

# PyTorch 入门教程：从 NumPy 到深度学习

Anson, 深度学习社 Cooperated with MiniMax M2 & DeepSeek V3.2 Exp

2025 年 11 月 30 日

## 摘要

本文是 PyTorch 深度学习框架入门教程。即使你只熟悉 Python 基础，也能轻松理解！我们将通过详细的步骤解释和大量的代码对比，帮助你从零开始掌握 PyTorch。文章从 NumPy 出发，逐步介绍为什么需要深度学习框架；然后详细讲解张量（Tensors）、自动求导（Autograd）、神经网络模块（nn.Module）等核心概念；最后通过构建一个手写数字识别器来展示完整的工作流程。每个概念都配有实际代码和运行结果，让你边学边练！

## 目录

<b>1 引言：为什么选择 PyTorch?</b>	<b>2</b>
1.1 从 NumPy 到 PyTorch：一场必然的进化 . . . . .	2
1.2 PyTorch 是什么？ . . . . .	3
1.3 PyTorch vs TensorFlow：两大框架的对决 . . . . .	3
<b>2 张量 (Tensors)：PyTorch 的核心数据结构</b>	<b>5</b>
2.1 什么是张量？ . . . . .	5
2.2 NumPy vs PyTorch：相同的操作，不同的能力 . . . . .	6
2.3 张量操作详解 . . . . .	7
2.4 广播 (Broadcasting)：让不同形状的张量一起运算 . . . . .	8
2.5 张量重塑 (Reshaping)：改变形状但不改变数据 . . . . .	10
2.6 索引和切片 (Indexing and Slicing)：访问张量的特定部分 . . . . .	12
2.7 动手练习：你的第一个 PyTorch 程序 . . . . .	16
<b>3 自动求导 (Autograd)：让梯度计算自动化</b>	<b>18</b>
3.1 为什么需要自动求导？ . . . . .	18
3.2 Autograd 工作原理 . . . . .	19
3.3 Autograd 的使用方法 . . . . .	20

3.4 链式法则: Autograd 的数学基础 . . . . .	20
3.5 控制流: 动态计算图的神奇之处 . . . . .	21
<b>4 神经网络模块 (<code>nn.Module</code>): 构建网络的利器</b>	<b>22</b>
4.1 从零实现到高级 API . . . . .	22
4.2 构建你的第一个神经网络 . . . . .	23
4.3 常见的网络层类型 . . . . .	24
<b>5 优化器: 让网络自己学习</b>	<b>25</b>
5.1 从手工更新到自动优化 . . . . .	25
5.2 常见优化算法对比 . . . . .	26
<b>6 完整训练流程: 从数据到模型</b>	<b>27</b>
<b>7 调试与可视化: 让训练过程透明化</b>	<b>29</b>
7.1 常见错误及解决方案 . . . . .	29
7.2 调试工具和技巧 . . . . .	31
<b>8 PyTorch 生态系统: 超越框架本身</b>	<b>32</b>
8.1 核心组件 . . . . .	32
<b>9 结论: PyTorch 开启深度学习之旅</b>	<b>33</b>

# 1 引言：为什么选择 PyTorch?

## 1.1 从 NumPy 到 PyTorch：一场必然的进化

如果你已经熟悉 Python 和 NumPy，那么恭喜你！你已经掌握了深度学习的重要基础。NumPy 是 Python 中进行科学计算的强大工具，让我们能够高效地处理多维数组。但当你尝试用 NumPy 构建深度神经网络时，会遇到一些挑战：

### NumPy 训练神经网络的挑战

1. **手动实现反向传播**: 每当你修改网络结构，都需要重新推导和实现梯度计算
2. **缺乏自动优化**: 你需要手动实现 SGD、Adam 等优化算法
3. **无法利用 GPU**: NumPy 的运算默认在 CPU 上，无法利用 GPU 的并行计算能力
4. **缺少神经网络层**: 从零实现卷积层、循环层等工作量巨大

### 核心问题

每次调整网络结构（比如加一层），都需要重新推导数学公式和计算梯度。对于复杂的网络（比如识别猫的图片），手动计算梯度几乎是不可能的！

PyTorch 就是来解决这些问题的！它就像 NumPy 的“智能升级版”，专为深度学习设计，但保留了 NumPy 的直观性。

### 特别说明

如果你对 NumPy 还不熟悉，别担心！我们会在教程中详细对比 NumPy 和 PyTorch，让你同时学习两个工具。记住：PyTorch 的语法和 NumPy 非常相似，学会一个，另一个就很容易理解！

## 1.2 PyTorch 是什么？

### PyTorch 的定义

PyTorch 是一个开源的深度学习框架，由 Facebook（现 Meta）的研究团队开发。它提供了一种灵活、高效的方式来构建、训练和部署神经网络。

**核心特点：**

- **语法简单：**就像 NumPy 的“升级版”，学习起来很自然
- **调试方便：**可以像普通 Python 代码一样调试，哪里出错改哪里
- **GPU 加速：**一行代码就能用显卡加速计算，速度提升 10-100 倍！
- **社区活跃：**遇到问题随时可以找到解决方案
- **研究首选：**大多数 AI 论文都用 PyTorch 实现

### PyTorch vs NumPy 对比

- **NumPy：**提供基础的数组操作、数学函数和计算功能。你可以实现任何算法，但需要手动处理梯度计算和优化
- **PyTorch：**提供智能的深度学习工具——自动求导、预定义网络层、预训练模型。它保留了 NumPy 的所有功能，但让深度学习变得更简单、更可靠

### 好消息！

- **不需要高深数学：**PyTorch 会自动计算所有复杂的导数
- **不需要硬件知识：**一行代码就能用 GPU 加速
- **不需要从头开始：**有很多预训练模型可以直接使用
- **学习曲线平缓：**从简单项目开始，逐步深入

## 1.3 PyTorch vs TensorFlow：两大框架的对决

很多初学者会问：应该学 PyTorch 还是 TensorFlow？让我们用简单的方式来对比：

对比维度	PyTorch	TensorFlow
学习难度	简单，像写 Python 代码	初期较复杂，需要理解概念
调试方式	容易，像普通 Python 调试	调试较困难，需要特殊工具
代码风格	直观自然，所见即所得	需要先定义再执行
研究使用度	学术界首选（论文多）	工业界应用广泛
GPU 使用	一行代码切换 CPU/GPU	需要复杂设置
适合人群	初学者、研究人员	工程师、部署专家

表 1: PyTorch vs TensorFlow 对比

### 如何选择？

- 选择 PyTorch 如果你是：
  1. 深度学习初学者，想快速上手
  2. 喜欢动手实验，需要灵活调试
  3. 想读 AI 论文，大多数论文用 PyTorch 实现
  4. 参加竞赛，需要快速迭代模型
- 选择 TensorFlow 如果你是：
  1. 想开发手机 App，需要移动端部署
  2. 团队项目，已有 TensorFlow 基础设施
  3. 喜欢可视化，想看训练过程动画
  4. 生产环境，需要稳定部署

建议：从 PyTorch 开始！理由：

- 学习更简单：语法直观，调试方便
- 资源更丰富：大多数教程和论文都用 PyTorch
- 未来更广阔：学术界和工业界都在转向 PyTorch
- 转换更容易：学会 PyTorch 后，TensorFlow 也不难

### 重要提醒

不要纠结于选择哪个框架！掌握深度学习概念比掌握特定框架更重要。PyTorch 只是帮助你实现想法的工具，就像画笔对于画家一样。

## 2 张量 (Tensors): PyTorch 的核心数据结构

### 2.1 什么是张量?

张量听起来很复杂，但其实很简单！在深度学习中，我们处理的数据可能是：

- 一个数字（标量，0维张量）- 比如温度：25°C
- 一个向量（1维张量）- 比如学生成绩：[85, 92, 78]
- 一个矩阵（2维张量）- 比如班级成绩表
- 更高维度（如图像：3维，批量图像：4维）

#### 张量的定义

张量 (Tensor) 就是 PyTorch 中的“多维数组”。它就像 NumPy 的 ndarray，但有两个超能力：

**超能力 1：**自动计算梯度（后面会详细讲）**超能力 2：**可以用 GPU 加速计算  
**维度对应（实际例子）：**

- **0维张量**：一个数字，比如考试成绩：95
- **1维张量**：一行数字，比如一周温度：[20, 22, 19, 21, 23]
- **2维张量**：一个表格，比如班级成绩表
- **3维张量**：一张彩色图片（高度 × 宽度 × 颜色通道）
- **4维张量**：一批图片（图片数量 × 高度 × 宽度 × 颜色通道）

#### 记住这个类比！

- 标量（0维）：一个点
- 向量（1维）：一条线
- 矩阵（2维）：一个平面
- 张量（3维+）：一个立体空间

就像从点到线，再到面，最后到立体空间一样自然！

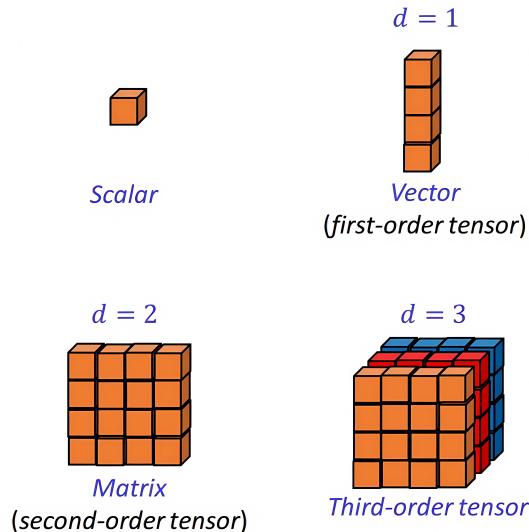


图 1: 不同维度的张量可视化 [1]

## 2.2 NumPy vs PyTorch: 相同的操作，不同的能力

让我们通过实际代码对比来理解 PyTorch 张量。你会发现它们非常相似！

**NumPy 代码** (你可能已经熟悉):

```

1 import numpy as np
2
3 # 创建数组
4 a = np.array([1, 2, 3])          # 向量 [1, 2, 3]
5 b = np.zeros((3, 3))            # 3x3 的零矩阵
6 c = np.random.rand(2, 2)        # 2x2 的随机矩阵
7
8 # 运算
9 d = a + 1                      # 每个元素加1
10 e = np.dot(a, a)               # 点积

```

**PyTorch 代码** (新的，但很相似):

```

1 import torch
2
3 # 创建张量
4 a = torch.tensor([1, 2, 3])      # 向量 [1, 2, 3]
5 b = torch.zeros(3, 3)            # 3x3 的零矩阵
6 c = torch.rand(2, 2)             # 2x2 的随机矩阵
7
8 # 运算
9 d = a + 1                      # 每个元素加1

```

```
10 e = torch.dot(a, a)          # 点积  
11  
12 # PyTorch 的超能力!  
13 device = torch.device('cuda')    # 使用 GPU  
14 c_gpu = c.to(device)          # 把张量移到 GPU
```

重要发现：语法几乎一样！

- 创建数组：`np.array()` → `torch.tensor()`
- 零矩阵：`np.zeros()` → `torch.zeros()`
- 随机矩阵：`np.random.rand()` → `torch.rand()`
- 数学运算：完全一样的语法！

最大的区别：PyTorch 支持 GPU 计算和自动求导，这是 NumPy 没有的超能力！

### 练习建议

1. 打开 Python 环境（Jupyter Notebook 或 Python 终端）
2. 先运行 NumPy 代码，确保理解每行代码的作用
3. 再运行 PyTorch 代码，对比输出结果
4. 尝试修改：改变数字、矩阵大小，观察变化
5. 挑战：创建一个 3x3 的矩阵，计算它的转置

## 2.3 张量操作详解

让我们逐一探索常用的张量操作。这些操作在 NumPy 和 PyTorch 中非常相似！

## 基础操作对比（边学边练）

### 1. 查看形状和维度

NumPy	PyTorch
<code>a.shape</code>	<code>a.shape</code>
<code>a.ndim</code>	<code>a.dim()</code>

实际例子：

```
1 # NumPy
2 a = np.array([[1, 2], [3, 4]])
3 print(a.shape) # 输出: (2, 2)
4 print(a.ndim) # 输出: 2
5
6 # PyTorch
7 a = torch.tensor([[1, 2], [3, 4]])
8 print(a.shape) # 输出: torch.Size([2, 2])
9 print(a.dim()) # 输出: 2
```

### 2. 数学运算（完全一样！）

NumPy	PyTorch
<code>np.sum(a)</code>	<code>torch.sum(a)</code>
<code>np.mean(a)</code>	<code>torch.mean(a)</code>
<code>np.max(a)</code>	<code>torch.max(a)</code>

### 3. 矩阵运算（几乎一样）

NumPy	PyTorch
<code>a @ b 或 np.dot(a, b)</code>	<code>torch.matmul(a, b)</code>
<code>a.T</code>	<code>a.T 或 a.t()</code>

## 2.4 广播（Broadcasting）：让不同形状的张量一起运算

广播是 PyTorch 和 NumPy 中的一个重要概念，它允许我们对不同形状的张量进行数学运算。这就像数学中的“自动扩展”功能！

## 什么是广播？

广播是一种机制，允许 PyTorch 自动扩展较小张量的形状，使其与较大张量的形状兼容，从而进行逐元素运算。

广播规则：

1. **从尾部对齐**：从最后一个维度开始比较
2. **维度为 1**：维度为 1 的轴会被扩展以匹配其他张量
3. **缺失维度**：缺失的维度被视为 1
4. **其他情况**：如果维度不匹配且都不是 1，则报错

## 广播的实际例子

```
1 import torch
2
3 # 例子1：标量与向量相加
4 a = torch.tensor([1, 2, 3])    # 形状: (3,)
5 b = 10                         # 形状: 标量
6 c = a + b                      # 广播: 10 [10, 10, 10]
7 print(c) # 输出: tensor([11, 12, 13])
8
9 # 例子2：向量与矩阵相加
10 matrix = torch.tensor([[1, 2, 3],
11                      [4, 5, 6]])   # 形状: (2, 3)
12 vector = torch.tensor([10, 20, 30]) # 形状: (3,)
13 result = matrix + vector         # 广播: vector []
14
15 print(result)
16 # 输出: tensor([[11, 22, 33],
17 #                 [14, 25, 36]])
18
19 # 例子3：不同形状的矩阵运算
20 A = torch.ones(3, 1, 4)  # 形状: (3, 1, 4)
21 B = torch.ones(2, 4)     # 形状: (2, 4)
22 C = A + B              # 广播: A [3, 2, 4], B [3, 2, 4]
23 print(C.shape)          # 输出: torch.Size([3, 2, 4])
```

## 广播的应用场景

- **归一化**: 从每个元素减去均值，除以标准差
- **批量运算**: 对批量数据应用相同的操作
- **权重更新**: 用标量学习率更新所有参数
- **特征缩放**: 对特征进行统一的缩放处理

## 2.5 张量重塑 (Reshaping): 改变形状但不改变数据

在深度学习中，我们经常需要改变张量的形状。PyTorch 提供了多种方法来重塑张量：

### 常用的重塑方法

- `view()`: 返回相同数据的新视图（不复制数据）
- `reshape()`: 返回重塑后的张量（可能复制数据）
- `permute()`: 重新排列维度顺序
- `transpose()`: 交换两个维度
- `squeeze()`: 移除长度为 1 的维度
- `unsqueeze()`: 添加长度为 1 的维度

## 张量重塑的实际例子

```
1 import torch
2
3 # 创建一个 2x3x4 的张量
4 original = torch.randn(2, 3, 4)
5 print(f"原始形状: {original.shape}") # torch.Size([2, 3, 4])
6
7 # 使用 view 改变形状 (不复制数据)
8 flattened = original.view(2, -1) # -1 表示自动计算该维度大小
9 print(f"展平后: {flattened.shape}") # torch.Size([2, 12])
10
11 # 使用 reshape 改变形状
12 reshaped = original.reshape(6, 4)
13 print(f"重塑后: {reshaped.shape}") # torch.Size([6, 4])
14
15 # 交换维度
16 transposed = original.transpose(0, 1) # 交换第 0 维和第 1 维
17 print(f"转置后: {transposed.shape}") # torch.Size([3, 2, 4])
18
19 # 重新排列维度顺序
20 permuted = original.permute(2, 0, 1) # 新顺序: 原第 2 维 → 第 0 维,
21     原第 0 维 → 第 1 维, 原第 1 维 → 第 2 维
22 print(f"重排后: {permuted.shape}") # torch.Size([4, 2, 3])
23
24 # 添加和移除维度
25 tensor_1d = torch.tensor([1, 2, 3])
26 tensor_2d = tensor_1d.unsqueeze(0) # 在第 0 维添加维度
27 print(f"添加维度后: {tensor_2d.shape}") # torch.Size([1, 3])
28
29 tensor_1d_again = tensor_2d.squeeze(0) # 移除第 0 维
30 print(f"移除维度后: {tensor_1d_again.shape}") # torch.Size([3])
```

## view() vs reshape() 区别

- `view()`: 要求张量在内存中是连续的, 不复制数据, 更快
- `reshape()`: 总是返回所需形状, 必要时会复制数据, 更安全
- **建议:** 优先使用 `reshape()`, 除非确定张量是连续的

## 2.6 索引和切片 (Indexing and Slicing): 访问张量的特定部分

索引和切片是访问和修改张量特定元素的重要操作。PyTorch 的索引语法与 NumPy 非常相似！

### 基本的索引操作

- 整数索引：访问特定位置的元素
- 切片索引：访问一个范围内的元素
- 布尔索引：使用布尔条件选择元素
- 高级索引：使用整数数组索引

索引和切片的实际例子：

```
1 import torch
2
3 # 创建一个3x4的矩阵
4 matrix = torch.tensor([[1, 2, 3, 4],
5                      [5, 6, 7, 8],
6                      [9, 10, 11, 12]])
7 print(f"原始矩阵:\n{matrix}")
8
9 # 整数索引
10 first_row = matrix[0]           # 第0行
11 first_element = matrix[0, 0]     # 第0行第0列
12 print(f"第0行: {first_row}")    # tensor([1, 2, 3, 4])
13 print(f"第0行第0列: {first_element}") # tensor(1)
14
15 # 切片索引
16 first_two_rows = matrix[:2]      # 前2行
17 middle_columns = matrix[:, 1:3]   # 所有行的第1-2列
18 print(f"前2行:\n{first_two_rows}")
19 print(f"中间列:\n{middle_columns}")
20
21 # 布尔索引
22 mask = matrix > 5               # 创建布尔掩码
23 selected = matrix[mask]         # 选择大于5的元素
24 print(f"大于5的元素: {selected}") # tensor([6, 7, 8, 9, 10, 11,
25                                12])
```

```
26 # 高级索引
27 rows = torch.tensor([0, 2])      # 选择第0行和第2行
28 cols = torch.tensor([1, 3])      # 选择第1列和第3列
29 selected_elements = matrix[rows, cols]
30 print(f"选择的行列: {selected_elements}") # tensor([2, 12])
31
32 # 修改元素
33 matrix[0, 0] = 100             # 修改单个元素
34 matrix[1] = torch.tensor([50, 60, 70, 80]) # 修改整行
35 print(f"修改后的矩阵:\n{matrix}")
```

## 索引注意事项

- **视图 vs 副本**: 大多数索引操作返回视图 (不复制数据)
- **原地修改**: 使用索引修改会原地改变原始张量
- **梯度跟踪**: 索引操作会保持梯度跟踪
- **性能**: 避免在循环中使用复杂索引

## 张量的高级功能 (PyTorch 的超能力!)

除了 NumPy 的所有功能, PyTorch 张量还提供:

- **自动求导**: 设置 `requires_grad=True` 自动跟踪梯度
- **GPU 加速**: 使用 `.to(device)` 在 CPU/GPU 间切换
- **梯度累积**: 自动计算梯度并更新参数
- **内存共享**: 与 NumPy 共享内存 (CPU 张量)
- **广播机制**: 自动处理不同形状张量的运算
- **灵活重塑**: 多种方法改变张量形状
- **强大索引**: 灵活的索引和切片操作

实际例子:

```
1 # 自动求导示例
2 x = torch.tensor(3.0, requires_grad=True)
3 y = x**2 + 2*x + 1
4 y.backward()
5 print(x.grad) # 自动计算导数: 2*x + 2 = 8.0
```

### 学习提示

- **不要死记硬背**: 大多数操作和 NumPy 一样, 用的时候查文档就行
- **多动手实验**: 在 Python 环境中尝试不同的操作
- **理解概念**: 重点是理解张量是什么, 而不是记住所有函数
- **循序渐进**: 先掌握基础操作, 再学习高级功能

## 实战：NumPy vs PyTorch 图像处理

假设我们有一张  $28 \times 28$  的灰度图像（MNIST 数字）：

**NumPy 实现：**

```
1 import numpy as np
2
3 # 加载图像 (28x28)
4 image = np.load('digit.npy')
5
6 # 展平为向量
7 pixels = image.flatten() # 784 维
8
9 # 标准化
10 normalized = (pixels - 128) / 128
11
12 # 添加批次维度
13 batch = normalized.reshape(1, -1)
```

**PyTorch 实现：**

```
1 import torch
2 import torchvision.transforms
3
4 # 加载图像 (28x28)
5 image = torch.load('digit.pt')
6
7 # 展平为向量
8 pixels = image.flatten() # 784 维
9
10 # 标准化
11 normalized = (pixels - 0.1307) / 0.3081
12
13 # 添加批次维度
14 batch = normalized.view(1, -1)
15
16 # 轻松切换到 GPU !
17 device = torch.device('cuda')
18 batch = batch.to(device)
```

## PyTorch vs NumPy 核心优势

虽然 NumPy 和 PyTorch 语法相似，但 PyTorch 提供了深度学习的关键功能：

- **自动求导**: 一行代码 `loss.backward()` 自动计算所有梯度
- **GPU 加速**: GPU 上 PyTorch 比 CPU NumPy 快 10-100 倍
- **神经网络模块**: 预定义层、损失函数、优化器
- **动态计算图**: 支持 Python 控制流，调试方便

## 2.7 动手练习：你的第一个 PyTorch 程序

现在让我们通过一个完整的练习来巩固所学知识！

### 练习：计算二次函数的导数

**任务**：计算函数  $f(x) = x^2 + 3x + 2$  在  $x = 2$  处的导数

**步骤**：

1. 创建需要梯度的张量
2. 计算函数值
3. 自动求导
4. 查看结果

**完整代码**：

```
1 import torch
2
3 # 1. 创建需要梯度的张量
4 x = torch.tensor(2.0, requires_grad=True)
5
6 # 2. 计算函数值
7 y = x**2 + 3*x + 2
8
9 # 3. 自动求导
10 y.backward()
11
12 # 4. 查看结果
13 print(f"x = {x.item()}")
```

```
14 print(f"y = {y.item()}")
15 print(f"导数 dy/dx = {x.grad.item()}")
16
17 # 手动验证：导数应该是 2*x + 3 = 2*2 + 3 = 7
18 print(f"手动验证：2*x + 3 = {2*x.item() + 3}")
```

预期输出：

```
1 x = 2.0
2 y = 12.0
3 导数 dy/dx = 7.0
4 手动验证：2*x + 3 = 7.0
```

## 扩展练习

尝试修改代码完成以下任务：

1. 简单：计算  $f(x) = x^3$  在  $x = 3$  处的导数
2. 中等：计算  $f(x) = \sin(x)$  在  $x = \pi/4$  处的导数
3. 高级：计算  $f(x, y) = x^2 + y^2$  在  $(x = 2, y = 3)$  处的偏导数

提示：

- 多个变量需要设置多个张量的 `requires_grad=True`
- 使用 `torch.sin()` 计算正弦函数
- 使用 `torch.pi` 获取  $\pi$  的值

## 学习建议

- 不要只看不练：一定要在电脑上运行这些代码
- 遇到错误别怕：错误是最好的学习机会
- 多尝试修改：改变数字、函数，观察变化
- 记录结果：把运行结果和理解记下来

## 3 自动求导 (Autograd): 让梯度计算自动化

### 3.1 为什么需要自动求导?

在深度学习中，我们通过梯度下降法更新网络参数。手动计算梯度不仅容易出错，而且非常耗时。想象一下：

#### 手动计算梯度的挑战

对于一个简单的 3 层网络：

$$y = W_3(W_2(W_1x + b_1) + b_2) + b_3 \quad (1)$$

要计算  $\frac{\partial y}{\partial W_1}$ ，你需要：

1. 使用链式法则展开
2. 计算每个中间变量的导数
3. 手动实现每个步骤
4. 每次修改网络结构都要重新推导！

对于 ResNet、Transformer 等复杂网络，这几乎是不可能的！

#### 自动求导的价值

PyTorch 的 Autograd 自动计算所有梯度。你只需要写前向传播代码，反向传播交给 Autograd 处理！

## 3.2 Autograd 工作原理

### 计算图 (Computational Graph)

Autograd 通过构建计算图来跟踪操作序列。计算图是一个有向无环图 (DAG) , 记录了数据如何通过操作流动。

节点:

- 叶子节点: 张量 (通常是输入和参数)
- 中间节点: 操作 (如加法、乘法、矩阵乘法)
- 根节点: 最终结果

边: 表示数据流向

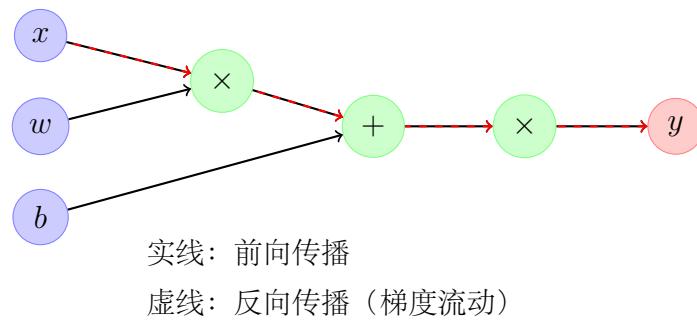


图 2: 计算图示意图

### 3.3 Autograd 的使用方法

最简单的例子:  $y = x^2$

让我们从一个简单例子开始:

```
1 import torch
2
3 # 创建需要梯度的张量
4 x = torch.tensor(2.0, requires_grad=True)
5
6 # 前向计算
7 y = x ** 2
8
9 # 反向传播
10 y.backward()
11
12 # 查看梯度
13 print(f"x = {x.item()}")
14 print(f"y = {y.item()}")
15 print(f"dy/dx = {x.grad.item()}") # 应该是 2*x = 4.0
```

#### 关键函数

- `requires_grad=True`: 告诉 PyTorch 需要计算这个张量的梯度
- `.backward()`: 从当前张量开始反向传播, 计算所有依赖张量的梯度
- `.grad`: 存储计算得到的梯度
- `torch.no_grad()`: 上下文管理器, 在此范围内的操作不会被跟踪 (用于参数更新)

### 3.4 链式法则: Autograd 的数学基础

PyTorch 使用链式法则自动计算梯度。让我们手动推导一个例子:

```
1 # 链式法则示例: z = (x + y)^2
2 x = torch.tensor(1.0, requires_grad=True)
3 y = torch.tensor(2.0, requires_grad=True)
4
5 # 中间变量
```

```

6 u = x + y          # u = 3.0
7 z = u ** 2          # z = 9.0
8
9 # 反向传播
10 z.backward()
11
12 # 手动验证链式法则：
13 # dz/dx = dz/du * du/dx = 2u * 1 = 2*3 = 6.0
14 # dz/dy = dz/du * du/dy = 2u * 1 = 2*3 = 6.0
15
16 print(f"dz/dx = {x.grad.item()}") # 应该是 6.0
17 print(f"dz/dy = {y.grad.item()}") # 应该是 6.0

```

## Autograd 优势

- **自动化**: 无需手动推导梯度公式
- **准确性**: 计算机计算，避免笔误
- **灵活性**: 任何可以用 Python 表达的计算图都能自动求导
- **高效性**: 使用高效的 C++ 后端计算
- **可组合**: 复杂的网络可以组合简单的操作

## 3.5 控制流：动态计算图的神奇之处

PyTorch 的动态计算图允许我们使用 Python 的控制流 (if、for、while) :

```

1 # 动态控制流示例
2 x = torch.tensor(1.0, requires_grad=True)
3
4 # 根据条件选择不同的计算路径
5 if x > 0:
6     y = x ** 2
7 else:
8     y = x ** 3
9
10 y.backward()
11 print(f"x = {x.item()}, y = {y.item()}, dy/dx = {x.grad.item()}")

```

## 静态图 vs 动态图

- 静态图 (TensorFlow 1.x): 先定义整个计算图, 再执行
  - 优点: 可能更高效
  - 缺点: 调试困难, 不支持动态控制流
- 动态图 (PyTorch): 运行时构建计算图
  - 优点: 灵活、易调试、支持 Python 控制流
  - 缺点: 某些优化较困难

最新版本的 TensorFlow (2.x) 也支持动态图了!

## 4 神经网络模块 (nn.Module): 构建网络的利器

### 4.1 从零实现到高级 API

到目前为止, 我们都是直接操作张量和自动求导。但构建复杂神经网络时, 这样做会很繁琐。PyTorch 提供了高级 API 来简化工作。

```
1 # 手动实现 vs nn.Module
2
3 # 手动实现 (繁琐)
4 class ManualNet:
5     def __init__(self, input_size, hidden_size, output_size):
6         self.w1 = torch.randn(input_size, hidden_size, requires_grad=True)
7         self.b1 = torch.randn(hidden_size, requires_grad=True)
8         self.w2 = torch.randn(hidden_size, output_size, requires_grad=True)
9         self.b2 = torch.randn(output_size, requires_grad=True)
10
11     def forward(self, x):
12         h = torch.relu(x @ self.w1 + self.b1)
13         return h @ self.w2 + self.b2
14
15 # 使用nn.Module (简洁)
16 import torch.nn as nn
17
```

```
18 class SimpleNet(nn.Module):
19     def __init__(self, input_size, hidden_size, output_size):
20         super().__init__()
21         self.fc1 = nn.Linear(input_size, hidden_size)
22         self.fc2 = nn.Linear(hidden_size, output_size)
23
24     def forward(self, x):
25         x = torch.relu(self.fc1(x))
26         return self.fc2(x)
```

### 使用 nn.Module 的优势

- 参数管理：自动管理权重和偏置
- 设备管理：自动处理 CPU/GPU 切换
- 状态保存：一键保存/加载模型
- 多种层：预定义卷积层、池化层、RNN 等
- 简洁代码：少写很多样板代码

## 4.2 构建你的第一个神经网络

让我们用 nn.Module 构建一个简单的全连接网络：

```
1 import torch.nn as nn
2 import torch.nn.functional as F
3
4 class SimpleNet(nn.Module):
5     def __init__(self, input_size=784, hidden_size=128, output_size
6                  =10):
7         super().__init__()
8         self.fc1 = nn.Linear(input_size, hidden_size)
9         self.fc2 = nn.Linear(hidden_size, output_size)
10        self.dropout = nn.Dropout(0.2)
11
12    def forward(self, x):
13        # 展平输入 (batch, 784)
14        x = x.view(x.size(0), -1)
15
16        # 第一层 + ReLU + Dropout
```

```

16     x = F.relu(self.fc1(x))
17     x = self.dropout(x)
18
19     # 输出层 (不需要softmax, CrossEntropyLoss会处理)
20     x = self.fc2(x)
21
22
23 # 使用模型
24 model = SimpleNet()
25 print(f"模型参数数量: {sum(p.numel() for p in model.parameters())}")

```

### nn.Module 关键方法

- `__init__(self, ...)`: 初始化层和参数
- `forward(self, x)`: 定义前向传播
- `parameters()`: 返回所有可学习参数
- `to(device)`: 将模型移动到 GPU/CPU
- `state_dict()`: 获得参数字典 (用于保存)
- `load_state_dict(state_dict)`: 加载参数

## 4.3 常见的网络层类型

PyTorch 提供了丰富的预定义层:

层类型	代码示例	用途
全连接层	<code>nn.Linear(784, 256)</code>	图像分类、特征学习
卷积层	<code>nn.Conv2d(1, 32, 3)</code>	图像处理、特征提取
池化层	<code>nn.MaxPool2d(2)</code>	降采样、减少计算
Dropout	<code>nn.Dropout(0.5)</code>	正则化、防止过拟合
批归一化	<code>nn.BatchNorm2d(32)</code>	加速训练、稳定梯度
循环层	<code>nn.LSTM(100, 64)</code>	序列数据、自然语言
嵌入层	<code>nn.Embedding(1000, 50)</code>	词向量、离散特征

表 2: 常见的 PyTorch 网络层

## 形状变化规律

理解张量形状变化是掌握 CNN 的关键：

- 输入: `(batch, 1, 28, 28)` -  $28 \times 28$  灰度图, 批量 2
- 卷积: `Conv2d(1, 32, 3, padding=1) → (batch, 32, 28, 28)`
- 池化: `MaxPool2d(2) → (batch, 32, 14, 14)`
- 展平: `view(-1) → (batch, 6272)`
- 全连接: `Linear(6272, 128) → (batch, 128)`

卷积层保持空间维度, 池化层减半空间维度!

## 5 优化器：让网络自己学习

### 5.1 从手工更新到自动优化

之前我们手动更新参数：

```
1 with torch.no_grad():
2     w -= learning_rate * w.grad
3     b -= learning_rate * b.grad
```

但深度网络有数十万参数, 手动更新不现实! 优化器 (Optimizer) 来帮忙!

#### 优化器职责

- 存储所有参数
- 计算梯度
- 根据优化算法更新参数
- 记录优化历史 (动量、Adam 的历史梯度等)

## 梯度清零的重要性

PyTorch 会累积梯度。如果不清零，梯度会不断累加！

错误示范：

```
1 for data, target in loader:  
2     output = model(data)  
3     loss = criterion(output, target)  
4     loss.backward()  
5     optimizer.step()  
6     # 错误：每次都累积梯度！
```

正确做法：

```
1 for data, target in loader:  
2     optimizer.zero_grad() # 清零！  
3     output = model(data)  
4     loss = criterion(output, target)  
5     loss.backward()  
6     optimizer.step()
```

## 5.2 常见优化算法对比

算法	代码	特点
SGD	SGD(model.parameters(), lr=0.1)	简单但震荡，可能陷入局部最优
SGD + 动量	SGD(model.parameters(), lr=0.1, momentum=0.9)	减少震荡，加速收敛
Adam	Adam(model.parameters(), lr=0.001)	最常用，自适应学习率
RMSprop	RMSprop(model.parameters())	适合 RNN、非平稳目标
AdaGrad	Adagrad(model.parameters())	稀疏数据效果好

表 3: 常见优化算法对比

## 现代实践：AdamW

AdamW 是 Adam 的改进版，解决了权重衰减的问题：

```
1 # AdamW = Adam + 改进的权重衰减
2 optimizer = optim.AdamW(
3     model.parameters(),
4     lr=0.001,
5     weight_decay=0.01,  # 更合理的权重衰减
6     betas=(0.9, 0.999)
7 )
```

## 6 完整训练流程：从数据到模型

现在让我们把 everything together，训练一个完整的 MNIST 分类器：

```
1 import torch
2 import torch.nn as nn
3 import torch.optim as optim
4 from torch.utils.data import DataLoader
5 import torchvision.datasets as datasets
6 import torchvision.transforms as transforms
7
8 # 1. 设置设备
9 device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
10
11 # 2. 数据加载
12 transform = transforms.Compose([
13     transforms.ToTensor(),
14     transforms.Normalize((0.1307,), (0.3081,)))
15 ])
16
17 train_dataset = datasets.MNIST(root='./data', train=True,
18                                download=True, transform=transform)
19 train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
20
21 # 3. 定义模型
22 model = SimpleNet().to(device)
```

```
23
24 # 4. 选择损失函数和优化器
25 criterion = nn.CrossEntropyLoss()
26 optimizer = optim.Adam(model.parameters(), lr=0.001)
27
28 # 5. 训练循环
29 for epoch in range(5):
30     model.train()
31     for batch_idx, (data, target) in enumerate(train_loader):
32         data, target = data.to(device), target.to(device)
33
34         # 前向传播
35         output = model(data)
36         loss = criterion(output, target)
37
38         # 反向传播
39         optimizer.zero_grad()
40         loss.backward()
41         optimizer.step()
42
43         if batch_idx % 100 == 0:
44             print(f'Epoch: {epoch}, Batch: {batch_idx}, Loss: {loss.
item():.4f}')
```

### 训练流程关键步骤

1. **设置设备**: 确定使用 GPU 还是 CPU
2. **数据加载**: 使用 DataLoader 批量加载数据
3. **定义模型**: 使用 nn.Module 构建网络
4. **选择损失函数**: CrossEntropyLoss 适合分类任务
5. **选择优化器**: Adam 是最安全的选择
6. **训练循环**: 前向 → 清零梯度 → 反向 → 更新
7. **评估模型**: 在测试集上验证性能

## 训练技巧

- 数据增强: `transforms.RandomRotation`, `transforms.RandomAffine`
- 学习率调度: `torch.optim.lr_scheduler`
- 早停: 验证损失不再下降时停止训练
- 梯度裁剪: 防止梯度爆炸
- 混合精度训练: 使用 float16 加速训练
- 梯度累积: 模拟大批量训练

# 7 调试与可视化：让训练过程透明化

## 7.1 常见错误及解决方案

作为初学者，你可能会遇到一些常见错误。别担心，这些都很正常！

### 错误 1：忘记导入 PyTorch

错误信息: `NameError: name 'torch' is not defined`

解决方案:

```
1 # 忘记导入!
2 x = torch.tensor([1, 2, 3]) # 错误!
3
4 # 正确做法:
5 import torch
6 x = torch.tensor([1, 2, 3]) # 正确!
```

## 错误 2：忘记清零梯度

错误现象：梯度越来越大，训练不稳定

解决方案：

```
1 # 错误示范
2 for data, target in loader:
3     output = model(data)
4     loss = criterion(output, target)
5     loss.backward()
6     optimizer.step() # 梯度累积!
7
8 # 正确做法
9 for data, target in loader:
10    optimizer.zero_grad() # 清零梯度!
11    output = model(data)
12    loss = criterion(output, target)
13    loss.backward()
14    optimizer.step()
```

## 错误 3：张量形状不匹配

错误信息： RuntimeError: The size of tensor a must match the size of tensor b

解决方案：

```
1 # 检查张量形状
2 print(f"张量1形状: {tensor1.shape}")
3 print(f"张量2形状: {tensor2.shape}")
4
5 # 常见形状问题：
6 # - 矩阵乘法: A.shape = (3,4), B.shape = (4,5) 形状匹配!
7 # - 矩阵乘法: A.shape = (3,4), B.shape = (3,5) 形状不匹配!
```

## 7.2 调试工具和技巧

### 调试技巧

- 打印张量形状: `print(tensor.shape)`
- 检查数据类型: `print(tensor.dtype)`
- 查看梯度: `print(tensor.grad)`
- 检查设备: `print(tensor.device)`
- 使用断言: `assert tensor.shape == expected_shape`

### 调试示例

```
1 import torch
2
3 # 创建张量
4 x = torch.randn(2, 3)
5 y = torch.randn(3, 4)
6
7 # 调试信息
8 print(f"x 形状: {x.shape}") # torch.Size([2, 3])
9 print(f"y 形状: {y.shape}") # torch.Size([3, 4])
10 print(f"x 设备: {x.device}") # cpu
11 print(f"x 数据类型: {x.dtype}") # torch.float32
12
13 # 矩阵乘法
14 z = torch.matmul(x, y)
15 print(f"z 形状: {z.shape}") # torch.Size([2, 4])
```

### 模型保存最佳实践

- 生产环境: 只保存 `state_dict`, 避免版本兼容问题
- 研究环境: 保存完整检查点, 方便恢复训练
- 模型部署: 使用 `torch.jit.script` 转换为 TorchScript
- 云端部署: 考虑 ONNX 格式 (跨框架兼容)

## 调试建议

- **从简单开始**: 先运行小例子, 确保基础正确
- **逐步构建**: 每加一行代码就测试一次
- **善用打印**: 多用 `print()` 查看中间结果
- **理解错误**: 不要只看错误信息, 要理解为什么出错
- **寻求帮助**: 遇到问题先搜索, 再问老师或同学

## 8 PyTorch 生态系统: 超越框架本身

PyTorch 不仅仅是一个框架, 它是一个完整的生态系统:

### 8.1 核心组件

组件	作用
TorchVision	计算机视觉模型和工具
TorchText	自然语言处理
TorchAudio	音频处理
TorchServe	模型部署服务
PyTorch Lightning	简化训练代码
Accelerate	多 GPU/分布式训练

表 4: PyTorch 生态系统核心组件

## 学习 PyTorch 的价值

- **学术研究**: PyTorch 是顶会论文的首选框架, 跟上最新研究
- **工业应用**: Meta、OpenAI、特斯拉等公司都在使用
- **学习曲线**: 语法直观, 快速上手深度学习
- **调试友好**: 动态图让调试像 Python 一样自然
- **社区支持**: 活跃的开发者社区, 丰富的教程和工具
- **未来发展**: PyTorch 2.0 进一步提升性能和易用性

## 学习路径建议

建议的学习路径：

1. **基础阶段**: 熟悉 NumPy, 学习张量操作
2. **核心概念**: 理解自动求导, 完成练习
3. **网络构建**: 使用 nn.Module 构建网络
4. **完整项目**: 完成 MNIST 分类项目
5. **进阶学习**: 学习卷积神经网络, 完成图像分类
6. **序列处理**: 学习循环神经网络, 处理序列数据
7. **迁移学习**: 尝试迁移学习, 使用预训练模型
8. **深入研究**: 参与开源项目, 阅读论文

## 9 结论：PyTorch 开启深度学习之旅

恭喜你！你已经完成了 PyTorch 的入门学习。从 NumPy 出发，我们一步步探索了 PyTorch 的核心概念和实际应用。现在你已经掌握了：

- **张量 (Tensors)**: PyTorch 的核心数据结构，就像 NumPy 数组但有超能力
- **自动求导 (Autograd)**: 让计算机自动计算梯度，省去手动推导的麻烦
- **神经网络模块 (nn.Module)**: 构建深度学习模型的利器
- **优化器 (Optimizer)**: 让网络自己学习参数
- **完整训练流程**: 从数据加载到模型训练的全过程

## 学习原则总结

- **循序渐进**: 从 NumPy 思维到 PyTorch 思维，一步步来
- **对比学习**: 通过与 NumPy、TensorFlow 对比理解差异
- **实践导向**: 每个概念都配合实际代码示例，边学边练
- **问题驱动**: 解答“为什么要这样做”的疑问，理解原理
- **生态思维**: 理解 PyTorch 在整个深度学习工作流中的作用

## 下一步行动建议

建议立即开始实践：

1. **复制代码**: 运行本文所有代码, 确保能跑通
2. **修改参数**: 尝试不同的 batch\_size、学习率, 观察变化
3. **扩展模型**: 在 SimpleNet 基础上加一层, 观察效果
4. **尝试 CNN**: 用卷积层替换全连接层, 体验图像处理
5. **实际项目**: 选择感兴趣的数据集进行分类
6. **阅读源码**: 查看 PyTorch 官方文档和示例
7. **参与社区**: 在 GitHub、Stack Overflow 提问和回答

**深度学习是实践的学科!** 读 10 遍不如动手做 1 遍。犯错是学习的一部分, 不要害怕尝试!

开始你的 PyTorch 之旅吧!

## 参考文献

- [1] X. Chen, “Intuitive Understanding of Tensors in Machine Learning,” [Online]. Available: <https://medium.com/@xinyu.chen/intuitive-understanding-of-tensors-in-machine-learning-33635c64b596>. [Accessed: Nov. 30, 2025].