大模型原理与应用

第 3 章：预训练 | 大模型能力的起点

理解模型为何先学语言，再学任务

这一章看什么

- 语言模型为什么成为 NLP 的总任务
- Transformer 为什么变成预训练底座
- GPT、BERT、T5 怎样代表三条主路线
- Scaling Law 为什么改变了训练策略
- Prompt 为什么会变成预训练模型的运行时接口

本章主线

1. 先回到语言模型的历史起点
2. 再看 Transformer 怎样把预训练推到新阶段
3. 然后理解规模、数据和计算如何共同决定能力
4. 最后看 Prompt 怎样把这些能力调出来

从任务专用模型到通用基础模型

早期做 NLP 时，常见套路是：

- 为每个任务单独收集标注数据
- 训练一个随机初始化模型
- 让模型只服务某一个任务

问题很明显：

- 标注数据贵
- 任务迁移差
- 每个任务都要从头学语言

预训练范式改变的就是这一点。

为什么语言模型会成为一个“总任务”

因为很多 NLP 任务本质上都依赖同一件事：

- 理解上下文
- 预测合理的语言延续
- 建立可迁移的表示

所以，语言模型从一个具体任务，逐渐变成一个“上层任务”：

> 先把语言学会，再把能力迁移到分类、问答、摘要、翻译和对话

先看语言模型最早在做什么

最经典的问题非常简单：

- 给定前文，预测下一个词

比如：

- `The cat sat on __`

这个目标看起来朴素，但它逼模型学习：

- 词序
- 搭配
- 句法
- 局部语义

n-gram 语言模型解决了什么

早期方法常用统计语言模型：

- unigram
- bigram
- trigram
- 更高阶 n-gram

它们的优点是：

- 简单
- 可解释
- 在语音识别和机器翻译里很有用

但核心限制也很明显：

- 上下文窗口短
- 数据稀疏严重
- 看不到更长依赖

为什么“只靠计数”不够

n-gram 的本质是：

- 统计词串共现

但语言里很快会遇到问题：

- 新词
- 稀有词
- 远距离依赖
- 语义相近但表面不同的表达

所以语言模型后来逐渐转向：

- 让模型自己学习分布式表示
- 不再把词当作完全孤立的符号

从 one-hot 到词向量

传统 one-hot 表示的问题是：

- 维度很高
- 词与词之间完全正交
- `hotel` 和 `motel` 看不出相似性

词向量的意义在于：

- 让语义相近的词在向量空间里更接近

这也是 Word2Vec / GloVe 为什么重要。

Word2Vec / GloVe 改变了什么

它们告诉我们：

- 词的 meaning 可以用连续向量表示
- 共现统计能转化成语义结构

这一步很重要，但还没有真正解决：

- 一个词在不同语境里含义会变

例如：

- python 可能是蛇，也可能是编程语言
- bank 可能是河岸，也可能是银行

从静态词向量到上下文化表示

ELMo 是一个很关键的过渡：

- 静态词向量：同一个词永远一个 embedding
- 上下文化表示：同一个词在不同句子里有不同表示

这意味着预训练学到的不只是词表映射，而是：

- 词在上下文中的功能
- 句法结构
- 语义依赖
- 世界知识

神经语言模型带来了什么新东西

从 Bengio 2003 的 fixed-window neural LM 开始，语言模型被正式看成一个机器学习问题：

- 输入上下文词的 embedding
- 用神经网络预测右边的目标词

这个变化很关键，因为它让后来的一切成为可能：

- RNN
- LSTM
- Transformer

固定窗口神经 LM 为什么还不够

它比 n-gram 更强，但仍然有明显限制：

- 只能看有限窗口
- 窗口外的信息直接丢掉
- 长距离依赖仍然困难

这说明真正理想的语言模型需要：

- 更灵活的上下文建模
- 更强的长程依赖表达

RNN / LSTM 为什么也没成为终点

RNN 的贡献很大，但也暴露了几个经典问题：

- 长序列上容易遗忘
- 梯度消失 / 爆炸
- 很难并行

所以它虽然让语言模型往前走了一大步，但并没有成为大规模预训练的最终底座。

Transformer 为什么改变了预训练

Transformer 最关键的变化是：

- 用 self-attention 取代 recurrence

这带来几件决定性的事：

- 每个 token 都能直接看别的 token
- 计算可以并行
- 更适合大规模矩阵运算
- 更适合在 GPU / TPU 上扩展

这正是它后来成为预训练底座的原因。

Self-Attention 真正解决了什么

Self-attention 的核心不是“一个新模块”而已，而是：

- 让上下文表示直接由整个序列共同决定

它带来的直观效果是：

- token 不再只靠相邻位置传信息
- 每个位置都能动态关注更重要的上下文
- 语义相关的词可以直接交互

代价也很清楚：

- 计算复杂度常常是 $O(n^2)$

Transformer 为什么特别适合做预训练

预训练需要同时满足几件事：

- 能处理海量文本
- 能扩展到大模型
- 能在现代硬件上高效训练

Transformer 几乎正好匹配这三点：

- 可并行
- 可堆叠
- 可扩展

所以从 2017 年之后，预训练几乎全面转向 Transformer。

子词建模为什么重要

从 BPE / WordPiece / SentencePiece 讲起

因为真实语言里：

- 有新词
- 有拼写变化
- 有复杂形态变化
- 有不同书写系统

如果词表完全固定，就会不断遇到 UNK 的问题。

所以现代预训练通常不是直接以“整词”为单位，而是用子词。

子词建模真正解决的是什么

它解决的不只是“切词技术”问题，而是：

- 长尾词覆盖
- 未登录词处理
- 复杂形态语言建模
- 参数共享效率

这也是为什么今天几乎所有主流预训练模型，都离不开 tokenizer 设计。

预训练到底在做什么

预训练也叫 self-supervised learning：

- 不依赖人工标签
- 从原始文本本身构造训练目标

例如：

- 给前文预测下一个 token
- 把部分词遮住，再预测被遮住内容
- 对被腐蚀的文本做重建

本质上是在逼模型从文本中提取规律。

模型在预训练里学到的到底是什么

材料里反复举的例子说明：

- 句法：代词指代、主谓一致
- 语义：情感、主题、搭配
- 常识：时间、地点、工具、因果
- 世界知识：大学位置、历史事实、领域知识

所以“预训练”不是单纯记忆文本，而是在学一种可迁移的语言表示与概率结构。

三类主流预训练范式

1. Decoder-only
2. Encoder-only
3. Encoder-Decoder

后面的 GPT、BERT、T5，基本就是这三大路线的代表。

Decoder-only：GPT 路线

目标：

- 预测下一个 token

特点：

- 自回归
- 只能看左侧上下文
- 天然适合生成

优点：

- 统一了语言建模和生成
- 可以直接延续文本
- 后来很自然走向 in-context learning 和 chat model

Encoder-only：BERT 路线

BERT 的核心思路：

- 使用双向 Transformer encoder
- 通过 Masked Language Modeling 学上下文表示

也就是：

- 随机遮住一些词
- 让模型根据左右两边上下文恢复它们

优点：

- 更适合理解任务
- 分类、匹配、抽取类任务很强

BERT 预训练为什么重要

BERT 的突破不只是一个新模型，而是一个新结论：

- 双向上下文化表示可以大幅提升理解类任务效果

它也让很多人真正接受：

- 先做大规模预训练，再做任务适配

从此成为 NLP 默认范式。

Encoder-Decoder：T5 路线

- 把所有任务统一写成 text-to-text
- 输入是文本
- 输出也是文本

例如：

- 翻译
- 摘要
- 分类
- 相似度

都可以写成“给定一句话，生成另一段话”。

T5 的意义

T5 不只是一个模型，而是一种统一接口：

- task prefix + input text
- output text

这让预训练和下游任务之间的接口变得更自然。

它强调的是：

- span corruption
- text-to-text 统一建模
- encoder-decoder 在生成任务中的优势

预训练数据为什么会成为核心问题

一旦模型变大，真正的问题就不再只是“模型结构对不对”，还包括：

- 数据够不够大
- 数据够不够多样
- 数据有没有重复和污染
- 训练预算够不够

所以现代预训练其实是三件事一起扩展：

- 参数规模
- 数据规模
- 计算规模

什么叫做 Scaling

材料里强调了一个关键点：

- scaling 不只是把参数变大

真正的 scaling 同时涉及：

- 更大的模型容量
- 更多的训练计算
- 更多、更丰富的数据

也就是：

> scale = capacity + compute + data 的共同扩展

为什么 Scaling Law 重要

因为真正训练前沿模型非常贵。

所以工程上最重要的问题之一是：

- 在给定数据和计算预算下，怎样选模型规模最合适?

Scaling law 的价值就在于：

- 用更小实验预测更大训练的趋势
- 避免完全靠 trial and error

Scaling Law 给出的核心直觉

多份材料都在强调类似结论：

- loss 和 scale 常常呈现可预测关系
- 性能会随着参数、数据、计算增加而改善
- 但如果其中一个维度被卡住，收益会变差
- 当模型或数据单独固定时，边际收益会下降

所以“大就一定好”并不准确，关键是搭配。

为什么今天大家会谈 Emergent Abilities

模型规模继续上升后，人们观察到：

- zero-shot / few-shot 能力增强
- in-context learning 更稳定
- 某些能力像是跨过阈值后才出现

这就是大家讨论“emergence”的背景。

它提醒我们：

- scale 改变的不只是分数，也可能改变行为模式

Prompt 为什么应该放在这一章里

因为学生最容易误解的一点是：

- Prompt 好像只是“提问技巧”

但更准确的说法是：

- Prompt 是我们和预训练模型交互的运行时接口

它本身不改变参数，却能决定：

- 模型看到了什么上下文
- 模型被要求扮演什么角色
- 模型要按什么格式输出

什么叫 Prompt

最朴素的理解就是：

- 你给模型的一段输入文本

它可以很短：

- 请解释什么是 `Transformer`

也可以很完整：

- 角色说明
- 任务描述
- 输出格式
- 示例
- 约束条件

所以 Prompt 不是一句“咒语”，而是一次任务接口设计。

为什么 Prompt 会突然变重要

在小模型时代，大家更多依赖：

- 为任务单独训练模型

但预训练模型变大之后，人们发现：

- 即使不改参数，只改输入形式，模型行为也会明显变化

这说明：

- 预训练已经在参数里存了很多通用能力
- Prompt 的作用，是把这些潜在能力调动出来

Zero-shot：不给例子，直接做

最简单的 Prompt 形式就是：

- 直接告诉模型任务是什么

例如：

- 判断这句话是正面还是负面：这家餐厅非常值得再来。

这叫 zero-shot。

它能工作，本身就说明：

- 预训练模型已经学到了一定的任务抽象能力

Few-shot：先给例子，再让模型照着做

再进一步，我们可以在 Prompt 里放几个例子：

- 输入是什么
- 输出应该长什么样

模型并没有更新参数，但会临时表现得更像“学会了任务”。

这就是 few-shot prompting。

它背后的关键现象就是：

- in-context learning

In-Context Learning 真正在说什么

ICL 的重点不是“模型记住了提示词”，而是：

- 冻结参数的模型
- 只靠 prompt 里的上下文
- 就能临时适配一个新任务

这件事非常重要，因为它说明：

- 预训练学到的不是死知识
- 而是一种可以被上下文重新组织的能力

为什么 GPT 路线会特别适合 Prompt

因为 decoder-only 模型天然在做一件事：

- 根据前文继续生成后文

所以当你把：

- 任务说明
- 输入文本
- 示例答案

都写进前文时，模型就会把它们一起当成“接下来该如何续写”的条件。

这也是为什么 GPT-3 之后，zero-shot / few-shot 会突然变成主角。

Prompt Template 在做什么

很多时候，真正稳定效果的不是一句灵感式提问，而是模板：

- 角色
- 任务
- 输入槽位
- 输出格式

例如：

- 你是一名助教
- 请用三句话总结下面内容
- 内容：`{text}`
- 输出格式：三条 bullet

模板的价值是：

- 减少随机性
- 让任务接口稳定
- 方便批量调用

Chain-of-Thought 为什么有效

很多材料都在强调：

- 对复杂问题，直接问答案常常不够

如果 Prompt 明确要求：

- 一步一步想
- 先写中间推理
- 再给最后结论

模型在数学、常识、符号推理上常常会更好。

这就是 Chain-of-Thought，简称 CoT。

CoT 真正带来的是什么

它不只是“答案更长”。

它真正做的是：

- 把多步问题拆开
- 让模型显式暴露中间过程
- 让错误更容易定位

材料里反复出现的一句话就是：

- `Let's think step by step.`

这句话背后体现的是：

- 推理过程也可以被 Prompt 显式引导

Prompt 的能力边界在哪里

Prompt 很强，但它不是万能的。

原因包括：

- 小模型常常没有足够能力被“调出来”
- Prompt 只能重组已有能力，不能凭空创造能力
- 对稳定格式、专业领域和行为约束，单靠 Prompt 往往不稳

所以课程后面才需要继续讲：

- 微调
- 对齐
- RL 后训练

用一句话串起来这一段

预训练提供的是：

- 参数里的通用能力底座

Prompt 提供的是：

- 运行时的任务接口

两者结合后，模型才开始表现出：

- zero-shot
- few-shot
- in-context learning
- chain-of-thought reasoning

预训练到底给了什么

如果把这一章压缩成一句话，那就是：

- 预训练先给模型通用语言能力和世界知识底座

这包括：

- 表示能力
- 参数初始化
- 概率建模能力
- 跨任务迁移能力

也正因为如此，后面的任务适配才会变得可能。

只有预训练还不够

预训练模型会：

- 补全
- 模仿
- 延续文本分布

但用户真正需要的是：

- 听懂指令
- 回答问题
- 遵守规范
- 拒绝危险请求

所以：

> 语言建模能力 ≠ 指令遵循能力

为什么预训练之后还会走向微调

预训练解决的是：

- 模型先学会语言

但没有直接解决：

- 具体任务接口
- 指令遵循
- 输出风格
- 使用偏好

所以后续还需要：

- SFT
- PEFT
- alignment / RL post-training

这也是为什么这一章之后，课程会继续拆成：

- 11：微调

预训练这一章的完整位置

把课程主线串起来，可以得到：

1. 先用海量无标注文本做预训练
2. 得到 base model
3. 再用任务 / 指令数据做微调
4. 如有需要，再进入 alignment / RL post-training

所以预训练回答的是最底层的问题：

> 模型的通用能力到底从哪里来？

这节课你该带走什么

1. 语言模型之所以重要，是因为它逐渐变成了 NLP 的“总任务”
2. Transformer 之所以重要，是因为它让大规模预训练真正变得可扩展
3. GPT、BERT、T5 对应三条最重要的预训练路线
4. 规模、数据和计算共同决定了预训练能力边界
5. Prompt 是把预训练能力调出来的运行时接口
6. 预训练提供底座，但不会自动替代后续微调和对齐

为什么这一章定义了能力底座

后面无论讲微调、对齐、多模态还是应用落地，都会回到这张图：

- 能力从哪来？先靠预训练
- 能力怎么调出来？先靠 Prompt
- 行为怎么调？再靠微调和对齐
- 成本怎么控？再靠工程适配方法

所以这一章真正建立的是：

> 大模型先学语言世界，再进入任务世界

第 3 章收束

预训练给能力底座，后续章节再给任务接口与行为约束