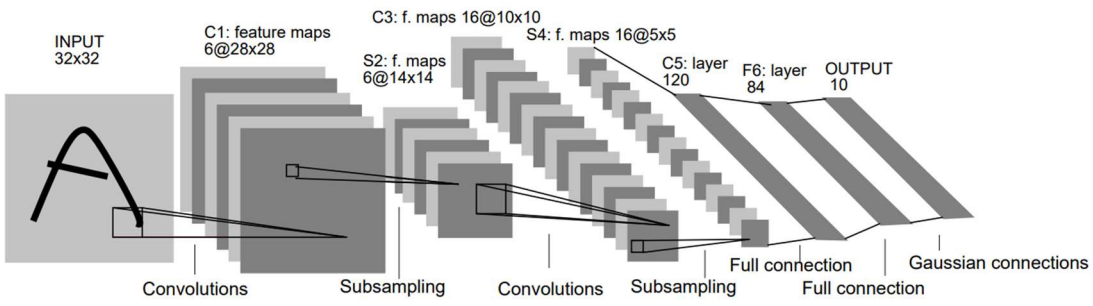

图 1 Gradient-based learning applied to document recognition 中 LeNet 结构图

论文中采用了两层卷积层，两层池化层，三层全连接层进行训练。**我将这个网络结构理解为：两个卷积池化层（卷积后进行池化）以及全连接层（本文中是三个全连接网络组成）**，其中卷积池化层的作用是提取特征图，全连接层则是模型主要的参数所在。

LeNet 是最早出现的经典卷积神经网络，卷积操作就是为提取特征而被适用于神经网络训练的。在绘画中，我们需要先打好画面框架，然后勾线，最后上色，而卷积的操作其实就是将这些步骤逆着推，我们查看第一层卷积出的结果时可以看出这很像是一种线稿（后面深度学习网络通常第一层是颜色、边缘特征，目的性不同）：

图 2 卷积操作可视化（参考北邮鲁鹏老师 PPT）

所以 LeNet 第一个卷积层就是为了得到一幅幅线稿（后面称之为特征响应图），然后就是池化层，LeNet 论文中的池化层采用了平均池化（Average Pooling）的方法，符合当时下采样兼顾各个像素点的启发式想法，但是在如今一般都采用了最大池化（Max Pooling），且获得的效果通常来说更好，我们也可以像理解绘画那样，思考一下绘画中构图搭建框架的时候，通常先是不管细节的，细节其实对整幅画作只起到了美观的作用，而不是识别出画作需要细节，就比如勾勒人物轮廓只需要一条细线即可，但是在打磨细节时需要稍稍将其加粗；MNIST 手写数据集的粗细其实并不影响识别。因此我们隔几个像素取其中最大的（也是最有特征的）像素点作为特征放入下一层可能得到的效果更好，实验证明确实如此。

全连接网络作为更早提出的一种机器学习方法，是一种类似拟合函数的方法，由于反向传播的提出，这变成了一种准确率较高的黑盒模型，这里不多赘述细节。全连接网络最大的诟病是容易过拟合和参数太多，但是我们可以设想，只要我们给全连接神经网络的数据够多，其算力够强，完全是可以模拟出任何函数的，而数据够多也就变相解决了过拟合问题。因此全连接神经网络可以说是神经网络模型的基础，但问题就是算力不够，数据不够，因此需要卷积层来提取特征，减小

训练数据。

在 LeNet 中，第一个卷积池化层就像是提取线稿，第二个卷积池化层就像是提取框架，然后放入全连接网络中进行训练，“拟合”出一个函数。

第一个卷积池化层：

在论文中，第一个卷积层卷积核大小为 5*5*1，个数为 6，步长为 1，补充两个 zero padding，第一个池化层大小为 2*2，步长为 2。

```
        self.conv_pool_1 = nn.Sequential(
            # 卷积层 (1*28*28) -> 6*28*28)
            nn.Conv2d(in_channels=1, out_channels=6, kernel_size=5,
stride=1, padding=2),
            nn.ReLU(),

            # 池化层 (6*28*28) -> (6*14*14)
            nn.MaxPool2d(kernel_size=2, stride=2)
        )
```

第二个卷积池化层：

在论文中，第二个卷积层大小为 5*5*6，个数为 16，步长为 1，无 padding，池化层大小为 2*2，步长为 2。

```
        self.conv_pool_2 = nn.Sequential(
            # 卷积层 (6*14*14) -> (16*10*10)
            nn.Conv2d(in_channels=6, out_channels=16, kernel_size=5,
stride=1, padding=0),
            nn.ReLU(),
            # 池化层 (16*10*10) -> (16*5*5)
            nn.MaxPool2d(2, 2)
        )
```

全连接层：

在论文中，第一个全连接层有 120 个输出，第二个全连接层有 84 个输出，第三个全连接层即类别个数。

```
        self.fc = nn.Sequential(
            # 将卷积池化后的 tensor 拉成向量
            nn.Flatten(),
            # 全连接层 16*5*5 -> 120
            nn.Linear(16 * 5 * 5, 120),
            nn.ReLU(),
            # 全连接层 120 -> 84
            nn.Linear(120, 84),
            nn.ReLU(),
```

```
            # 全连接层 84 -> 10
            nn.Linear(84, 10)
        )
```

在本次实验中， LeNet 的复现并未使用 Average Pooling，采用了更合适的 Max Pooling（理由在上面提到），激活函数则使用了 ReLU，因为如果使用 Sigmoid 激活，网络很容易出现梯度消失（再加上 Average Pooling 效果更差，故不做展示），可以看到 learning rate 大于 0.01 或者小于 1e-5 的时候就无法训练了：

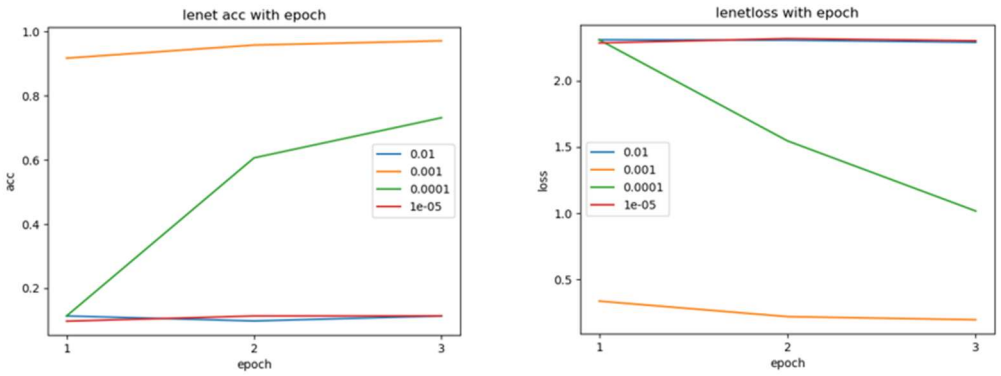

图 3 LeNet 使用 Sigmoid 激活时 acc/loss 随 epoch 的变化（不同学习率）

在起初的实验过程中，由于在 nn.Sequential 中未放入 ReLU 函数（因为以为 Pytorch 会自动补上，后面查证资料与实验验证是会在每个自定义的层后面补上，即上述例子的 conv_pool_1 后会补加 ReLU，而 Sequential 内部并不会补加），导致训练结果较差，之后在每一层全连接层后添加 ReLU 解决此问题。

LeNet 实验结果（由于实验模型中无 dropout，因此并不将其作为超参）：

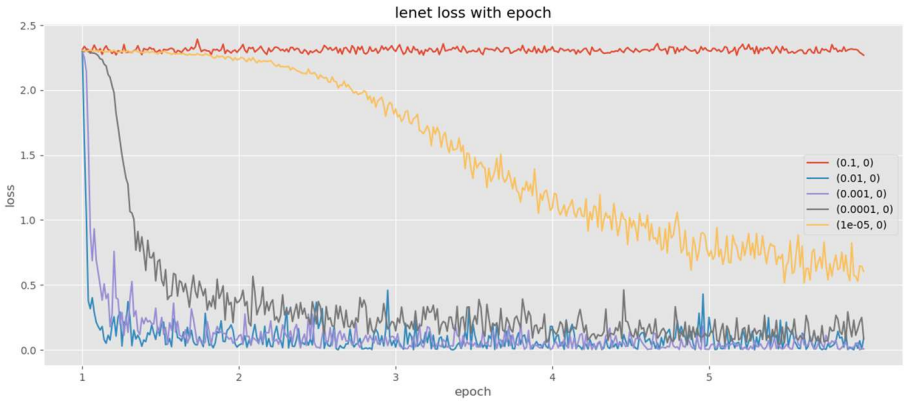

图 4 LeNet 使用 ReLU 激活时 loss 随 epoch 的变化（不同学习率）

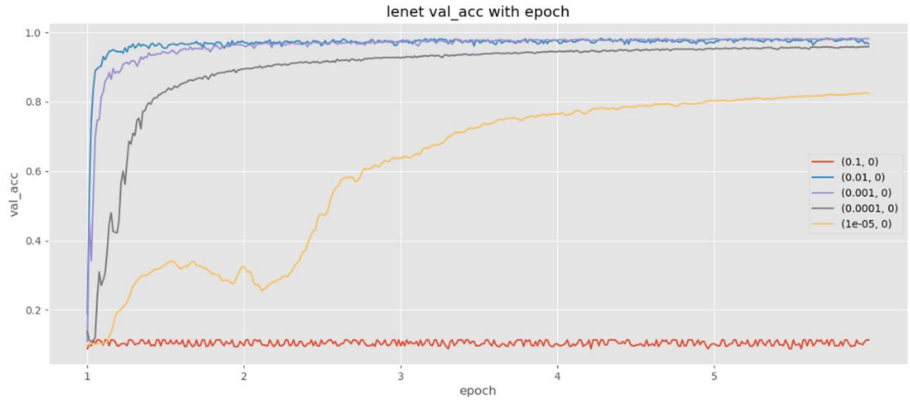

图 5 LeNet 使用 ReLU 激活时 acc 随 epoch 的变化（不同学习率）

在实验中，当学习率大于 0.1 时 acc 和 loss 会几乎保持不变，虽然使用了 Adam 优化器，但是学习率过大依旧会造成参数根本无法学习，由于学习率太大会导致梯度更新的时候过大，进而导致梯度爆炸；学习率小于 1e-5 时学习较慢。

因此这个模型我们采用 0.01 或者 0.001 的学习率训练就可以了，但是 0.001 从 loss 和 acc 的趋势来说更稳定，因此在最后模型比较时选择 0.001 作为超参数。

最后这五组参数在测试集上的结果：

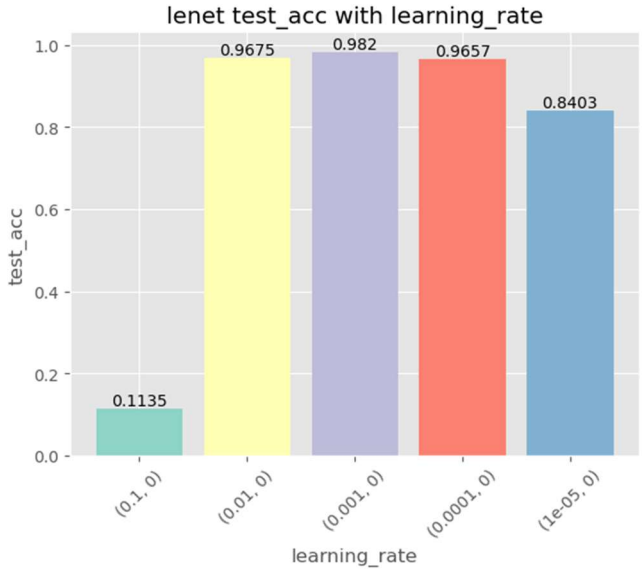

图 6 LeNet 在测试集的表现

AlexNet：

AlexNet 首次使用了 ReLU 激活函数和 dropout，并采用了 GPU 训练，但是本次实验中将下图架构中的两个并行训练网络合并成一个，并调整了原论文的参数，因为原始参数不适合这 MNIST 数据集。

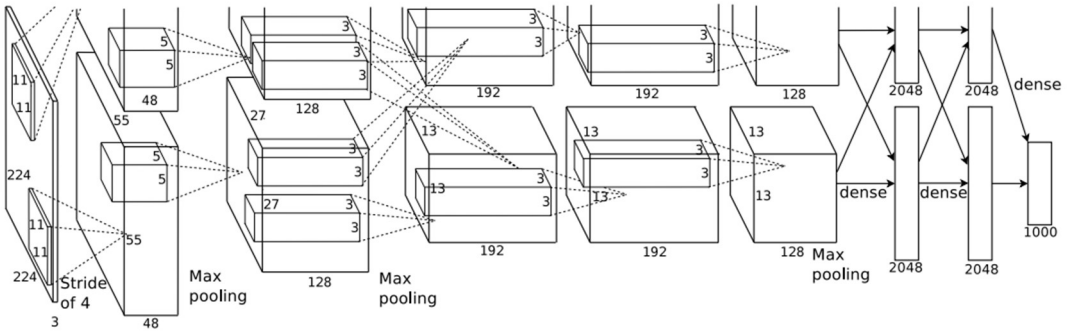

图 7 Imagenet classification with deep convolutional neural networks **中的** AlexNet **结构图**

AlexNet 是首次在 ImageNet 中使用的 CNN 模型，也是深度学习的奠基网络，正是因为 AlexNet 的出现，使得深度学习迎来了新的春天。

个人认为 LeNet 是简单地从绘画的视角出发，第一层提取线稿，第二层提取结构，然后用于分类等任务，但是有些图片并不是那么容易找到线稿的，在 MNIST 数据集中，LeNet 的网络完全可以胜任；而自然界真实的图片中，各种颜色、纹理、形状可能并不能一次性提取出来。因此 AlexNet 加深了网络结构，使得网络可以更好的提取特征，将图片不同的特征反应在“特征响应图”中，不同的特征响应图用来响应不同的特征（LeNet 更多的意思是结合不同的“线稿”生成绘图，是都考虑的），卷积层的深度就会越来越深，提取的特征也就越来越明显：

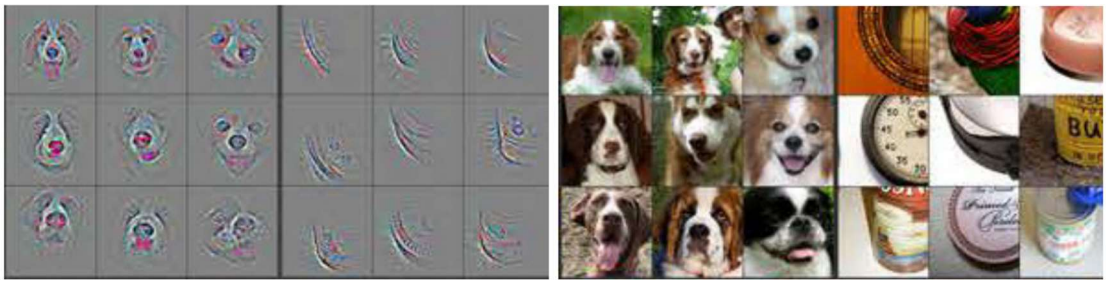

图 8 Visualizing and Understanding Convolutional Networks **第四层特征响应图可视化**

同 LeNet 一样，我将 AlexNet 理解成三个卷积池化层和一个全连接层，我认为前两个卷积池化层在原论文中是为了提取颜色、纹理、边缘等大致特征，然后第三个卷积池化层则是提取更细致准确的特征，然后将这些特征响应图放入全连接网络进行训练。

第一层卷积池化层：

原论文中使用了 11*11*3 的卷积核，个数为 96，但是原论文中的卷积核并不适合 MNIST 数据集，因此我将第一层的卷积核设为 3*3*1，个数为 24，步长为

1，padding 为 1，池化层使用了 2*2 的核，步长为 2，AlexNet 还提出了 LRN，用于模型泛化，LRN 的 size 为 3，其余按照 Pytorch 默认参数。

```
        self.conv_pool_1 = nn.Sequential(
            # 卷积层 (1*28*28) -> (24*28*28)
            nn.Conv2d(in_channels=1, out_channels=24, kernel_size=3,
stride=1, padding=1),
            nn.ReLU(),
            # 池化层 (24*28*28) -> (24*14*14)
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.LocalResponseNorm(size=3)
        )
```

第二层卷积池化层：

参数与第一层基本相同，只是通道数增加了，为了分离各个特征响应。

```
        self.conv_pool_2 = nn.Sequential(
            # 卷积层 (24*14*14) -> (64*14*14)
            nn.Conv2d(in_channels=24, out_channels=64, kernel_size=3,
stride=1, padding=1),
            nn.ReLU(),
            # 池化层 (64*14*14) -> (64*7*7)
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.LocalResponseNorm(size=3)
        )
```

第三层卷积池化层：

此层是为了找到更细致的特征，由三个卷积层构成，卷积核的大小均为 3*3，但是通道数稍作改变，步长和 padding 均为 1，最后一层池化大小为 2*2，步长为 2，参数相较原论文也有所缩小。

```
        self.conv_pool_3 = nn.Sequential(
            # 卷积层 (64*7*7) -> (96*7*7)
            nn.Conv2d(in_channels=64, out_channels=96, kernel_size=3,
stride=1, padding=1),
            nn.ReLU(),
            # 卷积层 (96*7*7) -> (96*7*7)
            nn.Conv2d(in_channels=96, out_channels=96, kernel_size=3,
stride=1, padding=1),
            nn.ReLU(),
            # 卷积层 (96*7*7) -> (64*7*7)
            nn.Conv2d(in_channels=96, out_channels=64, kernel_size=3,
stride=1, padding=1),
            nn.ReLU(),
            # 池化层 (64*7*7) -> (64*3*3)
```

```
            nn.MaxPool2d(kernel_size=2, stride=2)
        )
```

全连接层：

在本层中，AlexNet 还添加了 dropout，然后以三层全连接层模拟出最后的分类函数，参数相比原论文也是有所缩小。

```
        self.fc = nn.Sequential(
            # 将卷积池化后的 tensor 拉成向量
            nn.Flatten(),
            # dropout
            nn.Dropout(dropout),
            # 全连接层 (64*3*3) -> (512)
            nn.Linear(64 * 3 * 3, 512),
            nn.ReLU(),
            # dropout
            nn.Dropout(dropout),
            # 全连接层 (512) -> (512)
            nn.Linear(512, 512),
            nn.ReLU(),
            # dropout
            nn.Dropout(dropout),
            # 全连接层 (512) -> (10)
            nn.Linear(512, label_num)
        )
```

AlexNet 实验结果：

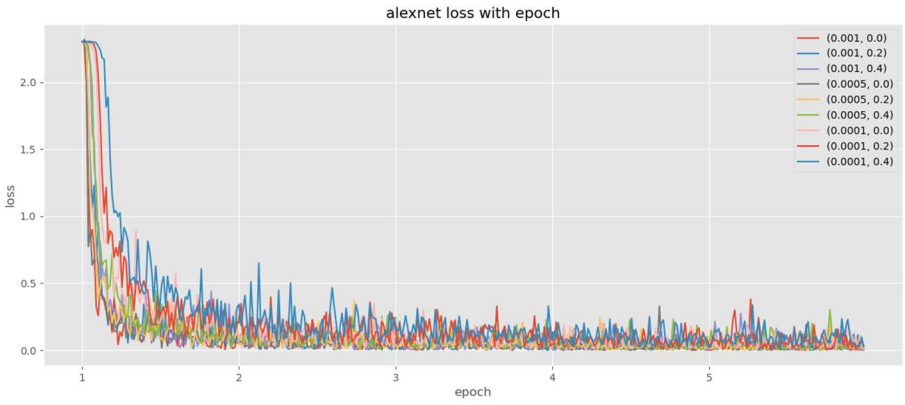

图 9 AlexNet loss 随 epoch 的变化（不同学习率和 dropout）

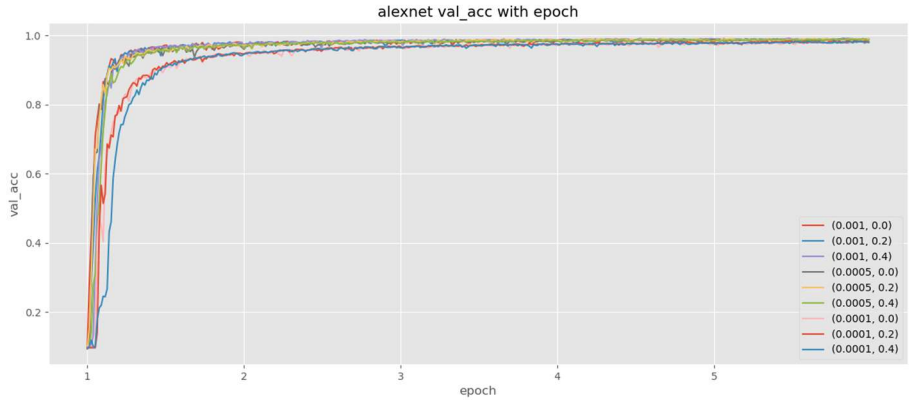

图 10 AlexNet acc 随 epoch 的变化（不同学习率和 dropout）

我们可以看到当学习率为 0.001 或者 0.0005 的时候 loss 下降和 acc 上升都比较快，同时，dropout 对于模型性能的影响也不大，应该是因为 MNIST 数据集太过于简单。

为了追求更快更稳定的结果，我就选择了学习率 0.001 和 dropout 0.2 作为模型比较时的参数。

最后这九组参数在测试集上的结果：

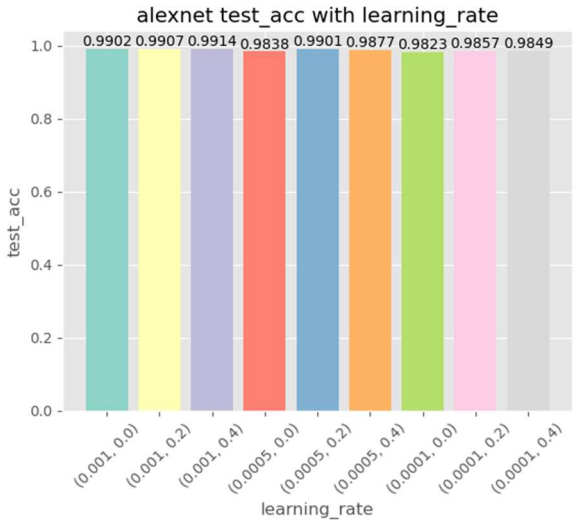

图 11 AlexNet 在测试集的表现

VGGNet：

无论是从网络结构还是实验效果来说，VGGNet 都是一种“升级版”的 AlexNet，AlexNet 由于网络深度和卷积核结构，还不能很好的提取出图像的特征，AlexNet 的感受野可以说是一蹴而就的，前两层就将特征响应图缩小到一定程度，而 VGGNet 是通过一次次卷积，慢慢将特征响应图缩小，能够更好地提取特征。但

随之而来的结果就是梯度问题，容易出现梯度爆炸或者消失，因此当时 VGGNet 无法将深度提升到更深的层次。

由于 VGGNet 和 AlexNet 基本思想相同，这里也不多赘述了，同 AlexNet 一样，我将 VGGNet 分为 5 个卷积池化层（VGG16，VGG19 为 6 个卷积池化层）和 1 个全连接层，由于和 AlexNet 思想相同，故不再细致说明各层参数。

第一个卷积池化层：

```
        self.conv_pool_1 = nn.Sequential(
            nn.Conv2d(in_channels=1, out_channels=16, kernel_size=3,
stride=1, padding=1),
            nn.ReLU(),
            nn.Conv2d(in_channels=16, out_channels=16, kernel_size=3,
stride=1, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=1, padding=1)
        )
```

第二个卷积池化层：

```
        self.conv_pool_2 = nn.Sequential(
            nn.Conv2d(in_channels=16, out_channels=32, kernel_size=3,
stride=1, padding=1),
            nn.ReLU(),
            nn.Conv2d(in_channels=32, out_channels=32, kernel_size=3,
stride=1, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=1, padding=1)
        )
```

第三个卷积池化层：

```
        self.conv_pool_3 = nn.Sequential(
            nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3,
stride=1, padding=1),
            nn.ReLU(),
            nn.Conv2d(in_channels=64, out_channels=64, kernel_size=3,
stride=1, padding=1),
            nn.ReLU(),
            nn.Conv2d(in_channels=64, out_channels=64, kernel_size=3,
stride=1, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2)
        )
```

第四个卷积池化层：

```
        self.conv_pool_4 = nn.Sequential(
```

```
            nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3,
stride=1, padding=1),
            nn.ReLU(),
            nn.Conv2d(in_channels=128, out_channels=128, kernel_size=3,
stride=1, padding=1),
            nn.ReLU(),
            nn.Conv2d(in_channels=128, out_channels=128, kernel_size=3,
stride=1, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2)
        )
```

第五个卷积池化层：

```
        self.conv_pool_5 = nn.Sequential(
            nn.Conv2d(in_channels=128, out_channels=128, kernel_size=3,
stride=1, padding=1),
            nn.ReLU(),
            nn.Conv2d(in_channels=128, out_channels=128, kernel_size=3,
stride=1, padding=1),
            nn.ReLU(),
            nn.Conv2d(in_channels=128, out_channels=128, kernel_size=3,
stride=1, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2)
        )
```

VGG19 的第六个卷积池化层：

```
        if self.layer == 19:
            self.conv_pool_6 = nn.Sequential(
                nn.Conv2d(in_channels=128, out_channels=128,
kernel_size=3, stride=1, padding=1),
                nn.ReLU(),
                nn.Conv2d(in_channels=128, out_channels=128,
kernel_size=3, stride=1, padding=1),
                nn.ReLU(),
                nn.Conv2d(in_channels=128, out_channels=128,
kernel_size=3, stride=1, padding=1),
                nn.ReLU(),
                nn.MaxPool2d(kernel_size=3, stride=1,
padding=1)
            )
```

全连接层：

```
        self.fc = nn.Sequential(
            nn.Flatten(),
            nn.Dropout(dropout),
```

```
            nn.Linear(128 * 3 * 3, 256),
            nn.ReLU(),
            nn.Dropout(dropout),
            nn.Linear(256, 256),
            nn.ReLU(),
            nn.Dropout(dropout),
            nn.Linear(256, 256),
            nn.ReLU(),
            nn.Dropout(dropout),
            nn.Linear(256, label_num)
        )
```

VGGNet 实验结果：

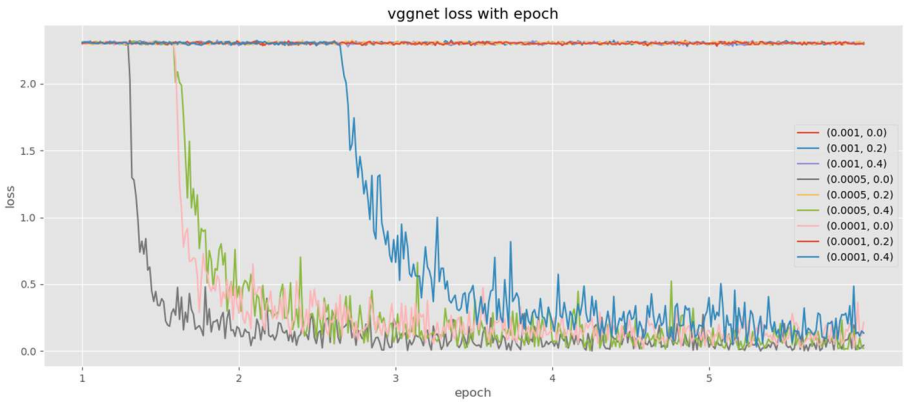

图 12 VGGNet loss 随 epoch 的变化（不同学习率和 dropout）

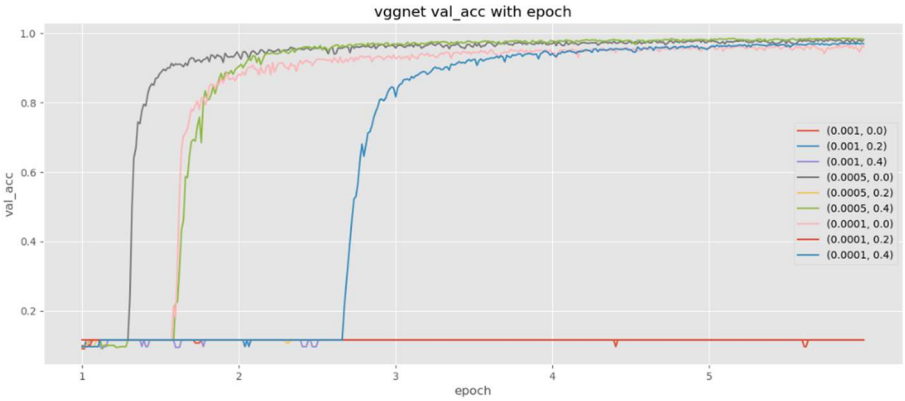

图 13 VGGNet acc 随 epoch 的变化（不同学习率和 dropout）

从图中我们可以看出，VGG 是较难以训练的一种网络，时不时就无法训练，这就是网络深度达到了一定程度，就难以训练了，同时期的 GoogLeNet 则设计了 inception 解决了这个问题。另外，当我多次运行训练 VGG 的代码时，有时候无法训练的学习率又可以训练了，可能与参数初始化也有一定关系，所以不好选

择哪个是最好的。

因此我选择了较快收敛的 0.0005 学习率和 0.4 dropout 作为模型比较的参数。

最后这九组参数在测试集上的结果：

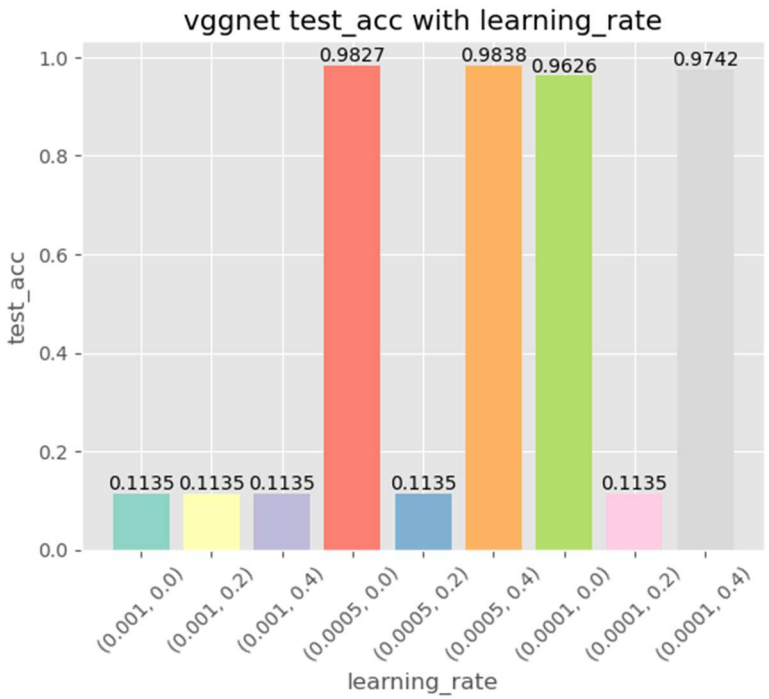

图 14 VGGNet 在测试集上的表现

GoogLeNet：

GoogLeNet 最大的特点就是 inception 的设计，主要是为了解决网络训练问题，随着网络深度越来越深，参数也越来越多，这使得网络训练越来越慢，同时也会带来其他副作用：梯度消失/爆炸，过拟合等。而 inception 的设计就是为了缓解这些情况。

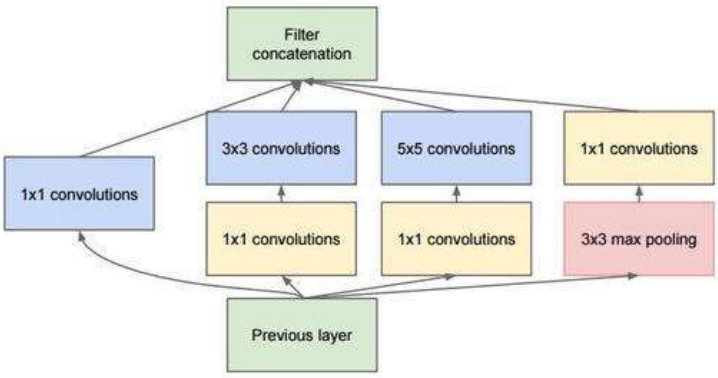

图 15 inception 模块结构

其中 1*1 的卷积核目的是为了压缩参数，使得网络更好训练。网络参数越多，网络越难训练，也越容易过拟合，但是很多时候有些参数可能是没有用的或者重复的，个人认为 1*1 的卷积核就是用来过滤这些参数的，因为网络初始化是随机的，所以可能有些特征响应图可能是重复的或者什么也不响应的。

而把不同的卷积核同时对一个特征响应图进行卷积，然后再组合，就意味着最后组合的特征响应图能包含细节方面的特征，也能包含大局方面的特征，分类更加准确。

GoogLeNet 还有一个特点是辅助分类器，为了防止梯度无法回传导致浅层神经网络无法训练，就在适当的位置再添加一个或几个分类器，但是 Pytorch 的框架是自动回传梯度的方法，无法从模型中间改变前面的参数，查阅相关资料发现大多数人都是以一个权重来将所有分类器的 loss 叠加再 backward，因此我只是写了辅助分类器的类，在训练时并没有调用。

论文中还说明用平均池化层代替全连接层进一步减少参数，但是实际上平均池化成 1*1 后还是放入全连接层进行拟合。

Inception：

```
class Inception(nn.Module):
    def __init__(self, in_channels, out_channels_1, out_channels_2_1,
out_channels_2_2, out_channels_3_1, out_channels_3_2, out_channels_4):
        super(Inception, self).__init__()

        self.branch1 = nn.Sequential(
            nn.Conv2d(in_channels=in_channels,
out_channels=out_channels_1, kernel_size=1),
            nn.BatchNorm2d(out_channels_1),
            nn.ReLU()
        )

        self.branch2 = nn.Sequential(
            nn.Conv2d(in_channels=in_channels,
out_channels=out_channels_2_1, kernel_size=1),
            nn.BatchNorm2d(out_channels_2_1),
            nn.ReLU(),
            nn.Conv2d(in_channels=out_channels_2_1,
out_channels=out_channels_2_2,kernel_size=3, padding=1),
            nn.BatchNorm2d(out_channels_2_2),
            nn.ReLU()
        )
```

```
        self.branch3 = nn.Sequential(
            nn.Conv2d(in_channels=in_channels,
out_channels=out_channels_3_1, kernel_size=1),
            nn.BatchNorm2d(out_channels_3_1),
            nn.ReLU(),
            nn.Conv2d(in_channels=out_channels_3_1,
out_channels=out_channels_3_2, kernel_size=5, padding=2),
            nn.BatchNorm2d(out_channels_3_2),
            nn.ReLU()
        )

        self.branch4 = nn.Sequential(
            nn.MaxPool2d(kernel_size=3, stride=1, padding=1),
            nn.Conv2d(in_channels=in_channels,
out_channels=out_channels_4, kernel_size=1),
            nn.BatchNorm2d(out_channels_4),
            nn.ReLU()
        )

    def forward(self, x):
        x_1 = self.branch1(x)
        x_2 = self.branch2(x)
        x_3 = self.branch3(x)
        x_4 = self.branch4(x)
        x = torch.cat([x_1, x_2, x_3, x_4], 1)
        return x
```

AuxClassifier：

```
class AuxClassifier(nn.Module):
    def __init__(self, in_channels, label_num=10, dropout=0.5):
        super(AuxClassifier, self).__init__()

        self.average_pool = nn.AvgPool2d(kernel_size=5, stride=3)

        self.conv = nn.Sequential(
            nn.Conv2d(in_channels=in_channels, out_channels=128,
kernel_size=1),
            nn.BatchNorm2d(128),
            nn.ReLU()
        )

        self.fc = nn.Sequential(
```

```
            nn.Flatten(),
            nn.Dropout(dropout),
            nn.Linear(128 * 4 * 4, 1024),
            nn.ReLU(),
            nn.Dropout(dropout),
            nn.Linear(1024, label_num),
            nn.ReLU()
        )

        self.softmax = nn.Softmax(dim=1)

    def forward(self, x):
        x = self.average_pool(x)
        x = self.conv(x)
        x = self.fc(x)
        x = self.softmax(x)
        return x
```

GoogLeNet：

```
class GoogLeNet(nn.Module):
    def __init__(self, label_num=10, dropout=0.5, aux=False):
        super(GoogLeNet, self).__init__()

        self.aux = aux

        self.conv_pool = nn.Sequential(
            # (1*28*28) -> (8*28*28)
            nn.Conv2d(in_channels=1, out_channels=8, kernel_size=7,
stride=1, padding=3),
            nn.BatchNorm2d(8),
            nn.ReLU(),
            # (8*28*28) -> (8*14*14)
            nn.MaxPool2d(kernel_size=3, stride=2, padding=1),

            # (8*14*14) -> (8*14*14)
            nn.Conv2d(in_channels=8, out_channels=8, kernel_size=1),
            nn.BatchNorm2d(8),
            nn.ReLU(),

            # (8*14*14) -> (24*14*14)
            nn.Conv2d(in_channels=8, out_channels=24, kernel_size=3,
stride=1, padding=1),
            nn.BatchNorm2d(24),
```

```
            nn.ReLU(),
            # (24*14*14) -> (24*7*7)
            nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        )

        self.inceptions_1 = nn.Sequential(
            Inception(24, 8, 12, 16, 2, 4, 4),          # inception3a
            Inception(32, 16, 16, 24, 4, 12, 8),        # inception3b
            # (24*7*7) -> (24*3*3)
            nn.MaxPool2d(kernel_size=3, stride=2),      # MaxPool
3*3+2(S)
            Inception(60, 24, 12, 26, 2, 6, 8)          # inception4a
        )

        self.aux1 = AuxClassifier(64, label_num, dropout)

        self.inceptions_2 = nn.Sequential(
            Inception(64, 20, 14, 28, 3, 8, 8),         # inception4b
            Inception(64, 16, 16, 32, 3, 8, 8),         # inception4c
            Inception(64, 14, 18, 36, 4, 8, 8),         # inception4d
            Inception(66, 32, 20, 40, 4, 16, 16),       # inception4e
            # (24*3*3) -> (24*1*1)
            nn.MaxPool2d(kernel_size=3, stride=2)       # MaxPool
3*3+2(S)
        )

        self.aux2 = AuxClassifier(66, label_num, dropout)

        self.inceptions_3 = nn.Sequential(
            Inception(104, 32, 20, 40, 4, 16, 16),      # inception5a
            Inception(104, 48, 24, 48, 6, 16, 16)       # inception5b
        )

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

        self.fc = nn.Sequential(
            nn.Flatten(),
            nn.Dropout(dropout),
            nn.Linear(128, label_num),
            nn.ReLU()
        )

    def forward(self, x):
        x = self.conv_pool(x)
```

```
        x = self.inceptions_1(x)
        if self.training and self.aux:
            x_aux_1 = self.aux1(x)

        x = self.inceptions_2(x)
        if self.training and self.aux:
            x_aux_2 = self.aux2(x)

        x = self.inceptions_3(x)
        x = self.avg_pool(x)
        x = self.fc(x)

        if self.aux:
            return x, x_aux_1, x_aux_2

        return x
```

原论文中并不使用 BatchNorm 而是 LRN，但是在原论文提出后不久就提出了 BatchNorm，因而很多时候 GoogLeNet 还是使用了 BatchNorm 进行归一化。

GoogLeNet 实验结果：

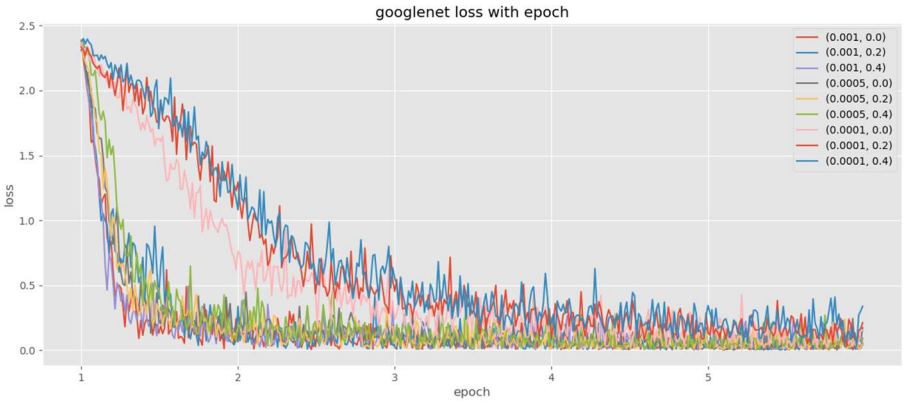

图 16 GoogLeNet loss **随** epoch **的变化（不同学习率和** dropout）

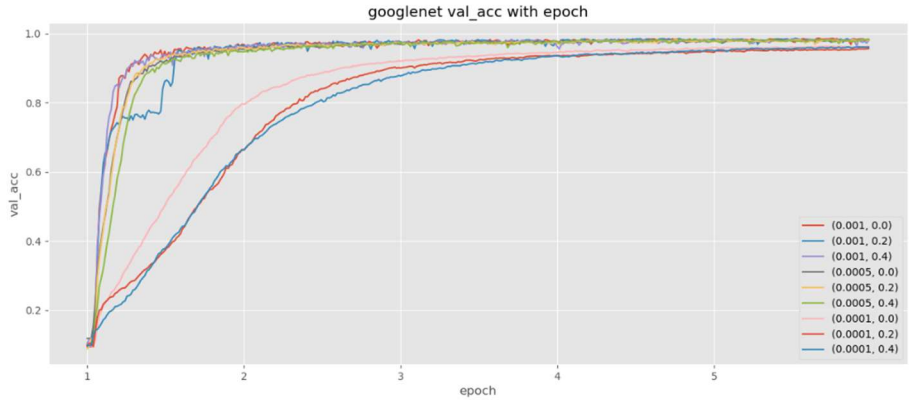

图 17 GoogLeNet acc 随 epoch 的变化（不同学习率和 dropout）

可以看出 0.0005 的学习率曲线较为稳定，0.4 的 dropout 较 0.2 慢了些，**所以模型比较时选择学习率 0.0005，dropout 0.2 为参数**。

最后这九组参数在测试集上的结果：

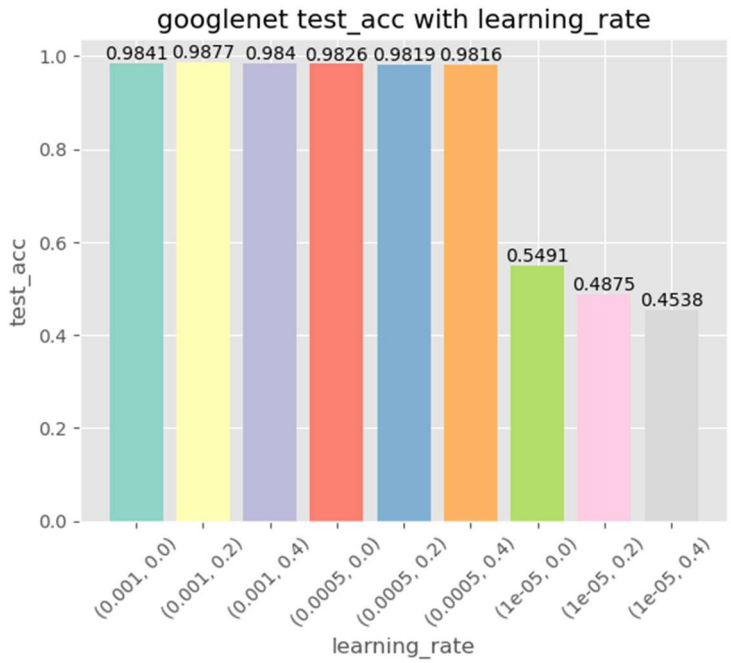

图 18 GoogLeNet 在测试集的表现

虽然模型选择并没有选择在测试集表现较好的 0.001 学习率，但是我们在测试前是无法得知测试集的，且 0.0005 学习率较为稳定，因此还是选择这一组。

ResNet：

从上面几个网络的实验结果可以看出：神经网络深度达到一定程度后，很容易出现梯度爆炸/梯度消失的现象，导致网络训练困难，而且随着网络深度不断

增加，“退化现象（Degradation）”越来越明显，这是限制神经网络深度继续加深的主要原因，而何恺明团队针对这个问题提出了“快捷连接（Shortcut Connection）”，并结合 Batch Normalization，发明了残差神经网络，很好的解决了梯度传播问题和“退化现象”，使得神经网络的深度首次突破了 100 层、最大的神经网络甚至超过了 1000 层。

我们可以设想，如果训练一个较深的神经网络和一个较浅的神经网络（假设浅层神经网络效果更好），而较深的神经网络训练参数肯定是要比浅层更多的，可是即使参数多了，要是训练得够多，理应出现和浅层神经网络差不多得结果，也就是有些层是无用的，那么为什么会出现“退化现象”呢？一个很自然的想法是训练方法有问题，我们知道梯度下降法确实会存在问题，鞍点和局部极值点会影响参数更改，虽然现有的优化器已经缓解了很大一部分问题，但是当参数越来越多的时候，是不是可能部分参数处在鞍点和局部极值点的可能性也越大了呢？但是我们又很难去找到另一个训练方法去替代梯度下降法，所以更实际的方法是直接将网络中某些层“舍弃”，也就是如果这一层效果不好的话，我们直接绕过这一层，与后面的层进行交互，我想这也就是 Shortcut Connection 的想法。

ResNet 最大的特点就是 Shortcut Connection：

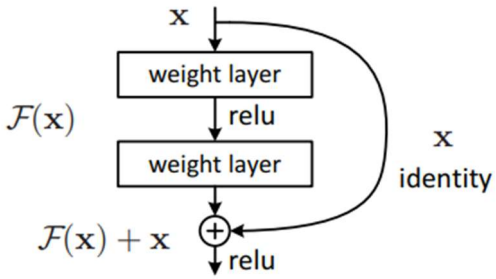

图 19　Residual learning: a building block

从上图我们可以看出，梯度有一部分是直接传递到上一层的（identity），而另一部分才是通过 F 函数传递的，所以梯度$\frac{\partial los}{\partial x}=\frac{\partial lo}{\partial(F(x)+x)}\cdot\frac{\partial(F(x)+x)}{\partial x}=\frac{\partial los}{\partial(F(x)+x)}\cdot(\frac{\partial F(x)}{\partial x}+1)$几乎不会为 0，而当 F 函数几乎无效的时候，大部分梯度都是直接回传到上一层，同时有 Batch Normalization 进一步防止梯度消失和梯度爆炸。

由于 ResNet 有两种 Block，且是反复使用，而我起初并不了解 Pytorch 如何使用循环创建 Module 中的层数，查阅相关资料后得知，可以将所有 nn.Module

放入列表 layers 中，然后使用 nn.Sequential(*layers)即可，Python 中的*符号可以将列表、元组等数据结构“解包”成内部的数据一一当作参数传入。

按原论文的分类，残差块分为 BasicBlock 和 BottleneckBlock(Bottleneck)，前者是简单的基本残差模块，和前几个网络的思想一样是缩小特征响应图提取特征，但是将 Pooling 层省去（神经网络有两大主张——其一主张只用 Pooling 层减小特征响应图，另一主张只用卷积层来减小特征响应图即可，没必要使用 Pooling 层，而 ResNet 属于后者，虽然它的第一层和最后一层使用了 Pooling，但是大部分还是未使用 Pooling 的），但是 Bottleneck 块则是以一种先缩小后放大的方法对特征进行压缩成较少的特征响应图，再将少部分特征响应图区分成更多的特征响应图（感觉有点像 Seq2Seq 的思想）。

BasicBlock：

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

    def __init__(self, in_channels, out_channels, stride=1):
        super(BasicBlock, self).__init__()

        self.conv1 = nn.Sequential(
            nn.Conv2d(in_channels=in_channels,
out_channels=out_channels, kernel_size=3, stride=stride, padding=1,
bias=False),
            nn.BatchNorm2d(out_channels),
            nn.ReLU()
        )

        self.conv2 = nn.Sequential(
            nn.Conv2d(in_channels=out_channels,
out_channels=out_channels * self.multiplier, kernel_size=3, stride=1,
padding=1, bias=False),
            nn.BatchNorm2d(out_channels * self.multiplier),
        )

        self.shortcut = nn.Sequential()
        if in_channels != out_channels * self.multiplier or stride != 1:
            self.shortcut = nn.Sequential(
```

```
                nn.Conv2d(in_channels=in_channels,
out_channels=out_channels * self.multiplier, kernel_size=1,
stride=stride, padding=0, bias=False),
                nn.BatchNorm2d(out_channels * self.multiplier)
            )

        self.relu = nn.ReLU()

    def forward(self, x):
        residual = self.conv2(self.conv1(x))
        shortcut = self.shortcut(x)
        return self.relu(residual + shortcut)
```

BasicBlock 的第一个卷积通常不改变通道数，仅缩小特征响应图，第二个卷积通常只增大通道数，不改变特征响应图大小，而 shortcut 仅将通道数改为和两次卷积出的特征响应图个数相同即可，然后将二者相加进行 ReLU 操作。

Bottleneck：

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

    def __init__(self, in_channels, out_channels, stride=1):
        super(Bottleneck, self).__init__()

        self.conv1 = nn.Sequential(
            nn.Conv2d(in_channels=in_channels,
out_channels=out_channels, kernel_size=1, stride=1, padding=0,
bias=False),
            nn.BatchNorm2d(out_channels),
            nn.ReLU()
        )

        self.conv2 = nn.Sequential(
            nn.Conv2d(in_channels=out_channels,
out_channels=out_channels, kernel_size=3, stride=stride, padding=1,
bias=False),
            nn.BatchNorm2d(out_channels),
            nn.ReLU()
        )

        self.conv3 = nn.Sequential(
```

```
            nn.Conv2d(in_channels=out_channels,
out_channels=out_channels * self.multiplier, kernel_size=1, stride=1,
padding=0, bias=False),
            nn.BatchNorm2d(out_channels * self.multiplier)
        )

        self.shortcut = nn.Sequential()
        if in_channels != out_channels * self.multiplier or stride != 1:
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_channels=in_channels,
out_channels=out_channels * self.multiplier, kernel_size=1,
stride=stride, padding=0, bias=False),
                nn.BatchNorm2d(out_channels * self.multiplier)
            )

        self.relu = nn.ReLU()

    def forward(self, x):
        resiudual = self.conv3(self.conv2(self.conv1(x)))
        shortcut = self.shortcut(x)
        return self.relu(resiudual + shortcut)
```

Bottleneck 相较于 BasicBlock 仅多加了第一层卷积层，通常用于降低通道数，第三层卷积层也略有改变，从卷积核 3*3 步长为 1，padding 为 1 改成卷积核 1*1 步长为 1，padding 为 0，作用也是增大通道数。

ResNet：

```
class ResNet(nn.Module):

    def __init__(self, layer_num=18, label_num=10):
        super(ResNet, self).__init__()
        self.base_channels = 64

        block_type, block_nums = self.res_net_params(layer_num)

        self.conv_pool_layer = nn.Sequential(
            nn.Conv2d(in_channels=1, out_channels=self.base_channels,
kernel_size=7, stride=2, padding=3, bias=False),
            nn.BatchNorm2d(self.base_channels),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=3, stride=1, padding=1)
        )
```

```
        self.res_layers = nn.Sequential(
            self.res_layer(block_type, 64, block_nums[0], stride=1),
            self.res_layer(block_type, 128, block_nums[1], stride=2),
            self.res_layer(block_type, 256, block_nums[2], stride=2),
            self.res_layer(block_type, 512, block_nums[3], stride=2)
        )

        # 平均池化，平均池化成 1*1
        self.avg_pool_layer = nn.AdaptiveAvgPool2d((1, 1))

        self.fc_layer = nn.Sequential(
            nn.Flatten(),
            nn.Linear(512 * block_type.multiplier, label_num)
        )

    def res_layer(self, block_type, out_channel, block_num, stride):
        blocks = []
        for _ in range(block_num):
            new_block = block_type(in_channels=self.base_channels,
out_channels=out_channel, stride=stride)
            blocks.append(new_block)
            self.base_channels = out_channel * new_block.multiplier
        return nn.Sequential(*blocks)

    def res_net_params(self, layer_num):
        if layer_num == 18:
            return BasicBlock, [2, 2, 2, 2]
        if layer_num == 34:
            return BasicBlock, [3, 4, 6, 3]
        if layer_num == 50:
            return Bottleneck, [3, 4, 6, 3]
        if layer_num == 101:
            return Bottleneck, [3, 4, 23, 3]
        if layer_num == 152:
            return Bottleneck, [3, 8, 36, 3]

    def forward(self, x):
        x = self.conv_pool_layer(x)
        x = self.res_layers(x)
        x = self.avg_pool_layer(x)
        x = self.fc_layer(x)
        return x
```

主模型 ResNet 调用其他两个残差块实现完整的残差神经网络，我将其归纳

为：卷积池化层、残差层（由多个残差块构成）、平均池化层、全连接层。其中卷积池化层主要目的是将输入图片的通道转为 base_channels 大小，然后残差层调用两个残差模型，不断循环调用（如代码所示，其中 res_net_params 根据 layer_num 来确定使用 BasicBlock 还是 Bottleneck 以及各层的个数，然后通过 res_layer 函数组合各个 Block，最后组合成 res_layers），平均池化层是将所有特征响应图平均池化成 1*1 的图，然后放入全连接层模拟函数。

ResNet 实验结果（由于实验模型中无 dropout，因此并不将其作为超参）：

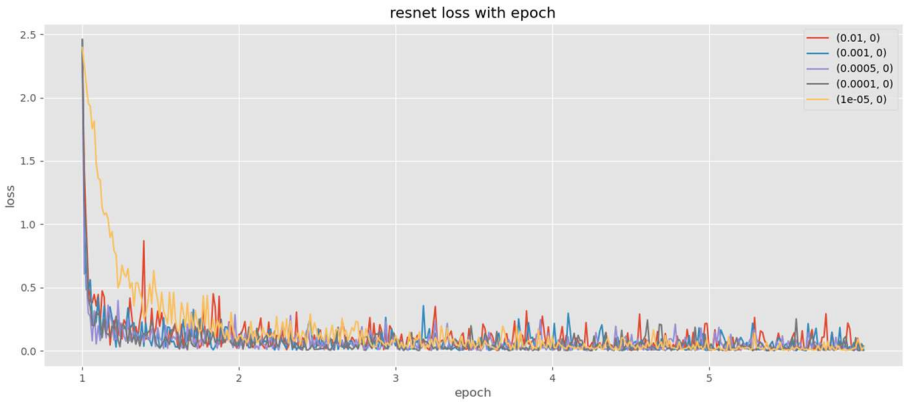

图 20 ResNet loss 随 epoch 的变化（不同学习率）

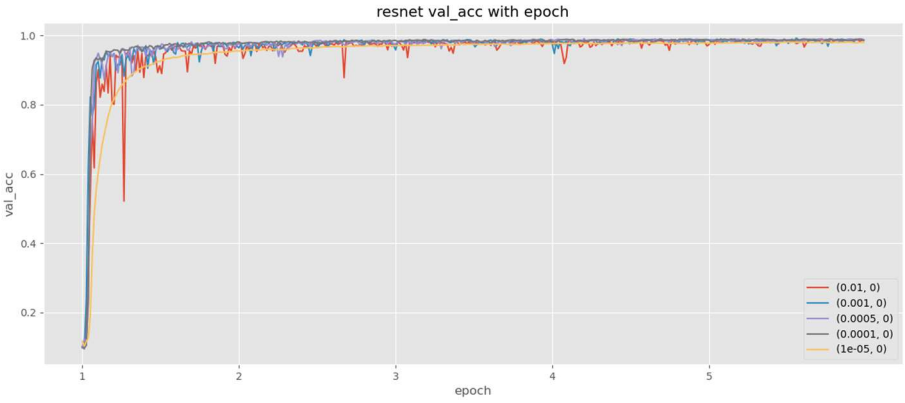

图 21 ResNet acc 随 epoch 的变化（不同学习率）

从 loss/acc 随 epoch 的变化曲线来看，ResNet 的表现非常优秀，最后都能趋向较好的结果，只是当学习率过大的时候波动起伏较大，而学习率较小的时候收敛较慢。

因此我们挑选起伏较小，收敛较快的学习率 0.0005 作为模型比较的学习率。

最后这五组参数在测试集上的结果：

图 22 ResNet 在测试集的表现

虽然模型选择并没有选择在测试集表现较好的 0.001 学习率，但是我们在测试前是无法得知测试集的，而且二者差距不大，可能是存在误差，因此还是选择了 0.0005 作为比较时的学习率参数。

五大网络实验对比：

我选择了 LeNet(learning rate 0.001, dropout 0)、AlexNet(learning rate 0.001, dropout 0.2)、VGGNet(learning rate 0.001, dropout 0.4)、GoogLeNet(learning rate 0.0005, dropout 0.2)、ResNet(learning rate 0.0005, dropout 0)，以所有训练集训练，最终在测试集上有如下的结果。

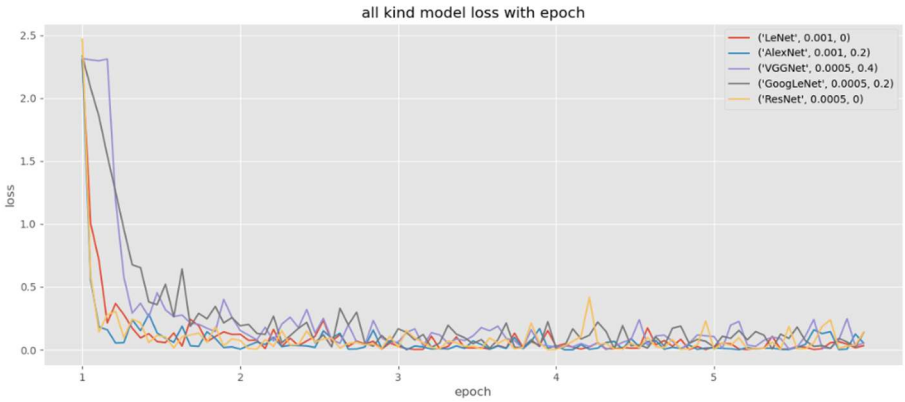

图 23 五个经典网络 loss 随 epoch 变化

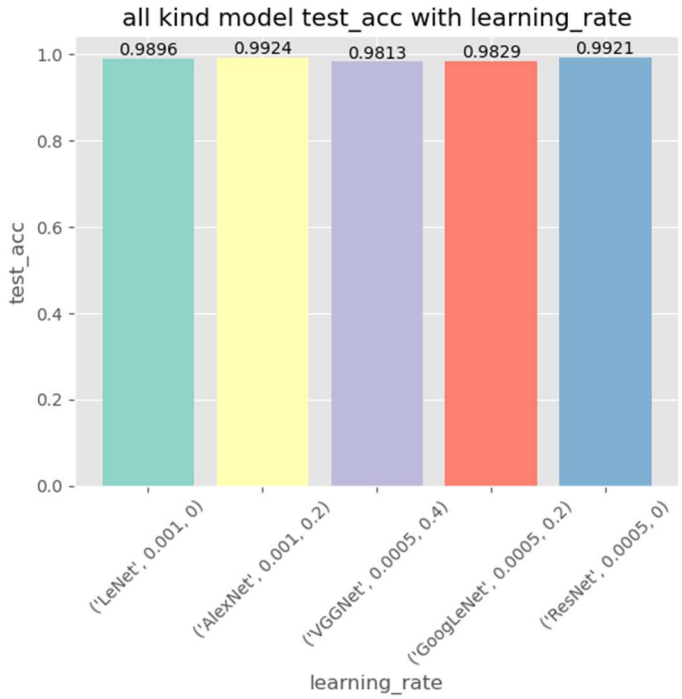

图 24 五个经典网络结果对比

从测试集结果来看 AlexNet 效果最好，当然各个网络之间相差并不大，个人认为是**因为 MNIST 数据集过于简单，后面的 VGGNet、GoogLeNet、ResNet 并没有体现出它们的优势**。其实在网络结构设想和结果来说，LeNet 在 MNIST 数据集上就有着相当不错的理论基础和实验效果：提取 MNIST 数据集的“线稿”，再将其放入全连接层拟合出函数实现分类器。而 AlexNet 则相当于进一步保证了特征提取的准确性和添加了 dropout 防止过拟合，因而效果上比 LeNet 更好。VGGNet 和 GoogLeNet 过于深层次，训练参数过多反而更不好了，有种“退化现象”的感觉。ResNet 目的也是为了缓解“退化现象”而提出的，因而效果也是较好的。

实验结论：

本次实验中实现了 LeNet、AlexNet、VGGNet、GoogLeNet、ResNet 五种经典的 CNN 网络，并进一步熟练运用了 Pytorch 框架，同时学习使用了 argparse 包用于解析命令行命令。

从亲手实现这些网络中也体会到了各个网络各层的想法，并加以自己的理解，在其中也有一些难以处理的 bug，也要调整各个层数的参数使之适合 MNSIT 数据集，好在有部分查阅资料后可以解决，但是在辅助分类器在框架内如何反向传播还是存在疑问。

在实验结果中，虽然 AlexNet 效果较好，但是大部分原因还是 MNIST 过于简单，在使用这些网络结构训练人脸识别数据集时，后面的网络优势慢慢就体现出来了，尤其是 ResNet，大部分学习率均可以达到较优的结果，也进一步说明了残差模块更加有实际应用意义。

参考文献：

[1] Y. Lecun, L. Bottou, Y. Bengio and P. Haffner, "**Gradient-based learning applied to document recognition**," in Proceedings of the IEEE, vol. 86, no. 11, pp. 2278-2324, Nov. 1998, doi: 10.1109/5.726791.

[2] Krizhevsky, Alex et al. "**ImageNet classification with deep convolutional neural networks**." Communications of the ACM 60 (2012): 84 - 90.

[3] Simonyan, Karen and Andrew Zisserman. "**Very Deep Convolutional Networks for Large-Scale Image Recognition**." CoRR abs/1409.1556 (2015): n. pag.

[4] Szegedy, Christian et al. "**Going deeper with convolutions.**" 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (2015): 1-9.

[5] He, Kaiming et al. "**Deep Residual Learning for Image Recognition.**" 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (2016): 770-778.

[6] Zeiler, Matthew D. and Rob Fergus. "**Visualizing and Understanding Convolutional Networks.**" ECCV (2014).