

大模型中的 PyTorch 数学大纲

这篇文档先不追求把每个公式都展开证明，而是先回答一个更实际的问题：**在大模型训练与推理代码里，PyTorch 背后到底用了哪些数学内容？**

如果后面要继续细化，我建议始终沿着下面这条主线展开：

1. 先看前向计算在算什么
2. 再看反向传播如何把梯度传回来
3. 最后看优化器如何用梯度更新参数

目录

1. 前向计算基础	2
1.1. 线性层	2
1.2. Embedding	2
1.3. 矩阵乘法与批量矩阵乘法	3
1.4. 向量相似度、范数与投影	3
2. 微积分与自动求导：反向传播的核心	3
2.1. 计算图	3
2.2. 链式法则	4
2.3. 标量 loss 到参数梯度	4
2.4. 一个最小反向传播例子	4
3. 概率论与信息论：loss 从哪里来	5
3.1. logits、softmax 与概率	5
3.2. 交叉熵损失	6
3.3. KL 散度与分布匹配	6
3.4. 均值、方差与归一化	6
4. 优化算法：从梯度到参数更新	6
4.1. 梯度下降	6
4.2. 小批量梯度下降	6
4.3. SGD、Momentum、AdamW	7
4.4. 梯度裁剪与权重衰减	7
5. Transformer 中最关键的数学模块	7
5.1. Self-Attention	7
5.2. Multi-Head Attention	7
5.3. MLP、激活函数与门控	7
5.4. 残差连接与 LayerNorm	8
6. 数值计算与训练稳定性	8
6.1. 浮点数精度	8
6.2. softmax 的数值稳定性	8
6.3. 梯度消失与梯度爆炸	9
7. 从 PyTorch API 映射回数学概念	9
8. 一条完整训练链路	9
9. 建议的后续展开顺序	9
10. 本文档后续可细化的具体问题	9
11. 问题	10

1. 前向计算基础

1.1. 线性层

大模型里最常见的基础模块之一是线性变换：

$$Y = XW + b \quad (1)$$

它在 PyTorch 里通常对应 `nn.Linear`。从数学上看，线性层实际做的是：

- 输入输出张量的 shape 如何对应
- 为什么线性层本质上是一个**仿射变换**
- **反向传播** 时如何求 $\partial \frac{L}{\partial} W$ 、 $\partial \frac{L}{\partial} X$

这里的“仿射变换”指的是“线性变换 + 平移”：

$$f(x) = xW + b \quad (2)$$

如果只有 xW ，它是严格的线性变换，必须把零向量映射到零向量；加上 b 之后，输出整体被平移，所以叫仿射变换。`nn.Linear` 虽然名字里叫 Linear，但默认 `bias=True` 时实际就是仿射变换。

“反向传播”指的是：前向计算得到 loss 之后，沿着计算图反方向把梯度传回来。对线性层来说，反向传播要回答三个问题：

- loss 对权重 W 的梯度是多少，也就是 $\partial \frac{L}{\partial} W$
- loss 对 bias b 的梯度是多少，也就是 $\partial \frac{L}{\partial} b$
- loss 对输入 X 的梯度是多少，也就是 $\partial \frac{L}{\partial} X$

它们分别用于不同目的： $\partial \frac{L}{\partial} W$ 和 $\partial \frac{L}{\partial} b$ 给优化器更新参数； $\partial \frac{L}{\partial} X$ 继续传给前一层，让更前面的参数也能被更新。

1.2. Embedding

词嵌入层可以理解为一个查表操作，但从数学上看，也可以理解为从一个大矩阵中取出若干行。

one-hot 是一种表示离散编号的方法。假设词表大小是 5，编号为 2 的 token 可以写成：

$$[0, 0, 1, 0, 0] \quad (3)$$

这个向量只有一个位置是 1，其余都是 0，所以叫 one-hot。

embedding table 是一个可学习矩阵。假设词表大小是 V ，每个 token 的 hidden size 是 d ，那么 embedding table 的 shape 就是：

$$E \in \mathbb{R}^{V \times d} \quad (4)$$

第 i 行 E_i 就是第 i 个 token 的向量表示。把 token id 送进 embedding 层，本质上就是取出对应的那一行。

为什么叫 embedding？因为它把一个离散符号“嵌入”到连续向量空间里。token id 原本只是一个编号，编号之间没有自然的距离关系；变成向量之后，模型就可以用点积、矩阵乘法、梯度下降这些连续数学工具来处理它。

后续可以细化：

- **one-hot** 与 **embedding table** 的关系
- 为什么 embedding lookup 可以看成特殊的矩阵乘法

- embedding 参数为什么也能通过梯度学习

1.3. 矩阵乘法与批量矩阵乘法

Transformer 里到处都是矩阵乘法：

- token 表示乘权重矩阵
- attention 中 QK^T
- attention 权重再乘 V

后续可以细化：

- `torch.matmul` 在不同维度下分别对应什么数学运算
- 为什么 `batched matmul` 是大模型高频操作
- 为什么 GPU 对这类运算特别友好

1.4. 向量相似度、范数与投影

很多模型计算都和“相似不相似”有关。

L_2 范数就是向量的欧几里得长度。对向量

$$x = [x_1, x_2, \dots, x_n] \quad (5)$$

它的 L_2 范数定义为：

$$\|x\|_2 = \sqrt{x_1^2 + x_2^2 + \dots + x_n^2} \quad (6)$$

二维里它就是平面上点到原点的距离，三维里就是空间中箭头的长度。更高维时没法直接画出来，但数学含义仍然一样：它衡量这个向量整体有多大。

例如：

$$\|[3, 4]\|_2 = \sqrt{3^2 + 4^2} = 5 \quad (7)$$

在大模型里，范数经常用来衡量 hidden state、embedding、梯度或参数的尺度。如果一个向量的 L_2 范数很大，说明它整体数值尺度很大；如果很小，说明它靠近零向量。

后续可以细化：

- 点积为什么能衡量相似度
- L_2 范数和向量长度的关系
- cosine similarity 和 attention score 的关系

2. 微积分与自动求导：反向传播的核心

2.1. 计算图

PyTorch 的 `autograd` 可以看成是在维护一张计算图。前向阶段记录运算关系，反向阶段按图回传梯度。

后续可以细化：

- 什么叫“一个张量需要梯度”
- 什么叫叶子节点
- `detach`、`no_grad` 为什么会切断梯度

2.2. 链式法则

反向传播的数学核心是链式法则。假设

$$z = f(y), \quad y = g(x) \quad (8)$$

那么

$$d \frac{z}{d} x = d \frac{z}{d} y \cdot d \frac{y}{d} x \quad (9)$$

后续可以细化：

- 单变量链式法则如何推广到向量和矩阵
- 为什么反向传播本质上是在重复使用链式法则
- 为什么“局部梯度 times 上游梯度”是一个统一套路

2.3. 标量 loss 到参数梯度

训练时通常先得到一个标量损失 L ，然后对参数 θ 求梯度：

$$\nabla_{\theta} L \quad (10)$$

这一步在代码里通常对应：

```
loss.backward()
```

后续可以细化：

- 为什么通常要求 loss 是标量
- Jacobian、gradient、vector-Jacobian product 之间的关系
- 为什么 autograd 不需要显式构造完整 Jacobian

2.4. 一个最小反向传播例子

用最简单的线性模型看反向传播：

$$y = wx + b \quad (11)$$

$$L = (y - t)^2 \quad (12)$$

这里 x 是输入， t 是目标值， w 和 b 是要学习的参数， L 是 loss。前向计算顺序是：

1. 先用 $wx + b$ 得到预测值 y
2. 再比较 y 和目标 t ，得到平方误差 L

反向传播则按相反方向走。先看 loss 对预测值 y 的梯度：

$$\partial \frac{L}{\partial} y = 2(y - t) \quad (13)$$

这个量表示：如果当前预测 y 变大一点，loss 会怎么变。接着通过 $y = wx + b$ 继续往回传：

$$\partial \frac{y}{\partial} w = x \quad (14)$$

$$\partial \frac{y}{\partial} b = 1 \quad (15)$$

$$\partial \frac{y}{\partial} x = w \quad (16)$$

根据链式法则：

$$\partial \frac{L}{\partial} w = \partial \frac{L}{\partial} y \cdot \partial \frac{y}{\partial} w = 2(y - t)x \quad (17)$$

$$\partial \frac{L}{\partial} b = \partial \frac{L}{\partial} y \cdot \partial \frac{y}{\partial} b = 2(y - t) \quad (18)$$

$$\partial \frac{L}{\partial} x = \partial \frac{L}{\partial} y \cdot \partial \frac{y}{\partial} x = 2(y - t)w \quad (19)$$

这就是反向传播最核心的套路：每个节点只需要知道自己的局部梯度，再乘以上游传来的梯度。

在 PyTorch 里可以写成：

```
import torch

x = torch.tensor(2.0)
t = torch.tensor(5.0)

w = torch.tensor(1.0, requires_grad=True)
b = torch.tensor(0.5, requires_grad=True)

y = w * x + b
loss = (y - t) ** 2
loss.backward()

print(w.grad)
print(b.grad)
```

这段代码中，`loss.backward()` 会沿着计算图自动应用上面的链式法则。执行之后：

- `w.grad` 存的是 $\partial \frac{L}{\partial} w$
- `b.grad` 存的是 $\partial \frac{L}{\partial} b$

如果 `x` 也设置了 `requires_grad=True`，那么 `x.grad` 里会存 $\partial \frac{L}{\partial} x$ 。实际训练时，通常只让参数需要梯度；普通输入一般不需要保存梯度。

后续可以继续细化：

- $\partial \frac{L}{\partial} y$
- $\partial \frac{L}{\partial} w$
- $\partial \frac{L}{\partial} b$
- 梯度如何对应到 PyTorch 中的 `.grad`

3. 概率论与信息论：loss 从哪里来

3.1. logits、softmax 与概率

分类模型最后通常先输出 logits，再转成概率：

$$p_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)} \quad (20)$$

后续可以细化：

- logits 为什么不是概率

- softmax 为什么能把任意实数变成概率分布
- 为什么 softmax 常和交叉熵一起出现

3.2. 交叉熵损失

语言模型训练里最常见的损失之一是交叉熵。

后续可以细化：

- 交叉熵的定义
- 为什么 next-token prediction 会落到 cross entropy 上
- `F.cross_entropy` 到底把哪几步合在一起做了

3.3. KL 散度与分布匹配

在蒸馏、RLHF、策略约束等场景里，经常会出现 KL 散度。

后续可以细化：

- KL 散度不是距离，但为什么常被当成“分布差异”
- teacher-student 蒸馏为什么会用 KL
- PPO 一类方法里为什么要控制策略偏移

3.4. 均值、方差与归一化

很多训练技巧都和统计量有关。

后续可以细化：

- batch mean / variance 的作用
- LayerNorm 在算什么
- 为什么归一化有助于训练稳定

4. 优化算法：从梯度到参数更新

4.1. 梯度下降

最基本的更新公式是：

$$\theta \leftarrow \theta - \eta \nabla_{\theta} L \quad (21)$$

这里 η 是学习率。

后续可以细化：

- 为什么梯度给出了局部最陡上升方向
- 为什么减去梯度是在做下降
- 学习率太大或太小分别会发生什么

4.2. 小批量梯度下降

大模型训练几乎不会用全量数据一步一更，而是使用 mini-batch。

后续可以细化：

- 为什么 batch 梯度是全数据梯度的估计
- batch size 对噪声和吞吐的影响
- gradient accumulation 想解决什么问题

4.3. SGD、Momentum、AdamW

PyTorch 中常见的优化器背后对应不同的数学假设和数值策略。

后续可以细化：

- SGD 在做什么
- 动量为什么能减小震荡
- Adam 的一阶矩、二阶矩估计是什么意思
- AdamW 为什么把 weight decay 独立出来

4.4. 梯度裁剪与权重衰减

这些看起来像工程技巧，但背后也都有明确数学含义。

后续可以细化：

- gradient clipping 如何限制梯度爆炸
- weight decay 为什么常被解释为正则化
- 这两者和学习率是如何相互作用的

5. Transformer 中最关键的数学模块

5.1. Self-Attention

Transformer 的核心公式之一是：

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \quad (22)$$

后续可以细化：

- Query、Key、Value 各自是什么意思
- 为什么要除以 $\sqrt{d_k}$
- mask 是如何加进去的

5.2. Multi-Head Attention

多头注意力不是简单重复，而是让模型在不同子空间中并行建模关系。

后续可以细化：

- 为什么要拆成多个 head
- 每个 head 的维度如何分配
- 最后 concat 再投影的意义是什么

5.3. MLP、激活函数与门控

Attention 不是 Transformer 的全部，MLP 也非常重要。

MLP 是 **Multi-Layer Perceptron** 的缩写，中文通常叫“多层感知机”。名字里虽然有“多层”，但在 Transformer 中，它通常就是由两个线性层和一个非线性激活函数组成的前馈网络，因此也经常叫 **FFN (Feed-Forward Network)**：

$$h = \varphi(xW_1 + b_1) \quad (23)$$

$$y = hW_2 + b_2 \quad (24)$$

其中，第一层通常把 hidden size 从 d 扩大到更高的中间维度，激活函数 φ 对结果做非线性变换，第二层再把维度缩回 d 。如果只有连续的线性层而没有激活函数，它们可以合并成一个线性层，表达能力不会真正增加；加入 GELU、ReLU 或 SiLU 等激活函数后，MLP 才能表示更复杂的非线性关系。

在 Transformer 中，Attention 负责让不同 token 之间交换和汇聚信息；MLP 则对每个 token 的 hidden state 独立进行变换。也就是说，同一组 MLP 参数会应用到序列中的每个位置，但一次 MLP 计算本身不会混合不同位置的 token。可以把它粗略理解成：Attention 负责“从上下文中取信息”，MLP 负责“加工当前 token 已经取得的信息”。

在 PyTorch 中，一个最简单的 Transformer MLP 可以写成：

```
mlp = nn.Sequential(
    nn.Linear(hidden_size, intermediate_size),
    nn.GELU(),
    nn.Linear(intermediate_size, hidden_size),
)
```

后续可以细化：

- 两层前馈网络在做什么
- GELU 和 ReLU 有什么差异
- SwiGLU 这类门控结构为什么有效

5.4. 残差连接与 LayerNorm

这部分是大模型能稳定训练的重要支撑。

后续可以细化：

- 残差为什么能改善梯度传播
- LayerNorm 的数学公式
- Pre-Norm 和 Post-Norm 的差别

6. 数值计算与训练稳定性

6.1. 浮点数精度

大模型训练并不只是在实数域里做数学，而是在有限精度浮点数上做数值计算。

后续可以细化：

- float32、float16、bfloat16 的区别
- 为什么会出现上溢、下溢
- mixed precision 为什么既省显存又更复杂

6.2. softmax 的数值稳定性

直接算 $\exp(z_i)$ 可能会溢出，因此实现时经常写成减去最大值的形式。

后续可以细化：

- 为什么 $\text{softmax}(z)$ 与 $\text{softmax}(z - \max(z))$ 等价
- logsumexp 技巧在什么地方出现
- 为什么数值稳定性不是“实现细节”，而是训练能否跑通的前提

6.3. 梯度消失与梯度爆炸

这类问题表面上是训练现象，根子上是链式法则和数值尺度共同作用的结果。

后续可以细化：

- 为什么深层网络容易出现这两个问题
- 初始化、归一化、残差怎样缓解
- 梯度裁剪能解决什么，不能解决什么

7. 从 PyTorch API 映射回数学概念

这一节可以后续做成一张对照表。先给出一个最小版本：

- `nn.Linear` -> 仿射变换
- `torch.matmul` -> 矩阵乘法 / 批量矩阵乘法
- `softmax` -> 归一化指数映射
- `F.cross_entropy` -> $\log_{\text{softmax}} + \text{NLL}$
- `loss.backward()` -> 反向传播
- `optimizer.step()` -> 参数更新
- `optimizer.zero_grad()` -> 清空上一轮梯度缓存

8. 一条完整训练链路

如果把大模型训练压缩成一条主线，可以先按下面这几个步骤理解：

1. 输入 token 先变成 embedding
2. embedding 经过一层层线性变换、attention、MLP，得到 logits
3. logits 和真实标签计算 loss
4. autograd 根据 loss 反向传播，得到每个参数的梯度
5. optimizer 根据梯度更新参数
6. 重复很多轮，直到模型在目标分布上表现更好

后续可以把这节进一步细化成“从一段最小 PyTorch 训练代码逐行对应数学公式”的形式。

9. 建议的后续展开顺序

如果后面要逐章展开，我建议按这个顺序补：

1. 张量维度、矩阵乘法、线性层
2. 链式法则、autograd、反向传播手推
3. softmax、cross entropy、next-token prediction
4. 梯度下降、AdamW、梯度裁剪、学习率调度
5. attention、mask、multi-head、LayerNorm
6. 混合精度、数值稳定性、训练中的常见坏现象

这样安排的原因是：先把“能看懂代码里的基本运算”解决，再去“为什么它能训练起来”，最后再处理 Transformer 的结构细节。

10. 本文档后续可细化的具体问题

为了方便继续扩写，先把可拆分的小问题列出来：

- 为什么 `loss.backward()` 能自动得到所有参数的梯度
- 反向传播到底是在算 Jacobian，还是在算别的东西

- attention 为什么是 QK^T ，而不是别的形式
- softmax 和交叉熵为什么经常一起出现
- AdamW 和 SGD 在直觉上差别是什么
- LayerNorm 到底归一化了哪些维度
- 梯度裁剪具体裁掉了什么量
- 混合精度为什么需要 loss scaling

这份文档目前只负责把地图画出来。后面每次只要挑一个问题，就可以往下补成独立小节。

11. 问题

- 还是感觉很奇怪，感觉训练过程，为什么就可以这样，这和我学习的求导似乎不一样啊
- 看了 transformer 中的 残差网络，更加感觉这个东西很奇怪。

$$y = x + \text{SubLayer}(x) \tag{25}$$