使用CNN对CIFAR-10图像集分类

文件总览

```
baseline.py          #原始模型和最终模型的定义
general.ipynb        #训练并测试最终模型
tutorial.pth         #原始模型的网络参数
finalmodel.pth       #最终模型的网络参数
result_confirm.ipynb #快速验收，直接得到实验结果
```

代码结构：主要为general.ipynb

1. 导入实验所需library

```python
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import matplotlib.pyplot as plt
import numpy as np
from torch.utils.data import DataLoader,Subset,ConcatDataset    #数据集的拆分与合并
from torchvision import datasets, transforms
from tqdm import tqdm
from baseline import Net,CIFAR10Model                           #加载模型
from torchsummary import summary                                #展示模型结构
from torch.optim.lr_scheduler import ReduceLROnPlateau          #学习率调整策略
from torchvision.transforms import ToPILImage                   #转换图片的数据格式
```

2. 加载数据

```
torch.manual_seed(42)

#mean=[0.4914, 0.4822, 0.4465]
#std=[0.2470, 0.2435, 0.2616]
#在模型优化环节尝试使用训练集数据的均值和方差进行归一化，结果一方面对于模型优化并无实际的正面影响，另一方面对于在图像显示环节调用imshow()造成数据越界，最终仍使用[0.5,0.5,0,5]进行归一化
mean=[0.5, 0.5, 0.5]
std=mean

transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize(mean, std)
])

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

modified_labels = np.load('./modified_labels.npy')
train_dataset.targets = modified_labels.tolist()

# 创建DataLoader，这里只创建测试集的Dataloader是因为训练集数据要先进行样本筛选
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("Using device:", device)

def to_device(data, device):
    if isinstance(data, (list,tuple)):
        return [to_device(x, device) for x in data]
    return data.to(device, non_blocking=True)
```

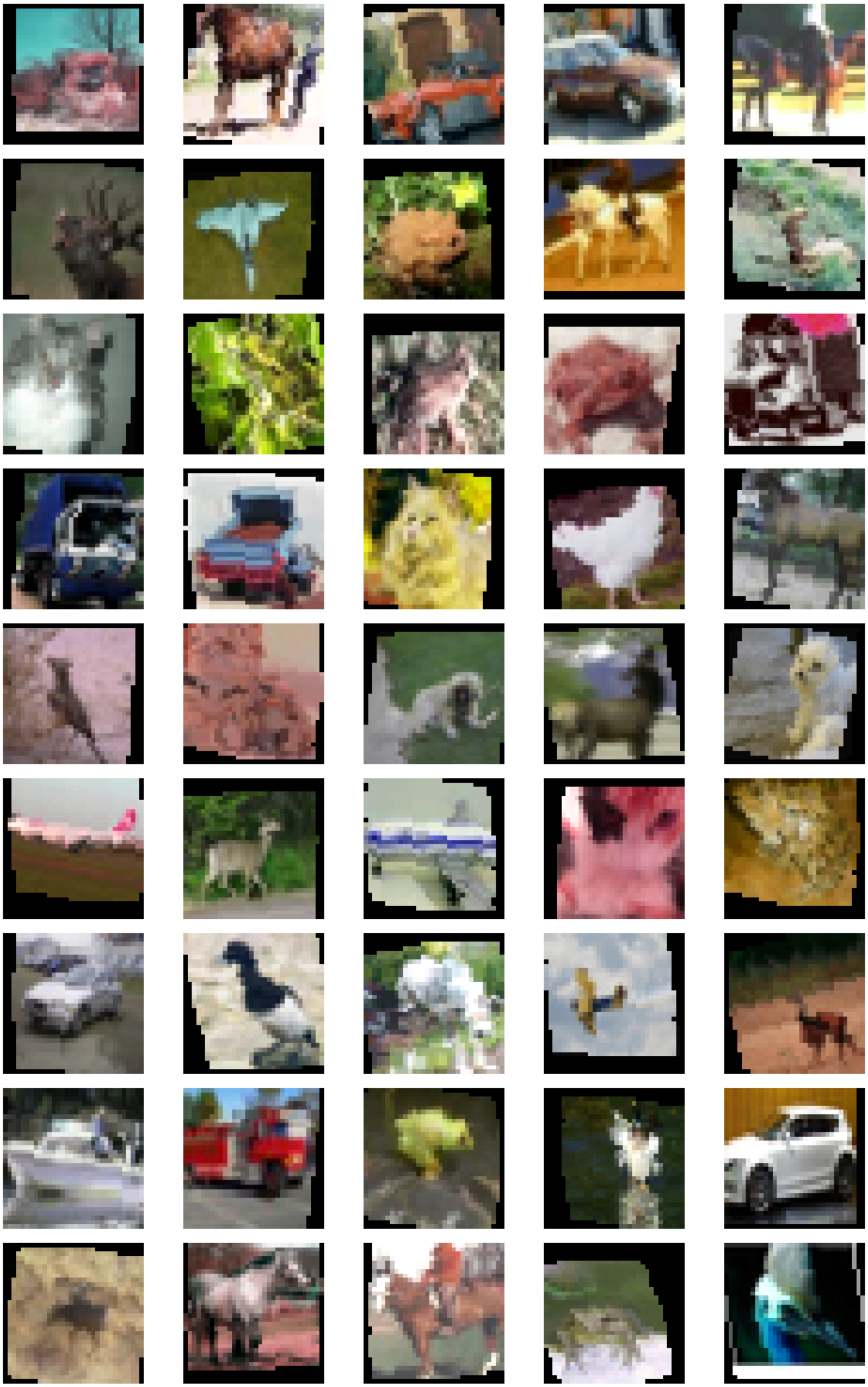
3. 样本筛选

```python
#baseline.py
#原始的基础网络，结构简单，模仿LeNet结构，只含有2个卷积层，将原始网络进行简单训练后用于筛选训练集样本
import torch
import torch.nn as nn
import torch.nn.functional as F

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()

        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool1 = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.pool2 = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self,x):
        x = F.relu(self.pool1(self.conv1(x)))
        x = F.relu(self.pool2(self.conv2(x)))
        x = torch.flatten(x, 1)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)

        return x

model=Net()
print(model)
```

```python
net = Net()
# 加载经过简单训练的原始模型参数。使用发布作业时的train_test.ipynb文件训练，故不再说明和引用
net.load_state_dict(torch.load("tutorial.pth"))

# 创建新的数据子集，包含训练数据集的后50%，这是因为只有后50%含有错误标注的样本
train_dataset_length = len(train_dataset)
start_idx = train_dataset_length // 2
end_idx = train_dataset_length
subset_indices = list(range(start_idx, end_idx))
rough_dataset = Subset(train_dataset, subset_indices)
roughdata_loader = DataLoader(rough_dataset, batch_size=64, shuffle=False)

# 对于待过滤数据集，计算其中每个样本的置信度
net.eval()
all_confidences = []

with torch.no_grad():
    for data in roughdata_loader:
        images, labels = data
        outputs = net(images)
        probabilities = torch.softmax(outputs, dim=1)
        confidences, predictions = torch.max(probabilities, dim=1)
        all_confidences.extend(confidences.cpu().numpy())

all_confidences = np.array(all_confidences)

# 根据设定的阈值过滤数据，置信度小于阈值的视为错误样本/低质量样本被过滤掉
confidence_threshold = 0.8
filtered_indices = np.where(all_confidences >= confidence_threshold)[0]
filtered_dataset = torch.utils.data.Subset(rough_dataset, filtered_indices)
print(f"Length of filtered dataset: {len(filtered_dataset)}")

# 合并过滤后的数据集与训练集的前50%样本得到最终训练集
train_subset = Subset(train_dataset, list(range(0, len(train_dataset) // 2)))
final_trainset = ConcatDataset([train_subset, filtered_dataset])
print(f"Length of final train set: {len(final_trainset)}")

# 已知后50%样本中错误标注样本的比例为30%，测试过滤后的数据集长度确定一个合适的过滤阈值，宁缺毋滥
# threshold  filtered_dataset_length
# 0.95       1795
# 0.9        3410
# 0.8        6181
# 0.7        8804
# 0.6        11495
```

4. 数据增强

```python
# 在最终的训练集数量较少的情况下使用数据增强策略，提升模型的泛化能力
transform1 = transforms.Compose([
    transforms.RandomRotation(15),              # 随机旋转角度 ±15°
    transforms.RandomHorizontalFlip(),          # 随机水平翻转

    transforms.ColorJitter(brightness=0.1, contrast=0.1, saturation=0.1, hue=0.1),                                                    # 随机调整亮度
    transforms.RandomAffine(degrees=0,          # 随机仿射变换
                            translate=(0.1, 0.1),
                            scale=(0.9, 1.1),
                            shear=10),
#    transforms.RandomResizedCrop(32),           # 随机裁剪和缩放
#    transforms.RandomApply([transforms.RandomPerspective(distortion_scale=0.2)], p=0.5),                                       # 随机透视变换
#    实验时发现图片尺寸太小，不适合使用随机裁剪&随机透视
#    transforms.RandomApply([transforms.RandomErasing(p=1, scale=(0.02, 0.2))], p=0.5),
#    实验时发现因为随机擦除涉及到图片的'shape'属性，与这里的图片数据格式冲突
    transforms.ToTensor(),
    transforms.Normalize(mean, std)
])

test_augmented_dataset = datasets.CIFAR10(root='./data', train=True, download=True, transform=transform1)

# 显示数据增强后的图像
def show_images(test_dataset, num_images=5, h=1, w=5):

    fig, axs = plt.subplots(h, w, figsize=(10, 5))
    if h == 1:
        axs = np.expand_dims(axs, axis=0)
    for i in range(h):
        for j in range(w):
            idx = torch.randint(len(test_dataset), size=(1,)).item()
            image, _ = test_dataset[idx]

            npimg = np.array(image) / 2 + 0.5  # unnormalize
            npimg = np.transpose(npimg, (1, 2, 0))
            axs[i, j].imshow(npimg)
            axs[i, j].axis('off')

    plt.tight_layout()
    plt.show()

show_images(test_augmented_dataset, num_images=15, h=3, w=5)
# 数据增强后的样本图片展示如下
```



5. 模型定义

```
"""
模型设计思路借鉴VGG Net，每个卷积块都使用2层3x3卷积核叠加。模型整体由4个结构相同的卷积块，1个flatten层，1个dense层依次连接形成。每个卷积块的结构均为卷积层-批归一化层-卷积层-批归一化层-最大池化层-dropout层，最后的dense layer由3个全连接层和2个dropout层组成。相较于原始的基础模型，最终模型大大提升了复杂程度，并且大量使用dropout层。这是因为在实验过程中发现在模型训练过程中常常出现较为明显的过拟合现象，使用dropout并逐渐增大参数有利于减少过拟合。
"""
#baseline.py
class CIFAR10Model(nn.Module):
    def __init__(self):
        super(CIFAR10Model, self).__init__()
        # Block 1
        self.conv1_1 = nn.Conv2d(in_channels=3, out_channels=32, kernel_size=3, padding=1)
        self.conv1_2 = nn.Conv2d(in_channels=32, out_channels=32, kernel_size=3, padding=1)
        self.bn0 = nn.BatchNorm2d(32)
        self.bn1 = nn.BatchNorm2d(32)
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.drop1 = nn.Dropout(0.2)

        # Block 2
        self.conv2_1 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, padding=1)
        self.conv2_2 = nn.Conv2d(in_channels=64, out_channels=64, kernel_size=3, padding=1)
        self.bn2 = nn.BatchNorm2d(64)
        self.bn3 = nn.BatchNorm2d(64)
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.drop2 = nn.Dropout(0.3)

        # Block 3
        self.conv3_1 = nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, padding=1)
        self.conv3_2 = nn.Conv2d(in_channels=128, out_channels=128, kernel_size=3, padding=1)
        self.bn4 = nn.BatchNorm2d(128)
        self.bn5 = nn.BatchNorm2d(128)
        self.pool3 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.drop3 = nn.Dropout(0.4)

        # Block 4
        self.conv4_1 = nn.Conv2d(in_channels=128, out_channels=256, kernel_size=3, padding=1)
        self.conv4_2 = nn.Conv2d(in_channels=256, out_channels=256, kernel_size=3, padding=1)
        self.bn6 = nn.BatchNorm2d(256)
        self.bn7 = nn.BatchNorm2d(256)
        self.pool4 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.drop4 = nn.Dropout(0.5)

        # Flatten layer
        self.flatten = nn.Flatten()

        # Dense layer
        self.fc1 = nn.Linear(256 * 2 * 2, 512)
        self.drop_fc1 = nn.Dropout(0.5)
        self.fc2 = nn.Linear(512,256)
        self.drop_fc2 = nn.Dropout(0.5)
        self.out = nn.Linear(256, 10)

    def forward(self, x):
        # Block 1
        x = F.relu(self.conv1_1(x))
        x = self.bn0(x)
        x = F.relu(self.conv1_2(x))
        x = self.bn1(x)
        x = self.pool1(x)
        x = self.drop1(x)

        # Block 2
        x = F.relu(self.conv2_1(x))
        x = self.bn2(x)
        x = F.relu(self.conv2_2(x))
        x = self.bn3(x)
        x = self.pool2(x)
        x = self.drop2(x)

        # Block 3
        x = F.relu(self.conv3_1(x))
        x = self.bn4(x)
        x = F.relu(self.conv3_2(x))
        x = self.bn5(x)
        x = self.pool3(x)
        x = self.drop3(x)

        # Block 4
        x = F.relu(self.conv4_1(x))
        x = self.bn6(x)
        x = F.relu(self.conv4_2(x))
        x = self.bn7(x)
        x = self.pool4(x)
        x = self.drop4(x)

        # Flatten
        x = self.flatten(x)

        # Dense
        x = F.relu(self.fc1(x))
        x = self.drop_fc1(x)
        x = F.relu(self.fc2(x))
```

```python
        x = self.drop_fc2(x)
        x = self.out(x)

        return x
```

```
模型结构展示如下：
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 32, 32, 32]             896
       BatchNorm2d-2           [-1, 32, 32, 32]              64
            Conv2d-3           [-1, 32, 32, 32]           9,248
       BatchNorm2d-4           [-1, 32, 32, 32]              64
         MaxPool2d-5           [-1, 32, 16, 16]               0
           Dropout-6           [-1, 32, 16, 16]               0
            Conv2d-7           [-1, 64, 16, 16]          18,496
       BatchNorm2d-8           [-1, 64, 16, 16]             128
            Conv2d-9           [-1, 64, 16, 16]          36,928
      BatchNorm2d-10           [-1, 64, 16, 16]             128
        MaxPool2d-11             [-1, 64, 8, 8]               0
          Dropout-12             [-1, 64, 8, 8]               0
           Conv2d-13            [-1, 128, 8, 8]          73,856
      BatchNorm2d-14            [-1, 128, 8, 8]             256
           Conv2d-15            [-1, 128, 8, 8]         147,584
      BatchNorm2d-16            [-1, 128, 8, 8]             256
        MaxPool2d-17            [-1, 128, 4, 4]               0
          Dropout-18            [-1, 128, 4, 4]               0
           Conv2d-19            [-1, 256, 4, 4]         295,168
      BatchNorm2d-20            [-1, 256, 4, 4]             512
           Conv2d-21            [-1, 256, 4, 4]         590,080
      BatchNorm2d-22            [-1, 256, 4, 4]             512
        MaxPool2d-23            [-1, 256, 2, 2]               0
          Dropout-24            [-1, 256, 2, 2]               0
          Flatten-25                 [-1, 1024]               0
           Linear-26                  [-1, 512]         524,800
          Dropout-27                  [-1, 512]               0
           Linear-28                  [-1, 256]         131,328
          Dropout-29                  [-1, 256]               0
           Linear-30                   [-1, 10]           2,570
================================================================
Total params: 1,832,874
Trainable params: 1,832,874
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.01
Forward/backward pass size (MB): 2.13
Params size (MB): 6.99
Estimated Total Size (MB): 9.13
----------------------------------------------------------------
```

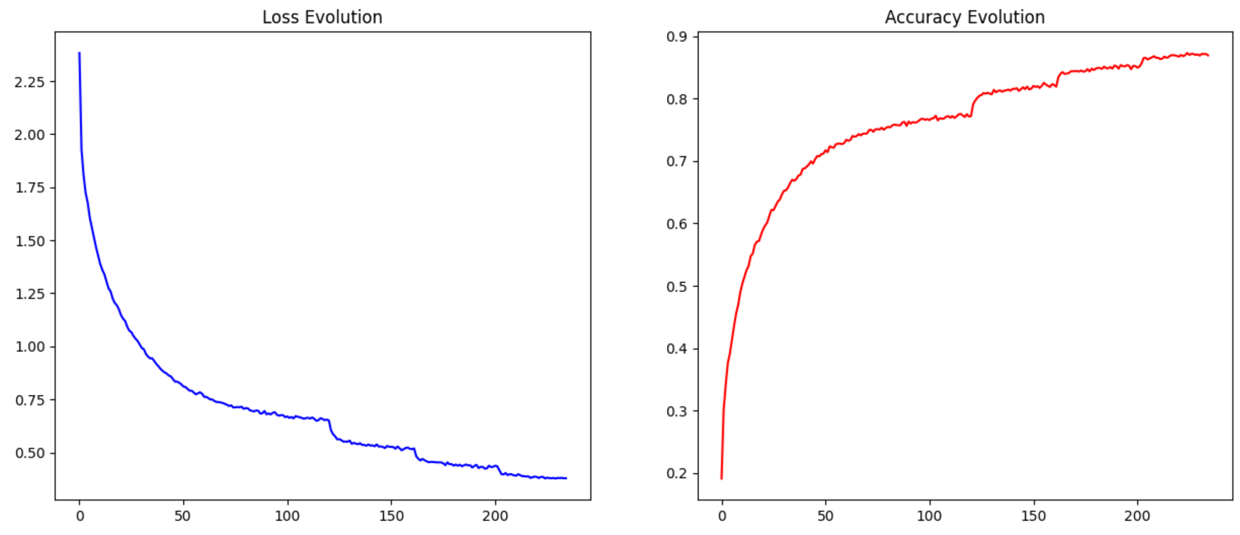
6. 模型训练

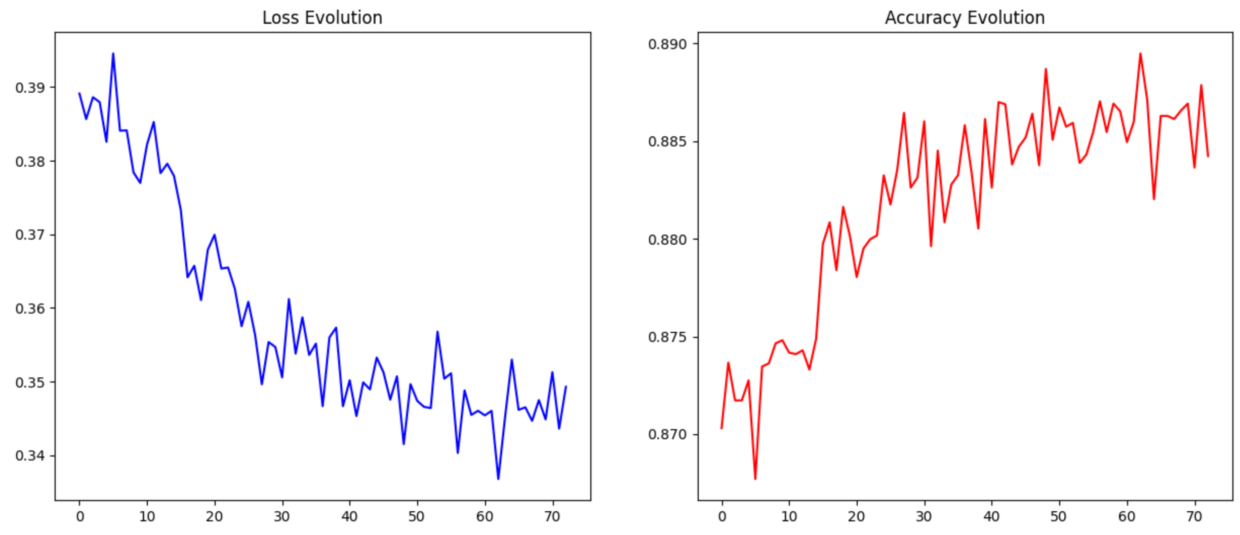
```python
to_pil = ToPILImage()
# 采用早停策略，经过10个epoch后如果训练损失不降低，终止训练过程
class EarlyStopping:
    def __init__(self, patience=7, delta=0):
        self.patience = patience
        self.delta = delta
        self.best_score = None
        self.early_stop = False
        self.counter = 0

    def __call__(self, train_loss):
        score = train_loss
        if self.best_score is None:
            self.best_score = score
            self.save_checkpoint(train_loss)
        elif score > self.best_score - self.delta:
            self.counter += 1
            if self.counter >= self.patience:
                self.early_stop = True
        else:
            self.best_score = score
            self.save_checkpoint(train_loss)
            self.counter = 0

    def save_checkpoint(self, train_loss):
        # 保存最优模型
        torch.save(model.state_dict(), 'checkpoint.pth')
        self.train_loss_min = train_loss

early_stopping = EarlyStopping(patience=10, delta=0.001)

model = CIFAR10Model().to(device)
criterion = nn.CrossEntropyLoss()
# 采用SGD优化器，引入L2正则化和nesterov动量，提升收敛能力
optimizer = optim.SGD(model.parameters(), lr=2e-2, momentum=0.9,
weight_decay=5e-4, nesterov=True)
# 采用动态学习率调度，经过4个epoch的训练后若损失不降低，学习率降为原来的一半
plateau_scheduler = ReduceLROnPlateau(optimizer, 'min', patience=4,
factor=0.5,min_lr=1e-5)

num_epochs = 300
# 记录学习过程中的损失，准确率和学习率，方便可视化
train_loss = []
train_accuracy = []
lr_list = []
lr_curr=2e-2

for epoch in range(num_epochs):
    # 重新生成训练集的数据加载器，这是为了每个epoch使用的都是训练集经过随机数据增强之后的不同样本
    train_loader = DataLoader(final_trainset, batch_size=64, shuffle=True,
                              collate_fn=lambda batch: (torch.stack([transform1(to_pil(item[0])) for item in batch]),
                                                    torch.tensor([item[1] for item in batch])))
    model.train()
    lr_list.append(optimizer.param_groups[0]['lr'])
    with tqdm(train_loader, unit='batch') as tepoch:
        epoch_loss = 0.0
        correct = 0
        total = 0
        for images, labels in tepoch:
            images, labels = to_device(images, device), to_device(labels, device)
            tepoch.set_description(f'Epoch {epoch+1}/{num_epochs}')
            optimizer.zero_grad()
            outputs = model(images)
            loss = criterion(outputs, labels)
            loss.backward()
            # 采用梯度裁剪策略，避免梯度爆炸
            nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)

            optimizer.step()
            epoch_loss += loss.item()

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

            tepoch.set_postfix(loss=loss.item(), accuracy=correct / total)
        # 记录训练过程中的参数，并在进度条后面的附加信息中显示
        epoch_loss /= len(train_loader)
        train_loss.append(epoch_loss)
        train_accuracy.append(correct / total)
        print(f"Training Loss: {epoch_loss:.4f}")
        if lr_curr!=optimizer.param_groups[0]['lr']:
            lr_curr=optimizer.param_groups[0]['lr']
            print(f"Learning Rate at epoch{epoch}: {lr_curr:.6f}")

    # 调整学习率
    plateau_scheduler.step(epoch_loss)
    # 检查是否需要早停
    early_stopping(epoch_loss)
    if early_stopping.early_stop:
        print("Early stopping")
        break

# 打印最终学习率
print(f"Final Learning Rate: {optimizer.param_groups[0]['lr']:.6f}")

```

```python
# 可视化训练过程
plt.figure(figsize=(15,6))
plt.subplot(1, 2, 1)
plt.plot(train_loss, label='Train Loss', color='blue')
plt.title('Loss Evolution')

plt.subplot(1, 2, 2)
plt.plot(train_accuracy, label='Train Accuracy', color='red')
plt.title('Accuracy Evolution')
plt.show()
```

由于训练轮次较多且每轮训练的学习曲线具有很相似的变化情况，这里只展示两批具有代表性的学习曲线，前期是模型从初始化开始直到逐渐收敛的过程，可见模型的学习速度逐渐减缓。由于采用了ReduceLROnPlateau策略，在学习率成倍减小时学习速度会出现一个小阶跃，图线有清晰体现。

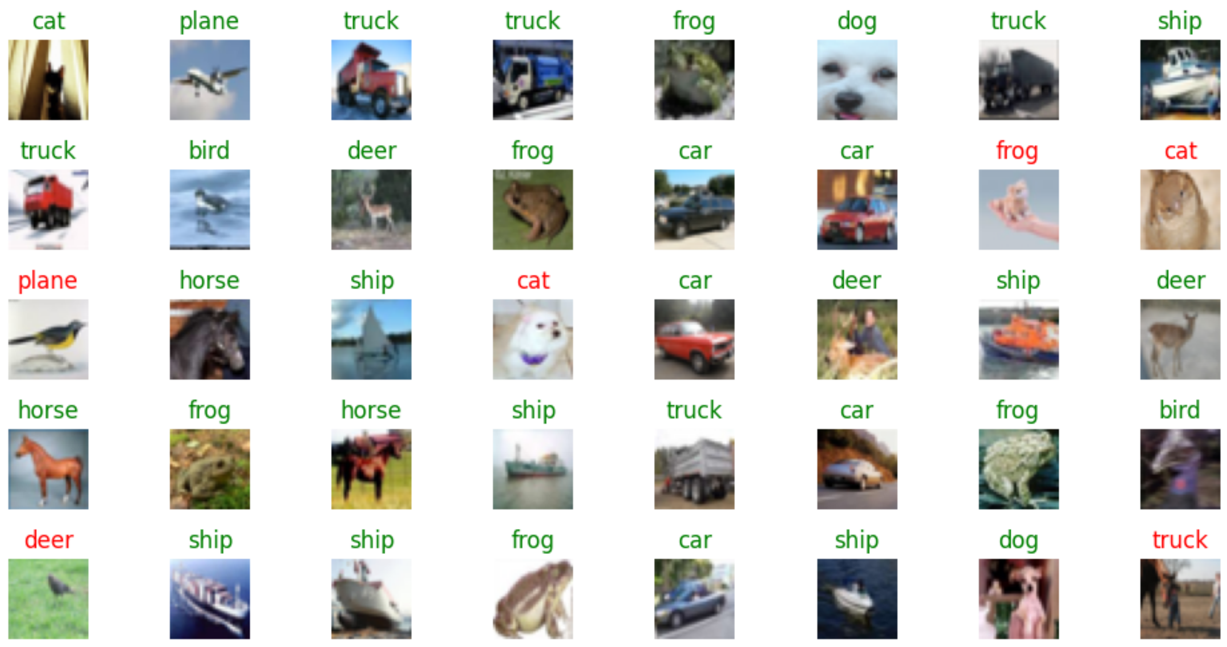
后期是在不断调整训练细节以得到最优模型数据的过程，训练普遍较短，学习效果只有比较细微的提升。虽然图线上体现出模型在训练集(代替验证集)上的最大准确率超过88.5%，但是当时存在较严重的过拟合，测试集准确率不超过80%，减小过拟合的措施在“实验总结”处有说明。





7. 模型测试

```python
classes = ('plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck')

# 统计各类别的准确率
correct_pred = {classname: 0 for classname in classes}
total_pred = {classname: 0 for classname in classes}

# 加载最优模型的参数
model.load_state_dict(torch.load("finalmodel.pth"))
model.eval()
test_correct = 0
test_total = 0
with torch.no_grad():
    for images, labels in test_loader:
        images, labels = to_device(images, device), to_device(labels, device)

        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        test_total += labels.size(0)
        test_correct += (predicted == labels).sum().item()

        for label, prediction in zip(labels, predicted):
            if label == prediction:
                correct_pred[classes[label]] += 1
            total_pred[classes[label]] += 1

# 打印每个类别的准确率
for classname, correct_count in correct_pred.items():
    accuracy = 100 * float(correct_count) / total_pred[classname]
    print(f'Accuracy for class: {classname:5s} is {accuracy:.1f} %')
    accuracy = test_correct / test_total

print(f'Accuracy of the network on the 10000 test images: {100*accuracy:.2f}%')

num_images = 40
fig, axs = plt.subplots(5, 8, figsize=(10, 5))
axs = axs.ravel()

for i in range(num_images):
    idx = torch.randint(len(test_dataset), size=(1,)).item()
    image, label = test_dataset[idx]

    model.eval()
    with torch.no_grad():
        image = image.unsqueeze(0)
        image = image.to(device)
        output = model(image)
        _, predicted = torch.max(output.data, 1)
        predicted_label = predicted.item()

    image = image / 2 + 0.5
    npimg = image.cpu().numpy()[0]
    axs[i].imshow(np.transpose(npimg, (1, 2, 0)))
    axs[i].set_title(classes[predicted_label], color='red' if label !=
predicted_label else 'green')
    axs[i].axis('off')
plt.tight_layout()
plt.show()
```



实验结果：快速验收

```
Accuracy for class: plane is 80.6 %
Accuracy for class: car   is 90.4 %
Accuracy for class: bird  is 73.2 %
Accuracy for class: cat   is 79.4 %
Accuracy for class: deer  is 79.7 %
Accuracy for class: dog   is 55.4 %
Accuracy for class: frog  is 80.4 %
Accuracy for class: horse is 78.3 %
Accuracy for class: ship  is 87.8 %
Accuracy for class: truck is 89.7 %
Accuracy of the network on the 10000 test images: 79.49%
#在配置好的环境下直接运行result_confirm.ipynb即可得到实验结果，cpu用时11秒左右
```

实验总结

实验中使用到的优化模型的策略包括增加模型复杂程度，使用简单网络度错误样本进行筛选，数据增强。在训练过程中为了提升训练效率采用的策略包括早停机制，动态调整学习率。一方面这些措施提升了模型的能力上限，另一方面，模型的待训练参数大大增加使得训练时间延长，样本筛选秉承宁缺毋滥的准则导致最终训练集的样本数量偏少，以及动态调整学习率的策略在时间上有滞后性。

为解决样本数量偏少的问题已经采用了数据增强策略，在模型中增加dropout层防止过拟合。为解决动态调整学习率的滞后，在训练过程中发现学习进度大大放缓时手动停止，调整学习率再继续训练。

在实验过程中为了优化模型性能进行了诸多尝试。在优化器方面，尝试使用AdamW优化器，但是由于当时在模型结构的最后误添加了softmax导致训练过程出现严重失误，又因为AdamW在训练后期性能表现不如SGD，所以最终仍然选择SGD优化器。

在学习率调整策略方面，在进行初步的探索之后确定了初期学习率预热+前中期余弦退火+中后期动态调整的策略。在实际过程中发现，学习率预热和余弦退火如果不进行比较精确的参数调整效果较差。另外由于模型的训练时间充裕，全程采用动态调整策略较稳定，所以最终只采用全程的ReduceLROnPlateau策略。

在后期训练过程中，由于各类别的准确率不同，曾尝试修改损失函数中各类别的权重。这样操作之后虽然权重更大的类别准确率有所提升，但是整体准确率不升反降，没能实现整体的优化。

由于在实际训练过程中，发现损失下降速度放缓时多次手动终止训练过程并调整代码·，所以实验没有得到模型从初始化直到训练为最优模型的完整学习曲线，报告中给出的曲线代表了两个阶段的过程：前期从模型初始化到损失函数逐渐收敛，后期微调训练环节以得到最优模型。实际训练过程中进行了将近20轮的长短不一的训练。

对于实验的最终结果：模型在测试集上的准确率为79.49%。在实验前期没有为模型添加更多dropout层时，模型在训练集上的准确率一度接近90%，但是存在较严重的过拟合。在修改模型之后，模型的过拟合大大消除，但是测试集准确率收敛在80%附近一直无法提升。

在决定提交作业之前，我了解到DenseNet的网络结构能够高效的利用参数，对图像分类任务可能有更优秀的表现，于是尝试训练一个DenseNet。然而尽管DenseNet的参数数量较之前减少将近一半，鉴于它后层同时被多个前层影响的特殊传播机制，整个网络所占空间大大增加，训练一个epoch的时间也从原来的不到一分半直接上升到超过十分钟。但是由于没有对优化器，学习率调度器和网络内参数进行细致的调整和其他未知原因，DenseNet的性能收敛在75%左右。由于缺乏更多精力再进行细致的探索，最终只能放弃DenseNet的调试。代码请见附录。

实验的可能的进一步优化策略是设计性能更优秀的模型结构，优化训练策略以及建立更加完善的样本筛选机制等等。此外为提升训练效率可采用GPU进行训练。

实验过程中的模型和训练机制是逐渐建立的，期间也产生不少思考，问题与失误，实验过程的部分记录在general.ipynb最开始的markdown cell有体现。通过咨询助教孙浩然学长，查找资料等，这些问题都得到了解决。通过这次实验，我掌握了设计，训练，测试和优化CNN的流程与基本方法。

附录

```python
# 含dropout层的简单DenseNet结构
import torch
import torch.nn as nn
import torch.nn.functional as F

class DenseLayer(nn.Module):
    def __init__(self, in_channels, growth_rate, drop_rate=0.2):
        super(DenseLayer, self).__init__()
        self.bn1 = nn.BatchNorm2d(in_channels)
        self.conv1 = nn.Conv2d(in_channels, 4 * growth_rate, kernel_size=1, bias=False)
        self.bn2 = nn.BatchNorm2d(4 * growth_rate)
        self.conv2 = nn.Conv2d(4 * growth_rate, growth_rate, kernel_size=3, padding=1, bias=False)
        self.drop_rate = drop_rate

    def forward(self, x):
        out = self.conv1(F.relu(self.bn1(x)))
        if self.drop_rate > 0:
            out = F.dropout(out, p=self.drop_rate, training=self.training)
        out = self.conv2(F.relu(self.bn2(out)))
        if self.drop_rate > 0:
            out = F.dropout(out, p=self.drop_rate, training=self.training)
        out = torch.cat([x, out], 1)
        return out

class DenseBlock(nn.Module):
    def __init__(self, in_channels, growth_rate, n_layers, drop_rate=0.2):
        super(DenseBlock, self).__init__()
        self.layers = nn.ModuleList()
        for i in range(n_layers):
            self.layers.append(DenseLayer(in_channels + i * growth_rate, growth_rate, drop_rate))

    def forward(self, x):
        for layer in self.layers:
            x = layer(x)
        return x

class TransitionLayer(nn.Module):
    def __init__(self, in_channels, out_channels, drop_rate=0.2):
        super(TransitionLayer, self).__init__()
        self.bn = nn.BatchNorm2d(in_channels)
        self.conv = nn.Conv2d(in_channels, out_channels, kernel_size=1, bias=False)
        self.pool = nn.AvgPool2d(kernel_size=2, stride=2)
        self.drop_rate = drop_rate

    def forward(self, x):
        x = self.conv(F.relu(self.bn(x)))
        if self.drop_rate > 0:
            x = F.dropout(x, p=self.drop_rate, training=self.training)
        x = self.pool(x)
        return x

class CIFAR10DenseNet(nn.Module):
    def __init__(self, growth_rate=12, block_layers=[6, 12, 24, 16], num_classes=10, drop_rate=0.2):
        super(CIFAR10DenseNet, self).__init__()
        num_init_features = 2 * growth_rate
        self.conv1 = nn.Conv2d(3, num_init_features, kernel_size=3, stride=1, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(num_init_features)

        self.features = nn.ModuleList()
        num_features = num_init_features
        for i, num_layers in enumerate(block_layers):
            self.features.append(DenseBlock(num_features, growth_rate, num_layers, drop_rate))
            num_features += num_layers * growth_rate
            if i != len(block_layers) - 1:
                self.features.append(TransitionLayer(num_features, num_features // 2, drop_rate))
                num_features = num_features // 2

        self.bn2 = nn.BatchNorm2d(num_features)
        self.classifier = nn.Linear(num_features, num_classes)

    def forward(self, x):
        x = self.conv1(x)
        x = self.bn1(x)
        for layer in self.features:
            x = layer(x)
        x = F.relu(self.bn2(x))
        x = F.adaptive_avg_pool2d(x, (1, 1)).view(x.size(0), -1)
        x = F.dropout(x, p=0.5, training=self.training)
        x = self.classifier(x)
        return x
```