多媒体信息处理作业

姓名：张宗梓
专业：人工智能
学号：2023K8009991013

目录

1 实验背景与任务 **2**

1.1 CIFAR-10 数据集 2

1.2 ResNet 网络架构 2

1.3 实验配置 4

2 激活函数的对比与分析 **5**

2.1 数学原理 5

2.1.1 Sigmoid 5

2.1.2 ReLU 5

2.1.3 LeakyReLU 5

2.1.4 GELU 5

2.2 实验结果对比 6

3 优化策略的对比与分析 **7**

3.1 数学原理 7

3.1.1 SGD (随机梯度下降) 7

3.1.2 SGD with Momentum 8

3.1.3 Adam (Adaptive Moment Estimation) 8

3.2 实验结果对比 9

4 模块设计创新：Ghost Module **10**

4.1 改进动机 10

4.2 改进原理 10

4.3 代码实现 11

4.4 性能评估 12

5 实验总结 **14**

1 实验背景与任务

1.1 CIFAR-10 数据集

CIFAR-10 是计算机视觉领域经典的图像分类数据集，包含 10 个类别，共 60000 张 32×32 彩色图像。其中训练集 50000 张，测试集 10000 张。

1.2 ResNet 网络架构

ResNet 通过引入“残差学习”解决了深层网络中的梯度消失和退化问题。本实验采用 ResNet-18 作为基础架构。

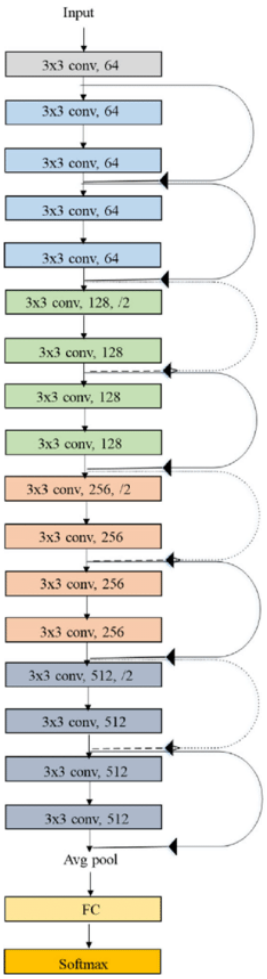


图 1: ResNet-18 结构示意图

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

    def __init__(self, in_planes, planes, stride=1, activation_class=nn.ReLU):
        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.act1 = activation_class()

        self.conv2 = nn.Conv2d(planes, planes, kernel_size=3, stride=1, padding
    ↪ =1, bias=False)
        self.bn2 = nn.BatchNorm2d(planes)
        self.act2 = activation_class()

        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.act1(self.bn1(self.conv1(x)))
        out = self.bn2(self.conv2(out))
        out += self.shortcut(x)
        out = self.act2(out)
        return out

class ResNet(nn.Module):
    def __init__(self, block, num_blocks, num_classes=10, activation_class=nn.
    ↪ ReLU):
        super(ResNet, self).__init__()
        self.in_planes = 64
        self.activation_class = activation_class
        self.conv1 = nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1, bias=
    ↪ False)
        self.bn1 = nn.BatchNorm2d(64)
        self.act1 = activation_class()

        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)
```

```
41
42          self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
43          self.linear = nn.Linear(512 * block.expansion, num_classes)
44
45      def _make_layer(self, block, planes, num_blocks, stride):
46          strides = [stride] + [1]*(num_blocks-1)
47          layers = []
48          for stride in strides:
49              layers.append(block(self.in_planes, planes, stride, self.
    ↪ activation_class))
50              self.in_planes = planes * block.expansion
51          return nn.Sequential(*layers)
52
53      def forward(self, x):
54          out = self.act1(self.bn1(self.conv1(x)))
55          out = self.layer1(out)
56          out = self.layer2(out)
57          out = self.layer3(out)
58          out = self.layer4(out)
59          out = self.avgpool(out)
60          out = out.view(out.size(0), -1)
61          out = self.linear(out)
62          return out
63
64  def ResNet18(activation_class):
65      return ResNet(BasicBlock, [2, 2, 2, 2], activation_class=activation_class)
```

Listing 1: ResNet 核心代码

1.3 实验配置

本实验基于 PyTorch 框架实现，使用云端 Nividia 4090 GPU + 4颗cpu核心进行训练。主要超参数设置如下：

- 批大小 (Batch Size)：128
- 训练轮数 (Epochs)：40
- 学习率 (Learning Rate)：0.01
- 优化器 (Optimizer)：SGD / Adam / SGD with Momentum
- 激活函数 (Activation Function)：ReLU / LeakyReLU / Sigmoid / GELU

2 激活函数的对比与分析

激活函数为神经网络引入了非线性因素。本实验在保持 ResNet-18 结构和优化策略以及超参数不变的情况下，对比了四种激活函数的性能表现。

2.1 数学原理

2.1.1 Sigmoid

Sigmoid 函数将输入映射到 $(0, 1)$ 区间：

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{1}$$

缺点：在 x 绝对值较大时，导数趋近于 0，容易导致梯度消失；且输出不是以 0 为中心的。

2.1.2 ReLU

ReLU 是目前最常用的激活函数，公式如下：

$$\text{ReLU}(x) = \max(0, x) \tag{2}$$

优点：计算简单，在正区间解决了梯度消失问题，收敛速度快。

2.1.3 LeakyReLU

为了解决 ReLU 在 $x < 0$ 时神经元“死亡”的问题，LeakyReLU 引入了一个小的负斜率（实验中默认 slope=0.01）：

$$\text{LeakyReLU}(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha x & \text{if } x \leq 0 \end{cases} \tag{3}$$

优点：在负区间也有非零输出，缓解了神经元死亡问题。

2.1.4 GELU

GELU 在 Transformer 和 BERT 等模型中被广泛应用，它结合了随机正则化的思想：

$$\text{GELU}(x) = x\Phi(x) = x \cdot \frac{1}{2}\left[1 + \text{erf}\left(\frac{x}{\sqrt{2}}\right)\right] \tag{4}$$

优点：GELU 在 $x = 0$ 附近是连续可导的，且在负区间有非零输出，有助于信息流动。

2.2 实验结果对比

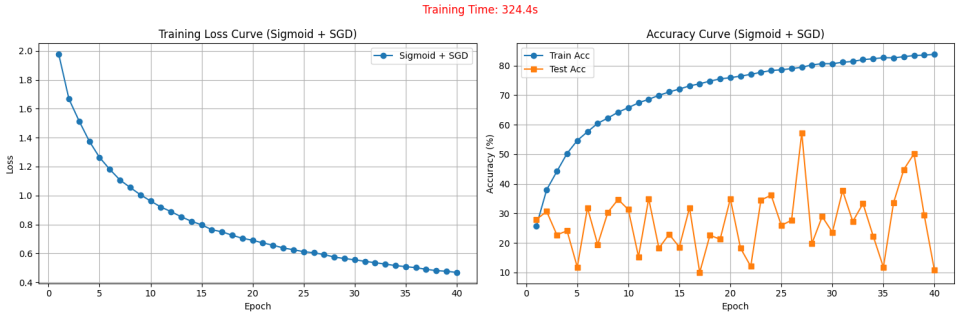


图 2: Sigmoid 激活函数下的训练曲线

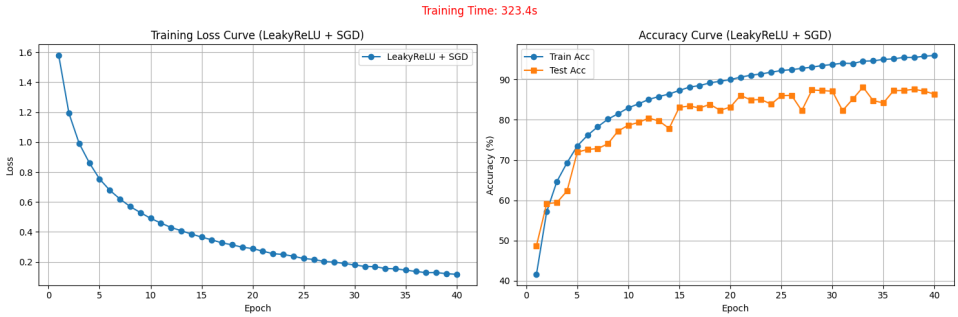


图 3: LeakyReLU 激活函数下的训练曲线

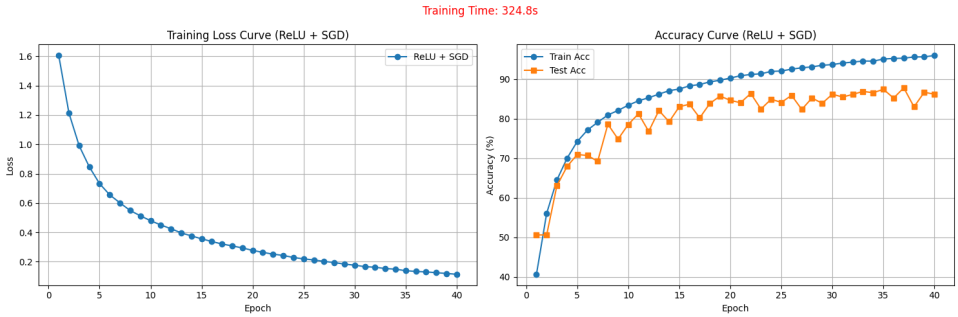


图 4: ReLU 激活函数下的训练曲线

图 5: GELU 激活函数下的训练曲线

结果分析：

- Sigmoid 激活函数表现最差，训练过程震荡较大，最终准确率最低，验证了其梯度消失以及非零中心，配合上SGD优化器这种单纯的梯度下降，导致优化路径不稳定，不利于深层网络训练的缺点。
- ReLU 和 LeakyReLU 表现较好，收敛速度快且稳定，但优化过程中，出现一定程度上的局部抖动。
- GELU 虽然整体趋势与 ReLU 和 LeakyReLU 相似，但其测试集准确率的上升曲线显得更为平滑。由于在 $x = 0$ 附近是连续可导的，不像 ReLU 那样有一个尖锐的折点。这种平滑性有助于 SGD 优化器在逼近极小值时更加顺畅，减少了优化过程中的局部抖动。

3 优化策略的对比与分析

优化器决定了网络参数更新的方式。实验在固定学习率 ($LR = 0.01$) 和激活函数的条件下对比了以下三种优化器。

3.1 数学原理

3.1.1 SGD (随机梯度下降)

最基本的参数更新方法，每次迭代使用一个 Batch 的数据计算梯度：

$$w_{t+1} = w_t - \eta \nabla L(w_t) \tag{5}$$

缺点：收敛速度慢，容易陷入局部极小值。

3.1.2 SGD with Momentum

为了抑制震荡并加速收敛，引入动量项 v_t（实验中 $\mu = 0.9$）：

$$v_{t+1} = \mu v_t + \nabla L(w_t) \tag{6}$$

$$w_{t+1} = w_t - \eta v_{t+1} \tag{7}$$

优点：动量项帮助参数更新沿着历史梯度的平均方向前进，减少震荡，加速收敛。

3.1.3 Adam (Adaptive Moment Estimation)

Adam 结合了动量法和 RMSProp，自适应地调整每个参数的学习率。它利用了一阶矩估计 m_t 和二阶矩估计 v_t：

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \tag{8}$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \tag{9}$$

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1 - \beta_2^t} \tag{10}$$

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t \tag{11}$$

优点：自适应调整学习率，适合处理稀疏梯度和非平稳目标，收敛速度快。

3.2 实验结果对比

图 6: Adam 优化器下的训练曲线

图 7: SGD with Momentum 优化器下的训练曲线

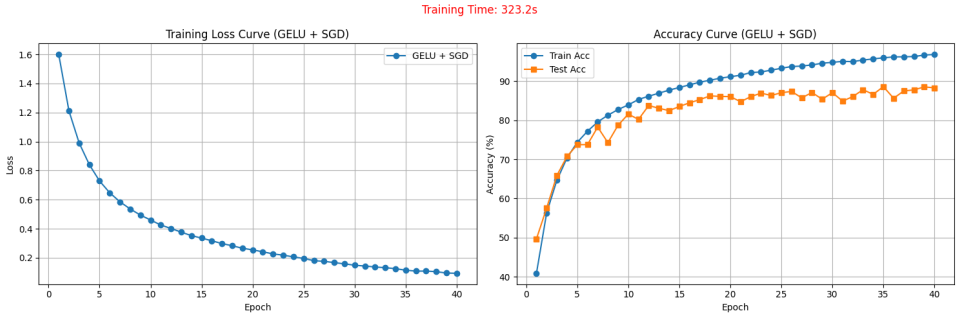


图 8: SGD 优化器下的训练曲线

结果分析：

- Adam 优化器的测试集准确率在 45% 到 65% 之间剧烈跳动，且最终测试集准确率只有 60%-65% 左右，远低于 SGD。这可能是因为Adam 收敛虽快，但往往收敛到“尖锐的极小值”，而 SGD 更容易收敛到“平坦的极小值”，平坦极小值的泛化能力更好。此外，Adam对学习率相当敏感，固定学习率0.01可能并非最佳选择，导致性能不佳。

- SGD with Momentum 在此次实验中表现优异。在收敛速度上优于纯 SGD，且最终测试集准确率较高，说明动量项有助于跳出局部极小值，找到更优的全局解。

- 纯 SGD 收敛最慢，但训练过程较为平稳，最终的测试集准确率也较高。但是标准

SGD 每次更新只依赖当前的梯度 g_t。虽然它最终能找到不错的极小值，但在遇到峡谷状的损失曲面时，SGD 容易在坑壁间来回震荡，导致收敛速度受限。

4 模块设计创新：Ghost Module

4.1 改进动机

在传统的 CNN 中，特征图通常存在大量的冗余（相似的特征图）。*GhostNet* : *MoreFeaturesfromCheapOperations* 指出，这些冗余特征图可以通过廉价的线性变换从少量“本质特征图”生成，从而减少计算量和参数量。

4.2 改进原理

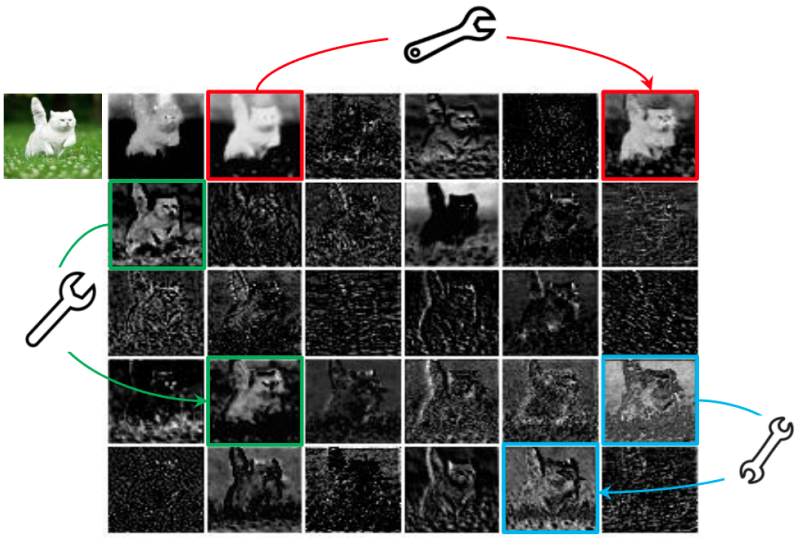


图 9: ResNet-50 特征图相似性示意图

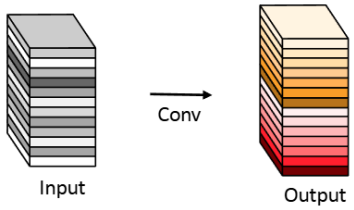
如图，论文作者观察到，在训练好的深度神经网络（如 ResNet-50）中，特征图之间存在惊人的相似性，例如很多特征图都可以由一张图经过微小的变换生成。既然这些图高度相关，那么我们就没有必要用独立的卷积核去学习它们，而是可以通过廉价的线性变换生成这些冗余特征图，从而节省计算资源。

Table 2. MSE error *v.s.* different kernel sizes.

MSE (10^{-3})	d=1	d=3	d=5	d=7
red pair	4.0	3.3	3.3	3.2
green pair	25.0	24.3	24.1	23.9
blue pair	12.1	11.2	11.1	11.0

图 10: 特征图拟合实验结果

作者做了一个简单的实验：取 ResNet-50 中的一对相似特征图，尝试用一个微小的深度卷积核去拟合它们之间的变换关系。结果发现，重建误差（MSE）极小。这证明了复杂的卷积操作生成的很多特征，完全可以用极低成本的线性变换来替代，而不会丢失太多信息。



(a) The convolutional layer.

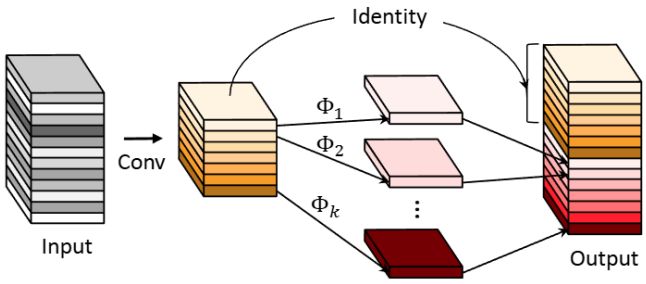


图 11: Ghost Module 原始结构示意图

如图是$GhostNet:MoreFeaturesfromCheapOperations$论文中的 Ghost Module 结构示意图。在Ghost Module 中，首先通过普通卷积核，且核和通道的数量少于原始计划，生成“本质特征图”。然后通过廉价的线性操作(本实验采用卷积$nn.Conv2d$)生成冗余特征图。最后将两部分特征图拼接，得到最终输出。

4.3 代码实现

本实验基于$GhostNet:MoreFeaturesfromCheapOperations$，设计了 `GhostModule` 替代 ResNet 中的标准卷积层。核心代码如下：

```python
class GhostModule(nn.Module):
    def __init__(self, inp, oup, kernel_size=1, ratio=2, dw_size=3, stride=1, relu=True):
        super(GhostModule, self).__init__()
        self.oup = oup
        init_channels = math.ceil(oup / ratio)
        new_channels = init_channels * (ratio - 1)
        self.primary_conv = nn.Sequential(
            nn.Conv2d(inp, init_channels, kernel_size, stride, kernel_size//2, bias=False),
            nn.BatchNorm2d(init_channels),
            nn.ReLU(inplace=True) if relu else nn.Sequential(),
        )
        self.cheap_operation = nn.Sequential(
            nn.Conv2d(init_channels, new_channels, dw_size, 1, dw_size//2, groups=init_channels, bias=False),
            nn.BatchNorm2d(new_channels),
            nn.ReLU(inplace=True) if relu else nn.Sequential(),
        )
    def forward(self, x):
        x1 = self.primary_conv(x)
        x2 = self.cheap_operation(x1)
        out = torch.cat([x1, x2], dim=1)
        return out[:, :self.oup, :, :]
```

Listing 2: Ghost Module 核心实现代码

我们将原 ResNet 中的 3×3 卷积替换为 `GhostModule`，设置 `ratio=2`，意味着理论上减少了一半的计算量。

4.4 性能评估

将改进后的 ResNet-Ghost 与原始 ResNet-18 进行对比：

表 1: ResNet-18 与 ResNet-Ghost 大小对比

Model	Parameters
ResNet-18	11.17M
ResNet-Ghost	5.7M

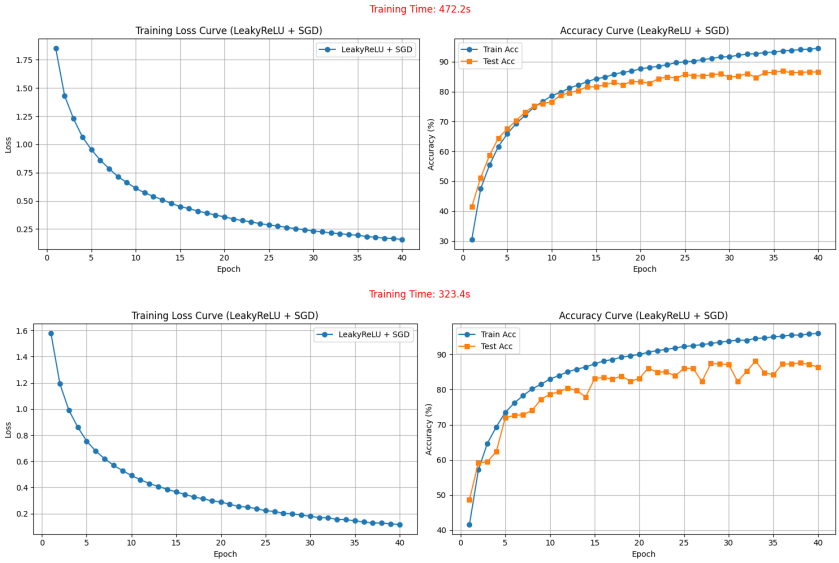


图 12: 训练过程对比1：上为 ResNet-Ghost，下为原 ResNet-18

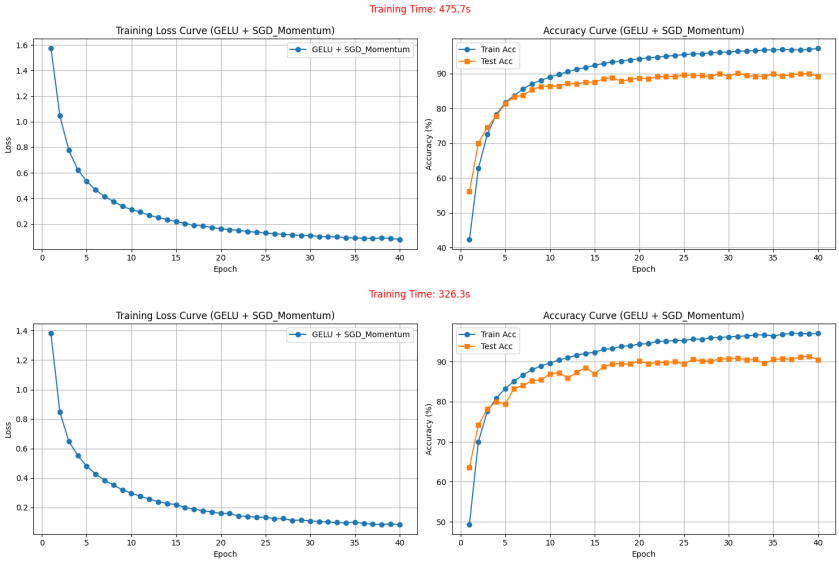


图 13: 训练过程对比2：上为 ResNet-Ghost，下为原 ResNet-18

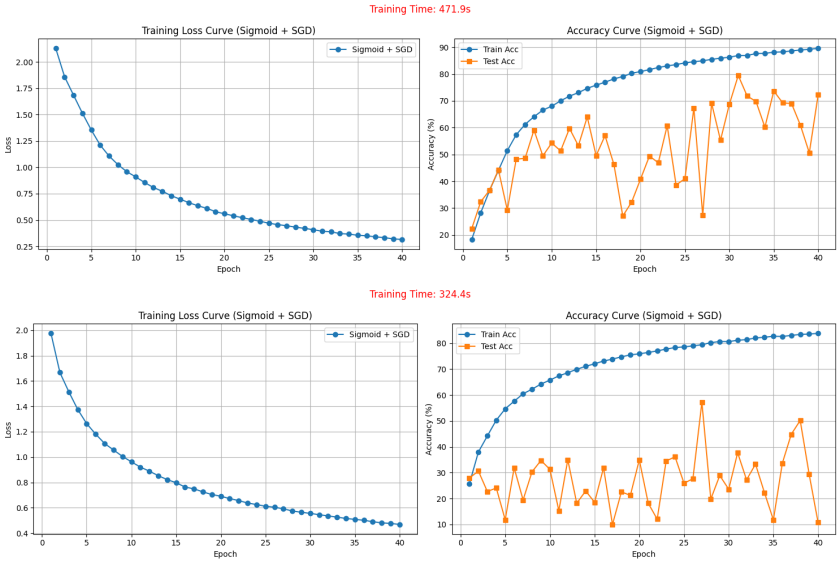


图 14: 训练过程对比3：上为 ResNet-Ghost，下为原 ResNet-18

分析

- Ghost 版本在参数量砍半的情况下，最终的测试集准确率几乎没有明显的损失,验证了特征图冗余性的假设。
- 虽然参数减少，但是由于Ghost Module 将一个大卷积拆成了两步：少量标准卷积 + Depthwise 卷积 + Concat。这增加了操作的碎片化程度。GPU 需要启动更多的 Kernel，频繁的 Kernel Launch 开销抵消了计算量的减少，这体现在训练时间不降反增上。
- 此外，还可以注意到，Ghost 版本的训练曲线更平滑，测试集准确率的波动更小，这可能是因为Ghost Module 自带的“结构正则化”效应，即Ghost Module 限制了模型的表达能力，减少了过拟合的可能性。

5 实验总结

本次实验通过复现 ResNet-18 并进行模块化改进，得出以下结论：

1. 激活函数方面，ReLU 及其变体（LeakyReLU）在深层网络中表现稳定，优于 Sigmoid。
2. 优化器方面，在本实验条件下 SGD+Momentum 能获得更好的最终泛化能力。虽然Adam理论收敛速度快，但是由于对于学习率的敏感性，固定学习率0.01并非最佳选择，导致性能不佳。
3. 通过引入 Ghost Module，模型在保持精度的同时，参数利用率得到了提升，验证了特征图冗余性假设的合理性，且由于结构正则化，测试集准确率的波动减小。