深度学习原理与实践 实验一：基于CNN的CIFAR-10分类

PB22151796-莫环欣

实验目的

- 掌握 `CNN` 的基本结构与实现方法
- 学习图像分类任务的完整流程
  - 数据预处理
  - 模型设计
  - 训练与评估
- 熟悉 `PyTorch` 框架的基础操作
- 分析模型性能并提出改进方法
- 尝试实现拓展内容

`CNN` 可以通过多层结构（主要是其本身的卷积层）提取出图像的特征，之后可以在全连接层根据特征对图像进行分类

实验环境

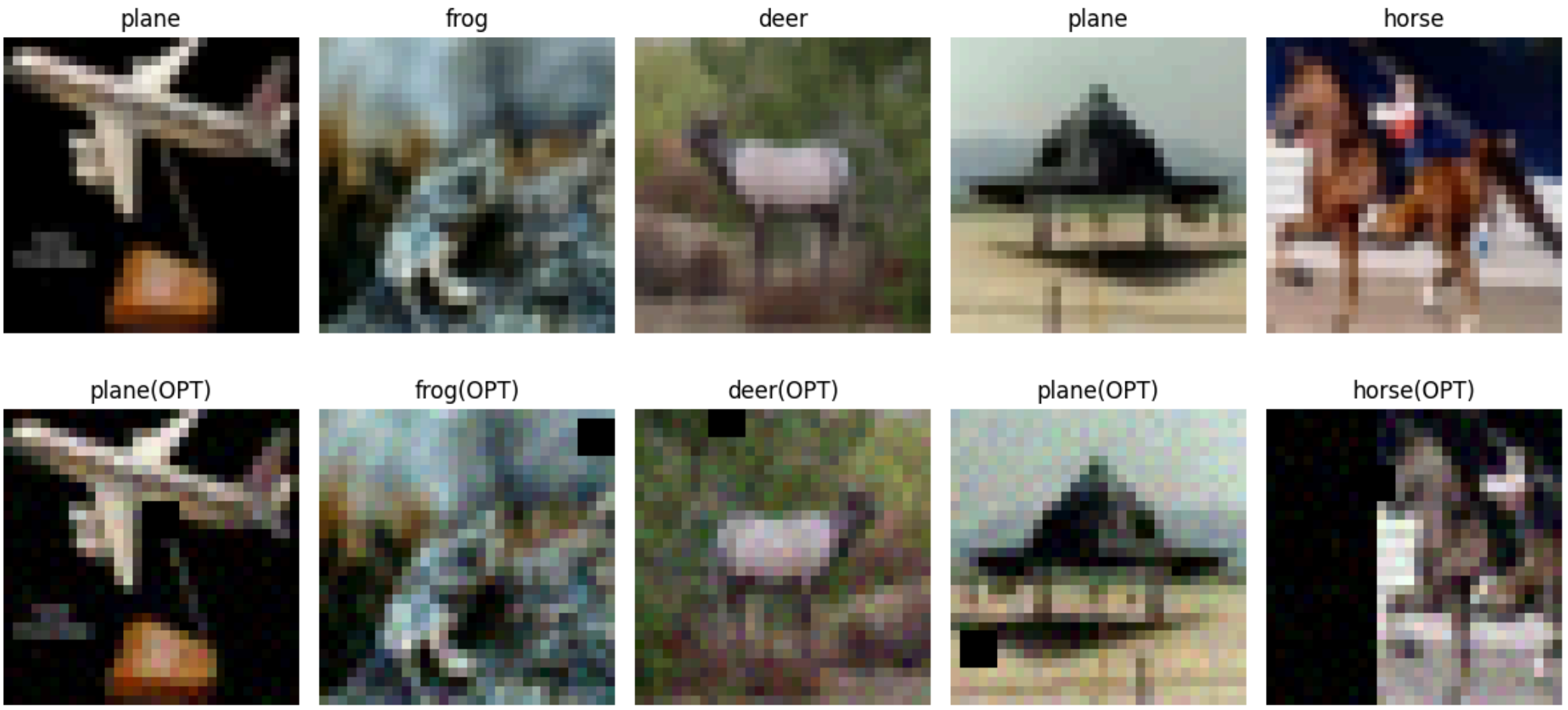
- 包环境
  - `python` :3.10.10
  - `torch` :2.6.0
  - `numpy` :2.0.2
  - `seaborn` :0.13.2
  - `scikit-learn` :1.6.1
- 训练环境
  - `colab GPU` : T4
- 已知测试环境支持
  - `CPU` : 12th Gen Intel(R) Core(TM) i7-12700H 2.30 GHz
  - `GPU` : T4

数据集描述

经过对未特殊处理的图片的`AI`（人工自能）识别，确定`label`中的编号对应以下含义（类别与数字索引对应）

```
["plane","car","bird","cat","deer","dog","frog","horse","ship","truck",]
```

下面展示五张图片（上方）与其经过强数据增强后（下方）的示例



训练集与测试集统计信息

```
训练集统计信息:
数据集大小: 50000
通道数: 3，图像尺寸: 32x32
类别分布:
    plane: 5000 张
    car: 5000 张
    bird: 5000 张
    cat: 5000 张
    deer: 5000 张
    dog: 5000 张
    frog: 5000 张
    horse: 5000 张
    ship: 5000 张
    truck: 5000 张
每通道均值: [-0.192211   -0.17834751 -0.12485716]
每通道标准差: [1.2077042 1.2133846 1.1511583]
像素值范围: -2.6582560539245605 ~ 2.730001449584961
测试集统计信息:
数据集大小: 10000
通道数: 3，图像尺寸: 32x32
类别分布:
    plane: 1000 张
    car: 1000 张
    bird: 1000 张
    cat: 1000 张
    deer: 1000 张
    dog: 1000 张
    frog: 1000 张
    horse: 1000 张
    ship: 1000 张
    truck: 1000 张
```

```
每通道均值: [0.01139386 0.01203861 0.0149431 ]
每通道标准差: [0.9985933  0.99750423 0.999971  ]
像素值范围: -1.9894737005233765 ~ 2.12648868560791
```

基础实验步骤

数据加载

首先需要加载数据，这里使用`torch`内置的`CIFAR`接口，并将像素值从`[0,255]`缩放到`[0,1]`

```
transform = transforms.Compose(
    [
        transforms.ToTensor(),
        transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2470, 0.2435, 0.2616)),
    ]
)

train_dataset = datasets.CIFAR10(
    root="./data", train=True, download=True, transform=transform
)
test_dataset = datasets.CIFAR10(
    root="./data",
    train=False,
    download=True,
    transform=transform
)
```

随后将数据集作二次划分，从训练集中划出`10000`个样本单独分给验证集

```
# 划分验证集
validation_size = int(0.2 * len(train_subset))
train_size = len(train_subset) - validation_size
train_subset, validation_subset = torch.utils.data.random_split(
    train_subset, [train_size, validation_size]
)

DL_batch_size = 1024
train_loader = DataLoader(train_subset, batch_size=DL_batch_size, shuffle=True, num_workers=2, pin_memory=True)
validation_loader = DataLoader(validation_subset, batch_size=DL_batch_size, num_workers=2, pin_memory=True)
test_loader = DataLoader(test_subset, batch_size=DL_batch_size, num_workers=2, pin_memory=True)
```

这里`colab`允许的`num_workers`上限为`2`

基础模型结构

基础部分的模型参考了文档中的示例，卷积部分共分三组，每组首先使用一个卷积层来提取出局部特征，再使用`ReLU`激活函数引入非线性，最后接到一个`2x2`的最大池化层以减少特征图尺寸（下采样）。第一个卷积层的输入通道为`3`（即`RGB`），之后每层的输入通道都对应上一层的输出通道（且为`2`的倍数），卷积核大小设置为`3x3`，使用`padding=1`保持特征图的空间尺寸不变

全连接部分含两个全连接层，第一次连接之前首先需要将卷积部分的结果展平为一维向量，两个全连接层之间接上了`ReLU`激活函数与`Dropout`层（随机丢弃部分神经元，防止过拟合），最后输出`10`个分类结果

（这里仅为基础示例，最终结果请看下文拓展部分）

```python
class BasicCNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv_layers = nn.Sequential(
            nn.Conv2d(in_channels= 3,out_channels= 32, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2, 2),  # 32/2 , 32/2

            nn.Conv2d(in_channels= 32,out_channels= 64, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2, 2),  # 16/2 , 16/2

            nn.Conv2d(in_channels= 64,out_channels= 128, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2, 2),  # 8/2 , 8/2   -> 4x4
        )
        self.fc_layers = nn.Sequential(
            nn.Flatten(),
            nn.Linear(128 * 4 * 4, 512),
            nn.ReLU(),
            nn.Dropout(0.5),
            nn.Linear(512, 10),
        )
    def forward(self, x):
        x = self.conv_layers(x)
        x = self.fc_layers(x)
        return x
```

训练与验证

- **损失函数**：交叉熵损失函数（`nn.CrossEntropyLoss`）
- **优化器**: 选择了更适合分类任务的`Adam`
- **训练超参数**:
  - `Bath Size`：这里为了更合理地利用`colab`资源，选定为`1024`（其实`2048`也可以装得下）
    - 较大的`Batch Size`可以更高效利用计算资源，梯度估计更平滑，更新方向、训练过程更稳定，但收敛可能变慢，通常需要较大的学习率
      - 老师`28`号的课上说越大越好，还可以对应调高学习率
  - 学习率：初值设定为`0.03`，并结合学习率调度器，容忍损失`5`轮不发生变化，超过则将学习率乘`0.1`（默认值）
    - `torch.optim.lr_scheduler.ReduceLROnPlateau(self.optimizer, "min", patience=5)`

训练过程分为`epochs`轮，每轮分为训练、验证、调整学习率、保存最佳模型四步

- 训练部分：
  - 对于每轮训练，循环从训练数据加载器中读出输入特征图与其标签，装载到设备上，然后就是非常经典的五步走（梯度清零->前向传播->计算损失->反向传播计算梯度->更新参数），随后就是累计损失与预测准确数
  - 数据全部处理完后，计算并返回训练准确率与训练损失
    - 为避免训练过程中可能发生的错误，这里处理的数据总数在训练过程中累加，而非直接使用装载器数据总量的统计值
    - `total += labels.size(0)`

```python
self.model.train()
loss_cnt = 0.0
correct = 0
total = 0
for inputs, labels in self.train_loader:
    inputs, labels = inputs.to(self.device), labels.to(self.device)
    self.optimizer.zero_grad()              # 梯度清零
    outputs = self.model(inputs)            # 调用上面搭建的前向传播
    loss = self.loss_func(outputs, labels)  # 计算损失
    loss.backward()                         # 反向传播，计算梯度
    self.optimizer.step()                   # 更新参数

    loss_cnt += loss.item()
    _, predicted = torch.max(outputs, 1)
    total += labels.size(0)     # 加载多少算多少
    correct += (predicted == labels).sum().item()
```

- 验证部分：
  - 同样是从加载器（验证集）中循环读出数据并装载，随后进行预测并计算损失，另外还需禁用梯度

```
self.model.eval()
validation_loss = 0.0
correct = 0
total = 0
with torch.no_grad():  # 验证阶段不需要计算梯度
    for inputs, labels in self.validation_loader:
        inputs, labels = inputs.to(self.device), labels.to(self.device)
        outputs = self.model(inputs)  # 调用上面搭建的前向传播
        loss = self.loss_func(outputs, labels)
        validation_loss += loss.item()

        _, predicted = torch.max(
            outputs, 1
        )  # 新版torch可以直接传，不需要带.data，如果zj复现要用旧版可能得带上
        total += labels.size(0)
        correct += (predicted == labels).sum().item()
```

测试部分主要内容与上面的验证部分基本一致，不再赘述。学习率的调整直接使用上面学习率调度器的接口。最后比对当前轮的预测准确率，如果比历史最佳准确率更好，就将其保存为最佳模型

测试部分主要多出的内容是各分类准确率的统计，在分类过程中，会记录预测标签与实际标签信息

```
test_labels.extend(labels.cpu().numpy())
test_preds.extend(predicted.cpu().numpy())
```

分类全部结束后输出

```
rp = classification_report(
    test_labels,
    test_preds,
    target_names=[str(i) for i in range(10)],
    output_dict=True,
)
for i in range(10):
    print(f"Accuracy for class {cifar10_classes[i]}: {rp[str(i)]['precision']:.4f}")
```

这一部分训练器的代码与后续部分的相同，复现时可以直接拿来使用，后续完整模型的接口与此处也相同

结果分析

定量结果

Epoch	Train Loss	Train Acc	Validation Loss	Validation Acc
1	0.0026	0.2084	0.0019	0.3198
2	0.0018	0.3599	0.0017	0.3990
3	0.0017	0.3981	0.0016	0.4224
4	0.0016	0.4263	0.0016	0.4192
5	0.0015	0.4458	0.0015	0.4441
6	0.0015	0.4589	0.0015	0.4716
7	0.0015	0.4759	0.0015	0.4743
8	0.0015	0.4762	0.0014	0.4871
9	0.0014	0.4913	0.0015	0.4759
10	0.0014	0.4941	0.0014	0.5046

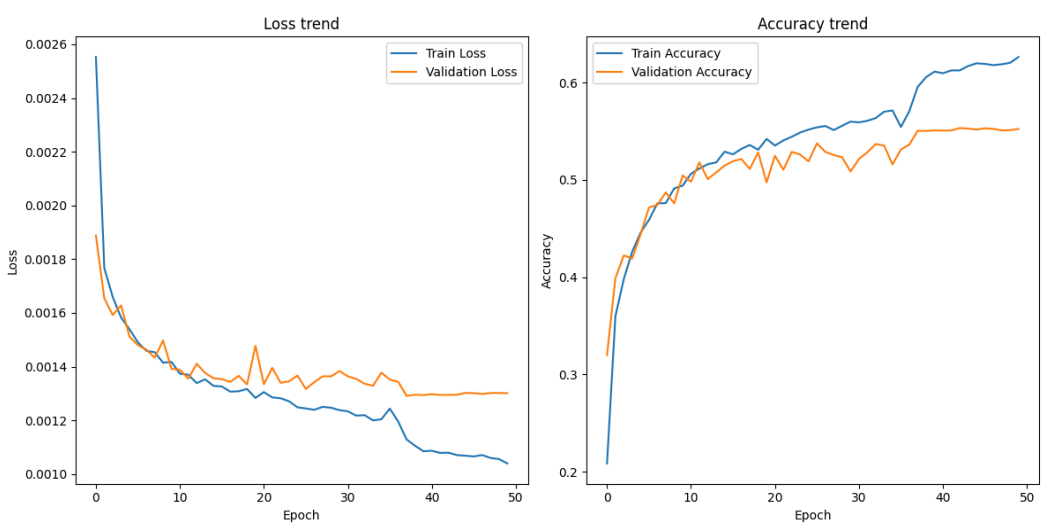
Epoch	Train Loss	Train Acc	Validation Loss	Validation Acc
11	0.0014	0.5063	0.0014	0.4982
12	0.0014	0.5117	0.0014	0.5180
13	0.0013	0.5162	0.0014	0.5008
14	0.0014	0.5180	0.0014	0.5078
15	0.0013	0.5291	0.0014	0.5147
16	0.0013	0.5263	0.0014	0.5193
17	0.0013	0.5319	0.0013	0.5215
18	0.0013	0.5359	0.0014	0.5113
19	0.0013	0.5310	0.0013	0.5283
20	0.0013	0.5422	0.0015	0.4974
21	0.0013	0.5353	0.0013	0.5248
22	0.0013	0.5406	0.0014	0.5105
23	0.0013	0.5444	0.0013	0.5288
24	0.0013	0.5487	0.0013	0.5261
25	0.0012	0.5518	0.0014	0.5191
26	0.0012	0.5541	0.0013	0.5376
27	0.0012	0.5554	0.0013	0.5288
28	0.0013	0.5513	0.0014	0.5256
29	0.0012	0.5557	0.0014	0.5233
30	0.0012	0.5599	0.0014	0.5087
31	0.0012	0.5592	0.0014	0.5215
32	0.0012	0.5608	0.0014	0.5284
33	0.0012	0.5636	0.0013	0.5369
34	0.0012	0.5702	0.0013	0.5353
35	0.0012	0.5715	0.0014	0.5159
36	0.0012	0.5545	0.0014	0.5313
37	0.0012	0.5707	0.0013	0.5363
38	0.0011	0.5957	0.0013	0.5506
39	0.0011	0.6057	0.0013	0.5503
40	0.0011	0.6114	0.0013	0.5509
41	0.0011	0.6097	0.0013	0.5507
42	0.0011	0.6126	0.0013	0.5509
43	0.0011	0.6126	0.0013	0.5533
44	0.0011	0.6171	0.0013	0.5528
45	0.0011	0.6199	0.0013	0.5518
46	0.0011	0.6193	0.0013	0.5531
47	0.0011	0.6181	0.0013	0.5525
48	0.0011	0.6189	0.0013	0.5509

Epoch	Train Loss	Train Acc	Validation Loss	Validation Acc
49	0.0011	0.6204	0.0013	0.5512
50	0.0010	0.6264	0.0013	0.5524

Class	Accuracy
plane	0.6349
car	0.6687
bird	0.4625
cat	0.3546
deer	0.4715
dog	0.4195
frog	0.5988
horse	0.5809
ship	0.6370
truck	0.6105
average	**0.5460**

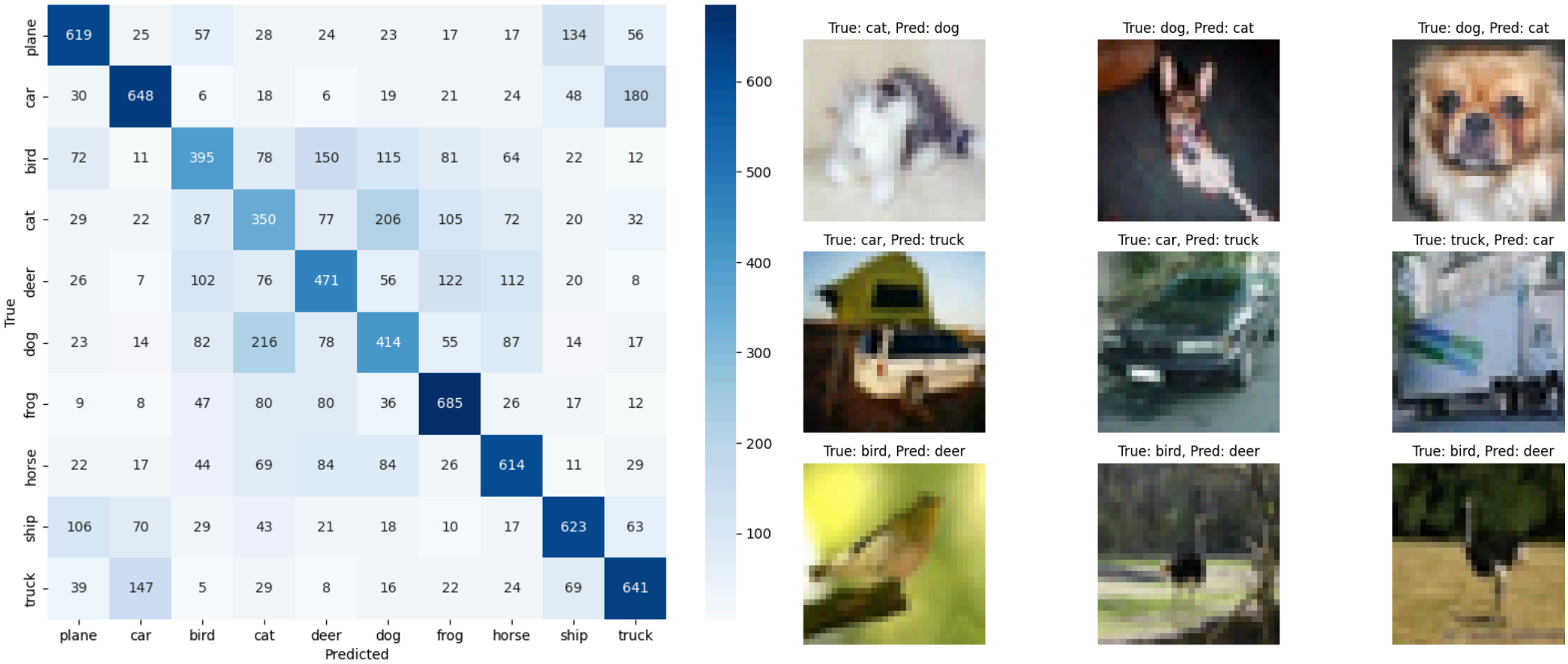
可以看到整体的分类准确率都较低，而动物类的准确率尤其低（特别是猫和狗）

定性分析



可以看到验证集的损失曲线在第`40`轮时就开始收敛了，其分类准确率也基本不再上升；而尽管训练集还有些上升趋势，但对验证集的影响已经不大，双方差距已经拉开，后续必然过拟合

再看混淆矩阵，最难分辨的首先是猫和狗，其次是卡车和小汽车，接着是船和飞机，最后是鹿和鸟



一方面，上面提到的几组各自在图片中的大小就很相似，像猫狗的毛发、尾巴、耳朵等细节特征不明显（比如第一组中的后两张图，耳朵有长有短）；飞机和船都很大很长；卡车和小汽车都接近长方形，外部配饰也相似。特征相近、图片又没那么清晰的情况下（原图就很模糊，更别说训练集还做了数据增强），模型提取细节特征相对就比较困难；而不同类（动物和机械）之间由于全局特征差别很大，这种程度的图片模糊不足以掩盖它们之间的区别，相对来说就比较好区分些；同理，在动物、机械类自身内，关键特征相差大的也相对比较好区分

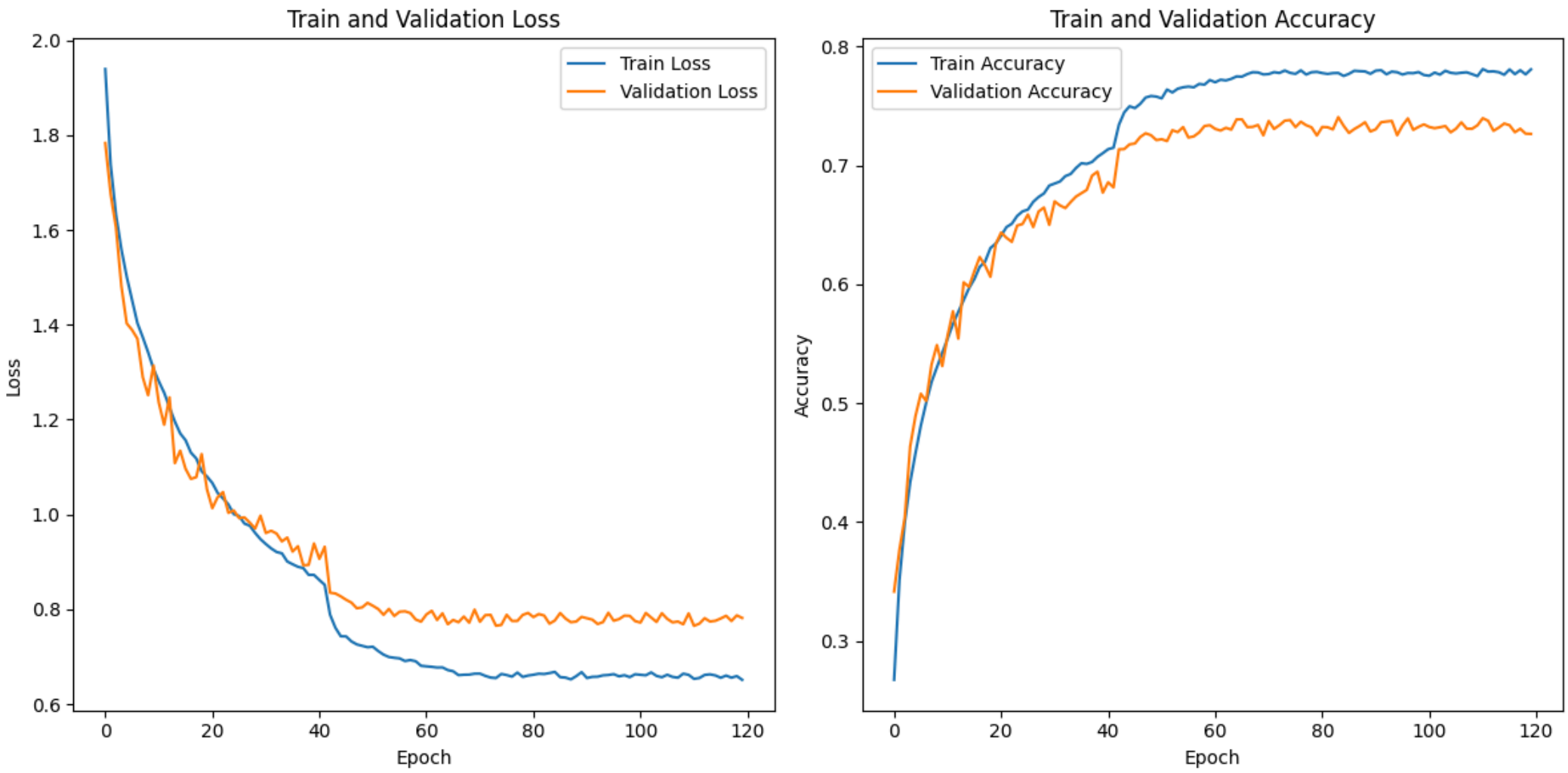
拓展内容

此处为美观起见，中间优化仅展示部分关键结果，最后再展示经过拓展的最终版本模型的训练损失/准确率表格等完整数据

模型改进

- `Dropout`：
  - `Dropout` 是一种正则化技术，用于防止神经网络过拟合。在训练过程中，`Dropout` 随机将一部分神经元的输出设为 `0`，从而迫使网络的其他神经元学习更鲁棒的特征
  - `dropout_rate` 是 `Dropout` 的概率，表示每个神经元被“丢弃”的概率
- `Batch Normalization`：
  - `Batch Normalization` 是一种加速神经网络训练并提高稳定性的方法。它通过在每一层的激活函数之前对输入进行标准化，使得每一层的输入具有零均值和单位方差，从而减小内部协变量偏移
  - `nn.BatchNorm2d` 是 `PyTorch` 中用于 `2D` 卷积层的 `Batch Normalization`

丢弃层与归一化层可以在一定程度上显著提高预测的正确率，在`0.1`份数据集及相同配置情况下，启用之后在测试集与验证集上的平均预测准确率上升了约`10%`（`54% ---> 64%`）



之后在文档提供的基础代码上增大卷积层数，并微调了一部分参数：

- 卷积部分：
  - 保留原先的三组结构，每组内使用两组卷积层（包含各自的`BatchNorm`和`ReLU`），最后的输出依然经过最大池化
  - 各组内的第二个卷积层保持输入维度与输出维度等同（即继续提取更多内容，同时保持计算效率和稳定性）
  - 第一组的输出维度扩大到`128`，后续各组的输入与输出依然保持一倍的增幅
  - 其余参数不变
    - 对`CIFAR-10`任务，设置`kernel_size`为`3x3`比较好
- 全连接部分：
  - 新增一个全连接层，每个全连接层之间新增一个`BatchNorm`，不过`Dropout`只需要一层就够了（不然丢得太多影响会很大）

```python
class SimpleCNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv_layers = nn.Sequential(
            # 第一组卷积层
            nn.Conv2d(in_channels=3, out_channels=128, kernel_size=3, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(),
            nn.Conv2d(in_channels=128, out_channels=128, kernel_size=3, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(),
            nn.MaxPool2d(2, 2),  # 池化    16x16

            # 第二组卷积层
            nn.Conv2d(in_channels=128, out_channels=256, kernel_size=3, padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU(),
            nn.Conv2d(in_channels=256, out_channels=256, kernel_size=3, padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU(),
            nn.MaxPool2d(2, 2),  # 池化    8x8

            # 第三组卷积层
            nn.Conv2d(in_channels=256, out_channels=512, kernel_size=3, padding=1),
            nn.BatchNorm2d(512),
            nn.ReLU(),
            nn.Conv2d(in_channels=512, out_channels=512, kernel_size=3, padding=1),
            nn.BatchNorm2d(512),
            nn.ReLU(),
            nn.MaxPool2d(2, 2),  # 池化    4x4
        )
        self.fc_layers = nn.Sequential(
            nn.Flatten(),
            nn.Linear(512 * 4 * 4, 512),
            nn.BatchNorm1d(512),
            nn.ReLU(),
            nn.Dropout(0.5),
            nn.Linear(512, 256),
            nn.BatchNorm1d(256),
            nn.ReLU(),
            nn.Linear(256, 10),
        )

    def forward(self, x):
        x = self.conv_layers(x)
        x = self.fc_layers(x)
        return x
```

一开始训练质量很一般，由于算力有限，一次修改了很多内容，除了模型结构、数据增强之外还向训练器的损失函数中加入了标签平滑、优化器中加入了正则化（`1e-4`），然后发现效果更差了（稳定性也不好），而且训练很困难，删掉训练代码中的这两处参数之后准确率大幅提升（可能是跟我的其它部分没有配合好）

```
Epoch 1/70
Train Loss: 0.0024, Acc: 0.1512 | validation Loss: 0.0023, Acc: 0.1224
Epoch 2/70
Train Loss: 0.0021, Acc: 0.2430 | validation Loss: 0.0023, Acc: 0.1464
Epoch 3/70
Train Loss: 0.0020, Acc: 0.3038 | validation Loss: 0.0023, Acc: 0.1866
Epoch 4/70
Train Loss: 0.0018, Acc: 0.3756 | validation Loss: 0.0023, Acc: 0.2076
Epoch 5/70
Train Loss: 0.0017, Acc: 0.4241 | validation Loss: 0.0021, Acc: 0.3083
Epoch 6/70
Train Loss: 0.0017, Acc: 0.4676 | validation Loss: 0.0019, Acc: 0.3519
Epoch 7/70
Train Loss: 0.0016, Acc: 0.4996 | validation Loss: 0.0019, Acc: 0.3367
Epoch 8/70
Train Loss: 0.0015, Acc: 0.5385 | validation Loss: 0.0021, Acc: 0.3279
Epoch 9/70
Train Loss: 0.0015, Acc: 0.5595 | validation Loss: 0.0025, Acc: 0.2245
Epoch 10/70
Train Loss: 0.0015, Acc: 0.5683 | validation Loss: 0.0019, Acc: 0.3797
Epoch 11/70
Train Loss: 0.0014, Acc: 0.5905 | validation Loss: 0.0019, Acc: 0.3825
Epoch 12/70
Train Loss: 0.0014, Acc: 0.5990 | validation Loss: 0.0016, Acc: 0.4850
Epoch 13/70
Train Loss: 0.0014, Acc: 0.5981 | validation Loss: 0.0015, Acc: 0.5448
Epoch 14/70
Train Loss: 0.0014, Acc: 0.6117 | validation Loss: 0.0016, Acc: 0.4810
Epoch 15/70
Train Loss: 0.0014, Acc: 0.6185 | validation Loss: 0.0024, Acc: 0.2569
```

此外，注意力机制也尝试引入过（`SEBlock`），但提升并不明显，也可能是各层级之间关系没微调好（算力有限，悲）

训练策略优化

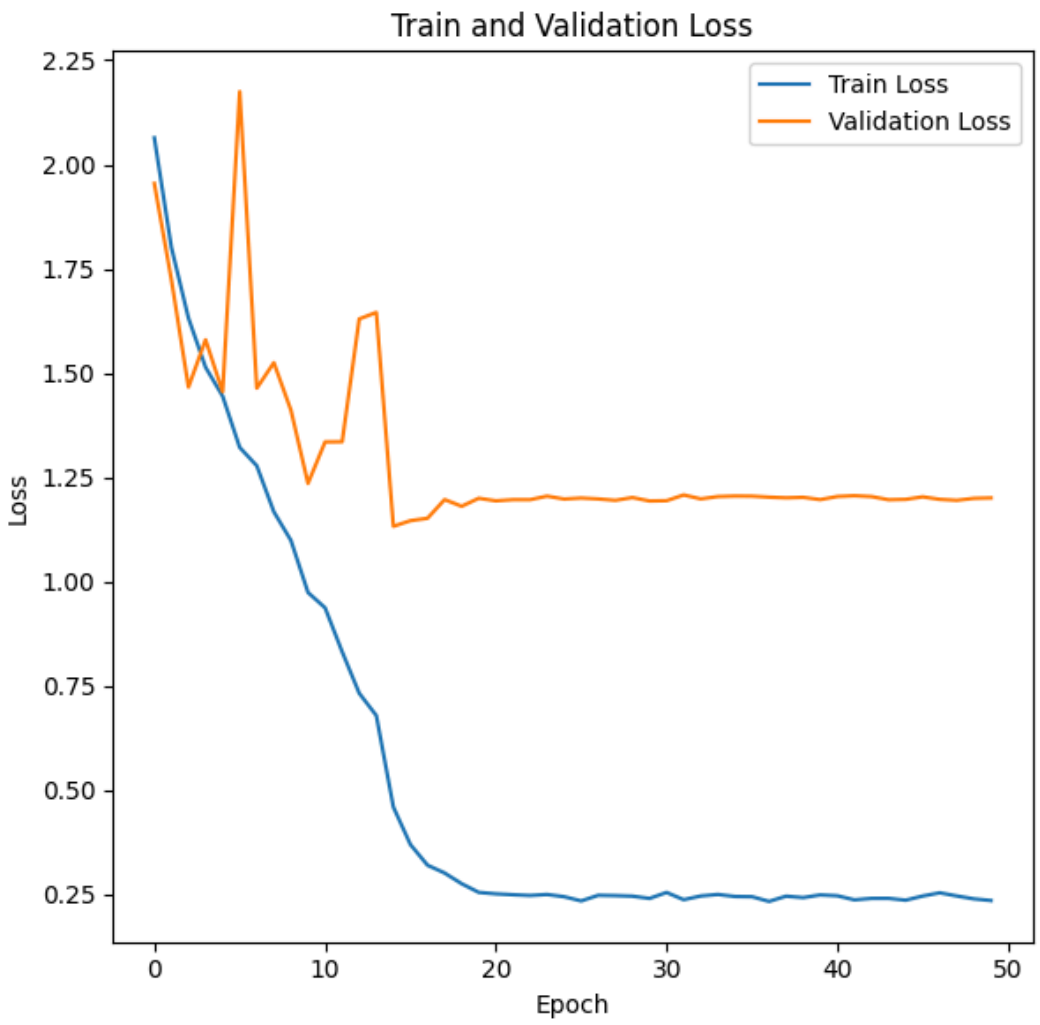
这里在上面基础部分有提到过，使用`torch.optim.lr_scheduler.ReduceLROnPlateau(self.optimizer, "min", patience=5)`策略来动态调整学习率

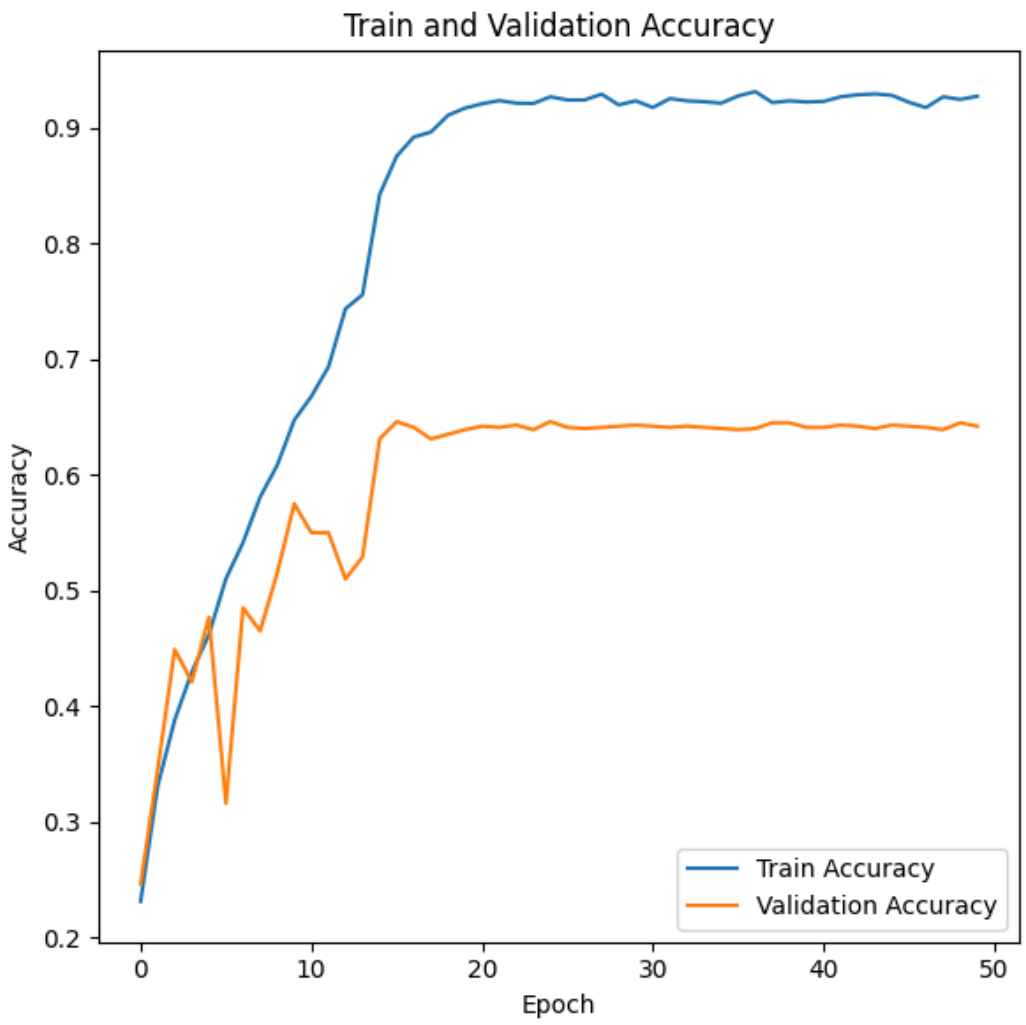
模型评估

使用了混淆矩阵热力图，也分别统计了各类准确率，参考其它部分的分析

数据增强与预处理优化

一开始的时候出现了很严重的过拟合





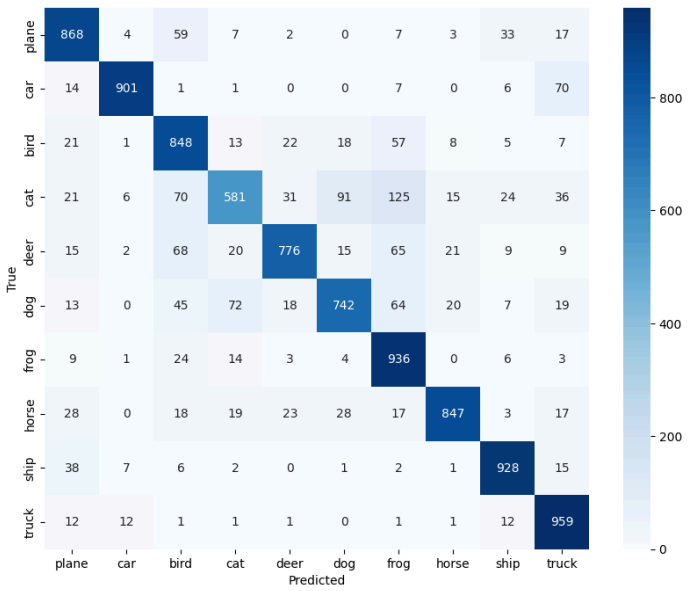
之后发现其实最好的改进可能是想办法减少过拟合，只要训练集收敛的速度下降、验证集的准确率就有机会随之不断提高，同时，由于引入了一些干扰因素（本身数据集的数据量属于比较适中的），模型的泛化能力也有提升；但如果干扰太大，又会使得模型在训练后期学不到什么东西，导致欠拟合现象的出现。

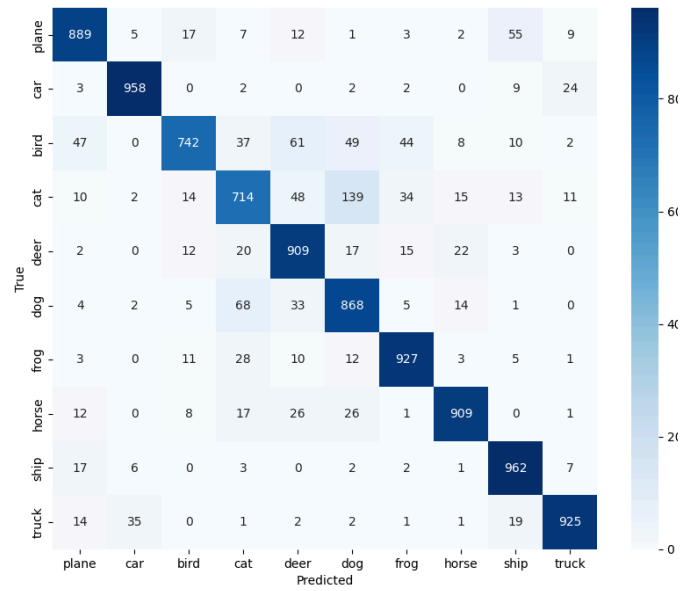
```python
transform = transforms.Compose(
    [
        transforms.ToTensor(),
        transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2470, 0.2435, 0.2616)),
    ]
)
# 先几何变换再裁剪比较适合分类任务，不然容易丢失全局特征
# 强数据增强（前期）
strong_transforms = transforms.Compose(
    [
        AutoAugment(policy=AutoAugmentPolicy.CIFAR10),
        transforms.RandomHorizontalFlip(p=0.5),  # 随机水平翻转
        transforms.ToTensor(),
        DataSetOpt(std=0.03, use_cutout=True),  # 高斯噪声和 Cutout
        transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2470, 0.2435, 0.2616)),
    ]
)
# 弱数据增强（后期）
weak_transforms = transforms.Compose(
    [
        transforms.RandomHorizontalFlip(p=0.5),  # 随机水平翻转
        transforms.RandomCrop(32, padding=4),  # 随机裁剪
        transforms.ToTensor(),
        transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2470, 0.2435, 0.2616)),
    ]
)
# 数据增强
class DataSetOpt:
    def __init__(self, add_gaussian_noise=True, mean=0.0, std=0.03, use_cutout=True, num_holes=1, size=4):
        # 高斯噪声
        self.add_gaussian_noise = add_gaussian_noise
        self.mean = mean
        self.std = std
        # Cutout
        self.use_cutout = use_cutout
        self.num_holes = num_holes
        self.size = size

    def __call__(self, img):
        if self.add_gaussian_noise:
            img = img + torch.randn(img.size()) * self.std + self.mean
        if self.use_cutout:
            h, w = img.size(1), img.size(2)
            for _ in range(self.num_holes):
                y = random.randint(0, h)
                x = random.randint(0, w)
                img[
                    :,
                    max(0, y - self.size // 2) : min(h, y + self.size // 2),
                    max(0, x - self.size // 2) : min(w, x + self.size // 2),
                ] = 0

        return img
```

一开始只使用了随机水平翻转/平移，在四层卷积层结构（最初的结构优化）上测试准确率为`83.86%`；之后改用`AutoAugment(policy=AutoAugmentPolicy.CIFAR10)`自动数据增强，并随机水平翻转（外加高斯噪声与随机打洞），同时使用上面优化后六层卷积层结构（上面提到的三组结构），准确率来到了`88.03%`，再将`batch_size`提高到`1024`，准确率达到`89.48%`





Acc	Case 1	Case 2	Case 3
plane	0.8354	0.8881	0.8612
car	0.9647	0.9504	0.9328
bird	0.7439	0.9172	0.8534
cat	0.7959	0.7960	0.8326
deer	0.8858	0.8256	0.8579
dog	0.8254	0.7764	0.8422
frog	0.7307	0.8965	0.9076
horse	0.9247	0.9323	0.9528
ship	0.8984	0.8932	0.9464
truck	0.8325	0.9439	0.9619
average	**0.8386**	**0.8803**	**0.8948**

可以看到引入上述数据增强后整体分类准确率有了很大提升，尤其是动物类

而在最后一组的基础上，针对细节修改训练时的策略，引入动态的数据增强，总计训练`70`轮，在训练到一定轮次后切换策略，最后十轮切换到无增强的数据集，以期望模型能学到更真实、更细致的内容（此时的学习率也降的差不多了，拿走数据增强不会导致曲线出现震荡）

```
if epoch == 45:  # 在训练中期切换到弱数据增强
    train_dataset.transform = weak_transforms
elif epoch == 60:
    train_dataset.transform = transform
```

从分类准确率的变化趋势可以看出训练策略变化的效果，其中分成了很明显的三段，每一段在即将收敛或收敛一段时间后都有跳跃式的上升，正好对应增强措施的切换。这种处理可以在确保模型泛化能力的同时再进一步提升其分类准确率，最终在测试集上的分类准确率较之前单一强增强策略时又稳定提高了`1%`！

> 本来还想试试`FGSM`的，但是内存爆炸了lol

结果分析

按最终策略对模型训练了两次，结果均能稳定在`90%`，下面的数据中，左侧是最先炼制出的模型结果（附件中的结果），右侧是复现结果（为了不白跑还试图调高`epochs`，强数据增强的时间段提升了`10`轮，其余时间段顺延，最后结果基本不变）

定量分析

Epoch	Train Loss	Train Acc	Validation Loss	Validation Acc	Train Loss	Train Acc	Validation Loss	Validation Acc
1	0.0023	0.1767	0.0021	0.2250	0.0023	0.1681	0.0022	0.2048
2	0.0019	0.2862	0.0019	0.2840	0.0020	0.2586	0.0020	0.2678
3	0.0017	0.3647	0.0017	0.3933	0.0018	0.3228	0.0020	0.3022

Epoch	Train Loss	Train Acc	Validation Loss	Validation Acc	Train Loss	Train Acc	Validation Loss	Validation Acc
4	0.0015	0.4335	0.0016	0.4296	0.0017	0.3906	0.0018	0.3541
5	0.0014	0.5046	0.0016	0.4672	0.0015	0.4535	0.0018	0.4080
6	0.0012	0.5547	0.0014	0.5185	0.0014	0.5113	0.0015	0.4811
7	0.0011	0.5929	0.0012	0.5606	0.0012	0.5559	0.0013	0.5263
8	0.0010	0.6304	0.0013	0.5658	0.0011	0.5972	0.0011	0.6032
9	0.0009	0.6614	0.0011	0.6242	0.0010	0.6344	0.0014	0.5333
10	0.0009	0.6766	0.0010	0.6454	0.0010	0.6444	0.0013	0.5648
11	0.0009	0.6891	0.0010	0.6565	0.0009	0.6742	0.0011	0.6165
12	0.0008	0.7061	0.0010	0.6434	0.0009	0.6837	0.0011	0.6213
13	0.0008	0.7141	0.0009	0.6829	0.0009	0.6834	0.0011	0.6112
14	0.0008	0.7261	0.0009	0.6924	0.0008	0.7129	0.0010	0.6690
15	0.0008	0.7267	0.0009	0.6838	0.0008	0.7241	0.0014	0.5696
16	0.0007	0.7448	0.0008	0.7058	0.0007	0.7443	0.0010	0.6728
17	0.0007	0.7603	0.0009	0.7038	0.0007	0.7481	0.0009	0.6923
18	0.0007	0.7618	0.0008	0.7219	0.0007	0.7497	0.0008	0.7328
19	0.0007	0.7659	0.0008	0.7233	0.0007	0.7617	0.0012	0.6076
20	0.0007	0.7745	0.0009	0.7145	0.0007	0.7637	0.0008	0.7310
21	0.0006	0.7775	0.0009	0.7161	0.0006	0.7729	0.0007	0.7402
22	0.0006	0.7907	0.0007	0.7612	0.0006	0.7788	0.0008	0.7257
23	0.0006	0.7886	0.0008	0.7286	0.0007	0.7757	0.0009	0.6985
24	0.0006	0.8005	0.0007	0.7705	0.0006	0.7777	0.0008	0.7329
25	0.0005	0.8157	0.0008	0.7207	0.0006	0.7961	0.0007	0.7608
26	0.0005	0.8116	0.0007	0.7549	0.0006	0.7994	0.0007	0.7522
27	0.0005	0.8157	0.0007	0.7583	0.0006	0.8074	0.0007	0.7582
28	0.0005	0.8256	0.0006	0.7820	0.0005	0.8126	0.0008	0.7223
29	0.0005	0.8284	0.0007	0.7534	0.0005	0.8116	0.0007	0.7519
30	0.0005	0.8245	0.0008	0.7339	0.0005	0.8127	0.0007	0.7675
31	0.0005	0.8239	0.0007	0.7680	0.0005	0.8204	0.0008	0.7354
32	0.0005	0.8350	0.0006	0.7872	0.0005	0.8296	0.0007	0.7526
33	0.0005	0.8357	0.0006	0.7862	0.0005	0.8320	0.0006	0.7859
34	0.0005	0.8390	0.0006	0.7915	0.0005	0.8388	0.0007	0.7766
35	0.0004	0.8492	0.0007	0.7678	0.0005	0.8395	0.0007	0.7576
36	0.0004	0.8493	0.0007	0.7723	0.0005	0.8385	0.0006	0.7915
37	0.0004	0.8521	0.0007	0.7810	0.0004	0.8502	0.0006	0.7848
38	0.0004	0.8521	0.0007	0.7737	0.0004	0.8557	0.0006	0.7938
39	0.0005	0.8382	0.0007	0.7810	0.0004	0.8554	0.0006	0.7938
40	0.0004	0.8463	0.0006	0.7889	0.0004	0.8629	0.0006	0.7991
41	0.0004	0.8690	0.0005	0.8261	0.0004	0.8553	0.0007	0.7843

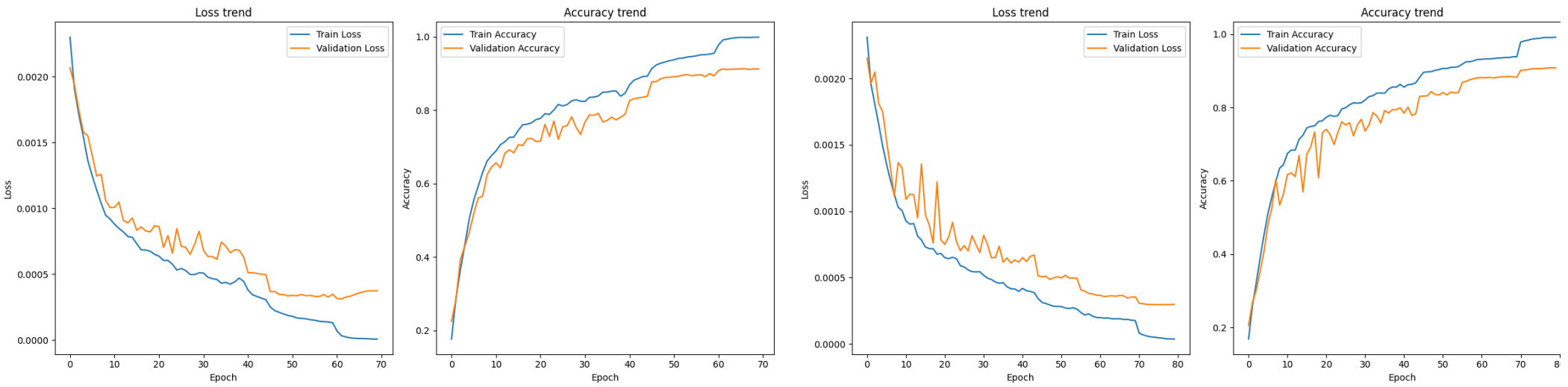
Epoch	Train Loss	Train Acc	Validation Loss	Validation Acc	Train Loss	Train Acc	Validation Loss	Validation Acc
42	0.0003	0.8818	0.0005	0.8313	0.0004	0.8619	0.0006	0.8008
43	0.0003	0.8865	0.0005	0.8335	0.0004	0.8630	0.0007	0.7782
44	0.0003	0.8919	0.0005	0.8354	0.0004	0.8669	0.0007	0.7822
45	0.0003	0.8927	0.0005	0.8383	0.0003	0.8820	0.0005	0.8303
46	0.0003	0.9134	0.0004	0.8773	0.0003	0.8956	0.0005	0.8311
47	0.0002	0.9232	0.0004	0.8780	0.0003	0.8969	0.0005	0.8316
48	0.0002	0.9281	0.0003	0.8856	0.0003	0.8978	0.0005	0.8432
49	0.0002	0.9315	0.0003	0.8889	0.0003	0.9011	0.0005	0.8354
50	0.0002	0.9350	0.0003	0.8899	0.0003	0.9034	0.0005	0.8339
51	0.0002	0.9373	0.0003	0.8914	0.0003	0.9064	0.0005	0.8408
52	0.0002	0.9413	0.0003	0.8924	0.0003	0.9063	0.0005	0.8340
53	0.0002	0.9421	0.0003	0.8958	0.0003	0.9092	0.0005	0.8419
54	0.0002	0.9450	0.0003	0.8969	0.0003	0.9096	0.0005	0.8398
55	0.0002	0.9462	0.0003	0.8939	0.0003	0.9114	0.0005	0.8408
56	0.0001	0.9480	0.0003	0.8957	0.0002	0.9183	0.0004	0.8685
57	0.0001	0.9509	0.0003	0.8967	0.0002	0.9248	0.0004	0.8714
58	0.0001	0.9513	0.0003	0.8917	0.0002	0.9245	0.0004	0.8759
59	0.0001	0.9528	0.0003	0.8997	0.0002	0.9276	0.0004	0.8787
60	0.0001	0.9554	0.0003	0.8936	0.0002	0.9309	0.0004	0.8809
61	0.0001	0.9781	0.0003	0.9081	0.0002	0.9313	0.0004	0.8815
62	0.0000	0.9915	0.0003	0.9125	0.0002	0.9326	0.0004	0.8806
63	0.0000	0.9937	0.0003	0.9107	0.0002	0.9322	0.0004	0.8822
64	0.0000	0.9962	0.0003	0.9120	0.0002	0.9333	0.0004	0.8802
65	0.0000	0.9976	0.0003	0.9122	0.0002	0.9348	0.0004	0.8824
66	0.0000	0.9984	0.0004	0.9126	0.0002	0.9352	0.0004	0.8831
67	0.0000	0.9982	0.0004	0.9130	0.0002	0.9362	0.0004	0.8838
68	0.0000	0.9980	0.0004	0.9109	0.0002	0.9362	0.0003	0.8846
69	0.0000	0.9988	0.0004	0.9127	0.0002	0.9382	0.0004	0.8833
70	0.0000	0.9991	0.0004	0.9123	0.0002	0.9383	0.0004	0.8821
71					0.0001	0.9785	0.0003	0.9012
72					0.0001	0.9813	0.0003	0.9020
73					0.0001	0.9838	0.0003	0.9036
74					0.0001	0.9865	0.0003	0.9054
75					0.0001	0.9879	0.0003	0.9061
76					0.0000	0.9888	0.0003	0.9057
77					0.0000	0.9906	0.0003	0.9066
78					0.0000	0.9909	0.0003	0.9076
79					0.0000	0.9906	0.0003	0.9079

Epoch	Train Loss	Train Acc	Validation Loss	Validation Acc	Train Loss	Train Acc	Validation Loss	Validation Acc
80					0.0000	0.9918	0.0003	0.9077

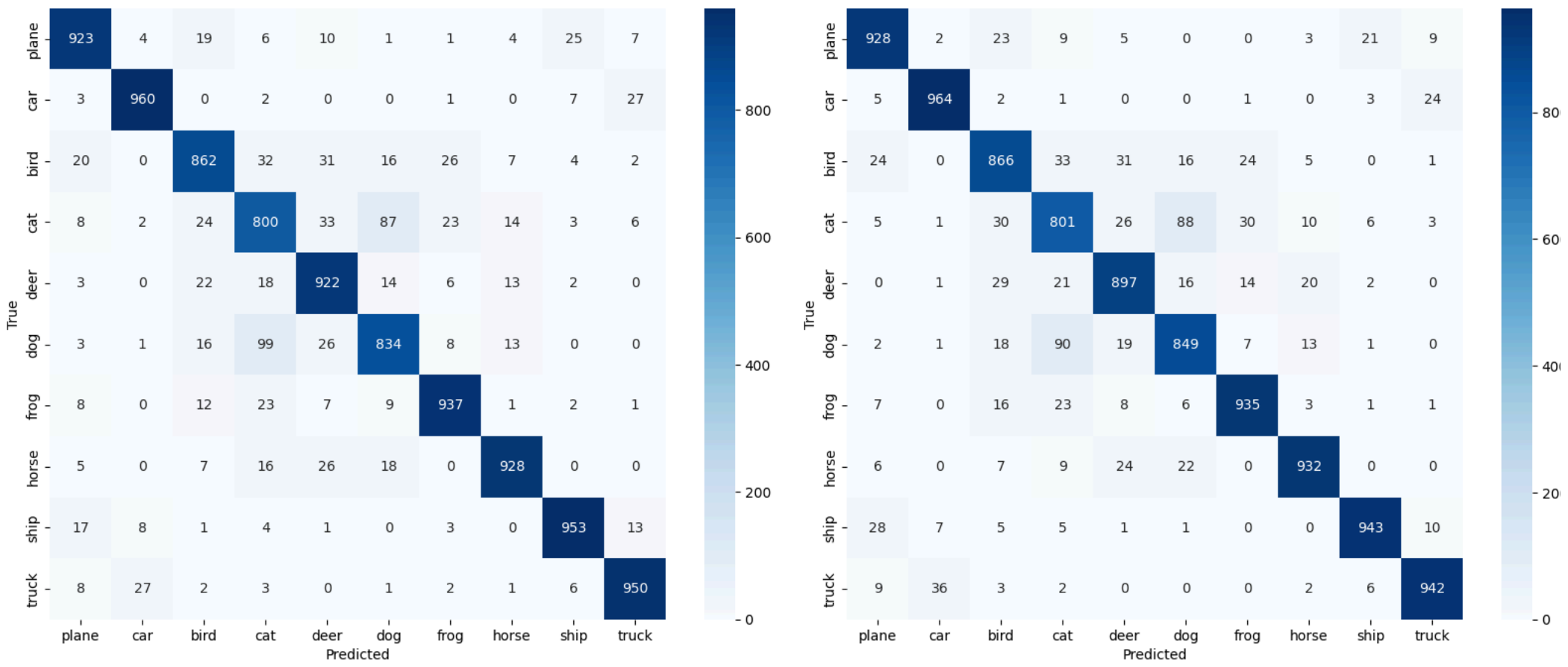
Class	Accuracy	Accuracy
plane	0.9248	0.9152
car	0.9581	0.9526
bird	0.8933	0.8669
cat	0.7976	0.8058
deer	0.8731	0.8872
dog	0.8510	0.8507
frog	0.9305	0.9248
horse	0.9460	0.9433
ship	0.9511	0.9593
truck	0.9443	0.9515
average	**0.9069**	**0.9057**

可以看到在经过上面的诸多优化策略后，模型的分类效果已经稳定较好，两次训练的结果相近

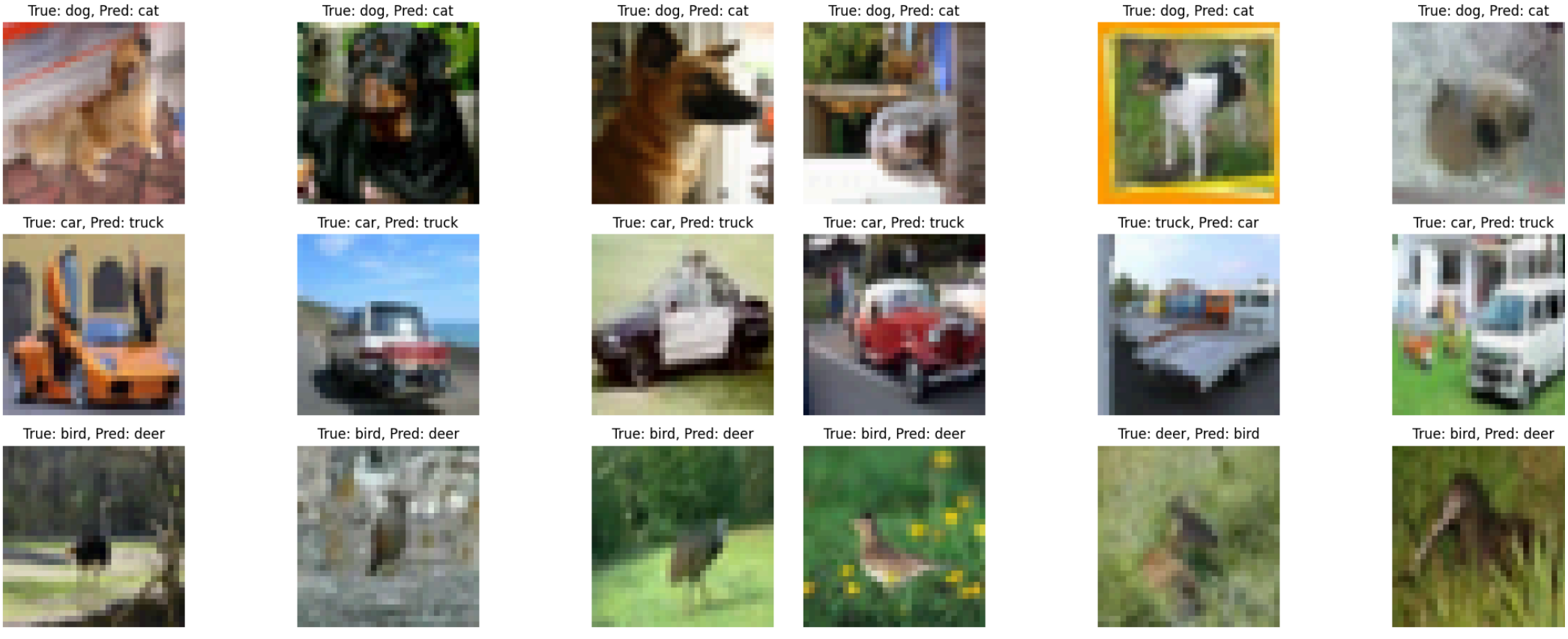
定性分析



从趋势图可以看到训练/验证的损失与准确率都收敛到了较高值，没有出现过拟合或欠拟合现象（左侧的验证损失稍有回升，但可能是偶发的，第二次的训练中就没有这个问题）



从混淆矩阵中可以看出，猫与狗、船与飞机、卡车与小汽车依旧容易混淆，原因与上面的相同（对细节特征的挖掘不足），这其实可以通过增大网络深度等一系列措施来优化，但鉴于已经得到了比较好的结果且本次实验已经写了很久，这里就不再继续微调了。



*迁移学习

在迁移的时候，直接训练会出错（因为最终分类目标数量不同）

```
RuntimeError: Error(s) in loading state_dict for SimpleCNN:
    size mismatch for fc_layers.8.weight:
    copying a param with shape torch.Size([10, 256]) from checkpoint, the shape in current model is torch.Size([100, 256]).
    size mismatch for fc_layers.8.bias:
    copying a param with shape torch.Size([10]) from checkpoint, the shape in current model is torch.Size([100]).
```

这个问题可以通过禁用已有权重最后一层（从上面得知是`fc_layers.8`）并替换为对应数量的输出来解决

```python
# 加载权重时忽略最后一层
pretrained_dict = cifar_10_model
model_dict = model.state_dict()

# 过滤掉最后一层（fc_layers.8）的权重
pretrained_dict = {k: v for k, v in pretrained_dict.items() if k in model_dict and "fc_layers.8" not in k}

# 更新模型的权重
model_dict.update(pretrained_dict)
model.load_state_dict(model_dict)

# 替换最后一层为适应 CIFAR-100 的分类层
model.fc_layers[8] = nn.Linear(256, 100)  # 输出改为 100 类
model = model.to(device)
```

随后在其它条件基本不变的情况下切换数据集为`CIFAR-100`并进行训练与测试

定量分析

训练与验证

Epoch	Train Loss	Train Acc	Validation Loss	Validation Acc
1	0.0038	0.1167	0.0032	0.1933
2	0.0030	0.2205	0.0029	0.2582
3	0.0028	0.2825	0.0026	0.3113
4	0.0026	0.3222	0.0025	0.3324
5	0.0025	0.3474	0.0024	0.3523
6	0.0024	0.3713	0.0023	0.3803
7	0.0023	0.3937	0.0023	0.3841

Epoch	Train Loss	Train Acc	Validation Loss	Validation Acc
8	0.0022	0.4103	0.0022	0.4091
9	0.0021	0.4309	0.0022	0.4220
10	0.0020	0.4435	0.0022	0.4288
11	0.0020	0.4610	0.0022	0.4265
12	0.0020	0.4669	0.0021	0.4428
13	0.0019	0.4808	0.0021	0.4522
14	0.0018	0.4929	0.0020	0.4533
15	0.0018	0.5051	0.0020	0.4597
16	0.0018	0.5154	0.0020	0.4688
17	0.0017	0.5256	0.0020	0.4781
18	0.0017	0.5366	0.0020	0.4754
19	0.0016	0.5425	0.0020	0.4870
20	0.0016	0.5525	0.0020	0.4894
21	0.0016	0.5552	0.0020	0.4866
22	0.0016	0.5645	0.0020	0.4796
23	0.0015	0.5723	0.0020	0.4831
24	0.0015	0.5826	0.0019	0.4937
25	0.0015	0.5916	0.0019	0.5006
26	0.0015	0.5918	0.0019	0.5003
27	0.0015	0.6011	0.0019	0.5013
28	0.0014	0.6070	0.0019	0.4993
29	0.0014	0.6199	0.0019	0.5105
30	0.0014	0.6176	0.0019	0.5086
31	0.0013	0.6281	0.0019	0.5043
32	0.0013	0.6270	0.0019	0.5038
33	0.0013	0.6383	0.0019	0.5147
34	0.0013	0.6393	0.0020	0.5026
35	0.0013	0.6429	0.0019	0.5061
36	0.0012	0.6645	0.0018	0.5248
37	0.0012	0.6725	0.0018	0.5243
38	0.0012	0.6753	0.0018	0.5266
39	0.0012	0.6767	0.0018	0.5239
40	0.0012	0.6793	0.0018	0.5277
41	0.0011	0.6817	0.0018	0.5230
42	0.0011	0.6833	0.0018	0.5284
43	0.0011	0.6842	0.0018	0.5394
44	0.0011	0.6878	0.0018	0.5328
45	0.0011	0.6888	0.0018	0.5317

Epoch	Train Loss	Train Acc	Validation Loss	Validation Acc
46	0.0011	0.6886	0.0018	0.5268
47	0.0011	0.6892	0.0018	0.5320
48	0.0011	0.6924	0.0018	0.5318
49	0.0011	0.6897	0.0018	0.5324
50	0.0011	0.6970	0.0018	0.5290
51	0.0011	0.6972	0.0018	0.5326
52	0.0011	0.6979	0.0018	0.5274
53	0.0011	0.6953	0.0018	0.5302
54	0.0011	0.6943	0.0018	0.5378
55	0.0011	0.6963	0.0019	0.5314
56	0.0012	0.6603	0.0016	0.5708
57	0.0011	0.6628	0.0016	0.5737
58	0.0011	0.6654	0.0016	0.5767
59	0.0011	0.6711	0.0016	0.5793
60	0.0011	0.6699	0.0016	0.5804
61	0.0011	0.6705	0.0016	0.5800
62	0.0011	0.6753	0.0016	0.5818
63	0.0011	0.6767	0.0015	0.5860
64	0.0011	0.6778	0.0016	0.5750
65	0.0011	0.6777	0.0016	0.5813
66	0.0011	0.6805	0.0015	0.5839
67	0.0011	0.6839	0.0016	0.5884
68	0.0011	0.6819	0.0015	0.5827
69	0.0011	0.6848	0.0015	0.5906
70	0.0011	0.6864	0.0015	0.5869
71	0.0006	0.8370	0.0014	0.6220
72	0.0006	0.8439	0.0014	0.6231
73	0.0005	0.8476	0.0014	0.6290
74	0.0005	0.8497	0.0014	0.6275
75	0.0005	0.8533	0.0014	0.6325
76	0.0005	0.8581	0.0014	0.6293
77	0.0005	0.8565	0.0014	0.6326
78	0.0005	0.8619	0.0014	0.6331
79	0.0005	0.8638	0.0014	0.6331
80	0.0005	0.8627	0.0014	0.6317
81	0.0005	0.8671	0.0013	0.6341
82	0.0005	0.8669	0.0013	0.6336
83	0.0005	0.8713	0.0013	0.6359

Epoch	Train Loss	Train Acc	Validation Loss	Validation Acc
84	0.0005	0.8729	0.0013	0.6346
85	0.0005	0.8747	0.0013	0.6369
86	0.0004	0.8754	0.0013	0.6351
87	0.0004	0.8772	0.0013	0.6356
88	0.0004	0.8757	0.0013	0.6360
89	0.0004	0.8773	0.0013	0.6366
90	0.0004	0.8800	0.0013	0.6386

测试

如果冻结原有权重的卷积层，分类准确率大概会收敛到`50%`，之后由于对细节的捕捉能力不足会出现欠拟合

```
Epoch 31/200
Train Loss: 0.0022, Acc: 0.4303 | validation Loss: 0.0023, Acc: 0.4130
Epoch 32/200
Train Loss: 0.0022, Acc: 0.4320 | validation Loss: 0.0022, Acc: 0.4160
Epoch 33/200
Train Loss: 0.0021, Acc: 0.4392 | validation Loss: 0.0023, Acc: 0.4128
Epoch 34/200
Train Loss: 0.0021, Acc: 0.4388 | validation Loss: 0.0022, Acc: 0.4255
Epoch 35/200
Train Loss: 0.0021, Acc: 0.4462 | validation Loss: 0.0022, Acc: 0.4173
Epoch 36/200
Train Loss: 0.0021, Acc: 0.4511 | validation Loss: 0.0022, Acc: 0.4181
Epoch 37/200
Train Loss: 0.0021, Acc: 0.4534 | validation Loss: 0.0022, Acc: 0.4246
Epoch 38/200
Train Loss: 0.0021, Acc: 0.4568 | validation Loss: 0.0022, Acc: 0.4213
Epoch 39/200
Train Loss: 0.0021, Acc: 0.4568 | validation Loss: 0.0022, Acc: 0.4279
Epoch 40/200
Train Loss: 0.0021, Acc: 0.4587 | validation Loss: 0.0022, Acc: 0.4258
Epoch 41/200
Train Loss: 0.0020, Acc: 0.4650 | validation Loss: 0.0022, Acc: 0.4313
Epoch 42/200
Train Loss: 0.0020, Acc: 0.4697 | validation Loss: 0.0022, Acc: 0.4357
Epoch 43/200
Train Loss: 0.0020, Acc: 0.4692 | validation Loss: 0.0022, Acc: 0.4259
Epoch 44/200
Train Loss: 0.0020, Acc: 0.4693 | validation Loss: 0.0022, Acc: 0.4282
Epoch 45/200
Train Loss: 0.0020, Acc: 0.4751 | validation Loss: 0.0022, Acc: 0.4284
Epoch 46/200
Train Loss: 0.0020, Acc: 0.4773 | validation Loss: 0.0022, Acc: 0.4303
Epoch 47/200
Train Loss: 0.0020, Acc: 0.4778 | validation Loss: 0.0022, Acc: 0.4232
Epoch 48/200
Train Loss: 0.0020, Acc: 0.4844 | validation Loss: 0.0022, Acc: 0.4322
Epoch 49/200
Train Loss: 0.0019, Acc: 0.4868 | validation Loss: 0.0022, Acc: 0.4401
Epoch 50/200
Train Loss: 0.0019, Acc: 0.4850 | validation Loss: 0.0022, Acc: 0.4334
Epoch 51/200
Train Loss: 0.0019, Acc: 0.4870 | validation Loss: 0.0022, Acc: 0.4362
Epoch 52/200
Train Loss: 0.0019, Acc: 0.4922 | validation Loss: 0.0022, Acc: 0.4347
```

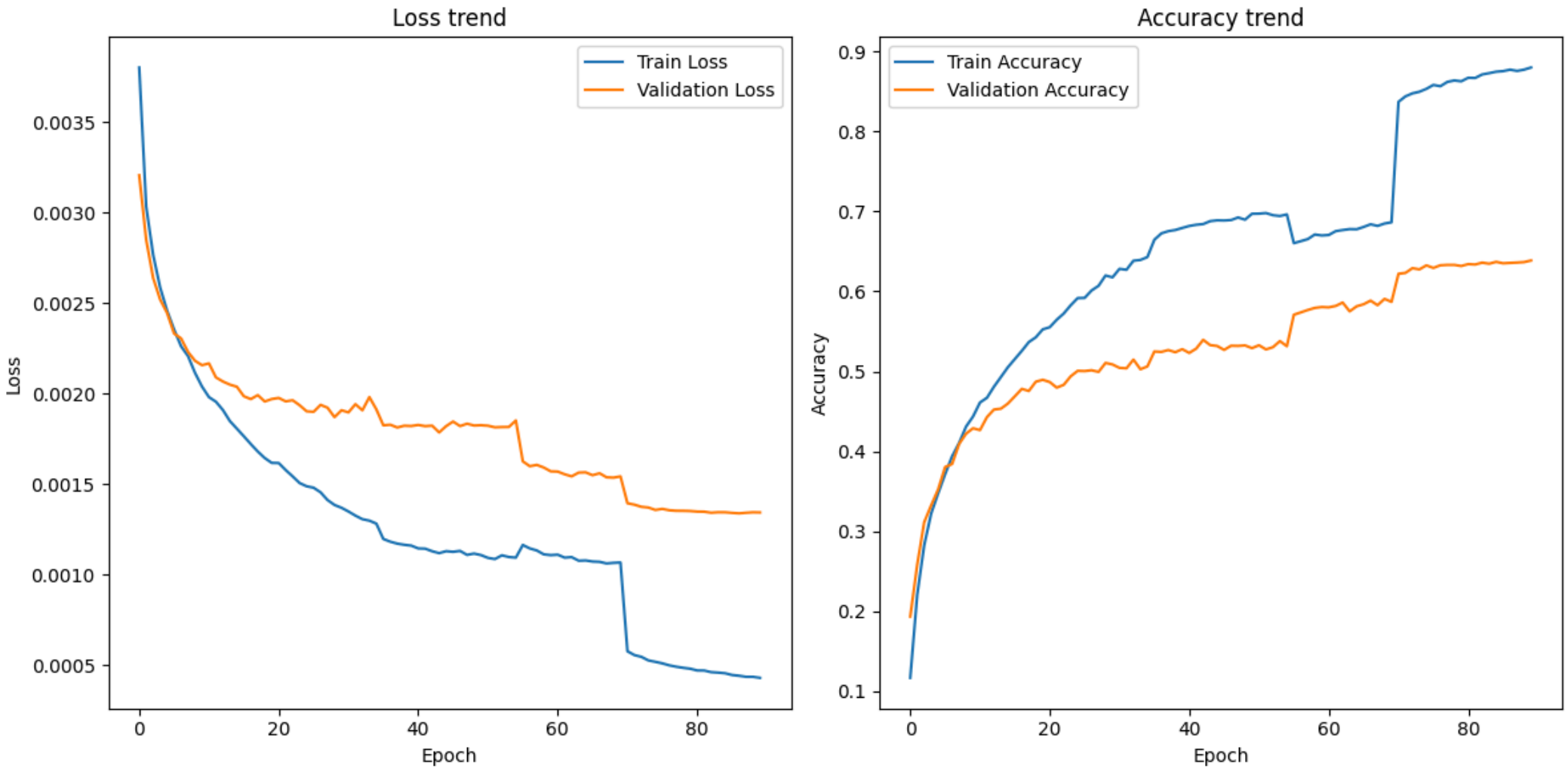
如果不冻结卷积层，最终整体的分类准确率收敛到`63.88%`，若加大训练轮次可能收敛到`65%`（这其中对于动物的区分能力普遍较弱，针对`CIFAR-10`设计的模型结构相对来说可能还是太简单了，捕捉不到那么细致的特征）

类别	准确率	类别	准确率	类别	准确率	类别	准确率
otter	0.2650	lizard	0.3143	girl	0.3176	seal	0.3600
mouse	0.3964	woman	0.3980	squirrel	0.4158	bear	0.4167
turtle	0.4217	lobster	0.4224	shark	0.4250	rabbit	0.4272

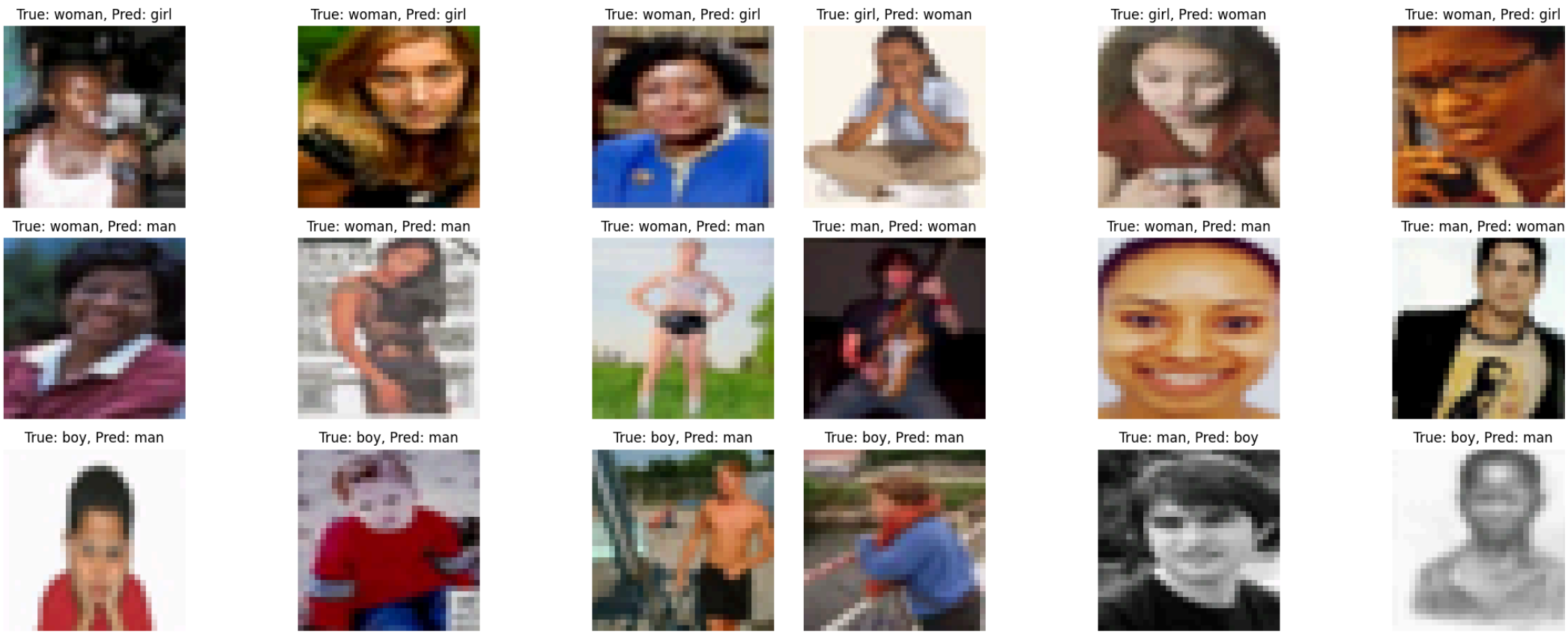
类别	准确率	类别	准确率	类别	准确率	类别	准确率
beaver	0.4393	crocodile	0.4444	man	0.4444	shrew	0.4486
possum	0.4536	baby	0.4667	boy	0.4940	willow_tree	0.5200
fox	0.5333	tulip	0.5393	snail	0.5437	caterpillar	0.5487
ray	0.5488	snake	0.5657	forest	0.5684	couch	0.5714
dolphin	0.5727	kangaroo	0.5825	lamp	0.5895	whale	0.5895
oak_tree	0.5915	butterfly	0.5918	leopard	0.5926	mushroom	0.5938
camel	0.6061	bowl	0.6145	bed	0.6147	bee	0.6161
tiger	0.6170	streetcar	0.6210	poppy	0.6306	sweet_pepper	0.6316
crab	0.6344	beetle	0.6395	orchid	0.6410	bus	0.6494
elephant	0.6559	flatfish	0.6598	spider	0.6598	worm	0.6606
wolf	0.6700	pine_tree	0.6705	porcupine	0.6744	dinosaur	0.6813
can	0.6822	table	0.6824	trout	0.6893	sea	0.6903
television	0.6957	train	0.6970	mountain	0.7059	raccoon	0.7093
pear	0.7100	house	0.7113	telephone	0.7115	clock	0.7126
bridge	0.7143	rose	0.7195	chimpanzee	0.7196	lion	0.7253
tractor	0.7264	maple_tree	0.7375	cattle	0.7429	plate	0.7526
aquarium_fish	0.7527	cloud	0.7547	hamster	0.7667	cup	0.7670
rocket	0.7700	apple	0.8000	road	0.8087	orange	0.8113
cockroach	0.8132	skunk	0.8182	tank	0.8200	plain	0.8211
motorcycle	0.8241	keyboard	0.8265	palm_tree	0.8316	bottle	0.8333
castle	0.8333	wardrobe	0.8349	pickup_truck	0.8485	bicycle	0.8557
skyscraper	0.8614	sunflower	0.8710	lawn_mower	0.8791	chair	0.8830

定性分析

正如上面所说，禁用卷积层的情况下会导致欠拟合（从少类别区分任务切换到对细节把控要求更高的多类别区分任务）；而如果启用了卷积层，同样由于前面的原因，整体分类准确率会由于动物类的区分能力弱而卡着不动，大致在第 `15` 轮开始验证集分类准确率的提升速度就比不上训练集了（动物分类的瓶颈）；后续的两次阶梯式上升来自数据增强策略的切换，期望通过减少对图像的干扰来让模型学到更细致一些的特征，从结果中看到这确实有一些用处（如果不换的话最终可能会收敛在`50%+`，且依然过拟合）



可以看到对于类别更多、单类数据量更少（`CIFAR-100`数据集每类图片只有`600`张）的情况下，原先模型结构中的缺点被放得更大（对细节的捕捉不足），上面的分类准确率排了升序，准确率较低的基本都是动物；而对于男人、女人、男孩、女孩这种对细节特征要求更为严格的情况，模型的区分也相对的更为困难（而且有些图确实很难区分，比如最左上角的那张图）。对应的，如果只需要区分出这些都是人，那模型的区分能力就很好了（比如下面即使预测错误，预测值也还是在人的范围）



从上面的数据与分析中可以得知模型的特征提取能力不足，那相应的就可以采用更深的卷积层（例如四组、五组，或更多，具体需要微调），还可以引入注意力机制捕捉更多的细节特征（比如`SEBlock`，本身实现挺简单的，可以放在每组卷积层之后）。此外，从上面的分类准确率的值域来看，极差非常大，可以按照上面的分类结果在预处理的时候给不同的类设置不同的权重（当前准确率越低，权值就可以越大，整体应该可以呈一个`log`函数的变化趋势？），设置好之后传给交叉熵损失函数就可以了

不过仅出于验证模型泛化性的目的，到这一步就可以了，由于我们之前就已经分析过模型对于动物类的区分能力相对较弱（只不过在大数据量与少类别情况下表现不明显），迁移到`CIFAR-100`后出现的现象只不过是放大后的版本，在预期内；同时其对于非动物类的分类表现很好，只不过由于动物类的准确率一直较低，导致非动物类的区分能力在长时间的训练过程中被丢弃了一部分（这一结论的得出是由于测试发现，训练十几轮之后一些非动物类的分类准确率可以达到`90%+`，其中`road`可以达到`100%`）。

综上，模型可以做到迁移部分知识到新任务中，但是在细粒度特征的提取方面存在明显不足（不过在`CIFAR-10`中已经够用）。可以认为模型具有一定的泛化能力：如果迁移到类似的不同任务上可以表现出较好的效果；但如果遇到更大型、对细粒度区分要求更为严格的任务表现就会较差。

结论

综上所述，模型在目标数据集上的整体性能表现已经很好（`90.69%`），但在对于细节特征的把控能力相对较为薄弱，在将模型权重迁移到对特征细粒度区分要求更高的`CIFAR-100`数据集上后这一现象更为明显（细粒度高要求高的类与细粒度要求低的类，分类准确率的极差非常大）。

而为了提取到更多的细节特征，接下来的改进方向可以为：加深卷积层、引入注意力模块`SEBlock`。同时为了避免上述改动导致的过拟合现象，可以将最大池化改为平均池化，同时在卷积层中也考虑使用`Dropout`；针对当前分类准确率较低的类别适当调高其权重；另外，目前的数据增强策略和训练过程中的动态切换策略感觉上已经很不错，但进行上述修改后可能还需进行一些微调。

本次实验写了很长时间，自己动手从基础`CNN`结构开始搭建了整个训练框架，并最终微调出了表现较好的一版模型结构与训练策略，收获还是很大的，而且文档的描述也很清晰、提供了多种可选方向，基本上照着做下来就能达到不错的效果。唯一想说的一点是既然要求提交时附带代码文件，那“实验报告中包含可复现的完整代码”感觉不是很必要（而且“基础实验步骤”一节中也已经要求贴出关键代码片段并解析了），类似的训练过程这些可以要求放到单独的文件里提交（还是比较长的），不过这个问题不大，总体上给个好评。

完整代码展示

按文档要求，下面展示完整的实验代码，具体也可参考附件中的`*.ipynb`

```python
import torch
import random
import numpy as np
import seaborn as sns
import torch.nn as nn
import matplotlib.pyplot as plt
import torch.nn.functional as F
from torchvision import datasets, transforms
from torch.utils.data import DataLoader, Subset
from torchvision.transforms import AutoAugment, AutoAugmentPolicy
from sklearn.metrics import confusion_matrix, classification_report
# 数据增强
class DataSetOpt:
    def __init__(self, add_gaussian_noise=True, mean=0.0, std=0.03, use_cutout=True, num_holes=1, size=4):
        # 高斯噪声
        self.add_gaussian_noise = add_gaussian_noise
        self.mean = mean
        self.std = std
        # Cutout
        self.use_cutout = use_cutout
        self.num_holes = num_holes
        self.size = size

    def __call__(self, img):
        if self.add_gaussian_noise:
            img = img + torch.randn(img.size()) * self.std + self.mean
        if self.use_cutout:
            h, w = img.size(1), img.size(2)
            for _ in range(self.num_holes):
                y = random.randint(0, h)
                x = random.randint(0, w)
                img[
                    :,
                    max(0, y - self.size // 2) : min(h, y + self.size // 2),
                    max(0, x - self.size // 2) : min(w, x + self.size // 2),
                ] = 0

        return img
torch.manual_seed(42)

transform = transforms.Compose(
    [
        transforms.ToTensor(),
        transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2470, 0.2435, 0.2616)),
    ]
)
# 先几何变换再裁剪比较适合分类任务，不然容易丢失全局特征
# 强数据增强（前期）
# 数据增强还是不太适合乱搞，之前本来都练出82%的准确率了，乱搞一通掉到70%
strong_transforms = transforms.Compose(
    [
        AutoAugment(policy=AutoAugmentPolicy.CIFAR10),  # 包含注释掉的那些了
        transforms.RandomHorizontalFlip(p=0.5),  # 随机水平翻转
        # transforms.RandomAffine(
        #     degrees=0, translate=(0.1, 0.1), scale=(0.8, 1.2)
        # ),  # 平移和缩放
        # transforms.RandomCrop(24),  # 随机裁剪为 24x24
        # transforms.Resize(32),  # 恢复到 32x32，确保后面输入正常
        # transforms.ColorJitter(
        #     brightness=0.2, contrast=0.2, saturation=0.2
        # ),  # 颜色扰动
        transforms.ToTensor(),
        DataSetOpt(std=0.03, use_cutout=True),  # 高斯噪声和 Cutout
        transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2470, 0.2435, 0.2616)),
    ]
)
# 弱数据增强（后期）
```

```python
weak_transforms = transforms.Compose(
    [
        transforms.RandomHorizontalFlip(p=0.5),  # 随机水平翻转
        transforms.RandomCrop(32, padding=4),  # 随机裁剪
        transforms.ToTensor(),
        transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2470, 0.2435, 0.2616)),
    ]
)

# 加载完整数据集
train_dataset = datasets.CIFAR10(
    root="./data", train=True, download=True, transform=strong_transforms
)
test_dataset = datasets.CIFAR10(
    root="./data",
    train=False,
    download=True,
    transform=transform
)



# 随机抽取数据
def get_subset(dataset, ratio=1):
    indices = np.random.choice(
        len(dataset), size=int(len(dataset) * ratio), replace=False
    )
    return Subset(dataset, indices)

print("原始训练集大小: ", len(train_dataset))
print("原始测试集大小: ", len(test_dataset))
train_subset = get_subset(train_dataset)
test_subset = get_subset(test_dataset)
imgSize = train_subset[0][0].shape[1]
print("图像尺寸" + str(imgSize) + "x" + str(imgSize))
# 划分验证集
validation_size = int(0.2 * len(train_subset))
train_size = len(train_subset) - validation_size
train_subset, validation_subset = torch.utils.data.random_split(
    train_subset, [train_size, validation_size]
)

DL_batch_size = 1024
train_loader = DataLoader(train_subset, batch_size=DL_batch_size, shuffle=True, num_workers=2, pin_memory=True)
validation_loader = DataLoader(validation_subset, batch_size=DL_batch_size, num_workers=2, pin_memory=True)
test_loader = DataLoader(test_subset, batch_size=DL_batch_size, num_workers=2, pin_memory=True)

print(
    f"训练样本数: {len(train_subset)}, 验证样本数: {len(validation_subset)}, 测试样本数: {len(test_subset)}"
)
cifar10_classes = [
    "plane",
    "car",
    "bird",
    "cat",
    "deer",
    "dog",
    "frog",
    "horse",
    "ship",
    "truck",
]

# 随机选择 5 张图像
show_size = 5
show_imgs = random.sample(range(len(train_dataset)), show_size)

_, axes = plt.subplots(2, show_size, figsize=(12, 6))
```

```python
for i, idx in enumerate(show_imgs):
    # 数据增强后的图需要变换回来才能正常显示
    opt_img, _ = train_dataset[idx]
    opt_img = opt_img.numpy().transpose((1, 2, 0))

    # 反归一化（不然颜色会变得非常奇怪非常深，第一次跑的时候还以为我的数据增强出了什么问题）
    mean = np.array([0.4914, 0.4822, 0.4465])
    std = np.array([0.2470, 0.2435, 0.2616])
    opt_img = std * opt_img + mean
    opt_img = np.clip(opt_img, 0, 1)

    axes[0, i].imshow(train_dataset.data[idx])
    axes[0, i].set_title(f"{cifar10_classes[train_dataset.targets[idx]]}")
    axes[0, i].axis("off")

    axes[1, i].imshow(opt_img)
    axes[1, i].set_title(f"{cifar10_classes[train_dataset.targets[idx]]}(OPT)")
    axes[1, i].axis("off")


def dataset_info(dataset):
    print(f"数据集大小: {len(dataset)}")

    # 获取图像形状
    img_shape = dataset[0][0].shape
    print(f"通道数: {img_shape[0]}，图像尺寸: {img_shape[1]}x{img_shape[2]}")

    # 提取所有标签
    labels = [dataset[i][1] for i in range(len(dataset))]
    labels, counts = np.unique(labels, return_counts=True)  # 统计每个类别的数量

    print("类别分布:")
    for label, cnt in zip(labels, counts):
        print(f"    {cifar10_classes[label]}: {cnt} 张")

    forAll = torch.stack(
        [dataset[i][0] for i in range(len(dataset))]
    )  # 将所有图像堆叠为一个张量
    forAll = forAll.view(-1, img_shape[0], img_shape[1] * img_shape[2])  # 展平

    print(f"每通道均值: {forAll.mean(dim=(0, 2)).numpy()}")
    print(f"每通道标准差: {forAll.std(dim=(0, 2)).numpy()}")
    print(f"像素值范围: {forAll.min().item()} ~ {forAll.max().item()}")


plt.tight_layout()
plt.show()

print("训练集统计信息:")
dataset_info(train_dataset)

print("\n测试集统计信息:")
dataset_info(test_dataset)
class BasicCNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv_layers = nn.Sequential(
            nn.Conv2d(in_channels=3, out_channels=32, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2, 2),  # 32/2 , 32/2
            nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2, 2),  # 16/2 , 16/2
            nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2, 2),  # 8/2 , 8/2   -> 4x4
        )
        self.fc_layers = nn.Sequential(
```

```
            nn.Flatten(),
            nn.Linear(128 * 4 * 4, 512),
            nn.ReLU(),
            nn.Dropout(0.5),
            nn.Linear(512, 10),
        )

    def forward(self, x):
        x = self.conv_layers(x)
        x = self.fc_layers(x)
        return x
class SimpleCNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv_layers = nn.Sequential(
            # 第一组卷积层
            nn.Conv2d(in_channels=3, out_channels=128, kernel_size=3, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(),
            nn.Conv2d(in_channels=128, out_channels=128, kernel_size=3, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(),
            nn.MaxPool2d(2, 2),  # 池化    16x16
            # 第二组卷积层
            nn.Conv2d(in_channels=128, out_channels=256, kernel_size=3, padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU(),
            nn.Conv2d(in_channels=256, out_channels=256, kernel_size=3, padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU(),
            nn.MaxPool2d(2, 2),  # 池化    8x8
            # 第三组卷积层
            nn.Conv2d(in_channels=256, out_channels=512, kernel_size=3, padding=1),
            nn.BatchNorm2d(512),
            nn.ReLU(),
            nn.Conv2d(in_channels=512, out_channels=512, kernel_size=3, padding=1),
            nn.BatchNorm2d(512),
            nn.ReLU(),
            nn.MaxPool2d(2, 2),  # 池化    4x4
        )
        self.fc_layers = nn.Sequential(
            nn.Flatten(),
            nn.Linear(512 * 4 * 4, 512),
            nn.BatchNorm1d(512),
            nn.ReLU(),
            nn.Dropout(0.5),
            nn.Linear(512, 256),
            nn.BatchNorm1d(256),
            nn.ReLU(),
            nn.Linear(256, 10),
        )

    def forward(self, x):
        x = self.conv_layers(x)
        x = self.fc_layers(x)
        return x
class ModelTrainer:
    def __init__(self, model, train_loader, validation_loader, epochs=200, lr=0.01):
        self.device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
        self.model = model.to(self.device)
        self.train_loader = train_loader
        self.validation_loader = validation_loader
        self.epochs = epochs
        self.lr = lr

        self.loss_func = nn.CrossEntropyLoss()  # 交叉熵损失函数适合分类  标签平滑的效果好像不是那么好
        self.optim = torch.optim.Adam(self.model.parameters(), lr=self.lr)  # 加上正则化前期震荡有点大，需要微调，但是现在效果已经很好了就不加了
        self.scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(
```

```
            self.optim, "min", patience=5
        )
        self.max_acc = 0.0

        # 拿来画图的
        self.train_losses, self.validation_losses = [], []
        self.train_accs, self.validation_accs = [], []

    def train(self):
        for epoch in range(self.epochs):
            # 动态切换数据增强策略
            if epoch == 45:  # 换到弱增强，这时候泛化能力已经不错了，希望能多学一点
                train_dataset.transform = weak_transforms
            elif epoch == 60:   # 关闭增强
                train_dataset.transform = transform

            # 训练
            train_loss, train_acc = self.train_part()

            # 验证
            validation_loss, validation_acc = self.validation_part()

            # 调整学习率
            self.scheduler.step(validation_loss)

            # 保存最佳模型
            if validation_acc > self.max_acc:
                self.max_acc = validation_acc
                torch.save(self.model.state_dict(), "CIFAR10_CNN1.pth")

            print(f"Epoch {epoch+1}/{self.epochs}")
            print(
                f"Train Loss: {train_loss:.4f}, Acc: {train_acc:.4f} | validation Loss: {validation_loss:.4f}, Acc: {validation_acc:.4f}"
            )

        return self.train_losses, self.validation_losses, self.train_accs, self.validation_accs

    def train_part(self):
        self.model.train()
        loss_sum = 0.0
        correct_cnt = 0
        total = 0
        for inputs, labels in self.train_loader:
            inputs, labels = inputs.to(self.device), labels.to(self.device)
            self.optim.zero_grad()                  # 梯度清零
            outputs = self.model(inputs)            # 调用上面搭建的前向传播
            loss = self.loss_func(outputs, labels)  # 计算损失
            loss.backward()                         # 反向传播，计算梯度
            self.optim.step()                       # 更新参数

            loss_sum += loss.item()
            total += labels.size(0)     # 加载多少算多少
            _, predicted = torch.max(outputs, 1)    # 新版torch可以直接传，不需要带.data
            correct_cnt += (predicted == labels).sum().item()

        train_loss = loss_sum / total
        train_acc = correct_cnt / total
        self.train_losses.append(train_loss)
        self.train_accs.append(train_acc)
        return train_loss, train_acc

    def validation_part(self):
        self.model.eval()
        validation_loss = 0.0
        correct_cnt = 0
        total = 0
        with torch.no_grad():  # 验证阶段不需要计算梯度
            for inputs, labels in self.validation_loader:
```

```
                inputs, labels = inputs.to(self.device), labels.to(self.device)
                outputs = self.model(inputs)
                validation_loss += self.loss_func(outputs, labels).item()

                total += labels.size(0)
                _, predicted = torch.max(outputs, 1)
                correct_cnt += (predicted == labels).sum().item()

        validation_loss = validation_loss / total
        validation_acc = correct_cnt / total
        self.validation_losses.append(validation_loss)
        self.validation_accs.append(validation_acc)
        return validation_loss, validation_acc

print("单元格加载成功")
def evaluate_model(model, test_loader, model_path="CIFAR10_CNN1.pth"):
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    model.load_state_dict(torch.load(model_path, map_location=device)) # 在colab上拿T4练的，不加上后面那个本地跑不了
    # model = torch.load(model_path, map_location=device) # 加载更快（不需要重新实例化模型），但不太安全
    model.to(device)
    model.eval()

    test_labels = []
    test_preds = []
    with torch.no_grad():
        correct_cnt = 0
        total = 0
        for inputs, labels in test_loader:
            inputs, labels = inputs.to(device), labels.to(device)
            total += labels.size(0)

            _, predicted = torch.max(model(inputs), 1)
            correct_cnt += (predicted == labels).sum().item()

            test_labels.extend(labels.cpu().numpy())
            test_preds.extend(predicted.cpu().numpy())

    print(f"Test Accuracy: {correct_cnt / total:.4f}")

    # 计算每个分类的准确率
    rp = classification_report(
        test_labels,
        test_preds,
        target_names=[str(i) for i in range(10)],
        output_dict=True,
    )
    for i in range(10):
        print(f"Accuracy for class {cifar10_classes[i]}: {rp[str(i)]['precision']:.4f}")
    plt.figure(figsize=(10, 8))
    sns.heatmap(
        confusion_matrix(test_labels, test_preds),
        annot=True,
        fmt="d",
        cmap="Blues",
        xticklabels=cifar10_classes,
        yticklabels=cifar10_classes,
    )
    plt.xlabel("Predicted")
    plt.ylabel("True")
    plt.show()


print("单元格加载成功")
# 保存预测出错图片信息
error_images = {0: [], 1: [], 2: []}  # 0: {3,5}, 1: {1,9}, 2: {2,4}
error_labels = {0: [], 1: [], 2: []}  # 真实标签
error_preds = {0: [], 1: [], 2: []}  # 预测标签
```

```python
# 错误率比较高的三组
label_sets = [{3, 5}, {1, 9}, {2, 4}]

def evaluate_model(model, test_loader, model_path="CIFAR10_CNN.pth"):
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    model.load_state_dict(torch.load(model_path, map_location=device))  # 在colab上拿T4练的，不加上后面那个本地跑不了
    # model = torch.load(model_path, map_location=device) # 加载更快（不需要重新实例化模型），但不太安全
    model.to(device)
    model.eval()

    test_labels = []
    test_preds = []
    with torch.no_grad():
        correct_cnt = 0
        total = 0
        for inputs, labels in test_loader:
            inputs, labels = inputs.to(device), labels.to(device)
            total += labels.size(0)

            _, predicted = torch.max(model(inputs), 1)
            correct_cnt += (predicted == labels).sum().item()

            test_labels.extend(labels.cpu().numpy())
            test_preds.extend(predicted.cpu().numpy())

            # 展示错误分类
            if (
                len(error_images[0]) == 3
                and len(error_images[1]) == 3
                and len(error_images[2]) == 3
            ):
                continue
            for i in range(len(labels)):
                true_label = labels[i].item()
                pred_label = predicted[i].item()
                if true_label != pred_label:
                    for idx, label_set in enumerate(label_sets):
                        if true_label in label_set and pred_label in label_set:
                            if len(error_images[idx]) < 3:  # 展示三张
                                error_images[idx].append(inputs[i].cpu())
                                error_labels[idx].append(true_label)
                                error_preds[idx].append(pred_label)
                            else:
                                break
                if (
                    len(error_images[0]) == 3
                    and len(error_images[1]) == 3
                    and len(error_images[2]) == 3
                ):
                    break
    print(f"Test Accuracy: {correct_cnt / total:.4f}")

    # 计算每个分类的准确率
    rp = classification_report(
        test_labels,
        test_preds,
        target_names=[str(i) for i in range(10)],
        output_dict=True,
    )
    for i in range(10):
        print(f"Accuracy for class {cifar10_classes[i]}: {rp[str(i)]['precision']:.4f}")
    plt.figure(figsize=(10, 8))
    sns.heatmap(
        confusion_matrix(test_labels, test_preds),
        annot=True,
        fmt="d",
        cmap="Blues",
        xticklabels=cifar10_classes,
```

```
        yticklabels=cifar10_classes,
    )
    plt.xlabel("Predicted")
    plt.ylabel("True")
    plt.show()

    mean = np.array([0.4914, 0.4822, 0.4465])
    std = np.array([0.2470, 0.2435, 0.2616])
    _, axes = plt.subplots(3, 3, figsize=(12, 8))
    for i in range(3):  # 遍历每类
        for j in range(3):  # 每类展示3张
            img = error_images[i][j].permute(1, 2, 0).numpy()
            img = std * img + mean
            img = np.clip(img, 0, 1)

            axes[i, j].imshow(img)
            axes[i, j].set_title(
                f"True: {cifar10_classes[error_labels[i][j]]}, Pred: {cifar10_classes[error_preds[i][j]]}"
            )
            axes[i, j].axis("off")
    plt.tight_layout()
    plt.show()
```

```
model = SimpleCNN()
train_losses, validation_losses, train_accs, validation_accs = ModelTrainer(
    model, train_loader, validation_loader, epochs=70, lr=0.03
).train()

# 训练/验证损失
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.plot(train_losses, label="Train Loss")
plt.plot(validation_losses, label="Validation Loss")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.title("Loss trend")
plt.legend()

# 训练/验证准确率
plt.subplot(1, 2, 2)
plt.plot(train_accs, label="Train Accuracy")
plt.plot(validation_accs, label="Validation Accuracy")
plt.xlabel("Epoch")
plt.ylabel("Accuracy")
plt.title("Accuracy trend")
plt.legend()
```

```
plt.tight_layout()
plt.show()

evaluate_model(model, test_loader)
```