大模型原理与应用

第 2 课：Transformer 核心机制｜注意力、结构与训练

理解大模型最关键的一层结构

这一课看什么

- 为什么 RNN 最终让位给 Transformer
- Self-Attention 到底在算什么
- Transformer 为什么强，也为什么贵

本课主线

1. 从 RNN 局限走到注意力机制
2. 从 Q、K、V 走到 Transformer block
3. 从位置编码走到架构与代价

为什么需要 Transformer

RNN 的几个老问题：

- 长距离依赖难处理，容易“忘”
- 梯度消失 / 梯度爆炸
- 顺序计算，难以并行
- 序列越长，训练和推理越慢

Transformer 的核心回答是：

- 让每个位置都能直接看见其它位置
- 把顺序依赖改造成并行的注意力计算

把 Transformer 放回语言模型发展史

- n-gram 语言模型
- 浅层神经语言模型
- RNN / LSTM
- 预训练深度语言模型
- 以 Self-Attention 为核心的 Transformer

也就是说，Transformer 不是凭空冒出来的，而是语言建模长期演进后的一个关键转折点。

先抓住 Attention 的核心直觉

Attention 要解决的问题是：

> 当前这个 token，应该重点参考上下文中的哪些位置？

它不是把整个历史平均使用，而是：

- 选择性关注
- 对不同位置分配不同权重
- 形成上下文相关的表示

这也是它能处理长距离依赖的关键原因。

Self-Attention 到底特别在哪

- 普通 attention：常见于 encoder 和 decoder 之间
- self-attention：query、key、value 都来自同一段输入序列

所以 self-attention 的本质是：

- 每个 token 看整句
- 决定自己应该参考谁
- 再把这些信息聚合成新的表示

Q、K、V 是什么

材料里给了最标准的定义：

- Query：我要找什么
- Key：我能提供什么匹配线索
- Value：真正要被取出的信息

对每个输入向量 x_i，都线性映射出：

- $q_i = W_q x_i$
- $k_i = W_k x_i$
- $v_i = W_v x_i$

Self-Attention 的计算过程

对位置 `i`：

1. 用 q_i 去和所有 k_j 计算相似度
2. 得到每个位置的注意力分数
3. 经过 softmax 变成权重
4. 对所有 v_j 做加权求和
5. 得到当前位置新的表示 b_i

直觉上，它是在回答：

> “当前词应该从哪些词那里拿多少信息？”

Scaled Dot-Product Attention

经典公式是：

$$Attention(X) = softmax(QK^T/\sqrt{d})V$$

为什么要除以 $\sqrt{d}$?

- 如果维度大，点积会变得很大
- softmax 输入过大，会过度尖锐，训练不稳定
- 缩放之后，数值范围更合理

这点在 `3-transformers.pptx` 里有明确解释。

Self-Attention 的矩阵视角

矩阵形式特别重要，因为它解释了 Transformer 为什么容易并行：

- 一次性计算所有 Q
- 一次性计算所有 K
- 一次性计算所有 V
- 一次性得到相似度矩阵 QK^T
- 一次性输出整句新的表示

结论：

- 对整段序列可以并行计算
- 比 RNN 更适合 GPU

Attention 为什么强，也为什么贵

优点：

- 长距离依赖更容易建模
- 每层顺序操作数是 $O(1)$
- 更适合并行

代价：

- attention 矩阵是全连接的
- 对长度为 n 的序列，计算和存储通常带有 n^2 项

Transformer 强，但不便宜。

为什么还要 Multi-Head Attention

单头注意力有容量限制：

- 一次往往只擅长关注一种关系
- 可能被某一个强模式“主导”

多头注意力的想法是：

- 并行做多组独立 attention
- 每个 head 有自己的 W_q, W_k, W_v
- 不同 head 去关注不同关系
- 最后拼接，再做线性投影

Multi-Head Attention 在做什么

可以把它理解成：

- 一个头学主谓关系
- 一个头学指代关系
- 一个头学局部短语
- 一个头学长距离依赖

虽然这是直观理解，不是硬性规定，但它说明了“多个表示子空间”的意义。

只有 Attention 还不够

材料里反复强调：

- self-attention 本身本质上仍是线性组合
- 如果层层只叠 attention，表达能力不够

所以每个 block 还要加：

- Position-wise Feed Forward Network

即：

- 对每个位置单独做一个小型前馈网络
- 引入额外非线性
- 提升表示能力

一个标准 Transformer Block

一个标准 block 通常包括：

1. Multi-Head Self-Attention
2. Add & Norm
3. Feed Forward Network
4. Add & Norm

其中 Add & Norm 对应：

- residual connection
- layer normalization

Residual 与 LayerNorm 的作用

为什么它们重要?

- residual 让梯度更容易传播
- 深层网络训练更稳定
- layer norm 稳定激活分布
- 避免训练过程数值漂移太大

简单说:

- attention 负责“看哪里”
- FFN 负责“怎么变换”
- residual + norm 负责“怎么训得动”

没有位置编码会发生什么

Self-Attention 天生对顺序不敏感。

如果没有位置编码：

- “dog bites man”
- “man bites dog”

从 bag-of-words 角度可能差不多，但语义完全不同。

所以 Transformer 需要额外告诉模型：

- 这个 token 在第几个位置
- 它和别的 token 相距多远

位置编码的几种思路

材料覆盖了几类典型方案：

- 绝对位置编码
- 学习式位置编码
- 正弦 / 余弦位置编码
- 相对位置编码
- Rotary Position Embedding（RoPE）
- ALiBi 等长度泛化方案

这不是边角料，而是 Transformer 长上下文能力的核心问题之一。

正弦 / 余弦位置编码

原始 Transformer 的经典做法：

- 为每个位置构造一个固定向量
- 不同维度使用不同频率的 sin / cos

好处：

- 不需要额外学习所有位置参数
- 具有一定的相对位置信息
- 可推广到未见过的长度

局限：

- 仍然不是最理想的相对位置建模方式

绝对位置 vs 相对位置

绝对位置编码回答：

- “我在第 17 个位置”

相对位置编码回答：

- “我和你相隔 3 个 token”

为什么后者常常更重要?

- 语言中很多规律依赖相对距离
- 长度外推时，相对编码往往更稳

这也是后来 T5、DeBERTa、Transformer-XL、RoPE、ALiBi 持续演化的原因。

Decoder-only Transformer

- 一层层堆叠 self-attention block
- 顶部接 prediction head
- 每个位置预测下一个 token

这就是今天 GPT 系列最核心的骨架。

生成时怎么工作

推理时的流程是：

1. 输入已有上下文
2. 计算最后一个位置的输出分布
3. 采样 / 选择下一个 token
4. 把新 token 接到末尾
5. 继续重复，直到 EOS 或停止条件

所以生成是自回归的：

- 一步接一步
- 每一步都依赖前面已经生成的内容

训练时怎么工作

训练语言模型时：

- 输入序列右移一位
- 每个位置预测“下一个词”
- 对所有位置同时计算 loss
- 把各位置 loss 加总
- 再做反向传播

这使得训练可以并行，而不是像生成那样一步一步来。

为什么生成模型一定要 Mask

如果训练时让当前位置看到未来 token，会发生什么？

- 模型直接偷看答案
- 形成 data leakage
- 学不到真正的自回归预测能力

所以 decoder-only 模型里要用：

- causal mask
- 或者说 upper-triangular mask

保证每个位置只能看自己和过去。

经典 Encoder-Decoder Transformer

- Encoder：双向读输入
- Decoder：带 mask 地逐步生成输出
- Cross-Attention：decoder 读取 encoder 的表示

适合：

- 机器翻译
- 摘要
- 条件生成

三种 Attention 形式

在完整的 encoder-decoder 结构里会出现三种 attention：

1. Encoder self-attention
2. Decoder masked self-attention
3. Decoder cross-attention

这是理解 T5、翻译模型、摘要模型的关键。

把几种主流架构放到一张图里

Transformer 变体：

Encoder-only

- 例子：BERT、RoBERTa、SciBERT
- 擅长理解、分类、检索、判别任务

Decoder-only

- 例子：GPT-2、GPT-3、GPT-4、LLaMA、Mistral
- 擅长生成，自回归建模

Encoder-Decoder

- 例子：T5、Meena
- 擅长条件生成、序列到序列任务

为什么 GPT 类模型是 Decoder-only

因为它们的目标是：

- 基于已有上下文继续写下去

这天然对应：

- masked self-attention
- next-token prediction
- auto-regressive decoding

所以 GPT 的核心不是“更神秘的网络”，而是“在 Transformer decoder 上做极大规模预训练”。

计算复杂度为什么重要

材料后半段专门讨论了 FLOPs、IO 和 bottleneck。

对长度为 n 的序列：

- attention 的关键项带 n^2
- 序列变长时，代价增长很快

这解释了为什么：

- 长上下文很贵
- 内存很快爆
- inference 优化极其重要

推理为什么比训练更麻烦

训练时：

- 整段序列一起喂进去
- 计算可以并行

推理时：

- 每次只生成一个新 token
- 需要反复使用前面上下文
- 如果每次都重算历史 K、V，会很浪费

所以工程上必须优化。

KV Cache 在解决什么

核心思想：

- 过去 token 的 Key 和 Value 不要每步重算
- 把它们缓存起来
- 只为新 token 计算新的 `q, k, v`

好处：

- 避免重复计算
- 单步生成代价显著下降

代价：

- 需要额外显存存储 cache

Encoder-Decoder 的复杂度

材料里有一个很典型的问题：

如果输入长度是 N，输出长度是 M，那么 attention 复杂度包含：

- encoder self-attention：$O(N^2)$
- decoder self-attention：$O(M^2)$
- cross-attention：$O(NM)$

这比只看一个模块更完整，也更接近真实系统代价。

Transformer 为什么替代了 RNN

优势：

- 长距离依赖更强
- 更容易并行
- 更容易堆更深
- 在翻译、摘要、语言建模中效果更好

劣势：

- 注意力代价随长度平方增长
- 实现细节更多
- 长上下文与推理效率需要专门优化

总体上，收益远大于代价。

写 Transformer 时要知道的模块

- `Embedding`
- `PositionalEncoding`
- `MultiHeadAttention`
- `FeedForward`
- `SublayerConnection`
- `EncoderLayer`
- `DecoderLayer`
- `Prediction Head`

理解这些模块，后面读 GPT / BERT / T5 的源码才不会迷路。

这节课你该带走什么

1. Self-Attention 替代了循环结构
2. Q / K / V 定义了匹配、参考和取出
3. 标准 block 由 attention、FFN、残差、归一化组成
4. 位置编码决定模型是否真正理解顺序
5. 三大范式是 decoder-only、encoder-only、encoder-decoder
6. 工程约束来自 n^2 复杂度、KV-cache 和 IO

为什么这一课是大模型的结构母语

后面学到的几乎所有大模型，都可以回到今天这张地图上：

- GPT：decoder-only Transformer
- BERT：encoder-only Transformer
- T5：encoder-decoder Transformer
- 长上下文模型：重点改进位置编码与注意力效率
- 推理优化：重点改 KV-cache、并行策略、IO 与内存布局

所以第二课不是“讲一个模型”，而是在建立大模型时代最核心的结构语言。

第 2 课收束

Transformer 不只是一个模块，而是一种时代结构

- 它是一整套结构设计
- 目标是更强的建模能力
- 代价是更高的计算与工程复杂度