目录

CIFAR10 图像分类

实验任务是使用卷积网络实现 CIFAR10 数据集图像分类。本次实验基于 torch 库实现，所选数据集为 CIFAR10 官方数据集，源码已在云平台提交，所有代码已同步上传至 github。

一、模型搭建

1.1、模型搭建

（1）模型搭建

本次实验采用的是卷积神经网络，而卷积操作是在整张图像上进行，所以我们的输入应是原始数据的大小，即 3*32*32，使用卷积后算子后，输入数据会映射到高维语义空间中，表现为通道数量（即特征图的数量）增多，而尺寸减小（即视野变大）。

所以我们定义了从输入通道到高维输出通道的卷积操作：

```
class SimpleCNN(nn.Module):
    def __init__(self):
        super(SimpleCNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 32, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        self.conv3 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
        self.pool = nn.MaxPool2d(2, 2)
```

```
    def forward(self, x):
        x = self.pool(self.relu(self.conv1(x)))
        x = self.pool(self.relu(self.conv2(x)))
        x = self.pool(self.relu(self.conv3(x)))
```

接着我们需要将卷积后的特征图展平，作为全连接层的输入，全连接层经过降维后输出 10 通道作为分类输出：

```
class SimpleCNN(nn.Module):
    def __init__(self):
        super(SimpleCNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 32, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        self.conv3 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
        self.pool = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(128 * 4 * 4, 256)
        self.fc2 = nn.Linear(256, 128)
        self.fc3 = nn.Linear(128, 10)
        self.relu = nn.ReLU()
```

```
    def forward(self, x):
        x = self.pool(self.relu(self.conv1(x)))
        x = self.pool(self.relu(self.conv2(x)))
        x = self.pool(self.relu(self.conv3(x)))
        x = x.view(-1, 128 * 4 * 4)
        x = self.relu(self.fc1(x))
        x = self.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

这样就定义了 CNN 模型的基本框架，之后的 dropout、batch normalization 等正则化技术都是基于该网络架构进行修改。

（2）数据处理

首先引入数据，使用 torch 的 CIFAR10 数据集:

```
train_dataset = torchvision.datasets.CIFAR10(root='Datasets', train=True,
                                             download=True, transform=transform)
test_dataset = torchvision.datasets.CIFAR10(root='Datasets', train=False,
                                            download=True, transform=transform_test)
```

其中 transform 参数是我们对数据预处理的操作，在这个操作中我们可以使用数据增强技术引入正则，从而增强模型的泛化能力。

然后进行数据集划分，将数据集按照 4:1:1 划分出训练集、验证集和测试集。

```
# 划分数据集
train_size = int(0.8 * len(train_dataset)) 整个训练集为50000张，分出10000张作为验证集
valid_size = len(train_dataset) - train_size
train_subset, valid_subset = random_split(train_dataset, [train_size, valid_size])

train_loader = DataLoader(dataset=train_subset, batch_size=batch_size, shuffle=True)
valid_loader = DataLoader(dataset=valid_subset, batch_size=batch_size, shuffle=False)
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False) 10000张
```

最后直接通过 torch 的数据迭代器将数据做成可迭代的批次对象，batch_size 为超参数：

```
train_loader = DataLoader(dataset=train_subset, batch_size=batch_size, shuffle=True)
valid_loader = DataLoader(dataset=valid_subset, batch_size=batch_size, shuffle=False)
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)
```

（3）训练与验证

首先定义好模型、优化器和损失函数，以 Adam 优化器与交叉熵损失为例：

```
model = SimpleCNN().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=lr)
```

之后定义训练函数，统计 loss 值和 acc 值，如图所示：

```
def train(num_epochs):
    for epoch in range(num_epochs):
        model.train()  # 切换到训练模式
        train_loss = 0.0 # 每一轮都初始化损失
        train_acc = 0.0

        for images, labels in train_loader:
            images, labels = images.to(device), labels.to(device)
            optimizer.zero_grad() # 每一次迭代都将模型参数的梯度清零
            outputs = model(images) # 穿入模型
            loss = criterion(outputs, labels) # 计算损失
            loss.backward() # 反向传播
            optimizer.step() # 更新

            train_loss += loss.item() * images.size(0) # 累计该批次损失
            _, predictions = torch.max(outputs.data, 1)
            correct_counts = predictions.eq(labels.data.view_as(predictions))
            acc = torch.mean(correct_counts.type(torch.FloatTensor)) # 统计正确率
            train_acc += acc.item() * images.size(0) # 累计该批次准确率

        valid_loss = 0.0
        valid_acc = 0.0
        with torch.no_grad():
            model.eval()  # 切换到验证模式
            for images, labels in valid_loader:
                images, labels = images.to(device), labels.to(device)
                outputs = model(images)
                loss = criterion(outputs, labels)
                valid_loss += loss.item() * images.size(0)

                _, predictions = torch.max(outputs.data, 1)
                correct_counts = predictions.eq(labels.data.view_as(predictions))
                acc = torch.mean(correct_counts.type(torch.FloatTensor))
                valid_acc += acc.item() * images.size(0)

        avg_train_loss = train_loss / train_data_size  # 得到所有批次的平均损失
        avg_train_acc = train_acc / train_data_size # 得到所有批次的平均准确率
        avg_valid_loss = valid_loss / valid_data_size
```

（4）测试

针对测试，我们将测试集输入进训练好的 model，得到测试集的 outputs，同样计算每个批次的准确度，最后求取平均作为模型的评估指标。

```python
def test():
    model.eval()  # 切换到评估模式
    correct = 0
    total = 0
    with torch.no_grad():
        for images, labels in test_loader:
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
    print(f"Test Accuracy: {100 * correct / total:.2f}%")
```

1.2、网络结构优化

上一节我们理解了 CNN 模型的模型结构以及训练、测试流程，下面我们可以采用不同的网络结构提升模型性能。

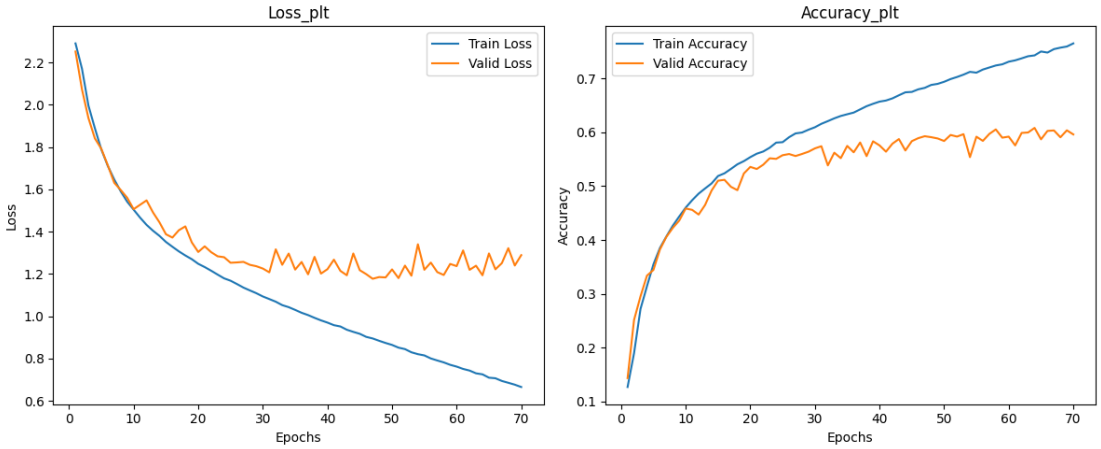
1.2.1、LeNet

首先我们采用最简单的 LeNet 卷积网络。LeNet 采用了卷积层、平均池化层与全连接层，并使用 sigmod 激活函数，是最早的卷积神经网络。为了适配我们的数据输入维度，我们将 LeNet 模型定义如下：

```python
# 定义 LeNet 模型
class LeNet(nn.Module):
    def __init__(self):
        super(LeNet, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, kernel_size=5, padding=2)  # 第一层卷积
        self.pool = nn.AvgPool2d(2, 2)  # 平均池化层
        self.conv2 = nn.Conv2d(6, 16, kernel_size=5)  # 第二层卷积
        self.fc1 = nn.Linear(16 * 6 * 6, 120)  # 第一个全连接层
        self.fc2 = nn.Linear(120, 84)  # 第二个全连接层
        self.fc3 = nn.Linear(84, 10)  # 输出层，对应10个类别

    def forward(self, x):
        x = self.pool(torch.relu(self.conv1(x)))
        x = self.pool(torch.relu(self.conv2(x)))
        x = x.view(-1, 16 * 6 * 6)  # 展平处理
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

LeNet 模型的训练结果：

```
Epoch [22/70] Training: Loss: 1.2156, Accuracy: 56.44%, Validation: Loss: 1.3033, Accuracy: 53.99%
Epoch [23/70] Training: Loss: 1.1973, Accuracy: 57.14%, Validation: Loss: 1.2838, Accuracy: 55.18%
...
Epoch [68/70] Training: Loss: 0.6858, Accuracy: 75.72%, Validation: Loss: 1.3222, Accuracy: 59.08%
Epoch [69/70] Training: Loss: 0.6769, Accuracy: 75.93%, Validation: Loss: 1.2403, Accuracy: 60.37%
Epoch [70/70] Training: Loss: 0.6658, Accuracy: 76.51%, Validation: Loss: 1.2893, Accuracy: 59.61%
Test Accuracy: 59.55%
```



结论：

LeNet 作为最简单的卷积网络，训练效果不是特别理想。可见模型在后半程已经发生了严重过拟合，并且测试准确度也仅有 60%（但已经超越了 A1 中的所有 MLP 网络）。

LeNet 由于使用了卷积网络，引入了参数共享与空间不变性，使得网络参数大大减少。同时 LeNet 引入了池化层，进一步降低了计算量，并提供了一定程度的形变不变性，使模型对小的位移、旋转和缩放变得不敏感。

1.2.2、AlexNet

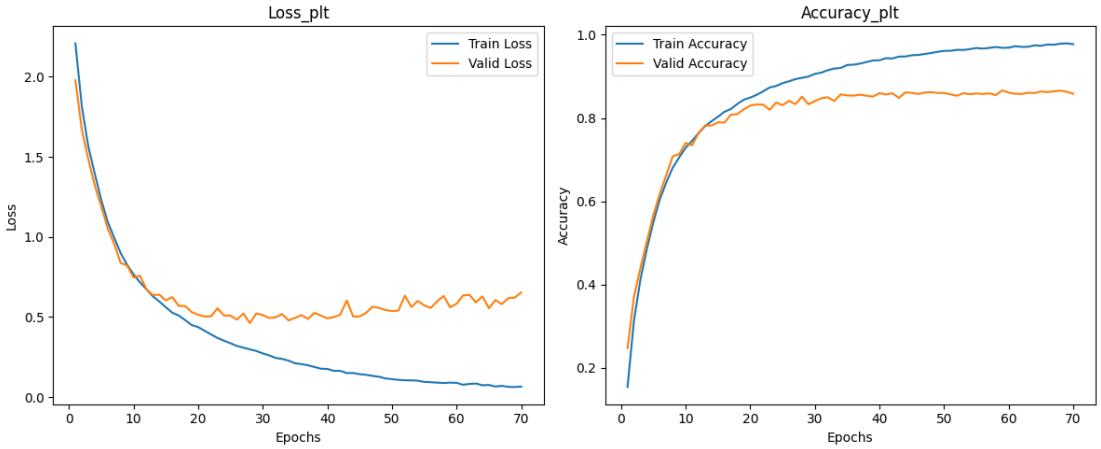
为了进一步加深网络层数，我们使用改进后的 AlexNet。AlexNet 是 2012 年 ImageNet 大赛的冠军，它首次提出了 ReLU 激活、Dropout 正则化、多 GPU 训练和数据增强等一系列创新方法，使得网络的训练效率和模型能力大幅增强。

原始的 AlexNet 采用了 5 层卷积层+3 层全连接层的结构，并使用了 11x11 的大卷积核。为了与我们的数据输入维度匹配，我们调整了通道数与卷积核大小。网络结构如图：

```python
class AlexNet(nn.Module):
    def __init__(self):
        super(AlexNet, self).__init__()
        self.features = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1),
            nn.ReLU(inplace=True),  # Alex采用的新型的relu激活函数
            nn.MaxPool2d(kernel_size=2, stride=2),  # 最大池化
            nn.Conv2d(64, 192, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Conv2d(192, 384, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(384, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(256, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2)
        )
        self.classifier = nn.Sequential(
            nn.Dropout(0.5), # Alex采用了Dropout
            nn.Linear(256 * 4 * 4, 4096),
            nn.ReLU(inplace=True),
            nn.Dropout(0.5),
            nn.Linear(4096, 2048),
            nn.ReLU(inplace=True),
            nn.Linear(2048, 10)
        )
    def forward(self, x):
        x = self.features(x)
        x = x.view(x.size(0), 256 * 4 * 4)
        x = self.classifier(x)
        return x
```

AlexNet 的训练结果：

```
Epoch [22/70] Training: Loss: 0.3914, Accuracy: 86.41%, Validation: Loss: 0.5044, Accuracy: 83.21%
Epoch [23/70] Training: Loss: 0.3692, Accuracy: 87.33%, Validation: Loss: 0.5549, Accuracy: 81.97%
...
Epoch [68/70] Training: Loss: 0.0642, Accuracy: 97.85%, Validation: Loss: 0.6181, Accuracy: 86.56%
Epoch [69/70] Training: Loss: 0.0631, Accuracy: 97.91%, Validation: Loss: 0.6215, Accuracy: 86.29%
Epoch [70/70] Training: Loss: 0.0658, Accuracy: 97.73%, Validation: Loss: 0.6534, Accuracy: 85.82%
Test Accuracy: 86.86%
```



结论：

由于网络层数加深以及采用了 dropout 正则，模型的性能得到非常大的提升，测试准确度达到 86.86%。可见增大模型层数的确有利于模型性能的提升。然而受制于 AlexNet 的层数过多，尤其是在全连接的分类层参数达到 4096*2048，很容易产生过拟合。所以后续我们需要降低全连接层的参数量，减轻过拟合的风险。

1.2.3、ResNet

Resnet 通过构造残差块，使得网络层间形成 shortcut connection，可以极大缓解由于层数增多导致的梯度消失或梯度爆炸问题。

典型的残差块构造是：

第一个卷积层 + 批归一化 + 激活函数：用于提取特征。

第二个卷积层 + 批归一化：进一步学习特征。

残差连接：将输入 x 和经过两个卷积层处理后的输出 F(x) 相加。

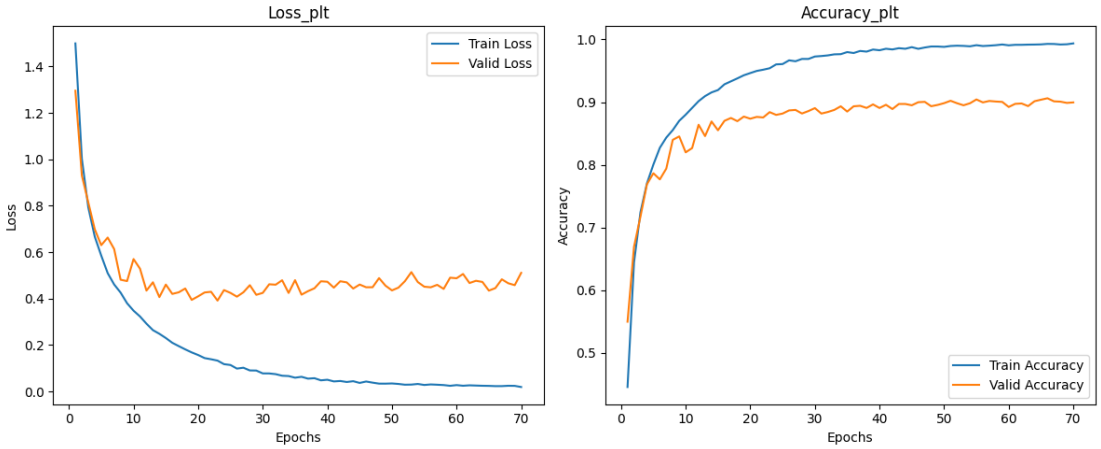
我们基于 AlexNet 构造了一个 15 层的残差网络，包括 5 个残差块和 3 个池化层，同时去除了 AlexNet 的 dropout，并将全连接分类层压缩为两层：

```
# 残差块
class ResidualBlock(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(ResidualBlock, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3, padding=1)

        # 用于调整输入和输出的维度
        self.shortcut = nn.Sequential()
        if in_channels != out_channels:
            self.shortcut = nn.Conv2d(in_channels, out_channels, kernel_size=1)

    def forward(self, x):
        out = self.conv1(x)
        out = self.relu(out)
        out = self.conv2(out)
        return self.relu(out + self.shortcut(x))  # 残差连接
```

```python
# 带有残差连接的 AlexNet 模型
class AlexNetWithResiduals(nn.Module):
    def __init__(self):
        super(AlexNetWithResiduals, self).__init__()
        self.features = nn.Sequential(
            ResidualBlock(3, 64),
            nn.MaxPool2d(kernel_size=2, stride=2),
            ResidualBlock(64, 192),
            nn.MaxPool2d(kernel_size=2, stride=2),
            ResidualBlock(192, 384),
            ResidualBlock(384, 256),
            ResidualBlock(256, 256),
            nn.MaxPool2d(kernel_size=2, stride=2)
        )
        self.classifier = nn.Sequential(
            nn.Linear(256 * 4 * 4, 1024),  # 修改全连接层输入尺寸
            nn.ReLU(inplace=True),
            nn.Linear(1024, 10),
        )
```

训练结果：

```
Epoch [22/70] Training: Loss: 0.1389, Accuracy: 95.16%, Validation: Loss: 0.4295, Accuracy: 87.55%
Epoch [23/70] Training: Loss: 0.1331, Accuracy: 95.39%, Validation: Loss: 0.3915, Accuracy: 88.40%
...
Epoch [69/70] Training: Loss: 0.0243, Accuracy: 99.22%, Validation: Loss: 0.4580, Accuracy: 89.87%
Epoch [70/70] Training: Loss: 0.0193, Accuracy: 99.36%, Validation: Loss: 0.5110, Accuracy: 89.95%
Test Accuracy: 90.33%
All Time: 30 分 29 秒
```



结论：

增加了残差层后，测试精度达到了 90.33%，比 8 层的 AlexNet 还多 4 个点，说明残差结构确实有利于模型收敛和加深训练程度。不足是全连接层的参数量仍然很大，没有有效的正则恶化约束，导致后期模型出现了过拟合。

1.3、Baseline 选择

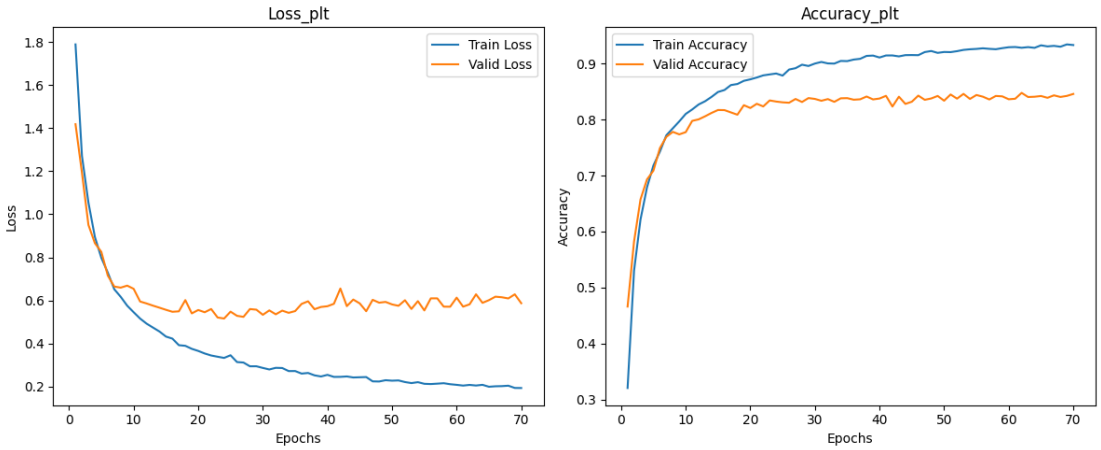
经过上一节网络结构的调优，我们选择 15 层不带任何正则化措施的残差网络作为 Baseline，其中包含了 5 个残差块、3 个池化层以及两层全连接分类网络，并选用如下超参数：

```
-- batch_size      批次大小          128
-- epochs          轮次大小          70
-- lr              学习率            0.001
-- optim           优化器            Adam
-- activation      激活函数          relu
-- dropout         随机丢弃比率      0.0
-- momentum        动量              0
```

基线测试结果：

```
Epoch [22/70] Training: Loss: 0.3449, Accuracy: 87.88%, Validation: Loss: 0.5605, Accuracy: 82.34%
Epoch [23/70] Training: Loss: 0.3391, Accuracy: 88.06%, Validation: Loss: 0.5211, Accuracy: 83.41%
...
Epoch [69/70] Training: Loss: 0.1943, Accuracy: 93.41%, Validation: Loss: 0.6287, Accuracy: 84.22%
Epoch [70/70] Training: Loss: 0.1939, Accuracy: 93.30%, Validation: Loss: 0.5874, Accuracy: 84.57%
Test Accuracy: 84.86%
All Time: 30 分 13 秒
```

Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...



结论：

Baseline 的测试集上的准确度为 84.86%。与上一节的 ResNet 主要差别是缺少了 Dropout 措施，导致过拟合现象突出，同时动量设为 0 导致收敛程度和训练程度表现不佳，模型测试精度降低。

二、正则化

以下基于 Baseline 采取不同正则化策略

2.1、随机丢弃

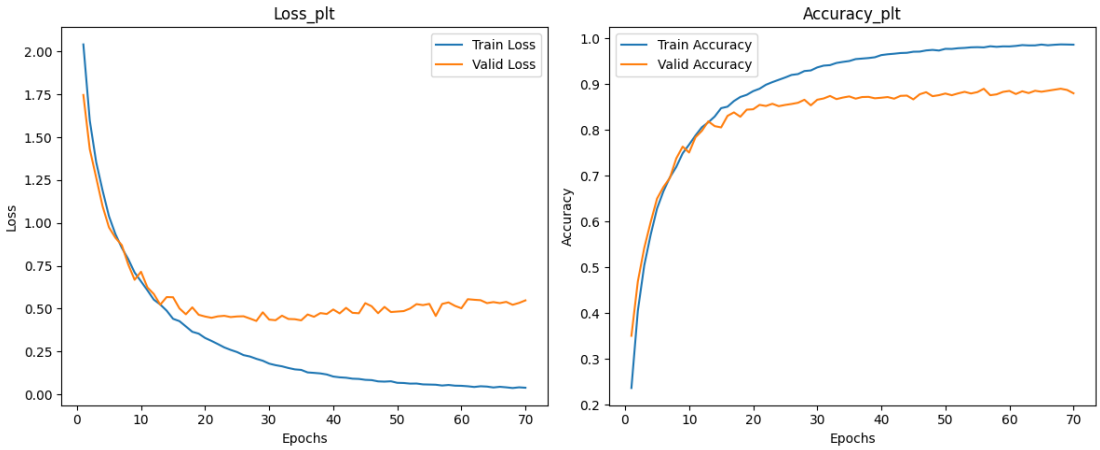
Dropout 是一种常用的正则化策略，可以有效避免过拟合。在 CNN 中，由于卷积层的参数量比较小，dropout 的情形比较少，所以我们将 dropout 放到全连接层的激活函数之后。汲取第一节的教训，配合自适应池化降低全连接的参数量，实现细节如下：

```
# 带有残差连接的 AlexNet 模型
class AlexNetWithResiduals(nn.Module):
    def __init__(self):
        super(AlexNetWithResiduals, self).__init__()
        self.features = nn.Sequential(
            ResidualBlock(3, 64),
            nn.MaxPool2d(kernel_size=2, stride=2),
            ResidualBlock(64, 192),
            nn.MaxPool2d(kernel_size=2, stride=2),
            ResidualBlock(192, 384),
            ResidualBlock(384, 256),
            ResidualBlock(256, 256),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.AdaptiveAvgPool2d((1, 1))  # 使用自适应平均池化,用于通道降维
        )
        self.classifier = nn.Sequential(
            nn.Linear(256 , 64),
            nn.ReLU(inplace=True),
            nn.Dropout(0.5),
            nn.Linear(64, 10)
        )

    def forward(self, x):
        x = self.features(x)
        x = x.view(x.size(0), 256)
        x = self.classifier(x)
        return x
```

训练结果：

```
Epoch [22/70] Training: Loss: 0.2932, Accuracy: 89.89%, Validation: Loss: 0.4551, Accuracy: 85.25%
Epoch [23/70] Training: Loss: 0.2738, Accuracy: 90.46%, Validation: Loss: 0.4582, Accuracy: 85.73%
...
Epoch [69/70] Training: Loss: 0.0410, Accuracy: 98.67%, Validation: Loss: 0.5334, Accuracy: 88.75%
Epoch [70/70] Training: Loss: 0.0386, Accuracy: 98.63%, Validation: Loss: 0.5483, Accuracy: 88.02%
Test Accuracy: 88.57%
All Time: 30 分 0 秒
```

Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...



结论：

在 Dropout 的使用后，可以看到模型过拟合现象减弱，表现为测试精度与验证精度分叉时间变晚，间距减小。Loss 降低到更低值，说明损失函数曲面收敛到更谷底，模型训练的程度加深。测试集的表现也有所提升，上升约 3.5 个点。说明正则化策略能有效避免过拟合，从而使模型的训练程度加深。

2.2、归一化

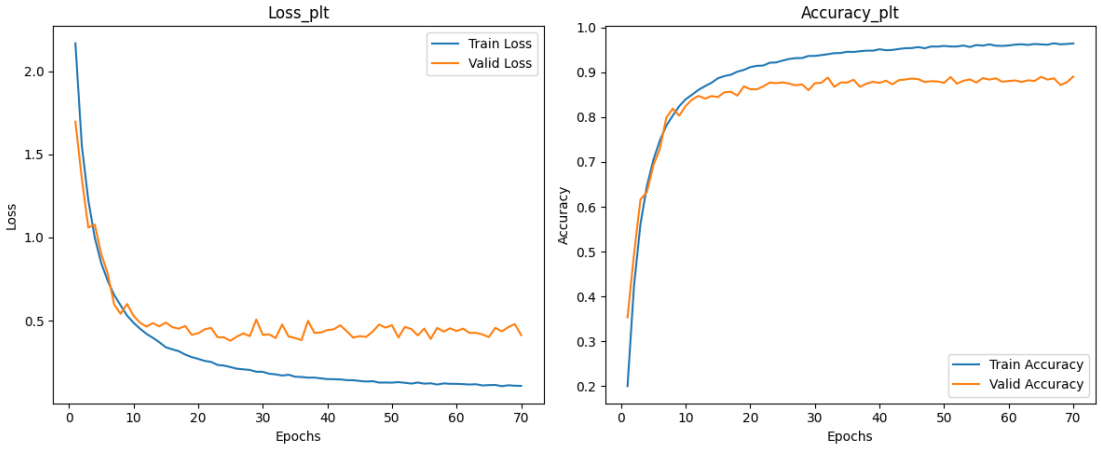
归一化是另一种常用的正则策略，通过在每一批数据中对神经网络的输入或输出进行标准化，使数据具有稳定的分布。我们在 Baseline 加上 Dropout 的基础上，在每个残差块的每个激活函数前引入批归一化：

```
# 残差块
class ResidualBlock(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(ResidualBlock, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3, padding=1)
        self.bn2 = nn.BatchNorm2d(out_channels)  # 添加归一化层

        # 用于调整输入和输出的维度
        self.shortcut = nn.Sequential()
        if in_channels != out_channels:
            self.shortcut = nn.Conv2d(in_channels, out_channels, kernel_size=1)

    def forward(self, x):
        out = self.conv1(x)
        out = self.bn1(out)  # 应用归一化
        out = self.relu(out)
        out = self.conv2(out)
        out = self.bn2(out)  # 应用归一化
        return self.relu(out + self.shortcut(x))  # 残差
```

训练结果：

```
Epoch [22/70] Training: Loss: 0.2520, Accuracy: 91.49%, Validation: Loss: 0.4568, Accuracy: 86.84%
Epoch [23/70] Training: Loss: 0.2337, Accuracy: 92.13%, Validation: Loss: 0.4005, Accuracy: 87.71%
...
Epoch [69/70] Training: Loss: 0.1087, Accuracy: 96.30%, Validation: Loss: 0.4796, Accuracy: 87.76%
Epoch [70/70] Training: Loss: 0.1077, Accuracy: 96.44%, Validation: Loss: 0.4130, Accuracy: 89.05%
Test Accuracy: 89.18%
All Time: 29 分 36 秒
```

Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...



结论：

通过模型曲线，可以看到模型收敛速度加快，有效缓解了梯度弥散问题。同时模型过拟合的现象进一步减弱，因为归一化改变了数据的方差大小和均值，使得新的分布更切合数据的真实分布，保证了模型的泛化能力。同时模型在测试集的准确度相较于只使用 Dropout 的情况也有所提升，提升约 0.51 个点。

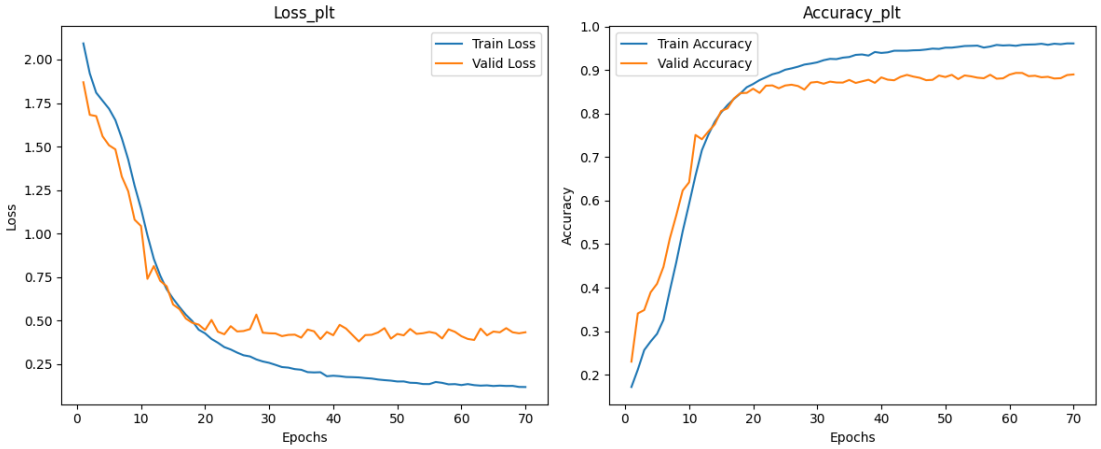
2.3、数据增强

与批归一化等归一化策略通过调整数据分布的手段不同，数据增强直接在原始数据上施加变换，使模型能够学习到更丰富、更多样的特征，从而应对不同场景下的数据变化。数据增强尤其适用于数据量有限的场景，通过增加数据的多样性，降低过拟合风险。在前面正则化的基础上，我们利用 torch 的 transform 技术，对数据采取随机翻转、随机裁切、随机染色、随机翻转裁切、Cutout 的操作：

```python
# 数据增强
transform = transforms.Compose([
    transforms.RandomHorizontalFlip(),
    transforms.RandomCrop(32, padding=4),
    transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2, hue=0.1),
    transforms.RandomRotation(15),
    transforms.RandomResizedCrop(32, scale=(0.8, 1.0)),
    transforms.ToTensor(),
    Cutout(num_holes=1, max_h_size=8, max_w_size=8),  #  Cutout 数据增强
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)),
])
```

训练结果：

```
Epoch [22/70] Training: Loss: 0.3719, Accuracy: 88.35%, Validation: Loss: 0.4354, Accuracy: 86.42%
Epoch [23/70] Training: Loss: 0.3467, Accuracy: 89.05%, Validation: Loss: 0.4206, Accuracy: 86.49%
...
Epoch [69/70] Training: Loss: 0.1179, Accuracy: 96.15%, Validation: Loss: 0.4261, Accuracy: 88.87%
Epoch [70/70] Training: Loss: 0.1173, Accuracy: 96.14%, Validation: Loss: 0.4322, Accuracy: 89.01%
Test Accuracy: 88.88%
All Time: 29 分 57 秒
```



结论：

可以看到模型的过拟合现象大为减弱，前 20 轮验证精度都领先测试精度，说明模型的泛化能力很强。后半段的验证精度稍落后于训练精度，可能因为后期模型已经学到了训练样本足够多的细微特征（包括进行数据增强后新引入的噪声），从而领先了验证集的准确度。

测试集表现上，模型的测试准确度为 88.88%，稍低于只使用 Dropout 和 batch normalization 的模型，可能因为模型学习到一些与测试集无关的变形特征，偏离了测试数据分布，从而影响在测试集上的表现。

2.4、权重衰退

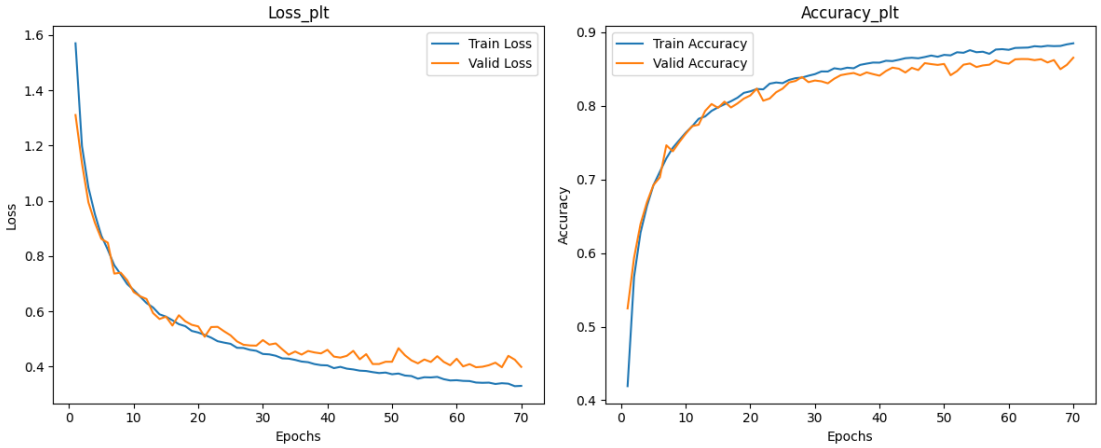
权重衰退是一种正则化技术，用于防止神经网络中的权重值过大，以提升模型

的泛化能力，减少过拟合风险。其主要通过在损失函数中加入一个惩罚项，将过大的权重值压缩到较小的范围内，使模型更简单且不易对噪声敏感。在以前优化的基础上，我们在优化器开启权重衰退选项，并将衰退值设置为较大的1e-4,从而增强模型的泛化性：

训练结果：

```
optimizer = optim.Adam(model.parameters(), lr=best_lr, weight_decay=1e-4) # 启用权重衰退
```

```
Epoch [22/70] Training: Loss: 0.5047, Accuracy: 82.21%, Validation: Loss: 0.5432, Accuracy: 80.67%
Epoch [23/70] Training: Loss: 0.4919, Accuracy: 82.98%, Validation: Loss: 0.5440, Accuracy: 80.97%
...
Epoch [68/70] Training: Loss: 0.3383, Accuracy: 88.10%, Validation: Loss: 0.4390, Accuracy: 84.95%
Epoch [69/70] Training: Loss: 0.3293, Accuracy: 88.32%, Validation: Loss: 0.4250, Accuracy: 85.54%
Epoch [70/70] Training: Loss: 0.3303, Accuracy: 88.46%, Validation: Loss: 0.3991, Accuracy: 86.51%
Test Accuracy: 88.45%
```



结论：

可以看到全程过拟合现象大为减弱，只有后 20 轮发生了轻微过拟合现象。因为采用了 L2 惩罚项，使得模型复杂度降低，不容易学到细微噪声，所以模型的验证精度、验证损失一直随训练误差变化。

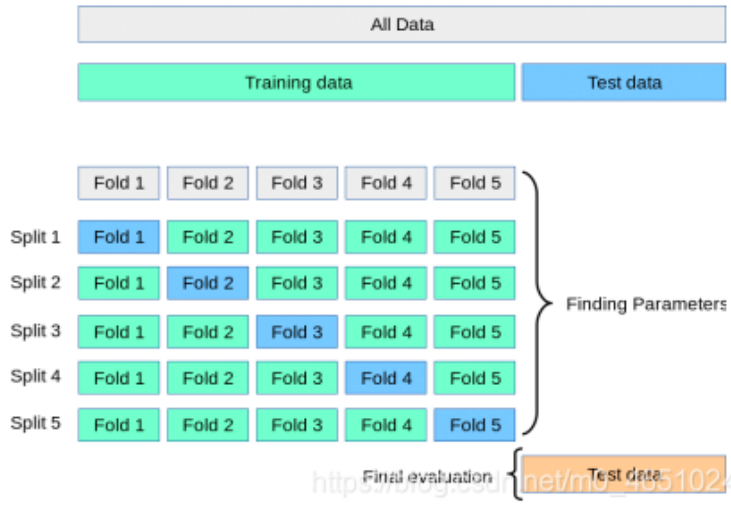
在损失上，训练损失没有降低到很小，推测是因为模型学到的更多是全局特征，对于特定训练集的学习程度减弱，而泛化能力增强。

测试准确度上，模型的准确度为 88.45%，比验证集上的准确度多 2 个点，说明模型对于未知数据的预测能力同样很强，说明了经过各种正则后，模型泛化能力已经达到很高的水平。

三、交叉验证与超参数搜索

3.1、交叉验证

交叉验证是一种用于评估模型性能的技术，它将数据分成多个子集、反复训练和验证模型，从而得到更稳定的模型评估结果。前面我们在训练集上按照 4:1 划分出了验证集和测试集。但这种随机划分单一且固定，模型容易学到单次划分出的随机噪声。现在我们将训练集划分成 5 折，每一折都当做一次验证集，其余折当做训练集，共训练 5 次，数据集的划分示意图如下：



下面我们使用 KFold 库进行 5 折交叉验证，模型选用正则化的 15 层残差网络，实现细节如图：

```
# K折交叉验证设置
k_folds = 5
kf = KFold(n_splits=k_folds, shuffle=True)
fold_train_accuracies = [] # 初始化保存每折的结果
fold_valid_accuracies = []

# 模型训练
def train(num_epochs):···

# 绘图函数
def PLT(epochs, train_losses, valid_losses, train_accuracies, valid_accuracies):··
```

```python
# 5折交叉验证
for fold, (train_idx, valid_idx) in enumerate(kf.split(train_dataset)):
    print(f"Fold {fold + 1}/{k_folds} :")
    train_losses, valid_losses = [], []
    train_accuracies, valid_accuracies = [], []
    # 构造每折的训练和验证数据
    train_subset = torch.utils.data.Subset(train_dataset, train_idx)
    valid_subset = torch.utils.data.Subset(train_dataset, valid_idx)
    train_data_size = len(train_subset)
    valid_data_size = len(valid_subset)
    test_data_size = len(test_dataset)
    train_loader = DataLoader(dataset=train_subset, batch_size=batch_size, shuffle=True)
    valid_loader = DataLoader(dataset=valid_subset, batch_size=batch_size, shuffle=False)
```

训练结果：

```
Fold 1/5 :
Fold: Train Accuracy: 95.09%, Validation Accuracy: 88.20%


 Fold 2/5 :
 Fold: Train Accuracy: 94.99%, Validation Accuracy: 88.52%


 Fold 3/5 :
 Fold: Train Accuracy: 94.70%, Validation Accuracy: 87.96%


 Fold 4/5 :
 Fold: Train Accuracy: 94.95%, Validation Accuracy: 87.80%


 Fold 5/5 :
 Fold: Train Accuracy: 94.89%, Validation Accuracy: 88.37%


 k-fold average Train Accuracy: 94.93%
 k-fold average Validation Accuracy: 88.17%
 Test Accuracy: 88.83%
```

结论：

模型在每一折上的准确率都大致相同，原因是数据集预处理为乱序，使得模型的随机性减弱。并且当数据集的样本量较大时，每一折的训练集和验证集都能较好地代表整个数据集的分布，减少了由于样本划分带来的波动。

交叉验证同样也是一种防止过拟合的方法，它采用不同折上的平均（或加权平均）准确率作为评估标准，相当于一种变换的“数据增强”。

交叉验证非常适合小样本的情况，因为它能够有效地利用有限的数据资源，最大化模型的学习和评估效果。

3.2、交叉验证调参

交叉验证同样是一种有效的调参方法，因为它能够帮助模型在有限数据上评估性能的稳定性和泛化能力。通过交叉验证，可以找到模型在不同超参数设置下的最佳配置，同时减少过拟合和低估模型性能的风险。现代机器学习已经有很多 AutoML 的方案，如使用 Optuna 或 Hyperopt 自动化调参等，但这些方法对算力的消耗巨大，需要多卡并行，价格昂贵。为了演示效果，我们自己制作了超参数网格，为每组超参数使用 5 折交叉验证，最终得到最优的超参数配置。

我们使用了 0.25 采样率的训练集来缩小数据集的规模，并使用三层嵌套循环进行超参数搜索，总的计算复杂度为 O(N4)。

定义的超参数网格如图：

```
# 超参数网格
param_grid = {
    'batch_size': [64, 128, 256],
    'lr': [0.005, 0.001, 0.0005],
    'epochs': [70]
}
```

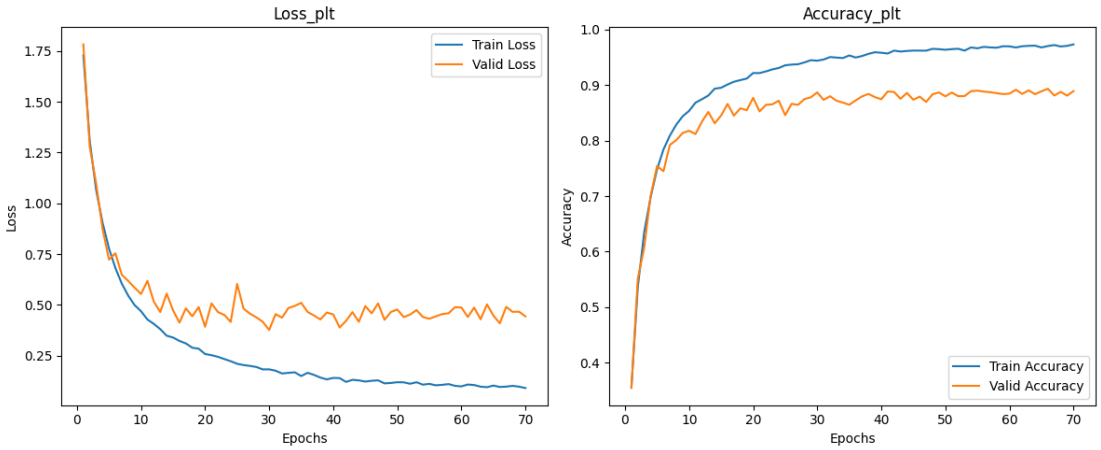
主要代码如图：

```
# 交叉验证
for batch_size in param_grid['batch_size']:
    for lr in param_grid['lr']:
        for epochs in param_grid['epochs']:
            print(f"Training with batch_size={batch_size}, lr={lr}, epochs={epochs}")

            # 使用 KFold 进行交叉验证
            kf = KFold(n_splits=5, shuffle=True)
            fold_valid_accuracies = []

            for fold, (train_idx, valid_idx) in enumerate(kf.split(sampled_dataset)):
                # print(f"Fold {fold + 1}/{k_folds} :")
                train_losses, valid_losses = [], []
                train_accuracies, valid_accuracies = [], []

                train_subset = torch.utils.data.Subset(sampled_dataset, train_idx)
                valid_subset = torch.utils.data.Subset(sampled_dataset, valid_idx)

                train_loader = DataLoader(dataset=train_subset, batch_size=batch_size, shuffle=True)
                valid_loader = DataLoader(dataset=valid_subset, batch_size=batch_size, shuffle=False)

                # 模型、损失函数、优化器
                model = RRRR().to(device)
                criterion = nn.CrossEntropyLoss()
                optimizer = optim.Adam(model.parameters(), lr=lr, weight_decay=1e-4)

                # 调用训练函数，返回该折结果
                avg_train_acc, avg_valid_acc = train(epochs)
```

训练结果：

```
Files already downloaded and verified
Files already downloaded and verified
Training with batch_size=64, lr=0.005, epochs=70
Fold 1: Train Accuracy: 78.01%, Validation Accuracy: 71.76%
Fold 2: Train Accuracy: 74.57%, Validation Accuracy: 71.36%
Fold 3: Train Accuracy: 77.90%, Validation Accuracy: 72.12%
Fold 4: Train Accuracy: 76.31%, Validation Accuracy: 73.92%
Fold 5: Train Accuracy: 80.59%, Validation Accuracy: 75.28%
Mean Validation Accuracy for batch_size=64, lr=0.005, epochs=70: 72.89%
----------------------------------------------------------------
Training with batch_size=64, lr=0.001, epochs=70
Fold 1: Train Accuracy: 95.98%, Validation Accuracy: 81.04%
Fold 2: Train Accuracy: 95.60%, Validation Accuracy: 80.64%
Fold 3: Train Accuracy: 96.35%, Validation Accuracy: 80.64%
Fold 4: Train Accuracy: 95.93%, Validation Accuracy: 81.04%
Fold 5: Train Accuracy: 95.70%, Validation Accuracy: 82.92%
Mean Validation Accuracy for batch_size=64, lr=0.001, epochs=70: 81.26%
----------------------------------------------------------------
Training with batch_size=64, lr=0.0005, epochs=70
Fold 1: Train Accuracy: 97.35%, Validation Accuracy: 81.24%
Fold 2: Train Accuracy: 97.76%, Validation Accuracy: 81.68%
Fold 3: Train Accuracy: 96.34%, Validation Accuracy: 83.04%
Fold 4: Train Accuracy: 97.16%, Validation Accuracy: 78.08%
Fold 5: Train Accuracy: 97.43%, Validation Accuracy: 81.96%
Mean Validation Accuracy for batch_size=64, lr=0.0005, epochs=70: 81.20%
...
----------------------------------------------------------------
Best Params: Batch Size=64, Learning Rate=0.001, Epochs=70 with Validation Accuracy=81.26%
Best model performance: Train Accuracy: 97.34%, Validation Accuracy: 88.93%
----------------------------------------------------------------
```

Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...




```
Test Accuracy: 88.40%
All Time: 284 分 2 秒
```

结论：

在超参数网络搜索下，找到的最优参数是：Batch Size=64, Learning Rate=0.001, Epochs=70 其验证准确度为 81.26%，因为这是五折交叉验证出的结果，结果具有更高的置信度。

在最优超参数配置下，重新用整个训练集训练模型，得到的测试精度为 88.4%，

总体超过了以往其它所有参数配置下的测试准确度，说明超参数在该配置下的确得到了最优结果。

交叉验证搜索超参数计算量非常大，总的训练时长为 284min，所以需要较高的调参配比，使得模型在有限的算力下尽可能遍历更大的超参数空间。

四、模型融合

模型融合是一种集成学习方法，它结合了多个模型的预测结果，以提升整体预测性能。模型融合的核心思想是通过多个不同模型的优势互补，来减少单个模型预测中的偏差和方差，减少过拟合风险，从而得到更稳健的结果。

在以往测试中，模型都是由我们自己定义，存在一定局限性，为了使模型发挥出更好的性能，我们将自己设计的 CNN 模型同预训练的 ResNet18 模型融合，获得更优良的性能。

具体来讲，先并行训练多个模型，然后设计多模型输出结果的融合函数，在测试集上得到多模型的融合测试结果。

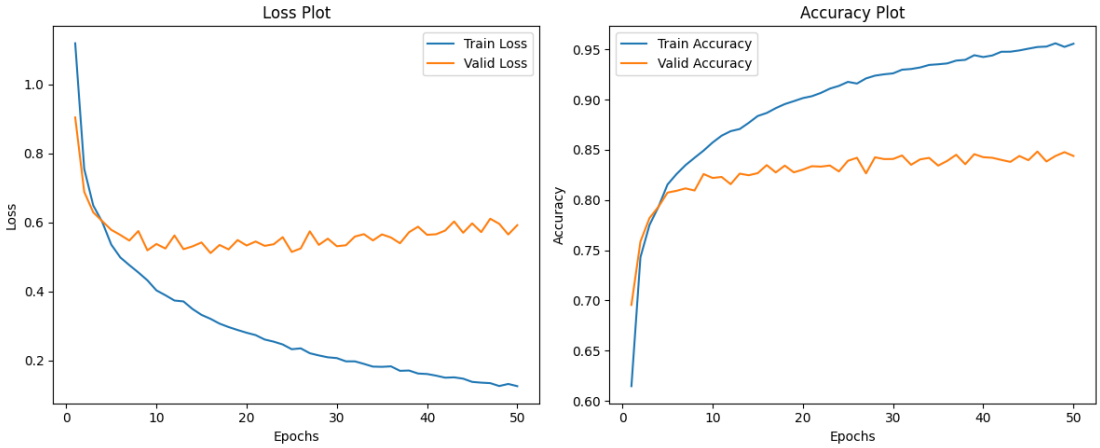
实现细节：

```
# 融合函数的定义
def ensemble(models_set, images, device):
    """进行模型融合"""
    outputs = [m(images) for m in models_set]
    avg_output = sum(outputs) / len(models_set)  # 计算加权平均
    return avg_output
# 模型集合
models_set = []
```

```
# 执行训练和评估
def test_ensemble(models_set):
    """测试融合模型的准确性"""
    correct = 0
    total = 0
    with torch.no_grad():
        for images, labels in test_loader:
            images, labels = images.to(device), labels.to(device)
            outputs = ensemble(models_set, images, device)  # 修改此处
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
    print(f"Ensemble Test Accuracy: {100 * correct / total:.2f}%")
```

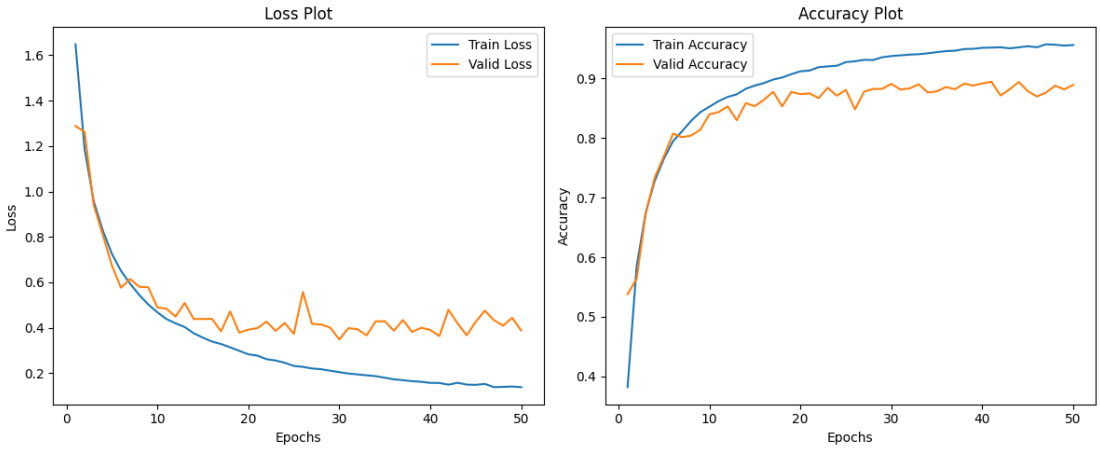
训练结果：

```
Epoch [21/50] Training: Loss: 0.2729, Accuracy: 90.34%, Validation: Loss: 0.5442, Accuracy: 83.37%
Epoch [22/50] Training: Loss: 0.2601, Accuracy: 90.67%, Validation: Loss: 0.5316, Accuracy: 83.33%
Epoch [23/50] Training: Loss: 0.2539, Accuracy: 91.10%, Validation: Loss: 0.5362, Accuracy: 83.44%
...
Epoch [47/50] Training: Loss: 0.1337, Accuracy: 95.29%, Validation: Loss: 0.6102, Accuracy: 83.84%
Epoch [48/50] Training: Loss: 0.1253, Accuracy: 95.62%, Validation: Loss: 0.5954, Accuracy: 84.39%
Epoch [49/50] Training: Loss: 0.1313, Accuracy: 95.26%, Validation: Loss: 0.5645, Accuracy: 84.77%
Epoch [50/50] Training: Loss: 0.1249, Accuracy: 95.57%, Validation: Loss: 0.5915, Accuracy: 84.39%
```



图表 1

```
Epoch [24/50] Training: Loss: 0.2454, Accuracy: 92.14%, Validation: Loss: 0.4206, Accuracy: 87.11%
Epoch [25/50] Training: Loss: 0.2316, Accuracy: 92.75%, Validation: Loss: 0.3726, Accuracy: 88.09%
...
Epoch [47/50] Training: Loss: 0.1380, Accuracy: 95.73%, Validation: Loss: 0.4333, Accuracy: 87.63%
Epoch [48/50] Training: Loss: 0.1391, Accuracy: 95.67%, Validation: Loss: 0.4088, Accuracy: 88.80%
Epoch [49/50] Training: Loss: 0.1408, Accuracy: 95.52%, Validation: Loss: 0.4433, Accuracy: 88.17%
Epoch [50/50] Training: Loss: 0.1379, Accuracy: 95.60%, Validation: Loss: 0.3876, Accuracy: 88.93%
```



图表 2

```
Ensemble Test Accuracy: 90.61%
```

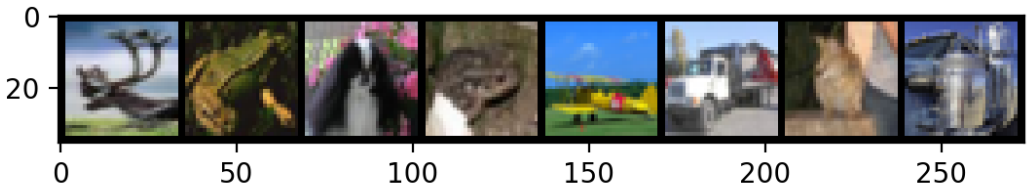
结论：

图表一是 ResNet18 的训练结果，图表二是基于 AlexNet 设计的 15 层残差网络的结果，两个模型的验证准确率分别是 84.39%、88.93%，在模型融合后，测试集上的预测准确度为 90.61%，可见模型融合极大提升了模型的预测能力，相交于单一模型具有更多的特征学习能力。尽管两个模型各自表现已经足够良好，

但在特征学习和泛化能力上有所差异，通过模型融合，将两个模型的特征学习能力可以进一步互补，从而更全面地捕捉数据的特征模式，提高了整体的预测准确性。

五、模型应用

最后简单评测一下模型在小样本上的预测能力，同样选用正则化后的 15 层 ResNet 模型，选取测试集中的小样本如下：



模型预测结果：

```
PS C:\E\PycharmProject\DL\CNN> & C:/Users/86151/anaconda3/envs/DGL_cpu/python.exe
Files already downloaded and verified
Predicted:  deer  frog  dog   dog   deer  truck dog   dog
真实标签为:  deer  frog  dog   frog  plane truck cat   truck
```

随机选取的 8 个小样本中，模型预测对了 4 个，说明我们训练的模型符合预期，且具有一定的可解释性。

六、总结

本次实验基于 CNN 完成了图像分类任务，采用分类准确率作为主要的评价指标。在实验过程中，我尝试了多种 CNN 网络结构，并应用了 dropout、weight_decay、batch_normalization、data_augmentation 等正则化方法,掌握了一系列对抗过拟合的技巧。此外，训练策略上，我还采用交叉验证测试模型在数据集的不同划分层次上的综合表现，并使用五折交叉验证完成了超参数网格搜索，最终确定了表现最佳的超参数配置。最后，我用加权平均完成了 ResNet18 预训练模型与自己设计的 CNN 模型的融合，并获得了最优的测试预测结果。

通过本次实验，我更加深入地理解了卷积神经网络训练、验证和测试的整体流程，尤其是深入了解了交叉验证等训练策略。本次实验的深入探索使我掌握了从模型设计、正则化应用、超参数调优到模型集成的完整流程，提升了我在应对复杂问题时的解决能力。

参考代码

[1] 《动手学深度学习》7.6. 残差网络（ResNet）

[2] 5.11 残差网络

[3] Deep Learning -- Normalization

[4] 查看 CutOut 代码

[5] pytorch - K 折交叉验证过程说明及实现

[6]apachecn\apachecn-dl-zh