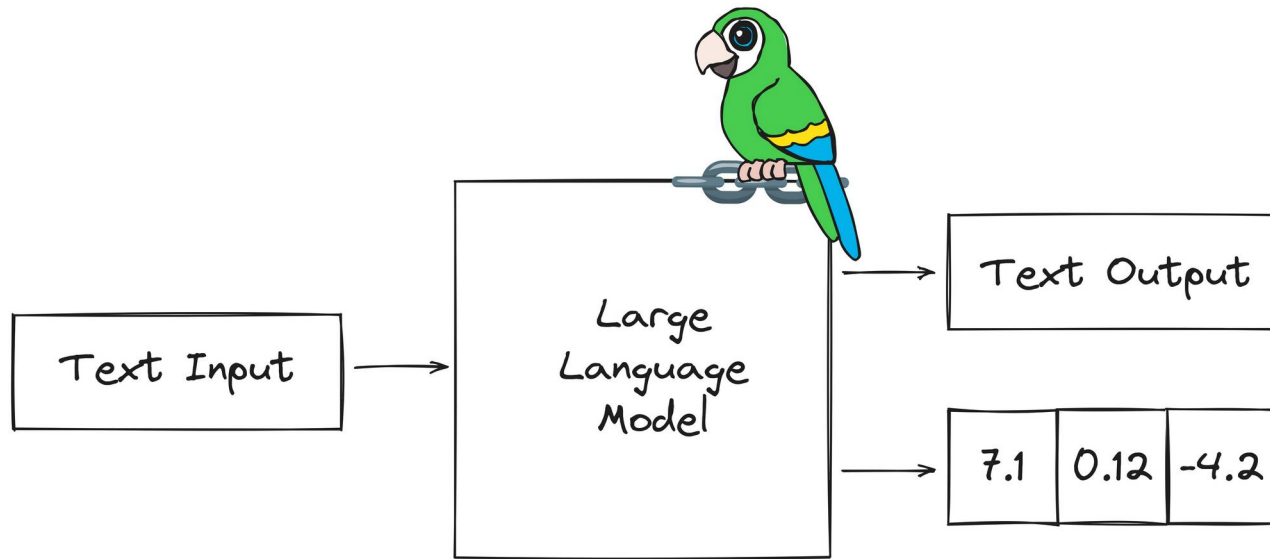


# LangChain: Models



## Lecture 2

Instructor: Dr. Ivan Reznikov, Principal Data Scientist

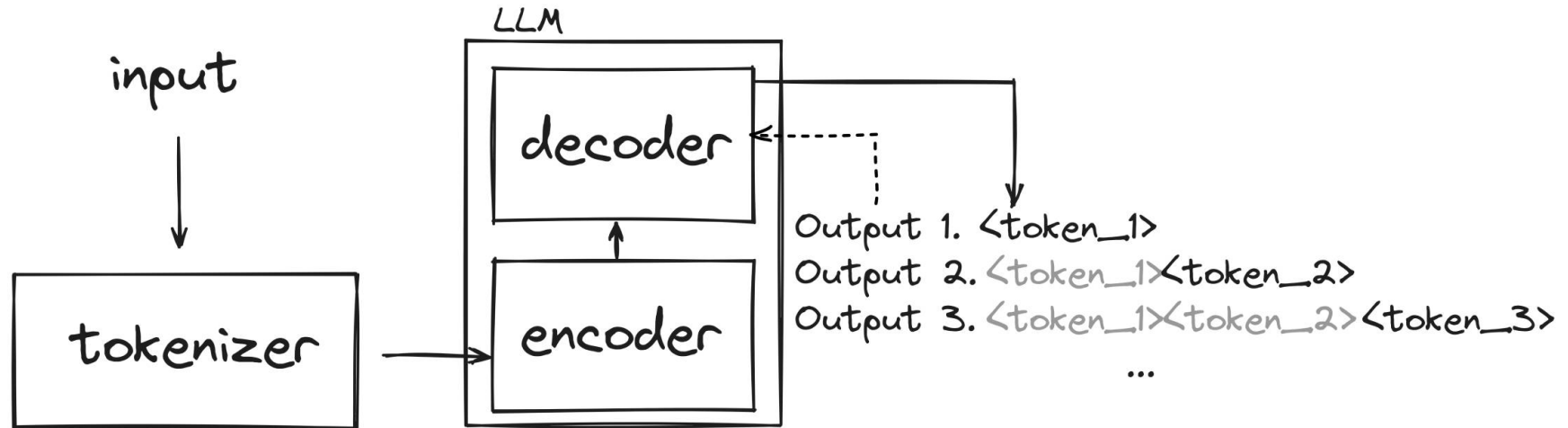
<https://www.linkedin.com/in/reznikovivan/>

# Plan

- Text generation
- Models
- APIs
- Embeddings
- Chat Models and LLMs
- Running locally
- Finetuning

# 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.



# Tokenizers

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

GPT-3 Codex

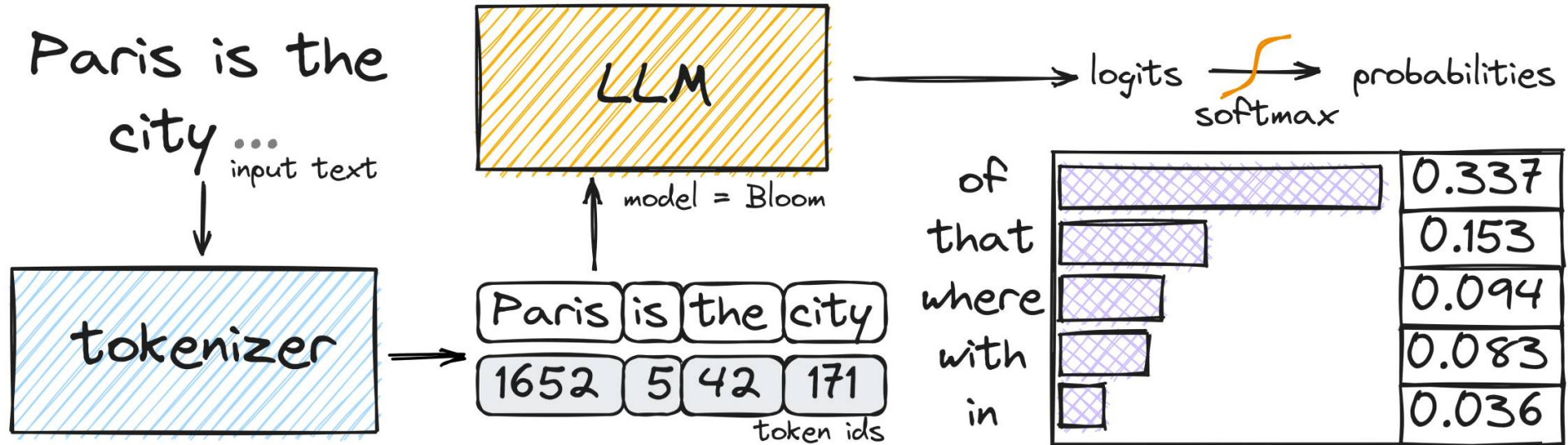
```
Hello, how are you?  
I'm doing great, thanks!  
  
What's your favorite color?  
I love blue and green!
```

Tokens	Characters
29	96

Hello, how are you?  
I'm doing great, thanks!  
  
What's your favorite color?  
I love blue and green!

# Decoding the outputs

Say, we want to generate the continuation of the phrase "Paris is the city ...". The Encoder 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.



# **.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")
```

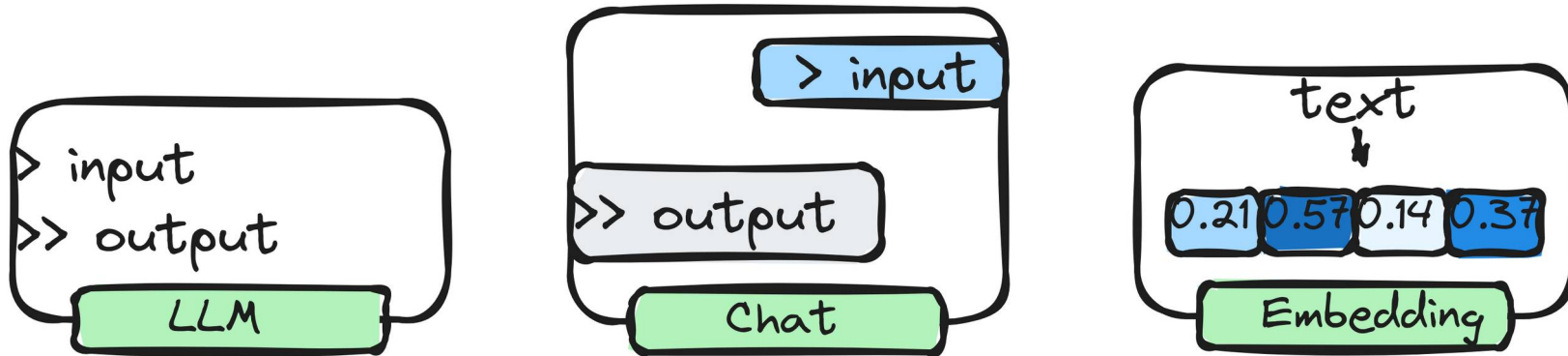
# Cost per token

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

# 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
```

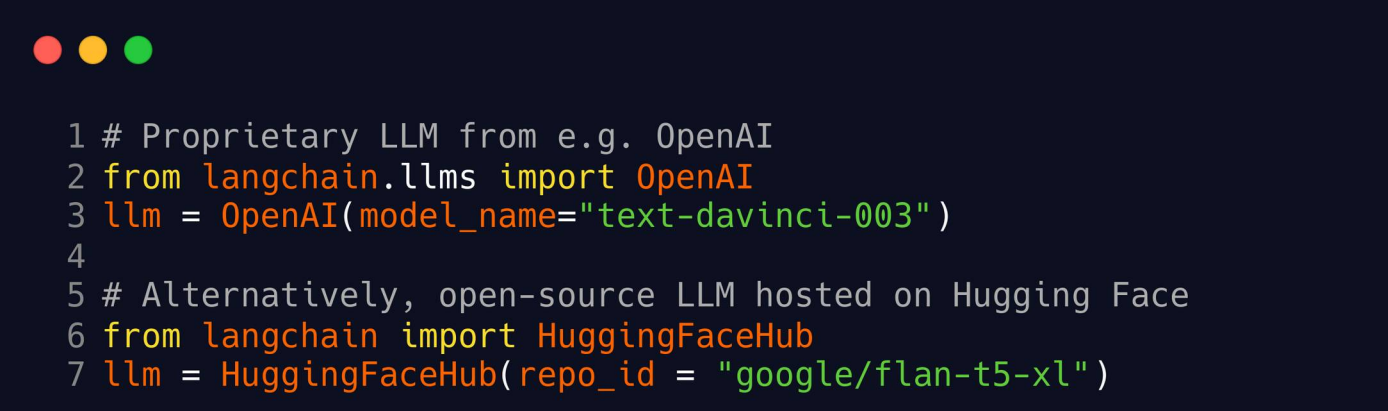
# API keys

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.*

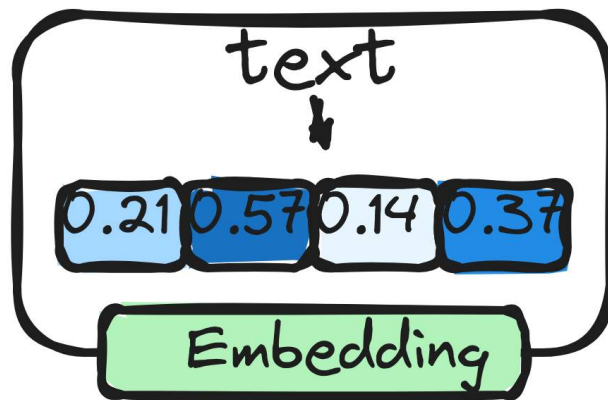


```
1 # Proprietary LLM from e.g. OpenAI
2 from langchain.llms import OpenAI
3 llm = OpenAI(model_name="text-davinci-003")
4
5 # Alternatively, open-source LLM hosted on Hugging Face
6 from langchain import HuggingFaceHub
7 llm = HuggingFaceHub(repo_id = "google/flan-t5-xl")
```

# 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.

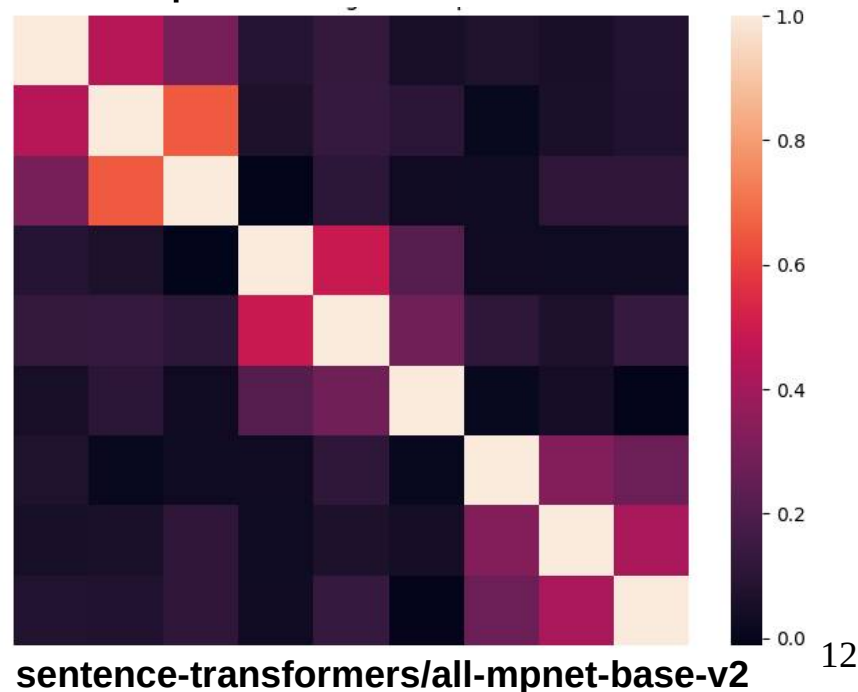
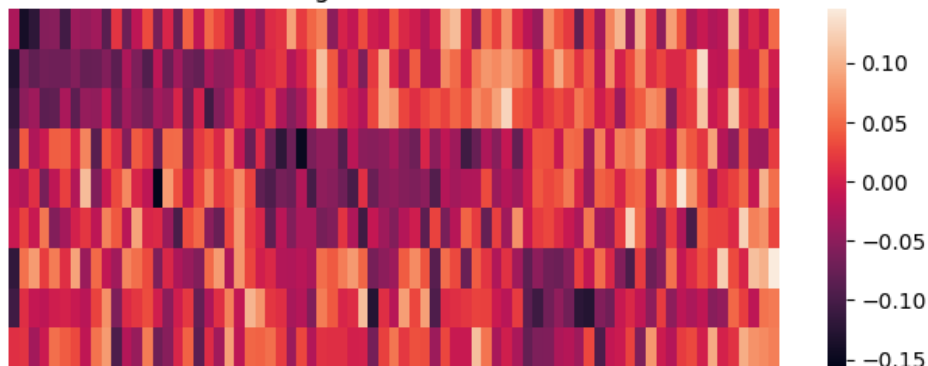
- OpenAIEmbeddings
- HuggingFaceEmbeddings
- GPT4AllEmbeddings
- etc
- SpacyEmbeddings
- FakeEmbeddings



# 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

Embeddings with 75 dimensions



# 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**.

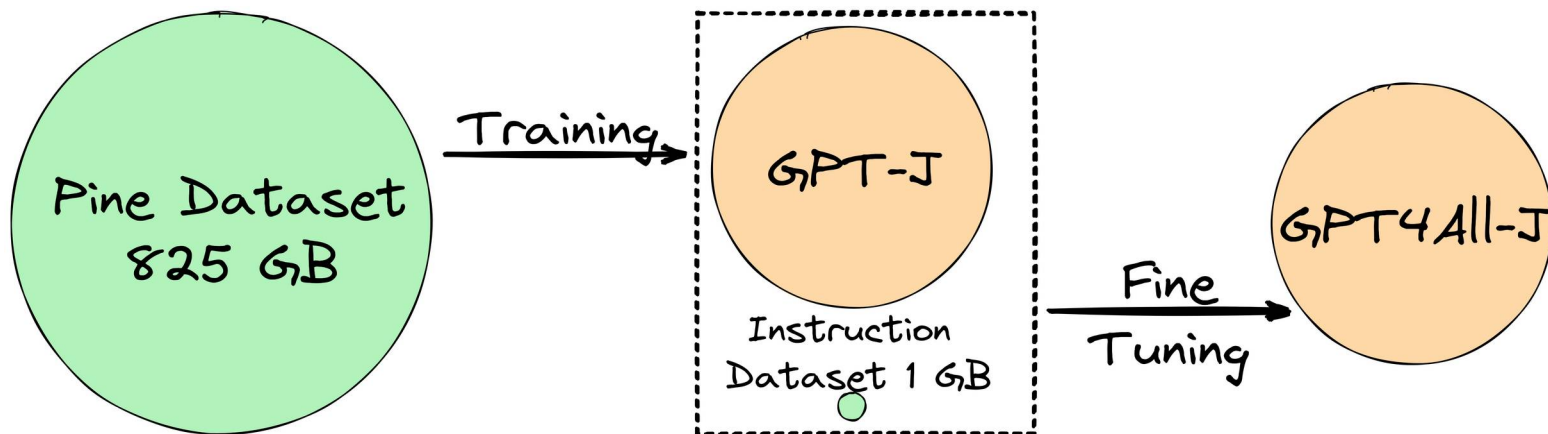
**Chat models** are usually backed by a language model, but their APIs are more structured. These models take a list of Chat Messages as input, and return a Chat Message.

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)	prompt (string)	response (string)
"I've learned the nitrogen vacancies used in Memristors are for "switching", between excited...	"<p>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...
"Volunteer Services Volunteer Services As Charleston Area Medical Center volunteers, our...	"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...
"Q: Python: My return variable is always None So I found a strange thing that happens in python...	"<p>i am using the following code:</p> <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.

```
1 import openai
2
3 def get_completion(prompt, model="gpt-3.5-turbo"):
4     """
5     This function takes a user prompt, sends it to the model, and returns the generated response.
6
7     Parameters:
8         prompt (str): The user prompt to be sent to the model.
9         model (str): The name of the language model to use. Defaults to "gpt-3.5-turbo".
10
11     Returns:
12         str: The generated response from the language model.
13     """
14
15     # Prepare the user prompt in a format suitable for the OpenAI Chat API
16     messages = [{"role": "user", "content": prompt}]
17
18     # Use the OpenAI Chat API to get a completion (response) from the language model
19     response = openai.ChatCompletion.create(
20         model=model,
21         messages=messages,
22         temperature=0.0,
23     )
24
25     # Extract the generated response from the API response
26     generated_response = response.choices[0].message["content"]
27
28     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
2 last 5 US presidents?")
3 >> The average age of retirement of the last 5 US presidents is
4 approximately 77 years old. Here are the retirement ages of the
5 last 5 presidents:
6 1. Donald Trump - Retired at the age of 74 in 2021.
7 2. Barack Obama - Retired at the age of 55 in 2017.
8 3. George W. Bush - Retired at the age of 62 in 2009.
9 4. Bill Clinton - Retired at the age of 54 in 2001.
10 5. George H. W. Bush - Retired at the age of 68 in 1993.
11 Adding up these retirement ages and dividing by 5 gives an
    average retirement age of 77.
```

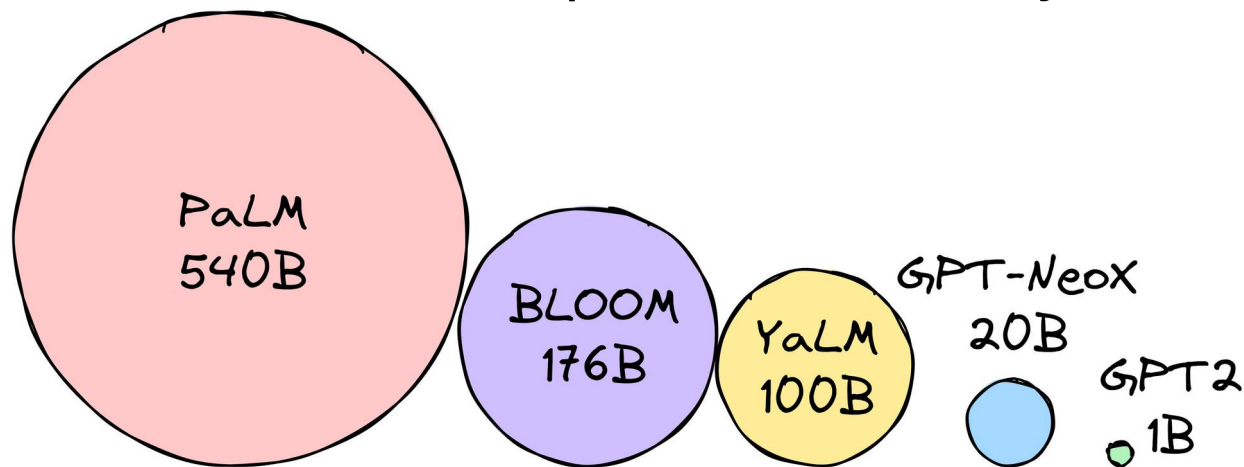


# How Much RAM?

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

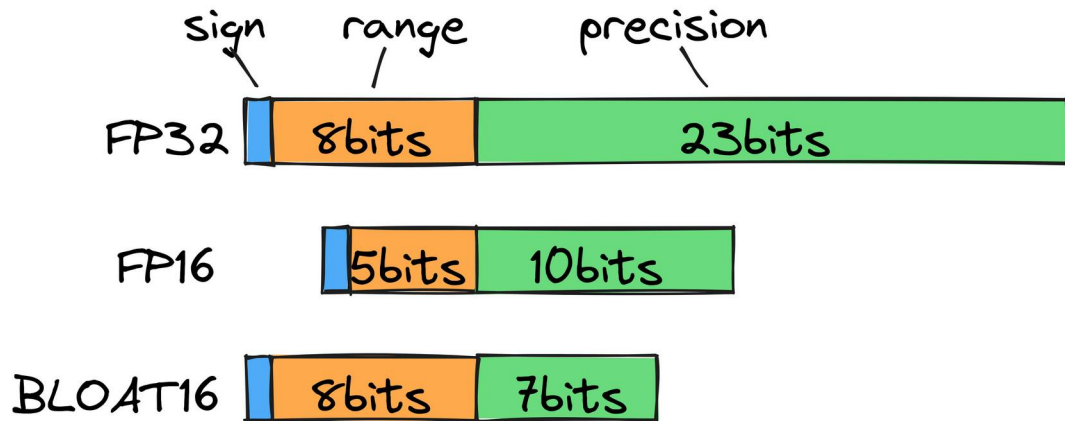
Let's say the precision we've chosen is bfloat16 (16 bits = 2 bytes).  
We want to use BLOOM-176B model.

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.

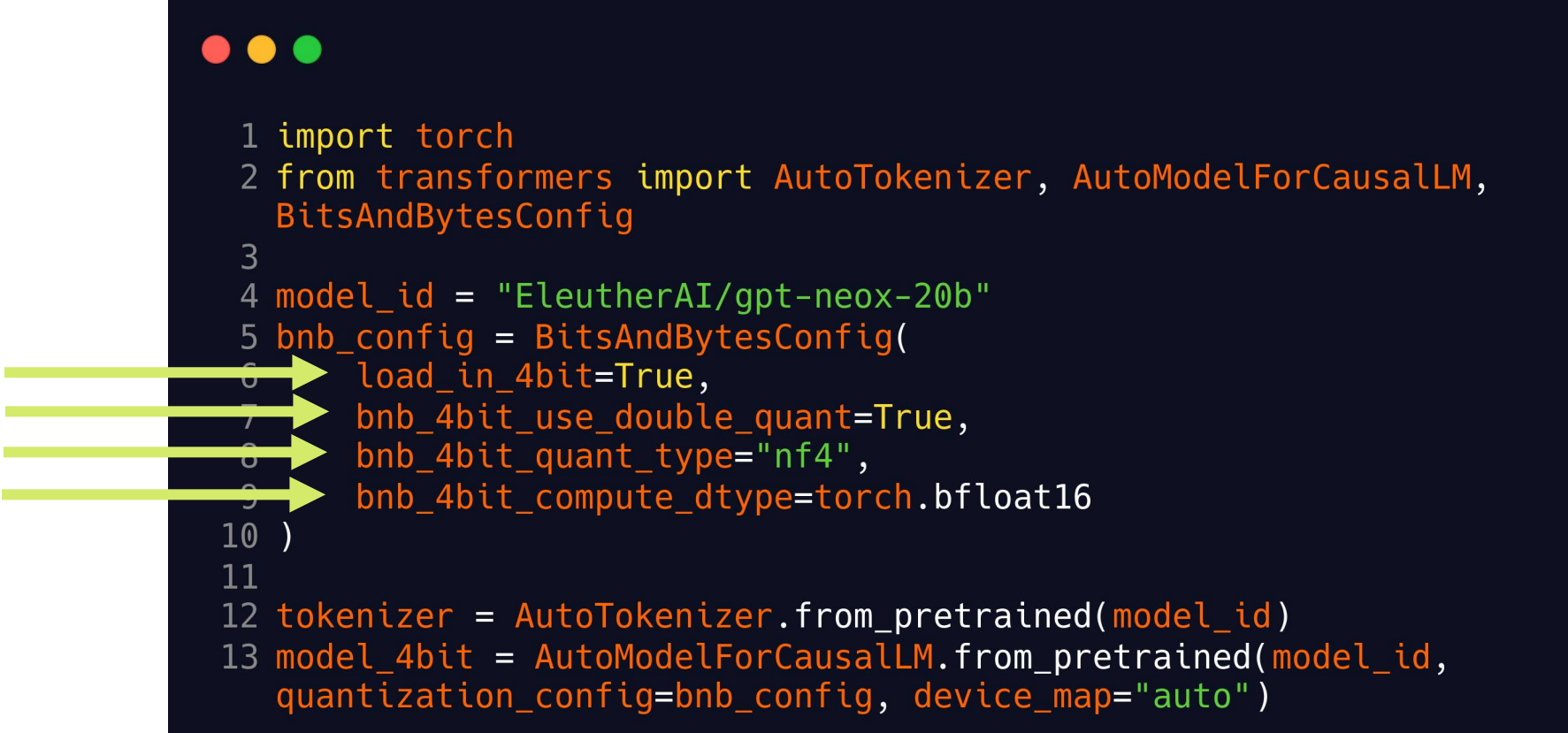
$$\begin{array}{c}
 \begin{array}{c} [-4.913, 4.913] \\ [-1000, 1000] \\ [-5.937, 5.937] \end{array}
 \begin{array}{|c|c|c|c|} \hline -1.671 & 2.599 & 3.755 & -4.913 \\ \hline 7.408 & 1000 & 4.027 & 3.582 \\ \hline 1.150 & -5.937 & -3.710 & 0.813 \\ \hline \end{array}
 \times
 \begin{array}{c} [-3.277, 3.277] \quad [-5.87, 5.87] \\ \begin{array}{|c|c|} \hline -2.231 & 5.870 \\ \hline -1.570 & -1.834 \\ \hline 3.277 & 2.076 \\ \hline -1.455 & 0.623 \\ \hline \end{array} \end{array}
 =
 \end{array}$$
  

$$\begin{array}{c}
 =
 \begin{array}{|c|c|c|c|} \hline -1.663 & 2.592 & 3.752 & -4.913 \\ \hline 7.874 & 1000 & 7.874 & 0 \\ \hline 1.169 & -5.937 & -3.693 & 0.795 \\ \hline \end{array}
 \times
 \begin{array}{c} \text{qfloat} \\ \begin{array}{|c|c|} \hline -2.219 & 5.87 \\ \hline -1.574 & -1.849 \\ \hline 3.277 & 2.08 \\ \hline -1.445 & 0.601 \\ \hline \end{array} \end{array}
 =
 \begin{array}{|c|c|} \hline 2.573 & -15.73 \\ \hline -1565 & -1786 \\ \hline -6.499 & 10.63 \\ \hline \end{array}
 \end{array}$$
  

$$\begin{array}{c}
 \text{absmax naive 8bit quantization} \\
 \text{qfloat}
 \end{array}
 \times
 \begin{array}{c} \text{qfloat} \end{array}
 =
 \begin{array}{|c|c|} \hline 2.651 & -15.90 \\ \hline -1579 & -1780 \\ \hline -6.585 & 10.44 \\ \hline \end{array}$$

True values

# Quantization example

A dark-themed terminal window with three colored window control buttons (red, yellow, green) in the top-left corner. It contains Python code for quantizing a GPT model. Four yellow arrows point from the left to lines 6, 7, 8, and 9 of the code.

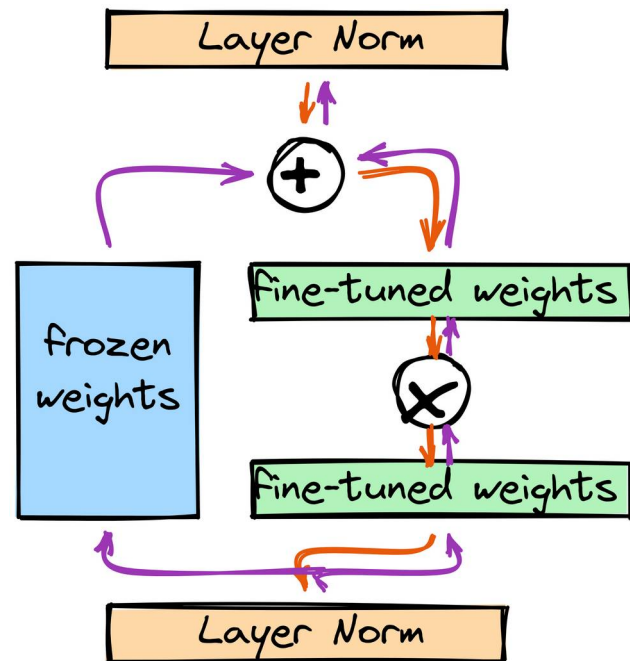
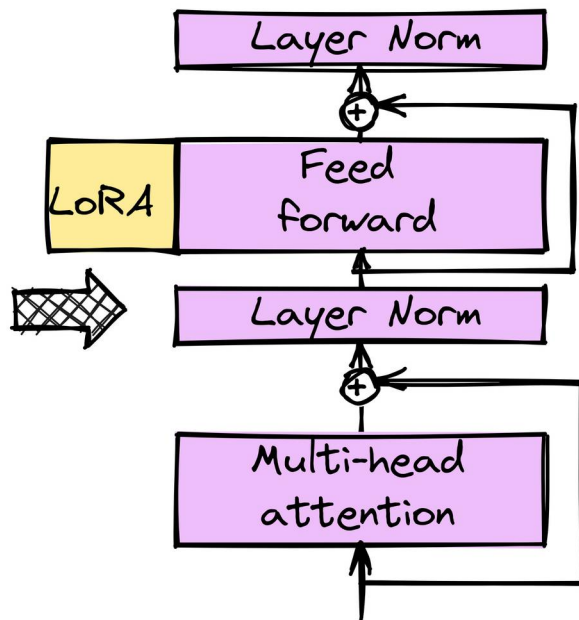
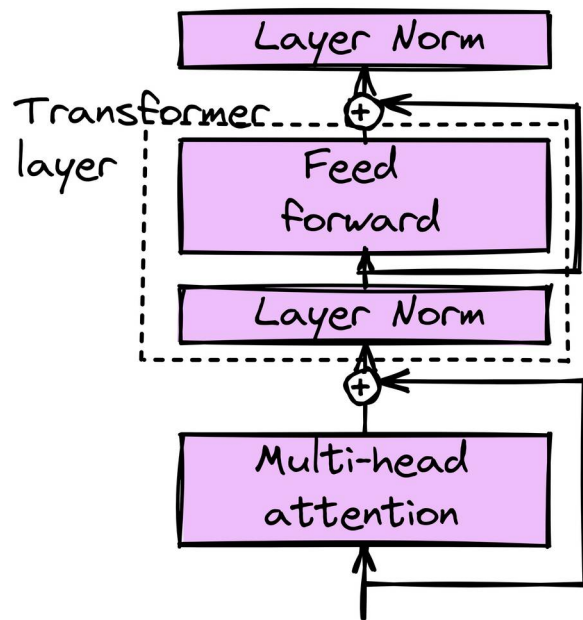
```
1 import torch
2 from transformers import AutoTokenizer, AutoModelForCausalLM,
  BitsAndBytesConfig
3
4 model_id = "EleutherAI/gpt-neox-20b"
5 bnb_config = BitsAndBytesConfig(
6     load_in_4bit=True,
7     bnb_4bit_use_double_quant=True,
8     bnb_4bit_quant_type="nf4",
9     bnb_4bit_compute_dtype=torch.bfloat16
10 )
11
12 tokenizer = AutoTokenizer.from_pretrained(model_id)
13 model_4bit = AutoModelForCausalLM.from_pretrained(model_id,
  quantization_config=bnb_config, device_map="auto")
```

# Running Model CPU-only



```
1 # Download GGML Model
2 !wget --output-document="llama-2-7b-chat.bin
   <LLAMA_2_URL>.ggmlv3.q8_0.bin"
3
4 from langchain.llms import CTransformers
5
6 # Local CTransformers wrapper for Llama-2-7B-Chat
7 llm = CTransformers(model='llama-2-7b-chat.bin',
8                     model_type='llama',
9                     config={'max_new_tokens': 256,
10                           'temperature': 0.1})
11
12 llm_chain = LLMChain(prompt=prompt, llm=llm, verbose=True)
13
14 pprint.pprint(llm_chain(request_text))
```

# Finetuning Models



train: update LoRA weights  
inference: add layer outputs

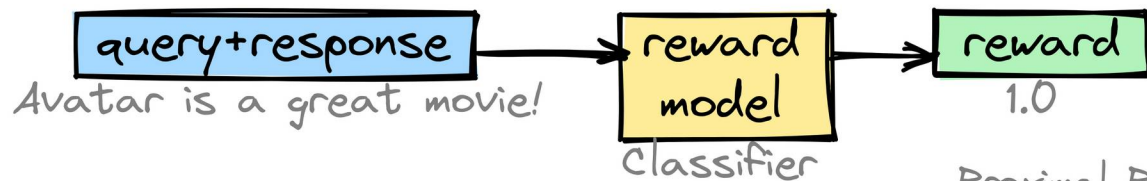


# Finetuning Models

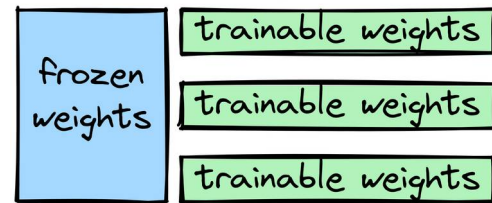
## Rollout



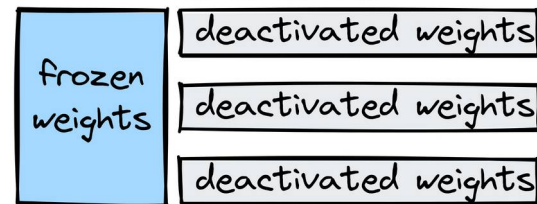
## Evaluation



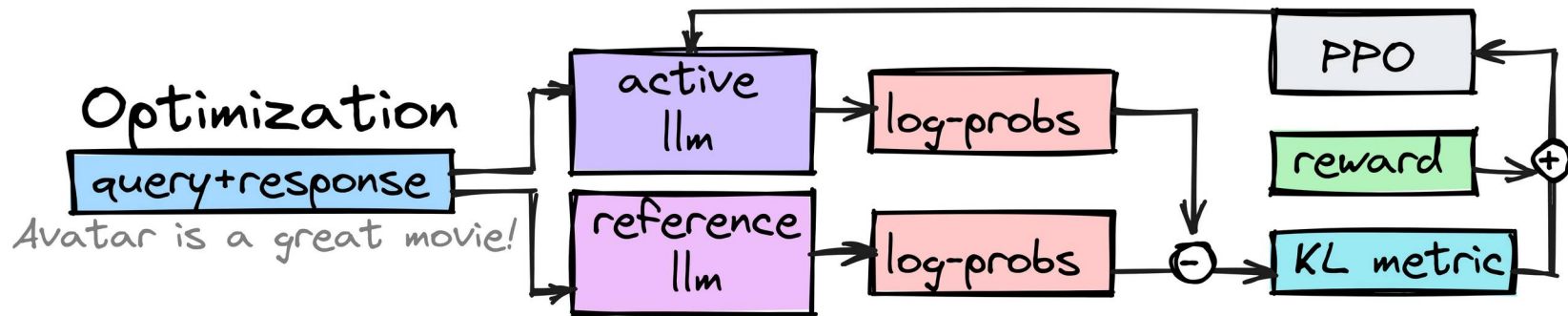
## Active llm



## Reference llm



## Proximal Policy Optimization (PPO)



# Finetuning Models

Step 1: Load your active model in 8-bit precision

Step 2: Add extra trainable adapters using **peft**

```
1 from peft import LoraConfig, get_peft_model
2 config = LoraConfig(
3     #params
4     r=8,
5     lora_alpha=32,
6 )
7 model = get_peft_model(model, config)
```

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

```
1 trainer = transformers.Trainer(
2     model=model,
3     train_dataset=data["train"],
4     args=transformers.TrainingArguments(
5         #training params
6         gradient_accumulation_steps=4,
7     )
8 )
9 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

