Running and fine tuning LLM:s locally and using custom data

Ivan Reznikov
@qburst @mdxdubai

Shameless Self-Promotion

- PhD in Computational Sciences
- 12+ years of Python and Data Science experience
- Worked for small/medium/large enterprise companies, startups
- Principal Data Scientist at QBurst
- Kaggle Competition Expert
- TEDx Speaker (2017), GITEX/PyCON(2021)
  CoderHQ(2022, 2023)

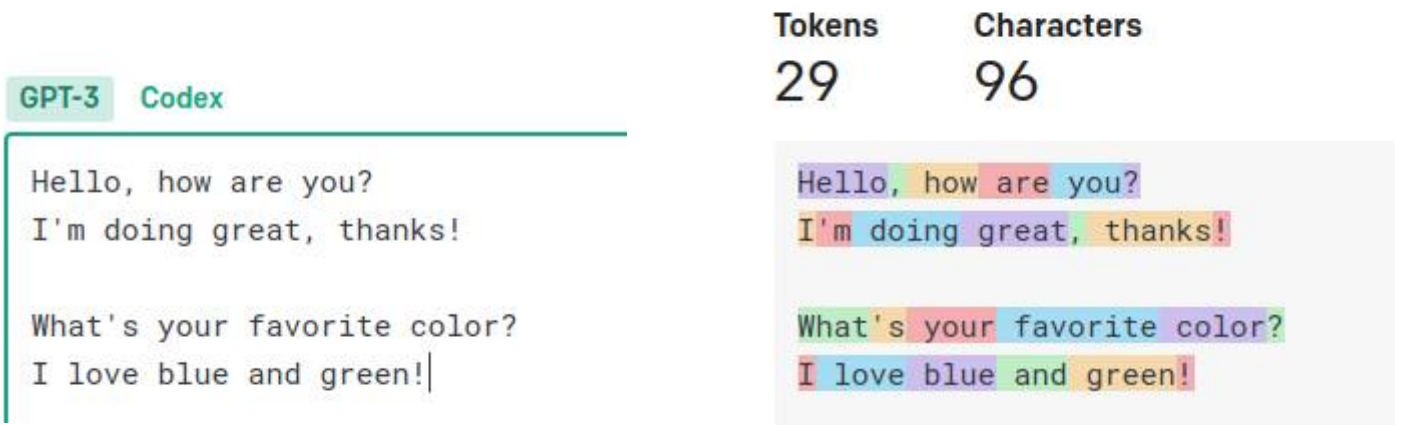
https://www.linkedin.com/in/reznikovivan/
https://github.com/IvanReznikov

Roadmap

- Text generation
- Models
- APIs
- Embeddings
- Chat Models and LLMs
- Running locally
- Finetuning
- Have ~~pain~~ fun with code

Tokenizers

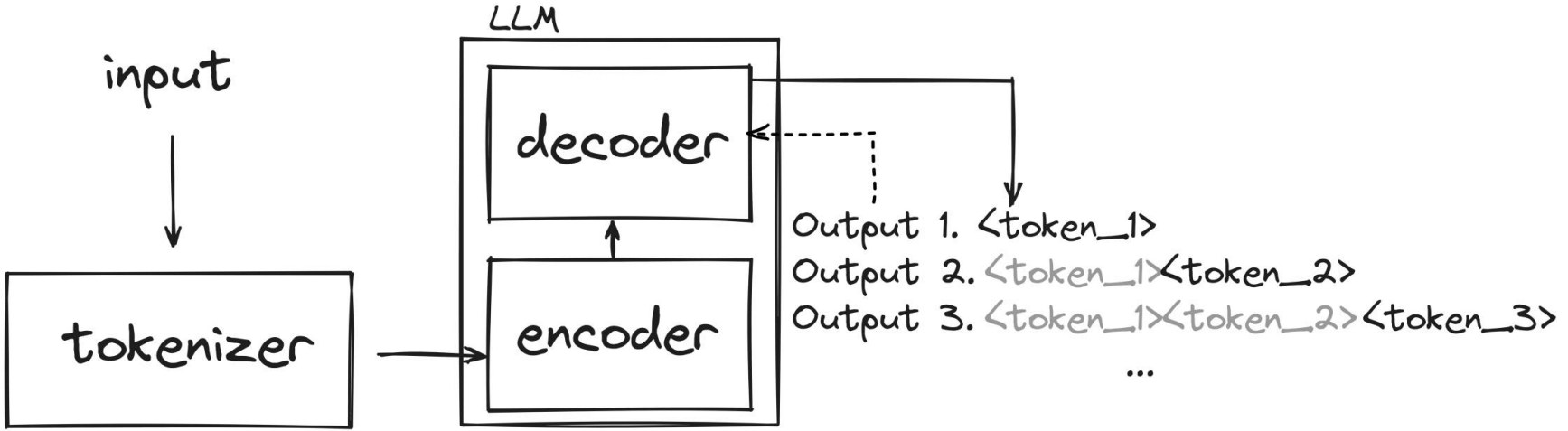
Tokenization is breaking down human language into smaller units like words or subwords to convert text into machine-readable numerical representations.



https://platform.openai.com/tokenizer

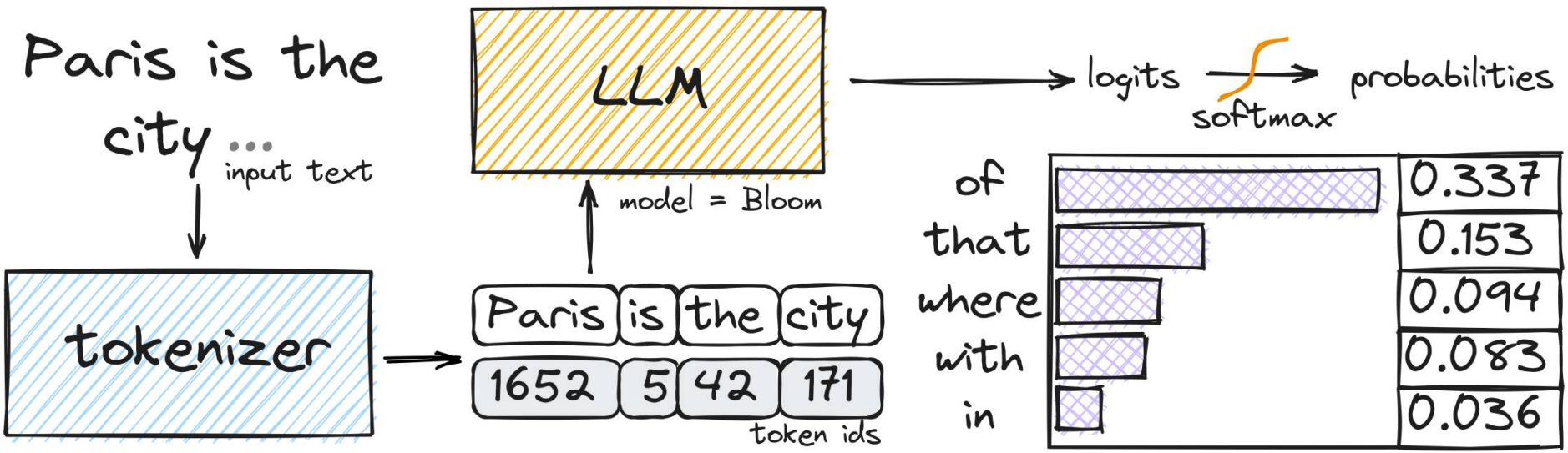
Three main steps for text generation

- The input text is passed to a tokenizer that generates unique numerical representation of every token.
- The tokenized input text is passed to the Encoder part of the pre-trained model. The Encoder processes the input and generates a feature representation that encodes inputs meaning and context.
- The Decoder takes the feature representation from the Encoder and starts generating new text based on that context token by token.



Decoding the outputs

Say, we want to generate the continuation of the phrase "*Paris is the city ...*".
The Encoder (we'll be using Bloom-560m model) sends logits for all the tokens we have (if you don't know what logits are — consider them as scores) that can be converted, using softmax function, to probabilities of the token being selected for generation.

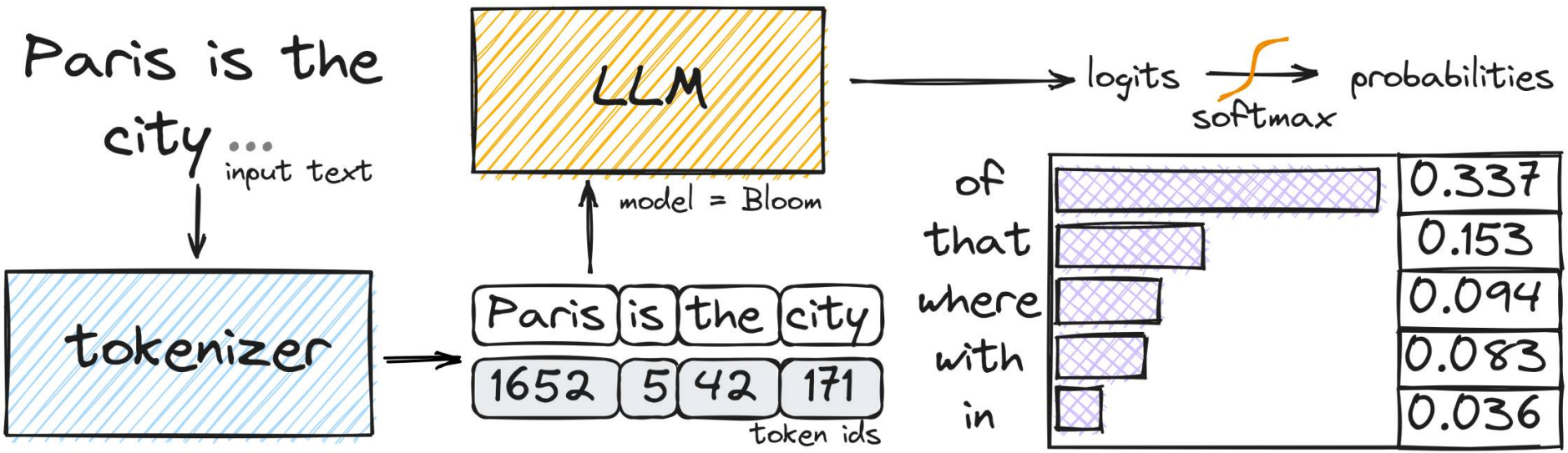


Possible following tokens

Paris is the city **of** love.

- Paris is the city **that** never sleeps.
- Paris is the city **where** art and culture flourish.
- Paris is the city **with** iconic landmarks.
- Paris is the city **in** which history has a unique charm.

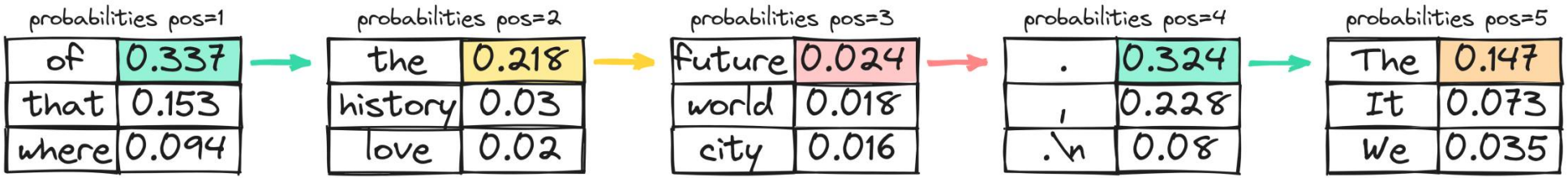
How to choose the next token?



1. Greedy Sampling

In the greedy strategy, the model always chooses the token it believes is the most probable at each step — it doesn't consider other possibilities or explore different options. The model selects the token with the highest probability and continues generating text based on the selected choice.

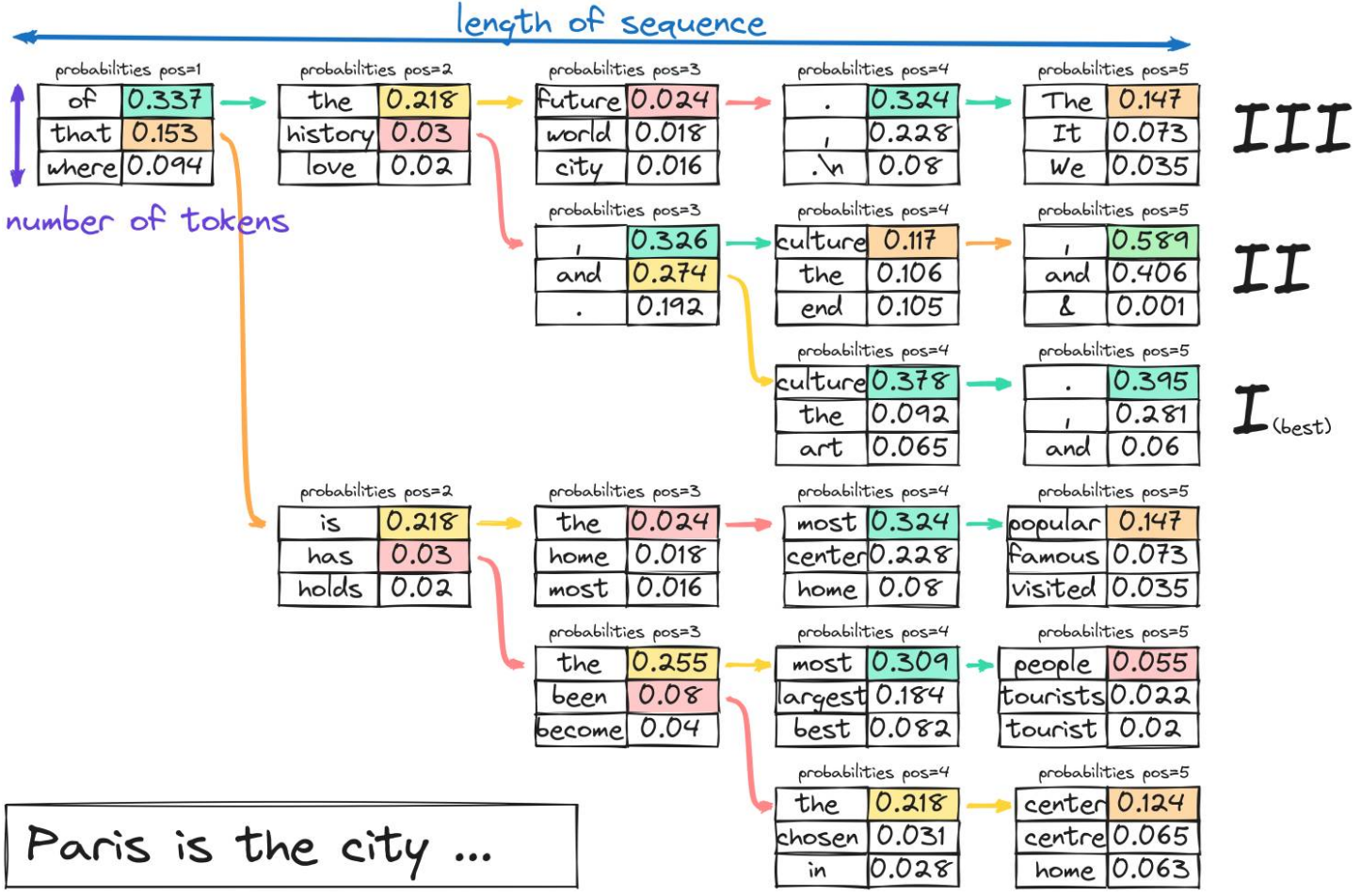
Generated output: Paris is the city of the future. The



2. Beam search

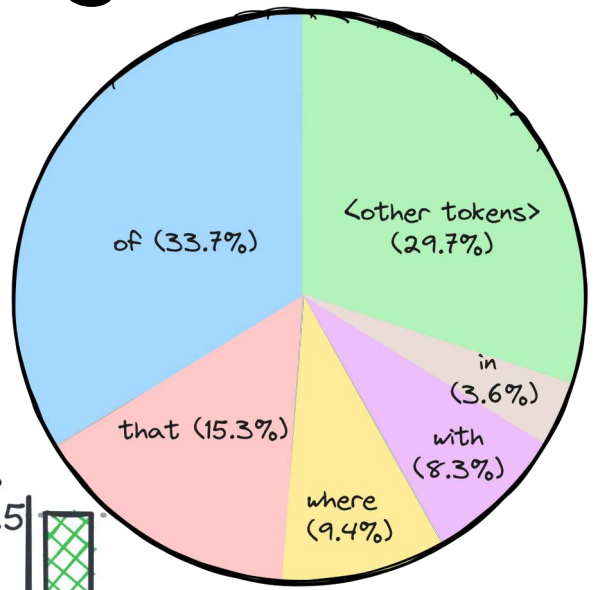
In beam search, instead of just considering the most likely token at each step, the model considers a set of the top "k" most probable tokens. This set of k tokens is called a "beam."

Generated output: Paris is the city of history and culture.



3. Normal Random Sampling

In normal random sampling you select the next word by choosing a random value and mapping it to the token got picket. Imagine it as spinning a wheel, where the area of each token is defined by its probability. The higher the probability — the more chances the token would get selected.

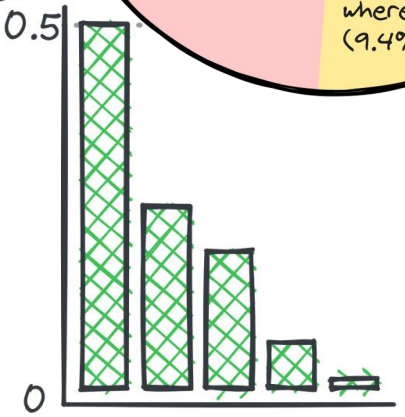


	logits
token A	-1.806
token B	-2.499
token C	-2.786
token D	-3.885
token E	-5.494

softmax →

probabilities
0.494
0.247
0.185
0.062
0.012

$$softmax(x_i) = \frac{e^{x_i}}{\sum_i e^{x_i}}$$

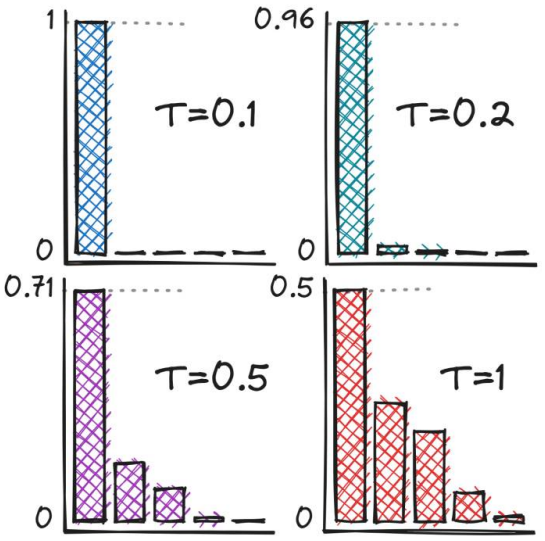


4. Random Sampling + Temperature

We've been using the softmax function to convert logits to probabilities. We now introduce temperature — a hyperparameter that affects the randomness of the text generation. The difference with the classic softmax is in the denominator - we divide by T. Higher values of temperature (e.g., 1.0) make the output more diverse, while lower values (e.g., 0.1) make it more focused and deterministic. In fact, T = 1 will lead to the softmax function we used initially.

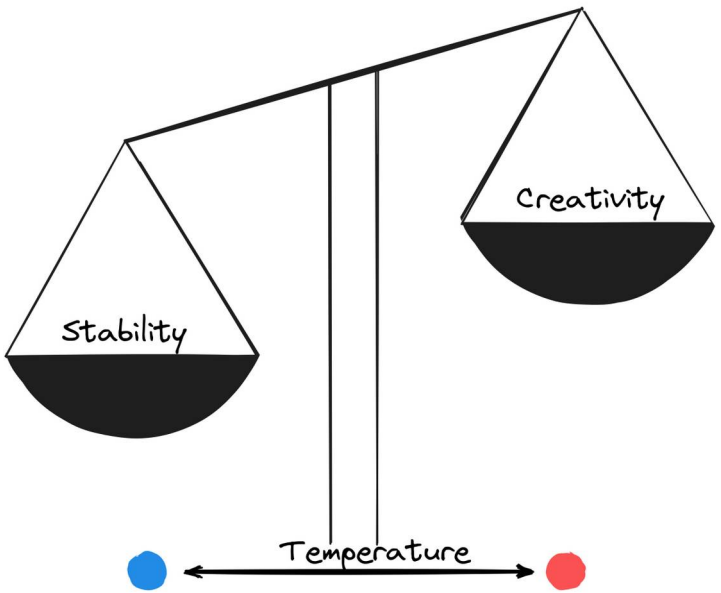
	logits
token A	-1.806
token B	-2.499
token C	-2.786
token D	-3.885
token E	-5.494

$$softmax(x_i) = \frac{e^{x_i/T}}{\sum_i e^{x_i/T}}$$



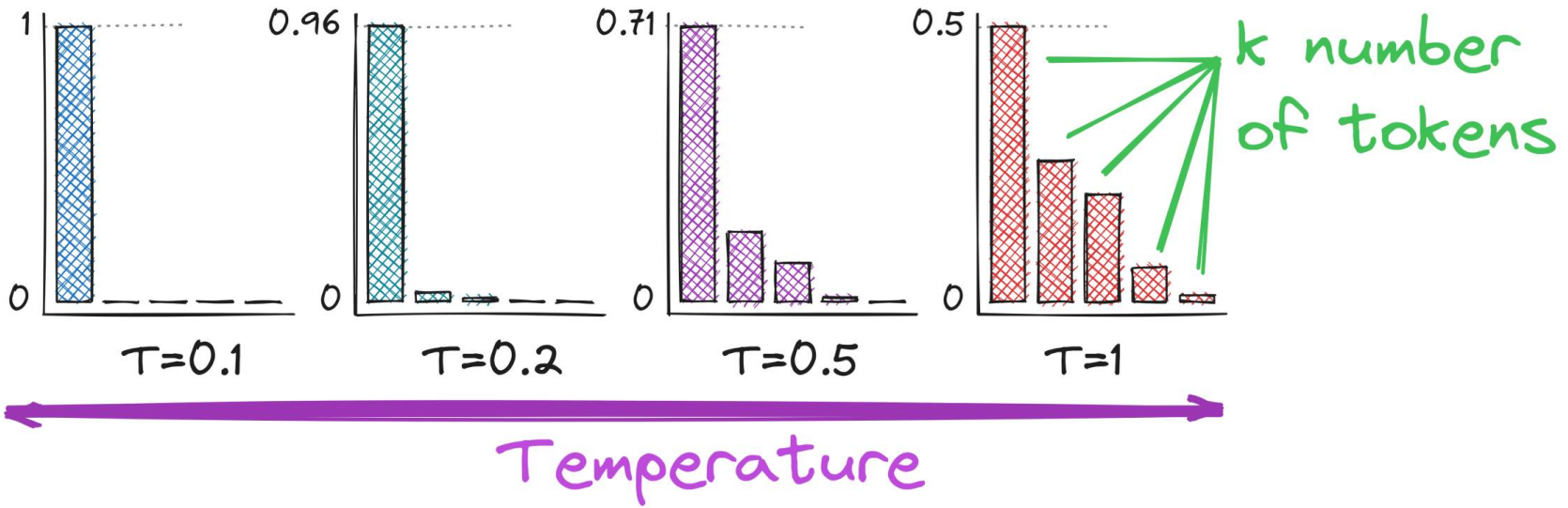
Temperature

Temperature controls the randomness of token selection during decodingHigher temperature boosts creativity, whereas lower temperature is more about coherence and structure. While embracing creativity allows for fascinating linguistic adventures, tempering it with stability ensures the elegance of the generated text.



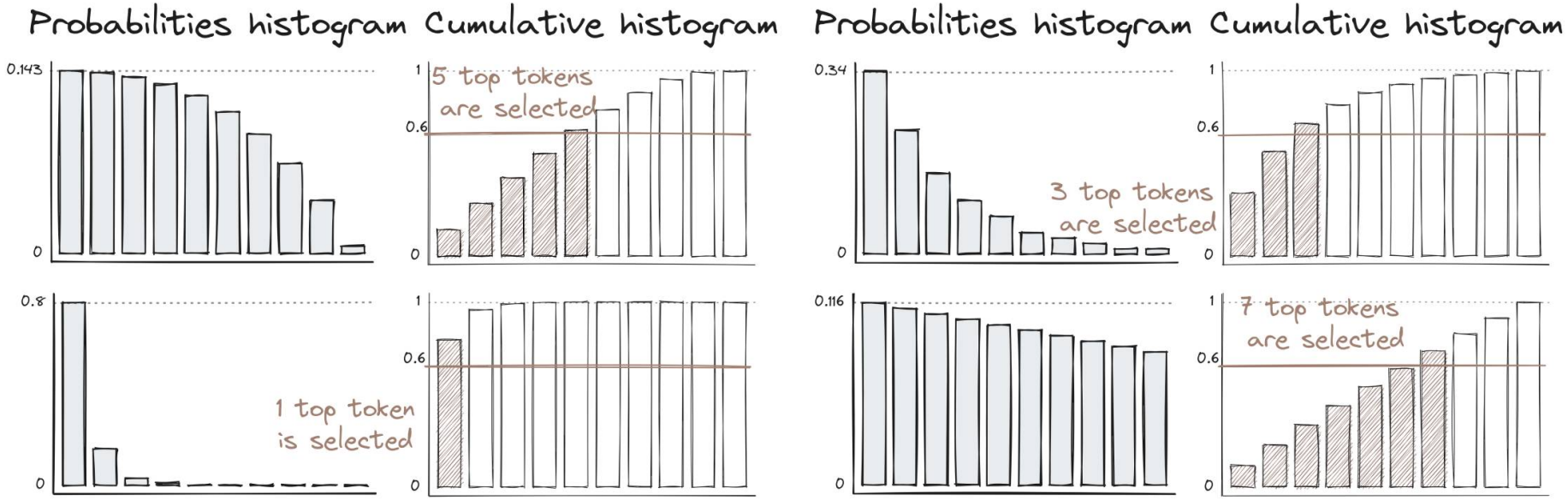
5. Top-k sampling

We can now shift probabilities with temperature. Another enhancement is to use top-k tokens rather than all of them. This will increase the stability of the text generation, not decreasing creativity too much. Basically, it's now random sampling with temperature for only top k tokens.



6. Top-p or nucleus sampling

Nucleus sampling is designed to address some limitations of different sampling techniques. Instead of specifying a fixed number of "k" tokens to consider, a probability threshold "p" is used. This threshold represents the cumulative probability that you want to include in the sampling. The model calculates the probabilities of all possible tokens at each step and then sorts them in descending order.



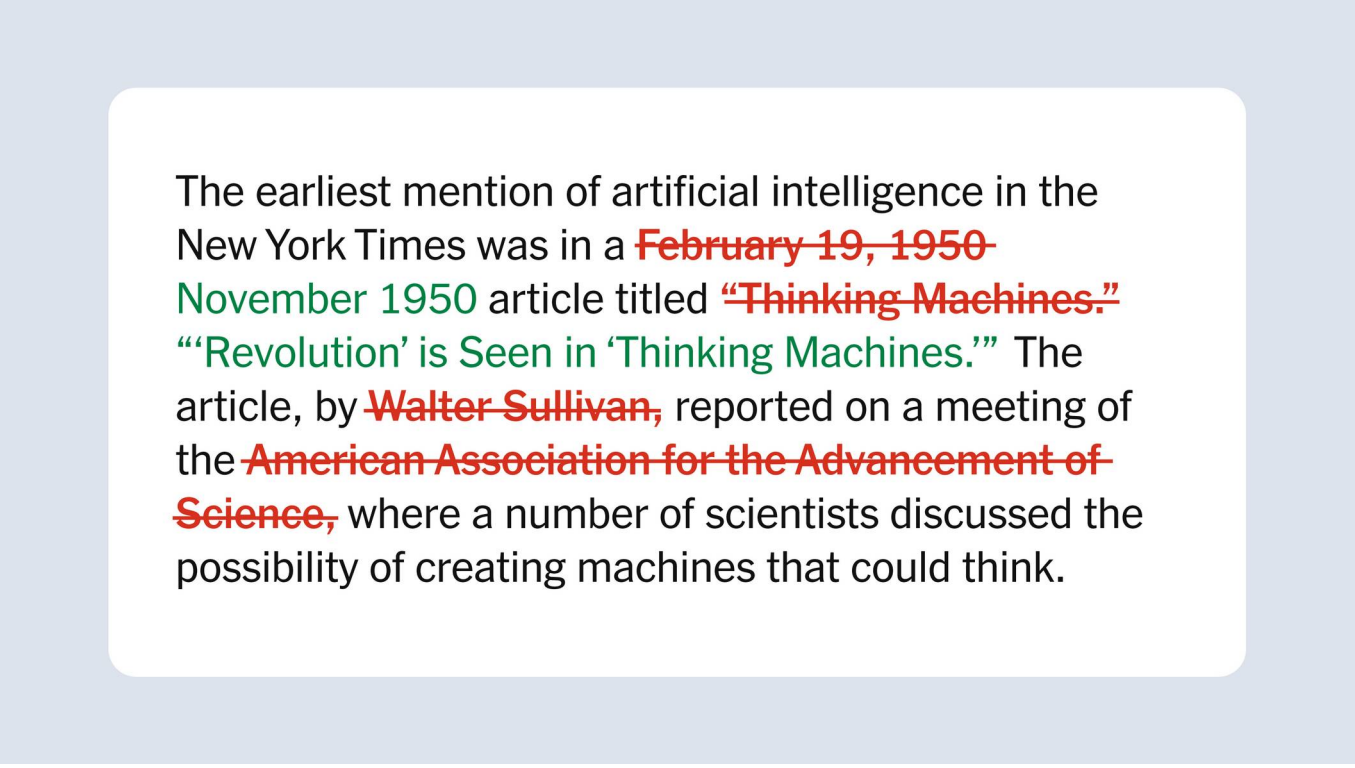
Summary

Method	Description	Diversity	Performance	Quality of Output
Greedy Search	Selects the most probable token at each step.	Low	High	May lack diversity
Beam Search	Considers top-K probable tokens in parallel.	Moderate	Medium-High	Improved diversity
Random Sampling	Randomly selects tokens based on probabilities.	High	Medium	May lack coherence
Random Sampling with Temperature	Adds randomness based on temperature.	High	Medium	More controlled randomness
Top-K Sampling	Selects top-K probable tokens randomly.	High	Medium	Improved diversity
Nucleus Sampling	Selects tokens until cumulative probability ≤ p.	High	Medium-High	Improved coherence

Hallucinations

Hallucinations can be understood as answering confidently when asked for the weather forecast in a fictional city or providing fabricated references in an academic context

But what are the reasons of that?



.from_pretrained(<model>)

In many pre-trained language models, the tokenizer and the model architecture are designed and trained together. The reason for this coupling is that the tokenizer and the model architecture need to be compatible with each other to ensure consistent tokenization and decoding.

If the tokenizer and the model architecture were different or not synchronized, it could lead to tokenization errors, mismatched embeddings, and incorrect predictions.

```
1 tokenizer = AutoTokenizer.from_pretrained("some_model")
2 model = BloomForCausalLM.from_pretrained("some_model")
```

Cost per token

Model	Training	Usage
Ada	$0.0004 / 1K tokens	$0.0016 / 1K tokens
Babbage	$0.0006 / 1K tokens	$0.0024 / 1K tokens
Curie	$0.0030 / 1K tokens	$0.0120 / 1K tokens
Davinci	$0.0300 / 1K tokens	$0.1200 / 1K tokens

Model	Input	Output
GPT-3.5 Turbo (4K context)	$0.0015 / 1K tokens	$0.002 / 1K tokens
GPT-3.5 Turbo (16K context)	$0.003 / 1K tokens	$0.004 / 1K tokens

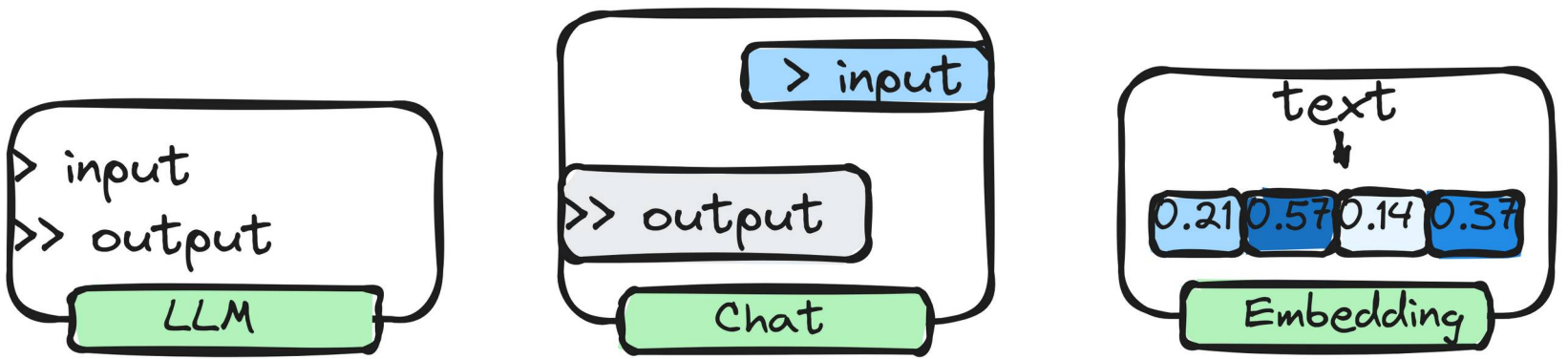
Model	Input	Output
GPT-4 (8K context)	$0.03 / 1K tokens	$0.06 / 1K tokens
GPT-4 (32K context)	$0.06 / 1K tokens	$0.12 / 1K tokens

August 2023

Reminder: Models

LangChain supports a variety of different models, so you can choose the one that suits your needs best:

- **Language Models** are used **for generating text**
    - LLMs utilize APIs that take input text and generate text outputs
    - ChatModels employ models that process **chat messages** and produce responses
- **Text Embedding Models** convert text into **numerical representations**



API keys

You will first need an API key for the LLM provider you want to use. Currently, we must choose between proprietary or open-source foundation models based on a trade-off mainly between performance and cost.

Many open-source models are organized and hosted on Hugging Face as a community hub

```
1 os.environ["OPENAI_API_KEY"] = ... # API_TOKEN
2 os.environ["HUGGINGFACEHUB_API_TOKEN"] = ... # API_TOKEN
```

API keys

Proprietary models are closed-source foundation models owned by companies. They usually are larger than open-source models and thus have better performance, but they may have expensive APIs.
Examples: OpenAI, co:here, AI21 Labs, Anthropic, etc.

Open-source models are usually smaller models with lower capabilities than proprietary models, but they are more cost-effective than proprietary ones.
Examples: BLOOM (BigScience), LLaMA (Meta), Flan-T5 (Google), etc.

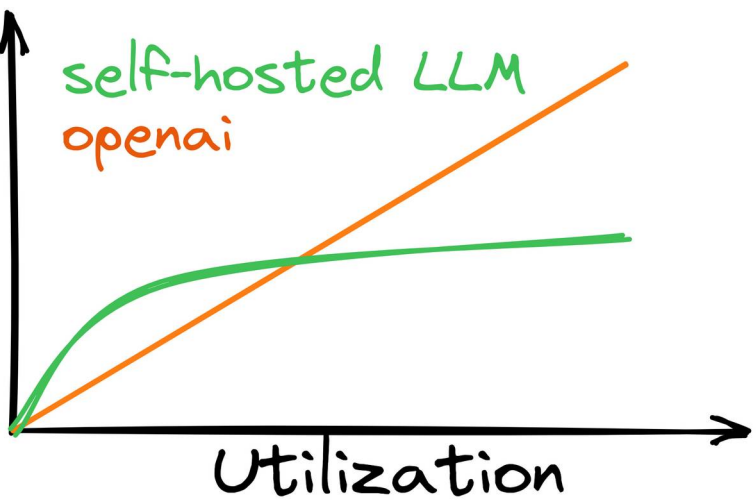
```
# Proprietary LLM from e.g. OpenAI
from langchain.llms import OpenAI
llm = OpenAI(model_name="text-davinci-003")

# Alternatively, open-source LLM hosted on Hugging Face
from langchain import HuggingFaceHub
llm = HuggingFaceHub(repo_id = "google/flan-t5-xl")
```

How much to run a LLAMA2?

	per hour	per month
AWS	$12	~ $8500
GCP	$9	~ $6000
Azure	$7	~ $4500
openai		~$1K - 10K

for LLAMA-2-70B



What to choose?

You want	To choose
private data usage	self-hosted LLM
control over model	self-hosted LLM
quick development	openai
low costs	openai (probably)
minimal infra	openai
top quality	openai (currently)

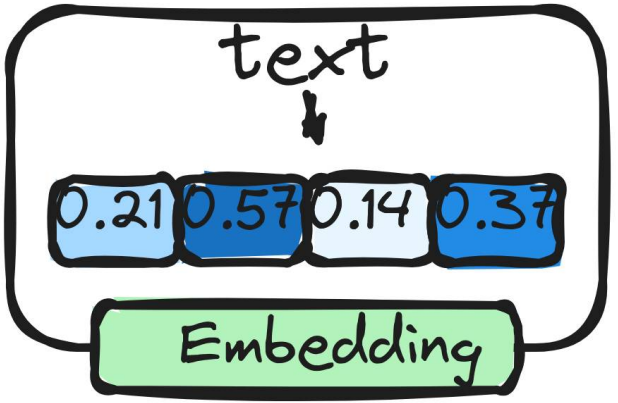
Embeddings

Embeddings are compact numerical representations of words or entities that help computers understand and process language more effectively. These representations encode the meaning and context of words, allowing machines to work with language in a more meaningful and efficient way.

- OpenAIEmbeddings
- HuggingFaceEmbeddings
- GPT4AllEmbeddings
- etc

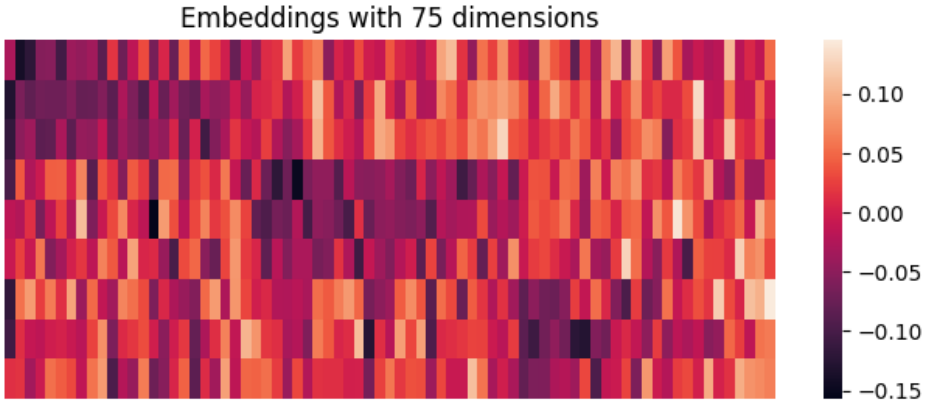
- SpacyEmbeddings

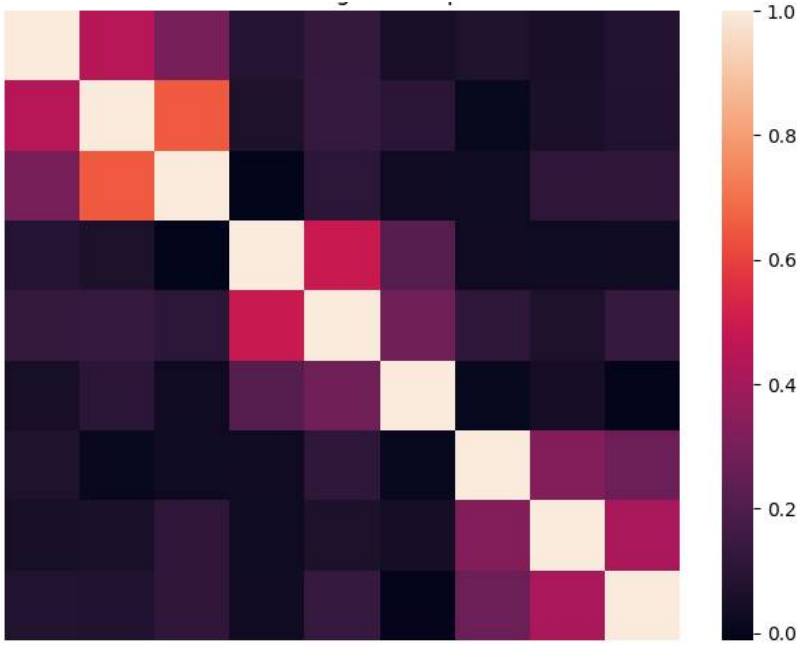
- FakeEmbeddings



Embeddings example

1)Best travel neck pillow for long flights
2)Lightweight backpack for hiking and travel
3)Waterproof duffel bag for outdoor adventures
4)Stainless steel cookware set for induction cooktops
5)High-quality chef's knife set
6)High-performance stand mixer for baking
7)New releases in fiction literature
8)Inspirational biographies and memoirs
9)Top self-help books for personal growth





sentence-transformers/all-mpnet-base-v2

Large Language and Chat Models

A **language model** is a probabilistic model of a natural language that can generate probabilities of a series of words, **based on text corpora** in one or multiple languages **it was trained on**.

A **large language model** is an advanced type of language model that is trained using deep learning techniques on **massive amounts of text data.**

Chat models are usually backed by a language model, but their APIs are more structured. These models take a list of Chat Messages as input, and return a Chat Message.

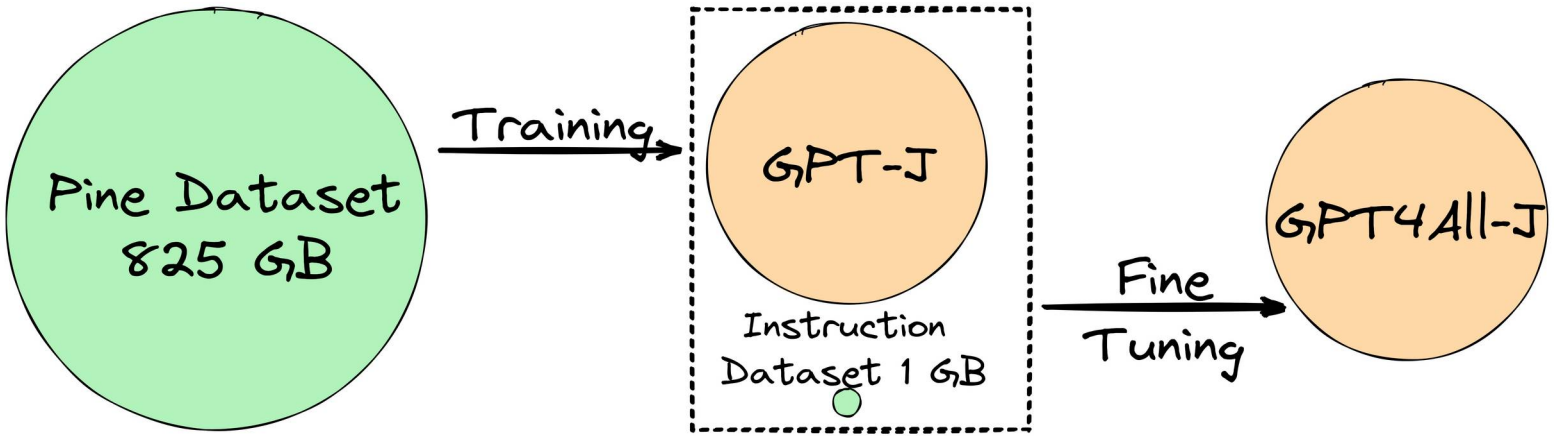
Text
<any form of text>
<article X>
<book Y>
<history Z>
etc

Prompt	Response
<stackoverflow question X>	<response X>
<Q&A platform question Y>	<Q&A platform answer Y>
<Human generated request Z>	<Human generated response Z>
<Chat history Person A>	<Chat history Person B>

Large Language and Chat Models

text (string)
"I've learned the nitrogen vacancies used in Memristors are for "switching", between excited…
"Volunteer Services Volunteer Services As Charleston Area Medical Center volunteers, our…
"Q: Python: My return variable is always None So I found a strange thing that happens in python…

prompt (string)	response (string)
" In Angular if I had a few panels, similar to tabs how do you open the panel on click and animat…	"To accomplish this in Angular, you can use ngAnimate and ng-show/ng-hide directives. Here's…
"Given the following scientific paper: This paper presents a novel approach for temporal and semanti…	"This scientific paper describes three different approaches for segmenting and analyzing edited…
" i am using the following code: <pre> <code>package Presentacion; import…	"The error is caused by a typo in the code. The correct class name is "PrintWriter" and not…



Asking GPT a question

In order to ask model a question there is even no need to connect LangChain.

In this example, we're preparing the prompt, connecting to *openai* package and returning extracted response.

```python
import openai

def get_completion(prompt, model="gpt-3.5-turbo"):
    """
    This function takes a user prompt, sends it to the model, and returns the generated response.

    Parameters:
        prompt (str): The user prompt to be sent to the model.
        model (str): The name of the language model to use. Defaults to "gpt-3.5-turbo".

    Returns:
        str: The generated response from the language model.
    """

    # Prepare the user prompt in a format suitable for the OpenAI Chat API
    messages = [{"role": "user", "content": prompt}]

    # Use the OpenAI Chat API to get a completion (response) from the language model
    response = openai.ChatCompletion.create(
        model=model,
        messages=messages,
        temperature=0.0,
    )

    # Extract the generated response from the API response
    generated_response = response.choices[0].message["content"]

    return generated_response
```

Asking GPT a question

As one can see, simple questions can be answered by the model correctly.

But for complicated requests LangChain may be the way to go.

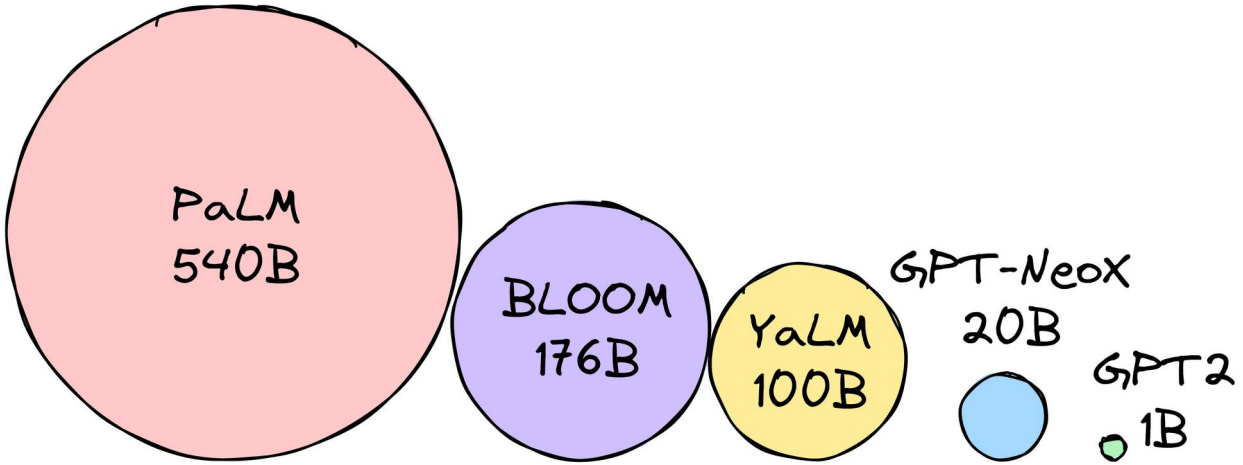
```
1 get_completion("What do they call a Big Mac in France?")
2
3 >> 'In France, a Big Mac is called "Le Big Mac."'
```

```
1 get_completion("What is the average age of retirement of the
  last 5 US presidents?")
2
3 >> The average age of retirement of the last 5 US presidents is
  approximately 77 years old. Here are the retirement ages of the
  last 5 presidents:
4
5 1. Donald Trump - Retired at the age of 74 in 2021.
6 2. Barack Obama - Retired at the age of 55 in 2017.
7 3. George W. Bush - Retired at the age of 62 in 2009.
8 4. Bill Clinton - Retired at the age of 54 in 2001.
9 5. George H. W. Bush - Retired at the age of 68 in 1993.
10
11 Adding up these retirement ages and dividing by 5 gives an
  average retirement age of 77.
```

How Much RAM?

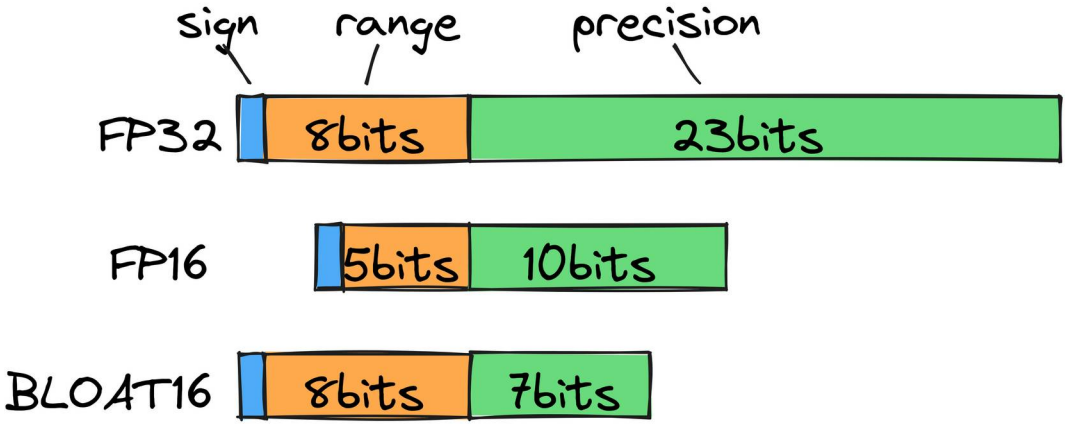
To determine the model size in bytes, you multiply the number of parameters by the chosen precision size.

Let’s say the precision we’ve chosen is bfloat16 (16 bits = 2 bytes).
We want to use BLOOM-176B model.
This means we need 176 billion parameters * 2 bytes = 352GB!



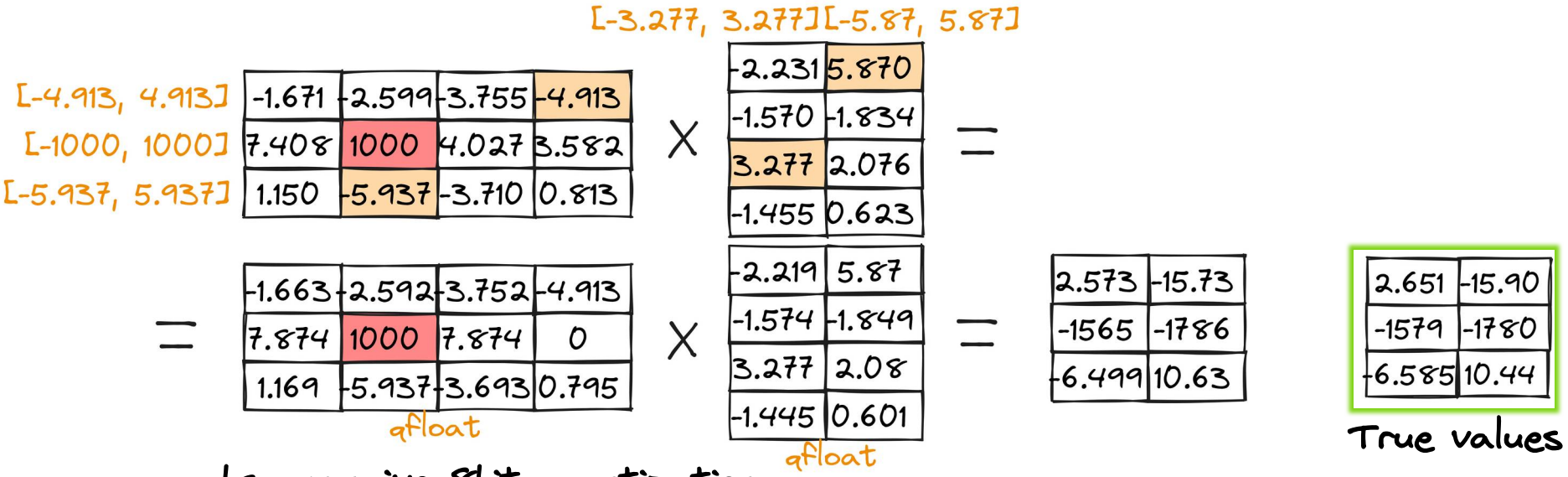
How to Run a Large Model?

A mixed precision approach is used in machine learning, where training and inference are ideally done in FP32 but computation is done in FP16/BF16 for faster training. The weights are held in FP32, and FP16/BF16 gradients are used to update them. During inference, half-precision weights can often provide similar quality as FP32, allowing for faster processing with fewer GPUs.

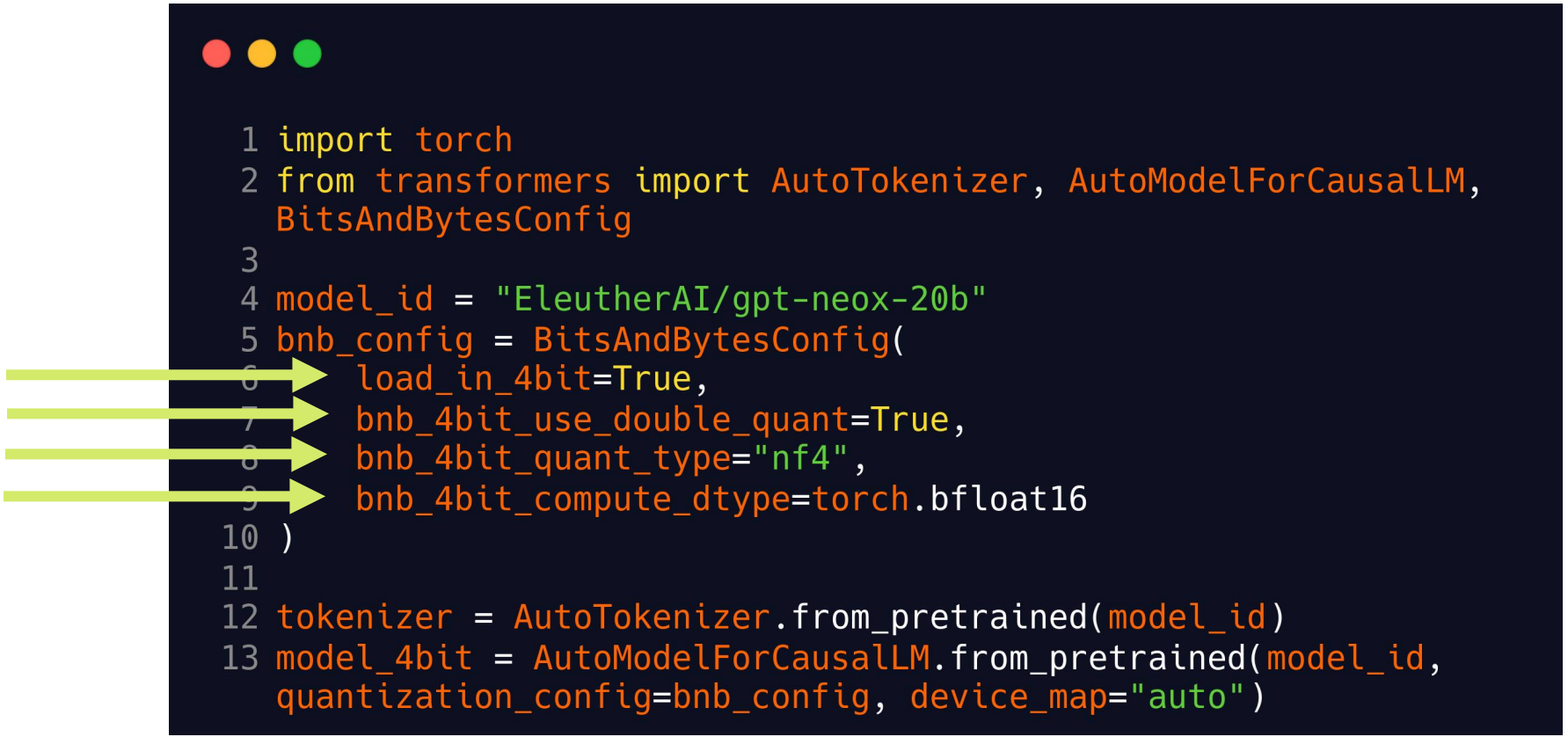


Quantization

Quantization is a compression technique aimed at reducing model memory usage and enhancing inference efficiency.
Post-Training Quantization is about converting the weights of a pre-trained model into lower precision, without retraining.



Quantization example



```python
import torch
from transformers import AutoTokenizer, AutoModelForCausalLM, BitsAndBytesConfig

model_id = "EleutherAI/gpt-neox-20b"
bnb_config = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_use_double_quant=True,
    bnb_4bit_quant_type="nf4",
    bnb_4bit_compute_dtype=torch.bfloat16
)

tokenizer = AutoTokenizer.from_pretrained(model_id)
model_4bit = AutoModelForCausalLM.from_pretrained(model_id, quantization_config=bnb_config, device_map="auto")
```

Running Model CPU-only

```
# Download GGML Model
!wget --output-document="llama-2-7b-chat.bin
<LLAMA_2_URL>.ggmlv3.q8_0.bin"

from langchain.llms import CTransformers

# Local CTransformers wrapper for Llama-2-7B-Chat
llm = CTransformers(model='llama-2-7b-chat.bin',
                    model_type='llama',
                    config={'max_new_tokens': 256,
                            'temperature': 0.1})

llm_chain = LLMChain(prompt=prompt, llm=llm, verbose=True)

pprint.pprint(llm_chain(request_text))
```

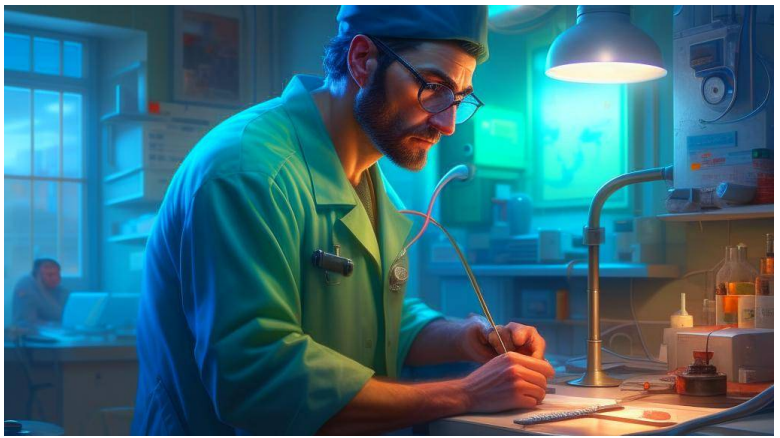
Finetuning Models

Model finetuning, also known as transfer learning, is a machine learning technique used to improve the performance of a pre-existing model on a specific task by training it further on new data related to that task.

Finetuning is commonly employed in scenarios where a pretrained model has learned useful representations of a general domain (in our case natural language) and is then adapted to perform well on a narrower, more specific task.

Finetuning Models

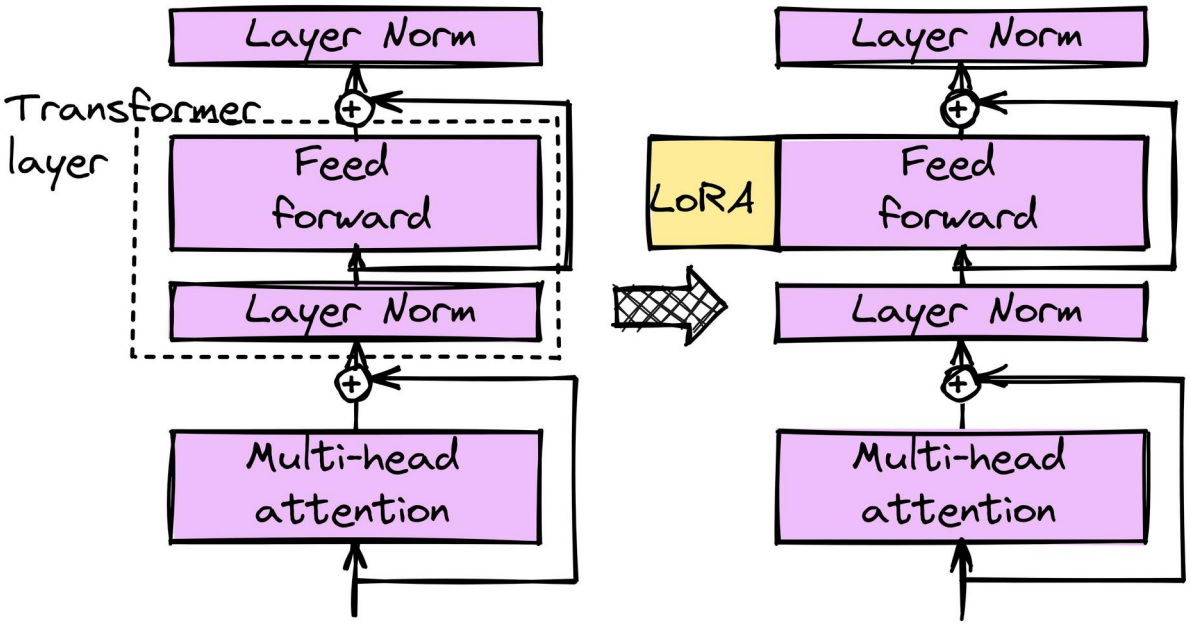
A person, that can **search** all doctors operations

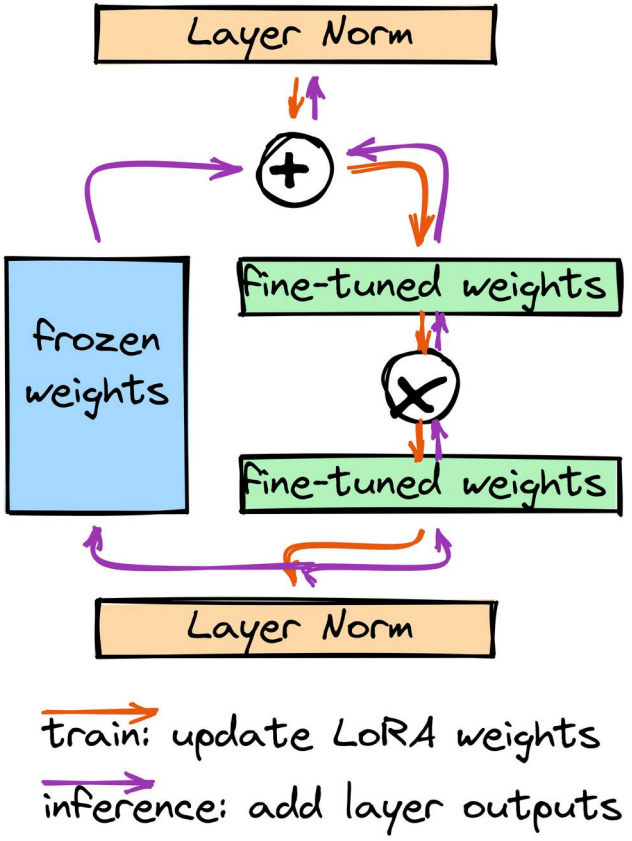


An person **acting** as a doctor (prompt)

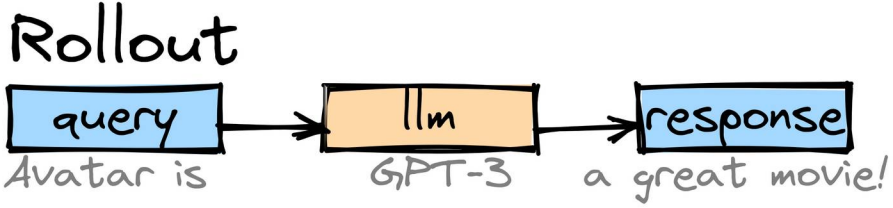
A **trained** doctor, specialized in a specific domain

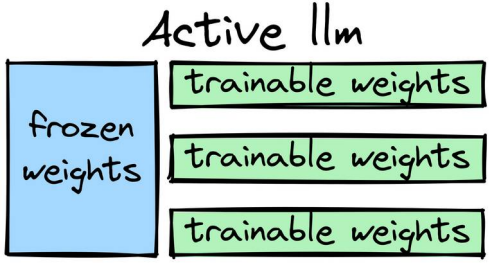
Finetuning Models

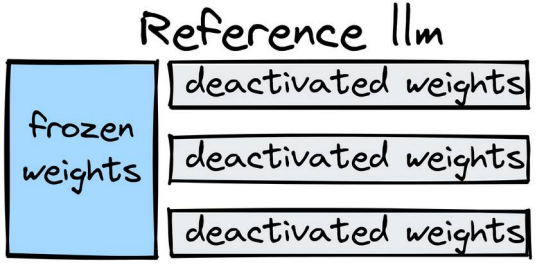


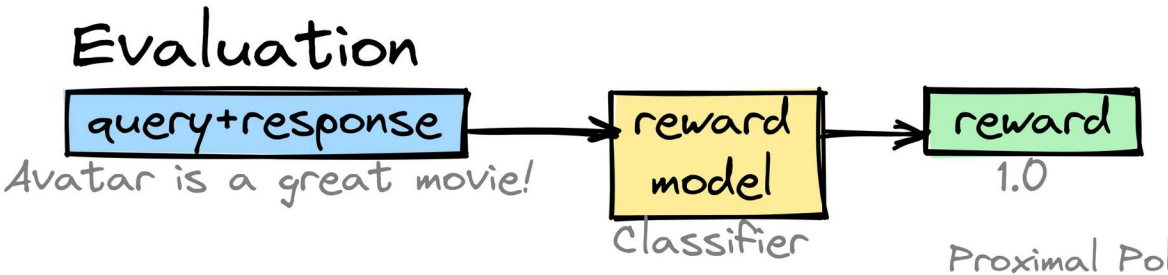


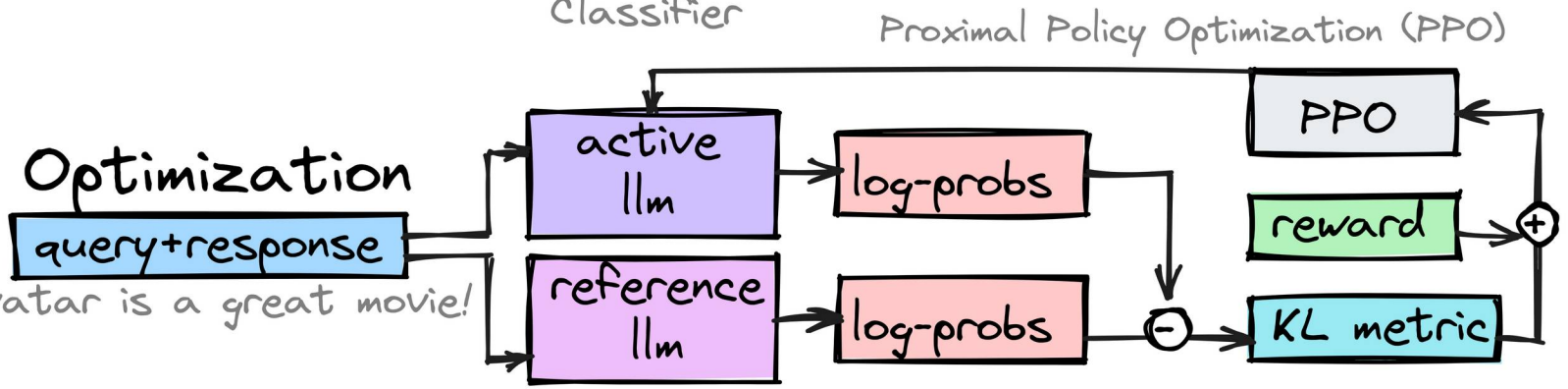
Finetuning Models











Finetuning Models

Step 1: Load your active model in 8-bit precision
Step 2: Add extra trainable adapters using **peft**

```
from peft import LoraConfig, get_peft_model
config = LoraConfig(
    #params
    r=8,
    lora_alpha=32,
)
model = get_peft_model(model, config)
```

Step 3: Use the same model to get the reference and active logits (model training)

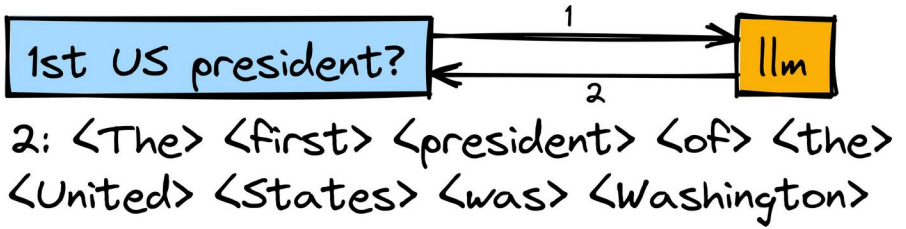
```
trainer = transformers.Trainer(
    model=model,
    train_dataset=data["train"],
    args=transformers.TrainingArguments(
        #training params
        gradient_accumulation_steps=4,
    )
)
trainer.train()
```

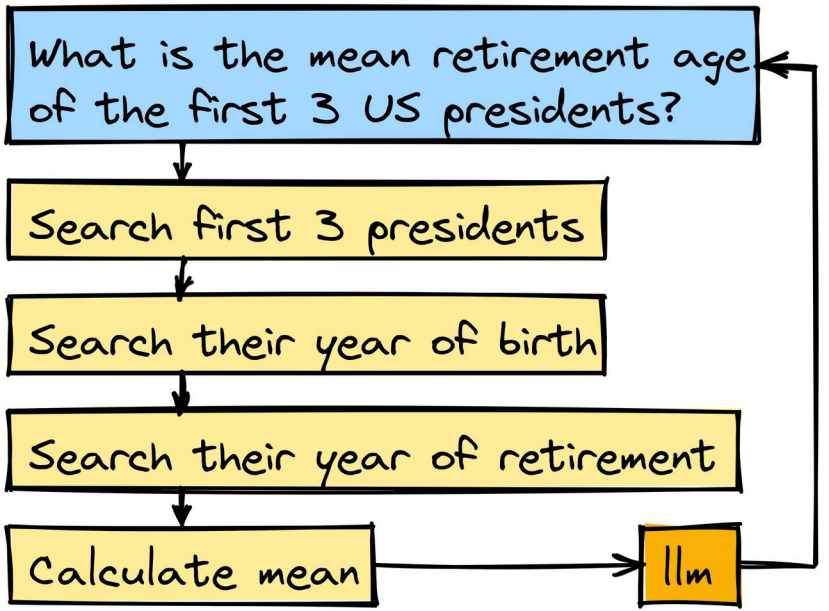
LLM LangChain Application

LLMs are used in LangChain applications in several purposes:

- language model to answer user response

- “llm butter” for LangChain components





Github repo

Notebooks and pdfs