Beyond Decoding: Meta-Generation Algorithms for Large Language Models

Presenters: Matthew Finlayson, Hailey Schoelkopf, Sean Welleck

December 11, 2024

Today's talk

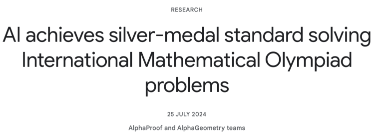
Algorithms for generating outputs with a language model

Today's talk

Algorithms for generating outputs with a language model

Why? *Use **test-time compute** to improve performance*

Language models

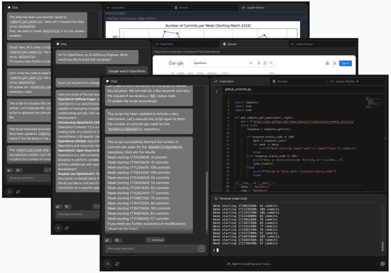

RESEARCH

AI achieves silver-medal standard solving International Mathematical Olympiad problems

25 JULY 2024

AlphaProof and AlphaGeometry teams

Solving olympiad problems

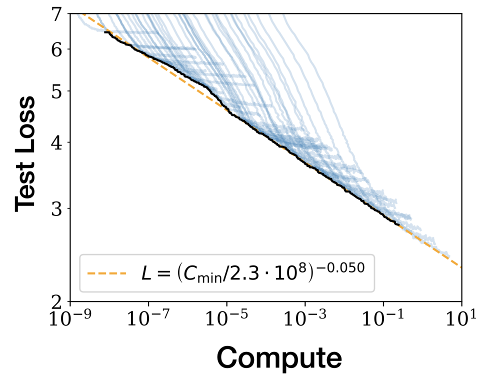


Writing code

Tasks framed as generating sequences: many other applications

Approach 1: scale pretraining compute

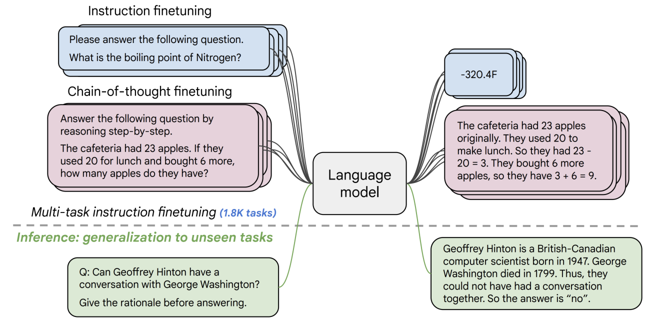
[2020-] **Scaling pretraining:** larger model, larger dataset



Scaling Laws for Neural Language Models [Kaplan et al., 2020]

Approach 2: scale post-training compute

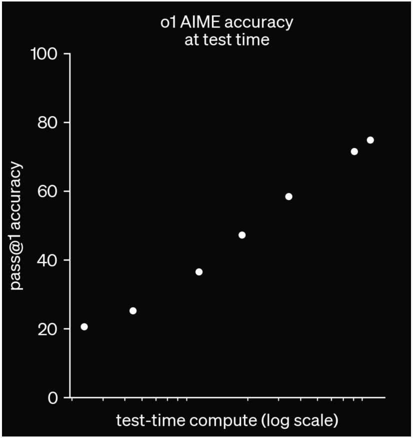
[2022-] **Scaling post-training:** e.g., fine-tune on (input, output) pairs



Scaling Instruction-Finetuned Language Models [Chung et al., 2022]

Approach 3: scale *test-time* compute

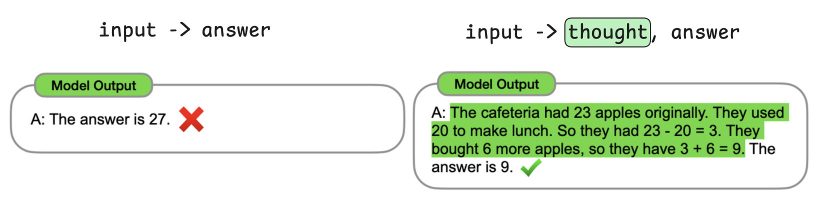
[Now] **Test-time scaling:** increase compute at generation time



Test-time compute vs. accuracy ([OpenAI, 2024])

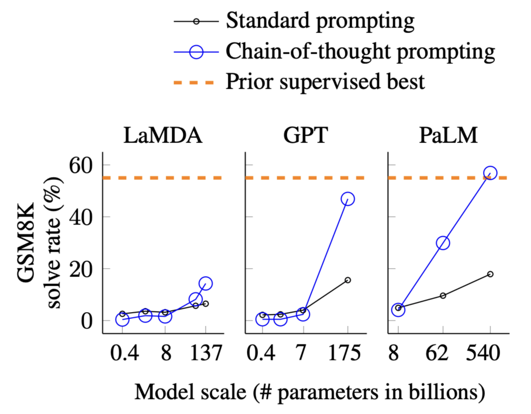
Approach 3: scale *test-time* compute | How?

1. Generate extra tokens



[Wei et al., 2022]

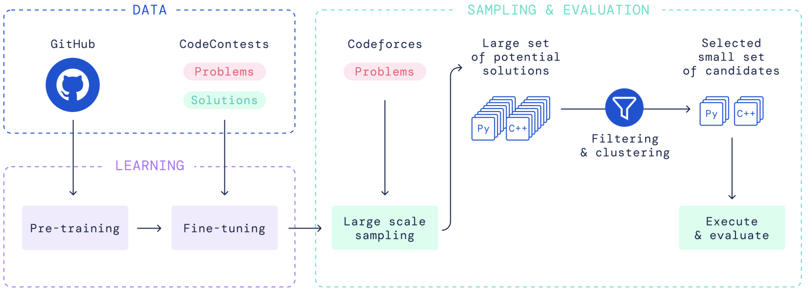
1. Generate extra tokens



[Wei et al., 2022]

Approach 3: scale *test-time* compute | How?

1. Generate extra tokens
2. Call generator multiple times

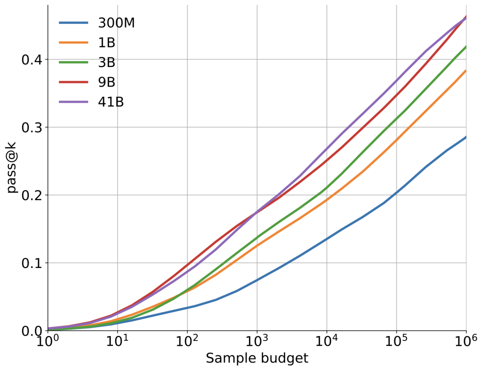


Overview of AlphaCode.

AlphaCode [Li et al., 2022]

Approach 3: scale *test-time* compute | How?

1. Generate extra tokens
2. Call generator multiple times

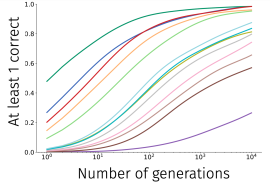


(b) Unlimited attempts per problem

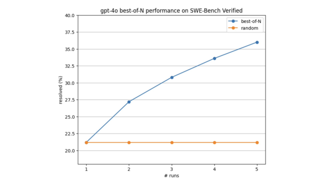
AlphaCode [Li et al., 2022]

Approach 3: scale *test-time* compute | How?

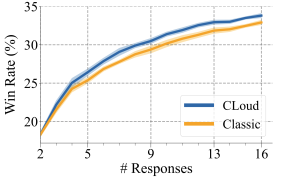
1. Generate extra tokens
2. Call generator multiple times



Math [Brown et al., 2024]



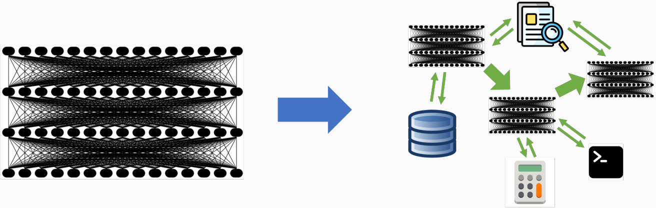
Agents [Nebius, 2024]



Chat [Ankner et al., 2024]

Approach 3: scale *test-time* compute | How?

1. Generate extra tokens
2. Call generator multiple times
3. Incorporate other models/tools

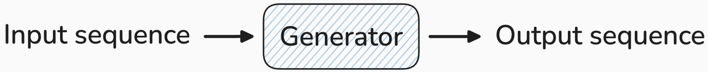


[Zaharia et al., 2024]

Verifiers, code interpreters, search engines, ...

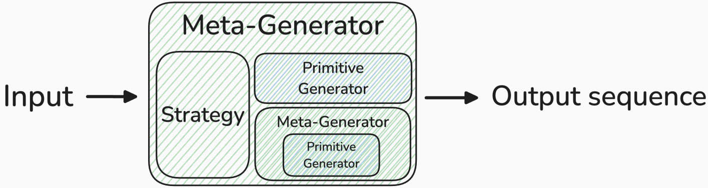
This tutorial: How? *Meta-Generation Algorithms*

Generator: Generates a sequence with a language model.



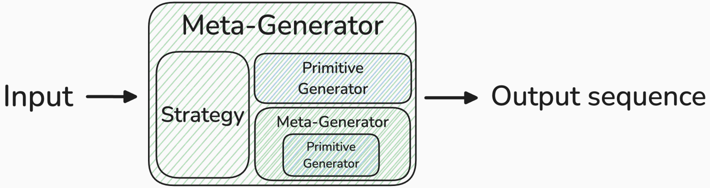
- Example: calling an LLM API
- Traditional algorithms
  - Greedy decoding
  - Temperature sampling
  - ...

Meta-generator: High-level strategies for calling generators and using external information.



- Example: call API multiple times, select the best sequence with a separate model

Meta-generator: High-level strategies for calling generators and using external information.



Why?

- Generate more to improve task performance
- Combine multiple models (verifiers, retrievers, . . .)
- Incorporate external information (tools, feedback, . . .)

This tutorial

Beyond Decoding: Meta-Generation Algorithms for LLMs

- I: **Primitive generators:** Generating one token at a time
- II: **Meta-generators:** High-level strategies for calling generators
- III: **Efficient meta-generation:** Generating quickly and efficiently

Panel session at the end!

Presenters

Part I	Intro/Part II	Part III
		
Matthew Finlayson	Sean Welleck	Hailey Schoelkopf
USC	CMU	EleutherAI
@mattf1n	@wellecks	@haileysch__

Panel

Beidi Chen
CMU
@BeidiChen

Nouha Dziri
AI2
@nouhadziri

Rishabh Agarwal
DeepMind/McGill
@agarwl_

Jakob Foerster
Oxford/Meta AI
@j_foerst

Noam Brown
OpenAI
@polynoamial

Ilia Kulikov (Moderator)
Meta AI
@uralik1

This tutorial | team and survey

Survey (TMLR 2024): *From Decoding to Meta-Generation: Inference-time Algorithms for Large Language Models* [Welleck et al., 2024]

This tutorial | resources

cmu-l3.github.io/neurips2024-inference-tutorial

Code examples, reading list, slides

I. Primitive Generators

I. Primitive Generators

Generating one token at a time

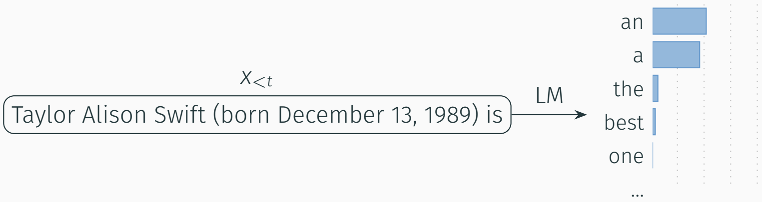
This tutorial

Beyond Decoding: Meta-Generation Algorithms for LLMs

- <u>Primitive Generators</u>
- Meta-generators
- Efficient meta-generation

Token-level generation

Auto-regressive language modeling uses a causal language model, which defines a conditional distribution over tokens $p_\theta[x_t \mid x_{<t}]$.



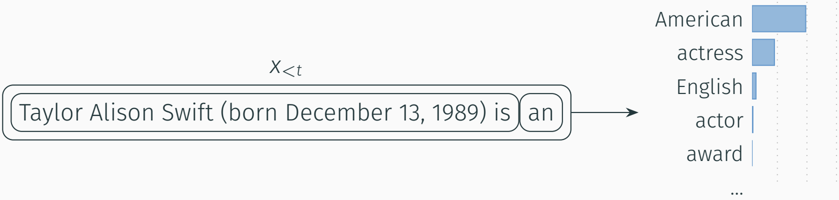
Token-level generation

Auto-regressive language modeling uses a causal language model, which defines a conditional distribution over tokens $p_\theta[x_t \mid x_{<t}]$.

$x_{<t}$	x_t
Taylor Alison Swift (born December 13, 1989) is	an

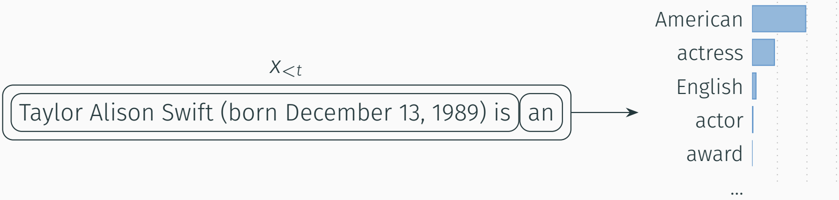
Token-level generation

Auto-regressive language modeling uses a causal language model, which defines a conditional distribution over tokens $p_\theta[x_t \mid x_{<t}]$.



Token-level generation

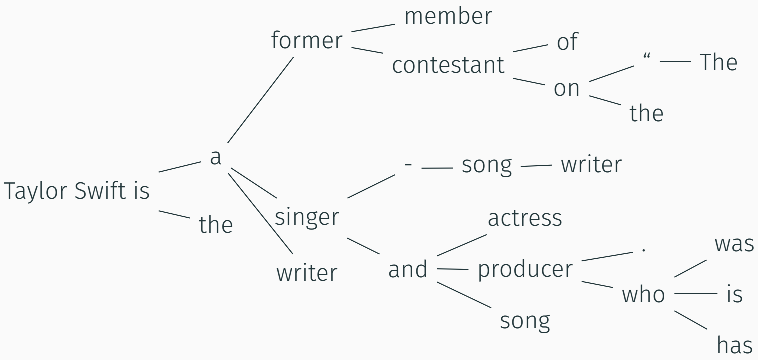
Auto-regressive language modeling uses a causal language model, which defines a conditional distribution over tokens $p_\theta[x_t \mid x_{<t}]$.



Token-level decoding algorithms are primarily concerned with *how to choose the next token.*

Decoding is search

Each time-step during decoding requires a choice.



But a search for what? What is our *objective*? How do we make *local* choices that achieve the objective?

Token-level generation (outline)

Objectives for decoding

- Optimization
- Sampling
- Constrained generation, structured outputs

I. Primitive Generators

Decoding as optimization

Maximum A Posteriori (MAP)

MAP decoding seeks to find the *most likely sequence*

$$\arg\max_{x} p_{\theta}[x]$$

- Greedy decoding
- Beam search

Greedy decoding

- Choose the *most-likely* token at each step.

$$x_t = \arg\max_x p_\theta[x \mid x_{<t}]$$

Greedy decoding

- Choose the *most-likely* token at each step.

$$x_t = \arg\max_x p_\theta[x \mid x_{<t}]$$

- Does not guarantee the most-likely sequence.

	Prefix	Continuation			Prob.
Greedy	Taylor Swift is a	former	contestant	on	
Token prob.		0.023	0.022	0.80	0.0004

Greedy decoding

- Choose the *most-likely* token at each step.

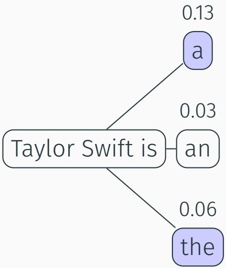
$$x_t = \arg\max_x p_\theta[x \mid x_{<t}]$$

- Does not guarantee the most-likely sequence.

	Prefix	Continuation			Prob.
Greedy	Taylor Swift is a	former	contestant	on	
Token prob.		0.023	0.022	0.80	0.0004
Non-greedy	Taylor Swift is a	singer	,	song	
Token prob.		0.012	0.26	0.21	**0.0007**

Beam Search

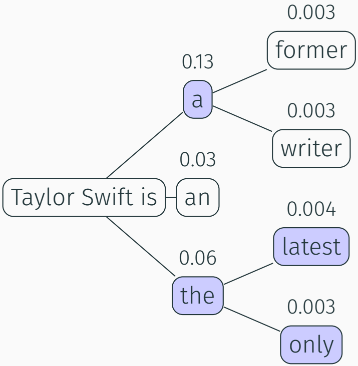
Beam-search is a width-limited breadth-first search (BFS).



GPT2, beam size 2

Beam Search

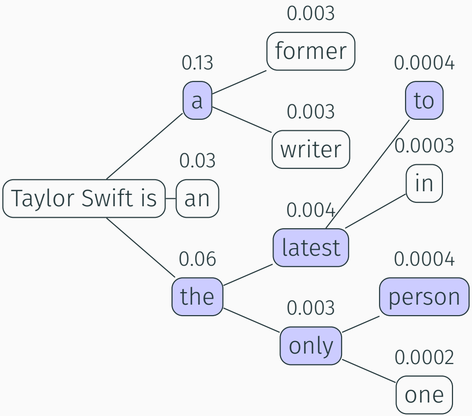
Beam-search is a width-limited breadth-first search (BFS).



GPT2, beam size 2

Beam Search

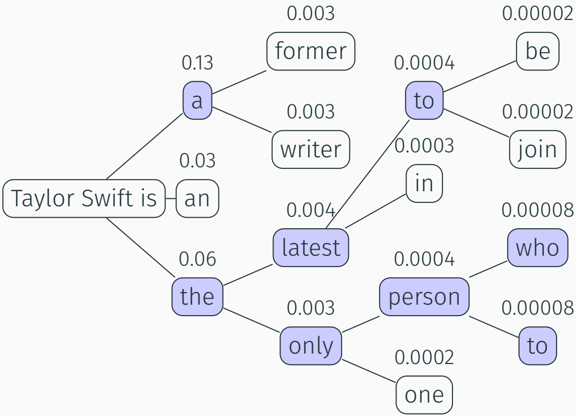
Beam-search is a width-limited breadth-first search (BFS).



GPT2, beam size 2

Beam Search

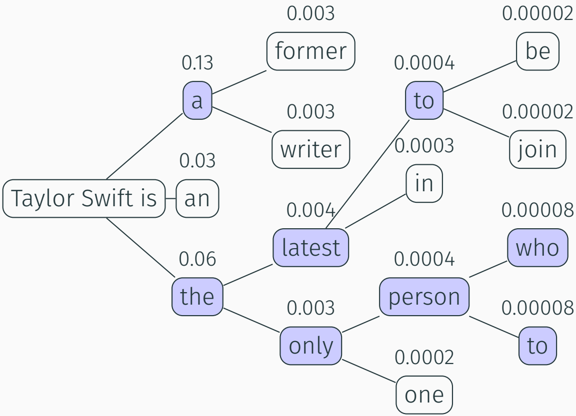
Beam-search is a width-limited breadth-first search (BFS).



GPT2, beam size 2

Beam Search

Beam-search is a width-limited breadth-first search (BFS).

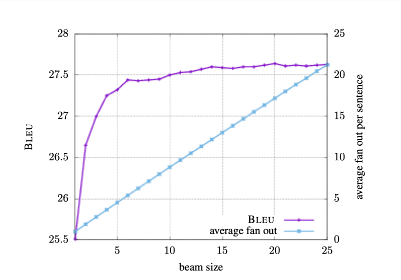


GPT2, beam size 2

Note: Beam search with beam size 1 is greedy decoding.

Benefits of MAP

MAP decoding works well for closed-ended tasks like translation, question answering.



[Freitag and Al-Onaizan, 2017]

Model	Dataset	Metric	Greedy	BS
Llama2-7B	HumanEval	Pass@1	12.80	15.24
	MBPP		17.80	19.40
	GSM8K	Acc	13.87	17.21
	XSUM	R-L	27.21	21.88
	CNN/DM		23.43	20.69
	De⇒En	B-4	28.80	30.14
	En⇒De		22.63	23.99
	Zh⇒En		19.44	20.11
	En⇒Zh		15.15	14.50
	CQA	Acc	62.90	64.37
	SQA		60.76	62.25

[Shi et al., 2024]

Pitfalls of MAP

Probability maximization causes decoding problems.

- Repetition traps
- Short sequences [Stahlberg and Byrne, 2019]
- Atypicality [Meister et al., 2022]

Pitfalls of MAP

Probability maximization causes decoding problems.

- **Repetition traps**
- Short sequences [Stahlberg and Byrne, 2019]
- Atypicality [Meister et al., 2022]

GPT2, Beam size 32.

> *Taylor Alison Swift (born December 13, 1989) is an American singer-songwriter, singer-songwriter, songwriter, and songwriter. She is best known for her work as a singer-songwriter, songwriter-songwriter, songwriter-songwriter, songwriter-songwriter...*

Remedies:

- repetition penalty
- unlikelihood training [Welleck et al., 2020]

Pitfalls of MAP

Probability maximization causes decoding problems.

- Repetition traps
- **Short sequences** [Stahlberg and Byrne, 2019]
- Atypicality [Meister et al., 2022]

$$\textbf{Pr}[\text{Taylor Swift is <eos>}] > \textbf{Pr}[\text{Taylor Swift is an American singer-...}]$$

Remedy: length normalization

Pitfalls of MAP

Probability maximization causes decoding problems.

- Repetition traps
- Short sequences [Stahlberg and Byrne, 2019]
- **Atypicality** [Meister et al., 2022]

- Biased coin $\Pr[\text{H}] = 0.6$, $\Pr[\text{T}] = 0.4$.
- Most likely outcome from 100 flips is all heads
  H H H H H H H H H H ...
- But this outcome is *atypical*.
- Similarly, the *most likely generation* may *also* be atypical.

Remedy to all of the above: *sampling*

Pitfalls of MAP

Probability maximization causes decoding problems.

- Repetition traps
- Short sequences [Stahlberg and Byrne, 2019]
- Atypicality [Meister et al., 2022]

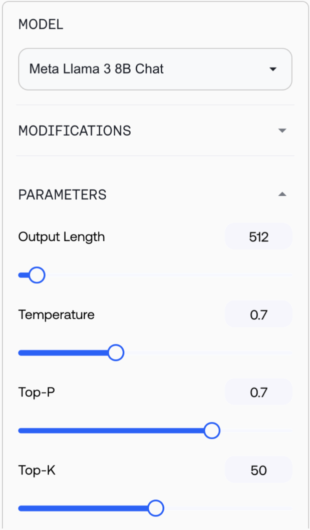
Takeaway: Approximate MAP (e.g., narrow beam search) works better than exact MAP [Meister et al., 2020].

I. Primitive Generators

Sampling

Objective: Sampling

Modern LLM APIs like Together.AI offer settings for *sampling*.



Together.ai playground.

Ancestral sampling

- $y_1 \sim p_\theta(\cdot \mid x)$
- $y_2 \sim p_\theta(\cdot \mid x, y_1)$
- $y_3 \sim p_\theta(\cdot \mid x, y_2, y_3)$
- ...

Ancestral sampling

- $y_1 \sim p_\theta(\cdot \mid x)$
- $y_2 \sim p_\theta(\cdot \mid x, y_1)$
- $y_3 \sim p_\theta(\cdot \mid x, y_2, y_3)$
- ...

Ancestral sampling is equivalent to sequence sampling.

$$p_\theta(\mathbf{y}) = p_\theta(y_1)p_\theta(y_2 \mid y_1)p_\theta(y_3 \mid y_1 y_2)\ldots p_\theta(y_T \mid \mathbf{y}_{<T})$$

What is wrong with ancestral sampling?

- Greedy decoding causes repetition traps

Greedy (repetition trap)
Taylor Swift is a former contestant on the reality show … "I think it's a very sad day for the show," he said. "It's a very sad day for the show. It's a very sad day for the show. It's a very sad …

What is wrong with ancestral sampling?

- Greedy decoding causes repetition traps
- But ancestral sampling causes incoherence. Why?
- Low-probability tokens are *too likely*
- I.e., the distribution has a *heavy tail*.

Greedy (repetition trap)	Ancestral (incoherent)
Taylor Swift is a former contestant on the reality show … "I think it's a very sad day for the show," he said. "It's a very sad day for the show. It's a very sad day for the show. It's a very sad …	Taylor Swift is a huge fan of her latest album 'Famous.' The singer got her first reaction when she uploaded to Twitter a video of her dancing and singing at a reception for a Grammy-nominated female songstress, Beyoncé.

What is wrong with ancestral sampling?

- Greedy decoding causes repetition traps
- But ancestral sampling causes incoherence. Why?
- Low-probability tokens are *too likely*
- I.e., the distribution has a *heavy tail*.
- Solution: chop off the tail!

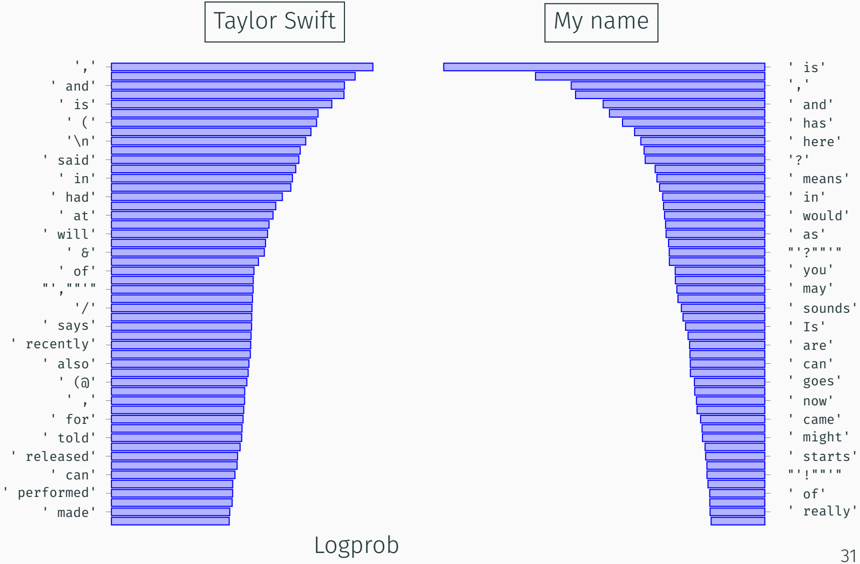
Greedy (repetition trap)	Ancestral (incoherent)	**Top-k (acceptable)**
Taylor Swift is a former contestant on the reality show … "I think it's a very sad day for the show," he said. "It's a very sad day for the show. It's a very sad day for the show. It's a very sad …	Taylor Swift is a huge fan of her latest album 'Famous.' The singer got her first reaction when she uploaded to Twitter a video of her dancing and singing at a reception for a Grammy-nominated female songstress, Beyoncé.	Taylor Swift is a writer for IGN and a member of IGN's Television Critics Association. You can follow her on Twitter at @_MsSwift, IGN at MsSwiftIGN, Facebook at MrsSwift, or subscribe to her video channels.

Truncation sampling

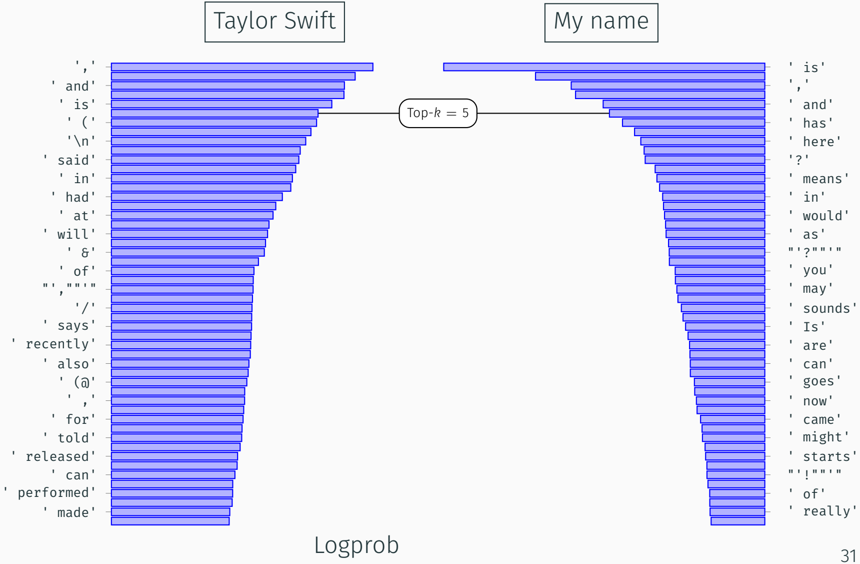
Truncation sampling interpolates greedy and ancestral sampling by choosing a minimum probability threshold at each time step.

Method	Threshold strategy
Top-k	Sample from k-most-probable
Top-p	Cumulative probability at most p
ϵ	Probability at least ϵ
η	Min prob. proportional to entropy
Min-p	Prob. at least p_{min} scaled by max token prob.

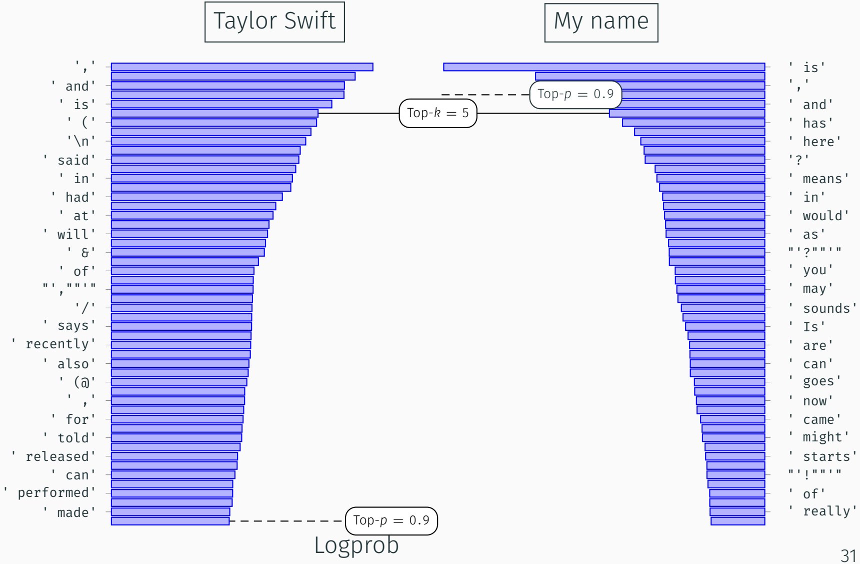
Truncation sampling



Truncation sampling



Truncation sampling



Temperature Sampling

Instead of truncating the tail, make the distribution more "peaked".

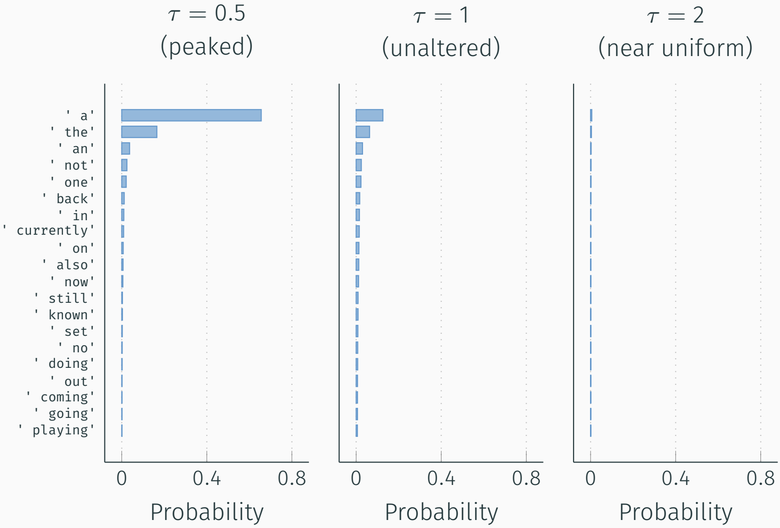
$$\text{softmax}(\boldsymbol{x}, \tau) = \frac{\exp(\boldsymbol{x}/\tau)}{\sum_i \exp(x_i/\tau)}$$

Temperature	Parameter	Pro	Con
High	$\tau \geq 1$	Diverse	Incoherent
Low	$\tau < 1$	Coherent	Repetitive

Temperature Sampling

Taylor Swift is...

softmax$(\boldsymbol{x}/\tau)$



Sampling implementations

```
probs = model(sequence)

# Greedy
indices, weights = probs.argmax(keepdim=True), None

# Ancestral
indices, weights = vocab_size, probs

# Top-k
topk = probs.topk(k)
indices, weights = topk.indices, topk.values

# Top-p
argsort = probs.argsort(descending=True)
top_p = (argsort.values.cumsum() < p).sum() + 1
indices, weights = argsort.indices[:top_p], argsort.values[:top_p]

# Epsilon
indices, weights = vocab_size, probs * (probs > epsilon)

# Temperature
indices, weights = vocab_size, (logits / temp).softmax(-1)

# Sample
next_token = random.choices(indices, weights=weights, k=1)
```

Batteries-included inference frameworks

```python
# vLLM
from vllm import LLM, SamplingParams
llm = LLM(model="facebook/opt-125m")
prompts = ["Hello, my name is"]
sampling_params = SamplingParams(temperature=0.8, top_p=0.95)
outputs = llm.generate(prompts, sampling_params)

# Huggingface
from transformers import AutoModelForCausalLM, AutoTokenizer
model = AutoModelForCausalLM.from_pretrained("gpt2")
tokenizer = AutoTokenizer.from_pretrained("gpt2")
text = "Hello, my name is"
tokens = tokenizer(text, return_tensors="pt")
output = model(**tokens).generate(
    temperature=0.8, top_p=0.95, do_sample=True
)
```

Why are next-token distributions heavy-tailed?

- Under-training

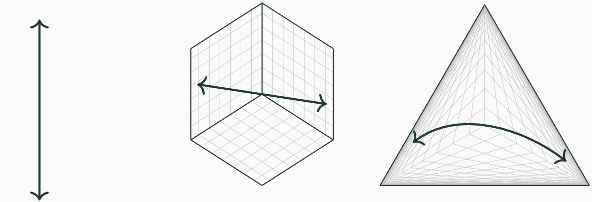
Why are next-token distributions heavy-tailed?

- Under-training
- Mode-seeking: cross-entropy loss punishes probability *underestimation* more than overestimation.

Why are next-token distributions heavy-tailed?

- Under-training
- Mode-seeking: cross-entropy loss punishes probability *underestimation* more than overestimation.
- By *design* low-rank constraints on the LLM outputs [Finlayson et al., 2024].

$$\mathbb{R}^{\text{embed}} \xrightarrow{\text{unembed}} \mathbb{R}^{\text{vocab}} \xrightarrow{\text{softmax}} \Delta_{\text{vocab}}$$



Sampling adapters

A sampling adapter takes a token distribution $p_\theta(\cdot \mid x)$ and re-adjusts the probabilities.

- Truncation and temperature are adapters.

Sampling adapters

A sampling adapter takes a token distribution $p_\theta(\cdot \mid x)$ and re-adjusts the probabilities.

- Truncation and temperature are adapters.
- Contrastive decoding [Li et al., 2023a, Liu et al., 2021]

$$p(\cdot \mid x) \propto \frac{p_{\text{expert}}(\cdot \mid x)}{p_{\text{antiexpert}}(\cdot \mid x)}$$

Sampling adapters

A sampling adapter takes a token distribution $p_\theta(\cdot \mid x)$ and re-adjusts the probabilities.

- Truncation and temperature are adapters.
- Contrastive decoding [Li et al., 2023a, Liu et al., 2021]

$$p(\cdot \mid x) \propto \frac{p_{\text{expert}}(\cdot \mid x)}{p_{\text{antiexpert}}(\cdot \mid x)}$$

- Many others

Method	Purpose	Adapter
Ancestral sampling	$y \sim p_\theta$	–
Temperature sampling [Ackley et al., 1985]	$y \sim q(p_\theta)$	Rescale
Greedy decoding	$y \leftarrow \max p_\theta$	Argmax (temperature$\rightarrow$ 0)
Top-k sampling [Fan et al., 2018]	$y \sim q(p_\theta)$	Truncation (top-k)
Nucleus sampling [Holtzman et al., 2020]	$y \sim q(p_\theta)$	Truncation (cumulative prob.)
Typical sampling [Meister et al., 2023]	$y \sim q(p_\theta)$	Truncation (entropy)
Epsilon sampling [Hewitt et al., 2022]	$y \sim q(p_\theta)$	Truncation (probability)
η sampling [Hewitt et al., 2022]	$y \sim q(p_\theta)$	Truncation (prob. and entropy)
Mirostat decoding [Basu et al., 2021]	Target perplexity	Truncation (adaptive top-k)
Basis-aware sampling [Finlayson et al., 2024]	$y \sim q(p_\theta)$	Truncation (linear program)
Contrastive decoding [Li et al., 2023a]	$y \sim q(p_\theta)$	$\log p_{\theta'} - \log p_\theta$ and truncation
DExperts [Liu et al., 2021]	$y \sim q_*(\cdot \mid x, c)$	$\propto p_\theta \cdot (p_{\theta+}/p_{\theta-})^\alpha$
Inference-time adapters [Lu et al., 2023]	$y \sim q_* \propto r(y)$	$\propto (p_\theta \cdot p_{\theta'})^\alpha$
Proxy tuning [Liu et al., 2024]	$y \sim q_*(\cdot \mid x, c)$	$\propto p_\theta \cdot (p_{\theta+}/p_{\theta-})^\alpha$

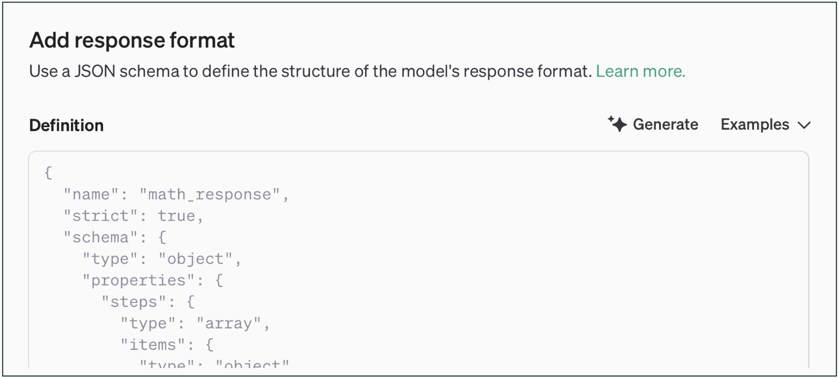
I. Primitive Generators

Constrained decoding

Constrained decoding

Embedding LLMs in larger systems requires that they can *communicate* with the larger system, e.g., with JSON.

Can we force LLMs to generate **structured outputs?**



From OpenAI Playground.

Worked example: decoding valid JSON

Language models can stuggle with controlled and structured generation. Prompt:

Key	Type
`name`	string
`birth year`	int

> *Format the following information using the JSON schema: "Taylor Swift was born December 13, 1989."*

Worked example: decoding valid JSON

Language models can stuggle with controlled and structured generation. Prompt:

Key	Type
`name`	string
`birth year`	int

Format the following information using the JSON schema: ”Taylor Swift was born December 13, 1989.”

LLM:

```
{"name": "Taylor Swift", "birth": "1998-12-13T01:00:00Z", "age…
```

The LLM output does not match the JSON schema.

Format the following information using the JSON schema:

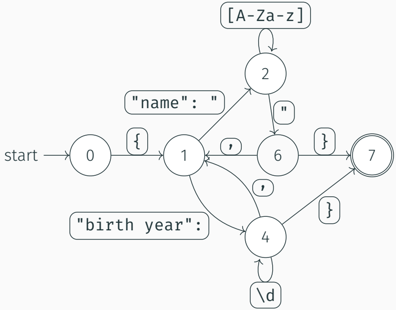
Taylor Swift was born December 13, 1989.

Key	Type
`name`	string
`birth year`	int

Format the following information using the JSON schema:

Taylor Swift was born December 13, 1989.

Key	Type
`name`	string
`birth year`	int

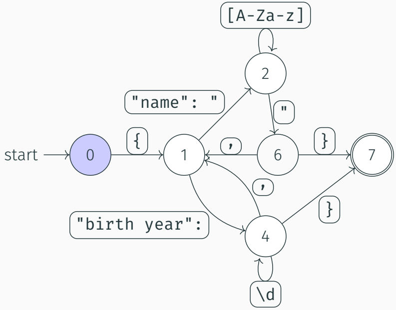


1. Compile the schema into a state machine.

Format the following information using the JSON schema:

Taylor Swift was born December 13, 1989.

Key	Type
`name`	string
`birth year`	int



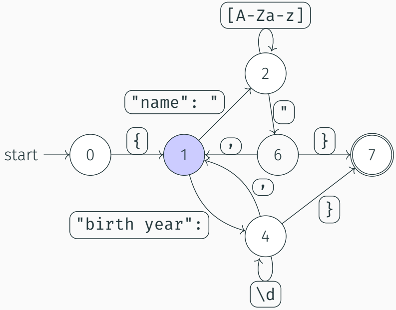
1. Compile the schema into a state machine.
2. Filter the next-token distribution for valid tokens.

Token	Prob.
`\n`	~~0.36~~
`"`	~~0.16~~
`{`	0.026
`https`	~~0.025~~
…	…

Format the following information using the JSON schema:

Taylor Swift was born December 13, 1989.

Key	Type
`name`	string
`birth year`	int



1. Compile the schema into a state machine.
2. Filter the next-token distribution for valid tokens.

GPT2:

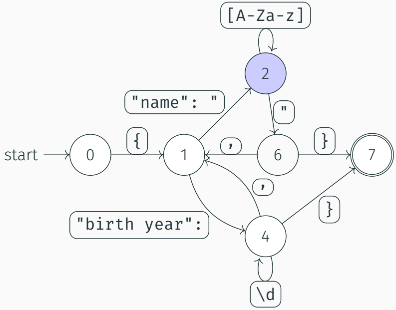
```
{
```

Token	Prob.
`name`	~~0.31~~
`date`	~~0.069~~
`"`	0.039
`id`	~~0.033~~
...	...

Format the following information using the JSON schema:

Taylor Swift was born December 13, 1989.

Key	Type
`name`	string
`birth year`	int



1. Compile the schema into a state machine.
2. Filter the next-token distribution for valid tokens.

GPT2:

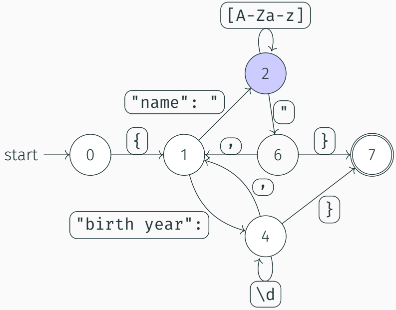
```
{"name": "
```

Token	Prob.
`Taylor`	0.85
`T`	0.034
`S`	0.024
`The`	0.022
...	...

Format the following information using the JSON schema:

Taylor Swift was born December 13, 1989.

Key	Type
`name`	string
`birth year`	int



1. Compile the schema into a state machine.
2. Filter the next-token distribution for valid tokens.

GPT2:

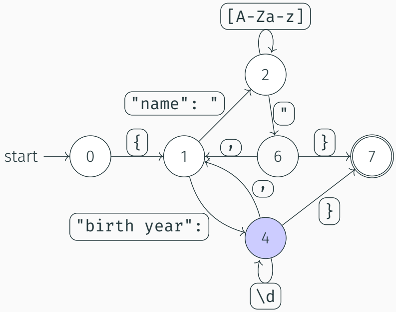
```
{"name": "Taylor Swift
```

Token	Prob.
`",`	0.85
`,"`	~~0.034~~
`"`	0.024
`,`	~~0.022~~
...	...

Format the following information using the JSON schema:

Taylor Swift was born December 13, 1989.

Key	Type
`name`	string
`birth year`	int



1. Compile the schema into a state machine.
2. Filter the next-token distribution for valid tokens.

GPT2:

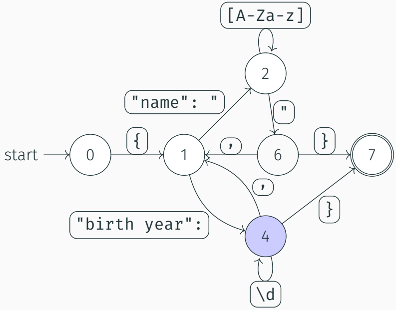
```
{"name": "Taylor Swift", "birth
year":
```

Token	Prob.
`"`	~~0.46~~
`int`	~~0.041~~
`'`	~~0.026~~
`1989`	0.020
…	…

Format the following information using the JSON schema:

Taylor Swift was born December 13, 1989.

Key	Type
`name`	string
`birth year`	int



1. Compile the schema into a state machine.
2. Filter the next-token distribution for valid tokens.

GPT2:

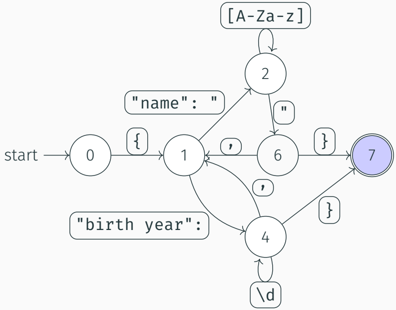
```
{"name": "Taylor Swift", "birth
year": 1989
```

Token	Prob.
,	0.39
}	0.34
},	~~0.11~~
}	0.082
...	...

Format the following information using the JSON schema:

Taylor Swift was born December 13, 1989.

Key	Type
`name`	string
`birth year`	int



1. Compile the schema into a state machine.
2. Filter the next-token distribution for valid tokens.

GPT2:

```
 {"name": "Taylor Swift", "birth
year": 1989}
```

Side effects of templated/constrained decoding

- Generation speedup
- Reduced performance

Token healing

- Templated generation can force unnatural token boundaries

`The␣url␣is␣http:`

Token healing

- Templated generation can force unnatural token boundaries

$$\boxed{\texttt{The}}\boxed{\texttt{␣url}}\boxed{\texttt{␣is}}\boxed{\texttt{␣http}}\boxed{\texttt{:}}\boxed{\texttt{␣//}}$$

- The model has rarely seen the tokenization $\boxed{\texttt{␣http}}\boxed{\texttt{:}}\boxed{\texttt{//}}$ during training compared to $\boxed{\texttt{␣http}}\boxed{\texttt{://}}$.

Token healing

- Templated generation can force unnatural token boundaries

`The␣url␣is␣http:␣//`

- The model has rarely seen the tokenization `␣http` `:` `//` during training compared to `␣http` `://`.
- Token healing rewinds the tokenizer and enforces the untokenized text as a prefix to the next token.

`The␣url␣is␣http:`

Candidates
~~s://~~
://

Token healing

- Templated generation can force unnatural token boundaries

  `The` `␣url` `␣is` `␣http` `:` `␣//`

- The model has rarely seen the tokenization `␣http` `:` `//` during training compared to `␣http` `://`.
- Token healing rewinds the tokenizer and enforces the untokenized text as a prefix to the next token.

  `The` `␣url` `␣is` `␣http` `://`

Candidates
~~`s://`~~
`://`

- Alternative fix: tokenizer regularization during training [Kudo, 2018].

Summary

- Two views of decoding: optimization, sampling
- The diversity-coherence trade-off
- Constrained decoding enforces structure on LLM outputs

These are the building blocks of modern LLM generation methods.

Meta-generators

Goal (system designer)

Design a system G that generates acceptable sequences:

$$\arg\max_{G} \mathbb{E}_{y \sim G(\cdot)} A(y) \tag{1}$$

Example acceptability: correctness, human preferences

Goal (system designer)

Design a system G that generates acceptable sequences:

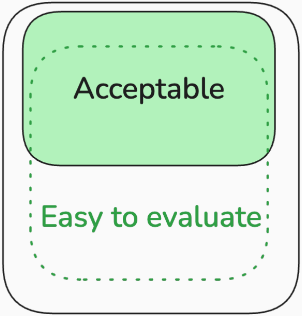
$$\arg\max_{G} \mathbb{E}_{y \sim G(\cdot)} A(y) \tag{1}$$

Example acceptability: correctness, human preferences

We know how to sample *probable* outputs, $y \sim p_\theta(y|x)$

- What if these outputs are not *acceptable*?

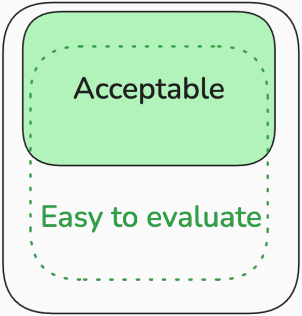
1. Take advantage of external information during generation



- Example: Learn an evaluator $v(y) \approx A(y)$ and use it in generation

Meta-generation | Key ideas

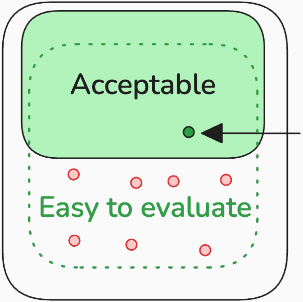
1. Take advantage of external information during generation



- Example: Learn an evaluator $v(y) \approx A(y)$ and use it in generation

Terminology: Evaluator ≈ critic ≈ verifier ≈ value ≈ reward model ≈ scoring model

1. Take advantage of external information during generation
2. Call the generator more than once to search for good sequences



x:

Input:
Let $f(r) = \sum_{j=2}^{2008} \frac{1}{j^r} = \frac{1}{2^r} + \frac{1}{3^r} + \cdots + \frac{1}{2008^r}$. Find $\sum_{k=2}^{\infty} f(k)$.

LLEMMA 34B solution:
We have

z:

$$\sum_{k=2}^{\infty} f(k) = \sum_{k=2}^{\infty}\sum_{j=2}^{2008} \frac{1}{j^k} = \sum_{j=2}^{2008}\sum_{k=2}^{\infty} \frac{1}{j^k} = \sum_{j=2}^{2008} \frac{1}{j^2} \sum_{k=0}^{\infty} \frac{1}{j^k} = \sum_{j=2}^{2008} \frac{1}{j^2} \frac{1}{1-\frac{1}{j}}$$
$$= \sum_{j=2}^{2008} \frac{1}{j(j-1)} = \sum_{j=2}^{2008} \left(\frac{1}{j-1} - \frac{1}{j}\right)$$
$$= \left(\frac{1}{1} - \frac{1}{2}\right) + \left(\frac{1}{2} - \frac{1}{3}\right) + \cdots + \left(\frac{1}{2007} - \frac{1}{2008}\right)$$
$$= 1 - \frac{1}{2008}$$
$$= \boxed{\frac{2007}{2008}}.$$

y: Final Answer: The final answer is $\frac{2007}{2008}$.

Example: solve a math problem

Meta-generation | Key ideas

What if we had an oracle verifier, $v(y)$?

Repeat:

- $z \sim p_\theta(z|x)$
- $y \sim p_\theta(y|x, z)$
- Stop if $v(y)$ says answer is correct

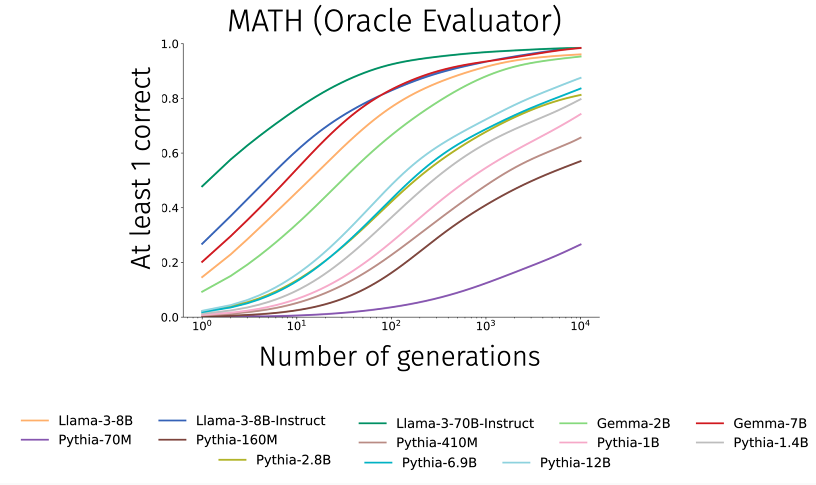
x: **Input:**
Let $f(r) = \sum_{j=2}^{2008} \frac{1}{j^r} = \frac{1}{2^r} + \frac{1}{3^r} + \cdots + \frac{1}{2008^r}$. Find $\sum_{k=2}^{\infty} f(k)$.

Llemma 34B solution:
We have

z:
$$\begin{aligned}\sum_{k=2}^{\infty} f(k) &= \sum_{k=2}^{\infty}\sum_{j=2}^{2008} \frac{1}{j^k} = \sum_{j=2}^{2008}\sum_{k=2}^{\infty} \frac{1}{j^k} = \sum_{j=2}^{2008} \frac{1}{j^2}\sum_{k=0}^{\infty} \frac{1}{j^k} = \sum_{j=2}^{2008} \frac{1}{j^2}\frac{1}{1-\frac{1}{j}} \\ &= \sum_{j=2}^{2008} \frac{1}{j(j-1)} = \sum_{j=2}^{2008}\left(\frac{1}{j-1} - \frac{1}{j}\right) \\ &= \left(\frac{1}{1} - \frac{1}{2}\right) + \left(\frac{1}{2} - \frac{1}{3}\right) + \cdots + \left(\frac{1}{2007} - \frac{1}{2008}\right) \\ &= 1 - \frac{1}{2008} \\ &= \boxed{\frac{2007}{2008}}.\end{aligned}$$

y: Final Answer: The final answer is $\frac{2007}{2008}$.

Meta-generation | Key ideas[1]



[1] Adapted from [Brown et al., 2024]. See also [Li et al., 2022, Cobbe et al., 2021, Jiang et al., 2023]

Meta-generation

We formalize these kinds of strategies as *meta-generators* [2]

$$y \sim G(y|x; \underbrace{g_1, g_2, \ldots, g_G}_{\text{generators}}, \underbrace{\phi}_{\text{Other parameters}})$$

Key design choices:

- G: strategy for calling generators
- $g_1, g_2, \ldots, g_G$: choice of generators
- ϕ: other models, number of tokens to generate, ...

[2][Welleck et al., 2024] *From Decoding to Meta-Generation: Inference-time Algorithms for LLMs.* S. Welleck, A. Bertsch*, M. Finlayson*, H. Schoelkopf*, A. Xie, G. Neubig, I. Kulikov, Z. Harchaoui.

Meta-generation

Token-level generators from part 1 are a special case of calling:

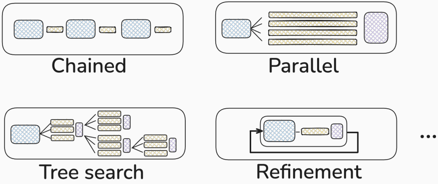
$$y \sim g(y|x; p_\theta, \phi)$$

Design choices:

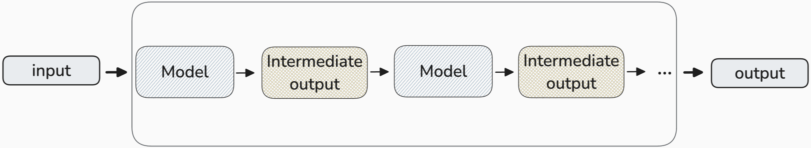
- g: sampling adapters, beam search, ...
- ϕ: temperature, beam width, ...

Meta-generators | outline

- Strategies
  - **Chain**
  - Parallel
  - Tree search
  - Refinement/Self-Correction
- Scaling meta-generators



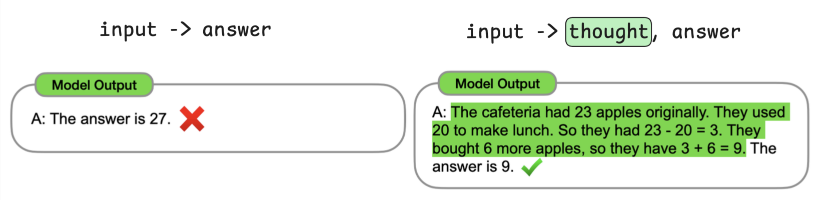
Meta-generators | chain



Compose generators:

$$
\begin{aligned}
y_1 &\sim g_1(x) \\
y_2 &\sim g_2(x, y_1) \\
y_3 &\sim g_3(x, y_2) \\
&\vdots
\end{aligned}
$$

Motivating example: *Chain-of-thought* [Wei et al., 2022]:

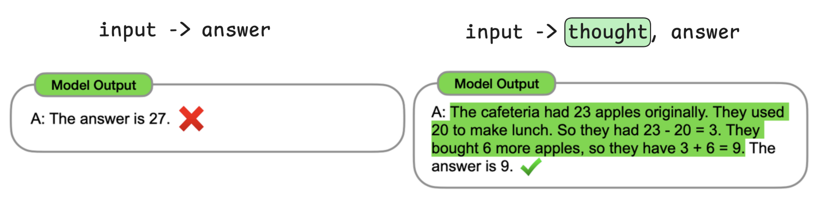


A simple decomposition:

- Generate a thought, $z \sim g(\cdot|x)$
- Generate an answer, $a \sim g(\cdot|x, z)$

Meta-generators | chain

Motivating example: *Chain-of-thought* [Wei et al., 2022]:



Increases expressivity[3]

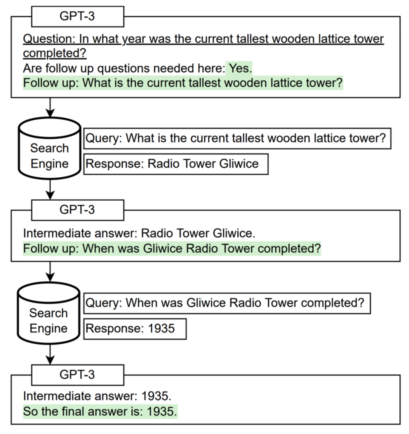
- Variable output length, analogous to a writeable tape

[3] E.g., [Feng et al., 2023, Merrill and Sabharwal, 2024, Nowak et al., 2024]

Meta-generators | chain

Extend to multiple steps:

- Each step:
  - Generate query
  - Call API
- Then generate an answer



Self-Ask [Press et al., 2023]

Meta-generators | chain[4]

View as programs:

- Outer function ≈ meta-generator
- Inner function ≈ generator

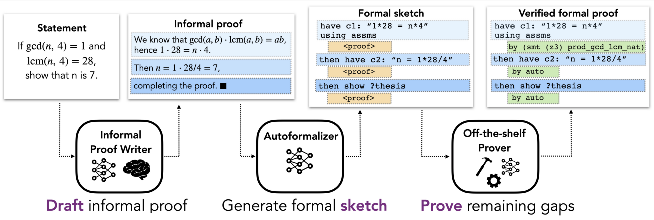
```
def search(x: Example) -> Example:
  x.hop1 = generate(hop_template)(x).pred
  x.psg1 = retrieve(x.hop1, k=1)[0]
  x.hop2 = generate(hop_template)(x).pred
  x.psg2 = retrieve(x.hop2, k=1)[0]
  return x

def predict(x: Example) -> Example:
  x.context = [x.psg1, x.psg2]
  x.pred = generate(qa_template)(x).pred
  return x
```

Demonstrate-Search-Predict (DSP)
[Khattab et al., 2022]

[4] [Khattab et al., 2022, Dohan et al., 2022, Schlag et al., 2023, Zheng et al., 2024]

Many other examples!

- Rewrite input before generating
  (*System-2 Attention* [Weston and Sukhbaatar, 2023])
- Sketch proof, fill gaps, check proof
  (*Draft-Sketch-Prove* [Jiang et al., 2023])
- ...

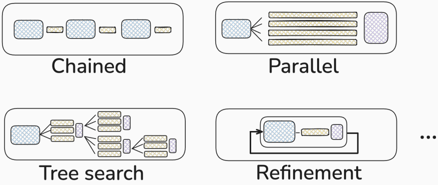


Meta-generators | chain

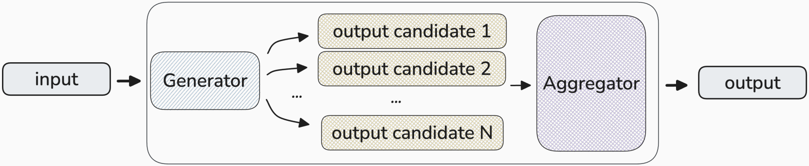
Chained meta-generation

- **Key idea:** decompose generation and incorporate tools/models
- Chaining alone does not explore the output space

- Strategies
  - Chain
  - Parallel
  - Tree search
  - Refinement



Meta-generators | parallel



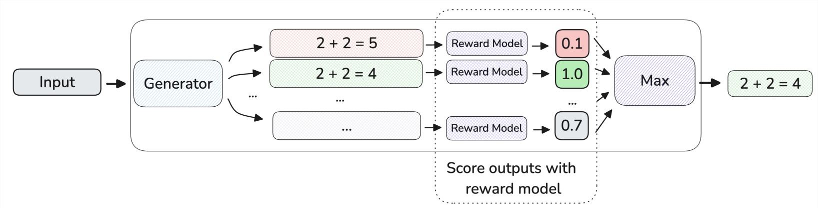
- Generate candidates:

$$\{y^{(1)}, \ldots, y^{(N)}\} \sim G(\cdot|x)$$

- Aggregate:

$$y = h(y^{(1)}, \ldots, y^{(N)})$$

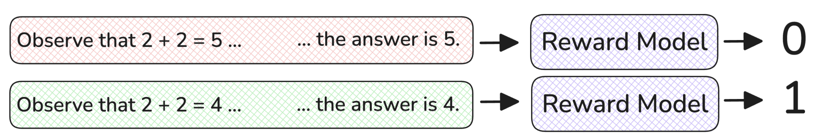
Meta-generators | parallel | Best-of-N/Rejection Sampling[5]



$$\underset{\{y^{(1)},\ldots,y^{(N)}\}}{\arg\max} \underbrace{v(y)}_{\text{reward model}}$$

[5] [Stiennon et al., 2020, Nakano et al., 2022]

Reward model $v(y) \rightarrow [0, 1]$:

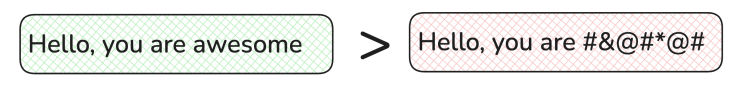


Train reward model with correct and incorrect examples.[6]

[6] E.g., [Cobbe et al., 2021]

Meta-generators | parallel | Best-of-N/Rejection Sampling

Reward model $v(y) \to [0, 1]$:



Train reward model with preference data.[6]

[6] E.g., [Stiennon et al., 2020]

Meta-generators | parallel | Best-of-N/Rejection Sampling

Why Best-of-N?

- Approximates maximum acceptability:

$$\begin{aligned}
\text{Best-of-}N &= \arg\max_{y \in \{y^{(1)},\ldots,y^{(N)}\}} v(y) \\
&\approx \arg\max_{y} v(y) \qquad (2)\\
&\approx \arg\max_{y} A(y) \qquad (3)
\end{aligned}$$

Why Best-of-N?

- Approximates maximum acceptability:

$$\begin{aligned} \text{Best-of-}N &= \underset{y \in \{y^{(1)},\ldots,y^{(N)}\}}{\arg\max}\; v(y) \\ &\approx \arg\max_y v(y) && (2) \\ &\approx \arg\max_y A(y) && (3) \end{aligned}$$

(2) gets better as number of generations N increases!

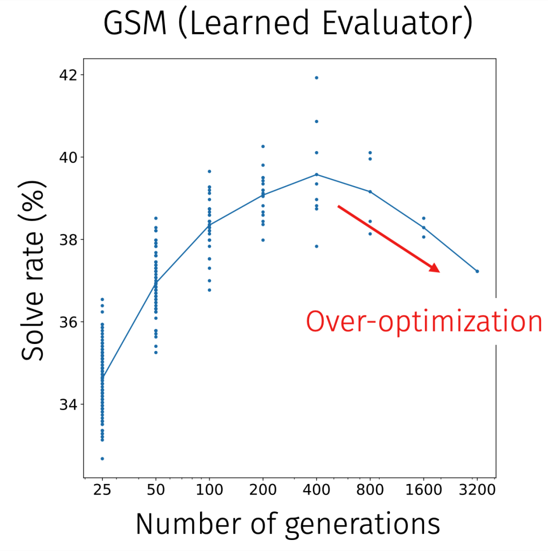
Why Best-of-N?

- Approximates maximum acceptability:

$$\begin{aligned} \text{Best-of-}N &= \operatorname*{arg\,max}_{y \in \{y^{(1)},\ldots,y^{(N)}\}} v(y) \\ &\approx \operatorname*{arg\,max}_{y} v(y) \qquad (2) \\ &\approx \operatorname*{arg\,max}_{y} A(y) \qquad (3) \end{aligned}$$

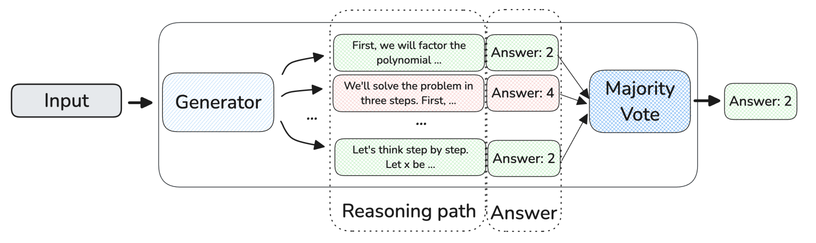
(2) gets better as number of generations N increases!

(3) Suffers from imperfect reward model, aka “over-optimization”



[7] Plot adapted from *Training Verifiers to Solve Math Word Problems* [Cobbe et al., 2021]

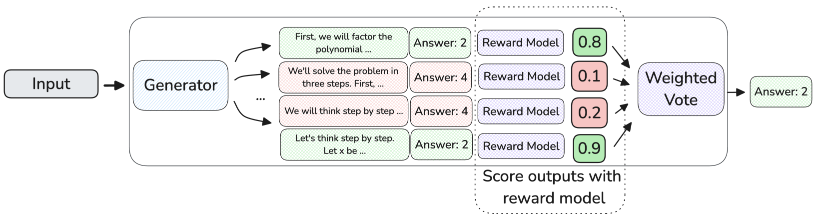
Voting aggregation:[8]



$$\arg\max_{a} \sum_{i=1}^{N} 1\{y^{(i)} = a\},$$

8[Wang et al., 2023]

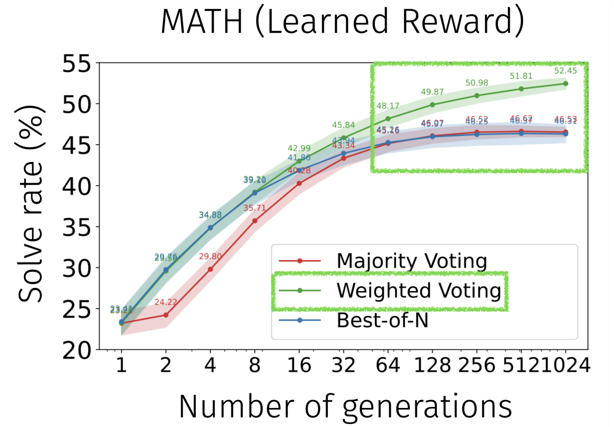
Weighted Voting:



$$\arg\max_{a} \sum_{i=1}^{N} \underbrace{v(y^{(i)})}_{\text{reward model}} \cdot \mathbb{1}\{y^{(i)} = a\},$$

[9] [Li et al., 2023b]

Can outperform Best-of-N, e.g.:[10]



[10][Sun et al., 2024] *Easy-to-Hard Generalization: Scalable Alignment Beyond Human Supervision.* Z. Sun, L. Yu, Y. Shen, W. Liu, Y. Yang, S. Welleck, C. Gan. NeurIPS 2024.

Meta-generators | parallel | why (weighted) voting?

As the number of candidates $N \to \infty$, voting accuracy converges to...[11]

$$\frac{1}{M}\sum_{i=1}^{M}\mathbb{I}\left[a_i^* = \arg\max_a \underbrace{\sum_z v(x,z,a)g(z,a|x)}_{\text{“Marginalize out paths } z\text{”}}\right]$$

Notation:

- (x, z, a): (input, solution, answer)
- M: number of test examples

[11] Theorem 2, [Wu et al., 2024b] *Inference Scaling Laws.* Y. Wu, Z. Sun, S. Li, S. Welleck, Y. Yang.

Meta-generators | parallel | why (weighted) voting?

As the number of candidates $N \to \infty$, voting accuracy converges to...[11]

$$\frac{1}{M}\sum_{i=1}^{M}\mathbb{I}\left[a_i^* = \arg\max_a \underbrace{\sum_z v(x,z,a)g(z,a|x)}_{\text{“Marginalize out paths } z\text{”}}\right]$$

Takeaway 1: Will accuracy keep improving with more samples?

- **No**, it eventually converges to the accuracy shown above

[11] Theorem 2, [Wu et al., 2024b] *Inference Scaling Laws.* Y. Wu, Z. Sun, S. Li, S. Welleck, Y. Yang.

Meta-generators | parallel | why (weighted) voting?

As the number of candidates $N \to \infty$, voting accuracy converges to...[11]

$$\frac{1}{M}\sum_{i=1}^{M}\mathbb{I}\left[a_i^* = \arg\max_a \underbrace{\sum_z v(x,z,a)g(z,a|x)}_{\text{“Marginalize out paths } z\text{”}}\right]$$

Takeaway 2: When is weighted voting better than voting?

- When $v \cdot g$ assigns more total mass to correct answers than g

[11] Theorem 2, [Wu et al., 2024b] *Inference Scaling Laws.* Y. Wu, Z. Sun, S. Li, S. Welleck, Y. Yang.

Meta-generators | parallel | why (weighted) voting?

As the number of candidates $N \to \infty$, voting accuracy converges to...[11]

$$\frac{1}{M}\sum_{i=1}^{M}\mathbb{I}\left[a_i^* = \arg\max_a \underbrace{\sum_z v(x,z,a)g(z,a|x)}_{\text{“Marginalize out paths } z\text{”}}\right]$$

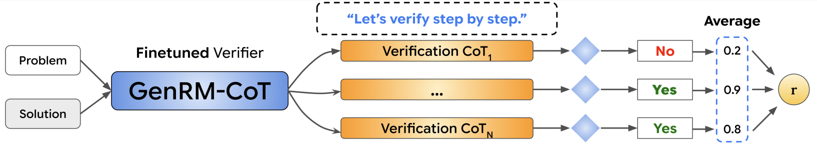
Takeaway 3: How do we improve performance further?

- Improve the reward model v
- Improve the generator g (better model and/or better algorithm)

[11] Theorem 2, [Wu et al., 2024b] *Inference Scaling Laws.* Y. Wu, Z. Sun, S. Li, S. Welleck, Y. Yang.

Meta-generators | parallel

Improve the reward model:



Parallel generation *in the reward model too*[12]

Active area of research!

[12] [Zhang et al., 2024]

Meta-generators | parallel

Parallel meta-generators

- Explores output space by generating full sequences
- Large performance gains in practice
- Bounded by the quality of the evaluator and generator

Meta-generators | parallel

Parallel meta-generators

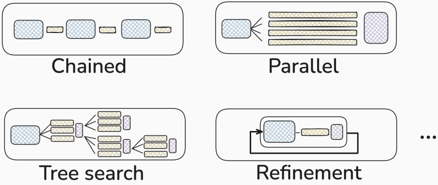
- Explores output space by generating full sequences
- Large performance gains in practice
- Bounded by the quality of the evaluator and generator

Insight: only uses the verifier at the end (on full sequences)

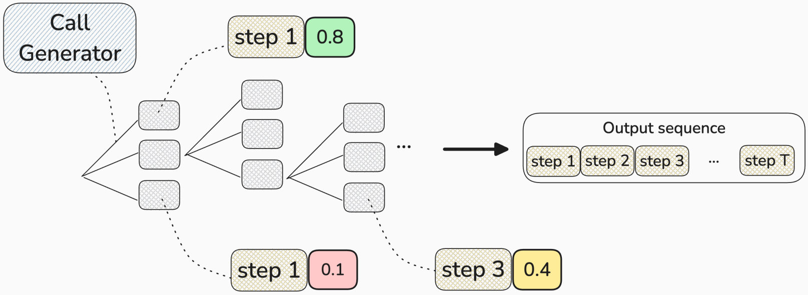
- *Next:* Can we better leverage *intermediate* evaluation?

Meta-generators | outline

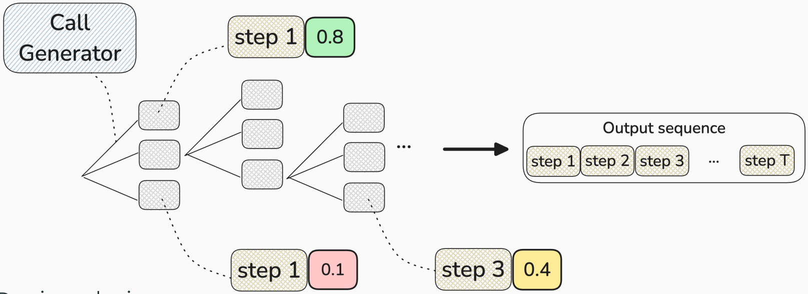
- Strategies
  - Chain
  - Parallel
  - Tree search
  - Refinement



Meta-generators | tree search | basic idea



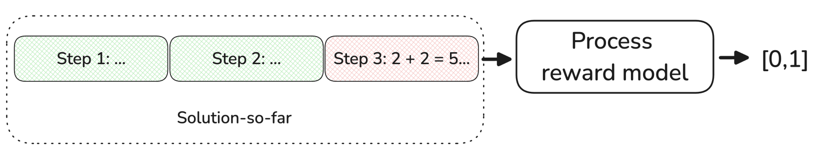
Meta-generators | tree search | basic idea



Design choices:

- States s
- Transitions $s \to s'$
- Scores $v(s)$
- Strategy (breadth-first, depth-first, ...)

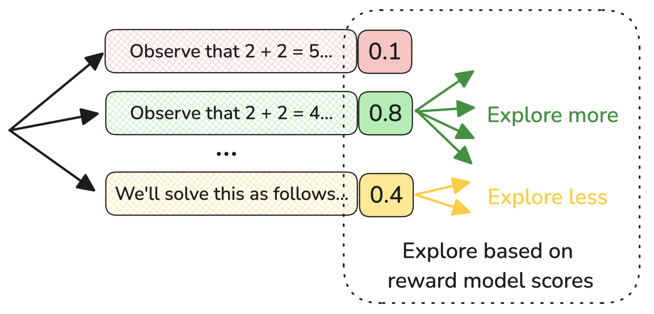
1. Scores: “process reward model (PRM)”[13]



$$v(x, s_1, s_2, \ldots, s_t) \rightarrow [0, 1]$$

[13] [Uesato et al., 2022, Lightman et al., 2024, Wang et al., 2024a]

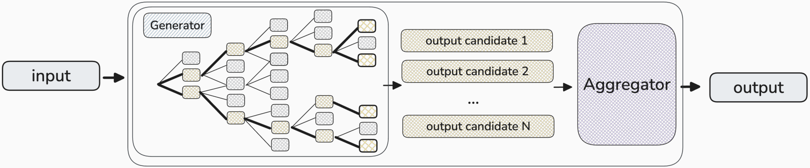
2. *Reward Balanced Search (Rebase)*[14]



$$\text{explore}_i = \text{Round}\left(\text{Budget}\frac{\exp\left(v(s_i)/\tau\right)}{\sum_j \exp\left(v(s_j)/\tau\right)}\right), \tag{4}$$

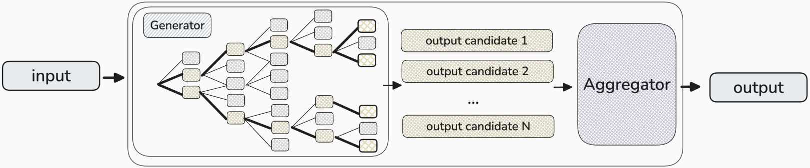
[14] [Wu et al., 2024b] *Inference Scaling Laws: An Empirical Analysis of Compute-Optimal Inference.*

Meta-generators | tree search | aggregation



Run tree search to get candidates for aggregation (e.g., voting).

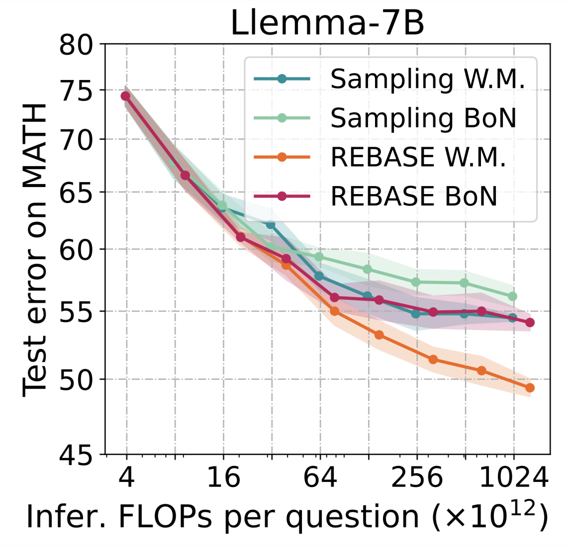
Meta-generators | tree search | aggregation



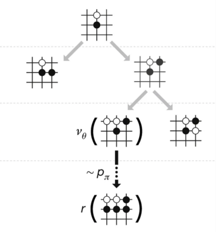
Run tree search to get candidates for aggregation (e.g., voting).

- **Key idea:** Leverages scores on *intermediate* states
  - Backtracking
  - Exploration

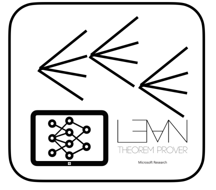
Meta-generators | tree search | example[15]



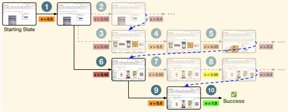
[15] [Wu et al., 2024b] *Inference Scaling Laws: An Empirical Analysis of Compute-Optimal Inference.*



Go [Silver et al., 2016]



Proofs [Polu and Sutskever, 2020]



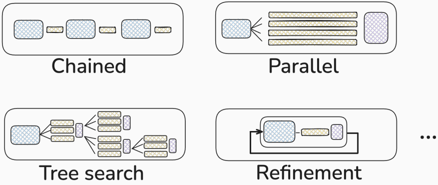
Agents [Koh et al., 2024]

Meta-generators | tree search

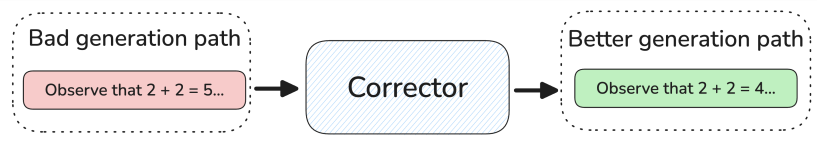
Tree-search meta-generators

- Can backtrack and explore using intermediate scores
- Requires a suitable environment and value function
  - Decomposition into states
  - Good reward signal

- Strategies
    - Chain
    - Parallel
    - Tree search
    - Refinement/self-correction
- Scaling meta-generators

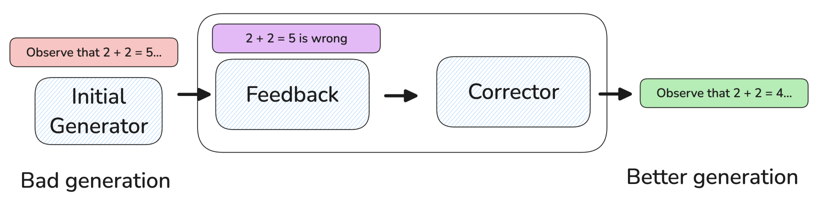


Meta-generators | refinement / self-correction



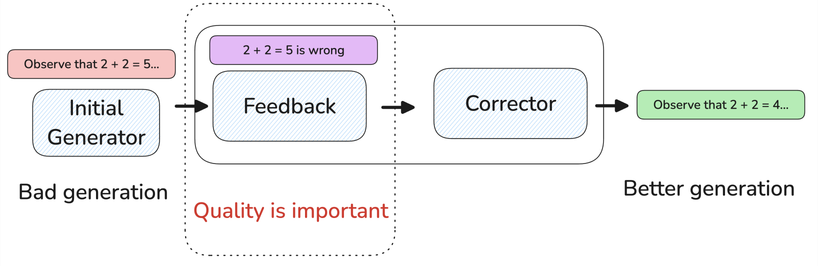
Improve a generation

Meta-generators | refinement / self-correction



Improve a generation using feedback

Meta-generators | refinement / self-correction



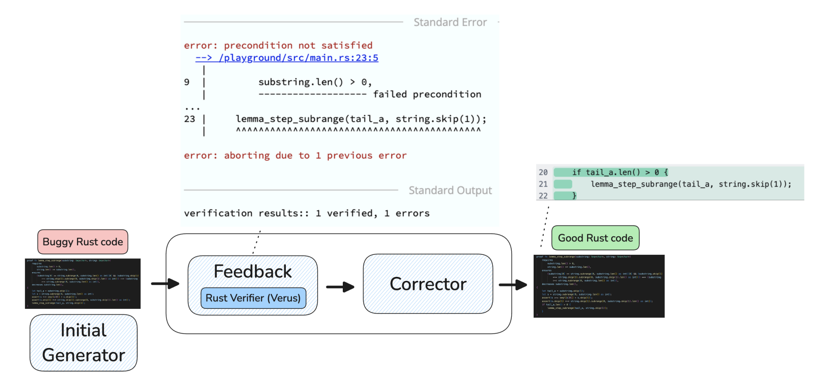
Improve a generation using feedback

Meta-generators | refinement

In practice, the quality and source of feedback is crucial:

- **Extrinsic**: external information at inference time
- **Intrinsic**: no external information at inference time

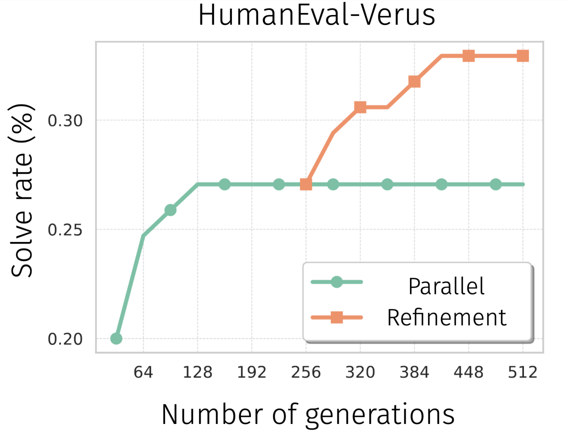
1. **Extrinsic**: external feedback



Feedback: external program verifier[16]

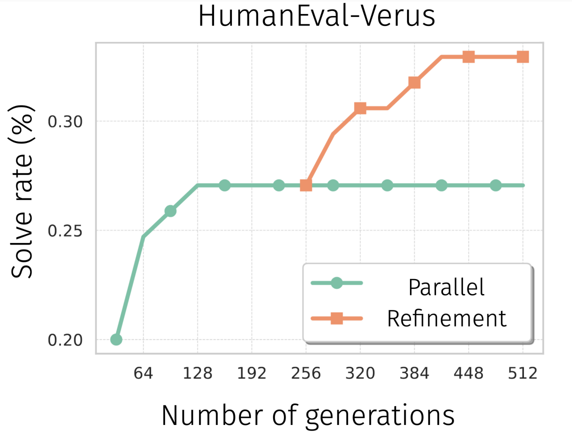
[16] [Aggarwal et al., 2024], *AlphaVerus*. P. Aggarwal, B. Parno, S. Welleck.

1. **Extrinsic**: external feedback



AlphaVerus. P. Aggarwal, B. Parno, S. Welleck.

1. **Extrinsic**: external feedback



Tutorial code demo: github.com/cmu-l3/neurips2024-inference-tutorial-code

1. **Extrinsic**: external feedback

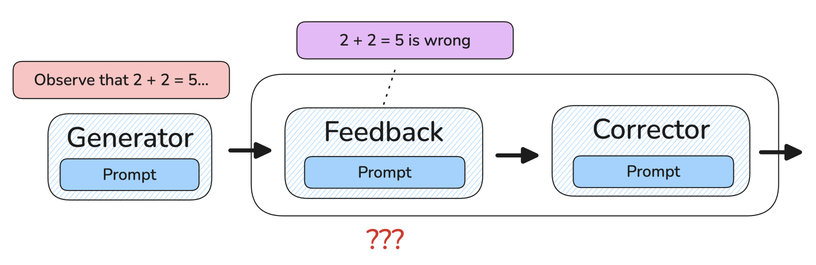
Several success cases:

- Verifiers [Aggarwal et al., 2024]
- Code interpreters [Chen et al., 2024b]
- Retrievers [Asai et al., 2024]
- Tools + agent environment[16]
- ...

Intuition: adds new information, can detect and localize errors

[16] https://x.com/gneubig/status/1866172948991615177

2. **Intrinsic**: Re-prompt the same model:



Re-prompt a single LLM, e.g. [Madaan et al., 2023]

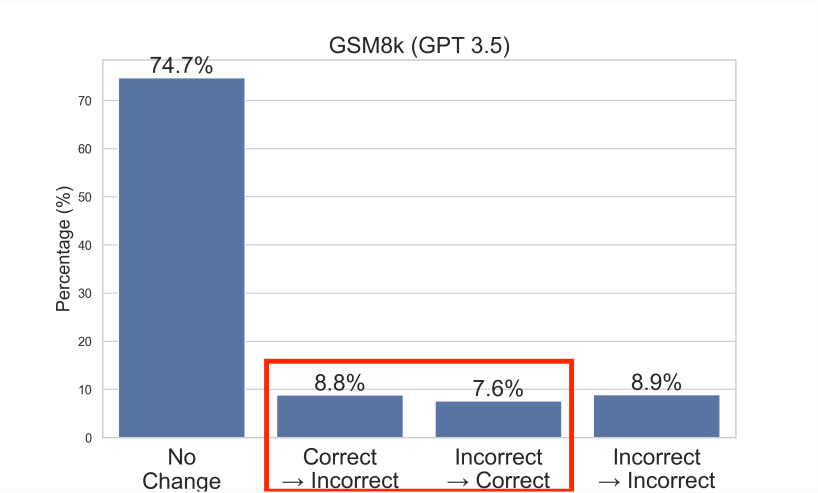
Meta-generators | refinement | intrinsic

Mixed results:

- Easy to evaluate tasks: positive [Wang et al., 2024b]
    - E.g., missing info [Asai et al., 2024]
- Mathematical reasoning: mixed[17]

[17] E.g., [Huang et al., 2024] *Large Language Models Cannot Self-Correct Reasoning Yet*

Meta-generators | refinement | intrinsic

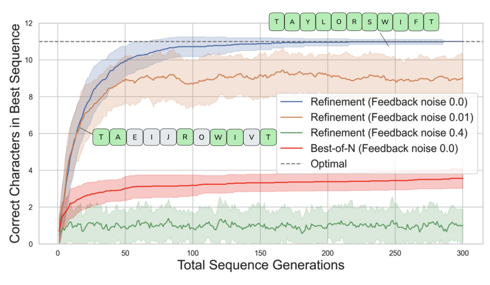


Takeaway: feedback is too noisy From [Huang et al., 2024]

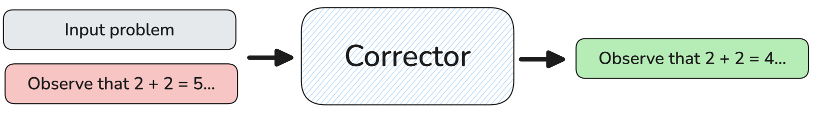
Meta-generators | refinement

Generate "TAYLORSWIFT"

- Generator:
  - p(character)
- Feedback:
  - Incorrect characters
- Corrector:
  - Regenerate incorrect



3. **Intrinsic**: trained corrector



Directly *learn* to correct[17]

[17][Welleck et al., 2023], *Generating Sequences by Learning to [Self-]Correct.*

Meta-generators | refinement | Case 3: fine-tuning

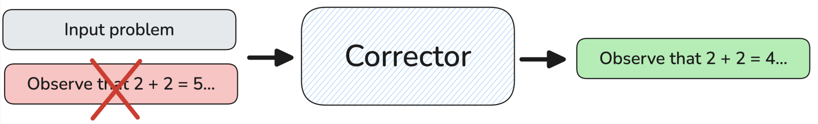
General pattern:[18]

- Collect (bad, better) pairs by generating and evaluating reward
- Update corrector $p_\theta(better|bad)$ using the collected data
- Repeat

[18] E.g., Self-corrective learning [Welleck et al., 2023], SCoRe [Kumar et al., 2024].

General pattern:[18]

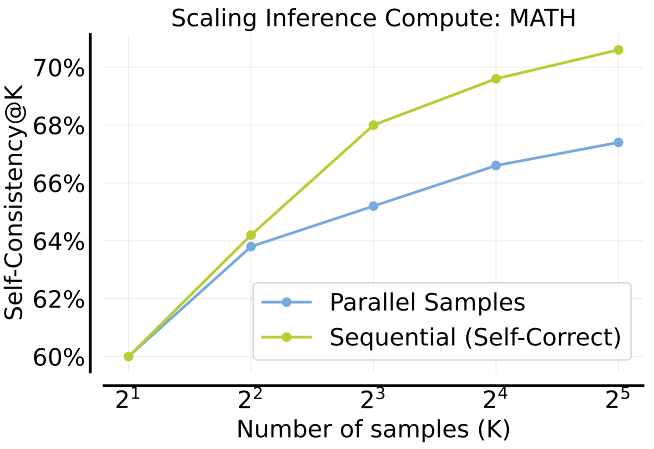
- Collect (bad, better) pairs by generating and evaluating reward
- Update corrector $p_\theta(better|bad)$ using the collected data
- Repeat



Prone to *behavior collapse*

- [Kumar et al., 2024]: overcome with regularization + RL

[18] E.g., Self-corrective learning [Welleck et al., 2023], SCoRe [Kumar et al., 2024].



From *SCoRe* [Kumar et al., 2024]

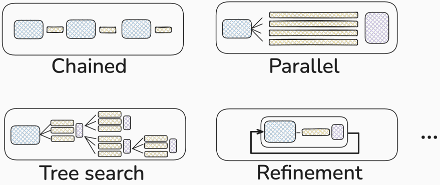
Meta-generators | refinement / self-correction

Refinement / self-correction

- Extrinsic
  - Positive results for environments that detect or localize errors
- Intrinsic, prompted
  - Mixed results, depends on difficulty of verification
- Intrinsic, trained
  - Possible improvements, requires specific training strategies

This talk:

- Strategies
  - Chain
  - Parallel
  - Tree search
  - Refinement
- Scaling meta-generators

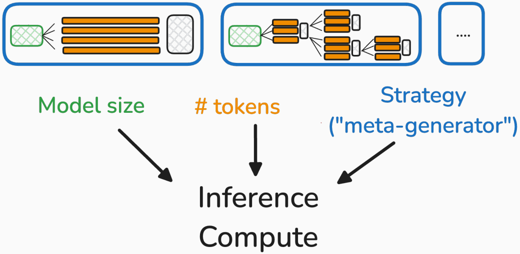


Meta-generation | how do we allocate test-time compute?

Choose strategies based on **task performance** and **compute cost**

Cost is a function of:

- Model size
- Number of generated tokens



Meta-generation | compute-optimal inference[19]

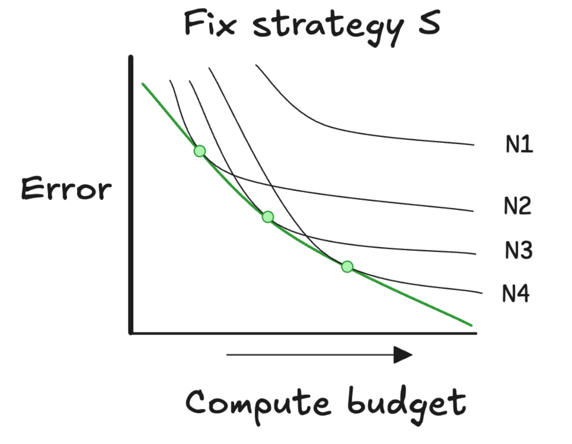
For a compute budget C:

$$\mathrm{argmin}_{N,T,S \text{ s.t. } cost(N,T,S)=C}\mathrm{error}(N,T,S)$$

- N: number of model parameters
- T: number of generated tokens
- S: inference strategy
- $cost(N,T,S)$: in floating-point operations

[19][Wu et al., 2024b] *Inference Scaling Laws: An Empirical Analysis of Compute-Optimal Inference.*

Meta-generation | compute-optimal inference[20]



Choose configurations on the *compute-optimal frontier* (green)

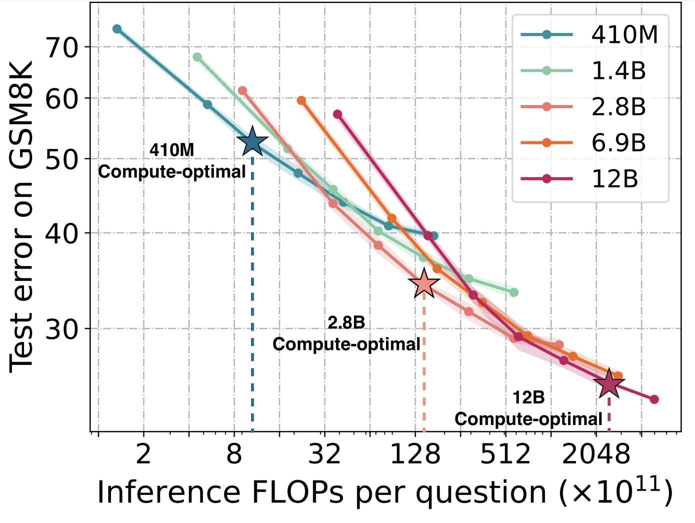
[20][Wu et al., 2024b] *Inference Scaling Laws: An Empirical Analysis of Compute-Optimal Inference.*

Meta-generation | how do we choose a meta-generator?

Question 1: is it better to use:

- A **small** model and more generations
- A **large** model and fewer generations

Experiment: Fix strategy, vary model size N and number of tokens T



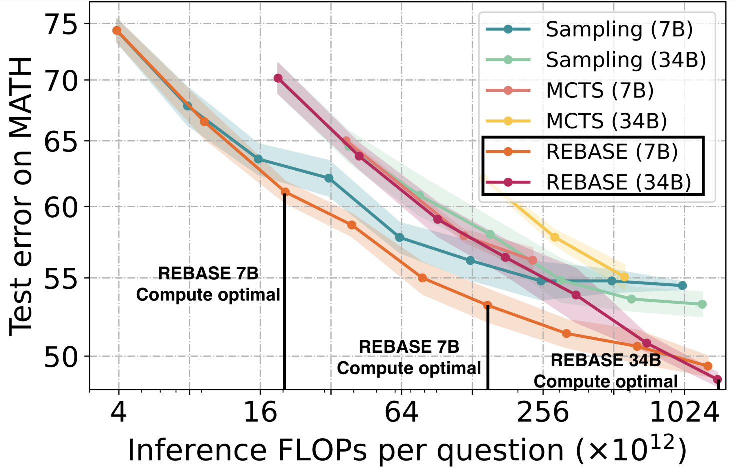
Smaller models can be compute optimal [Wu et al., 2024b].

Meta-generation | how do we choose a meta-generator?

Question 2: what is the compute-optimal meta-generation strategy?

Experiment: vary strategy (and model size and number of tokens)

Meta-generation | how do we choose a meta-generator?



Tree search (REBASE) can be compute-optimal [Wu et al., 2024b].

Meta-generation | recap

- Performance improves with increased compute...
  - ... but it varies by the choice of model size and meta-generator
- The optimal model size and strategy varies with the compute budget
  - Sometimes smaller models are better!
  - Goal: design strategies that are universally optimal

Meta-generation | recap

- Meta-generators: strategies for calling generators and incorporating external information
- Several patterns: chain, parallel, tree search, refinement
- They can be combined and mixed together
- Choose and design methods based on task performance *and* cost

Meta-generation | speeding up

Next: The preceding meta-generators

- Generate many tokens
- In diverse ways (e.g., tree search)

How do we do this quickly and efficiently?

Efficient meta-generation

Efficiency | goals

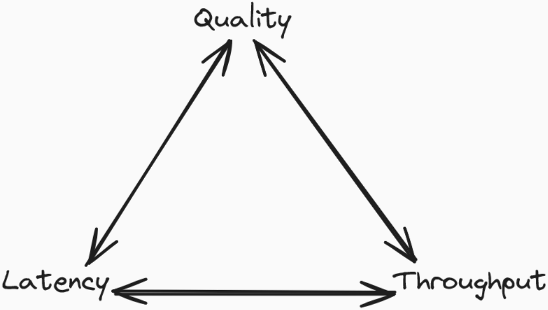
Scope:

- Basics of efficient generation
- How can we make meta-generation faster?
- Which specific meta-generators are most efficient?

Efficiency | basics

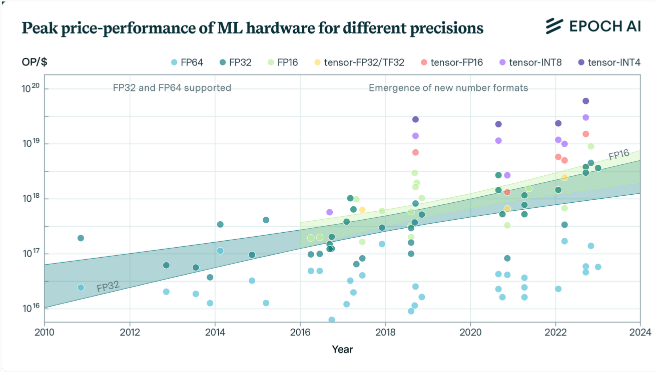
How do we measure “efficiency”?

- Latency
    - *How long does a user wait for a response?*
- Throughput
    - *How many requests are completed per second?*



Latency, **Throughput**, and **Quality** often trade off at a given budget.

Hardware improvements have driven model improvements [21]



The largest efficiency wins come from mapping operations onto hardware (more) effectively!

---

[21] Figure: [Hobbhahn et al., 2023]

How do ML accelerator designs impact generation efficiency?

- How much data can we keep on-device?
  - VRAM (GB)
- How many operations/second can the device perform?
  - FLOP/s
- How long does it take to send operands from GPU memory (HBM) to the processor?
  - Memory Bandwidth (GB/s)

Efficiency | bottlenecks

- Loading inputs (activations) from memory
    - Memory Bandwidth
- Loading *weights* from memory
    - Memory Bandwidth
- Performing computation
    - FLOP/s
- Communicating across devices
    - Communication Speeds (GB/s)
- ...

Time per operation can be modeled as[22]:

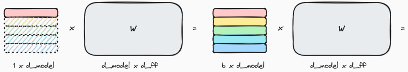
$$\text{Time} = \max\left(\frac{\text{Operation FLOP}}{\text{Device FLOP/s}}, \frac{\text{Data Transferred (GB)}}{\text{Memory Bandwidth (GB/s)}}\right)$$

Operations are either “compute-bound” or “memory-bound”[23]

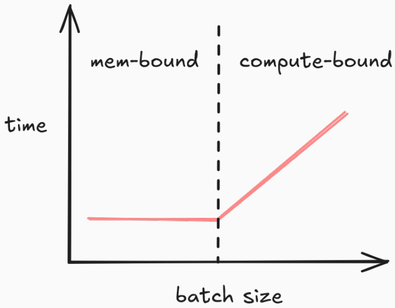
[22] [He, 2022]

[23] H100 SXM: BF16 dense tensor core max FLOP/s $\approx 1 \times 10^{15}$ FLOP/s, Memory bandwidth $\approx 3.35 \times 10^{12}$ byte/s. $\gg$ 100 FLOP/byte is “free”!

Efficiency | batching

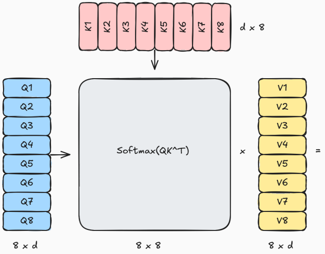


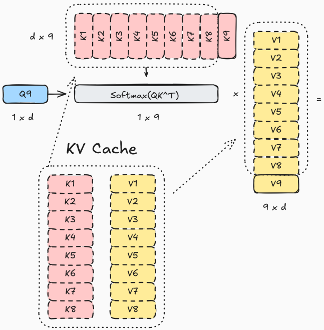
Inputs to a model can be “batched” together and computed simultaneously.



Batching can be cost-free for memory-bound operations![a]

[a] https://www.artfintel.com/p/how-does-batching-work-on-modern





Prefill Stage: process prompt all at once. Keys and values retained and initialize the "KV Cache".

Decode Stage: use cached KV values to compute attention for current timestep. Append new K, V to KV cache

$$\text{Size} = (\text{batch} \cdot \text{n\_ctx}) \cdot (2 \cdot \text{n\_layer} \cdot \text{n\_heads} \cdot \text{head\_dim}) \cdot (\text{n\_bytes})$$

Efficient meta-generation

How to speed up sampling a single token?

For a single decoding step, how do we work around hardware constraints?

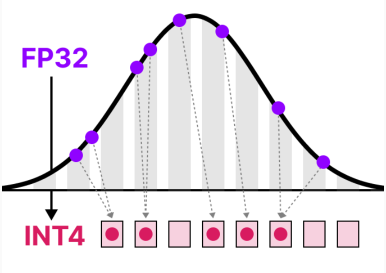
- Memory Bandwidth $\downarrow$
- FLOP/s $\uparrow$
- FLOP $\downarrow$

Efficiency | single-token

Memory Bandwidth $\downarrow$: reduce data transferred

- Quantize weights or activations[24]

$$\color{red}{(\text{bytes per parameter})} \cdot (\text{total parameters})$$



- Compress or distill model

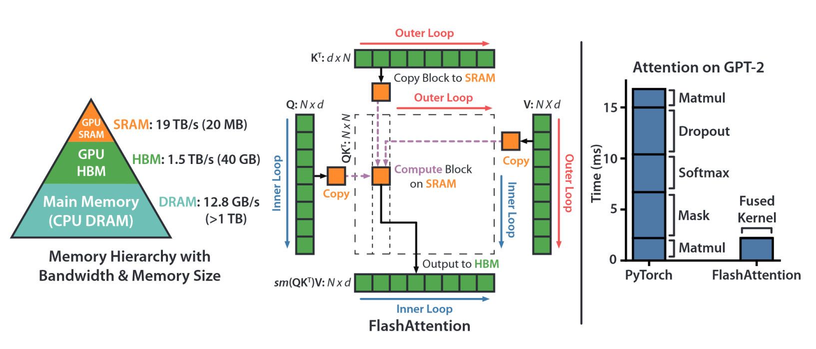
$$(\text{bytes per parameter}) \cdot \color{red}{(\text{total parameters})}$$

[24] Visual from https://newsletter.maartengrootendorst.com/p/a-visual-guide-to-quantization

Efficiency | single-token

FLOP/s $\uparrow$: improve hardware utilization

$$\color{red}{(\text{FLOP per second})} \cdot (\text{total operation FLOP})$$

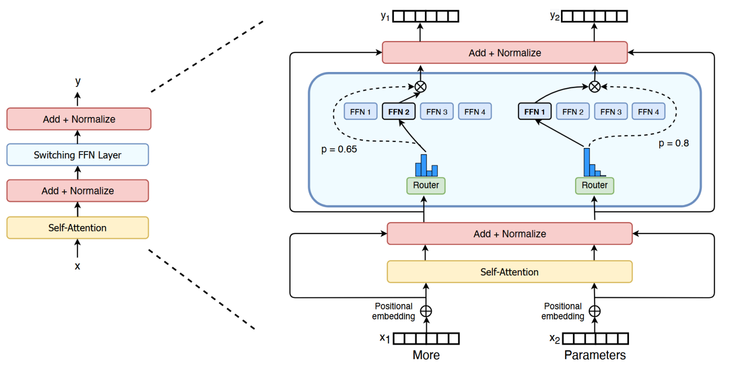


Flash Attention [Dao et al., 2022] performs the same operations, but optimizes the implementation to achieve far greater speed

Efficiency | single-token

FLOP ↓: reduce operations required

$$(\text{FLOP per second}) \cdot {\color{red}(\text{total operation FLOP})}$$



Mixture-of-Experts models use fewer FLOP per token than equi-parameter dense models [Fedus et al., 2022]

Efficient meta-generation

How to speed up a single generation?

Efficiency | single-generation

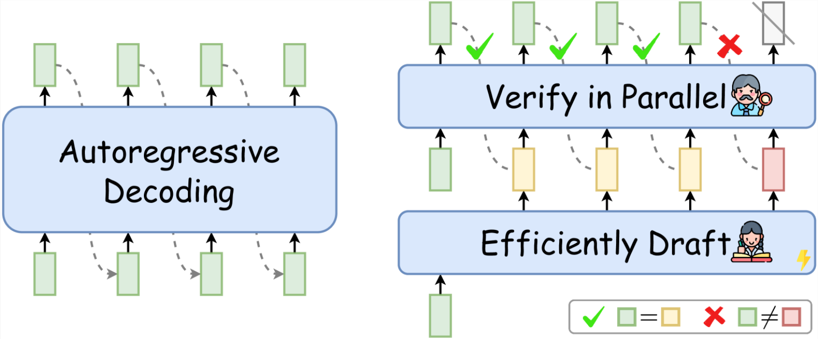
Generation of long outputs is bottlenecked by sequential next-token prediction. But not all tokens are created equal!

… The cow jumped over the moon . <EOS>

How can we spend less time on "easier" tokens?

Efficiency | single-generation

Decoding is typically **memory-bound**.

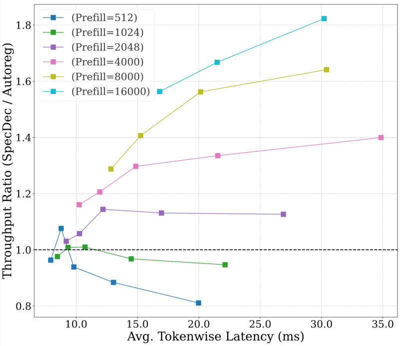


Speculative decoding uses a smaller **draft** model to produce “guesses” for the next N tokens cheaply, which are then “accepted” or “rejected” in parallel by the main model [Xia et al., 2024]

In **speculative decoding**:

- A lighter-weight *draft* model generates N “proposal” tokens
- These N “proposal” tokens can be passed **in parallel** into the main generator
- All tokens which match the main generator’s predictions are retained, and ones that do not are discarded

Efficiency | single-generation



Speculative decoding can harm throughput at low context but improves both throughput and latency at long context lengths [Chen et al., 2024a]

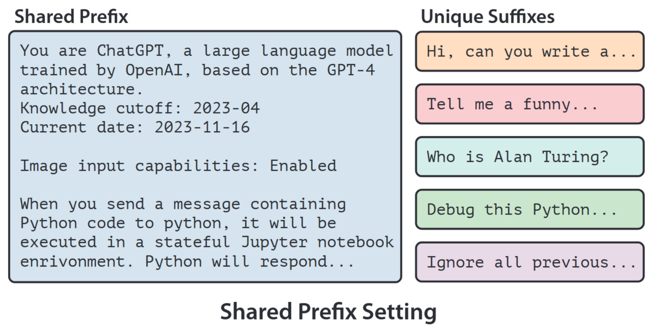
Efficient meta-generation

How to speed up meta-generation?

Efficiency | meta-generators

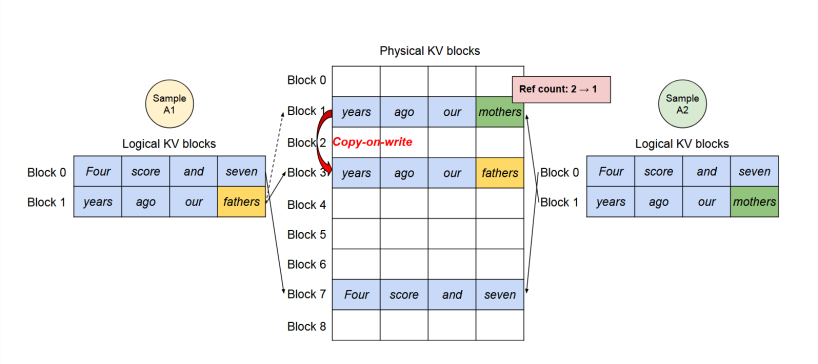
- How do meta-generators interact with **real-world efficiency** and **hardware utilization**?
- **Which** meta-generators are the fastest? Can we design more efficient meta-generators?

Efficiency | meta-generators | KV Cache reuse



Common deployment and parallel generation scenarios have redundant **shared prefix** content in prompts[25]

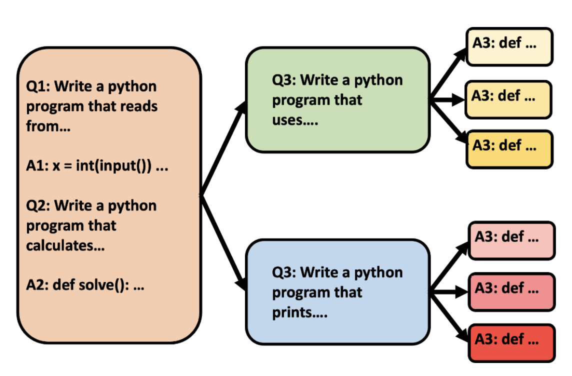
[25] Figure from [Juravsky et al., 2024]



PagedAttention [Kwon et al., 2023] prevents redundant storage costs by mapping KV cache blocks to physical “pages” of VRAM

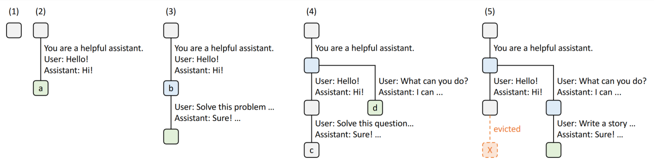
Efficiency | meta-generators | KV cache reuse

KV Cache reuse is not limited to single-level shared prefixes!



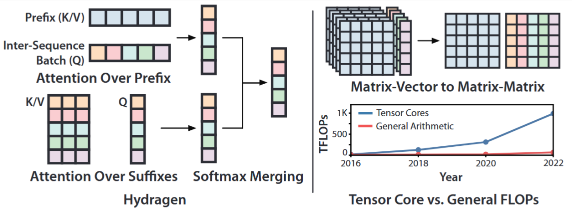
Multiple levels of prefix sharing can arise frequently: for example, combining a long few-shot prompt with Best-of-N generation[26]

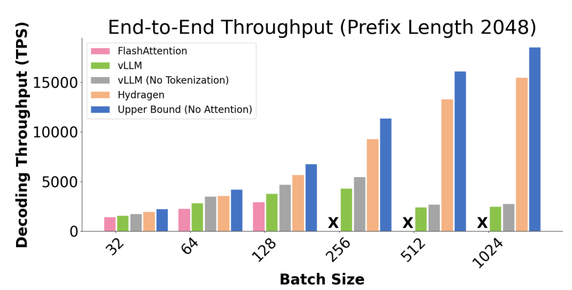
[26] Figure from [Juravsky et al., 2024]



RadixAttention enables complex prefix sharing patterns [Zheng et al., 2024], evicting least-recently-used KV cache blocks from memory when needed

Efficiency | meta-generators | KV Cache reuse





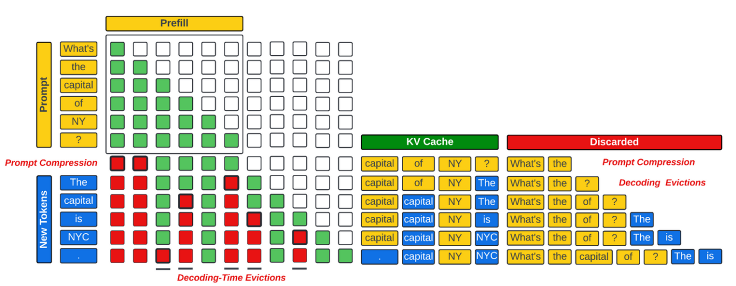
Hydragen [Juravsky et al., 2024] makes shared-prefix attention components faster via leveraging Tensor Cores

KV Cache size is a key bottleneck to larger batches and to longer context inference

- Token Dropping: Selectively remove tokens from the KV Cache
- Quantization: Modify KV Cache datatype
- Architectural Modification: Reduce inherent size of a prospective model's KV Cache

Token Dropping:

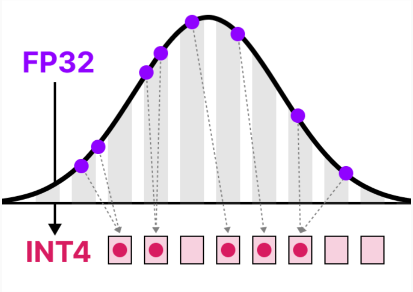
$$(\text{batch} \cdot \color{red}{\text{n\_ctx}}\color{black}) \cdot (2 \cdot \text{n\_layer} \cdot \text{n\_heads} \cdot \text{head\_dim}) \cdot (\text{n\_bytes})$$



An overview of approaches to control KV Cache size via *token dropping* [Adams et al., 2024]

Quantization:

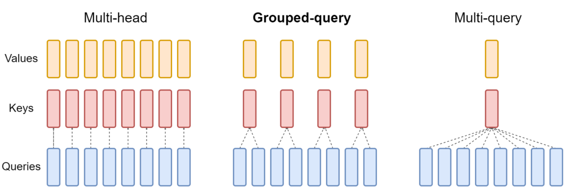
$$(\text{batch} \cdot \text{n\_ctx}) \cdot (2 \cdot \text{n\_layer} \cdot \text{n\_heads} \cdot \text{head\_dim}) \cdot \color{red}{(\text{n\_bytes})}$$



As with model weights, elements of the KV cache can be *quantized* to reduce memory overheads

Architectural Modification:

$$(\text{batch} \cdot \text{n\_ctx}) \cdot (2 \cdot \text{n\_layer} \cdot \textcolor{red}{\text{n\_heads}} \cdot \text{head\_dim}) \cdot (\text{n\_bytes})$$



Architectural tweaks such as Multi-Query Attention [Shazeer, 2019] or Grouped-Query Attention [Ainslie et al., 2023] reduce the number of Key + Value attention heads to shrink the required KV Cache size

Efficiency | meta-generators | recap

Which meta-generators are most efficient?

- **Parallelizable**: trajectories can be run in parallel; not sequentially bottlenecked
- **Prefix-shareable**: long inputs are presented as identical shared prefix content, whose KV Caches can be reused across many model calls

Token budget is not the only indicator of meta-generator efficiency!

Recap and takeaways

This tutorial

Beyond Decoding: Meta-Generation Algorithms for LLMs

- **Primitive generators:** Generating one token at a time
- **Meta-generators:** High-level strategies for calling generators
- **Efficient meta-generation:** Generating quickly and efficiently

Takeaways | Meta-generators

Meta-generation: strategies for calling generators

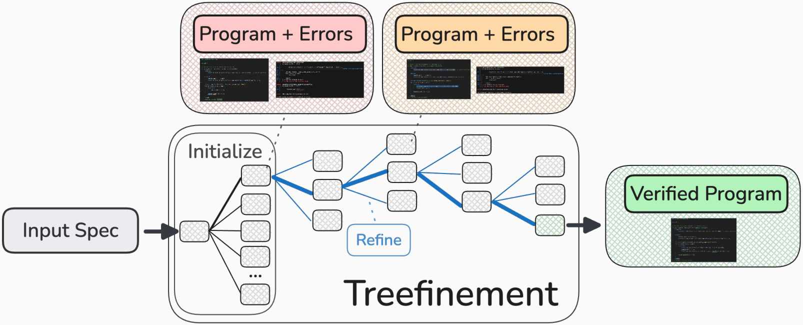
- Various strategies: chained, parallel, tree search, refinement
- Spend test-time compute to improve performance
- Use cost-performance tradeoffs to choose/design

Takeaways | Efficient meta-generation

- **Parallelizability** decreases latency and boosts throughput of meta-generation
- Long inputs can be amortized via **Prefix Sharing** of KV Cache
- Prompt design and meta-generator structure can change real-world efficiency significantly. **Token budget** can be an oversimplification!

Looking ahead

- Hybrid meta-generators



[Aggarwal et al., 2024], *AlphaVerus*. P. Aggarwal, B. Parno, S. Welleck.

Looking ahead

- Hybrid meta-generators
- Learning to search (e.g., explore, backtrack, self-correct)
- Agent environments
- How should we allocate compute?

Looking ahead

- Hybrid meta-generators
- Learning to search (e.g., explore, backtrack, self-correct)
- Agent environments
- How should we allocate compute?

Science: many conclusions are based on a few tasks!

Looking ahead

Survey Paper (TMLR 2024):

From Decoding to Meta-Generation:
Inference-time Algorithms for Large Language Models.
Sean Welleck, Amanda Bertsch*, Matt Finlayson*, Hailey Schoelkopf*, Alex Xie, Graham Neubig, Ilia Kulikov, Zaid Harchaoui. TMLR 2024.
https://arxiv.org/abs/2406.16838

Thank you!

https://cmu-l3.github.io/neurips2024-inference-tutorial

Panel

Beidi Chen
CMU
`@BeidiChen`

Nouha Dziri
AI2
`@nouhadziri`

Rishabh Agarwal
DeepMind/McGill
`@agarwl_`

Jakob Foerster
Oxford/Meta AI
`@j_foerst`

Noam Brown
OpenAI
`@polynoamial`

Ilia Kulikov (Moderator)
Meta AI
`@uralik1`

https://cmu-l3.github.io/neurips2024-inference-tutorial

Appendix

Meta-generators | parallel | pairwise

Pairwise: *Minimum Bayes Risk*

$$\mathrm{MBR}(g, v, N) = \underset{y \in \{y^{(1)}, \ldots, y^{(N)}\}}{\arg\max} \underbrace{\frac{1}{N} \sum_{i=1}^{N} v(y, y^{(i)})}_{\approx \mathbb{E}_{y' \sim p}[v(y, y')]},$$

where $\{y^{(1)}, \ldots, y^{(N)}\} \sim g$ and $v(y, y')$ is a “utility” function.

Pairwise: *Minimum Bayes Risk*

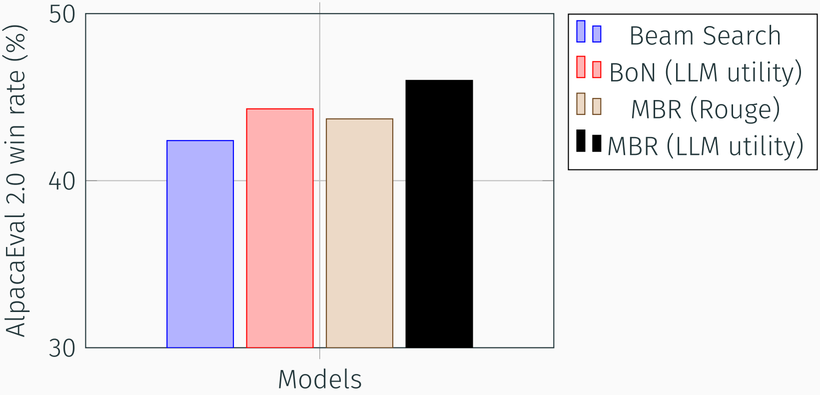
$$\mathrm{MBR}(g, v, N) = \underset{y \in \{y^{(1)}, \ldots, y^{(N)}\}}{\arg\max} \underbrace{\frac{1}{N} \sum_{i=1}^{N} v(y, y^{(i)})}_{\approx \mathbb{E}_{y' \sim p}[v(y, y')]},$$

where $\{y^{(1)}, \ldots, y^{(N)}\} \sim g$ and $v(y, y')$ is a “utility” function.

Intuitively, selects the candidate with the highest “consensus” utility.

Meta-generators | parallel | pairwise[27]

Utility: $LLM(y, y^{(i)}) \rightarrow \{1, 2, 3, 4, 5\}$:



[27] Example from [Wu et al., 2024a] (Llama 3 70B). Utility: Prometheus 2 [Kim et al., 2024].

Meta-generators | parallel | connecting MBR and voting

Weighted voting is an instance of Minimum Bayes Risk:[28]

$$\underbrace{v(y, y^{(i)})}_{\text{utility}} = \underbrace{1\left[a = a^{(i)}\right]}_{\text{same answer}} \cdot \underbrace{v(y^{(i)})}_{\text{sequence score}}, \tag{5}$$

where $y = (z, a)$, $y^{(i)} = (z^{(i)}, a^{(i)})$.

[28][Bertsch et al., 2023] *It's MBR All the Way Down: Modern Generation Techniques Through the Lens of Minimum Bayes Risk. A. Bertsch, A. Xie, G. Neubig, M. Gormley.*

Code examples

speculative decoding

```python
def speculative_decode(tgt_m, drf_m, tok, inp: torch.Tensor, max_tok:
     int, n_spec: int = 5, t: float = 1.0):
    gen = inp; max_len = inp.shape[1] + max_tok
    while gen.shape[1] < max_len:
        tok_left = max_len - gen.shape[1]
        spec_size = min(n_spec, tok_left - 1)
        if spec_size > 0:
            spec_id, spec_lprob = generate(drf_m, tok, gen, spec_size, t)
            tgt_lprob = tgt_m(spec_id)  # forwarding tgt model
            rejs = compute_ll_rejs(tgt_lprob, spec_lprob)
            if len(rejs) > 0:
                accepted = spec_id[:, :rejs[0]]
                adj_probs = compute_adjusted_dist(tgt_lprob, spec_lprob)
                next_tok = Categorical(adj_probs)
            else:
                accepted = spec_id
                next_tok = Categorical(tgt_lprob.exp())
        gen = torch.cat([gen, accepted, next_tok])
```

speculative decoding

```
def compute_ll_rejs(tgt_lprob: torch.Tensor, spec_lprob: torch.Tensor,
     spec_tok_id: torch.Tensor) -> torch.Tensor:
    llrs = tgt_lprob[spec_tok_id] - spec_lprob[spec_tok_id]
    uniform_lprobs = torch.log(torch.rand_like(llrs))
    rej_idx = torch.nonzero((llrs <= uniform_lprobs))
    return rej_idx

def compute_adjusted_dist(tgt_lprob: torch.Tensor, spec_lprob:
     torch.Tensor, rej_idx: torch.Tensor) -> torch.Tensor:
    adj_dist = torch.clamp(
        torch.exp(tgt_lprob[rej_idx]) - torch.exp(spec_lprob[rej_idx]),
        min=0
    )
    adj_dist = torch.div(adj_dist, adj_dist.sum())
    return adj_dist
```

Ackley, D. H., Hinton, G. E., and Sejnowski, T. J. (1985).
A learning algorithm for boltzmann machines.
Cognitive Science, 9(1):147–169.

Adams, G., Ladhak, F., Schoelkopf, H., and Biswas, R. (2024).
Cold compress: A toolkit for benchmarking kv cache compression approaches.

Aggarwal, P., Parno, B., and Welleck, S. (2024).
Alphaverus: Bootstrapping formally verified code generation through self-improving translation and treefinement.
`https://arxiv.org/abs/2412.06176`.

Ainslie, J., Lee-Thorp, J., de Jong, M., Zemlyanskiy, Y., Lebrón, F., and Sanghai, S. (2023).
Gqa: Training generalized multi-query transformer models from multi-head checkpoints.

Ankner, Z., Paul, M., Cui, B., Chang, J. D., and Ammanabrolu, P. (2024).
Critique-out-loud reward models.

Asai, A., He⋆, J., Shao⋆, R., Shi, W., Singh, A., Chang, J. C., Lo, K., Soldaini, L., Feldman, Tian, S., Mike, D., Wadden, D., Latzke, M., Minyang, Ji, P., Liu, S., Tong, H., Wu, B., Xiong, Y., Zettlemoyer, L., Weld, D., Neubig, G., Downey, D., Yih, W.-t., Koh, P. W., and Hajishirzi, H. (2024).
OpenScholar: Synthesizing scientific literature with retrieval-augmented language models.

Arxiv.

Basu, S., Ramachandran, G. S., Keskar, N. S., and Varshney, L. R. (2021).
Mirostat: a neural text decoding algorithm that directly controls perplexity.
In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net.

Bertsch, A., Xie, A., Neubig, G., and Gormley, M. (2023).
It's MBR all the way down: Modern generation techniques through the lens of minimum Bayes risk.
In Elazar, Y., Ettinger, A., Kassner, N., Ruder, S., and A. Smith, N., editors, *Proceedings of the Big Picture Workshop*, pages 108–122, Singapore. Association for Computational Linguistics.

Brown, B., Juravsky, J., Ehrlich, R., Clark, R., Le, Q. V., Ré, C., and Mirhoseini, A. (2024).
Large language monkeys: Scaling inference compute with repeated sampling.
https://arxiv.org/abs/2407.21787.

Chen, J., Tiwari, V., Sadhukhan, R., Chen, Z., Shi, J., Yen, I. E.-H., and Chen, B. (2024a).
Magicdec: Breaking the latency-throughput tradeoff for long context generation with speculative decoding.

Chen, X., Lin, M., Schärli, N., and Zhou, D. (2024b).
Teaching large language models to self-debug.
In *The Twelfth International Conference on Learning Representations*.

- Chung, H. W., Hou, L., Longpre, S., Zoph, B., Tay, Y., Fedus, W., Li, Y., Wang, X., Dehghani, M., Brahma, S., Webson, A., Gu, S. S., Dai, Z., Suzgun, M., Chen, X., Chowdhery, A., Castro-Ros, A., Pellat, M., Robinson, K., Valter, D., Narang, S., Mishra, G., Yu, A., Zhao, V., Huang, Y., Dai, A., Yu, H., Petrov, S., Chi, E. H., Dean, J., Devlin, J., Roberts, A., Zhou, D., Le, Q. V., and Wei, J. (2022).
  Scaling instruction-finetuned language models.
  https://arxiv.org/abs/2210.11416.
- Cobbe, K., Kosaraju, V., Bavarian, M., Chen, M., Jun, H., Kaiser, L., Plappert, M., Tworek, J., Hilton, J., Nakano, R., Hesse, C., and Schulman, J. (2021).
  Training verifiers to solve math word problems.
  https://arxiv.org/abs/2110.14168.

Dao, T., Fu, D. Y., Ermon, S., Rudra, A., and R'e, C. (2022).
Flashattention: Fast and memory-efficient exact attention with io-awareness.
ArXiv preprint, abs/2205.14135.

Dohan, D., Xu, W., Lewkowycz, A., Austin, J., Bieber, D., Lopes, R. G., Wu, Y., Michalewski, H., Saurous, R. A., Sohl-dickstein, J., Murphy, K., and Sutton, C. (2022).
Language model cascades.
`https://arxiv.org/abs/2207.10342`.

Fan, A., Lewis, M., and Dauphin, Y. (2018).
Hierarchical neural story generation.
In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 889–898. Association for Computational Linguistics.

Fedus, W., Zoph, B., and Shazeer, N. (2022).
Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity.

Feng, G., Zhang, B., Gu, Y., Ye, H., He, D., and Wang, L. (2023).
Towards revealing the mystery behind chain of thought: A theoretical perspective.
In *Thirty-seventh Conference on Neural Information Processing Systems.*

Finlayson, M., Hewitt, J., Koller, A., Swayamdipta, S., and Sabharwal, A. (2024).
Closing the curious case of neural text degeneration.
In *The Twelfth International Conference on Learning Representations.*

Freitag, M. and Al-Onaizan, Y. (2017).
Beam search strategies for neural machine translation.
In *Proceedings of the First Workshop on Neural Machine Translation*, pages 56–60. Association for Computational Linguistics.

He, H. (2022).
Making deep learning go brrrr from first principles.

Hewitt, J., Manning, C., and Liang, P. (2022).
Truncation sampling as language model desmoothing.
In *Findings of the Association for Computational Linguistics: EMNLP 2022*, pages 3414–3427. Association for Computational Linguistics.

Hobbhahn, M., Heim, L., and Aydos, G. (2023).
Trends in machine learning hardware.
Accessed: 2024-11-26.

Holtzman, A., Buys, J., Du, L., Forbes, M., and Choi, Y. (2020).
The curious case of neural text degeneration.
In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net.

Huang, J., Chen, X., Mishra, S., Zheng, H. S., Yu, A. W., Song, X., and Zhou, D. (2024).
Large language models cannot self-correct reasoning yet.
In *The Twelfth International Conference on Learning Representations*.

Jiang, A. Q., Welleck, S., Zhou, J. P., Lacroix, T., Liu, J., Li, W., Jamnik, M., Lample, G., and Wu, Y. (2023).
Draft, sketch, and prove: Guiding formal theorem provers with informal proofs.
In *The Eleventh International Conference on Learning Representations*.

Juravsky, J., Brown, B., Ehrlich, R., Fu, D. Y., Ré, C., and Mirhoseini, A. (2024).
Hydragen: High-throughput llm inference with shared prefixes.

Kaplan, J., McCandlish, S., Henighan, T., Brown, T. B., Chess, B., Child, R., Gray, S., Radford, A., Wu, J., and Amodei, D. (2020).
Scaling laws for neural language models.
`https://arxiv.org/abs/2001.08361`.

Khattab, O., Santhanam, K., Li, X. L., Hall, D. L. W., Liang, P., Potts, C., and Zaharia, M. A. (2022).
Demonstrate-search-predict: Composing retrieval and language models for knowledge-intensive nlp.
ArXiv, abs/2212.14024.

Kim, S., Suk, J., Longpre, S., Lin, B. Y., Shin, J., Welleck, S., Neubig, G., Lee, M., Lee, K., and Seo, M. (2024).
Prometheus 2: An open source language model specialized in evaluating other language models.
https://arxiv.org/abs/2405.01535.

Koh, J. Y., McAleer, S., Fried, D., and Salakhutdinov, R. (2024).
Tree search for language model agents.
arXiv preprint arXiv:2407.01476.

Kudo, T. (2018).
Subword regularization: Improving neural network translation models with multiple subword candidates.
In Gurevych, I. and Miyao, Y., editors, *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 66–75, Melbourne, Australia. Association for Computational Linguistics.

Kumar, A., Zhuang, V., Agarwal, R., Su, Y., Co-Reyes, J. D., Singh, A., Baumli, K., Iqbal, S., Bishop, C., Roelofs, R., Zhang, L. M., McKinney, K., Shrivastava, D., Paduraru, C., Tucker, G., Precup, D., Behbahani, F., and Faust, A. (2024).
Training language models to self-correct via reinforcement learning.

Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J. E., Zhang, H., and Stoica, I. (2023).
Efficient memory management for large language model serving with pagedattention.

Li, X. L., Holtzman, A., Fried, D., Liang, P., Eisner, J., Hashimoto, T., Zettlemoyer, L., and Lewis, M. (2023a).
Contrastive decoding: Open-ended text generation as optimization.
In Rogers, A., Boyd-Graber, J., and Okazaki, N., editors, *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 12286–12312. Association for Computational Linguistics.

Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., Eccles, T., Keeling, J., Gimeno, F., Lago, A. D., Hubert, T., Choy, P., de Masson d'Autume, C., Babuschkin, I., Chen, X., Huang, P.-S., Welbl, J., Gowal, S., Cherepanov, A., Molloy, J., Mankowitz, D. J., Robson, E. S., Kohli, P., de Freitas, N., Kavukcuoglu, K., and Vinyals, O. (2022).
Competition-level code generation with alphacode.
Science, 378(6624):1092–1097.

Li, Y., Lin, Z., Zhang, S., Fu, Q., Chen, B., Lou, J.-G., and Chen, W. (2023b).
Making language models better reasoners with step-aware verifier.
In Rogers, A., Boyd-Graber, J., and Okazaki, N., editors, *Proceedings of the 61st Annual Meeting of the Association for*

Computational Linguistics (Volume 1: Long Papers), pages 5315–5333, Toronto, Canada. Association for Computational Linguistics.

Lightman, H., Kosaraju, V., Burda, Y., Edwards, H., Baker, B., Lee, T., Leike, J., Schulman, J., Sutskever, I., and Cobbe, K. (2024).
Let's verify step by step.
In *The Twelfth International Conference on Learning Representations*.

Liu, A., Han, X., Wang, Y., Tsvetkov, Y., Choi, Y., and Smith, N. A. (2024).
Tuning language models by proxy.

Liu, A., Sap, M., Lu, X., Swayamdipta, S., Bhagavatula, C., Smith, N. A., and Choi, Y. (2021).
DExperts: Decoding-time controlled text generation with experts and anti-experts.
In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 6691–6706. Association for Computational Linguistics.

Lu, X., Brahman, F., West, P., Jung, J., Chandu, K., Ravichander, A., Ammanabrolu, P., Jiang, L., Ramnath, S., Dziri, N., Fisher, J., Lin, B., Hallinan, S., Qin, L., Ren, X., Welleck, S., and Choi, Y. (2023). **Inference-time policy adapters (IPA): Tailoring extreme-scale LMs without fine-tuning.** In Bouamor, H., Pino, J., and Bali, K., editors, *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 6863–6883. Association for Computational Linguistics.

Madaan, A., Tandon, N., Gupta, P., Hallinan, S., Gao, L., Wiegreffe, S., Alon, U., Dziri, N., Prabhumoye, S., Yang, Y., Gupta, S., Majumder, B. P., Hermann, K., Welleck, S., Yazdanbakhsh, A., and Clark, P. (2023).
Self-refine: Iterative refinement with self-feedback.
In *Thirty-seventh Conference on Neural Information Processing Systems*.

Meister, C., Cotterell, R., and Vieira, T. (2020).
If beam search is the answer, what was the question?
In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 2173–2185. Association for Computational Linguistics.

Meister, C., Pimentel, T., Wiher, G., and Cotterell, R. (2022).
Locally typical sampling.
Transactions of the Association for Computational Linguistics, 11:102–121.

Meister, C., Pimentel, T., Wiher, G., and Cotterell, R. (2023).
Locally typical sampling.
Transactions of the Association for Computational Linguistics, 11:102–121.

Merrill, W. and Sabharwal, A. (2024).
The expressive power of transformers with chain of thought.
In *The Twelfth International Conference on Learning Representations*.

Nakano, R., Hilton, J., Balaji, S., Wu, J., Ouyang, L., Kim, C., Hesse, C., Jain, S., Kosaraju, V., Saunders, W., Jiang, X., Cobbe, K., Eloundou, T., Krueger, G., Button, K., Knight, M., Chess, B., and Schulman, J. (2022).
Webgpt: Browser-assisted question-answering with human feedback.
https://arxiv.org/abs/2112.09332.

Nebius (2024).
Leveraging training and search for better software engineering agents.
https://nebius.com/blog/posts/training-and-search-for-software-engineering-agents.

- Nowak, F., Svete, A., Butoi, A., and Cotterell, R. (2024).
  On the representational capacity of neural language models with chain-of-thought reasoning.
  In Ku, L.-W., Martins, A., and Srikumar, V., editors, *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 12510–12548, Bangkok, Thailand. Association for Computational Linguistics.
- OpenAI (2024).
  Learning to reason with llms.
  `https://openai.com/index/learning-to-reason-with-llms/`.
- Polu, S. and Sutskever, I. (2020).
  Generative language modeling for automated theorem proving.

Press, O., Zhang, M., Min, S., Schmidt, L., Smith, N., and Lewis, M. (2023).
Measuring and narrowing the compositionality gap in language models.
In Bouamor, H., Pino, J., and Bali, K., editors, *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 5687–5711. Association for Computational Linguistics.

Schlag, I., Sukhbaatar, S., Celikyilmaz, A., tau Yih, W., Weston, J., Schmidhuber, J., and Li, X. (2023).
Large language model programs.
`https://arxiv.org/abs/2305.05364`.

Shazeer, N. (2019).
Fast transformer decoding: One write-head is all you need.

Shi, C., Yang, H., Cai, D., Zhang, Z., Wang, Y., Yang, Y., and Lam, W. (2024).
A thorough examination of decoding methods in the era of LLMs.
In Al-Onaizan, Y., Bansal, M., and Chen, Y.-N., editors, *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 8601–8629, Miami, Florida, USA. Association for Computational Linguistics.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., and Hassabis, D. (2016).

Mastering the game of go with deep neural networks and tree search.
Nature, 529:484–503.

Stahlberg, F. and Byrne, B. (2019).
On nmt search errors and model errors: Cat got your tongue?
ArXiv, abs/1908.10090.

Stiennon, N., Ouyang, L., Wu, J., Ziegler, D., Lowe, R., Voss, C., Radford, A., Amodei, D., and Christiano, P. F. (2020).
Learning to summarize with human feedback.
In Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M., and Lin, H., editors, *Advances in Neural Information Processing Systems*, volume 33, pages 3008–3021. Curran Associates, Inc.

Sun, Z., Yu, L., Shen, Y., Liu, W., Yang, Y., Welleck, S., and Gan, C. (2024).
Easy-to-hard generalization: Scalable alignment beyond human supervision.
In *The Thirty-eighth Annual Conference on Neural Information Processing Systems.*

Uesato, J., Kushman, N., Kumar, R., Song, F., Siegel, N., Wang, L., Creswell, A., Irving, G., and Higgins, I. (2022).
Solving math word problems with process- and outcome-based feedback.

Wang, P., Li, L., Shao, Z., Xu, R., Dai, D., Li, Y., Chen, D., Wu, Y., and Sui, Z. (2024a).
Math-shepherd: Verify and reinforce LLMs step-by-step without human annotations.
In Ku, L.-W., Martins, A., and Srikumar, V., editors, *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 9426–9439, Bangkok, Thailand. Association for Computational Linguistics.

Wang, X., Wei, J., Schuurmans, D., Le, Q. V., Chi, E. H., Narang, S., Chowdhery, A., and Zhou, D. (2023).
Self-consistency improves chain of thought reasoning in language models.
In *The Eleventh International Conference on Learning Representations*.

Wang, Y., Wu, Y., Wei, Z., Jegelka, S., and Wang, Y. (2024b).
A theoretical understanding of self-correction through in-context alignment.
https://arxiv.org/abs/2405.18634.

Wei, J., Wang, X., Schuurmans, D., Bosma, M., brian ichter, Xia, F., Chi, E. H., Le, Q. V., and Zhou, D. (2022).
Chain of thought prompting elicits reasoning in large language models.
In Oh, A. H., Agarwal, A., Belgrave, D., and Cho, K., editors, *Advances in Neural Information Processing Systems*.

Welleck, S., Bertsch, A., Finlayson, M., Schoelkopf, H., Xie, A., Neubig, G., Kulikov, I., and Harchaoui, Z. (2024).
From decoding to meta-generation: Inference-time algorithms for large language models.
https://arxiv.org/abs/2406.16838.

Welleck, S., Kulikov, I., Roller, S., Dinan, E., Cho, K., and Weston, J. (2020).
Neural text generation with unlikelihood training.
In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020.* OpenReview.net.

Welleck, S., Lu, X., West, P., Brahman, F., Shen, T., Khashabi, D., and Choi, Y. (2023).
Generating sequences by learning to self-correct.
In *The Eleventh International Conference on Learning Representations*.

Weston, J. and Sukhbaatar, S. (2023).
System 2 attention (is something you might need too).

Wu, I., Fernandes, P., Bertsch, A., Kim, S., Pakazad, S., and Neubig, G. (2024a).
Better instruction-following through minimum bayes risk.
https://arxiv.org/abs/2410.02902.

Wu, Y., Sun, Z., Li, S., Welleck, S., and Yang, Y. (2024b).
Inference scaling laws: An empirical analysis of compute-optimal inference for problem-solving with language models.
https://arxiv.org/abs/2408.00724.

Xia, H., Yang, Z., Dong, Q., Wang, P., Li, Y., Ge, T., Liu, T., Li, W., and Sui, Z. (2024).
Unlocking efficiency in large language model inference: A comprehensive survey of speculative decoding.

Zaharia, M., Khattab, O., Chen, L., Davis, J. Q., Miller, H., Potts, C., Zou, J., Carbin, M., Frankle, J., Rao, N., and Ghodsi, A. (2024).
The shift from models to compound ai systems.
https://bair.berkeley.edu/blog/2024/02/18/compound-ai-systems/.

Zhang, L., Hosseini, A., Bansal, H., Kazemi, M., Kumar, A., and Agarwal, R. (2024).
Generative verifiers: Reward modeling as next-token prediction.

Zheng, L., Yin, L., Xie, Z., Sun, C., Huang, J., Yu, C. H., Cao, S., Kozyrakis, C., Stoica, I., Gonzalez, J. E., Barrett, C., and Sheng, Y. (2024).
Sglang: Efficient execution of structured language model programs.