从零训练 LLM 基础模型

杨岢瑞

Dec 09, 2025

想象一下，OpenAI 的 GPT-3 模型从零训练耗费了数月时间和数亿美元的计算资源，但如今借助开源工具，即使是个人开发者或小团队，也能在合理预算内从零训练一个小型 LLM，例如参数规模达到 10 亿的模型。这不仅仅是技术进步的体现，更是 AI 民主化的里程碑。本文将带你从头开始，完整走一遍训练 LLM 基础模型的流程，我们将聚焦于实际操作，避免空洞理论，而是提供可复现的步骤和代码。如果你是一名有 Python 和 PyTorch 基础的 AI 工程师或研究者，并且拥有 GPU 环境，这篇指南将助你快速上手。

LLM，即大型语言模型，指的是基于 Transformer 架构的参数规模庞大的神经网络，通常拥有数亿到数万亿参数。基础模型特指未经指令微调的预训练模型，它从随机初始化的权重开始，通过自监督学习（如下一 token 预测）在海量文本上训练而成。这与后续的 fine-tune 模型（如 ChatGPT）不同，后者针对特定任务进行了额外优化。「从零训练」意味着我们不依赖现有 checkpoint，而是全新构建模型架构、准备数据并启动预训练过程。这种方式赋予了你最大灵活性，能自定义一切从架构到数据集。

为什么选择从零训练 LLM 基础模型呢？它的优势在于完全自定义架构和数据，避免了现有开源模型可能存在的版权污染问题，同时让你深入理解底层机制，例如注意力机制如何捕捉长距离依赖。当然，这也伴随着挑战：计算资源需求高，训练周期从几天到数月不等，成本可能达到数千美元，主要适用于领域特定模型如医疗文本生成或代码补全、企业私有模型开发，以及学术实验场景。对于资源有限的读者，我们将优先讨论 1B 参数规模的模型，这在单机多卡 GPU 上即可实现。

本文的目标是提供一个端到端、可操作的指南，包括代码仓库链接（假设为 GitHub: yourusername/llm-from-scratch）。我们将按以下路线图展开：先规划环境与资源，然后设计模型架构，构建数据管道，配置训练核心，实现优化与调试，最后评估迭代并部署。跟随本文，你能在两周内训练出一个功能性 1B 模型。

1　准备阶段：环境与资源规划

训练 LLM 的第一步是评估硬件需求。以 100M 参数的小型模型为例，一张 A100 40GB GPU 搭配 80GB 系统 RAM 和 1TB SSD 即可在几天内完成；对于 1B 参数模型，需要 4 张 A100，总内存达 512GB，存储 10TB NVMe，训练时长约 1 到 2 周；7B 参数则要求 8 张 H100 或多机集群，周期超过一个月。如果你没有本地硬件，云服务是理想选择，如 AWS 的 p4d 实例、GCP 的 A3 系列，或更实惠的 Lambda Labs 和 RunPod，这些平台按小时计费，支持弹性扩展。

软件栈的选择至关重要。核心是 PyTorch 2.x，它集成了 torch.compile 进行图优化，以及 DeepSpeed 和 FlashAttention2 用于加速。我们推荐从 nanoGPT 或 Hugging Face Transformers 框架起步，前者简洁适合从零实现，后者提供丰富工具。对于自定义需求，从头用 PyTorch 构建 Transformer 是最佳实践。数据集是训练的命脉，优先选用开源资源如 C4、The Pile 或 RedPajama，后者提供万亿 token 级干净数据。数据准备需至少 100B tokens 以匹配 1B 模型规模，按 Chinchilla 定律，optimal 参数量约等于 token 数的 1/20。

数据集清洗是关键，避免低质量文本拖累模型。首先用 Hugging Face datasets 库加载数据，然后去重和过滤。这里是一个简单的清洗脚本示例：

```
from datasets import load_dataset
import pandas as pd

dataset = load_dataset("c4", "en", split="train", streaming=True)
def clean_text(example):
    text = example["text"]
    if len(text) < 128 or len(text) > 8192: # 过滤长度异常
        return {"text": None}
    # 简单去重：移除常见噪声
    text = text.replace("\n", "␣").strip()
    return {"text": text}

cleaned = dataset.filter(clean_text)
cleaned.save_to_disk("cleaned_c4")
```

这段代码首先从 Hugging Face Hub 流式加载 C4 数据集，然后定义 clean_text 函数过滤长度小于 128 或大于 8192 的文本（避免碎片或过长序列），并移除换行符等噪声。如果文本不符合条件，返回 None 以过滤掉。最终用 save_to_disk 保存清洗后数据集。这个过程可扩展到并行处理 TB 级数据，确保输入质量高，从而降低后续 perplexity。

分词化使用 BPE 或 SentencePiece，最简单是 tiktoken 库自建 tokenizer，支持 50k 词汇表：

```
import tiktoken
from tokenizers import Tokenizer

enc = tiktoken.get_encoding("cl100k_base")
tokenizer = Tokenizer.from_pretrained("gpt2") # 或自训
tokens = enc.encode("你的文本")
```

tiktoken 的 get_encoding 加载预训练编码器，encode 将文本转为 token ID 序列。自训 tokenizer 可基于你的数据集调用 Tokenizer.from_pretrained 后 fine-tune，确保覆盖领域特定词汇。

2 模型架构设计

Transformer 是现代 LLM 的基石，与早期的 Encoder-Decoder 不同，当今 LLM 主流采用 Decoder-only 架构，仅含自注意力层，专为自回归生成优化。回忆一下，自注意力计算查询（Query）、键（Key）和值（Value）的点积相似度：$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$，其中 d_k 是键维度，softmax 确保概率分布。

关键组件设计从嵌入层开始。词汇表大小设为 50k，模型维度 d_model=1024，使用 Rotary Positional Embedding（RoPE）注入位置信息，比绝对位置编码更鲁棒。注意力层采用多头机制（heads=16），集成

FlashAttention2 加速，避免显存爆炸；进一步优化用 Grouped-Query Attention（GQA），共享部分键值头以降低推理延迟。前馈网络（FFN）用 SwiGLU 激活：$\text{SwiGLU}(x) = (xW_1 \cdot \text{silu}(xW_2))W_3$，比 ReLU 更平滑；归一化选用 RMSNorm 置于注意力前：$\text{RMSNorm}(x) = \frac{x}{\sqrt{\text{mean}(x^2)+\epsilon}} \cdot g$。整体架构堆叠 24 层，总参数约 1B。

以下是从零实现的 PyTorch 模型核心伪代码：

```
import torch.nn as nn
import torch

class RMSNorm(nn.Module):
   def __init__(self, dim):
      super().__init__()
      self.scale = nn.Parameter(torch.ones(dim))

   def forward(self, x):
      norm = x.norm(eps=1e-6, dim=-1, keepdim=True)
      return x / norm * self.scale

class CausalSelfAttention(nn.Module):
   def __init__(self, dim, heads):
      super().__init__()
      self.heads = heads
      self.scale = dim ** -0.5
      self.to_qkv = nn.Linear(dim, dim * 3, bias=False)
      self.to_out = nn.Linear(dim, dim)

   def forward(self, x):
      b, t, c = x.shape
      qkv = self.to_qkv(x).chunk(3, dim=-1)
      q, k, v = map(lambda y: y.view(b, t, self.heads, c // self.heads).transpose(1,
          ↪ 2), qkv)
      dots = torch.matmul(q, k.transpose(-1, -2)) * self.scale
      mask = torch.ones(t, t, device=x.device).tril() # 因果掩码
      dots.masked_fill_(~mask.bool(), float('-inf'))
      attn = dots.softmax(dim=-1)
      out = torch.matmul(attn, v).transpose(1, 2).contiguous().view(b, t, c)
      return self.to_out(out)

class TransformerBlock(nn.Module):
   def __init__(self, dim, heads):
```

```
        super().__init__()
        self.norm1 = RMSNorm(dim)
        self.attn = CausalSelfAttention(dim, heads)
        self.norm2 = RMSNorm(dim)
        self.ffn = nn.Sequential( # SwiGLU 简化版
            nn.Linear(dim, dim * 4),
            nn.SiLU(),
            nn.Linear(dim * 4, dim)
        )

    def forward(self, x):
        x = x + self.attn(self.norm1(x))
        x = x + self.ffn(self.norm2(x))
        return x

class LLM(nn.Module):
    def __init__(self, vocab_size, dim=1024, layers=24, heads=16):
        super().__init__()
        self.token_emb = nn.Embedding(vocab_size, dim)
        self.blocks = nn.Sequential(*[TransformerBlock(dim, heads) for _ in range(
            ↪ layers)])
        self.norm = RMSNorm(dim)
        self.head = nn.Linear(dim, vocab_size, bias=False)

    def forward(self, idx):
        b, t = idx.shape
        tok_emb = self.token_emb(idx)
        x = self.blocks(tok_emb)
        x = self.norm(x)
        logits = self.head(x)
        return logits
```

这段代码定义了完整 Decoder-only 模型。首先，RMSNorm 实现根均方归一化，计算每个 token 向量的 L2 范数后缩放，避免梯度爆炸。CausalSelfAttention 处理多头自注意力：to_qkv 投影输入到 Q/K/V，chunk 分割后 reshape 为多头格式；计算点积分数 dots，乘以缩放因子 scale，并应用下三角因果掩码（tril 生成，masked_fill 屏蔽未来 token）；softmax 后与 V 相乘，重塑输出。TransformerBlock 是残差块：先 norm1 + attn，再 norm2 + ffn，使用 SiLU 近似 SwiGLU。顶层 LLM 嵌入 token，堆叠 blocks，末尾线性头预测 logits。初始化后，总参数通过 torch.sum(p.numel() for p in model.parameters()) 验证约 1B。这个实现简洁高效，对比 Hugging Face 的 GPT2Config，更易自定义。

规模选择遵循 Chinchilla 定律：为 T tokens，最优参数 N ≈ T / 20。计算 FLOPs 预算：单步前向约 $6Nd$，

其中 d 是模型维度，确保不超过硬件极限。

3 数据预处理与 Pipeline 构建

数据管道从原始文本开始，经过去重过滤、分词、sharding，最终进入 DataLoader。流程简述为：Raw Data 通过 Dedup/Filter 清洗，Tokenize 转为 ID 序列，用 WebDataset 分区，最后 torch DataLoader 分布式加载。

分词脚本扩展前述清洗：

```
import tiktoken
enc = tiktoken.get_encoding("cl100k_base")

def tokenize_dataset(path):
    dataset = load_dataset("text", data_files=path)
    def tokenize(examples):
        tokens = enc.encode_batch(examples["text"])
        return {"tokens": [t for t in tokens if len(t) > 128]} # 过滤短序列
    tokenized = dataset.map(tokenize, batched=True, remove_columns=["text"])
    tokenized.save_to_disk("tokenized_data")
```

这里，encode_batch 批量编码文本为 token 列表，过滤长度不足 128 的序列，避免无效样本。map 操作移除原始文本列，节省空间。保存后数据以 Hugging Face 格式存储，支持流式读取。

分布式加载用 torch.distributed 和 FSDP：

```
import torch.distributed as dist
from torch.utils.data import DataLoader
from datasets import load_from_disk

dist.init_process_group(backend="nccl")
dataset = load_from_disk("tokenized_data")["train"].shuffle()

def collate_fn(batch):
    tokens = torch.stack([torch.tensor(x) for x in batch["tokens"]])
    return {"input_ids": tokens[:, :-1], "labels": tokens[:, 1:]}

dataloader = DataLoader(dataset, batch_size=8, collate_fn=collate_fn, num_workers=4)
```

dist.init_process_group 初始化 NCCL 后端，用于多 GPU 通信。shuffle 确保随机性，collate_fn 堆叠 token 张量，移位生成 input_ids（预测目标）和 labels（下一 token）。batch_size=8 视 GPU 调整。这个 pipeline 支持万亿 token 级高效迭代。

质量控制通过 perplexity 评估：$\text{PPL} = \exp\left(\frac{1}{N}\sum \mathcal{L}\right)$，并可视化 token 分布直方图（用 matplotlib.pyplot.hist）确认无偏倚。

4 训练核心：配置与实现

训练核心是下一 token 预测，使用交叉熵损失：$\mathcal{L} = -\sum \log p(x_{t+1}|x_{<t+1})$。优化器 AdamW，学习率调度 Cosine + Warmup：初始 LR=6e-4，warmup 10% 步数线性增至峰值，后余弦衰减至 10%。

分布式策略依规模：小模型用 DDP（Data Distributed Parallel），大模型 FSDP（Fully Sharded Data Parallel）或 DeepSpeed ZeRO-3，后者分片参数/梯度/优化器状态至多 GPU。超参数示例：全局 batch size 1M tokens（每步 512 序列 × 2048 长度），上下文从 2048 渐增至 8192，weight decay 0.1。

完整训练脚本仿 nanoGPT：

```
import torch
from torch.optim import AdamW
from torch.utils.data.distributed import DistributedSampler
import wandb

model = LLM(vocab_size=50257).cuda().train()
optimizer = AdamW(model.parameters(), lr=6e-4, weight_decay=0.1)
sampler = DistributedSampler(dataset, shuffle=True)

for epoch in range(100):
   sampler.set_epoch(epoch)
   for batch in dataloader:
      optimizer.zero_grad()
      logits = model(batch["input_ids"])
      loss = torch.nn.functional.cross_entropy(logits.view(-1, logits.size(-1)),
          ↪ batch["labels"].view(-1))
      loss.backward()
      torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
      optimizer.step()
      wandb.log({"loss": loss.item()})
   torch.save(model.state_dict(), f"checkpoint_epoch{epoch}.pt")
```

这段代码初始化模型至 GPU，AdamW 优化器设置 LR 和 decay。DistributedSampler 确保多进程数据均匀。循环中 zero_grad 清零梯度，前向计算 logits（移位输入），cross_entropy 损失展平计算（view(-1) 转为 1D）。backward 反传，clip_grad_norm_ 裁剪梯度防爆炸（max_norm=1.0），step 更新。wandb.log 记录损失，save checkpoint 支持 resume。监控指标包括 loss 曲线（应平稳下降至 2.0 左右）、perplexity（exp(loss)）和 throughput（tokens/sec/GPU，通过 timer 计算）。

5 训练过程与优化技巧

训练分阶段：前 80% 为纯预训练自监督，之后扩展上下文用 YaRN 或 ALiBi 位置编码，支持更长序列；实现早停基于验证 perplexity，并用 torch.load resume 从 checkpoint 恢复：model.load_state_dict(torch.load(checkpoint.pt))。
常见问题包括 loss 爆炸，通常因 LR 过高，用 gradient clipping 解决，如上代码所示；OOM（Out of Memory）时 batch 太大，启用 ZeRO-Offload 将状态卸载至 CPU；收敛慢源于数据质量，解决方案是多轮 shuffle 或增广合成数据。
加速技巧有 BF16 混合精度：torch.autocast(cuda, dtype=torch.bfloat16)，Torch 2.0 dynamo（torch.compile(model)）和自定义 kernel 如 FlashAttention。成本估算：1B 模型在 A100 租赁（$1/小时/GPU)$下，约 1000 GPU 小时，即 $1000$1000。

6 评估与迭代

评估从内在指标入手，零样本 perplexity 于留出验证集：计算 held-out 数据损失并 exp 得到 PPL，优秀模型应达 10 以下。外在评估用 EleutherAI eval harness 测试下游任务如 GLUE 或 MMLU：pip install lm-eval，运行 lm_eval –model hf –model_args pretrained=model_path –tasks mmlu。
可视化 loss/perplexity 曲线确认收敛，注意力热图用 matplotlib.imshow 显示头关注模式。迭代循环基于评估调整：PPL 高则优化数据，重训超参。

7 部署与下一步

导出模型至 Hugging Face Hub：model.push_to_hub(your-llm-base)。量化用 GGUF 4-bit 加速推理，减少内存。推理优化 vLLM 或 TGI，支持 paged attention；进一步 AWQ/GPTQ 量化至 INT4。
下一步从 SFT（监督微调）转向指令模型，用 Axolotl 框架一键 RLHF。
从零训练 LLM 的核心是数据质量、计算资源和工程实践，坚持 Chinchilla 定律与分布式工具，你将收获自定义基座模型。未来挑战包括 MoE 稀疏架构、合成数据和多模态扩展。
行动起来：fork 本文代码仓库（GitHub: yourusername/llm-from-scratch），分享你的 100M Colab demo 结果！资源包括论文《GPT-3》、《LLaMA》、《Chinchilla》；工具 nanoGPT、Lit-GPT、Hugging Face Accelerate；社区 EleutherAI Discord 和 HF 论坛。
FAQ：Q: 小团队如何起步？A: 从 100M 模型 + Colab Pro 用 nanoGPT。Q: 数据从哪来？A: RedPajama 开源免费。Q: 成本超支？A: 监控 throughput，优先 BF16。