## 卷积神经网络实验报告

姓名：张铭徐 学号：2113615

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

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

同样，在本次实验中，我们使用8卡3090进行实验，并使用Wandb记录训练数据，我们本次实验并未使用提供版本CNN，使用的是自行实现的CNN。

我们自行实现的网络结构如下所示：

class Net(nn.Module):

    def \_\_init\_\_(self):

        super(Net, self).\_\_init\_\_()

        self.conv1 = nn.Conv2d(3, 32, 3, padding=1)

        self.conv2 = nn.Conv2d(32, 64, 3, padding=1)

        self.conv3 = nn.Conv2d(64, 128, 3, padding=1)

        self.conv4 = nn.Conv2d(128, 256, 3, padding=1)

        self.pool = nn.MaxPool2d(2, 2)

        self.fc1 = nn.Linear(256 \* 2 \* 2, 500)

        self.fc2 = nn.Linear(500, 10)

    def forward(self, x):

        x = self.pool(F.relu(self.conv1(x)))

        x = self.pool(F.relu(self.conv2(x)))

        x = self.pool(F.relu(self.conv3(x)))

        x = self.pool(F.relu(self.conv4(x)))

        x = x.view(-1, 256 \* 2 \* 2)

        x = F.relu(self.fc1(x))

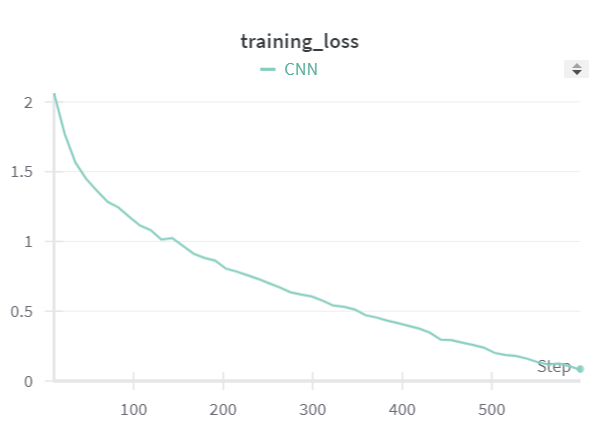
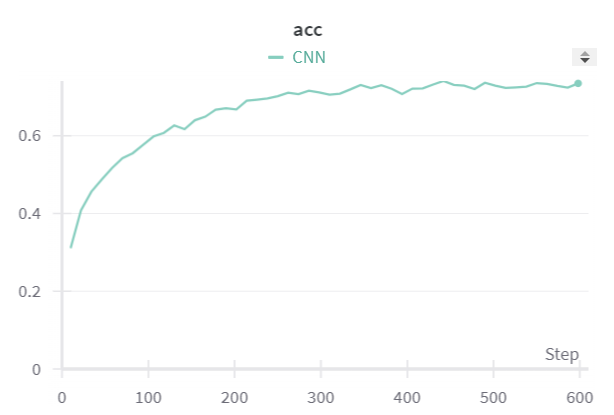
        x = self.fc2(x)

        return x

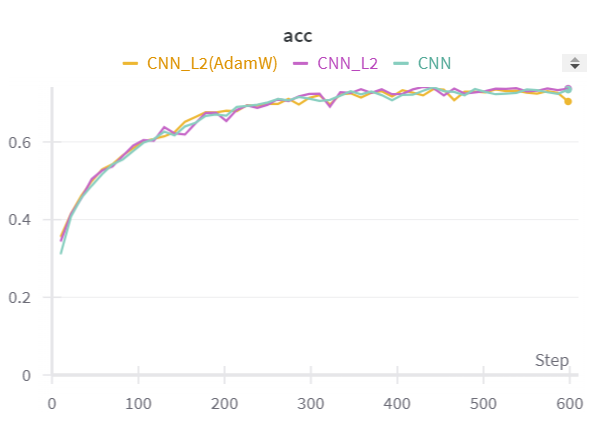
我们将对应网络结构整理，使用graphviz进行可视化，得到以下结构：



在CNN中，我们同时使用卷积和池化操作，我们使用wandb记录每个类别的准确率和训练损失以及准确率，如下图所示：



最终对于原始版本的CNN，我们可以取得73.46%的准确率，我们考虑采用多种惩罚方法和优化器，如下图所示，准确率并没有显著性差异。



接下来我们考虑实现ResNet结构，ResNet是由若干Residual Block构成的，每一个子Block内部都具有跳链接的性质，我们在这里给出实现的ResNet：

class ResidualBlock(nn.Module):

    def \_\_init\_\_(self, in\_channels, out\_channels, stride=1):

        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.shortcut = nn.Sequential()

        if stride != 1 or in\_channels != out\_channels:

            self.shortcut = nn.Sequential(

                nn.Conv2d(in\_channels, out\_channels, kernel\_size=1, stride=stride, bias=False),

                nn.BatchNorm2d(out\_channels)

            )

    def forward(self, x):

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

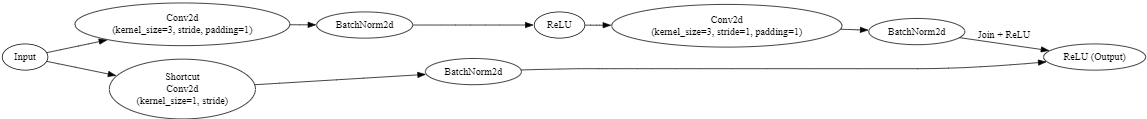
        out = self.bn2(self.conv2(out))

        out += self.shortcut(x)

        out = F.relu(out)

        return out

在每一个Residual Block的输出端，都是前一层神经元的输出加上最原始的输入，这样做可以有效的防止梯度消失的现象出现，也正是由于ResNet，使得深度神经网络层数变得更深成为了可能。下图为我们实现的Residual Block的示意图。



在给出Residual Block后，我们可以着手实现ResNet整体的网络结构：

class ResNet(nn.Module):

    def \_\_init\_\_(self, block, num\_blocks, num\_classes=10, dropout\_rate=0.7):

        super(ResNet, self).\_\_init\_\_()

        self.in\_channels = 32

        self.conv = nn.Conv2d(3, 32, kernel\_size=3, stride=1, padding=1, bias=False)

        self.bn = nn.BatchNorm2d(32)

        self.layer1 = self.make\_layer(block, 32, num\_blocks[0], stride=1)

        self.layer2 = self.make\_layer(block, 64, num\_blocks[1], stride=2)

        self.layer3 = self.make\_layer(block, 128, num\_blocks[2], stride=2)

        self.layer4 = self.make\_layer(block, 256, num\_blocks[3], stride=2)

        self.dropout = nn.Dropout(dropout\_rate)

        self.fc = nn.Linear(256, num\_classes)

    def make\_layer(self, block, out\_channels, num\_blocks, stride):

        strides = [stride] + [1] \* (num\_blocks - 1)

        layers = []

        for stride in strides:

            layers.append(block(self.in\_channels, out\_channels, stride))

            self.in\_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)

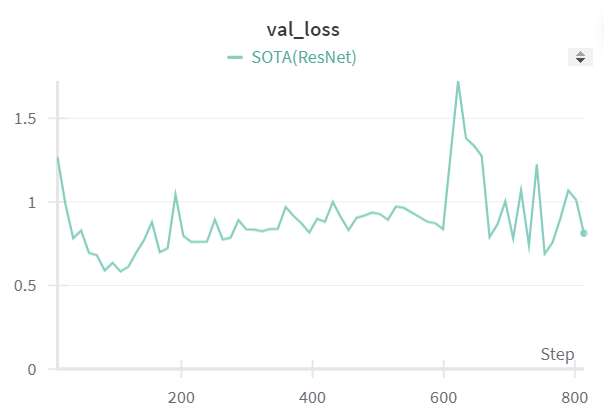
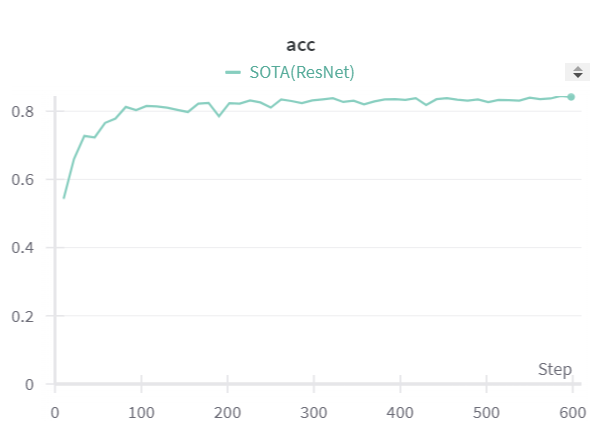
        out = self.dropout(out)

        out = self.fc(out)

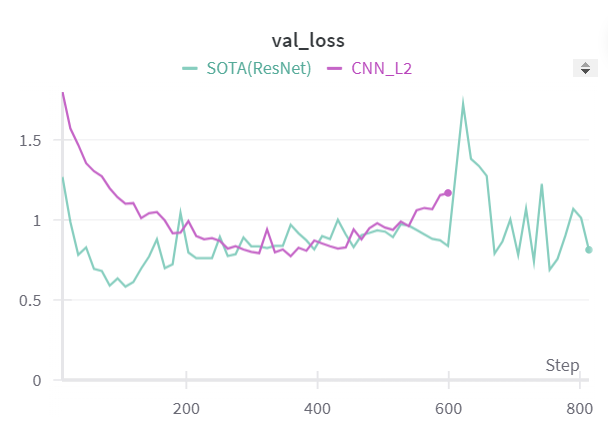
        return out

我们的make\_layer方法负责将多个residual blocks连接起来，并且我们有若干层，每一层都可能堆叠若干个residual Block；由于计算资源等限制，我们在此处实现的仅仅是四层的ResNet，距离标准的ResNet18有一定的差距。

我们对上述模型进行评估测试，经过若干次调整参数和堆叠结构，我们找到了最优参数为[3,4,6,4]，即四层中分别有3个，4个，6个，4个residual blocks，并且学习率设定为5e-4进行实验，我们给出训练曲线：



最终我们取得了84.2%的准确率，相较于原本的73.46%，提高了10%以上的准确率。



通过对比ResNet和原本CNN的训练曲线，我们可以发现，ResNet可以有效的减少过拟合的现象发生，并且在整个训练过程中，也更不容易陷入局部最优解。对于ResNet在训练过程中突然超调的现象，我们推测其原因为学习率相对大，令模型直接跑出了优化域外，但是随着后续梯度下降的进行，模型最后的loss也在随之下降直到收敛。

接下来我们考虑DenseNet；相较于ResNet而言，其网络结构是更加稠密的，网络中的每一层都与之前的所有层通过特征图连接，我们直接给出对应代码：

class Bottleneck(nn.Module):

    def \_\_init\_\_(self, in\_channels, growth\_rate, dropout\_rate=0.75):

        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)

        self.dropout = nn.Dropout(dropout\_rate)

    def forward(self, x):

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

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

        out = self.dropout(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)

    def forward(self, x):

        out = self.conv(F.relu(self.bn(x)))

        out = F.avg\_pool2d(out, 2)

        return out

class DenseNet(nn.Module):

    def \_\_init\_\_(self, num\_blocks, growth\_rate=64, reduction=0.8, num\_classes=10):

        super(DenseNet, self).\_\_init\_\_()

        num\_channels = 2 \* growth\_rate

        self.conv1 = nn.Conv2d(3, num\_channels, kernel\_size=3, stride=1, padding=2, bias=False)

        self.dense1 = self.make\_dense\_layers(Bottleneck, num\_channels, growth\_rate, num\_blocks[0])

        num\_channels += num\_blocks[0] \* growth\_rate

        out\_channels = int(reduction \* num\_channels)

        self.trans1 = Transition(num\_channels, out\_channels)

        num\_channels = out\_channels

        self.dense2 = self.make\_dense\_layers(Bottleneck, num\_channels, growth\_rate, num\_blocks[1])

        num\_channels += num\_blocks[1] \* growth\_rate

        out\_channels = int(reduction \* num\_channels)

        self.trans2 = Transition(num\_channels, out\_channels)

        num\_channels = out\_channels

        self.dense3 = self.make\_dense\_layers(Bottleneck, num\_channels, growth\_rate, num\_blocks[2])

        num\_channels += num\_blocks[2] \* growth\_rate

        out\_channels = int(reduction \* num\_channels)

        self.trans3 = Transition(num\_channels, out\_channels)

        num\_channels = out\_channels

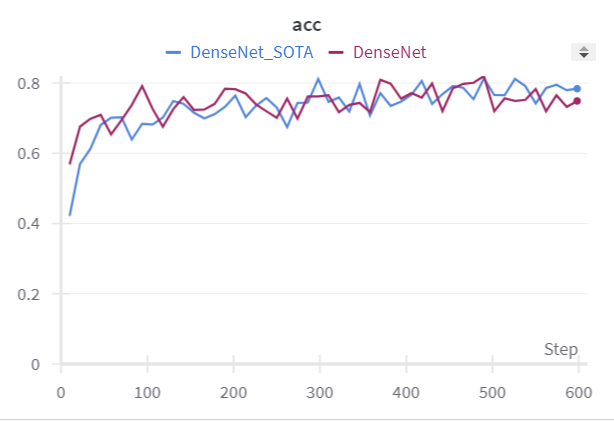
        self.dense4 = self.make\_dense\_layers(Bottleneck, num\_channels, growth\_rate, num\_blocks[3])

        num\_channels += num\_blocks[3] \* growth\_rate

        self.bn = nn.BatchNorm2d(num\_channels)

        self.fc = nn.Linear(num\_channels, num\_classes)

我们同样对于DenseNet调节参数，寻找到最佳组合：其学习率为1e-4，且每层由6，12，24, 48个Blocks组成。



上图给出了我们跑出最优情况下的DenseNet和平常DenseNet的曲线图，我们最终跑出最好的DenseNet的准确率为75%；由于其网络参数有限，且没有残差连接，所以其性能要略逊色于ResNet，但是其训练过程相对更平滑。

对于带有SE结构的ResNet，我们使用Squeeze and Excitation Block，其主要有三个功能，第一个为Squeeze：我们通过全局平均池化将每个通道的空间信息压缩成一个单一的标量，这样可以总结每个通道的全局信息；第二个为Excitation，这个标量通过一个全连接层首先将通道数降低，再升高，这个过程中引入非线性激活函数（ReLU）。这样做可以学习到一个从全局信息中挤压出的每个通道的重要性权重。第三个是 Scale，通过 sigmoid 函数将权重标准化到 [0, 1] 的范围，这些权重随后被用来缩放原始特征图的通道。这种缩放操作是按元素进行的，可以动态调整每个通道的重要性，整体上来看，类似于“通道attention”.我们给出实现的代码如下：

# implement a CNN with resiudal block

class SEBlock(nn.Module):

    def \_\_init\_\_(self, channel, reduction=16):

        super(SEBlock, self).\_\_init\_\_()

        self.avg\_pool = nn.AdaptiveAvgPool2d(1)

        self.fc = nn.Sequential(

            nn.Linear(channel, channel // reduction, bias=False),

            nn.ReLU(inplace=True),

            nn.Linear(channel // reduction, channel, 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)

class ResidualBlockWithSE(nn.Module):

    def \_\_init\_\_(self, in\_channels, out\_channels, stride=1):

        super(ResidualBlockWithSE, self).\_\_init\_\_()

        # Residual block components

        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.shortcut = nn.Sequential()

        if stride != 1 or in\_channels != out\_channels:

            self.shortcut = nn.Sequential(

                nn.Conv2d(in\_channels, out\_channels, kernel\_size=1, stride=stride, bias=False),

                nn.BatchNorm2d(out\_channels)

            )

        self.se = SEBlock(out\_channels)  # SE Block

    def forward(self, x):

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

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

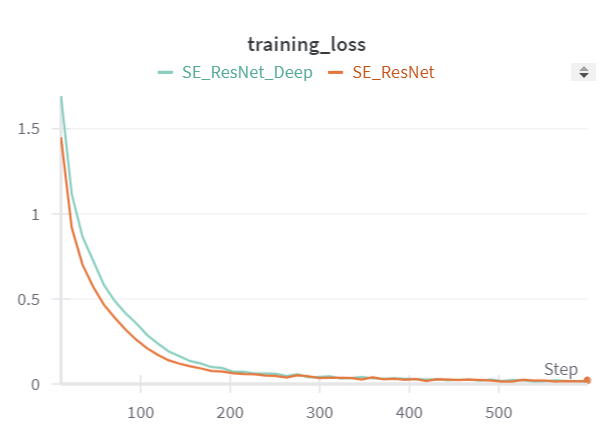
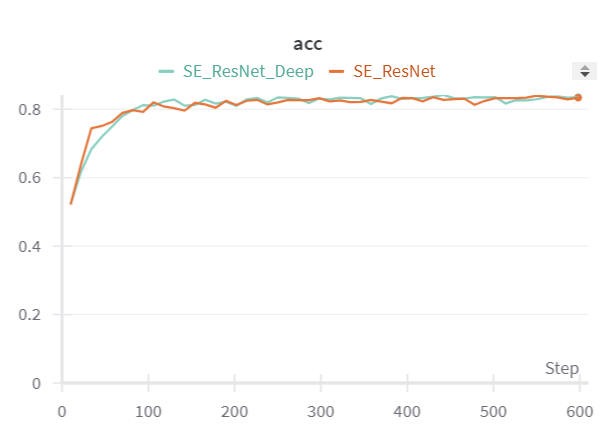
        out = self.se(out)  # Apply SE block

        out += self.shortcut(x)

        out = F.relu(out)

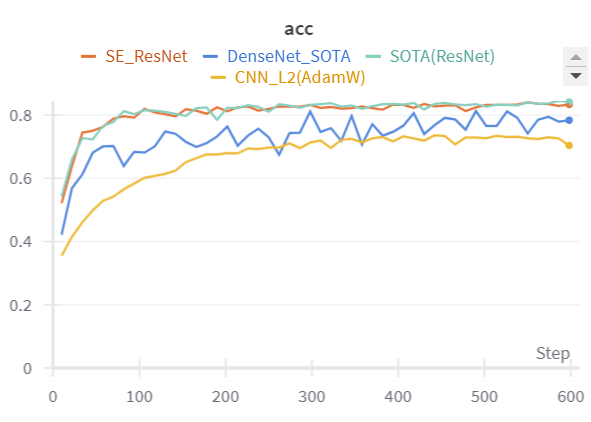
        return out

· 我们在实验中，参数设定与ResNet完全一致，对比其性能差异：



并且我们同时简单加深了网络层数测试其性能，我们发现，性能差距很小，最终我们能够获得84%的准确率，要高于ResNet。

我们给出上述所有模型的性能对比图：



对于训练过程中的loss下降，我们可以发现，对于普通的CNN整体上loss下降速度相对比较缓慢，这是由梯度决定的，我们知道反向传播更新参数是由当前梯度决定，其梯度下降较为缓慢说明其梯度较小。我们知道，跳连的主要作用就是提供一个out = f(x) + x这样一个函数，这样我们对out 求偏导后，不仅仅由f’(x)决定，还有一个系数1，这样可以有效避免梯度消失的现象，那么也就是说，反应在训练过程中，其梯度就要更大，收敛的速度就要更快，loss下降速度也就更快了。