HW-2: CNN实现CIFAR-10图像分类

姓名：罗威　学号：SA24218095

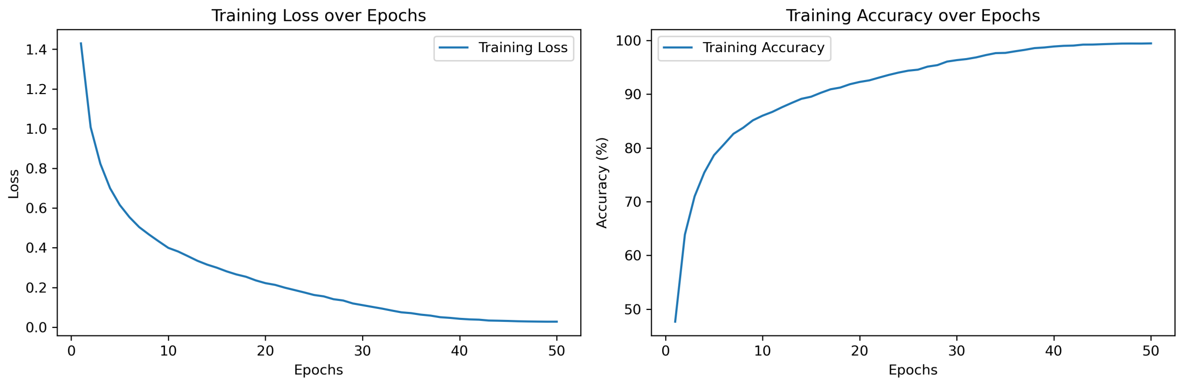
2.1 整体定义及结果总结

本次作业旨在使用ResNet网络对CIFAR-10数据集进行图像分类，通过调整网络结构、训练参数和优化策略，提高模型的分类准确率，并观察学习曲线的变化。

代码文件： `resnet.py`

训练测试时的终端输出： `terminal_output.txt`

训练的9层ResNet的测试结果表明，该模型在测试集上的正确率达到 **91.89%**。可视化曲线如下图：




```
Average Test Loss: 0.3073
Accuracy on the 10000 test images: 91.89 %

--- Per-class Accuracy ---
Accuracy of plane : 92.70 %
Accuracy of car   : 95.80 %
Accuracy of bird  : 88.60 %
Accuracy of cat   : 85.00 %
Accuracy of deer  : 91.70 %
Accuracy of dog   : 87.40 %
Accuracy of frog  : 94.20 %
Accuracy of horse : 93.10 %
Accuracy of ship  : 95.90 %
Accuracy of truck : 94.50 %

--- Per-class Precision ---
Precision of plane : 92.51 %  (TP: 927, Predicted as plane: 1002)
Precision of car   : 96.28 %  (TP: 958, Predicted as car: 995)
Precision of bird  : 90.22 %  (TP: 886, Predicted as bird: 982)
Precision of cat   : 84.08 %  (TP: 850, Predicted as cat: 1011)
Precision of deer  : 90.79 %  (TP: 917, Predicted as deer: 1010)
Precision of dog   : 87.84 %  (TP: 874, Predicted as dog: 995)
Precision of frog  : 93.27 %  (TP: 942, Predicted as frog: 1010)
Precision of horse : 95.49 %  (TP: 931, Predicted as horse: 975)
Precision of ship  : 96.09 %  (TP: 959, Predicted as ship: 998)
Precision of truck : 92.47 %  (TP: 945, Predicted as truck: 1022)
```

2.2 实验环境

- **编程语言**：`Python`
- **PyTorch**：`2.5.0+cu124`
- **数据集**：`CIFAR-10`
- **硬件设备**：`GPU: RTX 4060`

2.3 训练步骤

2.3.1 库版本检查

检查PyTorch和Torchvision的版本，以及CUDA是否可用。

```python
print(f"PyTorch Version: {torch.__version__}")
print(f"Torchvision Version: {torchvision.__version__}")
print(f"CUDA Available: {torch.cuda.is_available()}")
```

2.3.2 使用GPU并行训练

根据CUDA的可用性选择使用GPU或CPU进行训练。

```python
DEVICE = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

2.3.3 加载和预处理数据

- 定义训练集和测试集的预处理步骤，包括随机裁剪、随机水平翻转、转换为张量和归一化。
- 下载CIFAR-10数据集，并创建数据加载器。

```python

train_transform = transforms.Compose([
    transforms.RandomCrop(32, padding=4),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))
])

test_transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))
])

# 下载训练集和测试集
trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                        download=True, transform=train_transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=BATCH_SIZE,
                                          shuffle=True, num_workers=2)

testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                       download=True, transform=test_transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=BATCH_SIZE,
                                         shuffle=False, num_workers=2)
```

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

```

2.3.4 定义带残差连接的DeepCNN：ResNet

- 定义BasicBlock模块，包含两个卷积层和一个残差连接。
- 定义ResNet网络，由多个BasicBlock模块组成，并包含全局平均池化和全连接层。
- 定义ResNet18和ResNet9两种网络结构。

```python
class BasicBlock(nn.Module):
    expansion = 1 # BasicBlock的输出通道数与目标输出通道数相同
    def __init__(self, in_planes, planes, stride=1):
        super(BasicBlock, self).__init__()
        self.conv1 = nn.Conv2d(in_planes, planes, kernel_size=3, stride=stride, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(planes)
        self.conv2 = nn.Conv2d(planes, planes, kernel_size=3, stride=1, padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(planes)

        self.shortcut = nn.Sequential()
        # 如果输入输出通道数不同，或者stride不为1（特征图尺寸改变），就使用一层卷积和BatchNorm来修正维度
        if stride != 1 or in_planes != self.expansion*planes:
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_planes, self.expansion*planes, kernel_size=1, stride=stride, bias=False),
                nn.BatchNorm2d(self.expansion*planes)
            )
    def forward(self, x):
        out = F.relu(self.bn1(self.conv1(x)))
        out = self.bn2(self.conv2(out))
        out += self.shortcut(x) # 残差连接
        out = F.relu(out)
        return out

class ResNet(nn.Module):
    def __init__(self, block, num_blocks, num_classes=10):
        super(ResNet, self).__init__()
        self.in_planes = 64 # 初始通道数

        # 初始卷积层：3x3的卷积
        self.conv1 = nn.Conv2d(3, self.in_planes, kernel_size=3, stride=1, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(self.in_planes)

        self.layer1 = self._make_layer(block, 64, num_blocks[0], stride=1)
        self.layer2 = self._make_layer(block, 128, num_blocks[1], stride=2) # stride=2 实现下采样
        self.layer3 = self._make_layer(block, 256, num_blocks[2], stride=2)
        self.layer4 = self._make_layer(block, 512, num_blocks[3], stride=2)

        self.avgpool = nn.AdaptiveAvgPool2d((1, 1)) # 全局平均池化
        self.linear = nn.Linear(512*block.expansion, num_classes)
```

```python

    def _make_layer(self, block, planes, num_blocks, stride):
        strides = [stride] + [1]*(num_blocks-1)
        layers = []
        for s in strides:
            layers.append(block(self.in_planes, planes, s))
            self.in_planes = planes * block.expansion
        return nn.Sequential(*layers)

    def forward(self, x):
        out = F.relu(self.bn1(self.conv1(x)))
        out = self.layer1(out)
        out = self.layer2(out)
        out = self.layer3(out)
        out = self.layer4(out)
        out = self.avgpool(out)
        out = torch.flatten(out, 1)
        out = self.linear(out)
        return out

# 4.初始化ResNet配置
# 4.1 这是一个简化的ResNet18结构，适应CIFAR-10
def ResNet18():
    # ResNet18的block配置是 [2, 2, 2, 2]
    return ResNet(BasicBlock, [2, 2, 2, 2], num_classes=10)

# 4.2 一个更小的ResNet变体，包括全连接层在内一共9层
def ResNet9():
    return ResNet(BasicBlock, [1,1,1,1], num_classes=10)
```

2.3.5 初始化ResNet配置

- 选择ResNet9作为训练模型，并将其移动到指定的设备上。

```python
# model = ResNet18().to(DEVICE)

# 使用较小的ResNet来进行训练，因为这个网络的性能已经足够
model = ResNet9().to(DEVICE)
print(model)
```

2.3.6 定义损失函数和优化器

- 使用交叉熵损失函数作为损失函数。
- 使用随机梯度下降（SGD）优化器，并设置学习率、动量和权重衰减。
- 使用余弦退火调度器调整学习率。

```python
criterion = nn.CrossEntropyLoss()
# 优化器：对于ResNet结构，SGD with momentum比Adam优化器更好
optimizer = optim.SGD(model.parameters(), lr=LEARNING_RATE, momentum=0.9, weight_decay=WEIGHT_DECAY)

# 学习率调度器：CosineAnnealingLR——余弦退火调度来调整学习率
scheduler = optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=EPOCHS)
```

2.3.7 训练模型

- 定义训练函数，包括前向传播、计算损失、反向传播和更新参数。
- 在训练过程中，记录训练损失和准确率，并打印每个epoch的训练结果。

```
def train_model(model, trainloader, criterion, optimizer, scheduler, epochs):
    print("\n--- Training Started ---")
    train_losses = []
    train_accuracies = []

    for epoch in range(epochs):
        model.train()
        running_loss = 0.0
        correct_train = 0
        total_train = 0

        current_lr = optimizer.param_groups[0]['lr']
        print(f"Epoch [{epoch+1}/{epochs}], Current Learning Rate: {current_lr:.6f}")

        for i, data in enumerate(trainloader, 0):
            inputs, labels = data
            inputs, labels = inputs.to(DEVICE), labels.to(DEVICE)

            optimizer.zero_grad()
            outputs = model(inputs)
            # 1. 计算loss
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            # 2. 累加每个batch的总损失
            running_loss += loss.item() * inputs.size(0)

            # 3. 计算准确率
            _, predicted = torch.max(outputs.data, 1)
            total_train += labels.size(0)
            correct_train += (predicted == labels).sum().item()

        # 4. 正确计算epoch平均损失
        epoch_train_loss = running_loss / total_train
        epoch_train_acc = 100 * correct_train / total_train

        train_losses.append(epoch_train_loss)
        train_accuracies.append(epoch_train_acc)

        print(f"Epoch {epoch+1} finished. Avg Training Loss: {epoch_train_loss:.4f}, Training Accuracy: {epoch_train_acc:.2f}%")
        scheduler.step()

    return train_losses, train_accuracies

train_losses, train_accuracies = train_model(model, trainloader, criterion, optimizer, scheduler, EPOCHS)
```

2.3.8 测试模型并计算评价指标

- 定义测试函数，包括前向传播、计算损失和准确率。
- 在测试过程中，计算每一类别的准确率和精度，并打印测试结果。

```python
# 测试模型并计算评价指标
def test_model_and_evaluate(model, testloader, classes):
    print("\n--- Testing Started ---")
    model.eval()
    correct = 0
    total = 0
    test_loss = 0.0

    class_true_positives = defaultdict(int)
    class_predicted_positives = defaultdict(int)
    class_correct = list(0. for i in range(len(classes)))
    class_total = list(0. for i in range(len(classes)))


    with torch.no_grad():
        for data in testloader:
            images, labels = data
            images, labels = images.to(DEVICE), labels.to(DEVICE)
            outputs = model(images)
            loss = criterion(outputs, labels)
            test_loss += loss.item()

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

            # 计算每一类别的准确率
            c = (predicted == labels).squeeze()
            for i in range(labels.size(0)):
                label = labels[i]
                class_correct[label] += c[i].item()
                class_total[label] += 1

            # 计算每一类别的精度
            for i in range(labels.size(0)):
                label = labels[i].item()
                pred = predicted[i].item()
                class_predicted_positives[pred] += 1
                if label == pred:
                    class_true_positives[label] += 1

    avg_test_loss = test_loss / len(testloader)
    overall_accuracy = 100 * correct / total
    print(f'\nAverage Test Loss: {avg_test_loss:.4f}')
    print(f'Accuracy on the {total} test images: {overall_accuracy:.2f} %')

    print("\n--- Per-class Accuracy ---")
    for i in range(len(classes)):
        if class_total[i] > 0:
```

```
                print(f'Accuracy of {classes[i]:5s} : {100 * class_correct[i] / class_total[i]:.2f} %')
            else:
                print(f'Accuracy of {classes[i]:5s} : N/A (no instances in test set)')


    print("\n--- Per-class Precision ---")
    for i in range(len(classes)):
        class_name = classes[i]
        tp = class_true_positives[i]
        tp_plus_fp = class_predicted_positives[i]
        precision = 0
        if tp_plus_fp > 0:
            precision = 100 * tp / tp_plus_fp
        print(f'Precision of {class_name:5s} : {precision:.2f} %  (TP: {tp}, Predicted as {class_name}: {tp_plus_fp})')

    print('--- Finished Testing ---')
    return avg_test_loss, overall_accuracy

avg_test_loss, test_accuracy = test_model_and_evaluate(model, testloader, classes)

```

2.3.9 绘制学习曲线

- 绘制训练损失和准确率随epoch的变化曲线，并保存为图像文件。

```
matplotlib.use('Agg')
plt.figure(figsize=(12, 4))
plt.subplot(1, 2, 1)
plt.plot(range(1, EPOCHS + 1), train_losses, label='Training Loss')
plt.title('Training Loss over Epochs')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(range(1, EPOCHS + 1), train_accuracies, label='Training Accuracy')
plt.title('Training Accuracy over Epochs')
plt.xlabel('Epochs')
plt.ylabel('Accuracy (%)')
plt.legend()
plt.tight_layout()
plt.tight_layout()
plt.savefig('result.png',  # 保存路径
            dpi=300,             # 分辨率
            bbox_inches='tight',   # 去除多余白边
            format='png'           # 输出格式
           )
```

2.4 训练结果

2.4.1 模型训练结果

- 训练损失随epoch的增加逐渐减小，最终收敛到较低水平。
- 训练准确率随epoch的增加逐渐提高，最终达到**99.46%**。

2.4.2 模型测试结果

- 测试损失和准确率与训练结果相似，表明模型具有较好的泛化能力。
- 每一类别的准确率和精度最低为**85%**，最高为**95.9%**，表现出很好的识别能力。

2.4.3 学习曲线

- 训练损失曲线和准确率曲线呈现出典型的收敛趋势，表明模型的训练过程是稳定的。