



NI-NLM – Lecture 4

LLM inference

Zdeněk Kasner

 10 Mar 2026

From training to text generation

Recap: training

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 🧑

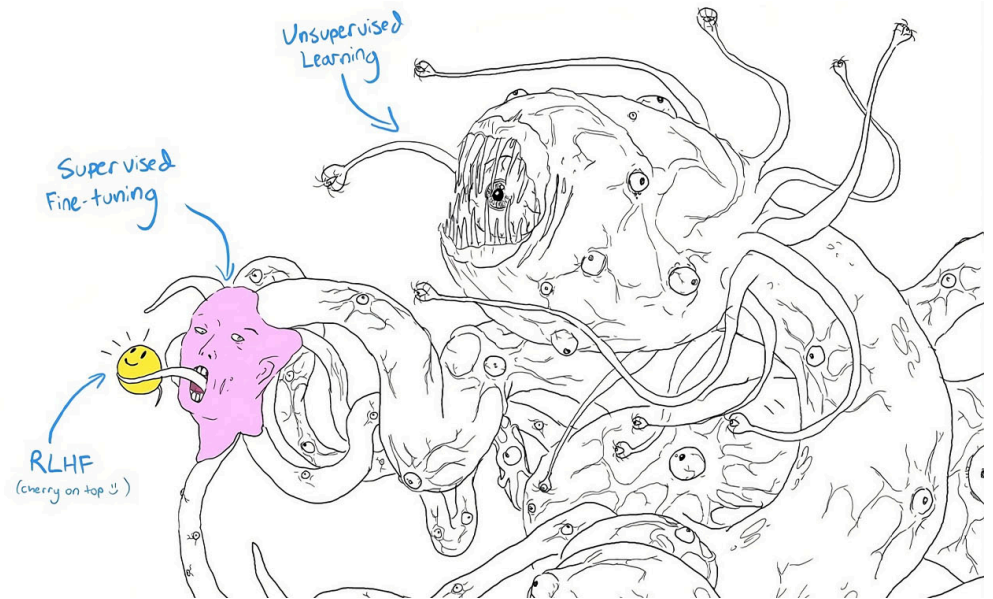
user: What is the capital of Czechia?

answer #1: Prague.

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

LLM inference

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



Question

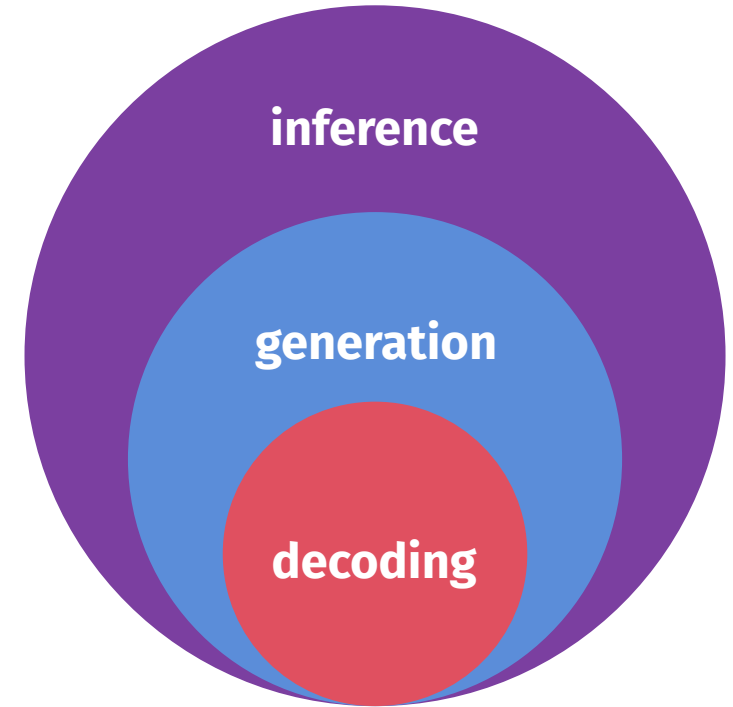
What is the difference between inference, generation, and decoding?

LLM Inference

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

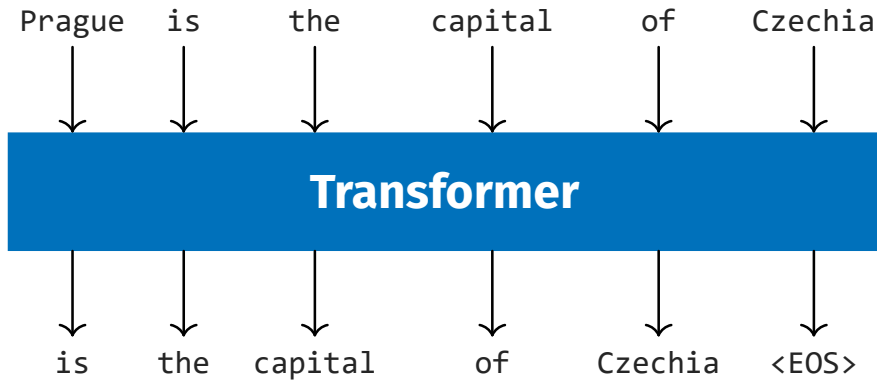
Generation – The process of using a trained model for **producing a sequence of tokens**.

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



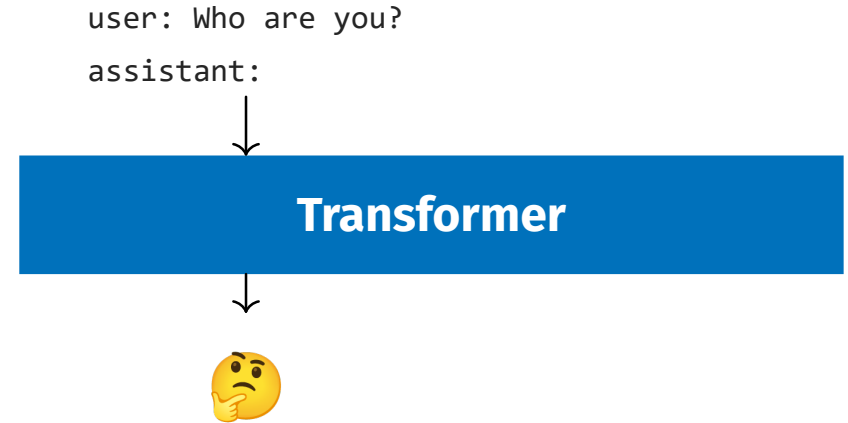
Training vs. inference

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.

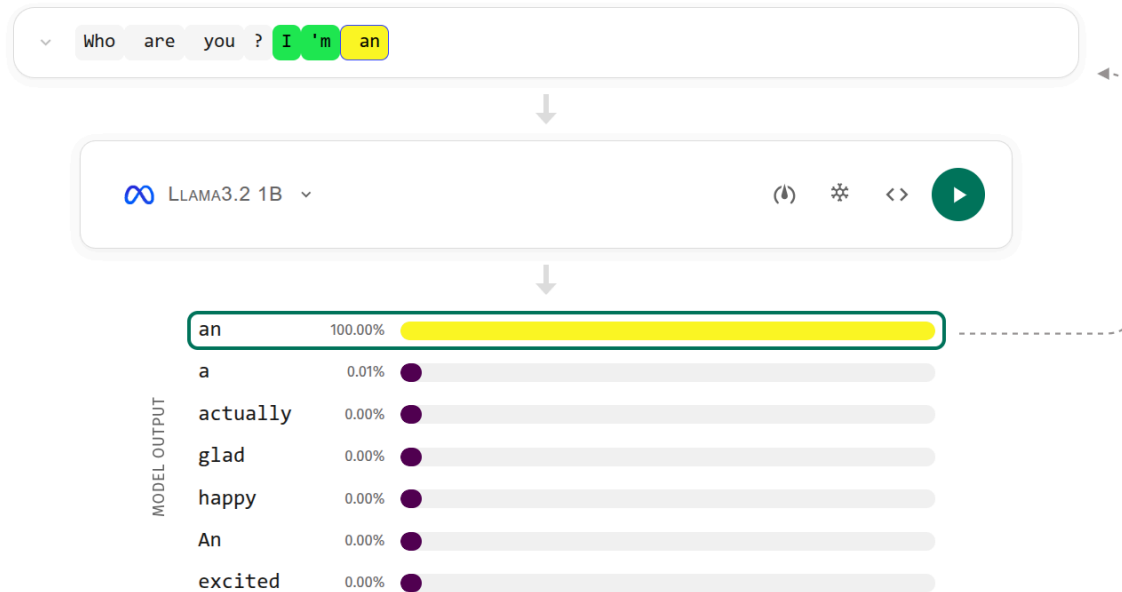


Recipe: How to generate text autoregressively

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.

What happens during LLM inference?

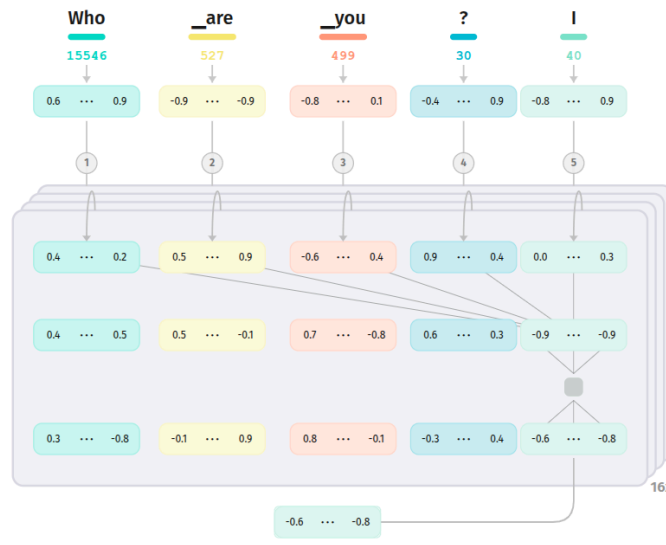
Inference for dummies 🐥



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

What happens during LLM inference?

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>

What happens during LLM inference?

Hardcore version 🤘

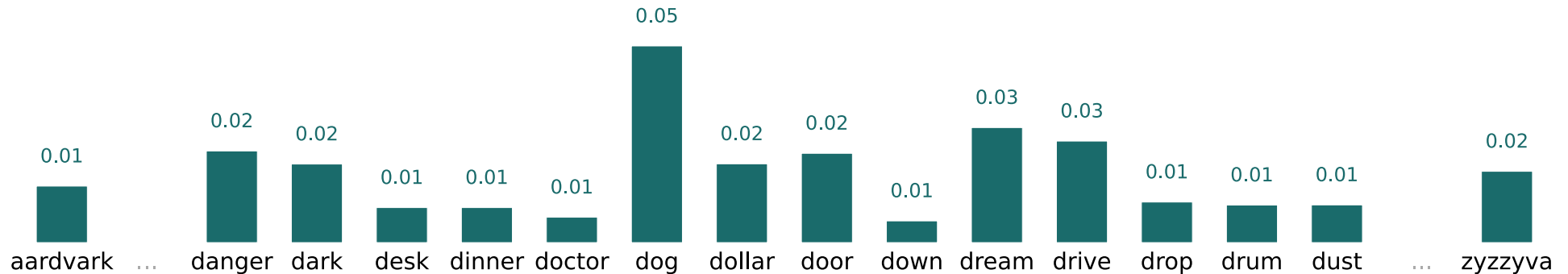
The screenshot displays the 'LLM Visualization' website interface. At the top, there are navigation buttons for 'GPT-2 (small)', 'nano-gpt' (selected), 'GPT-2 (XL)', and 'GPT-3'. Below this, a search bar and a 'nano-gpt' label with '85,584' parameters are visible. The main content area is split into two panels. The left panel, titled 'Chapter: Overview', shows a flowchart of the LLM architecture. It starts with 'How to predict text' (tokens and words) leading to 'tok embed' and 'pos embed'. These are combined and fed into 'transformer i', which consists of 'layer norm', 'multi-head, causal self-attention', 'layer norm', 'feed forward', 'layer norm', 'linear', and 'softmax' layers. The right panel shows a 3D visualization of the model's internal structure, with a 'nano-gpt' label and a search bar. A 'Table of Contents' is visible on the right side of the left panel, listing sections like 'Introduction', 'Preliminaries', 'Components', 'Embedding', 'Layer Norm', 'Self Attention', 'Projection', 'MLP', 'Transformer', 'Softmax', and 'Output'.

<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.**

Exact inference

 **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

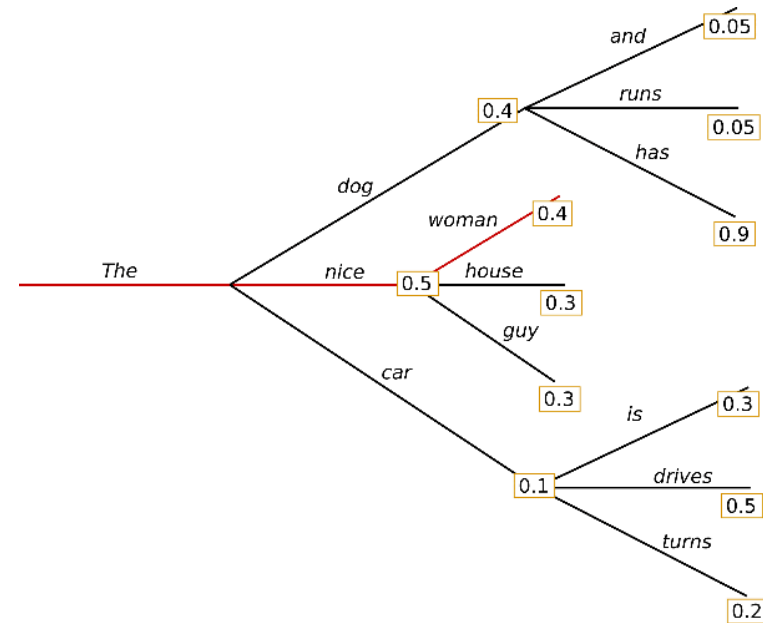
Adding stochasticity 🎲

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.

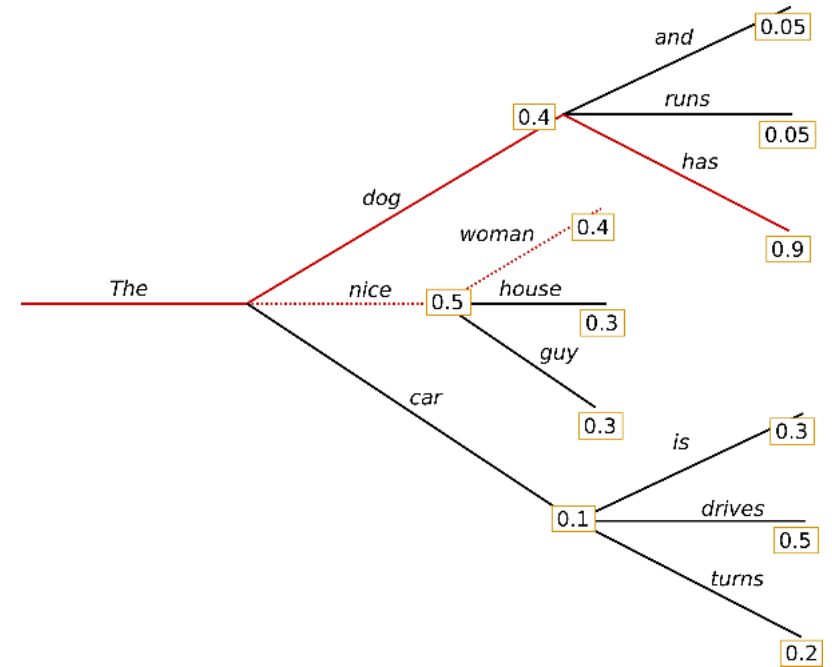


Algorithm

Parameter k : number of sequences (beams).

Each step t :

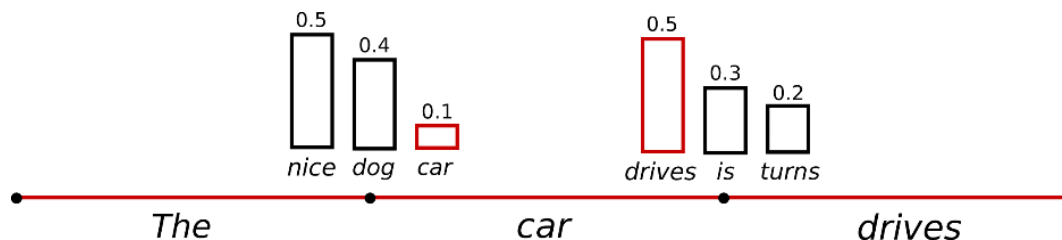
1. Extend the sequences from step $t - 1$ with all possible tokens.
2. Select the k most probable sequences for step $t + 1$.



- $k = 1 \rightarrow$ 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:

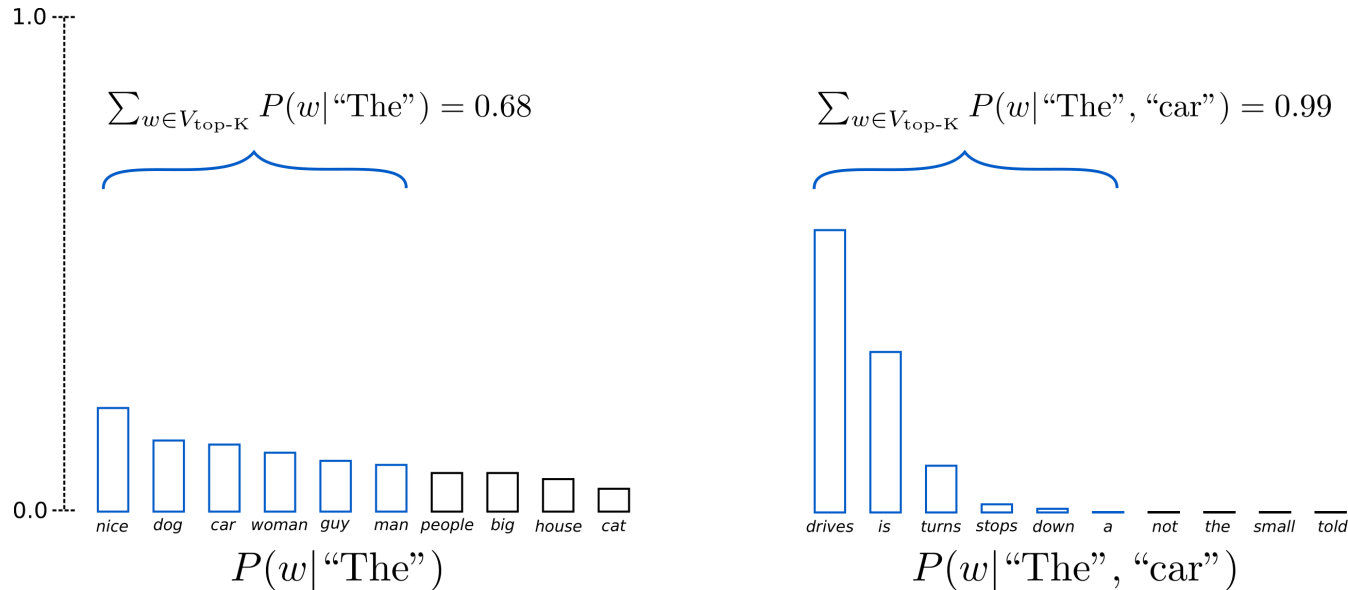
$$y_t \sim P(y_t \mid y_1, \dots, y_{t-1})$$



Question

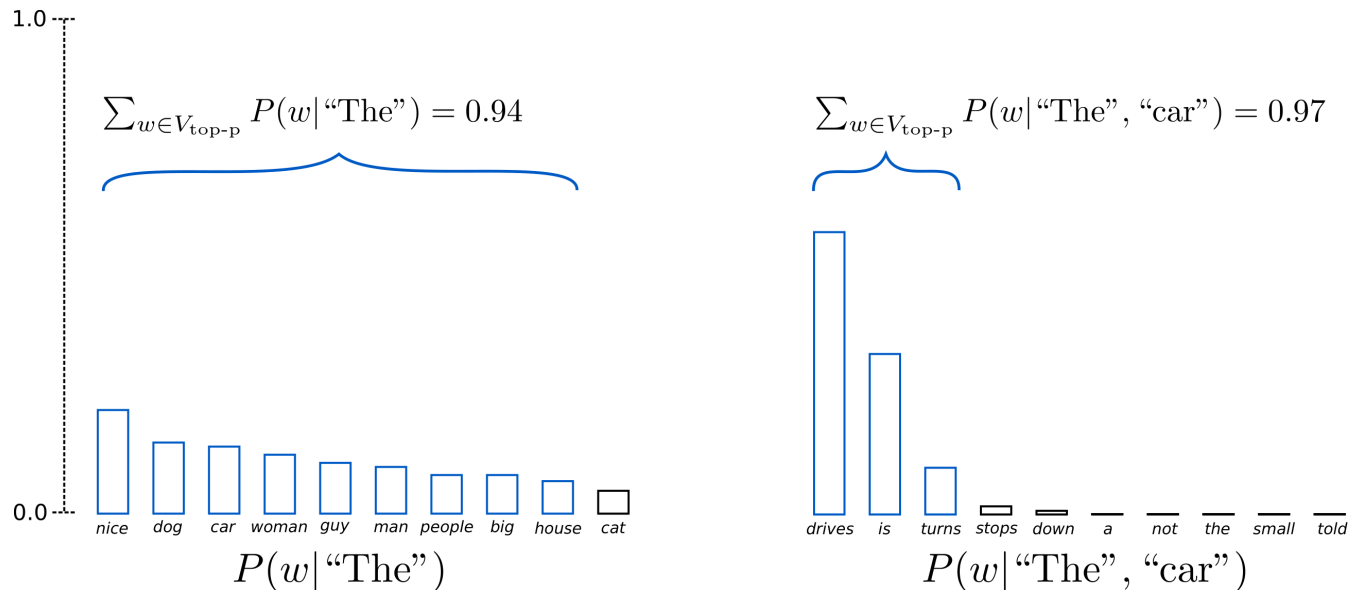
Why is this sampling from the entire vocabulary not a good idea?

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 $p \in (0, 1]$.

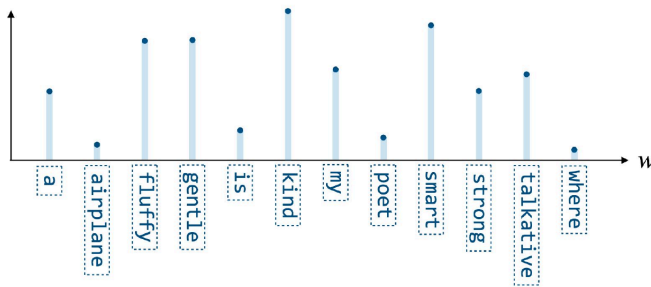
Similar to top-k, but with a **variable** k in each step.



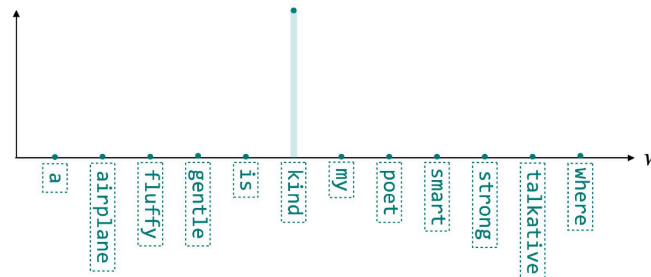
The shape of the output distribution can be adjusted using the **temperature** T :

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

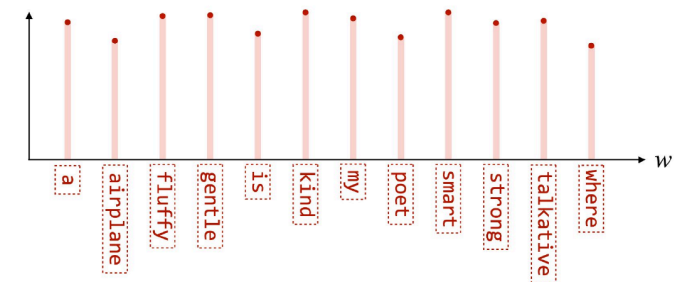
$T = 1$: original distribution





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



$T > 1$: more uniformly random



Is greediness all you need?

  **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.

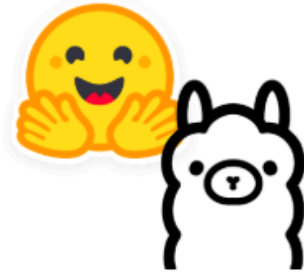
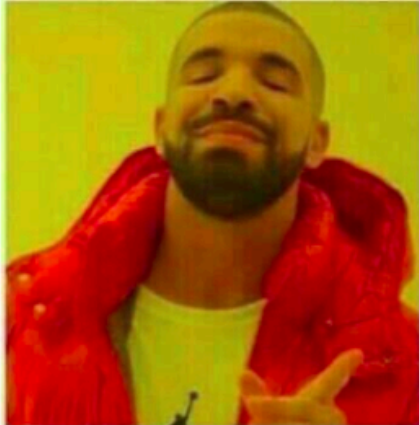
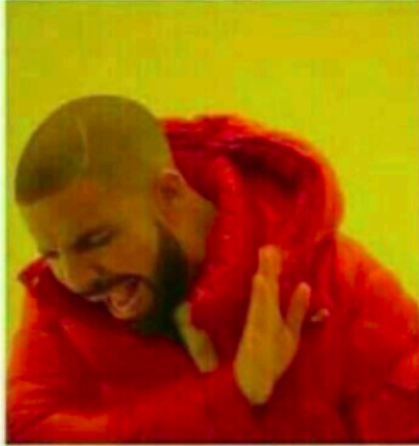
So are all these custom decoding strategies a thing of the past, and we don't need to worry about degenerate content generation anymore?

 23   12   Share

[source: Reddit](#)

Running LLMs locally

How to use LLMs



How to use 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

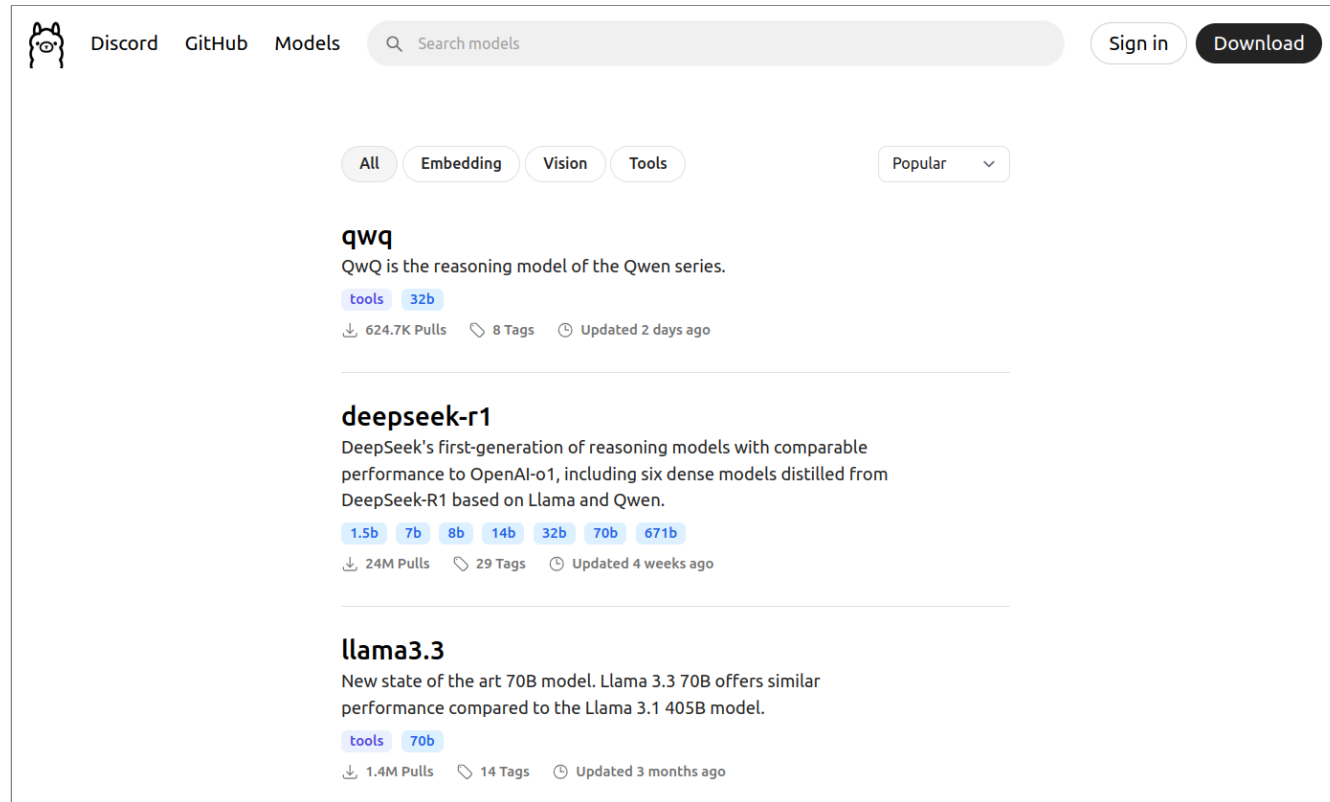
✗ Requires hardware (GPU)

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



The screenshot shows the top section of the HuggingFace Transformers website. At the top center is a yellow smiley face emoji with its hands raised, followed by the word "Transformers" in a large, bold, black font. Below this is a horizontal row of seven status boxes: "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 boxes is a list of language links: "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 in a bold, black font.

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



vLLM: efficient library for serving of LLMs at scale.

The screenshot shows the vLLM documentation website. On the left is a navigation sidebar with the following sections:

- Getting Started**
 - Installation
 - Quickstart
 - Examples
 - Troubleshooting
 - Frequently Asked Questions
- Models**
 - Generative Models
 - Pooling Models
 - List of Supported Models
 - Built-in Extensions
- Features**
 - Quantization
 - LoRA Adapters
 - Tool Calling

The main content area has a header with the vLLM logo and the text "Welcome to vLLM". Below this is a large vLLM logo and the tagline "Easy, fast, and cheap LLM serving for everyone". There are three GitHub buttons: "Star 40,848", "Watch", and "Fork".

The main text reads: "vLLM is a fast and easy-to-use library for LLM inference and serving. Originally developed in the [Sky Computing Lab](#) at UC Berkeley, vLLM has evolved into a community-driven project with contributions from both academia and industry."

Below this, it states "vLLM is fast with:" followed by a bulleted list:

- State-of-the-art serving throughput
- Efficient management of attention key and value memory with [PagedAttention](#)
- Continuous batching of incoming requests

Hands-on session

(continue until 14:50)

<http://tiny.cc/nlm-gen>

Summary

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.**

Links and resources

- [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](#)

Further reading

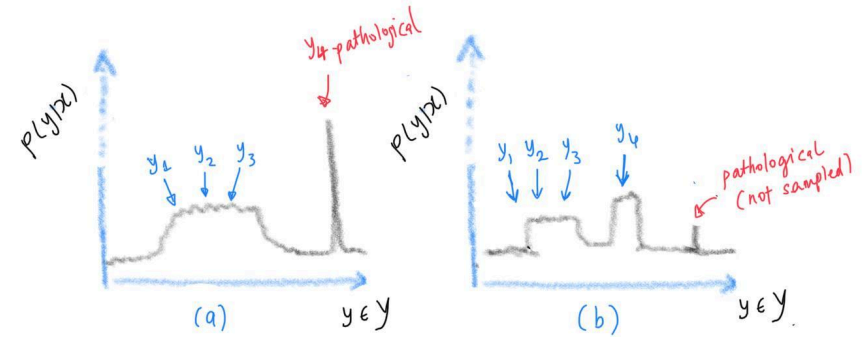
- [On NMT Search Errors and Model Errors: Cat Got Your Tongue? \(Stahlberg and Byrne, 2019\)](#) – what happens if one manages to approximate exact inference
- [On decoding strategies for neural text generators \(Wiher et al., 2022\)](#) – language generation tasks vs. decoding strategies.
- [If beam 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”:

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

- 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): $k = \left(\frac{\hat{\epsilon} 2^\mu}{1 - N^{-\hat{\epsilon}}} \right)^{\frac{1}{8}}$

 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

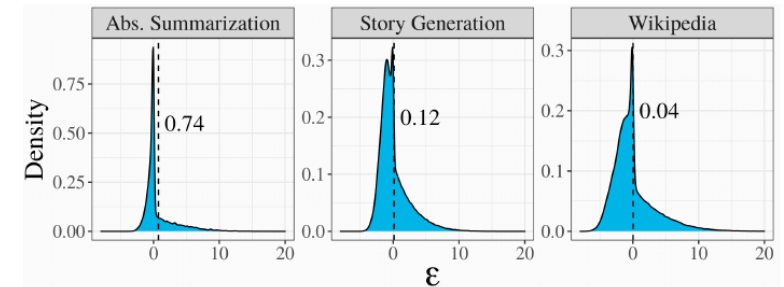
Similar to Mirostat, but **dynamic**: the perplexity is not pre-specified.

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.