Natural Language Processing with Deep Learning

CS224N/Ling284

Yejin Choi

Lecture 14: Reasoning 2/2

Announcement

- A4 is due this Thursday. We highly recommend to start working on it **now**, since it involves querying APIs. If everyone is working on it all right before the deadline, we'll get rate limit issues.

- The **Project Milestone** instructions are out now, and we will be doing our best to get the Project Proposals graded with feedback tonight/tomorrow morning.

Lecture Plan

Speculative decoding (20 mins)

Off-policy drift & on-policy distillation (20 mins)

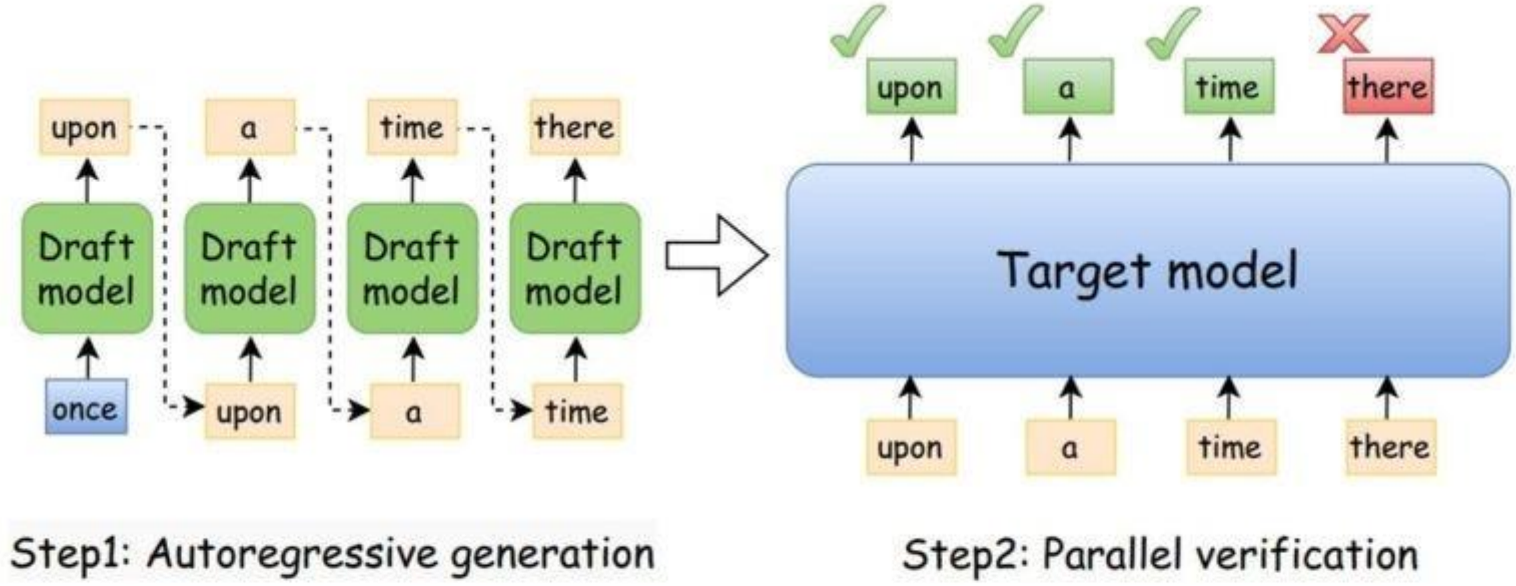
- Off-policy, on-policy, online RL, off-line RL
- RL infra and off-policy drift
- On-policy distillation

Long context extension (25 mins)

Inference-time scaling (15 min)

Speculative decoding

- Problem: Generating with a large LM takes a long time
  Intuition: Not all tokens are equally hard to generate!



- Idea: Use a generation from small LM to assist large LM generation
  - Same idea independently proposed from Google Research (Leviathan et al., Nov 2022) and DeepMind (Chen et al., Feb 2023)

Image credit: https://medium.com/@genai.works/speed-up-llm-inference-with-speculative-decoding-1fc79701e9d6

Speculative decoding

- First, sample a draft of length K (= 5 in this example) from a small LM M_p

$$y_1 \sim p(\cdot\,|x), y_2 \sim p(\cdot\,|x, y_1), \cdots, y_5 \sim p(\cdot\,|x, y_1, y_2, y_3, y_4)$$

- Then, compute the token distribution at each time step with a large target LM M_q

$$q(\cdot\,|x), q(\cdot\,|x, y_1), q(\cdot\,|x, y_1, y_2), \cdots, q(\cdot\,|x, y_1, \cdots, y_5)$$

  - Note: This can be computed in a *single forward pass* of M_q) Why?)

- Let's denote $p_i = p(\cdot\,|x, y_1, \cdots, y_{i-1})$ and $q_i = q(\cdot\,|x, y_1, \cdots y_{i-1})$
  e.g., $q_2 = q(\cdot\,|x, y_1)$,i.e. next token distribution predicted by the target model M_q , when given x and y_1

Speculative decoding

- Now, we can compare the probability of each token assigned by draft model M_p and target model M_q

		y_1	y_2	y_3	y_4	y_5
		dogs	love	chasing	after	cars
Draft model (1B)	p_i	0.8	0.7	0.9	0.8	0.7
Target model (100B)	q_i	0.9	0.8	0.8	0.3	0.8

- Starting from y_1, decide whether to accept the tokens generated by the draft model.

Speculative decoding

- Now, we can compare the probability of each token assigned by draft model M_p and target model M_q

		y_1	y_2	y_3	y_4	y_5
		dogs	love	chasing	after	cars
Draft model (1B)	p_i	0.8	0.7	0.9	0.8	0.7
Target model (100B)	q_i	0.9	0.8	0.8	0.3	0.8

- Starting from y_1 ,decide whether to accept the tokens generated by the draft model.
- Case 1: $q_i \geq p_i$
  The target model (100B) likes this token, even more than the draft model.
  => Accept this token!

Generation after step 1:
dogs

Speculative decoding

- Now, we can compare the probability of each token assigned by draft model M_p and target model M_q

		y_1	y_2	y_3	y_4	y_5
		dogs	love	chasing	after	cars
Draft model (1B)	p_i	0.8	0.7	0.9	0.8	0.7
Target model (100B)	q_i	0.9	0.8	0.8	0.3	0.8

- Starting from y_1 ,decide whether to accept the tokens generated by the draft model.
- Case 2: $q_i < p_i$) accept)
  Target model doesn't like this token as much as the draft model...
  => Accept it with the probability $\frac{q_i}{p_i}$

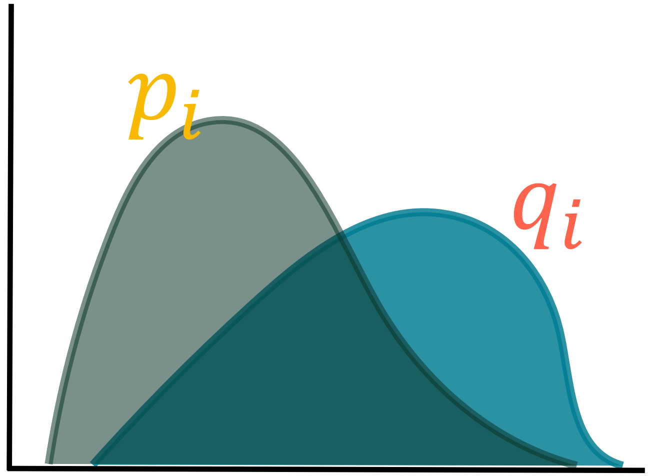
Generation after step 3:
dogs love chasing
(assuming we accepted chasing w/ prob 0.8/09)

Speculative decoding

- Now, we can compare the probability of each token assigned by draft model M_p and target model M_q

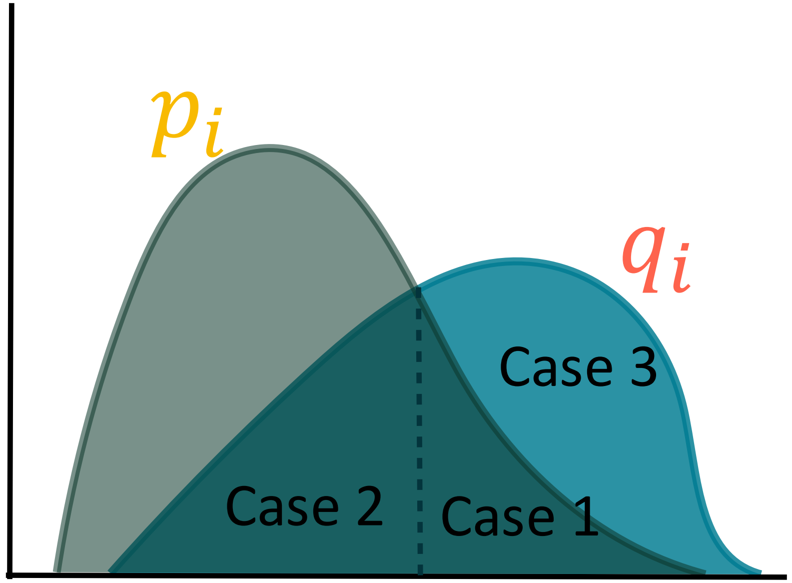
		y_1	y_2	y_3	y_4	y_5
		dogs	love	chasing	after	cars
Draft model (1B)	p_i	0.8	0.7	0.9	0.8	0.7
Target model (100B)	q_i	0.9	0.8	0.8	0.3	0.8

- Starting from y_1 ,decide whether to accept the tokens generated by the draft model.
- Case 3: $q_i < p_i$) reject)
  If q_i >>> p_i ,we likely would have rejected it.
  In this case, we sample a new token from target model
  => Specifically, we sample from $(q_i - p_i)_+$



Speculative decoding

- But why specifically $(q_i - p_i)_+$?
  - because our goal: to cover target LM distribution q_i
- Case 1: $q_i \geq p_i$
  Accept this token.
- Case 2: $q_i < p_i$) accept)
  Accept it with the probability $\frac{q_i}{p_i}$
- Case 3: $q_i < p_i$) reject)
  If q_i >>> p_i ,we likely would have rejected it.
  In this case, we sample a new token from target model
  => Specifically, we sample from $(q_i - p_i)_+$



Note: This sampling procedure,
though sampling from small LM (p_i), has
the same effect as sampling from target LM (q_i).
Formal proof in Appendix I of (Chen et al., 2023)

Speculative decoding

- Speculative sampling is a form of rejection sampling.
  - To sample from an easy-to-sample distribution p (small LM), in order to ample from a more complex distribution q (large LM).
- Using 60M LM (T5-small) as a draft model and 11B (T5-XXL) LM as a target model, we get 2~3x acceleration with identical outputs!

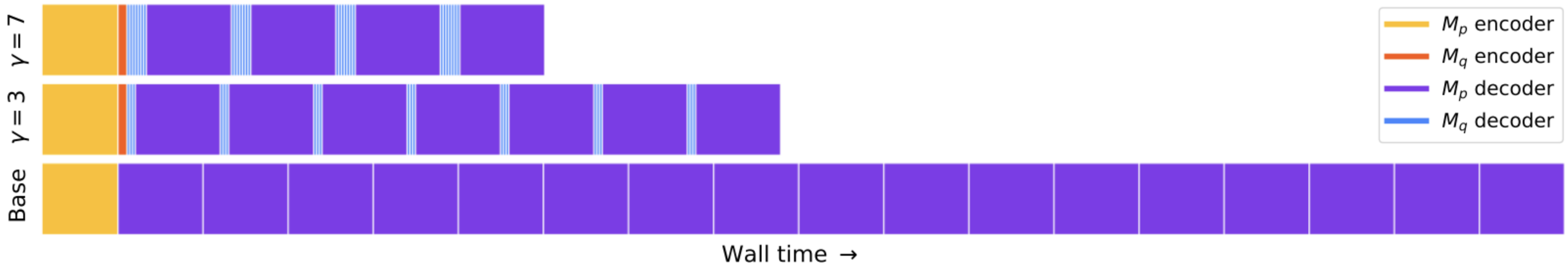


Figure 5. A simplified trace diagram for a full encoder-decoder Transformer stack. The top row shows speculative decoding with $\gamma = 7$ so each of the calls to M_p (the purple blocks) is preceded by 7 calls to M_q (the blue blocks). The yellow block on the left is the call to the encoder for M_p and the orange block is the call to the encoder for M_q. Likewise the middle row shows speculative decoding with $\gamma = 3$, and the bottom row shows standard decoding.

Dynamic speculative decoding

- adaptively adjusts the "lookahead" size (the number of candidate tokens) at each iteration by using a lightweight classifier or confidence threshold to decide on-the-fly whether the draft model should continue drafting or switch to the target model for verification.
- See more: Mamou et al., 2024 & https://huggingface.co/blog/dynamic_speculation_lookahead

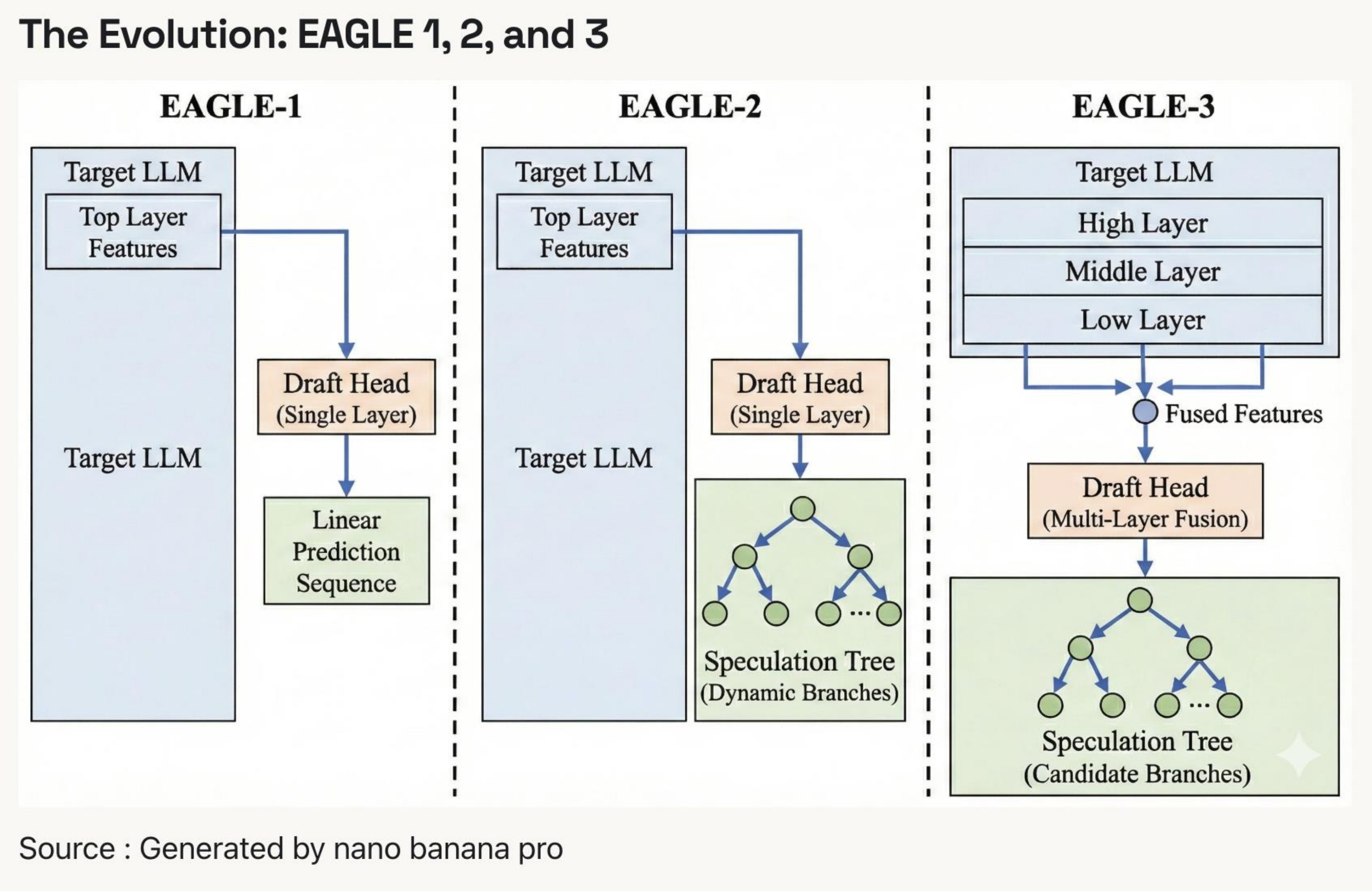
Universal speculative decoding

- Original spec decoding requires identical tokenization between draft and target models
- Models with heterogeneous tokenization can be supported via re-encoding and alignment techniques.
- See more: https://huggingface.co/blog/universal_assisted_generation

Even better speculative decoding algorithms are rapidly developed, and all your favorite inference engines support several options!

Algorithm	vLLM	TRT-LLM	HF TGI	HF Transformers	SGLang
EAGLE-3	✅ (Native)	✅ (Opt)	❌	⚠️ (Manual head)	✅
SuffixDecoding	✅ (Arctic)	❌	❌	❌	✅ (Beta)
Medusa	✅	✅	✅ (SOTA)	✅ (Pipeline)	✅
Draft-Target	✅	✅	✅	✅ (Universal)	✅
DFlash (2026)	⚠️ (Fork)	✅ (Custom)	❌	❌	⚠️ (Fork)

EAGLE-3 (Extrapolation Algo for Greater LM Efficiency)



Key ideas:
Eagle-1: let the draft model "read the mind" of the target model by sneaking into its internal representation ("features-level" speculation)

Eagle-2: context-aware dynamic draft trees!

Eagle-3: "fused features" across different layers

Image credit: https://docs.jarvislabs.ai/blog/speculative-decoding-vllm-faster-llm-inference & nano banana pro

EAGLE-3 (Extrapolation Algo for Greater LM Efficiency)

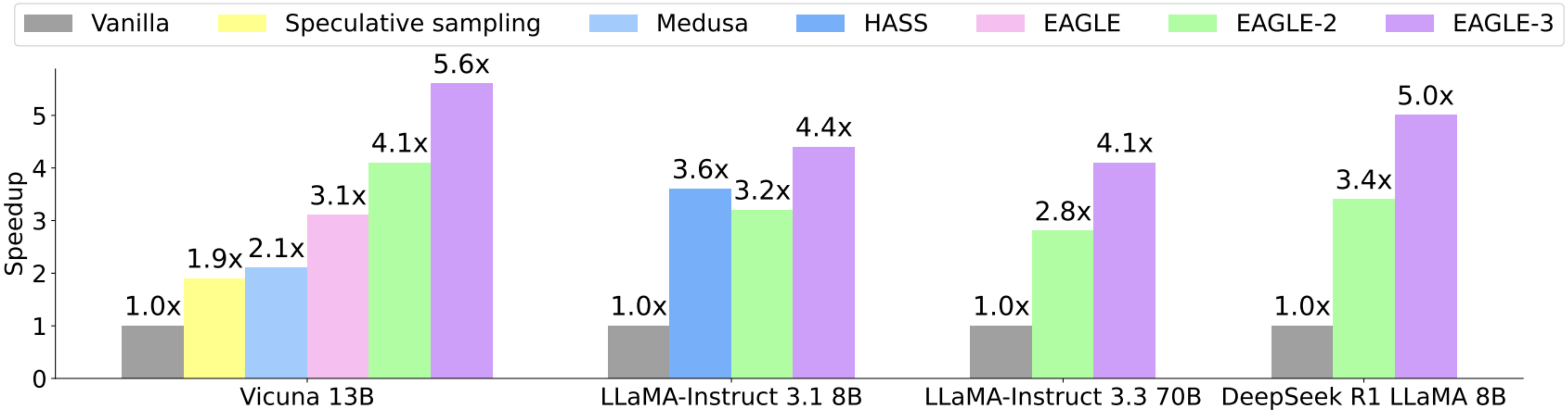


Figure 2: Speedup ratios of different methods at temperature=0. For the standard speculative sampling, Vicuna-13B uses Vicuna-68M as the draft model. In Table 1, we present comparisons with additional methods, but this figure only showcases a subset. Chat model's evaluation dataset is MT-bench, and the reasoning model's evaluation dataset is GSM8K. DeepSeek R1 LLaMA 8B refers to DeepSeek-R1-Distill-LLaMA 8B.

Suffix Decoding: Extreme Speculative Decoding (Oliaro et al., 2025)

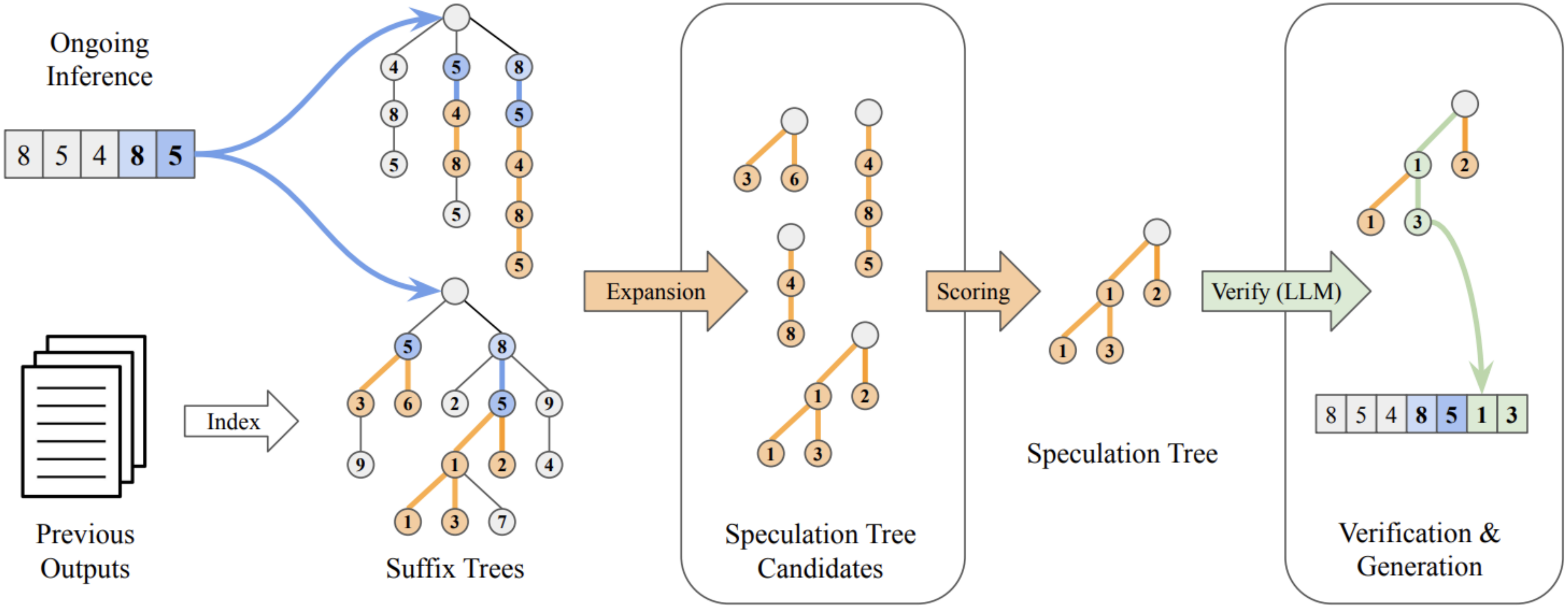
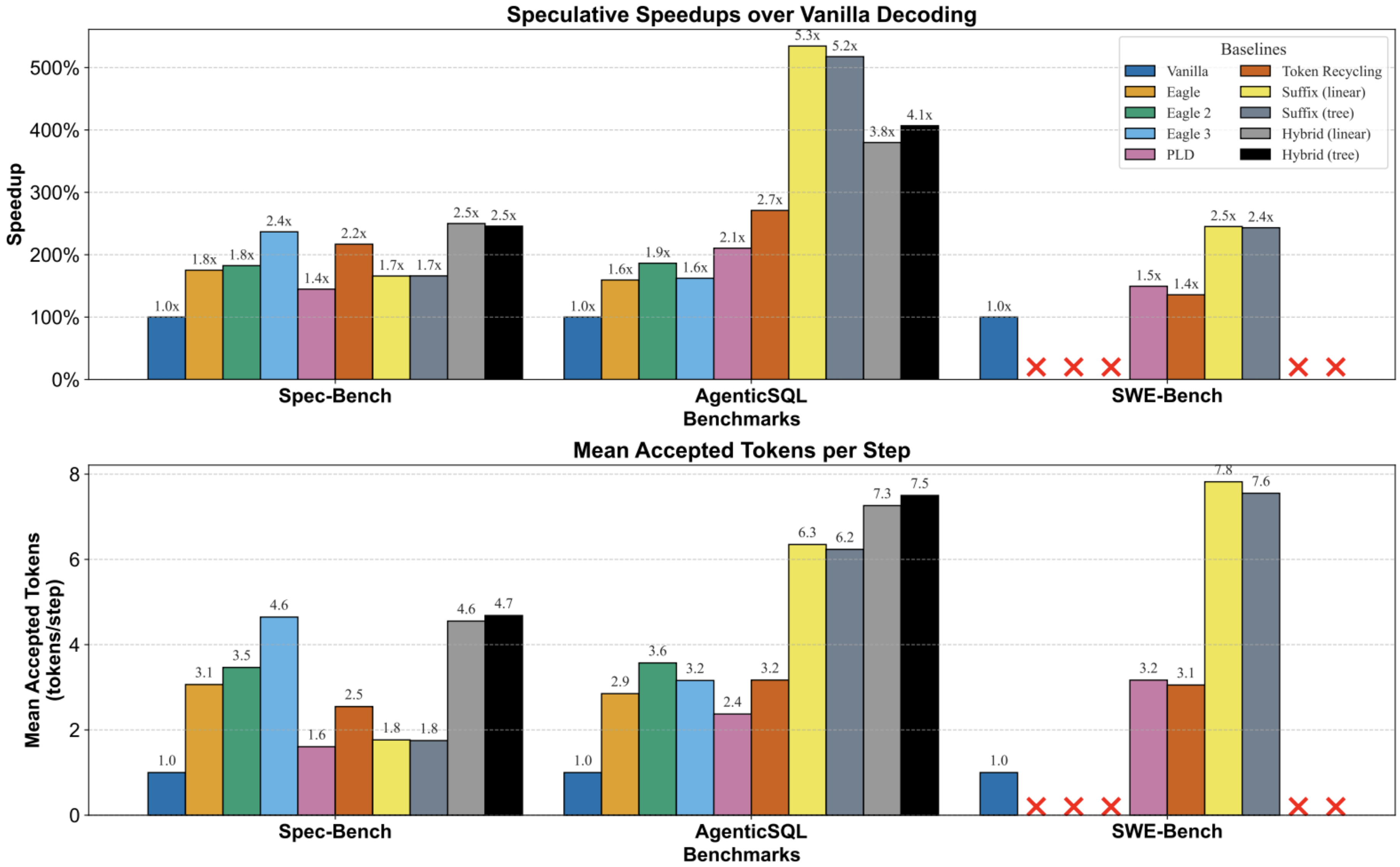


Figure 1: Overview of SuffixDecoding's algorithm. Two suffix trees track ongoing inference (top-left) and previous outputs (bottom-left). SuffixDecoding uses these trees to find matching patterns based on recently generated tokens. It constructs a speculation tree (middle) by selecting the most likely continuations, scoring them based on frequency statistics. Finally, the best candidate is verified by the LLM in a single forward pass (right), with accepted tokens (shown in green) being added to the output and used for the next round of speculation.

Suffix Decoding: Extreme Speculative Decoding (Oliaro et al., 2025)



Suffix Decoding vs Eagle-3

Feature	SuffixDecoding (2025/2026)	EAGLE-3 (2025/2026)
Mechanism	**Model-Free:** Uses a Suffix Tree to cache and match repetitive sequences in the prompt and past outputs.	**Model-Based:** Uses a small, trained Transformer head to predict future hidden features of the target model.
Compute Location	**CPU-Bound:** Runs speculation on the CPU while the GPU handles the target model's verification.	**GPU-Bound:** The draft head runs on the GPU, sharing VRAM with the main model.
Drafting Speed	~20 μs per token (Ultra-fast).	Slower (requires a GPU forward pass).
Best Performance	**Highly repetitive tasks** (Coding, Agentic loops, RAG, SQL generation).	**Open-ended tasks** (Creative writing, general chat, unpredictable reasoning).
Training Needs	Zero. It is a "plug-and-play" data structure.	Requires training a small auxiliary head on the target model’s feature space.

Even better speculative decoding algorithms are rapidly developed, and all your favorite inference engines support several options!

Algorithm	vLLM	TRT-LLM	HF TGI	HF Transformers	SGLang
EAGLE-3	✅ (Native)	✅ (Opt)	❌	⚠️ (Manual head)	✅
SuffixDecoding	✅ (Arctic)	❌	❌	❌	✅ (Beta)
Medusa	✅	✅	✅ (SOTA)	✅ (Pipeline)	✅
Draft-Target	✅	✅	✅	✅ (Universal)	✅
DFlash (2026)	⚠️ (Fork)	✅ (Custom)	❌	❌	⚠️ (Fork)

Lecture Plan

Speculative decoding (20 mins)

Off-policy drift & on-policy distillation (20 mins)

- Off-policy, on-policy, online RL, off-line RL
- RL infra and off-policy drift
- On-policy distillation

Long context extension (25 mins)

Inference-time scaling (15 min)

Online RL: The agents can interact with the env during training

Offline RL (Batch RL): Learning happens strictly from a pre-recorded dataset (human logs or previous agents). The agent cannot "explore" or test new actions.

On-Policy RL: The data used for training is generated exactly by the *current* policy. Once the policy updates, old data from the old policy is discarded as "stale."

REINFORCE, PPO, GRPO

Off-Policy RL: The agent learns from data generated by any policy (older policy, different policy, or even humans). It saves experiences in a *Replay Buffer* for repeated use.

DQN (Deep Q-Network)

Online vs **Offline RL**: whether the agent can interact with the environment

On-policy vs **Off-policy RL**: whether rollouts for training are from the up-to-date policy

Offline On-Policy RL?

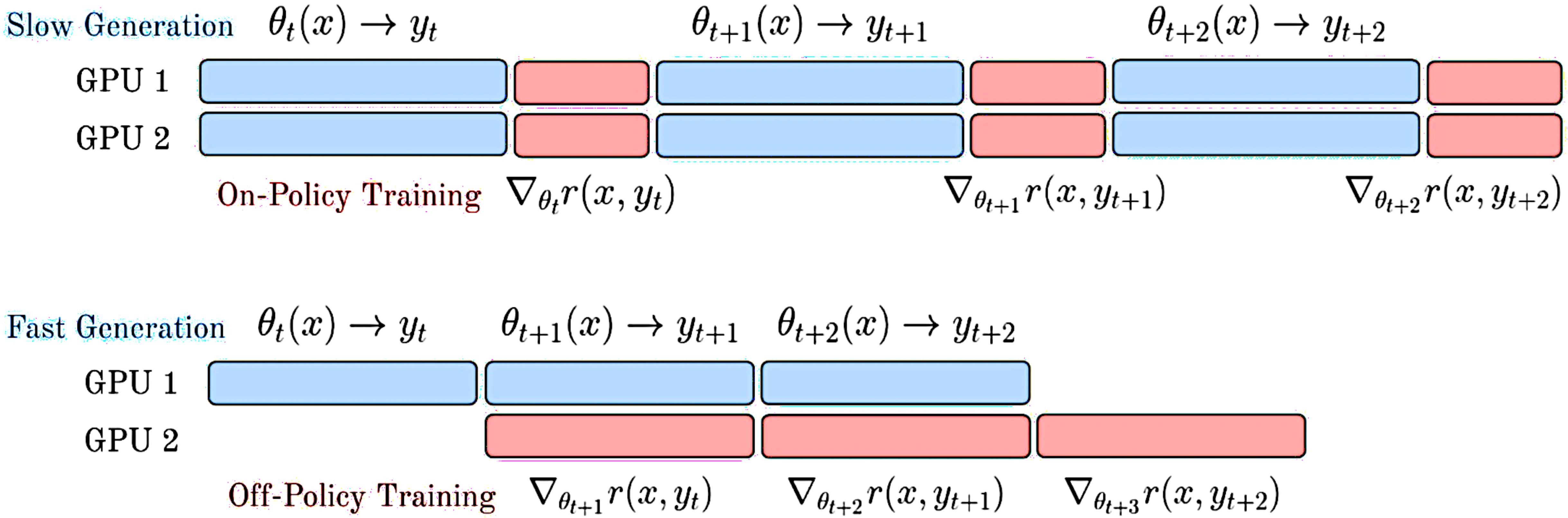
- no true offline on-policy RL (unless the interpretation is somewhat stretched…)

Online Off-Policy RL?

- This is very common. An agent interacts with a simulator (Online) but stores everything in a replay buffer to learn from later (Off-Policy), like **DQN**.
- Asynchronous RL
- PPO with "off-policy drift" due to RL infra optimization

Asynchronous RLHF (Noukhovitch et al., 2024)

- "Why": Classic RLHF is synchronous, which wastes GPU throughput



PipelineRL: in-flight weight updates (Piché et al., 2025)

- The generation engine receives new model weights mid-generation — briefly pausing, loading fresh weights, then continuing in-progress sequences. This creates mixed-policy sequences where early tokens are generated by a staler policy and later tokens by a fresher one.

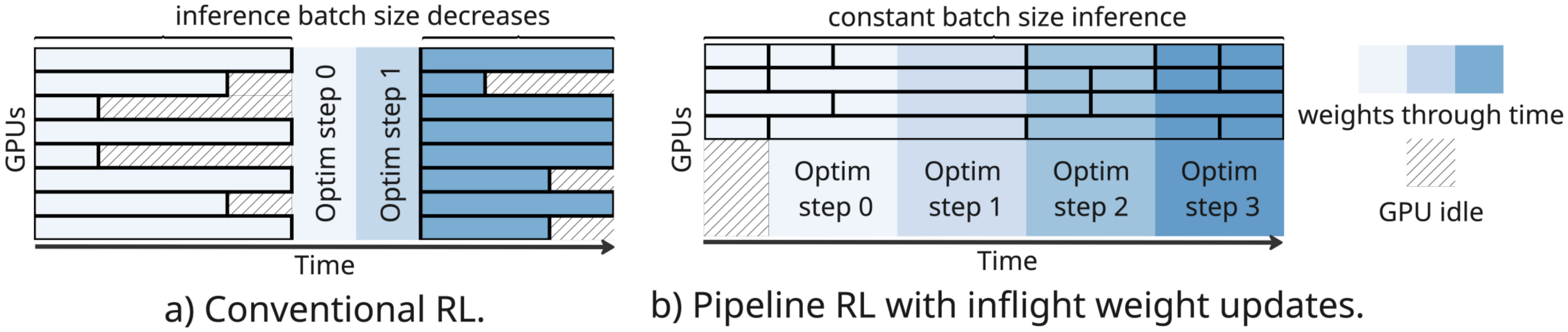


Figure 1: **a)** Conventional RL alternates between using all the GPUs for generation and then training. **b)** PipelineRL runs generation and training concurrently, always using the freshest model weights for generations thanks to the in-flight weight updates.

Why off-policy situations arise in practice

The Two-Engine Architecture

1. The Rollout Generator

   Input: The current policy (π_θ)

   Output: Rollouts

2. The Learner

   Input: Batches of rollouts from the generator.

   Output: Updated policy ($\pi_{\theta+1}$)

- Why rollouts become ”stale"
  - Deliberate asynchronous RL
  - Multiple gradient steps per batch
  - Large replay buffers
  - Separate generation and training clusters (one cluster optimized for inference, while another cluster optimized for gradient computation) require weight transfer between clusters, which introduce latency

Off-policy mitigations

Why off-policy is a problem

- Biased gradient estimates
- Importance weight explosion
- Reward hacking amplification

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$$

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t\left[\min(r_t(\theta)\hat{A}_t, \operatorname{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)\right]$$

- Mitigation strategies
  - PPO clipping
  - KL penalty against the reference policy
  - Algorithms without critiques/advantages (reducing the window of off-policy drifts) such as GRPO or REINFORCE
  - Less epochs
  - SGLang / vLLM's continuous batching with weight streaming
  - Pipeline RL's in-flight weight update

Lecture Plan

Speculative decoding (20 mins)

Off-policy drift & on-policy distillation (20 mins)

- Off-policy, on-policy, online RL, off-line RL
- RL infra and off-policy drift
- On-policy distillation

Long context extension (25 mins)

Inference-time scaling (15 min)

On-policy distillation (aka, generalized knowledge distillation)

- First introduced by (Gu et al, 2023) and (Agarwal et al, 2023), and later amplified by Qwen3's Tech report and ThinkingMachine's blog (https://thinkingmachines.ai/blog/on-policy-distillation/)
- Prior distillation methods were teacher-centric (thus off-policy w.r.t the learner)
- On-policy distillation is student-centric (thus on-policy w.r.t the learner)

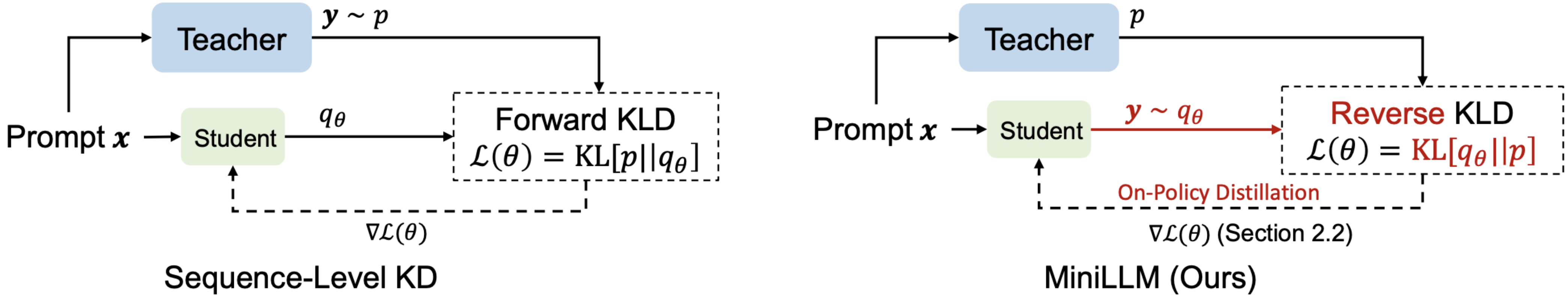


Figure 3: Comparison between sequence-level KD (left) and MiniLLM (right). Sequence-level KD forces the student to memorize all samples generated by the teacher model, while MiniLLM improves its generated texts with the teacher model's feedback.

Gu et al. 2023

KL Divergence: how Q differs from the target P

<u>Forward KL</u> — "**Mean-seeking" or "mass-covering"**

$$\mathrm{KL}(p \,\|\, q) \;=\; \mathbb{E}_{x\sim p}\left[\log \frac{p(x)}{q(x)}\right] \;=\; \sum_{x} p(x) \log \frac{p(x)}{q(x)}$$

- Weighted by P(x): penalizes Q where P has mass but Q does not
- Q covers **all modes** of P → over-estimates support
- Used in variational inference (ELBO)

<u>Reverse KL</u> — "**Mode-seeking" or "mode-collapsing"**

$$\mathrm{KL}(q \,\|\, p) \;=\; \mathbb{E}_{x\sim q}\left[\log \frac{q(x)}{p(x)}\right] \;=\; \sum_{x} q(x) \log \frac{q(x)}{p(x)}$$

- Weighted by Q(x): penalizes Q where Q has mass but P does not
- Q concentrates on **one mode** of P → under-estimates support
- Used in RLHF: KL penalty keeps policy close to reference model

On-policy distillation vs standard knowledge distillation

Let p_T = teacher, p_θ= student, $y*$ = ground truth, x = input

- Knowledge distillation (Hinton et al., 2015)

loss = forward KL

off-policy

$$\mathcal{L}_{\text{KD}} = -\sum_{t=1}^{T} \sum_{v \in \mathcal{V}} p_T^{(\tau)}(v \mid y^*_{<t}, x) \; \log p_\theta^{(\tau)}(v \mid y^*_{<t}, x)$$

- Sequential knowledge distillation (Kim & Rush, 2016)

loss = NLL

off-policy

$$\mathcal{L}_{\text{SeqKD}} = -\sum_{t=1}^{T} \log p_\theta(\hat{y}_t \mid \hat{y}_{<t}, x) \qquad \text{where} \quad \hat{y} \sim p_T(\cdot \mid x)$$

- On-policy knowledge distillation (Gu et al., 2023; Agarwal et al, 2023)

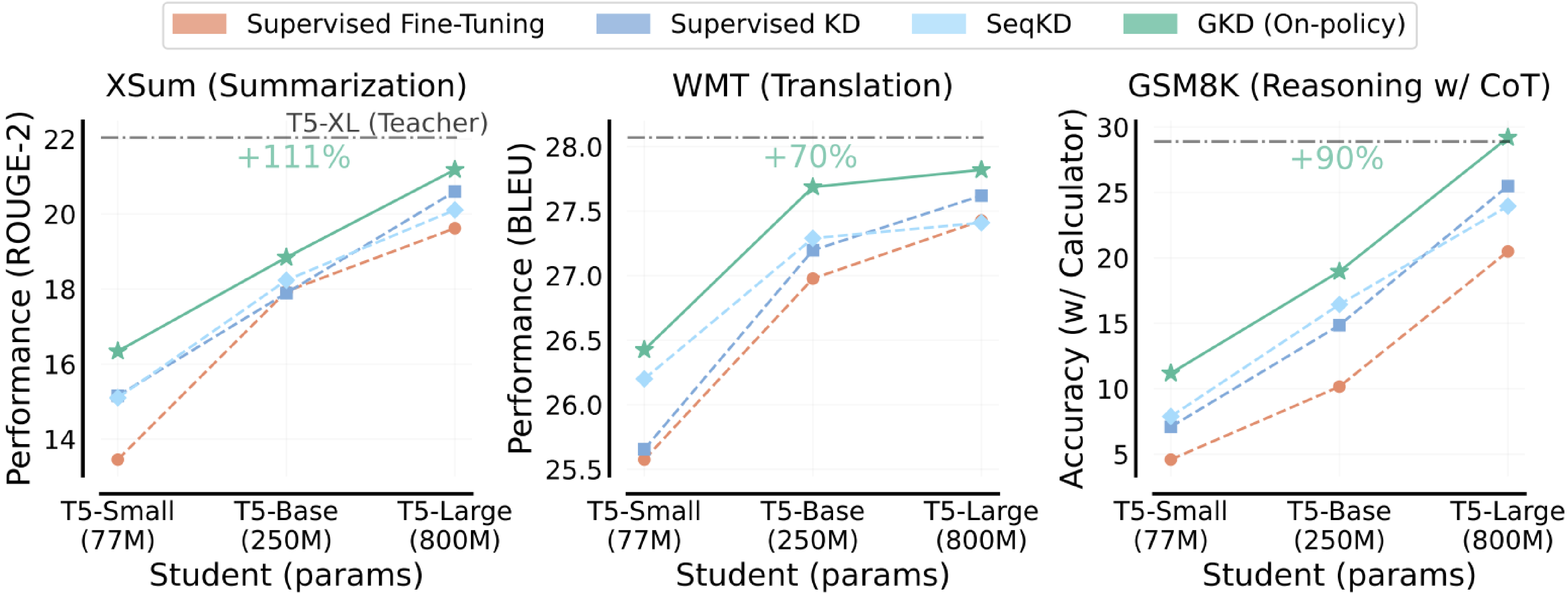
loss = reverse KL

on-policy

$$\mathcal{L}_{\text{GKD}} = \sum_{t=1}^{T} \sum_{v \in \mathcal{V}} p_\theta(v \mid \tilde{y}_{<t}, x) \left[\log p_\theta(v \mid \tilde{y}_{<t}, x) - \log p_T(v \mid \tilde{y}_{<t}, x)\right]$$

$$\text{where} \quad \tilde{y} \sim p_\theta(\cdot \mid x)$$

	loss	context	Hard vs soft	Learning style
SFT	NLL on gold output	Ground-truth context	Hard labels	
KD	Forward KL	Ground-truth context	Soft dist	Mass-covering
SeqKD	NLL on teacher's output	Teacher-generated context	Hard labels	Mass-covering
GKD	Reverse KL	Student-generated context	Soft dist	Mode-seeking



Agarwal et al. 2023

On-policy distillation

- The implication of reverse-KL: the context comes from the student's own generation.
- This eliminates train-test mismatch (exposure bias): the student learns to recover from its own mistakes.
- The mode-seeking property encourages the student to be sharp and confident on its best behaviors rather than diffusely covering all of the teacher's modes.

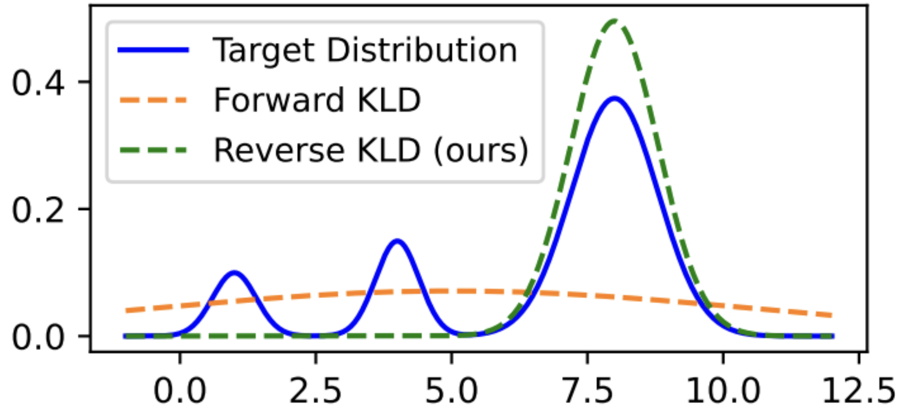


Figure 2: The toy experiment. We fit a Gaussian mixture distribution with a single Gaussian distribution using *forward* KLD and *reverse* KLD.

Gu et al. 2023

On policy distillation vs RL

- On policy distillation can be viewed as the best of both worlds of SFT and RL ---

	sampling	Reward signal	GPU requirement
Supervised fine-tuning	Off-policy	dense	light
Reinforcement learning	On-policy	sparse	heavy
On-policy distillation	On-policy	dense	light

Table 21: Comparison of reinforcement learning and on-policy distillation on Qwen3-8B. Numbers in parentheses indicate pass@64 scores.

Method	AIME'24	AIME'25	MATH500	LiveCodeBench v5	MMLU -Redux	GPQA -Diamond	GPU Hours
Off-policy Distillation	55.0 (90.0)	42.8 (83.3)	92.4	42.0	86.4	55.6	-
+ Reinforcement Learning	67.6 (90.0)	55.5 (83.3)	94.8	52.9	86.9	61.3	17,920
+ On-policy Distillation	**74.4 (93.3)**	**65.5 (86.7)**	**97.0**	**60.3**	**88.3**	**63.3**	1,800

On-policy distillation for domain-specific adaptation 🥳

- One very cool use case of on-policy distillation is domain-specific adaptation of OSS models, e.g., fine-tuning Qwen-8B on a domain-specific corpus (i.e., internal documents of a company), which can be viewed as a form of midtraining.
- Doing this in a naïve way (applying midtraining on top of the off-the-shelf OOS model which has been already post-trained) will cause degradation of capabilities acquired through the previous post-training (e.g., instruction following)
- On-policy distillation (on an instruction-tuning dataset, 30%) recovers the instruction following capability after acquiring the new domain-specific knowledge thru mid-training (70%)

Model	Internal QA Eval (Knowledge)	IF-eval (Chat)
Qwen3-8B	18%	<u>**85%**</u>
+ midtrain (100%)	<u>**43%**</u>	45%
+ midtrain (70%)	36%	79%
+ midtrain (70%) + distill	<u>**41%**</u>	<u>**83%**</u>

Lecture Plan

Speculative decoding (20 mins)

Off-policy drift & on-policy distillation (20 mins)

- Off-policy, on-policy, online RL, off-line RL
- RL infra and off-policy drift
- On-policy distillation

Long context extension (25 mins)

Inference-time scaling (15 min)

Scaling reasoning requires scaling long-context

- When applications require very long-context windows (100k to 2 million tokens):
  - Software engineering tasks that demand understanding the entire repository
  - Legal analysis that involves careful review of documents spanning hundreds of pages
  - Personalized chat interactions conditioned on prolonged interaction histories
  - Solving extremely challenging math problems that require elaborate sequences of trial and error across different problem-solving strategies

Scaling reasoning requires scaling long-context

What makes long-context challenging for LLMs

- Data limitation: most internet documents aren’t long enough to support pre-training with extremely long context windows
- Compute/memory limitation: standard attention requires quadratic computation
- Generalization limitation of positional embeddings: models trained on shorter sequences doesn’t generalize well on longer sequences

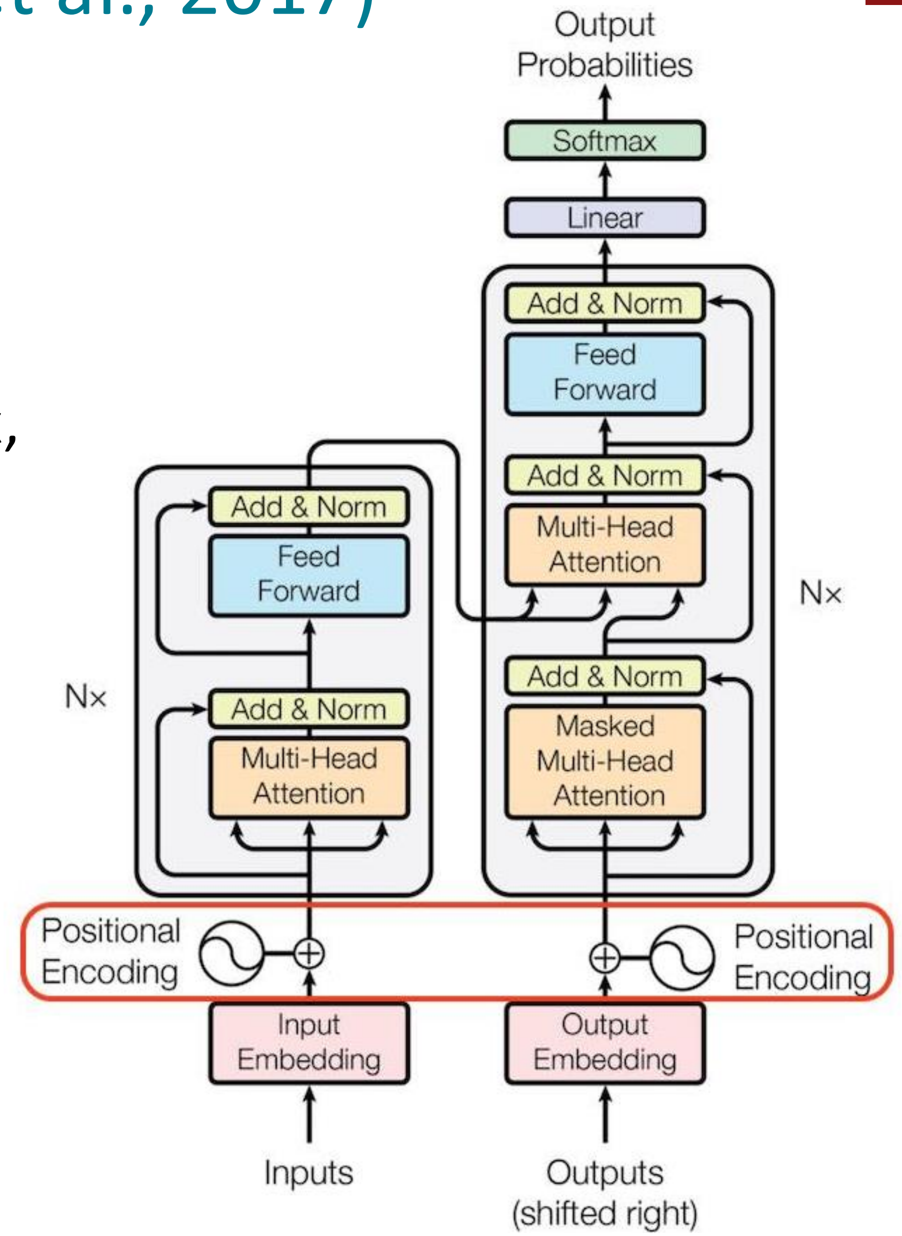
Typical document lengths of internet data

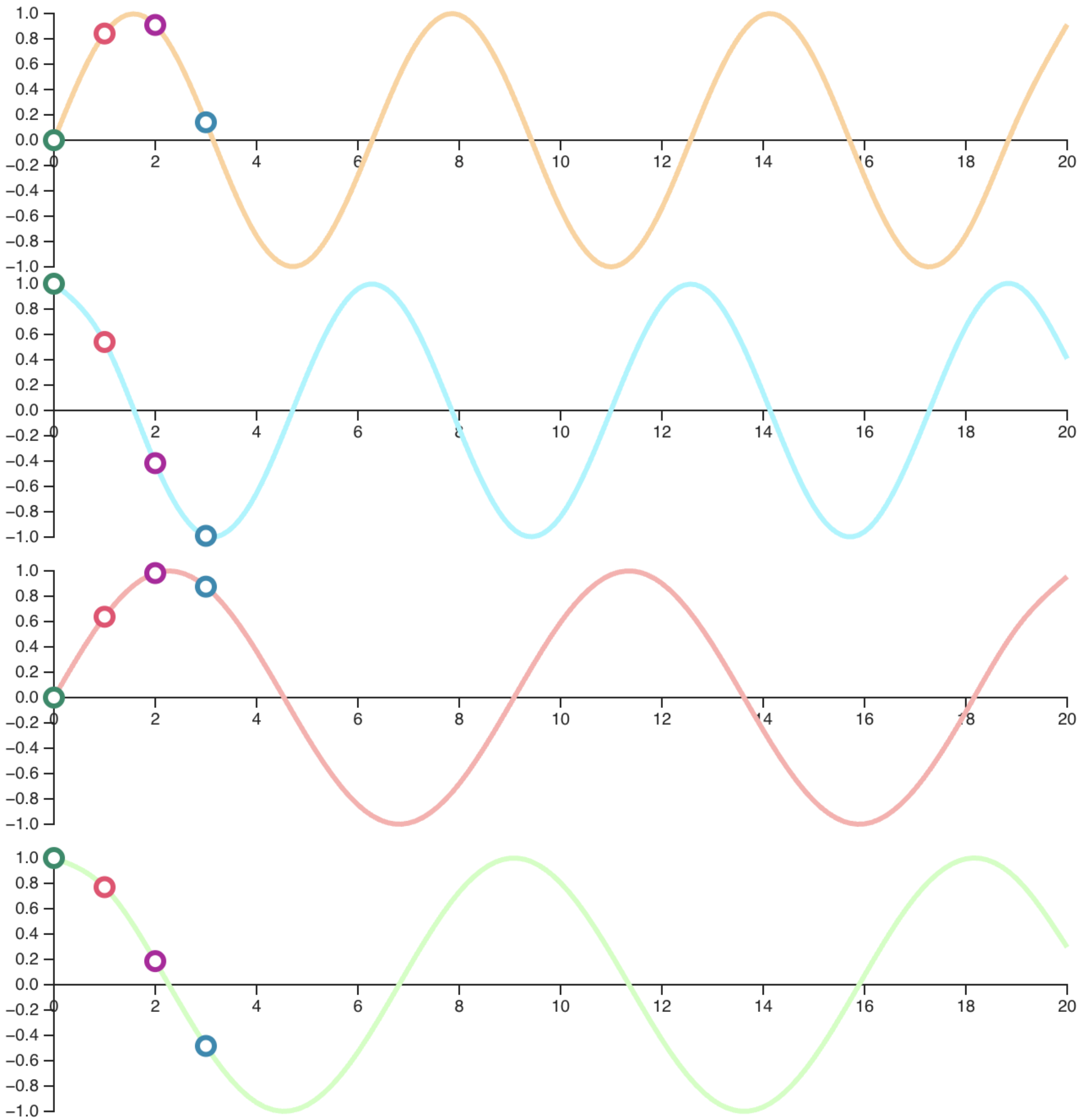
Source Type	Typical Length (Approx. Tokens)	Description
Common Crawl (Web)	600 – 1,200	Includes blog posts, news, and landing pages. Many "documents" are highly fragmented.
Wikipedia	500 – 1,500	While some entries are long, the median article is relatively concise.
Scientific Papers (arXiv)	5,000 – 10,000+	These are among the longest "natural" documents in most sets.
Books (Project Gutenberg)	50,000 – 100,000+	The "long-tail" of the data; rare but critical for long-range dependency.
GitHub (Code)	100 – 5,000	Code files vary wildly; many scripts are quite short, while libraries are long.

Sinusoidal positional embeddings (Vaswani et al., 2017)

$$PE(pos, 2i) = \sin(pos/10000^{2i/d})$$
$$PE(pos, 2i+1) = \cos(pos/10000^{2i/d})$$

- Where pos is the position index, i is the dimension index, and d is the model dimension.
- Used in the original Transformer (Vaswani et al., 2017)
- We are revisiting this because it helps to learn RoPE (Rotary positional embedding) that is recently important for LLMs and long-context extension





p0	p1	p2	p3	
0.000	0.841	0.909	0.141	i=0
1.000	0.540	-0.416	-0.990	i=1
0.000	0.638	0.983	0.875	i=2
1.000	0.770	0.186	-0.484	i=3

Positional Encoding

$$PE_{(pos,2i)} = sin\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right)$$

$$PE_{(pos,2i+1)} = cos\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right)$$

Settings: d = 50

The value of each positional encoding depends on the *position* (*pos*) and *dimension* (*d*). We calculate result for every *index* (*i*) to get the whole vector.

pos0 pos1 pos2 pos3

https://erdem.pl/2021/05/understanding-positional-encoding-in-transformers

Sinusoidal positional embeddings (Vaswani et al., 2017)

$$PE(pos, 2i) = \sin(pos/10000^{2i/d})$$

$$PE(pos, 2i+1) = \cos(pos/10000^{2i/d})$$

- Where pos is the position index, i is the dimension index, and d is the model dimension.
- Why this particular sinusoidal form?
  - Unique absolute positional embedding pre-defined for *any* position (whether seen during training or not)
  - Captures relative distance! For any offset k, PE_{pos+k} can be represented as a linear function of PE_{pos}
- Two important things to note:
  - Positional embedding is "added" to the token embedding
  - Previously unseen positional embeddings during training, while can be defined, are still unseen to the model, thus the model can't interpret them

Learned positional embeddings

- Randomly initialized, and then learned via backprop
- Used broadly in early days of LLMs, such as BERT, Roberta, GPT-2, GPT-3, Albert, Electra, BART
- It is *not* possible to define the positional embedding for a previously unseen position.
- Also doesn't generalize to positions beyond those seen during training.
- But performance was better when the model learns the positional encoding themselves
- This became a major bottleneck for long-context extension however, thus no longer used in recent LLMs

RoPE: Rotary Position Embedding (Su et al., 2021)

- Recall Attention

$$Q = W_Q X, \qquad K = W_K X, \qquad V = W_V X$$

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V$$

After query and key vectors are computed, multiply them with the rotation matrix!

- Attention with RoPE

$$\tilde{q}_m = R_\Theta^{(m)} q_m, \qquad \tilde{k}_n = R_\Theta^{(n)} k_n$$

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{\tilde{Q}\tilde{K}^\top}{\sqrt{d_k}}\right)V$$

Rotations are applied to just query and key vectors, not value vectors!

RoPE: Rotary Position Embedding (Su et al., 2021)

$$\begin{pmatrix} x'_{2i} \\ x'_{2i+1} \end{pmatrix} = \begin{pmatrix} \cos(m\theta_i) & -\sin(m\theta_i) \\ \sin(m\theta_i) & \cos(m\theta_i) \end{pmatrix} \begin{pmatrix} x_{2i} \\ x_{2i+1} \end{pmatrix}$$

Where m is the position index, i is the dimension index, $\theta_i = b^{-2i/d}$ is the frequency for dimension pair i, and b is the base frequency (typically $b = 10{,}000$)

The full rotation matrix of RoPE would then look like this:

$$\boldsymbol{R}^d_{\Theta,m} = \begin{pmatrix} \cos m\theta_1 & -\sin m\theta_1 & 0 & 0 & \cdots & 0 & 0 \\ \sin m\theta_1 & \cos m\theta_1 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & \cos m\theta_2 & -\sin m\theta_2 & \cdots & 0 & 0 \\ 0 & 0 & \sin m\theta_2 & \cos m\theta_2 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & \cos m\theta_{d/2} & -\sin m\theta_{d/2} \\ 0 & 0 & 0 & 0 & \cdots & \sin m\theta_{d/2} & \cos m\theta_{d/2} \end{pmatrix}$$

RoPE: Rotary Position Embedding (Su et al., 2021)

$$\begin{pmatrix} x'_{2i} \\ x'_{2i+1} \end{pmatrix} = \begin{pmatrix} \cos(m\theta_i) & -\sin(m\theta_i) \\ \sin(m\theta_i) & \cos(m\theta_i) \end{pmatrix} \begin{pmatrix} x_{2i} \\ x_{2i+1} \end{pmatrix}$$

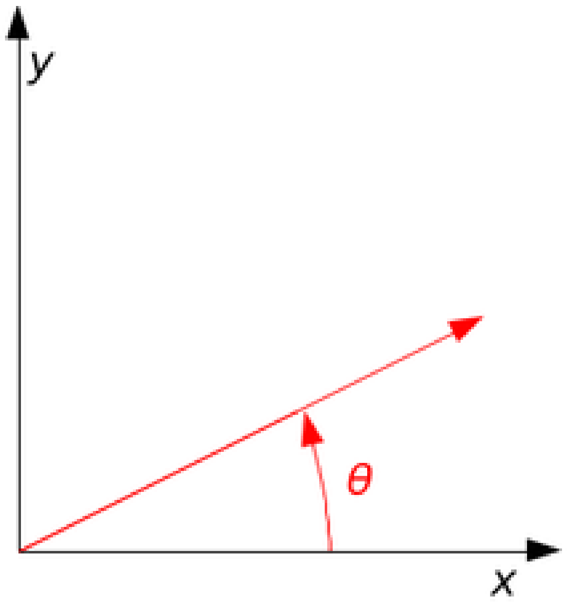
Where m is the position index, i is the dimension index, $\theta_i = b^{-2i/d}$ is the frequency for dimension pair i, and b is the base frequency (typically $b = 10{,}000$)

- RoPE encodes position by *rotating* the query and key vectors in the 2D plane.

Recall the rotation matrix:

$$R(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$



RoPE: Rotary Position Embedding (Su et al., 2021)

- RoPE encodes position by *rotating* the query and key vectors in the 2D plane.

- **High-frequency dimensions** (small i): rotate rapidly, encoding fine-grained local positional information. These dimensions cycle through many full rotations over the training length.
- **Low-frequency dimensions** (large i): rotate slowly, encoding broad/global positional information. These dimensions complete very few full rotations even over very long sequences.

- Dimension pair 0: $\theta_0 = 1.0$ — highest frequency, one full rotation every ~ 6.28 tokens
- Dimension pair 31: $\theta_{31} = 0.01$ — one rotation every ~ 628 tokens
- Dimension pair 63: $\theta_{63} = 0.0001$ — one rotation every $\sim 62{,}832$ tokens

RoPE: Rotary Position Embedding (Su et al., 2021)

- Attention with RoPE

$$Q = W_Q X, \qquad K = W_K X, \qquad V = W_V X$$

$$\tilde{q}_m = R_\Theta^{(m)} q_m, \qquad \tilde{k}_n = R_\Theta^{(n)} k_n$$

$$\mathrm{Attention}(Q, K, V) = \mathrm{softmax}\left(\frac{\tilde{Q}\tilde{K}^\top}{\sqrt{d_k}}\right) V$$

“Taylor” and “Swift” at position 10 and 11 will get the exact same result as “Taylor and “Swift” at position 100 and 101!

- Relative distance
  - because the transposed R rotates backward, we have

$$R^{(m)\top} R^{(n)} = R^{(n-m)}$$

$$\tilde{q}_m^\top \tilde{k}_n = q_m^\top \left(R_\Theta^{(m)}\right)^\top R_\Theta^{(n)} k_n = q_m^\top R_\Theta^{(n-m)} k_n$$

While their corresponding rotation matrices are all different, by the time q and k are multiplied, the rotational angle becomes identical! 😍 v

RoPE: Roterary Positional Embedding (Su et al., 2021)

Feature	**Original Sinusoidal (2017)**	**RoPE (LLaMA, PaLM, etc.)**
Integration	**Additive:** Added to the input embedding before the first layer	**Multiplicative:** Applied to Q and K at every attention layer
Norm preservation	Increases the token embedding's norm	Preserves the token embedding's norm
Relative Distance	Mathematically captured, but due to the additive integration, the information gets muddled up by the time QKV attention is computed	Mathematically captured via the rotation angle and the information is cleanly available when the QKV attention is computed
Extrapolation	Struggles significantly	Allows for additional long context extension techniques such as "Position Interpolation", "NTK scaling", and YARN

Long context extension practices 🧵

1. Position Encoding Modifications

RoPE scaling appears to be the most common family of approaches (as of 2026):

- Linear interpolation (Position Interpolation): Scales position indices down by a factor so the model sees familiar relative positions. Introduced by Meta for extending LLaMA from 2K→32K+. Requires light fine-tuning.
- NTK-aware interpolation: Adjusts the rotary base frequency rather than linearly compressing positions. Better preserves high-frequency components. Has "dynamic NTK" variants that adapt scaling based on sequence length at inference.
- YaRN (Yet another RoPE extensioN): Combines NTK scaling with an attention temperature correction and a ramp function that treats different frequency bands differently—interpolating low frequencies while extrapolating high frequencies.

Long context extension practices 🧵

2. Progressive / Staged Training

A common recipe (used by LongLLaMA, LongAlign, Llama 3.1, etc.):

- Start from a base model (e.g., 4K–8K context).
- Apply position encoding modification.
- **Continual pretraining** on long documents with progressively increasing sequence lengths (e.g., 8K → 32K → 64K → 128K), often with a relatively small amount of data (billions of tokens, not trillions).
- Fine-tune on long-context instruction data.
- The key insight is that you don't need the full pretraining budget—typically 0.1–1% of original pretraining tokens suffices for adaptation.

Long context extension practices 🧵

3. Data Engineering

- **Upsampling long documents** in the continued pretraining mix (books, code repos, long-form articles, concatenated related documents).
- **Synthetic long-context tasks**: Needle-in-a-haystack retrieval, long-range QA, multi-document summarization.
- **LongAlign-style** instruction tuning with tasks specifically requiring the model to use information spread across the full context.
- **Self-instruct for long contexts**: Using a capable model to generate long-context instruction-response pairs.

Long context extension practices 🧵

4. Attention Architecture Modifications

- **Sparse / sliding window attention** combined with global attention tokens (Longformer-style).
- **Flash Attention** (and FlashAttention-2/3) to reduce memory from $O(n^2)$ to $O(n)$ during training.
- **Grouped Query Attention (GQA)** or **Multi-Query Attention (MQA)**: Reduces KV cache memory, enabling longer contexts at inference.

Lecture Plan

Speculative decoding (20 mins)

Off-policy drift & on-policy distillation (20 mins)

- Off-policy, on-policy, online RL, off-line RL
- RL infra and off-policy drift
- On-policy distillation

Long context extension (20 mins)

Inference-time scaling (10 min)

Test time compute scaling 🚀

- This notion was intensely popularized by OpenAI's O1 release in Sep 2024
- Though it was preceded by the GDM paper from Aug 2024
- And it also appeared less explicitly in the "Let's verify step by step" paper from May 2023

Google DeepMind 2024

Scaling LLM Test-Time Compute Optimally can be More Effective than Scaling Model Parameters

Charlie Snell[♦, 1], Jaehoon Lee[2], Kelvin Xu[♣, 2] and Aviral Kumar[♣, 2]

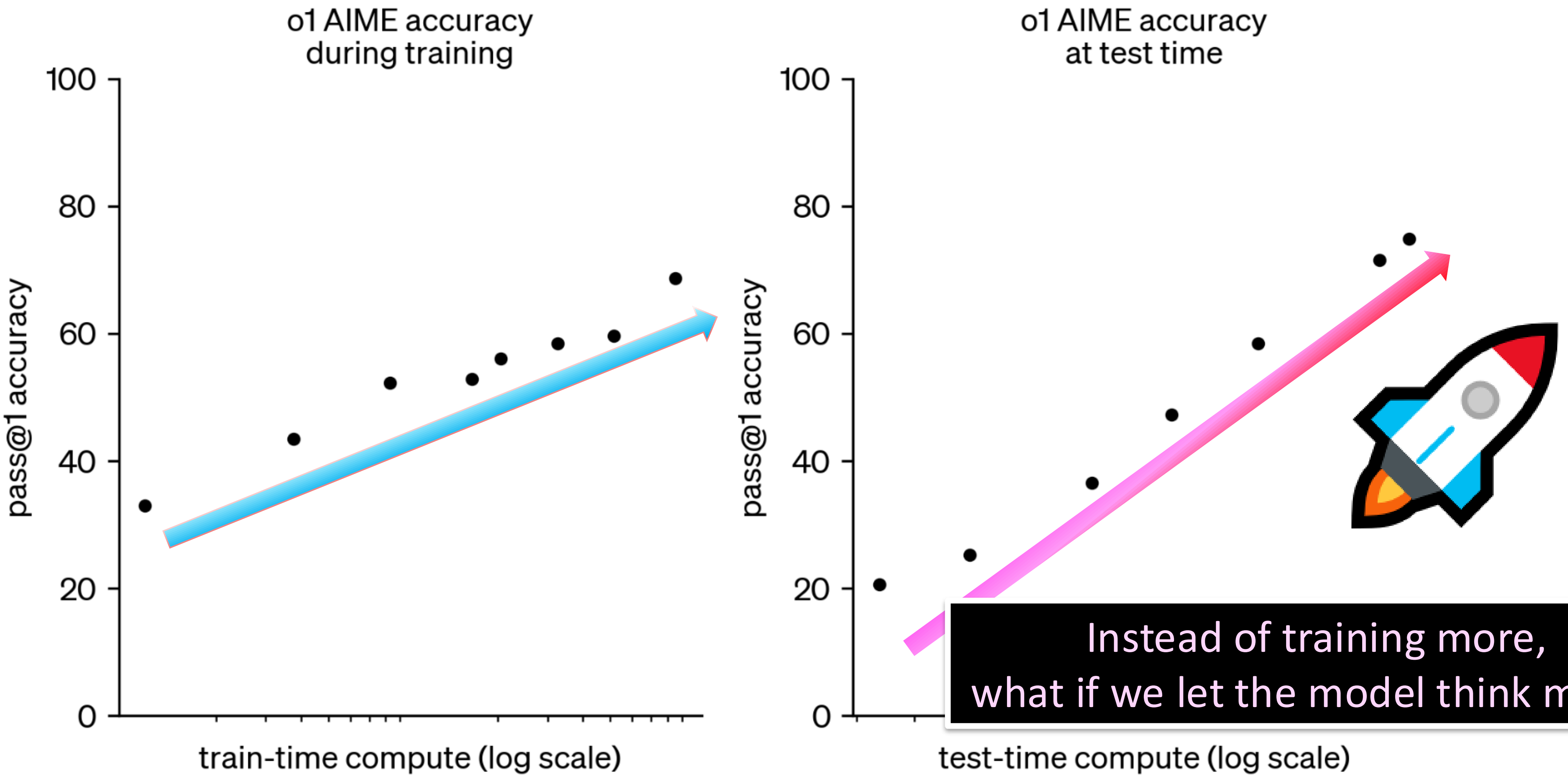
Let's Verify Step by Step

Hunter Lightman* Vineet Kosaraju* Yura Burda* Harri Edwards
Bowen Baker Teddy Lee Jan Leike John Schulman Ilya Sutskever
Karl Cobbe*

OpenAI

Test time compute scaling

Instead of scaling only training time compute, what if we scale test time compute?



Instead of training more,
what if we let the model think more?

Test time compute scaling 🚀

Google DeepMind 2024

Scaling LLM Test-Time Compute Optimally can be More Effective than Scaling Model Parameters

Charlie Snell[♦,1], Jaehoon Lee[2], Kelvin Xu[♣,2] and Aviral Kumar[♣,2]

- This paper challenges the dominant paradigm in LLM development by demonstrating that intelligently scaling test-time compute can yield larger performance gains than simply scaling model parameters
- For a fixed inference budget, smaller models using smart test-time compute strategies can outperform larger models using standard decoding
- For compute-optimal allocation, seek an optimal balance between
  - Model size (pretraining FLOPs)
  - Number of generated samples or revision steps (test-time FLOPs)

Google DeepMind

202

Scaling LLM Test-Time Compute Optimally can be More Effective than Scaling Model Parameters

Charlie Snell[♦, 1], Jaehoon Lee[2], Kelvin Xu[♣, 2] and Aviral Kumar[♣, 2]

Test time compute scaling 🚀

- Test-time compute scaling strategies evaluated:
  - **Best-of-N sampling**: Generate N independent solutions, select best via verifier
  - **Weighted Best-of-N**: Sample from compute-optimal temperature distributions
  - **Sequential revisions**: Iteratively refine single solutions using model self-critique
  - **Beam search with PRMs**: Maintain multiple solution candidates, pruning based on step-level rewards
  - **Diverse beam search**: Encourage exploration across different solution approaches
- Verification methods evaluated:
  - Outcome-supervised reward models (ORMs): Trained to predict solution correctness
  - Process-supervised reward models (PRMs): Trained to evaluate correctness at each reasoning step
  - Domain-specific verifiers: For problems with checkable solutions (e.g., code execution)

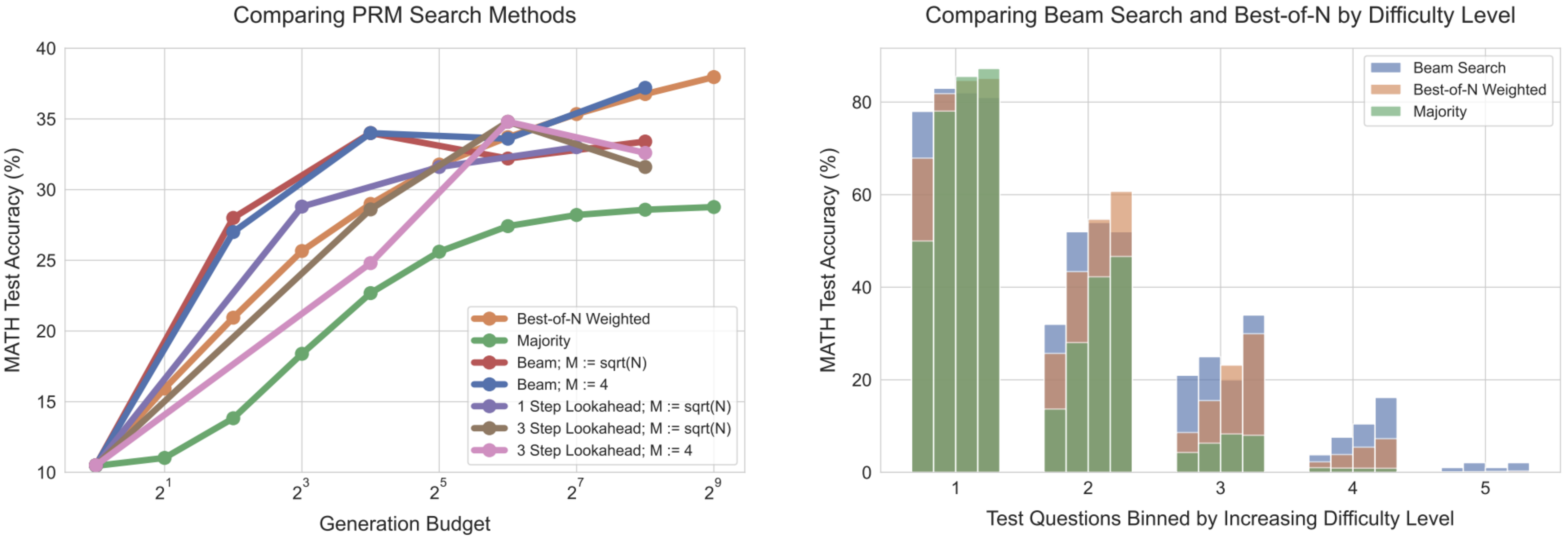


Figure 3 | **Left:** ***Comparing different methods for conducting search against PRM verifiers***. We see that at low generation budgets, beam search performs best, but as we scale the budget further the improvements diminish, falling below the best-of-N baseline. Lookahead-search generally underperforms other methods at the same generation budget. **Right:** ***Comparing beam search and best-of-N binned by difficulty level***. The four bars in each difficulty bin correspond to increasing test-time compute budgets (4, 16, 64, and 256 generations). On the easier problems (bins 1 and 2), beam search shows signs of over-optimization with higher budgets, whereas best-of-N does not. On the medium difficulty problems (bins 3 and 4), we see beam search demonstrating consistent improvements over best-of-N.

Test time compute scaling 🚀

Google DeepMind 202

Scaling LLM Test-Time Compute Optimally can be More Effective than Scaling Model Parameters

Charlie Snell[♦, 1], Jaehoon Lee[2], Kelvin Xu[♣, 2] and Aviral Kumar[♣, 2]

- Key findings:
  - On MATH-500, a 14B parameter model with optimized test-time compute matched or exceeded the performance of models 4× larger
  - The compute-optimal strategy allocated roughly equal FLOPs to pretraining and inference for their experimental setup
  - Sequential revision strategies showed particularly strong scaling on tasks requiring refinement and error correction
  - PRMs enabled 4-8× more efficient compute usage compared to ORMs w/ beam search
- Practical takeaways:
  - Don't default to the largest model!
  - Invest in verification!
    - High-quality reward models, especially PRMs, dramatically improve test-time scaling