From Decoding to Meta-Generation: Inference-time Algorithms for Large Language Models

Anonymous authors
Paper under double-blind review

Abstract

One of the most striking findings in modern research on large language models (LLMs) is that scaling up compute during training leads to better results. However, less attention has been given to the benefits of scaling compute during inference. This survey focuses on these inference-time approaches. We explore three areas under a unified mathematical formalism: token-level generation algorithms, meta-generation algorithms, and efficient generation. Token-level generation algorithms, often called decoding algorithms, operate by sampling a single token at a time or constructing a token-level search space and then selecting an output. These methods typically assume access to a language model's logits, next-token distributions, or probability scores. Meta-generation algorithms work on partial or full sequences, incorporating domain knowledge, enabling backtracking, and integrating external information. Efficient generation methods aim to reduce token costs and improve the speed of generation. Our survey unifies perspectives from three research communities: traditional natural language processing, modern LLMs, and machine learning systems.

1 Introduction

One of the most striking findings in modern research on large language models (LLMs) is that, given a model and dataset of sufficient scale, scaling up compute at training time leads to better final results (Kaplan et al., 2020; Hoffmann et al., 2022). However, there is also another lesser-mentioned scaling phenomenon, where adopting more sophisticated methods or scaling compute at *inference time* (Jones, 2021) can result in substantially better outputs from LLMs. This survey focuses on these approaches by exploring three connected themes: token-level generation algorithms, meta-generation algorithms, and efficient generation.

Token-level generation algorithms, often called decoding algorithms, have a rich history in natural language processing, ranging from classical greedy decoding and beam search to modern sampling algorithms such as nucleus (Holtzman et al., 2020) and η-sampling (Hewitt et al., 2022). These methods operate by sampling one token at a time or constructing a token-level search space. They assume varying levels of access to a language model's internals, such as logits, next-token distributions, or probability scores.

Recently there has been growing interest in *meta-generation algorithms*—algorithms that operate on partial or full sequences, and treat the LLM as a black box that is called as part of a larger generation program (Figure 1; Dohan et al. (2022); Schlag et al. (2023)). For example, a meta-generation algorithm for solving a math problem might generate multiple solution paths, evaluate the solutions with a calculator, then select the most common answer. Meta-generators can increase the compute resources devoted to generation by making multiple model calls, augmenting the model with search algorithms (Yao et al., 2023; Madaan et al., 2023), or incorporating external data sources. Doing so has seen success in improving task performance (e.g., problem solving (Lewkowycz et al., 2022)) and steering the output distribution (e.g., with human preferences (Stiennon et al., 2020)), and potentially offers a way to overcome limitations of standard LLMs such as error accumulation (Dziri et al., 2023) and computational capacity (Merrill & Sabharwal, 2024). Moreover, meta-generation research is widely accessible, as it often only requires black-box LLM access.

Finally, generation needs to be fast and cost-effective. Fast generation becomes increasingly challenging as models grow in size, while cost becomes critical to consider as LLMs are integrated into algorithms that call

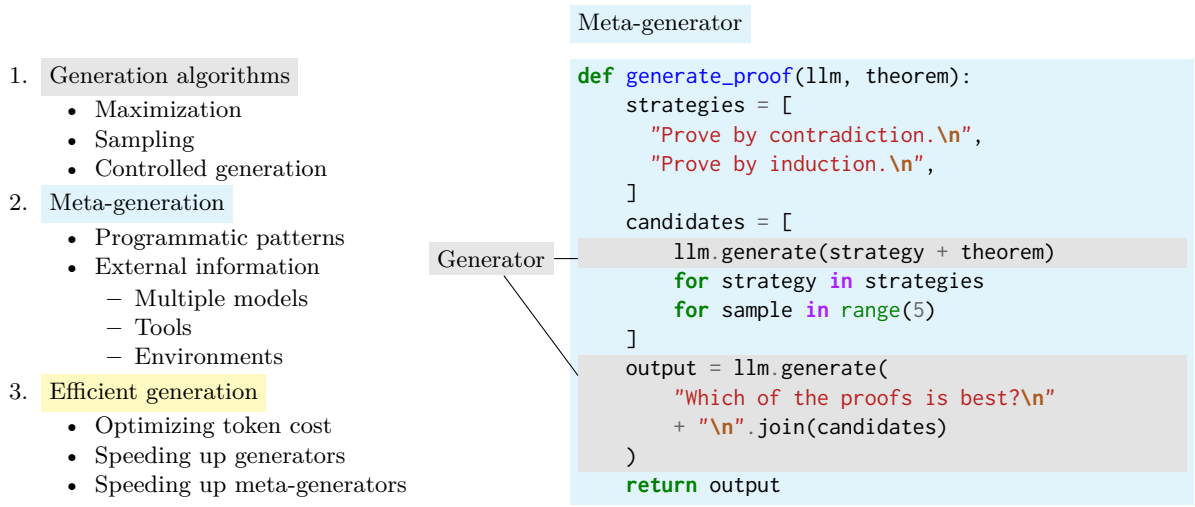


Figure 1: Generation algorithms produce output text using a language model. Meta-generation algorithms are programs that interleave calls to generation algorithms with control flow and external information, yielding text. Our survey covers generation algorithms and their goals (§3), meta-generation patterns (§4) and sources of external information (§5), and efficiency in terms of token cost (§6) and speed (§7).

models many times. As a result, there is growing interest in *efficient generation algorithms* that speed up generation and reduce token costs by drawing on ideas from machine learning systems and related areas. Efficient generation in turn expands the frontier of algorithms that are feasible to experiment with and develop, leading to a virtuous cycle of algorithmic development.

Our survey provides a unified treatment of these three themes: token-level generation algorithms, meta-generation algorithms, and techniques for making generation fast and cost-effective. We integrate ideas from traditional natural language processing, modern LLMs, and machine learning systems, and present a mathematical formalism that includes both classical generation algorithms and modern meta-generators. This unified view is particularly important as the field expands. For example, practitioners working on novel meta-generation algorithms may benefit from learning about the historical context of generation algorithms or practical efficiency constraints, while researchers interested in efficiency may benefit from learning about major algorithmic patterns. More broadly, we aim to promote further research on inference-time approaches.

Comparison to existing surveys. Several prior surveys have focused on training-time methods for better text generation (Li et al., 2021; Lu et al., 2018). Wiher et al. (2022) presents a detailed analysis of a smaller set of decoding strategies, while Zarrieß et al. (2021) spotlight token-level methods, with a particular focus on considerations for encoder-decoder models. In parallel, several surveys have addressed prompting and related methods (Liu et al., 2021b; Sahoo et al., 2024), though these works do not address token-level methods. Recent surveys have also considered strategies for speeding up inference (Chitty-Venkata et al., 2023; Miao et al., 2023a; Khoshnoodi et al., 2024; Wang et al., 2024a). However, these works focus primarily on token-level generation, not meta-generation; as a result, the discussion of inference-time compute-performance tradeoffs is limited. Our survey unifies and draws connections across these three areas. Finally, Xiao et al. (2023) focus on non-autoregressive generation, while our survey focuses on autoregressive generation.

Roadmap. This paper provides a survey of algorithms for token-level generation, meta-generation, and efficient generation, summarized in Figure 1. First, we consider why we use generation algorithms at all. Generally, a user's intent is to surface a high-quality output from the model, which we formalize and discuss in §2. Readers who would like to review terminology or follow the mathematical formulation of the survey in depth should start in this section. Next, we discuss token-level generation algorithms in detail in §3. Most algorithms referred to as "decoding algorithms" in the literature are covered in this section. We discuss these

Symbol	Name	Explanation/example
p_θ	Model distribution	The conditional distribution defined by an LM with parameters θ
s_θ	Model logit function	Assigns LM scores to tokens. Normalizing gives p_θ.
p_*	Training distribution	The target distribution for which an LM was trained.
A	Acceptability function	Takes a set S of outputs, assigns an acceptability score.
r	Reward function	Proxy for A.
g	A generator	E.g., a sampling algorithm for an LM.
q	Generator distribution	The distribution defined by a sampling algorithm.
q_*	Target distribution	E.g., the distribution of utterances in the style of a helpful assistant.
ϕ	A generator's parameters	E.g., temperature, number of samples.
d	Distance function	Measures distance, e.g., KL divergence, between two distributions.
$\mathcal{Y}$	Output space	The set of possible LM outputs, i.e., strings.
$\mathcal{V}$	Vocabulary	The set of tokens that make up sequences in $\mathcal{Y}$.
$\mathcal{P}(\mathcal{Y})$	Probability distributions	The set of probability distributions over outputs.
y_t	Current token	The token at index t in a sequence $y \in \mathcal{Y}$.
$y_{<t}$	Prefix, or context	The tokens preceeding index t in a sequence $y \in \mathcal{Y}$.

Table 1: An overview of symbols for convenience.

methods' theoretical motivation, practical impact, commonalities, and provide a unified frame for discussion. These methods generally require some degree of access to the model's internals.

A growing set of methods operate over partial or full sequences rather than individual tokens. These *meta-generation* algorithms have emerged from several communitites, including researchers interested in designing new decoding algorithms or prompting methods, as well as researchers interested in language model alignment and reasoning. Works from these communities often have different motivations and use different terminology. We present a unified picture in §4, classifying them according to their *programmatic structure* (e.g., parallel generation, search, or refinement), and discussing their motivations.

In addition to wanting a high-quality output, we often care about the *efficiency* of generation. We consider two definitions of efficient generation. In §6 we consider the token cost of generation algorithms, which is especially relevant to those using API-access models that charge by the token. In §7, we discuss methods for speeding up generation primarily from a systems perspective, where access to the model weights is assumed and latency and throughput are the key considerations. In this section, we draw upon work primarily from the machine learning systems (MLSys) community. The section serves as both an introduction to this area for machine learning researchers whose work does not focus on systems, and a practical exploration of tools for speeding up generation. We include a review of libraries that implement the described techniques.

We conclude the survey by discussing takeaways, broader directions, and future work in §8.

2 Preliminaries

Generation algorithms are used to produce outputs from a trained language model. Language models are probabilistic models over sequences, $p_\theta(y|x)$, and most generation algorithms attempt to either find highly probable sequences or sample from the model's distribution. A natural question is *why are sophisticated generation algorithms needed at all?* For example, we might imagine that simply sampling once from the model's unmodified output distribution, $y \sim p_\theta(y|x)$ is sufficient. We begin by defining some terminology (summarized in Table 1), and then present general goals of generation which shed some light on this question.

2.1 The user's goal in generation

When a user is generating outputs with a language model, it may be with one or more goals in mind. The user may want output that is as high quality as possible for some notion of quality, such as a correct answer to a math problem or a factual and well-written summary. The user may want multiple outputs, such as alternative solutions to a problem or multiple summaries to read through and synthesize. In general,

users now access language models through general-purpose text-in text-out APIs, making it impossible to enumerate all of the specific use cases or goals that a user might have.

As a result, to formalize an overall goal for generation, we will need to take a fairly general perspective. We assume that the user has some underlying measure of "acceptability" for any set S of outputs, $A(S) \in \mathbb{R}$. For example, a single sequence set may have high acceptability if it represents a correct solution to a problem, while in a different context a set S may have high acceptability if it balances some notion of diversity with some notion of quality. The acceptability scores, when normalized, form a probability distribution that we call the *target distribution* q_*,

$$q_*(S) \propto A(S). \tag{1}$$

Next, we treat generating outputs with a language model as sampling from a *generator* $S \sim g$ that produces a set of sequences each time it is called. Finally, we assume that a user wants the distribution of outputs from the generator to be "close" to the distribution of their acceptability scores according to some proximity measurement d between distributions. An ideal generator g would thus satisfy:

$$\arg\min_g d(q_*, g). \tag{2}$$

In practice, we typically do not know how to measure the user's acceptability nor their desired notion of proximity, let alone how to design a generator that is guaranteed to produce outputs with high acceptability. At a high level, the remainder of this survey can be seen as surveying ways to design generators that optimize some proxy of acceptability in an efficient way. For example, some algorithms will try to produce a single output that is acceptable with a language model's probability as a proxy of acceptability. Other algorithms will try to directly sample from some target distribution that we may interpret as being a proxy to a user's target distribution. To begin with, let us go into more detail on what a "generator" is, starting with the definition of a language model, a generation model, and a generation algorithm.

2.2 The modeling problem

Language models. Let p_θ be a language model that approximates the distribution p_*, denoted $p_\theta \approx p_*$. We consider autoregressive language models $p_\theta(y|x) = \prod_{t=1}^{T} p_\theta(y_t|y_{<t}, x)$, where y is a sequence of tokens from vocabulary $\mathcal{V}$. Each conditional distribution is of the form, $p_\theta(\cdot|y_{<t}, x) = \exp(s_\theta(\cdot|y_{<t}, x))/Z$, where $s_\theta(\cdot|y_{<t}, x) \in \mathbb{R}^{|\mathcal{V}|}$ are referred to as logits and $Z = \sum_{i=1}^{|\mathcal{V}|} \exp\left(s_\theta(\cdot|y_{<t}, x)\right)_i$. We henceforth refer to a model of this form as simply a language model (LM) for brevity.

A **generation model associated with a language model** p_θ is a function $g : \mathcal{X} \times \Theta \times \Phi \to \mathcal{P}(\mathcal{Y})$ that maps an input $x \in \mathcal{X}$, a model p_θ with $\theta \in \Theta$, and any additional parameters $\phi \in \Phi$ to a probability distribution over outputs, $q(y|x; p_\theta, \phi) \in \mathcal{P}(\mathcal{Y})$.

Calculating the probability distribution over outputs $q(y|x; p_\theta, \phi) \in \mathcal{P}(\mathcal{Y})$ is in most situations analytically intractable. One can use the **generation algorithm** in order to obtain independent or dependent samples from $q(y|x; p_\theta, \phi) \in \mathcal{P}(\mathcal{Y})$; we refer to this process as **generating** $y \sim q(y|x; p_\theta, \phi))$. We will also refer to g as a **generator** and q as a **generation distribution**. Generation algorithms may be deterministic or stochastic. While methods to maximize a scoring function are often deterministic and methods for generating sets of outputs are often stochastic, in practice each kind of method can be used toward either goal.

Let us now return to the general goal of generation that we formulated above. For notational simplicity, let us consider generating a single sequence, i.e. $S = \{y\}$. In practice, we can design a generation algorithm to maximize some proxy $r(\cdot)$ of acceptability:

$$\arg\max_g r\left(q_*(\cdot|x), g(\cdot|x; p_\theta, \phi)\right), \tag{3}$$

where $r : \mathcal{P}(\mathcal{Y}) \times \mathcal{P}(\mathcal{Y}) \to \mathbb{R}$ is some reward function between distributions. We group generation methods into 3 categories: methods for maximization (§2.2.1), sampling from the model (§2.2.2), and sampling from a target distribution (§2.2.3). These are special cases of (3). For maximization, $q_* \propto v$ for a scoring function v, and we have $r(q_*, g) = \mathbb{E}_{y\sim g} q_*(y|x)$. For sampling from a target distribution, r is a divergence between the target distribution and the generation distribution.

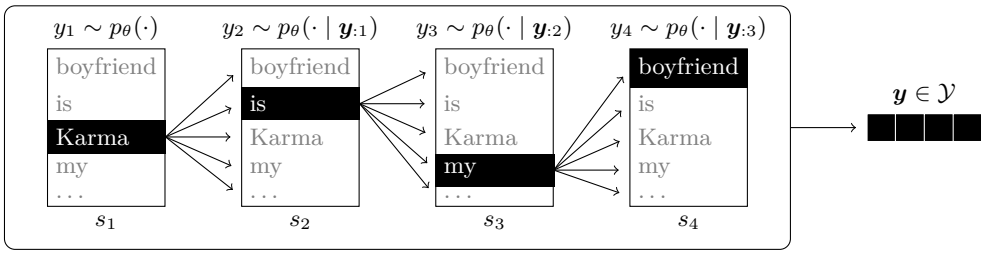


Figure 2: Sampling algorithms choose the next token at each time step s_i by sampling from the conditional distribution $p_\theta(\cdot \mid \boldsymbol{y}_{:i-1})$ and appending it to the context.

2.2.1 Maximization

We define *decoding* as the process of maximizing a score either deterministically or with a high probability:

Definition 1 (Score maximizing algorithm)**.** A score maximizing algorithm for score $v : \mathcal{S} \to \mathbb{R}$ refers to an algorithm that approximates:

$$f(x; p_\theta, \phi) = \arg\max_{S \in P(\mathcal{Y})} v(S), \tag{4}$$

where $P(\mathcal{Y})$ is the set of all subsets of $\mathcal{Y}$. Decoding algorithms can be greedy algorithms, concave maximization algorithms, combinatorial optimization algorithms, or stochastic algorithms.

2.2.2 Sampling

Some algorithms are designed to sample from a distribution. The samples can then be used for any purpose, such as for maximizing some task-specific metric (e.g., a metric that balances quality and diversity).

Definition 2 (Sampler for $q(p_\theta)$)**.** A sampler for $q(p_\theta)$ gives a sample from some distribution q_p proportional to $q(p_\theta)$, where q is a map between probability distributions:

$$y \sim q_p \propto q(p_\theta(y|x)). \tag{5}$$

2.2.3 Sampling from a specified target distribution

In some cases we can specify which target distribution q_* an algorithm is aiming to sample from.

Definition 3 (Sampler for a target distribution)**.** A generator $q = g(x, p_\theta, \phi)$ is called a sampler for target distribution q_* if it approximates:

$$\max_q -\mathrm{D_{KL}}\left(q_*(\cdot|x) \| q(\cdot|x; p_\theta, \phi)\right).^1 \tag{6}$$

An optimal sampling algorithm for q_* yields an unbiased sample $y \sim q_*(\cdot|x)$.

Next, we will see examples of algorithms that achieve these goals by generating token-by-token.

3 Token-level generation algorithms

In this section, we discuss representative methods that operate on the token-level (e.g. by sampling a single token at a time, or constructing a token-level search space and then selecting an output at the end).

Methods in this category generally assume access to a language model's logits, next-token distributions, or probability scores. These methods will later be treated as black boxes that are called by meta-generators.

[1]Any divergence d with the property that $q_* = q$ iff $d = 0$ is suitable.

3.1 MAP decoding algorithms

When faced with the question of what sequence to choose from the distribution defined by a language model, a natural objective is to choose the *most likely sequence*. Several popular decoding algorithms therefore attempt to find a generation y that maximizes $p_\theta(y|x)$, referred to as maximum a-posteriori (MAP) decoding.

Definition 4 (MAP decoding algorithms)**.** A maximum a-posteriori (MAP) decoding algorithm approximates

$$f(x; p_\theta, \phi) = \arg\max_{y \in \mathcal{Y}} p_\theta(y|x). \tag{7}$$

The term MAP comes from viewing p_θ as a posterior over outputs y given the observed input x, and *decoding* comes from information theory.

Greedy decoding. Arguably the simplest MAP decoding algorithm is *greedy decoding*, which generates a sequence $\hat{y}_1, \ldots, \hat{y}_T$ by recursively selecting the highest probability token from the next-token distribution:

$$\hat{y}_t = \arg\max_{y_t \in \mathcal{V}} p_\theta(y_t|\hat{y}_{<t}, x), \tag{8}$$

for $t = 1, \ldots, T$, with T determined by a stopping condition (e.g., a fixed T or y including a certain string). Greedy decoding is an *approximate* MAP decoding algorithm, meaning that it finds a sequence that is not necessarily a maximizer of (7). Specifically, it approximates (7) as:

$$\arg\max_{y \in \mathcal{Y}} p_\theta(y|x) = \arg\max_{(y_1,\ldots,y_T) \in \mathcal{Y}} \prod_{t=1}^{T} p_\theta(y_t|y_{<t}, x) \tag{9}$$

$$\approx \left(\hat{y}_1 = \arg\max_{y_1 \in \mathcal{V}} p_\theta(y_1|x), \cdots, \hat{y}_T = \arg\max_{y_T \in \mathcal{V}} p_\theta(y_T|\hat{y}_{<T}, x) \right). \tag{10}$$

Despite its naive approximation, greedy decoding is a widely-used generation algorithm. For instance, it is used in Google's Gemini report (Gemini Team et al., 2023), and is available on typical language model APIs.

Other MAP decoding algorithms. Several algorithms have been designed that typically return better approximations (i.e., more probable sequences) than greedy decoding. In the context of neural sequence-to-sequence models, *beam search* (Graves, 2012; Sutskever et al., 2014) is a widely-studied MAP decoding algorithm. It maintains a data structure of multiple prefixes $y_{<t}$ at each generation step, expands each prefix with each possible next-token, $y_{<t} \circ y_t$, scores each expanded prefix with $p_\theta(y_{<t} \circ y_t|x)$, and retains the top-$K$ expanded prefixes for the next iteration. This can be seen as a generalization of greedy decoding, which expands only a single prefix. In practice, beam search has been shown to improve upon greedy decoding in terms of downstream task performance in many settings (e.g., Sutskever et al. (2014); Freitag & Al-Onaizan (2017); Kulikov et al. (2019)). It has several variations and generalizations that we will return to when we take the perspective of generation as search (§4.3). Although the space of possible outputs is extremely large, it is sometimes possible to find an *exact* MAP solution (i.e., a sequence that maximizes (7)). For instance, Stahlberg & Byrne (2019) combine elements of beam search and depth-first search to perform exact search with machine translation models, which was improved upon in Stahlberg et al. (2022).

Pitfalls of MAP decoding. Despite its popularity, several studies suggest that the MAP decoding objective is not desirable (Meister et al., 2020). Empirically, MAP decoding has a tendency to produce degenerate results. For example, Koehn & Knowles (2017) found that wide beam search (which approaches exact MAP decoding in the limit) degrades neural machine translation (NMT) outputs by favoring shorter outputs. In fact, Stahlberg & Byrne (2019) found that exact MAP decoding often returned the *empty sequence* in NMT. Length normalization (e.g., dividing the log-probability of the sequence by its length) can mitigate MAP decoding's tendency to favor shorter sequences (see Murray & Chiang, 2018), but this is only a heuristic and does not fully counteract degradation for the largest beam sizes (Koehn & Knowles, 2017). Approximate

MAP decoding, e.g., greedy, can also fail by getting trapped in repetitive sequences (Holtzman et al., 2020; Welleck et al., 2020; Eikema & Aziz, 2020).

There are several explanations for degenerate behavior in MAP decoding, a phenomenon known as the *inadequacy of the mode* (Eikema, 2024). Some studies attribute degenerative phenomena in MAP decoding to the tendency of the most likely generations to accumulate so little probability that the mode becomes arbitrary due to small errors in probability estimation (Eikema & Aziz, 2020; Stahlberg et al., 2022). In an alternative explanation, Meister et al. (2023b) use information-theoretic analysis to show that MAP decoding generations often fall outside of the *typical set* of sequences in the language model's distribution. To illustrate how this occurs, consider that the most probable outcome of 100 flips of a slightly biased coin (with 0.51 probability of heads, 0.49 probability of tails) is a sequence of 100 heads. However, this result would be atypical; a close-to-even mix of heads and tails would be more typical (Dieleman, 2020).

Unreasonable effectiveness of approximate MAP decoding. Despite the drawbacks of MAP decoding, rough approximations of MAP decoding remain popular in the forms of greedy decoding and narrow beam search. Meister et al. (2020) hypothesize that these decoding methods are effective because they inadvertently enforce information-theoretic patterns that are characteristic of human text.

3.2 Sampling and adapters

A popular alternative to the MAP objective is to sample directly from the language model's distribution $y \sim p_\theta(y|x)$.

Ancestral sampling. The most basic sampling algorithm for p_θ is motivated by the fact that autoregressive models decompose sequence probabilities into a product of next-token conditionals:

$$p_\theta(y|x) = \prod_{t=1}^{|y|} p_\theta(y_t|y_{<t}, x), \tag{11}$$

where $y = (y_1, \ldots, y_{|y|})$ and y_t are individual tokens. As shown in Figure 2, sampling from this model can be done recursively,

$$y_t \sim p_\theta(\cdot|y_{<t}, x), \tag{12}$$

where y_0 is a given starting token, and the algorithm terminates upon reaching a particular token or a given length. The result is mathematically equivalent to sampling a sequence y directly from $p_\theta(\cdot \mid x)$, and is known as *ancestral sampling*. Other algorithms such as speculative sampling (Leviathan et al., 2022) aim to sample from p_θ more efficiently, which we will discuss in more detail later in the review (§7).

Sampling, MAP, and the diversity-coherence trade-off. Ancestral sampling avoids many of the degenerate behaviors of MAP decoding, such as repetition traps, and introduces more diversity into LM generations. However, ancestral sampling can suffer from *incoherence*, i.e., over-sampling highly-unlikely tokens due to model error (Zhang et al., 2021). Hewitt et al. (2022) hypothesize that this occurs because perplexity-based loss functions encourage language models to over-estimate the probability of unlikely tokens to avoid large loss penalties (a behavior called mode-seeking). Alternatively, Finlayson et al. (2024a) hypothesize that constraints imposed by the LM's output layer, i.e., the softmax bottleneck (Yang et al., 2018), cause model errors, and propose a method, basis-aware truncation (BAT), to avoid these errors.

Balancing the diversity-coherence tradeoff. Several decoding strategies attempt to balance the diversity-coherence tradeoff by interpolating between greedy and ancestral sampling. These include nucleus (Holtzman et al., 2020), top-k (Fan et al., 2018), and η- and ϵ-sampling (Hewitt et al., 2022), which use various heuristics to choose a threshold at each time step and only sample tokens with probability greater than the threshold. Another approach, temperature sampling (Ackley et al., 1985; Hinton et al., 2015), scales the LM logits to interpolate between greedy sampling and uniform sampling (setting all token probabilities equal), which can be useful when one wants *more* diversity than ancestral sampling offers.

Method	Purpose	Adapter	Extrinsic
Ancestral sampling	$y \sim p_\theta$	–	–
Temperature sampling [1]	$y \sim q(p_\theta)$	Rescale	–
Greedy decoding	$y \leftarrow \max p_\theta$	Argmax (temperature$\rightarrow 0$)	–
Top-k sampling [53]	$y \sim q(p_\theta)$	Truncation (top-k)	–
Nucleus sampling [82]	$y \sim q(p_\theta)$	Truncation (cumulative prob.)	–
Typical sampling [143]	$y \sim q(p_\theta)$	Truncation (entropy)	–
Epsilon sampling [78]	$y \sim q(p_\theta)$	Truncation (probability)	–
η sampling [78]	$y \sim q(p_\theta)$	Truncation (prob. and entropy)	–
Mirostat decoding [12]	Target perplexity	Truncation (adaptive top-k)	–
Basis-aware sampling [54]	$y \sim q(p_\theta)$	Truncation (linear program)	LP Solver
Contrastive decoding [121]	$y \sim q(p_\theta)$	$\log p_{\theta'} - \log p_\theta$ and truncation	Model $p_{\theta'}$
DExperts [129]	$y \sim q_*(\cdot\|x, c)$	$\propto p_\theta \cdot (p_{\theta+}/p_{\theta-})^\alpha$	Models $p_{\theta+}, p_{\theta-}$
Inference-time adapters [137]	$y \sim q_* \propto R(y)$	$\propto (p_\theta \cdot p_{\theta'})^\alpha$	Model $p_{\theta'}$
Proxy tuning [130]	$y \sim q_*(\cdot\|x, c)$	$\propto p_\theta \cdot (p_{\theta+}/p_{\theta-})^\alpha$	Models $p_{\theta+}, p_{\theta-}$

Table 2: Survey of token-level generation. $R(y)$ is a scalar reward function. c is a control attribute. Extrinsic refers to a model or solver separate from the underlying language model p_θ.

3.3 Token-level sampling adapters

Except for beam search, all of the token-level sampling methods discussed so far can be viewed as *sampling adapters* q_t (Meister et al., 2023a) which adjust each next-token distribution,

$$y_t \sim q_t(p_\theta(y_t|y_{<t}, x)). \tag{13}$$

Example 1 (Temperature sampling as an adapter)**.** Temperature sampling adjusts the distribution by dividing the logits by a scalar *temperature* parameter τ:

$$q_t(y_t|y_{<t}, x; p_\theta, \tau) \propto \exp\left(s_\theta(y_{<t}, x)/\tau\right). \tag{14}$$

Sending τ to 0 yields greedy decoding, $\tau = 1$ yields ancestral sampling, and $\tau > 1$ approaches uniform sampling (all tokens have the same probability).

Many other token-level decoding methods can be cast as sampling adapters, including methods that reweight logits with outputs from another model (Liu et al., 2021a; Li et al., 2023a), and a variety of other transformations summarized in Table 2. Many of these token-level generation algorithms assume access to the language model's next-token distributions. In practice, next-token distributions are increasingly not provided by common generation APIs, both for practical reasons and for security (Finlayson et al., 2024b; Carlini et al., 2024). Instead, token-level algorithms are often implemented by the API provider, and used by setting hyperparameters (e.g., setting a temperature τ).

Adapters for statistical control. Several decoding methods use sampling adapters to control the statistical and information-theoretic properties of model outputs and align them with those of human text. These include locally typical sampling (Meister et al., 2023b), which aims to sample from the LM distribution's typical set (MacKay, 2004); and mirostat sampling (Basu et al., 2021), which attempts to match the perplexity of the generated text to the expected perplexity under Zipf's law (Zipf, 1999; Powers, 1998).

Autoregression and lookahead adapters. Token-level algorithms generate from left-to-right, meaning that they generate each token without knowing the eventual identity of tokens to the right. Several algorithms have incorporated various heuristic scores $v(y_{\leq t})$ that adjust the next-token distribution using information from potential *future* tokens. This includes explicitly generating several tokens ahead (e.g., Lu et al. (2022); Leviathan et al. (2022)), or learning a function $v_\phi(y_{\leq t})$ that predicts a property of a full sequence (e.g., its style score or correctness) (Yang & Klein, 2021). Doing so can aid in satisfying sequence-level criteria.

Distribution adjustment with another language model. Some algorithms adjust the next-token distribution using another language model. This can arise from several motivations, including removing abnormalities in the model's next-token distributions (Li et al., 2023a), speeding up generation (Leviathan et al., 2022), or shifting the generation distribution to one with a property (e.g., a style) (Liu et al., 2021a).

3.4 Controlled generation

Many scenarios can be framed as aiming to sample from a language model's distribution modulated by a sequence-level criterion $c(y)$ (Korbak et al., 2022a;c; Hu et al., 2024; Zhao et al., 2024a):

$$q_* \propto p_\theta(y|x)c(y). \tag{15}$$

For example, $c(y)$ may assign high values to sequences with a particular style, or low values to sequences with toxic content or buggy code. Another way of phrasing (15) is sampling from a particular energy-based model (LeCun et al., 2006; Khalifa et al., 2021). We discuss three examples based on the structure of $c(y)$.

Classifier. In some cases $c(y)$ is a classifier $p(a|x, y)$, which predicts the probability that y contains an "attribute" a, such as a style or non-toxicity. The goal is then to sample from:

$$q_* \propto p_\theta(y|x)p(a|x, y)^\beta, \tag{16}$$

where β is a hyperparameter assigning more weight to the classifier at higher values of β. Various generation algorithms have been developed for this purpose, such as approximations based on reweighting next-token distributions with other language models (Liu et al., 2021a), reweighting with a learned classifier that approximates the sequence-level classification $p_\phi(a|y_{<t}, x) \approx p(a|y, x)$ (Yang & Klein, 2021), or additional training to sample from q_* (Khalifa et al., 2021; Hu et al., 2024; Zhao et al., 2024a).

Indicator. A special case is $c(y)$ indicating whether y falls into a target set Y_x^*, such as the set of correct solutions to a reasoning problem, or sequences that have desired keywords. The goal is then to sample from:

$$q_* \propto p_\theta(y|x)\mathbb{I}[y \in Y_x^*], \tag{17}$$

where $\mathbb{I}[y \in Y_x^*]$ is 0 when $y \in Y_x^*$ and 1 when $y \notin Y_x^*$. Various generation algorithms incorporate a learned verifier $v_\phi(x, y) \approx \mathbb{I}[y \in Y_x^*]$ to aid in achieving this goal (Cobbe et al., 2021; Lightman et al., 2024), or design beam search heuristics for the case of desired keywords (Hokamp & Liu, 2017; Lu et al., 2022).

There is a clear connection between sampling from a target distribution of the form (17) and maximizing a scoring function (§2.2.1): sampling from (17) maximizes $v(y) = \mathbb{I}[y \in Y_x^*]$, e.g., correctness.

Reward. An important case is when $c(y)$ is governed by a *reward function* $r(x, y) \to \mathbb{R}$:

$$q_* \propto p_\theta(y|x) \exp\left(\frac{1}{\beta} r(x, y)\right), \tag{18}$$

where $\beta \in \mathbb{R}$ interpolates between sampling from p_θ ($\beta \to \infty$) and maximizing reward ($\beta \to 0$).

A notable example is aligning the distribution of generated text with a distribution of text preferred by humans (Ziegler et al., 2020; Stiennon et al., 2020; Ouyang et al., 2022). One way of operationalizing this problem is as one of finding a policy π that balances maximizing a reward $r(x, y)$ that quantifies human preferences with generating sequences that are probable under a pretrained model p_θ:

$$\max_\pi \mathbb{E}_{x\sim p, y\sim p_\theta(y|x)}[r(x, y)] - \beta \mathrm{KL}\left(\pi(y|x) \| p_\theta(y|x)\right). \tag{19}$$

The policy that maximizes the above objective is (Korbak et al., 2022b; Rafailov et al., 2023):

$$q_*(y|x) = \frac{1}{Z(x)} p_\theta(y|x) \exp\left(\frac{1}{\beta} r(x, y)\right), \tag{20}$$

where $Z(x)$ is a normalization factor. One strategy for sampling from q_* is updating p_θ with reinforcement learning, then using ancestral sampling. This strategy is referred to as reinforcement learning from human feedback (Askell et al., 2021). Later, we will discuss meta-generation algorithms for addressing this problem.

A second approach is to re-weight each next-token distribution during autoregressive sampling. For example, reward-augmented decoding (Deng & Raffel, 2023) assumes access to a reward function $r(y_{\leq t}, x)$ that assigns a scalar reward to partial generations $y_{\leq t}$. It re-weights the tokens with the top-k next-token probabilities using the reward, and samples from the re-weighted distribution. That is,

$$y_t \sim \text{softmax}\left(s_\theta^{1:k}(y_{<t}, x) + \beta r^{1:k}\right), \tag{21}$$

where $s_\theta^{1:k}(y_{<t}, x) \in \mathbb{R}^k$ are the top-k logits at timestep t, $r^{1:k} \in \mathbb{R}^k$ are the rewards evaluated after appending each of the top-k tokens to the prefix $y_{<t}$, and $\beta \in \mathbb{R}$ is a hyper-parameter. Inference-time policy adapters (Lu et al., 2023) directly optimizes an "adapter" language model to adjust a base language model's next-token distributions so that the combined model's generations receive higher rewards. Specifically, an adapter language model p_ϕ is combined with a base language model to form a "tailored policy",

$$p(\cdot|y_{<t}, x) \propto p_\theta(\cdot|y_{<t}, x)p_\phi(\cdot|y_{<t}, x), \tag{22}$$

and the tailored policy is updated with reinforcement learning while freezing the base model p_θ's parameters.

In summary, we have seen several strategies for constructing a token-level search space and adjusting the next-token distributions of a model during sampling. Next, we will treat these algorithms as black-boxes that can be used to generate partial or full sequences, and survey algorithms that construct search spaces on the (partial-)sequence level or operate by drawing multiple samples.

4 Meta-generation algorithms

Some generation algorithms have the distinctive property of requiring access to a separate generation subroutine. For instance, best-of-N calls a generator to sample N sequences from the language model. This sub-generator is interchangable; it can be freely chosen from top-k, temperature sampling, or any other sequence generator. We coin the term *meta-generation* to describe algorithms that call sub-generators, and identify four common patterns among meta-generators. In particular, we find that they can be classified into the categories of chained, parallel, step-level, and refinement-based meta-generators.

4.1 Chained meta-generators

The first programmatic pattern chains multiple generators together. We start by explaining this idea in the context of prompted language models.

Chaining prompted language models. It is increasingly common to perform input-output tasks with a language model by specifying a prompt z,

$$y = f(x; p_\theta, z, \phi), \tag{23}$$

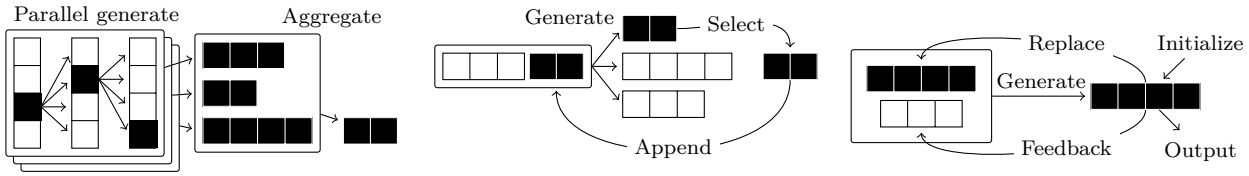
where $f(\cdot)$ is a generation algorithm, and the prompt z is a sequence of tokens that specifies the desired behavior through a natural language instruction or input-output examples (Brown et al., 2020; Ouyang et al., 2022). For instance, given $z =$ `multiply the two numbers` and $x =$ 1432 293, we can generate an output y that contains an (attempted) solution. It is natural to compose the generator call with other operations, such as composing a generator that outputs Python code with a function that executes Python code.

Similarly, it is natural to combine multiple calls to generators, e.g., generating a story using:

$$y = f_3 \circ f_2 \circ f_1, \tag{24}$$

where f_1 generates a story outline, f_2 fills in the sections, and f_3 revises the story to meet a length constraint. Notice that the composition is itself a generation algorithm,

$$f(x; p_\theta, (f_1, f_2, f_3)), \tag{25}$$



(a) Parallel search (b) Heuristic step-level search (c) Refinement

Figure 3: Three meta-generation patterns.

Algorithm	Aggregation type	Scoring / transforming with
Best-of-N 23	Rerank	LLM score or external score
Noisy-channel 152	Rerank	Log-linear combination score
Majority voting 10	Transform	Empirical vote frequency
Weighted majority voting 200	Transform	Empirical distribution over answers
Self-consistency 205	Transform	Marginal distribution over answers
Universal self-consistency 29	Transform	Answer aggregation using an LLM generator
Branch-Solve-Merge 174	Transform	Answer aggregation using an LLM generator / rule-based parsing
QE-fusion 202	Transform	Answer contains spans from candidates

Table 3: Parallel meta-generators.

i.e., a mapping from an input x, model p_θ, and other parameters ϕ, to an output (here, ϕ contains the generation algorithms f_1, f_2, f_3), or in general, a distribution over outputs $q(y|x, p_\theta, \phi)$. In general, we can view calls to generation algorithms as steps in a *program* whose execution yields a generated output. We refer to a program $f(x; p_\theta, F)$, which calls generation algorithms $f' \in F$, as a *meta-generation algorithm*.

Related ideas appear in the literature under various names, including language model cascade (Dohan et al., 2022), LLM program (Schlag et al., 2023), and recently, scaffolding program (Zelikman et al., 2024b). We introduce the term meta-generation as an abstraction that is agnostic to the implementation of the underlying generator model(s) (which need not be LLMs), and to clarify the connection with other generation algorithms.

Problem decomposition. A variety of algorithms have adopted the chain pattern in order to decompose an input-output problem into multiple steps, with each step implemented by a language model or external function. For instance, Self-Ask (Press et al., 2023) alternates between prompting a language model to generate a sub-question and calling a search engine to find an answer to the question. System 2 Attention (Weston & Sukhbaatar, 2023) uses multi-step generation to help the model refrain from attending to irrelevant information. More generally, a number of tools such as LangChain (Chase, 2022) and MiniChain (Rush, 2023) provide domain-specific languages for declaring and executing chains involving prompted language models.

4.2 Parallel meta-generators

Another pattern is to generate multiple trajectories in parallel, then merge the resulting terminal states to arrive at a final generated sequence. For instance, various *sequence-level generation algorithms* generate an *N-best list* $\{y^{(n)}\}_{n=1}^N \sim g$, then apply an *aggregation* function $h(y^{(1)}, \ldots, y^{(N)})$ to arrive at a final generated sequence. The N-best list of sequences might come from sampled generations, a beam search algorithm, or any other generator $y \sim g$ that generates full sequences. We discuss aggregation functions that rerank (§4.2.1) or transform (§4.2.2) the N-best list, then discuss sequence-level statistical rejection sampling (§4.2.3). Table 3 presents a brief summary of algorithms from the classes that we discuss.

4.2.1 Reranking algorithms

Reranking (or rescoring) is a classical approach (Collins, 2000; Huang & Chiang, 2007) originally developed for parsing and automatic speech recognition to achieve a trade-off between the computational complexity of MAP decoding and its tendency to rule out good hypotheses. A reranking algorithm orders an N-best list with a *reranking function* $h(y^{(1)}, \ldots, y^{(N)}) \to (y^{\sigma(1)}, \ldots, y^{\sigma(N)})$, then selects the top-$k$ ranked sequences. Reranking has recently found new applications in text generation (e.g., Cobbe et al. (2021); Stiennon et al.

(2020); Krishna et al. (2022); Ni et al. (2023); Lightman et al. (2024)) by using various reranking functions and various sources of data to learn the reranking functions. A simple and effective method is *best-of-N*.

Best-of-N. Best-of-N (Charniak & Johnson, 2005; Pauls & Klein, 2009) refers to generating an N-best list and picking the best sequence according to a scoring function.

Definition 5 (Best-of-N: BoN$(x, g, v, N; \phi)$)**.** Let g be a generation algorithm with output space $\mathcal{Y}$, and $v : \mathcal{Y} \to \mathbb{R}$ a scoring function. Assume that $\epsilon \in \phi$ governs the randomness in g. The best-of-N generation algorithm is defined as:

$$f(x, g, N, \phi) = \underset{y^{(n)}|n\in\{1,\ldots,N\}}{\arg\max} \left\{ v(y^{(n)}) \mid y^{(n)} \sim g(\cdot|x), n \in \{1, 2, \ldots, N\} \right\}, \tag{26}$$

where each $y^{(n)}$ is a generated sequence.

Best-of-N can be performed with any algorithm that can be used to generate a list of N sequences, including temperature sampling, beam search, Viterbi decoding, or many others. In the context of language modeling, best-of-N was developed for parsing (Charniak & Johnson, 2005; Pauls & Klein, 2009), and traditionally involved modifying a decoding algorithm originally developed to find the top-1 hypothesis so that it obtains the top-N highest scoring decodings. An attractive property is that Best-of-N usually incurs only a linear increase in computational complexity compared to top-1 decoding. In the context of LLMs, best-of-N is amenable to black-box generators (e.g., accessed via an API), since it does not require knowledge of the generator for populating the N-best list. Modern instances of best-of-N use learned scoring functions that are often themselves parameterized by LLMs. We discuss examples from reasoning and preference alignment.

Best-of-N in reasoning. In some settings the goal is to generate correct sequences, such as a correct solution to a mathematical problem or a program that passes test cases. A common approach in these cases is to learn a *verifier* $v_{\theta'}(y) \to [0, 1]$ that predicts the probability that an output y is correct, and use it within Best-of-N. Doing so has seen success in mathematical reasoning (e.g., Cobbe et al. (2021); Uesato et al. (2022); Lightman et al. (2024)), code generation (Ni et al., 2023), and other settings with similar properties. Naturally, the performance depends on the quality of the verifier, which we return to in (§5.1).

Best-of-N in alignment. Previously in §3.4, we discussed how the problem of aligning the distribution of generated text with a distribution of text preferred by humans can be framed as sampling from

$$q_*(y|x) = \frac{1}{Z(x)} p_\theta(y|x) \exp\left(\frac{1}{\beta} r(x, y)\right). \tag{27}$$

When a single high-reward sequence is desired (e.g., at low values of β), a natural strategy is to use best-of-N with a learned approximation of the reward, $r_\phi(x, y)$, as the scoring function. In practice, this strategy is an effective alternative to reinforcement learning from human feedback (RLHF) methods (Gao et al., 2022; Beirami et al., 2024). For example, AlpacaFarm (Dubois et al., 2023) found that Best-of-1024 with a human-preference reward model was competitive with more standard decoding methods with a model trained using RLHF. A potential benefit is that Best-of-N does not require updating the model p_θ's parameters, at the expense of generation-time compute.

Best-of-N depends on the quality of the reward function, which is typically a learned function $r_\phi(x, y)$. It can suffer from *reward over-optimization*–i.e., returning an undesired sequence that nevertheless receives high reward. Specifically, suppose that $q_*(y|x) \propto r_*(y)$, where r_* perfectly captures the desired outcome of generation. Best-of-N at high values of N can be seen as approximating:

$$\arg\max_{y\in\mathcal{Y}} q_*(y|x) \approx \underset{y_n|n\in\{1,\ldots,N\}}{\arg\max} r_\phi(x, y_n), \tag{28}$$

where $y_n \sim g$. In practice, the learned model r_ϕ typically does not match r_*, especially on out-of-distribution sequences, so best-of-N may find sequences that "overoptimize" the reward.

Noisy-channel reranking in Neural Machine Translation. A wide range of reranking methods precede the era of large language models. A classic approach is a noisy-channel model (Brown et al., 1993). *Noisy-channel* means that the observed output from the system (e.g., a machine translation system) is distorted by some unknown noise pattern (i.e., noisy channel). If we consider $p_\theta(y|x)$ as the probability of the translation y of the source language text x, then Bayes rule suggests the following relationship: $p_\theta(y|x) \propto p(x|y)p(y)$, where $p(x|y)$ is a channel-model, and $p(y)$ is the target language LM.

As an example from the literature, Och & Ney (2002); Ng et al. (2019) propose to use the following log linear combination to rerank translation candidates in beam search:

$$s_{\text{noisy-channel}}(y) = \log p(y|x) + \lambda_1 \log p(x|y) + \lambda_2 \log p(y), \tag{29}$$

where the log-linear coefficients λ_1 and λ_2 are tuned empirically on a development set. Therefore, the reranking function h in this case is defined so that the order of candidates is given by a decreasing order of noisy channel scores $s_{\text{noisy-channel}}$ computed for every translation candidate.

4.2.2 Transformation algorithms

In contrast to reranking elements of the N-best list, other algorithms transform the list into a new sequence which might not be part of the N-best list itself. For instance, mathematical question answering is an example of a task where the potential outputs (answers to math questions) are produced as *part* of a much longer decoded sequences from the LLM. In other cases we might draft N summaries, then synthesize them into a new, final summary. This requires a transformation of the N summaries rather than a simple reranking.

Majority voting. First, this algorithm processes an N-best list $(y^{(1)}, \ldots, y^{(N)})$ and counts how each of the candidates $y^{(i)}$ votes towards a different set of outputs $(a_1, \ldots, a_K)$:

$$h(y^{(1)}, \ldots, y^{(N)}) \rightarrow (v(y^{(1)}), \ldots, v(y^{(N)})), \tag{30}$$

where $v : \mathcal{Y} \rightarrow 1, \ldots, K$ is a voting function that maps from sequence space to an output from $(a_1, \ldots, a_K)$. Second, it selects the output that received the largest number of votes:

$$\hat{a} = \arg\max_k \sum_{j=1}^{K} \sum_{i=1}^{N} \mathbb{I}(v(y^{(i)}) = j). \tag{31}$$

Weighted majority voting. Integer votes counted towards each output have a tendency to result in multiple outputs taking exactly the same number of votes. In such situations, votes assigned by every sequence from the N-best list can be associated with a scalar score rather than a count: $w(y^{(i)}) : \mathcal{Y} \rightarrow \mathbb{R}$. The final output selection is done by aggregating (e.g., summing) scores associated with the votes:

$$\hat{a} = \arg\max_k \sum_{j=1}^{K} \sum_{i=1}^{N} w(y^{(i)}) \mathbb{I}(v(y^{(i)}) = j). \tag{32}$$

Figure 5 shows a performance comparison of best-of-N, majority voting, and weighted majority voting on a standard mathematical reasoning benchmark.

Self-consistency. Wang et al. (2023b) proposed a probabilistic perspective on aggregating scores for each output using the probabilities of the N-best list of candidates associated with the given final output. As a concrete example, every candidate from $(y^{(1)}, \ldots, y^{(N)})$ is expected to have a *trailing* substring that matches at least one output from $(a_1, \ldots, a_K)$. Then the probability of the output a_i can be approximated by marginalizing over all candidates that ends with a_i:

$$p(a_i) = \sum_{j=1}^{N} p(a_i | \text{prefix}(y^{(j)})) \mathbb{I}(v(y^{(j)}) = i), \tag{33}$$

where we adapt the voting function $v(y^{(j)})$ to map to those outputs which correspond the ending substring of the given candidate. While being similar to weighted majority voting, this approach gives an explicit probability distribution over outputs. It is, however, not a hard task to normalize any given set of scores provided by weighted majority voting to obtain a proper distribution over outputs $(a_1, \ldots, a_K)$.

Minimum Bayes risk decoding. Rather than seeking the most probable sequence as in MAP decoding, minimum Bayes risk algorithms aim to find the best sequence in terms of a pairwise *utility* function $u(y, y')$:

Example 2 (Minimum Bayes risk algorithm)**.** A minimum Bayes risk (MBR) decoding algorithm refers to an algorithm of the form (Bickel & Doksum, 1977; Kumar & Byrne, 2004):

$$f(x) \triangleq \arg\max_{y' \in \mathcal{Y}} \sum_{y \in \mathcal{Y}} u(y, y') p_*(y|x), \tag{34}$$

where $u : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}$. A dependence on p_θ is introduced in specific instances of MBR decoding algorithms.

The MBR objective is motivated by decision theory. Intuitively, it can be thought of as seeking an output with highest average "similarity," as measured by the utility function u, to other candidates, particularly those assigned high probability under p_*.

Various algorithms provide approximate solutions to the Minimum Bayes Risk (MBR) objective. They typically consist of providing a metric $m(\cdot, \cdot) \to \mathbb{R}$, populating a *hypothesis set* $\mathcal{Y}_h$ using a generator, and populating a *evidence set* $\mathcal{Y}_e$ to estimate the risk of each hypothesis:

$$\hat{y} = \arg\max_{y' \in \mathcal{Y}_h} \frac{1}{\mathcal{Y}_h} \sum_{y \in \mathcal{Y}_e} m(y, y'). \tag{35}$$

The hypothesis set is typically akin to an N-best set, populated by calling a generator $\{y^{(n)}\}_{n=1}^N \sim g(\cdot|x)$. Simple strategies sample from the model, $y^{(n)} \sim p_\theta$. Others take the best-k outputs from a ranked list of generations, or use more sophisticated strategies such as iteratively adding hypotheses or transforming them (González-Rubio et al., 2011; González-Rubio & Casacuberta, 2013). Freitag et al. (2023) investigate the impact of the underlying sampling strategy, finding variation across strategies, with epsilon sampling performing best for machine translation. The evidence set is typically sampled from a generator, or set to the hypothesis set to save on computation. Finally, the metric impacts performance. MBR with a particular metric tends to inflate performance on that metric, sometimes by gaming it (Freitag et al., 2023), akin to our discussion of reward over-optimization.

MBR methods have a rich history in the machine translation and speech recognition literature (Goel et al., 2004; Heigold et al., 2005; GOEL, 2003; Kingsbury et al., 2012; Eikema & Aziz, 2020), and have also been applied across other tasks (Shi et al., 2022; Suzgun et al., 2023). Interestingly, Bertsch et al. (2023) show that self-consistency is a special case of MBR. In general, there are several other dimensions along which MBR methods are categorized. We refer the reader to Bertsch et al. (2023) for further in-depth study and taxonomy of MBR methods.

Generate-and-transform. In general, we can view the algorithms above as first generating an N best list, followed by transforming the N best list using a transformation $g(y^{(1)}, \ldots, y^{(N)})$ such as voting or one that internally estimates risk. Several other algorithms fall into this generate-and-transform pattern.

For instance, universal self-consistency (Chen et al., 2023c) prompts a language model to generate a final sequence given the N-best list, which can avoid the aforementioned issue of parsing sequences into an answer. Branch-solve-merge (Saha et al., 2023) transforms an input into N different prompts, generates with those prompts, then merges the results by providing the generations to a language model. Finally, Bertsch et al. (2023) show that several voting techniques are instances of MBR decoding methods.

4.2.3 Sequence-level rejection sampling

Previously we discussed the goal of designing a generation algorithm that samples from a target distribution q_* (§2.2.3). A related pattern is using a stochastic sequence generator $y \sim g$ to sample from q_* using rejection sampling. This involves sampling multiple sequences from g and is thus akin to a parallel meta-generator.

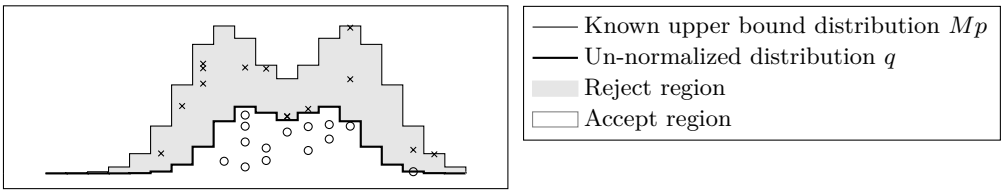


Figure 4: In rejection sampling, the aim is to sample from a distribution q whose normalizing constant is unknown. To do so, use a known distribution p that serves as an upper bound for the unknown distribution when scaled by a constant, i.e., for some constant M and all values y, $Mp(y) \geq q(y)$. Next, obtain a sample $y \sim p$ and accept this sample with probability $q(y)/Mp(y))$, otherwise reject the sample and repeat the process. This is equivalent to sampling from q.

Specifically, statistical rejection sampling is a technique for sampling from a target distribution q_* with an unknown normalizing constant. This is accomplished by first sampling from a known distribution $y \sim g$ which serves as an upper bound for q_*, (e.g., for some constant M, $Mg(y) \geq q_*(y)$), then accepting the sample with probability $q_*(y)/Mg(y)$. Figure 4 illustrates this process. Rejection sampling is a useful tool for sampling from a specified target distribution over an intractably large support, e.g., the set of sequences.

One example of *sequence-level* rejection sampling for LMs is sampling valid JSON strings from an LM. The space of valid JSON strings is infinite and the normalizing factor is unknown, but we can sample from this distibution by first sampling from the LM distribution p_θ, then rejecting any string that is not valid JSON. Here, the *un-normalized* distribution we are sampling from is

$$q_*(y) \propto \begin{cases} p_\theta(y) & y \text{ is valid JSON} \\ 0 & \text{Otherwise} \end{cases},$$

and we must use rejection sampling since the normalization term is unknown.

Best-of-N and rejection sampling. Above we introduced best-of-N as a deterministic algorithm (Definition 5). Another view is that calling best-of-N with a stochastic generator g is itself a stochastic generator,

$$y \sim \textsc{Bon}(p_\theta, g, N, v), \tag{36}$$

where Bon means generating N sequences $y^{(1)}, \ldots, y^{(N)} \sim g$, then selecting the sequence with the highest score v. This idea has been termed the *best-of-N policy* (Stiennon et al., 2020; Gao et al., 2022). Interestingly, Gao et al. (2022) find that the best-of-N policy may give similar reward maximization to reinforcement learning, though with a different pattern of divergence from the underlying language model. Their analysis uses an analytical expression for the KL divergence from (Stiennon et al., 2020), $\mathrm{D}_{KL}\left(\pi_{\textsc{Bon}} \| p_\theta\right) \approx \log N - (N-1)/N$. Beirami et al. (2024) show that this expression is an upper bound on the actual KL divergence and propose an estimator that empirically provides a tighter approximation.

Finally, $y \sim \textsc{Bon}$ can be understood as internally performing rejection sampling (Stiennon et al., 2020). We refer the reader to Liu et al. (2024c) for a more detailed discussion of this connection, as well as an improved algorithm that builds on the connection between rejection sampling and best-of-N.

Pseudo-rejection sampling. Several decoding methods employ various forms of *pseudo*-rejection sampling. One example of this is Li et al. (2024a), where the authors sample a set of k outputs from the LM, compute the "value" of each of these outputs, and then sample from the output set by interpreting the values as logits. As k tends toward infinity, this method approaches sampling from the value function with a regularization term that keeps the distribution close to the LM distribution. Pseudo-rejection sampling is often employed when the prerequisites for rejection sampling are not met, for instance when there is no known upper bound on the target distribution.

4.3 Step-level search algorithms

Next, we discuss meta-generation algorithms that implement classical search algorithms by calling generators. To introduce these, it is helpful to view generation as navigating a state space $s \in \mathcal{S}$ by taking actions $a \in \mathcal{A}$ using a generator, and receiving new states from an environment $\mathcal{E} : \mathcal{S} \times \mathcal{A} \to \mathcal{P}(\mathcal{S})$, yielding a trajectory $(s_0, a_1, s_1, \ldots, a_T, s_T)$. The start state s_0 contains the input to the generation algorithm, i.e. $x \in s_0$, while the terminal state contains the output of the generation algorithm, i.e. $y \in s_T$. Generation consists of running the resulting process until reaching a terminal state.

As a basic example, recall that greedy decoding is defined as:

$$\hat{y}_t = \arg\max_{y_t \in \mathcal{V}} p_\theta(y_t | \hat{y}_{<t}, x), \tag{37}$$

for $t = 1, \ldots, T$. The search perspective allows u interprets this as taking next-token actions $\hat{y}_t$ given states $(x, \hat{y}_{<t})$, a generator that selects the most probable next-token from p_θ, and an environment that appends a next-token to form a state $(x, \hat{y}_{<t} \circ \hat{y}_t)$. Since greedy decoding is an approximate MAP decoding algorithm, it aims to end in a state that maximizes $p_\theta(y|x)$. In other cases the environment is less trivial, such as those involving code execution and visual observations (Shinn et al., 2023; Zhou et al., 2023). Many algorithms can be recovered by varying the states, actions, environment, and/or generator.

In particular, reasoning tasks such as mathematical problem solving or theorem proving have served as a testbed for developing step-level search algorithms. In these tasks, the final output (a solution or a proof) naturally decomposes into 'steps', $y = (\mathbf{y}_1, \ldots, \mathbf{y}_T)$, where each $\mathbf{y}_t$ is itself a sequence of tokens. One can then consider a partial solution $\mathbf{y}_{<t}$ as the state s_t, and generating a next-step $\mathbf{y}_t$ as the action. The environment appends the next-step to the partial solution, $\mathbf{y}_{<t} \circ \mathbf{y}_t$. There is also a natural notion of success (i.e., a correct answer, a valid proof), leading to the idea of a *value function* $\hat{v}(s_t) \to [0, 1]$ that is used to predict whether a solution-so-far will eventually be correct (or in general, predict the expected reward of the state).

Several algorithms maintain a queue of states that contain partially generated outputs, and iteratively select states for exploration. Exploring a state involves expanding the state's partial output and scoring the expanded output with a value function $\hat{v}(s_t)$. The scores are then used to prune or prioritize states for the next iteration. Conceptually, step-level search is typically a tree search, consisting of states as nodes and actions plus environment transitions as edges. Although the algorithms below typically contain domain-agnostic ideas, we will ground the discussion below by discussing reasoning tasks as the running examples.

Warmup: token-level beam search. Traditional beam search (Graves, 2012; Sutskever et al., 2014) maintains a queue of prefixes $\{y^k_{<t}\}_{k=1}^K$ termed a *beam*, expands each prefix using each possible next-token, $\{y^k_{<t} \circ y_t \mid k \in \{1, \ldots, K\}, y_t \in \mathcal{V}\}$, scores each expanded prefix using $\log p_\theta(y^k_{<t} \circ y_t|x)$, and prunes the queue by keeping only the top-K scored expansions for the next iteration. In this case, the value function is $\hat{v}(y_{<t} \circ y_t) = \log p_\theta(y_{<t} \circ y_t|x)$, and it is used to prune states by selecting the top-K expanded prefixes. Traditional beam search operates on the token-level, using the specific strategy of expanding each possible next-token, which assumes access to primitive operations (e.g., next-token distributions).

Partial sequence expansion. We can consider higher-level algorithms that operate on the partial sequence (i.e., "step") level rather than the token level, and call an arbitrary generator to expand states, e.g., $\{\mathbf{y}^{(k)}_t\}_{k=1}^K \sim g(\cdot|s_t)$. For example, stepwise beam search (Welleck et al., 2022; Xie et al., 2023) performs a beam search over steps of mathematical problems or proofs. Tree-of-thoughts (Yao et al., 2023) considers a beam search over generated steps that include additional "thought" sequences. Potential benefits of partial sequence expansion over traditional beam search include efficiency due to executing the value function less often, and not requiring access to all of a model's next-token probabilities.

Alternate search strategy. Another axis of variation is the underlying search strategy. Beam search is a pruned breadth-first search, which has been used with contemporary LLMs in methods such as stepwise beam search (Welleck et al., 2022) and tree-of-thoughts (Yao et al., 2023). However, other search algorithms are available, such as depth-first search, also used in tree-of-thoughts, and best-first search, used in the context of formal theorem proving (Polu & Sutskever, 2020; Polu et al., 2023; Yang et al., 2023a; Welleck

Method	Search	State	Generation	Value $\hat{v}(s_t)$	Tasks
`gpt-f` proof search [165]	Best-first	Proof-so-far	Proof step	$\log p_\theta$	Formal proving
`gpt-f` +outcome [165]	Best-first	Proof-so-far	Proof step	$\hat{v}_\phi \approx \mathbb{E}(\texttt{success})$	Formal proving
Proofsize search [166]	Best-first	Proof-so-far	Proof step	$\hat{v}_\phi \approx \mathbb{E}(\texttt{length})$	Formal proving
Stepwise++ [209]	Beam	Proof-so-far	Proof step	$\log p_\theta$ + n-grams	Informal proving
Self-evaluation [217]	Beam	Steps-so-far	Reasoning step	$\log p_\theta$ + LLM	Multi-step correctness
Tree-of-Thought BFS [224]	Beam	Steps-so-far	Generation step	Prompted LLM	Multi-step generation
Tree-of-Thought DFS [224]	DFS	Steps-so-far	Generation step	Prompted LLM	Multi-step generation
Graph-of-Thought [15]	BFS/DFS	Steps-so-far	Generation step	Prompted LLM	Multi-step generation
HyperTree Proof Search [111]	MCTS	Proof-so-far	Proof step	$\hat{v}_\phi \approx \mathbb{E}(\texttt{success})$	Formal proving
AlphaLLM [196]	MCTS	Steps-so-far	Reasoning steps	$\hat{v}_\phi \approx \mathbb{E}(\texttt{correct})$	Multi-step correctness
ThoughtSculpt [34]	MCTS	Steps-so-far	Generation step	Prompted LLM	Multi-step generation

Table 4: Survey of step-level search methods.

& Saha, 2023). Formal theorem proving has a natural decomposition of outputs (i.e., a proof) into steps (termed "tactics"), which has historically made it a fruitful testbed for more advanced search algorithms. For example, HyperTree Proof Search (Lample et al., 2022) draws on monte-carlo tree search (MCTS), which prioritizes states according to a confidence bound and scores states by partially rolling out trajectories. Recently, similar ideas have translated to other LLM generation tasks. For instance, ThoughtSculpt (Chi et al., 2024) incorporates MCTS selection and rollouts for a variety of tasks.

Alternate value functions. Another axis of variation is the choice of value function. For instance, in traditional beam search, a value function can be manually designed to score candidates ($y_{<t} \circ y_t$) more highly if they contain desired n-grams (Lu et al., 2022), while others use learned heuristics $v_\phi(y_{\leq t})$ trained to predict a property of the full sequence (e.g., its style score or correctness) (Yang & Klein, 2021). In the context of large language models, a recent trend is to use a prompted language model to evaluate states (Yao et al., 2023). As mentioned previously, to achieve better results one can train a model that predicts whether the solution-so-far will eventually be correct, $v_\phi(y_{<t}) \to [0, 1]$ (or more generally, predict the expected reward from the current state), termed a process-based verifier (Uesato et al., 2022). Liu et al. (2024b) repurposes the value function from proximal policy optimization (Schulman et al., 2017) as a MCTS value function.

4.4 Refinement algorithms

A *refinement* algorithm consists of (1) an initial generator g_0, (2) an information source h, (3) a refiner g:

$$y^{(0)} \sim g_0(y|x), \tag{38}$$

$$z^{(t)} \sim h(z|x, y^{(<t)}, z^{(<t)}), \tag{39}$$

$$y^{(t)} \sim g(y|x, y^{(<t)}, z^{(\leq t)}). \tag{40}$$

Intuitively, the refiner generates a "revised" output $y^{(t)}$ given previous versions $y^{(<t)}$ and extra information $z^{(\leq t)}$, such as feedback or environment observations. The algorithm alternates between receiving information, $z \sim h$, and refining, $y \sim g$, until a stopping condition is met. Refinement algorithms vary based on choice of initial generator, the refiner, the content and source of extra information z, and the stopping condition.

Learned refiners. Self-correction, introduced by Welleck et al. (2023), provides a recipe for training a refiner model $p_\theta(y^{(t)}|x, y^{(t-1)}, z^{(t)})$ which iteratively refines an output to improve the score from a reward function $r(x, y)$ using $(z^{(t)}, y^{(t-1)}, y^{(t)})$ examples collected from model trajectories. Here z is either 0/1 (apply the refiner or do not apply the refiner) or a feedback string. z is assumed to be given to the system at generation time, which is a limitation for some tasks (e.g., we often do not know whether a mathematical solution should be revised). GLoRe (Havrilla et al., 2024) relaxes this limitation by training a verifier to determine whether to apply the refiner, and to localize per-step errors.

Prompted refiners. A second option is to parameterize the refiner using a prompted language model,

$$y^{(t)} \sim g_\theta \left(y | P_{\text{refine}}(x, y^{(<t)}, z^{(\leq t)}) \right), \tag{41}$$

where g_θ is a generation algorithm that involves prompting a model p_θ with a prompt $P_{\text{refine}}(x, y^{(<t)}, z^{(\leq t)})$, as introduced in Self-Refine (Madaan et al., 2023) and Reflexion (Shinn et al., 2023). This allows the initial generator and the refiner to share a single language model that is not necessarily tuned for a specific task.

Prompted feedback. It is common for the information $z \sim h$ to include "feedback" on a preceding version y, with the feedback being a sequence of tokens generated with a prompted language model:

$$z^{(t)} \sim h_\theta \left(z | P_{\text{feedback}}(x, y^{(<t)}, z^{(<t)}) \right). \tag{42}$$

This feedback is often also termed "critique" (Matiana et al., 2021; Castricato et al., 2022; Bai et al., 2022; Saunders et al., 2022). Self-Refine (Madaan et al., 2023) shares θ across the feedback provider, refiner, and initial generator, yielding a refinement algorithm given only a model p_θ and 3 prompts. Similarly, Reflexion (Shinn et al., 2023) uses a prompted feedback provider. In these cases, the feedback is termed *self-feedback* or *self-reflection*.

Environment feedback. As we discussed previously, the search perspective treats generation as a trajectory $(s_0, a_1, s_1, \ldots, a_T, s_T)$, with state transitions determined by an environment $\mathcal{E} : \mathcal{S} \times \mathcal{A} \to \mathcal{P}(\mathcal{S})$. In some cases the environment transitions are nontrivial, such as executing generated code or clicking a link in a webpage. We can view the resulting observations (e.g., code execution results or an image of a webpage) as extra information $\tilde{z}^{(t)} \in s^{(t)}$ that is contained in the state. This information can be passed to the feedback provider to generate feedback $\bar{z}$, and the refiner (e.g., via prompts):

$$\bar{z}^{(t)} \sim h_\theta \left(\bar{z} | P_{\text{feedback}}(x, y^{(<t)}, \tilde{z}^{(\leq t)}, \bar{z}^{(<t)}) \right), \tag{43}$$

$$y^{(t)} \sim g_\theta \left(y | P_{\text{refine}}(x, y^{(<t)}, \tilde{z}^{(\leq t)}, \bar{z}^{(\leq t)}) \right), \tag{44}$$

meaning that each iteration refines based on new environment information (e.g., code execution results).

Reflexion (Shinn et al., 2023) adopts this perspective of generation as a trajectory involving an environment, with the refiner akin to an "actor" or "policy". Subsequently the idea has been adapted to digital agents (Kim et al., 2023; Pan et al., 2024), code (Chen et al., 2024b; Shi et al., 2024), and other environments (Pan et al., 2023).

Does refinement work? Notice that a refinement algorithm is a 3-tuple (g_0, h, g). Intuitively, if the information source h adds new information beyond that contained in the initial generator $g_0(\cdot|p_\theta)$ to the refinement algorithm, it is plausible that a refinement algorithm can outperform the initial generator alone.

For instance, if $z \sim h$ contains the results of code execution or an image of webpage after it is clicked, the refiner is likely to receive new information. Similarly, if $z \sim h$ represents feedback, and the feedback comes from a source outside of the model p_θ (e.g., a human, a model with additional parameters, supervision, or a different objective), the feedback function may be expected to add new information beyond that in $g_0(\cdot|p_\theta)$. For challenging grounded tasks, this external feedback can be essential; for instance, Reflexion (Shinn et al., 2023) finds that without execution, refinement yields little to no gain on code generation.

Moreover, the quality of the feedback $z \sim h$ is important. Self-Refine (Madaan et al., 2023) finds that generic feedback hurts refinement quality, and Olausson et al. (2024) finds that higher quality natural language feedback (i.e. from humans) helps models to perform better on code self-repair. However, for complex tasks without environment feedback, it can be difficult for models to generate meaningful and accurate feedback. This stems from the fact that models often struggle with evaluating the correctness of (their own) outputs (Huang et al., 2024) and ties into a fundamental challenge of refinement methods: *when should we refine?*

Clearly, if at some iteration the model yields a correct output, there is no need to continue refining. Additional rounds of refinement may spur a model to modify correct parts of its prior output, hurting performance. For

tasks with access to a ground-truth environment evaluator (as in Shinn et al. (2023)), it is easy to know when to stop; for instance, in code self-repair we can simply stop once the code passes all test cases. However, in tasks where this is not possible, refinement methods resort either setting a fixed number of iterations (Welleck et al., 2023), leveraging the generator itself (Madaan et al., 2023), or using learned external models (Havrilla et al., 2024). Presently, none of these approaches are ideal. Tyen et al. (2024) and Huang et al. (2024) find that self-refinement often fails to improve reasoning due to models' inability to gauge the correctness of their outputs and localize errors. Havrilla et al. (2024) observe that reward models trained to predict correctness tend to have high false positive rates, triggering spurious refinements.

At present, refinement usually works best for tasks that either have rich environment feedback or can be reliably evaluated by current language models. As language models improve as verifiers, the range of tasks for which refinement is effective will likely grow. However, this may be paired with improvements in the abilities of the initial generator g_0, potentially even to the extent that refinement is no longer necessary.

5 Incorporating external information

Next, we consider what kinds of information a generation or meta-generation algorithm incorporates outside of the language model, such as other models or tools. Algorithms use external information by calling operations beyond primitive operations from p_θ (e.g., those from another model), or through assumptions on the inputs or outputs. We comment on common patterns related to incorporating external information.

5.1 Multiple models

A variety of generation algorithms incorporate multiple models. More formally, recall that in (§2.2) we defined a generation algorithm as a function that maps an input x, model p_θ, and other inputs ϕ to a distribution $q(y|x; p_\theta, \phi)$. A generator *uses multiple models* if ϕ contains other models (e.g., an additional language model), and operations from the model are used in the algorithm. In this sense, the external models can add new information beyond that contained in p_θ to the generator.

Small and large language models. A notable pattern is using a small language model to either adjust a model's distribution or to speed up generation. Lu et al. (2023) train a small model p_β with reinforcement learning such that it adjusts the next-token probabilities of a larger model p_θ to maximize a reward function. The models are combined into a token-level "product of experts" (Liu et al., 2021a),

$$p'(y_t|y_{<t}, x) = \frac{1}{Z} p_\theta(y_t|y_{<t}, x) p_\beta(y_t|y_{<t}, x)^\alpha, \tag{45}$$

where p_β is a separate language model, $\alpha \in \mathbb{R}$, and $Z \in \mathbb{R}$ is a normalization constant. Liu et al. (2024a) adopt a similar idea but with supervised finetuning of p_β. In order to amplify the improvement of a large, strong model over a small, weak one, contrastive decoding (Li et al., 2023a) defines a scoring function for beam search that returns the difference between the likelihood under the model p_θ with that of a smaller language model p'_θ,

$$s(y_{<t} \circ y_t) = \log p_\theta(y_t|y_{<t}) - \log p'_\theta(y_t|y_{<t}), \tag{46}$$

along with a truncation criterion that sets the score to zero for some tokens. Intuitively, the smaller model often has larger model errors on unfavorable tokens (e.g., assigning more probability to tokens leading to repetition or incoherence compared to p_θ). Assuming there is a nontrivial difference in probability assigned to these tokens, the score will reduce their prevalence in generated texts.

Finally, *speculative decoding* (Leviathan et al., 2022) is motivated by speeding up generation, which we will discuss further in (§7). It uses a small *draft* model to propose generations that are verified or rejected in parallel by the larger model p_θ, hence speeding up generation when the rejection rate is not too high.

Scalar feedback models. A common pattern is learning a *verifier model* $v_\theta(y) \to [0, 1]$ that predicts the probability that a generation is correct (Cobbe et al., 2021). The verifier can be used to select outputs

in best-of-N, or for weighted majority voting (Li et al., 2023b). This pattern is particularly suitable for mathematical problem solving and code generation (Ni et al., 2023), which have well-defined notions of correctness. Several works have iterated on the verifier model $v_\theta(y)$'s design and learning procedure. Uesato et al. (2022) show that a verifier trained to predict the correctness of each *step* in an output (termed a *process-based* verifier) can outperform a verifier trained to predict the correctness of a full solution (termed an *outcome-based* verifier), and Lightman et al. (2024) obtain new human annotations for a process-based verifier. Math Shepherd (Wang et al., 2023a) propose a method for obtaining supervision from generations.

An underlying idea is that a generation algorithm that incorporates the verifier may have capabilities beyond those of p_θ. This may be due to additional supervision, or factors that stem from the intuitive idea that *evaluation is often easier than generation.* For example, Sun et al. (2024a) show that weighted majority voting with a verifier can improve the generator's ability to generalize to harder problems.

More generally, a learned verifier is a special case of learning a scalar reward model $r_\phi(x, y)$ that can be used to select or score outputs. For instance, in (§4.2.3) we discussed using a reward model of human preference ratings to select outputs in best-of-N (Stiennon et al., 2020; Ouyang et al., 2022; Touvron et al., 2023). As we discussed previously (§4.2.3), we can view this as shifting the generation distribution.

Information conveyed in prompts. Rather than using a separate model in a multi-model system, it is now common to parameterize different models by providing different prompts. For instance, we can obtain a feedback model by prompting a model to provide feedback. It is important to note that the prompts can add new information to the generation algorithm.

5.2 External environment information

More generally, generation algorithms can incorporate information from an external environment.

Calling an external tool. Certain functionality such as reliably performing a calculation or a web search may either be outside of a model's capabilities or inefficient to perform with the language model. A natural alternative is to issue a call to an external routine that performs the functionality at generation time.

One way to do this is through special tokens that denote a call to the routine, followed by replacing the prefix with the result. For instance, suppose the preceding tokens $y_{<t}$ include `[CALC]4+4[/CALC]`. Then at step t of a token-level decoding algorithm, a calculator would be called on the query `4+4`, and in subsequent steps, the prefix $y_{\leq t}$ would contain the result 8, along with possible reformatting (e.g., removing `[CALC]`).

A second common use of an external routine is as a verifier following the generation of a full sequence. For instance, in language-model based theorem proving the proof assistant is used to verify or reject generated proofs, while in code generation it is common to execute test cases. More generally, the notion of "tool use" (i.e., calling external programs) is now widespread, and has been incorporated into libraries such as LangChain (Chase, 2022) and products. Refer to Wang et al. (2024b) for further discussion.

Receiving observations from an environment. The search perspective framed generation as a sequential decision making process that involves observations from an environment (§4.3). A notable application area is code generation, which has natural environment information (e.g., interpreters, compilers). For instance, Lever (Ni et al., 2023) feeds execution results into a reward model used for best-of-N, while Self-Debugging (Chen et al., 2024b) incorporates error messages into refinement. A recent line of work tailors generation algorithms to language-conditioned digital agents–models that operate on diverse observation spaces $\mathcal{X}$ such as images of web pages, and output sequences y representing actions–including variants of refinement (Shinn et al., 2023) combined with learned evaluators (Pan et al., 2024).

6 Token cost and performance analysis

A natural question is the cost of executing a given meta-generator, and its relationship with performance. There are several ways to measure cost, including the number of tokens generated, the overall compute used during generation, or the runtime. In some cases, we would like to design an algorithm that improves as

Method	Input	Output	External	Cost Params
Ancestral Sampling	T_{in}	T	–	–
Reranking (general)	$T_{\text{in}} * N$	$T * N$	$N * C_s$	N, C_s
Best-of-N (log-p)	$T_{\text{in}} * N$	$T * N$	–	N
Best-of-N (LLM sequence scorer)	$T_{\text{in}} * N$	$T * N$	$N * (T_{\text{in}} + T + 1)$	N
Transformation (general)	$T_{\text{in}} * N$	$T * N$	C_t	N, C_t
Self-consistency	$T_{\text{in}} * N$	$T * N$	–	N
Weighted SC (seq. scorer)	$T_{\text{in}} * N$	$T * N$	$N * C_s$	N, C_s
Step-level beam (log-p) [209]	$T_{\text{in}} * N_b * N_e * S$	$T_s * N_b * N_e * S$	–	N_b, N_e, S
Step-level beam (seq. scorer) [224]	$T_{\text{in}} * N_b * N_e * S$	$T_s * N_b * N_e * S$	$N_b * N_e * S * C_s$	N_b, N_e, S, C_s
Step-level DFS (seq. scorer) [224]	$T_{\text{in}} * N_e * S$	$T_s * N_e * S$	$N_e * S * C_s$	N_e, S, C_s
Refinement (general)	$T_{\text{in}} * (1 + N_r)$	$T * (1 + N_r)$	$N_r * C_z$	N_r, C_z
Refinement (self-feedback) [139]	$T_{\text{in}} + (2T_{\text{in}} + T) * N_r$	$T + 2T * N_r$	–	N_r

Table 5: Token budget for representative algorithms from each meta-generation class. **Reranking.** T_{in} and T are the number of input tokens and output tokens for each call to the generator, respectively. For simplicity, we assume the number of input and output tokens is constant across calls to the generator. C_s refers to the number of tokens required to call a scoring model (e.g., a prompted LLM) on an input and output sequence. LLM scorer refers to prompting a LLM with an input and output, and generating a scalar score (assumed to be 1 token). **Transformation.** C_t refers to the number of tokens required to call a transformation function (e.g., a prompted LLM) on N sequences. **Step-level search.** T_s is the number of output tokens in a step, with S the maximum number of steps, such that $T_s * S \geq T$. N_b is the number of candidates to keep after pruning (e.g., "beam size"), and N_e is the number of expansions per iteration. We assume the cost of the scorer is equal to the cost of scoring a full sequence (C_s). **Refinement.** N_r is the number of refinement iterations. C_z refers to the number of tokens required to obtain external information during a refinement iteration.

we add more cost, such as improving problem solving ability by generating more tokens. In other cases, we would like to minimize the cost at a fixed level of performance.

6.1 Token budget

Meta-generators consist of calling generators, which leads to costs associated with generating tokens. For instance, common APIs charge by the number of tokens in the input prompt and the number of output tokens. In general, meta-generators incur token costs from input tokens, output tokens, and external information.

For instance, a reranker that generates N sequences incurs a cost of $T_{\text{in}} * N$ input tokens, $T * N$ output tokens, and $N * C_s$ tokens to run the scoring model, where C_s is the token cost of calling the scoring model on one sequence. When the scoring model is implemented by prompting an LLM and generating a scalar quality score (assumed to cost 1 token), the external information cost is $N * (T_{\text{in}} + T + 1)$. Table 5 shows the token budget for representative algorithms from each meta-generation class.

Step-level vs. sequence-level search. Consider solving a mathematical problem by generating a solution that consists of multiple steps. Two strategies for doing so are (1) generating one step at a time using a step-level search algorithm, or (2) generating full solutions in a transformation or re-ranking algorithm. In this case, we can assume that $T = T_s * S$, i.e., the total number of tokens in a solution (T) equals the number of tokens in a step (T_s) times the number of steps (S). We can then use Table 5 to reason about when step-level search can cost fewer tokens than sequence-level search.

From Table 5, we see that step-level methods incur a cost from generating output tokens that depends on the pruning parameter N_b, the number of expansions per iteration N_e, and the number of iterations S. Assuming that $T_s * S = T$, step-level search has fewer output tokens than sequence-level search when $N_b * N_e < N$. For example, under these assumptions step-level beam with a beam size of 16 and 64 expansions per iteration has the same number of output tokens as best-of-1024, while lowering the expansions per iteration to 32 would be half the output token cost compared to best-of-1024.

On the other hand, Table 5 shows that step-level search calls the scoring model more often than sequence-level search methods. For instance, when $N_b * N_e = N$, step-level beam search calls the scoring model $N * S$ times compared to N times with reranking. Therefore, one must also account for potential token costs associated with external information (e.g., sequence scores) when comparing meta-generator token budgets.

Refinement vs. sequence-level search. Similarly, we can compare the token budget for refinement versus sequence-level search. As seen in Table 5, general refinement algorithms have a lower output cost when $N_r < N$, i.e., the number of refinements is less than the N in best-of-N. In practice this is often the case, e.g. Madaan et al. (2023) use $N_r = 3$ in many experiments, while N typically ranges from 8 to 1024 in the literature. However, we need to factor in the cost of external information. For instance, when generating self-feedback as in Madaan et al. (2023), the output cost becomes $T + 2T * N_r$, meaning that 3 refinements costs $7T$ output tokens, which is still cheaper than best-of-8.

6.2 Increasing the token budget to improve performance

In various reasoning-related tasks such as mathematical problem solving, it has been widely observed that generation algorithms which generate multiple sequences and choose among the sequences (e.g., best-of-N, majority voting) can outperform generation algorithms that generate a single sequence (e.g., greedy decoding) (Cobbe et al., 2021; Wang et al., 2023b; Azerbayev et al., 2024; Lightman et al., 2024; Wang et al., 2023a; Sun et al., 2024a).

Figure 5 shows a plot from Sun et al. (2024a) that compares the relationship between the generation budget (in units of sequences) with three sequence-level approaches on the MATH500 benchmark (Lightman et al., 2024). The plot shows that these algorithms can improve monotonically by increasing the generation budget. Moreover, each algorithm has a different improvement as a function of the generation budget. For instance, at a budget of 1024 sequences, weighted voting is preferred to majority voting or best-of-N in terms of task performance. Recently, Chen et al. (2024a) found that some models can have a non-monotonic relationship between generation budget and voting performance.

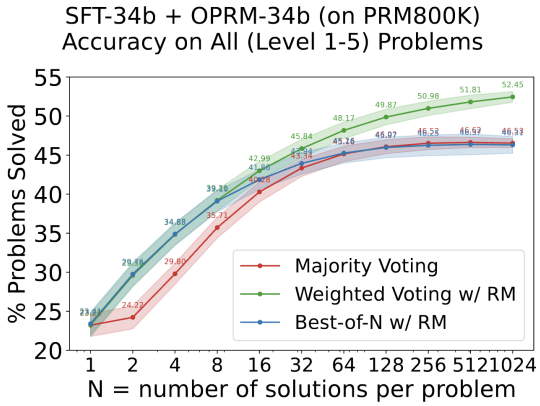


Figure 5: Plot from Sun et al. (2024a). Scaling behavior of three meta-generators in the number of samples N on mathematical problem solving (MATH500).

The idea of increasing the generation budget to improve performance has appeared in many applications. For instance, AlphaCode (Li et al., 2022) generates up to a million sampled programs that are then filtered using heuristics and execution results. In theorem proving, Draft-Sketch-Prove (Jiang et al., 2023) leverage the proof checker at generation time by generating and checking many formal proof candidates, resulting in a monotonically increasing percentage of proven theorems as a function of the generation budget.

More formally, let $q_*(y|x) \propto 1$ if y is correct, and 0 otherwise, where correctness may mean a correct solution to a mathematical problem, a valid proof, a program that passes test cases, etc. Then the goal of generation is $y_* = \arg\max_{y \in \mathcal{Y}} q_*(y|x)$. Since the space of solutions $\mathcal{Y}$ is too large, a meta-generator can approximate it by calling a generator multiple times,

$$y_* = \arg\max_{y \in \mathcal{Y}} q_*(y|x) \tag{47}$$

$$\approx \arg\max_{y^n \in y^1, \ldots, y^N} q_*(y^n|x), \tag{48}$$

where $y^n \sim q(\cdot|x, p_\theta)$. It is clear that performance should improve as N increases, so long as the generator q assigns probability mass to correct solutions. However, in practice we do not have access to q_* at test

time, so different meta-generators approximate (48), e.g. with a learned verifier $v_\phi(x, y)$, or with a voting algorithm. The plot above shows that different approximations have different levels of effectiveness.

6.3 Minimizing the token budget

A complementary direction is minimizing the generation budget to achieve a given level of performance. One direction is to route generations to progressively more costly models. For instance, FrugalGPT (Chen et al., 2023b) first generates with a cheap model, then uses a learned scoring function to determine whether to generate again with a more expensive model, leading to significant cost reductions over calling GPT-4 in their experimental setting. Another direction is leveraging properties of specific meta-generation algorithms to reduce the number of calls. Aggarwal et al. (2023) propose to stop sampling in majority voting upon converging to a majority. Kapoor et al. (2024b;a) argue that performance comparisons of meta-generation algorithms must be performed with respect to token budget and monetary cost, and that simple meta-generation baselines can provide a pareto-optimal cost-performance tradeoff compared to complex algorithms.

Wu et al. (2024) formalize these tradeoffs in terms of *compute-optimal inference*: the problem of choosing a model size, number of generated tokens, and meta-generation algorithm that minimizes error subject to a compute budget. Specifically, let the error rate $E(M, T; g)$ be a function of the number of model parameters M, the number of generated tokens T, and the generator g. The goal is to minimize E under the constraint $c(M, T, g) = C$, where $c(M, T, g)$ is the compute used during inference, measured in floating-point operations:

$$(M_{\text{opt}}(C), T_{\text{opt}}(C), g_{\text{opt}}) = \underset{M,T,g \text{ s.t. } c(M,T;g)=C}{\arg\min} E(M, T, g), \tag{49}$$

where $M_{\text{opt}}(C)$, $T_{\text{opt}}(C)$, g_{opt} denotes the choice of model size, generated tokens, and meta-generator that achieves the lowest error with compute budget C. Wu et al. (2024) study tradeoffs between best-of-N, majority voting, and tree search variants, finding that sampling more tokens from a smaller model often had better cost-performance tradeoffs compared to using a larger model at a given compute budget. Moreover, Monte-Carlo tree search often had worse cost-performance tradeoffs than the other meta-generators. Snell et al. (2024) study similar cost-performance trade-offs between best-of-N, step-level search, and refinement.

6.4 Dependence on the underlying generator(s)

The defining property of meta-generators is that they rely on calling other generation algorithms. Hence a second natural question is to what degree their performance depends on the underlying generation algorithms.

Sampling parameters. Chen et al. (2021) found that the optimal temperature in best-of-N was dependent on N for code generation with the Codex model, with higher temperatures returning better performance for higher N. Many prior studies use temperatures or sampling parameters that are either unexplained or ad-hoc. For instance, Minerva (Lewkowycz et al., 2022) uses majority voting with temperature 0.6 and nucleus sampling $p = 0.95$. These settings have propagated into subsequent studies (Azerbayev et al., 2024).

For some classes of meta-generators such as minimum Bayes risk (§4.2.2), the effect of sampling parameters is relatively well-studied. For example, Freitag et al. (2023) investigate the impact of the underlying sampling strategy in MBR, finding variation across strategies, with epsilon sampling performing best for translation.

7 Speeding up generation

In the preceding sections, we introduced generation algorithms (e.g., ancestral sampling, beam search) and meta-generation algorithms (i.e., programs involving multiple generation calls), and discussed one aspect of efficient generation: making generation cost-effective in terms of the token budget. Next we turn to another aspect of efficiency: the speed of (meta-)generation. Speed is an inherent concern of almost any practical application of generation algorithms: users typically want outputs quickly.

Meta-generators in particular raise demands for fast generation, since they often involve generating many sequences and coordinating multiple components. For example, the meta-generators shown in Figure 5

require generating and scoring 1024 sequences. There are at least two high-level strategies one can take to speed up generation: (1) speeding up the generation of each individual sequence, and (2) leveraging structure that comes from multiple generator calls, such as shared partial outputs or the structure of the overall meta-generation program. We will consider both of these below.

Before we start, it is worth noting two points. First, the notion of "speeding up" itself needs to be made more precise and measurable. To that end we provide background on the notions of latency, throughput, and the idea that speed is often dependent on the hardware environment in which a meta-generator is run.

Second, the topics in this section are part of a rich, rapidly evolving research field that ranges from machine learning systems to programming language design. It goes without saying that our survey here merely scratches the surface. We focus our discussion on introducing key ideas, and on examining the *interaction* between the design space of (meta-)generation algorithms and generation speed.

7.1 Background

Goals of speeding up generation. Speeding up generation requires balancing between three high-level metrics: (1) **latency**, the time it takes to generate a single output; (2) **throughput**, the rate at which outputs can be produced; and (3) **quality**, measures of model quality such as loss or downstream task metrics. For instance, one might change the generation algorithm in a way that speeds up a single generation (improving latency), but removes the ability to generate outputs in parallel (degrading throughput). Other cases such as reducing the precision of model weights may improve latency and throughput, but degrade the model's task performance. Ideally, we would like to reduce latency, increase throughput, and maintain quality.

Hardware-aware optimization. The **underlying hardware** is a key consideration for speeding up generation. LLMs are typically run on accelerators such as GPUs or TPUs.

In the case of GPUs, performance is largely dictated by **compute** and **memory bandwidth**. Compute is typically measured via the number of floating-point operations (FLOP) used in a given operation, while memory bandwidth refers to the rate at which data can be transferred to and from memory. For example, the operation:

$$A = BC, \tag{50}$$

reads the matrices B, C from memory, computes BC on-chip, and writes the result out to memory. Similarly,

$$Y = \mathrm{ReLU}(X) \tag{51}$$

must read X from memory, compute $\mathrm{ReLU}(X)$ on-chip, and write the result out to memory. However, these two operations have very different **arithmetic intensities**, defined as the ratio of compute (in FLOP) to unit of memory read or written. This results in (50), for large enough B, C, being **compute-bound** (bottlenecked by the rate at which operations can be performed) while (51) for large X is **memory-bound** (bottlenecked by the speed of reading inputs and writing outputs to memory).

Thus, *reducing the quantity of operations performed (in FLOP) for a given step may not always proportionately transfer to an equivalent real-world speedup or cost reduction.* This is exacerbated by the properties of recent accelerators–GPUs and TPUs are heavily specialized for matrix multiplication and other high-arithmetic intensity, heavily parallelizable workloads (NVIDIA, 2017; 2020). For example, the H100 can perform up to 989.4 TFLOP/s in BF16 within a dense matrix multiplication using Tensor Cores, but only 133.8 TFLOP/s of BF16 arithmetic (NVIDIA, 2022). This specialization–and the fact that "naive" attempts to optimize performance oblivious to which operations may be the key bottlenecks may not achieve the anticipated gains–implies that **hardware-aware optimization** is a key viewpoint to take when seeking speedy generation. Algorithmic and architectural co-design with the hardware (Dao et al., 2022; Dao, 2023; Anthony et al., 2024) has yielded some of the most significant speed gains in recent years, in contrast to approaches seeking to minimize theoretical complexity that are disconnected from the hardware level. On the flip side, however, Hooker (2020) discuss the notion of the *hardware lottery*–the idea that co-design of novel techniques creates adverse selection effects, where research ideas "off the beaten path" are dispreferred because they interact less well with existing hardware.

Type	Selected Examples	Strategy
Architectural	MQA [180], GQA [4], MLA [42], $\cdots$ RWKV [163], Mamba [74], $\cdots$	Efficient attention Transformer alternative
Compression	GPTQ [57], AWQ [127], SqueezeLLM [101] , $\cdots$ LLM.int8() [44], Smoothquant [214], QuaRot [8], $\cdots$ FlexGen [181], KVQuant [84], W4A8KV4 [128], $\cdots$	Quantize weights Quantize activations Quantize KV Cache
Hardware-aware impl.	Rabe & Staats (2022), FlashAttention [39; 38], $\cdots$ Triton [197], Torch compile [169, Cutlass [195], $\cdots$	Efficient attention Libraries/tooling
Parallelize over time	Speculative decoding [115; 25], SpecInfer [146], $\cdots$	Draft-then-verify

Table 6: Outline of classes of techniques for speeding up a single generation call. Refer to the main text for additional examples.

7.2 Speeding up the generator

Generation algorithms with autoregressive language models depend on computing next-token distributions. Given an input sequence $(y_{<t}, x)$, typical implementations start with an initial "prefill" step that computes

$$p_\theta(\cdot|y_{<t}, x) = \text{softmax}(s_\theta(\cdot|y_{<t}, x)). \tag{52}$$

Performing this step returns *two outputs*: $p_\theta(\cdot|y_{<t}, x)$, the probability distribution over immediate next tokens following $(y_{<t}, x)$ that we have discussed previously, and a "state" $S_{y_{<t},x}$ created as a byproduct of processing $y_{<t}, x$. For a Transformer (Vaswani et al., 2017) $S_{y_{<t},x}$ is produced by retaining all keys and values from timesteps up to y_{t-1} within the attention for each layer.[2] This is termed the "Key-Value (KV) Cache" produced by attention at each layer. At this step, we may sample a next-token y_t from $p_\theta(\cdot|y_{<t}, x)$.

Subsequently, to generate additional new tokens we may perform any number of "decoding" steps, where each step selects a token from a next-token distribution. For example, the $t+1$'th step selects a token using:

$$p_\theta(\cdot|y_{<t+1}, x, S_{y_{\leq t},x}) = \text{softmax}(s_\theta(\cdot|y_{<t+1}, x, S_{y_{\leq t},x})). \tag{53}$$

Here, we feed the state $S_{y_{\leq t},x}$ into the model, representing the already-processed sequence. Each decoding step saves on computations that are cached in the state, such as the attention keys and values from the preceding steps. After selecting the $t+1$'th token, the state is updated to $S_{y_{\leq t+1},x}$. These decoding steps may be repeated until we have finished generating a sequence.

One can accelerate a single generation from an LM by speeding up the time taken per step, such as through architectural modifications, model compression, hardware-aware implementation decisions, or by clever parallelization during autoregressive generation. We discuss each of these in the following paragraphs.

Architectural modifications. One strategy is to modify the model architecture. For example, multi-query (Shazeer, 2019) and grouped-query (Ainslie et al., 2023a) attention propose the use of fewer key and value heads in transformers' attention, leading to reduced KV Cache sizes. Smaller KV Cache sizes can lower memory bandwidth demands, or provide the ability to process larger batches concurrently at a time by enabling more requests to be stored in GPU memory. Similarly, DeepSeek-AI (2024) propose multi-headed latent attention, attempting to retain the reduced KV Cache of GQA while improving model quality. The $O(t^2)$ complexity of attention ($O(t)$ for each decoding step) can slow generation down as sequences become longer, so another option is to forego the transformer architecture or its attention layer altogether. For example, traditional recurrent language models (Elman, 1990; Mikolov et al., 2010) compute a next-token distribution by maintaining a hidden state, leading to a $O(t)$ time and space complexity ($O(1)$ per step). Recent architectures draw on ideas from recurrent language models (Hutchins et al., 2022; Peng et al., 2023; De et al., 2024; Yang et al., 2024) and/or state-space models (Gu & Dao, 2023; Lieber et al., 2024; Gu et al., 2022; Smith et al., 2023; Poli et al., 2023; Fu et al., 2023; Arora et al., 2024) to achieve sub-quadratic time and space complexities. Although models can occasionally be adapted post-hoc from a transformer

[2]The core attention operation is softmax $\left(QK^T/\sqrt{d}\right)V$, where $Q, K, V \in \mathbb{R}^{t\times d}$ are referred to as queries, keys, and values, respectively, t is the time dimension, and d is the hidden dimension.

architecture to one of these more efficient variants (Zhang et al., 2024; Ainslie et al., 2023a), this adaptation can degrade model quality or require substantial compute.

Model compression. Adjacent to architectural modifications, one can *compress* a model into a more efficient form after the fact. *Distillation* can transfer knowledge from a more capable teacher model into a smaller one (Hinton et al., 2015; Sanh et al., 2020), or models can be *quantized* to reduce the floating-point precision of the model's weights which reduces the memory footprint of the model and in turn speeds up generation in memory bandwidth-constrained settings (Dettmers et al. (2022); Frantar et al. (2023); Dettmers et al. (2023); PyTorch (2023), *inter alia*). Model activations can also be quantized (Ashkboos et al., 2024; Xiao et al., 2024a; Lin et al., 2024b). Approaches to sparsify or prune model weights (Frantar & Alistarh (2023), *inter alia*) can also be used. Such compression approaches frequently, but not always, degrade performance and require training to perform or to recover performance on a limited distribution.

Hardware-aware implementation. A number of optimizations may be performed without modifying the model architecture or *what* operations must be performed, simply *how* they are performed.

For instance, Flash Attention (Dao et al., 2022; Dao, 2023) famously overcomes the $O(t^2)$ space complexity of self-attention by adapting the algorithm proposed by Rabe & Staats (2022) for computing self-attention based on online softmax (Milakov & Gimelshein, 2018; Jang et al., 2019) and blockwise computation, crucially without changing the output of the attention mechanism, simply its mapping to hardware. Similarly, Flash Decoding (Dao et al., 2023) accelerates the attention operation during decoding by adding extra parallelism over the sequence dimension, allowing the GPU to be fully saturated even for small query and batch sizes, but only changing the order and mapping of operations on-device, not the end result (up to numeric precision).

Numerous software tools (Tillet et al. (2019); PyTorch (2023); Thakkar et al. (2023), *inter alia*) can enable fast decoding and efficient low-level implementation in practice. Overall, while architectural modifications to the model itself can increase the *ceiling* on generation speed, effective *implementation* is key for achieving performance anywhere near this ceiling on current accelerators.

Parallelization across time. Rather than speeding up the core next-token operation, the *draft-then-verify* (also called "speculative sampling" or "speculative decoding") pattern leverages clever parallelization during autoregressive generation. Draft-then-verify consists of generating proposed next-tokens with a fast method (e.g., a smaller model), computing next-token distributions given the proposed tokens *in parallel*, and either keeping or rejecting the proposed tokens.

For example, previously we briefly referred to speculative sampling (Leviathan et al., 2022; Chen et al., 2023a). This method assumes a language model $p_\theta(y_t|y_{<t})$ and an efficient *draft* model $q(y_t|y_{<t})$. At a given step t, it generates a continuation $y_t, y_{t+1}, \ldots, y_{t+k}$ using q, then computes the next token distributions $p_\theta(y_t|y_{<t}), \ldots, p_\theta(y_{t+k}|y_{<t+k})$ in parallel. Finally, it processes each proposed token, keeping it if $q(y_{t'}|y_{<t'}) \leq p_\theta(y_{t'}|y_{<t'})$, and rejecting it when $q(y_{t'}|y_{<t'}) > p_\theta(y_{t'}|y_{<t'})$ with probability $p_\theta(y_{t'}|y_{<t'})/q(y_{t'}|y_{<t'})$ or if a preceding token was rejected. Intuitively, as long as (i) generating with $q()$ is much faster than computing the distributions with p_θ in sequence, and (ii) the rejection rate is not too high, then speculative sampling will speed up generation without affecting the original model's output distribution or quality.

Several methods iterate on ideas underlying speculative sampling, including guessing and verifying a tree of proposed tokens (Miao et al., 2023b; Sun et al., 2024b; Chen et al., 2024c), using alternative proposal models q (Miao et al., 2023b; Cai et al., 2024), using prompt n-grams as proposals (Yang et al., 2023b), or generating in parallel and reusing the generated n-grams as proposals (Fu et al., 2024).

Interestingly, many speculative sampling approaches which require a separate draft model $q()$ require *more* total FLOP in order to generate a given sequence (Chen et al., 2023a; Leviathan et al., 2023; Fu et al., 2024). However, because the decoding step is typically memory-bound, the increased parallelism afforded is more than sufficient to provide substantial generation speedups.

Type	Selected Examples
State reuse	PagedAttention memory sharing [110], RadixAttention [234]
State compression	Gisting [149], KV Cache compression [128; 233]
Improved batching	Continuous batching [225], Disaggregated prefill [235]
Program-level optimization	GPT-4 graph rewriting [234], DSPy [98]

Table 7: Outline of techniques for speeding up meta-generation algorithms, requiring many calls to an underlying generator with often-predictable traffic patterns. Refer to the main text for more examples.

7.3 Speeding up meta-generation algorithms

While in §7.2 we note approaches to speeding up a *single* autoregressive generation call, the space of possible optimizations is larger when considering usage patterns such as those found in meta-generation algorithms, where multiple or many calls are made to the same model over time, often in a predictable way.

7.3.1 Leveraging shared prefixes.

The repeated model generation calls that occur in meta-generation algorithms crucially often share similarities in input. Most importantly, they often share *prefixes* across generation calls. This provides an opportunity to save on computation and dramatically speed up generation throughput.

KV Cache and state reuse. In typical transformers, because the KV Cache is updated by appending the keys and values of a new token to the cache, the KV Cache for shared prefixes can be "prefilled" only a single time and reused across generation calls that share this input prefix.

For example, in parallel meta-generation algorithms (§4.2) such as Best-of-N, when producing an N-best list $\{y^{(n)}\}_{n=1}^{N} \sim g$, generating each candidate y^i requires a "prefill" step computing S_x in order to sample y_1^i and each successive token in y^i. Simply computing S_x once and reusing it when sampling each y^i can save significant computation and time, in effect reducing the input token count for such algorithms by a factor of N (Table 5).

Making such state sharing efficient requires careful handling of the state in memory, but can significantly speed up throughput by allowing more outputs to be processed at a time as a result of lightened GPU memory requirements (Kwon et al., 2023). It can also be generalized beyond a single prefix being shared (Zheng et al., 2023) in order to handle branching, complex trees of already-processed inputs. Later work has shown that redundant computation can be eliminated even further, allowing specific speed optimizations in the presence of shared prefixes (Juravsky et al., 2024).

KV Cache and state compression. A complementary line of work approaches the challenge of handling reused model states or KV Caches efficiently by *compressing* them, reducing the storage required. Gisting (Mu et al., 2023) and other related techniques (Chevalier et al., 2023; Ge et al., 2024b) tackle the sub-problem of *long, frequently-recurring input prompts* by learning to produce a series of "soft" tokens (trained token embeddings) which compress a given large input prompt into a much smaller, more compact state. These methods can be viewed as a generalization of prefix tuning or prompt tuning (Li & Liang, 2021; Lester et al., 2021). Other methods explore the shortening of KV Caches via determining which items to retain or evict from the input prompt, or at each step whether to append new keys and values to the cache (Ge et al., 2024a; Liu et al., 2023; Zhang et al., 2023; Li et al., 2024b; Nawrot et al., 2024; Raposo et al., 2024; Xiao et al., 2024b).

Much like model weights, the KV Cache can also be compressed via reducing its storage precision, such as via quantization (Sheng et al., 2023; Lin et al., 2024b; Zhao et al., 2024b; Zirui Liu et al., 2023). Thus, the memory bandwidth cost of loading the KV Cache from memory is reduced, and more tokens' caches can be fit onto GPU memory. However, again, these compression techniques can lose model quality when applied aggressively.

7.3.2 Optimizing computational graphs

Finally, a class of optimizations takes into consideration the programmatic structure of the meta-generator.

Caching. Caching model state across calls to a generator as done by Zheng et al. (2023); Juravsky et al. (2024); Kwon et al. (2023) and discussed previously can be beneficial for algorithms that involve backtracking (e.g., tree search), or in general programs that involve duplicate generator calls.

Graph optimization. Additionally, the computational graph of such programs or algorithms can be optimized and rewritten with efficiency in mind, by hand or automatically. For example, SGLang uses GPT-4 in its optimization of programs to reorder computational graph nodes when possible (Zheng et al., 2023), and DSPy optimizes performance or cost of LM programs via automating prompt engineering (Khattab et al., 2023).

Algorithm-specific optimization. When the specific algorithm is known, optimizations can be made even more targeted, such as speeding up voting algorithms by stopping early upon converging to a majority (Aggarwal et al., 2023), or a host of methods that optimize MBR-style algorithms, including confidence-based hypothesis pruning (Cheng & Vlachos, 2023) or leveraging a reduction of MBR to the medioid identification problem (Jinnai & Ariu, 2024).

7.4 Libraries and tools for fast generation

We briefly note a few useful libraries and tools for fast and efficient generation, although the space of useful tools and libraries is in particular especially subject to fast change.

vLLM (Kwon et al., 2023) is a highly popular library that introduced PagedAttention and implements a number of up-to-date optimizations for fast generation, including continuous batching, prefix caching and reuse, various model and KV cache quantization techniques, speculative decoding, and more. TensorRT-LLM is another highly efficient LLM serving library. Especially relevant to this survey, SGLang (Zheng et al., 2023) builds on vLLM to provide a domain-specific language optimized for meta-generation.

GPT-Fast (PyTorch, 2023) provides a minimal implementation of latency-constrained fast decoding in PyTorch, and is designed to be useful for prototyping new ideas and to demonstrate the ease of optimizing low-latency unbatched decoding workloads using simple tools such as `torch.compile`.

For end users, especially those without easy access to data center-grade or high-end consumer-grade GPUs, a number of libraries also implement fast decoding on CPU, which presents its own set of challenges not fully explored in this paper. Libraries such as Llama.cpp[3] are popular for consumers, and libraries such as DeepSpeed-Inference (Aminabadi et al., 2022) or PowerInfer (Song et al., 2023; Xue et al., 2024) explore optimizations when *offloading* activations or parameters to slower-access storage or CPU RAM, which require systems considerations beyond those discussed for the more typical homogenous accelerator setting.

8 Discussion: why use sophisticated generation algorithms?

Finally, we return to the question that we posed in the introduction: *why are sophisticated generation algorithms needed at all?* For example, we might imagine that simply sampling once from the model's unmodified output distribution, $y \sim p_\theta(y|x)$ is sufficient. We offer some takeaways based on our survey.

Takeaway 1: iron out degeneracies in the learned distribution. In §3.2 and §3.3 we discussed introducing token-level adapter algorithms to avoid errors in the model's distribution (for instance, when the learned model assigns too much probability to sequences that have low or zero probability under the true distribution). At a qualitative level, examples encountered in practice include ancestral sampling resulting in incoherent sequences. At another extreme, MAP decoding algorithms (§3.1) can result in unnaturally repetitive sequences that are nevertheless assigned high probability by the model, or even empty sequences.

[3] https://github.com/ggerganov/llama.cpp

These degeneracies motivate the use of generation algorithms with alternative goals, such as minimizing Bayes Risk (§4.2.2), or the use of a token-level adapter. In these cases, generation algorithms offer a mechanism for modifying the resulting generation distribution to remove these errors.

Moving forward. As models improve, will generation algorithms for these cases still be needed? Since the aforementioned errors stem from a model imperfection, it is plausible that future models will not have these imperfections. Moreover, taking simplicity of the overall generation system as an objective to strive for, we might explicitly strive to ultimately remove the generation algorithm for these cases. On the other hand, imperfections may come from subtle design choices, such as the use of softmax (Finlayson et al., 2024a), or the choice of an autoregressive architecture. Therefore, we speculate that on the way to achieving this objective, it will remain important to identify degeneracies in existing models, introduce practical methods to mitigate them, and ultimately gain an understanding that can be used in the design of new methods.

Takeaway 2: align the generation distribution with a new objective. Above we discussed how the learned distribution p_θ may not equal the desired distribution of generations q_* (e.g., §2.2.3, §3.4). For instance, language modeling corpora often contain sequences considered offensive in many contexts (Gehman et al., 2020), and we may want to generate only non-offensive outputs. We have seen examples of generation algorithms that reweight the model's distribution using another model (e.g., one trained to adjust the distribution so that it optimizes a non-toxicity reward, §3.4), or draw a large number of samples from the model and select one with a reward function (a form of rejection sampling, §4.2.3). In these cases, the generation algorithm offers a layer of "control" over the generation distribution, allowing us to shift the generation distribution to a desired one.

Moving forward. Moving forward, we speculate that the language model's learned distribution will not always align with the desired distribution for all possible uses. Thus, using a generation algorithm to shift the model's distribution to a desired one may withstand the test of time. On the other hand, previously we discussed the connection between generation algorithms and reinforcement learning (e.g., see refer to the discussion of Equations 19, 27). Indeed, if we have a reward function, then for a particular application a model may be finetuned to match the distribution induced by the reward function, offering a potential channel to removing the complexity associated with generation algorithms.

In the near term, users may interact with models through black-box APIs that are simply imperfect for their application of interest, but obtaining a filtering function or scoring model is relatively straightforward. Thus we speculate that in practical cases, users will continue to benefit from algorithms such as rejection sampling (§4.2.3) that shift the generation distribution using repeated calls to the generator model.

Takeaway 3: dynamic computation. In §4, we discussed how generation algorithms can be viewed as searching through the output space for a desired sequence (§4.3), and that generation algorithms offer a mechanism for using compute to expand the coverage of states explored during the search. For example, best-of-N algorithms (Equation 5) use compute to search for a solution by generating N hypotheses. More formally, we can write down the objective of many algorithms as approximating the maximization of some scoring function, and doing so with a sampling-based approximation indeed results in a better approximate solution as the number of samples increases (e.g., see Equations 28, 47). We saw an example (Figure 5) where taking this approach was effective for increasing the probability of a generator's output being correct in reasoning problems. In this sense, generation algorithms offer a mechanism for dynamically expending compute with a language model to improve performance.

Moving forward. Moving forward, we see several cases. For some tasks, models may become so good that a single sequence is all that is needed to arrive at a desired state with near 100% probability. In these cases, the generation algorithm may not be useful. In challenging cases, the model may have acquired a useful representation, but may benefit from exploring the output space through backtracking, revision, etc. before arriving at a final solution. In these cases, the computation expended at inference time remains useful. Nevertheless, there are at least two potential alternatives to generation algorithms with respect to dynamic computation. One is building dynamic computation into the architecture (Raposo et al., 2024) or learning algorithm (Goyal et al., 2024; Zelikman et al., 2024a), so that (for instance) allocating more compute to harder problems is automatically handled by the model. A second alternative is to *learn the*

search algorithm (Chang et al., 2015; Lehnert et al., 2024; Gandhi et al., 2024), then use a vanilla generation algorithm.

In the nearer term, at the application level users may interact with models through black-box APIs that are simply imperfect. Thus we speculate that users will continue to benefit from using the API as a hypothesis generator that is called multiple times within a search algorithm, rather than a model that is called once for an answer. From a research perspective, the relationship between generation-time compute and performance requires further investigation. For instance, we saw above that the relationship varies by generation algorithm in voting settings (§6.2). In general, designing algorithms that optimally leverage test-time compute and studying their trade-offs requires further research (Chen et al., 2024a; Snell et al., 2024; Wu et al., 2024).

Takeaway 4: leveraging external information. In §5 we saw several ways in which generation algorithms incorporate information that is external to the language model. This includes predictions from other models, instructions or few-shot examples in prompts, external tools or verifiers, or generally inputs and outputs from an external environment.

Moving forward. Moving forward, we may expect that a generation algorithm that has information beyond that present in a language model may exceed the performance of the language model on its own. We speculate that language models on their own may either be unable to solve subproblems that are required for a complete generation (e.g., performing an algebraic computation that is simple for a computer algebra system, or retrieving a piece of information that is outside of the model's parametric knowledge), or that doing so is an inefficient allocation of computation. Moreover, we expect that in many challenging settings, it may be necessary to decompose problems into a cascade of generations, interact with an environment, and iteratively arrive at a final generation. In all of these cases, generation algorithms that incorporate external information, be it in prompts that alter a model's distribution within a chain (§4.1) or environmental feedback (§4.4, §5.2), offer a range of possibilities to explore in future research.

Finally, we expect that discrete autoregressive sequence generators will increasingly be used in domains traditionally outside of those considered in natural language processing, such as a component in an 'agent' that receives states and takes actions in a potentially stochastic environment. In these cases, using external information is inherent to the problem. How this translates to generation algorithms remains an open area of research. For instance, notions of 'planning' that are inherent to several methods discussed in this review (e.g., §4.3, §5.2) are natural fits for planning actions in interactive environments. In the near term, we expect to see generation algorithms developed in the context of language generation such as refinement (§4.4) integrated into these settings.

Takeaway 5: speed up generation. Finally, in §7 we saw examples of how generation algorithms can themselves be used to speed up generation, even in the case of ancestral sampling from a language model (§7.2). On the other hand, the increasingly diverse configurations of meta-generation algorithms present new demands and opportunities for fast generation (§7.3).

Moving forward. Regardless of the future form of sequence generators, we expect that there will always be a need and opportunity for speeding up generation. Naturally, one must consider the expected marginal benefit of developing a new method for speeding up generation compared to existing methods. Given the evolving nature of model scale, architectures, applications, and compute environments, we expect the marginal benefits of new methods to remain high for the foreseeable future. In particular, optimized generation algorithms that involve multiple models, external information, and/or cascaded generation is a nascent research area.

9 Conclusion

We surveyed generation algorithms for language models. We motivated generation algorithms, formalized their goals, and provided a unified treatment of three themes: token-level generation algorithms, meta-generation algorithms, and efficient generation. Our survey brings together past research from the decoding, LLM reasoning, and machine learning systems communities, and identifies directions for future work.

References

David H. Ackley, Geoffrey E. Hinton, and Terrence J. Sejnowski. A learning algorithm for boltzmann machines. *Cognitive Science*, 9(1):147–169, 1985. ISSN 0364-0213. doi: https://doi.org/10.1016/S0364-0213(85)80012-4. URL https://www.sciencedirect.com/science/article/pii/S0364021385800124.

Pranjal Aggarwal, Aman Madaan, Yiming Yang, and Mausam. Let's sample step by step: Adaptive-consistency for efficient reasoning and coding with llms. In *Conference on Empirical Methods in Natural Language Processing*, 2023. URL https://api.semanticscholar.org/CorpusID:258823191.

Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebron, and Sumit Sanghai. GQA: Training generalized multi-query transformer models from multi-head checkpoints. In Houda Bouamor, Juan Pino, and Kalika Bali (eds.), *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 4895–4901. Association for Computational Linguistics, 2023a. doi: 10.18653/v1/2023.emnlp-main.298. URL https://aclanthology.org/2023.emnlp-main.298.

Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. Gqa: Training generalized multi-query transformer models from multi-head checkpoints, 2023b.

Reza Yazdani Aminabadi, Samyam Rajbhandari, Minjia Zhang, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Jeff Rasley, Shaden Smith, Olatunji Ruwase, and Yuxiong He. Deepspeed inference: Enabling efficient inference of transformer models at unprecedented scale, 2022.

Quentin Anthony, Jacob Hatef, Deepak Narayanan, Stella Biderman, Stas Bekman, Junqi Yin, Aamir Shafi, Hari Subramoni, and Dhabaleswar Panda. The case for co-designing model architectures with hardware, 2024.

Simran Arora, Sabri Eyuboglu, Michael Zhang, Aman Timalsina, Silas Alberti, Dylan Zinsley, James Zou, Atri Rudra, and Christopher Ré. Simple linear attention language models balance the recall-throughput tradeoff, 2024.

Saleh Ashkboos, Amirkeivan Mohtashami, Maximilian L. Croci, Bo Li, Martin Jaggi, Dan Alistarh, Torsten Hoefler, and James Hensman. Quarot: Outlier-free 4-bit inference in rotated llms, 2024.

Amanda Askell, Yuntao Bai, Anna Chen, Dawn Drain, Deep Ganguli, Tom Henighan, Andy Jones, Nicholas Joseph, Ben Mann, Nova DasSarma, Nelson Elhage, Zac Hatfield-Dodds, Danny Hernandez, Jackson Kernion, Kamal Ndousse, Catherine Olsson, Dario Amodei, Tom Brown, Jack Clark, Sam McCandlish, Chris Olah, and Jared Kaplan. A general language assistant as a laboratory for alignment, 2021.

Zhangir Azerbayev, Hailey Schoelkopf, Keiran Paster, Marco Dos Santos, Stephen Marcus McAleer, Albert Q. Jiang, Jia Deng, Stella Biderman, and Sean Welleck. Llemma: An open language model for mathematics. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=4WnqRR915j.

Yuntao Bai, Saurav Kadavath, Sandipan Kundu, Amanda Askell, John Kernion, Andy Jones, A. Chen, Anna Goldie, Azalia Mirhoseini, C. McKinnon, Carol Chen, Catherine Olsson, C. Olah, Danny Hernandez, Dawn Drain, Deep Ganguli, Dustin Li, Eli Tran-Johnson, E. Perez, Jamie Kerr, J. Mueller, Jeff Ladish, J. Landau, Kamal Ndousse, Kamilė Lukošiūtė, Liane Lovitt, Michael Sellitto, Nelson Elhage, Nicholas Schiefer, Noem'i Mercado, Nova Dassarma, R. Lasenby, Robin Larson, Sam Ringer, Scott Johnston, Shauna Kravec, S. E. Showk, Stanislav Fort, Tamera Lanham, Timothy Telleen-Lawton, Tom Conerly, T. Henighan, Tristan Hume, Sam Bowman, Zac Hatfield-Dodds, Benjamin Mann, Dario Amodei, Nicholas Joseph, Sam McCandlish, Tom B. Brown, and Jared Kaplan. Constitutional ai: Harmlessness from ai feedback. *ArXiv preprint*, abs/2212.08073, 2022. URL https://arxiv.org/abs/2212.08073.

Sourya Basu, Govardana Sachitanandam Ramachandran, Nitish Shirish Keskar, and Lav R. Varshney. 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, 2021. URL https://openreview.net/forum?id=W1G1JZEIy5_.

Ahmad Beirami, Alekh Agarwal, Jonathan Berant, Alex D'Amour, Jacob Eisenstein, Chirag Nagpal, and Ananda Theertha Suresh. Theoretical guarantees on the best-of-n alignment policy. 2024. URL https://api.semanticscholar.org/CorpusID:266741736.

Amanda Bertsch, Alex Xie, Graham Neubig, and Matthew Gormley. It's MBR all the way down: Modern generation techniques through the lens of minimum Bayes risk. In Yanai Elazar, Allyson Ettinger, Nora Kassner, Sebastian Ruder, and Noah A. Smith (eds.), *Proceedings of the Big Picture Workshop*, pp. 108–122. Association for Computational Linguistics, 2023. doi: 10.18653/v1/2023.bigpicture-1.9. URL https://aclanthology.org/2023.bigpicture-1.9.

Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Michał Podstawski, Hubert Niewiadomski, Piotr Nyczyk, and Torsten Hoefler. Graph of Thoughts: Solving Elaborate Problems with Large Language Models. *Proceedings of the AAAI Conference on Artificial Intelligence*, 38(16):17682–17690, 2024. doi: 10.1609/aaai.v38i16.29720. URL https://ojs.aaai.org/index.php/AAAI/article/view/29720.

Peter Bickel and Kjell Doksum. *Mathematical Statistics: Basic Ideas and Selected Topics.*, volume 56. 1977. doi: 10.2307/2286373.

Peter F. Brown, Stephen A. Della Pietra, Vincent J. Della Pietra, and Robert L. Mercer. The mathematics of statistical machine translation: Parameter estimation. *Computational Linguistics*, 19(2):263–311, 1993. URL https://aclanthology.org/J93-2003.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin (eds.), *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020. URL https://proceedings.neurips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html.

Tianle Cai, Yuhong Li, Zhengyang Geng, Hongwu Peng, Jason D. Lee, Deming Chen, and Tri Dao. Medusa: Simple llm inference acceleration framework with multiple decoding heads. *arXiv preprint arXiv: 2401.10774*, 2024.

Nicholas Carlini, Daniel Paleka, Krishnamurthy Dvijotham, Thomas Steinke, Jonathan Hayase, A. F. Cooper, Katherine Lee, Matthew Jagielski, Milad Nasr, Arthur Conmy, Eric Wallace, David Rolnick, and Florian Tramèr. Stealing part of a production language model. *ArXiv preprint*, abs/2403.06634, 2024. URL https://arxiv.org/abs/2403.06634.

Louis Castricato, Alexander Havrilla, Shahbuland Matiana, Michael Pieler, Anbang Ye, Ian Yang, Spencer Frazier, and Mark Riedl. Robust preference learning for storytelling via contrastive reinforcement learning, 2022.

Kai-Wei Chang, Akshay Krishnamurthy, Alekh Agarwal, Hal Daumé III, and John Langford. Learning to search better than your teacher. In Francis R. Bach and David M. Blei (eds.), *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*, volume 37 of *JMLR Workshop and Conference Proceedings*, pp. 2058–2066. JMLR.org, 2015. URL http://proceedings.mlr.press/v37/changb15.html.

Eugene Charniak and Mark Johnson. Coarse-to-fine n-best parsing and MaxEnt discriminative reranking. In *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL'05)*, pp. 173–180. Association for Computational Linguistics, 2005. doi: 10.3115/1219840.1219862. URL https://aclanthology.org/P05-1022.

Harrison Chase. LangChain, 2022. URL https://github.com/langchain-ai/langchain.

Charlie Chen, Sebastian Borgeaud, Geoffrey Irving, Jean-Baptiste Lespiau, Laurent Sifre, and John Jumper. Accelerating large language model decoding with speculative sampling, 2023a.

Lingjiao Chen, Matei Zaharia, and James Zou. Frugalgpt: How to use large language models while reducing cost and improving performance, 2023b.

Lingjiao Chen, Jared Quincy Davis, Boris Hanin, Peter Bailis, Ion Stoica, Matei Zaharia, and James Zou. Are more llm calls all you need? towards scaling laws of compound inference systems, 2024a.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde, Jared Kaplan, Harrison Edwards, Yura Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, S. Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, F. Such, D. Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William H. Guss, Alex Nichol, Igor Babuschkin, S. Balaji, Shantanu Jain, A. Carr, J. Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, M. Knight, Miles Brundage, Mira Murati, Katie Mayer, P. Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, I. Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *ArXiv preprint*, abs/2107.03374, 2021. URL https://arxiv.org/abs/2107.03374.

Xinyun Chen, Renat Aksitov, Uri Alon, Jie Ren, Kefan Xiao, Pengcheng Yin, Sushant Prakash, Charles Sutton, Xuezhi Wang, and Denny Zhou. Universal self-consistency for large language model generation, 2023c.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. In *The Twelfth International Conference on Learning Representations*, 2024b. URL https://openreview.net/forum?id=KuPixIqPiq.

Zhuoming Chen, Avner May, Ruslan Svirschevski, Yuhsun Huang, Max Ryabinin, Zhihao Jia, and Beidi Chen. Sequoia: Scalable, robust, and hardware-aware speculative decoding, 2024c.

Julius Cheng and Andreas Vlachos. Faster minimum Bayes risk decoding with confidence-based pruning. In Houda Bouamor, Juan Pino, and Kalika Bali (eds.), *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 12473–12480. Association for Computational Linguistics, 2023. doi: 10.18653/v1/2023.emnlp-main.767. URL https://aclanthology.org/2023.emnlp-main.767.

Alexis Chevalier, Alexander Wettig, Anirudh Ajith, and Danqi Chen. Adapting language models to compress contexts, 2023.

Yizhou Chi, Kevin Yang, and Dan Klein. Thoughtsculpt: Reasoning with intermediate revision and search. 2024. URL https://api.semanticscholar.org/CorpusID:269010005.

Krishna Teja Chitty-Venkata, Sparsh Mittal, Murali Emani, Venkatram Vishwanath, and Arun K. Somani. A survey of techniques for optimizing transformer inference, 2023. URL https://arxiv.org/abs/2307.07982.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems. *ArXiv preprint*, abs/2110.14168, 2021. URL https://arxiv.org/abs/2110.14168.

Michael Collins. Discriminative reranking for natural language parsing. In Pat Langley (ed.), *Proceedings of the Seventeenth International Conference on Machine Learning (ICML 2000), Stanford University, Stanford, CA, USA, June 29 - July 2, 2000*, pp. 175–182. Morgan Kaufmann, 2000.

Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning. *ArXiv preprint*, abs/2307.08691, 2023. URL https://arxiv.org/abs/2307.08691.

Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher R'e. Flashattention: Fast and memory-efficient exact attention with io-awareness. *ArXiv preprint*, abs/2205.14135, 2022. URL https://arxiv.org/abs/2205.14135.

Tri Dao, Daniel Haziza, Francisco Massa, and Grigory Sizov. Flash-decoding for long-context inference. 2023. URL https://pytorch.org/blog/flash-decoding/.

Soham De, Samuel L. Smith, Anushan Fernando, Aleksandar Botev, George Cristian-Muraru, Albert Gu, Ruba Haroun, Leonard Berrada, Yutian Chen, Srivatsan Srinivasan, Guillaume Desjardins, Arnaud Doucet, David Budden, Yee Whye Teh, Razvan Pascanu, Nando De Freitas, and Caglar Gulcehre. Griffin: Mixing gated linear recurrences with local attention for efficient language models, 2024.

DeepSeek-AI. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model, 2024.

Haikang Deng and Colin Raffel. Reward-augmented decoding: Efficient controlled text generation with a unidirectional reward model. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 2023. doi: 10.18653/v1/2023.emnlp-main.721. URL http://dx.doi.org/10.18653/v1/2023.emnlp-main.721.

Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. GPT3.int8(): 8-bit matrix multiplication for transformers at scale. In Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho (eds.), *Advances in Neural Information Processing Systems*, 2022. URL https://openreview.net/forum?id=dXiGWqBoxaD.

Tim Dettmers, Ruslan Svirschevski, Vage Egiazarian, Denis Kuznedelev, Elias Frantar, Saleh Ashkboos, Alexander Borzunov, Torsten Hoefler, and Dan Alistarh. Spqr: A sparse-quantized representation for near-lossless llm weight compression, 2023.

Sander Dieleman. Musings on typicality, 2020. URL https://benanne.github.io/2020/09/01/typicality.html.

David Dohan, Winnie Xu, Aitor Lewkowycz, Jacob Austin, David Bieber, Raphael Gontijo Lopes, Yuhuai Wu, Henryk Michalewski, Rif A. Saurous, Jascha Sohl-dickstein, Kevin Murphy, and Charles Sutton. Language model cascades, 2022. URL https://arxiv.org/abs/2207.10342.

Yann Dubois, Xuechen Li, Rohan Taori, Tianyi Zhang, Ishaan Gulrajani, Jimmy Ba, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. Alpacafarm: A simulation framework for methods that learn from human feedback, 2023.

Nouha Dziri, Ximing Lu, Melanie Sclar, Xiang Lorraine Li, Liwei Jiang, Bill Yuchen Lin, Sean Welleck, Peter West, Chandra Bhagavatula, Ronan Le Bras, Jena D. Hwang, Soumya Sanyal, Xiang Ren, Allyson Ettinger, Zaid Harchaoui, and Yejin Choi. Faith and fate: Limits of transformers on compositionality. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=Fkckkr3ya8.

Bryan Eikema. The effect of generalisation on the inadequacy of the mode. In Raúl Vázquez, Hande Celikkanat, Dennis Ulmer, Jörg Tiedemann, Swabha Swayamdipta, Wilker Aziz, Barbara Plank, Joris Baan, and Marie-Catherine de Marneffe (eds.), *Proceedings of the 1st Workshop on Uncertainty-Aware NLP (UncertaiNLP 2024)*, pp. 87–92. Association for Computational Linguistics, 2024. URL https://aclanthology.org/2024.uncertainlp-1.9.

Bryan Eikema and Wilker Aziz. Is MAP decoding all you need? the inadequacy of the mode in neural machine translation. In *Proceedings of the 28th International Conference on Computational Linguistics*, pp. 4506–4520. International Committee on Computational Linguistics, 2020. doi: 10.18653/v1/2020.coling-main.398. URL https://aclanthology.org/2020.coling-main.398.

Jeffrey L. Elman. Finding structure in time. *Cogn. Sci.*, 14:179–211, 1990. URL https://api.semanticscholar.org/CorpusID:2763403.

Angela Fan, Mike Lewis, and Yann Dauphin. Hierarchical neural story generation. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 889–898. Association for Computational Linguistics, 2018. doi: 10.18653/v1/P18-1082. URL https://aclanthology.org/P18-1082.

Matthew Finlayson, John Hewitt, Alexander Koller, Swabha Swayamdipta, and Ashish Sabharwal. Closing the curious case of neural text degeneration. In *The Twelfth International Conference on Learning Representations*, 2024a. URL https://openreview.net/forum?id=dONpC9GL1o.

Matthew Finlayson, Xiang Ren, and Swabha Swayamdipta. Logits of api-protected llms leak proprietary information. *ArXiv preprint*, abs/2403.09539, 2024b. URL https://arxiv.org/abs/2403.09539.

Elias Frantar and Dan Alistarh. Sparsegpt: Massive language models can be accurately pruned in one-shot, 2023.

Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. OPTQ: Accurate quantization for generative pre-trained transformers. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=tcbBPnfwxS.

Markus Freitag and Yaser Al-Onaizan. Beam search strategies for neural machine translation. In *Proceedings of the First Workshop on Neural Machine Translation*, pp. 56–60. Association for Computational Linguistics, 2017. doi: 10.18653/v1/W17-3207. URL https://aclanthology.org/W17-3207.

Markus Freitag, Behrooz Ghorbani, and Patrick Fernandes. Epsilon sampling rocks: Investigating sampling strategies for minimum Bayes risk decoding for machine translation. In Houda Bouamor, Juan Pino, and Kalika Bali (eds.), *Findings of the Association for Computational Linguistics: EMNLP 2023*, pp. 9198–9209. Association for Computational Linguistics, 2023. doi: 10.18653/v1/2023.findings-emnlp.617. URL https://aclanthology.org/2023.findings-emnlp.617.

Daniel Y. Fu, Tri Dao, Khaled K. Saab, Armin W. Thomas, Atri Rudra, and Christopher Ré. Hungry hungry hippos: Towards language modeling with state space models, 2023.

Yichao Fu, Peter Bailis, Ion Stoica, and Hao Zhang. Break the sequential dependency of llm inference using lookahead decoding, 2024.

Kanishk Gandhi, Denise Lee, Gabriel Grand, Muxin Liu, Winson Cheng, Archit Sharma, and Noah D. Goodman. Stream of search (sos): Learning to search in language, 2024.

Leo Gao, John Schulman, and Jacob Hilton. Scaling laws for reward model overoptimization. In *International Conference on Machine Learning*, 2022. URL https://api.semanticscholar.org/CorpusID:252992904.

Suyu Ge, Yunan Zhang, Liyuan Liu, Minjia Zhang, Jiawei Han, and Jianfeng Gao. Model tells you what to discard: Adaptive kv cache compression for llms, 2024a.

Tao Ge, Jing Hu, Lei Wang, Xun Wang, Si-Qing Chen, and Furu Wei. In-context autoencoder for context compression in a large language model, 2024b.

Samuel Gehman, Suchin Gururangan, Maarten Sap, Yejin Choi, and Noah A. Smith. RealToxicityPrompts: Evaluating neural toxic degeneration in language models. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pp. 3356–3369. Association for Computational Linguistics, 2020. doi: 10.18653/v1/2020.findings-emnlp.301. URL https://aclanthology.org/2020.findings-emnlp.301.

Gemini Team et al. Gemini: A family of highly capable multimodal models, 2023.

V GOEL. Minimum bayes-risk automatic speech recognition. *Pattern Recognition in Speech and Language Processing*, 2003.

V. Goel, S. Kumar, and W. Byrne. Segmental minimum bayes-risk decoding for automatic speech recognition. *IEEE Transactions on Speech and Audio Processing*, 12(3):234–249, 2004. doi: 10.1109/TSA.2004.825678.

Jesús González-Rubio and Francisco Casacuberta. Improving the minimum Bayes' risk combination of machine translation systems. In *Proceedings of the 10th International Workshop on Spoken Language Translation: Papers*, 2013. URL https://aclanthology.org/2013.iwslt-papers.4.

Jesús González-Rubio, Alfons Juan, and Francisco Casacuberta. Minimum Bayes-risk system combination. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pp. 1268–1277. Association for Computational Linguistics, 2011. URL https://aclanthology.org/P11-1127.

Sachin Goyal, Ziwei Ji, Ankit Singh Rawat, Aditya Krishna Menon, Sanjiv Kumar, and Vaishnavh Nagarajan. Think before you speak: Training language models with pause tokens, 2024.

Alex Graves. Sequence transduction with recurrent neural networks, 2012.

Albert Gu and Tri Dao. Mamba: Linear-time sequence modeling with selective state spaces. *ArXiv preprint*, abs/2312.00752, 2023. URL https://arxiv.org/abs/2312.00752.

Albert Gu, Karan Goel, and Christopher Ré. Efficiently modeling long sequences with structured state spaces. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net, 2022. URL https://openreview.net/forum?id=uYLFoz1vlAC.

Alex Havrilla, Sharath Chandra Raparthy, Christoforus Nalmpantis, Jane Dwivedi-Yu, Maksym Zhuravinskyi, Eric Hambro, and Roberta Railneau. Glore: When, where, and how to improve llm reasoning via global and local refinements. *ArXiv preprint*, abs/2402.10963, 2024. URL https://arxiv.org/abs/2402.10963.

Georg Heigold, Wolfgang Macherey, Ralf Schluter, and Hermann Ney. Minimum exact word error training. In *IEEE Workshop on Automatic Speech Recognition and Understanding, 2005.*, pp. 186–190. IEEE, 2005.

John Hewitt, Christopher Manning, and Percy Liang. Truncation sampling as language model desmoothing. In *Findings of the Association for Computational Linguistics: EMNLP 2022*, pp. 3414–3427. Association for Computational Linguistics, 2022. URL https://aclanthology.org/2022.findings-emnlp.249.

Geoffrey Hinton, Oriol Vinyals, and Jeffrey Dean. Distilling the knowledge in a neural network. volume abs/1503.02531, 2015. URL https://arxiv.org/abs/1503.02531.

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katherine Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Oriol Vinyals, Jack William Rae, and Laurent Sifre. An empirical analysis of compute-optimal large language model training. In Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho (eds.), *Advances in Neural Information Processing Systems*, 2022. URL https://openreview.net/forum?id=iBBcRUlOAPR.

Chris Hokamp and Qun Liu. Lexically constrained decoding for sequence generation using grid beam search. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1535–1546. Association for Computational Linguistics, 2017. doi: 10.18653/v1/P17-1141. URL https://aclanthology.org/P17-1141.

Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. 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, 2020. URL https://openreview.net/forum?id=rygGQyrFvH.

Sara Hooker. The hardware lottery, 2020.

Coleman Hooper, Sehoon Kim, Hiva Mohammadzadeh, Michael W. Mahoney, Yakun Sophia Shao, Kurt Keutzer, and Amir Gholami. Kvquant: Towards 10 million context length llm inference with kv cache quantization, 2024.

Edward J Hu, Moksh Jain, Eric Elmoznino, Younesse Kaddar, Guillaume Lajoie, Yoshua Bengio, and Nikolay Malkin. Amortizing intractable inference in large language models. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=Ouj6p4ca60.

Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. Large language models cannot self-correct reasoning yet. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=IkmD3fKBPQ.

Liang Huang and David Chiang. Forest rescoring: Faster decoding with integrated language models. In *Proceedings of the 45th Annual Meeting of the Association of Computational Linguistics*, pp. 144–151. Association for Computational Linguistics, 2007. URL https://aclanthology.org/P07-1019.

DeLesley Hutchins, Imanol Schlag, Yuhuai Wu, Ethan Dyer, and Behnam Neyshabur. Block-recurrent transformers. In Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho (eds.), *Advances in Neural Information Processing Systems*, 2022. URL https://openreview.net/forum?id=uloenYmLCAo.

Hanhwi Jang, Joonsung Kim, Jae-Eon Jo, Jaewon Lee, and Jangwoo Kim. Mnnfast: a fast and scalable system architecture for memory-augmented neural networks. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, pp. 250–263. Association for Computing Machinery, 2019. ISBN 9781450366694. doi: 10.1145/3307650.3322214. URL https://doi.org/10.1145/3307650.3322214.

Albert Qiaochu Jiang, Sean Welleck, Jin Peng Zhou, Timothee Lacroix, Jiacheng Liu, Wenda Li, Mateja Jamnik, Guillaume Lample, and Yuhuai Wu. Draft, sketch, and prove: Guiding formal theorem provers with informal proofs. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=SMa9EAovKMC.

Yuu Jinnai and Kaito Ariu. Hyperparameter-free approach for faster minimum bayes risk decoding, 2024.

Andy L. Jones. Scaling scaling laws with board games, 2021.

Jordan Juravsky, Bradley Brown, Ryan Ehrlich, Daniel Y. Fu, Christopher Ré, and Azalia Mirhoseini. Hydragen: High-throughput llm inference with shared prefixes, 2024.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models, 2020.

Sayash Kapoor, Benedikt Stroebl, and Arvind Narayanan. Ai leaderboards are no longer useful. it's time to switch to pareto curves. AI Snake Oil blog, 2024a. URL https://www.aisnakeoil.com/p/ai-leaderboards-are-no-longer-useful.

Sayash Kapoor, Benedikt Stroebl, Zachary S. Siegel, Nitya Nadgir, and Arvind Narayanan. Ai agent benchmarks that matter. Manuscript submitted for publication, 2024b.

Muhammad Khalifa, Hady Elsahar, and Marc Dymetman. A distributional approach to controlled text generation. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021. URL https://openreview.net/forum?id=jWkw45-9AbL.

O. Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. Dspy: Compiling declarative language model calls into self-improving pipelines. *ArXiv preprint*, abs/2310.03714, 2023. URL https://arxiv.org/abs/2310.03714.

Mahsa Khoshnoodi, Vinija Jain, Mingye Gao, Malavika Srikanth, and Aman Chadha. A comprehensive survey of accelerated generation techniques in large language models, 2024. URL https://arxiv.org/abs/2405.13019.

Geunwoo Kim, Pierre Baldi, and Stephen McAleer. Language models can solve computer tasks. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (eds.), *Advances in Neural Information Processing Systems*, volume 36, pp. 39648–39677. Curran Associates, Inc., 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/7cc1005ec73cfbaac9fa21192b622507-Paper-Conference.pdf.

Sehoon Kim, Coleman Hooper, Amir Gholami, Zhen Dong, Xiuyu Li, Sheng Shen, Michael W. Mahoney, and Kurt Keutzer. Squeezellm: Dense-and-sparse quantization, 2024.

Brian Kingsbury, Tara N. Sainath, and Hagen Soltau. Scalable minimum bayes risk training of deep neural network acoustic models using distributed hessian-free optimization. In *Interspeech*, 2012. URL https://api.semanticscholar.org/CorpusID:9762862.

Philipp Koehn and Rebecca Knowles. Six challenges for neural machine translation. In *Proceedings of the First Workshop on Neural Machine Translation*, pp. 28–39. Association for Computational Linguistics, 2017. doi: 10.18653/v1/W17-3204. URL https://aclanthology.org/W17-3204.

Tomasz Korbak, Hady Elsahar, Germán Kruszewski, and Marc Dymetman. Controlling conditional language models without catastrophic forgetting. In Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvári, Gang Niu, and Sivan Sabato (eds.), *International Conference on Machine Learning, ICML 2022, 17-23 July 2022, Baltimore, Maryland, USA*, volume 162 of *Proceedings of Machine Learning Research*, pp. 11499–11528. PMLR, 2022a. URL https://proceedings.mlr.press/v162/korbak22a.html.

Tomasz Korbak, Hady ElSahar, Germán Kruszewski, and Marc Dymetman. On reinforcement learning and distribution matching for fine-tuning language models with no catastrophic forgetting. *ArXiv preprint*, abs/2206.00761, 2022b. URL https://arxiv.org/abs/2206.00761.

Tomasz Korbak, Ethan Perez, and Christopher Buckley. RL with KL penalties is better viewed as Bayesian inference. In *Findings of the Association for Computational Linguistics: EMNLP 2022*, pp. 1083–1091. Association for Computational Linguistics, 2022c. URL https://aclanthology.org/2022.findings-emnlp.77.

Kalpesh Krishna, Yapei Chang, John Wieting, and Mohit Iyyer. RankGen: Improving text generation with large ranking models. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pp. 199–232. Association for Computational Linguistics, 2022. URL https://aclanthology.org/2022.emnlp-main.15.

Ilia Kulikov, Alexander Miller, Kyunghyun Cho, and Jason Weston. Importance of search and evaluation strategies in neural dialogue modeling. In *Proceedings of the 12th International Conference on Natural Language Generation*, pp. 76–87. Association for Computational Linguistics, 2019. doi: 10.18653/v1/W19-8609. URL https://aclanthology.org/W19-8609.

Shankar Kumar and William Byrne. Minimum Bayes-risk decoding for statistical machine translation. In *Proceedings of the Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics: HLT-NAACL 2004*, pp. 169–176. Association for Computational Linguistics, 2004. URL https://aclanthology.org/N04-1022.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention, 2023.

Guillaume Lample, Timothee Lacroix, Marie anne Lachaux, Aurelien Rodriguez, Amaury Hayat, Thibaut Lavril, Gabriel Ebner, and Xavier Martinet. Hypertree proof search for neural theorem proving. In Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho (eds.), *Advances in Neural Information Processing Systems*, 2022. URL https://openreview.net/forum?id=J4pX8Q8cxHH.

Yann LeCun, Sumit Chopra, Raia Hadsell, Aurelio Ranzato, and Fu Jie Huang. A tutorial on energy-based learning. 2006. URL https://api.semanticscholar.org/CorpusID:8531544.

Lucas Lehnert, Sainbayar Sukhbaatar, Paul Mcvay, Michael Rabbat, and Yuandong Tian. Beyond a*: Better planning with transformers via search dynamics bootstrapping, 2024.

Brian Lester, Rami Al-Rfou, and Noah Constant. The power of scale for parameter-efficient prompt tuning. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pp. 3045–3059. Association for Computational Linguistics, 2021. doi: 10.18653/v1/2021.emnlp-main.243. URL https://aclanthology.org/2021.emnlp-main.243.

Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. volume abs/2211.17192, 2022. URL https://arxiv.org/abs/2211.17192.

Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding, 2023.

Aitor Lewkowycz, Anders Johan Andreassen, David Dohan, Ethan Dyer, Henryk Michalewski, Vinay Venkatesh Ramasesh, Ambrose Slone, Cem Anil, Imanol Schlag, Theo Gutman-Solo, Yuhuai Wu, Behnam Neyshabur, Guy Gur-Ari, and Vedant Misra. Solving quantitative reasoning problems with language models. In Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho (eds.), *Advances in Neural Information Processing Systems*, 2022. URL https://openreview.net/forum?id=IFXTZERXdM7.

Junyi Li, Tianyi Tang, Wayne Xin Zhao, and Ji-Rong Wen. Pretrained language model for text generation: A survey. In Zhi-Hua Zhou (ed.), *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI-21*, pp. 4492–4499. International Joint Conferences on Artificial Intelligence Organization, 8 2021. doi: 10.24963/ijcai.2021/612. URL https://doi.org/10.24963/ijcai.2021/612. Survey Track.

Kenneth Li, Samy Jelassi, Hugh Zhang, Sham M. Kakade, Martin Wattenberg, and David Brandfonbrener. Q-probe: A lightweight approach to reward maximization for language models. 2024a. URL https://api.semanticscholar.org/CorpusID:267782636.

Xiang Lisa Li and Percy Liang. Prefix-tuning: Optimizing continuous prompts for generation. 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)*, pp. 4582–4597. Association for Computational Linguistics, 2021. doi: 10.18653/v1/2021.acl-long.353. URL https://aclanthology.org/2021.acl-long.353.

Xiang Lisa Li, Ari Holtzman, Daniel Fried, Percy Liang, Jason Eisner, Tatsunori Hashimoto, Luke Zettlemoyer, and Mike Lewis. Contrastive decoding: Open-ended text generation as optimization. In Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki (eds.), *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 12286–12312. Association for Computational Linguistics, 2023a. doi: 10.18653/v1/2023.acl-long.687. URL https://aclanthology.org/2023.acl-long.687.

Yifei Li, Zeqi Lin, Shizhuo Zhang, Qiang Fu, Bei Chen, Jian-Guang Lou, and Weizhu Chen. Making language models better reasoners with step-aware verifier. In Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki (eds.), *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 5315–5333. Association for Computational Linguistics, 2023b. doi: 10.18653/v1/2023.acl-long.291. URL https://aclanthology.org/2023.acl-long.291.

Yuhong Li, Yingbing Huang, Bowen Yang, Bharat Venkitesh, Acyr Locatelli, Hanchen Ye, Tianle Cai, Patrick Lewis, and Deming Chen. Snapkv: Llm knows what you are looking for before generation, 2024b.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022. doi: 10.1126/science.abq1158. URL https://www.science.org/doi/abs/10.1126/science.abq1158.

Opher Lieber, Barak Lenz, Hofit Bata, Gal Cohen, Jhonathan Osin, Itay Dalmedigos, Erez Safahi, Shaked Meirom, Yonatan Belinkov, Shai Shalev-Shwartz, Omri Abend, Raz Alon, Tomer Asida, Amir Bergman, Roman Glozman, Michael Gokhman, Avashalom Manevich, Nir Ratner, Noam Rozen, Erez Shwartz, Mor Zusman, and Yoav Shoham. Jamba: A hybrid transformer-mamba language model, 2024.

Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=v8L0pN6EOi.

Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. Awq: Activation-aware weight quantization for llm compression and acceleration. In *MLSys*, 2024a.

Yujun Lin, Haotian Tang, Shang Yang, Zhekai Zhang, Guangxuan Xiao, Chuang Gan, and Song Han. Qserve: W4a8kv4 quantization and system co-design for efficient llm serving, 2024b.

Alisa Liu, Maarten Sap, Ximing Lu, Swabha Swayamdipta, Chandra Bhagavatula, Noah A. Smith, and Yejin Choi. 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)*, pp. 6691–6706. Association for Computational Linguistics, 2021a. doi: 10.18653/v1/2021.acl-long.522. URL https://aclanthology.org/2021.acl-long.522.

Alisa Liu, Xiaochuang Han, Yizhong Wang, Yulia Tsvetkov, Yejin Choi, and Noah A. Smith. Tuning language models by proxy, 2024a.

Jiacheng Liu, Andrew Cohen, Ramakanth Pasunuru, Yejin Choi, Hannaneh Hajishirzi, and Asli Celikyilmaz. Don't throw away your value model! generating more preferable text with value-guided monte-carlo tree search decoding. In *First Conference on Language Modeling*, 2024b. URL https://openreview.net/forum?id=kh9Zt2Ldmn.

Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing, 2021b. URL https://arxiv.org/abs/2107.13586.

Tianqi Liu, Yao Zhao, Rishabh Joshi, Misha Khalman, Mohammad Saleh, Peter J Liu, and Jialu Liu. Statistical rejection sampling improves preference optimization. In *The Twelfth International Conference on Learning Representations*, 2024c. URL https://openreview.net/forum?id=xbjSwwrQOe.

Zichang Liu, Aditya Desai, Fangshuo Liao, Weitao Wang, Victor Xie, Zhaozhuo Xu, Anastasios Kyrillidis, and Anshumali Shrivastava. Scissorhands: Exploiting the persistence of importance hypothesis for llm kv cache compression at test time, 2023.

Sidi Lu, Yaoming Zhu, Weinan Zhang, Jun Wang, and Yong Yu. Neural text generation: Past, present and beyond, 2018. URL https://arxiv.org/abs/1803.07133.

Ximing Lu, Sean Welleck, Peter West, Liwei Jiang, Jungo Kasai, Daniel Khashabi, Ronan Le Bras, Lianhui Qin, Youngjae Yu, Rowan Zellers, Noah A. Smith, and Yejin Choi. NeuroLogic a*esque decoding: Constrained text generation with lookahead heuristics. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 780–799. Association for Computational Linguistics, 2022. doi: 10.18653/v1/2022.naacl-main.57. URL https://aclanthology.org/2022.naacl-main.57.

Ximing Lu, Faeze Brahman, Peter West, Jaehun Jung, Khyathi Chandu, Abhilasha Ravichander, Prithviraj Ammanabrolu, Liwei Jiang, Sahana Ramnath, Nouha Dziri, Jillian Fisher, Bill Lin, Skyler Hallinan, Lianhui Qin, Xiang Ren, Sean Welleck, and Yejin Choi. Inference-time policy adapters (IPA): Tailoring extreme-scale LMs without fine-tuning. In Houda Bouamor, Juan Pino, and Kalika Bali (eds.), *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 6863–6883. Association for Computational Linguistics, 2023. doi: 10.18653/v1/2023.emnlp-main.424. URL https://aclanthology.org/2023.emnlp-main.424.

David John Cameron MacKay. Information theory, inference, and learning algorithms. *IEEE Transactions on Information Theory*, 50:2544–2545, 2004. URL https://api.semanticscholar.org/CorpusID:5436619.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-refine: Iterative refinement with self-feedback. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=S37hOerQLB.

Shahbuland Matiana, JR Smith, Ryan Teehan, Louis Castricato, Stella Biderman, Leo Gao, and Spencer Frazier. Cut the carp: Fishing for zero-shot story evaluation, 2021.

Clara Meister, Ryan Cotterell, and Tim Vieira. If beam search is the answer, what was the question? In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 2173–2185. Association for Computational Linguistics, 2020. doi: 10.18653/v1/2020.emnlp-main.170. URL https://aclanthology.org/2020.emnlp-main.170.

Clara Meister, Tiago Pimentel, Luca Malagutti, Ethan Wilcox, and Ryan Cotterell. On the efficacy of sampling adapters. In Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki (eds.), *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1437–1455. Association for Computational Linguistics, 2023a. doi: 10.18653/v1/2023.acl-long.80. URL https://aclanthology.org/2023.acl-long.80.

Clara Meister, Tiago Pimentel, Gian Wiher, and Ryan Cotterell. Locally typical sampling. *Transactions of the Association for Computational Linguistics*, 11:102–121, 2023b. doi: 10.1162/tacl_a_00536. URL https://aclanthology.org/2023.tacl-1.7.

William Merrill and Ashish Sabharwal. The expressive power of transformers with chain of thought. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=NjNGlPh8Wh.

Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Hongyi Jin, Tianqi Chen, and Zhihao Jia. Towards efficient generative large language model serving: A survey from algorithms to systems, 2023a. URL https://arxiv.org/abs/2312.15234.

Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Rae Ying Yee Wong, Zhuoming Chen, Daiyaan Arfeen, Reyna Abhyankar, and Zhihao Jia. Specinfer: Accelerating generative llm serving with speculative inference and token tree verification. *ArXiv preprint*, abs/2305.09781, 2023b. URL https://arxiv.org/abs/2305.09781.

Tomas Mikolov, M. Karafiát, L. Burget, J. Černocký, and S. Khudanpur. Recurrent neural network based language model. pp. 1045–1048, 2010.

Maxim Milakov and Natalia Gimelshein. Online normalizer calculation for softmax, 2018.

Jesse Mu, Xiang Lisa Li, and Noah Goodman. Learning to compress prompts with gist tokens. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=2DtxPCL3T5.

Kenton Murray and David Chiang. Correcting length bias in neural machine translation. In *Proceedings of the Third Conference on Machine Translation: Research Papers*, pp. 212–223. Association for Computational Linguistics, 2018. doi: 10.18653/v1/W18-6322. URL https://aclanthology.org/W18-6322.

Piotr Nawrot, Adrian Łańcucki, Marcin Chochowski, David Tarjan, and Edoardo M. Ponti. Dynamic memory compression: Retrofitting llms for accelerated inference, 2024.

Nathan Ng, Kyra Yee, Alexei Baevski, Myle Ott, Michael Auli, and Sergey Edunov. Facebook FAIR's WMT19 news translation task submission. In *Proceedings of the Fourth Conference on Machine Translation (Volume 2: Shared Task Papers, Day 1)*, pp. 314–319. Association for Computational Linguistics, 2019. doi: 10.18653/v1/W19-5333. URL https://aclanthology.org/W19-5333.

Ansong Ni, Srini Iyer, Dragomir Radev, Ves Stoyanov, Wen-tau Yih, Sida I Wang, and Xi Victoria Lin. Lever: Learning to verify language-to-code generation with execution. In *Proceedings of the 40th International Conference on Machine Learning (ICML'23)*, 2023.

NVIDIA. Nvidia Tesla V100 GPU architecture, 2017.

NVIDIA. Nvidia A100 tensor core GPU architecture, 2020.

NVIDIA. Nvidia H100 tensor core GPU architecture, 2022.

Franz Josef Och and Hermann Ney. Discriminative training and maximum entropy models for statistical machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pp. 295–302. Association for Computational Linguistics, 2002. doi: 10.3115/1073083.1073133. URL https://aclanthology.org/P02-1038.

Theo X. Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. Is self-repair a silver bullet for code generation? In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=y0GJXRungR.

Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke E. Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Francis Christiano, Jan Leike, and Ryan J. Lowe. Training language models to follow instructions with human feedback. *ArXiv preprint*, abs/2203.02155, 2022. URL https://arxiv.org/abs/2203.02155.

Jiayi Pan, Yichi Zhang, Nicholas Tomlin, Yifei Zhou, Sergey Levine, and Alane Suhr. Autonomous evaluation and refinement of digital agents, 2024.

Liangming Pan, Michael Stephen Saxon, Wenda Xu, Deepak Nathani, Xinyi Wang, and William Yang Wang. Automatically correcting large language models: Surveying the landscape of diverse self-correction strategies. *ArXiv preprint*, abs/2308.03188, 2023. URL https://arxiv.org/abs/2308.03188.

Adam Pauls and Dan Klein. K-best A* parsing. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP*, pp. 958–966. Association for Computational Linguistics, 2009. URL https://aclanthology.org/P09-1108.

Bo Peng, Eric Alcaide, Quentin Anthony, Alon Albalak, Samuel Arcadinho, Stella Biderman, Huanqi Cao, Xin Cheng, Michael Chung, Leon Derczynski, Xingjian Du, Matteo Grella, Kranthi Gv, Xuzheng He, Haowen Hou, Przemyslaw Kazienko, Jan Kocon, Jiaming Kong, Bartłomiej Koptyra, Hayden Lau, Jiaju Lin, Krishna Sri Ipsit Mantri, Ferdinand Mom, Atsushi Saito, Guangyu Song, Xiangru Tang, Johan Wind, Stanisław Woźniak, Zhenyuan Zhang, Qinghua Zhou, Jian Zhu, and Rui-Jie Zhu. RWKV: Reinventing RNNs for the transformer era. In Houda Bouamor, Juan Pino, and Kalika Bali (eds.), *Findings of the Association for Computational Linguistics: EMNLP 2023*, pp. 14048–14077. Association for Computational Linguistics, 2023. doi: 10.18653/v1/2023.findings-emnlp.936. URL https://aclanthology.org/2023.findings-emnlp.936.

Michael Poli, Stefano Massaroli, Eric Nguyen, Daniel Y. Fu, Tri Dao, Stephen Baccus, Yoshua Bengio, Stefano Ermon, and Christopher Ré. Hyena hierarchy: Towards larger convolutional language models, 2023.

Stanislas Polu and Ilya Sutskever. Generative language modeling for automated theorem proving, 2020.

Stanislas Polu, Jesse Michael Han, Kunhao Zheng, Mantas Baksys, Igor Babuschkin, and Ilya Sutskever. Formal mathematics statement curriculum learning. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=-P7G-8dmSh4.

David M. W. Powers. Applications and explanations of Zipf's law. In *New Methods in Language Processing and Computational Natural Language Learning*, 1998. URL https://aclanthology.org/W98-1218.

Ofir Press, Muru Zhang, Sewon Min, Ludwig Schmidt, Noah Smith, and Mike Lewis. Measuring and narrowing the compositionality gap in language models. In Houda Bouamor, Juan Pino, and Kalika Bali (eds.), *Findings of the Association for Computational Linguistics: EMNLP 2023*, pp. 5687–5711. Association for Computational Linguistics, 2023. doi: 10.18653/v1/2023.findings-emnlp.378. URL https://aclanthology.org/2023.findings-emnlp.378.

Team PyTorch. Accelerating generative ai with pytorch ii: Gpt, fast. https://pytorch.org/blog/accelerating-generative-ai-2/, 2023.

Markus N. Rabe and Charles Staats. Self-attention does not need $o(n^2)$ memory, 2022.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. Direct preference optimization: Your language model is secretly a reward model. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=HPuSIXJaa9.

David Raposo, Sam Ritter, Blake Richards, Timothy Lillicrap, Peter Conway Humphreys, and Adam Santoro. Mixture-of-depths: Dynamically allocating compute in transformer-based language models, 2024.

Alexander Rush. MiniChain, 2023. URL https://github.com/srush/MiniChain.

Swarnadeep Saha, Omer Levy, Asli Celikyilmaz, Mohit Bansal, Jason Weston, and Xian Li. Branch-solve-merge improves large language model evaluation and generation. *ArXiv preprint*, abs/2310.15123, 2023. URL https://arxiv.org/abs/2310.15123.

Pranab Sahoo, Ayush Kumar Singh, Sriparna Saha, Vinija Jain, Samrat Mondal, and Aman Chadha. A systematic survey of prompt engineering in large language models: Techniques and applications, 2024. URL https://arxiv.org/abs/2402.07927.

Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter, 2020.

William Saunders, Catherine Yeh, Jeff Wu, Steven Bills, Long Ouyang, Jonathan Ward, and Jan Leike. Self-critiquing models for assisting human evaluators, 2022.

Imanol Schlag, Sainbayar Sukhbaatar, Asli Celikyilmaz, Wen tau Yih, Jason Weston, Jürgen Schmidhuber, and Xian Li. Large language model programs, 2023.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *ArXiv*, abs/1707.06347, 2017. URL https://api.semanticscholar.org/CorpusID:28695052.

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

Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Daniel Y. Fu, Zhiqiang Xie, Beidi Chen, Clark Barrett, Joseph E. Gonzalez, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. Flexgen: High-throughput generative inference of large language models with a single gpu, 2023.

Freda Shi, Daniel Fried, Marjan Ghazvininejad, Luke Zettlemoyer, and Sida I. Wang. Natural language to code translation with execution. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pp. 3533–3546. Association for Computational Linguistics, 2022. URL https://aclanthology.org/2022.emnlp-main.231.

Quan Shi, Michael Tang, Karthik Narasimhan, and Shunyu Yao. Can language models solve olympiad programming?, 2024.

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning, 2023.

Jimmy T. H. Smith, Andrew Warrington, and Scott W. Linderman. Simplified state space layers for sequence modeling, 2023.

Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling llm test-time compute optimally can be more effective than scaling model parameters, 2024. URL https://arxiv.org/abs/2408.03314.

Yixin Song, Zeyu Mi, Haotong Xie, and Haibo Chen. Powerinfer: Fast large language model serving with a consumer-grade gpu, 2023.

Felix Stahlberg and Bill Byrne. On NMT search errors and model errors: Cat got your tongue? In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pp. 3356–3362. Association for Computational Linguistics, 2019. doi: 10.18653/v1/D19-1331. URL https://aclanthology.org/D19-1331.

Felix Stahlberg, Ilia Kulikov, and Shankar Kumar. Uncertainty determines the adequacy of the mode and the tractability of decoding in sequence-to-sequence models. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 8634–8645. Association for Computational Linguistics, 2022. doi: 10.18653/v1/2022.acl-long.591. URL https://aclanthology.org/2022.acl-long.591.

Nisan Stiennon, Long Ouyang, Jeff Wu, Daniel M. Ziegler, Ryan Lowe, Chelsea Voss, Alec Radford, Dario Amodei, and Paul Christiano. Learning to summarize from human feedback. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS'20. Curran Associates Inc., 2020. ISBN 9781713829546.

Zhiqing Sun, Longhui Yu, Yikang Shen, Weiyang Liu, Yiming Yang, Sean Welleck, and Chuang Gan. Easy-to-hard generalization: Scalable alignment beyond human supervision, 2024a.

Ziteng Sun, Ananda Theertha Suresh, Jae Hun Ro, Ahmad Beirami, Himanshu Jain, and Felix Yu. Spectr: Fast speculative decoding via optimal transport, 2024b.

Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. In Zoubin Ghahramani, Max Welling, Corinna Cortes, Neil D. Lawrence, and Kilian Q. Weinberger (eds.), *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada*, pp. 3104–3112, 2014. URL https://proceedings.neurips.cc/paper/2014/hash/a14ac55a4f27472c5d894ec1c3c743d2-Abstract.html.

Mirac Suzgun, Luke Melas-Kyriazi, and Dan Jurafsky. Follow the wisdom of the crowd: Effective text generation via minimum Bayes risk decoding. In Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki (eds.), *Findings of the Association for Computational Linguistics: ACL 2023*, pp. 4265–4293. Association for Computational Linguistics, 2023. doi: 10.18653/v1/2023.findings-acl.262. URL https://aclanthology.org/2023.findings-acl.262.

Vijay Thakkar, Pradeep Ramani, Cris Cecka, Aniket Shivam, Honghao Lu, Ethan Yan, Jack Kosaian, Mark Hoemmen, Haicheng Wu, Andrew Kerr, Matt Nicely, Duane Merrill, Dustyn Blasig, Fengqi Qiao, Piotr Majcher, Paul Springer, Markus Hohnerbach, Jin Wang, and Manish Gupta. Cutlass, 2023. URL https://github.com/NVIDIA/cutlass/tree/v3.0.0.

Ye Tian, Baolin Peng, Linfeng Song, Lifeng Jin, Dian Yu, Haitao Mi, and Dong Yu. Toward self-improvement of llms via imagination, searching, and criticizing, 2024.

Philippe Tillet, H. T. Kung, and David Cox. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2019, pp. 10–19. Association for Computing Machinery, 2019. ISBN 9781450367196. doi: 10.1145/3315508.3329973. URL https://doi.org/10.1145/3315508.3329973.

Hugo Touvron, Louis Martin, Kevin R. Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Daniel M. Bikel, Lukas Blecher, Cristian Cantón Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony S. Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel M. Kloumann, A. V. Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, R. Subramanian, Xia Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zhengxu Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. Llama 2: Open foundation and fine-tuned chat models. *ArXiv preprint*, abs/2307.09288, 2023. URL https://arxiv.org/abs/2307.09288.

Gladys Tyen, Hassan Mansoor, Victor Cărbune, Peter Chen, and Tony Mak. Llms cannot find reasoning errors, but can correct them given the error location, 2024.

Jonathan Uesato, Nate Kushman, Ramana Kumar, Francis Song, Noah Siegel, Lisa Wang, Antonia Creswell, Geoffrey Irving, and Irina Higgins. Solving math word problems with process- and outcome-based feedback, 2022.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett (eds.), *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pp. 5998–6008, 2017. URL https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html.

Giorgos Vernikos and Andrei Popescu-Belis. Don't rank, combine! combining machine translation hypotheses using quality estimation, 2024.

Peiyi Wang, Lei Li, Zhihong Shao, R. X. Xu, Damai Dai, Yifei Li, Deli Chen, Y.Wu, and Zhifang Sui. Math-shepherd: Verify and reinforce llms step-by-step without human annotations. *ArXiv preprint*, abs/2312.08935, 2023a. URL https://arxiv.org/abs/2312.08935.

Wenxiao Wang, Wei Chen, Yicong Luo, Yongliu Long, Zhengkai Lin, Liye Zhang, Binbin Lin, Deng Cai, and Xiaofei He. Model compression and efficient inference for large language models: A survey, 2024a. URL https://arxiv.org/abs/2402.09748.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. In *The Eleventh International Conference on Learning Representations*, 2023b. URL https://openreview.net/forum?id=1PL1NIMMrw.

Zhiruo Wang, Zhoujun Cheng, Hao Zhu, Daniel Fried, and Graham Neubig. What are tools anyway? a survey from the language model perspective, 2024b.

Sean Welleck and Rahul Saha. Llmstep: Llm proofstep suggestions in lean. *ArXiv preprint*, abs/2310.18457, 2023. URL https://arxiv.org/abs/2310.18457.

Sean Welleck, Ilia Kulikov, Jaedeok Kim, Richard Yuanzhe Pang, and Kyunghyun Cho. Consistency of a recurrent language model with respect to incomplete decoding. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 5553–5568. Association for Computational Linguistics, 2020. doi: 10.18653/v1/2020.emnlp-main.448. URL https://aclanthology.org/2020.emnlp-main.448.

Sean Welleck, Jiacheng Liu, Ximing Lu, Hannaneh Hajishirzi, and Yejin Choi. Naturalprover: Grounded mathematical proof generation with language models. In Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho (eds.), *Advances in Neural Information Processing Systems*, 2022. URL https://openreview.net/forum?id=rhdfTOiXBng.

Sean Welleck, Ximing Lu, Peter West, Faeze Brahman, Tianxiao Shen, Daniel Khashabi, and Yejin Choi. Generating sequences by learning to self-correct. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=hH36JeQZDaO.

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

Gian Wiher, Clara Meister, and Ryan Cotterell. On decoding strategies for neural text generators, 2022. URL https://arxiv.org/abs/2203.15721.

Yangzhen Wu, Zhiqing Sun, Shanda Li, Sean Welleck, and Yiming Yang. An empirical analysis of compute-optimal inference for problem-solving with language models, 2024. URL https://arxiv.org/abs/2408.00724.

Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. Smoothquant: Accurate and efficient post-training quantization for large language models, 2024a.

Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. Efficient streaming language models with attention sinks, 2024b.

Yisheng Xiao, Lijun Wu, Junliang Guo, Juntao Li, Min Zhang, Tao Qin, and Tie yan Liu. A survey on non-autoregressive generation for neural machine translation and beyond, 2023. URL https://arxiv.org/abs/2204.09269.

Yuxi Xie, Kenji Kawaguchi, Yiran Zhao, Xu Zhao, Min-Yen Kan, Junxian He, and Qizhe Xie. Self-evaluation guided beam search for reasoning. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=Bw82hwg5Q3.

Zhenliang Xue, Yixin Song, Zeyu Mi, Le Chen, Yubin Xia, and Haibo Chen. Powerinfer-2: Fast large language model inference on a smartphone, 2024.

Kaiyu Yang, Aidan Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan Prenger, and Anima Anandkumar. LeanDojo: Theorem proving with retrieval-augmented language models. In *Neural Information Processing Systems (NeurIPS)*, 2023a.

Kevin Yang and Dan Klein. FUDGE: Controlled text generation with future discriminators. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 3511–3535. Association for Computational Linguistics, 2021. doi: 10.18653/v1/2021.naacl-main.276. URL https://aclanthology.org/2021.naacl-main.276.

Nan Yang, Tao Ge, Liang Wang, Binxing Jiao, Daxin Jiang, Linjun Yang, Rangan Majumder, and Furu Wei. Inference with reference: Lossless acceleration of large language models. *ArXiv preprint*, abs/2304.04487, 2023b. URL https://arxiv.org/abs/2304.04487.

Songlin Yang, Bailin Wang, Yikang Shen, Rameswar Panda, and Yoon Kim. Gated linear attention transformers with hardware-efficient training, 2024.

Zhilin Yang, Zihang Dai, Ruslan Salakhutdinov, and William W. Cohen. Breaking the softmax bottleneck: A high-rank RNN language model. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018. URL https://openreview.net/forum?id=HkwZSG-CZ.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik R Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=5Xc1ecxO1h.

Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for Transformer-Based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pp. 521–538. USENIX Association, 2022. ISBN 978-1-939133-28-1. URL https://www.usenix.org/conference/osdi22/presentation/yu.

Sina Zarrieß, Henrik Voigt, and Simeon Schüz. Decoding methods in neural language generation: A survey. *Information*, 12(9), 2021. ISSN 2078-2489. doi: 10.3390/info12090355. URL https://www.mdpi.com/2078-2489/12/9/355.

Eric Zelikman, Georges Harik, Yijia Shao, Varuna Jayasiri, Nick Haber, and Noah D. Goodman. Quiet-star: Language models can teach themselves to think before speaking, 2024a.

Eric Zelikman, Eliana Lorch, Lester Mackey, and Adam Tauman Kalai. Self-taught optimizer (stop): Recursively self-improving code generation, 2024b.

Hugh Zhang, Daniel Duckworth, Daphne Ippolito, and Arvind Neelakantan. Trading off diversity and quality in natural language generation. In *Proceedings of the Workshop on Human Evaluation of NLP Systems (HumEval)*, pp. 25–33. Association for Computational Linguistics, 2021. URL https://aclanthology.org/2021.humeval-1.3.

Michael Zhang, Kush S. Bhatia, Hermann Kumbong, and Christopher R'e. The hedgehog & the porcupine: Expressive linear attentions with softmax mimicry. *ArXiv preprint*, abs/2402.04347, 2024. URL https://arxiv.org/abs/2402.04347.

Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, Zhangyang Wang, and Beidi Chen. H_2o: Heavy-hitter oracle for efficient generative inference of large language models, 2023.

Stephen Zhao, Rob Brekelmans, Alireza Makhzani, and Roger Grosse. Probabilistic inference in language models via twisted sequential monte carlo, 2024a.

Yilong Zhao, Chien-Yu Lin, Kan Zhu, Zihao Ye, Lequn Chen, Size Zheng, Luis Ceze, Arvind Krishnamurthy, Tianqi Chen, and Baris Kasikci. Atom: Low-bit quantization for efficient and accurate llm serving, 2024b.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Jeff Huang, Chuyue Sun, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. Efficiently programming large language models using sglang, 2023.

Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. Distserve: Disaggregating prefill and decoding for goodput-optimized large language model serving, 2024.

Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Yonatan Bisk, Daniel Fried, Uri Alon, et al. Webarena: A realistic web environment for building autonomous agents. *ArXiv preprint*, abs/2307.13854, 2023. URL https://arxiv.org/abs/2307.13854.

Daniel M. Ziegler, Nisan Stiennon, Jeffrey Wu, Tom B. Brown, Alec Radford, Dario Amodei, Paul Christiano, and Geoffrey Irving. Fine-tuning language models from human preferences, 2020.

G. Zipf. The psycho-biology of language: An introduction to dynamic philology. 1999.

Zirui Liu, Jiayi Yuan, Hongye Jin, Shaochen Zhong, Zhaozhuo Xu, Vladimir Braverman, Beidi Chen, and Xia Hu. Kivi : Plug-and-play 2bit kv cache quantization with streaming asymmetric quantization. 2023. doi: 10.13140/RG.2.2.28167.37282. URL https://rgdoi.net/10.13140/RG.2.2.28167.37282.