5 PyTorch

徐辉，xuh@fudan.edu.cn

本章学习目标：

- ⋆⋆⋆ 熟悉 PyTorch 软件栈的基本结构，能够阅读并理解基于 PyTorch 编写的模型代码；
- ⋆⋆⋆ 掌握 CNN 模型的基本原理与实现方法，能够使用 PyTorch 构建并实现 CNN 模型；
- ⋆⋆⋆ 掌握 RNN 模型的基本原理与实现方法，能够使用 PyTorch 构建并实现 RNN 模型。

5.1 PyTorch 软件栈

PyTorch[1] 是一个开源的深度学习框架，由 Facebook AI Research（FAIR）团队主导开发。其软件栈设计强调灵活性、易用性与高性能，广泛应用于学术研究与工业部署。从底层硬件到高层 API，PyTorch 构建了一个层次清晰、模块解耦的软件体系，使得开发者既能快速原型设计，也能高效部署模型。

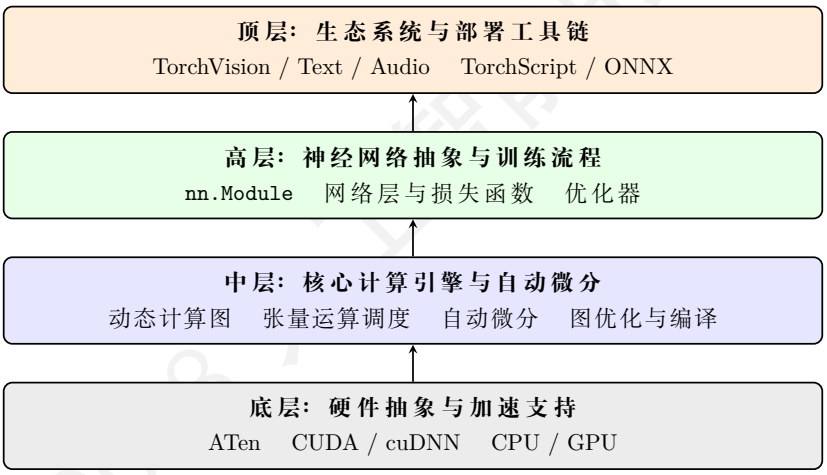

图 5.1: PyTorch 软件栈的分层结构示意图

如图 5.1所示，整个软件栈可大致分为四个层次：

- **底层：硬件抽象与加速支持**。该层直接面向计算硬件，负责张量存储、内存管理以及并行计算的具体实现。通过统一的后端抽象机制，框架能够在 CPU、GPU 及多种专用加速器上提供一致的张量操作接口，并充分利用诸如 CUDA、cuDNN、cuBLAS 等高度优化的底层库，以实现高性能计算。
  - ATen：提供统一的张量接口与算子调度机制；在 GPU 后端实现中，会调用 cuDNN / cuBLAS 等库来完成具体计算；
  - cuDNN/cuBLAS：构建在 CUDA 之上的高性能算子库，提供针对深度学习与线性代数运算的优化实现，供 ATen 在 GPU 后端调用；
  - CUDA：底层的通用 GPU 并行计算平台，是 cuDNN / cuBLAS 等高性能库的运行基础。

[1] PyTorch: `https://pytorch.org`

- **中层：核心计算引擎与自动微分系统**。这一层以计算图为核心，将模型计算统一表示为由张量运算构成的计算图结构。在前向传播过程中，计算图按执行顺序动态构建；在反向传播过程中，基于该计算图自动完成梯度计算，从而支持模型训练。
  - 动态计算图：以计算图为核心抽象，在前向传播过程中动态记录张量之间的运算关系，刻画整个计算流程；
  - 张量运算：构成计算图的基本单元，各类算子作用于张量并产生新的张量，是前向计算的具体实现；
  - 自动微分：基于计算图，通过反向传播机制自动计算各节点的梯度，实现从输出到输入的梯度传递。
  - 图优化与编译（torch.compile）：对计算图进行捕获与优化，并生成高效执行代码，以提升模型训练与推理性能；
- **高层：神经网络抽象与训练机制**。该层面向模型构建与训练过程，提供对神经网络结构、损失函数以及参数优化过程的高层抽象，使用户能够以模块化方式定义模型，并便捷地完成训练与评估。
  - 模型基础抽象（nn.Module）：神经网络模型的基础抽象，通常在实现具体网络时通过继承使用，提供统一的参数管理与运行机制，从而支持模型的模块化构建、组织与训练；
  - 网络层与损失函数：提供常用神经网络层（如卷积层、全连接层等）以及损失函数，用于构建模型并定义优化目标；
  - 优化器：根据计算得到的梯度对模型参数进行更新（如 Adam、SGD 等）。
- **顶层：生态系统与部署工具链**。该层提供丰富的领域扩展库以及模型导出与部署工具，支持模型从开发到实际应用的完整流程。
  - 领域扩展库（TorchVision / TorchText / TorchAudio）：面向计算机视觉、自然语言处理和语音等领域，提供数据集、预处理方法及常用模型；
  - 模型表示与导出（TorchScript / ONNX）：支持将模型转换为中间表示或标准交换格式。其中，TorchScript 用于将模型转化为可脱离 Python 运行的静态表示，以便在不同运行环境中部署；ONNX 用于将模型导出为跨框架的通用格式，从而实现不同深度学习框架之间的模型迁移与部署。

在 PyTorch 中，构建神经网络的核心是继承 `torch.nn.Module` 类并实现其 `forward` 方法。接下来通过两个经典模型讲解如何使用 PyTorch 编写神经网络，包括卷积神经网络（CNN）和循环神经网络（RNN）。

5.2 使用 PyTorch 编写 CNN

卷积神经网络（Convolutional Neural Network, CNN）是一类专门用于处理图像等具有网格结构数据的深度神经网络模型。本节首先介绍 CNN 的核心操作-卷积，然后以 LeNet-5 为例讲解如何使用 PyTorch 编写 CNN 模型。

5.2.1 卷积

图像可以表示为一个像素网格，其中灰度图像的每个像素包含一个强度值，而彩色图像的每个像素则包含对应于红、绿、蓝（RGB）三个通道的三个数值。卷积操作使用小矩阵（卷积核）在输入图像上滑动，并在每一个位置对覆盖区域内的像素进行加权求和，从而实现对局部模式的检测。

形式化地，设输入图像为 I，卷积核 K 的大小为 $h \times w$，则输出特征图 O 定义为：

$$O(i,j) = \sum_{u=0}^{h-1} \sum_{v=0}^{w-1} K(u,v) \cdot I(i+u, j+v).$$

输出中的每一个元素 $O(i,j)$ 都是卷积核与输入图像中对应局部区域之间的点积。卷积核在图像上不断滑动，最终生成输入图像相对该卷积核的特征图。

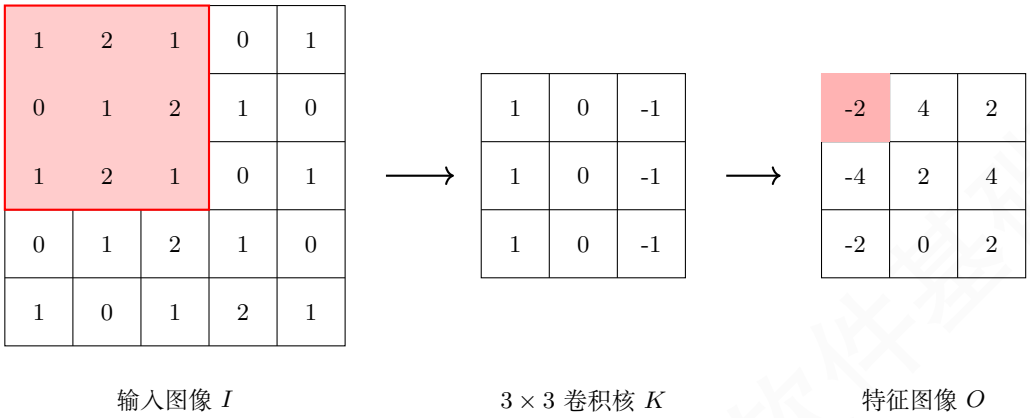

图 5.2: 卷积示例

图 5.2 展示了一个具体示例：一个 5×5 的输入图像与一个 3×3 的卷积核进行卷积运算，最终得到 3×3 的特征图像。

- 左侧输入图像 I 中每个元素为具体的像素值。红色高亮区域表示当前与卷积核进行计算的区域；
- 中间卷积核 K 中每个元素为参数值；
- 右侧特征图 O 中每个元素为卷积核与对应图像区域的点积。红色高亮表示与当前区域对应的结果。

5.2.2 LeNet-5 模型构建

LeNet-5 是由 Yann LeCun 等人在 1998 年提出的经典卷积神经网络，最初用于手写数字识别或 MNIST 数据集。

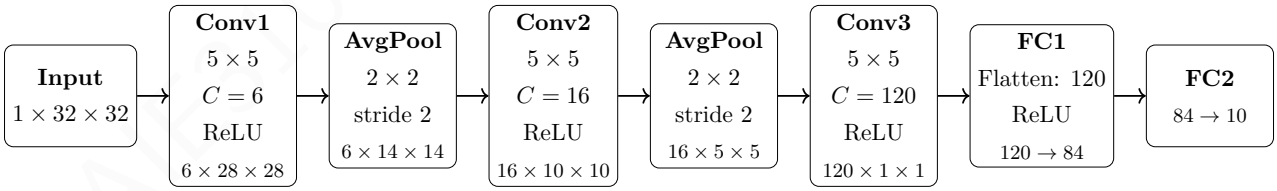

图 5.3: LeNet-5 网络结构

LeNet-5 的主要结构如图 5.3所示。每层的功能和作用如下：

- **输入层 (Input)**: 接受单通道灰度图像，尺寸为 32×32。
- **卷积层 1 (Conv1)**: 使用 5×5 卷积核，将 1 通道映射为 6 个特征通道，并通过 ReLU 激活函数增加非线性表达能力，输出特征图尺寸为 28×28。
- **池化层 1 (AvgPool)**: 使用 2×2 平均池化，步幅为 2，将特征图尺寸从 28×28 缩小到 14×14。
- **卷积层 2 (Conv2)**: 使用 5×5 卷积核，将 6 通道映射到 16 通道，ReLU 激活。输出尺寸为 10×10。

- **池化层 2 (AvgPool)**：使用 2×2 平均池化，步幅为 2，将尺寸从 10×10 缩小到 5×5。
- **卷积层 3 (Conv3)**：使用 5×5 卷积核，将 16 通道映射到 120 通道，输出尺寸 1×1。
- **全连接层 1 (FC1)**：将 Conv3 输出展平为长度 120 的向量，经过 ReLU 激活，映射为 84 维特征。
- **全连接层 2 (FC2)**：将 84 维特征映射到 10 个输出节点，分别对应 10 类分类结果，实现最终分类。

LeNet-5 的设计思想是逐层提取特征：低层卷积捕捉基本边缘和纹理，中层卷积组合局部特征，高层卷积和全连接层整合全局模式。池化层通过下采样提高平移不变性并降低计算量，全连接层将局部特征整合为分类结果。这种结构在手写数字识别和小尺寸图像分类任务中表现优秀。

代码 5.1 展示了基于 PyTorch 的 LeNet-5 实现，主要包括 `__init__` 和 `forward` 两个函数。这两个函数分别定义了网络的层结构和前向传播流程：`__init__` 方法中声明了卷积层（`conv1`、`conv2`、`conv3`）和全连接层（`fc1`、`fc2`），对应图 5.3 中的各层；而 `forward` 方法则实现了数据从输入层流经各卷积层、池化层和全连接层到输出层的过程，完整体现了图中箭头表示的前向传播路径。

```python
import torch
import torch.nn as nn
import torch.nn.functional as F

class LeNet5(nn.Module):
    def __init__(self):
        # 调用父类 (nn.Module) 构造函数
        super(LeNet5, self).__init__()
        # 卷积层: 输入 1 通道 -> 输出 6 通道, 卷积核 5*5
        self.conv1 = nn.Conv2d(1, 6, kernel_size=5)
        # 卷积层: 输入 6 通道 -> 输出 16 通道, 卷积核 5*5
        self.conv2 = nn.Conv2d(6, 16, kernel_size=5)
        # 卷积层: 输入 16 通道 -> 输出 120 通道, 卷积核 5*5
        self.conv3 = nn.Conv2d(16, 120, kernel_size=5)
        # 全连接层: 输入 120 -> 输出 84
        self.fc1 = nn.Linear(120, 84)
        # 全连接层: 输入 84 -> 输出 10 (分类数量)
        self.fc2 = nn.Linear(84, 10)

    def forward(self, x):
        # Conv1 -> ReLU -> AvgPool (2x2)
        x = F.relu(self.conv1(x))
        x = F.avg_pool2d(x, 2)
        # Conv2 -> ReLU -> AvgPool (2x2)
        x = F.relu(self.conv2(x))
        x = F.avg_pool2d(x, 2)
        # Conv3 -> ReLU
        x = F.relu(self.conv3(x))
        # 展平操作, (BatchSize, C, H, W) => (BatchSize, C * H * W)
        x = x.view(x.size(0), -1)
        # 全连接层 FC1 -> ReLU -> FC2
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

代码 5.1: 使用 PyTorch 搭建 LeNet-5

5.2.3 MNIST 数据集准备

模型训练前的一个步骤是准备数据。这里我们使用 LeNet-5 模型解决手写数字识别问题，因此采用经典的 MNIST（Modified National Institute of Standards and Technology）数据集。MNIST 包含 70,000 张灰度图像（60,000 张训练图像和 10,000 张测试图像），每张图像为 28×28 像素，分为 10 类数字（0–9）。

在 PyTorch 中，`torchvision.datasets.MNIST` 提供了对 MNIST 的访问接口，支持在线下载以及本地缓存（如果此前已下载过数据，则会直接使用本地文件）。通过 `download=True` 参数，用户可以控制数据是否自动下载到指定路径。如果是使用自定义数据集，需要通过 PyTorch 中 `torch.utils.data` 模块的 `Dataset` 抽象实现，我们在下一章 RNN 的例子中会使用这种方式。

```python
from torch.utils.data import DataLoader
from torchvision import datasets, transforms

# ----------数据预处理方式----------
# transforms.Compose 用于将多个图像预处理操作按顺序组合
transform = transforms.Compose([
    transforms.Resize((32, 32)),           # 调整图像尺寸以匹配 LeNet-5 输入
    transforms.ToTensor(),                 # 将 PIL 图像转换为 PyTorch Tensor
    transforms.Normalize((0.5,), (0.5,))   # 灰度图像标准化，输出约在 [-1, 1]
])

# ----------数据集定义----------
# datasets.MNIST 是 torchvision 提供的 Dataset 实现。
train_dataset = datasets.MNIST(
    root='./data',                         # 数据集本地存放的根目录
    train=True,                            # 训练集（60,000 张图像）
    download=True,                         # 若本地不存在则自动下载
    transform=transform                    # 样本级预处理流程
)

test_dataset = datasets.MNIST(
    root='./data',                         # 与训练集共用同一数据根目录
    train=False,                           # 测试集（10,000 张图像）
    download=True,
    transform=transform
)

# ----------数据加载器定义----------
# DataLoader 负责将 Dataset 组织成批量数据，并提供随机打乱等机制。
train_loader = DataLoader(
    train_dataset,
    batch_size=64,                         # 训练阶段常用较大的 batch size
    shuffle=True                           # 每个 epoch 开始前打乱样本顺序
)

test_loader = DataLoader(
    test_dataset,
    batch_size=4,
    shuffle=False                          # 测试集不需要打乱顺序
)
```

代码 5.2: MNIST 数据集加载

5.2.4 LeNet-5 模型训练

模型训练过程首先对输入数据进行前向传播以产生预测结果，然后通过损失函数度量预测值与真实标签之间的差异，并利用反向传播算法结合自动微分机制计算模型参数的梯度。随后，优化器根据梯度及其优化策略对模型参数进行更新。PyTorch 提供了损失函数、自动微分以及多种优化器的高效实现，用户只需构建模型结构、选择合适的损失函数与优化器，即可通过迭代式训练循环（多个 epoch）不断优化模型参数。

代码 5.3实现了上述训练流程，并最终将训练得到的模型参数保存至 `lenet5_mnist.pth` 文件中。

```python
import torch
import torch.nn as nn
import torch.optim as optim

# ---------------- 模型、损失函数和优化器 ----------------
model = LeNet5()                            # 实例化 LeNet-5 网络模型
criterion = nn.CrossEntropyLoss()           # 多分类任务常用的交叉熵损失函数
optimizer = optim.Adam(model.parameters(), lr=0.001) # 选取Adam优化器，学习率为0.001

# ---------------- 训练循环 ----------------
num_epochs = 5
for epoch in range(num_epochs):
    model.train()                           # 将模型设置为训练模式
    running_loss = 0.0

    # 遍历 DataLoader 提供的每一个 batch
    for batch_idx, (inputs, labels) in enumerate(train_loader):
        optimizer.zero_grad()               # 清空上一轮迭代的梯度
        outputs = model(inputs)             # 前向传播，计算模型输出
        loss = criterion(outputs, labels) # 计算预测结果与真实标签之间的损失
        loss.backward()                     # 反向传播，计算梯度
        optimizer.step()                    # 根据梯度更新模型参数

        running_loss += loss.item()
        if batch_idx % 100 == 99:           # 每处理 100 个 batch 输出一次平均损失
            print(
                f'Epoch [{epoch+1}/{num_epochs}], '
                f'Batch [{batch_idx+1}], '
                f'Loss: {running_loss/100:.4f}'
            )
            running_loss = 0.0

print("训练完成！")
torch.save(model.state_dict(), "lenet5_mnist.pth") # 保存模型参数
```

代码 5.3: LeNet-5 训练示例

PyTorch 的模型包括 train 和 eval 两种模式，分别用于训练阶段和推理阶段。在 train 模式下，模型中的某些层（如 Dropout 和 Batch Normalization）会启用训练行为；而在 eval 模式下，这些层会切换为推理行为，从而保证模型输出的稳定性。在本实验所实现的 LeNet-5 模型中，由于网络结构未包含 Dropout 或 Batch Normalization 等在训练与推理阶段行为不同的层，因此 `model.train()` 与 `model.eval()` 在功能上无显著差异。

5.2.5 LeNet-5 模型测试

在模型效果测试环节，我们首先加载训练好的模型参数，然后使用前面加载好的 MNIST 测试集验证模型效果。代码 5.4 展示了具体实现过程。

```python
import torch

# ---------------- 创建模型 ----------------
model = LeNet5()                                    # 构造模型结构
model.load_state_dict(torch.load("lenet5_mnist.pth")) # 加载模型参数
model.eval()  # 设置为评估模式，关闭 dropout/batchnorm 等训练专用操作

# ---------------- 前向推理 ----------------
for batch_idx, (input_batch, label_batch) in enumerate(test_loader):
    output_batch = model(input_batch) # 前向传播
    predicted_classes = torch.argmax(output_batch, dim=1) # 获取预测类别

    # 输出结果
    print(f"Batch {batch_idx+1}")
    print("输出 logits:", output_batch)
    print("输出形状:", output_batch.shape)  # [batch_size, 10]
    print("预测类别:", predicted_classes)
    print("真实标签:", label_batch)
    print("-"*40)
```

代码 5.4: 使用 LeNet-5 对 MNIST 数据集进行前向推理

5.3 使用 PyTorch 编写 RNN

5.3.1 RNN 的基本原理

循环神经网络（Recurrent Neural Network, RNN）是一类适合处理序列数据的神经网络。

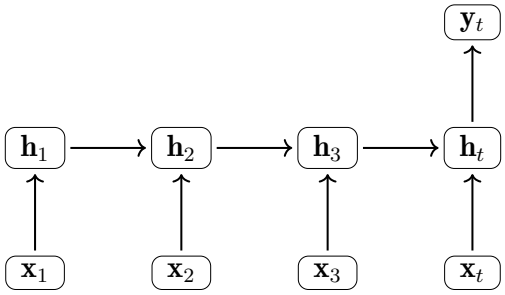

图 5.4: RNN 展开结构示意图

如图 5.4所示，在每一个时间步 t，RNN 会接收当前输入 x_t 和上一时间步的隐藏状态 h_{t-1}，计算当前隐藏状态 h_t：

$$h_t = \tanh(W_x x_t + W_h h_{t-1} + b_h),$$

其中：

- $x_t \in \mathbb{R}^d$ 是当前时间步的输入向量；
- $h_t \in \mathbb{R}^h$ 是当前时间步的隐藏状态；
- $W_x \in \mathbb{R}^{h\times d}$ 和 $W_h \in \mathbb{R}^{h\times h}$ 是可训练权重；
- $b_h \in \mathbb{R}^h$ 是偏置项。

RNN 的输出 y_t 通常通过隐藏状态映射得到：

$$y_t = W_{hy} h_t + b_y.$$

RNN 的关键特点是参数共享，同一组 W_x, W_h, W_y 被重复使用于每个时间步，这使得 RNN 可以处理任意长度的序列。

5.3.2 RNN 模型构建

代码 5.5 展示了如何使用 PyTorch 搭建一个用于回归的单层 RNN 模型。该模型包括一个 RNN 层和一个输出的全连接层。RNN 层是模型的核心，用于处理序列数据，通过隐藏状态 h_t 在时间维上记忆前面输入的信息，从而捕捉序列的动态规律。输出层将序列最后一个时间步的隐藏状态映射为最终的预测值，适用于下一步预测或回归任务。

```python
import torch
import torch.nn as nn

# SimpleRNN 是一个单层 RNN 回归模型，用于预测序列的下一步值
class SimpleRNN(nn.Module):
    def __init__(self, input_size=1, hidden_size=32, output_size=1):
        super(SimpleRNN, self).__init__()
        self.hidden_size = hidden_size

        # RNN 层
        self.rnn = nn.RNN(
            input_size=input_size,
            hidden_size=hidden_size,
            batch_first=True        # 表示输入和输出形状为 (batch, seq_len, input_size)
        )

        # 输出层，将最后一个时间步的 hidden state 映射为预测值
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        # x: (batch_size, seq_len, input_size)
        out, h_n = self.rnn(x)    # out: (batch, seq_len, hidden_size)
        out = out[:, -1, :]       # 取最后一个时间步的输出
        out = self.fc(out)        # 输出预测值
        return out
```

代码 5.5: 使用 PyTorch 搭建 RNN

在 `forward` 方法中，输入张量的形状为 (batch_size, seq_len, input_size)，经过 RNN 层得到每个时间步的隐藏状态后，取最后一个时间步的隐藏状态再通过全连接层输出预测结果。参数 `batch_first=True` 指定输入张量的第 0 维为 batch size，第 1 维为序列长度，第 2 维为特征维度，这样可以让训练循环更直观，减少张量维度混淆的可能。该模型结构简单，非常适合像正弦序列这种平滑、规律的时间序列预测任务。

5.3.3 数据集准备

```python
from torch.utils.data import Dataset, DataLoader
import torch
import math
# ==================================================
# 自定义数据集：SineDataset
# ==================================================
class SineDataset(Dataset):
    """
    用于生成正弦序列数据的 Dataset
    每个样本包含一个长度为 seq_len 的正弦序列 x，以及下一步预测值 y
    可以选择为训练集添加噪声，测试集保持干净
    """
    def __init__(self, seq_len=20, num_samples=1000, noise_std=0.0):
        self.data = []         # 存储输入序列，shape: (seq_len, 1)
        self.targets = []      # 存储目标值，即序列下一步，shape: (1,)
        self.seq_len = seq_len
        self.noise_std = noise_std  # 噪声标准差，训练集可设置非零，测试集设为0

        for _ in range(num_samples):
            start = torch.rand(1).item() * 2 * math.pi  # 随机起点：[0, 2] 之间

            # 生成长度为 seq_len 的正弦序列，并加上高斯噪声
            x = torch.tensor([
                math.sin(start + j*0.1) + noise_std * torch.randn(1).item()
                for j in range(seq_len)
            ])
            y = torch.tensor([math.sin(start + seq_len*0.1)])  # 不加噪声，保持真实值

            self.data.append(x.unsqueeze(-1))   # unsqueeze(-1) 将 x 变为 (seq_len, 1)
            self.targets.append(y)

    def __len__(self):
        """返回样本数量"""
        return len(self.data)

    def __getitem__(self, idx):
        """返回第 idx 个样本"""
        return self.data[idx], self.targets[idx]

# ----------数据加载器定义----------
seq_len = 20    # 序列长度
train_dataset = SineDataset(seq_len=seq_len, num_samples=1000, noise_std=0.02)  # 训练集，加噪声
train_loader = DataLoader(train_dataset, batch_size=16, shuffle=True)
test_dataset = SineDataset(seq_len=seq_len, num_samples=200, noise_std=0.0)  # 测试集，不加噪声
test_loader = DataLoader(test_dataset, batch_size=4, shuffle=False)
```

代码 5.6: 使用 PyTorch 搭建 RNN

5.3.4 模型训练与测试

代码 5.7 展示了如何对我们前面定义的 SimpleRNN 模型进行训练和测试。训练和测试过程使用上一步自定义的 SineDataset 数据集。整体流程与 LeNet-5 的训练和测试类似。

```
model = SimpleRNN()
criterion = nn.MSELoss()         # 回归任务常用均方误差损失
optimizer = optim.Adam(model.parameters(), lr=0.01)

num_epochs = 10
for epoch in range(num_epochs):
    model.train()                # 设置模型为训练模式
    running_loss = 0.0

    for batch_idx, (inputs, targets) in enumerate(train_loader):
        optimizer.zero_grad()              # 清空上一轮梯度
        outputs = model(inputs.float())  # 前向传播
        loss = criterion(outputs, targets.float())  # 计算损失
        loss.backward()                    # 反向传播
        optimizer.step()                   # 参数更新

        running_loss += loss.item()

    # 每个 epoch 输出一次平均损失
    print(f"Epoch [{epoch+1}/{num_epochs}], Loss: {running_loss/len(train_loader):.4f}")

print("训练完成！")

torch.save(model.state_dict(), "simple_rnn_sine.pth") # 保存模型参数

model.eval()  # 切换到评估模式，关闭 dropout / batchnorm 等
test_loss = 0.0

with torch.no_grad():  # 测试阶段不需要计算梯度
    for batch_idx, (inputs, targets) in enumerate(test_loader):
        outputs = model(inputs.float())               # 前向传播
        loss = criterion(outputs, targets.float())    # 计算损失
        test_loss += loss.item()

# 输出平均损失
print(f"测试集平均损失: {test_loss/len(test_loader):.4f}")
```

代码 5.7: RNN 模型训练和测试

练习

1) 相比手写 LeNet、RNN 等神经网络，PyTorch 简化了哪些步骤?

2) 分析 PyTorch 中 `nn.Conv2d`、`nn.Linear` 等算子的实现原理。

3) 改动 LeNet 的卷积核大小以及通道数，观察模型效果的变化。

4) 改动 RNN 的隐藏层维度，观察模型效果的变化。