The Smol Training Playbook: The Secrets to Building World-Class LLMs
Smol 培训手册：构建世界级 LLM 的秘诀

A practical journey through the challenges, decisions, and messy reality behind training state-of-the-art language models

穿越训练最先进语言模型背后的挑战、决策和混乱现实的实践之旅

AUTHORS 作者

Loubna Ben Allal, Lewis Tunstall, 刘易斯·坦斯托尔,
卢布纳·本·阿拉尔,
Nouamane Tazi, 努阿曼·塔齐,
Elie Bakouch, 埃利·巴库奇, Ed Beeching, 埃德·比钦,
Carlos Miguel Patiño,
卡洛斯·米格尔·帕蒂尼奥,
Clémentine Fourrier, 克莱门汀·福里尔,
Thibaud Frere, 蒂博·弗雷尔,
Anton Lozhkov, 安东·洛日科夫,
Colin Raffel, 科林·拉菲尔, Leandro von Werra,
莱安德罗·冯·韦拉,
Thomas Wolf 托马斯·沃尔夫

AFFILIATION 联系

Hugging Face 拥抱脸

PUBLISHED 发表

Oct. 30, 2025 10 月 30, 2025

PDF PDF 格式 - pro only 仅限专业版

Subscribe to Pro 订阅专业版

Table of Contents 目录

Introduction 介绍

Reading time: 2-4 days.

阅读时间：2-4 天。

What does it actually take to train a high-performance LLM today?

如今，训练高性能 LLM 实际上需要什么？

Published research makes it look straightforward: strategic architecture choices, carefully curated datasets, and sufficient compute. The results are polished, the ablations are structured and clean. Every decision seems obvious in hindsight.

已发表的研究使它看起来很简单：战略架构选择、精心策划的数据集和足够的计算。结果经过抛光，消融结构化且干净。事后看来，每一个决定似乎都是显而易见的。

But those reports only show what worked and apply a bit of rosy retrospection – they don't capture the 2am dataloader debugging sessions, the loss spikes, or the subtle tensor parallelism bug (see later!) that quietly sabotages your training.

但这些报告只显示了哪些是有效的，并应用了一些美好的回顾——它们没有捕捉到凌晨 2 点的数据加载器调试会话、损失峰值或悄悄破坏训练的微妙张量并行性错误（见后文！

The reality is messier, more iterative, and full of decisions that don't make it into the final paper.

现实更加混乱，更加迭代，并且充满了没有进入最终论文的决定。

Join us as we look behind the scenes of training SmolLM3, a 3B multilingual reasoning

model trained on 11T tokens. This is not an ordinary blog post, but rather the untangling of a spiderweb of decisions, discoveries, and dead ends that led to deep insights into what it takes to build world-class language models.

加入我们，了解训练 SmolLM3 的幕后花絮，这是一种在 3T 标记上训练的 11B 多语言推理模型。这不是一篇普通的博客文章，而是解开了决策、发现和死胡同的蜘蛛网，从而深入了解构建世界级语言模型需要什么。

It is also the final opus in our model-training long-form series: we've worked through building datasets at scale (FineWeb), orchestrating thousands of GPUs to sing in unison (Ultra Scale Playbook), and selecting the best evaluations at each step of the process (Evaluation Guidebook). Now we shape it all together to build a strong AI model. We'll walk you through the complete journey – not just the final recipe that worked, but the failures, infrastructure breakdowns, and debugging processes that shaped every decision.

这也是我们模型训练长篇系列的最后一部作品：我们通过大规模构建数据集（FineWeb），编排数千个 GPU 齐声唱歌（Ultra Scale Playbook），并在流程的每个步骤中选择最佳评估（评估指南）。现在，我们共同塑造这一切，构建一个强大的 AI 模型。我们将引导您完成完整的旅程——不仅仅是最终有效的秘诀，还有影响每个决策的故障、基础设施故障和调试过程。

The story reads like a drama: you'll see how promising small-scale ablations sometimes don't translate at scale, why we restarted a training after 1T tokens, how we balanced the competing objectives of multilinguality, math, and code while maintaining strong English performance, and finally how we post-trained a hybrid reasoning model.

这个故事读起来就像一部戏剧：你会看到有前途的小规模消融有时无法大规模转化，为什么我们在 1T 标记后重新开始训练，我们如何在保持强大的英语表现的同时平衡多语言、数学和代码的竞争目标，最后我们如何对混合推理模型进行后期训练。

We also tried to avoid a cold list of all we did in favour of an organized story through our adventure. Think of this as a guide for anyone trying to go from "we have a great dataset and GPUs" to "we built a really strong model".

我们还试图避免列出我们所做的一切的冷清单，转而通过我们的冒险来讲述一个有条理的故事。将此视为任何试图从"我们拥有出色的数据集和 GPU"转变为"我们构建了一个非常强大的模型"的人的指南。

We hope being this open will help close the gap between research and production, and make your next training run a little less chaotic.

我们希望这种开放将有助于缩小研究和生产之间的差距，并使您的下一次培训运行不那么混乱。

How to read this blog post
如何阅读这篇博文

You don't need to read this blog post from top to bottom, and at this point it's too long to realistically read end-to-end in one sitting anyway. The blog post is structured in several distinct pieces that can be skipped or read individually:

你不需要从上到下阅读这篇博文，而且在这一点上，它太长了，无论如何都无法一次真实地阅读端到端。这篇博文由几个不同的部分组成，可以跳过或单独阅读：

- **Training compass:** A high-level discussion about whether or not you should pretrain your own model. We walk you through fundamental questions to ask yourself before burning through all your VC money, and how to think systematically through the decision process.

  训练指南针：关于是否应该预训练自己的模型的高级讨论。我们将引导您了解在耗尽所有风险投资资金之前要问自己的基本问题，以及如何在决策过程中系统地思考。

  This is a high-level section, if you want to skip straight to the technical content, scroll quickly past this part.

  这是一个高级部分，如果您想直接跳到技术内容，请快速滚动通过这部分。

- **Pretraining:** The sections following the training compass cover everything you need to know to build a solid recipe for your own pretraining run: how to run ablations, select evaluations, mix data sources, make architecture choices, tune hyperparameters, and finally endure the training marathon.

  预训练：训练指南针之后的部分涵盖了为自己的预训练运行构建可靠配方所需了解的所有内容：如何运行消融、选择评估、混合数据源、做出架构选择、调整超参数，以及最后忍受训练马拉松。

  This section also applies if you're not planning to pretrain from scratch but are interested in continued pretraining (aka mid-training).

如果您不打算从头开始进行预训练，但对继续进行预训练（又称中期训练）感兴趣，则本部分也适用。

- **Post-training:** In this part of the blog you'll learn all the tricks needed to get most out of your pretrained models. Learn the whole post-training alphabet starting with SFT, DPO and GRPO as well as the dark arts and alchemy of model merging.

  训练后：在博客的这一部分中，您将学习充分利用预训练模型所需的所有技巧。学习从 SFT、DPO 和 GRPO 开始的整个训练后字母表，以及模型合并的黑魔法和炼金术。

  Most of the knowledge about making these algorithms work well is learned through painful lessons, and we'll share our experience here to hopefully spare you some of them.

  关于使这些算法正常工作的大部分知识都是通过痛苦的教训学到的，我们将在这里分享我们的经验，希望能让您免于其中的一些经验。

- **Infrastructure:** If pretraining is the cake and post-training is the icing and cherry on top, then infrastructure is the industrial-grade oven. Without it, nothing happens, and if it's broken, your happy Sunday baking session turns into a fire hazard.

  基础设施：如果预培训是蛋糕，培训后是锦上添花，那么基础设施就是工业级烤箱。没有它，什么也不会发生，如果它坏了，你快乐的周日烘焙过程就会变成火灾隐患。

  Knowledge about how to understand, analyse, and debug GPU clusters is scattered across the internet in various libraries, docs, and forums. This section walks through GPU layout, communication patterns between CPU/GPU/nodes/storage, and how to identify and overcome bottlenecks.

  有关如何理解、分析和调试 GPU 集群的知识分散在互联网上的各种库、文档和论坛中。本节将介绍 GPU 布局、CPU/GPU/节点/存储之间的通信模式，以及如何识别和克服瓶颈。

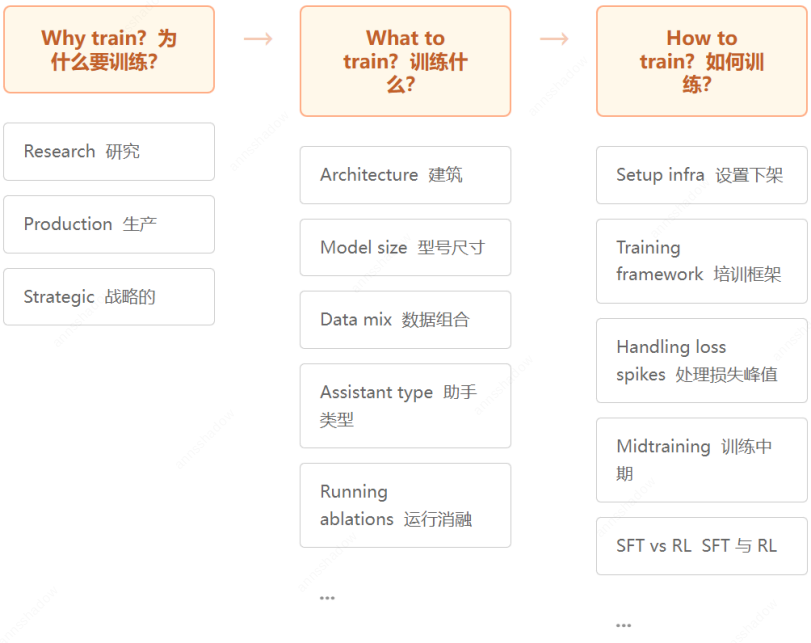
So where do we even start? Pick the section that you find most exciting and let's go!

那么我们从哪里开始呢？选择您觉得最令人兴奋的部分，然后开始吧！

If you have questions or remarks, open a discussion on the Community tab!

如果您有任何问题或意见，请在"社区"选项卡上展开讨论！

Training compass: why → what → how
训练指南针：为什么→什么→如何



The field of machine learning has an obsessive relationship with optimisation. We fixate on loss curves, model architectures, and throughput; after all, machine learning is fundamentally about optimising the loss function of a model.

机器学习领域与优化有着密切的关系。我们关注损耗曲线、模型架构和吞吐量;毕竟，机器学习从根本上讲是关于优化模型的损失函数。

Yet before diving into these technical details, there's a more fundamental question that often goes unasked:*should we even be training this model?*

然而，在深入研究这些技术细节之前，有一个更基本的问题经常被忽视：我们是否应该训练这个模型？

As shown in the heatmap below, the open-source AI ecosystem releases world-class models on a nearly daily basis: Qwen, Gemma, DeepSeek, Kimi, Llama 🪦, Olmo, and the list grows longer every month.

如下图热图所示，开源 AI 生态系统几乎每天都会发布世界级的模型：Qwen、Gemma、DeepSeek、Kimi、Llama 🪦、Olmo，而且这个列表每个月都在增长。

These aren't just research prototypes or toy examples: they're production-grade models covering an astonishing breadth of use cases from multilingual understanding to code generation and reasoning.

这些不仅仅是研究原型或玩具示例：它们是生产级模型，涵盖了从多语言理解到代码生成和推理的惊人广泛用例。

Most of them come with permissive licenses and active communities ready to help you use them.

它们中的大多数都带有宽松的许可证和活跃的社区，随时可以帮助您使用它们。

Which raises an uncomfortable truth: maybe you *don't need to train your own model*

这提出了一个令人不安的事实：也许你不需要训练自己的模型 .

This might seem like an odd way to start an "LLM training guide". But many failed training projects didn't fail because of bad hyperparameters or buggy code, they failed because someone decided to train a model they didn't need. So before you commit to training, and dive into *how* to execute it, you need to answer two questions: *why* are you training this model? And *what* model should you train? Without clear answers, you'll waste months of compute and engineering time building something the world already has, or worse, something nobody needs.

这似乎是开始"LLM 培训指南"的一种奇怪方式。但许多失败的训练项目失败并不是因为糟糕的超参数或错误的代码，而是因为有人决定训练他们不需要的模型而失败。因此，在您承诺进行训练并深入了解如何执行它之前，您需要回答两个问题：为什么要训练这个模型？你应该训练什么模型？如果没有明确的答案，您将浪费数月的计算和工程时间来构建世界上已经拥有的东西，或者更糟糕的是，没有人需要的东西。

Let's start with the *why*, because without understanding your purpose, you can't make coherent decisions about anything that follows.

让我们从 为什么 开始，因为如果不了解您的目的，您就无法对接下来的任何事情做出连贯的决定。

> 📍 **About this section 关于本节**
>
> This section is different from the rest of the blog: it's less about experiments and technical details, more about strategic planning. We'll guide you through deciding **whether you need to train from scratch and what model to build.** If you've already thought deeply about your why and what, feel free to jump to the Every big model starts with a small ablation chapter for the technical deep-dive. But if you're uncertain, investing time here will save you a lot of effort later.
>
> 本节与博客的其余部分不同：它不是关于实验和技术细节，而是关于战略规划。我们将指导您决定是否需要从头开始训练以及构建什么模型。如果您已经深入思考了原因和内容，请随时跳转到每个大模型都从一个小的消融章节开始，进行技术深入探讨。但如果您不确定，在这里投入时间将为您以后节省很多精力。

Why: the question nobody wants to answer
为什么：没人愿意回答的问题

Let's be blunt about what happens in practice. Someone (if they're lucky) gets access to a GPU cluster, maybe through a research grant, maybe through a company's spare capacity, and the thought process goes roughly like this: "We have 100 H100s for three months.

让我们直言不讳地谈谈实践中发生的事情。有人（如果幸运的话）可以通过研究资助，也许通过公司的闲置容量来访问 GPU 集群，思维过程大致是这样的："我们有 100 个 H100，为期三个月。

Let's train a model!" The model size gets picked arbitrarily, the dataset gets assembled from whatever's available. Training starts.

我们来训练一个模型吧！模型大小是任意选择的，数据集是从可用的任何东西中组装的。培训开始。

And six months later, after burning through compute budget and team morale, the resulting model sits unused because nobody ever asked *why.*

六个月后，在耗尽了计算预算和团队士气之后，最终的模型没有被使用，因为没有人问过为什么。

Here are some reasons why you shouldn't train a model:

以下是不应训练模型的一些原因：

We have compute available
我们提供可用的计算

↳ *That's a resource, not a goal*
这是一种资源，而不是目标

Everyone else is doing it
其他人都在这样做

↳ *That's peer pressure, not strategy*
这是同侪压力，而不是策略

AI is the future
人工智能是未来

↳ *That's a platitude, not a plan*
这是陈词滥调，而不是计划

We want the best model possible
我们想要最好的模型

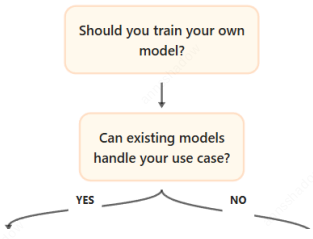
↳ *That's not specific enough to guide decisions*
这还不够具体，无法指导决策

The allure of "we trained our own model" is powerful, but before investing a lot of time and resources, it makes sense to ask: **why do you need to train this model?**

"我们训练了自己的模型"的诱惑力是强大的，但在投入大量时间和资源之前，有理由问：为什么需要训练这个模型？

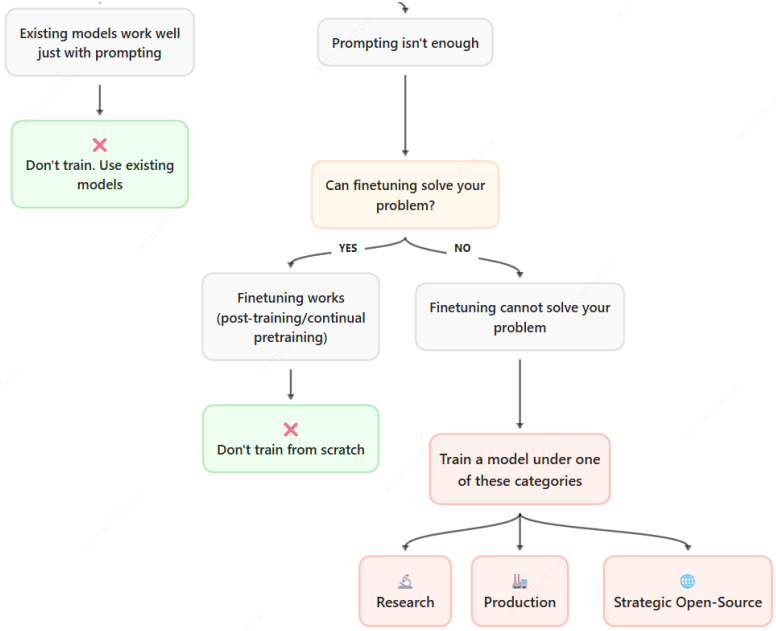
The flowchart below guides the thought process one should go through before starting a big pretraining project. From a technical perspective, you should essentially first find out if there isn't an existing model that you can either prompt of fine-tune to do the job.

下面的流程图指导了在开始大型预训练项目之前应该经历的思维过程。从技术角度来看，您基本上应该首先找出是否没有可以提示微调以完成这项工作的现有模型。



The "why" we discuss is about training from scratch. We don't cover distillation or pruning in this blog. These are valid paths to efficient models but represent different workflows than training from scratch. We recommend NVIDIA's Minitron paper for an overview of these topics.

我们讨论的"为什么"是关于从头开始的训练。我们不在此博客中介绍蒸馏或修剪。这些是通往高效模型的有效途径，但代表了与从头开始训练不同的工作流程。我们推荐 NVIDIA 的 Minitron 论文来概述这些主题。



There are essentially three common areas where custom pretraining can make sense: you want to do novel research, you have very specific needs for production use-case, or you want to fill a gap in the open model ecosystem. Let's have a quick look at each:

自定义预训练基本上有三个共同领域是有意义的：你想做新颖的研究，你对生产用例有非常具体的需求，或者你想填补开放模型生态系统中的空白。让我们快速浏览一下每个：

RESEARCH: WHAT DO YOU WANT TO UNDERSTAND?
研究：你想了解什么？

There is plenty of research one can do in the LLM space. What LLM research projects have in common is that you normally start with a clearly defined question:

在法学硕士领域可以做很多研究。法学硕士研究项目的共同点是，您通常从一个明确定义的问题开始：

- Can we scale training on this new optimiser to a 10B+ model? From Muon is Scalable for LLM Training

  我们能否将这个新优化器的训练扩展到 10B+ 模型？From Muon 可扩展用于 LLM 培训

- Can reinforcement learning alone, without SFT, produce reasoning capabilities? From DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning

  没有 SFT 的强化学习能否单独产生推理能力？来自 DeepSeek-R1：通过强化学习激励 LLM 的推理能力

- Can we train good small models on purely synthetic textbooks data? From Textbooks Are All You Need

  我们能用纯合成的教科书数据训练好的小模型吗？从教科书就是你所需要的

- Can we achieve competitive performance by training on only openly licensed data? From The Common Pile v0.1: An 8TB Dataset of Public Domain and Openly Licensed Text

  我们能否通过仅使用开放许可的数据进行训练来获得有竞争力的性能？来自 The Common Pile v0.1：一个 8TB 的公共领域和开放许可文本数据集

Making the hypothesis as concrete as possible and thinking about the necessary experiment scale increases the chance of success.

使假设尽可能具体并考虑必要的实验规模可以增加成功的机会。

PRODUCTION: WHY CAN'T YOU USE AN EXISTING MODEL?
生产：为什么不能使用现有模型？

There are mainly three reasons why companies can't use off the shelf models for their use-case. Two of them are technical and the other is due to governance.

公司无法将现成的型号用于其用例的原因主要有三个。其中两个是技术性的，另一个是由于治理。

The first reason to train your own model is **domain specificity:** when your data or tasks involve highly specialized vocabulary or structure that existing models can't handle well. For example:

训练自己的模型的第一个原因是领域特异性：当数据或任务涉及现有模型无法很好地处理的高度专业化词汇表或结构时。例如：

- A DNA model with a unique vocabulary and long-range dependencies.

  具有独特词汇和远程依赖关系的 DNA 模型。

- A legal or financial model requiring deep familiarity with domain-specific jargon and logic.

  需要深入了解特定领域术语和逻辑的法律或财务模型。

A second, related reason is deployment constraints: when you need a model tailored to your hardware, latency, or privacy requirements. For instance, an LLM running on a drone or on-prem system with custom hardware like FPGAs.

第二个相关原因是部署限制：当您需要根据硬件、延迟或隐私要求定制模型时。例如，在无人机或本地系统上运行的 LLM，具有 FPGA 等定制硬件。

Here's a simple test: spend a few days building on top of Qwen3, Gemma3, or another current SOTA model. Can you reach your performance goals through prompting, tool-use, or post-training? If not, it's probably time to train your own.

这是一个简单的测试：花几天时间在 Qwen3、Gemma3 或其他当前 SOTA 模型之上构建。你可以通过提示、工具使用或后期培训来达到你的绩效目标吗？如果没有，可能是时候训练自己的了。

Even if the post-training budget needed to meet your requirements is immense, it might still be cheaper than starting from scratch. Fine-tuning your model for 1T tokens is still more economic than starting from scratch to train for 10T+ tokens.

At this point LLM trainers start to miraculously call it mid-training instead of post-training

此时，LLM 培训师开始奇迹般地将其称为培训中期而不是培训后

即使满足您的要求所需的培训后预算是巨大的，它仍然可能比从头开始便宜。针对 1T 代币微调模型仍然比从头开始为 10T+ 代币进行训练更经济。

The third reason to build you own in-house language model is **safety and governance:** You need complete control over training data, model behaviour, and update cycles because you're in a regulated industry or high-stakes application. You need to know *exactly* what went into the model and be able to prove it to regulators. In some cases you might have no other option than building your own model.

构建自己的内部语言模型的第三个原因是安全性和治理：您需要完全控制训练数据、模型行为和更新周期，因为您处于受监管的行业或高风险应用程序中。您需要确切地知道模型中的内容，并能够向监管机构证明这一点。在某些情况下，除了构建自己的模型之外，您可能别无选择。

These are the main reasons companies train in-house models, but what about companies or organisations that release open models?

这些是公司培训内部模型的主要原因，但是发布开放模型的公司或组织呢?

STRATEGIC OPEN-SOURCE: DO YOU SEE A GAP YOU CAN FILL?
战略开源：你认为可以填补的空白吗?

One of the most common reasons experienced AI labs release new open models is that they've identified a specific gap or new AI use-case in the open-source ecosystem.

经验丰富的 AI 实验室发布新的开放模型的最常见原因之一是他们已经确定了开源生态系统中的特定差距或新的 AI 用例。

The pattern typically looks like this; you notice an under explored area, maybe there are no strong on-device models with very long context, or multilingual models exist but they're weak on low resource languages, or the field is moving towards interactive world-models like Genie3 and no good open-weight model exists.

该模式通常如下所示;你注意到一个未被探索的领域，也许没有具有很长上下文的强大设备上模型，或者存在多语言模型，但它们在低资源语言上很弱，或者该领域正在向像 Genie3 这样的交互式世界模型发展，并且不存在好的开放权重模型。

You have reasons to believe you can do better; perhaps you've curated better training data, developed better training recipes, or have the compute to overtrain where others couldn't.

你有理由相信你可以做得更好;也许你已经策划了更好的训练数据，开发了更好的训练配方，或者拥有在其他人无法做到的地方进行过度训练的计算能力。

Your goal is concrete: not “the best model ever,” but “the best 3B model for on-device use” or “the first small model with 1M context”.

你的目标是具体的：不是“有史以来最好的模型”，而是“设备上使用的最佳 3B 模型”或“第一个具有 1M 上下文的小型模型”。

This is a real goal and success creates value: developers adopt your model, it becomes infrastructure for others, or it establishes technical credibility. But success requires experience.

这是一个真正的目标，成功创造了价值：开发人员采用您的模型，它成为其他人的基础设施，或者它建立了技术信誉。但成功需要经验。

You need to know what’s actually feasible and how to execute reliably in a competitive space. To make this concrete, let’s look at how we think about this question at Hugging Face.

您需要知道什么是实际可行的，以及如何在竞争激烈的空间中可靠地执行。为了具体化这一点，让我们看看我们是如何看待 Hugging Face 的这个问题的。

HUGGING FACE’S JOURNEY 拥抱脸的旅程

So why does Hugging Face train open models? The answer is simple: we build things that are useful to the open-source ecosystem and fill gaps that few others are filling.

那么为什么 Hugging Face 要训练开放式模型呢？答案很简单：我们构建了对开源生态系统有用的东西，并填补了很少有人填补的空白。

Although there are millions of open-weight models, there are very few organisations that train fully open models. In addition to Hugging Face, there’s Ai2 and Stanford’s Marin community.

尽管有数百万个开放权重模型，但训练完全开放模型的组织却很少。除了 Hugging Face，还有 Ai2 和斯坦福大学的 Marin 社区。

This includes datasets, tooling and training models. Every LLM training project we’ve started began with noticing a gap and believing we could contribute something meaningful.

这包括数据集、工具和训练模型。我们启动的每个 LLM 培训项目都是从注意到差距并相信我们可以做出有意义的事情开始的。

We started our first LLM project after GPT-3 (Brown et al., 2020) was released. At the time, it felt like no one else was building an open alternative, and we were worried that the knowledge would end up locked away in just a few industry labs. So we launched the BigScience workshop to train an open version of GPT-3. The resulting model was Bloom, and came from dozens of contributors working for a year to build the training stack, tokenizer, and pretraining corpus to pre-train a 175B parameter model.

我们在 GPT-3（Brown 等人，2020 年）发布后开始了我们的第一个 LLM 项目。当时，感觉没有其他人在构建一个开放的替代方案，我们担心这些知识最终会被锁定在几个行业实验室中。因此，我们启动了 BigScience 研讨会来训练 GPT-3 的开放版本。由此产生的模型是 Bloom，它来自数十名贡献者，他们花了一年时间构建训练堆栈、分词器和预训练语料库，以预训练 175B 参数模型。

The successor of Bloom was StarCoder in 2022 (Li et al., 2023). OpenAI had developed Codex for GitHub Copilot (Chen et al., 2021), but it was closed-source. Building an open-source alternative clearly would provide value to the ecosystem. So in collaboration with ServiceNow, under the BigCode umbrella, we built The Stack dataset, and we trained StarCoder 15B to reproduce Codex. StarCoder2 (Lozhkov et al., 2024) came from learning we could have trained longer, and recognising that smaller models trained longer might be more valuable than one large model. We trained a family (3B/7B/15B) on multiple trillions of tokens, far beyond what anyone had done for open code models at the time.

Bloom 的继任者是 2022 年的 StarCoder（Li 等人，2023 年）。OpenAI 为 GitHub Copilot 开发了 Codex（Chen 等人，2021 年），但它是闭源的。构建开源替代方案显然将为生态系统提供价值。因此，我们与 ServiceNow 合作，在 BigCode 旗下构建了 The Stack 数据集，并训练了 StarCoder 15B 来重现 Codex。StarCoder2（Lozhkov 等人，2024 年）来自于了解我们可以训练更长时间，并认识到训练时间更长的小型模型可能比一个大型模型更有价值。我们在数万亿个代币上训练了一个家族（3B/7B/15B），这远远超出了当时任何人对开放代码模型所做的。

The SmolLM family followed a similar pattern. We noticed there were very few strong small models and we had just built FineWeb-Edu (Penedo et al., 2024), which was a strong pre-training dataset. SmolLM (135M/360M/1.7B) was our first version. SmolLM2 (Allal et al., 2025) focused on better data and training longer, reaching SOTA performance on multiple fronts. SmolLM3 scaled to 3B while adding hybrid reasoning, multilinguality and long context, features that the community values in 2025.

SmolLM 家族遵循类似的模式。我们注意到强大的小模型很少，我们刚刚构建了 FineWeb-Edu（Penedo 等人，2024 年），这是一个强大的预训练数据集。SmolLM（135M/360M/1.7B）是我们的第一个版本。SmolLM2（Allal et al.，2025）专注于更好的数据和更长的训练时间，在多个方面达到 SOTA 性能。SmolLM3 扩展到 3B，同时增加了混

合推理、多语言和长上下文，这些功能是社区在 2025 年重视的功能。

This pattern extends beyond pretraining: we trained Zephyr (Tunstall et al., 2023) to show DPO works at scale, started Open-R1 to reproduce DeepSeek R1's distillation pipeline and released OlympicCoder for competitive programming, with SOTA performance in the International Olympiad in Informatics. We've also explored other modalities with SmolVLM (Marafioti et al., 2025) for vision and SmolVLA (Shukor et al., 2025) for robotics.

这种模式超出了预训练的范围：我们训练了 Zephyr（Tunstall 等人，2023 年）以展示 DPO 的大规模工作，启动了 Open-R1 来重现 DeepSeek R1 的蒸馏管道，并发布了用于竞争性编程的 OlympicCoder，并在国际信息学奥林匹克竞赛中表现出色。我们还探索了其他模式，包括用于视觉的 SmolVLM（Marafioti 等人，2025 年）和用于机器人的 SmolVLA（Shukor 等人，2025 年）。

If you're curious about the HF science projects, you can find an overview here https://huggingface.co/science

如果您对 HF 科学项目感到好奇，可以在此处找到概述 https://huggingface.co/science

Hopefully, this section has convinced you that there is value in thinking deeply about why you want to train a model.

希望本节能让您相信，深入思考为什么要训练模型是有价值的。

For the rest of this blog post, we'll assume you've done this soul-searching and have a legitimate reason to train.

在这篇博文的其余部分，我们将假设您已经进行了自我反省，并且有正当理由进行培训。

What: translating goals into decisions
内容：将目标转化为决策

Now that you know *why* you're training, what should you train? By "what", we mean: model type (dense, MoE, hybrid, something new), model size, architecture details and data mixture. Once you've settled on the why, you can derive the what, for example:

既然你知道你为什么要训练，你应该训练什么？我们所说的"什么"是指：模型类型（密集、MoE、混合、新事物）、模型大小、架构细节和数据混合。确定了 why 后，您可以推导出 what，例如：

- fast model for on device → small efficient model

  设备上的快速模型→小型高效模型

- multilingual model → large tokenizer vocab

  多语言模型→大量分词器词汇

- super long context → hybrid architecture

  超长上下文→混合架构

Besides decisions driven by the use-case, there are also some choices that optimise the training itself, either by being more stable, more sample efficient, or faster. These decisions are not always so clear cut, but you can divide the decision process roughly into two phases:

除了由用例驱动的决策外，还有一些选择可以优化训练本身，要么更稳定，要么更高效，要么更快。这些决策并不总是那么明确，但您可以将决策过程大致分为两个阶段：

Planning: Before running experiments, map your use case to the components you need to decide on. Your deployment environment determines model size constraints. Your timeline determines which architectural risks you can take. Your target capabilities determine dataset requirements.

规划：在运行实验之前，将您的用例映射到您需要决定的组件。您的部署环境确定模型大小约束。您的时间表决定了您可以承担哪些架构风险。您的目标功能决定了数据集要求。

This phase is about connecting each constraint from your "why" to concrete specifications in your "what.

这个阶段是关于将"为什么"中的每个约束连接到"什么"中的具体规范。

"

Validation: Once you have a starting point and a list of potential modifications, test systematically. Since testing is expensive, focus on changes that could meaningfully improve performance for your use case or optimise your training. This is where ablations come in, covered in the ablations section.

验证：一旦有了起点和潜在修改列表，就可以系统地进行测试。由于测试成本高昂，因此请专注于可以显着提高用例性能或优化训练的更改。这就是消融的用武之地，在消融部分中介绍。

> 📍 **Learn to identify what's worth testing, not just how to run tests.**
>
> **学会确定值得测试的内容，而不仅仅是如何运行测试。**
>
> Perfect ablations on irrelevant choices waste as much compute as sloppy ablations on important ones.
>
> 对不相关选择的完美消融浪费的计算与对重要选择的草率消融一样多。

In the following chapters you will learn about all the options you have to define your model and how to narrow down the choices with systematic experiments.

在以下章节中，您将了解定义模型所需的所有选项，以及如何通过系统实验缩小选择范围。

Before going there we want to share a few learnings on how to setup teams and projects from training our own models as well as observing amazing other teams building great LLMs.

在去那里之前，我们想分享一些关于如何设置团队和项目的经验，包括训练我们自己的模型，以及观察其他令人惊叹的团队构建出色的 LLM。

Super power: speed and data
超能力：速度和数据

Of course there are many ways to get to Rome, but we've found that what consistently sets successful LLM training teams apart is **iteration speed.** Training LLMs is really a learning by training discipline, the more often you train, the better your team will become. So between the teams that train a model a year and the ones that train one per quarter, the latter will improve so much faster.

当然，到达罗马的方式有很多，但我们发现，成功的 LLM 培训团队始终与众不同的是迭代速度。培训法学硕士实际上是一门循规累的学习，你训练的次数越多，你的团队就会变得越好。因此，在每年训练一个模型的团队和每季度训练一个模型的团队之间，后者的改进速度会快得多。

You can look at the teams from Qwen and DeepSeek for example. Now household names, they have a long track record of consistently releasing new models on a fast cadence.

例如，您可以查看 Qwen 和 DeepSeek 的团队。现在是家喻户晓的名字，他们在不断快速发布新车型方面有着悠久的记录。

Besides iteration speed, by far the most influential aspect of LLM training is **data curation.** There's a natural tendency to dive into architecture choices to improve the model, but the teams that excel in LLM training are the ones that are obsessed with high quality data more than anything else.

除了迭代速度之外，到目前为止，LLM 训练最有影响力的方面是数据管理。人们自然会深入研究架构选择来改进模型，但在 LLM 培训方面表现出色的团队是那些最痴迷于高质量数据的团队。

Another aspect that is tied to iteration speed is the team size: for the main pretraining tasks you only need a handful of people equipped with enough compute to execute. To pre-train a model like Llama 3 today you probably only need 2-3 people.

与迭代速度相关的另一个方面是团队规模：对于主要的预训练任务，您只需要少数配备足够计算的人来执行。要预训练像 Llama 3 这样的模型，今天你可能只需要 2-3 个人。

Only once you start to venture into more diverse trainings and downstream tasks (multimodal, multilingual, post-training etc) will you slowly need to add a few more people to excel at each domain.

只有当你开始冒险进行更多样化的培训和下游任务（多模式、多语言、培训后等）时，你才会慢慢需要增加一些人来在每个领域表现出色。

So start with a small, well equipped team, and build a new model every 2-3 months and within short amount of time you'll climb to the top. Now the rest of the blog will focus on the technical day-to-day of this team!

因此，从一个装备精良的小团队开始，每 2-3 个月构建一个新模型，在很短的时间内你就会爬到顶峰。现在，博客的其余部分将重点介绍这个团队的日常技术！

Every big model starts with a small ablation

每个大模型都是从小消融开始的

Before we can start training an LLM, we need to make many decisions that will shape the model's performance and training efficiency. Which architecture will best serve our use case? What optimiser and learning rate schedule to use and which data sources to mix in?

在开始训练 LLM 之前，我们需要做出许多决定，这些决定将影响模型的性能和训练效率。哪种架构最适合我们的用例？要使用什么优化器和学习率计划，以及要混合哪些数据源？

How these decisions are made is a frequently asked question. People sometimes expect that they are made by thinking deeply about them. And while strategic thinking is essential—as we covered in the previous section where we discussed identifying which architecture changes are worth testing—reasoning alone isn't enough. Things are not always intuitive with LLMs, and hypotheses about what should work sometimes don't pan out in practice.

如何做出这些决定是一个经常被问到的问题。人们有时期望它们是通过深入思考而产生的。虽然战略思维至关重要——正如我们在上一节中讨论的那样，我们讨论了确定哪些架构更改值得测试——但仅靠推理是不够的。法学硕士的事情并不总是直观的，关于什么应该有效的假设有时在实践中并不成功。

For example, using what seems like "the highest quality data" doesn't always yield stronger models. Take arXiv for example, which is a vast collection of humanity's scientific knowledge. Intuitively, training on such rich STEM data should produce superior models, right? In practice, it doesn't and especially for smaller models, where it can even hurt performance (Shao et al., 2024). Why? The reason is that while arXiv papers are full of knowledge, they're highly specialized and written in a narrow

academic style that's quite different from the diverse, general text that models learn best from.

例如，使用看似"最高质量的数据"并不总是产生更强大的模型。以 arXiv 为例，它是人类科学知识的大量集合。直观地说，对如此丰富的 STEM 数据进行训练应该会产生更好的模型，对吧？在实践中，它不会，尤其是对于较小的型号，它甚至会损害性能（Shao 等人，2024 年）。为什么？原因是，虽然 arXiv 论文充满了知识，但它们高度专业化，并且以狭隘的学术风格编写，与模型从中学习最多的多样化、通用文本有很大不同。

So, how can we know what works if staring at the problem long and hard doesn't help? We run a lot of experiments, like good empiricists! Machine learning is not pure math, but actually very much an experimental science.

那么，如果长时间认真地盯着问题没有帮助，我们怎么知道什么有效呢？我们进行了很多实验，就像优秀的经验主义者一样！机器学习不是纯粹的数学，而实际上是一门实验科学。

In many ways, machine learning resembles thermodynamics before the discovery of statistical mechanics: we have reliable empirical laws and design principles that work remarkably well, even if deeper theoretical explanations are still emerging.

在许多方面，机器学习类似于统计力学发现之前的热力学：我们拥有可靠的经验定律和设计原则，即使更深层次的理论解释仍在出现，它们仍然运行得非常好。

Since those experiments will guide many of our crucial decisions, it is really important to set them up well. There are essentially two main attributes we want from them:

由于这些实验将指导我们的许多关键决策，因此做好它们非常重要。我们基本上希望从它们那里获得两个主要属性：

1. **Speed:** they should run as fast as possible so we can iterate often. The more ablations we can run, the more hypotheses we can test.

   速度：它们应该尽可能快地运行，以便我们可以经常迭代。我们可以运行的消融越多，我们可以检验的假设就越多。

2. **Reliability:** they should provide strong discriminative power. If the metrics we look at can't meaningfully distinguish between different setups early on, our ablations may reveal little (and if they're noisy, we risk chasing noise!). For more details, check out the FineTaks blog post.

   可靠性：它们应该提供强大的辨别力。如果我们查看的指标无法在早期有意义地区分不同的设置，我们的消融可能几乎没有什么信息（如果它们很吵，我们就有可能追逐噪音！有关更多详细信息，请查看 FineTaks 博客文章。

But before we can set up our ablations, we need to make some foundational choices about architecture type and model size. These decisions—guided by our compass—impact which training framework to use, how to allocate our compute budget, and which baseline to start from.

但在设置烧蚀之前，我们需要对架构类型和模型大小做出一些基本选择。这些决策（由我们的指南针指导）会影响使用哪个训练框架、如何分配我们的计算预算以及从哪个基线开始。

For SmolLM3, we went with a dense Llama-style architecture at 3B parameters because we were targeting small on-device models. But as you'll see in the Designing the model architecture chapter, a MoE or hybrid model might be better suited for your use case, and different model sizes come with different tradeoffs. We'll explore these choices in depth later, and show you how to make these decisions.

对于 SmolLM3，我们采用了 3B 参数的密集 Llama 式架构，因为我们的目标是小型设备上模型。但是，正如您将在设计模型架构一章中看到的那样，MoE 或混合模型可能更适合您的用例，并且不同的模型大小会带来不同的权衡。稍后我们将深入探讨这些选择，并向您展示如何做出这些决定。

For now, let's start with the most practical first step: choosing your baseline.

现在，让我们从最实用的第一步开始：选择基线。

Choosing your baseline 选择基线

Every successful model builds on a proven foundation and modifies it for their needs. When Qwen trained their first model family (Bai et al., 2023), they started from Llama's architecture. When Meta trained Llama 3, they started from Llama 2. Kimi K2, started from DeepSeek-V3's MoE architecture. This applies to architectures, but also training hyperparameters and optimisers.

每个成功的模型都建立在经过验证的基础上，并根据自己的需要对其进行修改。当 Qwen 训练他们的第一个模型家族时（Bai et al.， 2023），他们从 Llama 的架构开始。Meta 训练 Llama 3 时，他们是从 Llama 2 开始的。Kimi K2，从 DeepSeek-V3 的 MoE 架构开始。这适用于架构，也适用于训练超参数和优化器。

Why? Good architectures and training setups design take years of iteration across many organisations. The standard transformer and optimisers like Adam have been refined through thousands of experiments.

为什么？良好的架构和培训设置设计需要在许多组织中进行多年的迭代。像 Adam 这样的标准转换器和优化器已经通过数千次实验得到了改进。

People have found their failure modes, debugged instabilities, optimised implementations. Starting from a proven foundation means inheriting all that accumulated knowledge. Starting fresh means rediscovering every problem yourself.

人们已经找到了他们的故障模式，调试了不稳定性，优化了实施。从经过验证的基础开始意味着继承所有积累的知识。重新开始意味着自己重新发现每一个问题。

Here's what makes a good starting point for an architecture:

以下是架构的良好起点：

- Matches your constraints: aligns with your deployment target and use case.

  符合您的约束：与您的部署目标和用例保持一致。

- Proven at scale: multi-trillion token runs at similar or larger sizes.

  大规模验证：数万亿个代币以相似或更大的规模运行。

- Well-documented: known hyperparameters which have been proven to work in open models.

  有据可查：已知的超参数已被证明可以在开放模型中工作。

- Framework support: It should ideally be supported in the training frameworks you are considering and the inference frameworks you are planning to use.

  框架支持：理想情况下，您正在考虑的训练框架和您计划使用的推理框架应支持它。

Below is a non-exhaustive list of strong 2025 baseline options for various architectures and model sizes:

以下是针对各种架构和模型尺寸的强大 2025 年基线选项的非详尽列表：

Architecture Type 架构类型	Model Family 型号系列	
Dense 稠	Llama 3.1 骆驼 3.1	8B, 70B 8B、70B
Dense 稠	Llama 3.2 骆驼 3.2	1B, 3B 1B、3B
Dense 稠	Qwen3	0.6B, 1.7B, 4B, 14 0.6B、1.7B、4B、
Dense 稠	Gemma3 杰玛 3	12B, 27B 12B、2
Dense 稠	SmolLM2, SmolLM3 SmolLM2、SmolLM3	135M, 360M, 1.7
MoE 教育部	Qwen3 MoE Qwen3 教育部	30B-A3B, 235B-A
MoE 教育部	GPT-OSS	21B-A3B, 117B-A
MoE 教育部	Kimi Moonlight Kimi 月光	16B-A3B
MoE 教育部	Kimi-k2 基米-k2	1T-A32B
MoE 教育部	DeepSeek V3 深度搜索 V3	671B-A37B
Hybrid 混合	Zamba2 赞巴 2	1.2B, 2.7B, 7B 1.
Hybrid 混合	Falcon-H1 猎鹰-H1	0.5B, 1.5B, 3B, 7B 0.5B、1.5B、3B、
MoE + Hybrid MoE + 混合	Qwen3-Next Qwen3-下一个	80B-A3B
MoE + Hybrid MoE + 混合	MiniMax-01 迷你机-01	456B-A46B

So go to your architecture type and pick a baseline close to the number of parameters you'd like your model to have. Don't overthink it too much as the architecture you start from is not set in stone.

因此，转到您的架构类型，然后选择一个接近您希望模型具有的参数数量的基线。不要想太多，因为您开始的架构并不是一成不变的。

In the next section, we will see how to go from a baseline to a final architecture that is optimal for you.

在下一节中，我们将了解如何从基线到最适合您的最终架构。

MODIFYING YOUR BASELINE: THE DISCIPLINE OF DERISKING
修改你的基线：去风险的纪律

Now you have a baseline that works and fits your use case. You could stop here, train it on your data mixture (assuming it's good) and likely get a decent model. Many successful projects do exactly that.

现在，您有一个有效且适合您的用例的基线。你可以在这里停下来，在你的数据混合上训练它（假设它很好），并可能得到一个不错的模型。许多成功的项目正是这样做的。

But baselines aren't optimised for your specific constraints, they're designed for the use cases and deployment targets of whoever built them. So there are likely modifications worth making to better align with your goals.

但是，基线并不是针对您的特定约束进行优化的，它们是针对构建它们的人的用例和部署目标而设计的。因此，可能值得进行修改以更好地符合您的目标。

However, every architectural change carries risk: it might boost performance, tank it, or do nothing while wasting your ablation compute.

然而，每一次架构更改都会带来风险：它可能会提高性能、降低性能，或者在浪费消融计算的同时什么都不做。

The discipline that keeps you on track is **derisking** : never change anything unless you've tested that it helps.

让你走上正轨的纪律是去风险：永远不要改变任何东西，除非你已经测试过它有帮助。

> 📍 **What counts as derisked?**
>
> **什么算是去风险的?**
>
> A change is derisked when testing shows it either improves performance on your target capabilities, or provides a meaningful benefit (e.g. faster inference, lower memory, better stability) without hurting performance beyond your acceptable tradeoffs.
>
> 当测试表明更改可以提高目标功能的性能，或提供有意义的好处（例如，更快的推理、更低的内存、更好的稳定性）而不会损害超出您可接受的权衡的性能时，就会降低风险。

The tricky part is that your baseline and training setup have many components you could modify: attention mechanisms, positional encodings, activation functions, optimisers, training hyperparameters, normalisation schemes, model layout, and more.

棘手的部分是，您的基线和训练设置有许多可以修改的组件：注意力机制、位置编码、激活函数、优化器、训练超参数、归一化方案、模型布局等等。

Each represents a potential experiment, and these components often interact in non-linear ways.

每个都代表一个潜在的实验，并且这些组件通常以非线性方式相互作用。

You have neither the time nor compute to test everything or explore every interaction.

您既没有时间也没有计算来测试所有内容或探索每一次交互。

Start by testing promising changes against your current baseline. When something works, integrate it to create a new baseline, then test the next change against that. If your compute budget allows it you could test changes individually and run a leave-one-out analysis.

首先根据当前基线测试有希望的更改。当某些东西有效时，将其集成以创建新的基线，然后根据该基线测试下一个更改。如果计算预算允许，可以单独测试更改并运行留一分析。

Check out the ScaleRL paper (Khatri et al., 2025) for an example of this methodology in practice.

查看 ScaleRL 论文（Khatri 等人，2025 年），了解这种方法在实践中的示例。

Don't fall into the trap of exhaustive grid searches over every hyperparameter or testing every architectural variant that comes out.

不要陷入对每个超参数进行详尽网格搜索或测试每个出现的架构变体的陷阱。

> 🎯 **Strategic experimentation**
>
> **战略实验**
>
> Knowing how to run experiments isn't enough if you don't know which experiments are worth running. Ask yourself two questions before testing any modification:
>
> 如果您不知道哪些实验值得运行，那么仅仅知道如何运行实验是不够的。在测试任何修改之前问自己两个问题：
>
> - Will this help my specific use case?
>
>   这对我的特定用例有帮助吗?
>
> - Will this optimise my training?
>
>   这会优化我的训练吗?
>
> If a modification doesn't clearly address either question, skip it.
>
> 如果修改没有明确解决这两个问题，请跳过它。

Now that you know how to identify what's promising through strategic planning, it's time to move to the **empirical validation.** In the next sections, we'll show you *how* to actually test these changes in practice. We'll cover how to set up reliable experiments, interpret results, and avoid common pitfalls. Then in the following chapters, we'll walk through concrete examples of testing popular architectural, data, infra and training decisions.

现在您已经知道如何通过战略规划确定有希望的东西，是时候转向实证验证了。在接下来的部分中，我们将向您展示如何在实践中实际测试这些更改。我们将介绍如何设置可靠的实验、解释结果并避免常见陷阱。然后在接下来的章节中，我们将介绍测试流行的架构、数据、基础设施和训练决策的具体示例。

So let's build a simple ablation setup we can use for our experiments. First, we need to decide which training framework to pick.

因此，让我们构建一个可用于实验的简单消融设置。首先，我们需要决定选择哪个培训框架。

Picking a training framework
选择培训框架

The first decision we need to make is which framework to use for training our model, and by extension, for running all our ablations. This choice involves balancing three key considerations:

我们需要做出的第一个决定是使用哪个框架来训练我们的模型，进而用于运行我们所有的消融。这种选择涉及平衡三个关键考虑因素：

Don't be a hero and switch the training framework between ablations and your final run. That is the road to suffering.

不要成为英雄，在消融和最后一次跑步之间切换训练框架。这就是通往苦难的道路。

1. The framework must support our target architecture or let us easily extend it.

   框架必须支持我们的目标架构，或者让我们轻松扩展它。

2. It needs to be stable and production-ready, and not prone to mysteriously breaking midway through training.

   它需要稳定并准备好生产，并且不容易在训练中途神秘地损坏。

3. It should deliver strong throughput so we can iterate quickly and make the most of our compute budget.

   它应该提供强大的吞吐量，以便我们可以快速迭代并充分利用我们的计算预算。

In practice, these requirements might pull against each other, creating trade-offs. Let's look at the available options.

在实践中，这些要求可能会相互拉扯，从而产生权衡。让我们看看可用的选项。

Framework 框架	Features 特征	Battle-tested 久经考验	
Megatron-LM 威震天-登月舱	✅ Extensive ✅ 广泛	✅ Kimi-K2, Nemotron ✅ Kimi-K2，尼莫特	✅ Pione ✅ 3D 并
DeepSpeed 深速	✅ Extensive ✅ 广泛	✅ BLOOM, GLM ✅ 布鲁姆，GLM	✅ Pione ✅ ZeRO
TorchTitan 火炬泰坦	⚡ Growing feature set ⚡ 不断增长的功能集	⚠️ Newer but tested by PyTorch team ⚠️ 较新但经过 PyTorch 团队测试	⚡ Optim ⚡ 针对密
Nanotron 纳米创	🎯 Minimal, tailored for HF pretraining 🎯 最小，专为 HF 预训练量身定制	✅ Yes (StarCoder, SmolLM) ✅ 是（StarCoder、SmolLM）	✅ Optim ✅ 优化

The table above summarises the key trade-offs between popular frameworks. Lines of code for the first three frameworks are from the TorchTitan technical report (Liang et al., 2025). Let's discuss each in more detail:

上表总结了流行框架之间的关键权衡。前三个框架的代码行来自 TorchTitan 技术报告（Liang et al.， 2025）。让我们更详细地讨论每一个：

Megatron-LM from Nvidia has been around for years and is battle-tested. It's what powers models like Kimi's K2 (Team et al., 2025), it delivers solid throughput and has

most of the production features we'd want. But that maturity comes with complexity: the codebase can be hard to navigate and modify when you're new to it.

Nvidia 的 Megatron-LM 已经存在多年，并经过实战考验。这就是 Kimi 的 K2（Team 等人，2025 年）等型号的动力，它提供了稳定的吞吐量，并具有我们想要的大部分生产功能。但这种成熟度伴随着复杂性：当您刚接触代码库时，它可能很难导航和修改。

DeepSpeed falls into a similar category. It's the pioneer of ZeRO optimisation and powered models like BLOOM and GLM. Like Megatron-LM, it's extensively battle-tested and optimised, but shares the same complexity challenges.

DeepSpeed 属于类似的类别。它是 ZeRO 优化和 BLOOM 和 GLM 等动力模型的先驱。与威震天-LM 一样，它经过了广泛的实战测试和优化，但也面临着相同的复杂性挑战。

The large codebase (194k total lines) can be intimidating when you're getting started, particularly for implementing custom features or debugging unexpected behavior.

当您开始时，大型代码库（总共 194k 行）可能会令人生畏，尤其是在实现自定义功能或调试意外行为时。

On the other side, PyTorch's recent TorchTitan library is much lighter and simpler to navigate, thanks to its compact and modular codebase. It has the core features needed for pretraining and is great for rapid experimentation.

另一方面，PyTorch 最近的 TorchTitan 库由于其紧凑的模块化代码库而更轻且更易于导航。它具有预训练所需的核心功能，非常适合快速实验。

However, being newer, it isn't as battle-tested and can still be a bit unstable as it's actively developed.

然而，由于较新，它没有经过实战考验，并且由于它正在积极开发，因此仍然可能有点不稳定。

We took a different path and built our own framework, nanotron, from scratch. This gave us full flexibility and a deep understanding of large-scale pretraining; insights that later evolved into the Ultra Scale Playbook. Since we open-sourced the library, we also got valuable feedback from the community, though for most cases we had to battle-test features ourselves first.

我们走了一条不同的道路，从头开始构建了自己的框架 nanotron。这给了我们充分的灵活性和对大规模预训练的深刻理解;后来演变成 Ultra Scale Playbook。由于我们开源了库，我们也从社区那里得到了宝贵的反馈，尽管在大多数情况下，我们必须先自己对功能进行战斗测试。

The framework now supports all the production features we need for training, but we're still building out areas like MoE support.

该框架现在支持我们训练所需的所有生产功能，但我们仍在构建 MoE 支持等领域。

Building from scratch made sense then, but it demands major investment in team expertise and time to debug issues and add missing features. A strong alternative is forking an existing framework and enhancing it for your needs.

从头开始构建当时是有意义的，但它需要对团队专业知识和时间进行大量投资来调试问题和添加缺失的功能。一个强有力的替代方案是分叉现有框架并根据您的需求对其进行增强。

For example, Thinking Machines Lab built their internal pretraining library as a fork of TorchTitan (source).

例如，Thinking Machines Lab 将他们的内部预训练库构建为 TorchTitan 的一个分支（来源）。

Ultimately, your choice depends on your team's expertise, target features, and how much time you're willing to invest in development versus using the most production-ready option.

最终，您的选择取决于您团队的专业知识、目标功能以及您愿意在开发上投入多少时间与使用最适合生产的选项。

If multiple frameworks support your needs, compare their throughput on your specific hardware. For quick experiments and speed runs, simpler codebases often win.

如果多个框架支持你的需求，请比较它们在特定硬件上的吞吐量。对于快速实验和快速运行，更简单的代码库通常会获胜。

Ablation setup 消融设置

With our framework chosen, we now need to design our ablation setup. We need experiments that are fast enough to iterate on quickly, but large enough that the results give us signal and transfer to the final model. Let's see how to set this up.

选择框架后，我们现在需要设计烧蚀设置。我们需要足够快的实验来快速迭代，但又足够大，以便结果为我们提供信号并转移到最终模型。让我们看看如何设置它。

SETTING UP OUR ABLATION FRAMEWORK
设置我们的消融框架

The goal of ablations is to run experiments at a small scale and get results we can confidently extrapolate to our final production run.

消融的目标是小规模进行实验，并获得我们可以自信地推断到最终生产运行的结果。

There are two main approaches. First, we can take our target model size and train it on fewer tokens. For the SmolLM3 ablations, we trained the full 3B model on 100B tokens instead of the final 11T.

主要有两种方法。首先，我们可以采用目标模型大小并在更少的代币上训练它。对于 SmolLM3 消融，我们在 100B 标记上训练了完整的 3B 模型，而不是最终的 11T。

Second, if our target model is too large, we can train a smaller proxy model for ablations.

其次，如果我们的目标模型太大，我们可以训练一个更小的代理模型进行消融。

For example, when Kimi was developing their 1T parameter Kimi K2 model with 32B active parameters, using the full size for all ablations would have been prohibitively expensive, so they ran some ablations on a 3B MoE with 0.

例如，当 Kimi 开发具有 32B 活动参数的 1T 参数 Kimi K2 模型时，对所有消融使用全尺寸的成本会高得令人望而却步，因此他们在具有 0 的 3B MoE 上运行了一些消融。

5B active parameters(Team et al., 2025).

5B 主动参数（Team et al.， 2025）。

One key question is whether these small-scale findings actually transfer. In our experience, if something hurts performance at small scale, you can confidently rule it out for large scale.

一个关键问题是这些小规模的发现是否真的转移了。根据我们的经验，如果某些东西在小规模上损害了性能，您可以自信地将其排除在大规模之外。

But if something works at small scale, you should still make sure you've trained on a reasonable number of tokens to conclude with high probability that these findings will extrapolate to larger scales.

但是，如果某些东西在小规模上有效，您仍然应该确保您已经训练了合理数量的标记，以便得出这些发现很有可能推断到更大规模的结论。

The longer you train and the closer the ablation models are to the final model, the better.

训练时间越长，消融模型越接近最终模型越好。

In this blog post, we'll use a baseline vanilla transformer for all ablations. Our main setup is a 1B transformer following Llama3.2 1B architecture trained on 45B tokens. This takes about 1.5 days to train on a node of 8xH100s using this nanotron config (42k tokens per second per GPU). During SmolLM3 training, we ran these ablations on a 3B model trained on 100B tokens (config here.). We'll share those results at the end of each chapter (you'll see that the conclusions align).

We train for 45B tokens to ensure we get stable signal, though ~35B is Chinchilla-optimal for this model size.

我们针对 45B 标记进行训练，以确保我们获得稳定的信号，尽管 ~35B 对于这个模型大小来说是龙猫的最佳选择。

在这篇博文中，我们将使用基线香草转换器进行所有消融。我们的主要设置是一个遵循 Llama3.2 1B 架构的 1B 转换器，在 45B 代币上训练。使用此 nanotron 配置（每个 GPU 每秒 1.5k 个令牌）在 8xH100s 的节点上训练大约需要 42 天。在 SmolLM3 训练期间，我们在 3B 标记上训练的 100B 模型上运行了这些消融（配置在这里）。我们将在每章末尾分享这些结果（您会看到结论一致）。

Our baseline 1B config captures all the essential training details in a structured YAML format. Here are the key sections:

我们的基线 1B 配置以结构化 YAML 格式捕获所有基本训练细节。以下是关键部分：

```yaml
## Datasets and mixing weights
data_stages:
- data:
Â
    dataset:
      dataset_folder:
      - fineweb-edu
      - stack-edu-python
      - finemath-3plus
Â
      dataset_weights:
      - 0.7
      - 0.2
      - 0.1
Â
## Model architecture, Llama3.2 1B configuration
```

```yaml
model:
  model_config:
    hidden_size: 2048
    num_hidden_layers: 16
    num_attention_heads: 32
    num_key_value_heads: 8
    intermediate_size: 8192
    max_position_embeddings: 4096
    rope_theta: 50000.0
    tie_word_embeddings: true
Â
## Training hyperparameters, AdamW with cosine schedule
optimizer:
  clip_grad: 1.0
  learning_rate_scheduler:
    learning_rate: 0.0005
    lr_decay_starting_step: 2000
    lr_decay_steps: 18000
    lr_decay_style: cosine
    lr_warmup_steps: 2000
    lr_warmup_style: linear
    min_decay_lr: 5.0e-05
  optimizer_factory:
    adam_beta1: 0.9
    adam_beta2: 0.95
    adam_eps: 1.0e-08
    name: adamW
Â
## Parallelism, 1 node
parallelism:
  dp: 8  # Data parallel across 8 GPUs
  tp: 1  # No tensor or pipeline parallelism needed at 1B scale
  pp: 1
Â
## Tokenizer
tokenizer:
  tokenizer_max_length: 4096
  tokenizer_name_or_path: HuggingFaceTB/SmolLM3-3B
Â
## Batch size, sequence length and total training for 30B tokens
tokens:
  batch_accumulation_per_replica: 16
  micro_batch_size: 3 # GBS (global batch size)=dp * batch_acc* MBS * sequence=1.
  sequence_length: 4096
  train_steps: 20000 # GBS * 20000 = 30B

 ...(truncated)
```

For our ablations, we'll modify different sections depending on what we're testing while keeping everything else constant: the `model` section for architecture choices, the `optimizer` section for optimizer and training hyperparameters, and the `data_stages` section for data curation.

对于我们的消融，我们将根据我们正在测试的内容修改不同的部分，同时保持其他所有内容不变：架构选择 `model` 部分、 `optimizer` 优化器和训练超参数部分以及数据管理 `data_stages` 部分。

> ☝ **Modify one thing at a time**
>
> **一次修改一件事**
>
> Change only one variable per ablation while keeping everything else constant. If you change multiple things and performance improves, you won't know what caused it. Test modifications individually, then combine successful ones and reassess.
>
> 每次消融仅更改一个变量，同时保持其他所有变量不变。如果您更改了多项内容并且性能有所提高，您将不知道是什么原因造成的。单独测试修改，然后组合成功的修改并重新评估。

When running ablations, some architectural changes can significantly alter parameter count. For instance, switching from tied to untied embeddings doubles our embedding parameters, while going from MHA to GQA or MQA decreases our attention parameters substantially.

运行烧蚀时，某些架构更改可能会显着改变参数计数。例如，从绑定嵌入切换到不绑定嵌入会使我们的嵌入参数翻倍，而从 MHA 切换到 GQA 或 MQA 会大大降低我们的注意力参数。

To ensure fair comparisons, we need to track parameter counts and occasionally adjust other hyperparameters (like hidden size or layer count) to keep model sizes roughly the same.

为了确保公平的比较，我们需要跟踪参数计数，并偶尔调整其他超参数（如隐藏大小或层数），以保持模型大小大致相同。

Here is a simple function that we use to estimate parameter counts for different

configurations:

这是一个简单的函数，我们用它来估计不同配置的参数计数：

```python
from transformers import LlamaConfig, LlamaForCausalLM
Â
def count_parameters(
    tie_embeddings=True,
    num_key_value_heads=4,
    num_attention_heads=32,
    hidden_size=2048,
    num_hidden_layers=16,
    intermediate_size=8192,
    vocab_size=128256,
    sequence_length=4096,
):
    config = LlamaConfig(
        hidden_size=hidden_size,
        num_hidden_layers=num_hidden_layers,
        num_attention_heads=num_attention_heads,
        num_key_value_heads=num_key_value_heads,
        intermediate_size=intermediate_size,
        vocab_size=vocab_size,
        max_position_embeddings=sequence_length,
        tie_word_embeddings=tie_embeddings,
    )
    model = LlamaForCausalLM(config)
    return f"{sum(p.numel() for p in model.parameters())/1e9:.2f}B"
```

We also provide an interactive tool to visualise LLM parameter distributions, in the case of a dense transformer. This can come in handy when making architecture decisions or setting up configs for ablations.

我们还提供了一个交互式工具来可视化 LLM 参数分布，在密集转换器的情况下。这在做出架构决策或设置烧蚀配置时会派上用场。

This calculator assumes standard architectural choices: gated feedforward networks, standard head dimensions for attention (hidden_size / num_heads), and 2 layer norms per transformer layer. It doesn't include bias terms (if used).

该计算器假设标准架构选择：门控前馈网络、注意力标准磁头尺寸 （hidden_size / num_heads） 以及每个变压器层的 2 层规范。它不包括偏见术语（如果使用）。

UNDERSTANDING WHAT WORKS: EVALUATION
了解什么是有效的：评估

Once we launch our ablations, how do we know what works or not?

一旦我们启动消融，我们如何知道什么有效或无效？

The first instinct of anyone who trains models might be to look at the loss, and yes, that's indeed important. You want to see it decreasing smoothly without wild spikes or instability.

任何训练模型的人的第一反应可能是查看损失，是的，这确实很重要。您希望看到它平稳下降，而不会出现剧烈的峰值或不稳定。

For many architectural choices, the loss correlates well with downstream performance and can be sufficient(Y. Chen et al., 2025). However, looking at the loss only is not always reliable. Taking the example of data ablations, you would find that training on Wikipedia gives a lower loss than training on web pages (the next token is easier to predict), but that doesn't mean you'd get a more capable model.

对于许多架构选择，损耗与下游性能密切相关，并且可能足够（Y. Chen et al.， 2025）。然而，只看损失并不总是可靠的。以数据消融为例，你会发现在维基百科上训练比在网页上训练损失更低（下一个标记更容易预测），但这并不意味着你会得到一个更强大的模型。

Similarly, if we change the tokenizer between runs, the losses aren't directly comparable since text gets split differently. Some changes might also specifically affect certain capabilities like reasoning and math and get washed away in the average loss.

同样，如果我们在运行之间更改分词器，则损失无法直接比较，因为文本的拆分方式不同。一些变化还可能特别影响推理和数学等某些功能，并在平均损失中被冲走。

Last but not least, models can continue improving on downstream tasks even after pretraining loss has converged(Liu et al., 2022).

最后但并非最不重要的一点是，即使在预训练损失收敛之后，模型也可以继续改进下游任务（Liu et al.， 2022）。

We need more fine-grained evaluation to see the full picture and understand these nuanced effects and a natural approach is to use downstream evaluations that test knowledge, understanding, reasoning, and whatever other domains matter for us.

我们需要更细粒度的评估来了解全貌并理解这些细微差别的影响，一种自然的方法是使用下游评估来测试知识、理解、推理以及对我们重要的任何其他领域。

For these ablations, it's good to focus on tasks that give good early signal and avoid noisy benchmarks. In FineTasks and FineWeb2, reliable evaluation tasks are defined by four key principles:

对于这些消融，最好专注于提供良好早期信号并避免嘈杂基准的任务。在 FineTasks 和 FineWeb2 中，可靠的评估任务由四个关键原则定义：

- **Monotonicity:** The benchmark scores should consistently improve as models train longer.

  单调性：随着模型训练时间的延长，基准分数应该会持续提高。

- **Low noise:** When we train models with the same setup but different random seeds, the benchmark scores shouldn't vary wildly.

  低噪声：当我们使用相同的设置但不同的随机种子训练模型时，基准分数应该不会有很大差异。

- **Above-random performance:** Many capabilities only emerge later in training, so tasks that show random-level performance for extended periods aren't useful for ablations. This is the case, for example, for MMLU in multiple choice format as we will explain later.

  高于随机性能：许多功能仅在训练后期出现，因此长时间显示随机级别性能的任务对消融没有用处。例如，对于多项选择格式的 MMLU，我们稍后将解释。

- **Ranking consistency:** If one approach outperforms another at early stages, this ordering should remain stable as training continues.

  排名一致性：如果一种方法在早期阶段优于另一种方法，则随着训练的继续，这种排序应该保持稳定。

The quality of a task also depends on the task formulation (how we ask the model questions) and metric choice (how we compute the answer score).

任务的质量还取决于任务的表述（我们如何向模型提问）和指标选择（我们如何计算答案分数）。

Three common task formulations are multiple choice format (MCF), cloze formulation (CF) and freeform generation (FG).

三种常见的任务公式是多项选择格式 （MCF）、完形填空公式 （CF） 和自由格式生成（FG）。

Multiple choice format requires models to select an option from a number of choices explicitly presented in the prompt and prefixed with A/B/C/D (as is done in MMLU, for example).

多项选择格式要求模型从提示中显式显示并以 A/B/C/D 为前缀的多个选项中选择一个选项（例如，在 MMLU 中所做的那样）。

In cloze formulation, we compare the likelihood of the different choices to see which one is more likely without having provided them in the prompt. In FG, we look at the accuracy of the greedy generation for a given prompt.

在完形填空公式中，我们比较不同选择的可能性，看看哪一个更有可能，而无需在提示中提供它们。在 FG 中，我们查看给定提示的贪婪生成的准确性。

FG requires a lot of latent knowledge in the model and is usually too difficult a task for the models to be really useful in short pre-training ablations before a full of training. We thus focus on multiple choice formulations when running small sized ablations (MCF or CF).

FG 需要模型中的大量潜在知识，并且对于模型来说通常太困难了，在完全训练之前的短训练前消融中，模型无法真正有用。因此，在运行小尺寸消融（MCF 或 CF）时，我们专注于多项选择配方。

> 📍 **Heads-up 抬头**
>
> For post-trained models, FG becomes the primary formulation since we're evaluating whether the model can actually generate useful responses. We'll cover evaluation for these models in the post-training chapter.
>
> 对于后训练模型，FG 成为主要公式，因为我们正在评估模型是否真的可以生成有用的响应。我们将在训练后章节中介绍这些模型的评估。

Research has also shown that models struggle with MCF early in training, only learning this skill after extensive training, making CF better for early signal (Du et al., 2025; Gu et al., 2025; J. Li et al., 2025). We thus use CF for small ablations, and integrate MCF in the main run as it gives better mid-training signal once a model has passed a threshold to get sufficiently high signal-over-noise ratio for MCF.

研究还表明，模型在训练早期与 MCF 作斗争，只有在广泛训练后才学会这项技能，这使得 CF 更适合早期信号（Du 等人，2025 年;Gu 等人，2025 年;J. Li 等人，2025 年）。因此，我们将 CF 用于小消融，并将 MCF 集成到主运行中，因为一旦模型通过阈值以获得足够高的 MCF 信噪比，它就会提供更好的训练中期信号。

A quick note also that, to score a model's answer in sequence likelihood evaluations like CF, we compute accuracy as the percentage of questions where the the correct answer has the highest log probability normalised by character count.

还需要快速说明的是，为了在 CF 等序列似然评估中对模型的答案进行评分，我们将准确性计算为正确答案具有最高对数概率的问题百分比，按字符数归一化。

This normalisation prevents a bias toward shorter answers.

这种标准化可以防止偏向于较短的答案。

Our ablations evaluation suite includes the benchmarks from FineWeb ablations, except for SIQA which we found to be too noisy. We add math and code benchmarks like GSM8K and HumanEval and the long context benchmark RULER for long context ablations.

我们的消融评估套件包括 FineWeb 消融的基准测试，但我们发现噪音太大的 SIQA 除外。我们添加了 GSM8K 和 HumanEval 等数学和代码基准测试，以及用于长上下文消融的长上下文基准测试 RULER。

This aggregation of tasks test world knowledge, reasoning, and common sense across a variety of formats, as shown in the table below.

这种任务聚合测试了各种格式的世界知识、推理和常识，如下表所示。

To speed up evaluations at the expense of some additional noise, we only evaluate on 1,000 questions from each benchmark (except for GSM8K, HumanEval & RULER, which we used in full for the 3B SmolLM3 ablations but omit from the 1B experiments below).

为了以牺牲一些额外的噪音为代价来加快评估速度，我们只评估每个基准测试中的 1,000 个问题（GSM8K、HumanEval 和 RULER 除外，我们将它们全部用于 3B SmolLM3 消融，但从下面的 1B 实验中省略）。

We also use the cloze fomulation (CF) way of evaluating for all multiple-choice

The point at which MMLU MCF becomes non-random depends on the model size and training data. For a 7B transformer, the OLMES paper (Gu et al., 2025) found the model starts showing non-random performance after 500B tokens. For 1.7B model, we found this happens after 6T tokens in SmolLM2 (Allal et al., 2025). Du et al. (2025) argue this is fundamentally about the pre-training loss reaching a certain threshold.

MMLU MCF 变为非随机的点取决于模型大小和训练数据。对于 7B 变压器，OLMES 论文（Gu et al.， 2025）发现模型在 500B 标记后开始显示非随机性能。对于 1.7B 模型，我们发现这种情况发生在 SmolLM2 中的 6T 标记之后（Allal 等人，2025 年）。Du 等人（2025 年）认为，这从根本上说是关于训练前损失达到一定阈值的问题。

benchmarks, as explained above. Note that for multilingual ablations and actual training, we add more benchmarks to test multilinguality, which we detail later.

如上所述，我们还使用完形填空 fomulation （CF） 方式来评估所有多项选择基准。请注意，对于多语言消融和实际训练，我们添加了更多基准来测试多语言，稍后我们将详细介绍。

These evaluations are run usingLightEval and the table below summarises the key characteristics of each benchmark:

这些评估是使用 LightEval 运行的，下表总结了每个基准测试的主要特征：

Benchmark 基准	Domain 域	Task Type 任务
MMLU MMLU 的	Knowledge 知识	Multiple choice 多项选
ARC 弧	Science & reasoning 科学与推理	Multiple choice 多项选
HellaSwag 海拉赃物	Commonsense reasoning 常识性推理	Multiple choice 多项选
WinoGrande 维诺格兰德	Commonsense reasoning 常识性推理	Binary choice 二元选择
CommonSenseQA 常识 QA	Commonsense reasoning 常识性推理	Multiple choice 多项选

OpenBookQA 开放图书 QA	Science 科学	Multiple choice 多项选
PIQA 皮卡	Physical commonsense 物理常识	Binary choice 二元选择
GSM8K GSM8K 的	Math 数学	Free-form generation
HumanEval 人类评估	Code 法典	Free-form generation

Let's look at a few example questions from each to get a concrete sense of what these evaluations actually test:

让我们看看每个问题中的几个示例问题，以具体了解这些评估实际测试的内容：

Browse through the examples above to see the types of questions in each benchmark. Notice how MMLU and ARC test factual knowledge with multiple choices, GSM8K requires computing numerical answers to math problems, and HumanEval requires generating complete Python code.

浏览上面的示例，查看每个基准中的问题类型。请注意 MMLU 和 ARC 如何使用多项选择来测试事实知识，GSM8K 需要计算数学问题的数值答案，而 HumanEval 需要生成完整的 Python 代码。

This diversity ensures we're testing different aspects of model capability throughout our ablations.

这种多样性确保了我们在整个消融过程中测试模型能力的不同方面。

Which data mixture for the ablations?

消融采用哪种数据混合?

For *architecture ablations* , we train on a fixed mix of high-quality datasets that provide early signal across a wide range of tasks. We use English (FineWeb-Edu), math (FineMath), and code (Stack-Edu-Python). Architectural findings should extrapolate well to other datasets and domains, including multilingual data so we can keep our data mixture simple.

对于架构消融，我们在固定的高质量数据集组合上进行训练，这些数据集为各种任务提供早期信号。我们使用英语 （FineWeb-Edu）、数学 （FineMath） 和代码 （Stack-Edu-Python）。架构发现应该很好地推断到其他数据集和领域，包括多语言数据，这样我们就可以保持数据混合简单。

For *data ablations* , we take the opposite approach: we fix the architecture and systematically vary the data mixtures to understand how different data sources affect model performance.

对于数据消融，我们采取相反的方法：我们修复架构并系统地改变数据混合，以了解不同的数据源如何影响模型性能。

The real value of a solid ablation setup goes beyond just building a good model.

可靠的消融设置的真正价值不仅仅是构建一个好的模型。

When things inevitably go wrong during our main training run (and they will, no matter how much we prepare), we want to be confident in every decision we made

Sometimes the differences in the evaluations can be small. If you have enough compute, it might be worth re-running the same ablations with different seeds to see how much the results vary.

有时评估的差异可能很小。如果您有足够的计算能力，则可能值得使用不同的种子重新运行相同的消融，以查看结果的

and quickly identify which components weren't properly tested and could be causing the issues.

当在我们的主要训练运行中不可避免地出现问题时（无论我们准备多少，它们都会出错），我们希望对我们所做的每一个决定充满信心，并快速确定哪些组件没有经过适当的测试并可能导致问题。

This preparation saves debugging time and bullet proof our future mental sanity.

这种准备工作节省了调试时间，并防弹了我们未来的精神理智。

ESTIMATING ABLATIONS COST
估算消融成本

Ablations are amazing but they require GPU time and it's worth understanding the cost of these experiments.

消融是惊人的，但它们需要 GPU 时间，并且值得了解这些实验的成本。

The table below shows our complete compute breakdown for SmolLM3 pretraining: the main run (accounting for occasional downtimes), ablations before and during training, plus compute spent on an unexpected scaling issue that forced a restart and some debugging (which we'll detail later).

下表显示了 SmolLM3 预训练的完整计算细分：主运行（考虑偶尔的停机时间）、训练前和训练期间的消融，以及用于强制重启和一些调试的意外扩展问题的计算（我们稍后会详细介绍）。

Phase 阶段	GPUs GPU	Days 日	GPU-hours GPU 小时
Main pretraining run 主要预训练运行	384	30	276,480
Ablations (pretraining) 消融（预训练）	192	15	69,120
Ablations (mid-training) 消融（训练中）	192	10	46,080
Training reset & debugging 训练重置和调试	384/192	3/4	46,080
Total cost 总成本	-	-	**437,760**

The numbers reveal an important fact: ablations and debugging consumed a total of 161,280 GPU hours, **more than half the cost of our main training run** (276,480 GPU hours) **.** We run over 100 ablations total across SmolLM3's development: we spent 20 days on pre-training ablations, 10 days on mid-training ablations, and 7 days recovering from an unexpected training issue that forced a restart and some debugging (which we'll detail later).

这些数字揭示了一个重要的事实：消融和调试总共消耗了 161,280 个 GPU 小时，超过我们主要训练运行成本（276,480 个 GPU 小时）的一半以上。在 SmolLM3 的整个开发过程中，我们总共运行了 100 多次消融：我们花了 20 天进行训练前消融，10 天用于训练中期消融，7 天从意外的训练问题中恢复，该问题迫使我们重新启动和进行一些调试（我们稍后会详细介绍）。

This highlights why ablation costs must be factored into your compute budget: plan for training cost plus ablations plus buffer for surprises.

这突出了为什么必须将消融成本纳入您的计算预算：计划训练成本、消融和意外缓冲区。

If you're targeting SOTA performance, implementing new architecture changes, or don't already have a proven recipe, ablations become a substantial cost center rather than minor experiments.

如果您的目标是 SOTA 性能、实施新的架构更改，或者还没有经过验证的配方，那么烧蚀将成为一个重要的成本中心，而不是小型实验。

Before we move to the next section, let's establish some ground rules that every person running experiments should follow.

在进入下一节之前，让我们先建立一些每个运行实验的人都应该遵循的基本规则。

Rules of engagement 交战规则

TL;DR: Be paranoid. TL;DR：偏执。

Validate your evaluation suite. Before training any models, make sure your evaluation suite can reproduce the published results of models you will compare against.

差异程度。

We estimate evaluation costs to be slightly under 10,000 GPU hours. Our full evaluation suite (english, multilingual, math & code) takes around 1.5 hours per GPU, and we evaluate every 10B tokens throughout the 11T tokens, in addition to numerous ablations.

我们估计评估成本略低于 10,000 GPU 小时。我们完整的评估套件（英语、多语言、数学和代码）每个 GPU 大约需要 1.5 小时，除了多次消融之外，我们还会评估整个 11T 令牌中的每 10B 个令牌。

The long context evaluations were particularly expensive, taking around 1 hour on 8 GPUs per run.

长上下文评估特别昂贵，每次运行 8 个 GPU 大约需要 1 小时。

When DeepSeek-V3 came out, the world fixated on its reported $5.6M training cost. Many interpreted that number as the full R&D cost. In reality, it only reflects the final training run.

当 DeepSeek-V3 问世时，全世界都关注其报告的 $5.6M 训练成本。许多人将这个数字解释为全部研发成本。实际上，它仅反映最终的训练运行。

The much larger — and usually invisible — expense is in the research itself: the ablations, failed runs, and debugging that lead to a final recipe. Given the scale and novelty of the model, their research costs were certainly higher.

更大的——而且通常是看不见的——费用在于研究本身：消融、失败的运行和调试，这些都会导致最终配方。鉴于该模

验证评估套件。在训练任何模型之前，请确保您的评估套件可以重现您将与之比较的模型的已发布结果。

If any benchmarks are generative in nature (e.g. GSM8k), be extra paranoid and manually inspect a few samples to ensure the prompt is formatted correctly and that any post-processing is extracting the correct information.

如果任何基准测试本质上是生成性的（例如 GSM8k），请格外偏执并手动检查一些样本，以确保提示格式正确，并且任何后处理都提取了正确的信息。

Since evals will guide every single decision, getting this step right is crucial for the success of the project!

由于评估将指导每一个决策，因此正确迈出这一步对于项目的成功至关重要！

Test every change, no matter how small. Don't underestimate the impact of that seemingly innocent library upgrade or the commit that "only changed two lines". These small changes can introduce subtle bugs or performance shifts that will contaminate your results.

测试每一个变化，无论多么小。不要低估那个看似无辜的库升级或"只改了两行"的提交的影响。这些小的变化可能会引入细微的错误或性能变化，从而污染您的结果。

You need a library with a strong test suite on the cases which matter to you to avoid regression.

您需要一个库，其中包含一个强大的测试套件，用于对您来说重要的案例，以避免回归。

Change one thing at a time. Keep everything else identical between experiments. Some changes can interact with each other in unexpected ways, so we first want to assess the individual contribution of each change, then try combining them to see their overall impact.

一次改变一件事。在实验之间保持其他所有内容相同。有些更改可能会以意想不到的方式相互交互，因此我们首先要评估每个更改的个体贡献，然后尝试将它们组合起来以查看其整体影响。

Train on enough tokens and use sufficient evaluations. As we mentioned earlier, we need to make sure we have good coverage in our evaluation suite and train long enough to get reliable signal. Cutting corners here will lead to noisy results and bad decisions.

训练足够的代币并使用足够的评估。正如我们之前提到的，我们需要确保我们的评估套件具有良好的覆盖范围，并训练足够长的时间以获得可靠的信号。在这里偷工减料会导致嘈杂的结果和错误的决定。

Following these rules might feel overly cautious, but the alternative is spending days debugging mysterious performance drops that turn out to be caused by an unrelated dependency update from days earlier. The golden principle: once you have a good setup, *no change should go untested!*

遵循这些规则可能会让人感觉过于谨慎，但另一种选择是花费数天时间调试神秘的性能下降，这些下降结果是由几天前不相关的依赖项更新引起的。黄金原则：一旦你有了良好的设置，任何变化都不应该未经测试！

Designing the model architecture
设计模型体系结构

Now that we have our experimental framework in place, it's time to make the big decisions that will define our model.

现在我们已经有了实验框架，是时候做出定义我们模型的重大决策了。

Every choice we make, from model size to attention mechanisms to tokenizer choice, creates constraints and opportunities that will affect model training and usage.

我们所做的每一个选择，从模型大小到注意力机制，再到分词器选择，都会产生影响模型训练和使用的约束和机会。

Remember the training compass: before making any technical choices, we need clarity on the *why* and *what* . Why are we training this model, and what should it look like?

记住训练指南针：在做出任何技术选择之前，我们需要清楚 为什么 和 什么 .我们为什么要训练这个模型，它应该是什么样子？

It sounds obvious, but as we explained in the training compass, being deliberate here shapes our decisions and keeps us from getting lost in the endless space of possible experiments. Are we aiming for a SOTA model in English? Is long context a priority?

型的规模和新颖性，他们的研究成本肯定更高。

In some cases, a bug can be solved by upgrading the library to the latest version. For a beautiful example of this with some detective debugging, see the blog post by Elana Simon.

在某些情况下，可以通过将库升级到最新版本来解决错误。有关一些侦探调试的漂亮示例，请参阅 Elana Simon 的博客文章。

这听起来很明显，但正如我们在训练指南针中所解释的那样，在这里深思熟虑可以塑造我们的决定，并使我们不会迷失在可能实验的无尽空间中。我们的目标是英语的 SOTA 模型吗？长上下文是优先事项吗？

Or a we trying to validate a new architecture? The training loop may look similar in all these cases, but the experiments we run and the trade-offs we accept will be different.

或者我们试图验证新架构？在所有这些情况下，训练循环可能看起来相似，但我们运行的实验和我们接受的权衡会有所不同。

Answering this question early helps us decide how to balance our time between data and architecture work, and how much to innovate in each before starting the run.

尽早回答这个问题有助于我们决定如何在数据和架构工作之间平衡我们的时间，以及在开始运行之前在每一项工作中进行多少创新。

So, let's lead by example and walk through the goals that guided SmolLM3's design. We wanted a strong model for on-device applications with competitive multilingual performance, solid math and coding capabilities, and robust long context handling.

因此，让我们以身作则，逐步了解指导 SmolLM3 设计的目标。我们想要一个强大的设备端应用程序模型，具有具有竞争力的多语言性能、可靠的数学和编码功能以及强大的长上下文处理能力。

As we mentioned earlier, this led us to a dense model with 3B parameters: large enough for strong capabilities but small enough to fit comfortably on phones.

正如我们之前提到的，这导致我们得到了一个具有 3B 参数的密集模型：足够大以具有强大的功能，但又足够小以舒适地安装在手机上。

We went with a dense transformer rather than MoE or Hybrid given the memory constraints of edge devices and our project timeline (roughly 3 months).

考虑到边缘设备的内存限制和我们的项目时间表（大约 3 个月），我们选择了密集变压器，而不是 MoE 或混合。

We had a working recipe from SmolLM2 for English at a smaller scale (1.7B parameters), but scaling up meant re-validating everything and tackling new challenges like multilinguality and extended context length. One clear example of how having defined goals shaped our approach.

SmolLM2 was our previous generation of small language models, with three variants at 135M, 360M, and 1.7B parameters designed for on-device deployment. They were English only with 8k context length.

SmolLM2 是我们上一代的小型语言模型，具有 135M、360M 和 1.7B 参数的三种变体，专为设备上部署而设计。它们只有英语，上下文长度为 8k。

我们有一个来自 SmolLM2 的英语工作配方，规模较小（1.7B 参数），但扩大规模意味着重新验证所有内容并应对新的挑战，例如多语言和扩展上下文长度。明确目标如何塑造我们的方法的一个明显例子。

For example, in SmolLM2, we struggled to extend the context length at the end of pretraining, so for SmolLM3 we made architectural choices from the start — like using NoPE and intra-document masking (see later) — to maximise our chances of getting it right, and it worked.

例如，在 SmolLM2 中，我们在预训练结束时很难延长上下文长度，因此对于 SmolLM3，我们从一开始就做出了架构选择——比如使用 NoPE 和文档内屏蔽（见后文）——以最大限度地提高我们正确完成它的机会，并且它奏效了。

Once our goals are clear, we can start making the technical decisions that will bring them to life. In this chapter, we'll go through our systematic approach to these core decisions: architecture, data, and hyperparameters.

一旦我们的目标明确，我们就可以开始做出技术决策，将其变为现实。在本章中，我们将介绍这些核心决策的系统方法：架构、数据和超参数。

Think of this as our strategic planning phase, getting these fundamentals right will save us from costly mistakes during the actual training marathon.

将此视为我们的战略规划阶段，正确掌握这些基本原理将使我们在实际的马拉松训练中避免代价高昂的错误。

Architecture choices 体系结构选择

If you look at recent models like Qwen3, Gemma3, or DeepSeek v3, you'll see that despite their differences, they all share the same foundation — the transformer architecture introduced in 2017 (Vaswani et al., 2023). What's changed over the years isn't the fundamental structure, but the refinements to its core components. Whether you're building a dense model, a Mixture of Experts, or a hybrid architecture, you're working with these same building blocks.

如果你看看最近的模型，如 Qwen3、Gemma3 或 DeepSeek v3，你会发现尽管它们存在差异，但它们都有相同的基础——2017 年推出的 transformer 架构（Vaswani 等人，2023 年）。这些年来的变化不是基本结构，而是对其核心组件的改进。无论您是构建密集模型、混合专家还是混合架构，您都在使用这些相同的构建块。

These refinements emerged from teams pushing for better performance and tackling

specific challenges: memory constraints during inference, training instability at scale, or the need to handle longer contexts.

这些改进源于团队推动更好的性能和应对特定挑战：推理过程中的内存限制、大规模训练不稳定性或处理更长上下文的需要。

Some modifications, like shifting from Multi-Head Attention (MHA) to more compute efficient attention variants like Grouped Query Attention (GQA)(Ainslie et al., 2023), are now widely adopted. Others, like different positional encoding schemes, are still being debated. Eventually, today's experiments will crystalize into tomorrow's baselines.

一些修改，例如从多头注意力 （MHA） 转向计算效率更高的注意力变体，如分组查询注意力 （GQA） （Ainslie 等人，2023 年），现在已被广泛采用。其他的，如不同的位置编码方案，仍在争论中。最终，今天的实验将具体化为明天的基线。

So what do modern LLMs actually use today? Let's look at what leading models have converged on. Unfortunately, not all models disclose their training details, but we have enough transparency from families like DeepSeek, OLMo, Kimi, and SmolLM to see the current landscape:

那么，现代法学硕士今天实际上使用什么呢？让我们看看领先模型已经趋同了什么。不幸的是，并非所有模型都披露了它们的训练细节，但我们从 DeepSeek、OLMo、Kimi 和 SmolLM 等家族那里获得了足够的透明度来查看当前的情况：

Model 型	Architecture 建筑	Parameters 参数	Training Tokens 训
DeepSeek LLM 7B DeepSeek 法学硕士 7B	Dense 稠	7B 7 乙	2T 2 吨
DeepSeek LLM 67B DeepSeek 法学硕士 67B	Dense 稠	67B 67 乙	2T 2 吨
DeepSeek V2 深度搜索 V2	MoE 教育部	236B (21B active) 236B （21B 活跃）	8.1T 8.1 吨
DeepSeek V3 深度搜索 V3	MoE 教育部	671B (37B active) 671B （37B 活跃）	14.8T 14.8 吨
MiniMax-01 迷你机-01	MoE + Hybrid MoE + 混合	456B (45.9 active) 456B （45.9 活跃）	11.4T 11.4 吨
Kimi K2 基米 K2	MoE 教育部	1T (32B active) 1T （32B 活动）	15.5T 15.5 吨
OLMo 2 7B	Dense 稠	7B 7 乙	5T 5 吨
SmolLM3 斯莫尔 LM3	Dense 稠	3B 3 乙	11T 11 吨

If you don't understand some of these terms yet, such as MLA or NoPE or WSD, don't worry. We'll explain each one in this section.

如果您还不了解其中一些术语，例如 MLA 或 NoPE 或 WSD，请不要担心。我们将在本节中逐一解释。

For now, just notice the variety: different attention mechanisms (MHA, GQA, MLA), position encodings (RoPE, NoPE, partial RoPE), and learning rate schedules (Cosine, Multi-Step, WSD).

现在，请注意多样性：不同的注意力机制（MHA、GQA、MLA）、位置编码（RoPE、NoPE、部分 RoPE）和学习率计划（余弦、多步、WSD）。

Looking at this long list of architecture choices it's a bit overwhelming to figure out where to even start. As in most such situations, we'll take it step by step and gradually build up all the necessary know-how.

查看这一长串架构选择，甚至不知道从哪里开始有点不知所措。与大多数此类情况一样，我们将逐步进行，并逐步积累所有必要的专业知识。

We'll focus on the simplest base architecture first (a dense model) and investigate each architectural aspect in detail. Later, we'll dive deep into MoE and Hybrid models and discuss when using them is a good choice.

我们将首先关注最简单的基础架构（密集模型），并详细研究每个架构方面。稍后，我们将深入研究 MoE 和混合模型，并讨论何时使用它们是一个不错的选择。

Finally we explore the tokenizer, an often overlooked and underrated component. Should we use an existing one or train our own? How do we even evaluate if our tokenizer is good?

最后，我们探讨分词器，这是一个经常被忽视和低估的组件。我们应该使用现有的还是训练我们自己的？我们如何评估我们的分词器是否好？

📍 **Ablation setup 消融设置**

Throughout the rest of this chapter, we validate most of the architectural choices through ablations using the setup described in the chapter above: our 1B baseline model (following the Llama3.

在本章的其余部分，我们使用上一章中描述的设置通过消融来验证大多数架构选择：我们的 1B 基线模型（遵循 Llama3.

2 1B architecture) trained on 45B tokens from a mix of FineWeb-Edu, FineMath, and Python-Edu. For each experiment, we show both training loss curves and downstream evaluation scores to assess the impact of each modification.

2 1B 架构）在来自 FineWeb-Edu、FineMath 和 Python-Edu 混合的 45B 令牌上进行训练。对于每个实验，我们展示了训练损失曲线和下游评估分数，以评估每次修改的影响。

You can find the configs for all the runs inHuggingFaceTB/training-guide-nanotron-configs.

您可以在 HuggingFaceTB/training-guide-nanotron-configs 中找到所有运行的配置。

This blog post by Sebastian Raschka gives a good overview of modern LLM architectures in 2025.

Sebastian Raschka 的这篇博文很好地概述了 2025 年的现代 LLM 架构。

But now let's start with the core of every LLM: the attention mechanism.

但现在让我们从每个 LLM 的核心开始：注意力机制。

ATTENTION 注意力

One of the most active areas of research around transformer architectures is the attention mechanism.

围绕变压器架构最活跃的研究领域之一是注意力机制。

While feedforward layers dominate compute during pretraining, attention becomes the main bottleneck at inference (especially with long contexts), where it drives up compute cost and the KV cache quickly consumes GPU memory, reducing throughput.

虽然前馈层在预训练期间主导计算，但注意力成为推理的主要瓶颈（尤其是在长上下文中），它会推高计算成本，并且 KV 缓存会快速消耗 GPU 内存，从而降低吞吐量。

Let's take a quick tour around the main attention mechanisms and how they trade-off capacity and speed.

让我们快速浏览一下主要的注意力机制以及它们如何权衡容量和速度。

How many heads for my attention?

有多少个头值得我注意?

Multi-head attention (MHA) is the standard attention introduced with the original transformer (Vaswani et al., 2023).

多头注意力 （MHA） 是原始变压器引入的标准注意力（Vaswani 等人，2023 年）。

Checkout Jay Alamar's famous blog post for a quick refresher!

查看 Jay Alamar 的著名博客文章以快速复习！

The main idea is that you have N attention heads each independently doing the same retrieval task: transform the hidden state into queries, keys, and values, then use the current query to retrieve the most relevant token by match on the keys and finally forward the value associated with the matched tokens.

主要思想是，你有 N 个注意力头，每个注意力头独立执行相同的检索任务：将隐藏状态转换为查询、键和值，然后使用当前查询通过匹配键来检索最相关的令牌，最后转发与匹配令牌关联的值。

At inference time we don't need to recompute the KV values for past tokens and can reuse them.

在推理时，我们不需要重新计算过去令牌的 KV 值，可以重用它们。

The memory for past KV values is called the*KV-Cache* . As context windows grow, this cache can quickly become an inference bottleneck and consume a large share of GPU memory. Here's a simple calculation to estimate the KV-Cache memory s_{KV} for the Llama 3 architecture with MHA and a sequence length of 8192:

过去 KV 值的内存称为 KV-Cache 。随着上下文窗口的增长，此缓存可能很快成为推理瓶颈，并消耗大量 GPU 内存。这是一个简单的计算，用于估计具有 MHA 和序列长度为 8192 的 Llama 3 架构的 KV-Cache 内存 s_{KV} ：

$$
\begin{aligned}
s_{KV} &= 2 \times n_{bytes} \times seq \times n_{layers} \times n_{heads} \times dim_{heads} \\
&= 2 \times 2 \times 8192 \times 32 \times 32 \times 128 = 4 \text{ GB } \textit{(Llama 3 8B)} \\
&= 2 \times 2 \times 8192 \times 80 \times 64 \times 128 = 20 \text{ GB } \textit{(Llama 3 70B)}
\end{aligned} \tag{1}
$$

Note that the leading factor of 2 comes from storing both key and value caches. As you can see, the cache increases linearly with sequence length, but context windows have grown exponentially, now reaching millions of tokens.

请注意，前导因素 2 来自存储键缓存和值缓存。如您所见，缓存随序列长度线性增加，但上下

文窗口呈指数级增长，现在达到数百万个令牌。

So improving the efficiency of the cache would make scaling context at inference time much easier.

因此，提高缓存的效率将使推理时扩展上下文变得更加容易。

The natural question to ask is: do we really need new KV values for each head? Probably not and both Multi-Query Attention (MQA) (Shazeer, 2019) and Grouped Query Attention (GQA) (Ainslie et al., 2023) address this. The simplest case is to share the KV values across all heads, thus dividing the size of the KV cache by n_{heads} , which is e.g. a 64 decrease for Llama 3 70B! This is the idea of MQA and was used in some models like StarCoder as an alternative to MHA.

自然而然的问题是：我们真的需要每个磁头的新 KV 值吗？可能不是，多查询注意力（MQA）（Shazeer，2019 年）和分组查询注意力（GQA）（Ainslie 等人，2023 年）都解决了这个问题。最简单的情况是在所有头上共享 KV 值，从而将 KV 缓存的大小除以 n_{heads} ，例如 Llama 3 70B 减少 64！这是 MQA 的想法，并被用于某些模型，例如 StarCoder，作为 MHA 的替代品。

However, we might give away a bit more attention capacity than we are willing to, so we could consider the middle ground and share the KV values across groups of heads e.g. 4 heads sharing the same KV values.

但是，我们可能会放弃比我们愿意的更多的注意力容量，因此我们可以考虑中间立场并在头组之间共享 KV 值，例如 4 个头共享相同的 KV 值。

This is the GQA approach and strikes a middle ground between MQA and MHA.

这是 GQA 方法，介于 MQA 和 MHA 之间。

More recently, DeepSeek-v2 (and also used in v3) introduced *Multi-Latent Attention (MLA)* (DeepSeek-AI et al., 2024), which uses a different strategy to compress the cache: rather than reducing the number KV-values it reduces their size and simply stores a latent variable which can be decompressed into KV values at runtime.

RoPE (Rotary Position Embeddings) is a method for encoding positional information by rotating query and key vectors based on their positions in the sequence. It's commonly used in today's LLMs.

RoPE（Rotary Position Embeddings）是一种通过根据查询向量和关键向量在序列中的位置旋转查询和关键向量来编码位置信息的方法。它通常用于当今的 LLM。

最近，DeepSeek-v2（也用于 v3）引入了多潜在注意力（MLA）（DeepSeek-AI 等人，2024 年），它使用不同的策略来压缩缓存：它不是减少 KV 值的数量，而是减小了它们的大小，并简单地存储了一个潜在变量，可以在运行时解压缩为 KV 值。

With this approach they managed to reduce the cache to an equivalent of GQA with 2.25 groups while giving stronger performance than MHA! In order to make this work with RoPE, a small tweak with an extra small latent vector is needed.

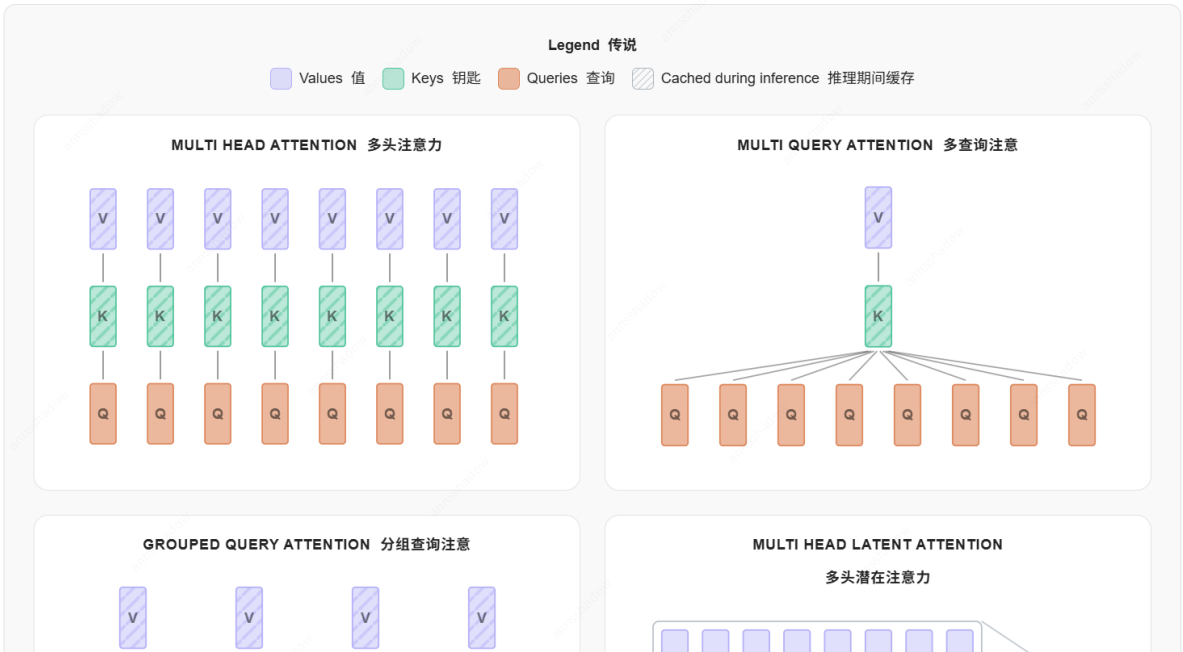
通过这种方法，他们设法将缓存减少到相当于 GQA 的 2.25 组，同时提供比 MHA 更强的性能！为了使这与 RoPE 一起工作，需要对超小的潜在向量进行小调整。

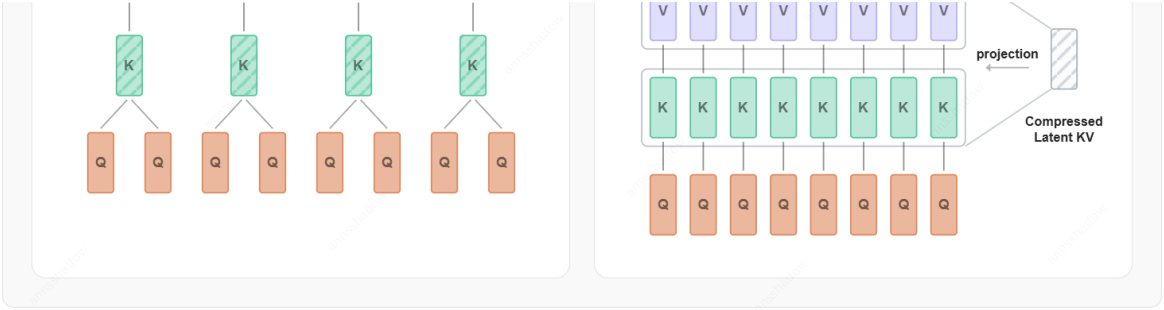
In DeepSeek-v2 they chose$4 * dim_{head}$ for the main latent variable and $1/2 * dim_{head}$ for the RoPE part so a total fo $4.5 * dim_{head}$ which is used for both K and V simultaneously thus dropping the leading factor of 2.

在 DeepSeek-v2 中，他们选择 $4 * dim_{head}$ 了主潜在变量和 $1/2 * dim_{head}$ RoPE 部分，因此同时用于 K 和 V $4.5 * dim_{head}$ 的总 fo，从而降低了 2 的前导因子。

You can see a visual explanation of each attention mechanism in the following graphic:

您可以在下图中看到每种注意力机制的可视化解释：





Simplified illustration of Multi-Head Attention (MHA), Grouped-Query Attention (GQA), Multi-Query Attention (MQA), and Multi-head Latent Attention (MLA). Through jointly compressing the keys and values into a latent vector, MLA significantly reduces the KV cache during inference.

多头注意力 （MHA）、分组查询注意力 （GQA）、多查询注意力 （MQA） 和多头潜在注意力 （MLA） 的简化图示。通过将键和值联合压缩为潜在向量，MLA 显著减少了推理过程中的 KV 缓存。

The following table compares the attention mechanisms we just discussed in this section. For simplicity we compare the parameters used per token, if you want to compute total memory simply multiply by bytes per parameter (typically 2) and sequence length:

下表比较了我们刚刚在本节中讨论的注意力机制。为简单起见，我们比较每个令牌使用的参数，如果您想计算总内存，只需乘以每个参数的字节数（通常为 2）和序列长度：

Attention Mechanism 注意力机制	KV-Cache parameters per token 每个令牌的 KV-Cache 参数
MHA MHA 的	$= 2 \times n_{heads} \times n_{layers} \times dim_{head}$
MQA MQA 的	$= 2 \times 1 \times n_{layers} \times dim_{head}$
GQA GQA 认证	$= 2 \times g \times n_{layers} \times dim_{head}$ (typically g=2,4,8)
MLA	$= 4.5 \times n_{layers} \times dim_{head}$

Now let's see how these attention mechanisms fare in real experiments!

现在让我们看看这些注意力机制在实际实验中的表现如何！

Ablation - GQA beats MHA

消融 - GQA 击败 MHA

Here we compare different attention mechanisms. Our baseline model uses 32 heads and 8 KV heads which corresponds to GQA with ratio 32/8=4. How would performance change if we used MHA, or if we went for even less KV heads and a higher GQA ratio?

在这里，我们比较不同的注意力机制。我们的基线模型使用 32 个头和 8 KV 头，对应于 GQA，比率为 32/8=4。如果我们使用 MHA，或者如果我们选择更少的 KV 磁头和更高的 GQA 比率，性能会发生怎样的变化？

Some libraries call the GQA ratio: Query groups = Query heads / KV heads

有些库调用 GQA 比率：查询组 = 查询头 / KV 头

Changing the number of KV heads affects parameter count especially for the MHA case. For consistency, we adjust the number of layers for the MHA run since it would otherwise have a 100M+ parameter discrepancy; for the rest we keep the default 16 layers.

更改 KV 头的数量会影响参数计数，特别是对于 MHA 的情况。为了保持一致性，我们调整了 MHA 运行的层数，否则它将具有 100M+ 参数差异;对于其余部分，我们保留默认的 16 层。

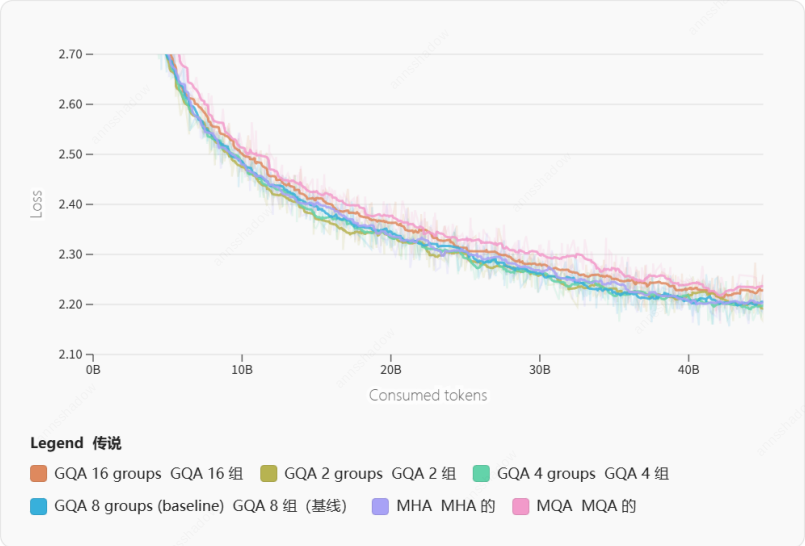
Attention Type 注意力类型	Query Heads 查询头	KV Heads KV 头	Layers 层	Pa
MQA MQA 的	32	1	16	1.
GQA (ratio 16) GQA （比率 16）	32	2	16	1.
GQA (ratio 8) GQA （比率 8）	32	4	16	1.
GQA (ratio 4) GQA （比率 4）	32	8	16	1.
GQA (ratio 2) GQA （比率 2）	32	16	15	1.
MHA MHA 的	32	32	14	1.
GQA (ratio 2) GQA （比率 2）	32	16	16	1.
MHA MHA 的	32	32	16	1.

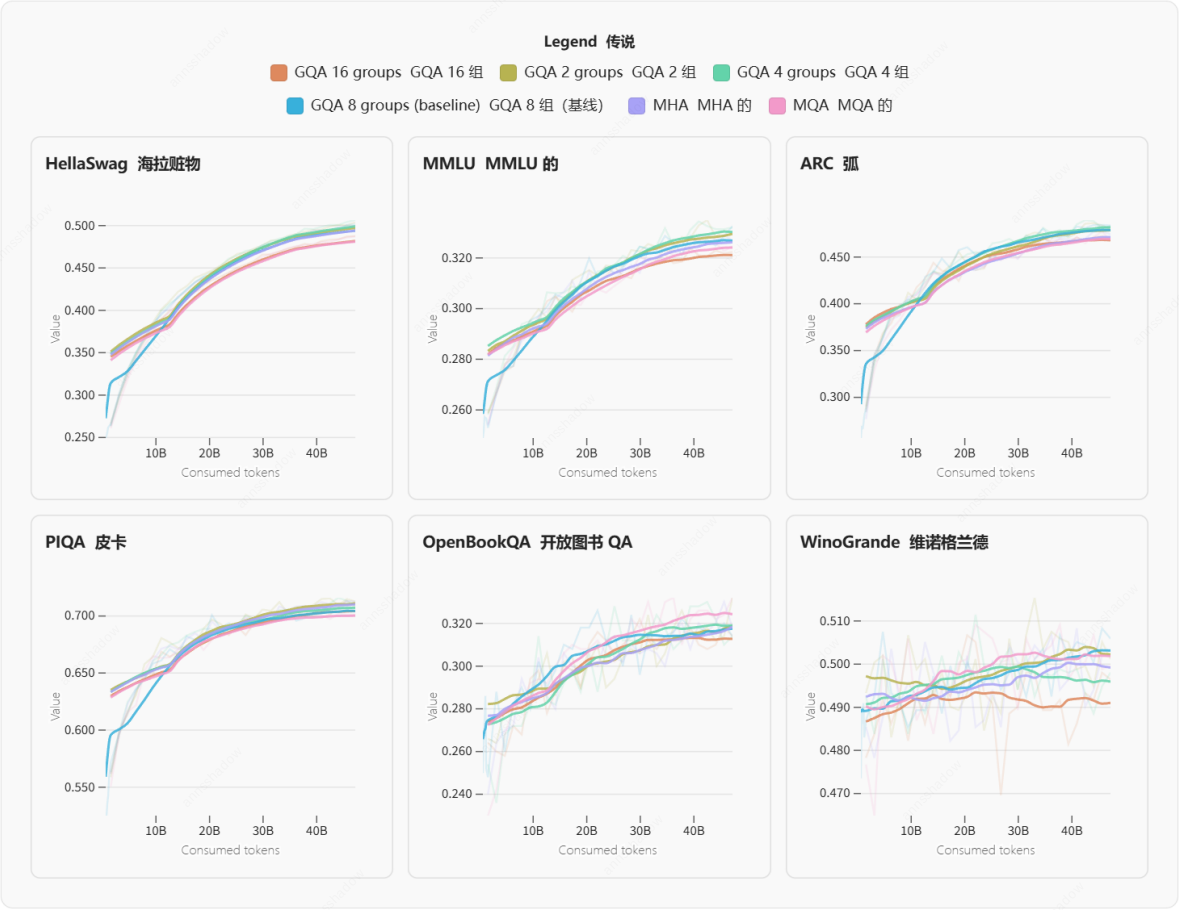
So we compare MHA, MQA and 4 setups for GQA (ratios 2, 4, 8, 16). You can find the nanotron configs here.

因此，我们比较了 GQA 的 MHA、MQA 和 4 种设置（比率 2、4、8、16）。你可以在这里找到 nanotron 配置。

Looking at the ablation results, we find that MQA and GQA with 16 groups (leaving only 1 and 2 KV heads respectively) underperform MHA significantly. On the other hand, GQA configurations with 2, 4, and 8 groups roughly match MHA performance.

从消融结果来看，我们发现 16 组的 MQA 和 GQA（分别只留下 1KV 和 2KV 头）明显落后于 MHA。另一方面，具有 2、4 和 8 组的 GQA 配置与 MHA 性能大致匹配。





The results are consistent across both loss curves and downstream evaluations. We observe this clearly in benchmarks like HellaSwag, MMLU, and ARC, while benchmarks like OpenBookQA and WinoGrande show a bit of noise.

损失曲线和下游评估的结果是一致的。我们在 HellaSwag、MMLU 和 ARC 等基准测试中清楚地观察到了这一点，而 OpenBookQA 和 WinoGrande 等基准测试则显示出一些噪音。

Based on these ablations, GQA is a solid alternative to MHA. It preserves performance while being more efficient at inference. Some recent models have adopted MLA for even greater KV cache compression, though it hasn't been as widely adopted yet.

基于这些消融，GQA 是 MHA 的可靠替代品。它保持性能，同时提高推理效率。一些最近的模型已经采用了 MLA 来实现更大的 KV 缓存压缩，尽管它还没有被广泛采用。

We didn't ablate MLA since it wasn't implemented in nanotron at the time of the ablations. For SmolLM3, we used GQA with 4 groups.

我们没有消融 MLA，因为在消融时它没有在 nanotron 中实施。对于 SmolLM3，我们使用 GQA 对 4 组。

Beyond the attention architecture itself, the attention pattern we use during training also matters. Let's have a look at attention masking.

除了注意力架构本身之外，我们在训练期间使用的注意力模式也很重要。让我们来看看注意力掩蔽。

Document masking 文档遮罩

How we apply attention across our training sequences impacts both computational efficiency and model performance. This brings us to *document masking* and the broader question of how we structure our training samples in the dataloader.

我们如何在训练序列中应用注意力会影响计算效率和模型性能。这给我们带来了文档掩码和更广泛的问题，即我们如何在数据加载器中构建训练样本。

During pretraining, we train with fixed sequence lengths but our documents have variable lengths. A research paper might be 10k tokens while a short code snippet might only have few hundred tokens. How do we fit variable-length documents into fixed-length training sequences?

We could also add BOS tokens at the beginning of documents. In this case you'll notice a different `bos_token_id` present in the model/tokenizer configs.

我们还可以在文档的开头添加 BOS 代币。在这种情况下，您会注意到模型/分词器配置中存在不同的 `bos_token_id` 内容。

在预训练期间，我们使用固定的序列长度进行训练，但我们的文档具有可变长度。一篇研究论文可能是 10k 个令牌，而一个短代码片段可能只有几百个令牌。我们如何将可变长度文档拟合到固定长度的训练序列中？

Padding shorter documents to reach our target length wastes compute on meaningless padding tokens.

填充较短的文档以达到我们的目标长度会浪费在无意义的填充标记上的计算。

Instead, we use**packing** : shuffle and concatenate documents with end-of-sequence (EOS) tokens, then split the result into fixed-length chunks matching the sequence size.

相反，我们使用打包：将文档与序列结束 （EOS） 标记打乱并连接起来，然后将结果拆分为与序列大小匹配的固定长度块。

Here's what this looks like in practice:

这是实践中的样子：

```
File 1: "Recipe for granola bars..." (400 tokens) <EOS>
File 2: "def hello_world()..." (300 tokens) <EOS>
File 3: "Climate change impacts..." (1000 tokens) <EOS>
File 4: "import numpy as np..." (3000 tokens) <EOS>
...
Â
After concatenation and chunking into 4k sequences:
Sequence 1: [File 1] + [File 2] + [File 3] + [partial File 4]
Sequence 2: [rest of File 4] + [File 5] + [File 6] + ...
```

A training sequence might contain one complete file if it's long enough to fill our 4k context, but in most cases files are short, so sequences contain concatenations of multiple random files.

如果训练序列足够长以填充我们的 4k 上下文，则它可能包含一个完整的文件，但在大多数情况下，文件很短，因此序列包含多个随机文件的串联。

With standard causal masking, tokens can attend to all previous tokens in the packed sequence.

使用标准因果掩码，标记可以关注打包序列中的所有先前标记。

In the examples above, a token in that Python function of file 4 can attend to the granola bars recipe, the climate change article, and any other content that happened to be packed together. Let's quickly take a look at what a typical 4k pre-training context would contain.

在上面的示例中，文件 4 的 Python 函数中的标记可以处理格兰诺拉麦片棒食谱、气候变化文章以及碰巧打包在一起的任何其他内容。让我们快速看一下典型的 4k 预训练上下文将包含哪些内容。

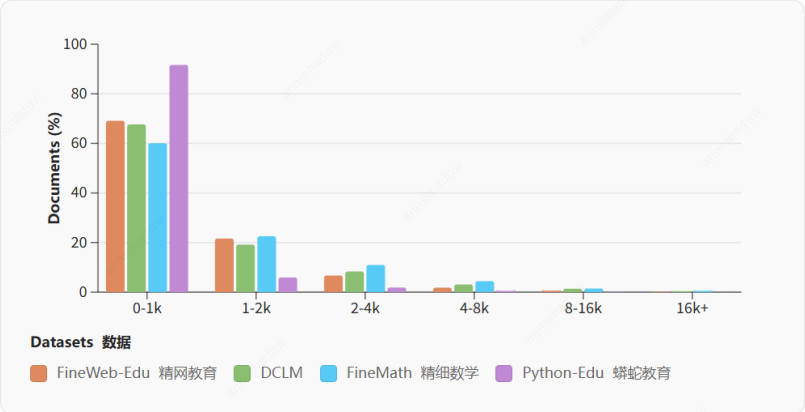
A quick analysis reveals that a substantial portion (about 80-90%) of files in

CommonCrawl and GitHub are shorter than 2k tokens.

快速分析显示，CommonCrawl 和 GitHub 中很大一部分（约 80-90%）的文件都短于 2k 代币。

The chart below examines token distribution for more recent datasets, used throughout this blog:

下图检查了本博客中使用的最新数据集的令牌分布：



More than 80% of documents in FineWeb-Edu, DCLM, FineMath and Python-Edu contain fewer than 2k tokens. This means with a 2k or 4k training sequence and

standard causal masking, the vast majority of tokens would spend compute attending to unrelated documents packed together.

FineWeb-Edu、DCLM、FineMath 和 Python-Edu 中超过 80% 的文档包含少于 2k 标记。这意味着使用 2k 或 4k 训练序列和标准因果屏蔽，绝大多数令牌将花费计算来处理打包在一起的不相关文档。

Longer documents in PDFs

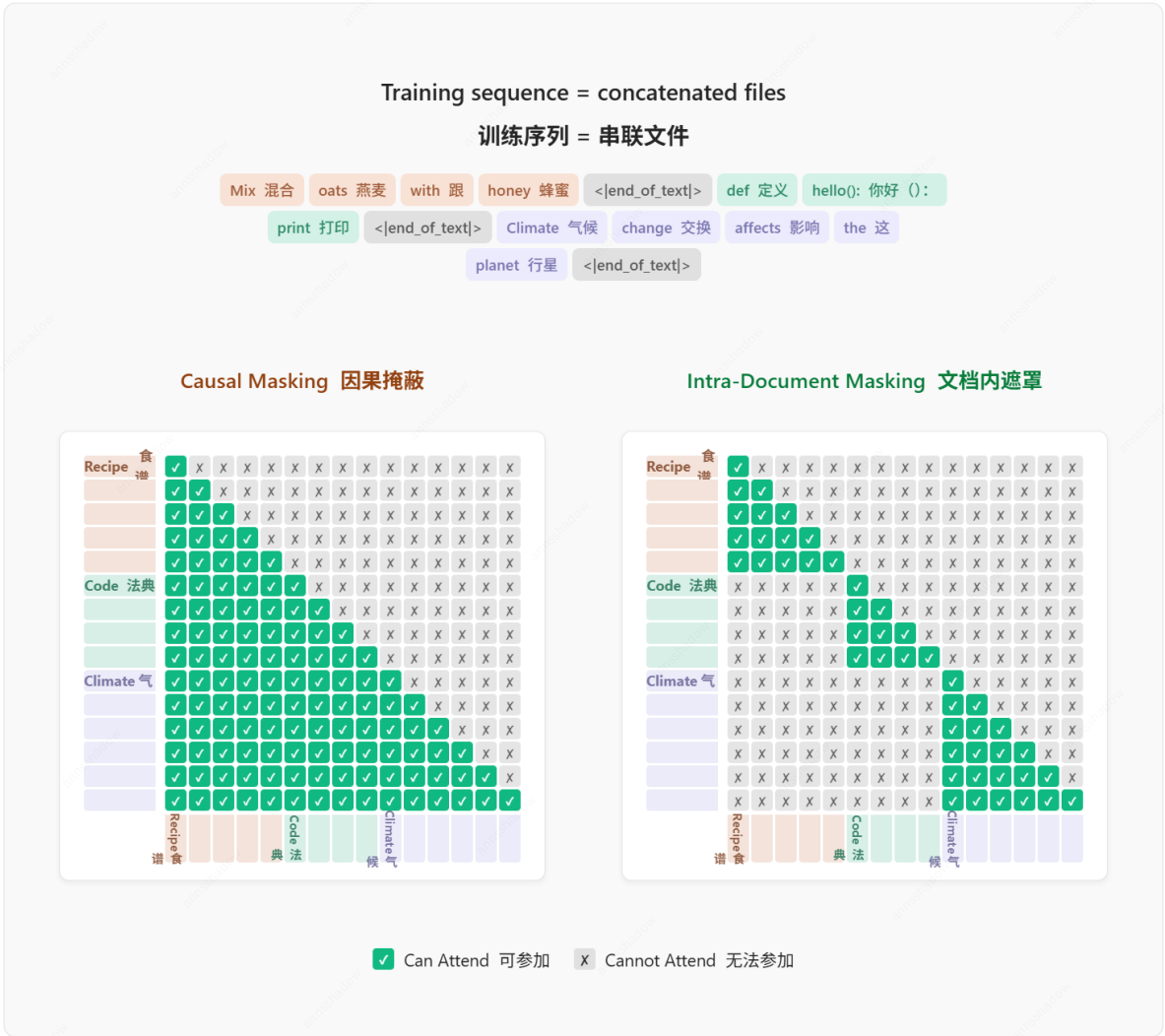
PDF 中的较长文档

While most web-based datasets consist of short documents, PDF-based datasets contain substantially longer content. FinePDFs documents are on average 2× longer than web text, and they improve performance when mixed with FineWeb-Edu and DCLM.

虽然大多数基于 Web 的数据集由简短的文档组成，但基于 PDF 的数据集包含更长的内容。FinePDFs 文档平均比 Web 文本长 2×，并且与 FineWeb-Edu 和 DCLM 混合使用时可以提高性能。

Besides computational inefficiency, Zhao et al. (2024) find that this approach introduces noise from unrelated content that can degrade performance. They suggest using *intra-document masking* , where we modify the attention mask so tokens can only attend to previous tokens within the same document. The visualization below illustrates this difference:

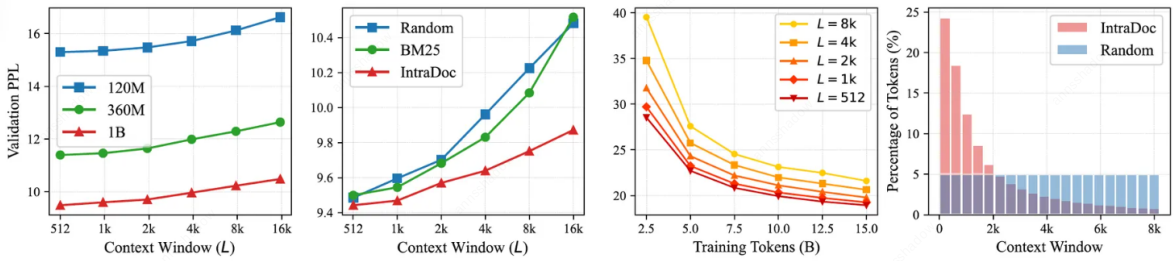
除了计算效率低下之外，Zhao 等人（2024）发现这种方法还会引入来自不相关内容的噪声，从而降低性能。他们建议使用 文档内屏蔽 ，我们修改注意力掩码，以便标记只能关注同一文档中的先前标记。下面的可视化说明了这种差异：



Zhu et al. (2025) in SkyLadder found similar benefits from intra-document masking, but offer a different explanation. They found that shorter context lengths work better for training, and intra-document masking effectively reduces the average context length.

Zhu 等人（2025 年）在 SkyLadder 中发现了文档内掩蔽的类似好处，但提供了不同的解释。

他们发现，较短的上下文长度更适合训练，而文档内屏蔽有效地减少了平均上下文长度。



(a) Validation perplexity of models with varying sizes across different context window sizes.

(b) Validation perplexity of different packing and masking strategies across varying context window sizes.

(c) Validation perplexity of models without positional encoding across varying context window sizes

(d) Distribution of context window sizes for IntraDoc and Random methods.

These plots from SkyLadder demonstrate multiple findings: (a) shorter contexts often perform better during pretraining (lower validation perplexity), (b) intra-document masking (IntraDoc) achieves lower perplexity than both random packing (Random) and semantic grouping (BM25), (c) the shorter context advantage holds even without positional encoding, and (d) IntraDoc creates a distribution skewed toward shorter effective context lengths.

来自 SkyLadder 的这些图展示了多个发现：（a） 较短的上下文通常在预训练期间表现更好（较低的验证困惑度），（b） 文档内掩蔽 （IntraDoc） 比随机打包 （Random） 和语义分组 （BM25） 实现更低的困惑度，（c） 即使没有位置编码，较短的上下文优势也成立，以及 （d） IntraDoc 创建的分布偏向较短的有效上下文长度。

Llama3 (Grattafiori et al., 2024) also trained with intra-document masking, they found limited impact during short context pretraining but significant benefits for long-context extension, where the attention overhead becomes more significant. In addition, the ProLong paper (Gao et al., 2025) showed that using document masking to extend Llama3 8B's context in continual pretraining, benefits both long context and short context benchmarks.

Llama3（Grattafiori 等人，2024 年）也使用文档内掩蔽进行了训练，他们发现在短上下文预训练期间影响有限，但对长上下文扩展有显着好处，因为注意力开销变得更加显着。此外，ProLong 论文（Gao et al.， 2025）表明，在持续预训练中使用文档屏蔽来扩展 Llama3 8B 的上下文，有利于长上下文和短上下文基准。

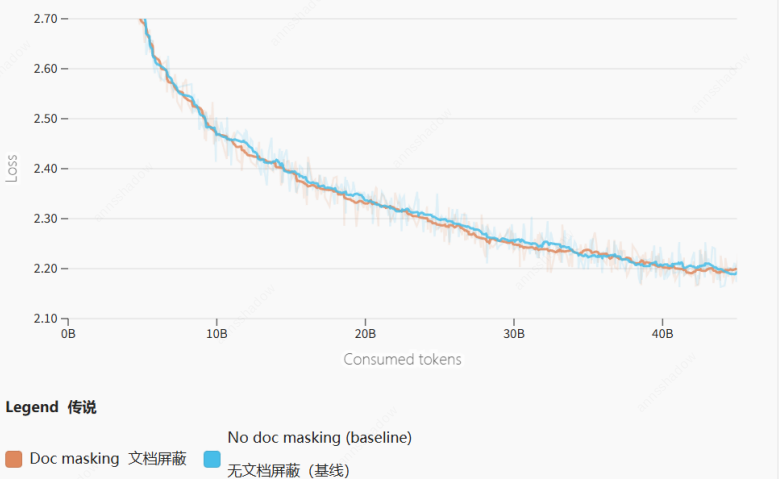
We decided to run an ablation on our 1B baseline model and test whether document masking impacts short-context performance. You can find the config here. The results showed identical loss curves and downstream evaluation scores compared to standard causal masking, as shown in the charts below.

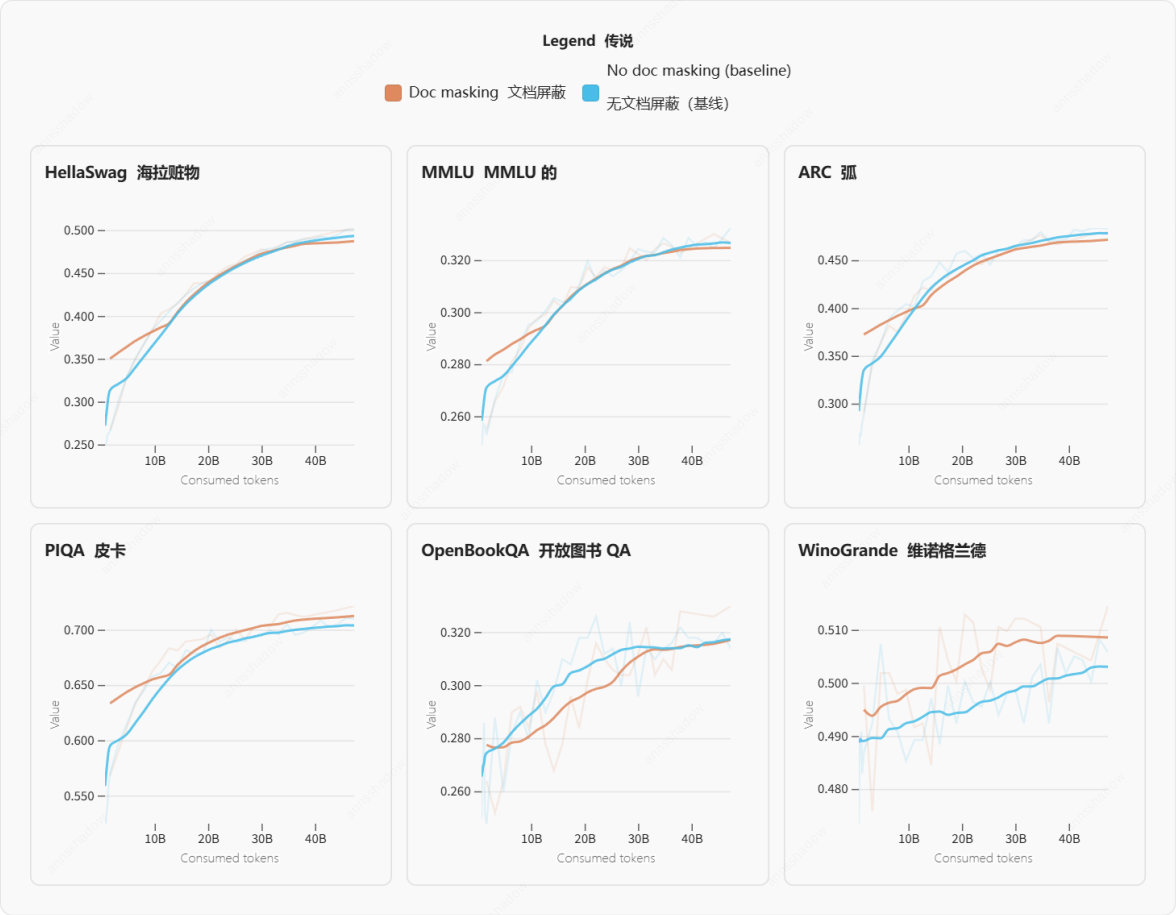
我们决定在 1B 基线模型上运行消融，并测试文档屏蔽是否会影响短上下文性能。你可以在这里找到配置。结果显示，与标准因果掩蔽相比，损失曲线和下游评估分数相同，如下图所示。

To enable document masking in nanotron, simply set this flag to `true` in the model config:

要在 nanotron 中启用文档屏蔽，只需 `true` 在模型配置中将此标志设置为：

```
model_config:
  _attn_implementation: flash_attention_2
  _fused_rms_norm: true
  _fused_rotary_emb: true
- _use_doc_masking: false
+ _use_doc_masking: true
Â
```





Similar to Llama3, we don't observe a noticeable impact on short context tasks, except for a small improvement on PIQA. However, document masking becomes crucial when scaling to long sequences to speed up the training.

与 Llama3 类似，除了对 PIQA 的小幅改进外，我们没有观察到对短上下文任务的明显影响。然而，当扩展到长序列以加快训练速度时，文档屏蔽变得至关重要。

This is particularly important for our long context extension, where we scale from 4k to 64k tokens (detailed in the Training marathon chapter). We therefore adopted it for SmolLM3 throughout the full training run.

这对于我们的长上下文扩展尤为重要，我们将从 4k 标记扩展到 64k（详见训练马拉松一章）。因此，我们在整个训练运行中将其用于 SmolLM3。

We've covered in this section how attention processes sequences. Now let's look at another major parameter block in transformers: the embeddings.

我们在本节中介绍了注意力如何处理序列。现在让我们看看 Transformer 中的另一个主要参数块：嵌入。

EMBEDDING SHARING 嵌入共享

If you look at the config of our baseline ablation model, one thing that is different from a standard transformer is embedding sharing enabled by the flag `tie_word_embeddings` .

如果您查看我们的基线消融模型的配置，会发现与标准 Transformer 不同的一件事是由 flag `tie_word_embeddings` 启用的嵌入共享。

LLMs have two embedding components: input embeddings that serve as a token-to-vector lookup table (of size vocab_size × hidden_dim) and the output embeddings, which is the final linear layer mapping hidden states to vocabulary logits (hidden_dim × vocab_size).

LLM 有两个嵌入组件：作为标记到向量查找表（大小为 vocab_size × hidden_dim）的输入嵌入和输出嵌入，这是将隐藏状态映射到词汇 logits （hidden_dim × vocab_size） 的最终线性层。

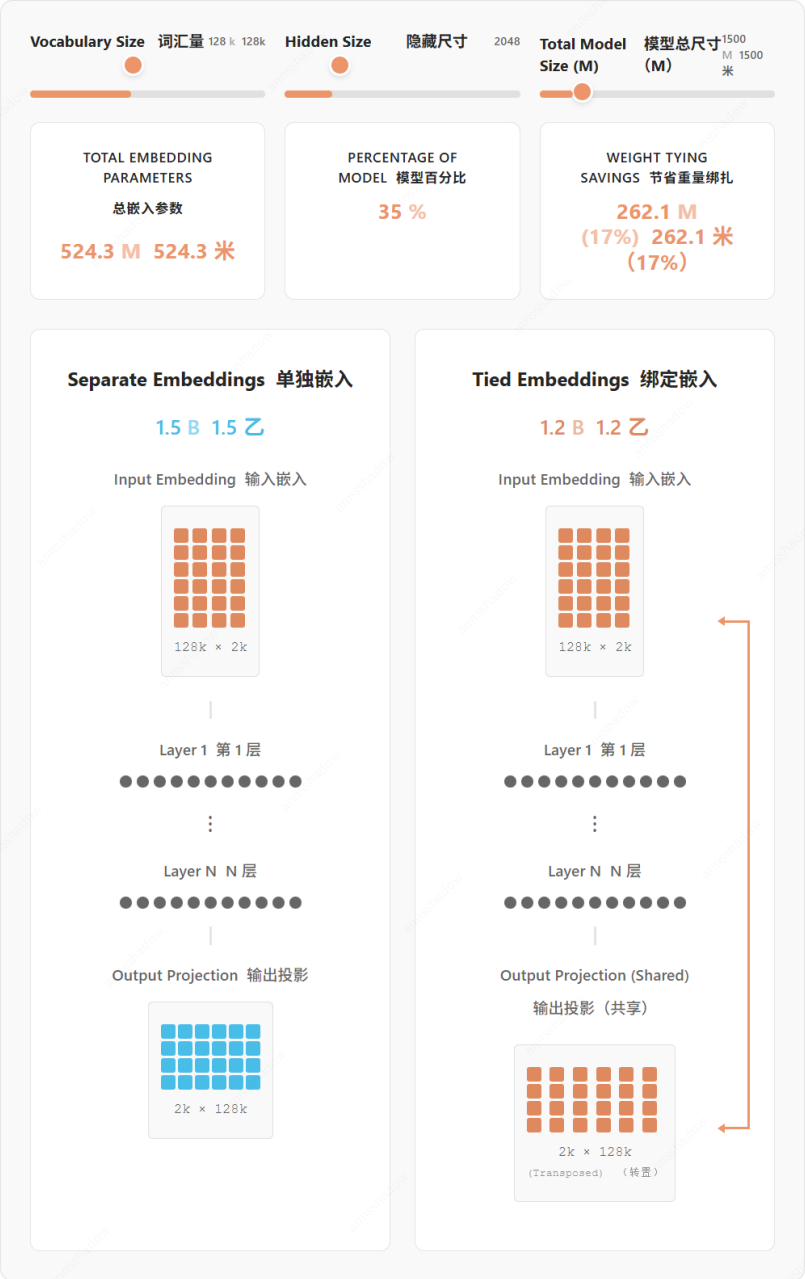
In the classic case where these are separate matrices, total embedding parameters are 2 × vocab_size × hidden_dim. Therefore, in small language models, embeddings can

constitute a large portion of the total parameter count, especially with a large vocabulary size.

在这些是单独矩阵的经典情况下，总嵌入参数为 2 × vocab_size × hidden_dim。因此，在小型语言模型中，嵌入可以占总参数数的很大一部分，尤其是在词汇量较大的情况下。

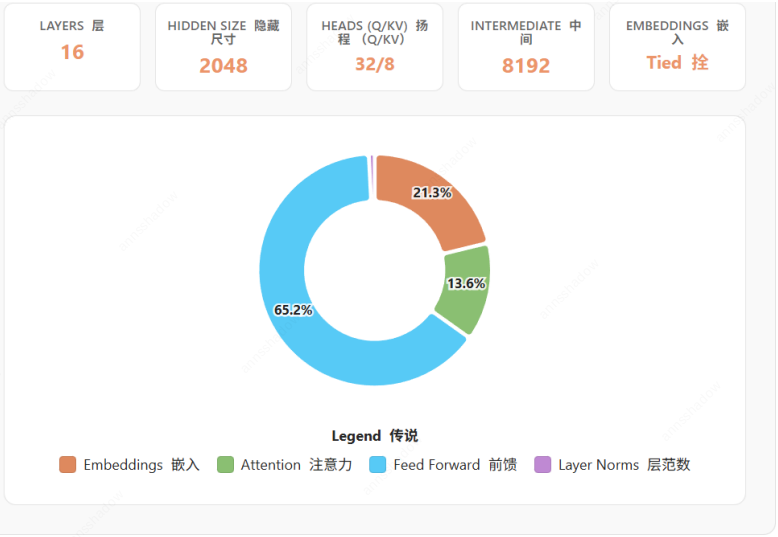
This makes embedding sharing (reusing input embeddings in the output) a natural optimization for small models.

这使得嵌入共享（在输出中重用输入嵌入）成为小型模型的自然优化。



Larger models don't typically use this technique since embeddings represent a smaller fraction of their parameter budget. For example, total embeddings without sharing account for only 13% in Llama3.2 8B and 3% in Llama3.1 70B as show in the pie chart below.

较大的模型通常不使用此技术，因为嵌入只占其参数预算的一小部分。例如，没有共享的总嵌入在 Llama3.2 8B 中仅占 13%，在 Llama3.1 70B 中占 3%，如下图所示。



Ablation - models with tied embeddings match larger untied variants

消融 - 具有绑定嵌入的模型匹配较大的未绑定变体

Now we will assess the impact of embedding sharing on our ablation model. We draw insights from MobileLLM's comprehensive ablations on this technique at 125M scale, where they demonstrated that sharing reduced parameters by 11.8% with minimal accuracy degradation.

现在我们将评估嵌入共享对我们的消融模型的影响。我们从 MobileLLM 对该技术的 125M 规模的全面消融中汲取了见解，他们证明共享将参数减少了 11.8%，而精度下降最小。

Since untied embeddings increase our parameter count from 1.2B to 1.46B, we will train another model with untied parameters but less layers so it matches the baseline 1.2B in parameters count. We will compare two 1.

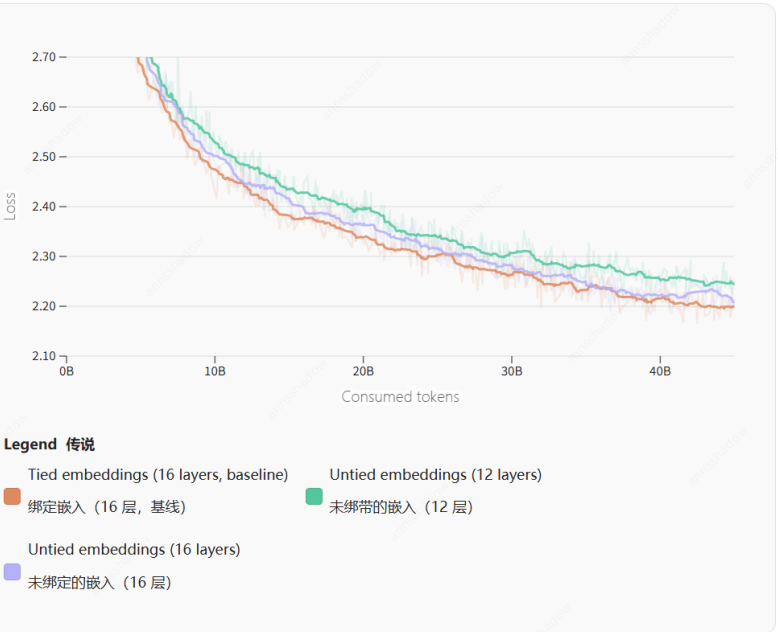
由于未绑定的嵌入将我们的参数计数从 1.2B 增加到 1.46B，因此我们将使用未绑定的参数训练另一个模型，但层数较少，使其在参数计数中与基线 1.2B 匹配。我们将比较两个 1.

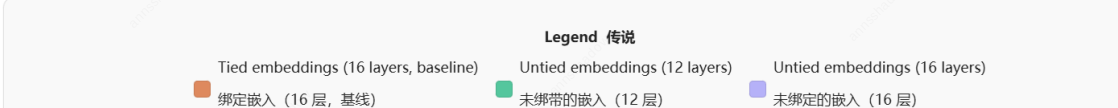
2B models: our baseline with tied embeddings (16 layers) versus an untied version with fewer layers (12 layers) to maintain the same parameter budget, and the 1.46B model with untied embeddings and the same layer count as our baseline (16) as an additional reference point.

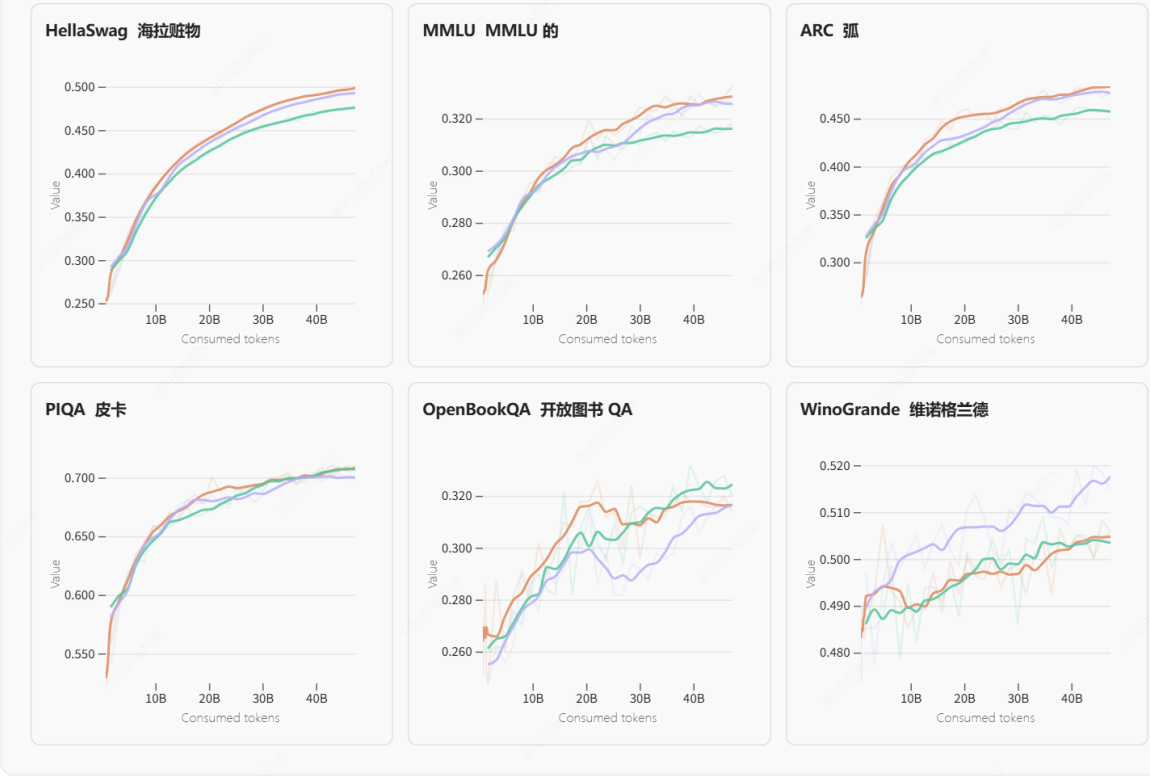
2B 模型：我们的基线具有绑定的嵌入（16 层）与具有较少层数（12 层）的未绑定版本，以保持相同的参数预算，以及具有未绑定嵌入和与基线相同的层数的 1.46B 模型（16）作为额外的参考点。

You can find the nanotron configshere.

你可以在这里找到 nanotron 配置。







The loss and evaluation results demonstrate that our baseline 1.2B model with tied embeddings achieves comparable performance to the 1.46B untied equivalent, on all the benchmarks except for WinoGrande, despite having 18% fewer parameters. The 1.

损失和评估结果表明，尽管参数减少了 18%，但在除 WinoGrande 之外的所有基准测试中，我们的基线 1.2B 模型具有绑定嵌入的性能与 1.46B 未绑定等效模型相当。1.

2B model with untied embeddings and reduced layers (12 vs 16) underperforms both configurations, exhibiting higher loss and lower downstream evaluation scores.

具有未绑定嵌入和减少层（12 与 16）的 2B 模型表现不佳，表现出更高的损失和更低的下游评估分数。

This suggests that increasing model depth provides greater benefits than untying embeddings at equivalent parameter budgets.

这表明，增加模型深度比在等效参数预算下解绑嵌入提供了更大的好处。

Based on these results, we kept tied embeddings for our SmolLM3 3B model.

基于这些结果，我们为 SmolLM3 3B 模型保留了绑定嵌入。

We've now explored the embedding sharing strategy and its tradeoffs. But embeddings alone don't capture the order of tokens in a sequence; providing this information is the role of positional encodings.

我们现在已经探讨了嵌入共享策略及其权衡。但仅嵌入并不能捕获序列中标记的顺序;提供此信息是位置编码的作用。

In the next section, we will look at how positional encoding strategies have evolved, from standard RoPE to newer approaches like NoPE (No Position Embedding), which enable more effective modeling for long contexts.

在下一节中，我们将了解位置编码策略是如何演变的，从标准 RoPE 到 NoPE（无位置嵌入）等更新方法，这些方法可以为长上下文进行更有效的建模。

POSITIONAL ENCODINGS & LONG CONTEXT
位置编码和长上下文

When transformers process text, they face a fundamental challenge: they naturally have no sense of word order, since they consume entire sequences simultaneously through parallel attention operations. This enables efficient training but creates a problem.

当转换器处理文本时，它们面临着一个基本挑战：它们自然没有词序感，因为它们通过并行注意力作同时消耗整个序列。这可以实现高效的训练，但会产生问题。

Without explicit position information, "Adam beats Muon" looks similar to "Muon beats Adam" from the model's perspective.

如果没有明确的位置信息，从模型的角度来看，“Adam beats Muon”看起来与“Muon beats

Adam"相似。

The solution is positional embeddings: mathematical encodings that give each token a unique "address" in the sequence.

解决方案是位置嵌入：数学编码，在序列中为每个标记提供唯一的"地址"。

But as we push toward longer and longer contexts - from the 512 tokens of early BERT to today's million-token models - the choice of positional encoding becomes increasingly critical for both performance and computational efficiency.

但随着我们朝着越来越长的上下文迈进——从早期 BERT 的 512 个令牌到今天的百万令牌模型——位置编码的选择对于性能和计算效率变得越来越重要。

The Evolution of Position Encoding

位置编码的演变

Early transformers used simple **Absolute Position Embeddings (APE)** (Vaswani et al., 2023), essentially learned lookup tables that mapped each position (1, 2, 3...) to a vector that gets added to token embeddings.

早期的 transformers 使用简单的绝对位置嵌入 （APE） （Vaswani 等人，2023 年），本质上是学习查找表，将每个位置（1、2、3...）映射到添加到标记嵌入中的向量。

This worked fine for short sequences but had a major limitation: models max input sequence length was limited to the max input sequence length they were trained on. They had no out-of-the-box generalisation capabilities to longer sequences.

这对于短序列效果很好，但有一个主要限制：模型的最大输入序列长度被限制为它们训练的最大输入序列长度。它们没有开箱即用的对较长序列的泛化能力。

The field evolved toward **relative position encodings** that capture the distance between tokens rather than their absolute positions. This makes intuitive sense, whether two words are 3 positions apart matters more than whether they're at positions (5,8) versus (105,108).

该领域向相对位置编码发展，捕获标记之间的距离而不是它们的绝对位置。这很直观，两个词是否相隔 3 个位置比它们是否位于位置 （5,8） 与 （105,108） 更重要。

For a deeper dive into positional encoding, this blog walks through the step-by-step development from basic positioning to rotational encoding.

为了更深入地了解位置编码，本博客将逐步介绍从基本定位到旋转编码的开发过程。

ALiBi (Attention with Linear Biases) (Press et al., 2022), in particular, modifies the attention scores based on token distance. The further apart two tokens are, the more their attention gets penalized through simple linear biases applied to attention weights. For a detailed implementation of Alibi, check this resource.

特别是 ALiBi（线性偏差注意力） （Press 等人，2022 年）根据标记距离修改注意力分数。两个标记相距越远，它们的注意力就越会因应用于注意力权重的简单线性偏差而受到惩罚。有关 Ablibi 的详细实现，请查看此资源。

But the technique that has dominated recent large language models is Rotary Position Embedding (RoPE) (Su et al., 2023).

但最近主导大型语言模型的技术是旋转位置嵌入（RoPE） （Su et al.， 2023）。

RoPE: Position as Rotation

RoPE：位置作为旋转

RoPE's core insight is to encode position information as rotation angles in a high-dimensional space. Instead of adding position vectors to token embeddings, RoPE rotates the query and key vectors by angles that depend on their absolute positions.

RoPE 的核心见解是将位置信息编码为高维空间中的旋转角度。RoPE 不是将位置向量添加到标记嵌入中，而是按取决于其绝对位置的角度旋转查询和键向量。

The intuition is that we treat each pair of dimensions in our embeddings as coordinates on a circle and rotate them by an angle determined by:

直觉是，我们将嵌入中的每一对维度视为圆上的坐标，并按以下决定的角度旋转它们：

- The token's position in the sequence

  令牌在序列中的位置

- Which dimension pair we're working with (different pairs rotate at different frequencies which are exponents of a base/reference frequency)

  我们正在使用哪个维度对（不同的维度对以不同的频率旋转，这些频率是基频/参考频率的指数）

```python
import torch
Â
def apply_rope_simplified(x, pos, dim=64, base=10000):
    """
```

```
    Rotary Position Embedding (RoPE)
Â
    Idea:
    - Each token has a position index p (0, 1, 2, ...).
    - Each pair of vector dimensions has an index k (0 .. dim/2 - 1).
    - RoPE rotates every pair [x[2k], x[2k+1]] by an angle θ_{p,k}.
Â

    Formula:
      θ_{p,k} = p * base^(-k / (dim/2))
Â
    - Small k (early dimension pairs) → slow oscillations → capture long-range ir
    - Large k (later dimension pairs) → fast oscillations → capture fine detail.
Â
    """
    rotated = []
    for i in range(0, dim, 2):
        k = i // 2  # index of this dimension pair
Â
        # Frequency term: higher k → faster oscillation
        inv_freq = 1.0 / (base ** (k / (dim // 2)))
        theta = pos * inv_freq  # rotation angle for position p and pair k
Â
        cos_t = torch.cos(torch.tensor(theta, dtype=x.dtype, device=x.device))
        sin_t = torch.sin(torch.tensor(theta, dtype=x.dtype, device=x.device))
Â
        x1, x2 = x[i], x[i+1]
Â
        # Apply 2D rotation
        rotated.extend([x1 * cos_t - x2 * sin_t,
                        x1 * sin_t + x2 * cos_t])
Â
    return torch.stack(rotated)


## Q, K: [batch, heads, seq, d_head]
Q = torch.randn(1, 2, 4, 8)
K = torch.randn(1, 2, 4, 8)
Â
## 👉 apply RoPE to Q and K *before* the dot product
Q_rope = torch.stack([apply_rope(Q[0,0,p], p) for p in range(Q.size(2))])
K_rope = torch.stack([apply_rope(K[0,0,p], p) for p in range(K.size(2))])
Â
scores = (Q_rope @ K_rope.T) / math.sqrt(Q.size(-1))
attn_weights = torch.softmax(scores, dim=-1)
```

This code might seem complex so let's break it down with a concrete example. Consider the word *"fox"* from the sentence *"The quick brown fox"*. In our baseline 1B model, each attention head works with a 64-dimensional query/key vector. RoPE groups this vector into 32 pairs: (x_1, x_2), (x_3, x_4), (x_5, x_6), and so on. We work on pairs because we rotate around circles in 2D space.

这段代码可能看起来很复杂，所以让我们用一个具体的例子来分解它。考虑一下句子"The quick brown fox"中的"fox"这个词。在我们的基线 1B 模型中，每个注意力头都使用 64 维查询/键向量。RoPE 将此向量分为 32 对：(x_1, x_2)、(x_3, x_4)、(x_5, x_6) 等。我们处理成对的工作是因为我们在 2D 空间中围绕圆旋转。

For simplicity, let's focus on the first pair (x_1, x_2).

为简单起见，让我们关注第一对 (x_1, x_2)。

The word "fox" appears at position 3 in our sentence, so RoPE will rotate this first dimension pair by:

"狐狸"这个词出现在我们句子中的位置 3，因此 RoPE 将通过以下方式旋转这个第一维对：

```
rotation_angle = position × θ₀
               = 3 × (1/10000^(0/32))
               = 3 × 1.0
               = 3.0 radians
               = 172° degrees
```

Our base frequency is 10000 but for the first dimension pair (k=0) our exponent is zero so the base frequency doesn't affect the calculation (we raise to the power of 0). The visualization below illustrates this:

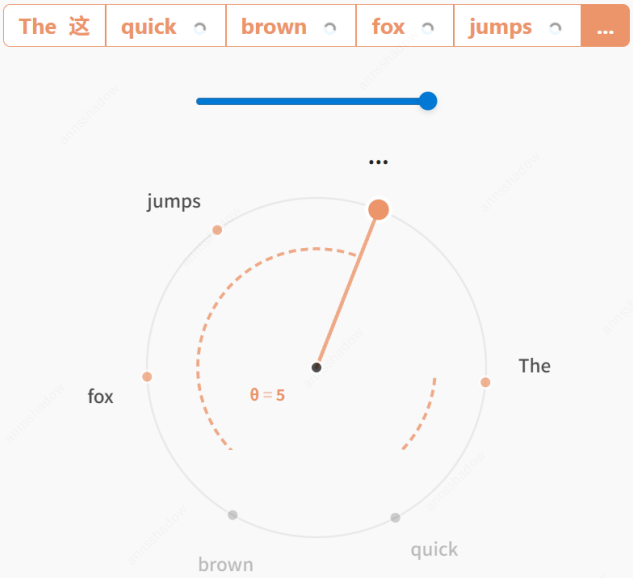
我们的基频是 10000，但对于第一个维度对 (k=0)，我们的指数为零，因此基频不会影响计算（我们提高到 0 的幂）。下面的可视化效果说明了这一点：

RoPE rotation of the first (x_1, x_2) pair in Q/K vectors

Q/K 向量中第一对 (x_1, x_2) 的 RoPE 旋转

based on token position

基于代币头寸



The at position 0 gets rotated by

位置 0 的旋转由

θ = 0 rad (0°)

θ = 0 rad (0°)

RoPE Formula: θ (theta) = position × 1 / base$^{2 \times \text{pair_index/h_dim}}$ (pair_index=0 here)

RoPE 公式：θ （theta） = 位置 × 1 / 底数 $^{2 \times \text{pair\_index/h\_dim}}$ （此处为 pair_index=0）

Key insight: The first dimension pair gets the largest rotations, and the relative angle between words depends only on their distance apart.

关键见解：第一个维度对获得最大的旋转，单词之间的相对角度仅取决于它们之间的距离。

Now the magic happens when two tokens interact through attention. The dot product between their rotated representations directly encodes their relative distance through the phase difference between their rotation angles (where `m` and `n` are the token positions)

现在，当两个代币通过注意力相互作用时，奇迹就会发生。它们旋转表示之间的点积通过它们的旋转角度之间的相位差直接编码它们的相对距离（其中 `m` 和 `n` 是标记位置）

```
dot_product(RoPE(x, m), RoPE(y, n)) = Σₖ [xₖ * yₖ * cos((m-n) * θₖ)]
```

The attention pattern depends only on (m-n), so tokens that are 5 positions apart will always have the same angular relationship, regardless of their absolute positions in the sequence.

注意力模式仅取决于 （m-n），因此相距 5 个位置的标记将始终具有相同的角度关系，无论它们在序列中的绝对位置如何。

到更长的序列。

How to set RoPE Frequency?

如何设置 RoPE 频率?

In practice, most LLM pretraining starts with relatively short context lengths (2K-4K tokens) using RoPE base frequencies of a few tens thousands like 10K or 50K.

在实践中，大多数 LLM 预训练从相对较短的上下文长度（2K-4K 标记）开始，使用 RoPE 基本频率为几万，如 10K 或 50K。

Training with very long sequences from the start would be computationally expensive due to attention's quadratic scaling with sequence length and the limited availability of long-context data (samples > 4K context length) as we've seen before in the

document masking section ofAttention. Research also suggests it can hurt short-context performance (Zhu et al., 2025). Models typically start by learning short range correlation between words so long sequences don't help much. The typical approach is to do most pretraining with shorter sequences, then do continual pretraining or spend the final few hundred billion tokens on longer sequences.

正如我们之前在 Attention 的文档掩蔽部分看到的那样，由于注意力随序列长度的二次缩放以及长上下文数据（> 4K 上下文长度的样本）的可用性有限，从一开始就使用非常长的序列进行训练在计算上会很昂贵。研究还表明，它可能会损害短期环境绩效（Zhu 等人，2025 年）。模型通常从学习单词之间的短程相关性开始，因此长序列没有多大帮助。典型的方法是使用较短的序列进行大多数预训练，然后进行持续的预训练或将最后的数千亿个代币用于较长的序列。

However, as sequence lengths grow, the rotation angles which are proportional to token positions, grow and can cause attention scores for distant tokens to decay too rapidly(Rozière et al., 2024; Xiong et al., 2023):

然而，随着序列长度的增长，与标记位置成正比的旋转角度会增长，并可能导致远处标记的注意力分数衰减过快（Rozière 等人，2024 年;Xiong 等人，2023 年）：

```
θ = position x 1 / (base^(k/(dim/2)))
```

The solution is to increase the base frequency as the sequence length is increased in order to prevent such decaying, using methods like ABF and YaRN.

解决方案是使用 ABF 和 YaRN 等方法，随着序列长度的增加而增加碱基频率，以防止这种衰减。

RoPE ABF (RoPE with Adjusted Base Frequency) (Xiong et al., 2023b): addresses the attention decay problem in long contexts by increasing the base frequency in RoPE's formulation. This adjustment slows down the rotation angles between token positions, preventing distant tokens' attention scores from decaying too rapidly.

RoPE ABF（具有调整基频的 RoPE）（Xiong 等人，2023b）：通过增加 RoPE 公式中的基频来解决长时间上下文中的注意力衰减问题。这种调整减慢了代币位置之间的旋转角度，防止远处代币的注意力分数衰减过快。

ABF can be applied in a single stage (direct frequency boost) or multi-stage (gradual increases as context grows).

ABF 可以应用于单级（直接频率提升）或多级（随着环境的增长逐渐增加）。

The method is straightforward to implement and distributes embedded vectors with increased granularity, making distant positions easier for the model to differentiate.

该方法实施简单，并以更高的粒度分布嵌入向量，使模型更容易区分远处位置。

While simple and effective, ABF's uniform scaling across all dimensions may not be optimal for extremely long contexts.

虽然简单有效，但 ABF 在所有维度上的统一缩放对于极长的上下文可能不是最佳选择。

YaRN (Yet another RoPE extensioN) (Peng et al., 2023): takes a more sophisticated approach by interpolating frequencies unevenly across RoPE dimensions using a ramp or scaling function. Unlike ABF's uniform adjustment, YaRN applies different scaling factors to different frequency components, optimizing the extended context window.

YaRN（另一个 RoPE 扩展）（Peng 等人，2023 年）：采用更复杂的方法，使用斜坡或缩放函数在 RoPE 维度上不均匀地插值频率。与 ABF 的统一调整不同，YaRN 对不同的频率分量应用不同的缩放因子，优化了扩展的上下文窗口。

It includes additional techniques like dynamic attention scaling and temperature adjustment in attention logits, which help preserve performance at very large context sizes.

它包括其他技术，例如动态注意力缩放和注意力 logit 中的温度调整，这有助于在非常大的上下文大小下保持性能。

YaRN enables efficient "train short, test long" strategies, requiring fewer tokens and less fine-tuning for robust extrapolation.

YaRN 支持高效的"训练短，测试长"策略，需要更少的标记和更少的微调来进行稳健的外推。

While more complex than ABF, YaRN generally delivers better empirical performance for extremely long contexts by providing smoother scaling and mitigating catastrophic attention loss.

虽然 YaRN 比 ABF 更复杂，但通过提供更平滑的缩放和减轻灾难性的注意力损失，通常为极长的上下文提供更好的经验性能。

It can also be leveraged in inference alone without any finetuning.

它也可以单独用于推理，无需任何微调。

These frequency adjustment methods slow down the attention score decay effect and maintain the contribution of distant tokens.

这些频率调整方式减缓了注意力分数衰减效应，保持了远方代币的贡献。

For instance, the training of Qwen3 involved increasing the frequency from 10k to 1M using ABF as the sequence length was extended from 4k to 32k context (the team then applies YaRN to reach 131k, 4x extrapolation).

例如，Qwen3 的训练涉及使用 ABF 将频率从 10k 增加到 1M，因为序列长度从 4k 扩展到 32k（然后团队应用 YaRN 达到 131k，4x 外推）。

Note that there's no strong consensus on optimal values, and it's usually good to experiment with different RoPE values during the context extension phase to find what works best for your specific setup and evaluation benchmarks.

请注意，对于最佳值没有强烈的共识，通常最好在上下文扩展阶段尝试不同的 RoPE 值，以找到最适合您的特定设置和评估基准的值。

Most major models today use RoPE: Llama, Qwen, Gemma, and many others. The technique has proven robust across different model sizes and architectures (dense, MoE, Hybrid). Let's have a look at a few flavours of rope that have emerged recently.

如今，大多数主要模型都使用 RoPE：Llama、Qwen、Gemma 等。该技术已被证明在不同的模型大小和架构（密集、MoE、混合）中都是稳健的。让我们来看看最近出现的几种绳索口味。

Hybrid Positional Encoding Approaches

混合位置编码方法

However as models push toward increasingly large contexts (Meta AI, 2025; Yang et al., 2025), even RoPE started to hit performance challenges. The standard approach of

increasing RoPE's frequency during long context extension has limitations when evaluated on long context benchmarks more challenging than Needle in the Haystack (NIAH) (Kamradt, 2023), such as Ruler and HELMET (Hsieh et al., 2024; Yen et al., 2025). Newer techniques have been introduced to help.

然而，随着模型向越来越大的上下文迈进（Meta AI，2025 年;Yang 等人，2025 年），甚至 RoPE 也开始遇到性能挑战。在长上下文扩展期间增加 RoPE 频率的标准方法在比大海捞针（NIAH） 更具挑战性的长上下文基准上进行评估时存在局限性（Kamradt，2023），例如 Ruler 和 HELMET（Hsieh 等人，2024 年;Yen 等人，2025 年）。已经引入了更新的技术来提供帮助。

We started this section by saying that transformers need positional information to understand token order but recent research has challenged this assumption. What if explicit positional encodings weren't necessary after all?

我们在本节开始时说，transformer 需要位置信息来理解标记顺序，但最近的研究挑战了这一假设。如果毕竟不需要显式位置编码怎么办?

NoPE (No Position Embedding) (Kazemnejad et al., 2023) trains transformers without any explicit positional encoding, allowing the model to implicitly learn positional information through causal masking and attention patterns. The authors show that this approach demonstrates better length generalisetion compared to ALiBi and RoPE.

NoPE（No Position Embedding）（Kazemnejad 等人，2023）在没有任何显式位置编码的情况下训练 transformer，允许模型通过因果掩蔽和注意力模式隐式学习位置信息。作者表明，与 ALiBi 和 RoPE 相比，这种方法表现出更好的长度泛化。

Without explicit position encoding to extrapolate beyond training lengths, NoPE naturally handles longer contexts.

如果没有显式位置编码来外推超出训练长度，NoPE 自然会处理更长的上下文。

In practice though, NoPE models tend to show weaker performance on short context reasoning and knowledge tasks compared to RoPE (Yang et al.). This suggests that while explicit positional encodings may limit extrapolation, they provide useful inductive biases for tasks within the training context length.

但在实践中，与 RoPE 相比，NoPE 模型在短上下文推理和知识任务上往往表现出较弱的性能（Yang 等人）。这表明，虽然显式位置编码可能会限制外推，但它们为训练上下文长度内的任务提供了有用的归纳偏差。

RNoPE Hybrid Approach: Given these trade-offs; B. Yang et al. (2025) suggest that combining different positional encoding strategies might be interesting. They introduce, RNoPE alternates between RoPE and NoPE layers throughout the model.

RNoPE 混合方法：鉴于这些权衡;B. Yang 等人（2025）认为，结合不同的位置编码策略可能会很有趣。他们引入了 RNoPE 在整个模型中的 RoPE 和 NoPE 层之间交替。

RoPE layers provide explicit positional information and handle local context with recency bias, while NoPE layers improve information retrieval across long distances.

RoPE 层提供显式位置信息，并处理具有新近偏差的本地上下文，而 NoPE 层则改善了长距离信息检索。

This technique was recently used in Llama4, Command A and**SmolLM3** .

该技术最近用于 Llama4、Command A 和 SmolLM3 。

> 📍 **Naming convention 命名约定**
>
> We'll call RNoPE "NoPE" for the rest of this blog to keep things simple. (You'll often see people use "NoPE" to mean RNoPE in discussions).
>
> 为了简单起见，我们将在本博客的其余部分将 RNoPE 称为"NoPE"。（您经常会看到人们在讨论中使用"NoPE"来表示 RNoPE）。

Ablation - NoPE matches RoPE on short context

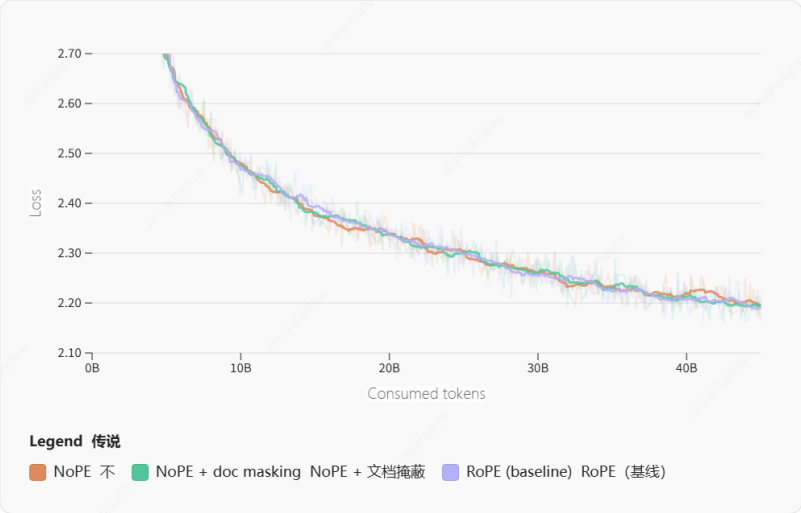
消融 - NoPE 在短上下文中与 RoPE 匹配

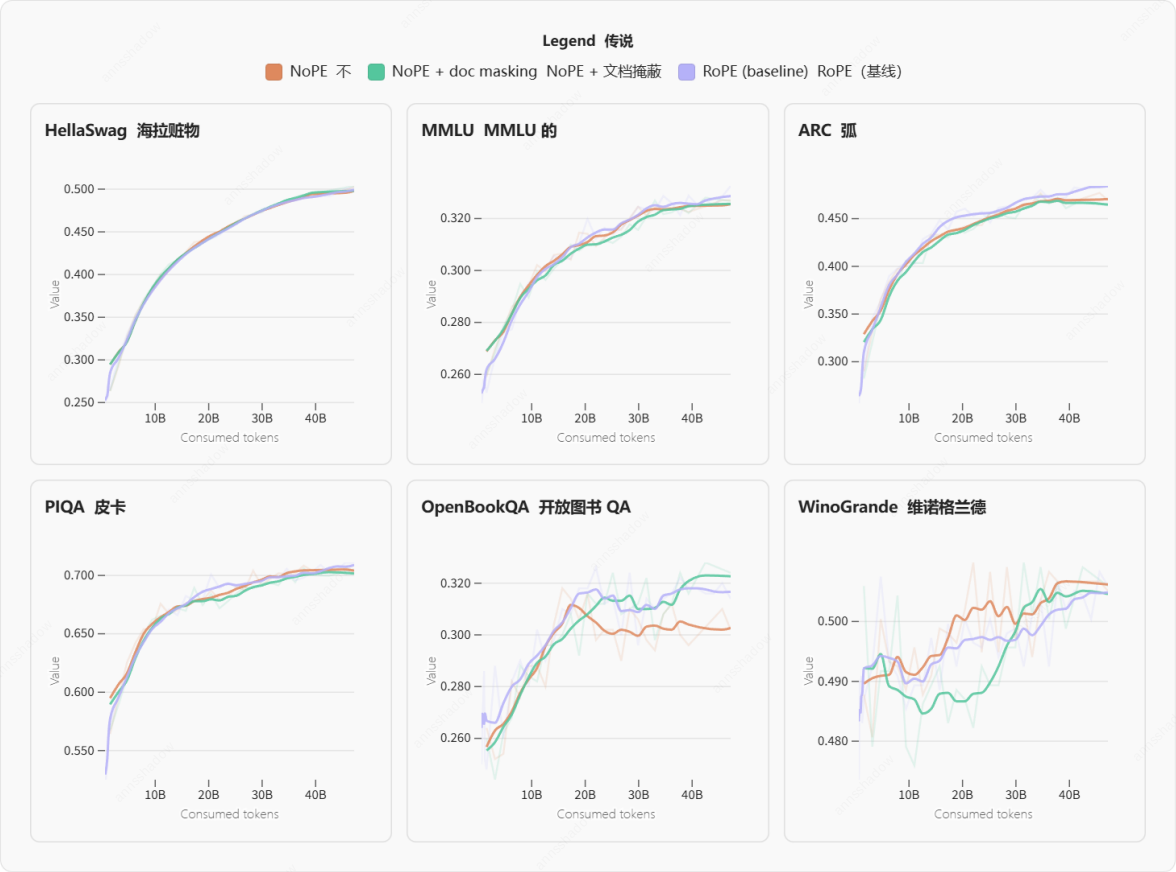
Let's test the hybrid NoPE approach. We'll compare a pure RoPE 1B ablation baseline against a NoPE variant that removes positional encoding every 4th layer, and a third configuration combining NoPE with document masking to test the interaction between these techniques.

让我们测试一下混合 NoPE 方法。我们将比较纯 RoPE 1B 消融基线与每 4 层删除一次位置编码的 NoPE 变体，以及将 NoPE 与文档屏蔽相结合的第三种配置，以测试这些技术之间的相互作用。

Our base question is: can we maintain strong short-context performance while gaining long-context capabilities?

我们的基本问题是：我们能否在获得长上下文能力的同时保持强大的短上下文性能?





The loss and evaluation results show similar performance across all three configurations, indicating that NoPE maintains strong short-context capabilities while providing the foundation for better long-context handling.

损失和评估结果显示，所有三种配置的性能相似，表明 NoPE 保持了强大的短上下文能力，同时为更好的长上下文处理提供了基础。

Given these results, we adopted the NoPE + document masking combination for SmolLM3.

鉴于这些结果，我们对 SmolLM3 采用了 NoPE + 文档掩蔽组合。

Partial/Fractional RoPE: Another complementary idea is to only apply RoPE on a subset of the model dimension. Unlike RNoPE, which alternates entire layers between RoPE and NoPE, Partial RoPE mixes them within the same layer. Recent models such as GLM-4.5 (5 Team et al., 2025) or Minimax-01 (MiniMax et al., 2025) adopt this strategy but this was also present in older models such as gpt-j (Wang & Komatsuzaki, 2021). You will also see this in every model using MLA since it's a must have to have reasonable inference cost.

部分/部分 RoPE：另一个补充的想法是仅将 RoPE 应用于模型维度的子集。与 RNoPE 不同，RNoPE 在 RoPE 和 NoPE 之间交替使用整个层，部分 RoPE 将它们混合在同一层内。最近的模型，如 GLM-4.5（5 Team 等人，2025 年）或 Minimax-01（MiniMax 等人，2025 年）采用了这种策略，但这也存在于 gpt-j（Wang & Komatsuzaki，2021 年）等旧模型中。您

还将在使用 MLA 的每个模型中看到这一点，因为它必须具有合理的推理成本。

> **Technical explanation: Why Partial RoPE is essential for MLA**
>
> **技术解释：为什么部分 RoPE 对 MLA 至关重要**
>
> MLA makes inference efficient with projection absorption: instead of storing per-head keys $k_i^{(h)}$, it caches a small shared latent $c_i = x_i W_c \in \mathbb{R}^{d_c}$ and merges the head's query/key maps so each score is cheap. With $q_t^{(h)} = x_t W_q^{(h)}$ and $k_i^{(h)} = c_i E^{(h)}$, define $U^{(h)} = W_q^{(h)} E^{(h)}$ to get:
>
> MLA 通过投影吸收使推理变得高效：它不是存储每个头的键 $k_i^{(h)}$，而是缓存一个小的共享潜在键 $c_i = x_i W_c \in \mathbb{R}^{d_c}$ 并合并头部的查询/键映射，因此每个分数都很便宜。使用 $q_t^{(h)} = x_t W_q^{(h)}$ 和 $k_i^{(h)} = c_i E^{(h)}$，define $U^{(h)} = W_q^{(h)} E^{(h)}$ 以获得：
>
> $$s_{t,i}^{(h)} = \frac{1}{\sqrt{d_k}} \left(q_t^{(h)}\right)^\top k_i^{(h)} = \frac{1}{\sqrt{d_k}} \left(x_t U^{(h)}\right)^\top c_i$$
>
> so you compute with $\tilde{q}_t^{(h)} = x_t U^{(h)} \in \mathbb{R}^{d_c}$ against the tiny cache c_i (no per-head k stored). RoPE breaks this because it inserts a pair-dependent rotation between the two maps: with full-dim RoPE,
>
> 因此，您针对微小的缓存 c_i （不存储每人头 k） $\tilde{q}_t^{(h)} = x_t U^{(h)} \in \mathbb{R}^{d_c}$ 进行计算。RoPE 打破了这一点，因为它在两个映射之间插入了对依赖的旋转：使用全暗 RoPE，
>
> $$s_{t,i}^{(h)} = \frac{1}{\sqrt{d_k}} \left(x_t W_q^{(h)}\right)^\top \underbrace{R_{t-i}}_{\text{depends on } t-i} \left(c_i E^{(h)}\right)$$
>
> so you can't pre-merge $W_q^{(h)}$ and $E^{(h)}$ into a fixed $U^{(h)}$. Fix: partial RoPE. Split head dims $d_k = d_{\text{nope}} + d_{\text{rope}}$, apply no rotation on the big block (absorb as before: $(x_t U_{\text{nope}}^{(h)})^\top c_i$) and apply RoPE only on a small block.
>
> 因此，您无法预先合并 $W_q^{(h)}$ 并 $E^{(h)}$ 进入固定 $U^{(h)}$ 的 .修复：部分 RoPE。分头调暗 $d_k = d_{\text{nope}} + d_{\text{rope}}$，在大块上不旋转（像以前一样吸收：$(x_t U_{\text{nope}}^{(h)})^\top c_i$），只在小块上应用 RoPE。

Limiting Attention Scope for Long Contexts

限制长上下文的注意力范围

So far, we've explored how to handle positional information for long contexts: activating RoPE, disabling it (NoPE), applying it partially on some layers (RNoPE) or on some hidden dimensions (Partial RoPE), or adjusting its frequency (ABF, YaRN).

到目前为止，我们已经探索了如何处理长上下文的位置信息：激活 RoPE、禁用它（NoPE）、将其部分应用于某些层 （RNoPE） 或某些隐藏维度 （Partial RoPE），或调整其频率 （ABF、YaRN）。

These approaches modify how the model encodes position to handle sequences longer than those seen during training. But there's a complementary strategy: instead of adjusting positional encodings, we can limit which tokens attend to each other.

这些方法修改了模型对位置进行编码的方式，以处理比训练期间看到的序列更长的序列。但有一个互补的策略：我们可以限制哪些标记相互关注，而不是调整位置编码。

To see why this matters, consider a model pretrained with sequences of 8 tokens. At inference time, we want to process 16 tokens (more than the training length). Positions 8-15 are out of distribution for the model's positional encodings.

要了解为什么这很重要，请考虑一个使用 8 个标记序列进行预训练的模型。在推理时，我们希望处理 16 个标记（超过训练长度）。位置 8-15 不适用于模型的位置编码。

While techniques like RoPE ABF address this by adjusting position frequencies, attention scope methods take a different approach: they strategically restrict which tokens can attend to each other, keeping attention patterns within familiar ranges while still processing the full sequence.

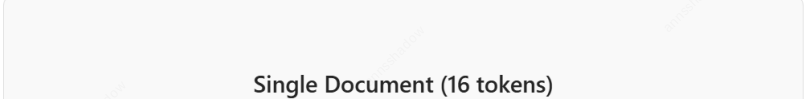
虽然像 RoPE ABF 这样的技术通过调整位置频率来解决这个问题，但注意力范围方法采取了不同的方法：它们战略性地限制哪些标记可以相互关注，将注意力模式保持在熟悉的范围内，同时仍处理整个序列。

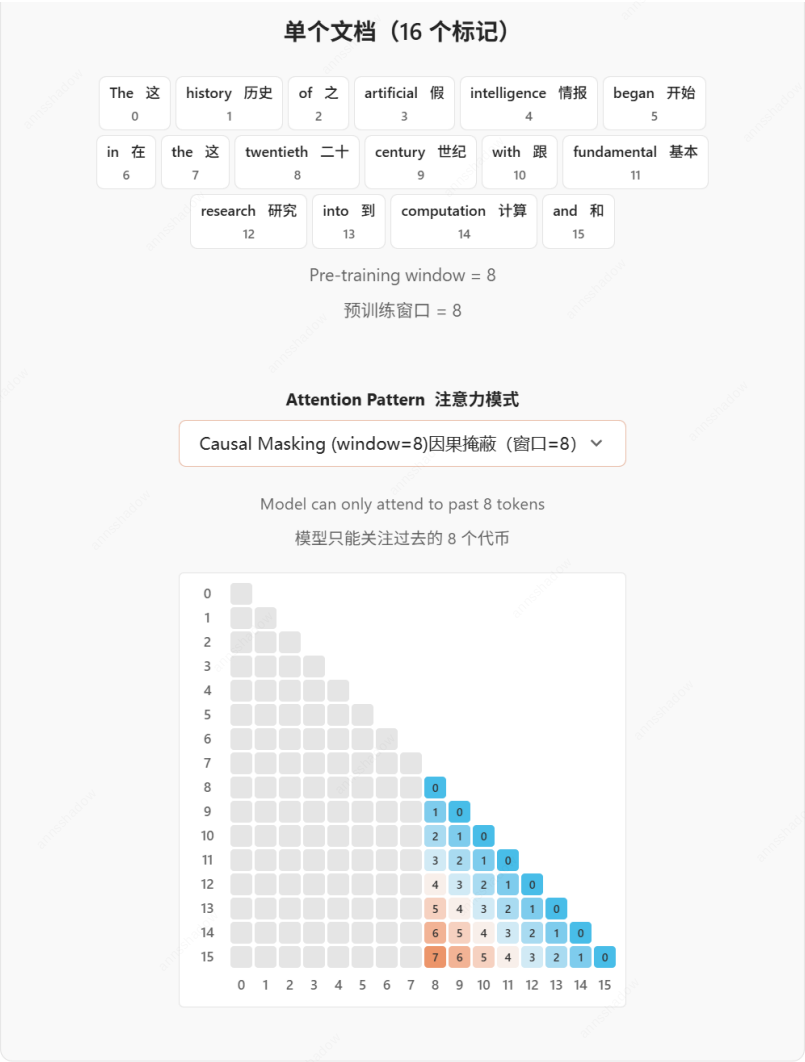
This reduces both computational cost and memory requirements.

这降低了计算成本和内存需求。

The diagram below compares five strategies for handling our 16-token sequence with a pretraining window of 8:

下图比较了处理 16 个标记序列的五种策略，预训练窗口为 8：





Chunked Attention divides the sequence into fixed size chunks, where tokens can only attend within their chunk. In our example, the 16 tokens are split into two 8 token chunks (0 to 7 and 8 to 15), and each token can only see others within its own chunk.

分块注意力将序列划分为固定大小的块，其中令牌只能在其块内参加。在我们的示例中，16 个令牌被分成两个 8 个令牌块（0 到 7 和 8 到 15），每个令牌只能在自己的块中看到其他令牌。

Notice how tokens 8 through 15 cannot attend back to the earlier chunk at all. This creates isolated attention windows that reset at chunk boundaries.

请注意，标记 8 到 15 根本无法处理较早的块。这会创建在块边界处重置的隔离注意力窗口。

Llama 4(Meta AI, 2025) uses chunked attention with 8192 token chunks in RoPE layers (three out of four decoder layers), while NoPE layers maintain full context access.

Llama 4（Meta AI，2025 年）在 RoPE 层中使用 8192 个代币块（解码器层中的四个中的三个）的分块注意力，而 NoPE 层保持完整的上下文访问。

This reduces memory requirements by limiting the KV cache size per layer, though it means tokens cannot attend to previous chunks, which may impact some long context tasks.

这通过限制每层的 KV 缓存大小来减少内存需求，尽管这意味着令牌无法处理以前的块，这可能会影响一些长上下文任务。

Sliding Window Attention (SWA) , popularized by Mistral 7B (Child et al., 2019; Jiang et al., 2023), takes a different approach based on the intuition that recent tokens are most relevant. Instead of hard chunk boundaries, each token attends only to the most recent N tokens.

滑动窗注意力 （SWA），由 Mistral 7B 推广（Child 等人，2019 年;江等人，2023 年），基于最近的代币最相关的直觉，采取了不同的方法。每个令牌不是硬块边界，而是只关注最近的 N 个令牌。

In the diagram, every token can see up to 8 positions back, creating a sliding window that moves continuously through the sequence. Notice how token 15 can attend to positions 8 through 15, while token 10 attends to positions 3 through 10.

在图中，每个标记最多可以看到 8 个位置，从而创建一个在序列中连续移动的滑动窗口。请注意，令牌 15 如何处理位置 8 到 15，而标记 10 处理位置 3 到 10。

The window slides forward, maintaining local context across the entire sequence without the artificial barriers of chunking. Gemma 3 combines SWA with full attention in alternating layers, similar to how hybrid positional encoding approaches mix

different strategies.

窗口向前滑动，在整个序列中保持局部上下文，而没有人为的分块障碍。Gemma 3 将 SWA 与交替层的充分关注相结合，类似于混合位置编码方法混合不同策略的方式。

Dual Chunk Attention (DCA) (An et al., 2024) is a training free method that extends **chunked attention** while maintaining cross chunk information flow. In our example, we use chunk size s=4, dividing the 16 tokens into 4 chunks (visualize 4×4 squares along the diagonal).

双块注意力 （DCA） （An et al.， 2024） 是一种免费训练的方法，在保持跨块信息流的同时扩展分块注意力。在我们的示例中，我们使用块大小 s=4，将 16 个标记分成 4 个块（沿对角线可视化 4×4 个正方形）。

DCA combines three mechanisms: (1) Intra chunk attention where tokens attend normally within their chunk (the diagonal pattern). (2) Inter chunk attention where queries use position index c−1=7 to attend to previous chunks, creating relative positions capped at 7.

DCA 结合了三种机制： （1） 块内注意力，其中代币在其块内正常出现（对角线模式）。 （2） 块间注意，其中查询使用位置索引 c−1=7 来关注前面的块，创建上限为 7 的相对位置。

(3) Successive chunk attention with local window w=3 that preserves locality between neighboring chunks. This keeps all relative positions within the training distribution (0 to 7) while maintaining smooth transitions across chunk boundaries. DCA enables models like Qwen2.

（3） 具有本地窗口 w=3 的连续块注意力，保留相邻块之间的局部性。这将使所有相对位置保持在训练分布（0 到 7）中，同时保持跨块边界的平滑过渡。DCA 支持像 Qwen2 这样的模型。

5 to support ultra-long context windows up to 1 million tokens at inference time, without requiring continual training on million-token sequences.

5 支持在推理时最多 100 万个标记的超长上下文窗口，而无需对百万个标记序列进行持续训练。

Attention Sinks 注意力汇

An interesting phenomenon emerges in transformer models with long contexts: the model assigns unusually high attention scores to the initial tokens in the sequence, even when these tokens aren't semantically important. This behavior is called **attention sinks** (Xiao et al.). These initial tokens act as a stabilisetion mechanism for the attention distribution, serving as a "sink" where attention can accumulate.

在具有长上下文的 Transformer 模型中出现了一个有趣的现象：该模型为序列中的初始标记分配异常高的注意力分数，即使这些标记在语义上并不重要。这种行为称为注意力下沉（Xiao 等人）。这些初始代币充当注意力分布的稳定机制，充当注意力积累的"水槽"。

The practical insight is that keeping the KV cache of just the initial few tokens alongside a sliding window of recent tokens largely recovers performance when context exceeds the cache size.

实用的见解是，当上下文超过缓存大小时，仅保留最初几个令牌的 KV 缓存以及最近令牌的滑动窗口，可以在很大程度上恢复性能。

This simple modification enables models to handle much longer sequences without fine-tuning or performance degradation.

这种简单的修改使模型能够处理更长的序列，而无需微调或降低性能。

Modern implementations leverage attention sinks in different ways. The original research suggests adding a dedicated placeholder token during pretraining to serve as an explicit attention sink. More recently, models like **gpt-oss** implement attention sinks as **learned per-head bias logits** that are appended to the attention scores rather than actual tokens in the input sequence. This approach achieves the same stabilisetion effect without modifying the tokenized inputs.

现代实现以不同的方式利用注意力吸收。最初的研究表明，在预训练期间添加一个专用的占位符标记，以作为显式注意力接收器。最近，像 gpt-oss 这样的模型将注意力汇实现为学习到的每人头偏差 logit，这些 logit 附加到注意力分数而不是输入序列中的实际标记。这种方法在不修改标记化输入的情况下实现了相同的稳定效果。

Interestingly, gpt-oss also uses bias units in the attention layers themselves, a design choice rarely seen since GPT-2. While these bias units are generally considered redundant for standard attention operations (empirical results from Dehghani et al. show minimal impact on test loss), they can serve the specialized function of implementing attention sinks.

有趣的是，gpt-oss 还在注意力层本身中使用了偏置单元，这是自 GPT-2 以来很少见的设计选择。虽然这些偏差单元通常被认为是标准注意力作的冗余（Dehghani 等人的经验结果表明对测试损失的影响很小），但它们可以发挥实现注意力吸收的特殊功能。

The key insight: whether implemented as special tokens, learned biases, or per-head logits, attention sinks provide a stable "anchor" for attention distributions in long-context scenarios, allowing the model to store generally useful information about the entire sequence even as context grows arbitrarily long.

关键见解：无论是作为特殊标记、学习偏差还是每人头 logit 实现，注意力汇都为长上下文场景中的注意力分布提供了稳定的"锚点"，允许模型存储有关整个序列的一般有用信息，即使上下文任意增长。

We've now covered the core components of attention: the different head configurations that balance memory and compute (MHA, GQA, MLA), the positional encoding strategies that help models understand token order (RoPE, NoPE, and their variants), and the attention scope techniques that make long contexts tractable (sliding windows, chunking, and attention sinks).

我们现在已经介绍了注意力的核心组件：平衡内存和计算的不同头部配置（MHA、GQA、MLA）、帮助模型理解令牌顺序的位置编码策略（RoPE、NoPE 及其变体），以及使长上下文易于处理的注意力范围技术（滑动窗口、分块和注意力接收器）。

We've also examined how embedding layers should be configured and initialized. These architectural choices define how your model processes and represents sequences.

我们还研究了如何配置和初始化嵌入层。这些架构选择定义了模型如何处理和表示序列。

But having the right architecture is only half the battle. Even well-designed models can suffer from training instability, especially at scale. Let's look at techniques that help keep training stable.

但拥有正确的架构只是成功的一半。即使是设计良好的模型也可能受到训练不稳定性的影响，尤其是在大规模情况下。让我们看看有助于保持训练稳定的技术。

keep training stable.

但拥有正确的架构只是成功的一半。即使是设计良好的模型也可能受到训练不稳定性的影响，尤其是在大规模情况下。让我们看看有助于保持训练稳定的技术。

IMPROVING STABILITY 提高稳定性

Let's now turn to one of the biggest challenges in LLM pretraining: instabilities. Often manifesting as loss spikes or sudden jumps in training loss, these issues become especially common at scale.

现在让我们转向 LLM 预训练中最大的挑战之一：不稳定性。这些问题通常表现为训练损失的损失峰值或突然跳跃，在大规模上变得尤为常见。

While we'll dive deeper into the different types of spikes and how to handle them in the Training Marathon section (diving in floating point precision, optimizers and learning rate), certain architectural and training techniques can also help us reduce instability so let's take a moment to study them here.

虽然我们将在训练马拉松部分（浮点精度、优化器和学习率）中更深入地探讨不同类型的尖峰以及如何处理它们，但某些架构和训练技术也可以帮助我们减少不稳定性，所以让我们花点时间在这里研究它们。

We'll cover a few simple techniques used in recent large-scale training runs (e.g., Olmo2(OLMo et al., 2025) and Qwen3 (A. Yang, Li, et al., 2025)) to improve stability: Z-loss, removing weight decay from embeddings, and QK-norm.

我们将介绍最近大规模训练运行中使用的一些简单技术（例如，Olmo2（OLMo 等人，2025 年）和 Qwen3（A. Yang， Li 等人，2025 年））以提高稳定性：Z 损失、从嵌入中去除权重衰减和 QK 范数。

Z-loss Z 损耗

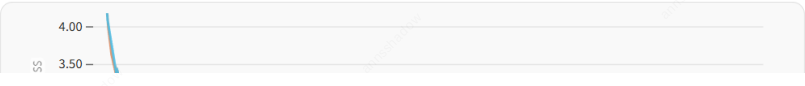
Z-loss (Chowdhery et al., 2022) is a regularisation technique that prevents the final output logits from growing too large by adding a penalty term to the loss function.

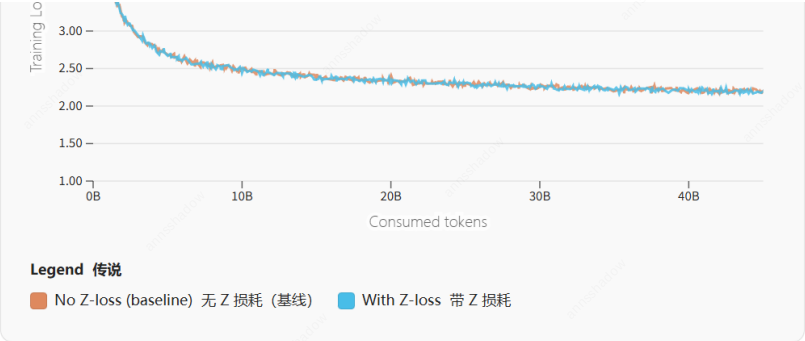
Z 损失（Chowdhery 等人，2022 年）是一种正则化技术，它通过向损失函数添加惩罚项来防止最终输出对数变得太大。

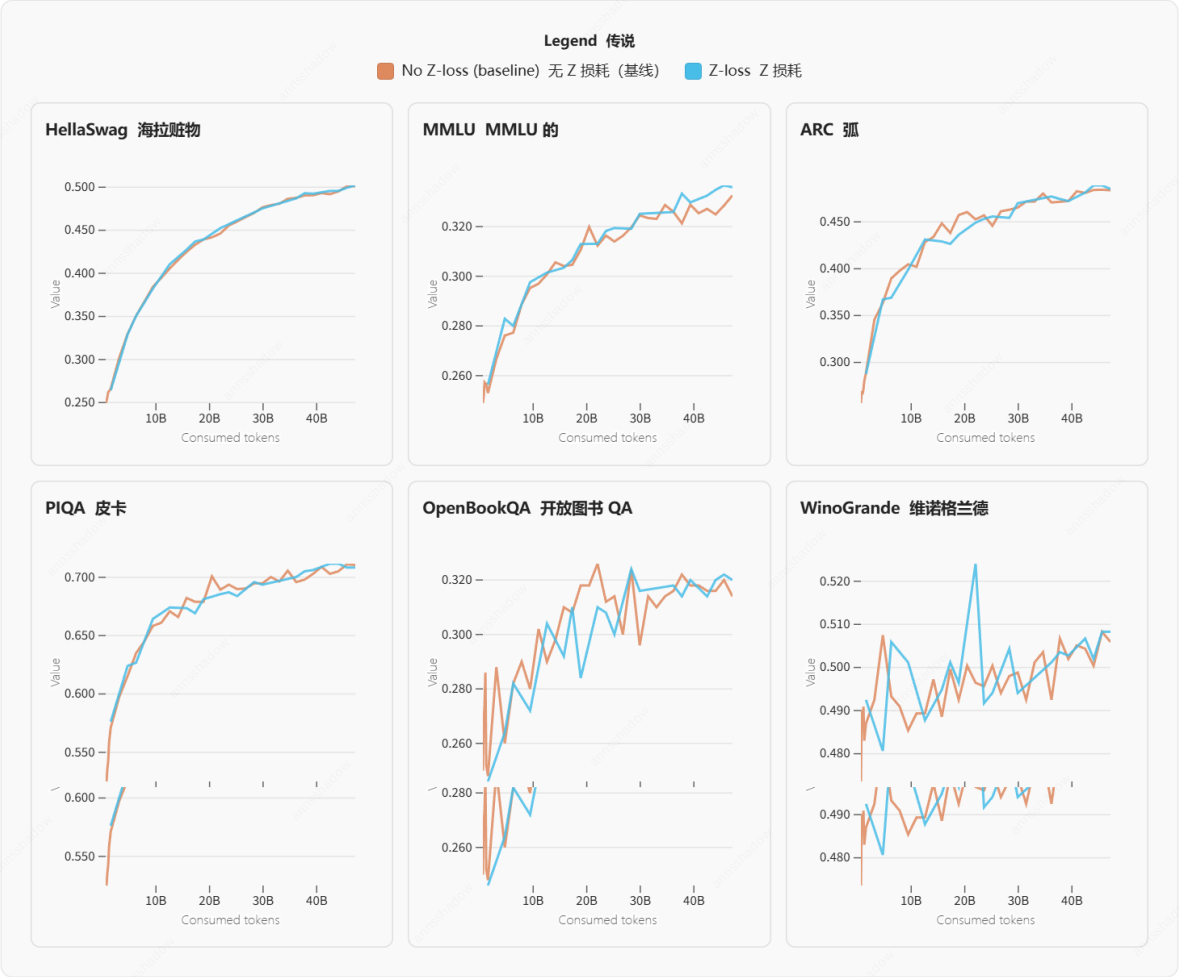
The regularisation encourages the denominator of the softmax over the logits to stay within a reasonable range, which helps maintain numerical stability during training.

正则化鼓励 softmax 的分母在 logits 上保持在合理的范围内，这有助于在训练期间保持数值稳定性。

$$\mathcal{L}_{\text{z-loss}} = \lambda \cdot \log^2(Z)$$







The ablation results below on our 1B model show that adding Z-loss doesn't impact the training loss or downstream performance.

下面在我们的 1B 模型上的消融结果表明，添加 Z 损失不会影响训练损失或下游性能。

For SmolLM3, we ended up not using it because our Z-loss implementation introduced some training overhead that we didn't optimize by the time we started training.

对于 SmolLM3，我们最终没有使用它，因为我们的 Z 损失实现引入了一些训练开销，我们在开始训练时没有优化这些开销。

Removing weight decay from embeddings

消除嵌入中的权重衰减

Weight decay is commonly applied to all model parameters as a regularization technique, but OLMo et al. (2025) found that excluding embeddings from weight decay improves training stability.

权重衰减通常作为正则化技术应用于所有模型参数，但 OLMo 等人（2025）发现，从权重衰减中排除嵌入可以提高训练稳定性。

The reasoning is that weight decay causes embedding norms to gradually decrease during training, which can lead to larger gradients in early layers since the Jacobian of layer normalization is inversely proportional to the input norm(Takase et al., 2025).

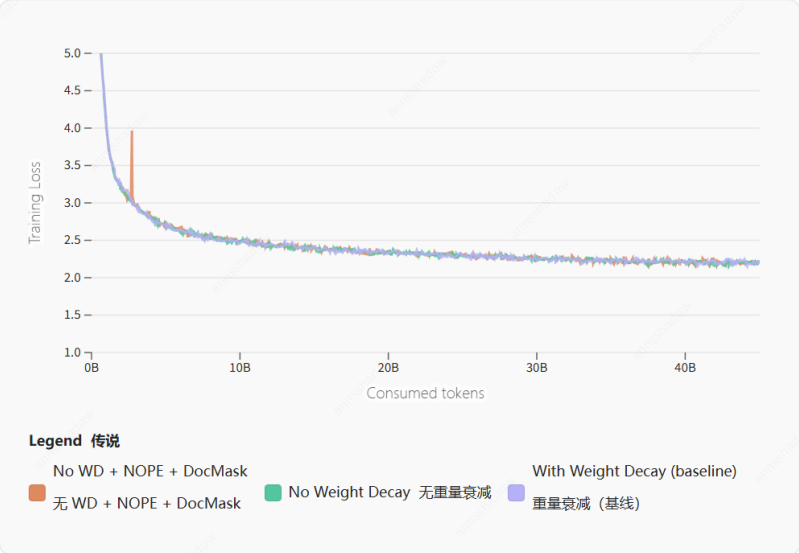
其理由是，权重衰减导致嵌入范数在训练过程中逐渐降低，这可能导致早期层的梯度更大，因为层归一化的雅可比与输入范数成反比（Takase 等人，2025 年）。

We tested this approach by training three configurations: our baseline with standard weight decay, a variant with no weight decay on embeddings, and a third configuration combining all our adopted changes (no weight decay on embeddings + NoPE + document masking) to ensure no negative interactions between techniques.

我们通过训练三种配置来测试这种方法：我们的基线具有标准权重衰减，一个在嵌入上没有权重衰减的变体，以及结合我们采用的所有更改（嵌入时没有权重衰减 + NoPE + 文档屏蔽）的第三种配置，以确保技术之间没有负面交互。

The loss curves and evaluation results were nearly identical across all three.
So we adopted all 3 changes in SmolLM3 training.

因此，我们在 SmolLM3 训练中采用了所有 3 项更改。



QKnorm QK 规范

QK-norm (Dehghani et al., 2023) applies layer normalization to both the query and key vectors before computing attention. This technique helps prevent attention logits

from becoming too large and was used in many recent models to improve stability.

QK-norm（Dehghani 等人，2023 年）在计算注意力之前将层归一化应用于查询和关键向量。这种技术有助于防止注意力 logit 变得过大，并在许多最近的模型中被使用以提高稳定性。

However, B. Yang et al. (2025) found that QK-norm hurts long-context tasks. Their analysis revealed that QK-norm results in lower attention mass on relevant tokens (needles) and higher attention mass on irrelevant context.

然而，B. Yang 等人（2025 年）发现 QK 规范会损害长期上下文任务。他们的分析表明，QK 规范导致相关标记（针）上的注意力质量较低，而对不相关上下文的注意力质量较高。

They argue this occurs because the normalization operation removes magnitude information from the query-key dot product, which makes the attention logits closer in terms of magnitude. Due to this reason, we didn't use QK-norm in SmolLM3.

他们认为，发生这种情况是因为归一化作从查询键点积中删除了量级信息，这使得注意力 logit 在量级方面更接近。由于这个原因，我们没有在 SmolLM3 中使用 QK-norm。

Additionally, as a small 3B parameter model, it faces less risk of training instability compared to the larger models where QK-norm has proven most beneficial.

此外，作为一个小型 3B 参数模型，与 QK-norm 已被证明最有益的大型模型相比，它面临的训练不稳定风险较小。

OTHER CORE COMPONENTS 其他核心组件

Beyond the components we've covered, there are a couple other architectural decisions worth noting for completeness.

除了我们介绍的组件之外，为了完整性，还有其他一些架构决策值得注意。

To initialize parameters, modern models typically use truncated normal initialization (mean=0, std=0.02 or std=0.006) or initialization scheme like muP (G. Yang & Hu, 2022), for instance Cohere's Command A (Cohere et al., 2025). This could be another topic of ablations.

为了初始化参数，现代模型通常使用截断的正态初始化（mean=0、std=0.02 或 std=0.006）或 muP 等初始化方案（G. Yang 和 胡，2022），例如 Cohere 的命令 A（Cohere 等人，2025 年）。这可能是消融的另一个话题。

In terms of activation functions, SwiGLU has become a de facto standard in modern LLMs (except Gemma2 using GeGLU and nvidia using relu^2 (Nvidia et al., 2024; NVIDIA et al., 2025)), replacing older choices like ReLU or GELU.

在激活函数方面，SwiGLU 已成为现代 LLM 中的事实上的标准（除了使用 GeGLU 的 Gemma2 和使用 relu^2 的 nvidia（Nvidia et al.， 2024;NVIDIA 等人，2025 年）），取代了 ReLU 或 GELU 等旧选择。

At a broader scale, architectural layout choices also play a role in shaping model behavior. Although the total parameter count largely determines a language model's capacity, how those parameters are distributed across depth and width also matters. Petty et al. (2024) found that deeper models outperform equally sized wider ones on language-modeling and compositional tasks until the benefit saturates. This "deep-and-thin" strategy works well for sub-billion-parameter LLMs in MobileLLM ablations (Z. Liu et al., 2024), whereas wider models tend to offer faster inference thanks to greater parallelism. Modern architectures reflect this trade-off differently as noted in this blog post.

在更广泛的范围内，架构布局选择在塑造模型行为方面也发挥着作用。尽管总参数数量在很大程度上决定了语言模型的容量，但这些参数如何在深度和宽度上分布也很重要。Petty 等人（2024 年）发现，在语言建模和组合任务上，更深层次的模型优于同等大小的更宽的模型，直到收益饱和。这种"深而薄"的策略适用于 MobileLLM 消融中的十亿以下参数 LLM（Z. Liu 等人，2024 年），而由于更大的并行性，更宽的模型往往提供更快的推理。正如本博文中所述，现代架构以不同的方式反映了这种权衡。

We now covered the most important aspects of the dense transformer architecture worth optimizing for you training run. However, recently other architecture interventions that concern the model as a whole have emerged, namely MoE and hybrid models.

我们现在介绍了密集变压器架构中值得优化的最重要方面，以便进行训练运行。然而，最近出现了其他涉及整个模型的架构干预措施，即 MoE 和混合模型。

Let's have a look what they have to offer, starting with the MoEs.

让我们看看他们必须提供什么，从 MoE 开始。

GOING SPARSE: MOE 变得稀疏：萌

The intuition of *Mixture-of-Experts (MoEs)* is that we don't need the full model for every token prediction, similarly to how our brain activates different areas depending on the task at hand (e.g. the visual or motor cortex).

混合专家 （MoE） 的直觉是，我们不需要每个标记预测的完整模型，类似于我们的大脑如何根据手头的任务（例如视觉或运动皮层）激活不同的区域。

For an LLM this could mean that the parts that learned about coding syntax don't need to be used when the model performs a translation task. If we can do this well, it means we can save a lot of compute as we only need to run parts of the full model at inference time.

对于 LLM 来说，这可能意味着在模型执行翻译任务时不需要使用学习编码语法的部分。如果我们能做好这一点，这意味着我们可以节省大量计算，因为我们只需要在推理时运行完整模型的一部分。

On a technical level MoEs have a simple goal: grow total parameters without increasing the number of "active" parameters for each token.

在技术层面上，MoE 有一个简单的目标：在不增加每个代币的"活动"参数数量的情况下增加总参数。

Somewhat simplified the total parameters impact the total learning capacity of the model while the active parameters determine the training cost and inference speed.

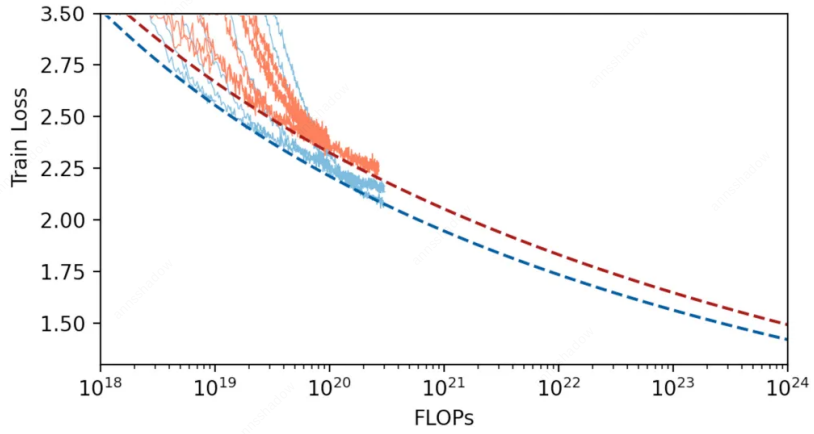
稍微简化的是，总参数会影响模型的总学习能力，而活动参数则决定训练成本和推理速度。

That's why you see many frontier systems (e.g., DeepSeek V3, K2, and in closed-source models like Gemini, Grok...) using MoE architectures these days. This plot from Ling 1.

这就是为什么你会看到现在许多前沿系统（例如 DeepSeek V3、K2 以及 Gemini、Grok 等闭源模型）使用 MoE 架构。这个情节来自灵 1.

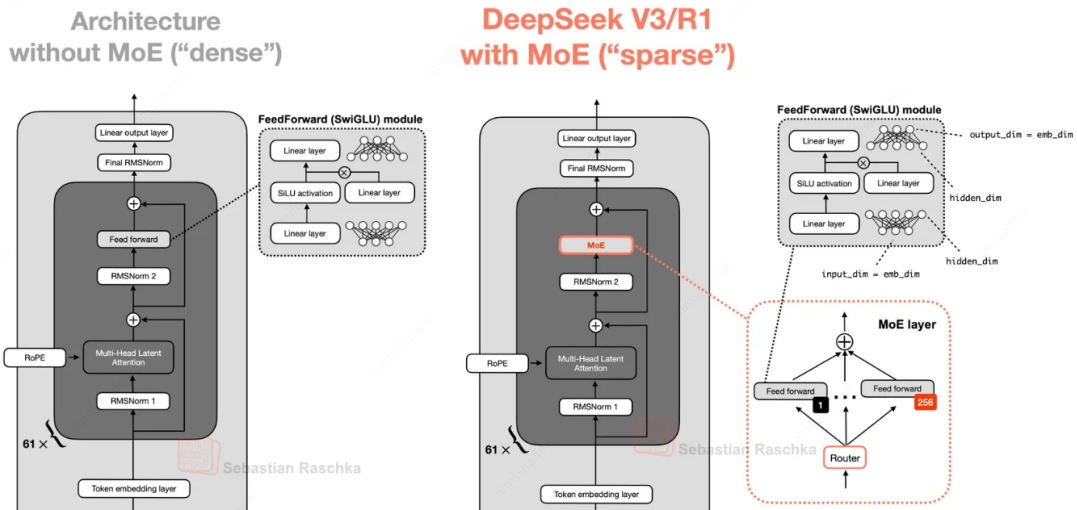
5 paper(L. Team et al., 2025) compares the scaling laws of MoE and dense models:

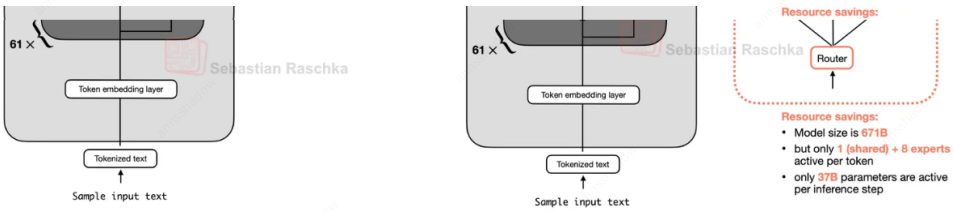
5 篇论文（ L. Team et al.， 2025）比较了 MoE 和密集模型的缩放规律：



If this is your first time encountering MoE, don't worry, the mechanics are not complicated. Let's start with the standard dense architecture and have a look at the necessary changes for the MoE (figure by Sebastian Raschka):

如果这是您第一次遇到 MoE，请不要担心，机制并不复杂。让我们从标准密集架构开始，看看 MoE 的必要更改（Sebastian Raschka 的图）：





With MoEs, we replace that single MLP with multiple MLPs (“experts”) and add a learnable router before the MLPs. For each token, the router selects a small subset of experts to execute.

使用 MoE，我们将单个 MLP 替换为多个 MLP（“专家”），并在 MLP 之前添加一个可学习的路由器。对于每个令牌，路由器会选择一小部分专家来执行。

This is where the distinction between total parameters and active parameters comes from: the model has many experts, but any given token only uses a few.

这就是总参数和活动参数之间的区别来源：模型有许多专家，但任何给定的代币都只使用少数专家。

Designing an MoE layer raises a few core questions:

设计 MoE 层会引发几个核心问题：

- Expert shape & sparsity: Should you use many small experts or fewer large ones? How many experts should be active per token, how many do you need in total

experts (i.e., the sparsity or “top-k”)? Should some experts be universal and thus always active?

专家形状和稀疏性：您应该使用许多小型专家还是更少的大型专家？每个代币应该有多少个专家活跃，总数需要多少个专家（即稀疏性或“top-k”）？一些专家是否应该具有普遍性，从而始终活跃？

- Utilization & specialization: How do you select the routed experts and keep them well-used (avoid idle capacity) while still encouraging them to specialize? In practice this is a load-balancing problem and has a significant impact on training and inference efficiency.

  利用率和专业化：您如何选择路由专家并保持他们得到充分利用（避免空闲容量），同时仍然鼓励他们专业化？在实践中，这是一个负载平衡问题，对训练和推理效率有重大影响。

Here we focus on one objective: given a fixed compute budget, how do we choose an MoE configuration that minimizes loss? That’s a different question from pure systems efficiency (throughput/latency), and we’ll come back to that later.

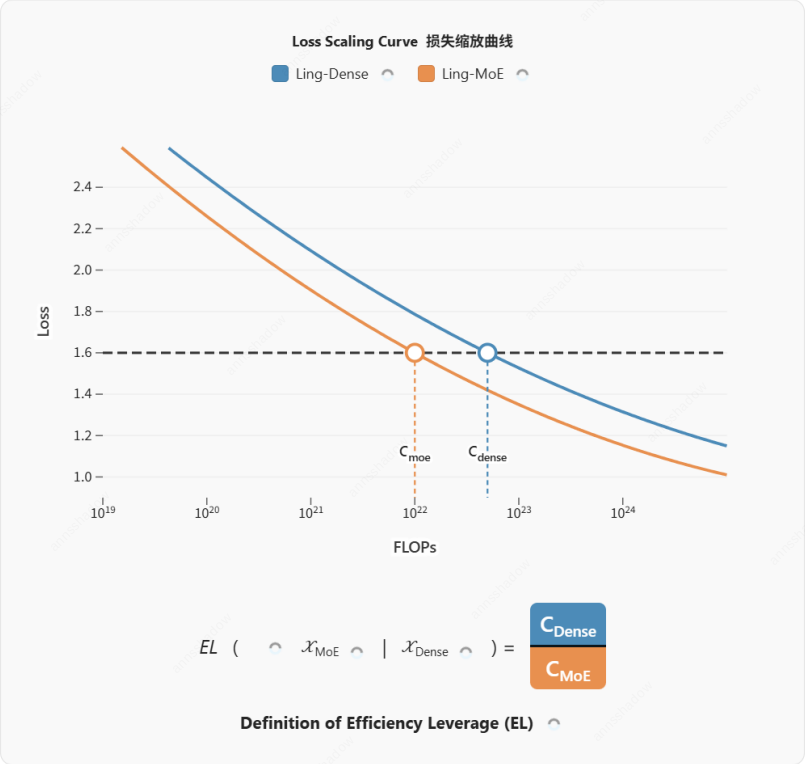
在这里，我们专注于一个目标：给定固定的计算预算，我们如何选择能够最大限度地减少损失的 MoE 配置？这与纯粹的系统效率（吞吐量/延迟）是一个不同的问题，我们稍后会回到这个问题。

Much of this section follows the analysis in Ant Group’s MoE scaling laws paper(Tian et al., 2025).

本节的大部分内容遵循蚂蚁集团的 MoE 缩放定律论文（Tian et al.， 2025）中的分析。

We’ll use their notion of *Efficiency Leverage (EL)* . Simply put, EL measures how much dense compute you’d need to match the loss achieved by an MoE design where the unit of measurement is FLOPs. A higher EL means the MoE configuration is delivering more loss improvement per unit of compute compared to dense training.

我们将使用他们的效率杠杆 （EL） 概念。简而言之，EL 衡量您需要多少密集计算才能匹配 MoE 设计所实现的损失，其中测量单位是 FLOP。与密集训练相比，更高的 EL 意味着 MoE 配置每单位计算的损失改善幅度更大。



Let’s have a closer look how we can setup the sparsity of the MoE to improve the efficiency leverage.

让我们仔细看看如何设置 MoE 的稀疏性以提高效率杠杆。

Sparsity / activation ratio

稀疏性/活化率

> **TL;DR:** more sparsity → better FLOPs efficiency → diminishing returns at very high sparsity → sweetspot depends on your compute budget.
>
> TL;DR：更高的稀疏性→更好的 FLOP 效率→在非常高的稀

疏性→最佳点下的收益递减取决于您的计算预算。

In this section we want to find out which MoE setting is best. Asymptotically it's easy to see that the two extremes are not ideal settings. On the one hand, activating all experts all the time brings us back to the dense setting where all parameters are used all the time.

在本节中，我们想找出哪种 MoE 设置最好。渐近地看，很容易看出这两个极端并不是理想的设置。一方面，一直激活所有专家会让我们回到一直使用所有参数的密集设置。

On the other hand if the active parameters are very low (as an extreme think just of just 1 parameter being active) clearly it won't be enough to solve a task even in a narrow domain. So clearly we need to find some middle ground.

另一方面，如果活动参数非常低（极端情况下，只需 1 个参数处于活动状态），那么即使在狭窄的领域中，显然也不足以解决任务。显然，我们需要找到一些中间立场。

Before we get deeper into finding the optimal setup it is useful to define two quantities:***activation ratio*** and its inverse the ***sparsity***

在我们更深入地寻找最佳设置之前，定义两个量是有用的：激活率及其反稀疏性：

$$\text{activation ratio} = \frac{\#\text{activated experts}}{\#\text{total experts}}$$

$$\text{sparsity} = \frac{\#\text{total experts}}{\#\text{activated experts}} = \frac{1}{\text{activation ratio}}$$

From a compute perspective the cost is driven by active parameters only.

从计算的角度来看，成本仅由活动参数驱动。

If you keep the number (and size) of activated experts fixed and increase the total number of experts, your inference/training FLOPs budget stays somewhat the same, but you're adding model capacity so the model should be generally better as long as you train long enough.

如果您保持激活专家的数量（和大小）不变并增加专家总数，您的推理/训练 FLOP 预算在某种程度上保持不变，但您正在增加模型容量，因此只要您训练足够长的时间，模型通常应该会更好。

There are some interesting empirical takeaways if you survey the recent MoE papers: Holding the number and size of active experts fixed, increasing the total number of experts (i.e., lowering activation ratio / increasing sparsity) improves loss, with diminishing returns once sparsity gets very high.

如果你调查最近的教育部论文，有一些有趣的实证要点：保持活跃专家的数量和规模是固定的，增加专家的总数（即降低激活率/增加稀疏性）会改善损失，一旦稀疏性变得非常高，回报就会递减。

Two examples: 两个例子：

- Kimi K2 plot (K. Team et al., 2025): shows both effects: higher sparsity improves performance, but gains taper off as sparsity grows.

  Kimi K2 图（K. Team 等人，2025 年）：显示了这两种效果：较高的稀疏性提高了性能，但随着稀疏性的增加，增益逐渐减弱。

- Ant Group plot (Tian et al., 2025): Same conclusion as K2, with additional result that higher sparsity MoE benefit more from increasing compute.

  Ant Group 图（Tian et al.， 2025）：与 K2 的结论相同，但额外的结果是，更高的稀疏度 MoE 从增加计算中获益更多。

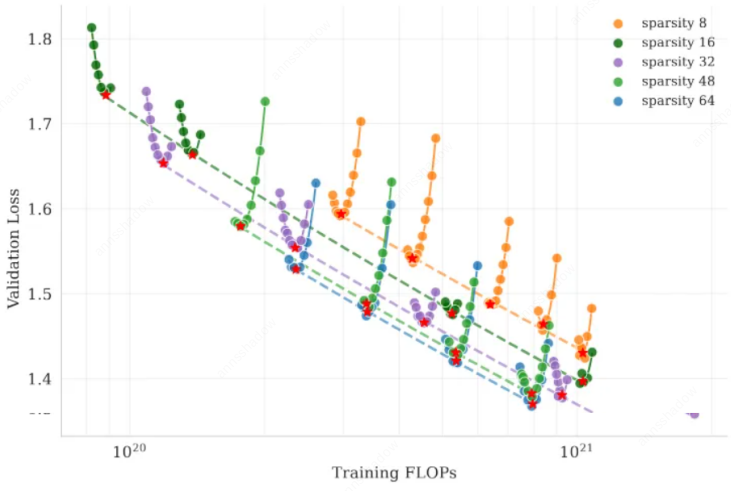
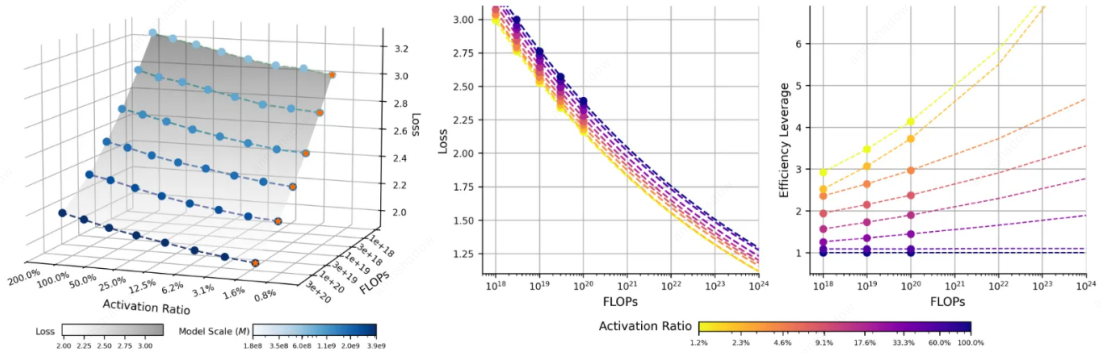


Figure 5: Sparsity Scaling Law. Increasing sparsity leads to improved model performance. We fixed the number of activated experts to 8 and the number of shared experts to 1, and varied the total number of experts, resulting in models with different sparsity levels.



(a) IsoFLOPs curves for varying A **(b)** Loss and EL scaling curve over varying A.

Figure 5 Impact of the Activation Ratio A on Loss and Efficiency. (a) At any fixed compute budget (each colored line), lower activation ratios yield lower loss. The orange stars mark the optimal (lowest) loss point. (b) Loss and EL scaling curves illustrate that EL increases with both higher compute budgets and lower activation ratios, showing that MoE advantages are magnified at scale.

Here is a table with the sparsity of some MoE model:

下面是一些 MoE 模型稀疏度的表格：

Model 型	Total experts 总专家	Activated per token (incl. shared) 按代币激活（包括共享）	Sparsity 稀疏
Mixtral-8×7B 混合-8×7B	8	2	4.0
Grok-1 格罗克-1	8	2	4.0
Grok-2 格罗克-2	8	2	4.0
OLMoE-1B-7B-0924	64	8	8.0
gpt-oss 20b GPT-作系统 20B	32	4	8
Step-3 步骤 3	48 routed + 1 shared = 49 48 个路由 + 1 个共享 = 49	3 routed + 1 shared = 4 3 个路由 + 1 个共享 = 4	12.25
GLM-4.5-Air GLM-4.5-空气	128 routed + 1 shared = 129 128 路由 + 1 共享 = 129	8 routed + 1 shared = 9 8 个路由 + 1 个共享 = 9	14.3
Qwen3-30B-A3B	128	8	16.0
Qwen3-235B-A22B	128	8	16.0
GLM-4.5	160 routed + 1 shared = 161 160 路由 + 1 共享 = 161	8 routed + 1 shared = 9 8 个路由 + 1 个共享 = 9	17.8
DeepSeek-V2	160 routed + 2 shared = 162 160 路由 + 2 共享 = 162	6 routed + 2 shared = 8 6 个路由 + 2 个共享 = 8	20.25
DeepSeek-V3	256 routed + 1 shared = 257 256 路由 + 1 共享 = 257	8 routed + 1 shared = 9 8 个路由 + 1 个共享 = 9	28.6
gpt-oss 120b GPT-作系统 120B	128	4	32
Kimi K2 基米 K2	384 routed + 1 shared = 385 384 路由 + 1 共享 = 385	8 routed + 1 shared = 9 8 个路由 + 1 个共享 = 9	42.8

Qwen3-Next-80B-A3B-Instruct	512 routed + 1 shared = 513 512 路由 + 1 共享 = 513	10 total active + 1 shared = 11 10 个活跃 + 1 个共享 = 11	46.6

The recent trend is clear: MoE models are getting sparser. That said, the optimal sparsity still depends on hardware and end-to-end efficiency.

For example, Step-3 targets peak efficiency and intentionally doesn't max out sparsity to fit their specific hardware and bandwidth constraints, while gpt-oss-20b have a low sparsity due to on-device memory constraints (the passive expert still take some memory).

例如，Step-3 以峰值效率为目标，并且故意不最大化稀疏性以适应其特定的硬件和带宽限制，而 gpt-oss-20b 由于设备上的内存限制而具有较低的稀疏性（被动专家仍然占用一些内存）。

Granularity 粒度

Beyond sparsity, we need to decide how large each expert should be. This is captured by granularity, a metric introduced by Ant Group. Let's pin down what we mean by this term. Terminology varies across papers, and some use slightly different formulas.

除了稀疏性之外，我们还需要决定每个专家应该有多大。这是通过粒度捕获的，粒度是蚂蚁集团引入的指标。让我们确定这个术语的含义。不同论文的术语各不相同，有些使用的公式略有不同。

Here, we'll use the definition that matches the plots we reference:

在这里，我们将使用与我们引用的图匹配的定义：

$$G = \frac{\alpha * d_{model}}{d_{expert}} \text{ with } \alpha = 2 \text{ or } 4$$

A higher granularity value corresponds to having more experts with smaller dimension (given a fixed number of parameters). This metric is a ratio between the expert dimension (d_{expert}) and the model dimension (d_{model}).

较高的粒度值对应于具有较小维度的更多专家（给定固定数量的参数）。该指标是专家维度 （ d_{expert} ） 和模型维度 （ d_{model} ） 之间的比率。

In dense models, a common rule of thumb is to have the dimension of the MLP set to $d_{intermediate} = 4 * d_{model}$. If $\alpha = 4$ (like Krajewski et al. (2024)). You can loosely view granularity as **how many experts it would take to match the dense MLP width** ($4\,d_{\text{model}} = d_{\text{intermediate}} = G\,d_{\text{expert}}$).

在密集模型中，常见的经验法则是将 MLP 的维度设置为 $d_{intermediate} = 4 * d_{model}$ 。如果 $\alpha = 4$ （如 Krajewski 等人 （2024））。您可以粗略地将粒度视为匹配密集 MLP 宽度 （ $4\,d_{\text{model}} = d_{\text{intermediate}} = G\,d_{\text{expert}}$ ） 所需的专家数量。

This interpretation is only a rough heuristic: modern MoE designs often allocate much larger total capacity than a single dense MLP, so the one-to-one match breaks down in practice. The Ant team setup choose $\alpha = 2$ which is simply a different normalization choice. For consistency we will pick this convention and stick to it.

Because G scales with d_{model}, cross-model comparisons can be tricky when model widths differ.

由于 G 与 缩 d_{model} 放，因此当模型宽度不同时，跨模型比

这种解释只是一种粗略的启发式方法：现代 MoE 设计通常分配的总容量比单个密集 MLP 大得

Still here is a table with the different value for some of the MoE release:

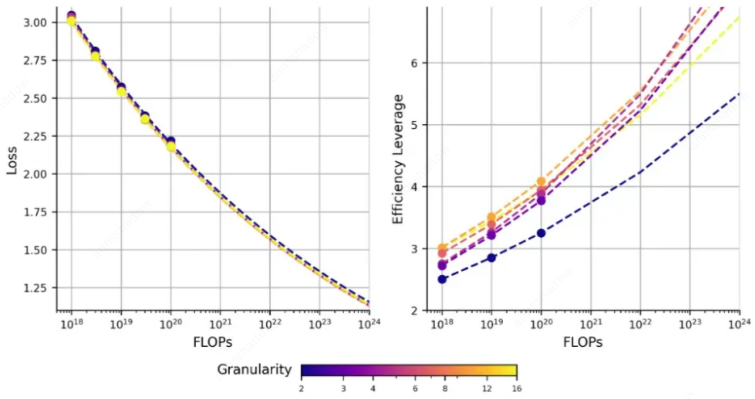
这里仍然是一个表，其中包含某些 MoE 版本的不同值：

Model 型	(d_{model})	(d_{expert})	$(G = 2d_{\text{model}}/d_{\text{expert}})$	Year 年
Mixtral-8×7B 混合-8×7B	4,096	14,336	0.571	2023
gpt-oss-120b	2880	2880	2.0	2025
gpt-oss-20b	2880	2880	2.0	2025
Grok 2 格罗克 2	8,192	16,384	1.0	2024
StepFun Step-3 StepFun 第 3 步	7,168	5,120	2.8	2025
OLMoE-1B-7B	2,048	1,024	4.0	2025
Qwen3-30B-A3B	2,048	768	5.3	2025
Qwen3-235B-A22B	4,096	1,536	5.3	2025
GLM-4.5-Air GLM-4.5-空气	4,096	1,408	5.8	2025
DeepSeek V2 深度搜索 V2	5,120	1,536	6.6	2024
GLM-4.5	5,120	1,536	6.6	2025
Kimi K2 基米 K2	7,168	2,048	7.0	2025

DeepSeek V3 深度搜索 V3	7168	2048	7.0	2024
Qwen3-Next-80B-A3B	2048	512	8.0	2025

Let's talk about how granularity shapes behavior (from Ant Group's paper):

我们来谈谈粒度如何塑造行为（来自蚂蚁集团的论文）：



(b) Loss and efficiency leverage scaling curve over varying G.

Granularity doesn't look like the primary driver of EL—it helps, especially going above 2, but it's not the dominant factor determining the loss. There's a sweet spot though: pushing granularity higher helps up to a point, and then gains flatten.

粒度看起来不像是 EL 的主要驱动因素——它有帮助，尤其是超过 2，但它不是决定损失的主要因素。不过有一个最佳点：将粒度推高到一定程度会有所帮助，然后收益会趋于平缓。

So granularity is a useful tuning knob with a clear trend toward higher values in recent releases, but it shouldn't be optimized in isolation.

因此，粒度是一个有用的调音旋钮，在最近的版本中具有明显的更高值趋势，但不应孤立地优化它。

Another method that is used widely to improve MoEs is the concept of shared experts. Let's have a look!

另一种广泛用于改进 MoE 的方法是共享专家的概念。让我们一起来看看吧！

Shared experts 共享专家

A shared-expert setup routes every token to a small set of always-on experts. These shared experts absorb the basic, recurring patterns in the data so the remaining experts can specialize more aggressively.

共享专家设置将每个令牌路由到一小部分始终在线的专家。这些共享专家吸收数据中的基本重复模式，以便其余专家可以更积极地进行专业化。

In practice, you usually don't need many of them; model designers commonly choose one, at most two. As granularity increases (e.g., moving from a Qwen3-style setting to something closer to Qwen3-Next), shared experts tend to become more useful.

在实践中，您通常不需要很多;模型设计师通常选择一个，最多两个。随着粒度的增加（例如，从 Qwen3 风格的设置移动到更接近 Qwen3-Next 的设置），共享专家往往会变得更有用。

Looking at the following plot, the overall impact is modest, it doesn't dramatically change the EL.

看下面的图，整体影响不大，并没有显着改变 EL。

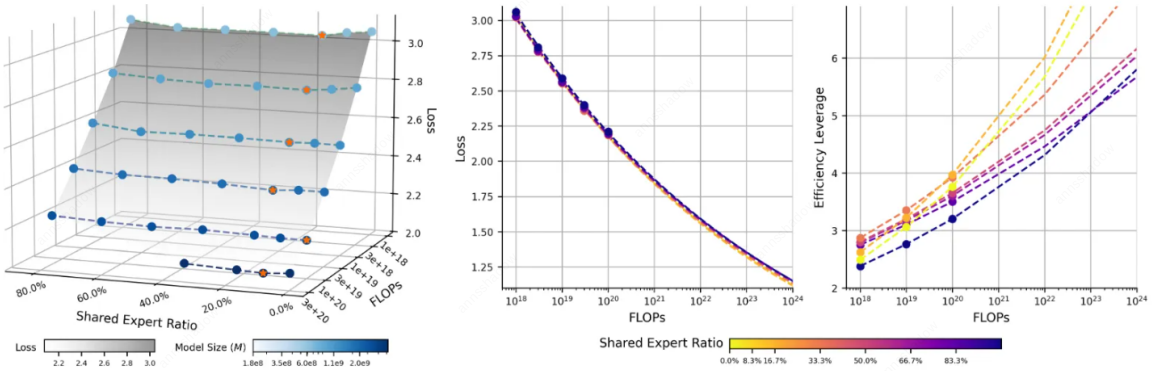
A simple rule of thumb works well in most cases: just use one shared expert, which matches choices in models like DeepSeek V3, K2, and Qwen3-Next and tends to maximize efficiency without adding unnecessary complexity.

A simple rule of thumb works well in most cases: just use one shared expert, which matches choices in models like DeepSeek V3, K2, and Qwen3-Next and tends to maximize efficiency without adding unnecessary complexity.

在大多数情况下，一个简单的经验法则效果很好：只需使用一个共享专家，它与 DeepSeek V3、K2 和 Qwen3-Next 等模型中的选择相匹配，并且往往会在不增加不必要的复杂性的情况下最大限度地提高效率。

Figure fromTian et al. (2025) .

图来自 Tian 等人 （2025）。



(a) IsoFLOPs curves over varying G. **(b)** Loss and efficiency leverage scaling curve over varying G.

So a shared expert is an expert where some tokens are always routed through. What about the other experts? How do we learn when to rout to each expert and make sure that we don't just use a handful of experts? Next we'll discuss load balancing which tackles exactly that problem.

因此，共享专家是一些代币始终通过的专家。其他专家呢？我们如何学习何时向每个专家溃败，并确保我们不会只使用少数专家？接下来，我们将讨论负载平衡，它正是解决了这个问题。

Load balancing 负载平衡

Load balancing is the critical piece in MoE. If it is set up poorly, it can undermine every other design choice. We can see why poor load balancing will cause us a lot of pain on the following example.

负载平衡是 MoE 中的关键部分。如果设置不当，可能会破坏所有其他设计选择。我们可以在下面的示例中看到为什么负载平衡不佳会给我们带来很多痛苦。

Consider a very simple distributed training setup where we have 4 GPUs and we distribute the 4 experts of our model evenly across the GPUs.

考虑一个非常简单的分布式训练设置，其中我们有 4 个 GPU，我们将模型的 4 个专家均匀分布在 GPU 上。

If the routing collapses and all tokens are routed to expert 1 this means that only 1/4 of our GPUs is utilized which is very bad for training and inference efficiency.

如果路由崩溃，所有令牌都被路由到专家 1，这意味着我们只有 1/4 的 GPU 被利用，这对训练和推理效率非常不利。

Besides that, it means that the effective learning capacity of our model also decreased as not all experts are activated.

除此之外，这意味着我们模型的有效学习能力也下降了，因为并非所有专家都被激活。

To address this issue we can we can add an extra loss term to the router. Below you can see the standard auxiliary loss–based load balancing (LBL):

为了解决这个问题，我们可以 我们可以 为路由器添加一个额外的损失项。下面是标准的基于损耗的辅助负载平衡 （LBL）：

$$\mathcal{L}_{\text{Bal}} = \alpha \sum_{i=1}^{N_r} f_i \, P_i$$

This simple formula just uses three factors: the coefficient α determines the strength of the loss, f_i is the traffic fraction so just the fraction of tokens going through expert i and finally P_i which is the probability mass and simply sums the probability of the tokens going through the expert. They are both necessary, f_i correspond to the actual balancing, while P_i is smooth and differentiable allowing the gradient to flows. If we achieve perfect load balancing we get $f_i = P_i = 1/N_r$, however we need to be careful how we tune α as a value too small we don't guide routing enough and if it's too big routing uniformity becomes more important than the primary language model loss.

这个简单的公式只使用三个因素：系数 α 决定了损失的强度， f_i 是流量分数，所以只是通过专家的代币的分数 i ，最后 P_i 是概率质量，简单地将代币通过专家的概率相加。它们都是必要的， f_i 对应于实际的平衡，同时 P_i 平滑且可微分，允许梯度流动。如果我们实现完美的负载平衡，我们会得到 $f_i = P_i = 1/N_r$ ，但是我们需要小心如何调整， α 因为值太小，我们无法充分指导路由，如果它太大，路由均匀性就变得比主要语言模型损失更重要。

> It is also possible to achieve balancing without an explicit loss term. DeepSeek v3

(DeepSeek-AI et al., 2025) introduced a simple bias term added to the affinity scores that go into the routing softmax. If a router is overloaded they decrease the score a bit (a constant factor γ) thus making it less likely to be selected and increase it by γ if the expert is underutilized. With this simple adaptive rule they also achieve load balancing.

也可以在没有明确损失项的情况下实现平衡。DeepSeek v3（DeepSeek-AI 等人，2025 年）引入了一个简单的偏差项，添加到路由软 max 的亲和力分数中。如果路由器过载，它们会稍微降低分数（一个恒定因素 γ），从而降低选择分数的可能性，γ 如果专家未得到充分利用，则增加分数。通过这个简单的自适应规则，它们还可以实现负载平衡。

A key detail is the scope at which you compute routing statistics: are f_i and P_i computed per local batch (each worker's mini-batch) or globally (aggregated across workers/devices)? Qwen team's analysis (Qiu et al., 2025) shows that when there isn't enough token diversity in each local batch and that local computation can hurt both expert specialization (a good proxy for routing health) and overall model performance.

一个关键细节是计算路由统计信息的范围：是 f_i P_i 按本地批处理（每个辅助角色的迷你批处理）还是全局（跨辅助角色/设备聚合）计算？Qwen 团队的分析（Qiu 等人，2025 年）表明，当每个本地批次中没有足够的令牌多样性时，本地计算会损害专家专业化（路由健康状况的良好代理）和整体模型性能。

Expert specialization is the phenomenon where one or more experts are activated more often than others for a specific domain. In other words, if a local batch is narrow, its routing stats become noisy/biased, and don't lead to good balancing.

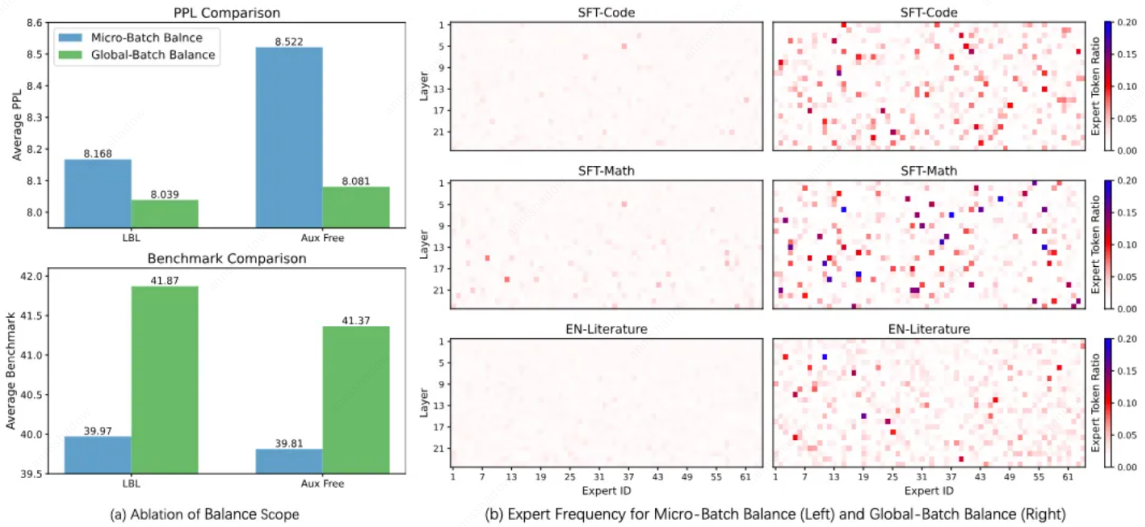
专家专业化是指在特定领域中，一个或多个专家比其他专家更频繁地被激活的现象。换句话说，如果本地批次很窄，它的路由统计信息就会变得嘈杂/有偏差，并且不会导致良好的平衡。

This implies that we should use global statistics (or at least cross-device aggregation) whenever feasible. Notably, at the time of that paper, many frameworks—including Megatron—computed these statistics locally by default.

这意味着我们应该在可行的情况下使用全局统计（或至少跨设备聚合）。值得注意的是，在撰写该论文时，许多框架（包括威震天）默认在本地计算这些统计数据。

The following plot from Qwen's paper illustrates the difference of micro-batch vs global batch aggregation and it's impact on performance and specialization:

Qwen 论文中的下图说明了微批次聚合与全局批次聚合的差异及其对性能和专业化的影响：



(a) Ablation of Balance Scope

(b) Expert Frequency for Micro-Batch Balance (Left) and Global-Batch Balance (Right)

Generally, ablating architecture choices around MoE is tricky as there is an interplay with many aspects. For example the usefulness of a shared expert might depend on the granularity of the model.

通常，围绕 MoE 消除架构选择是很棘手的，因为许多方面都存在相互作用。例如，共享专家的有用性可能取决于模型的粒度。

So it's worth spending some time to make sure you have a good set of experiments to really get the insights you are looking for!

因此，值得花一些时间来确保您有一套好的实验，以真正获得您正在寻找的见解！

We have now covered the fundamentals of MoEs, however there is still more to

discover. A non-exhaustive list of items to further study:

我们现在已经介绍了 MoE 的基础知识，但还有更多内容有待发现。需要进一步研究的非详尽项目列表：

- Zero-computation experts, MoE layer rescaling and training monitoring (LongCat-Flash paper).

  零计算专家、MoE 层重缩和训练监控（LongCat-Flash 论文）。

- Orthogonal loss load balancing (as in ERNIE 4.5).

  零计算专家、MoE 层重缩和训练监控（LongCat-Flash 论文）。

- Orthogonal loss load balancing (as in ERNIE 4.5).

  正交损耗负载平衡（如 ERNIE 4.5 中）。

- Scheduling the load-balancing coefficient over training.

  调度训练的负载平衡系数。

- Architecture/optimization interactions with MoE, like:

  与 MoE 的架构/优化交互，例如：

  - Whether optimizer rankings change for MoE.

    优化器排名是否会更改 MoE.

  - How to apply MuP to MoE.

    如何将 MuP 应用于 MoE。

  - How to adapt the learning rate for MoE (since they don't see the same number of token per batch).

    如何调整 MoE 的学习率（因为他们不会看到每个批次的令牌数量相同）。

- Number of dense layers at the start.

  开始时密集层数。

- Many more.. 还有很多..

We leave it up to you, eager reader, to follow the rabbit whole further down, while we now move on to the last major architecture choice: hybrid models!

我们留给你，热切的读者，跟随兔子更进一步，而我们现在继续讨论最后一个主要架构选择：混合模型！

EXCURSION: HYBRID MODELS
游览：混合动力车型

A recent trend is to augment the standard dense or MoE architecture with state space models (SSM) or linear attention mechanisms (MiniMax et al., 2025; Zuo et al., 2025). ** These new classes of models try to address some of the fundamental weaknesses of transformers: dealing efficiently with very long context.

最近的趋势是通过状态空间模型（SSM）或线性注意力机制来增强标准密集或 MoE 架构（MiniMax 等人，2025 年;Zuo 等人，2025 年）。** 这些新类别的模型试图解决 Transformer 的一些基本弱点：有效地处理很长的上下文。

They take a middle ground between recurrent models, that can efficiently deal with arbitrary length context and scale linearly, but might suffer to keep utilize the information in context and transformers that get very expensive with long context but can leverage the patterns in context very well.

它们在循环模型之间采取了中间立场，循环模型可以有效地处理任意长度的上下文并线性扩展，但可能会在上下文中继续利用信息和转换器之间，转换器在长上下文中变得非常昂贵，但可以很好地利用上下文中的模式。

There have been some studies (Waleffe et al., 2024) to understand the weaknesses for example of Mamba models (a form of SSM) and found that such models perform well on many benchmarks but for example underperform on MMLU and hypothesize that it's the lack of in-context learning causing the gap.

已经有一些研究（Waleffe 等人，2024 年）来了解 Mamba 模型（SSM 的一种形式）的弱点，并发现此类模型在许多基准测试中表现良好，但在 MMLU 上表现不佳，并假设是缺乏上下文学习导致了差距。

That's the reason that they are combined with blocks from dense or MoE models to get the best of both worlds, thus the name hybrid models.

这就是为什么它们与密集或 MoE 模型的块相结合以获得两全其美的原因，因此得名混合模

型。

The core idea behind these linear attention methods is to reorder computations so attention no longer costs O(n^2d), which becomes intractable at long context. How does that work? First, recall the attention formulation at inference. Producing the output for token t looks like:

这些线性注意力方法背后的核心思想是重新排序计算，以便注意力不再花费 O（n^2d），这在长上下文中变得棘手。这是如何工作的？首先，回想一下推理时的注意力公式。生成标记 t 的输出如下所示：

$$\mathbf{o}_t = \sum_{j=1}^{t} \frac{\exp(\mathbf{q}_t^\top \mathbf{k}_j)\mathbf{v}_j}{\sum_{l=1}^{t} \exp(\mathbf{q}_t^\top \mathbf{k}_l)}$$

Now drop the softmax:

现在删除 softmax：

$$o_t = \sum_{j=1}^{t} (q_t^\top k_j)\, v_j$$

Reordering gives: 重新排序可提供：

$$\sum_{j=1}^{t} (q_t^\top k_j)\, v_j = \Big(\sum_{j=1}^{t} v_j k_j^\top\Big) q_t.$$

Define the running state:

定义运行状态：

$$S_t \triangleq \sum_{j=1}^{t} k_j v_j^\top = K_{1:t}^\top V_{1:t} \in \mathbb{R}^{d\times d}$$

with the simple update:

通过简单的更新：

$$S_t = S_{t-1} + k_t v_t^\top$$

So we can write:

所以我们可以写：

$$o_t = S_t q_t \quad = S_{t-1} q_t + v_t (k_t^\top q_t)$$

Why is the reordering important: the left form $\sum_{j\le t}(q_t^\top k_j)v_j$ means "for each past token j, take a dot $q_t^\top k_j$ (a scalar), use it to scale v_j, and add those t vectors up"—that's about $O(td)$ work at step t. The right form rewrites this as $\big(\sum_{j\le t} v_j k_j^\top\big) q_t$: you keep a single running state matrix $S_t = \sum_{j\le t} v_j k_j^\top \in \mathbb{R}^{d\times d}$ that already summarizes all past (k_j, v_j). Each new token updates it with one outer product $v_t k_t^\top$ cost $O(d^2)$, then the output is just one matrix–vector multiply $S_t q_t$ (another $O(d^2)$). So generating T tokens by the left form from scratch is $O(T^2 d)$, while maintaining S_t and using the right form is $O(Td^2)$. Intuitively: left = "many small dot-scale-adds each step"; right = "one pre-summarized matrix times the query," trading dependence on sequence length for dependence on dimension.

为什么重新排序很重要：左侧形式 $\sum_{j\le t}(q_t^\top k_j)v_j$ 表示"对于每个过去的标记 j，取一个点 $q_t^\top k_j$（标量），用它来缩放 v_j，然后将这些 t 向量相加"——这是关于步骤 t 的工作。$O(td)$ 正确的形式将其重写为 $\big(\sum_{j\le t} v_j k_j^\top\big) q_t$：您保留一个运行状态矩阵 $S_t = \sum_{j\le t} v_j k_j^\top \in \mathbb{R}^{d\times d}$，该矩阵已经总结了过去 (k_j, v_j) 的所有 。每个新标记都用一个外部乘积 $v_t k_t^\top$ 成本 $O(d^2)$ 更新它，然后输出只是一个矩阵向量乘法 $S_t q_t$（另一个 $O(d^2)$）。因此，通过左形式从头开始生成 T 标记是 $O(T^2 d)$，而保持 S_t 和使用右形式是 $O(Td^2)$。直观地说：left = "每个步骤都有很多小点比例加";right = "一个预先汇总的矩阵乘以查询"，将对序列长度的依赖性换取对维度的依赖性。

We here focus on inference and recurring form, but it's also more efficient in training, where the reordering is as simple as the following equation:

我们在这里专注于推理和重复形式，但它在训练中也更有效，其中重新排序就像以下等式一样

$$\underset{n\times n}{(QK^\top)}\, V \;=\; Q\,\underset{d\times d}{(K^\top V)}$$

So we can see that this looks now very similar to an RNN like structure. This solved our issue, right? Almost. In practice, the softmax plays an important stabilizing role, and the naïve linear form can be unstable without some normalization.

因此，我们可以看到，这现在看起来与类似 RNN 的结构非常相似。这解决了我们的问题，对吧？几乎。在实践中，softmax 起着重要的稳定作用，如果没有一些归一化，朴素线性形式可能会不稳定。

This motivates a practical variant called lightning or norm attention!

这激发了一种称为闪电或常态注意力的实用变体！

Lightning and norm attention

闪电和规范注意力

This family appears in Minimax01 (MiniMax et al., 2025) and, more recently, in Ring-linear (L. Team, Han, et al., 2025). Building on the Norm Attention idea (Qin et al., 2022). The key step is simple: **normalize the output** . The "Lightning" variant focuses on making the implementation fast and efficient and make the formula a bit different. Here is the formula for both:

该家族出现在 Minimax01 （MiniMax et al.， 2025） 中，最近出现在环线性 （L. Team， Han， et al.， 2025） 中。建立在常态注意力思想的基础上（Qin et al.， 2022）。关键步骤很简单：规范化输出。"闪电"变体侧重于使实现快速高效，并使公式略有不同。以下是两者的公式：

NormAttention: 规范注意：

$$\text{RMSNorm}(Q(K^TV))$$

LightningAttention: 闪电注意：

$$Q = \text{Silu(Q)},\ K = \text{Silu(K)},\ V = \text{Silu(V)}$$

$$O = \text{SRMSNorm}(Q(KV^T))$$

Empirically, hybrid model with Norm attention match softmax on most of the tasks according to Minimax01.

根据经验，根据 Minimax01，具有 Norm 注意力的混合模型在大多数任务上都与 softmax 匹配。

What's interesting here is that on retrieval tasks like Needle in a Haystack (NIAH) it can do much much better than full softmax attention, which seems surprising but might indicate that there is some synergy when the softmax and the linear layer work

together!

> **MiniMax M2 迷你 M2**
>
> Surprisingly, the recently released MiniMax M2 does not use hybrid or linear attention. According to their pretraining lead, while their early MiniMax M1 experiments with Lightning Attention looked promising at smaller scales on the popular benchmarks at the time (MMLU, BBH, MATH), they found it had “clear deficits in complex, multi-hop reasoning tasks” at larger scales.
>
> 令人惊讶的是，最近发布的 MiniMax M2 没有使用混合或线性注意力。根据他们的预训练负责人，虽然他们早期的 MiniMax M1 Lightning Attention 实验在当时流行的基准测试（MMLU、BBH、MATH）上在较小的尺度上看起来很有希望，但他们发现它在更大的尺度上“在复杂的多跳推理任务中存在明显的缺陷”。
>
> They also cite numerical precision issues during RL training and infrastructure maturity as key blockers.
>
> 他们还将 RL 训练期间的数值精度问题和基础设施成熟度列为关键障碍。
>
> They conclude that making architecture at scale is a multivariable problem that is hard and compute intensive due to the sensitivity to other parameters like data distribution, optimizer...
>
> 他们得出的结论是，大规模构建架构是一个多变量问题，由于对其他参数（如数据分布、优化器）的敏感性，因此很难且计算密集型.....
>
> However, they acknowledge that “as GPU compute growth slows while data length keeps increasing, the benefits of linear and sparse attention will gradually emerge.”

This highlights both the complexity of architecture ablations and the gap between research and production reality.

然而，他们承认，“随着 GPU 计算增长放缓而数据长度不断增加，线性和稀疏注意力的好处将逐渐显现。这凸显了建筑烧蚀的复杂性以及研究与生产现实之间的差距。

Now let's have a look at some more of these methods and how they can be understood with a unified framework.

现在让我们再看看其中的一些方法，以及如何通过统一的框架来理解它们。

Advanced linear attention

高级线性注意力

A helpful lesson from recurrent models is to let the state occasionally let go of the past. In practice, that means introducing a gate $\mathbf{G}_t$ for the previous state:

从反复出现的模式中吸取的一个有益教训是让国家偶尔放下过去。在实践中，这意味着 $\mathbf{G}_t$ 为前一个状态引入一个门：

$$\mathbf{S}_t \;=\; \mathbf{G}_t \odot \mathbf{S}_{t-1} \;+\; \mathbf{v}_t \mathbf{k}_t^\top$$

Almost all recent linear attention methods have this gating component with just different implementations of $\mathbf{G}_t$. Here is a list of the different variants for the gate and the corresponding architecture from this paper:

几乎所有最近的线性注意力方法都有这个门控组件，只是 $\mathbf{G}_t$ 实现了不同的 。以下是本文中门的不同变体和相应架构的列表：

Model 型	Parameterization 参数化
Mamba (A. Gu & Dao, 2024) 曼巴（A. Gu & Dao，2024 年）	$\mathbf{G}_t = \exp(-(\mathbf{1}^\top \boldsymbol{\alpha}_t) \odot \exp(\mathbf{A})), \quad \boldsymbol{\alpha} t = \text{softplus}(\mathbf{x} t \mathbf{W} \alpha_1 \mathbf{W} \alpha_2)$
Mamba-2 (Dao & Gu, 2024) 曼巴二号（Dao & Gu，2024）	$\mathbf{G}_t = \gamma_t \mathbf{1}^\top \mathbf{1}, \quad \gamma_t = \exp(-\text{softplus}(\mathbf{x} t \mathbf{W} \gamma) \exp(a))$
mLSTM (Beck et al., 2025; H. Peng et al., 2021) mLSTM（Beck 等人，2025 年;H. Peng 等人，2021 年）	$\mathbf{G}_t = \gamma_t \mathbf{1}^\top \mathbf{1}, \quad \gamma_t = \sigma(\mathbf{x} t \mathbf{W} \gamma)$
Gated Retention (Sun et al., 2024) 门控保留（Sun 等人，2024 年）	$\mathbf{G}_t = \gamma_t \mathbf{1}^\top \mathbf{1}, \quad \gamma_t = \sigma(\mathbf{x} t \mathbf{W} \gamma)^{\frac{1}{\tau}}$
DFW (Mao, 2022; Pramanik et al., 2023) (Mao, 2022) DFW（毛，2022 年;Pramanik 等人，2023 年）（毛，2022 年）	$\mathbf{G}_t = \boldsymbol{\alpha}_t^\top \boldsymbol{\beta}_t, \quad \boldsymbol{\alpha}_t = \sigma(\mathbf{x} t \mathbf{W} \alpha), \quad \boldsymbol{\beta}_t = \sigma(\mathbf{x} t \mathbf{W} \beta)$
GateLoop (Katsch, 2024) GateLoop（卡奇，2024 年）	$\mathbf{G}_t = \boldsymbol{\alpha}_t^\top \mathbf{1}, \quad \boldsymbol{\alpha}_t = \sigma(\mathbf{x} t \mathbf{W} \alpha_1) \exp(\mathbf{x} t \mathbf{W} \alpha_2 \mathrm{i})$
HGRN-2 (Qin et al., 2024) HGRN-2（秦等人，2024 年）	$\mathbf{G}_t = \boldsymbol{\alpha}_t^\top \mathbf{1}, \quad \boldsymbol{\alpha}_t = \gamma + (1-\gamma)\sigma(\mathbf{x} t \mathbf{W} \alpha)$
RWKV-6 (B. Peng et al., 2024) RWKV-6 （B. Peng et al.， 2024）	$\mathbf{G}_t = \boldsymbol{\alpha}_t^\top \mathbf{1}, \quad \boldsymbol{\alpha}_t = \exp(-\exp(\mathbf{x} t \mathbf{W} \alpha))$
Gated Linear Attention (GLA) 门控线性注意力 （GLA）	$\mathbf{G}_t = \boldsymbol{\alpha}_t^\top \mathbf{1}, \quad \boldsymbol{\alpha} t = \sigma(\mathbf{x} t \mathbf{W} \alpha_1 \mathbf{W} \alpha_2)^{\frac{1}{\tau}}$

Gated linear attention formulation of recent models, which vary in their parameterization of $\mathbf{G}_t$. The bias terms are omitted.

最近模型的门控线性注意力公式，这些模型的参数化为 $\mathbf{G}_t$ 。偏差项被省略。

One notable variant is Mamba-2 (Dao & Gu, 2024) on the list. It's used in many of the hybrid models like Nemotron-H (NVIDIA, :, Blakeman, et al., 2025), Falcon H1 (Zuo et al., 2025), and Granite-4.0-h (IBM Research, 2025) .

一个值得注意的变体是名单上的 Mamba-2（Dao & Gu，2024）。它用于许多混合模型，如 Nemotron-H（NVIDIA，：，Blakeman 等人，2025 年）、Falcon H1（Zuo 等人，2025 年）和 Granite-4.0-h（IBM Research，2025 年）。

However, it's still early days and there's important nuance to consider when scaling to large hybrid models. While they show promise, MiniMax's experience with M2 highlights that benefits at small scale don't always translate to large-scale production

systems, particularly for complex reasoning tasks, RL training stability, and infrastructure maturity.

然而，现在还处于早期阶段，在扩展到大型混合模型时需要考虑重要的细微差别。虽然 MiniMax 在 M2 方面的经验显示出前景，但小规模的优势并不总是转化为大规模生产系统，特别是对于复杂的推理任务、RL 训练稳定性和基础设施成熟度。

That said, hybrid models are moving quickly and remain a solid choice for frontier training.

也就是说，混合模型正在迅速发展，并且仍然是前沿训练的可靠选择。

Qwen3-Next (with a gated DeltaNet update)(Qwen Team, 2025) reports they are faster at inference for long context, faster to train, and stronger on usual benchmarks. We are also looking forward for Kimi's next model that will most likely use their new "Kimi Delta Attention". Let's also mention Sparse Attention which solves the same issue for long context as linear attention by selecting block or query to compute attention. Some examples are Native Sparse Attention (Yuan et al., 2025), DeepSeek Sparse Attention (DeepSeek-AI, 2025) and InfLLM v2 (M. Team, Xiao, et al., 2025).

Qwen3-Next（带有门控 DeltaNet 更新）（Qwen Team，2025 年）报告称，它们在长上下文的推理方面更快，训练更快，并且在通常的基准测试中更强。我们也期待 Kimi 的下一款车型，它很可能会使用他们的新"Kimi Delta Attention"。我们还要提到稀疏注意力，它通过选择块或查询来计算注意力，解决了与线性注意力相同的长上下文问题。一些例子是 Native Sparse Attention（Yuan et al.， 2025）、DeepSeek Sparse Attention（DeepSeek-AI，2025）和 InfLLM v2（M. Team、Xiao 等人，2025 年）。

We'll wrap up the architecture choices before moving to tokenizers by building a small decision tree to determine weather to train a dense, a MoE or a Hybrid model.

在转向分词器之前，我们将通过构建一个小型决策树来确定天气来训练密集模型、MoE 或混合模型，从而总结架构选择。

TO MOE OR NOT MOE: CHOOSING A BASE ARCHITECTURE
TO MOE OR NOT MOE：选择基础架构

We've now seen dense, MoE and hybrid models so you might naturally be curious to know which one you should use? Your architecture choice typically depends on where you'll deploy the model, your team's expertise, and your timeline.

我们现在已经看到了密集、MoE 和混合模型，所以您可能自然会好奇应该使用哪一个？体系结构选择通常取决于部署模型的位置、团队的专业知识和时间表。

Let's go briefly over the pros and cons of each architecture and come up with a simple guiding process to find the right architecture for you.

让我们简要介绍一下每种架构的优缺点，并提出一个简单的指导过程来找到适合您的架构。

Dense transformers are the foundation standard decoder-only transformers where every parameter activates for every token. See The Annotated Transformers for the math and The Illustrated Transformers to build your intuition.

密集转换器是基础标准纯解码器转换器，其中每个参数都会为每个标记激活。请参阅 The Annotated Transformers 了解数学知识，并参见 The Illustrated Transformers 以建立您的直觉。

Pros: Widely supported, well-understood, stable training, good performance per parameter.

优点：支持广泛，理解良好，训练稳定，每个参数性能良好。

Cons: Compute scales linearly with size, a 70B model costs ~23× more than 3B.

缺点：计算随大小线性扩展，70B 模型的成本比 3B 高出 ~23×。

> This is usually the default choice for memory-contrained use cases or new LLM trainers.
>
> 这通常是内存受限用例或新 LLM 训练器的默认选择。

Mixture of Experts (MoE) replaces feed-forward layers in the transformer with multiple "experts". For each token, a gating network routes it to only a few experts. The result is the capacity of a large network at a fraction of the compute. For example Kimi K2 has 1T total parameters but only 32B active per token. The catch is that all experts must be loaded in memory. For a visual guide and reminder, check this blog.

混合专家 （MoE） 用多个"专家"替换变压器中的前馈层。对于每个代币，门控网络将其仅路由给少数专家。结果是大型网络的容量仅为计算量的一小部分。例如，Kimi K2 总参数为 1T，但每个代币只有 32B 活跃。问题是所有专家都必须加载到内存中。如需视觉指南和提醒，请查看此博客。

Pros: Better performance per compute for training and inference.

优点：训练和推理的每个计算性能更好。

Cons: High memory (all experts must be loaded). More complex training than dense transformers. Framework support is improving but less mature than dense models. And distributed training is a nightmare with expert placement, load balancing, and all-to-all communication challenges.

缺点：内存高（必须加载所有专家）。比密集变压器更复杂的训练。框架支持正在改进，但不如密集模型成熟。分布式培训是一场噩梦，存在专家安置、负载平衡和全方位通信挑战。

> Use when you're not memory-constrained and want maximum performance per compute.
>
> 当内存不受限制并且希望每个计算获得最大性能时使用。

Hybrid models combine transformers with State Space Models (SSMs) like Mamba, offering linear complexity for some operations versus attention's quadratic scaling. (Mathy blog | Visual guide)

混合模型将 Transformer 与 Mamba 等状态空间模型 （SSM） 相结合，为某些作提供线性复杂性，而不是注意力的二次缩放。（ 数学博客 |视觉指南）

Pros: Potentially better long-context handling. More efficient for very long sequences.

优点：可能更好的长上下文处理。对于很长的序列更有效。

Cons: Less mature than dense and MoE with fewer proven training recipes. Limited framework support.

缺点：不如密集和 MoE 成熟，经过验证的训练食谱较少。有限的框架支持。

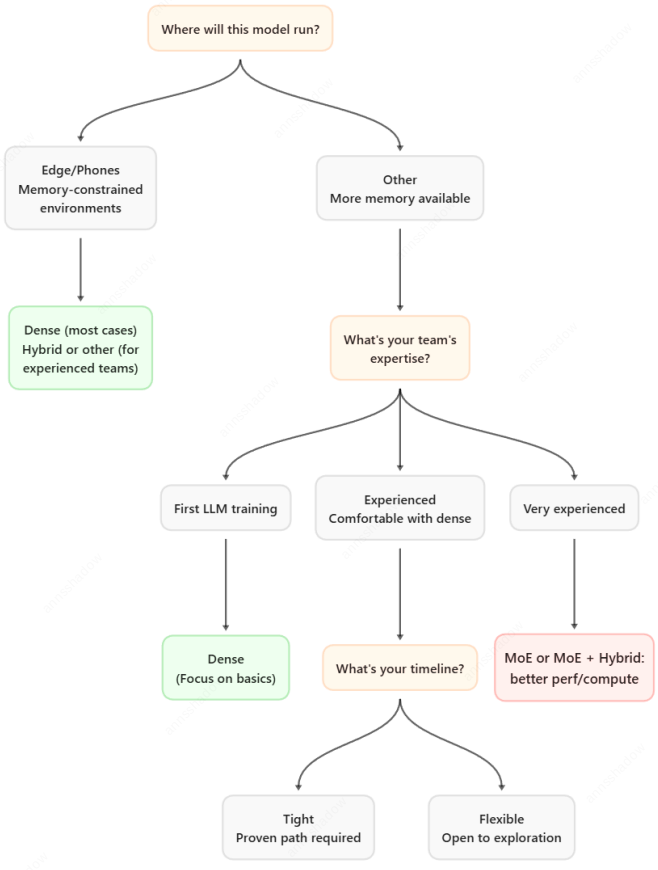
> Use if you want to scale to very large contexts while reducing the inference overhead of standard transformers.
>
> 如果要扩展到非常大的上下文，同时减少标准转换器的推理开销，请使用。

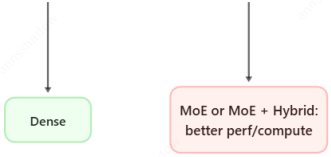
We're also now seeing some teams explore diffusion models for text, but these models (and other experimental alternatives) are very early stage that we consider them out of scope for the document.

我们现在还看到一些团队正在探索文本的扩散模型，但这些模型（和其他实验性替代方案）还处于非常早期的阶段，我们认为它们超出了文档的范围。

So to recap, start by asking where your model will be deployed. Then consider your team's expertise and your training timeline to assess how much exploration you can afford:

因此，回顾一下，首先询问您的模型将部署在哪里。然后考虑您团队的专业知识和培训时间表，以评估您能负担得起多少探索：





For SmolLM3 we wanted to build a strong small model for on-device deployment, we had roughly a 3-month timeline, and have mostly trained dense models in the past.

对于 SmolLM3，我们希望构建一个强大的小型模型用于设备上部署，我们大约有 3 个月的时间表，并且过去主要训练密集模型。

This ruled out MoE (memory constraints) and hybrid (short timeline to explore a new architecture, and dense models could get the long context we targeted of 128k tokens max), so we went for a dense model llama-style.

这排除了 MoE（内存约束）和混合（探索新架构的短时间线，密集模型可以获得我们目标的最多 128k 令牌的长上下文），因此我们选择了密集模型骆驼风格。

Now that we have studied the internals of the model architecture let's look at the tokenizer which forms the bridge between the data and our model.

现在我们已经研究了模型架构的内部结构，让我们看看分词器，它在数据和模型之间架起桥梁。

THE TOKENIZER 分词器

While it rarely steals the spotlight from architecture innovations, the tokenization scheme is likely one of the most underrated components of any language model.

虽然它很少从架构创新中抢走风头，但代币化方案可能是任何语言模型中最被低估的组件之一。

Think of it as the translator between human language and the mathematical world our model lives in, and just like any translator, the quality of the translation matters a lot. So how do we build or choose the right tokenizer for our needs?

将其视为人类语言和我们的模型所处的数学世界之间的翻译器，就像任何翻译器一样，翻译的质量非常重要。那么我们如何构建或选择适合我们需求的分词器呢？

Tokenizer fundamentals 分词器基础知识

At its core, a tokenizer converts raw text into sequences of numbers that our model can process, by segmenting a running text into individual processable units called tokens.

分词器的核心是将原始文本转换为我们的模型可以处理的数字序列，方法是将运行文本分割为称为标记的单个可处理单元。

Before diving into the technical details, we should first answer some fundamental questions that will guide our tokenizer design:

在深入研究技术细节之前，我们应该首先回答一些指导我们分词器设计的基本问题：

- **What languages do we want to support?** If we're building a multilingual model but our tokenizer has only seen English, the model will be inefficient when encountering non-English text, which will get split into many more tokens than necessary. This directly impacts performance, training cost and inference speed.

  我们想要支持哪些语言？如果我们正在构建一个多语言模型，但我们的分词器只看到英语，那么当遇到非英语文本时，该模型将效率低下，这些文本将被拆分为比必要的更多的标记。这直接影响性能、训练成本和推理速度。

- **Which domains matter to us?** Beyond languages, domains like math and code require careful representation of digits.

  哪些领域对我们很重要？除了语言之外，数学和代码等领域也需要仔细表示数字。

- **Do we know our target data mixture?** If we plan to train our tokenizer from scratch, ideally, we should train it on a sample that mirrors our final training mixture.

  我们知道我们的目标数据组合吗？如果我们计划从头开始训练分词器，理想情况下，我们应该在反映最终训练混合物的样本上训练它。

Once we have answered these questions, we can examine the main design decisions:

回答了这些问题后，我们可以检查主要的设计决策：

Vocabulary size 词汇量

The vocabulary is essentially a dictionary listing all tokens (minimal text units, like words, subwords, or symbols) our model recognizes.

For a deep dive into tokenization fundamentals, Andrej Karpathy's "Let's build the GPT Tokenizer" is an excellent hands-on tutorial. You can also check this resource which provides an introduction to tokenizers

词汇表本质上是一个字典，列出了我们的模型识别的所有标记（最小文本单位，如单词、子词或符号）。

Larger vocabularies compress text more efficiently since we generate fewer tokens per sentence, but there's a computational trade-off. The vocabulary size directly affects the size of our embedding matrices.

更大的词汇表可以更有效地压缩文本，因为我们每个句子生成的标记更少，但存在计算权衡。词汇量直接影响我们嵌入矩阵的大小。

If we have vocabulary size V and hidden dimension h, the input embeddings have V × h parameters, and the output layer has another V × h parameters.

如果我们有词汇大小 V 和隐藏维度 h，则输入嵌入具有 V × h 参数，输出层具有另一个 V × h 参数。

For smaller models, this becomes a significant chunk of total parameters as we've seen in the "Embedding Sharing" section, but the relative cost shrinks as models scale up.

对于较小的模型，正如我们在"嵌入共享"部分中看到的那样，这将成为总参数的重要组成部分，但随着模型的扩展，相对成本会缩小。

The sweet spot depends on our target coverage and model size. For English-only models, around 50k tokens usually suffices, but multilingual models often need 100k+ to efficiently handle diverse writing systems and languages.

最佳点取决于我们的目标覆盖范围和型号尺寸。对于纯英语模型，大约 50k 标记通常就足够了，但多语言模型通常需要 100k+ 才能有效地处理不同的书写系统和语言。

Modern state-of-the-art models like Llama3 have adopted vocabularies in the 128k+ range to improve token efficiency across diverse languages.

像 Llama3 这样的现代最先进的模型已经采用了 128k+ 范围内的词汇表，以提高不同语言的代币效率。

Smaller models in the same family apply embedding sharing to reduce the percentage of embedding parameters while still benefiting from the larger vocabulary.Dagan et al. (2024) analyze the impact of vocabulary size on compression, inference and memory. They observe that compression gains from larger vocabularies decrease exponentially, suggesting an optimal size exists.

同一家族中的较小模型应用嵌入共享来降低嵌入参数的百分比，同时仍受益于较大的词汇表。Dagan 等人（2024 年）分析了词汇量对压缩、推理和记忆的影响。他们观察到，较大词汇表的压缩增益呈指数级下降，这表明存在最佳大小。

For inference, larger models benefit from bigger vocabularies because compression saves more on the forward pass than the additional embedding tokens cost in softmax.

对于推理，较大的模型受益于更大的词汇表，因为压缩比 softmax 中额外的嵌入标记成本节省了更多的前向传递。

For memory, optimal size depends on sequence length and batch size: longer contexts and large batches benefit from larger vocabs due to KV cache savings from having fewer tokens.

对于内存，最佳大小取决于序列长度和批量大小：较长的上下文和大批量受益于更大的词汇，因为拥有更少的令牌可以节省 KV 缓存。

Tokenization algorithm 标记化算法

BPE (Byte-Pair Encoding) (Sennrich et al., 2016) remains the most popular choice, other algorithms like WordPiece or SentencePiece exist but are less adopted. There's also growing research interest in tokenizer-free approaches that work directly on bytes or characters, potentially eliminating tokenization altogether.

BPE（字节对编码）（Sennrich 等人，2016 年）仍然是最受欢迎的选择，其他算法如 WordPiece 或 SentencePiece 也存在，但较少采用。人们对直接用于字节或字符的无分词器方法的研究兴趣也越来越大，有可能完全消除标记化。

Now that we've seen the key parameters that define a tokenizer, we face a practical decision: should we use an existing tokenizer or train from scratch?

现在我们已经了解了定义分词器的关键参数，我们面临着一个实际的决定：我们应该使用现有的分词器还是从头开始训练？

The answer depends on coverage: whether existing tokenizers with our target vocabulary size handle our languages and domains well.

答案取决于覆盖范围：具有我们目标词汇量的现有分词器是否能很好地处理我们的语言和领域。

The figure below compares how GPT-2's English-only tokenizer (Radford et al., 2019) and Gemma 3's multilingual tokenizer (G. Team, Kamath, et al., 2025) segment the same English and Arabic sentence.

and a number of external resources.

要深入了解标记化基础知识，Andrej Karpathy 的"让我们构建 GPT 标记器"是一个出色的实践教程。您还可以查看此资源，其中介绍了分词器和许多外部资源。

Set vocabulary size to a multiple of 128 (e.g., 50,304 instead of 50,000) to optimize throughput.

将词汇量设置为 128 的倍数（例如，50,304 而不是 50,000）以优化吞吐量。

Modern GPUs perform matrix operations better when dimensions are divisible by higher powers of 2, as this ensures efficient memory alignment and reduces overhead from misaligned memory accesses.

当维度能被 2 的更高幂整除时，现代 GPU 可以更好地执行矩阵运算，因为这可以确保高效的内存对齐并减少内存访问未对齐造成的开销。

More details in thisblog.

更多详细信息，请参见此博客。

下图比较了 GPT-2 的纯英语分词器（Radford et al.， 2019）和 Gemma 3 的多语言分词器（G. Team， Kamath， et al.， 2025）如何分割同一个英语和阿拉伯语句子。

GPT-2

English 英语

TEXT 发短信

The development of large language models has revolutionized artificial intelligence research and applications across multiple domains especially education.

大型语言模型的发展彻底改变了人工智能在多个领域，尤其是教育领域的研究和应用。

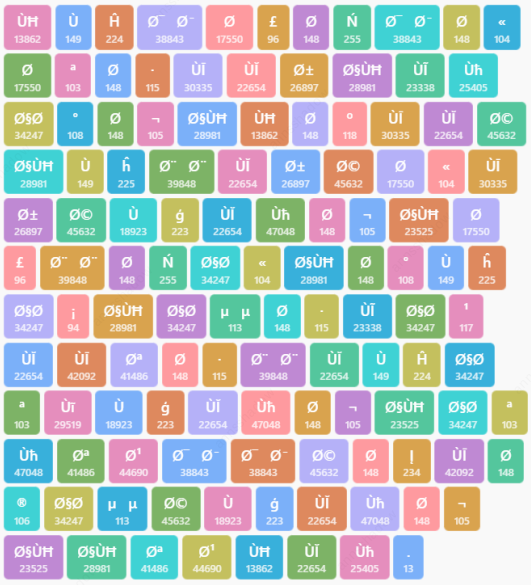
TOKENS 令牌 20

Arabic 阿拉伯语

TEXT 发短信

لقد أحدث تطوير النماذج اللغوية الكبيرة ثورة في مجال أبحاث الذكاء الاصطناعي وتطبيقاته في مجالات متعددة، وخاصة في مجال التعليم.

TOKENS 令牌 122



GEMMA 杰玛

English 英语

TEXT 发短信

The development of large language models has revolutionized artificial intelligence research and applications across multiple domains especially education.

大型语言模型的发展彻底改变了人工智能在多个领域，尤其是教育领域的研究和应用。

TOKENS 令牌 19

Arabic 阿拉伯语

TEXT 发短信

لقد أحدث تطوير النماذج اللغوية الكبيرة ثورة في مجال أبحاث الذكاء الاصطناعي وتطبيقاته في مجالات متعددة، وخاصة في مجال التعليم.

TOKENS 令牌 44

While both tokenizers seem to perform similarly on English, the difference becomes striking for Arabic: GPT2 breaks the text into over a hundred fragments, while Gemma3 produces far fewer tokens thanks to its multilingual training data and larger, more inclusive vocabulary.

虽然这两种分词器在英语上的表现似乎相似，但对于阿拉伯语来说，差异变得很明显：GPT2 将文本分解成一百多个片段，而 Gemma3 由于其多语言训练数据和更大、更具包容性的词汇量，产生的标记要少得多。

But to measure a tokenizer's quality we can't just eyeball a few tokenization examples and call it good, the same way we can't make architecture changes based on intuition without running ablations. We need concrete metrics to evaluate tokenizer quality.

但是，要衡量分词器的质量，我们不能只关注几个分词化示例并称其为好，就像我们不能在不

运行消融的情况下根据直觉进行架构更改一样。我们需要具体的指标来评估分词器的质量。

Measuring Tokenizer Quality

衡量分词器质量

To evaluate how well a tokenizer performs, we can use two key metrics used in FineWeb2 (Penedo et al., 2025).

为了评估分词器的性能，我们可以使用 FineWeb2 中使用的两个关键指标（Penedo 等人，2025 年）。

Fertility: 肥力:

It measures the average number of tokens needed to encode a word. Lower fertility means better compression, which translates to faster training and inference.

它测量对单词进行编码所需的平均标记数。较低的生育能力意味着更好的压缩，这意味着更快的训练和推理。

Think of it this way: if one tokenizer needs one or two more tokens to encode most words while another does it in less tokens, the second one is clearly more efficient.

可以这样想：如果一个分词器需要一两个多的标记来编码大多数单词，而另一个分词器用更少的标记来编码，那么第二个分词器显然更有效。

The standard approach for measuring fertility is to calculate **words-to-tokens ratio** (word fertility), which measures how many tokens are needed per word on average. This metric is defined around the concept of words because it provides meaningful cross-linguistic comparisons when appropriate word tokenizers are available, for example in Spacy and Stanza (Penedo et al., 2025).

衡量生育率的标准方法是计算单词与标记的比率（单词生育率），它衡量每个单词平均需要多少标记。该指标是围绕单词的概念定义的，因为它在适当的单词标记器可用时提供有意义的跨语言比较，例如在 Spacy 和 Stanza 中（Penedo 等人，2025 年）。

When comparing tokenizers for a single language, you can also use the number of characters or bytes instead of words to get the characters-to-tokens ratio or bytes-to-tokens ratio (Dagan et al., 2024). However, these metrics have limitations for cross-linguistic comparison. Bytes can be skewed since characters in different scripts require different byte representations (e.g., Chinese characters use three bytes in UTF-8 while Latin characters use one to two).

在比较单一语言的分词器时，您还可以使用字符数或字节数而不是单词数来获得字符与标记的比率或字节与标记的比率（Dagan 等人，2024 年）。然而，这些指标对于跨语言比较有局限性。字节可能会倾斜，因为不同脚本中的字符需要不同的字节表示（例如，中文字符在 UTF-8 中使用三个字节，而拉丁字符使用 1 到 2 个字节）。

Similarly, using the number of characters doesn't account for the fact that words vary dramatically in length across languages. For instance, Chinese words tend to be much shorter than German compound words.

同样，使用字符数并不能解释不同语言的单词长度差异很大的事实。例如，中文单词往往比德语复合词短得多。

Proportion of continued words:

连字比例:

This metric tells us what percentage of words get split into multiple pieces. Lower percentages are better since it means fewer words get fragmented, leading to more efficient tokenization.

该指标告诉我们有多少百分比的单词被分成多个部分。百分比越低越好，因为这意味着碎片化的单词更少，从而实现更有效的标记化。

Let's implement these metrics:

让我们实现这些指标：

```
import numpy as np
Â
def compute_tokenizer_metrics(tokenizer, word_tokenizer, text):
    """
    Computes fertility and proportion of continued words.

    Returns:
        tuple: (fertility, proportion_continued_words)
            - fertility: average tokens per word (lower is better)
            - proportion_continued_words: percentage of words split into 2+ toker
Â
    """
    words = word_tokenizer.word_tokenize(text)
    tokens = tokenizer.batch_encode_plus(words, add_special_tokens=False)
```

```
    tokens_per_word = np.array(list(map(len, tokens["input_ids"])))

    fertility = np.mean(tokens_per_word).item()
    proportion_continued_words = (tokens_per_word >= 2).sum() / len(tokens_per_wo

    return fertility, proportion_continued_words
```

For specialized domains like code and math, though, besides fertility we need to dig deeper and look at how well the tokenizer handles domain-specific patterns. Most modern tokenizers do single-digit splitting (so "123" becomes ["1", "2", "3"]) (Chowdhery et al., 2022; DeepSeek-AI et al., 2024). It might seem counterintuitive to break numbers apart, but it actually helps models learn arithmetic patterns more effectively.

For a deeper dive into how tokenization impacts arithmetic performance, From Digits to Decisions: How Tokenization Impacts Arithmetic in LLMs compares different tokenization schemes on math tasks.

不过，对于代码和数学等专业领域，除了生育能力之外，我们还需要更深入地挖掘并研究分词器处理特定领域模式的能力。大多数现代分词器都进行个位数拆分（因此"123"变为 ["1"、"2"、"3"]）（Chowdhery 等人，2022 年;DeepSeek-AI 等人，2024 年）。将数字分开似乎有悖常理，但它实际上可以帮助模型更有效地学习算术模式。

要更深入地了解标记化如何影响算术性能，《从数字到决策：标记化如何影响法学硕士中的算术》比较了数学任务中的不同标记化方案。

If "342792" is encoded as one indivisible token, the model must memorize what happens when you add, subtract, or multiply that specific token with every other number token. But when it's split, the model learns how digit-level operations work.

如果"342792"被编码为一个不可分割的标记，则模型必须记住当您将该特定标记与其他每个数字标记相加、减去或相乘时会发生什么。但是当它被拆分时，模型会学习数字级作的工作原理。

Some tokenizers like Llama3(Grattafiori et al., 2024) encode numbers from 1 to 999 as unique tokens and the rest are composed of these tokens.

一些分词器，如 Llama3（Grattafiori 等人，2024 年）将 1 到 999 的数字编码为唯一标记，其余的则由这些标记组成。

So we can measure fertility on our target domains to assess the weaknesses and strengths of a tokenizer. The table below compares fertility across popular tokenizers for different languages and domains.

因此，我们可以衡量目标域的生育能力，以评估分词器的弱点和优势。下表比较了不同语言和域的流行分词器的生育能力。

Evaluating tokenizers 评估分词器

To compare tokenizers across different languages, we'll use the setup from FineWeb2 tokenizer analysis, using Wikipedia articles as our evaluation corpus. For each language, we'll sample 100 articles to get a meaningful sample while keeping computation manageable.

为了比较不同语言的分词器，我们将使用 FineWeb2 分词器分析中的设置，使用维基百科文章作为我们的评估语料库。对于每种语言，我们将对 100 篇文章进行采样，以获得有意义的样本，同时保持计算的可管理性。

First, let's install dependencies and define which tokenizers and languages we want to compare:

首先，让我们安装依赖项并定义我们要比较的分词器和语言：

```
pip install transformers datasets sentencepiece 'datatrove[multilingual]'
## we need datatrove to load word tokenizers
```

```
tokenizers = [
    ("Llama3", "meta-llama/Llama-3.2-1B"),
    ("Gemma3", "google/gemma-3-1b-pt"),
    ("Mistral (S)", "mistralai/Mistral-Small-24B-Instruct-2501"),
    ("Qwen3", "Qwen/Qwen3-4B")
]
Â
languages = [
    ("English", "eng_Latn", "en"),
    ("Chinese", "cmn_Hani", "zh"),
    ("French", "fra_Latn", "fr"),
    ("Arabic", "arb_Arab", "ar"),
]
```

Now let's load our Wikipedia samples, we use streaming to avoid downloading entire datasets:

现在让我们加载我们的维基百科示例，我们使用流式处理来避免下载整个数据集：

```
from datasets import load_dataset
Â
wikis = {}
```

```
for lang_name, lang_code, short_lang_code in languages:
        wiki_ds = load_dataset("wikimedia/wikipedia", f"20231101.{short_lang_code
        wiki_ds = wiki_ds.shuffle(seed=42, buffer_size=10_000)
        # Sample 100 articles per language
    ds_iter = iter(wiki_ds)
    wikis[lang_code] = "\n".join([next(ds_iter)["text"] for _ in range(100)])
```

With our data ready, we can now evaluate each tokenizer on each language. For each combination, we load the appropriate word tokenizer from datatrove and compute both metrics:

准备好数据后，我们现在可以评估每种语言上的每个分词器。对于每个组合，我们从 datatrove 加载适当的单词分词器并计算这两个指标：

```
from transformers import AutoTokenizer
from datatrove.utils.word_tokenizers import load_word_tokenizer
import pandas as pd
Â
results = []

for tokenizer_name, tokenizer_path in tokenizers:
    tokenizer = AutoTokenizer.from_pretrained(tokenizer_path, trust_remote_code=T

    for lang_name, lang_code, short_lang_code in languages:
        word_tokenizer = load_word_tokenizer(lang_code)

        # Compute metrics on Wikipedia
        fertility, pcw = compute_tokenizer_metrics(tokenizer, word_tokenizer, wik

        results.append({
            "tokenizer": tokenizer_name,
            "language": lang_name,
            "fertility": fertility,
            "pcw": pcw
        })
Â
df = pd.DataFrame(results)
print(df)
```

```
      tokenizer    language  fertility       pcw
0        Llama3     English   1.481715  0.322058
1        Llama3     Chinese   1.601615  0.425918
2        Llama3      French   1.728040  0.482036
3        Llama3     Spanish   1.721480  0.463431
4        Llama3  Portuguese   1.865398  0.491938
5        Llama3     Italian   1.811955  0.541326
6        Llama3      Arabic   2.349994  0.718284
7        Gemma3     English   1.412533  0.260423
8        Gemma3     Chinese   1.470705  0.330617
9        Gemma3      French   1.562824  0.399101
10       Gemma3     Spanish   1.586070  0.407092
11       Gemma3  Portuguese   1.905458  0.460791
12       Gemma3     Italian   1.696459  0.484186
13       Gemma3      Arabic   2.253702  0.700607
14  Mistral (S)     English   1.590875  0.367867
15  Mistral (S)     Chinese   1.782379  0.471219
16  Mistral (S)      French   1.686307  0.465154
17  Mistral (S)     Spanish   1.702656  0.456864
18  Mistral (S)  Portuguese   2.013821  0.496445
19  Mistral (S)     Italian   1.816314  0.534061
20  Mistral (S)      Arabic   2.148934  0.659853
21        Qwen3     English   1.543511  0.328073
22        Qwen3     Chinese   1.454369  0.307489
23        Qwen3      French   1.749418  0.477866
24        Qwen3     Spanish   1.757938  0.468954
25        Qwen3  Portuguese   2.064296  0.500651
26        Qwen3     Italian   1.883456  0.549402
27        Qwen3      Arabic   2.255253  0.660318
```

The results reveal some winners and trade-offs depending on your priorities:

结果根据您的优先事项揭示了一些赢家和权衡：

Fertility (tokens per word)

生育力（每个单词的代币）

Lower is better 越低越好

Tokenizer (Vocab Size) 分词器（词汇量）	English 英语	Chinese 中文	French 法语	Arabic 阿拉伯语
Llama3 (128k) 骆驼 3 （128k）	1.48	1.60	1.73	2.35

Proportion of Continued Words

连续字数比例（%）

Lower is better 越低越好

Tokenizer (Vocab Size) 分
Llama3 (128k) 骆驼 3 （128

Mistral Small (131k) 米斯特拉尔小号 (131k)	1.59	1.78	1.69	2.15
Qwen3 (151k) Qwen3 (151k)	1.54	1.45	1.75	2.26
Gemma3 (262k) 杰玛 3 (262k)	1.41	1.47	1.56	2.25

Gemma3 tokenizer achieves low fertilities and word-splitting rates across multiple languages, notably on English, French, and Spanish, which can be explained by its tokenizer training data and very large vocabulary size of 262k, roughly 2x larger than Llama3's 128k.

Gemma3 分词器在多种语言中实现了低生育率和分词率，特别是在英语、法语和西班牙语上，这可以通过其分词器训练数据和 262k 的非常大的词汇量来解释，大约是 Llama3 的 2k 的 128 倍。

Qwen3 tokenizer excels on Chinese, but falls behind Llama3's tokenizer on English, French and Spanish.

Qwen3 分词器在中文方面表现出色，但在英语、法语和西班牙语方面落后于 Llama3 的分词器。

Mistral Small's tokenizer(Mistral AI, 2025) performs best on Arabic but falls short of the other tokenizers on English and Chinese.

Mistral Small 的分词器（Mistral AI，2025 年）在阿拉伯语上表现最好，但在英语和中文上不如其他分词器。

Choosing Between Existing and Custom Tokenizers

在现有分词器和自定义分词器之间进行选择

Currently, there's a good selection of strong tokenizers available. Many recent models start with something like GPT4's tokenizer (OpenAI et al., 2024) and augment it with additional multilingual tokens. As we can see in the table above, Llama 3's tokenizer performs well on average across multilingual text and code, while Qwen 2.5 excels particularly on Chinese and some low-resource languages.

目前，有很多强大的分词器可供选择。许多最近的模型都是从 GPT4 的分词器（OpenAI 等人，2024 年）之类的东西开始的，并使用额外的多语言令牌对其进行增强。正如我们在上表中看到的，Llama 3 的分词器在多语言文本和代码中平均表现良好，而 Qwen 2.5 在中文和一些资源匮乏的语言上尤其出色。

- **When to use existing tokenizers:** If our target use case matches the language or domain coverage of the best tokenizers above (Llama, Qwen, Gemma), then they are solid choices that were battle-tested.

  何时使用现有分词器：如果我们的目标用例与上述最佳分词器（Llama、Qwen、Gemma）的语言或领域覆盖范围相匹配，那么它们是经过实战考验的可靠选择。

  For SmolLM3 training we chose Llama3's tokenizer: it offers competitive tokenization quality on our target languages (English, French, Spanish, Portuguese, Italian) with a modest vocabulary size that made sense for our small model size.

  对于 SmolLM3 训练，我们选择了 Llama3 的分词器：它在我们的目标语言（英语、法语、西班牙语、葡萄牙语、意大利语）上提供具有竞争力的分词化质量，词汇量适中，这对于我们的小模型规模来说是有意义的。

  For larger models where embeddings are a smaller fraction of total parameters, Gemma3's efficiency gains become more attractive.

  对于嵌入占总参数较小比例的大型模型，Gemma3 的效率提升变得更具吸引力。

- **When to train our own:** If we're training for low-resource languages or have a very different data mixture, we'll likely need to train our own tokenizer to ensure good coverage.

  何时训练我们自己的：如果我们正在训练低资源语言或具有非常不同的数据组合，我们可能需要训练我们自己的分词器以确保良好的覆盖率。

  In which case it's important that we train the tokenizer on a dataset close to what we believe the final training mixture will look like. This creates a bit of a chicken-and-egg problem since we need a tokenizer to run data ablations and find the mixture.

  在这种情况下，重要的是我们在接近我们认为最终训练组合的外观的数据集上训练分词器。这会产生一些先有鸡还是先有蛋的问题，因为我们需要一个分词器来运行数据消融并找到混合物。

  But we can retrain the tokenizer before launching the final run and verify that downstream performance improves and fertilities are still good.

  但我们可以在启动最终运行之前重新训练分词器，并验证下游性能是否得到改善，生育率仍然良好。

Your tokenizer choice might seem like a technical detail, but it ripples through every aspect of your model's performance. So don't be afraid to invest time in getting it right.

您的分词器选择可能看起来像是一个技术细节，但它会波及模型性能的各个方面。因此，不要害怕投入时间来把事情做好。

SMOLLM3

Now that we've explored the architectural landscape and run our systematic ablations, let's see how this all comes together in practice for a model like SmolLM3.

现在我们已经探索了架构景观并运行了系统烧蚀，让我们看看这一切如何在实践中结合到像 SmolLM3 这样的模型中。

The SmolLM family is about pushing the boundaries of what's possible with small models. SmolLM2 delivered three capable models at 135M, 360M, and 1.7B parameters, all designed to run efficiently on-device.

SmolLM 系列旨在突破小型模型的可能性界限。SmolLM2 提供了三种功能强大的模型，参数分别为 135M、360M 和 1.7B，所有这些模型都旨在在设备上高效运行。

For SmolLM3, we wanted to scale up performance while staying small enough for phones, and tackle SmolLM2's weak spots: multilinguality, very long context handling, and strong reasoning capabilities. We chose 3B parameters as the sweet spot for this balance.

对于 SmolLM3，我们希望在保持足够小的手机容量的同时扩展性能，并解决 SmolLM2 的弱

点：多语言、非常长的上下文处理和强大的推理能力。我们选择了 3B 参数作为此平衡的最佳点。

Since we were scaling up a proven recipe, we naturally gravitated toward dense transformers. MoE wasn't implemented in nanotron yet, and we already had the expertise and infrastructure for training strong small dense models.

由于我们正在扩大经过验证的配方，我们自然而然地倾向于密集的变压器。MoE 尚未在 nanotron 中实现，我们已经拥有训练强大的小型密集模型的专业知识和基础设施。

More importantly, for edge device deployment we're memory-bound, an MoE with many parameters even if only a few are active would be limiting since we still need to load all experts into memory, making dense models more practical for our edge deployment targets.

更重要的是，对于边缘设备部署，我们是内存受限的，即使只有少数几个处于活动状态，具有许多参数的 MoE 也会受到限制，因为我们仍然需要将所有专家加载到内存中，这使得密集模型对于我们的边缘部署目标来说更实用。

Ablations: We started with SmolLM2 1.7B's architecture as our foundation, then trained a 3B ablation model on 100B tokens using Qwen2.5-3B layout. This gave us a solid baseline to test each modification individually.

消融：我们从 SmolLM2 1.7B 的架构开始，然后使用 Qwen2.5-3B 布局在 100B 标记上训练了一个 3B 消融模型。这为我们提供了一个坚实的基线来单独测试每个修改。

Each architecture change needed to either improve the loss and downstream performance on English benchmarks or provide measurable benefits like inference speed without quality degradation.

每次架构更改都需要提高英语基准测试中的损耗和下游性能，或者提供可衡量的好处，例如推理速度而不降低质量。

Here's what we tested before launching the run that made the cut:

以下是我们在启动晋级运行之前测试的内容：

Tokenizer: Before diving into architecture modifications, we needed to choose a tokenizer. We found a good set of tokenizers that covered our target languages and domains. Based on our fertility analysis, Llama3.

分词器：在深入研究架构修改之前，我们需要选择分词器。我们找到了一套很好的分词器，涵盖了我们的目标语言和领域。根据我们的生育力分析，Llama3.

2's tokenizer gave us the best tradeoff between our 6 target languages while keeping the vocabulary at 128k, large enough for multilingual efficiency but not so large that it bloated our 3B parameter count with embedding weights.

2 的分词器为我们在 6 种目标语言之间提供了最佳权衡，同时将词汇量保持在 128k，足够大以提高多语言效率，但又不会太大，以至于它因嵌入权重而使我们的 3B 参数计数变得臃肿。

Grouped Query Attention (GQA) : We reconfirmed our earlier finding that GQA with 4 groups matches Multi-Head Attention performance, but this time at 3B scale with 100B tokens. The KV cache efficiency gains were too good to pass up, especially for on-device deployment where memory is precious.

分组查询注意力 （GQA）：我们再次确认了我们之前的发现，即 4 组的 GQA 与多头注意力性能相匹配，但这次是 3B 规模，有 100B 令牌。KV 缓存效率的提升非常好，不容错过，特别是对于内存宝贵的设备上部署。

NoPE for long context : We implemented NoPE, by removing RoPE every 4th layer. Our 3B ablation confirmed the findings in the section above. NoPE improved long context handling without sacrificing short context performance.

NoPE 用于长上下文：我们通过每第 4 层删除 RoPE 来实现 NoPE。我们的 3B 消融证实了上一节中的发现。NoPE 改进了长上下文处理，而不会牺牲短上下文性能。

Intra-document attention masking : We prevent cross-document attention during training to help with training speed and stability when training on very large sequences, again we find that this doesn't impact downstream performance.

文档内注意力屏蔽：我们在训练期间防止跨文档注意力，以帮助在非常大的序列上训练时提高训练速度和稳定性，同样我们发现这不会影响下游性能。

Model layout optimization : We compared layouts from recent 3B models in the literature, some prioritizing depth, others width. We tested Qwen2.5-3B (3.1B), Llama3.2-3B (3.2B), and Falcon3-H1-3B (3.1B) layouts on our training setup, where depth and width varied.

模型布局优化：我们比较了文献中最近 3B 模型的布局，有些优先考虑深度，有些则优先考虑宽度。我们在训练设置上测试了 Qwen2.5-3B （3.1B）、Llama3.2-3B （3.2B） 和 Falcon3-H1-3B （3.1B） 布局，其中深度和宽度各不相同。

The results were interesting: all layouts achieved nearly identical loss and downstream performance, despite Qwen2.5-3B actually having fewer parameters. But Qwen2.

结果很有趣：尽管 Qwen2.5-3B 的参数实际上更少，但所有布局都实现了几乎相同的损耗和下游性能。但 Qwen2.

5-3B's deeper architecture aligned with research showing that network depth benefits generalization(Petty et al., 2024). Therefore, we went with the deeper layout, betting it would help as training progressed.

5-3B 的更深层次架构与研究表明网络深度有利于泛化相一致（Petty 等人，2024 年）。因此，我们采用了更深层次的布局，并打赌随着训练的进行，它会有所帮助。

Stability improvements : We kept tied embeddings from SmolLM2 but added a new trick inspired by OLMo2, removing weight decay from embeddings. Our ablations showed this didn't hurt performance while lowering embedding norms, which can help for preventing training divergence.

稳定性改进：我们保留了 SmolLM2 的绑定嵌入，但添加了一个受 OLMo2 启发的新技巧，消除了嵌入中的权重衰减。我们的消融表明，这不会损害性能，同时降低嵌入规范，这有助于防止训练发散。

The beauty of this systematic ablations approach is that we could confidently combine all these modifications, knowing each had been validated.

这种系统消融方法的美妙之处在于，我们可以自信地将所有这些修改结合起来，因为我们知道每一项都已经过验证。

> 💡 **Combining changes in ablations**
>
> **合并消融变化**
>
> In practice we test changes incrementally: once a feature was validated, it became part of the baseline for testing the next feature. Testing order matters: start with the battle-tested features first (tie embeddings → GQA → document masking → NoPE → remove weight decay).
>
> 在实践中，我们以增量方式测试更改：一旦一个功能被验证，它就会成为测试下一个功能的基线的一部分。测试顺序很重要：首先从经过实战测试的功能开始（→ GQA →文档屏蔽→→消除权重衰减的 GQA 文档屏蔽）。

RULES OF ENGAGEMENT 交战规则

> TL;DR: Your use case drives your choices.
>
> TL;DR：您的用例决定您的选择。

Let your deployment target guide architectural decisions. Consider how and where your model will actually run when evaluating new architectural innovations.

让您的部署目标指导架构决策。在评估新的体系结构创新时，请考虑模型的实际运行方式和位置。

Strike the right balance between innovation and pragmatism. We can't afford to ignore major architectural advances - using Multi-Head Attention today when GQA and better alternatives exist would be a poor technical choice. Stay informed about the latest research and adopt techniques that offer clear, validated benefits at scale.

在创新和实用主义之间取得适当的平衡。我们不能忽视重大的架构进步——在存在 GQA 和更好的替代方案的今天使用多头注意力将是一个糟糕的技术选择。随时了解最新研究，并采用能够大规模提供明确、经过验证的好处的技术。

But resist the temptation to chase every new paper that promises marginal gains (unless you have the resources to do so or your goal is architecture research).

但要抵制住追逐每一篇承诺边际收益的新论文的诱惑（除非你有资源这样做，或者你的目标是建筑研究）。

Systematic beats intuitive. Validate every architecture change, no matter how promising it looks on paper. Then test modifications individually before combining them to understand their impact.

系统节拍直观。验证每一次架构更改，无论它在纸面上看起来多么有希望。然后在组合修改之前单独测试修改以了解其影响。

Scale effects are real - re-ablate at target size when possible. Don't assume your small-scale ablations will hold perfectly at your target model size. If you have the compute, try to reconfirm them.

缩放效果是真实的 - 尽可能以目标大小重新烧蚀。不要假设您的小规模消融可以完美地保持在目标模型尺寸下。如果您有计算，请尝试重新确认它们。

Validate tokenizer efficiency on your actual domains. Fertility metrics across your target languages and domains matter more than following what the latest model used. A 50k English tokenizer won't cut it for serious multilingual work, but you don't need a 256k vocab if you're not covering that many languages either.

验证实际域上的分词器效率。目标语言和领域的生育率指标比遵循最新模型使用的内容更重要。50k 英语分词器不适合严肃的多语言工作，但如果您也没有涵盖那么多语言，则不需要 256k 词汇。

Now that the model architecture is now decided, it's time to tackle the optimizer and hyperparameters that will drive the learning process.

现在模型架构已经确定，是时候解决将驱动学习过程的优化器和超参数了。

Optimiser and training hyperparameters
优化器和训练超参数

The pieces are coming into place. We've run our ablations, settled on the architecture, and chosen a tokenizer. But before we can actually launch the training, there are still some crucial missing pieces: which optimizer should we use? What learning rate and batch size?

各个部分正在就位。我们已经运行了我们的消融，确定了架构，并选择了一个分词器。但在我们真正启动训练之前，仍然有一些关键的缺失部分：我们应该使用哪个优化器？学习率和批量大小是多少？

How should we schedule the learning rate over training?

我们应该如何安排学习率而不是培训？

The tempting approach here is to just borrow values from another strong model in the literature. After all, if it worked for big labs, it should work for us, right? And for many cases that will work just fine if we're taking values from a similar architecture and model size.

这里诱人的方法是从文献中的另一个强大模型中借用值。毕竟，如果它适用于大型实验室，那么它也应该适用于我们，对吧？在许多情况下，如果我们从类似的架构和模型大小中获取值，这将正常工作。

However, we risk leaving performance on the table by not tuning these values for our specific setup. Literature hyperparameters were optimized for specific data and constraints, and sometimes those constraints aren't even about performance.

但是，我们可能会因为不针对我们的特定设置调整这些值而将性能留在桌面上。文献超参数针对特定数据和约束进行了优化，有时这些约束甚至与性能无关。

Maybe that learning rate was picked early in development and never revisited. Even when model authors do thorough hyperparameter sweeps, those optimal values were found for their exact combination of architecture, data, and training regime, not ours.

也许这个学习率是在开发早期选择的，从未被重新审视过。即使模型作者进行了彻底的超参数扫描，这些最佳值也是根据它们的架构、数据和训练制度的精确组合找到的，而不是我们的。

Literature values are always a good starting point, but it's a good idea to explore if we can find better values in the neighbourhood.

文学价值观总是一个好的起点，但最好探索一下我们是否可以在社区中找到更好的价值观。

In this chapter, we'll explore the latest optimizers (and see if trusty old AdamW (Kingma, 2014) still stands the test of time 🎉), dive into learning rate schedules that go beyond the standard cosine decay and figure out how to tune the learning rate and batch size given a model and data size.

在本章中，我们将探索最新的优化器（并看看值得信赖的旧 AdamW（Kingma，2014）是否仍然经得起时间🎉的考验），深入研究超越标准余弦衰减的学习率计划，并弄清楚如何在给定模型和数据大小的情况下调整学习率和批量大小。

Let's start with the the optimizer wars.

让我们从优化器大战开始。

OPTIMIZERS: ADAMW AND BEYOND
优化器：ADAMW 及其他

The optimizer is at the heart of the whole LLM training operation. It decides for every parameter what the actual update step will be based on the past updates, the current

If you're uncertain what an optimizer is and what it's

weights and the gradients derived from the loss. At the same time it is also a memory and compute hungry beast and thus can impact how many GPUs you need and how fast your training is.

优化器是整个 LLM 训练作的核心。它根据过去的更新、当前权重和从损失得出的梯度来决定每个参数的实际更新步骤。同时，它也是一头对内存和计算产生渴望的野兽，因此会影响您需要多少 GPU 以及训练速度。

We didn't spare any efforts to summarize the current landscape of optimizers used for LLM pretraining:

我们不遗余力地总结了用于 LLM 预训练的优化器的当前情况：

Model 型	Optimizer 优化
Kimi K2, GLM 4.5 基米 K2，GLM 4.5	Muon μ子
Everyone else 别人	AdamW 亚当 W

So, you might wonder why everyone is using AdamW?

那么，你可能想知道为什么每个人都在使用 AdamW?

The person writing this part of the blog post thinks it's because "people are lazy" (hi, it's Elie), but others might more realistically say that AdamW has been working well/better at different scales for a long time, and it's always a bit scary to change such a core component especially if it's hard (ie expensive) to test how well it does in very long trainings.

写这部分博文的人认为这是因为"人们很懒"（嗨，我是 Elie），但其他人可能会更现实地说，AdamW 长期以来一直在不同规模上工作良好/更好，并且更改这样一个核心组件总是有点可怕，特别是如果很难（即昂贵）测试它在很长时间的培训中的表现。

Moreover, comparing optimizers fairly is harder than it looks. Scale changes the dynamics in ways that can be hard to simulate in small ablations, so hyperparameter tuning is complex. You could say: *"it's ok, I've tuned my AdamW for weeks, I can just reuse the same hyperparameters to compare!"* and we wish so much this would be true. But unfortunately, for each optimizer, you need to do proper hyperparameter search (1D? 2D? 3D?), which makes optimizer research hard and costly.

此外，公平地比较优化器比看起来更难。缩放以难以在小型烧蚀中模拟的方式改变动力学，因此超参数调整很复杂。你可以说："没关系，我已经调整了我的 AdamW 几个星期，我可以重复使用相同的超参数来进行比较！但不幸的是，对于每个优化器，您需要进行适当的超参数搜索（1D、2D、3D？），这使得优化器研究变得困难且成本高昂。

So let's start with the classic and the foundation of Durk Kingma's scary Google scholar domination: AdamW.

因此，让我们从经典和 Durk Kingma 可怕的 Google 学术统治的基础开始：AdamW。

AdamW 亚当 W

Adam (Adaptive Momentum Estimation) is a first order optimization technique. It means that in addition to looking at the gradients alone, we also consider how much the weights changed in the previous steps.

Adam（自适应动量估计）是一种一阶优化技术。这意味着除了单独查看梯度外，我们还考虑了前面步骤中权重的变化程度。

This makes the learning rate for each parameter adapt based on the momentum.

这使得每个参数的学习率根据动量进行调整。

The careful reader might wonder: hey there, aren't you missing a W? Indeed! The reason we specifically add the W (=weight decay) is the following. In standard SGD we can simply add a $\lambda\theta^2$ (where θ are the weights) to the loss to apply L2 regularization. However, if we do the same with Adam, the adaptive learning rate will also affect the L2 regularization. This means the regularization strength becomes dependent on gradient magnitudes, weakening its effect.

细心的读者可能会想：嘿，你不是错过了一个 W 吗？事实上！我们专门添加 W（=权重衰减）的原因如下。在标准 SGD 中，我们可以简单地在损失中添加一个 $\lambda\theta^2$ （权重在哪里 θ ）以应用 L2 正则化。然而，如果我们对 Adam 做同样的事情，自适应学习率也会影响 L2 正则化。这意味着正则化强度取决于梯度大小，从而削弱其效果。

This is not what we want and that's why AdamW applies it decoupled from the main optimization loop to fix this.

这不是我们想要的，这就是为什么 AdamW 将其与主优化循环解耦来解决这个问题。

Interestingly, across the last few years the AdamW hyperparameters have barely moved:

useful for, check Ruder's blog on Gradient Descent and optimizers, which notably compares cool optimizers

如果您不确定优化器是什么以及它的用途，请查看 Ruder 关于梯度下降和优化器的博客，其中特别比较了很酷的优化器

Often, the baseline is not tuned very well, so new optimizers are compared against a weak AdamW settings. A recent study (Wen et al., 2025) shows how much that alone skews reported gains.

通常，基线没有很好地调整，因此将新的优化器与弱 AdamW 设置进行比较。最近的一项研究（温等人，2025 年）表明，仅此一项就对报告的收益产生了多大的偏差。

有趣的是，在过去的几年里，AdamW 超参数几乎没有移动：

- $\beta_1 = 0.9$, $\beta_2 = 0.95$
- grad norm clipping = 1.0

  grad 范数裁剪 = 1.0
- weight decay = 0.1 (Llama-3-405B drops this to 0.01)

  权重衰减 = 0.1（Llama-3-405B 将其降至 0.01）

The same triplet is almost reused from Llama 1,2,3 to DeepSeek-V1,2,3 671B, no change. Was Durk Kingma right all along or can we do better?

从 Llama 1,2,3 到 DeepSeek-V1,2,3 671B 几乎重复使用了相同的三元组，没有变化。杜尔克·金玛一直都是对的，还是我们可以做得更好？

Muon in one line 一线中的 μ 子

Adam is a first-order methods, as it is only uses the gradients. Muon is a second-order optimizer that acts on the *matrix* view of a parameter tensor.

Adam 是一阶方法，因为它只使用梯度。Muon 是一种二阶优化器，作用于参数张量的矩阵视图。

$$
\begin{aligned}
G_t &= \nabla_\theta \mathcal{L}_t(\theta_{t-1}) \\
B_t &= \mu\, B_{t-1} + G_t \\
O_t &= \mathrm{NewtonSchulz5}(B_t) \;\approx\; UV^\top \quad \text{if } B_t = U\Sigma V^\top \text{ (SVD)} \\
\theta_t &= \theta_{t-1} - \eta\, O_t
\end{aligned}
$$

Looking at these equations you might wonder why is this a second order method, I only see gradients and no higher oder terms. The second order optimization actually happens inside the Newton Schulz step, but we won't go into further details here.

查看这些方程，您可能想知道为什么这是二阶方法，我只看到梯度，没有看到更高的项。二阶优化实际上发生在牛顿舒尔茨步长内，但我们在这里不再详细介绍。

There are already high-quality blogs that explain Muon in depth, so here we'll just list the three key ideas of Muon:

已经有高质量的博客深入解释了 Muon，所以这里我们只列出 Muon 的三个关键思想：

To learn more about Muon, we recommend checking out this blog by Keller Jordan, this one by Jeremy Bernstein and this video by Jia-Bin Huang which is a great starting point.

要了解有关 Muon 的更多信息，我们建议您查看 Keller Jordan 的这篇博客、Jeremy Bernstein 的这篇博客和 Jia-Bin Huang 的这段视频，这是一个很好的起点。

1. **Matrix-wise geometry vs. parameter-wise updates:** AdamW preconditions *per parameter* (diagonal second moment). Muon treats each weight **matrix** as a single object and updates along $G = UV^\top$, which captures row/column subspace structure.

   矩阵级几何与参数级更新：每个参数的 AdamW 预置条件（对角线秒力矩）。Muon 将每个权重矩阵视为单个对象并沿着 $G = UV^\top$ 进行更新，从而捕获行/列子空间结构。

2. **Isotropic steps via orthogonalization:** Decomposing $G = U\Sigma V^\top$ with singular value decomposition (SVD) separates magnitude (Σ) from directions (the left/right subspaces U, V). Replacing G by $UV^\top$ discards singular values and makes the step *isotropic* in the active subspaces. It's a bit counterintuitive at first—since throwing away Σ looks like losing information—but it reduces axis-aligned bias and encourages exploration of directions that would otherwise be suppressed by very small singular values.

   通过正交化实现各向同性步骤： $G = U\Sigma V^\top$ 使用奇异值分解 （SVD） 分解将大小 （ Σ ） 与方向（左/右子空间）分开 U, V 。 G 替换 丢 $UV^\top$ 弃奇异值并使步骤在活动子空间中各向同性。一开始这有点违反直觉——因为扔掉 Σ 看起来像是丢失信息——但它减少了轴对齐偏差，并鼓励探索原本会被非常小的奇异值抑制的方向。

   There's still an open question on whether this kind of exploration bakes different capabilities into the model that aren't obvious if you only look at the loss.

   关于这种探索是否将不同的功能融入到模型中，如果您只看损失，仍然存在一个悬而未决的问题。

3. **Empirical tolerance to larger batch sizes:** In practice, Muon often tolerates higher batch sizes. We'll talk about this more in depth in the batch size section, but this might be a key point of Muon adoption!

   对较大批量的经验容忍度：在实践中，Muon 通常可以容忍更高的批量大小。我们将在批量大小部分更深入地讨论这一点，但这可能是采用 Muon 的关键点！

For years, the community mostly settled on AdamW and the optimizer recipes of frontier labs are often kept secret (Qwen doesn't talk about theirs, for instance), but recently Muon has seen uptake in high-profile releases (e.g., Kimi K2, GLM-4.5).

多年来，社区大多选择了 AdamW，前沿实验室的优化器配方经常被保密（例如，Qwen 不谈论他们的），但最近 Muon 在备受瞩目的版本（例如 Kimi K2、GLM-4.5）中得到了采用。

Hopefully we'll see more open and robust recipes to use

希望我们能看到更多开放和强大的食谱可供使用

There is a wild zoo of optimizers and the only thing researchers are more creative at than combining all possible momentums and derivates is coming up with names for them: Shampoo, SOAP, PSGD, CASPR, DION, Sophia, Lion... even AdamW has its own variants like NAdamW, StableAdamW, etc. Diving into all these optimizers would be worth its own blog, but we'll keep that for another time.

优化器种类繁多，研究人员唯一比组合所有可能的动量和衍生物更有创意的就是为它们命名：洗发水、SOAP、PSGD、CASPR、DION、Sophia、Lion.....甚至 AdamW 也有自己的变体，如 NAdamW、StableAdamW 等。深入研究所有这些优化器将值得自己的博客，但我们将留到下次再说。

In the meantime, we recommend this amazing paper by the stanford/marin team(Wen et al., 2025) who benchmarked many different optimizer to show how important hyperparemeter tuning is when doing comparisons.

与此同时，我们推荐斯坦福大学/马林团队（温等人，2025 年）的这篇令人惊叹的论文，他们对许多不同的优化器进行了基准测试，以表明超参数调整在进行比较时的重要性。

Hand in hand with almost every optimizer is the question on how strongly we should update the weights determined by the learning rate which typically appears as a scalar value in the optimizer equations.

几乎每个优化器都面临着一个问题，即我们应该在多大程度上更新由学习率确定的权重，学习率通常在优化器方程中显示为标量值。

Let's have a look how this seemingly simple topic still has many facets to it.

让我们来看看这个看似简单的话题，它仍然有很多方面。

LEARNING RATE 学习率

The learning rate is one of the most important hyperparameters we'll have to set. At each training step, it controls how much we adjust our model weights based on the computed gradients.

学习率是我们必须设置的最重要的超参数之一。在每个训练步骤中，它控制我们根据计算的梯度调整模型权重的程度。

Choosing a learning rate too low and our training gets painfully slow and we can get trapped in a bad local minima. The loss curves will look flat, and we'll burn through our compute budget without making meaningful progress.

选择太低的学习率，我们的训练会变得非常缓慢，我们可能会陷入糟糕的局部最小值。损失曲线看起来很平坦，我们将耗尽我们的计算预算而不会取得有意义的进展。

On the other hand if we set the learning rate too high we cause the optimizer to take massive steps that overshoot optimal solutions and never converge or, the unimaginable happens, the loss diverges and the loss shoots to the moon.

另一方面，如果我们将学习率设置得太高，我们会导致优化器采取超出最优解的大量步骤，并且永远不会收敛，或者，发生难以想象的事情，损失发散，损失射向月球。

But the best learning rate isn't even constant since the learning dynamics change during training. High learning rates work early when we're far from good solutions, but cause instability near convergence.

但最佳学习率甚至不是恒定的，因为学习动态在训练过程中会发生变化。当我们远离好的解决方案时，高学习率会在早期发挥作用，但在收敛附近会导致不稳定。

This is where learning rate schedules come in: warmup from zero to avoid early chaos, then decay to settle into a good minimum. These patterns (warmup + cosine decay, for example) have been validated for neural networks training for many years.

这就是学习率计划的用武之地：从零开始热身以避免早期混乱，然后衰减以稳定到一个好的最低值。这些模式（例如，预热 + 余弦衰减）已在神经网络训练中得到验证多年。

> 💡 **Warmup steps 预热步骤**
>
> Most modern LLMs use a fixed number of warmup steps (for example 2000) regardless of model size and length of training, as shown in Table 1. We've found that for long trainings, increasing the number of warmup steps doens't have an impact on performance, but for very short trainings people usually use 1% to 5% of training steps.
>
> 大多数现代 LLM 使用固定数量的预热步骤（例如 2000 步），无论模型大小和训练长度如何，如表 1 所示。我们发现，对于长时间的训练，增加热身步数不会对表现产生影响，但对于非常短的训练，人们通常会使用 1% 到 5% 的训练步数。

Let's look at the common schedules, then discuss how to pick the peak value.

让我们看看常见的时间表，然后讨论如何选择峰值。

Learning Rate Schedules: Beyond Cosine Decay

学习率表：超越余弦衰减

It has been known for years that changing the learning rate helps convergence (Smith & Topin, 2018) and the cosine decay (Loshchilov & Hutter, 2017) was the go-to schedule for training LLMs: start at a peak learning rate after warmup, then smoothly decrease following a cosine curve. It's simple and works well.

多年来，人们都知道改变学习率有助于收敛（Smith & Topin，2018），而余弦衰减（Loshchilov & Hutter，2017）是训练法学硕士的首选时间表：在预热后以峰值学习率开始，然后按照余弦曲线平滑下降。它很简单，而且效果很好。

But its main disadvantage is inflexibility; we need to know our total training steps upfront, as the cosine cycle length must match your total training duration.

但它的主要缺点是不灵活;我们需要预先知道我们的总训练步数，因为余弦周期长度必须与您的总训练持续时间相匹配。

This becomes a problem in common scenarios: your model hasn't plateaued yet or you get access to more compute and want to train longer, or you're running scaling laws and need to train the same model on different token counts. Cosine decay forces you to restart from scratch.

这在常见情况下会成为一个问题：模型尚未稳定下来，或者你可以访问更多计算并希望训练更长时间，或者你正在运行缩放定律，需要在不同的令牌计数上训练同一模型。余弦衰减迫使您从头开始。

Interestingly, this inflexibility skewed early scaling laws research (Kaplan et al., 2020), because they used a fixed cosine schedule length when training models on varying token counts, underestimating the impact of data size. The Chinchilla study (Hoffmann et al., 2022) corrected this and matched the schedule length to each model's actual training duration.

有趣的是，这种不灵活性扭曲了早期的缩放定律研究（Kaplan et al.， 2020），因为他们在训练模型时使用了固定的余弦计划长度，低估了数据大小的影响。Chinchilla 研究（Hoffmann 等人，2022 年）纠正了这一点，并将计划长度与每个模型的实际训练持续时间相匹配。

Many teams now use schedules where you don't need to start decaying immediately after warmup. This is the case for **Warmup-Stable-Decay (** WSD) (Hu et al., 2024) and **Multi-Step** (DeepSeek-AI, :, et al., 2024) variants shown in the plot below.

许多团队现在使用的时间表，您不需要在热身后立即开始衰减。下图所示的 Warmup-Stable-Decay （WSD） （胡等人，2024 年）和多步 （DeepSeek-AI， ：， et al.， 2024） 变体就是这种情况。

You maintain a constant high learning rate for most of training, and either sharply decay in the final phase (typically the last 10-20% of tokens) for WSD, or do discrete drops (steps) to decrease the learning rate, for example after 80% of training and then after 90% as it was done inDeepSeek LLM's Multi-Step schedule.

在大部分训练中，您保持恒定的高学习率，要么在最后阶段（通常是最后 10-20% 的标记）急剧衰减 WSD，要么进行离散的下降（步骤）以降低学习率，例如在 80% 的训练之后，然后在 90% 之后，就像在 DeepSeek LLM 的多步骤计划中所做的那样。

Legend 传说

CSV load error: CSV file not found

These schedules offer practical advantages over cosine decay.

与余弦衰变相比，这些时间表具有实际优势。

We can extend training mid-run without restarting, whether we want to train longer than initially planned, are decay early to get a more measure of training progress and we can run scaling law experiments across different token counts with one main training run.

我们可以在运行中延长训练而不重新启动，无论我们是否想训练的时间比最初计划的更长，还是提前衰减以获得更多训练进度的衡量标准，并且我们可以通过一次主训练运行跨不同的标记计数运行缩放定律实验。

Moreover, studies show that both WSD and Multi-Step match cosine decay(DeepSeek-AI, :, et al., 2024; Hägele et al., 2024) while being more practical for real-world training scenarios.

此外，研究表明，WSD 和多步匹配余弦衰变（DeepSeek-AI， ：， et al.， 2024;Hägele 等人，2024 年），同时对于现实世界的训练场景更实用。

recently GLM 4.5 mentions that WSD perform worse on general benchmarks (SimpleQA, MMLU), but they don't provide any results.

最近 GLM 4.5 提到 WSD 在一般基准测试（SimpleQA、MMLU）上表现较差，但它们没有提供任何结果。

But you probably noticed that these schedules introduce new hyperparameters

compared to cosine: How long should the decay phase last in WSD? And how long should each step be in the Multi-Step variant?

但您可能注意到，与余弦相比，这些计划引入了新的超参数：WSD 中的衰减阶段应该持续多长时间？多步变体中的每个步骤应该有多长？

- For WSD: The required cooldown duration to match cosine performance decreases with longer training runs, and it is recommended to allocate 10-20% of total tokens to the decay phase (Hägele et al., 2024). We will confirm this setup matches cosine in our ablations below.

  对于 WSD：匹配余弦性能所需的冷却时间会随着训练运行时间的延长而减少，建议将总代币的 10-20% 分配给衰减阶段（Hägel 等人，2024 年）。我们将在下面的烧蚀中确认此设置与余弦匹配。

- For Multi-Step: DeepSeek LLM's ablations found that while their baseline 80/10/10 split (stable until 80%, first step from 80-90%, second step from 90-100%) matches cosine, tweaking these proportions can even outperform it, for instance when using 70/15/15 and 60/20/20 splits.

  对于多步：DeepSeek LLM 的消融发现，虽然它们的基线 80/10/10 分割（稳定到 80%，第一步从 80-90%，第二步从 90-100%）匹配余弦，但调整这些比例甚至可以优于它，例如在使用 70/15/15 和 60/20/20 分割时。

But we can get even more creative with these schedules. Let's look at the schedules used in each family of the DeepSeek models:

但我们可以在这些时间表上发挥更大的创造力。让我们看看 DeepSeek 模型的每个系列中使用的计划：

Legend 传说

CSV load error: CSV file not found

DeepSeek LLM used the baseline Multi-Step schedule (80/10/10). DeepSeek V2 adjusted the proportions to 60/30/10, giving more time to the first decay step. DeepSeek V3 took the most creative approach: instead of maintaining a constant learning rate followed by two sharp steps, they transition from the constant phase with a cosine decay (from 67% to 97% of training), then apply a brief constant phase before the final sharp step.

DeepSeek LLM 使用基线多步骤计划 （80/10/10）。DeepSeek V2 将比例调整为 60/30/10，为第一个衰减步骤提供了更多时间。DeepSeek V3 采用了最具创意的方法：它们不是保持恒定的学习率，然后是两个尖锐的步骤，而是从具有余弦衰减的恒定阶段过渡（从训练的 67% 到 97%），然后在最后一个尖锐的步骤之前应用一个简短的恒定阶段。

> **DeepSeek Schedules Change**
>
> **DeepSeek 时间表更改**
>
> DeepSeek-V2 and V3's technical reports don't include ablations on these schedule changes. For your setup, start with simple WSD or Multi-Step schedules, then consider tuning the parameters through ablations.
>
> DeepSeek-V2 和 V3 的技术报告不包括这些时间表变更的消融。对于您的设置，请从简单的 WSD 或多步骤计划开始，然后考虑通过烧蚀来调整参数。

Let's stop our survey of exotic learning rate schedules here and burn some GPU hours to determine what works in practice!

让我们在这里停止对奇异学习率计划的调查，并消耗一些 GPU 小时来确定哪些在实践中有效！

Ablation - WSD matches Cosine

消融 - WSD 匹配余弦

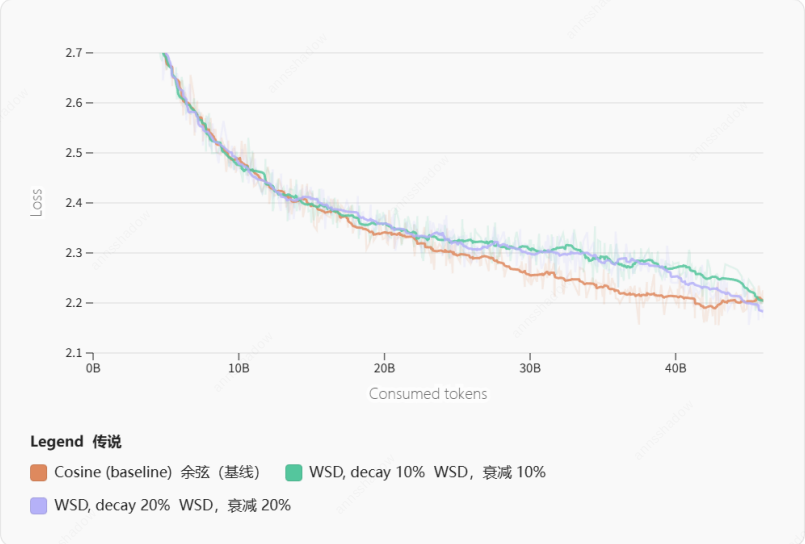
Now it's time for an ablation! Let's test whether WSD actually matches cosine's performance in practice. We won't show Multi-Step ablations here but we recommend DeepSeek LLM's ablations where they showed that Multi-Step matches cosine with

different phase splits.

现在是消融的时候了！让我们测试一下 WSD 在实践中是否真的与余弦的性能相匹配。我们不会在这里展示多步消融，但我们推荐 DeepSeek LLM 的消融，它们表明多步匹配具有不同相位分裂的余弦。

In this section, we'll compare cosine decay against WSD with two decay windows: 10% and 20%.

在本节中，我们将比较余弦衰减与具有两个衰减窗口的 WSD：10% 和 20%。



The evaluation results show similar final performance across all three configurations.

评估结果显示，所有三种配置的最终性能相似。

Looking at the loss and evaluation curves (specifically HellaSwag), we see an interesting pattern: cosine achieves better loss and evaluation scores during the stable phase (before WSD's decay begins).

查看损失和评估曲线（特别是 HellaSwag），我们看到一个有趣的模式：余弦在稳定阶段（在 WSD 衰减开始之前）获得了更好的损失和评估分数。

However, once WSD enters its decay phase, there's an almost linear improvement in both loss and downstream metrics allowing WSD to catch up to cosine by the end of training.

然而，一旦 WSD 进入衰减阶段，损失和下游指标几乎都会发生线性改善，使 WSD 能够在训练结束时赶上余弦。

This confirms that WSD's 10-20% decay window is sufficient to match cosine's final performance while maintaining the flexibility to extend training mid-run. We opted for WSD with 10% decay for SmolLM3.

这证实了 WSD 的 10-20% 衰减窗口足以匹配余弦的最终性能，同时保持在运行中延长训练的灵活性。我们选择了 SmolLM3 衰变 10% 的 WSD。

> ⚠️ **Comparing models trained with different schedulers mid-run**
>
> **比较运行中用不同调度器训练的模型**
>
> If you're comparing intermediate checkpoints between cosine and WSD during the stable phase, make sure to apply a decay to the WSD checkpoint for a fair comparison.
>
> 如果在稳定阶段比较余弦和 WSD 之间的中间检查点，请确保对 WSD 检查点应用衰减以进行公平比较。

Now that we have a good overview of popular learning rate schedules, the next question is: what should the peak learning rate actually be?

现在我们已经很好地了解了流行的学习率计划，下一个问题是：峰值学习率实际上应该是多少？

Finding The Optimal Learning Rate

寻找最佳学习率

How do we pick the right learning rates for our specific learning rate scheduler and training setups?

我们如何为特定的学习率调度程序和培训设置选择正确的学习率？

We could run learning rate sweeps on short ablations like we did for architecture choices. But optimal learning rate depends on training duration: a learning rate that converges fastest in a short ablation might not be the best one for the full run.

我们可以像对架构选择所做的那样对短消融进行学习率扫描。但最佳学习率取决于训练持续时间：在短时间消融中收敛最快的学习率可能不是完整运行的最佳学习率。

And we can't afford to run expensive multi-week trainings multiple times just to test different learning rates.

而且我们不能仅仅为了测试不同的学习率而多次进行昂贵的多周培训。

Let's first look at simple sweeps we can quickly run that help us rule out learning rates that are much too high or low and then we'll discuss scaling laws for hyperparameters.

让我们首先看看我们可以快速运行的简单扫描，这些扫描可以帮助我们排除过高或过低的学习率，然后我们将讨论超参数的缩放定律。

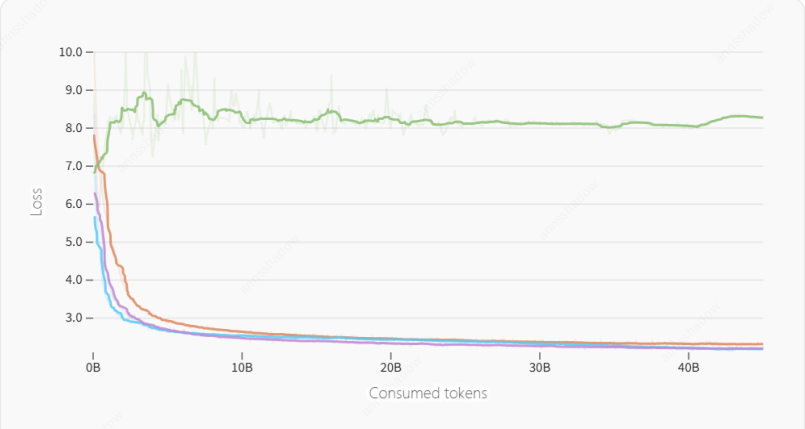
Ablation - LR sweeps 消融 - LR 扫描

To illustrate the impact of different learning rates, let's look at a sweep on our 1B ablation model trained on 45B tokens. We train the same model, under the same setup with 4 different learning rates: 1e-4, 5e-4, 5e-3, 5e-2.

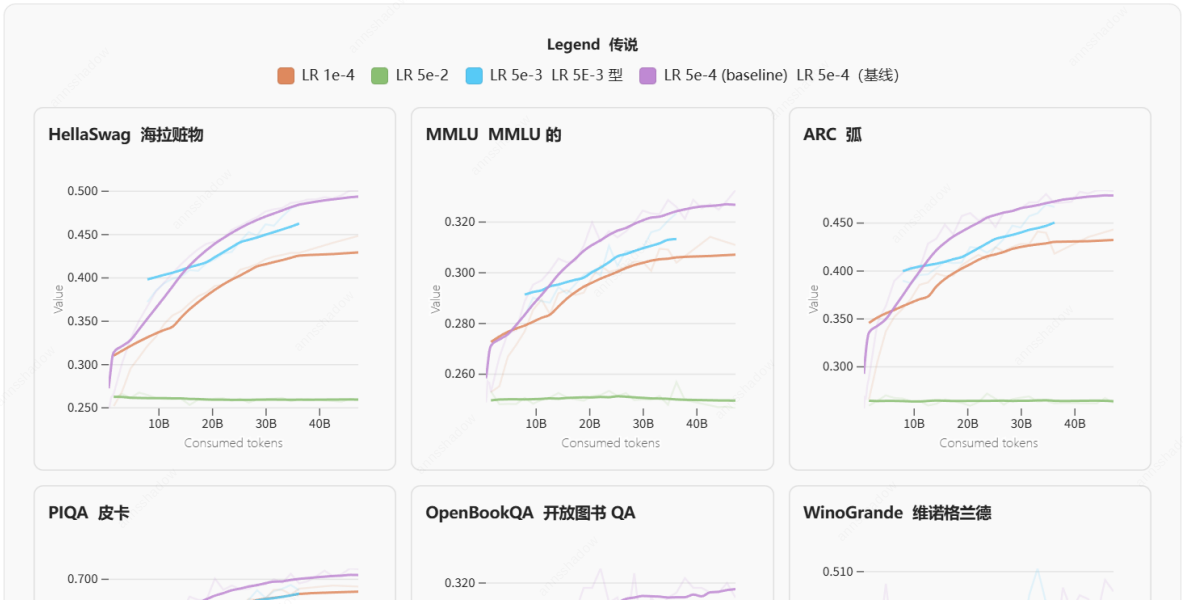
为了说明不同学习率的影响，让我们看一下在 45B 标记上训练的 1B 消融模型的扫描。我们训练相同的模型，在相同的设置下，具有 4 种不同的学习率：1e-4、5e-4、5e-3、5e-2。

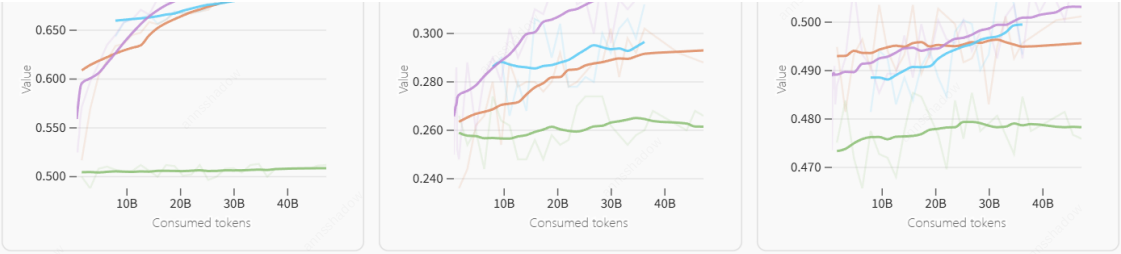
The results clearly show the dangers at both extremes:

结果清楚地显示了两个极端的危险：




Legend 传说
LR 1e-4
LR 5e-2
LR 5e-3 LR 5E-3 型
LR 5e-4 (baseline) LR 5e-4（基线）


Legend 传说
LR 1e-4
LR 5e-2
LR 5e-3 LR 5E-3 型
LR 5e-4 (baseline) LR 5e-4（基线）
HellaSwag 海拉赃物
0.500
0.450
0.400
0.350
0.300
0.250
Value
10B
20B
30B
40B
Consumed tokens
MMLU MMLU 的
0.320
0.300
0.280
0.260
Value
10B
20B
30B
40B
Consumed tokens
ARC 弧
0.450
0.400
0.350
0.300
Value
10B
20B
30B
40B
Consumed tokens
PIQA 皮卡
0.700
OpenBookQA 开放图书 QA
0.320
WinoGrande 维诺格兰德
0.510



LR 5e-2 diverges almost immediately, the loss spikes early and never recovers, making the model unusable. LR 1e-4 is too conservative, while it trains stably, it converges much more slowly than the other learning rates.

LR 5e-2 几乎立即发散，损失早期飙升并且永远不会恢复，使模型无法使用。LR 1e-4 过于保守，虽然训练稳定，但收敛速度比其他学习率慢得多。

The middle ground of 5e-4 and 5e-3 show better convergence and comparable performance. But running sweeps for every model size gets expensive quickly, and more importantly, it doesn't account for the planned number of training tokens as we previously stated.

5e-4 和 5e-3 的中间地带显示出更好的收敛性和可比的性能。但是，为每个模型大小运行扫描很快就会变得昂贵，更重要的是，它没有像我们之前所说的那样考虑计划的训练令牌数量。

This is where scaling laws become invaluable.

这就是缩放法变得无价的地方。

For SmolLM3, we trained 3B models on 100B tokens with AdamW using the WSD schedule, comparing several learning rates. We found that 2e-4 converged much faster than 1e-4 in both loss and downstream performance, while 3e-4 was only slightly better than 2e-4.

对于 SmolLM3，我们使用 WSD 计划使用 AdamW 在 100B 标记上训练 3B 模型，比较了几种学习率。我们发现，2e-4 在损耗和下游性能方面的收敛速度都比 1e-4 快得多，而 3e-4 仅略好于 2e-4。

The marginal gains from 3e-4 came with increased risk of instability during long training runs, so we chose 2e-4 as our sweet spot.

3e-4 的边际增益伴随着长时间训练期间不稳定的风险增加，因此我们选择了 2e-4 作为最佳选择。

These sweeps help us rule out learning rates that are clearly too high (divergence) or too low (slow convergence), but running sweeps for every model size gets expensive quickly, and more importantly, it doesn't account for the planned number of training tokens as we previously stated.

这些扫描帮助我们排除了明显过高（发散）或过低（收敛缓慢）的学习率，但对每种模型大小运行扫描很快就会变得昂贵，更重要的是，它没有考虑我们之前所说的训练令牌的计划数量。

This is where scaling laws become invaluable.

这就是缩放法变得无价的地方。

But before we dive into scaling laws for hyperparameters, let's discuss the other critical hyperparameter that interacts with learning rate: batch size.

但在我们深入研究超参数的缩放定律之前，让我们先讨论一下与学习率交互的另一个关键超参数：批量大小。

BATCH SIZE 批量大小

The batch size is the number of samples processed before updating model weights. It directly impacts both training efficiency and final model performance. Increasing the batch size improves throughput if your hardware and training stack scale well across devices.

批量大小是在更新模型权重之前处理的样本数。它直接影响训练效率和最终模型性能。如果硬件和训练堆栈可以跨设备很好地扩展，则增加批处理大小可以提高吞吐量。

Throughput is the number of tokens processed per second during training.

吞吐量是训练期间每秒处理的令牌数。

But beyond a certain point, larger batches start to hurt data efficiency: the model needs more total tokens to reach the same loss.

但超过一定点，更大的批次开始损害数据效率：该模型需要更多的总代币才能达到相同的损失。

The breakpoint where this happens is known as the**critical batch size** (McCandlish et al., 2018).

发生这种情况的断点称为临界批量大小（McCandlish 等人，2018）。

- **Increasing the batch size while staying below critical:** after increasing the batch size and retuning the learning rate, you reach the same loss with the same number of tokens as the smaller batch size run, no data is wasted.

  增加批大小，同时保持在临界以下：增加批大小并重新调整学习率后，使用与较小的批次大小运行相同的令牌数量达到相同的损失，不会浪费任何数据。

- **Increasing the batch size while staying above critical:** larger batches start to sacrifice data efficiency; reaching the same loss now requires more total tokens (and thus more money), even if wall-clock time drops because more chips are busy.

  增加批次大小，同时保持在临界值以上：较大的批次开始牺牲数据效率;达到相同的损失现在需要更多的代币总量（因此需要更多的钱），即使挂钟时间因为更多的筹码忙碌而减少。

Let's try to give some intuition about why we need to retune the learning rate, and how to compute an estimation of what the critical batch size should be.

让我们试着给出一些直觉，说明为什么我们需要重新调整学习率，以及如何计算临界批量大小的估计值。

When batch size grows, each mini-batch gradient is a better estimate of the true gradient, so you can safely take a larger step (i.e., increase the learning rate) and reach a target loss in fewer updates. The question is how to scale it.

当批次大小增长时，每个小批量梯度都是对真实梯度的更好估计，因此您可以安全地采取更大的步骤（即提高学习率）并在更少的更新中达到目标损失。问题是如何扩展它。

Averaging over B samples

对 B 样本进行平均

- Batch gradient: $\tilde{g}_B = \frac{1}{B}\sum_{i=1}^{B}\tilde{g}^{(i)}$
  批量梯度：$\tilde{g}_B = \frac{1}{B}\sum_{i=1}^{B}\tilde{g}^{(i)}$
- Mean stays the same: $\mathbb{E}[\tilde{g}_B] = g$
  均值保持不变：$\mathbb{E}[\tilde{g}_B] = g$
- But covariance shrinks: $\mathrm{Cov}(\tilde{g}_B) = \frac{\Sigma}{B}$
  但协方差会缩小：$\mathrm{Cov}(\tilde{g}_B) = \frac{\Sigma}{B}$

The SGD parameter update is:

SGD 参数更新为：

- $\Delta w = -\eta\,\tilde{g}_B$

The variance of this update is proportional to:

此更新的方差与以下内容成正比：

- $\mathrm{Var}(\Delta w) \propto \eta^2\frac{\Sigma}{B}$

so to keep the update variance roughly constant, if you scale the batch size by k, you want to scale the learning rate by $\sqrt{k}$. So let's say you have you have computed your optimal batch size and learning rate and you've find that increasing to the critical batch size is possible and increasing the throughput, you'll need to adapt the optimal learning rate as well.

因此，为了保持更新方差大致恒定，如果将批次大小缩放为 k，则需要将学习率缩放为 $\sqrt{k}$ 。因此，假设您已经计算了最佳批次大小和学习率，并且您发现可以增加到临界批次大小并提高吞吐量，您还需要调整最佳学习率。

$$B_{\text{critical}} \to kB_{\text{optimal}} \quad \Rightarrow \quad \eta_{\text{critical}} \to \sqrt{k}\eta_{\text{optimal}}$$

A useful rule of thumb for optimizers like AdamW or Muon is ***square-root LR scaling*** as batch size grows, but this also depends on the optimizer. For instance using AdamW there are interactions with `beta1` / `beta2` that can introduce very different behavior.

对于 AdamW 或 Muon 等优化器来说，一个有用的经验法则是随着批量大小的增长而进行平方根 LR 缩放，但这也取决于优化器。例如，使用 AdamW 时，与 / `beta2` 的 `beta1` 交互可能会引入非常不同的行为。

A pragmatic alternative is to branch training for a brief window: keep one run at the original batch, start a second with the larger batch and a rescaled LR, and only adopt the larger batch if the two loss curves align after the rescale(Merrill et al., 2025). In the paper, they re-warm up the learning rate and reset the optimizer state when switching the batch size. They also set a tolerance and a time window to decide whether the losses "match", both knobs are chosen empirically. They found that the B_{simple} estimate - which is also noisy - is underestimating the "actual" critical batch size. This

See the (amazing) Jianlin Su's series for more math on this: https://kexue.fm/archives/11260

请参阅（惊人的）苏建林的系列，了解更多有关此的数学知识：https://kexue.fm/archives/11260

gives you a quick, low-risk check that the new batch/LR pair preserves training losses “match”, both knobs are chosen empirically. They found that the B_{simple} estimate - which is also noisy - is underestimating the “actual” critical batch size. This gives you a quick, low-risk check that the new batch/LR pair preserves training dynamics.

一个务实的替代方案是在短暂的窗口内进行分支训练：在原始批次上保留一次运行，用更大的批次和重新缩放的 LR 开始第二次运行，并且只有在重新缩放后两条损失曲线对齐时才采用较大的批次（Merrill 等人，2025 年）。在论文中，他们在切换批量大小时重新预热学习率并重置优化器状态。他们还设置了一个容差和一个时间窗口来决定损失是否“匹配”，这两个旋钮都是根据经验选择的。他们发现，这个 B_{simple} 估计值——也很嘈杂——低估了“实际”临界批量大小。这可以让您快速、低风险地检查新的批次/LR 对是否保留了训练动态。

The critical batch size isn’t fixed, it grows as training progresses. Early in training, the model is making big gradient step, so $\|g\|^2$ is big which means B_{simple} is small, hence the model have a smaller critical batch size. Later, as the model updates stabilizes and larger batches become more effective. This is why some large-scale trainings don’t keep the batch size constant and use what we can batch size warmup.

临界批大小不是固定的，它会随着训练的进行而增长。在训练的早期，模型正在进行较大的梯度步长，因此 $\|g\|^2$ big B_{simple} 意味着较小，因此模型具有较小的临界批量大小。稍后，随着模型更新稳定下来，大批量变得更加有效。这就是为什么一些大规模训练不会保持批量大小恒定，而是使用我们可以批量大小预热的内容。

For example, DeepSeek-V3 begins with a 12.6 M batch for the first ~469 B tokens, then increases to 62.9M for the remainder of training.

例如，DeepSeek-V3 从前 ~469 个 B 令牌的 12.6 M 批次开始，然后在剩余的训练中增加到 62.9M。

A batch-size warmup schedule like this serves the same purpose as a learning-rate warmup: it keeps the model on the efficient frontier as the gradient noise scale grows, maintaining stable and efficient optimization throughout.

像这样的批量大小预热计划与学习率预热具有相同的目的：随着梯度噪声尺度的增长，它使模型保持在有效的前沿，从而始终保持稳定和高效的优化。

Another interesting approach is treating the loss as a proxy for the critical batch size. Minimax01 used this and in the last stage they trained with a 128M batch size!

另一种有趣的方法是将损失视为临界批量大小的代理。Minimax01 使用了这个，在最后阶段他们以 128M 批量进行了训练！

This is a bit different because they don’t increase the learning rate, so their batch-size schedule acts like a learning-rate decay schedule.

这有点不同，因为它们不会提高学习率，因此它们的批量大小计划就像学习率衰减计划一样。

> **Tuning batch size and learning rate**
>
> **调整批量大小和学习率**
>
> In practice, here’s how you can choose the batch size and learning rate:
>
> 在实践中，您可以通过以下方式选择批量大小和学习率：
>
> - You first pick the batch size and learning rate you consider optimal, either from scaling laws (see later!) or from literature.
>
>   您首先从缩放定律（见后文）或文献中选择您认为最佳的批量大小和学习率。
>
> - Then, you can tune the batch size to see if you can improve the training throughput.
>
>   然后，可以调整批处理大小，看看是否可以提高训练吞吐量。
>
> The key insight is that there’s often a range between your starting batch size and the critical batch size where you can increase it to improve hardware utilization without sacrificing data efficiency, but you must retune the learning rate accordingly.
>
> 关键见解是，起始批大小和关键批大小之间通常存在一个范围，您可以增加它以提高硬件利用率，而不会牺牲数据效率，但您必须相应地重新调整学习率。
>
> If the throughput gain isn’t significant, or if testing a larger batch size (with rescaled learning rate) shows worse data efficiency, stick with the initial values.
>
> 如果吞吐量增益不显着，或者如果测试更大的批量大小（使用重新缩放的学习率）显示数据效率较差，请坚持使用初始值。

As mentioned in the note above, one way to pick your starting points for the batch size and learning rate is through scaling laws. Let’s see how these scaling laws work and how they predict both hyperparameters as a function of your compute budget.

如上文所述，选择批量大小和学习率起点的一种方法是通过缩放定律。让我们看看这些缩放定律是如何工作的，以及它们如何预测这两个超参数作为计算预算的函数。

SCALING LAWS FOR HYPERPARAMETERS
超参数的缩放定律

The optimal learning rate and batch size aren't just about model architecture and size, they also depends on our compute budget, which combines both the number of model parameters and the number of training tokens.

最佳学习率和批量大小不仅与模型架构和大小有关，还取决于我们的计算预算，该预算结合了模型参数的数量和训练令牌的数量。

This is where scaling laws come in.

这就是缩放法的用武之地。

Scaling laws establish empirical relationships describing how model performance evolves as we increase training scale, whether that's through larger models or more training data (see the "Scaling laws" section at the end of this chapter for the full history).

缩放定律建立了经验关系，描述了模型性能如何随着我们增加训练规模而演变，无论是通过更大的模型还是更多的训练数据（有关完整历史，请参阅本章末尾的"缩放定律"部分）。

But scaling laws can also help us predict how to adjust key hyperparameters like the learning rate and batch size as we scale up training, as it was done in recent work byDeepSeek and Qwen2.5. This gives us principled defaults rather than relying entirely on hyperparameter sweeps.

但缩放定律还可以帮助我们预测如何在扩大训练时调整关键超参数，例如学习率和批量大小，就像 DeepSeek 和 Qwen2.5 最近的工作所做的那样。这为我们提供了有原则的默认值，而不是完全依赖超参数扫描。

To apply scaling laws in this context, we need a way to quantify training scale. The standard metric is the compute budget, denoted C and measured in FLOPs, which can be approximated as:

为了在这种情况下应用缩放定律，我们需要一种量化训练尺度的方法。标准指标是计算预算，表示为 C 并以 FLOP 为单位，可以近似为：

$$C \approx 6 \times N \times D$$

N is the number of model parameters (e.g., 1B = 1e9), D is the number of training tokens. This is often measured in FLOPs (floating-point operations), a hardware-agnostic way of quantifying how much actual computation is being done.

N 是模型参数的数量（例如，1B = 1e9），D 是训练标记的数量。这通常以 FLOP（浮点运算）来衡量，这是一种与硬件无关的量化实际计算量的方法。

But if FLOPs feel too abstract, just think of it this way: training a 1B parameter model on 100B tokens consumes about 2× fewer FLOPs than training a 2B model on 100B tokens, or a 1B model on 200B tokens.

但如果 FLOP 感觉过于抽象，可以这样想：在 100B 代币上训练 1B 参数模型比在 100B 代币上训练 2B 模型或在 200B 代币上训练 1B 模型消耗的 FLOP 少约 2×。

The constant 6 comes from empirical estimates of how many floating-point operations are required to train a Transformer, roughly 6 FLOPs per parameter per token.

常数 6 来自对训练 Transformer 需要多少个浮点运算的经验估计，每个标记每个参数大约 6 个 FLOP。

The constant 6 comes from empirical estimates of how many floating-point operations are required to train a Transformer, roughly 6 FLOPs per parameter per token.

常数 6 来自对训练 Transformer 需要多少个浮点运算的经验估计，每个标记每个参数大约 6 个 FLOP。

If you want a more precise measure taking into account MoE layers and Hybrid layers you can checkout the num_floating_point_operations function in Megatron-LM.

如果您想要更精确的测量，同时考虑 MoE 层和混合层，您可以查看 Megatron-LM 中的 num_floating_point_operations 功能。

Now, how does this relate to learning rate? We can derive scaling laws that predict optimal learning rates and batch sizes as functions of total compute budget (C). They help answer questions like:

现在，这与学习率有什么关系？我们可以推导出缩放定律，将最佳学习率和批量大小预测为总计算预算 （C） 的函数。它们有助于回答以下问题：

- How should the learning rate change as I scale from 1B to 7B parameters?

  当我从 1B 参数扩展到 7B 参数时，学习率应该如何变化？

- If I double my training data, should I adjust the learning rate?

  如果我的训练数据翻倍，我应该调整学习率吗？

Let's see how this works be walking through the approach DeepSeek used: First, we

choose our learning rate schedule, ideally WSD for its flexibility.

让我们看看这是如何工作的，通过 DeepSeek 使用的方法：首先，我们选择我们的学习率计划，最好是 WSD，因为它具有灵活性。

Then, we train models across a range of compute budgets (e.g., 1e17, 5e17, 1e18, 5e18, 1e19, 2e19 FLOPs) with different combinations of batch sizes and learning rates.

然后，我们使用不同的批量大小和学习率组合，在一系列计算预算（例如，1e17、5e17、1e18、5e18、1e19、2e19 FLOP）中训练模型。

In simpler terms: we train different model sizes for different numbers of tokens, testing different hyperparameter settings.

简单来说：我们针对不同数量的代币训练不同的模型大小，测试不同的超参数设置。

This is where the WSD schedule shines, we can extend the same training run to different token counts without restarting.

这就是 WSD 计划的亮点，我们可以将相同的训练运行扩展到不同的令牌计数，而无需重新启动。

For each setup, we perform sweeps over learning rate and batch size and identify the configurations that result in near-optimal performance, typically defined as being within a small margin (e.g., 0.25%) of the best validation loss (computed on an independent validation set, with a similar distribution to the training set).

对于每个设置，我们对学习率和批量大小进行扫描，并确定导致接近最佳性能的配置，通常定义为在最佳验证损失（在独立验证集上计算，与训练集具有相似分布）的很小的边际（例如，0.25%）内。

Each near-optimal configuration gives us a data point — a tuple of (compute budget C, optimal learning rate η) or (C, optimal batch size B).

每个接近最优的配置都为我们提供了一个数据点——（计算预算 C，最佳学习率 η）或 （C，最佳批量大小 B） 的元组。

When plotted on a log-log scale, these relationships typically follow power-law behavior, appearing as approximately straight lines (as shown in the figure above).

当绘制在对数-对数尺度上时，这些关系通常遵循幂律行为，显示为近似直线（如上图所示）。

By fitting these data points, we can extract scaling laws that describe how optimal hyperparameters evolve with compute.

通过拟合这些数据点，我们可以提取扩展定律，描述最佳超参数如何随着计算而演变。

An important finding from this process is that for a fixed model size and compute budget, performance remains stable across a wide range of hyperparameters. This means there's a broad sweet spot rather than a narrow optimum.

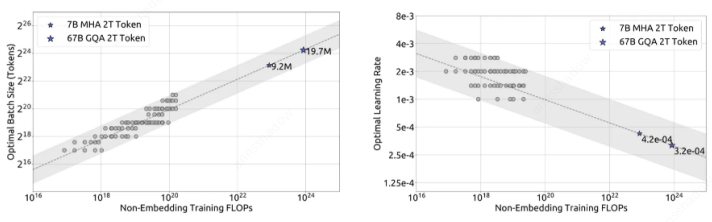
这个过程的一个重要发现是，对于固定的模型大小和计算预算，性能在各种超参数上保持稳定。这意味着有一个广泛的最佳点，而不是一个狭窄的最佳点。

We don't need to find the perfect value, just a value that's close enough, which makes the whole process much more practical.

我们不需要找到完美的值，只需要一个足够接近的值，这使得整个过程更加实用。

Here you can see the results of the scaling laws DeepSeek derived, where each dot represents a near optimal setting:

在这里，您可以看到 DeepSeek 推导的缩放定律的结果，其中每个点代表一个接近最佳的设置：



(a) Batch size scaling curve (b) Learning rate scaling curve

Figure 3 | Scaling curves of batch size and learning rate. The grey circles represent models whose generalization error exceeded the minimum by no more than 0.25%. The dotted line represents the power law fitting the smaller model. The blue stars represent DeepSeek LLM 7B and 67B.

The core intuition behind these results is that as training becomes larger and longer, we want **more stable updates** (hence, smaller learning rates) and **more efficient gradient estimation** (hence, larger batch sizes).

这些结果背后的核心直觉是，随着训练变得更大、更长，我们需要更稳定的更新（因此，学习率更小）和更高效的梯度估计（因此，更大的批量大小）。

These scaling laws give us starting points for the learning rate and batch size. But the

objective is not “optimal samples per gradient” but “lower loss reachable within our time and number of GPUd constraint” while still extracting the full signal from every token.

这些缩放定律为我们提供了学习率和批量大小的起点。但目标不是“每个梯度的最佳样本”，而是“在我们的时间和 GPU 数量约束内可达到的更低损耗”，同时仍然从每个标记中提取完整信号。

In practice, you may be able to increase the batch size beyond the predicted optimal batch size to significantly improve throughput without meaningfully hurting data efficiency, up to the critical batch size we discussed earlier.

在实践中，您可以将批次大小增加到超出预测的最佳批次大小，以显着提高吞吐量，而不会显着损害数据效率，直至达到我们之前讨论的临界批次大小。

SMOLLM3

So what did we end up using for SmolLM3? At the time of ablations before launching SmolLM3, we compared AdamW, AdEMAMix, and Muon on a 1B model trained on 100B tokens. Muon could outperform AdamW when properly tuned but was sensitive to learning rate and prone to divergence.

那么我们最终为 SmolLM3 使用了什么？在启动 SmolLM3 之前进行消融时，我们在训练了 100B 代币的 1B 模型上比较了 AdamW、AdEMAMix 和 Muon。如果调整得当，Muon 的性能可以优于 AdamW，但对学习率敏感并且容易出现发散。

AdeMaMix was less sensitive and achieved similar loss to Muon. AdamW was the most stable but reached a higher final loss than the tuned alternatives.

AdeMaMix 的灵敏度较低，其损失与 Muon 相似。AdamW 是最稳定的，但最终损失高于调整后的替代方案。

However, when we scaled up to 3B, we encountered more frequent divergence with Muon and AdeMaMix. This may have been due to a parallelism bug we discovered after finishing the ablations (see The Training Marathon chapter), though we haven't confirmed this.

然而，当我们扩展到 3B 时，我们遇到了 Muon 和 AdeMaMix 更频繁的分歧。这可能是由于我们在完成消融后发现的并行性错误（参见训练马拉松一章），尽管我们尚未证实这一点。

We decided to use AdamW (beta1: 0.9, beta2: 0.95) with weight decay 0.1 and gradient clipping 1.

我们决定使用 AdamW（beta1：0.9，beta2：0.95），权重衰减 0.1，梯度裁剪 1。

After all a very vanilla setting.

毕竟，这是一个非常普通的设置。

For the learning rate schedule, we chose WSD. We had used it successfully in SmolLM2, and it proved to be one of our best decisions for ease of use and flexibility regarding total training duration plus the ability to run mid-training decay experiments.

对于学习率表，我们选择了 WSD。我们在 SmolLM2 中成功地使用了它，事实证明，它是我们关于总训练持续时间的易用性和灵活性以及运行训练中衰减实验的能力的最佳决策之一。

We ran learning rate sweeps and settled on 2e-4. For the global batch size, we tested values from 2M to 4M tokens but found minimal impact on the loss or downstream performance, so we chose 2.36M tokens - the size that gave us the best throughput.

我们进行了学习率扫描，最终确定了 2e-4。对于全局批量大小，我们测试了从 2M 到 4M 令牌的值，但发现对损失或下游性能的影响很小，因此我们选择了 2.36M 令牌——这个大小为我们提供了最佳吞吐量。

RULES OF ENGAGEMENT 交战规则

> **TL;DR:** Balance exploration and execution, done is better than perfect.
>
> TL;DR：平衡探索和执行，完成总比完美好。

We've talked a lot about the “what” (optimizer, learning rate, batch size) but just as important is the **how** . How do we decide what's worth experimenting with? How do we structure our time? When do we stop exploring and just train?

我们已经讨论了很多关于“什么”（优化器、学习率、批量大小），但同样重要的是 how。我们如何决定什么值得尝试？我们如何安排我们的时间？我们什么时候停止探索而只训练？

Allocate your time wisely between exploration and execution. Spending weeks

perfecting a minor improvement from a new method is less valuable than investing that same compute in better data curation or more thorough architecture ablations.

在探索和执行之间明智地分配您的时间。花数周时间完善新方法的微小改进，不如将相同的计算投入到更好的数据管理或更彻底的架构烧蚀上更有价值。

From our experience, and though it might disappoint architecture enthusiasts, the biggest performance gains usually come from data curation.

根据我们的经验，尽管这可能会让架构爱好者失望，但最大的性能提升通常来自数据管理。

When in doubt, choose flexibility and stability over peak performance. If two methods perform equally well, pick the one that offers more flexibility or that has better implementation maturity and stability.

如有疑问，请选择灵活性和稳定性而不是最佳性能。如果两种方法的性能同样出色，请选择提供更多灵活性或具有更好实现成熟度和稳定性的方法。

A learning rate schedule like WSD that lets us extend training or run mid-training experiments is more valuable than a rigid schedule that might converge slightly better.

像 WSD 这样的学习率计划可以让我们延长训练或运行训练中期实验，比可能收敛得稍微好的僵化计划更有价值。

Know when to stop optimizing and start training. There's always one more hyperparameter to tune or one more optimizer to try. Set a deadline for exploration and stick to it - the model we actually finish training will always beat the perfect model we never start.

知道何时停止优化并开始训练。总是需要调整一个超参数或尝试多个优化器。设定一个探索的最后期限并坚持下去——我们实际完成训练的模型总是会击败我们从未开始的完美模型。

One more ablation won't hurt (Spoiler: It did). Credits to sea_snell

再消融一次不会有什么坏处（剧透：确实如此）。sea_snell 的功劳

Perfect is the enemy of good, especially when we're working with finite compute budgets and deadlines.

完美是好的敌人，尤其是当我们处理有限的计算预算和截止日期时。

Scaling laws: how many parameters, how much data?
缩放定律：有多少参数，多少数据？

In the early days of deep learning, before language models (and the clusters they were trained on) were "large", training runs were often not heavily constrained by compute.

在深度学习的早期，在语言模型（以及它们被训练的集群）变得"大"之前，训练运行通常不受计算的严重限制。

When training a model, you'd just pick the largest model and batch size that fit on your hardware and then train until the model started overfitting or you ran out of data.

训练模型时，只需选择适合硬件的最大模型和批量大小，然后进行训练，直到模型开始过度拟合或数据用完。

However, even in these early days there was a sense that scale was helpful — for example,Hestness et al. provided a comprehensive set of results in 2017 showing that training larger models for longer produced predictable gains.

然而，即使在早期，也有一种感觉，规模是有帮助的——例如，Hestness 等人在 2017 年提供了一组全面的结果，表明训练更大的模型以获得更长的时间会产生可预测的收益。

In the era of large language models, we are *always* compute-constrained. Why? These early notions of scalability were formalized by Kaplan et al.'s work on Scaling Laws for Neural Language Models, where it was shown that language model performance is remarkably predictable across many orders of magnitude of scale. This set off an explosion in the size and training duration of language models, because it provided a way to *accurately predict* how much performance would improve from increasing scale.

在大型语言模型时代，我们总是受到计算的约束。为什么？Kaplan 等人关于神经语言模型缩放定律的工作正式确定了这些早期的可扩展性概念，其中表明语言模型的性能在许多数量级上都是可以显着预测的。这引发了语言模型的规模和训练持续时间的爆炸式增长，因为它提供了一种准确预测规模增加会提高多少性能的方法。

Consequently, the race to build better language models became a race to train larger models on larger amounts of data with ever-growing compute budgets, and the development of language models quickly became compute-constrained.

因此，构建更好语言模型的竞赛变成了在计算预算不断增长的情况下在大量数据上训练更大模型的竞赛，语言模型的开发很快变得受计算限制。

When faced with compute constraints, the most important question is whether to train a larger model or to train on more data.

当面临计算约束时，最重要的问题是是训练更大的模型还是训练更多数据。

Surprisingly, Kaplan et al.'s scaling laws suggested that it was advantageous to allocate much more compute towards model scale than previous best practices — motivating, for example, training the gargantuan (175B parameters) GPT-3 model on a relatively modest token budget (300B tokens).

令人惊讶的是，Kaplan 等人的扩展定律表明，与以前的最佳实践相比，为模型规模分配更多的计算是有利的——例如，激励在相对适中的令牌预算（175B 令牌）上训练庞大（300B 参数）的 GPT-3 模型。

On reexamination,Hoffman et al. found a methodological issue with Kaplan et al.'s approach, ultimately re-deriving scaling laws that suggested allocating much more compute to training duration which indicated, for example, that compute-optimal training of the 175B-parameter GPT-3 should have consumed 3.

在重新检查时，Hoffman 等人发现 Kaplan 等人的方法存在方法论问题，最终重新推导了缩放定律，该定律建议为训练持续时间分配更多的计算，这表明，例如，175B 参数 GPT-3 的计算

This shifted the field from "make models bigger" to "train them longer and better."

However, most modern trainings still don't strictly follow the Chinchilla laws, because they have an shortcoming: They aim to predict the model size and *training* duration that achieves the best performance given a certain compute budget, but they fail to account for the fact that larger models are more expensive *after* training. Put another way, we might actually prefer to use a given compute budget train a smaller model for longer — even if this isn't "compute-optimal" — because this will make inference costs cheaper (Sardana et al., de Vries). This could be the case if we expect that a model will be see a lot of inference usage (for example, because it's being released openly 🤗).

这将该领域从"让模型变大"转变为"训练它们更长、更好"。然而，大多数现代训练仍然没有严格遵循龙猫定律，因为它们有一个缺点：它们旨在预测在给定一定计算预算的情况下实现最佳性能的模型大小和训练持续时间，但它们没有考虑到更大的模型在训练后更昂贵的事实。换句话说，我们实际上可能更愿意使用给定的计算预算来训练一个更小的模型更长时间——即使这不是"计算最优"——因为这将使推理成本更便宜（Sardana 等人，de Vries）。如果我们预计一个模型将看到大量的推理使用（例如，因为它是公开🤗发布的），则可能是这种情况。

Recently, this practice of "overtraining" models beyond the training duration

suggested by scaling laws has become standard practice, and is the approach we took when developing SmolLM3.

最近，这种超出缩放定律建议的训练持续时间的"过度训练"模型的做法已成为标准做法，也是我们在开发 SmolLM3 时采取的方法。

While scaling laws provide a suggestion for the model size and training duration given a particular compute budget, choosing to overtrain means you have to decide these factors yourself. For SmolLM3, we started by picking a target model size of 3 billion parameters.

虽然缩放定律为给定特定计算预算的模型大小和训练持续时间提供了建议，但选择过度训练意味着您必须自己决定这些因素。对于 SmolLM3，我们首先选择了 30 亿个参数的目标模型大小。

Based on recent models of a similar scale like Qwen3 4B, Gemma 3 4B, and Llama 3.2 3B, we considered 3B to be large enough to have meaningful capabilities (such as reasoning and tool calling), but small enough to enable super fast inference and efficient local usage.

基于最近类似规模的模型，如 Qwen3 4B、Gemma 3 4B 和 Llama 3.2 3B，我们认为 3B 足够大，可以具有有意义的能力（例如推理和工具调用），但又足够小，可以实现超快速推理和高效的本地使用。

To pick a training duration, we first noted that recent models have been*extremely* overtrained — for example, the aforementioned Qwen3 series is claimed to have been trained for 36T tokens! As a result, training duration is often dictated by the amount of compute available.

为了选择训练持续时间，我们首先注意到最近的模型已经被极度过度训练——例如，前面提到的 Qwen3 系列据称已经针对 36T 代币进行了训练！因此，训练持续时间通常取决于可用的计算量。

We secured 384 H100s roughly a month, which provided a budget for training on 11 trillion tokens (assuming an MFU of ~30%).

我们大约一个月获得了 384 个 H100，这为 11 万亿个代币的训练提供了预算（假设 MFU 为 ~30%）。

> **Scaling laws 缩放定律**
>
> Despite these deviations, scaling laws remain practically valuable. They provide baselines for experimental design, people often use Chinchilla-optimal setups to get signal on ablations, and they help predict whether a model size can reach a target performance.
>
> 尽管存在这些偏差，缩放法则仍然具有实际价值。它们为实验设计提供基线，人们经常使用龙猫最佳设置来获取消融信号，并帮助预测模型大小是否可以达到目标性能。
>
> As de Vries notes in thisblog, by scaling down model size you can hit a critical model size: the minimal capacity required to reach a given loss, below which you start getting diminishing return.
>
> 正如 de Vries 在本博客中指出的那样，通过缩小模型大小，您可以达到关键模型大小：达到给定损失所需的最小容量，低于该容量，您开始获得收益递减。

Now that we're settled on our model architecture, training setup, model size, and training duration, we need to prepare two critical components: the data mixture that will teach our model, and the infrastructure that will train it reliably.

现在我们已经确定了模型架构、训练设置、模型大小和训练持续时间，我们需要准备两个关键组件：将教导模型的数据混合，以及将可靠地训练它的基础设施。

With SmolLM3's architecture set at 3B parameters, we needed to curate a data mixture that would deliver strong multilingual, math and code performance, and set up infrastructure robust enough for 11 trillion tokens of training.

由于 SmolLM3 的架构设置为 3B 参数，我们需要策划一个能够提供强大的多语言、数学和代码性能的数据组合，并建立足够强大的基础设施来进行 11 万亿个训练。

Getting these fundamentals right is essential, even the best architectural choices won't save us from poor data curation or unstable training systems.

正确掌握这些基础知识至关重要，即使是最好的架构选择也无法使我们免于糟糕的数据管理或不稳定的训练系统。

The art of data curation
数据管理的艺术

Picture this: you've spent weeks perfecting your architecture, tuning hyperparameters, and setting up the most robust training infrastructure.

想象一下：您花了数周时间完善架构、调整超参数并设置最强大的训练基础设施。

Your model converges beautifully, and then… it can't write coherent code, struggles with basic math, and maybe even switches languages mid-sentence. What went wrong? The answer usually lies in the data.

您的模型收敛得很漂亮，然后……它无法编写连贯的代码，在基本数学方面遇到困难，甚至可能在句子中途切换语言。哪里出了问题？答案通常在于数据。

While we obsess over fancy architectural innovations and hyperparameter sweeps, data curation often determines whether our model becomes genuinely useful or just another expensive experiment.

虽然我们痴迷于花哨的架构创新和超参数扫描，但数据管理通常决定我们的模型是否真正有用，或者只是另一个昂贵的实验。

It's the difference between training on random web crawls versus carefully curated, high-quality datasets that actually teach the skills we want our model to learn.

这是随机网络爬虫训练与精心策划的高质量数据集之间的区别，这些数据集实际上教授了我们希望模型学习的技能。

If model architecture defines *how* your model learns, then data defines *what* it learns, and no amount of compute or optimizer tuning can compensate for training on the wrong content. Moreover, getting the training data right isn't just about having good datasets. It's about assembling the right **mixture** : balancing conflicting objectives (like strong English vs. robust multilinguality) and tuning data proportions to align with our performance goals.

如果模型体系结构定义了模型的学习方式，那么数据就定义了它学习的内容，再多的计算或优化器调整也无法补偿对错误内容的训练。此外，获得正确的训练数据不仅仅是拥有良好的数据集。这是关于组装正确的组合：平衡相互冲突的目标（例如强大的英语与强大的多语言）并调整数据比例以符合我们的绩效目标。

This process is less about finding a universal best mix and more about asking the right questions and devising concrete plans to answer them:

这个过程不是寻找通用的最佳组合，而是提出正确的问题并制定具体的计划来回答这些问题：

- What do we want our model to be good at?

  我们希望我们的模型擅长什么？

- Which datasets are best for each domain and how do we mix them?

  哪些数据集最适合每个领域，我们如何混合它们？

- Do we have enough high-quality data for our target training scale?

  我们是否有足够的高质量数据来满足我们的目标训练规模？

This section is about navigating these questions using a mix of principled methods, ablation experiments, and a little bit of alchemy, to turn a pile of great datasets into a great training mixture.

本节是关于使用原则性方法、消融实验和一点炼金术的组合来解决这些问题，将一堆很棒的数据集变成一个很棒的训练混合物。

What's a good data mixture and why it matters most
什么是好的数据混合，以及为什么它最重要

We expect a lot from our language models, they should be able to help us write code, give us advice, answer questions about pretty much anything, complete tasks using tools, and more.

我们对语言模型寄予厚望，它们应该能够帮助我们编写代码、为我们提供建议、回答几乎任何事情的问题、使用工具完成任务等等。

Plentiful pre-training data sources like the web don't cover the full range of knowledge and capabilities needed for these tasks. As a result, recent models additionally rely on more specialized pre-training datasets that target specific domains like math and coding.

大量的预训练数据源（如网络）并不能涵盖这些任务所需的全部知识和能力。因此，最近的模型还依赖于针对数学和编码等特定领域的更专业的预训练数据集。

We have done a lot of past work on curating datasets, but for SmolLM3 we primarily made use of preexisting datasets.

我们过去在整理数据集方面做了很多工作，但对于 SmolLM3，我们主要利用了预先存在的数

据集。

To learn more about dataset curation, check out our reports on buildingFineWeb and FineWeb-Edu, FineWeb2, Stack-Edu, and FineMath.

要了解有关数据集管理的更多信息，请查看我们关于构建 FineWeb 和 FineWeb-Edu、FineWeb2、Stack-Edu 和 FineMath 的报告。

FineWeb2、Stack-Edu 和 FineMath 的报告。

THE UNINTUITIVE NATURE OF DATA MIXTURES
数据混合的不直观性质

If you're new to training language models, finding a good data mixture might seem straightforward: identify your target capabilities, gather high-quality datasets for each domain, and combine them.

如果您是训练语言模型的新手，找到一个好的数据组合可能看起来很简单：确定您的目标能力，为每个领域收集高质量的数据集，并将它们组合起来。

The reality is more complex, since some domains might compete with each other for your training budget. When focusing on some particular capability like coding, it can be tempting to upweight task-relevant data like source code.

现实情况更为复杂，因为某些领域可能会相互竞争您的培训预算。当专注于某些特定功能（如编码）时，可能很容易对与任务相关的数据（如源代码）进行加权。

However, upweighting one source implicitly downweights all of the other sources, which can harm the language model's capabilities in other settings. Training on a

collection of different sources therefore involves striking some kind of balance between downstream capabilities.

然而，提高一个源的权重会隐式降低所有其他源的权重，这可能会损害语言模型在其他设置中的能力。因此，对不同来源的集合进行训练涉及在下游功能之间取得某种平衡。

Additionally, across all of these sources and domains, there's often a subset of "high-quality" data that is especially helpful at improving the language model's capabilities. Why not just throw out all the lower quality data and train on the highest quality data only?

此外，在所有这些来源和领域中，通常有一个"高质量"数据的子集，这对于提高语言模型的功能特别有帮助。为什么不直接丢弃所有低质量数据，只使用最高质量的数据进行训练呢?

For SmolLM3's large training budget of 11T tokens, doing such extreme filtering would result in repeating data many times.

对于 SmolLM3 的 11T token 的大量训练预算，进行如此极端的过滤会导致数据多次重复。

Prior work has shown that this kind of repetition can be harmful(Muennighoff et al., 2025), so we should ideally be able to make use of higher and lower quality while still maximizing model performance.

先前的研究表明，这种重复可能是有害的（Muennighoff et al.， 2025），因此理想情况下，我们应该能够利用更高和更低的质量，同时仍然最大限度地提高模型性能。

To balance data across sources and make use of high-quality data, we need to source.

为了平衡跨来源的数据并利用高质量的数据，我们需要仔细设计混合：来自每个来源的训练文档的相对比例。

Since a language model's performance on some particular task or domain depends heavily on the amount of data it saw that is relevant to that task, tuning the mixing weights provides a direct way of balancing the model's capabilities across domains.

由于语言模型在某些特定任务或领域的性能在很大程度上取决于它看到的与该任务相关的数据量，因此调整混合权重提供了一种跨领域平衡模型功能的直接方法。

Because these trade-offs are model-dependent and difficult to predict, ablations are essential.

由于这些权衡依赖于模型且难以预测，因此消融至关重要。

But the mixture doesn't have to stay fixed throughout training. By adjusting the mixture as training progresses, what we call multi-stage training **** or curriculum, we can make better use of both high-quality and lower-quality data.

但混合物不必在整个训练过程中保持固定。通过随着培训的进行调整组合，我们称之为多阶段培训 **** 或课程，我们可以更好地利用高质量和低质量的数据。

THE EVOLUTION OF TRAINING CURRICULA
培训课程的演变

In the early days of large language model training, the standard approach was to fix a single data mixture for the entire training run. Models like GPT3 and early versions of Llama trained on a static mixture from start to finish. More recently, the field has shifted toward **multi-stage training** (Allal et al., 2025) where the data mixture changes over the course of training. The main motivation is that a language model's final behavior is strongly influenced by data seen toward the end of training (Y. Chen et al., 2025b). This insight enables a practical strategy: upweighting more plentiful sources early in training and mixing in smaller, higher quality sources towards the end.

在大型语言模型训练的早期，标准方法是在整个训练运行中修复单个数据混合。GPT3 等模型和早期版本的 Llama 从头到尾都在静态混合物上进行训练。最近，该领域已转向多阶段训练（Allal 等人，2025 年），其中数据混合在训练过程中发生变化。主要动机是语言模型的最终行为受到训练结束时看到的数据的强烈影响（Y. Chen 等人，2025b）。这种洞察力使我们能够制定一个实用的策略：在训练的早期加权更丰富的资源，并在训练结束时混合更小、更高质量的资源。

A common question is: how do you decide when to change the mixture? While there's no universal rule, but we typically follow these principles:

一个常见的问题是：您如何决定何时更换混合物？虽然没有通用规则，但我们通常遵循以下原
A common question is: how do you decide when to change the mixture? While there's no universal rule, but we typically follow these principles:

一个常见的问题是：您如何决定何时更换混合物？虽然没有通用规则，但我们通常遵循以下原则：

1. **Performance-driven interventions** : Monitor evaluation metrics on key benchmarks and adapt dataset mixtures to address specific capability bottlenecks. For example, if math performance plateaus while other capabilities continue

improving, that's a signal to introduce higher-quality math data.

绩效驱动的干预措施：监控关键基准的评估指标并调整数据集组合以解决特定的能力瓶颈。例如，如果数学性能趋于稳定，而其他功能继续提高，则表示需要引入更高质量的数学数据。

2. **Reserve high-quality data for late stages** : small high quality math and code datasets are most impactful when introduced during the annealing phase (final stage with learning rate decay).

   为后期阶段保留高质量数据：在退火阶段（学习率衰减的最后阶段）引入小型高质量数学和代码数据集时影响最大。

Now that we've established why mixtures matter and how curricula work, let's discuss how to tune both.

现在我们已经确定了为什么混合很重要以及课程如何运作，让我们讨论如何调整两者。

Ablation setup: how to systematically test data recipes
消融设置：如何系统地测试数据配方

When testing data mixtures, our approach is similar to how we run architecture ablations, with one difference: we try to run them at the target model scale.

在测试数据混合时，我们的方法类似于我们运行架构消融的方式，但有一个区别：我们尝试在目标模型规模上运行它们。

Small and large models have different capacities, for example a very small model might struggle to handle many languages, while a larger one can absorb them without sacrificing performance elsewhere.

小型和大型模型具有不同的容量，例如，非常小的模型可能难以处理多种语言，而较大的模型可以在不牺牲其他地方性能的情况下吸收它们。

Therefore running data ablations at too small a scale risks drawing the wrong conclusions about the optimal mix.

因此，以太小的规模运行数据消融可能会得出关于最佳组合的错误结论。

For SmolLM3, we ran our main data ablations directly on the 3B model, using shorter training runs of 50B and 100B tokens. We also used another type of ablation setup: ****** **annealing experiments** . Instead of training from scratch with different mixtures, we took an intermediate checkpoint from the main run (for example at 7T tokens) and continued training with modified data compositions.

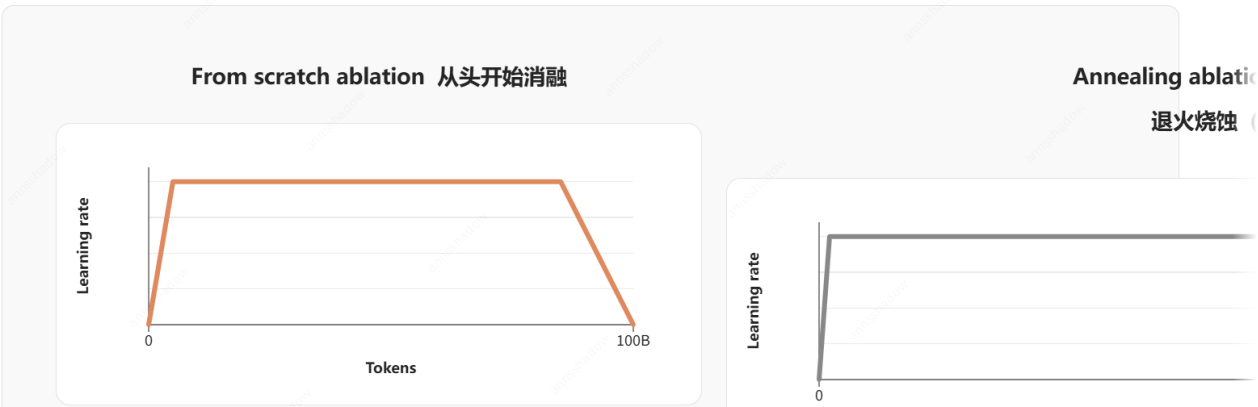
对于 SmolLM3，我们直接在 3B 模型上运行了主要数据消融，使用较短的 50B 和 100B 标记训练运行。我们还使用了另一种类型的烧蚀装置：******退火实验。我们没有使用不同的混合物从头开始训练，而是从主运行中获取一个中间检查点（例如在 7T 标记处），并继续使用修改后的数据组合进行训练。

This approach, allows us to test data mixture changes for doing multi-stage training (i.e changing the training mixture mid-training), and was used in recent work such as SmolLM2, Llama3 and Olmo2.

这种方法使我们能够测试数据混合变化以进行多阶段训练（即在训练中改变训练混合），并被用于最近的工作，例如 SmolLM2、Llama3 和 Olmo2。

For evaluation, we expanded our benchmark suite to include multilingual tasks alongside our standard English evaluations, ensuring we could properly assess the trade-offs between different language ratios.

在评估方面，我们扩展了基准套件，将多语言任务与标准英语评估一起纳入其中，确保我们能够正确评估不同语言比例之间的权衡。



Recent work has proposed automated approaches for finding optimal data proportions, including:

最近的工作提出了寻找最佳数据比例的自动化方法，包括：

- **DoReMi** (Xie et al., 2023): Uses a small proxy model to learn domain weights that minimize validation loss

  DoReMi（Xie et al.， 2023）：使用小型代理模型来学习域权重，从而最大限度地减少验证损失

- **Rho Loss** (Mindermann et al., 2022): Selects individual training points based on a holdout loss, prioritizing samples that are learnable, task-relevant, and not yet learned by the model

  Rho Loss（Mindermann 等人，2022 年）：根据保留损失选择单个训练点，优先考虑可学习、任务相关且模型尚未学习的样本

- **RegMix** (Q. Liu et al., 2025): Determines optimal data mixture proportions through regularized regression that balances performance across multiple evaluation objectives and data domains

  RegMix（Q. Liu 等人，2025 年）：通过正则化回归确定最佳数据混合比例，平衡多个评估目标和数据域的性能

We experimented with DoReMi and Rho Loss in past projects, but found they tend to converge toward distributions that roughly mirror the natural distribution of dataset sizes, essentially suggesting to use more of what we have more of.

我们在过去的项目中对 DoReMi 和 Rho Loss 进行了实验，但发现它们倾向于趋同于大致反映数据集大小自然分布的分布，本质上是建议使用更多我们拥有的更多内容。

While theoretically appealing, they didn't outperform careful manual ablations in our setting. Recent SOTA models still rely on manual mixture tuning through systematic ablations and annealing experiments, which is the approach we adopted for SmolLM3.

虽然理论上很有吸引力，但在我们的环境中，它们并没有优于仔细的手动消融。最近的 SOTA 模型仍然依赖于通过系统烧蚀和退火实验进行手动混合调谐，这是我们对 SmolLM3 采用的方法。

SmolLM3: Curating the data mixture (web, multilingual, math, code)

For SmolLM3, we wanted a model that can handle English and multiple other languages, and excel in math and code.

对于 SmolLM3，我们想要一个能够处理英语和多种其他语言，并且在数学和代码方面表现出色的模型。

These domains — web text, multilingual content, code and math — are common in most LLMs, but the process we'll describe here applies equally if you're training for a low-resource language or a specific domain such as finance or healthcare.

这些领域（Web 文本、多语言内容、代码和数学）在大多数 LLM 中都很常见，但如果您正在为资源匮乏的语言或特定领域（例如金融或医疗保健）进行培训，我们将在此处描述的过程同样适用。

The method is the same: identify good candidate datasets, run ablations, and design a mixture that balances all the target domains.

方法是相同的：确定好的候选数据集，运行消融，并设计平衡所有目标域的混合物。

We won't cover how to build high-quality datasets here, since we've already detailed that extensively in earlier work (FineWeb, FineWeb2, FineMath and Stack-Edu). Instead, this section focuses on how we *combine* those datasets into an effective pretraining mixture.

我们不会在这里介绍如何构建高质量的数据集，因为我们已经在早期的工作（FineWeb、FineWeb2、FineMath 和 Stack-Edu）中详细介绍了这一点。相反，本节重点介绍我们如何将这些数据集组合成有效的预训练混合物。

BUILDING ON PROVEN FOUNDATIONS

建立在经过验证的基础上

When it comes to pretraining data, the good news is that we rarely have to start from scratch. The open-source community has already built strong datasets for most common domains.

当谈到预训练数据时，好消息是我们很少需要从头开始。开源社区已经为大多数常见领域构建了强大的数据集。

Sometimes we will need to create something new — as we did with the Fine series (FineWeb, FineMath, etc.) — but more often, the challenge is in selecting and combining existing sources rather than reinventing them.

有时我们需要创造一些新的东西——就像我们对 Fine 系列（FineWeb、FineMath 等）所做的那样——但更多时候，挑战在于选择和组合现有资源，而不是重新发明它们。

That was our situation with SmolLM3. SmolLM2 had already established a strong recipe at 1.7B parameters for English web data, and identified the best math and code

这就是我们对 SmolLM3 的情况。SmolLM2 已经为英语网络数据建立了 1.7B 参数的强大配方，并确定了我们可以访问的最佳数学和代码数据集。

Our goal was to scale that success to 3B while adding the certain capabilities: robust multilinguality, stronger math reasoning, and better code generation.

我们的目标是将这一成功扩展到 3B，同时增加某些功能：强大的多语言、更强的数学推理和更好的代码生成。

ENGLISH WEB DATA: THE FOUNDATION LAYER
英文 WEB 数据：基础层

Web text forms the backbone of any general-purpose LLM, but quality matters as much as quantity.

Web 文本构成了任何通用 LLM 的支柱，但质量与数量一样重要。

From SmolLM3, we knew that FineWeb-Edu and DCLM were the strongest open English web datasets at the time of training. Together, they gave us 5.1T tokens of high-quality English web data. The question was: what's the optimal mixing ratio?

从 SmolLM3 中，我们知道 FineWeb-Edu 和 DCLM 是训练时最强的开放英语 Web 数据集。他们共同为我们提供了 5.1T 代币的高质量英文网络数据。问题是：最佳混合比例是多少？

FineWeb-Edu helps on educational and STEM benchmarks, while DCLM improves commonsense reasoning.

FineWeb-Edu 有助于教育和 STEM 基准测试，而 DCLM 则改进了常识性推理。

Following the SmolLM2 methodology, we ran a sweep on our 3B model over 100B tokens, testing ratios of 20/80, 40/60, 50/50, 60/40, and 80/20 (FineWeb-Edu/DCLM). Mixing them (around 60/40 or 50/50) gave the best trade-offs. We rerun the same ablations as the SmolLM2 paper on our 3B model trained on 100B tokens and found the same conclusion.

按照 SmolLM2 方法，我们对 3B 模型进行了 100B 代币的扫描，测试比率为 20/80、40/60、50/50、60/40 和 80/20 （FineWeb-Edu/DCLM）。将它们混合（大约 60/40 或 50/50）给出了最好的权衡。我们在 100B 标记上训练的 3B 模型上重新运行与 SmolLM2 论文相同的消融，并得出了相同的结论。

Using 60/40 or 50/50 provided the best balance across benchmarks, matching our SmolLM2 findings. We used 50/50 ratio ratio for Stage 1.

使用 60/40 或 50/50 提供了跨基准测试的最佳平衡，与我们的 SmolLM2 结果相匹配。我们在第一阶段使用了 50/50 的比率。

We also added other datasets like Pes2o, Wikipedia & Wikibooks and StackExchange, these datasets didn't have any impact on the performance but we included them to improve diversity.

我们还添加了其他数据集，如 Pes2o、维基百科和维基教科书以及 StackExchange，这些数据集对性能没有任何影响，但我们将它们包含在内以提高多样性。

MULTILINGUAL WEB DATA 多语言网络数据

For multilingual capability, we targeted 5 other languages: French, Spanish, German, Italian, and Portuguese. We selected them from FineWeb2-HQ, which gave us 628B tokens total.

对于多语言功能，我们瞄准了其他 5 种语言：法语、西班牙语、德语、意大利语和葡萄牙语。我们从 FineWeb2-HQ 中选择了它们，这总共给了我们 628B 代币。

We also included 10 other languages at smaller ratios, such as Chinese, Arabic, and Russian, not to target state of the art performance for them, but to allow people to easily do continual pretraining of SmolLM3 on them.

我们还以较小的比例纳入了其他 10 种语言，例如中文、阿拉伯语和俄语，不是为了为它们提供最先进的性能，而是为了让人们轻松地在它们上进行 SmolLM3 的持续预训练。

We used FineWeb2 for the languages not supported in FineWeb2-HQ.

对于 FineWeb2-HQ 不支持的语言，我们使用 FineWeb2。

The key question was: how much of our web data should be non-English? We know that more data a model sees in a language or domain, the better it gets at that language or domain.

关键问题是：我们的网络数据中有多少应该是非英语的？我们知道，模型在某种语言或领域中看到的数据越多，它在该语言或领域中的表现就越好。

The trade-off comes from our fixed compute budget: increasing data for one language means reducing data for the other languages including English.

权衡来自我们固定的计算预算：增加一种语言的数据意味着减少其他语言（包括英语）的数据。

Through ablations on the 3B model, we found that 12% multilingual content in the web mix struck the right balance, improving multilingual performance without degrading English benchmarks. This fit SmolLM3's expected usage, where English would remain the primary language.

通过对 3B 模型的消融，我们发现 Web 组合中 12% 的多语言内容达到了适当的平衡，在不降低英语基准测试的情况下提高了多语言性能。这符合 SmolLM3 的预期用法，其中英语仍将是主要语言。

It's also worth noting that with only 628B tokens of non-English data versus 5.

还值得注意的是，非英语数据只有 628B 个代币，而 5 个。

1T English tokens, going much higher would require doing more repetition of the multilingual data.

1T 英文代币，要走得更高，就需要对多语言数据进行更多重复。

CODE DATA 代码数据

Our code sources for Stage 1 are extracted from The Stack v2 and StarCoder2 training corpus:

我们第 1 阶段的代码源是从 The Stack v2 和 StarCoder2 训练语料库中提取的：

- The Stack v2 (16 languages) as our basis, filtered as StarCoder2Data.
  Stack v2（16 种语言）作为我们的基础，过滤为 StarCoder2Data。
- StarCoder2 GitHub pull requests for real-world code review reasoning.
  StarCoder2 GitHub 拉取请求，用于真实世界的代码审查推理。
- Jupyter and Kaggle notebooks for executable, step-by-step workflows.
  Jupyter 和 Kaggle 笔记本，用于可执行的分步工作流。
- GitHub issues and StackExchange threads for contextual discussions around code.
  GitHub 问题和 StackExchange 线程，用于围绕代码进行上下文讨论。

Aryabumi et al. (2024) highlight that code improves language models' performance beyond coding, for example on natural language reasoning and world knowledge and recommend using 25% code in the training mixture. Motivated by this, we started our ablations with 25% code in the mixture.

Aryabumi 等人（2024 年）强调，代码提高了语言模型的性能，超越了编码，例如在自然语言推理和世界知识方面，并建议在训练组合中使用 25% 的代码。受此启发，我们开始在混合物中使用 25% 的代码进行消融。

However, we observed significant degradation on English benchmarks (HellaSwag, ARC-C, MMLU).

然而，我们在英语基准（HellaSwag、ARC-C、MMLU）上观察到显着降级。

Reducing to 10% code, we didn't see improvements on our English benchmark suite compared to 0% code, but we included it anyway since code was a very important capability to have in the model.

减少到 10% 的代码，与 0% 的代码相比，我们没有看到英语基准测试套件的改进，但我们还是将其包括在内，因为代码是模型中非常重要的功能。

We delayed adding Stack-Edu — our educationally filtered subset of StarCoder2Data — until later stages, following the principle of staging high-quality data for maximum late-training impact.

我们推迟到后期阶段添加 Stack-Edu（我们经过教育过滤的 StarCoder2Data 子集），遵循分

级高质量数据以最大限度地影响后期训练的原则。

MATH DATA 数学数据

Math followed a similar philosophy to code. Early on, we used the larger, more general sets FineMath3+ and InfiWebMath3+ and later we upsampled FineMath4+ and InfiWebMath4+, and introduced new high quality datasets:

数学遵循与代码类似的哲学。早期，我们使用了更大、更通用的集合 FineMath3+ 和 InfiWebMath3+，后来我们上采样了 FineMath4+ 和 InfiWebMath4+，并引入了新的高质量数据集：

- MegaMath (Zhou et al., 2025)

  超级数学（周等人，2025 年）

- Instruction and reasoning datasets like OpenMathInstruct (Toshniwal et al., 2024) and OpenMathReasoning (Moshkov et al., 2025)

  指令和推理数据集，如 OpenMathInstruct （Toshniwal et al.， 2024） 和 OpenMathReasoning （Moshkov et al.， 2025）

We use 3% of math in Stage 1 equally split between FineMath3+ and InfiWebMath3+. With only 54B tokens available and an estimated 8T to 9T token Stage 1, using more than 3% math would require more than 5 epochs on the dataset.

我们在第一阶段使用 3% 的数学，在 FineMath3+ 和 InfiWebMath3+ 之间平分。由于只有 54B 个代币可用，并且估计第 1 阶段有 8T 到 9T 代币，因此使用超过 3% 的数学需要在数据集上花费超过 5 个 epoch。

FINDING THE RIGHT MIXTURE FOR NEW STAGES
为新阶段寻找合适的混合物

While we ran ablations from scratch to determine the stage 1 mixture, to test new datasets for new stages (in our case two new stages) we used annealing ablations: we took a checkpoint at around 7T tokens (late in stage 1) and ran a 50B token annealing experiments with the following setup:

datasets for new stages (in our case two new stages) we used annealing ablations: we took a checkpoint at around 7T tokens (late in stage 1) and ran a 50B token annealing experiments with the following setup:

虽然我们从头开始运行消融以确定第 1 阶段的混合物，但为了测试新阶段的新数据集（在我们的例子中是两个新阶段），我们使用了退火消融：我们在大约 7T 标记（第 1 阶段后期）取了一个检查点，并使用以下设置运行了 50B 标记退火实验：

- **40% baseline mixture** : The exact stage 1 mixture we'd been training on

  40% 基线混合物：我们一直在训练的确切的第 1 阶段混合物

- **60% new dataset** : The candidate dataset we wanted to evaluate

  60% 新数据集：我们想要评估的候选数据集

For example, to test whether MegaMath would improve our math performance, we ran 40% Stage 1 mixture (maintaining the 75/12/10/3 domain split) and 60% MegaMath.

例如，为了测试 MegaMath 是否会提高我们的数学表现，我们运行了 40% 的第 1 阶段混合（保持 75/12/10/3 域拆分）和 60% 的 MegaMath。

The can find the composition of the 3 stages in the following section.

可以在下一节中找到 3 个阶段的组成。

With our data carefully curated and our mixture validated through ablations, we're ready to embark on the actual training journey.

经过精心策划的数据并通过消融验证我们的混合物，我们已准备好开始实际的训练之旅。

The chapter that follows is the story of SmolLM3's month-long training run: the preparation, the unexpected challenges, and the lessons learned along the way.

接下来的章节是 SmolLM3 为期一个月的训练运行的故事：准备工作、意想不到的挑战以及一路上学到的教训。

The training marathon 训练马拉松

You've made it this far, congrats! The real fun is about to begin.

你已经走到了这一步，恭喜你！真正的乐趣即将开始。

At this point, we have everything in place: a validated architecture, a finalized data mixture, and tuned hyperparameters. The only thing left is setting up the infrastructure and hitting "train".

在这一点上，我们已经准备好了一切：经过验证的架构、最终的数据混合和调整的超参数。唯一剩下的就是建立基础设施并点击"火车"。

For SmolLM3, we trained on 384 H100 GPUs (48 nodes) for nearly a month, processing 11 trillion tokens. This section walks you through what actually happens during a long training run: the pre-flight checks, the inevitable surprises, and how we kept things stable.

对于 SmolLM3，我们在 384 个 H100 GPU（48 个节点）上训练了近一个月，处理了 11 万亿个代币。本节将引导您了解长时间训练期间实际发生的情况：飞行前检查、不可避免的意外以及我们如何保持稳定。

You'll see firsthand why both solid ablation practices and reliable infrastructure matter.

您将亲眼目睹为什么可靠的消融实践和可靠的基础设施都很重要。

We cover the technical infrastructure details of GPU hardware, storage systems, and optimizing throughputs in thefinal chapter.

我们在最后一章中介绍了 GPU 硬件、存储系统和优化吞吐量的技术基础设施细节。

Our team has been through this many times: from StarCoder and StarCoder2, to SmolLM, SmolLM2, and now SmolLM3. Every single run is different. Even if you've trained a dozen models, each new run finds a fresh way to surprise you.

我们的团队已经经历了很多次：从 StarCoder 和 StarCoder2，到 SmolLM、SmolLM2，再到现在的 SmolLM3。每一次运行都是不同的。即使您已经训练了十几个模型，每次新的运行都会找到一种新的方式来给您带来惊喜。

This section is about stacking the odds in your favour so you're ready for those surprises.

本节是关于增加对您有利的几率，以便您为这些惊喜做好准备。

Pre-flight checklist: what to verify before hitting "train"
飞行前清单：在"火车"之前要验证什么

Before hitting "train", we go through a checklist to ensure everything works end-to-end:

在点击"训练"之前，我们会检查一份清单，以确保一切端到端地工作：

Infrastructure readiness:

基础设施就绪情况：

- If your cluster supports Slurm reservations, use them. For SmolLM3, we had a fixed 48-node reservation for the entire run. That meant no queueing delays, consistent throughput, and the ability to track node health over time.

  如果您的集群支持 Slurm 预留，请使用它们。对于 SmolLM3，我们在整个运行中保留了固定的 48 个节点。这意味着没有排队延迟、一致的吞吐量以及随着时间的推移跟踪节点运行状况的能力。

- Stress-test GPUs before launch (we use GPU Fryer and DCGM Diagnostics) to catch throttling or performance degradation. For SmolLM3, we found two GPUs throttling and replaced them before starting the run.

  在启动前对 GPU 进行压力测试（我们使用 GPU Fryer 和 DCGM Diagnostics）以捕获限制或性能下降。对于 SmolLM3，我们发现两个 GPU 受到限制，并在开始运行之前更换了它们。

- Avoid storage bloat: our system uploads each checkpoint to S3, then deletes the local copy right after saving the next one, so we never store more than one on the fast local GPU SSDs.

  避免存储膨胀：我们的系统将每个检查点上传到 S3，然后在保存下一个检查点后立即删除本地副本，因此我们永远不会在快速本地 GPU SSD 上存储多个。

Evaluation setup: Evaluations are deceptively time-consuming. Even with everything implemented, running them manually, logging results, and making plots can eat up hours each time. So try to automate them completely, and ensure they are running

and logging correctly before the run starts.

评估设置：评估看似耗时。即使实现了所有内容，手动运行它们、记录结果和制作绘图每次也会占用数小时。因此，请尝试将它们完全自动化，并确保它们在运行开始之前正确运行和记录。

For SmolLM3, every saved checkpoint automatically triggered an evaluation job on the cluster that got logged to Wandb and Trackio.

对于 SmolLM3，每个保存的检查点都会自动触发集群上的评估作业，该作业记录到 Wandb 和 Trackio。

Checkpoint & auto-resume system: Verify that checkpoints are saved correctly and that the training job can resume from the latest one without manual intervention. On Slurm, we use `--requeue` option so a failed job gets automatically relaunched, resuming from the most recent checkpoint.

检查点和自动恢复系统：验证检查点是否已正确保存，以及训练作业是否可以从最新的检查点恢复，而无需手动干预。在 Slurm 上，我们使用 `--requeue` 选项，以便失败的作业会自动重新启动，并从最近的检查点恢复。

Metrics logging: Confirm that you're logging all the metrics you care about: evaluation scores, throughput (tokens/sec), training loss, gradient norm, node health (GPU utilization, temperature, memory usage), and any custom debug metrics specific to your run.

指标日志记录：确认你正在记录你关心的所有指标：评估分数、吞吐量（令牌/秒）、训练损失、梯度范数、节点运行状况（GPU 利用率、温度、内存使用情况）以及特定于运行的任何自定义调试指标。

Training configuration sanity check: Double-check your training config, launch scripts, and Slurm submission commands.

训练配置健全性检查：仔细检查训练配置、启动脚本和 Slurm 提交命令。

> **Infrastructure deep-dive**
>
> **基础设施深入探讨**
>
> For detailed guidance on GPU testing, storage benchmarking, monitoring setup, and building resilient training systems, see the Infrastructure chapter.
>
> 有关 GPU 测试、存储基准测试、监控设置和构建弹性训练系统的详细指南，请参阅基础设施一章。

Scaling surprises 扩展惊喜

After running extensive ablations for SmolLM3, we were ready for the full-scale run. Our 3B ablations on 100B tokens looked promising. The architectural changes compared to SmolLM2 (detailed in Architecture Choices: GQA, NoPE, document masking, tokenizer) either improved or maintained performance, and we found a good data mixture that balances English, multilingual, code, and math performance (see The art of data curation. We optimized our configuration for around 30% MFU on 384 GPUS (48 nodes).

We ran some ablations on 48 nodes to validate the throughput before launching the run. You can find more details about them in the Infrastructure chapter.

我们在 48 个节点上运行了一些烧蚀，以在启动运行之前验证吞吐量。您可以在基础设施一章中找到有关它们的更多详细信息。

在对 SmolLM3 进行大量消融后，我们已准备好进行全面运行。我们对 100B 代币的 3B 消融看起来很有希望。与 SmolLM2 相比，架构变化（详见架构选择：GQA、NoPE、文档屏蔽、分词器）提高了或保持了性能，我们发现了一个很好的数据混合，可以平衡英语、多语言、代码和数学性能（参见数据管理的艺术。我们优化了配置，在 30 个 GPU（384 个节点）上实现了大约 48% 的 MFU。

We were ready for the big one: 11T tokens. That's when reality started throwing curveballs.

我们已经为大事件做好了准备：11T 代币。从那时起，现实开始抛出曲线球。

MYSTERY #1 – THE VANISHING THROUGHPUT
谜团 #1 – 消失的吞吐量

Within hours of launch, throughput plummeted. It was a big jump with repeated sharp drops.

在发布后的几个小时内，吞吐量直线下降。这是一次反复急剧下降的大跳跃。

> 📍 **Why throughput matters 为什么吞吐量很重要**
>
> Throughput measures how many tokens per second our system processes during

training. It directly impacts our training time, a 50% drop in throughput means our month-long run becomes a two-month run.

吞吐量衡量我们的系统在训练期间每秒处理多少令牌。它直接影响我们的训练时间，吞吐量下降 50% 意味着我们长达一个月的运行变成了两个月的运行。

In the Infrastructure chapter, we'll show how we optimized throughput for SmolLM3 before starting the run.

在基础设施一章中，我们将展示如何在开始运行之前优化 SmolLM3 的吞吐量。

This didn't happen in any ablation run, so what changed? Three things:

这在任何消融运行中都没有发生，那么发生了什么变化呢？三件事：

1. Hardware state can change over time. GPUs that worked fine in ablations might fail and network connections might degrade under sustained load.

   硬件状态可能会随时间而变化。在消融中工作正常的 GPU 可能会发生故障，并且网络连接可能会在持续负载下降级。

2. The size of the training datasets. We now used the full ~24 TB training dataset instead of the smaller subsets from ablations, though the data sources themselves were the same.

   训练数据集的大小。我们现在使用完整的 ~24 TB 训练数据集，而不是来自消融的较小子集，尽管数据源本身是相同的。

3. The number of training steps. We set the real step count for 11T tokens instead of the short 100B-token ablation horizon.

   训练步骤数。我们为 11T 代币设置了实际步数，而不是短暂的 100B 代币消融范围。

Everything else remained exactly the same as in the throughput ablations: number of nodes, dataloader configuration, model layout, and parallelism setup…

其他一切都与吞吐量消融完全相同：节点数量、数据加载器配置、模型布局和并行度设置……

Intuitively, neither dataset size nor step count should cause throughput drops, so we naturally suspected hardware issues first. We checked our node monitoring metrics, which showed that the big throughput jump correlated with spikes in disk read latency.

直观地说，数据集大小和步数都不会导致吞吐量下降，因此我们自然会首先怀疑硬件问题。我们检查了节点监控指标，结果显示吞吐量的大幅跳跃与磁盘读取延迟的峰值相关。

That pointed us straight at our data storage.

这直接指向了我们的数据存储。

> 📍 **Storage options in our cluster**
>
> **我们集群中的存储选项**
>
> Our cluster has three storage tiers for training data:
>
> 我们的集群有三个用于训练数据的存储层：
>
> - **FSx** : Network-attached storage which uses Weka, a "keep-hot" caching model that stores frequently accessed files locally and evicts inactive "cold" files to S3 as capacity fills up.
>
>   FSx：使用 Weka 的网络附加存储，Weka 是一种"保持热"缓存模型，将经常访问的文件存储在本地，并在容量填满时将非活动的"冷"文件逐出到 S3。
>
> - **Scratch (Local NVMe RAID)** : Fast local storage on each node (8×3.5TB NVMe drives in RAID), which is faster than FSx but limited to local node access.

Scratch（本地 NVMe RAID）：每个节点上的快速本地存储（RAID 中的 8×3.5TB NVMe 驱动器），比 FSx 更快，但仅限于本地节点访问。

- **S3** : Remote object storage for cold data and backups.

  S3：用于冷数据和备份的远程对象存储。

You can find more details in the Infrastructure chapter.

您可以在基础设施一章中找到更多详细信息。

For SmolLM3's 24TB dataset, we initially stored the data in FSx (Weka). With 24TB of training data, on top of storage already used by several other teams, we were pushing Weka's storage to the limit.

对于 SmolLM3 的 24TB 数据集，我们最初将数据存储在 FSx （Weka） 中。凭借 24TB 的训练数据，加上其他几个团队已经使用的存储空间，我们将 Weka 的存储空间推向了极限。

So it started evicting dataset shards mid-training, which meant we had to fetch them back, creating stalls, which explained the big throughput jump.

因此，它开始在训练过程中驱逐数据集分片，这意味着我们必须将它们取回，从而产生停滞，这解释了吞吐量的大幅增长。

Worse: there was no way to pin our dataset folders as hot for the full training.

更糟糕的是：无法将我们的数据集文件夹固定为热，以进行完整训练。

Fix #1 – Changing data storage

修复 #1 – 更改数据存储

We didn't find a way to pin our dataset folders as hot for the full training in Weka, so we tried to change the storage method. Streaming directly from S3 was slow, so we decided to store the data in each node in its local storage `/scratch` .

我们没有找到一种方法将我们的数据集文件夹固定为热，以便在 Weka 中进行完整训练，因此我们尝试更改存储方法。直接从 S3 流式传输速度很慢，因此我们决定将每个节点中的数据存储在其本地存储 `/scratch` 中。

This came with a catch: If a node died and was replaced, the new replacement GPUs had no data. Downloading 24TB from S3 with `s5cmd` took 3h. We cut that to 1h30 by copying from another healthy node using `fpsync` instead of going through S3. This was faster given all the nodes were in the same datacenter.

这有一个问题：如果一个节点死机并被替换，新的替换 GPU 就没有数据。从 S24 下载 3TB `s5cmd` 需要 3 小时。我们通过使用而不是通过 S3 从 `fpsync` 另一个健康节点复制，将其缩短到 1 小时 30 分。考虑到所有节点都位于同一个数据中心，这速度更快。

Still, 1h30 of downtime per node failure, and the need for manually copying the data to the new node immediately, was painful. The hack that finally made it bearable: reserve a spare node in our Slurm reservation with the dataset preloaded.

尽管如此，每个节点故障需要 1 小时 30 分的停机时间，并且需要立即手动将数据复制到新节点，这还是很痛苦的。最终让它变得可以忍受的黑客：在我们的 Slurm 预留中预留一个备用节点，并预加载数据集。

If a node died, we swapped it instantly with the spare node, so zero recovery delay. While idle, the spare ran evals or dev jobs, so it wasn't wasted.

如果一个节点死了，我们会立即将其与备用节点交换，因此零恢复延迟。闲置时，备用运行评估或开发作业，因此不会浪费。

This fixed Mystery #1... or so we thought.

这个修复的谜团 #1...至少我们是这么想的。

MYSTERY #2 – THE PERSISTING THROUGHPUT DROPS

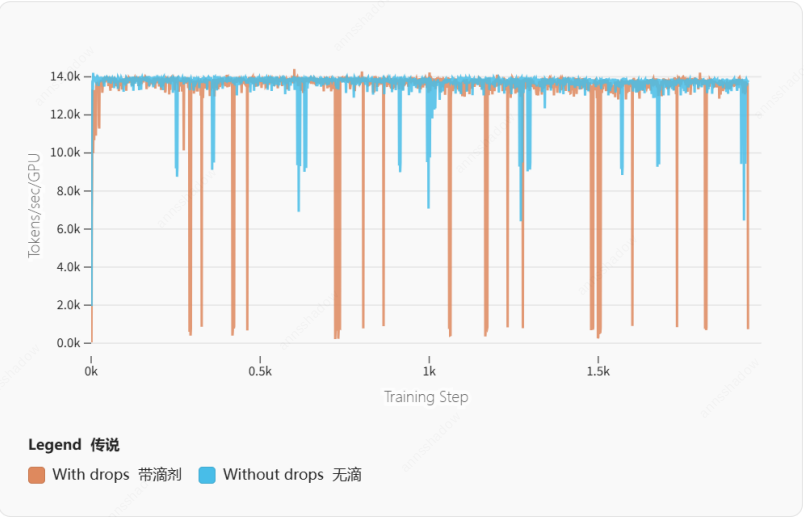
谜团 #2 – 吞吐量持续下降

Even after moving to scratch, the individual throughput drops kept happening although we didn't find any anomaly in the hardware monitoring metrics.

即使在转向从头开始之后，尽管我们在硬件监控指标中没有发现任何异常，但单个吞吐量下降仍在继续发生。

The chart below compares the throughput we got, after fixing the storage issue in orange, to the throughput we were getting during the ablations in blue. As you can see, the drops became much sharper.

下图将修复存储问题后获得的吞吐量（橙色）与我们在消融期间获得的吞吐量（蓝色）进行了比较。如您所见，水滴变得更加尖锐。



Still suspecting hardware, we decided to test on fewer nodes. With 384 GPUs, there's a high chance something could be failing. Surprisingly, we could reproduce the exact same throughput drops on a single node, no matter which specific node we tested.

仍然怀疑硬件，我们决定在更少的节点上进行测试。使用 384 个 GPU，很有可能会出现故障。令人惊讶的是，无论我们测试哪个特定节点，我们都可以在单个节点上重现完全相同的吞吐量下降。

This ruled out hardware issues.

这排除了硬件问题。

Remember the three things that changed from our ablations? We had already addressed the data storage issue by moving to local node storage. Hardware was now eliminated. That left only one variable: the step count.

还记得消融改变的三件事吗？我们已经通过迁移到本地节点存储来解决数据存储问题。硬件现在被淘汰了。这只剩下一个变量：步数。

We tested this by rolling back to smaller step counts (from 3M to 32k) and the thoughput drops became smaller! Larger step counts produced sharper, more frequent drops.

我们通过回滚到更小的步数（从 3M 到 32k）来测试这一点，并且推杆下降变得更小！更大的步数会产生更尖锐、更频繁的下降。

To test this, we ran identical configurations with only the training steps changed from 32k to 3.2M. You can see the exact configs we used:

为了测试这一点，我们运行了相同的配置，仅将训练步骤从 32k 更改为 3.2M。您可以看到我们使用的确切配置：

```
## Short run (32k steps)
- "lr_decay_starting_step": 2560000
- "lr_decay_steps": 640000
- "train_steps": 3200000
Â
## Long run (3.2M steps)
+ "lr_decay_starting_step": 26000
+ "lr_decay_steps": 6000
+ "train_steps": 32000
Â
```

The results shown in the figure below were clear: shorter runs had small throughput drops, while longer step counts produced sharper, more frequent drops. So the issue was not the hardware, but a software bottleneck, likely in the dataloader!

下图所示的结果很明确：较短的运行具有较小的吞吐量下降，而较长的步数产生更尖锐、更频繁的下降。所以问题不在于硬件，而在于软件瓶颈，可能出在数据加载器中！

Given that most other training components process each batch identically regardless of step count.

鉴于大多数其他训练组件以相同的方式处理每个批次，而不管步数如何。

Legend 传说

CSV load error: CSV file not found

That's when we realized we'd never actually done large-scale pretraining with nanotron's dataloader. SmolLM2 had been trained with steady throughput using a Megatron-LM derived dataloader (TokenizedBytes) through an internal wrapper around nanotron. For SmolLM3, we switched to nanotron's built-in dataloader (`nanosets`).

那时我们意识到我们从未真正使用 nanotron 的数据加载器进行过大规模预训练。SmolLM2 已通过围绕 nanotron 的内部包装器使用 Megatron-LM 衍生的数据加载器（TokenizedBytes）以稳定的吞吐量进行训练。对于 SmolLM3，我们切换到 nanotron 的内置数据加载器 （ `nanosets` ）。

After deep diving into its implementation, we found that it was naively building one giant index that grew with each training step. For very large steps, this caused a higher shared memory which triggered throughput drops.

在深入研究其实施后，我们发现它正在天真地构建一个随着每个训练步骤而增长的巨型索引。对于非常大的步骤，这会导致更高的共享内存，从而触发吞吐量下降。

Fix #2 – Bring in TokenizedBytes dataloader

修复 #2 – 引入 TokenizedBytes 数据加载器

To confirm that the dataloader was indeed the culprit, we launched the same

configuration with our internal SmolLM2 framework using `TokenizedBytes` dataloader. No drops. Even on 48 nodes using the same datasets.

为了确认数据加载器确实是罪魁祸首，我们使用数据加载器在 `TokenizedBytes` 内部 SmolLM2 框架中启动了相同的配置。没有水滴。即使在使用相同数据集的 48 个节点上也是如此。

Fastest path forward: copy this dataloader into nanotron. The drops were gone and the throughput back to target.

最快的前进路径：将此数据加载器复制到 nanotron 中。下降消失了，吞吐量又回到了目标。

We were ready to relaunch... until the next curveball.

我们准备重新启动……直到下一个曲线球。

MYSTERY #3 – THE NOISY LOSS

谜团 #3 – 嘈杂的损失

With the new dataloader, we didn't have throughput drops but the loss curve looked more noisy.

使用新的数据加载器，我们没有吞吐量下降，但损失曲线看起来更嘈杂。

`nanosets` had been producing smoother loss, and the difference rang a bell from an old debugging war: a few years ago, we'd found a shuffling bug in our pretraining code where documents were shuffled, but sequences inside a batch were not, leading to small spikes.

`nanosets` 一直在产生更平滑的损失，这种差异敲响了旧调试战争的警钟：几年前，我们在预训练代码中发现了一个随机错误，文档被打乱，但批次内的序列没有，导致小峰值。

Checking our new dataloader confirmed it: it was reading sequences sequentially from each document. That's fine for short files, but with domains like code, a single long low-quality file can fill an entire batch and cause loss spikes.

检查我们的新数据加载器证实了这一点：它从每个文档中按顺序读取序列。这对于短文件来说很好，但对于像代码这样的域，一个长的低质量文件可能会填满整个批次并导致丢失峰值。

Fix #3 – Shuffle at the sequence level

修复 #3 – 在序列级别随机播放

We had two options:

我们有两个选择：

1. Change the dataloader to do random access (risk: higher memory usage).

   更改数据加载器以执行随机访问（风险：内存使用率更高）。

2. Pre-shuffle tokenized sequences offline.

   离线预随机播放标记化序列。

With the time pressure to start the run and our cluster reservation running, we went with option #2 as the safer, faster fix. Tokenized data was already on each node, so reshuffling locally was cheap (~1 h).

由于开始运行的时间压力和集群预留的运行，我们选择了选项 #2 作为更安全、更快的修复方法。每个节点上都已经有代币化数据，因此在本地重新洗牌很便宜（~1 小时）。

We also generated shuffled sequences for each epoch with different seeds to avoid repeating shuffling patterns across epochs.

我们还为每个纪元生成了具有不同种子的洗牌序列，以避免跨纪元重复洗牌模式。

> **Know when to patch vs. fix**
>
> **了解何时修补与修复**
>
> When facing urgent deadlines, it might be faster to adopt a proven solution or quick workaround than to debug your own broken implementation. Earlier, we plugged in TokenizedBytes dataloader rather than fixing nanosets' index implementation.
>
> 当面临紧急的最后期限时，采用经过验证的解决方案或快速解决方法可能比调试自己损坏的实现更快。早些时候，我们插入了 TokenizedBytes 数据加载器，而不是修复 nanoset 的索引实现。
>
> Here, we chose offline pre-shuffling over dataloader changes. But know when to take shortcuts, or you'll end up with a patchwork system that's hard to maintain or optimize.
>
> 在这里，我们选择了离线预洗牌而不是数据加载器更改。但要知道何时走捷径，否则你最终会

得到一个难以维护或优化的拼凑系统。

LAUNCH, TAKE TWO 发射，取两个

By now we had:

到目前为止，我们已经：

- **Stable throughput** (scratch storage + spare node strategy)
  稳定的吞吐量（暂存 + 备用节点策略）
- **No step-count-induced drops** (`TokenizedBytes` dataloader)
  无步数引起的下降（ `TokenizedBytes` 数据加载器）
- **Clean, sequence-level shuffling** (offline pre-shuffle per epoch)
  干净的序列级随机播放（每个 epoch 的离线预随机播放）

We relaunched. This time, everything held. The loss curve was smooth, throughput was consistent, and we could finally focus on training instead of firefighting.

我们重新启动。这一次，一切都成立了。损失曲线平滑，吞吐量一致，我们终于可以专注于训练而不是消防。

Mystery #4 – Unsatisfactory performance

谜团 #4 – 表现不理想

After fixing the throughput and dataloader issues, we launched the run again and trained smoothly for the first two days. Throughput was stable, loss curves looked as expected, and nothing in the logs suggested any problems.

在修复吞吐量和数据加载器问题后，我们再次启动运行，并在前两天顺利训练。吞吐量稳定，损失曲线看起来符合预期，日志中没有任何内容表明有任何问题。

At around the 1T token mark, however, the evaluations revealed something unexpected.

然而，在 1T 代币大关附近，评估揭示了一些意想不到的东西。

As part of our monitoring, we evaluate intermediate checkpoints and compare them to historical runs. For instance, we had the intermediate checkpoints from SmolLM2 (1.7B) trained on a similar recipe, so we could track how both models progressed at the same stages of training. The results were puzzling: despite having more parameters and a better data mixture, the 3B model was performing worse than the 1.

作为监控的一部分，我们评估中间检查点并将其与历史运行进行比较。例如，我们让 SmolLM2 （1.7B） 的中间检查点在类似的配方上进行了训练，因此我们可以跟踪两个模型在相同训练阶段的进展情况。结果令人费解：尽管具有更多参数和更好的数据混合，但 3B 模型的性能比 1 差。

7B at the same training point. Loss was still decreasing, and benchmark scores were improving, but the improvement rate was clearly below expectations.

7B 在同一训练点。亏损仍在减少，基准分数在提高，但改善速度明显低于预期。

Given that we had thoroughly tested every architecture and data change introduced in SmolLM3 compared to SmolLM2, we validated the training framework and there were only a few remaining untested differences between the two training setups. The most obvious was tensor parallelism.

鉴于我们已经彻底测试了 SmolLM3 中引入的每个架构和数据更改，与 SmolLM2 相比，我们验证了训练框架，并且两种训练设置之间只有一些未测试的差异。最明显的是张量并行性。

SmolLM2 could fit on a single GPU and was trained without TP, while SmolLM3 required TP=2 to fit in memory. We didn't suspect it or think of testing it before, since TP was used in the 3B ablations and their results made sense.

SmolLM2 可以安装在单个 GPU 上，并且无需 TP 即可进行训练，而 SmolLM3 需要 TP=2 才能适应内存。我们之前没有怀疑过它，也没有想过测试它，因为 TP 被用于 3B 消融，而且它们的结果是有意义的。

Fix #4 - The final fix

修复 #4 - 最终修复

To test the TP bug hypothesis, we trained a 1.7B model with the exact same setup as SmolLM3 — same architecture changes (document masking, NoPE), same data mixture, same hyperparameters — both with and without TP.

为了检验 TP 错误假设，我们训练了一个 1.7B 模型，其设置与 SmolLM3 完全相同——相同的架构更改（文档屏蔽，NoPE）、相同的数据混合、相同的超参数——无论有没有 TP。

The difference was immediate: the TP version consistently had a higher loss and lower downstream performance than the non-TP version.

差异是立竿见影的：TP 版本始终比非 TP 版本具有更高的损耗和更低的下游性能。

That confirmed we were looking at a TP-related bug.

这证实了我们正在研究一个与 TP 相关的错误。

We then examined the TP implementation in detail, comparing weights from TP and non-TP runs. The problem turned out to be subtle but significant: we were using identical random seeds across all TP ranks, when each rank should have been initialised with a different seed.

然后，我们详细检查了 TP 实现，比较了 TP 和非 TP 运行的权重。事实证明，这个问题很微妙，但很重要：我们在所有 TP 等级中使用相同的随机种子，而每个等级都应该用不同的种子初始化。

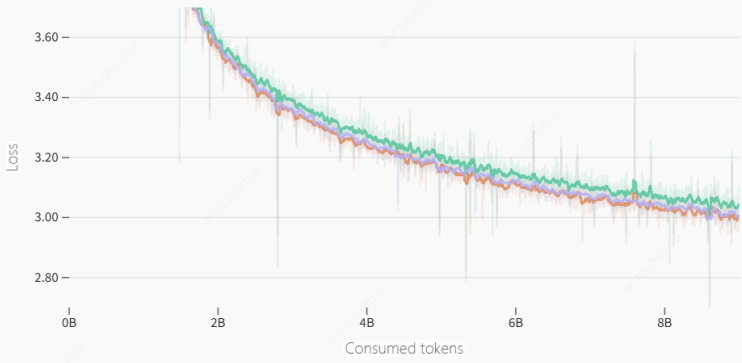
This caused correlated weight initialisation across shards, which affected convergence. The effect was not catastrophic — the model still trained and improved — but it introduced enough inefficiency to explain the gap we observed at scale.

这导致分片之间相关的权重初始化，从而影响收敛。这种影响并不是灾难性的——模型仍在训练和改进——但它引入了足够的低效率来解释我们在大规模上观察到的差距。

Below is the bug fix:

以下是错误修复：

```
diff --git a/src/nanotron/trainer.py b/src/nanotron/trainer.py
index 1234567..abcdefg 100644
--- a/src/nanotron/trainer.py
+++ b/src/nanotron/trainer.py
@@ -185,7 +185,10 @@ class DistributedTrainer:
     ):
         # Set random states
-        set_random_seed(self.config.general.seed)
+        # Set different random seed for each TP rank to ensure diversity
+        tp_rank = dist.get_rank(self.parallel_context.tp_pg)
+        set_random_seed(self.config.general.seed + tp_rank)
Â
+
```



Once we fixed the seeds so that each TP rank used a different seed, we repeated the ablations experiments and confirmed that TP and non-TP runs now matched in both loss curves and downstream performance.

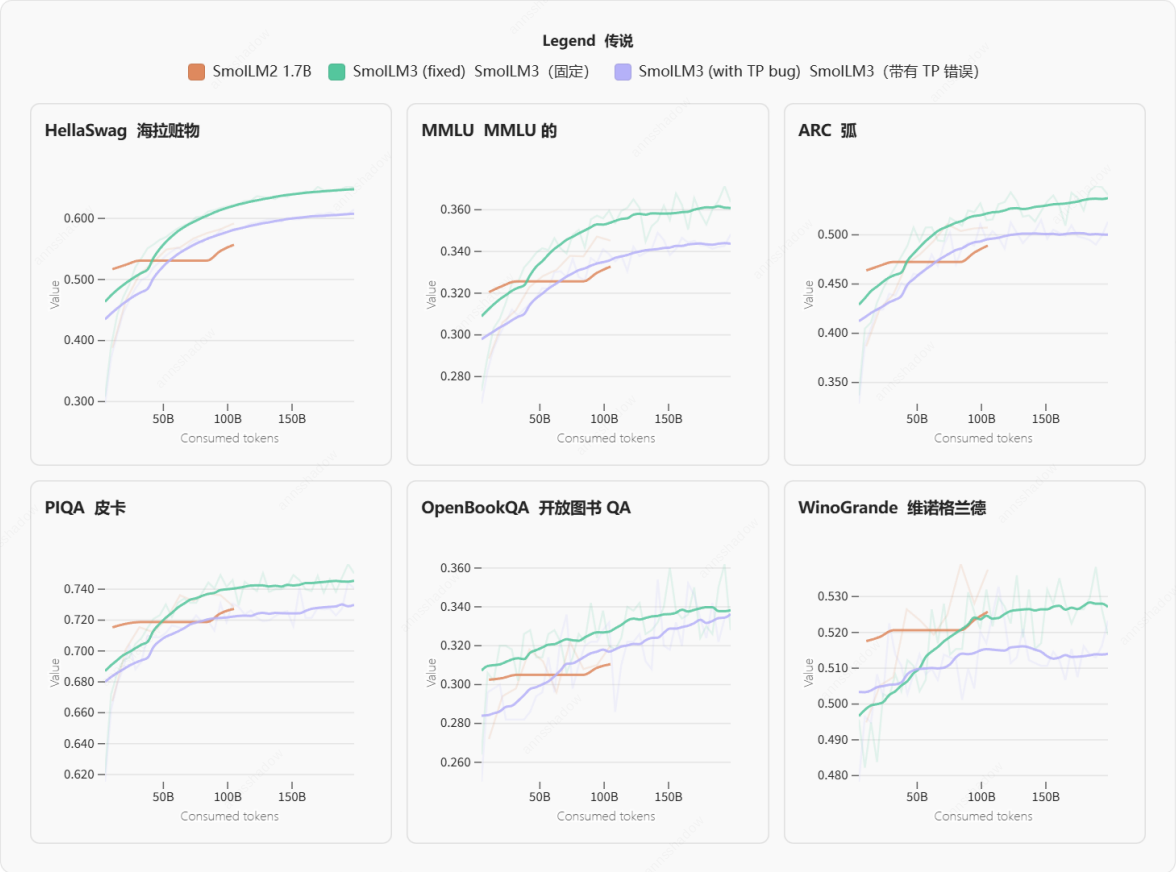
一旦我们固定了种子，使每个 TP 等级使用不同的种子，我们重复了消融实验，并确认 TP 和非 TP 运行现在在损失曲线和下游性能方面都匹配。

To make sure there were no other hidden issues, we ran additional sanity checks: a SmolLM2-style (architecture and data wise) run at 3B parameters, and a separate SmolLM3 run at 3B parameters, comparing both to SmolLM2's checkpoints.

为了确保没有其他隐藏问题，我们运行了额外的健全性检查：以 3B 参数运行的 SmolLM2 风格（架构和数据方面）和以 3B 参数运行的单独 SmolLM3，将两者与 SmolLM2 的检查点进行比较。

The results now aligned with expectations: the 1.7B SmolLM2 performed worse than the 3B SmolLM2 variant, which in turn was below SmolLM3's 3B performance.

现在的结果符合预期：1.7B SmolLM2 的性能比 3B SmolLM2 变体差，而 3B SmolLM2 变体又低于 SmolLM3 的 3B 性能。



This debugging process reinforced one of the core principles we outlined earlier in this blog:

此调试过程强化了我们在本博客前面概述的核心原则之一：

“The real value of a solid ablation setup goes beyond just building a good model.

“可靠的消融装置的真正价值不仅仅是构建一个好的模型。

When things inevitably go wrong during our main training run (and they will, no matter how much we prepare), we want to be confident in every decision we made and quickly identify which components weren’t properly tested and could be causing the issues.

当在我们的主要训练运行中不可避免地出现问题时（无论我们准备多少，它们都会出错），我们希望对我们所做的每一个决定充满信心，并快速确定哪些组件没有经过适当的测试并可能导致问题。

This preparation saves debugging time and keeps our sanity intact. There’s nothing worse than staring at a mysterious training failure with no idea where the bug could be hiding.

这种准备工作节省了调试时间并保持了我们的理智完好无损。没有什么比盯着一个神秘的训练失败而不知道虫子可能藏在哪里更糟糕的了。

”

Because every other component in our training had been validated, we could pinpoint TP as the only plausible cause and fix the bug within a single day of detecting the performance gap.

由于我们训练中的所有其他组件都已经过验证，因此我们可以将 TP 确定为唯一合理的原因，并在检测到性能差距后的一天内修复错误。

With that, we had resolved the last in a series of unexpected issues that had surfaced since launch.

至此，我们解决了自发布以来出现的一系列意外问题中的最后一个。

Third time’s a charm, from that point on, the remaining month of training was relatively uneventful, just the steady work of turning trillions of tokens into a finished model, interrupted by occasional restarts due to node failures.

第三次是魅力，从那时起，剩下的一个月训练相对平淡无奇，只是将数万亿个代币转化为成品模型的稳定工作，偶尔会因节点故障而重启。

Staying the course 坚持到底

As the previous section showed, scaling from ablations to full pretraining wasn't just "plug and play." It brought unexpected challenges, but we successfully identified and resolved each issue.

正如上一节所示，从消融到完全预训练的扩展不仅仅是"即插即用"。它带来了意想不到的挑战，但我们成功地识别并解决了每个问题。

This section covers the essential monitoring setup and considerations for large-scale training runs. We'll address critical questions: when should you restart training after encountering problems? How do you handle issues that surface deep into a run? Which metrics truly matter?

本部分介绍大规模训练运行的基本监控设置和注意事项。我们将解决关键问题：遇到问题后应该什么时候重新开始培训？您如何处理在运行过程中出现的问题？哪些指标真正重要？

Should you maintain a fixed data mixture throughout training?

您是否应该在整个训练过程中保持固定的数据混合？

TRAINING MONITORING: BEYOND LOSS CURVES
训练监控：超越损失曲线

The reason we caught the tensor-parallelism bug was not the loss curve, which looked fine, but the fact that downstream evaluations were lagging behind expectations.

我们发现张量并行性错误的原因不是损失曲线，它看起来不错，而是下游评估落后于预期的事实。

Additionally, having evaluations from SmolLM2's intermediate checkpoints was critical: they gave us a sanity check that the 3B model wasn't on the right track early.

此外，从 SmolLM2 的中间检查点进行评估至关重要：他们给了我们一个健全性检查，表明 3B 模型没有在早期走上正确的轨道。

So if you're training large models, start running downstream evaluations early, and if you're comparing to an open-source model, ask whether the authors can provide intermediate checkpoints, those can be invaluable as reference points.

因此，如果您正在训练大型模型，请尽早开始运行下游评估，如果您正在与开源模型进行比较，请询问作者是否可以提供中间检查点，这些检查点可以作为参考点非常宝贵。

On the infrastructure side, the most important metric is throughput, measured in tokens per second. For SmolLM3, we expected stable throughput between 13,500–14,000 tokens/sec across the run, and any sustained deviation was a red flag.

在基础设施方面，最重要的指标是吞吐量，以每秒令牌数来衡量。对于 SmolLM3，我们预计整个运行过程中的稳定吞吐量在 13,500–14,000 个代币/秒之间，任何持续的偏差都是一个危险信号。

But throughput alone is not enough: you also need continuous hardware health monitoring to anticipate and detect hardware failures. Some of the key metrics we tracked included: GPU temperatures, memory usage and compute utilisation.

但仅靠吞吐量是不够的：您还需要持续的硬件运行状况监控来预测和检测硬件故障。我们跟踪的一些关键指标包括：GPU 温度、内存使用情况和计算利用率。

We log them into Grafana dashboards and set up real-time Slack alerts for hardware anomalies.

我们将他们登录到 Grafana 仪表板中，并针对硬件异常设置实时 Slack 警报。

FIX AND RESTART VS FIX ON THE FLY
修复和重新启动与动态修复

Given that we restarted our run after 1T tokens, an important question arises: do you always need to restart when something goes wrong? The answer depends on the severity and root cause of the issue.

鉴于我们在 1T 代币之后重新开始运行，一个重要的问题出现了：当出现问题时，您是否总是需要重新启动？答案取决于问题的严重性和根本原因。

In our case, the TP seeding bug meant we were starting on the wrong foot, half our weights weren't properly initialised.

在我们的例子中，TP 播种错误意味着我们开始时走错了路，我们一半的权重没有正确初始化。

The model was showing performance similar to SmolLM2 and plateauing at similar points, meaning we'd likely end up with a model that performed the same but cost almost twice as much to train. Restarting made sense.

该模型表现出与 SmolLM2 相似的性能，并在相似的点上趋于稳定，这意味着我们最终可能会得到一个性能相同但训练成本几乎是原来两倍的模型。重新启动是有道理的。

However, many issues can be course-corrected mid-run to avoid wasting compute.

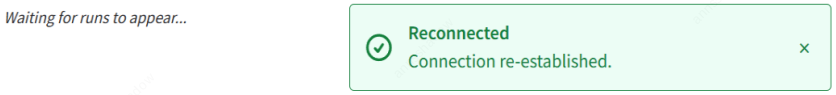
但是，许多问题可以在运行过程中进行纠正，以避免浪费计算。

The most common issue involves *loss spikes*, those sudden jumps in training loss that can either signal minor hiccups or divergence.

最常见的问题涉及损失峰值，训练损失的突然跳跃可能预示着小问题或分歧。

As Stas Bekman nicely puts it in the Machine Learning Engineering Open Book "Training loss plots are similar to heartbeat patterns—there's the good, the bad, and the you-should-worry ones."

正如 Stas Bekman 在《机器学习工程开放书》中很好地指出的那样，“训练损失图类似于心跳模式——有好的、坏的和你应该担心的。



Loss spikes fall into two categories:

损失峰值分为两类：

- Recoverable spikes: These can recover either fast (immediately after the spike) or slow (requiring several more training steps to return to the pre-spike trajectory). You can usually continue training through these.

  可恢复的峰值：这些峰值可以快速恢复（在峰值后立即恢复）或缓慢恢复（需要更多训练步骤才能恢复到峰值前的轨迹）。您通常可以通过这些继续训练。

  If recovery is very slow, you can try rewinding to a previous checkpoint to skip problematic batches.

  如果恢复速度非常慢，可以尝试倒回到以前的检查点以跳过有问题的批次。

- Non-recoverable spikes: The model either diverges or plateaus at worse performance than before the spike. These require more significant intervention than simply rewinding to a previous checkpoint.

  不可恢复的峰值：模型要么发散，要么停滞不前，性能比峰值之前更差。这些需要比简单地倒回到以前的检查点更重要的干预。

While we don't fully understand training instabilities, we know they become more frequent at scale. Common culprits, assuming a conservative architecture and optimizer, include:

虽然我们并不完全了解训练不稳定性，但我们知道它们在大规模上变得更加频繁。假设架构和优化器保守，常见的罪魁祸首包括：

- High learning rates: These cause instability early in training and can be fixed by reducing the learning rate.

  高学习率：这些会导致训练早期的不稳定，可以通过降低学习率来解决。

- Bad data: Usually the main cause of recoverable spikes, though recovery may be slow. This can happen deep into training when the model encounters low-quality data.

  数据错误：通常是可恢复峰值的主要原因，但恢复速度可能很慢。当模型遇到低质量数据时，这可能会在训练的深入处发生。

- Data-parameter state interactions: PaLM (Chowdhery et al., 2022) observed that spikes often result from specific combinations of data batches and model parameter states, rather than "bad data" alone. Training on the same problematic batches from a different checkpoint didn't reproduce the spikes.

  数据-参数状态交互：PaLM（Chowdhery 等人，2022 年）观察到，峰值通常是由数据批次和模型参数状态的特定组合引起的，而不是仅仅是“坏数据”。对来自不同检查点的相同有问题的批次进行训练没有重现峰值。

- Poor initialisation: Recent work by OLMo2 (OLMo et al., 2025) showed that switching from scaled initialisation to a simple normal distribution (mean=0,

std=0.02) improved stability.

初始化不良：OLMo2 最近的工作（OLMo 等人，2025 年）表明，从尺度初始化切换到简单正态分布（平均值=0，标准=0.02）提高了稳定性。

- Precision issues: While no one trains with FP16 anymore, BLOOM found it highly unstable compared to BF16.

  精度问题：虽然不再有人使用 FP16 进行训练，但 BLOOM 发现与 BF16 相比它非常不稳定。

Before spikes happen, build in stability:

在峰值发生之前，建立稳定性：

Small models with conservative learning rates and good data rarely spike, but larger models require proactive stability measures. As more teams have trained at scale, we've accumulated a toolkit of techniques that help prevent training instability:

具有保守学习率和良好数据的小型模型很少出现峰值，但较大的模型需要主动的稳定性措施。随着越来越多的团队进行大规模培训，我们积累了一个有助于防止培训不稳定的技术工具包：

Data filtering and shuffling: By this point in the blog, you've noticed how often we circle back to data. Making sure your data is clean and well-shuffled can prevent spikes.

数据过滤和洗牌：在博客的这一点上，您已经注意到我们经常返回数据。确保您的数据干净且洗牌良好可以防止峰值。

For instance, OLMo2 found that removing documents with repeated n-grams (32+ repetitions of 1-13 token spans) significantly reduced spike frequency.

例如，OLMo2 发现删除具有重复 n-gram 的文档（32+ 重复 1-13 个标记跨度）会显着降低尖峰频率。

Training modifications: Z-loss regularisation keeps output logits from growing too large without affecting performance. And excluding embeddings from weight decay also helps.

训练修改：Z 损失正则化可防止输出 logit 变得过大而不会影响性能。将嵌入排除在权重衰减之外也有帮助。

Architectural changes: QKNorm (normalising query and key projections before attention) has proven effective. OLMo2 and other teams found it helps with stability, and interestingly, Marin team found that it can even be applied mid-run to fix divergence issues.

架构变化：QKNorm（在注意力之前规范化查询和键预测）已被证明是有效的。OLMo2 和其他团队发现它有助于提高稳定性，有趣的是，Marin 团队发现它甚至可以在运行中应用来修复背离问题。

When spikes happen anyway - damage control:

无论如何，当尖峰发生时 - 损害控制：

Even with these precautions, spikes can still occur. Here are some options for fixing them:

即使采取了这些预防措施，峰值仍然可能发生。以下是修复它们的一些选项：

- **Skip problematic batches** : Rewind to before the spike and skip the problematic batches. This is the most common fix for spikes. The Falcon team (Almazrouei et al., 2023) skipped 1B tokens to resolve their spikes, while the PaLM team (Chowdhery et al., 2022) found that skipping 200-500 batches around the spike location prevented recurrence.

  跳过有问题的批次：倒回到峰值之前并跳过有问题的批次。这是最常见的峰值修复方法。Falcon 团队（Almazrouei 等人，2023 年）跳过 1B 代币来解决其峰值，而 PaLM 团队（Chowdhery 等人，2022 年）发现跳过峰值位置周围的 200-500 批次可以防止再次发生。

- **Tighten gradient clipping** : Reduce the gradient norm threshold temporarily

  收紧渐变剪切：暂时降低梯度范数阈值

- **Apply architectural fixes** such as QKnorm, as done in Marin.

  应用架构修复，例如 QKnorm，就像在 Marin 中所做的那样。

We've walked through the scaling challenges, from throughput drops to the TP bug, the monitoring practices to catch problems early, and strategies for preventing and fixing loss spikes.

我们已经介绍了扩展挑战，从吞吐量下降到 TP 错误、及早发现问题的监控实践，以及预防和

修复损失峰值的策略。

Let's finish this chapter by discussing how multi-stage training can enhance your model's final performance.

让我们通过讨论多阶段训练如何提高模型的最终性能来结束本章。

Mid-training 培训中期

Modern LLM pretraining typically involves multiple stages with different data mixtures, often followed by a final phase to extend context length. For example, Qwen3 (A. Yang, Li, et al., 2025) uses a three-stage approach: a general stage on 30T tokens at 4k context, a reasoning stage with 5T higher-quality tokens emphasising STEM and coding, and finally a long context stage on hundreds of billions of tokens at 32k context length.

现代 LLM 预训练通常涉及具有不同数据混合的多个阶段，通常随后是最后阶段以扩展上下文长度。例如，Qwen3（A. Yang， Li， et al.， 2025）使用了三阶段方法：在 4k 上下文下对 30T 代币进行一般阶段，以强调 STEM 和编码的 5T 更高质量代币进行推理阶段，最后是对 32k 上下文长度的数千亿个代币进行长上下文阶段。

SmolLM3 follows a similar philosophy, with planned interventions to introduce higher-quality datasets and extend context, alongside reactive adjustments based on performance monitoring.

SmolLM3 遵循类似的理念，计划进行干预以引入更高质量的数据集并扩展上下文，同时基于性能监控进行反应性调整。

As we explained in the data curation section, the data mixture doesn't have to stay fixed throughout training. Multi-stage training allows us to strategically shift dataset proportions as training progresses.

正如我们在数据管理部分中所解释的那样，数据混合不必在整个训练过程中保持固定。多阶段训练使我们能够随着训练的进行战略性地改变数据集比例。

Some interventions are planned from the start: for SmolLM3, we knew we'd introduce higher-quality FineMath4+ and Stack-Edu in Stage 2, then add curated Q&A and reasoning data during the final decay phase.

一些干预措施从一开始就计划好：对于 SmolLM3，我们知道我们将在第 2 阶段引入更高质量的 FineMath4+ 和 Stack-Edu，然后在最后的衰减阶段添加精选的问答和推理数据。

Other interventions are reactive, driven by monitoring performance during training. For example, in SmolLM2, when we found math and code performance lagging behind our targets, we curated entirely new datasets (FineMath and Stack-Edu) and introduced them mid-training.

其他干预措施是被动的，由训练期间的表现监测驱动。例如，在 SmolLM2 中，当我们发现数学和代码性能落后于我们的目标时，我们策划了全新的数据集（FineMath 和 Stack-Edu），并在训练过程中引入了它们。

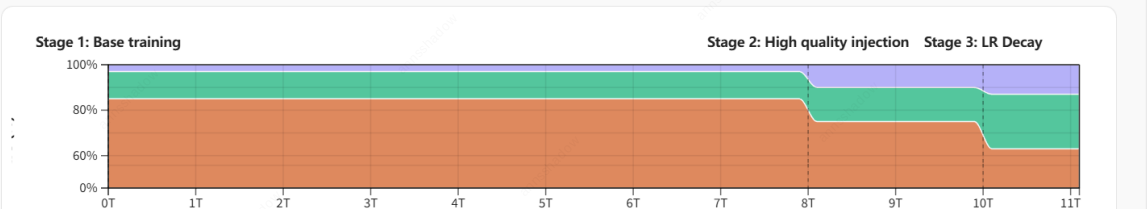
This flexibility—whether following a planned curriculum or adapting to emerging gaps—is what allows us to maximize the value of our compute budget.

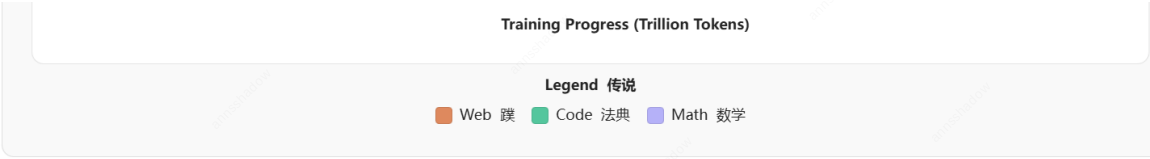
这种灵活性——无论是遵循计划的课程还是适应新出现的差距——使我们能够最大限度地发挥计算预算的价值。

STAGE 2 AND STAGE 3 MIXTURES
第 2 阶段和第 3 阶段混合物

The chart below show our 3 training stages and the progression of our web/code/math ratios during training. The SmolLM3 training configs for each stage are available here with exact data weights. For more details on the rationale and composition behind each stage, refer to the data curation section.

下图显示了我们的 3 个训练阶段以及训练期间我们的 web/code/math 比率的进展。此处提供了每个阶段的 SmolLM3 训练配置，其中包含精确的数据权重。有关每个阶段背后的基本原理和组成的更多详细信息，请参阅数据管理部分。





Stage 1: Base training (8T tokens, 4k context) The foundation stage uses our core pretraining mixture: web data (FineWeb-Edu, DCLM, FineWeb2, FineWeb2-HQ), code from The Stack v2 and StarCoder2, and math from FineMath3+ and InfiWebMath3+. All training happens at 4k context length.

第 1 阶段：基础训练（8T 令牌，4k 上下文）基础阶段使用我们的核心预训练组合：Web 数据（FineWeb-Edu、DCLM、FineWeb2、FineWeb2-HQ）、来自 The Stack v2 和 StarCoder2 的代码，以及来自 FineMath3+ 和 InfiWebMath3+ 的数学。所有训练都以 4k 上下文长度进行。

Stage 2: High-quality injection (2T tokens, 4k context) We introduce higher-quality filtered datasets: Stack-Edu for code, FineMath4+ and InfiWebMath4+ for math, and MegaMath for advanced mathematical reasoning (we add the Qwen Q&A data, synthetic rewrites, and text-code interleaved blocks).

第 2 阶段：高质量注入（2T 代币，4k 上下文）我们引入了更高质量的过滤数据集：用于代码的 Stack-Edu、用于数学的 FineMath4+ 和 InfiWebMath4+ 以及用于高级数学推理的 MegaMath（我们添加了 Qwen Q&A 数据、合成重写和文本代码交错块）。

Stage 3: LR decay with reasoning & Q&A data (1.1T tokens, 4k context) During the learning rate decay phase, we further upsample high-quality code and math datasets while introducing instruction and reasoning data like OpenMathReasoning, OpenCodeReasoning and OpenMathInstruct. The Q&A samples are simply concatenated and separated by new lines.

第 3 阶段：LR 衰减与推理和问答数据（1.1T token，4k 上下文）在学习率衰减阶段，我们进一步对高质量的代码和数学数据集进行上采样，同时引入 OpenMathReasoning、OpenCodeReasoning 和 OpenMathInstruct 等指令和推理数据。Q&A 样本只是简单地连接并由换行符分隔。

LONG CONTEXT EXTENSION: FROM 4K TO 128K TOKENS
长上下文扩展：从 4K 到 128K 代币

Context length determines how much text your model can process, it's crucial for tasks

长上下文扩展：从 4K 到 128K 代币

Context length determines how much text your model can process, it's crucial for tasks like analysing long documents, maintaining coherent multi-turn conversations, or processing entire codebases.

上下文长度决定了模型可以处理多少文本，它对于分析长文档、维护连贯的多轮对话或处理整个代码库等任务至关重要。

SmolLM3 started training at 4k tokens, but we needed to scale to 128k for real-world applications.

SmolLM3 开始以 4k 代币进行训练，但我们需要扩展到 128k 以用于实际应用程序。

Why extend context mid-training?

为什么要在训练中扩展上下文?

Training on long contexts from the start is computationally expensive since attention mechanisms scale quadratically with sequence length.

从一开始就在长上下文上进行训练在计算上是昂贵的，因为注意力机制随序列长度呈二次方缩放。

Moreover, research shows that extending context with a few dozen to a hundred billion tokens toward the end of training, or during continual pretraining, is enough to reach good long context performance(Gao et al., 2025).

此外，研究表明，在训练结束时或持续的预训练期间，用几十到一千亿个标记扩展上下文，足以达到良好的长上下文性能（Gao et al.， 2025）。

Sequential scaling: 4k→32k→64k

顺序缩放：4k→32k→64k

We didn't jump straight to 128k. Instead, we gradually extended context in stages, giving the model time to adapt at each length before pushing further.

我们没有直接跳到 128k。相反，我们逐步分阶段扩展上下文，让模型有时间适应每个长度，然后再进一步推进。

We ran two long context stages: first from 4k to 32k, then from 32k to 64k (the 128k capability comes from inference-time extrapolation, not training).

我们运行了两个长上下文阶段：首先从 4k 到 32k，然后从 32k 到 64k（128k 能力来自推理时间外推，而不是训练）。

We found that starting a fresh learning rate schedule for each stage over 50B tokens worked better than extending context during the last 100B tokens of the main decay phase.

我们发现，在主要衰减阶段的最后 100B 个标记中，为每个阶段启动一个新的学习率计划比扩展上下文效果更好。

At each stage, we ran ablations to find a good long context data mix and RoPE theta value, and evaluated on the Ruler benchmark.

在每个阶段，我们运行消融以找到良好的长上下文数据组合和 RoPE θ 值，并在 Ruler 基准上进行评估。

> 💡 **Long context evals on base model**
>
> **基本模型上的长上下文评估**
>
> During the long context ablations, we found HELMET benchmark to be very noisy on base models (the same training with different seeds gives variable results). Gao et al. recommend doing SFT on top to reduce variance on the benchmarks' tasks. Instead we opted for RULER, which we found to give more reliable signal at the base model

level.

在长上下文消融期间，我们发现 HELMET 基准在基础模型上非常嘈杂（使用不同种子进行相同的训练会给出不同的结果）。Gao 等人建议在顶部进行 SFT，以减少基准任务的差异。相反，我们选择了 RULER，我们发现它可以在基本模型级别提供更可靠的信号。

During this phase, it's common to upsample long context documents such as lengthy web pages and books to improve long context performance (Gao et al., 2025). We ran several ablations upsampling books, articles, and even synthetically generated documents for tasks like retrieval and fill-in-the-middle, following Qwen2.5-1M's approach (A. Yang, Yu, et al., 2025) with FineWeb-Edu and Python-Edu. Surprisingly, we didn't observe any improvement over just using the baseline mixture from Stage 3, which was already competitive with other state-of-the-art models like Llama 3.2 3B and Qwen2.5 3B on Ruler.

For more insights into long context extension, we recommend reading the paper How to Train Long-Context Language Models (Effectively)

有关长上下文扩展的更多见解，我们建议阅读论文 如何训练长上下文语言模型（有效）

在此阶段，通常对长上下文文档（例如冗长的网页和书籍）进行上采样，以提高长上下文性能（Gao 等人，2025 年）。我们按照 Qwen2.5-1M 的方法（A. Yang， Yu， et al.， 2025）使用 FineWeb-Edu 和 Python-Edu 对书籍、文章甚至合成生成的文档进行了多次消融，用于检索和中间填充等任务。令人惊讶的是，与仅使用第 3 阶段的基线混合物相比，我们没有观察到任何改进，该混合物已经与 Ruler 上的 Llama 3.2 3B 和 Qwen2.5 3B 等其他最先进的模型具有竞争力。

We hypothesise this is because the baseline mixture naturally includes long documents from web data and code (estimated at 10% of tokens), and that using NoPE helped.

我们假设这是因为基线组合自然包括来自 Web 数据和代码的长文档（估计占令牌的 10%），并且使用 NoPE 有所帮助。

RoPE ABF (RoPE with Adjusted Base Frequency): When going from 4k to 32k, we increased RoPE theta (base frequency) to 2M, and to go from 32k to 64k, we increased it to 5M.

We also experimented with sliding window attention (window sizes of 4k, 8k, and 16k) during the 4k→32k extension, but found it performed worse on RULER compared to full attention.

我们还在 4k→32k 扩展期间尝试了滑动窗口注意力（窗口大小为 4k、8k 和 16k），但发现与完全注意力相比，它在 RULER 上的表现更差。

RoPE ABF（调整基频的 RoPE）：当从 4k 到 32k 时，我们将 RoPE theta（基频）增加到 2M，为了从 32k 增加到 64k，我们将其增加到 5M。

We found that using larger values like 10M slightly improves RULER score but hurts some short context task like GSM8k, so we kept 5M which didn't impact short context.

我们发现，使用像 10M 这样的较大值会稍微提高 RULER 分数，但会损害一些短上下文任务，例如 GSM8k，因此我们保留了 5M，这不会影响短上下文。

During this context extension phase, we also used the opportunity to further upsampled math, code, and reasoning Q&A data, and we added few hundred thousand samples in ChatML format.

在这个上下文扩展阶段，我们还利用这个机会进一步上采样了数学、代码和推理 Q&A 数据，并添加了 ChatML 格式的数十万个样本。

YARN extrapolation: Reaching 128k Even after training on 64k contexts, we wanted SmolLM3 to handle 128k at inference. Rather than training on 128k sequences (prohibitively expensive), we used YARN (Yet Another RoPE extensioN method) (B. Peng et al., 2023), which allows the model to extrapolate beyond its training length. In theory, YARN allows a four-fold increase in sequence length.

YARN 外推：达到 128k 即使在 64k 上下文上训练后，我们也希望 SmolLM3 在推理时处理 128k。我们没有在 128k 序列上进行训练（成本高得令人望而却步），而是使用 YARN（Yet Another RoPE extensioN 方法）（B. Peng 等人，2023 年），它允许模型推断超出其训练长度。理论上，YARN 允许序列长度增加四倍。

We found that using the 64k checkpoint gave better performance at 128k than using the 32k checkpoint, confirming the benefit of training closer to the target inference length.

我们发现，使用 64k 检查点在 128k 时比使用 32k 检查点具有更好的性能，这证实了更接近目标推理长度的训练的好处。

However, pushing to 256k (four-fold from 64k) showed degraded Ruler performance, so we recommend using the model up to 128k.

但是，推到 256k（从 64k 增加四倍）显示 Ruler 性能下降，因此我们建议使用高达 128k 的模型。

And with that, we've walked through the full pretraining journey for SmolLM3, from planning and ablations to the final training run, with all the behind-the-scenes challenges along the way.

至此，我们已经完成了 SmolLM3 的完整预训练之旅，从规划和消融到最终的训练运行，以及在此过程中的所有幕后挑战。

Wrapping up pretraining 总结预训练

We've covered a lot of ground.

我们已经涵盖了很多领域。

From the training compass that helped us decide why and what to train, through strategic planning, systematic ablations that validated every architectural choice, to the actual training marathon where surprises emerged at scale (throughput mysteriously collapsing, dataloader bottlenecks, and a subtle tensor parallelism bug that forced a restart at 1T tokens).

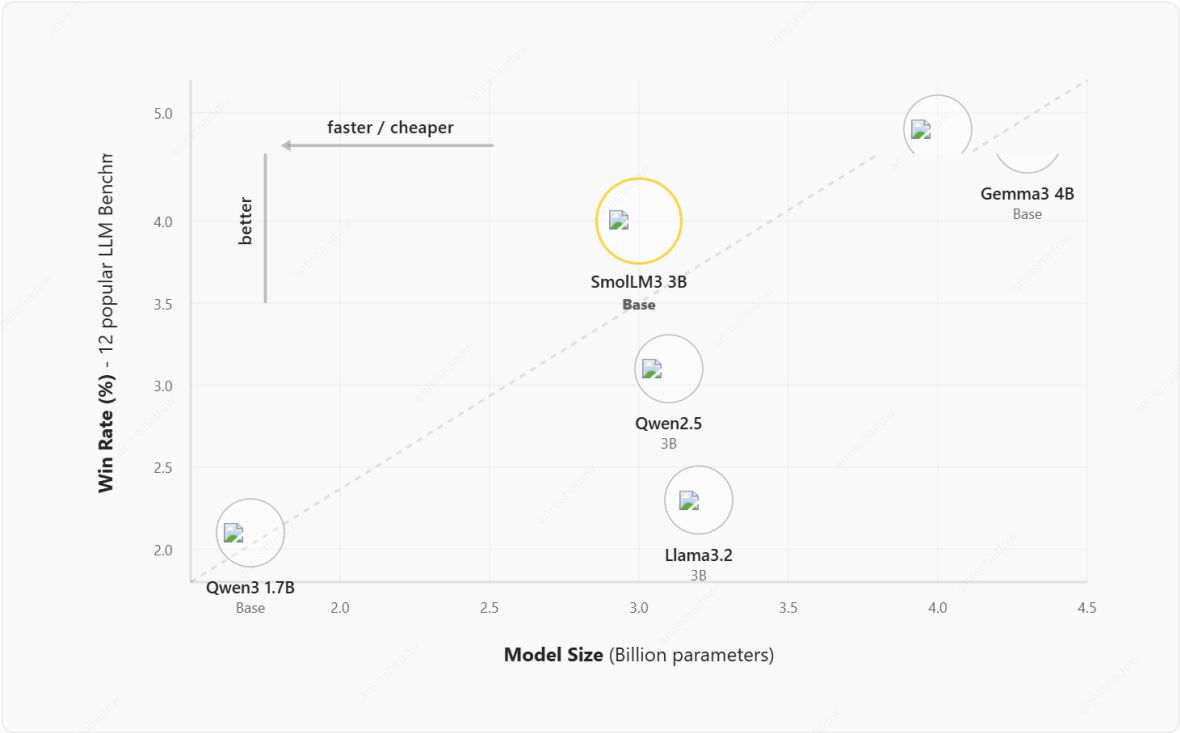
从帮助我们决定为什么训练和训练什么的训练指南针，到战略规划、验证每个架构选择的系统消融，再到大规模出现惊喜的实际训练马拉松（吞吐量神秘崩溃、数据加载器瓶颈以及迫使在 1T 令牌处重新启动的微妙张量并行性错误）。

The messy reality behind those polished technical reports is now visible: **training LLMs is as much about disciplined experimentation and rapid debugging as it is about architectural innovations and data curation.** Planning identifies what's worth testing. Ablations validate each decision. Monitoring catches problems early. And when things inevitably break, systematic derisking tells you exactly where to look.

这些精美的技术报告背后的混乱现实现在已经可见：培训法学硕士既关乎规范的实验和快速调试，也关乎架构创新和数据管理。规划确定值得测试的内容。消融验证每个决定。监控及早发现问题。当事情不可避免地出现故障时，系统性的去风险会准确地告诉你该去哪里寻找。

For SmolLM3 specifically, this process delivered what we set out to build: a 3B model trained on 11T tokens that's competitive on math, code, multilingual understanding, and long-context tasks, in the Pareto frontier of Qwen3 models.

特别是对于 SmolLM3，这个过程实现了我们着手构建的东西：一个在 11T 标记上训练的 3B 模型，在 Qwen3 模型的帕累托前沿中，在数学、代码、多语言理解和长上下文任务方面具有竞争力。



The win rate of of base models evaluation on: HellaSwag, ARC, Winogrande, CommonsenseQA, MMLU-CF, MMLU Pro CF, PIQA, OpenBookQA, GSM8K, MATH, HumanEval+, MBPP+

基础模型评估胜率：HellaSwag、ARC、Winogrande、CommonsenseQA、MMLU-CF、MMLU Pro CF、PIQA、OpenBookQA、GSM8K、MATH、HumanEval+、MBPP+

With our base model checkpoint saved, training complete, and GPUs finally cooling down, we might be tempted to call it done. After all, we have a model that predicts text well, achieves strong benchmark scores, and demonstrates the capabilities we targeted.

随着我们的基础模型检查点保存、训练完成以及 GPU 最终冷却，我们可能会想称其完成。毕竟，我们有一个模型可以很好地预测文本，获得强大的基准分数，并展示我们所瞄准的能力。

Not quite. Because what people want today are assistants and coding agents, not raw next-token predictors.

差一点。因为今天人们想要的是助手和编码代理，而不是原始的下一个令牌预测器。

This is where post-training comes in. And just like pretraining, the reality is messier than the papers suggest.

这就是培训后的用武之地。就像预训练一样，现实比论文所暗示的要混乱。

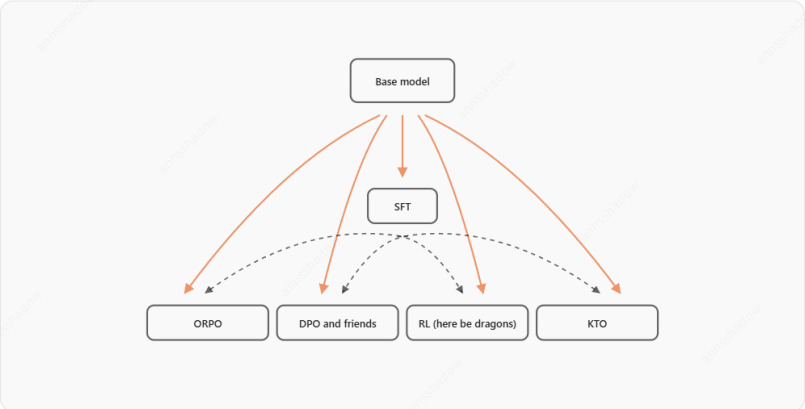
这就是培训后的用武之地。就像预训练一样，现实比论文所暗示的要混乱。

Beyond base models — post-training in 2025
超越基础模型 — 2025 年训练后

> Once the pre-training finishes we should have an SFT baseline within a day
>
> 预训练完成后，我们应该在一天内获得 SFT 基线

Lewis Tunstall, optimistic LLM expert.

Lewis Tunstall，乐观的法学硕士专家。



Choose your own (post-training) adventure.

选择您自己的（训练后）冒险。

Pre-training gave us SmolLM3's raw ability, but before the GPUs are cooled down we enter the next frontier of model capabilities: *post-training* . This includes supervised fine-tuning, reinforcement learning, model merging, and more — all designed to bridge the gap between "a model that predicts text" to "a model people can actually use".

If pre-training is about brute-forcing knowledge into weights, post-training is about sculpting that raw capability into something reliable and steerable.

如果说预训练是将知识蛮力转化为权重，那么后训练就是将原始能力塑造成可靠且可纵的东西。

And just like pre-training, the polished post-training papers don't capture the late-night surprises: GPU meltdowns, finicky data mixtures, or the way a seemingly minor chat template decision can ripple through downstream benchmarks.

就像预训练一样，精美的训练后论文并没有捕捉到深夜的惊喜：GPU 崩溃、挑剔的数据混合，或者看似微不足道的聊天模板决策可能会在下游基准测试中产生涟漪。

In this section, we'll show how we navigated the messy world of post-training to turn SmolLM3 from a strong base model into a state-of-the-art hybrid reasoner.

在本节中，我们将展示我们如何驾驭混乱的后训练世界，将 SmolLM3 从强大的基础模型转变为最先进的混合推理器。

> 📌 **What is a hybrid reasoning model?**
>
> **什么是混合推理模型？**
>
> A hybrid reasoning model operates in two distinct modes: one for concise, direct responses and another for extended step-by-step reasoning. Typically, the operating mode is set by the user in the system message.
>
> 混合推理模型以两种不同的模式运行：一种用于简洁、直接的响应，另一种用于扩展的分步推理。通常，作模式由用户在系统消息中设置。

> Following Qwen3, we make this explicit with lightweight commands: "/think" invokes extended reasoning, while "/no_think" enforces concise answers. This way, the user controls whether the model prioritises depth or speed.
>
> 在 Qwen3 之后，我们用轻量级命令明确了这一点："/think"调用扩展推理，而"/no_think"强制执行简洁的答案。这样，用户就可以控制模型是优先考虑深度还是速度。

Post-training compass: why → what → how
训练后指南针：为什么→什么→如何

Just like pre-training, post-training benefits from a clear compass to avoid wasted research and engineering cycles. Here's how to frame it:

就像预培训一样，培训后受益于清晰的指南针，以避免浪费研究和工程周期。以下是如何构建它：

就像预培训一样，培训后受益于清晰的指南针，以避免浪费研究和工程周期。以下是如何构建它：

1. **Why post-train?** The three motivations for training that we outlined in the pretraining compass—research, production, and strategic open-source—apply equally to post-training.

   为什么要进行培训后？我们在预训练指南针中概述的三种培训动机——研究、生产和战略开源——同样适用于培训后。

   For example, you might be exploring whether RL can unlock new reasoning capabilities in an existing model (research), or you might need to distill a large model into a smaller one for latency reasons (production), or you may have identified a gap where no strong open model exists for a specific use case (strategic open-source).

   例如，您可能正在探索 RL 是否可以在现有模型中解锁新的推理功能（研究），或者出于延迟原因（生产）可能需要将大型模型提炼成较小的模型，或者您可能已经确定了一个差距，即特定用例（战略开源）不存在强大的开放模型。

   The distinction is that post-training builds on existing capabilities rather than creating them from scratch.

   区别在于，后期培训建立在现有功能的基础上，而不是从头开始创建它们。

   However, before reaching for your GPUs, ask yourself:

   但是，在使用 GPU 之前，请问问自己：

   - Do you really need to post-train? Many open-weight models now rival proprietary ones on a wide range of tasks. Some can even be run locally with quantisation and modest compute. If you want a generalist assistant, an off-the-shelf model from the Hugging Face Hub may already meet your needs.

     你真的需要后期训练吗？现在，许多开放式重量模型在广泛的任务上可与专有模型相媲美。有些甚至可以通过量化和适度的计算在本地运行。如果您想要一个多面手助手，Hugging Face Hub 的现成型号可能已经满足您的需求。

   - Do you have access to high-quality, domain specific data? Post-training makes most sense when you are targeting a specific task or domain where a generalist model underperforms.

     您是否可以访问高质量的特定领域数据？当您针对通用模型表现不佳的特定任务或领域时，后训练最有意义。

     With the right data, you can tune the model to produce more accurate outputs for the applications you care most about.

     使用正确的数据，您可以调整模型，为您最关心的应用程序生成更准确的输出。

   - Can you measure success? If you don't have clear evaluation criteria, you'll never know if post-training actually helped.

     你能衡量成功吗？如果没有明确的评估标准，您将不知道培训后是否真的有帮助。

2. **What should post-training achieve?** This depends on your priorities:

   培训后应该达到什么目标？这取决于您的优先事项：

   - Do you want a crisp instruction-follower that rarely drifts off-topic?

     您想要一个很少偏离主题的清晰指令遵循者吗？

   - A versatile assistant that can switch tones and roles on demand?

     一个可以按需切换语气和角色的多功能助手？

   - A reasoning engine that can tackle math, code, or agentic problems?

     一个可以解决数学、代码或代理问题的推理引擎？

- A model that can converse in multiple languages?

  可以用多种语言交谈的模型？

3. **How will you get there?** That's where recipes matter. We'll cover:

   你将如何到达那里？这就是食谱很重要的地方。我们将介绍：

   - **Supervised fine-tuning (SFT)** to instil core capabilities.

     监督微调 （SFT） 以灌输核心功能。

   - **Preference optimisation (PO)** to directly learn from human or AI preferences.

     偏好优化 （PO） 可直接从人类或 AI 偏好中学习。

   - **Reinforcement learning (RL)** to refine reliability and reasoning beyond supervised data.

     强化学习 （RL） 用于完善监督数据之外的可靠性和推理。

   - **Data curation** to strike the right balance between diversity and quality.

     数据管理，在多样性和质量之间取得适当的平衡。

   - **Evaluation** to track progress and catch regressions early.

     评估以跟踪进度并及早发现回归。

This compass keeps the chaos of post-training grounded. The *why* gives direction, the *what* sets priorities, and the *how* turns ambitions into a practical training loop.

这个指南针让训练后的混乱保持脚踏实地。“为什么”给出了方向，“什么”设定了优先事项，“如何”将雄心壮志变成了一个实际的培训循环。

This compass keeps the chaos of post-training grounded. The *why* gives direction, the *what* sets priorities, and the *how* turns ambitions into a practical training loop.

这个指南针让训练后的混乱保持脚踏实地。“为什么”给出了方向，“什么”设定了优先事项，“如何”将雄心壮志变成了一个实际的培训循环。

Let's walk through how we answered these questions for SmolLM3:

让我们来看看我们是如何回答 SmolLM3 的这些问题的：

- **Why?** For us, the "why" was straightforward as we had a base model that needed post-training before release. At the same time, hybrid reasoning models like Qwen3 were becoming increasingly popular, yet open recipes showing how to train them were scarce.

  为什么？对我们来说，“为什么”很简单，因为我们有一个基础模型，在发布前需要进行后期训练。与此同时，像 Qwen3 这样的混合推理模型变得越来越流行，但展示如何训练它们的开放配方却很少。

  SmolLM3 gave us an opportunity to address both: prepare a model for real-world use and contribute a fully open recipe to sit on the Pareto front alongside Qwen3's 1.7B and 4B models.

  SmolLM3 为我们提供了解决这两个问题的机会：准备一个用于实际使用的模型，并贡献一个完全开放的配方，与 Qwen3 的 1.7B 和 4B 模型并列在帕累托前沿。

- **What?** We set out to train a hybrid reasoning model that was tailored to SmolLM3's strengths, chiefly that reasoning quality should hold up across languages other than English.

  什么？我们着手训练一个混合推理模型，该模型是根据 SmolLM3 的优势量身定制的，主要是推理质量应该在英语以外的语言中保持。

  And since real-world use increasingly involves tool calling and long-context workflows, those became core requirements in our post-training recipe.

  由于实际使用越来越多地涉及工具调用和长上下文工作流程，这些成为我们训练后配方的核心要求。

- **How?** That's the rest of this chapter 😀.

  如何？这就是本章😀的其余部分。

Just like with pre-training, we start with the fundamentals: evals and baselines, because every big model starts with a small ablation. But there's a key difference in how we ablate. In pre-training, "small" usually means smaller models and datasets.

就像预训练一样，我们从基础开始：评估和基线，因为每个大模型都从一个小的消融开始。但是，我们消融的方式有一个关键的区别。在预训练中，“小”通常意味着较小的模型和数据集。never use a different base model for ablations because behaviour is too model-dependent, and runs are short enough to iterate on the target model directly.

The main exception to this rule is when you're using an off-the-shelf base model from the Hugging Face Hub.

此规则的主要例外是当您使用 Hugging Face Hub 中的现成

In this case, ablating base models can make sense,

在后训练中，“小”意味着更小的数据集和更简单的算法。我们几乎从不使用不同的基础模型进行消融，因为行为过于依赖于模型，并且运行时间足够短，可以直接迭代目标模型。

Let's start with the topic many model trainers avoid until too late into a project: evals.

让我们从许多模型训练师在项目中为时已晚才回避的话题开始：评估。

since there's a world of difference between one model trained on 1T tokens and another on 10T, even if they have the same size.

在这种情况下，消融基础模型是有意义的，因为一个在 1T 代币上训练的模型和另一个在 10T 上训练的模型之间存在天壤之别，即使它们具有相同的大小。

First things first: evals before everything else
首先要做的事情：评估先于其他一切

The very first step in post-training — just like in pre-training — is to decide on the right set of evals. Since most LLMs today are used as assistants, we've found that aiming for a model that "works well" is a better goal than chasing abstract benchmarks of "intelligence" like ARC-AGI. So what does a good assistant need to do? At minimum, it should be able to:

培训后的第一步——就像在培训前一样——是决定正确的评估集。由于当今大多数 LLM 都被用作助手，我们发现，瞄准一个“运行良好”的模型比追逐 ARC-AGI 等“智能”的抽象基准是更好的目标。那么一个好的助手需要做什么呢？至少，它应该能够：

- Handle ambiguous instructions

  处理模棱两可的指令
- Plan step-by-step 逐步规划
- Write code 编写代码
- Call tools when appropriate

  适当时调用工具

These behaviours draw on a mix of reasoning, long-context handling, and skills with math, code, and tool use. Models as small as or even smaller than 3B parameters can work well as assistants, though performance usually falls off steeply below 1B.

这些行为结合了推理、长期上下文处理以及数学、代码和工具使用技能。小至 3B 参数甚至小于 3B 参数的模型可以很好地用作助手，尽管性能通常会急剧下降到 1B 以下。

It remains an interesting, yet open question whether tiny models can use tool calling to offset their limited capacity and thus act as viable assistants. See the models from LiquidAI for recent work in this direction.

微型模型是否可以使用工具调用来抵消其有限的能力，从而充当可行的助手，这仍然是一个有趣但悬而未决的问题。请参阅 LiquidAI 的模型，了解该方向的最新工作。

At Hugging Face, we use a layered eval suite, echoing the pre-training principles (monotonicity, low noise, above-random signal, ranking consistency) that we detailed in the ablations section for pretraining.

在 Hugging Face，我们使用分层评估套件，呼应我们在预训练的消融部分中详细介绍的预训练原则（单调性、低噪声、高于随机的信号、排名一致性）。

> 📍 **Keep your evals current**
>
> **保持您的评估最新**
>
> The list of evals to consider is continuously evolving as models improve and the ones discussed below reflect our focus in mid 2025. See the Evaluation Guidebook for a comprehensive overview of post-training evals.
>
> 随着模型的改进，要考虑的评估列表也在不断发展，下面讨论的评估反映了我们在 2025 年年中的重点。有关训练后评估的全面概述，请参阅评估指南。

Here are the many ways one can evaluate a post trained model:

以下是评估后训练模型的多种方法：

1. **Capability evals 能力评估**

This class of evals target fundamental skills, like reasoning and competitive math and coding.

此类评估针对基本技能，例如推理、竞争性数学和编码。

- **Knowledge.** We currently use GPQA Diamond (Rein et al., 2024) as the main eval for scientific knowledge. This benchmark consists of graduate-level, multiple-choice questions. For small models, it's far from saturated and gives better signal than MMLU and friends, while being much faster to run. Another good test of factuality is SimpleQA (Wei et al., 2024), although small models tend to struggle significantly on this benchmark due to their limited knowledge.

  知识。我们目前使用 GPQA Diamond（Rein 等人，2024 年）作为科学知识的主要评估。该基准由研究生水平的多项选择题组成。对于小型模型来说，它远未饱和，并且比 MMLU 和朋友提供更好的信号，同时运行速度要快得多。另一个很好的事实性测试是 SimpleQA

（Wei 等人，2024 年），尽管小型模型由于知识有限而往往在这个基准测试上表现不佳。

- **Math.** To measure mathematical ability, most models today are evaluated on the latest version of AIME (currently the 2025 version). MATH-500 (Lightman et al., 2023) remains a useful sanity test for small models, but is largely saturated by reasoning models. For a more comprehensive set of math evals, we recommend those from MathArena.

  数学。为了衡量数学能力，当今大多数模型都是在最新版本的 AIME（目前是 2025 版本）上进行评估的。MATH-500（Lightman 等人，2023 年）对于小型模型来说仍然是一个有

Although targeted towards competitive programming problems, we've found that improvements on LiveCodeBench do translate into better coding models, albeit limited to Python. SWE-bench Verified is a more sophisticated measure of coding skill, but tends to be too hard for small models and thus is not one we usually consider.

法典。我们使用最新版本的 LiveCodeBench 来跟踪编码能力。尽管针对的是竞争性编程问题，但我们发现 LiveCodeBench 的改进确实转化为更好的编码模型，尽管仅限于 Python。SWE-bench Verified 是衡量编码技能的更复杂的指标，但对于小型模型来说往往太难了，因此不是我们通常考虑的。

- **Multilinguality.** Unfortunately, there are not many options when it comes to testing the multilingual capabilities of models. We currently rely on Global MMLU (Singh et al., 2025) to target the main languages our models should perform well in, with MGSM (Shi et al., 2022) included as a test of multilingual mathematical ability.

  多语言。不幸的是，在测试模型的多语言能力时，没有太多选择。我们目前依靠 Global MMLU（Singh 等人，2025 年）来定位我们的模型应该表现良好的主要语言，其中 MGSM（Shi 等人，2022 年）作为多语言数学能力的测试。

1. **Integrated task evals 集成任务评估**

These evals test things that are close to what we'll ship: multi-turn reasoning, long-context use, and tool calls in semi-realistic settings.

这些评估测试了与我们将发布的内容相近的东西：多轮推理、长上下文使用以及半现实环境中的工具调用。

- **Long context.** The most commonly used test for long-context retrieval is the Needle in a Haystack (NIAH) (Kamradt, 2023), where a random fact ("needle") is placed in somewhere within a long document ("haystack") and the model has to retrieve it. However, this benchmark is too superficial to discriminate long-context understanding, so the community has developed more comprehensive evals like RULER (Hsieh et al., 2024) and HELMET (Yen et al., 2025). More recently, OpenAI have released the MRCR and GraphWalks benchmarks which extend the difficulty of long-context evals.

  See also this excellent blog post on the limitations of long-context evals and how to design realistic ones.

  另请参阅这篇关于长上下文评估的局限性以及如何设计现实评估的优秀博客文章。

  长上下文。长上下文检索最常用的测试是大海捞针 （NIAH）（Kamradt，2023），其中随机事实（"针"）被放置在长文档（"大海捞针"）中的某个地方，模型必须检索它。然而，这个基准过于肤浅，无法区分长期上下文理解，因此社区开发了更全面的评估，如 RULER （Hsieh et al.， 2024） 和 HELMET （ Yen et al.， 2025）。最近，OpenAI 发布了 MRCR 和 GraphWalks 基准测试，这扩展了长上下文评估的难度。

- **Instruction following.** IFEval (J. Zhou et al., 2023) is currently the most popular

MRCR 和 GraphWalks 基准测试，这扩展了长上下文评估的难度。

- **Instruction following.** IFEval (J. Zhou et al., 2023) is currently the most popular eval to measure instruction following, and uses automatic scoring against "verifiable instructions". IFBench (Pyatkin et al., 2025) is a new extension from Ai2 which includes a more diverse set of constraints than IFEval and mitigates some benchmaxxing that has occurred in recent model releases. For multi-turn instruction following, we recommend Multi-IF (He et al., 2024) or MultiChallenge (Sirdeshmukh et al., 2025).

  说明如下。IFEval（J. 周等人，2023 年）是目前最流行的衡量指令遵循的评估方法，它使用针对"可验证指令"的自动评分。IFBench （Pyatkin et al.， 2025） 是 Ai2 的一个新扩展，它包含比 IFEval 更多样化的约束集，并缓解了最近模型版本中发生的一些 benchmaxxing。对于多轮指令遵循，我们推荐 Multi-IF（He 等人，2024 年）或 MultiChallenge（Sirdeshmukh 等人，2025 年）。

- **Alignment.** Measuring how well models align to user intent is typically done through human annotators or by public leaderboards like LMArena. This is because qualities such as free-form generation, style, or overall helpfulness are difficult to measure quantitatively with automated metrics.

  对准。衡量模型与用户意图的一致性通常通过人工注释器或 LMArena 等公共排行榜来完成。这是因为自由格式生成、风格或整体有用性等品质很难用自动化指标进行定量衡量。

However, in all cases it is very expensive to run these evaluations which is why the community has resorted to using LLMs as a proxy for human preferences.

然而，在所有情况下，运行这些评估的成本都非常昂贵，这就是为什么社区求助于使用法学硕士作为人类偏好的代理。

The most popular benchmarks of this flavour include AlpacaEval(Dubois et al., 2025), ArenaHard (T. Li et al., 2024) and MixEval (Ni et al., 2024), with the latter having the strongest correlation with human Elo ratings on LMArena.

这种口味最受欢迎的基准包括 AlpacaEval（Dubois 等人，2025 年）、ArenaHard（T. Li 等人，2024 年）和 MixEval（Ni 等人，2024 年），后者与 LMArena 上的人类 Elo 评级相关性最强。

- **Tool calling.** BFCL provides a comprehensive test of tool calling, albeit one that is often saturated quite quickly. TAU-Bench (Barres et al., 2025) provides a test of a model's ability to use tools and resolve user problems in simulated customer service settings and has also become a popular benchmark to report on.

  工具调用。BFCL 提供了对工具调用的全面测试，尽管这种测试通常很快就会饱和。TAU-Bench（Barres 等人，2025 年）测试了模型在模拟客户服务设置中使用工具和解决用户问题的能力，也已成为流行的报告基准。

  题的能力，也已成为流行的报告基准。

1. **Overfitting-prevention evals**

   过度拟合预防评估

To test whether our models are overfitting to a specific skill, we include some robustness or adaptability evals in our set, like GSMPlus (Q. Li et al., 2024), which perturbs problems from GSM8k (Cobbe et al., 2021) to test whether models can still solve problems of similar difficulty.

为了测试我们的模型是否过度拟合特定技能，我们在集合中包含了一些鲁棒性或适应性评估，例如 GSMPlus（Q. Li 等人，2024 年），它扰动了 GSM8k（Cobbe 等人，2021 年）中的问题，以测试模型是否仍然可以解决类似难度的问题。

1. **Internal evals 内部评估**

Although public benchmarks can provide some useful signal during model development, they are no substitute for implementing your own internal evals to target specific capabilities, or asking internal experts to interact with your model.

尽管公共基准测试可以在模型开发过程中提供一些有用的信号，但它们不能替代实现您自己的内部评估以针对特定功能，或要求内部专家与您的模型交互。

This is especially true if you are building an AI product. See Hamel Husain's wonderful blog post for specific advice on this topic.

如果您正在构建人工智能产品，则尤其如此。有关此主题的具体建议，请参阅 Hamel Husain 的精彩博客文章。

For example, for SmolLM3 we needed a benchmark to evaluate whether the model was capable of *multi-turn reasoning* , so we implemented a variant of Multi-IF to measure this.

例如，对于 SmolLM3，我们需要一个基准来评估模型是否能够进行多轮推理，因此我们实现了 Multi-IF 的变体来测量这一点。

1. **Vibe evaluations and arenas**

   氛围评估和竞技场

Similarly, we have found that "vibe testing" intermediate checkpoints (aka interacting with your model) is essential for uncovering subtle quirks in model behaviour that are not captured by eval scores.

同样，我们发现"共振度测试"中间检查点（也就是与模型交互）对于发现模型行为中未被评估分数捕获的细微怪癖至关重要。

As we discuss later, vibe testing uncovered a bug in our data processing code where all system messages were deleted from the corpus!

正如我们稍后讨论的那样，vibe 测试发现了我们的数据处理代码中的一个错误，即所有系统消息都从语料库中删除了！

This is also something that can be done at scale to measure human preference, like on 的人类评估在很脆弱（倾向于阿谀奉承和幸福的答比而不是实际有用性），因此特类视为反馈信号反馈很重要。

> ☝ **Decontaminate your training data**
>
> **对训练数据进行净化**
>
> One risk with relying on public benchmarks is that the models can easily be overfit to them, especially when synthetic data is used to generate prompts and responses that are similar to the target benchmarks.
>
> 依赖公共基准的一个风险是模型很容易与它们过度拟合，尤其是当使用合成数据生成与目标基

准相似的提示和响应时。

For this reason, it is essential to decontaminate your training data against the evals you will use to guide model development.

因此，必须根据您将用于指导模型开发的评估对训练数据进行净化。

You can do this with N-gram matching using scripts like those inOpen-R1.

您可以使用 Open-R1 中的脚本通过 N-gram 匹配来做到这一点。

For SmolLM3 specifically, we wanted a hybrid reasoning model that could reliably follow instructions and reason well in popular domains like mathematics and code. We also wanted to ensure we preserved the base model's capabilities of multilinguality and long-context retrieval.

特别是对于 SmolLM3，我们想要一个混合推理模型，该模型能够可靠地遵循指令并在数学和代码等流行领域进行良好的推理。我们还希望确保保留基础模型的多语言和长上下文检索功能。

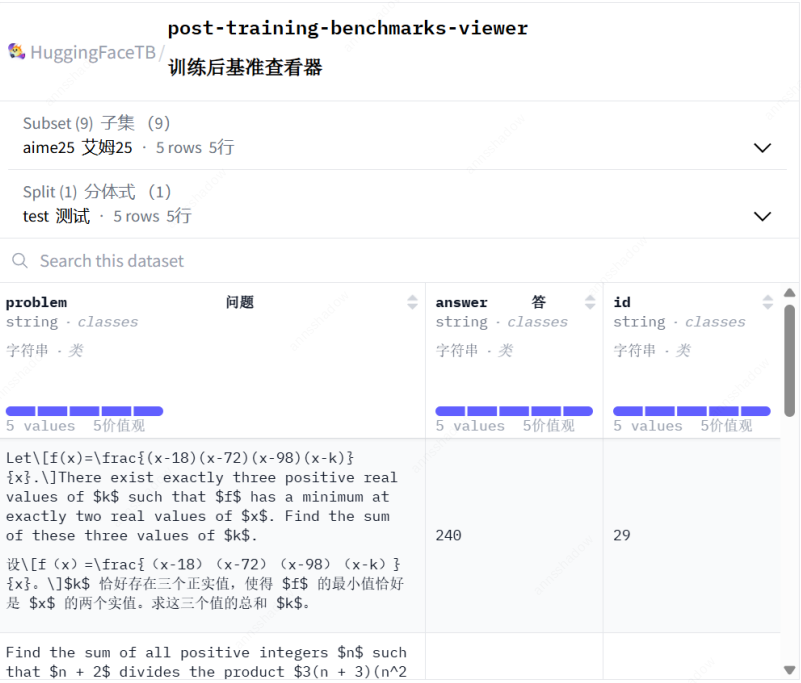
This led us to the following set of evals:

这导致我们得出了以下一组评估：

Benchmark 基准	Category 类别
AIME25	Competitive mathematics 竞争性数学
LiveCodeBench (v4 for validation, v5 for final release) LiveCodeBench（v4 用于验证，v5 用于最终版本）	Competitive programming 竞技节目
GPQA Diamond GPQA 钻石	Graduate-level reasoning 研究生水平
IFEval	Instruction following 说明遵循
MixEval Hard 混合评估硬	Alignment 对准
MixEval Hard 混合评估硬	Alignment 对准
BFCL v3	Tool use 工具使用
Global MMLU (lite for validation) 全局 MMLU（用于验证的精简版）	Multilingual Q&A 多语言问答
GSMPlus (mini for validation) GSMPlus（用于验证的迷你）	Robustness 鲁棒性
RULER 统治者	Long context 长上下文

Let's look at a few example questions from each to get a concrete sense of what these evaluations actually test:

让我们看看每个问题中的几个示例问题，以具体了解这些评估实际测试的内容：



Browse through the examples above to see the types of questions in each benchmark. Notice how the diversity of domains ensures we're testing different aspects of model capability throughout our ablations.

浏览上面的示例，查看每个基准中的问题类型，请注意，域的多样性如何确保我们在整个消融

For the 3B model scale we were working with, we felt these evals would give us actionable signal, run faster than training itself, and give us confidence that improvements are real and not just noise from sampling. We also tracked our pre-training evals (see the ablation section for a full list) to make sure we weren't regressing too much on the base model performance.

对于我们正在使用的 3B 模型规模，我们认为这些评估会给我们提供可作的信号，运行速度比训练本身更快，并让我们相信改进是真实的，而不仅仅是采样产生的噪音。我们还跟踪了我们的预训练评估（有关完整列表，请参阅消融部分），以确保我们不会在基础模型性能上倒退太多。

> ☝️ **Prioritise your evals 确定评估的优先级**
>
> The story above suggests that we got together as a team, converged on the set of evals, and had them ready to go before training any models.
>
> 上面的故事表明，我们作为一个团队聚集在一起，汇聚在评估集上，并在训练任何模型之前准备好它们。
>
> The reality was far messier: we had a tight deadline and rushed ahead with model training before many of the above evals were implemented (e.g. RULER was not available until a few days before the model release 🙈).
>
> 现实要混乱得多：我们的期限很紧，在上述许多评估实施之前就匆忙地进行了模型训练（例如，RULER 直到模型发布🙈前几天才可用）。
>
> In hindsight, this was a**mistake** and we should have discussed with the pre-training team which core evals should be preserved across post-training and also prioritised implementing them long before the base model was finished training. In other words, prioritise your evals before all else!
>
> 事后看来，这是一个错误，我们应该与预训练团队讨论哪些核心评估应该在训练后保留，并在基础模型完成训练之前很久就优先实施它们。换句话说，优先考虑您的评估！

RULES OF ENGAGEMENT 交战规则

Let's summarise this section with a few hard-earned lessons we've acquired from evaluating thousands of models:

让我们用我们从评估数千个模型中获得的一些来之不易的经验来总结本节：

- Use **small subsets to accelerate evals** during model development. For example, LiveCodeBench v4 is highly correlated with v5, but runs in half the time. Alternatively, use methods like those from tinyBenchmarks (Polo et al., 2024) which seek to find the smallest subset of prompts that reliably match the full evaluation.

  在模型开发过程中使用小子集来加速评估。例如，LiveCodeBench v4 与 v5 高度相关，但运行时间只有一半。或者，使用诸如 tinyBenchmarks （Polo et al.， 2024） 的方法，这些方法试图找到可靠匹配完整评估的最小提示子集。

- For **reasoning models**, strip the chain-of-thought from the **scored** output. This eliminates false positives and also directly impacts benchmarks like IFEval which penalise responses that violate constraints like "write a poem in under 50 words".

  对于推理模型，从评分输出中剥离思维链。这消除了误报，也直接影响了 IFEval 等基准测试，这些基准测试会惩罚违反"写一首诗少于 50 字"等约束的响应。

- If an eval uses **LLM judges, pin the judge and version** for apples-to-apples comparisons over time. Even better, use an open weight model so that the eval is reproducible even if a provider deprecates the judge model.

  如果评估使用 LLM 评判，请固定评审和版本，以便随着时间的推移进行同类比较。更好的是，使用开放权重模型，以便即使提供程序弃用 judge 模型，评估也是可重现的。

- Be wary of **contamination** in the base models. For example, most models released before AIME 2025 performed much worse than AIME 2024, suggesting some benchmaxxing was at play.

  警惕基本模型中的污染。例如，AIME 2025 之前发布的大多数车型的表现都比 AIME 2024 差得多，这表明存在一些 benchmaxxing 在起作用。

- If possible, treat anything used during ablations as **validation**, not **test.** This means keeping a set of held-out benchmarks for the final model reports, similar to the Tulu3 evaluation framework (Lambert et al., 2025).

  如果可能，将消融期间使用的任何东西视为验证，而不是测试。这意味着为最终模型报告保留一套保留的基准，类似于 Tulu3 评估框架（Lambert 等人，2025 年）。

- Always include a small set of **"vibe evals"** on your own data and tasks to catch

overfitting to public suites.

始终在您自己的数据和任务中包含一小组“氛围评估”，以发现过度适应公共套房。

- For evals with a small number of problems (typically less than ~2k), sample `k` times and report the `avg@k` accuracy. This is important to mitigate noise which can lead to incorrect decisions during development.

  对于问题数量较少（通常小于 ~2k）的评估，请采样 `k` 时间并报告准确性 `avg@k` 。这对于减轻噪音非常重要，噪音可能导致开发过程中做出错误的决策。

- When implementing a new eval, make sure you **can replicate the published result of a few models** (within some error). Failing to do this will waste a lot of time later on if you need to fix the implementation and re-evaluate many checkpoints.

  实现新的评估时，请确保可以复制几个模型的已发布结果（在某些错误中）。如果以后需要修复实现并重新评估许多检查点，如果不这样做，将浪费大量时间。

- When in doubt, always go back to the evaluation data, and notably inspect what you are prompting your models with

  如有疑问，请始终返回评估数据，并特别检查您提示模型的内容

With the evals at hand, it’s time to train some models! Before doing that, we first need to pick a post-training framework.

有了评估结果，是时候训练一些模型了！在此之前，我们首先需要选择一个训练后框架。

Tools of the trade

贸易工具

Behind every post-training recipe lies a toolbox of frameworks and libraries that enable large-scale experimentation. Each frameworks brings its own set of supported algorithms, fine-tuning methods, and scalability features.

每个训练后配方的背后都有一个框架和库工具箱，可以进行大规模实验。每个框架都有自己的一组受支持的算法、微调方法和可扩展性功能。

The table below summarises the main areas of support, from supervised fine-tuning (SFT) to preference optimisation (PO) and reinforcement learning (RL):

下表总结了从监督微调 （SFT） 到偏好优化 （PO） 和强化学习 （RL） 的主要支持领域：

Framework 框架	SFT SFT 的	PO 采购订单	RL	Multi-modal 多模态	FullFT 全
TRL 总链	✅	✅	✅	✅	✅
Axolotl 美西螈	✅	✅	✅	✅	✅
OpenInstruct 开放指示	✅	✅	✅	❌	✅
Unsloth 无树懒	✅	✅	✅	✅	✅
vERL vERL 的	✅	❌	✅	✅	✅
Prime RL 素数 RL	✅	❌	✅	❌	✅
PipelineRL 管道 RL	❌	❌	✅	❌	✅
ART 艺术	❌	❌	✅	❌	❌
TorchForge 火炬熔炉	✅	❌	✅	❌	✅
NemoRL 尼莫 RL	✅	✅	✅	❌	✅
OpenRLHF 开放 RLHF	✅	✅	✅	❌	✅

Here *FullFT* refers to **full fine-tuning**, where all model parameters are updated during training. *LoRA* stands for **Low-Rank Adaptation**, a parameter-efficient approach that updates only small low-rank matrices while keeping the base model frozen. *Multi-modal* refers to whether support for training on modalities beyond text (e.g. images) is supported and *Distributed* indicates whether training models on more than one GPU is possible.

这里的 FullFT 指的是完全微调，其中所有模型参数在训练过程中都会更新。LoRA 代表低秩适应，一种参数高效的方法，它只更新小型的低秩矩阵，同时保持基本模型冻结。多模态是指是否支持支持文本以外的模态（例如图像）的训练，分布式表示是否可以在多个 GPU 上训练模型。

At Hugging Face, we develop and maintain TRL, so it's our framework of choice and the one we used to post-train SmolLM3.

在 Hugging Face，我们开发和维护 TRL，因此它是我们选择的框架，也是我们用来后期训练 SmolLM3 的框架。

> 📍 **Fork your frameworks 分叉你的框架**
>
> Given the fast-moving pace of the field, we've found it quite effective to run our experiments on an internal fork of TRL. This allows us to add new features very quickly, which are later upstreamed to the main library.
>
> 鉴于该领域的快速发展，我们发现在 TRL 的内部分支上运行我们的实验非常有效。这使我们能够非常快速地添加新功能，这些功能稍后会上游到主库。
>
> If you're comfortable working with your framework's internals, adopting a similar workflow can be a powerful approach for rapid iteration.
>
> 如果您习惯于使用框架的内部结构，那么采用类似的工作流程可能是快速迭代的有效方法。

WHY BOTHER WITH FRAMEWORKS AT ALL?
为什么要为框架而烦恼呢?

There is a class of researchers that love to bemoan the use of training frameworks and instead argue that you should implement everything from scratch, all the time.

有一类研究人员喜欢哀叹训练框架的使用，而是认为你应该一直从头开始实现所有内容。

The implicit claim here is that "real" understanding only comes from re-implementing every RL algorithm, manually coding every distributed training primitive, or hacking together a one-off eval harness.

这里隐含的主张是，“真正的”理解只能来自重新实现每个 RL 算法、手动编码每个分布式训练

原语或破解一次性评估工具。

But this position ignores the reality of modern research and production. Take RL for example. Algorithms like PPO and GRPO are notoriously tricky to implement correctly (Huang et al., 2024), and tiny mistakes in normalisation or KL penalties can lead to days of wasted compute and effort.

但这种立场忽视了现代研究和生产的现实。以 RL 为例。众所周知，像 PPO 和 GRPO 这样的算法很难正确实现（Huang 等人，2024 年），归一化或 KL 惩罚中的微小错误可能会导致数天的计算和精力浪费。

Similarly, although it's tempting to write a single-file implementation of some algorithm, can that same script scale from 1B to 100B+ parameters?

同样，尽管编写某些算法的单文件实现很诱人，但相同的脚本可以从 1B 扩展到 100B+ 参数吗？

Frameworks exist precisely because the basics are already well-understood and endlessly reinventing them is a poor use of time. That's not to say there's no value in low-level tinkering. Implementing PPO from scratch once is an excellent learning exercise.

框架之所以存在，正是因为基础知识已经被充分理解，而无休止地重新发明它们是对时间的不良利用。这并不是说低级修补没有价值。从头开始实施一次 PPO 是一项极好的学习练习。

Writing a toy transformer without a framework teaches you how attention really works. But in most cases, just pick a framework you like and hack it for your purposes.

在没有框架的情况下编写玩具变形金刚可以教你注意力是如何真正发挥作用的。但在大多数情况下，只需选择一个您喜欢的框架并根据您的目的破解它即可。

With that rant out of the way, let's take a look at where we often start our training runs.

抛开咆哮，让我们来看看我们经常从哪里开始训练。

Why (almost) every post-training pipeline starts with SFT
为什么（几乎）每个训练后管道都从 SFT 开始

As we'll see later in this chapter, RL really does work, but comes with practical tradeoffs we discuss below.

正如我们将在本章后面看到的那样，RL 确实有效，但也带来了我们在下面讨论的实际权衡。

If you spend any time on X these days, you'd think reinforcement learning (RL) is the only game in town. Every day brings new acronyms, algorithmic tweaks, and heated debates (Chu et al., 2025; Yue et al., 2025) about whether RL can elicit new capabilities or not.

如果你现在花时间在 X 上，你会认为强化学习 （RL） 是城里唯一的游戏。每天都会带来新的首字母缩略词、算法调整和激烈的辩论（Chu 等人，2025 年;Yue et al.， 2025）关于 RL 是否可以引发新功能。

RL isn't new of course. OpenAI and other labs relied heavily on RL from human feedback (RLHF) (Lambert et al., 2022) to align their early models, but it wasn't until the release of DeepSeek-R1 (DeepSeek-AI, Guo, et al., 2025) that RL-based post-training really caught on in the open-source ecosystem.

当然，RL 并不新鲜。OpenAI 和其他实验室严重依赖来自人类反馈的 RL （RLHF）（Lambert 等人，2022 年）来调整他们的早期模型，但直到 DeepSeek-R1 的发布（DeepSeek-AI，Guo 等人，2025 年），基于 RL 的后训练才真正在开源生态系统中流行起来。

But one thing hasn't changed: almost every effective post-training pipeline still begins with supervised fine-tuning (SFT). The reasons are straightforward:

但有一件事没有改变：几乎每个有效的训练后管道仍然从监督微调 （SFT） 开始。原因很简单：

- **It's cheap:** SFT requires modest compute compared to RL. You can usually get meaningful gains without needing to burn a bonfire of silicon, and in fraction of the time required for RL.

  它很便宜：与 RL 相比，SFT 需要适度的计算。您通常无需燃烧硅篝火即可获得有意义的收益，而且所需时间仅为 RL 所需时间的一小部分。

- **It's stable:** unlike RL, which is notoriously sensitive to reward design and hyperparameters, SFT "just works."

  它是稳定的：与对奖励设计和超参数非常敏感的 RL 不同，SFT“只是工作”。

- **It's the right baseline:** a good SFT checkpoint usually gives most of the gains you're after, and it makes later methods like DPO or RLHF far more effective.

这是正确的基线：一个好的 SFT 检查点通常会提供您所追求的大部分收益，并且它使 DPO 或 RLHF 等后续方法更加有效。

In practice, this means that SFT isn't just the first step because it's easy; it's the step that consistently improves performance before anything more complex should be attempted. This is especially true when you are working with base models.

在实践中，这意味着 SFT 不仅仅是第一步，因为它很容易;在尝试任何更复杂的作之前，该步骤会持续提高性能。当您使用基础模型时尤其如此。

With a few exceptions, base models are too unrefined to benefit from advanced post-training methods.

除了少数例外，基础模型过于未完善，无法从高级训练后方法中受益。

> 📍 **What about DeepSeek R1-Zero?**
>
> **DeepSeek R1-Zero 怎么样?**
>
> At the frontier, the usual reasons for starting with SFT don't always apply. There's no stronger model to distill from and human annotations are too noisy for complex behaviors like long chain-of-thought. That's why DeepSeek skipped SFT and went straight to RL with R1-Zero; to *discover* reasoning behaviors that couldn't be taught with standard supervision.
>
> 在前沿，从 SFT 开始的通常原因并不总是适用。没有比这更强大的模型可以提炼出来，而且人类注释对于长思维链等复杂行为来说太嘈杂了。这就是为什么 DeepSeek 跳过了 SFT，直接转向了 R1-Zero 的 RL;发现标准监督无法教授的推理行为。
>
> If you're in that regime, starting with RL can make sense. But if you're operating there … you probably aren't reading this blog post anyway 😜.
>
> 如果您处于该制度中，那么从 RL 开始可能是有意义的。但如果你在那里运营……无论如何 😜，您可能没有阅读这篇博文。

So, if SFT is where most pipelines begin, the next question is: *what* should you fine-tune? That starts with choosing the right base model.

因此，如果 SFT 是大多数管道的起点，那么下一个问题是：您应该微调什么？首先要选择正确的基本型号。

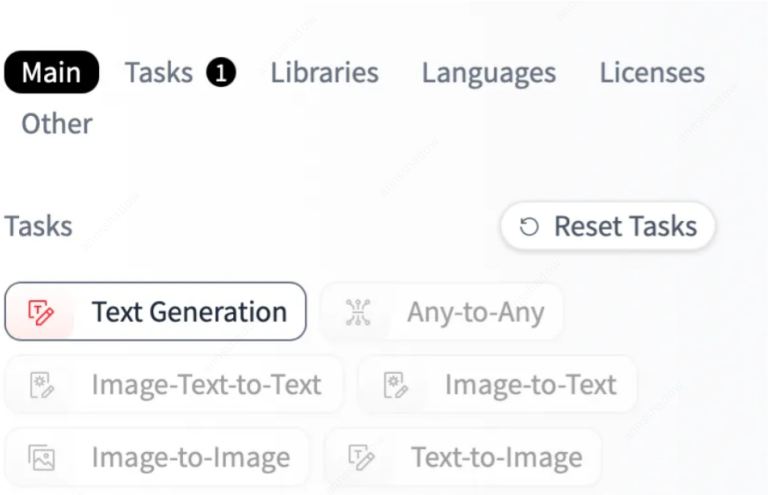
PICKING A BASE MODEL
选择基本模型

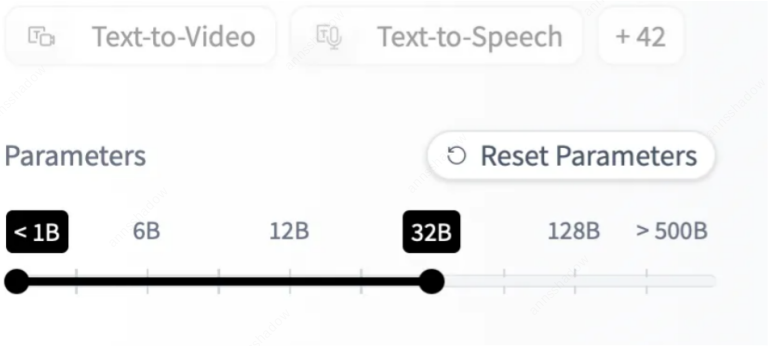
When choosing a base model for post-training, a few practical dimensions matter most:

在选择用于后训练的基础模型时，几个实用维度最为重要：

- **Model size:** although smol models have dramatically improved over time, it is still the case today that larger models generalise better, and often with fewer samples. Pick a model size that is representative of how you plan to use or deploy the model after training. On the Hugging Face Hub, you can filter models by modality and size to find suitable candidates:

  模型大小：尽管 SMOL 模型随着时间的推移而显着改进，但今天仍然如此，较大的模型可以更好地概括，而且通常样本更少。选择一个模型大小，以代表你计划在训练后如何使用或部署模型。在 Hugging Face Hub 上，您可以按模态和大小筛选模型以找到合适的候选模型：





- **Architecture (MoE vs dense):** MoE models activate a subset of parameters per token and offer higher capacity per unit of compute. They're great for large-scale serving, but trickier to fine-tune in our experience. By contrast, dense models are simpler to train and often outperform MoEs at smaller scales.

  架构（MoE 与密集）：MoE 模型激活每个令牌的参数子集，并提供更高的每单位计算容量。它们非常适合大规模服务，但根据我们的经验，微调起来比较棘手。相比之下，密集模型更易于训练，并且在较小规模下通常优于 MoE。

  型更易于训练，并且在较小规模下通常优于 MoE。

- **Post-training track record:** benchmarks are useful, but it's even better if the base model has already spawned a collection of strong post-trained models that resonate with the community. This provides a proxy for whether the model trains well.

  训练后记录：基准测试很有用，但如果基础模型已经催生了一系列与社区产生共鸣的强大后训练模型，那就更好了。这为模型是否训练良好提供了代理。

The LocalLLaMa subreddit is a great place to understand the broad vibes of new models. Artificial Analysis and LMArena also provide independent evaluation of new models, although these platforms are sometimes benchmaxxed by model providers.

LocalLLaMa Reddit 子版块是了解新模型广泛氛围的好地方。Artificial Analysis 和 LMArena 还提供对新模型的独立评估，尽管这些平台有时会被模型提供商 benchmaxxed。

In our experience, the base models from Qwen, Mistral, and DeepSeek are the most amenable to post-training, with Qwen being a clear favourite since each model series typically covers a large parameter range (e.g. Qwen3 models range in size from 0.6B to 235B!).

根据我们的经验，Qwen、Mistral 和 DeepSeek 的基础模型最适合后训练，其中 Qwen 显然是最受欢迎的，因为每个模型系列通常涵盖较大的参数范围（例如 Qwen3 模型的大小范围从 0.6B 到 235B！

This feature makes scaling far more straightforward.

此功能使扩展变得更加简单。

Once you've chosen a base model that matches your deployment needs, the next step is to establish a simple, fast SFT baseline to probe its core skills.

选择符合部署需求的基础模型后，下一步就是建立一个简单、快速的 SFT 基线来探测其核心技能。

TRAINING SIMPLE BASELINES
训练简单基线

For SFT, a good baseline should be fast to train, focused on the model's core skills, and simple to extend with more data when a particular capability isn't up to scratch.

对于 SFT，一个好的基线应该是快速训练的，专注于模型的核心技能，并且在特定功能不符合标准时易于扩展更多数据。

Choosing which datasets to use for an initial baseline involves some taste and familiarity with those that are likely to be of high quality.

选择用于初始基线的数据集需要一些品味和熟悉那些可能具有高质量的数据集。

In general, avoid over-indexing on public datasets which report high scores on academic benchmarks and instead focus on those which have been used to train great models likeOpenHermes. For example, in the development of SmolLM1, we initially ran SFT on WebInstruct, which is a great dataset on paper. However, during our vibe tests, we discovered it was too science focused because the model would respond with equations to simple greetings like "How are you?".

一般来说，避免对在学术基准上报告高分的公共数据集进行过度索引，而是专注于那些已用于训练 OpenHermes 等优秀模型的数据集。例如，在 SmolLM1 的开发中，我们最初在 WebInstruct 上运行了 SFT，这是一个很棒的纸面数据集。然而，在我们的氛围测试中，我们发现它过于注重科学，因为模型会用方程式来响应简单的问候语，例如"你好吗？

The use of vibe testing to uncover quirks in the training data is a recurrent theme in this chapter — don't underestimate the power of just chatting with your model!

使用共振度测试来发现训练数据中的怪癖是本章中反复出现的主题——不要低估与模型聊天的力量！

This led us to create the Everyday Conversations dataset which turned out to be crucial for instilling basic chat capabilities in small models.

这促使我们创建了日常对话数据集，事实证明，这对于在小型模型中灌输基本聊天功能至关重

要。

For SmolLM3, we set out to train a hybrid reasoning model and initially picked a small set of datasets to target reasoning, instruction following, and steerabilty. The table below shows the statistics of each dataset: [1]

对于 SmolLM3，我们着手训练一个混合推理模型，最初选择了一小组数据集来瞄准推理、指令遵循和可作性。下表显示了每个数据集的统计数据： [1]

Dataset 数据	Reasoning mode 推理模式	# examples # 示例	% of examples 示例百分比	# tokens (M)
Everyday Conversations 日常对话	/no_think	2,260	2.3	0.6
SystemChats 30k 系统聊天 30k	/no_think	33,997	35.2	21.5
Tulu 3 SFT Personas IF 图卢 3 SFT 角色 IF	/no_think	29,970	31.0	13.3
Everyday Conversations (Qwen3-32B) 日常对话 （Qwen3-32B）	/think /想	2,057	2.1	3.1
SystemChats 30k (Qwen3-32B) 系统聊天 30k （Qwen3-32B）	/think /想	27,436	28.4	29.4
s1k-1.1	/think /想	835	0.9	8.2
Total 总	-	96,555	100.0	76.1

Data mixture for hybrid reasoning baselines

混合推理基线的数据混合

As we learned throughout the development of SmolLM3, training hybrid reasoning models is trickier than standard SFT because you can't just mix datasets together; you need to *pair* data across modes. Each example has to clearly indicate whether the model should engage in extended reasoning or give a concise answer, and ideally you want parallel examples that teach it when to switch modes.

正如我们在整个 SmolLM3 开发过程中了解到的那样，训练混合推理模型比标准 SFT 更棘手，因为您不能只是将数据集混合在一起;您需要跨模式配对数据。每个示例都必须清楚地表明模型是否应该进行扩展推理或给出简洁的答案，理想情况下，您需要并行示例来教它何时切换模式。

Another thing to note from the above table is that you should balance your data mixture in terms of *tokens* not *examples*: for instance, the s1k-1.1 dataset is ~1% of the total examples but accounts for ~11% of the total tokens due to the long reasoning responses.

从上表中需要注意的另一件事是，您应该在标记而不是示例方面平衡数据混合：例如，s1k-1.1 数据集占总示例的 ~1%，但由于推理响应较长，占总标记的 ~11%。

This gave us basic coverage across the skills we cared about most, but also introduced a new challenge: each dataset had to be formatted differently, depending on whether it should enable extended thinking or not. To unify these formats, we needed a consistent chat template.

这为我们提供了我们最关心的技能的基本覆盖，但也带来了一个新的挑战：每个数据集的格式必须不同，具体取决于它是否应该支持扩展思维。为了统一这些格式，我们需要一个一致的聊天模板。

PICKING A GOOD CHAT TEMPLATE
选择一个好的聊天模板

When it comes to choosing or designing a chat template, there isn't a one-size-fits-all answer. In practice, we've found there are a few questions worth asking up front:

在选择或设计聊天模板时，没有一个放之四海而皆准的答案。在实践中，我们发现有几个问题值得预先问：

- **Can users customise the system role?** If users should be able to define their own system prompts (e.g. "act like a pirate"), the template needs to handle that cleanly.

  用户可以自定义系统角色吗？如果用户应该能够定义自己的系统提示（例如"像海盗一样行

事”），模板需要干净地处理它。

- **Does the model need tools?** If your model needs to call APIs, the template needs to accommodate structured outputs for tool calls and responses.

  模型需要工具吗？如果您的模型需要调用 API，则模板需要适应工具调用和响应的结构化输出。

- **Is it a reasoning model?** Reasoning models use templates like `<think> ... </think>` to separate the model's "thoughts" from its final answer. Some models discard the reasoning tokens across turns in a conversation, and the chat template needs to handle that logic.

  是推理模型吗？推理模型使用模板，将 `<think> ... </think>` 模型的“想法”与最终答案分开。一些模型会丢弃对话中跨回合的推理令牌，聊天模板需要处理该逻辑。

- **Will it work with inference engines?** Inference engines like vLLM and SGLang have dedicated parsers for reasoning and tools [2]. Compatibility with these parsers saves a lot of pain later, especially in complex agent benchmarks where consistent tool calls are essential. [3]

  它能与推理引擎一起使用吗？vLLM 和 SGLang 等推理引擎具有专用的推理解析器和工具 [2]。与这些解析器的兼容性可以节省很多以后的痛苦，尤其是在复杂的代理基准测试中，一致的工具调用至关重要。 [3]

The table below shows a few popular chat templates and how they compare across the key considerations:

下表显示了一些流行的聊天模板，以及它们在关键考虑因素中的比较：

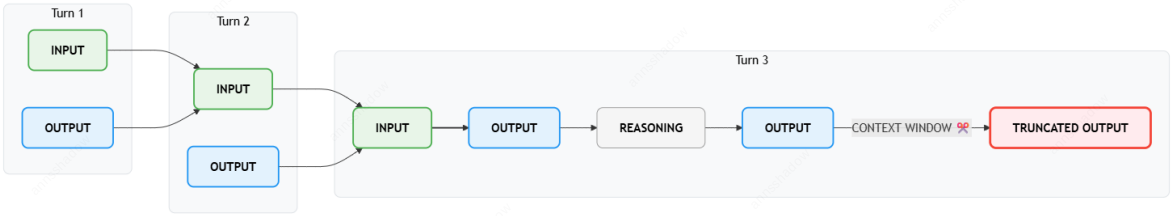
Chat template 聊天模板	System role customisation 系统角色定制	Tools 工具	Reasoning 推理	Inference compatibility 推理兼容
ChatML 聊天机器学习	✅	✅	❌	✅
Qwen3	✅	✅	✅	✅
DeepSeek-R1 深搜索-R1	❌	❌	✅	✅
Llama 3 骆驼 3	✅	✅	❌	✅
Llama 3 骆驼 3	✅	✅	❌	✅
Gemma 3 杰玛 3	✅	❌	❌	❌
Command A Reasoning 命令 A 推理	✅	✅	✅	❌
GPT-OSS	✅	✅	✅	✅

In most cases, we've found that ChatML or Qwen's chat templates are an excellent place to start. For SmolLM3, we needed a template for hybrid reasoning and found that Qwen3 was one of the few templates that struck a good balance across the dimensions we cared about.

在大多数情况下，我们发现 ChatML 或 Qwen 的聊天模板是一个很好的起点。对于 SmolLM3，我们需要一个用于混合推理的模板，发现 Qwen3 是少数几个在我们关心的维度上取得良好平衡的模板之一。

However, it had one quirk that we weren't entirely happy with: the reasoning content is*discarded* for all but the final turn in a conversation. As shown in the figure below, this is similar to how OpenAI's reasoning models work:

然而，它有一个我们并不完全满意的怪癖：推理内容被丢弃除了对话的最后一个回合。如下图所示，这类似于 OpenAI 的推理模型的工作原理：



Although this makes sense for *inference* (to avoid blowing up the context), we concluded that for *training* it is important to *retain the reasoning tokens across all turns* in order to condition the model appropriately.

尽管这对于推理是有意义的（以避免破坏上下文），但我们得出的结论是，对于训练，重要的是要在所有回合中保留推理标记，以便适当地调节模型。

Instead, we decided to craft our own chat template with the following features:

相反，我们决定制作自己的聊天模板，具有以下功能：

- A structured system prompt, like Llama 3's and those jailbroken from proprietary models. We also wanted to offer the flexibility to override the system prompt entirely.

  结构化系统提示，如 Llama 3 和从专有模型越狱的提示。我们还希望提供完全覆盖系统提示的灵活性。

- Support for code agents, which execute arbitrary Python code instead of making JSON tool calls.

  支持代码代理，它执行任意 Python 代码，而不是进行 JSON 工具调用。

- Explicit control of the reasoning mode via the system message.

  通过系统消息显式控制推理模式。

To iterate on the design of the chat template, we used the Chat Template Playground. This handy application was developed by our colleagues at Hugging Face and makes it easy to preview how messages are rendered and debug formatting issues. Here's an embedded version of the playground so you can try it out directly:

为了迭代聊天模板的设计，我们使用了聊天模板游乐场。这个方便的应用程序是由我们在 Hugging Face 的同事开发的，可以轻松预览消息的呈现方式和调试格式问题。这是 Playground 的嵌入式版本，因此您可以直接试用：

Chat template for HuggingFaceTB/SmolLM3-3B change model

Copy Formatted

```
{#  ——— defaults ———  #}
{%- if enable_thinking is not defined -%}
```

Copy Formatted

```
{#  ——— defaults ———  #}
{%- if enable_thinking is not defined -%}
    {%- set enable_thinking = true -%}
{%- endif -%}
{#  ——— reasoning mode ———  #}
{%- if enable_thinking -%}
    {%- set reasoning_mode = "/think" -%}
{%- else -%}
    {%- set reasoning_mode = "/no_think" -%}
{%- endif -%}
{#  ——— header (system message) ———  #}
{{- "<|im_start|>system\n" -}}
{%- if messages[0].role == "system" -%}
    {%- set system_message = messages[0].content -%}
    {%- if "/no_think" in system_message -%}
        {%- set reasoning_mode = "/no_think" -%}
    {%- elif "/think" in system_message -%}
        {%- set reasoning_mode = "/think" -%}
    {%- endif -%}
    {%- set custom_instructions =
system_message.replace("/no_think",
"").replace("/think", "").rstrip() -%}
```

JSON Input hello world example

Copy

```
{
  messages: [
    {
      role: 'system',
      content: 'You are a helpful assistant.',
```

```
  messages: [
    {
      role: 'system',
      content: 'You are a helpful assistant.',
    },
    {
      role: 'user',
      content: 'Hello, how are you?',
    },
    {
```

Rendered Output

Copy

```
<|im_start|>system
## Metadata

Knowledge Cutoff Date: June 2025
Today Date: 13 十一月 2025
Reasoning Mode: /think

## Custom Instructions
```

```
{%- endif -%}
{%- if "/system_override" in system_message -%}
    {{- custom_instructions.replace("/system_override",
```

```
You are a helpful assistant.
```

Select different examples from the drop-down to see how the chat template works for multi-turn dialogues, reasoning, or tool-use. You can even change the JSON input manually to enable different behaviour. For example, see what happens if you provide `enable_thinking: false` or append `/no_think` to the system message.

从下拉列表中选择不同的示例，查看聊天模板如何用于多回合对话、推理或工具使用。您甚至可以手动更改 JSON 输入以启用不同的行为。例如，查看提供 `enable_thinking: false` 或附加 `/no_think` 到系统消息时会发生什么情况。

Once you've settled on some initial datasets and a chat template, it's time to train some baselines!

确定了一些初始数据集和聊天模板后，就该训练一些基线了！

BABY BASELINES 婴儿基线

Before we dive into optimisation and squeezing every point of performance, we need to establish some "baby baselines".

在我们深入研究优化和压缩性能的每一点之前，我们需要建立一些"婴儿基线"。

These baselines aren't about reaching state-of-the-art (yet), but aim to validate that the chat template does what you want and that the initial set of hyperparameters produce stable training.

这些基线并不是要达到最先进的水平，而是旨在验证聊天模板是否执行所需的作，以及初始超参数集是否产生稳定的训练。

Only after we have this foundation do we start heavily tuning hyperparameters and training mixtures.

只有在我们有了这个基础之后，我们才会开始大量调整超参数和训练混合物。

When it comes to training SFT baselines, here's the main things to consider:

在训练 SFT 基线时，需要考虑以下主要事项：

- Will you use full fine-tuning (FullFT) or parameter efficient methods like LoRA or QLoRA? As described in the wonderful blog post by Thinking Machines, LoRA can match FullFT under certain conditions (usually determined by the size of the dataset).

  您会使用完全微调 （FullFT） 还是 LoRA 或 QLoRA 等参数高效的方法？正如 Thinking Machines 的精彩博客文章中所描述的，LoRA 可以在某些条件下（通常由数据集的大小决定）与 FullFT 匹配。

- What type of parallelism do you need? For small models or those trained with LoRA, you can usually get by with data parallel. For larger models you will need FSDP2 or DeepSpeed ZeRO-3 to shared the model weights and optimiser states.

  您需要哪种类型的并行度？对于小型模型或使用 LoRA 训练的模型，您通常可以使用并行数据。对于较大的模型，您将需要 FSDP2 或 DeepSpeed ZeRO-3 来共享模型权重和优化器状态。

  For models trained with long context, use methods likecontext parallelism.

  对于使用长上下文训练的模型，请使用上下文并行性等方法。

- Use kernels like FlashAttention and Liger if your hardware supports them. Many of these kernels are hosted on the Hugging Face Hub and can be set via a simple argument in TRL to dramatically lower the VRAM usage.

  如果您的硬件支持，请使用 FlashAttention 和 Liger 等内核。其中许多内核托管在 Hugging Face Hub 上，可以通过 TRL 中的简单参数进行设置，以显着降低 VRAM 使用率。

- Mask the loss to train only on assistant tokens. As we discuss below, this can be achieved by wrapping the assistant turns of your chat template with a special `{% generation %}` keyword.

  掩盖损失以仅在助手令牌上进行训练。正如我们在下面讨论的，这可以通过用特殊 `{% generation %}` 关键字包装聊天模板的助手轮次来实现。

- Tune the learning rate; aside from the data, this is the most important factor that determines whether your model is "meh" vs "great".

  调整学习率;除了数据之外，这是决定您的模型是"一般"还是"优秀"的最重要因素。

- Pack the training samples and tune the sequence length to match the distribution of your data. This will dramatically speed up training. TRL has a handy application to do this for you.

  打包训练样本并调整序列长度以匹配数据的分布。这将大大加快训练速度。TRL 有一个方便的应用程序可以为您完成此作。

Let's look at how some of these choices panned out for SmolLM3. For our first baseline experiments, we wanted a simple sanity check: does the chat template actually elicit hybrid reasoning? To test this, we compared three data mixtures from our table:

让我们看看其中一些选择是如何为 SmolLM3 实现的。对于我们的第一个基线实验，我们想要一个简单的健全性检查：聊天模板是否真的引发了混合推理？为了测试这一点，我们比较了表中的三种数据混合：

- **Instruct:** train on the non-reasoning examples.

  指导：训练非推理示例。

- **Thinking:** train on the reasoning examples.

  思考：根据推理示例进行训练。

- **Hybrid:** train on all examples.

  混合：对所有示例进行训练。

For each mixture, we ran SFT on SmolLM3-3B-Base using FullFT with a learning rate of 1e-5, an effective batch size of 128, and trained for 1 epoch.

We have found that for most models and datasets, this choice of hyperparameters works well as a baseline.

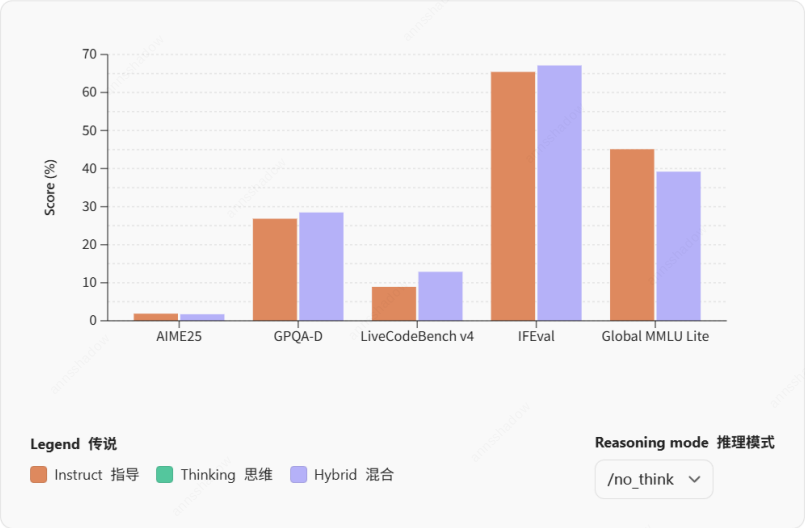
我们发现，对于大多数模型和数据集，这种超参数的选择可以很好地作为基线。

对于每种混合物，我们使用 FullFT 在 SmolLM3-3B-Base 上运行 SFT，学习率为 1e-5，有效批量为 128，并训练了 1 个 epoch。

Since these are small datasets, we did not use packing, capping sequences at 8,192 tokens for the Instruct subset and 32,768 tokens for the rest. On one node of 8 x H100s, these experiments were quick to run, taking between 30-90 minutes depending on the subset.

[illegible] reasoning mode:

下图比较了每个子集在相应推理模式下的性能：



These results quickly showed us that hybrid models exhibit a type of "split brain", where the data mixture for one reasoning mode has little effect on the other.

这些结果很快向我们表明，混合模型表现出一种"大脑分裂"，其中一种推理模式的数据混合对另一种推理模式影响不大。

This is evident by most evals having similar scores between the Instruct, Thinking and Hybrid subsets, with LiveCodeBench v4 and IFEval being the exception where hybrid data boosts the overall performance.

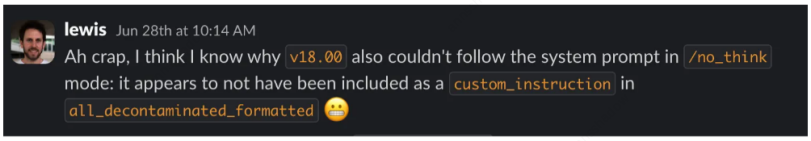
大多数评估在 Instruct、Thinking 和 Hybrid 子集之间具有相似的分数就可以看出这一点，LiveCodeBench v4 和 IFEval 是混合数据提高整体性能的例外。

VIBE-TEST YOUR BASELINES VIBE-测试您的基线

Although the evals looked OK, when we tried getting the hybrid model to act in different personas (e.g. like a pirate), it consistently ignored anything we placed in the system message. After a bit of digging, we found the reason was due to the way we

had formatted the data:

尽管评估看起来不错，但当我们尝试让混合模型以不同的角色（例如海盗一样）行事时，它始终忽略我们在系统消息中放置的任何内容。经过一番挖掘，我们发现原因是由于我们格式化数据的方式：



What happened is that in the design of our chat template, we exposed a `custom_instructions` argument to store the system prompts. For example, here's how we set a persona in a dialogue:

发生的事情是，在聊天模板的设计中，我们公开了一个 `custom_instructions` 参数来存储系统提示。例如，以下是我们在对话中设置角色的方法：

```python
from transformers import AutoTokenizer
Â
tok = AutoTokenizer.from_pretrained("HuggingFaceTB/SmolLM3-3B")
Â
messages = [
    {
        "content": "I'm trying to set up my iPhone, can you help?",
        "role": "user",
    },
    {
        "content": "Of course, even as a vampire, technology can be a bit of a ch
        "role": "assistant",
    },
]
chat_template_kwargs = {
    "custom_instructions": "You are a vampire technologist",
    "enable_thinking": False,
}
rendered_input = tok.apply_chat_template(
    messages, tokenize=False, **chat_template_kwargs
)
print(rendered_input)
## <|im_start|>system
### Metadata
Â
## Knowledge Cutoff Date: June 2025
## Today Date: 28 October 2025
## Reasoning Mode: /no_think
Â
### Custom Instructions
Â
## You are a vampire technologist
Â
## <|im_start|>user
## I'm trying to set up my iPhone, can you help?<|im_end|>
## <|im_start|>assistant
## <think>
Â
## </think>
## Of course, even as a vampire, technology can be a bit of a challenge sometimes
```

The issue was that our data samples looked like this:

问题是我们的数据样本如下所示：

```python
{
    "messages": [
        {
            "content": "I'm trying to set up my iPhone, can you help?",
            "role": "user",
        },
        {
            "content": "Of course, even as a vampire, technology can be a bit of
            "role": "assistant",
        },
    ],
    "chat_template_kwargs": {
        "custom_instructions": None,
        "enable_thinking": False,
        "python_tools": None,
        "xml_tools": None,
    },
}
```

A bug in our processing code had set `custom_instructions` to `None`, which effectively removed the system message from *every single training sample* 🙈! So instead of getting a nice persona for these training samples, we ended up with the SmolLM3 default system prompt:

我们的处理代码中的一个错误设置为 `custom_instructions` `None`，这有效地从每个训练样本🙈中删除了系统消息！因此，我们没有为这些训练样本获得一个漂亮的角色，而是最终得到了 SmolLM3 默认系统提示：

```
chat_template_kwargs = {"custom_instructions": None, "enable_thinking": False}
rendered_input = tok.apply_chat_template(messages, tokenize=False, **chat_templat
print(rendered_input)
## <|im_start|>system
#### Metadata
Â
## Knowledge Cutoff Date: June 2025
## Today Date: 28 October 2025
## Knowledge Cutoff Date: June 2025
## Today Date: 28 October 2025
## Reasoning Mode: /no_think
Â
#### Custom Instructions
Â
## You are a helpful AI assistant named SmolLM, trained by Hugging Face.
Â
## <|im_start|>user
## I'm trying to set up my iPhone, can you help?<|im_end|>
## <|im_start|>assistant
## <think>
Â
## </think>
## Of course, even as a vampire, technology can be a bit of a challenge sometimes
```

This was especially problematic for the SystemChats subset, where all the personas are defined via `custom_instructions` and thus the model had a tendency to randomly switch character mid-conversation. This brings us to the following rule:

这对于 SystemChats 子集来说尤其成问题，其中所有角色都是通过定义的 `custom_instructions`，因此模型倾向于在对话中随机切换角色。这给我们带来了以下规则：

> ☝ **Rule 统治**
>
> Always vibe-test your models, even if the evals look fine. More often than not, you will uncover subtle bugs in your training data.
>
> 始终对模型进行振动测试，即使评估结果看起来不错。通常情况下，您会发现训练数据中的细微错误。

Fixing this bug had no impact on the evals, but finally we were confident the chat template and dataset formatting was working. Once your setup is stable and your data pipeline checks out, the next step is to focus on developing specific capabilities.

修复此错误对评估没有影响，但最终我们确信聊天模板和数据集格式是有效的。一旦您的设置稳定并且您的数据管道签出，下一步就是专注于开发特定功能。

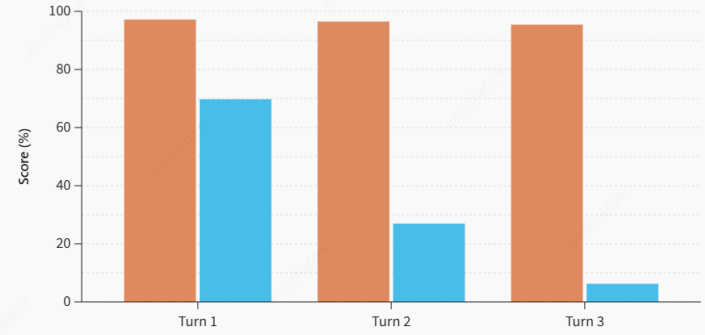
TARGETING SPECIFIC CAPABILITIES
针对特定功能

During the development of Open-R1, we noticed that training a base model entirely on single-turn reasoning data would fail to generalise to multi-turn. This is not a surprise; absent such examples, the model is being tested outside its training distribution.

在开发 Open-R1 的过程中，我们注意到完全基于单轮推理数据训练基础模型将无法推广到多轮。这并不奇怪;如果没有这样的例子，该模型正在其训练分布之外进行测试。

To measure this quantitatively for SmolLM3, we took inspiration from Qwen3, who developed an internal eval called *ThinkFollow*, which randomly inserts `/think` or `/no_think` tags to test whether the model can consistently switch reasoning modes. In our implementation, we took the prompts from Multi-IF and then checked if the model generated empty or non-empty think blocks enclosed in the `<think>` and `</think>` tags. As expected, the results from our hybrid baseline showed the model failing abysmally to enable the reasoning mode beyond the first turn:

为了对 SmolLM3 进行定量测量，我们从 Qwen3 中汲取了灵感，Qwen3 开发了一个名为 ThinkFollow 的内部评估，它随机插入 `/think` 或 `/no_think` 标记以测试模型是否能够持续切换推理模式。在我们的实现中，我们从 Multi-IF 中获取提示，然后检查模型是否生成了包含在 `<think>` 和 `</think>` 标签中的空或非空思考块。正如预期的那样，我们的混合基线的结果显示，该模型在第一轮之后启用推理模式方面非常失败：

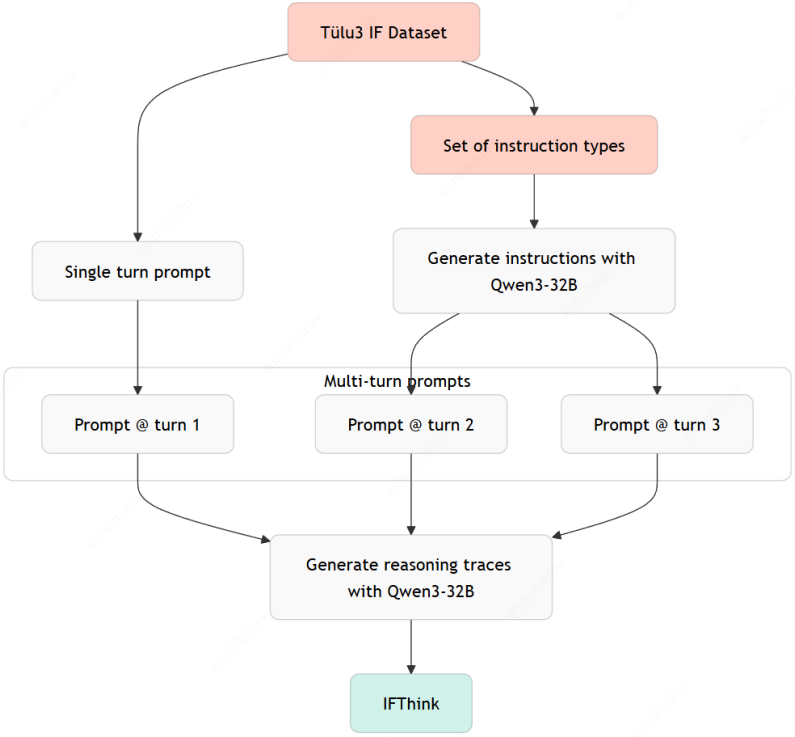


To fix this capability, we constructed a new dataset called IFThink. Based on the Multi-IF pipeline, we used single-turn instructions from Tulu 3's instruction-following subset and expanded them into multi-turn exchanges using Qwen3-32B to both generate both verifiable instructions and reasoning traces. The method is illustrated below:

为了修复此功能，我们构建了一个名为 IFThink 的新数据集。基于多中频管道，我们使用了 Tulu 3 指令跟随子集中的单轮指令，并使用 Qwen3-32B 将它们扩展为多轮交换，以生成可验

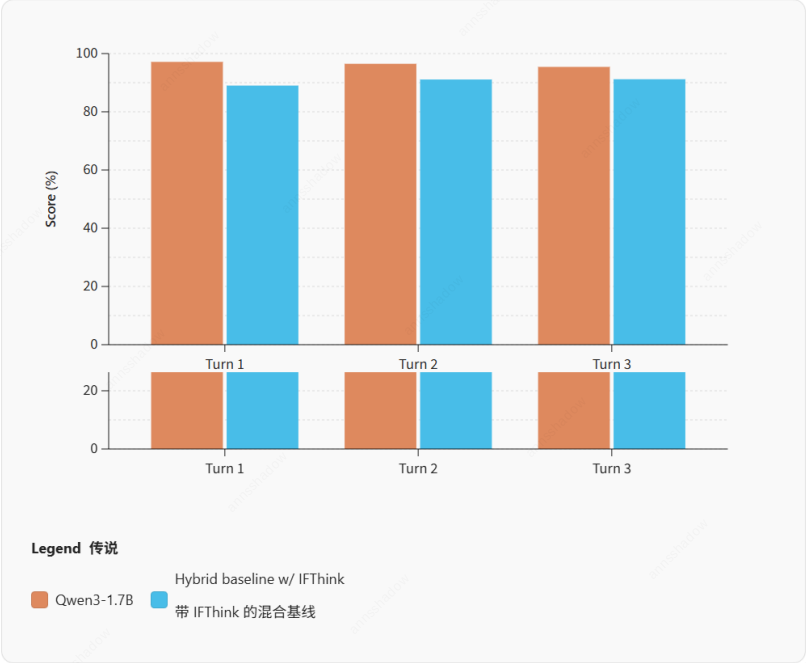
We considered filtering out conflicting instructions, but the initial results were strong enough to skip this step.

我们考虑过滤掉冲突的指令，但最初的结果足够强大，可以



Including this data in our baseline mix produced a dramatic improvement:

将这些数据纳入我们的基线组合产生了显着的改进：



After fixing the multi-turn reasoning issue with IFThink, our baseline finally behaved as intended; it could stay consistent across turns, follow instructions, and use the chat template correctly.

在修复了 IFThink 的多轮推理问题后，我们的基线终于按预期运行;它可以在各个回合保持一致，遵循说明并正确使用聊天模板。

With that foundation in place, we turned back to the basics: tuning the training setup itself.

有了这个基础，我们又回到了基础：调整训练设置本身。

WHICH HYPERPARAMETERS ACTUALLY MATTER?
哪些超参数真正重要?

In SFT, there are only a few hyperparameters that actually matter. Learning rate, batch size, and packing determine almost everything about how efficiently your model trains and how well it generalises.

在 SFT 中，只有几个超参数真正重要。学习率、批量大小和打包几乎决定了模型训练效率和泛化程度的所有因素。

In our baby baselines, we picked reasonable defaults just to validate the data and chat template. Now that the setup was stable, we revisited these choices to see how much

impact they have on our baseline.

在我们的婴儿基线中，我们选择了合理的默认值，只是为了验证数据和聊天模板。现在设置已经稳定，我们重新审视了这些选择，看看它们对我们的基线有多大影响。

Masking user turns 屏蔽用户轮次

One subtle design choice for the chat template is whether to mask the user turns during training. In most chat-style datasets, each training example consists of alternating user and assistant messages (possibly with interleaved tool calls).

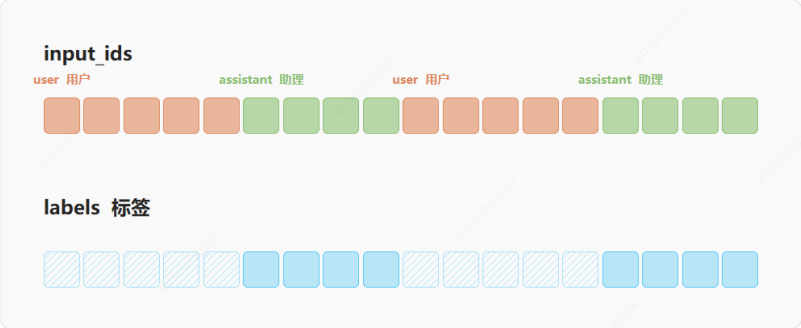
聊天模板的一个微妙设计选择是是否在训练期间屏蔽用户轮次。在大多数聊天式数据集中，每个训练示例都由交替的用户和助手消息组成（可能带有交错的工具调用）。

If we train the model to predict all tokens, it effectively learns to autocomplete user queries, rather than focusing on producing high-quality assistant responses.

如果我们训练模型来预测所有标记，它就会有效地学习自动完成用户查询，而不是专注于生成高质量的助手响应。

As shown in the figure below, masking user turns prevents this by ensuring the model's loss is only computed on assistant outputs, not user messages:

如下图所示，屏蔽用户轮次可以通过确保仅在助手输出而不是用户消息上计算模型的损失来防止这种情况：



In TRL, masking is applied for chat templates that can return the assistant tokens mask. In practice, this involves including a `{% generation %}` keyword in the template as follows:

在 TRL 中，对可以返回助手令牌掩码的聊天模板应用屏蔽。在实践中，这涉及在模板中包含一个 `{% generation %}` 关键字，如下所示：

```
{%- for message in messages -%}
  {%- if message.role == "user" -%}
    {{ "<|im_start|>" + message.role + "\n" + message.content + "<|im_end|>\n" }}
  {%- elif message.role == "assistant" -%}
{% generation %}
{{ "<|im_start|>assistant" + "\n" + message.content + "<|im_end|>\n" }}
{% endgeneration %}
  {%- endif %}
{%- endfor %}
{%- if add_generation_prompt %}
  {{ "<|im_start|>assistant\n" }}
{%- endif %}
```

Then, when `apply_chat_template()` is used with `return_assistant_tokens_mask=True` the chat template will indicate which parts of the dialogue should be masked. Here's a simple example, which shows how the assistant tokens are given ID 1, while the user tokens are masked with ID 0:

然后，与聊天模板一起使用 `return_assistant_tokens_mask=True` 时 `apply_chat_template()` 将指示对话的哪些部分应该被屏蔽。下面是一个简单的示例，它显示了如何为助手令牌提供 ID 1，而用户令牌如何使用 ID 0 进行屏蔽：

```
chat_template = '''
{%- for message in messages -%}
  {%- if message.role == "user" -%}
    {{ "<|im_start|>" + message.role + "\n" + message.content + "<|im_end|>\n" }}
  {%- elif message.role == "assistant" %}
    {% generation %}
    {{ "<|im_start|>assistant" + "\n" + message.content + "<|im_end|>\n" }}
    {% endgeneration %}
  {%- endif %}
{%- endfor %}
{%- if add_generation_prompt %}
  {{ "<|im_start|>assistant\n" }}
{%- endif %}
'''
rendered_input = tok.apply_chat_template(messages, chat_template=chat_template, r
```

```
print(rendered_input)
## {'input_ids': [128011, 882, 198, 40, 2846, 4560, 311, 743, 709, 856, 12443, 11
```

In practice, masking doesn't have a huge impact on downstream evals and usually provides a few points improvement in most cases.

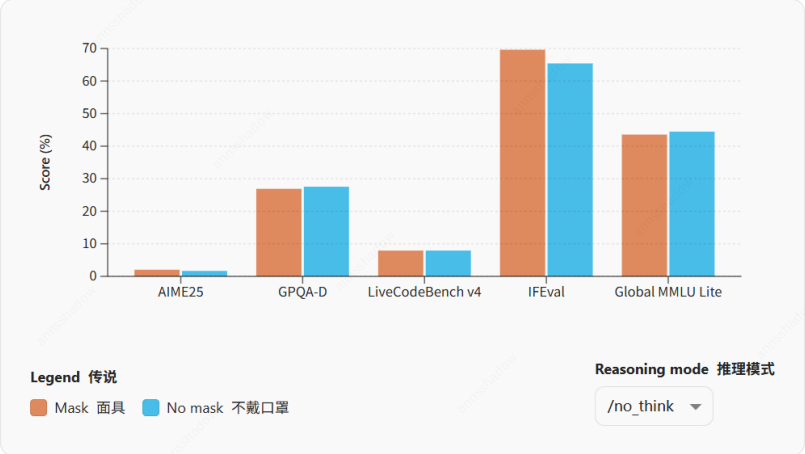
在实践中，掩码不会对下游评估产生巨大影响，并且在大多数情况下通常会提供一些点改进。

With SmolLM3, we found it had the most impact on IFEval, likely because the model is less inclined to restate the prompt and follow the various constraints more closely.

对于 SmolLM3，我们发现它对 IFEval 的影响最大，可能是因为模型不太倾向于重述提示并更严格地遵循各种约束。

A comparison of how user masking affected each eval and reasoning mode is shown below:

下面显示了用户掩码如何影响每种评估和推理模式的比较：



To pack or not to pack?

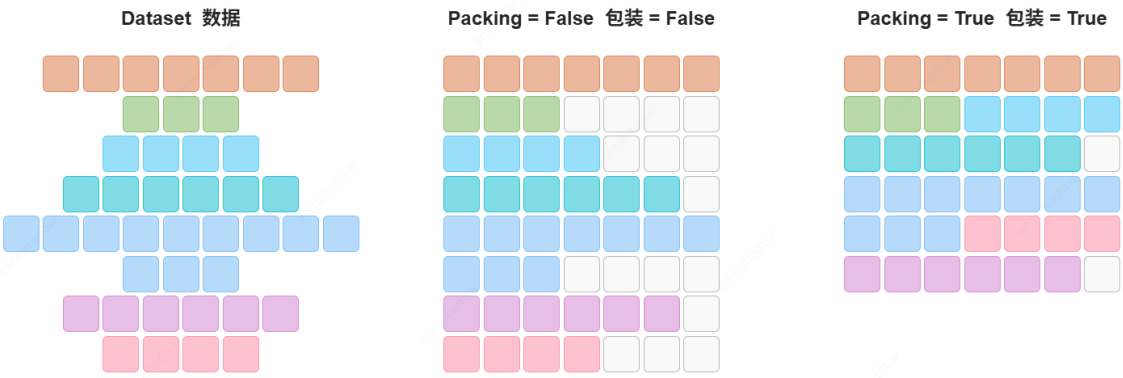
打包还是不打包?

As we Sequence packing is one of the training details that makes a huge difference to training efficiency. In SFT, most datasets contain samples of variable length, which means each batch contains a large number of padding tokens that waste compute and slow convergence.

正如我们所说，序列打包是对训练效率产生巨大影响的训练细节之一。在 SFT 中，大多数数据集都包含可变长度的样本，这意味着每个批次都包含大量填充标记，这些标记会浪费计算并缓慢收敛。

Packing solves this by concatenating multiple sequences together until a desired maximum token length is achieved. There are various ways to perform the concatenation, with TRL adopting a "best-fit decreasing" strategy (Ding et al., 2024), where the ordering of sequences to pack is determined by their length. As shown below, this strategy minimises truncation of documents across batch boundaries, while also reducing the amount of padding tokens:

打包通过将多个序列连接在一起直到达到所需的最大标记长度来解决这个问题。执行串联的方法有很多种，TRL 采用"最佳拟合递减"策略（Ding 等人，2024 年），其中要打包的序列的顺序由它们的长度决定。如下所示，此策略可最大程度地减少跨批处理边界的文档截断，同时减少填充令牌的数量：



Packing in post-training vs pre-training

打包训练后与训练前

In pre-training, this isn't really a question. When training on trillions of tokens, packing is essential to avoid wasting significant amounts of compute on padding. Pretraining frameworks like Megatron-LM and Nanotron implement packing by default. Post-training is different.

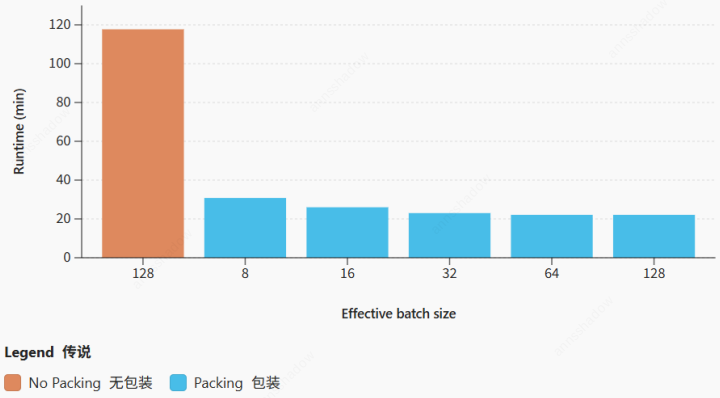
在预训练中，这并不是一个真正的问题。在数万亿个令牌上进行训练时，打包对于避免在填充上浪费大量计算至关重要。默认情况下，Megatron-LM 和 Nanotron 等预训练框架实现打包。培训后则不同。

Because the runs are shorter, the trade-offs change.

由于运行时间较短，因此权衡发生了变化。

To get a sense of how efficient packing is for training, below we compare the runtimes between packing and no-packing over one epoch of our baseline dataset:

为了了解打包对训练的效率，下面我们比较了基线数据集的一个时期内打包和不打包之间的运行时间：



The reason the runtime flattens out after an effective batch size of 32 is because this is the largest size possible without invoking gradient accumulation.

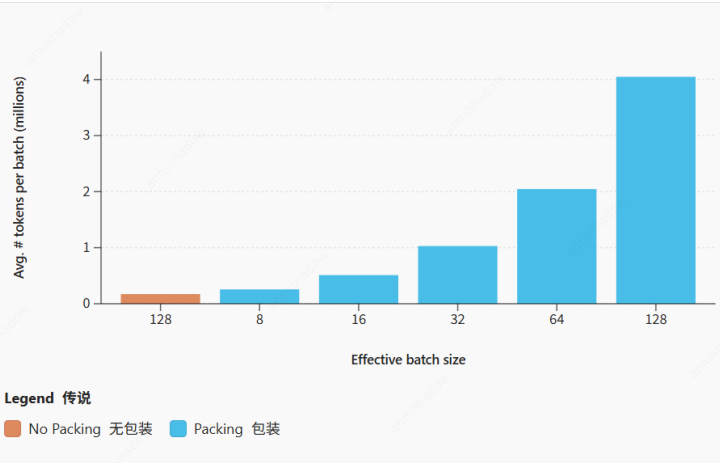
运行时在有效批量大小为 32 后变平的原因是，这是在不调用梯度累积的情况下可能的最大大小。

Depending on the batch size, we see that packing improves throughput by a factor of 3-5x! So, should you *always* use packing? To some extent, the answer depends on how large your dataset is, because packing reduces the number of optimisation steps per epoch by fitting more tokens into each step.

根据批量大小，我们看到包装可将吞吐量提高 3-5 倍！那么，你应该总是使用包装吗？在某种程度上，答案取决于数据集的大小，因为打包通过在每个步骤中插入更多标记来减少每个时期的优化步骤数。

You can see this in the following figure, where we plot the average number of non-padding tokens per batch:

您可以在下图中看到这一点，我们在其中绘制了每批次非填充令牌的平均数量：



With packing, the number of tokens per batch scales linearly with the batch size and compared to training without packing, can include up to 33x more tokens per optimisation step!

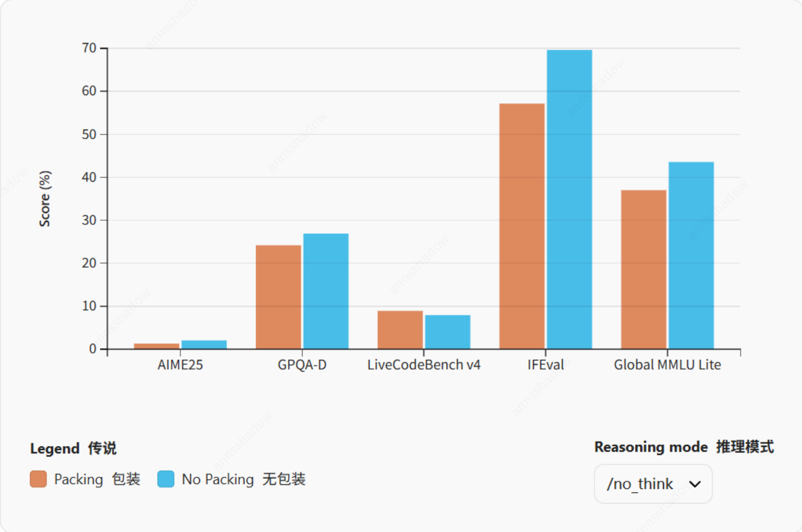
通过打包，每个批次的 token 数量与批次大小线性增长，与不打包的训练相比，每个优化步骤可以包含多达 33 倍的 token！

However, packing can slightly alter the training dynamics: while you process more data overall, you take fewer gradient updates which can influence final performance,

especially on small datasets where each sample matters more.

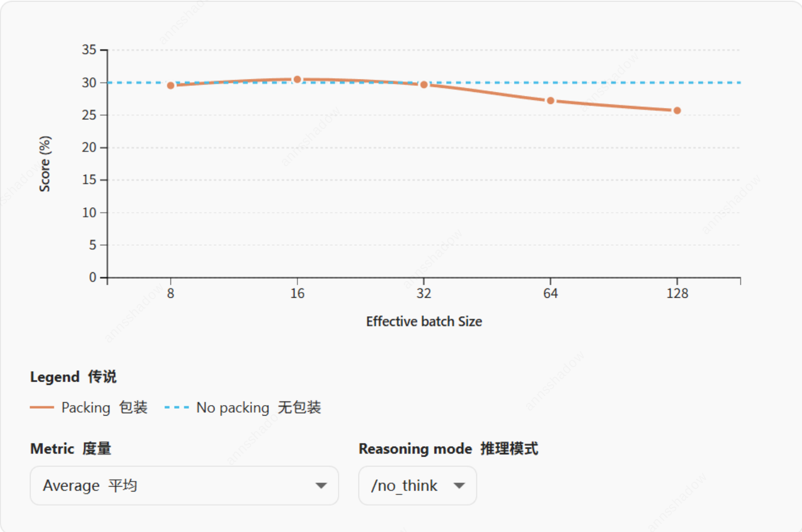
[illegible] nearly 10 percentage points:

例如，如果我们在相同的有效批量大小 128 下比较打包与无打包，我们会发现一些评估（如 IFEval）的性能受到了近 10 个百分点的显著影响：



More generally, we see that once the effective batch size is large than 32, there is an average drop in performance for this particular model and dataset:

更一般地说，我们看到，一旦有效批量大小大于 32，此特定模型和数据集的性能就会平均下降：



In practice, for large-scale SFT where the dataset is massive, packing is almost always beneficial since the compute savings far outweigh any minor differences in gradient frequency.

在实践中，对于数据集很大的大规模 SFT，打包几乎总是有益的，因为节省的计算远远超过梯度频率的任何细微差异。

However, for smaller or more diverse datasets—like domain-specific fine-tuning or instruction-tuning on limited human-curated data—it might be worth disabling packing to preserve sample granularity and ensure every example contributes cleanly to optimisation.

但是，对于较小或更多样化的数据集（例如特定领域的微调或对有限的人工管理数据进行指令调整），可能值得禁用打包以保留样本粒度并确保每个示例都干净地有助于优化。

Ultimately, the best strategy is empirical: start with packing enabled, monitor both throughput and downstream evals, and adjust based on whether the speed gains translate into equivalent or improved model quality.

最终，最好的策略是经验性的：从启用打包开始，监控吞吐量和下游评估，并根据速度增益是否转化为同等或改进的模型质量进行调整。

Tuning the learning rate 调整学习率

We now come to the last, but still important hyperparameter: the learning rate. Set it too high and training may diverge; too low and convergence is painfully slow.

我们现在来谈谈最后一个但仍然重要的超参数：学习率。设置得太高，训练可能会发散;太低，

收敛速度非常慢。

In SFT, the optimal learning rate is typically an order of magnitude (or more) smaller than the one used during pretraining. This is because we're initialising from a model with rich representations, and aggressive updates can lead to catastrophic forgetting.

在 SFT 中，最佳学习率通常比预训练期间使用的学习率小一个数量级（或更多）。这是因为我们是从具有丰富表示的模型进行初始化的，而激进的更新可能会导致灾难性的遗忘。

> **Tuning the learning rate in post-training vs pretraining**
>
> **调整训练后与预训练的学习率**
>
> Unlike pre-training, where hyperparameter sweeps on the full run are prohibitively expensive, post-training runs are short enough that we can actually do full learning rate sweeps.
>
> 与预训练不同，在预训练中，全程运行的超参数扫描成本高得令人望而却步，而训练后运行足够短，我们实际上可以进行全学习率扫描。

In our experiments, we've found that the "best" learning rate varies with both model family, size and the use of packing. Since a high learning rate can lead to exploding gradients, we find it's often safer to slightly decrease the learning rate when packing is enabled.

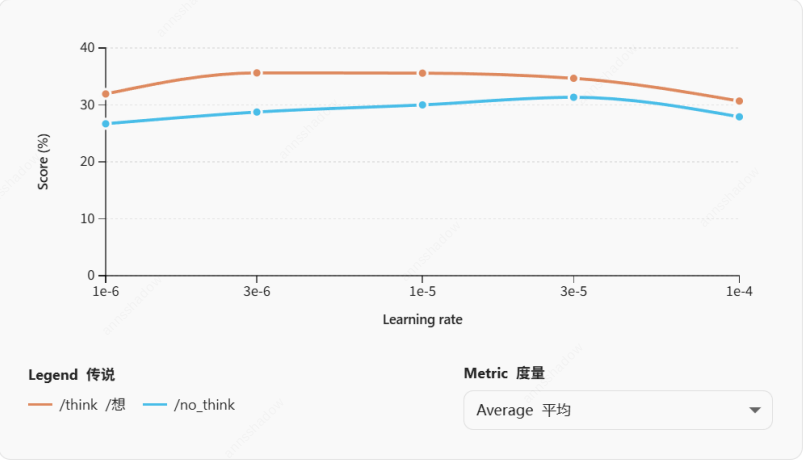
在我们的实验中，我们发现"最佳"学习率因模型系列、大小和包装的使用而异。由于高学习率会导致梯度爆炸，因此我们发现在启用打包时稍微降低学习率通常更安全。

When picking a range of learning rate values to scan over, we find is useful to pick an initial range like [1e-6, 3e-6, 1e-5, 3e-5, 1e-4]. This covers two orders of magnitude and allows us to hone in on a region where some additional tuning can be applied

在选择要扫描的学习率值范围时，我们发现选择初始范围（如 [1e-6， 3e-6， 1e-5， 3e-5， 1e-4] 很有用。这涵盖了两个数量级，使我们能够专注于可以应用一些额外调音的区域

You can see this below, where using a small learning rate of 3e-6 or 1e-5 gives better overall performance than large values:

您可以在下面看到这一点，其中使用 3e-6 或 1e-5 的小学习率比大值提供更好的整体性能：



Although a few points on average may not seem like much, if you look at individual benchmarks like AIME25, you'll see the performance drop dramatically when the learning rate is larger than 1e-5.

虽然平均几分看起来并不多，但如果你查看像 AIME25 这样的个别基准测试，你会发现当学习率大于 1e-5 时，性能会急剧下降。

Scaling the number of epochs

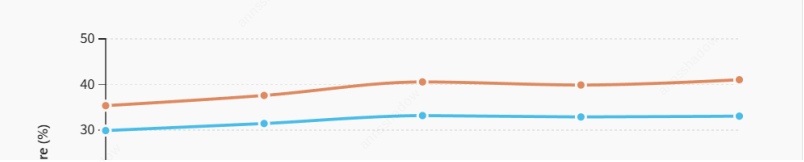
缩放纪元数

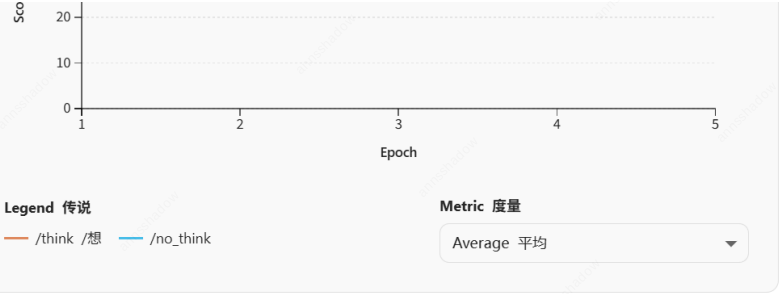
In our ablations, we usually train for a single epoch to iterate quickly. Once you've identified a good data mixture and tuned key parameters like the learning rate, the next step is to increase the number of epochs for final training.

在我们的消融中，我们通常会训练单个 epoch 以快速迭代。一旦确定了良好的数据组合并调整了学习率等关键参数，下一步就是增加最终训练的纪元数。

For example, if we take our baseline data mixture and train for five epochs, we see it is possible to squeeze a few more percentage points of performance on average:

例如，如果我们采用基线数据混合并训练五个纪元，我们会发现平均可以再挤压几个百分点的性能：





As we saw with the learning rate scan, the average performance obscures the impact that scaling the number of epochs has on individual evals: in the case of LiveCodeBench v4 with extended thinking, we nearly double the performance over one epoch!

正如我们在学习率扫描中看到的那样，平均性能掩盖了扩展纪元数对单个评估的影响：在具有扩展思维的 LiveCodeBench v4 的情况下，我们在一个纪元内将性能提高了近一倍！

Once you've iterated on your SFT data mixture and the model has reached a reasonable level of performance, the next step is often to explore more advanced methods like [preference optimisation](#from-sft-to-preference-optimisation:-teaching-models-what- *better*- means) or reinforcement learning. However, before diving into those, it's worth considering whether the additional compute would be better spent on strengthening the base model through *continued pretraining*.

teaching-models-what-*better*-means) or reinforcement learning. However, before diving into those, it's worth considering whether the additional compute would be better spent on strengthening the base model through *continued pretraining*.

一旦你迭代了你的 SFT 数据混合并且模型达到了合理的性能水平，下一步通常是探索更高级的方法，例如 [偏好优化]（#from-sft-to-preference-optimisation：-teaching-models-what- better- means） 或强化学习。然而，在深入研究这些之前，值得考虑一下额外的计算是否更适合通过持续的预训练来加强基础模型。

> 📌 **Optimisers in post-training**
>
> **培训后的优化器**
>
> Another important component we mentioned in the pre-training section is the optimiser. Similarly, AdamW remains the default choice for post-training. An open question is whether models pre-trained with alternative optimisers like Muon should be post-trained with the *same* optimiser. The Kimi team found that using the same optimise for pre- and post-training yielded best performance for their Moonlight model.
>
> 我们在预训练部分提到的另一个重要组件是优化器。同样，AdamW 仍然是后期训练的默认选择。一个悬而未决的问题是，是否应该使用相同的优化器对使用替代优化器进行预训练的模型。Kimi 团队发现，在训练前和训练后使用相同的优化可以为他们的 Moonlight 模型带来最佳性能。

BOOSTING REASONING THROUGH CONTINUED PRETRAINING
通过持续的预训练提高推理能力

Continued pretraining—or mid-training if you want to sound fancy—means taking a base model and training it further on large amounts of domain-specific tokens before doing SFT.

持续的预训练（或者如果您想听起来很花哨的话）意味着在进行 SFT 之前，采用基础模型并在大量特定于领域的令牌上进一步训练它。

Mid-training is useful when your target capabilities for SFT share a common core skill, such as coding or reasoning. In practice, this shifts the model toward a distribution that better supports reasoning, a specific language, or any other capability you care about.

当 SFT 的目标能力共享共同的核心技能（例如编码或推理）时，中期训练非常有用。在实践中，这将模型转向更好地支持推理、特定语言或您关心的任何其他功能的分布。

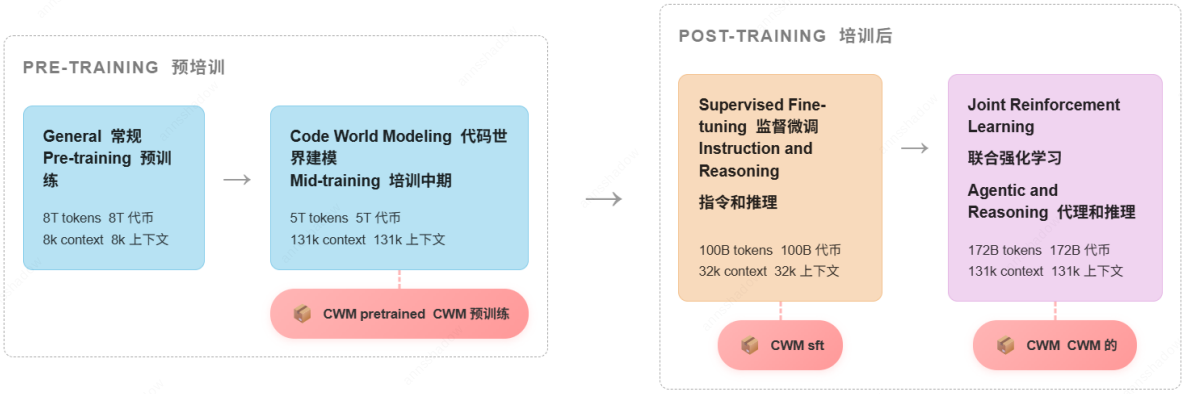
Starting SFT from a model that has already integrated that core skill allows your model to better focus on the specific topics in your SFT data rather than using compute to learn the core skill from scratch.

从已集成该核心技能的模型启动 SFT，使模型能够更好地专注于 SFT 数据中的特定主题，而不是使用计算从头开始学习核心技能。

The mid-training approach traces back to ULMFit (Howard & Ruder, 2018), which pioneered the three-stage pipeline of general pretraining → mid-training → post-

training that is now common in modern LLMs like FAIR's Code World Model (team et al., 2025):

中期训练方法可以追溯到 ULMFit（Howard & Ruder，2018），它开创了通用预训练→中期训练→后训练的三阶段管道，这现在在现代 LLM 中很常见，例如 FAIR 的代码世界模型（团队等人，2025 年）：



This approach was also used in the training of Phi-4-Mini-Reasoning (Xu et al., 2025), but with a twist: instead of doing continued pretraining on web data, the authors used distilled reasoning tokens from DeepSeek-R1 for the mid-training corpus. The results were compelling, showing consistent and large gains through multi-stage training:

这种方法也用于（Xu et al.， 2025）的Phi-4-Mini-Reasoning训练，但有一个转折点：作者没有对 Web 数据进行持续的预训练，而是使用 DeepSeek-R1 中提炼的推理令牌进行中期训练语料库。结果令人信服，通过多阶段训练显示出一致且巨大的收益：

Model 型	AIME24 AIME24 系列	MATH-500 数学-500	GPQ/
Phi-4-Mini Phi-4-迷你	10.0	71.8	36.9
+ Distill Mid-training + 蒸馏中训练	30.0	82.9	42.6
+ Distill Fine-tuning + 蒸馏微调	43.3	89.3	48.3
+ Roll-Out DPO + 推出 DPO	50.0	93.6	49.0
+ RL (Phi-4-Mini-Reasoning) + RL （Phi-4-Mini-Reasoning）	**57.5**	**94.6**	**52.0**

These results prompted us to try a similar approach. From our prior experience with creating and evaluating reasoning datasets in Open-R1, we had three main candidates to work with:

这些结果促使我们尝试类似的方法。根据我们之前在 Open-R1 中创建和评估推理数据集的经验，我们有三个主要候选者可以使用：

- **Mixture of Thoughts**：350k reasoning samples distilled from DeepSeek-R1 across math, code, and science.

  思想的混合：从 DeepSeek-R1 中提炼的 350k 推理样本，涵盖数学、代码和科学。

- **Llama-Nemotron-Post-Training-Dataset:** NVIDIA's large-scale dataset of distilled from a wide variety of models such as Llama3 and DeepSeek-R1. We filtered the dataset for the DeepSeek-R1 outputs, which resulted in about 3.64M samples or 18.7B tokens.

  Llama-Nemotron-Post-Training-Dataset：NVIDIA 的大规模数据集，从 Llama3 和 DeepSeek-R1 等多种模型中提炼而成。我们过滤了 DeepSeek-R1 输出的数据集，结果产生了大约 3.64M 个样本或 18.7B 个标记。

- **OpenThoughts3-1.2M:** one of the highest-quality reasoning datasets, with 1.2M samples distilled from QwQ-32B, comprising 16.5B tokens.

  OpenThoughts3-1.2M：最高质量的推理数据集之一，从 QwQ-32B 中提炼出 1.2M 个样本，包括 16.5B 个标记。

Since we planned to include reasoning data in the final SFT mix, we decided to keep Mixture of Thoughts for that stage the others for mid-training. We used ChatML as the chat template to avoid "burning in" the SmolLM3 one too early on.

由于我们计划在最终的 SFT 组合中包含推理数据，因此我们决定将该阶段的 Mixture of Thoughts 保留在训练中期的其他阶段。我们使用 ChatML 作为聊天模板，以避免过早地“烧入”SmolLM3。

We also trained for 5 epochs with a learning rate of 2e-5, using 8 nodes to accelerate training with an effective batch size of 128.

我们还训练了 5 个 epoch，学习率为 2e-5，使用 8 个节点加速训练，有效批量大小为 128。

> 📍 **When to mid-train? 什么时候进行中途训练？**
>
> You might wonder why we're discussing mid-training *after* we did some SFT runs. Chronologically, mid-training happens before SFT on the base model. But the decision to do mid-training only becomes clear after you've run initial SFT experiments and identified performance gaps.
>
> 您可能想知道为什么我们在进行一些 SFT 运行后讨论训练中期。按时间顺序，中间训练发生在基础模型上的 SFT 之前。但是，只有在您运行初始 SFT 实验并确定性能差距后，进行中期训练的决定才会变得清晰。
>
> In practice, you'll often iterate: run SFT to identify weak areas, then do targeted mid-training, then run SFT again.
>
> 在实践中，您经常会迭代：运行 SFT 以识别薄弱环节，然后进行有针对性的中间训练，然后再次运行 SFT。
>
> Think of this section as**"what to do when SFT alone isn't enough."**
>
> 将本节视为“当仅靠 SFT 还不够时该怎么办”。

The mystery of the melting GPUs

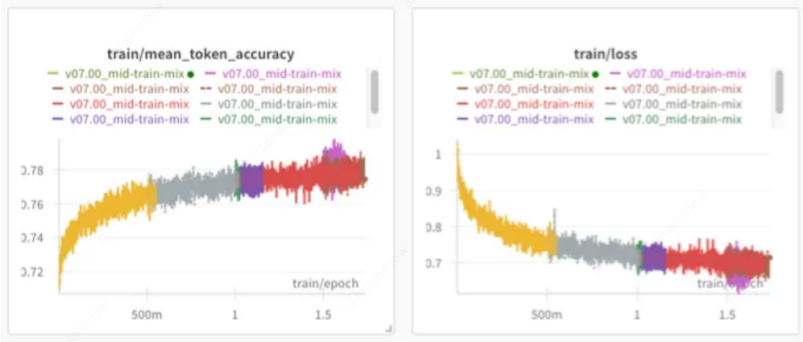
熔化 GPU 的奥秘

Running these experiments turned out to be a surprising challenge on our cluster: the aging GPUs would get throttled at various points which would lead to hardware failures and forced restarts of each run.

事实证明，运行这些实验对我们的集群来说是一个令人惊讶的挑战：老化的 GPU 会在不同点受到限制，这将导致硬件故障和每次运行的强制重启。

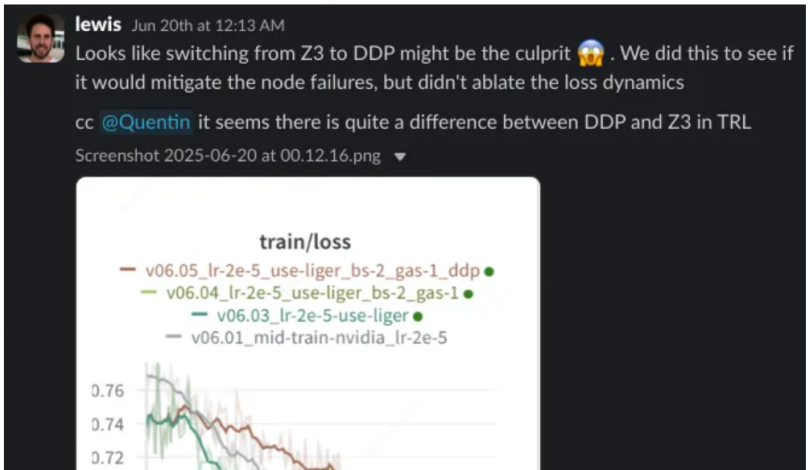
To give you a taste of what it was like, here's the logs from one of the runs, where each colour represents a restart:

为了让您体验一下它是什么样子，以下是其中一次运行的日志，其中每种颜色代表重新开始：



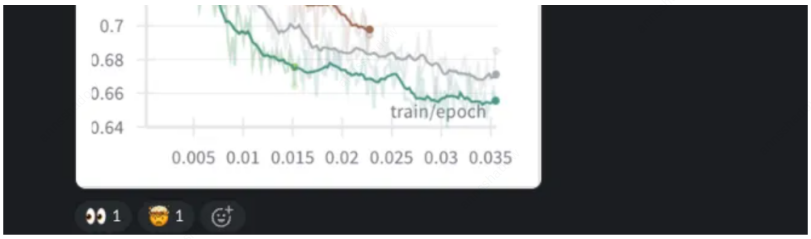
We initially thought DeepSpeed might be the culprit, since the accelerator is highly optimised for throughput. To test this, we switched to DP, which helped somewhat, but then the loss was dramatically different!

我们最初认为 DeepSpeed 可能是罪魁祸首，因为该加速器针对吞吐量进行了高度优化。为了测试这一点，我们切换到 DP，这有一定帮助，但随后损失却大不相同！



Finding bugs in your code at midnight is more common than you might think. In hindsight, for long runs at this scale, it would have made more sense to use nanotron since it was battle-tested and had faster throughput.

在午夜发现代码中的错误比您想象的更常见。事后看来，对于这种规模的长期运行，使用纳米电子会更有意义，因为它经过了实战测试并且具有更快的吞吐量。



As we later discovered, a bug with DP in Accelerate meant that the weights and gradients were stored in the model's native precision (BF16 in this case), which led to numerical instability and loss of gradient accuracy during accumulation and optimisation.

正如我们后来发现的那样，Accelerate 中的 DP 存在一个错误，这意味着权重和梯度存储在模型的原始精度（在本例中为 BF16）中，这导致在累积和优化过程中出现数值不稳定和梯度精度损失。

To prevent this, most accelerators use FP32 for the "master weights" and optimiser states, and only cast back to BF16 for the forward and backward passes.

为了防止这种情况，大多数加速器将 FP32 用于"主权重"和优化器状态，并且仅在前向和后向传递时投回 BF16。

So we switched back to DeepSpeed and added aggressive checkpointing to minimise the time lost from GPUs overheating and "falling off the bus". This strategy proved successful and is something we recommend more generally:

因此，我们切换回 DeepSpeed 并添加了积极的检查点，以最大限度地减少 GPU 过热和"从总线上掉下来"造成的时间损失。事实证明，这种策略是成功的，我们更普遍地推荐这种策略：

> ☝ **Rule 统治**
>
> As we emphasized in pre-training, save model checkpoints frequently during a

training run, and ideally push them to the Hugging Face Hub to avoid accidental overwrites. Also, make your training framework robust to failures and capable of automatic restarts.

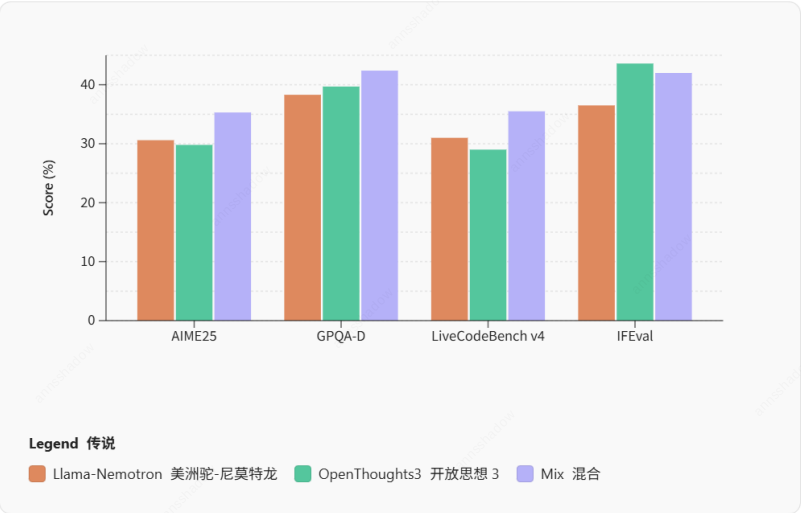
正如我们在预训练中强调的那样，在训练运行期间经常保存模型检查点，理想情况下将它们推送到 Hugging Face Hub 以避免意外覆盖。此外，使训练框架能够抵御故障并能够自动重新启动。

Both of these strategies will save you time, especially for long-running jobs like mid-training ones.

这两种策略都可以节省您的时间，尤其是对于长期运行的工作，例如培训中期的工作。

After babysitting the runs for a week or so, we finally had our results:

在照顾跑步一周左右后，我们终于有了结果：

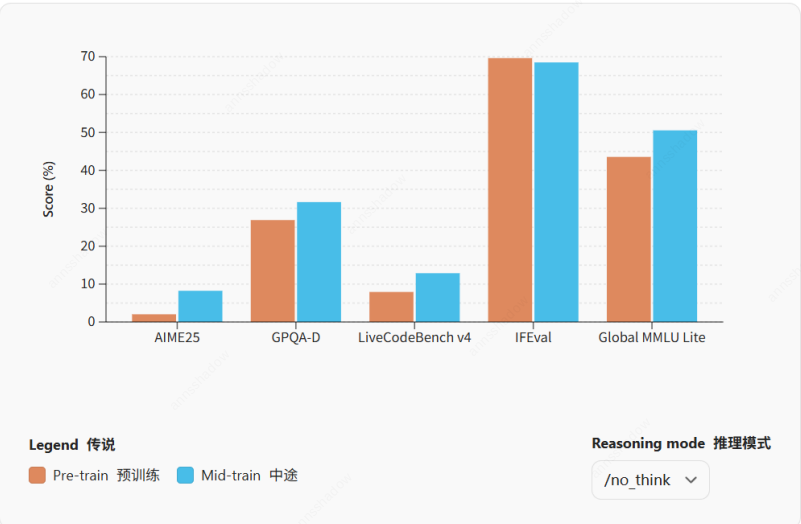


Overall, we found that NVIDIA's post-training dataset gave better performance than OpenThoughts, but the combination was best overall.

总体而言，我们发现 NVIDIA 的训练后数据集比 OpenThoughts 提供了更好的性能，但总体组合是最好的。

Now let's take a look at the effect of taking one of these checkpoints and applying our same baseline data mixture:

现在让我们来看看采用其中一个检查点并应用相同的基线数据混合的效果：



The effect of using a mid-trained reasoning model instead of the pre-trained one are dramatic: with extended thinking, we nearly triple the performance on AIME25 and LiveCodeBench v4, while GPQA-D receives a full 10 point gain.

使用中训练推理模型而不是预训练推理模型的效果是显着的：通过扩展思维，我们在 AIME25 和 LiveCodeBench v4 上的性能几乎提高了两倍，而 GPQA-D 获得了整整 10 分的增益。

Somewhat surprisingly, the reasoning core also translated partially to the `/no_think` reasoning mode, with about 4-6 point improvements on the reasoning benchmarks.

有点令人惊讶的是，推理核心也部分转化为 `/no_think` 推理模式，在推理基准测试上提高了大约 4-6 个百分点。

These results gave us clear evidence that for reasoning models, it almost always makes sense to perform some amount of mid-training if your base model hasn't already seen a lot of reasoning data during pre-training.

这些结果为我们提供了明确的证据，即对于推理模型，如果您的基础模型在预训练期间还没有看到大量推理数据，那么执行一定数量的中期训练几乎总是有意义的。

> 📍 **When not to mid-train**
>
> **什么时候不在火车中**
>
> Mid-training shines when your model must learn a new, core skill. It's less useful when the base model already has the skill or if you're trying to elicit shallow capabilities like style or conversational chit-chat.
>
> 当模型必须学习一项新的核心技能时，训练中期就会大放异彩。当基本模型已经具备该技能，或者你试图引出风格或对话式闲聊等浅层功能时，它就不太有用了。
>
> In these cases, we recommend skipping mid-training and allocating your compute to other methods like preference optimisation or reinforcement learning.
>
> 在这些情况下，我们建议跳过训练中期，并将计算分配给其他方法，例如偏好优化或强化学习。

Once you're confident in your SFT data mixture and the model's broad capabilities, the focus naturally shifts from learning skills to refining them. In most cases, the most effective way forward is *preference optimisation*.

一旦您对 SFT 数据混合和模型的广泛功能充满信心，重点自然会从学习技能转移到完善技能。在大多数情况下，最有效的方法是优化偏好。

From SFT to preference optimisation: teaching models what *better* means

从 SFT 到偏好优化：教学模型意味着什么更好

Although you can keep scaling SFT with more data, at some point you'll observe diminishing gains or failure modes like your model being unable to fix its own buggy code. Why? Because SFT is a form of *imitation learning*, and so the model only learns to reproduce patterns in the data it was trained on. If the data doesn't already contain good fixes, or if the desired behaviour is hard to elicit with distillation, the model has no clear signal for what counts as "better".

尽管您可以使用更多数据继续扩展 SFT，但在某些时候，您会观察到增益递减或故障模式，例如您的模型无法修复自己的错误代码。为什么？因为 SFT 是一种形式 模仿学习 ，因此模型只学习在它训练的数据中重现模式。如果数据还没有包含良好的修复，或者如果通过蒸馏很难引出所需的行为，则该模型没有明确的信号来表明什么算作"更好"。

The problem persists even if your dataset contains an even mix of traces (i.e. some that reach the correct solution immediately, and others where the model first makes a mistake and corrects it).

即使您的数据集包含均匀混合的跟踪（即一些立即到达正确解决方案，而另一些则模型首先犯错误并纠正它），问题仍然存在。

In this case, the model may simply learn that making an initial error is part of the desired pattern. What we actually want, of course, is a model that can produce the correct solution from the start.

在这种情况下，模型可能只是了解到犯初始错误是所需模式的一部分。当然，我们真正想要的是一个能够从一开始就产生正确解决方案的模型。

This is where preference optimisation comes in. Instead of just copying demonstrations, we give the model comparative feedback like "response A is better than response B".

这就是偏好优化的用武之地。我们不只是复制演示，而是给模型一些比较反馈，比如"响应 A 比响应 B 更好"。

These preferences provide a more direct training signal for quality and enable to model performance to scale beyond the limits of SFT alone.

这些偏好为质量提供了更直接的训练信号，并能够对性能进行建模，使其超出单独 SFT 的限制。

Another benefit of preference optimisation is that you typically need far less data than SFT, since the starting point is already a pretty good model that can follow instructions and has knowledge from previous training stages.

偏好优化的另一个好处是，您通常需要的数据比 SFT 少得多，因为起点已经是一个非常好的模型，可以遵循指令并具有先前训练阶段的知识。

Let's take a look at how these datasets are created.

让我们来看看这些数据集是如何创建的。

CREATING PREFERENCE DATASETS

创建首选项数据集

Historically, preference datasets were created by providing human annotators with pairs of model responses and asking them to grade which one is better (possibly on a scale). This approach is still used by LLM providers to collect *human preference* labels, but it is extremely expensive and scales poorly. Recently, LLMs have become capable of producing high-quality responses, and often in a cost-effective way. These

advances make it practical for LLMs to *generate* preferences for many applications. In practice, there are two common approaches:

从历史上看，偏好数据集是通过向人类注释者提供成对的模型响应并要求他们对哪一个更好进行评分（可能按比例）来创建的。LLM 提供商仍在使用这种方法来收集人类偏好标签，但它非常昂贵且扩展性很差。最近，法学硕士已经能够以一种具有成本效益的方式产生高质量的响应。这些进步使得法学硕士可以为许多应用程序生成偏好。在实践中，有两种常见的方法：

Strong vs. weak 强与弱

1. Take a fixed set of prompts x (often curated for coverage and difficulty).

   采用一组固定的提示 x （通常针对覆盖范围和难度进行策划）。

2. Generate one response from a weaker or baseline model, and another from a high-performing model.

   从较弱的模型或基线模型生成一个响应，从高性能模型生成另一个响应。

3. Label the stronger model's output as the chosen response y_c and the weaker one as rejected y_r .

   将较强模型的输出标记为所选响应，将较弱的输出 y_c 标记为拒绝 y_r 。

This produces a dataset of "stronger vs. weaker" comparisons $(\{x, y_c, y_r\})$, which is simple to construct because we assume the stronger model's output is reliably better.

这会产生一个"更强与更弱"比较的数据集 $(\{x, y_c, y_r\})$ ，该数据集很容易构建，因为我们假设更强模型的输出可靠地更好。

Below is a popular example from Intel, who took an SFT dataset with responses from gpt-3.5 and gpt-4 and converted it into a preference dataset by selecting the gpt-4 responses as chosen and the gpt-3.5 ones as rejected:

以下是英特尔的一个流行示例，英特尔采用了一个包含 gpt-3.5 和 gpt-4 响应的 SFT 数据集，并通过选择选择 gpt-4 响应和拒绝 gpt-3.5 响应将其转换为偏好数据集：

On-policy with grading 符合政策，分级

1. Use the *same model* you will train to generate multiple candidate responses to the same prompt. This creates data that is "on-policy" because it reflects the distribution of outputs the model would naturally produce.

   使用您将训练的同一模型来生成对同一提示的多个候选响应。这会创建"符合策略"的数据，因为它反映了模型自然产生的输出分布。

2. Instead of relying on a stronger model as the reference, introduce an *external grader:* either a verifier or a reward model that scores responses along one or more quality axes (e.g., helpfulness or factual accuracy).

   不要依赖更强大的模型作为参考，而是引入外部评分器：验证者或奖励模型，沿一个或多个质量轴（例如，有用性或事实准确性）对响应进行评分。

3. The grader then assigns preference labels among the candidate responses, producing a more nuanced and flexible preference dataset.

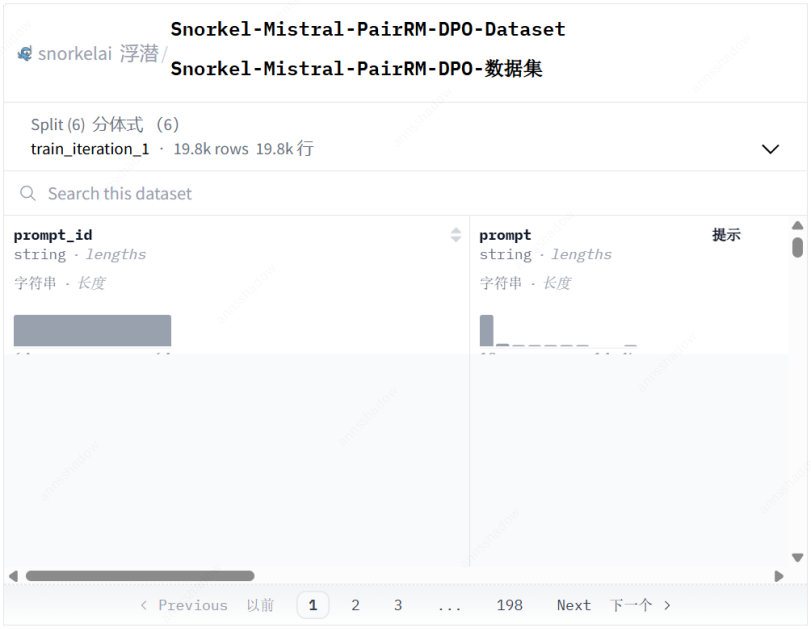
   然后，评分器在候选响应中分配偏好标签，生成更细致、更灵活的偏好数据集。

This method allows ongoing bootstrapping of preference data as the model improves, but its quality depends heavily on the evaluator's reliability and calibration.

随着模型的改进，这种方法允许持续引导偏好数据，但其质量在很大程度上取决于评估者的可靠性和校准。

A nice example of such a dataset is from SnorkelAI, who took the prompts from a popular preference dataset called UltraFeedback, partitioned them into 3 sets, and then applied the above recipe iteratively to improve their model:

SnorkelAI 就是一个很好的例子，他从一个名为 UltraFeedback 的流行偏好数据集中获取提示，将它们分成 3 组，然后迭代应用上述配方来改进他们的模型：



At the time of SmolLM3's development, there did not exist any preference data with reasoning traces, so we decided to generate some of our own using the "strong vs weak" approach. We used the prompts from Ai2's Tulu 3 preference mixture to generate responses from Qwen3-0.

在 SmolLM3 开发时，不存在任何带有推理痕迹的偏好数据，因此我们决定使用"强与弱"方法生成一些我们自己的数据。我们使用 Ai2 的 Tulu 3 偏好混合中的提示来生成来自 Qwen3-0 的响应。

6B and Qwen3-32B in the `/think` mode. The result was a large-scale dataset of 250k+ LLM-generated preferences, ready to simultaneously improve our SFT checkpoint across multiple axes using preference optimisation algorithms.

6B 和 Qwen3-32B 模式。`/think` 结果是一个包含 250k+ LLM 生成的偏好的大规模数据集，准备使用偏好优化算法同时改进我们的跨多个轴的 SFT 检查点。

WHICH ALGORITHM DO I PICK?
我选择哪种算法？

Direct Preference Optimization (DPO) (Rafailov et al., 2024) was the first preference optimisation algorithm to gain widespread adoption in open source.

直接偏好优化 （DPO）（Rafailov 等人，2024 年）是第一个在开源中广泛采用的偏好优化算法。

Its appeal came from being simple to implement, stable in practice, and effective even with modest amounts of preference data. As a result, DPO has become the default method to improve SFT models before reaching for more complex techniques like RL.

它的吸引力在于易于实施、实践稳定，即使使用少量偏好数据也有效。因此，DPO 已成为在采用更复杂的技术（如 RL）之前改进 SFT 模型的默认方法。

But researchers quickly discovered there are many ways to improve upon DPO, and nowadays there is a wide variety of alternatives to explore. Below we list a few of the ones we've found most effective:

但研究人员很快发现有很多方法可以改进 DPO，如今有各种各样的替代方案可供探索。下面我们列出了一些我们发现最有效的方法：

- **Kahneman-Tversky Optimisation (KTO) [** Ethayarajh et al. (2024) **]:** Instead of relying on preference pairs, KTO models whether an individual response is "desirable or not", using ideas from human decision making. This is a good choice if you don't have access to paired preference data (e.g. raw responses like 👍 or 👎 collected from end-users).

When the DPO paper came out in mid-2023, there was heated debate online about whether it could match RL methods, and there were no recipes showing it's effectiveness in industrial settings. To address that, we released Zephyr 7B a few months later, training the entire model on synthetic data and showing significant performance gains from DPO.

当 DPO 论文于 2023 年年中发布时，网上就它是否可以与 RL 方法相媲美展开了激烈的争论，并且没有配方表明它在工业环境中的有效性。为了解决这个问题，我们在几个月后发布了 Zephyr 7B，根据合成数据训练整个模型，并显示出 DPO 的显着性能提升。

Kahneman-Tversky 优化 （KTO） [Ethayarajh 等人 （2024）]：KTO 不依赖偏好对，而是使用人类决策的想法对个人反应是否“可取”进行建模。如果您无法访问配对偏好数据（e.g. raw 响应，👍例如最终用户或👎从最终用户收集），这是一个不错的选择。

- **Odds Ratio Preference Optimisation (ORPO) [** Hong et al. (2024) **]:** integrates preference optimisation directly into SFT by adding an odds ratio to the cross-entropy loss. As a result there is no need for a reference model or SFT stage, which makes this method more computationally efficient.

  优势比偏好优化 （ORPO） [Hong et al. （2024）]：通过将优势比添加到交叉熵损失中，将偏好优化直接集成到 SFT 中。因此，不需要参考模型或 SFT 级，这使得该方法的计算效率更高。

- **Anchored Preference Optimisation (APO) [** D'Oosterlinck et al. (2024) **]:** this is a more controllable objective the explicitly regularises how much the model's likelihoods for chosen vs. rejected outputs should shift, rather than just optimising their difference.

  锚定偏好优化 （APO） [D'Oosterlinck et al. （2024）]：这是一个更可控的目标，它明确地正则化了模型对选择输出与拒绝输出的可能性应该变化多少，而不仅仅是优化它们的差异。

  There are two variants (APO-zero and APO-down) who's choice depends on the relationship between your model and the preference data, i.e. whether the chosen outputs are better than the model or worse.

  有两种变体（APO-zero 和 APO-down），它们的选择取决于您的模型和偏好数据之间的关系，即所选输出是比模型好还是差。

Luckily, many of these choices are just a one-line change in TRL's `DPOTrainer`, so for our initial baseline we did the following:

幸运的是，其中许多选择只是 TRL 中的一行更改 `DPOTrainer`，因此对于我们的初始基线，我们执行了以下作：

- Use the prompts and completions from Ai2's Tülu3 Preference Personas IF dataset to measure the improvements for instruction-following on IFEval with the `/no_think` reasoning mode.
  模式下 IFEval 上指令遵循的改进。

- Re-use the prompts from the above, but now generate "strong vs. weak" preference pairs with Qwen3-32B and Qwen3-0.6B. This gave us preference data for the `/think` reasoning mode.

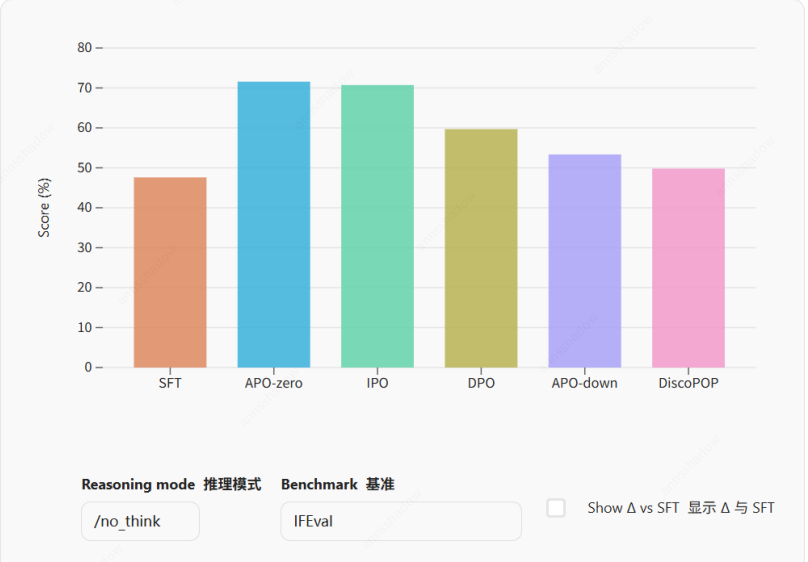
  重用上述提示，但现在生成 Qwen3-32B 和 Qwen3-0.6B 的“强与弱”首选项对。这为我们提供了 `/think` 推理模式的偏好数据。

- Train for 1 epoch and measure the *in-domain* improvements on IFEval, along with the *out-of-domain* impact on other evals like AIME25 which are directly correlated with instruction-following.

  训练 1 个 epoch 并测量 IFEval 的域内改进，以及对 AIME25 等其他评估的域外影响，这些评估与指令遵循直接相关。

As shown in the figure below, the in-domain improvements for both reasoning modes were significant: on IFEval, APO-zero improved over the SFT checkpoint by 15-20 percentage points!

如下图所示，两种推理模式的域内改进都是显着的：在 IFEval 上，APO-zero 比 SFT 检查点提高了 15-20 个百分点！



Since APO-zero also had the best overall out-of-domain performance, we settled on using it for the remainder of our ablations.

由于 APO-zero 也具有最佳的整体域外性能，因此我们决定在剩余的消融中使用它。

> 📍 **Preference optimisation works for reasoning**
>
> **偏好优化适用于推理**
>
> As our results above show, preference optimisation doesn't just make models more helpful or aligned, it teaches them to *reason better*. If you need a quick way to improve your reasoning model, try generating strong-vs-weak preferences and ablate different loss functions: you may find significant gains over vanilla DPO!
>
> 正如我们上面的结果所示，偏好优化不仅使模型更有帮助或更一致，还教会它们更好地推理。如果您需要一种快速的方法来改进您的推理模型，请尝试生成强与弱偏好并消除不同的损失函数：您可能会发现与普通 DPO 相比有显着收益！

WHICH HYPERPARAMETERS MATTER MOST FOR PREFERENCE OPTIMISATION?

哪些超参数对偏好优化最重要?

For preference optimisation, there are typically only three hyperparameters that impact the training dynamics:

对于偏好优化，通常只有三个超参数会影响训练动态：

- The learning rate, typically a factor of 10-100x smaller than the one used for SFT.

  学习率通常比用于 SFT 的学习率小 10-100 倍。

- The β parameter, which typically controls the size of the margin between preference pairs.

  β参数，通常控制首选项对之间的边距大小。

- The batch size. 批量大小。

Let's take a look at how these played out for SmolLM3, starting from the SFT checkpoint we trained over the whole of `smoltalk2`.

让我们看看这些在 SmolLM3 中是如何发挥作用的，从我们在整个 `smoltalk2`

Use small learning rates for best performance

使用较小的学习率以获得最佳性能

The first ablation we ran was to check the influence of the learning rate on model performance. We ran experiments to determine the influence of learning rates between ~200x smaller (1e-7) and ~2x smaller (1e-5) than the SFT learning rate (2e-5).

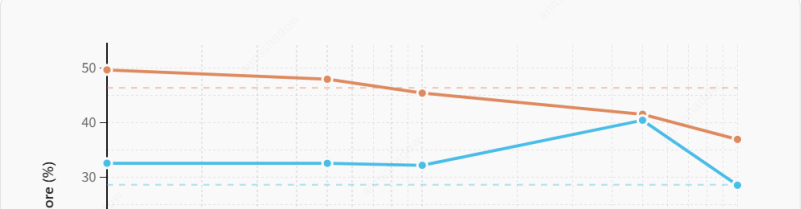
我们运行的第一个消融是检查学习率对模型性能的影响。我们进行了实验，以确定学习率比 SFT 学习率 （2e-5） 小 ~200 倍 （1e-7） 和 ~2 倍 （1e-5） 之间的学习率的影响。

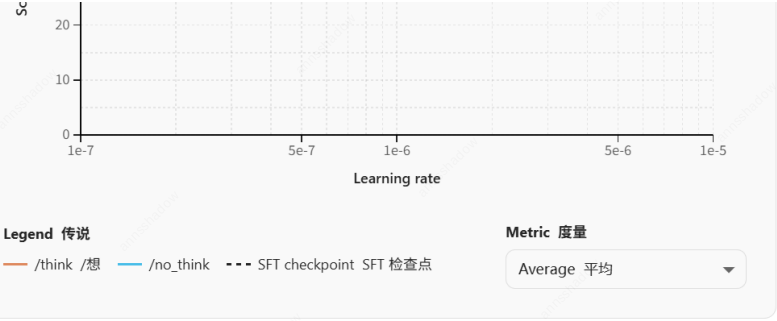
Previous projects like Zephyr 7B had taught us that the best learning rate for preference optimisation methods is around 10x smaller than the one used for SFT, and the ablations we ran for SmolLM3 confirmed this rule of thumb.

Zephyr 7B 等先前项目告诉我们，偏好优化方法的最佳学习率比用于 SFT 的方法小约 10 倍，我们为 SmolLM3 运行的消融证实了这一经验法则。

As shown in the figure below, learning rates ~10x smaller improve the performance of the SFT model in both reasoning modes, but all learning rates beyond that 10x limit result in worse performance for the extended thinking mode:

如下图所示，学习率 ~10 倍小，SFT 模型在两种推理模式下的性能都会提高，但所有超过该 10 倍限制的学习率都会导致扩展思维模式的性能更差：





The trend for the `/no_think` reasoning mode is more stable, with the best learning rate at 5e-6. This is mostly driven by a single benchmark (LiveCodeBench v4), so we opted for 1e-6 in our SmolLM3 runs.

`/no_think` 推理模式的趋势更加稳定，学习率最好，为 5e-6。这主要由单个基准测试（LiveCodeBench v4）驱动，因此我们在 SmolLM3 运行中选择了 1e-6。

Our recommendation for your training runs is to run scans of your learning rate at a range of 5x to 20x smaller than your SFT learning rate. It is highly likely that you will find your optimal performance within that range!

我们对训练运行的建议是以比 SFT 学习率小 5 倍到 20 倍的范围运行学习率扫描。您很可能会在该范围内找到最佳性能！

Tune your β 调整您的β

The experiments we ran for the ß parameter ranged from 0.01 to 0.99 to explore values that encourage different degrees of alignment to the reference model.

我们对ß参数进行的实验范围为 0.01 至 0.99，以探索鼓励与参考模型不同程度对齐的值。

As a reminder, lower values of beta encourage staying close to the reference model while higher values allow the model to match the preference data more closely. The model performance for β=0.

提醒一下，较低的 beta 值鼓励与参考模型保持接近，而较高的值则允许模型更接近偏好数据。β=0 的模型性能。

1 is the highest for both reasoning modes and improves compared to the metrics from the SFT checkpoint.

1 是两种推理模式的最高值，并且与 SFT 检查点的指标相比有所改善。

Using a low beta value hurts model performance and results in a worse model than the SFT checkpoint, while performance remains stable across multiple ß values without extended thinking.

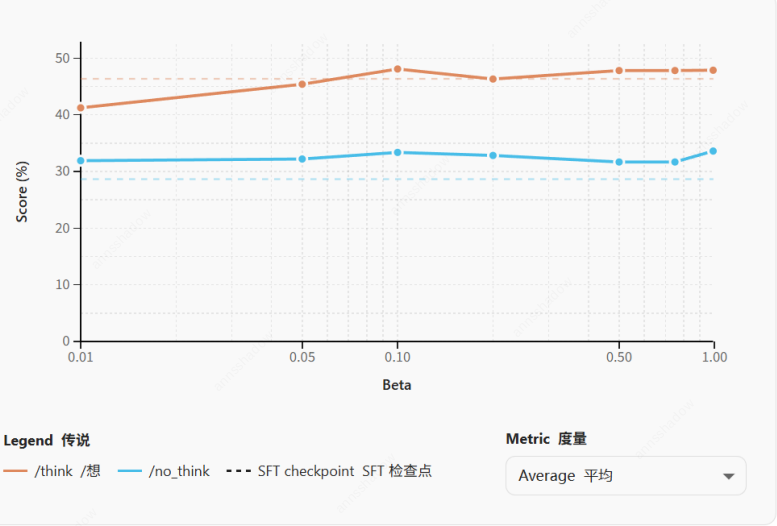
使用低 beta 值会损害模型性能，并导致模型比 SFT 检查点更差，而性能在多个 ß 值上保持稳定，无需扩展思考。

These results suggest that values greater than 0.1 are preferable for preference optimisation, and that aligning the model with the preference data is more beneficial than staying close to the reference model. However, we suggest exploring ß values in the range 0.01 and 0.5.

这些结果表明，大于 0.1 的值更适合偏好优化，并且将模型与偏好数据对齐比接近参考模型更有益。但是，我们建议探索 0.01 和 0.5 范围内的 ß 值。

Higher values may erase capabilities from the SFT checkpoint that we might not be capturing in the evals shown on the plot.

较高的值可能会从 SFT 检查点中擦除我们可能未在图上显示的评估中捕获的功能。



Scaling the preference data

缩放首选项数据

We also ran experiments to determine how dataset size influences results, testing values from 2k to 340k preference pairs. Across this range, performance remained stable.

我们还进行了实验以确定数据集大小如何影响结果，测试了 2k 到 340k 偏好对的值。在此范围内，性能保持稳定。

Performance drops in the extended thinking mode occur for datasets beyond 100k preference pairs, but the drop is not as pronounced as we saw with different learning rate values.

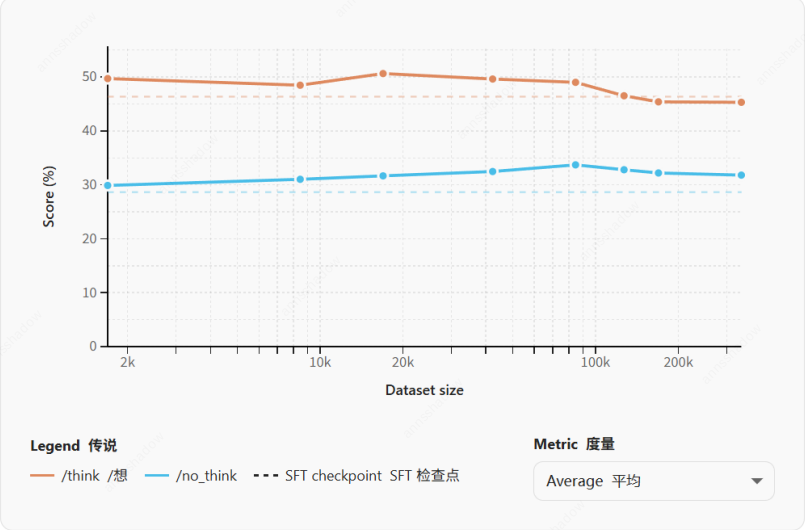
对于超过 100k 个偏好对的数据集，扩展思维模式下的性能下降会发生，但这种下降并不像我们在不同学习率值下看到的那么明显。

The dataset we used for the SmolLM3 training run was 169k preference pairs, but the results show that smaller datasets also show improvements over the SFT checkpoint.

我们用于 SmolLM3 训练运行的数据集是 169k 个偏好对，但结果表明，较小的数据集也显示出比 SFT 检查点的改进。

For future projects, we know we can experiment with smaller datasets during the iteration phase, as it is important to try multiple ideas and quickly identify the most promising configurations.

对于未来的项目，我们知道我们可以在迭代阶段试验较小的数据集，因为尝试多种想法并快速确定最有前途的配置非常重要。



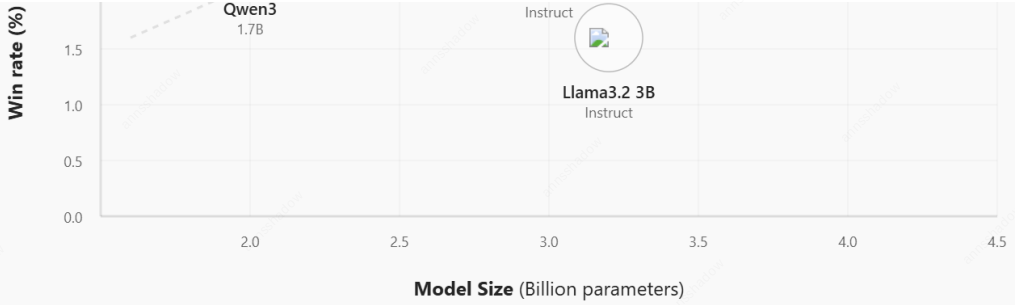
Bringing it all together 将所有内容整合在一起

Bringing all these threads together produced the final SmolLM3-3B model: best-in-class for its size and sat on the Pareto front with Qwen's own hybrid reasoning models.

将所有这些线索结合在一起，产生了最终的 SmolLM3-3B 模型：就其尺寸而言是一流的，并且与 Qwen 自己的混合推理模型一起处于帕累托前沿。

Instruct models without reasoning

无需推理即可指导模型



Not too shabby for a few weeks work!

几周的工作不会太寒酸！

RULES OF ENGAGEMENT 交战规则

To summarise our findings about preference optimisation that could be useful for your future projects:

总结一下我们关于偏好优化的发现，这些研究结果可能对您未来的项目有用：

- Don't be afraid to create your own preference data! With inference becoming "too cheap to meter", it's nowadays simple and cost-effective to generate LLM preferences from various inference providers.

  不要害怕创建自己的偏好数据！随着推理变得"太便宜而无法计量"，如今从各种推理提供商生成 LLM 偏好既简单又经济高效。

- Pick DPO as your initial baseline and iterate from there. We've found that depending on the type of preference data, other algorithms like ORPO, KTO, or APO can provide significant gains over DPO.

  选择 DPO 作为初始基线，然后从那里进行迭代。我们发现，根据偏好数据的类型，其他算法（如 ORPO、KTO 或 APO）可以提供比 DPO 显着的收益。

- Use a learning rate that's around 10x smaller than the one used for SFT.

  使用比 SFT 低约 10 倍的学习率。

- Scan over β, usually in the range 0.01 to 0.5

  扫描超过 β，通常在 0.01 到 0.5 的范围内

- Since most preference algorithms overfit after one epoch, partition your data and train iteratively for best performance.

  由于大多数偏好算法在一个纪元后会过度拟合，因此请对数据进行分区并迭代训练以获得最佳性能。

Preference optimisation is often the sweet spot between simplicity and performance, but it still inherits a key limitation: it's only as good as the offline preference data you can collect.

偏好优化通常是简单性和性能之间的最佳平衡点，但它仍然继承了一个关键限制：它的好坏取决于您可以收集的离线偏好数据。

At some point, static datasets run out of signal and you need methods that can generate fresh training feedback online as the model interacts with prompts and environment.

在某些时候，静态数据集会耗尽信号，您需要能够在模型与提示和环境交互时在线生成新训练反馈的方法。

That's where preference optimisation meets the broader family of *on-policy and RL-based methods.*

这就是偏好优化与更广泛的政策和基于 RL 的方法相遇的地方。

Going on-policy and beyond supervised labels
符合政策并超越监管标签

If you want your model to consistently solve math problems, generate executable code, or plan across multiple steps, you often need a **reward signal** rather than just "A is better than B".

如果您希望您的模型始终如一地解决数学问题、生成可执行代码或跨多个步骤进行计划，您通常需要一个奖励信号，而不仅仅是“A 比 B 更好”。

This is where RL starts to make sense. Instead of supervising the model with preferences, you let it interact with an environment (which could be a math verifier, a code executor, or even real user feedback), and learn directly from the outcomes. RL shines when:

这就是 RL 开始有意义的地方。您无需使用偏好来监督模型，而是让它与环境（可以是数学验证器、代码执行器，甚至是真实的用户反馈）交互，并直接从结果中学习。RL 在以下情况下大放异彩：

- **You can check correctness automatically,** e.g., unit tests, mathematical proofs, API calls, or have access to a high-quality verifier or reward model.

  您可以自动检查正确性，例如单元测试、数学证明、API 调用，或者访问高质量的验证器或奖励模型。

- **The task requires multi-step reasoning or planning** , where local preferences may not capture long-term success.

  这项任务需要多步骤推理或计划，其中当地的偏好可能无法获得长期成功。

- **You want to optimise for objectives beyond preference labels** , like passing unit tests for code or maximising some objective.

  您希望针对首选项标签以外的目标进行优化，例如通过代码的单元测试或最大化某些目标。

When it comes to LLMs, there are two main flavours of RL:

说到 LLM，RL 有两种主要风格：

- **Reinforcement Learning from Human Feedback (RLHF):** this is the approach that was popularised by OpenAI's InstructGPT paper (Ouyang et al., 2022) and the basis for gpt-3.5 and many modern LLMs. Here, human annotators compare model outputs (e.g. “A is better than B”) and a reward model is trained to predict those preferences. The policy is then fine-tuned with RL to maximise the learned reward.

  来自人类反馈的强化学习（RLHF）：这是 OpenAI 的 InstructGPT 论文（Ouyang et al.， 2022）推广的方法，也是 gpt-3.5 和许多现代 LLM 的基础。在这里，人类注释者比较模型输出（例如“A 比 B 更好”），并训练奖励模型来预测这些偏好。然后使用 RL 对策略进行微调，以最大化学习的奖励。

  Because the reward model only approximates human preferences, it can sometimes encourage reward hacking, where the policy emits an out-of-distribution sequence like “the the the the” which is given a spurious high reward and is then baked into the model via the RL loop.

  由于奖励模型仅近似于人类偏好，因此有时会鼓励奖励黑客攻击，即策略发出一个分布外序列，例如“the the the”，该序列被赋予虚假的高奖励，然后通过 RL 循环烘焙到模型中。

- **Reinforcement Learning with Verifiable Rewards (RLVR):** this is the approach that was popularised by DeepSeek-R1 and involves the use of verifiers that check whether a model's output meets some clearly defined correctness criteria (e.g. does the code compile and pass all tests, or is the mathematical answer correct?).

  具有可验证奖励的强化学习 （RLVR）：这是 DeepSeek-R1 推广的方法，涉及使用验证器来检查模型的输出是否满足一些明确定义的正确性标准（例如，代码是否编译并通过所有测试，或者数学答案是否正确？

  The policy is then fine-tuned with RL to produce more verifiably-correct outputs.

  然后使用 RL 对策略进行微调，以产生更可验证的正确输出。

Both RLHF and RLVR define *what* the model is being optimised for, but they don't tell us *how* that optimisation should be carried out. In practice, the efficiency and stability of RL-based training depends heavily on whether the learning algorithm is **on-policy** or **off-policy** .

RLHF 和 RLVR 都定义了模型的优化目标，但它们没有告诉我们应该如何进行优化。在实践中，基于 RL 的训练的效率和稳定性在很大程度上取决于学习算法是策略上的还是策略外的。

Methods such as GRPO typically fall into the category of on-policy optimisation algorithms, where the model (the policy) that generates the completions is the same as the one being optimised.

GRPO 等方法通常属于策略优化算法类别，其中生成完成的模型（策略）与被优化的模型相同。

While it is broadly the case that GRPO is an on-policy algorithm, there are a few caveats.

虽然 GRPO 是一种基于策略的算法，但也有一些注意事项。

First, to optimise the generation step, several batches of generations may be sampled and then k updates are made to the model, with the first batch being on-policy with the next few batches being slightly off-policy.

首先，为了优化生成步骤，可以对几批代进行采样，然后 k 对模型进行更新，第一批是符合策略的，接下来的几批是稍微偏离策略的。

To account for policy-lag between the model used for generation and the current

We mention here off-policy

model being optimised, importance sampling and clipping are used to re-weight the token probabilities and restrict the size of the updates.

为了考虑用于生成的模型与当前正在优化的模型之间的策略滞后，重要性采样和裁剪用于重新加权标记概率并限制更新的大小。

As autoregressive generation from LLMs is slow, many frameworks like verl and PipelineRL have added asynchronous generation of completions and "in-flight" updates of model weights to maximise training throughput. These approaches require more complex and careful implementation, but can achieve training speeds that are 4-5x higher than synchronous training methods.

由于从 LLM 生成自回归速度很慢，因此许多框架（如 verl 和 PipelineRL）添加了异步生成补全和模型权重的"动态"更新，以最大限度地提高训练吞吐量。这些方法需要更复杂和仔细的实施，但可以实现比同步训练方法高 4-5 倍的训练速度。

As we'll see later, these improvements in training efficiency are especially pronounced for reasoning models, which have long-tail token distributions.

正如我们稍后将看到的，这些训练效率的改进对于具有长尾代币分布的推理模型来说尤为明显。

For SmolLM3, we skipped RL altogether, mostly due to time constraints and having a model that was already best-in-class with offline preference optimisation.

对于 SmolLM3，我们完全跳过了 RL，主要是由于时间限制，并且模型已经是一流的离线偏好优化。

However, since the release, we have revisited the topic and will close out the post-training chapter by sharing some of our lessons from applying RLVR to hybrid

RL, but there are several truly off-policy RL algorithms, such as Q-learning, where the policy used for generate trajectories can be totally different to the policy being optimized.

我们在这里提到了非策略 RL，但有几种真正非策略的 RL 算法，例如 Q-learning，其中用于生成轨迹的策略可能与被优化的策略完全不同。

When GRPO is applied to LLMs, the policy used for generation can lag behind the one used for optimization, but typically there are less than 16 steps difference between the two.

当 GRPO 应用于 LLM 时，用于生成的策略可能会滞后于用于优化的策略，但通常两者之间的差异不到 16 步。

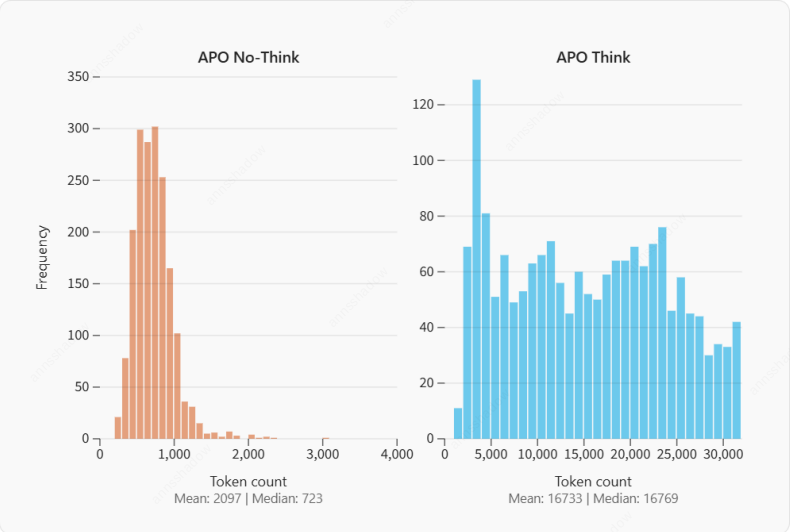
reasoning models.

然而，自发布以来，我们重新审视了这个主题，并将通过分享我们将 RLVR 应用于混合推理模型的一些经验教训来结束训练后章节。

APPLYING RLVR TO HYBRID REASONING MODELS
将 RLVR 应用于混合推理模型

Hybrid reasoning models pose additional complexity for RLVR because generation lengths vary considerably depending on the reasoning mode. For example, in the figure below, we plot the token length distributions on AIME25 for the final APO checkpoint from SmolLM3:

混合推理模型给 RLVR 带来了额外的复杂性，因为生成长度因推理模式而异。例如，在下图中，我们绘制了 AIME25 上 SmolLM3 最终 APO 检查点的令牌长度分布：



As you can see, the `/no_think` mode generates solutions with a median length of around 2k tokens, while the `/think` mode is much larger with 16k tokens and a fat-tailed distribution. Ideally, we would like to improve the overall performance of both modes with RLVR, without changing their respective length distributions too radically.

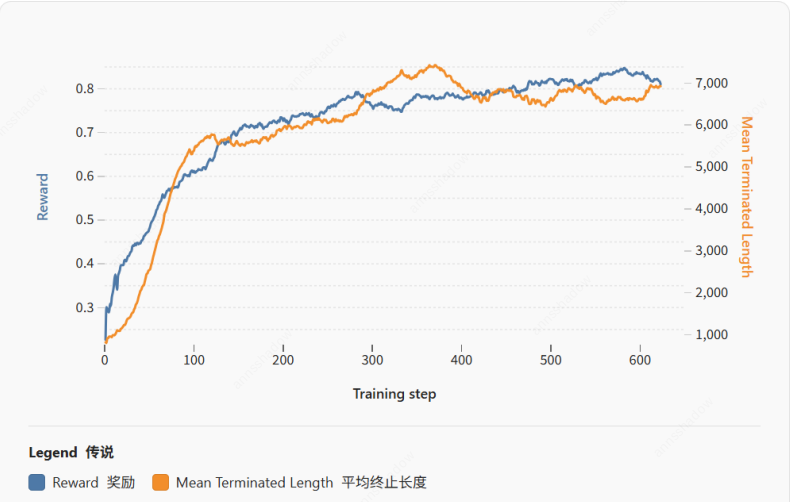
如您所见，该 `/no_think` 模式生成的解决方案的中位长度约为 2k 个令牌，而该 `/think` 模式则更大，有 16k 个令牌和肥尾分布。理想情况下，我们希望通过 RLVR 提高两种模式的整体性能，而不会从根本上改变它们各自的长度分布。

To explore this, we focused on optimising the `/no_think` mode first and took a subset of prompts from Big-Math, a dataset of over 250k math problems with verified answers.

为了探索这一点，我们首先专注于优化 `/no_think` 模式，并从 Big-Math 中获取了一部分提示，这是一个包含超过 250k 个数学问题的数据集，并具有经过验证的答案。

To our surprise, naively applying GRPO leads to a form of *reward hacking* : despite never being prompted to emit a long CoT, the model learns to exploit its base reasoning capabilities to increase the reward. As shown in the figure below, not only does the reward go up, but so too does the length of the completions:

令我们惊讶的是，天真地应用 GRPO 会导致一种形式的奖励黑客攻击：尽管从未被提示发出长 CoT，但该模型学会了利用其基本推理能力来增加奖励。如下图所示，不仅奖励会增加，完成的时长也会增加：



Naively applying RLVR to SmolLM3 achieves high reward and improved downstream performance, but comes at the cost of exploding completion lengths.

将 RLVR 天真地应用于 SmolLM3 可以获得高回报并提高下游性能，但代价是完成长度爆炸式增长。

In other words, RLVR with GRPO has converted the `/no_think` mode of our reasoning model into one which looks very similar to the `/think` mode!

换句话说，带有 GRPO 的 RLVR 已经将我们的推理模型的 `/no_think` 模式转换为看起来与 `/think` 模式非常相似的模式！

When we review the model's completions, we see that the model now generates a long CoT and even includes some of the cognitive behaviours (Gandhi et al., 2025) like "Wait, ..." that are associated with reasoning models. An example of this is shown below:

当我们回顾模型的完成时，我们看到该模型现在生成了一个长 CoT，甚至包括一些认知行为（Gandhi 等人，2025 年），例如"等等，.....”与推理模型相关联。下面显示了一个示例：

```
However, since the jogger and the train are moving in the same direction, the rel
\\( v_{\\text{rel}} = v_t - v_j = 12.7778 - 2.7778 = 10 \\) m/s

 **Wait,**  let me double-check the calculation:

 **Wait,**  46 km/hr to m/s:
46 * 1000 m / 3600 s = (46 * 10)/36 = 460/36 ≈ 12.7778 m/s (correct)
10 km/hr is:
10 * 1000 / 3600 = 10000 / 3600 ≈ 2.7778 m/s (correct)
Difference: 12.7778 - 2.7778 = 10 m/s (exact, since 46-10=36 km/hr → 36*(1000/366

The jogger is 340 meters ahead of the train's engine. When the train passes the j

 **Wait,**  actually, when the problem says the train takes 46 seconds to pass th

But first, the initial position: the jogger is 340 meters ahead of the engine. So

 **Wait**  the problem says: "A jogger running at 10 km/hr alongside a railway tr

Hmm, so the jogger is 340 m ahead of the engine along the track. Since they're mo
```

Mitigating reward hacking with overlong penalties

通过过长的惩罚来缓解奖励黑客攻击

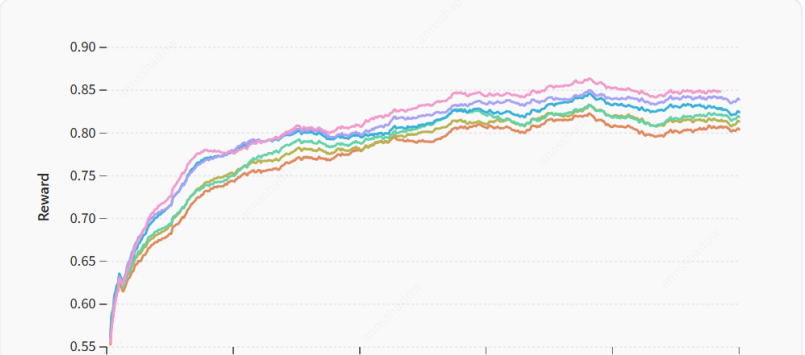
This issue can mitigated by including an *overlong completion penalty*, that penalises completions over a certain length. The penalty is parameterised by two arguments max completion length L_{max} and soft punishment cache L_{cache}. This penalty was one of the improvements proposed in the DAPO paper (Yu et al., 2025) and amounts to applying a reward function as follows:

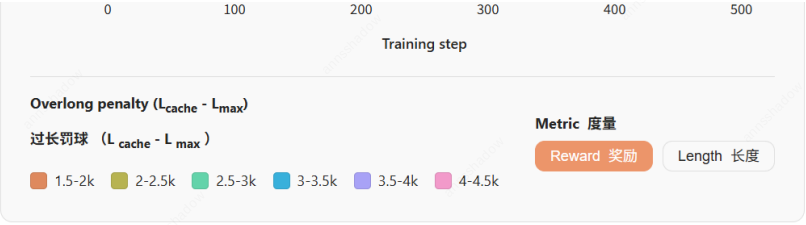
这个问题可以通过包括一个 过长完成惩罚，它惩罚了一定长度的完成。惩罚由两个参数参数化：max completion length L_{max} 和 soft punishment cache L_{cache}。这种惩罚是 DAPO 论文中提出的改进之一（Yu et al.， 2025），相当于应用奖励函数，如下所示：

$$R_{\text{length}}(y) = \begin{cases} 0, & |y| \leq L_{\max} - L_{\text{cache}} \\ \frac{(L_{\max} - L_{\text{cache}} - |y|)}{L_{\text{cache}}}, & L_{\max} - L_{\text{cache}} < |y| \leq L_{\max} \\ -1, & L_{\max} < |y| \end{cases}$$

Using this penalty, we can directly control the model's output distribution and measure the tradeoff between increasing response length and performance. An example is shown in the figure below, where we vary the overlong penalty from 1.5k to 4k in steps of 512 tokens:

利用这种惩罚，我们可以直接控制模型的输出分布，并衡量增加响应长度和性能之间的权衡。下图显示了一个示例，我们以 512 个代币为步长，将过长惩罚从 1.5k 变为 4k：



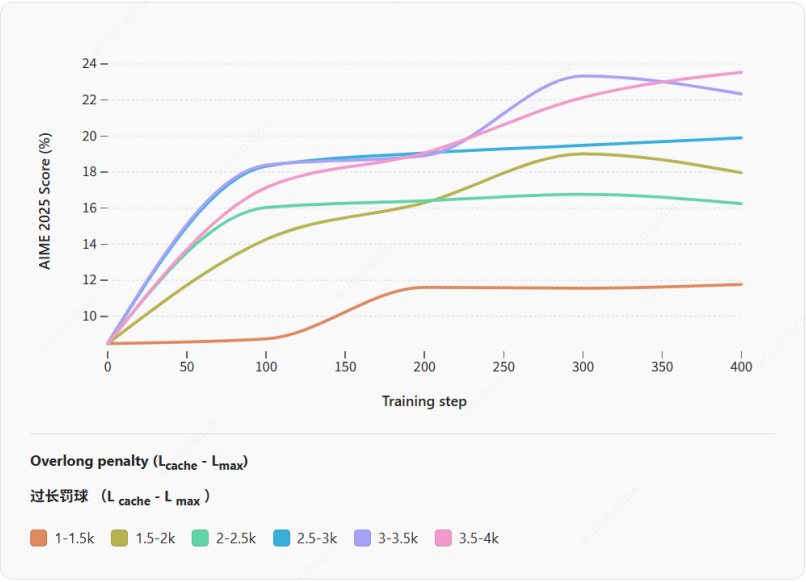


Applying an overlong penalty constrains the length of each rollout, while also reducing the average reward.

应用过长的惩罚会限制每次推出的长度，同时也会降低平均奖励。

The tradeoff between response length and performance is clearer when we examine the improvements on AIME25:

当我们检查 AIME25 的改进时，响应长度和性能之间的权衡更加清晰：



Downstream performance of Smollm3 with RLVR on AIME25.

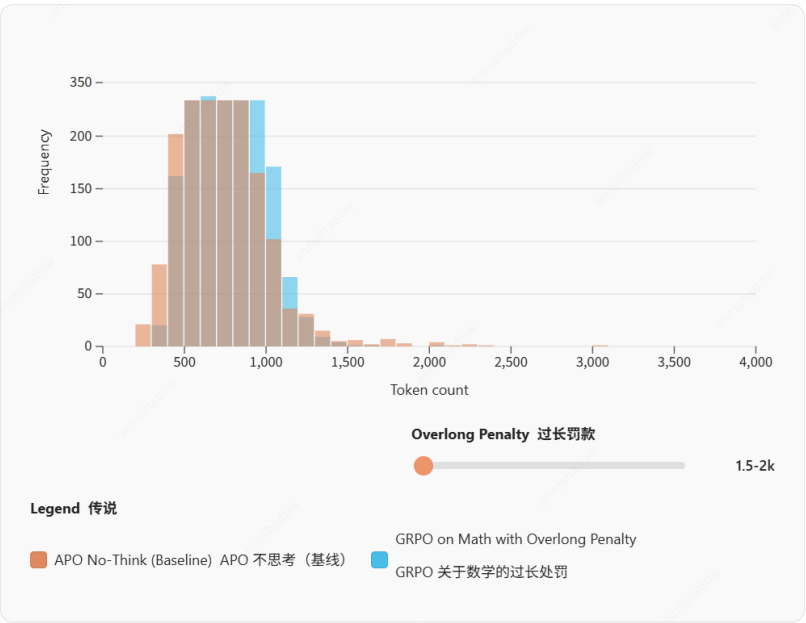
Smollm3 与 RLVR 在 AIME25 上的下游性能。

Now we can clearly see how the overlong penalty impacts downstream performance, with penalties in the range 2-4k producing significant improvements, while keeping the token distribution in check.

现在我们可以清楚地看到过长的惩罚如何影响下游性能，2-4k 范围内的惩罚产生了显着的改进，同时控制了代币分配。

As shown in the figure below, if we take the checkpoints from step 400, we can compare the output token distributions between the initial policy and final model across a range of different penalties:

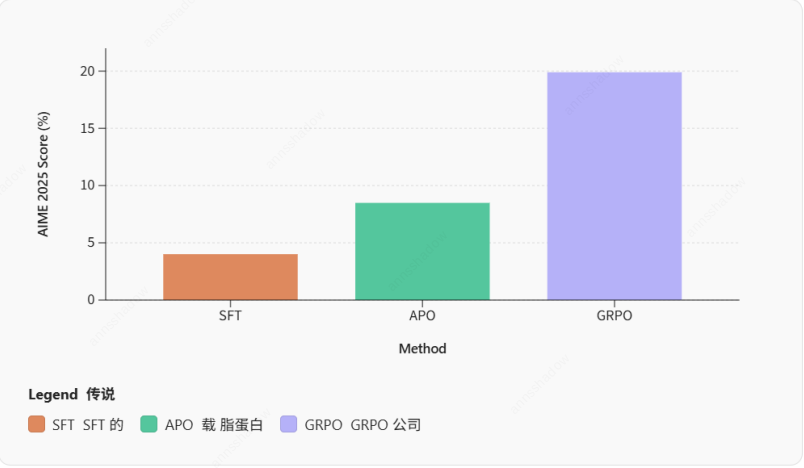
如下图所示，如果我们从步骤 400 中获取检查点，我们可以比较一系列不同惩罚的初始策略和最终模型之间的输出代币分布：



Bringing it all together 将所有内容整合在一起

We find that applying a length penalty in the range 2.5-3k gives the best tradeoff between performance and response length, with the figure below showing that GRPO nearly doubles the performance on AIME 2025 over offline methods like APO:

我们发现，在 2.5-3k 范围内应用长度惩罚可以在性能和响应长度之间进行最佳权衡，下图显示 GRPO 在 AIME 2025 上的性能几乎是 APO 等离线方法的两倍：



Now that we know how to improve performance in the `/no_think` reasoning mode, the next step in the RL training pipeline would be *joint training* of the model in both reasoning modes at once. However, we have found this to be quite a tough nut to crack because each mode requires it's own length penalty and the interplay has thus far produced unstable training.

现在我们知道如何提高 `/no_think` 推理模式下的性能，RL 训练管道的下一步将是同时在两种推理模式下联合训练模型。然而，我们发现这是一个很难破解的难题，因为每种模式都需要自己的长度惩罚，而且到目前为止，相互作用已经产生了不稳定的训练。

This highlights the main challenge with trying to apply RL on hybrid reasoning models, and we can see this reflected in a new trend from model developers like Qwen to release theinstruct and reasoning variants separately.

这凸显了尝试在混合推理模型上应用 RL 的主要挑战，我们可以看到这反映在像 Qwen 这样的模型开发人员分别发布指令和推理变体的新趋势中。

Our experiments show that RLVR can steer reasoning behaviour effectively, but only with careful reward shaping and stability mechanisms. Given this complexity, it's worth asking whether reinforcement learning is the only viable path forward.

我们的实验表明，RLVR 可以有效地引导推理行为，但前提是需要仔细的奖励塑造和稳定机制。鉴于这种复杂性，值得问的是强化学习是否是唯一可行的前进道路。

In fact, several lighter-weight, on-policy optimisation strategies have been proposed in recent literature, yet remain surprisingly under explored by the open-source community.

事实上，最近的文献中已经提出了几种更轻量级的、符合策略的优化策略，但令人惊讶的是，开源社区仍然没有充分探索。

Let's close out this chapter by taking a look at some of them.

让我们通过看看其中的一些来结束本章。

IS RL THE ONLY GAME IN TOWN?
RL 是镇上唯一的游戏吗?

Other approaches to on-policy learning extend preference optimisation and distillation into iterative loops that refresh the training signal as the model evolves:

其他策略学习方法将偏好优化和提炼扩展到迭代循环中，随着模型的发展刷新训练信号：

- **Online DPO:** rather than training once on a fixed preference dataset, the model continually samples new responses, collects fresh preference labels (from reward models or LLM graders), and updates itself. This keeps the optimisation *on-policy* al., 2024).

  在线 DPO：该模型不会在固定的偏好数据集上进行一次训练，而是不断采样新的响应，收集新的偏好标签（来自奖励模型或 LLM 评分器），并自行更新。这使优化保持在政策上，并减少了训练数据与模型当前行为之间的漂移（Guo et al.， 2024）。

- **On-policy distillation:** instead of preferences, the signal comes from a stronger teacher model. The student samples responses at every training step and the KL divergence between the student and teacher logits on these samples provides the

learning signal.

政策提炼：信号不是偏好，而是来自更强大的教师模式。学生在每个训练步骤中对响应进行采样，这些样本上学生和教师 logit 之间的 KL 差异提供了学习信号。

This allows the student to continuously absorb the teacher's capabilities, without needing explicit preference labels or verifiers(Agarwal et al., 2024).

这使得学生能够不断吸收教师的能力，而不需要明确的偏好标签或验证者（Agarwal 等人，2024 年）。

These methods blur the line between static preference optimisation and full RL: you still get the benefits of adapting to the model's current distribution, but without the full complexity of designing and stabilising a reinforcement learning loop.

这些方法模糊了静态偏好优化和完全 RL 之间的界限：您仍然可以获得适应模型当前分布的好处，但无需设计和稳定强化学习循环的全部复杂性。

WHICH METHOD DO I PICK?
我该选择哪种方法?

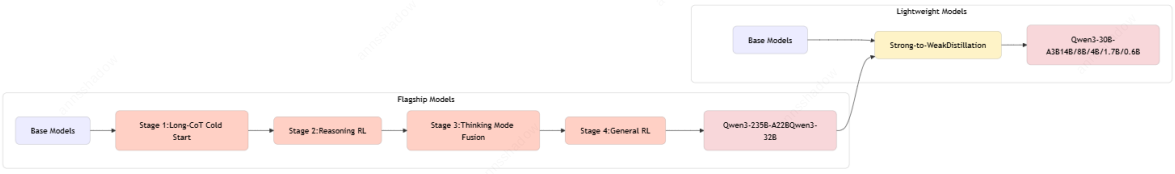
Although there are a gazillion research papers about which on-policy method is "best", in practice the decision depends on a few factors shown in the table below:

尽管有无数关于哪种政策方法是"最佳"的研究论文，但实际上，决定取决于下表所示的几个因素：

Algorithm 算法	
Online DPO 在线 DPO	You can get preference labels cheaply. Best for aligning behaviour with evolving distributions. 您可以便宜地获得偏好标签。最适合使行为与不断发展的分布保持一致。
On-policy distillation 政策蒸馏	You have access to a stronger teacher model and want to transfer capabilities efficiently. 您可以获得更强大的教师模型，并希望有效地转移能力。
Reinforcement learning 强化学习	Best when you have verifiable rewards or tasks requiring multi-step reasoning/planning. Can be u 当您有可验证的奖励或需要多步骤推理/计划的任务时，最好。可以与奖励模型一起使用，但存在奖励黑客

In the open-source ecosystem, reinforcement learning methods like GRPO and REINFORCE tend to be the most widely used, although the Qwen3 tech report (A. Yang, Li, et al., 2025) highlighted the use of on-policy distillation to train the models under 32B parameters:

在开源生态系统中，GRPO 和 REINFORCE 等强化学习方法往往是最广泛使用的，尽管 Qwen3 技术报告（A. Yang， Li， et al.， 2025）强调了使用策略蒸馏来训练 32B 参数下的模型：



One interesting property of on-policy distillation with small models is that it typically outperforms RL-based methods at a fraction of the compute cost.

使用小型模型进行策略蒸馏的一个有趣特性是，它通常以计算成本的一小部分优于基于 RL 的方法。

This is because instead of generating multiple rollouts per prompt, we only sample one, which is then graded by the teacher in a single forward-backward pass.

这是因为我们不是为每个提示生成多个推出，而是只对一个进行采样，然后由教师在一次前向-后退传递中对其进行评分。

As the Qwen3 tech report shows, the gains over GRPO can be significant:

正如 Qwen3 技术报告所示，相对于 GRPO 的收益可能是显着的：

Method 方法	AIME'24 AIME'24	AIME'25 AIME'25	MATH500	LiveCodeBench v5	MMLU -Redux
Off-policy Distillation 政策外蒸馏	55.0	42.8	92.4	42.0	86.4
+ Reinforcement Learning + 强化学习	67.6	55.5	94.8	52.9	86.9
+ On-policy Distillation + 政策蒸馏	**74.4**	**65.5**	**97.0**	**60.3**	**88.3**

More recently, Thinking Machines have shown that on-policy distillation is also effective at mitigating *catastrophic forgetting*, where a post-trained model is further trained on a new domain and it's prior performance regresses.

最近，Thinking Machines 表明，策略蒸馏也可以有效减轻灾难性遗忘，即后训练模型在新领域进一步训练，其先前的性能会倒退。

In the table below, they show that although the chat performance of Qwen3-8b (IFEval) tanks when it's fine-tuned on internal data, the behaviour can be restored with cheap distillation:

在下表中，他们表明，尽管 Qwen3-8b （IFEval） 的聊天性能在对内部数据进行微调时会有所下降，但可以通过廉价的蒸馏来恢复行为：

Model	Internal QA Eval (Knowledge)	IF-eval (Chat)
Qwen3-8B	18%	85%
+ midtrain (100%)	43%	45%
+ midtrain (70%)	36%	79%
+ midtrain (70%) + distill	41%	83%

We ourselves are quite excited by on-policy distillation as there's a huge diversity of capable, open-weight LLMs that can be distilled into smaller, task-specific models.

我们自己对政策蒸馏感到非常兴奋，因为有各种各样的有能力的开放权重法学硕士可以提炼成更小的、特定于任务的模型。

However, one weakness with all on-policy distillation methods is that the teacher and student must share the same tokenizer.

然而，所有策略蒸馏方法的一个弱点是教师和学生必须共享相同的分词器。

Distillation (GOLD), which allows any teacher to be distilled into any student.

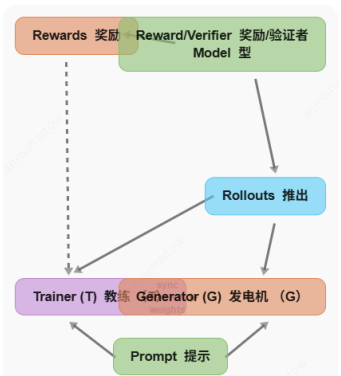
为了解决这个问题，我们开发了一种名为 General On-Policy Logit Distillation （GOLD） 的新方法，它允许将任何教师提炼成任何学生。

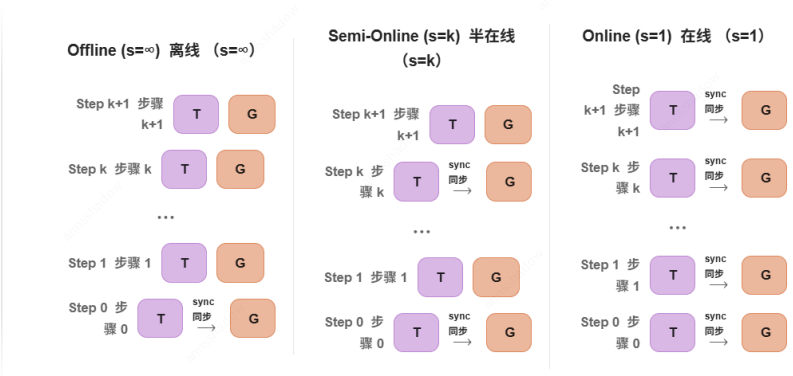
We recommend checking out ourtechnical write-up if you're interested in these topics.

如果您对这些主题感兴趣，我们建议您查看我们的技术文章。

Similarly, researchers at FAIR have compared the effect of being fully off-policy to on-policy for DPO and shown that it's possible to match the performance of GRPO using far less compute (Lanchantin et al., 2025):

同样，FAIR 的研究人员比较了 DPO 完全脱离政策与符合政策的影响，并表明可以使用更少的计算来匹配 GRPO 的性能（Lanchantin 等人，2025 年）：





As shown in their paper, online DPO works well for math tasks and even the semi-on-policy variant achieves comparable performance despite being many steps off-policy:

正如他们的论文所示，在线 DPO 适用于数学任务，即使是半策略变体，尽管许多步偏离策略，但也能实现相当的性能：

Training method 培训方法	Math500 数学 500	NuminaMath 努米纳数学	AMC23
Seed (Llama-3.1-8B-Instruct) 种子（Llama-3.1-8B-Instruct）	47.4	33.9	23.7
Offline DPO (s = ∞) 脱机 DPO （s = ∞）	53.7	36.4	28.8
Semi-online DPO (s = 100) 半在线 DPO （s = 100）	**58.9**	39.3	**35.1**
Semi-online DPO (s = 10) 半在线 DPO （s = 10）	57.2	39.4	31.4
Online DPO (s = 1) 在线 DPO （s = 1）	58.7	**39.6**	32.9
GRPO GRPO 公司	58.1	38.8	33.6

Overall, we feel that there still remains much to be done with both scaling RL effectively (Khatri et al., 2025) and exploring other methods for computational efficiency. Exciting times indeed!

总体而言，我们认为在有效扩展 RL（Khatri 等人，2025 年）和探索其他计算效率方法方面仍有很多工作要做。确实是激动人心的时刻！

Wrapping up post-training
培训后总结

If you've made it this far, congrats: you now have all the core ingredients needed for success with post-training. You're now ready to run many experiments and test different algorithms to get SOTA results.

如果您已经走到了这一步，那么恭喜您：您现在已经具备了成功完成培训后所需的所有核心要素。现在，您已准备好运行许多实验并测试不同的算法以获得 SOTA 结果。

But as you've probably realised, knowing how to train great models is only half the story. To actually bring those models to life, you need the right infrastructure. Let's finish this opus with the unsung hero of LLM training.

但您可能已经意识到，了解如何训练出色的模型只是故事的一半。要真正将这些模型变为现实，您需要正确的基础设施。让我们以法学硕士培训的无名英雄来结束这部作品。

Infrastructure - the unsung hero
基础设施 - 无名英雄

Now that you know everything we know about model creation and training, let's address the critical yet *underrated* component that can make or break your project (and your bank account): infrastructure.

现在您已经了解了我们所知道的有关模型创建和训练的所有信息，让我们来讨论一个关键但被低估的组件，它可能会成就或破坏您的项目（和您的银行账户）：基础设施。

Whether you focus on frameworks, architecture, or data curation, understanding infrastructure basics helps identify training bottlenecks, optimise parallelism strategies, and debug throughput issues. (At the minimum, it improves communication with infrastructure teams 😊).

无论您专注于框架、架构还是数据管理，了解基础设施基础知识都有助于识别训练瓶颈、优化并行度策略和调试吞吐量问题。（至少，它改善了与基础设施团队😊的沟通）。

Most people training models care deeply about architecture and data, yet very few understand the infrastructure details.

大多数训练模型的人都非常关心架构和数据，但很少有人了解基础设施的细节。

Infrastructure expertise typically lives with framework developers and cluster engineers, and gets treated by the rest as a solved problem: rent some GPUs, install PyTorch, and you're good to go.

基础设施专业知识通常与框架开发人员和集群工程师一起存在，并被其他人视为已解决的问题：租用一些 GPU，安装 PyTorch，然后就可以开始了。

We trained SmolLM3 on 384 H100s for nearly a month, processing a total of 11 trillion tokens… and this was not a smooth ride!

我们在 384 个 H100 上训练了 SmolLM3 近一个月，总共处理了 11 万亿个代币……这并不是一帆风顺的旅程！

During that time, we dealt with node failures, storage issues and run restarts (see thetraining marathon section). You need to have good contingency plans and strategies to prepare for these issues, and keep training smooth and low-maintenance.

在此期间，我们处理了节点故障、存储问题和运行重启（参见训练马拉松部分）。您需要制定良好的应急计划和策略来为这些问题做好准备，并保持培训顺利且维护成本低。

This chapter aims to bridge that knowledge gap. Think of it as a practical guide to the hardware layer, focused on the questions that matter for training. (Note: Each subsection starts with a TL;DR so you can choose your depth level.)

本章旨在弥合这一知识差距。将其视为硬件层的实用指南，重点关注对培训很重要的问题。（注意：每个小节都以 TL;DR 开头，以便您可以选择深度级别。

本章旨在弥合这一知识差距。将其视为硬件层的实用指南，重点关注对培训很重要的问题。
（注意：每个小节都以 TL 开头;DR，以便您可以选择深度级别。

The first two sections tackle the fundamentals of how hardware works: what does a GPU actually consist of? How does memory hierarchy work? How do CPUs and GPUs communicate?

前两节讨论硬件工作原理的基本原理：GPU 实际上由什么组成？内存层次结构如何工作？CPU 和 GPU 如何通信？

We'll also go over you what to consider when acquiring GPUs and how to test them before committing to long training runs. Most importantly, we'll show you at each step how to measure and diagnose these systems yourself.

我们还将介绍获取 GPU 时应考虑的事项，以及如何在进行长时间训练运行之前对其进行测试。最重要的是，我们将在每一步向您展示如何自己测量和诊断这些系统。

The next sections are then more applied, and we'll see how to make your infra resilient to failure, and how to maximally optimize your training throughput.

接下来的部分将得到更多应用，我们将了解如何使基础设施能够抵御故障，以及如何最大限度地优化训练吞吐量。

The name of the game of this chapter is to find and fix the bottlenecks!

本章的游戏名称就是找到并修复瓶颈！

Think of this as building your intuition for why certain design decisions matter.

可以将其视为建立对某些设计决策为何重要的直觉。

When you understand that your model's activations need to flow through multiple levels of cache, each with different bandwidth and latency characteristics, you'll naturally start thinking about how to structure your training to minimise data movement.

当您了解模型的激活需要流经多个级别的缓存，每个级别具有不同的带宽和延迟特征时，您自然会开始考虑如何构建训练以最大限度地减少数据移动。

When you see that inter-node communication is orders of magnitude slower than intra-node, you'll understand why parallelism strategies matter so much.

当您看到节点间通信比节点内通信慢几个数量级时，您就会明白为什么并行性策略如此重要。

Let's start by cracking open a GPU and seeing what's inside.

让我们首先打开 GPU 并查看里面有什么。

Inside a GPU: Internal Architecture
GPU 内部：内部架构

A GPU is fundamentally a massively parallel processor optimised for throughput over latency. Unlike CPUs, which excel at executing a few complex instruction streams quickly, GPUs achieve performance by executing thousands of simple operations

simultaneously.

GPU 从根本上说是一种大规模并行处理器，针对吞吐量和延迟进行了优化。与擅长快速执行一些复杂指令流的 CPU 不同，GPU 通过同时执行数千个简单作来实现性能。

The key to understanding GPU performance lies in recognising that it's not just about raw compute power, it's about the interplay between computation and data movement.

了解 GPU 性能的关键在于认识到它不仅与原始计算能力有关，还与计算和数据移动之间的相互作用有关。

A GPU can have teraflops of theoretical compute, but if data can't reach the compute units fast enough, that potential goes unused. This is why we need to understand both the memory hierarchy (how data moves) and the compute pipelines (how work gets done).

GPU 可以具有 teraflops 的理论计算，但如果数据无法足够快地到达计算单元，则该潜力将得不到利用。这就是为什么我们需要了解内存层次结构（数据如何移动）和计算管道（如何完成工作）。

At the highest level, a GPU therefore performs two essential tasks:

因此，在最高级别，GPU 执行两项基本任务：

1. **Move and store data** (the memory system)

   移动和存储数据（内存系统）

2. **Do useful work with the data** (the compute pipelines)

   对数据（计算管道）执行有用的工作

COMPUTE UNITS AND FLOPS
计算单元和 FLOPS

> **TL;DR:** GPUs measure performance in FLOPs (floating-point operations per second). Modern GPUs like the H100 deliver dramatically higher throughput at lower precision: 990 TFLOPs at BF16 vs 67 TFLOPs at FP32.
> 提供更高的吞吐量：BF16 为 990 TFLOP，FP32 为 67 TFLOP。
>
> However, real-world performance is 70-77% of theoretical peaks due to memory bottlenecks. State-of-the-art training achieves 20-41% end-to-end efficiency, also known as model flops utilization (MFU). Use realistic numbers, not marketing specs, when planning training runs.
>
> 然而，由于内存瓶颈，实际性能是理论峰值的 70-77%。最先进的训练可实现 20-41% 的端到端效率，也称为模型失败利用率 （MFU）。在规划训练运行时，使用现实的数字，而不是营销规范。

GPU compute performance is measured in **FLOPs** (floating-point operations per second). A FLOP is a single arithmetic operation, typically a floating numbers addition like `a + b`, and modern GPUs can execute trillions of these per second (TFLOPs).

GPU 计算性能以 FLOP（每秒浮点运算数）为单位。FLOP 是一种单一的算术运算，通常是像这样的 `a + b` 浮点数字加法，现代 GPU 每秒可以执行数万亿次 （TFLOP）。

The fundamental building blocks of GPU compute are **Streaming Multiprocessors (SMs)** , independent processing units that execute instructions in parallel. Each SM contains two types of **cores** : **CUDA cores** for standard floating-point operations, and specialized **Tensor Cores** optimized for matrix multiplication, the workhorse operation in deep learning (critical for transformer performance).

GPU 计算的基本构建块是 流式多处理器 （SM） ，并行执行指令的独立处理单元。每个 SM 包含两种类型的内核：用于标准浮点运算的 CUDA 内核，以及针对矩阵乘法优化的专用 Tensor 内核，矩阵乘法是深度学习中的主力作（对 Transformer 性能至关重要）。

Modern GPUs organize hundreds of these SMs across the chip! For example, the H100 SXM5 version (which is the GPU we're using on our cluster) contains 132 SMs. Each SM operates independently, executing groups of 32 threads called **warps** in lockstep. To help there, the SMs rely on another component, the **warp schedulers:** by balancing instructions to different warps, they enable the SM to "hide latency" by switching between warps when one is held up. This **SIMT** (Single Instruction, Multiple Thread) execution model means all threads in a warp execute the same instruction simultaneously on different data.

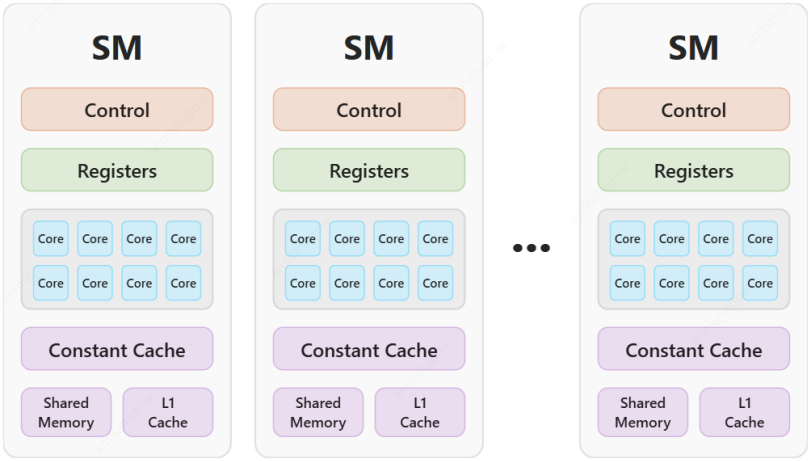
现代 GPU 在芯片上组织了数百个这样的 SM！例如，H100 SXM5 版本（这是我们在集群上使用的 GPU）包含 132 个 SM。每个 SM 独立运行，执行 32 个线程组，称为 warps 锁步。为了提供帮助，SM 依赖于另一个组件，即扭曲调度程序：通过平衡对不同扭曲的指令，它们使

Warps are named in reference to weaving, "the first parallel thread technology", according to Lindholm et al., 2008. The equivalent of warps in other GPU programming models include subgroups in WebGPU, waves in DirectX, and simdgroups in Metal.

根据 Lindholm 等人，2008 年的说法，经纱是针对编织的命

SM 能够在扭曲被保留时在扭曲之间切换来“隐藏延迟”。这种 SIMT（单指令、多线程）执行模型意味着扭曲中的所有线程同时对不同的数据执行相同的指令。

名，这是“第一种平行线技术”。其他 GPU 编程模型中的扭曲等效物包括 WebGPU 中的子组、DirectX 中的波形和 Metal 中的 simdgroup。



Multiple SMs within a single GPU - Source

单个 GPU 中的多个 SM - 来源

With hundreds of SMs each executing multiple warps concurrently, a single GPU can run tens of thousands of threads simultaneously. This massive parallelism is what enables GPUs to excel at the matrix operations that dominate deep learning workloads!

由于数百个 SM 同时执行多个 warp，单个 GPU 可以同时运行数万个线程。这种大规模的并行性使 GPU 能够在主导深度学习工作负载的矩阵运算中表现出色！

Precision matters significantly when discussing FLOPs. Tensor Cores can operate at different precisions (FP64, FP32, FP16/BF16, FP8, FP4 - see here for a reminder on floating point numbers). The achievable throughput therefore varies dramatically, often by orders of magnitude, depending on the data type.

在讨论 FLOP 时，精度非常重要。张量核心可以以不同的精度运行（FP64、FP32、FP16/BF16、FP8、FP4 - 有关浮点数的提醒，请参阅此处）。因此，可实现的吞吐量差异很大，通常有数量级，具体取决于数据类型。

If you want to learn more about our experience with FP8 mixed precision training check out the Ultra Scale Playbook.

如果您想了解更多关于我们在 FP8 混合精准训练方面的经验，请查看 Ultra Scale Playbook。

Lower precision formats enable higher throughput because they require less data movement and can pack more operations into the same silicon area, but were formerly avoided because of training instabilities.

较低精度的格式可实现更高的吞吐量，因为它们需要更少的数据移动，并且可以将更多作打包到同一硅区域中，但以前由于训练不稳定性而被避免。

However, nowadays, both training and inference are increasingly pushed toward lower precision, reaching FP8 and FP4, thanks to a range of new techniques.

然而，如今，由于一系列新技术，训练和推理都越来越接近更低的精度，达到 FP8 和 FP4。

然而，如今，由于一系列新技术，训练和推理都越来越接近更低的精度，达到 FP8 和 FP4。

The table below shows theoretical peak performance across different NVIDIA GPU generations and precision:

下表显示了不同 NVIDIA GPU 代和精度的理论峰值性能：

Precision\GPU Type 精度\GPU 类型	A100 A100 型	H100 H100 型	H200 H200 型
FP64 FP64 型	9.7	34	34
FP32	19.5	67	67
FP16/BF16	312	990	990
FP8	-	3960	3960
FP4	-	-	-

Table showing theoretical TFLOPs depending on precision and GPU generation. Source: Nvidia, SemiAnalysis

该表显示了取决于精度和 GPU 生成的理论 TFLOP。资料来源：英伟达，SemiAnalysis

The dramatic increase in throughput at a lower precision isn't just about raw speed, it reflects a fundamental shift in how we think about numerical computation. FP8 and FP4 enable models to perform more operations per **watt** and per **second** , making them essential for both training and inference at scale. The H100's 3960 TFLOPs at FP8 represents a 4x improvement over FP16/BF16, while the B200's 10,000 TFLOPs at FP4

pushes this even further.

在较低精度下吞吐量的急剧增加不仅仅是原始速度，它还反映了我们对数值计算的看法的根本转变。FP8 和 FP4 使模型能够每瓦和每秒执行更多作，这使得它们对于大规模训练和推理至关重要。H100 在 FP3960 时的 8 TFLOP 比 FP4/BF16 提高了 16 倍，而 B200 在 FP10,000 时的 4 TFLOP 进一步推动了这一点。

Understanding the numbers : These theoretical peak FLOPs represent the *maximum computational throughput achievable under ideal conditions* , when all compute units are fully utilized and data is readily available. In practice, actual performance depends heavily on how well your workload can keep the compute units fed with data and whether your operations can be efficiently mapped to the available hardware.

理解数字：这些理论峰值 FLOP 表示在理想条件下可实现的最大计算吞吐量，当所有计算单元都得到充分利用并且数据随时可用时。在实践中，实际性能在很大程度上取决于工作负载为计算单元提供数据的能力，以及您的作是否可以有效地映射到可用硬件。

For SmolLM3, we were going to train on NVIDIA H100 80GB HBM3 GPUs, so we first performance. For this, we used the SemiAnalysis GEMM benchmark: it tests throughput on real-world matrix multiplication shapes from Meta's Llama 70B training.

对于 SmolLM3，我们将在 NVIDIA H100 80GB HBM3 GPU 上进行训练，因此我们首先想根据实际性能测试 H100 的理论 TFLOP 规格。为此，我们使用了 SemiAnalysis GEMM 基准测试：它测试来自 Meta 的 Llama 70B 训练的真实世界矩阵乘法形状的吞吐量。

Shape (M, N, K) 形状（M、N、K）	FP64 torch.matmul FP64 火炬.matmul	FP32 torch.matmul FP32 割炬.matmul	FP16 torch.
(16384, 8192, 1280)	51.5 TFLOPS	364.5 TFLOPS 364.5 千叶百分秒	686.5 TFLOI
(16384, 1024, 8192)	56.1 TFLOPS	396.1 TFLOPS	720.0 TFLOI
(16384, 8192, 7168)	49.5 TFLOPS	356.5 TFLOPS 356.5 千叶百分点	727.1 TFLOI
(16384, 3584, 8192)	51.0 TFLOPS	373.3 TFLOPS	732.2 TFLOI
(8192, 8192, 8192)	51.4 TFLOPS	372.7 TFLOPS	724.9 TFLOI

Table showing achieved TFLOPs on H100 80GB depending on precision and matrix shape from the Llama 70B training workload

表格显示了 H100 80GB 上实现的 TFLOPs，具体取决于 Llama 70B 训练工作负载的精度和矩阵形状

Validating theoretical performance : Our experiments revealed the gap between theoretical peaks and achievable performance.

验证理论性能：我们的实验揭示了理论峰值与可实现性能之间的差距。

For **FP64 Tensor Core** operations, we achieved 49-56 TFLOPs, representing 74-84% of the theoretical peak (67 TFLOPs). For **TF32** (TensorFloat-32, which PyTorch uses by default for FP32 tensors on Tensor Cores), we achieved 356-396 TFLOPs, representing 72-80% of the theoretical peak (~495 TFLOPs dense).

对于 FP64 Tensor Core 作，我们实现了 49-56 TFLOPs，占理论峰值（67 TFLOPs）的 74-84%。对于 TF32（TensorFloat-32，PyTorch 默认用于 Tensor Core 上的 FP32 张量），我们实现了 356-396 TFLOP，占理论峰值（~495 TFLOPs 密集）的 72-80%。

While these show excellent hardware utilization, these precisions are rarely used in modern deep learning training: FP64 due to its computational cost, and TF32 because lower precisions like BF16 and FP8 offer better performance.

虽然这些显示出出色的硬件利用率，但这些精度很少用于现代深度学习训练：FP64 因其计算成本而使用，而 TF32 则因为 BF16 和 FP8 等较低精度提供更好的性能。

lower precisions like BF16 and FP8 offer better performance.

虽然这些显示出出色的硬件利用率，但这些精度很少用于现代深度学习训练：FP64 因其计算成本而使用，而 TF32 则因为 BF16 和 FP8 等较低精度提供更好的性能。

NVIDIA specs often list sparse performance (989 TFLOPs for TF32) which assumes 2:4 structured sparsity patterns. Dense operations, which our benchmark tests, achieve roughly half the sparse peak (~495 TFLOPs).

NVIDIA 规范通常列出稀疏性能（TF989 为 32 TFLOP），它假设 2：4 结构化稀疏模式。我们的基准测试的密集作达到了稀疏峰值的大约一半（~495

（TF989 为 32 TFLOP），它假设 2：4 结构化稀疏模式。我们的基准测试的密集作达到了稀疏峰值的大约一半（~495 TFLOP）。

For **BF16** operations, we consistently achieved 714-758 TFLOPs across different matrix shapes, approximately 72-77% of the H100's theoretical 990 TFLOPs peak. This is, in practice, an excellent utilisation rate for a real-world workload!

对于 BF16 作，我们在不同的基质形状上始终实现了 714-758 TFLOPs，大约是 H100 理论 990 TFLOPs 峰值的 72-77%。在实践中，对于实际工作负载来说，这是一个极好的利用率！

Model FLOPs Utilization (MFU)

模型 FLOP 利用率 （MFU）

While kernel benchmarks measure raw TFLOPS, end-to-end training efficiency is captured by **Model FLOPs Utilization (MFU)**: the ratio of useful model computation to theoretical peak hardware performance.

虽然内核基准测试测量原始 TFLOPS，但端到端训练效率由模型 FLOPs 利用率 （MFU） 捕获：有用模型计算与理论峰值硬件性能的比率。

Our BF16 matmul benchmarks showed we achieved 72-77% of the H100's theoretical peak. This represents the upper bound for what's achievable at the kernel level for our setup.

我们的 BF16 matmul 基准测试显示，我们达到了 H72 理论峰值的 77-100%。这代表了我们的设置在内核级别可实现的上限。

End-to-end training MFU will necessarily be lower due to more complex non-matmul operations, communication overhead, and other auxiliary computations.

由于更复杂的非 matmul 作、通信开销和其他辅助计算，端到端训练 MFU 必然会降低。

State-of-the-art MFU in training: Meta achieved 38-41% when training Llama 3 405B, while DeepSeek-v3 reached ~20-30% on GPUs with tighter communication bottlenecks related to the MoE architecture. For SmolLM3, we achieved ~30% MFU as we'll see later.

训练中最先进的 MFU：Meta 在训练 Llama 3 405B 时达到了 38-41%，而 DeepSeek-v3 在与 MoE 架构相关的通信瓶颈更严格的 GPU 上达到了 ~20-30%。对于 SmolLM3，我们实现了 ~30% 的 MFU，我们将在后面看到。

Much of the gap comes from inter-node communication overhead in distributed

training. Given our kernel-level ceiling of ~77%, these end-to-end numbers represent roughly 50-55% efficiency relative to achievable matmul performance.

大部分差距来自分布式训练中的节点间通信开销。鉴于我们的内核级上限为 ~77%，这些端到端数字相对于可实现的 matmul 性能代表大约 50-55% 的效率。

Inference workloads can reach higher MFU >70%, closer to raw matmul performance, though published results from production deployments are scarce.

推理工作负载可以达到更高的 MFU >70%，更接近原始 matmul 性能，尽管生产部署的已发布结果很少。

The **FP8** results are more nuanced. Let's look at our results on 3 different matrix multiplication methods/kernels.

A kernel is the unit of CUDA code.

内核是 CUDA 代码的单位。

FP8 的结果更加微妙。让我们看看我们在 3 种不同矩阵乘法/内核上的结果。

Using PyTorch's `torch._scaled_mm` kernel with e4m3 precision, we achieved 1,210-1,457 TFLOPs depending on the matrix shape, roughly 31-37% of the theoretical 3,960 TFLOPs peak. 😵 Why?

使用 PyTorch 的 e4m3 `torch._scaled_mm` 精度内核，我们实现了 1,210-1,457 TFLOPs，具体取决于基质形状，大约是理论 3,960 TFLOPs 峰值的 31-37%。😵 为什么？

This lower utilization percentage (in FP8) actually doesn't indicate poor performance; rather, it reflects that these operations become increasingly memory-bound as compute throughput grows.

这种较低的利用率百分比（在 FP8 中）实际上并不表示性能不佳;相反，它反映了随着计算吞吐量的增长，这些作变得越来越受内存限制。

TheTensor Cores can process FP8 data faster than the memory system can deliver it, making memory bandwidth the limiting factor.

Tensor Core 处理 FP8 数据的速度比内存系统传输数据的速度要快，这使得内存带宽成为限制因素。

The Transformer Engine's `TE.Linear` achieved 547-1,121 TFLOPs depending on the shape, while `torch._scaled_mm` consistently delivered higher throughput. This highlights an important lesson: ***kernel implementation matters*** *significantly, and the choice of API can impact performance by 2-3x even when targeting the same hardware capabilities.*

变压器引擎根据 `TE.Linear` 形状实现了 547-1,121 TFLOP，同时 `torch._scaled_mm` 始终提供更高的吞吐量。这凸显了一个重要的教训：内核实现非常重要，即使针对相同的硬件功能，API 的选择也会对性能产生 2-3 倍的影响。

For SmolLM3's training, these practical measurements helped us set realistic throughput expectations. When planning your own training runs, use these achievable numbers rather than theoretical peaks to set your expectations.

对于 SmolLM3 的训练，这些实际测量帮助我们设定了切合实际的吞吐量预期。在计划您自己的训练运行时，请使用这些可实现的数字而不是理论峰值来设定您的期望。

🔧 **Compute Capability 计算能力**

Besides choosing the right kernel API, we also need to ensure those kernels are compiled for the right hardware generation. Compute Capability (CC) is NVIDIA's versioning system that abstracts physical GPU details from the PTX instruction set.

除了选择正确的内核 API 外，我们还需要确保这些内核是针对正确的硬件生成编译的。计算能力 （CC） 是 NVIDIA 的版本控制系统，可从 PTX 指令集中抽象出物理 GPU 详细信息。

It determines which instructions and features your GPU supports.

它决定了 GPU 支持哪些指令和功能。

Why this matters: Kernels compiled for a specific compute capability may not run on older hardware, and you might miss optimizations if your code isn't compiled for your target GPU's CC.

为什么这很重要：为特定计算能力编译的内核可能无法在较旧的硬件上运行，如果您的代码不是针对目标 GPU 的 CC 编译的，您可能会错过优化。

Even worse, frameworks can silently select suboptimal kernels—we discovered PyTorch selecting sm_75 kernels (compute capability 7.5, designed for Turing GPUs) on our H100s, causing mysterious slowdowns.

更糟糕的是，框架可以悄无声息地选择次优内核——我们发现 PyTorch 在我们的 H100 上选择了 sm_75 内核（计算能力 7.5，专为 Turing GPU 设计），导致神秘的减速。

This is asimilar issue documented in the PyTorch community, where frameworks often default to older, more compatible kernels rather than optimal ones. This seemingly minor detail can make the difference between getting 720 TFLOPS or 500 TFLOPS from the same hardware.

这是 PyTorch 社区中记录的类似问题，其中框架通常默认使用更旧、更兼容的内核，而不是最佳内核。这个看似微不足道的细节可以决定从同一硬件获得 720 TFLOPS 或 500 TFLOPS。

When using pre-compiled libraries or custom kernels, always verify they're built for your hardware's compute capability to ensure compatibility and optimal performance. For example, sm90_xmma_gemm_..._cublas indicates a kernel compiled for SM 9.0 (compute capability 9.

使用预编译库或自定义内核时，请务必验证它们是针对硬件的计算能力构建的，以确保兼容性和最佳性能。例如，sm90_xmma_gemm_..._cublas 表示为 SM 9.0 编译的内核（计算能力 9.

0, used by H100). 0，由 H100 使用）。

You can check your GPU's compute capability with `nvidia-smi --query-gpu=compute_cap` or find the technical specifications in the Compute Capability section of the NVIDIA CUDA C Programming Guide.

As we saw, the GPU memory seems to become a bottleneck when computations get too fast at a low precision. Let's have a look at how GPU memory works, and at what causes bottlenecks to occur!

正如我们所看到的，当计算速度过快且精度低时，GPU 内存似乎会成为瓶颈。让我们来看看 GPU 内存是如何工作的，以及导致瓶颈发生的原因！

GPU MEMORY HIERARCHY: FROM REGISTERS TO HBM
GPU 内存层次结构：从寄存器到 HBM

In order to make calculations, GPUs need to read/write to memory, so it's important to know at what speed these transfers happen. Understanding GPU memory hierarchy is crucial for writing high-performance kernels.

为了进行计算，GPU 需要读/写内存，因此了解这些传输发生的速度非常重要。了解 GPU 内存层次结构对于编写高性能内核至关重要。

TL;DR: GPUs organize memory in a hierarchy from fast-but-small (registers, shared memory) to slow-but-large (HBM main memory). Understanding this hierarchy is critical because modern AI is often memory-bound: the bottleneck is moving data, not computing on it.

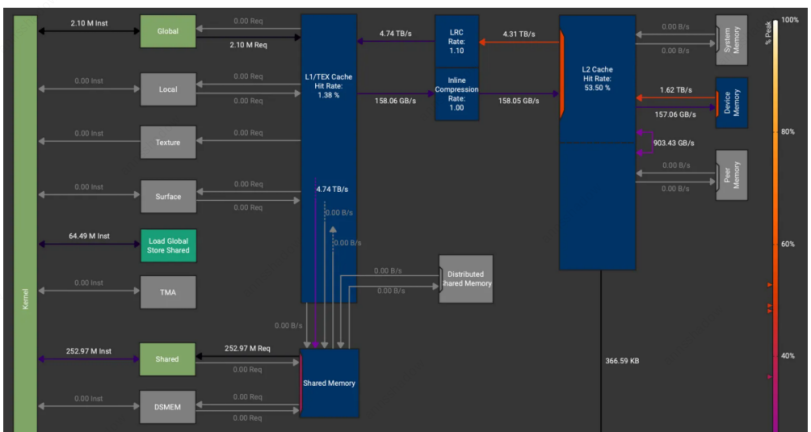
TL;DR：GPU 按层次结构组织内存，从快速但小（寄存器、共享内存）到慢速但大（HBM 主内存）。理解这种层次结构至关重要，因为现代人工智能通常是内存受限的：瓶颈是移动数据，而不是在数据上计算。

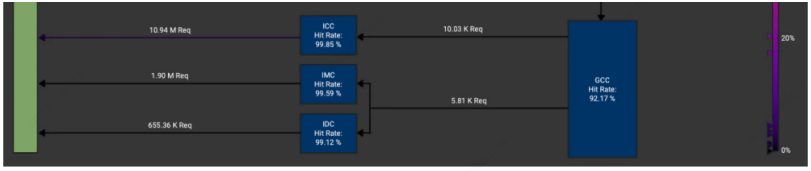
Operator fusion (like Flash Attention) achieves 2-4× speedups by keeping intermediate results in fast on-chip memory instead of writing to slow HBM. Benchmarks show H100's HBM3 delivers ~3 TB/s in practice, matching theoretical specs for large transfers.

运算符融合（如 Flash Attention）通过将中间结果保留在快速片上存储器中而不是写入慢速 HBM 来实现 2-4× 的加速。基准测试显示 H100 的 HBM3 在实践中可提供 ~3 TB/s，与大型传输的理论规格相匹配。

To visualize how memory operations flow through a GPU in practice, let's first look at the Memory Chart from NVIDIA Nsight Compute, a profiling graph which shows a graphical representation of how data moves between different memory units for any kernel of your choice:

为了直观地了解内存作在实践中如何流经 GPU，让我们首先看看 NVIDIA Nsight Compute 中的内存图表，这是一个分析图，它显示了您选择的任何内核的数据如何在不同内存单元之间移动的图形表示：





Memory Chart showing data flow through GPU memory hierarchy during FP64 matrix multiplication on H100

内存图表显示了在 H100 上进行 FP64 矩阵乘法期间通过 GPU 内存层次结构的数据流

In general, a Memory Chart shows both **logical units** (in green) like Global, Local, Texture, Surface, and Shared memory, and **physical units** (in blue) like L1/TEX Cache, Shared Memory, L2 Cache, and Device Memory.

通常，内存图表显示逻辑单元（绿色），如全局、局部、纹理、表面和共享内存，以及物理单元（蓝色），如 L1/TEX 缓存、共享内存、L2 缓存和设备内存。

Links between units represent the number of instructions (Inst) or requests (Req) happening between units, with colors indicating the percentage of peak utilization: from unused (0%) to operating at peak performance (100%).

单元之间的链接表示单元之间发生的指令 （Inst） 或请求 （Req） 的数量，颜色表示峰值利用率的百分比：从未使用 （0%） 到以峰值性能运行 （100%）。

You can generate this memory chart for any kernel using NVIDIA Nsight Compute:

您可以使用 NVIDIA Nsight 计算为任何内核生成此内存图表：

```
## Profile a specific kernel with memory workload analysis
ncu --set full --kernel-name "your_kernel_name" --launch-skip 0 --launch-count 1
## Once profiling is complete, open the results in the Nsight Compute GUI to view
```

It provides several key insights:

它提供了几个关键见解：

- **Bottleneck identification** : Saturated links (shown in red/orange) indicate where data movement is constrained

  瓶颈识别：饱和链接（以红色/橙色显示）指示数据移动受到限制的位置

- **Cache efficiency** : Hit rates for L1/TEX and L2 caches reveal how well your kernel utilizes the memory hierarchy

  缓存效率： L1/TEX 和 L2 缓存的命中率揭示了内核对内存层次结构的利用程度

- **Memory access patterns** : The flow between logical and physical units shows whether your kernel has good spatial/temporal locality

  内存访问模式：逻辑单元和物理单元之间的流动显示您的内核是否具有良好的空间/时间局部性

- **Port utilization** : Individual memory ports may be saturated even when aggregate bandwidth appears underutilized

  端口利用率：即使聚合带宽似乎未得到充分利用，单个内存端口也可能已饱和

In our specific case above, you can see how kernel instructions flow through the memory hierarchy (for FP64 matrix multiplications on our hardware): global load instructions generate requests to the L1/TEX cache, which may hit or miss and generate further requests to L2, which ultimately accesses device memory (HBM) on misses.

在上面的具体案例中，你可以看到内核指令如何流经内存层次结构（对于我们硬件上的 FP64 矩阵乘法）： 全局加载指令生成对 L1/TEX 缓存的请求，该缓存可能会命中或未命中，并生成对 L2 的进一步请求，最终在未命中时访问设备内存 （HBM）。

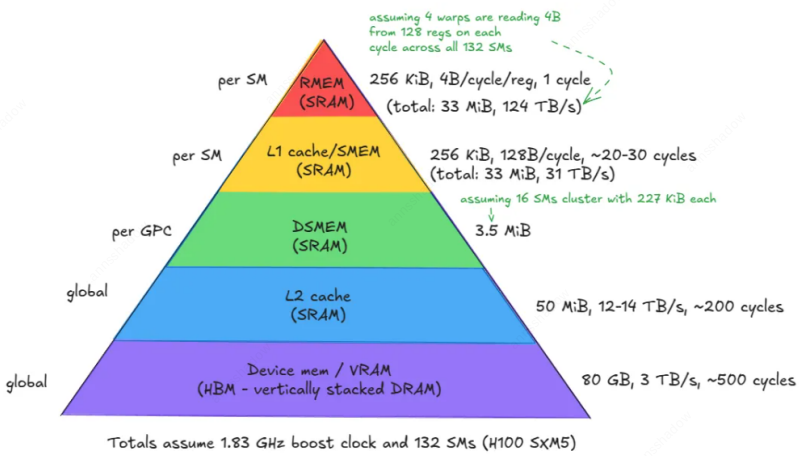
The colored rectangles inside units show port utilization, even if individual links operate below peak, the shared data port may be saturated.

单元内部的彩色矩形显示端口利用率，即使单个链路在峰值以下运行，共享数据端口也可能饱和。

> **优化内存层次结构利用**
>
> For optimal performance, aim to minimize traffic to slower memory tiers (HBM) while maximizing utilization of faster tiers (shared memory, registers).
>
> 为了获得最佳性能，请尽量减少到较慢内存层 （HBM） 的流量，同时最大限度地提高较快层（共享内存、寄存器）的利用率。

Now let's understand the underlying memory hierarchy that makes this chart possible. Modern GPUs organize memory in a hierarchy that balances speed, capacity, and cost, a design dictated by fundamental physics and circuit constraints.

现在让我们了解使此图表成为可能的底层内存层次结构。现代 GPU 将内存组织在平衡速度、容量和成本的层次结构中，这种设计由基本物理和电路约束决定。



Memory hierarchy of the H100 (SXM5) GPU. Source

H100 （SXM5） GPU 的内存层次结构。源

At the bottom of this hierarchy sits **HBM (High Bandwidth Memory):** the GPU's main memory, also called global memory or device memory. The H100 features HBM3 with a theoretical bandwidth of 3.35 TB/s. HBM is the largest but slowest tier in the memory hierarchy.

该层次结构的底部是 HBM（高带宽内存）： GPU 的主内存，也称为全局内存或设备内存。H100 采用 HBM3，理论带宽为 3.35 TB/s。HBM 是内存层次结构中最大但最慢的层。

Moving up the hierarchy toward the compute units, we find progressively faster but smaller memory tiers:

向上移动到计算单元，我们发现内存层速度逐渐加快但更小：

- **L2 cache** : A large SRAM-based cache shared across the GPU, typically several tens of megabytes. On H100, this is 50 MB with a bandwidth of ~13 TB/s

  L2 缓存： 在 GPU 之间共享的基于 SRAM 的大型缓存，通常为几十兆字节。在 H100 上，这是 50 MB，带宽为 ~13 TB/s

- **L1 cache and Shared Memory (SMEM)** : Each Streaming Multiprocessor (SM) has its own L1 cache and programmer-managed shared memory, which share the same physical SRAM storage. On H100, this combined space is 256 KB per SM with bandwidth of ~31 TB/s per SM

  L1 缓存和共享内存 （SMEM） ： 每个流式多处理器 （SM） 都有自己的 L1 缓存和程序员管理的共享内存，它们共享相同的物理 SRAM 存储。在 H100 上，此组合空间为每 SM 256 KB，带宽为每 SM ~31 TB/s

- **Register File (RMEM)** : At the top of the hierarchy, registers are the fastest storage, located directly next to compute units. Registers are private to individual threads and offer bandwidth measured in ~100s TB/s per SM

  寄存器文件 （RMEM） ： 在层次结构的顶部，寄存器是最快的存储，位于计算单元旁边。寄存器对单个线程是专用的，并提供以每个 SM ~100s TB/s 为单位的带宽

This hierarchy exists because SRAM (used for caches and registers) is fast but physically large and expensive, while DRAM (used for HBM) is dense and cheap but slower.

这种层次结构之所以存在，是因为 SRAM（用于缓存和寄存器）速度快但物理上庞大且昂贵，而 DRAM（用于 HBM）密集且便宜但速度较慢。

The result: fast memory comes in small quantities close to compute, backed by progressively larger pools of slower memory further away.

结果：快速内存在接近计算的少量内存中出现，并由越来越大的慢速内存池支持。

Why this matters : Understanding this hierarchy is essential for kernel optimization. The key insight is that memory-bound operations are limited by how fast you can move data, not how fast you can compute. As Horace He explains in Making Deep Learning Go Brrrr From First Principles, *"load from memory" → "multiply by itself*

twice" → *"write to memory"* takes essentially the same time as *"load from memory"* → *"multiply by itself once"* → *"write to memory"*: the computation is "free" compared to the memory access.

为什么这很重要：了解这个层次结构对于内核优化至关重要。关键的见解是，内存绑定作受移动数据的速度限制，而不是计算速度。正如 Horace He 在《从第一性原理让深度学习变得 Brrrr》中所解释的那样，"从内存加载"→"自身乘以两次"→"写入内存"与"从内存加载"→"乘以自身一次"→"写入内存"所花费的时间基本相同：与内存访问相比，计算是"免费的"。

This is why **operator fusion** is so powerful: by combining multiple operations into a single kernel, you can keep intermediate results in fast SRAM instead of writing them back to slow HBM between operations. Flash Attention is a perfect example of this principle in action.

这就是运算符融合如此强大的原因：通过将多个作组合到一个内核中，您可以将中间结果保留在快速 SRAM 中，而不是在作之间将它们写回慢速 HBM。闪光是这一原则的完美例子。

> **Flash Attention: A Case Study in Memory Hierarchy Optimization**
>
> **闪光：内存层次结构优化案例研究**
>
> Standard attention implementations are memory-bound because they materialize the full attention matrix in HBM:
>
> 标准注意力实现是内存绑定的，因为它们在 HBM 中实现了完整的注意力矩阵：
>
> 1. Compute `Q @ K^T` → write N×N attention scores to HBM
>
>   计算 `Q @ K^T` →将 N×N 注意力分数写入 HBM
>
> 2. Apply softmax → read from HBM, compute, write back to HBM
>
>   应用 softmax→从 HBM 读取、计算、写回 HBM
>
> 3. Multiply by V → read attention scores from HBM again
>
>   乘以 V →再次读取 HBM 的注意力分数
>
> Flash Attention achieves its 2-4× speedup by **fusing these operations** and keeping intermediate results in SRAM:
>
> Flash Attention 通过融合这些作并将中间结果保留在 SRAM 中来实现 2-4× 的加速：
>
> - Instead of computing the full attention matrix, it processes attention in tiles that fit in SRAM
>
>   它不是计算完整的注意力矩阵，而是在适合 SRAM 的图块中处理注意力
>
> - Intermediate attention scores never leave the fast on-chip memory
>
>   中间注意力分数永远不会离开快速片上存储器
>
> - Only the final output is written back to HBM
>
>   只有最终输出才会写回 HBM
>
> **The result**: Flash Attention reduces HBM accesses from O(N²) to O(N), transforming a memory-bound operation into one that better utilizes the GPU's compute capabilities. This is the essence of efficient kernel design: *minimize slow memory movement, maximize fast computation* .
>
> 结果：Flash Attention 将 HBM 访问从 O（N²） 减少到 O（N），将内存受限的作转变为更好地利用 GPU 计算能力的作。这就是高效内核设计的本质：最大限度地减少缓慢的内存移动，最大化快速计算。

Example: Validating our HBM3 Bandwidth in Practice

示例：在实践中验证我们的 HBM3 带宽

Now that we understand the memory hierarchy, let's put theory into practice and validate the actual bandwidth on our H100 GPUs! This is where benchmarking tools become essential.

现在我们了解了内存层次结构，让我们将理论付诸实践并验证 H100 GPU 上的实际带宽！这就是基准测试工具变得必不可少的地方。

NVBandwidth is NVIDIA's open-source benchmarking tool designed specifically for measuring bandwidth and latency across GPU systems.

NVBandwidth 是 NVIDIA 的开源基准测试工具，专为测量 GPU 系统的带宽和延迟而设计。

It evaluates data transfer rates for various memory copy patterns—host-to-device, device-to-host, and device-to-device operations—using both copy engines and kernel-based methods.

它使用复制引擎和基于内核的方法评估各种内存复制模式（主机到设备、设备到主机和设备到设备作）的数据传输速率。

The tool is particularly valuable for assessing inter-GPU communication (for exampleNVLink and PCIe, two types of connectors) and validating system performance in multi-GPU environments.

该工具对于评估 GPU 间通信（例如 NVLink 和 PCIe，两种类型的连接器）和验证多 GPU 环境中的系统性能特别有价值。

You can install NVBandwidth from NVIDIA's GitHub repository. The tool outputs detailed bandwidth matrices showing how efficiently data transfers between different devices, making it ideal for diagnosing performance bottlenecks or verifying healthy GPU interconnects.

您可以从 NVIDIA 的 GitHub 存储库安装 NVBandwidth。该工具输出详细的带宽矩阵，显示不同设备之间数据传输的效率，使其成为诊断性能瓶颈或验证 GPU 互连健康的理想选择。

Let's use it to measure our H100's local memory bandwidth using the `device_local_copy` test, which measures bandwidth of `cuMemcpyAsync` between device buffers local to the GPU across different message sizes.

让我们用它来测量 H100 的本地内存带宽，该 `device_local_copy` 测试测量不同消息大小的 GPU 本地设备缓冲区 `cuMemcpyAsync` 之间的带宽。

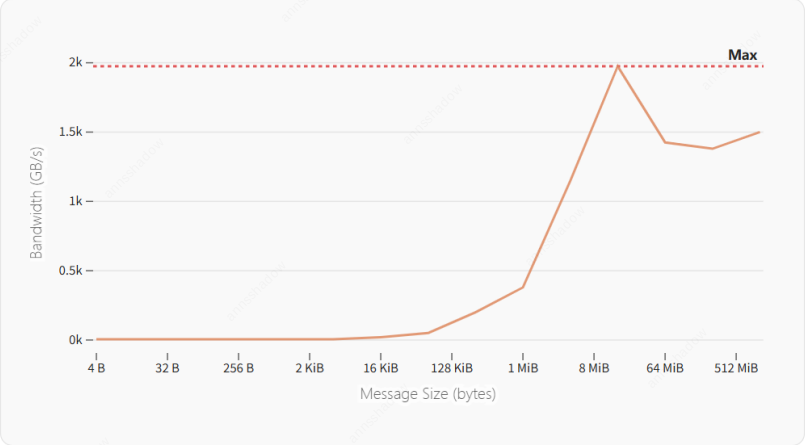
cuMemcpyAsync is a CUDA driver API function that asynchronously copies data between two memory pointers, inferring the type of transfer (host-to-host, host-to-device, device-to-device, or device-to-host)

cuMemcpyAsync 是一个 CUDA 驱动程序 API 函数，用于在两个内存指针之间异步复制数据，推断传输类型（主机到主机、主机到设备、设备到设备或设备到主机）

```
$ ./nvbandwidth -t device_local_copy -b 2048
memcpy local GPU(column) bandwidth (GB/s)
           0         1         2         3         4         5         6
 0   1519.07   1518.93   1519.07   1519.60   1519.13   1518.86   1519.13   1519.3
```

Measured H100 Local Memory Bandwidth

实测的 H100 本地内存带宽



The results reveal an important characteristic of memory systems: **for small message sizes (< 1 MB), we're latency-bound** rather than bandwidth-bound. The overhead of initiating memory transfers dominates performance, preventing us from reaching peak bandwidth. However, **for large message sizes (≥ 1 MB), we achieve sustained bandwidth of ~1,500 GB/s for both read and write operations** .

结果揭示了内存系统的一个重要特征：对于小消息大小（< 1 MB），我们受到延迟限制，而不是带宽限制。启动内存传输的开销主导了性能，阻止我们达到峰值带宽。但是，对于大消息大小（≥ 1 MB），我们为读取和写入作实现了 ~1,500 GB/s 的持续带宽。

Since HBM bandwidth accounts for both reads and writes happening simultaneously, we sum these to get **3 TB/s total bidirectional bandwidth** (1,519 read + 1,519 write), which closely validates the H100's theoretical 3.35 TB/s HBM3 specification.

由于 HBM 带宽考虑了同时发生的读取和写入，因此我们将这些频加得到 3 TB/s 的总双向带宽（1,519 次读取 + 1,519 次写入），这密切验证了 H100 的理论 3.35 TB/s HBM3 规范。

ROOFLINE MODEL 车顶线模型

Understanding whether your kernel is compute-bound or memory-bound determines which optimizations will help.

了解内核是计算受限还是内存受限，可以确定哪些优化会有所帮助。

There are two scenarios:

有两种情况：

- If you're **memory-bound** (spending most time moving data), increasing compute throughput won't help: You need to reduce memory traffic through techniques like operator fusion.

  如果内存受限（将大部分时间花在移动数据上），则增加计算吞吐量将无济于事：您需要通过运算符融合等技术来减少内存流量。

- If you're **compute-bound** (spending most time on FLOPs), optimizing memory access patterns won't help: You need more compute power or better algorithms.

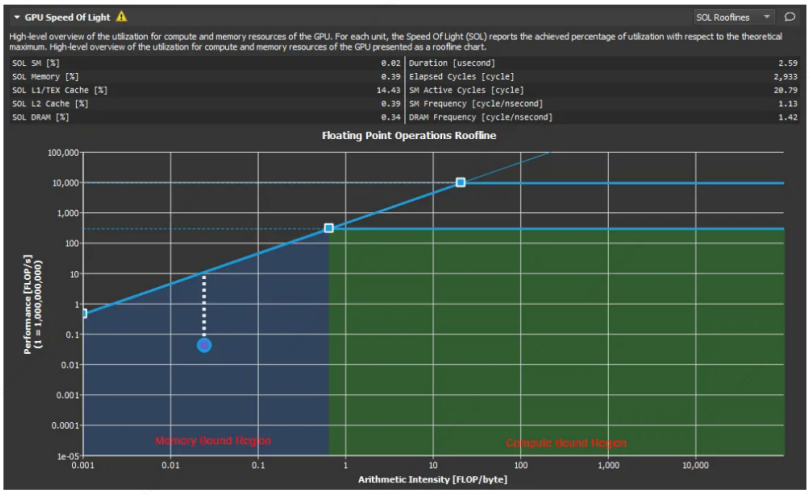
  如果你受计算限制（大部分时间都花在 FLOP 上），优化内存访问模式将无济于事：你需要更多的计算能力或更好的算法。

The *roofline model* provides a visual framework for understanding these performance characteristics and identifying optimization opportunities.

车顶线模型提供了一个可视化框架，用于理解这些性能特征并识别优化机会。

Let's apply it to a real kernel analysis. It's available in the NSight Compute profiling tool we mentioned before (under "roofline analysis view"). Here's what we get:

让我们将其应用于真正的内核分析。它可在我们之前提到的 NSight Compute 分析工具（在“屋顶线分析视图”下）中找到。这是我们得到的：



Roofline chart showing kernel performance boundaries - Source: NVIDIA NSight Compute Profiling Guide

显示内核性能边界的屋顶图 - 来源：NVIDIA NSight 计算分析指南

Let's see how we can read this chart, which has two axes:

让我们看看如何阅读这张图表，它有两个轴：

- **Vertical axis (FLOP/s)** : Shows achieved floating-point operations per second, using a logarithmic scale to accommodate the large range of values

  垂直轴 （FLOP/s） ： 显示每秒实现的浮点运算，使用对数刻度来适应大范围的值

- **Horizontal axis (Arithmetic Intensity)** : Represents the ratio of work (FLOPs) to memory traffic (bytes), measured in FLOPs per byte. This also uses a logarithmic scale.

  水平轴（算术强度）： 表示工作 （FLOP） 与内存流量（字节）的比率，以每字节的 FLOP 为单位。这也使用对数刻度。

The roofline itself consists of two boundaries:

车顶线本身由两个边界组成：

- **Memory Bandwidth Boundary** (sloped line): Determined by the GPU's memory transfer rate (HBM bandwidth). Performance along this line is limited by how fast data can be moved.

  内存带宽边界（斜线）： 由 GPU 的内存传输速率（HBM 带宽）决定。这方面的性能受到数据移动速度的限制。

- **Peak Performance Boundary** (flat line): Determined by the GPU's maximum compute throughput. Performance along this line is limited by how fast computations can be executed.

  峰值性能边界（平线）： 由 GPU 的最大计算吞吐量决定。这方面的性能受到计算执行速度的限制。

The **ridge point** where these boundaries meet represents the transition between memory-bound and compute-bound regimes.

这些边界相交的脊点表示内存绑定和计算绑定状态之间的转换。

We can interpret the performance by looking at the two divided regions of the chart:

我们可以通过查看图表的两个划分区域来解释性能：

- **Memory Bound** (below the sloped boundary): Kernels in this region are limited by memory bandwidth. The GPU is waiting for data, increasing compute power won't help.

  内存边界（倾斜边界以下）：此区域中的内核受内存带宽限制。GPU 正在等待数据，增加计算能力无济于事。

  Optimizations should focus on reducing memory traffic through techniques like operator fusion, better memory access patterns, or increasing arithmetic intensity.

  优化应侧重于通过运算符融合、更好的内存访问模式或增加算术强度等技术来减少内存流量。

- **Compute Bound** (below the flat boundary): Kernels in this region are limited by compute throughput. The GPU has enough data but can't process it fast enough. Optimizations should focus on algorithmic improvements or leveraging specialized hardware like Tensor Cores.

  计算边界（低于平面边界）：此区域中的内核受计算吞吐量的限制。GPU 有足够的数据，但处理速度不够快。优化应侧重于算法改进或利用 Tensor Core 等专用硬件。

The **achieved value** (the plotted point) shows where your kernel currently sits. The distance from this point to the roofline boundary represents your optimization headroom, the closer to the boundary, the more optimal your kernel's performance.

实现的值（绘制的点）显示内核当前所在的位置。从该点到屋顶线边界的距离代表您的优化余量，越靠近边界，内核的性能就越优化。

In our example, the kernel sits in the memory-bound region, indicating that there's still room for improvement by optimizing memory traffic!

在我们的示例中，内核位于内存绑定区域，这表明通过优化内存流量仍有改进的空间！

For a deeper dive into GPU internals including detailed explanations of CUDA cores, Tensor Cores, memory hierarchies, and low-level optimization techniques, check out the Ultrascale Playbook! Now that we understand what happens *inside* a GPU, let's zoom out and explore how GPUs communicate with the rest of the world.

要更深入地了解 GPU 内部结构，包括 CUDA 内核、张量核心、内存层次结构和低级优化技术的详细解释，请查看 Ultrascale Playbook！现在我们了解了 GPU 内部发生的事情，让我们缩小并探索 GPU 如何与世界其他地方通信。

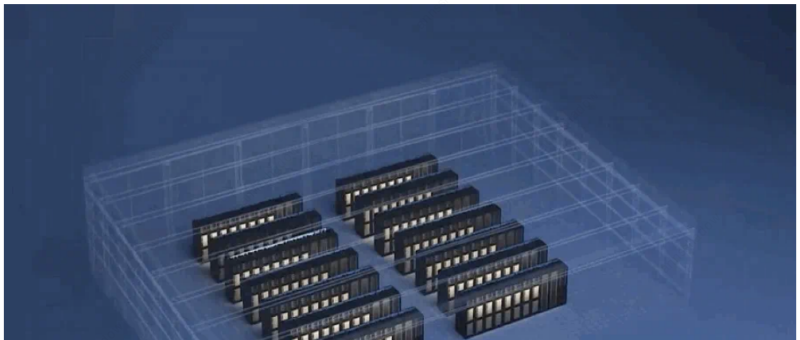
Outside a GPU: How GPUs Talk to the World
GPU 之外：GPU 如何与世界对话

Now that we understand how a GPU performs computation using its internal memory hierarchy, we need to address a critical reality: a GPU doesn't operate in isolation. Before any computation can happen, data must be loaded into the GPU's memory.

现在我们了解了 GPU 如何使用其内部内存层次结构执行计算，我们需要解决一个关键现实：GPU 不是孤立运行的。在进行任何计算之前，必须将数据加载到 GPU 的内存中。

The CPU needs to schedule kernels and coordinate work. And in distributed training, GPUs must constantly exchange activations, gradients, and model weights with each other.

CPU 需要调度内核并协调工作。而在分布式训练中，GPU 必须不断地相互交换激活、梯度和模型权重。



DGX H100. Source: NVIDIA

DGX H100。来源：NVIDIA

This is where the external communication infrastructure becomes crucial. No matter how powerful your GPU's compute units are, if data can't reach them fast enough, whether from the CPU, from storage, or from other GPUs, your expensive hardware sits idle.

这就是外部通信基础设施变得至关重要的地方。无论您的 GPU 的计算单元多么强大，如果数据无法足够快地到达它们，无论是来自 CPU、存储还是其他 GPU，您昂贵的硬件都会闲置。

Understanding these communication pathways and their bandwidth characteristics is essential for maximizing hardware utilization and minimizing bottlenecks.

了解这些通信路径及其带宽特性对于最大限度地提高硬件利用率和最大限度地减少瓶颈至关重要。

In this section, we'll look at four critical communication links that connect a GPU to the outside world:

在本节中，我们将了解将 GPU 连接到外部世界的四个关键通信链路：

- **GPU-CPU** : How the CPU schedules work and transfers data to GPUs

  GPU-CPU：CPU 如何调度工作并将数据传输到 GPU

- **GPU-GPU intra-node** : How GPUs on the same machine communicate

  GPU-GPU 节点内：同一台机器上的 GPU 如何通信

- **GPU-GPU inter-node** : How GPUs across different machines communicate over the network

  GPU-GPU 节点间：不同机器上的 GPU 如何通过网络进行通信

- **GPU-Storage** : How data flows from storage to GPU memory

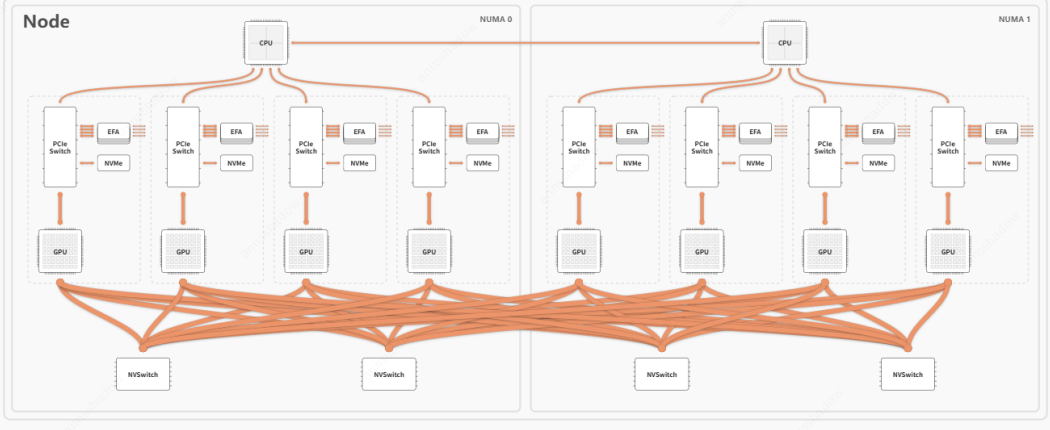
  GPU-存储：数据如何从存储流向 GPU 内存

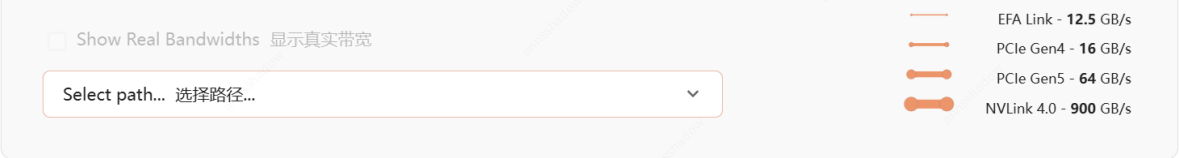
Each of these links has different bandwidth and latency characteristics, and understanding them will help you identify where your training pipeline might be bottlenecked.

这些链接中的每一个都有不同的带宽和延迟特征，了解它们将有助于你确定训练管道可能出现瓶颈的位置。

To make this easier to understand, we've created a simplified diagram that highlights the most important components and communication links:

为了使其更易于理解，我们创建了一个简化的图表，突出显示了最重要的组件和通信链接：





Simplified diagram of the key components and communication links in our AWS p5 instance setup

AWS p5 实例设置中关键组件和通信链路的简化图

If this looks overwhelming, don't worry. We'll dive into each of these connections in detail and measure their actual bandwidths to understand the performance characteristics of each link.

如果这看起来势不可挡，请不要担心。我们将详细研究每个连接并测量它们的实际带宽，以了解每个链路的性能特征。

GPU-TO-CPU GPU 到 CPU

> **TL;DR:** The CPU orchestrates GPU work via PCIe connections, which bottleneck at ~14.2 GB/s (PCIe Gen4 x8) for CPU-to-GPU transfers in our p5 instance. CPU-GPU latency is ~1.4 microseconds, which adds kernel launch overhead that is problematic for workloads with many small kernels.
>
> TL;DR：CPU 通过 PCIe 连接编排 GPU 工作，在我们的 p5 实例中，CPU 到 GPU 的传输速

度为 ~14.2 GB/s （PCIe Gen8 x5）。CPU-GPU 延迟为 ~1.4 微秒，这增加了内核启动开销，这对于具有许多小内核的工作负载来说是有问题的。

CUDA Graphs can reduce this overhead by batching operations. NUMA affinity is critical on multi-socket systems; running GPU processes on the wrong CPU socket adds significant latency.

CUDA 图可以通过批处理作来减少这种开销。NUMA 亲和性在多插槽系统上至关重要;在错误的 CPU 插槽上运行 GPU 进程会增加显着的延迟。

Modern architectures like Grace Hopper eliminate PCIe bottlenecks with NVLink-C2C (900 GB/s vs 128 GB/s).

Grace Hopper 等现代架构通过 NVLink-C2C 消除了 PCIe 瓶颈（900 GB/s 与 128 GB/s）。

The CPU is the orchestrator of GPU computation. It's responsible for launching kernels, managing memory allocations, and coordinating data transfers. But how fast can the CPU actually communicate with the GPU? This is determined by the **PCIe (Peripheral Component Interconnect Express)** connection between them.

CPU 是 GPU 计算的编排器。它负责启动内核、管理内存分配和协调数据传输。但 CPU 与 GPU 的实际通信速度有多快呢？这是由它们之间的 PCIe（外围组件互连 Express）连接决定的。

Understanding this link is crucial because it affects:

了解这种联系至关重要，因为它会影响：

- **Kernel launch latency** : How quickly the CPU can schedule work on the GPU

  内核启动延迟：CPU 在 GPU 上调度工作的速度

- **Data transfer speed** : How fast we can move data between CPU and GPU memory

  数据传输速度： 我们在 CPU 和 GPU 内存之间移动数据的速度有多快

- **Synchronization overhead** : The cost of CPU-GPU coordination points

  同步开销：CPU-GPU 协调点的成本

In modern GPU servers, the CPU-GPU connection has evolved significantly. While earlier systems used direct PCIe connections, modern high-performance systems like the DGX H100 use more sophisticated topologies with PCIe *switches* to manage multiple GPUs efficiently. And with the latest GB200 architecture, NVIDIA has taken this even further by placing the CPU and GPU on the same printed circuit board, eliminating the need for external switches altogether.

在现代 GPU 服务器中，CPU-GPU 连接已经发生了显着变化。早期的系统使用直接 PCIe 连接，而 DGX H100 等现代高性能系统则使用更复杂的拓扑和 PCIe 交换机来有效管理多个 GPU。借助最新的 GB200 架构，NVIDIA 更进一步，将 CPU 和 GPU 放置在同一印刷电路板上，完全消除了对外部开关的需求。

Let's examine the physical topology of our p5 instance using `lstopo` and then measure the actual performance of this critical link, to identify potential bottlenecks.

让我们检查 `lstopo` p5 实例的物理拓扑，然后测量此关键链路的实际性能，以识别潜在的瓶颈。

```
$ lstopo -v
...
HostBridge L#1 (buses=0000:[44-54])
    PCIBridge L#2 (busid=0000:44:00.0 id=1d0f:0200 class=0604(PCIBridge) link=15.
        PCIBridge L#3 (busid=0000:45:00.0 id=1d0f:0200 class=0604(PCIBridge) link
            ...
            PCIBridge L#12 (busid=0000:46:01.4 id=1d0f:0200 class=0604(PCIBridge)
                PCI L#11 (busid=0000:53:00.0 id=10de:2330 class=0302(3D) link=63.
                    Co-Processor(CUDA) L#8 (Backend=CUDA GPUVendor="NVIDIA Corpor
                    GPU(NVML) L#9 (Backend=NVML GPUVendor="NVIDIA Corporation" GP
            ...
...

```

From the `lstopo` output, we can see two key PCIe bandwidth values in our system:

从 `lstopo` 输出中，我们可以看到系统中的两个关键 PCIe 带宽值：

- **15.75GB/s** : corresponds to PCIe Gen4 x8 links (CPU to PCIe switches)

  15.75GB/s： 对应 PCIe Gen4 x8 链路（CPU 到 PCIe 交换机）

- **63.02GB/s** : corresponds to PCIe Gen5 x16 links (PCIe switches to GPUs)

63.02GB/s： 对应 PCIe Gen5 x16 链路（PCIe 切换到 GPU）

To have a better understanding of the whole topology, we can visualize it using:

为了更好地理解整个拓扑，我们可以使用以下方法对其进行可视化：

```
$ lstopo --whole-system lstopo-diagram.png

```

This diagram showcases the hierarchical structure of our system:

这张图展示了我们系统的层次结构：

- It contains two **NUMA** (Non-Uniform Memory Access) nodes (NUMA is memory zone per CPU socket)

  它包含两个 NUMA（非统一内存访问）节点（NUMA 是每个 CPU 插槽的内存区域）

- Each **CPU socket** connects to four **PCIe switches** via **PCIe Gen4** x8 links (15.75GB/s)

  每个 CPU 插槽通过 PCIe Gen4 x8 链路 （15.75GB/s） 连接到四个 PCIe 交换机

- Each **PCIe switch** connects to one **H100 GPU** via **PCIe Gen5** x16 links (63.02GB/s)

  每个 PCIe 交换机通过 PCIe Gen5 x16 链路 （63.02GB/s） 连接到一个 H100 GPU

- ... (We'll explore the other components like NVSwitch, EFA network cards and NVMe drives in next sections.)

  ...（我们将在下一节中探讨其他组件，如 NVSwitch、EFA 网卡和 NVMe 驱动器。

The PCIe specification differs between generations, each doubling the transfer rate per lane.

PCIe 规格因代而异，每代人将每通道的传输速率提高了一倍。

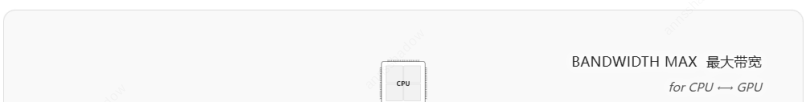
Note that Transfer Rate is measured in GT/s (GigaTransfers per second), which represents the raw signaling rate, while Throughput is measured in GB/s (Gigabytes per second), which accounts for encoding overhead and represents the actual usable bandwidth:

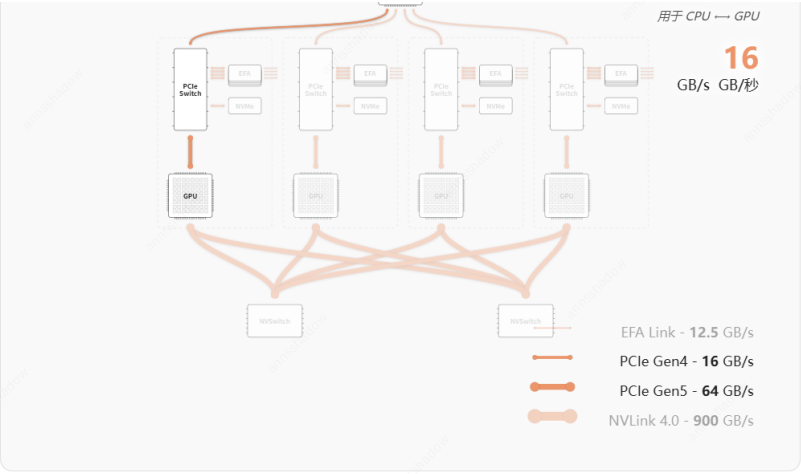
请注意，传输速率以 GT/s（每秒千兆传输数）为单位，它代表原始信令速率，而吞吐量以 GB/s（千兆字节每秒）为单位，它考虑了编码开销并表示实际可用带宽：

PCIe Version PCIe 版本	Transfer Rate (per lane) 传输速率（每通道）	Throughput (GB/
×1	×2	×4
1.0	2.5 GT/s 2.5 吨/秒	0.25
2.0	5.0 GT/s 5.0 GT/秒	0.5
3.0	8.0 GT/s 8.0 GT/秒	0.985
4.0	16.0 GT/s 16.0 吨/秒	1.969
5.0	32.0 GT/s 32.0 吨/秒	3.938
6.0	64.0 GT/s 64.0 吨/秒	7.563
7.0	128.0 GT/s 128.0 吨/秒	15.125

Theoretical PCIe bandwidths. Source: https://en.wikipedia.org/wiki/PCI_Express

理论 PCIe 带宽。来源：https://en.wikipedia.org/wiki/PCI_Express





CPU-to-GPU communication path

From the topology diagram and the PCIe bandwidth table, we can see that the CPU-to-GPU path goes through two PCIe hops: first from the CPU to the PCIe switch via **PCIe Gen4** x8 (15.754 GB/s), then from the PCIe switch to the GPU via **PCIe Gen5** x16 (63.015 GB/s). *This means the bottleneck for CPU-GPU communication is the first hop at 15.754 GB/s* . Let's validate this with another util, `nvbandwidth` !

从拓扑图和 PCIe 带宽表中，我们可以看到 CPU 到 GPU 的路径经过两个 PCIe 跃点：首先通过 PCIe Gen4 x8 （15.754 GB/s） 从 CPU 到 PCIe 交换机，然后通过 PCIe Gen5 x16 （63.015 GB/s） 从 PCIe 交换机到 GPU。这意味着 CPU-GPU 通信的瓶颈是 15.754 GB/s 的第一跳。让我们用另一个实用程序！ `nvbandwidth`

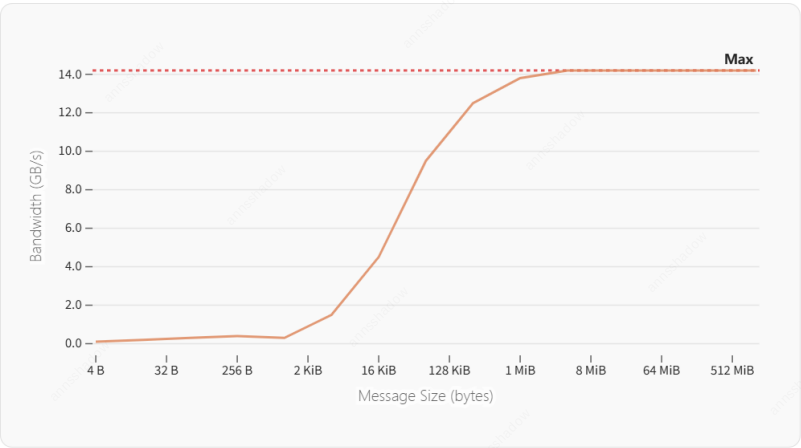
The `host_to_device_memcpy_ce` command measures bandwidth of `cuMemcpyAsync` from host (CPU) memory to device (GPU) memory using the GPU's copy engines.

该 `host_to_device_memcpy_ce` 命令使用 GPU 的复制引擎测量从主机 （CPU） 内存到设备 （GPU） 内存的 `cuMemcpyAsync` 带宽。

```
./nvbandwidth -t host_to_device_memcpy_ce -b <message_size> -i 5
```

CPU-> GPU measured bandwidth

CPU > GPU 实测带宽



CPU-to-GPU bandwidth measured with nvbandwidth's host_to_device test, showing the PCIe Gen4 x8 bottleneck at ~14.2 GB/s for large transfers

使用 nvbandwidth 的 host_to_device 测试测量的 CPU 到 GPU 带宽，显示 PCIe Gen4 x8 瓶颈为 ~14.2 GB/s 用于大传输

large message sizes we achieve **~14.2 GB/s** , which is about 90% of the theoretical 15.754 GB/s bandwidth for PCIe Gen4 x8. This confirms that in **CPU-GPU** communication, the CPU-to-PCIe switch link is indeed our bottleneck.

结果确实表明，对于小消息大小，我们受到延迟限制，但对于大消息大小，我们实现了 ~14.2 GB/s ， 这大约是 PCIe Gen90 x15.754 理论 4 GB/s 带宽的 8%。这证实了在 CPU-GPU 通信中，CPU 到 PCIe 的交换机链路确实是我们的瓶颈。

Beyond bandwidth, **latency** is equally important for CPU-GPU communication since it determines how quickly we can schedule kernels. To measure this, we use `nvbandwidth` 's `host_device_latency_sm` test, which employs a pointer-chase kernel to measure round-trip latency. The `host_device_latency_sm` test measures round-trip latency by allocating a buffer on the host (CPU) and accessing it from the GPU using a pointer-chase kernel. This simulates the real-world latency of CPU-GPU communication.

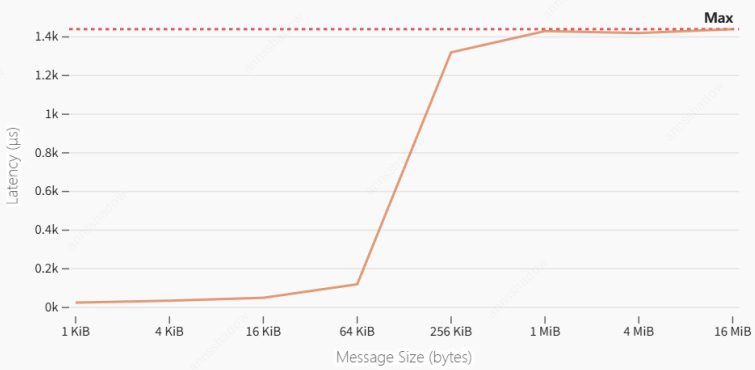
除了带宽之外，延迟对于 CPU-GPU 通信同样重要，因为它决定了我们调度内核的速度。为了测量这一点，我们使用 `nvbandwidth` 的 `host_device_latency_sm` 测试，它使用指针跟踪内核

来测量往返延迟。该 `host_device_latency_sm` 测试通过在主机 （CPU） 上分配缓冲区并使用指针追踪内核从 GPU 访问它来测量往返延迟。这模拟了 CPU-GPU 通信的真实延迟。

```
./nvbandwidth -t host_device_latency_sm -i 5
```

CPU-> GPU measured latency

CPU > GPU 测量延迟



CPU-to-GPU latency measured with nvbandwidth's host_device_latency_sm test (adapted to make buffer size variable), showing approximately 1.4 microseconds round-trip latency

使用 nvbandwidth 的 host_device_latency_sm 测试（调整为使缓冲区大小可变）测量的 CPU 到 GPU 延迟，显示大约 1.4 微秒的往返延迟

The results show that **latency** is approximately **1.4 microseconds** . This explains the kernel launch overhead of a few microseconds we often observe in ML workloads. For workloads launching many small kernels, the added latency can become a bottleneck; otherwise, overhead is hidden by overlapping execution.

结果显示，延迟约为 1.4 微秒。这解释了我们在 ML 工作负载中经常观察到的几微秒的内核启动开销。对于启动许多小内核的工作负载，增加的延迟可能会成为瓶颈;否则，重叠执行将隐藏开销。

For example for small models or with small batches we can see inference saturate on the GPU because of the kernel launches. FlashFormer addresses this by fusing a whole layer to gain speedups (Nrusimha et al., 2025).

例如，对于小型模型或小批量模型，我们可以看到由于内核启动而导致 GPU 上的推理饱和。FlashFormer 通过融合整层来获得加速来解决这个问题（Nrusimha 等人，2025 年）。

> 🚀 **CUDA Graphs for Reducing Launch Overhead**
>
> **用于减少启动开销的 CUDA 图**
>
> CUDA Graphs can significantly reduce kernel launch overhead by capturing a sequence of operations and replaying them as a single unit, eliminating microseconds of CPU-GPU round-trip latency for each kernel launch.
>
> CUDA Graphs 可以通过捕获一系列作并将它们作为单个单元重播来显着减少内核启动开销，从而消除每次内核启动的 CPU-GPU 往返延迟微秒。
>
> This is particularly beneficial for workloads with many small kernels or frequent CPU-GPU synchronization.
>
> 这对于具有许多小内核或频繁 CPU-GPU 同步的工作负载特别有利。
>
> For more details on understanding and optimizing launch overhead, seeUnderstanding the Visualization of Overhead and Latency in NVIDIA Nsight Systems.
>
> 有关了解和优化启动开销的更多详细信息，请参阅了解 NVIDIA Nsight 系统中开销和延迟的可视化。

> ⚠️ **MoE Models and CPU-GPU Synchronization Overhead**
>
> **MoE 模型和 CPU-GPU 同步开销**
>
> Some implementations of Mixture-of-Experts (MoE) models require CPU-GPU synchronization in each iteration to schedule the appropriate kernels for the selected experts.
>
> 混合专家 （MoE） 模型的某些实现需要在每次迭代中进行 CPU-GPU 同步，以便为所选专家安排适当的内核。
>
> This introduces kernel launch overhead that can significantly affect throughput, especially when the CPU-GPU connection is slow.
>
> 这会引入内核启动开销，从而显着影响吞吐量，尤其是在 CPU-GPU 连接速度较慢的情况下。
>
> For example, inMakoGenerate's optimization of DeepSeek MOE kernels, the reference implementation dispatched 1,043 kernels with 67 CPU-GPU synchronization points per forward pass.

例如，在 MakoGenerate 对 DeepSeek MOE 内核的优化中，参考实现分派了 1,043 个内核，每次前向传递有 67 个 CPU-GPU 同步点。

By restructuring the expert routing mechanism, they reduced this to 533 kernel launches and just 3 synchronization points, achieving a 97% reduction in synchronization overhead and 44% reduction in end-to-end latency.

通过重组专家路由机制，他们将其减少到 533 次内核启动和仅 3 个同步点，从而实现了 97% 的同步开销和 44% 的端到端延迟减少。

Note that not all MoE implementations require CPU-GPU synchronization (modern implementations often keep routing entirely on the GPU), but for those that do, efficient CPU-GPU communication becomes critical for performance.

请注意，并非所有 MoE 实现都需要 CPU-GPU 同步（现代实现通常将路由完全放在 GPU 上），但对于那些需要的实现来说，高效的 CPU-GPU 通信对于性能至关重要。

> 🔗 **Grace Hopper Superchips: A Different Approach to CPU-GPU Communication**
>
> **Grace Hopper 超级芯片：一种不同的 CPU-GPU 通信方法**
>
> NVIDIA's Grace Hopper superchips take a fundamentally different approach to CPU-GPU communication compared to traditional x86+Hopper systems. Key improvements include:
>
> 与传统的 x86+Hopper 系统相比，NVIDIA 的 Grace Hopper 超级芯片采用了根本不同的 CPU-GPU 通信方法。主要改进包括：
>
> - **1:1 GPU to CPU ratio** (compared to 4:1 for x86+Hopper), providing 3.5x higher CPU memory bandwidth per GPU
>
> GPU 与 CPU 的比率为 1：1（相比之下，x86+Hopper 为 4：1），每个 GPU 的 CPU 内存带宽提高了 3.5 倍
>
> - **NVLink-C2C** replacing PCIe Gen5 lanes, delivering 900 GB/s vs 128 GB/s (7x higher GPU-CPU link bandwidth)
>
> NVLink-C2C 取代 PCIe Gen5 通道，提供 900 GB/s 和 128 GB/s（GPU-CPU 链路带宽高 7 倍）
>
> - **NVLink Switch System** providing 9x higher GPU-GPU link bandwidth than InfiniBand NDR400 NICs connected via PCIe Gen4
>
> NVLink 交换机系统提供比通过 PCIe Gen4 连接的 InfiniBand NDR400 NIC 高 9 倍的 GPU-GPU 链路带宽
>
> For more details, see the NVIDIA Grace Hopper Superchip Architecture Whitepaper (page 11).
>
> 有关更多详细信息，请参阅 NVIDIA Grace Hopper 超级芯片架构白皮书（第 11 页）。

⚠️ **NUMA Affinity: Critical for Multi-Socket Performance**

⚠️ **NUMA 亲和性：对多插槽性能至关重要**

On multi-socket systems like our AMD EPYC 7R13 nodes (2 sockets, 48 cores each), **** **NUMA affinity**is crucial for GPU performance** . It refers to running processes on CPU cores that share the same socket as their target devices (like GPUs).

在多插槽系统上，例如我们的 AMD EPYC 7R13 节点（2 个插槽，每个 48 个内核），**** NUMA 亲和力对于 GPU 性能至关重要**。它是指在与目标设备（如 GPU）共享同一插槽的 CPU 内核上运行进程。

When your GPU process runs on CPUs from a different NUMA node than where the GPU is attached, operations must traverse the CPU interconnect (AMD Infinity Fabric), adding significant latency and bandwidth constraints.

当您的 GPU 进程在与 GPU 连接位置不同的 NUMA 节点的 CPU 上运行时，作必须遍历 CPU 互连 （AMD Infinity Fabric），从而增加显着的延迟和带宽限制。

First, let's examine the NUMA topology and node distances to understand the performance implications :

首先，让我们检查一下 NUMA 拓扑和节点距离，以了解性能影响：

```
$ numactl --hardware
node distances:
node   0   1
  0:  10  32
  1:  32  10
```

The distance values show that accessing memory on the same NUMA node (distance

10) is much faster than crossing to the other NUMA node (distance 32). This **3.2x difference** in memory access **latency** can significantly impact GPU performance when your process is pinned to the wrong NUMA node.

距离值显示，访问同一 NUMA 节点上的内存（距离 10）比跨越到另一个 NUMA 节点（距离 32）快得多。当您的进程固定到错误的 NUMA 节点时，内存访问延迟的 3.2 倍差异可能会显着影响 GPU 性能。

For detailed steps on diagnosing and resolving NUMA-related performance issues, see the Troubleshooting Interconnect Performance section.

有关诊断和解决与 NUMA 相关的性能问题的详细步骤，请参阅 互连性能故障排除 部分。

GPU-TO-GPU INTRANODE GPU 到 GPU 节点内

In distributed training, GPUs must frequently exchange gradients, weights, and activations, often gigabytes of data per iteration. This huge amount of data requires careful handling of communication.

在分布式训练中，GPU 必须频繁交换梯度、权重和激活，每次迭代通常交换 GB 的数据。如此庞大的数据量需要谨慎处理通信。

While the H100's internal HBM can read at about 3 TB/s, accidentally using the wrong flags can completely flunk your GPU-to-GPU communication bandwidth!

虽然 H100 的内部 HBM 可以以大约 3 TB/s 的速度读取，但不小心使用错误的标志可能会完全破坏您的 GPU 到 GPU 的通信带宽！

Let's see why by examining all the ways you can do communication between GPUs within the same node (and all the flags you should - or should not - set) 😊

让我们通过检查同一节点内 GPU 之间进行通信的所有方式（以及您应该设置或不应该设置😊的所有标志）来了解原因

> **TL;DR:** GPUs within a node can communicate three ways: through CPU (slowest, ~3 GB/s, bottlenecked by PCIe), via GPUDirect RDMA over EFA NICs (~38 GB/s), or GPUDirect RDMA via NVLink (~786 GB/s bidirectional). NVLink is 9-112x faster and bypasses CPU/PCIe entirely.
>
> TL;DR：节点内的 GPU 可以通过三种方式进行通信：通过 CPU（最慢，~3 GB/s，受 PCIe 瓶颈）、通过 EFA NIC 上的 GPUDirect RDMA（~38 GB/s）或通过 NVLink 的 GPUDirect RDMA（~786 GB/s 双向）。NVLink 的速度提高了 9-112 倍，并且完全绕过了 CPU/PCIe。
>
> NCCL automatically prioritizes NVLink when available. NVLink SHARP (NVLS) provides hardware-accelerated collectives, boosting allreduce performance by 1.3x to 480 GB/s.
>
> NCCL 会在可用时自动确定 NVLink 的优先级。NVLink SHARP （NVLS） 提供硬件加速集体，将 allreduce 性能提高 1.3 倍，达到 480 GB/s。
>
> However, alltoall operations (340 GB/s) don't benefit from NVLS acceleration.
>
> 但是，alltoall 作 （340 GB/s） 不会从 NVLS 加速中受益。

> 但是，alltoall 作 （340 GB/s） 不会从 NVLS 加速中受益。

THROUGH CPU 通过 CPU

The naive approach uses host memory (SHM): data travels from GPU1 through the PCIe switch to the CPU, into host memory, back through the CPU, through the PCIe switch again, and finally to GPU2. This can be achieved (although not recommended) using `NCCL_P2P_DISABLE=1` and `FI_PROVIDER=tcp` environment variable by NCCL. When this mode is activated, you can verify it by setting `NCCL_DEBUG=INFO`, which will show messages like:

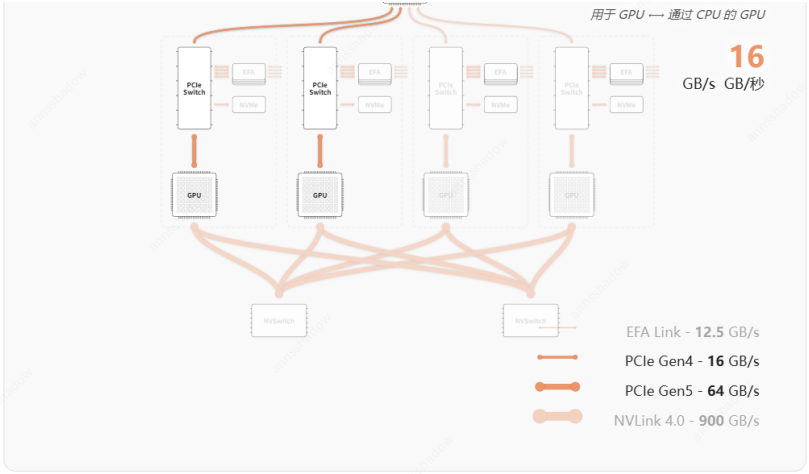
朴素的方法使用主机内存 （SHM）：数据从 GPU1 通过 PCIe 交换机传输到 CPU，进入主机内存，再返回 CPU，再次通过 PCIe 交换机，最后到达 GPU2。这可以通过 NCCL 使用 和 `FI_PROVIDER=tcp` 环境变量来 `NCCL_P2P_DISABLE=1` 实现（尽管不推荐）。激活此模式后，您可以通过设置 `NCCL_DEBUG=INFO` 进行验证，这将显示如下消息：

```
NCCL INFO Channel 00 : 1[1] -> 0[0] via SHM/direct/direct

```

BANDWIDTH MAX 最大带宽

for GPU ⟷ GPU via CPU



GPU-to-GPU communication path through CPU and main memory, showing the inefficient roundtrip through the PCIe switch and CPU.

GPU 到 GPU 的通信路径通过 CPU 和主内存，显示出通过 PCIe 交换机和 CPU 的低效往返。

This roundabout path involves multiple memory copies and saturates both PCIe and CPU memory buses, causing congestion.

这条迂回路径涉及多个内存副本，并使 PCIe 和 CPU 内存总线饱和，从而导致拥塞。

In our topology where 4 H100s share the same CPU memory buses, this congestion becomes even more problematic when multiple GPUs attempt simultaneous communication, as they compete for the same limited CPU memory bandwidth... 😳

在我们的拓扑中，4 个 H100 共享相同的 CPU 内存总线，当多个 GPU 尝试同时通信时，这种拥塞变得更加成问题，因为它们争夺相同的有限 CPU 内存带宽.....😳

With this CPU-mediated approach, we're fundamentally bottlenecked by the PCIe Gen4 x8 link at ~16 GB/s between the CPU and PCIe switch. Fortunately, there's a better way for our GPUs to communicate without involving the CPU: **GPUDirect RDMA** .

通过这种 CPU 介导的方法，我们从根本上受到 CPU 和 PCIe 交换机之间 ~4 GB/s 的 PCIe Gen8 x16 链路的瓶颈。幸运的是，我们的 GPU 有一种更好的方式可以在不涉及 CPU 的情况下进行通信： GPUDirect RDMA .

THROUGH LIBFABRIC EFA 通过 LIBFABRIC EFA

GPUDirect RDMA (Remote Direct Memory Access or GDRDMA) is a technology that enables direct communication between NVIDIA GPUs by allowing direct access to GPU memory.

GPUDirect RDMA（远程直接内存访问或 GDRDMA）是一种通过允许直接访问 GPU 内存来实现 NVIDIA GPU 之间直接通信的技术。

This eliminates the need for data to pass through the system CPU and avoids buffer copies via system memory, resulting in up to 10x better performance compared to traditional CPU-mediated transfers.

这消除了数据通过系统 CPU 的需要，并避免了通过系统内存进行缓冲区副本，与传统的 CPU 介导传输相比，性能提高了 10 倍。

GPUDirect RDMA works over PCIe to enable fast GPU-to-GPU communication both within a node (as we see here) and across nodes using**NICs** (network interface cards, as we'll see in a future section) with RDMA capabilities. For more details, see NVIDIA GPUDirect.

GPUDirect RDMA 通过 PCIe 工作，在节点内（如我们在这里看到的）和使用具有 RDMA 功能的 NIC（网络接口卡，我们将在以后的部分中看到）实现节点之间的快速 GPU 到 GPU 通信。有关更多详细信息，请参阅 NVIDIA GPUDirect。

EFA exposes a **libfabric** interface (a specific communication API for high performance computations) that applications can use, and provides RDMA-like capabilities that enable **GPUDirect RDMA** for direct GPU-to-GPU communication across nodes.

回顾我们的拓扑图，我们可以看到每个 PCIe 交换机都有 4 个 EFA（弹性结构适配器）NIC，这意味着每个 GPU 可以访问 4 个 EFA 适配器。EFA 是 AWS 为云实例定制的高性能网络接口，旨在提供低延迟、高吞吐量的实例间通信。在 p5 实例上，EFA 公开了应用程序可以使用的 libfabric 接口（用于高性能计算的特定通信 API），并提供类似 RDMA 的功能，使 GPUDirect RDMA 能够跨节点进行直接的 GPU 到 GPU 通信。

to use commodity datacenter networks (with a large number of network paths). Learn about its importance here.

EFA 使用一种可靠的基于以太网的传输协议，称为可扩展可靠数据报 （SRD），旨在使用商用数据中心网络（具有大量网络路径）。在此处了解其重要性。

```
$ lstopo -v
...
## We can see 4 such EFA devices per each PCIe switch
```

```
PCIBridge L#8 (busid=0000:46:01.0 id=1d0f:0200 class=0604(PCIBridge) link=15.75GB
PCI L#6 (busid=0000:4f:00.0 id=1d0f:efa1 class=0200(Ethernet) link=15.75GB/s PCIS
    OpenFabrics L#4 (NodeGUID=cd77:f833:0000:1001 SysImageGUID=0000:0000:0000:000
...

$ fi_info --verbose
        fi_link_attr:
            address: EFA-fe80::14b0:33ff:fef8:77cd
            mtu: 8760           # maximum packet size is 8760 bytes
            speed: 100000000000  # each EFA link provides 100 Gbps of bandwidth
            state: FI_LINK_UP
            network_type: Ethernet
```

Each **EFA link** provides 100 Gbps (12.5 GB/s) of bandwidth. With **4 EFA NICs** per GPU and 8 GPUs per node, this gives an aggregate bandwidth of 100 × 4 × 8 = **3200 Gbps per node** (400GB/s).

每个 EFA 链路提供 100 Gbps （12.5 GB/s） 的带宽。每个 GPU 有 4 个 EFA NIC，每个节点有 8 个 GPU，因此每个节点的总带宽为 100 × 4 × 8 = 3200 Gbps （400GB/s）。

For a detailed exploration of how to fully harness this 3200 Gbps bandwidth using libfabric and EFA, see Lequn Chen's excellent blog series: Harnessing 3200 Gbps Network: A Journey with RDMA, EFA, and libfabric.

有关如何使用 libfabric 和 EFA 充分利用这 3200 Gbps 带宽的详细探索，请参阅 Lequn Chen 的优秀博客系列：利用 3200 Gbps 网络：RDMA、EFA 和 libfabric 之旅。

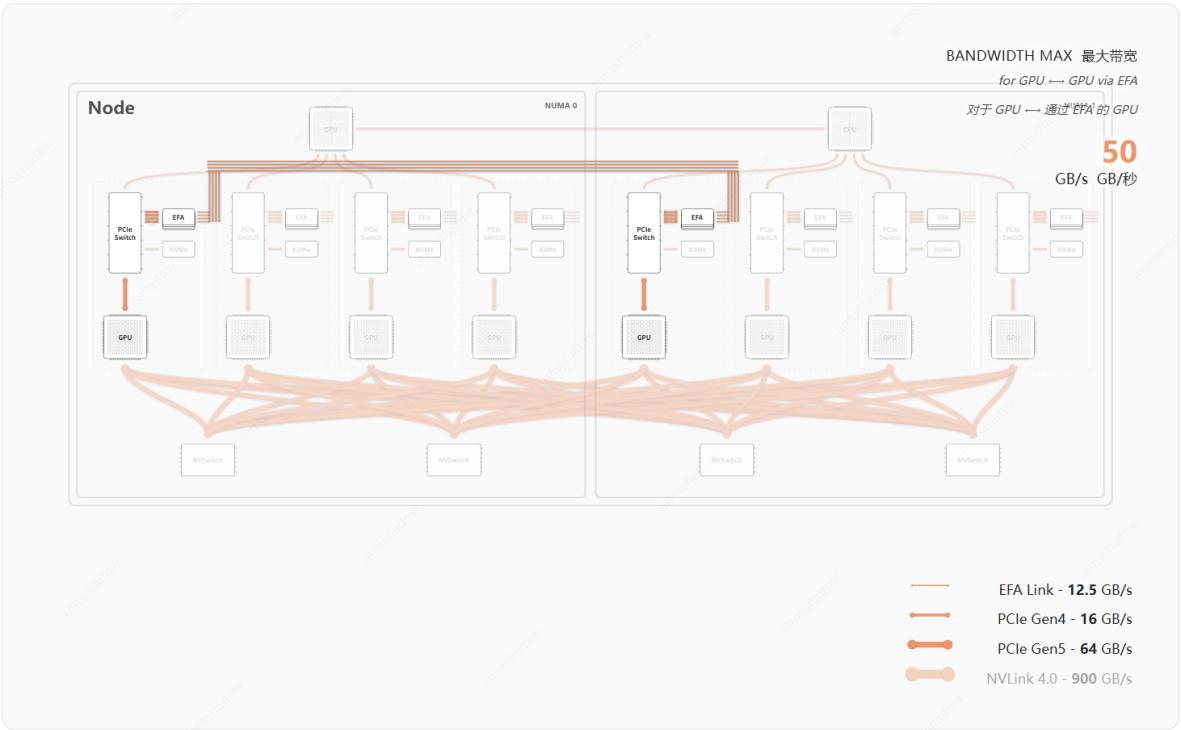
To make sure we're enabling GPUDirect RDMA over EFA, you should set the `FI_PROVIDER=efa` and `NCCL_P2P_DISABLE=1` environment variables. When this mode is activated, you can verify it that it's working by setting `NCCL_DEBUG=INFO`, which will show messages like:

为确保我们启用 GPUDirect RDMA over EFA，您应该设置 `FI_PROVIDER=efa` 和 `NCCL_P2P_DISABLE=1` 环境变量。激活此模式后，您可以通过设置 `NCCL_DEBUG=INFO` 来验证它是否正常工作，这将显示如下消息：

```
NCCL INFO Channel 01/1 : 1[1] -> 0[0] [receive] via NET/Libfabric/0/GDRDMA/Shared
```

```
NCCL INFO Channel 01/1 : 1[1] -> 0[0] [receive] via NET/Libfabric/0/GDRDMA/Shared

```



GPU-to-GPU communication path through Libfabric EFA. Note that this is less efficient for intranode communications compared to using NVLink.

通过 Libfabric EFA 的 GPU 到 GPU 通信路径。请注意，与使用 NVLink 相比，这对于节点内通信的效率较低。

While GPUDirect RDMA over EFA provides significant improvements over CPU-mediated transfers, achieving around **50 GB/s** with 4 EFA cards per GPU, can we do even better? This is where NVLink comes into play.

虽然 GPUDirect RDMA over EFA 比 CPU 介导的传输提供了显着改进，每个 GPU 使用 4 个 EFA 卡实现了大约 50 GB/s，但我们能做得更好吗？这就是 NVLink 发挥作用的地方。

THROUGH NVLINK 通过 NVLINK

NVLink is NVIDIA's high-speed, direct GPU-to-GPU interconnect technology that enables fast multi-GPU communication within servers. The **H100** employs fourth-generation NVLink (NVLink 4.0), providing 900 GB/s bidirectional bandwidth per GPU through 18 links, each operating at 50 GB/s bidirectional (NVIDIA H100 Tensor Core GPU Datasheet).

NVLink 是 NVIDIA 的高速、直接 GPU 到 GPU 互连技术，可在服务器内实现快速的多 GPU 通信。H100 采用第四代 NVLink （NVLink 4.0），通过 900 个链路为每个 GPU 提供 18 GB/s 的双向带宽，每个链路以 50 GB/s 的双向运行速度（NVIDIA H100 Tensor Core GPU 数据表）。

In the **DGX H100** architecture, 4 third-generation NVSwitches connect the 8 GPUs using a layered topology where each GPU connects with 5+4+4+5 links across the switches.

在 DGX H100 架构中，4 个第三代 NVSwitche 使用分层拓扑连接 8 个 GPU，其中每个 GPU 通过交换机连接 5+4+4+5 链路。

This configuration ensures multiple direct paths between any GPU pair with a constant hop count of just 1 NVSwitch, resulting in 3.6 TB/s total bidirectional NVLink network bandwidth.

此配置可确保任何 GPU 对之间具有多个直接路径，恒定跃点数仅为 1 个 NVSwitch，从而产生 3.6 TB/s 的双向 NVLink 网络总带宽。

NVLink 2.0 (Volta) NVLink 2.0 （伏特）	NVLink 3.0 (Ampere) NVLink 3.0（安培）	NV
Bandwidth 带宽	300 GB/s 300 GB/秒	60

Table: NVLink bandwidth comparison across generations, showing theoretical specifications

表：NVLink 各代带宽比较，显示理论规格

By default, NCCL prioritizes NVLink for intra-node GPU communication when available, as it provides the lowest latency and highest bandwidth path between GPUs on the same machine. However, if you did not set your flags properly, you could be preventing the use of NVLink! 😱

默认情况下，NCCL 会优先考虑 NVLink 进行节点内 GPU 通信（如果可用），因为它在同一台机器上的 GPU 之间提供最低延迟和最高带宽路径。但是，如果您没有正确设置标志，则可能会阻止使用 NVLink！😱

NVLink enables direct GPU-to-GPU memory access without involving the CPU or system memory. When NVLink is unavailable, NCCL falls back to GPUDirect P2P over PCIe, or uses Shared Memory (SHM) transport when inter-socket PCIe transfers would be suboptimal.

NVLink 支持直接的 GPU 到 GPU 内存访问，而无需涉及 CPU 或系统内存。当 NVLink 不可用时，NCCL 会回退到 GPUDirect P2P over PCIe，或者在插槽间 PCIe 传输不理想时使用共享内存 （SHM） 传输。

To verify that NVLink is being used, set `NCCL_DEBUG=INFO` and look for messages like:

要验证是否正在使用 NVLink，请设置 `NCCL_DEBUG=INFO` 并查找如下消息：

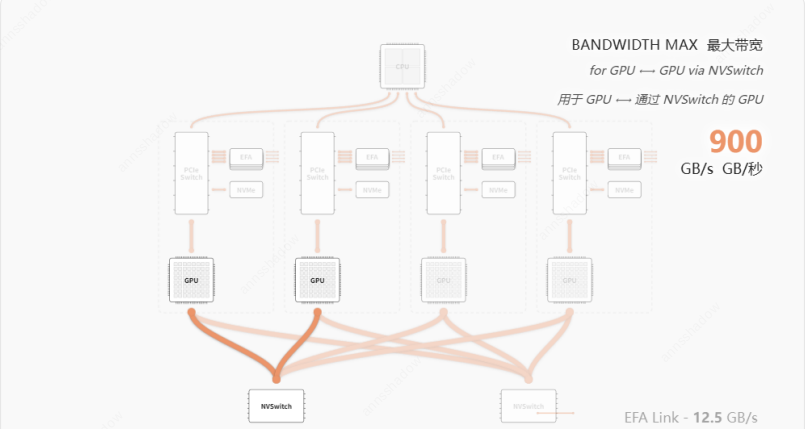
```
NCCL INFO Channel 00/1 : 0[0] -> 1[1] via P2P/CUMEM
```

CUMEM indicates that the peer-to-peer operations use **CUDA memory handles (cuMem API).** Learn more here.

CUMEM 表示对等作使用 CUDA 内存句柄 （cuMem API）。在此处了解更多信息。

The following diagram illustrates the direct path that data takes when using NVLink:

下图说明了使用 NVLink 时数据所采用的直接路径：



GPU-to-GPU communication path through NVLink.

通过 NVLink 的 GPU 到 GPU 通信路径。

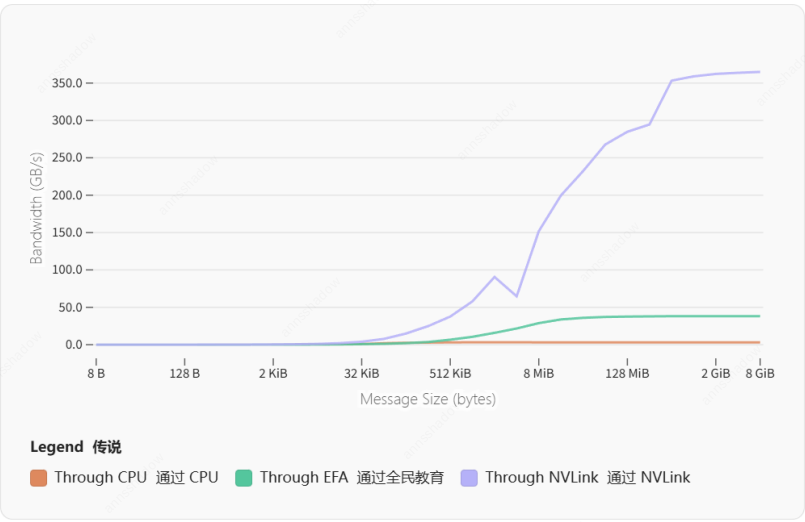
With **NVLink 4.0** 's theoretical bandwidth of 900 GB/s compared to **EFA** 's ~50 GB/s, we expect an 18x advantage for intra-node communication. To validate this in practice, we ran NCCL's SendRecv performance test to measure actual bandwidth across different communication paths:

与 EFA 的 ~50 GB/s 相比，NVLink 4.0 的理论带宽为 900 GB/s，我们预计节点内通信将有 18 倍的优势。为了在实践中验证这一点，我们运行了 NCCL 的 SendRecv 性能测试来测量不同通信路径的实际带宽：

```
$ FI_PROVIDER=XXX NCCL_P2P_DISABLE=X sendrecv_perf -b 8 -e 8G -f 2 -g 1 -c 1 -n 1
```

GPU-> GPU measured bandwidth with NCCL's SendRecv test (H100 GPUs, 1 Node, 2 GPUs)

使用 NCCL 的 SendRecv 测试（H100 GPU、1 节点、2 个 GPU）测量的 GPU > GPU 测量带宽



This shows without a doubt how much more efficient NVLink is: it achieves 364.93 GB/s compared to EFA's 38.16 GB/s (9x faster, or 18x bidirectional) and the CPU baseline's 3.24 GB/s (112.6x faster).

这无疑表明 NVLink 的效率要高得多：与 EFA 的 38.16 GB/s（快 9 倍，或双向快 18 倍）和 CPU 基线的 3.24 GB/s（快 112.6 倍）相比，它实现了 364.93 GB/s。

These measurements confirm why NCCL prioritizes NVLink for intra-node GPU communication, but to further check NVLink's performance, let's use `nvbandwidth` to measure bidirectional bandwidth between all GPU pairs using simultaneous copies in both directions:

查 NVLink 的性能，让我们使用 `nvbandwidth` 双向同步副本来测量所有 GPU 对之间的双向带宽：

```
./nvbandwidth -t device_to_device_bidirectional_memcpy_write_ce -b <message_size>
memcpy CE GPU(row) <-> GPU(column) Total bandwidth (GB/s)
           0         1         2         3         4         5         6
 0       N/A    785.81    785.92    785.90    785.92    785.78    785.92    785.9
 1    785.83       N/A    785.87    785.83    785.98    785.90    786.05    785.9
 2    785.87    785.89       N/A    785.83    785.96    785.83    785.96    786.0
 3    785.89    785.85    785.90       N/A    785.96    785.89    785.90    785.9
 4    785.87    785.96    785.92    786.01       N/A    785.98    786.14    786.0
 5    785.81    785.92    785.85    785.89    785.89       N/A    786.10    786.0
 6    785.94    785.92    785.99    785.99    786.10    786.05       N/A    786.0
 7    785.94    786.07    785.99    786.01    786.05    786.05    786.14       N/

SUM device_to_device_bidirectional_memcpy_write_ce_total 44013.06
```

The measured bidirectional bandwidth of **786 GB/s** represents 85% of **NVLink 4.0** 's theoretical 900 GB/s specification. Using NVLink has bypassed the CPU bottleneck entirely (for gpu-to-gpu communication)!

实测的双向带宽为 786 GB/s，是 NVLink 4.0 理论 900 GB/s 规格的 85%。使用 NVLink 完

全绕过了 CPU 瓶颈（用于 GPU 到 GPU 的通信）！

But how does this translate to collective communication patterns? Let's measure `allreduce` performance within a single node with the `all_reduce_perf` benchmark from NCCL tests.

但这如何转化为集体沟通模式呢？让我们使用 NCCL 测试的 `all_reduce_perf` 基准来衡量 `allreduce` 单个节点内的性能。

For a quick refresher on collective communication patterns, see the UltraScale Playbook Appendix.

有关集体通信模式的快速复习，请参阅 UltraScale Playbook 附录。

```
$ ./all_reduce_perf -b 8 -e 16G -f 2 -g 1 -c 1 -n 100

```

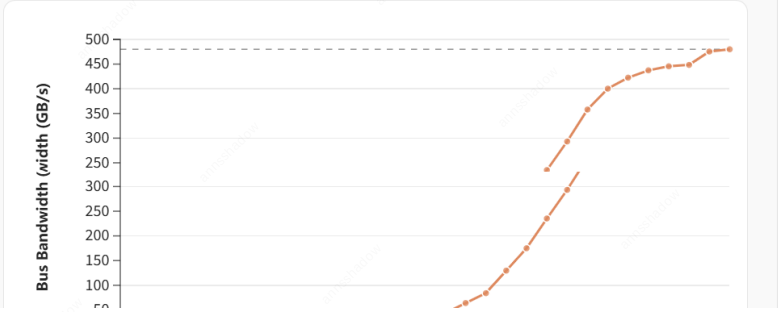
NCCL's All-Reduce performance test intranode

NCCL 的 All-Reduce 节点内性能测试

For comprehensive benchmarking scripts and configurations, see the excellent collection at AWS Distributed Training Samples.

有关全面的基准测试脚本和配置，请参阅 AWS 分布式训练示例中的优秀集合。

例中的优秀集合。

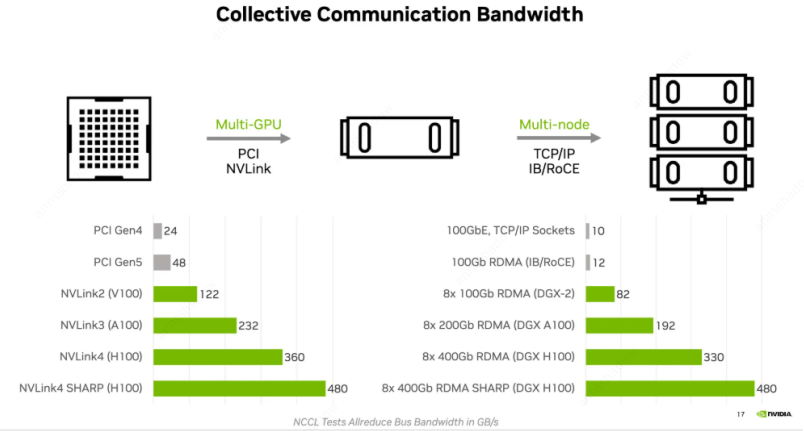


But wait... We are achieving 480 GB/s, which exceeds the theoretical unidirectional bandwidth of 450 GB/s for NVLink 4.0 😱 What is this sorcery, and how is this possible?

但是等等.....我们正在实现 480 GB/s，这超过了 NVLink 450 的理论单向带宽 4.0 😱 GB/s 这是什么魔法，这怎么可能?

Diving a bit in the docs, it seems like the answer lies in **NVLink SHARP (NVLS)** , NVIDIA's hardware-accelerated collective operations technology. They provide approximately 1.3x speedup for allreduce operations on a single node with H100 GPUs!

深入研究文档，答案似乎就在 NVLink SHARP （NVLS） ，NVIDIA 的硬件加速集体运营技术。它们为具有 H100 GPU 的单个节点上的 allreduce 作提供了大约 1.3 倍的加速!



For technical details on how NVSwitch enables these hardware-accelerated collective operations, see the NVSwitch architecture presentation.
For technical details on how NVSwitch enables these hardware-accelerated collective operations, see the NVSwitch architecture presentation.

有关 NVSwitch 如何实现这些硬件加速的集体作的技术详细信息，请参阅 NVSwitch 架构演示文稿。

Can they help in other places too? Let's examine alltoall performance:

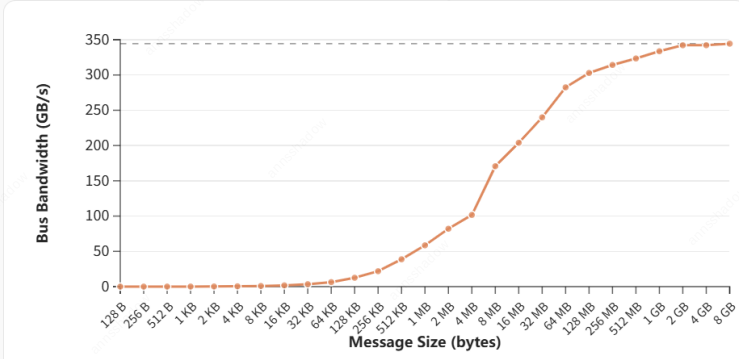
他们也能在其他地方提供帮助吗？让我们检查一下 alltoall 的性能：

```
$ ./all_to_all_perf -b 8 -e 16G -f 2 -g 1 -c 1 -n 100

```

NCCL's All-to-All performance test intranode

NCCL 的 All-to-All 性能测试节点内



We achieve **340 GB/s** for alltoall operations, which aligns with published benchmarks showing similar performance characteristics for H100 systems with NVLink 4.0 (source). Unlike allreduce, alltoall operations don't benefit from NVLS hardware acceleration, which explains why we see 340 GB/s here compared to the 480 GB/s achieved with allreduce.

For custom NVLink communication patterns, keep an eye on PyTorch's SymmetricMemory API, which enables fine-grained control over NVLink and

我们实现了 340 GB/s 的 alltoall 作，这与已发布的基准测试一致，显示具有 NVLink 100 的 H4.0 系统具有相似的性能特征（来源）。与 allreduce 不同，alltoall 作不会从 NVLS 硬件加速中受益，这解释了为什么我们在这里看到 340 GB/s，而 allreduce 实现的 480 GB/s。

The alltoall pattern requires more complex point-to-point data exchanges between all GPU pairs, relying purely on NVLink's base bandwidth rather than NVSwitch's collective acceleration features.

alltoall 模式需要所有 GPU 对之间进行更复杂的点对点数据交换，纯粹依赖于 NVLink 的基本带宽，而不是 NVSwitch 的集体加速功能。

NVLS operations.

对于自定义 NVLink 通信模式，请关注 PyTorch 的 SymmetricMemory API，它可以对 NVLink 和 NVLS 作进行细粒度控制。

> **Advanced Kernel Optimization**
>
> **高级内核优化**
>
> Some optimized kernels separate NVLink communication from compute by assigning dedicated warps to handle transfers.
>
> 一些优化的内核通过分配专用的 warp 来处理传输，将 NVLink 通信与计算分开。
>
> For example, ThunderKittens uses a warp-level design where specific warps issue NVLink transfers and wait for completion, while other warps continue compute operations.
>
> 例如，ThunderKittens 使用扭曲级设计，其中特定扭曲发出 NVLink 传输并等待完成，而其他扭曲则继续计算作。
>
> This fine-grained overlap of SM compute and NVLink communication can hide most inter-GPU communication latency.
>
> SM 计算和 NVLink 通信的这种细粒度重叠可以隐藏大多数 GPU 间通信延迟。
>
> For implementation details, see theThunderKittens blog post on multi-GPU kernels.
>
> 有关实现的详细信息，请参阅有关多 GPU 内核的 ThunderKittens 博客文章。

While NVLink provides exceptional bandwidth within a single node, training frontier models requires scaling across multiple nodes.

虽然 NVLink 在单个节点内提供了出色的带宽，但训练前沿模型需要跨多个节点进行扩展。

This introduces a new potential bottleneck: the inter-node network interconnect, which operates at significantly lower bandwidths than NVLink.

这引入了一个新的潜在瓶颈：节点间网络互连，其运行带宽明显低于 NVLink。

GPU-TO-GPU INTERNODE GPU 到 GPU 节点间

> **TL;DR** Multi-node GPU communication uses high-speed networks like InfiniBand (400 Gbps) or RoCE (100 Gbps). Allreduce scales well (320-350 GB/s stable across nodes), enabling massive training clusters. Alltoall degrades more sharply due to algorithm complexity.
>
> TL;DR 多节点 GPU 通信使用 InfiniBand （400 Gbps） 或 RoCE （100 Gbps） 等高速网络。Allreduce 可很好地扩展（跨节点稳定 320-350 GB/s），支持大规模训练集群。由于算法复杂性，Alltoall 的退化幅度更大。
>
> Latency jumps from ~13µs intra-node to 55µs+ inter-node. For MoE workloads requiring frequent all-to-all operations, NVSHMEM offers asynchronous GPU-initiated communication with significantly better performance than CPU-orchestrated transfers.
>
> 节点内延迟从 ~13µs 跃升至节点间 55µs+。对于需要频繁进行全对所有作的 MoE 工作负载，NVSHMEM 提供异步 GPU 启动的通信，其性能明显优于 CPU 编排的传输。

As models scale beyond what a single node can accommodate, training requires distributing computation across multiple nodes connected via high-speed networks.

随着模型的扩展超出单个节点的容纳范围，训练需要将计算分布在通过高速网络连接的多个节点上。

Before diving into the benchmarks, let's look at the 3 key networking technologies connecting nodes you'll encounter in multi-node GPU clusters:

在深入研究基准测试之前，让我们先看看连接多节点 GPU 集群中节点的 3 种关键网络技术：

- **Ethernet** has evolved from 1 Gbps to 100+ Gbps speeds and remains widely used in HPC and data center clusters.

  以太网已经从 1 Gbps 发展到 100+ Gbps 的速度，并且仍然广泛应用于 HPC 和数据中心集群。

- **RoCE (RDMA over Converged Ethernet)** brings RDMA capabilities to Ethernet

networks, using ECN for congestion control instead of traditional TCP mechanisms.

RoCE（RDMA over Converged Ethernet）将 RDMA 功能引入以太网，使用 ECN 进行拥塞控制，而不是传统的 TCP 机制。

- **InfiniBand** is NVIDIA's industry-standard switch fabric, providing up to 400 Gbps bandwidth and sub-microsecond latency with RDMA support that enables direct GPU-to-GPU memory access while bypassing the host CPU through GPUDirect RDMA.

  InfiniBand 是 NVIDIA 的行业标准交换机结构，提供高达 400 Gbps 的带宽和亚微秒级延迟，并支持 RDMA，可实现直接的 GPU 到 GPU 内存访问，同时通过 GPUDirect RDMA 绕过主机 CPU。

As a summary: 总结一下：

Name 名字	Ethernet (25–100 Gbps) 以太网 （25–100 Gbps）	Eth
Manufacturer 制造者	Many 多	Ma
Unidirectional Bandwidth (Gbps) 单向带宽 （Gbps）	25–100	200
End to End Latency (µs) 端到端延迟 （µs）	10-30	N/A
RDMA RDMA 的	No 不	No

Table: Comparison of Interconnects. Source: https://www.sciencedirect.com/science/article/pii/S2772485922000618

表：互连比较。来源：https://www.sciencedirect.com/science/article/pii/S2772485922000618

For AWS p5 instances we have Elastic Fabric Adapter (**EFA**) as the **NIC** (Network Interface Card), where each GPU connects to four 100 Gbps EFA network cards through PCIe Gen5 x16 lanes, as we've seen before.

对于 AWS p5 实例，我们有 Elastic Fabric Adapter （EFA） 作为 NIC（网络接口卡），其中每个 GPU 通过 PCIe Gen100 x5 通道连接到四个 16 Gbps EFA 网卡，正如我们之前看到的那样。

Internode GPU-to-GPU communication path through Libfabric EFA

通过 Libfabric EFA 的节点间 GPU 到 GPU 通信路径

As illustrated above, when GPUs and network cards are connected to the same PCIe switch, **GPUDirect RDMA** enables their communication to occur solely through that switch. This setup allows for full utilization of the PCIe Gen5 x16 bandwidth and avoids

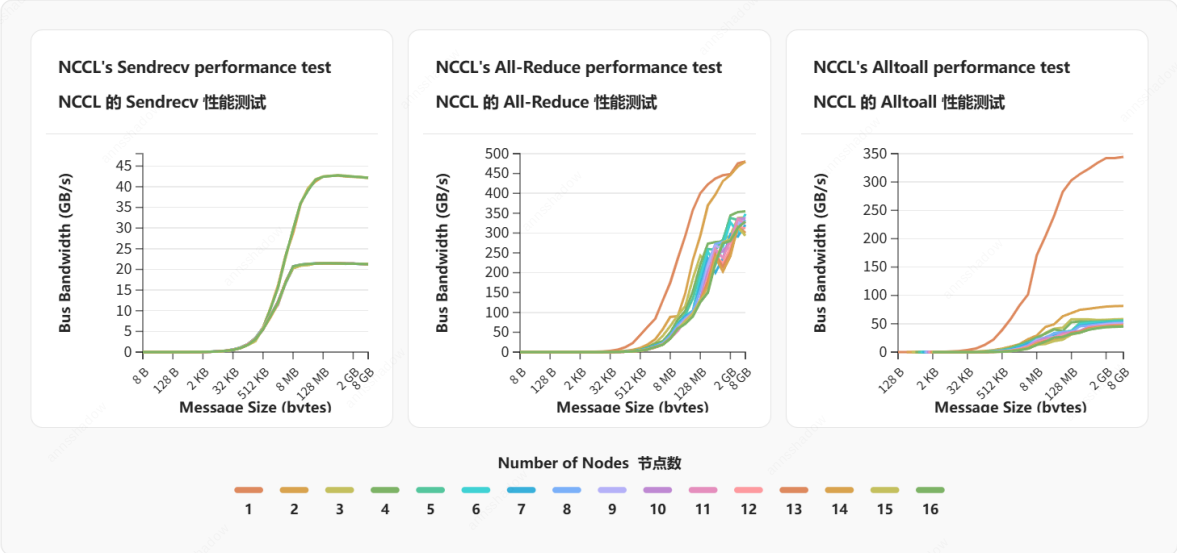
involving other PCIe switches or the CPU memory bus.

如上图所示，当 GPU 和网卡连接到同一 PCIe 交换机时，GPUDirect RDMA 使它们能够仅通过该交换机进行通信。此设置允许充分利用 PCIe Gen5 x16 带宽，并避免涉及其他 PCIe 交换机或 CPU 内存总线。

Theoretically, 8 PCIe Switches per node x 4 EFA NICs per switch x 100 Gbps each EFA NIC gives**3200 Gbps**(400GB/s)** of bandwidth which is the bandwidth we find in AWS p5's specs). So how does it hold in practice? Let's find out by running the same benchmarks as before but across different nodes!

理论上，每个节点 8 个 PCIe 交换机 x 每个交换机 4 个 EFA NIC x 每个 EFA NIC 100 Gbps 提供 3200 Gbps（400GB/s）** 的带宽，这是我们在 AWS p5 规范中找到的带宽）。那么它在实践中是如何成立的呢？让我们通过运行与以前相同的基准测试但跨不同的节点来找出答案！

Bandwidth Analysis 带宽分析



Bandwidth scaling of collective operations across different number of nodes on our AWS p5 instances, using recommendations from aws-samples/awsome-distributed-training.

使用 aws-samples/awsome-distributed-training 的建议，跨 AWS p5 实例上不同数量的节点对集体作进行带宽扩展。

Point-to-point send/receive operations achieve around **42-43 GB/s** for 2-4 nodes but drop to approximately 21 GB/s for 5+ nodes.

对于 2-4 个节点，点对点发送/接收作可实现约 42-43 GB/s，但对于 5+ 节点，速度降至约 21 GB/s。

This performance degradation occurs because NCCL automatically reduces the number of point-to-point channels per peer from 2 to 1 when scaling beyond 4 nodes, effectively halving the available bandwidth utilization, while theoretical maximum remains ~50 GB/s (4 EFA NICs × 12.

出现这种性能下降是因为 NCCL 在扩展超过 4 个节点时会自动将每个对等点的点对点通道数量从 2 个减少到 1 个，从而有效地将可用带宽利用率减半，而理论最大值仍为 ~50 GB/s（4 个 EFA 网卡× 12 个。

5 GB/s each). 每个 5 GB/s）。
We successfully managed to restore the full throughput for this test on 5+ nodes by setting `NCCL_NCHANNELS_PER_NET_PEER=2`, though this flag should be used with caution as it may degrade all-to-all performance for example (see GitHub issue #1272 for details).

我们通过设置 `NCCL_NCHANNELS_PER_NET_PEER=2` 成功地在 5+ 节点上恢复了此测试的全部吞吐量，但应谨慎使用此标志，因为它可能会降低全对全的性能（有关详细信息，请参阅 GitHub 问题 #1272）。

The all-reduce operation demonstrates excellent performance within a single node, achieving **480 GB/s** of bus bandwidth. When scaling to 2 nodes, bandwidth remains nearly identical at 479 GB/s, after which it stabilizes at around 320-350 GB/s for 3-16 nodes.

全缩减作在单个节点内表现出出色的性能，可实现 480 GB/s 的总线带宽。当扩展到 2 个节点时，带宽几乎保持在 479 GB/s 的 479 GB/s，之后 320-350 个节点的带宽稳定在 16-16

crossing node boundaries due to the transition from NVLink to the inter-node network fabric,*the bandwidth then scales almost constantly as we add more nodes.*

这种模式揭示了一个重要特征：虽然由于从 NVLink 过渡到节点间网络结构，跨越节点边界时会出现初始下降，但随着我们添加更多节点，带宽几乎会不断扩展。

> **Scaling All-Reduce Across Nodes**
>
> **跨节点扩展 All-Reduce**
>
> This near-constant scaling behavior beyond 2 nodes is actually quite encouraging for large-scale training.
>
> 这种超过 2 个节点的近乎恒定的扩展行为实际上对于大规模训练来说是相当令人鼓舞的。
>
> The relatively stable 320-350 GB/s across 3-16 nodes suggests that parallelism strategies relying on all-reduce operations (for example, in data parallelism) can scale to hundreds or even thousands of GPUs without significant per-GPU bandwidth degradation.
>
> 3-16 个节点上相对稳定的 320-350 GB/s 表明，依赖全缩减作（例如，数据并行性）的并行性策略可以扩展到数百甚至数千个 GPU，而不会显着降低每个 GPU 的带宽。
>
> This logarithmic scaling characteristic is typical of well-designed multi-tier network topologies using 8-rail optimized fat trees, where each of the 8 GPUs connects to a separate switch rail to maximize bisection bandwidth.
>
> 这种对数缩放特性是使用 8 轨优化胖树的精心设计的多层网络拓扑的典型特征，其中 8 个 GPU 中的每一个都连接到单独的交换机轨，以最大限度地提高平分带宽。
>
> Modern frontier training clusters routinely operate at 100,000+ GPUs, and this stable scaling behavior is what makes such massive deployments feasible.
>
> 现代前沿训练集群通常在 100,000+ 个 GPU 上运行，这种稳定的扩展行为使这种大规模部署成为可能。
>
> When working with different bandwidth links (NVLink within nodes vs. inter-node network), consider adapting your parallelism strategy to each bandwidth tier to fully utilize all available bandwidths. See the Ultrascale playbook for detailed guidance on optimizing parallelism configurations for heterogeneous network topologies.
>
> 使用不同的带宽链路（节点内的 NVLink 与节点间网络）时，请考虑根据每个带宽层调整并行性策略，以充分利用所有可用带宽。有关优化异构网络拓扑的并行度配置的详细指南，请参阅 Ultrascale playbook。

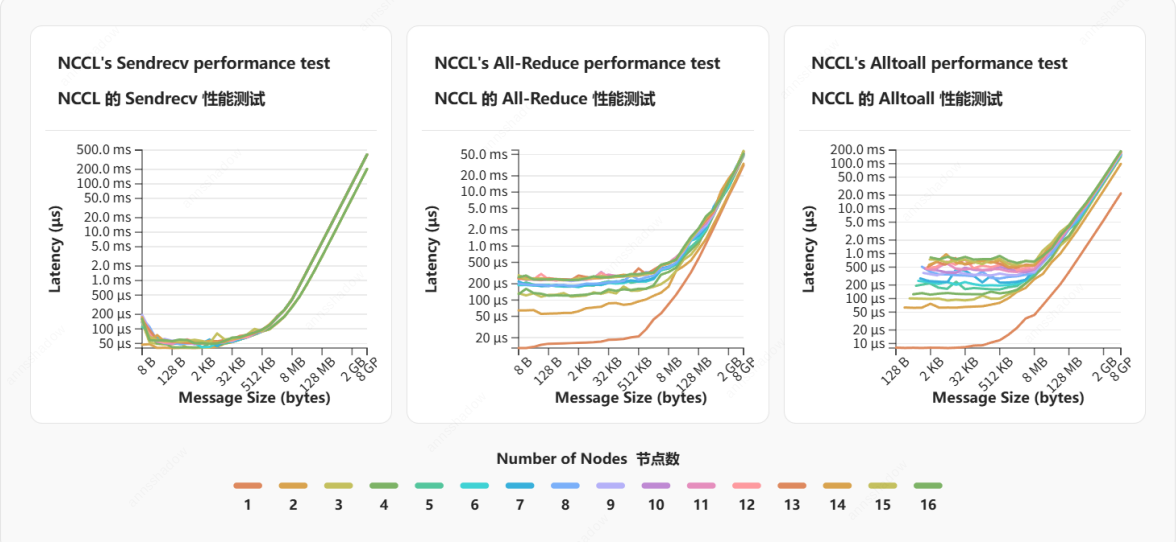
The all-to-all operation shows more dramatic scaling challenges: starting at 344 GB/s for a single node, bandwidth drops to 81 GB/s at 2 nodes and continues declining to approximately 45-58 GB/s for larger clusters.

all-to-all 作显示出更严峻的扩展挑战：从单个节点的 344 GB/s 开始，在 2 个节点的带宽下降到 81 GB/s，对于较大的集群，带宽继续下降到大约 45-58 GB/s。

This steeper degradation reflects the all-to-all pattern's intensive network demands, where each GPU must communicate with every other GPU across nodes, creating significantly more network congestion than all-reduce operations.

这种更陡峭的退化反映了全对全模式的密集网络需求，其中每个 GPU 必须跨节点与所有其他 GPU 通信，从而造成比全缩作更多的网络拥塞。

Latency Analysis 延迟分析



Latency scaling of collective operations across different number of nodes on our AWS p5 instances, using recommendations from [aws-

samples/awsome-distributed-training](https://github.com/aws-samples/awsome-distributed-training/blob/main/micro-benchmarks/nccl-tests/slurm/nccl-tests-container.sbatch).

使用 [aws-samples/awsome-distributed-training]（https://github.com/aws-samples/awsome-distributed-training/blob/main/micro-benchmarks/nccl-tests/slurm/nccl-tests-container.sbatch） 中的建议，跨 AWS p5 实例上不同数量的节点对集体作进行延迟扩展。

Latency measurements reveal the fundamental cost of crossing node boundaries. Send/receive operations maintain relatively stable latencies of **40-53 µs** across all
is primarily determined by the base network round-trip time rather than cluster size, though some variation suggests network topology and routing effects still play a role.

延迟测量揭示了跨越节点边界的基本成本。发送/接收作在所有多节点配置中保持 40-53 µs 的相对稳定的延迟，这表明点对点通信延迟主要由基本网络往返时间而不是集群大小决定，尽管一些变化表明网络拓扑和路由效应仍然发挥作用。

All-reduce operations show minimal latency of **12.9 µs** within a single node, but this jumps to **55.5 µs** for 2 nodes and continues increasing nearly linearly with cluster size, reaching **235 µs** at 16 nodes. This progression reflects both the increased communication distance and the growing complexity of the reduction tree across more nodes.

全缩减作显示单个节点内的最小延迟为 12.9 µs，但对于 2 个节点，延迟会跃升至 55.5 µs，并随着集群大小的增加而几乎线性增加，在 16 个节点上达到 235 µs。这种进展反映了通信距离的增加以及更多节点上简化树的复杂性不断增加。

All-to-all operations exhibit similar trends, starting at **7.6 µs** for single-node communication but climbing to **60 µs** at 2 nodes and reaching **621** µs at 16 nodes. The superlinear growth in latency for all-to-all operations indicates that network congestion and coordination overhead compound as more nodes participate in the collective.

全对全运算也表现出类似的趋势，单节点通信的起点为 7.6 µs，但 2 个节点的速率攀升至 60 µs，16 个节点的速率达到 621 µs。全对全作延迟的超线性增长表明，随着越来越多的节点参与集体，网络拥塞和协调开销会加剧。

> 🚀 **NVSHMEM for Optimized GPU Communication**
>
> **NVSHMEM 用于优化 GPU 通信**
>
> With the rise of Mixture of Experts (MoE) architectures, which require frequent all-to-all communication patterns for expert routing, optimized GPU communication libraries have become increasingly critical.
>
> 随着专家混合 （MoE） 架构的兴起，需要频繁的全对多通信模式来进行专家路由，优化的 GPU 通信库变得越来越重要。
>
> NVSHMEM is gaining significant traction as a high-performance communication library that combines the memory of multiple GPUs into a partitioned global address space (PGAS).
>
> NVSHMEM 作为一种高性能通信库，它将多个 GPU 的内存组合到一个分区的全局地址空间（PGAS） 中，正在获得巨大的关注。
>
> Unlike traditional MPI-based approaches that rely on CPU-orchestrated data transfers, NVSHMEM enables asynchronous, GPU-initiated operations that eliminate CPU-GPU
> Unlike traditional MPI-based approaches that rely on CPU-orchestrated data transfers, NVSHMEM enables asynchronous, GPU-initiated operations that eliminate CPU-GPU synchronization overhead.
>
> 与依赖 CPU 编排数据传输的传统基于 MPI 的方法不同，NVSHMEM 支持异步、GPU 启动的作，从而消除 CPU-GPU 同步开销。
>
> NVSHMEM offers several key advantages for GPU communication: Through technologies like GPUDirect Async, GPUs can bypass the CPU entirely when issuing internode communication, achieving up to 9.5x higher throughput for small messages (<1 KiB).
>
> NVSHMEM 为 GPU 通信提供了几个关键优势：通过 GPUDirect Async 等技术，GPU 在发出节点间通信时可以完全绕过 CPU，从而实现高达 9.5 倍的小消息吞吐量 （<1 KiB）。
>
> This is particularly beneficial for collective operations that require intensive network communication patterns.
>
> 这对于需要密集网络通信模式的集体作特别有利。
>
> The library currently supports InfiniBand/RoCE with Mellanox adapters (CX-4 or later), Slingshot-11 (Libfabric CXI), and Amazon EFA (Libfabric EFA).
>
> 该库目前支持带有 Mellanox 适配器（CX-4 或更高版本）、Slingshot-11 （Libfabric CXI） 和 Amazon EFA （Libfabric EFA） 的 InfiniBand/RoCE。
>
> For applications requiring strong scaling with fine-grain communication, NVSHMEM's

low-overhead, one-sided communication primitives can significantly improve performance compared to traditional CPU-proxy methods.

对于需要通过细粒度通信进行强扩展的应用程序，与传统的 CPU 代理方法相比，NVSHMEM 的低开销、单侧通信原语可以显着提高性能。

Learn more in the NVSHMEM documentation and this detailed blog post on GPUDirect Async.

在 NVSHMEM 文档和这篇关于 GPUDirect Async 的详细博客文章中了解更多信息。

When bandwidth measurements fall short of expectations, several factors could be limiting performance. Understanding these potential bottlenecks is essential for achieving optimal interconnect utilization.

当带宽测量值低于预期时，有几个因素可能会限制性能。了解这些潜在瓶颈对于实现最佳互连利用率至关重要。

TROUBLESHOOTING INTERCONNECT

互连故障排除

following areas:

如果遇到带宽低于预期的情况，请系统地检查以下区域：

Library Versions 库版本

Outdated NCCL, EFA, or CUDA libraries may lack critical performance optimizations or bug fixes.

过时的 NCCL、EFA 或 CUDA 库可能缺乏关键的性能优化或错误修复。

Always verify you're running recent, compatible versions of all communication libraries. e.g. AWS regularly updates their Deep Learning AMIs with optimized library versions for their hardware. It's also recommended to log these library versions for important experiments.

始终验证您正在运行所有通信库的最新兼容版本。例如，AWS 会定期更新其深度学习 AMI，为其硬件提供优化的库版本。还建议记录这些库版本以进行重要实验。

CPU Affinity Configuration

CPU 亲和性配置

Improper CPU affinity settings can significantly impact NCCL performance by causing unnecessary cross-NUMA traffic. Each GPU should be bound to CPUs on the same NUMA node to minimize memory access latency. In practice, this Github issue demonstrates how using `NCCL_IGNORE_CPU_AFFINITY=1` and `--cpu-bind none` helped reduce container latency significantly. You can read more about it here.

不正确的 CPU 关联性设置会导致不必要的跨 NUMA 流量，从而严重影响 NCCL 性能。每个 GPU 都应绑定到同一 NUMA 节点上的 CPU，以最大程度地减少内存访问延迟。在实践中，这个 Github 问题演示了如何使用 `NCCL_IGNORE_CPU_AFFINITY=1` 并 `--cpu-bind none` 帮助显着减少容器延迟。你可以在这里阅读更多关于它的信息。

Network Topology and Placement

网络拓扑和布局

Understanding your network topology is crucial for diagnosing performance issues. Cloud placement groups, while helpful, don't guarantee minimal network hops between instances.

了解网络拓扑对于诊断性能问题至关重要。云置放组虽然有用，但不能保证实例之间的网络跃点最小。

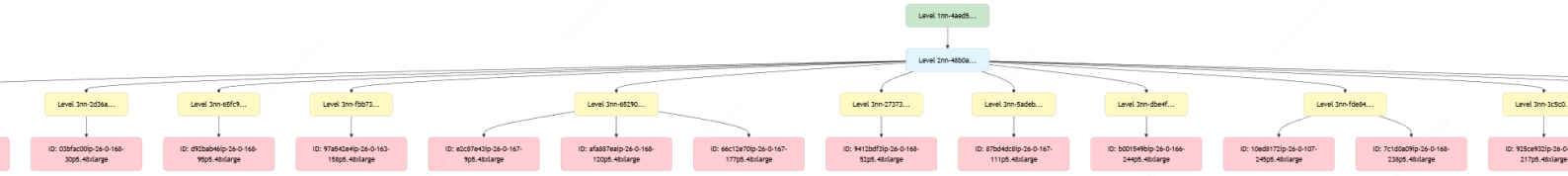
In modern datacenter fat-tree topologies, instances placed under different top-level switches will experience higher latency and potentially lower bandwidth due to additional network hops in the routing path.

In modern datacenter fat-tree topologies, instances placed under different top-level switches will experience higher latency and potentially lower bandwidth due to additional network hops in the routing path.

在新式数据中心胖树拓扑中，由于路由路径中的额外网络跃点，放置在不同顶级交换机下的实例将遇到更高的延迟和潜在的较低带宽。

For **AWS EC2** users, the Instance Topology API provides valuable visibility into network node placement. Instances sharing t"he same network node at the bottom layer (directly connected to the instance) are physically closest and will achieve the lowest latency communication.

对于 AWS EC2 用户，实例拓扑 API 提供了对网络节点放置的宝贵可见性。在底层共享同一网络节点（直接连接到实例）的实例在物理上是最接近的，并且将实现最低延迟的通信。



Network topology visualization showing instance placement.

显示实例放置的网络拓扑可视化。

Minimizing network hops between communicating nodes directly translates to better interconnect performance.

最大限度地减少通信节点之间的网络跃点直接转化为更好的互连性能。

For small-scale experiments and ablations, ensuring your instances are co-located on the same network switch can make a measurable difference in both latency and bandwidth utilization.

对于小规模实验和消融，确保您的实例位于同一网络交换机上可以在延迟和带宽利用率方面产生可衡量的差异。

Correct Environment Variables

正确的环境变量

Missing or incorrect environment variables for your network adapter can severely limit bandwidth utilization.

网络适配器的环境变量缺失或不正确可能会严重限制带宽利用率。

Communication libraries like NCCL rely on specific configuration flags to enable optimal performance features such as adaptive routing, GPU-initiated transfers, and proper buffer sizing.

NCCL 等通信库依靠特定的配置标志来实现最佳性能功能，例如自适应路由、GPU 启动的传输和适当的缓冲区大小调整。

For example, when using AWS EFA (Elastic Fabric Adapter), ensure you're setting the recommended NCCL and EFA environment variables for your instance type. The AWS EFA cheatsheet provides comprehensive guidance on optimal flag configurations for different scenarios.

例如，使用 AWS EFA（Elastic Fabric 适配器）时，请确保为实例类型设置推荐的 NCCL 和 EFA 环境变量。AWS EFA 备忘单提供了有关不同场景的最佳标志配置的全面指导。

Container-specific Considerations

特定于容器的注意事项

When using containers (Docker/Enroot), several configuration steps are critical for optimal NCCL performance:

使用容器 （Docker/Enroot） 时，几个配置步骤对于实现最佳 NCCL 性能至关重要：

- **Shared and Pinned Memory** : Docker containers default to limited shared and pinned memory resources. Launch containers with `-shm-size=1g --ulimit memlock=-1` to prevent initialization failures.

  共享和固定内存： Docker 容器默认为有限的共享和固定内存资源。启动 `-shm-size=1g --ulimit memlock=-1` 容器以防止初始化失败。

- **NUMA Support** : Docker disables NUMA support by default, which can prevent cuMem host allocations from working correctly. Enable NUMA support by invoking Docker with `-cap-add SYS_NICE` .

  NUMA 支持： Docker 默认禁用 NUMA 支持，这可能会阻止 cuMem 主机分配正常工作。通过使用 `-cap-add SYS_NICE` 调用 Docker 来启用 NUMA 支持。

- **PCI Topology Discovery** : Ensure `/sys` is properly mounted to allow NCCL to discover the PCI topology of GPUs and network cards. Having `/sys` expose a virtual PCI topology can result in sub-optimal performance.

PCI 拓扑发现：确保 `/sys` 正确安装，以允许 NCCL 发现 GPU 和网卡的 PCI 拓扑。公开虚拟 PCI 拓扑可能会导致 `/sys` 性能不佳。

> 🤗 **Community Troubleshooting**
>
> **社区故障排除**
>
> We're gathering troubleshooting findings here as a community effort. If you've encountered performance issues or discovered effective debugging methods, please jump to the Discussion Tab and share your experience to help others optimize their interconnect utilization.
>
> 作为社区的努力，我们正在此处收集故障排除结果。如果您遇到性能问题或发现有效的调试方法，请跳转到"讨论"选项卡并分享您的经验，以帮助其他人优化其互连利用率。

Now that you know how to debug bottlenecks in GPU-CPU and GPU-GPU communication let's have a look at a GPU communication part that typically gets less attention, namely communication with the storage layer!

现在您已经知道如何调试 GPU-CPU 和 GPU-GPU 通信中的瓶颈，让我们来看看通常不太受关注的 GPU 通信部分，即与存储层的通信！

GPU-TO-STORAGE GPU 到存储

The connection between GPUs and storage systems is often overlooked but can significantly impact training efficiency.

GPU 和存储系统之间的连接经常被忽视，但会显着影响训练效率。

During training, GPUs need to continuously read data from storage (data loading, especially for multimodal data with large image/video files) and periodically write model states back to storage (aka checkpointing).

在训练过程中，GPU 需要不断从存储中读取数据（数据加载，特别是对于具有大型图像/视频文件的多模态数据），并定期将模型状态写回存储（又名检查点）。

For modern large-scale training runs, these I/O operations can become bottlenecks if not properly optimized.

对于现代大规模训练运行，如果优化不当，这些 I/O 作可能会成为瓶颈。

> **TL;DR:** GPU-storage I/O impacts training through data loading and checkpointing. GPUDirect Storage (GDS) enables direct GPU-to-storage transfers, bypassing CPU for better performance. Even without GDS enabled in our cluster, local NVMe RAID (8×3.5TB drives in RAID 0) delivers 26.
>
> TL;DR：GPU 存储 I/O 通过数据加载和检查点影响训练。GPUDirect Storage （GDS） 支持直接 GPU 到存储传输，绕过 CPU 以获得更好的性能。即使我们的集群中没有启用 GDS，本地 NVMe RAID（RAID 8×3.5 中的 0TB 驱动器）也能提供 26.
>
> 59 GiB/s and 337K IOPS (6.3x faster than network storage), making it ideal for checkpoints.
>
> 59 GiB/s 和 337K IOPS（比网络存储快 6.3 倍），使其成为检查点的理想选择。

Understanding Storage Topology

了解存储拓扑

The physical connection between GPUs and storage devices follows a similar hierarchical structure to GPU interconnects. Storage devices connect through PCIe bridges, and understanding this topology helps explain performance characteristics and potential bottlenecks.

GPU 和存储设备之间的物理连接遵循与 GPU 互连类似的分层结构。存储设备通过 PCIe 桥接器连接，了解这种拓扑有助于解释性能特征和潜在瓶颈。

Looking at the system topology from `lstopo`, we can see how NVMe drives connect to the system. In our p5 instance, we have 1 NVMe SSD per GPU:

查看系统拓扑 `lstopo`，我们可以看到 NVMe 驱动器如何连接到系统。在我们的 p5 实例中，每个 GPU 有 1 个 NVMe SSD：

```
PCIBridge L#13 (busid=0000:46:01.5 id=1d0f:0200 class=0604(PCIBridge) link=15.75(
PCI L#11 (busid=0000:54:00.0 id=1d0f:cd01 class=0108(NVMExp) link=15.75GB/s PCIS
    Block(Disk) L#9 (Size=3710937500 SectorSize=512 LinuxDeviceID=259:2 Model="Am

```

A natural question would be whether GPUs can directly access NVMe drives without involving the CPU. The answer is yes, through **GPUDirect Storage (GDS)** .

一个自然的问题是 GPU 是否可以在不涉及 CPU 的情况下直接访问 NVMe 驱动器。答案是肯定的，通过 GPUDirect Storage （GDS） 。

GPUDirect Storage is part of NVIDIA's GPUDirect family of technologies that enables a direct data path between storage (local NVMe or remote NVMe-oF) and GPU memory.

GPUDirect Storage 是 NVIDIA GPUDirect 系列技术的一部分，可在存储（本地 NVMe 或远程 NVMe-oF）和 GPU 内存之间实现直接数据路径。

It eliminates unnecessary memory copies through CPU bounce buffers by allowing the DMA engine near the storage controller to move data directly into or out of GPU memory.

它允许存储控制器附近的 DMA 引擎将数据直接移入或移出 GPU 内存，从而消除了通过 CPU 反弹缓冲区不必要的内存副本。

This reduces CPU overhead, decreases latency, and significantly improves I/O performance for data-intensive workloads like training on large multimodal datasets.

这减少了 CPU 开销，减少了延迟，并显着提高了数据密集型工作负载（例如在大型多模态数据集上进行训练）的 I/O 性能。

To verify if GPUDirect Storage is properly configured on your system, you can check the GDS configuration file and use the provided diagnostic tools:

要验证您的系统上是否正确配置了 GPUDirect 存储，您可以检查 GDS 配置文件并使用提供的诊断工具：

```
$ /usr/local/cuda/gds/tools/gdscheck.py -p
 =====================
 DRIVER CONFIGURATION:
 =====================
 NVMe               : Supported
 NVMeOF             : Unsupported
 SCSI               : Unsupported
 ScaleFlux CSD      : Unsupported
 NVMesh             : Unsupported
 DDN EXAScaler      : Unsupported
 IBM Spectrum Scale : Unsupported
 NFS                : Unsupported
 BeeGFS             : Unsupported
 WekaFS             : Unsupported
 Userspace RDMA     : Unsupported
 --Mellanox PeerDirect : Enabled
 --rdma library        : Not Loaded (libcufile_rdma.so)
 --rdma devices        : Not configured
 --rdma_device_status  : Up: 0 Down: 0
 =====================

```

We see `NVMe: Supported` which informs that GDS is currently configured to work for NVMe drives, and all other storage types are not properly configured as apparent from the Unsupported flag. If GDS is not properly configured for your storage type, refer to the NVIDIA GPUDirect Storage Configuration Guide for instructions on modifying the configuration file at `/etc/cufile.json` .

我们看到 `NVMe: Supported` 哪个通知 GDS 当前配置为适用于 NVMe 驱动器，并且所有其他存储类型都未正确配置，从不受支持的标志中可以明显看出。如果未为您的存储类型正确配置 GDS，请参阅 NVIDIA GPUDirect 存储配置指南，了解有关修改配置文件的说明，位于

To understand the storage devices available on your system, you can use `lsblk` to display the block device hierarchy:

要了解系统上可用的存储设备，您可以使用 来 `lsblk` 显示块设备层次结构：

```
$ lsblk --fs -M
    NAME        FSTYPE            LABEL                 UUID
...
    nvme0n1
    └─nvme0n1p1 ext4              cloudimg-rootfs       24ec7991-cb5c-4fab-99e5
┌─▶ nvme1n1     linux_raid_member ip-26-0-164-236:MY_RAID d0795631-71f0-37e5-133
├─▶ nvme2n1     linux_raid_member ip-26-0-164-236:MY_RAID d0795631-71f0-37e5-133
├─▶ nvme3n1     linux_raid_member ip-26-0-164-236:MY_RAID d0795631-71f0-37e5-133
├─▶ nvme8n1     linux_raid_member ip-26-0-164-236:MY_RAID d0795631-71f0-37e5-133
├─▶ nvme5n1     linux_raid_member ip-26-0-164-236:MY_RAID d0795631-71f0-37e5-133
├─▶ nvme4n1     linux_raid_member ip-26-0-164-236:MY_RAID d0795631-71f0-37e5-133
├─▶ nvme6n1     linux_raid_member ip-26-0-164-236:MY_RAID d0795631-71f0-37e5-133
└─▶ nvme7n1     linux_raid_member ip-26-0-164-236:MY_RAID d0795631-71f0-37e5-133
  └─md0         xfs                                     dddb6849-e5b5-4828-9034

```

This output shows the block device hierarchy on the system. The key observations:

此输出显示系统上的块设备层次结构。主要观察结果：

- `nvme0n1p1` is the root Amazon EBS filesystem mounted at `/`, using 35% of its full ~300GB capacity

  `nvme0n1p1` 是挂载在 `/` 的根 Amazon EBS 文件系统，使用其全部 ~300GB 容量的 35%

- Eight NVMe drives (`nvme1n1` through `nvme8n1`) are configured as a RAID array named `MY_RAID`

  八个 NVMe 驱动器（ `nvme1n1` 通过 `nvme8n1` ）配置为名为 `MY_RAID`

- The RAID array is exposed as `/dev/md0`, formatted with XFS, and mounted at `/scratch` with 28TB available (8x3.5TB)

  RAID 阵列公开为 `/dev/md0`，使用 XFS 格式化，并以 28TB 的可用容量 （8x3.5TB） 安装 `/scratch`

The arrows (┈▶) indicate that multiple NVMe devices are members of the same RAID array, which then combines into the single `md0` device.

箭头 （┈▶） 表示多个 NVMe 设备是同一 RAID 阵列的成员，然后将其组合成单个 `md0` 设备。

Network Storage 网络存储

Network Storage 网络存储

In addition to local NVMe storage, the system has access to network-attached storage

Amazon Elastic Block Store (EBS) is a high-performance, scalable block storage service designed for use with Amazon EC2 instances.

Amazon Elastic Block Store （EBS） 是一种高性能、可扩展的块存储服务，专为与

In addition to local NVMe storage, the system has access to network-attached storage systems:

除了本地 NVMe 存储之外，系统还可以访问网络附加存储系统：

```
$ df -h
Filesystem                                      Size  Used Avail Use% Mounted
/dev/root                                       291G  101G  190G  35% /
weka-hopper.hpc.internal.huggingface.tech/default  393T  263T  131T  67% /fsx
10.53.83.155@tcp:/fg7ntbev                      4.5T  2.9T  1.7T  63% /admin
/dev/md0                                         28T  206G   28T   1% /scratch
```

This output shows: 此输出显示：

- `/dev/root` (291GB Amazon EBS) is the root filesystem at 35% capacity

  `/dev/root` （291GB Amazon EBS） 是容量为 35% 的根文件系统

- `/fsx` (393TB WekaFS) is 67% full with 131TB available

  `/fsx` （393TB WekaFS） 已装满 67%，可用容量为 131TB

- `/admin` (4.5TB FSx Lustre) is 63% full with 1.7TB available

  `/admin` （4.5TB FSx Lustre） 已满 63%，可用 1.7TB

- `/dev/md0` (28TB local NVMe RAID) is only 1% full with 28TB available at `/scratch` . This is our 8×3.5TB SSD NVMe instance store drives in RAID.

  `/dev/md0` （28TB 本地 NVMe RAID）仅满 1%，28TB 可用 `/scratch` 。这是我们在 RAID 中的 8×3.5TB SSD NVMe 实例存储驱动器。

The local NVMe RAID array (`/scratch`) provides the fastest I/O performance, while the network filesystems offer larger capacity for shared data storage.

本地 NVMe RAID 阵列 （ `/scratch` ） 提供最快的 I/O 性能，而网络文件系统则为共享数据存储提供更大的容量。

> 💾 **Storage Technologies 存储技术**
>
> **RAID (Redundant Array of Independent Disks)**: Combines multiple drives to improve performance and/or reliability through data striping, parity, or mirroring.
>
> RAID（独立磁盘冗余阵列）：组合多个驱动器，通过数据条带化、奇偶校验或镜像提高性能和/或可靠性。
>
> **NVMe (Non-Volatile Memory Express)**: High-performance storage protocol for SSDs that connects directly to PCIe, delivering higher throughput and lower latency than SATA/SAS.
>
> NVMe（非易失性内存 Express）：用于 SSD 的高性能存储协议，直接连接到 PCIe，提供比 SATA/SAS 更高的吞吐量和更低的延迟。
>
> **WekaFS**: A high-performance parallel file system designed for AI/ML workloads, providing low-latency access and high throughput across multiple nodes.
>
> WekaFS：专为 AI/ML 工作负载设计的高性能并行文件系统，提供跨多个节点的低延迟访问和高吞吐量。
>
> **FSx Lustre**: A parallel file system designed for HPC that separates metadata and data services across different servers to enable parallel access. While effective for large files, it can struggle with metadata-intensive AI/ML workloads involving many small files.
>
> FSx Lustre：专为 HPC 设计的并行文件系统，可将元数据和数据服务分开，以实现并行访问。虽然它对大文件有效，但它可能会难以处理涉及许多小文件的元数据密集型 AI/ML 工作负载。

Benchmarking Storage Bandwidth

基准测试存储带宽

To understand the performance characteristics of each storage system, we can benchmark their read/write speeds using GPUDirect Storage (GDS). Here's a comprehensive parametric benchmark script that tests various configurations:

为了了解每个存储系统的性能特征，我们可以使用 GPUDirect Storage （GDS） 对它们的读/写速度进行基准测试。这是一个全面的参数基准测试脚本，用于测试各种配置：

Amazon EC2 实例配合使用而设计。

Note that `/fsx` isn't actually Amazon FSx, but WekaFS. We kept the same mount point name for convenience when we migrated from FSx to WekaFS.

请注意，这 `/fsx` 实际上不是 Amazon FSx，而是 WekaFS。为了方便起见，我们在从 FSx 迁移到 WekaFS 时保留了相同的挂载点名称。

```
gdsio -f /<disk_path>/gds_test.dat -d 0 -w <n_threads> -s 10G -i <io_size> -x 1 -
```

The benchmark evaluates storage system performance across Throughput, Latency, IOPS but also:

该基准测试评估吞吐量、延迟、IOPS 方面的存储系统性能，但还评估：

Scalability : How performance changes with different thread counts and I/O sizes. This helps identify the optimal configuration for your workload.

可扩展性：性能如何随不同的线程数和 I/O 大小而变化。这有助于为您的工作负载确定最佳配置：

- Small I/O sizes (64K to 256K) typically maximize IOPS but may not saturate bandwidth

  小 I/O 大小（64K 到 256K）通常可以最大化 IOPS，但可能不会使带宽饱和

- Large I/O sizes (2M to 8M) typically maximize throughput but reduce IOPS

  大型 I/O 大小（2M 到 8M）通常可最大限度地提高吞吐量，但会降低 IOPS

- Thread count affects both: more threads can increase total IOPS and throughput up to hardware limits

  线程数会影响两者：更多线程可以将总 IOPS 和吞吐量提高到硬件限制

Transfer Method Efficiency : Comparing GPU_DIRECT vs CPU_GPU vs CPUONLY shows the benefit of bypassing CPU memory:

传输方法效率：比较 GPU_DIRECT、CPU_GPU 和 CPUONLY 显示了绕过 CPU 内存的好处：

- **GPU_DIRECT** : Uses RDMA to transfer data directly to GPU memory, bypassing CPU entirely (lowest latency, highest efficiency, best IOPS for small operations)

  GPU_DIRECT：使用 RDMA 将数据直接传输到 GPU 内存，完全绕过 CPU（最低延迟、最高效率、小型作的最佳 IOPS）

- **CPU_GPU** : Traditional path where data goes to CPU memory first, then copied to GPU (adds CPU overhead and memory bandwidth contention, reduces effective IOPS)

  CPU_GPU：数据先进入 CPU 内存，然后复制到 GPU 的传统路径（增加 CPU 开销和内存带宽争用，降低有效 IOPS）

- **CPUONLY** : Baseline CPU-only I/O without GPU involvement

  仅 CPU：基线仅 CPU I/O，不涉及 GPU

> **IOPS (I/O Operations Per Second)**
>
> **IOPS（每秒 I/O 作数）**
>
> IOPS is the number of individual I/O operations completed per second. Calculated as `ops / total_time` from the gdsio output. IOPS is particularly important for:
>
> IOPS 是每秒完成的单个 I/O 作数。根据 gdsio 输出计算。 `ops / total_time` IOPS 对于以下情况尤为重要：
>
> IOPS 是每秒完成的单个 I/O 作数。根据 gdsio 输出计算。 `ops / total_time` IOPS 对于以下情况尤为重要：
>
> - Random access patterns with small I/O sizes
>
>   具有小 I/O 大小的随机访问模式
>
> - Workloads with many small files or scattered data access
>
>   具有许多小文件或分散数据访问的工作负载
>
> - Database-like operations where latency per operation matters more than raw bandwidth
>
>   类似数据库的作，其中每个作的延迟比原始带宽更重要
>
> - Higher IOPS indicates better ability to handle concurrent, fine-grained data access
>
>   IOPS 越高，表示处理并发细粒度数据访问的能力越好

/scratch - Bandwidth (GiB/s)

/root - Bandwidth (GiB/s)

Benchmark results comparing storage system performance across varying thread counts and I/O sizes. The heatmaps visualize throughput (GiB/s) and IOPS patterns, revealing optimal configurations for each storage tier.

比较不同线程数和 I/O 大小的存储系统性能的基准测试结果。热图可视化吞吐量 （GiB/s） 和 IOPS 模式，揭示每个存储层的最佳配置。

Note: GPUDirect Storage (GDS) is not currently supported in this cluster configuration.

注意：此集群配置当前不支持 GPUDirect Storage （GDS）。

The benchmarks reveal dramatic performance differences across our four storage systems:

基准测试显示了我们四个存储系统之间的巨大性能差异：

/scratch (Local NVMe RAID) dominates with **26.59 GiB/s** throughput and **337K IOPS** , making it 6.3× faster than FSx for throughput and 6.6× better for IOPS. This local RAID array of 8×3.

/scratch（本地 NVMe RAID）以 26.59 GiB/s 的吞吐量和 337K IOPS 占据主导地位，使其在吞吐量方面比 FSx 快 6.3×，在 IOPS 方面比 FSx 快 6.6×。这个 8×3 的本地 RAID 阵列。

5TB NVMe drives delivers the lowest latency (190µs at peak IOPS) and scales exceptionally well with thread count, achieving peak performance at 64 threads with 1M I/O sizes for throughput.

5TB NVMe 驱动器提供最低的延迟（峰值 IOPS 时为 190µs），并且随线程数的扩展性非常好，在 64 个线程和 1M I/O 大小的吞吐量下实现峰值性能。

/fsx (WekaFS) provides solid network storage performance at **4.21 GiB/s** and **51K IOPS** , making it the best choice for shared data that needs reasonable performance. FSx achieves its best throughput (4.21 GiB/s) using CPUONLY transfer, while its best IOPS (51K) uses GPUD transfer type.

/fsx （WekaFS） 提供 4.21 GiB/s 和 51K IOPS 的稳定网络存储性能，使其成为需要合理性能的共享数据的最佳选择。FSx 仅使用 CPU 传输实现最佳吞吐量 （4.21 GiB/s），而其最佳 IOPS （51K） 使用 GPUD 传输类型。

/admin (FSx Lustre) and **/root (EBS)** filesystems show similar modest performance around **1.1 GiB/s** throughput, but differ significantly in IOPS capability. Admin achieves its peak throughput (1.13 GiB/s) with GPUD transfer and peaks at 17K IOPS with CPU_GPU transfer (24× better than Root), making it more suitable for workloads with many small operations.

/admin （FSx Lustre） 和 /root （EBS） 文件系统在 1.1 GiB/s 吞吐量左右表现出相似的适度性能，但在 IOPS 能力方面存在显着差异。Admin 通过 GPUD 传输实现其峰值吞吐量 （1.13 GiB/s），在 CPU_GPU 传输时达到 17K IOPS 的峰值（比 Root 好 24×），使其更适

合具有许多小作的工作负载。

Root's poor IOPS performance (730) confirms it's best suited for large sequential operations only.

Root 较差的 IOPS 性能 （730） 证实它最适合仅用于大型顺序作。

> **Note on GPU_DIRECT Results:** GPUDirect Storage (GDS) is not currently enabled in our cluster, which explains why GPUD results for NVMe storage (Scratch and Root) underperform compared to CPUONLY transfers.
>
> 关于 GPU_DIRECT 结果的说明：我们的集群中当前未启用 GPUDirect 存储 （GDS），这解释了为什么与仅 CPU 传输相比，NVMe 存储（暂存和根）的 GPUD 结果表现不佳。
>
> With GDS properly configured, we would expect GPUD to show significant advantages for direct GPU-to-storage transfers, particularly for the high-performance NVMe arrays.
>
> 正确配置 GDS 后，我们预计 GPUD 将在直接 GPU 到存储传输方面显示出显着的优势，特别是对于高性能 NVMe 阵列。

Optimal Configuration Patterns : Across all storage types, maximum throughput occurs at 1M I/O sizes, while maximum IOPS occurs at the smallest tested size (64K). This classic tradeoff means choosing between raw bandwidth (large I/O) and operation concurrency (small I/O) based on workload characteristics.

最佳配置模式：在所有存储类型中，最大吞吐量出现在 1M I/O 大小时，而最大 IOPS 出现在最小测试大小 （64K） 时。这种经典的权衡意味着根据工作负载特征在原始带宽（大 I/O）和作并发（小 I/O）之间进行选择。

For ML training with large checkpoint files, the 1M-8M range on Scratch provides optimal performance.

对于具有大型检查点文件的 ML 训练，Scratch 上的 1M-8M 范围可提供最佳性能。

SUMMARY 总结

training infrastructure. But here's the key insight we hope you take home: **identifying bottlenecks is what separates theoretical knowledge from practical optimization** .

如果您已经走到了这一步，那么恭喜您！您现在已经全面了解了存储层次结构以及不同组件在我们的训练基础架构中如何交互。但这是我们希望您带回家的关键见解：识别瓶颈是理论知识与实际优化的区别。

Throughout this guide, we've measured actual bandwidths at every level of the stack: HBM3's 3TB/s within a single GPU, NVLink's 786 GB/s between GPUs in a node, PCIe Gen4 x8's 14.

在本指南中，我们测量了堆栈每个级别的实际带宽：HBM3 在单个 GPU 内的 3TB/s、NVLink 在节点中 GPU 之间的 786 GB/s、PCIe Gen4 x8 的 14。

2 GB/s for CPU-GPU transfers, inter-node network's 42 GB/s for point-to-point communication, and storage systems ranging from 26.59 GB/s (local NVMe) down to 1.1 GB/s (shared filesystems).

CPU-GPU 传输速度为 2 GB/s，节点间网络的点对点通信速度为 42 GB/s，存储系统速度范围从 26.59 GB/s（本地 NVMe）到 1.1 GB/s（共享文件系统）。

These measurements reveal where your training pipeline will slow down and are essential for achieving high Model FLOPs Utilization (MFU).

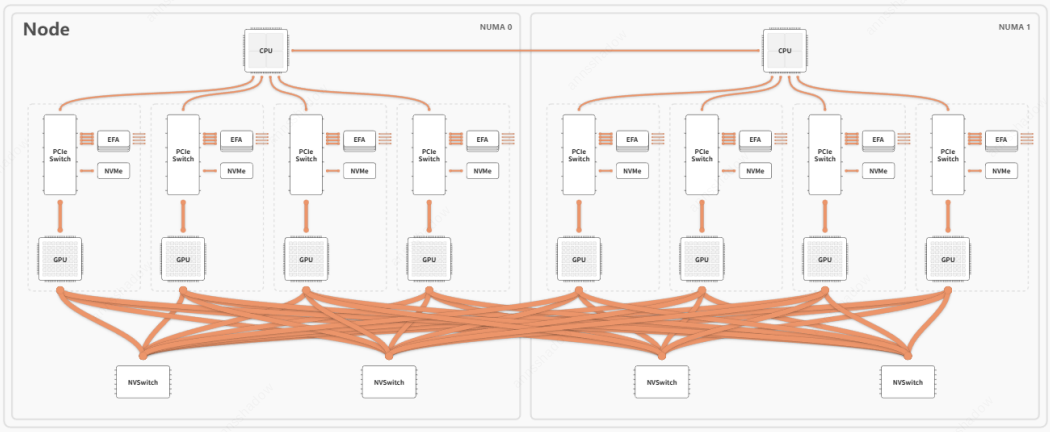
这些测量揭示了您的训练管道在哪些方面会变慢，这对于实现高模型 FLOP 利用率 （MFU） 至关重要。

However, raw bandwidth numbers alone don't tell the complete story. Modern training systems can **overlap computation with communication** , effectively hiding communication costs behind compute operations. This parallelization helps alleviate the bottleneck even when interconnects is slow. For detailed strategies on overlapping compute and communication to maximize throughput, see the Ultra-Scale Playbook.

然而，仅靠原始带宽数字并不能说明完整的情况。现代训练系统可以将计算与通信重叠，从而有效地将通信成本隐藏在计算作后面。这种并行化有助于缓解瓶颈，即使互连速度很慢。有关重叠计算和通信以最大化吞吐量的详细策略，请参阅超大规模 Playbook。

The diagram below synthesizes all our benchmarked measurements into a single view, showing how bandwidth decreases dramatically as we move further from the GPU:

下图将我们所有的基准测量结果综合到一个视图中，显示了随着我们远离 GPU 的距离，带宽如何急剧下降：



Now that we know how to identify bottlenecks in our hardware and software setup, lets see how we can go one step further and ensure we have a resilient system that can run stable for months.

现在我们知道如何识别硬件和软件设置中的瓶颈，让我们看看如何更进一步，确保我们拥有一个可以稳定运行数月的弹性系统。

Building Resilient Training Systems
建立有弹性的培训系统

Having fast hardware is just the entry ticket to having good and stable infrastructure for LLM training.

拥有快速的硬件只是拥有良好且稳定的 LLM 培训基础设施的入场券。

To go from a training amateur to a professional, we need to think beyond raw speed and focus on the less glamorous but critical infrastructure pieces that make the entire training experience smoother and with minimal downtime.

要从培训业余爱好者成为专业人士，我们需要超越原始速度，专注于不那么光鲜亮丽但关键的基础设施部分，使整个培训体验更加顺畅，停机时间最短。

In this section, we shift from hardware & software optimization to **production readiness** : building systems **robust** enough to survive inevitable failures, **automated** enough to run without constant babysitting, and **flexible** enough to adapt when things go wrong.

在本节中，我们将从硬件和软件优化转向生产准备：构建足够强大的系统以承受不可避免的故障，足够自动化以无需持续的保姆即可运行，并且足够灵活以在出现问题时进行调整。

NODE HEALTH MONITORING AND REPLACEMENT
节点健康监控和更换

Having enough fast GPUs is important for training, but since LLM trainings run for weeks or months rather than single days, tracking GPU health over time becomes critical.

拥有足够的快速 GPU 对于训练很重要，但由于 LLM 训练持续数周或数月而不是单天，因此随着时间的推移跟踪 GPU 运行状况变得至关重要。

GPUs that pass initial benchmarks can develop thermal throttling, memory errors, or performance degradation during extended training runs. In this section, we will share how we approach this challenge and the tools we use.

通过初始基准测试的 GPU 可能会在延长训练运行期间出现热节流、内存错误或性能下降。在本节中，我们将分享我们如何应对这一挑战以及我们使用的工具。

Upfront tests: Before launching SmolLM3, we ran comprehensive GPU diagnostics using multiple tools. We used GPU Fryer, an internal tool that stress-tests GPUs for thermal throttling, memory errors, and performance anomalies. We also ran NVIDIA's DCGM diagnostics, a widely-used tool for validating GPU hardware, monitoring performance, and identifying root causes of failures or power anomalies through deep diagnostic tests covering compute, PCIe connectivity, memory integrity, and thermal stability.

前期测试：在启动 SmolLM3 之前，我们使用多种工具运行了全面的 GPU 诊断。我们使用了 GPU Fryer，这是一种内部工具，可以对 GPU 的热节流、内存错误和性能异常进行压力测试。我们还运行了 NVIDIA 的 DCGM 诊断，这是一种广泛使用的工具，用于验证 GPU 硬件、监控性能，并通过涵盖计算、PCIe 连接、内存完整性和热稳定性的深度诊断测试来识别故障或电源异常的根本原因。

These upfront tests caught two problematic GPUs that would have caused issues during training.

这些前期测试发现了两个有问题的 GPU，这些 GPU 可能会在训练过程中引起问题。

You can see in the following table what can be tested with the DCGM diagnostic tool:

您可以在下表中看到可以使用 DCGM 诊断工具测试的内容：

Test Level 测试级别	Duration 期间	Software 软件	PCIe + NVLink	GPU
r1 (Short) r1（短）	Seconds 秒	✓	✓	✓
r2 (Medium) r2（中）	< 2 mins < 2 分钟	✓	✓	✓
r3 (Long) r3（长）	< 30 mins < 30 分钟	✓	✓	✓
r4 (Extra Long) r4（超长）	1-2 hours 1-2 小时	✓	✓	✓

DCGM diagnostic run levels. Source: NVIDIA DCGM Documentation

DCGM 诊断运行级别。来源：NVIDIA DCGM 文档

```
$ dcgmi diag -r 2 -v -d VERB
Successfully ran diagnostic for group.
+---------------------------+------------------------------------------------+
| Diagnostic | Result |
+===========================+================================================+
| -----  Metadata  ----------+------------------------------------------------ |
| DCGM Version | 3.3.1 |
| Driver Version Detected | 575.57.08 |
| GPU Device IDs Detected | 2330,2330,2330,2330,2330,2330,2330,2330 |
| -----  Deployment  --------+------------------------------------------------ |
| Denylist | Pass |
| NVML Library | Pass |
| CUDA Main Library | Pass |
| Permissions and OS Blocks | Pass |
| Persistence Mode | Pass |
| Environment Variables | Pass |
| Page Retirement/Row Remap | Pass |
| Graphics Processes | Pass |
| Inforom | Pass |

+-----  Integration  -------+------------------------------------------------+
| PCIe | Pass - All |
| Info | GPU 0 GPU to Host bandwidth:  14.26 GB/s, GPU |
| 0 Host to GPU bandwidth:  8.66 GB/s, GPU 0 b |
| idirectional bandwidth: 10.91 GB/s, GPU 0 GPU |
| to Host latency:  2.085 us, GPU 0 Host to GP |
| U latency:  2.484 us, GPU 0 bidirectional lat |
| ency:  3.813 us |

...
+-----  Hardware  ----------+------------------------------------------------+
| GPU Memory | Pass - All |
| Info | GPU 0 Allocated 83892938283 bytes (98.4%) |
| Info | GPU 1 Allocated 83892938283 bytes (98.4%) |
| Info | GPU 2 Allocated 83892938283 bytes (98.4%) |
| Info | GPU 3 Allocated 83892938283 bytes (98.4%) |
| Info | GPU 4 Allocated 83892938283 bytes (98.4%) |
| Info | GPU 5 Allocated 83892938283 bytes (98.4%) |
| Info | GPU 6 Allocated 83892938283 bytes (98.4%) |
| Info | GPU 7 Allocated 83892938283 bytes (98.4%) |

+-----  Stress  -----------+------------------------------------------------+


```

Node reservation: Because SmolLM3 was trained on a Slurm managed cluster, we booked a fixed 48-node reservation for the entire run. This setup allowed us to track

the health and performance of the exact same nodes over time, it was also necessary to solve a data storage issues we discussed.

节点预留：由于 SmolLM3 是在 Slurm 托管集群上训练的，因此我们在整个运行中预订了固定的 48 个节点预留。这种设置使我们能够随着时间的推移跟踪完全相同的节点的运行状况和性能，这也是解决我们讨论的数据存储问题的必要条件。

We also reserved a spare node (like a car's spare wheel) so if one failed, we could swap it in immediately without waiting for repairs. While idle, the spare node ran eval jobs or dev experiments.

我们还预留了一个备用节点（如汽车的备胎），因此如果一个节点出现故障，我们可以立即将其更换，而无需等待维修。空闲时，备用节点运行评估作业或开发试验。

Continuous monitoring: During training, we tracked key metrics across all nodes such as GPU temperatures, memory usage, compute utilization and throughput fluctuations. We use Prometheus to collect DCGM metrics from all GPUs and visualize them in Grafana dashboards for real-time monitoring. For detailed setup instructions on deploying Prometheus and Grafana for GPU monitoring on AWS infrastructure, see this example setup guide. A Slack bot alerted us when any node showed suspicious behavior, allowing us to proactively replace failing hardware before it crashed the entire training run.

持续监控：在训练过程中，我们跟踪了所有节点的关键指标，例如 GPU 温度、内存使用情况、计算利用率和吞吐量波动。我们使用 Prometheus 从所有 GPU 收集 DCGM 指标，并在 Grafana 仪表板中可视化以进行实时监控。有关在 AWS 基础设施上部署 Prometheus 和 Grafana 以进行 GPU 监控的详细设置说明，请参阅此示例设置指南。当任何节点表现出可疑
Grafana 仪表板中可视化以进行实时监控。有关在 AWS 基础设施上部署 Prometheus 和 Grafana 以进行 GPU 监控的详细设置说明，请参阅此示例设置指南。当任何节点表现出可疑行为时，Slack 机器人会向我们发出警报，使我们能够在故障硬件导致整个训练运行崩溃之前主动更换故障硬件。

Access to dashboard This multi-layered approach meant hardware issues became manageable interruptions.

访问仪表板这种多层方法意味着硬件问题变成了可管理的中断。

Thermal Reality Check: When GPUs Slow Down

热现实检查：当 GPU 变慢时

Marketing specs assume perfect cooling, but reality is messier. GPUs automatically reduce clock speeds when they overheat, cutting performance below theoretical maximums even in well-designed systems.

营销规范假设完美冷却，但现实更加混乱。GPU 在过热时会自动降低时钟速度，即使在设计良好的系统中，性能也会降低到理论最大值以下。

This Grafana dashboard shows thermal throttling events across our GPU cluster. The bars in the bottom panel indicate when GPUs automatically reduced clock speeds due to overheating.

此 Grafana 仪表板显示了整个 GPU 集群中的热限制事件。底部面板中的条形图指示 GPU 何时因过热而自动降低时钟速度。

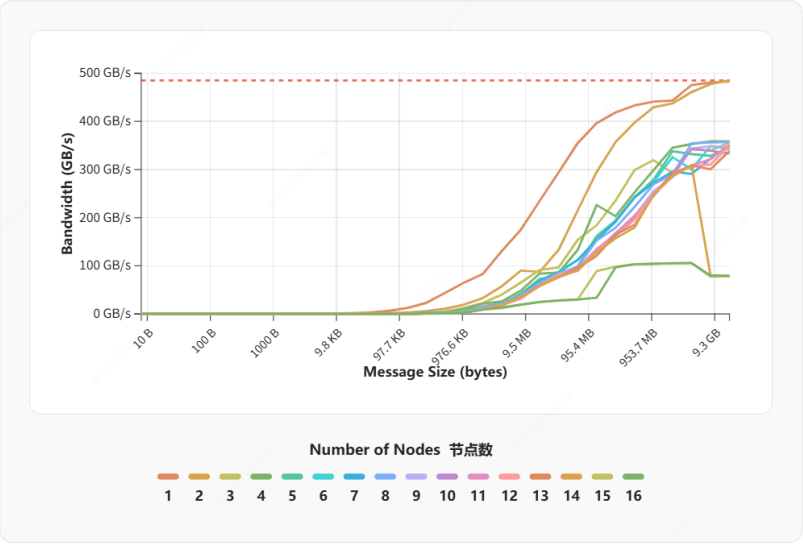
We monitor the `DCGM_FI_DEV_CLOCK_THROTTLE_REASONS` metric from NVIDIA's DCGM to detect thermal throttling. When this metric shows non-zero values, the GPUs are automatically reducing clock speeds due to overheating. The dashboard above shows how these throttling events manifest in practice.

我们监控 NVIDIA DCGM 的 `DCGM_FI_DEV_CLOCK_THROTTLE_REASONS` 指标以检测热节流。当此指标显示非零值时，GPU 会因过热而自动降低时钟速度。上面的仪表板显示了这些限制事件在实践中是如何体现的。

Thermal throttling doesn't just hurt the affected GPU; it cascades across your entire

distributed training setup. During our testing, we observed how a single throttling node can dramatically impact collective communication performance.

热节流不仅会伤害受影响的 GPU，还会伤害受影响的 GPU。它会在整个分布式训练设置中级联。在我们的测试过程中，我们观察到单个节流节点如何显着影响集体通信性能。



AllReduce bandwidth degradation across nodes during our stress testing. The sharp drop after 14 nodes (from 350 GB/s to 100 GB/s) was caused by a single thermally throttled GPU, demonstrating how one slow node can bottleneck the entire distributed training pipeline.

AllReduce 在压力测试期间跨节点的带宽下降。14 个节点后的急剧下降（从 350 GB/s 到 100 GB/s）是由单个热节流 GPU 引起的，这表明一个缓慢的节点如何成为整个分布式训练管道的瓶颈。

The chart above shows AllReduce bandwidth degrading as we scale from 1 to 16 nodes. Notice the sharp drop after 14 nodes, from **350 GB/s** to **100 GB/s** while we expect the bandwidth to stay above 300GB/s as we've seen before. This wasn't a network issue: a single node with thermal throttling became the bottleneck, forcing all other nodes to wait during gradient synchronization.

上图显示了当我们从 1 个节点扩展到 16 个节点时 AllReduce 带宽下降。请注意，在 14 个
如我们之前看到的那样。这不是网络问题：具有热节流的单个节点成为瓶颈，迫使所有其他节点在梯度同步期间等待。

In distributed training, you're only as fast as your slowest node.

在分布式训练中，您的速度取决于最慢的节点。

👉 **Key lesson:** Before committing to long training runs, stress-test your hardware using the tools mentioned earlier to identify thermal and power limitations. Monitor temperatures continuously using DCGM telemetry and plan for real-world thermal limits.

👉 重要教训：在进行长时间训练运行之前，请使用前面提到的工具对硬件进行压力测试，以确定热和功率限制。使用 DCGM 遥测技术持续监控温度，并规划实际的热限制。

It's also good practice to verify that GPU clocks are set to maximum performance.

验证 GPU 时钟是否设置为最高性能也是一个很好的做法。

For a deeper dive into why GPUs can't sustain their advertised performance due to power constraints, seethis excellent analysis on power throttling.

要更深入地了解为什么 GPU 由于功率限制而无法维持其宣传的性能，请参阅这篇关于功率限制的出色分析。

CHECKPOINT MANAGEMENT 检查点管理

Checkpoints are our safety net during long training runs. We save them regularly for three practical reasons: recovering from failures, monitoring training progress through evaluation and sharing intermediate models with the community for research.

检查站是我们在长时间训练中的安全网。我们定期保存它们有三个实际原因：从故障中恢复、通过评估监控训练进度以及与社区共享中间模型进行研究。

The recovery aspect matters most. If our run fails, we want to restart from the latest saved checkpoint so we lose at most the save interval if we resume immediately (e.g., 4 hours of training if we save every 4 hours).

恢复方面最重要。如果运行失败，我们希望从最新保存的检查点重新启动，因此如果立即恢复，我们最多会丢失保存间隔（例如，如果我们每 4 小时保存一次，则最多会丢失 4 小时的训练）。

Automate your resume process

> 💡 **自动化您的简历流程**
>
> Try to automate your resume process. On Slurm, for example, you can you can just use `SBATCH --requeue` so the job restarts automatically from the latest checkpoint. That way, you avoid losing time waiting for someone to notice the failure and manually restart.
> **自动化您的简历流程**
>
> Try to automate your resume process. On Slurm, for example, you can you can just use `SBATCH --requeue` so the job restarts automatically from the latest checkpoint. That way, you avoid losing time waiting for someone to notice the failure and manually restart.
>
> 尝试自动化您的简历流程。例如，在 Slurm 上，您可以使用 `SBATCH --requeue` 作业从最新的检查点自动重新启动。这样，您就可以避免浪费等待有人注意到故障并手动重新启动的时间。

There's two important details to keep in mind when implementing your resume mechanism:

实施简历机制时需要牢记两个重要细节：

- Checkpoint saving should happen in the background without impacting training throughput.

  检查点保存应在后台进行，而不会影响训练吞吐量。

- Watch your storage, over a 24-day run, saving every 4 hours means ~144 checkpoints. With large models and optimizer states, this adds up fast.

  在 24 天的运行中监视您的存储，每 4 小时保存一次意味着 ~144 个检查点。对于大型模型和优化器状态，这会快速增加。

  In our case, we store only one local checkpoint (the latest saved) at a time and offload the rest to S3 to avoid filling up cluster storage.

  在我们的例子中，我们一次只存储一个本地检查点（最新保存的），并将其余的检查点卸载到 S3，以避免集群存储被填满。

A painful lesson from the past:

过去的惨痛教训：

During our first large-scale run (StarCoder 15B), training proceeded smoothly through multiple restarts. On the final day, we discovered the entire checkpoint folder had been deleted by a leftover `rm -rf $CHECKPOINT_PATH` command at the very end of the script from old throughput tests. This destructive command only triggered when the Slurm job actually finished, which hadn't happened in previous restarts.

在我们的第一次大规模运行（StarCoder 15B）中，训练通过多次重启顺利进行。在最后一天，我们发现整个检查点文件夹已被旧吞吐量测试中脚本末尾的剩余 `rm -rf $CHECKPOINT_PATH` 命令删除。只有当 Slurm 作业实际完成时才会触发此破坏性命令，这在之前的重启中没有发生过。

Luckily, we had the checkpoint from the day before saved, so it only cost us one day of retraining.

幸运的是，我们保存了前一天的检查站，所以我们只花了一天的再培训时间。

The takeaways were clear: never leave destructive commands in production scripts, and automate checkpoint backups immediately after saving rather than relying on manual intervention.

要点很明确：永远不要在生产脚本中留下破坏性命令，并在保存后立即自动执行检查点备份，而不是依赖人工干预。

In our nanotron trainings, we save checkpoints every 2 hours locally, immediately upload each one to S3, then delete the local copy once backup is confirmed. On resume, we pull from S3 if the latest checkpoint isn't available locally.

在我们的 nanotron 培训中，我们每 2 小时在本地保存一次检查点，立即将每个检查点上传到 S3，然后在确认备份后删除本地副本。在恢复时，如果最新的检查点在本地不可用，我们会从 S3 中提取。

This approach saves storage, ensures backups, and enables quick recovery.

这种方法可以节省存储空间、确保备份并实现快速恢复。

AUTOMATED EVALUATIONS 自动评估

Running evaluations manually becomes a bottleneck fast. They look simple until you're doing them repeatedly. Running benchmarks, tracking and plotting results for every run adds up to significant overhead. The solution? Automate everything upfront.

手动运行评估很快就会成为瓶颈。它们看起来很简单，直到你反复做它们。每次运行的运行基

准、跟踪和绘制结果都会产生巨大的开销。解决方案？预先自动化一切。

For SmolLM3, we use LightEval to run evaluations on nanotron checkpoints. Every saved checkpoint triggers an evaluation job on the cluster. The results are pushed directly to Weights & Biases or Trackio, so we just open the dashboard and watch the curves evolve. This saved us a huge amount of time and kept eval tracking consistent throughout the run.

对于 SmolLM3，我们使用 LightEval 对纳米电子检查点进行评估。每个保存的检查点都会在集群上触发一个评估作业。结果直接推送到 Weights & Biases 或 Trackio，因此我们只需打开仪表板并观察曲线的演变。这为我们节省了大量时间，并在整个运行过程中保持了评估跟踪的一致性。

If you can automate only one thing in your training setup, automate evaluations.

如果训练设置中只能自动执行一件事，请自动执行评估。

Finally, lets have a look at how we can optimize the training layout, ie how the model is distributed across the available GPUs, to maximize the throughput.

最后，让我们看看如何优化训练布局，即模型如何在可用 GPU 上分布，以最大限度地提高吞吐量。

Optimizing Training Throughput
优化训练吞吐量

HOW MANY GPUS DO WE NEED?
我们需要多少个 GPU?

Great question! After all this talk about specs and benchmarks, you still need to figure out the practical question: how many GPUs should you actually rent or buy?

好问题！在讨论了所有这些规格和基准测试之后，您仍然需要弄清楚一个实际问题：您实际应该租用或购买多少个 GPU?

Determining the right number of GPUs requires balancing training time, cost, and scaling efficiency. Here's the framework we used:

确定正确的 GPU 数量需要平衡训练时间、成本和扩展效率。这是我们使用的框架：

Basic Sizing Formula: 基本尺码公式：

$$\text{GPU Count} = \frac{\text{Total FLOPs Required}}{\text{Per-GPU Throughput} \times \text{Target Training Time}}$$

This formula breaks down the problem into three key components:

该公式将问题分解为三个关键组成部分：

- **Total FLOPs Required** : The computational work needed to train your model (depends on model size, training tokens, and architecture)

  所需的 FLOP 总数：训练模型所需的计算工作（取决于模型大小、训练令牌和架构）

- **Per-GPU Throughput** : How many FLOPs per second each GPU can actually deliver (not theoretical peak!)

  Per-GPU Throughput ： 每个 GPU 每秒实际可以提供多少次 FLOP（不是理论峰值！

- **Target Training Time** : How long you're willing to wait for training to complete

  目标训练时间：您愿意等待训练完成的时间

The key insight: you need to estimate **realistic throughput** , not peak specs. This means accounting for Model FLOPs Utilization (MFU): the percentage of theoretical peak performance you actually achieve in practice.

关键见解：您需要估计实际的吞吐量，而不是峰值规格。这意味着要考虑模型 FLOP 利用率（MFU）：您在实践中实际达到的理论峰值性能的百分比。

For SmolLM3, our calculation looked like this:

对于 SmolLM3，我们的计算如下所示：

- **Model size** : 3B parameters

  型号尺寸：3B 参数

- **Training tokens** : 11 trillion tokens

  训练代币：11 万亿代币

- **Target training time** : ~4 weeks

  目标培训时间：~4 周

- **Expected MFU** : 30% (based on similar scale experiments)

  预期 MFU：30%（基于类似规模的实验）

First, we calculate total FLOPs needed using the standard transformer approximation of **6N FLOPs per token** (where N = parameters):

首先，我们使用每个令牌 6N 个 FLOP 的标准 transformer 近似值（其中 N = 参数）计算所需的总 FLOP：

$$\text{Total FLOPs} = 6 \times 3 \times 10^9 \text{ params} \times 11 \times 10^{12} \text{ tokens} = 1.98 \times 10^{23} \text{ FLOPs}$$

With our expected MFU of 30%, our effective per-GPU throughput becomes:

我们的预期 MFU 为 30%，我们的每个 GPU 有效吞吐量变为：

$$\text{Effective Throughput} = 720 \times 10^{12} \text{ FLOPs/sec} \times 0.30 = 216 \times 10^{12} \text{ FLOPs/sec}$$

Now plugging into our sizing formula:

现在插入我们的尺码公式：

$$\begin{aligned}\text{GPU Count} &= \frac{1.98 \times 10^{23} \text{ FLOPs}}{216 \times 10^{12} \text{ FLOPs/sec} \times 4 \text{ weeks} \times 604,800 \text{ sec/week}} \\ &= \frac{1.98 \times 10^{23}}{5.23 \times 10^{20}} \approx 379 \text{ GPUs}\end{aligned}$$

This calculation pointed us toward 375-400 H100s, and we secured 384 H100s, a number that aligned well with our parallelism strategy and gave us a realistic 4-week timeline with some buffer for unexpected issues like node failures and restarts.

该计算为我们指出了 375-400 个 H100，我们获得了 384 个 H100，这个数字与我们的并行性策略非常吻合，并为我们提供了一个现实的 4 周时间表，并为节点故障和重启等意外问题提供了一些缓冲。

Why More GPUs Isn't Always Better: Amdahl's Law in Action

为什么更多的 GPU 并不总是更好：阿姆达尔定律的实际应用

Here's a counterintuitive truth: **adding more GPUs can actually make your training slower** . This is where Amdahl's Law comes into play.

这是一个违反直觉的事实：添加更多 GPU 实际上会使您的训练变慢 .这就是阿姆达尔定律发挥作用的地方。

Amdahl's Law states that the speedup from parallelization is fundamentally limited by the serial (non-parallelizable) portion of your workload. In LLM training, this "serial" portion is primarily **communication overhead:** the time spent synchronizing gradients/weights/activations across GPUs that can't be parallelized away (read more here).

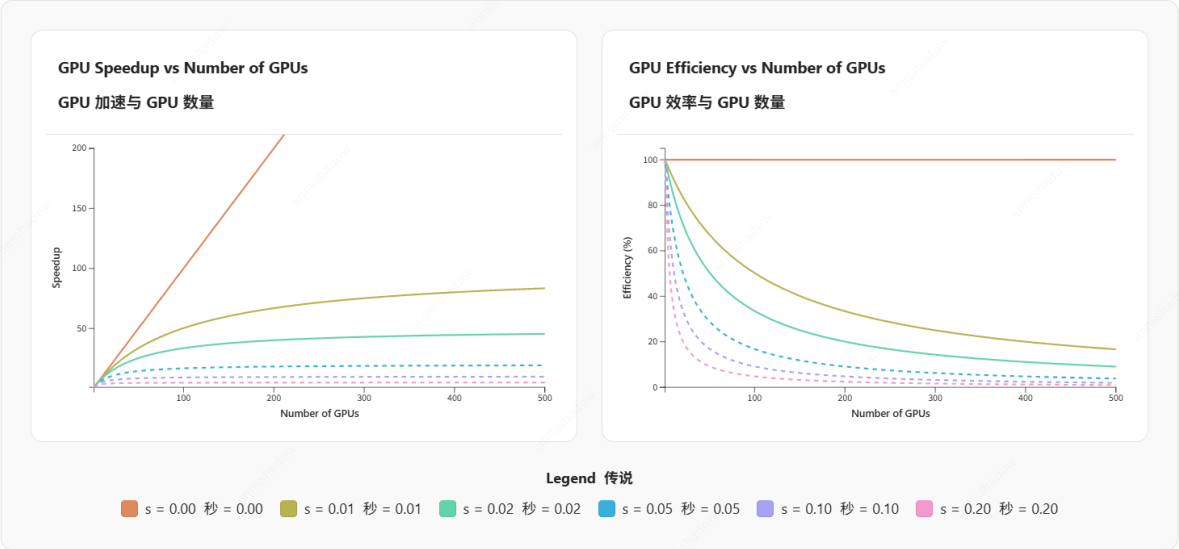
阿姆达尔定律指出，并行化的加速从根本上受到工作负载的串行（不可并行化）部分的限制。在 LLM 训练中，这个"串行"部分主要是通信开销：在无法并行化的 GPU 之间同步梯度/权重/激活所花费的时间（在此处阅读更多内容）。

The formula is: **Maximum Speedup = 1 / (Serial Fraction + Parallel Fraction / Number of Processors)**

公式为：最大加速 = 1 /（串行分数 + 并行分数 / 处理器数）

GPU Scaling in distributed LLM Training: Amdahl's Law in action

分布式 LLM 训练中的 GPU 扩展：阿姆达尔定律的实际应用



For SmolLM3's 3B model, if communication takes 10% of each training step, then no matter how many GPUs you add, you'll never get more than 10x speedup. Worse, as you add more GPUs, the communication fraction often *increases* because:

对于 SmolLM3 的 3B 模型，如果通信占用每个训练步骤的 10%，那么无论你添加多少个 GPU，你永远不会获得超过 10 倍的加速。更糟糕的是，随着您添加更多 GPU，通信分数通常会增加，因为：

- More GPUs = more AllReduce participants = longer synchronization

  更多 GPU = 更多 AllReduce 参与者 = 更长的同步时间

- Network latency/bandwidth becomes the bottleneck

  网络延迟/带宽成为瓶颈

- Small models can't hide communication behind compute

  小型模型无法将通信隐藏在计算后面

For SmolLM3, we used weak scaling principles: our global batch size scaled with our

GPU count, maintaining roughly 8K tokens per GPU globally. This kept our communication-to-computation ratio reasonable while maximizing throughput.

对于 SmolLM3，我们使用了弱扩展原则：我们的全局批量大小随我们的 GPU 数量而扩展，在全球范围内每个 GPU 保持大约 8K 个令牌。这使我们的通信与计算比率保持合理，同时最大限度地提高吞吐量。

FINDING THE OPTIMAL PARALLELISM CONFIGURATION
寻找最佳并行度配置

Once you've secured your GPUs, the next challenge is configuring them to actually train efficiently. For this the parallelism strategy becomes critical.

一旦您保护了 GPU，下一个挑战是配置它们以实际有效地训练。为此，并行策略变得至关重要。

We follow the **Ultra-Scale Playbook** 's approach to finding optimal training configurations. The playbook breaks the problem into three sequential steps: first ensure the model fits in memory, then achieve your target batch size, and finally optimize for maximum throughput. Let's walk through how we applied this to SmolLM3.

我们遵循 Ultra-Scale Playbook 的方法来寻找最佳训练配置。该 playbook 将问题分解为三个连续步骤：首先确保模型适合内存，然后达到目标批量大小，最后优化以获得最大吞吐量。让我们来看看我们是如何将其应用于 SmolLM3 的。

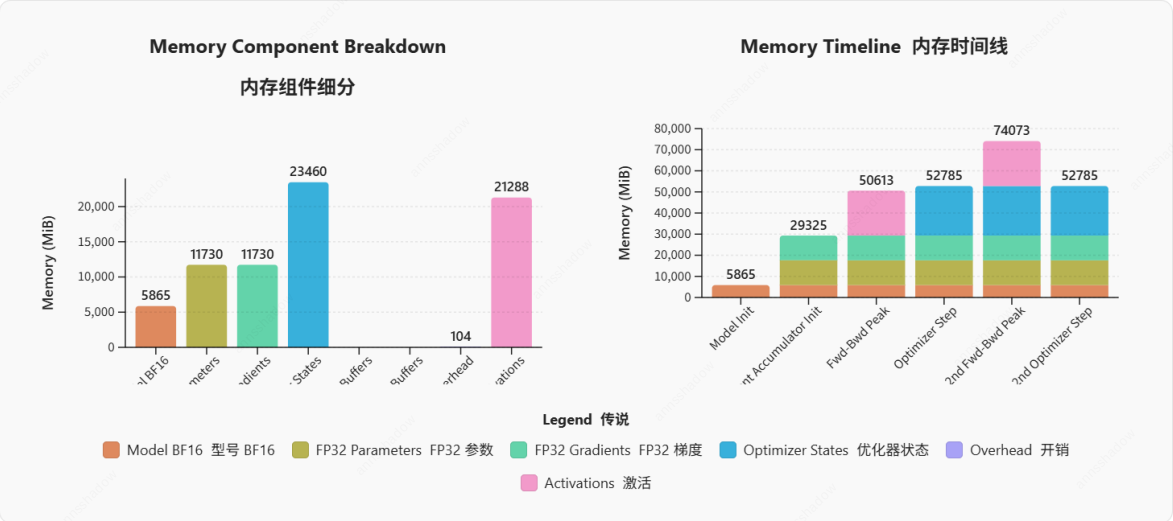
For detailed explanations of different parallelism strategies (Data Parallelism, Tensor Parallelism, Pipeline Parallelism, ZeRO, etc.), we urge you once again to checkout the Ultra-Scale Playbook *inserts Bernie meme*

有关不同并行性策略（数据并行性、张量并行性、管道并行性、ZeRO 等）的详细说明，我们再次敦促您查看 Ultra-Scale Playbook *插入 Bernie 模因*

STEP 1: FITTING A TRAINING STEP IN MEMORY
步骤 1：在内存中拟合训练步骤

The first question is simple: does our SmolLM3 3B model even fit in a single H100's 80GB of memory? To answer this, we use nanotron's `predict_memory` tool, which estimates memory consumption for model parameters, optimizer states, gradients, and activations.

第一个问题很简单：我们的 SmolLM3 3B 型号是否适合单个 H100 的 80GB 内存？为了回答这个问题，我们使用了 nanotron 的工具，该 `predict_memory` 工具可以估计模型参数、优化器状态、梯度和激活的内存消耗。



Memory timeline from nanotron's predict_memory tool showing SmolLM3 3B peaks at 74GB, approaching the H100's 80GB limit.

来自 nanotron predict_memory 工具的内存时间线显示 SmolLM3 3B 峰值为 74GB，接近 H100 的 80GB 限制。

The results show we're pushing close to the 80GB limit.

结果显示，我们正在接近 80GB 的限制。

This means we need some form of parallelism that reduces per-GPU memory footprint, whether that's Tensor Parallelism (splitting model layers across GPUs), Pipeline Parallelism (splitting model depth across GPUs), or ZeRO optimizer sharding (distributing optimizer states).

这意味着我们需要某种形式的并行性来减少每个 GPU 的内存占用，无论是张量并行性（跨 GPU 拆分模型层）、管道并行性（跨 GPU 拆分模型深度）还是 ZeRO 优化器分片（分布优化器状态）。

Without at least one of these strategies, we won't be able to train efficiently or at all.

如果没有这些策略中的至少一种，我们将无法有效地训练或根本无法训练。

STEP 2: ACHIEVING THE TARGET GLOBAL BATCH SIZE
第 2 步：实现目标全局批量大小

Now that we know the model fits in memory with some form of parallelism, we need to determine how to achieve our target global batch size (GBS) of approximately 2 million tokens. This constraint gives us our first equation:

现在我们知道模型以某种形式的并行性适合内存，我们需要确定如何实现大约 200 万个令牌的目标全局批量大小 （GBS）。这个约束给了我们第一个方程：

$$\text{GBS} = \text{DP} \times \text{MBS} \times \text{GRAD\_ACC} \times \text{SEQLEN} \approx 2\text{M tokens}$$

Where: 哪里：

- **DP (Data Parallelism)** : Number of data-parallel replicas

  DP（数据并行性）： 数据并行副本数

- **MBS (Micro Batch Size)** : Tokens processed per GPU per micro-batch

  MBS（微批量大小）： 每个 GPU 每个微批处理的代币

- **GRAD_ACC (Gradient Accumulation)** : Number of forward-backwards before optimizer step

  GRAD_ACC（梯度累积）： 优化器步骤前的前后退次数

- **SEQLEN (Sequence Length)** : Tokens per sequence (4096 for the 1st pretraining stage)

  SEQLEN（序列长度）： 每个序列的标记（第一个预训练阶段为 4096）

We also have a hardware constraint from our 384 H100s:

我们的 384 H100 也有硬件限制：

$$\text{DP} \times \text{TP} \times \text{PP} = 384 = 2^7 \times 3$$

Where: 哪里：

- **TP (Tensor Parallelism)** : GPUs per model layer (splits weight matrices)

  TP（张量并行性）： 每个模型层的 GPU（拆分权重矩阵）

- **PP (Pipeline Parallelism)** : GPUs per model depth (splits layers vertically)

  PP（Pipeline Parallelism）： 每个模型深度的 GPU（垂直拆分层）

These two equations define our search space. We need to find values that satisfy both constraints while maximizing training throughput.

这两个方程定义了我们的搜索空间。我们需要找到满足这两个约束条件的值，同时最大限度地提高训练吞吐量。

STEP 3: OPTIMIZING TRAINING THROUGHPUT
第 3 步：优化训练吞吐量

With our constraints established, we need to find the parallelism configuration that maximizes training throughput. The search space is defined by our hardware topology and model architecture.

建立约束后，我们需要找到最大化训练吞吐量的并行度配置。搜索空间由我们的硬件拓扑和模型架构定义。

Our hardware setup presents two distinct types of interconnects, as we saw in the section above: NVLink for intra-node communication (900 GB/s) and EFA for inter-node communication (~50 GB/s).

我们的硬件设置提供了两种不同类型的互连，正如我们在上一节中看到的那样：用于节点内通信的 NVLink （900 GB/s） 和用于节点间通信的 EFA （~50 GB/s）。

This topology naturally suggests using at least two forms of parallelism to match our network characteristics.

这种拓扑自然建议使用至少两种形式的并行性来匹配我们的网络特征。

The dramatic bandwidth difference between these interconnects will heavily influence which parallelism strategies work best.

这些互连之间的巨大带宽差异将严重影响哪种并行策略效果最好。

From a model perspective, SmolLM3's architecture constrains our options. Since we're not using a Mixture-of-Experts architecture, we don't need **Expert Parallelism** . Similarly, training with a 4096 sequence length in the first stage means **Context Parallelism** isn't required. This leaves us with three primary parallelism dimensions to explore: **Data Parallelism** (DP), **Tensor Parallelism** (TP), and **Pipeline Parallelism** (PP).

从模型的角度来看，SmolLM3 的架构限制了我们的选择。由于我们没有使用专家混合架构，因此我们不需要专家并行性。同样，在第一阶段使用 4096 序列长度进行训练意味着不需要上下文并行性。这给我们留下了三个主要的并行性维度来探索：数据并行性 （DP）、张量并行性 （TP） 和管道并行性 （PP）。

Given our constraints from Step 2, we need to sweep across several parameters:

鉴于步骤 2 中的约束，我们需要扫描几个参数：

- **DP with ZeRO variants** (ZeRO-0, ZeRO-1, ZeRO-3): Values from 1 to 384, constrained to multiples of 2 and/or 3

  具有 ZeRO 变体（ZeRO-0、ZeRO-1、ZeRO-3）的 DP：值从 1 到 384，限制为 2 和/或 3 的倍数

- **TP** (1, 2, 3, 4, 6, 8): Keep within a single node to fully leverage NVLink's high bandwidth

  TP（1、2、3、4、6、8）：保持在单个节点内，充分利用 NVLink 的高带宽

- **PP** (1..48): Split model depth across GPUs

  PP （1..48）：跨 GPU 拆分模型深度

- **MBS** (2, 3, 4, 5): Depending on memory savings from parallelism, we can increase MBS to better utilize Tensor Cores

  MBS （2， 3， 4， 5）： 根据并行性节省的内存，我们可以增加 MBS 以更好地利用 Tensor Core

- **Activation checkpointing** (none, selective, full): Trade additional compute for reduced memory and communication

  激活检查点（无、选择性、完整）： 用额外的计算来减少内存和通信

- **Kernel optimizations** : CUDA graphs and optimized kernels where available

  内核优化： CUDA 图和优化的内核（如果可用）

While this may seem like an overwhelming number of combinations, a practical approach is to benchmark each dimension independently first, then eliminate configurations that significantly hurt throughput. The key insight is that not all parallelism strategies are created equal.

虽然这似乎是压倒性的组合数量，但一种实用的方法是首先独立对每个维度进行基准测试，然后消除严重损害吞吐量的配置。关键的见解是，并非所有并行度策略都是一样的。

Some introduce communication overhead that far outweighs their benefits, especially at our scale.

有些引入的通信开销远远超过其好处，尤其是在我们的规模上。

In our case, **Pipeline Parallelism** showed poor performance characteristics. PP requires frequent pipeline bubble synchronization across nodes, and with our relatively small 3B model, the communication overhead dominated any potential benefits.

在我们的例子中，管道并行性表现出较差的性能特征。PP 需要跨节点频繁的管道气泡同步，而对于我们相对较小的 3B 模型，通信开销主导了任何潜在好处。

Additionally, we didn't have access to highly efficient PP schedules that could eliminate the pipeline bubble entirely, which further limited PP's viability.

此外，我们无法获得可以完全消除管道泡沫的高效 PP 计划，这进一步限制了 PP 的可行性。

Similarly,**ZeRO** levels above 0 introduced significant all-gather and reduce-scatter operations that hurt throughput more than they helped with memory. These early benchmarks allowed us to narrow our search space dramatically, focusing on configurations that combined **Data Parallelism** with modest **Tensor Parallelism** .

同样，高于 0 的 ZeRO 水平引入了显着的全聚集和减少散射作，这些作对吞吐量的损害大于对内存的帮助。这些早期的基准测试使我们能够显着缩小搜索空间，专注于将数据并行性与适度张量并行性相结合的配置。

👉 To evaluate each configuration, we run benchmarks for 5 iterations and record **tokens per second per GPU (tok/s/gpu)** , which is ultimately the metric we care about. We use Weights & Biases and Trackio to log throughputs and configurations, making it easy to compare different parallelism strategies.

👉 为了评估每个配置，我们运行了 5 次迭代的基准测试，并记录每个 GPU 每秒的令牌数 making it easy to compare different parallelism strategies.

👉 为了评估每个配置，我们运行了 5 次迭代的基准测试，并记录每个 GPU 每秒的令牌数（tok/s/gpu），这最终是我们关心的指标。我们使用 Weights & Biases 和 Trackio 来记录吞吐量和配置，从而可以轻松比较不同的并行策略。

After systematically benchmarking the available options in nanotron, we settled on **DP = 192** , which leverages inter-node EFA bandwidth for data-parallel gradient synchronization. This means 192 independent model replicas, each processing different batches of data. For Tensor Parallelism, we chose **TP = 2** , keeping tensor-parallel communication within a single node to fully exploit NVLink's high bandwidth. This splits each layer's weight matrices across two GPUs, requiring fast communication for the forward and backward passes.

在系统地对 nanotron 中的可用选项进行基准测试后，我们确定了 DP = 192，它利用节点间 EFA 带宽进行数据并行梯度同步。这意味着 192 个独立的模型副本，每个副本处理不同批次的数据。对于张量并行性，我们选择了 TP = 2，在单个节点内保持张量并行通信，以充分利用 NVLink 的高带宽。这将每个层的权重矩阵拆分为两个 GPU，需要为前向和后向传递进行快速通信。

Our **Micro Batch Size = 3** strikes a balance between memory usage and compute efficiency. Larger batch sizes would better utilize Tensor Cores, but we're already pushing close to memory limits. Finally, we opted for ZeRO-0, meaning no optimizer state sharding.

我们的微批量大小 = 3 在内存使用和计算效率之间取得了平衡。更大的批量大小会更好地利用 Tensor Core，但我们已经接近内存限制。最后，我们选择了 ZeRO-0，这意味着没有优化器状态分片。

While ZeRO-1 or ZeRO-3 could reduce memory footprint, the communication overhead from gathering and scattering optimizer states across our 384 GPUs would significantly hurt throughput.

虽然 ZeRO-1 或 ZeRO-3 可以减少内存占用，但在我们的 384 个 GPU 上收集和分散优化器状态所带来的通信开销会严重损害吞吐量。

This configuration achieves our target global batch size of approximately 2 million tokens (192 × 3 × 1 × 4096 ≈ 2.3M) while maximizing throughput on our 384 H100 cluster. You can see the full training configuration in our stage1_8T.yaml.

此配置实现了我们目标的全球批量大小，即大约 200 万个令牌（192 × 3 × 1 × 4096 ≈ 2.3M），同时最大限度地提高了 384 H100 集群的吞吐量。您可以在我们的 stage1_8T.yaml 中查看完整的训练配置。

Many of these parallelism decisions were influenced by the state of libraries at the time of experiments. For instance, nanotron didn't support ZeRO-3 yet, and we lacked access to highly optimized Pipeline Parallelism schedules that could eliminate pipeline bubbles.

其中许多并行度决策都受到实验时库状态的影响。例如，nanotron 尚不支持 ZeRO-3，而且我们无法访问高度优化的管道并行性计划来消除管道气泡。

As the framework evolves, some of these trade-offs may shift. Contributions are always welcome!

随着框架的发展，其中一些权衡可能会发生变化。随时欢迎贡献！

We started this journey with a simple question: what does it actually take to train a high-performance LLM in 2025? After walking through the complete pipeline—from pretraining to post-training—we've shown you not just the techniques, but the methodology that makes them work.

我们从一个简单的问题开始了这段旅程：在 2025 年训练高性能法学硕士实际上需要什么？在完成了从训练前到训练后的完整管道之后，我们不仅向您展示了技术，还向您展示了使它们发挥作用的方法。

Pretraining at scale. We walked through the Training Compass framework for deciding whether to train at all, then showed how to translate goals into concrete architectural decisions.

大规模预训练。我们介绍了用于决定是否进行培训的培训指南针框架，然后展示了如何将目标转化为具体的架构决策。

You've seen how to set up reliable ablation pipelines, test changes individually, and scale from few-billion-token experiments to multi-trillion-token runs.

你已经了解了如何设置可靠的消融管道、单独测试更改，以及从几十亿个令牌的实验扩展到数万亿个令牌的运行。

We documented the infrastructure challenges that can emerge at scale (throughput collapses, dataloader bottlenecks, subtle bugs) and how monitoring and systematic derisking help you catch them early and debug quickly.

我们记录了大规模可能出现的基础设施挑战（吞吐量崩溃、数据加载器瓶颈、细微的错误），以及监控和系统去风险如何帮助您及早发现这些挑战并快速调试。

Post-training in practice. We showed that going from a base model to a production assistant requires its own systematic approach: establishing evals before training anything, iterating on SFT data mixtures, applying preference optimization, and optionally pushing further with RL.

培训后实践。我们表明，从基础模型到生产助手需要自己的系统方法：在训练任何东西之前建

立评估，迭代 SFT 数据混合，应用偏好优化，并可选择使用 RL 进一步推动。

You've seen how vibe testing catches bugs that metrics miss, how chat templates can silently break instruction-following, and why data mixture balance matters as much in post-training as it does in pretraining.

你已经了解了氛围测试如何发现指标遗漏的错误，聊天模板如何悄无声息地破坏指令遵循，以及为什么数据混合平衡在训练后和训练前一样重要。

Throughout both phases, we kept coming back to the same core insights: validate everything through experiments, change one thing at a time, expect scale to break things in new ways, and let your use case drive decisions rather than chasing every new paper.

new paper.

在这两个阶段，我们不断回到相同的核心见解：通过实验验证一切，一次改变一件事，期望规模以新的方式打破事物，并让你的用例推动决策，而不是追逐每一篇新论文。

Following this process, we trained SmolLM3: a competitive 3B multilingual reasoner with long context. Along the way, we learned a lot about what works, what breaks, and how to debug when things go wrong. We've tried to document it all, the successes and failures alike.

在此过程中，我们训练了 SmolLM3：一个具有长上下文的竞争性 3B 多语言推理器。在此过程中，我们学到了很多关于什么有效、什么坏了以及在出现问题时如何调试的知识。我们试图记录这一切，包括成功和失败。

What's next? 下一步是什么?

This blog covers the fundamentals of modern LLM training, but the field evolves rapidly. Here are ways to go deeper:

本博客涵盖了现代 LLM 培训的基础知识，但该领域发展迅速。以下是深入了解的方法：

- **Run experiments yourself.** Reading about ablations is useful; running your own teaches you what actually matters. Pick a small model, set up evals, and start experimenting.

  自己进行实验。阅读有关消融的信息是有用的;经营自己的活动教会了你什么真正重要。选择一个小模型，设置评估，然后开始试验。

- **Read the source code.** Training frameworks like nanotron, TRL, and others are open source. Understanding their implementations reveals details that papers gloss over.

  阅读源代码。nanotron、TRL 等训练框架是开源的。了解它们的实现可以揭示论文掩盖的细节。

- **Follow recent work.** Papers of recent state-of-the-art models show where the field is heading. The references section contains our curated list of impactful papers and resources.

  关注最近的工作。最近最先进模型的论文显示了该领域的发展方向。参考文献部分包含我们精选的有影响力的论文和资源列表。

We hope this blog helps you approach your next training project with clarity and confidence, whether you're at a large lab pushing the frontier or a small team solving a specific problem.

我们希望这篇博客能帮助您清晰、自信地处理下一个培训项目，无论您是在推动前沿的大型实验室还是在解决特定问题的小型团队中。

every great model has debugging stories behind it. May the force of open source and open science always be with you!

现在去训练一些东西。当您的损失在凌晨 2 点神秘飙升时，请记住：每个伟大的模型背后都有调试故事。愿开源和开放科学的力量永远伴随着你！

ACKNOWLEDGMENTS 确认

We thank Guilherme, Hugo and Mario for their valuable feedback, and Abubakar for his help with Trackio features.

我们感谢 Guilherme、Hugo 和 Mario 的宝贵反馈，以及 Abubakar 对 Trackio 功能的帮助。

References 引用

Below is a curated list of papers, books, and blog posts that have informed us the most on our LLM training journey.

以下是在我们的 LLM 培训之旅中为我们提供最多信息的论文、书籍和博客文章的精选列表。

LLM ARCHITECTURE LLM 架构

- Dense models: Llama3, Olmo2, MobileLLM

  密集模型：Llama3、Olmo2、MobileLLM

- MoEs: DeepSeek V2, DeepSeek V3, Scaling Laws of Efficient MoEs

  MoE：DeepSeek V2、DeepSeek V3、高效 MoE 的扩展定律

- Hybrid: MiniMax-01, Mamba2

  混合动力：MiniMax-01、Mamba2

OPTIMISERS & TRAINING PARAMETERS
优化器和训练参数

- Muon is Scalable for LLM Training, Fantastic pretraining optimisers

  Muon 可扩展用于 LLM 训练，出色的预训练优化器

- Large Batch Training, DeepSeekLLM

  大批量训练，DeepSeekLLM

DATA CURATION 数据管理

- Web: FineWeb & FineWeb-Edu, FineWeb2, DCLM

  网站：FineWeb 和 FineWeb-Edu、FineWeb2、DCLM

- Code: The Stack v2, To Code or Not to Code

  代码：The Stack v2，编码或不编码

- Math: DeepSeekMath, FineMath, MegaMath

  数学：DeepSeekMath、FineMath、MegaMath

- Data mixtures: SmolLM2, Does your data spark joy

  数据混合：SmolLM2，你的数据会激发快乐吗

SCALING LAWS 缩放法则

- Kaplan, Chinchilla, Scaling Data-Constrained Language Models

  Kaplan， Chinchilla，扩展数据受限的语言模型

POST-TRAINING 培训后

- InstructGPT: OpenAI's foundational paper to turn base models into helpful assistants. The precursor to ChatGPT and a key step on humanity's path up the Kardashev scale.

  InstructGPT：OpenAI 的基础论文，旨在将基础模型转变为有用的助手。ChatGPT 的前身，也是人类攀登卡尔达舍夫量表的关键一步。

- Llama 2 & 3: Extremely detailed tech reports from Meta on the training behind their Llama models (may they rest in peace). They each contain many insights into human data collection, both for human preferences and model evaluation.

  Llama 2 和 3：Meta 关于其 Llama 模型背后的训练的极其详细的技术报告（愿他们安息）。它们都包含许多关于人类数据收集的见解，包括人类偏好和模型评估。

- Secrets of RLHF in LLMs, Part I & II: these papers contain lots of goodies on the nuts and bolts for RLHF, specifically on how to train strong reward models.

  法学硕士中 RLHF 的秘密，第一部分和第二部分：这些论文包含许多关于 RLHF 具体细节的好东西，特别是关于如何训练强奖励模型。

- Direct Preference Optimisation: the breakthrough paper from 2023 that convinced everyone to stop doing RL with LLMs.

  直接偏好优化：2023 年的突破性论文，说服了所有人停止使用 LLM 进行 RL。

- DeepSeek-R1: the breakthrough paper from 2025 that convinced everyone to start doing RL with LLMs.

  DeepSeek-R1：2025 年的突破性论文，说服了每个人开始用法学硕士做 RL。

- Dr. GRPO: one of the most important papers on understanding the baked-in biases with GRPO and how to fix them.

  GRPO 博士：关于理解 GRPO 的固有偏见以及如何解决这些偏见的最重要的论文之一。

- DAPO: Bytedance shares many implementation details to unlock stable R1-Zero-like training for the community.

  DAPO：字节跳动分享了许多实现细节，为社区解锁稳定的 R1-Zero 类训练。

- ScaleRL: a massive flex from Meta to derive scaling laws for RL. Burns over 400k GPU hours to establish a training recipe that scales reliably over many orders of compute.

  ScaleRL：Meta 的大规模弹性，用于推导出 RL 的缩放定律。燃烧超过 400k GPU 小时，以建立在多个计算阶数上可靠扩展的训练配方。

- LoRA without Regret: a beautifully written blog post which finds that RL with low-rank LoRA can match full-finetuning (a most surprising result).

  LoRA 无悔：一篇写得很漂亮的博客文章，发现具有低秩 LoRA 的 RL 可以匹配完全微调（一个最令人惊讶的结果）。

- Command A: a remarkably detailed tech report from Cohere on various strategies to post-train LLMs effectively.

  命令 A：Cohere 的一份非常详细的技术报告，内容涉及有效培训 LLM 的各种策略。

INFRASTRUCTURE 基础设施

- UltraScale Playbook UltraScale 手册
- Jax scaling book 贾克斯缩放书
- Modal GPU Glossary 模态 GPU 术语表

- Jax scaling book 贾克斯缩放书
- Modal GPU Glossary 模态 GPU 术语表

TRAINING FRAMEWORKS 培训框架

- Megatron-LM 威震天-登月舱
- DeepSpeed 深速
- Torchtitan 火炬泰坦
- Nanotron 纳米创
- NanoChat 纳米聊天
- TRL 总链

EVALUATION 评估

- The LLM Evaluation Guidebook
  LLM 评估指南
- OLMES 奥尔姆斯
- FineTasks 精细任务
- Lessons from the trenches
  战壕的教训

Citation 引文

For attribution in academic contexts, please cite this work as

对于学术背景下的归属，请将此作品引用为

```
Loubna Ben Allal, Lewis Tunstall, Nouamane Tazi, Elie Bakouch, Ed Beeching, Carlos Miguel Patiño,
Clémentine Fourrier, Thibaud Frere, Anton Lozhkov, Colin Raffel, Leandro von Werra, Thomas Wolf (2025).
"The Smol Training Playbook: The Secrets to Building World-Class LLMs".
```

BibTeX citation BibTeX 引文

```
@misc{allal2025_the_smol_training_playbook_the_secrets_to_building_world_class_llms,
  title={The Smol Training Playbook: The Secrets to Building World-Class LLMs},
  author={Loubna Ben Allal and Lewis Tunstall and Nouamane Tazi and Elie Bakouch and Ed Beeching and
Carlos Miguel Patiño and Clémentine Fourrier and Thibaud Frere and Anton Lozhkov and Colin Raffel and
Leandro von Werra and Thomas Wolf},
  year={2025},

}
```

References 引用

1. Agarwal, R., Vieillard, N., Zhou, Y., Stanczyk, P., Ramos, S., Geist, M., & Bachem, O. (2024). *On-Policy Distillation of Language Models: Learning from Self-Generated Mistakes*. https://arxiv.org/abs/2306.13649

   阿加瓦尔，R.，维亚拉德，N.，周，Y.，斯坦奇克，P.，拉莫斯，S.，盖斯特，M.，和巴赫姆，O.（2024）。语言模型的政策蒸馏：从自生成的错误中学习。https://arxiv.org/abs/2306.13649

   ↑

2. Ainslie, J., Lee-Thorp, J., de Jong, M., Zemlyanskiy, Y., Lebrón, F., & Sanghai, S. (2023). *GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints*. https://arxiv.org/abs/2305.13245 ↑ back: 1, 2

   Ainslie，J.、Lee-Thorp，J.、de Jong，M.、Zemlyanskiy，Y.、Lebrón，F. 和 Sanghai，S.（2023）。GQA：从多头检查点训练广义多查询转换器模型。https://arxiv.org/abs/2305.13245 背：1、2

3. Allal, L. B., Lozhkov, A., Bakouch, E., Blázquez, G. M., Penedo, G., Tunstall, L., Marafioti, A., Kydlíček, H., Lajarín, A. P., Srivastav, V., Lochner, J., Fahlgren, C., Nguyen, X.-S., Fourrier, C., Burtenshaw, B., Larcher, H., Zhao, H., Zakka, C., Morlon, M., ... Wolf, T. (2025). *SmolLM2: When Smol Goes Big – Data-Centric Training of a Small Language Model*. https://arxiv.org/abs/2502.02737 ↑ back: 1, 2, 3

   Allal，L. B.，Lozhkov，A.，Bakouch，E.，Blázquez，G. M.，Penedo，G.，Tunstall，L.，Marafioti，A.，Kydlíček，H.，Lajarín，A. P.，Srivastav，V.，Lochner，J.，Fahlgren，C.，Nguyen，X.-S.，Fourrier，C.，Burtenshaw，B.，Larcher，H.，Zhao，H.，Zakka，C.，Morlon，M.，...沃尔夫，T.（2025）。SmolLM2：当 Smol 变大时 – 小型语言模型的以数据为中心的训练。https://arxiv.org/abs/2502.02737 背：1、2、3

4. Almazrouei, E., Alobeidli, H., Alshamsi, A., Cappelli, A., Cojocaru, R., Debbah, M., Goffinet, É., Hesslow, D., Launay, J., Malartic, Q., Mazzotta, D., Noune, B., Pannier, B., & Penedo, G. (2023). *The Falcon Series of Open Language Models*. https://arxiv.org/abs/2311.16867

   Almazrouei，E.、Alobeidli，H.、Alshamsi，A.、Cappelli，A.、Cojocaru，R.、Debbah，M.、Goffinet，É.、Hesslow，D.、Launay，J.、Malartic，Q.、Mazzotta，D.、Noune，B.、Pannier，B. 和 Penedo，G.（2023）。Falcon 系列开放语言模型。https://arxiv.org/abs/2311.16867

   ↑

5. An, C., Huang, F., Zhang, J., Gong, S., Qiu, X., Zhou, C., & Kong, L. (2024). *Training-Free Long-Context Scaling of Large Language Models*. https://arxiv.org/abs/2402.17463

   An，C.、Huang，F.、Zhang，J.、Gong，S.、Qiu，X.、周 C. 和 Kong，L.（2024）。大型语言模型的免训练长上下文扩展。https://arxiv.org/abs/2402.17463

   ↑

6. Aryabumi, V., Su, Y., Ma, R., Morisot, A., Zhang, I., Locatelli, A., Fadaee, M., Üstün, A., & Hooker, S. (2024). *To Code, or Not To Code? Exploring Impact of Code in Pre-training*. https://arxiv.org/abs/2408.10914

   Aryabumi，V.、Su，Y.、马 R.、莫里索特 A.、张 I.、洛卡特利 A.、法达伊 M.、乌斯图恩 A. 和胡克 S.（2024）。编码，还是不编码？探索代码在预训练中的影响。https://arxiv.org/abs/2408.10914

   ↑

7. Bai, J., Bai, S., Chu, Y., Cui, Z., Dang, K., Deng, X., Fan, Y., Ge, W., Han, Y., Huang, F., Hui, B., Ji, L., Li, M., Lin, J., Lin, R., Liu, D., Liu, G., Lu, C., Lu, K., ... Zhu, T. (2023). *Qwen Technical Report*. https://arxiv.org/abs/2309.16609

   Bai，J.，Bai，S.，Chu，Y.，Cui，Z.，Dang，K.，邓 X.，Fan，Y.，Ge，W.，Han，Y.，Huang，F.，Hui，B.，Ji，L.，Li，M.，Lin，J.，Lin，R.，Liu，D.，Liu，G.，Lu，C.，Lu，K.，...朱，T.（2023）。Qwen 技术报告。https://arxiv.org/abs/2309.16609

   ↑

8. Barres, V., Dong, H., Ray, S., Si, X., & Narasimhan, K. (2025). *τ^2-Bench: Evaluating Conversational Agents in a Dual-Control Environment*. https://arxiv.org/abs/2506.07982

   巴雷斯，V.，董，H.，雷，S.，Si，X.，和纳拉辛汉，K.（2025）。τ^2-Bench：在双重控制环境中评估对话代理。https://arxiv.org/abs/2506.07982

   ↑

9. Beck, M., Pöppel, K., Lippe, P., & Hochreiter, S. (2025). *Tiled Flash Linear Attention: More Efficient Linear RNN and xLSTM Kernels*. https://arxiv.org/abs/2503.14376

   贝克，M.、波佩尔，K.、利佩，P. 和霍赫赖特，S.（2025）。平铺闪存线性注意力：更高效的线性 RNN 和 xLSTM 内核。https://arxiv.org/abs/2503.14376

   ↑

10. Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., ... Amodei, D. (2020). *Language Models are Few-Shot Learners*. https://arxiv.org/abs/2005.14165

    Brown，T. B.，Mann，B.，Ryder，N.，Subbiah，M.，Kaplan，J.，Dhariwal，P.，Neelakantan，A.，Shyam，P.，Sastry，G.，Askell，A.，Agarwal，S.，Herbert-Voss，A.，Krueger，G.，Henighan，T.，Child，R.，Ramesh，A.，Ziegler，DM，Wu，J.，Winter，C.，...阿莫迪，D.（2020）。语言模型是少数样本学习器。https://arxiv.org/abs/2005.14165

    ↑

11. Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto, H. P., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., ... Zaremba, W. (2021). *Evaluating Large Language Models Trained on Code*. https://arxiv.org/abs/2107.03374

Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto, H. P., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., ...扎伦巴, W.（2021）。评估在代码上训练的大型语言模型。https://arxiv.org/abs/2107.03374

↑

12. Chen, Y., Huang, B., Gao, Y., Wang, Z., Yang, J., & Ji, H. (2025a). *Scaling Laws for Predicting Downstream Performance in LLMs*. https://arxiv.org/abs/2410.08527

陈 Y.、黄 B.、高 Y.、王 Z.、杨 J. 和季 H.（2025a）。用于预测法学硕士下游性能的缩放定律 https://arxiv.org/abs/2410.08527

↑

13. Chen, Y., Huang, B., Gao, Y., Wang, Z., Yang, J., & Ji, H. (2025b). *Scaling Laws for Predicting Downstream Performance in LLMs*. https://arxiv.org/abs/2410.08527

陈 Y.、黄 B.、高 Y.、王 Z.、杨 J. 和季 H.（2025b）。用于预测法学硕士下游性能的缩放定律 https://arxiv.org/abs/2410.08527

↑

14. Child, R., Gray, S., Radford, A., & Sutskever, I. (2019). Generating long sequences with sparse transformers. *arXiv Preprint arXiv:1904.10509*.

Child, R.、Gray, S.、Radford, A. 和 Sutskever, I.（2019）。使用稀疏变压器生成长序列。arXiv 预印本 arXiv：1904.10509。

↑

15. Chowdhery, A., Narang, S., Devlin, J., Bosma, M., Mishra, G., Roberts, A., Barham, P., Chung, H. W., Sutton, C., Gehrmann, S., Schuh, P., Shi, K., Tsvyashchenko, S., Maynez, J., Rao, A., Barnes, P., Tay, Y., Shazeer, N., Prabhakaran, V., ... Fiedel, N. (2022). *PaLM: Scaling Language Modeling with Pathways*. https://arxiv.org/abs/2204.02311 ↑ back: 1, 2, 3, 4

Chowdhery, A., Narang, S., Devlin, J., Bosma, M., Mishra, G., Roberts, A., Barham, P., Chung, HW, Sutton, C., Gehrmann, S., Schuh, P., Shi, K., Tsvyashchenko, S., Maynez, J., Rao, A., Barnes, P., Tay, Y., Shazeer, N., Prabhakaran, V., ...菲德尔, N.（2022）。PaLM：使用路径扩展语言建模。https://arxiv.org/abs/2204.02311 背：1、2、3、4

16. Chu, T., Zhai, Y., Yang, J., Tong, S., Xie, S., Schuurmans, D., Le, Q. V., Levine, S., & Ma, Y. (2025). *SFT Memorizes, RL Generalizes: A Comparative Study of Foundation Model Post-training*. https://arxiv.org/abs/2501.17161

Chu, T.、Zhai, Y.、Yang, J.、Tong, S.、Xie, S.、Schuurmans, D.、Le, Q. V.、Levine, S. 和 马, Y.（2025）。SFT 记忆，RL 泛化：基础模型训练后的比较研究。https://arxiv.org/abs/2501.17161

↑

17. Cobbe, K., Kosaraju, V., Bavarian, M., Chen, M., Jun, H., Kaiser, L., Plappert, M., Tworek, J., Hilton, J., Nakano, R., Hesse, C., & Schulman, J. (2021). *Training Verifiers to Solve Math Word Problems*. https://arxiv.org/abs/2110.14168

Cobbe, K.、Kosaraju, V.、Bavarian, M.、Chen, M.、Jun, H.、Kaiser, L.、Plappert, M.、Tworek, J.、Hilton, J.、Nakano, R.、Hesse, C. 和 Schulman, J.（2021）。训练验证者解决数学应用题。https://arxiv.org/abs/2110.14168

↑

18. Cohere, T., :, Aakanksha, Ahmadian, A., Ahmed, M., Alammar, J., Alizadeh, M., Alnumay, Y., Althammer, S., Arkhangorodsky, A., Aryabumi, V., Aumiller, D., Avalos, R., Aviv, Z., Bae, S., Baji, S., Barbet, A., Bartolo, M., Bebensee, B., ... Zhao, Z. (2025). *Command A: An Enterprise-Ready Large Language Model*. https://arxiv.org/abs/2504.00698

Cohere, T., :, Aakanksha, Ahmadian, A., Ahmed, M., Alammar, J., Alizadeh, M., Alnumay, Y., Althammer, S., Arkhangorodsky, A., Aryabumi, V., Aumiller, D., Avalos, R., Aviv, Z., Bae, S., Baji, S., Barbet, A., Bartolo, M., Bebensee, B., ...赵, Z.（2025）。命令 A：企业级大型语言模型。https://arxiv.org/abs/2504.00698

↑

19. Dagan, G., Synnaeve, G., & Rozière, B. (2024). *Getting the most out of your tokenizer for pre-training and domain adaptation*. https://arxiv.org/abs/2402.01035 ↑ back: 1, 2

达根，G.、辛纳夫，G. 和罗齐埃，B.（2024）。充分利用分词器进行预训练和域适配。

20. Dao, T. & Gu, A. (2024). *Transformers are SSMs: Generalized Models and Efficient Algorithms through Structured State Space Duality*. https://arxiv.org/abs/2405.21060 ↑ back: 1, 2

Dao, T. 和 Gu, A.（2024）。Transformer 是 SSM：通过结构化状态空间二元性的广义模型和高效算法。https://arxiv.org/abs/2405.21060 背：1、2

21. DeepSeek-AI. (2025). *DeepSeek-V3.2-Exp: Boosting Long-Context Efficiency with DeepSeek Sparse Attention*. DeepSeek. https://github.com/deepseek-ai/DeepSeek-V3.2-Exp/blob/main/DeepSeek_V3_2.pdf

DeepSeek-人工智能。（2025）. DeepSeek-V3.2-Exp：利用 DeepSeek 稀疏注意力提高长上下文效率。深度搜索。https://github.com/deepseek-ai/DeepSeek-V3.2-Exp/blob/main/DeepSeek_V3_2.pdf

↑

22. DeepSeek-AI, :, Bi, X., Chen, D., Chen, G., Chen, S., Dai, D., Deng, C., Ding, H., Dong, K., Du, Q., Fu, Z., Gao, H., Gao, K., Gao, W., Ge, R., Guan, K., Guo, D., Guo, J., ... Zou, Y. (2024). *DeepSeek LLM: Scaling Open-Source Language Models with Longtermism*. https://arxiv.org/abs/2401.02954 ↑ back: 1, 2

DeepSeek-AI, :, Bi, X., Chen, D., Chen, G., Chen, S., Dai, D., 邓 C., Ding, H., Dong, K., Du, Q., Fu, Z., Gao, H., Gao, K., Gao, W., Ge, R., Guan, K., Guo, D., Guo, J., ...邹, Y.（2024）。DeepSeek LLM：以长期主义扩展开源语言模型。https://arxiv.org/abs/2401.02954 背：1、2

23. DeepSeek-AI, Guo, D., Yang, D., Zhang, H., Song, J., Zhang, R., Xu, R., Zhu, Q., Ma, S., Wang, P., Bi, X., Zhang, X., Yu, X., Wu, Y., Wu, Z. F., Gou, Z., Shao, Z., Li, Z., Gao, Z., ... Zhang, Z. (2025). *DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning.* https://arxiv.org/abs/2501.12948

DeepSeek-AI, Guo, D., Yang, D., Zhang, H., Song, J., Zhang, R., Xu, R., Zhu, Q., 马, S., Wang, P., Bi, X., Zhang, X., Yu, X., Wu, Y., Wu, Z. F., Gou, Z., Shao, Z., Li, Z., Gao, Z., ...张, Z.（2025）。DeepSeek-R1：通过强化学习激励法学硕士的推理能力。https://arxiv.org/abs/2501.12948

↑

24. DeepSeek-AI, Liu, A., Feng, B., Wang, B., Wang, B., Liu, B., Zhao, C., Dengr, C., Ruan, C., Dai, D., Guo, D., Yang, D., Chen, D., Ji, D., Li, E., Lin, F., Luo, F., Hao, G., Chen, G., ... Xie, Z. (2024). *DeepSeek-V2: A Strong, Economical, and Efficient Mixture-of-Experts Language Model.* https://arxiv.org/abs/2405.04434↑ back: 1, 2

DeepSeek-AI, Liu, A., Feng, B., Wang, B., Wang, B., Liu, B., Zhao, C., Dengr, C., Ruan, C., Dai, D., Guo, D., Yang, D., Chen, D., Ji, D., Li, E., Lin, F., Luo, F., Hao, G., Chen, G., ...谢 Z.（2024）。DeepSeek-V2：一种强大、经济且高效的专家混合语言模型。https://arxiv.org/abs/2405.04434 背：1、2

25. DeepSeek-AI, Liu, A., Feng, B., Xue, B., Wang, B., Wu, B., Lu, C., Zhao, C., Deng, C., Zhang, C., Ruan, C., Dai, D., Guo, D., Yang, D., Chen, D., Ji, D., Li, E., Lin, F., Dai, F., ... Pan, Z. (2025). *DeepSeek-V3 Technical Report.* https://arxiv.org/abs/2412.19437

DeepSeek-AI, Liu, A., Feng, B., Xue, B., Wang, B., Wu, B., Lu, C., Zhao, C., 邓 C., Zhang, C., Ruan, C., Dai, D., Guo, D., Yang, D., Chen, D., Ji, D., Li, E., Lin, F., Dai, F., ...潘, Z.（2025）。DeepSeek-V3 技术报告。https://arxiv.org/abs/2412.19437

↑

26. Dehghani, M., Djolonga, J., Mustafa, B., Padlewski, P., Heek, J., Gilmer, J., Steiner, A., Caron, M., Geirhos, R., Alabdulmohsin, I., Jenatton, R., Beyer, L., Tschannen, M., Arnab, A., Wang, X., Riquelme, C., Minderer, M., Puigcerver, J., Evci, U., ... Houlsby, N. (2023). *Scaling Vision Transformers to 22 Billion Parameters.* https://arxiv.org/abs/2302.05442

Dehghani, M., Djolonga, J., Mustafa, B., Padlewski, P., Heek, J., Gilmer, J., Steiner, A.

Caron, M., Geirhos, R., Alabdulmohsin, I., Jenatton, R., Beyer, L., Tschannen, M., Arnab, A., Wang, X., Riquelme, C., Minderer, M., Puigcerver, J., Evci, U., ...霍尔斯比, N. (2023)。将视觉转换器扩展到 220 亿个参数。https://arxiv.org/abs/2302.05442

↑

27. Ding, H., Wang, Z., Paolini, G., Kumar, V., Deoras, A., Roth, D., & Soatto, S. (2024). *Fewer Truncations Improve Language Modeling.* https://arxiv.org/abs/2404.10830

丁 H.、王 Z.、保利尼 G.、库马尔 V.、德奥拉斯 A.、罗斯 D. 和索阿托 S.（2024）。更少的截断改进了语言建模。https://arxiv.org/abs/2404.10830

↑

28. D'Oosterlinck, K., Xu, W., Develder, C., Demeester, T., Singh, A., Potts, C., Kiela, D., & Mehri, S. (2024). *Anchored Preference Optimization and Contrastive Revisions: Addressing Underspecification in Alignment.* https://arxiv.org/abs/2408.06266

D'Oosterlinck，K.、Xu，W.、Develder，C.、Demeester，T.、Singh，A.、Potts，C.、Kiela，D. 和 Mehri，S.（2024）。锚定偏好优化和对比修订：解决一致性中的规格不足问题。https://arxiv.org/abs/2408.06266

↑

29. Du, Z., Zeng, A., Dong, Y., & Tang, J. (2025). *Understanding Emergent Abilities of Language Models from the Loss Perspective.* https://arxiv.org/abs/2403.15796↑ back: 1, 2

杜 Z.、曾 A.、董 Y. 和唐 J.（2025）。从损失的角度理解语言模型的涌现能力。https://arxiv.org/abs/2403.15796 背：1、2

30. Dubois, Y., Galambosi, B., Liang, P., & Hashimoto, T. B. (2025). *Length-Controlled AlpacaEval: A Simple Way to Debias Automatic Evaluators.* https://arxiv.org/abs/2404.04475

Dubois，Y.、Galambosi，B.、Liang，P. 和 Hashimoto，TB（2025）。长度控制的 AlpacaEval：一种对自动评估器进行去偏差的简单方法。https://arxiv.org/abs/2404.04475

↑

31. Ethayarajh, K., Xu, W., Muennighoff, N., Jurafsky, D., & Kiela, D. (2024). *KTO: Model Alignment as Prospect Theoretic Optimization.* https://arxiv.org/abs/2402.01306

Ethayarajh，K.、Xu，W.、Muennighoff，N.、Jurafsky，D. 和 Kiela，D.（2024）。KTO：模型对齐作为前景理论优化。https://arxiv.org/abs/2402.01306

↑

32. Gandhi, K., Chakravarthy, A., Singh, A., Lile, N., & Goodman, N. D. (2025). *Cognitive Behaviors that Enable Self-Improving Reasoners, or, Four Habits of Highly Effective STaRs.* https://arxiv.org/abs/2503.01307

甘地，K.、查克拉瓦蒂，A.、辛格，A.、莱尔，N. 和古德曼，ND（2025）。能够自我完善推理的认知行为，或者高效 STaR 的四种习惯。https://arxiv.org/abs/2503.01307

↑

33. Gao, T., Wettig, A., Yen, H., & Chen, D. (2025). *How to Train Long-Context Language Models (Effectively).* https://arxiv.org/abs/2410.02660↑ back: 1, 2, 3

高 T.、韦蒂格 A.、Yen，H. 和陈 D.（2025）。如何（有效）训练长上下文语言模型。https://arxiv.org/abs/2410.02660 背：1、2、3

34. Grattafiori, A., Dubey, A., Jauhri, A., Pandey, A., Kadian, A., Al-Dahle, A., Letman, A., Mathur, A., Schelten, A.,

34. Grattafiori, A., Dubey, A., Jauhri, A., Pandey, A., Kadian, A., Al-Dahle, A., Letman, A., Mathur, A., Schelten, A., Vaughan, A., Yang, A., Fan, A., Goyal, A., Hartshorn, A., Yang, A., Mitra, A., Sravankumar, A., Korenev, A., Hinsvark, A., ... Ma, Z. (2024). *The Llama 3 Herd of Models.* https://arxiv.org/abs/2407.21783↑ back: 1, 2, 3

Grattafiori，A.、Dubey，A.、Jauhri，A.、Pandey，A.、Kadian，A.、Al-Dahle，A.、Letman，A.、Mathur，A.、Schelten，A.、Vaughan，A.、Yang，A.、Fan，A.、Goyal，A.、Hartshorn，A.、Yang，A.、Mitra，A.、Sravankumar，A.、Korenev，A.、Hinsvark，A.、...马，Z.（2024）。骆驼 3 模型群。https://arxiv.org/abs/2407.21783 背：1、2、3

35. Gu, A., & Dao, T. (2024). *Mamba: Linear-Time Sequence Modeling with Selective State Spaces.* https://arxiv.org/abs/2312.00752

顾 A. 和道 T.（2024）。Mamba：具有选择性状态空间的线性时间序列建模。https://arxiv.org/abs/2312.00752↑

36. Gu, Y., Tafjord, O., Kuehl, B., Haddad, D., Dodge, J., & Hajishirzi, H. (2025). *OLMES: A Standard for Language Model Evaluations.* https://arxiv.org/abs/2406.08446↑ back: 1, 2

Gu，Y.、Tafjord，O.、Kuehl，B.、Haddad，D.、Dodge，J. 和 Hajishirzi，H.（2025）。OLMES：语言模型评估标准。https://arxiv.org/abs/2406.08446 背：1、2

37. Guo, S., Zhang, B., Liu, T., Liu, T., Khalman, M., Llinares, F., Rame, A., Mesnard, T., Zhao, Y., Piot, B., Ferret, J., & Blondel, M. (2024). *Direct Language Model Alignment from Online AI Feedback.* https://arxiv.org/abs/2402.04792

郭 S.、张 B.、刘 T.、刘 T.、哈尔曼 M.、利纳雷斯 F.、拉梅 A.、梅斯纳德 T.、赵 Y.、皮奥特 B.、雪貂 J. 和布隆德尔 M.（2024）。来自在线 AI 反馈的直接语言模型对齐。https://arxiv.org/abs/2402.04792

↑

38. Hägele, A., Bakouch, E., Kosson, A., Allal, L. B., Werra, L. V., & Jaggi, M. (2024). *Scaling Laws and Compute-Optimal Training Beyond Fixed Training Durations.* https://arxiv.org/abs/2405.18392↑ back: 1, 2

Hägele，A.、Bakouch，E.、Kosson，A.、Allal，L. B.、Werra，LV 和 Jaggi，M.（2024）。扩展定律和计算优化训练超出固定训练持续时间。https://arxiv.org/abs/2405.18392 背：1、2

39. He, Y., Jin, D., Wang, C., Bi, C., Mandyam, K., Zhang, H., Zhu, C., Li, N., Xu, T., Lv, H., Bhosale, S., Zhu, C., Sankararaman, K. A., Helenowski, E., Kambadur, M., Tayade, A., Ma, H., Fang, H., & Wang, S. (2024). *Multi-IF: Benchmarking LLMs on Multi-Turn and Multilingual Instructions Following.* https://arxiv.org/abs/2410.15553

He，Y.、Jin，D.、Wang，C.、Bi，C.、Mandyam，K.、Zhang，H.、Zhu，C.、Li，N.、Xu，T.、Lv，H.、Bhosale，S.、Zhu，C.、Sankararaman，KA、Helenowski，E.、Kambadur，M.、Tayade，A.、马，H.、方，H. 和王，S.（2024）。Multi-IF：对 LLM 进行多轮次和多语言指令的基准测试。https://arxiv.org/abs/2410.15553

↑

40. Hoffmann, J., Borgeaud, S., Mensch, A., Buchatskaya, E., Cai, T., Rutherford, E., de Las Casas, D., Hendricks, L. A., Welbl, J., Clark, A., Hennigan, T., Noland, E., Millican, K., van den Driessche, G., Damoc, B., Guy, A., Osindero, S., Simonyan, K., Elsen, E., ... Sifre, L.

Hoffmann，J.、Borgeaud，S.、Mensch，A.、Buchatskaya，E.、Cai，T.、Rutherford，E.、de Las Casas，D.、Hendricks，L. A.、Welbl，J.、Clark，A.、Hennigan，T.、Noland，E.、Millican，K.、van den Driessche，G.、Damoc，B.、Guy，A.、Osindero，S.、Simonyan，K.、Elsen，E.、...西弗雷，L.

(2022). *Training Compute-Optimal Large Language Models.* https://arxiv.org/abs/2203.15556

(2022).训练计算最优的大型语言模型。https://arxiv.org/abs/2203.15556↑

41. Hong, J., Lee, N., & Thorne, J. (2024). *ORPO: Monolithic Preference Optimization without Reference Model.* https://arxiv.org/abs/2403.07691

洪，J.、李，N.、和索恩，J.（2024）。ORPO：无参考模型的单片偏好优化。https://arxiv.org/abs/2403.07691↑

42. Howard, J., & Ruder, S. (2018). *Universal Language Model Fine-tuning for Text Classification.* https://arxiv.org/abs/1801.06146

霍华德，J.，和鲁德，S.（2018）。用于文本分类的通用语言模型微调。https://arxiv.org/abs/1801.06146↑

43. Hsieh, C.-P., Sun, S., Kriman, S., Acharya, S., Rekesh, D., Jia, F., Zhang, Y., & Ginsburg, B. (2024). *RULER: What's the Real Context Size of Your Long-Context Language Models?* https://arxiv.org/abs/2404.06654↑ back: 1, 2

谢 C.-P.、孙 S.、克里曼 S.、阿查里亚 S.、雷克什 D.、贾 F.、张 Y. 和金斯伯格 B.（2024）。RULER：你们的长上下文语言模型的真实上下文大小是多少？https://arxiv.org/abs/2404.06654 背：1、2

44. Hu, S., Tu, Y., Han, X., He, C., Cui, G., Long, X., Zheng, Z., Fang, Y., Huang, Y., Zhao, W., Zhang, X., Thai, Z. L., Zhang, K., Wang, C., Yao, Y., Zhao, C., Zhou, J., Cai, J., Zhai, Z., ... Sun, M. (2024). *MiniCPM: Unveiling the Potential of Small Language Models with Scalable Training Strategies.* https://arxiv.org/abs/2404.06395

胡 S.、涂 Y.、韩 X.、何 C.、崔 G.、龙 X.、郑 Z.、方 Y.、黄 Y.、赵 W.、张 X.、泰国 Z. L.、张 K.、王 C.、姚 Y.、赵 C.、周 J.、蔡 J.、翟 Z.、...孙，M.（2024）。MiniCPM：通过可扩展的训练策略揭示小型语言模型的潜力。https://arxiv.org/abs/2404.06395

↑

45. Huang, S., Noukhovitch, M., Hosseini, A., Rasul, K., Wang, W., & Tunstall, L. (2024). *The N+ Implementation Details of RLHF with PPO: A Case Study on TL;DR Summarization.* https://arxiv.org/abs/2403.17031

Huang，S.、Noukhovitch，M.、Hosseini，A.、Rasul，K.、Wang，W. 和 Tunstall，L.（2024）。RLHF 与 PPO 的 N+ 实施细节：TL 案例研究;DR 总结。https://arxiv.org/abs/2403.17031

↑

46. IBM Research. (2025). *IBM Granite 4.0: Hyper-efficient, High Performance Hybrid Models for Enterprise.* https://www.ibm.com/new/announcements/ibm-granite-4-0-hyper-efficient-high-performance-hybrid-models

IBM 研究院。（2025）. IBM Granite 4.0：面向企业的超高效、高性能混合模型。https://www.ibm.com/new/announcements/ibm-granite-4-0-hyper-efficient-high-performance-hybrid-models

↑

47. Jiang, A. Q., Sablayrolles, A., Mensch, A., Bamford, C., Chaplot, D. S., de las Casas, D., Bressand, F., Lengyel, G., Lample, G., Saulnier, L., Lavaud, L. R., Lachaux, M.-A., Stock, P., Scao, T. L., Lavril, T., Wang, T., Lacroix, T., & Sayed, W. E. (2023). *Mistral 7B*. https://arxiv.org/abs/2310.06825

江 A. Q.、萨布莱罗尔斯 A.、门施 A.、班福德 C.、查普洛特 DS、德拉斯卡萨斯 D.、布雷曼德 F.、伦耶尔 G.、兰普尔 G.、索尔尼尔 L.、拉沃 LR、拉肖马萨诺塞州-A.、斯托克 P.、斯考 TL、拉夫里尔 T.、王 T.、拉克鲁瓦 T. 和赛义德 WE（2023）。米斯特拉尔 7B. https://arxiv.org/abs/2310.06825

↑

48. Kamradt, G. (2023). Needle In A Haystack - pressure testing LLMs. In *GitHub repository*. GitHub. https://github.com/gkamradt/LLMTest_NeedleInAHaystack↑ back: 1, 2

卡姆拉特，G.（2023）。大海捞针 - 压力测试法学硕士。在 GitHub 存储库中。GitHub 中。https://github.com/gkamradt/LLMTest_NeedleInAHaystack 背：1、2

49. Kaplan, J., McCandlish, S., Henighan, T., Brown, T. B., Chess, B., Child, R., Gray, S., Radford, A., Wu, J., & Amodei, D. (2020). *Scaling Laws for Neural Language Models*. https://arxiv.org/abs/2001.08361

Kaplan，J.、McCandlish，S.、Henighan，T.、Brown，TB、Chess，B.、Child，R.、Gray，S.、Radford，A.、Wu，J. 和 Amodei，D.（2020）。神经语言模型的缩放定律。https://arxiv.org/abs/2001.08361

↑

50. Katsch, T. (2024). *GateLoop: Fully Data-Controlled Linear Recurrence for Sequence Modeling*. https://arxiv.org/abs/2311.01927

卡奇，T.（2024）。GateLoop：用于序列建模的完全数据控制的线性递归。https://arxiv.org/abs/2311.01927↑

51. Kazemnejad, A., Padhi, I., Ramamurthy, K. N., Das, P., & Reddy, S. (2023). *The Impact of Positional Encoding on Length Generalization in Transformers*. https://arxiv.org/abs/2305.19466

Kazemnejad，A.、Padhi，I.、Ramamurthy，KN、Das，P. 和 Reddy，S.（2023）。位置编码对变压器长度泛化的影响。https://arxiv.org/abs/2305.19466

↑

52. Khatri, D., Madaan, L., Tiwari, R., Bansal, R., Duvvuri, S. S., Zaheer, M., Dhillon, I. S., Brandfonbrener, D., & Agarwal, R. (2025). *The Art of Scaling Reinforcement Learning Compute for LLMs*. https://arxiv.org/abs/2510.13786↑ back: 1, 2

Khatri，D.、Madaan，L.、Tiwari，R.、Bansal，R.、Duvvuri，SS、Zaheer，M.、Dhillon，IS、Brandfonbrener，D. 和 Agarwal，R.（2025）。LLM 扩展强化学习计算的艺术。https://arxiv.org/abs/2510.13786 返回：1、2

53. Kingma, D. P. (2014). Adam: A method for stochastic optimization. *arXiv Preprint arXiv:1412.6980*.

金马，DP（2014）。Adam：一种随机优化的方法。arXiv 预印本 arXiv：1412.6980。↑

54. Krajewski, J., Ludziejewski, J., Adamczewski, K., Pióro, M., Krutul, M., Antoniak, S., Ciebiera, K., Król, K., Odrzygóźdź, T., Sankowski, P., Cygan, M., & Jaszczur, S. (2024). *Scaling Laws for Fine-Grained Mixture of Experts*. https://arxiv.org/abs/2402.07871

Krajewski，J.、Ludziejewski，J.、Adamczewski，K.、Pióro，M.、Krutul，M.、Antoniak，S.、Ciebiera，K.、Król，K.、Odrzygóźdź，T.、Sankowski，P.、Cygan，M. 和 Jaszczur，S.（2024）。专家细粒度混合物的缩放定律。https://arxiv.org/abs/2402.07871

↑

55. Lambert, N., Castricato, L., von Werra, L., & Havrilla, A. (2022). Illustrating Reinforcement Learning from Human Feedback (RLHF). *Hugging Face Blog*.

兰伯特，N.，卡斯特里卡托，L.，冯·韦拉，L.，和哈夫里拉，A.（2022）。说明来自人类反馈的强化学习（RLHF）。拥抱脸博客。

↑

56. Lambert, N., Morrison, J., Pyatkin, V., Huang, S., Ivison, H., Brahman, F., Miranda, L. J. V., Liu, A., Dziri, N., Lyu, S., Gu, Y., Malik, S., Graf, V., Hwang, J. D., Yang, J., Bras, R. L., Tafjord, O., Wilhelm, C., Soldaini, L., ... Hajishirzi, H. (2025). *Tulu 3: Pushing Frontiers in Open Language Model Post-Training*. https://arxiv.org/abs/2411.15124

Lambert，N.，Morrison，J.，Pyatkin，V.，Huang，S.，Ivison，H.，Brahman，F.，Miranda，L.J.V.，Liu，A.，Dziri，N.，Lyu，S.，Gu，Y.，Malik，S.，Graf，V.，Hwang，JD，Yang，J.，Bras，RL，Tafjord，O.，Wilhelm，C.，Soldaini，L.，...哈吉希尔齐，H.（2025）。Tulu 3：推动开放语言模型后训练的前沿。https://arxiv.org/abs/2411.15124

↑

57. Lanchantin, J., Chen, A., Lan, J., Li, X., Saha, S., Wang, T., Xu, J., Yu, P., Yuan, W., Weston, J. E., Sukhbaatar, S., & Kulikov, I. (2025). *Bridging Offline and Online Reinforcement Learning for LLMs*. https://arxiv.org/abs/2506.21495

Lanchantin，J.、Chen，A.、Lan，J.、Li，X.、Saha，S.、Wang，T.、Xu，J.、Yu，P.、Yuan，W.、Weston，JE、Sukhbaatar，S. 和 Kulikov，I.（2025）。为法学硕士架起离线和在线强化学习的桥梁 https://arxiv.org/abs/2506.21495。

↑

58. Li, J., Fang, A., Smyrnis, G., Ivgi, M., Jordan, M., Gadre, S., Bansal, H., Guha, E., Keh, S., Arora, K., Garg, S., Xin, R., Muennighoff, N., Heckel, R., Mercat, J., Chen, M., Gururangan, S., Wortsman, M., Albalak, A., ... Shankar, V. (2025). *DataComp-LM: In search of the next generation of training sets for language models*. https://arxiv.org/abs/2406.11794

Li，J.，Fang，A.，Smyrnis，G.，Ivgi，M.，Jordan，M.，Gadre，S.，Bansal，H.，Guha，E.，Keh，S.，Arora，K.，Garg，S.，Xin，R.，Muennighoff，N.，Heckel，R.，Mercat，J.，Chen，M.，Gururangan，S.，Wortsman，M.，Albalak，A.，...香卡，V.（2025）。DataComp-LM：寻找语言模型的下一代训练集。https://arxiv.org/abs/2406.11794

↑

59. Li, Q., Cui, L., Zhao, X., Kong, L., & Bi, W. (2024). *GSM-Plus: A Comprehensive Benchmark for Evaluating the Robustness of LLMs as Mathematical Problem Solvers*. https://arxiv.org/abs/2402.19255

李 Q.、崔 L.、赵 X.、孔 L 和毕 W.（2024）。GSM-Plus：评估法学硕士作为数学问题解决者的稳健性的综合基准。https://arxiv.org/abs/2402.19255

↑

60. Li, R., Allal, L. B., Zi, Y., Muennighoff, N., Kocetkov, D., Mou, C., Marone, M., Akiki, C., Li, J., Chim, J., Liu, Q., Zheltonozhskii, E., Zhuo, T. Y., Wang, T., Dehaene, O., Davaadorj, M., Lamy-Poirier, J., Monteiro, J., Shliazhko, O., ... de Vries, H. (2023). *StarCoder: may the source be with you!* https://arxiv.org/abs/2305.06161

Li，R.，Allal，L.B.，Zi，Y.，Muennighoff，N.，Kocetkov，D.，Mou，C.，Marone，M.，Akiki，C.，Li，J.，Chim，J.，Liu，Q.，Zheltonozhskii，E.，Zhuo，T.Y.，Wang，T.，Dehaene，O.，Davaadorj，M.，Lamy-Poirier，J.，Monteiro，J.，Shliazhko，O.，...德弗里斯，H.（2023）。StarCoder：愿源头与你同在！https://arxiv.org/abs/2305.06161

↑

61. Li, T., Chiang, W.-L., Frick, E., Dunlap, L., Wu, T., Zhu, B., Gonzalez, J. E., & Stoica, I. (2024). *From Crowdsourced Data to High-Quality Benchmarks: Arena-Hard and BenchBuilder Pipeline*. https://arxiv.org/abs/2406.11939

李 T.、蒋 WL、弗里克 E.、邓拉普 L.、吴 T.、朱 B.、冈萨雷斯 JE 和斯托伊卡 I.（2024）。从众包数据到高质量基准测试：Arena-Hard 和 BenchBuilder Pipeline。https://arxiv.org/abs/2406.11939

↑

62. Liang, W., Liu, T., Wright, L., Constable, W., Gu, A., Huang, C.-C., Zhang, I., Feng, W., Huang, H., Wang, J., Purandare, S., Nadathur, G., & Idreos, S. (2025). *TorchTitan: One-stop PyTorch native solution for production ready LLM pre-training*. https://arxiv.org/abs/2410.06511

63. Lightman, H., Kosaraju, V., Burda, Y., Edwards, H., Baker, B., Lee, T., Leike, J., Schulman, J., Sutskever, I., & Cobbe, K. (2023). *Let's Verify Step by Step*. https://arxiv.org/abs/2305.20050

Lightman，H.、Kosaraju，V.、Burda，Y.、Edwards，H.、Baker，B.、Lee，T.、Leike，J.、Schulman，J.、Sutskever，I. 和 Cobbe，K.（2023）。让我们一步一步验证一下。https://arxiv.org/abs/2305.20050

↑

64. Liu, H., Xie, S. M., Li, Z., & Ma, T. (2022). *Same Pre-training Loss, Better Downstream: Implicit Bias Matters for Language Models*. https://arxiv.org/abs/2210.14199

刘 H.、谢 SM、李 Z. 和马 T.（2022）。相同的预训练损失，更好的下游：隐式偏差对语言模型很重要。https://arxiv.org/abs/2210.14199

↑

65. Liu, Q., Zheng, X., Muennighoff, N., Zeng, G., Dou, L., Pang, T., Jiang, J., & Lin, M. (2025). *RegMix: Data Mixture as Regression for Language Model Pre-training*. https://arxiv.org/abs/2407.01492

Liu，Q.，Zheng，X.，Muennighoff，N.，Zeng，G.，Dou，L.，Pang，T.，江 J. 和林 M.（2025）。RegMix：数据混合作为语言模型预训练的回归。https://arxiv.org/abs/2407.01492

↑

66. Liu, Z., Zhao, C., Iandola, F., Lai, C., Tian, Y., Fedorov, I., Xiong, Y., Chang, E., Shi, Y., Krishnamoorthi, R., Lai, L., & Chandra, V. (2024). *MobileLLM: Optimizing Sub-billion Parameter Language Models for On-Device Use Cases*. https://arxiv.org/abs/2402.14905

Liu，Z.，Zhao，C.，Iandola，F.，Lai，C.，Tian，Y.，Fedorov，I.，Xiong，Y.，Chang，E.，Shi，Y.，Krishnamoorthi，R.，Lai，L. 和 Chandra，V.（2024）。MobileLLM：针对设备上的用例优化十亿以下参数语言模

型。https://arxiv.org/abs/2402.14905

↑

67. Loshchilov, I., & Hutter, F. (2017). *SGDR: Stochastic Gradient Descent with Warm Restarts.* https://arxiv.org/abs/1608.03983

Loshchilov，I. 和 Hutter，F.（2017）。SGDR：具有热重启的随机梯度下降。https://arxiv.org/abs/1608.03983↑

68. Lozhkov, A., Li, R., Allal, L. B., Cassano, F., Lamy-Poirier, J., Tazi, N., Tang, A., Pykhtar, D., Liu, J., Wei, Y., Liu, T., Tian, M., Kocetkov, D., Zucker, A., Belkada, Y., Wang, Z., Liu, Q., Abulkhanov, D., Paul, I., ... de Vries, H. (2024). *StarCoder 2 and The Stack v2: The Next Generation.* https://arxiv.org/abs/2402.19173

Lozhkov，A.，Li，R.，Allal，L. B.，Cassano，F.，Lamy-Poirier，J.，Tazi，N.，Tang，A.，Pykhtar，D.，Liu，J.，Wei，Y.，Liu，T.，Tian，M.，Kocetkov，D.，Zucker，A.，Belkada，Y.，Wang，Z.，Liu，Q.，Abulkhanov，D.，Paul，I.，...德弗里斯，H.（2024）。StarCoder 2 和 The Stack v2：下一代。https://arxiv.org/abs/2402.19173

↑

69. Mao, H. H. (2022). *Fine-Tuning Pre-trained Transformers into Decaying Fast Weights.* https://arxiv.org/abs/2210.04243

毛，HH（2022）。将预训练的 Transformer 微调为衰减的快速权重。https://arxiv.org/abs/2210.04243↑

70. Marafioti, A., Zohar, O., Farré, M., Noyan, M., Bakouch, E., Cuenca, P., Zakka, C., Allal, L. B., Lozhkov, A., Tazi, N., Srivastav, V., Lochner, J., Larcher, H., Morlon, M., Tunstall, L., von Werra, L., & Wolf, T. (2025). *SmolVLM: Redefining small and efficient multimodal models.* https://arxiv.org/abs/2504.05299

Marafioti，A.，Zohar，O.，Farré，M.，Noyan，M.，Bakouch，E.，Cuenca，P.，Zakka，C.，Allal，L. B.，Lozhkov，A.，Tazi，N.，Srivastav，V.，Lochner，J.，Larcher，H.，Morlon，M.，Tunstall，L.，von Werra，L. 和 Wolf，T.（2025）。SmolVLM：重新定义小型高效多模态模型。https://arxiv.org/abs/2504.05299

↑

71. McCandlish, S., Kaplan, J., Amodei, D., & Team, O. D. (2018). *An Empirical Model of Large-Batch Training.* https://arxiv.org/abs/1812.06162

McCandlish，S.，Kaplan，J.，Amodei，D. 和 Team，OD（2018）。大批量训练的经验模型.https://arxiv.org/abs/1812.06162

↑

72. Merrill, W., Arora, S., Groeneveld, D., & Hajishirzi, H. (2025). *Critical Batch Size Revisited: A Simple Empirical Approach to Large-Batch Language Model Training.* https://arxiv.org/abs/2505.23971

Merrill，W.，Arora，S.，Groeneveld，D. 和 Hajishirzi，H.（2025）。重温临界批量大小：大批量语言模型训练的简单经验方法。https://arxiv.org/abs/2505.23971

↑

73. Meta AI. (2025). *The Llama 4 herd: The beginning of a new era of natively multimodal AI innovation.* https://ai.meta.com/blog/llama-4-multimodal-intelligence/↑ back: 1, 2

元人工智能。（2025）。鸵鸟 4 群：原生多模态人工智能创新新时代的开始。https://ai.meta.com/blog/llama-4-multimodal-intelligence/背：1,2

74. Mindermann, S., Brauner, J., Razzak, M., Sharma, M., Kirsch, A., Xu, W., Höltgen, B., Gomez, A. N., Morisot, A., Farquhar, S., & Gal, Y. (2022). *Prioritized Training on Points that are Learnable, Worth Learning, and Not Yet Learnt.* https://arxiv.org/abs/2206.07137

Mindermann，S.，Brauner，J.，Razzak，M.，Sharma，M.，Kirsch，A.，Xu，W.，Höltgen，B.，Gomez，AN.，Morisot，A.，Farquhar，S. 和 Gal，Y.（2022）。优先培训可学习、值得学习和尚未学习的点。https://arxiv.org/abs/2206.07137

↑

75. MiniMax, Li, A., Gong, B., Yang, B., Shan, B., Liu, C., Zhu, C., Zhang, C., Guo, C., Chen, D., Li, D., Jiao, E., Li, G., Zhang, G., Sun, H., Dong, H., Zhu, J., Zhuang, J., Song, J., ... Wu, Z. (2025). *MiniMax-01: Scaling Foundation Models with Lightning Attention.* https://arxiv.org/abs/2501.08313↑ back: 1, 2, 3

MiniMax，Li，A.，Gong，B.，Yang，B.，Shan，B.，Liu，C.，Zhu，C.，Zhang，C.，Guo，C.，Chen，D.，Li，D.，Jiao，E.，Li，G.，Zhang，G.，Sun，H.，Dong，H.，Zhu，J.，Zhuang，J.，Song，J.，...吴，Z.（2025）。MiniMax-01：使用闪电注意力扩展基础模型。https://arxiv.org/abs/2501.08313背：1、2、3

76. Mistral AI. (2025). *Mistral Small 3.1.* https://mistral.ai/news/mistral-small-3-1

米斯特拉尔人工智能。（2025）。米斯特拉尔小 3.1。https://mistral.ai/news/mistral-small-3-1↑

77. Moshkov, I., Hanley, D., Sorokin, I., Toshniwal, S., Henkel, C., Schifferer, B., Du, W., & Gitman, I. (2025). *AIMO-2 Winning Solution: Building State-of-the-Art Mathematical Reasoning Models with OpenMathReasoning dataset.* https://arxiv.org/abs/2504.16891

莫什科夫，I.、汉利，D.、索罗金，I.、托什尼瓦尔，S.、汉高，C.、希弗勒，B.、杜，W. 和吉特曼，I.（2025）。AIMO-2 成功解决方案：使用 OpenMathReasoning 数据集构建最先进的数学推理模型。https://arxiv.org/abs/2504.16891

↑

78. Muennighoff, N., Rush, A. M., Barak, B., Scao, T. L., Piktus, A., Tazi, N., Pyysalo, S., Wolf, T., & Raffel, C. (2025). *Scaling Data-Constrained Language Models.* https://arxiv.org/abs/2305.16264

Muennighoff，N.，Rush，AM.，Barak，B.，Scao，TL.，Piktus，A.，Tazi，N.，Pyysalo，S.，Wolf，T. 和 Raffel，C.（2025）。扩展数据受限的语言模型。https://arxiv.org/abs/2305.16264

↑

79. Ni, J., Xue, F., Yue, X., Deng, Y., Shah, M., Jain, K., Neubig, G., & You, Y. (2024). *MixEval: Deriving Wisdom of the Crowd from LLM Benchmark Mixtures.* https://arxiv.org/abs/2406.06565

倪 J.、薛 F.、岳 X.、邓 Y.、沙阿 M.、詹那敦 K.、纽比格 G. 和尤 Y.（2024）。MixEval：从 LLM 基准混合物中得出人群的智慧。https://arxiv.org/abs/2406.06565

↑

80. Nrusimha, A., Brandon, W., Mishra, M., Shen, Y., Panda, R., Ragan-Kelley, J., & Kim, Y. (2025). *FlashFormer: Whole-Model Kernels for Efficient Low-Batch Inference.* https://arxiv.org/abs/2505.22758

Nrusimha，A.，Brandon，W.，Mishra，M.，Shen，Y.，Panda，R.，Ragan-Kelley，J. 和 Kim，Y.（2025）。FlashFormer：用于高效低批量推理的全模型内核。https://arxiv.org/abs/2505.22758

↑

81. Nvidia, :, Adler, B., Agarwal, N., Aithal, A., Anh, D. H., Bhattacharya, P., Brundyn, A., Casper, J., Catanzaro, B., Clay, S., Cohen, J., Das, S., Dattagupta, A., Delalleau, O., Derczynski, L., Dong, Y., Egert, D., Evans, E., ... Zhu, C. (2024). *Nemotron-4 340B Technical Report.* https://arxiv.org/abs/2406.11704

英伟达，：，阿德勒，B.，阿加瓦尔，N.，艾塔尔，A.，安，DH，巴塔查里亚，P.，布伦丁，A.，卡斯珀，J.，卡坦扎罗，B.，克莱，S.，科恩，J.，达斯，S.，达古普塔，A.，德拉洛，O.，德钦斯基，L.，董，Y.，埃格特，D.，埃文斯，E.，...朱，C.（2024）。Nemotron-4 340B 技术报告。https://arxiv.org/abs/2406.11704

↑

82. NVIDIA, :, Basant, A., Khairnar, A., Paithankar, A., Khattar, A., Renduchintala, A., Malte, A., Bercovich, A., Hazare, A., Rico, A., Ficek, A., Kondratenko, A., Shaposhnikov, A., Bukharin, A., Taghibakhshi, A., Barton, A., Mahabaleshwarkar, A. S., Shen, A., ... Chen, Z. (2025). *NVIDIA Nemotron Nano 2: An Accurate and Efficient Hybrid Mamba-Transformer Reasoning Model.* https://arxiv.org/abs/2508.14444

NVIDIA，：，Basant，A.，Khairnar，A.，Paithankar，A.，Khattar，A.，Renduchintala，A.，Malte，A.，Bercovich，A.，Hazare，A.，Rico，A.，Ficek，A.，Kondratenko，A.，Shaposhnikov，A.，Bukharin，A.，Taghibakhshi，A.，Barton，A.，Mahabaleshwarkar，AS，Shen，A.，...陈，Z.（2025）。NVIDIA Nemotron Nano 2：准确高效的混合曼巴-变压器推理模型。https://arxiv.org/abs/2508.14444

↑

83. NVIDIA, :, Blakeman, A., Basant, A., Khattar, A., Renduchintala, A., Bercovich, A., Ficek, A., Bjorlin, A., Taghibakhshi, A., Deshmukh, A. S., Mahabaleshwarkar, A. S., Tao, A., Shors, A., Aithal, A., Poojary, A., Dattagupta, A., Buddharaju, B., Chen, B., ... Chen, Z. (2025). *Nemotron-H: A Family of Accurate and Efficient Hybrid Mamba-Transformer Models.* https://arxiv.org/abs/2504.03624

英伟达，：，布莱克曼，A.，巴桑特，A.，哈塔尔，A.，伦杜钦塔拉，A.，贝尔科维奇，A.，菲切克，A.，比约林，A.，塔吉巴赫希，A.，德什穆克，AS，Mahabaleshwarkar，AS，陶，A.，肖尔斯，A.，艾塔尔，A.，普贾里，A.，达塔古普塔，A.，布达拉朱，B.，陈，B.，...陈，Z.（2025）。Nemotron-H：一系列准确高效的混合曼巴-变压器模型。https://arxiv.org/abs/2504.03624

↑

84. OLMo, T., Walsh, P., Soldaini, L., Groeneveld, D., Lo, K., Arora, S., Bhagia, A., Gu, Y., Huang, S., Jordan, M., Lambert, N., Schwenk, D., Tafjord, O., Anderson, T., Atkinson, D., Brahman, F., Clark, C., Dasigi, P., Dziri, N., ... Hajishirzi, H. (2025). *2 OLMo 2 Furious.* https://arxiv.org/abs/2501.00656↑ back: 1, 2, 3

OLMo，T.，Walsh，P.，Soldaini，L.，Groeneveld，D.，Lo，K.，Arora，S.，Bhagia，A.，Gu，Y.，Huang，S.，Jordan，M.，Lambert，N.，Schwenk，D.，Tafjord，O.，Anderson，T.，Atkinson，D.，Brahman，F.，Clark，C.，Dasigi，P.，Dziri，N.，...哈吉希尔齐，H.（2025）。2 OL 莫 2 愤怒。https://arxiv.org/abs/2501.00656 背：1、2、3

85. OpenAI, Achiam, J., Adler, S., Agarwal, S., Ahmad, L., Akkaya, I., Aleman, F. L., Almeida, D., Altenschmidt, J., Altman, S., Anadkat, S., Avila, R., Babuschkin, I., Balaji, S., Balcom, V., Baltescu, P., Bao, H., Bavarian, M., Belgum, J., ... Zoph, B. (2024). *GPT-4 Technical Report.* https://arxiv.org/abs/2303.08774

OpenAI, Achiam, J., Adler, S., Agarwal, S., Ahmad, L., Akkaya, I., Aleman, FL, Almeida, D., Altenschmidt, J., Altman, S., Anadkat, S., Avila, R., Babuschkin, I., Balaji, S., Balcom, V., Baltescu, P., Bao, H., Bavarian, M., Belgum, J., ...佐夫，B.（2024）。GPT-4 技术报告。https://arxiv.org/abs/2303.08774

↑

86. Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C. L., Mishkin, P., Zhang, C., Agarwal, S., Slama, K., Ray, A., Schulman, J., Hilton, J., Kelton, F., Miller, L., Simens, M., Askell, A., Welinder, P., Christiano, P., Leike, J., & Lowe, R. (2022). *Training language models to follow instructions with human feedback.* https://arxiv.org/abs/2203.02155

欧阳 L.、吴 J.、江 X.、阿尔梅达 D.、温赖特 CL、米什金 P.、张 C.、阿加瓦尔 S.、斯拉马 K.、雷 A.、舒尔曼 J.、希尔顿 J.、凯尔顿 F.、米勒 L.、西门斯 M.、阿斯克尔 A.、韦林德 P.、克里斯蒂亚诺 P.、莱克 J. 和洛 R.（2022）。训练语言模型以遵循人类反馈的指令。https://arxiv.org/abs/2203.02155

↑

87. Penedo, G., Kydlíček, H., allal, L. B., Lozhkov, A., Mitchell, M., Raffel, C., Werra, L. V., & Wolf, T. (2024). *The FineWeb Datasets: Decanting the Web for the Finest Text Data at Scale.* https://arxiv.org/abs/2406.17557

Penedo, G.、Kydlíček, H.、allal, LB、Lozhkov, A.、Mitchell, M.、Raffel, C.、Werra, LV 和 Wolf, T.（2024）。FineWeb 数据集：大规模地对 Web 进行倾析以获得最精细的文本数据。https://arxiv.org/abs/2406.17557

↑

88. Penedo, G., Kydlíček, H., Sabolčec, V., Messmer, B., Foroutan, N., Kargaran, A. H., Raffel, C., Jaggi, M., Werra, L. V., & Wolf, T. (2025). *FineWeb2: One Pipeline to Scale Them All – Adapting Pre-Training Data Processing to Every Language.* https://arxiv.org/abs/2506.20920↑ back: 1, 2

Penedo, G.、Kydlíček, H.、Sabolčec, V.、Messmer, B.、Foroutan, N.、Kargaran, AH、Raffel, C.、Jaggi, M.、Werra, LV 和 Wolf, T.（2025）。FineWeb2：一个管道来扩展它们——使预训练数据处理适应每种语言。https://arxiv.org/abs/2506.20920 背：1、2

89. Peng, B., Goldstein, D., Anthony, Q., Albalak, A., Alcaide, E., Biderman, S., Cheah, E., Du, X., Ferdinan, T., Hou, H., Kazienko, P., GV, K. K., Kocoń, J., Koptyra, B., Krishna, S., Jr, R. M., Lin, J., Muennighoff, N., Obeid, F., ... Zhu, R.-J. (2024). *Eagle and Finch: RWKV with Matrix-Valued States and Dynamic Recurrence.* https://arxiv.org/abs/2404.05892

Peng, B., Goldstein, D., Anthony, Q., Albalak, A., Alcaide, E., Biderman, S., Cheah, E., Du, X., Ferdinan, T., Hou, H., Kazienko, P., GV, K. K., Kocoń, J., Koptyra, B., Krishna, S., Jr, RM, Lin, J., Muennighoff, N., Obeid, F., ...朱，R.-J.（2024）. Eagle 和 Finch：具有矩阵值状态和动态递归的 RWKV。https://arxiv.org/abs/2404.05892

↑

90. Peng, B., Quesnelle, J., Fan, H., & Shippole, E. (2023). *YaRN: Efficient Context Window Extension of Large Language Models.* https://arxiv.org/abs/2309.00071↑ back: 1, 2

彭，B.，克内尔，J.，范，H.，和希普波尔，E.（2023）。YaRN：大型语言模型的高效上下文窗口扩展。https://arxiv.org/abs/2309.00071 背：1、2

91. Peng, H., Pappas, N., Yogatama, D., Schwartz, R., Smith, N. A., & Kong, L. (2021). *Random Feature Attention.* https://arxiv.org/abs/2103.02143

彭，H.，帕帕斯，N.，Yogatama，D.，施瓦茨，R.，史密斯，NA，和孔，L.（2021）。随机特征注意力。https://arxiv.org/abs/2103.02143

↑

92. Petty, J., van Steenkiste, S., Dasgupta, I., Sha, F., Garrette, D., & Linzen, T. (2024). *The Impact of Depth on Compositional Generalization in Transformer Language Models.* https://arxiv.org/abs/2310.19956↑ back: 1, 2

佩蒂，J.，范斯廷基斯特，S.，达斯古普塔，I.，沙，F.，加雷特，D.，和林岑，T.（2024）。深度对 Transformer 语言模型中组合泛化的影响。https://arxiv.org/abs/2310.19956 背：1、2

93. Polo, F. M., Weber, L., Choshen, L., Sun, Y., Xu, G., & Yurochkin, M. (2024). *tinyBenchmarks: evaluating LLMs with fewer examples.* https://arxiv.org/abs/2402.14992

Polo，FM、Weber，L.、Choshen，L.、Sun，Y.、Xu，G. 和 Yurochkin，M.（2024）。tinyBenchmarks：用更少的例子评估 LLM。https://arxiv.org/abs/2402.14992

↑

94. Press, O., Smith, N. A., & Lewis, M. (2022). *Train Short, Test Long: Attention with Linear Biases Enables Input Length Extrapolation.* https://arxiv.org/abs/2108.12409

Press，O.、Smith，NA 和 Lewis，M.（2022）。训练短，测试长：线性偏差的注意力可以实现输入长度外推。https://arxiv.org/abs/2108.12409

↑

95. Pyatkin, V., Malik, S., Graf, V., Ivison, H., Huang, S., Dasigi, P., Lambert, N., & Hajishirzi, H. (2025). *Generalizing Verifiable Instruction Following.* https://arxiv.org/abs/2507.02833

Pyatkin，V.、Malik，S.、Graf，V.、Ivison，H.、Huang，S.、Dasigi，P.、Lambert，N. 和 Hajishirzi，H.（2025）。推广可验证指令遵循。https://arxiv.org/abs/2507.02833

↑

96. Qin, Z., Han, X., Sun, W., Li, D., Kong, L., Barnes, N., & Zhong, Y. (2022). *The Devil in Linear Transformer.* https://arxiv.org/abs/2210.10340

Qin，Z.，Han，X.，Sun，W.，Li，D.，Kong，L.，Barnes，N.，& Zhong，Y.（2022）。线性变压器中的魔鬼。https://arxiv.org/abs/2210.10340

↑

97. Qin, Z., Yang, S., Sun, W., Shen, X., Li, D., Sun, W., & Zhong, Y. (2024). *HGRN2: Gated Linear RNNs with State Expansion.* https://arxiv.org/abs/2404.07904

Qin，Z.，Yang，S.，Sun，W.，Shen，X.，Li，D.，Sun，W. 和 Zhong，Y.（2024）。HGRN2：具有状态扩展的门控线性 RNN。https://arxiv.org/abs/2404.07904

↑

98. Qiu, Z., Huang, Z., Zheng, B., Wen, K., Wang, Z., Men, R., Titov, I., Liu, D., Zhou, J., & Lin, J. (2025). *Demons in the Detail: On Implementing Load Balancing Loss for Training Specialized Mixture-of-Expert Models.* https://arxiv.org/abs/2501.11873

邱 Z.、黄 Z.、郑 B.、温 K.、王 Z.、门 R.、蒂托夫 I.、刘 D.、周 J. 和林 J.（2025）。细节中的恶魔：关于为训练专门的混合专家模型实现负载平衡损失。https://arxiv.org/abs/2501.11873

↑

99. Qwen Team. (2025). *Qwen3-Next: Towards Ultimate Training & Inference Efficiency.* Alibaba Cloud. https://qwen.ai/blog?id=4074cca80393150c248e508aa62983f9cb7d27cd&from=research.latest-advancements-list

Qwen 团队。Qwen3-Next：迈向终极训练和推理效率。阿里云。https://qwen.ai/blog?id=4074cca80393150c248e508aa62983f9cb7d27cd&from=research.latest-advancements-list

↑

100. Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., Sutskever, I., & others. (2019). Language models are unsupervised multitask learners. In *OpenAI blog* (Vol. 1, p. 9).

Radford，A.、Wu，J.、Child，R.、Luan，D.、Amodei，D.、Sutskever，I. 等。（2019）. 语言模型是无监督的多任务学习器。在 OpenAI 博客（第 1 卷，第 9 页）中。

↑

101. Rafailov, R., Sharma, A., Mitchell, E., Ermon, S., Manning, C. D., & Finn, C. (2024). *Direct Preference Optimization: Your Language Model is Secretly a Reward Model.* https://arxiv.org/abs/2305.18290

拉法洛夫，R.、夏尔马，A.、米切尔，E.、埃尔蒙，S.、曼宁，CD 和芬恩，C.（2024）。直接偏好优化：你的语言模型暗地里是一个奖励模型。https://arxiv.org/abs/2305.18290

↑

102. Rein, D., Hou, B. L., Stickland, A. C., Petty, J., Pang, R. Y., Dirani, J., Michael, J., & Bowman, S. R. (2024). Gpqa: A graduate-level google-proof q&a benchmark. *First Conference on Language Modeling.*

Rein，D.、Hou，BL、Stickland，AC、Petty，J.、Pang，RY、Dirani，J.、Michael，J. 和 Bowman，SR（2024）。Gpqa：研究生水平的谷歌证明问答基准。第一届语言建模会议。

↑

103. Rozière, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X. E., Adi, Y., Liu, J., Sauvestre, R., Remez, T., Rapin, J., Kozhevnikov, A., Evtimov, I., Bitton, J., Bhatt, M., Ferrer, C. C., Grattafiori, A., Xiong, W., Défossez, A., ... Synnaeve, G. (2024). *Code Llama: Open Foundation Models for Code.* https://arxiv.org/abs/2308.12950

Rozière，B.，Gehring，J.，Gloeckle，F.，Sootla，S.，Gat，I.，Tan，X. E.，Adi，Y.，Liu，J.，Sauvestre，R.，Remez，T.，Rapin，J.，Kozhevnikov，A.，Evtimov，I.，Bitton，J.，Bhatt，M.，Ferrer，C. C.，Grattafiori，A.，Xiong，W.，Défossez，A.，...辛纳夫，G.（2024）。Code Llama：代码的开放基础模型。https://arxiv.org/abs/2308.12950

↑

104. Sennrich, R., Haddow, B., & Birch, A. (2016). *Neural Machine Translation of Rare Words with Subword Units.* https://arxiv.org/abs/1508.07909

Sennrich，R.、Haddow，B. 和 Birch，A.（2016）。使用子词单元对稀有词进行神经机器翻译。https://arxiv.org/abs/1508.07909

↑

105. Shao, Z., Wang, P., Zhu, Q., Xu, R., Song, J., Bi, X., Zhang, H., Zhang, M., Li, Y. K., Wu, Y., & Guo, D. (2024). *DeepSeekMath: Pushing the Limits of Mathematical Reasoning in Open Language Models.* https://arxiv.org/abs/2402.03300

邵 Z.、王 P.、朱 Q.、徐 R.、宋 J.、毕 X.、张 H.、张 M.、李 YK、吴 Y. 和郭 D.（2024）。DeepSeekMath：突破开放语言模型中数学推理的极限。https://arxiv.org/abs/2402.03300

↑

106. Shazeer, N. (2019). *Fast Transformer Decoding: One Write-Head is All You Need.* https://arxiv.org/abs/1911.02150

沙泽尔，N.（2019）。快速变压器解码：只需一个写入头即可。https://arxiv.org/abs/1911.02150↑

107. Shi, F., Suzgun, M., Freitag, M., Wang, X., Srivats, S., Vosoughi, S., Chung, H. W., Tay, Y., Ruder, S., Zhou, D., Das, D., & Wei, J. (2022). *Language Models are Multilingual Chain-of-Thought Reasoners.* https://arxiv.org/abs/2210.03057

Shi，F.、Suzgun，M.、Freitag，M.、Wang，X.、Srivats，S.、Vosoughi，S.、Chung，HW、Tay，Y.、Ruder，S.、周 D.、Das，D. 和 Wei，J.（2022）。语言模型是多语言思维链推理器。https://arxiv.org/abs/2210.03057

↑

108. Shukor, M., Aubakirova, D., Capuano, F., Kooijmans, P., Palma, S., Zouitine, A., Aractingi, M., Pascal, C., Russi, M., Marafioti, A., Alibert, S., Cord, M., Wolf, T., & Cadene, R. (2025). *SmolVLA: A Vision-Language-Action Model for Affordable and Efficient Robotics.* https://arxiv.org/abs/2506.01844

舒科尔，M.，奥巴基罗娃，D.，卡普阿诺，F.，库伊曼斯，P.，帕尔马，S.，祖伊廷，A.，阿拉廷吉，M.，帕斯卡尔，C.，鲁西，M.，马拉菲奥蒂，A.，阿利伯特，S.，科德，M.，沃尔夫，T.，和卡德内，R.（2025）。SmolVLA：一种视觉-语言-动作模型，用于经济实惠且高效的机器人技术。https://arxiv.org/abs/2506.01844

↑

109. Singh, S., Romanou, A., Fourrier, C., Adelani, D. I., Ngui, J. G., Vila-Suero, D., Limkonchotiwat, P., Marchisio, K., Leong, W. Q., Susanto, Y., Ng, R., Longpre, S., Ko, W.-Y., Ruder, S., Smith, M., Bosselut, A., Oh, A., Martins, A. F. T., Choshen, L., … Hooker, S. (2025). *Global MMLU: Understanding and Addressing Cultural and Linguistic Biases in Multilingual Evaluation.* https://arxiv.org/abs/2412.03304

Singh，S.，Romanou，A.，Fourrier，C.，Adelani，DI，Ngui，JG，Vila-Suero，D.，Limkonchotiwat，P.，Marchisio，K.，Leong，WQ，Susanto，Y.，Ng，R.，Longpre，S.，Ko，W.-Y.，Ruder，S.，Smith，M.，Bosselut，A.，Oh，A.，Martins，AFT，Choshen，L.，…胡克，S.（2025）。全球 MMLU：理解和解决多语言评估中的文化和语言偏见。https://arxiv.org/abs/2412.03304

↑

110. Sirdeshmukh, V., Deshpande, K., Mols, J., Jin, L., Cardona, E.-Y., Lee, D., Kritz, J., Primack, W., Yue, S., & Xing, C. (2025). *MultiChallenge: A Realistic Multi-Turn Conversation Evaluation Benchmark Challenging to Frontier LLMs.* https://arxiv.org/abs/2501.17399

Sirdeshmukh，V.、Deshpande，K.、Mols，J.、Jin，L.、Cardona，E.-Y.、Lee，D.、Kritz，J.、Primack，W.、Yue，S. 和 Xing，C.（2025）。MultiChallenge：对前沿 LLM 具有挑战性的现实多轮对话评估基准 https://arxiv.org/abs/2501.17399

↑

111. Smith, L. N., & Topin, N. (2018). *Super-Convergence: Very Fast Training of Neural Networks Using Large Learning Rates.* https://arxiv.org/abs/1708.07120

史密斯，LN 和托平，N.（2018）。超级收敛：使用大学习率对神经网络进行非常快速的训练。https://arxiv.org/abs/1708.07120

↑

112. Su, J., Lu, Y., Pan, S., Murtadha, A., Wen, B., & Liu, Y. (2023). *RoFormer: Enhanced Transformer with Rotary Position Embedding.* https://arxiv.org/abs/2104.09864

苏 J.、卢 Y.、潘 S.、穆尔塔达 A.、温 B. 和刘 Y.（2023）。RoFormer：具有旋转位置嵌入的增强型变压器。https://arxiv.org/abs/2104.09864

↑

113. Sun, Y., Dong, L., Zhu, Y., Huang, S., Wang, W., Ma, S., Zhang, Q., Wang, J., & Wei, F. (2024). *You Only Cache Once: Decoder-Decoder Architectures for Language Models.* https://arxiv.org/abs/2405.05254

Sun，Y.、Dong，L.、Zhu，Y.、Huang，S.、Wang，W.、马 S.、Zhang，Q.、Wang，J. 和 Wei，F.（2024）。您只能缓存一次：语言模型的解码器-解码器架构。https://arxiv.org/abs/2405.05254

↑

114. Takase, S., Kiyono, S., Kobayashi, S., & Suzuki, J. (2025). *Spike No More: Stabilizing the Pre-training of Large Language Models.* https://arxiv.org/abs/2312.16903

高濑 S.、清野 S.、小林 S. 和铃木 J.（2025）。不再尖峰：稳定大语言模型的预训练。https://arxiv.org/abs/2312.16903

↑

115. Team, 5, Zeng, A., Lv, X., Zheng, Q., Hou, Z., Chen, B., Xie, C., Wang, C., Yin, D., Zeng, H., Zhang, J., Wang, K., Zhong, L., Liu, M., Lu, R., Cao, S., Zhang, X., Huang, X., Wei, Y., … Tang, J. (2025). *GLM-4.5: Agentic, Reasoning, and Coding (ARC) Foundation Models.* https://arxiv.org/abs/2508.06471

团队，5，Zeng，A.，Lv，X.，Zheng，Q.，Hou，Z.，Chen，B.，Xie，C.，Wang，C.，Yin，D.，Zeng，H.，Zhang，J.，Wang，K.，Zhong，L.，Liu，M.，Lu，R.，Cao，S.，Zhang，X.，Huang，X.，Wei，Y.，…唐，J.（2025）。GLM-4.5：代理、推理和编码（ARC）基础模型。https://arxiv.org/abs/2508.06471

↑

116. team, F. C., Copet, J., Carbonneaux, Q., Cohen, G., Gehring, J., Kahn, J., Kossen, J., Kreuk, F., McMilin, E., Meyer, M., Wei, Y., Zhang, D., Zheng, K., Armengol-Estapé, J., Bashiri, P., Beck, M., Chambon, P., Charnalia, A., Cummins, C., … Synnaeve, G. (2025). *CWM: An Open-Weights LLM for Research on Code Generation with World Models.* https://arxiv.org/abs/2510.02387

团队，FC，Copet，J.，Carbonneaux，Q.，Cohen，G.，Gehring，J.，Kahn，J.，Kossen，J.，Kreuk，F.，McMilin，E.，Meyer，M.，Wei，Y.，Zhang，D.，Zheng，K.，Armengol-Estapé，J.，Bashiri，P.，Beck，M.，Chambon，P.，Charnalia，A.，Cummins，C.，…辛纳夫，G.（2025）。CWM：一种开放权重法学硕士，用于研究使用世界模型生成代码。https://arxiv.org/abs/2510.02387

… Hussenot, L. (2025). *Gemma 3 Technical Report.* https://arxiv.org/abs/2503.19786

Team，G.，Kamath，A.，Ferret，J.，Pathak，S.，Vieillard，N.，Merhej，R.，Perrin，S.，Matejovicova，T.，Ramé，A.，Rivière，M.，Rouillard，L.，Mesnard，T.，Cideron，G.，bastien Jean-Grill，Ramos，S.，Yvinec，E.，Casbon，M.，Pot，E.，Penchev，I.，…侯塞诺，L.（2025）。Gemma 3 技术报告。https://arxiv.org/abs/2503.19786

↑

118. Team, K., Bai, Y., Bao, Y., Chen, G., Chen, J., Chen, N., Chen, R., Chen, Y., Chen, Y., Chen, Y., Chen, Z., Cui, J., Ding, H., Dong, M., Du, A., Du, C., Du, D., Du, Y., Fan, Y., … Zu, X. (2025). *Kimi K2: Open Agentic Intelligence.* https://arxiv.org/abs/2507.20534↑ back: 1, 2, 3

Team，K.，Bai，Y.，Bao，Y.，Chen，G.，Chen，J.，Chen，N.，Chen，R.，Chen，Y.，Chen，Y.，Chen，Z.，Cui，J.，Ding，H.，Dong，M.，Du，A.，Du，C.，Du，D.，Du，Y.，Fan，Y.，…祖，X.（2025）。Kimi K2：开放智能智能。https://arxiv.org/abs/2507.20534 背：1、2、3

119. Team, L., Han, B., Tang, C., Liang, C., Zhang, D., Yuan, F., Zhu, F., Gao, J., Hu, J., Li, L., Li, M., Zhang, M., Jiang, P., Jiao, P., Zhao, Q., Yang, Q., Shen, W., Yang, X., Zhang, Y., … Zhou, J. (2025). *Every Attention Matters: An Efficient Hybrid Architecture for Long-Context Reasoning.* https://arxiv.org/abs/2510.19338

团队，L.，韩，B.，唐，C.，梁，C.，张 D.，袁 F.，朱 F.，高 J.，胡 J.，李 L.，李 M.，张 M.，江 P.，焦 P.，赵 Q.，杨 Q.，沈 W.，杨 X.，张 Y.，…周，J.（2025）。每一个注意力都很重要：用于长上下文推理的高效混合架构。https://arxiv.org/abs/2510.19338

↑

120. Team, L., Zeng, B., Huang, C., Zhang, C., Tian, C., Chen, C., Jin, D., Yu, F., Zhu, F., Yuan, F., Wang, F., Wang, G., Zhai, G., Zhang, H., Li, H., Zhou, J., Liu, J., Fang, J., Ou, J., … He, Z. (2025). *Every FLOP Counts: Scaling a 300B Mixture-of-Experts LING LLM without Premium GPUs.* https://arxiv.org/abs/2503.05139

团队，L.，曾，B.，黄，C.，张 C.，田 C.，陈 C.，金 D.，于 F.，朱 F.，袁 F.，王 F.，王 G.，翟 G.，张 H.，李 H.，周 J.，刘 J.，方 J.，欧 J.，…他，Z.（2025）。每一次失败都很重要：在没有高级 GPU 的情况下扩展 300B 专家混合 LING LLM。https://arxiv.org/abs/2503.05139

↑

121. Team, M., Xiao, C., Li, Y., Han, X., Bai, Y., Cai, J., Chen, H., Chen, W., Cong, X., Cui, G., Ding, N., Fan, S., Fang, Y., Fu, Z., Guan, W., Guan, Y., Guo, J., Han, Y., He, B., … Sun, M. (2025). *MiniCPM4: Ultra-Efficient LLMs on End Devices.* https://arxiv.org/abs/2506.07900

Team，M.，Xiao，C.，Li，Y.，Han，X.，Bai，Y.，Cai，J.，Chen，H.，Chen，W.，Cong，X.，Cui，G.，Ding，N.，Fan，S.，Fang，Y.，Fu，Z.，Guan，W.，Guan，Y.，Guo，J.，Han，Y.，He，B.，…孙，M.（2025）。MiniCPM4：终端设备上的超高效 LLM。https://arxiv.org/abs/2506.07900

↑

122. Tian, C., Chen, K., Liu, J., Liu, Z., Zhang, Z., & Zhou, J. (2025). *Towards Greater Leverage: Scaling Laws for Efficient Mixture-of-Experts Language Models*. https://arxiv.org/abs/2507.17702↑ back: 1, 2, 3

田 C.、陈 K.、刘 J.、刘 Z.、张 Z. 和周 J.（2025）。迈向更大的杠杆作用：高效混合专家语言模型的扩展定律。https://arxiv.org/abs/2507.17702 背：1、2、3

123. Toshniwal, S., Moshkov, I., Narenthiran, S., Gitman, D., Jia, F., & Gitman, I. (2024). *OpenMathInstruct-1: A 1.8 Million Math Instruction Tuning Dataset*. https://arxiv.org/abs/2402.10176

Toshniwal、S.、Moshkov、I.、Narenthiran、S.、Gitman、D.、Jia、F. 和 Gitman、I.（2024）。OpenMathInstruct-1：一个 180 万个数学指令调整数据集。https://arxiv.org/abs/2402.10176
Toshniwal、S.、Moshkov、I.、Narenthiran、S.、Gitman、D.、Jia、F. 和 Gitman、I.（2024）。OpenMathInstruct-1：一个 180 万个数学指令调整数据集。https://arxiv.org/abs/2402.10176

↑

124. Tunstall, L., Beeching, E., Lambert, N., Rajani, N., Rasul, K., Belkada, Y., Huang, S., von Werra, L., Fourrier, C., Habib, N., Sarrazin, N., Sanseviero, O., Rush, A. M., & Wolf, T. (2023). *Zephyr: Direct Distillation of LM Alignment*. https://arxiv.org/abs/2310.16944

Tunstall、L.、Beeching、E.、Lambert、N.、Rajani、N.、Rasul、K.、Belkada、Y.、Huang、S.、von Werra、L.、Fourrier、C.、Habib、N.、Sarrazin、N.、Sanseviero、O.、Rush、AM 和 Wolf、T.（2023）。Zephyr：LM 对准的直接蒸馏。https://arxiv.org/abs/2310.16944

↑

125. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., & Polosukhin, I. (2023). *Attention Is All You Need*. https://arxiv.org/abs/1706.03762↑ back: 1, 2, 3

Vaswani、A.、Shazeer、N.、Parmar、N.、Uszkoreit、J.、Jones、L.、Gomez、AN.、Kaiser、L. 和 Polosukhin、I.（2023）。注意力就是你所需要的。https://arxiv.org/abs/1706.03762 背：1、2、3

126. Waleffe, R., Byeon, W., Riach, D., Norick, B., Korthikanti, V., Dao, T., Gu, A., Hatamizadeh, A., Singh, S., Narayanan, D., Kulshreshtha, G., Singh, V., Casper, J., Kautz, J., Shoeybi, M., & Catanzaro, B. (2024). *An Empirical Study of Mamba-based Language Models*. https://arxiv.org/abs/2406.07887

Waleffe、R.、Byeon、W.、Riach、D.、Norick、B.、Korthikanti、V.、Dao、T.、Gu、A.、Hatamizadeh、A.、Singh、S.、Narayanan、D.、Kulshreshtha、G.、Singh、V.、Casper、J.、Kautz、J.、Shoeybi、M. 和 Catanzaro、B.（2024）。基于 Mamba 的语言模型的实证研究。https://arxiv.org/abs/2406.07887

↑

127. Wang, B., & Komatsuzaki, A. (2021). *GPT-J-6B: A 6 Billion Parameter Autoregressive Language Model*. https://github.com/kingoflolz/mesh-transformer-jax

王、B.、和小松崎、A.（2021）。GPT-J-6B：一个 60 亿参数的自回归语言模型。https://github.com/kingoflolz/mesh-transformer-jax

↑

128. Wei, J., Karina, N., Chung, H. W., Jiao, Y. J., Papay, S., Glaese, A., Schulman, J., & Fedus, W. (2024). Measuring short-form factuality in large language models. *arXiv Preprint arXiv:2411.04368*.

Wei、J.、Karina、N.、Chung、HW.、Jiao、YJ.、Papay、S.、Glaese、A.、Schulman、J. 和 Fedus、W.（2024）。测量大型语言模型中的简短事实性。arXiv 预印本 arXiv：2411.04368。

↑

129. Wen, K., Hall, D., Ma, T., & Liang, P. (2025). *Fantastic Pretraining Optimizers and Where to Find Them*. https://arxiv.org/abs/2509.02046↑ back: 1, 2

温、K.、霍尔、D.、马、T.、和梁、P.（2025）。出色的预训练优化器以及在哪里可以找到它们。https://arxiv.org/abs/2509.02046 背：1、2

130. Xie, S. M., Pham, H., Dong, X., Du, N., Liu, H., Lu, Y., Liang, P., Le, Q. V., Ma, T., & Yu, A. W. (2023). *DoReMi: Optimizing Data Mixtures Speeds Up Language Model Pretraining*. https://arxiv.org/abs/2305.10429

谢 SM、范 H.、董 X.、杜 N.、刘 H.、卢 Y.、梁 P.、乐 QV、马 T. 和于，AW（2023）。DoReMi：优化数据混合可加快语言模型预训练速度。https://arxiv.org/abs/2305.10429

↑

131. Xiong, W., Liu, J., Molybog, I., Zhang, H., Bhargava, P., Hou, R., Martin, L., Rungta, R., Sankararaman, K. A., Oguz, B., Khabsa, M., Fang, H., Mehdad, Y., Narang, S., Malik, K., Fan, A., Bhosale, S., Edunov, S., Lewis, M., ... Ma, H. (2023a). *Effective Long-Context Scaling of Foundation Models*. https://arxiv.org/abs/2309.16039

Xiong、W.、Liu、J.、Molybog、I.、Zhang、H.、Bhargava、P.、Hou、R.、Martin、L.、Rungta、R.、Sankararaman、K. A.、Oguz、B.、Khabsa、M.、Fang、H.、Mehdad、Y.、Narang、S.、Malik、K.、Fan、A.、Bhosale、S.、Edunov、S.、Lewis、M.、...马、H.（2023a）。基础模型的有效长上下文扩展。https://arxiv.org/abs/2309.16039

↑

132. Xiong, W., Liu, J., Molybog, I., Zhang, H., Bhargava, P., Hou, R., Martin, L., Rungta, R., Sankararaman, K. A., Oguz, B., Khabsa, M., Fang, H., Mehdad, Y., Narang, S., Malik, K., Fan, A., Bhosale, S., Edunov, S., Lewis, M., ... Ma, H. (2023b). *Effective Long-Context Scaling of Foundation Models*. https://arxiv.org/abs/2309.16039

Xiong、W.、Liu、J.、Molybog、I.、Zhang、H.、Bhargava、P.、Hou、R.、Martin、L.、Rungta、R.、Sankararaman、K. A.、Oguz、B.、Khabsa、M.、Fang、H.、Mehdad、Y.、Narang、S.、Malik、K.、Fan、A.、Bhosale、S.、Edunov、S.、Lewis、M.、...马、H.（2023b）。基础模型的有效长上下文扩展。https://arxiv.org/abs/2309.16039

↑

133. Xu, H., Peng, B., Awadalla, H., Chen, D., Chen, Y.-C., Gao, M., Kim, Y. J., Li, Y., Ren, L., Shen, Y., Wang, S., Xu, W., Gao, J., & Chen, W. (2025). *Phi-4-Mini-Reasoning: Exploring the Limits of Small Reasoning Language Models in Math*. https://arxiv.org/abs/2504.21233

徐 H.、彭 B.、阿瓦达拉 H.、陈 D.、陈 Y.-C.、高 M.、金 YJ.、李 Y.、任任 L.、沈 Y.、王 S.、徐 W.、高 J. 和陈 W.（2025）。Phi-4-Mini-Reasoning：探索数学中小推理语言模型的局限性 https://arxiv.org/abs/2504.21233。

↑

134. Yang, A., Li, A., Yang, B., Zhang, B., Hui, B., Zheng, B., Yu, B., Gao, C., Huang, C., Lv, C., Zheng, C., Liu, D., Zhou, F., Huang, F., Hu, F., Ge, H., Wei, H., Lin, H., Tang, J., ... Qiu, Z. (2025). *Qwen3 Technical Report*. https://arxiv.org/abs/2505.09388↑ back: 1, 2, 3

杨 A.、李 A.、杨 B.、张 B.、辉 B.、郑 B.、宇 B.、高 C.、黄 C.、吕 C.、郑 C.、刘 D.、周 F.、黄 F.、胡 F.、葛 H.、魏 H.、林 H.、唐 J.、...邱、Z.（2025）。Qwen3 技术报告。https://arxiv.org/abs/2505.09388 背：1、2、3

135. Yang, A., Yu, B., Li, C., Liu, D., Huang, F., Huang, H., Jiang, J., Tu, J., Zhang, J., Zhou, J., Lin, J., Dang, K., Yang, K., Yu, L., Li, M., Sun, M., Zhu, Q., Men, R., He, T., ... Zhang, Z. (2025). *Qwen2.5-1M Technical Report*. https://arxiv.org/abs/2501.15383↑ back: 1, 2

杨 A.、Yu、B.、Li、C.、Liu、D.、Huang、F.、Huang、H.、江 J.、涂 J.、张 J.、周 J.、林 J.、党 K.、杨 K.、余 L.、李 M.、孙 M.、朱 Q.、门 R.、何 T.、...张、Z.（2025）。Qwen2.5-1M 技术报告。https://arxiv.org/abs/2501.15383 背：1、2

136. Yang, B., Venkitesh, B., Talupuru, D., Lin, H., Cairuz, D., Blunsom, P., & Locatelli, A. (2025). *Rope to Nope and Back Again: A New Hybrid Attention Strategy*. https://arxiv.org/abs/2501.18795↑ back: 1, 2

杨、B.、文基特什、B.、塔卢普鲁、D.、林、H.、凯鲁兹、D.、布伦森、P.、和洛卡特利、A.（2025）。绳索到不，然后再回来：一种新的混合注意力策略。https://arxiv.org/abs/2501.18795 背：1、2

137. Yang, G., & Hu, E. J. (2022). *Feature Learning in Infinite-Width Neural Networks*. https://arxiv.org/abs/2011.14522

Yang、G. 和 Hu、EJ.（2022）。无限宽度神经网络中的特征学习。https://arxiv.org/abs/2011.14522

138. Yen, H., Gao, T., Hou, M., Ding, K., Fleischer, D., Izsak, P., Wasserblat, M., & Chen, D. (2025). *HELMET: How to Evaluate Long-Context Language Models Effectively and Thoroughly*. https://arxiv.org/abs/2410.02694↑ back: 1, 2

Yen、H.、Gao、T.、Hou、M.、Ding、K.、Fleischer、D.、Izsak、P.、Wasserblat、M. 和 Chen、D.（2025）。HELMET：如何有效、彻底地评估长上下文语言模型。https://arxiv.org/abs/2410.02694 背：1、2

139. Yu, Q., Zhang, Z., Zhu, R., Yuan, Y., Zuo, X., Yue, Y., Dai, W., Fan, T., Liu, G., Liu, L., Liu, X., Lin, H., Lin, Z., Ma, B., Sheng, G., Tong, Y., Zhang, C., Zhang, M., Zhang, W., ... Wang, M. (2025). *DAPO: An Open-Source LLM Reinforcement Learning System at Scale*. https://arxiv.org/abs/2503.14476

Yu、Q.、Zhang、Z.、Zhu、R.、Yuan、Y.、Zuo、X.、Yue、Y.、Dai、W.、Fan、T.、Liu、G.、Liu、L.、Liu、X.、Lin、H.、Lin、Z.、马 B.、Sheng、G.、Tong、Y.、Zhang、C.、Zhang、M.、Zhang、W.、...王、M.（2025）。DAPO：大规模开源 LLM 强化学习系统。https://arxiv.org/abs/2503.14476

↑

140. Yuan, J., Gao, H., Dai, D., Luo, J., Zhao, L., Zhang, Z., Xie, Z., Wei, Y. X., Wang, L., Xiao, Z., Wang, Y., Ruan, C., Zhang, M., Liang, W., & Zeng, W. (2025). *Native Sparse Attention: Hardware-Aligned and Natively Trainable Sparse Attention*. https://arxiv.org/abs/2502.11089

Yuan、J.、Gao、H.、Dai、D.、Luo、J.、Zhao、L.、Zhang、Z.、Xie、Z.、Wei、Y. X.、Wang、L.、Xiao、Z.、Wang、Y.、Ruan、C.、Zhang、M.、Liang、W.、& Zeng、W.（2025）。本机稀疏注意力：硬件对齐且本机可训练的稀疏注意力。https://arxiv.org/abs/2502.11089

↑

141. Yue, Y., Chen, Z., Lu, R., Zhao, A., Wang, Z., Yue, Y., Song, S., & Huang, G. (2025). *Does Reinforcement Learning Really Incentivize Reasoning Capacity in LLMs Beyond the Base Model?* https://arxiv.org/abs/2504.13837

Yue、Y.、Chen、Z.、Lu、R.、Zhao、A.、Wang、Z.、Yue、Y.、Song、S. 和 Huang、G.（2025）。强化学习真的能激励法学硕士超越基本模型的推理能力吗？https://arxiv.org/abs/2504.13837

↑

142. Zhao, Y., Qu, Y., Staniszewski, K., Tworkowski, S., Liu, W., Miłoś, P., Wu, Y., & Minervini, P. (2024). Analysing The Impact of Sequence Composition on Language Model Pre-Training. *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 7897–7912. 10.18653/v1/2024.acl-long.427

赵 Y.、瞿 Y.、斯坦尼谢夫斯基 K.、特沃科夫斯基 S.、刘 W.、米沃什 P.、吴 Y. 和密涅维尼 P. （2024）。分析序列组合对语言模型预训练的影响。计算语言学协会第 62 届年会论文集（第 1 卷：长论文），7897–7912。10.18653/v1/2024.acl-long.427

↑

143. Zhou, F., Wang, Z., Ranjan, N., Cheng, Z., Tang, L., He, G., Liu, Z., & Xing, E. P. (2025). *MegaMath: Pushing the Limits of Open Math Corpora.* https://arxiv.org/abs/2504.02807

周 F.、王 Z.、冉詹 N.、程 Z.、唐 L.、何 G.、刘 Z. 和邢 EP （2025）。MegaMath：突破开放数学语料库的极限。https://arxiv.org/abs/2504.02807
143. Zhou, F., Wang, Z., Ranjan, N., Cheng, Z., Tang, L., He, G., Liu, Z., & Xing, E. P. (2025). *MegaMath: Pushing the Limits of Open Math Corpora.* https://arxiv.org/abs/2504.02807

周 F.、王 Z.、冉詹 N.、程 Z.、唐 L.、何 G.、刘 Z. 和邢 EP （2025）。MegaMath：突破开放数学语料库的极限。https://arxiv.org/abs/2504.02807
↑

144. Zhou, J., Lu, T., Mishra, S., Brahma, S., Basu, S., Luan, Y., Zhou, D., & Hou, L. (2023). *Instruction-Following Evaluation for Large Language Models.* https://arxiv.org/abs/2311.07911

周 J.、卢 T.、米什拉 S.、梵天 S.、巴苏 S.、栾 Y.、周 D. 和侯 L. （2023）。大型语言模型的指令遵循评估。https://arxiv.org/abs/2311.07911

↑

145. Zhu, T., Liu, Q., Wang, H., Chen, S., Gu, X., Pang, T., & Kan, M.-Y. (2025). *SkyLadder: Better and Faster Pretraining via Context Window Scheduling.* https://arxiv.org/abs/2503.15450↑ back: 1, 2

Zhu，T.，Liu，Q.，Wang，H.，Chen，S.，Gu，X.，Pang，T.，& Kan，M.-Y.（2025）. SkyLadder：通过上下文窗口调度更好更快地进行预训练。https://arxiv.org/abs/2503.15450 背：1、2

146. Zuo, J., Velikanov, M., Chahed, I., Belkada, Y., Rhayem, D. E., Kunsch, G., Hacid, H., Yous, H., Farhat, B., Khadraoui, I., Farooq, M., Campesan, G., Cojocaru, R., Djilali, Y., Hu, S., Chaabane, I., Khanna, P., Seddik, M. E. A., Huynh, N. D., … Frikha, S. (2025). *Falcon-H1: A Family of Hybrid-Head Language Models Redefining Efficiency and Performance.* https://arxiv.org/abs/2507.22448↑ back: 1, 2

Zuo，J.，Velikanov，M.，Chahed，I.，Belkada，Y.，Rhayem，D. E.，Kunsch，G.，Hacid，H.，Yous，H.，Farhat，B.，Khadraoui，I.，Farooq，M.，Campesan，G.，Cojocaru，R.，Djilali，Y.，胡，S.，Chaabane，I.，Khanna，P.，Seddik，MEA，Huynh，ND，...弗里卡，S.（2025）。Falcon-H1：一系列混合头语言模型，重新定义了效率和性能。https://arxiv.org/abs/2507.22448 背：1、2

Footnotes 脚注

1. The idea to compute these statistics comes from the Llama 3 tech report (Grattafiori et al., 2024).

   计算这些统计数据的想法来自 Llama 3 技术报告（Grattafiori 等人，2024 年）。 ↑

2. For vLLM see: Reasoning parsers, Tool parsers. For SGLang, see: Reasoning parsers, Tool parsers

   对于 vLLM，请参阅：推理解析器、工具解析器。对于 SGLang，请参阅：推理解析器、工具解析器 ↑

3. The Transformers team has recently added parsers for extract tool calling and reasoning outputs. If adopted by engines like vLLM, the compatibility criterion may become less important in the future.

   Transformers 团队最近添加了用于提取工具调用和推理输出的解析器。如果被 vLLM 等引擎采用，兼容性标准在未来可能会变得不那么重要。