LLM Reasoning

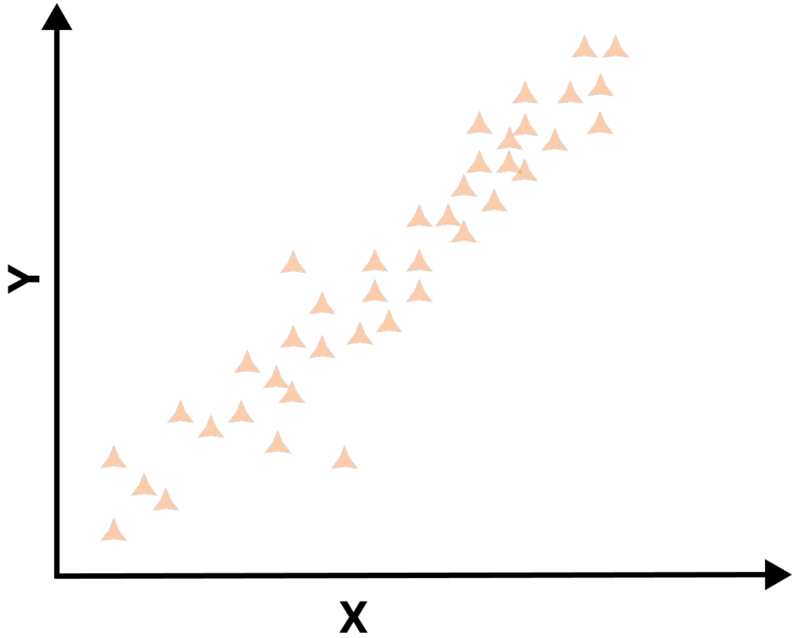
Course 1: Sampling Methods

Yaya Sy

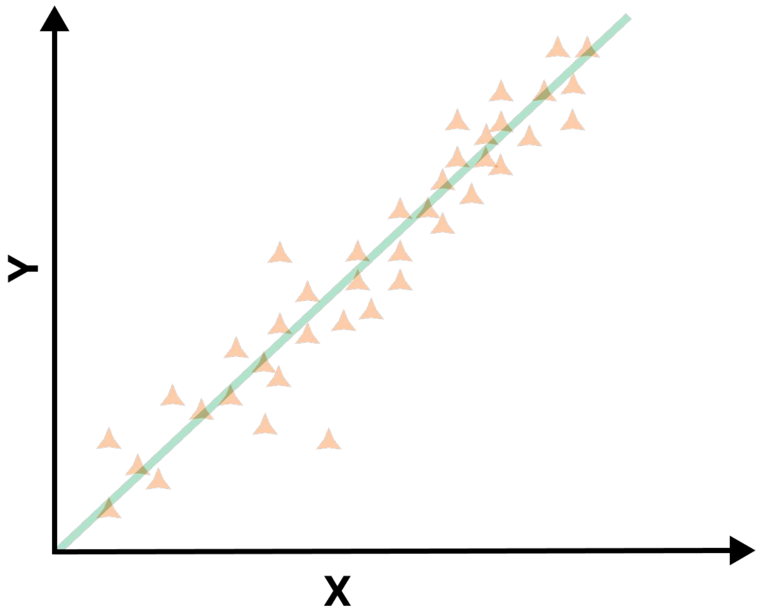
12/2/2025

Introduction

LLM = function that predicts the next token

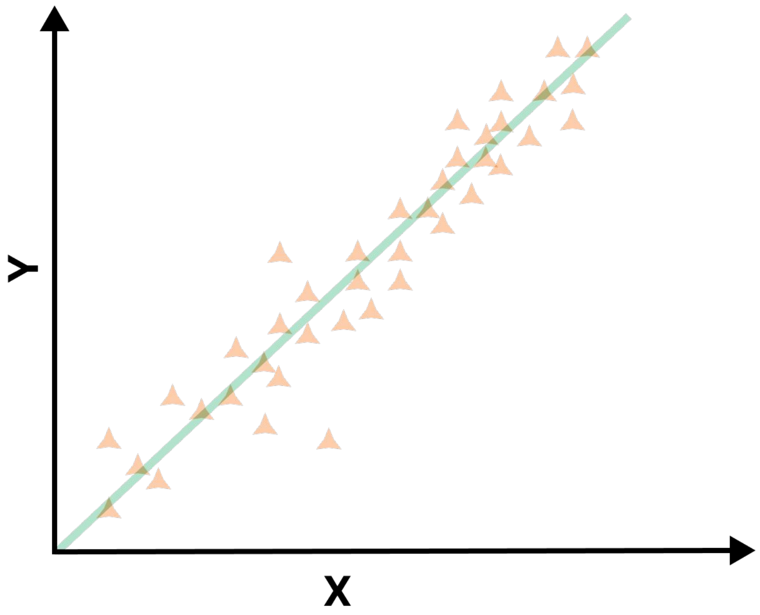


LLM = function that predicts the next token



$$f(x) = ax + b$$

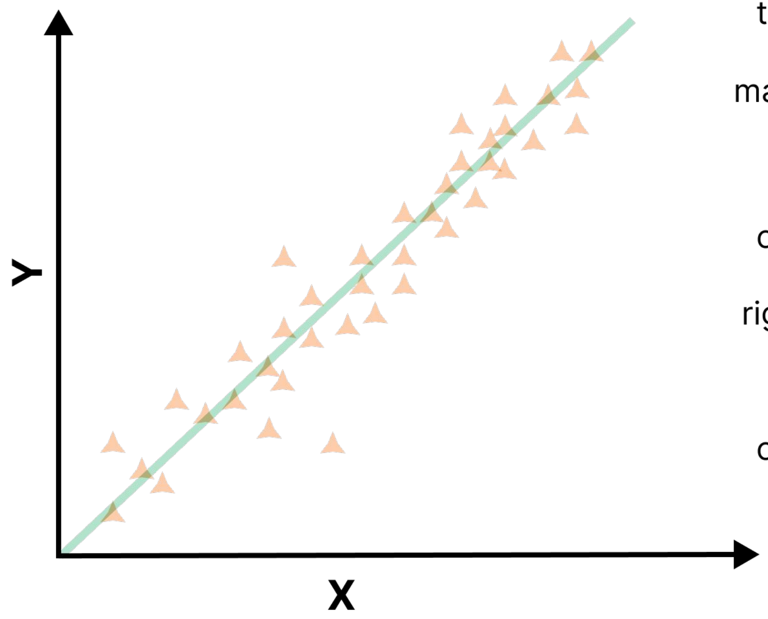
LLM = function that predicts the next token



$$f(x) = ax + b$$

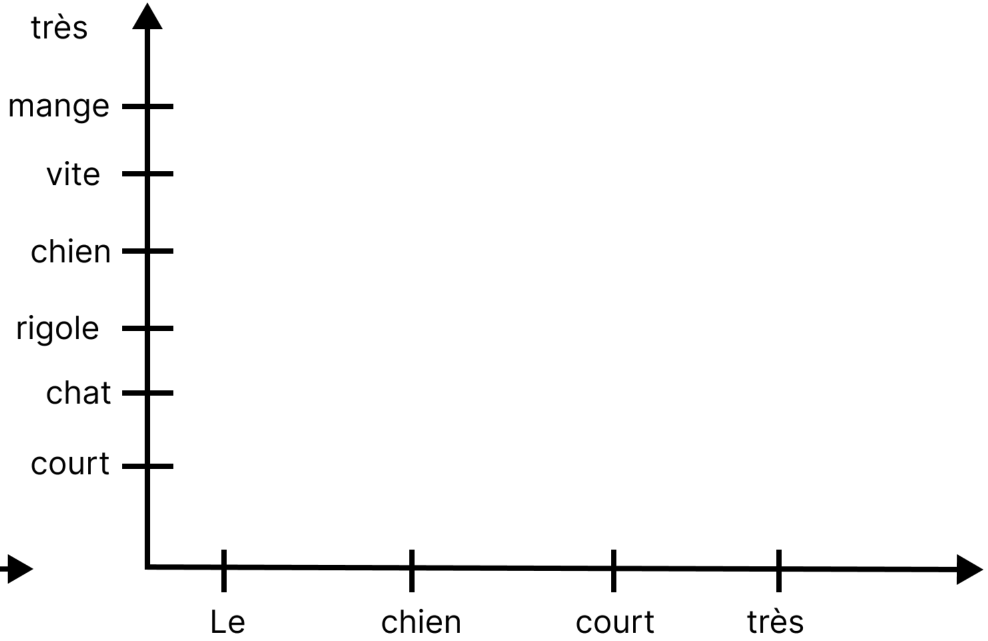
2 parameters a, b

LLM = function that predicts the next token

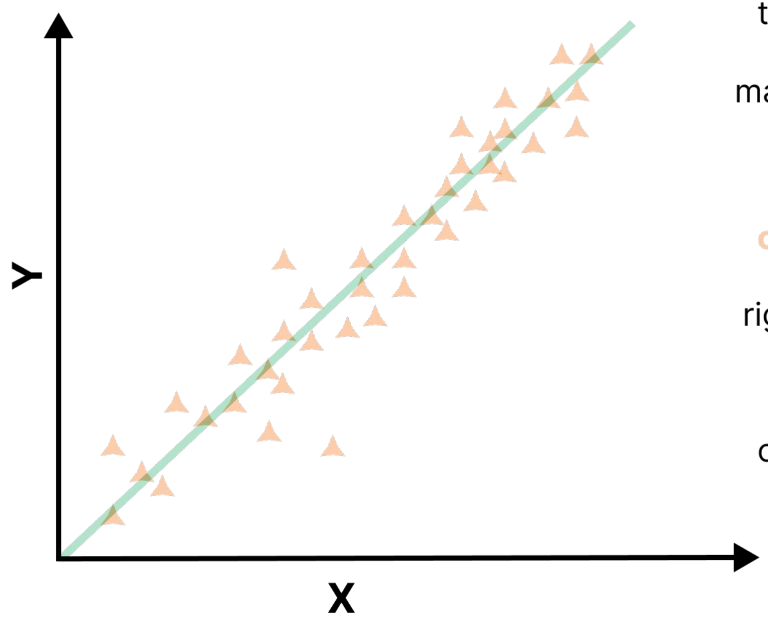


$$f(x) = ax + b$$

2 parameters a, b

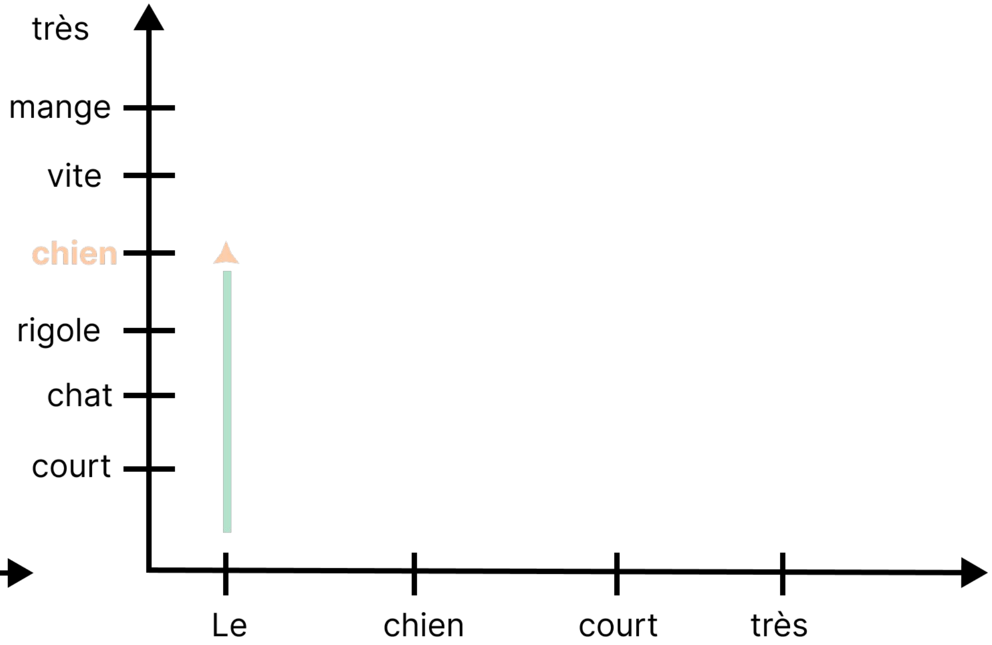


LLM = function that predicts the next token

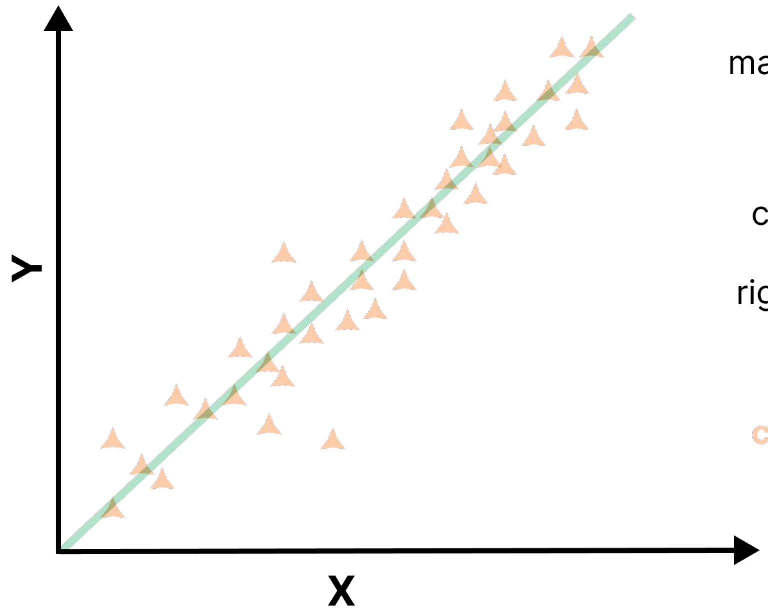


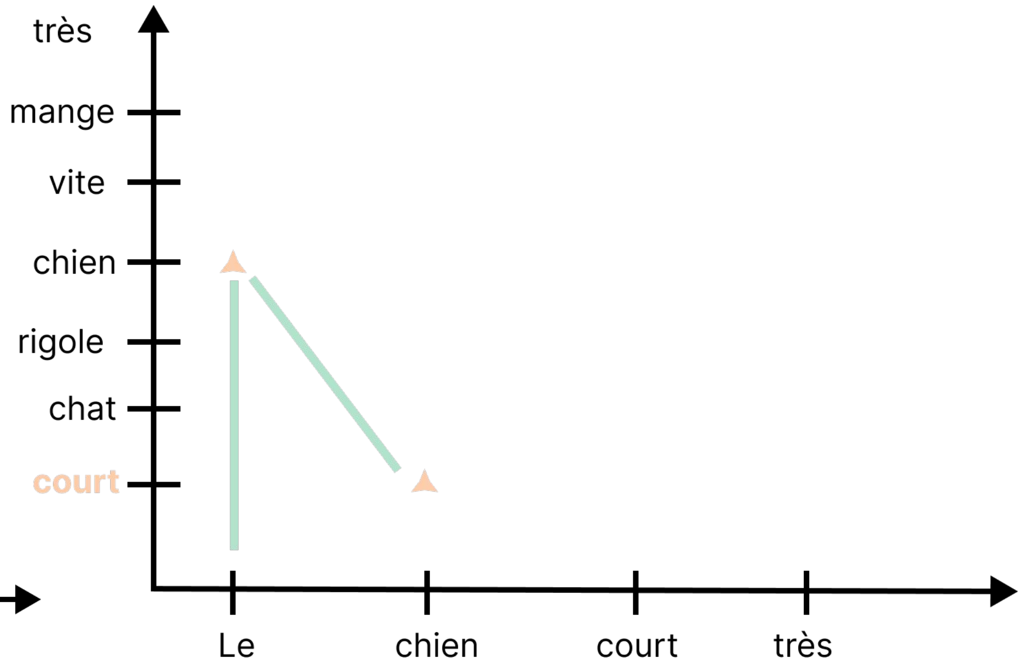
$$f(x) = ax + b$$

2 parameters a, b



LLM = function that predicts the next token

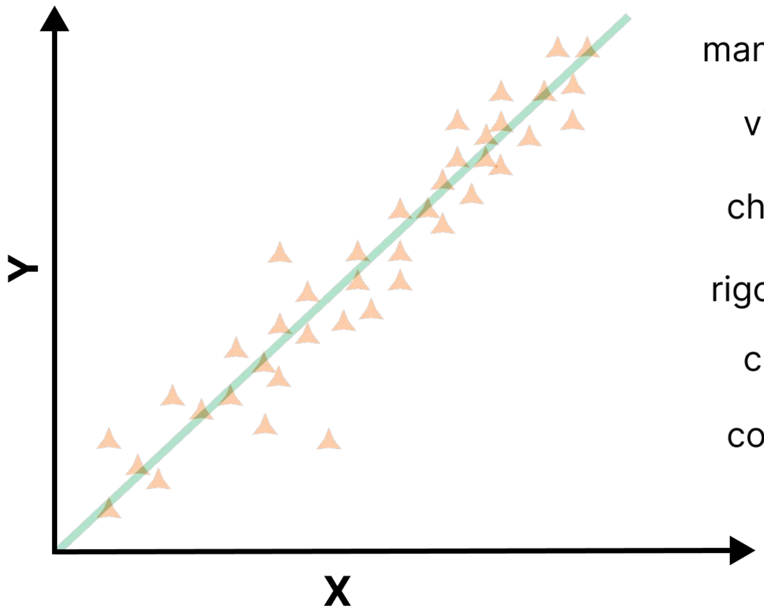




$$f(x) = ax + b$$

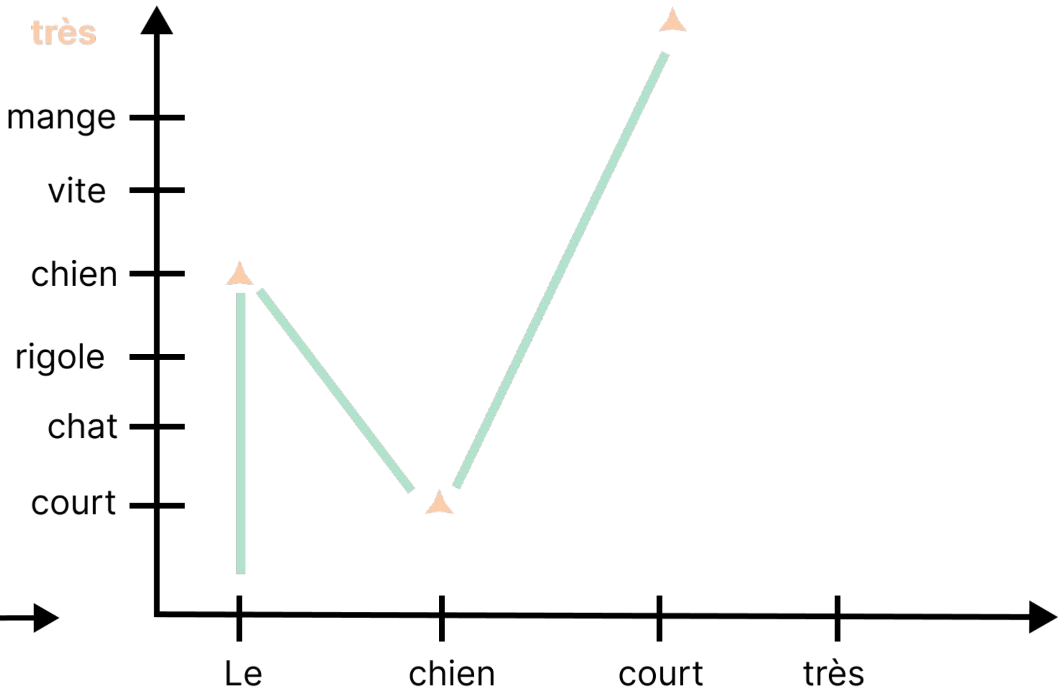
2 parameters a, b

LLM = function that predicts the next token

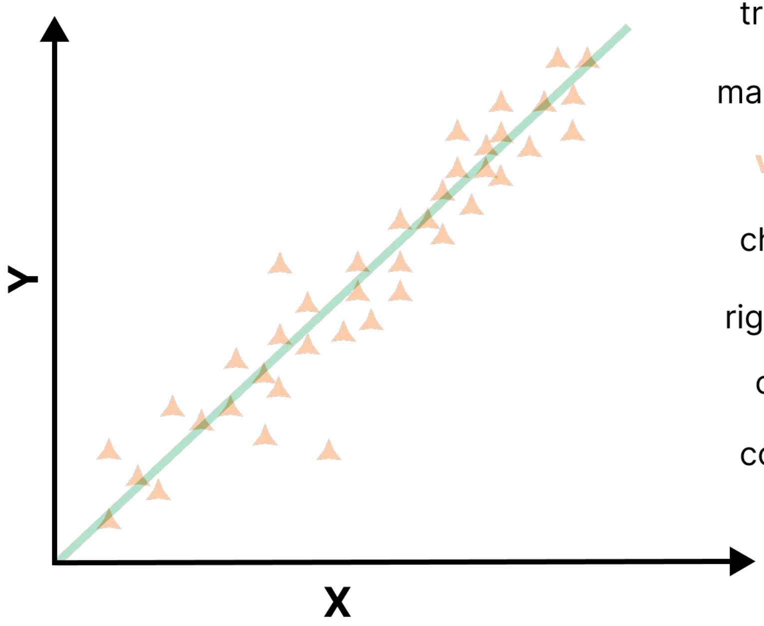


$$f(x) = ax + b$$

2 parameters a, b

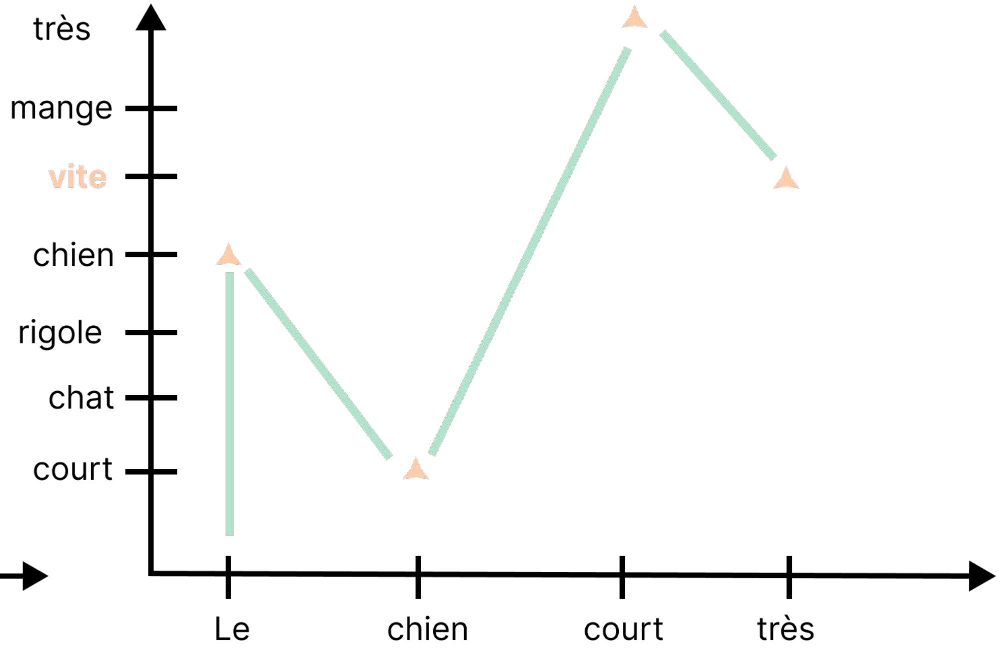


LLM = function that predicts the next token

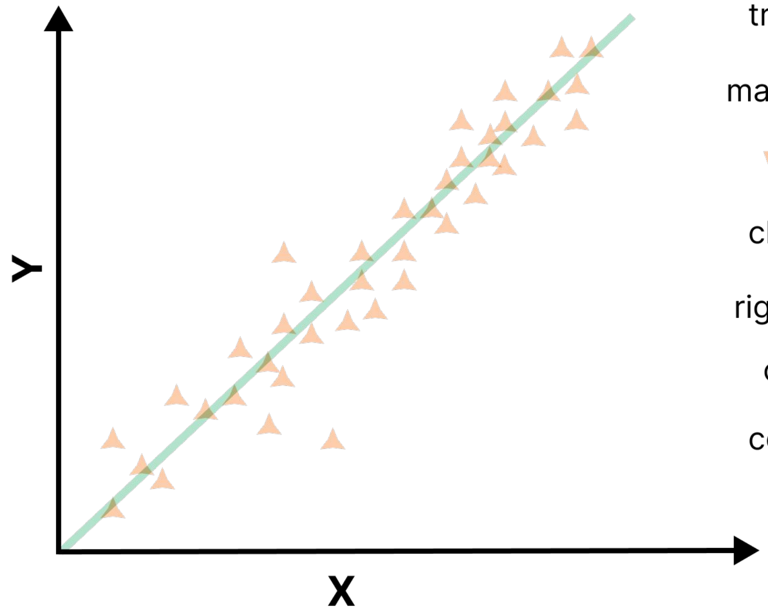


$$f(x) = ax + b$$

2 parameters a, b

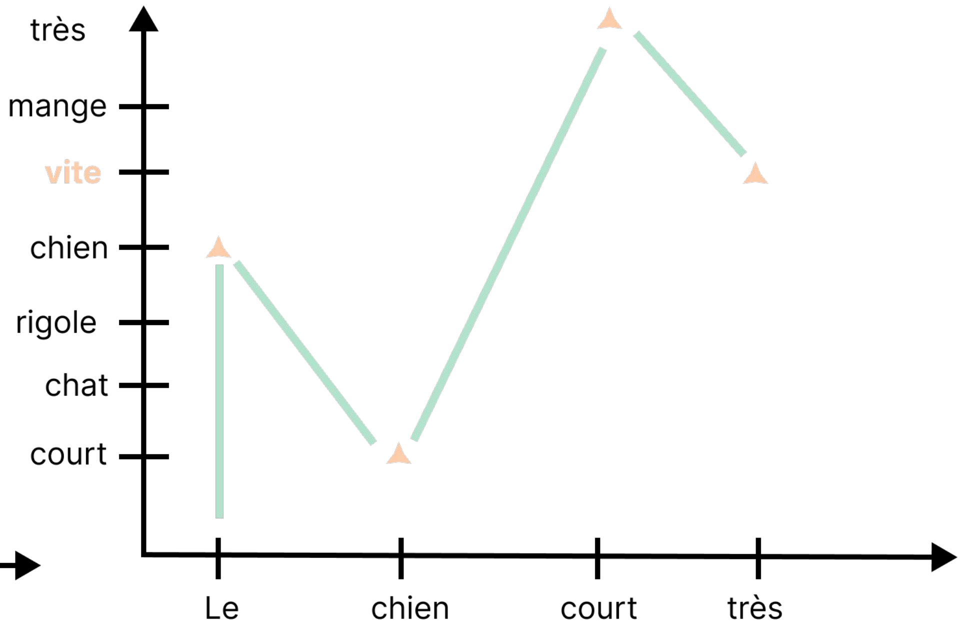


LLM = function that predicts the next token



$$f(x) = ax + b$$

2 parameters: a, b



$$f(\mathbf{x}) = \mathrm{LLM}(\mathbf{x})$$

billion of parameters!

Language Models are Unsupervised Multitask Learners

Alec Radford [*1] **Jeffrey Wu** [*1] **Rewon Child** [1] **David Luan** [1] **Dario Amodei** [**1] **Ilya Sutskever** [**]

Many natural language tasks can be solved with next-token prediction.

- *Translation:* The dog runs fast → **Le chien court**

Language Models are Unsupervised Multitask Learners

Alec Radford [*1] Jeffrey Wu [*1] Rewon Child [1] David Luan [1] Dario Amodei [**1] Ilya Sutskever [**]

Many natural language tasks can be solved with next-token prediction.

- *Translation:* The dog runs fast → **Le chien court vite**

Language Models are Unsupervised Multitask Learners

Alec Radford [*1] **Jeffrey Wu** [*1] **Rewon Child** [1] **David Luan** [1] **Dario Amodei** [**1] **Ilya Sutskever** [**]

Many natural language tasks can be solved with next-token prediction.

- *Translation:* The dog runs fast → **Le chien court vite**
- *Classification:* Super le film. J’ai adoré ! →

Language Models are Unsupervised Multitask Learners

Alec Radford [*1] Jeffrey Wu [*1] Rewon Child [1] David Luan [1] Dario Amodei [**1] Ilya Sutskever [**]

Many natural language tasks can be solved with next-token prediction.

- *Translation:* The dog runs fast → **Le chien court vite.**

- *Classification:* Super le film. J'ai adoré ! → **Positive**

Language Models are Unsupervised Multitask Learners

Alec Radford [*1] Jeffrey Wu [*1] Rewon Child [1] David Luan [1] Dario Amodei [**1] Ilya Sutskever [**]

Many natural language tasks can be solved with next-token prediction.

- *Translation:* The dog runs fast → **Le chien court vite.**

- *Classification:* Super le film. J’ai adoré ! → **Positive**

- *Chat:* Hello, who are you? →

Language Models are Unsupervised Multitask Learners

Alec Radford [*1] Jeffrey Wu [*1] Rewon Child [1] David Luan [1] Dario Amodei [**1] Ilya Sutskever [**]

Many natural language tasks can be solved with next-token prediction.

- *Translation:* The dog runs fast → **Le chien court vite.**

- *Classification:* Super le film. J’ai adoré ! → **Positive**

- *Chat:* Hello, who are you? → **I**

Language Models are Unsupervised Multitask Learners

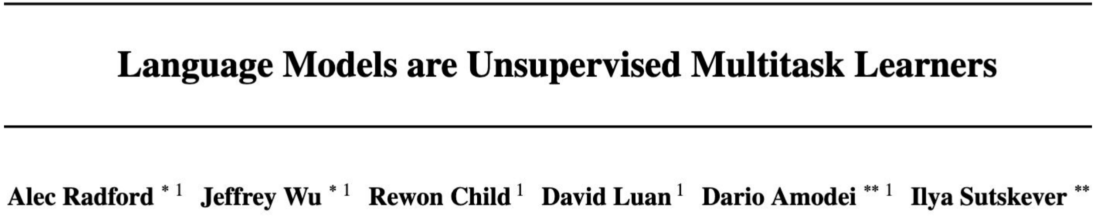
Alec Radford [* 1] **Jeffrey Wu** [* 1] **Rewon Child** [1] **David Luan** [1] **Dario Amodei** [** 1] **Ilya Sutskever** [**]

Many natural language tasks can be solved with next-token prediction.

- *Translation:* The dog runs fast → **Le chien court vite.**

- *Classification:* Super le film. J’ai adoré ! → **Positive**

- *Chat:* Hello, who are you? → **I**



Many natural language tasks can be solved with next-token prediction.

- *Translation:* The dog runs fast → **Le chien court vite.**

- *Classification:* Super le film. J'ai adoré ! → **Positif**

- *Chat:* Hello, who are you? → **I am**

Language Models are Unsupervised Multitask Learners

Alec Radford [*1] **Jeffrey Wu** [*1] **Rewon Child** [1] **David Luan** [1] **Dario Amodei** [**1] **Ilya Sutskever** [**]

Many natural language tasks can be solved with next-token prediction.

- *Translation:* The dog runs fast → **Le chien court vite.**

- *Classification:* Super le film. J'ai adoré ! → **Positif**

- *Chat:* Hello, who are you? → **I am an**

Language Models are Unsupervised Multitask Learners

Alec Radford [*1] **Jeffrey Wu** [*1] **Rewon Child** [1] **David Luan** [1] **Dario Amodei** [**1] **Ilya Sutskever** [**]

Many natural language tasks can be solved with next-token prediction.

- *Translation:* The dog runs fast → **Le chien court vite.**

- *Classification:* Super le film. J’ai adoré ! → **Positif**

- *Chat:* Hello, who are you? → **I am an AI**

Language Models are Unsupervised Multitask Learners

Alec Radford [* 1] **Jeffrey Wu** [* 1] **Rewon Child** [1] **David Luan** [1] **Dario Amodei** [** 1] **Ilya Sutskever** [**]

Many natural language tasks can be solved with next-token prediction.

- *Translation:* The dog runs fast → **Le chien court vite.**

- *Classification:* Super le film. J’ai adoré ! → **Positif**

- *Chat:* Hello, who are you? → **I am an AI assistant.**

Learning to predict is learning a probability function

An LLM as a probability function

An LLM is a function that predicts the next token t given its parameters θ and the context c:

$$LLM(t|c,\theta)$$

And this function is a probability function:

$$p(t|c,\theta)$$

θ are the neural network parameters which is the Transformer.

c is the context vector computed by the *Transformer*, (e.g, c = *Transformer*(text))

$$p(t|c,\theta) = \frac{score(c,t)}{\sum_{t' \in \mathcal{V}} score(c,t')}$$

An LLM as a probability function

$$p(t|c,\theta) = \frac{score(c,t)}{\sum_{t' \in \mathcal{V}} score(c,t')}$$

$score(c,t)$ measures how strong the context *c* is associated with the token *t*

$score(c,t')$ measures how strong the context *c* is associated with any other token *t'* in the vocabulary $\mathcal{V}$

$score(c,t) = \mathbf{c} \cdot \mathbf{w}_t^T$ The score can be defined as a dot product between the context vector and the token weight. This is also called ***logits***

An LLM as a probability function

$$p(t|c,\theta) = \cancel{\frac{\mathbf{c} \cdot \mathbf{w}_t^T}{\sum_{t' \in \mathcal{V}} \mathbf{c} \cdot \mathbf{w}_{t'}^T}}$$

→ A probability is between 0 and 1, and cannot be negative. We need positive scores.

→ *The dot product score returns any real number*, it can be positive or negative. It is not suited for a probability function

→ To always have positive scores, the solution is to take the *exponential of the scores*:

$$p(t|c,\theta) = \frac{\exp(\mathbf{c} \cdot \mathbf{w}_t^T)}{\sum_{t' \in \mathcal{V}} \exp(\mathbf{c} \cdot \mathbf{w}_{t'}^T)}$$

→ This is ***softmax***

An LLM as a probability function

→ Now that we know that an LLM is a probability function, we can use it to compute the probability of any text.

→ The probability of the text “<b> I love this <e>” is:

$$
\begin{aligned}
p(\text{<b> I love this <e>}) = \; & p(t = \text{I} \mid c = \text{<b>}, \theta) \\
& \times p(t = \text{love} \mid c = \text{<b> I}, \theta) \\
& \times p(t = \text{this} \mid c = \text{<b> I love}, \theta) \\
& \times p(t = \text{<e>} \mid c = \text{<b> I love this}, \theta)
\end{aligned}
$$

An LLM as a probability function

→ In general, the probability of any text *s* under the LLM θ *:*

$$p(\mathrm{s}) = \prod_{i=1}^{N} p(t = s_i \mid c = s_{<i}, \theta)$$

→ In practice, for numerical stability reason, we compute the log-probability:

$$\log p(\mathrm{s}) = \sum_{i=1}^{N} \log p(t = s_i \mid c = s_{<i}, \theta)$$

How this probability function is learned?

How an LLM learns a good probability function ?

To summarize:

> An LLM is a probability function ***p*** based on the Transformer architecture with parameters $\boldsymbol{\theta}$, that computes the probability of any sequence ***p(s)***

But we don’t know in advance what is the probability function governing the texts. The LLMs needs to learn one.

How an LLM learns a good probability function?

The LLM is trained to learn the probability function that maximizes the probability of the training data ***D***. We call this approach ***Maximum Likelihood Estimation*** (**MLE**).

$$\log P(\mathcal{D};\theta) = \sum_{s\in\mathcal{D}} \sum_{i=1}^{|s|} \log p(t = s_i \mid c = s_{<i}, \theta)$$

Then the LLM θ learns to maximize the probability of the training data ***D***

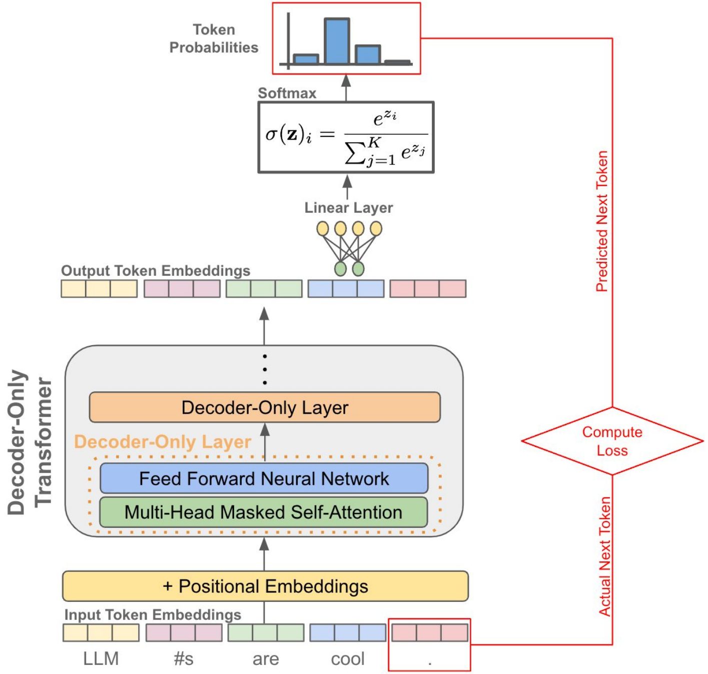
$$\hat{\theta} = \arg\max_{\theta}\ \log P(\mathcal{D};\theta)$$

Maximizing the log-probabilities is equivalent to minimizing the negative log-probabilities, which is equivalent to ***cross-entropy loss***

The different steps of LLM Training

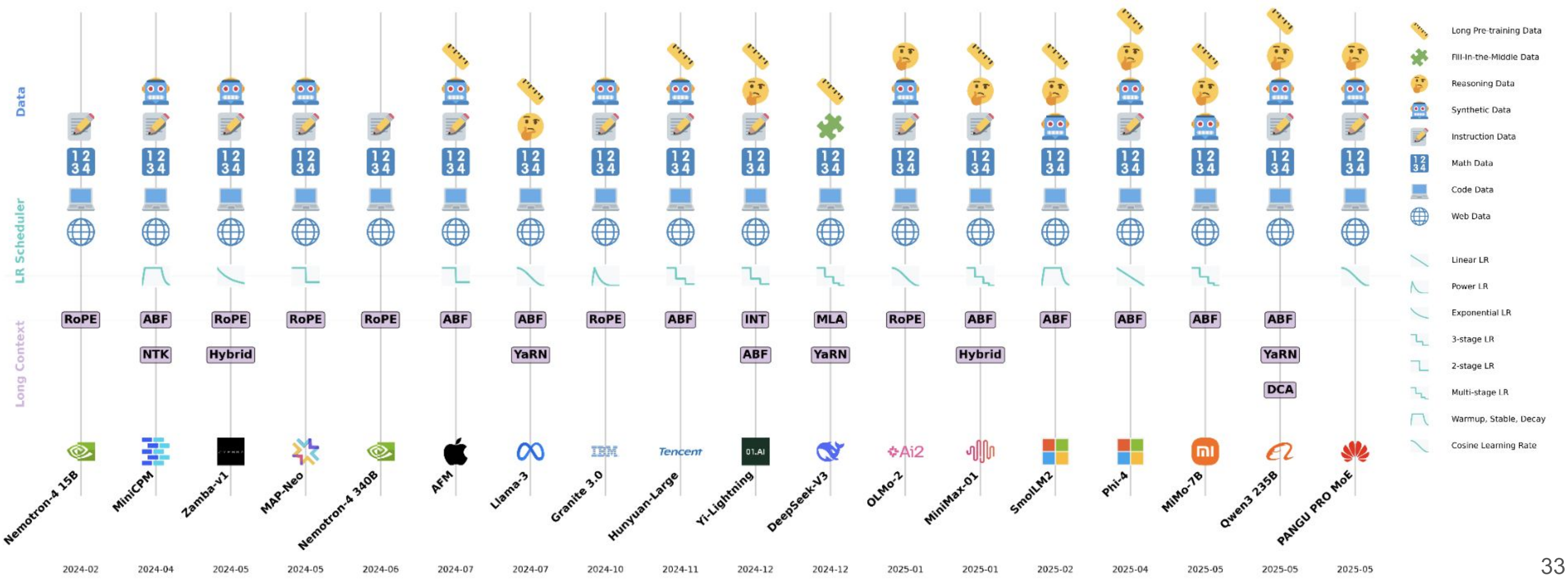
1. Pre-training

The step where the LLM learns to maximize the probability of the raw training data (*from the web)* is called ***pre-training***.



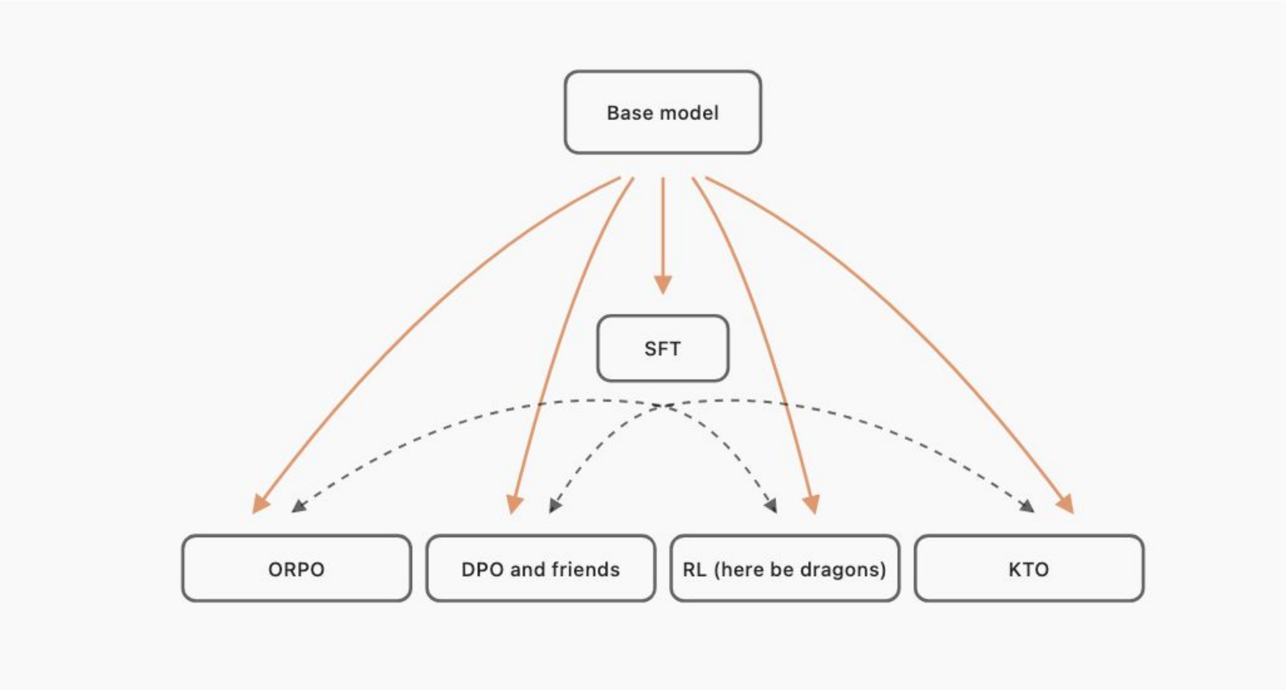
The different steps of LLM Training

2. Mid-training: Long-context extension, Multilingual data, annealing, etc.



The different steps of LLM Training

3. Pos-training: Instruction following, preference alignment, agents, etc.



Sampling from an LLM

To fully benefits from the knowledge of the pre-trained LLM, we need to find good methods to sample tokens from the probability distribution.

This is not trivial.

You need to find a good trade-off between diversity of the generated texts, and coherence.

The general generation loop

```
def generate(prompt):
    generated ← prompt
    while True:
        next token t = sample ~ p(·|c=generated)
        generated = generated + t
        if t == end_of_sequence token:
            return generated
        continue
```

Greedy generation loop

```
def greedy(prompt):
    generated ← prompt
    while True:
        # pick the token t with the highest score
        next token t = max ~ p(·|c=generated)
        generated = generated + t
        if t == end_of_sequence token:
            return generated
        continue
```

Top-K generation loop

```
def topk(prompt, k):
    generated ← prompt
    while True:
        # pick the token t with the highest score
        filtered_logits = keep the k highest scores
        p(·|c=generated) = softmax(filtered_logits)
        next token t = sample ~ p(·|c=generated)
        generated = generated + t
        if t == end_of_sequence token:
            return generated
        continue
```

For more diversity

Greedy generation algorithm will always output the same text

Topk will also very often output the same set of texts

Diversity is important for many reasons:

- creativity
- more exploration (we will see this for RL)
- avoid looping: “Hi, i am i am i am i am i am…”

The temperature hack

Observation: exponential in softmax “exaggerates” the differences in the logits

Solution: make the logits smaller by dividing them with a constant

```
logits = [10, 5, 1] # a list of logits of 3 tokens

# without temperature
exp(logits) = [22026, 148, 2.7]
probabilities = [0.99, 0.01, 0.00]

# with temperature t = 5
logits = [10, 5, 1] / 5 = [2.0, 1.0, 0.2]
exp(logits) = [7.39, 2.72, 1.22]
probabilities = [0.65, 0.24, 0.11]
# the token 2 and 3 have more chance to be sampled now!
```

Temperature generation loop

```
def temperature_generation(prompt, t):
    generated ← prompt
    while True:
        # pick the token t with the highest score
        temperature_logits = logits / t
        p(·|generated) = softmax(temperature_logits)
        next token t = sample ~ p(·|generated)
        generated = generated + t
        if t == end_of_sentence token:
            return generated
        continue
```

Diversity / coherence trade-off

With previous sampling methods, the generation can drift quickly if we inadvertently sample the wrong token. We need a trade-off between diversity and coherence.

sorted_probs = [0.70, 0.10, 0.08, 0.05, 0.03, 0.02, 0.02]

Top-k (k=3): Samples from the first 3 tokens

- *Problem*: Includes token 3 with low score (0.08)
- If sampled, this weak token can drift the generation

Top-p (p=0.8): Samples from tokens where probability mass is concentrated

cumsum = [**0.70, 0.80**, 0.88, 0.93, 0.96, 0.98, 1.00]

Advantages:

- More adaptive: keeps only high-confidence tokens
- Prevents sampling from weak candidates

Top_p generation loop

```
def top_p(prompt, p):
    generated ← prompt
    while True:
        probabilites = p(·|generated)
        cumsum_probabilites = cumsum(sorted(probabilities))
        # filter token that are <p
        logits = filter logits where cumsum_probabilites <p
        p(·|generated) = softmax(logits)
        next token t = sample ~ p(·|generated)
        generated = generated + t
        if t == end_of_sentence token:
            return generated
        continue
```

Conclusion

→ The LLM estimates the probability distribution underlying the training data (the web), thanks to ***Maximum Likelihood Estimation*** by learning to predict next tokens

→ After training, we can sample from the learned probability distribution

→ Sampling is a trade-off between diversity and coherence:

- **Greedy**: no diversity
- **Temperature**: introduces diversity, but high temperature leads to girbish texts
- **TopK:** coherence but less diversity, and the generation can drift
- **TopP:** Adaptative to the probability distribution. A good trade-off between diversity and coherence. But it is not easy to find a good value for top_p.
- **Many other sampling methods:** MinP, Typical Sampling, Beam-Search, etc.