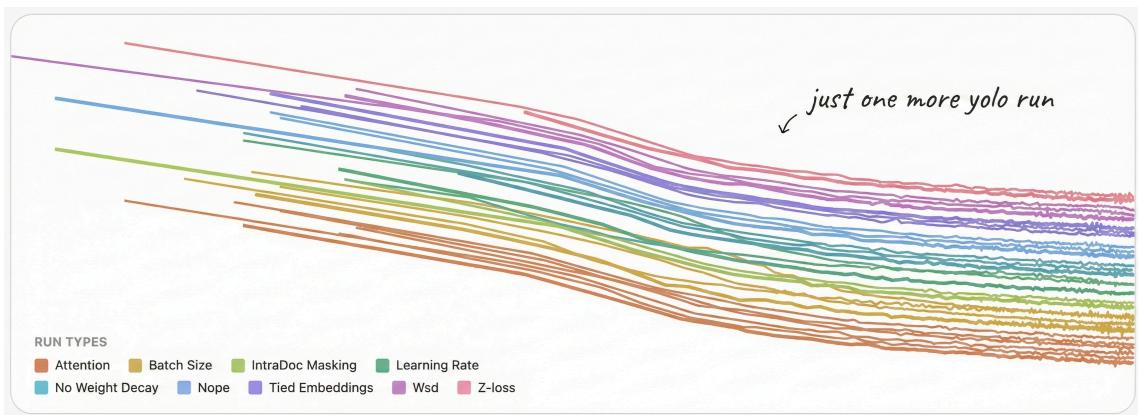


The Smol Training Playbook:

The Secrets to Building World-Class LLMs



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

AUTHORS	AFFILIATION	PUBLISHED
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	Hugging Face	Oct. 30, 2025

Translated by Dasol Kang

LinkedIn: <https://www.linkedin.com/in/dasol-kang-103a24217/>

Read Online: wikidocs.net/318788

Email: englishmt4118@gmail.com

서론 (Introduction)

오늘날 고성능 LLM을 훈련하려면 실제로 무엇이 필요할까?

공개된 연구 논문들을 보면 모든 것이 간단해 보입니다: 전략적인 아키텍처 선택, 신중하게 큐레이션된 데이터셋, 그리고 충분한 컴퓨팅 자원이 있으면 됩니다. 결과는 깔끔하게 정리되어 있고, ablation study(변인 통제 실험)는 체계적이고 명확합니다. 모든 결정이 돌이켜보면 당연해 보이죠. 하지만 이런 보고서들은 성공한 것만 보여주고, 약간의 장밋빛 회고(rosy retrospection)를 적용합니다 – 새벽 2시에 dataloader(데이터 로더)를 디버깅하던 시간들, loss spike(손실 급등 현상), 또는 훈련을 조용히 망치고 있던 미묘한 tensor parallelism(텐서 병렬화) 버그(나중에 자세히 다룹니다!)는 담겨 있지 않습니다. 현실은 훨씬 더 지저분하고, 반복적이며, 최종 논문에는 담기지 않는 수많은 결정들로 가득합니다.

11조(11T) 개의 토큰으로 훈련된 3B(30억) 파라미터 다국어 추론 모델인 [SmollM3](#)의 훈련 과정 비하인드 스토리에 함께 해 주세요. 이것은 평범한 블로그 포스트가 아닙니다. 세계 수준의 언어 모델을 구축하는 데 무엇이 필요한지에 대한 깊은 통찰로 이어진 결정, 발견, 그리고 막다른 길들로 얹힌 거미줄을 풀어나가는 여정입니다.

이 글은 우리의 모델 훈련 장편 시리즈의 마지막 작품이기도 합니다: 우리는 대규모 데이터셋 구축([FineWeb](#)), 수천 개의 GPU를 조율하여 완벽한 하모니를 만드는 과정([Ultra Scale Playbook](#)), 그리고 프로세스의 각 단계에서 최적의 평가 방법 선택([Evaluation Guidebook](#))을 다뤄왔습니다. 이제 이 모든 것을 하나로 엮어 강력한 AI 모델을 구축합니다. 최종적으로 성공한 레시피만이 아니라, 실패, 인프라 장애, 그리고 모든 결정을 형성한 디버깅 과정까지 완전한 여정을 함께 걸어가겠습니다.

이 이야기는 드라마처럼 읽힙니다. 소규모에서 유망했던 ablation들이 대규모에서는 왜 때때로 제대로 작동하지 않는지, 1T 토큰 후에 왜 훈련을 재시작했는지, 영어 성능을 유지하면서 다국어성(multilinguality), 수학(math), 코드(code)라는 경쟁하는 목표들을 어떻게 균형 있게 맞췄는지, 그리고 마지막으로 하이브리드 추론 모델(hybrid reasoning model)을 어떻게 포스트 트레이닝(post-training)했는지 보게 될 것입니다.

우리는 또한 우리가 한 모든 일을 차갑게 나열하는 대신 모험을 통한 체계적인 이야기를 선택했습니다. "훌륭한 데이터셋과 GPU가 있다"에서 "정말 강력한 모델을 구축했다"로 나아가려는 모든 이들을 위한 가이드로 생각해주세요. 이렇게 개방적으로 공유하는 것이 연구와 실제 프로덕션 사이의 격차를 좁히고, 여러분의 다음 훈련이 조금 덜 혼란스럽도록 도움이 되기를 바랍니다.

이 블로그 포스트를 읽는 방법

이 블로그 포스트를 처음부터 끝까지 읽을 필요는 없으며, 실제로 한 번에 통독하기엔 너무 긴 분량입니다. 블로그 포스트는 여러 독립적인 섹션으로 구성되어 있어 건너뛰거나 개별적으로 읽을 수 있습니다:

훈련 나침반(Training compass): 자체 모델을 사전학습해야 하는지에 대한 고수준 논의입니다. VC 자금을 모두 태워버리기 전에 스스로에게 물어봐야 할 근본적인 질문들과, 의사결정 과정을 체계적으로 고민하는 방법을 안내합니다. 이것은 고수준 섹션이므로 기술적 내용으로 바로 넘어가고 싶다면 이 부분은 빠르게 스킵하세요.

사전학습(Pretraining): 훈련 나침반 다음 섹션들은 자체 사전학습 실행을 위한 탄탄한 레시피를 구축하는데 필요한 모든 것을 다룹니다: 절제 실험(ablation) 실행 방법, 평가 선택, 데이터 소스 혼합, 아키텍처 선택, 하이퍼파라미터 튜닝, 그리고

마지막으로 훈련 마라톤 견디기. 이 섹션은 처음부터 사전학습을 계획하지 않더라도 지속적 사전학습(일명 중간 학습, mid-training)에 관심이 있는 경우에도 적용됩니다.

후속 학습(Post-training): 블로그의 이 부분에서는 사전학습된 모델을 최대한 활용하는 데 필요한 모든 기법을 배웁니다. SFT, DPO, GRPO로 시작하는 후속 학습의 전체 알파벳과 모델 병합(model merging)의 어두운 기술과 연금술을 익히세요. 이러한 알고리즘을 잘 작동시키는 방법에 대한 대부분의 지식은 고통스러운 교훈을 통해 얻어지며, 우리는 여기서 우리의 경험을 공유하여 여러분이 그런 고통을 일부나마 피할 수 있도록 도울 것입니다.

인프라(Infrastructure): 사전학습이 케이크이고 후속 학습이 아이싱과 위의 체리라면, 인프라는 산업용 오븐입니다. 인프라가 없으면 아무 일도 일어나지 않고, 고장 나면 즐거운 일요일 베이킹 시간이 화재 위험으로 변합니다. GPU 클러스터를 이해하고, 분석하고, 디버깅하는 방법에 대한 지식은 인터넷의 다양한 라이브러리, 문서, 포럼에 흩어져 있습니다. 이 섹션에서는 GPU 레이아웃, CPU/GPU/노드/스토리지 간의 통신 패턴, 그리고 병목 현상을 식별하고 극복하는 방법을 안내합니다.

그래서 어디서부터 시작할까요? 가장 흥미로운 섹션을 선택하고 시작해봅시다!

Training Compass (훈련 나침반): Why → What → How

이 섹션은 모델 훈련의 의사결정 프레임워크를 세 단계로 나눕니다.

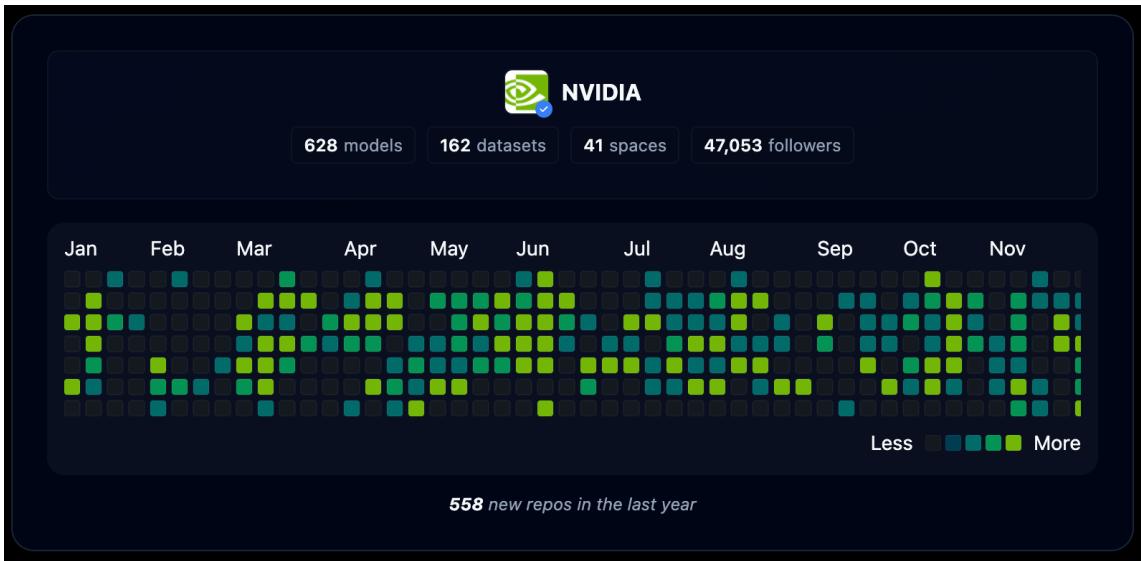
단계	핵심 질문	세부 고려사항
Why train? (왜 훈련하는가?)	훈련의 목적	Research (연구), Production (프로덕션), Strategic (전략적 목적)
What to train? (무엇을 훈련하는가?)	훈련 대상 결정	Architecture (아키텍처), Model size (모델 크기), Data mix (데이터 혼합), Assistant type (어시스턴트 유형), Running ablations (ablation 실험 수행)...
How to train? (어떻게 훈련하는가?)	훈련 방법	Setup infra (인프라 구축), Training framework (훈련 프레임워크), Handling loss spikes (loss spike 처리), Midtraining (중간 훈련), SFT vs RL (SFT 대 RL)...

용어 설명:

- Ablation:** 모델의 특정 구성요소를 제거하거나 변경하여 그 영향을 측정하는 실험 방법
- SFT (Supervised Fine-Tuning):** 지도 학습 방식의 미세조정
- RL (Reinforcement Learning):** 강화학습
- Midtraining:** 사전 훈련 중간에 데이터 구성이나 학습률 등을 변경하는 단계

머신러닝 분야는 최적화(optimisation)와 강박적인 관계를 맺고 있습니다. 우리는 손실 곡선, 모델 아키텍처, 처리량에 집착합니다. 결국 머신러닝은 근본적으로 모델의 손실 함수를 최적화하는 것이니까요. 하지만 이러한 기술적 세부 사항에 뛰어들기 전에, 종종 묻지 않는 더 근본적인 질문이 있습니다. 우리가 애초에 이 모델을 훈련시켜야 하는가?

아래 히트맵에서 볼 수 있듯이, 오픈소스 AI 생태계는 거의 매일 세계 수준의 모델을 출시하고 있습니다. Qwen, Gemma, DeepSeek, Kimi, Llama, Olmo 등, 매달 목록은 더 길어집니다. 이것들은 단순한 연구 프로토타입이나 장난감 예제가 아닙니다: 다국어 이해부터 코드 생성과 추론에 이르기까지 놀라울 정도로 광범위한 사용 사례를 다루는 프로덕션급 모델들입니다. 대부분은 관대한 라이선스와 사용을 도와줄 준비가 된 활발한 커뮤니티를 갖추고 있습니다.



(이 외에도 구글, 허깅페이스 등 다양한 기업의 히트맵을 볼 수 원문에서 볼 수 있습니다.)

이는 불편한 진실을 제기합니다: 어쩌면 당신은 자체 모델을 훈련할 필요가 없을지도 모릅니다.

이것은 "LLM 훈련 가이드"를 시작하는 이상한 방법처럼 보일 수 있습니다. 하지만 많은 실패한 훈련 프로젝트들은 잘못된 하이퍼파라미터나 버그가 있는 코드 때문에 실패한 것이 아니라, 누군가가 필요하지 않은 모델을 훈련시키기로 결정했기 때문에 실패했습니다. 따라서 훈련에 착수하고 실행 방법을 파고들기 전에 두 가지 질문에 답해야 합니다: 왜 이 모델을 훈련하는가? 그리고 어떤 모델을 훈련해야 하는가? 명확한 답이 없으면 세상이 이미 가지고 있는 것, 또는 더 나쁘게는 아무도 필요로 하지 않는 것을 구축하는 데 몇 달간의 컴퓨팅 자원과 엔지니어링 시간을 낭비하게 될 것입니다.

왜부터 시작합시다. 목적을 이해하지 못하면 이후에 따르는 어떤 것에 대해서도 일관된 결정을 내릴 수 없기 때문입니다.

☞ 이 섹션에 대하여 이 섹션은 블로그의 나머지 부분과 다릅니다. 실험과 기술적 세부사항에 대한 것이 아니라, **전략적 계획(strategic planning)**에 대한 것입니다. 처음부터 훈련해야 하는지, 어떤 모델을 만들어야 하는지 결정하는 과정을 안내해 드리겠습니다. 이미 "why"와 "what"에 대해 깊이 생각해 보셨다면, ["Every big model starts with a small ablation\(모든 큰 모델은 작은 ablation에서 시작한다\)"](#) 챕터로 바로 넘어가셔도 됩니다. 하지만 확신이 없다면, 여기에 시간을 투자하는 것이 나중에 많은 노력을 절약해 줄 것입니다.

Why: 아무도 대답하고 싶지 않은 질문

실제로 무슨 일이 일어나는지 솔직하게 말해봅시다. 누군가(운이 좋다면) GPU 클러스터에 접근할 수 있게 되는데, 연구 보조금을 통해서든 회사의 여유 용량을 통해서든, 사고 과정은 대략 이렇습니다: "3개월간 H100 100대를 사용할 수 있어. 모델을 훈련시키자!" 모델 크기는 임의로 선택되고, 데이터셋은 사용 가능한 것들로 조합됩니다. 훈련이 시작됩니다. 그리고 6개월 후, 컴퓨팅 예산과 팀 사기를 모두 소진한 후, 결과 모델은 아무도 왜라고 묻지 않았기 때문에 사용되지 않은 채 방치됩니다.

다음은 모델을 훈련해서는 안 되는 이유들입니다:

우리는 컴퓨팅 자원을 사용할 수 있습니다

↳ 그것은 자원이지, 목표가 아닙니다

다른 모든 사람들도 하고 있습니다

↳ 그것은 동료 압박이지, 전략이 아닙니다

AI는 미래입니다

↳ 그것은 상투적인 문구이지, 계획이 아닙니다

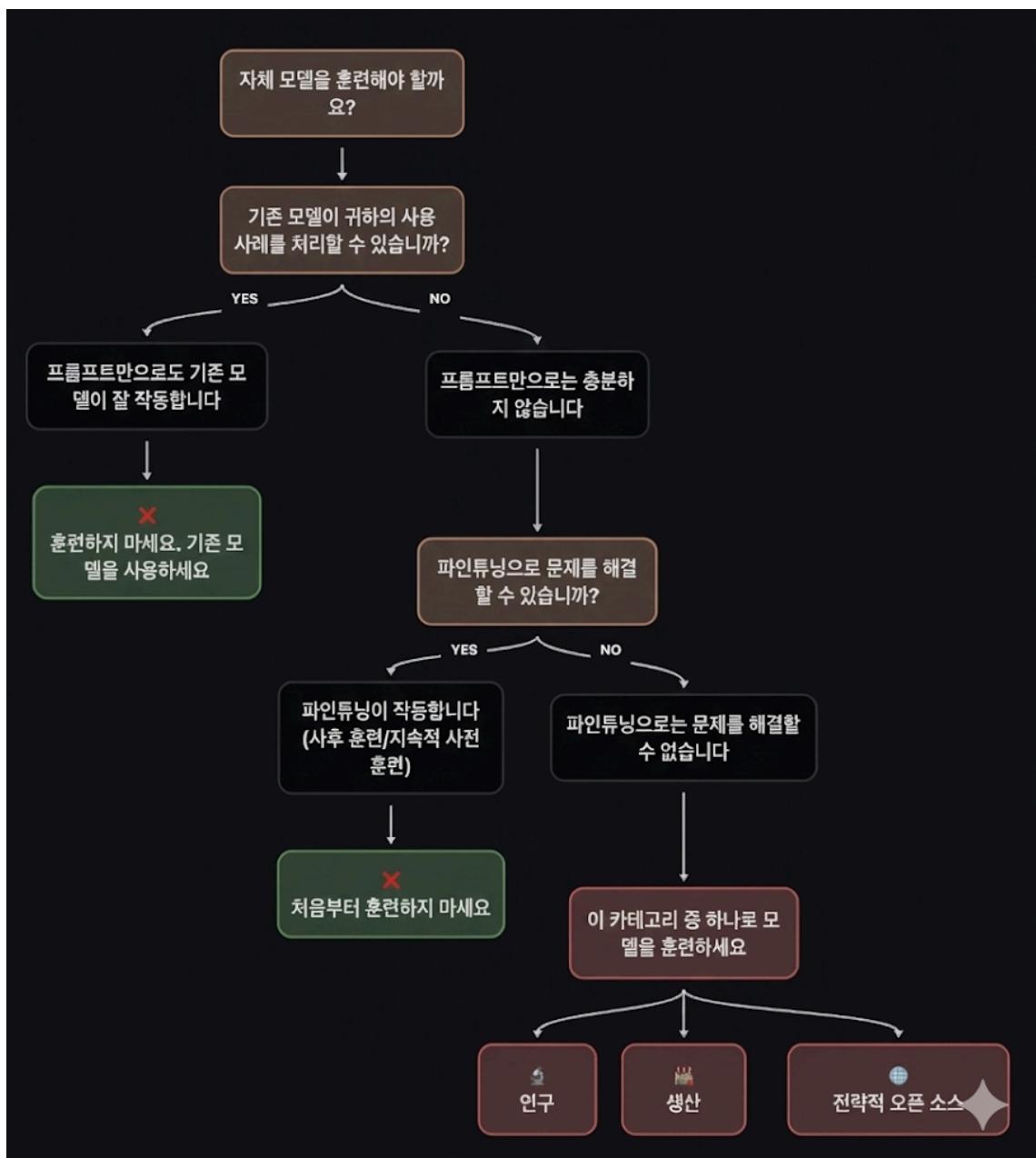
우리는 가능한 최고의 모델을 원합니다

↳ 그것은 결정을 안내하기에 충분히 구체적이지 않습니다



"우리가 자체 모델을 훈련시켰다"는 매력은 강력하지만, 많은 시간과 자원을 투자하기 전에 다음을 묻는 것이 합리적입니다:
왜 이 모델을 훈련해야 하는가?

아래 플로우차트는 대규모 사전학습 프로젝트를 시작하기 전에 거쳐야 할 사고 과정을 안내합니다. 기술적 관점에서, 기본적으로 먼저 프롬프팅하거나 파인튜닝하여 작업을 수행할 수 있는 기존 모델이 없는지 확인해야 합니다.



맞춤형 사전학습이 의미를 가질 수 있는 세 가지 일반적인 영역이 있습니다: 새로운 연구를 하고 싶거나, 프로덕션 사용 사례에 대한 매우 구체적인 요구사항이 있거나, 오픈 모델 생태계의 공백을 메우고 싶은 경우입니다. 각각을 간단히 살펴보겠습니다.

Research: 무엇을 이해하고 싶은가?

LLM 분야에서 할 수 있는 연구는 많습니다. LLM 연구 프로젝트의 공통점은 일반적으로 명확하게 정의된 질문으로 시작한다는 것입니다:

- 이 새로운 옵티マイ저로 10B+ 모델까지 훈련을 확장할 수 있는가? ([Muon is Scalable for LLM Training](#))

- SFT 없이 강화 학습만으로 추론 능력을 만들어낼 수 있는가? ([DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning](#))
- 순수한 합성 교과서 데이터로 좋은 소형 모델을 훈련할 수 있는가? ([Textbooks Are All You Need](#))
- 오픈 라이선스 데이터만으로 훈련하여 경쟁력 있는 성능을 달성할 수 있는가? ([The Common Pile v0.1: An 8TB Dataset of Public Domain and Openly Licensed Text](#))

가설을 최대한 구체적으로 만들고 필요한 실험 규모에 대해 생각하는 것이 성공 가능성을 높입니다.

Production: 왜 기존 모델을 사용할 수 없는가?

기업이 자신의 사용 사례에 기성 모델을 사용할 수 없는 주요 이유는 세 가지입니다. 그중 두 가지는 기술적이고 나머지 하나는 거버넌스 때문입니다.

자체 모델을 훈련하는 첫 번째 이유는 도메인 특수성입니다: 데이터나 작업이 기존 모델이 잘 처리하지 못하는 고도로 전문화된 어휘나 구조를 포함하는 경우입니다. 예를 들어,

- 고유한 어휘와 장거리 의존성을 가진 DNA 모델
- 도메인별 전문 용어와 논리에 대한 깊은 이해가 필요한 법률 또는 금융 모델

두 번째 관련된 이유는 배포 제약입니다. 하드웨어, 지역 시간 또는 개인정보 보호 요구사항에 맞춤화된 모델이 필요한 경우입니다. 예를 들어, 드론에서 실행되거나 FPGA와 같은 맞춤형 하드웨어를 가진 온프레미스 시스템에서 실행되는 LLM입니다.

다음은 간단한 테스트입니다. 며칠 동안 Qwen3, Gemma3 또는 다른 현재 SOTA 모델 위에 구축해보세요. 프롬프팅, 도구 사용 또는 후속 학습을 통해 성능 목표에 도달할 수 있나요? 그렇지 않다면 자체 모델을 훈련할 때가 된 것입니다.

요구사항을 충족하는 데 필요한 후속 학습 예산이 막대하더라도, 처음부터 시작하는 것보다 여전히 저렴할 수 있습니다. 모델을 1T 토큰에 대해 파인튜닝하는 것은 처음부터 시작하여 10T+ 토큰에 대해 훈련하는 것보다 여전히 더 경제적입니다.

이 시점에서 LLM 훈련자들은 기적적으로 이것을 후속 학습 대신 중간 학습이라고 부르기 시작합니다.

자체 사내 언어 모델을 구축하는 세 번째 이유는 안전성과 거버넌스입니다. 규제 산업이나 고위험 애플리케이션에 있기 때문에 훈련 데이터, 모델 동작 및 업데이트 주기에 대한 완전한 통제가 필요합니다. 모델에 무엇이 들어갔는지 정확히 알고 규제 당국에 이를 증명할 수 있어야 합니다. 경우에 따라서는 자체 모델을 구축하는 것 외에 다른 선택지가 없을 수도 있습니다.

이것들이 기업이 사내 모델을 훈련하는 주요 이유들이지만, 오픈 모델을 공개하는 기업이나 조직은 어떨까요?

Strategic Open-Source: 메울 수 있는 공백이 보이는가?

경험 있는 AI 랩이 새로운 오픈 모델을 공개하는 가장 일반적인 이유 중 하나는 오픈소스 생태계에서 특정 공백이나 새로운 AI 사용 사례를 식별했기 때문입니다.

패턴은 일반적으로 다음과 같습니다: 충분히 탐색되지 않은 영역을 발견합니다. 아마도 매우 긴 컨텍스트를 가진 강력한 온디바이스 모델이 없거나, 다국어 모델이 존재하지만 저자원 언어에서 약하거나, 분야가 [Genie3](#)와 같은 상호작용 세계 모델로 이동하고 있지만 좋은 오픈 웨이트 모델이 존재하지 않습니다.

더 잘할 수 있다고 믿을 만한 이유가 있습니다: 더 나은 훈련 데이터를 큐레이션했거나, 더 나은 훈련 레시피를 개발했거나, 다른 사람들이 할 수 없었던 곳에서 오버트레이닝할 수 있는 컴퓨팅 자원이 있습니다. 목표는 구체적입니다: "역대 최고의 모델"이 아니라 "온디바이스 사용을 위한 최고의 3B 모델" 또는 "100만 컨텍스트를 가진 최초의 소형 모델"입니다.

이것은 실제 목표이며 성공은 가치를 창출합니다. 개발자들이 당신의 모델을 채택하고, 그것이 다른 사람들을 위한 인프라가 되거나, 기술적 신뢰성을 확립합니다. 하지만 성공에는 경험이 필요합니다. 실제로 실현 가능한 것이 무엇인지, 경쟁적인 공간에서 어떻게 안정적으로 실행할 수 있는지 알아야 합니다. 이를 구체화하기 위해 Hugging Face에서 이 질문에 대해 어떻게 생각하는지 살펴보겠습니다.

Hugging Face의 여정

그렇다면 Hugging Face는 왜 오픈 모델을 훈련하는 걸까요? 답은 간단합니다. 우리는 오픈소스 생태계에 유용하고 다른 사람들이 거의 채우지 않는 공백을 메우는 것들을 만듭니다.

수백만 개의 오픈 웨이트 모델이 있지만, 완전히 오픈된 모델을 훈련하는 조직은 매우 적습니다. Hugging Face 외에도 Ai2와 Stanford의 Marin 커뮤니티가 있습니다.

여기에는 데이터셋, 도구 및 훈련 모델이 포함됩니다. 우리가 시작한 모든 LLM 훈련 프로젝트는 공백을 발견하고 의미 있는 기여를 할 수 있다고 믿는 것으로 시작되었습니다.

우리는 GPT-3 ([Brown et al., 2020](#))가 출시된 후 첫 번째 LLM 프로젝트를 시작했습니다. 당시에는 다른 누구도 오픈 대안을 만들지 않는 것처럼 느껴졌고, 우리는 그 지식이 소수의 산업 랩에만 갇히게 될 것을 우려했습니다. 그래서 우리는 GPT-3의 오픈 버전을 훈련하기 위해 [BigScience](#) 워크숍을 시작했습니다. 그 결과 모델은 Bloom이었으며, 수십 명의 기여자들이 1년 동안 175B 파라미터 모델을 사전학습하기 위한 훈련 스택, 토크나이저, 사전학습 코퍼스를 구축한 결과물이었습니다.

Bloom의 후속작은 2022년의 StarCoder였습니다 ([Li et al., 2023](#)). OpenAI는 GitHub Copilot을 위한 Codex를 개발했지만 ([Chen et al., 2021](#)) 클로즈드 소스였습니다. 오픈소스 대안을 구축하는 것이 생태계에 명백한 가치를 제공할 것입니다. 그래서 [BigCode](#) 우산 아래 ServiceNow와 협력하여 [The Stack](#) 데이터셋을 구축했고, Codex를 재현하기 위해 [StarCoder 15B](#)를 훈련시켰습니다. [StarCoder2](#) ([Lozhkov et al., 2024](#))는 더 오래 훈련할 수 있었다는 것을 배우고, 더 오래 훈련된 작은 모델들이 하나의 큰 모델보다 더 가치 있을 수 있다는 것을 인식한 결과였습니다. 우리는 당시 오픈 코드 모델에 대해 누구도 한 적 없는 것보다 훨씬 많은 수조 토큰에 대해 패밀리(3B/7B/15B)를 훈련시켰습니다.

[SmolLM Family](#)도 유사한 패턴을 따랐습니다. 우리는 강력한 소형 모델이 거의 없다는 것을 발견했고, 방금 강력한 사전학습 데이터셋인 [FineWeb-Edu](#) ([Penedo et al., 2024](#))를 구축했습니다. [SmolLM](#) (135M/360M/1.7B)이 우리의 첫 번째 버전이었습니다. [SmolLM2](#) ([Allal et al., 2025](#))는 더 나은 데이터와 더 긴 훈련에 초점을 맞춰 여러 분야에서 SOTA 성능에 도달했습니다. [SmolLM3](#)는 3B로 확장하면서 2025년 커뮤니티가 중요하게 여기는 기능인 하이브리드 추론, 다국어 및 긴 컨텍스트를 추가했습니다.

이 패턴은 사전학습을 넘어 확장됩니다. 우리는 DPO가 대규모에서 작동함을 보여주기 위해 [Zephyr](#) ([Tunstall et al., 2023](#))를 훈련시켰고, DeepSeek R1의 종류 파이프라인을 재현하기 위해 [Open-R1](#)을 시작했으며, 국제 정보 올림피아드에서 SOTA 성능을 가진 경쟁 프로그래밍용 [OlympicCoder](#)를 출시했습니다. 또한 비전을 위한 [SmolVLM](#) ([Marafioti et al., 2025](#))과 로보틱스를 위한 [SmolVLA](#) ([Shukor et al., 2025](#))로 다른 모달리티도 탐색했습니다.

이 섹션이 모델을 훈련하려는 이유에 대해 깊이 생각하는 것의 가치를 확신시켰기를 바랍니다.

이 블로그 포스트의 나머지 부분에서는 이러한 자기 성찰을 마치고 훈련할 정당한 이유가 있다고 가정하겠습니다.

What: 목표를 결정으로 전환하기

이제 왜 훈련하는지 알았으니, 무엇을 훈련해야 할까요? "무엇"이란 다음을 의미합니다. 모델 타입(밀집형, MoE, 하이브리드, 새로운 것), 모델 크기, 아키텍처 세부 사항 및 데이터 훈합. 왜를 결정하면 무엇을 도출할 수 있습니다. 예를 들어.

- 온디바이스를 위한 빠른 모델 → 작고 효율적인 모델
- 다국어 모델 → 큰 토크나이저 어휘
- 초장 컨텍스트 → 하이브리드 아키텍처

사용 사례에 의해 결정되는 결정 외에도, 더 안정적이거나, 샘플 효율적이거나, 더 빠르게 만들어 훈련 자체를 최적화하는 몇 가지 선택이 있습니다. 이러한 결정이 항상 명확한 것은 아니지만, 의사결정 과정을 대략 두 단계로 나눌 수 있습니다.

계획(Planning) : 실험을 실행하기 전에 사용 사례를 결정해야 할 구성 요소에 매핑합니다. 배포 환경은 모델 크기 제약을 결정합니다. 일정은 어떤 아키텍처 위험을 감수할 수 있는지 결정합니다. 목표 능력은 데이터셋 요구사항을 결정합니다. 이 단계는 "왜"의 각 제약 조건을 "무엇"의 구체적인 사양과 연결하는 것입니다.

검증(Validation) : 시작점과 잠재적 수정 사항 목록이 있으면 체계적으로 테스트합니다. 테스트는 비용이 많이 들기 때문에 사용 사례의 성능을 의미 있게 개선하거나 훈련을 최적화할 수 있는 변경 사항에 집중합니다. 여기서 절제 실험이 등장하며, 이는 [ablations section](#) 섹션에서 다룹니다.

👉 테스트를 실행하는 방법만이 아니라 무엇을 테스트할 가치가 있는지 식별하는 법을 배우세요. 무관한 선택에 대한 완벽한 절제 실험은 중요한 것에 대한 부실한 절제 실험만큼 많은 컴퓨팅 자원을 낭비합니다.

다음 챕터에서는 모델을 정의하기 위한 모든 옵션과 체계적인 실험으로 선택의 폭을 좁히는 방법을 배울 것입니다. 그 전에 우리 자신의 모델을 훈련하고 훌륭한 LLM을 구축하는 놀라운 다른 팀들을 관찰하면서 얻은 팀 및 프로젝트 설정에 대한 몇 가지 학습 내용을 공유하고자 합니다.

Super power: 속도와 데이터

물론 로마에 가는 방법은 많지만, 성공적인 LLM 훈련 팀을 지속적으로 구별하는 것은 반복 속도입니다. LLM 훈련은 실제로 훈련을 통한 학습 분야이며, 더 자주 훈련할수록 팀은 더 나아질 것입니다. 따라서 1년에 모델 하나를 훈련하는 팀과 분기당 하나를 훈련하는 팀 사이에서 후자가 훨씬 더 빠르게 향상될 것입니다. 예를 들어 Qwen과 DeepSeek 팀을 볼 수 있습니다. 이제 잘 알려진 이름이 된 이들은 빠른 주기로 새로운 모델을 지속적으로 출시한 오랜 실적을 가지고 있습니다.

반복 속도 외에도 LLM 훈련에서 단연 가장 영향력 있는 측면은 데이터 큐레이션입니다. 모델을 개선하기 위해 아키텍처 선택에 뛰어드는 자연스러운 경향이 있지만, LLM 훈련에서 탁월한 팀은 무엇보다도 고품질 데이터에 집착하는 팀입니다.

반복 속도와 연관된 또 다른 측면은 팀 규모입니다. 주요 사전학습 작업의 경우 실행할 충분한 컴퓨팅 자원을 갖춘 소수의 사람만 필요합니다. 오늘날 Llama 3와 같은 모델을 사전학습하려면 아마도 2-3명만 필요할 것입니다. 더 다양한 훈련과 다운 스트림 작업(멀티모달, 다국어, 후속 학습 등)에 착수하기 시작해야만 각 도메인에서 탁월하기 위해 몇 명을 더 추가해야 할 것입니다.

따라서 잘 갖춰진 소규모 팀으로 시작하여 2-3개월마다 새로운 모델을 구축하면 짧은 시간 내에 정상에 오를 것입니다. 이제 블로그의 나머지 부분은 이 팀의 기술적 일상에 초점을 맞출 것입니다.

모든 큰 모델은 작은 ablation으로 시작한다

LLM 훈련을 시작하기 전에 모델의 성능과 훈련 효율성을 형성할 많은 결정을 내려야 합니다. 어떤 아키텍처가 우리의 사용 사례에 가장 적합할까요? 어떤 옵티마이저와 학습률 스케줄을 사용하고 어떤 데이터 소스를 혼합할까요?

이러한 결정이 어떻게 내려지는지는 자주 묻는 질문입니다. 사람들은 때때로 이것들이 깊이 생각함으로써 결정된다고 기대합니다. 그리고 전략적 사고가 필수적이긴 하지만—[이전 섹션](#)에서 어떤 아키텍처 변경이 테스트할 가치가 있는지 식별하는 것을 다루었듯이—추론만으로는 충분하지 않습니다. LLM에서는 항상 직관적인 것이 아니며, 작동해야 한다는 가설이 실제로는 결실을 맺지 못하는 경우가 있습니다.

예를 들어, "최고 품질의 데이터"처럼 보이는 것을 사용한다고 해서 항상 더 강력한 모델이 나오는 것은 아닙니다. 인류의 과학적 지식이 방대하게 모인 [arXiv](#)를 예로 들어보겠습니다. 직관적으로 이러한 풍부한 STEM 데이터에 대한 훈련은 우수한 모델을 생성해야 하지 않을까요? 실제로는 그렇지 않으며, 특히 작은 모델의 경우 성능을 해칠 수도 있습니다 ([Shao et al., 2024](#)). 왜일까요? 그 이유는 arXiv 논문들이 지식으로 가득하지만, 매우 전문화되어 있고 모델이 가장 잘 학습하는 다양하고 일반적인 텍스트와는 상당히 다른 좁은 학술적 스타일로 작성되어 있기 때문입니다.

그렇다면 문제를 오래 열심히 들여다보는 것이 도움이 되지 않는다면 무엇이 작동하는지 어떻게 알 수 있을까요? 좋은 경험 주의자처럼 많은 실험을 실행합니다. 머신러닝은 순수 수학이 아니라 실제로 매우 실험적인 과학입니다.

여러 면에서 머신러닝은 통계역학이 발견되기 전의 열역학과 유사합니다. 우리는 더 깊은 이론적 설명이 여전히 나오고 있음에도 불구하고 놀랍도록 잘 작동하는 신뢰할 수 있는 경험적 법칙과 설계 원칙을 가지고 있습니다.

이러한 실험들이 많은 중요한 결정을 안내할 것이므로, 잘 설정하는 것이 정말 중요합니다. 기본적으로 우리가 원하는 두 가지 주요 속성이 있습니다.

속도. 가능한 한 빨리 실행되어 자주 반복할 수 있어야 합니다. 더 많은 절제 실험을 실행할수록 더 많은 가설을 테스트할 수 있습니다.

신뢰성. 강력한 판별력을 제공해야 합니다. 초기에 다른 설정을 의미 있게 구별할 수 없는 메트릭을 본다면, 우리의 절제 실험은 거의 드러내지 못할 것입니다(그리고 노이즈가 많다면 노이즈를 쫓는 위험이 있습니다!). 자세한 내용은 [FineTaks blog post](#)를 확인하세요.

하지만 절제 실험을 설정하기 전에 아키텍처 유형과 모델 크기에 대한 몇 가지 기본적인 선택을 해야 합니다. 나침반에 의해 안내되는 이러한 결정은 어떤 훈련 프레임워크를 사용할지, 컴퓨팅 예산을 어떻게 할당할지, 어떤 베이스라인에서 시작할지에 영향을 미칩니다.

SmoLM3의 경우, 우리는 작은 온디바이스 모델을 목표로 했기 때문에 3B 파라미터의 밀집형 Llama 스타일 아키텍처를 선택했습니다. 하지만 "모델 아키텍처 설계" 챕터에서 볼 수 있듯이, MoE 또는 하이브리드 모델이 사용 사례에 더 적합할 수 있으며, 다른 모델 크기에는 다른 트레이드오프가 있습니다. 나중에 이러한 선택을 심층적으로 탐구하고 이러한 결정을 내리는 방법을 보여드리겠습니다. 지금은 가장 실용적인 첫 번째 단계인 베이스라인 선택부터 시작하겠습니다.

베이스라인 선택

모든 성공적인 모델은 검증된 기반 위에 구축되고 필요에 맞게 수정됩니다. Qwen이 첫 번째 모델 패밀리를 훈련했을 때 ([Bai et al., 2023](#)), Llama의 아키텍처에서 시작했습니다. Meta가 Llama 3를 훈련했을 때, Llama 2에서 시작했습니다. Kimi K2는 DeepSeek-V3의 MoE 아키텍처에서 시작했습니다. 이것은 아키텍처뿐만 아니라 훈련 하이퍼파라미터와 옵티마이저에도 적용됩니다.

왜일까요? 좋은 아키텍처와 훈련 설정 설계는 많은 조직에 걸쳐 수년간의 반복이 필요합니다. 표준 트랜스포머와 Adam과 같은 옵티마이저는 수천 번의 실험을 통해 개선되었습니다. 사람들은 실패 모드를 발견하고, 불안정성을 디버깅하고, 구현을 최적화했습니다. 검증된 기반에서 시작한다는 것은 그 축적된 지식을 모두 물려받는다는 의미입니다. 새로 시작한다는 것은 모든 문제를 스스로 재발견한다는 의미입니다.

아키텍처의 좋은 시작점을 만드는 것은 다음과 같습니다.

- 제약 조건과 일치함 : 배포 대상 및 사용 사례와 일치합니다.
- 대규모에서 검증됨 : 유사하거나 더 큰 크기에서 수조 토큰 실행.
- 잘 문서화됨 : 오픈 모델에서 작동하는 것으로 입증된 알려진 하이퍼파라미터.
- 프레임워크 지원 : 고려 중인 훈련 프레임워크와 사용할 계획인 추론 프레임워크에서 지원되어야 합니다.

아래는 다양한 아키텍처 및 모델 크기에 대한 강력한 2025년 베이스라인 옵션의 비포괄적 목록입니다.

아키텍처 타입	모델 패밀리	크기
Dense	Llama 3.1	8B, 70B
Dense	Llama 3.2	1B, 3B
Dense	Qwen3	0.6B, 1.7B, 4B, 14B, 32B
Dense	Gemma3	12B, 27B
Dense	SmolLM2, SmolLM3	135M, 360M, 1.7B, 3B
MoE	Qwen3 MoE	30B-A3B, 235B-A122B
MoE	GPT-OSS	21B-A3B, 117B-A5B
MoE	Kimi Moonlight	16B-A3B
MoE	Kimi-k2	1T-A32B
MoE	DeepSeek V3	671B-A37B
Hybrid	Zamba2	1.2B, 2.7B, 7B
Hybrid	Falcon-H1	0.5B, 1.5B, 3B, 7B, 34B
MoE + Hybrid	Qwen3-Next	80B-A3B
MoE + Hybrid	MiniMax-01	456B-A46B

따라서 아키텍처 유형으로 이동하여 모델에 원하는 파라미터 수에 가까운 베이스라인을 선택하세요. 시작하는 아키텍처가 확정된 것이 아니므로 너무 깊이 생각하지 마세요. 다음 섹션에서는 베이스라인에서 최적의 최종 아키텍

처로 이동하는 방법을 살펴보겠습니다.

MODIFYING YOUR BASELINE : 위험 제거의 원칙

이제 작동하고 사용 사례에 맞는 베이스라인이 있습니다. 여기서 멈추고, 데이터 훈련에 대해 훈련하고(좋다고 가정할 때) 괜찮은 모델을 얻을 수 있습니다. 많은 성공적인 프로젝트가 정확히 그렇게 합니다. 하지만 베이스라인은 특정 제약 조건에 최적화되어 있지 않으며, 그것을 만든 사람의 사용 사례와 배포 대상을 위해 설계되었습니다. 따라서 목표에 더 잘 맞추기 위해 수정할 가치가 있는 것들이 있을 것입니다. 그러나 모든 아키텍처 변경에는 위험이 따릅니다. 성능을 향상시킬 수도, 망칠 수도, 아무것도 하지 않으면서 절제 실험 컴퓨팅 자원을 낭비할 수도 있습니다.

궤도를 유지하는 원칙은 **위험 제거(derisking)**입니다. 도움이 된다는 것을 테스트하지 않고는 아무것도 변경하지 마세요.

💡 무엇이 위험 제거로 간주되는가?

변경 사항이 목표 능력에서 성능을 향상시키거나, 수용 가능한 트레이드오프 범위를 넘어서 성능을 해치지 않으면서 의미 있는 이점(예. 더 빠른 추론, 더 낮은 메모리, 더 나은 안정성)을 제공한다는 것을 테스트가 보여줄 때 위험이 제거됩니다.

까다로운 부분은 베이스라인과 훈련 설정에 수정할 수 있는 많은 구성 요소가 있다는 것입니다. 어텐션 메커니즘, 위치 인코딩, 활성화 함수, 옵티마이저, 훈련 하이퍼파라미터, 정규화 방식, 모델 레이아웃 등. 각각은 잠재적인 실험을 나타내며, 이러한 구성 요소는 종종 비선형적으로 상호 작용합니다. 모든 것을 테스트하거나 모든 상호 작용을 탐색할 시간이나 컴퓨팅 자원이 없습니다.

현재 베이스라인에 대해 유망한 변경 사항을 테스트하는 것부터 시작하세요. 무언가가 작동하면 통합하여 새로운 베이스라인을 만든 다음, 그것에 대해 다음 변경 사항을 테스트하세요. 컴퓨팅 예산이 허용한다면 변경 사항을 개별적으로 테스트하고 leave-one-out 분석을 실행할 수 있습니다.

모든 하이퍼파라미터에 대한 철저한 그리드 검색이나 나오는 모든 아키텍처 변형을 테스트하는 함정에 빠지지 마세요.

🎯 전략적 실험

실험을 실행하는 방법을 아는 것만으로는 어떤 실험이 실행할 가치가 있는지 모른다면 충분하지 않습니다. 수정 사항을 테스트하기 전에 다음 두 가지 질문을 스스로에게 하세요.

이것이 나의 특정 사용 사례에 도움이 될까? 이것이 나의 훈련을 최적화할까?

수정 사항이 두 질문 중 어느 것도 명확하게 다루지 않는다면 건너뛰세요.

이제 전략적 계획을 통해 유망한 것을 식별하는 방법을 알았으니, 경험적 검증으로 넘어갈 시간입니다. 다음 섹션에서는 이러한 변경 사항을 실제로 테스트하는 방법을 보여드리겠습니다. 신뢰할 수 있는 실험을 설정하고, 결과를 해석하고, 일반적인 함정을 피하는 방법을 다루겠습니다. 그런 다음 다음 챕터에서는 인기 있는 아키텍처, 데이터, 인프라 및 훈련 결정을 테스트하는 구체적인 예를 살펴보겠습니다.

그러니 실험에 사용할 수 있는 간단한 절제 실험 설정을 구축해봅시다. 먼저 어떤 훈련 프레임워크를 선택할지 결정해야 합니다.

훈련 프레임워크 선택

내려야 할 첫 번째 결정은 모델 훈련에 어떤 프레임워크를 사용할지, 그리고 확장하여 모든 절제 실험을 실행하는 데 어떤 프레임워크를 사용할지입니다. 이 선택은 세 가지 주요 고려 사항의 균형을 포함합니다.

1. 프레임워크는 대상 아키텍처를 지원하거나 쉽게 확장할 수 있어야 합니다.
2. 안정적이고 프로덕션 준비가 되어 있어야 하며, 훈련 중간에 신비롭게 중단되는 경향이 없어야 합니다.
3. 빠르게 반복하고 컴퓨팅 예산을 최대한 활용할 수 있도록 강력한 처리량을 제공해야 합니다.

실제로 이러한 요구 사항은 서로 충돌하여 트레이드오프를 만들 수 있습니다. 사용 가능한 옵션을 살펴봅시다.

프레임워크	기능	전투 테스트	최적화	코드 라인 수 (핵심 / 전체)	확장성 및 디버깅
Megatron-LM	✓ 광범위	✓ Kimi-K2, Nemotron	✓ 3D 병렬화의 선구자	93k / 269k	⚠ 초보자에게 어려움
DeepSpeed	✓ 광범위	✓ BLOOM, GLM	✓ ZeRO & 3D 병렬화의 선구자	94k / 194k	⚠ 초보자에게 어려움
TorchTitan	⚡ 증가하는 기능 세트	⚠ 더 새롭지만 PyTorch 팀이 테스트	⚡ 밀집형 모델에 최적화, MoE 개선 진행 중	7k / 9k	⚡ 보통 : 병렬화 노하우 필요
Nanotron	⌚ 최소한, HF 사전학습에 맞춤화	✓ 예 (StarCoder, SmolLM)	✓ 최적화됨 (UltraScale Playbook)	15k / 66k	⚡ 보통 : 병렬화 노하우 필요

위 표는 인기 있는 프레임워크 간의 주요 트레이드오프를 요약합니다. 처음 세 프레임워크의 코드 라인 수는 TorchTitan 기술 보고서 ([Liang et al., 2025](#))에서 가져왔습니다. 각각에 대해 더 자세히 논의해봅시다.

Nvidia의 [Megatron-LM](#)은 수년간 존재해왔으며 전투 테스트를 거쳤습니다. Kimi의 K2 ([Team et al., 2025](#))와 같은 모델을 구동하는 것이며, 견고한 처리량을 제공하고 원하는 대부분의 프로덕션 기능을 가지고 있습니다. 하지만 그 성숙도에는 복잡성이 따릅니다. 처음 접할 때 코드베이스를 탐색하고 수정하기 어려울 수 있습니다.

[DeepSpeed](#)는 유사한 범주에 속합니다. ZeRO 최적화의 선구자이며 BLOOM 및 GLM과 같은 모델을 구동했습니다.

Megatron-LM처럼 광범위하게 전투 테스트를 거쳤고 최적화되었지만 동일한 복잡성 문제를 공유합니다. 큰 코드베이스 (총 194k 라인)는 시작할 때, 특히 사용자 정의 기능을 구현하거나 예상치 못한 동작을 디버깅할 때 위험적일 수 있습니다.

반면에 PyTorch의 최근 [TorchTitan](#) 라이브러리는 컴팩트하고 모듈식 코드베이스 덕분에 훨씬 가볍고 탐색하기 쉽습니다. 사전학습에 필요한 핵심 기능을 가지고 있으며 빠른 실험에 적합합니다. 그러나 더 새로운 만큼 전투 테스트를 거치지 않았고 활발하게 개발되고 있어 여전히 약간 불안정할 수 있습니다.

우리는 다른 길을 택하여 nanotron이라는 자체 프레임워크를 처음부터 구축했습니다. 이것은 우리에게 완전한 유연성과 대규모 사전학습에 대한 깊은 이해를 제공했습니다. 나중에 [Ultra Scale Playbook](#)으로 발전한 통찰력입니다. 라이브러리를 오픈소스화한 이후 커뮤니티로부터 귀중한 피드백을 받았지만, 대부분의 경우 먼저 기능을 직접 전투 테스트해야 했습니다. 프레임워크는 이제 훈련에 필요한 모든 프로덕션 기능을 지원하지만, MoE 지원과 같은 영역은 여전히 구축 중입니다.

처음부터 구축하는 것이 당시에는 합리적이었지만, 문제를 디버깅하고 누락된 기능을 추가하는 데 팀 전문 지식과 시간에 대한 주요 투자가 필요합니다. 강력한 대안은 기존 프레임워크를 포크하여 필요에 맞게 향상시키는 것입니다. 예를 들어, Thinking Machines Lab은 TorchTitan의 포크로 내부 사전학습 라이브러리를 구축했습니다 ([Source](#)).

궁극적으로 선택은 팀의 전문 지식, 목표 기능, 그리고 개발에 투자할 의향이 있는 시간 대 가장 프로덕션 준비가 된 옵션을 사용하는 것에 달려 있습니다.

여러 프레임워크가 필요를 지원한다면 특정 하드웨어에서 처리량을 비교하세요. 빠른 실험과 속도 실행의 경우 더 간단한 코드베이스가 종종 승리합니다.

Ablation 설정

프레임워크를 선택했으니 이제 Ablation 설정을 설계해야 합니다. 빠르게 반복할 수 있을 만큼 빠르지만, 결과가 신호를 제공하고 최종 모델로 전이될 만큼 충분히 큰 실험이 필요합니다. 이를 설정하는 방법을 살펴봅시다.

Ablation 프레임워크 설정

절제 실험의 목표는 소규모로 실험을 실행하고 최종 프로덕션 실행에 자신 있게 외삽할 수 있는 결과를 얻는 것입니다.

두 가지 주요 접근 방식이 있습니다. 첫째, 대상 모델 크기를 가져와서 더 적은 토큰으로 훈련할 수 있습니다. SmoLM3 절제 실험의 경우 최종 11T 대신 100B 토큰으로 전체 3B 모델을 훈련했습니다. 둘째, 대상 모델이 너무 크면 절제 실험을 위해 더 작은 프록시 모델을 훈련할 수 있습니다. 예를 들어, Kimi가 32B 활성 파라미터를 가진 1T 파라미터 Kimi K2 모델을 개발할 때, 모든 절제 실험에 전체 크기를 사용하는 것은 비용이 너무 많이 들었기 때문에 0.5B 활성 파라미터를 가진 3B MoE에서 일부 절제 실험을 실행했습니다 ([Team et al., 2025](#)).

한 가지 핵심 질문은 이러한 소규모 발견이 실제로 전이되는지 여부입니다. 우리의 경험상 소규모에서 성능을 해치는 것이 있다면 대규모에 대해서는 자신 있게 배제할 수 있습니다. 하지만 소규모에서 작동하는 것이 있다면, 이러한 발견이 더 큰 규모로 외삽될 것이라는 결론을 높은 확률로 내리기 위해 합리적인 수의 토큰에 대해 훈련했는지 확인해야 합니다. 더 오래 훈련하고 절제 실험 모델이 최종 모델에 가까울수록 더 좋습니다.

이 블로그 포스트에서는 모든 절제 실험에 베이스라인 바닐라 트랜스포머를 사용하겠습니다. 주요 설정은 45B 토큰으로 훈련된 [Llama3.2](#) 1B 아키텍처를 따르는 1B 트랜스포머입니다. 이 nanotron 설정([config](#))을 사용하여 8xH100 노드에서 훈련하는 데 약 1.5일이 걸립니다 (GPU당 초당 42k 토큰). SmoLM3 훈련 중에 우리는 100B 토큰으로 훈련된 3B 모델에서 이러한 절제 실험을 실행했습니다 ([config here](#)). 각 챕터의 끝에서 해당 결과를 공유하겠습니다 (결론이 일치하는 것을 볼 수 있습니다).

베이스라인 1B 설정은 구조화된 YAML 형식으로 모든 필수 훈련 세부 사항을 포착합니다. 주요 섹션은 다음과 같습니다.

```
## 데이터셋 및 혼합 가중치
data_stages:
- data:

    dataset:
        dataset_folder:
        - fineweb-edu
        - stack-edu-python
        - finemath-3plus

    dataset_weights:
    - 0.7
    - 0.2
    - 0.1

## 모델 아키텍처, Llama3.2 1B 구성
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

## 훈련 하이퍼파라미터, 코사인 스케줄을 사용한 AdamW
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

## 병렬화, 1 노드
parallelism:
  dp: 8 # 8개 GPU에 걸친 데이터 병렬화
  tp: 1 # 1B 규모에서는 텐서 또는 파이프라인 병렬화 불필요
  pp: 1

## 토크나이저
tokenizer:
  tokenizer_max_length: 4096
  tokenizer_name_or_path: HuggingFaceTB/SmoLLM3-3B

## 배치 크기, 시퀀스 길이 및 30B 토큰에 대한 총 훈련
tokens:
  batch_accumulation_per_replica: 16
  micro_batch_size: 3 # GBS (글로벌 배치 크기)=dp * batch_acc* MBS * sequence=1.5M 토큰
  sequence_length: 4096
  train_steps: 20000 # GBS * 20000 = 30B

... (생략)

```

ablations 실험의 경우 테스트하는 내용에 따라 다른 섹션을 수정하고 나머지는 일정하게 유지합니다. [architecture choices](#) 을 위한 모델 섹션, [optimizer and training hyperparameters](#), 섹션, [data curation](#)을 위한 data_stages 섹션입니다.

👉 한 번에 한 가지만 수정하세요

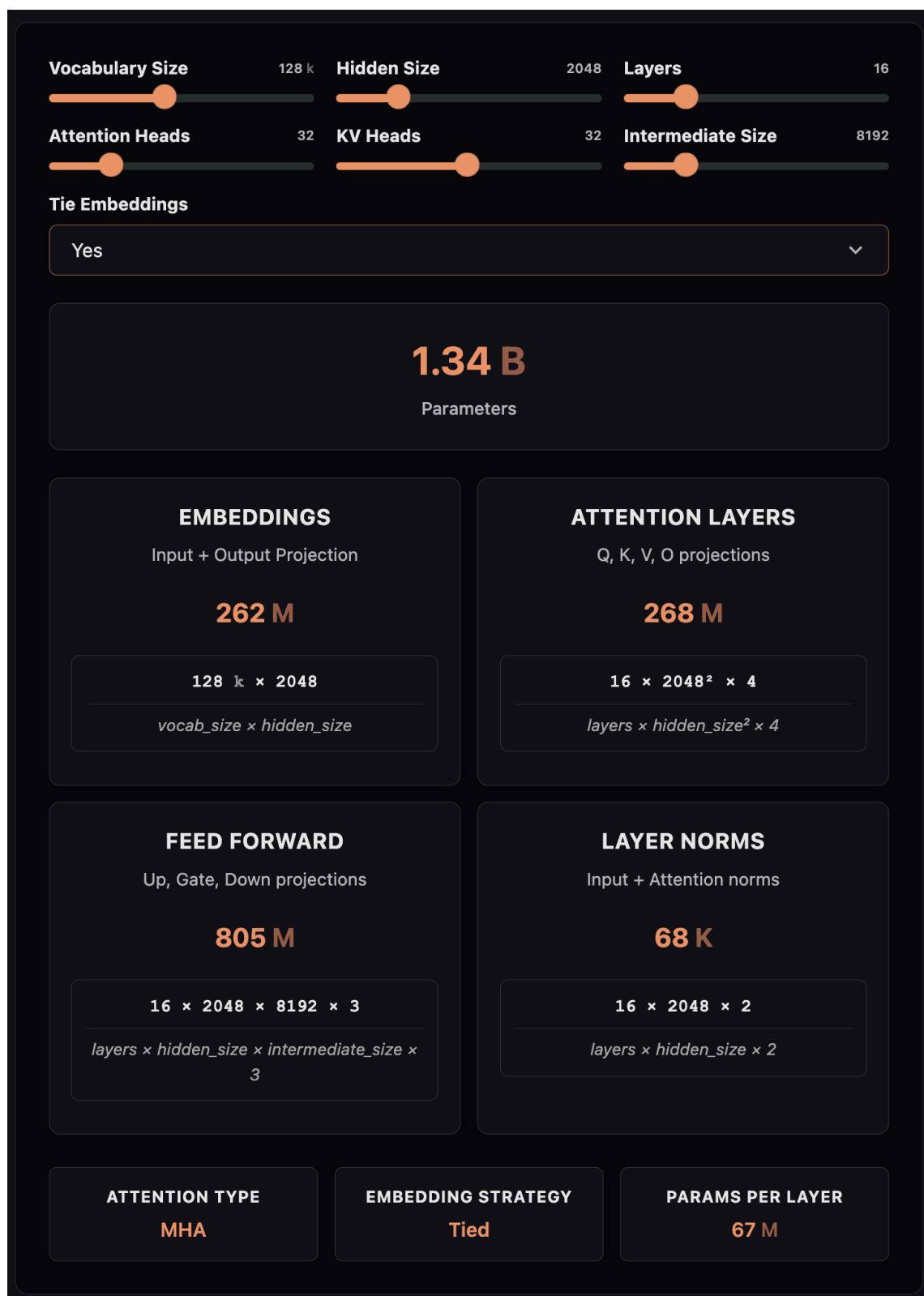
다른 모든 것을 일정하게 유지하면서 절제 실험당 하나의 변수만 변경하세요. 여러 가지를 변경하고 성능이 향상되면 무엇이 원인인지 알 수 없습니다. 수정 사항을 개별적으로 테스트한 다음 성공한 것들을 결합하고 재평가하세요.

절제 실험을 실행할 때 일부 아키텍처 변경은 파라미터 수를 크게 변경할 수 있습니다. 예를 들어, 타이드 임베딩에서 언타이드 임베딩으로 전환하면 임베딩 파라미터가 두 배가 되는 반면, MHA에서 GQA 또는 MQA로 전환하면 어텐션 파라미터가 상당히 감소합니다. 공정한 비교를 보장하기 위해 파라미터 수를 추적하고 때때로 다른 하이퍼파라미터(예. 읔닉 크기 또는 레이어 수)를 조정하여 모델 크기를 대략 동일하게 유지해야 합니다. 다음은 다양한 구성에 대한 파라미터 수를 추정하는데 사용하는 간단한 함수입니다.

```
from transformers import LlamaConfig, LlamaForCausalLM

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"
```

또한 밀집형 트랜스포머의 경우 LLM 파라미터 분포를 시각화하는 대화형 도구를 제공합니다. 이것은 아키텍처 결정을 내리거나 절제 실험을 위한 설정을 구성할 때 유용할 수 있습니다.



Understanding what works: 평가

절제 실험(ablation)을 시작한 후, 무엇이 효과가 있고 없는지 어떻게 알 수 있을까요?

모델을 학습시키는 사람이라면 가장 먼저 손실(loss)을 확인하려 할 것이며, 실제로 이는 중요합니다. 손실이 급격한 스파이크나 불안정성 없이 부드럽게 감소하는 것을 확인해야 합니다. 많은 아키텍처 선택에서 손실은 다운스트림 성능과 잘 상관관계를 보이며 이것만으로 충분할 수 있습니다([Y. Chen et al., 2025](#)). 그러나 손실만 보는 것이 항상 신뢰할 수 있는 것은 아닙니다. 데이터 절제 실험의 예를 들면, 위키피디아로 학습하면 웹 페이지로 학습하는 것보다 낮은 손실을 얻을 수 있지만(다음 토큰을 예측하기가 더 쉽기 때문에), 그렇다고 해서 더 유능한 모델을 얻는다는 의미는 아닙니다. 마찬가지로, 실행 간에 토크나이저를 변경하면 텍스트가 다르게 분할되므로 손실을 직접 비교할 수 없습니다. 일부 변경 사항은 추론이나 수학과 같은 특정 능력에만 영향을 미쳐 평균 손실에서는 회석될 수 있습니다. 마지막으로 중요한 점은, 사전학습 손실이 수렴한 후에도 모델이 다운스트림 태스크에서 계속 개선될 수 있다는 것입니다([Liu et al., 2022](#)).

이러한 미묘한 효과를 전체적으로 파악하고 이해하려면 더 세밀한 평가가 필요하며, 자연스러운 접근 방식은 지식, 이해력, 추론 능력 및 우리에게 중요한 다른 영역을 테스트하는 다운스트림 평가를 사용하는 것입니다.

이러한 절제 실험에서는 조기에 좋은 신호를 제공하는 태스크에 집중하고 노이즈가 많은 벤치마크는 피하는 것이 좋습니다. [FineTasks](#) 와 [FineWeb2](#)에서 신뢰할 수 있는 평가 태스크는 네 가지 핵심 원칙으로 정의됩니다.

단조성(Monotonicity): 벤치마크 점수는 모델이 더 오래 학습할수록 일관되게 향상되어야 합니다. **낮은 노이즈(Low noise):** 동일한 설정에서 다른 랜덤 시드로 모델을 학습시킬 때, 벤치마크 점수가 크게 변동하지 않아야 합니다. **랜덤 이상의 성능(Above-random performance):** 많은 능력은 학습 후반에야 나타나므로, 오랜 기간 동안 랜덤 수준의 성능을 보이는 태스크는 절제 실험에 유용하지 않습니다. 예를 들어 객관식 형식의 MMLU가 이에 해당하며, 이는 나중에 설명하겠습니다. **순위 일관성(Ranking consistency):** 한 접근 방식이 초기 단계에서 다른 접근 방식보다 우수하다면, 이 순서는 학습이 계속되어도 안정적으로 유지되어야 합니다.

태스크의 품질은 태스크 구성 방식(모델에 질문하는 방법)과 메트릭 선택(정답 점수를 계산하는 방법)에 따라서도 달라집니다.

세 가지 일반적인 태스크 구성 방식은 객관식 형식(MCF, Multiple Choice Format), 빈칸 채우기 형식(CF, Cloze Formulation), 자유 생성 형식(FG, Freeform Generation)입니다. 객관식 형식은 모델이 프롬프트에 명시적으로 제시되고 A/B/C/D로 접두사가 붙은 여러 선택지 중에서 옵션을 선택하도록 요구합니다(예: MMLU). 빈칸 채우기 형식에서는 프롬프트에 선택지를 제공하지 않고 각 선택지의 가능성(likelihood)를 비교하여 어느 것이 더 가능성이 높은지 확인합니다. 자유 생성 형식에서는 주어진 프롬프트에 대한 그리디 생성(greedy generation)의 정확도를 봅니다. 자유 생성 형식은 모델에 많은 잠재 지식이 필요하며, 전체 학습 전의 짧은 사전학습 절제 실험에서는 태스크가 너무 어려워서 실질적으로 유용하지 않습니다. 따라서 소규모 절제 실험을 실행할 때는 객관식 형식(MCF 또는 CF)에 집중합니다.

☞ **참고 사항:** 후속 학습(post-training)된 모델의 경우, 모델이 실제로 유용한 응답을 생성할 수 있는지 평가하기 때문에 자유 생성 형식(FG)이 주요 구성 방식이 됩니다. 이러한 모델에 대한 평가는 후속 학습 장([post-training chapter](#))에서 다룰 것입니다.

연구에 따르면 모델은 학습 초기에 객관식 형식에 어려움을 겪으며, 광범위한 학습 후에야 이 기술을 습득하기 때문에 초기 신호를 얻기에는 빈칸 채우기 형식이 더 좋습니다([Du et al., 2025; Gu et al., 2025; J. Li et al., 2025](#)). 따라서 소규모 절제 실험에는 빈칸 채우기 형식을 사용하고, 모델이 객관식 형식에서 충분히 높은 신호 대 잡음비를 얻을 수 있는 임계값을 통과하면 더 나은 중간 학습(mid-training) 신호를 제공하므로 메인 실행에 객관식 형식을 통합합니다. 또한 빈칸 채우기 형

식과 같은 시퀀스 가능성도 평가에서 모델의 답변을 채점할 때, 정확도는 정답이 문자 수로 정규화된 최고 로그 확률을 가진 질문의 백분율로 계산합니다. 이 정규화는 짧은 답변에 대한 편향을 방지합니다.

[Margin note] MMLU 객관식 형식이 랜덤이 아니게 되는 시점은 모델 크기와 학습 데이터에 따라 달라집니다. 7B 트랜스포머의 경우, OLMES 논문(Gu et al., 2025)에서는 모델이 500B 토큰 이후에 랜덤이 아닌 성능을 보이기 시작한다는 것을 발견했습니다. 1.7B 모델의 경우, SmolLM2에서 이것이 6T 토큰 이후에 발생한다는 것을 발견했습니다 (Allal et al., 2025). Du et al. (2025)은 이것이 근본적으로 사전학습 손실이 특정 임계값에 도달하는 것과 관련이 있다고 주장합니다.

우리의 절제 평가 스위트에는 [FineWeb](#) 절제 실험의 벤치마크가 포함되어 있으며, 노이즈가 너무 많다고 판단된 SIQA는 제외했습니다. GSM8K와 HumanEval과 같은 수학 및 코드 벤치마크와 긴 컨텍스트 절제 실험을 위한 긴 컨텍스트 벤치마크 RULER를 추가했습니다. 아래 표에 나와 있듯이, 이 태스크 집합은 다양한 형식에 걸쳐 세계 지식, 추론, 상식을 테스트 합니다. 약간의 추가 노이즈를 감수하고 평가 속도를 높이기 위해, 각 벤치마크에서 1,000개의 질문만 평가합니다 (GSM8K, HumanEval, RULER는 예외로, 3B SmolLM3 절제 실험에서는 전체를 사용했지만 아래 1B 실험에서는 제외했습니다). 또한 위에서 설명한 대로 모든 객관식 벤치마크에 대해 빈칸 채우기 형식(CF)으로 평가합니다. 다국어 절제 실험과 실제 학습에서는 다국어 성능을 테스트하기 위한 벤치마크를 더 추가하며, 이는 나중에 자세히 설명합니다. 이러한 평가는 [LightEval](#)을 사용하여 실행되며, 아래 표는 각 벤치마크의 주요 특성을 요약합니다.

벤치마크	도메인	태스크 유형	질문 수	테스트 내용
MMLU	지식	객관식	14k	57개 과목에 걸친 광범위한 학술 지식
ARC	과학 및 추론	객관식	7k	초등학교 수준의 과학 추론
HellaSwag	상식 추론	객관식	10k	일상적인 상황에 대한 상식 추론 (서술 완성)
WinoGrande	상식 추론	이진 선택	1.7k	세계 지식이 필요한 대명사 해소
CommonSenseQA	상식 추론	객관식	1.1k	일상적인 개념에 대한 상식 추론
OpenBookQA	과학	객관식	500	추론이 필요한 기초 과학 사실
PIQA	물리적 상식	이진 선택	1.8k	일상 사물에 대한 물리적 상식
GSM8K	수학	자유 생성	1.3k	초등학교 수학 문장형 문제
HumanEval	코드	자유 생성	164	독스트링으로부터 Python 함수 합성

이러한 평가가 실제로 무엇을 테스트하는지 구체적으로 이해하기 위해 각 벤치마크의 몇 가지 예시 질문을 살펴보겠습니다.

The screenshot shows a table interface for a dataset named "mmlu". The table has three columns: "question", "choices", and "answer".

question	choices	answer
string · classes A homeowner purchased a new vacuum cleaner. A few days later...	list · lengths ["recover, because the homeowner knew about the..."	string · classes 0
What is meant by the phrase CSR?	["Corporate Social Responsibility", "Company..."	0
Which of the following is a true statement?	["While observational studies gather information..."	3
In response to Sandel's "social justice" argument, Kamm argues...	["even if we were able to enhance ourselves or others..."	3

위의 예시들을 살펴보면서 각 벤치마크의 질문 유형을 확인해 보세요. MMLU와 ARC가 객관식으로 사실적 지식을 테스트하는 반면, GSM8K는 수학 문제에 대한 수치 답을 계산해야 하고, HumanEval은 완전한 Python 코드를 생성해야 한다는 점에 주목하세요. 이러한 다양성은 절제 실험 전반에 걸쳐 모델 능력의 다양한 측면을 테스트할 수 있도록 보장합니다.

절제 실험을 위한 데이터 혼합은?

아키텍처 절제 실험의 경우, 광범위한 태스크에서 조기 신호를 제공하는 고품질 데이터셋의 고정된 혼합으로 학습합니다. 영어([FineWeb-Edu](#)), 수학([FineMath](#)), 코드([Stack-Edu-Python](#))를 사용합니다. 아키텍처 관련 발견은 다국어 데이터를 포함한 다른 데이터셋과 도메인에도 잘 일반화되어야 하므로, 데이터 혼합을 단순하게 유지할 수 있습니다.

데이터 절제 실험의 경우, 반대 접근 방식을 취합니다. 아키텍처를 고정하고 데이터 혼합을 체계적으로 변경하여 다른 데이터 소스가 모델 성능에 어떻게 영향을 미치는지 이해합니다.

때때로 평가에서의 차이가 작을 수 있습니다. 충분한 컴퓨팅 자원이 있다면, 결과가 얼마나 변동하는지 확인하기 위해 다른 시드로 동일한 절제 실험을 다시 실행해 볼 가치가 있습니다.

건고한 절제 실험 설정의 진정한 가치는 단순히 좋은 모델을 구축하는 것 이상입니다. 메인 학습 실행 중에 문제가 발생하면 (그리고 아무리 준비해도 반드시 발생합니다), 우리가 내린 모든 결정에 확신을 갖고 제대로 테스트되지 않아 문제를 일으킬 수 있는 구성 요소를 신속하게 식별하고 싶습니다. 이러한 준비는 디버깅 시간을 절약하고 미래의 정신 건강을 보호합니다.

Ablation 비용 추정

절제 실험은 훌륭하지만 GPU 시간이 필요하며, 이러한 실험의 비용을 이해할 가치가 있습니다. 아래 표는 SmoLM3 사전 학습에 대한 전체 컴퓨팅 분석을 보여줍니다. 메인 실행(간헐적인 다운타임 포함), 학습 전후의 절제 실험, 그리고 재시작과 일부 디버깅을 강제한 예상치 못한 스케일링 문제에 소비된 컴퓨팅(이는 나중에 자세히 설명하겠습니다)이 포함됩니다.

단계	GPU	일수	GPU 시간
메인 사전학습 실행	384	30	276,480
절제 실험 (사전학습)	192	15	69,120
절제 실험 (중간 학습)	192	10	46,080
학습 재설정 및 디버깅	384/192	3/4	46,080
총 비용	-	-	437,760

평가 비용은 10,000 GPU 시간 미만으로 추정됩니다. 전체 평가 스위트(영어, 다른 언어, 수학 및 코드)는 GPU당 약 1.5시간이 소요되며, 11T 토큰 전반에 걸쳐 10B 토큰마다 평가하고, 여기에 수많은 절제 실험이 추가됩니다. 긴 컨텍스트 평가는 특히 비용이 많이 들어, 실행당 8개 GPU에서 약 1시간이 소요되었습니다.

이 수치는 중요한 사실을 보여줍니다. 절제 실험과 디버깅은 총 **161,280 GPU 시간**을 소비했으며, 이는 메인 학습 실행 비용(**276,480 GPU 시간**)의 절반 이상입니다. SmoLM3 개발 전반에 걸쳐 총 100개 이상의 절제 실험을 실행했습니다. 사전학습 절제 실험에 20일, 중간 학습 절제 실험에 10일, 그리고 재시작과 일부 디버깅을 강제한 예상치 못한 학습 문제를 복구하는 데 7일을 소비했습니다(이는 나중에 자세히 설명하겠습니다).

이는 절제 실험 비용이 컴퓨팅 예산에 반영되어야 하는 이유를 강조합니다. 학습 비용에 절제 실험 비용과 예상치 못한 상황에 대한 버퍼를 더해 계획하세요. SOTA 성능을 목표로 하거나, 새로운 아키텍처 변경을 구현하거나, 이미 검증된 레시피가 없다면, 절제 실험은 사소한 실험이 아닌 상당한 비용 센터가 됩니다.

DeepSeek-V3가 출시되었을 때, 세계는 보고된 560만 달러의 학습 비용에 주목했습니다. 많은 사람들이 이 숫자를 전체 R&D 비용으로 해석했습니다. 실제로 이는 최종 학습 실행만을 반영합니다. 훨씬 더 크고 보통 보이지 않는 비용은 연구 자체에 있습니다. 최종 레시피로 이어지는 절제 실험, 실패한 실행, 디버깅 말입니다. 모델의 규모와 참신함을 고려할 때, 그들의 연구 비용은 확실히 더 높았을 것입니다.

다음 섹션으로 넘어가기 전에, 실험을 실행하는 모든 사람이 따라야 할 몇 가지 기본 규칙을 정립하겠습니다.

참여 규칙

요약: 편집증적으로 의심하세요.

평가 스위트를 검증하세요. 모델을 학습시키기 전에, 평가 스위트가 비교할 모델들의 공개된 결과를 재현할 수 있는지 확인하세요. 벤치마크가 생성적 성격인 경우(예: GSM8k), 더욱 편집증적으로 몇 가지 샘플을 수동으로 검사하여 프롬프트가 올바르게 포맷되었는지, 후처리가 올바른 정보를 추출하고 있는지 확인하세요. 평가가 모든 결정을 안내할 것으로, 이 단계를 올바르게 하는 것이 프로젝트 성공에 매우 중요합니다!

아무리 작아 보여도 모든 변경을 테스트하세요. 겉보기에 무해한 라이브러리 업그레이드나 "단 두 줄만 변경한" 커밋의 영향을 과소평가하지 마세요. 이러한 작은 변경은 결과를 오염시킬 미묘한 버그나 성능 변화를 도입할 수 있습니다. 회귀를 방지하기 위해 중요한 케이스에 대한 강력한 테스트 스위트가 있는 라이브러리가 필요합니다.

일부 경우에는 라이브러리를 최신 버전으로 업그레이드하여 버그를 해결할 수 있습니다. 팀정 같은 디버깅의 아름다운 예시는 [Elana Simon의 블로그 포스트](#)를 참조하세요.

한 번에 하나만 변경하세요. 실험 간에 다른 모든 것은 동일하게 유지하세요. 일부 변경은 예상치 못한 방식으로 서로 상호작용할 수 있으므로, 먼저 각 변경의 개별 기여를 평가한 다음, 이를 조합하여 전체적인 영향을 확인하고자 합니다.

충분한 토큰으로 학습하고 충분한 평가를 사용하세요. 앞서 언급했듯이, 평가 스위트에서 좋은 커버리지를 확보하고 신뢰할 수 있는 신호를 얻기 위해 충분히 오래 학습해야 합니다. 여기서 지름길을 택하면 노이즈가 많은 결과와 나쁜 결정으로 이어질 것입니다.

이러한 규칙을 따르는 것이 지나치게 조심스럽게 느껴질 수 있지만, 대안은 며칠 전의 관련 없는 의존성 업데이트로 인해 발생한 신비로운 성능 저하를 디버깅하는 데 며칠을 보내는 것입니다. 황금 원칙: 좋은 설정이 완성되면, 어떤 변경도 테스트 없이 진행해서는 안 됩니다!

모델 아키텍처 설계

이제 실험 프레임워크가 준비되었으니, 모델을 정의할 큰 결정을 내릴 때입니다. 모델 크기부터 어텐션 메커니즘, 토크나이저 선택까지 우리가 내리는 모든 선택은 모델 학습과 사용에 영향을 미칠 제약과 기회를 만들어냅니다.

학습 나침반([training compass](#))을 기억하세요. 기술적 선택을 하기 전에 **왜와 무엇에** 대한 명확성이 필요합니다. 왜 이 모델을 학습시키며, 어떤 모습이어야 할까요?

당연해 보이지만, 학습 나침반에서 설명했듯이 여기서 신중하게 결정하면 우리의 결정을 형성하고 가능한 실험의 끝없는 공간에서 길을 잊지 않게 해줍니다. 영어에서 SOTA 모델을 목표로 하고 있나요? 긴 컨텍스트가 우선순위인가요? 아니면 새로운 아키텍처를 검증하려고 하나요? 학습 루프는 이 모든 경우에 비슷해 보일 수 있지만, 실행하는 실험과 수용하는 트레이드 오프는 다를 것입니다. 이 질문에 일찍 답하면 데이터와 아키텍처 사이에 시간을 어떻게 균형 있게 배분할지, 그리고 실행을 시작하기 전에 각각에서 얼마나 혁신할지 결정하는 데 도움이 됩니다.

그래서 예시를 통해 SmoILM3의 설계를 이끈 목표들을 살펴보겠습니다. 우리는 경쟁력 있는 다국어 성능, 견고한 수학 및 코딩 능력, 그리고 강력한 긴 컨텍스트 처리 기능을 갖춘 온디바이스 애플리케이션용 강력한 모델을 원했습니다. 앞서 언급했듯이, 이로 인해 3B 파라미터의 밀집 모델을 선택하게 되었습니다. 강력한 능력을 위해 충분히 크지만 휴대폰에 편안하게 맞을 만큼 충분히 작습니다. 엣지 디바이스의 메모리 제약과 프로젝트 일정(약 3개월)을 고려하여 MoE나 하이브리드 대신 밀집 트랜스포머를 선택했습니다.

더 작은 규모(1.7B 파라미터)에서 영어용 SmoILM2의 작동하는 레시피가 있었지만, 스케일업은 모든 것을 재검증하고 다른 국어 및 확장된 컨텍스트 길이와 같은 새로운 도전을 해결해야 함을 의미했습니다. 명확한 목표가 우리 접근 방식을 어떻게 형성했는지 보여주는 좋은 예가 있습니다. 예를 들어, SmoILM2에서는 사전학습 말미에 컨텍스트 길이를 확장하는 데 어려움을 겪었기 때문에, SmoILM3에서는 처음부터 NoPE와 문서 내 마스킹(나중에 설명)과 같은 아키텍처적 선택을 하여 성공 가능성을 최대화했고, 이것이 효과가 있었습니다.

SmoILM2는 온디바이스 배포를 위해 설계된 135M, 360M, 1.7B 파라미터의 세 가지 변형을 가진 이전 세대의 소형 언어 모델이었습니다. 영어 전용이었고 8k 컨텍스트 길이를 가졌습니다.

목표가 명확해지면, 이를 실현할 기술적 결정을 시작할 수 있습니다. 이 장에서는 아키텍처, 데이터, 하이퍼파라미터라는 핵심 결정에 대한 체계적인 접근 방식을 살펴볼 것입니다. 이것을 전략적 계획 단계로 생각하세요. 이러한 기본 사항을 올바르게 하면 실제 학습 마라톤 동안의 비용이 많이 드는 실수를 방지할 수 있습니다.

아키텍처 선택

Qwen3, Gemma3, DeepSeek v3와 같은 최근 모델들을 보면, 차이점에도 불구하고 모두 동일한 기반인 2017년에 소개된 트랜스포머 아키텍처([Vaswani et al., 2023](#))를 공유하고 있음을 알 수 있습니다. 수년간 변한 것은 근본적인 구조가 아니라 핵심 구성 요소에 대한 개선입니다. 밀집 모델, Mixture of Experts, 하이브리드 아키텍처 중 무엇을 구축하든, 동일한 빌딩 블록으로 작업하게 됩니다.

이러한 개선은 더 나은 성능을 추구하고 특정 도전을 해결하는 팀들에서 나왔습니다. 추론 시 메모리 제약, 대규모 학습 불안정성, 또는 더 긴 컨텍스트를 처리할 필요성 등입니다. Multi-Head Attention(MHA)에서 Grouped Query Attention(GQA)([Ainslie et al., 2023](#))과 같은 더 컴퓨팅 효율적인 어텐션 변형으로의 전환과 같은 일부 수정은 이제 널리 채택되었습니다. 다른 위치 인코딩 방식과 같은 다른 수정은 여전히 논쟁 중입니다. 결국, 오늘의 실험은 내일의 기준선으로 결정화될 것입니다.

그렇다면 현대 LLM은 실제로 오늘날 무엇을 사용할까요? 선도적인 모델들이 수렴한 것을 살펴보겠습니다. 안타깝게도 모든 모델이 학습 세부 사항을 공개하지는 않지만, DeepSeek, OLMo, Kimi, SmoLLM과 같은 패밀리에서 현재 환경을 볼 수 있을 만큼 충분한 투명성이 있습니다.

모델	아키텍처	파라미터	학습 토큰	어텐션	컨텍스트 길이 (최종)	위치 인코딩	정밀도	초기화 (std)	옵티마이저	최대 LR	LR 스케줄	웜업 스텝	배치 크기
DeepSeek LLM 7B	Dense	7B	2T	GQA	4K	RoPE	BF16	0.006	AdamW	4.2×10^{-4}	Multi-Step	2K	9.4M
DeepSeek LLM 67B	Dense	67B	2T	GQA	4K	RoPE	BF16	0.006	AdamW	3.2×10^{-4}	Multi-Step	2K	18.9M
DeepSeek V2	MoE	236B (21B active)	8.1T	MLA	128K	Partial RoPE	-	0.006	AdamW	2.4×10^{-4}	Multi-Step	2K	$9.4M \rightarrow 37.7M$ (warmup 225B)
DeepSeek V3	MoE	671B (37B active)	14.8T	MLA	129K	Partial RoPE	FP8	0.006	AdamW	2.2×10^{-4}	Multi-Step + Cosine	2K	$12.6M \rightarrow 62.9M$ (warmup 469B)
MiniMax-01	MoE + Hybrid	456B (45.9 active)	11.4T	Linear attention + GQA	4M	Partial RoPE	-	Xavier init with deepnorm scaling	AdamW	2×10^{-4}	Multi-Step	500	$16M \rightarrow 32M \rightarrow 64M \rightarrow 128M$
Kimi K2	MoE	1T (32B active)	15.5T	MLA	128K	Partial RoPE	BF16	likely 0.006	MuonClip	2×10^{-4}	WSD	500	67M
OLMo 2 7B	Dense	7B	5T	MHA	4K	RoPE	BF16	0.02	AdamW	3×10^{-4}	Cosine	2K	4.2M
SmoLLM3	Dense	3B	11T	GQA	128K	NoPE	BF16	0.02	AdamW	2×10^{-4}	WSD	2K	2.3M

MLA, NoPE, WSD와 같은 일부 용어를 아직 이해하지 못하더라도 걱정하지 마세요. 이 섹션에서 각각을 설명할 것입니다. 지금은 다양한 어텐션 메커니즘(MHA, GQA, MLA), 위치 인코딩(RoPE, NoPE, partial RoPE), 학습률 스케줄(Cosine, Multi-Step, WSD)의 다양성에 주목하세요.

이 긴 아키텍처 선택 목록을 보면 어디서부터 시작해야 할지 다소 압도적입니다. 대부분의 그런 상황처럼, 단계별로 접근하여 점차적으로 필요한 모든 노하우를 쌓아갈 것입니다. 가장 단순한 기본 아키텍처(밀집 모델)에 먼저 집중하고 각 아키텍처 측면을 자세히 조사할 것입니다. 나중에 MoE와 하이브리드 모델을 깊이 다루고 언제 사용하는 것이 좋은 선택인지 논의할 것입니다. 마지막으로 종종 간과되고 과소평가되는 구성 요소인 토크나이저를 탐구합니다. 기준 것을 사용해야 할까요, 아니면 직접 학습시켜야 할까요? 토크나이저가 좋은지 어떻게 평가할 수 있을까요?

📍 절제 실험 설정

이 장의 나머지 부분에서, 위 장에서 설명한 설정을 사용한 절제 실험을 통해 대부분의 아키텍처 선택을 검증합니다.

*FineWeb-Edu, FineMath, Python-Edu*의 혼합에서 45B 토큰으로 학습된 1B 기준 모델(Llama3.2 1B 아키텍처를 따름)입니다. 각 실험에서 학습 손실 곡선과 다운스트림 평가 점수를 모두 보여주어 각 수정의 영향을 평가합니다. 모든 실행의 설정은 [HuggingFaceTB/training-guide-nanotron-configs](#)에서 찾을 수 있습니다.

Sebastian Raschka의 이 블로그 포스트는 2025년 현대 LLM 아키텍처에 대한 좋은 개요를 제공합니다.

이제 모든 LLM의 핵심인 어텐션 메커니즘부터 시작하겠습니다.

어텐션

트랜스포머 아키텍처를 둘러싼 가장 활발한 연구 영역 중 하나는 어텐션 메커니즘입니다. 피드포워드 레이어가 사전학습 종 컴퓨팅을 지배하는 반면, 어텐션은 추론 시(특히 긴 컨텍스트에서) 주요 병목이 되어 컴퓨팅 비용을 높이고 KV 캐시가 GPU 메모리를 빠르게 소비하여 처리량을 줄입니다. 주요 어텐션 메커니즘과 이들이 용량과 속도 사이에서 어떻게 트레이드오프 하는지 간략히 살펴보겠습니다.

어텐션에 몇 개의 헤드가 필요할까요?

Multi-head attention(MHA)은 원래 트랜스포머([Vaswani et al., 2023](#))와 함께 소개된 표준 어텐션입니다. 핵심 아이디어는 N개의 어텐션 헤드가 각각 독립적으로 동일한 검색 태스크를 수행한다는 것입니다. 은닉 상태를 쿼리, 키, 값으로 변환한 다음, 현재 쿼리를 사용하여 키 매칭으로 가장 관련 있는 토큰을 검색하고, 마지막으로 매칭된 토큰과 연관된 값을 전달합니다. 추론 시에는 과거 토큰에 대한 KV 값을 다시 계산할 필요가 없고 재사용할 수 있습니다. 과거 KV 값을 위한 메모리를 **KV 캐시**라고 합니다. 컨텍스트 윈도우가 커짐에 따라, 이 캐시는 빠르게 추론 병목이 되어 GPU 메모리의 큰 부분을 소비할 수 있습니다. 다음은 MHA와 8192 시퀀스 길이를 가진 Llama 3 아키텍처의 KV 캐시 메모리 s_{KV} 를 추정하는 간단한 계산입니다.

빠른 복습을 위해 Jay Alammar의 유명한 블로그 포스트를 확인하세요!

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

선행 인자 2는 키와 값 캐시를 모두 저장하기 때문입니다. 보시다시피, 캐시는 시퀀스 길이에 따라 선형적으로 증가하지만, 컨텍스트 윈도우는 기하급수적으로 성장하여 이제 수백만 토큰에 도달합니다. 따라서 캐시의 효율성을 개선하면 추론 시 컨텍스트 스케일링이 훨씬 쉬워질 것입니다.

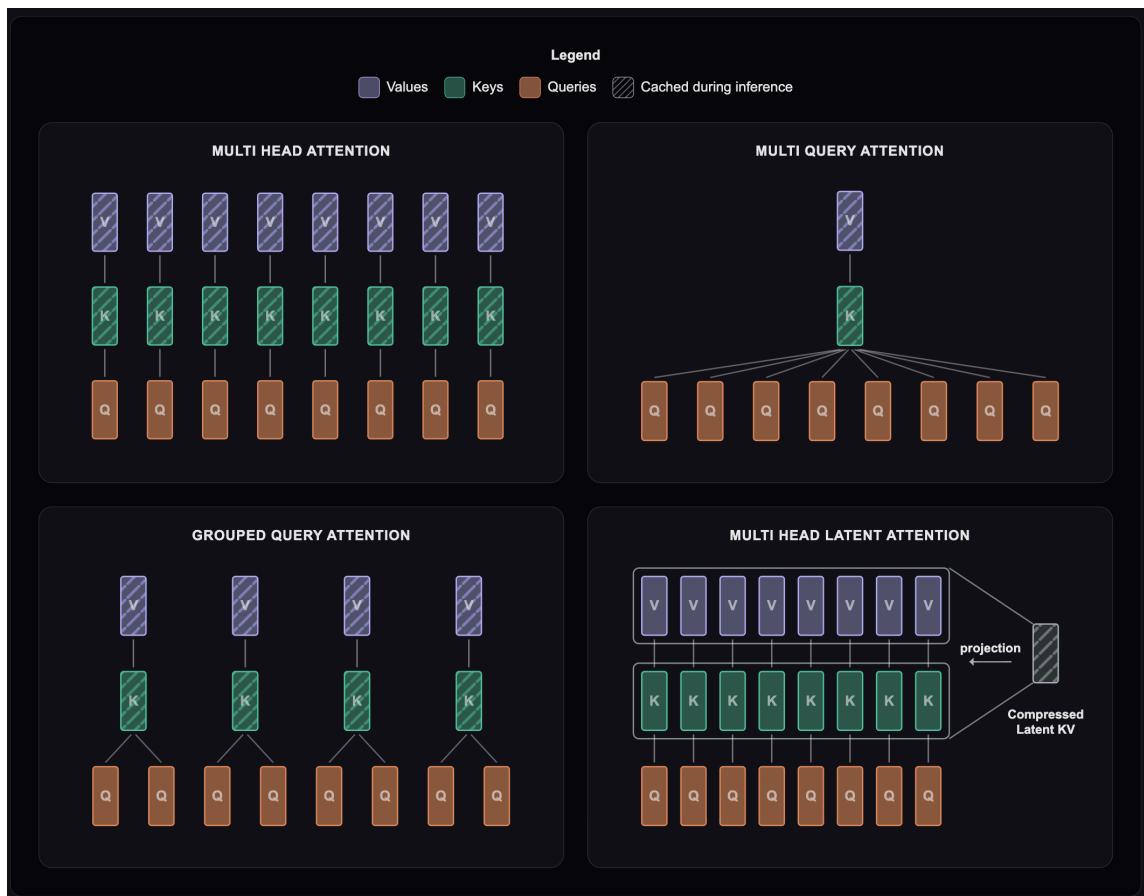
자연스럽게 물어볼 질문은 각 헤드에 대해 정말로 새로운 KV 값이 필요한가?입니다. 아마도 아닐 것이며, Multi-Query Attention(MQA)([Shazeer, 2019](#))과 Grouped Query Attention(GQA)([Ainslie et al., 2023](#)) 모두 이 문제를 해결합니다. 가장 단순한 경우는 모든 헤드에서 KV 값을 공유하여 KV 캐시 크기를 n_{heads} 로 나누는 것입니다. 예를 들어 Llama 3 70B의 경우 64배 감소입니다! 이것이 MQA의 아이디어이며 MHA의 대안으로 StarCoder와 같은 일부 모델에서 사용되었습니다. 그러나 우리가 원하는 것보다 더 많은 어텐션 용량을 포기할 수 있으므로, 중간 지점을 고려하여 헤드 그룹 간에 KV 값을 공유할 수 있습니다. 예를 들어 4개의 헤드가 동일한 KV 값을 공유합니다. 이것이 GQA 접근 방식이며 MQA와 MHA 사이의 중간 지점을 취합니다.

더 최근에 DeepSeek-v2(그리고 v3에서도 사용됨)는 Multi-Latent Attention(MLA)([DeepSeek-AI et al., 2024](#))을 도입했으며, 이는 캐시를 압축하기 위한 다른 전략을 사용합니다. KV 값의 수를 줄이는 대신 크기를 줄이고, 런타임에 KV 값으로 압축 해제할 수 있는 잠재 변수를 저장합니다. 이 접근 방식으로 MHA보다 더 강한 성능을 제공하면서 2.25 그룹을 가진 GQA와 동등한 수준으로 캐시를 줄일 수 있습니다! RoPE와 함께 작동하려면 추가 작은 잠재 벡터와 함께 약간의 조

정이 필요합니다. DeepSeek-v2에서는 주요 잠재 변수에 $4 * \dim_{\text{head}}$ 를, RoPE 부분에 $1/2 * \dim_{\text{head}}$ 를 선택했으므로 총 $4.5 * \dim_{\text{head}}$ 이며, 이는 K와 V 모두에 동시에 사용되어 선행 인자 2를 제거합니다.

RoPE(Rotary Position Embeddings)는 시퀀스에서의 위치를 기반으로 쿼리와 키 벡터를 회전시켜 위치 정보를 인코딩하는 방법입니다. 오늘날의 LLM에서 일반적으로 사용됩니다.

다음 그래픽에서 각 어텐션 메커니즘의 시각적 설명을 볼 수 있습니다.



Multi-Head Attention(MHA), Grouped-Query Attention(GQA), Multi-Query Attention(MQA), Multi-head Latent Attention(MLA)의 간략화된 그림. 키와 값을 잠재 벡터로 함께 압축함으로써, MLA는 추론 시 KV 캐시를 크게 줄입니다.

다음 표는 이 섹션에서 방금 논의한 어텐션 메커니즘을 비교합니다. 단순화를 위해 토큰당 사용되는 파라미터를 비교하며, 총 메모리를 계산하려면 파라미터당 바이트 수(일반적으로 2)와 시퀀스 길이를 곱하면 됩니다.

어텐션 메커니즘	토큰당 KV 캐시 파라미터
MHA	$= 2 \times n_{\text{heads}} \times n_{\text{layers}} \times \dim_{\text{head}}$
MQA	$= 2 \times 1 \times n_{\text{layers}} \times \dim_{\text{head}}$
GQA	$= 2 \times g \times n_{\text{layers}} \times \dim_{\text{head}}$ (일반적으로 $g=2,4,8$)
MLA	$= 4.5 \times n_{\text{layers}} \times \dim_{\text{head}}$

이제 이러한 어텐션 메커니즘이 실제 실험에서 어떤 성능을 보이는지 살펴보겠습니다!

Ablation - GQA가 MHA를 능가함

여기서는 다양한 어텐션 메커니즘을 비교합니다. [baseline](#)은 32개의 헤드와 8개의 KV 헤드를 사용하며, 이는 비율 $32/8=4$ 인 GQA에 해당합니다. MHA를 사용하거나 더 적은 KV 헤드와 더 높은 GQA 비율을 사용하면 성능이 어떻게 변할까요?

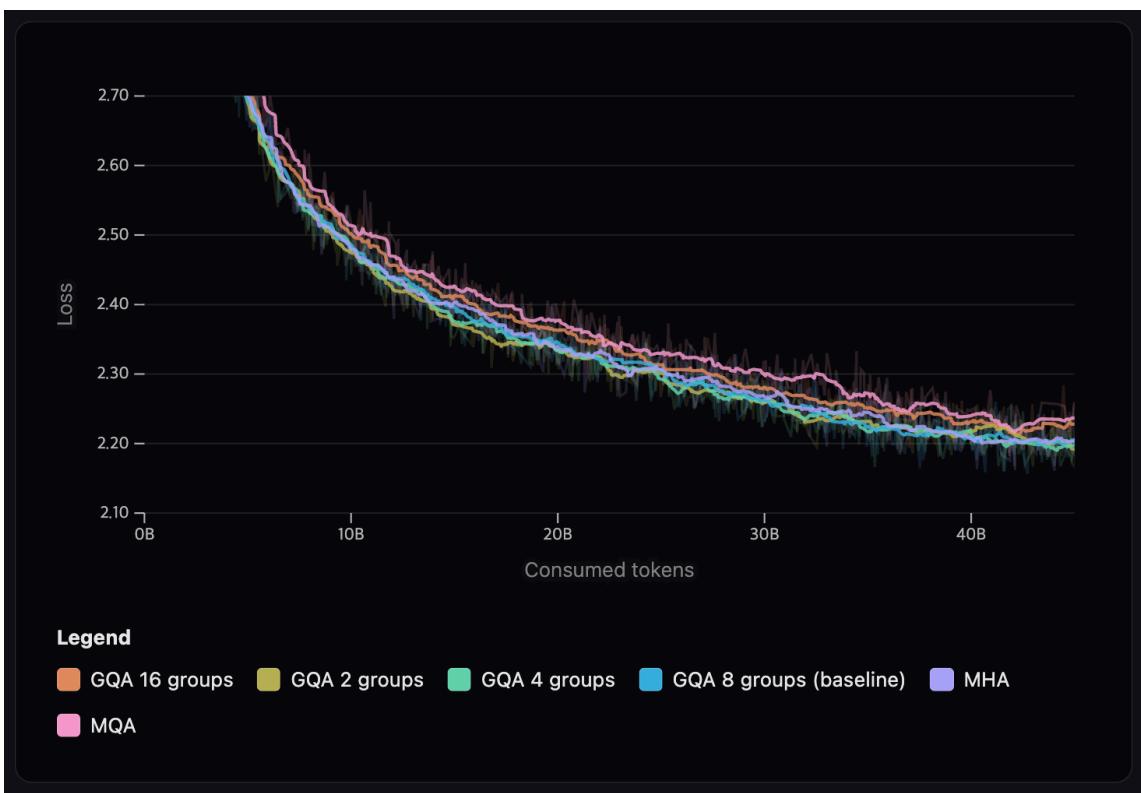
일부 라이브러리에서는 GQA 비율을 쿼리 그룹 = 쿼리 헤드 / KV 헤드라고 부릅니다.

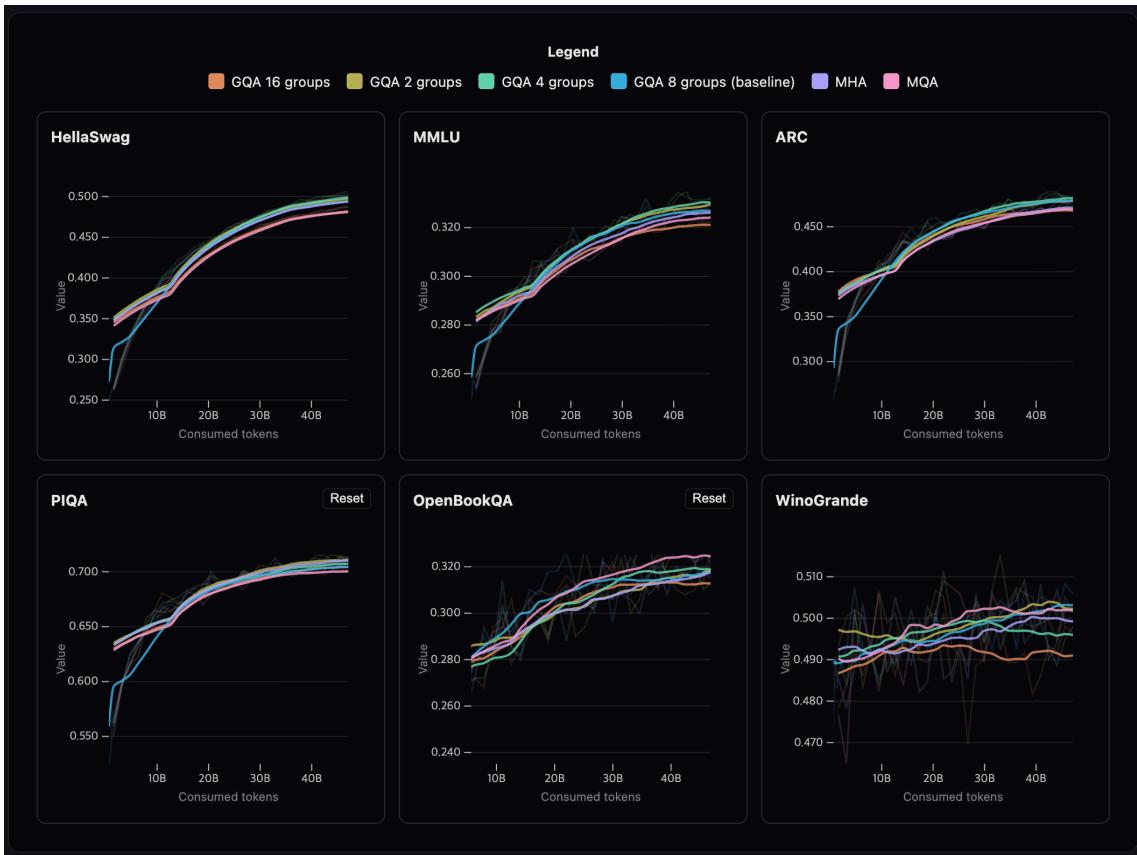
KV 헤드 수를 변경하면 특히 MHA의 경우 파라미터 수에 영향을 미칩니다. 일관성을 위해, MHA 실행의 경우 레이어 수를 조정했는데, 그렇지 않으면 1억 개 이상의 파라미터 차이가 발생하기 때문입니다. 나머지는 기본 16개 레이어를 유지합니다.

어텐션 유형	쿼리 헤드	KV 헤드	레이어	파라미터 수	비고
MQA	32	1	16	1.21B	
GQA (비율 16)	32	2	16	1.21B	
GQA (비율 8)	32	4	16	1.22B	우리의 기준 모델
GQA (비율 4)	32	8	16	1.24B	
GQA (비율 2)	32	16	15	1.22B	레이어 감소
MHA	32	32	14	1.20B	레이어 감소
GQA (비율 2)	32	16	16	1.27B	너무 큼 - 절제 실험 제외
MHA	32	32	16	1.34B	너무 큼 - 절제 실험 제외

따라서 MHA, MQA, 그리고 4가지 GQA 설정(비율 2, 4, 8, 16)을 비교합니다. nanotron 설정은 [여기서](#) 찾을 수 있습니다.

절제 실험 결과를 보면, 16개 그룹을 가진 MQA와 GQA(각각 1개와 2개의 KV 헤드만 남김)가 MHA보다 상당히 낮은 성능을 보입니다. 반면에, 2, 4, 8 그룹을 가진 GQA 구성은 MHA 성능과 대략 일치합니다.





결과는 손실 곡선과 다운스트림 평가 모두에서 일관됩니다. HellaSwag, MMLU, ARC와 같은 벤치마크에서 이를 명확하게 관찰할 수 있으며, OpenBookQA와 WinoGrande와 같은 벤치마크에서는 약간의 노이즈가 보입니다.

이러한 절제 실험을 바탕으로, GQA는 MHA의 견고한 대안입니다. 추론에서 더 효율적이면서 성능을 유지합니다. 일부 최근 모델들은 더 큰 KV 캐시 압축을 위해 MLA를 선택했지만, 아직 널리 채택되지는 않았습니다. 절제 실험 당시 nanotron에 MLA가 구현되어 있지 않았기 때문에 MLA에 대한 절제 실험은 수행하지 않았습니다. SmolLM3에서는 4개 그룹을 가진 GQA를 사용했습니다.

어텐션 아키텍처 자체를 넘어서, 학습 중에 사용하는 어텐션 패턴도 중요합니다. 어텐션 마스킹에 대해 살펴보겠습니다.

문서 마스킹

학습 시퀀스에 어텐션을 어떻게 적용하는지는 계산 효율성과 모델 성능 모두에 영향을 미칩니다. 이것은 문서 마스킹과 데이터로더에서 학습 샘플을 어떻게 구성하는지에 대한 더 넓은 질문으로 이어집니다.

사전학습 중에는 고정된 시퀀스 길이로 학습하지만 문서의 길이는 가변적입니다. 연구 논문은 10k 토큰일 수 있지만 짧은 코드 스니펫은 수백 토큰에 불과할 수 있습니다. 가변 길이 문서를 고정 길이 학습 시퀀스에 어떻게 맞출까요? 짧은 문서에 패딩을 추가하여 목표 길이에 도달하는 것은 의미 없는 패딩 토큰에 컴퓨팅을 낭비하게 됩니다. 대신 **패킹(packing)**을 사용합니다. 문서를 셔플하고 시퀀스 종료(EOS) 토큰과 함께 연결한 다음, 결과를 시퀀스 크기에 맞는 고정 길이 청크로 분할합니다.

문서 시작 부분에 *BOS* 토큰을 추가할 수도 있습니다. 이 경우 모델/토크나이저 설정에 다른 *bos_token_id* 가 있음을 알 수 있습니다.

실제로 이것이 어떻게 보이는지 살펴보겠습니다.

```
파일 1: "그래놀라 바 레시피..." (400 토큰) <EOS>
파일 2: "def hello_world()..." (300 토큰) <EOS>
파일 3: "기후 변화 영향..." (1000 토큰) <EOS>
파일 4: "import numpy as np..." (3000 토큰) <EOS>
...
...
```

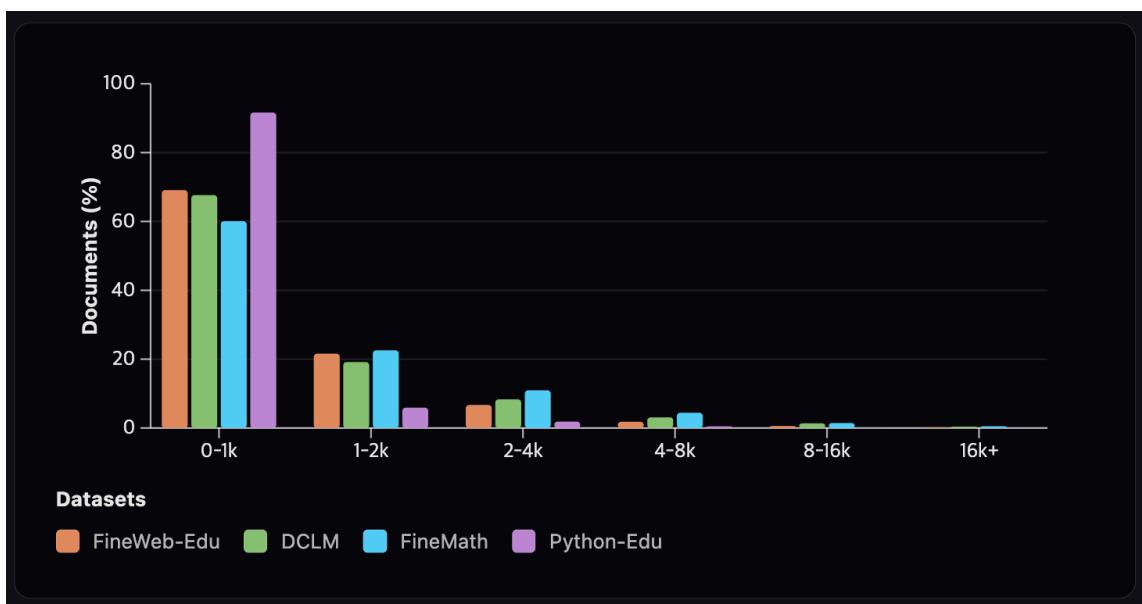
4k 시퀀스로 연결 및 청킹 후:

```
시퀀스 1: [파일 1] + [파일 2] + [파일 3] + [파일 4 일부]
시퀀스 2: [파일 4 나머지] + [파일 5] + [파일 6] + ...
```

학습 시퀀스는 4k 컨텍스트를 채울 만큼 충분히 긴 경우 하나의 완전한 파일을 포함할 수 있지만, 대부분의 경우 파일이 짧기 때문에 시퀀스는 여러 랜덤 파일의 연결을 포함합니다.

표준 인과적 마스킹(causal masking)에서는 토큰이 패킹된 시퀀스의 모든 이전 토큰에 어텐션을 줄 수 있습니다. 위의 예에서, 파일 4의 Python 함수에 있는 토큰은 그래놀라 바 레시피, 기후 변화 기사, 그리고 함께 패킹된 다른 모든 콘텐츠에 어텐션을 줄 수 있습니다. 일반적인 4k 사전학습 컨텍스트가 무엇을 포함하는지 빠르게 살펴보겠습니다. 간단한 [분석](#)에 따르면 CommonCrawl과 GitHub 파일의 상당 부분(약 80-90%)이 2k 토큰보다 짧습니다.

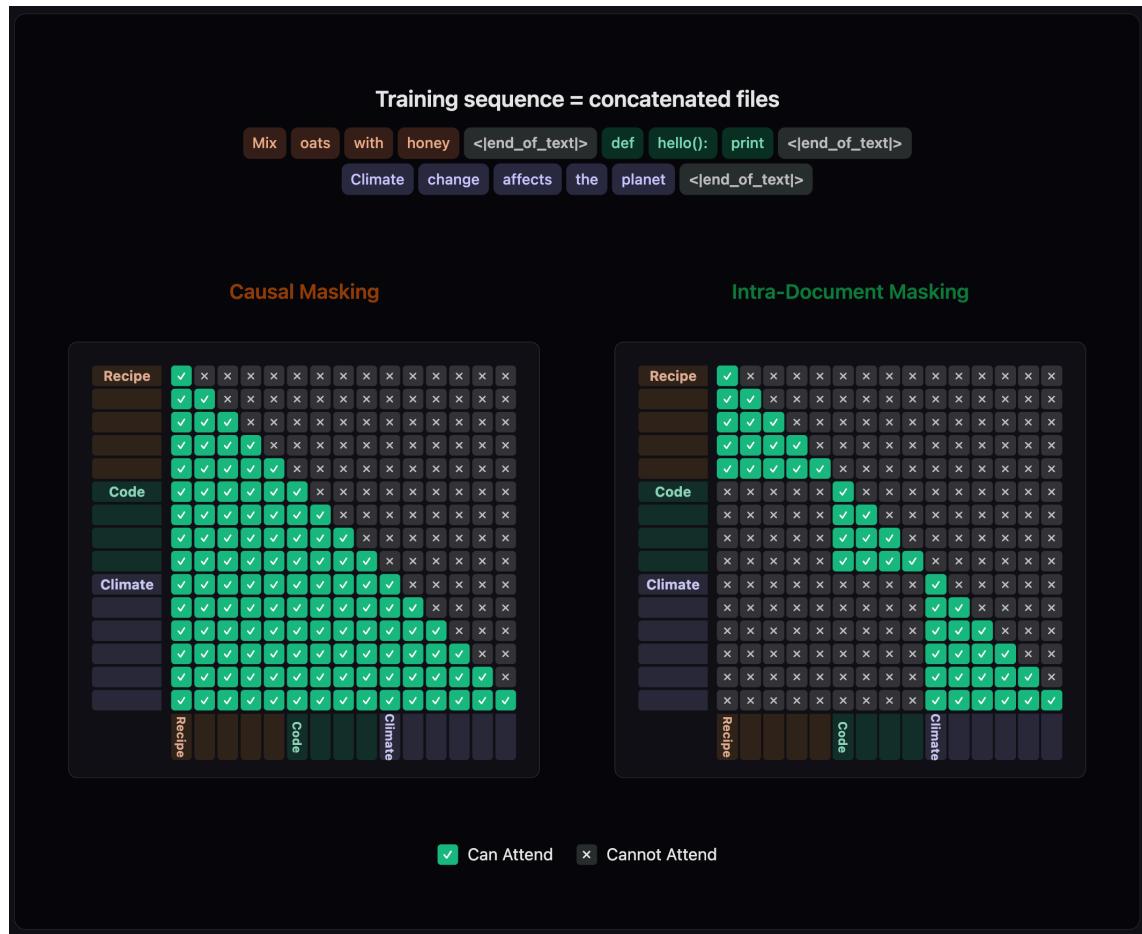
아래 차트는 이 블로그 전반에 걸쳐 사용된 최근 데이터셋의 토큰 분포를 살펴봅니다.



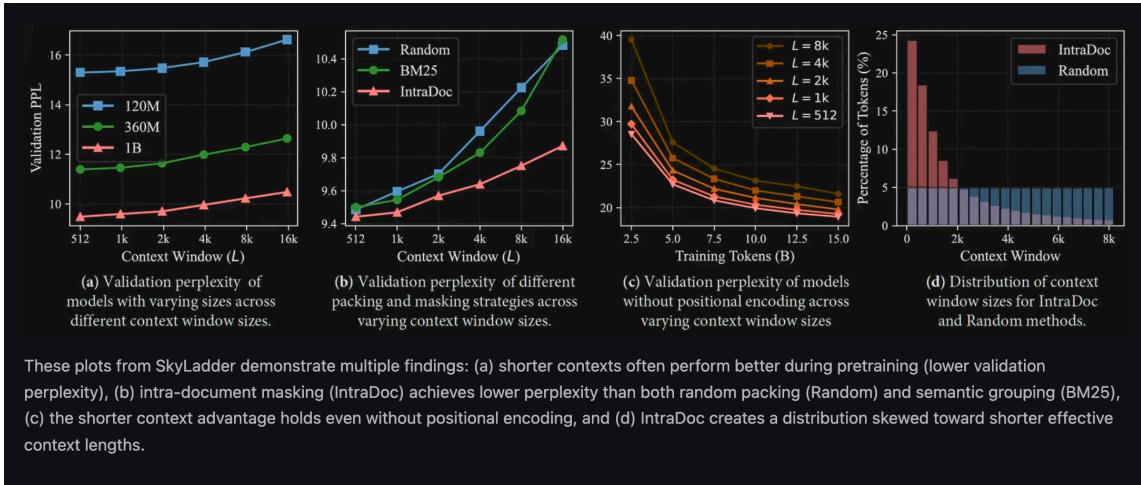
FineWeb-Edu, DCLM, FineMath, Python-Edu의 문서 중 80% 이상이 2k 토큰 미만을 포함합니다. 이는 2k 또는 4k 학습 시퀀스와 표준 인과적 마스킹을 사용할 경우, 대다수의 토큰이 함께 패킹된 관련 없는 문서에 어텐션을 주는데 컴퓨팅을 소비하게 된다는 것을 의미합니다.

PDF의 더 긴 문서 대부분의 웹 기반 데이터셋이 짧은 문서로 구성되어 있는 반면, PDF 기반 데이터셋은 상당히 더 긴 콘텐츠를 포함합니다. [FinePDFs](#) 문서는 평균적으로 웹 텍스트보다 2배 더 깁니다. 그리고 FineWeb-Edu 및 DCLM 과 혼합할 때 성능을 향상시킵니다.

계산 비효율성 외에도, [Zhao et al. \(2024\)](#)은 이 접근 방식이 관련 없는 콘텐츠로부터 노이즈를 도입하여 성능을 저하시킬 수 있다는 것을 발견했습니다. 그들은 **문서 내 마스킹(intra-document masking)**을 사용할 것을 제안하는데, 이는 토큰이 동일한 문서 내의 이전 토큰에만 어텐션을 줄 수 있도록 어텐션 마스크를 수정하는 것입니다. 아래 시각화는 이 차이를 보여줍니다.



[Zhu et al. \(2025\)](#)은 SkyLadder에서 문서 내 마스킹으로부터 유사한 이점을 발견했지만, 다른 설명을 제시합니다. 그들은 더 짧은 컨텍스트 길이가 학습에 더 효과적이며, 문서 내 마스킹이 효과적으로 평균 컨텍스트 길이를 줄인다는 것을 발견했습니다.



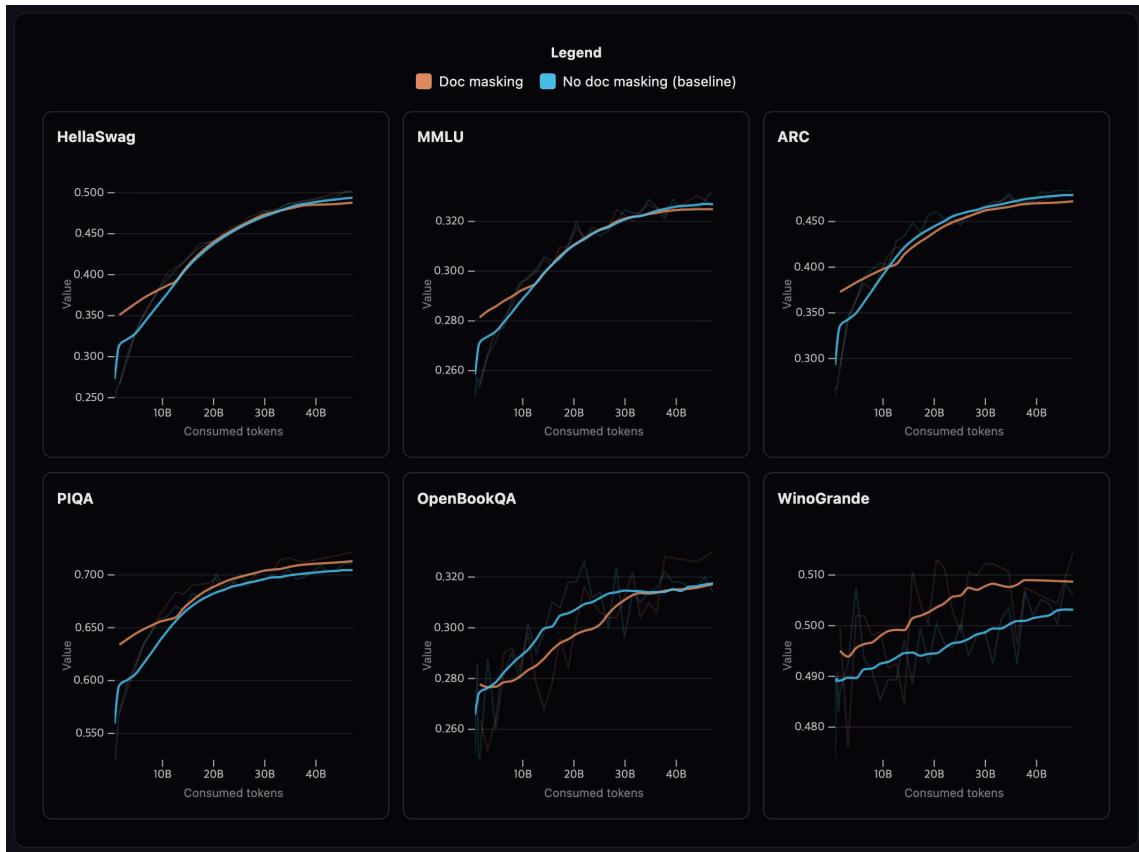
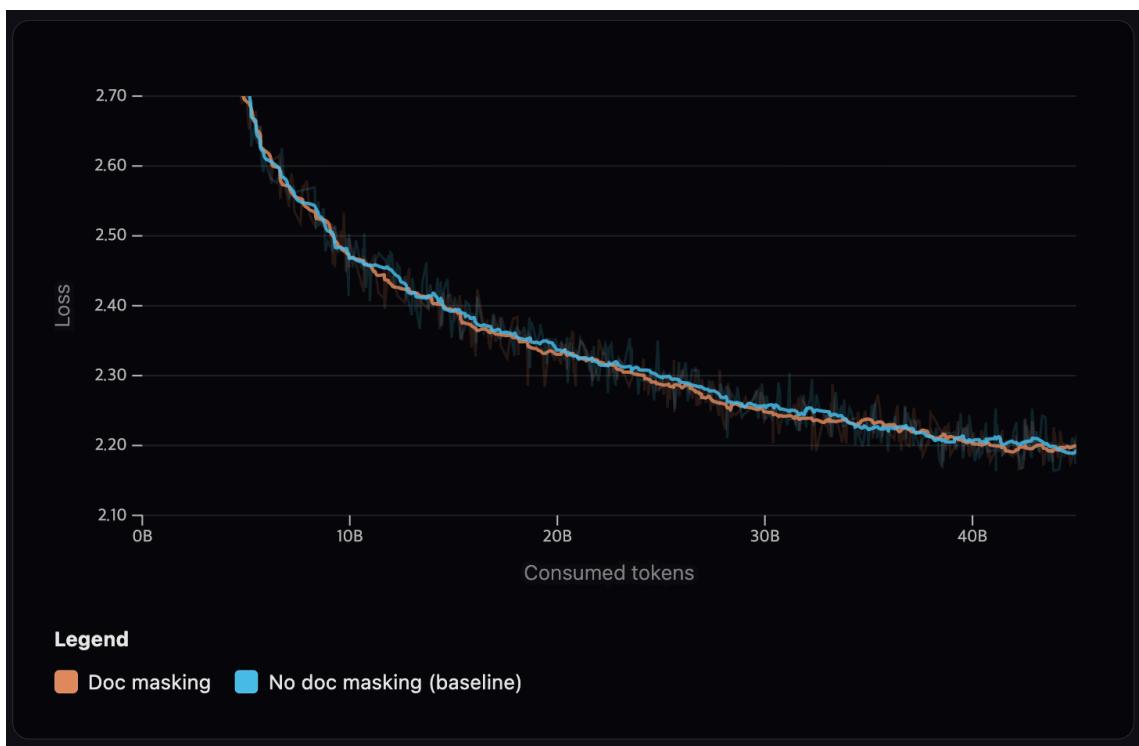
SkyLadder의 이 플롯들은 여러 발견을 보여줍니다. (a) 더 짧은 컨텍스트가 사전학습 중에 종종 더 나은 성능을 보입니다 (더 낮은 검증 퍼플렉시티). (b) 문서 내 마스킹(IntraDoc)은 랜덤 패킹(Random)과 의미적 그룹화(BM25) 모두보다 더 낮은 퍼플렉시티를 달성합니다. (c) 더 짧은 컨텍스트의 이점은 위치 인코딩 없이도 유지됩니다. (d) IntraDoc은 더 짧은 유효 컨텍스트 길이 쪽으로 치우친 분포를 생성합니다.

Llama3([Grattafiori et al., 2024](#))도 문서 내 마스킹으로 학습했으며, 짧은 컨텍스트 사전학습 중에는 제한적인 영향을 발견했지만 어텐션 오버헤드가 더 중요해지는 긴 컨텍스트 확장에서는 상당한 이점을 발견했습니다. 또한 ProLong 논문 ([Gao et al., 2025](#))은 지속적 사전학습에서 Llama3 8B의 컨텍스트를 확장하기 위해 문서 마스킹을 사용하면 긴 컨텍스트와 짧은 컨텍스트 벤치마크 모두에 이점이 있음을 보여주었습니다.

우리는 1B 기준 모델에서 절제 실험을 실행하여 문서 마스킹이 짧은 컨텍스트 성능에 영향을 미치는지 테스트하기로 결정했습니다. 설정은 [여기서](#) 찾을 수 있습니다. 결과는 아래 차트에서 보이는 것처럼 표준 인과적 마스킹과 비교하여 동일한 손실 곡선과 다운스트림 평가 점수를 보여주었습니다.

nanotron에서 문서 마스킹을 활성화하려면, 모델 설정에서 이 플래그를 `true`로 설정하면 됩니다.

```
model_config:
  _attn_implementation: flash_attention_2
  _fused_rms_norm: true
  _fused_rotary_emb: true
  _use_doc_masking: false
  _use_doc_masking: true
```



Llama3와 유사하게, PIQA에서의 작은 개선을 제외하고는 짧은 컨텍스트 테스크에서 눈에 띄는 영향을 관찰하지 못했습니다. 그러나 문서 마스킹은 학습 속도를 높이기 위해 긴 시퀀스로 스케일링할 때 매우 중요해집니다. 이는 특히 4k에서 64k 토큰으로 스케일링하는 긴 컨텍스트 확장에서 중요합니다(학습 마라톤 장에서 자세히 설명합니다). 따라서 SmoLLM3의 전체 학습 실행 전반에 걸쳐 이를 채택했습니다.

이 섹션에서 어텐션이 시퀀스를 어떻게 처리하는지 다루었습니다. 이제 트랜스포머의 또 다른 주요 파라미터 블록인 임베딩을 살펴보겠습니다.

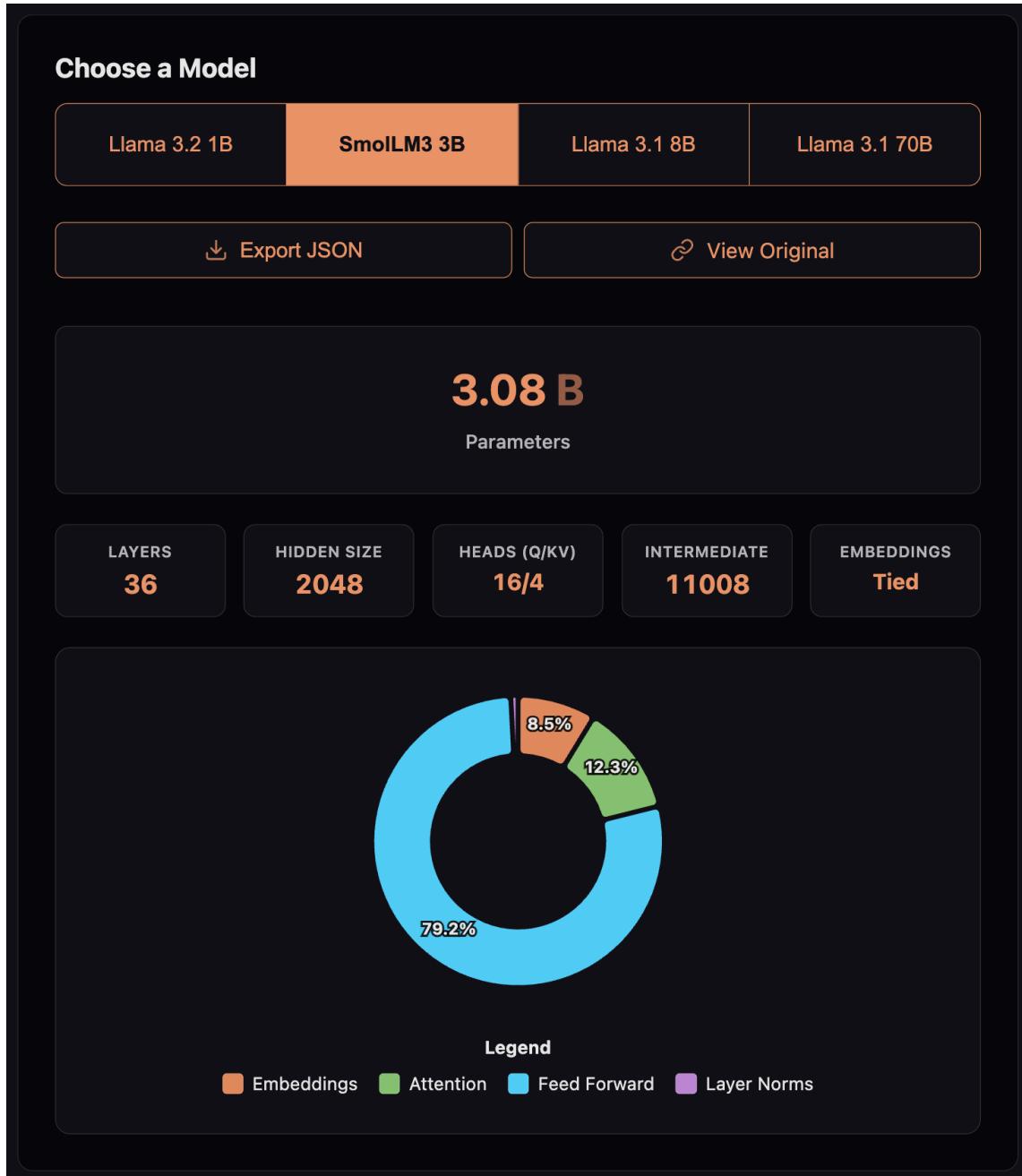
임베딩 공유

기준 절제 실험 모델의 [config](#)을 보면, 표준 트랜스포머와 다른 한 가지는 `tie_word_embeddings` 플래그로 활성화된 임베딩 공유입니다.

LLM에는 두 가지 임베딩 구성 요소가 있습니다. 토큰-벡터 룩업 테이블 역할을 하는 입력 임베딩(`vocab_size × hidden_dim` 크기)과 은닉 상태를 어휘로짓으로 매핑하는 최종 선형 레이어인 출력 임베딩(`hidden_dim × vocab_size`)입니다. 이들이 별도의 행렬인 고전적인 경우, 총 임베딩 파라미터는 $2 \times \text{vocab_size} \times \text{hidden_dim}$ 입니다. 따라서 소형 언어 모델에서 임베딩은 특히 어휘 크기가 큰 경우 전체 파라미터 수의 큰 부분을 차지할 수 있습니다. 이것이 임베딩 공유(출력에서 입력 임베딩을 재사용)가 소형 모델에 자연스러운 최적화가 되는 이유입니다.



더 큰 모델은 일반적으로 이 기술을 사용하지 않는데, 임베딩이 전체 파라미터 예산에서 더 작은 비율을 차지하기 때문입니다. 예를 들어, 아래 파이 차트에서 보이는 것처럼 공유 없이 총 임베딩은 Llama3.2 8B에서 13%에 불과하고 Llama3.1 70B에서는 3%에 불과합니다.

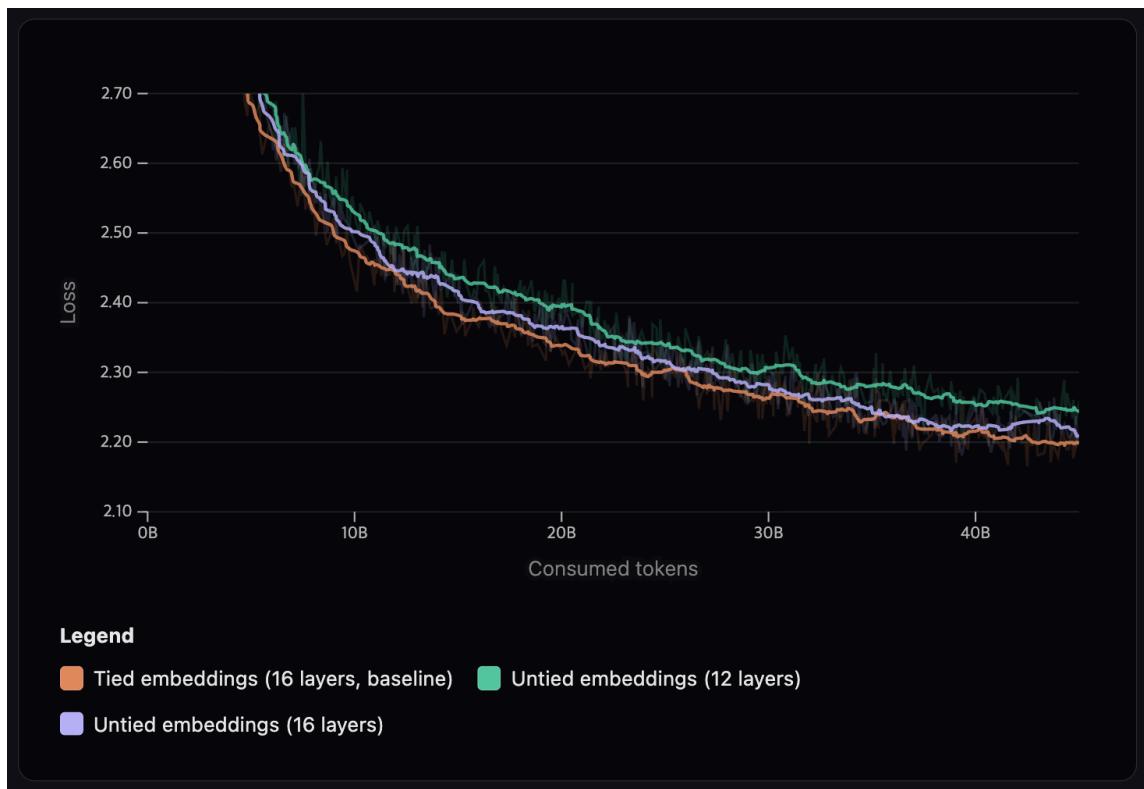


Ablation - 임베딩을 공유한 모델이 더 큰 비공유 변형과 동등한 성능을 보임

이제 절제 실험 모델에서 임베딩 공유의 영향을 평가할 것입니다. [MobileLLM's](#)의 125M 규모에서 이 기술에 대한 포괄적인 절제 실험에서 통찰을 얻었는데, 그들은 공유가 최소한의 정확도 저하로 파라미터를 11.8% 줄인다는 것을 보여주었습니다.

다.

임베딩을 공유하지 않으면 파라미터 수가 1.2B에서 1.46B로 증가하므로, 파라미터를 공유하지 않지만 레이어 수를 줄여 기준 1.2B 파라미터 수와 일치하는 또 다른 모델을 학습시킬 것입니다. 두 개의 1.2B 모델을 비교할 것입니다. 임베딩을 공유한 기준 모델(16개 레이어)과 동일한 파라미터 예산을 유지하기 위해 더 적은 레이어(12개 레이어)를 가진 비공유 버전, 그리고 추가 참조점으로 비공유 임베딩과 기준 모델과 동일한 레이어 수(16개)를 가진 1.46B 모델입니다. nanotron 설정은 [여기서](#) 찾을 수 있습니다.





손실과 평가 결과는 임베딩을 공유한 1.2B 기준 모델이 WinoGrande를 제외한 모든 벤치마크에서 18% 적은 파라미터에도 불구하고 1.46B 비공유 동등 모델과 비슷한 성능을 달성함을 보여줍니다. 임베딩을 공유하지 않고 레이어를 줄인(12개 vs 16개) 1.2B 모델은 두 구성 모두보다 낮은 성능을 보이며, 더 높은 손실과 더 낮은 다운스트림 평가 점수를 나타냅니다. 이는 동등한 파라미터 예산에서 임베딩 공유를 해제하는 것보다 모델 깊이를 늘리는 것이 더 큰 이점을 제공함을 시사합니다.

이러한 결과를 바탕으로, SmolLM3 3B 모델에서 임베딩 공유를 유지했습니다.

이제 임베딩 공유 전략과 그 트레이드오프를 살펴보았습니다. 그러나 임베딩만으로는 시퀀스에서 토큰의 순서를 포착하지 못합니다. 이 정보를 제공하는 것이 위치 인코딩의 역할입니다. 다음 섹션에서는 표준 RoPE에서 긴 컨텍스트를 위해 더 효과적인 모델링을 가능하게 하는 NoPE(No Position Embedding)와 같은 새로운 접근 방식까지, 위치 인코딩 전략이 어떻게 발전해 왔는지 살펴볼 것입니다.

위치 인코딩과 긴 컨텍스트

트랜스포머가 텍스트를 처리할 때, 근본적인 도전에 직면합니다. 병렬 어텐션 연산을 통해 전체 시퀀스를 동시에 소비하기 때문에 자연스럽게 단어 순서에 대한 감각이 없습니다. 이것은 효율적인 학습을 가능하게 하지만 문제를 만들어냅니다. 명시적인 위치 정보 없이는 모델의 관점에서 "Adam이 Muon을 이김"과 "Muon이 Adam을 이김"이 비슷하게 보입니다.

해결책은 위치 임베딩입니다. 시퀀스에서 각 토큰에 고유한 "주소"를 부여하는 수학적 인코딩입니다. 그러나 초기 BERT의 512 토큰에서 오늘날의 백만 토큰 모델까지 점점 더 긴 컨텍스트를 향해 나아감에 따라, 위치 인코딩의 선택은 성능과 계산 효율성 모두에 점점 더 중요해집니다.

위치 인코딩의 진화

초기 트랜스포머는 단순한 절대 위치 임베딩(APE, Absolute Position Embeddings)([Vaswani et al., 2023](#))을 사용했으며, 이는 본질적으로 각 위치(1, 2, 3...)를 토큰 임베딩에 더해지는 벡터로 매핑하는 학습된 룩업 테이블이었습니다. 이것은 짧은 시퀀스에서는 잘 작동했지만 주요 제한이 있었습니다. 모델의 최대 입력 시퀀스 길이가 학습된 최대 입력 시퀀스 길이로 제한되었습니다. 더 긴 시퀀스로의 기본 일반화 능력이 없었습니다.

이 분야는 절대 위치가 아닌 토큰 간의 거리를 포착하는 상대 위치 인코딩으로 발전했습니다. 이것은 직관적으로 이해가 됩니다. 두 단어가 3개 위치 떨어져 있다는 것이 그들이 위치 (5,8)에 있는지 (105,108)에 있는지보다 더 중요합니다.

위치 인코딩에 대한 더 깊은 이해를 위해, 이 블로그는 기본 위치 지정에서 회전 인코딩까지의 단계별 발전을 안내합니다.

특히 ALiBi(Attention with Linear Biases)([Press et al., 2022](#))는 토큰 거리를 기반으로 어텐션 점수를 수정합니다. 두 토큰이 멀리 떨어져 있을수록, 어텐션 가중치에 적용되는 단순한 선형 바이어스를 통해 어텐션이 더 많이 페널티를 받습니다. ALiBi의 자세한 구현은 이 [resource](#)를 확인하세요.

그러나 최근 대형 언어 모델을 지배한 기술은 Rotary Position Embedding(RoPE)([Su et al., 2023](#)).입니다.

RoPE: 회전으로서의 위치

RoPE의 핵심 통찰은 위치 정보를 고차원 공간에서 회전 각도로 인코딩하는 것입니다. 위치 벡터를 토큰 임베딩에 더하는 대신, RoPE는 쿼리와 키 벡터를 절대 위치에 따라 달라지는 각도로 회전시킵니다.

직관적으로, 임베딩의 각 차원 쌍을 원 위의 좌표로 취급하고 다음에 의해 결정되는 각도로 회전시킵니다.

- 시퀀스에서 토큰의 위치
- 작업 중인 차원 쌍(서로 다른 쌍은 기본/참조 주파수의 지수인 서로 다른 주파수로 회전)

```
import torch

def apply_rope_simplified(x, pos, dim=64, base=10000):
    """
    Rotary Position Embedding (RoPE)
    아이디어:
    - 각 토큰은 위치 인덱스 p (0, 1, 2, ...)를 가짐.
    - 각 벡터 차원 쌍은 인덱스 k (0 .. dim/2 - 1)를 가짐.
    - RoPE는 모든 쌍 [x[2k], x[2k+1]]을 각도 θ_{p,k}로 회전시킴.
    """

    # Compute theta for each token position and dimension
    theta = pos * (math.pi / base).repeat(dim // 2, 1)

    # Apply rotation to the corresponding dimensions
    x[:, :, ::2] = x[:, :, ::2].cos(theta)
    x[:, :, 1::2] = x[:, :, 1::2].sin(theta)

    return x
```

공식:

$$\theta_{p,k} = p * \text{base}^{(-k / (\text{dim}/2))}$$

- 작은 k (초기 차원 쌍) → 느린 진동 → 장거리 정보 포착.

- 큰 k (후기 차원 쌍) → 빠른 진동 → 세밀한 디테일 포착.

```
....  
rotated = []  
for i in range(0, dim, 2):  
    k = i // 2 # 이 차원 쌍의 인덱스  
  
    # 주파수 항: 더 높은 k → 더 빠른 진동  
    inv_freq = 1.0 / (base ** (k / (dim // 2)))  
    theta = pos * inv_freq # 위치 p와 쌍 k에 대한 회전 각도  
  
    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))  
  
    x1, x2 = x[i], x[i+1]  
  
    # 2D 회전 적용  
    rotated.extend([x1 * cos_t - x2 * sin_t,  
                   x1 * sin_t + x2 * cos_t])  
  
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)  
  
## 👉 내적 *전에* Q와 K에 RoPE 적용  
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))])  
  
scores = (Q_rope @ K_rope.T) / math.sqrt(Q.size(-1))  
attn_weights = torch.softmax(scores, dim=-1)
```

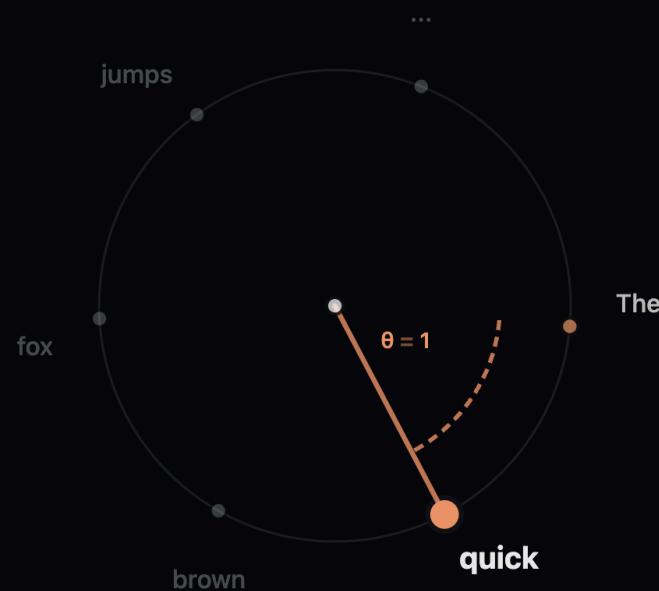
이 코드가 복잡해 보일 수 있으니 구체적인 예시로 분해해 보겠습니다. "The quick brown fox" 문장에서 "fox"라는 단어를 생각해 보세요. 1B 기준 모델에서 각 어텐션 헤드는 64차원 퀘리/키 벡터로 작업합니다. RoPE는 이 벡터를 32개 쌍으로 그룹화합니다: $(x_1, x_2), (x_3, x_4), (x_5, x_6)$ 등. 2D 공간에서 원 주위를 회전하기 때문에 쌍으로 작업합니다. 단순화를 위해 첫 번째 쌍 (x_1, x_2) 에 집중해 보겠습니다. "fox"라는 단어는 문장에서 위치 3에 나타나므로, RoPE는 이 첫 번째 차원 쌍을 다음과 같이 회전시킵니다.

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

기본 주파수는 10000이지만 첫 번째 차원 쌍($k=0$)의 경우 지수가 0이므로 기본 주파수가 계산에 영향을 미치지 않습니다 (0의 거듭제곱을 취함). 아래 시각화는 이를 설명합니다.

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

The quick brown fox jumps ...



quick at position 1 gets rotated by

$$\theta = 1 \text{ rad} (57^\circ)$$

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

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

이제 두 토큰이 어텐션을 통해 상호작용할 때 마법이 일어납니다. 회전된 표현 간의 내적은 회전 각도 간의 위상 차이를 통해 상대적 거리를 직접 인코딩합니다(여기서 m 과 n 은 토큰 위치입니다).

$$\text{dot_product}(\text{RoPE}(x, m), \text{RoPE}(y, n)) = \sum_k [x_k * y_k * \cos((m-n) * \theta_k)]$$

어텐션 패턴은 $(m-n)$ 에만 의존하므로, 5개 위치 떨어진 토큰은 시퀀스에서의 절대 위치에 관계없이 항상 동일한 각도 관계를 갖습니다. 따라서 모델은 시퀀스의 어떤 절대 위치에서도 작동하고 더 긴 시퀀스로 외삽할 수 있는 거리 기반 패턴을 학습합니다.

RoPE 주파수를 어떻게 설정할까요?

실제로 대부분의 LLM 사전학습은 10K 또는 50K와 같은 수만의 RoPE 기본 주파수를 사용하여 상대적으로 짧은 컨텍스트 길이(2K-4K 토큰)로 시작합니다. 처음부터 매우 긴 시퀀스로 학습하는 것은 어텐션의 시퀀스 길이에 대한 이차적 스케일링과 어텐션의 문서 마스킹 섹션에서 이전에 본 것처럼 긴 컨텍스트 데이터(4K 컨텍스트 길이 이상의 샘플)의 제한된 가용성으로 인해 계산 비용이 많이 듭니다. 연구에 따르면 이것이 짧은 컨텍스트 성능을 저하시킬 수도 있습니다([Zhu et al., 2025](#)). 모델은 일반적으로 단어 간의 단거리 상관관계를 먼저 학습하므로 긴 시퀀스가 크게 도움이 되지 않습니다. 일반적인 접근 방식은 대부분의 사전학습을 짧은 시퀀스로 수행한 다음, 지속적 사전학습을 하거나 마지막 수천억 토큰을 더 긴 시퀀스에 사용하는 것입니다. 그러나 시퀀스 길이가 늘어남에 따라, 토큰 위치에 비례하는 회전 각도가 커지고 먼 토큰에 대한 어텐션 점수가 너무 빠르게 감쇠할 수 있습니다([Rozière et al., 2024; Xiong et al., 2023](#)).

$$\theta = \text{position} \times 1 / (\text{base}^{(k/(dim/2))})$$

해결책은 ABF와 YaRN과 같은 방법을 사용하여 이러한 감쇠를 방지하기 위해 시퀀스 길이가 증가할 때 기본 주파수를 높이는 것입니다.

RoPE ABF (조정된 기본 주파수를 가진 RoPE) ([Xiong et al., 2023b](#))는 RoPE 공식에서 기본 주파수를 높여 긴 컨텍스트에서의 어텐션 감쇠 문제를 해결합니다. 이 조정은 토큰 위치 간의 회전 각도를 느리게 하여 먼 토큰의 어텐션 점수가 너무 빠르게 감쇠하는 것을 방지합니다. ABF는 단일 단계(직접 주파수 증가) 또는 다단계(컨텍스트가 커짐에 따라 점진적 증가)로 적용할 수 있습니다. 이 방법은 구현이 간단하고 임베딩 벡터를 증가된 세분성으로 분배하여 모델이 먼 위치를 더 쉽게 구별할 수 있게 합니다. 단순하고 효과적이지만, ABF의 모든 차원에 대한 균일한 스케일링은 극도로 긴 컨텍스트에는 최적이 아닐 수 있습니다.

YaRN (Yet another RoPE extensioN) ([Peng et al., 2023](#))은 램프 또는 스케일링 함수를 사용하여 RoPE 차원에 걸쳐 주파수를 불균일하게 보간하는 더 정교한 접근 방식을 취합니다. ABF의 균일한 조정과 달리, YaRN은 다른 주파수 구성 요소에 다른 스케일링 계수를 적용하여 확장된 컨텍스트 윈도우를 최적화합니다. 여기에는 등적 어텐션 스케일링과 어텐션 로짓의 온도 조정과 같은 추가 기술이 포함되어 매우 큰 컨텍스트 크기에서 성능을 유지하는 데 도움이 됩니다. YaRN은 효율적인 "짧게 학습하고, 길게 테스트" 전략을 가능하게 하며, 강력한 외삽을 위해 더 적은 토큰과 더 적은 파인튜닝이 필요합니다. ABF보다 복잡하지만, YaRN은 일반적으로 더 부드러운 스케일링을 제공하고 치명적인 어텐션 손실을 완화하여 극도로 긴 컨텍스트에서 더 나은 경험적 성능을 제공합니다. 또한 파인튜닝 없이 추론에서만 활용할 수도 있습니다.

이러한 주파수 조정 방법은 어텐션 점수 감쇠 효과를 늦추고 먼 토큰의 기여를 유지합니다. 예를 들어, Qwen3의 학습에서는 시퀀스 길이가 4k에서 32k 컨텍스트로 확장됨에 따라 ABF를 사용하여 주파수를 10k에서 1M으로 높였습니다(팀은 이후 131k, 4배 외삽에 도달하기 위해 YaRN을 적용합니다). 최적의 값에 대한 강한 합의는 없으며, 특정 설정과 평가 벤치마크에 가장 적합한 것을 찾기 위해 컨텍스트 확장 단계에서 다양한 RoPE 값을 실험하는 것이 보통 좋습니다.

오늘날 대부분의 주요 모델은 RoPE를 사용합니다. Llama, Qwen, Gemma 등이 있습니다. 이 기술은 다양한 모델 크기와 아키텍처(밀집, MoE, 하이브리드)에서 견고함이 입증되었습니다. 최근 등장한 몇 가지 RoPE 변형을 살펴보겠습니다.

하이브리드 위치 인코딩 접근 방식

그러나 모델이 점점 더 큰 컨텍스트를 향해 나아감에 따라([Meta AI, 2025; Yang et al., 2025](#)), RoPE조차도 성능 문제에 부딪히기 시작했습니다. 긴 컨텍스트 확장 중에 RoPE의 주파수를 높이는 표준 접근 방식은 Needle in the Haystack(NIAH)([Kamradt, 2023](#))보다 더 어려운 Ruler와 HELMET([Hsieh et al., 2024; Yen et al., 2025](#))과 같은 긴 컨텍스트 벤치마크에서 평가할 때 한계가 있습니다. 도움이 되는 새로운 기술이 도입되었습니다.

이 섹션을 시작할 때 트랜스포머가 토큰 순서를 이해하려면 위치 정보가 필요하다고 말했지만, 최근 연구는 이 가정에 도전했습니다. 명시적인 위치 인코딩이 결국 필요하지 않다면 어떨까요?

NoPE (No Position Embedding)([Kazemnejad et al., 2023](#))는 명시적인 위치 인코딩 없이 트랜스포머를 학습시켜, 모델이 인과적 마스킹과 어텐션 패턴을 통해 위치 정보를 암묵적으로 학습할 수 있게 합니다. 저자들은 이 접근 방식이 ALiBi와 RoPE에 비해 더 나은 길이 일반화를 보여준다고 밝힙니다. 학습 길이를 넘어 외삽할 명시적 위치 인코딩이 없기 때문에, NoPE는 자연스럽게 더 긴 컨텍스트를 처리합니다. 그러나 실제로 NoPE 모델은 RoPE에 비해 짧은 컨텍스트 추론 및 지식 태스크에서 더 약한 성능을 보이는 경향이 있습니다([Yang et al.](#)). 이는 명시적 위치 인코딩이 외삽을 제한할 수 있지만, 학습 컨텍스트 길이 내의 태스크에 유용한 귀납적 편향을 제공함을 시사합니다.

RNoPE 하이브리드 접근 방식: 이러한 트레이드오프를 고려하여, [B. Yang et al. \(2025\)](#)은 다른 위치 인코딩 전략을 결합하는 것이 흥미로울 수 있다고 제안합니다. 그들은 모델 전체에서 RoPE와 NoPE 레이어를 번갈아 사용하는 RNoPE를 도입합니다. RoPE 레이어는 명시적 위치 정보를 제공하고 최근성 편향으로 로컬 컨텍스트를 처리하는 반면, NoPE 레이어는 긴 거리에 걸친 정보 검색을 개선합니다. 이 기술은 최근 Llama4, Command A, SmolLM3에서 사용되었습니다.

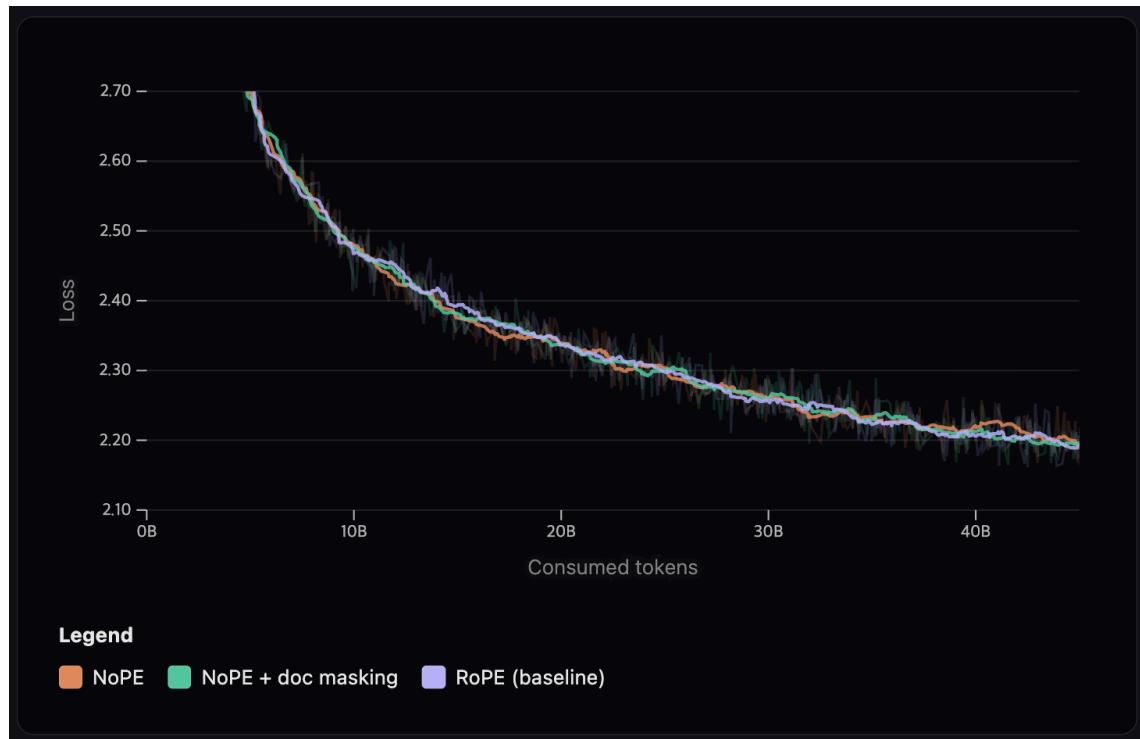
💡 명명 규칙

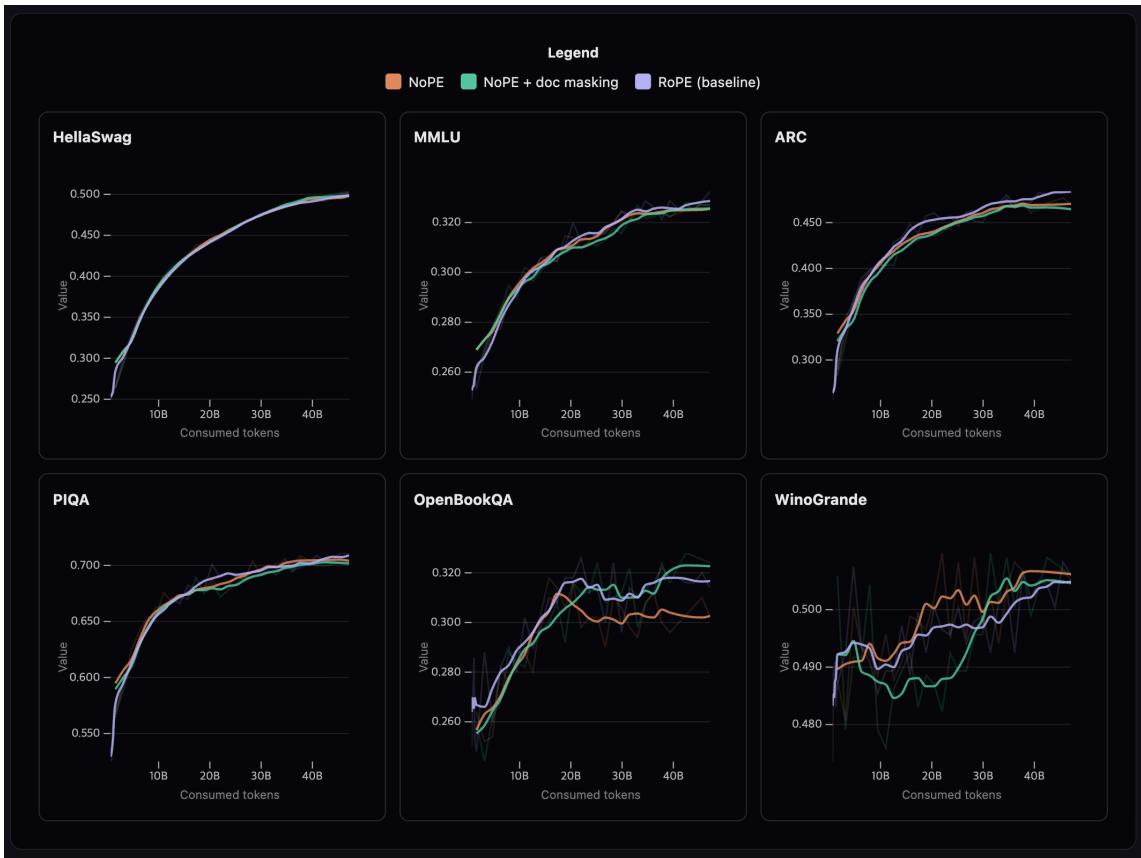
이 블로그의 나머지 부분에서는 간단하게 하기 위해 RNoPE를 "NoPE"라고 부를 것입니다. (토론에서 "NoPE"를 RNoPE의 의미로 사용하는 것을 종종 볼 수 있습니다.)

Ablation - NoPE가 짧은 컨텍스트에서 RoPE와 동등한 성능을 보임

하이브리드 NoPE 접근 방식을 테스트해 보겠습니다. 순수 RoPE 1B 절제 실험 기준 모델과 4번째 레이어마다 위치 인코딩을 제거하는 NoPE 변형, 그리고 이러한 기술 간의 상호작용을 테스트하기 위해 NoPE와 문서 마스킹을 결합한 세 번째 구성을 비교할 것입니다. 우리의 기본 질문은 긴 컨텍스트 능력을 얻으면서 강력한 짧은 컨텍스트 성능을 유지할 수 있는가입니다.

니다.





손실과 평가 결과는 세 가지 구성 모두에서 유사한 성능을 보여주며, 이는 NoPE가 더 나은 긴 컨텍스트 처리를 위한 기반을 제공하면서 강력한 짧은 컨텍스트 능력을 유지함을 나타냅니다. 이러한 결과를 바탕으로, SmolLM3에 NoPE + 문서 마스킹 조합을 채택했습니다.

Partial/Fractional RoPE: 또 다른 보완적인 아이디어는 모델 차원의 일부에만 RoPE를 적용하는 것입니다. RoPE와 NoPE 사이에서 전체 레이어를 번갈아 사용하는 RNoPE와 달리, Partial RoPE는 동일한 레이어 내에서 이들을 혼합합니다. GLM-4.5([5 Team et al., 2025](#))과 Minimax-01([MiniMax et al., 2025](#))과 같은 최근 모델들이 이 전략을 채택했지만, 이는 gpt-j([Wang & Komatsuzaki, 2021](#))와 같은 이전 모델에서도 존재했습니다. MLA를 사용하는 모든 모델에서도 이를 볼 수 있는데, 합리적인 추론 비용을 위해 필수이기 때문입니다.

🔧 기술적 설명: Partial RoPE가 MLA에 필수적인 이유

MLA는 프로젝션 흡수를 통해 추론을 효율적으로 만듭니다. 헤드별 키 $k_i^{(h)}$ 를 저장하는 대신, 작은 공유 잠재 변수 $c_i = \mathbf{x}_i \mathbf{W}_c \in \mathbb{R}^{d_c}$ 를 캐시하고 헤드의 쿼리/키 맵을 병합하여 각 점수를 저렴하게 계산합니다. $q_t^{(h)} = \mathbf{x}_t \mathbf{W}_q^{(h)}$ 와 $k_i^{(h)} = c_i \mathbf{E}^{(h)}$ 로, $\mathbf{U}^{(h)} = \mathbf{W}_q^{(h)} \mathbf{E}^{(h)}$ 를 정의하면 다음을 얻습니다.

$$s_{t,i}^{(h)} = \frac{1}{\sqrt{d_k}} (q_t^{(h)})^\top k_i^{(h)} = \frac{1}{\sqrt{d_k}} (\mathbf{x}_t \mathbf{U}^{(h)})^\top c_i$$

따라서 작은 캐시 c_i 에 대해 $\tilde{\mathbf{q}}_t^{(h)} = \mathbf{x}_t \mathbf{U}^{(h)} \in \mathbb{R}^{d_c}$ 로 계산합니다(헤드별 k 가 저장되지 않음). RoPE는 두 맵 사이에 상의존적 회전을 삽입하기 때문에 이를 깨뜨립니다. 전체 차원 RoPE에서,

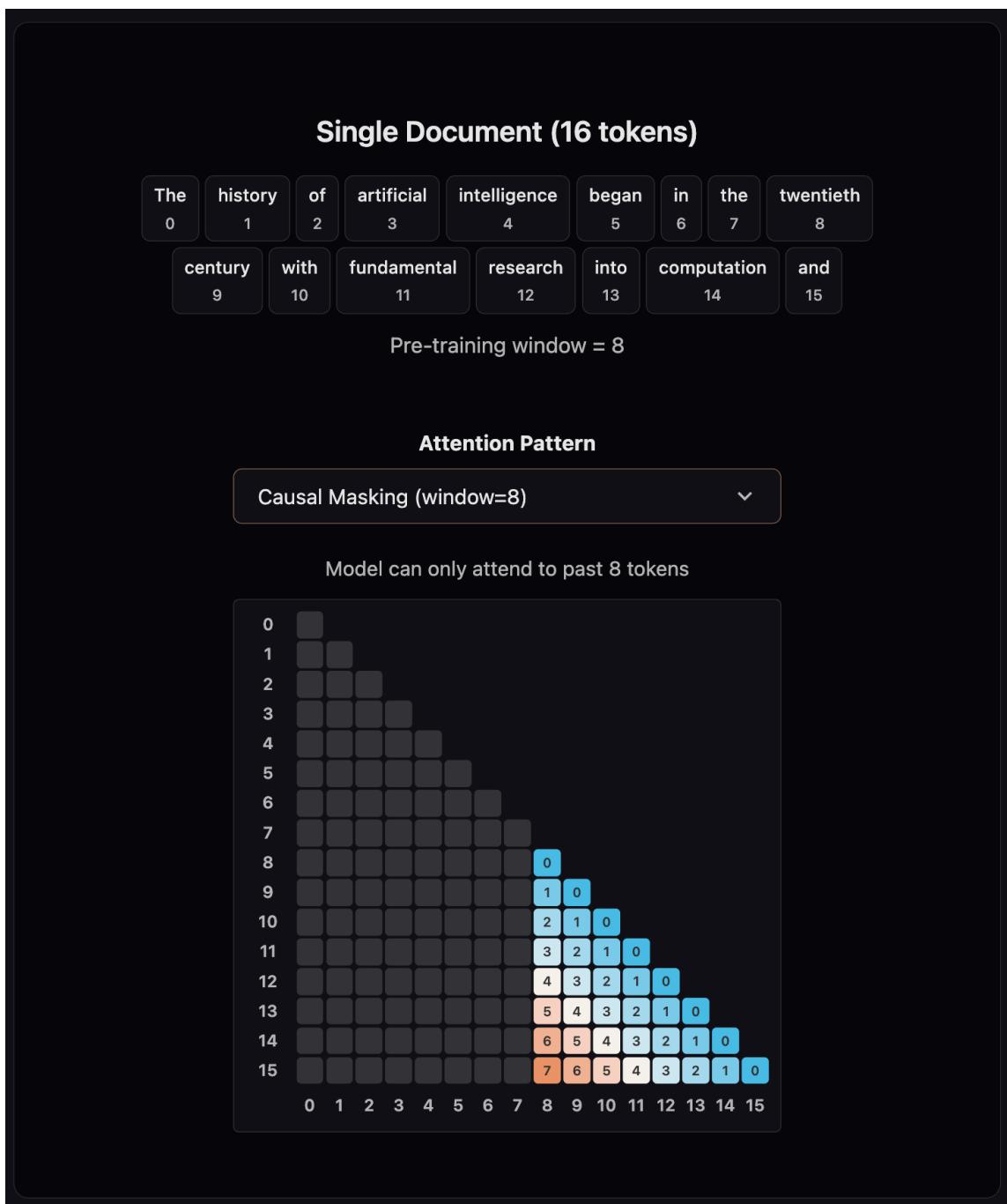
$$s_{t,i}^{(h)} = \frac{1}{\sqrt{d_k}} (x_t W_q^{(h)})^\top \underbrace{R_{t-i}}_{\text{depends on } t-i} (c_i E^{(h)})$$

따라서 $W_q^{(h)}$ 와 $E^{(h)}$ 를 고정된 $U^{(h)}$ 로 미리 병합할 수 없습니다. 해결책: partial RoPE. 헤드 차원을 $d_k = d_{\text{nope}} + d_{\text{rope}}$ 로 분할하고, 큰 블록에는 회전을 적용하지 않고(이전처럼 흡수: $(x_t U_{\text{nope}}^{(h)})^\top c_i$), 작은 블록에만 RoPE를 적용합니다.

긴 컨텍스트를 위한 어텐션 범위 제한

지금까지 긴 컨텍스트를 위한 위치 정보 처리 방법을 탐구했습니다. RoPE 활성화, 비활성화(NoPE), 일부 레이어(RNoPE)나 일부 은닉 차원(Partial RoPE)에 부분적으로 적용, 또는 주파수 조정(ABF, YaRN) 등입니다. 이러한 접근 방식은 모델이 학습 중에 본 것보다 긴 시퀀스를 처리하기 위해 위치를 인코딩하는 방법을 수정합니다. 그러나 보완적인 전략이 있습니다. 위치 인코딩을 조정하는 대신, 어떤 토큰이 서로 어텐션을 주는지 제한할 수 있습니다.

이것이 왜 중요한지 보기 위해, 8개 토큰 시퀀스로 사전학습된 모델을 생각해 보세요. 추론 시에 16개 토큰(학습 길이보다 많음)을 처리하고 싶습니다. 위치 8-15는 모델의 위치 인코딩에 대해 분포 외(out of distribution)입니다. RoPE ABF와 같은 기술이 위치 주파수를 조정하여 이를 해결하는 반면, 어텐션 범위 방법은 다른 접근 방식을 취합니다. 전체 시퀀스를 처리하면서도 어텐션 패턴을 익숙한 범위 내에 유지하기 위해 어떤 토큰이 서로 어텐션을 줄 수 있는지 전략적으로 제한합니다. 이는 계산 비용과 메모리 요구 사항을 모두 줄입니다. 아래 다이어그램은 8의 사전학습 윈도우로 16개 토큰 시퀀스를 처리하기 위한 다섯 가지 전략을 비교합니다.



첨크 어텐션(Chunked Attention)은 시퀀스를 고정 크기 청크로 나누며, 토큰은 자신의 청크 내에서만 어텐션을 줄 수 있습니다. 우리 예시에서 16개 토큰은 두 개의 8 토큰 청크(0-7과 8-15)로 분할되며, 각 토큰은 자신의 청크 내의 다른 토큰만 볼 수 있습니다. 토큰 8-15가 이전 청크에 전혀 어텐션을 줄 수 없다는 점에 주목하세요. 이는 청크 경계에서 재설정되는 격리된 어텐션 윈도우를 생성합니다. Llama 4([Meta AI, 2025](#))는 RoPE 레이어(4개 디코더 레이어 중 3개)에서 8192 토큰 청크를 가진 청크 어텐션을 사용하고, NoPE 레이어는 전체 컨텍스트 접근을 유지합니다. 이는 레이어당 KV 캐시 크기

를 제한하여 메모리 요구 사항을 줄이지만, 토큰이 이전 청크에 어텐션을 줄 수 없어 일부 긴 컨텍스트 태스크에 영향을 미칠 수 있습니다.

슬라이딩 윈도우 어텐션(SWA, Sliding Window Attention)은 Mistral 7B([Child et al., 2019; Jiang et al., 2023](#))에 의해 대중화되었으며, 최근 토큰이 가장 관련성이 높다는 직관을 기반으로 다른 접근 방식을 취합니다. 딱딱한 청크 경계 대신, 각 토큰은 가장 최근 N개 토큰에만 어텐션을 줍니다. 다이어그램에서 모든 토큰은 최대 8개 위치 뒤까지 볼 수 있어, 시퀀스를 통해 연속적으로 이동하는 슬라이딩 윈도우를 생성합니다. 토큰 15가 위치 8-15에 어텐션을 줄 수 있는 반면, 토큰 10은 위치 3-10에 어텐션을 준다는 점에 주목하세요. 윈도우가 앞으로 슬라이딩하여 청킹의 인위적인 장벽 없이 전체 시퀀스에 걸쳐 로컬 컨텍스트를 유지합니다. Gemma 3는 하이브리드 위치 인코딩 접근 방식이 다른 전략을 혼합하는 것과 유사하게, 교대 레이어에서 SWA와 전체 어텐션을 결합합니다.

듀얼 청크 어텐션(DCA, Dual Chunk Attention) ([An et al., 2024](#))은 청크 간 정보 흐름을 유지하면서 청크 어텐션을 확장하는 학습이 필요 없는 방법입니다. 우리 예시에서는 청크 크기 $s=4$ 를 사용하여 16개 토큰을 4개 청크로 나눕니다(대각선을 따라 4×4 정사각형을 시각화하세요). DCA는 세 가지 메커니즘을 결합합니다. (1) 토큰이 청크 내에서 정상적으로 어텐션을 주는 **청크 내 어텐션**(대각선 패턴). (2) 쿼리가 위치 인덱스 $c-1=7$ 을 사용하여 이전 청크에 어텐션을 주어 7로 제한된 상대 위치를 생성하는 **청크 간 어텐션**. (3) 이웃 청크 간의 지역성을 보존하는 로컬 윈도우 $w=3$ 을 가진 **연속 청크 어텐션**. 이는 청크 경계에서 부드러운 전환을 유지하면서 모든 상대 위치를 학습 분포(0-7) 내에 유지합니다. DCA는 Qwen2.5와 같은 모델이 백만 토큰 시퀀스에 대한 지속적 학습 없이도 추론 시 최대 100만 토큰의 초장문 컨텍스트 윈도우를 지원할 수 있게 합니다.

어텐션 싱크(Attention Sinks)

긴 컨텍스트를 가진 트랜스포머 모델에서 흥미로운 현상이 나타납니다. 모델이 시퀀스의 초기 토큰에 비정상적으로 높은 어텐션 점수를 부여하는데, 이 토큰들이 의미적으로 중요하지 않더라도 그렇습니다. 이 동작을 **어텐션 싱크**([Xiao et al.](#))라고 합니다. 이러한 초기 토큰은 어텐션 분포의 안정화 메커니즘 역할을 하며, 어텐션이 축적될 수 있는 "싱크" 역할을 합니다.

실용적인 통찰은 컨텍스트가 캐시 크기를 초과할 때 최근 토큰의 슬라이딩 윈도우와 함께 처음 몇 개 토큰의 KV 캐시만 유지해도 성능을 크게 회복한다는 것입니다. 이 간단한 수정으로 모델이 파인튜닝이나 성능 저하 없이 훨씬 더 긴 시퀀스를 처리할 수 있습니다.

현대 구현은 다양한 방식으로 어텐션 싱크를 활용합니다. 원래 연구에서는 명시적인 어텐션 싱크 역할을 하도록 사전학습 중에 전용 플레이스홀더 토큰을 추가할 것을 제안합니다. 더 최근에는 gpt-oss와 같은 모델이 입력 시퀀스의 실제 토큰이 아닌 어텐션 점수에 추가되는 학습된 헤드별 바이어스 로짓으로 어텐션 싱크를 구현합니다. 이 접근 방식은 토큰화된 입력을 수정하지 않고도 동일한 안정화 효과를 달성합니다.

흥미롭게도 gpt-oss는 어텐션 레이어 자체에도 바이어스 유닛을 사용하는데, 이는 GPT-2 이후 거의 볼 수 없는 설계 선택입니다. 이러한 바이어스 유닛은 일반적으로 표준 어텐션 연산에 불필요하다고 여겨지지만 [Dehghani et al.](#), 경험적 결과는 테스트 손실에 미치는 영향이 미미함을 보여주었고, 어텐션 싱크를 구현하는 특수한 기능을 수행할 수 있습니다.

핵심 통찰: 특수 토큰, 학습된 바이어스, 또는 헤드별 로짓으로 구현되든, 어텐션 싱크는 긴 컨텍스트 시나리오에서 어텐션 분포에 안정적인 "앵커"를 제공하여, 컨텍스트가 임의로 길어져도 모델이 전체 시퀀스에 대한 일반적으로 유용한 정보를 저장할 수 있게 합니다.

이제 어텐션의 핵심 구성 요소를 다루었습니다. 메모리와 컴퓨팅의 균형을 맞추는 다양한 헤드 구성(MHA, GQA, MLA), 모델이 토큰 순서를 이해하는 데 도움이 되는 위치 인코딩 전략(RoPE, NoPE 및 그 변형), 긴 컨텍스트를 다루기 쉽게 만드는 어텐션 범위 기술(슬라이딩 윈도우, 청킹, 어텐션 싱크) 등입니다. 또한 임베딩 레이어가 어떻게 구성되고 초기화되어야 하는지도 살펴보았습니다. 이러한 아키텍처 선택은 모델이 시퀀스를 처리하고 표현하는 방법을 정의합니다.

그러나 올바른 아키텍처를 갖추는 것은 절반에 불과합니다. 잘 설계된 모델도 특히 대규모에서 학습 불안정으로 어려움을 겪을 수 있습니다. 학습을 안정적으로 유지하는 데 도움이 되는 기술을 살펴보겠습니다.

안정성 향상

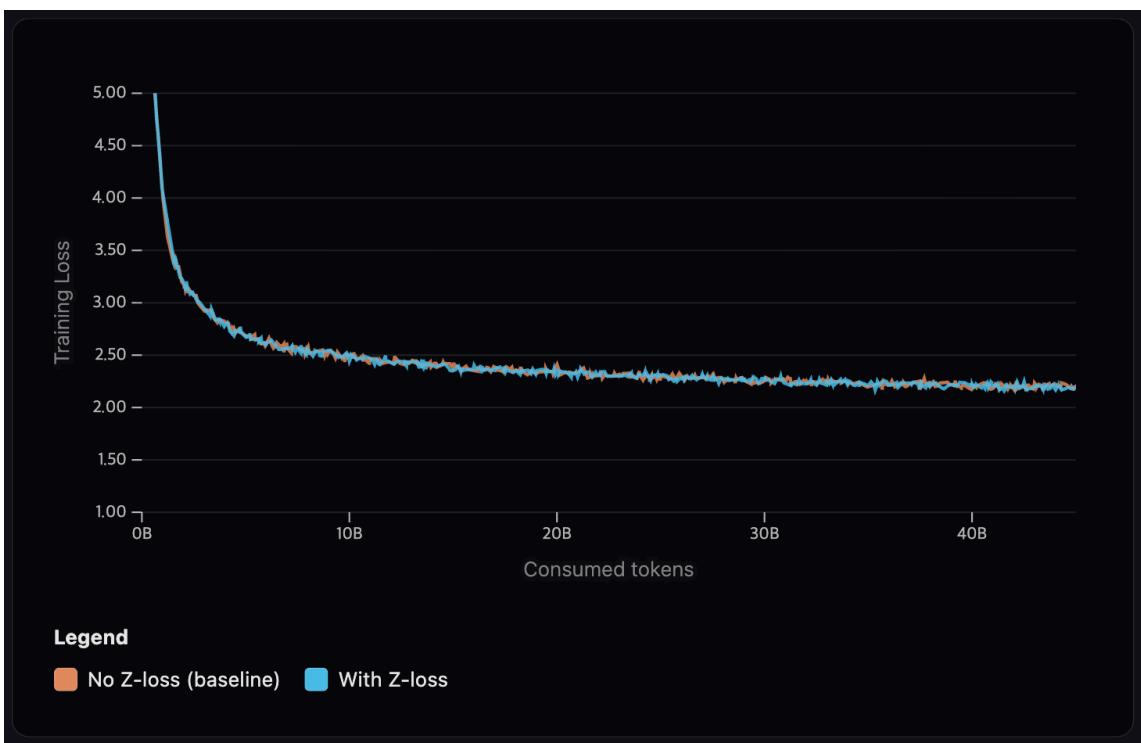
이제 LLM 사전학습에서 가장 큰 도전 중 하나인 불안정성으로 넘어가겠습니다. 종종 손실 스파이크나 학습 손실의 갑작스러운 점프로 나타나는 이러한 문제는 특히 대규모에서 흔해집니다.

[Training Marathon](#) 섹션에서 다양한 유형의 스파이크와 처리 방법에 대해 더 깊이 다를 것이지만(부동 소수점 정밀도, 옵티마이저, 학습률 등), 특정 아키텍처 및 학습 기술도 불안정성을 줄이는 데 도움이 될 수 있으므로 여기서 잠시 살펴보겠습니다. 최근 대규모 학습 실행(예: Olmo2([OlMo et al., 2025](#)) 및 Qwen3([A. Yang, Li, et al., 2025](#)))에서 안정성을 개선하기 위해 사용된 몇 가지 간단한 기술을 다룰 것입니다. Z-loss, 임베딩에서 가중치 감소 제거, QK-norm입니다.

Z-loss

Z-loss([Chowdhery et al., 2022](#))는 손실 함수에 페널티 항을 추가하여 최종 출력 로짓이 너무 커지는 것을 방지하는 정규화 기술입니다. 이 정규화는 로짓에 대한 소프트맥스의 분모가 합리적인 범위 내에 머무르도록 장려하여 학습 중 수치적 안정성을 유지하는 데 도움이 됩니다.

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



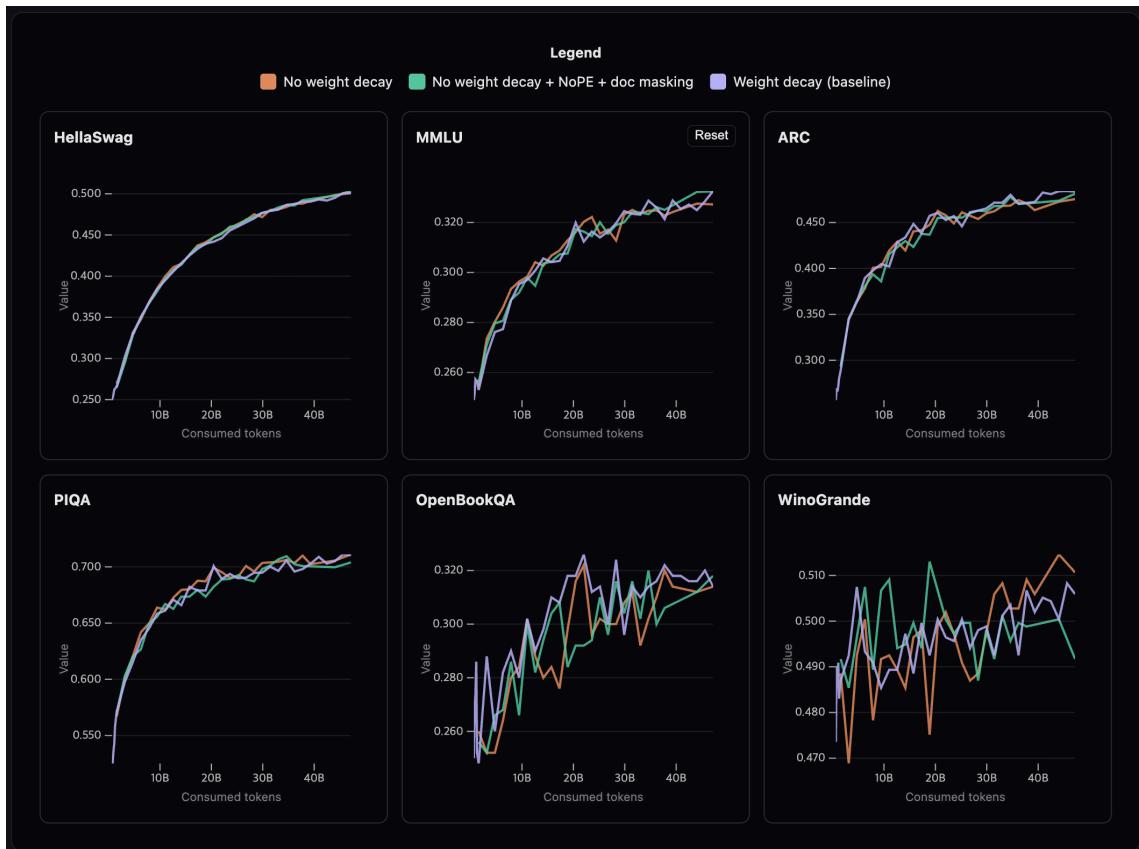
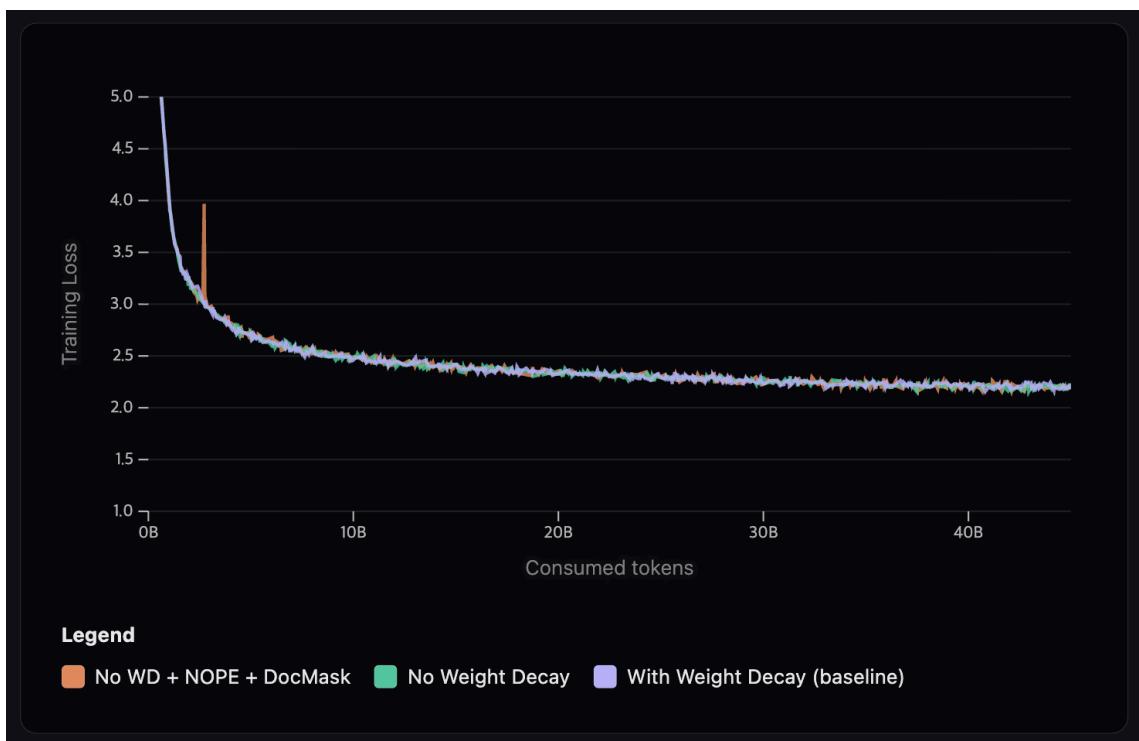


아래 1B 모델에 대한 절제 실험 결과는 Z-loss를 추가해도 학습 손실이나 다운스트림 성능에 영향을 미치지 않음을 보여줍니다. SmolLM3의 경우, 학습을 시작할 때까지 최적화하지 못한 Z-loss 구현이 일부 학습 오버헤드를 도입했기 때문에 결국 사용하지 않았습니다.

임베딩에서 가중치 감쇠 제거

가중치 감쇠(weight decay)는 정규화 기술로서 모든 모델 파라미터에 일반적으로 적용되지만, [OLMo et al. \(2025\)](#)은 임베딩을 가중치 감쇠에서 제외하면 학습 안정성이 향상된다는 것을 발견했습니다. 그 이유는 가중치 감쇠가 학습 중에 임베딩 노름을 점진적으로 감소시키고, 레이어 정규화의 야코비안이 입력 노름에 반비례하기 때문에 초기 레이어에서 더 큰 그래디언트로 이어질 수 있기 때문입니다([Takase et al., 2025](#)).

우리는 세 가지 구성을 학습시켜 이 접근 방식을 테스트했습니다. 표준 가중치 감쇠를 사용한 기준 모델, 임베딩에 가중치 감쇠가 없는 변형, 그리고 기술 간의 부정적인 상호작용이 없는지 확인하기 위해 채택한 모든 변경 사항을 결합한 세 번째 구성(임베딩에 가중치 감쇠 없음 + NoPE + 문서 마스킹)입니다. 손실 곡선과 평가 결과는 세 가지 구성 모두에서 거의 동일했습니다. 따라서 SmolLM3 학습에서 3가지 변경 사항을 모두 채택했습니다.



QK-norm

QK-norm([Dehghani et al., 2023](#))은 어텐션을 계산하기 전에 쿼리와 키 벡터 모두에 레이어 정규화를 적용합니다. 이 기술은 어텐션 로짓이 너무 커지는 것을 방지하는 데 도움이 되며, 안정성을 개선하기 위해 많은 최근 모델에서 사용되었습니다.

그러나 [B. Yang et al. \(2025\)](#)은 QK-norm이 긴 컨텍스트 태스크에 해롭다는 것을 발견했습니다. 그들의 분석에 따르면 QK-norm은 관련 토큰(니들)에 대한 어텐션 질량을 낮추고 관련 없는 컨텍스트에 대한 어텐션 질량을 높입니다. 그들은 정규화 연산이 쿼리-키 내적에서 크기 정보를 제거하여 어텐션 로짓이 크기 면에서 더 가까워지기 때문에 이런 현상이 발생한다고 주장합니다. 이러한 이유로 SmoLLM3에서는 QK-norm을 사용하지 않았습니다. 또한 작은 3B 파라미터 모델로서, QK-norm이 가장 유익한 것으로 입증된 더 큰 모델에 비해 학습 불안정의 위험이 적습니다.

기타 핵심 구성 요소

우리가 다른 구성 요소 외에도, 완전성을 위해 언급할 가치가 있는 몇 가지 다른 아키텍처 결정이 있습니다.

파라미터를 초기화하기 위해, 현대 모델은 일반적으로 절단 정규 초기화(mean=0, std=0.02 또는 std=0.006) 또는 muP([G. Yang & Hu, 2022](#))와 같은 초기화 방식을 사용합니다. 예를 들어 Cohere의 Command A([Cohere et al., 2025](#))가 있습니다. 이것은 절제 실험의 또 다른 주제가 될 수 있습니다.

활성화 함수 측면에서, SwiGLU는 현대 LLM에서 사실상 표준이 되었으며(GeGLU를 사용하는 Gemma2와 relu²를 사용하는 nvidia([Nvidia et al., 2024](#); [NVIDIA et al., 2025](#))는 예외), ReLU나 GELU와 같은 이전 선택을 대체했습니다.

더 넓은 규모에서, 아키텍처 레이아웃 선택도 모델 동작을 형성하는데 역할을 합니다. 총 파라미터 수가 언어 모델의 용량을 크게 결정하지만, 이러한 파라미터가 깊이와 너비에 걸쳐 어떻게 분배되는지도 중요합니다. [Petty et al. \(2024\)](#)은 더 깊은 모델이 이점이 포화될 때까지 언어 모델링과 구성 태스크에서 동일한 크기의 더 넓은 모델보다 우수한 성능을 보인다는 것을 발견했습니다. 이 "깊고 얕은" 전략은 MobileLLM 절제 실험에서 10억 파라미터 미만의 LLM에 잘 작동하는 반면([Z. Liu et al., 2024](#)), 더 넓은 모델은 더 큰 병렬성 덕분에 더 빠른 추론을 제공하는 경향이 있습니다. 현대 아키텍처는 이 블로그 포스트에서 언급된 것처럼 이 트레이드오프를 다르게 반영합니다.

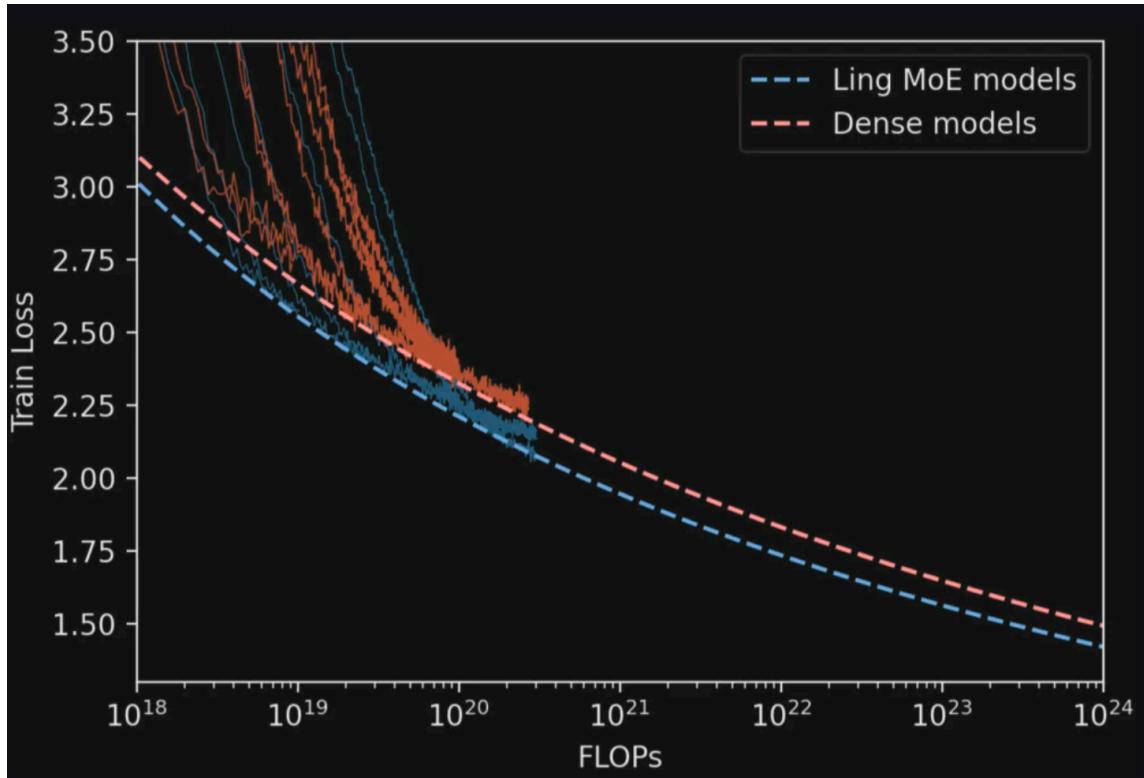
이제 학습 실행을 위해 최적화할 가치가 있는 밀집 트랜스포머 아키텍처의 가장 중요한 측면을 다루었습니다. 그러나 최근 모델 전체와 관련된 다른 아키텍처 개입이 등장했는데, 바로 MoE와 하이브리드 모델입니다. MoE부터 시작하여 이들이 제공하는 것을 살펴보겠습니다.

Going Sparse: MoE

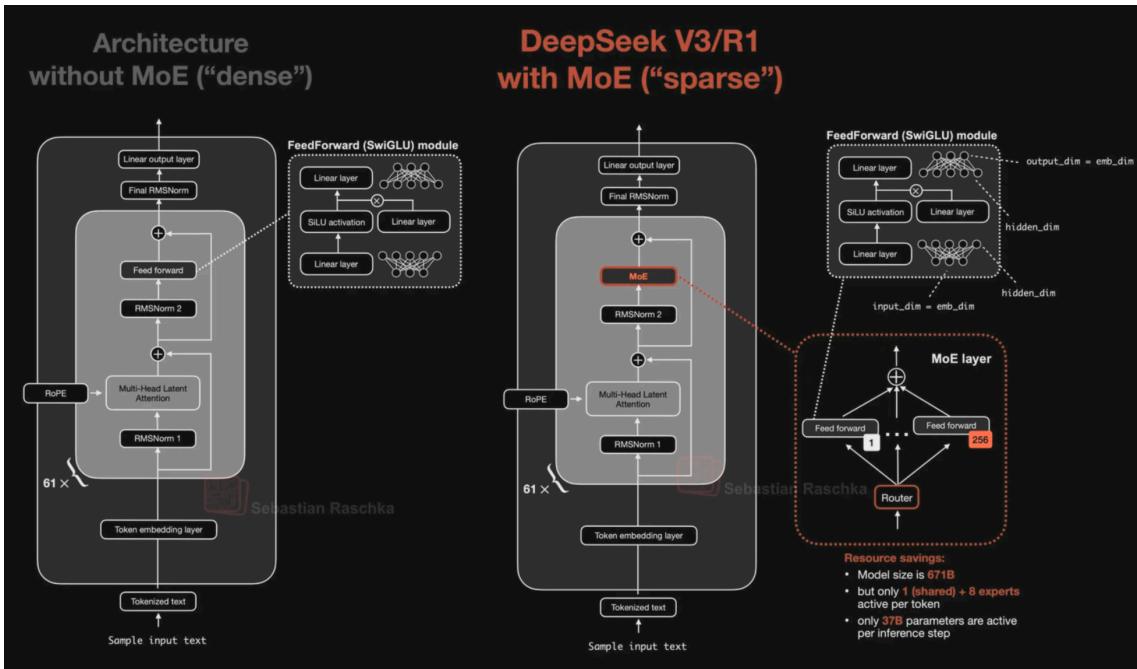
Mixture-of-Experts(MoE)의 직관은 모든 토큰 예측에 전체 모델이 필요하지 않다는 것입니다. 우리 뇌가 당면한 작업에 따라 다른 영역(예: 시각 피질 또는 운동 피질)을 활성화하는 것과 유사합니다. LLM의 경우 이는 코딩 구문을 학습한 부분이 모델이 번역 태스크를 수행할 때 사용될 필요가 없다는 것을 의미할 수 있습니다. 이것을 잘 할 수 있다면, 추론 시 전체 모델의 일부만 실행하면 되므로 많은 컴퓨팅을 절약할 수 있습니다.

기술적 수준에서 MoE는 간단한 목표를 가지고 있습니다. 각 토큰에 대한 "활성" 파라미터 수를 늘리지 않으면서 총 파라미터를 늘리는 것입니다. 다소 단순화하면, 총 파라미터는 모델의 총 학습 용량에 영향을 미치고 활성 파라미터는 학습 비용과 추론 속도를 결정합니다. 그래서 요즘 많은 프론티어 시스템(예: DeepSeek V3, K2, 그리고 Gemini, Grok과 같은 비공개 모델들...)이 MoE 아키텍처를 사용하는 것을 볼 수 있습니다. Ling 1.5 논문([L. Team et al., 2025](#))의 이 플롯은 MoE와

밀집 모델의 스케일링 법칙을 비교합니다.



MoE를 처음 접하더라도 걱정하지 마세요. 메커니즘은 복잡하지 않습니다. 표준 밀집 아키텍처에서 시작하여 MoE에 필요 한 변경 사항을 살펴보겠습니다([Sebastian Raschka](#)의 그림).



MoE에서는 단일 MLP를 여러 MLP("전문가")로 교체하고 MLP 앞에 학습 가능한 라우터를 추가합니다. 각 토큰에 대해 라우터는 실행할 전문가의 작은 부분집합을 선택합니다. 이것이 총 파라미터와 활성 파라미터의 구분이 나오는 지점입니다. 모델은 많은 전문가를 가지고 있지만, 주어진 토큰은 소수만 사용합니다.

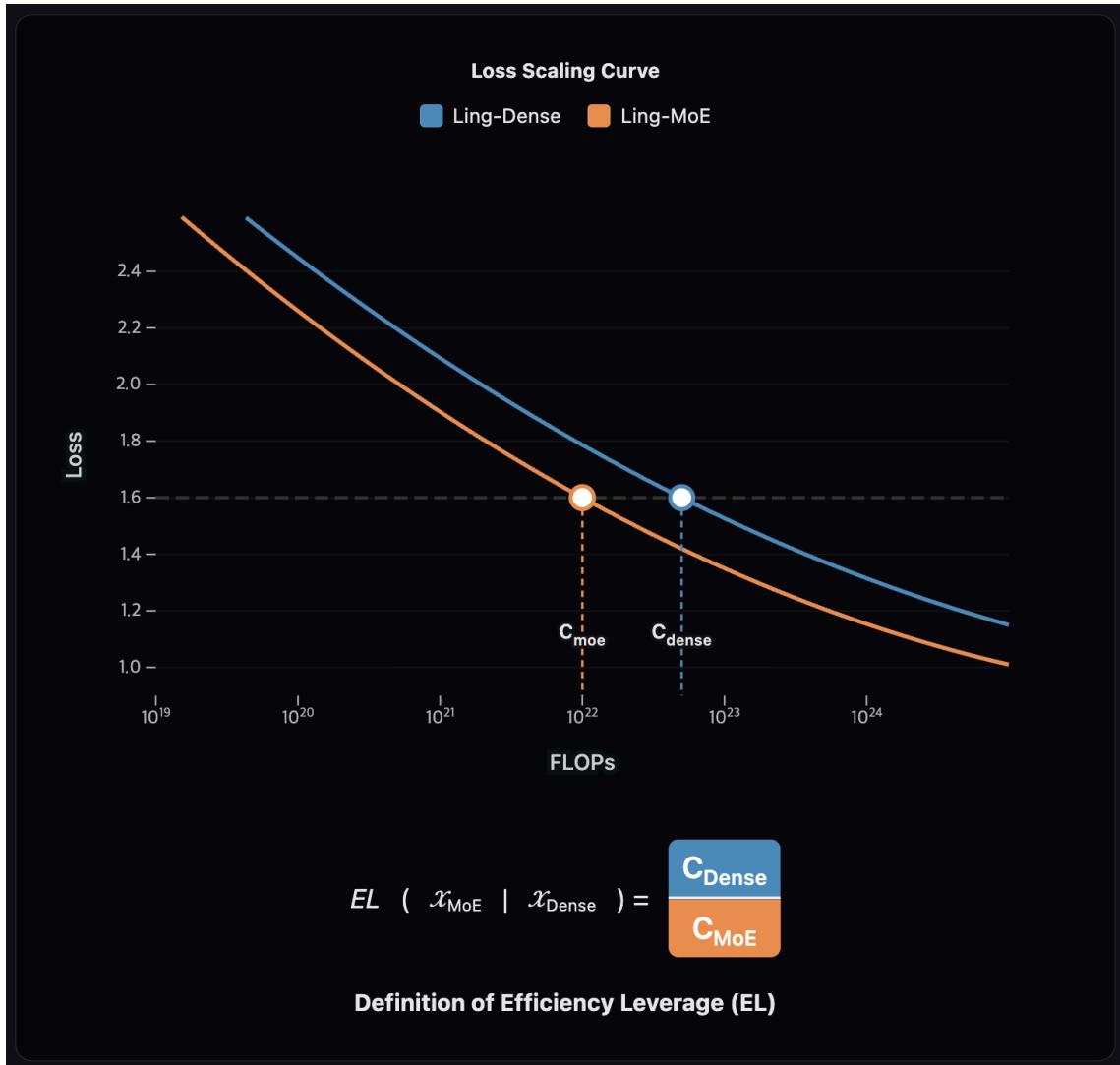
MoE 레이어를 설계할 때 몇 가지 핵심 질문이 제기됩니다.

- **전문가 형태 및 희소성:** 많은 작은 전문가를 사용해야 할까요, 아니면 더 적은 큰 전문가를 사용해야 할까요? 토큰 당 몇 개의 전문가가 활성화되어야 하고, 총 몇 개의 전문가가 필요할까요(즉, 희소성 또는 "top-k")? 일부 전문가는 범용이어서 항상 활성화되어야 할까요?
- **활용 및 전문화:** 라우팅된 전문가를 어떻게 선택하고 잘 활용하면서(유휴 용량 방지) 동시에 전문화를 장려할 수 있을까요? 실제로 이것은 부하 분산 문제이며 학습 및 추론 효율성에 상당한 영향을 미칩니다.

여기서는 하나의 목표에 집중합니다. 고정된 컴퓨팅 예산이 주어졌을 때, 손실을 최소화하는 MoE 구성은 어떻게 선택할까요? 이것은 순수한 시스템 효율성(처리량/지연 시간)과는 다른 질문이며, 나중에 다시 다루겠습니다. 이 섹션의 많은 부분은 Ant Group의 MoE 스케일링 법칙 논문([Tian et al., 2025](#))의 분석을 따릅니다.

그들의 **효율성 레버리지(EL, Efficiency Leverage)** 개념을 사용할 것입니다. 간단히 말해, EL은 MoE 설계가 달성한 손실을 맞추기 위해 필요한 밀집 컴퓨팅의 양을 측정하며, 측정 단위는 FLOPs입니다. 더 높은 EL은 MoE 구성이 밀집 학습에

비해 컴퓨팅 단위당 더 많은 손실 개선을 제공한다는 것을 의미합니다.



TL;DR: 더 많은 희소성 → 더 나은 FLOPs 효율성 → 매우 높은 희소성에서는 수익 체감 → 최적점은 컴퓨팅 예산에 따라 달라집니다.

이 섹션에서는 어떤 MoE 설정이 최적인지 알아보고자 합니다. 접근적으로 두 극단이 이상적인 설정이 아님을 쉽게 알 수 있습니다. 한편으로는, 모든 전문가를 항상 활성화하면 모든 파라미터가 항상 사용되는 dense 설정으로 돌아갑니다. 다른 한편으로는, 활성 파라미터가 매우 낮으면(극단적으로 단 1개의 파라미터만 활성화된다고 생각해 보세요) 좁은 도메인에서도 명확히 작업을 해결하기에 충분하지 않을 것입니다. 따라서 명확히 어떤 중간 지점을 찾아야 합니다. 최적의 설정을 찾는 것에 더 깊이 들어가기 전에 두 가지 양을 정의하는 것이 유용합니다: 활성화 비율과 그 역수인 희소성.

$$\text{activation ratio} = \frac{\#\text{activated experts}}{\#\text{total experts}}$$

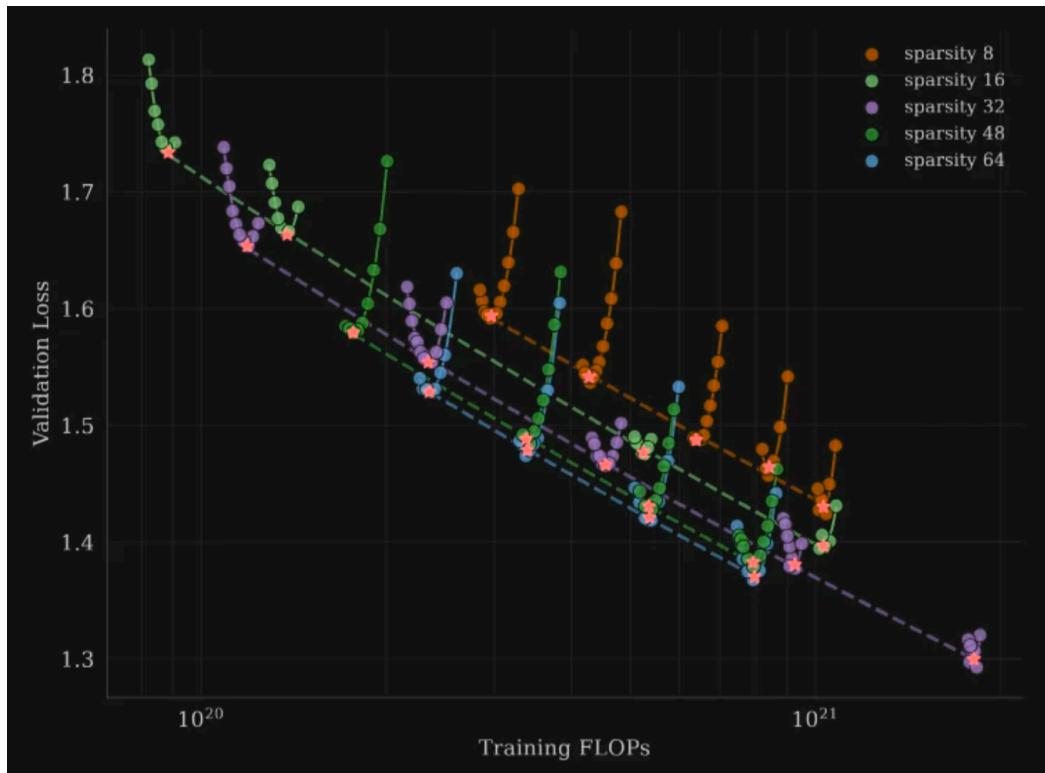
$$\text{sparsity} = \frac{\#\text{total experts}}{\#\text{activated experts}} = \frac{1}{\text{activation ratio}}$$

컴퓨팅 관점에서 비용은 활성 파라미터에 의해서만 구동됩니다. 활성화된 전문가의 수(와 크기)를 고정하고 전체 전문가 수를 늘리면, 추론/학습 FLOPs 예산은 어느 정도 동일하게 유지되지만, 모델 용량을 추가하는 것이므로 충분히 오래 학습하는 한 모델은 일반적으로 더 좋아질 것입니다.

최근 MoE 논문들을 조사하면 몇 가지 흥미로운 경험적 교훈이 있습니다: 활성 전문가의 수와 크기를 고정한 상태에서, 전체 전문가 수를 늘리면(즉, 활성화 비율을 낮추거나 희소성을 높이면) 손실이 개선되지만, 희소성이 매우 높아지면 수익이 체감합니다.

두 가지 예시:

- **Kimi K2 플롯**([K. Team et al., 2025](#)): 두 효과를 모두 보여줍니다. 높은 희소성이 성능을 개선하지만, 희소성이 커질수록 이득이 감소합니다.
- **Ant Group 플롯**([Tian et al., 2025](#)): K2와 동일한 결론이며, 높은 희소성 MoE가 컴퓨팅 증가로부터 더 많은 이점을 얻는다는 추가 결과가 있습니다.



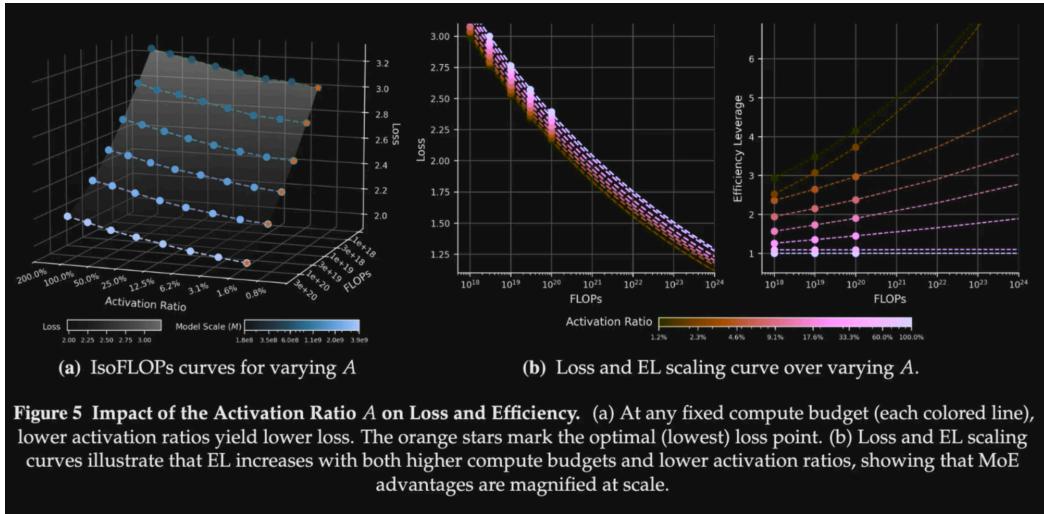


Figure 5 Impact of the Activation Ratio A on Loss and Efficiency. (a) At any fixed compute budget (each colored line), lower activation ratios yield lower loss. The orange stars mark the optimal (lowest) loss point. (b) Loss and EL scaling curves illustrate that EL increases with both higher compute budgets and lower activation ratios, showing that MoE advantages are magnified at scale.

다음은 일부 MoE 모델의 희소성을 보여주는 표입니다.

모델	총 전문가	토큰당 활성화 (공유 포함)	희소성
Mixtral-8x7B	8	2	4.0
Grok-1	8	2	4.0
Grok-2	8	2	4.0
OLMoE-1B-7B-0924	64	8	8.0
gpt-oss 20b	32	4	8
Step-3	48 routed + 1 shared = 49	3 routed + 1 shared = 4	12.25
GLM-4.5-Air	128 routed + 1 shared = 129	8 routed + 1 shared = 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	8 routed + 1 shared = 9	17.8
DeepSeek-V2	160 routed + 2 shared = 162	6 routed + 2 shared = 8	20.25
DeepSeek-V3	256 routed + 1 shared = 257	8 routed + 1 shared = 9	28.6
gpt-oss 120b	128	4	32
Kimi K2	384 routed + 1 shared = 385	8 routed + 1 shared = 9	42.8
Qwen3-Next-80B-A3B-Instruct	512 routed + 1 shared = 513	10 total active + 1 shared = 11	46.6

최근 추세는 명확합니다. MoE 모델이 점점 더 희소해지고 있습니다. 그렇긴 하지만, 최적의 희소성은 여전히 하드웨어와 엔드투엔드 효율성에 따라 달라집니다. 예를 들어, Step-3는 최고 효율성을 목표로 하며 특정 하드웨어 및 대역폭 제약에 맞추기 위해 의도적으로 희소성을 최대화하지 않는 반면, gpt-oss-20b는 온디바이스 메모리 제약으로 인해 낮은 희소성을 가집니다(비활성 전문가도 여전히 일부 메모리를 차지함).

세분성(Granularity)

희소성 외에도, 각 전문가가 얼마나 커야 하는지 결정해야 합니다. 이것은 Ant Group이 도입한 메트릭인 세분성으로 포착됩니다. 이 용어가 무엇을 의미하는지 명확히 해보겠습니다. 용어는 논문마다 다르며, 일부는 약간 다른 공식을 사용합니다. 여기서는 참조하는 플롯과 일치하는 정의를 사용하겠습니다.

$$G = \frac{\alpha * d_{model}}{d_{expert}} \text{ with } \alpha = 2 \text{ or } 4$$

더 높은 세분성 값은 (고정된 파라미터 수가 주어졌을 때) 더 작은 차원을 가진 더 많은 전문가를 갖는 것에 해당합니다. 이 메트릭은 전문가 차원(d_{expert})과 모델 차원(d_{model}) 사이의 비율입니다.

밀집 모델에서 일반적인 경험 법칙은 MLP의 차원을 $d_{intermediate} = 4 * d_{model}$ 로 설정하는 것입니다. $\alpha = 4$ 인 경우 ([Krajewski et al. \(2024\)](#)), 세분성을 밀집 MLP 너비와 일치시키는 데 필요한 전문가 수로 대략 볼 수 있습니다($4d_{model} = d_{intermediate} = G \cdot d_{expert}$).

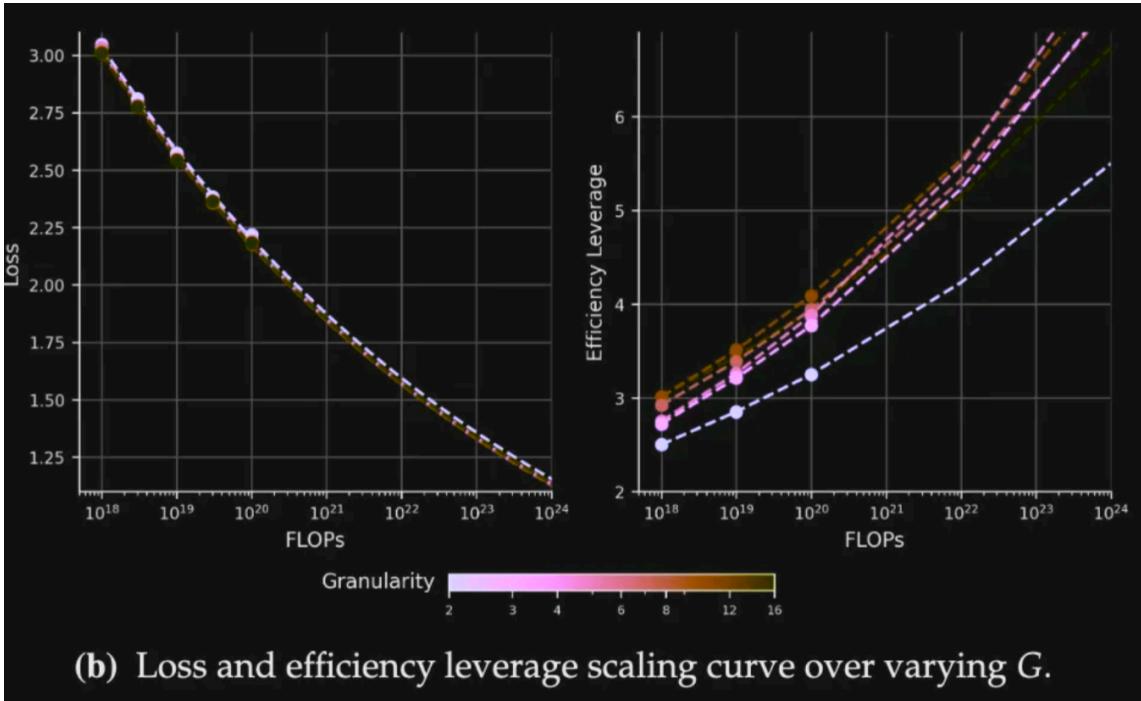
이 해석은 대략적인 휴리스틱일 뿐입니다. 현대 MoE 설계는 종종 단일 밀집 MLP보다 훨씬 더 큰 총 용량을 할당하므로, 일대일 매칭은 실제로 성립하지 않습니다. Ant 팀 설정은 단순히 다른 정규화 선택인 $\alpha = 2$ 를 선택합니다. 일관성을 위해 이 규칙을 선택하고 유지하겠습니다.

G 가 d_{model} 에 따라 스케일링되기 때문에, 모델 너비가 다를 때 모델 간 비교가 까다로울 수 있습니다.

그래도 일부 MoE 릴리스에 대한 다양한 값을 보여주는 표가 있습니다.

모델	(d_{model})	(d_{expert})	($G = 2d_{model}/d_{expert}$)	연도
Mixtral-8x7B	4,096	14,336	0.57	2023
gpt-oss-120b	2880	2880	2.0	2025
gpt-oss-20b	2880	2880	2.0	2025
Grok 2	8,192	16,384	1.0	2024
StepFun Step-3	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	4,096	1,408	5.8	2025
DeepSeek V2	5,120	1,536	6.6	2024
GLM-4.5	5,120	1,536	6.6	2025
Kimi K2	7,168	2,048	7.0	2025
DeepSeek V3	7168	2048	7.0	2024
Qwen3-Next-80B-A3B	2048	512	8.0	2025

세분성이 동작을 어떻게 형성하는지 살펴보겠습니다([Ant Group's paper](#))

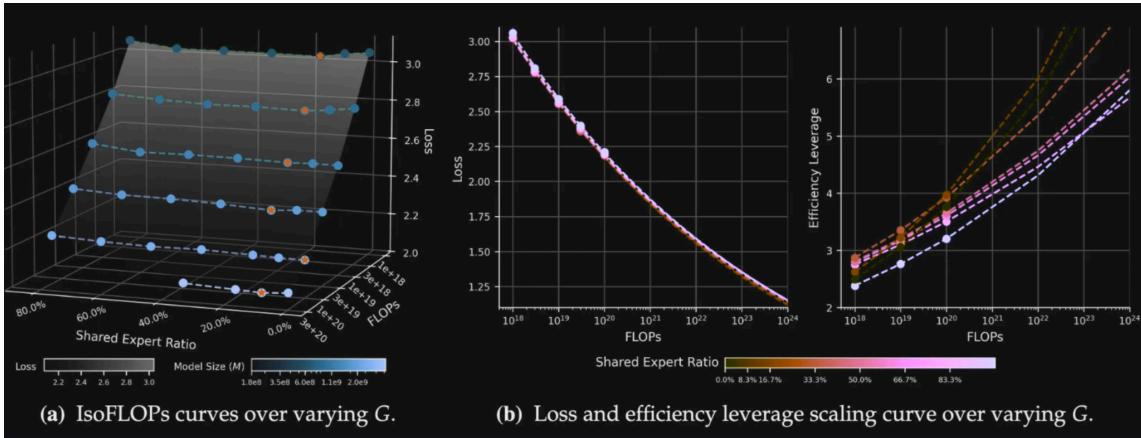


세분성은 EL의 주요 동인처럼 보이지 않습니다. 특히 2 이상으로 올라가면 도움이 되지만, 손실을 결정하는 지배적인 요인은 아닙니다. 그러나 최적 지점이 있습니다. 세분성을 높이는 것은 어느 지점까지는 도움이 되고, 그 이후에는 이득이 평탄해집니다. 따라서 세분성은 최근 릴리스에서 더 높은 값을 향한 명확한 추세를 가진 유용한 조정 노브이지만, 단독으로 최적화해서는 안 됩니다.

MoE를 개선하기 위해 널리 사용되는 또 다른 방법은 shared experts의 개념입니다. 살펴보겠습니다!

Shared experts

shared experts 설정은 모든 토큰을 항상 활성화된 소수의 전문가 집합으로 라우팅합니다. 이러한 shared experts는 데이터의 기본적이고 반복되는 패턴을 흡수하여 나머지 전문가가 더 공격적으로 전문화할 수 있게 합니다. 실제로 보통 많은 수가 필요하지 않습니다. 모델 설계자들은 일반적으로 하나, 많아야 두 개를 선택합니다. 세분성이 증가할수록(예: Qwen3 스타일 설정에서 Qwen3-Next에 더 가까운 것으로 이동), shared experts가 더 유용해지는 경향이 있습니다. 다음 플롯을 보면, 전체적인 영향은 적당합니다. EL을 극적으로 변화시키지는 않습니다. 간단한 경험 법칙이 대부분의 경우 잘 작동합니다. 하나의 공유 전문가만 사용하세요. 이는 DeepSeek V3, K2, Qwen3-Next와 같은 모델의 선택과 일치하며 불필요한 복잡성을 추가하지 않으면서 효율성을 최대화하는 경향이 있습니다. [Tian et al. \(2025\)](#).



따라서 shared expert는 일부 토큰이 항상 라우팅되는 전문가입니다. 다른 전문가들은 어떨까요? 각 전문가에게 언제 라우팅할지 어떻게 학습하고, 단지 소수의 전문가만 사용하지 않도록 어떻게 보장할까요? 다음으로 정확히 그 문제를 다루는 부하 분산에 대해 논의하겠습니다.

Load balancing

Load balancing(부하 분산)은 MoE에서 중요한 부분입니다. 잘못 설정되면 다른 모든 설계 선택을 약화시킬 수 있습니다. 다음 예시에서 부하 분산이 좋지 않으면 왜 많은 문제가 발생하는지 알 수 있습니다. 4개의 GPU가 있고 모델의 4개 전문가를 GPU에 균등하게 분배하는 매우 간단한 분산 학습 설정을 생각해 보세요. 라우팅이 봉괴되어 모든 토큰이 전문가 1로 라우팅되면, GPU의 1/4만 활용된다는 의미이며 이는 학습 및 추론 효율성에 매우 나쁩니다. 게다가, 모든 전문가가 활성화되지 않으므로 모델의 효과적인 학습 용량도 감소했음을 의미합니다.

이 문제를 해결하기 위해 라우터에 추가 손실 항을 추가할 수 있습니다. 아래에서 표준 보조 손실 기반 부하 분산(LBL)을 볼 수 있습니다.

$$L_{Bal} = \alpha \sum_{i=1}^{N_r} f_i P_i$$

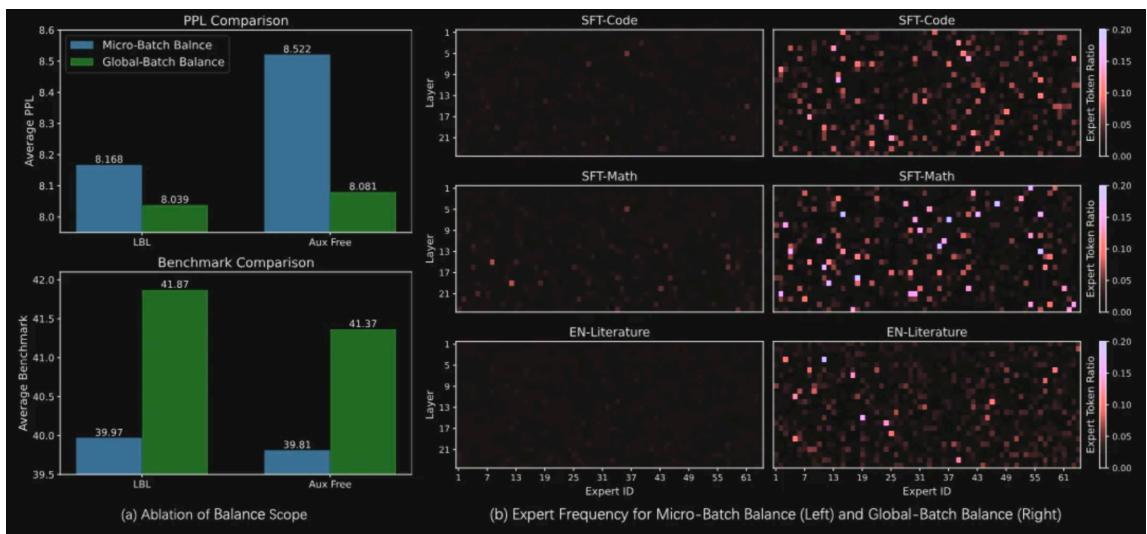
이 간단한 공식은 세 가지 요소만 사용합니다. 계수 α 는 손실의 강도를 결정하고, f_i 는 트래픽 비율로 전문가 i 를 통과하는 토큰의 비율이며, 마지막으로 P_i 는 확률 질량으로 전문가를 통과하는 토큰의 확률을 단순히 합산합니다. 둘 다 필요합니다. f_i 는 실제 분산에 해당하고, P_i 는 부드럽고 미분 가능하여 그래디언트가 흐를 수 있게 합니다. 완벽한 부하 분산을 달성하면 $f_i = P_i = 1/N_r$ 을 얻지만, α 를 어떻게 조정하는지 주의해야 합니다. 값이 너무 작으면 라우팅을 충분히 안내하지 못하고, 너무 크면 라우팅 균일성이 주요 언어 모델 손실보다 더 중요해집니다.

💡 손실 없는 Load balancing

명시적인 손실 항 없이도 분산을 달성할 수 있습니다. DeepSeek v3([DeepSeek-AI et al., 2025](#))는 라우팅 소프트웨어에 들어가는 친화도 점수에 추가되는 간단한 바이어스 항을 도입했습니다. 라우터가 과부하 상태이면 점수를 약간(상수 인자 γ) 감소시켜 선택될 가능성을 낮추고, 전문가가 과소 활용되면 γ 만큼 증가시킵니다. 이 간단한 적응 규칙으로도 부하 분산을 달성합니다.

핵심 세부 사항은 라우팅 통계를 계산하는 범위입니다. f_i 와 P_i 가 로컬 배치(각 워커의 미니 배치)당 계산되나요, 아니면 전역적으로(워커/디바이스 전체에 걸쳐 집계) 계산되나요? Qwen 팀의 분석(Qiu et al., 2025)에 따르면 각 로컬 배치에 충분한 토큰 다양성이 없으면 로컬 계산이 전문가 전문화(라우팅 건강의 좋은 프록시)와 전체 모델 성능 모두에 해를 끼칠 수 있습니다. 전문가 전문화는 하나 이상의 전문가가 특정 도메인에서 다른 전문가보다 더 자주 활성화되는 현상입니다. 다시 말해, 로컬 배치가 좁으면 라우팅 통계가 노이즈가 많거나 편향되어 좋은 분산으로 이어지지 않습니다. 이는 가능할 때마다 전역 통계(또는 최소한 디바이스 간 집계)를 사용해야 함을 의미합니다. 주목할 점은 해당 논문 당시 Megatron을 포함한 많은 프레임워크가 기본적으로 이러한 통계를 로컬로 계산했다는 것입니다.

Qwen 논문의 다음 플롯은 마이크로 배치 vs 전역 배치 집계의 차이와 성능 및 전문화에 미치는 영향을 보여줍니다.



일반적으로 MoE 주변의 아키텍처 선택을 절제 실험하는 것은 많은 측면과의 상호작용이 있기 때문에 까다롭습니다. 예를 들어 shared expert의 유용성은 모델의 세분성에 따라 달라질 수 있습니다. 따라서 정말로 원하는 통찰을 얻기 위해 좋은 실험 세트를 갖추는 데 시간을 투자할 가치가 있습니다!

이제 MoE의 기본 사항을 다루었지만, 아직 발견할 것이 더 있습니다. 추가로 연구할 항목의 비완전한 목록입니다.

- 제로 계산 전문가, MoE 레이어 재스케일링 및 학습 모니터링 (LongCat-Flash 논문).
- 직교 손실 부하 분산 (ERNIE 4.5에서처럼).
- 학습 전반에 걸쳐 부하 분산 계수 스케줄링.
- MoE와의 아키텍처/최적화 상호작용, 예를 들어:
 - MoE에서 옵티마이저 순위가 변경되는지 여부.
 - MoE에 MuP를 적용하는 방법.
 - MoE의 학습률을 조정하는 방법 (배치당 동일한 수의 토큰을 보지 않으므로).
 - 시작 부분의 밀집 레이어 수.
- 그 외 많은 것들...

열정적인 독자 여러분이 토끼굴을 더 깊이 따라가는 것은 여러분에게 맡기고, 이제 마지막 주요 아키텍처 선택인 하이브리드 모델로 넘어가겠습니다!

Excursion: 하이브리드 모델

최근 추세는 표준 밀집 또는 MoE 아키텍처를 상태 공간 모델(SSM) 또는 선형 어텐션 메커니즘으로 보강하는 것입니다 ([\(MiniMax et al., 2025; Zuo et al., 2025\)](#)). 이러한 새로운 종류의 모델은 트랜스포머의 근본적인 약점 중 일부를 해결하려고 합니다. 매우 긴 컨텍스트를 효율적으로 처리하는 것입니다. 이들은 임의 길이의 컨텍스트를 효율적으로 처리하고 선형적으로 스케일링할 수 있지만 컨텍스트의 정보를 활용하는데 어려움을 겪을 수 있는 순환 모델과, 긴 컨텍스트에서 매우 비용이 많이 들지만 컨텍스트의 패턴을 매우 잘 활용할 수 있는 트랜스포머 사이의 중간 지점을 취합니다.

예를 들어 Mamba 모델(SSM의 한 형태)의 약점을 이해하기 위한 몇 가지 연구([Waleffe et al., 2024](#))가 있었으며, 그러한 모델이 많은 벤치마크에서 잘 수행되지만 예를 들어 MMLU에서는 성능이 떨어지며, 인컨텍스트 학습의 부족이 격차를 야기한다고 가정합니다. 그래서 두 세계의 장점을 얻기 위해 밀집 또는 MoE 모델의 블록과 결합되며, 따라서 하이브리드 모델이라는 이름이 붙었습니다.

이러한 선형 어텐션 방법의 핵심 아이디어는 계산을 재정렬하여 어텐션이 더 이상 긴 컨텍스트에서 다루기 어려운 $O(n^2d)$ 비용이 들지 않게 하는 것입니다. 어떻게 작동할까요? 먼저 추론 시 어텐션 공식을 상기해 보세요. 토큰 t 에 대한 출력을 생성하는 것은 다음과 같습니다.

$$o_t = \sum_{j=1}^t \frac{\exp(q_t^\top k_j)}{\sum_{l=1}^t \exp(q_t^\top k_l)} v_j$$

이제 소프트맥스를 제거합니다.

$$o_t = \sum_{j=1}^t (q_t^\top k_j) v_j$$

재정렬하면 다음을 얻습니다.

$$\sum_{j=1}^t (q_t^\top k_j) v_j = \left(\sum_{j=1}^t v_j k_j^\top \right) q_t$$

누적 상태를 정의합니다.

$$S_t \triangleq \sum_{j=1}^t k_j v_j^\top = K_{1:t}^\top V_{1:t} \in \mathbb{R}^{d \times d}$$

간단한 업데이트로:

$$S_t = S_{t-1} + k_t v_t^\top$$

따라서 다음과 같이 쓸 수 있습니다.

$$o_t = S_t q_t = S_{t-1} q_t + v_t (k_t^\top q_t)$$

재정렬이 왜 중요한지: 왼쪽 형태 $\sum_{j \leq t} (q_t^\top k_j) v_j$ 는 "각 과거 토큰 j 에 대해, 내적 $q_t^\top k_j$ (스칼라)를 계산하고, 이를 사용하여 v_j 를 스케일링하고, 그 t 개 벡터를 더함"을 의미합니다. 이는 스텝 t 에서 약 $O(td)$ 작업입니다.

오른쪽 형태는 이를 $(\sum_{j \leq t} v_j k_j^\top) q_t$ 로 다시 씁니다. 모든 과거 (k_j, v_j) 를 이미 요약하는 단일 누적 상태 행렬 $S_t = \sum_{j \leq t} v_j k_j^\top \in \mathbb{R}^{d \times d}$ 를 유지합니다.

각 새 토큰은 하나의 외적 $v_t k_t^\top$ 으로 업데이트하며 비용은 $O(d^2)$ 이고, 출력은 단지 하나의 행렬-벡터 곱 $S_t q_t$ (또 다른 $O(d^2)$)입니다.

따라서 왼쪽 형태로 처음부터 T 개 토큰을 생성하는 것은 $O(T^2d)$ 인 반면, S_t 를 유지하고 오른쪽 형태를 사용하는 것은 $O(Td^2)$ 입니다.

직관적으로: 왼쪽 = "각 스텝마다 많은 작은 내적-스케일-더하기"; 오른쪽 = "미리 요약된 하나의 행렬 곱하기 쿼리", 시퀀스 길이에 대한 의존성을 차원에 대한 의존성으로 교환합니다.

여기서는 추론과 순환 형태에 집중하지만, 학습에서도 더 효율적이며, 재정렬은 다음 방정식처럼 간단합니다.

$$\underbrace{(QK^\top)}_{n \times n} V = Q \underbrace{(K^\top V)}_{d \times d}$$

이제 이것이 RNN과 같은 구조와 매우 유사해 보임을 알 수 있습니다. 문제가 해결되었죠? 거의요. 실제로 소프트맥스는 중요한 안정화 역할을 하며, 순진한 선형 형태는 어떤 정규화 없이는 불안정할 수 있습니다. 이것이 lightning 또는 norm attention이라는 실용적인 변형을 둘기 부여합니다!

Lightning과 norm attention

이 계열은 Minimax01([MiniMax et al., 2025](#))에 나타나며, 더 최근에는 Ring-linear([L. Team, Han, et al., 2025](#))에 나타납니다. Norm Attention 아이디어([Qin et al., 2022](#))를 기반으로 합니다. 핵심 단계는 간단합니다. 출력을 정규화하는 것입니다. "Lightning" 변형은 구현을 빠르고 효율적으로 만드는데 집중하고 공식을 약간 다르게 만들습니다. 다음은 둘 다의 공식입니다.

NormAttention:

$$\text{RMSNorm}(Q(K^T V))$$

LightningAttention:

$$Q = \text{Silu}(Q), \quad K = \text{Silu}(K), \quad V = \text{Silu}(V)$$

$$O = \text{SRMSNorm}(Q(KV^T))$$

경험적으로, Minimax01에 따르면 Norm attention을 가진 하이브리드 모델은 대부분의 태스크에서 소프트맥스와 일치합니다.



여기서 흥미로운 점은 Needle in a Haystack(NIAH)와 같은 검색 태스크에서 전체 소프트맥스 어텐션보다 훨씬 더 잘 수행할 수 있다는 것인데, 이는 놀라워 보이지만 소프트맥스와 선형 레이어가 함께 작동할 때 어떤 시너지가 있음을 나타낼 수 있습니다!

MiniMax M2

놀랍게도 최근 출시된 MiniMax M2는 하이브리드나 선형 어텐션을 사용하지 않습니다. 그들의 [pretraining lead](#)에 따르면, 초기 MiniMax M1의 Lightning Attention 실험이 당시 인기 있던 벤치마크(MMLU, BBH, MATH)에서 더 작은 규모에서는 유망해 보였지만, 더 큰 규모에서는 “복잡한 다중 흡수 태스크에서 명확한 결함”이 있음을 발견했습니다. 그들은 또한 RL 학습 중 수치 정밀도 문제와 인프라 성숙도를 주요 장애물로 언급합니다. 그들은 대규모 아키텍처를 만드는 것이 데이터 분포, 옵티마이저 등과 같은 다른 파라미터에 대한 민감성으로 인해 어렵고 컴퓨팅 집약적인 단변수 문제라고 결론짓습니다.

그러나 그들은 “GPU 컴퓨팅 성장이 둔화되는 반면 데이터 길이가 계속 증가함에 따라, 선형 및 희소 어텐션의 이점이 점차 나타날 것”이라고 인정합니다. 이는 아키텍처 절제 실험의 복잡성과 연구와 프로덕션 현실 사이의 격차를 모두 강조합니다.

이제 이러한 방법 중 일부를 더 살펴보고 통합된 프레임워크로 어떻게 이해할 수 있는지 살펴보겠습니다.

고급 선형 어텐션

순환 모델에서 얻은 유용한 교훈은 상태가 때때로 과거를 놓아주게 하는 것입니다. 실제로 이는 이전 상태에 대한 게이트 G_t 를 도입하는 것을 의미합니다.

$$S_t = G_t \odot S_{t-1} + v_t k_t^\top$$

거의 모든 최근 선형 어텐션 방법은 G_t 의 다른 구현만 가진 이 게이팅 구성 요소를 가지고 있습니다. 다음은 이 [this paper](#)에서 가져온 게이트의 다양한 변형과 해당 아키텍처의 목록입니다.

모델	파라미터화	학습 가능한 파라미터
Mamba (Gu & Dao, 2024)	$G_t = \exp(-(\mathbf{1}^\top \alpha_t) \odot \exp(A))$, $\alpha_t = \text{softplus}(x_t W_{\alpha_1} W_{\alpha_2})$	$A \in \mathbb{R}^{d_k \times d_v}$, $W_{\alpha_1} \in \mathbb{R}^{d \times \frac{d}{16}}$, $W_{\alpha_2} \in \mathbb{R}^{\frac{d}{16} \times d_v}$
Mamba-2 (Dao & Gu, 2024)	$G_t = \gamma_t \mathbf{1}^\top \mathbf{1}$, $\gamma_t = \exp(-\text{softplus}(x_t W_\gamma) \exp(a))$	$W_\gamma \in \mathbb{R}^{d \times 1}$, $a \in \mathbb{R}$
mLSTM (Beck et al., 2025 ; H. Peng et al., 2021)	$G_t = \gamma_t \mathbf{1}^\top \mathbf{1}$, $\gamma_t = \sigma(x_t W_\gamma)$	$W_\gamma \in \mathbb{R}^{d \times 1}$
Gated Retention (Sun et al., 2024)	$G_t = \gamma_t \mathbf{1}^\top \mathbf{1}$, $\gamma_t = \sigma(x_t W_\gamma)^{\frac{1}{2}}$	$W_\gamma \in \mathbb{R}^{d \times 1}$
DFW (Mao, 2022; Pramanik et al., 2023) (Mao, 2022)	$G_t = \alpha_t^\top \beta_t$, $\alpha_t = \sigma(x_t W_\alpha)$, $\beta_t = \sigma(x_t W_\beta)$	$W_\alpha \in \mathbb{R}^{d \times d_k}$, $W_\beta \in \mathbb{R}^{d \times d_v}$
GateLoop (Katsch, 2024)	$G_t = \alpha_t^\top \mathbf{1}$, $\alpha_t = \sigma(x_t W_{\alpha_1}) \exp(x_t W_{\alpha_2} \mathbf{1})$	$W_{\alpha_1} \in \mathbb{R}^{d \times d_k}$, $W_{\alpha_2} \in \mathbb{R}^{d \times d_k}$
HGRN-2 (Qin et al., 2024)	$G_t = \alpha_t^\top \mathbf{1}$, $\alpha_t = \gamma + (1 - \gamma) \sigma(x_t W_\alpha)$	$W_\alpha \in \mathbb{R}^{d \times d_k}$, $\gamma \in (0, 1)^{d_k}$
RWKV-6 (B. Peng et al., 2024)	$G_t = \alpha_t^\top \mathbf{1}$, $\alpha_t = \exp(-\exp(x_t W_\alpha))$	$W_\alpha \in \mathbb{R}^{d \times d_k}$
Gated Linear Attention (GLA)	$G_t = \alpha_t^\top \mathbf{1}$, $\alpha_t = \sigma(x_t W_{\alpha_1} W_{\alpha_2})^{\frac{1}{2}}$	$W_{\alpha_1} \in \mathbb{R}^{d \times 16}$, $W_{\alpha_2} \in \mathbb{R}^{16 \times d_k}$

G_t 의 파라미터화가 다른 최근 모델들의 게이트 선형 어텐션 공식. 바이어스 항은 생략되었습니다.

목록에서 주목할 만한 변형 중 하나는 Mamba-2([Dao & Gu, 2024](#))입니다. Nemotron-H([NVIDIA, :, Blakeman, et al., 2025](#)), Falcon H1([Zuo et al., 2025](#)), Granite-4.0-h([IBM Research, 2025](#))와 같은 많은 하이브리드 모델에서 사용됩니다.

그러나 아직 초기 단계이며 대규모 하이브리드 모델로 스케일링할 때 고려해야 할 중요한 뉘앙스가 있습니다. 유망해 보이지만, MiniMax의 [M2](#) 경험은 소규모에서의 이점이 항상 대규모 프로덕션 시스템으로 전환되지 않는다는 것을 강조합니다. 특히 복잡한 추론 태스크, RL 학습 안정성, 인프라 성숙도에서 그렇습니다. 그렇긴 하지만, 하이브리드 모델은 빠르게 발전하고 있으며 프론티어 학습을 위한 견고한 선택으로 남아 있습니다. Qwen3-Next(게이트 DeltaNet 업데이트 포함)([Qwen Team, 2025](#))는 긴 컨텍스트에서 추론이 더 빠르고, 학습이 더 빠르며, 일반적인 벤치마크에서 더 강하다고 보고합니다. 우리는 또한 새로운 [“Kimi Delta Attention”](#)을 사용할 가능성이 높은 Kimi의 다음 모델을 기대하고 있습니다. 선형 어텐션과 동일한 기 컨텍스트 문제를 어텐션을 계산할 블록이나 쿼리를 선택하여 해결하는 Sparse Attention도 언급하겠습니다. 몇 가지 예로는 Native Sparse Attention([Yuan et al., 2025](#)), DeepSeek Sparse Attention([DeepSeek-AI, 2025](#)), InfLLM v2([M. Team, Xiao, et al., 2025](#))가 있습니다.

토크나이저로 넘어가기 전에 밀집 모델, MoE 또는 하이브리드 모델을 학습시킬지 결정하기 위한 작은 의사결정 트리를 구축하여 아키텍처 선택을 마무리하겠습니다.

MoE를 할 것인가 말 것인가: 기본 아키텍처 선택

이제 Dense, MoE, 하이브리드 모델을 보았으므로 어떤 것을 사용해야 하는지 자연스럽게 궁금할 것입니다. 아키텍처 선택은 일반적으로 모델을 배포할 위치, 팀의 전문성, 일정에 따라 달라집니다. 각 아키텍처의 장단점을 간략히 살펴보고 적합한 아키텍처를 찾기 위한 간단한 안내 프로세스를 제시하겠습니다.

Dense Transformers는 모든 파라미터가 모든 토큰에 대해 활성화되는 기본 표준 디코더 전용 트랜스포머입니다. 수학은 [The Annotated Transformers](#)를, 직관을 쌓으려면 [The Illustrated Transformers](#)를 참조하세요.

- **장점:** 널리 지원됨, 잘 이해됨, 안정적인 학습, 파라미터당 좋은 성능.
- **단점:** 컴퓨팅이 크기에 따라 선형적으로 스케일링됨, 70B 모델은 3B보다 ~23배 더 비용이 들.

이것은 보통 메모리 제약이 있는 사용 사례나 새로운 LLM 학습자에게 기본 선택입니다.

Mixture of Experts(MoE)는 트랜스포머의 피드포워드 레이어를 여러 "전문가"로 교체합니다. 각 토큰에 대해 게이팅 네트워크가 소수의 전문가에게만 라우팅합니다. 결과는 컴퓨팅의 일부로 대규모 네트워크의 용량을 얻는 것입니다. 예를 들어 [Kimi K2](#)는 총 1T 파라미터를 가지지만 토큰당 32B만 활성화됩니다. 문제는 모든 전문가가 메모리에 로드되어야 한다는 것입니다. 시각적 가이드와 복습은 [이 블로그](#)를 확인하세요.

- **장점:** 학습과 추론에서 컴퓨팅당 더 나은 성능.
- **단점:** 높은 메모리(모든 전문가가 로드되어야 함). 밀집 트랜스포머보다 더 복잡한 학습. 프레임워크 지원이 개선되고 있지만 밀집 모델보다 덜 성숙함. 그리고 분산 학습은 전문가 배치, 부하 분산, all-to-all 통신 문제로 인해 악동임.

메모리 제약이 없고 컴퓨팅당 최대 성능을 원할 때 사용하세요.

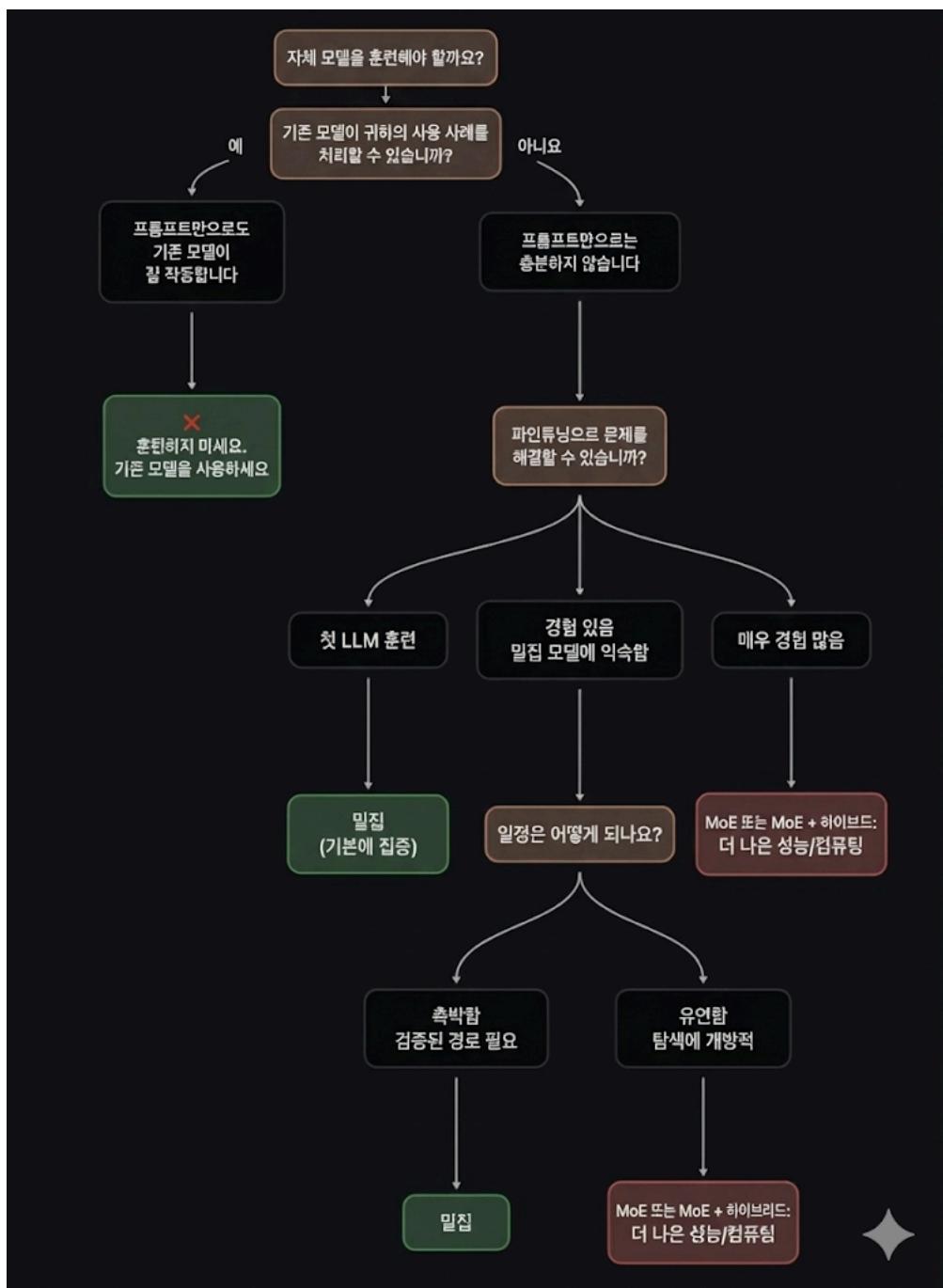
하이브리드 모델은 트랜스포머를 Mamba와 같은 상태 공간 모델(SSM)과 결합하여, 어텐션의 이차 스케일링에 비해 일부 연산에 선형 복잡도를 제공합니다. ([Mathy blog](#) | [Visual guide](#))

- **장점:** 잠재적으로 더 나은 긴 컨텍스트 처리. 매우 긴 시퀀스에 더 효율적.
- **단점:** 검증된 학습 레시피가 적어 밀집 및 MoE보다 덜 성숙함. 제한된 프레임워크 지원.

표준 트랜스포머의 추론 오버헤드를 줄이면서 매우 큰 컨텍스트로 스케일링하고 싶다면 사용하세요.

일부 팀이 텍스트용 디퓨전 모델을 탐구하는 것도 보고 있지만, 이러한 모델(및 기타 실험적 대안)은 이 문서의 범위를 벗어난 매우 초기 단계로 간주합니다.

요약하면, 모델이 배포될 위치를 먼저 질문하세요. 그런 다음 팀의 전문성과 학습 일정을 고려하여 얼마나 많은 탐구를 감당할 수 있는지 평가하세요.



SmoILM3의 경우 온디바이스 배포를 위한 강력한 소형 모델을 구축하고자 했고, 약 3개월의 일정이 있었으며, 과거에 주로 밀집 모델을 학습시켰습니다. 이로 인해 MoE(메모리 제약)와 하이브리드(새로운 아키텍처를 탐구할 짧은 일정, 그리고 밀집 모델이 목표로 한 최대 128k 토큰의 긴 컨텍스트를 달성할 수 있었음)가 제외되어 llama 스타일의 밀집 모델을 선택했습니다.

이제 모델 아키텍처의 내부를 연구했으니 데이터와 모델 사이의 다리 역할을 하는 토크나이저를 살펴보겠습니다.

토크나이저

아키텍처 혁신에서 주목받는 경우는 드물지만, 토큰화 방식은 아마도 모든 언어 모델에서 가장 과소평가된 구성 요소 중 하나입니다. 이것을 인간 언어와 모델이 살고 있는 수학적 세계 사이의 번역기라고 생각하세요. 모든 번역기와 마찬가지로 번역의 품질이 매우 중요합니다. 그렇다면 우리의 필요에 맞는 올바른 토크나이저를 어떻게 구축하거나 선택할 수 있을까요?

토크나이저 기본 사항

핵심적으로 토크나이저는 원시 텍스트를 토큰이라고 불리는 개별 처리 가능한 단위로 분할하여 모델이 처리할 수 있는 숫자 시퀀스로 변환합니다. 기술적 세부 사항에 들어가기 전에, 토크나이저 설계를 안내할 몇 가지 근본적인 질문에 먼저 답해야 합니다.

- 어떤 언어를 지원하고 싶은가요? 다국어 모델을 구축하지만 토크나이저가 영어만 본 경우, 비영어 텍스트를 만날 때 모델이 비효율적이 되어 필요 이상으로 훨씬 더 많은 토큰으로 분할됩니다. 이는 성능, 학습 비용, 추론 속도에 직접 영향을 미칩니다.
- 어떤 도메인이 중요한가요? 언어 외에도 수학과 코드와 같은 도메인은 숫자의 신중한 표현이 필요합니다.
- 목표 데이터 훈합을 알고 있나요? 토크나이저를 처음부터 학습시킬 계획이라면, 이상적으로는 최종 학습 훈합을 반영하는 샘플에서 학습시켜야 합니다.

이러한 질문에 답한 후, 주요 설계 결정을 검토할 수 있습니다.

토큰화 기본 사항에 대한 깊은 이해를 위해 Andrey Karpathy의 "Let's build the GPT Tokenizer"는 훌륭한 실습 튜토리얼입니다. 토크나이저 소개와 여러 외부 리소스를 제공하는 이 리소스도 확인할 수 있습니다.

어휘 크기

어휘는 본질적으로 모델이 인식하는 모든 토큰(단어, 서브워드, 기호와 같은 최소 텍스트 단위)을 나열하는 사전입니다.

더 큰 어휘는 문장당 더 적은 토큰을 생성하므로 텍스트를 더 효율적으로 압축하지만, 계산적 트레이드오프가 있습니다. 어휘 크기는 임베딩 행렬의 크기에 직접 영향을 미칩니다. 어휘 크기가 V 이고 은닉 차원이 h 이면, 입력 임베딩은 $V \times h$ 파라미터를 가지고, 출력 레이어는 또 다른 $V \times h$ 파라미터를 가집니다. 소형 모델의 경우 "임베딩 공유" 섹션에서 본 것처럼 이것이 총 파라미터의 상당한 부분이 되지만, 모델이 스케일업됨에 따라 상대적 비용은 줄어듭니다.

최적 지점은 목표 커버리지와 모델 크기에 따라 달라집니다. 영어 전용 모델의 경우 약 50k 토큰이 보통 충분하지만, 다국어 모델은 다양한 문자 체계와 언어를 효율적으로 처리하기 위해 종종 100k 이상이 필요합니다. Llama3와 같은 현대 최신 모델은 다양한 언어에서 토큰 효율성을 개선하기 위해 128k 이상 범위의 어휘를 채택했습니다. 동일 패밀리의 소형 모델은 더 큰 어휘의 이점을 누리면서 임베딩 파라미터의 비율을 줄이기 위해 임베딩 공유를 적용합니다. [Dagan et al. \(2024\)](#)은 어휘 크기가 압축, 추론, 메모리에 미치는 영향을 분석합니다. 그들은 더 큰 어휘로부터의 압축 이득이 기하급수적으로 감소하여 최적의 크기가 존재함을 시사한다고 관찰합니다. 추론의 경우, 더 큰 모델은 압축이 추가 임베딩 토큰이 소프트맥스에서 비용이 드는 것보다 순방향 패스에서 더 많이 절약하기 때문에 더 큰 어휘의 이점을 얻습니다. 메모리의 경우, 최적의 크기는 시퀀스 길이와 배치 크기에 따라 달라집니다. 더 긴 컨텍스트와 큰 배치는 더 적은 토큰으로 인한 KV 캐시 절약 덕분에 더 큰 어휘의 이점을 얻습니다.

처리량을 최적화하려면 어휘 크기를 128의 배수로 설정하세요(예: 50,000 대신 50,304). 현대 GPU는 차원이 2의 더 높은 거듭제곱으로 나누어질 때 행렬 연산을 더 잘 수행하는데, 이는 효율적인 메모리 정렬을 보장하고 정렬되지 않

은 메모리 접근으로 인한 오버헤드를 줄이기 때문입니다. 자세한 내용은 이 블로그에서 확인하세요.

토큰화 알고리즘

BPE(Byte-Pair Encoding)([Sennrich et al., 2016](#))가 가장 인기 있는 선택으로 남아 있으며, WordPiece나 SentencePiece와 같은 다른 알고리즘도 존재하지만 덜 채택되었습니다. 또한 바이트나 문자에서 직접 작동하여 토큰화를 완전히 제거할 수 있는 토크나이저 없는 접근 방식에 대한 연구 관심이 증가하고 있습니다.

이제 토크나이저를 정의하는 주요 파라미터를 보았으니, 실용적인 결정에 직면합니다. 기존 토크나이저를 사용해야 할까요, 아니면 처음부터 학습시켜야 할까요? 답은 커버리지에 달려 있습니다. 목표 어휘 크기를 가진 기존 토크나이저가 우리의 언어와 도메인을 잘 처리하는지 여부입니다.

아래 그림은 GPT-2의 영어 전용 토크나이저([Radford et al., 2019](#))와 Gemma 3의 다국어 토크나이저([G. Team, Kamath, et al., 2025](#))가 동일한 영어 및 아랍어 문장을 어떻게 분할하는지 비교합니다.

하지만 토크나이저의 품질을 측정하기 위해 몇 가지 토큰화 예시만 눈으로 보고 좋다고 할 수는 없습니다. 절제 실험 없이 직관에 기반하여 아키텍처를 변경할 수 없는 것과 마찬가지입니다. 토크나이저 품질을 평가하기 위한 구체적인 메트릭이 필요합니다.

토크나이저 품질 측정

토크나이저가 얼마나 잘 수행되는지 평가하기 위해, FineWeb2([Penedo et al., 2025](#))에서 사용된 두 가지 주요 메트릭을 사용할 수 있습니다.

다산성(Fertility)

단어를 인코딩하는 데 필요한 평균 토큰 수를 측정합니다. 낮은 다산성은 더 나은 압축을 의미하며, 이는 더 빠른 학습과 추론으로 이어집니다. 이렇게 생각해 보세요. 한 토크나이저가 대부분의 단어를 인코딩하는 데 하나 또는 두 개 더 많은 토큰이 필요한 반면 다른 토크나이저는 더 적은 토큰으로 처리한다면, 두 번째 토크나이저가 분명히 더 효율적입니다.

다산성을 측정하는 표준 접근 방식은 단어 대 토큰 비율(단어 다산성)을 계산하는 것으로, 평균적으로 단어당 몇 개의 토큰이 필요한지 측정합니다. 이 메트릭은 적절한 단어 토크나이저가 사용 가능할 때(예: [Spacy](#) 와 [Stanza](#) ([Penedo et al., 2025](#))) 의미 있는 교차 언어 비교를 제공하기 때문에 단어 개념을 중심으로 정의됩니다.

단일 언어 대해 토크나이저를 비교할 때, 문자 대 토큰 비율이나 바이트 대 토큰 비율([Dagan et al., 2024](#))을 얻기 위해 단어 대신 문자 수나 바이트 수를 사용할 수도 있습니다. 그러나 이러한 메트릭은 교차 언어 비교에 한계가 있습니다. 바이트는 다른 스크립트의 문자가 다른 바이트 표현을 필요로 하기 때문에 왜곡될 수 있습니다(예: 중국어 문자는 UTF-8에서 3바이트를 사용하는 반면 라틴 문자는 1-2바이트를 사용). 마찬가지로 문자 수를 사용하는 것은 언어에 따라 단어 길이가 극적으로 다르다는 사실을 고려하지 않습니다. 예를 들어, 중국어 단어는 독일어 복합어보다 훨씬 짧은 경향이 있습니다.

연속 단어 비율(Proportion of continued words)

이 메트릭은 여러 조각으로 분할되는 단어의 비율을 알려줍니다. 더 낮은 비율이 더 좋은데, 이는 더 적은 단어가 조각화되어 더 효율적인 토큰화로 이어지기 때문입니다.

이러한 메트릭을 구현해 보겠습니다.

```
import numpy as np

def compute_tokenizer_metrics(tokenizer, word_tokenizer, text):
    """
    다산성과 연속 단어 비율을 계산합니다.

    Returns:
        tuple: (fertility, proportion_continued_words)
        - fertility: 단어당 평균 토큰 수 (낮을수록 좋음)
        - proportion_continued_words: 2개 이상 토큰으로 분할된 단어 비율 (낮을수록 좋음)

    """
    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 = sum(tokens_per_word > 1) / len(tokens_per_word)

    return fertility, proportion_continued_words
```

```
proportion_continued_words = (tokens_per_word >= 2).sum() / len(tokens_per_word)

return fertility, proportion_continued_words
```

그러나 코드와 수학과 같은 특수 도메인의 경우, 다산성 외에도 토크나이저가 도메인별 패턴을 얼마나 잘 처리하는지 더 깊이 살펴봐야 합니다. 대부분의 현대 토크나이저는 단일 숫자 분할을 수행합니다(따라서 "123"은 ["1", "2", "3"]이 됨) ([Chowdhery et al., 2022](#); [DeepSeek-AI et al., 2024](#)) 숫자를 분리하는 것이 직관에 반하는 것처럼 보일 수 있지만, 실제로 모델이 산술 패턴을 더 효과적으로 학습하는 데 도움이 됩니다. "342792"가 하나의 분할 불가능한 토큰으로 인코딩되면, 모델은 그 특정 토큰을 다른 모든 숫자 토큰과 더하거나 빼거나 곱할 때 무슨 일이 일어나는지 암기해야 합니다. 그러나 분할되면 모델은 숫자 수준 연산이 어떻게 작동하는지 학습합니다. Llama3([Grattafiori et al., 2024](#))와 같은 일부 토크나이저는 1부터 999까지의 숫자를 고유한 토큰으로 인코딩하고 나머지는 이러한 토큰으로 구성됩니다.

토큰화가 산술 성능에 미치는 영향에 대한 더 깊은 이해를 위해, "*From Digits to Decisions: How Tokenization Impacts Arithmetic in LLMs*"는 수학 태스크에서 다양한 토큰화 방식을 비교합니다.

따라서 토크나이저의 약점과 강점을 평가하기 위해 목표 도메인에서 다산성을 측정할 수 있습니다. 아래 표는 다양한 언어와 도메인에 대해 인기 있는 토크나이저의 다산성을 비교합니다.

토크나이저 평가

다양한 언어에서 토크나이저를 비교하기 위해, 평가 말뭉치로 위키피디아 기사를 사용하는 [FineWeb2](#) 토크나이저 분석의 설정을 사용하겠습니다. 각 언어에 대해 100개의 기사를 샘플링하여 계산을 관리 가능하게 유지하면서 의미 있는 샘플을 얻습니다.

먼저 의존성을 설치하고 비교할 토크나이저와 언어를 정의합니다.

```
pip install transformers datasets sentencepiece 'datatrove[multilingual]'
```

```
## 단어 토크나이저를 로드하기 위해 datatrove가 필요합니다
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")
]

languages = [
    ("English", "eng_Latn", "en"),
    ("Chinese", "cmn_Hani", "zh"),
    ("French", "fra_Latn", "fr"),
    ("Arabic", "arb_Arab", "ar"),
]
```

이제 위키피디아 샘플을 로드합니다. 전체 데이터셋 다운로드를 피하기 위해 스트리밍을 사용합니다.

```
from datasets import load_dataset

wikis = {}
```

```

for lang_name, lang_code, short_lang_code in languages:
    wiki_ds = load_dataset("wikimedia/wikipedia", f"2023101.{short_lang_code}", streaming=True,
                           split="train")
    wiki_ds = wiki_ds.shuffle(seed=42, buffer_size=10_000)
    # 언어당 100개 기사 샘플링
    ds_iter = iter(wiki_ds)
    wikis[lang_code] = "\n".join([next(ds_iter)["text"] for _ in range(100)])

```

데이터가 준비되면, 이제 각 언어에서 각 토크나이저를 평가할 수 있습니다. 각 조합에 대해 datatrove에서 적절한 단어 토크나이저를 로드하고 두 메트릭을 계산합니다.

```

from transformers import AutoTokenizer
from datatrove.utils.word_tokenizers import load_word_tokenizer
import pandas as pd

results = []

for tokenizer_name, tokenizer_path in tokenizers:
    tokenizer = AutoTokenizer.from_pretrained(tokenizer_path, trust_remote_code=True)

    for lang_name, lang_code, short_lang_code in languages:
        word_tokenizer = load_word_tokenizer(lang_code)

        # 위키피디아에서 메트릭 계산
        fertility, pcw = compute_tokenizer_metrics(tokenizer, word_tokenizer, wikis[lang_code])

        results.append({
            "tokenizer": tokenizer_name,
            "language": lang_name,
            "fertility": fertility,
            "pcw": pcw
        })

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

결과는 우선순위에 따라 몇 가지 승자와 트레이드오프를 보여줍니다.

다산성 (단어당 토큰)

토크나이저 (어휘 크기)	영어	중국어	프랑스어	아랍어
Llama3 (128k)	1.48	1.60	1.73	2.35
Mistral Small (131k)	1.59	1.78	1.69	2.15
Qwen3 (151k)	1.54	1.45	1.75	2.26
Gemma3 (262k)	1.41	1.47	1.56	2.25

연속 단어 비율 (%)

토크나이저 (어휘 크기)	영어	중국어	프랑스어	아랍어
Llama3 (128k)	32.2%	42.6%	48.2%	71.8%
Mistral Small (131k)	36.8%	47.1%	46.5%	66.0%
Qwen3 (151k)	32.8%	30.7%	47.8%	66.0%
Gemma3 (262k)	26.0%	33.1%	39.9%	70.1%

Gemma3 토크나이저는 여러 언어에서 낮은 다산성과 단어 분할 비율을 달성하며, 특히 영어, 프랑스어, 스페인어에서 두드러집니다. 이는 토크나이저 학습 데이터와 Llama3의 128k보다 대략 2배 큰 262k의 매우 큰 어휘 크기로 설명할 수 있습니다. Qwen3 토크나이저는 중국어에서 뛰어나지만, 영어, 프랑스어, 스페인어에서는 Llama3 토크나이저보다 뒤처집니다. Mistral Small의 토크나이저([Mistral AI, 2025](#))는 아랍어에서 가장 잘 수행되지만 영어와 중국어에서는 다른 토크나이저에 미치지 못합니다.

기준 토크나이저와 커스텀 토크나이저 중 선택

현재 사용 가능한 강력한 토크나이저가 잘 선택되어 있습니다. 많은 최근 모델은 GPT4의 토크나이저([OpenAI et al., 2024](#))와 같은 것으로 시작하여 추가 다국어 토큰으로 보강합니다. 위 표에서 볼 수 있듯이, Llama 3의 토크나이저는 다국어 텍스트와 코드에서 평균적으로 잘 수행되는 반면, Qwen 2.5는 특히 중국어와 일부 저자원 언어에서 뛰어납니다.

기준 토크나이저를 사용할 때: 목표 사용 사례가 위의 최고 토크나이저(Llama, Qwen, Gemma)의 언어 또는 도메인 커버리지와 일치한다면, 이들은 실전에서 검증된 견고한 선택입니다. SmolLM3 학습을 위해 Llama3의 토크나이저를 선택했습니다. 목표 언어(영어, 프랑스어, 스페인어, 포르투갈어, 이탈리아어)에서 경쟁력 있는 토큰화 품질을 제공하며, 소형 모델 크기에 맞는 적당한 어휘 크기를 가지고 있습니다. 임베딩이 총 파라미터의 더 작은 비율인 더 큰 모델의 경우, Gemma3의 효율성이 이득이 더 매력적이 됩니다.

직접 학습시킬 때: 저자원 언어를 위해 학습하거나 매우 다른 데이터 혼합을 가지고 있다면, 좋은 커버리지를 보장하기 위해 직접 토크나이저를 학습시켜야 할 것입니다. 이 경우 최종 학습 혼합이 어떻게 될 것인지에 가까운 데이터셋에서 토크나이저를 학습시키는 것이 중요합니다. 이것은 약간의 달과 달걀 문제를 만드는데, 혼합을 찾기 위해 데이터 절제 실험을 실행하면 토크나이저가 필요하기 때문입니다. 그러나 최종 실행을 시작하기 전에 토크나이저를 재학습시키고 다운스트림 성능이 개선되고 다산성이 여전히 좋은지 확인할 수 있습니다.

토크나이저 선택이 기술적 세부 사항처럼 보일 수 있지만, 모델 성능의 모든 측면에 파급됩니다. 그러니 올바르게 하는 데 시간을 투자하는 것을 두려워하지 마세요.

SmolLM3

이제 아키텍처 환경을 탐구하고 체계적인 절제 실험을 실행했으니, SmolLM3와 같은 모델에서 이 모든 것이 실제로 어떻게 결합되는지 살펴보겠습니다.

SmolLM 패밀리는 소형 모델로 가능한 것의 한계를 밀어붙이는 것에 관한 것입니다. SmolLM2는 135M, 360M, 1.7B 파라미터의 세 가지 유능한 모델을 제공했으며, 모두 온디바이스에서 효율적으로 실행되도록 설계되었습니다. SmolLM3의 경우, 휴대폰에 맞을 만큼 충분히 작게 유지하면서 성능을 스케일업하고, SmolLM2의 약점인 다국어 지원, 매우 긴 컨텍스트 처리, 강력한 추론 능력을 해결하고 싶었습니다. 이 균형을 위한 최적 지점으로 3B 파라미터를 선택했습니다.

검증된 레시피를 스케일업하고 있었기 때문에, 자연스럽게 밀집 트랜스포머로 기울었습니다. MoE는 아직 nanotron에 구현되지 않았고, 강력한 소형 밀집 모델을 학습시키기 위한 전문 지식과 인프라가 이미 있었습니다. 더 중요한 것은, 엣지 디바이스 배포의 경우 메모리에 제약이 있다는 것입니다. 소수만 활성화되더라도 많은 파라미터를 가진 MoE는 모든 전문가를 메모리에 로드해야 하므로 제한적이며, 밀집 모델이 엣지 배포 목표에 더 실용적입니다.

절제 실험: SmolLM2 1.7B의 아키텍처를 기반으로 시작한 다음, Qwen2.5-3B 레이아웃을 사용하여 100B 토큰에서 3B 절제 실험 모델을 학습시켰습니다. 이를 통해 각 수정 사항을 개별적으로 테스트할 수 있는 견고한 기준선을 얻었습니다. 각 아키텍처 변경은 영어 벤치마크에서 손실과 다운스트림 성능을 개선하거나 품질 저하 없이 추론 속도와 같은 측정 가능한 이점을 제공해야 했습니다.

실행을 시작하기 전에 통과한 테스트 항목은 다음과 같습니다.

토크나이저: 아키텍처 수정에 들어가기 전에 토크나이저를 선택해야 했습니다. 목표 언어와 도메인을 커버하는 좋은 토크나이저 세트를 찾았습니다. 다산성 분석을 바탕으로, Llama3.2의 토크나이저가 6개 목표 언어 간의 최고의 트레이드오프를 제공하면서 어휘를 128k로 유지했습니다. 다국어 효율성에 충분히 크지만, 임베딩 가중치로 3B 파라미터 수를 부풀릴 만큼 크지 않습니다.

Grouped Query Attention(GQA): 4개 그룹을 가진 GQA가 Multi-Head Attention 성능과 일치한다는 이전 발견을 재확인했지만, 이번에는 100B 토큰으로 3B 규모에서였습니다. KV 캐시 효율성 이득은 특히 메모리가 귀중한 온디바이스 배포에서 놓치기 어려웠습니다.

긴 컨텍스트를 위한 NoPE: 4번째 레이어마다 RoPE를 제거하여 NoPE를 구현했습니다. 3B 절제 실험은 위 섹션의 발견을 확인했습니다. NoPE는 짧은 컨텍스트 성능을 희생하지 않으면서 긴 컨텍스트 처리를 개선했습니다.

문서 내 어텐션 마스킹: 매우 큰 시퀀스에서 학습할 때 학습 속도와 안정성을 둑기 위해 학습 중 문서 간 어텐션을 방지했으며, 다시 한번 이것이 다운스트림 성능에 영향을 미치지 않음을 발견했습니다.

모델 레이아웃 최적화: 문헌의 최근 3B 모델에서 레이아웃을 비교했는데, 일부는 깊이를 우선시하고 다른 일부는 너비를 우선시했습니다. 학습 설정에서 깊이와 너비가 다른 Qwen2.5-3B(3.1B), Llama3.2-3B(3.2B), Falcon3-H1-3B(3.1B) 레이아웃을 테스트했습니다. 결과는 흥미로웠습니다. 모든 레이아웃이 거의 동일한 손실과 다운스트림 성능을 달성했으며, Qwen2.5-3B가 실제로 더 적은 파라미터를 가지고 있음에도 불구하고 그랬습니다. 그러나 Qwen2.5-3B의 더 깊은 아키텍처는 네트워크 깊이가 일반화에 도움이 된다는 연구([Petty et al., 2024](#))와 일치했습니다. 따라서 학습이 진행됨에 따라 도움이 될 것이라는 기대로 더 깊은 레이아웃을 선택했습니다.

안정성 개선: SmolLM2에서 임베딩 공유를 유지했지만 OLMo2에서 영감을 받아 새로운 트릭인 임베딩에서 가중치 감쇠 제거를 추가했습니다. 절제 실험에서 이것이 성능을 해치지 않으면서 임베딩 노름을 낮추어 학습 발산 방지에 도움이 될 수 있음을 보여주었습니다.

이 체계적인 절제 실험 접근 방식의 장점은 각각이 검증되었음을 알고 이 모든 수정 사항을 자신 있게 결합할 수 있다는 것입니다.

💡 절제 실험에서 변경 사항 결합

실제로 변경 사항을 점진적으로 테스트합니다. 기능이 검증되면 다음 기능을 테스트하기 위한 기준선의 일부가 됩니다. 테스트 순서가 중요합니다. 실전에서 검증된 기능부터 시작하세요(임베딩 공유 → GQA → 문서 마스킹 → NoPE → 가중치 감쇠 제거).

참여 규칙

요약: 사용 사례가 선택을 이끕니다.

배포 목표가 아키텍처 결정을 안내하게 하세요. 새로운 아키텍처 혁신을 평가할 때 모델이 실제로 어떻게 어디서 실행될지 고려하세요.

혁신과 실용성 사이의 올바른 균형을 찾으세요. 주요 아키텍처 발전을 무시할 여유가 없습니다. GQA와 더 나은 대안이 존재 할 때 오늘날 Multi-Head Attention을 사용하는 것은 나쁜 기술적 선택일 것입니다. 최신 연구에 대해 알고 규모에서 명확하고 검증된 이점을 제공하는 기술을 채택하세요. 그러나 한계적인 이득을 약속하는 모든 새 논문을 쓰고 싶은 유혹에 저항하세요(그렇게 할 자원이 있거나 목표가 아키텍처 연구가 아니라면).

체계적인 것이 직관적인 것을 이깁니다. 논문에서 아무리 유망해 보여도 모든 아키텍처 변경을 검증하세요. 그런 다음 영향을 이해하기 위해 수정 사항을 결합하기 전에 개별적으로 테스트하세요.

스케일 효과는 실재합니다 - 가능하면 목표 크기에서 재절제 실험하세요. 소규모 절제 실험이 목표 모델 크기에서 완벽하게 유지될 것이라고 가정하지 마세요. 컴퓨팅이 있다면 재확인해 보세요.

실제 도메인에서 토크나이저 효율성을 검증하세요. 목표 언어와 도메인에 걸친 다산성 메트릭이 최신 모델이 사용한 것을 따르는 것보다 더 중요합니다. 50k 영어 토크나이저는 심각한 다국어 작업에 적합하지 않지만, 그만큼 많은 언어를 다루지 않는다면 256k 어휘도 필요하지 않습니다.

이제 모델 아키텍처가 결정되었으니, 학습 과정을 이끌 옵티마이저와 하이퍼파라미터를 다룰 시간입니다.

옵티마이저와 학습 하이퍼파라미터

조각들이 제자리를 찾아가고 있습니다. 절제 실험을 실행하고, 아키텍처를 결정하고, 토크나이저를 선택했습니다. 그러나 실제로 학습을 시작하기 전에 아직 중요한 누락된 조각이 있습니다. 어떤 옵티マイ저를 사용해야 할까요? 어떤 학습률과 배치 크기를 사용할까요? 학습 전반에 걸쳐 학습률을 어떻게 스케줄링해야 할까요?

여기서 유혹적인 접근 방식은 문헌의 다른 강력한 모델에서 값을 빌려오는 것입니다. 결국 대형 연구소에서 효과가 있었다면 우리에게도 효과가 있어야 하지 않을까요? 그리고 유사한 아키텍처와 모델 크기에서 값을 가져오는 경우 많은 경우에 잘 작동할 것입니다.

그러나 이러한 값을 특정 설정에 맞게 튜닝하지 않으면 성능을 높칠 위험이 있습니다. 문헌의 하이퍼파라미터는 특정 데이터와 제약에 최적화되었으며, 때때로 그 제약은 성능에 관한 것이 아닙니다. 아마도 그 학습률은 개발 초기에 선택되어 다시 검토되지 않았을 수 있습니다. 모델 저자가 철저한 하이퍼파라미터 스윕을 수행하더라도, 그 최적 값은 우리의 것이 아닌 그들의 정확한 아키텍처, 데이터, 학습 체제 조합에 대해 발견되었습니다. 문헌 값은 항상 좋은 출발점이지만, 인근에서 더 나은 값을 찾을 수 있는지 탐구하는 것이 좋습니다.

이 장에서는 최신 옵티マイ저를 탐구하고(신뢰할 수 있는 오래된 AdamW([Kingma, 2014](#))가 여전히 시간의 검증을 견디는지 확인([test of time](#)🎉), 표준 코사인 감쇠를 넘어서는 학습률 스케줄에 깊이 들어가고, 모델과 데이터 크기가 주어졌을 때 학습률과 배치 크기를 어떻게 튜닝하는지 알아보겠습니다.

옵티マイ저 전쟁부터 시작하겠습니다.

옵티マイ저: AdamW와 그 이후

옵티マイ저는 전체 LLM 학습 작업의 핵심입니다. 과거 업데이트, 현재 가중치, 손실에서 파생된 그래디언트를 기반으로 모든 파라미터에 대해 실제 업데이트 스텝이 무엇인지 결정합니다. 동시에 메모리와 컴퓨팅을 많이 소비하는 짐승([memory and compute hungry beast](#))이기도 해서 필요한 GPU 수와 학습 속도에 영향을 미칠 수 있습니다.

옵티マイ저가 무엇이고 무엇에 유용한지 확실하지 않다면, 특히 멋진 옵티マイ저를 비교하는 Ruder의 그래디언트 하강과 옵티マイ저에 관한 블로그를 확인하세요.

LLM 사전학습에 사용되는 옵티マイ저의 현재 환경을 요약하는 데 노력을 아끼지 않았습니다.

모델	옵티マイ저
Kimi K2, GLM 4.5	Muon
나머지 모두	AdamW

그래서 왜 모두가 AdamW를 사용하는지 궁금할 수 있습니다.

이 블로그 포스트의 이 부분을 쓰는 사람은 "사람들이 게으르기 때문"이라고 생각하지만(안녕하세요, [Elie](#)입니다), 다른 사람들은 더 현실적으로 AdamW가 오랫동안 다양한 규모에서 잘/더 잘 작동해 왔고, 특히 매우 긴 학습에서 얼마나 잘 수행되는지 테스트하기 어려운(즉, 비용이 많이 드는) 경우 그러한 핵심 구성 요소를 변경하는 것이 항상 약간 무섭다고 말할 수 있습니다.

더욱이 옵티マイ저를 공정하게 비교하는 것은 보이는 것보다 어렵습니다. 스케일은 소규모 절제 실험에서 시뮬레이션하기 어려운 방식으로 동역학을 변경하므로, 하이퍼파라미터 튜닝이 복잡합니다. "괜찮아, 내 AdamW를 몇 주 동안 튜닝했으니, 비교를 위해 동일한 하이퍼파라미터를 재사용할 수 있어!"라고 말할 수 있고, 이것이 사실이기를 정말 바랍니다. 그러나 불행히도 각 옵티マイ저에 대해 적절한 하이퍼파라미터 검색(1D? 2D? 3D?)을 수행해야 하므로, 옵티マイ저 연구가 어렵고 비용이 많이 듭니다.

종종 기준선이 잘 튜닝되지 않아서 새로운 옵티마이저가 약한 AdamW 설정과 비교됩니다. 최근 연구(Wen et al., 2025)는 그것만으로도 보고된 이득이 얼마나 왜곡되는지 보여줍니다.

그래서 고전이자 Durk Kingma의 무서운 [Google scholar](#) 지배의 기반인 AdamW부터 시작하겠습니다.

AdamW

Adam(Adaptive Momentum Estimation)은 1차 최적화 기술입니다. 그래디언트만 보는 것 외에도 이전 스텝에서 가중치가 얼마나 변했는지도 고려합니다. 이로 인해 각 파라미터의 학습률이 모멘텀을 기반으로 적용합니다.

주의 깊은 독자는 궁금해할 수 있습니다. 거기서 W가 빠지지 않았나요? 맞습니다! 우리가 특별히 W(=가중치 감쇠)를 추가하는 이유는 다음과 같습니다. 표준 SGD에서는 L2 정규화를 적용하기 위해 손실에 $\lambda\theta^2$ (여기서 θ 는 가중치)를 간단히 추가할 수 있습니다. 그러나 Adam에서 동일하게 하면, 적응형 학습률이 L2 정규화에도 영향을 미칩니다. 이는 정규화 강도가 그래디언트 크기에 의존하게 되어 효과가 약해짐을 의미합니다. 이것은 우리가 원하는 것이 아니며, 그래서 AdamW는 이를 수정하기 위해 주요 최적화 루프에서 분리하여 적용합니다.

흥미롭게도 지난 몇 년 동안 AdamW 하이퍼파라미터는 거의 변하지 않았습니다.

- $\beta_1 = 0.9, \beta_2 = 0.95$
- grad norm clipping = 1.0
- weight decay = 0.1 (Llama-3-405B는 이를 0.01로 낮춤)

동일한 삼중 값이 Llama 1,2,3에서 DeepSeek-V1,2,3 671B까지 거의 재사용되며, 변경이 없습니다. Durk Kingma가 처음부터 옮았던 것일까요, 아니면 더 잘할 수 있을까요?

Muon 한 줄 요약

Adam은 그래디언트만 사용하는 1차 방법입니다. Muon은 파라미터 텐서의 행렬 뷰에서 작동하는 2차 옵티마이저입니다.

$$\begin{aligned} G_t &= \nabla_{\theta} L_t(\theta_{t-1}) \\ B_t &= \mu B_{t-1} + G_t \\ O_t &= \text{NewtonSchulz5}(B_t) \approx UV^\top \text{ if } B_t = U\Sigma V^\top \text{ (SVD)} \\ \theta_t &= \theta_{t-1} - \eta O_t \end{aligned}$$

이 방정식을 보면서 왜 이것이 2차 방법인지, 그래디언트만 보이고 고차 항은 보이지 않는지 궁금할 수 있습니다. 2차 최적화는 실제로 Newton Schulz 스텝 내부에서 발생하지만, 여기서는 더 자세히 다루지 않겠습니다. Muon을 깊이 설명하는 고품질 블로그가 이미 있으므로, 여기서는 Muon의 세 가지 핵심 아이디어만 나열하겠습니다.

Muon에 대해 더 알고 싶다면, Keller Jordan의 이 블로그, Jeremy Bernstein의 이것, 그리고 좋은 출발점인 Jia-Bin Huang의 이 비디오를 확인하는 것을 추천합니다.

파라미터별 업데이트 vs 행렬별 기하학: AdamW는 파라미터별로 사전 조건화합니다(대각 2차 모멘트). Muon은 각 가중치 행렬을 단일 객체로 취급하고 행/열 부분공간 구조를 포착하는 $G = UV^\top$ 을 따라 업데이트합니다.

직교화를 통한 등방성 스텝: $G = UV^\top$ 을 특이값 분해(SVD)로 분해하면 크기(Σ)를 방향(좌/우 부분공간 U, V)에서 분리합니다. G 를 UV^\top 으로 대체하면 특이값을 버리고 활성 부분공간에서 스텝을 등방성으로 만듭니다. 처음에는 약간 직

관에 반하는데, Σ 를 버리는 것이 정보를 잃는 것처럼 보이기 때문입니다. 그러나 이는 축 정렬 편향을 줄이고 매우 작은 특이값에 의해 억제되었을 방향의 탐구를 장려합니다. 이러한 종류의 탐구가 손실만 보면 명확하지 않은 다른 능력을 모델에 베이크하는지에 대한 열린 질문이 여전히 있습니다.

더 큰 배치 크기에 대한 경험적 허용: 실제로 Muon은 종종 더 높은 배치 크기를 허용합니다. 배치 크기 섹션에서 이에 대해 더 깊이 이야기하겠지만, 이것이 Muon 채택의 핵심 포인트일 수 있습니다!

수년간 커뮤니티는 대부분 AdamW에 정착했고 프론티어 연구소의 옵티마이저 레시피는 종종 비밀로 유지되지만(예를 들어 Qwen은 자신들의 것에 대해 말하지 않음), 최근 Muon이 고프로필 릴리스(예: Kimi K2, GLM-4.5)에서 채택되는 것을 보고 있습니다. 바라건대 더 많은 개방적이고 강력한 레시피를 볼 수 있기를

옵티마이저의 야생 동물원이 있고, 연구자들이 가능한 모든 모멘텀과 도함수를 결합하는 것보다 더 창의적인 유일한 것은 이름을 짓는 것입니다: Shampoo, SOAP, PSGD, CASPR, DION, Sophia, Lion... AdamW조차 NAdamW, StableAdamW 등과 같은 자체 변형이 있습니다. 이 모든 옵티마이저에 대해 다루는 것은 자체 블로그의 가치가 있지만, 다음을 위해 남겨두겠습니다. 그동안 하이퍼파라미터 튜닝이 비교를 할 때 얼마나 중요한지 보여주기 위해 많은 다른 옵티마이저를 벤치마킹한 stanford/marin 팀의 놀라운 논문([Wen et al., 2025](#))을 추천합니다.

거의 모든 옵티마이저와 함께하는 것은 일반적으로 옵티마이저 방정식에서 스칼라 값으로 나타나는 학습률에 의해 결정되는 가중치를 얼마나 강하게 업데이트해야 하는지에 대한 질문입니다. 이 겉보기에 단순한 주제가 여전히 많은 측면을 가지고 있는지 살펴보겠습니다.

학습률

학습률은 우리가 설정해야 할 가장 중요한 하이퍼파라미터 중 하나입니다. 각 학습 스텝에서 계산된 그래디언트를 기반으로 모델 가중치를 얼마나 조정하는지 제어합니다. 학습률을 너무 낮게 선택하면 학습이 고통스럽게 느려지고 나쁜 지역 최솟값에 갇힐 수 있습니다. 손실 곡선이 평평해 보이고, 의미 있는 진전 없이 컴퓨팅 예산을 소진하게 됩니다. 반면에 학습률을 너무 높게 설정하면 옵티마이저가 최적의 솔루션을 지나치는 거대한 스텝을 취하여 절대 수렴하지 않거나, 상상할 수 없는 일이 일어나서 손실이 발산하고 하늘로 치솟습니다.

그러나 최고의 학습률도 학습 동역학이 학습 중에 변하기 때문에 일정하지 않습니다. 높은 학습률은 좋은 솔루션에서 멀 때 초기에 잘 작동하지만 수렴 근처에서 불안정을 유발합니다. 여기서 학습률 스케줄이 등장합니다. 초기 혼란을 피하기 위해 0에서 월업하고, 좋은 최솟값에 정착하기 위해 감쇠합니다. 이러한 패턴(예: 월업 + 코사인 감쇠)은 수년간 신경망 학습에 대해 검증되었습니다.

💡 월업 스텝

[표 1](#)에서 보여듯이, 대부분의 현대 LLM은 모델 크기와 학습 길이에 관계없이 고정된 수의 월업 스텝(예: 2000)을 사용합니다. 긴 학습의 경우 월업 스텝 수를 늘려도 성능에 영향을 미치지 않지만, 매우 짧은 학습의 경우 사람들은 보통 학습 스텝의 1%에서 5%를 사용합니다.

일반적인 스케줄을 살펴본 다음, 피크 값을 선택하는 방법에 대해 논의하겠습니다.

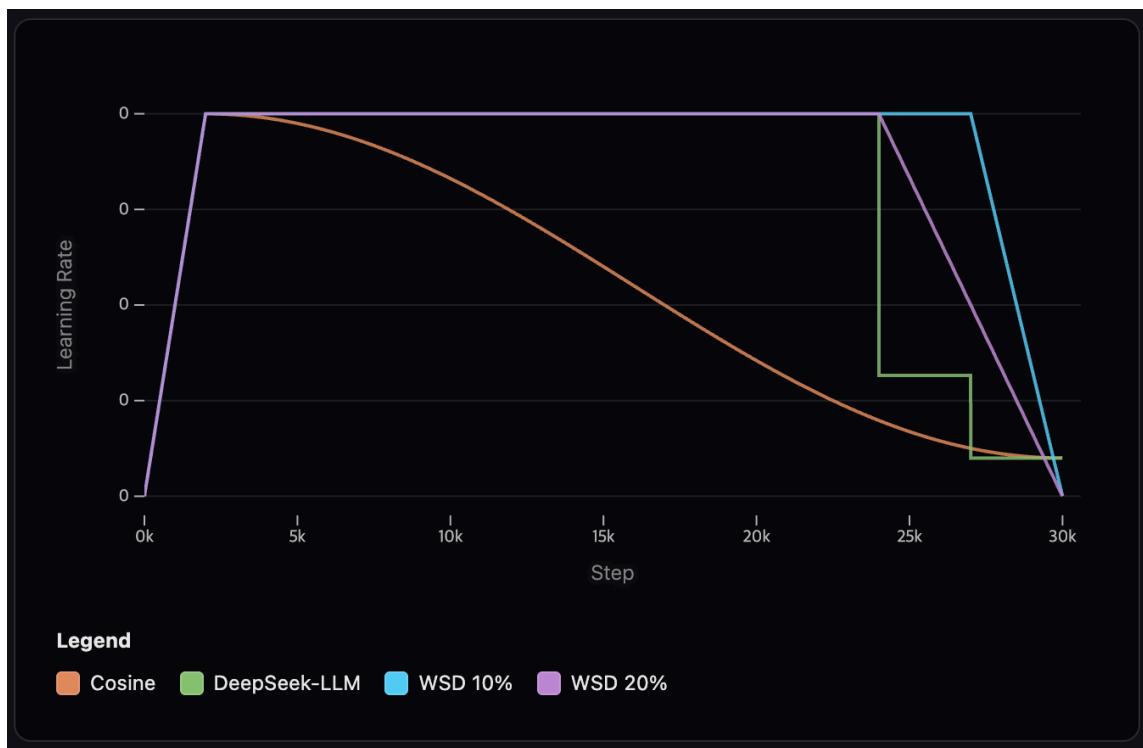
학습률 스케줄: 코사인 감쇠를 넘어서

학습률을 변경하는 것이 수렴에 도움이 된다는 것은 수년 동안 알려져 왔고([Smith & Topin, 2018](#)), 코사인 감쇠([Loshchilov & Hutter, 2017](#))는 LLM 학습을 위한 기본 스케줄이었습니다. 월업 후 피크 학습률에서 시작한 다음, 코사인

곡선을 따라 부드럽게 감소합니다. 간단하고 잘 작동합니다. 그러나 주요 단점은 유연하지 않다는 것입니다. 코사인 사이클 길이가 총 학습 기간과 일치해야 하므로 총 학습 스텝을 미리 알아야 합니다. 이것은 일반적인 시나리오에서 문제가 됩니다. 모델이 아직 정체되지 않았거나, 더 많은 컴퓨팅에 접근할 수 있어서 더 오래 학습하고 싶거나, 스케일링 법칙을 실행하고 있어서 동일한 모델을 다른 토큰 수에서 학습시켜야 합니다. 코사인 감쇠는 처음부터 다시 시작해야 합니다.

흥미롭게도 이 유연성 부족은 초기 스케일링 법칙 연구(Kaplan et al., 2020)를 왜곡했는데, 다양한 토큰 수에서 모델을 학습시킬 때 고정된 코사인 스케줄 길이를 사용하여 데이터 크기의 영향을 과소평가했기 때문입니다. Chinchilla 연구 (Hoffmann et al., 2022)는 이를 수정하고 스케줄 길이를 각 모델의 실제 학습 기간에 맞췄습니다.

많은 팀이 이제 웜업 후 바로 감쇠를 시작할 필요가 없는 스케줄을 사용합니다. 아래 플롯에 표시된 Warmup-Stable-Decay(WSD)(Hu et al., 2024)와 Multi-Step(DeepSeek-AI, :, et al., 2024) 변형이 이에 해당합니다. 대부분의 학습 동안 일정한 높은 학습률을 유지하고, WSD의 경우 최종 단계(일반적으로 마지막 10-20% 토큰)에서 급격히 감쇠하거나, Multi-Step의 경우 학습률을 낮추기 위해 이산적인 드롭(스텝)을 수행합니다. 예를 들어 DeepSeek LLM의 Multi-Step 스케줄에서처럼 학습의 80% 후에 그리고 90% 후에 수행됩니다.



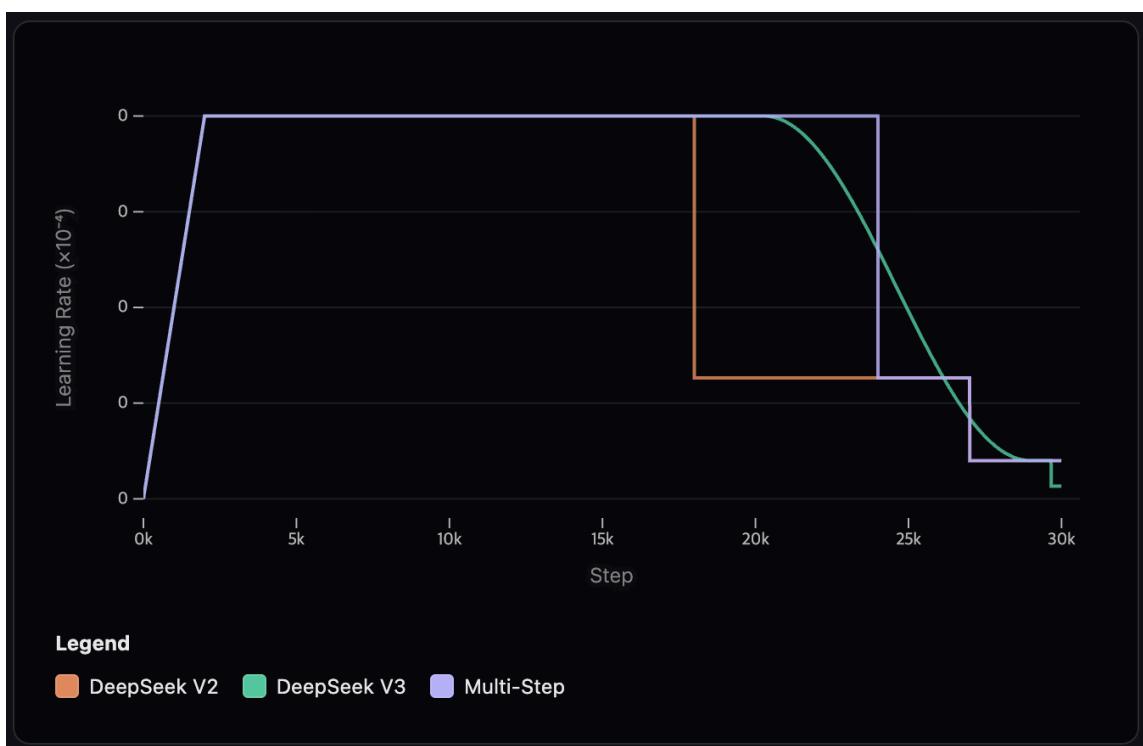
이러한 스케줄은 코사인 감쇠에 비해 실용적인 이점을 제공합니다. 처음 계획한 것보다 더 오래 학습하고 싶거나, 학습 진행 상황을 더 잘 측정하기 위해 일찍 감쇠하거나, 하나의 메인 학습 실행으로 다른 토큰 수에 걸쳐 스케일링 법칙 실험을 실행하든, 재시작 없이 학습 중간에 학습을 확장할 수 있습니다. 더욱이 연구에 따르면 WSD와 Multi-Step 모두 코사인 감쇠와 일치하면서(DeepSeek-AI, :, et al., 2024; Hägele et al., 2024) 실제 학습 시나리오에서 더 실용적입니다.

최근 GLM 4.5는 WSD가 일반 벤치마크(SimpleQA, MMLU)에서 더 나쁜 성능을 보인다고 언급하지만, 어떤 결과도 제공하지 않습니다.

하지만 이러한 스케줄이 코사인에 비해 새로운 하이퍼파라미터를 도입한다는 것을 눈치챘을 것입니다. WSD에서 감쇠 단계는 얼마나 오래 지속되어야 할까요? Multi-Step 변형에서 각 스텝은 얼마나 길어야 할까요?

- **WSD의 경우:** 코사인 성능과 일치하는 데 필요한 출다운 기간은 더 긴 학습 실행에서 감소하며, 총 토큰의 10-20%를 감쇠 단계에 할당하는 것이 권장됩니다([Hägele et al., 2024](#)). 아래 절제 실험에서 이 설정이 코사인과 일치하는지 확인할 것입니다.
- **Multi-Step의 경우:** DeepSeek LLM의 절제 실험에서 기준 80/10/10 분할(80%까지 안정, 80-90%에서 첫 번째 스텝, 90-100%에서 두 번째 스텝)이 코사인과 일치하지만, 이러한 비율을 조정하면 심지어 능가할 수 있음을 발견했습니다. 예를 들어 70/15/15와 60/20/20 분할을 사용할 때 그렇습니다.

그러나 이러한 스케줄로 훨씬 더 창의적일 수 있습니다. DeepSeek 모델의 각 패밀리에서 사용된 스케줄을 살펴보겠습니다.



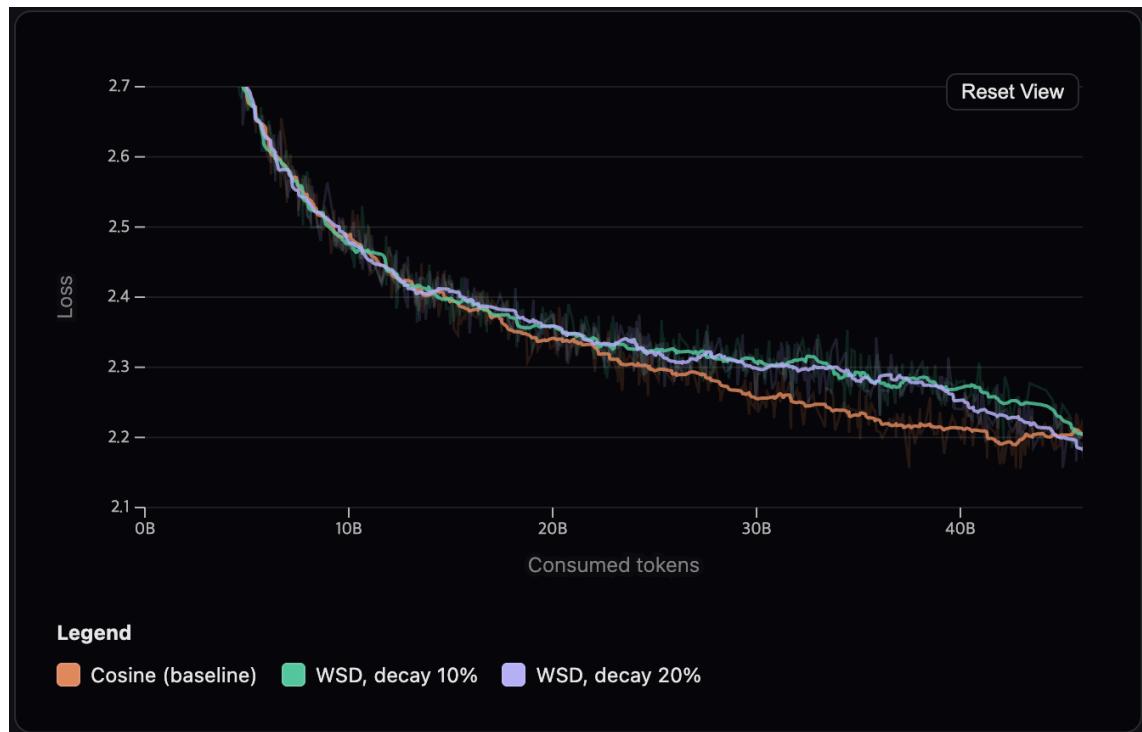
DeepSeek LLM은 기준 Multi-Step 스케줄(80/10/10)을 사용했습니다. [DeepSeek V2](#)는 비율을 60/30/10으로 조정하여 첫 번째 감쇠 스텝에 더 많은 시간을 부여했습니다. [DeepSeek V3](#)는 가장 창의적인 접근 방식을 취했습니다. 일정한 학습률 후 두 번의 급격한 스텝을 유지하는 대신, 일정한 단계에서 코사인 감쇠로 전환하고(학습의 67%에서 97%까지), 최종 급격한 스텝 전에 짧은 일정한 단계를 적용합니다.

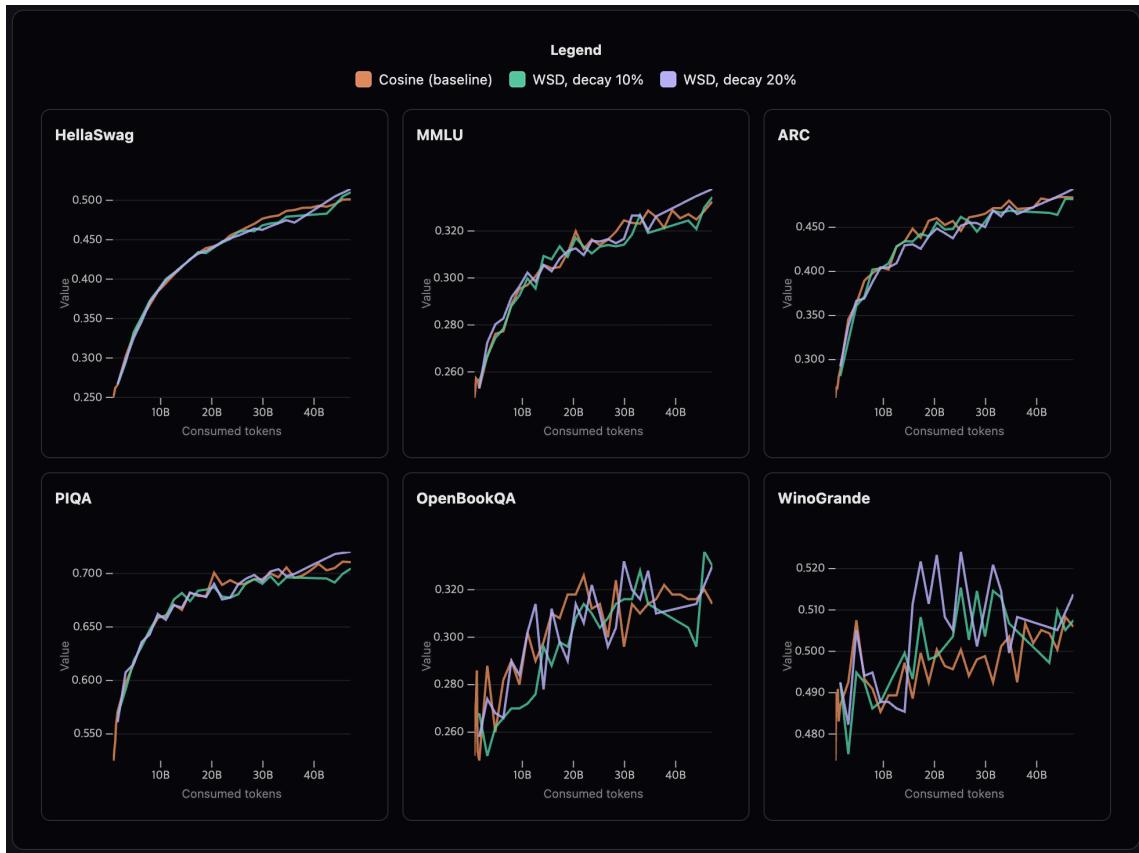
DeepSeek 스케줄 변경 DeepSeek-V2와 V3의 기술 보고서에는 이러한 스케줄 변경에 대한 절제 실험이 포함되어 있지 않습니다. 여러분의 설정에서는 간단한 WSD 또는 Multi-Step 스케줄로 시작한 다음, 절제 실험을 통해 파라미터를 튜닝하는 것을 고려하세요.

이국적인 학습률 스케줄 조사는 여기서 멈추고, 실제로 무엇이 효과가 있는지 확인하기 위해 GPU 시간을 좀 태워봅시다!

Ablation - WSD가 코사인과 일치함

이제 절제 실험 시간입니다! WSD가 실제로 실전에서 코사인의 성능과 일치하는지 테스트해 보겠습니다. 여기서는 Multi-Step 절제 실험을 보여주지 않지만, Multi-Step이 다양한 단계 분할로 코사인과 일치함을 보여준 DeepSeek LLM의 절제 실험을 추천합니다. 이 섹션에서는 코사인 감쇠와 두 가지 감쇠 윈도우(10%와 20%)를 가진 WSD를 비교할 것입니다.





평가 결과는 세 가지 구성 모두에서 유사한 최종 성능을 보여줍니다. 손실과 평가 곡선(특히 HellaSwag)을 보면, 흥미로운 패턴을 볼 수 있습니다. 코사인은 안정 단계(WSD의 감쇠가 시작되기 전) 동안 더 나은 손실과 평가 점수를 달성합니다. 그러나 WSD가 감쇠 단계에 들어가면, 손실과 다운스트림 메트릭 모두에서 거의 선형적인 개선이 있어 WSD가 학습 끝까지 코사인을 따라잡을 수 있습니다.

이것은 WSD의 10-20% 감쇠 윈도우가 학습 중간에 확장할 수 있는 유연성을 유지하면서 코사인의 최종 성능과 일치하기에 충분하다는 것을 확인합니다. SmoLM3에서는 10% 감쇠를 가진 WSD를 선택했습니다.

⚠ 다른 스케줄러로 학습된 모델을 중간에 비교할 때 안정 단계에서 코사인과 WSD 사이의 중간 체크포인트를 비교하는 경우, 공정한 비교를 위해 WSD 체크포인트에 감쇠를 적용해야 합니다.

이제 인기 있는 학습률 스케줄에 대한 좋은 개요를 얻었으니, 다음 질문은 피크 학습률이 실제로 얼마여야 하는가입니다.

최적의 학습률 찾기

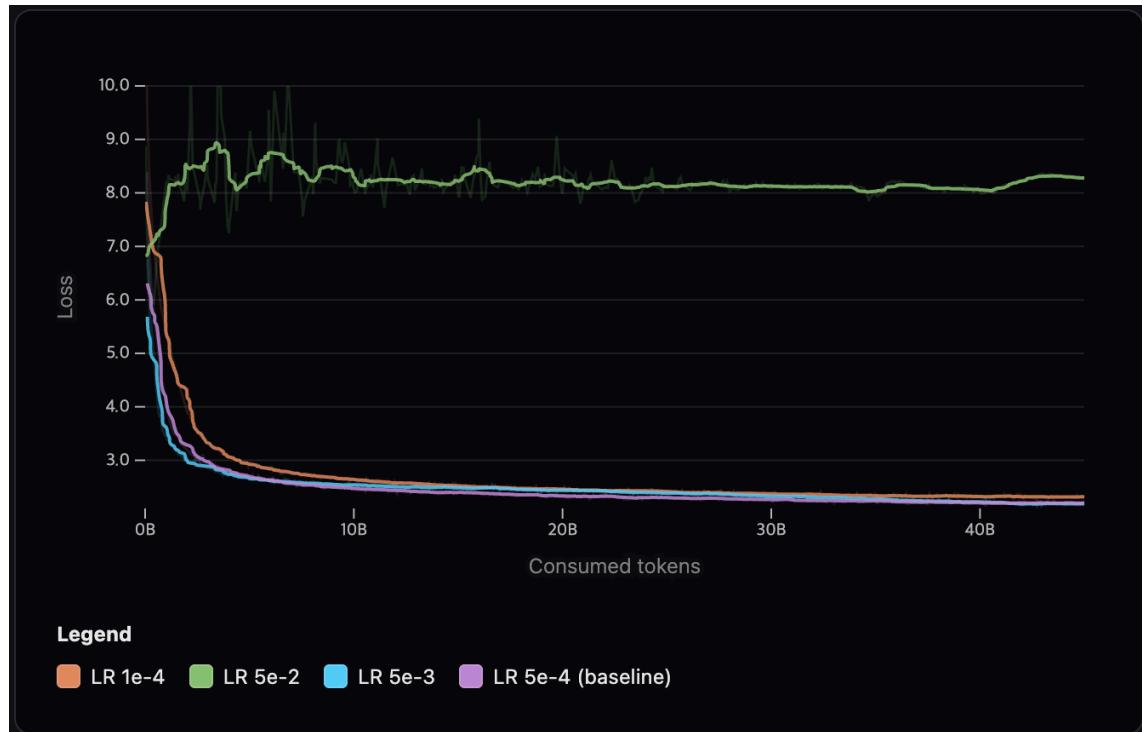
특정 학습률 스케줄러와 학습 설정에 맞는 올바른 학습률을 어떻게 선택할까요?

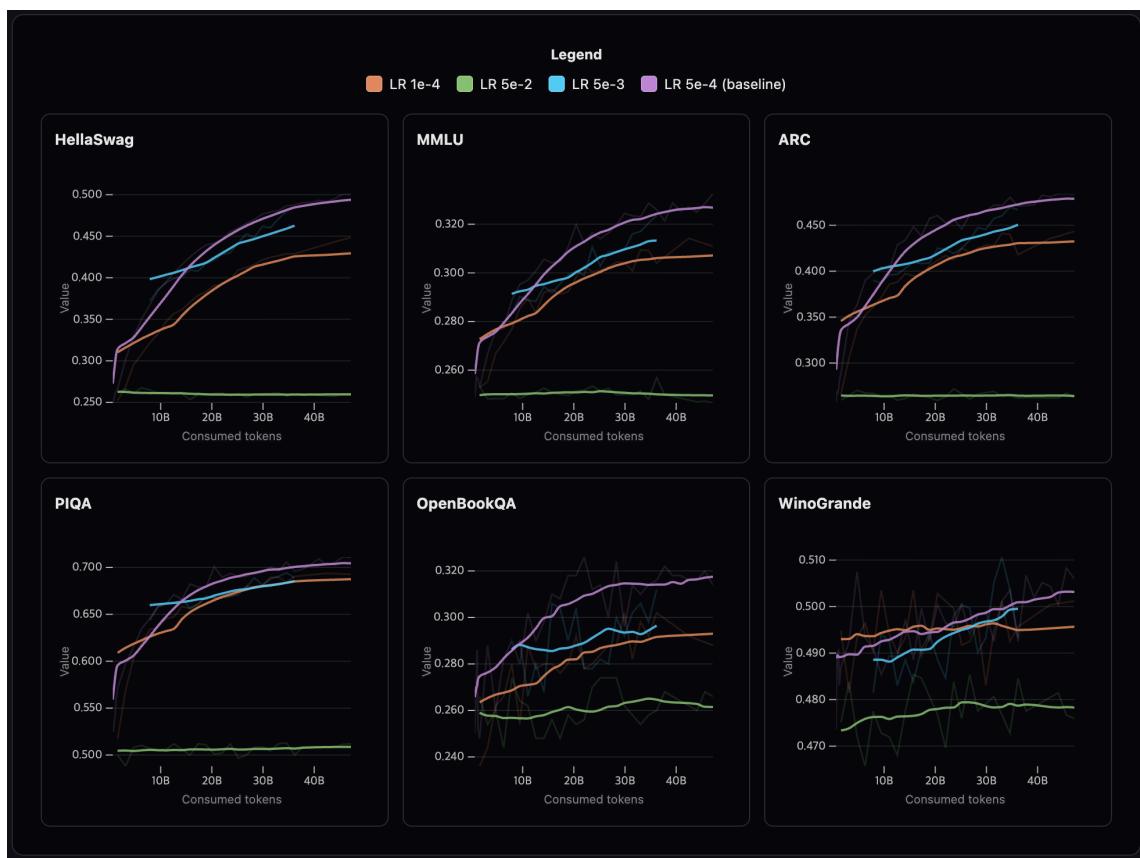
아키텍처 선택에서 했던 것처럼 짧은 절제 실험에서 학습률 스윕을 실행할 수 있습니다. 그러나 최적의 학습률은 학습 기간에 따라 달라집니다. 짧은 절제 실험에서 가장 빨리 수렴하는 학습률이 전체 실행에 가장 좋은 것이 아닐 수 있습니다. 그리고 다른 학습률을 테스트하기 위해 비용이 많이 드는 몇 주짜리 학습을 여러 번 실행할 여유가 없습니다.

먼저 너무 높거나 낮은 학습률을 배제하는 데 도움이 되는 빠르게 실행할 수 있는 간단한 스윕을 살펴본 다음, 하이퍼파라미터에 대한 스케일링 법칙에 대해 논의하겠습니다.

Ablation - LR 스윕

다양한 학습률의 영향을 설명하기 위해, 45B 토큰에서 학습된 1B 절제 실험 모델에 대한 스윕을 살펴보겠습니다. 동일한 설정에서 4가지 다른 학습률($1e-4$, $5e-4$, $5e-3$, $5e-2$)로 동일한 모델을 학습시킵니다. 결과는 양 극단에서의 위험을 명확하게 보여줍니다.





LR 5e-2는 거의 즉시 발산하며, 손실이 초기에 스파이크하고 절대 회복되지 않아 모델을 사용할 수 없게 만듭니다. LR 1e-4는 너무 보수적입니다. 안정적으로 학습되지만 다른 학습률보다 훨씬 더 느리게 수렴합니다. 5e-4와 5e-3의 중간 지점은 더 나은 수렴과 비슷한 성능을 보여줍니다. 하지만 모든 모델 크기에 대해 스윕을 실행하면 비용이 빠르게 늘어나고, 더 중요한 것은 앞서 언급했듯이 계획된 학습 토큰 수를 고려하지 않는다는 것입니다. 여기서 스케일링 법칙이 매우 유용해집니다.

SmoILM3의 경우, WSD 스케줄을 사용하여 AdamW로 100B 토큰에서 3B 모델을 학습시키며 여러 학습률을 비교했습니다. 2e-4가 손실과 다운스트림 성능 모두에서 1e-4보다 훨씬 빠르게 수렴했고, 3e-4는 2e-4보다 약간만 더 나았습니다. 3e-4에서의 한계적 이득은 긴 학습 실행 중 불안정성의 위험 증가와 함께 왔으므로, 2e-4를 최적 지점으로 선택했습니다.

이러한 스윕은 명확히 너무 높은(발산) 또는 너무 낮은(느린 수렴) 학습률을 배제하는 데 도움이 되지만, 모든 모델 크기에 대해 스윕을 실행하면 비용이 빠르게 늘어나고, 더 중요한 것은 앞서 언급했듯이 계획된 학습 토큰 수를 고려하지 않는다는 것입니다. 여기서 스케일링 법칙이 매우 유용해집니다.

하지만 하이퍼파라미터에 대한 스케일링 법칙에 들어가기 전에, 학습률과 상호작용하는 또 다른 중요한 하이퍼파라미터인 배치 크기에 대해 논의하겠습니다.

배치 크기

배치 크기는 모델 가중치를 업데이트하기 전에 처리되는 샘플 수입니다. 학습 효율성과 최종 모델 성능 모두에 직접적인 영향을 미칩니다. 배치 크기를 늘리면 하드웨어와 학습 스택이 디바이스 전반에 걸쳐 잘 스케일링되는 경우 처리량이 향상됩니다.

다. 그러나 특정 지점을 넘으면 더 큰 배치가 데이터 효율성을 해치기 시작합니다. 모델이 동일한 손실에 도달하기 위해 더 많은 총 토큰이 필요합니다. 이것이 발생하는 분기점을 임계 배치 크기(critical batch size)(McCandlish et al., 2018).라고 합니다.

처리량(Throughput)은 학습 중 초당 처리되는 토큰 수입니다.

임계 이하에서 배치 크기 증가: 배치 크기를 늘리고 학습률을 재튜닝한 후, 더 작은 배치 크기 실행과 동일한 토큰 수로 동일한 손실에 도달합니다. 데이터가 낭비되지 않습니다.

임계 이상에서 배치 크기 증가: 더 큰 배치가 데이터 효율성을 희생하기 시작합니다. 동일한 손실에 도달하려면 이제 더 많은 총 토큰(따라서 더 많은 비용)이 필요합니다. 더 많은 칩이 바쁘기 때문에 실제 경과 시간이 줄어들더라도 마찬가지입니다.

왜 학습률을 재튜닝해야 하는지에 대한 직관과 임계 배치 크기가 어떻게 되어야 하는지에 대한 추정을 계산하는 방법을 제공해 보겠습니다.

배치 크기가 커지면, 각 미니배치 그래디언트는 실제 그래디언트의 더 나은 추정이 되므로, 안전하게 더 큰 스텝을 취할 수 있고(즉, 학습률을 높이고) 더 적은 업데이트로 목표 손실에 도달할 수 있습니다. 질문은 어떻게 스케일링하느냐입니다.

B개 샘플에 대한 평균

배치 그래디언트:

$$\tilde{g}_B = \frac{1}{B} \sum_{i=1}^B \tilde{g}^{(i)}$$

평균은 동일하게 유지: $E[\tilde{g}_B] = g$

그러나 공분산은 축소: $\text{Cov}(\tilde{g}_B) = \frac{\Sigma}{B}$

SGD 파라미터 업데이트는:

$$\Delta w = -\eta \tilde{g}_B$$

이 업데이트의 분산은 다음에 비례합니다:

$$\text{Var}(\Delta w) \propto \eta^2 \frac{\Sigma}{B}$$

따라서 업데이트 분산을 대략 일정하게 유지하려면, 배치 크기를 k배 스케일링하면 학습률을 \sqrt{k} 배 스케일링해야 합니다. 따라서 최적의 배치 크기와 학습률을 계산했고 임계 배치 크기까지 증가하는 것이 가능하고 처리량을 높인다는 것을 발견했다면, 최적의 학습률도 조정해야 합니다.

$$B_{\text{critical}} \rightarrow kB_{\text{optimal}} \Rightarrow \eta_{\text{critical}} \rightarrow \sqrt{k}\eta_{\text{optimal}}$$

이에 대한 더 많은 수학은 (놀라운) Jianlin Su의 시리즈를 참조하세요: <https://kexue.fm/archives/11260>

AdamW나 Muon과 같은 옵티마이저에 대한 유용한 경험 법칙은 배치 크기가 커짐에 따라 제곱근 LR 스케일링이지만, 이것은 옵티마이저에 따라서도 달라집니다. 예를 들어 AdamW를 사용하면 매우 다른 동작을 도입할 수 있는 beta1/beta2와의 상호작용이 있습니다. 실용적인 대안은 짧은 기간 동안 학습률을 분기하는 것입니다. 원래 배치에서 하나의 실행을 유지하고, 더 큰 배치와 재스케일링된 LR로 두 번째를 시작하고, 재스케일링 후 두 손실 곡선이 정렬되는 경우에만 더 큰 배치를 채

택합니다([Merrill et al., 2025](#)). 논문에서 그들은 배치 크기를 전환할 때 학습률을 다시 웜업하고 옵티마이저 상태를 리셋 합니다. 또한 손실이 "일치"하는지 결정하기 위한 허용 오차와 시간 윈도우를 설정하며, 두 노브 모두 경험적으로 선택됩니다. 그들은 B_{simple} 추정치가 - 이것도 노이즈가 있지만 - "실제" 임계 배치 크기를 과소평가하고 있음을 발견했습니다. 이를 통해 새로운 배치/LR 쌍이 학습 동역학을 보존하는지 빠르고 저위험으로 확인할 수 있습니다.

임계 배치 크기는 고정되어 있지 않으며, 학습이 진행됨에 따라 커집니다. 학습 초기에 모델은 큰 그래디언트 스텝을 취하므로, $|g|^2$ 가 크고 이는 B_{simple} 이 작다는 것을 의미하며, 따라서 모델은 더 작은 임계 배치 크기를 가집니다. 나중에 모델 업데이트가 안정화되면 더 큰 배치가 더 효과적이 됩니다. 이것이 일부 대규모 학습이 배치 크기를 일정하게 유지하지 않고 배치 크기 웜업이라고 부르는 것을 사용하는 이유입니다. 예를 들어, DeepSeek-V3는 처음 ~469B 토큰 동안 12.6M 배치로 시작한 다음, 나머지 학습 동안 62.9M으로 증가합니다. 이와 같은 배치 크기 웜업 스케줄은 학습률을 웜업과 동일한 목적을 수행합니다. 그래디언트 노이즈 스케일이 커짐에 따라 모델을 효율적인 프론티어에 유지하여 전체에 걸쳐 안정적이고 효율적인 최적화를 유지합니다.

또 다른 흥미로운 접근 방식은 손실을 임계 배치 크기의 프록시로 취급하는 것입니다. Minimax01이 이것을 사용했고 마지막 단계에서 128M 배치 크기로 학습했습니다! 이것은 학습률을 높이지 않기 때문에 약간 다르며, 그들의 배치 크기 스케줄은 학습률 감쇠 스케줄처럼 작동합니다.

배치 크기와 학습률 튜닝

실제로 배치 크기와 학습률을 선택하는 방법은 다음과 같습니다.

1. 먼저 최적이라고 생각하는 배치 크기와 학습률을 스케일링 법칙(나중에 설명!)이나 문헌에서 선택합니다.
2. 그런 다음 배치 크기를 튜닝하여 학습 처리량을 개선할 수 있는지 확인합니다.

핵심 통찰은 시작 배치 크기와 임계 배치 크기 사이에 종종 범위가 있어서 데이터 효율성을 희생하지 않고 하드웨어 활용도를 개선하기 위해 증가시킬 수 있지만, 그에 따라 학습률을 재튜닝해야 한다는 것입니다. 처리량 이득이 크지 않거나, 더 큰 배치 크기(재스케일링된 학습률과 함께)를 테스트하면 더 나쁜 데이터 효율성을 보면, 초기 값을 유지하세요.

위의 노트에서 언급했듯이, 배치 크기와 학습률의 시작점을 선택하는 한 가지 방법은 스케일링 법칙을 통해서입니다. 이러한 스케일링 법칙이 어떻게 작동하고 컴퓨팅 예산의 함수로 두 하이퍼파라미터를 어떻게 예측하는지 살펴보겠습니다.

하이퍼파라미터에 대한 스케일링 법칙

최적의 학습률과 배치 크기는 모델 아키텍처와 크기에만 관한 것이 아니라, 모델 파라미터 수와 학습 토큰 수를 결합한 컴퓨팅 예산에도 따라 달라집니다. 실제로 이 두 요소 모두가 상호작용하여 업데이트가 얼마나 공격적이거나 보수적이어야 하는지 결정합니다. 여기서 스케일링 법칙이 등장합니다.

스케일링 법칙은 더 큰 모델이든 더 많은 학습 데이터든 학습 규모를 늘림에 따라 모델 성능이 어떻게 발전하는지 설명하는 경험적 관계를 확립합니다(전체 역사는 이 장 끝의 "스케일링 법칙" 섹션 참조). 그러나 스케일링 법칙은 [DeepSeek](#)와 [Qwen2.5](#)의 최근 연구에서 수행된 것처럼 학습률 스케일업함에 따라 학습률과 배치 크기와 같은 주요 하이퍼파라미터를 어떻게 조정하는지 예측하는 데도 도움이 될 수 있습니다. 이를 통해 전적으로 하이퍼파라미터 스윕에 의존하는 대신 원칙적인 기본값을 얻을 수 있습니다.

이 맥락에서 스케일링 법칙을 적용하려면 학습 규모를 정량화하는 방법이 필요합니다. 표준 메트릭은 FLOPs로 측정되는 컴퓨팅 예산 C이며, 다음과 같이 근사할 수 있습니다.

$$C \approx 6 \times N \times D$$

N 은 모델 파라미터 수(예: $1B = 1e9$)이고, D 는 학습 토큰 수입니다. 이것은 종종 얼마나 많은 실제 계산이 수행되는지 정량화하는 하드웨어에 독립적인 방법인 FLOPs(부동 소수점 연산)로 측정됩니다. 그러나 FLOPs가 너무 추상적으로 느껴진다면, 이렇게 생각하세요. $100B$ 토큰에서 $1B$ 파라미터 모델을 학습시키면 $100B$ 토큰에서 $2B$ 모델을 학습시키거나 $200B$ 토큰에서 $1B$ 모델을 학습시키는 것보다 약 2배 적은 FLOPs를 소비합니다.

상수 6은 트랜스포머를 학습시키는 데 필요한 부동 소수점 연산 수에 대한 경험적 추정에서 나오며, 파라미터당 토큰당 대략 6 FLOPs입니다.

MoE 레이어와 하이브리드 레이어를 고려한 더 정확한 측정을 원한다면 Megatron-LM의 num_floating_point_operations 함수를 확인할 수 있습니다.

이제 이것이 학습률과 어떻게 관련될까요? 총 컴퓨팅 예산(C)의 함수로 최적의 학습률과 배치 크기를 예측하는 스케일링 법칙을 도출할 수 있습니다. 이들은 다음과 같은 질문에 답하는 데 도움이 됩니다.

- $1B$ 에서 $7B$ 파라미터로 스케일업할 때 학습률이 어떻게 변해야 하나요?
- 학습 데이터를 두 배로 늘리면 학습률을 조정해야 하나요?

DeepSeek가 사용한 접근 방식을 살펴보며 이것이 어떻게 작동하는지 보겠습니다. 먼저 학습률 스케줄을 선택하며, 유연성을 위해 이상적으로 WSD를 선택합니다. 그런 다음 다양한 컴퓨팅 예산(예: $1e17, 5e17, 1e18, 5e18, 1e19, 2e19$ FLOPs)에 걸쳐 다양한 배치 크기와 학습률 조합으로 모델을 학습시킵니다. 더 간단한 용어로: 다른 하이퍼파라미터 설정을 테스트하면서 다른 모델 크기를 다른 토큰 수로 학습시킵니다. WSD 스케줄이 빛나는 곳이 여기입니다. 재시작 없이 동일한 학습 실행을 다른 토큰 수로 확장할 수 있습니다.

각 설정에 대해 학습률과 배치 크기에 대한 스윕을 수행하고 거의 최적의 성능을 달성하는 구성을 식별합니다. 일반적으로 최고 검증 손실(학습 세트와 유사한 분포를 가진 독립적인 검증 세트에서 계산)의 작은 마진(예: 0.25%) 내로 정의됩니다. 각 거의 최적의 구성은 데이터 포인트를 제공합니다. ($\text{컴퓨팅 예산 } C$, 최적 학습률 η) 또는 (C , 최적 배치 크기 B)의 튜플입니다. 로그-로그 스케일에 플롯하면, 이러한 관계는 일반적으로 거듭제곱 법칙 동작을 따르며, 대략 직선으로 나타납니다(위 그림에서 보여주듯이). 이 데이터 포인트를 피팅하면, 최적의 하이퍼파라미터가 컴퓨팅과 함께 어떻게 발전하는지 설명하는 스케일링 법칙을 추출할 수 있습니다.

이 과정에서 중요한 발견은 고정된 모델 크기와 컴퓨팅 예산에 대해 성능이 넓은 범위의 하이퍼파라미터에 걸쳐 안정적으로 유지된다는 것입니다. 이것은 좁은 최적값이 아닌 넓은 최적 지점이 있다는 것을 의미합니다. 완벽한 값을 찾을 필요가 없고, 충분히 가까운 값만 찾으면 됩니다. 이것이 전체 과정을 훨씬 더 실용적으로 만듭니다.

여기서 DeepSeek가 도출한 스케일링 법칙의 결과를 볼 수 있으며, 각 점은 거의 최적의 설정을 나타냅니다.

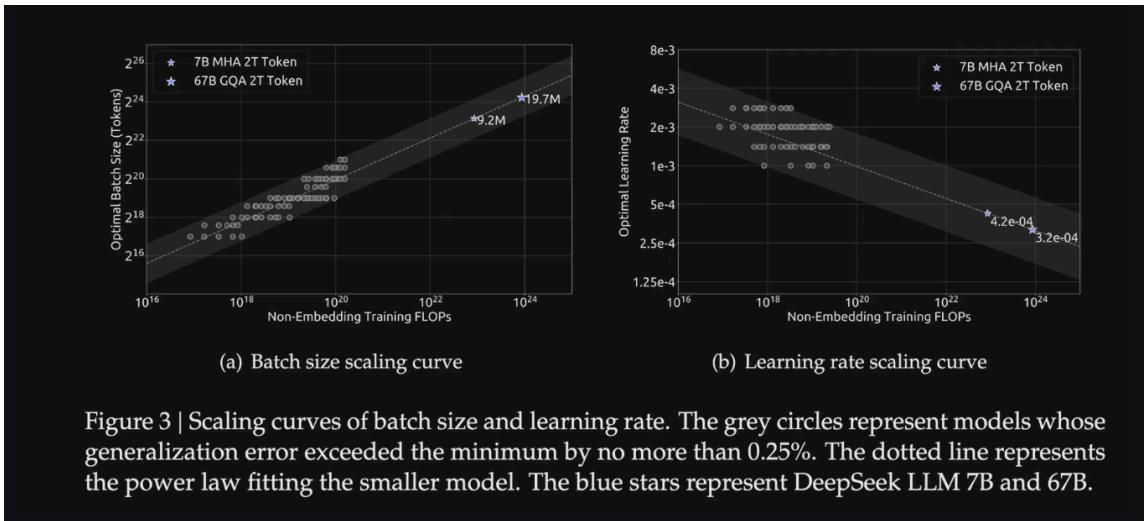


Figure 3 | Scaling curves of batch size and learning rate. The grey circles represent models whose generalization error exceeded the minimum by no more than 0.25%. The dotted line represents the power law fitting the smaller model. The blue stars represent DeepSeek LLM 7B and 67B.

이러한 결과 뒤의 핵심 직관은 학습이 더 크고 길어질수록, 더 안정적인 업데이트(따라서 더 작은 학습률)와 더 효율적인 그레디언트 추정(따라서 더 큰 배치 크기)을 원한다는 것입니다.

이러한 스케일링 법칙은 학습률과 배치 크기의 시작점을 제공합니다. 그러나 목표는 "그래디언트당 최적의 샘플"이 아니라 "시간과 GPU 수 제약 내에서 도달할 수 있는 더 낮은 손실"이면서 모든 토큰에서 전체 신호를 추출하는 것입니다.

실제로 앞서 논의한 임계 배치 크기까지 데이터 효율성을 의미 있게 해치지 않으면서 처리량을 크게 개선하기 위해 예측된 최적 배치 크기 이상으로 배치 크기를 늘릴 수 있을 수 있습니다.

SmolLM3

그래서 SmolLM3에서 최종적으로 무엇을 사용했을까요? SmolLM3를 시작하기 전 절제 실험 당시, 100B 토큰에서 학습된 1B 모델에서 AdamW, AdEMAMix, Muon을 비교했습니다. Muon은 적절히 튜닝되면 AdamW를 능가할 수 있었지만 학습률에 민감하고 발산하기 쉬웠습니다. AdeMaMix는 덜 민감했고 Muon과 유사한 손실을 달성했습니다. AdamW가 가장 안정적이었지만 튜닝된 대안보다 더 높은 최종 손실에 도달했습니다.

그러나 3B로 스케일업했을 때, Muon과 AdeMaMix에서 더 빈번한 발산을 경험했습니다. 이것은 절제 실험을 끝낸 후 발견한 병렬성 버그 때문일 수 있지만(학습 마라톤 장 참조), 이를 확인하지는 못했습니다. AdamW(beta1: 0.9, beta2: 0.95)를 가중치 감쇠 0.1과 그래디언트 클리핑 1로 사용하기로 결정했습니다. 결국 매우 기본적인 설정입니다.

학습률 스케줄로 WSD를 선택했습니다. SmolLM2에서 성공적으로 사용했고, 총 학습 기간에 대한 사용 용이성과 유연성, 그리고 중간 학습 감쇠 실험을 실행할 수 있는 능력에서 최고의 결정 중 하나임이 입증되었습니다. 학습률 스윕을 실행하고 2e-4로 결정했습니다. 전역 배치 크기의 경우, 2M에서 4M 토큰까지의 값을 테스트했지만 손실이나 다운스트림 성능에 미치는 영향이 미미하여, 최고의 처리량을 제공한 2.36M 토큰을 선택했습니다.

참여 규칙

요약: 탐구와 실행의 균형, 완벽한 것보다 완료된 것이 낫다.

우리는 "무엇"(옵티마이저, 학습률, 배치 크기)에 대해 많이 이야기했지만 "어떻게"도 마찬가지로 중요합니다. 무엇이 실험 할 가치가 있는지 어떻게 결정할까요? 시간을 어떻게 구성할까요? 언제 탐구를 멈추고 그냥 학습할까요?

탐구와 실행 사이에 시간을 현명하게 할당하세요. 새로운 방법에서 작은 개선을 완벽하게 하는 데 몇 주를 보내는 것은 동일 한 컴퓨팅을 더 나은 데이터 큐레이션이나 더 철저한 아키텍처 절제 실험에 투자하는 것보다 덜 가치가 있습니다. 우리 경험에서, 그리고 아키텍처 열성가들을 실망시킬 수 있지만, 가장 큰 성능 향상은 보통 데이터 큐레이션에서 나옵니다.

확신이 없을 때는 최고 성능보다 유연성과 안정성을 선택하세요. 두 방법이 동등하게 잘 수행된다면, 더 많은 유연성을 제공하거나 더 나은 구현 성숙도와 안정성을 가진 것을 선택하세요. 학습을 확장하거나 중간 학습 실험을 실행할 수 있게 해주는 WSD와 같은 학습률 스케줄은 약간 더 잘 수렴할 수 있는 엄격한 스케줄보다 더 가치가 있습니다.

언제 최적화를 멈추고 학습을 시작할지 알아야 합니다. 항상 튜닝할 하이퍼파라미터가 하나 더 있거나 시도할 옵티마이저가 하나 더 있습니다. 탐구 마감 시한을 설정하고 지키세요. 실제로 학습을 완료한 모델이 시작조차 하지 않은 완벽한 모델보다 항상 이깁니다.

완벽은 좋은 것의 적입니다, 특히 유한한 컴퓨팅 예산과 마감 시한으로 작업할 때는 더욱 그렇습니다.

스케일링 법칙: 얼마나 많은 파라미터, 얼마나 많은 데이터?

딥러닝 초기 시절, 언어 모델(그리고 이들이 학습된 클러스터)이 "대규모"가 되기 전에는, 학습 실행이 종종 컴퓨팅에 의해 크게 제약받지 않았습니다. 모델을 학습시킬 때, 하드웨어에 맞는 가장 큰 모델과 배치 크기를 선택한 다음 모델이 과적합되기 시작하거나 데이터가 부족할 때까지 학습했습니다. 그러나 이 초기 시절에도 스케일이 도움이 된다는 감각이 있었습니다. 예를 들어, [Hestness et al.](#)은 2017년에 더 큰 모델을 더 오래 학습시키면 예측 가능한 이득을 얻는다는 것을 보여주는 포괄적인 결과 세트를 제공했습니다.

대규모 언어 모델 시대에는 항상 컴퓨팅에 제약을 받습니다. 왜일까요? 이러한 초기 확장성 개념은 [Kaplan et al.의 신경 언어 모델에 대한 스케일링 법칙 연구](#)에서 공식화되었으며, 언어 모델 성능이 여러 차수의 규모에 걸쳐 놀라울 정도로 예측 가능하다는 것이 보여졌습니다. 이것은 규모 증가로 성능이 얼마나 향상될지 정확하게 예측하는 방법을 제공하기 때문에 언어 모델의 크기와 학습 기간의 폭발을 일으켰습니다. 결과적으로, 더 나은 언어 모델을 구축하기 위한 경쟁은 점점 커지는 컴퓨팅 예산으로 더 많은 양의 데이터에서 더 큰 모델을 학습시키는 경쟁이 되었고, 언어 모델의 개발은 빠르게 컴퓨팅 제약 상태가 되었습니다.

컴퓨팅 제약에 직면했을 때, 가장 중요한 질문은 더 큰 모델을 학습시킬지 아니면 더 많은 데이터로 학습시킬지입니다. 놀랍게도, Kaplan et al.의 스케일링 법칙은 이전 모범 사례보다 모델 규모에 훨씬 더 많은 컴퓨팅을 할당하는 것이 유리하다고 제안했습니다. 예를 들어, 상대적으로 적당한 토큰 예산(300B 토큰)으로 거대한(175B 파라미터) GPT-3 모델을 학습시키는 것을 동기 부여했습니다. 재검토 후, [Hoffman et al.](#)은 Kaplan et al.의 접근 방식에서 방법론적 문제를 발견했고, 궁극적으로 학습 기간에 훨씬 더 많은 컴퓨팅을 할당할 것을 제안하는 스케일링 법칙을 재도출했습니다. 예를 들어, 175B 파라미터 GPT-3의 컴퓨팅 최적 학습은 3.7T 토큰을 소비해야 했습니다!

이것은 분야를 "모델을 더 크게"에서 "더 오래 더 잘 학습시키기"로 전환시켰습니다. 그러나 대부분의 현대 학습은 여전히 Chinchilla 법칙을 엄격하게 따르지 않는데, 단점이 있기 때문입니다. 특정 컴퓨팅 예산이 주어졌을 때 최고의 성능을 달성하는 모델 크기와 학습 기간을 예측하는 것을 목표로 하지만, 더 큰 모델이 학습 후에 더 비싸다는 사실을 고려하지 않습니다. 다시 말해, 주어진 컴퓨팅 예산을 사용하여 더 작은 모델을 더 오래 학습시키는 것을 실제로 선호할 수 있습니다. "컴퓨팅 최적"이 아니더라도 이것은 추론 비용을 더 저렴하게 만들기 때문입니다([Sardana et al.](#), [de Vries](#)). 모델이 많은 추론 사

용을 볼 것으로 예상되는 경우(예를 들어, 공개적으로 릴리스되기 때문에 😊) 이런 경우가 될 수 있습니다. 최근에 스케일링 법칙이 제안하는 학습 기간을 넘어 모델을 "과학습"하는 이 관행이 표준 관행이 되었으며, SmolLM3를 개발할 때 우리가 취한 접근 방식입니다.

스케일링 법칙이 특정 컴퓨팅 예산이 주어졌을 때 모델 크기와 학습 기간에 대한 제안을 제공하지만, 과학습을 선택한다는 것은 이러한 요소를 직접 결정해야 한다는 것을 의미합니다. SmolLM3의 경우, 30억 파라미터의 목표 모델 크기를 선택하는 것으로 시작했습니다. Qwen3 4B, Gemma 3 4B, Llama 3.2 3B와 같은 유사한 규모의 최근 모델을 기반으로, 3B가 의미 있는 능력(추론 및 도구 호출과 같은)을 가지기에 충분히 크지만, 초고속 추론과 효율적인 로컬 사용을 가능하게 할 만큼 충분히 작다고 생각했습니다. 학습 기간을 선택하기 위해, 먼저 최근 모델이 극도로 과학습되었다는 것에 주목했습니다. 예를 들어, 앞서 언급한 Qwen3 시리즈는 36T 토큰 동안 학습되었다고 주장됩니다! 결과적으로, 학습 기간은 종종 사용 가능한 컴퓨팅 양에 의해 결정됩니다. 우리는 약 한 달 동안 384개의 H100을 확보했고, 이는 11조 토큰에서 학습할 예산을 제공했습니다(~30%의 MFU를 가정).

스케일링 법칙

이러한 편차에도 불구하고, 스케일링 법칙은 실용적으로 가치가 있습니다. 실험 설계를 위한 기준선을 제공하고, 사람들은 종종 절제 실험에서 신호를 얻기 위해 Chinchilla 최적 설정을 사용하며, 모델 크기가 목표 성능에 도달할 수 있는지 예측하는 데 도움이 됩니다. de Vries가 이 블로그에서 언급하듯이, 모델 크기를 줄이면 임계 모델 크기에 도달할 수 있습니다. 주어진 손실에 도달하는 데 필요한 최소 용량이며, 그 아래로 내려가면 수익 체감이 시작됩니다.

이제 모델 아키텍처, 학습 설정, 모델 크기, 학습 기간이 결정되었으니, 두 가지 중요한 구성 요소를 준비해야 합니다. 모델을 가르칠 데이터 혼합과 안정적으로 학습시킬 인프라입니다. SmolLM3의 아키텍처가 3B 파라미터로 설정된 상태에서, 강력한 다국어, 수학, 코드 성능을 제공할 데이터 혼합을 큐레이션하고, 11조 토큰의 학습에 충분히 견고한 인프라를 설정해야 합니다. 이러한 기본 사항을 올바르게 하는 것이 필수적입니다. 최고의 아키텍처 선택도 나쁜 데이터 큐레이션이나 불안정한 학습 시스템에서 우리를 구하지 못합니다.

The art of data curation

다음 상황을 상상해 보세요. 아키텍처를 완벽하게 하고, 하이퍼파라미터를 튜닝하고, 가장 견고한 학습 인프라를 설정하는 데 몇 주를 보냈습니다. 모델이 아름답게 수렴하고, 그런 다음... 일관된 코드를 작성할 수 없고, 기본적인 수학에 어려움을 겪고, 심지어 문장 중간에 언어를 바꿉니다. 무엇이 잘못되었을까요? 답은 보통 데이터에 있습니다. 우리가 화려한 아키텍처 혁신과 하이퍼파라미터 스윕에 집착하는 동안, 데이터 큐레이션이 종종 모델이 진정으로 유용해지는지 아니면 또 다른 비싼 실험이 되는지를 결정합니다. 랜덤 웹 크롤에서 학습하는 것과 모델이 배우기를 원하는 기술을 실제로 가르치는 신중하게 큐레이션된 고품질 데이터셋에서 학습하는 것의 차이입니다.

모델 아키텍처가 모델이

어떻게

학습하는지 정의한다면, 데이터는 무엇을 학습하는지 정의하며, 어떤 양의 컴퓨팅이나 옵티마이저 튜닝도 잘못된 콘텐츠에서 학습하는 것을 보상할 수 없습니다. 더욱이 학습 데이터를 올바르게 하는 것은 좋은 데이터셋을 갖는 것만이 아닙니다. 올바른 혼합을 조립하는 것입니다. 상충하는 목표(강한 영어 vs. 견고한 다국어 지원과 같은)의 균형을 맞추고 성능 목표에 맞게 데이터 비율을 튜닝하는 것입니다. 이 과정은 보편적인 최고의 혼합을 찾는 것보다 올바른 질문을 하고 이에 답하기 위한 구체적인 계획을 세우는 것에 더 가깝습니다.

- 모델이 무엇을 잘하기를 원하나요?
- 각 도메인에 가장 좋은 데이터셋은 무엇이고 어떻게 혼합하나요?
- 목표 학습 규모에 충분한 고품질 데이터가 있나요?

이 섹션은 원칙적인 방법, 절제 실험, 약간의 연금술을 혼합하여 이러한 질문을 탐색하고, 훌륭한 데이터셋 더미를 훌륭한 학습 혼합으로 바꾸는 것에 관한 것입니다.

좋은 데이터 혼합이란 무엇이고 왜 가장 중요한가

우리는 언어 모델에 많은 것을 기대합니다. 코드 작성, 조언, 모든 것에 대한 질문에 답하고, 도구를 사용하여 작업을 완료하는 등을 해야 합니다. 웹과 같은 풍부한 사전학습 데이터 소스는 이러한 작업에 필요한 지식과 능력의 전체 범위를 다루지 않습니다. 결과적으로, 최근 모델은 수학과 코딩과 같은 특정 도메인을 대상으로 하는 더 전문화된 사전학습 데이터셋에 추가로 의존합니다. 우리는 데이터셋 큐레이션에 대한 많은 과거 작업을 수행했지만, SmoLM3에서는 주로 기존 데이터셋을 사용했습니다. 데이터셋 큐레이션에 대해 더 알아보려면 [FineWeb](#)과 [FineWeb-Edu](#), [FineWeb2](#), [Stack-Edu](#), and [FineMath](#). 구축에 관한 보고서를 확인하세요.

데이터 혼합의 직관에 반하는 특성

언어 모델 학습이 처음이라면, 좋은 데이터 혼합을 찾는 것이 간단해 보일 수 있습니다. 목표 능력을 식별하고, 각 도메인에 대한 고품질 데이터셋을 수집하고, 결합합니다. 현실은 더 복잡합니다. 일부 도메인이 학습 예산을 놓고 서로 경쟁할 수 있기 때문입니다. 코딩과 같은 특정 능력에 집중할 때, 소스 코드와 같은 태스크 관련 데이터를 상향 조정하고 싶은 유혹이 있습니다. 그러나 한 소스를 상향 조정하면 암묵적으로 다른 모든 소스를 하향 조정하여, 다른 설정에서 언어 모델의 능력을 해칠 수 있습니다. 따라서 다양한 소스 모음에서 학습하는 것은 다운스트림 능력 간에 어떤 종류의 균형을 맞추는 것을 포함합니다.

또한, 이러한 모든 소스와 도메인에 걸쳐, 종종 언어 모델의 능력을 개선하는 데 특히 도움이 되는 "고품질" 데이터의 부분집합이 있습니다. 모든 낮은 품질 데이터를 버리고 최고 품질 데이터만으로 학습하면 어떨까요? SmoLM3의 11T 토큰이라는 큰 학습 예산에서, 그러한 극단적인 필터링을 수행하면 데이터를 여러 번 반복하게 됩니다. 이전 연구에서 이러한 종류의 반복이 해로울 수 있음을 보여주었으므로([Muennighoff et al., 2025](#)), 이상적으로는 모델 성능을 최대화하면서 높은 품질과 낮은 품질 모두를 활용할 수 있어야 합니다.

소스 간 데이터 균형을 맞추고 고품질 데이터를 활용하려면, 혼합을 신중하게 설계해야 합니다. 각 소스에서 학습 문서의 상대적 비율입니다. 언어 모델의 특정 태스크나 도메인에서의 성능은 해당 태스크와 관련된 데이터를 얼마나 보았는지에 크게 의존하므로, 혼합 가중치를 튜닝하면 도메인 간 모델의 능력 균형을 맞추는 직접적인 방법을 제공합니다. 이러한 트레이드오프는 모델에 의존적이고 예측하기 어렵기 때문에 절제 실험이 필수적입니다.

그러나 혼합이 학습 전반에 걸쳐 고정될 필요는 없습니다. 학습이 진행됨에 따라 혼합을 조정하면, **다단계 학습 또는 커리큘럼**이라고 부르는 것으로, 고품질과 저품질 데이터 모두를 더 잘 활용할 수 있습니다.

학습 커리큘럼의 진화

대규모 언어 모델 학습 초기에는, 전체 학습 실행에 대해 단일 데이터 혼합을 고정하는 것이 표준 접근 방식이었습니다.

GPT3와 초기 버전의 Llama와 같은 모델은 처음부터 끝까지 정적 혼합에서 학습했습니다. 더 최근에는 분야가 학습 과정에서 데이터 혼합이 변경되는 **다단계 학습**([Allal et al., 2025](#))으로 전환되었습니다. 주요 동기는 언어 모델의 최종 동작이 학

습 말미에 본 데이터에 강하게 영향을 받는다는 것입니다([Y. Chen et al., 2025b](#)). 이 통찰은 실용적인 전략을 가능하게 합니다. 학습 초기에 더 풍부한 소스를 상향 조정하고 말미에 더 작고 더 높은 품질의 소스를 혼합합니다.

일반적인 질문은: 언제 혼합을 변경할지 어떻게 결정하나요? 보편적인 규칙은 없지만, 일반적으로 다음 원칙을 따릅니다.

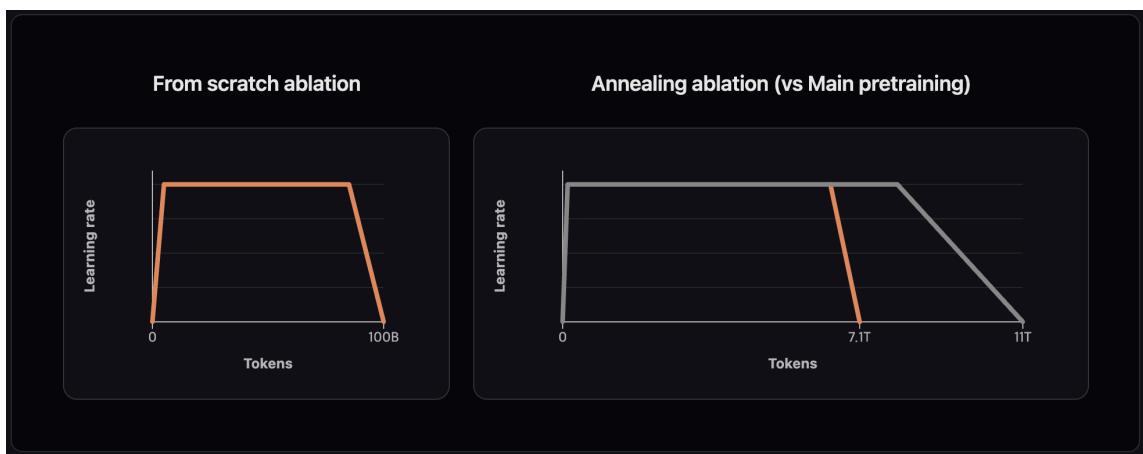
- **성능 주도 개입:** 주요 벤치마크에서 평가 메트릭을 모니터링하고 특정 능력 병목을 해결하기 위해 데이터셋 혼합을 조정합니다. 예를 들어, 다른 능력이 계속 향상되는 동안 수학 성능이 정체되면, 더 높은 품질의 수학 데이터를 도입해야 한다는 신호입니다.
- **고품질 데이터를 후기 단계에 예약:** 작은 고품질 수학 및 코드 데이터셋은 어닐링 단계(학습률 감쇠가 있는 최종 단계) 동안 도입될 때 가장 영향력이 있습니다.

이제 혼합이 왜 중요하고 커리큘럼이 어떻게 작동하는지 확립했으니, 둘 다 튜닝하는 방법에 대해 논의하겠습니다.

Ablation 설정: 데이터 레시피를 체계적으로 테스트하는 방법

데이터 혼합을 테스트할 때, 우리의 접근 방식은 아키텍처 절제 실험을 실행하는 방법과 유사하지만, 한 가지 차이가 있습니다. 목표 모델 규모에서 실행하려고 합니다. 작은 모델과 큰 모델은 다른 용량을 가지고 있습니다. 예를 들어 매우 작은 모델은 많은 언어를 처리하는 데 어려움을 겪을 수 있지만, 더 큰 모델은 다른 곳에서 성능을 희생하지 않고 흡수할 수 있습니다. 따라서 너무 작은 규모에서 데이터 절제 실험을 실행하면 최적의 혼합에 대해 잘못된 결론을 내릴 위험이 있습니다.

SmolLM3의 경우, 50B와 100B 토큰의 더 짧은 학습 실행을 사용하여 3B 모델에서 직접 주요 데이터 절제 실험을 실행했습니다. 또한 다른 유형의 절제 실험 설정인 어닐링 실험을 사용했습니다. 다른 혼합으로 처음부터 학습하는 대신, 메인 실행에서 중간 체크포인트(예: 7T 토큰에서)를 가져와서 수정된 데이터 구성으로 학습을 계속했습니다. 이 접근 방식은 단계 학습(즉, 학습 중간에 학습 혼합 변경)을 위한 데이터 혼합 변경을 테스트할 수 있게 해주며, SmolLM2, Llama3, Olmo2와 같은 최근 연구에서 사용되었습니다. 평가를 위해, 표준 영어 평가와 함께 다국어 태스크를 포함하도록 벤치마크 스위트를 확장하여 다양한 언어 비율 간의 트레이드오프를 적절히 평가할 수 있도록 했습니다.



최근 연구에서 최적의 데이터 비율을 찾기 위한 자동화된 접근 방식을 제안했으며, 다음을 포함합니다.

- **DoReMi** ([Xie et al., 2023](#)): 검증 손실을 최소화하는 도메인 가중치를 학습하기 위해 작은 프록시 모델을 사용

- **Rho Loss** ([Mindermann et al., 2022](#)): 훌드아웃 손실을 기반으로 개별 학습 포인트를 선택하여 학습 가능하고, 태스크 관련성이 있으며, 모델이 아직 학습하지 않은 샘플을 우선시
- **RegMix** ([Q. Liu et al., 2025](#)): 여러 평가 목표와 데이터 도메인에 걸쳐 성능의 균형을 맞추는 정규화된 회귀를 통해 최적의 데이터 혼합 비율을 결정

과거 프로젝트에서 DoReMi와 Rho Loss를 실험했지만, 데이터셋 크기의 자연 분포를 대략적으로 반영하는 분포로 수렴하는 경향이 있음을 발견했습니다. 본질적으로 우리가 더 많이 가지고 있는 것을 더 많이 사용하라고 제안합니다. 이론적으로는 매력적이지만, 우리 설정에서 신중한 수동 절제 실험을 능가하지 못했습니다. 최근 SOTA 모델은 여전히 체계적인 절제 실험과 어닐링 실험을 통한 수동 혼합 튜닝에 의존하며, 이것이 SmoILM3에서 채택한 접근 방식입니다.

SmoILM3: 데이터 혼합 큐레이션 (웹, 다국어, 수학, 코드)

SmoILM3의 경우, 영어와 여러 다른 언어를 처리하고 수학과 코드에서 뛰어난 모델을 원했습니다. 이러한 도메인(웹 텍스트, 다국어 콘텐츠, 코드, 수학)은 대부분의 LLM에서 일반적이지만, 여기서 설명할 프로세스는 저자원 언어나 금융 또는 의료와 같은 특정 도메인을 위해 학습하는 경우에도 동일하게 적용됩니다. 방법은 동일합니다. 좋은 후보 데이터셋을 식별하고, 절제 실험을 실행하고, 모든 목표 도메인의 균형을 맞추는 혼합을 설계합니다.

여기서 고품질 데이터셋을 구축하는 방법은 다루지 않겠습니다. 이미 이전 연구(FineWeb, FineWeb2, FineMath, Stack-Edu)에서 광범위하게 자세히 설명했기 때문입니다. 대신, 이 섹션은 이러한 데이터셋을 효과적인 사전학습 혼합으로 결합하는 방법에 초점을 맞춥니다.

검증된 기반 위에 구축

사전학습 데이터와 관련하여 좋은 소식은 처음부터 시작할 필요가 거의 없다는 것입니다. 오픈소스 커뮤니티가 이미 대부분의 일반적인 도메인에 대해 강력한 데이터셋을 구축했습니다. 때때로 Fine 시리즈(FineWeb, FineMath 등)에서 했던 것처럼 새로운 것을 만들어야 할 때도 있지만, 더 자주 도전은 기존 소스를 재발명하는 것이 아니라 선택하고 결합하는 것입니다.

SmoILM3에서 우리의 상황이 그랬습니다. SmoILM2는 이미 영어 웹 데이터에 대해 1.7B 파라미터에서 강력한 레시피를 확립했고, 우리가 접근할 수 있는 최고의 수학 및 코드 데이터셋을 식별했습니다. 우리의 목표는 그 성공을 3B로 스케일업하면서 특정 능력을 추가하는 것이었습니다. 견고한 다국어 지원, 더 강한 수학 추론, 더 나은 코드 생성입니다.

영어 웹 데이터: 기반 레이어

웹 텍스트는 모든 범용 LLM의 중추를 형성하지만, 품질이 양만큼 중요합니다.

SmoILM3에서 FineWeb-Edu와 DCLM이 학습 당시 가장 강력한 오픈 영어 웹 데이터셋임을 알았습니다. 함께 5.1T 토큰의 고품질 영어 웹 데이터를 제공했습니다. 질문은: 최적의 혼합 비율은 무엇인가요? FineWeb-Edu는 교육 및 STEM 벤치마크에 도움이 되고, DCLM은 상식 추론을 개선합니다.

SmoILM2 방법론을 따라, 100B 토큰에 걸쳐 3B 모델에서 스윕을 실행하여 20/80, 40/60, 50/50, 60/40, 80/20 비율 (FineWeb-Edu/DCLM)을 테스트했습니다. 혼합(약 60/40 또는 50/50)이 최고의 트레이드오프를 제공했습니다. 100B 토큰에서 학습된 3B 모델에서 [SmoILM2.paper](#)과 동일한 절제 실험을 다시 실행했고 동일한 결론을 발견했습니다.

60/40 또는 50/50을 사용하면 벤치마크 전반에 걸쳐 최고의 균형을 제공했으며, SmoILM2 발견과 일치했습니다. Stage 1에서 50/50 비율을 사용했습니다.

[Pes2o](#), [Wikipedia & Wikibooks](#), [StackExchange](#)와 같은 다른 데이터셋도 추가했는데, 이 데이터셋은 성능에 영향을 미치지 않았지만 다양성을 개선하기 위해 포함했습니다.

다국어 웹 데이터

다국어 능력을 위해, 5개의 다른 언어를 목표로 했습니다. 프랑스어, 스페인어, 독일어, 이탈리아어, 포르투갈어입니다. FineWeb2-HQ에서 선택했으며, 총 628B 토큰을 제공했습니다. 또한 중국어, 아랍어, 러시아어와 같은 10개의 다른 언어를 더 작은 비율로 포함했는데, 이들에 대해 최신 성능을 목표로 한 것이 아니라 사람들이 SmoLM3에서 이들에 대한 지속적 사전학습을 쉽게 할 수 있도록 하기 위해서였습니다. FineWeb2-HQ에서 지원되지 않는 언어에는 FineWeb2를 사용했습니다.

핵심 질문은: 웹 데이터 중 얼마나 많은 부분이 비영어여야 하나요? 모델이 언어나 도메인에서 더 많은 데이터를 볼수록 그 언어나 도메인에서 더 잘해진다는 것을 알고 있습니다. 트레이드오프는 고정된 컴퓨팅 예산에서 나옵니다. 한 언어의 데이터를 늘리면 영어를 포함한 다른 언어의 데이터를 줄이는 것을 의미합니다.

3B 모델에 대한 절제 실험을 통해, 웹 혼합에서 12% 다국어 콘텐츠가 적절한 균형을 맞추어 영어 벤치마크를 저하시키지 않으면서 다국어 성능을 개선함을 발견했습니다. 이것은 영어가 주요 언어로 남을 SmoLM3의 예상 사용에 맞았습니다. 5.1T 영어 토큰에 비해 628B 토큰의 비영어 데이터만 있으므로, 훨씬 더 높이 올라가면 다국어 데이터의 더 많은 반복이 필요할 것이라는 점도 주목할 가치가 있습니다.

코드 데이터

Stage 1의 코드 소스는 [The Stack v2](#)와 [StarCoder2](#) 학습 코퍼스에서 추출됩니다.

- [The Stack v2](#) (16개 언어)를 기반으로, StarCoder2Data로 필터링됨.
- 실제 코드 리뷰 추론을 위한 [StarCoder2 GitHub pull requests](#).
- 실행 가능한 단계별 워크플로우를 위한 [Kaggle notebooks](#).
- 코드 주변의 맥락적 논의를 위한 [GitHub issues](#) 와 [StackExchange threads](#).

Aryabumi et al. (2024)은 코드가 코딩 이상으로 언어 모델의 성능을 향상시킨다고 강조합니다. 예를 들어 자연어 추론과 세계 지식에서, 학습 혼합에서 25% 코드를 사용할 것을 권장합니다. 이에 동기를 받아, 혼합에서 25% 코드로 절제 실험을 시작했습니다. 그러나 영어 벤치마크(HellaSwag, ARC-C, MMLU)에서 상당한 저하를 관찰했습니다. 10% 코드로 줄였을 때, 0% 코드에 비해 영어 벤치마크 스위트에서 개선을 보지 못했지만, 코드가 모델에서 갖추어야 할 매우 중요한 능력이므로 어쨌든 포함했습니다.

후기 학습에서 최대 영향을 위해 고품질 데이터를 단계별로 배치하는 원칙에 따라, StarCoder2Data의 교육적으로 필터링된 부분집합인 Stack-Edu의 추가를 후기 단계까지 미루었습니다.

수학 데이터

수학은 코드와 유사한 철학을 따랐습니다. 초기에는 더 크고 더 일반적인 세트인 FineMath3+와 InfiWebMath3+를 사용했고 나중에 FineMath4+와 InfiWebMath4+를 업샘플링하고, 새로운 고품질 데이터셋을 도입했습니다.

- **MegaMath** ([Zhou et al., 2025](#))

- OpenMathInstruct ([Toshniwal et al., 2024](#))와 OpenMathReasoning ([Moshkov et al., 2025](#))과 같은 지시 및 추론 데이터셋

Stage 1에서 FineMath3+와 InfiWebMath3+ 사이에 균등하게 분할된 3%의 수학을 사용합니다. 54B 토큰만 사용 가능하고 Stage 1이 8T에서 9T 토큰으로 추정되므로, 3% 이상의 수학을 사용하면 데이터셋에서 5 에포크 이상이 필요합니다.

새로운 단계를 위한 올바른 혼합 찾기

Stage 1 혼합을 결정하기 위해 처음부터 절제 실험을 실행했지만, 새로운 단계(우리의 경우 두 개의 새로운 단계)를 위한 새로운 데이터셋을 테스트하기 위해 어닐링 절제 실험을 사용했습니다. 약 7T 토큰(Stage 1 후반)에서 체크포인트를 가져와서 다음 설정으로 50B 토큰 어닐링 실험을 실행했습니다.

- **40% 기준 혼합:** 학습해 온 정확한 Stage 1 혼합
- **60% 새 데이터셋:** 평가하고자 하는 후보 데이터셋

예를 들어, MegaMath가 수학 성능을 향상시킬지 테스트하기 위해, 40% Stage 1 혼합(75/12/10/3 도메인 분할 유지)과 60% MegaMath를 실행했습니다.

3개 단계의 구성은 다음 섹션에서 찾을 수 있습니다.

데이터가 신중하게 큐레이션되고 혼합이 절제 실험을 통해 검증되었으니, 실제 학습 여정을 시작할 준비가 되었습니다. 다음 장은 SmoLM3의 한 달간의 학습 실행 이야기입니다. 준비, 예상치 못한 도전, 그리고 그 과정에서 배운 교훈입니다.

학습 마라톤

여기까지 왔다니, 축하합니다! 진짜 재미가 시작됩니다.

이 시점에서 모든 것이 준비되었습니다. 검증된 아키텍처, 최종 확정된 데이터 혼합, 튜닝된 하이퍼파라미터. 남은 것은 인프라를 설정하고 "학습"을 누르는 것뿐입니다.

SmoLM3의 경우, 384개의 H100 GPU(48개 노드)에서 거의 한 달 동안 11조 토큰을 처리하며 학습했습니다. 이 섹션은 긴 학습 실행 중에 실제로 일어나는 일을 안내합니다. 사전 점검, 피할 수 없는 놀라움, 그리고 어떻게 안정성을 유지했는지. 견고한 절제 실험 관행과 신뢰할 수 있는 인프라가 모두 왜 중요한지 직접 보게 될 것입니다. GPU 하드웨어, 스토리지 시스템, 처리량 최적화의 기술적 인프라 세부 사항은 마지막 장에서 다룹니다.

우리 팀은 이것을 여러 번 겪었습니다. StarCoder와 StarCoder2에서 SmoLM, SmoLM2, 이제 SmoLM3까지. 모든 단일 실행이 다릅니다. 수십 개의 모델을 학습시켰더라도, 각각의 새로운 실행은 놀라게 할 새로운 방법을 찾습니다. 이 섹션은 그러한 놀라움에 대비할 수 있도록 확률을 유리하게 쌓는 것에 관한 것입니다.

사전 체크리스트: "학습"을 누르기 전에 확인할 것

"학습"을 누르기 전에, 모든 것이 종단 간 작동하는지 확인하기 위해 체크리스트를 거칩니다.

인프라 준비:

- 클러스터가 Slurm 예약을 지원한다면 사용하세요. SmoLM3의 경우, 전체 실행에 대해 48노드 고정 예약이 있습니다. 이것은 대기열 지연 없음, 일관된 처리량, 시간 경과에 따른 노드 상태 추적 능력을 의미했습니다.
- 시작 전에 GPU를 스트레스 테스트([GPU Fryer](#) 와 [DCGM Diagnostics](#) 사용)하여 스로틀링이나 성능 저하를 포착합니다. SmoLM3의 경우, 스로틀링되는 두 개의 GPU를 발견하고 실행을 시작하기 전에 교체했습니다.
- 스토리지 부풀림 방지:** 우리 시스템은 각 체크포인트를 S3에 업로드한 다음, 다음 것을 저장한 직후 로컬 복사본을 삭제하므로, 빠른 로컬 GPU SSD에 두 개 이상 저장하지 않습니다.

평가 설정: 평가는 기만적으로 시간이 많이 소요됩니다. 모든 것이 구현되어 있어도, 수동으로 실행하고 결과를 로깅하고 플롯을 만드는 것은 매번 몇 시간씩 먹을 수 있습니다. 따라서 완전히 자동화하고, 실행이 시작되기 전에 올바르게 실행되고 로깅되는지 확인하세요. SmoLM3의 경우, 저장된 모든 체크포인트가 자동으로 Wandb와 [Trackio](#)에 로깅되는 클러스터의 평가 작업을 트리거했습니다.

체크포인트 및 자동 재개 시스템: 체크포인트가 올바르게 저장되고 학습 작업이 수동 개입 없이 최신 체크포인트에서 재개할 수 있는지 확인합니다. Slurm에서는 --requeue 옵션을 사용하여 실패한 작업이 자동으로 다시 시작되어 가장 최근 체크포인트에서 재개됩니다.

메트릭 로깅: 관심 있는 모든 메트릭을 로깅하고 있는지 확인합니다. 평가 점수, 처리량(토큰/초), 학습 손실, 그래디언트 노름, 노드 상태(GPU 활용도, 온도, 메모리 사용량), 실행에 특정한 모든 커스텀 디버그 메트릭입니다.

학습 구성 정상성 검사: 학습 구성, 시작 스크립트, Slurm 제출 명령을 다시 확인합니다.

인프라 심층 분석 GPU 테스트, 스토리지 벤치마킹, 모니터링 설정, 복원력 있는 학습 시스템 구축에 대한 자세한 안내는 인프라 장을 참조하세요.

Scaling surprises

SmoLM3에 대한 광범위한 절제 실험을 실행한 후, 전체 규모 실행 준비가 되었습니다. 100B 토큰에서의 3B 절제 실험은 유망해 보였습니다. SmoLM2에 비해 아키텍처 변경([Architecture Choices](#): GQA, NoPE, 문서 마스킹, 토크나이저)은 성능을 개선하거나 유지했고, 영어, 다국어, 코드, 수학 성능의 균형을 맞추는 좋은 데이터 혼합을 찾았습니다([데이터 큐레이션의 기술 참조](#)). 384개 GPU(48개 노드)에서 약 30% MFU를 위해 구성을 최적화했습니다.

실행을 시작하기 전에 처리량을 검증하기 위해 48개 노드에서 몇 가지 절제 실험을 실행했습니다. 자세한 내용은 인프라 장에서 찾을 수 있습니다.

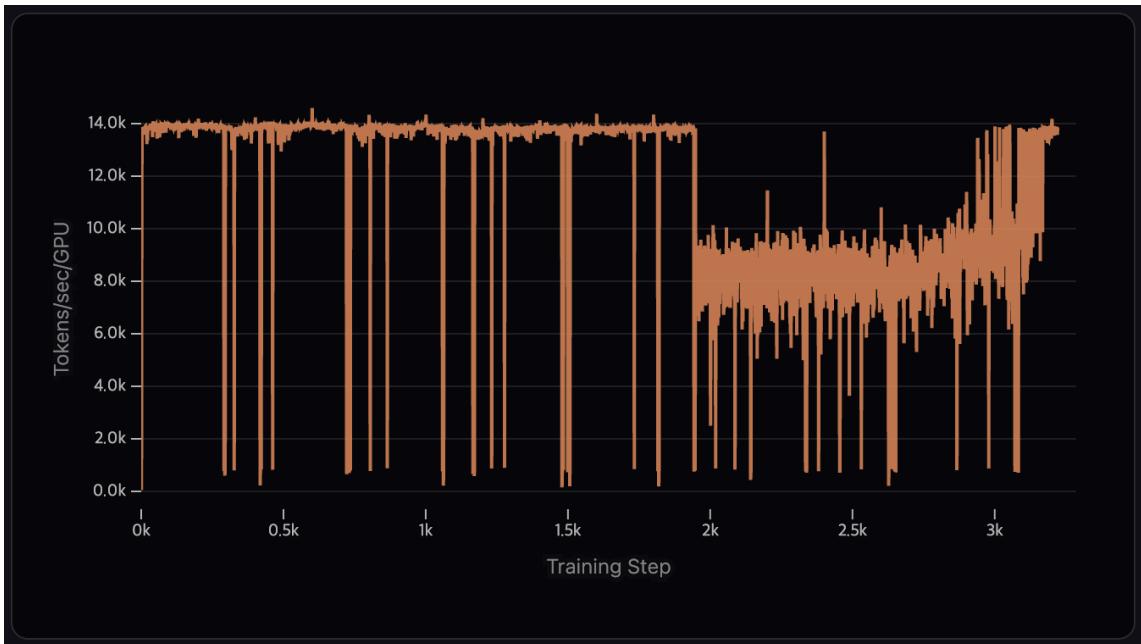
큰 것을 위한 준비가 되었습니다. 11T 토큰. 그때 현실이 커브볼을 던지기 시작했습니다.

미스터리 #1 – 사라지는 처리량

시작 몇 시간 내에 처리량이 급락했습니다. 반복되는 급격한 하락과 함께 큰 점프였습니다.

💡 왜 처리량이 중요한가

처리량은 학습 중 시스템이 초당 처리하는 토큰 수를 측정합니다. 학습 시간에 직접적인 영향을 미칩니다. 처리량이 50% 감소하면 한 달짜리 실행이 두 달짜리 실행이 됩니다. 인프라 장에서 실행을 시작하기 전에 SmoLM3의 처리량을 어떻게 최적화했는지 보여드리겠습니다.



이것은 어떤 Ablation 실행에서도 발생하지 않았는데, 무엇이 변했을까요? 세 가지입니다.

1. 하드웨어 상태는 시간이 지남에 따라 변할 수 있습니다. 절제 실험에서 잘 작동했던 GPU가 실패할 수 있고 네트워크 연결이 지속적인 부하에서 저하될 수 있습니다.
2. 학습 데이터셋의 크기. 이제 절제 실험의 더 작은 부분집합 대신 전체 ~24TB 학습 데이터셋을 사용했습니다. 데이터 소스 자체는 동일했지만요.
3. 학습 스텝 수. 짧은 100B 토큰 절제 실험 기간 대신 11T 토큰에 대한 실제 스텝 수를 설정했습니다.

다른 모든 것은 처리량 절제 실험과 정확히 동일하게 유지되었습니다. 노드 수, 데이터로더 구성, 모델 레이아웃, 병렬성 설정...

직관적으로 데이터셋 크기나 스텝 수가 처리량 하락을 유발해서는 안 되므로, 자연스럽게 먼저 하드웨어 문제를 의심했습니다. 노드 모니터링 메트릭을 확인했고, 큰 처리량 점프가 디스크 읽기 지연 시간 스파이크와 상관관계가 있음을 보여주었습니다. 이것은 우리를 데이터 스토리지로 곧바로 가리켰습니다.

📍 클러스터의 스토리지 옵션

우리 클러스터에는 학습 데이터를 위한 세 가지 스토리지 계층이 있습니다.

- **FSx:** [Weka](#)를 사용하는 네트워크 연결 스토리지로, 자주 접근하는 파일을 로컬에 저장하고 용량이 차면 비활성 "콜드" 파일을 S3로 퇴거하는 "keep-hot" 캐싱 모델입니다.
- **Scratch (로컬 NVMe RAID):** 각 노드의 빠른 로컬 스토리지(RAID의 8x3.5TB NVMe 드라이브)로, FSx보다 빠르지만 로컬 노드 접근으로 제한됩니다.
- **S3:** 콜드 데이터와 백업을 위한 원격 객체 스토리지. 자세한 내용은 인프라 장에서 찾을 수 있습니다.

SmolLM3의 24TB 데이터셋의 경우, 처음에 데이터를 FSx(Weka)에 저장했습니다. 24TB의 학습 데이터와 여러 다른 팀이 이미 사용하는 스토리지 위에서, Weka의 스토리지를 한계까지 밀어붙이고 있었습니다. 그래서 학습 중간에 데이터셋 샘플을 퇴거하기 시작했고, 이는 다시 가져와야 함을 의미했고, 정체를 만들었으며, 이것이 큰 처리량 절감을 설명했습니다. 더 나쁜 것은: 전체 학습 동안 데이터셋 폴더를 핫으로 고정할 방법이 없었습니다.

수정 #1 – 데이터 스토리지 변경

Weka에서 전체 학습 동안 데이터셋 폴더를 핫으로 고정하는 방법을 찾지 못해서, 스토리지 방법을 변경하려고 했습니다. S3에서 직접 스트리밍하는 것은 느렸으므로, 각 노드의 로컬 스토리지 `/scratch`에 데이터를 저장하기로 결정했습니다.

이것에는 문제가 있었습니다. 노드가 죽고 교체되면, 새 교체 GPU에는 데이터가 없었습니다. `s5cmd`로 S3에서 24TB를 다운로드하는 데 3시간이 걸렸습니다. S3를 거치는 대신 `fpsync`를 사용하여 다른 건강한 노드에서 복사하여 1시간 30분으로 줄였습니다. 모든 노드가 같은 데이터센터에 있었기 때문에 이것이 더 빨랐습니다.

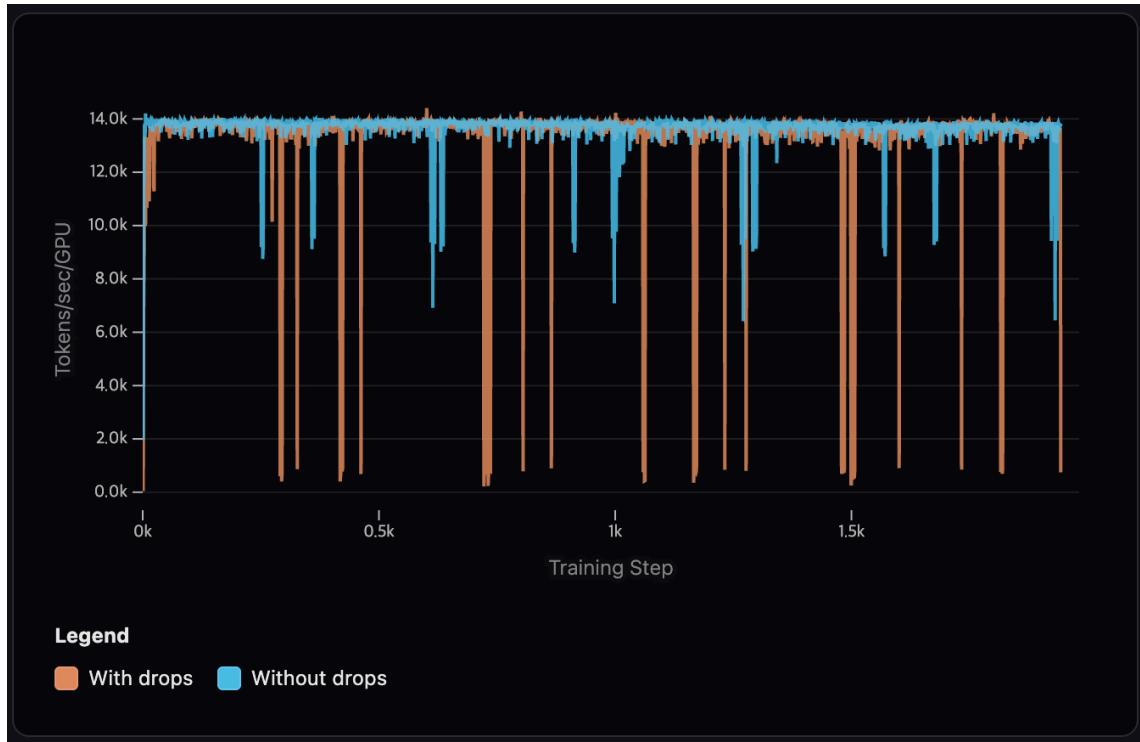
그래도 노드 장애당 1시간 30분의 다운타임과 새 노드에 즉시 데이터를 수동으로 복사해야 하는 필요는 고통스러웠습니다. 마침내 견딜 만하게 만든 해킹: Slurm 예약에 데이터셋이 미리 로드된 예비 노드를 예약합니다. 노드가 죽으면 예비 노드와 즉시 교체하여 복구 지연이 제로입니다. 유튜 상태에서 예비 노드는 평가나 개발 작업을 실행하므로 낭비되지 않았습니다.

이것이 미스터리 #1을 수정했습니다... 아니면 그렇게 생각했습니다.

미스터리 #2 – 지속되는 처리량 하락

스크래치로 이동한 후에도, 하드웨어 모니터링 메트릭에서 이상을 찾지 못했음에도 개별 처리량 하락이 계속 발생했습니다. 아래 차트는 스토리지 문제를 수정한 후 얻은 처리량(주황색)을 절제 실험 중에 얻었던 처리량(파란색)과 비교합니다. 보시

다시피, 하락이 훨씬 더 급격해졌습니다.



여전히 하드웨어를 의심하면서, 더 작은 노드에서 테스트하기로 결정했습니다. 384개의 GPU에서는 무언가가 실패할 가능성이 높습니다. 놀랍게도, 어떤 특정 노드를 테스트하든 단일 노드에서 정확히 동일한 처리량 하락을 재현할 수 있었습니다. 이것은 하드웨어 문제를 배제했습니다.

절제 실험에서 변경된 세 가지를 기억하나요? 로컬 노드 스토리지로 이동하여 데이터 스토리지 문제는 이미 해결했습니다. 이제 하드웨어도 제거되었습니다. 남은 것은 하나의 변수뿐이었습니다: 스텝 수입니다. 더 작은 스텝 수(3M에서 32k로)로 룰백하여 테스트했고 처리량 하락이 더 작아졌습니다! 더 큰 스텝 수는 더 급격하고 더 빈번한 하락을 만들었습니다.

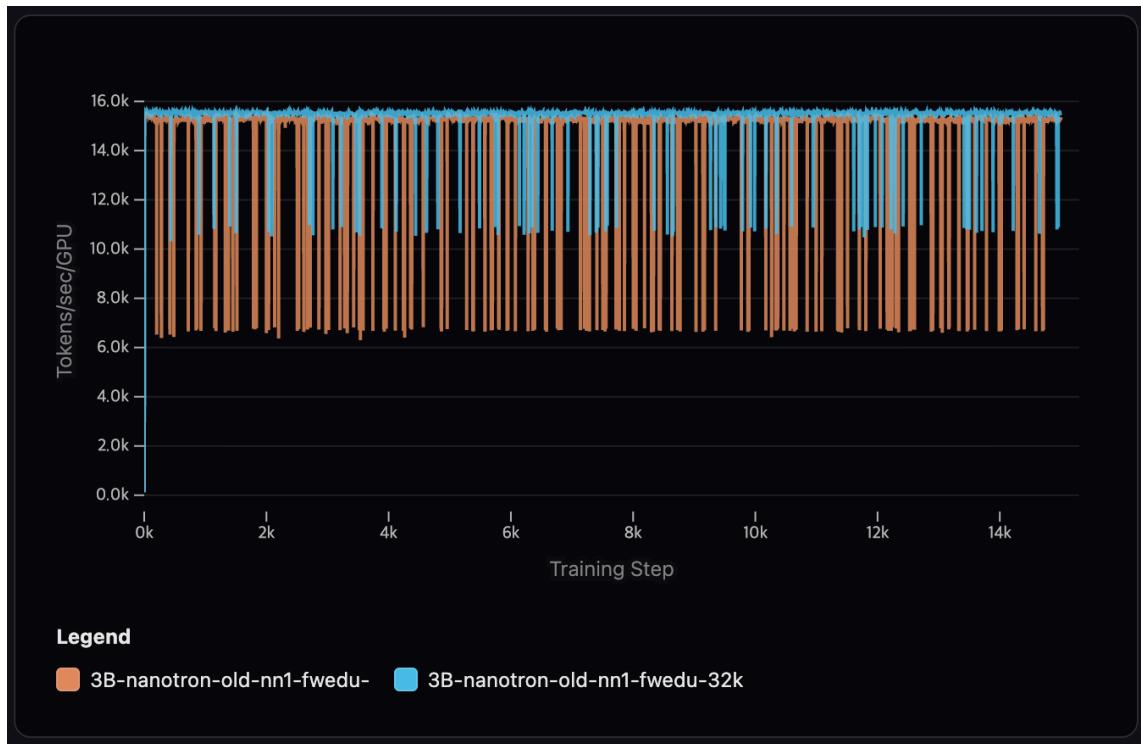
이것을 테스트하기 위해, 학습 스텝만 32k에서 3.2M으로 변경한 동일한 구성을 실행했습니다. [사용한 정확한 구성](#)을 볼 수 있습니다.

```
## 짧은 실행 (32k 스텝)
"lr_decay_starting_step": 2560000
"lr_decay_steps": 64000
"train_steps": 3200000

## 긴 실행 (3.2M 스텝)
"lr_decay_starting_step": 26000
"lr_decay_steps": 6000
"train_steps": 32000
```

아래 그림에 표시된 결과는 명확했습니다. 짧은 실행은 작은 처리량 하락을 보였고, 더 긴 스텝 수는 더 급격하고 더 빈번한 하락을 만들었습니다. 따라서 문제는 하드웨어가 아니라 소프트웨어 병목이었습니다. 아마도 데이터로더에서! 대부분의 다

른 학습 구성 요소는 스텝 수에 관계없이 각 배치를 동일하게 처리한다는 점을 감안하면요.



그때 우리는 nanotron의 데이터로더로 대규모 사전학습을 실제로 해본 적이 없다는 것을 깨달았습니다. SmoLM2는 nanotron을 감싸는 내부 래퍼를 통해 Megatron-LM에서 파생된 데이터로더(TokenizedBytes)를 사용하여 안정적인 처리량으로 학습되었습니다. SmoLM3에서는 nanotron의 내장 데이터로더(nanosets)로 전환했습니다.

구현을 깊이 파헤친 후, 각 학습 스텝마다 커지는 하나의 거대한 인덱스를 순진하게 구축하고 있음을 발견했습니다. 매우 큰 스텝의 경우, 이것은 더 높은 공유 메모리를 유발하여 처리량 하락을 트리거했습니다.

수정 #2 – TokenizedBytes 데이터로더 도입

데이터로더가 실제로 범인인지 확인하기 위해, TokenizedBytes 데이터로더를 사용하는 내부 SmoLM2 프레임워크로 동일한 구성을 시작했습니다. 하락이 없었습니다. 동일한 데이터셋을 사용하는 48개 노드에서도요.

가장 빠른 전진 경로: 이 데이터로더를 nanotron에 복사합니다. 하락이 사라지고 처리량이 목표로 돌아왔습니다.

재시작할 준비가 되었습니다... 다음 커브볼이 올 때까지.

미스터리 #3 – 노이즈가 많은 손실

새 데이터로더로 처리량 하락은 없었지만 손실 곡선이 더 노이즈가 많아 보였습니다.

nanosets는 더 부드러운 손실을 생성했고, 차이점은 오래된 디버깅 전쟁에서 종을 울렸습니다. 몇 년 전, 문서는 셔플되었지만 배치 내 시퀀스는 셔플되지 않아 작은 스파이크가 발생하는 사전학습 코드의 셔플링 버그를 발견했습니다.

새 데이터로더를 확인한 결과 이를 확인했습니다. 각 문서에서 시퀀스를 순차적으로 읽고 있었습니다. 짧은 파일에는 괜찮지만, 코드와 같은 도메인에서는 하나의 긴 저품질 파일이 전체 배치를 채우고 손실 스파이크를 유발할 수 있습니다.

수정 #3 – 시퀀스 수준에서 셔플

두 가지 옵션이 있었습니다.

- 데이터로더를 랜덤 액세스로 변경 (위험: 더 높은 메모리 사용량).
- 토큰화된 시퀀스를 오프라인에서 미리 셔플.

실행을 시작해야 하는 시간 압박과 클러스터 예약이 진행 중인 상황에서, 더 안전하고 빠른 수정으로 옵션 #2를 선택했습니다. 토큰화된 데이터는 이미 각 노드에 있었으므로, 로컬에서 재셔플하는 것은 저렴했습니다(~1시간). 또한 에포크 간 셔플링 패턴이 반복되는 것을 피하기 위해 각 에포크에 대해 다른 시드로 셔플된 시퀀스를 생성했습니다.

언제 패치할지 vs 수정할지 알아야 합니다 간급한 마감에 직면했을 때, 자신의 깨진 구현을 디버그하는 것보다 검증된 솔루션이나 빠른 해결책을 선택하는 것이 더 빠를 수 있습니다. 앞서 nanosets의 인덱스 구현을 수정하는 대신 TokenizedBytes 데이터로더를 연결했습니다. 여기서는 데이터로더 변경 대신 오프라인 미리 셔플을 선택했습니다. 그러나 언제 지름길을 택할지 알아야 합니다. 그렇지 않으면 유지하거나 최적화하기 어려운 패치워크 시스템으로 끝나게 됩니다.

LAUNCH, TAKE TWO

이제 다음을 갖추었습니다.

- 안정적인 처리량 (스크래치 스토리지 + 예비 노드 전략)
- 스텝 수로 인한 하락 없음 (TokenizedBytes 데이터로더)
- 깨끗한 시퀀스 수준 셔플링 (에포크별 오프라인 미리 셔플)

재시작했습니다. 이번에는 모든 것이 유지되었습니다. 손실 곡선이 부드러웠고, 처리량이 일관되었고, 마침내 화재 진압 대신 학습에 집중할 수 있었습니다.

미스터리 #4 – 불만족스러운 성능

처리량과 데이터로더 문제를 수정한 후, 실행을 다시 시작했고 처음 이를 동안 순조롭게 학습했습니다. 처리량이 안정적이었고, 손실 곡선이 예상대로 보였고, 로그에서 어떤 문제도 제시하지 않았습니다. 그러나 약 1T 토큰 지점에서 평가가 예상치 못한 것을 드러냈습니다.

모니터링의 일환으로, 중간 체크포인트를 평가하고 이전 실행과 비교합니다. 예를 들어, 유사한 레시피로 학습된 SmoILM2(1.7B)의 중간 체크포인트가 있었으므로, 학습의 동일한 단계에서 두 모델이 어떻게 진행되는지 추적할 수 있었습니다. 결과는 당혹스러웠습니다. 더 많은 파라미터와 더 나은 데이터 혼합을 가지고 있음에도 불구하고, 3B 모델이 동일한 학습 시점에서 1.7B보다 더 나쁜 성능을 보이고 있었습니다. 손실은 여전히 감소하고 있었고, 벤치마크 점수는 향상되고 있었지만, 개선율이 명확히 기대 이하였습니다.

SmoILM2에 비해 SmoILM3에서 도입된 모든 아키텍처와 데이터 변경을 철저히 테스트했으므로, 학습 프레임워크를 검증했고 두 학습 설정 간에 테스트되지 않은 남은 차이점은 몇 개뿐이었습니다. 가장 명백한 것은 텐서 병렬성이었습니다. SmoILM2는 단일 GPU에 맞을 수 있었고 TP 없이 학습되었지만, SmoILM3는 메모리에 맞추기 위해 TP=2가 필요했습니다.

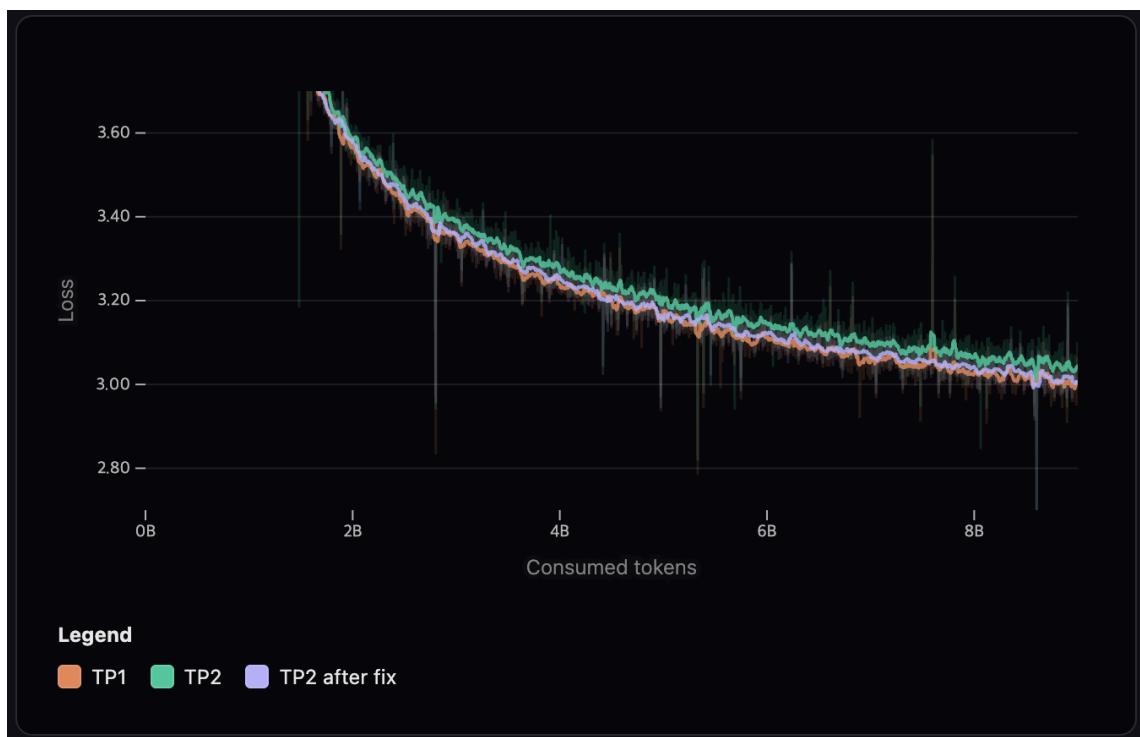
다. TP가 3B 절제 실험에서 사용되었고 그 결과가 말이 되었기 때문에, 이전에 의심하거나 테스트할 생각을 하지 않았습니다.

수정 #4 - 최종 수정

TP 버그 가설을 테스트하기 위해, SmoILM3와 정확히 동일한 설정으로 1.7B 모델을 학습시켰습니다. 동일한 아키텍처 변경(문서 마스킹, NoPE), 동일한 데이터 혼합, 동일한 하이퍼파라미터로, TP가 있는 것과 없는 것 모두요. 차이는 즉각적이었습니다. TP 버전이 일관되게 비TP 버전보다 더 높은 손실과 더 낮은 다운스트림 성능을 보였습니다. 이것은 TP 관련 버그를 보고 있음을 확인했습니다.

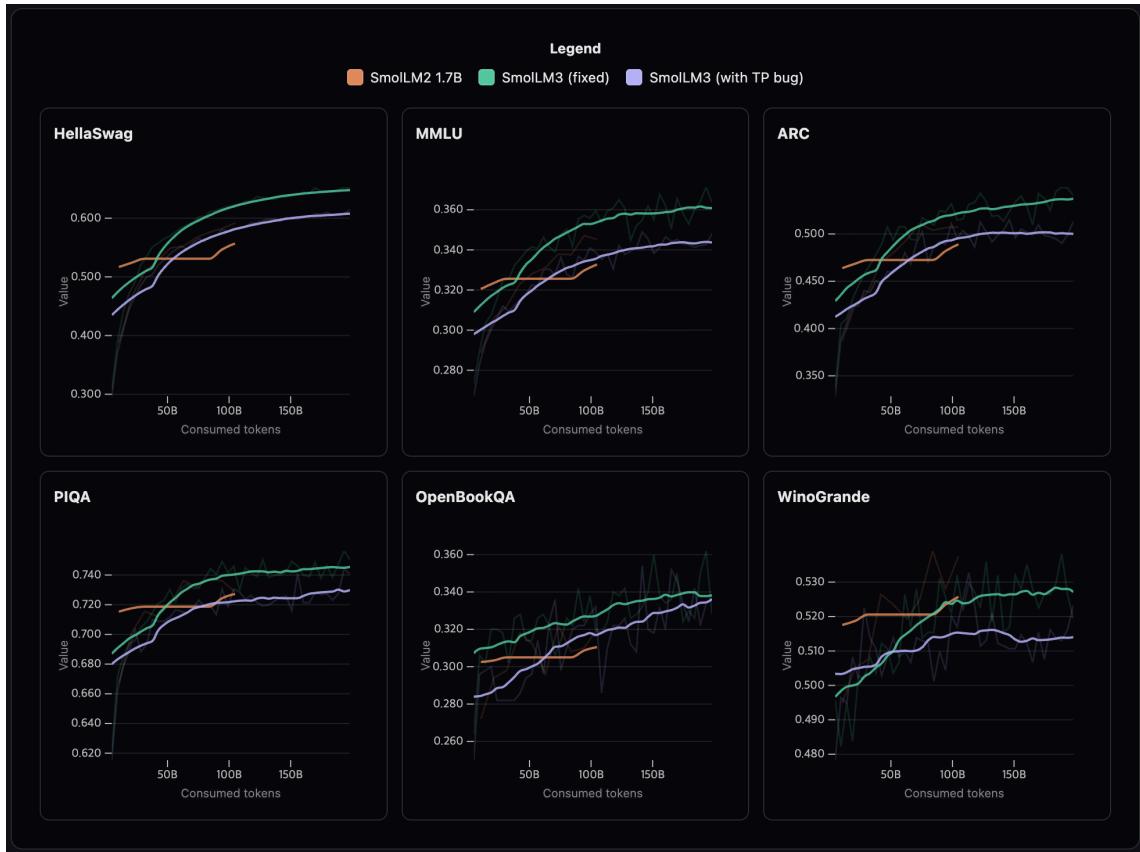
그런 다음 TP와 비TP 실행의 가중치를 비교하며 TP 구현을 자세히 검토했습니다. 문제는 미묘하지만 중대한 것으로 밝혀졌습니다. 각 랭크가 다른 시드로 초기화되어야 할 때 모든 TP 랭크에서 동일한 랜덤 시드를 사용하고 있었습니다. 이것은 샤드 간에 상관된 가중치 초기화를 유발했고, 이것이 수렴에 영향을 미쳤습니다. 효과는 재앙적이지 않았습니다. 모델은 여전히 학습되고 개선되었습니다. 그러나 규모에서 관찰한 격차를 설명하기에 충분한 비효율성을 도입했습니다. 아래는 버그 수정입니다.

```
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:
):
    # 랜덤 상태 설정
-    set_random_seed(self.config.general.seed)
+    # 다양성을 보장하기 위해 각 TP 랭크에 다른 랜덤 시드 설정
+    tp_rank = dist.get_rank(self.parallel_context.tp_pg)
+    set_random_seed(self.config.general.seed + tp_rank)
```



각 TP 랭크가 다른 시드를 사용하도록 시드를 수정한 후, 절제 실험을 반복했고 TP와 비TP 실행이 이제 손실 곡선과 다운스 트림 성능 모두에서 일치함을 확인했습니다. 다른 숨겨진 문제가 없는지 확인하기 위해, 추가적인 정상성 검사를 실행했습니다. 3B 파라미터에서 SmoILM2 스타일(아키텍처와 데이터 측면에서) 실행과 3B 파라미터에서 별도의 SmoILM3 실행을 하고, 둘 다 SmoILM2의 체크포인트와 비교했습니다. 결과는 이제 기대와 일치했습니다. 1.7B SmoILM2는 3B

SmolLM2 변형보다 더 나쁜 성능을 보였고, 이는 다시 SmolLM3의 3B 성능보다 아래였습니다.



이 디버깅 과정은 이 블로그 앞부분에서 설명한 핵심 원칙 중 하나를 강화했습니다.

"견고한 절제 실험 설정의 진정한 가치는 좋은 모델을 구축하는 것 이상입니다. 매번 학습 실행 중에 불가피하게 문제가 발생할 때(그리고 아무리 많이 준비해도 발생할 것입니다), 우리가 내린 모든 결정에 확신을 갖고 적절히 테스트되지 않고 문제를 유발할 수 있는 구성 요소를 빠르게 식별하고 싶습니다. 이 준비는 디버깅 시간을 절약하고 정신 건강을 유지합니다. 버그가 어디에 숨어 있을 수 있는지 전혀 모른 채 미스터리한 학습 실패를 응시하는 것보다 더 나쁜 것은 없습니다."

학습의 다른 모든 구성 요소가 검증되었기 때문에, TP를 유일하게 그럴듯한 원인으로 정확히 찾아내고 성능 격차를 감지한 후 만에 버그를 수정할 수 있었습니다.

이것으로 시작 이후 표면화된 일련의 예상치 못한 문제 중 마지막을 해결했습니다. 세 번째가 행운이라고, 그 시점부터 남은 한 달의 학습은 비교적 평온했습니다. 수조 개의 토큰을 완성된 모델로 바꾸는 꾸준한 작업이었고, 노드 장애로 인한 가끔의 재시작만 있었습니다.

Staying the course

이전 섹션에서 보여주었듯이, 절제 실험에서 전체 사전학습으로 스케일링하는 것은 단순히 "플러그 앤 플레이"가 아니었습니다. 예상치 못한 도전을 가져왔지만, 각 문제를 성공적으로 식별하고 해결했습니다. 이 섹션은 대규모 학습 실행을 위한 필

수 모니터링 설정과 고려 사항을 다룹니다. 중요한 질문에 답하겠습니다. 문제를 만났을 때 언제 학습을 재시작해야 할까요? 실행 깊숙이 표면화되는 문제를 어떻게 처리할까요? 어떤 메트릭이 정말로 중요할까요? 학습 전반에 걸쳐 고정된 데이터 훈합을 유지해야 할까요?

학습 모니터링: 손실 곡선을 넘어서

텐서 병렬성 버그를 잡은 이유는 괜찮아 보였던 손실 곡선이 아니라, 다운스트림 평가가 기대에 뒤처지고 있다는 사실이었습니다. 또한 SmolLM2의 중간 체크포인트에서 평가를 갖는 것이 중요했습니다. 3B 모델이 일찍부터 올바른 궤도에 있지 않는다는 정상성 검사를 제공했습니다. 따라서 대규모 모델을 학습시키고 있다면, 일찍 다운스트림 평가를 실행하기 시작하세요. 오픈소스 모델과 비교하고 있다면, 저자에게 중간 체크포인트를 제공할 수 있는지 물어보세요. 참조 지점으로 매우 귀중할 수 있습니다.

인프라 측면에서 가장 중요한 메트릭은 초당 토큰으로 측정되는 **처리량**입니다. SmolLM3의 경우, 실행 전반에 걸쳐 13,500-14,000 토큰/초 사이의 안정적인 처리량을 기대했고, 지속적인 편차는 위험 신호였습니다. 그러나 처리량만으로는 충분하지 않습니다. 하드웨어 상태를 예측하고 감지하기 위해 지속적인 **하드웨어 상태 모니터링**도 필요합니다. 주요 메트릭 중 일부는 GPU 온도, 메모리 사용량, 컴퓨팅 활용도를 포함했습니다. Grafana 대시보드에 로깅하고 하드웨어 이상에 대한 실시간 Slack 알림을 설정했습니다.

수정하고 재시작 vs 즉석에서 수정

1T 토큰 후에 실행을 재시작했다는 점을 감안할 때, 중요한 질문이 생깁니다. 무언가 잘못되면 항상 재시작해야 하나요? 답은 문제의 심각성과 근본 원인에 따라 달라집니다.

우리의 경우, TP 시딩 버그는 잘못된 발로 시작하고 있음을 의미했습니다. 가중치의 절반이 제대로 초기화되지 않았습니다. 모델은 SmolLM2와 유사한 성능을 보이고 유사한 지점에서 정체되고 있었으며, 이는 동일하게 수행되지만 학습 비용이 거의 두 배인 모델로 끝날 가능성이 높음을 의미했습니다. 재시작이 합리적이었습니다. 그러나 많은 문제는 컴퓨팅 낭비를 피하기 위해 실행 중간에 궤도 수정할 수 있습니다. 가장 일반적인 문제는 **손실 스파이크**입니다. 사소한 문제나 발산을 신호할 수 있는 학습 손실의 갑작스러운 점프입니다.

[Stas Bekman](#)이 [Machine Learning Engineering Open Book](#)에서 멋지게 표현했듯이 "학습 손실 플롯은 심박 패턴과 유사합니다. 좋은 것, 나쁜 것, 그리고 걱정해야 하는 것이 있습니다."



손실 스파이크는 두 가지 범주로 나뉩니다.

- **회복 가능한 스파이크:** 빠르게(스파이크 직후) 또는 느리게(스파이크 전 궤도로 돌아가기 위해 여러 학습 스텝이 더 필요) 회복될 수 있습니다. 보통 이런 스파이크를 통해 학습을 계속할 수 있습니다. 회복이 매우 느리면, 문제가 있는 배치를 건너뛰기 위해 이전 체크포인트로 되감기를 시도할 수 있습니다.
- **회복 불가능한 스파이크:** 모델이 발산하거나 스파이크 전보다 더 나쁜 성능에서 정체됩니다. 단순히 이전 체크포인트로 되감기보다 더 중요한 개입이 필요합니다.

학습 불안정성을 완전히 이해하지는 못하지만, 규모가 커질수록 더 빈번해진다는 것을 알고 있습니다. 보수적인 아키텍처와 옵티마이저를 가정할 때 일반적인 범인은 다음을 포함합니다.

- **높은 학습률:** 학습 초기에 불안정을 유발하며 학습률을 줄여서 수정할 수 있습니다.
- **나쁜 데이터:** 보통 회복 가능한 스파이크의 주요 원인이지만, 회복이 느릴 수 있습니다. 모델이 저품질 데이터를 만날 때 학습 깊숙이 발생할 수 있습니다.
- **데이터-파라미터 상태 상호작용:** PaLM([Chowdhery et al., 2022](#))은 스파이크가 종종 "나쁜 데이터"만이 아니라 데이터 배치와 모델 파라미터 상태의 특정 조합에서 발생한다고 관찰했습니다. 다른 체크포인트에서 동일한 문제 배치로 학습해도 스파이크가 재현되지 않았습니다.
- **잘못된 초기화:** OLMo2([OLMo et al., 2025](#))의 최근 연구에서 스케일된 초기화에서 간단한 정규 분포(평균=0, 표준편차=0.02)로 전환하면 안정성이 향상됨을 보여주었습니다.
- **정밀도 문제:** 더 이상 FP16으로 학습하는 사람은 없지만, [BLOOM](#)은 BF16에 비해 매우 불안정함을 발견했습니다.

스파이크가 발생하기 전에 안정성을 구축하세요:

보수적인 학습률과 좋은 데이터를 가진 소형 모델은 거의 스파이크하지 않지만, 대형 모델은 사전 안정성 조치가 필요합니다. 더 많은 팀이 규모에서 학습함에 따라, 학습 불안정을 방지하는 데 도움이 되는 기술 툴킷을 축적했습니다.

- **데이터 필터링과 셔플링:** 이 블로그의 이 시점에서, 데이터로 얼마나 자주 돌아오는지 눈치챘을 것입니다. 데이터가 깨끗하고 잘 셔플되어 있는지 확인하면 스파이크를 방지할 수 있습니다. 예를 들어, OLMo2는 반복된 n-gram이 있는 문서(1-13 토큰 범위의 32회 이상 반복)를 제거하면 스파이크 빈도가 크게 감소함을 발견했습니다.

- **학습 수정:** Z-loss 정규화는 성능에 영향을 주지 않으면서 출력 로짓이 너무 커지는 것을 방지합니다. 그리고 임베딩을 가중치 감쇠에서 제외하는 것도 도움이 됩니다.
- **아키텍처 변경:** QKNorm(어텐션 전에 쿼리와 키 프로젝션을 정규화)이 효과적임이 입증되었습니다. OLMo2와 다른 팀들은 이것이 안정성에 도움이 됨을 발견했고, 흥미롭게도 Marin 팀은 발산 문제를 수정하기 위해 실행 중간에도 적용할 수 있음을 발견했습니다.

어쨌든 스파이크가 발생할 때 - 피해 통제:

이러한 예방 조치에도 불구하고 스파이크는 여전히 발생할 수 있습니다. 다음은 수정을 위한 몇 가지 옵션입니다.

- **문제가 있는 배치 건너뛰기:** 스파이크 전으로 되감고 문제가 있는 배치를 건너뜁니다. 이것은 스파이크에 대한 가장 일반적인 수정입니다. Falcon 팀([Almazrouei et al., 2023](#))은 스파이크를 해결하기 위해 1B 토큰을 건너뛰었고, PaLM 팀([Chowdhery et al., 2022](#))은 스파이크 위치 주변의 200-500 배치를 건너뛰면 재발을 방지함을 발견했습니다.
- **그래디언트 클리핑 강화:** 그래디언트 노름 임계값을 일시적으로 줄입니다.
- **Marin team에서 수행한 것처럼 QKnorm과 같은 아키텍처 수정 적용**

처리량 하락부터 TP 버그까지 스케일링 도전, 문제를 일찍 잡기 위한 모니터링 관행, 손실 스파이크를 방지하고 수정하는 전략을 살펴보았습니다. 다단계 학습이 모델의 최종 성능을 어떻게 향상시킬 수 있는지 논의하며 이 장을 마무리하겠습니다.

중간 학습(Mid-training)

현대 LLM 사전학습은 일반적으로 다른 데이터 혼합을 가진 여러 단계를 포함하며, 종종 컨텍스트 길이를 확장하는 최종 단계가 뒤따릅니다. 예를 들어, Qwen3([A. Yang, Li, et al., 2025](#))는 3단계 접근 방식을 사용합니다. 4k 컨텍스트에서 30T 토큰에 대한 일반 단계, STEM과 코딩을 강조하는 5T 고품질 토큰에 대한 추론 단계, 마지막으로 32k 컨텍스트 길이에서 수천억 토큰에 대한 긴 컨텍스트 단계입니다. SmoILM3는 유사한 철학을 따르며, 고품질 데이터셋을 도입하고 컨텍스트를 확장하기 위한 계획된 개입과 함께 성능 모니터링을 기반으로 한 반응적 조정이 있습니다.

데이터 큐레이션 섹션에서 설명했듯이, 데이터 혼합이 학습 전반에 걸쳐 고정될 필요는 없습니다. 다단계 학습을 통해 학습이 진행됨에 따라 데이터셋 비율을 전략적으로 변경할 수 있습니다. 일부 개입은 처음부터 계획됩니다. SmoILM3의 경우, Stage 2에서 더 높은 품질의 FineMath4+와 Stack-Edu를 도입하고, 최종 감쇠 단계에서 큐레이션된 Q&A와 추론 데이터를 추가할 것임을 알았습니다. 다른 개입은 반응적이며, 학습 중 성능 모니터링에 의해 주도됩니다. 예를 들어, SmoILM2에서 수학과 코드 성능이 목표에 뒤처지는 것을 발견했을 때, 완전히 새로운 데이터셋(FineMath와 Stack-Edu)을 큐레이션하고 학습 중간에 도입했습니다. 계획된 커리큘럼을 따르는 새로운 격차에 적응하든, 이러한 유연성이 컴퓨팅 예산의 가치를 최대화할 수 있게 해줍니다.

Stage 2와 Stage 3 혼합

아래 차트는 3개의 학습 단계와 학습 중 웹/코드/수학 비율의 진행을 보여줍니다. 각 단계에 대한 SmoILM3 학습 구성은 정확한 데이터 가중치와 함께 여기에서 사용할 수 있습니다. 각 단계 뒤의 근거와 구성에 대한 자세한 내용은 데이터 큐레이션 섹션을 참조하세요.



Stage 1: 기본 학습 (8T 토큰, 4k 컨텍스트) 기반 단계는 핵심 사전학습 혼합을 사용합니다: 웹 데이터(FineWeb-Edu, DCLM, FineWeb2, FineWeb2-HQ), The Stack v2와 StarCoder2의 코드, FineMath3+와 InfiWebMath3+의 수학. 모든 학습은 4k 컨텍스트 길이에서 발생합니다.

Stage 2: 고품질 주입 (2T 토큰, 4k 컨텍스트) 더 높은 품질의 필터링된 데이터셋을 도입합니다: 코드용 Stack-Edu, 수학용 FineMath4+와 InfiWebMath4+, 고급 수학적 추론을 위한 MegaMath(Qwen Q&A 데이터, 합성 재작성, 텍스트-코드 인터리브 블록 추가).

Stage 3: 추론 및 Q&A 데이터와 함께 LR 감쇠 (1.1T 토큰, 4k 컨텍스트) 학습률 감쇠 단계 동안, OpenMathReasoning, OpenCodeReasoning, OpenMathInstruct와 같은 지시 및 추론 데이터를 도입하면서 고품질 코드와 수학 데이터셋을 추가로 업샘플링합니다. Q&A 샘플은 단순히 연결되고 새 줄로 구분됩니다.

긴 컨텍스트 확장: 4k에서 128k 토큰으로

컨텍스트 길이는 모델이 처리할 수 있는 텍스트의 양을 결정하며, 긴 문서 분석, 일관된 다중 턴 대화 유지, 전체 코드베이스 처리와 같은 작업에 중요합니다. SmolLM3는 4k 토큰에서 학습을 시작했지만, 실제 애플리케이션을 위해 128k로 스케일링해야 했습니다.

왜 학습 중간에 컨텍스트를 확장하나요?

어텐션 메커니즘이 시퀀스 길이에 따라 이차적으로 스케일링되므로 처음부터 긴 컨텍스트에서 학습하는 것은 계산적으로 비쌉니다. 더욱이 연구에 따르면 학습 말미나 지속적 사전학습 중에 수십에서 수백억 토큰으로 컨텍스트를 확장하면 좋은 긴 컨텍스트 성능에 도달하기에 충분합니다([Gao et al., 2025](#)).

순차적 스케일링: 4k→32k→64k

128k로 바로 점프하지 않았습니다. 대신 단계적으로 컨텍스트를 점진적으로 확장하여, 더 밀어붙이기 전에 각 길이에서 모델이 적응할 시간을 주었습니다. 두 번의 긴 컨텍스트 단계를 실행했습니다. 먼저 4k에서 32k로, 그 다음 32k에서 64k로 (128k 능력은 학습이 아닌 추론 시 외삽에서 나옵니다). 메인 감쇠 단계의 마지막 100B 토큰 동안 컨텍스트를 확장하는 것

보다 각 단계에서 50B 토큰에 걸쳐 새로운 학습률 스케줄을 시작하는 것이 더 잘 작동함을 발견했습니다. 각 단계에서 좋은 긴 컨텍스트 데이터 혼합과 RoPE 세타 값을 찾기 위해 절제 실험을 실행했고, Ruler 벤치마크에서 평가했습니다.

💡 베이스 모델에서의 긴 컨텍스트 평가

긴 컨텍스트 절제 실험 동안, [HELMET](#) 벤치마크가 베이스 모델에서 매우 노이즈가 많음을 발견했습니다(다른 시드로 동일한 학습이 가변적인 결과를 제공). [Gao et al.](#)은 벤치마크 태스크에서 분산을 줄이기 위해 위에 SFT를 수행할 것을 권장합니다. 대신 베이스 모델 수준에서 더 신뢰할 수 있는 신호를 제공하는 RULER를 선택했습니다.

이 단계에서 긴 컨텍스트 성능을 향상시키기 위해 긴 웹 페이지와 책과 같은 긴 컨텍스트 문서를 업샘플링하는 것이 일반적입니다([Gao et al., 2025](#)). Qwen2.5-1M의 접근 방식([A. Yang, Yu, et al., 2025](#))을 따라 FineWeb-Edu와 Python-Edu로 검색 및 fill-in-the-middle과 같은 태스크를 위해 책, 기사, 심지어 합성으로 생성된 문서를 업샘플링하는 여러 절제 실험을 실행했습니다. 놀랍게도, Stage 3의 기준 혼합만 사용하는 것에 비해 어떤 개선도 관찰하지 못했으며, 이는 이미 Ruler에서 Llama 3.2 3B 및 Qwen2.5 3B와 같은 다른 최신 모델과 경쟁력이 있습니다. 기준 혼합이 자연적으로 웹 데이터와 코드의 긴 문서를 포함하고(토큰의 약 10%로 추정), NoPE 사용이 도움이 되었기 때문이라고 가정합니다.

긴 컨텍스트 확장에 대한 더 많은 통찰을 위해 "How to Train Long-Context Language Models (Effectively)" 논문을 읽는 것을 추천합니다.

RoPE ABF (조정된 기본 주파수를 가진 RoPE): 4k에서 32k로 갈 때 RoPE 세타(기본 주파수)를 2M으로 늘렸고, 32k에서 64k로 갈 때 5M으로 늘렸습니다. 10M과 같은 더 큰 값을 사용하면 RULER 점수가 약간 향상되지만 GSM8k와 같은 일부 짧은 컨텍스트 태스크에 해를 끼치므로, 짧은 컨텍스트에 영향을 미치지 않는 5M을 유지했습니다. 이 컨텍스트 확장 단계에서 수학, 코드, 추론 Q&A 데이터를 추가로 업샘플링할 기회를 활용했고, ChatML 형식으로 수십만 개의 샘플을 추가했습니다.

4k→32k 확장 중에 슬라이딩 윈도우 어텐션(4k, 8k, 16k 윈도우 크기)도 실험했지만, 전체 어텐션에 비해 RULER에서 더 나쁜 성능을 보임을 발견했습니다.

YARN 외삽: 128k 도달 64k 컨텍스트에서 학습한 후에도 SmoLM3가 추론에서 128k를 처리하기를 원했습니다. 128k 시퀀스에서 학습하는 대신(비용이 엄청나게 비쌈), 모델이 학습 길이를 넘어 외삽할 수 있게 해주는 YARN(Yet Another RoPE extensioN method)([B. Peng et al., 2023](#))을 사용했습니다. 이론적으로 YARN은 시퀀스 길이의 4배 증가를 허용합니다. 64k 체크포인트를 사용하면 32k 체크포인트를 사용하는 것보다 128k에서 더 나은 성능을 제공함을 발견했으며, 목표 추론 길이에 더 가깝게 학습하는 것의 이점을 확인했습니다. 그러나 256k(64k의 4배)로 밀어붙이면 Ruler 성능이 저하되었으므로, 모델을 128k까지 사용하는 것을 권장합니다.

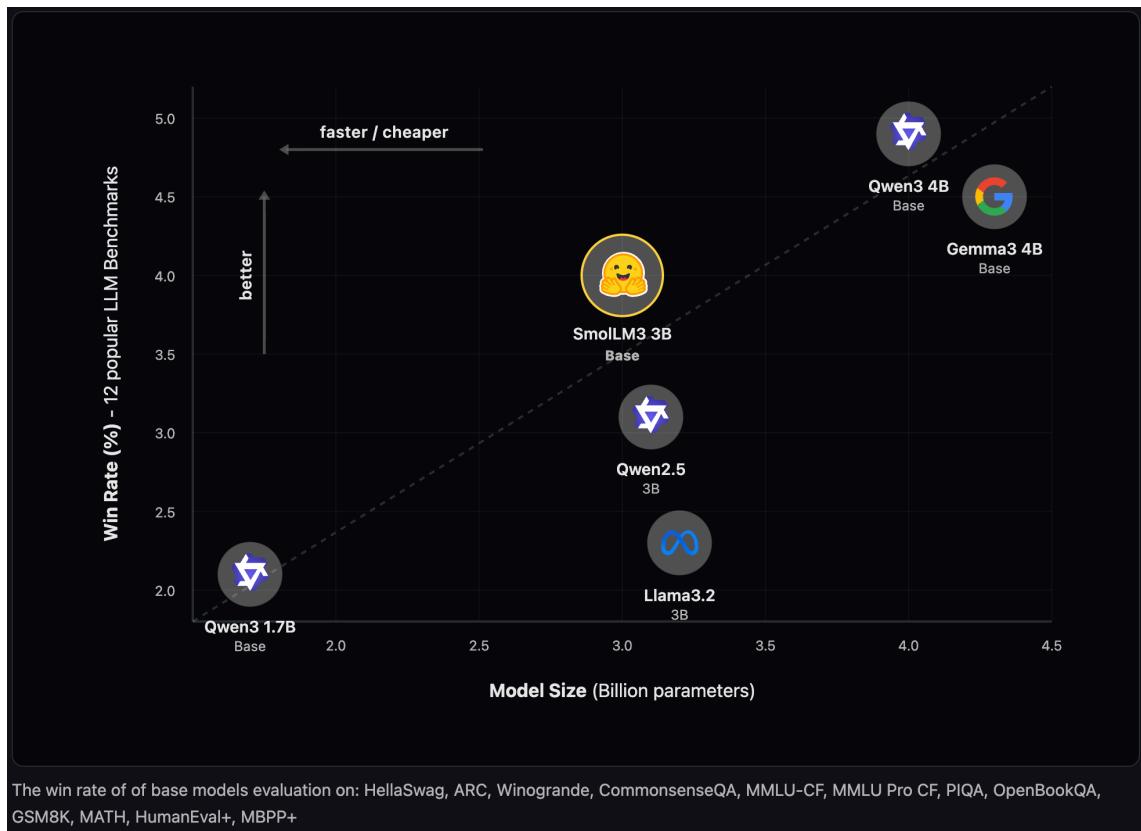
이것으로 계획과 절제 실험부터 최종 학습 실행까지, 그 과정의 모든 비하인드 도전과 함께 SmoLM3의 전체 사전학습 여정을 살펴보았습니다.

사전학습 마무리

많은 것을 다루었습니다. 무엇을 왜 학습할지 결정하는 데 도움이 된 학습 나침반부터, 전략적 계획, 모든 아키텍처 선택을 검증한 체계적인 절제 실험, 규모에서 놀라움이 나타난 실제 학습 마라톤(처리량이 미스터리하게 봉괴, 데이터로더 병목, 1T 토큰에서 재시작을 강제한 미묘한 템서 병렬성 버그)까지.

세련된 기술 보고서 뒤의 지저분한 현실이 이제 보입니다. LLM 학습은 아키텍처 혁신과 데이터 큐레이션만큼이나 규율 있는 실험과 빠른 디버깅에 관한 것입니다. 계획은 무엇이 테스트할 가치가 있는지 식별합니다. 절제 실험은 각 결정을 검증합니다. 모니터링은 문제를 일찍 잡습니다. 그리고 불가피하게 문제가 발생할 때, 체계적인 위험 제거가 정확히 어디를 봄아 하는지 알려줍니다.

SmolLM3에 특히, 이 과정은 우리가 구축하려고 했던 것을 제공했습니다. 11T 토큰에서 학습된 3B 모델로 수학, 코드, 다른 이해, 긴 컨텍스트 태스크에서 경쟁력이 있으며, Qwen3 모델의 파레토 프론티어에 있습니다.



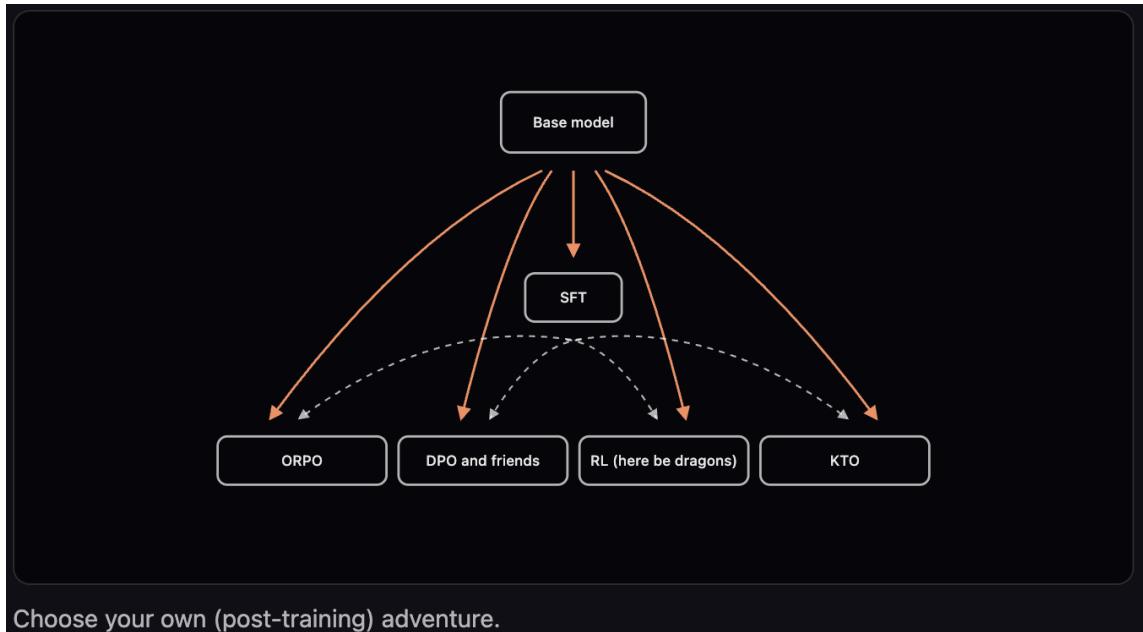
베이스 모델 체크포인트가 저장되고, 학습이 완료되고, GPU가 마침내 식어가면서, 완료라고 부르고 싶은 유혹이 있을 수 있습니다. 결국 텍스트를 잘 예측하고, 강력한 벤치마크 점수를 달성하고, 목표로 한 능력을 보여주는 모델이 있습니다.

아직 아닙니다. 오늘날 사람들이 원하는 것은 원시 다음 토큰 예측기가 아니라 어시스턴트와 코딩 에이전트이기 때문입니다.

여기서 후속 학습이 등장합니다. 그리고 사전학습과 마찬가지로, 현실은 논문이 제시하는 것보다 더 지저분합니다.

베이스 모델을 넘어서 – 2025년의 후속 학습

사전학습이 끝나면 하루 안에 SFT 기준선을 갖추어야 합니다.



사전학습이 SmoLM3의 원시 능력을 제공했지만, GPU가 식기 전에 모델 능력의 다음 프론티어인 **후속 학습**에 들어갑니다. 이것은 지도 파인튜닝, 강화 학습, 모델 병합 등을 포함하며, 모두 "텍스트를 예측하는 모델"에서 "사람들이 실제로 사용 할 수 있는 모델"로의 격차를 메우기 위해 설계되었습니다. 사전학습이 지식을 가중치에 무차별적으로 밀어넣는 것이라면, 후속 학습은 그 원시 능력을 신뢰할 수 있고 조종 가능한 것으로 조각하는 것입니다. 그리고 사전학습과 마찬가지로, 세련된 후속 학습 논문은 심야의 놀라움을 포착하지 못합니다: GPU 멀트다운, 까다로운 데이터 훈합, 또는 겉보기에 사소한 채팅 템플릿 결정이 다운스트림 벤치마크를 통해 파급되는 방식. 이 섹션에서는 SmoLM3를 강력한 베이스 모델에서 최신 하이브리드 추론기로 바꾸기 위해 후속 학습의 지저분한 세계를 어떻게 탐색했는지 보여드리겠습니다.

💡 하이브리드 추론 모델이란?

하이브리드 추론 모델은 두 가지 별개의 모드로 작동합니다: 간결하고 직접적인 응답을 위한 모드와 확장된 단계별 추론을 위한 모드입니다. 일반적으로 작동 모드는 시스템 메시지에서 사용자가 설정합니다. Qwen3를 따라, 가벼운 명령으로 이를 명시적으로 만듭니다: "/think"는 확장된 추론을 호출하고, "/no_think"는 간결한 답변을 강제합니다. 이렇게 사용자가 모델이 깊이를 우선시할지 속도를 우선시할지 제어합니다.

후속 학습 나침반: 왜 → 무엇을 → 어떻게

사전학습과 마찬가지로, 후속 학습은 낭비되는 연구와 엔지니어링 사이클을 피하기 위해 명확한 나침반의 이점을 얻습니다. 프레임하는 방법은 다음과 같습니다.

왜 후속 학습을 하나요? 사전학습 나침반([pretraining compass](#))에서 설명한 학습의 세 가지 동기(연구, 프로덕션, 전략적 오픈소스)는 후속 학습에도 동일하게 적용됩니다. 예를 들어, RL이 기존 모델에서 새로운 추론 능력을 열 수 있는지 탐구하고 있을 수 있고(연구), 지연 시간 이유로 대형 모델을 더 작은 모델로 종류해야 할 수 있고(프로덕션), 특정 사용 사례에 대해

강력한 오픈 모델이 없는 격차를 식별했을 수 있습니다(전략적 오픈소스). 차이점은 후속 학습이 처음부터 만드는 것이 아니라 기존 능력 위에 구축한다는 것입니다. 그러나 GPU에 손을 뻗기 전에 스스로에게 물어보세요:

- **정말로 후속 학습이 필요한가요?** 많은 오픈 가중치 모델이 이제 광범위한 태스크에서 독점 모델과 경쟁합니다. 일부는 양자화와 적당한 컴퓨팅으로 로컬에서도 실행할 수 있습니다. 범용 어시스턴트를 원한다면, [Hugging Face Hub](#)의 기성 모델이 이미 필요를 충족할 수 있습니다.
- **고품질, 도메인 특화 데이터에 접근할 수 있나요?** 후속 학습은 범용 모델이 성능이 떨어지는 특정 태스크나 도메인을 대상으로 할 때 가장 합리적입니다. 적절한 데이터로, 가장 관심 있는 애플리케이션에 대해 더 정확한 출력을 생성하도록 모델을 튜닝할 수 있습니다.
- **성공을 측정할 수 있나요?** 명확한 평가 기준 없이는 후속 학습이 정말로 도움이 되는지 알 수 없습니다.

후속 학습이 무엇을 달성해야 하나요? 이것은 우선순위에 따라 달라집니다:

- 주제에서 거의 벗어나지 않는 선명한 지시 따르기 모델을 원하나요?
- 요청에 따라 톤과 역할을 전환할 수 있는 다재다능한 어시스턴트?
- 수학, 코드, 에이전트 문제를 해결할 수 있는 추론 엔진?
- 여러 언어로 대화할 수 있는 모델?

어떻게 거기에 도달할 건가요? 여기서 레시피가 중요합니다. 다음을 다루겠습니다:

- **지도 파인튜닝(SFT)**: 핵심 능력을 주입하기 위해.
- **선호도 최적화(PO)**: 인간 또는 AI 선호도에서 직접 학습하기 위해.
- **강화 학습(RL)**: 지도 데이터를 넘어 신뢰성과 추론을 개선하기 위해.
- **데이터 큐레이션**: 다양성과 품질 사이의 적절한 균형을 맞추기 위해.
- **평가**: 진행 상황을 추적하고 회귀를 일찍 잡기 위해.

이 나침반은 후속 학습의 혼란을 기반에 두게 합니다. **왜**는 방향을 제공하고, **무엇을**은 우선순위를 설정하고, **어떻게**는 야망을 실용적인 학습 루프로 바꿉니다.

SmoILM3에 대해 이러한 질문에 어떻게 답했는지 살펴보겠습니다:

- **왜?** 우리에게 "왜"는 릴리스 전에 후속 학습이 필요한 베이스 모델이 있었으므로 간단했습니다. 동시에 Qwen3와 같은 하이브리드 추론 모델이 점점 인기를 얻고 있었지만, 어떻게 학습시키는지 보여주는 오픈 레시피는 부족했습니다. SmoILM3는 둘 다 해결할 기회를 주었습니다: 실제 사용을 위한 모델을 준비하고 Qwen3의 1.7B 및 4B 모델과 함께 파레토 프론트에 있을 완전히 오픈된 레시피에 기여합니다.
- **무엇을?** SmoILM3의 강점에 맞춤화된 하이브리드 추론 모델을 학습시키기로 했습니다. 주로 추론 품질이 영어 외의 언어에서도 유지되어야 합니다. 그리고 실제 사용이 점점 더 도구 호출과 긴 컨텍스트 워크플로우를 포함하므로, 이들이 후속 학습 레시피의 핵심 요구 사항이 되었습니다.
- **어떻게?** 이 장의 나머지입니다 😊.

사전학습과 마찬가지로, 기본부터 시작합니다: 평가와 기준선, 모든 큰 모델은 작은 절제 실험으로 시작하기 때문입니다. 그러나 절제 실험 방법에는 핵심 차이가 있습니다. 사전학습에서 "작은"은 보통 더 작은 모델과 데이터셋을 의미합니다. 후속 학습에서 "작은"은 더 작은 데이터셋과 더 간단한 알고리즘을 의미합니다. 동작이 너무 모델에 의존적이고 실행이 충분히 짧아서 목표 모델에서 직접 반복할 수 있으므로, 절제 실험에 다른 베이스 모델을 사용하는 경우는 거의 없습니다.

이 규칙의 주요 예외는 Hugging Face Hub의 기성 베이스 모델을 사용할 때입니다. 이 경우 베이스 모델을 절제 실험 하는 것이 합리적일 수 있습니다. 동일한 크기를 가지더라도 1T 토큰에서 학습된 모델과 10T에서 학습된 모델 사이에 는 큰 차이가 있기 때문입니다.

많은 모델 학습자가 프로젝트에 너무 늦게까지 피하는 주제인 평가부터 시작하겠습니다.

First things first: 다른 모든 것보다 평가 먼저

후속 학습의 가장 첫 번째 단계는 — 사전학습과 마찬가지로 — 적절한 평가 세트를 결정하는 것입니다. 오늘날 대부분의 LLM이 어시스턴트로 사용되므로, [ARC-AGI](#)와 같은 "지능"의 추상적인 벤치마크를 쓰는 것보다 "잘 작동하는" 모델을 목표로 하는 것이 더 나은 목표임을 발견했습니다. 그렇다면 좋은 어시스턴트가 무엇을 해야 할까요? 최소한 다음을 할 수 있어야 합니다:

- 모호한 지시 처리
- 단계별 계획
- 코드 작성
- 적절할 때 도구 호출

이러한 동작은 추론, 긴 컨텍스트 처리, 수학, 코드, 도구 사용 기술의 혼합을 활용합니다. 3B 파라미터 정도 또는 더 작은 모델도 어시스턴트로 잘 작동할 수 있지만, 성능은 보통 1B 아래에서 급격히 떨어집니다.

작은 모델이 도구 호출을 사용하여 제한된 용량을 상쇄하고 따라서 실행 가능한 어시스턴트로 작동할 수 있는지는 흥미롭지만 열린 질문으로 남아 있습니다. 이 방향의 최근 연구는 LiquidAI의 모델을 참조하세요.

Hugging Face에서는 사전학습을 위한 절제 실험 섹션에서 자세히 설명한 사전학습 원칙(단조성, 낮은 노이즈, 랜덤 이상 신호, 순위 일관성)을 반영하는 계층화된 평가 스위트를 사용합니다.

💡 평가를 최신 상태로 유지하세요

고려할 평가 목록은 모델이 개선됨에 따라 지속적으로 발전하고 있으며 아래에서 논의하는 것은 2025년 중반의 우리의 초점을 반영합니다. 후속 학습 평가에 대한 포괄적인 개요는 [Evaluation Guidebook](#)을 참조하세요.

다음은 후속 학습된 모델을 평가할 수 있는 여러 방법입니다.

1. 능력 평가

이 평가 클래스는 추론과 경쟁 수학 및 코딩과 같은 기본 기술을 대상으로 합니다.

- **지식.** 현재 과학 지식의 주요 평가로 GPQA Diamond([Rein et al., 2024](#))를 사용합니다. 이 벤치마크는 대학원 수준의 객관식 질문으로 구성됩니다. 소형 모델의 경우 포화와는 거리가 멀고 MMLU 등보다 더 나은 신호를 제공하면서 실행이 훨씬 빠릅니다. 사실성에 대한 또 다른 좋은 테스트는 SimpleQA([Wei et al., 2024](#))이지만, 소형 모델은 제한된 지식으로 인해 이 벤치마크에서 상당히 어려움을 겪는 경향이 있습니다.
- **수학.** 수학적 능력을 측정하기 위해, 오늘날 대부분의 모델은 최신 버전의 AIME(현재 2025 버전)에서 평가됩니다. MATH-500([Lightman et al., 2023](#))은 소형 모델에 유용한 정상성 테스트로 남아 있지만, 추론 모델에 의해 대체로 포화되었습니다. 더 포괄적인 수학 평가 세트는 [MathArena](#)의 것을 추천합니다.

- **코드.** 코딩 역량을 추적하기 위해 최신 버전의 [LiveCodeBench](#)를 사용합니다. 경쟁 프로그래밍 문제를 대상으로 하지만, LiveCodeBench의 개선이 더 나은 코딩 모델로 이어지는 것을 발견했습니다. 비록 Python으로 제한되지만요. [SWE-bench](#)는 코딩 기술의 더 정교한 측정이지만, 소형 모델에게는 너무 어려운 경향이 있어 보통 고려하지 않습니다.
- **다국어.** 불행히도 모델의 다국어 능력을 테스트할 때 많은 옵션이 없습니다. 현재 모델이 잘 수행해야 하는 주요 언어를 대상으로 Global MMLU([Singh et al., 2025](#))에 의존하며, 다국어 수학적 능력 테스트로 MGSM([Shi et al., 2022](#))을 포함합니다.

2. 통합 태스크 평가

이 평가는 다중 턴 추론, 긴 컨텍스트 사용, 준현실적 설정에서의 도구 호출 등 출시할 것에 가까운 것을 테스트합니다.

- **긴 컨텍스트.** 긴 컨텍스트 검색에 가장 일반적으로 사용되는 테스트는 Needle in a Haystack(NIAH) ([Kamradt, 2023](#))으로, 랜덤 사실("바늘")이 긴 문서("건초더미") 내 어딘가에 배치되고 모델이 이를 검색해야 합니다. 그러나 이 벤치마크는 긴 컨텍스트 이해를 구별하기에는 너무 피상적이어서, 커뮤니티가 RULER([Hsieh et al., 2024](#))와 HELMET([Yen et al., 2025](#))와 같은 더 포괄적인 평가를 개발했습니다. 더 최근에 OpenAI는 긴 컨텍스트 평가의 난이도를 확장하는 [MRCR](#)과 [GraphWalks](#) 벤치마크를 릴리스했습니다.

긴 컨텍스트 평가의 한계와 현실적인 평가를 설계하는 방법에 대한 훌륭한 블로그 포스트도 참조하세요.

- **지시 따르기.** IFEval([J. Zhou et al., 2023](#))은 현재 지시 따르기를 측정하는 가장 인기 있는 평가이며, "검증 가능한 지시"에 대한 자동 점수를 사용합니다. IFBench([Pyatkin et al., 2025](#))는 IFEval보다 더 다양한 제약 세트를 포함하고 최근 모델 릴리스에서 발생한 일부 벤치마크를 완화하는 Ai2의 새로운 확장입니다. 다중 턴 지시 따르기의 경우 Multi-IF([He et al., 2024](#)) 또는 MultiChallenge([Sirdeshmukh et al., 2025](#))를 추천합니다.
- **정렬.** 모델이 사용자 의도에 얼마나 잘 정렬되는지 측정하는 것은 일반적으로 인간 주석자나 [LMArena](#)와 같은 공개 리더보드를 통해 수행됩니다. 자유 형식 생성, 스타일, 전반적인 유용성과 같은 품질은 자동화된 메트릭으로 정량적으로 측정하기 어렵기 때문입니다. 그러나 모든 경우에 이러한 평가를 실행하는 것은 매우 비용이 많이 들며, 이것이 커뮤니티가 인간 선호도의 프록시로 LLM을 사용하는 것에 의존하게 된 이유입니다. 이 종류의 가장 인기 있는 벤치마크에는 AlpacaEval([Dubois et al., 2025](#)), ArenaHard([T. Li et al., 2024](#)), MixEval([Ni et al., 2024](#))이 포함되며, 후자가 LMArena에서 인간 Elo 등급과 가장 강한 상관관계를 가집니다.
- **도구 호출.** [BFCL](#)은 도구 호출의 포괄적인 테스트를 제공하지만, 종종 꽤 빠르게 포화됩니다. TAU-Bench([Barres et al., 2025](#))는 시뮬레이션된 고객 서비스 설정에서 도구를 사용하고 사용자 문제를 해결하는 모델의 능력 테스트를 제공하며 보고하기에 인기 있는 벤치마크가 되었습니다.

3. 과적합 방지 평가

모델이 특정 기술에 과적합되는지 테스트하기 위해, GSMPlus([Q. Li et al., 2024](#))와 같은 일부 견고성 또는 적응성 평가를 세트에 포함합니다. GSM8k([Cobbe et al., 2021](#))의 문제를 교란하여 모델이 유사한 난이도의 문제를 여전히 풀 수 있는지 테스트합니다.

4. 내부 평가

공개 벤치마크가 모델 개발 중에 유용한 신호를 제공할 수 있지만, 특정 능력을 대상으로 하는 자체 내부 평가를 구현하거나 내부 전문가에게 모델과 상호작용하도록 요청하는 것을 대체할 수 없습니다.

AI 제품을 구축하는 경우 특히 그렇습니다. 이 주제에 대한 구체적인 조언은 Hamel Husain의 훌륭한 블로그 포스트를 참조하세요.

예를 들어, SmoLM3의 경우 모델이 **다중 턴 추론**이 가능한지 평가하는 벤치마크가 필요했으므로, 이를 측정하기 위한 Multi-IF의 변형을 구현했습니다.

5. 바이브 평가와 아레나

마찬가지로, 중간 체크포인트를 "바이브 테스트"하는 것(즉, 모델과 상호작용하는 것)이 평가 점수에 포함되지 않는 모델 동작의 미묘한 특이점을 발견하는 데 필수적임을 발견했습니다. 나중에 논의하듯이, 바이브 테스트는 모든 시스템 메시지가 코퍼스에서 삭제된 데이터 처리 코드의 버그를 발견했습니다! 이것은 또한 인기 있는 [LMArena](#)에서처럼 인간 선호도를 측정하기 위해 대규모로 수행할 수 있는 것입니다. 그러나 크라우드소싱된 인간 평가는 취약한 경향이 있어(실제 유용성보다 아첨과 화려한 언어를 선호), 낮은 신호 피드백으로 보는 것이 중요합니다.

👉 학습 데이터를 오염 제거하세요

공개 벤치마크에 의존하는 것의 한 가지 위험은 모델이 쉽게 과적합될 수 있다는 것입니다. 특히 합성 데이터가 목표 벤치마크와 유사한 프롬프트와 응답을 생성하는 데 사용될 때 그렇습니다. 이러한 이유로, 모델 개발을 안내하는 데 사용할 평가에 대해 학습 데이터를 오염 제거하는 것이 필수적입니다. [Open-R1](#)의 스크립트와 같은 N-gram 매칭으로 이를 수행할 수 있습니다.

SmoLM3의 경우 특히, 수학과 코드와 같은 인기 있는 도메인에서 지시를 안정적으로 따르고 잘 추론할 수 있는 하이브리드 추론 모델을 원했습니다. 또한 베이스 모델의 다국어 및 긴 컨텍스트 검색 능력을 보존하기를 원했습니다.

이것은 다음 평가 세트로 이어졌습니다:

벤치마크	카테고리	프롬프트 수	메트릭
AIME25	경쟁 수학	30	avg@64
LiveCodeBench (검증용 v4, 최종 릴리스용 v5)	경쟁 프로그래밍	100 (268)	avg@16
GPQA Diamond	대학원 수준 추론	198	avg@8
IFEval	지시 따르기	541	accuracy
MixEval Hard	정렬	1000	accuracy
BFCL v3	도구 사용	4441	mixed
Global MMLU (검증용 lite)	다국어 Q&A	590,000 (6,400)	accuracy
GSMPlus (검증용 mini)	견고성	10,000 (2,400)	accuracy
RULER	긴 컨텍스트	6,500	accuracy

이러한 평가가 실제로 무엇을 테스트하는지 구체적인 감각을 얻기 위해 각각의 몇 가지 예시 질문을 살펴보겠습니다:

HuggingFaceTB/post-training-benchmarks-viewer

Subset (9) aime25 · 5 rows	Split (1) test · 5 rows	
Q Search this dataset		
problem string · classes 5 values	answer string · classes 5 values	id string · classes 5 values
Let $f(x) = \frac{(x-18)(x-72)(x-98)(x-k)}{x}$. There exist exactly three positive real...	240	29
Find the sum of all positive integers n such that $n + 2$ divides the product $3(n + 3)(n^2 - ...)$	49	16
Let k be a real number such that the system $\begin{aligned} 25 + 20i - z &= 5 \\ z - 4 &= k \end{aligned}$ has a unique solution.	77	7
Let S be the set of vertices of a regular 24 -gon. Find the number of ways to draw 12 ...	113	25
Sixteen chairs are arranged in a row. Eight people each select a chair in which to sit so...	907	24

위의 예시를 탐색하여 각 벤치마크의 질문 유형을 확인하세요. 도메인의 다양성이 절제 실험 전반에 걸쳐 모델 능력의 다양한 측면을 테스트하고 있음을 보장하는 방법에 주목하세요.

우리가 작업하던 3B 모델 규모에서, 이러한 평가가 실행 가능한 신호를 제공하고, 학습 자체보다 빠르게 실행되며, 개선이 샘플링의 노이즈가 아니라 실제임을 확신하게 해줄 것이라고 느꼈습니다. 또한 베이스 모델 성능에서 너무 많이 회귀하지 않는지 확인하기 위해 사전학습 평가(전체 목록은 절제 실험 섹션 참조)를 추적했습니다.

👉 평가의 우선순위를 정하세요

위의 이야기는 팀으로 모여서 평가 세트에 수렴하고 모델을 학습시키기 전에 준비가 되어 있었다는 것을 시사합니다. 현실은 훨씬 더 지저분했습니다. 빠듯한 마감이 있었고 위의 많은 평가가 구현되기 전에 모델 학습을 서둘러 진행했습니다 (예: RULER는 모델 릴리스 며칠 전까지 사용할 수 없었습니다 🤦). 돌아보면, 이것은 실수였고 후속 학습 전반에 걸쳐 어떤 핵심 평가가 보존되어야 하는지 사전학습 팀과 논의하고 베이스 모델 학습이 완료되기 훨씬 전에 구현하는 것을 우선시했어야 했습니다. 다시 말해, 다른 모든 것보다 평가의 우선순위를 정하세요!

참여 규칙

수천 개의 모델을 평가하면서 얻은 몇 가지 힘들게 얻은 교훈으로 이 섹션을 요약하겠습니다.

- **모델 개발 중 평가를 가속화하기 위해 작은 부분집합을 사용하세요.** 예를 들어, LiveCodeBench v4는 v5와 높은 상관관계가 있지만 절반의 시간에 실행됩니다. 또는 전체 평가와 안정적으로 일치하는 가장 작은 프롬프트 부분집합을 찾으려는 tinyBenchmarks([Polo et al., 2024](#))의 방법을 사용하세요.
- **추론 모델의 경우, 점수가 매겨진 출력에서 사고의 연쇄를 제거하세요.** 이것은 거짓 양성을 제거하고 "50단어 미만으로 시를 쓰세요"와 같은 제약을 위반하는 응답에 페널티를 주는 IFEval과 같은 벤치마크에 직접 영향을 미칩니다.
- **평가가 LLM 심사자를 사용하는 경우, 시간에 따른 동일한 비교를 위해 심사자와 버전을 고정하세요.** 더 좋은 것은, 제공자가 심사자 모델을 더 이상 사용하지 않더라도 평가가 재현 가능하도록 오픈 가중치 모델을 사용하세요.
- **베이스 모델의 오염을 조심하세요.** 예를 들어, AIME 2025 이전에 릴리스된 대부분의 모델은 AIME 2024보다 훨씬 나쁜 성능을 보였으며, 이는 일부 벤치마크성이 작용했음을 시사합니다.
- **가능하면, 절제 실험 중에 사용된 모든 것을 테스트가 아닌 검증으로 취급하세요.** 이것은 Tulu3 평가 프레임워크 ([Lambert et al., 2025](#))와 유사하게 최종 모델 보고서를 위한 홀드아웃 벤치마크 세트를 유지하는 것을 의미합니다.
- **항상 공개 스위트에 대한 과적합을 잡기 위해 자체 데이터와 태스크에 대한 작은 "바이브 평가" 세트를 포함하세요.**
- **작은 수의 문제(일반적으로 ~2k 미만)가 있는 평가의 경우, k번 샘플링하고 avg@k 정확도를 보고하세요.** 이것은 개발 중 잘못된 결정으로 이어질 수 있는 노이즈를 완화하는 데 중요합니다.
- **새로운 평가를 구현할 때, 몇 가지 모델의 게시된 결과를 복제할 수 있는지 확인하세요(약간의 오차 내에서).** 이렇게 하지 않으면 구현을 수정하고 많은 체크포인트를 재평가해야 할 경우 나중에 많은 시간을 낭비하게 됩니다.
- **의심스러우면, 항상 평가 데이터로 돌아가고, 특히 모델에 무엇을 프롬프트하고 있는지 검사하세요.**

평가가 준비되었으니, 모델을 학습시킬 시간입니다! 그 전에 먼저 후속 학습 프레임워크를 선택해야 합니다.

도구들

모든 후속 학습 레시피 뒤에는 대규모 실험을 가능하게 하는 프레임워크와 라이브러리의 도구 상자가 있습니다. 각 프레임워크는 지원되는 알고리즘, 파인튜닝 방법, 확장성 기능의 고유한 세트를 제공합니다. 아래 표는 지도 파인튜닝(SFT)에서 선호도 최적화(PO) 및 강화 학습(RL)까지 주요 지원 영역을 요약합니다.

프레임워크	SFT	PO	RL	멀티모달	FullF1T	LoRA	분산
TRL	✓	✓	✓	✓	✓	✓	✓
Axolotl	✓	✓	✓	✓	✓	✓	✓
OpenInstruct	✓	✓	✓	✗	✓	✓	✓
Unslloth	✓	✓	✓	✓	✓	✓	✓
vERL	✓	✗	✓	✓	✓	✓	✓
Prime RL	✓	✗	✓	✗	✓	✓	✓
PipelineRL	✗	✗	✓	✗	✓	✓	✓
ART	✗	✗	✓	✗	✗	✓	✗
TorchForge	✓	✗	✓	✗	✓	✗	✓
NemoRL	✓	✓	✓	✗	✓	✗	✓
OpenRLHF	✓	✓	✓	✗	✓	✓	✓

여기서 **FullF1T**는 전체 파인튜닝을 의미하며, 학습 중 모든 모델 파라미터가 업데이트됩니다. **LoRA**는 **Low-Rank Adaptation**의 약자로, 베이스 모델을 고정하면서 작은 저랭크 행렬만 업데이트하는 파라미터 효율적 접근 방식입니다. **멀티모달**은 텍스트 이외의 모달리티(예: 이미지)에 대한 학습 지원 여부를 나타내고, **분산**은 둘 이상의 GPU에서 모델 학습이 가능한지 여부를 나타냅니다.

Hugging Face에서는 TRL을 개발하고 유지 관리하므로, 이것이 우리가 선택한 프레임워크이며 SmolLM3를 후속 학습시키는 데 사용한 것입니다.

💡 프레임워크를 포크하세요

분야의 빠른 속도를 감안할 때, TRL의 내부 포크에서 실험을 실행하는 것이 꽤 효과적임을 발견했습니다. 이를 통해 새로운 기능을 매우 빠르게 추가할 수 있으며, 나중에 메인 라이브러리로 업스트림됩니다. 프레임워크의 내부와 함께 작업하는 것이 편하다면, 유사한 워크플로우를 채택하는 것이 빠른 반복을 위한 강력한 접근 방식이 될 수 있습니다.

왜 프레임워크에 신경 써야 하나요?

학습 프레임워크의 사용을 한탄하고 대신 항상 모든 것을 처음부터 구현해야 한다고 주장하는 연구자 계층이 있습니다. 여기서 암묵적인 주장은 "진정한" 이해는 모든 RL 알고리즘을 재구현하거나, 모든 분산 학습 프리미티브를 수동으로 코딩하거나, 일회성 평가 하네스를 해킹하는 것에서만 온다는 것입니다.

그러나 이 입장은 현대 연구와 프로덕션의 현실을 무시합니다. 예를 들어 RL을 보세요. PPO와 GRPO와 같은 알고리즘은 올바르게 구현하기가 악명 높게 까다롭고([Huang et al., 2024](#)), 정규화나 KL 페널티의 작은 실수가 며칠간의 컴퓨팅과 노력 낭비로 이어질 수 있습니다.

마찬가지로, 어떤 알고리즘의 단일 파일 구현을 작성하는 것이 유혹적이지만, 그 동일한 스크립트가 1B에서 100B+ 파라미터로 스케일링할 수 있을까요?

프레임워크는 정확히 기본이 이미 잘 이해되어 있고 끝없이 재발명하는 것이 시간의 나쁜 사용이기 때문에 존재합니다. 저수준 땀질에 가치가 없다는 말은 아닙니다. 처음부터 PPO를 한 번 구현하는 것은 훌륭한 학습 연습입니다. 프레임워크 없이 토이 트랜스포머를 작성하면 어텐션이 실제로 어떻게 작동하는지 가르쳐줍니다. 그러나 대부분의 경우, 마음에 드는 프레임워크를 선택하고 목적에 맞게 해킹하세요.

그 불평을 마치고, 학습 실행을 종종 어디서 시작하는지 살펴보겠습니다.

왜 (거의) 모든 후속 학습 파이프라인이 SFT로 시작하나요

요즘 X에서 시간을 보내면, 강화 학습(RL)이 유일한 게임인 것처럼 생각할 것입니다. 매일 새로운 약어, [algorithmic tweaks](#), RL이 새로운 능력을 이끌어낼 수 있는지에 대한 열띤 토론([Chu et al., 2025; Yue et al., 2025](#))이 있습니다.

이 장의 나중에 보게 되듯이, RL은 정말로 작동하지만, 아래에서 논의하는 실용적인 트레이드오프가 있습니다.

RL은 물론 새로운 것이 아닙니다. OpenAI와 다른 연구소들은 초기 모델을 정렬하기 위해 인간 피드백에서의 RL(RLHF)([Lambert et al., 2022](#))에 크게 의존했지만, DeepSeek-R1([DeepSeek-AI, Guo, et al., 2025](#))의 릴리스 이후에야 RL 기반 후속 학습이 오픈소스 생태계에서 정말로 자리 잡았습니다.

그러나 한 가지는 변하지 않았습니다: 거의 모든 효과적인 후속 학습 파이프라인은 여전히 지도 파인튜닝(SFT)으로 시작합니다. 이유는 간단합니다:

- **저렴합니다:** SFT는 RL에 비해 적당한 컴퓨팅이 필요합니다. 실리콘의 모닥불을 태울 필요 없이 보통 의미 있는 이득을 얻을 수 있으며, RL에 필요한 시간의 일부로 가능합니다.
- **안정적입니다:** 보상 설계와 하이퍼파라미터에 악명 높게 민감한 RL과 달리, SFT는 "그냥 작동합니다."

- 올바른 기준선입니다: 좋은 SFT 체크포인트는 보통 원하는 대부분의 이득을 제공하며, DPO나 RLHF와 같은 이후 방법을 훨씬 더 효과적으로 만듭니다.

실제로 이것은 SFT가 쉽기 때문에 첫 번째 단계가 아니라는 것을 의미합니다; 더 복잡한 것을 시도하기 전에 일관되게 성능을 향상시키는 단계입니다. 베이스 모델로 작업할 때 특히 그렇습니다. 몇 가지 예외를 제외하고, 베이스 모델은 고급 후속 학습 방법의 이점을 얻기에는 너무 정제되지 않았습니다.

💡 DeepSeek R1-Zero는 어떤가요?

프론티어에서는 SFT로 시작하는 일반적인 이유가 항상 적용되지 않습니다. 종류할 더 강한 모델이 없고 인간 주식은 긴 사고의 연쇄와 같은 복잡한 동작에 너무 노이즈가 많습니다. 그래서 DeepSeek는 SFT를 건너뛰고 표준 감독으로 가르칠 수 없는 추론 동작을 발견하기 위해서 R1-Zero로 바로 RL로 갑습니다.

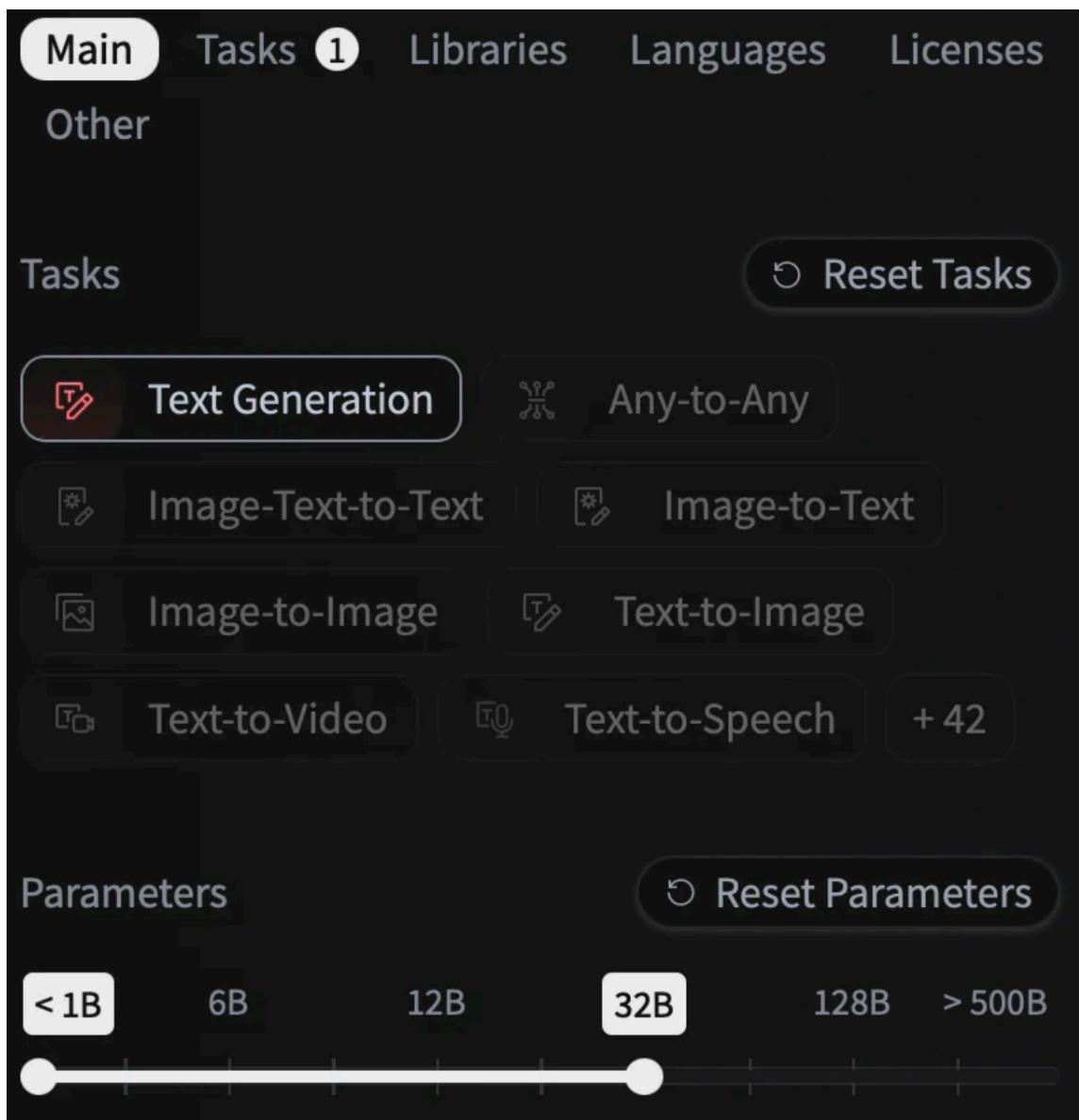
그 영역에 있다면, RL로 시작하는 것이 합리적일 수 있습니다. 그러나 거기서 작업하고 있다면... 어차피 이 블로그 포스트를 읽고 있지 않을 것입니다 😊.

따라서 SFT가 대부분의 파이프라인이 시작하는 곳이라면, 다음 질문은: 무엇을 파인튜닝해야 하나요? 그것은 올바른 베이스 모델을 선택하는 것으로 시작합니다.

베이스 모델 선택

후속 학습을 위한 베이스 모델을 선택할 때, 몇 가지 실용적인 차원이 가장 중요합니다:

- **모델 크기:** smol 모델이 시간이 지남에 따라 극적으로 개선되었지만, 오늘날에도 여전히 더 큰 모델이 더 잘 일반화하고, 종종 더 적은 샘플로 그렇습니다. 학습 후 모델을 사용하거나 배포할 계획에 대표적인 모델 크기를 선택하세요. Hugging Face Hub에서 모달리티와 크기로 모델을 필터링하여 적합한 후보를 찾을 수 있습니다:



- **아키텍처 (MoE vs 밀집):** MoE 모델은 토큰당 파라미터의 부분집합을 활성화하고 컴퓨팅 단위당 더 높은 용량을 제공합니다. 대규모 서빙에 좋지만, 우리 경험상 파인튜닝하기가 더 까다롭습니다. 반면, 밀집 모델은 학습하기가 더 간단하고 종종 더 작은 규모에서 MoE를 능가합니다.
- **후속 학습 실적:** 벤치마크가 유용하지만, 베이스 모델이 이미 커뮤니티에서 공감하는 강력한 후속 학습된 모델 모음을 생성했다면 더 좋습니다. 이것은 모델이 잘 학습되는지에 대한 프록시를 제공합니다.

LocalLLaMa 서브레딧은 새로운 모델의 전반적인 분위기를 이해하기에 좋은 곳입니다. Artificial Analysis와 LMArena도 새로운 모델의 독립적인 평가를 제공하지만, 이러한 플랫폼은 때때로 모델 제공자에 의해 벤치맥싱됩니다.

우리 경험상 Qwen, Mistral, DeepSeek의 베이스 모델이 후속 학습에 가장 적합하며, Qwen이 분명한 선호입니다. 각 모델 시리즈가 일반적으로 넓은 파라미터 범위를 다루기 때문입니다(예: Qwen3 모델은 0.6B에서 235B까지 크기가 다양합니다!). 이 기능은 스케일링을 훨씬 더 간단하게 만듭니다.

배포 필요에 맞는 베이스 모델을 선택했으면, 다음 단계는 핵심 기술을 탐색하기 위한 간단하고 빠른 SFT 기준선을 확립하는 것입니다.

간단한 기준선 학습

SFT의 경우, 좋은 기준선은 학습이 빠르고, 모델의 핵심 기술에 집중하며, 특정 능력이 기준에 미치지 못할 때 더 많은 데이터로 확장하기가 간단해야 합니다. 초기 기준선에 어떤 데이터셋을 사용할지 선택하는 것은 약간의 취향과 고품질 가능성 있는 데이터셋에 대한 친숙함을 포함합니다. 일반적으로 학술 벤치마크에서 높은 점수를 보고하는 공개 데이터셋에 과도하게 의존하지 말고 대신 [OpenHermes](#)와 같은 훌륭한 모델을 학습시키는 데 사용된 데이터셋에 집중하세요. 예를 들어, SmoILM1 개발에서 처음에 [WebInstruct](#)에서 SFT를 실행했는데, 이것은 종이 위에서는 훌륭한 데이터셋입니다. 그러나 바이브 테스트 중에 모델이 "어떻게 지내세요?"와 같은 간단한 인사에 방정식으로 응답하기 때문에 너무 과학에 집중되어 있음을 발견했습니다.

학습 데이터의 특이점을 발견하기 위한 바이브 테스트의 사용은 이 장에서 반복되는 주제입니다— 모델과 그냥 대화하는 것의 힘을 과소평가하지 마세요!

이로 인해 소형 모델에 기본 채팅 능력을 주입하는 데 중요한 것으로 밝혀진 Everyday Conversations 데이터셋을 만들게 되었습니다.

SmoILM3의 경우, 하이브리드 추론 모델을 학습시키기로 했고 처음에 추론, 지시 따르기, 조종 가능성을 대상으로 하는 작은 데이터셋 세트를 선택했습니다. 아래 표는 각 데이터셋의 통계를 보여줍니다:

데이터셋	추론 모드	# 예시	% 예시	# 토큰 (M)	% 토큰	예시당 평균 # 토큰	컨텍스트의 평균 # 토큰	응답의 평균 # 토큰	평균 # 턴
Everyday Conversations	/no_think	2,260	2.3	0.6	0.8	260.2	222.3	94.0	7.8
SystemChats 30k	/no_think	33,997	35.2	21.5	28.2	631.9	422.8	267.7	6.3
Tulu 3 SFT Personas IF	/no_think	29,970	31.0	13.3	17.5	444.5	119.8	380.7	2
Everyday Conversations (Qwen3-32B)	/think	2,057	2.1	3.1	4.1	1,522.4	376.8	1,385.6	4
SystemChats 30k (Qwen3-32B)	/think	27,436	28.4	29.4	38.6	1070.8	84.6	1,042.7	2
s1k-1.1	/think	835	0.9	8.2	10.8	8,859.3	370.9	9,728.5	2
총계	-	96,555	100.0	76.1	100.0	2,131.5	266.2	2,149.9	4.0

하이브리드 추론 기준선을 위한 데이터 혼합

SmoILM3 개발 전반에 걸쳐 배웠듯이, 하이브리드 추론 모델 학습은 표준 SFT보다 까다롭습니다. 데이터셋을 그냥 함께 혼합할 수 없고; 모드 간에 데이터를 짹지어야 합니다. 각 예시는 모델이 확장된 추론에 참여해야 하는지 간결한 답변을 제공해야 하는지 명확하게 나타내야 하며, 이상적으로는 언제 모드를 전환해야 하는지 가르치는 병렬 예시가 필요합니다. 위 표에서 주목할 또 다른 점은 예시가 아닌 토큰 측면에서 데이터 혼합의 균형을 맞춰야 한다는 것입니다: 예를 들어, s1k-1.1 데이터셋은 총 예시의 ~1%이지만 긴 추론 응답으로 인해 총 토큰의 ~11%를 차지합니다.

이것은 우리가 가장 신경 쓰는 기술에 걸쳐 기본 커버리지를 제공했지만, 새로운 도전도 도입했습니다: 각 데이터셋은 확장된 사고를 활성화해야 하는지 여부에 따라 다르게 형식화되어야 했습니다. 이러한 형식을 통합하기 위해 일관된 채팅 템플릿이 필요했습니다.

좋은 채팅 템플릿 선택

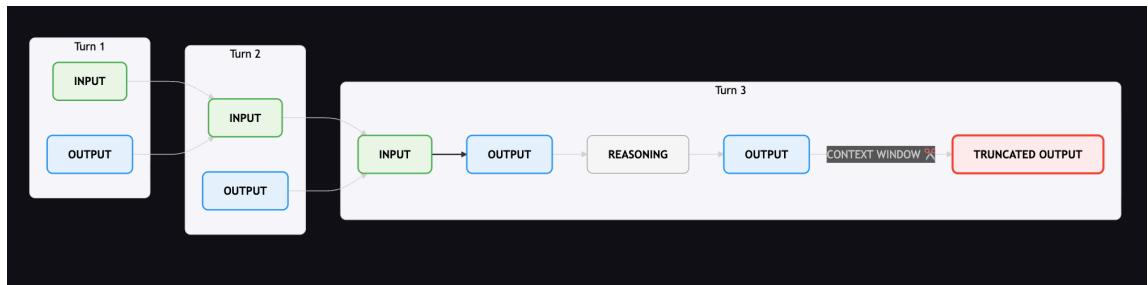
채팅 템플릿을 선택하거나 설계할 때, 모든 상황에 맞는 단일 정답은 없습니다. 실제로 미리 물어볼 가치가 있는 몇 가지 질문이 있음을 발견했습니다:

- **사용자가 시스템 역할을 커스터마이즈할 수 있나요?** 사용자가 자신만의 시스템 프롬프트를 정의할 수 있어야 한
다면(예: "해적처럼 행동하세요"), 템플릿이 이를 깔끔하게 처리해야 합니다.
- **모델에 도구가 필요한가요?** 모델이 API를 호출해야 한다면, 템플릿이 도구 호출과 응답을 위한 구조화된 출력을
수용해야 합니다.
- **추론 모델인가요?** 추론 모델은 모델의 "생각"을 최종 답변과 분리하기 위해 <think> ... </think> 와 같은 템
플릿을 사용합니다. 일부 모델은 대화의 턴 간에 추론 토큰을 버리며, 채팅 템플릿이 그로직을 처리해야 합니다.
- **추론 엔진과 작동할까요?** vLLM과 SGLang과 같은 추론 엔진에는 추론과 도구를 위한 전용 파서가 있습니다. 이
러한 파서와의 호환성은 나중에 많은 고통을 절약합니다. 특히 일관된 도구 호출이 필수적인 복잡한 에이전트 벤
치마크에서요.

아래 표는 몇 가지 인기 있는 채팅 템플릿과 주요 고려 사항에서 어떻게 비교되는지 보여줍니다:

채팅 템플릿	시스템 역할 커스터마이즈	도구	추론	추론 호환성	참고
ChatML	✓	✓	✗	✓	간단하고 대부분의 사용 사례에 좋음.
Qwen3	✓	✓	✓	✓	하이브리드 추론 템플릿
DeepSeek-R1	✗	✗	✓	✓	<think>로 추론 콘텐츠를 프리필.
Llama 3	✓	✓	✗	✓	Python 코드 인터프리터와 같은 내장 도구가 있음.
Gemma 3	✓	✗	✗	✗	시스템 역할 커스터마이즈는 첫 번째 사용자 턴에서 정의됨.
Command A Reasoning	✓	✓	✓	✗	모델당 여러 채팅 템플릿.
GPT-OSS	✓	✓	✓	✓	Harmony 응답 형식 기반. 복잡하지만 다재다능함.

대부분의 경우, ChatML이나 Qwen의 채팅 템플릿이 시작하기에 훌륭한 곳임을 발견했습니다. SmoLM3의 경우, 하이브리드 추론을 위한 템플릿이 필요했고 Qwen3가 우리가 신경 쓰는 차원에서 좋은 균형을 맞춘 몇 안 되는 템플릿 중 하나임을 발견했습니다. 그러나 완전히 만족하지 못한 한 가지 특이점이 있었습니다: 추론 콘텐츠가 대화의 마지막 턴을 제외한 모든 턴에서 버려집니다. 아래 그림에서 보여주듯이, 이것은 OpenAI의 추론 모델이 작동하는 방식 [OpenAI's reasoning models work](#)과 유사합니다:



추론에서는 이것이 합리적이지만(컨텍스트가 폭발하는 것을 피하기 위해), 학습에서는 모델을 적절히 조건화하기 위해 모든 턴에 걸쳐 추론 토큰을 유지하는 것이 중요하다고 결론지었습니다.

대신, 다음 기능을 가진 자체 채팅 템플릿을 만들기로 결정했습니다.

- [Llama 3](#)와 독점 모델에서 [탈옥된 것과 같은 구조화된 시스템 프롬프트](#). 시스템 프롬프트를 완전히 재정의할 수 있는 유연성도 제공하고 싶었습니다.
- JSON 도구 호출 대신 임의의 Python 코드를 실행하는 [코드 에이전트 지원](#).
- 시스템 메시지를 통한 추론 모드의 명시적 제어.

채팅 템플릿의 설계를 반복하기 위해 [Chat Template Playground](#)를 사용했습니다. 이 편리한 애플리케이션은 Hugging Face의 동료들이 개발했으며 메시지가 어떻게 렌더링되는지 미리보고 형식 문제를 디버그하기 쉽게 해줍니다. 다음은 직접 시도해볼 수 있도록 플레이그라운드의 임베디드 버전입니다:

The screenshot shows the Chat Template Playground interface. On the left, under 'Chat template', there's a section titled 'HuggingFaceTB/SmolLM3-3B' with a 'change model' button. Below it is a code editor containing a Jinja-like template with variables like `enable_thinking` and `reasoning_mode`. On the right, there are two panes: 'JSON Input' and 'Rendered Output'. The 'JSON Input' pane contains a JSON message structure with 'role' (system, user, assistant) and 'content' fields. The 'Rendered Output' pane shows the expanded template with system metadata, reasoning mode, and custom instructions.

```
1 {# ----- defaults ----- #}
2 {%- if enable_thinking is not defined -%}
3 | {%- set enable_thinking = true -%}
4 {%- endif -%}
5 {# ----- reasoning mode ----- #}
6 {%- if enable_thinking -%}
7 | {%- set reasoning_mode = "/think" -%}
8 {%- else -%}
9 | {%- set reasoning_mode = "/no_think" -%}
10 {%- endif -%}
11 {# ----- header (system message) ----- #}
12 {{- "<|im_start|>system\n" -}}
13 {%- if messages[0].role == "system" -%}
14 | {%- set system_message = messages[0].content -%}
15 | {%- if "/no_think" in system_message -%}
16 | | {%- set reasoning_mode = "/no_think" -%}
17 | {%- elif "/think" in system_message -%}
18 | | {%- set reasoning_mode = "/think" -%}
19 | {%- endif -%}
20 | {%- set custom_instructions =
system_message.replace("/no_think",
 "")).replace("/think", "").rstrip() -%}
21 {%- endif -%}
22 {%- if "/system_override" in system_message -%}
23 | {%- if
custom_instructions.replace("/system_override",
```

```
1   {
2     "messages": [
3       {
4         "role": "system",
5         "content": "You are a helpful assistant.",
6       },
7       {
8         "role": "user",
9         "content": "Hello, how are you?",
10      },
11      {
12        "role": "assistant",
```

```
<|im_start|>system
## Metadata

Knowledge Cutoff Date: June 2025
Today Date: 11 1월 2026
Reasoning Mode: /think

## Custom Instructions

You are a helpful assistant.

<|im_start|>user
```

드롭다운에서 다른 예시를 선택하여 다중 텐 대화, 추론, 도구 사용에서 채팅 템플릿이 어떻게 작동하는지 확인하세요.

JSON 입력을 수동으로 변경하여 다른 동작을 활성화할 수도 있습니다. 예를 들어, `enable_thinking: false`를 제공하거나 시스템 메시지에 `/no_think`를 추가하면 어떻게 되는지 확인해보세요.

초기 데이터셋과 채팅 템플릿을 결정했으면, 기준선을 학습할 시간입니다!

Baby baselines

최적화에 뛰어들어 모든 성능 포인트를 쥐어짜기 전에, 몇 가지 "베이비 기준선"을 확립해야 합니다. 이 기준선은 최신 기술에 도달하는 것이 아니라(아직은), 채팅 템플릿이 원하는대로 작동하고 초기 하이퍼파라미터 세트가 안정적인 학습을 생성하는지 검증하는 것을 목표로 합니다. 이 기반을 갖춘 후에야 하이퍼파라미터와 학습 혼합을 많이 튜닝하기 시작합니다.

SFT 기준선을 학습할 때, 고려해야 할 주요 사항은 다음과 같습니다.

- 전체 파인튜닝(FullFT)을 사용할 건가요, 아니면 LoRA나 QLoRA와 같은 파라미터 효율적 방법을 사용할 건가
요? Thinking Machines의 훌륭한 [blog post](#)에서 설명했듯이, LoRA는 특정 조건(보통 데이터셋의 크기에 의해 결정됨)에서 FullFT와 일치할 수 있습니다.

- 어떤 유형의 병렬성이 필요한가요? 소형 모델이나 LoRA로 학습된 모델의 경우, 보통 데이터 병렬로 충분합니다. 더 큰 모델의 경우 모델 가중치와 옵티마이저 상태를 공유하기 위해 FSDP2나 DeepSpeed ZeRO-3가 필요합니다. 긴 컨텍스트로 학습된 모델의 경우, [context parallelism](#)과 같은 방법을 사용하세요.
- 하드웨어가 지원한다면 FlashAttention과 Liger와 같은 커널을 사용하세요. 이러한 커널 중 많은 것이 Hugging Face Hub에 호스팅되어 있으며 TRL에서 [simple argument](#)를 통해 설정하여 VRAM 사용량을 극적으로 낮출 수 있습니다.
- [어시스턴트 토큰에서만 학습하도록 손실을 마스킹하세요.](#) 아래에서 논의하듯이, 채팅 템플릿의 어시스턴트 턴을 특별한 {% generation %} 키워드로 감싸서 이를 달성할 수 있습니다.
- [학습률을 튜닝하세요.](#) 데이터 외에, 이것이 모델이 "그저 그런" 것인지 "훌륭한" 것인지를 결정하는 가장 중요한 요소입니다.
- [학습 샘플을 패킹하고 데이터 분포에 맞게 시퀀스 길이를 튜닝하세요.](#) 이것은 학습 속도를 극적으로 높입니다. TRL에는 이를 위한 편리한 [application](#)이 있습니다.

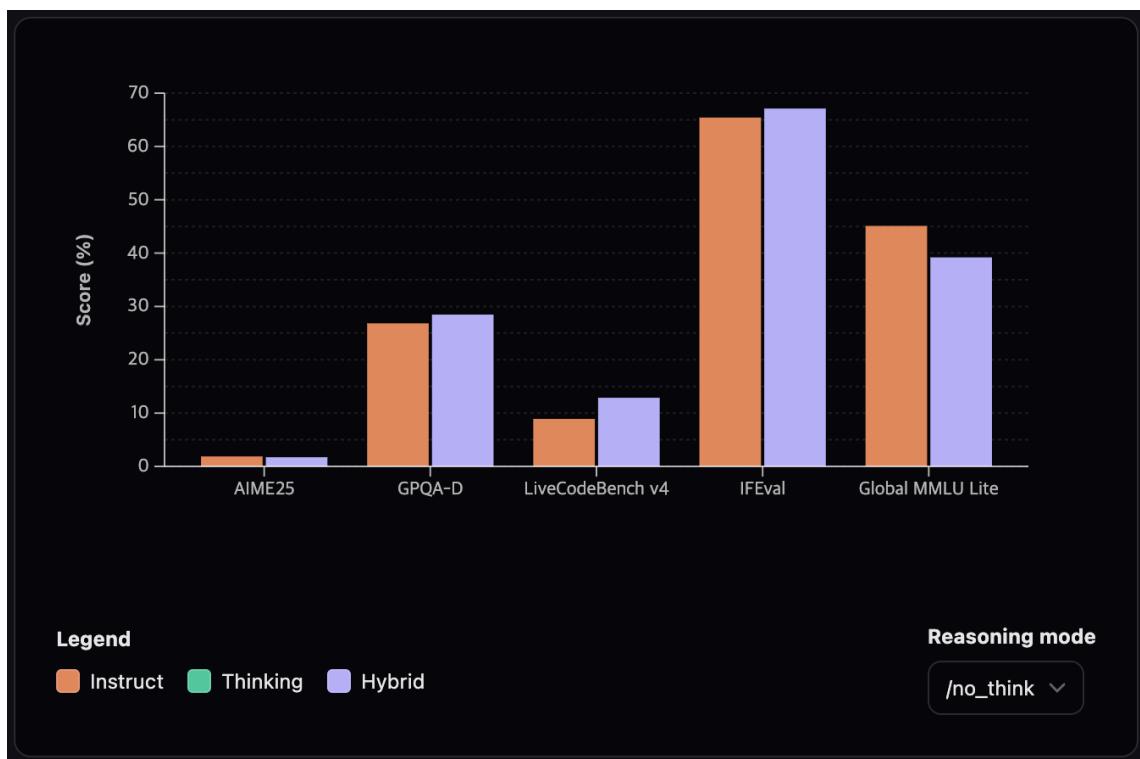
SmolLM3에서 이러한 선택이 어떻게 전개되었는지 살펴보겠습니다. 첫 번째 기준선 실험에서는 간단한 정상성 검사를 원했습니다: 채팅 템플릿이 실제로 하이브리드 추론을 이끌어내는가? 이를 테스트하기 위해, [표](#)에서 세 가지 데이터 혼합을 비교했습니다.

- **Instruct:** 비추론 예시에서 학습.
- **Thinking:** 추론 예시에서 학습.
- **Hybrid:** 모든 예시에서 학습.

각 혼합에 대해, 학습률 1e-5, 유효 배치 크기 128로 FullIFT를 사용하여 [SmolLM3-3B-Base](#)에서 SFT를 실행하고, 1에 포크 동안 학습했습니다.

대부분의 모델과 데이터셋에서 이러한 하이퍼파라미터 선택이 기준선으로 잘 작동함을 발견했습니다.

이것이 작은 데이터셋이므로, 패킹을 사용하지 않았고, Instruct 부분집합의 경우 시퀀스를 8,192 토큰으로, 나머지는 32,768 토큰으로 제한했습니다. 8 x H100 1노드에서 이 실험은 빠르게 실행되었으며, 부분집합에 따라 30-90분이 걸렸습니다. 아래 그림은 해당 추론 모드에 대한 각 부분집합의 성능을 비교합니다:



이러한 결과는 하이브리드 모델이 일종의 "분리된 뇌"를 나타내며, 한 주론 모드의 데이터 혼합이 다른 모드에 거의 영향을 미치지 않음을 빠르게 보여주었습니다. 이것은 대부분의 평가가 Instruct, Thinking, Hybrid 부분집합 간에 유사한 점수를 가지며, LiveCodeBench v4와 IFEval이 하이브리드 데이터가 전반적인 성능을 향상시키는 예외임을 통해 명백합니다.

Vibe-test your baselines

평가가 괜찮아 보였지만, 하이브리드 모델이 다른 페르소나(예: 해적처럼)로 행동하도록 시도했을 때, 시스템 메시지에 넣은 모든 것을 일관되게 무시했습니다. 약간의 조사 후에, 이유가 데이터를 형식화한 방식 때문임을 발견했습니다:

lewis Jun 28th at 10:14 AM
 Ah crap, I think I know why `v18.00` also couldn't follow the system prompt in `/no_think` mode: it appears to not have been included as a `custom_instruction` in `all_decontaminated_formatted` 😞

무슨 일이 일어났냐면, 채팅 템플릿 설계에서 시스템 프롬프트를 저장하기 위해 custom_instructions 인수를 노출했습니다. 예를 들어, 대화에서 페르소나를 설정하는 방법은 다음과 같습니다:

```
from transformers import AutoTokenizer
```

```

tok = AutoTokenizer.from_pretrained("HuggingFaceTB/SmoLLM3-3B")

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 challenge sometimes [TRUNCATED]",
        "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

## Knowledge Cutoff Date: June 2025
## Today Date: 28 October 2025
## Reasoning Mode: /no_think

### Custom Instructions

## You are a vampire technologist

## <|im_start|>user
## I'm trying to set up my iPhone, can you help?<|im_end|>
## <|im_start|>assistant
## <think>

## </think>
## Of course, even as a vampire, technology can be a bit of a challenge sometimes # [TRUNCATED]
<|im_end|>

```

문제는 우리의 데이터 샘플이 이렇게 생겼다는 것이었습니다:

```

{
    "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 challenge sometimes [TRUNCATED]",

```

```

        "role": "assistant",
    },
],
"chat_template_kwargs": {
    "custom_instructions": None,
    "enable_thinking": False,
    "python_tools": None,
    "xml_tools": None,
},
}

```

처리 코드의 버그가 custom_instructions를 None으로 설정했고, 이것이 효과적으로 모든 단일 학습 샘플에서 시스템 메시지를 제거했습니다 😱 ! 따라서 이러한 학습 샘플에 대해 좋은 페르소나를 얻는 대신, SmoLLM3 기본 시스템 프롬프트로 끝났습니다:

```

chat_template_kwargs = {"custom_instructions": None, "enable_thinking": False}
rendered_input = tok.apply_chat_template(messages, tokenize=False, **chat_template_kwargs)
print(rendered_input)
## <|im_start|>system
#### Metadata

## Knowledge Cutoff Date: June 2025
## Today Date: 28 October 2025
## Reasoning Mode: /no_think

#### Custom Instructions

## You are a helpful AI assistant named SmoLLM, trained by Hugging Face.

## <|im_start|>user
## I'm trying to set up my iPhone, can you help?<|im_end|>
## <|im_start|>assistant
## <think>

## </think>
## Of course, even as a vampire, technology can be a bit of a challenge sometimes [TRUNCATED]
<|im_end|>

```

이것은 SystemChats 부분집합에서 특히 문제가 되었는데, 모든 페르소나가 custom_instructions를 통해 정의되어 모델이 대화 중간에 무작위로 캐릭터를 바꾸는 경향이 있었습니다. 이것은 다음 규칙으로 이어집니다:

👉 규칙

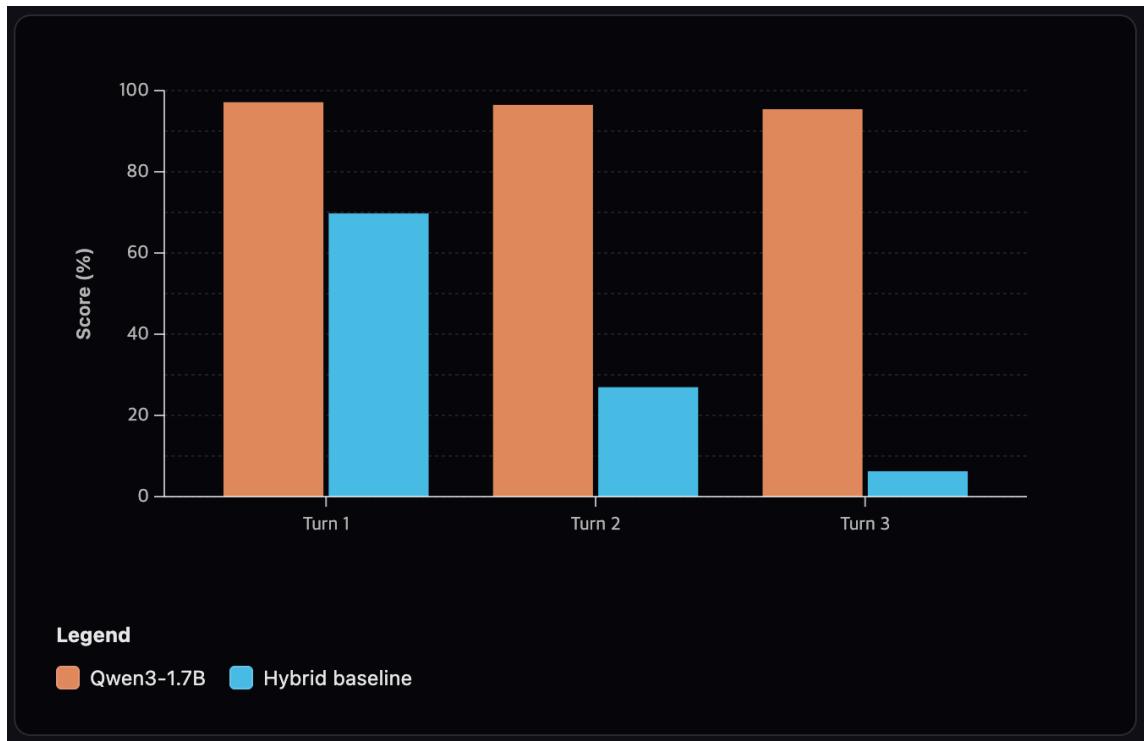
평가가 괜찮아 보여도 항상 모델을 바이브 테스트하세요. 더 자주, 학습 데이터의 미묘한 버그를 발견하게 될 것입니다.

이 버그를 수정해도 평가에는 영향이 없었지만, 마침내 채팅 템플릿과 데이터셋 형식이 작동하고 있다고 확신했습니다. 설정이 안정되고 데이터 파이프라인이 확인되면, 다음 단계는 특정 능력 개발에 집중하는 것입니다.

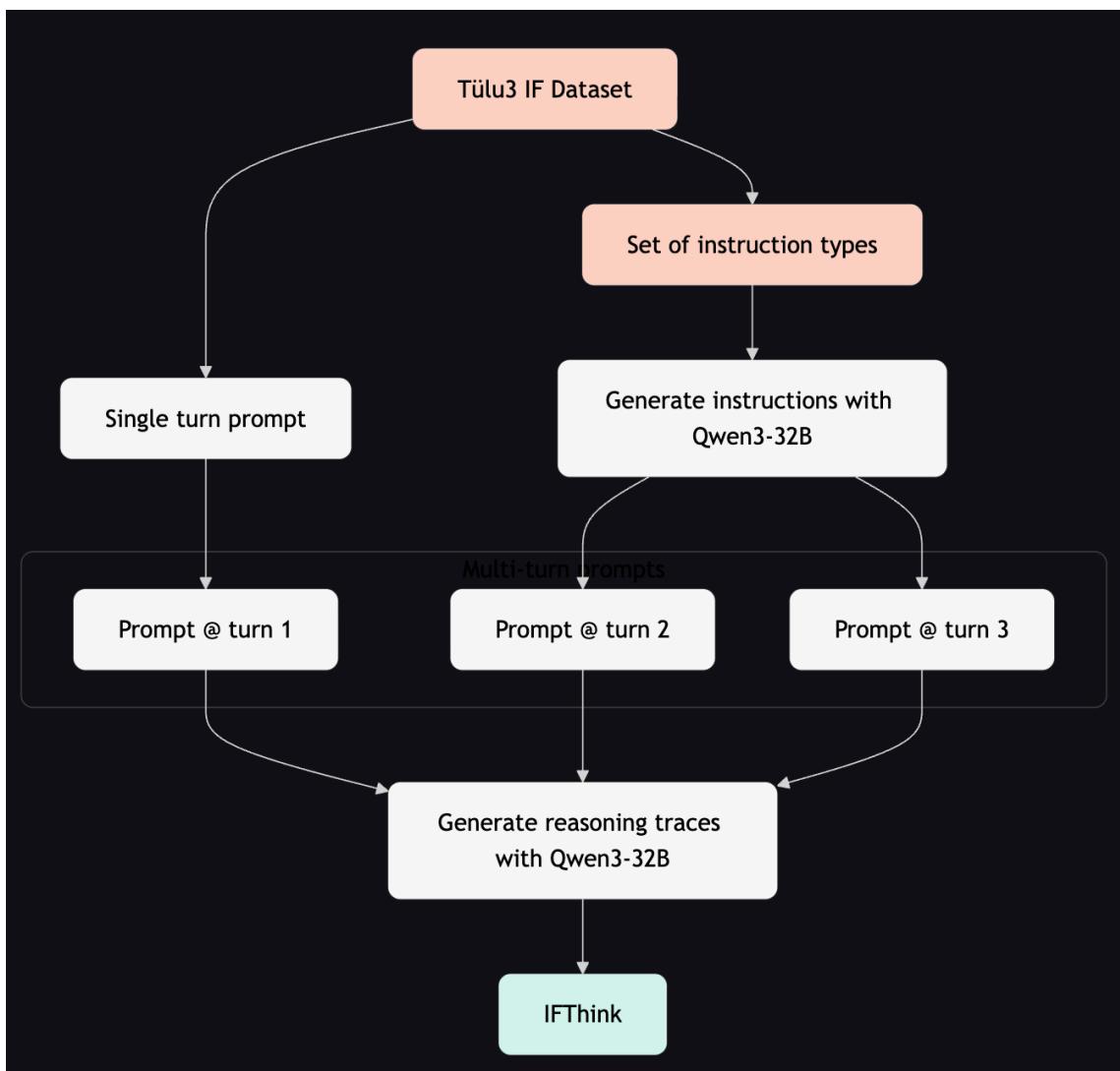
TARGETING SPECIFIC CAPABILITIES

[Open-R1](#) 개발 중에, 베이스 모델을 전적으로 단일 턴 추론 데이터에서 학습시키면 다중 턴으로 일반화하지 못한다는 것을 발견했습니다. 이것은 놀랍지 않습니다; 그러한 예시가 없으면, 모델은 학습 분포 밖에서 테스트되고 있습니다.

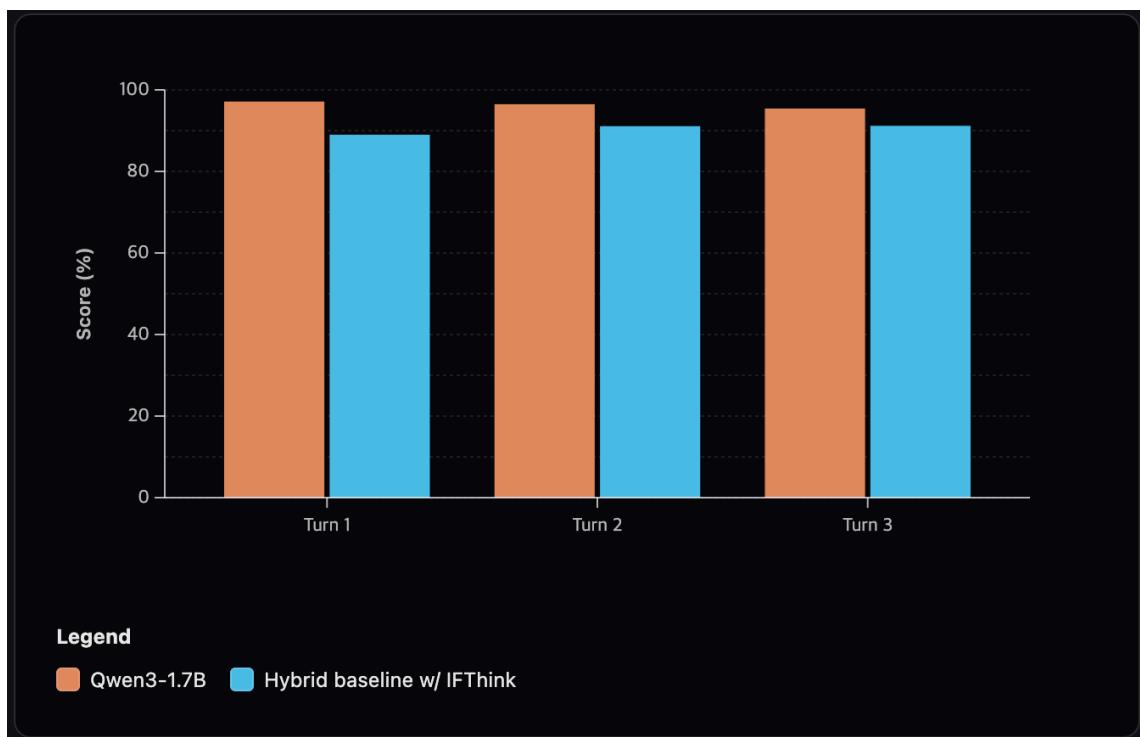
SmoLM3에 대해 이를 정량적으로 측정하기 위해, **ThinkFollow**라는 내부 평가를 개발한 Qwen3에서 영감을 얻었습니다. 이것은 모델이 추론 모드를 일관되게 전환할 수 있는지 테스트하기 위해 /think 또는 /no_think 태그를 무작위로 삽입합니다. 우리 구현에서는 Multi-IF의 프롬프트를 가져온 다음 모델이 <think> 와 </think> 태그로 둘러싸인 빈 또는 비어 있지 않은 think 블록을 생성했는지 확인했습니다. 예상대로, 하이브리드 기준선의 결과는 모델이 첫 번째 턴 이후에 추론 모드를 활성화하는 데 처참하게 실패함을 보여주었습니다:



이 능력을 수정하기 위해, IFTthink라는 새로운 데이터셋을 구성했습니다. Multi-IF 파이프라인을 기반으로, [Tulu 3의 지시 따르기 부분집합](#)에서 단일 턴 지시를 사용하고 Qwen3-32B를 사용하여 검증 가능한 지시와 추론 트레이스를 모두 생성하여 다중 턴 교환으로 확장했습니다. 방법은 아래에 설명되어 있습니다:



이 데이터를 기준선 혼합에 포함시키자 극적인 개선이 나타났습니다:



다중 턴 추론 문제를 IFThink로 수정한 후, 기준선이 마침내 의도한 대로 동작했습니다; 턴 간에 일관성을 유지하고, 지시를 따르고, 채팅 템플릿을 올바르게 사용할 수 있었습니다. 그 기반이 마련되자, 기본으로 돌아갔습니다: 학습 설정 자체를 튜닝하는 것입니다.

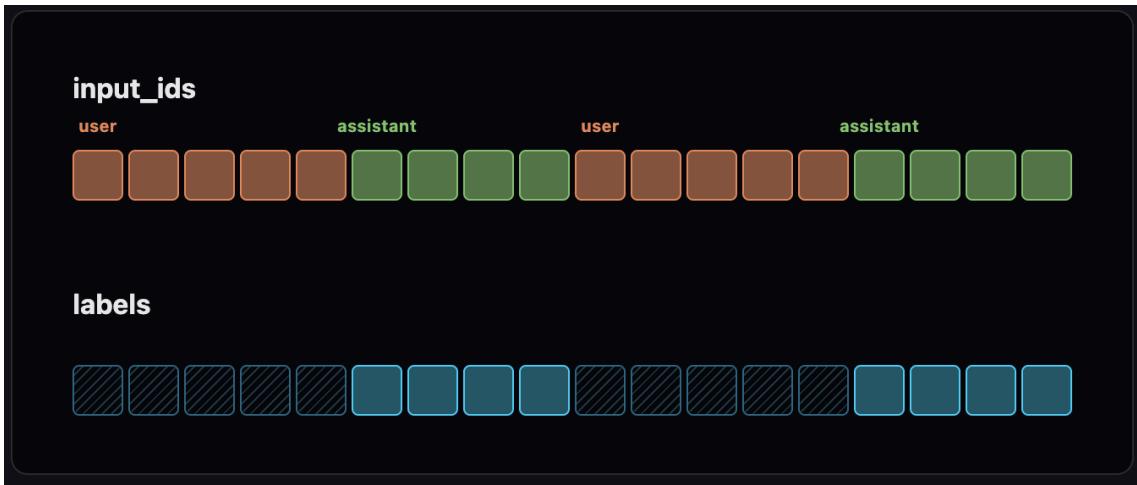
어떤 하이퍼파라미터가 실제로 중요한가요?

SFT에서 실제로 중요한 하이퍼파라미터는 몇 개뿐입니다. 학습률, 배치 크기, 패킹이 모델이 얼마나 효율적으로 학습하고 얼마나 잘 일반화하는지에 대해 거의 모든 것을 결정합니다. 베이비 기준선에서는 데이터와 채팅 템플릿을 검증하기 위해 합리적인 기본값을 선택했습니다. 이제 설정이 안정되었으므로, 이러한 선택이 기준선에 얼마나 영향을 미치는지 보기 위해 다시 검토했습니다.

사용자 턴 마스킹

채팅 템플릿에 대한 미묘한 설계 선택 중 하나는 학습 중에 사용자 턴을 마스킹할지 여부입니다. 대부분의 채팅 스타일 데이터셋에서, 각 학습 예시는 번갈아 나오는 사용자와 어시스턴트 메시지로 구성됩니다(가능하면 인터리브된 도구 호출과 함께). 모델을 모든 토큰을 예측하도록 학습시키면, 고품질 어시스턴트 응답을 생성하는 데 집중하는 것이 아니라 효과적으로 사용자 쿼리를 자동 완성하는 것을 배웁니다.

아래 그림에서 보여주듯이, 사용자 턴을 마스킹하면 모델의 손실이 사용자 메시지가 아닌 어시스턴트 출력에서만 계산되도록 보장하여 이를 방지합니다:



TRL에서 마스킹은 어시스턴트 토큰 마스크를 반환할 수 있는 채팅 템플릿에 적용됩니다. 실제로, 이것은 템플릿에 `{% generation %}` 키워드를 다음과 같이 포함하는 것을 수반합니다:

```
{%- 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 %}
```

그런 다음, `apply_chat_template()` or `return_assistant_tokens_mask=True` 와 함께 사용되면 채팅 템플릿이 대화의 어떤 부분이 마스킹되어야 하는지 나타냅니다. 다음은 어시스턴트 토큰에 ID 1이 주어지고 사용자 토큰은 ID 0으로 마스킹되는 방법을 보여주는 간단한 예시입니다:

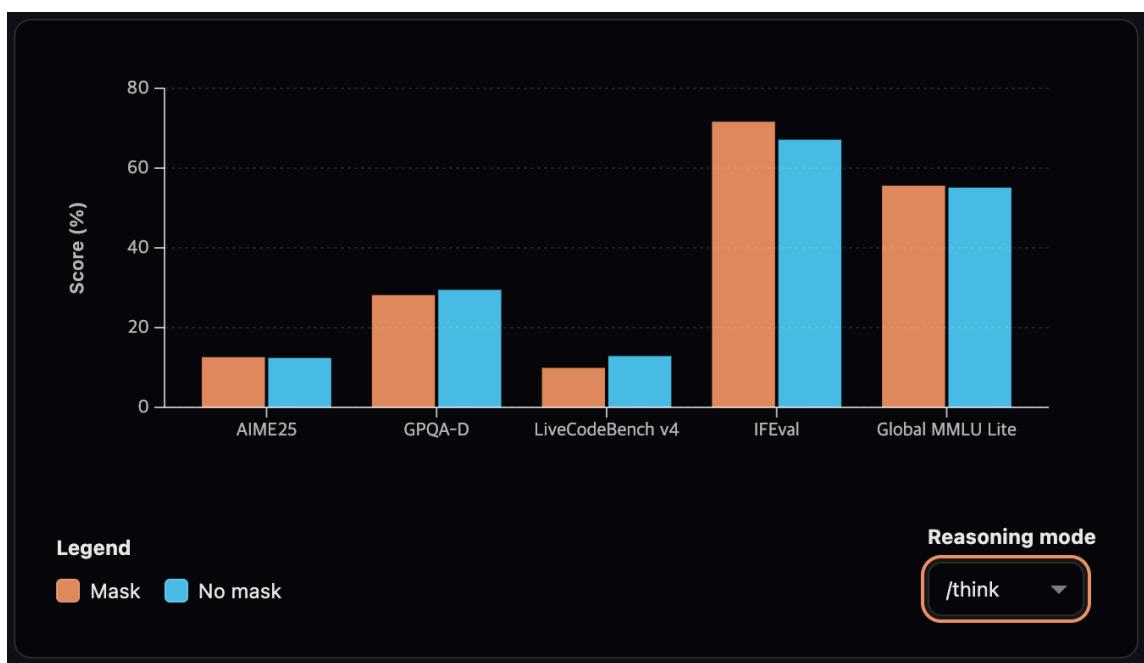
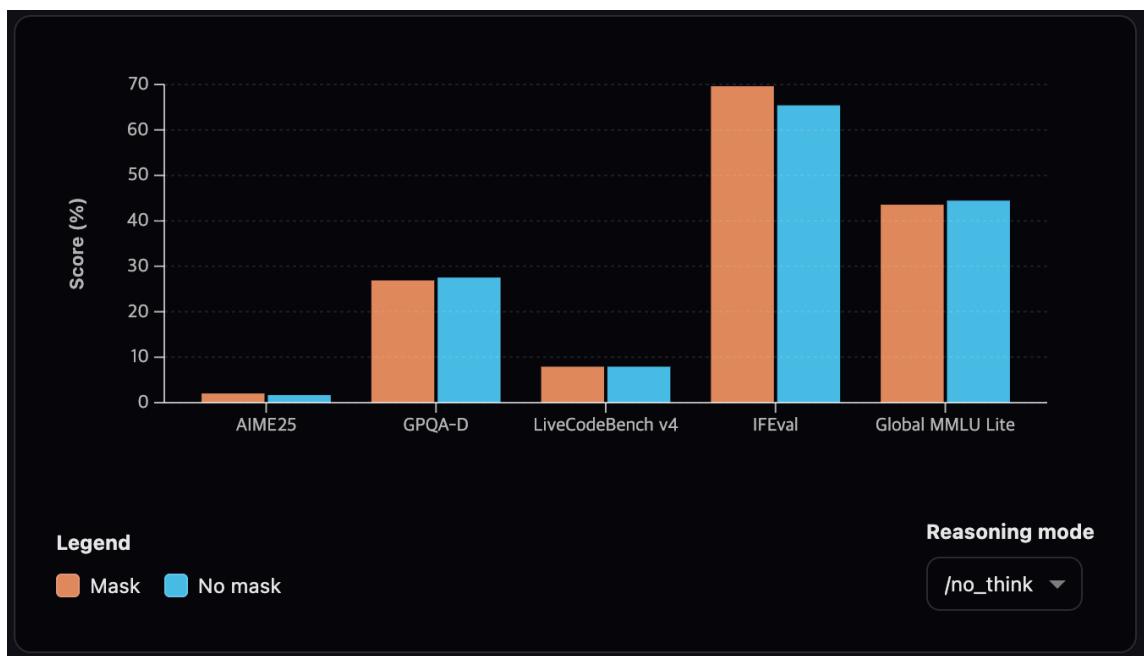
```
chat_template = '''
{%- for message in messages -%}
{%- if message.role == "user" -%}
{{ "<|im_start|>" + message.role + "\n" + message.content + "<|im_end|>\n" }}
{%- elif message.role == "assistant" -%}
{%- generation %}
{{ "<|im_start|>assistant" + "\n" + message.content + "<|im_end|>\n" }}
{%- endgeneration %}
{%- endif %}
{%- endfor %}
{%- if add_generation_prompt %}
{{ "<|im_start|>assistant\n" }}
```

```

{%- endif %}
```
rendered_input = tok.apply_chat_template(messages, chat_template=chat_template,
return_assistant_tokens_mask=True, return_dict=True)
print(rendered_input)
{'input_ids': [128011, 882, 198, 40, 2846, 4560, 311, 743, 709, 856, 12443, 11, 649, 499,
1520, 30, 128012, 198, 257, 128011, 78191, 198, 2173, 3388, 11, 1524, 439, 264, 51587, 11, 5557,
649, 387, 264, 2766, 315, 264, 8815, 7170, 510, 2434, 12921, 9182, 60, 128012, 271],
'attention_mask': [1, 1,
1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1], 'assistant_masks': [0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 1]}


```

실제로, 마스킹은 다운스트림 평가에 큰 영향을 미치지 않으며 대부분의 경우 몇 포인트의 개선을 제공합니다. SmolLM3에서는 IFEval에 가장 큰 영향을 미침을 발견했는데, 아마도 모델이 프롬프트를 재진술하는 경향이 줄어들고 다양한 제약을 더 밀접하게 따르기 때문일 것입니다. 사용자 마스킹이 각 평가와 추론 모드에 어떻게 영향을 미쳤는지에 대한 비교는 아래에 표시되어 있습니다:



### 패킹할 것인가 말 것인가?

시퀀스 패킹은 학습 효율성에 큰 차이를 만드는 학습 세부 사항 중 하나입니다. SFT에서 대부분의 데이터셋은 가변 길이의 샘플을 포함하며, 이는 각 배치가 컴퓨팅을 낭비하고 수렴을 늦추는 많은 수의 패딩 토큰을 포함함을 의미합니다.

패킹은 원하는 최대 토큰 길이가 달성될 때까지 여러 시퀀스를 연결하여 이를 해결합니다. 연결을 수행하는 다양한 방법이 있으며, TRL은 "best-fit decreasing" 전략([Ding et al., 2024](#))을 채택하여 패킹할 시퀀스의 순서가 길이에 의해 결정됩니다.

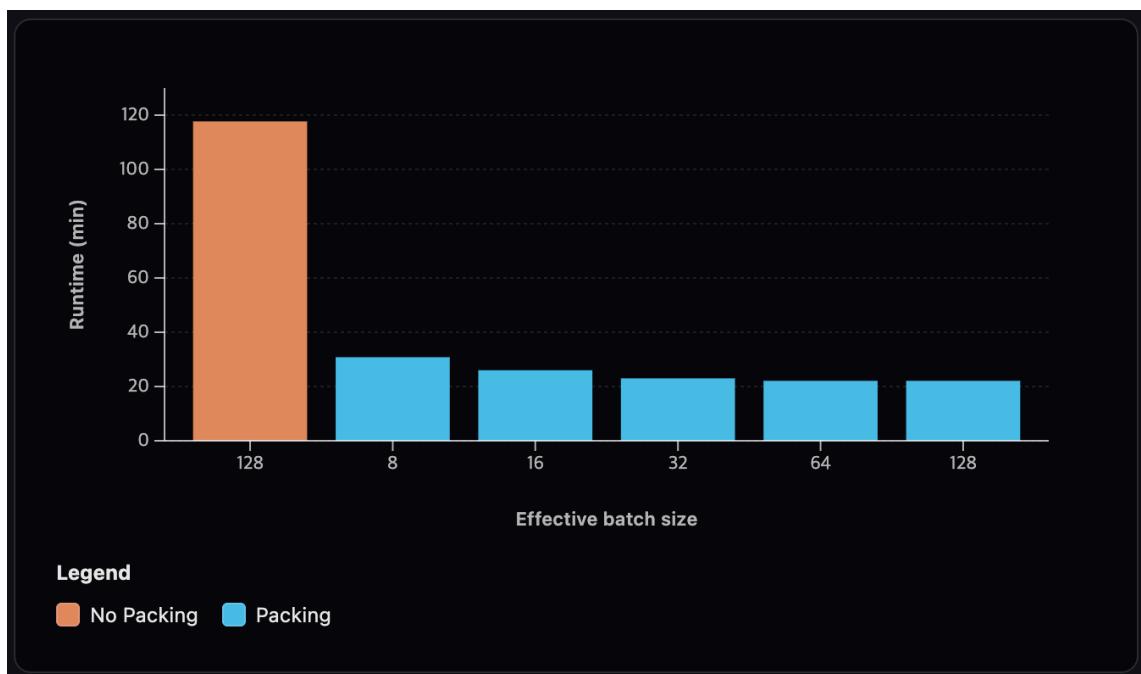
니다. 아래에서 보여주듯이, 이 전략은 배치 경계에 걸친 문서의 잘림을 최소화하면서 패딩 토큰의 양도 줄입니다:



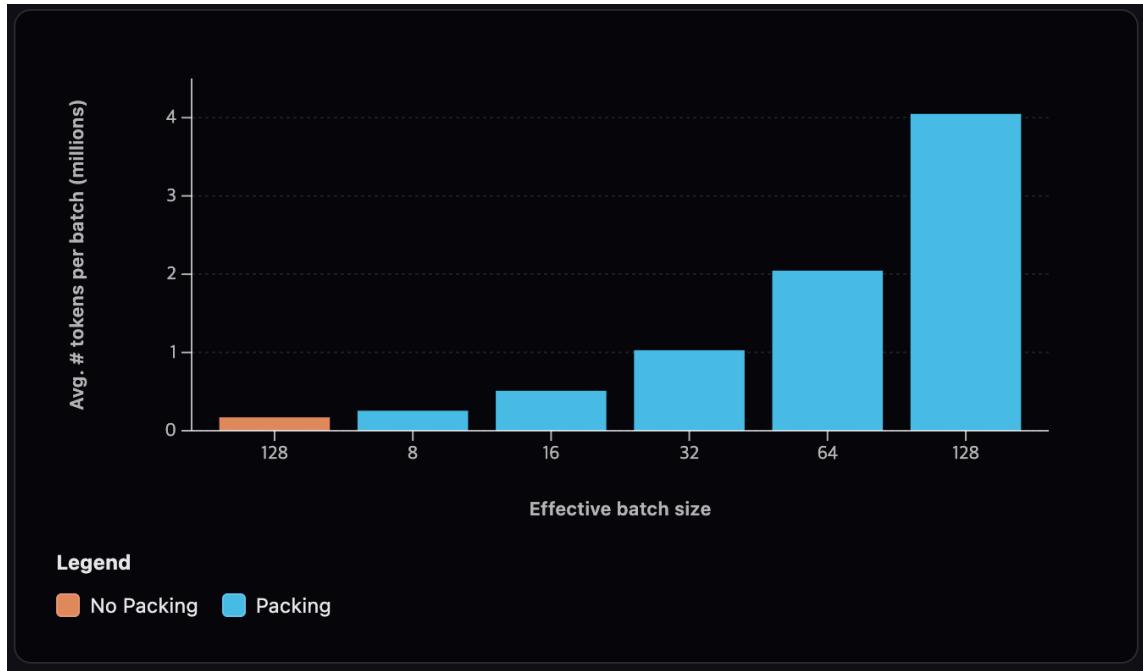
#### 후속 학습 vs 사전학습에서의 패킹

사전학습에서 이것은 실제로 질문이 아닙니다. 수조 개의 토큰에서 학습할 때, 패딩에 상당한 양의 컴퓨팅을 낭비하는 것을 피하기 위해 패킹은 필수적입니다. Megatron-LM과 Nanotron과 같은 사전학습 프레임워크는 기본적으로 패킹을 구현합니다. 후속 학습은 다릅니다. 실행이 더 짧기 때문에 트레이드오프가 변합니다.

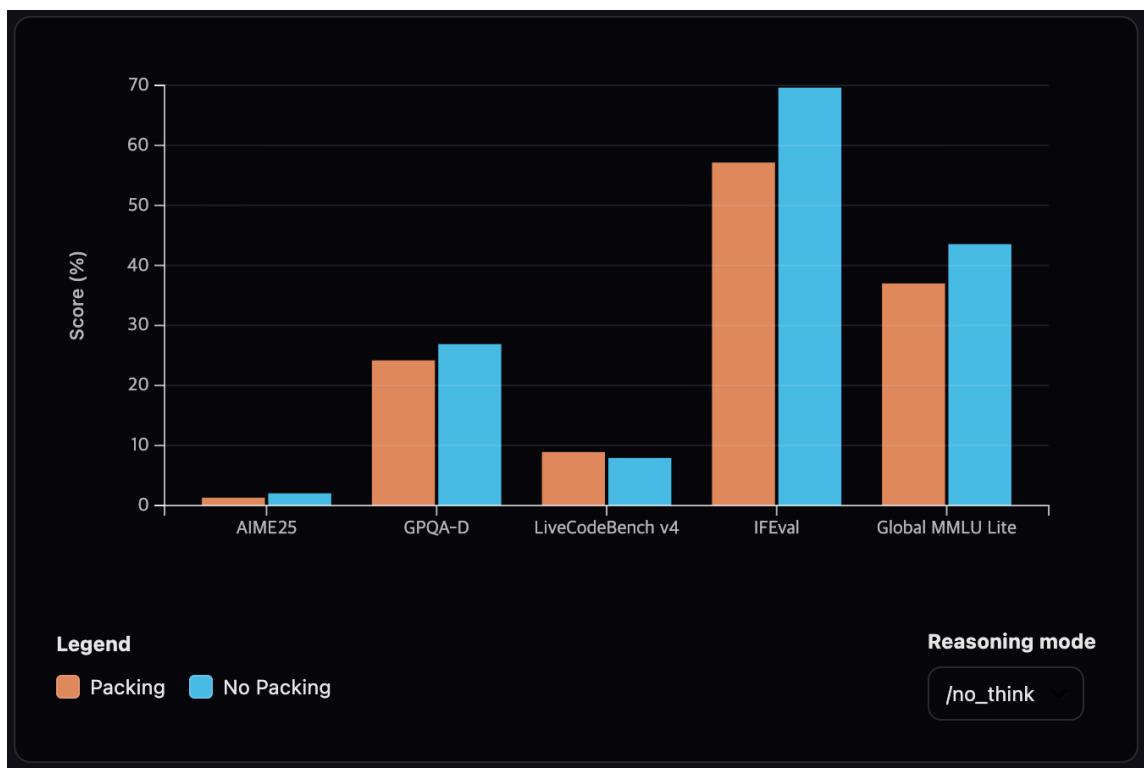
패킹이 학습에 얼마나 효율적인지 감을 잡기 위해, 아래에서 기준선 데이터셋의 한 에포크에 걸쳐 패킹과 비패킹 간의 런타임을 비교합니다:



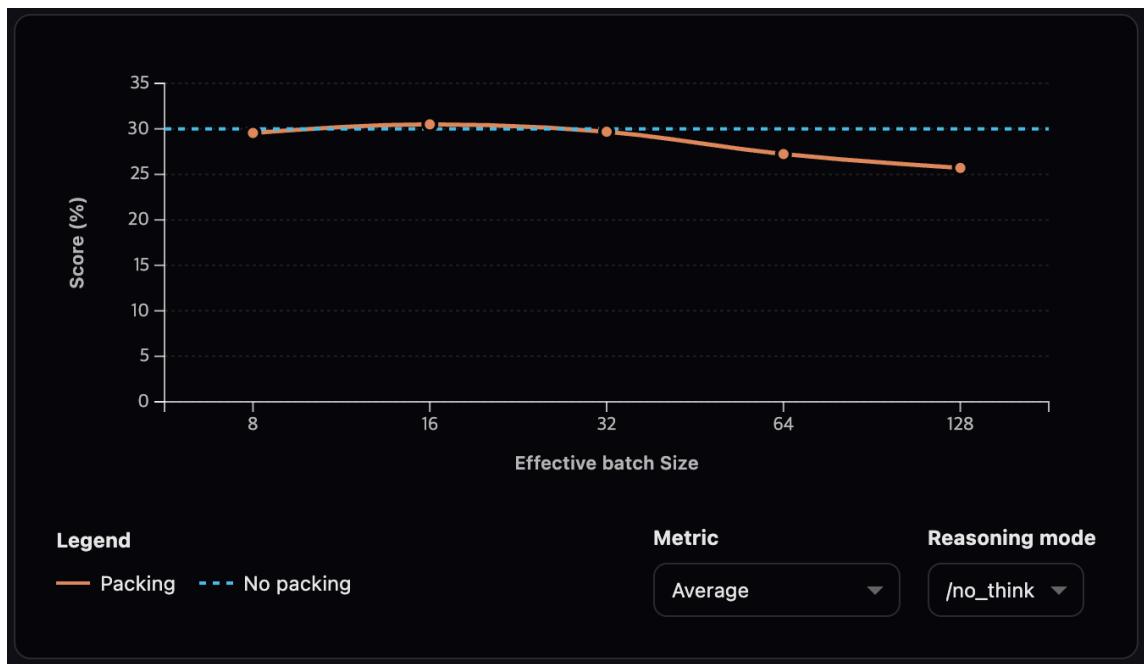
배치 크기에 따라, 패킹이 처리량을 3-5배 향상시킬 수 있습니다! 그렇다면 항상 패킹을 사용해야 할까요? 어느 정도 답은 데이터셋이 얼마나 큰지에 달려 있습니다. 패킹은 각 스텝에 더 많은 토큰을 맞춤으로써 에포크당 최적화 스텝 수를 줄이기 때문입니다. 다음 그림에서 이를 볼 수 있으며, 배치당 평균 비패킹 토큰 수를 플롯합니다:



패킹을 사용하면, 배치당 토큰 수가 배치 크기에 따라 선형적으로 스케일링되며, 패킹 없이 학습하는 것과 비교하여 최적화 스텝당 최대 33배 더 많은 토큰을 포함할 수 있습니다! 그러나 패킹은 학습 역학을 약간 변경할 수 있습니다: 전반적으로 더 많은 데이터를 처리하지만, 더 적은 그래디언트 업데이트를 수행하여 최종 성능에 영향을 미칠 수 있습니다. 특히 각 샘플이 더 중요한 작은 데이터셋에서도요. 예를 들어, 동일한 유효 배치 크기 128에서 패킹과 비패킹을 비교하면, IFEval과 같은 일부 평가가 거의 10 퍼센트 포인트의 상당한 성능 저하를 받는 것을 볼 수 있습니다:



더 일반적으로, 유효 배치 크기가 32보다 커지면, 이 특정 모델과 데이터셋에서 평균 성능 저하가 있음을 볼 수 있습니다:



실제로, 데이터셋이 거대한 대규모 SFT에서는 컴퓨팅 절약이 그래디언트 빈도의 사소한 차이를 훨씬 능가하므로 패킹이 거의 항상 유익합니다. 그러나 도메인 특화 파인튜닝이나 제한된 인간 큐레이션 데이터에 대한 지시 튜닝과 같은 더 작거나 더 다양한 데이터셋의 경우, 샘플 세분성을 보존하고 모든 예시가 최적화에 깔끔하게 기여하도록 보장하기 위해 패킹을 비활성화하는 것이 가치가 있을 수 있습니다.

궁극적으로 최고의 전략은 경험적입니다: 패킹을 활성화한 상태로 시작하고, 처리량과 다운스트림 평가를 모두 모니터링하고, 속도 향상이 동등하거나 개선된 모델 품질로 이어지는지에 따라 조정하세요.

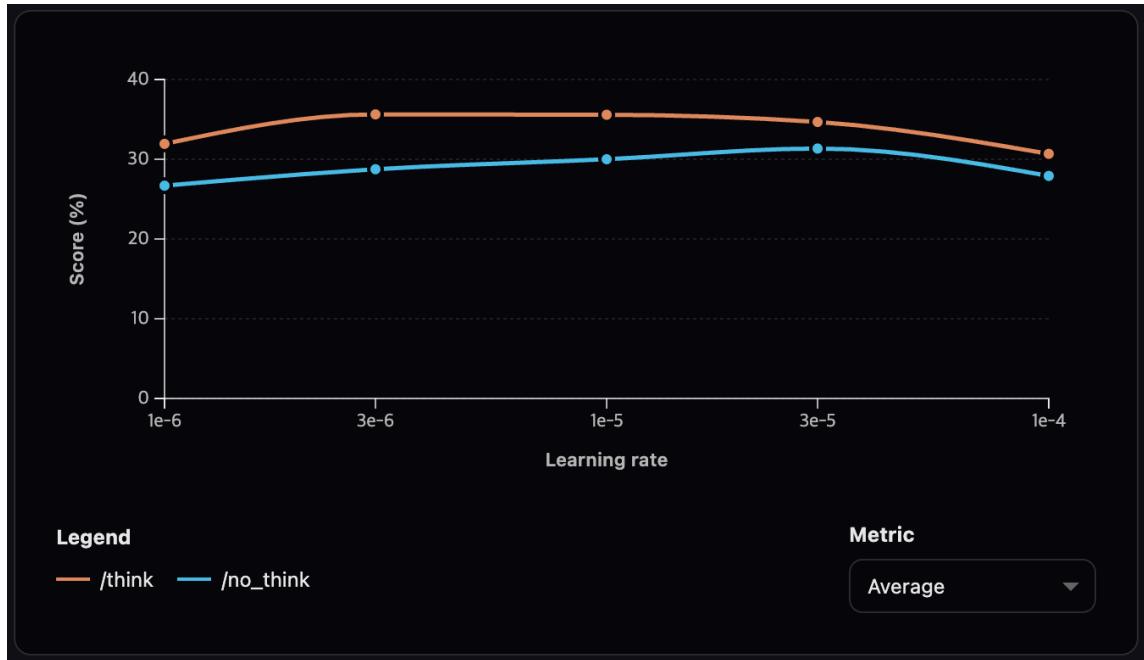
### 학습률 튜닝

이제 마지막이지만 여전히 중요한 하이퍼파라미터인 학습률에 도달합니다. 너무 높게 설정하면 학습이 발산할 수 있고; 너무 낮으면 수렴이 고통스럽게 느립니다.

SFT에서 최적의 학습률은 일반적으로 사전학습 중에 사용된 것보다 한 자릿수(또는 그 이상) 더 작습니다. 풍부한 표현을 가진 모델에서 초기화하고 있고, 공격적인 업데이트가 치명적 망각으로 이어질 수 있기 때문입니다.

**후속 학습 vs 사전학습에서 학습률 튜닝** 전체 실행에서 하이퍼파라미터 스윕이 엄청나게 비싼 사전학습과 달리, 후속 학습 실행은 충분히 짧아서 실제로 전체 학습률 스윕을 할 수 있습니다.

우리 실험에서, "최적의" 학습률은 모델 패밀리, 크기, 패킹 사용 모두에 따라 달라짐을 발견했습니다. 높은 학습률이 폭발하는 그래디언트로 이어질 수 있으므로, 패킹이 활성화되었을 때 학습률을 약간 줄이는 것이 종종 더 안전함을 발견했습니다. 아래에서 이를 볼 수 있으며, 3e-6 또는 1e-5의 작은 학습률을 사용하면 큰 값보다 더 나은 전반적인 성능을 제공합니다:

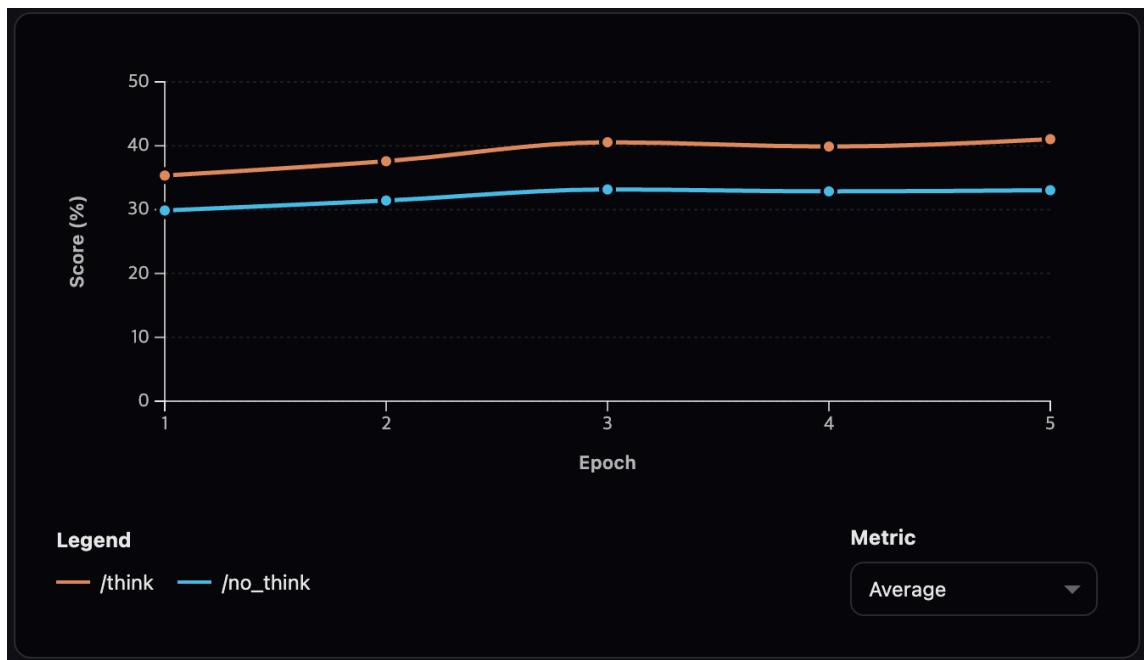


평균적으로 몇 포인트가 많아 보이지 않을 수 있지만, AIMIE25와 같은 개별 벤치마크를 보면 학습률이 1e-5보다 클 때 성능이 급격히 떨어지는 것을 볼 수 있습니다.

## 에포크 수 스케일링

절제 실험에서는 빠르게 반복하기 위해 보통 단일 에포크 동안 학습합니다. 좋은 데이터 혼합을 식별하고 학습률과 같은 핵심 파라미터를 튜닝했으면, 다음 단계는 최종 학습을 위해 에포크 수를 늘리는 것입니다.

예를 들어, 기준선 데이터 혼합을 가져와서 5 에포크 동안 학습하면, 평균적으로 몇 퍼센트 포인트의 성능을 더 짜낼 수 있음을 볼 수 있습니다:



학습률 스캔에서 보았듯이, 평균 성능은 에포크 수 스케일링이 개별 평가에 미치는 영향을 가립니다: 확장된 사고를 가진 LiveCodeBench v4의 경우, 1 에포크에 비해 성능이 거의 두 배가 됩니다!

SFT 데이터 혼합을 반복하고 모델이 합리적인 수준의 성능에 도달했으면, 다음 단계는 종종 선호도 최적화나 강화 학습과 같은 더 고급 방법을 탐색하는 것입니다. 그러나 그것들에 뛰어들기 전에, 추가 컴퓨팅이 지속적 사전학습을 통해 베이스 모델을 강화하는 데 더 잘 쓰일지 고려할 가치가 있습니다.

### 💡 후속 학습에서의 옵티마이저

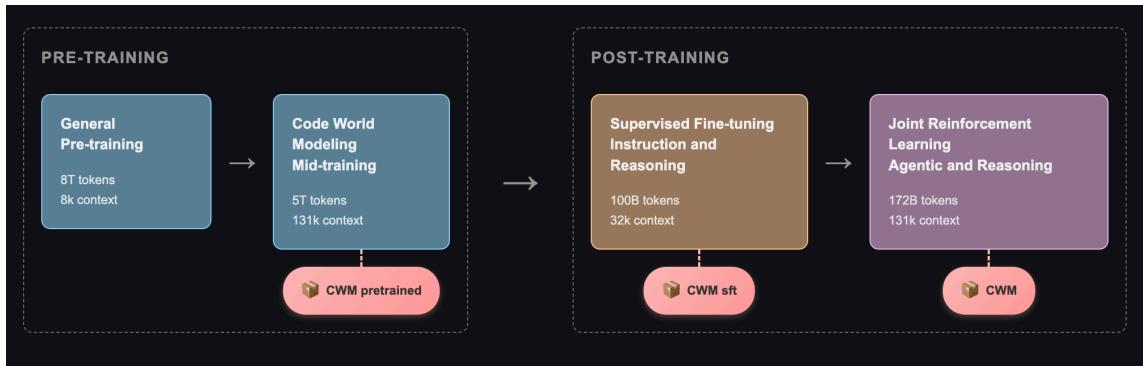
사전학습 섹션에서 언급한 또 다른 중요한 구성 요소는 옵티마이저입니다. 마찬가지로, AdamW는 후속 학습의 기본 선택으로 남아 있습니다. 열린 질문은 Muon과 같은 대체 옵티마이저로 사전학습된 모델이 동일한 옵티마이저로 후속 학습되어야 하는지입니다. Kimi 팀은 사전 및 후속 학습에 동일한 옵티마이저를 사용하면 [Moonlight](#) 모델에 대해 최고의 성능을 얻음을 발견했습니다.

## 지속적 사전학습을 통한 추론 강화

지속적 사전학습—또는 멋지게 들리고 싶다면 중간 학습—은 베이스 모델을 가져와서 SFT를 수행하기 전에 대량의 도메인 특화 토큰에서 추가로 학습시키는 것을 의미합니다. 중간 학습은 SFT의 목표 능력이 코딩이나 추론과 같은 공통 핵심 기술

을 공유할 때 유용합니다. 실제로, 이것은 모델을 추론, 특정 언어, 또는 관심 있는 다른 능력을 더 잘 지원하는 분포로 이동시킵니다. 이미 그 핵심 기술을 통합한 모델에서 SFT를 시작하면 모델이 핵심 기술을 처음부터 배우는 데 컴퓨팅을 사용하는 것이 아니라 SFT 데이터의 특정 주제에 더 잘 집중할 수 있습니다.

중간 학습 접근 방식은 ULMFit([Howard & Ruder, 2018](#))으로 거슬러 올라가며, 이것은 현재 FAIR의 Code World Model([team et al., 2025](#))과 같은 현대 LLM에서 일반적인 일반 사전학습 → 중간 학습 → 후속 학습의 3단계 파이프라인을 개척했습니다:



이 접근 방식은 Phi-4-Mini-Reasoning([Xu et al., 2025](#))의 학습에서도 사용되었지만, 변형이 있었습니다: 웹 데이터에서 지속적 사전학습을 하는 대신, 저자들은 중간 학습 코퍼스로 DeepSeek-R1에서 종류된 추론 토큰을 사용했습니다. 결과는 설득력이 있었으며, 다단계 학습을 통해 일관되고 큰 이득을 보여주었습니다:

| 모델                          | AIME24 | MATH-500 | GPQA Diamond |
|-----------------------------|--------|----------|--------------|
| Phi-4-Mini                  | 10.0   | 71.8     | 36.9         |
| + Distill Mid-training      | 30.0   | 82.9     | 42.6         |
| + Distill Fine-tuning       | 43.3   | 89.3     | 48.3         |
| + Roll-Out DPO              | 50.0   | 93.6     | 49.0         |
| + RL (Phi-4-Mini-Reasoning) | 57.5   | 94.6     | 52.0         |

이러한 결과는 유사한 접근 방식을 시도하도록 촉발했습니다. Open-R1에서 추론 데이터셋을 생성하고 평가한 이전 경험에서, 작업할 세 가지 주요 후보가 있었습니다:

- **Mixture of Thoughts**: 수학, 코드, 과학에 걸쳐 DeepSeek-R1에서 종류된 350k 추론 샘플.
- **Llama-Nemotron-Post-Training-Dataset**: Llama3와 DeepSeek-R1과 같은 다양한 모델에서 종류된 NVIDIA의 대규모 데이터셋. DeepSeek-R1 출력에 대해 데이터셋을 필터링했으며, 약 3.64M 샘플 또는 18.7B 토큰이 되었습니다.
- **OpenThoughts3-1.2M**: QwQ-32B에서 종류된 1.2M 샘플로 구성된 가장 고품질 추론 데이터셋 중 하나로, 16.5B 토큰을 포함합니다.

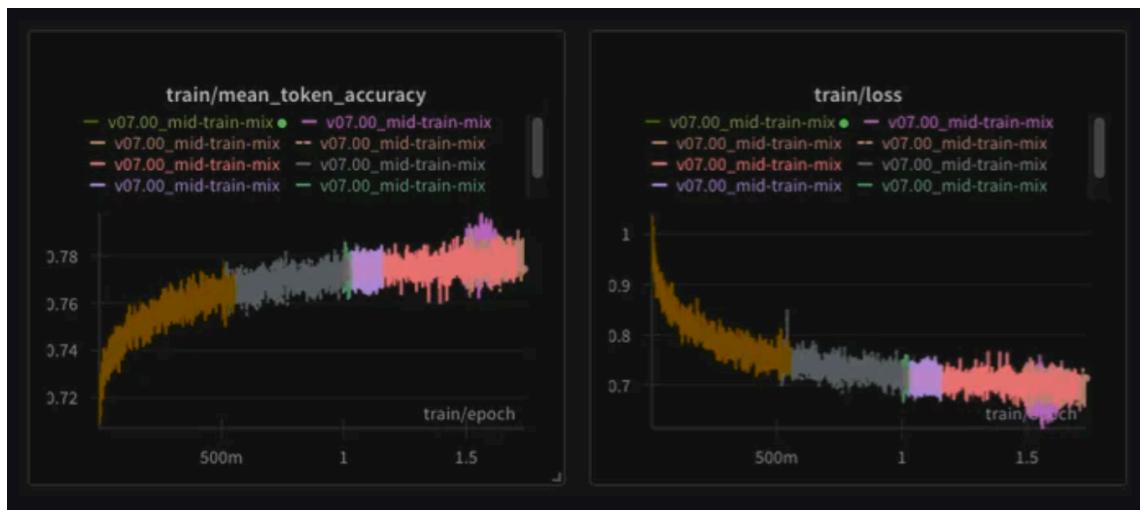
최종 SFT 훈련에 추론 데이터를 포함할 계획이었으므로, Mixture of Thoughts는 그 단계를 위해 남겨두고 나머지는 중간 학습을 위해 사용하기로 결정했습니다. SmoILM3 채팅 템플릿을 너무 일찍 "고정"하는 것을 피하기 위해 ChatML을 채팅 템플릿으로 사용했습니다. 또한 유효 배치 크기 128로 학습을 가속화하기 위해 8개 노드를 사용하여 학습률 2e-5로 5 에포크 동안 학습했습니다.

☞ 언제 중간 학습을 해야 하나요?

SFT 실행을 한 후에 중간 학습을 논의하는 이유가 궁금할 수 있습니다. 시간순으로, 중간 학습은 베이스 모델에서 SFT 전에 발생합니다. 그러나 중간 학습을 수행하기로 한 결정은 초기 SFT 실험을 실행하고 성능 격차를 식별한 후에야 명확해집니다. 실제로, 종종 반복하게 됩니다: 약점 영역을 식별하기 위해 SFT를 실행하고, 그런 다음 타겟팅된 중간 학습을 수행하고, 다시 SFT를 실행합니다. 이 섹션을 "SFT만으로는 충분하지 않을 때 무엇을 해야 하는지"로 생각하세요.

### 녹아내리는 GPU의 미스터리

이러한 실험을 실행하는 것은 우리 클러스터에서 놀라운 도전으로 밝혀졌습니다: 노후화된 GPU가 다양한 지점에서 스로틀링되어 하드웨어 장애와 각 실행의 강제 재시작으로 이어졌습니다. 어떤 느낌이었는지 맛보기로, 다음은 실행 중 하나의 로그이며, 각 색상은 재시작을 나타냅니다:



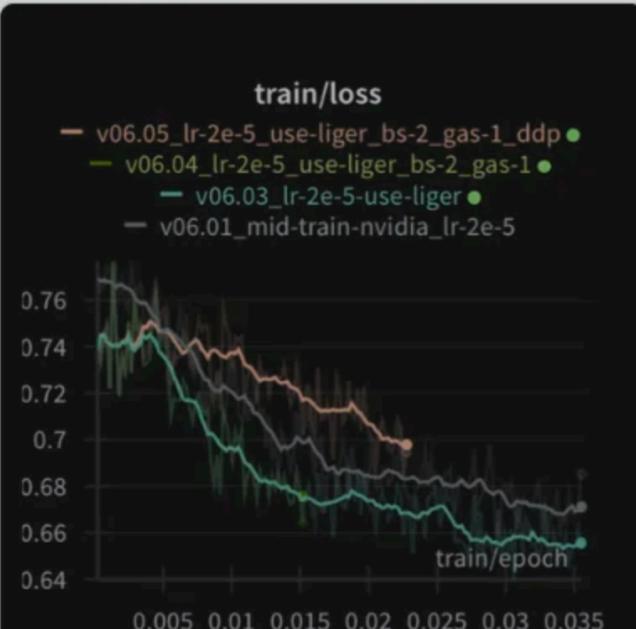
처음에는 가속기가 처리량에 고도로 최적화되어 있으므로 DeepSpeed가 범인일 수 있다고 생각했습니다. 이를 테스트하기 위해 DP로 전환했는데, 어느 정도 도움이 되었지만, 손실이 극적으로 달랐습니다!

 lewis Jun 20th at 12:13 AM

Looks like switching from Z3 to DDP might be the culprit 🤔. We did this to see if it would mitigate the node failures, but didn't ablate the loss dynamics

cc @Quentin it seems there is quite a difference between DDP and Z3 in TRL

Screenshot 2025-06-20 at 00.12.16.png ▾



1 1 1

나중에 발견했듯이, Accelerate의 DP 버그로 인해 가중치와 그래디언트가 모델의 네이티브 정밀도(이 경우 BF16)로 저장되었고, 이것이 누적과 최적화 중에 수치 불안정성과 그래디언트 정확도 손실로 이어졌습니다.

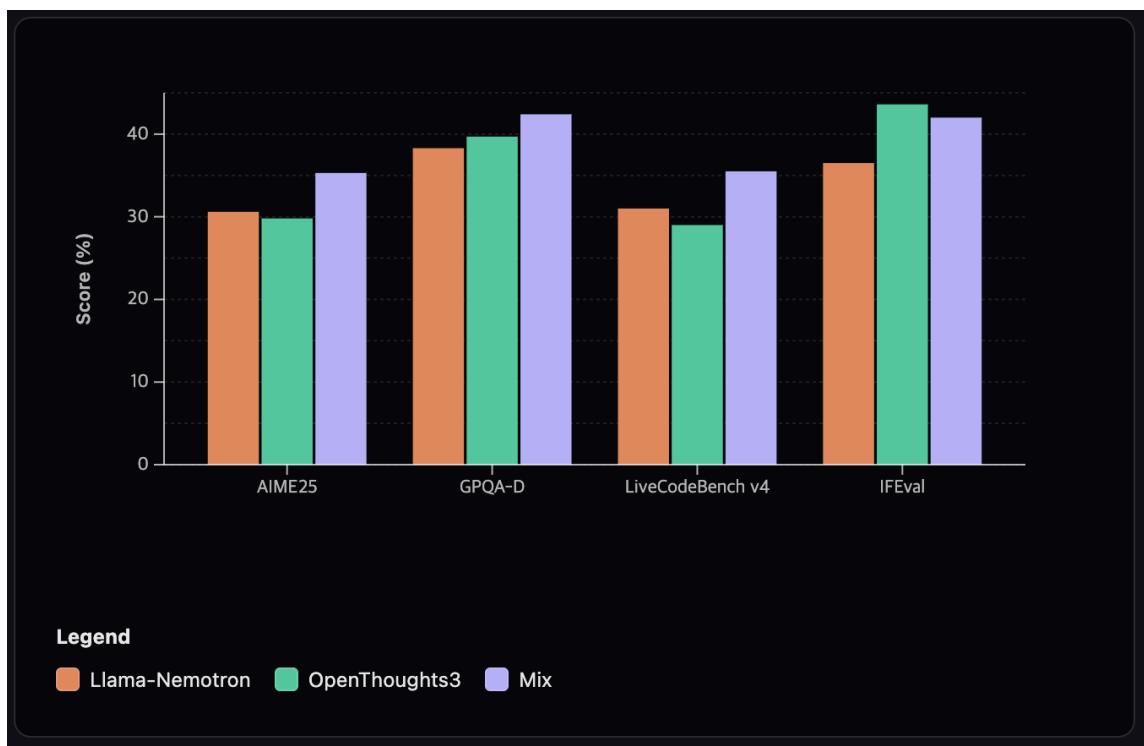
이를 방지하기 위해, 대부분의 가속기는 "마스터 가중치"와 옵티마이저 상태에 FP32를 사용하고, 순방향 및 역방향 패스에 대해서만 BF16으로 다시 캐스팅합니다.

그래서 DeepSpeed로 다시 전환하고 GPU 과열로 인해 손실되는 시간을 최소화하고 "버스에서 떨어지는" 것을 방지하기 위해 공격적인 체크포인팅을 추가했습니다. 이 전략은 성공적이었으며 더 일반적으로 권장하는 것입니다:

### 👉 규칙

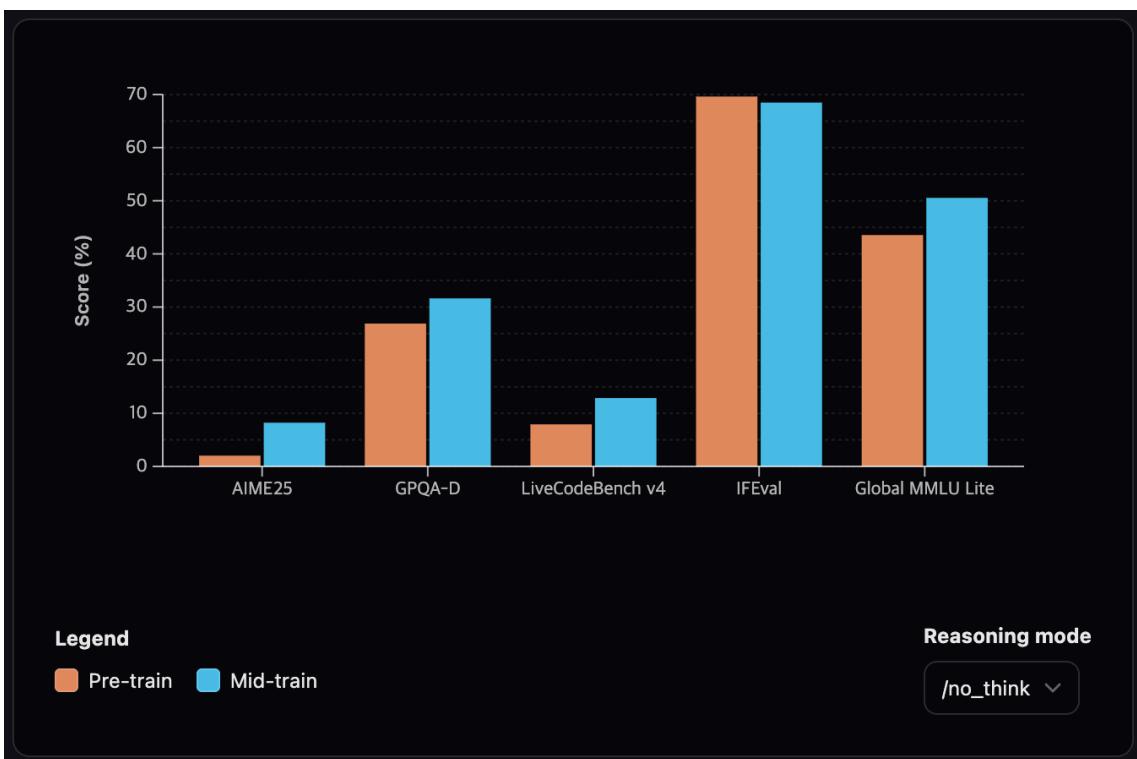
사전학습에서 강조했듯이, 학습 실행 중에 모델 체크포인트를 자주 저장하고, 이상적으로는 실수로 덮어쓰는 것을 피하기 위해 Hugging Face Hub에 푸시하세요. 또한 학습 프레임워크를 장애에 강건하고 자동 재시작이 가능하게 만드세요. 이 두 전략 모두 특히 중간 학습과 같은 장기 실행 작업에서 시간을 절약할 것입니다.

일주일 정도 실행을 돌보며 지켜본 후, 마침내 결과를 얻었습니다:



전반적으로, NVIDIA의 후속 학습 데이터셋이 OpenThoughts보다 더 나은 성능을 제공했지만, 조합이 전반적으로 가장 좋았습니다.

이제 이러한 체크포인트 중 하나를 가져와서 동일한 기준선 데이터 훈련을 적용하는 효과를 살펴보겠습니다:



사전학습된 모델 대신 중간 학습된 추론 모델을 사용하는 효과는 극적입니다: 확장된 사고로, AIME25와 LiveCodeBench v4에서 성능이 거의 3배가 되고, GPQA-D는 완전히 10포인트 상승합니다. 다소 놀랍게도, 추론 핵심은 /no\_think 추론 모드에도 부분적으로 전이되어, 추론 벤치마크에서 약 4-6 포인트 개선이 있었습니다. 이러한 결과는 추론 모델의 경우, 베이스 모델이 사전학습 중에 이미 많은 추론 데이터를 보지 않았다면 어느 정도의 중간 학습을 수행하는 것이 거의 항상 합리적이라는 명확한 증거를 제공했습니다.

#### 💡 언제 중간 학습을 하지 말아야 하나요

중간 학습은 모델이 새로운 핵심 기술을 배워야 할 때 빛납니다. 베이스 모델이 이미 그 기술을 가지고 있거나 스타일이나 대화적 잡담과 같은 얇은 능력을 이끌어내려고 할 때는 덜 유용합니다. 이러한 경우, 중간 학습을 건너뛰고 컴퓨팅을 선호도 최적화나 강화 학습과 같은 다른 방법에 할당하는 것을 권장합니다.

SFT 데이터 혼합과 모델의 광범위한 능력에 확신이 생기면, 초점은 자연스럽게 기술을 배우는 것에서 개선하는 것으로 이동 합니다. 대부분의 경우, 가장 효과적인 전진 방법은 선호도 최적화입니다.

## From SFT to preference optimisation: teaching models what better means

더 많은 데이터로 SFT를 계속 스케일링할 수 있지만, 어느 시점에서 수학 체감이나 모델이 자신의 버그가 있는 코드를 수정 할 수 없는 것과 같은 실패 모드를 관찰하게 됩니다. 왜일까요? SFT는 **모방 학습**의 한 형태이므로, 모델은 학습된 데이터의

패턴을 재현하는 것만 배우기 때문입니다. 데이터에 이미 좋은 수정이 포함되어 있지 않거나, 원하는 동작이 증류로 이끌어내기 어려운 경우, 모델은 무엇이 "더 좋은" 것인지에 대한 명확한 신호가 없습니다.

데이터셋에 트레이스의 균등한 혼합(즉, 일부는 즉시 올바른 솔루션에 도달하고 다른 일부는 모델이 먼저 실수를 한 다음 수정하는 것)이 포함되어 있어도 문제는 지속됩니다. 이 경우, 모델은 초기 오류를 만드는 것이 원하는 패턴의 일부라고 단순히 학습할 수 있습니다. 물론 우리가 실제로 원하는 것은 처음부터 올바른 솔루션을 생성할 수 있는 모델입니다.

여기서 선호도 최적화가 등장합니다. 단순히 시연을 복사하는 대신, 모델에 "응답 A가 응답 B보다 낫다"와 같은 비교 피드백을 제공합니다. 이러한 선호도는 품질에 대한 더 직접적인 학습 신호를 제공하고 모델 성능이 SFT만의 한계를 넘어 스케일링할 수 있게 합니다.

선호도 최적화의 또 다른 이점은 시작점이 이미 지시를 따를 수 있고 이전 학습 단계에서 지식을 가진 꽤 좋은 모델이므로, 일반적으로 SFT보다 훨씬 적은 데이터가 필요하다는 것입니다.

이러한 데이터셋이 어떻게 생성되는지 살펴보겠습니다.

## Creating preference datasets

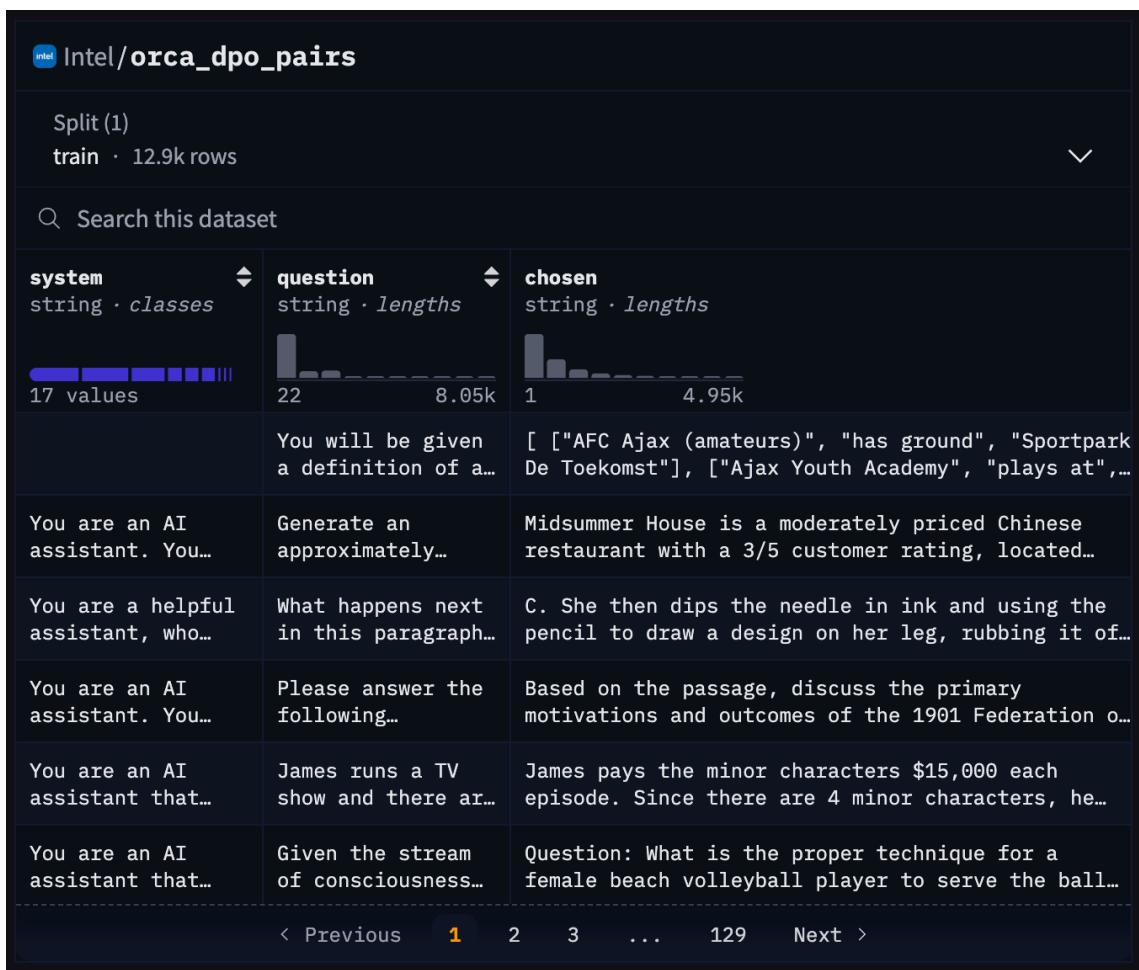
역사적으로, 선호도 데이터셋은 인간 주석자에게 모델 응답 쌍을 제공하고 어느 것이 더 나은지(가능하면 척도로) 등급을 매기도록 요청하여 생성되었습니다. 이 접근 방식은 여전히 LLM 제공자가 **인간 선호도** 레이블을 수집하는데 사용되지만, 매우 비싸고 스케일링이 잘 안 됩니다. 최근 LLM은 고품질 응답을 생성할 수 있게 되었고, 종종 비용 효율적인 방식으로 가능합니다. 이러한 발전은 LLM이 많은 애플리케이션에 대해 선호도를 생성하는 것을 실용적으로 만들었습니다. 실제로, 두 가지 일반적인 접근 방식이 있습니다:

### 강한 것 vs. 약한 것

1. 고정된 프롬프트 세트  $\mathbf{x}$ (종종 커버리지와 난이도를 위해 큐레이션됨)를 가져옵니다.
2. 더 약하거나 기준선 모델에서 하나의 응답을 생성하고, 고성능 모델에서 다른 응답을 생성합니다.
3. 더 강한 모델의 출력을 선택된 응답  $\mathbf{y}_c$ 로 레이블하고 더 약한 것을 거부됨  $\mathbf{y}_r$ 로 레이블합니다.

이것은 "더 강한 것 vs. 더 약한 것" 비교의 데이터셋 ( $\mathbf{x}, \mathbf{y}_c, \mathbf{y}_r$ )을 생성하며, 더 강한 모델의 출력이 안정적으로 더 낫다고 가정하기 때문에 구성하기 간단합니다.

아래는 Intel의 인기 있는 예시로, gpt-3.5와 gpt-4의 응답이 있는 SFT 데이터셋을 가져와서 gpt-4 응답을 선택으로, gpt-3.5 응답을 거부로 선택하여 선호도 데이터셋으로 변환했습니다:



## 그레이딩을 사용한 온풀리시

- 동일한 프롬프트에 대해 여러 후보 응답을 생성하기 위해 학습할 **동일한 모델**을 사용합니다. 이것은 모델이 자연스럽게 생성할 출력의 분포를 반영하기 때문에 "온풀리시"인 데이터를 생성합니다.
- 더 강한 모델을 참조로 의존하는 대신, **외부 그레이더**를 도입합니다: 하나 이상의 품질 측(예: 유용성 또는 사실적 정확성)을 따라 응답에 점수를 매기는 검증기 또는 보상 모델입니다.
- 그레이더는 그런 다음 후보 응답 간에 선호도 레이블을 할당하여, 더 세밀하고 유연한 선호도 데이터셋을 생성합니다.

이 방법은 모델이 개선됨에 따라 선호도 데이터의 지속적인 부트스트래핑을 허용하지만, 품질은 평가자의 신뢰성과 보정에 크게 의존합니다.

이러한 데이터셋의 좋은 예시는 SnorkelAI에서 나왔는데, [UltraFeedback](#)이라는 인기 있는 선호도 데이터셋에서 프롬프트를 가져와서 3개 세트로 분할한 다음, 위의 레시피를 반복적으로 적용하여 모델을 개선했습니다:

| snorkelai/Snorkel-Mistral-PairRM-DPO-Dataset                                                                              |                                                                                                                                                                                                                                                                                                                                               |
|---------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Split (6)<br>train_iteration_1 · 19.8k rows                                                                               |                                                                                                                                                                                                                                                                                                                                               |
| Search this dataset                                                                                                       |                                                                                                                                                                                                                                                                                                                                               |
| <b>prompt_id</b><br>string · lengths<br> | <b>prompt</b><br>string · lengths<br>                                                                                                                                                                                                                       |
| 64      64                                                                                                                | 12      14.4k                                                                                                                                                                                                                                                                                                                                 |
| 5cf991718c2849e6d0312f999314dfb0e6dfc1a10234f461ea3<br>21b9c038262be                                                      | Please provide the content structure of the following text using [Latex] data form.                                                                                                                                                                                                                                                           |
| ec39d34c20179b2d585bf5c1f42a78152187e3f08f06df8a58b<br>51d78e07c0d06                                                      | Summarize the movie Beer from 1985                                                                                                                                                                                                                                                                                                            |
| 28fbddad640bbfbe4806f33c0ef01c16ea12b0b6fd5392fb327<br>0006e44ec2219                                                      | api authentication in nextjs 13 Post:<br><a href="https://simple-books-api.glitch.me/api/auth">https://simple-books-api.glitch.me/api/auth</a>                                                                                                                                                                                                |
| a316bb25549cc62f4f690f9948b64e863044275e9aedb20d561<br>2abf6e08421b5                                                      | can you act as a C# expert, with lots experience in C# Domain Driven Design                                                                                                                                                                                                                                                                   |
| 40d389fbf5b88c6afec4ba0be12191244f2dde26a973f3b442a<br>86a92807ffb10                                                      | What is the proper way to create a SQL table using Go to store information about books?                                                                                                                                                                                                                                                       |
| 5d0f47c3f0f4d503814f1e359824284c73d22691df8bf72aec1<br>1698139c12eaa                                                      | Can you summarize the 2-hour event in sentences? Generate according to: The event was held on Saturday, April 14, 2024, at the New York City Marathon. It featured a variety of speakers and panels, including discussions on mental health, nutrition, and training techniques. The event was well-attended by runners and spectators alike. |
| <span>&lt; Previous</span> <span>1</span> <span>2</span> <span>3</span> ... <span>198</span> <span>Next &gt;</span>       |                                                                                                                                                                                                                                                                                                                                               |

SmolLM3 개발 당시, 추론 트레이스가 있는 선호도 데이터가 존재하지 않았으므로, "강한 것 vs 약한 것" 접근 방식을 사용하여 자체적으로 일부를 생성하기로 결정했습니다. Ai2의 Tulu 3 선호도 혼합에서 프롬프트를 사용하여 /think 모드에서 Qwen3-0.6B와 Qwen3-32B의 응답을 생성했습니다. 결과는 [250k+ LLM 생성 선호도의 대규모 데이터셋](#)으로, 선호도 최적화 알고리즘을 사용하여 여러 축에 걸쳐 SFT 체크포인트를 동시에 개선할 준비가 되었습니다.

### 어떤 알고리즘을 선택해야 하나요?

Direct Preference Optimization(DPO) ([Rafailov et al., 2024](#))은 오픈소스에서 널리 채택된 최초의 선호도 최적화 알고리즘이었습니다.

DPO 논문이 2023년 중반에 나왔을 때, RL 방법과 일치할 수 있는지에 대해 온라인에서 열띤 토론이 있었고, 산업 환경에서의 효과를 보여주는 레시피가 없었습니다. 이를 해결하기 위해, 몇 달 후 Zephyr 7B를 릴리스했으며, 전체 모델을 합성 데이터에서 학습시키고 DPO로부터 상당한 성능 향상을 보여주었습니다.

그 매력은 구현이 간단하고, 실제로 안정적이며, 적당한 양의 선호도 데이터로도 효과적이라는 데서 왔습니다. 결과적으로, DPO는 RL과 같은 더 복잡한 기술에 도달하기 전에 SFT 모델을 개선하는 기본 방법이 되었습니다.

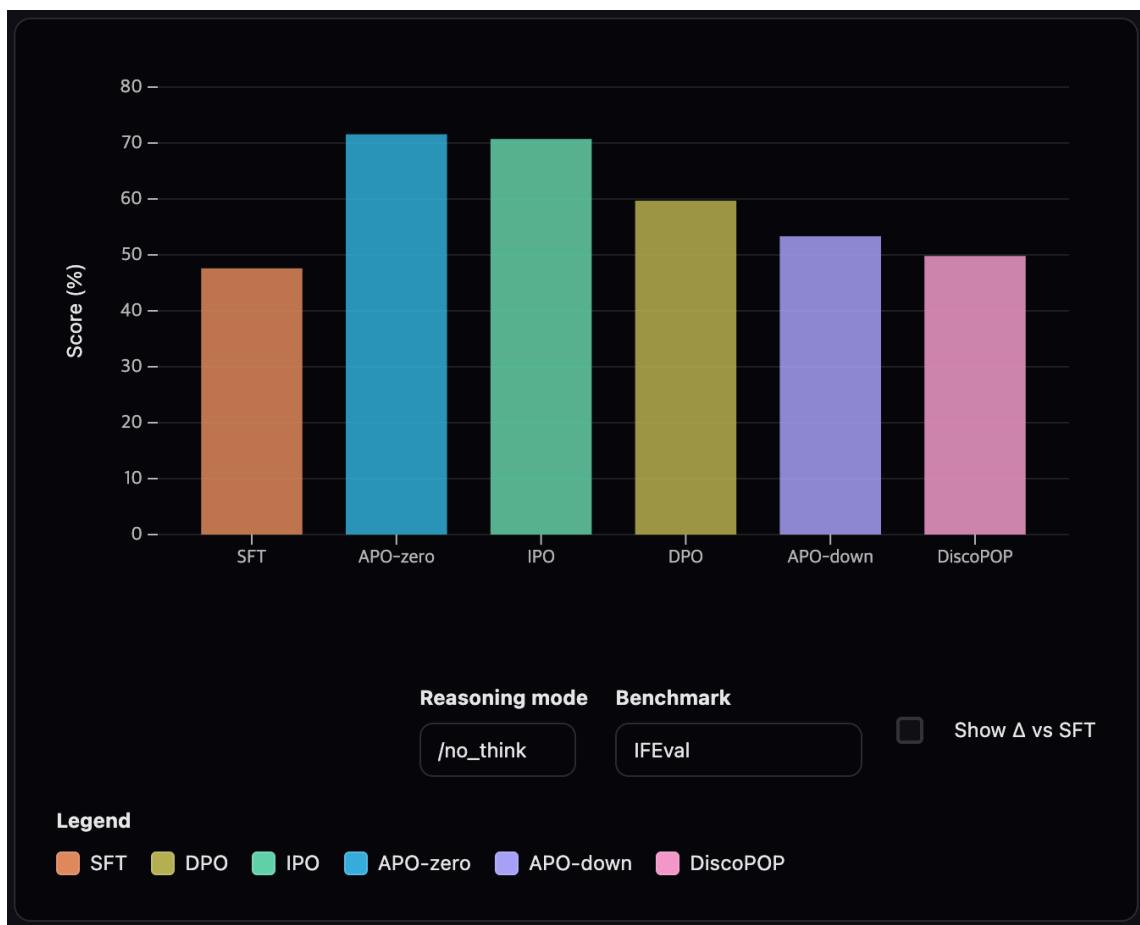
그러나 연구자들은 DPO를 개선하는 많은 방법이 있음을 빠르게 발견했고, 오늘날에는 탐색할 다양한 대안이 있습니다. 아래에 가장 효과적이라고 발견한 몇 가지를 나열합니다:

- **Kahneman-Tversky Optimisation(KTO)** [[Ethayarajh et al. \(2024\)](#)]: 선호도 쌍에 의존하는 대신, KTO는 인간 의사 결정의 아이디어를 사용하여 개별 응답이 "바람직한지 아닌지"를 모델링합니다. 쌍으로 된 선호도 데이터에 접근할 수 없는 경우(예: 최종 사용자로부터 수집된 👍 또는 🚫와 같은 원시 응답) 좋은 선택입니다.
- **Odds Ratio Preference Optimisation(ORPO)** [[Hong et al. \(2024\)](#)]: 교차 엔트로피 손실에 오즈비를 추가하여 선호도 최적화를 SFT에 직접 통합합니다. 결과적으로 참조 모델이나 SFT 단계가 필요 없어서, 이 방법을 더 계산적으로 효율적으로 만듭니다.
- **Anchored Preference Optimisation(APO)** [[D'Oosterlinck et al. \(2024\)](#)]: 단순히 차이를 최적화하는 것이 아니라 선택된 출력 vs. 거부된 출력에 대한 모델의 가능도가 얼마나 이동해야 하는지 명시적으로 정규화하는 더 제어 가능한 목적 함수입니다. 두 가지 변형(APO-zero와 APO-down)이 있으며, 선택은 모델과 선호도 데이터 간의 관계에 따라 달라집니다. 즉, 선택된 출력이 모델보다 더 나은지 더 나쁜지에 따라 달라집니다.

다행히도, 이러한 선택 중 많은 것이 TRL의 `DPOTrainer`에서 한 줄 변경일 뿐이므로, 초기 기준선을 위해 다음을 수행했습니다:

- Ai2의 Tülu3 Preference Personas IF 데이터셋의 프롬프트와 완성을 사용하여 `/no_think` 추론 모드로 IFEval에서 지시 따르기에 대한 개선을 측정합니다.
- 위의 프롬프트를 재사용하지만, 이제 Qwen3-32B와 Qwen3-0.6B로 "강한 것 vs. 약한 것" 선호도 쌍을 생성합니다. 이것은 `/think` 추론 모드에 대한 선호도 데이터를 제공했습니다.
- 1 에포크 동안 학습하고 IFEval에서의 도메인 내 개선과 함께 지시 따르기와 직접 상관관계가 있는 AIM25와 같은 다른 평가에 대한 도메인 외 영향을 측정합니다.

아래 그림에서 보여주듯이, 두 추론 모드에 대한 도메인 내 개선이 상당했습니다: IFEval에서 APO-zero가 SFT 체크포인트보다 15-20 퍼센트 포인트 개선되었습니다!



APO-zero가 전반적으로 가장 좋은 도메인 외 성능도 가지고 있었으므로, 나머지 절제 실험에서 이를 사용하기로 결정했습니다.

### 💡 선호도 최적화는 추론에 효과가 있습니다

위의 결과가 보여주듯이, 선호도 최적화는 모델을 더 유용하거나 정렬되게 만드는 것뿐만 아니라, 더 잘 추론하도록 가르칩니다. 추론 모델을 빠르게 개선할 방법이 필요하다면, 강한 것 vs 약한 것 선호도를 생성하고 다양한 손실 함수를 절제 실험해보세요: 바닐라 DPO보다 상당한 이득을 발견할 수 있습니다!

### 선호도 최적화에서 가장 중요한 하이퍼파라미터는 무엇인가요?

선호도 최적화에서, 일반적으로 학습 역학에 영향을 미치는 하이퍼파라미터는 세 가지뿐입니다:

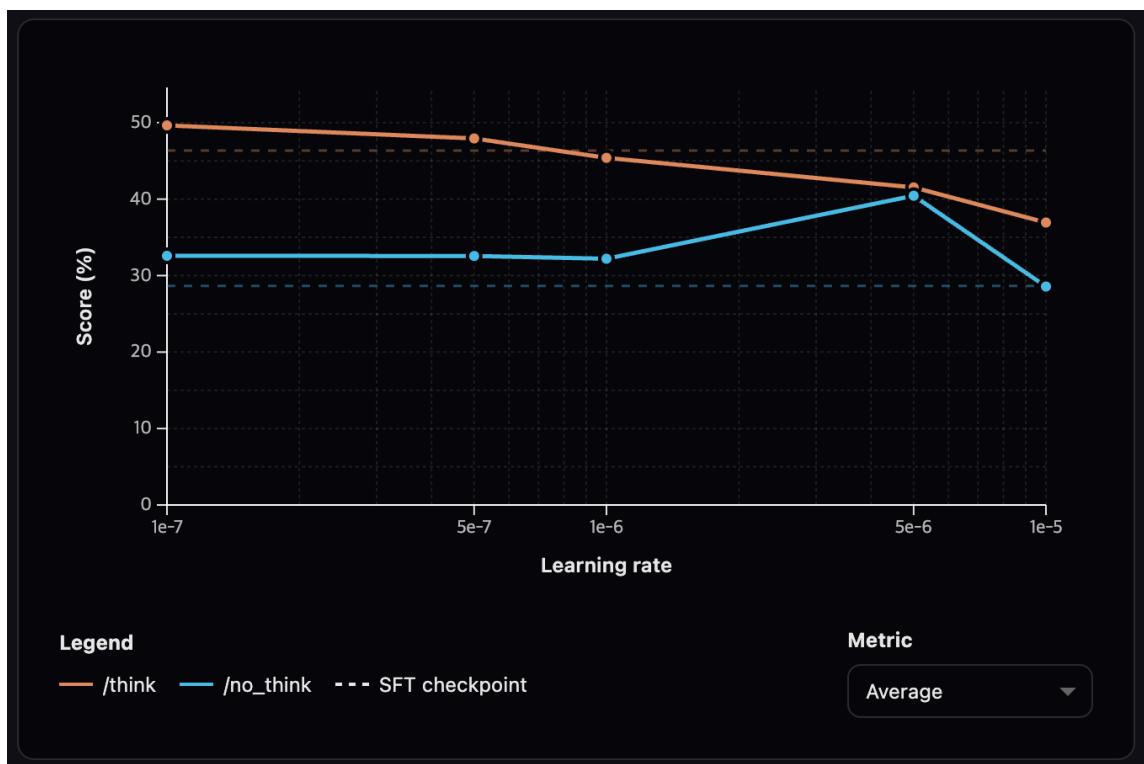
- **학습률**, 일반적으로 SFT에 사용된 것보다 10-100배 더 작은 팩터.
- **β 파라미터**, 일반적으로 선호도 쌍 간의 마진 크기를 제어합니다.
- **배치 크기**.

smoltalk2 전체에 걸쳐 학습한 [SFT checkpoint](#)에서 시작하여 SmolLM3에서 이것들이 어떻게 전개되었는지 살펴보겠습니다.

## 최고의 성능을 위해 작은 학습률을 사용하세요

실행한 첫 번째 절제 실험은 학습률이 모델 성능에 미치는 영향을 확인하는 것이었습니다. SFT 학습률( $2e-5$ )보다 ~200배 더 작은 것( $1e-7$ )과 ~2배 더 작은 것( $1e-5$ ) 사이의 학습률의 영향을 결정하기 위해 실험을 실행했습니다. Zephyr 7B와 같은 이전 프로젝트에서 선호도 최적화 방법에 가장 좋은 학습률은 SFT에 사용된 것보다 약 10배 더 작다는 것을 배웠고, SmoLLM3에 대해 실행한 절제 실험이 이 경험 법칙을 확인했습니다.

아래 그림에서 보여주듯이, ~10배 더 작은 학습률은 두 추론 모드에서 SFT 모델의 성능을 개선하지만, 그 10배 한계를 넘어서는 모든 학습률은 확장된 사고 모드에서 더 나쁜 성능을 초래합니다:



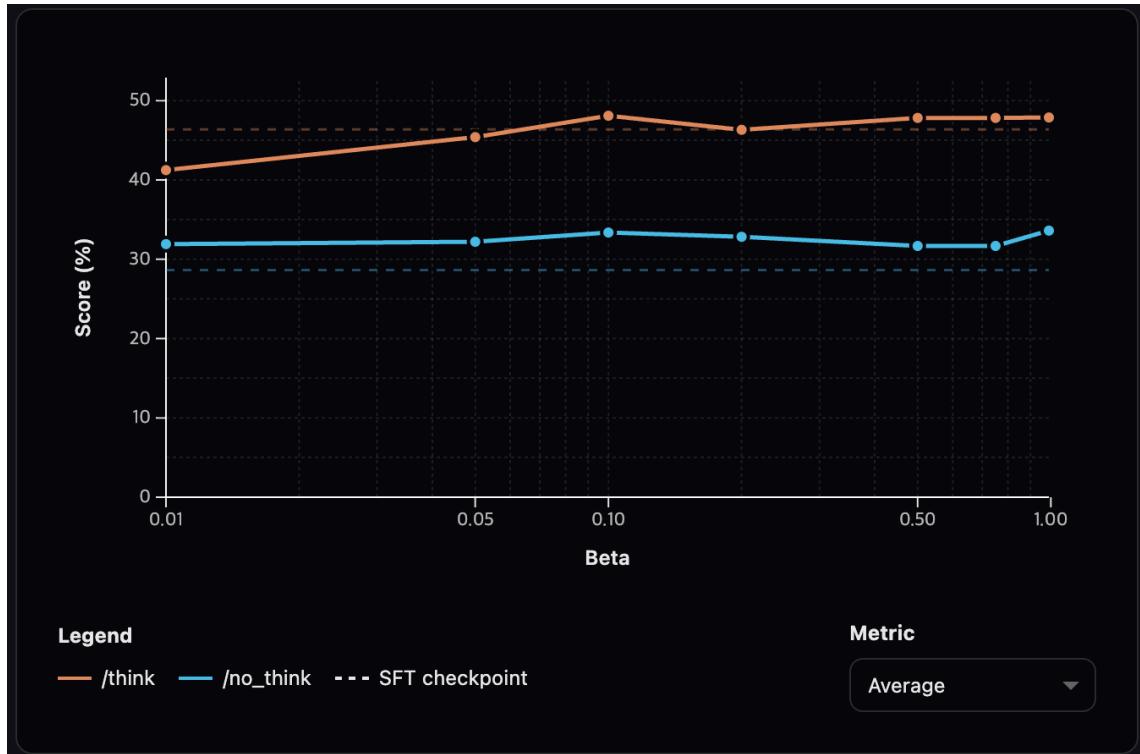
'/no\_think' 추론 모드의 추세는 더 안정적이며, 최적의 학습률은  $5e-6$ 입니다. 이것은 주로 단일 벤치마크 (LiveCodeBench v4)에 의해 주도되므로, SmoLLM3 실행에서  $1e-6$ 을 선택했습니다.

학습 실행에 대한 권장 사항은 SFT 학습률보다 5배에서 20배 더 작은 범위에서 학습률 스캔을 실행하는 것입니다. 그 범위 내에서 최적의 성능을 찾을 가능성이 매우 높습니다!

## $\beta$ 를 튜닝하세요

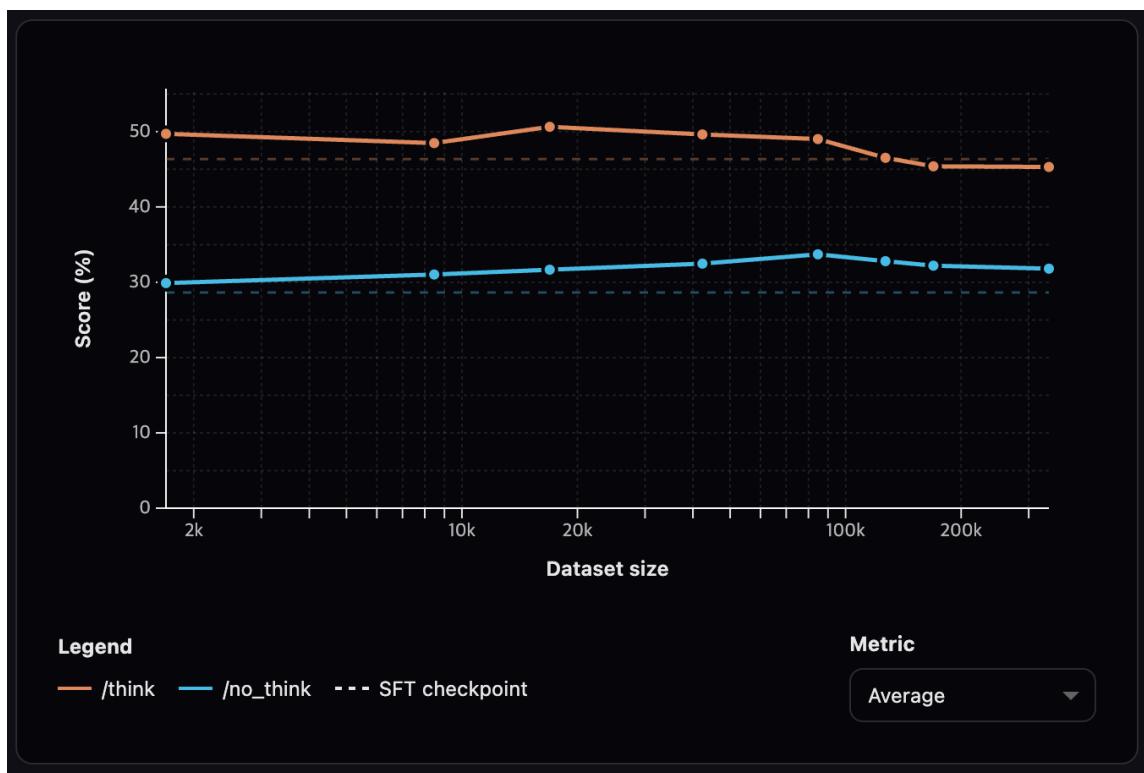
$\beta$  파라미터에 대해 실행한 실험은 참조 모델에 대한 다양한 정도의 정렬을 장려하는 값을 탐색하기 위해 0.01에서 0.99까지 범위였습니다. 상기시켜 드리면, 더 낮은 베타 값은 참조 모델에 가깝게 유지하도록 장려하고 더 높은 값은 모델이 선호도 데이터와 더 밀접하게 일치하도록 허용합니다.  $\beta=0.1$ 에서의 모델 성능은 두 추론 모드 모두에서 가장 높으며 SFT 체크포인트의 메트릭과 비교하여 개선됩니다. 낮은 베타 값을 사용하면 모델 성능이 저하되고 SFT 체크포인트보다 더 나쁜 모델이 됩니다. 반면 확장된 사고 없이는 여러  $\beta$  값에 걸쳐 성능이 안정적으로 유지됩니다.

이러한 결과는 0.1보다 큰 값이 선호도 최적화에 더 바람직하며, 모델을 선호도 데이터에 정렬하는 것이 참조 모델에 가깝게 유지하는 것보다 더 유익함을 시사합니다. 그러나 0.01에서 0.5 범위의  $\beta$  값을 탐색할 것을 제안합니다. 더 높은 값은 플롯에 표시된 평가에서 포착하지 못할 수 있는 SFT 체크포인트의 능력을 지울 수 있습니다.



### 선호도 데이터 스케일링

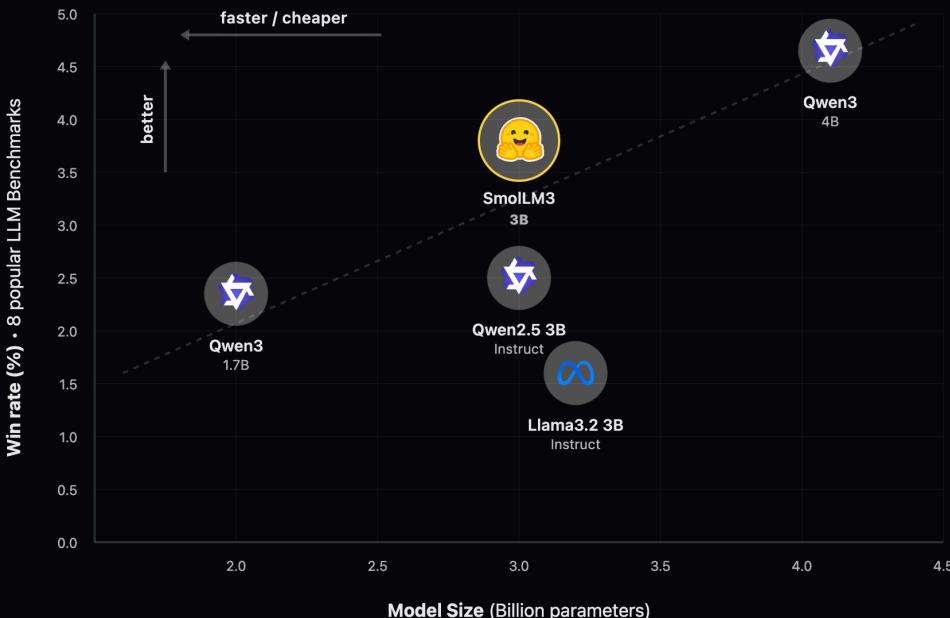
데이터셋 크기가 결과에 어떻게 영향을 미치는지 결정하기 위한 실험도 실행했으며, 2k에서 340k 선호도 쌍까지의 값을 테스트했습니다. 이 범위에 걸쳐 성능이 안정적으로 유지되었습니다. 확장된 사고 모드에서의 성능 저하는 100k 선호도 쌍을 넘어서는 데이터셋에서 발생하지만, 다른 학습률 값에서 보았던 것만큼 두드러지지는 않습니다. SmoILM3 학습 실행에 사용한 데이터셋은 169k 선호도 쌍이었지만, 결과는 더 작은 데이터셋도 SFT 체크포인트보다 개선을 보여줌을 나타냅니다. 향후 프로젝트에서는 반복 단계에서 더 작은 데이터셋으로 실험할 수 있음을 알게 되었습니다. 여러 아이디어를 시도하고 가장 유망한 구성을 빠르게 식별하는 것이 중요하기 때문입니다.



### 모든 것을 하나로 모으기

이 모든 스레드를 하나로 모아 최종 SmoLLM3-3B 모델을 생성했습니다: 그 크기에서 최고 수준이며 Qwen의 자체 하이브리드 추론 모델과 함께 파레토 프론트에 위치합니다.

#### Instruct models without reasoning



몇 주간의 작업치고는 나쁘지 않습니다!

#### 참여 규칙

향후 프로젝트에 유용할 수 있는 선호도 최적화에 대한 발견을 요약하면:

- 자체 선호도 데이터를 만드는 것을 두려워하지 마세요! 추론이 "측정하기에는 너무 저렴해지면서", 다양한 추론 제공자([inference providers](#))로부터 LLM 선호도를 생성하는 것이 오늘날 간단하고 비용 효율적입니다.
- DPO를 초기 기준선으로 선택하고 거기서부터 반복하세요. 선호도 데이터의 유형에 따라 ORPO, KTO, APO와 같은 다른 알고리즘이 DPO보다 상당한 이득을 제공할 수 있음을 발견했습니다.
- SFT에 사용된 것보다 약 10배 더 작은 학습률을 사용하세요.
- $\beta$ 를 스캔하세요, 보통 0.01에서 0.5 범위에서.
- 대부분의 선호도 알고리즘이 한 에포크 후에 과적합하므로, 데이터를 분할하고 최고의 성능을 위해 반복적으로 학습하세요.

선호도 최적화는 종종 단순성과 성능 사이의 스위트 스팟이지만, 여전히 핵심 한계를 물려받습니다: 수집할 수 있는 오프라인 선호도 데이터만큼만 좋습니다. 어느 시점에서 정적 데이터셋은 신호가 고갈되고 모델이 프롬프트 및 환경과 상호작용할 때 온라인으로 새로운 학습 피드백을 생성할 수 있는 방법이 필요합니다. 그것이 선호도 최적화가 온풀리시 및 RL 기반 방법의 더 넓은 패밀리를 만나는 곳입니다.

## 온폴리시로 가기와 지도 레이블을 넘어서

모델이 수학 문제를 일관되게 풀거나, 실행 가능한 코드를 생성하거나, 여러 단계에 걸쳐 계획하기를 원한다면, 종종 단순히 "A가 B보다 낫다"가 아닌 보상 신호가 필요합니다.

여기서 RL이 합리적이기 시작합니다. 선호도로 모델을 감독하는 대신, 환경(수학 검증기, 코드 실행기, 또는 실제 사용자 피드백일 수 있음)과 상호작용하게 하고 결과에서 직접 학습합니다. RL은 다음과 같은 경우에 빛납니다:

- 정확성을 자동으로 확인할 수 있을 때, 예: 유닛 테스트, 수학적 증명, API 호출, 또는 고품질 검증기나 보상 모델에 접근할 수 있을 때.
- 태스크가 다단계 추론이나 계획을 필요로 할 때, 로컬 선호도가 장기적 성공을 포착하지 못할 수 있는 경우.
- 선호도 레이블을 넘어서는 목표를 최적화하고 싶을 때, 코드의 유닛 테스트 통과나 일부 목표 최대화와 같은.

LLM과 관련하여 RL의 두 가지 주요 종류가 있습니다:

- **인간 피드백에서의 강화 학습(RLHF)**: 이것은 OpenAI의 InstructGPT 논문([Ouyang et al., 2022](#))에 의해 대중화된 접근 방식이며 gpt-3.5와 많은 현대 LLM의 기반입니다. 여기서 인간 주석자가 모델 출력을 비교하고(예: "A가 B보다 낫다") 보상 모델이 그러한 선호도를 예측하도록 학습됩니다. 그런 다음 정책이 학습된 보상을 최대화하기 위해 RL로 파인튜닝됩니다.

보상 모델이 인간 선호도를 근사할 뿐이므로, 때때로 보상 해킹을 장려할 수 있습니다. 정책이 가짜 높은 보상을 받는 "the the the the"와 같은 분포 밖 시퀀스를 내보내고 RL 루프를 통해 모델에 구워지는 경우입니다.

- **검증 가능한 보상을 가진 강화 학습(RLVR)**: 이것은 DeepSeek-R1에 의해 대중화된 접근 방식이며 모델의 출력이 명확하게 정의된 정확성 기준을 충족하는지 확인하는 검증기의 사용을 포함합니다(예: 코드가 컴파일되고 모든 테스트를 통과하는가, 또는 수학적 답이 맞는가?). 그런 다음 정책이 더 검증 가능하게 올바른 출력을 생성하기 위해 RL로 파인튜닝됩니다.

RLHF와 RLVR 모두 모델이 무엇을 위해 최적화되는지 정의하지만, 그 최적화가 어떻게 수행되어야 하는지 알려주지 않습니다. 실제로, RL 기반 학습의 효율성과 안정성은 학습 알고리즘이 온폴리시인지 오프폴리시인지에 크게 의존합니다.

GRPO와 같은 방법은 일반적으로 온폴리시 최적화 알고리즘의 범주에 속하며, 완성을 생성하는 모델(정책)이 최적화되는 것과 동일합니다. GRPO가 온폴리시 알고리즘이라는 것이 대체로 사실이지만, 몇 가지 주의 사항이 있습니다. 첫째, 생성 단계를 최적화하기 위해 여러 배치의 생성이 샘플링된 다음 모델에  $k$  업데이트가 이루어지며, 첫 번째 배치는 온폴리시이고 다음 몇 배치는 약간 오프폴리시입니다.

생성에 사용된 모델과 최적화되는 현재 모델 간의 정책 차이를 설명하기 위해, 중요도 샘플링과 클리핑이 토큰 확률을 재가중하고 업데이트 크기를 제한하는 데 사용됩니다.

여기서 오프폴리시 RL을 언급하지만, Q-learning과 같은 여러 진정한 오프폴리시 RL 알고리즘이 있으며, 궤적을 생성하는 데 사용되는 정책이 최적화되는 정책과 완전히 다를 수 있습니다. GRPO가 LLM에 적용될 때, 생성에 사용되는 정책이 최적화에 사용되는 정책보다 뒤처질 수 있지만, 일반적으로 둘 사이에 16스텝 미만의 차이가 있습니다.

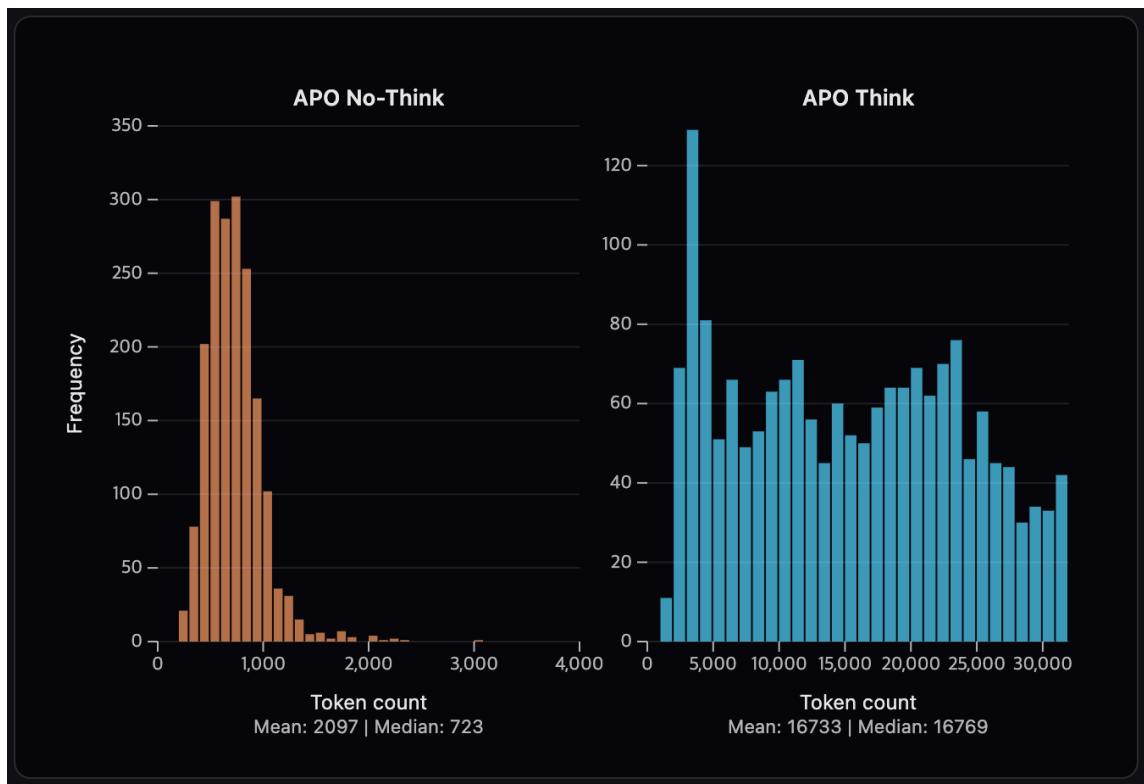
LLM에서 자기회귀 생성이 느리기 때문에, [verl](#) 과 [PipelineRL](#)과 같은 많은 프레임워크가 학습 처리량을 최대화하기 위해 완성의 비동기 생성과 모델 가중치의 "인플라이트" 업데이트를 추가했습니다. 이러한 접근 방식은 더 복잡하고 신중한 구현

이 필요하지만, 동기 학습 방법보다 4-5배 더 높은 학습 속도를 달성할 수 있습니다. 나중에 보겠지만, 학습 효율성의 이러한 개선은 긴 꼬리 토큰 분포를 가진 추론 모델에서 특히 두드러집니다.

SmoLM3의 경우, 주로 시간 제약과 오프라인 선호도 최적화로 이미 최고 수준인 모델이 있었기 때문에 RL을 완전히 건너뛰었습니다. 그러나 릴리스 이후, 주제를 다시 검토했으며 하이브리드 추론 모델에 RLVR을 적용한 교훈 중 일부를 공유하며 후속 학습 장을 마무리하겠습니다.

## 하이브리드 추론 모델에 RLVR 적용

하이브리드 추론 모델은 생성 길이가 추론 모드에 따라 상당히 다르기 때문에 RLVR에 추가적인 복잡성을 제기합니다. 예를 들어, 아래 그림에서 SmoLM3의 [final APO checkpoint](#)에 대한 AIMIE25의 토큰 길이 분포를 플롯합니다:



보시다시피, `/no_think` 모드는 약 2k 토큰의 중간 길이로 솔루션을 생성하는 반면, `/think` 모드는 16k 토큰과 두꺼운 꼬리 분포로 훨씬 더 큽니다. 이상적으로, 각각의 길이 분포를 너무 급진적으로 변경하지 않으면서 RLVR로 두 모드의 전반적인 성능을 개선하고 싶습니다.

이를 탐색하기 위해, 먼저 `/no_think` 모드 최적화에 집중했고 검증된 답이 있는 250k 이상의 수학 문제 데이터셋인 [Big-Math](#)에서 프롬프트의 부분집합을 가져왔습니다.

놀랍게도, GRPO를 순진하게 적용하면 일종의 **보상 해킹**으로 이어집니다: 긴 CoT를 내보내도록 프롬프트되지 않았음에도 불구하고, 모델은 보상을 증가시키기 위해 기본 추론 능력을 활용하는 것을 배웁니다. 아래 그림에서 보여주듯이, 보상뿐만 아니라 완성의 길이도 올라갑니다:



SmolLM3에 RLVR을 순진하게 적용하면 높은 보상과 개선된 다운스트림 성능을 달성하지만, 폭발하는 완성 길이의 비용이 따릅니다.

다시 말해, GRPO를 사용한 RLVR이 추론 모델의 /no\_think 모드를 /think 모드와 매우 유사하게 보이는 것으로 변환했습니다!

모델의 완성을 검토하면, 모델이 이제 긴 CoT를 생성하고 추론 모델과 관련된 "Wait, ..."와 같은 인지적 동작([Gandhi et al., 2025](#)) 중 일부도 포함하는 것을 볼 수 있습니다. 이에 대한 예시는 아래에 표시되어 있습니다:

However, since the jogger and the train are moving in the same direction, the relative speed between them is the difference of their speeds:

$$\text{v\_rel} = v_t - v_j = 12.7778 - 2.7778 = 10 \text{ m/s}$$

\*\*Wait,\*\* let me double-check the calculation:

\*\*Wait,\*\* 46 km/hr to m/s:

$$46 * 1000 \text{ m} / 3600 \text{ s} = (46 * 10) / 36 = 460 / 36 \approx 12.7778 \text{ m/s (correct)}$$

10 km/hr is:

$$10 * 1000 / 3600 = 10000 / 3600 \approx 2.7778 \text{ m/s (correct)}$$

$$\text{Difference: } 12.7778 - 2.7778 = 10 \text{ m/s (exact, since } 46-10=36 \text{ km/hr} \rightarrow 36*(1000/3600)=10 \text{ m/s exactly)}$$

The jogger is 340 meters ahead of the train's engine. When the train passes the jogger, the distance the train needs to cover relative to the jogger is 340 meters plus the length of the train? Wait, actually, let me think about the problem again.

\*\*Wait,\*\* actually, when the problem says the train takes 46 seconds to pass the jogger, that usually means the time from the front of the train to the end passing the jogger. The length of the train would then be equal to the relative speed multiplied by the time.

But first, the initial position: the jogger is 340 meters ahead of the engine. So the engine is 340 meters ahead of the jogger. Wait no, the wording says "340 m ahead of the engine", so the jogger is ahead by 340 meters. So when the jogger is running alongside, the engine of the train is 340 m away behind the jogger? Wait, maybe I need to clarify.

\*\*Wait\*\* the problem says: "A jogger running at 10 km/hr alongside a railway track is 340 m ahead of the engine of a train running at 46 km/hr in the same direction. The train takes 46 seconds to pass the jogger. How long is the train in meters?"

Hmm, so the jogger is 340 m ahead of the engine along the track. Since they're moving in the same direction, the train is behind the jogger by 340 meters. To pass the jogger, the train has to cover that 340 meters plus the length of the train itself?

### 과도한 길이 페널티로 보상 해킹 완화

이 문제는 특정 길이를 초과하는 완성에 페널티를 주는 **과도한 완성 페널티**를 포함하여 완화할 수 있습니다. 페널티는 두 개의 인수인 최대 완성 길이  $L_{max}$ 와 소프트 페널티 캐시  $L_{cache}$ 로 파라미터화됩니다. 이 페널티는 DAPO 논문([Yu et al., 2025](#))에서 제안된 개선 사항 중 하나이며 다음과 같이 보상 함수를 적용하는 것에 해당합니다:

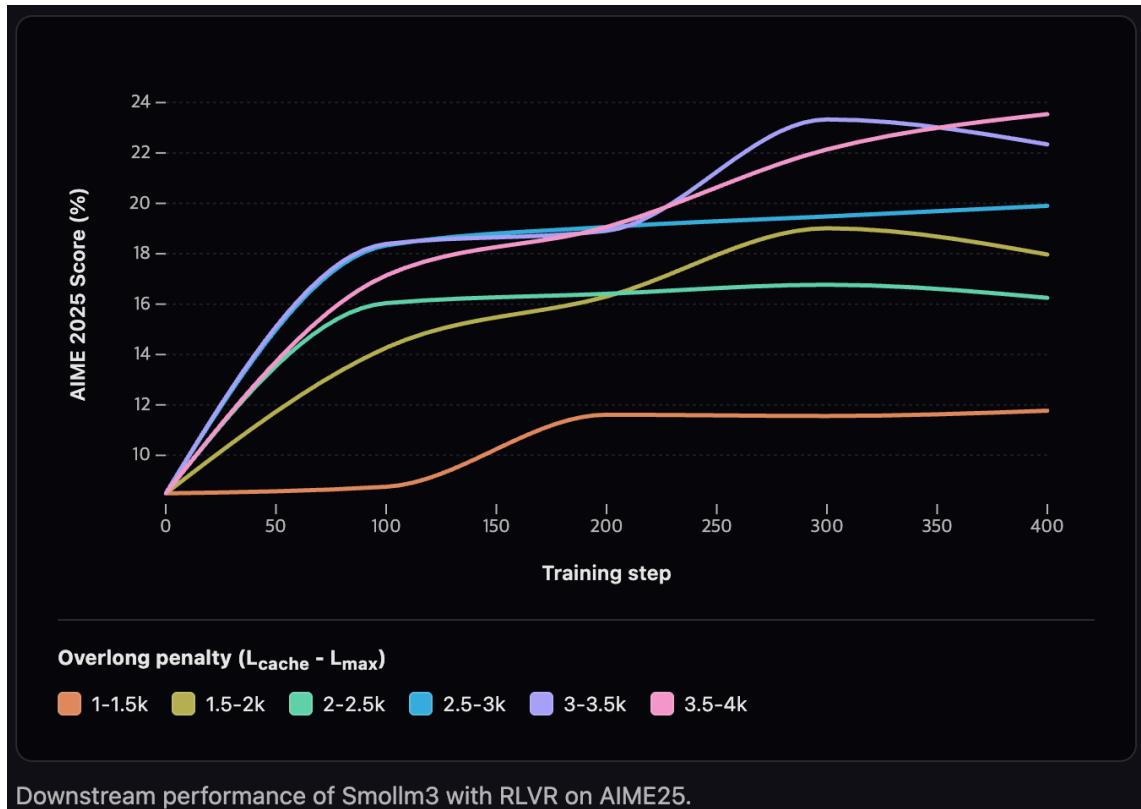
$$R_{\text{length}}(y) = \begin{cases} 0, & |y| \leq L_{max} - L_{cache} \frac{(L_{max} - L_{cache} - |y|)}{L_{cache}}, \\ L_{max} - L_{cache} < |y| \leq L_{max} - 1, & L_{max} - L_{cache} \end{cases}$$

이 페널티를 사용하면, 모델의 출력 분포를 직접 제어하고 응답 길이 증가와 성능 간의 트레이드오프를 측정할 수 있습니다.

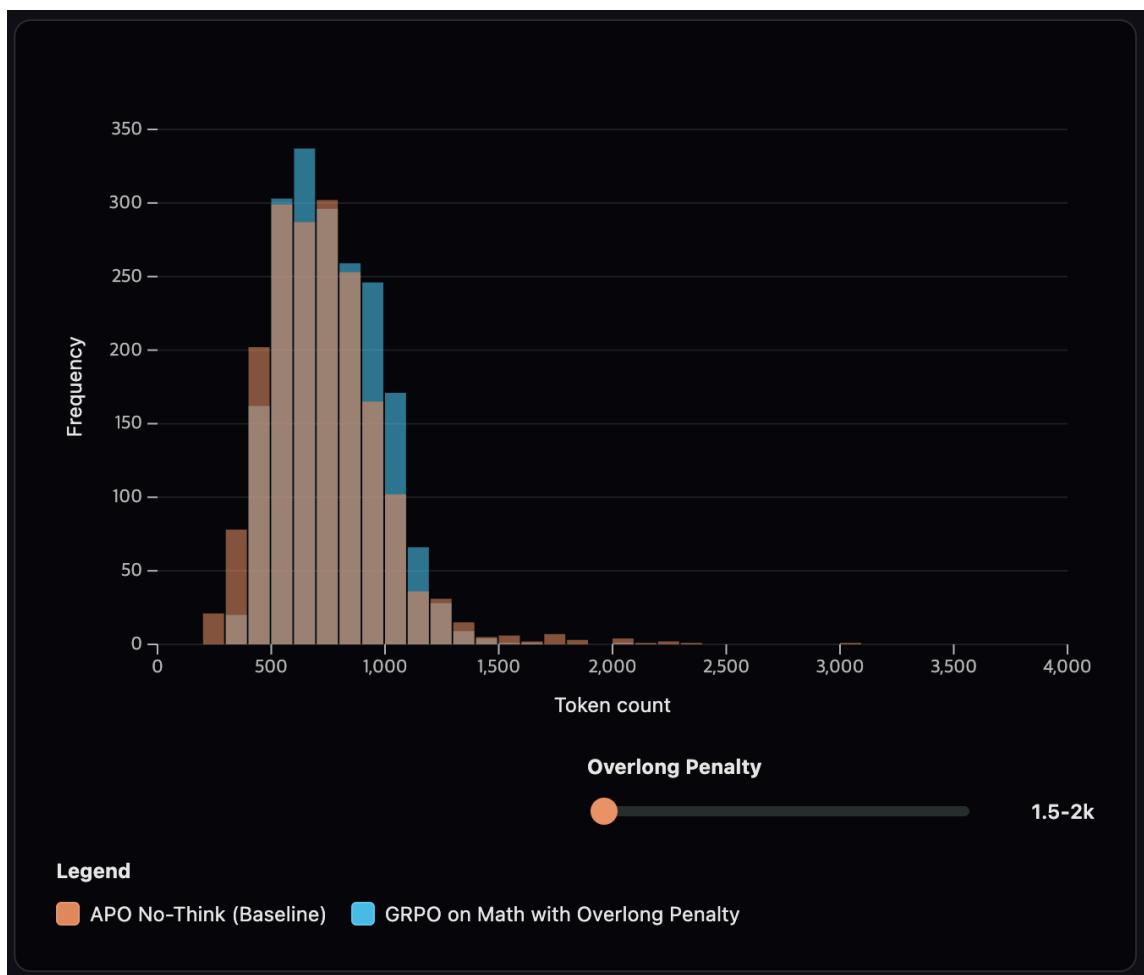
아래 그림에 예시가 표시되어 있으며, 과도한 길이 페널티를 512 토큰 단위로 1.5k에서 4k까지 변경합니다:



응답 길이와 성능 간의 트레이드오프는 AI-ME25에서의 개선을 검토할 때 더 명확해집니다:

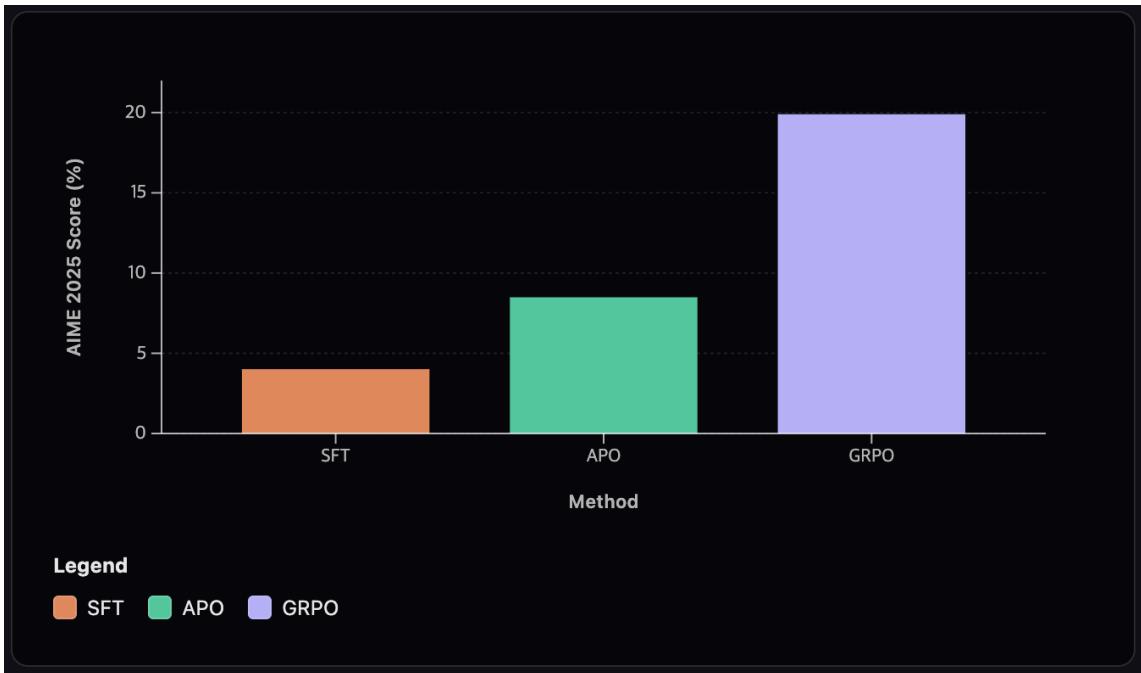


이제 과도한 길이 페널티가 다운스트림 성능에 어떻게 영향을 미치는지 명확하게 볼 수 있으며, 2-4k 범위의 페널티가 토큰 분포를 제어하면서 상당한 개선을 생성합니다. 아래 그림에서 보여주듯이, 스텝 400에서 체크포인트를 가져오면, 다양한 페널티 범위에 걸쳐 초기 정책과 최종 모델 간의 출력 토큰 분포를 비교할 수 있습니다:



### 모든 것을 하나로 모으기

2.5-3k 범위의 길이 페널티를 적용하면 성능과 응답 길이 간의 최고의 트레이드오프를 제공함을 발견했으며, 아래 그림은 GRPO가 APO와 같은 오프라인 방법에 비해 AIME 2025에서 성능을 거의 두 배로 높임을 보여줍니다:



이제 `/no_think` 추론 모드에서 성능을 개선하는 방법을 알았으므로, RL 학습 파이프라인의 다음 단계는 두 추론 모드에서 동시에 모델을 공동 학습하는 것입니다. 그러나 각 모드가 자체 길이 페널티를 필요로 하고 상호작용이 지금까지 불안정한 학습을 생성했기 때문에 이것이 꽤 어려운 문제임을 발견했습니다. 이것은 하이브리드 추론 모델에 RL을 적용하려는 주요 도전을 강조하며, Qwen과 같은 모델 개발자가 [instruct](#) 과 [reasoning](#) 변형을 별도로 릴리스하는 새로운 추세에 이것이 반영된 것을 볼 수 있습니다.

우리의 실험은 RLVRII 추론 동작을 효과적으로 조종할 수 있지만, 신중한 보상 형성과 안정성 메커니즘이 있어야만 가능함을 보여줍니다. 이러한 복잡성을 감안할 때, 강화 학습이 유일한 실행 가능한 전진 경로인지 물어볼 가치가 있습니다. 사실, 최근 문헌에서 여러 가지 더 가벼운 온폴리시 최적화 전략이 제안되었지만, 오픈소스 커뮤니티에서 놀랍게도 탐구가 부족합니다. 그 중 일부를 살펴보며 이 장을 마무리하겠습니다.

## RL이 유일한 게임인가요?

온폴리시 학습에 대한 다른 접근 방식은 선호도 최적화와 종류를 모델이 진화함에 따라 학습 신호를 새로 고치는 반복 루프로 확장합니다:

- **Online DPO:** 고정된 선호도 데이터셋에서 한 번 학습하는 대신, 모델이 지속적으로 새로운 응답을 샘플링하고, 새로운 선호도 레이블(보상 모델이나 LLM 그레이더로부터)을 수집하고, 자체를 업데이트합니다. 이것은 최적화를 온폴리시로 유지하고 학습 데이터와 모델의 현재 동작 간의 드리프트를 줄입니다([Guo et al., 2024](#)).
- **온폴리시 종류:** 선호도 대신, 신호가 더 강한 교사 모델에서 옵니다. 학생이 모든 학습 스텝에서 응답을 샘플링하고 이러한 샘플에서 학생과 교사 로짓 간의 KL 발산이 학습 신호를 제공합니다. 이를 통해 학생이 명시적인 선호도 레이블이나 검증기 없이 교사의 능력을 지속적으로 흡수할 수 있습니다([Agarwal et al., 2024](#)).

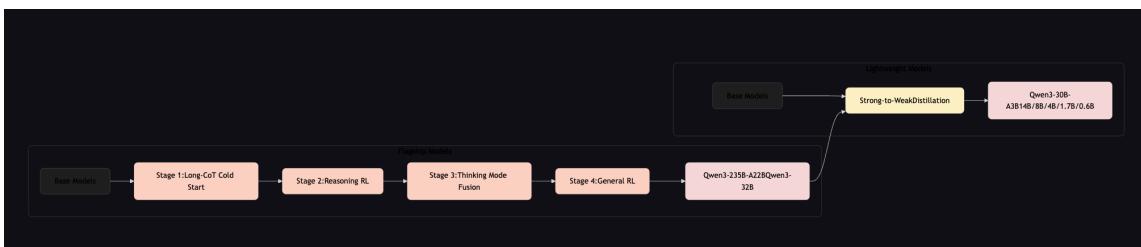
이러한 방법은 정적 선호도 최적화와 전체 RL 사이의 경계를 흐리게 합니다: 모델의 현재 분포에 적응하는 이점을 여전히 얻지만, 강화 학습 루프를 설계하고 안정화하는 전체 복잡성 없이요.

## 어떤 방법을 선택해야 하나요?

어떤 온풀리시 방법이 "최고"인지에 대한 수많은 연구 논문이 있지만, 실제로 결정은 아래 표에 표시된 몇 가지 요소에 따라 달라집니다:

| 알고리즘       | 사용 시기                                                                                               | 트레이드오프                                                                | 최적의 모델 크기                                        |
|------------|-----------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------|--------------------------------------------------|
| Online DPO | 선호도 레이블을 저렴하게 얻을 수 있을 때. 진화하는 분포에 동작을 정렬하는 데 최적.                                                    | 반복적으로 스케일링하기 쉽고, RL보다 안정적이지만, 레이블 품질과 커버리지에 의존. 소수의 학습 프레임워크에서 지원.    | 선호도가 모방 이상의 개선을 포착하는 모든 크기.                      |
| 온풀리시 종류    | 더 강한 교사 모델에 접근할 수 있고 능력을 효율적으로 전이하고 싶을 때.                                                           | 구현이 간단하고, 실행이 저렴하고, 교사 편향을 상속하고, 천장이 교사에 의해 제한됨. TRL과 NemoRL에서 지원.    | 소형에서 중형 모델(<30B)에 가장 효과적.                        |
| 강화 학습      | 검증 가능한 보상이나 다단계 추론/계획이 필요한 태스크가 있을 때 최적. 보상 모델과 함께 사용할 수 있지만, 모델이 보상 모델의 약점을 이용하는 보상 해킹과 같은 도전이 있음. | 유연하고 강력하지만, 비용이 많이 들고 안정화하기 어려움; 신중한 보상 형성 필요. 대부분의 후속 학습 프레임워크에서 지원. | 중형에서 대형 모델(20B+), 추가 용량이 구조화된 보상 신호를 활용할 수 있는 곳. |

오픈소스 생태계에서 GRPO와 REINFORCE와 같은 강화 학습 방법이 가장 널리 사용되는 경향이 있지만, Qwen3 기술 보고서([A. Yang, Li, et al., 2025](#))는 32B 미만의 모델을 학습시키기 위해 온풀리시 종류의 사용을 강조했습니다:



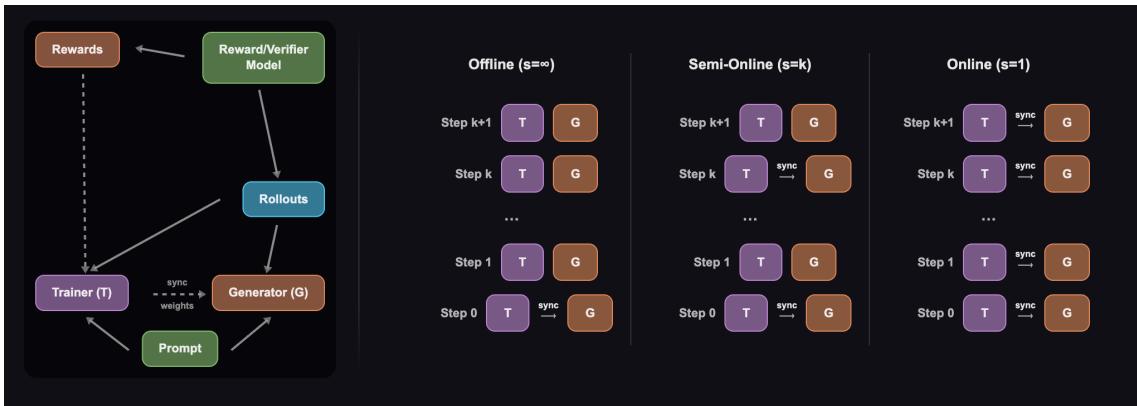
소형 모델에서 온풀리시 종류의 한 가지 흥미로운 특성은 일반적으로 컴퓨팅 비용의 일부로 RL 기반 방법을 능가한다는 것입니다. 이것은 프롬프트당 여러 룰아웃을 생성하는 대신, 단일 순방향-역방향 패스에서 교사가 등급을 매기는 하나만 샘플링하기 때문입니다. Qwen3 기술 보고서가 보여주듯이, GRPO에 비한 이득은 상당할 수 있습니다:

| 방법                       | AIME'24 | AIME'25 | MATH500 | LiveCodeBench v5 | MMLU-Redux | GPQA-Diamond | GPU 시간 |
|--------------------------|---------|---------|---------|------------------|------------|--------------|--------|
| Off-policy Distillation  | 55.0    | 42.8    | 92.4    | 42.0             | 86.4       | 55.6         | -      |
| + Reinforcement Learning | 67.6    | 55.5    | 94.8    | 52.9             | 86.9       | 61.3         | 17,920 |
| + On-policy Distillation | 74.4    | 65.5    | 97.0    | 60.3             | 88.3       | 63.3         | 1,800  |

더 최근에, [Thinking Machines](#)는 온풀리시 종류가 치명적 망각을 완화하는 데도 효과적임을 보여주었습니다. 후속 학습된 모델이 새로운 도메인에서 추가로 학습되고 이전 성능이 퇴행하는 경우입니다. 아래 표에서, Qwen3-8b의 채팅 성능 (IFEval)이 내부 데이터에서 파인튜닝될 때 급락하지만, 저렴한 종류로 동작을 복원할 수 있음을 보여줍니다:

우리 자신도 온풀리시 종류에 꽤 흥분하고 있습니다. 더 작고 태스크 특화된 모델로 종류할 수 있는 유능한 오픈 가중치 LLM의 거대한 다양성이 있기 때문입니다. 그러나 모든 온풀리시 종류 방법의 한 가지 약점은 교사와 학생이 동일한 토크나이저를 공유해야 한다는 것입니다. 이를 해결하기 위해, 어떤 교사는 어떤 학생으로든 종류할 수 있는 \*\*General On-Policy Logit Distillation(GOLD)\*\*라는 새로운 방법을 개발했습니다. 이러한 주제에 관심이 있다면 [technical write-up](#)를 확인하는 것을 권장합니다.

마찬가지로, FAIR의 연구자들은 DPO에 대해 완전히 오프풀리시에서 온풀리시로의 효과를 비교했고 훨씬 적은 컴퓨팅을 사용하여 GRPO의 성능을 일치시킬 수 있음을 보여주었습니다([Lanchantin et al., 2025](#)):



그들의 논문에서 보여주듯이, online DPO는 수학 태스크에서 잘 작동하며 심지어 반온풀리시 변형도 많은 스텝 오프풀리시임에도 불구하고 비슷한 성능을 달성합니다:

| 학습 방법                         | Math500 | NuminaMath | AMC23 |
|-------------------------------|---------|------------|-------|
| Seed (Llama-3.1-8B-Instruct)  | 47.4    | 33.9       | 23.7  |
| Offline DPO ( $s = \infty$ )  | 53.7    | 36.4       | 28.8  |
| Semi-online DPO ( $s = 100$ ) | 58.9    | 39.3       | 35.1  |
| Semi-online DPO ( $s = 10$ )  | 57.2    | 39.4       | 31.4  |
| Online DPO ( $s = 1$ )        | 58.7    | 39.6       | 32.9  |
| GRPO                          | 58.1    | 38.8       | 33.6  |

전반적으로, RL을 효과적으로 스케일링하는 것([Khatri et al., 2025](#))과 계산 효율성을 위한 다른 방법을 탐색하는 것 모두에서 아직 해야 할 일이 많이 남아 있다고 느낍니다. 정말 흥분되는 시기입니다!

## 후속 학습 마무리

여기까지 왔다면, 축하합니다: 이제 후속 학습에서 성공하는 데 필요한 모든 핵심 재료를 갖추었습니다. 이제 많은 실험을 실행하고 다양한 알고리즘을 테스트하여 SOTA 결과를 얻을 준비가 되었습니다.

그러나 아마 깨달았듯이, 훌륭한 모델을 학습시키는 방법을 아는 것은 이야기의 절반에 불과합니다. 실제로 그 모델들에 생명을 불어넣으려면 올바른 인프라가 필요합니다. LLM 학습의 숨은 영웅으로 이 대작을 마무리합시다.

## 인프라 - 숨은 영웅

이제 모델 생성과 학습에 대해 우리가 아는 모든 것을 알았으니, 프로젝트(그리고 은행 계좌)를 성공시키거나 망칠 수 있는 중요하지만 과소평가된 구성 요소인 인프라를 다루겠습니다. 프레임워크, 아키텍처, 데이터 큐레이션에 집중하든, 인프라 기본을 이해하면 학습 병목 현상을 식별하고, 병렬성 전략을 최적화하고, 처리량 문제를 디버그하는 데 도움이 됩니다. (최소한, 인프라 팀과의 커뮤니케이션을 개선합니다 😊).

모델을 학습시키는 대부분의 사람들은 아키텍처와 데이터에 깊이 관심을 갖지만, 인프라 세부 사항을 이해하는 사람은 거의 없습니다. 인프라 전문 지식은 일반적으로 프레임워크 개발자와 클러스터 엔지니어에게 있으며, 나머지에게는 해결된 문제로 취급됩니다: GPU를 임대하고, PyTorch를 설치하면 준비 완료. 우리는 거의 한 달 동안 384개의 H100에서 SmolLM3

를 학습시켰고, 총 11조 토큰을 처리했습니다... 그리고 이것은 순탄한 여정이 아니었습니다! 그 기간 동안 노드 장애, 스토리지 문제, 실행 재시작을 처리했습니다(학습 마라톤 섹션 참조). 이러한 문제에 대비하고 학습을 원활하고 저유지보수로 유지하기 위한 좋은 비상 계획과 전략이 필요합니다.

이 장은 그 지식 격차를 메우는 것을 목표로 합니다. 학습에 중요한 질문에 초점을 맞춘 하드웨어 레이어에 대한 실용적인 가이드로 생각하세요. (참고: 각 하위 섹션은 TL;DR로 시작하므로 깊이 수준을 선택할 수 있습니다.)

처음 두 섹션은 하드웨어가 어떻게 작동하는지의 기본을 다룹니다: GPU는 실제로 무엇으로 구성되나요? 메모리 계층은 어떻게 작동하나요? CPU와 GPU는 어떻게 통신하나요? GPU를 획득할 때 고려해야 할 사항과 장기 학습 실행에 커밋하기 전에 테스트하는 방법도 살펴보겠습니다. 가장 중요한 것은, 각 단계에서 이러한 시스템을 직접 측정하고 진단하는 방법을 보여드리겠습니다. 다음 섹션은 더 응용적이며, 인프라를 장애에 복원력 있게 만드는 방법과 학습 처리량을 최대로 최적화하는 방법을 볼 것입니다.

이 장의 게임 이름은 병목 현상을 찾아 수정하는 것입니다!

특정 설계 결정이 왜 중요한지에 대한 직관을 구축하는 것으로 생각하세요. 모델의 활성화가 각각 다른 대역폭과 지연 시간 특성을 가진 여러 수준의 캐시를 통해 흘러야 한다는 것을 이해하면, 자연스럽게 데이터 이동을 최소화하도록 학습을 구조화하는 방법에 대해 생각하기 시작할 것입니다. 노드 간 통신이 노드 내 통신보다 몇 자릿수 더 느리다는 것을 보면, 병렬성 전략이 왜 그렇게 중요한지 이해하게 될 것입니다.

GPU를 열어서 안에 무엇이 있는지 보는 것으로 시작합시다.

## GPU 내부: 내부 아키텍처

GPU는 근본적으로 지연 시간보다 처리량에 최적화된 대규모 병렬 프로세서입니다. 몇 가지 복잡한 명령 스트림을 빠르게 실행하는 데 뛰어난 CPU와 달리, GPU는 수천 개의 간단한 연산을 동시에 실행하여 성능을 달성합니다.

GPU 성능을 이해하는 핵심은 단순히 원시 컴퓨팅 파워에 관한 것이 아니라, 계산과 데이터 이동 간의 상호작용에 관한 것임을 인식하는 데 있습니다. GPU는 이론적 컴퓨팅의 테라플롭스를 가질 수 있지만, 데이터가 컴퓨팅 유닛에 충분히 빠르게 도달할 수 없다면 그 잠재력은 사용되지 않습니다. 이것이 메모리 계층(데이터가 어떻게 이동하는지)과 컴퓨팅 파이프라인(작업이 어떻게 수행되는지) 모두를 이해해야 하는 이유입니다.

따라서 가장 높은 수준에서 GPU는 두 가지 필수 작업을 수행합니다:

1. 데이터 이동 및 저장 (메모리 시스템)
2. 데이터로 유용한 작업 수행 (컴퓨팅 파이프라인)

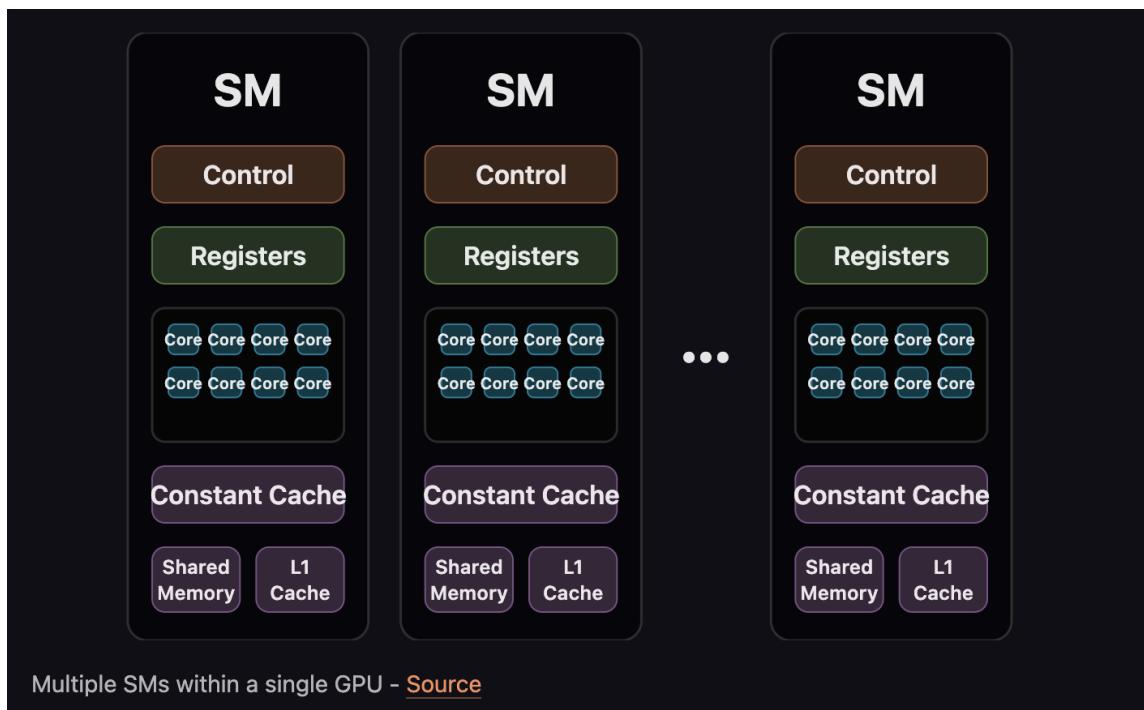
## 컴퓨팅 유닛과 FLOPs

**TL;DR:** GPU는 FLOPs(초당 부동소수점 연산)로 성능을 측정합니다. H100과 같은 현대 GPU는 더 낮은 정밀도에서 극적으로 더 높은 처리량을 제공합니다: FP32의 67 TFLOPs 대비 BF16의 990 TFLOPs. 그러나 실제 성능은 메모리 병목으로 인해 이론적 피크의 70-77%입니다. 최신 학습은 모델 플롭스 활용도(MFU)라고도 알려진 20-41%의 종단 간 효율성을 달성합니다. 학습 실행을 계획할 때 마케팅 사양이 아닌 현실적인 숫자를 사용하세요.

GPU 컴퓨팅 성능은 FLOPs(초당 부동소수점 연산)로 측정됩니다. FLOP는 단일 산술 연산으로, 일반적으로  $a + b$ 와 같은 부동 소수점 덧셈이며, 현대 GPU는 초당 수조 개(TFLOPs)를 실행할 수 있습니다.

GPU 컴퓨팅의 기본 구성 요소는 Streaming Multiprocessors(SMs)로, 명령을 병렬로 실행하는 독립적인 처리 유닛입니다. 각 SM은 두 가지 유형의 코어를 포함합니다: 표준 부동소수점 연산을 위한 CUDA 코어와 딥러닝의 주력 연산(트랜스포머 성능에 중요)인 행렬 곱셈에 최적화된 특수 Tensor 코어입니다.

현대 GPU는 칩 전체에 수백 개의 이러한 SM을 구성합니다! 예를 들어, H100 SXM5 버전(우리 클러스터에서 사용하는 GPU)은 132개의 SM을 포함합니다. 각 SM은 독립적으로 작동하며, 워프라고 불리는 32개 스레드 그룹을 동기화하여 실행합니다. 이를 돋기 위해 SM은 또 다른 구성 요소인 워프 스케줄러에 의존합니다: 다른 워프에 명령을 균형 있게 배분하여, 하나가 지연될 때 워프 간에 전환하여 SM이 "지연 시간을 숨길" 수 있게 합니다. 이 SIMT(Single Instruction, Multiple Thread) 실행 모델은 워프의 모든 스레드가 다른 데이터에서 동일한 명령을 동시에 실행함을 의미합니다.



수백 개의 SM이 각각 여러 워프를 동시에 실행하면서, 단일 GPU는 수만 개의 스레드를 동시에 실행할 수 있습니다. 이 대규모 병렬성이 GPU가 딥러닝 워크로드를 지배하는 행렬 연산에서 뛰어난 이유입니다!

FLOPs를 논의할 때 정밀도가 중요합니다. Tensor 코어는 다른 정밀도(FP64, FP32, FP16/BF16, FP8, FP4 - [부동소수점 숫자에 대한 상기는 여기 참조](#))에서 작동할 수 있습니다. 따라서 달성 가능한 처리량은 데이터 유형에 따라 종종 자릿수만큼 극적으로 달라집니다. 더 낮은 정밀도 형식은 더 적은 데이터 이동을 필요로 하고 동일한 실리콘 면적에 더 많은 연산을 압축할 수 있기 때문에 더 높은 처리량을 가능하게 하지만, 이전에는 학습 불안정 때문에 피했습니다. 그러나 오늘날에는 새로운 기술 범위 덕분에 학습과 추론 모두 점점 더 낮은 정밀도로 밀려나 FP8과 FP4에 도달하고 있습니다.

| FP8 혼합 정밀도 학습에 대한 우리의 경험에 대해 더 알고 싶다면 *Ultra Scale Playbook*을 확인하세요.

아래 표는 다양한 NVIDIA GPU 세대와 정밀도에 걸친 이론적 피크 성능을 보여줍니다:

| 정밀도/GPU 유형 | A100 | H100 | H200 | B100 | B200 |
|------------|------|------|------|------|------|
| FP64       | 9.7  | 34   | 34   | 40   | 40   |

|           |      |      |      |      |       |
|-----------|------|------|------|------|-------|
| FP32      | 19.5 | 67   | 67   | 80   | 80    |
| FP16/BF16 | 312  | 990  | 990  | 1750 | 2250  |
| FP8       | -    | 3960 | 3960 | 4500 | 5000  |
| FP4       | -    | -    | -    | 9000 | 10000 |

정밀도와 GPU 세대에 따른 이론적 TFLOPs를 보여주는 표. 출처: Nvidia, SemiAnalysis

더 낮은 정밀도에서의 처리량 극적인 증가는 단순히 원시 속도에 관한 것이 아니라, 수치 계산에 대해 생각하는 방식의 근본적인 변화를 반영합니다. FP8과 FP4는 모델이 **와트당 및 초당 더 많은 연산**을 수행할 수 있게 하여, 규모에서의 학습과 추론 모두에 필수적입니다. H100의 FP8에서 3960 TFLOPs는 FP16/BF16에 비해 4배 향상을 나타내고, B200의 FP4에서 10,000 TFLOPs는 이를 더욱 밀어붙입니다.

**숫자 이해하기:** 이러한 이론적 피크 FLOPs는 이상적인 조건에서 달성 가능한 최대 계산 처리량을 나타내며, 모든 컴퓨팅 유닛이 완전히 활용되고 데이터가 즉시 사용 가능할 때입니다. 실제로 실제 성능은 워크로드가 컴퓨팅 유닛에 데이터를 얼마나 잘 공급할 수 있는지와 연산이 사용 가능한 하드웨어에 효율적으로 매핑될 수 있는지에 크게 의존합니다.

SmoILM3의 경우, NVIDIA H100 80GB HBM3 GPU에서 학습할 예정이었으므로, 먼저 H100의 이론적 TFLOPs 사양을 실제 성능과 비교 테스트하고 싶었습니다. 이를 위해 [SemiAnalysis GEMM 벤치마크](#)를 사용했습니다: [Meta의 Llama 70B 학습에서 실제 행렬 곱셈 형태에 대한 처리량을 테스트합니다.](#)

| 형태 (M, N, K)        | FP64<br>torch.matmul | FP32<br>torch.matmul | FP16<br>torch.matmul | BF16<br>torch.matmul | FP8 TE.Linear<br>(autocast, bias=False) | FP8 torch._scaled_mm<br>(e5m2/e4m3fn) | FP8 torch._scaled_mm<br>(e4m3) |
|---------------------|----------------------|----------------------|----------------------|----------------------|-----------------------------------------|---------------------------------------|--------------------------------|
| (16384, 8192, 1280) | 51.5 TFLOPS          | 364.5 TFLOPS         | 686.5 TFLOPS         | 714.5 TFLOPS         | 837.6 TFLOPS                            | 1226.7 TFLOPS                         | 1209.7 TFLOPS                  |
| (16384, 1024, 8192) | 56.1 TFLOPS          | 396.1 TFLOPS         | 720.0 TFLOPS         | 757.7 TFLOPS         | 547.3 TFLOPS                            | 1366.2 TFLOPS                         | 1329.7 TFLOPS                  |
| (16384, 8192, 7168) | 49.5 TFLOPS          | 356.5 TFLOPS         | 727.1 TFLOPS         | 752.9 TFLOPS         | 1120.8 TFLOPS                           | 1464.6 TFLOPS                         | 1456.6 TFLOPS                  |
| (16384, 3584, 8192) | 51.0 TFLOPS          | 373.3 TFLOPS         | 732.2 TFLOPS         | 733.0 TFLOPS         | 952.9 TFLOPS                            | 1445.7 TFLOPS                         | 1370.3 TFLOPS                  |
| (8192, 8192, 8192)  | 51.4 TFLOPS          | 372.7 TFLOPS         | 724.9 TFLOPS         | 729.4 TFLOPS         | 1029.1 TFLOPS                           | 1404.4 TFLOPS                         | 1397.5 TFLOPS                  |

Llama 70B 학습 워크로드에서 정밀도와 행렬 형태에 따라 H100 80GB에서 달성된 TFLOPs를 보여주는 표

**이론적 성능 검증:** 우리 실험은 이론적 피크와 달성 가능한 성능 간의 격차를 드러냈습니다.

FP64 Tensor 코어 연산의 경우, 49-56 TFLOPs를 달성했으며, 이론적 피크(67 TFLOPs)의 74-84%를 나타냅니다. TF32(TensorFloat-32, PyTorch가 Tensor 코어에서 FP32 텐서에 기본적으로 사용)의 경우, 356-396 TFLOPs를 달성했으며, 이론적 피크(~495 TFLOPs 밀집)의 72-80%를 나타냅니다. 이것이 우수한 하드웨어 활용을 보여주지만, 이러한 정밀도는 현대 딥러닝 학습에서 거의 사용되지 않습니다: FP64는 계산 비용 때문에, TF32는 BF16과 FP8과 같은 더 낮은 정밀도가 더 나은 성능을 제공하기 때문입니다.

NVIDIA 사양은 종종 2:4 구조적 희소성 패턴을 가정하는 희소 성능(TF32의 경우 989 TFLOPs)을 나열합니다. 우리 벤치마크가 테스트하는 밀집 연산은 희소 피크의 대략 절반(~495 TFLOPs)을 달성합니다.

BF16 연산의 경우, 다양한 행렬 형태에 걸쳐 일관되게 714-758 TFLOPs를 달성했으며, H100의 이론적 990 TFLOPs 피크의 약 72-77%입니다. 이것은 실제로 실제 워크로드에 대한 우수한 활용률입니다!

### 모델 FLOPs 활용도(MFU)

커널 벤치마크가 원시 TFLOPS를 측정하는 반면, 종단 간 학습 효율성은 \*\*모델 FLOPs 활용도(MFU)\*\*로 포착됩니다. 유용한 모델 계산과 이론적 피크 하드웨어 성능의 비율입니다.

우리 BF16 matmul 벤치마크는 H100의 이론적 피크의 72-77%를 달성했음을 보여주었습니다. 이것은 우리 설정에서 커널 수준에서 달성 가능한 상한을 나타냅니다. 더 복잡한 비matmul 연산, 통신 오버헤드, 기타 보조 계산으로 인해 종단 간 학습 MFU는 반드시 더 낮을 것입니다.

**학습에서의 최신 MFU:** Meta는 Llama 3 405B를 학습할 때 38-41%를 달성했고, DeepSeek-v3는 MoE 아키텍처와 관련된 더 타이트한 통신 병목이 있는 GPU에서 ~20-30%에 도달했습니다. SmoILM3의 경우, 나중에 보겠듯이 ~30% MFU를 달성했습니다. 격차의 대부분은 분산 학습에서의 노드 간 통신 오버헤드에서 나옵니다. 커널 수준 상한이 ~77%임을 감안하면, 이러한 종단 간 숫자는 달성 가능한 matmul 성능에 비해 대략 50-55% 효율성을 나타냅니다. 추론 워크로드는 원시 matmul 성능에 더 가까운 >70%의 더 높은 MFU에 도달할 수 있지만, 프로덕션 배포의 공개된 결과는 드뭅니다.

FP8 결과는 더 미묘합니다. 3가지 다른 행렬 곱셈 방법/커널에 대한 결과를 살펴보겠습니다.

커널은 CUDA 코드의 단위입니다.

e4m3 정밀도로 PyTorch의 `torch._scaled_mm` 커널을 사용하여, 행렬 형태에 따라 1,210-1,457 TFLOPs를 달성했으며, 이론적 3,960 TFLOPs 피크의 대략 31-37%입니다. 😊 왜일까요? 이 더 낮은 활용률(FP8에서)은 실제로 나쁜 성능을 나타내지 않습니다; 오히려 컴퓨팅 처리량이 증가함에 따라 이러한 연산이 점점 메모리 바운드가 됨을 반영합니다.

[Tensor Cores](#)는 메모리 시스템이 전달할 수 있는 것보다 더 빠르게 FP8 데이터를 처리할 수 있어, 메모리 대역폭이 제한요소가 됩니다.

[Transformer Engine](#)의 `TE.Linear`은 형태에 따라 547-1,121 TFLOPs를 달성한 반면, `torch._scaled_mm`은 일관되게 더 높은 처리량을 제공했습니다. 이것은 중요한 교훈을 강조합니다: 커널 구현이 상당히 중요하며, API 선택이 동일한 하드웨어 능력을 대상으로 해도 성능에 2-3배 영향을 미칠 수 있습니다.

SmoILM3의 학습에서, 이러한 실용적인 측정은 현실적인 처리량 기대치를 설정하는 데 도움이 되었습니다. 자체 학습 실행을 계획할 때, 기대치를 설정하기 위해 이론적 피크보다 이러한 달성 가능한 숫자를 사용하세요.

## 🔧 Compute Capability

올바른 커널 API를 선택하는 것 외에도, 해당 커널이 올바른 하드웨어 세대를 위해 컴파일되었는지 확인해야 합니다.

Compute Capability(CC)는 물리적 GPU 세부 사항을 PTX 명령 세트에서 추상화하는 NVIDIA의 버전 관리 시스템입니다. GPU가 지원하는 명령과 기능을 결정합니다.

**왜 이것이 중요한가?** 특정 compute capability를 위해 컴파일된 커널은 더 오래된 하드웨어에서 실행되지 않을 수 있고, 코드가 대상 GPU의 CC를 위해 컴파일되지 않으면 최적화를 놓칠 수 있습니다. 더 나쁜 것은, 프레임워크가 조용히 차선의 커널을 선택할 수 있습니다. 우리는 H100에서 PyTorch가 `sm_75` 커널(compute capability 7.5, Turing GPU용으로 설계됨)을 선택하여 미스터리한 속도 저하를 유발하는 것을 발견했습니다. 이것은 [PyTorch 커뮤니티에서 문서화된 유사한 문제](#)로, 프레임워크가 종종 최적의 커널보다 더 오래되고 더 호환 가능한 커널을 기본값으로 설정합니다. 이 곁보기기에 사소한 세부 사항이 동일한 하드웨어에서 720 TFLOPS 또는 500 TFLOPS를 얻는 차이를 만들 수 있습니다.

사전 컴파일된 라이브러리나 커스텀 커널을 사용할 때, 호환성과 최적 성능을 보장하기 위해 하드웨어의 *compute capability*를 위해 빌드되었는지 항상 확인하세요. 예를 들어, *sm90\_xmma\_gemm\_...\_cublas*는 SM 9.0(H100) 사용하는 *compute capability* 9.0)을 위해 컴파일된 커널을 나타냅니다.

`nvidia-smi --query-gpu=compute_cap` 으로 GPU의 *compute capability*를 확인하거나 [NVIDIA CUDA C Programming Guide](#)의 *Compute Capability* 섹션에서 기술 사양을 찾을 수 있습니다.

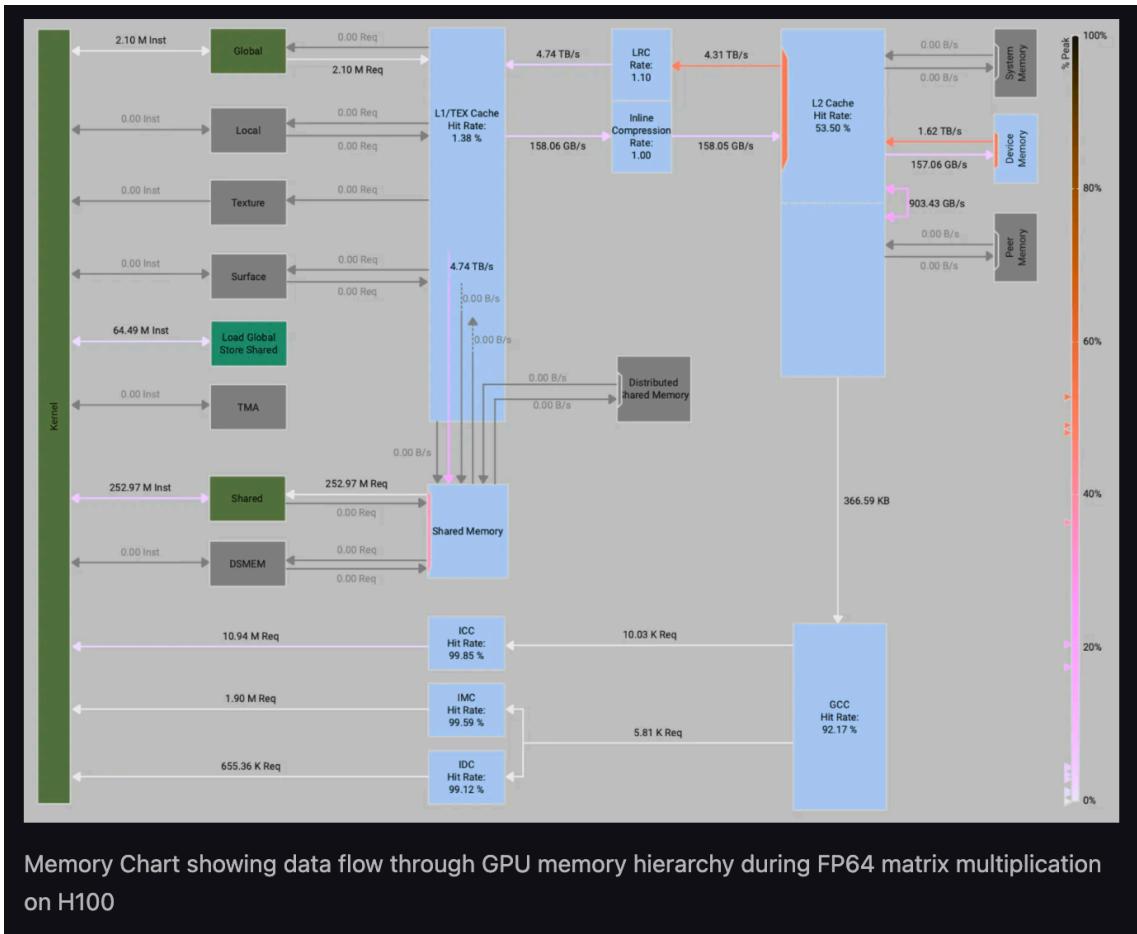
보았듯이, GPU 메모리는 계산이 낮은 정밀도에서 너무 빨라질 때 병목이 되는 것 같습니다. GPU 메모리가 어떻게 작동하는지, 그리고 병목이 발생하는 원인을 살펴보겠습니다!

## GPU 메모리 계층: 레지스터에서 HBM까지

계산을 수행하기 위해 GPU는 메모리에서 읽기/쓰기를 해야 하므로, 이러한 전송이 어떤 속도로 발생하는지 아는 것이 중요합니다. GPU 메모리 계층을 이해하는 것은 고성능 커널을 작성하는 데 중요합니다.

**TL;DR:** GPU는 빠르지만 작은 것(레지스터, 공유 메모리)에서 느리지만 큰 것(HBM 메인 메모리)까지 계층 구조로 메모리를 구성합니다. 이 계층 구조를 이해하는 것이 중요한 이유는 현대 AI가 종종 **메모리 바운드**이기 때문입니다. 병목은 데이터에 대한 계산이 아니라 데이터 이동입니다. 연산자 융합(Flash Attention과 같은)은 중간 결과를 느린 HBM에 쓰는 대신 빠른 온칩 메모리에 유지하여 2-4배 속도 향상을 달성합니다. 벤치마크는 H100의 HBM3가 실제로 ~3 TB/s를 제공하며, 대용량 전송에 대한 이론적 사양과 일치함을 보여줍니다.

실제로 [메모리 연산이 GPU를 통해 어떻게 흐르는지 시각화](#)하기 위해, 먼저 NVIDIA Nsight Compute의 Memory Chart를 살펴보겠습니다. 선택한 모든 커널에 대해 데이터가 다른 메모리 유닛 간에 어떻게 이동하는지 그래픽으로 보여주는 프로파일링 그래프입니다:



일반적으로, Memory Chart는 Global, Local, Texture, Surface, Shared 메모리와 같은 논리적 유닛(녹색)과 L1/TEX Cache, Shared Memory, L2 Cache, Device Memory와 같은 물리적 유닛(파란색) 모두를 보여줍니다. 유닛 간의 링크는 유닛 간에 발생하는 명령(Inst) 또는 요청(Req) 수를 나타내며, 색상은 피크 활용률의 백분율을 나타냅니다: 미사용(0%)에서 피크 성능에서 작동(100%)까지.

[NVIDIA Nsight Compute](#)를 사용하여 모든 커널에 대해 이 memory chart를 생성할 수 있습니다:

```
메모리 워크로드 분석으로 특정 커널 프로파일링
ncu --set full --kernel-name "your_kernel_name" --launch-skip 0 --launch-count 1 python
your_script.py
프로파일링이 완료되면, Memory Chart를 보기 위해 Nsight Compute GUI에서 결과를 업니다
```

여러 가지 핵심 통찰을 제공합니다:

- **병목 식별:** 포화된 링크(빨간색/주황색으로 표시)는 데이터 이동이 제한되는 위치를 나타냅니다
- **캐시 효율성:** L1/TEX 및 L2 캐시의 히트율은 커널이 메모리 계층을 얼마나 잘 활용하는지 드러냅니다
- **메모리 접근 패턴:** 논리적 유닛과 물리적 유닛 간의 흐름은 커널이 좋은 공간적/시간적 지역성을 가지고 있는지 보여줍니다

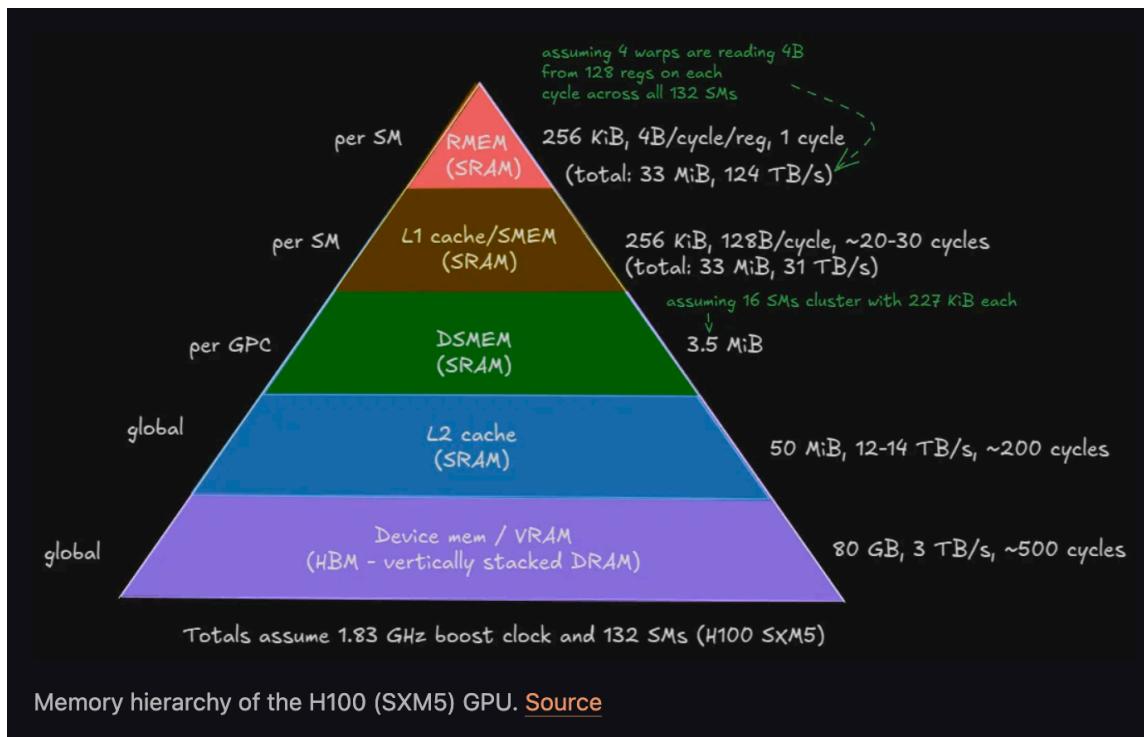
- **포트 활용도:** 집게 대역폭이 활용도가 낮아 보여도 개별 메모리 포트가 포화될 수 있습니다

위의 우리의 특정 경우에서, 커널 명령이 메모리 계층을 통해 어떻게 흐르는지 볼 수 있습니다(우리 하드웨어에서 FP64 행렬 곱셈의 경우): 전역 로드 명령이 L1/TEX 캐시에 요청을 생성하고, 이것이 히트하거나 미스하여 L2에 추가 요청을 생성하고, 궁극적으로 미스 시 디바이스 메모리(HBM)에 접근합니다. 유닛 내부의 색상이 있는 사각형은 포트 활용도를 보여줍니다. 개별 링크가 피크 아래에서 작동하더라도 공유 데이터 포트가 포화될 수 있습니다.

### 메모리 계층 접근 최적화

최적의 성능을 위해, 더 느린 메모리 계층(HBM)으로의 트래픽을 최소화하면서 더 빠른 계층(공유 메모리, 레지스터)의 활용을 최대화하는 것을 목표로 하세요.

이제 이 차트를 가능하게 하는 기본 메모리 계층을 이해해 보겠습니다. 현대 GPU는 속도, 용량, 비용의 균형을 맞추는 계층 구조로 메모리를 구성합니다. 이것은 기본 물리학과 회로 제약에 의해 결정된 설계입니다.



이 계층의 맨 아래에는 HBM(High Bandwidth Memory)이 있습니다: GPU의 메인 메모리로, 글로벌 메모리 또는 디바이스 메모리라고도 합니다. H100은 이론적 대역폭 3.35 TB/s의 HBM3를 특징으로 합니다. HBM은 메모리 계층에서 가장 크지만 가장 느린 계층입니다.

컴퓨팅 유닛을 향해 계층 위로 이동하면, 점진적으로 더 빠르지만 더 작은 메모리 계층을 찾습니다:

- **L2 캐시:** GPU 전체에서 공유되는 대형 SRAM 기반 캐시로, 일반적으로 수십 메가바이트입니다. H100에서는 ~13 TB/s의 대역폭으로 50 MB입니다.

- **L1 캐시와 공유 메모리(SMEM)**: 각 Streaming Multiprocessor(SM)는 자체 L1 캐시와 프로그래머 관리 공유 메모리를 가지며, 동일한 물리적 SRAM 스토리지를 공유합니다. H100에서 이 결합된 공간은 SM당 256 KB이며 SM당 ~31 TB/s의 대역폭을 가집니다.
- **레지스터 파일(RMEM)**: 계층의 최상단에서, 레지스터는 컴퓨팅 유닛 바로 옆에 위치한 가장 빠른 스토리지입니다. 레지스터는 개별 스레드에 비공개이며 SM당 ~100s TB/s로 측정되는 대역폭을 제공합니다.

이 계층이 존재하는 이유는 SRAM(캐시와 레지스터에 사용)은 빠르지만 물리적으로 크고 비싸며, DRAM(HBM에 사용)은 밀집되고 저렴하지만 느리기 때문입니다. 결과: 빠른 메모리는 컴퓨팅에 가까운 곳에 소량으로 제공되며, 더 멀리 있는 점진적으로 더 큰 느린 메모리 풀로 뒷받침됩니다.

**왜 이것이 중요한가:** 이 계층을 이해하는 것은 커널 최적화에 필수적입니다. 핵심 통찰은 **메모리 바운드 연산은 얼마나 빨리 계산할 수 있는지가 아니라 얼마나 빨리 데이터를 이동할 수 있는지에 의해 제한된다**는 것입니다. [Horace He가 "Making Deep Learning Go Brrrr From First Principles"](#)에서 설명듯이, "메모리에서 로드" → "자신으로 두 번 곱하기" → "메모리에 쓰기"는 "메모리에서 로드" → "자신으로 한 번 곱하기" → "메모리에 쓰기"와 본질적으로 같은 시간이 걸립니다: 계산은 메모리 접근에 비해 "무료"입니다.

이것이 **연산자 융합**이 그렇게 강력한 이유입니다: 여러 연산을 단일 커널로 결합함으로써, 연산 사이에 느린 HBM에 다시 쓰는 대신 중간 결과를 빠른 SRAM에 유지할 수 있습니다. Flash Attention은 이 원칙이 실제로 작동하는 완벽한 예입니다.

### ⚡ **Flash Attention: 메모리 계층 최적화의 사례 연구**

표준 어텐션 구현은 HBM에 전체 어텐션 행렬을 구체화하기 때문에 메모리 바운드입니다:

1.  $Q @ K^T$  계산 →  $N \times N$  어텐션 점수를 HBM에 쓰기
2. 소프트맥스 적용 → HBM에서 읽기, 계산, HBM에 다시 쓰기
3.  $V$ 로 곱하기 → HBM에서 어텐션 점수를 다시 읽기

Flash Attention은 이러한 연산을 융합하고 중간 결과를 SRAM에 유지하여 2-4배 속도 향상을 달성합니다:

- 전체 어텐션 행렬을 계산하는 대신, SRAM에 맞는 타일로 어텐션을 처리
- 중간 어텐션 점수가 빠른 온칩 메모리를 떠나지 않음
- 최종 출력만 HBM에 다시 쓰기

결과: Flash Attention은 HBM 접근을  $O(N^2)$ 에서  $O(N)$ 으로 줄여, 메모리 바운드 연산을 GPU의 컴퓨팅 능력을 더 잘 활용하는 것으로 변환합니다. 이것이 효율적인 커널 설계의 본질입니다. **느린 메모리 이동을 최소화하고, 빠른 계산을 최대화하세요.**

### 예시: 실제로 HBM3 대역폭 검증

이제 메모리 계층을 이해했으니, 이론을 실제로 적용하고 H100 GPU의 실제 대역폭을 검증해 봅시다! 여기서 벤치마킹 도구가 필수적입니다.

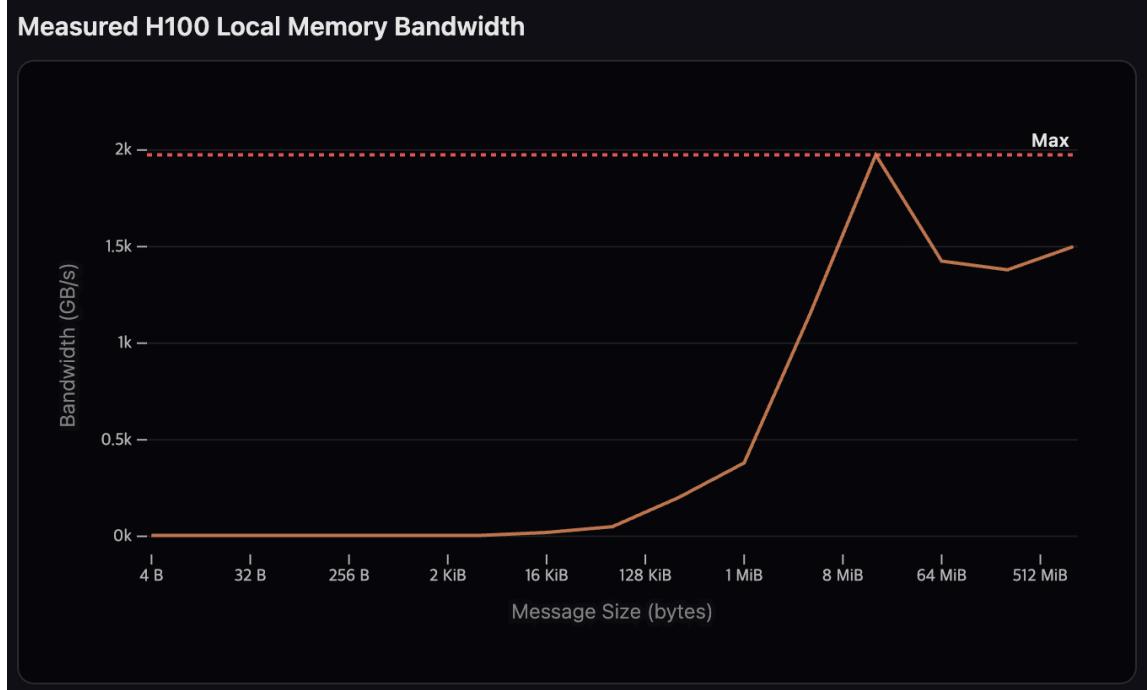
NVBandwidth는 GPU 시스템 전반의 대역폭과 지연 시간을 측정하기 위해 특별히 설계된 NVIDIA의 오픈소스 벤치마킹 도구입니다. 복사 엔진과 커널 기반 방법 모두를 사용하여 다양한 메모리 복사 패턴(호스트에서 디바이스로, 디바이스에서 호스트로, 디바이스에서 디바이스로 연산)의 데이터 전송 속도를 평가합니다. 이 도구는 GPU 간 통신(예: [NVLink](#)와 [PCIe](#), 두 가지 유형의 커넥터)을 평가하고 멀티 GPU 환경에서 시스템 성능을 검증하는 데 특히 유용합니다.

[NVIDIA의 GitHub 저장소에서 NVBandwidth](#)를 설치할 수 있습니다. 이 도구는 다른 디바이스 간에 데이터가 얼마나 효율적으로 전송되는지 보여주는 상세한 대역폭 행렬을 출력하여, 성능 병목을 진단하거나 건강한 GPU 상호 연결을 확인하는데 이상적입니다.

device\_local\_copy 테스트를 사용하여 H100의 로컬 메모리 대역폭을 측정해 봅시다. 이것은 다른 메시지 크기에 걸쳐 GPU에 로컬인 디바이스 버퍼 간의 cuMemcpyAsync 대역폭을 측정합니다.

cuMemcpyAsync는 두 메모리 포인터 간에 데이터를 비동기적으로 복사하는 CUDA 드라이버 API 함수로, 전송 유형 (호스트에서 호스트로, 호스트에서 디바이스로, 디바이스에서 디바이스로, 또는 디바이스에서 호스트로)을 추론합니다.

```
$./nvbandwidth -t device_local_copy -b 2048
memcpy local GPU(column) bandwidth (GB/s)
 0 1 2 3 4 5 6 7
0 1519.07 1518.93 1519.07 1519.60 1519.13 1518.86 1519.13 1519.33
```



결과는 메모리 시스템의 중요한 특성을 드러냅니다: 작은 메시지 크기( $< 1 \text{ MB}$ )의 경우, 대역폭 바운드가 아니라 자연 시간 바운드입니다. 메모리 전송을 시작하는 오버헤드가 성능을 지배하여 피크 대역폭에 도달하는 것을 방지합니다. 그러나 큰 메시지 크기( $\geq 1 \text{ MB}$ )의 경우, 읽기와 쓰기 연산 모두에 대해 **~1,500 GB/s**의 지속적인 대역폭을 달성합니다.

HBM 대역폭은 동시에 발생하는 읽기와 쓰기 모두를 고려하므로, 이를 합산하여 총 3 TB/s 양방향 대역폭(1,519 읽기 + 1,519 쓰기)을 얻으며, 이것은 H100의 이론적 3.35 TB/s HBM3 사양을 밀접하게 검증합니다.

## 루프라인 모델

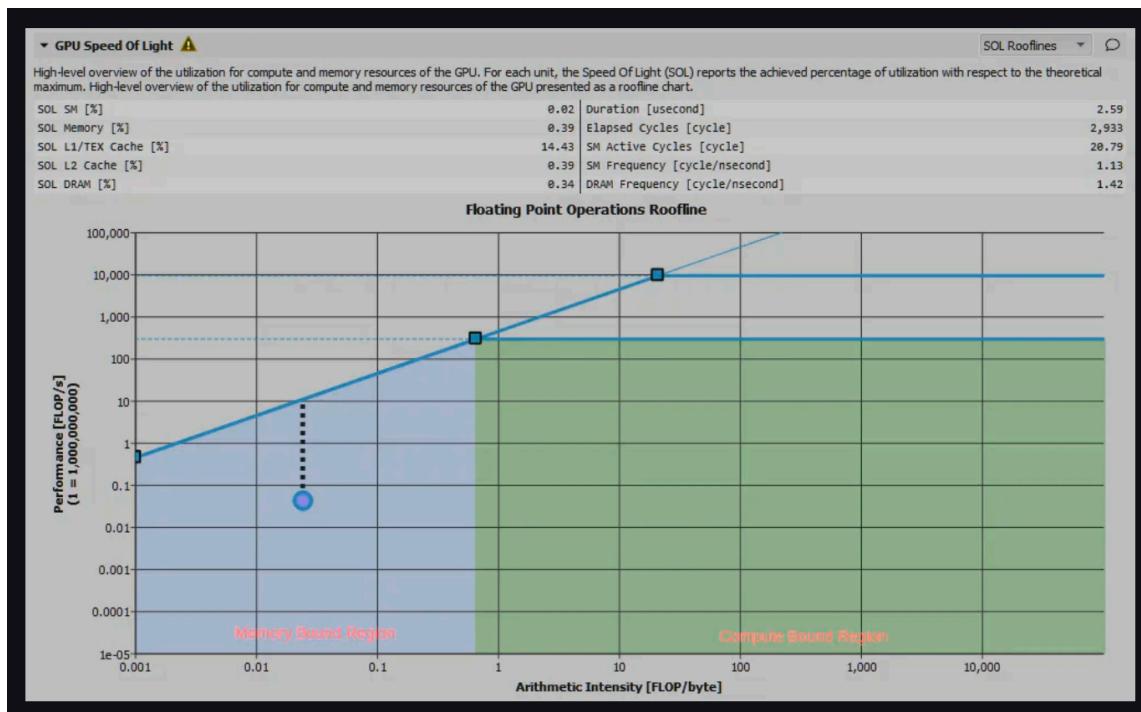
커널이 컴퓨팅 바운드인지 메모리 바운드인지 이해하면 어떤 최적화가 도움이 될지 결정합니다.

두 가지 시나리오가 있습니다:

- **메모리 바운드**(데이터 이동에 대부분의 시간을 소비)인 경우, 컴퓨팅 처리량을 늘려도 도움이 되지 않습니다: 연산자 융합과 같은 기술을 통해 메모리 트래픽을 줄여야 합니다.
- **컴퓨팅 바운드**(FLOPs에 대부분의 시간을 소비)인 경우, 메모리 접근 패턴을 최적화해도 도움이 되지 않습니다: 더 많은 컴퓨팅 파워나 더 나은 알고리즘이 필요합니다.

루프라인 모델은 이러한 성능 특성을 이해하고 최적화 기회를 식별하기 위한 시각적 프레임워크를 제공합니다.

실제 커널 분석에 적용해 봅시다. 이전에 언급한 NSight Compute 프로파일링 도구에서 사용할 수 있습니다("roofline analysis view" 아래). 다음은 얻은 결과입니다:



이 차트를 어떻게 읽는지 살펴봅시다. 두 개의 축이 있습니다:

- **수직 축 (FLOP/s):** 초당 달성된 부동소수점 연산을 보여주며, 넓은 범위의 값을 수용하기 위해 로그 스케일을 사용합니다.
- **수평 축 (산술 강도):** 작업(FLOPs)과 메모리 트래픽(바이트)의 비율을 나타내며, 바이트당 FLOPs로 측정됩니다. 이것도 로그 스케일을 사용합니다.

루프라인 자체는 두 개의 경계로 구성됩니다:

- **메모리 대역폭 경계 (기울어진 선):** GPU의 메모리 전송 속도(HBM 대역폭)에 의해 결정됩니다. 이 선을 따른 성능은 데이터가 얼마나 빨리 이동할 수 있는지에 의해 제한됩니다.
- **피크 성능 경계 (평평한 선):** GPU의 최대 컴퓨팅 처리량에 의해 결정됩니다. 이 선을 따른 성능은 계산이 얼마나 빨리 실행될 수 있는지에 의해 제한됩니다.

이러한 경계가 만나는 능선 지점은 메모리 바운드와 컴퓨팅 바운드 영역 간의 전환을 나타냅니다.

차트의 두 분할된 영역을 보면 성능을 해석할 수 있습니다:

- **메모리 바운드 (기울어진 경계 아래):** 이 영역의 커널은 메모리 대역폭에 의해 제한됩니다. GPU가 데이터를 기다리고 있으며, 컴퓨팅 파워를 늘려도 도움이 되지 않습니다. 최적화는 연산자 융합, 더 나은 메모리 접근 패턴, 또는 산술 강도 증가와 같은 기술을 통해 메모리 트래픽을 줄이는 데 집중해야 합니다.
- **컴퓨팅 바운드 (평평한 경계 아래):** 이 영역의 커널은 컴퓨팅 처리량에 의해 제한됩니다. GPU에는 충분한 데이터가 있지만 충분히 빠르게 처리할 수 없습니다. 최적화는 알고리즘 개선이나 Tensor 코어와 같은 특수 하드웨어 활용에 집중해야 합니다.

달성된 값(플롯된 점)은 커널이 현재 어디에 있는지 보여줍니다. 이 점에서 루프라인 경계까지의 거리는 최적화 여유 공간을 나타냅니다. 경계에 가까울수록 커널의 성능이 더 최적입니다.

우리 예시에서, 커널은 메모리 바운드 영역에 위치하여, 메모리 트래픽을 최적화하여 개선할 여지가 여전히 있음을 나타냅니다!

CUDA 코어, Tensor 코어, 메모리 계층, 저수준 최적화 기술에 대한 자세한 설명을 포함한 GPU 내부에 대한 더 깊은 탐구는 [Ultrascale Playbook](#)을 확인하세요! 이제 GPU 내부에서 무슨 일이 일어나는지 이해했으니, 확대하여 GPU가 나머지 세계와 어떻게 통신하는지 탐색해 봅시다.

## GPU 외부: GPU가 세상과 대화하는 방법

이제 GPU가 내부 메모리 계층을 사용하여 어떻게 계산을 수행하는지 이해했으니, 중요한 현실을 다루어야 합니다: GPU는 고립되어 작동하지 않습니다. 계산이 일어나기 전에 데이터가 GPU의 메모리에 로드되어야 합니다. CPU는 커널을 스케줄링하고 작업을 조정해야 합니다. 그리고 분산 학습에서 GPU는 서로 활성화, 그래디언트, 모델 가중치를 지속적으로 교환해야 합니다.

여기서 외부 통신 인프라가 중요해집니다. GPU의 컴퓨팅 유닛이 아무리 강력해도, CPU에서든, 스토리지에서든, 다른 GPU에서든 데이터가 충분히 빠르게 도달할 수 없다면, 비싼 하드웨어가 유휴 상태로 있게 됩니다. 이러한 통신 경로와 대역폭 특성을 이해하는 것은 하드웨어 활용을 최대화하고 병목을 최소화하는 데 필수적입니다.

이 섹션에서는 GPU를 외부 세계에 연결하는 네 가지 중요한 통신 링크를 살펴보겠습니다:

- **GPU-CPU:** CPU가 어떻게 작업을 스케줄링하고 GPU로 데이터를 전송하는지
- **GPU-GPU 노드 내:** 동일한 머신의 GPU가 어떻게 통신하는지
- **GPU-GPU 노드 간:** 다른 머신의 GPU가 네트워크를 통해 어떻게 통신하는지
- **GPU-스토리지:** 데이터가 스토리지에서 GPU 메모리로 어떻게 흐르는지

이러한 각 링크는 다른 대역폭과 지연 시간 특성을 가지며, 이를 이해하면 학습 파이프라인이 어디서 병목이 될 수 있는지 식별하는 데 도움이 됩니다. 이를 더 쉽게 이해할 수 있도록, 가장 중요한 구성 요소와 통신 링크를 강조하는 단순화된 다이어그램을 만들었습니다:



이것이 압도적으로 보인다면, 걱정하지 마세요. 이러한 각 연결을 자세히 살펴보고 각 링크의 성능 특성을 이해하기 위해 실제 대역폭을 측정할 것입니다.

## GPU에서 CPU로

**TL;DR:** CPU는 PCIe 연결을 통해 GPU 작업을 조율하며, 우리 p5 인스턴스에서 CPU에서 GPU로의 전송은 ~14.2 GB/s(PCle Gen4 x8)에서 병목이 됩니다. CPU-GPU 지연 시간은 ~1.4 마이크로초이며, 이것은 많은 작은 커널이 있는 워크로드에 문제가 되는 커널 시작 오버헤드를 추가합니다. CUDA Graphs는 연산을 일괄 처리하여 이 오버헤드를 줄일 수 있습니다. NUMA 친화성은 멀티 소켓 시스템에서 중요합니다; 잘못된 CPU 소켓에서 GPU 프로세스를 실행하면 상당한 지연 시간이 추가됩니다. Grace Hopper와 같은 현대 아키텍처는 NVLink-C2C(128 GB/s 대비 900 GB/s)로 PCIe 병목을 제거합니다.

CPU는 GPU 계산의 조율자입니다. 커널을 시작하고, 메모리 할당을 관리하고, 데이터 전송을 조정하는 책임이 있습니다. 그러나 CPU가 실제로 GPU와 얼마나 빠르게 통신할 수 있을까요? 이것은 그들 사이의 PCIe(Peripheral Component Interconnect Express) 연결에 의해 결정됩니다.

이 링크를 이해하는 것이 중요한 이유는 다음에 영향을 미치기 때문입니다:

- 커널 시작 지연 시간: CPU가 GPU에서 작업을 얼마나 빨리 스케줄링할 수 있는지
- 데이터 전송 속도: CPU와 GPU 메모리 간에 데이터를 얼마나 빠르게 이동할 수 있는지
- 동기화 오버헤드: CPU-GPU 조정 지점의 비용

현대 GPU 서버에서 CPU-GPU 연결은 상당히 발전했습니다. 초기 시스템은 직접 PCIe 연결을 사용했지만, DGX H100과 같은 현대 고성능 시스템은 여러 GPU를 효율적으로 관리하기 위해 PCIe 스위치가 있는 더 정교한 토플로지를 사용합니다. 그리고 최신 [GB200 아키텍처](#)에서 NVIDIA는 CPU와 GPU를 동일한 인쇄 회로 기판에 배치하여 외부 스위치의 필요성을 완전히 제거함으로써 이를 더욱 발전시켰습니다.

`lstopo` 를 사용하여 p5 인스턴스의 물리적 토플로지를 검사한 다음 이 중요한 링크의 실제 성능을 측정하여 잠재적 병목을 식별해 봅시다.

```
$ lstopo -v
...
HostBridge L#1 (buses=0000:[44-54])
 PCIBridge L#2 (busid=0000:44:00.0 id=1d0f:0200 class=0604(PCIBridge) link=15.75GB/s
 buses=0000:[45-54] PCISlot=64)
 PCIBridge L#3 (busid=0000:45:00.0 id=1d0f:0200 class=0604(PCIBridge) link=15.75GB/s
 buses=0000:[46-54] PCISlot=1-1)
 ...
 PCIBridge L#12 (busid=0000:46:01.4 id=1d0f:0200 class=0604(PCIBridge) link=63.02GB/s
 buses=0000:[53-53])
 PCI L#11 (busid=0000:53:00.0 id=10de:2330 class=0302(3D) link=63.02GB/s
 PCISlot=86-1)
 Co-Processor(CUDA) L#8 (Backend=CUDA GPUVendor="NVIDIA Corporation"
GPUModel="NVIDIA H100 80GB HBM3" CUDAGlobalMemorySize=83295872 CUDAL2CacheSize=51200
CUDAMultiProcessors=132 CUDAcoresPerMP=128 CUADSharedMemorySizePerMP=48) "cuda0"
 GPU(NVML) L#9 (Backend=NVML GPUVendor="NVIDIA Corporation" GPUModel="NVIDIA
H100 80GB HBM3" NVIDIAASerial=1654922006536 NVIDIAUUID=GPU-ba136838-6443-7991-9143-1bf4e48b2994)
"nvml0"
 ...
 ...

```

`lstopo` 출력에서, 시스템에서 두 가지 핵심 PCIe 대역폭 값을 볼 수 있습니다:

- **15.75GB/s**: PCIe Gen4 x8 링크(CPU에서 PCIe 스위치로)에 해당
- **63.02GB/s**: PCIe Gen5 x16 링크(PCIE 스위치에서 GPU로)에 해당

전체 토플로지를 더 잘 이해하기 위해, 다음을 사용하여 시각화할 수 있습니다:

```
$ lstopo --whole-system lstopo-diagram.png
```



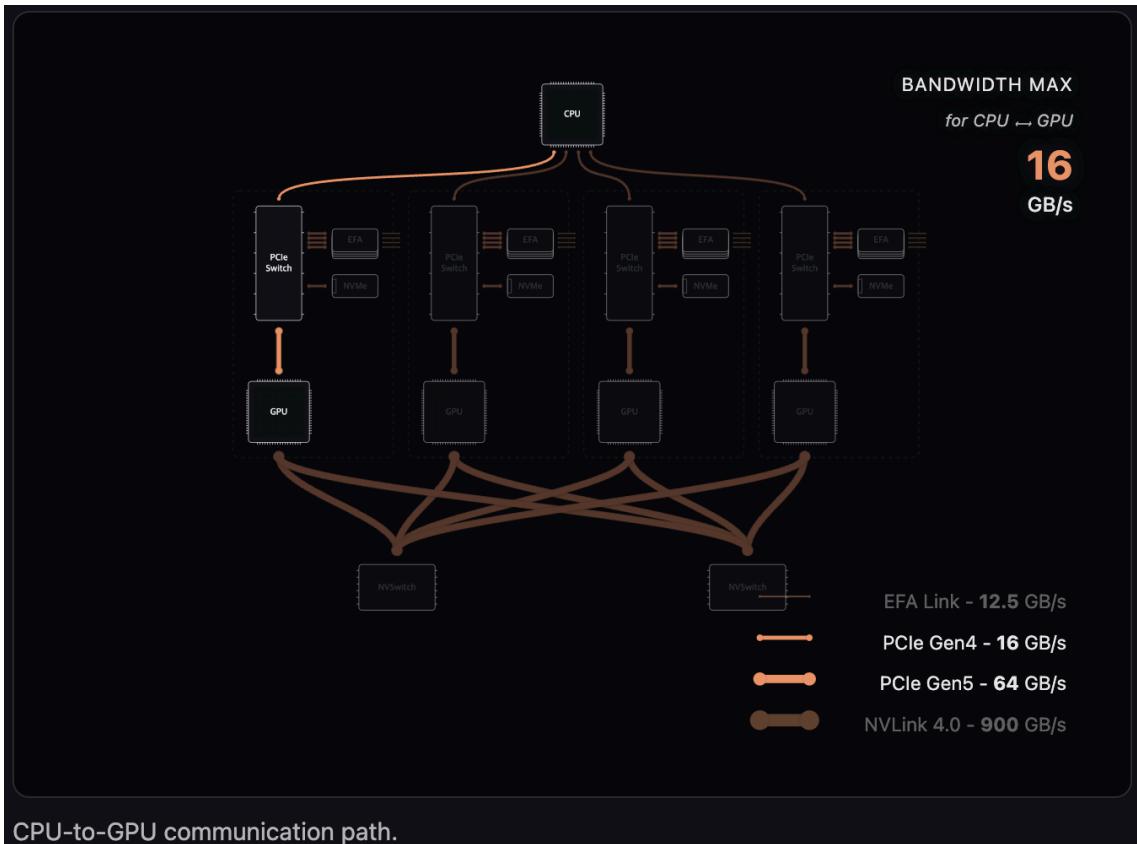
이 다이어그램은 시스템의 계층적 구조를 보여줍니다:

- 두 개의 NUMA(Non-Uniform Memory Access) 노드를 포함합니다(NUMA는 CPU 소켓당 메모리 영역입니다)
- 각 CPU 소켓은 PCIe Gen4 x8 링크(15.75GB/s)를 통해 네 개의 PCIe 스위치에 연결됩니다
- 각 PCIe 스위치는 PCIe Gen5 x16 링크(63.02GB/s)를 통해 하나의 H100 GPU에 연결됩니다
- ... (NVSwitch, EFA 네트워크 카드, NVMe 드라이브와 같은 다른 구성 요소는 다음 섹션에서 탐색할 것입니다.)

PCIe 사양은 세대 간에 다르며, 각각 레인당 전송 속도가 두 배가 됩니다. 전송 속도는 원시 신호 속도를 나타내는 GT/s(초당 기가트랜스퍼)로 측정되는 반면, 처리량은 인코딩 오버헤드를 고려하고 실제 사용 가능한 대역폭을 나타내는 GB/s(초당 기가바이트)로 측정됩니다:

| PCIe 버전 | 전송 속도 (레인당) | 처리량 (GB/s) ×1 | ×2 | ×4 |
|---------|-------------|---------------|----|----|
| 1.0     | 2.5 GT/s    | 0.25          |    |    |
| 2.0     | 5.0 GT/s    | 0.5           |    |    |
| 3.0     | 8.0 GT/s    | 0.985         |    |    |
| 4.0     | 16.0 GT/s   | 1.969         |    |    |
| 5.0     | 32.0 GT/s   | 3.938         |    |    |
| 6.0     | 64.0 GT/s   | 7.563         |    |    |
| 7.0     | 128.0 GT/s  | 15.125        |    |    |

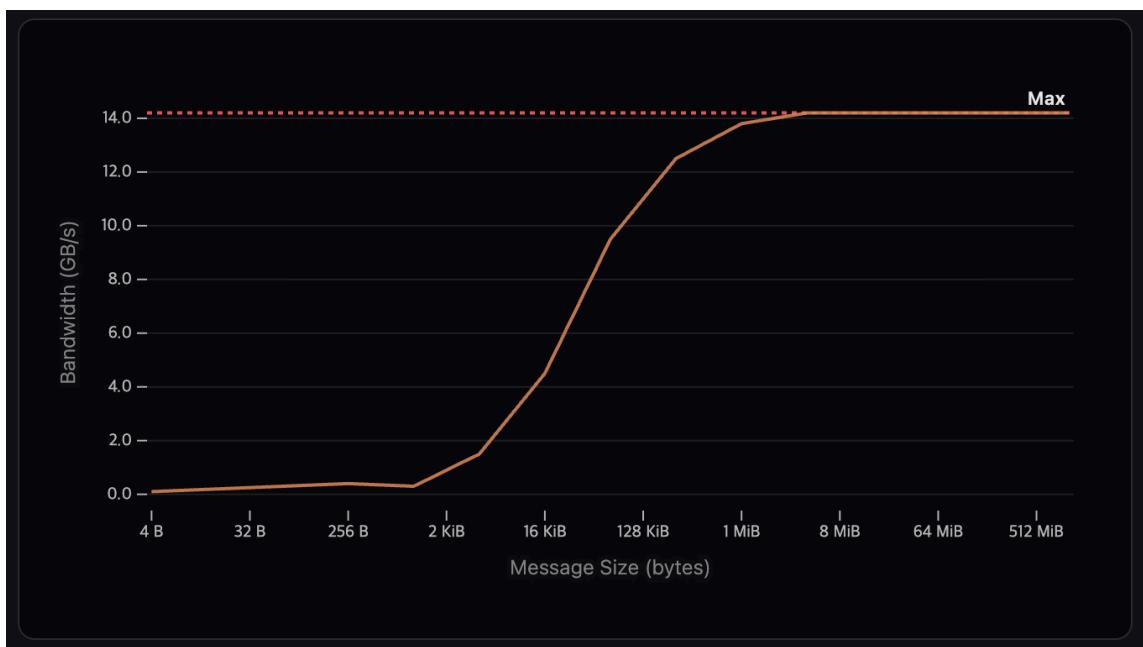
이론적 PCIe 대역폭. 출처: [https://en.wikipedia.org/wiki/PCI\\_Express](https://en.wikipedia.org/wiki/PCI_Express)



토플로지 다이어그램과 PCIe 대역폭 표에서, CPU에서 GPU로의 경로가 두 개의 PCIe 흡을 통과함을 볼 수 있습니다: 먼저 CPU에서 PCIe 스위치로 PCIe Gen4 x8(15.754 GB/s)을 통해, 그런 다음 PCIe 스위치에서 GPU로 PCIe Gen5 x16(63.015 GB/s)을 통해. 이것은 CPU-GPU 통신의 병목이 15.754 GB/s의 첫 번째 흡임을 의미합니다. 또 다른 유ти리티인 `nvbandwidth`로 이를 검증해 봅시다!

`host_to_device_memcpy_ce` 명령은 GPU의 복사 엔진을 사용하여 호스트(CPU) 메모리에서 디바이스(GPU) 메모리로의 `cuMemcpyAsync` 대역폭을 측정합니다.

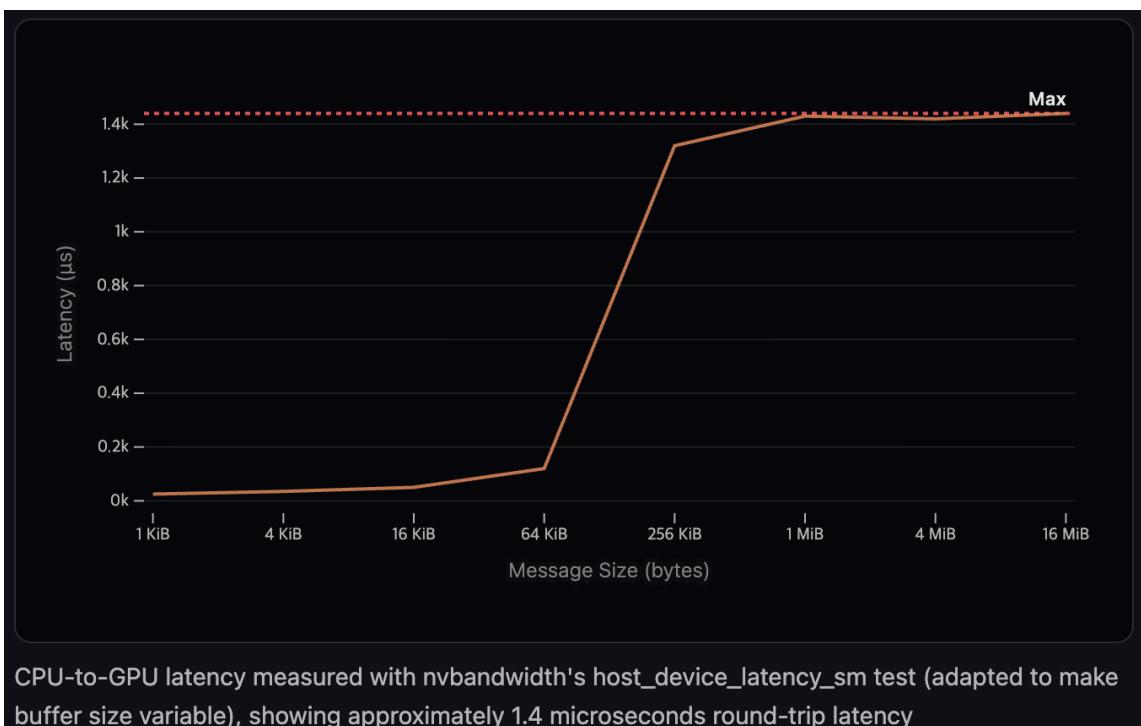
```
./nvbandwidth -t host_to_device_memcpy_ce -b <message_size> -i 5
```



결과는 실제로 작은 메시지 크기의 경우 지연 시간 바운드이지만, 큰 메시지 크기의 경우 **~14.2 GB/s**를 달성하며, 이것은 PCIe Gen4 x8의 이론적 15.754 GB/s 대역폭의 약 90%입니다. 이것은 CPU-GPU 통신에서 CPU에서 PCIe 스위치로의 링크가 실제로 병목임을 확인합니다.

대역폭 외에도, 지연 시간은 커널을 얼마나 빨리 스케줄링할 수 있는지 결정하므로 CPU-GPU 통신에서 똑같이 중요합니다. 이를 측정하기 위해, 왕복 지연 시간을 측정하는 포인터 추적 커널을 사용하는 `nvbandwidth` 의 `host_device_latency_sm` 테스트를 사용합니다. `host_device_latency_sm` 테스트는 호스트(CPU)에 버퍼를 할당하고 포인터 추적 커널을 사용하여 GPU에서 접근하여 왕복 지연 시간을 측정합니다. 이것은 CPU-GPU 통신의 실제 지연 시간을 시뮬레이션합니다.

```
./nvbandwidth -t host_device_latency_sm -i 5
```



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

결과는 지연 시간이 약 **1.4 마이크로초**임을 보여줍니다. 이것은 ML 워크로드에서 종종 관찰하는 몇 마이크로초의 커널 시작 오버헤드를 설명합니다. 많은 작은 커널을 시작하는 워크로드의 경우, 추가된 지연 시간이 병목이 될 수 있습니다; 그렇지 않으면 오버헤드는 중첩 실행에 의해 숨겨집니다.

예를 들어 작은 모델이나 작은 배치의 경우 커널 시작 때문에 GPU에서 추론이 포화되는 것을 볼 수 있습니다.

FlashFormer는 전체 레이어를 융합하여 속도 향상을 얻음으로써 이를 해결합니다(Nrusimha et al., 2025).

### 🚀 시작 오버헤드를 줄이기 위한 CUDA Graphs

CUDA Graphs는 일련의 연산을 캡처하고 단일 단위로 재생하여 각 커널 시작에 대한 마이크로초의 CPU-GPU 왕복 지연 시간을 제거함으로써 커널 시작 오버헤드를 상당히 줄일 수 있습니다. 이것은 많은 작은 커널이나 빈번한 CPU-GPU 동기화가 있는 워크로드에 특히 유익합니다. 시작 오버헤드를 이해하고 최적화하는 방법에 대한 자세한 내용은 "[Understanding the Visualization of Overhead and Latency in NVIDIA Nsight Systems](#)"를 참조하세요.

### ⚠️ MoE 모델과 CPU-GPU 동기화 오버헤드

일부 Mixture-of-Experts(MoE) 모델 구현은 선택된 전문가에 대한 적절한 커널을 스케줄링하기 위해 각 반복에서 CPU-GPU 동기화를 필요로 합니다. 이것은 특히 CPU-GPU 연결이 느릴 때 처리량에 상당히 영향을 미칠 수 있는 커널 시작 오버헤드를 도입합니다. 예를 들어, MakoGenerate의 DeepSeek MOE 커널 최적화에서, 첨조 구현은 순방향 패스당 67개의 CPU-GPU 동기화 지점과 함께 1,043개의 커널을 디스패치했습니다. 전문가 라우팅 메커니즘을 재구조화하여, 533개의 커널 시작과 단 3개의 동기화 지점으로 줄였으며, 동기화 오버헤드 97% 감소와 종단 간 지연 시간 44% 감소를 달성했습니다. 모든 MoE 구현이 CPU-GPU 동기화를 필요로 하는 것은 아니지만(현대 구현은 종종 라우팅을 전적으로 GPU에서 유지), 필요로 하는 구현의 경우 효율적인 CPU-GPU 통신이 성능에 중요해집니다.

## Grace Hopper 슈퍼칩: CPU-GPU 통신에 대한 다른 접근 방식

NVIDIA의 Grace Hopper 슈퍼칩은 전통적인 x86+Hopper 시스템과 비교하여 CPU-GPU 통신에 근본적으로 다른 접근 방식을 취합니다. 주요 개선 사항:

- **1:1 GPU 대 CPU 비율**(x86+Hopper의 4:1과 비교), GPU당 3.5배 더 높은 CPU 메모리 대역폭 제공
- **NVLink-C2C가 PCIe Gen5 레인을 대체**, 128 GB/s 대비 900 GB/s 제공(7배 더 높은 GPU-CPU 링크 대역폭)
- **NVLink Switch System**이 PCIe Gen4를 통해 연결된 InfiniBand NDR400 NIC보다 9배 더 높은 GPU-GPU 링크 대역폭 제공

자세한 내용은 [NVIDIA Grace Hopper Superchip Architecture Whitepaper](#)(11페이지)를 참조하세요.

### NUMA 친화성: 멀티 소켓 성능에 중요

AMD EPYC 7R13 노드(2소켓, 각 48코어)와 같은 멀티 소켓 시스템에서, **NUMA 친화성은 GPU 성능에 중요합니다**. 이 것은 대상 디바이스(GPU와 같은)와 동일한 소켓을 공유하는 CPU 코어에서 프로세스를 실행하는 것을 말합니다. GPU 프로세스가 GPU가 연결된 것과 다른 NUMA 노드의 CPU에서 실행될 때, 연산이 CPU 인터커넥트(AMD Infinity Fabric)를 통과해야 하며, 상당한 지연 시간과 대역폭 제약을 추가합니다.

먼저, 성능 영향을 이해하기 위해 NUMA 토폴로지와 노드 거리를 검사해 봅시다:

```
$ numactl --hardware
node distances:
node 0 1
 0: 10 32
 1: 32 10
```

거리 값은 동일한 NUMA 노드의 메모리에 접근하는 것(거리 10)이 다른 NUMA 노드로 교차하는 것(거리 32)보다 훨씬 빠릅니다. 메모리 접근 지연 시간의 이 3.2배 차이는 프로세스가 잘못된 NUMA 노드에 고정되었을 때 GPU 성능에 상당히 영향을 미칠 수 있습니다.

NUMA 관련 성능 문제를 진단하고 해결하는 자세한 단계는 상호 연결 성능 문제 해결 섹션을 참조하세요.

## GPU에서 GPU로 노드 내

분산 학습에서 GPU는 반복당 종종 기가바이트의 데이터인 그래디언트, 가중치, 활성화를 자주 교환해야 합니다. 이 거대한 양의 데이터는 통신의 신중한 처리를 필요로 합니다. H100의 내부 HBM은 약 3 TB/s로 읽을 수 있지만, 실수로 잘못된 플래그를 사용하면 GPU 간 통신 대역폭을 완전히 망칠 수 있습니다!

동일한 노드 내에서 GPU 간에 통신할 수 있는 모든 방법(그리고 설정해야 하거나 설정하지 말아야 할 모든 플래그)을 검사하여 그 이유를 살펴봅시다 😊

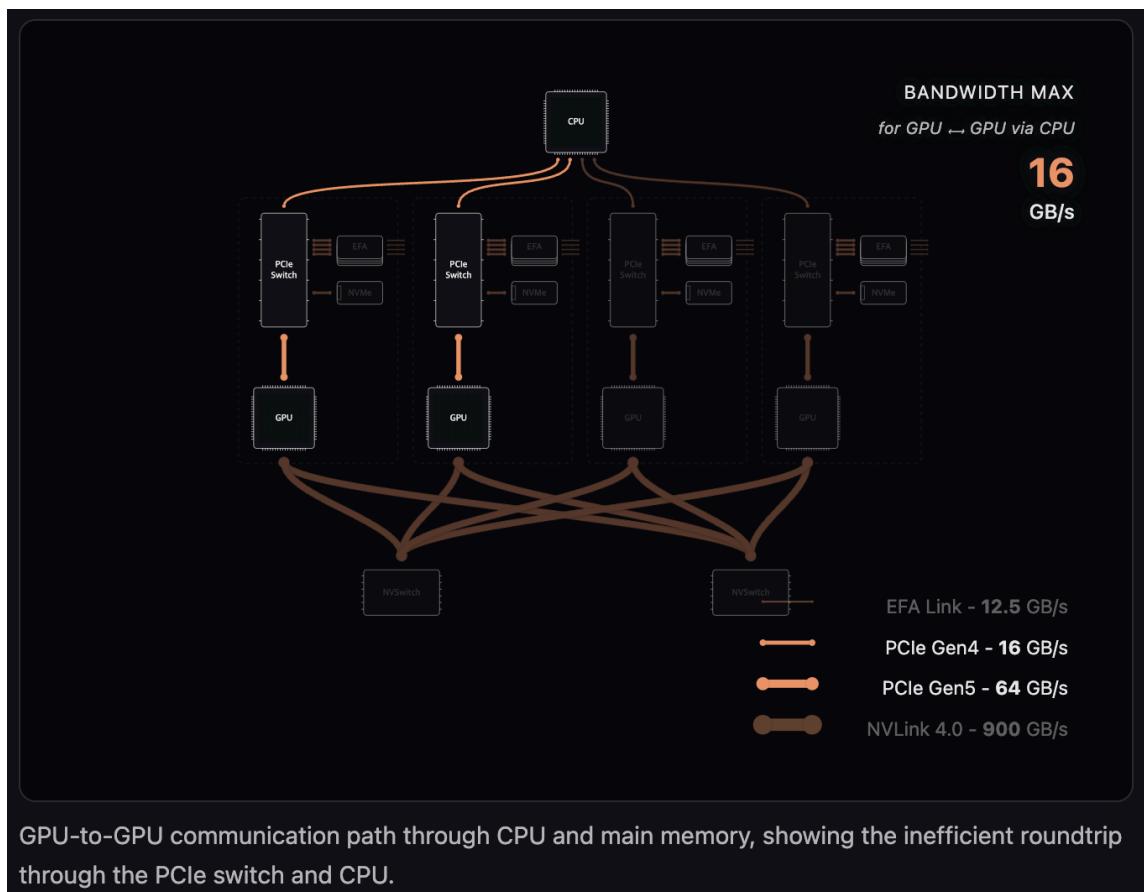
**TL;DR:** 노드 내의 GPU는 세 가지 방법으로 통신할 수 있습니다: CPU를 통해(가장 느림, 3 GB/s, PCIe에 의해 병목), EFA NIC를 통한 GPUDirect RDMA(38 GB/s), 또는 NVLink를 통한 GPUDirect RDMA(양방향 ~786 GB/s). NVLink는 9-112배 더 빠르며 CPU/PCIe를 완전히 우회합니다. NCCL은 사용 가능할 때 자동으로 NVLink를 우선시합니다.

NVLink SHARP(NVLS)는 하드웨어 가속 집합체를 제공하여 allreduce 성능을 1.3배 향상시켜 480 GB/s에 도달합니다. 그러나 alltoall 연산(340 GB/s)은 NVLS 가속의 이점을 받지 못합니다.

## CPU를 통해

순진한 접근 방식은 호스트 메모리(SHM)를 사용합니다: 데이터가 GPU1에서 PCIe 스위치를 통해 CPU로, 호스트 메모리로, CPU를 통해 다시, 다시 PCIe 스위치를 통해, 마침내 GPU2로 이동합니다. 이것은 NCCL의 `NCCL_P2P_DISABLE=1` 과 `FI_PROVIDER=tcp` 환경 변수를 사용하여 달성할 수 있습니다(권장하지 않음). 이 모드가 활성화되면, `NCCL_DEBUG=INFO`를 설정하여 다음과 같은 메시지를 볼 수 있습니다:

```
NCCL INFO Channel 00 : 1[1] -> 0[0] via SHM/direct/direct
```



이 우회 경로는 여러 메모리 복사를 포함하고 PCIe와 CPU 메모리 버스 모두를 포화시켜 혼잡을 유발합니다. 4개의 H100이 동일한 CPU 메모리 버스를 공유하는 우리 토플로지에서, 여러 GPU가 동시에 통신을 시도할 때 동일한 제한된 CPU 메모리 대역폭을 놓고 경쟁하므로 이 혼잡은 더욱 문제가 됩니다... 😢

이 CPU 매개 접근 방식으로는 CPU와 PCIe 스위치 사이의 ~16 GB/s의 PCIe Gen4 x8 링크에 근본적으로 병목이 됩니다. 다행히도, CPU를 포함하지 않고 GPU가 통신할 수 있는 더 나은 방법이 있습니다: [NVIDIA GPUDirect..](#)

## Libfabric EFA를 통해

GPUDirect RDMA(Remote Direct Memory Access 또는 GDRDMA)는 GPU 메모리에 직접 접근을 허용하여 NVIDIA GPU 간의 직접 통신을 가능하게 하는 기술입니다. 이것은 데이터가 시스템 CPU를 통과할 필요성을 제거하고 시스템 메모리를 통한 버퍼 복사를 피하여, 전통적인 CPU 매개 전송에 비해 최대 10배 더 나은 성능을 제공합니다. GPUDirect RDMA는 PCIe를 통해 작동하여 노드 내(여기서 보듯이)와 RDMA 능력을 가진 NIC(네트워크 인터페이스 카드, 향후 색션에서 볼 것처럼)를 사용하여 노드 간에 빠른 GPU 간 통신을 가능하게 합니다. 자세한 내용은 ([NVIDIA GPUDirect](#))를 참조하세요.

토플로지 다이어그램을 다시 보면, 각 PCIe 스위치에 4개의 EFA(Elastic Fabric Adapter) NIC가 있어, 각 GPU가 4개의 EFA 어댑터에 접근할 수 있음을 볼 수 있습니다. EFA는 클라우드 인스턴스를 위한 AWS의 커스텀 고성능 네트워크 인터페이스로, 낮은 지연 시간, 높은 처리량의 인스턴스 간 통신을 제공하도록 설계되었습니다. p5 인스턴스에서 EFA는 애플리케이션이 사용할 수 있는 libfabric 인터페이스(고성능 계산을 위한 특정 통신 API)를 노출하고, 노드 간 직접 GPU 간 통신을 위한 GPUDirect RDMA를 가능하게 하는 RDMA와 유사한 능력을 제공합니다.

EFA는 대규모 네트워크 경로를 가진 상용 데이터센터 네트워크를 사용하도록 설계된 Scalable Reliable Datagram(SRD)이라는 신뢰할 수 있는 이더넷 기반 전송 프로토콜을 사용합니다. 그 중요성에 대해 여기서 알아보세요.

```
$ lstopo -v
...
각 PCIe 스위치당 이러한 EFA 디바이스 4개를 볼 수 있습니다
PCIBridge L#8 (busid=0000:46:01.0 id=1d0f:0200 class=0604(PCIBridge) link=15.75GB/s buses=0000:[4f-4f] PCIVendor="Amazon.com, Inc.")
PCI L#6 (busid=0000:4f:00.0 id=1d0f:efa1 class=0200(Ethernet) link=15.75GB/s PCISlot=82-1
PCIVendor="Amazon.com, Inc.")

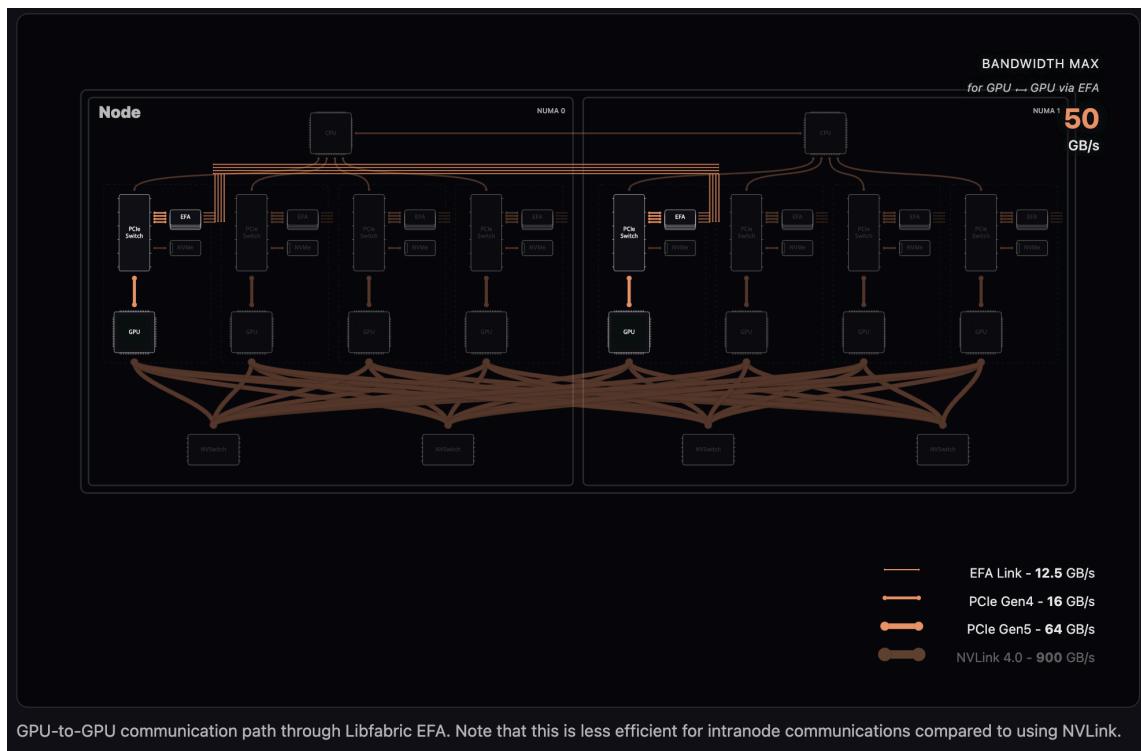
OpenFabrics L#4 (NodeGUID=cd77:f833:0000:1001 SysImageGUID=0000:0000:0000:0000 Port1State=4
Port1LID=0x0 Port1LMC=1 Port1GID0=fe80:0000:0000:14b0:33ff:fef8:77cd) "rdmap79s0"
...
$ fi_info --verbose
fi_link_attr:
 address: EFA-fe80::14b0:33ff:fef8:77cd
 mtu: 8760 # 최대 패킷 크기는 8760 바이트
 speed: 100000000000 # 각 EFA 링크는 100 Gbps의 대역폭 제공
 state: FI_LINK_UP
 network_type: Ethernet
```

각 EFA 링크는 100 Gbps(12.5 GB/s)의 대역폭을 제공합니다. GPU당 4개의 EFA NIC와 노드당 8개의 GPU가 있으므로, 노드당  $100 \times 4 \times 8 = 3200$  Gbps(400GB/s)의 집계 대역폭을 제공합니다.

libfabric과 EFA를 사용하여 이 3200 Gbps 대역폭을 완전히 활용하는 방법에 대한 자세한 탐구는 Lequn Chen의 훌륭한 블로그 시리즈를 참조하세요: "Harnessing 3200 Gbps Network: A Journey with RDMA, EFA, and libfabric".

EFA를 통한 GPUDirect RDMA를 활성화하려면, `FI_PROVIDER=efa` 와 `NCCL_P2P_DISABLE=1` 환경 변수를 설정해야 합니다. 이 모드가 활성화되면, `NCCL_DEBUG=INFO` 를 설정하여 작동 중인지 확인할 수 있으며, 다음과 같은 메시지가 표시됩니다:

NCCL INFO Channel 01/1 : 1[1] -> 0[0] [receive] via NET/Libfabric/0/GDRDMA/Shared



EFA를 통한 GPUDirect RDMA가 CPU 매개 전송에 비해 상당한 개선을 제공하여 GPU당 4개의 EFA 카드로 약 50 GB/s를 달성하지만, 더 잘할 수 있을까요? 여기서 NVLink가 등장합니다.

## NVLink을 통해

NVLink는 서버 내에서 빠른 멀티 GPU 통신을 가능하게 하는 NVIDIA의 고속 직접 GPU 간 상호 연결 기술입니다. H100은 4세대 NVLink(NVLink 4.0)를 사용하며, 각각 50 GB/s 양방향으로 작동하는 18개의 링크를 통해 GPU당 900 GB/s 양방향 대역폭을 제공합니다(NVIDIA H100 Tensor Core GPU Datasheet).

DGX H100 아키텍처에서, 4개의 3세대 NVSwitch가 계층화된 토플로지를 사용하여 8개의 GPU를 연결하며, 각 GPU는 스위치 전체에 걸쳐 5+4+4+5 링크로 연결됩니다. 이 구성은 단 1개의 NVSwitch라는 일정한 흡 수로 모든 GPU 쌍 간에 여러 직접 경로를 보장하여, 총 3.6 TB/s 양방향 NVLink 네트워크 대역폭을 제공합니다.

|     | NVLink 2.0 (Volta) | NVLink 3.0 (Ampere) | NVLink 4.0 (Hopper) | NVLink 5.0 (Blackwell) |
|-----|--------------------|---------------------|---------------------|------------------------|
| 대역폭 | 300 GB/s           | 600 GB/s            | 900 GB/s            |                        |

표: 세대별 NVLink 대역폭 비교, 이론적 사양 표시

기본적으로, NCCL은 사용 가능할 때 노드 내 GPU 통신에 NVLink를 우선시합니다. 동일한 머신의 GPU 간에 가장 낮은 지연 시간과 가장 높은 대역폭 경로를 제공하기 때문입니다. 그러나 플래그를 제대로 설정하지 않으면, NVLink 사용을 방해 할 수 있습니다! 😱

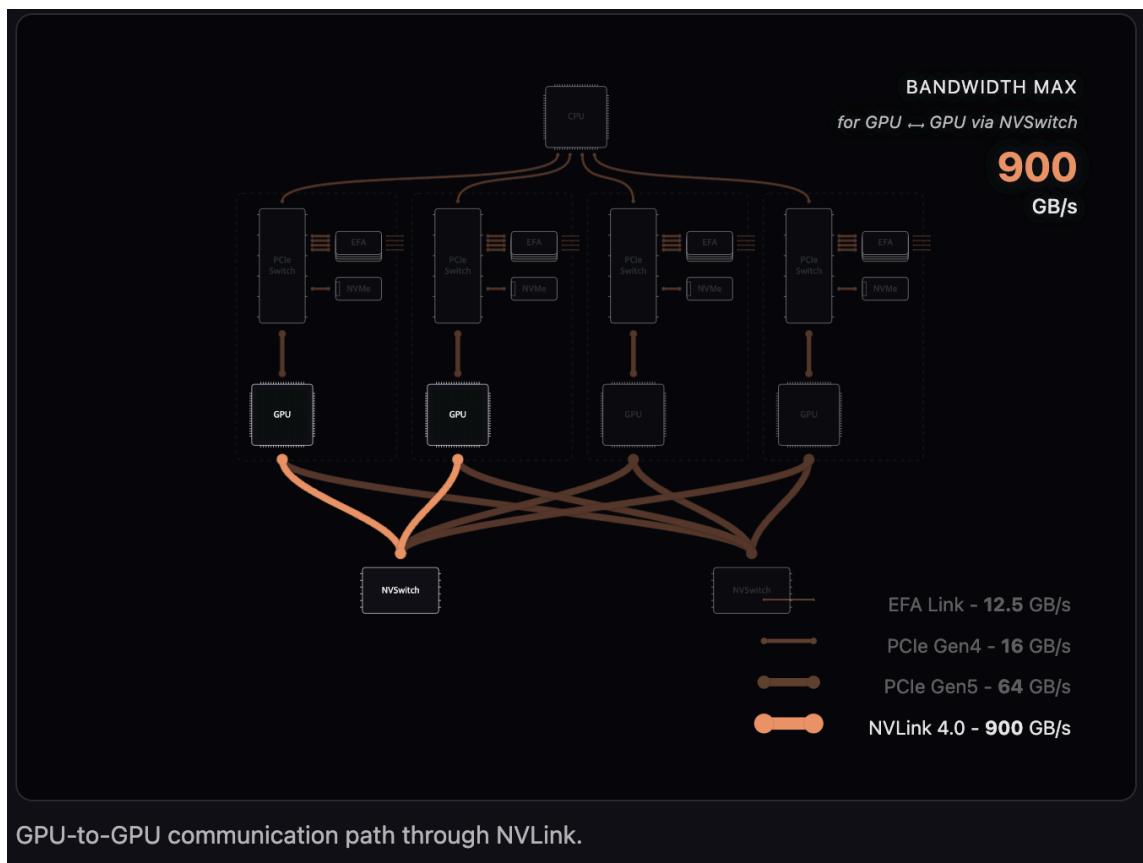
NVLink는 CPU나 시스템 메모리를 포함하지 않고 직접 GPU 간 메모리 접근을 가능하게 합니다. NVLink를 사용할 수 없을 때, NCCL은 PCIe를 통한 GPUDirect P2P로 풀백하거나, 소켓 간 PCIe 전송이 차선책일 때 공유 메모리(SHM) 전송을 사용합니다.

NVLink가 사용되고 있는지 확인하려면, `NCCL_DEBUG=INFO` 를 설정하고 다음과 같은 메시지를 찾으세요:

```
NCCL INFO Channel 00/1 : 0[0] -> 1[1] via P2P/CUMEM
```

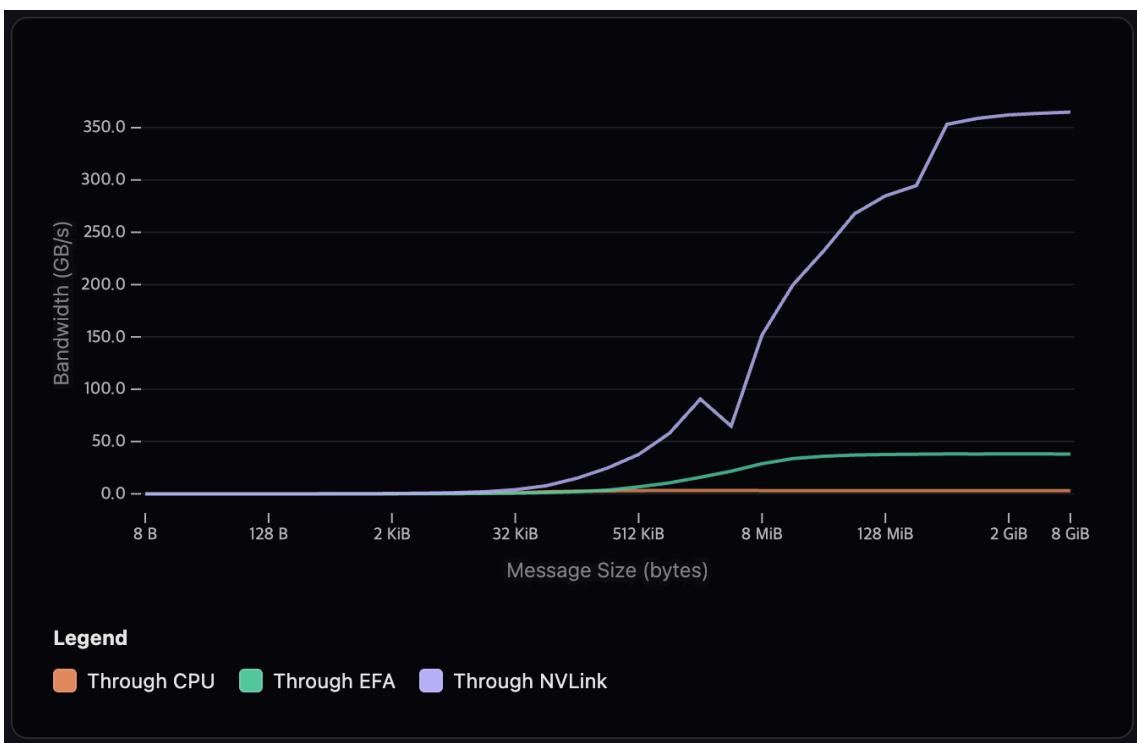
CUMEM은 피어 투 피어 연산이 CUDA 메모리 핸들(cuMem API)을 사용함을 나타냅니다. 여기서 더 알아보세요.

다음 다이어그램은 NVLink를 사용할 때 데이터가 취하는 직접 경로를 보여줍니다:



NVLink 4.0의 이론적 대역폭 900 GB/s를 EFA의 ~50 GB/s와 비교하면, 노드 내 통신에 대해 18배의 이점을 기대합니다. 이를 실제로 검증하기 위해, [NCCL의 SendRecv 성능 테스트](#)를 실행하여 다양한 통신 경로에 걸친 실제 대역폭을 측정했습니다:

```
$ FI_PROVIDER=XXX NCCL_P2P_DISABLE=X sendrecv_perf -b 8 -e 8G -f 2 -g 1 -c 1 -n 100
```



이것은 NVLink가 얼마나 더 효율적인지 의심의 여지 없이 보여줍니다: EFA의 38.16 GB/s(9배 더 빠름, 또는 양방향 18 배)와 CPU 기준선의 3.24 GB/s(112.6배 더 빠름)에 비해 364.93 GB/s를 달성합니다. 이러한 측정은 NCCL이 노드 내 GPU 통신에 NVLink를 우선시하는 이유를 확인하지만, NVLink의 성능을 추가로 확인하기 위해, `nvbandwidth` 를 사용하여 양방향으로 동시 복사를 사용하여 모든 GPU 쌍 간의 양방향 대역폭을 측정해 봅시다:

```
./nvbandwidth -t device_to_device_bidirectional_memcpy_write_ce -b <message_size> -i 5
memcpy CE GPU(row) <-> GPU(column) Total bandwidth (GB/s)

 0 1 2 3 4 5 6 7
0 N/A 785.81 785.92 785.90 785.92 785.78 785.92 785.90
1 785.83 N/A 785.87 785.83 785.98 785.90 786.05 785.94
2 785.87 785.89 N/A 785.83 785.96 785.83 785.96 786.03
3 785.89 785.85 785.90 N/A 785.96 785.89 785.90 785.96
4 785.87 785.96 785.92 786.01 N/A 785.98 786.14 786.08
5 785.81 785.92 785.85 785.89 785.89 N/A 786.10 786.03
6 785.94 785.92 785.99 785.99 786.10 786.05 N/A 786.07
7 785.94 786.07 785.99 786.01 786.05 786.05 786.14 N/A

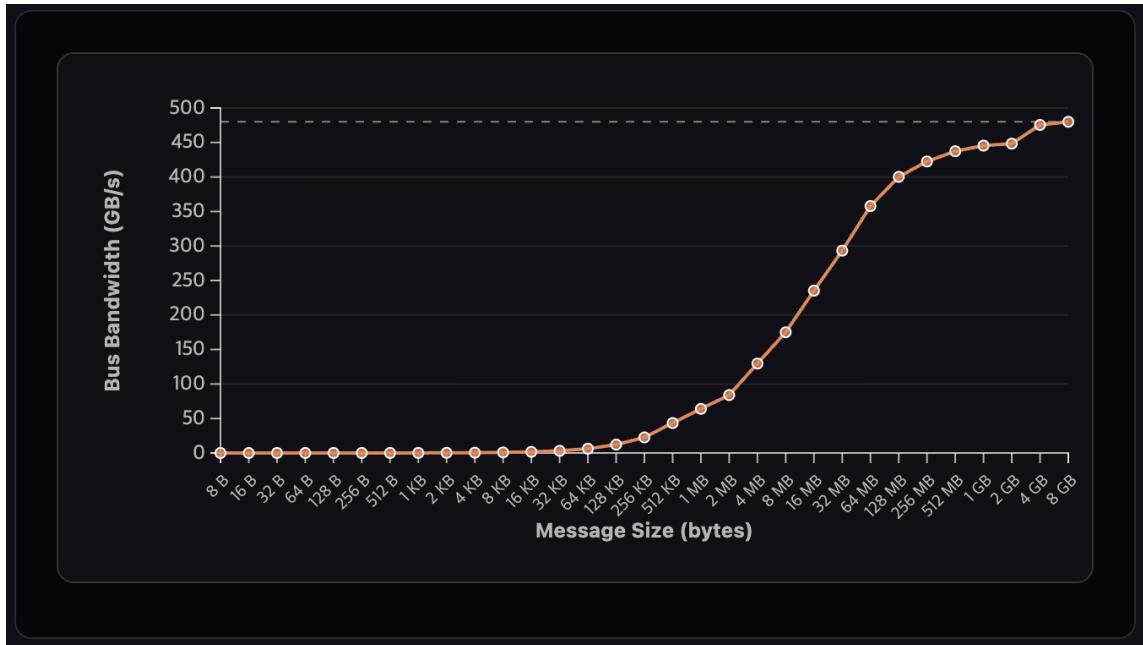
SUM device_to_device_bidirectional_memcpy_write_ce_total 44013.06
```

측정된 양방향 대역폭 786 GB/s는 NVLink 4.0의 이론적 900 GB/s 사양의 85%를 나타냅니다. NVLink를 사용하여 CPU 병목을 완전히 우회했습니다(GPU 간 통신에 대해)!

그러나 이것이 집합 통신 패턴으로 어떻게 변환될까요? [NCCL 테스트](#)의 `all_reduce_perf` 벤치마크로 단일 노드 내에서 `allreduce` 성능을 측정해 봅시다.

집합 통신 패턴에 대한 빠른 복습은 *UltraScale Playbook* 부록을 참조하세요.

```
$./all_reduce_perf -b 8 -e 16G -f 2 -g 1 -c 1 -n 100
```



그런데 잠깐... 480 GB/s를 달성하고 있는데, 이것은 NVLink 4.0의 이론적 단방향 대역폭 450 GB/s를 초과합니다 😲 이 마법은 무엇이며, 어떻게 가능한 걸까요?

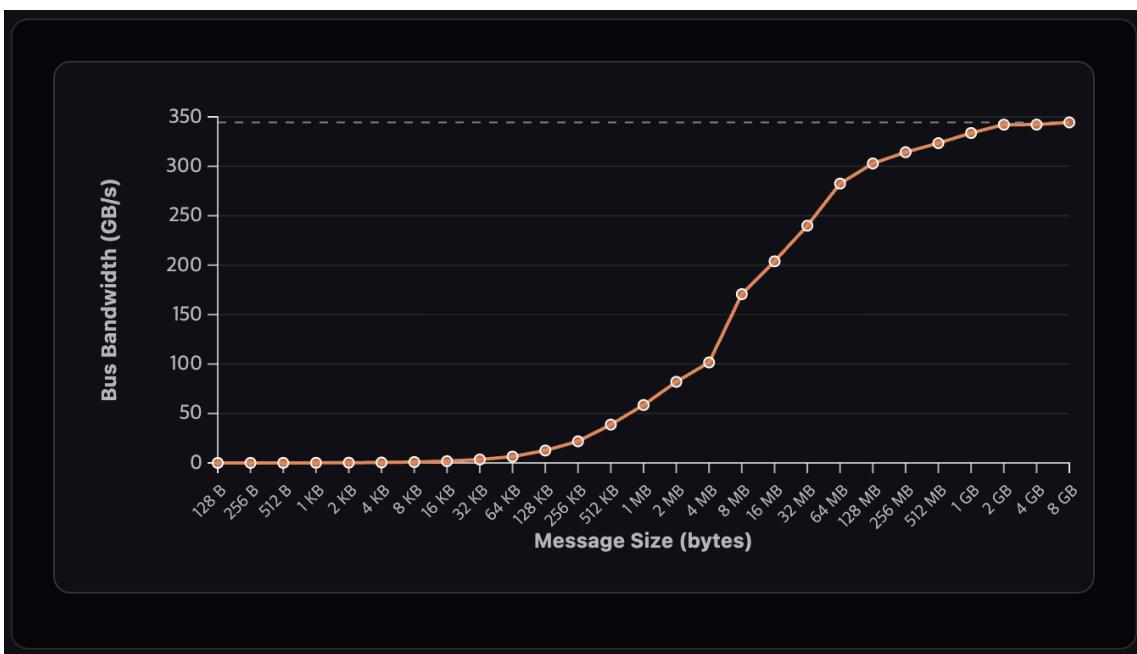
문서를 조금 살펴보면, 답은 NVIDIA의 하드웨어 가속 집합 연산 기술인 \*\*NVLink SHARP(NVLS)\*\*에 있는 것 같습니다. H100 GPU가 있는 단일 노드에서 allreduce 연산에 대해 약 1.3배 속도 향상을 제공합니다!



NVSwitch가 이러한 하드웨어 가속 집합 연산을 어떻게 가능하게 하는지에 대한 기술적 세부 사항은 [NVSwitch 아키텍처 프로젝트](#)를 참조하세요.

다른 곳에서도 도움이 될 수 있을까요? [alltoall 성능](#)을 검사해 봅시다:

```
$./all_to_all_perf -b 8 -e 16G -f 2 -g 1 -c 1 -n 100
```



alltoall 연산에서 340 GB/s를 달성하며, 이것은 NVLink 4.0을 가진 H100 시스템에 대한 유사한 성능 특성을 보여주는 공개된 벤치마크와 일치합니다([source](#)). allreduce와 달리, alltoall 연산은 NVLS 하드웨어 가속의 이점을 받지 못하며, 이것이 allreduce로 달성한 480 GB/s에 비해 여기서 340 GB/s를 보는 이유를 설명합니다. alltoall 패턴은 모든 GPU 쌍 간에 더 복잡한 점대점 데이터 교환을 필요로 하며, NVSwitch의 집합 가속 기능이 아닌 NVLink의 기본 대역폭에 순전히 의존합니다.

커스텀 NVLink 통신 패턴의 경우, NVLink와 NVLS 연산에 대한 세밀한 제어를 가능하게 하는 PyTorch의 SymmetricMemory API를 주목하세요.

### ⚡ 고급 커널 최적화

일부 최적화된 커널은 전송을 처리하기 위해 전용 워프를 할당하여 NVLink 통신을 컴퓨팅에서 분리합니다. 예를 들어, ThunderKittens는 특정 워프가 NVLink 전송을 발행하고 완료를 기다리는 반면, 다른 워프는 컴퓨팅 연산을 계속하는 워프 수준 설계를 사용합니다. SM 컴퓨팅과 NVLink 통신의 이러한 세밀한 중첩은 대부분의 GPU 간 통신 지연 시간을 줄일 수 있습니다. 구현 세부 사항은 멀티 GPU 커널에 대한 [ThunderKittens 블로그 포스트](#)를 참조하세요.

NVLink가 단일 노드 내에서 탁월한 대역폭을 제공하지만, 프론티어 모델 학습은 여러 노드에 걸쳐 스케일링해야 합니다.

이것은 새로운 잠재적 병목을 도입합니다: NVLink보다 상당히 낮은 대역폭에서 작동하는 노드 간 네트워크 상호 연결입니다.

## GPU에서 GPU로 노드 간

**TL;DR** 멀티 노드 GPU 통신은 InfiniBand(400 Gbps) 또는 RoCE(100 Gbps)와 같은 고속 네트워크를 사용합니다. Allreduce는 잘 스케일링됩니다(노드 전체에서 320-350 GB/s 안정), 대규모 학습 클러스터를 가능하게 합니다. Alltoall은 알고리즘 복잡성으로 인해 더 급격히 저하됩니다. 지연 시간은 노드 내 ~13μs에서 노드 간 55μs+로 점프합니다. 빈번한

all-to-all 연산이 필요한 MoE 워크로드의 경우, NVSHMEM은 CPU 조율 전송보다 상당히 더 나은 성능으로 비동기 GPU 시작 통신을 제공합니다.

모델이 단일 노드가 수용할 수 있는 것 이상으로 스케일링됨에 따라, 학습은 고속 네트워크를 통해 연결된 여러 노드에 걸쳐 계산을 분산해야 합니다. 벤치마크에 뛰어들기 전에, 멀티 노드 GPU 클러스터에서 만나게 될 노드를 연결하는 3가지 핵심 네트워킹 기술을 살펴봅시다:

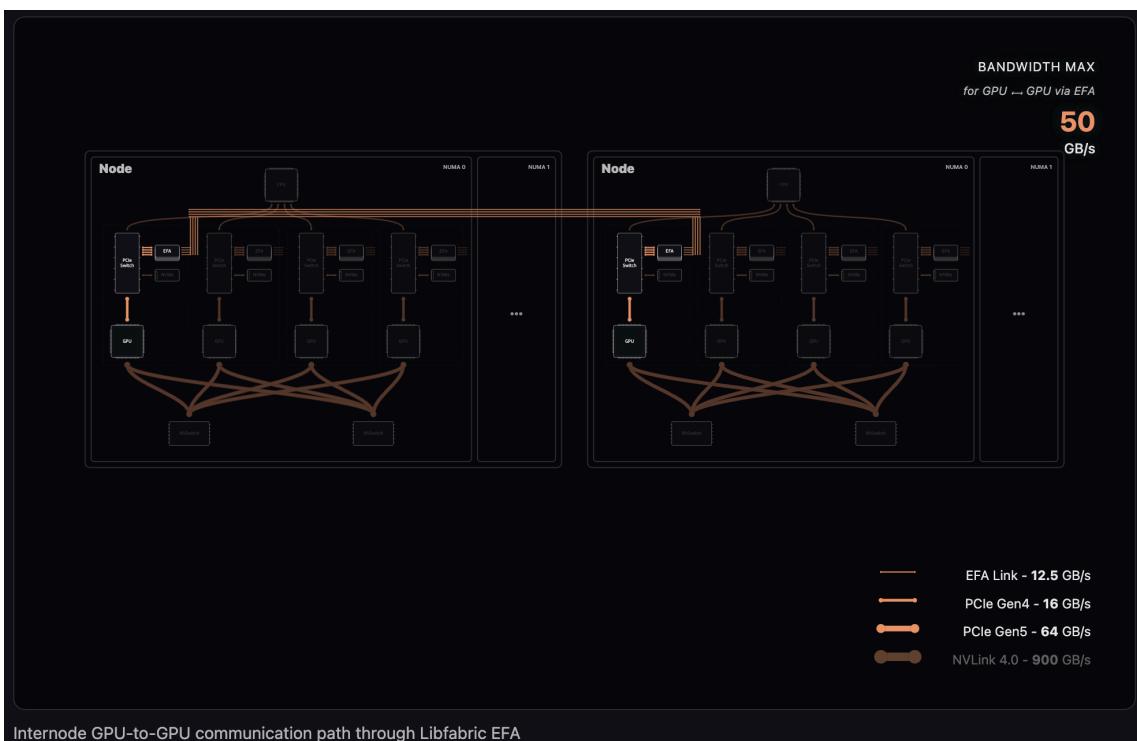
- **Ethernet**은 1 Gbps에서 100+ Gbps 속도로 발전했으며 HPC와 데이터 센터 클러스터에서 여전히 널리 사용 됩니다.
- \*\*RoCE(RDMA over Converged Ethernet)\*\*는 전통적인 TCP 메커니즘 대신 혼잡 제어를 위해 ECN을 사용 하여 이더넷 네트워크에 RDMA 능력을 제공합니다.
- **InfiniBand**은 NVIDIA의 산업 표준 스위치 패브릭으로, 최대 400 Gbps 대역폭과 서브 마이크로초 지연 시간 을 제공하며 GPUDirect RDMA를 통해 호스트 CPU를 우회하면서 직접 GPU 간 메모리 접근을 가능하게 하는 RDMA 지원을 제공합니다.

요약하면:

| 이름              | Ethernet (25–100 Gbps) | Ethernet (200–400 Gbps) | RoCE | Infiniband      |
|-----------------|------------------------|-------------------------|------|-----------------|
| 제조업체            | Many                   | Many                    | Many | NVIDIA/Mellanox |
| 단방향 대역폭 (Gbps)  | 25–100                 | 200–400                 | 100  | 400             |
| 종단 간 지연 시간 (μs) | 10–30                  | N/A                     | ~1   | <1              |
| RDMA            | No                     | No                      | Yes  | Yes             |

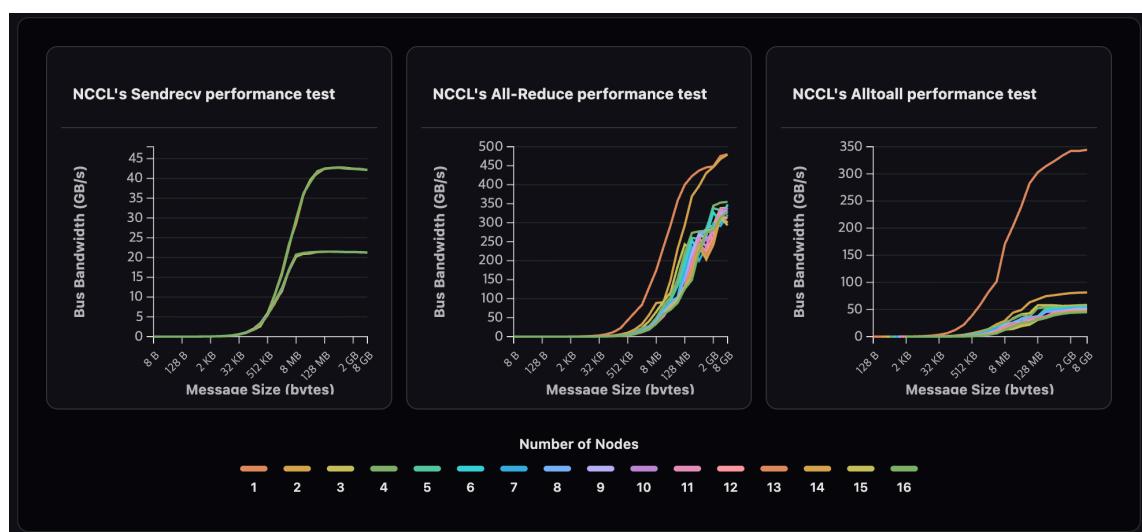
표: 상호 연결 비교. 출처: <https://www.sciencedirect.com/science/article/pii/S2772485922000618>

AWS p5 인스턴스의 경우 NIC(네트워크 인터페이스 카드)로 [Elastic Fabric Adapter \(EFA\)](#)가 있으며, 이전에 보았듯이 각 GPU는 PCIe Gen5 x16 레인을 통해 4개의 100 Gbps EFA 네트워크 카드에 연결됩니다.



위에 설명된 것처럼, GPU와 네트워크 카드가 동일한 PCIe 스위치에 연결되면, GPUDirect RDMA는 그들의 통신이 오직 그 스위치를 통해서만 발생하도록 합니다. 이 설정은 PCIe Gen5 x16 대역폭의 완전한 활용을 허용하고 다른 PCIe 스위치나 CPU 메모리 버스를 포함하는 것을 피합니다. 이론적으로, 노드당 8개의 PCIe 스위치 x 스위치당 4개의 EFA NIC x EFA NIC당 100 Gbps는 \*\*3200 Gbps(400GB/s)\*\*의 대역폭을 제공하며, 이것은 [AWS p5의 사양](#)에서 찾는 대역폭입니다. 그렇다면 실제로는 어떨까요? 이전과 동일한 벤치마크를 실행하되 다른 노드에 걸쳐 실행하여 알아봅시다!

대역폭 분석



점대점 송신/수신 연산은 2-4개 노드에서 약 42-43 GB/s를 달성하지만 5개 이상의 노드에서는 약 21 GB/s로 떨어집니다. 이 성능 저하는 NCCL이 4개 노드를 넘어 스케일링할 때 피어당 점대점 채널 수를 2에서 1로 자동으로 줄여 사용 가능한 대역폭 활용을 효과적으로 반으로 줄이기 때문에 발생하며, 이론적 최대값은 ~50 GB/s(4개 EFA NIC × 각각 12.5 GB/s)로 유지됩니다. NCCL\_NCHANNELS\_PER\_NET\_PEER=2 를 설정하여 5개 이상의 노드에서 이 테스트의 전체 처리량을 성공적으로 복원했지만, 이 플래그는 예를 들어 all-to-all 성능을 저하시킬 수 있으므로 주의해서 사용해야 합니다(자세한 내용은 [GitHub issue #1272](#) 참조).

all-reduce 연산은 단일 노드 내에서 480 GB/s의 버스 대역폭을 달성하며 우수한 성능을 보여줍니다. 2개 노드로 스케일링할 때 대역폭은 479 GB/s로 거의 동일하게 유지되며, 그 후 3-16개 노드에서 약 320-350 GB/s로 안정화됩니다. 이 패턴은 중요한 특성을 드러냅니다: NVLink에서 노드 간 네트워크 패브릭으로의 전환으로 인해 노드 경계를 넘을 때 초기 하락이 있지만, 더 많은 노드를 추가해도 대역폭은 거의 일정하게 스케일링됩니다.

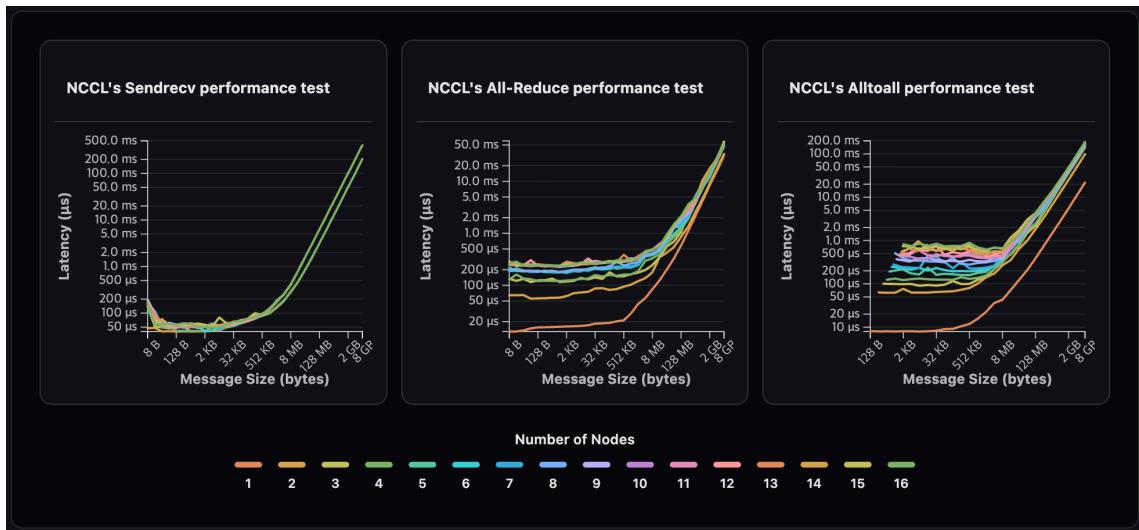
#### 노드 전체에서 All-Reduce 스케일링

2개 노드 이상에서의 이 거의 일정한 스케일링 동작은 실제로 대규모 학습에 꽤 고무적입니다. 3-16개 노드에 걸쳐 상대적으로 안정적인 320-350 GB/s는 all-reduce 연산에 의존하는 병렬성 전략(예: 데이터 병렬성)이 상당한 GPU당 대역폭 저하 없이 수백 또는 수천 개의 GPU로 스케일링할 수 있음을 시사합니다. 이 로그 스케일링 특성은 8개의 GPU가 각각 별도의 스위치 레일에 연결되어 이분 대역폭을 최대화하는 8-레일 최적화 패트리를 사용하는 잘 설계된 다중 계층 네트워크 토플로지의 전형입니다. 현대 프론티어 학습 클러스터는 일상적으로 100,000개 이상의 GPU에서 작동하며, 이 안정적인 스케일링 동작이 그러한 대규모 배포를 가능하게 합니다.

다른 대역폭 링크(노드 내 NVLink vs. 노드 간 네트워크)로 작업할 때, 사용 가능한 모든 대역폭을 완전히 활용하기 위해 각 대역폭 계층에 병렬성 전략을 적용시키는 것을 고려하세요. 이 기준 네트워크 토플로지에 대한 병렬성 구성 최적화에 대한 자세한 지침은 [Ultrascale playbook](#)을 참조하세요.

all-to-all 연산은 더 극적인 스케일링 도전을 보여줍니다: 단일 노드에서 344 GB/s로 시작하여, 2개 노드에서 81 GB/s로 떨어지고 더 큰 클러스터에서 약 45-58 GB/s로 계속 감소합니다. 이 더 가파른 저하는 각 GPU가 노드 전체의 다른 모든 GPU와 통신해야 하는 all-to-all 패턴의 집중적인 네트워크 요구를 반영하며, all-reduce 연산보다 상당히 더 많은 네트워크 혼잡을 생성합니다.

## 지연 시간 분석



지연 시간 측정은 노드 경계를 넘는 근본적인 비용을 드러냅니다. 송신/수신 연산은 모든 멀티 노드 구성에서 40-53 μs의 상대적으로 안정적인 지연 시간을 유지하며, 점대점 통신 지연 시간이 클러스터 크기보다 기본 네트워크 왕복 시간에 의해 주로 결정됨을 보여줍니다. 일부 변동은 네트워크 토플로지와 라우팅 효과가 여전히 역할을 한다는 것을 시사합니다.

All-reduce 연산은 단일 노드 내에서 12.9 μs의 최소 지연 시간을 보여주지만, 2개 노드에서 55.5 μs로 점프하고 클러스터 크기에 따라 거의 선형적으로 계속 증가하여 16개 노드에서 235 μs에 도달합니다. 이 진행은 증가된 통신 거리와 더 많은 노드에 걸친 리덕션 트리의 증가하는 복잡성을 모두 반영합니다.

All-to-all 연산은 유사한 추세를 보여주며, 단일 노드 통신에서 7.6 μs로 시작하지만 2개 노드에서 60 μs로 올라가고 16개 노드에서 621 μs에 도달합니다. all-to-all 연산의 초선형 지연 시간 증가는 더 많은 노드가 집합체에 참여함에 따라 네트워크 혼잡과 조정 오버헤드가 복합됨을 나타냅니다.

### 🚀 최적화된 GPU 통신을 위한 NVSHMEM

전문가 라우팅을 위해 빈번한 all-to-all 통신 패턴을 필요로 하는 Mixture of Experts(MoE) 아키텍처의 부상과 함께, 최적화된 GPU 통신 라이브러리가 점점 더 중요해졌습니다.

[NVSHMEM](#)은 여러 GPU의 메모리를 분할된 전역 주소 공간(PGAS)으로 결합하는 고성능 통신 라이브러리로 상당한 견인력을 얻고 있습니다. CPU 조율 데이터 전송에 의존하는 전통적인 MPI 기반 접근 방식과 달리, NVSHMEM은 CPU-GPU 동기화 오버헤드를 제거하는 비동기 GPU 시작 연산을 가능하게 합니다.

NVSHMEM은 GPU 통신에 여러 핵심 이점을 제공합니다: GPUDirect Async와 같은 기술을 통해, GPU는 노드 간 통신을 발행할 때 CPU를 완전히 우회할 수 있어 작은 메시지(<1 KiB)에 대해 최대 9.5배 더 높은 처리량을 달성합니다. 이것은 집중적인 네트워크 통신 패턴이 필요한 집합 연산에 특히 유익합니다.

라이브러리는 현재 Mellanox 어댑터(CX-4 이상)를 가진 InfiniBand/RoCE, Slingshot-11(Libfabric CXI), Amazon EFA(Libfabric EFA)를 지원합니다. 세밀한 통신으로 강력한 스케일링이 필요한 애플리케이션의 경우,

NVSHMEM의 저오버헤드 단방향 통신 프리미티브는 전통적인 CPU 프록시 방법에 비해 성능을 상당히 개선할 수 있습니다.

[NVSHMEM 문서와 GPUDirect Async에 대한 이 상세한 블로그 포스트](#)에서 더 알아보세요.

대역폭 측정이 기대에 미치지 못할 때, 여러 요소가 성능을 제한할 수 있습니다. 이러한 잠재적 병목을 이해하는 것은 최적의 상호 연결 활용을 달성하는 데 필수적입니다.

## 상호 연결 문제 해결

기대보다 낮은 대역폭을 경험하고 있다면, 다음 영역을 체계적으로 확인하세요:

### 라이브러리 버전

오래된 NCCL, EFA 또는 CUDA 라이브러리는 중요한 성능 최적화나 버그 수정이 부족할 수 있습니다. 항상 모든 통신 라이브러리의 최신 호환 버전을 실행하고 있는지 확인하세요. 예를 들어, AWS는 하드웨어에 최적화된 라이브러리 버전으로 Deep Learning AMI를 정기적으로 업데이트합니다. 중요한 실험에 대해 이러한 라이브러리 버전을 로깅하는 것도 권장됩니다.

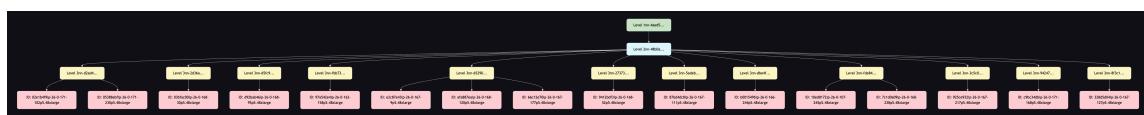
### CPU 친화성 구성

부적절한 CPU 친화성 설정은 불필요한 교차 NUMA 트래픽을 유발하여 NCCL 성능에 상당히 영향을 미칠 수 있습니다. 각 GPU는 메모리 접근 시간을 최소화하기 위해 동일한 NUMA 노드의 CPU에 바인딩되어야 합니다. 실제로, 이 [Github issue](#)는 NCCL\_IGNORE\_CPU\_AFFINITY=1 과 --cpu-bind none 을 사용하여 컨테이너 자연 시간을 상당히 줄이는 데 어떻게 도움이 되었는지 보여줍니다. [여기서](#) 더 읽을 수 있습니다.

### 네트워크 토폴로지와 배치

네트워크 토폴로지를 이해하는 것은 성능 문제를 진단하는 데 중요합니다. 클라우드 배치 그룹은 도움이 되지만, 인스턴스 간 최소 네트워크 흡을 보장하지 않습니다. 현대 데이터센터 패트 트리 토폴로지에서, 다른 최상위 스위치 아래에 배치된 인스턴스는 라우팅 경로의 추가 네트워크 흡으로 인해 더 높은 자연 시간과 잠재적으로 더 낮은 대역폭을 경험할 것입니다.

AWS EC2 사용자의 경우, Instance Topology API는 네트워크 노드 배치에 대한 귀중한 가시성을 제공합니다. 맨 아래 계층(인스턴스에 직접 연결)에서 동일한 네트워크 노드를 공유하는 인스턴스는 물리적으로 가장 가깝고 가장 낮은 자연 시간 통신을 달성할 것입니다.



통신하는 노드 간의 네트워크 흡을 최소화하면 더 나은 상호 연결 성능으로 직접 이어집니다. 소규모 실험과 절제 실험의 경우, 인스턴스가 동일한 네트워크 스위치에 함께 위치하도록 보장하면 자연 시간과 대역폭 활용 모두에서 측정 가능한 차이를 만들 수 있습니다.

### 올바른 환경 변수

네트워크 어댑터에 대한 누락되거나 잘못된 환경 변수는 대역폭 활용을 심각하게 제한할 수 있습니다. NCCL과 같은 통신 라이브러리는 적응형 라우팅, GPU 시작 전송, 적절한 버퍼 크기 조정과 같은 최적의 성능 기능을 활성화하기 위해 특정 구성 플래그에 의존합니다.

예를 들어, AWS EFA(Elastic Fabric Adapter)를 사용할 때, 인스턴스 유형에 대해 권장되는 NCCL 및 EFA 환경 변수를 설정하고 있는지 확인하세요. [AWS EFA cheatsheet](#)는 다양한 시나리오에 대한 최적의 플래그 구성에 대한 포괄적인 지침을 제공합니다.

### 컨테이너 관련 고려 사항

컨테이너(Docker/Enroot)를 사용할 때, 최적의 NCCL 성능을 위해 여러 구성 단계가 중요합니다:

- 공유 및 고정 메모리:** Docker 컨테이너는 기본적으로 제한된 공유 및 고정 메모리 리소스를 갖습니다. 초기화 실패를 방지하기 위해 `-shm-size=1g --ulimit memlock=-1`로 컨테이너를 시작하세요.
- NUMA 지원:** Docker는 기본적으로 NUMA 지원을 비활성화하여 cuMem 호스트 할당이 올바르게 작동하지 않을 수 있습니다. `-cap-add SYS_NICE`로 Docker를 호출하여 NUMA 지원을 활성화하세요.
- PCI 토플로지 검색:** NCCL이 GPU와 네트워크 카드의 PCI 토플로지를 검색할 수 있도록 `/sys` 가 올바르게 마운트되었는지 확인하세요. `/sys` 가 가상 PCI 토플로지를 노출하면 차선의 성능이 발생할 수 있습니다.

💡 **커뮤니티 문제 해결** 우리는 커뮤니티 노력으로 여기에 문제 해결 발견을 수집하고 있습니다. 성능 문제를 겪었거나 효과적인 디버깅 방법을 발견했다면, [Discussion Tab](#)으로 이동하여 다른 사람들이 상호 연결 활용을 최적화하는데 도움이 되도록 경험을 공유해 주세요.

이제 GPU-CPU와 GPU-GPU 통신의 병목을 디버그하는 방법을 알았으니, 일반적으로 덜 주목받는 GPU 통신 부분, 즉 스토리지 레이어와의 통신을 살펴봅시다!

### GPU에서 스토리지로

GPU와 스토리지 시스템 간의 연결은 종종 간과되지만 학습 효율성에 상당히 영향을 미칠 수 있습니다. 학습 중에 GPU는 스토리지에서 지속적으로 데이터를 읽고(데이터 로딩, 특히 대용량 이미지/비디오 파일이 있는 멀티모달 데이터의 경우) 주기적으로 모델 상태를 스토리지에 다시 써야 합니다(체크포인팅). 현대 대규모 학습 실행의 경우, 이러한 I/O 연산이 제대로 최적화되지 않으면 병목이 될 수 있습니다.

**TL;DR:** GPU-스토리지 I/O는 데이터 로딩과 체크포인팅을 통해 학습에 영향을 미칩니다. GPUDirect Storage(GDS)는 CPU를 우회하여 더 나은 성능을 위해 직접 GPU에서 스토리지로의 전송을 가능하게 합니다. 클러스터에서 GDS가 활성화되지 않았더라도, 로컬 NVMe RAID(RAID 0의 8x3.5TB 드라이브)는 26.59 GiB/s와 337K IOPS(네트워크 스토리지보다 6.3배 빠름)를 제공하여 체크포인트에 이상적입니다.

### 스토리지 토플로지 이해

GPU와 스토리지 디바이스 간의 물리적 연결은 GPU 상호 연결과 유사한 계층적 구조를 따릅니다. 스토리지 디바이스는 PCIe 브릿지를 통해 연결되며, 이 토플로지를 이해하면 성능 특성과 잠재적 병목을 설명하는 데 도움이 됩니다.

`lstopo` 의 시스템 토플로지를 보면, NVMe 드라이브가 시스템에 어떻게 연결되는지 볼 수 있습니다. p5 인스턴스에는 GPU당 1개의 NVMe SSD가 있습니다:

```
PCIBridge L#13 (busid=0000:46:01.5 id=1d0f:0200 class=0604(PCIBridge) link=15.75GB/s buses=0000:
[54-54] PCIVendor="Amazon.com, Inc."
PCI L#11 (busid=0000:54:00.0 id=1d0f:cd01 class=0108(NVMe) link=15.75GB/s PCISlot=87-1
PCIVendor="Amazon.com, Inc." PCIDevice="NVMe SSD Controller")
Block(Disk) L#9 (Size=3710937500 SectorSize=512 LinuxDeviceID=259:2 Model="Amazon EC2 NVMe
Instance Storage" Revision=0 SerialNumber=AWS110C9F44F9A530351) "nvme1n1"
```

자연스러운 질문은 GPU가 CPU를 포함하지 않고 NVMe 드라이브에 직접 접근할 수 있는지 여부입니다. 답은 예이며, \*\*GPUDirect Storage(GDS)\*\*를 통해 가능합니다.

GPUDirect Storage는 스토리지(로컬 NVMe 또는 원격 NVMe-oF)와 GPU 메모리 간의 직접 데이터 경로를 가능하게 하는 NVIDIA의 [GPUDirect](#) 기술 제품군의 일부입니다. 스토리지 컨트롤러 근처의 DMA 엔진이 데이터를 GPU 메모리로 직접 이동하거나 GPU 메모리에서 직접 이동할 수 있게 하여 CPU 바운스 버퍼를 통한 불필요한 메모리 복사를 제거합니다. 이것은 CPU 오버헤드를 줄이고, 지연 시간을 감소시키며, 대규모 멀티모달 데이터셋에 대한 학습과 같은 데이터 집약적 워크로드의 I/O 성능을 상당히 개선합니다.

시스템에서 GPUDirect Storage가 올바르게 구성되었는지 확인하려면, GDS 구성 파일을 확인하고 제공된 진단 도구를 사용할 수 있습니다:

```
$ /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
=====
```

NVMe: Supported를 볼 수 있으며, 이것은 GDS가 현재 NVMe 드라이브에서 작동하도록 구성되어 있고 다른 모든 스토리지 유형은 Unsupported 플래그에서 명백하듯이 올바르게 구성되지 않았음을 알려줍니다. 스토리지 유형에 대해 GDS 가 올바르게 구성되지 않았다면, [/etc/cufile.json](#) 의 구성 파일을 수정하는 지침에 대해 [NVIDIA GPUDirect Storage Configuration Guide](#)를 참조하세요.

## 블록 스토리지 디바이스

시스템에서 사용 가능한 스토리지 디바이스를 이해하려면, `lsblk` 를 사용하여 블록 디바이스 계층을 표시할 수 있습니다:

```
$ lsblk --fs -M
 NAME FSTYPE LABEL UUID
FSAVAIL FSUSE% MOUNTPOINT
...
nvme0n1
└─nvme0n1p1 ext4 cloudimg-rootfs 24ec7991-cb5c-4fab-99e5-52c45690ba30
189.7G 35% /
└─► nvme1n1 linux_raid_member ip-26-0-164-236:MY_RAID d0795631-71f0-37e5-133b-e748befec126
├─► nvme2n1 linux_raid_member ip-26-0-164-236:MY_RAID d0795631-71f0-37e5-133b-e748befec126
├─► nvme3n1 linux_raid_member ip-26-0-164-236:MY_RAID d0795631-71f0-37e5-133b-e748befec126
├─► nvme8n1 linux_raid_member ip-26-0-164-236:MY_RAID d0795631-71f0-37e5-133b-e748befec126
├─► nvme5n1 linux_raid_member ip-26-0-164-236:MY_RAID d0795631-71f0-37e5-133b-e748befec126
├─► nvme4n1 linux_raid_member ip-26-0-164-236:MY_RAID d0795631-71f0-37e5-133b-e748befec126
├─► nvme6n1 linux_raid_member ip-26-0-164-236:MY_RAID d0795631-71f0-37e5-133b-e748befec126
└─► nvme7n1 linux_raid_member ip-26-0-164-236:MY_RAID d0795631-71f0-37e5-133b-e748befec126
└...md0 xfs ddd6849-e5b5-4828-9034-96da65da27f0
27.5T 1% /scratch
```

이 출력은 시스템의 블록 디바이스 계층을 보여줍니다. 핵심 관찰:

- nvme0n1p1 은 / 에 마운트된 루트 [Amazon EBS](#) 파일시스템으로, 전체 ~300GB 용량의 35%를 사용 중
- 8개의 NVMe 드라이브( nvme1n1 부터 nvme8n1 까지)가 MY\_RAID라는 RAID 배열로 구성됨
- RAID 배열은 /dev/md0 으로 노출되고, XFS로 포맷되며, 28TB 사용 가능(8x3.5TB)으로 /scratch 에 마운트됨

화살표(→)는 여러 NVMe 디바이스가 동일한 RAID 배열의 멤버임을 나타내며, 그런 다음 단일 md0 디바이스로 결합됨

*Amazon Elastic Block Store(EBS)*는 Amazon EC2 인스턴스와 함께 사용하도록 설계된 고성능 확장 가능한 블록 스토리지 서비스입니다.

## 네트워크 스토리지

로컬 NVMe 스토리지 외에도, 시스템은 네트워크 연결 스토리지 시스템에 접근할 수 있습니다:

```
$ df -h
Filesystem Size Used Avail Use% Mounted on
/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
```

이 출력은:

- /dev/root (291GB Amazon EBS)는 35% 용량의 루트 파일시스템
- /fsx (393TB WekaFS)는 131TB 사용 가능으로 67% 찬 상태
- /admin (4.5TB FSx Lustre)은 1.7TB 사용 가능으로 63% 찬 상태
- /dev/md0 (28TB 로컬 NVMe RAID)는 /scratch 에서 28TB 사용 가능으로 단 1% 찬 상태. 이것은 RAID의 8x3.5TB SSD NVMe 인스턴스 스토어 드라이브입니다.

`/fsx` 는 실제로 Amazon FSx가 아니라 WekaFS입니다. FSx에서 WekaFS로 마이그레이션할 때 편의를 위해 동일한 마운트 포인트 이름을 유지했습니다.

로컬 NVMe RAID 배열( `/scratch` )은 가장 빠른 I/O 성능을 제공하고, 네트워크 파일시스템은 공유 데이터 스토리지를 위한 더 큰 용량을 제공합니다.

## ▣ 스토리지 기술

**RAID(Redundant Array of Independent Disks):** 데이터 스트라이핑, 패리티 또는 미러링을 통해 성능 및/또는 신뢰성을 개선하기 위해 여러 드라이브를 결합합니다.

**NVMe(Non-Volatile Memory Express):** PCIe에 직접 연결되어 SATA/SAS보다 더 높은 처리량과 더 낮은 지연 시간을 제공하는 SSD용 고성능 스토리지 프로토콜입니다.

**WekaFS:** AI/ML 워크로드를 위해 설계된 고성능 병렬 파일 시스템으로, 여러 노드에 걸쳐 낮은 지연 시간 접근과 높은 처리량을 제공합니다.

**FSx Lustre:** 병렬 접근을 가능하게 하기 위해 메타데이터와 데이터 서비스를 다른 서버에 분리하는 HPC용으로 설계된 병렬 파일 시스템입니다. 대용량 파일에는 효과적이지만, 많은 작은 파일을 포함하는 메타데이터 집약적 AI/ML 워크로드에서는 어려움을 겪을 수 있습니다.

## 스토리지 대역폭 벤치마킹

각 스토리지 시스템의 성능 특성을 이해하기 위해, GPUDirect Storage(GDS)를 사용하여 읽기/쓰기 속도를 벤치마킹할 수 있습니다. 다양한 구성을 테스트하는 포괄적인 파라메트릭 벤치마크 스크립트입니다:

```
gdsio -f <disk_path>/gds_test.dat -d 0 -w <n_threads> -s 10G -i <io_size> -x 1 -I 1 -T 10
```

벤치마크는 처리량, 지연 시간, IOPS뿐만 아니라 다음과 같은 스토리지 시스템 성능을 평가합니다:

**확장성:** 다른 스레드 수와 I/O 크기에 따라 성능이 어떻게 변하는지. 이것은 다른 워크로드 패턴에 대한 최적의 구성을 드러냅니다:

- 작은 I/O 크기(64K에서 256K)는 일반적으로 IOPS를 최대화하지만 대역폭을 포화시키지 못할 수 있음
- 큰 I/O 크기(2M에서 8M)는 일반적으로 처리량을 최대화하지만 IOPS를 줄임
- 스레드 수는 둘 다에 영향을 미침: 더 많은 스레드가 하드웨어 한계까지 총 IOPS와 처리량을 증가시킬 수 있음

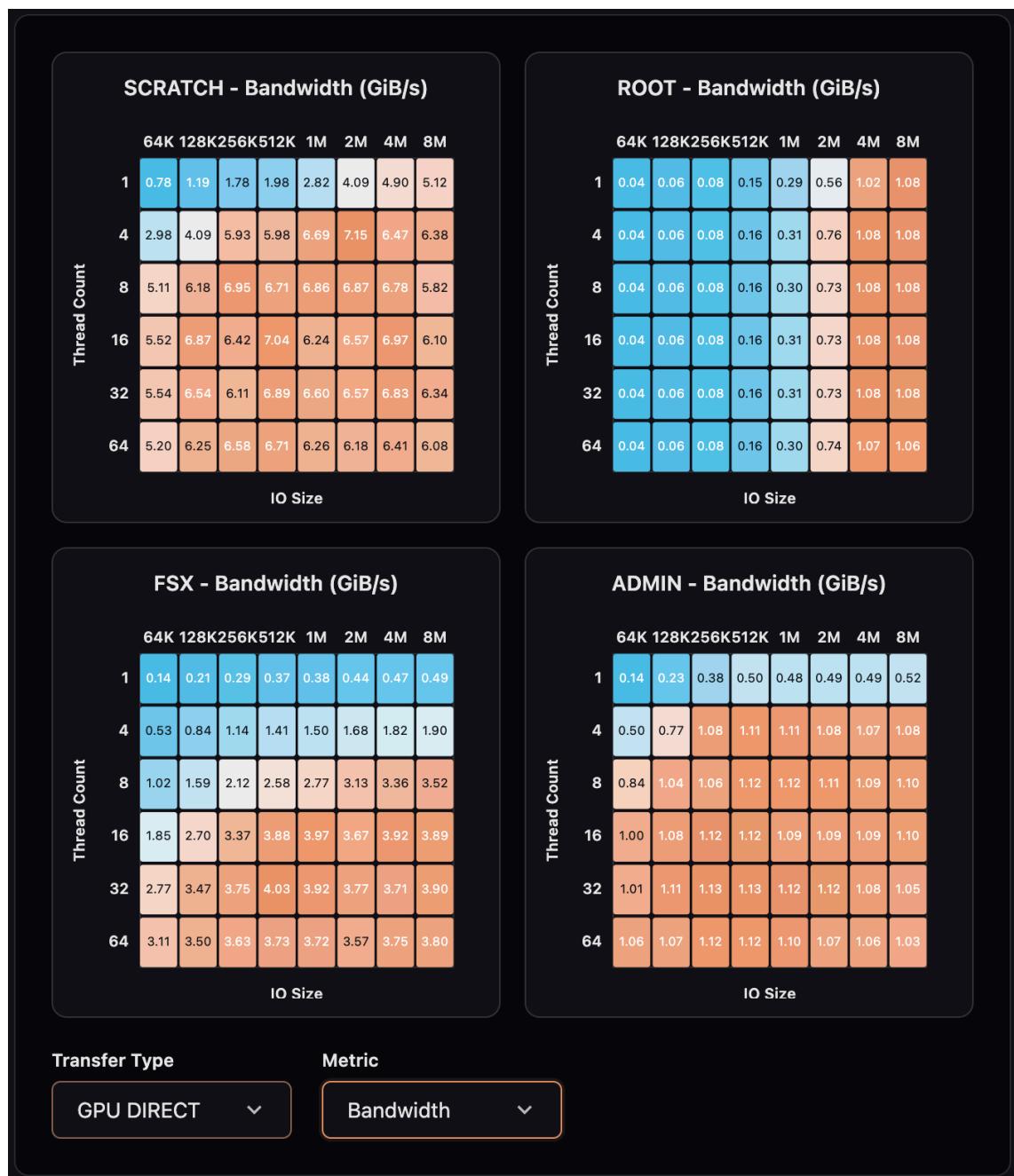
**전송 방법 효율성:** GPU\_DIRECT vs CPU\_GPU vs CPUONLY를 비교하면 CPU 메모리를 우회하는 이점을 보여줍니다:

- GPU\_DIRECT:** RDMA를 사용하여 CPU를 완전히 우회하고 데이터를 GPU 메모리로 직접 전송(가장 낮은 지연 시간, 가장 높은 효율성, 작은 연산에 대해 최고의 IOPS)
- CPU\_GPU:** 데이터가 먼저 CPU 메모리로 간 다음 GPU로 복사되는 전통적인 경로(CPU 오버헤드와 메모리 대역폭 경합 추가, 유효 IOPS 감소)
- CPUONLY:** GPU 참여 없는 기준 CPU 전용 I/O

## ▣ IOPS(초당 I/O 연산)

IOPS는 초당 완료된 개별 I/O 연산의 수입니다. gdsio 출력에서 `ops / total_time` 으로 계산됩니다. IOPS는 다음에 특히 중요합니다:

- 작은 I/O 크기의 랜덤 접근 패턴
- 많은 작은 파일이나 분산된 데이터 접근이 있는 워크로드
- 원시 대역폭보다 연산당 지연 시간이 더 중요한 데이터베이스와 같은 연산
- 더 높은 IOPS는 동시적이고 세밀한 데이터 접근을 처리하는 더 나은 능력을 나타냅니다



벤치마크는 네 가지 스토리지 시스템 전반에 걸쳐 극적인 성능 차이를 드러냅니다:

/scratch (로컬 NVMe RAID)는 26.59 GiB/s 처리량과 337K IOPS로 지배적이며, FSx보다 처리량에서 6.3배, IOPS에서 6.6배 더 좋습니다. 8x3.5TB NVMe 드라이브의 이 로컬 RAID 배열은 가장 낮은 지연 시간(피크 IOPS에서 190μs)을 제공하고 스레드 수에 따라 탁월하게 스케일링되며, 처리량에 대해 1M I/O 크기로 64개 스레드에서 피크 성능을 달성합니다.

/fsx (WekaFS)는 4.21 GiB/s와 51K IOPS로 견고한 네트워크 스토리지 성능을 제공하며, 합리적인 성능이 필요한 공유 데이터에 가장 좋은 선택입니다. FSx는 CPUONLY 전송을 사용하여 최고의 처리량(4.21 GiB/s)을 달성하고, GPUD 전송 유형을 사용하여 최고의 IOPS(51K)를 달성합니다.

/admin (FSx Lustre)과 /root (EBS) 파일시스템은 약 1.1 GiB/s 처리량 주위로 유사한 적당한 성능을 보여주지만, IOPS 능력에서 상당히 다릅니다. Admin은 GPUD 전송으로 피크 처리량(1.13 GiB/s)을 달성하고 CPU\_GPU 전송으로 17K IOPS에서 피크를 달성합니다(Root보다 24배 더 좋음). 이것은 많은 작은 연산이 있는 워크로드에 더 적합합니다. Root의 낮은 IOPS 성능(730)은 대용량 순차 연산에만 가장 적합함을 확인합니다.

**GPU\_DIRECT 결과에 대한 참고:** GPUDirect Storage(GDS)는 현재 클러스터에서 활성화되어 있지 않으며, 이것이 NVMe 스토리지(Scratch와 Root)에 대한 GPUD 결과가 CPUONLY 전송에 비해 성능이 낮은 이유를 설명합니다. GDS 가 올바르게 구성되면, 특히 고성능 NVMe 배열에 대해 직접 GPU에서 스토리지로의 전송에서 GPUD가 상당한 이점을 보일 것으로 예상합니다.

**최적 구성 패턴:** 모든 스토리지 유형에 걸쳐, 최대 처리량은 1M I/O 크기에서 발생하고, 최대 IOPS는 테스트된 가장 작은 크기(64K)에서 발생합니다. 이 고전적인 트레이드오프는 워크로드 특성에 따라 원시 대역폭(대용량 I/O)과 연산 동시성(소용량 I/O) 사이에서 선택하는 것을 의미합니다. 대용량 체크포인트 파일이 있는 ML 학습의 경우, Scratch의 1M-8M 범위가 최적의 성능을 제공합니다.

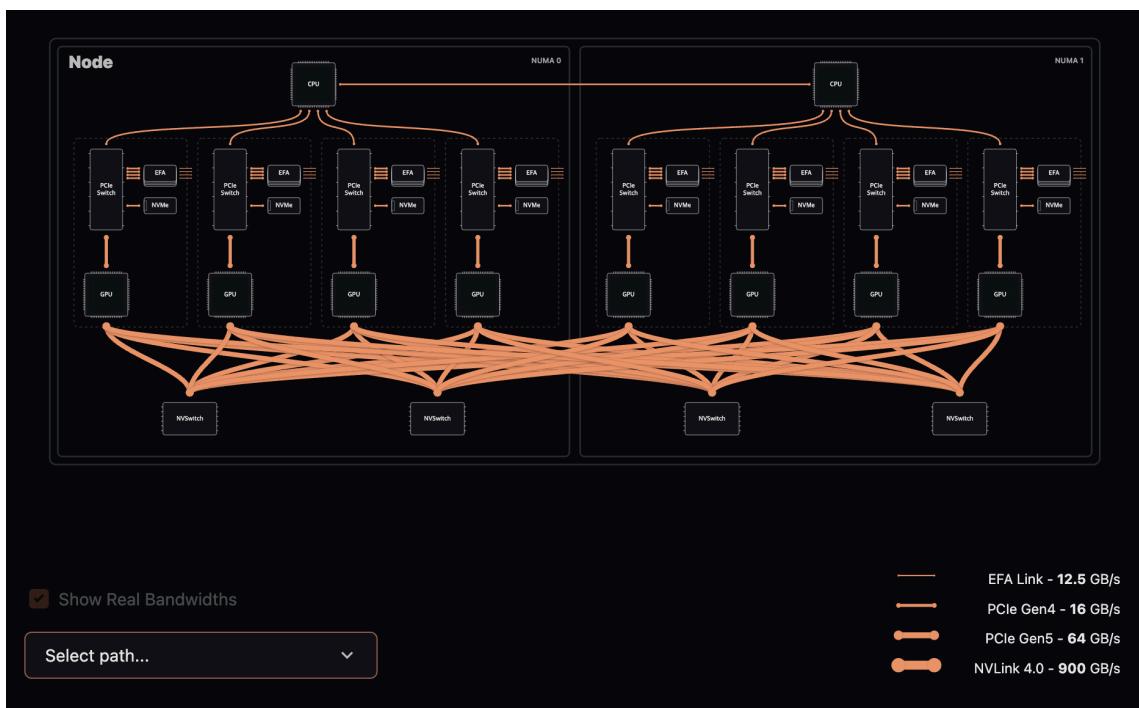
## 요약

여기까지 왔다면, 축하합니다! 이제 스토리지 계층과 학습 인프라에서 다양한 구성 요소가 어떻게 상호작용하는지에 대한 포괄적인 이해를 갖추었습니다. 그러나 집에 가져가길 바라는 핵심 통찰이 있습니다: **병목을 식별하는 것이 이론적 지식과 실용적 최적화를 구분합니다.**

이 가이드 전반에 걸쳐, 스택의 모든 수준에서 실제 대역폭을 측정했습니다: 단일 GPU 내 HBM3의 3TB/s, 노드 내 GPU 간 NVLink의 786 GB/s, CPU-GPU 전송을 위한 PCIe Gen4 x8의 14.2 GB/s, 점대점 통신을 위한 노드 간 네트워크의 42 GB/s, 그리고 26.59 GB/s(로컬 NVMe)에서 1.1 GB/s(공유 파일시스템)까지의 스토리지 시스템. 이러한 측정은 학습 파이프라인이 어디서 느려질지 드러내며 높은 모델 FLOPs 활용도(MFU)를 달성하는 데 필수적입니다.

그러나 원시 대역폭 숫자만으로는 완전한 이야기를 말해주지 않습니다. 현대 학습 시스템은 계산과 통신을 중첩할 수 있어, 효과적으로 컴퓨팅 연산 뒤에 통신 비용을 숨길 수 있습니다. 이 병렬화는 상호 연결이 느릴 때도 병목을 완화하는 데 도움이 됩니다. 처리량을 최대화하기 위해 컴퓨팅과 통신을 중첩하는 자세한 전략은 Ultra-Scale Playbook을 참조하세요.

아래 다이어그램은 모든 벤치마킹된 측정을 단일 뷰로 종합하여, GPU에서 멀어질수록 대역폭이 어떻게 극적으로 감소하는지 보여줍니다:



이제 하드웨어와 소프트웨어 설정에서 병목을 식별하는 방법을 알았으니, 한 단계 더 나아가 몇 달 동안 안정적으로 실행할 수 있는 복원력 있는 시스템을 보장하는 방법을 살펴봅시다.

## 복원력 있는 학습 시스템 구축

빠른 하드웨어를 갖추는 것은 LLM 학습을 위한 좋고 안정적인 인프라를 갖추기 위한 입장권에 불과합니다. 학습 아마추어에서 전문가로 가려면, 원시 속도를 넘어 생각하고 전체 학습 경험을 더 원활하게 만들고 다운타임을 최소화하는 덜 화려하지만 중요한 인프라 조각에 집중해야 합니다.

이 섹션에서는 하드웨어 및 소프트웨어 최적화에서 **프로덕션 준비**로 전환합니다: 불가피한 장애에서 살아남을 만큼 견고하고, 지속적인 감시 없이 실행될 만큼 자동화되고, 문제가 발생했을 때 적응할 만큼 유연한 시스템을 구축합니다.

### 노드 상태 모니터링과 교체

학습에 충분히 빠른 GPU를 갖추는 것이 중요하지만, LLM 학습은 단일 일이 아니라 몇 주 또는 몇 달 동안 실행되므로, 시간에 따른 GPU 상태 추적이 중요해집니다. 초기 벤치마크를 통과한 GPU가 확장된 학습 실행 중에 열 스로틀링, 메모리 오류 또는 성능 저하를 발생시킬 수 있습니다. 이 섹션에서는 이 도전에 어떻게 접근하고 어떤 도구를 사용하는지 공유하겠습니다.

**사전 테스트:** SmolLM3를 시작하기 전에, 여러 도구를 사용하여 포괄적인 GPU 진단을 실행했습니다. 열 스로틀링, 메모리 오류, 성능 이상에 대해 GPU를 스트레스 테스트하는 내부 도구인 [GPU Fryer](#)를 사용했습니다. 또한 GPU 하드웨어 검증, 성능 모니터링, 컴퓨팅, PCIe 연결성, 메모리 무결성, 열 안정성을 다루는 심층 진단 테스트를 통해 장애나 전력 이상의 근본

원인을 식별하는 데 널리 사용되는 도구인 [NVIDIA의 DCGM 진단](#)도 실행했습니다. 이러한 사전 테스트로 학습 중에 문제를 일으켰을 두 개의 문제가 있는 GPU를 잡아냈습니다.

다음 표에서 DCGM 진단 도구로 테스트할 수 있는 것을 볼 수 있습니다:

| 테스트 수준          | 기간    | 소프트웨어 | PCIe + NVLink | GPU 메모리 | 메모리 대역폭 | 진단 | 타겟 스트레스 | 타겟 전력 | NVBandwidth | 메모리 스트레스 | 입력 EDPP |
|-----------------|-------|-------|---------------|---------|---------|----|---------|-------|-------------|----------|---------|
| r1 (Short)      | 초     | ✓ ✓   | ✓             |         |         |    |         |       |             |          |         |
| r2 (Medium)     | < 2분  | ✓ ✓   | ✓             | ✓       | ✓       |    |         |       |             |          |         |
| r3 (Long)       | < 30분 | ✓ ✓   | ✓             | ✓       | ✓ ✓     | ✓  | ✓       |       |             |          |         |
| r4 (Extra Long) | 1-2시간 | ✓ ✓   | ✓             | ✓       | ✓ ✓     | ✓  | ✓       | ✓     | ✓           | ✓        | ✓       |

\*DCGM 진단 실행 수준. 출처:

[NVIDIA DCGM Documentation](#)

```
$ 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 |
| bidirectional 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 -----+
```

**노드 예약:** SmolLM3가 Slurm 관리 클러스터에서 학습되었기 때문에, 전체 실행을 위해 고정된 48노드 예약을 예약했습니다. 이 설정으로 시간에 따라 정확히 동일한 노드의 상태와 성능을 추적할 수 있었고, 논의한 데이터 스토리지 문제를 해결하는 데도 필요했습니다. 또한 예비 노드(자동차의 스페어 휠처럼)를 예약하여 하나가 실패하면 수리를 기다리지 않고 즉시 교체할 수 있었습니다. 유휴 상태일 때 예비 노드는 평가 작업이나 개발 실험을 실행했습니다.

**지속적 모니터링:** 학습 중에 GPU 온도, 메모리 사용량, 컴퓨팅 활용도, 처리량 변동과 같은 모든 노드의 핵심 메트릭을 추적했습니다. [Prometheus](#)를 사용하여 모든 GPU에서 [DCGM](#)메트릭을 수집하고 실시간 모니터링을 위해 [Grafana](#) 대시보드에서 시각화합니다. AWS 인프라에서 GPU 모니터링을 위한 Prometheus와 Grafana 배포에 대한 자세한 설정 지침은 [이 예시 설정 가이드를 참조하세요](#). Slack 봇이 노드가 의심스러운 동작을 보일 때 알림을 보내, 전체 학습 실행이 충돌하기 전에 실패하는 하드웨어를 사전에 교체할 수 있었습니다.

#### 대시보드에 접근

이 다층 접근 방식은 하드웨어 문제가 관리 가능한 중단이 됨을 의미했습니다.

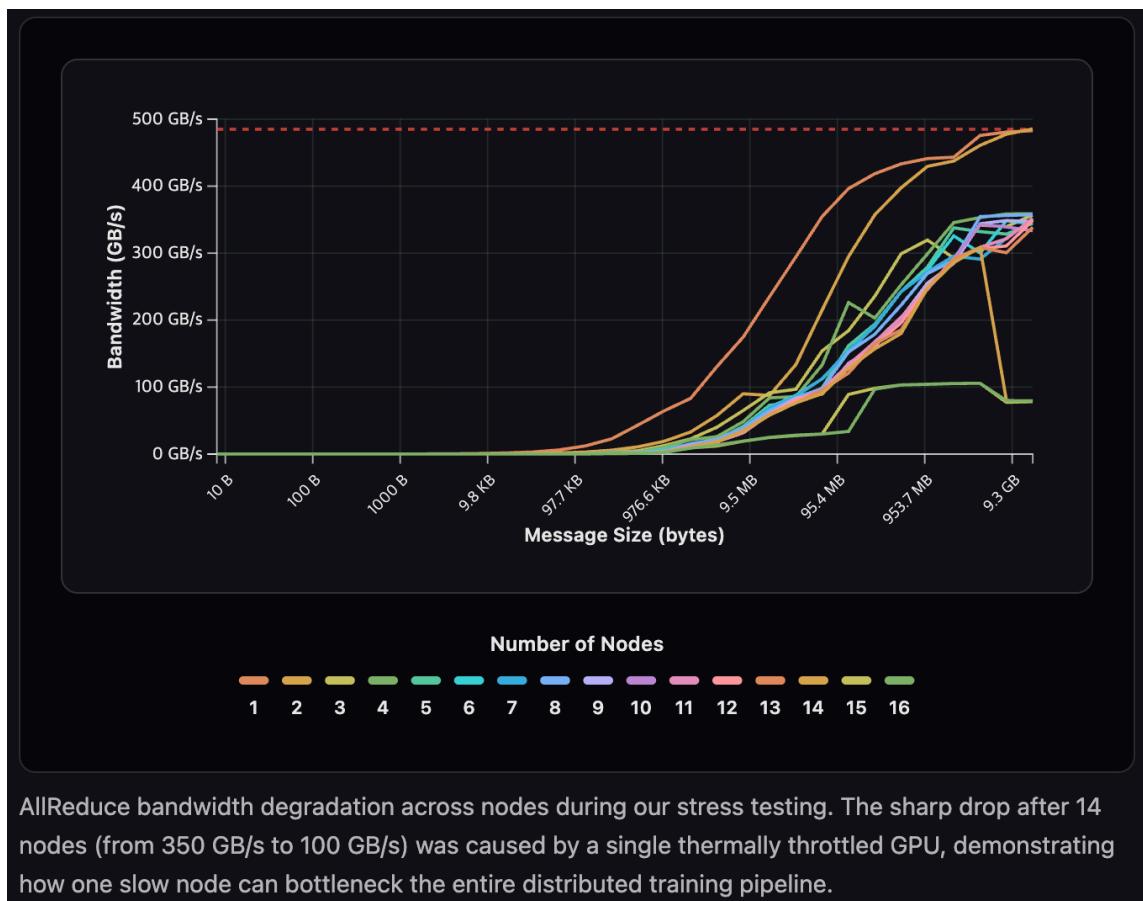
#### 열 현실 점검: GPU가 느려질 때

마케팅 사양은 완벽한 냉각을 가정하지만, 현실은 더 복잡합니다. GPU는 과열되면 자동으로 클럭 속도를 줄여, 잘 설계된 시스템에서도 이론적 최대값 이하로 성능을 낮춥니다.



[NVIDIA의 DCGM](#)에서 `DCGM_FI_DEV_CLOCK_THROTTLE_REASONS` 메트릭을 모니터링하여 열 스로틀링을 감지합니다. 이 메트릭이 0이 아닌 값을 보이면, GPU가 과열로 인해 자동으로 클럭 속도를 줄이고 있는 것입니다. 위의 대시보드는 이러한 스로틀링 이벤트가 실제로 어떻게 나타나는지 보여줍니다.

열 스로틀링은 영향받는 GPU만 해치는 것이 아니라; 전체 분산 학습 설정에 걸쳐 연쇄됩니다. 테스트 중에, 단일 스로틀링 노드가 집합 통신 성능에 어떻게 극적으로 영향을 미칠 수 있는지 관찰했습니다.



위의 차트는 1개에서 16개 노드로 스케일링할 때 AllReduce 대역폭이 저하되는 것을 보여줍니다. 14개 노드 이후의 급격한 하락을 주목하세요. 이전에 보았듯이 대역폭이 300GB/s 이상을 유지할 것으로 예상하지만 350 GB/s에서 100 GB/s로 떨어졌습니다. 이것은 네트워크 문제가 아니었습니다: 열 스로틀링이 있는 단일 노드가 병목이 되어, 그래디언트 동기화 중에 다른 모든 노드가 기다리도록 강제했습니다. 분산 학습에서, 가장 느린 노드만큼만 빠릅니다.

👉 **핵심 교훈:** 장기 학습 실행에 커밋하기 전에, 앞서 언급한 도구를 사용하여 하드웨어를 스트레스 테스트하여 열 및 전력 제한을 식별하세요. DCGM 텔레메트리를 사용하여 온도를 지속적으로 모니터링하고 실제 열 제한을 계획하세요. GPU 클럭이 최대 성능으로 설정되어 있는지 확인하는 것도 좋은 관행입니다. 전력 제약으로 인해 GPU가 광고된 성능을 유지할 수 없는 이유에 대한 더 깊은 탐구는 전력 스로틀링에 대한 [이 훌륭한 분석을 참조하세요](#).

## 체크포인트 관리

체크포인트는 장기 학습 실행 중 안전망입니다. 세 가지 실용적인 이유로 정기적으로 저장합니다: 장애 복구, 평가를 통한 학습 진행 모니터링, 연구를 위해 커뮤니티와 중간 모델 공유. 복구 측면이 가장 중요합니다. 실행이 실패하면, 즉시 재개할 경우 최대 저장 간격만 얇도록 최신 저장된 체크포인트에서 재시작하고 싶습니다(예: 4시간마다 저장하면 4시간의 학습).

### 💡 재개 프로세스를 자동화하세요

재개 프로세스를 자동화하려고 노력하세요. 예를 들어 Slurm에서는 `SBATCH --requeue` 를 사용하면 작업이 최신 체크포인트에서 자동으로 재시작됩니다. 그렇게 하면, 누군가가 장애를 발견하고 수동으로 재시작하기를 기다리는 시간 손실을 피할 수 있습니다.

재개 메커니즘을 구현할 때 염두에 두어야 할 두 가지 중요한 세부 사항이 있습니다:

1. 체크포인트 저장은 학습 처리량에 영향을 미치지 않고 백그라운드에서 발생해야 합니다.
2. 스토리지를 주시하세요. 24일 실행에서 4시간마다 저장하면 ~144개의 체크포인트를 의미합니다. 대형 모델과 옵티마이저 상태로 인해 빠르게 늘어납니다. 우리의 경우, 한 번에 하나의 로컬 체크포인트(최신 저장된 것)만 저장하고 나머지는 클러스터 스토리지가 가득 차는 것을 피하기 위해 S3로 오프로드합니다.

### 과거의 고통스러운 교훈:

첫 번째 대규모 실행(StarCoder 15B) 중에, 학습은 여러 재시작을 통해 순조롭게 진행되었습니다. 마지막 날, 오래된 처리량 테스트에서 스크립트 맨 끝에 남아 있던 `rm -rf $CHECKPOINT_PATH` 명령에 의해 전체 체크포인트 폴더가 삭제되었음을 발견했습니다. 이 파괴적인 명령은 Slurm 작업이 실제로 완료되었을 때만 트리거되었으며, 이전 재시작에서는 발생하지 않았습니다.

다행히도, 전날의 체크포인트가 저장되어 있어서, 하루의 재학습 비용만 들었습니다. 교훈은 명확했습니다: 프로덕션 스크립트에 파괴적인 명령을 남기지 마세요, 그리고 수동 개입에 의존하지 말고 저장 직후에 체크포인트 백업을 자동화하세요.

nanotron 학습에서, 2시간마다 로컬에 체크포인트를 저장하고, 각각을 즉시 S3에 업로드한 다음, 백업이 확인되면 로컬 복사본을 삭제합니다. 재개 시, 최신 체크포인트가 로컬에서 사용할 수 없으면 S3에서 가져옵니다. 이 접근 방식은 스토리지를 절약하고, 백업을 보장하며, 빠른 복구를 가능하게 합니다.

### 자동화된 평가

평가를 수동으로 실행하는 것은 빠르게 병목이 됩니다. 반복적으로 할 때까지는 간단해 보입니다. 벤치마크 실행, 모든 실행에 대한 결과 추적 및 플롯은 상당한 오버헤드로 합산됩니다. 해결책? 미리 모든 것을 자동화하세요.

SmolLM3의 경우, [LightEval](#)을 사용하여 nanotron 체크포인트에서 평가를 실행합니다. 저장된 모든 체크포인트가 클러스터에서 평가 작업을 트리거합니다. 결과는 Weights & Biases 또는 [Trackio](#)로 직접 퓨시되므로, 대시보드를 열고 곡선이 진화하는 것을 지켜보기만 하면 됩니다. 이것은 엄청난 시간을 절약했고 실행 전반에 걸쳐 평가 추적을 일관되게 유지했습니다.

학습 설정에서 한 가지만 자동화할 수 있다면, 평가를 자동화하세요.

마지막으로, 처리량을 최대화하기 위해 학습 레이아웃, 즉 모델이 사용 가능한 GPU에 어떻게 분산되는지를 최적화하는 방법을 살펴봅시다.

## 학습 처리량 최적화

### 얼마나 많은 GPU가 필요한가요?

좋은 질문입니다! 사양과 벤치마크에 대한 모든 이야기 후에도, 실용적인 질문을 파악해야 합니다: 실제로 몇 개의 GPU를 임대하거나 구매해야 할까요?

적절한 GPU 수를 결정하려면 학습 시간, 비용, 스케일링 효율성의 균형을 맞춰야 합니다. 다음은 우리가 사용한 프레임워크입니다:

기본 크기 조정 공식:

$$\text{GPU Count} = \frac{\text{Total FLOPs Required}}{\text{Per-GPU Throughput} \times \text{Target Training Time}}$$

이 공식은 문제를 세 가지 핵심 구성 요소로 분해합니다:

- 필요한 총 FLOPs:** 모델을 학습시키는 데 필요한 계산 작업(모델 크기, 학습 토큰, 아키텍처에 따라 달라짐)
- GPU당 처리량:** 각 GPU가 실제로 제공할 수 있는 초당 FLOPs(이론적 피크가 아님!)
- 목표 학습 시간:** 학습이 완료되기까지 기다릴 의향이 있는 시간

핵심 통찰: 피크 사양이 아닌 **현실적인 처리량**을 추정해야 합니다. 이것은 모델 FLOPs 활용도(MFU): 실제로 달성하는 이론적 피크 성능의 백분율을 고려하는 것을 의미합니다.

SmoILM3의 경우, 계산은 다음과 같았습니다:

- 모델 크기:** 3B 파라미터
- 학습 토큰:** 11조 토큰
- 목표 학습 시간:** ~4주
- 예상 MFU:** 30%(유사한 규모 실험 기반)

먼저, 토큰당 **6N FLOPs**(여기서 N = 파라미터)의 표준 트랜스포머 근사를 사용하여 필요한 총 FLOPs를 계산합니다:

$$\text{Total FLOPs} = 6 \times 3 \times 10^9 \text{ params} \times 11 \times 10^{12} \text{ tokens} = 1.98 \times 10^{23} \text{ FLOPs}$$

예상 MFU 30%로, 유효 GPU당 처리량은:

$$\text{Effective Throughput} = 720 \times 10^{12} \text{ FLOPs/sec} \times 0.30 = 216 \times 10^{12} \text{ FLOPs/sec}$$

이제 크기 조정 공식에 대입:

$$\begin{aligned}\text{GPU Count} &= \frac{1.98 \times 10^{23} \text{ FLOPs}}{216 \times 10^{12} \text{ FLOPs/sec} \times 4 \text{ weeks} \times 604,800 \text{ sec/week}} \\ &= \frac{1.98 \times 10^{23}}{5.23 \times 10^{20}} \approx 379 \text{ GPUs}\end{aligned}$$

이 계산은 375-400개의 H100을 가리켰고, 384개의 H100을 확보했습니다. 이 숫자는 병렬성 전략과 잘 맞았고 노드 장애 및 재시작과 같은 예상치 못한 문제에 대한 약간의 버퍼와 함께 현실적인 4주 일정을 제공했습니다.

## 더 많은 GPU가 항상 더 좋은 것은 아닌 이유: 암달의 법칙 실제 적용

여기 직관에 반하는 진실이 있습니다: 더 많은 GPU를 추가하면 실제로 학습이 더 느려질 수 있습니다. 여기서 암달의 법칙 ([Amdahl's Law](#))이 작용합니다.

암달의 법칙은 병렬화로부터의 속도 향상이 워크로드의 직렬(병렬화할 수 없는) 부분에 의해 근본적으로 제한된다고 말합니다. LLM 학습에서, 이 "직렬" 부분은 주로 통신 오버헤드입니다: GPU 간에 그래디언트/가중치/활성화를 동기화하는 데 소요되는 시간으로, 병렬화할 수 없습니다([여기서 더 읽기](#)).

공식은: 최대 속도 향상 =  $1 / (\text{직렬 비율} + \text{병렬 비율} / \text{프로세서 수})$



SmolLM3의 3B 모델의 경우, 통신이 각 학습 스텝의 10%를 차지한다면, 얼마나 많은 GPU를 추가해도 10배 이상의 속도 향상을 얻을 수 없습니다. 더 나쁜 것은, GPU를 추가할수록 통신 비율이 종종 증가합니다. 이유는:

- 더 많은 GPU = 더 많은 AllReduce 참여자 = 더 긴 동기화
- 네트워크 지연 시간/대역폭이 병목이 됨
- 작은 모델은 컴퓨팅 뒤에 통신을 숨길 수 없음

SmolLM3의 경우, 약한 스케일링 원칙을 사용했습니다: 글로벌 배치 크기가 GPU 수에 따라 스케일링되어, 전역적으로 GPU당 대략 8K 토큰을 유지했습니다. 이것은 처리량을 최대화하면서 통신 대 계산 비율을 합리적으로 유지했습니다.

## 최적의 병렬성 구성 찾기

GPU를 확보했으면, 다음 도전은 실제로 효율적으로 학습하도록 구성하는 것입니다. 이를 위해 병렬성 전략이 중요해집니다.

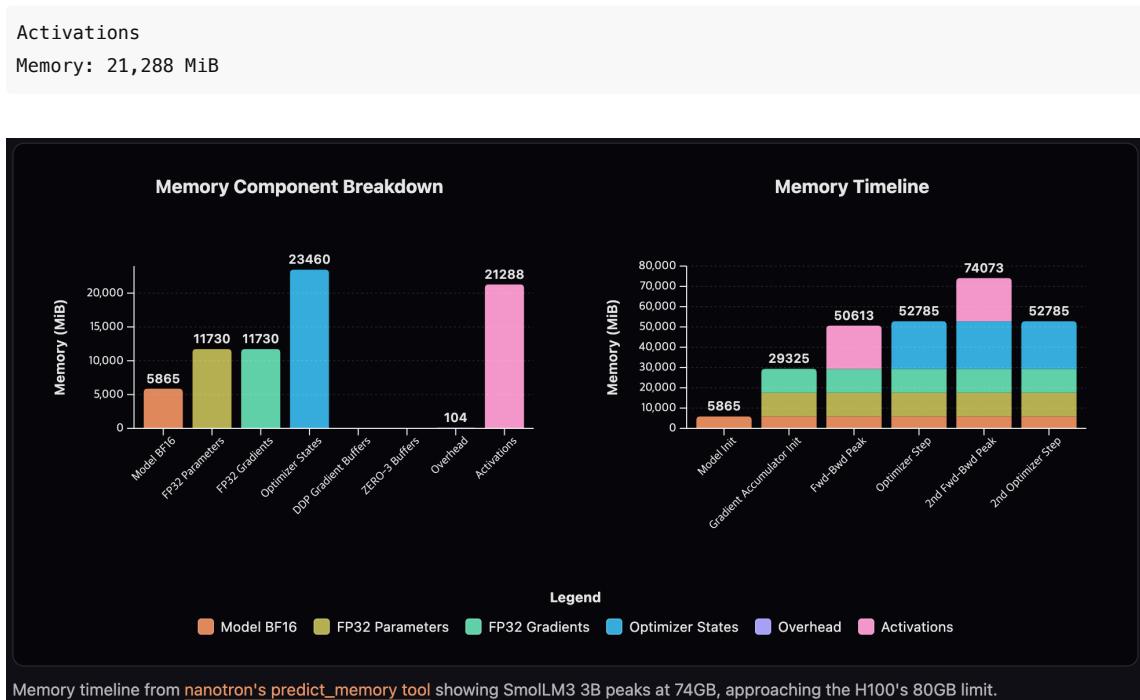
최적의 학습 구성을 찾기 위해 [Ultra-Scale Playbook](#)의 접근 방식을 따릅니다. 플레이북은 문제를 세 가지 순차적 단계로 나눕니다: 먼저 모델이 메모리에 맞는지 확인하고, 그런 다음 목표 배치 크기를 달성하고, 마지막으로 최대 처리량을 위해 최적화합니다. SmolLM3에 이를 어떻게 적용했는지 살펴봅시다.

다양한 병렬성 전략(데이터 병렬성, 텐서 병렬성, 파이프라인 병렬성, ZeRO 등)에 대한 자세한 설명은 다시 한번 *Ultra-Scale Playbook*을 확인하시기 바랍니다.

버니 밍 삽입

## Step 1: 메모리에 학습 스텝 맞추기

첫 번째 질문은 간단합니다: SmoLLM3 3B 모델이 단일 H100의 80GB 메모리에 맞기나 하나요? 이에 답하기 위해, 모델 파라미터, 옵티마이저 상태, 그래디언트, 활성화에 대한 메모리 소비를 추정하는 nanotron의 `predict_memory` 도구를 사용합니다.



결과는 80GB 제한에 가깝게 밀어붙이고 있음을 보여줍니다. 이것은 GPU당 메모리 풋프린트를 줄이는 어떤 형태의 병렬성이 필요하다는 것을 의미합니다. 텐서 병렬성(GPU 간에 모델 레이어 분할), 파이프라인 병렬성(GPU 간에 모델 깊이 분할), 또는 ZeRO 옵티마이저 샤딩(옵티마이저 상태 분산)이든. 이러한 전략 중 적어도 하나 없이는 효율적으로 또는 전혀 학습할 수 없습니다.

## Step 2: 목표 글로벌 배치 크기 달성

이제 모델이 어떤 형태의 병렬성으로 메모리에 맞는다는 것을 알았으니, 약 200만 토큰의 목표 글로벌 배치 크기(GBS)를 달성하는 방법을 결정해야 합니다. 이 제약은 첫 번째 방정식을 제공합니다:

$$\text{GBS} = \text{DP} \times \text{MBS} \times \text{GRAD\_ACC} \times \text{SEQLEN} \approx 2\text{M tokens}$$

여기서:

- **DP (데이터 병렬성):** 데이터 병렬 복제본 수

- **MBS (マイクロ バッチ サイズ)**: マイクロ バッチ당 GPU당 처리되는 토큰
- **GRAD\_ACC (그래디언트 누적)**: 옵티마이저 스텝 전 순방향-역방향 수
- **SEQLEN (시퀀스 길이)**: 시퀀스당 토큰(1차 사전학습 단계에서 4096)

384개의 H100에서 하드웨어 제약도 있습니다:

$$DP \times TP \times PP = 384 = 2^7 \times 3$$

여기서:

- **TP (텐서 병렬성)**: 모델 레이어당 GPU(가중치 행렬 분할)
- **PP (파이프라인 병렬성)**: 모델 깊이당 GPU(레이어를 수직으로 분할)

이 두 방정식은 탐색 공간을 정의합니다. 학습 처리량을 최대화하면서 두 제약을 모두 만족하는 값을 찾아야 합니다.

### Step 3: 학습 처리량 최적화

제약이 확립되었으므로, 학습 처리량을 최대화하는 병렬성 구성을 찾아야 합니다. 탐색 공간은 하드웨어 토플로지와 모델 아키텍처에 의해 정의됩니다.

하드웨어 설정은 위 섹션에서 보았듯이 두 가지 뚜렷한 유형의 상호 연결을 제공합니다: 노드 내 통신을 위한 NVLink(900 GB/s)와 노드 간 통신을 위한 EFA(~50 GB/s). 이 토플로지는 자연스럽게 네트워크 특성에 맞추기 위해 적어도 두 가지 형태의 병렬성을 사용하는 것을 제안합니다. 이러한 상호 연결 간의 극적인 대역폭 차이는 어떤 병렬성 전략이 가장 잘 작동하는지에 크게 영향을 미칩니다.

모델 관점에서, SmolLM3의 아키텍처는 옵션을 제한합니다. Mixture-of-Experts 아키텍처를 사용하지 않으므로 전문가 병렬성이 필요하지 않습니다. 마찬가지로, 1단계에서 4096 시퀀스 길이로 학습하므로 컨텍스트 병렬성이 필요하지 않습니다. 이것은 탐색할 세 가지 주요 병렬성 차원을 남깁니다: 데이터 병렬성(DP), 텐서 병렬성(TP), 파이프라인 병렬성(PP).

Step 2의 제약을 고려하여, 여러 파라미터를 스윕해야 합니다:

- **ZeRO 변형이 있는 DP(ZeRO-0, ZeRO-1, ZeRO-3)**: 1에서 384까지의 값, 2 및/또는 3의 배수로 제한
- **TP(1, 2, 3, 4, 6, 8)**: NVLink의 높은 대역폭을 완전히 활용하기 위해 단일 노드 내에서 유지
- **PP(1..48)**: GPU 간에 모델 깊이 분할
- **MBS(2, 3, 4, 5)**: 병렬성에서의 메모리 절약에 따라, Tensor 코어를 더 잘 활용하기 위해 MBS를 늘릴 수 있음
- **활성화 체크포인팅(none, selective, full)**: 메모리와 통신 감소를 위해 추가 컴퓨팅 교환
- **커널 최적화**: 사용 가능한 곳에서 CUDA 그래프와 최적화된 커널

이것이 압도적인 수의 조합처럼 보일 수 있지만, 실용적인 접근 방식은 먼저 각 차원을 독립적으로 벤치마킹한 다음, 처리량을 상당히 저해하는 구성을 제거하는 것입니다. 핵심 통찰은 모든 병렬성 전략이 동등하게 생성되지 않는다는 것입니다. 일부는 특히 우리 규모에서 이점을 훨씬 능가하는 통신 오버헤드를 도입합니다.

우리의 경우, 파이프라인 병렬성은 좋지 않은 성능 특성을 보였습니다. PP는 노드 간에 빈번한 파이프라인 버블 동기화를 필요로 하며, 상대적으로 작은 3B 모델로 인해 통신 오버헤드가 잠재적 이점을 지배했습니다. 또한, 파이프라인 버블을 완전히 제거할 수 있는 고효율 PP 스케줄에 접근할 수 없어 PP의 실행 가능성이 더욱 제한되었습니다. 마찬가지로, 0 이상의 ZeRO 수준은 메모리에 도움이 되는 것보다 처리량을 더 저해하는 상당한 all-gather와 reduce-scatter 연산을 도입했습니다.

니다. 이러한 초기 벤치마크를 통해 탐색 공간을 극적으로 좁힐 수 있었고, 데이터 병렬성과 적당한 텐서 병렬성을 결합한 구성에 집중했습니다.

👉 각 구성을 평가하기 위해, 5번의 반복에 대해 벤치마크를 실행하고 \*\*GPU당 초당 토큰(tok/s/gpu)\*\*을 기록합니다. 이것이 궁극적으로 우리가 관심 있는 메트릭입니다. Weights & Biases와 Trackio를 사용하여 처리량과 구성을 로깅하여, 다양한 병렬성 전략을 쉽게 비교합니다.

nanotron에서 사용 가능한 옵션을 체계적으로 벤치마킹한 후, **DP = 192**로 정착했습니다. 이것은 데이터 병렬 그래디언트 동기화를 위해 노드 간 EFA 대역폭을 활용합니다. 이것은 192개의 독립적인 모델 복제본을 의미하며, 각각 다른 데이터 배치를 처리합니다. 텐서 병렬성의 경우, NVLink의 높은 대역폭을 완전히 활용하기 위해 텐서 병렬 통신을 단일 노드 내에서 유지하는 **TP = 2**를 선택했습니다. 이것은 각 레이어의 가중치 행렬을 두 개의 GPU에 분할하여, 순방향 및 역방향 패스에 빠른 통신이 필요합니다.

**マイクロ 배치 크기 = 3**은 메모리 사용량과 컴퓨팅 효율성 사이의 균형을 맞춥니다. 더 큰 배치 크기가 Tensor 코어를 더 잘 활용하겠지만, 이미 메모리 제한에 가깝게 밀어붙이고 있습니다. 마지막으로, 옵티마이저 상태 샤딩이 없는 **ZeRO-0**을 선택했습니다. ZeRO-1이나 ZeRO-3가 메모리 풋프린트를 줄일 수 있지만, 384개의 GPU에 걸쳐 옵티마이저 상태를 수집하고 분산하는 통신 오버헤드가 처리량을 상당히 저해할 것입니다.

이러한 병렬성 결정 중 많은 부분이 실험 당시 라이브러리 상태에 영향을 받았습니다. 예를 들어, nanotron은 아직 ZeRO-3를 지원하지 않았고, 파이프라인 버블을 제거할 수 있는 고도로 최적화된 파이프라인 병렬성 스케줄에 접근할 수 없었습니다. 프레임워크가 발전함에 따라, 이러한 트레이드오프 중 일부가 변할 수 있습니다. 기억은 항상 환영합니다!

이 구성은 384개의 H100 클러스터에서 처리량을 최대화하면서 약 200만 토큰의 목표 글로벌 배치 크기를 달성합니다 ( $192 \times 3 \times 1 \times 4096 \approx 2.3M$ ). 전체 학습 구성은 stage1\_8T.yaml에서 볼 수 있습니다.

## 결론

우리는 간단한 질문으로 이 여정을 시작했습니다: 2025년에 고성능 LLM을 학습시키는 데 실제로 무엇이 필요한가요? 사전학습에서 후속 학습까지 전체 파이프라인을 걸어오면서, 기술뿐만 아니라 그것들이 작동하게 만드는 방법론을 보여드렸습니다.

**대규모 사전학습.** 학습을 할지 말지 결정하기 위한 Training Compass 프레임워크를 살펴본 다음, 목표를 구체적인 아키텍처 결정으로 변환하는 방법을 보여드렸습니다. 신뢰할 수 있는 절제 파이프라인을 설정하고, 변경 사항을 개별적으로 테스트하고, 수십억 토큰 실험에서 수조 토큰 실행으로 스케일링하는 방법을 보셨습니다. 규모에서 나타날 수 있는 인프라 도전(처리량 불안, 데이터로더 병목, 미묘한 버그)과 모니터링과 체계적인 위험 제거가 어떻게 조기에 잡아내고 빠르게 디버그하는 데 도움이 되는지 문서화했습니다.

**실제 후속 학습.** 베이스 모델에서 프로덕션 어시스턴트로 가는 것이 자체적인 체계적 접근을 필요로 함을 보여드렸습니다: 무엇이든 학습하기 전에 평가를 확립하고, SFT 데이터 혼합을 반복하고, 선호도 최적화를 적용하고, 선택적으로 RL로 더 밀어붙이는 것. 바이브 테스트가 메트릭이 놓치는 버그를 잡는 방법, 채팅 템플릿이 어떻게 조용히 지시 따르기를 깨뜨릴 수 있는지, 왜 데이터 혼합 균형이 사전학습에서만큼 후속 학습에서도 중요한지 보셨습니다.

두 단계 전반에 걸쳐, 동일한 핵심 통찰로 계속 돌아왔습니다: 실험을 통해 모든 것을 검증하고, 한 번에 한 가지만 변경하고, 규모가 새로운 방식으로 것들을 깨뜨릴 것을 예상하고, 모든 새 논문을 쫓기보다 사용 사례가 결정을 이끌도록 하세요. 이 과정을 따라, SmoLLM3를 학습시켰습니다: 긴 컨텍스트를 가진 경쟁력 있는 3B 다국어 추론기. 그 과정에서, 무엇이 작동하고, 무엇이 깨지고, 문제가 발생했을 때 어떻게 디버그하는지에 대해 많이 배웠습니다. 성공과 실패 모두를 문서화하려고 노력했습니다.

### 다음은 무엇인가요?

이 블로그는 현대 LLM 학습의 기본을 다루지만, 분야는 빠르게 발전합니다. 더 깊이 들어가는 방법은 다음과 같습니다:

- **직접 실험을 실행하세요.** 절제에 대해 읽는 것은 유용합니다; 직접 실행하면 실제로 무엇이 중요한지 배웁니다. 작은 모델을 선택하고, 평가를 설정하고, 실험을 시작하세요.
- **소스 코드를 읽으세요.** nanotron, TRL 등과 같은 학습 프레임워크는 오픈 소스입니다. 구현을 이해하면 논문이 대충 넘어가는 세부 사항이 드러납니다.
- **최근 작업을 따르세요.** 최근 최첨단 모델의 논문은 분야가 어디로 향하는지 보여줍니다. 참고문헌 섹션에는 영향력 있는 논문과 리소스의 큐레이션된 목록이 있습니다.

이 블로그가 프론티어를 밀어붙이는 대형 랩에 있든 특정 문제를 해결하는 소규모 팀에 있든, 명확성과 자신감을 가지고 다음 학습 프로젝트에 접근하는 데 도움이 되기를 바랍니다.

이제 무언가를 학습하려 가세요. 그리고 새벽 2시에 손실이 미스터리하게 스파이크할 때, 기억하세요: 모든 위대한 모델에는 그 뒤에 디버깅 이야기가 있습니다. 오픈 소스와 오픈 사이언스의 힘이 항상 여러분과 함께하기를!

## 감사의 글

귀중한 피드백을 주신 Guilherme, Hugo, Mario에게 감사드리고, Trackio 기능에 도움을 주신 Abubakar에게 감사드립니다.

귀한 경험을 공유해준 허깅페이스팀에 다시한번 감사드립니다!

## 참고문헌

아래는 LLM 학습 여정에서 가장 많은 정보를 제공한 논문, 책, 블로그 포스트의 큐레이션된 목록입니다.

### LLM 아키텍처

- Dense 모델: Llama3, Olmo2, MobileLLM
- MoEs: DeepSeek V2, DeepSeek V3, Scaling Laws of Efficient MoEs
- 하이브리드: MiniMax-01, Mamba2

### 옵티마이저 및 학습 파라미터

- Muon is Scalable for LLM Training, Fantastic pretraining optimisers
- Large Batch Training, DeepSeekLLM

## 데이터 큐레이션

- 웹: FineWeb & FineWeb-Edu, FineWeb2, DCLM
- 코드: The Stack v2, To Code or Not to Code
- 수학: DeepSeekMath, FineMath, MegaMath
- 데이터 훈합: SmolLM2, Does your data spark joy

## 스케일링 법칙

- Kaplan, Chinchilla, Scaling Data-Constrained Language Models

## 후속 학습

- InstructGPT:** 베이스 모델을 도움이 되는 어시스턴트로 바꾸는 OpenAI의 기초 논문. ChatGPT의 전신이자 인류가 카르다소프 척도를 올라가는 핵심 단계.
- Llama 2 & 3:** Llama 모델 뒤의 학습에 대한 Meta의 매우 상세한 기술 보고서(평화롭게 쉬시길). 각각 인간 선호도와 모델 평가 모두에 대한 인간 데이터 수집에 대한 많은 통찰을 담고 있습니다.
- Secrets of RLHF in LLMs, Part I & II:** 이 논문들은 RLHF의 세부 사항, 특히 강력한 보상 모델을 학습시키는 방법에 대한 많은 보물을 담고 있습니다.
- Direct Preference Optimisation:** 모든 사람에게 LLM으로 RL을 그만두라고 설득한 2023년의 획기적인 논문.
- DeepSeek-R1:** 모든 사람에게 LLM으로 RL을 시작하라고 설득한 2025년의 획기적인 논문.
- Dr. GRPO:** GRPO의 내재된 편향을 이해하고 수정하는 방법에 대한 가장 중요한 논문 중 하나.
- DAPO:** Bytedance가 커뮤니티를 위해 안정적인 R1-Zero 같은 학습을 가능하게 하는 많은 구현 세부 사항을 공유.
- ScaleRL:** RL에 대한 스케일링 법칙을 도출하기 위한 Meta의 거대한 플렉스. 여러 차수의 컴퓨팅에 걸쳐 안정적으로 스케일링되는 학습 레시피를 확립하기 위해 400k GPU 시간 이상을 소모.
- LoRA without Regret:** 저랭크 LoRA로 RL이 전체 파인튜닝과 일치할 수 있다는 것을 발견한 아름답게 쓰인 블로그 포스트(가장 놀라운 결과).
- Command A:** LLM을 효과적으로 후속 학습시키는 다양한 전략에 대한 Cohere의 놀랍도록 상세한 기술 보고서.

## 인프라

- UltraScale Playbook
- Jax scaling book
- Modal GPU Glossary

## 학습 프레임워크

- Megatron-LM
- DeepSpeed
- TorchTITAN
- Nanotron
- NanoChat

- TRL

## 평가

- The LLM Evaluation Guidebook
- OLMES
- FineTasks
- Lessons from the trenches