1.核心基础：张量（Tensor）操作

张量是 PyTorch 的“乐高积木”，所有计算都围绕它展开。这部分一定要练熟！

张量的创建与属性

```
import torch

# 常用创建方法
x = torch.tensor([1, 2, 3])  # 从列表创建
y = torch.zeros(2, 3)        # 全零张量
z = torch.randn(3, 3)        # 正态分布随机张量

# 查看属性
print(x.shape)  # 形状：torch.Size([3])
print(x.dtype)  # 数据类型：torch.int64（默认整数类型）
print(x.device) # 设备：cpu（默认在CPU上）
```

踩坑点：

- 张量默认在 CPU 上，要用 GPU 得手动移过去：`x = x.to('cuda')`（前提是装了 CUDA）；
- 数据类型要统一，比如 `int64` 和 `float32` 不能直接运算，用 `x.float()` 转换。

基本运算（和 NumPy 很像，上手快）

```
a = torch.tensor([1, 2, 3])
b = torch.tensor([4, 5, 6])

# element-wise 运算
print(a + b)  # 等价于 torch.add(a, b)
print(a * b)  # 对应元素相乘（不是矩阵乘法！）

# 矩阵乘法（重点！）
c = torch.randn(2, 3)
d = torch.randn(3, 4)
print(torch.matmul(c, d))  # 结果是 2x4 的矩阵
```

小技巧：
用 `@` 符号代替 `torch.matmul`，代码更简洁：`c @ d`

2.自动求导：让机器自己算梯度

深度学习的核心是反向传播求梯度，PyTorch 的 `autograd` 模块能自动搞定，不用手动推导公式！

核心用法：`requires_grad=True`

```python
x = torch.tensor([2.0], requires_grad=True)  # 标记需要求导
y = x ** 2 + 3 * x + 1

# 反向传播（计算梯度）
y.backward()
print(x.grad)  # 输出 dy/dx 在 x=2 处的值：7.0（导数是 2x+3，代入得7）
```

实战：线性回归求参数

假设我们有一组数据，想拟合 `y = w*x + b`，用自动求导优化参数：

```python
# 模拟数据
x = torch.randn(100, 1)
y = 3 * x + 2 + torch.randn(100, 1) * 0.1  # 真实 w=3, b=2，加了点噪声

# 初始化参数（需要求导）
w = torch.tensor([0.0], requires_grad=True)
b = torch.tensor([0.0], requires_grad=True)

# 梯度下降
lr = 0.1  # 学习率
for _ in range(1000):
    # 前向计算
    y_pred = w * x + b
    loss = torch.mean((y_pred - y) ** 2)  # 均方误差

    # 反向传播（先清零梯度，否则会累加）
    w.grad = None  # 或用 loss.backward(retain_graph=True) 不清零
    b.grad = None
    loss.backward()

    # 更新参数（用 with torch.no_grad() 关闭求导，节省算力）
    with torch.no_grad():
        w -= lr * w.grad
        b -= lr * b.grad

print(f"优化后：w≈{w.item():.2f}, b≈{b.item():.2f}")  # 接近 3 和 2
```

关键点：
每次反向传播前一定要清零梯度，否则梯度会叠加，导致更新错误！

3.神经网络模块：`torch.nn` 快速搭模型

手动写层太麻烦？`torch.nn` 提供了各种现成的层（全连接、卷积、激活函数等），直接拼起来就是模型！

用`nn.Module`定义模型

```python
import torch.nn as nn

class SimpleNet(nn.Module):
```

```
    def __init__(self):
        super().__init__()
        # 定义层：全连接层（输入10维→输出20维）+ ReLU激活
        self.fc1 = nn.Linear(10, 20)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(20, 2)  # 输出2维（比如二分类）

    def forward(self, x):
        # 定义前向传播
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        return x

# 实例化模型
model = SimpleNet()
print(model)  # 打印模型结构
```

损失函数和优化器

```
# 损失函数：二分类用交叉熵
criterion = nn.CrossEntropyLoss()

# 优化器：Adam（比SGD更智能，收敛快）
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
```

训练流程模板（记下来，所有模型都能用）：

```
for epoch in range(epochs):
    # 前向传播
    outputs = model(inputs)
    loss = criterion(outputs, labels)

    # 反向传播+更新参数
    optimizer.zero_grad()  # 清零梯度
    loss.backward()        # 求导
    optimizer.step()       # 更新参数

    if (epoch+1) % 100 == 0:
        print(f"Epoch [{epoch+1}/{epochs}], Loss: {loss.item():.4f}")
```

4.实战：用 PyTorch 1.4 做图像分类（MNIST 数据集）

综合案例：用简单的神经网络识别手写数字（MNIST）

加载数据（用 `torchvision` 现成工具）

```
import torchvision
import torchvision.transforms as transforms
```

```
# 数据预处理：转为张量+归一化
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))  # 均值0.5，标准差0.5
])

# 下载并加载训练集、测试集
trainset = torchvision.datasets.MNIST(
    root='./data', train=True, download=True, transform=transform
)
trainloader = torch.utils.data.DataLoader(
    trainset, batch_size=64, shuffle=True  # 批大小64，打乱数据
)

testset = torchvision.datasets.MNIST(
    root='./data', train=False, download=True, transform=transform
)
testloader = torch.utils.data.DataLoader(testset, batch_size=64, shuffle=False)
```

定义模型（两层全连接）

```
class MNISTNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(28*28, 128)  # 输入是28x28的图像，展平后784维
        self.fc2 = nn.Linear(128, 10)     # 输出10类（0-9）

    def forward(self, x):
        x = x.view(-1, 28*28)  # 展平图像（batch_size, 784）
        x = torch.relu(self.fc1(x))
        x = self.fc2(x)
        return x

model = MNISTNet()
```

训练与测试

```
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

# 训练
for epoch in range(5):  # 训练5轮
    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        inputs, labels = data  # 获取一个批次的数据

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

        running_loss += loss.item()
        if i % 100 == 99:  # 每100个批次打印一次
```

```
            print(f"[{epoch+1}, {i+1}] loss: {running_loss/100:.3f}")
            running_loss = 0.0

# 测试准确率
correct = 0
total = 0
with torch.no_grad():  # 测试时关闭求导，加速
    for data in testloader:
        images, labels = data
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)  # 取概率最大的类别
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print(f"测试集准确率: {100 * correct / total}%")  # 能到97%左右，效果不错!
```