

LLM Inference

How to generate text from LLMs?

Zdeněk Kasner

 12 Mar 2026




Charles
University



Charles University
Faculty of Mathematics and Physics
Institute of Formal and Applied Linguistics



unless otherwise stated

After today's lecture, you should be able to:

- Understand how to **generate text** with a Transformer-based language model.
- Explain differences between **decoding algorithms** and the role of decoding parameters.
- **Choose a suitable LLM** for your task.
- **Run a LLM** locally on your computer or computational cluster.

Text generation

Model stages:

random neural network



“autocomplete on steroids”

base / foundational model



assistant

instruction-tuned model



helpful assistant

Training stages:

1 Pre-training 🌐

↓ Prague is the capital of Czechia (...)

2 Instruction tuning 💬

user: What is the capital of Czechia?

↓ assistant: Prague

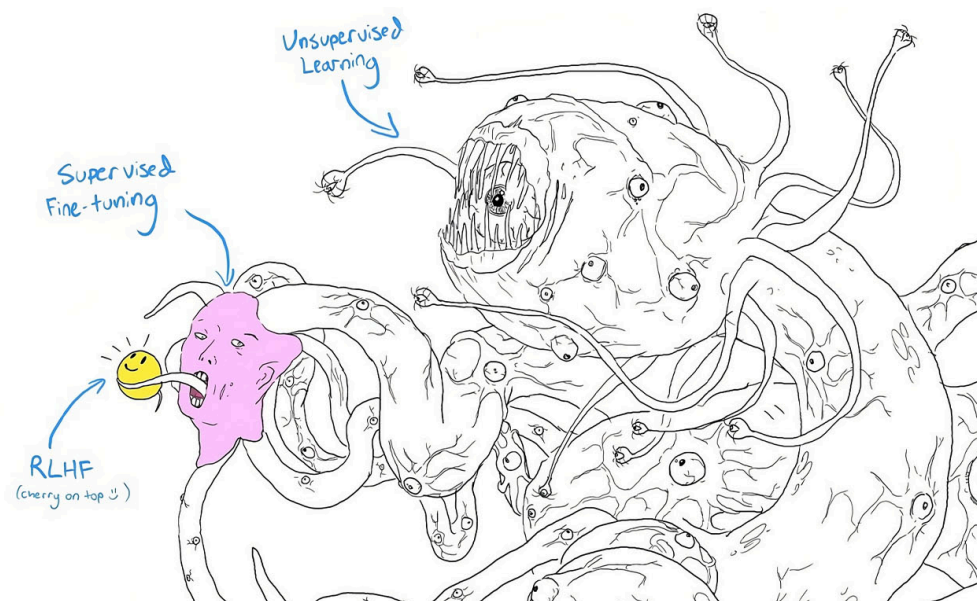
3 Human preference optimization 🧑🏫

user: What is the capital of Czechia?

answer #1: Prague.

answer #2: The capital of Czechia is Prague. It is the largest (...)

This lecture: **LLM inference** = we have a trained model and we want to **use it**.



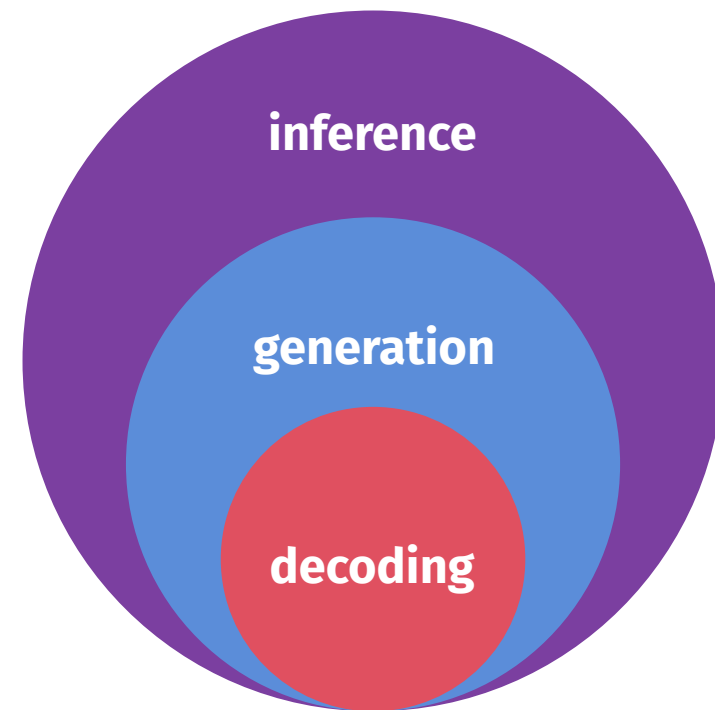
Question

What is the difference between inference, generation, and decoding?

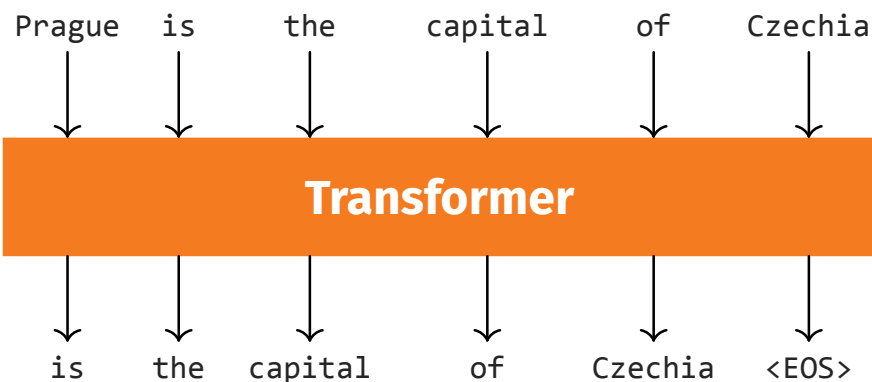
Inference – The concept of using a trained model for **making predictions** on new data (for classification, sequence tagging, text generation, ...).

Generation – The process of using a trained model for **producing a sequence of tokens**.

Decoding – The algorithm of **selecting the next token** using the model's internal representation.

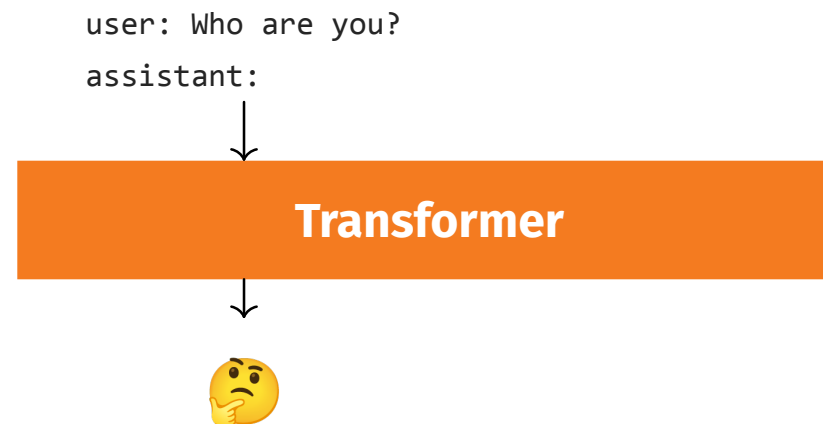


Training



Teacher forcing: We know what token should come next, so we use it to train the model.

Inference

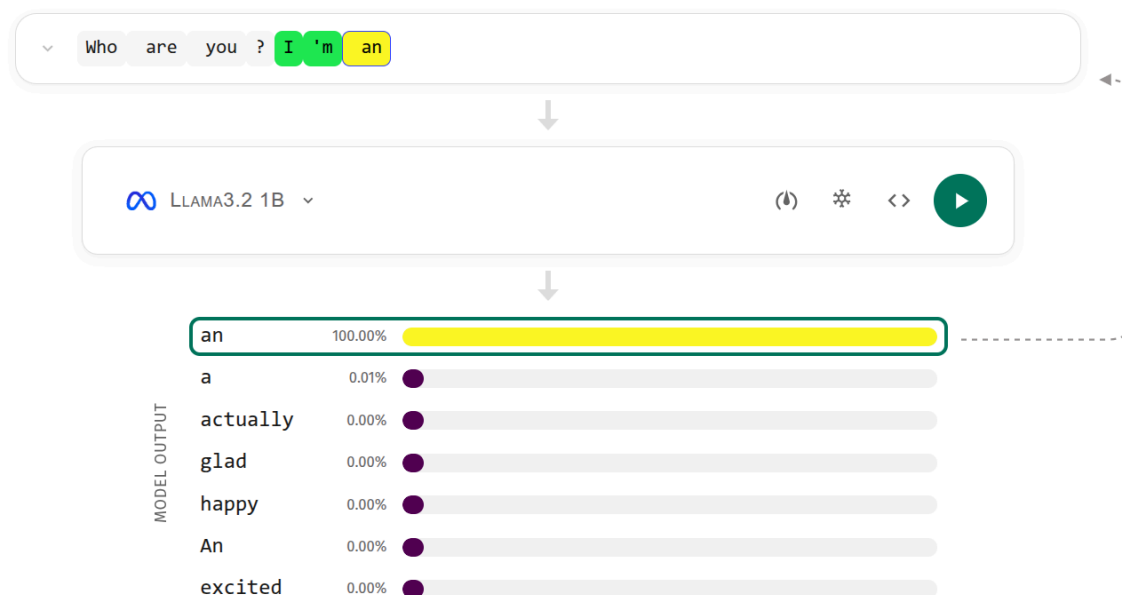


Decoding: We need to *select* what token should come next.

Recipe: How to generate text autoregressively

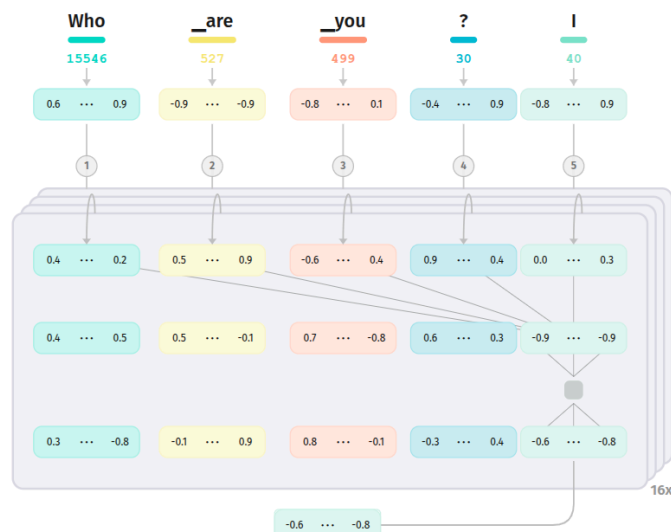
1. **Start with the context:** a sequence of tokens (a “prompt” if instruction-tuned).
2. **Feed the context** into the LLM.
3. **Select the next token** from the model-generated probability distribution.
4. **Append the selected token** to the sequence.
5. **Repeat** from (3) until the EOS (end-of-sequence) token is selected.

Inference for dummies 🐥



<https://animatedllm.github.io/generation-simple>

Advanced version



Tokenization

The text is split into smaller parts: tokens.

Input embeddings

Each token has its own vector representation.

Positional embeddings

Information about the token position is added to the representation.

Attention layer

Tokens share information with each other.

Feed-forward layer

The model updates information about each token

Last hidden state

Contains information about the next token.

<https://animatedllm.github.io/generation-model>

Hardcore version 🤘

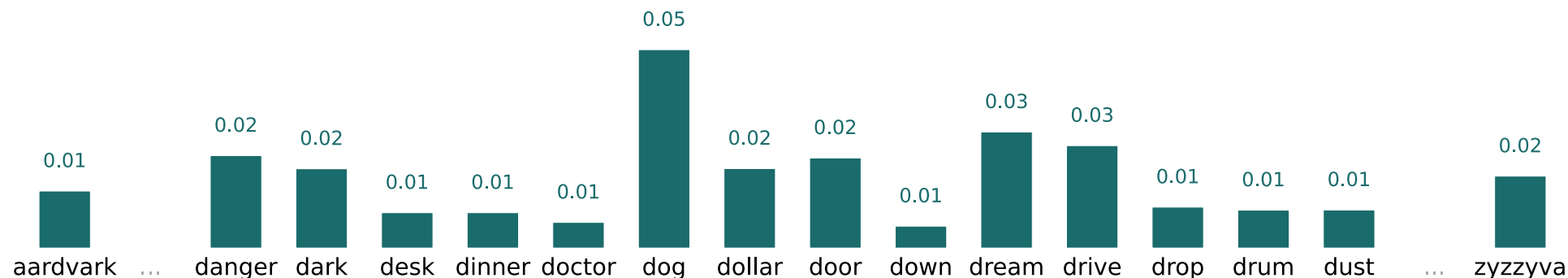
The screenshot shows the 'LLM Visualization' website interface. On the left, a 'Chapter: Overview' section contains a flowchart of the LLM architecture. It starts with 'How to predict text tokens' (with counts: 2537, 208, 4531, 2420, 16326, 2456) leading to 'tok embed', then 'pos embed', and a 'transformer 1' block. The transformer block includes 'layer norm', 'multi-head, causal self-attention', another 'layer norm', 'feed forward', a third 'layer norm', 'linear', and 'softmax'. On the right, a 'Table of Contents' lists: Intro, Introduction, Preliminaries, Components, Embedding, Layer Norm, Self Attention, Projection, MLP, Transformer, Softmax, and Output. Below the list are 'Continue' and 'Skip' buttons. The main area shows a 3D visualization of the 'nano-gpt' model with 85,584 parameters, with tabs for GPT-2 (small), nano-gpt, GPT-2 (XL), and GPT-3.

<https://bbycroft.net/llm>

Decoding algorithms

Decoding the next token

For each time step t , the decoder outputs **probability distribution** over the tokens given the previous context $P(y_t \mid y_{1:t-1}, X)$.



That is where the “job” of the Transformer decoder ends → it is up to us (or our decoding algorithm) to **use the distribution for decoding the next token.**

 **Holy grail:** Find the most probable continuation to our prompt:

$$y^* = \arg \max_{y \in \mathcal{Y}} P(y) = \arg \max_{y \in \mathcal{Y}} \prod_{i=1}^t P(y_i \mid y_1, \dots, y_{t-1})$$

Question

Why is this not possible in practice?

Intractable (exponential search space) \rightarrow we need to approximate it.

Question

And is it even our goal?

Two approaches we typically combine in practice:

**Approximating the most
probable sequence** 🤔

Greedy search / beam search

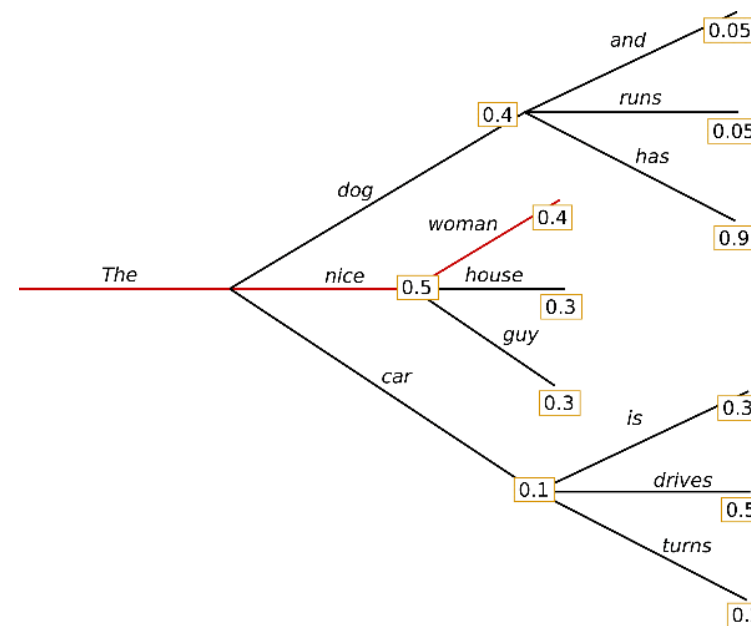
Adding stochasticity 🎲

Temperature / top-k sampling / nucleus
(top-p) sampling

Algorithm

In each step t , select the most probable token: $y_t = \arg \max_{y_t \in \mathcal{V}} P(y_t \mid y_1, \dots, y_{t-1})$

- Very fast, often works satisfactorily (especially with LLMs).
- Non-parametric (no hyperparameters to tune).
- But: may produce sequences that are too generic.

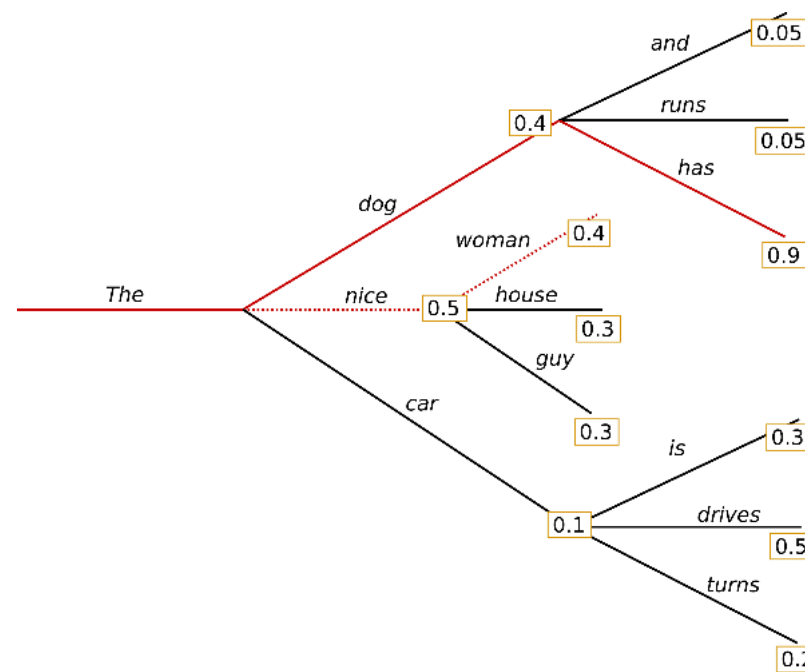


Algorithm

Parameter k : number of sequences (beams).

Each step t :

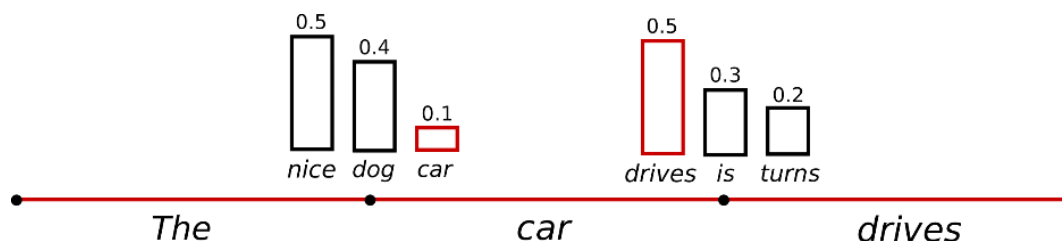
1. Extend the sequences from step $t - 1$ with all possible tokens.
2. Select the k most probable sequences for step $t + 1$.



- $k = 1 \rightarrow$ greedy decoding; larger $k \rightarrow$ slower, but better approximation.
- $k > 1$ allows re-ranking results.

Instead of picking the most probable token, we can randomly sample the next token y_t according to its conditional probability distribution:

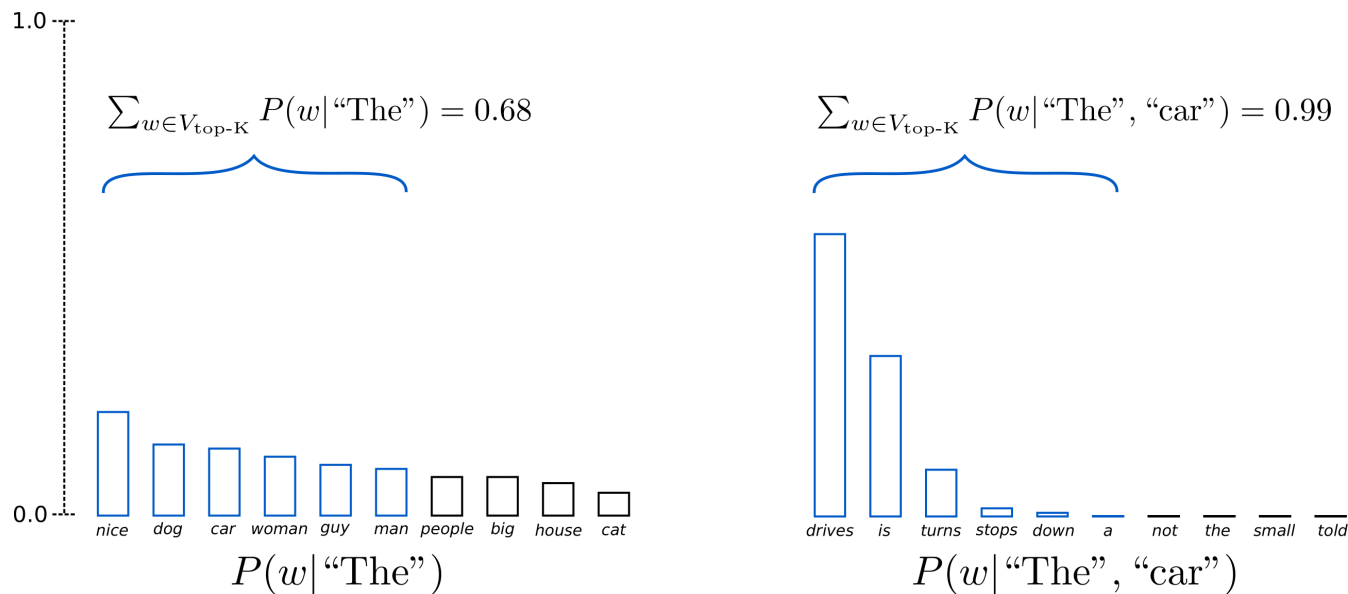
$$y_t \sim P(y_t \mid y_1, \dots, y_{t-1})$$



Question

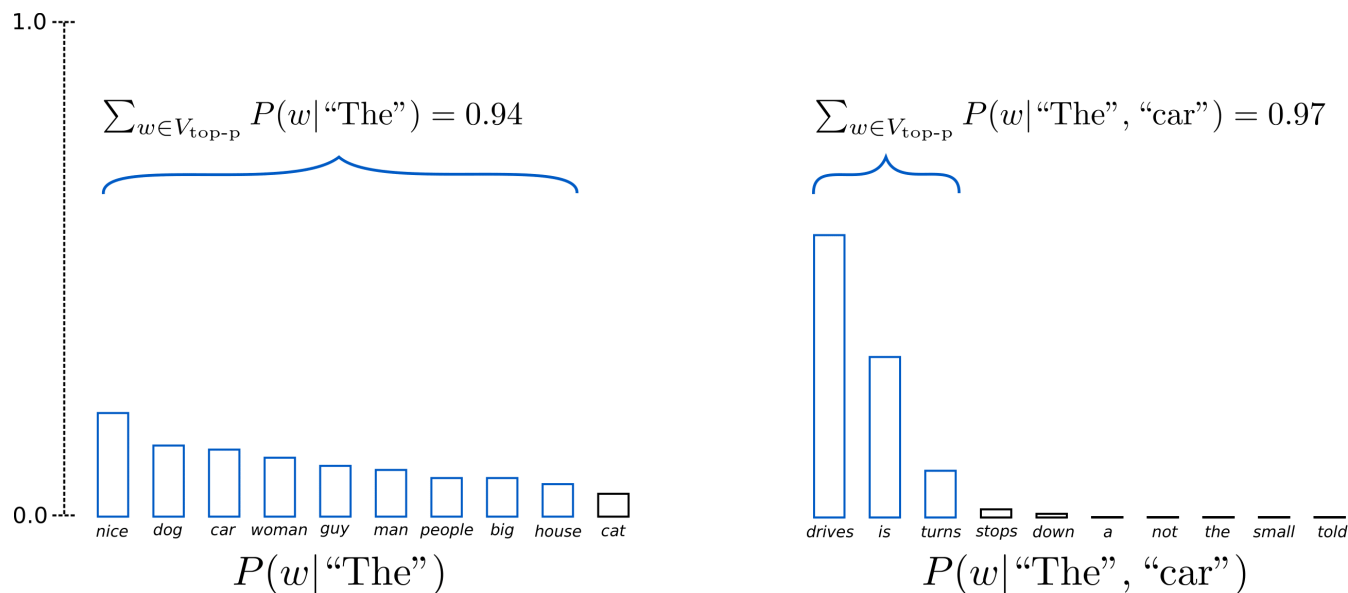
Why is this sampling from the entire vocabulary not a good idea?

Selecting the token in each step randomly from $k \in \{1, \dots, |\mathcal{V}|\}$ most probable tokens.
The truncated distribution is re-weighted using softmax.



Sampling from the **nucleus**: set of the most probable tokens with combined probability summing to $p \in (0, 1]$.

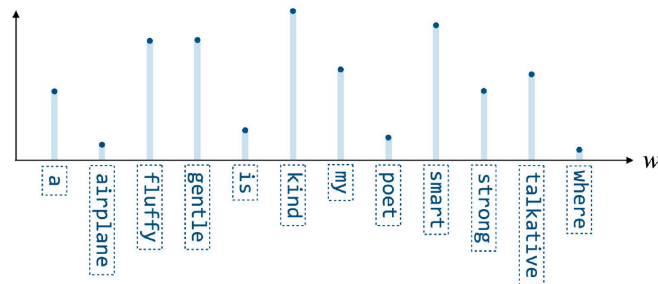
Similar to top-k, but with a **variable** k in each step.



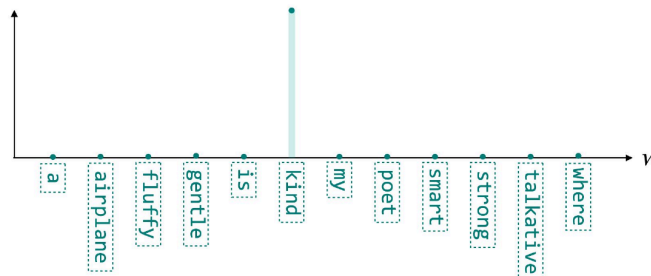
The shape of the output distribution can be adjusted using the **temperature** T :

$$\text{softmax}(z) = \frac{\exp\left(\frac{z_t}{T}\right)}{\sum_j \exp\left(\frac{z_j}{T}\right)}$$

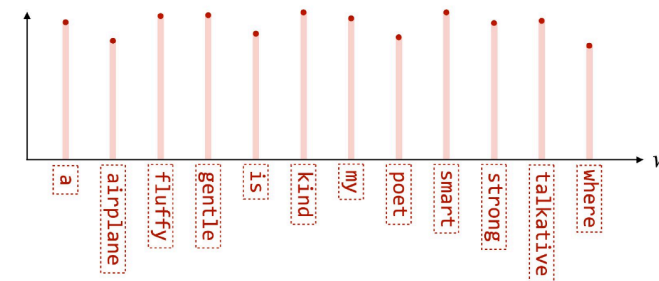
$T = 1$: **original distribution**






$T < 1$: **more peaked**
($T = 0 \rightarrow$ greedy decoding)



$T > 1$: **more uniformly random**



  **r/MachineLearning** · 8 mo. ago
zyl1024 






[D] What happened to "creative" decoding strategy?

Discussion

For GPT-2 and most models at that time, the naive greedy decoding is extremely prone to generating repetitive and nonsensical outputs very fast, and many techniques, such as top-p sampling, nucleus sampling, repetition penalty, n-gram penalty, etc. are needed. (e.g. <https://arxiv.org/pdf/1904.09751>)

For recent LLMs, I haven't been using any of these tricks, and instead, any temperature between 0 and 1 seems to work just fine. The only repetitive generation that I've observed seem to be in math reasoning, when the model wants to do some exhaustive search that didn't succeed.

So are all these custom decoding strategies a thing of the past, and we don't need to worry about degenerate content generation anymore?

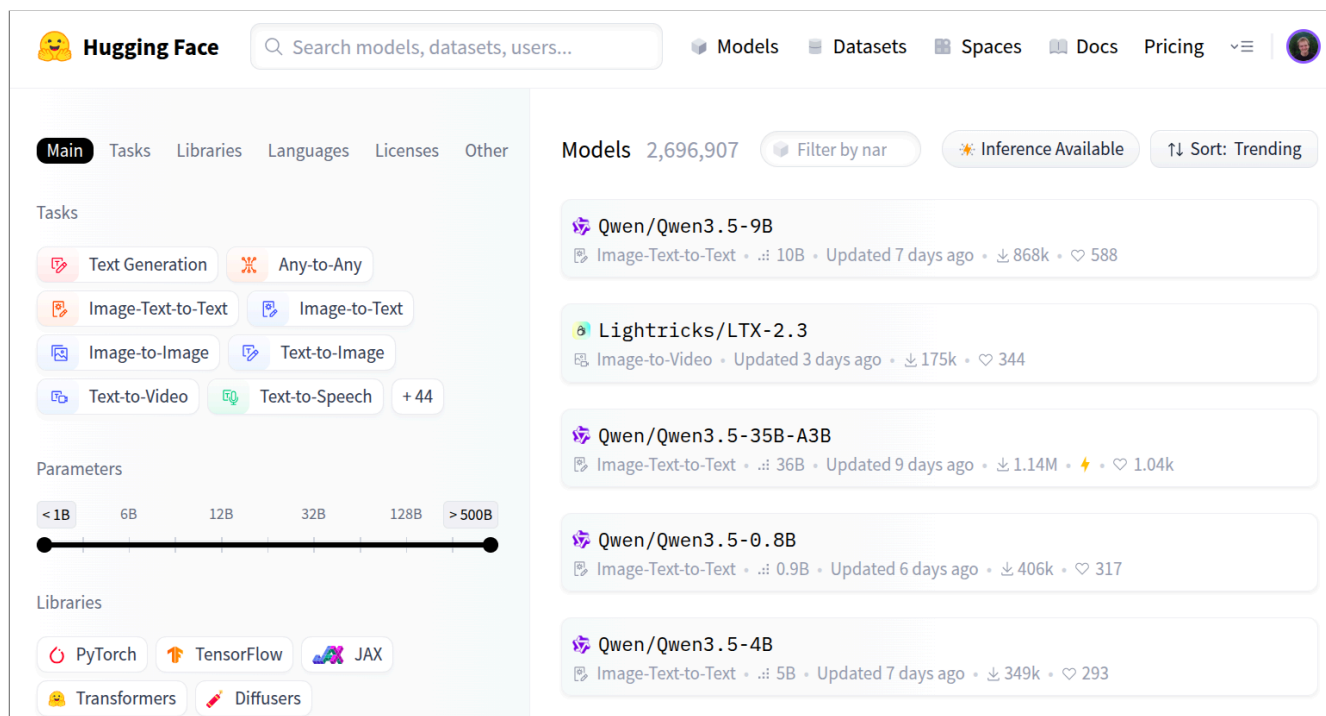
 23   12   Share

[source: Reddit](#)

Navigating the LLM zoo

HuggingFace: the largest repository of open LLMs.

As of March 2026, it contains 2.7M models (many of these are derivatives).



Artificial Analysis LLM Leaderboard: rating LLMs across many dimensions (context, window size, price, speed, performance...).

Model performance is grouped under a single “Intelligence index” (=combined score from 10 benchmarks).

Features		Intelligence		Price	Speed	Latency		Further Analysis	
Model	Context Window	Creator	Artificial Analysis Intelligence Index	Blended USD/1M Tokens	Median Tokens/s	Latency First Answer Chunk (s)		Model	Providers
Gemini 3.1 Pro Preview	1m	Google	57	\$4.50	119	33.59		Model	Providers
GPT-5.4 (xhigh)	1m	OpenAI	57	\$5.63	78	184.99		Model	Providers
GPT-5.3 Codex (xhigh)	400k	OpenAI	54	\$4.81	65	96.73		Model	Providers
Claude Opus 4.6 (max)	200k	Anthropic	53	\$10.00	46	15.33		Model	Providers
Claude Sonnet 4.6 (max)	200k	Anthropic	52	\$6.00	47	100.63		Model	Providers
GPT-5.2 (xhigh)	400k	OpenAI	51	\$4.81	67	81.58		Model	Providers
GLM-5	200k	Z AI	50	\$1.55	61	1.54		Model	Providers
GPT-5.2 Codex (xhigh)	400k	OpenAI	49	\$4.81	92	57.75		Model	Providers

Arena.ai: Elo rating of LLMs: for a pair of answers from different models, users decide which is better.

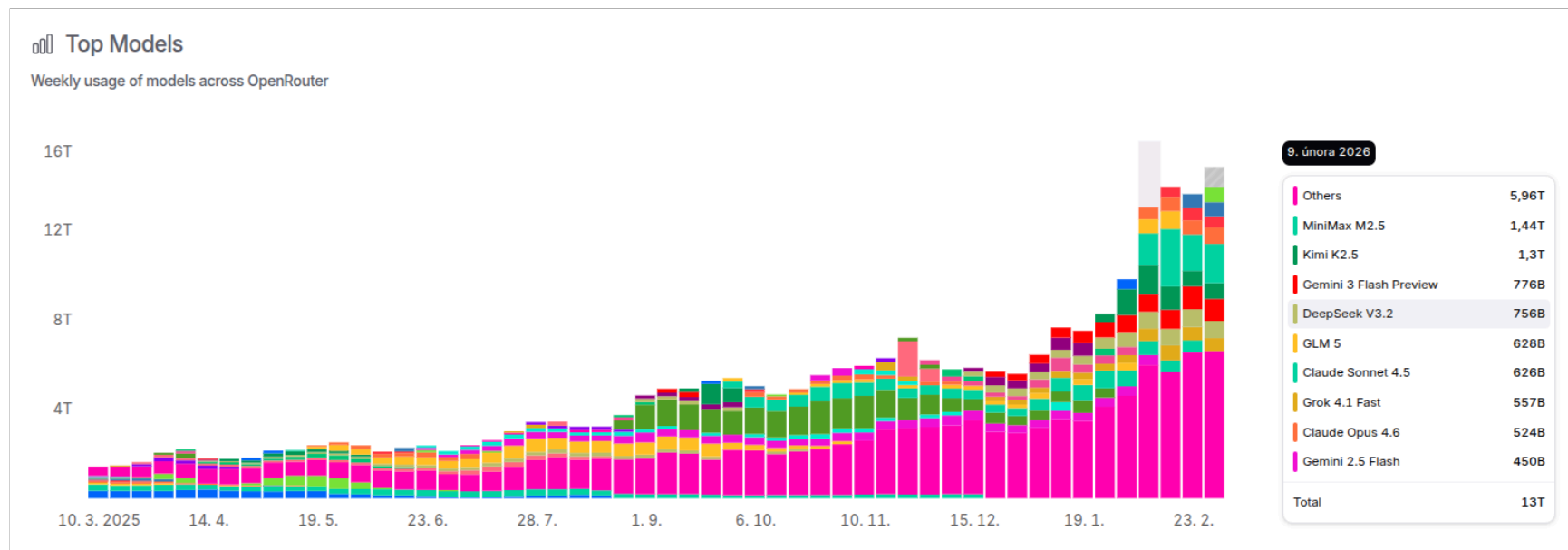
Leaderboard Overview

See how leading AI models stack up across text, image, vision, and more. This page provides a high-level snapshot of each Arena. Explore dedicated tabs for deeper insights. Learn more [here](#).

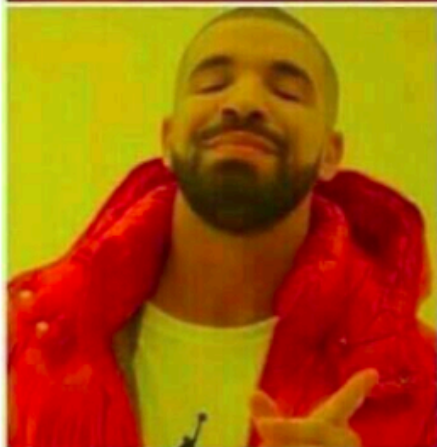
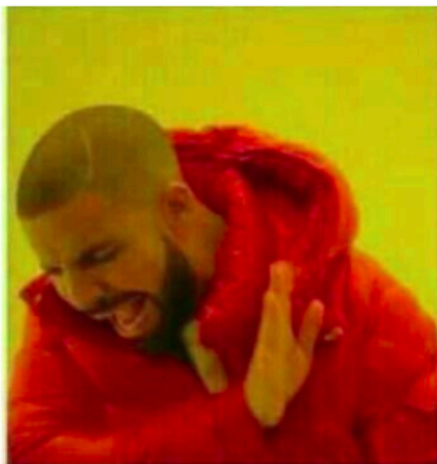
Text (4 days ago)			
Rank	Model	Score	Votes
1	Al claude-opus-4-6-...	1503	6 583
2	Al claude-opus-4-6	1503	7 454
3	G gemini-3.1-pro-p...	1500	4 052
4	XI grok-4.20-beta1	1495	3 818
5	G gemini-3-pro	1486	38 248
6	GPT-5.2-chat-lat...	1481	3 605
7	G gemini-3-flash	1473	29 334

Code (6 days ago)			
Rank	Model	Score	Votes
1	Al claude-opus-4-6	1560	2 845
2	Al claude-opus-4-6-...	1553	2 182
3	Al claude-sonnet-4-6	1531	1 839
4	Al claude-opus-4-5-...	1499	11 149
5	GPT-5.2-high	1471	1 696
6	Al claude-opus-4-5-...	1471	11 239
7	G gemini-3.1-pro-p...	1461	1 826

OpenRouter: routing traffic to various LLM providers, tracks real model usage through their proxy.



Deploying LLMs



Proprietary APIs

- OpenAI (ChatGPT, GPT-4o)
- Anthropic (Claude)
- Google (Gemini)
- ...

✓ Easy to use, no hardware needed

✗ Paid, no control over the model

Open models (local)

- Meta (Llama)
- Mistral
- Qwen
- ...

✓ Free, full control

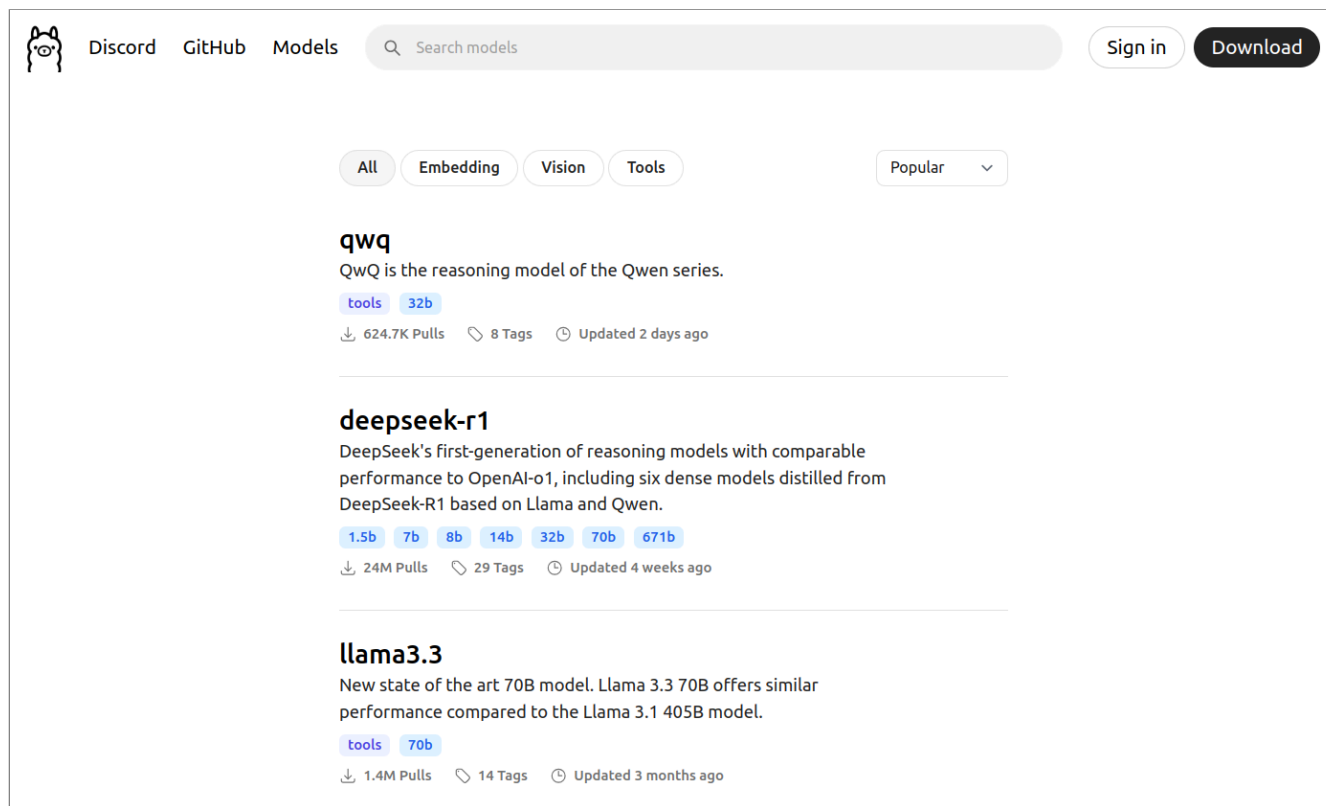
✗ Requires hardware (GPU)

HuggingFace transformers: Python library for loading models from the HuggingFace model repository.

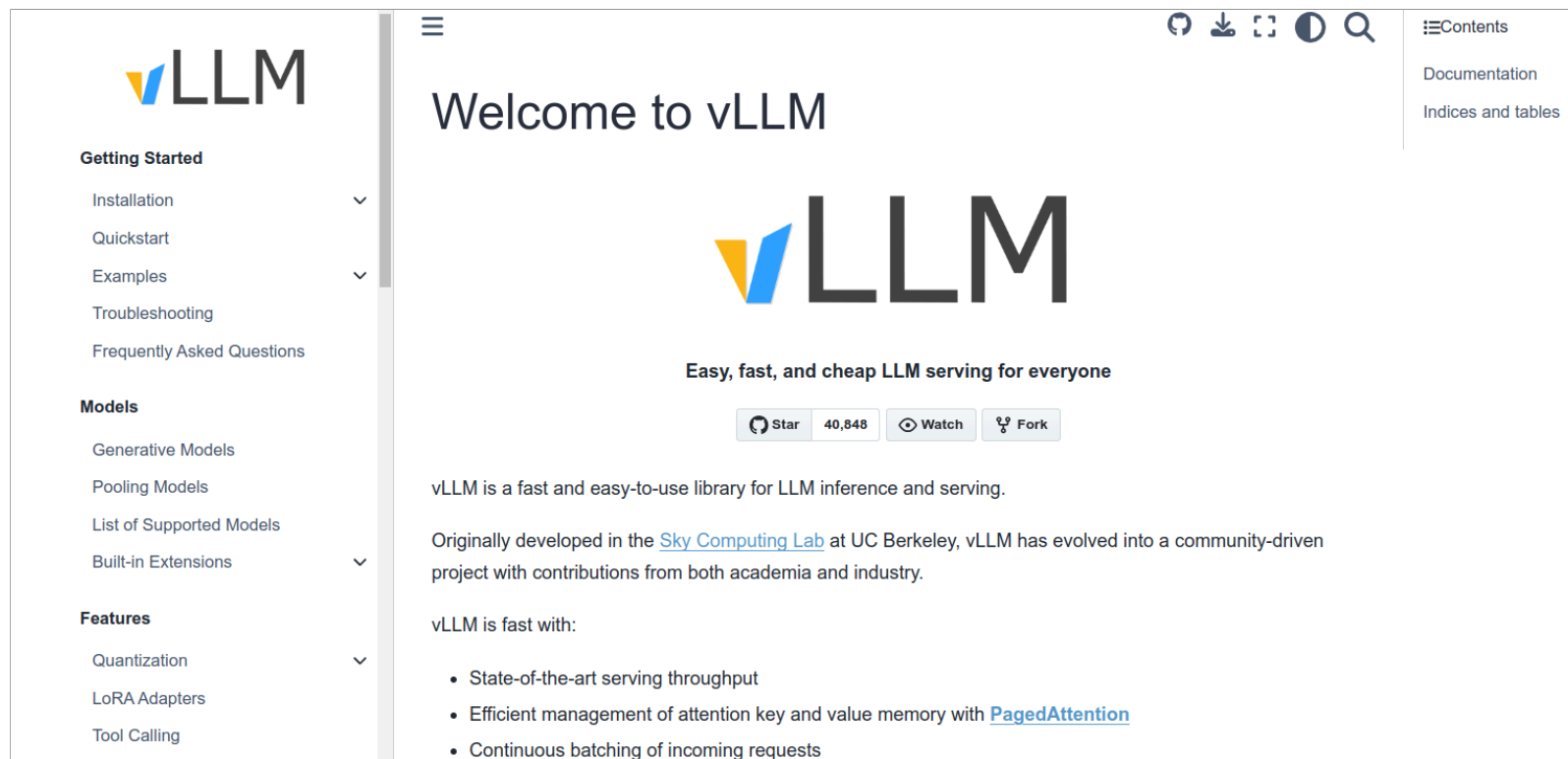


The screenshot shows the top of the HuggingFace Transformers repository page. It features the Transformers logo (a yellow smiley face with hands) and the word "Transformers" in a large, bold, black font. Below the logo, there is a row of status indicators: "build failing" (red), "license Apache-2.0" (blue), "website online" (green), "release v4.49.0" (blue), "Contributor Covenant v2.0 adopted" (pink), and "DOI 10.5281/zenodo.7391177" (blue). Underneath these indicators, there is a list of languages: English | 简体中文 | 繁體中文 | 한국어 | Español | 日本語 | हिन्दी | Русский | Português | తెలుగు | Français | Deutsch | Tiếng Việt | اردو | العربية. At the bottom of the screenshot, the text "State-of-the-art Machine Learning for JAX, PyTorch and TensorFlow" is displayed.

Ollama: running LLMs locally, easy to use, focus on quantized models.



vLLM: efficient library for serving of LLMs at scale.



The screenshot shows the GitHub repository page for vLLM. On the left is a navigation sidebar with sections: Getting Started (Installation, Quickstart, Examples, Troubleshooting, Frequently Asked Questions), Models (Generative Models, Pooling Models, List of Supported Models, Built-in Extensions), and Features (Quantization, LoRA Adapters, Tool Calling). The main content area has a large 'Welcome to vLLM' heading, the vLLM logo, and the tagline 'Easy, fast, and cheap LLM serving for everyone'. Below this are GitHub statistics: 40,848 stars, a Watch button, and a Fork button. The text describes vLLM as a fast and easy-to-use library for LLM inference and serving, originally developed at UC Berkeley. It lists three key features: state-of-the-art serving throughput, efficient memory management with PagedAttention, and continuous batching of requests.

Demo time 

 [HuggingFace LLM tutorial](#)

Summary

- **Terms:**
 - **inference** = using a trained model for predictions
 - **generation** = producing a sequence of tokens
 - **decoding** = selecting the next token.
- **Greedy decoding** is the simplest approach
- **Beam search** improves it by keeping k hypotheses.
- **Stochastic methods** (top-k, top-p, temperature) add randomness to avoid repetitive and dull outputs.
- Open LLMs can be run locally using **HuggingFace transformers, Ollama, or vLLM.**

- [HuggingFace models](#)
- [Awesome LLM: curated list of resources](#)
- [Transformer inference: 3D visualization](#)
- [HuggingFace decoding algorithms overview](#)
- [HuggingFace text generation strategies](#)
- [Common pitfalls when generating text with LLMs](#)
- [Visualizing decoding strategies](#)
- [Minimum Bayes Risk decoding](#)

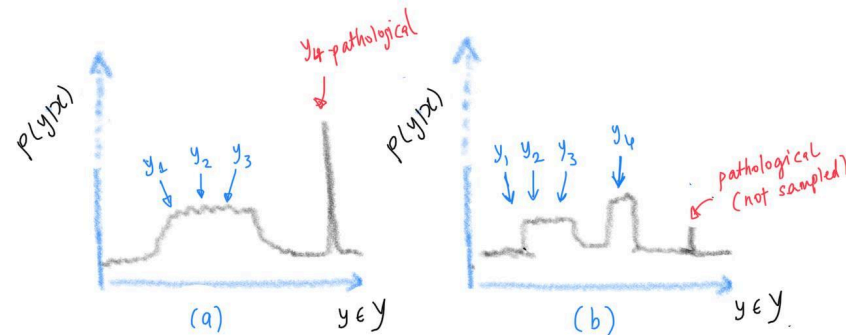
- [On NMT Search Errors and Model Errors: Cat Got Your Tongue? \(Stahlberg and Byrne, 2019\)](#) – what happens if one manages to approximate exact inference
- [On decoding strategies for neural text generators \(Wiher et al., 2022\)](#) – language generation tasks vs. decoding strategies.
- [If beam search is the answer, what was the question? \(Meister et al., 2020\)](#) – why does beam search work so well?
- [Understanding the properties of Minimum Bayes Risk decoding in neural machine translation \(Müller and Sennrich, 2021\)](#) – when can MBR be useful?

Bonus: extra decoding algorithms

Selecting the sequence **most similar to other sequences** = “consensus decoding”:

$$y^* = \arg \max_{y_k \in \mathcal{Y}} \sum_{y_l \in \mathcal{Y} \setminus \{y_k\}} \text{sim}(y_k, y_l)$$

- Useful for minimizing pathological behavior.
- Intractable → we need a sampling algorithm.
- Application in ASR and machine translation.



Aims to eliminate **repetition and incoherent text** in stochastic algorithms.

Adapting the k parameter based on the desired text perplexity.

Parameters:

- τ – the target perplexity
- η – learning rate

Algorithm 1: Adaptive top- k sampling for perplexity control

Target cross entropy τ , maximum cross entropy $\mu = 2 * \tau$, learning rate η

while *more words are to be generated* **do**

 Compute \hat{s} from (40): $\frac{\sum_{i=1}^{N-1} t_i b_i}{\sum_{i=1}^{N-1} t_i^2}$

 Compute k from (41): $k = \left(\frac{\hat{\epsilon} 2^\mu}{1 - N - \hat{\epsilon}} \right)^{\frac{1}{\hat{s}}}$

 Sample the next word X using top- k sampling

 Compute error: $e = \mathfrak{G}(X) - \tau$

 Update μ : $\mu = \mu - \eta * e$

end

Mirostat

“mirum” = surprise, “stat” = control

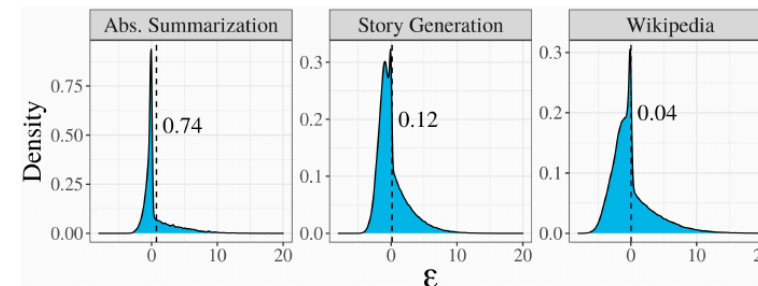
Similar to Mirostat, but **dynamic**: the perplexity is not pre-specified.

Ensures that in each decoding step, text perplexity is **close to the model perplexity**.

Example: coin toss

$$p(H) = 0.75, p(T) = 0.25:$$

- $HHHH \rightarrow$ most probable sequence
- $HTHH \rightarrow$ typical sequence



Originates from the information theory: **typical messages** are the messages that we would expect from the process.