中山大学计算机学院

人工智能

本科生实验报告

（2022 学年春季学期）

课程名称：Artificial Intelligence

教学班级	**计科 2 班**	专业（方向）	**计算机科学与技术**
学号	**22336126**	姓名	**李漾**

一、 实验题目

使用 ResNet 完成图像分类

二、 实验内容

论文理解：

ResNet（残差网络）之所以有效，主要归功于以下几个关键创新点：

1.残差学习：ResNet 的核心思想是让深层网络学习输入与输出间的残差（即差异），而非直接学习映射关系。这种方式简化了学习过程，因为当最优解接近于恒等映射时，学习残差比学习整个映射更为容易。

2.快捷连接（Shortcut Connections）：通过在网络中增加快捷连接，ResNet 允许信号绕过一层或多层直接传递，有效解决了深层网络训练中的梯度消失问题，使得网络能够训练得更深而不失性能。

3.退化问题的解决：在传统的深层网络中，增加网络深度会导致准确度饱和后迅速下降，即退化问题。ResNet 通过残差学习框架有效缓解了这一问题，使得网络在增加深度的同时能够获得更好的性能。

4.实验验证：在 ImageNet 等大规模数据集上的广泛实验表明，ResNet 能够随着网络深度的增加而提高准确度，这与传统的深层网络形成鲜明对比。

5.通用性和适用性：ResNet 不仅在图像分类任务上取得了突破性进展，还在对象检测等其他视觉识别任务上展现了其强大的通用性和适用性。

6.网络架构的灵活性：论文中提出的不同版本的 ResNet 展示了网络架构的灵活性，无论是浅层网络还是深层网络，通过引入残差学习都能获得性能上的提升。

7.优化策略：ResNet 在实现时采用了批量归一化（Batch Normalization）和适当的权重初始

化等技术，这些都有助于加速网络的收敛，进一步提高了训练效率。

综上所述，ResNet 之所以有效，是因为它通过创新的残差学习框架和快捷连接机制，解决了深层网络训练中的一些关键问题，并在多种视觉任务上展现了卓越的性能和泛化能力。

1. 算法原理

该代码实现了一个卷积神经网络（CNN）用于对 MNIST 手写数字数据集进行分类，采用了基于残差块（ResNet）的网络结构。整个流程分为以下几个主要步骤：

1. **数据预处理：**
    1. 下载 MNIST 数据集，并对图像进行缩放（32x32），转换为张量，并进行归一化。
2. **数据加载：**
    1. 使用 PyTorch 的 DataLoader 类，定义用于批量加载训练和测试数据的加载器。
3. **模型定义：**
    1. 定义一个残差块（BasicBlock），实现残差连接的基本结构。
    2. 定义一个包含多个残差块的 ResNet 模型（My_ResNet），通过调用不同层次的残差块构建深度网络。
4. **模型训练：**
    1. 使用交叉熵损失函数（CrossEntropyLoss）作为损失函数。
    2. 使用随机梯度下降（SGD）优化器来更新模型参数。
    3. 训练过程中，前向传播计算损失，反向传播更新参数。
5. **模型测试：**
    1. 在测试集上评估模型性能，计算平均损失和准确率。
6. **可视化：**
    1. 绘制训练和测试损失以及测试准确率随时间变化的图表，以便观察模型的收敛情况。

2. 伪代码

1. 初始化超参数：
    - batch_size = 50
    - learning_rate = 0.1
    - num_epochs = 10

2. 数据预处理和加载：
   - 下载并加载 MNIST 训练集和测试集，进行图像缩放和归一化
   - 定义 DataLoader 用于批量加载数据

3. 定义残差块（BasicBlock）：
   - 继承自 nn.Module
   - 初始化时定义卷积层、批归一化层、ReLU 激活函数
   - 实现前向传播函数 forward()

4. 定义残差网络（My_ResNet）：
   - 继承自 nn.Module
   - 初始化时定义网络层次结构，包括卷积层、批归一化层、残差块层、全局平均池化层、全连接层
   - 实现前向传播函数 forward()

5. 实例化模型、损失函数和优化器：
   - 将模型移动到 GPU（如果可用）
   - 使用交叉熵损失函数
   - 使用随机梯度下降优化器

6. 定义训练函数 train()：
   - 设置模型为训练模式
   - 对每个批次数据进行前向传播、计算损失、反向传播、更新参数
   - 记录并打印损失

7. 定义测试函数 test()：
   - 设置模型为评估模式
   - 对测试数据进行前向传播、计算损失和准确率
   - 打印平均损失和准确率

8. 训练和测试循环：
   - 进行 num_epochs 轮训练和测试
   - 记录训练和测试损失、测试准确率

9. 可视化训练和测试结果：
   - 绘制训练和测试损失、测试准确率随时间变化的曲线

10. 在测试集上评估模型并输出前 10 个样本的真实和预测标签

Initialize hyperparameters:

```
batch_size = 50
learning_rate = 0.1
num_epochs = 10
```

Data preprocessing and loading:

train_dataset = Load MNIST train dataset with transformations (resize, to_tensor, normalize)
test_dataset = Load MNIST test dataset with transformations (resize, to_tensor, normalize)
train_loader = DataLoader(train_dataset, batch_size, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size, shuffle=False)

Define BasicBlock class:

```
class BasicBlock(nn.Module):
    def __init__(self, in_channels, out_channels, stride=1):
        Define conv1, bn1, relu, conv2, bn2
        if in_channels != out_channels or stride != 1:
            Define downsample with conv and bn

    def forward(self, x):
        identity = x
        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)
        out = self.conv2(out)
        out = self.bn2(out)
        if self.downsample:
            identity = self.downsample(x)
        out += identity
        out = self.relu(out)
        return out
```

Define My_ResNet class:

```
class My_ResNet(nn.Module):
    def __init__(self, block, num_blocks, num_classes=10):
        Define conv1, bn1, relu
        Define layer1, layer2, layer3 using _make_layer
        Define avgpool, fc

    def _make_layer(self, block, out_channels, num_blocks, stride):
        layers = [block(self.in_channels, out_channels, stride)]
        self.in_channels = out_channels * block.expansion
        for _ in range(1, num_blocks):
            layers.append(block(self.in_channels, out_channels))
        return nn.Sequential(*layers)

    def forward(self, x):
        out = self.conv1(x)
        out = self.bn1(out)
```

```
        out = self.relu(out)
        out = self.layer1(out)
        out = self.layer2(out)
        out = self.layer3(out)
        out = self.avgpool(out)
        out = torch.flatten(out, 1)
        out = self.fc(out)
        return out
```

Instantiate model, loss function, and optimizer:

```
device = cuda if available else cpu
model = My_ResNet(BasicBlock, [1, 1, 1]).to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=learning_rate)
```

Define train function:

```
def train(model, device, train_loader, optimizer, criterion):
    model.train()
    for each batch (data, target) in train_loader:
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()
        output = model(data)
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()
```

Define test function:

```
def test(model, device, test_loader, criterion):
    model.eval()
    test_loss = 0
    correct = 0
    with no_grad():
        for data, target in test_loader:
            data, target = data.to(device), target.to(device)
            output = model(data)
            test_loss += criterion(output, target).item()
            pred = output.argmax(dim=1, keepdim=True)
            correct += pred.eq(target.view_as(pred)).sum().item()
    test_loss /= len(test_loader.dataset)
    accuracy = 100. * correct / len(test_loader.dataset)
    print(f'Average loss: {test_loss:.4f}, Accuracy: {accuracy:.0f}%')
```

Train and test loop:

```
    for epoch in range(num_epochs):
        train(model, device, train_loader, optimizer, criterion)
        test(model, device, test_loader, criterion)

Visualize results:
    Plot training and testing loss
    Plot testing accuracy

Evaluate model on test data:
    model.eval()
    with no_grad():
        test_data, test_target = next(iter(test_loader))
        test_data, test_target = test_data.to(device), test_target.to(device)
        output = model(test_data)
        predicted = output.argmax(dim=1)
        print('Real values:', test_target[:10].cpu().numpy())
        print('Predicted values:', predicted[:10].cpu().numpy())
```

3. 关键代码展示（带注释）

1.数据预处理：

通过对图像进行缩放和归一化，确保数据以适当的形式输入到网络中，提升模型性能。这些预处理步骤可以使得训练过程更加稳定和高效。采用了一系列转换操作来确保输入数据处于适当的形式。首先，通过 transforms.Resize((32, 32)) 将图像缩放到 32x32 像素大小，这是为了使得图像大小与网络模型匹配。接着，通过 transforms.ToTensor() 将图像转换为张量，使得图像能够被 PyTorch 所处理。最后，通过 transforms.Normalize((0.1307,), (0.3081,)) 对图像进行归一化处理，这个归一化参数是根据 MNIST 数据集的均值和标准差计算得到的。这些预处理步骤有助于确保数据在训练过程中保持一致性和稳定性，提高了模型的性能和收敛速度。

```
train_dataset = datasets.MNIST(
    root='./data',
    train=True,
    download=True,
    transform=transforms.Compose([
        transforms.Resize((32, 32)),  # 将图像大小调整为 32x32
        transforms.ToTensor(),  # 将图像转换为 Tensor
        transforms.Normalize((0.1307,), (0.3081,))  # 对图像进行标准化
    ])
)

test_dataset = datasets.MNIST(
```

```
    root='./data',
    train=False,
    download=True,
    transform=transforms.Compose([
        transforms.Resize((32, 32)),
        transforms.ToTensor(),
        transforms.Normalize((0.1307,), (0.3081,))
    ])
)
```

2. ResNet-10 的实现：

My_ResNet 使用了三个层次的残差块（layer1、layer2 和 layer3），每个层次各有一个残差块，总共三个残差块，加上初始的卷积层和全连接层，总计 10 个主要层（3 个卷积层 + 3 个残差块（每个 2 个卷积层） + 1 个全连接层 = 10 层）。
该架构在 MNIST 数据集上运行，输入图像大小为 32x32，通道数为 1（灰度图像）。通过该架构，ResNet 模型能够有效地学习图像特征，从而实现高效的分类。

初始卷积层和批量归一化层：

```
self.conv1 = nn.Conv2d(1, self.in_channels, kernel_size=3, stride=1, padding=1, bias=False)
self.bn1 = nn.BatchNorm2d(self.in_channels)
self.relu = nn.ReLU(inplace=True)
```

残差块层：

```
self.layer1 = self._make_layer(block, 64, num_blocks[0], stride=1)
self.layer2 = self._make_layer(block, 128, num_blocks[1], stride=2)
self.layer3 = self._make_layer(block, 256, num_blocks[2], stride=2)
```

全局平均池化层：

```
self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
```

全连接层：

```
self.fc = nn.Linear(256 * BasicBlock.expansion, num_classes)
```

_make_layer 方法

```
def _make_layer(self, block, out_channels, num_blocks, stride):
    layers = []
    layers.append(block(self.in_channels, out_channels, stride))
    self.in_channels = out_channels * BasicBlock.expansion
    for _ in range(1, num_blocks):
        layers.append(block(self.in_channels, out_channels))
    return nn.Sequential(*layers)
```

block：残差块的类型。
out_channels：每个块输出的通道数。
num_blocks：当前层包含的残差块数量。
stride：第一个残差块的步长。
该方法返回一个包含多个残差块的序列。它先创建一个带有步长的残差块，然后创建其余的残差块，这些块的输入通道数等于输出通道数。

forward 方法

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

前向传播过程包括以下步骤：
输入图像经过初始卷积层、批量归一化层和 ReLU 激活函数。
输入依次通过三个残差层（每个层次包括多个残差块）。
输出经过全局平均池化层，得到一个固定大小的输出。
输出经过展平层和全连接层，得到最终的分类结果。

4. 创新点&优化（如果有）

- **残差网络（ResNet）结构：**

  采用残差块（BasicBlock）结构，解决了深层神经网络中的梯度消失和梯度爆炸问题，使得训练更深层的网络成为可能。

  代码中的 `out += identity` 实现了残差连接，通过直接将输入添加到输出，有助于信息在网络中的传播。

- **自定义的残差块和残差网络：**

  代码中自定义了 BasicBlock 和 My_ResNet 类，通过灵活配置残差块数量和层次结构，展示了如何构建复杂的神经网络。这种灵活性允许根据具体需求调整网络结构。

优化：

1. 训练过程的记录和可视化：

记录了训练和测试过程中的损失和准确率，并绘制了相应的图表，帮助分析和优化模型。这样可以直观地观察模型的训练过程，了解模型是否过拟合或欠拟合。

```
# 绘制训练和测试损失以及测试准确率的图表
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.plot(train_losses, label='Training Loss')
plt.plot(test_losses, label='Testing Loss')
plt.title('Loss over epochs')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(test_accuracies, label='Testing Accuracy')
plt.title('Accuracy over epochs')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

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

2. 使用 GPU 加速训练：

如果 GPU 可用，将模型和数据移动到 GPU，显著加速训练和测试过程。代码中通过 device = torch.device("cuda" if torch.cuda.is_available() else "cpu") 自动检测并使用 GPU。

```
model = model.to(device)
```

3. 简化残差连接逻辑：

在 BasicBlock 类中，通过条件判断是否需要下采样，简化了残差连接的逻辑，使得代码更加简洁和高效。这种条件判断确保了输入和输出维度匹配，避免了尺寸不一致的问题。

在残差块中，如果输入和输出的维度相同（即通道数和特征图大小都相同），则可以直接将输入添加到输出上。如果输入和输出的维度不同（如通道数不同或者步幅不为

1 导致特征图大小不同），需要对输入进行下采样，使得输入和输出维度匹配后再相加。

```python
class BasicBlock(nn.Module):
    expansion = 1

    def __init__(self, in_channels, out_channels, stride=1):
        super(BasicBlock, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=stride, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3, stride=1, padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(out_channels)
        self.downsample = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, kernel_size=1, stride=stride, bias=False),
            nn.BatchNorm2d(out_channels)
        ) if in_channels != out_channels or stride != 1 else None

    def forward(self, x):
        identity = x
        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)
        out = self.conv2(out)
        out = self.bn2(out)
        if self.downsample is not None:
            identity = self.downsample(x)
        out += identity
        out = self.relu(out)
        return out
```

使用了 Python 中的条件表达式（ternary operator）来定义 downsample。if in_channels != out_channels or stride != 1:：这里判断输入通道数与输出通道数是否相同，以及步幅是否为 1。如果不相同或步幅不为 1，则需要对输入进行下采样。

self.downsample = nn.Sequential(...)：通过一个包含卷积层和批归一化层的顺序容器（nn.Sequential），对输入进行下采样。卷积核大小为 1，步幅为 stride，使得输入的尺寸与输出匹配。

这样就避免了不必要的下采样操作，使得代码更加简洁和高效。

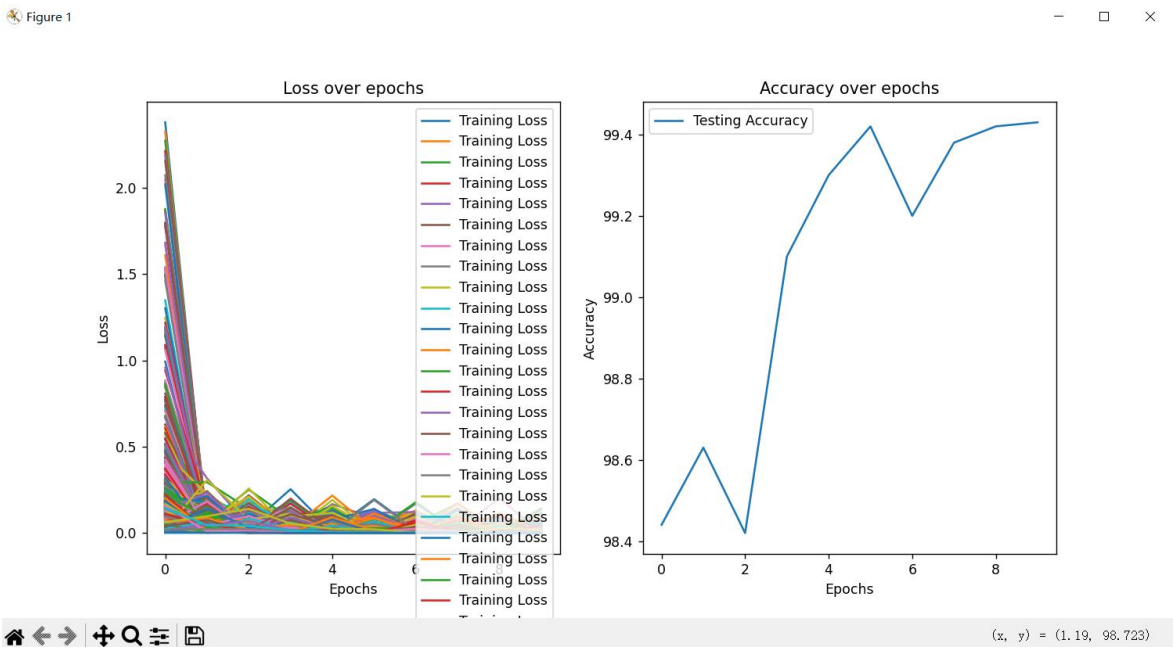
残差连接：

identity = x：保存输入 x。

如果需要下采样，则通过 self.downsample 对输入进行处理，使得 identity 的尺寸与输出 out 匹配。

out += identity：将输入添加到输出上，实现残差连接。

三、 实验结果及分析



如图为 ResNet-10 的 loss 曲线以及准确率，展示了每次的 train-loss 和 test-loss 曲线

```
Train Epoch: 9 [40000/60000] Loss: 0.000903
Train Epoch: 9 [45000/60000] Loss: 0.006431
Train Epoch: 9 [50000/60000] Loss: 0.001511
Train Epoch: 9 [55000/60000] Loss: 0.007493
Test set: Average loss: 0.0004, Accuracy: 9

d:\人工智能\ResNetLike.py:180: UserWarning:
ations.
  plt.tight_layout()
Real values: [7 2 1 0 4 1 4 9 5 9]
Predicted values: [7 2 1 0 4 1 4 9 5 9]
```

最后输出了实际数字与预测数字，可以看到准确率很高

此外，我编写了一个代码用于展示不同模型参数对于结果的影响，可以得到如下数据：

```
Experiment: ResNet-10_LR_0.01_BS_32, Epoch: 0, Train Loss: 0.014376, Test Loss: 0.003428, Accuracy: 97.30%
Experiment: ResNet-10_LR_0.01_BS_64, Epoch: 0, Train Loss: 0.011155, Test Loss: 0.012467, Accuracy: 72.58%
Experiment: ResNet-10_LR_0.01_BS_128, Epoch: 0, Train Loss: 0.008633, Test Loss: 0.003719, Accuracy: 90.33%
Experiment: ResNet-10_LR_0.1_BS_32, Epoch: 0, Train Loss: 0.005000, Test Loss: 0.001647, Accuracy: 98.55%
Experiment: ResNet-10_LR_0.1_BS_64, Epoch: 0, Train Loss: 0.003197, Test Loss: 0.001809, Accuracy: 96.72%
Experiment: ResNet-10_LR_0.1_BS_128, Epoch: 0, Train Loss: 0.002303, Test Loss: 0.000996, Accuracy: 96.33%
Experiment: ResNet-10_LR_0.001_BS_32, Epoch: 0, Train Loss: 0.054815, Test Loss: 0.039003, Accuracy: 78.50%
Experiment: ResNet-10_LR_0.001_BS_64, Epoch: 0, Train Loss: 0.030739, Test Loss: 0.026549, Accuracy: 61.63%
Experiment: ResNet-10_LR_0.001_BS_128, Epoch: 0, Train Loss: 0.016604, Test Loss: 0.015414, Accuracy: 45.65%
Experiment: ResNet-18_LR_0.01_BS_32, Epoch: 0, Train Loss: 0.006458, Test Loss: 0.001445, Accuracy: 98.55%
Experiment: ResNet-18_LR_0.01_BS_64, Epoch: 0, Train Loss: 0.004882, Test Loss: 0.001073, Accuracy: 98.23%
Experiment: ResNet-18_LR_0.01_BS_128, Epoch: 0, Train Loss: 0.003899, Test Loss: 0.002089, Accuracy: 92.11%
Experiment: ResNet-18_LR_0.1_BS_32, Epoch: 0, Train Loss: 0.003643, Test Loss: 0.001909, Accuracy: 98.08%
Experiment: ResNet-18_LR_0.1_BS_64, Epoch: 0, Train Loss: 0.001934, Test Loss: 0.001573, Accuracy: 96.54%
Experiment: ResNet-18_LR_0.1_BS_128, Epoch: 0, Train Loss: 0.001315, Test Loss: 0.000524, Accuracy: 97.97%
Experiment: ResNet-18_LR_0.001_BS_32, Epoch: 0, Train Loss: 0.030604, Test Loss: 0.007563, Accuracy: 96.47%
Experiment: ResNet-18_LR_0.001_BS_64, Epoch: 0, Train Loss: 0.023638, Test Loss: 0.011184, Accuracy: 91.96%
Experiment: ResNet-18_LR_0.001_BS_128, Epoch: 0, Train Loss: 0.014631, Test Loss: 0.011856, Accuracy: 69.00%
Experiment: ResNet-34_LR_0.01_BS_32, Epoch: 0, Train Loss: 0.003694, Test Loss: 0.002425, Accuracy: 97.28%
Experiment: ResNet-34_LR_0.01_BS_64, Epoch: 0, Train Loss: 0.002314, Test Loss: 0.000934, Accuracy: 97.88%
Experiment: ResNet-34_LR_0.01_BS_128, Epoch: 0, Train Loss: 0.001637, Test Loss: 0.000333, Accuracy: 98.64%
Experiment: ResNet-34_LR_0.1_BS_32, Epoch: 0, Train Loss: 0.007217, Test Loss: 0.001502, Accuracy: 98.66%
Experiment: ResNet-34_LR_0.1_BS_64, Epoch: 0, Train Loss: 0.004301, Test Loss: 0.000723, Accuracy: 98.59%
Experiment: ResNet-34_LR_0.1_BS_128, Epoch: 0, Train Loss: 0.003043, Test Loss: 0.000546, Accuracy: 97.81%
Experiment: ResNet-34 LR 0.001 BS 32, Epoch: 0, Train Loss: 0.013421, Test Loss: 0.002109, Accuracy: 98.17%

Experiment: ResNet-34_LR_0.001_BS_64, Epoch: 0, Train Loss: 0.010685, Test Loss: 0.002099, Accuracy: 96.99%
Experiment: ResNet-34_LR_0.001_BS_128, Epoch: 0, Train Loss: 0.008780, Test Loss: 0.002508, Accuracy: 94.06%
```

对实验数据进行整理后，得下表：

Model	LR	BS	Train Loss	Test Loss	Accuracy (%)
ResNet-10	0.01	32	0.014376	0.003428	97.30
ResNet-10	0.01	64	0.011155	0.012467	72.58
ResNet-10	0.01	128	0.008633	0.003719	90.33
ResNet-10	0.1	32	0.005000	0.001647	98.55
ResNet-10	0.1	64	0.003197	0.001809	96.72
ResNet-10	0.1	128	0.002303	0.000996	96.33
ResNet-10	0.001	32	0.054815	0.039003	78.50
ResNet-10	0.001	64	0.030739	0.026549	61.63
ResNet-10	0.001	128	0.016604	0.015414	45.65
ResNet-18	0.01	32	0.006458	0.001445	98.55
ResNet-18	0.01	64	0.004882	0.001073	98.23
ResNet-18	0.01	128	0.003899	0.002089	92.11
ResNet-18	0.1	32	0.003643	0.001909	98.08
ResNet-18	0.1	64	0.001934	0.001573	96.54
ResNet-18	0.1	128	0.001315	0.000524	97.97
ResNet-18	0.001	32	0.030604	0.007563	96.47
ResNet-18	0.001	64	0.023638	0.011184	91.96
ResNet-18	0.001	128	0.014631	0.011856	69.00

Model	LR	BS	Train Loss	Test Loss	Accuracy (%)
ResNet-34	0.01	32	0.003694	0.002425	97.28
ResNet-34	0.01	64	0.002314	0.000934	97.88
ResNet-34	0.01	128	0.001637	0.000333	98.64
ResNet-34	0.1	32	0.007217	0.001502	98.66
ResNet-34	0.1	64	0.004301	0.000723	98.59
ResNet-34	0.1	128	0.003043	0.000546	97.81
ResNet-34	0.001	32	0.013421	0.002109	98.17
ResNet-34	0.001	64	0.010685	0.002099	96.99
ResNet-34	0.001	128	0.008780	0.002508	94.06

进行数据分析：

从上表中可以看出，不同的学习率（Learning Rate，LR）和批量大小（Batch Size，BS）对 ResNet-10、ResNet-18 和 ResNet-34 模型的训练和测试性能都有显著影响。

（一）学习率的影响：

1. **LR=0.1**：当学习率为 0.1 时，模型通常能在第一个 epoch 达到较低的训练和测试损失，同时准确率也较高。例如，ResNet-34 在学习率为 0.1 和 BS=32 时，测试准确率达到 98.66%。
2. **LR=0.01**：学习率为 0.01 时，模型的性能稍低于 0.1，但仍能取得较好的结果。例如，ResNet-34 在 LR=0.01 和 BS=128 时，测试准确率达到 98.64%。
3. **LR=0.001**：学习率为 0.001 时，模型的训练和测试损失较高，准确率相对较低。这表明学习率过低可能导致模型训练不充分。例如，ResNet-18 在 LR=0.001 和 BS=128 时，测试准确率仅为 69%。

（二）批量大小的影响：

1. 批量大小对训练和测试性能的影响较为复杂。一般来说，较大的批量大小（如 128）能在某些情况下提高准确率和降低损失，但也可能导致测试损失的波动加大。例如，ResNet-10 在 LR=0.1 和 BS=128 时的测试损失最低为 0.000996，而在 LR=0.1 和 BS=32 时测试损失较高为 0.001647。
2. 小批量大小（如 32）有时可以达到较高的测试准确率，例如 ResNet-18 在 LR=0.01 和 BS=32 时测试准确率为 98.55%。

（三）不同模型的影响：

1. ResNet-10

- **LR=0.1, BS=32**：训练损失最低，测试损失较低，准确率最高（98.55%）。
- **LR=0.01, BS=32**：训练损失和测试损失较低，准确率也较高（97.30%）。
- **LR=0.001**：准确率显著下降，特别是 BS=128 时，准确率最低（45.65%）。

2. ResNet-18

- **LR=0.01, BS=32**：测试损失较低，准确率最高（98.55%）。
- **LR=0.1, BS=32**：训练损失和测试损失都较低，准确率也较高（98.08%）。
- **LR=0.001, BS=128**：测试损失高，准确率较低（69.00%）。

3. ResNet-34

- **LR=0.1, BS=32**：训练和测试损失较低，准确率最高（98.66%）。
- **LR=0.01, BS=128**：训练和测试损失最低，准确率也很高（98.64%）。
- **LR=0.001**：准确率相对较高，但低于 0.1 和 0.01 的学习率。

- ResNet-10:
  - 表现较为稳定，在较高学习率（0.1）时表现最佳，适合中等大小的批量（32 或 64）。
  - 在低学习率（0.001）时，表现明显下降，尤其是在大批量（128）时。
- ResNet-18:
  - 在不同的超参数设置下，表现也相对稳定，尤其是在高学习率（0.1）和中等大小批量（32）时表现最好。
  - 低学习率（0.001）下，表现也比 ResNet-10 好，但在大批量（128）时表现不佳。
- ResNet-34:
  - 在所有模型中表现最好，特别是在高学习率（0.1）和中等大小批量（32 或 64）时，测试准确率最高（98.66%）。
  - 即使在低学习率（0.001）下，表现也相对较好，适合于不同的批量大小。

结论

- 总体来看，学习率对模型的影响比批量大小更为显著。较高的学习率（如 0.1）通常能在初始训练阶段取得较低的损失和较高的准确率，而较低的学习率（如 0.001）可能导致训练不足。批量大小的影响则较为复杂，需要根据具体任务进行调试。综合考虑训练和测试损失以及准确率，推荐在学习率 0.1 和适当的批量大小（如 32 或 64）下进行模型训练。
- **ResNet-34** 是所有模型中最优的，特别是在高学习率（0.1）和中等大小批量（32 或 64）时，能够取得最高的准确率和最低的损失。
- **ResNet-18** 次之，在相似的超参数设置下，表现也相当不错。
- **ResNet-10** 在学习率为 0.1 时表现良好，但在低学习率和大批量下表现不佳。

建议

- **ResNet-34** 在训练和测试时是首选模型，尤其是在高学习率（0.1）和中等大小批量（32 或 64）下。
- **ResNet-18** 也是一个不错的选择，适合在资源有限的情况下使用。
- 对于快速实验，可以选择 **ResNet-10**，但应注意调整学习率和批量大小，以避免训练不足或过拟合。