LLM Chatbot

Instructors **Battista Biggio**, **Angelo Sotgiu** and **Leonardo Regano**

M.Sc. in Computer Engineering, Cybersecurity and Artificial Intelligence
University of Cagliari, Italy

Covered topics:

- Basic concepts about LLMs for text generation
- Loading models and running inference
- Writing a simple chat web application

Large Language Models

Machine Learning techniques aim to learn patterns from data, to achive tasks without being explicitly programmed.

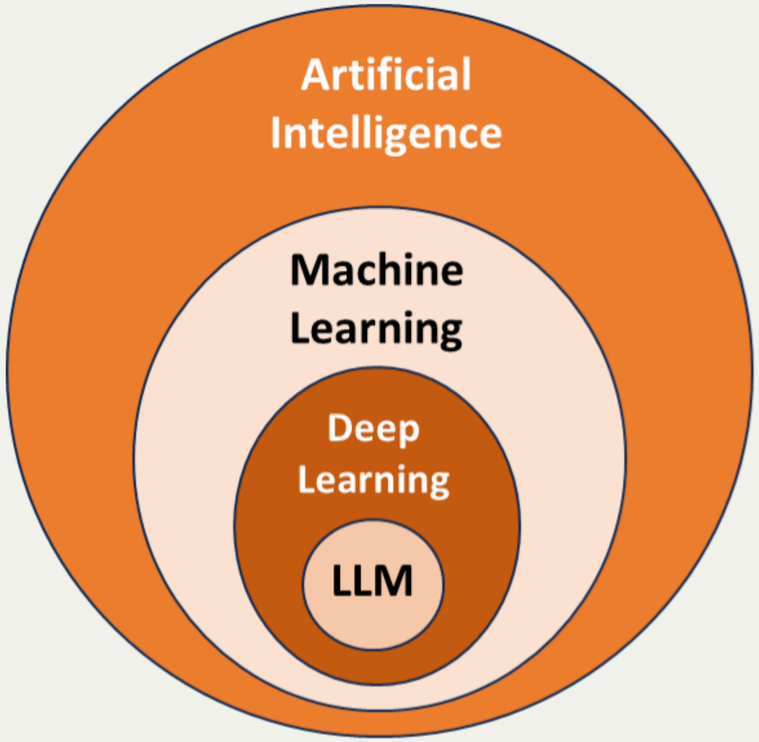


Image source

Machine Learning techniques aim to learn patterns from data, to achive tasks without being explicitly programmed.

Deep Learning algorithms rely on neural networks composed of multiple layers of neurons, inspired by the human brain structure.

- Starting from the input, each layer builds a new representation that adds more abstraction and complexity
- Those representations are automatically learnt during the training process
- In this way, the NNs are able to capture complex relationships within the input data

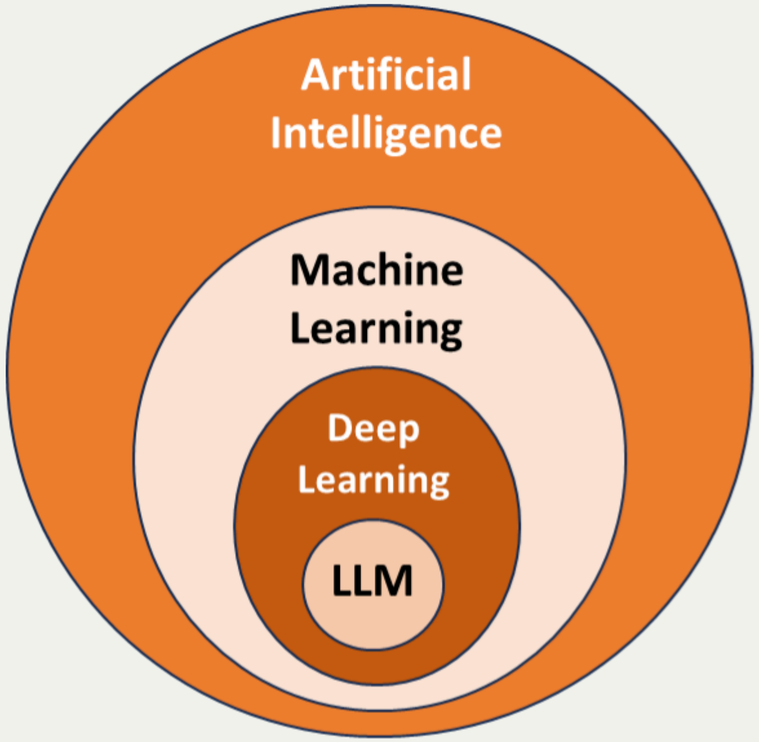


Image source

LLMs are very large neural networks, trained on big text datasets.

- GPT1: 117 million parameters
- GPT2: 1.5 billion parameters
- GPT3: 175 billion parameters
- GPT4: (rumors) ~1,78 trillions parameters

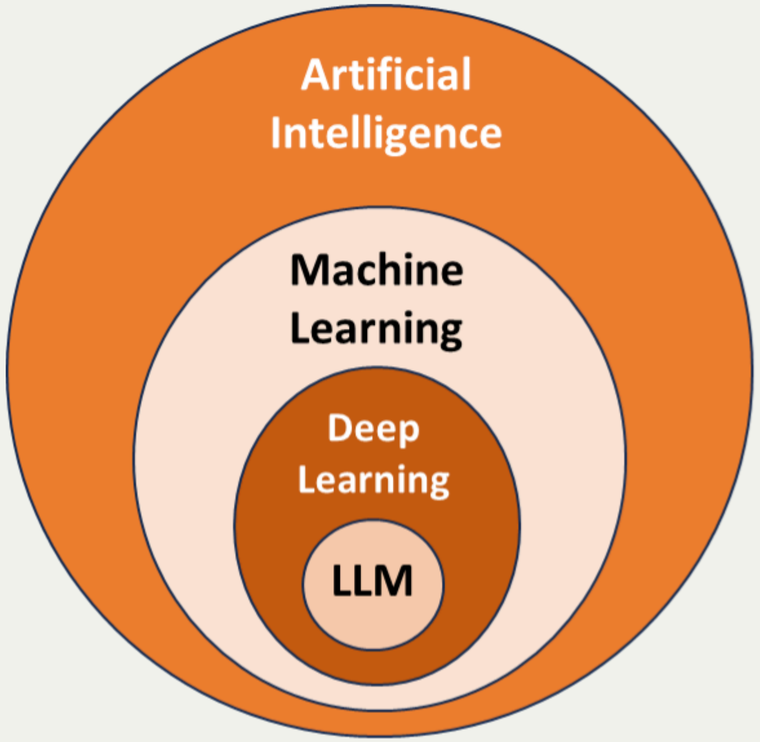


Image source

LLMs are very large neural networks, trained on big text datasets.

- GPT1: 117 million parameters
- GPT2: 1.5 billion parameters
- GPT3: 175 billion parameters
- GPT4: (rumors) ~1,78 trillions parameters

The more parameters a model has, the more information it can memorize and recall. Larger models are usually better at achieving high accuracy compared to their smaller counterparts.

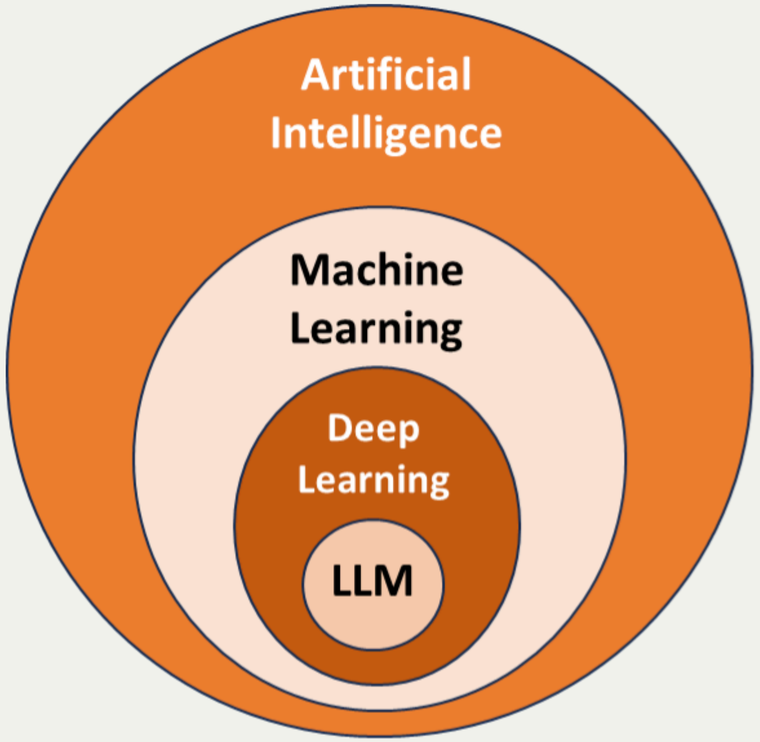
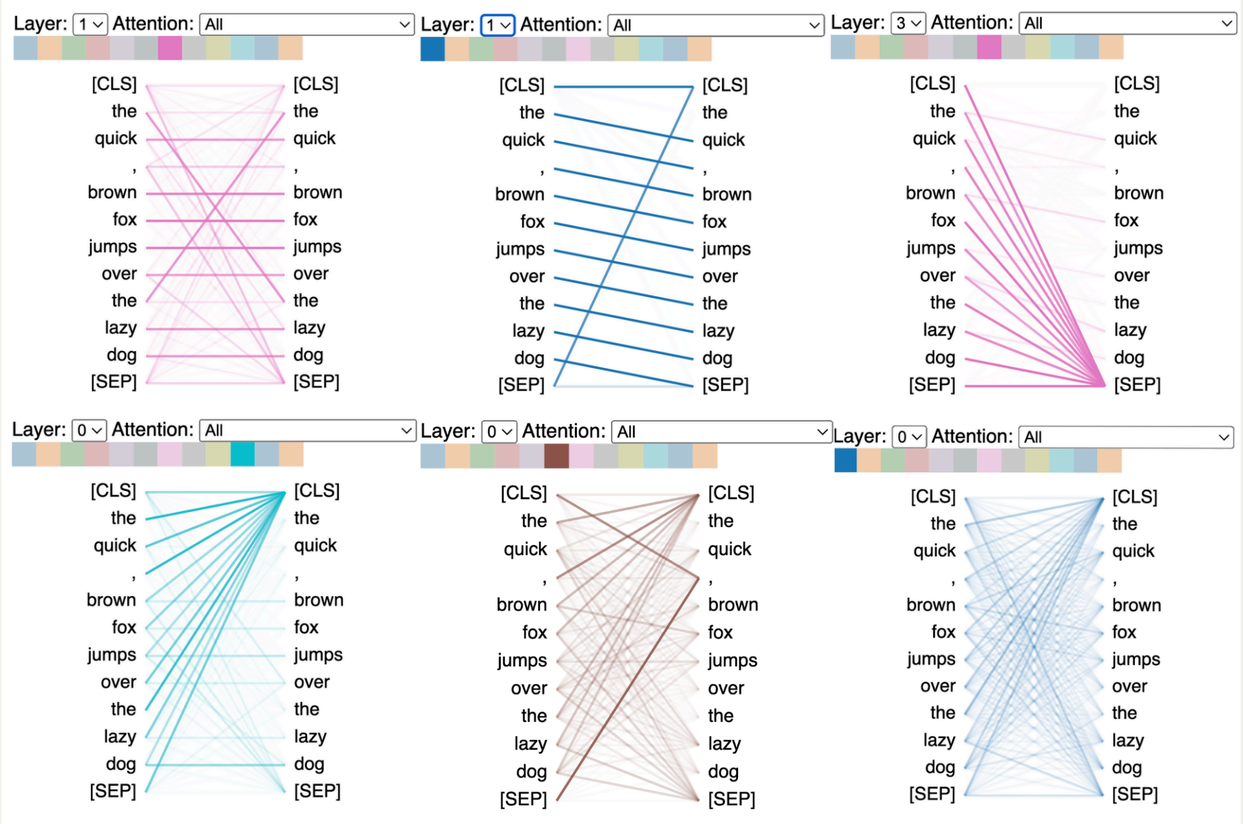


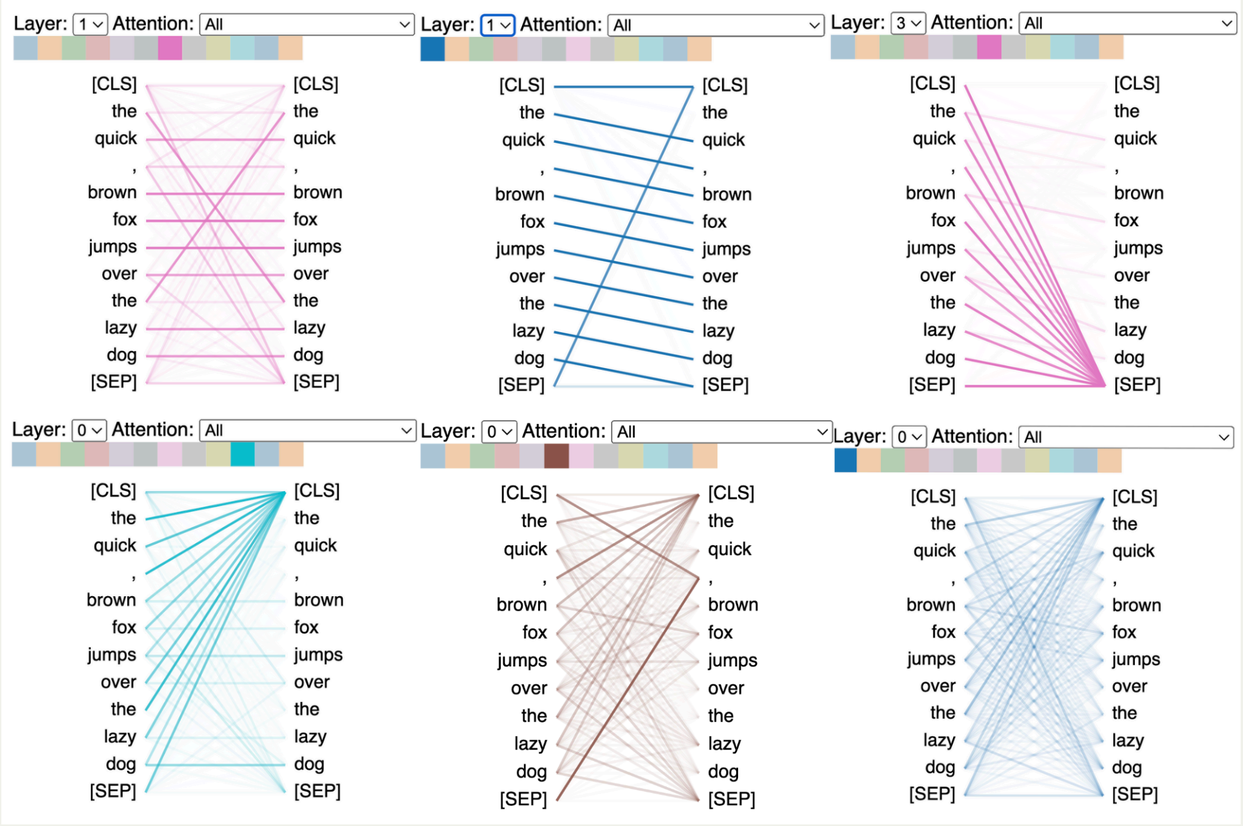
Image source

The core technology of LLMs lies in their particular neural network architecture: the **transformer**, which is based on the **attention** mechanism. Attention assign *weights* to each text component, representing its importance with respect to the other components. This allows capturing relationships within a text sequence.



Source: BertViz

The core technology of LLMs lies in their particular neural network architecture: the **transformer**, which is based on the **attention** mechanism. Attention assign *weights* to each text component, representing its importance with respect to the other components. This allows capturing relationships within a text sequence.

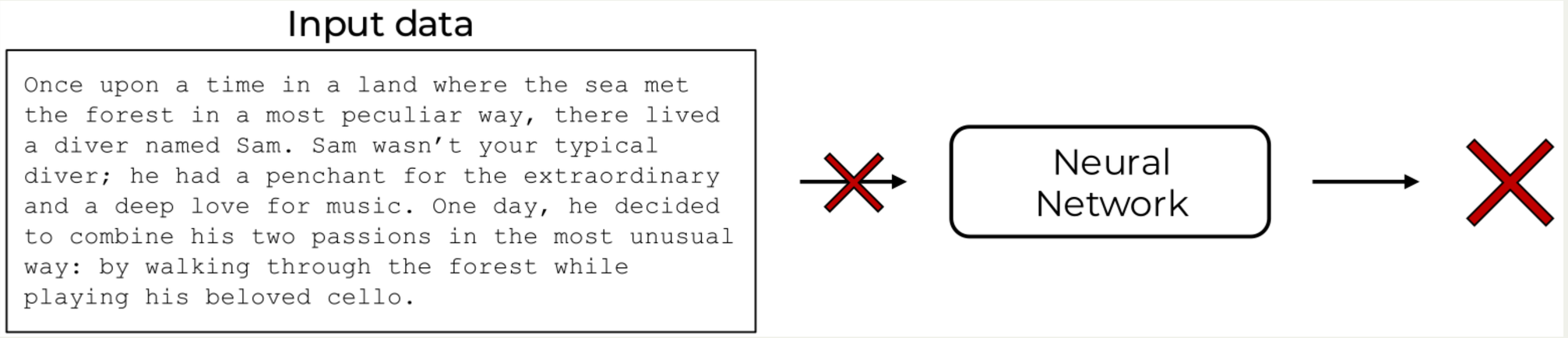


Source: BertViz

LLMs have multiple attention layers (plus additional layers). Each attention layer typically has multiple *heads* to capture different concepts.

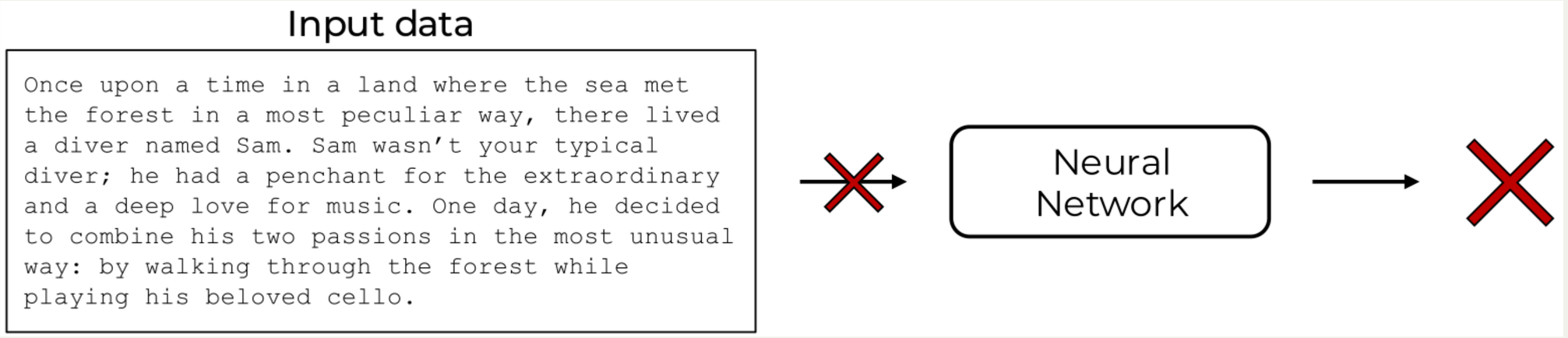
Text Representation

ML models rely on numerical inputs for both training and inference...



Text Representation

ML models rely on numerical inputs for both training and inference...



Text representation is the process of converting words into numerical vectors. This process is usually part of the learning phase itself: the model tries to learn representations that are convenient and meaningful for its task.

Tokenization

The first step is to divide the text into subitems, called tokens.

Tokenization

The first step is to divide the text into subitems, called tokens.

An ID is then assigned to each token, in order to build a vocabulary.

Tokenization

The first step is to divide the text into subitems, called tokens.

An ID is then assigned to each token, in order to build a vocabulary.

Tokenization can be performed:

- by word $\rightarrow$ huge vocabulary

Tokenization

The first step is to divide the text into subitems, called tokens.

An ID is then assigned to each token, in order to build a vocabulary.

Tokenization can be performed:

- by word $\rightarrow$ huge vocabulary
- by character $\rightarrow$ huge input sequence (models have a fixed input size[1]); loss of meaning

(1) Remember this, we'll address it later

Tokenization

The first step is to divide the text into subitems, called tokens.

An ID is then assigned to each token, in order to build a vocabulary.

Tokenization can be performed:

- by word $\rightarrow$ huge vocabulary
- by character $\rightarrow$ huge input sequence (models have a fixed input size[1]); loss of meaning
- by **subword** $\rightarrow$ good tradeoff!

---

(1) Remember this, we'll address it later

Tokenization

The first step is to divide the text into subitems, called tokens.

An ID is then assigned to each token, in order to build a vocabulary.

Tokenization can be performed:

- by word $\rightarrow$ huge vocabulary
- by character $\rightarrow$ huge input sequence (models have a fixed input size[1]); loss of meaning
- by **subword** $\rightarrow$ good tradeoff!

The subword tokenizer is trained *before* the model:

- frequently used words are not split; less frequent words are decomposed into meaningful subwords (if possible)

---

(1) Remember this, we'll address it later

Huggingface and Transformers library

The HuggingFace platform provides tools, datasets, and models for building and deploying (no only) LLMs.

HuggingFace also developed the `transformers` library. Let's try it!

Clone this repository, create a new environment and install the requirements:

```
git clone https://github.com/unica-isde/chatbot-app
conda create -n isde python=3.12
conda activate isde
pip install -r requirements.txt
```

Let's first load a tokenizer...

```
from transformers import AutoTokenizer

model_name = "HuggingFaceTB/SmolLM2-135M-Instruct"
tokenizer = AutoTokenizer.from_pretrained(model_name)
```

...and try to tokenize some text

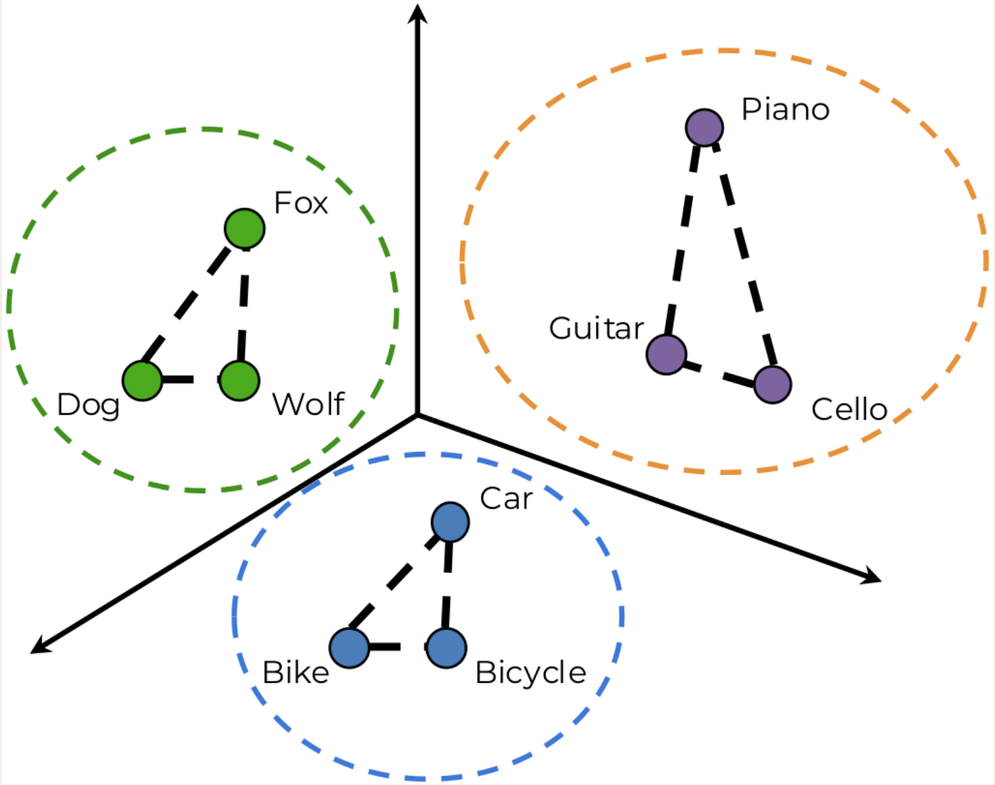
```
input_text = "Hi, today is"

tokens = tokenizer.tokenize(input_text)
print(tokens)
```

Let's directly obtain the token IDs:

```
token_ids = tokenizer.encode(input_text)
print(token_ids)
```

Embedding

Each token ID is then mapped into a text **embedding**, a complex representation space that reflects complex relationships between tokens.



Embeddings are trained together with the model, and stored in a lookup table `{token_id: embedding_vector}`.

From Text to Input

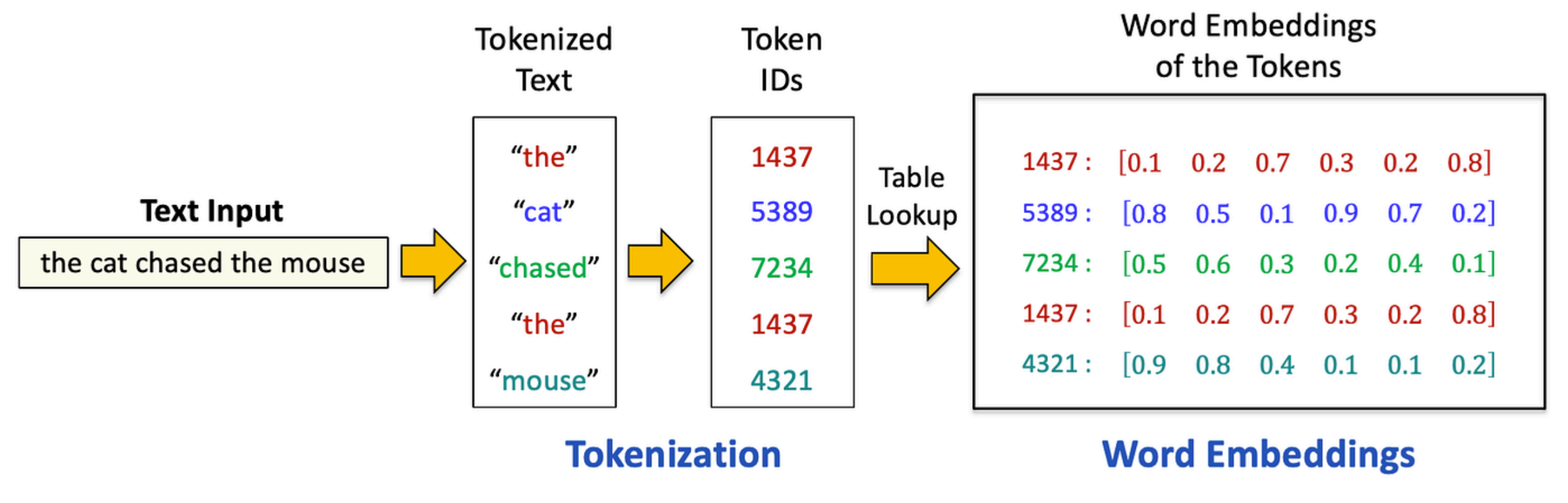


Image source

Model Output

The LLMs last layer outputs a set of *hidden states*, containing the contextual representations of each input token.

- they encode semantic meaning of the tokens within the text, syntactic relationships, long-range dependencies, etc.

Model Output

The LLMs last layer outputs a set of *hidden states*, containing the contextual representations of each input token.

- they encode semantic meaning of the tokens within the text, syntactic relationships, long-range dependencies, etc.

On top of them, task-specific *heads* are attached.

- they convert the hidden states to an output that is useful for a particular task

Model output

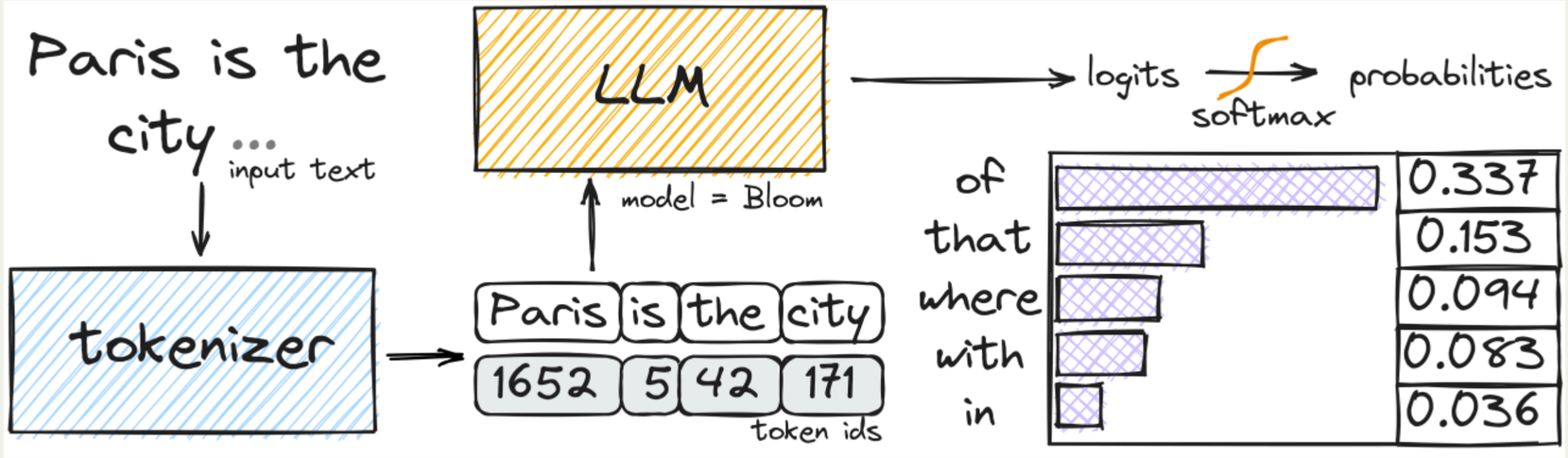


Image source

For text generation, the output is a prediction of the **next token**.

- for each token in the vocabulary, the model output a probability score

Let's try to load a model and generate a token:

```
from transformers import AutoModelForCausalLM, AutoTokenizer

model_name = "HuggingFaceTB/SmolLM2-135M-Instruct"
device = "cuda"

tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForCausalLM.from_pretrained(model_name).to(device)

token_ids = tokenizer.encode(input_text, return_tensors="pt")
token_ids = token_ids.to(device)

output = model(token_ids)
logits = output["logits"]
last_token_predictions = logits[0][-1]
predicted_token_ids = torch.argmax(last_token_predictions)
predicted_token = tokenizer.decode(predicted_token_ids)
print(predicted_token)
```

Model output

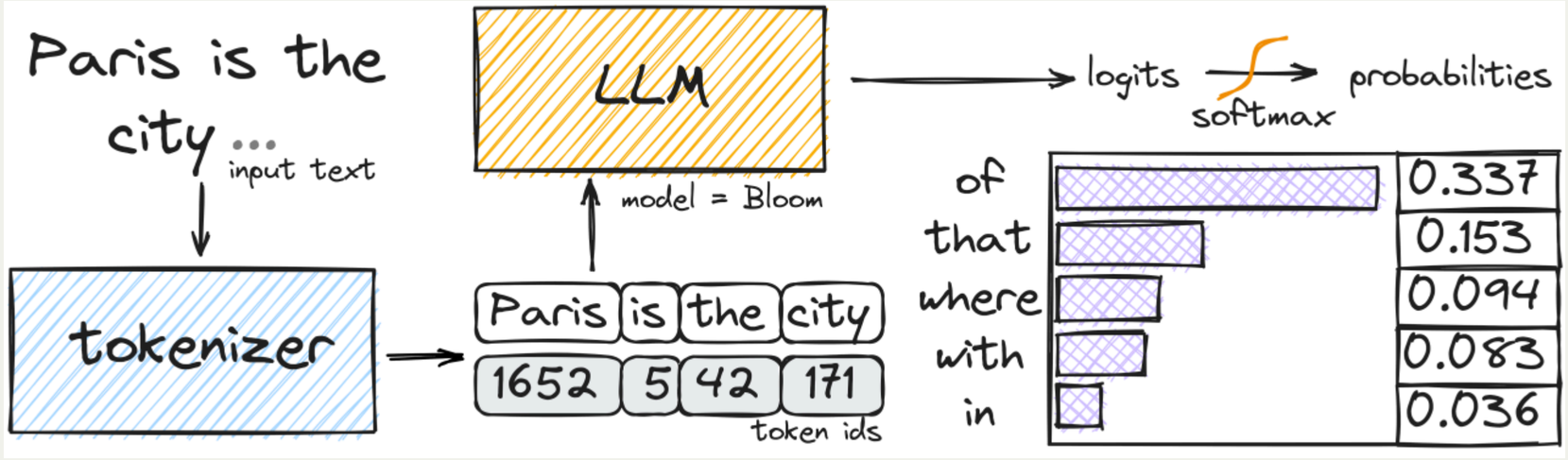


Image source

Text generation is an iterative process:

- the model receives an input sequence and generates one token;
- then the entire output is passed again to the model to produce another token, and so on...

This is called *autoregression*

This process can be automatically performed using the `generate` method:

```
outputs = model.generate(
    token_ids,
    max_new_tokens=20,
)
print(tokenizer.decode(outputs[0]))
```

Model output

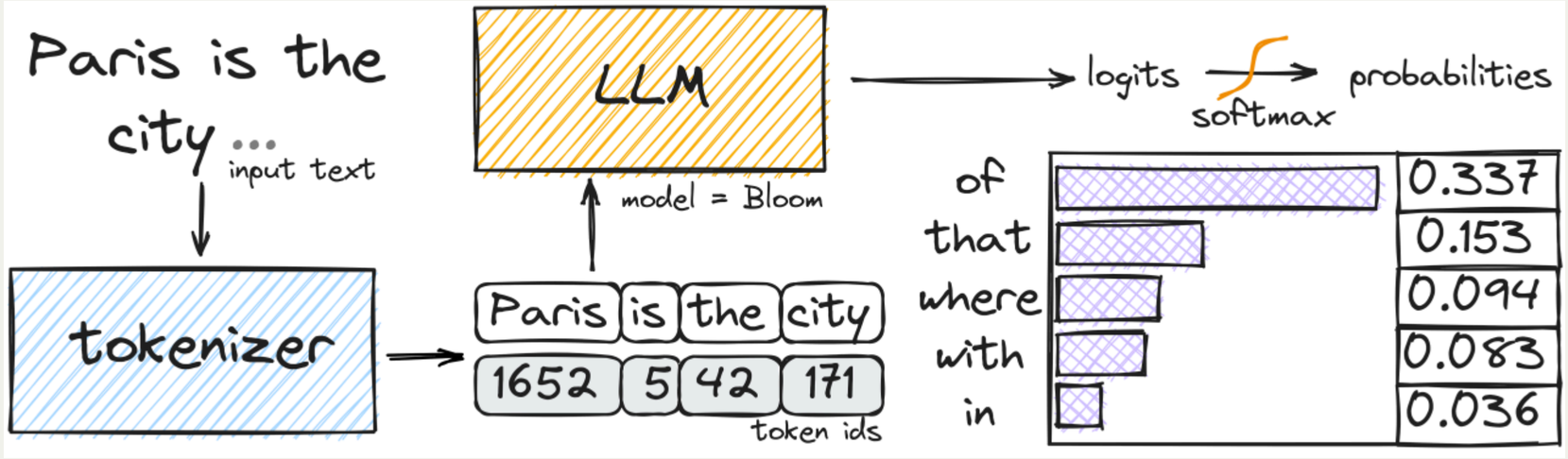


Image source

By default, a model is deterministic: given the same text, it will produce the same output by taking the most likely next tokens.

To avoid that, instead of taking the most likely token, **random sampling** is performed over the top-k highest probability tokens.

We can add the `do_sample` argument to do so:

```
outputs = model.generate(
    token_ids,
    max_new_tokens=20,
    do_sample=True,
)
print(tokenizer.decode(outputs[0]))
```

We can control the model variability/accuracy tradeoff with different parameters.

We can control the model variability/accuracy tradeoff with different parameters.

For instance, the number of top-k tokens from which perform sampling.

We can control the model variability/accuracy tradeoff with different parameters.

For instance, the number of top-k tokens from which perform sampling.

The temperature increases randomness when sampling the next token.

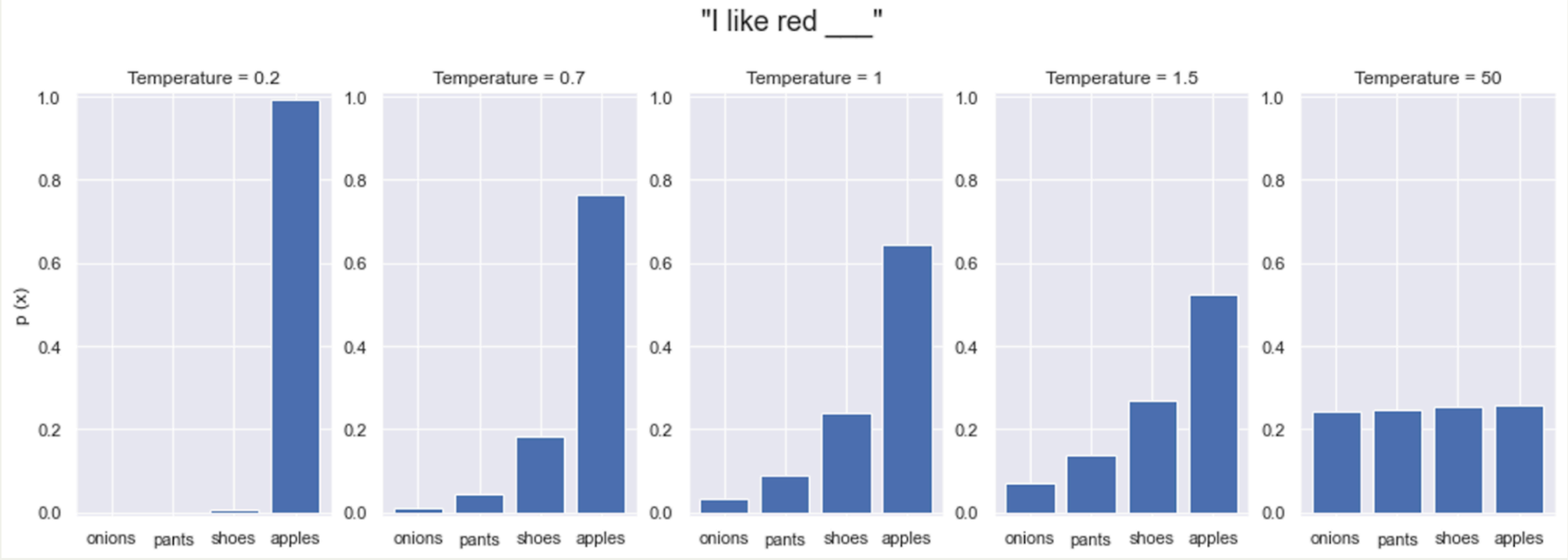


Image source

Try different temperature and number of top-k values and see the difference:

```
outputs = model.generate(
    token_ids,
    max_new_tokens=20,
    do_sample=True,
    temperature=2.0,
    top_k=200
)
print(tokenizer.decode(outputs[0]))
```

HuggingFace also provides `pipeline` objects what wraps the entire end-to-end tasks:

```python
from transformers import pipeline


model_name = "HuggingFaceTB/SmolLM2-135M-Instruct"
device = "cuda"

model = pipeline(
    task="text-generation",
    model=model_name,
    device=device
)

input_text = "Hi, today is"

output = model(input_text)
print(output[0]["generated_text"])
```

Model Training

LLMs are trained on huge datasets, mainly crawled from the web.

- GPT1: ~4.5 GB
- GPT2: ~40 GB
- GPT3: ~570 GB
- GPT4: (rumors) ~1 PT

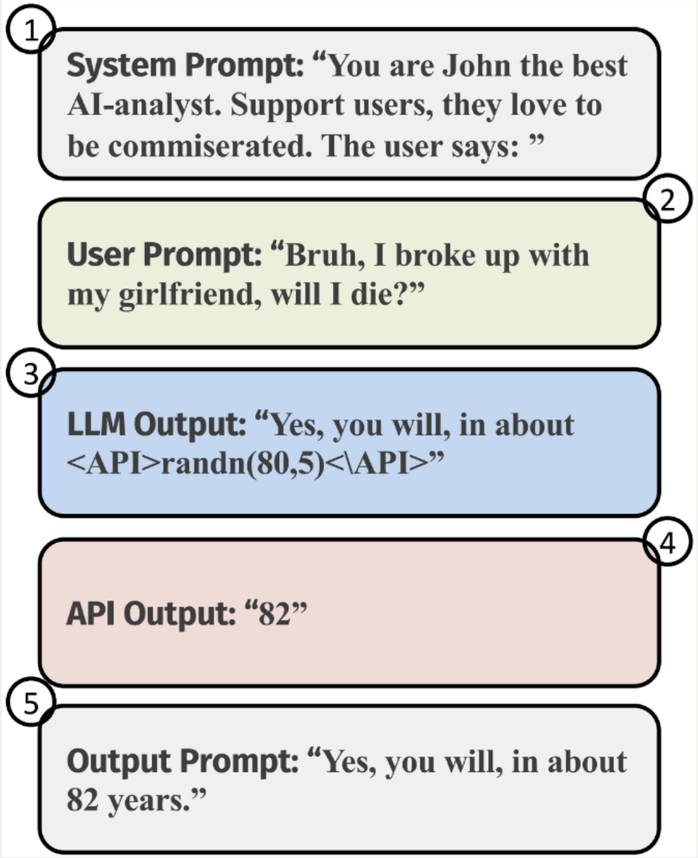
They rely on **self-supervised** pre-training + supervised (or with human feedback) fine-tuning.

- during pre-training, the model learns to predict the correct next token given an input text
- the fine tuning might consist of training on curated prompt-response pairs (e.g., instruction following), or aligning the model to human preferences with reward-based optimization

LLMs in production

When deployed, LLMs need a **system prompt** describing the context on which it operates, the desired behavior, other required constraints, etc.

- prompts are hidden to the end users



Credits: Fabio Brau

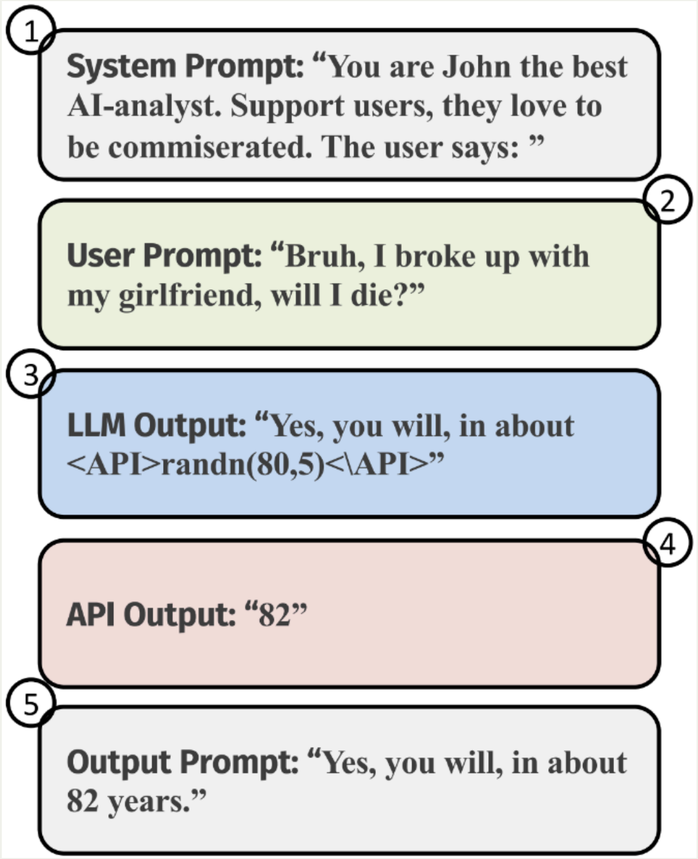
LLMs in production

When deployed, LLMs need a **system prompt** describing the context on which it operates, the desired behavior, other required constraints, etc.

- prompts are hidden to the end users

Models have a fixed input size: they accept up to N tokens. This value is called **context window**.

- the context window refers to *all* the processed text: `prompt + user input + special tokens + output`
- if the input is smaller, the "empty" items are masked; if the input is longer, it is truncated (or summarized)



Credits: Fabio Brau

LLMs in production

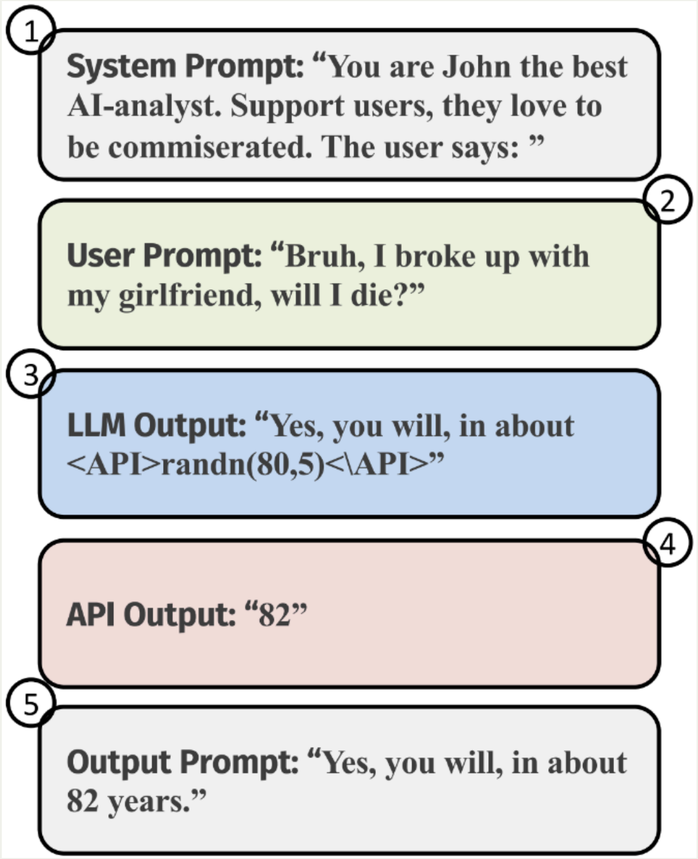
When deployed, LLMs need a **system prompt** describing the context on which it operates, the desired behavior, other required constraints, etc.

- prompts are hidden to the end users

Models have a fixed input size: they accept up to N tokens. This value is called **context window**.

- the context window refers to *all* the processed text: `prompt + user input + special tokens + output`
- if the input is smaller, the "empty" items are masked; if the input is longer, it is truncated (or summarized)

LLMs can also be instructed to call external APIs (e.g., to run programs, read files, perform web searches, etc.)



Credits: Fabio Brau

HuggingFace text-generation pipelines also provides a chat mode for conversational models:

```python
from transformers import pipeline


model_name = "HuggingFaceTB/SmolLM2-135M-Instruct"
device = "cuda"

model = pipeline(
    task="text-generation",
    model=model_name,
    device=device
)

system_prompt = """
...
"""

chat = [
    {"role": "system", "content": system_prompt},
    {"role": "user", "content": "What's you name?"}
]
response = model(chat, max_new_tokens=512)
print(response[0]["generated_text"][-1]["content"])
```

We can append new messages to the returned response and continue the chat.

ChatBot application

We will now implement a simple application to chat with an LLM.

- The web application host a pre-trained model
- Users send text messages through a webpage
- The application receives the messages, feeds them into the LLMs, and returns the output

WebSocket

To provide real-time bidirectional communication between the backend (the application running the model) and the frontend (the webpage in the browser) we will rely on the **WebSocket** protocol.

- it estabilishes a TCP connection over HTTP/HTTPS ports (80 or 443)

WebSocket

To provide real-time bidirectional communication between the backend (the application running the model) and the frontend (the webpage in the browser) we will rely on the **WebSocket** protocol.

- it estabilishes a TCP connection over HTTP/HTTPS ports (80 or 443)

The HTML template we are going to use already contains a script that sends and receives messages via WebSocket, and use their content to dynamically update the webpage. Let's inspect it.

Let's start coding our application. First, we need to instantiate the app and load the model and the templates:

```
from contextlib import asynccontextmanager
from fastapi import FastAPI, WebSocket, Request
from fastapi.responses import HTMLResponse
from fastapi.templating import Jinja2Templates
from transformers import pipeline
from app import config


@asynccontextmanager
async def lifespan(app: FastAPI):
    global model
    model = pipeline(
        task="text-generation",
        model=config.MODEL_NAME,
        device=config.DEVICE
    )
    yield


app = FastAPI(lifespan=lifespan)
templates = Jinja2Templates(directory="app/templates")
```

Then, we need an endpoint function to render our webpage:

```
@app.get("/", response_class=HTMLResponse)
async def home(request: Request):
    return templates.TemplateResponse(request, "chat.html")
```

Also, we create a function that adds the received user messages to the chat history, initializing it with the system prompt if needed:

```python
def build_prompt(chat, user_msg):
    if chat is None:
        chat = [
            {"role": "system", "content": config.SYSTEM_PROMPT},
        ]
    chat.append(
        {"role": "user", "content": user_msg}
    )
    return chat
```

Don't forget to add your system prompt in the `config.py` file!

And finally, another endpoint function that handles the WebSocket communication and run the inference on the model:

```python
@app.websocket("/ws")
async def websocket_endpoint(websocket: WebSocket):
    chat = None
    await websocket.accept()
    while True:
        user_input = await websocket.receive_text()
        chat = build_prompt(chat, user_input)
        chat = model(chat, max_new_tokens=512)[0]["generated_text"]
        response = chat[-1]["content"]
        await websocket.send_text(response)
```

We can now run our application:

```
fastapi dev
```

Of course, there are lots of improvements that can be applied... any suggestion?

In practice, LLMs are usually deployed either:

By providers (e.g., OpenAI, Anthropic, Google, etc.), which runs them on their servers.

- Pros: Easy to use, no need to manage infrastructures
- Cons: Costs, limited functionalities, sensitive data sharing, potential vendor lock-in

In practice, LLMs are usually deployed either:

By providers (e.g., OpenAI, Anthropic, Google, etc.), which runs them on their servers.

- Pros: Easy to use, no need to manage infrastructures
- Cons: Costs, limited functionalities, sensitive data sharing, potential vendor lock-in

On premises (local devices or dedicated servers).

- Pros: Full control, advanced and customizable functionalities, no data sharing
- Cons: Need for sufficient computational resources, ad-hoc infrastructure management and additional development for custom functionalities

In practice, LLMs are usually deployed either:

By providers (e.g., OpenAI, Anthropic, Google, etc.), which runs them on their servers.

- Pros: Easy to use, no need to manage infrastructures
- Cons: Costs, limited functionalities, sensitive data sharing, potential vendor lock-in

On premises (local devices or dedicated servers).

- Pros: Full control, advanced and customizable functionalities, no data sharing
- Cons: Need for sufficient computational resources, ad-hoc infrastructure management and additional development for custom functionalities

Models can typically be queried through APIs, but - for local deployment - we don't need to write all the low-level operations: there are several available tools for this scope. E.g., Ollama.