CIFAR-10 数据集实验报告

张以禾
学号：25800980010

2025年10月30日

1 摘要

本实验基于CIFAR-10数据集，探究了卷积神经网络(CNN)的训练过程中的损失函数变化、正则化技术的应用，通过Pytorch实现不同的CNN架构。通过实验分析了正则化在防止过拟合方面的效果。最后基于torchvision上的源码实现了适配CIFAR-10的DenseNet网络，实现了大于92.5%的准确率。

2 引言

图像分类是计算机视觉中的一个重要任务，其目的是将输入图像自动归类到预定义的类别中。卷积神经网络(Convolutional Neural Networks, CNNs)作为一种强大的深度学习模型，已经在图像分类任务中取得了显著的成功,并广泛运用于图像识别、目标检测等领域。经典网络架如LeNet、AlexNet、VGG、ResNet和DenseNet等，均在不同程度上推动了图像分类技术的发展，也为模型结构设计与训练策略提供了宝贵的经验。

CIFAR-10数据集作为一个经典的图像分类数据集，包含了10个类别的60000张32x32彩色图像，适用于中小规模CNN模型的训练与评估。通过在CIFAR-10数据集上进行实验，可以深入理解CNN的工作原理、训练过程中的挑战以及如何通过调整网络结构和训练策略来提升模型性能。本实验中，我们将基于CIFAR-10数据集和Pytorch，从LeNet网络结构出发，训练并构建CNN模型以完成CIFAR-10分类任务。实验内容包括绘制损失函数曲线，引入正则化技术和dropout层以探究其影响。通过这些实验深入理解CNN的训练过程，探索不同技术对模型性能的影响。

最后，为进一步探究网络结构对性能的影响，我们基于torchvision上的DenseNet源码设计并实现了适配CIFAR-10的小型DenseNet网络，并进行了训练与评估。

另外，在未说明的情况下，实验均在NVIDIA GeForce RTX 5060 Laptop GPU上进行，使用PyTorch深度学习框架，batch size设为64，优化器使用SGD，学习率设为0.001，动量(momentum)设为0.9，训练周期(epoch)设为100，Dropout概率（如有）设为0.5。同时网络结构重命名为net。

3 实验方法

3.1 Task1：绘制损失函数曲线

3.1.1 问题重述

在本实验中，我们将使用 CIFAR-10 数据集来训练一个卷积神经网络（CNN），根据以及提供好的代码框架补全绘图代码，绘制训练和测试损失函数和正确率曲线。

3.1.2 实验思路

为了绘制损失-准确率函数曲线，我们需要在每个训练和测试周期结束时记录损失值。具体步骤如下：

1. **初始化数据结构**：在训练开始前，创建空列表 `train_losses` ，用于存储每个 epoch 的训练和测试损失值。
2. **修改训练循环**：在每个 epoch 的训练循环结束后，计算并记录当前 epoch 的平均训练损失值，并将其添加到 `train_losses` 列表中。
3. **测试模型**：在每轮训练结束后，使用测试数据集评估模型性能，记录 `test_acc` 和 `train_acc` 。
4. **绘制损失曲线**：在训练完成后，将数据传入自定义的draw_loss_and_accuracy_curve函数，使用 Matplotlib 库绘制训练和测试损失曲线。X轴表示 epoch 数，Y轴表示损失值和准确率。同时使用不同颜色的线条区分训练和测试损失曲线及正确率，并添加图例、标题和轴标签以提高可读性。
5. **保存图像**：最后，将绘制的图像保存为文件。通过以上步骤，我们可以清晰地观察到模型在训练过程中的损失和准确率变化情况，从而评估模型的训练效果和泛化能力。

3.2 Task2：加入正则化

3.2.1 问题重述

我们需要通过修改Net 和optimizer 构建L2 正则化和暂退法(dropout)，其中dropout 要求在第一和第二线性层之间加入。

3.2.2 实验思路

首先来阐述为什么要加入正则化，正则化的出现是为了对抗过拟合这种现象的。所谓过拟合就是模型在训练数据上拟合的比在潜在分布中更接近的现象（overfitting）。可以想象，当具有足够多的神经元、层数和训练迭代周期，模型最终可以在训练集上达到完美的精度，此时测试集的准确性却下降了，这样的结果只能说明模型对训练的数据拟合的很好，但无法应用到更广泛的常场景，其泛化能力很差。而通过正则化，将限制限制模型的复杂度（限制参数的学习），通过在损失函数中加入额外项，惩罚过大的参数或者不必要的激活，从而抑制模型对训练数据的过度记忆，强化模型对数据的“共性特征”提取能力，提升在测试集上的泛化性能。

下面讨论两种正则化的原理。L2 正则化（也称为权重衰减）通过在损失函数中添加权重参数的平方和作为惩罚项，鼓励模型学习较小的权重值，从而减少模型的复杂度。具体来说，L2 正则化会在每次参数更新时对权重进行缩小，防止权重过大导致过拟合。

假设损失函数为 L，权重参数为 W，L2 正则化后的损失函数可以表示为：

$$L' = L + \lambda \cdot \|W\|^2 \tag{1}$$

其中，λ 是正则化强度的超参数，$\|W\|^2$是权重参数的平方和。

其中W为模型的参数，λ 为正则化强度（PyTorch中是`eight_decay`）。从直观上来看，L2 正则化鼓励模型中的权重W尽量小，但不强迫为零，小权重意味着每个神经元对输出的影响更“柔和”，不会过拟合某些训练样本；参数变得更平滑、稳定，不容易被个别训练样本干扰。它不一定让训练loss 更小，但能提升测试集准确率，这就是“泛化能力”的体现。

通过在pytorch 网站上查阅文档可以知道，L2 正则化的实现方法是通过在优化器中加入 `weight_decay`参数来实现的。我们可以在`optimizer = optim.SGD()` 中加入`weight_decay` 参数来实现L2 正则化。在实际代码中，可以直接在优化器中设置 `weight_decay` 参数，例如：

```
optimizer = optim.SGD(model.parameters(), lr=0.01, weight_decay=lambda)
```

接下来讨论Dropout 的原理。Dropout 是一种随机失活技术，在训练过程中，随机将一部分神经元的输出设为零，从而防止神经元之间的过度依赖。通过这种方式，Dropout 强制模型在每次训练迭代中使用不同的子网络，从而提升模型的鲁棒性和泛化能力。具体来说，在每个训练步骤中，对于每个神经元，以一定的概率 p 将其输出设为零，剩余的神经元的输出则按比例缩放，以保持整体输出的期望值不变。

假设神经元的输出为 h，Dropout 后的输出可以表示为：

$$h' = h \cdot \frac{\text{mask}}{1-p}$$

其中，mask 是一个与 h 形状相同的二进制掩码，表示哪些神经元被保留（1）或丢弃（0）。在实现过程中，我们可以使用 PyTorch 提供的 `nn.Dropout` ，利用`self.dropout = nn.Dropout(p)`启用 Dropout 层，并在`forward` 中用`self.dropout(x)`将其应用于需要进行Dropout 的层之间。

3.3 Task3: 调整参数

在这一部分实验中，我们将通过调整LeNet网络的超参数来进一步提升模型的性能。具体调整的参数包括学习率、批量大小、正则化强度和Dropout概率。

3.4 Task4：实现自己的网络

这一部分要求我们借助参考实现一种现代卷积神经网络。在本实验中，我们选择了DenseNet结构，并通过比较LeNet和DenseNet的性能和训练时间差异等做出结论。

4 实验

4.1 Task1：绘制损失函数曲线

这部分使用给定的Net结构和优化器，训练LeNet 网络，并在每个epoch 结束后记录训练和测试损失值以及准确率。通过调用`draw_loss_and_accuracy_curve`函数绘制损失和准确率曲线。该函数代码如Listing1：

Listing 1: 绘制损失和准确率曲线函数

```python
def draw_loss_and_accuracy_curve(loss, steps, train_acc, test_acc, epochs, path):
    fig, ax1 = plt.subplots()
    ax1.plot(steps, loss, 'b-', label='Training Loss')
    ax1.set_xlabel('Epochs')
    ax1.set_ylabel('Loss', color='b')
    ax1.tick_params(axis='y', labelcolor='b')
    ax2 = ax1.twinx()
    ax2.plot(steps, train_acc, 'r-', label='Train Accuracy')
    ax2.plot(steps, test_acc, 'g-', label='Test Accuracy')
    ax2.set_ylabel('Accuracy', color='r')
    ax2.tick_params(axis='y', labelcolor='r')
    lines1, labels1 = ax1.get_legend_handles_labels()
    lines2, labels2 = ax2.get_legend_handles_labels()
    ax1.legend(lines1 + lines2, labels1 + labels2, loc='upper right')
    plt.title(f'Training Loss and Accuracy Curve over {epochs} Epochs')
    fig.tight_layout()
    plt.savefig(f"{path}/loss_and_accuracy_curve.png")
    plt.close()
```

通过实验，我们得到了训练和测试损失曲线以及准确率曲线，如图1所示：

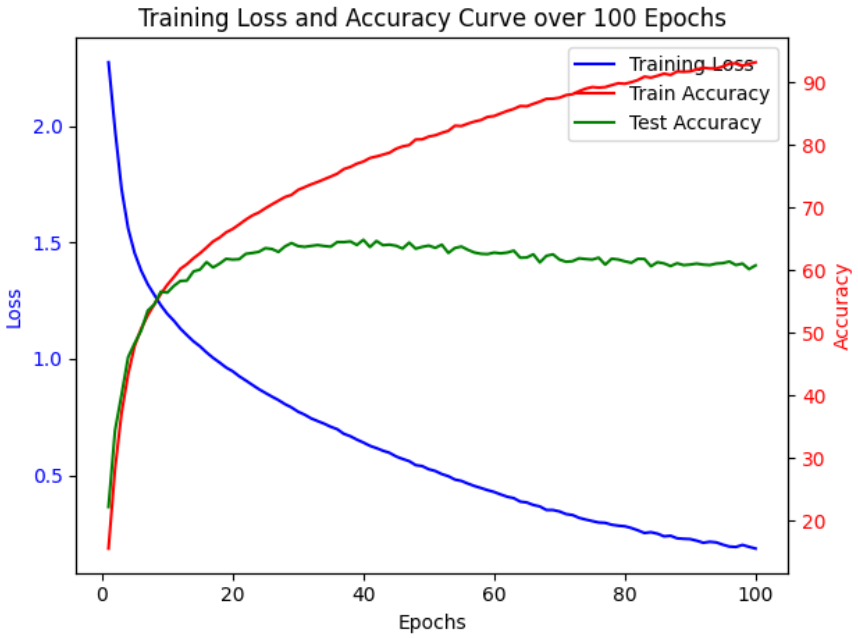


图 1: 损失和准确率随训练轮数的变化

从图中可以看出，随着训练的进行，训练损失逐渐下降，测试准确率逐渐上升，但在30个epoch后，测试准确率开始出现轻微震荡和下降，同时训练准确率持续上升，且差距逐渐增大。说明模型可能开始过拟合训练数据。

虽然训练集损失是衡量模型在训练数据上拟合程度的指标，但过低的训练损失并不一定意味着模型在测试集上表现良好。从图中可以看出，尽管训练损失持续下降，但测试准确率在达到一定水平后开始

波动，甚至略有下降，这表明模型在训练数据上过度拟合，无法很好地泛化到未见过的数据。因此，单纯依赖训练损失来评估模型性能是不够的，必须结合测试集的表现来全面评估模型的泛化能力。

4.2 Task2：加入正则化

在这一部分实验中，我们在LeNet网络中引入了L2正则化和Dropout技术，以探究其对模型性能的影响。

具体实现如下：

1. **L2正则化**：在优化器中添加`weight_decay`参数来实现L2正则化。代码示例如下：

```
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9,
    weight_decay=0.0001)
```

2. **Dropout**：在LeNet的`__init__`方法中添加Dropout层，并在`forward`方法中应用。代码示例如Listing2：

Listing 2: 修改后的LeNet网络结构

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.dropout = nn.Dropout(p=0.5)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)
    def forward(self, x):
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)
        x = x.view(x.size()[0], -1)
        x = F.relu(self.fc1(x))
        x = self.dropout(x)
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

其他参数不变，训练100个epoch，得到的损失和准确率曲线如图2所示：

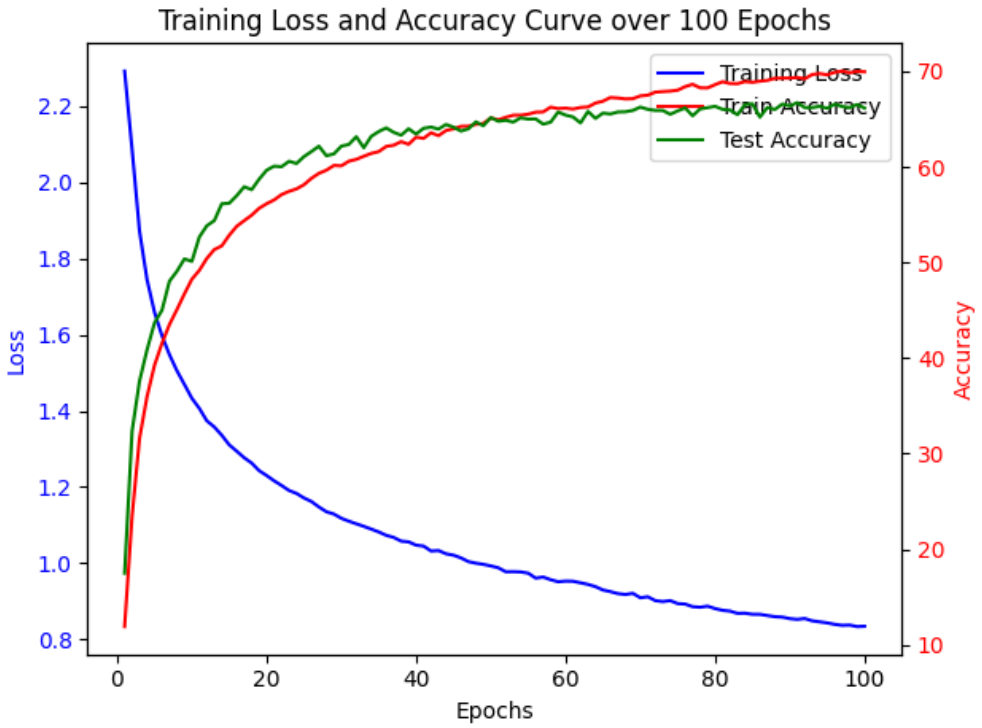


图 2: Dropout+L2正则化结果

从图中可以看出，引入正则化技术后，训练损失曲线相比未正则化的情况有所上升，表明模型在训练数据上的拟合程度有所降低。然而，测试准确率曲线表现出更稳定的上升趋势，且在训练后期没有出现明显的过拟合现象，测试准确率达到了65%左右。

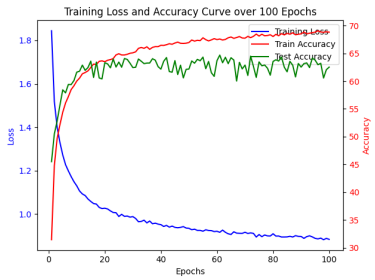
通过对比引入正则化前后的结果，可以明显看出，正则化技术有效地抑制了过拟合现象，提高了模型在测试集上的泛化能力。具体来说，L2正则化通过限制权重的大小，防止模型过度依赖某些特征，而Dropout通过随机丢弃神经元，增强了模型的鲁棒性。综上所述，引入正则化技术显著提升了模型的性能，使其在测试集上表现更加稳定和优越。

同时在训练中观察到加入正则化后，训练时间有所增加。这是因为正则化技术在每次参数更新时引入了额外的计算开销。但这一增加的训练时间在GPU加速下并不显著（均为2s左右一轮），且相对于提升的模型性能和泛化能力，这种时间开销是值得的。

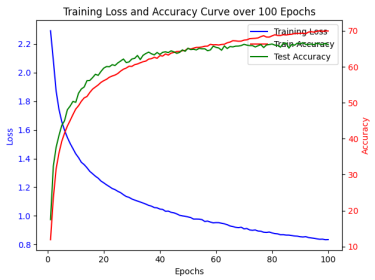
4.3 Task3: 调整参数

在这一部分实验中，我们对LeNet网络的超参数进行了调整，以进一步提升模型的性能。具体调整的参数如下：

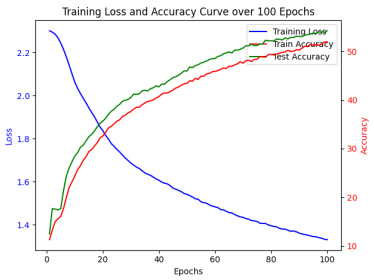
1. **学习率**：学习率 lr 将在 0.0001、0.001 和 0.01 之间进行调整。结果如图3所示：



$lr = 0.01$



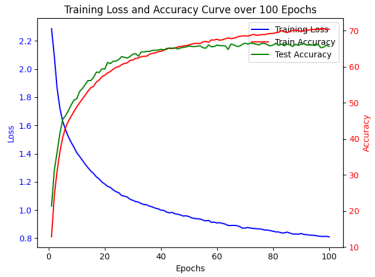
$lr = 0.001$



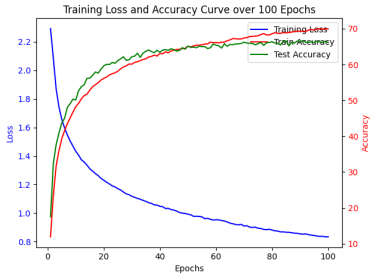
$lr = 0.0001$

图 3: 不同学习率下的训练结果

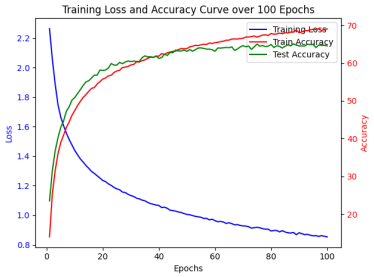
2. **批量大小**：批量大小 batch size 将在 16、64 和 256 之间进行调整。结果如图4所示：



batch size = 16



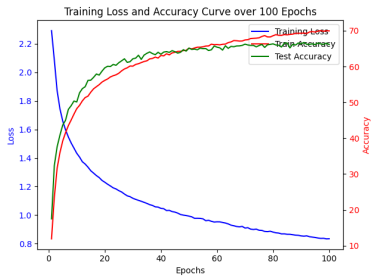
batch size = 64



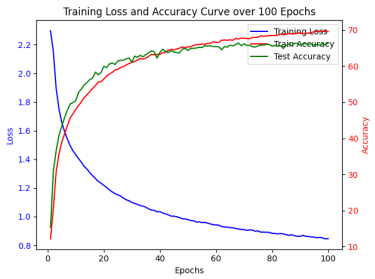
batch size = 256

图 4: 不同批量大小下的训练结果

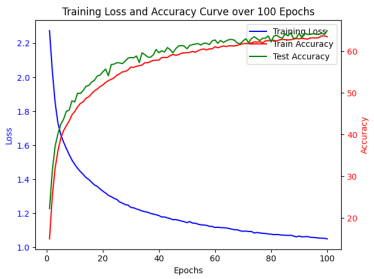
3. **正则化强度**：L2 正则化强度 weight decay 将在 0.0001、0.001 和 0.01 之间进行调整。结果如图5所示：



weight decay = 0.0001



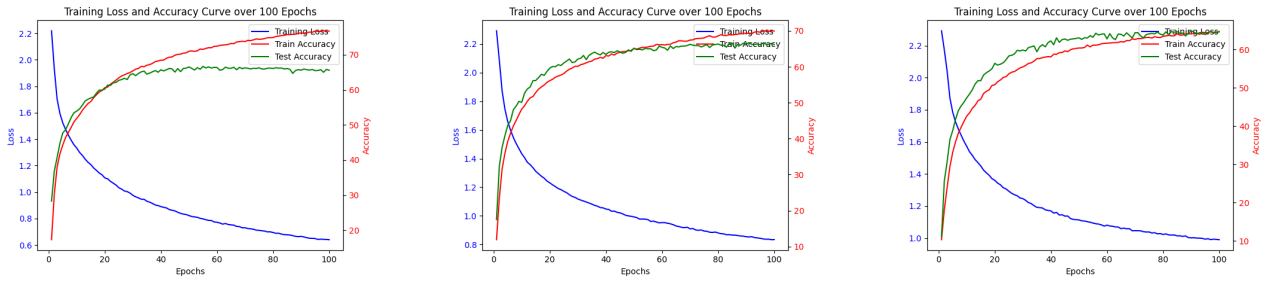
weight decay = 0.001



weight decay = 0.01

图 5: 不同正则化强度下的训练结果

4. **Dropout 概率**：Dropout 的丢弃概率 p 将在 0.3、0.5 和 0.7 之间进行调整。结果如图6所示：



$p = 0.3$ $p = 0.5$ $p = 0.7$

图 6: 不同Dropout概率下的训练结果

4.3.1 结果分析

通过对比不同超参数设置下的训练结果，我们可以得出以下结论：

1. **学习率**：较高的学习率（如0.01）可能导致训练过程不稳定，损失曲线震荡较大甚至损失爆炸，难以收敛；而较低的学习率（如0.0001）则使得训练过程过于缓慢，难以在有限的epoch内达到较好的性能。中等学习率（如0.001）通常能够在稳定性和收敛速度之间取得良好平衡。可以考虑使用梯度下降的学习率来适应不同阶段的训练。

2. **批量大小**：较小的批量大小（如16）虽然可以提供更频繁的参数更新，在训练初期表现良好，但导致训练过程不稳定，损失曲线波动较大；而较大的批量大小（如256）则可能导致模型陷入局部最优，且训练初期收敛慢，但在较大的epoch后表现较好。中等批量大小（如64）通常能够在稳定性和计算效率之间取得良好平衡。同时注意到，较大的批量大小会在一定程度上减少训练时间。且该参数对大epoch训练结果影响不大。

3. **正则化强度**：较强的正则化（如0.01）可能导致模型欠拟合，训练损失较高，测试准确率较低；当正则化强度小于一定值（0.001）时，继续降低正则化强度对训练过程的影响较小，但在随着正则化强度的进一步降低（0.0001）时，在大epoch的情况下过拟合现象会变得更明显。

4. **Dropout概率**：较高的Dropout概率（如0.7）可能导致模型信息丢失过多，训练损失较高，测试准确率较低；而较低的Dropout概率（如0.3）则可能无法有效防止过拟合，导致测试准确率波动较大。一般来说，需要根据数据集的情况来调整Dropout概率，数据集越复杂，Dropout就可以越小，以在防止过拟合和保持信息完整性之间取得平衡。

4.4 Task4：实现自己的网络

本任务参考了DenseNet网络结构，并对其进行了适配以适用于CIFAR-10数据集。DenseNet通过引入密集连接（dense connections），显著提升了信息流动和梯度传播效率，从而提高了模型的性能。标准结构如图7所示：

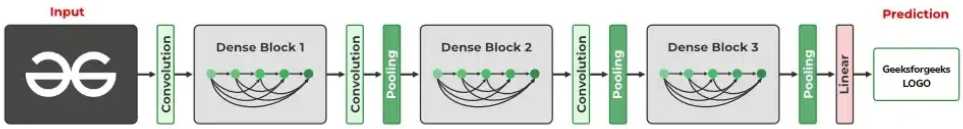


图 7: DenseNet Architecture

4.4.1 网络结构设计

DenseNet的核心思想是通过密集连接将每一层的输出与后续所有层的输入进行拼接，从而实现特征的复用和信息的高效传递。具体结构设计如下：

1. Bottleneck Layer:

Listing 3: Bottleneck Layer实现

```
    class Bottleneck(nn.Module):
        def __init__(self, in_channels, growth_rate):
            super().__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 = F.relu(self.bn1(x))
            out = F.relu(self.bn2(self.conv1(out)))
            out = self.conv2(out)
            return torch.cat([x, out], 1)
```

- 结构如下：
  - 1×1 Conv：将输入通道压缩到 4 × growth_rate（称为 bottleneck width）。
  - 3×3 Conv：输出 growth_rate 个新特征图。
- 输出：[x, out] 拼接，通道数增加 growth_rate。

2. Dense Block:

```
    class DenseBlock(nn.Module):
        def __init__(self, n_layers, in_channels, growth_rate):
            super().__init__()
            self.layers = nn.ModuleList()
            current_channels = in_channels
            for i in range(n_layers):
                self.layers.append(Bottleneck(current_channels, growth_rate))
                current_channels += growth_rate
        def forward(self, x):
            for layer in self.layers:
                x = layer(x)
            return x
```

- 由n_layers个 Bottleneck Layer 组成，每个层的输入是前面所有层的输出拼接而成。
- 每一层的输入 = 原始输入 + 所有前面层的输出（自动通过 torch.cat 累积）。
- 输出通道数 = in_channels + n_layers × growth_rate。

3. Transition Layer:

```
        class Transition(nn.Module):
        def __init__(self, in_channels, out_channels):
            super().__init__()
            self.bn = nn.BatchNorm2d(in_channels)
            self.conv = nn.Conv2d(in_channels, out_channels,
                kernel_size=1, bias=False)
            self.pool = nn.AvgPool2d(2)
        def forward(self, x):
            out = F.relu(self.bn(x))
            out = self.conv(out)
            return self.pool(out)
```

- 作用：
  - 降维：通过 1×1 Conv 将通道数减半（out_channels = in_channels // 2）。
  - 下采样：AvgPool2d(2) 将特征图尺寸减半。用于减少计算量和内存占用。

4. DenseNet 主体：

```
    class DenseNet_CIFAR(nn.Module):
        def __init__(self, block_config=(4, 4, 4, 4), growth_rate=16,
            num_classes=10):
            super().__init__()
            num_channels = 2 * growth_rate
            self.conv0 = nn.Conv2d(3, num_channels, kernel_size=3, padding=1,
                bias=False)
            self.dense_blocks = nn.ModuleList()
            self.transitions = nn.ModuleList()
```

```
            in_channels = num_channels
            for i, num_layers in enumerate(block_config):
                block = DenseBlock(num_layers, in_channels, growth_rate)
                self.dense_blocks.append(block)
                in_channels += num_layers * growth_rate
                if i != len(block_config) - 1:
                    out_channels = in_channels // 2
                    self.transitions.append(Transition(in_channels,
                        out_channels))
                    in_channels = out_channels
            self.final_bn = nn.BatchNorm2d(in_channels)
            self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
            self.classifier = nn.Sequential(
                nn.Dropout(0.2),
                nn.Linear(in_channels, num_classes)
            )
        def forward(self, x):
            out = self.conv0(x)
            for i, block in enumerate(self.dense_blocks):
                out = block(out)
                if i < len(self.transitions):
                    out = self.transitions[i](out)
            out = F.relu(self.final_bn(out))
            out = self.avgpool(out)
            out = torch.flatten(out, 1)
            out = self.classifier(out)
            return out
```

- 结构：
  - 卷积模块：
    - 初始卷积层：3×3 Conv，输出通道数为 2 × growth_rate。
    - 4 个 Dense Block，每个包含 4 个 Bottleneck Layer。
    - 3 个 Transition Layer，用于降维和下采样。
  - 分类模块：
    - 全局平均池化：AdaptiveAvgPool2d((1, 1))，将特征图尺寸变为 1×1。
    - 全连接层：包含 Dropout（丢弃概率为0.2）和线性层，输出类别数为10。
- 参数：
  - block_config=(4, 4, 4, 4)：每个 Dense Block 包含 4 层 Bottleneck Layer。
  - growth_rate=16：每层输出16个新特征图。
  - num_classes=10：CIFAR-10数据集的类别数。

4.4.2 训练设置

初始化代码和数据处理部分如Listing4：

Listing 4: DenseNet_CIFAR训练设置

```
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    print(f"Using device: {device}")
    if device.type == "cuda":
        print(f"GPU: {torch.cuda.get_device_name(0)}")
    mean = (0.4914, 0.4822, 0.4465)
    std = (0.2023, 0.1994, 0.2010)
    transform_train = transforms.Compose([
        transforms.RandomCrop(32, padding=4),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Normalize(mean, std),
        transforms.RandomErasing(p=0.3, scale=(0.02, 0.2))
    ])
    transform_test = transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize(mean, std),
    ])
    trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
        download=True, transform=transform_train)
    trainloader = DataLoader(trainset, batch_size=64, shuffle=True, num_workers=4,
        pin_memory=True)
    testset = torchvision.datasets.CIFAR10(root='./data', train=False,
        download=True, transform=transform_test)
    testloader = DataLoader(testset, batch_size=64, shuffle=False, num_workers=4,
        pin_memory=True)
```

以及优化器和损失函数设定如Listing5：

Listing 5: DenseNet_CIFAR优化器和损失函数设定

```
    model = DenseNet_CIFAR(block_config=(4, 4, 4, 4), growth_rate=16,
        num_classes=10).to(device)
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.SGD(model.parameters(), lr=0.1, momentum=0.9,
        weight_decay=1e-4)
    scheduler = optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=100,
        eta_min=0)
```

解释如下：

1. 使用GPU加速训练（如可用）。
2. 参考torchvision 推荐的均值和标准差对CIFAR-10数据集进行归一化处理。
3. 数据增强包括随机裁剪、水平翻转和随机擦除，以增强模型的泛化能力，减少过拟合。
4. 优化器使用SGD，学习率初始为0.1。
5. 使用余弦退火学习率调度器动态调整学习率，有助于模型收敛。

4.4.3 模型对比

结构上，我们采用了DenseNet的设计理念，通过密集连接提升信息流动和梯度传播效率。与前文使用的LeNet相比，结构差异如表1所示：

	DenseNet_CIFAR	LeNet
层数	36 层	5 层
连接方式	每层接收前面所有层的输出	逐层传递，无跨层连接
特征复用	通过拼接前面所有层的输出	每层仅使用前一层的输出
下采样	平均池化	最大池化
归一化	BatchNorm	无
正则化	Dropout(0.2) + weight_decay = 1e-4	Dropout(0.5) + weight_decay = 1e-4

表 1: DenseNet_CIFAR与LeNet结构对比

4.4.4 训练过程与实验结果

学习率等参数如4.4.2所示，训练100个epoch，得到的损失和准确率曲线如图8所示：

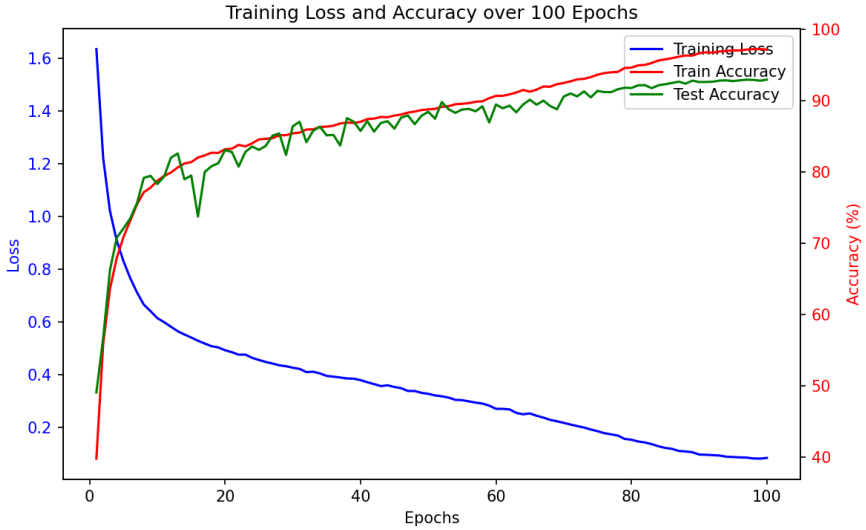


图 8: DenseNet训练结果

结果及结论如下：

1. **性能提升**：DenseNet_CIFAR在CIFAR-10数据集上表现出显著的性能提升，测试准确率达到了92.93%，远高于LeNet的65%左右。

2. **训练稳定性**：在余弦退火学习率调度器动态调整学习率的帮助下，DenseNet的训练过程更加稳定，损失曲线平滑下降，测试准确率持续上升，未见明显过拟合现象。

3. **训练时间**：DenseNet结构复杂，训练时间增加到了约35s/epoch，较带Dropout和L2的LeNet有明显增加(2s/epoch)。但考虑前者数据集并未加载到显存中，实际性能差距并没有非常大。

4. **特征复用**：DenseNet通过密集连接实现了高效的特征复用，提升了信息流动和梯度传播效率。

5. **正则化效果**：结合Dropout和L2正则化，有效抑制了过拟合，提升了模型的泛化能力。

查阅原始论文知，类似层数的DenseNet(L=40, k=12)在300轮训练后准确率接近94.76%（使用增强的CIFAR-10数据集）。由此看来，本次实验实现的DenseNet_CIFAR在100轮训练后达到92.93%的准确率，表现出色，验证了DenseNet结构在图像分类任务中的有效性。

5 结果与分析

通过本次实验，我们深入探讨了卷积神经网络在CIFAR-10数据集上的训练过程及其性能表现。以下是对各个任务的结果与分析：

1. **损失函数曲线绘制**：通过绘制训练和测试损失曲线，我们观察到模型在训练过程中逐渐收敛。然而，测试准确率在训练后期出现波动且与训练准确率相去甚远，模型存在较为严重的过拟合现象。

2. **正则化技术应用**：引入L2正则化和Dropout后，模型的训练损失有所上升，但测试准确率有所提升，且过拟合现象得到有效抑制。这表明正则化技术在提升模型泛化能力方面发挥了重要作用。

3. **超参数调整**：通过调整学习率、批量大小、正则化强度和Dropout概率，我们发现这些超参数对模型性能有显著影响。总的来说，我们需要根据数据集、模型结构、训练轮数和计算资源等因素综合考虑，选择合适的超参数组合以获得最佳性能。

4. **DenseNet实现与对比**：基于DenseNet结构的模型在CIFAR-10数据集上表现出色，测试准确率达到了92.93%。与LeNet相比，DenseNet通过密集连接实现了高效的特征复用，显著提升了模型性能。

5. **训练时间分析**：更复杂的模型在训练过程中需要的时间往往更长，但通过合理的结构设计和优化策略，可以在可接受的时间内获得优异的性能。同时我们注意到模型准确率与训练时间并非简单线性关系，更多的训练时间并不总是带来更高的准确率，关键在于模型结构和训练策略的优化。

总体而言，本次实验验证了卷积神经网络在图像分类任务中的有效性，并展示了正则化技术和现代网络结构在提升模型性能方面的重要作用。同时我们发现，在小数据场景下(如CIFAR-10)，合理设计网络结构和训练策略尤为关键，绝不能盲目追求更深更复杂的模型。如何在计算资源、模型复杂度、训练时间、泛化效果间找到平衡点，是深度学习中不可忽视的问题。

5.1 实验中的问题

最开始时，我们希望以ResNet为蓝本构建CNN，但测试过程中，在训练超过15轮后，损失开始增大，准确率下降且不可逆转。调参后未能解决，因而转向更复杂、训练时间更长，但已验证过在该电脑上可行的类DenseNet架构。

另外，在尝试将模型转移到GPU上训练时，发现了当batch_size过大时(如1024)，CPU会出现严重的性能瓶颈，导致训练速度极慢，甚至无法进行训练。后续在调低batch_size以及在最开始的LeNet中将所有数据集全部写入显存后恢复正常。

然而，在个人模型的训练中，由于需要对图像进行数据增强，最后放弃了将数据集全部写入显存的做法，转而使用DataLoader按batch加载数据。这无疑会带来训练时间的延长。

6 环境和其他模块

本实验在以下环境中进行：

- 操作系统：Windows 11 25H2 (版本 26200.7019)
- Python版本：CPython 3.14.0 with freethreaded
- 编程工具：PyCharm 25.2.4 (Pro Edition)
- 环境管理：uv 0.9.7
- 硬件配置：
  - GPU：NVIDIA GeForce RTX 5060 Laptop GPU
  - CUDA版本：13.0.2
  - 显存：8 GB
  - CPU：AMD Ryzen AI HX 370
  - 内存：32 GB RAM