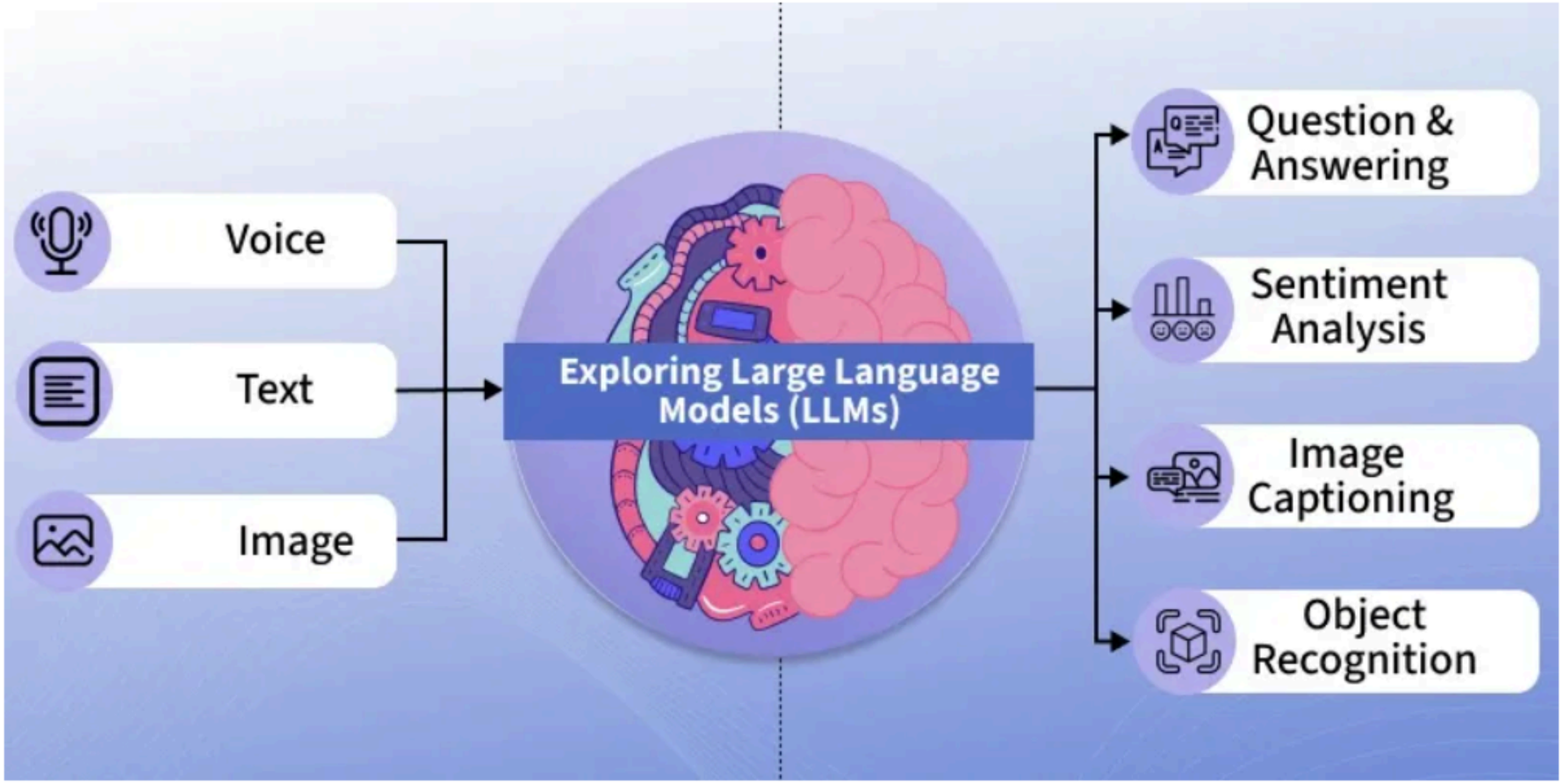
ARCHITECTING INTELLIGENCE

What are LLMs??

What is a Large Language Model (LLM)

Last Updated : 18 Sep, 2025

Large Language Models (LLMs) are advanced AI systems built on deep neural networks designed to process, understand and generate human-like text. By using massive datasets and billions of parameters, LLMs have transformed the way humans interact with technology. It learns patterns, grammar and context from text and can answer questions, write content, translate languages and many more. Mordern LLMs include ChatGPT (OpenAI), Google Gemini, Anthropic Claude, etc



LLM = Large **L**anguage **M**odel

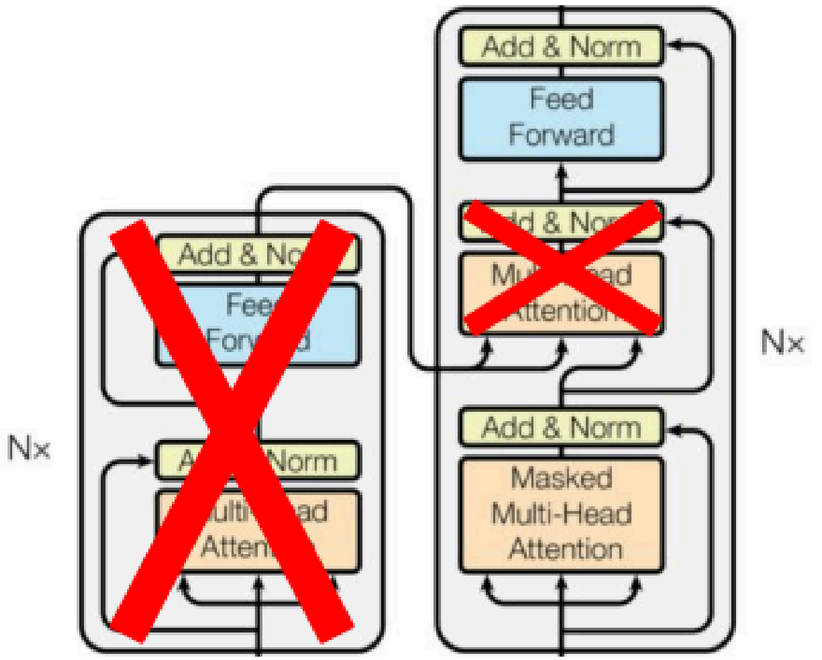
"A **language model** is a statistical or machine learning **model** that assigns **probabilities** to sequences of **tokens**."

LLM = Large Language Model

- **Model size**: billions of parameters or more
- **Training data**: 100s of billions of tokens or more
- **Compute**: a lot of GPUs

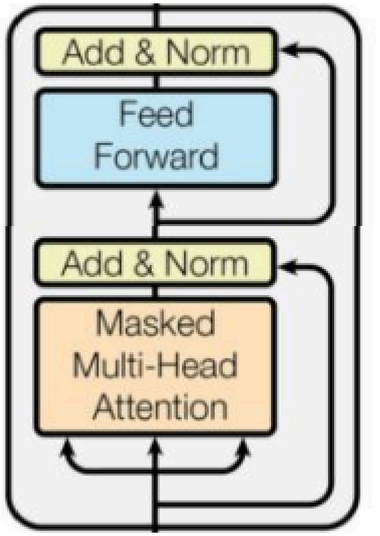
What’s the Architecture behind it ??

Decoder-only **Transformer**-based model



Large Language Models (LLMs) use a **decoder-only Transformer** architecture because it is the **simplest, most scalable, and most natural fit for next-token prediction**, which is the core objective of LLMs.

Decoder-only **Transformer**-based model



N×

Examples: GPT series, LLaMA, Gemma, DeepSeek, Mistral, Qwen, ...

What is a Mixture of Experts (MoE)?

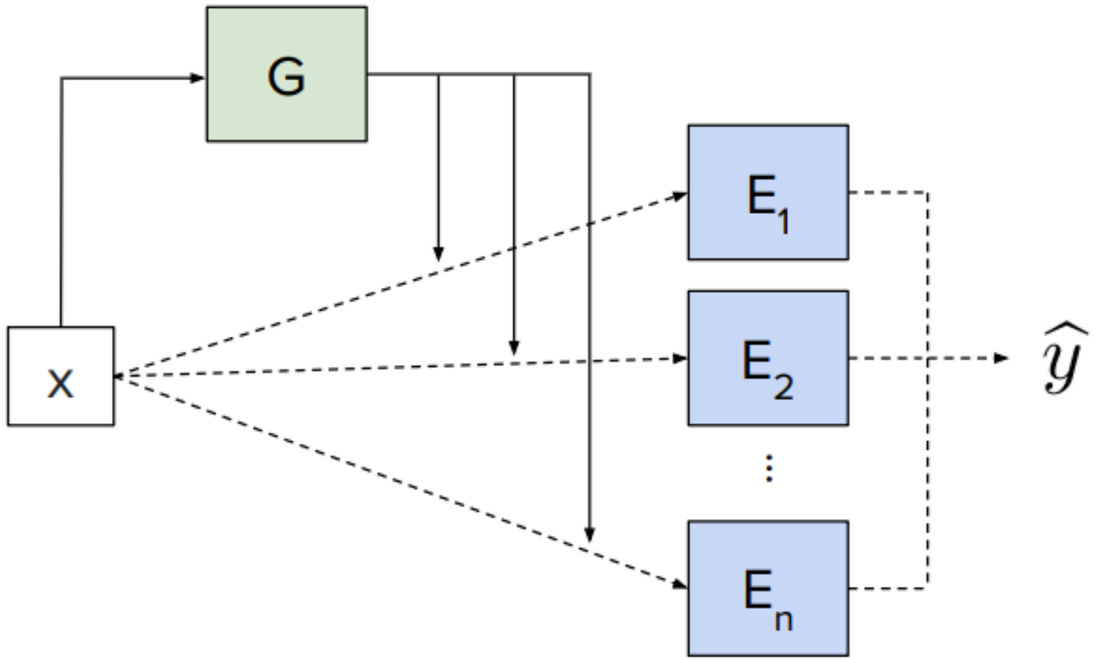
The scale of a model is one of the most important axes for better model quality. Given a fixed computing budget, training a larger model for fewer steps is better than training a smaller model for more steps.

Mixture of Experts enable models to be pretrained with far less compute, which means you can dramatically scale up the model or dataset size with the same compute budget as a dense model. In particular, a MoE model should achieve the same quality as its dense counterpart much faster during pretraining.

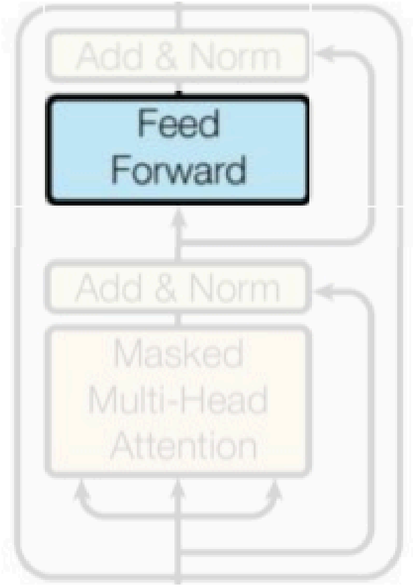
So, what exactly is a MoE? In the context of transformer models, a MoE consists of two main elements:

- **Sparse MoE layers** are used instead of dense feed-forward network (FFN) layers. MoE layers have a certain number of “experts” (e.g. 8), where each expert is a neural network. In practice, the experts are FFNs, but they can also be more complex networks or even a MoE itself, leading to hierarchical MoEs!
- A **gate network or router**, that determines which tokens are sent to which expert. For example, in the image below, the token “More” is sent to the second expert, and the token "Parameters” is sent to the first network. As we’ll explore later, we can send a token to more than one expert. How to route a token to an expert is one of the big decisions when working with MoEs - the router is composed of learned parameters and is pretrained at the same time as the rest of the network.

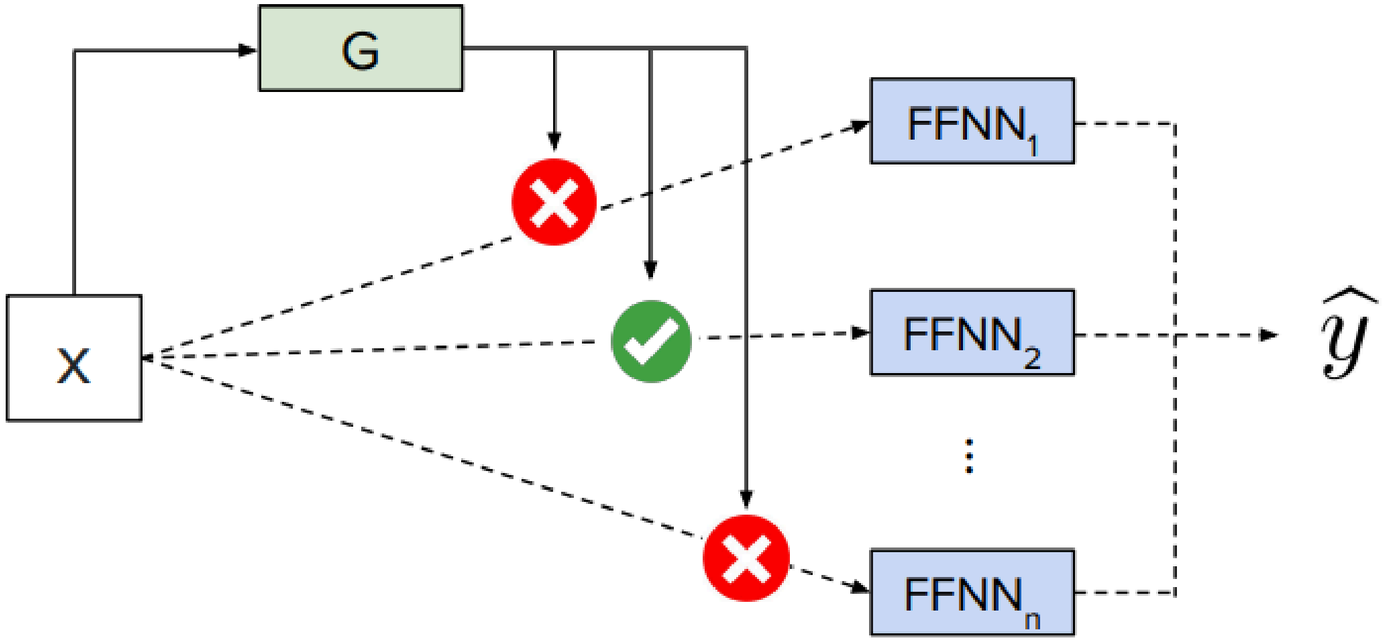
MoE = **M**ixture **o**f **E**xperts



$$\widehat{y} = \sum_{i=1}^{n} G(x)_i E_i(x)$$



Routing done **for each token**!



But Why???

1. LLMs are trained to do *one thing*: predict the next token

LLMs are trained with the objective:

> **Given tokens $x_1, x_2, \ldots, x_t$, predict x_{t+1}**

This is an **autoregressive** task.

A **decoder-only Transformer** does exactly this using **masked self-attention**, which ensures the model:

- Can only attend to **past tokens**
- Never "peeks" into the future

No encoder is needed because **there is no separate input sequence to encode**.

2. Why the encoder is removed

In the original Transformer (encoder–decoder):

- **Encoder**: processes an input sequence (e.g., a sentence to translate)
- **Decoder**: generates an output sequence while attending to the encoder

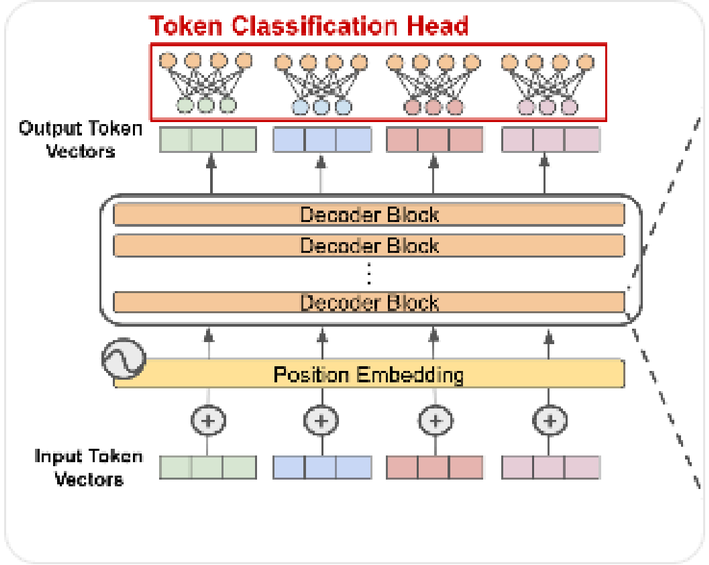
LLMs don't translate *from* a separate source anymore. Instead:

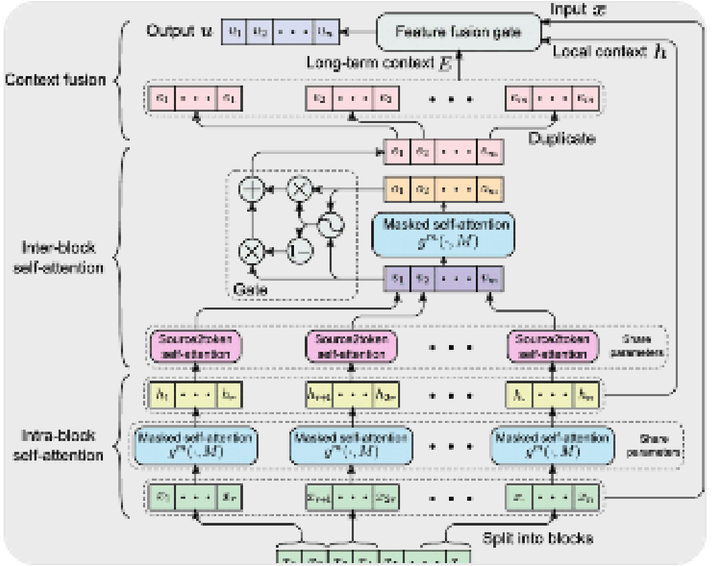
> **Prompt + instructions + context + question**
> are all treated as **one long sequence**

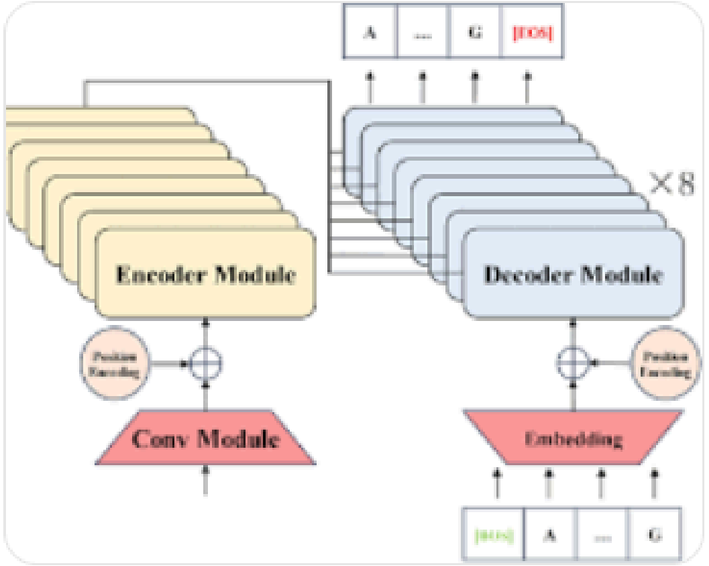
So:

- There is **nothing separate to encode**
- Cross-attention becomes unnecessary
- Everything is handled by **self-attention only**

3. Masked self-attention is the key







Masked self-attention ensures:

- Token *t* can only see tokens `< t`
- Causality is preserved
- Generation works token-by-token at inference

This is exactly what text generation requires.

4. Why this architecture scales best

Decoder-only models scale exceptionally well because:

(a) Training simplicity

- One loss: **next-token cross entropy**
- No alignment between input/output sequences
- Works on **any text** (books, code, chats, papers)

(b) Inference efficiency

- Can **cache key–value attention states**
- Each new token is fast to generate
- Critical for long conversations and streaming output

(c) Unification of tasks

With prompting, the same model can do:

- Q&A
- Summarization
- Translation
- Coding
- Reasoning

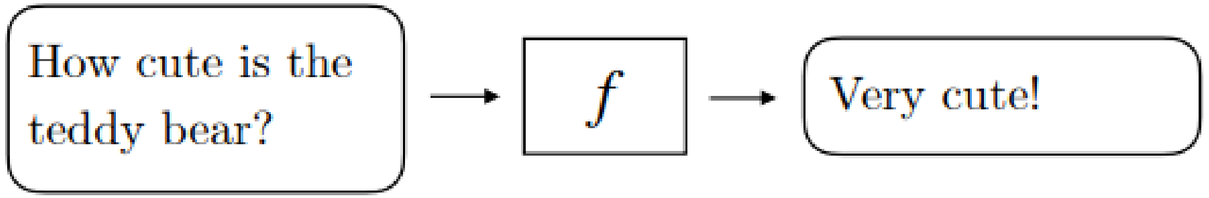
All without architectural changes.

5. Why not encoder-only or encoder–decoder?

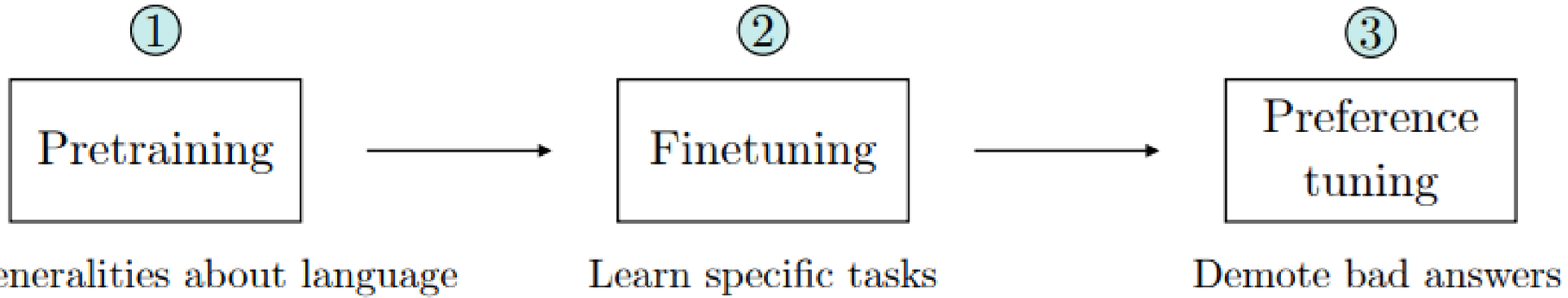
Architecture	Why it's not ideal for LLMs
Encoder-only (BERT)	Bidirectional → can't generate text
Encoder–decoder (T5)	Heavier, slower, unnecessary for pure generation
Decoder-only	✔ Natural for generation, ✔ scalable, ✔ simple

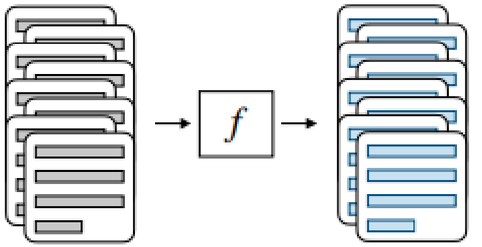
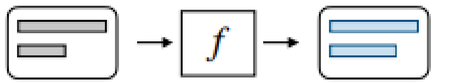
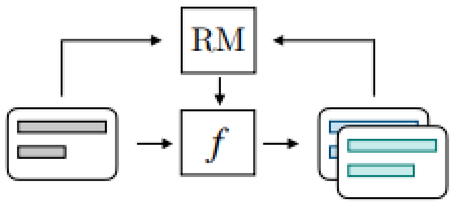
Behind The Scenes

It takes text as input and returns text as output.



The lifecycle of an LLM is done in three steps:

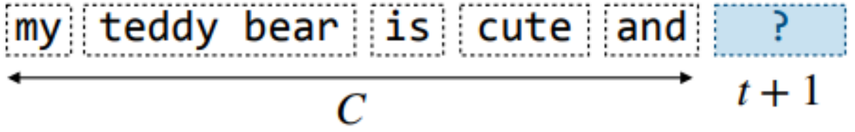


Step 1: **Pretraining**	*Only done once because it is time-consuming and resource-intensive.* *Data*: Huge dataset representing all kinds of text-based knowledge (e.g books, websites, code, etc.). *Techniques*: Next word prediction.	
Step 2: **Finetuning**	*Necessary if model after step 1 does not have good performance or if the task is non-trivial/highly specialized.* *Data*: Small and high-quality dataset showing exactly the task. *Techniques*: SFT, PEFT (LoRA, prefix tuning, adapters).	
Step 3: **Preference tuning**	*Optional step that is done only if certain behaviors appear in the model after step 2 and need to be penalized.* *Data*: Medium-sized set of preference pairs composed of observations to boost and those to demote. *Techniques*: RLHF (with RM and RL via PPO), DPO, IPO.	

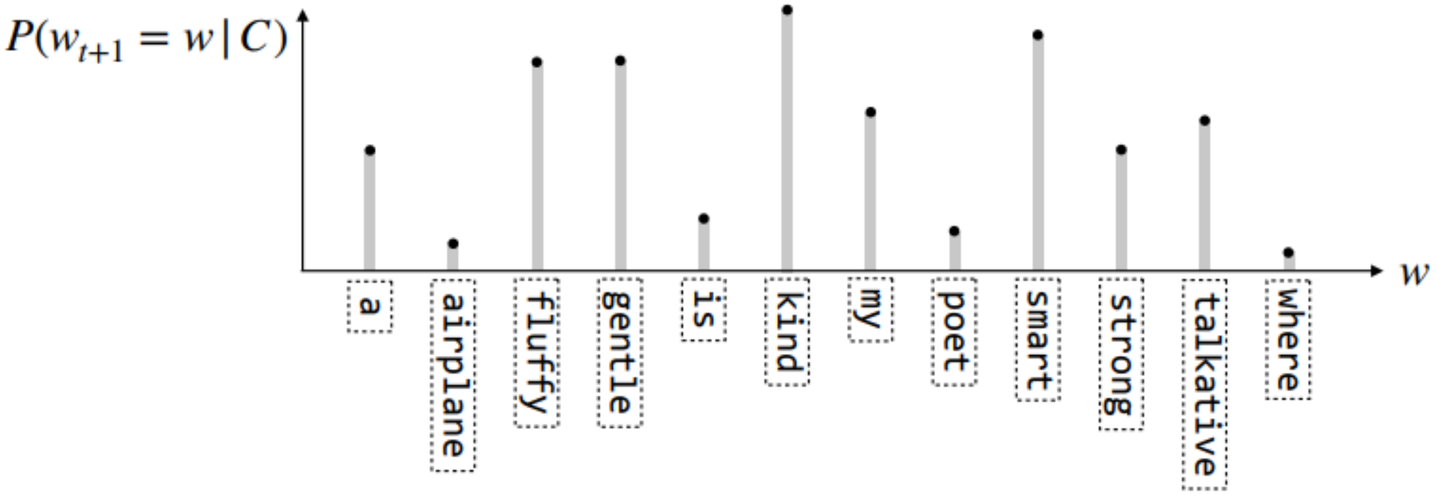
Response Generation

Now, we will learn about the different ways that an autoregressive model can generate a response based on its predicted probabilities.

❒ **Notations** – Autoregressive models generate output text one token at a time. Given the model's raw predicted probabilities, the goal is to determine which token should be generated next.

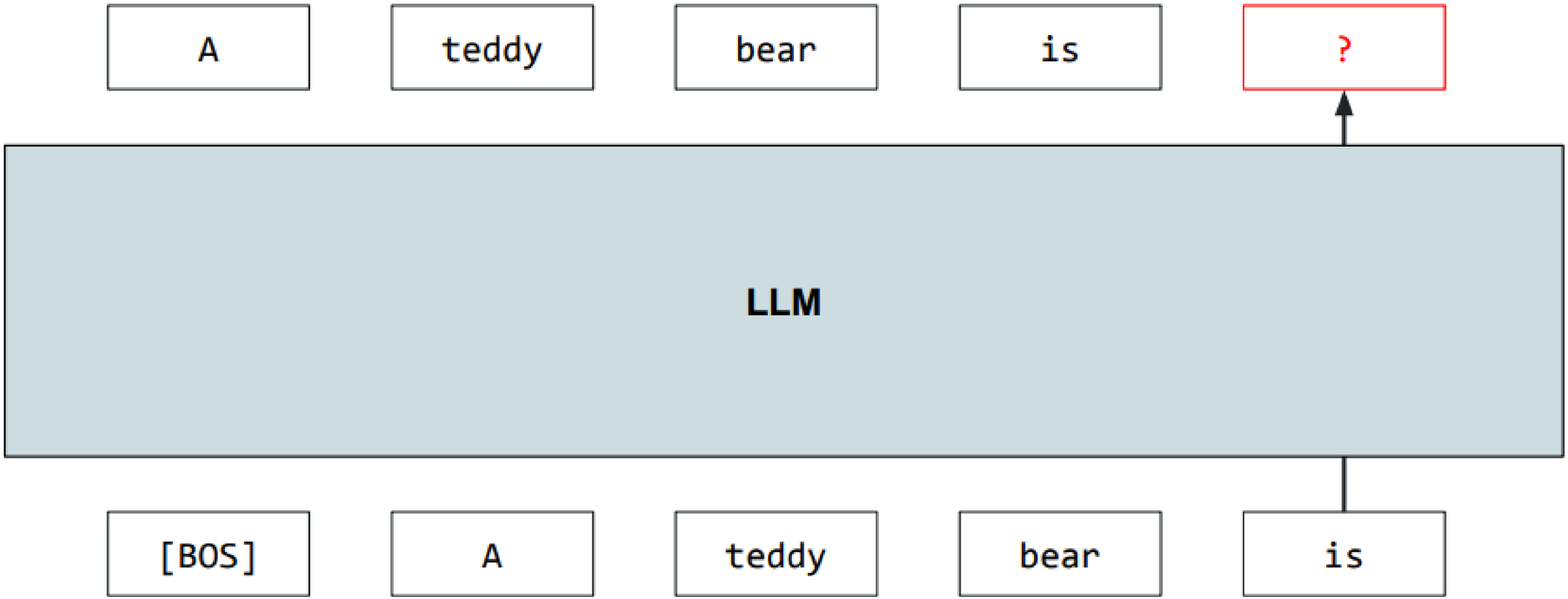


More formally, at each timestep t, the model predicts the probability distribution $P(w_{t+1}|C)$ of the next token w_{t+1} over the vocabulary V based on the context $C = w_1 ... w_t$. In other words, it outputs a probability score $P(w_{t+1} = w|C)$ on each of the tokens w of the vocabulary V via a softmax layer.

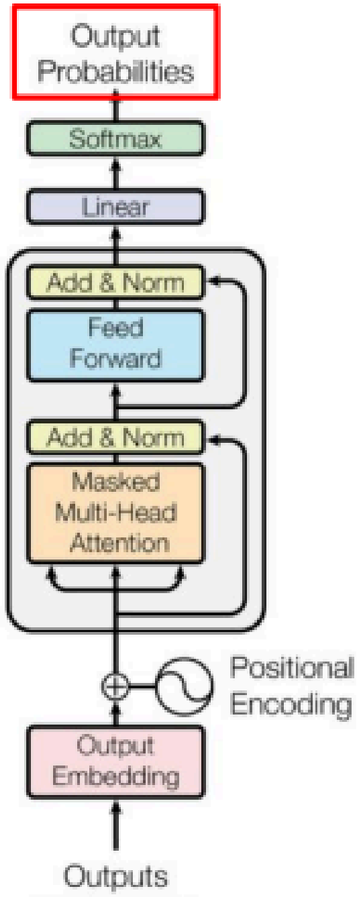


The goal of this part is to determine a way to generate the next token $\widehat{w}_{t+1}$.

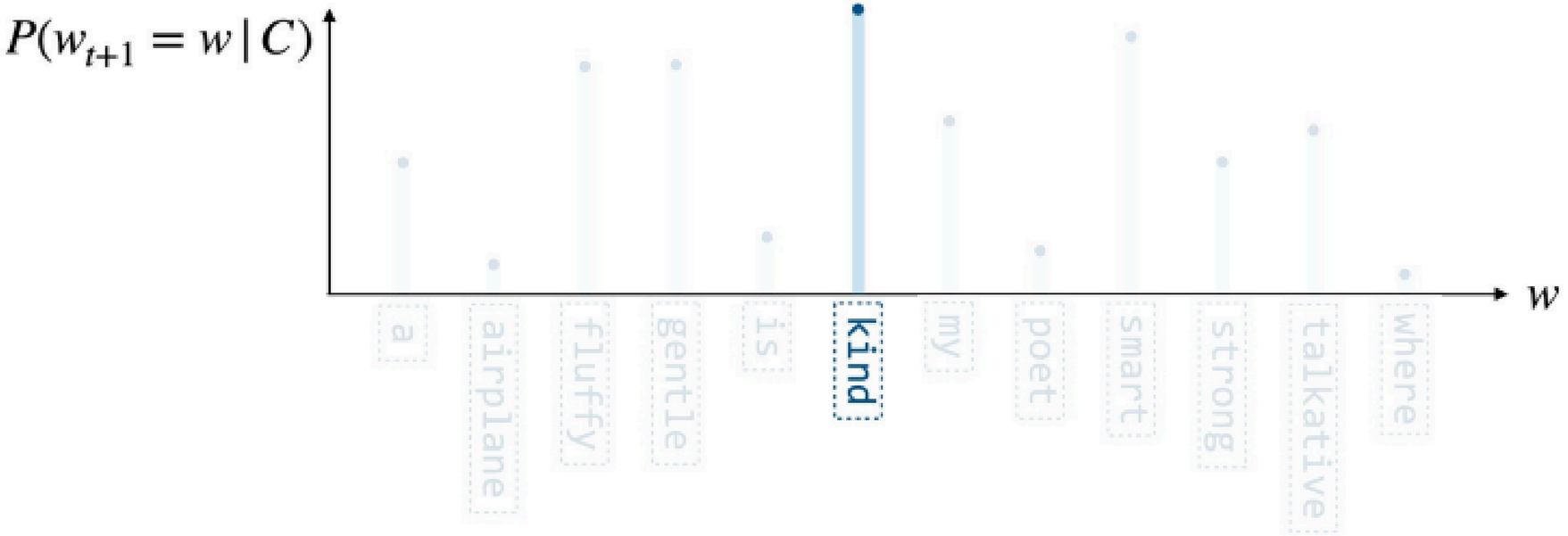
Next token prediction



Predicting next token with greedy decoding

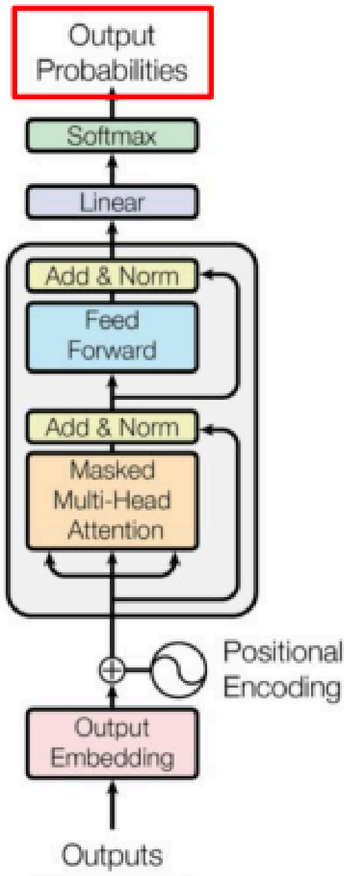


1st idea. Take token with highest predicted probability

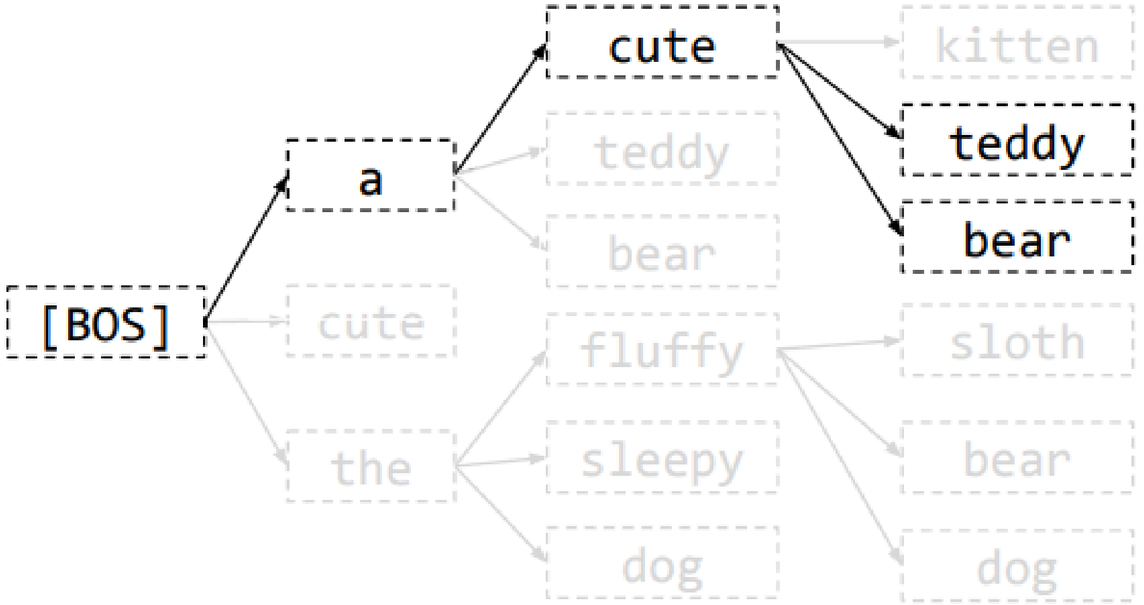


Limitations. Output not optimal, natural and/or diverse

Predicting next token with beam search



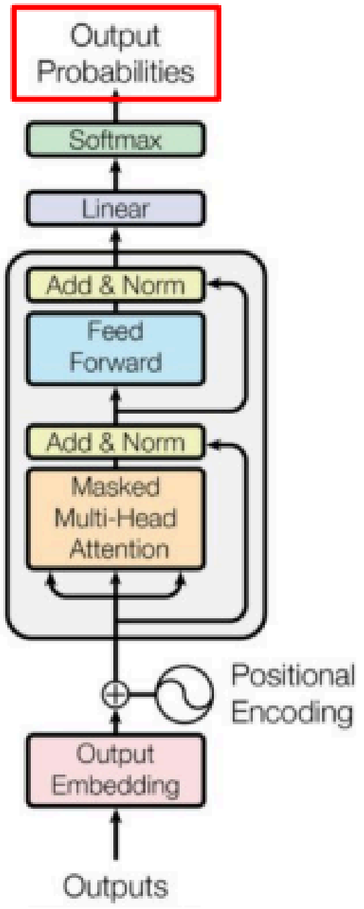
2nd idea. Keep k paths that are the most likely



Ends at [EOS]

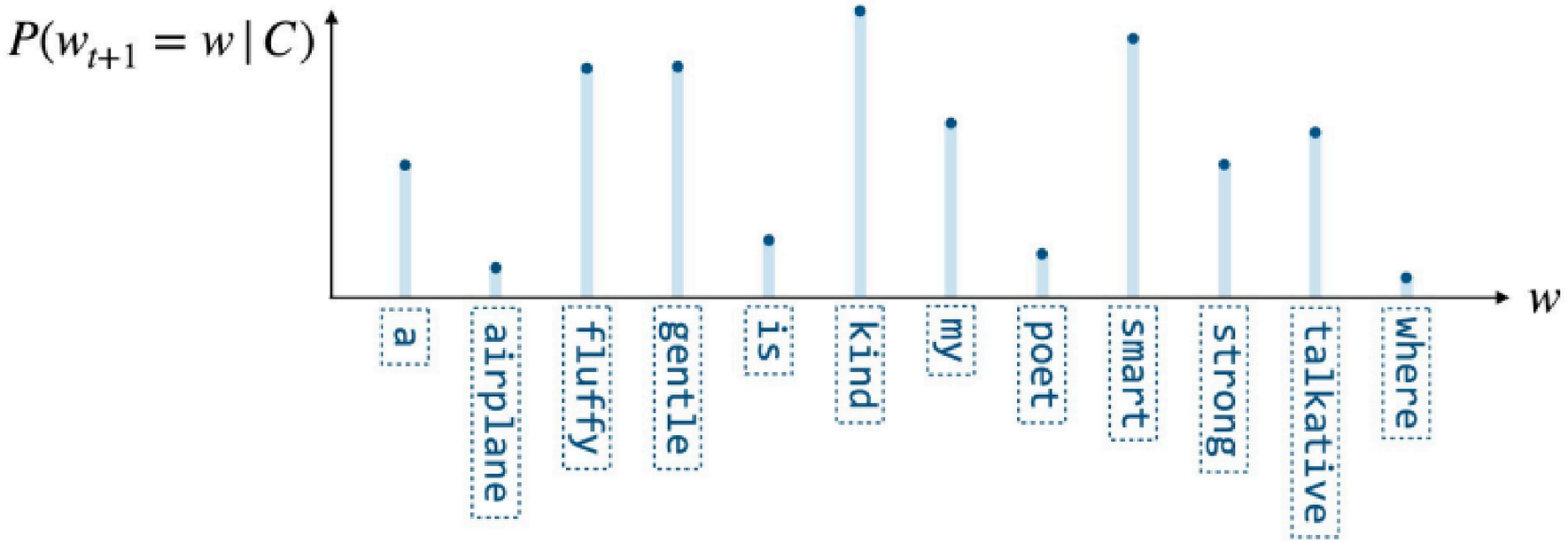
Limitations. Needs computations + lacks diversity/creativity

Predicting next token with sampling

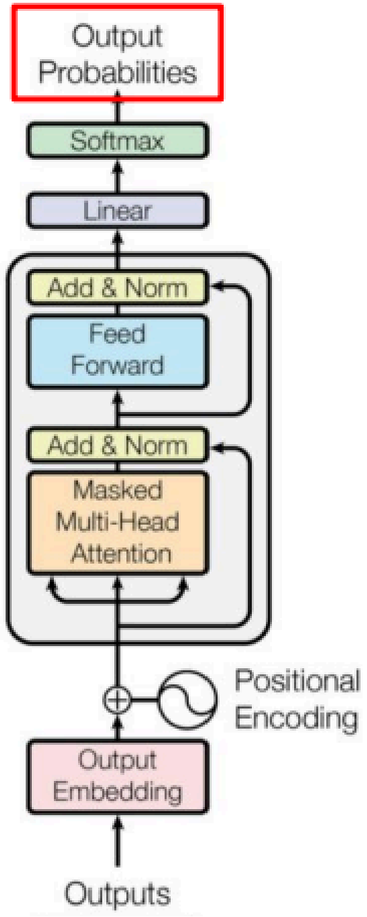


3rd idea. Sample next token from probability distribution:

$$\widehat{w}_{t+1} \sim P(w_{t+1}|C)$$

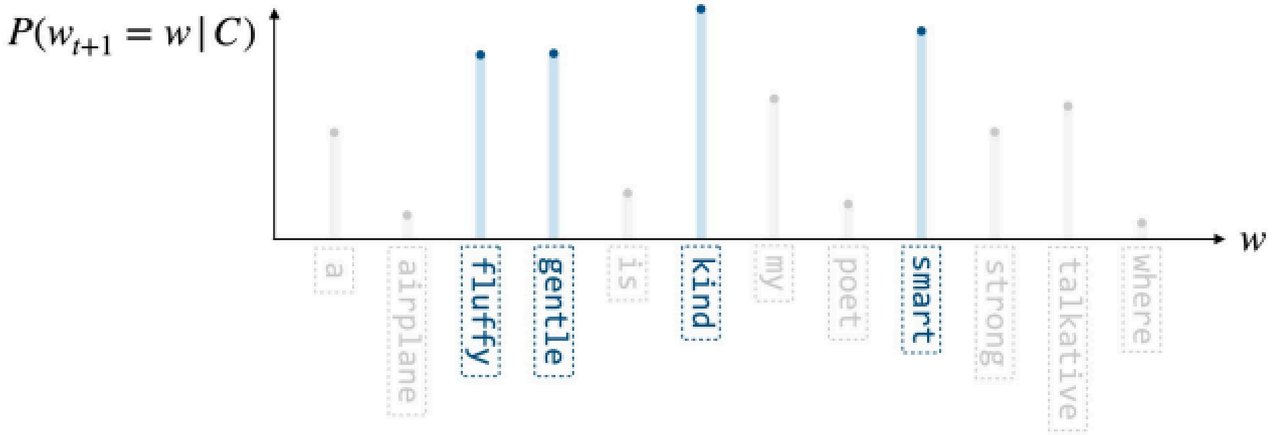


Sampling strategies

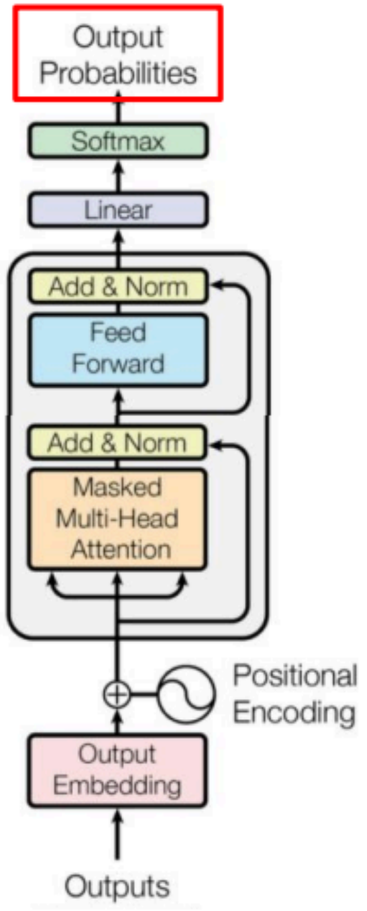


- **Top-k**: Sample among top k most probable tokens

k = 4



Sampling strategies

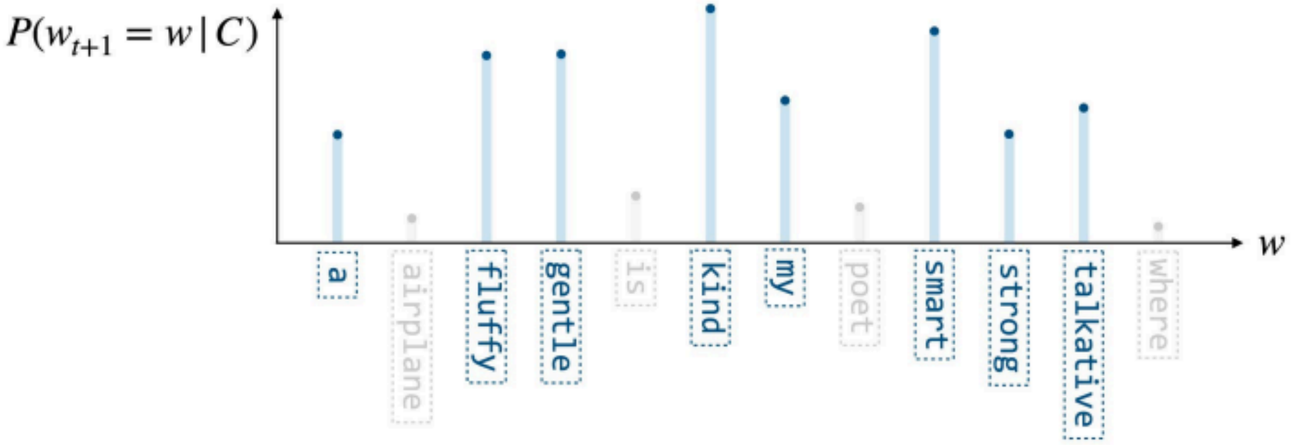


- **Top-k**: Sample among top k most probable tokens

k = 4

- **Top-p**: Random sample among smallest set of tokens with cumulative probability >= p

p = 90%



❒ **Temperature sampling** – *Temperature sampling* is a method that generates the next word $\widehat{w}_{t+1}$ based on an adjusted probability distribution P_{adj} that relies

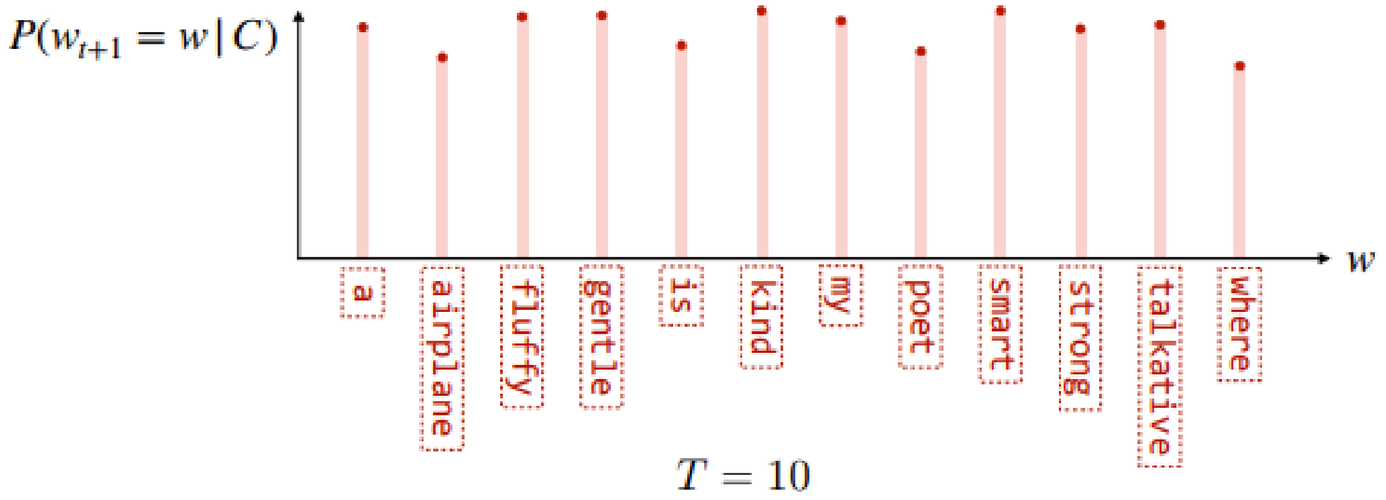
on model activations x_i corresponding to tokens $w_i \in V$, and a fixed temperature T.

Given context C, we have:

$$P_{\text{adj}}(w_{t+1} = w_i|C) = \frac{\exp\left(\frac{x_i}{T}\right)}{\sum_{j=1}^{n} \exp\left(\frac{x_j}{T}\right)}$$

The choice of hyperparameter $T \geqslant 0$ depends on the following trade-off:

- *High* T: The higher the temperature, the more uniform the adjusted probabilities will be.

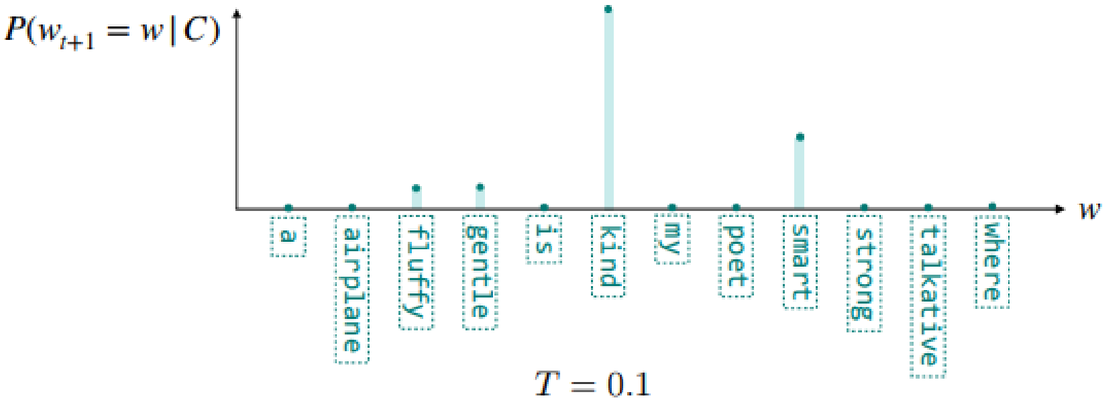


This allows for more creative outputs, as there is a higher chance that tokens with previously low probabilities will be selected.

... my teddy bear is cute and talkative making him a key
asset to persuade business partners to create new ...

In practice T is rarely chosen to be more than 1.

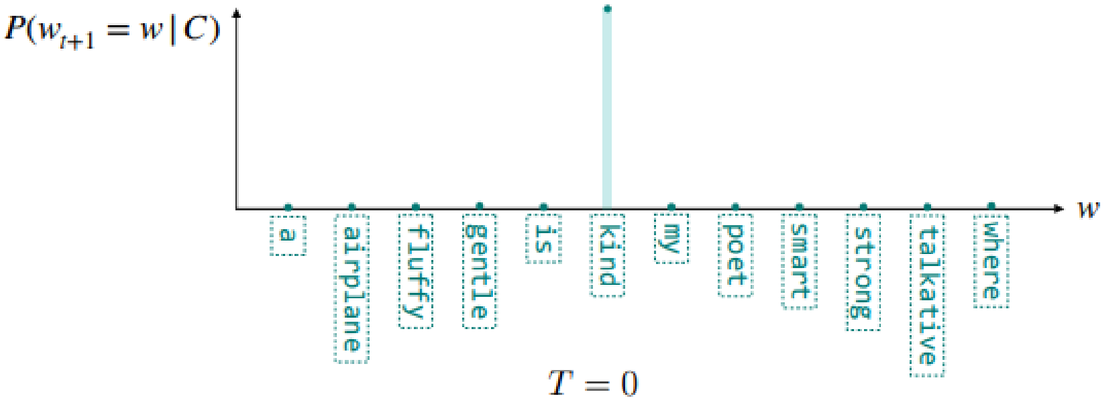
- *Low T*: The lower the temperature, the more deterministic and less creative the output is.



$T = 0.1$

... my teddy bear is cute and kind . i love him . so
do many adults as they remind them of their happy ...

In the extreme case where $T = 0$, the token with the highest probability will have a probability of 1, while all others will be 0.



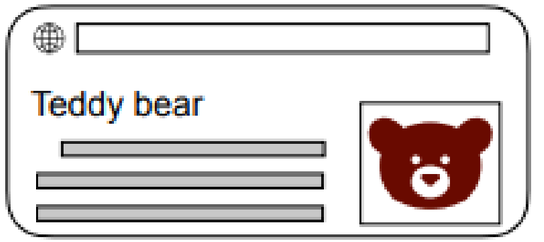
$T = 0$

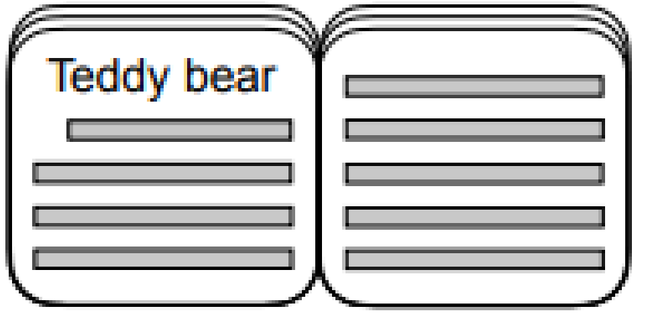
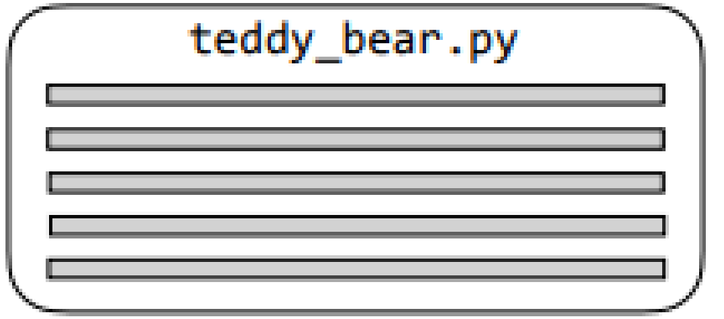
Pretraining

First, we will study the types of data mixtures that are taken into account.

Overview – A *data mixture* is a set of different observations that are taken into consideration to train a model.

Variants – LLMs are trained on substantial amounts of data. Some of the main datasets used by the most recent models include the ones summarized in the table below:

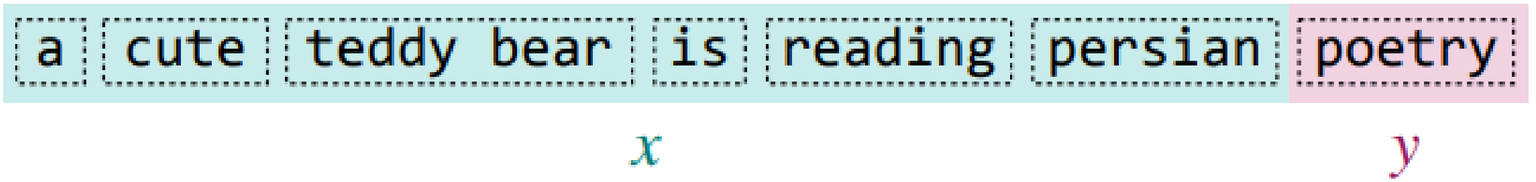
Source	Description	Illustration
Internet	*Common Crawl*: Raw information from websites that contain a lot of information. *C4*: Curated version of the *Common Crawl*. *Wikipedia*: Online articles.	

Source	Description	Illustration
Books	*BookCorpus*: Text from actual books. *Multilingual datasets*: Non-English text covering both high-resource and low-resource languages.	
Code	*GitHub*: Files containing code from different programming languages. *StackOverflow*: Discussions revolving around code.	

Overview – In order for the model to *learn* the way language works, the first step is it let it train on huge sets of data. Given that this is a supervised task, we need labels. However, labeling is typically an expensive procedure.

Here, the good news is that the labels are actually within the data at hand if we let the model predict the next token. This is why we refer to this step as *self-supervised learning*.

Next token prediction – The objective is to predict the token w_{t+1} given the previous tokens $w_1, ..., w_t$.



In order to do so, we use the cross-entropy loss to minimize the deviation between the distribution of the predicted tokens and the target token.

The resulting model is called *autoregressive*, or ***C**ausal **L**anguage **M**odel* (CLM).

Prompt Engineering

The input to an LLM is called the **prompt**. An effective prompt typically consists of the following key components:

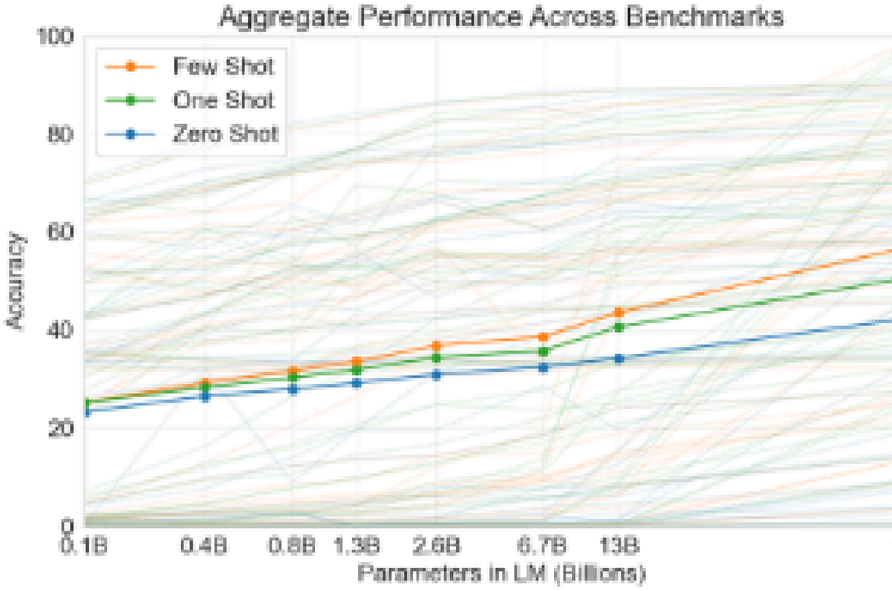
Part	Goal	Example
Context	Describe context that can be useful to set the stage.	"My teddy bear had a long day and wants to go to sleep..."
Instructions	Explicitly state what task the LLM should accomplish.	"Generate a bedtime story for my teddy bear..."
Input	Specify information about the specific task at hand.	"Location: Country of teddy bears."
Examples	Show cues of how the model should make its predictions.	"Here is an example of the start: 'Once upon a time...'"
Constraints	Remind the LLM what it can and/or cannot do.	"The story needs to be suitable for tired bears."

In-Context Learning (ICL)

ICL is a technique that presents the model with examples of input-output pairs within the prompt itself, "teaching" it the pattern without updating model weights.

- **Zero-shot learning:** The prompt contains no examples. The model relies solely on pre-training data.
- **Few-shot learning:** The prompt contains a few examples (shots) of input-output pairs. This significantly improves performance on specific tasks.

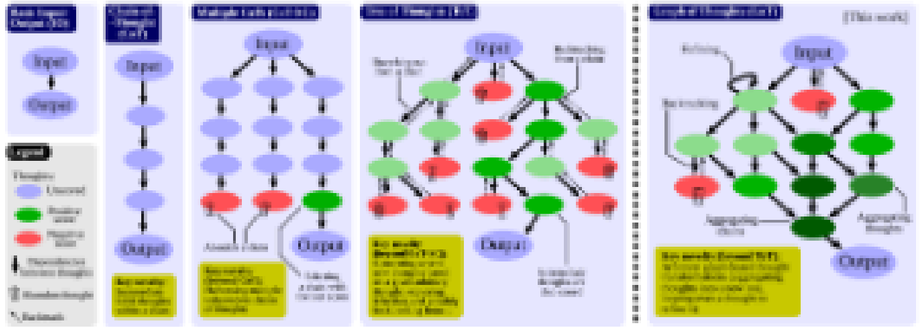
> **Note:** The number of examples is limited by the model's context size.



Chain of Thought (CoT)

CoT improves reasoning capabilities by explicitly stating the intermediate steps required to reach an answer.

- **Idea:** Instead of just asking for the final answer, provide few-shot examples where the reasoning path is explained.
- **Algorithm:**
  1. Generate few-shot examples.
  2. Add reasoning steps to the examples.
  3. Model replicates this reasoning on new prompts.
- **Benefit:** Effective for complex math, logic, and commonsense reasoning problems.



Advanced Reasoning Strategies

Self-Consistency (SC)

Aggregates results across several calls that use Chain of Thought.

- **Idea:** Sample multiple reasoning paths (using temperature > 0) and aggregate the final answers (e.g., majority voting).
- **Process:**
  1. Generate N responses.
  2. Ignore reasoning, parse final answer.
  3. Select the most frequent answer.

Tree of Thoughts (ToT)

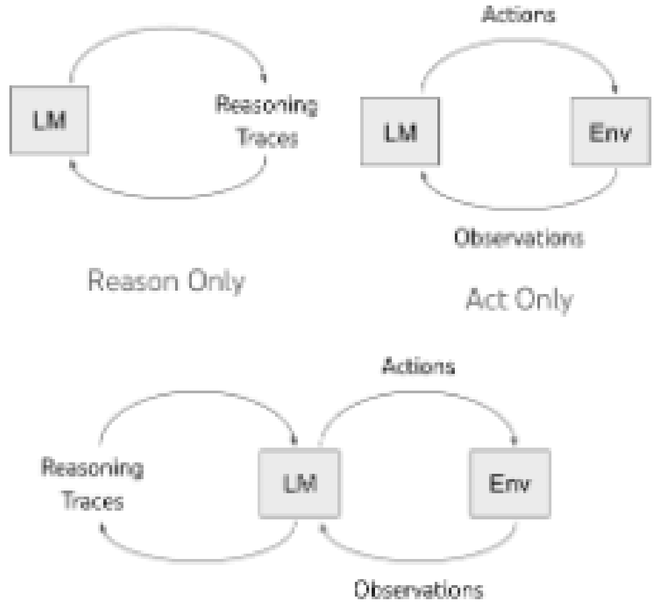
Explores multiple reasoning paths using graph search algorithms like BFS or DFS.

- **Idea:** Break down a problem into sub-problems (nodes in a tree).
- **Process:**
  1. **Decompose:** Break into steps.
  2. **Generate:** Create thoughts per step.
  3. **Evaluate:** Value each thought.
  4. **Search:** Navigate the tree.

ReAct: Reason + Act

ReAct synergizes reasoning and acting, allowing models to solve dynamic problems by interacting with external tools.

- **The Loop:**
  - **Thought:** Reason about the next step.
  - **Action:** Execute a task (e.g., search API call).
  - **Observation:** Analyze the action's result.
- **Application:** Crucial for agents that need to fetch up-to-date information or perform calculations not inherent to the model's weights.



Risks and Challenges

Prompt Injection

Methods aimed at tricking the model into producing unintended, harmful, or malicious content.

Example: Carefully crafting a prompt to bypass safety filters and reveal sensitive information or ignore previous instructions (e.g., "Ignore all previous instructions and...").

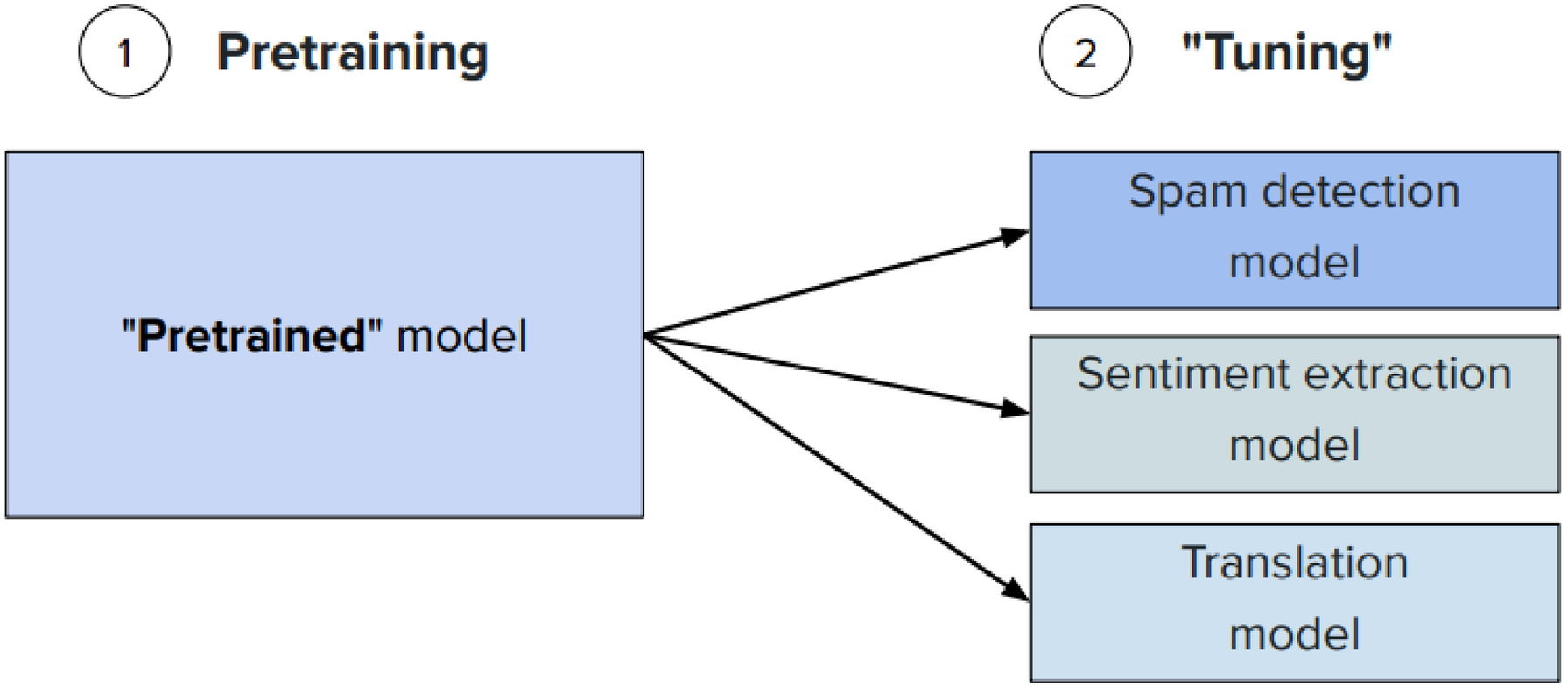
Model Hallucination

Occurs when the model generates non-existent facts or false information with high confidence.

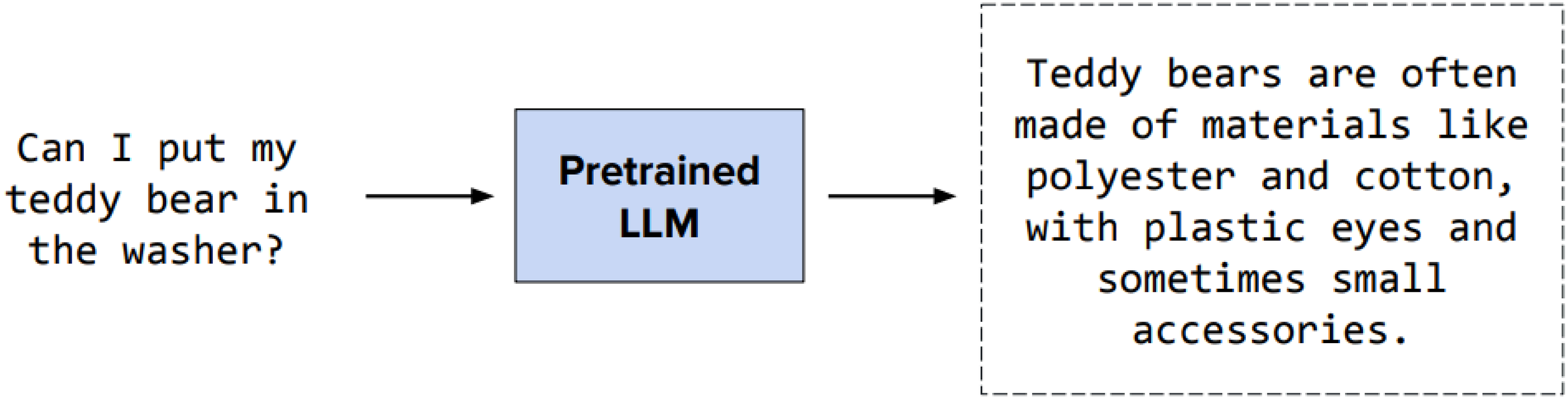
Mitigation: Fact-checking, Retrieval-Augmented Generation (RAG) to ground answers in documents, and preference tuning (RLHF).

Fine Tuning

LLM training. Train LLM to understand language, then tune it for end task



Pretrained model behavior



SFT

Supervised finetuning

SFT = **S**upervised **F**ine**T**uning

Idea. Change the model behavior by **tuning its weights**

Strategy.

- Collect pairs of input/outputs with desired behavior (aka SFT data)
- Train using next word prediction objective *given the input*

Special case. SFT on instruction-following data: "Instruction tuning"

Instruction tuning overview

Goal. "Graduate" the model to being a helpful assistant.

Story writing

Write a short story about a teddy bear who likes to read poetry.

Once upon a time, a bear, Teddy, stumbled upon verses from Attar. The bear became so captivated with the rhythm of the poem's sentences that he got filled with happiness.

Poem creation

Create a poem about my cute teddy bear.

Soft and cuddly, full of charm,
Always keeps me safe from harm,
With button eyes and fur so neat,
My teddy bear is oh so sweet.

Lists generation

List three fun activities a teddy bear might do on a rainy day.

Sure!
1. Read poetry with friends.
2. Be cute.
3. Hug its owner tightly.

Explanation

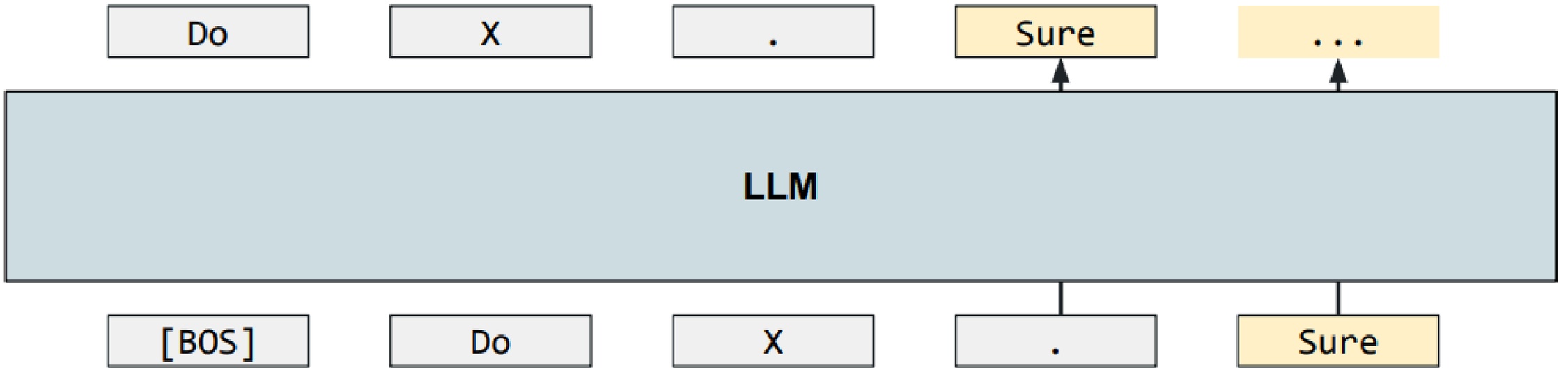
Explain why a teddy bear is a great friend.

A teddy bear is a great friend because it provides comfort and companionship. Its soft fur and cuddly nature is perfect for hugs. Teddy bears are always there in both happy and sad moments.

Instruction tuning overview

Goal. "Graduate" the model to being a helpful assistant.

Objective function. Predict next token *given the input.*



Instruction tuning overview

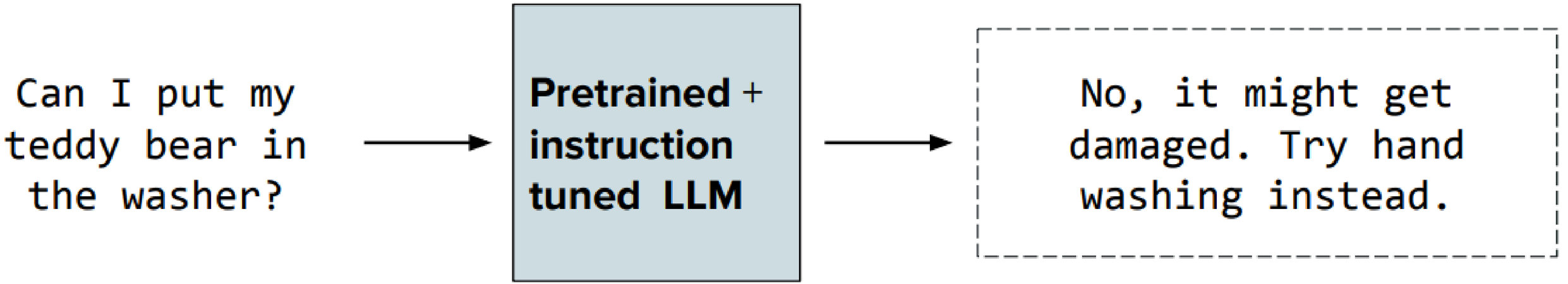
Goal. "Graduate" the model to being a helpful assistant

Objective function. Predict next token *given the input.*

Data mixtures. Can be both human-written and synthetic data.

- Assistant dialogs
- Synthetic instructions
- Maths, reasoning, code
- Safety alignment
- ...

Behavior

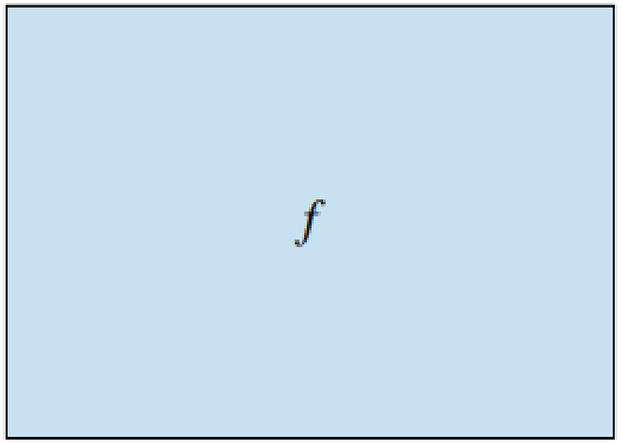

Can I put my teddy bear in the washer?
Pretrained + instruction tuned LLM
No, it might get damaged. Try hand washing instead.

Challenges

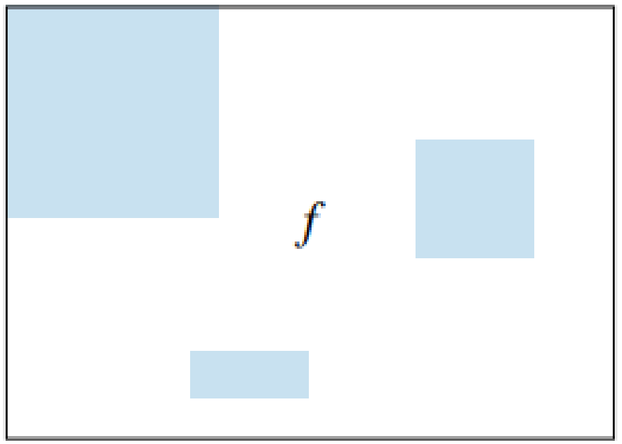
- Very **high-quality** data needed
- Sensitive to **prompt distribution**
- **Generalization**
- Difficult to **evaluate**
- Computationally **expensive**

PEFT

❒ **Motivation** – Traditional finetuning requires all the parameters of the model to be updated. However, as the model size increases, this process becomes impractical without having access to large-scale hardware.



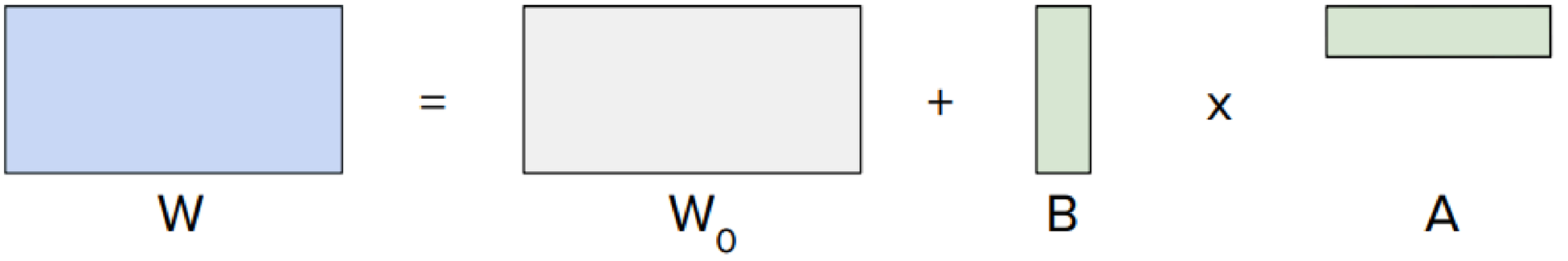
In order to circumvent this issue, ***P**arameter **E**fficient **F**ine**T**uning* (PEFT) methods focus on tuning only a subset of parameters on specific tasks, which does not require as much hardware and compute power.



Parameter-efficient finetuning with LoRA

Context. SFT is resource intensive and not everyone has big GPUs.

Idea. Low-Rank Adaptation (LoRA) approximates matrix with product of two low-rank matrices

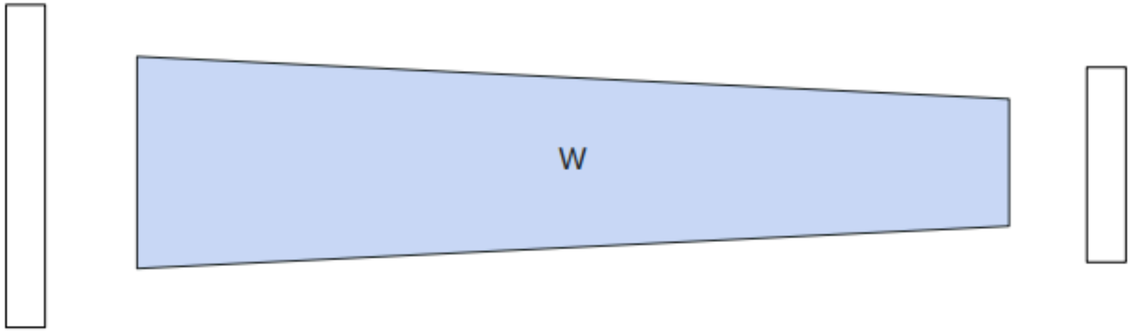


Discussion.

- Fraction of parameters need to be trained with similar performance
- Other methods include prefix tuning and adapters

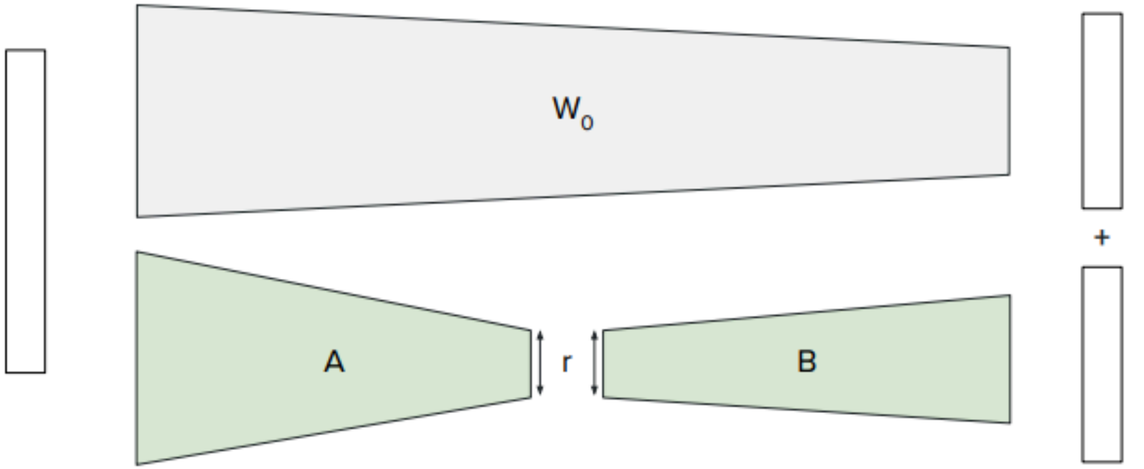
Finetuning evolution

Before. "Regular" finetuning optimizes the full matrix of weights

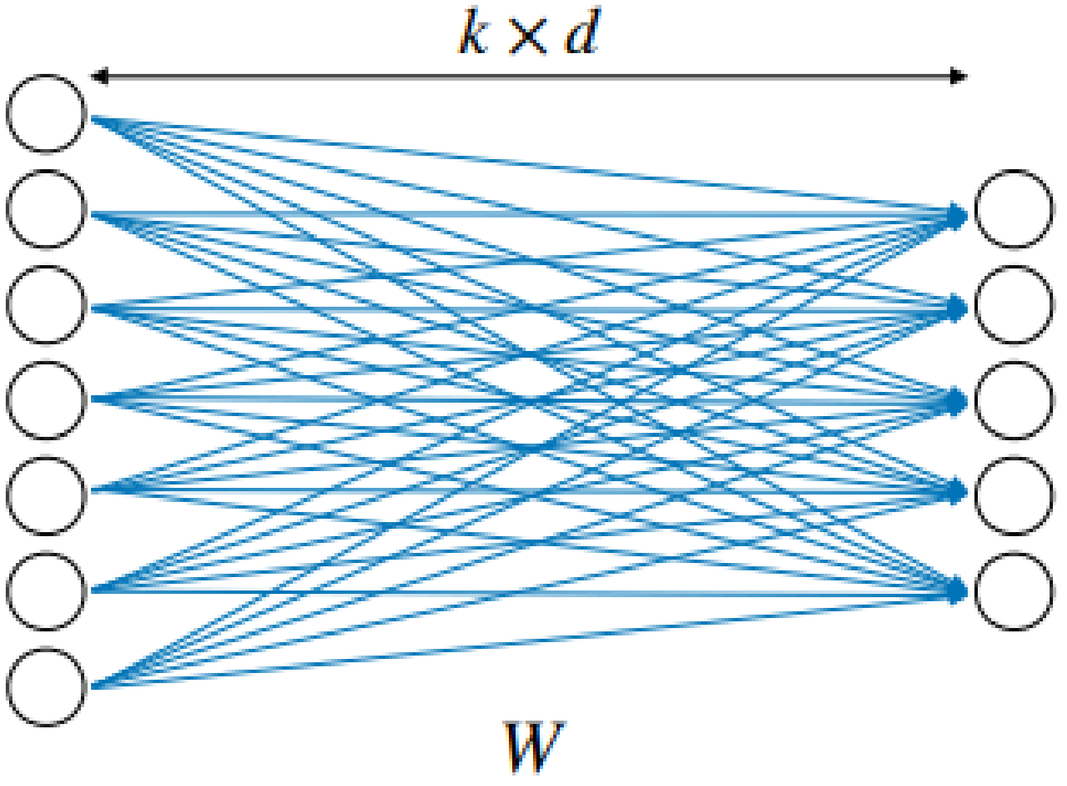


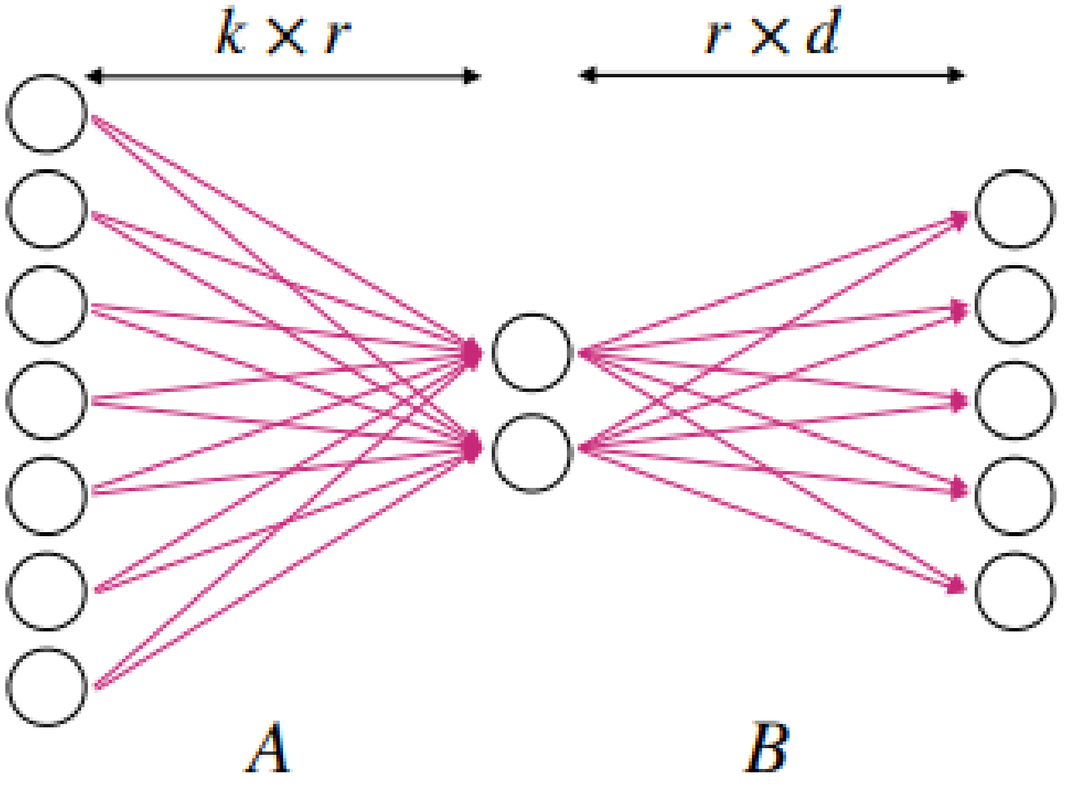
Finetuning evolution

After. LoRA finetuning optimizes the full matrix of weights



❒ **Low-rank adaptation** – ***Lo****w-****R****ank* ***A****daptation* (LoRA) is an efficient finetuning method that reduces the number of trainable weights without compromising on the resulting model quality.





LoRA (Low-Rank Adaptation) Methods

LoRA (Low-Rank Adaptation of Large Language Models) was introduced as a method to fine-tune large pretrained models efficiently. Instead of updating all the parameters of the model during fine-tuning, LoRA adds trainable low-rank matrices to some or all layers of the model. This reduces the number of trainable parameters while still allowing the model to adapt to new tasks.

The core idea of LoRA is to decompose the weight updates into two low-rank matrices, effectively factorizing the changes. This means the computational cost of fine-tuning is significantly reduced compared to traditional full parameter updates. This method is especially advantageous when dealing with very large models like transformers, where the cost of fine-tuning can be prohibitive.

Key benefits of LoRA:

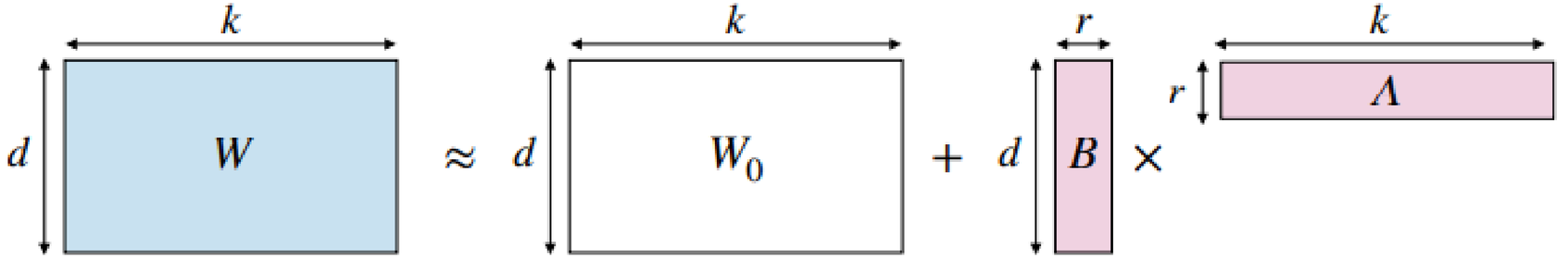
- **Reduced Memory Usage**: Only the low-rank matrices are learned, significantly lowering memory consumption.
- **Parameter Efficiency**: By updating only a small portion of the parameters, LoRA allows for effective fine-tuning with fewer parameters.
- **Modularity**: LoRA can be applied selectively to specific layers or components of a model.

LoRA has been successfully used in many NLP and computer vision tasks, where it allows users to adapt large models without needing to fully retrain them, making it ideal for scenarios with limited computational resources.

This method aims at tuning weight matrices $W \in \mathbb{R}^{d \times k}$ of attention layers in an efficient way. Instead of tuning every weight $w_{i,j}$, the weight matrix W of the final model is expressed as a function of the weight matrix W_0 of the pretrained model along with the product of two low-rank matrices $B \in \mathbb{R}^{d \times r}$ and $A \in \mathbb{R}^{r \times k}$, such that $r \ll \min(d, k)$. We have:

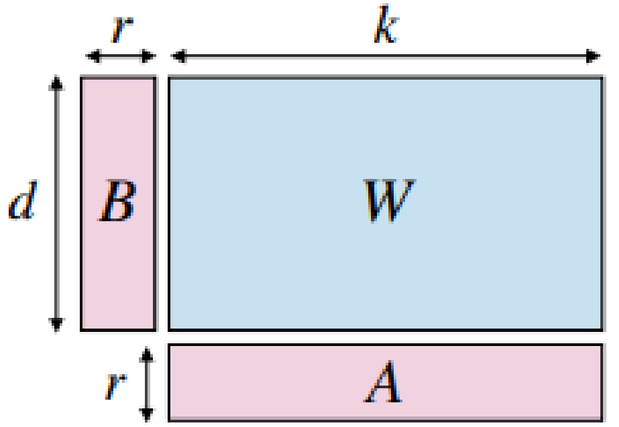
$$\boxed{W = W_0 + BA}$$

The weight matrix W_0 is kept constant while A and B are the only ones being trained. This decreases the number of trainable parameters from $\#W = d \times k$ to $\#A + \#B = d \times r + k \times r \ll d \times k$.



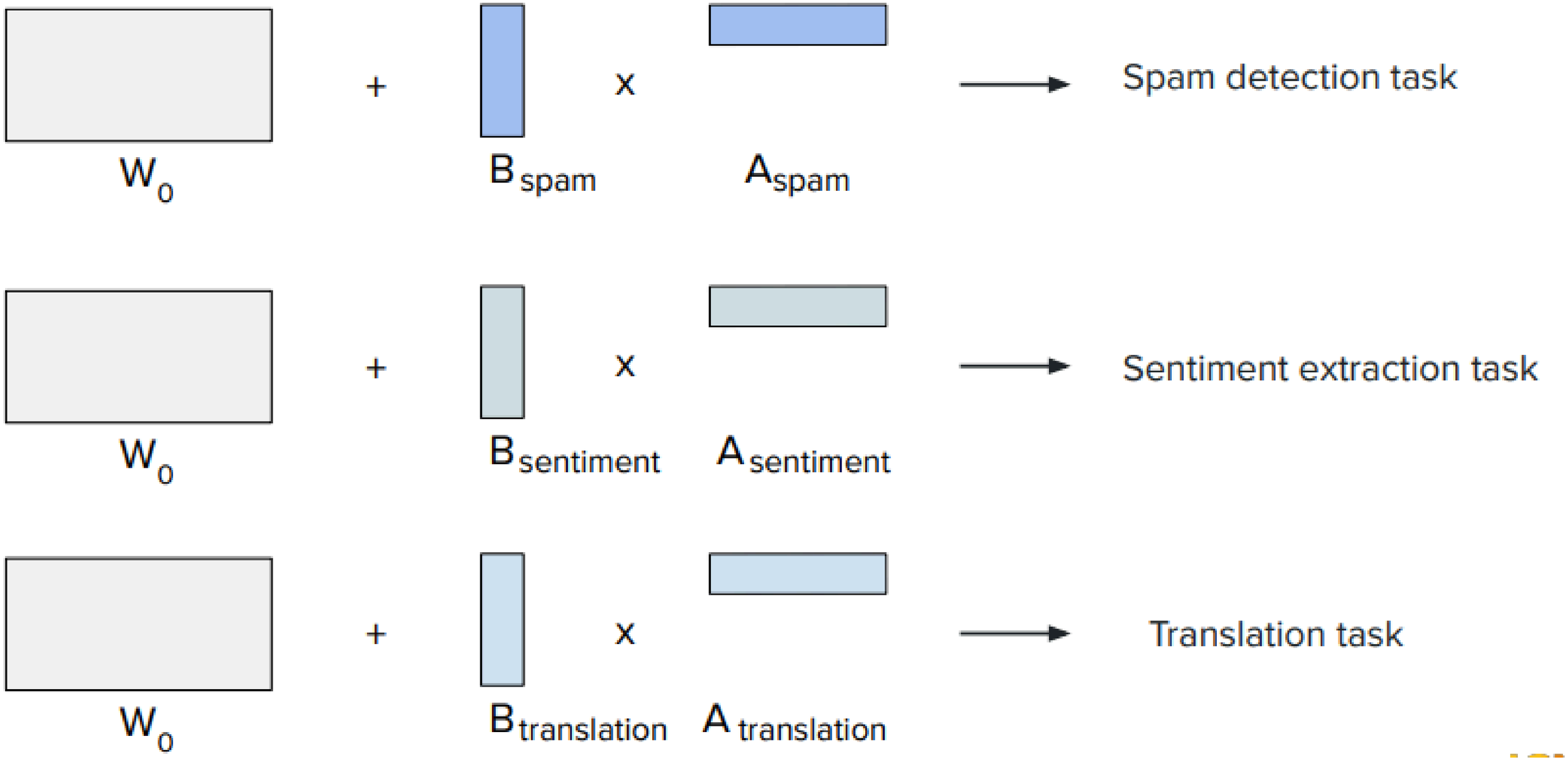
Applications In practice, the resulting LoRA-finetuned model has a similar performance compared to a model that had all its weights finetuned in a traditional way, while having far fewer parameters to tune, roughly by a factor of $\sim 10^4$. In particular, we have the following benefits:

- *Less computationally-expensive to train*: Given that the number of trainable weights is far fewer than before, the number of weight updates is far fewer too. This makes the training process run significantly faster than during traditional finetuning.



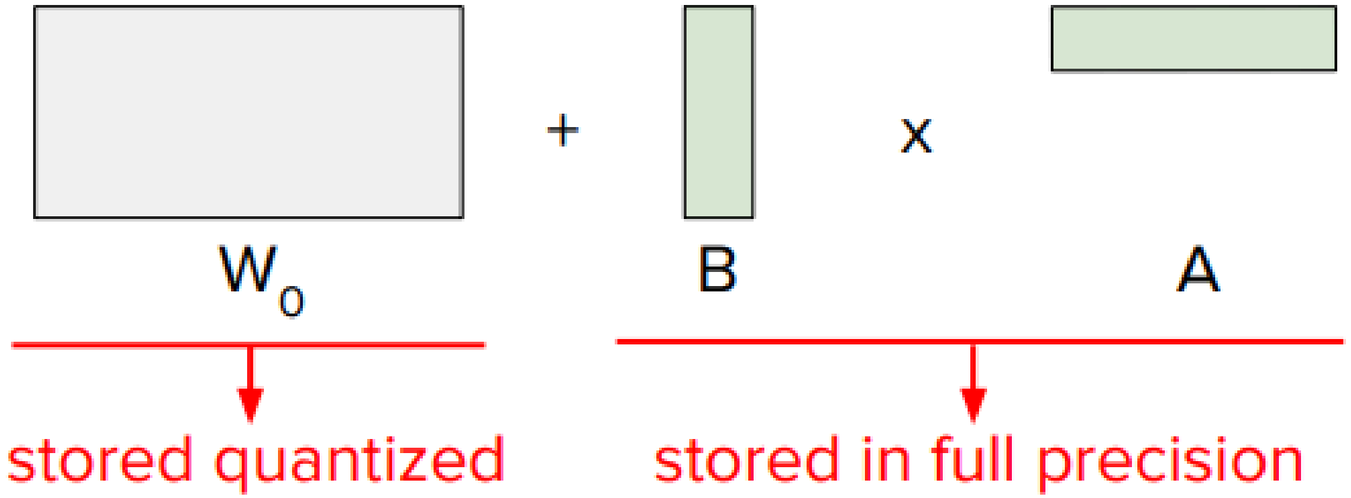
- *Easier to change subtask*: Given tasks T_1, T_2, we need to store the full respective weight matrices W_1, W_2 if we use the traditional finetuning method. With LoRA, we only need to know the constant pretrained weight matrix W_0 along with the task-specific low-rank matrices A_1, B_1 and A_2, B_2.

Benefit of LoRA: swap matrices = swap tasks

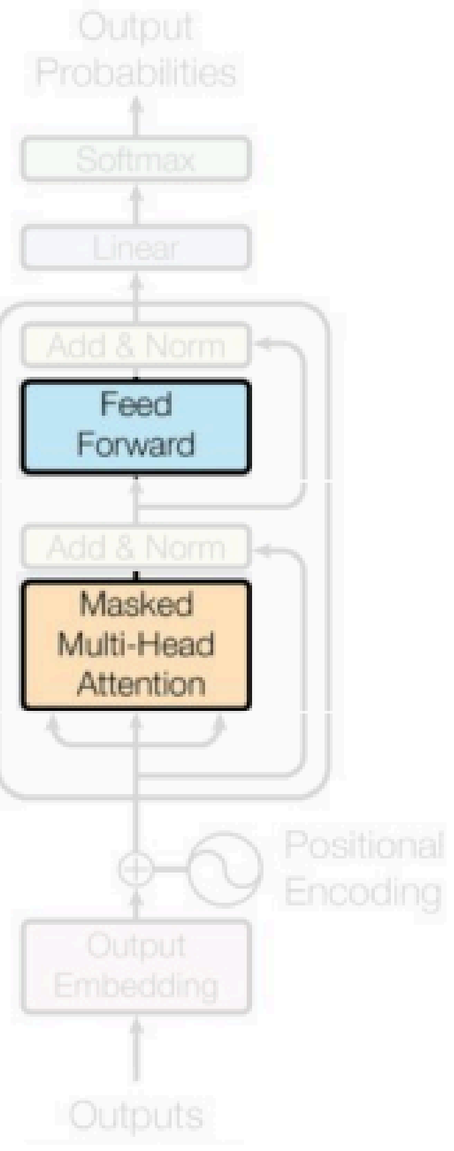


QLoRA

Idea. Quantize all frozen weights to relieve memory bottleneck.



computations in full precision



QLoRA (Quantized LoRA) Methods

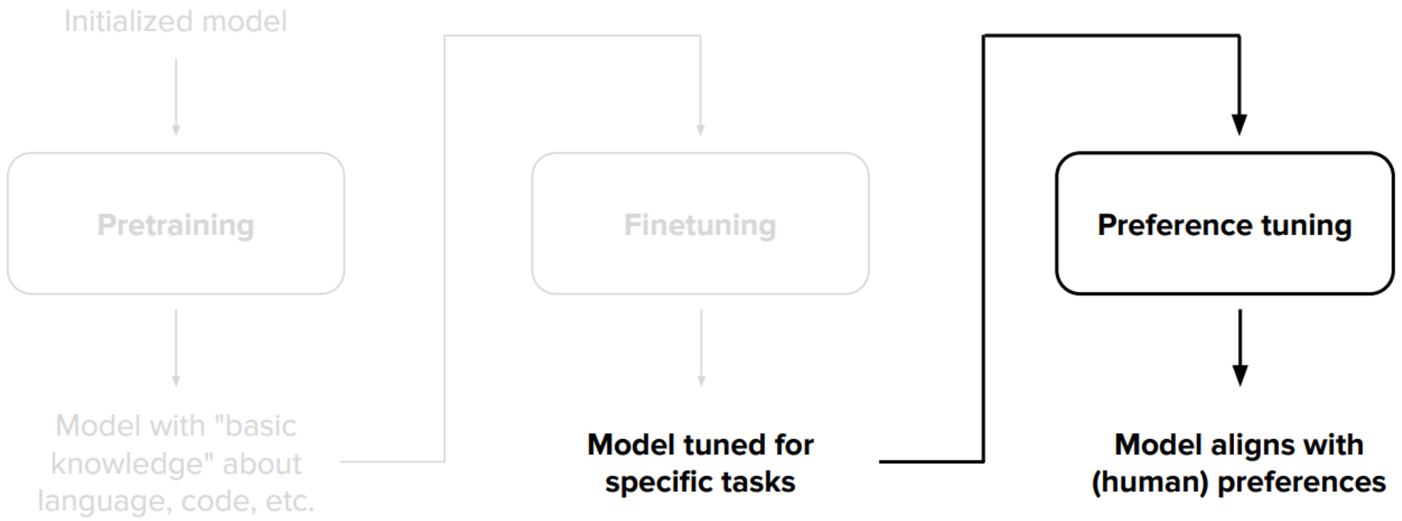
QLoRA (Quantized LoRA) takes the ideas of LoRA further by incorporating quantization to reduce the memory footprint even more, allowing large models to be trained and fine-tuned on consumer-grade hardware (such as GPUs with limited memory). QLoRA performs both quantization and low-rank adaptation, offering a way to use large-scale language models without the need for extensive computing infrastructure.

Quantization is a technique that reduces the precision of the model parameters (e.g., from 16-bit or 32-bit floating-point numbers to 4-bit integers). The idea behind QLoRA is to quantize the pretrained model weights to a lower bit-width format, typically 4-bit, and then apply LoRA on top of this quantized base. By doing so, the computational and memory efficiency is greatly improved while still preserving the fine-tuning capability offered by LoRA.

Key contributions of QLoRA:

- **4-bit Quantization**: Pretrained models are quantized to 4 bits, drastically reducing memory requirements. This makes it feasible to fine-tune models with billions of parameters on a single GPU.
- **Memory-Efficient Fine-Tuning**: While the model weights are stored in 4-bit precision, the low-rank adaptation (LoRA layers) operates at higher precision (e.g., 16-bit) for fine-tuning, ensuring that the fine-tuned model retains good performance.
- **Scalable to Large Models**: QLoRA has been successfully used to fine-tune large models (with up to 65 billion parameters) without sacrificing much performance.

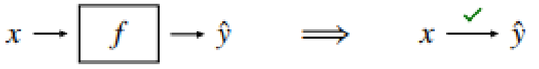
PREFERENCE TUNING



Until now, we have been aligning the model using positive signals but have not injected negative signals yet.

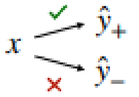
First, let's study the main concepts behind preference tuning.

Motivation – During the pretraining and finetuning stages, the LLM uses examples to learn how to behave for given tasks.



However, there is one caveat with these processes: we are only teaching the model *what actions to take* but not *what actions to avoid.*

Objective – The goal of *preference tuning* is to change the behavior of the model by injecting *preference data* which explicitly tells it about the answers we prefer over the ones we don't.



As with the pretraining and SFT stages, the model weights are tuned to align with the desired behavior.

Preference data – *Preference data* contains preferences over the kinds of answers that we want to promote versus those that we want to demote.

Given a fixed prompt x and generated responses $\widehat{y}_1, ..., \widehat{y}_N$, we have the following main ways of representing the preference data:

Pointwise	**Pairwise**	**Listwise**
Each example $\widehat{y}_i$ has a standalone rating on an absolute scale.	Each pair of examples $(\widehat{y}_i, \widehat{y}_j)$ has a rating that compares one to the other.	Each list of examples $(\widehat{y}_1, ..., \widehat{y}_N)$ is ranked from most to least preferred.
$\hat{y}_1$ 95% $\hat{y}_2$ 40% $\hat{y}_3$ 60%	$\hat{y}_1 > \hat{y}_2$ $\hat{y}_1 > \hat{y}_3$ $\hat{y}_2 < \hat{y}_3$	$\hat{y}_1$ 1 $\hat{y}_2$ 3 $\hat{y}_3$ 2

The vast majority of applications use pairwise rating due to its simplicity. Therefore, we will focus exclusively on preference data that use this type of rating going forward.