


Generation with LLMs

[Open in Colab](#)[Open](#)[Studio Lab](#)

LLMs, or Large Language Models, are the key component behind text generation. In a nutshell, they consist of large pretrained transformer models trained to predict the next word (or, more precisely, token) given some input text. Since they predict one token at a time, you need to do something more elaborate to generate new sentences other than just calling the model — you need to do autoregressive generation.

Autoregressive generation is the inference-time procedure of iteratively calling a model with its own generated outputs, given a few initial inputs. In  Transformers, this is handled by the `generate()` method, which is available to all models with generative capabilities.

If you want to jump straight to chatting with a model, [try our chat CLI](#).

This tutorial will show you how to:

- Generate text with an LLM
- Avoid common pitfalls
- Next steps to help you get the most out of your LLM

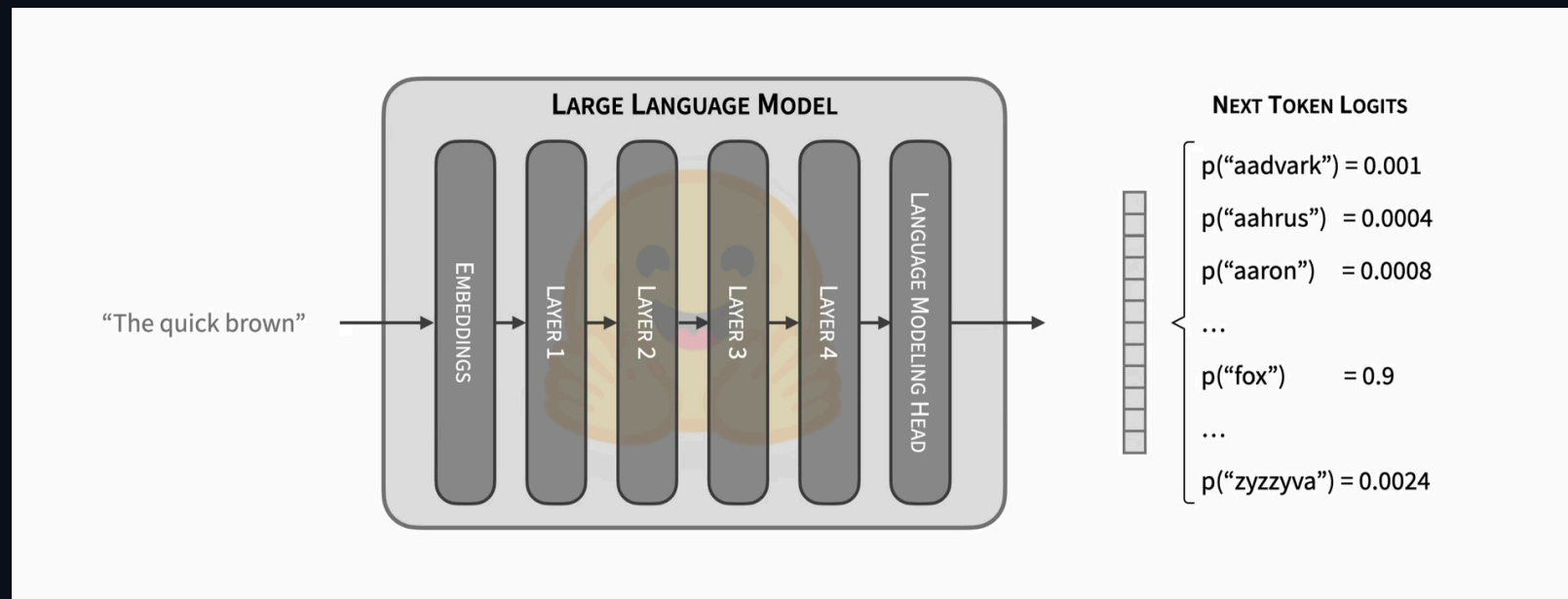
Before you begin, make sure you have all the necessary libraries installed:

```
pip install transformers bitsandbytes>=0.39.0 -q
```

Bitsandbytes supports multiple backends in addition to CUDA-based GPUs. Refer to the multi-backend installation [guide](#) to learn more.

Generate text

A language model trained for causal language modeling takes a sequence of text tokens as input and returns the probability distribution for the next token.



"Forward pass of an LLM"

A critical aspect of autoregressive generation with LLMs is how to select the next token from this probability distribution. Anything goes in this step as long as you end up with a token for the next iteration. This means it can be as simple as selecting the most likely token from the probability distribution or as complex as applying a dozen transformations before sampling from the resulting distribution.



"The quick brown"

"Autoregressive generation iteratively selects the next token from a probability distribution to generate text"

The process depicted above is repeated iteratively until some stopping condition is reached. Ideally, the stopping condition is dictated by the model, which should learn when to output an end-of-sequence (EOS) token. If this is not the case, generation stops when some predefined maximum length is reached.

Properly setting up the token selection step and the stopping condition is essential to make your model behave as you'd expect on your task. That is why we have a `GenerationConfig` file associated with each model, which contains a good default generative parameterization and is loaded alongside your model.

Let's talk code!

If you're interested in basic LLM usage, our high-level `Pipeline` interface is a great starting point. However, LLMs often require advanced features like quantization and fine control of the token selection step, which is best done through `generate()`. Autoregressive generation with LLMs is also resource-intensive and should be executed on a GPU for adequate throughput.

First, you need to load the model.

```
>>> from transformers import AutoModelForCausalLM

>>> model = AutoModelForCausalLM.from_pretrained(
...     "mistralai/Mistral-7B-v0.1", device_map="auto", load_in_4bit=True
... )
```

You'll notice two flags in the `from_pretrained` call:

- `device_map` ensures the model is moved to your GPU(s)
- `load_in_4bit` applies 4-bit dynamic quantization to massively reduce the resource requirements

There are other ways to initialize a model, but this is a good baseline to begin with an LLM.

Next, you need to preprocess your text input with a tokenizer.

```
>>> from transformers import AutoTokenizer
>>> from accelerate.test_utils.testing import get_backend

>>> DEVICE, _, _ = get_backend() # automatically detects the underlying device type (CUDA, CPU, XPU, MPS,
>>> tokenizer = AutoTokenizer.from_pretrained("mistralai/Mistral-7B-v0.1", padding_side="left")
>>> model_inputs = tokenizer(["A list of colors: red, blue"], return_tensors="pt").to(DEVICE)
```

The `model_inputs` variable holds the tokenized text input, as well as the attention mask. While `generate()` does its best effort to infer the attention mask when it is not passed, we recommend passing it whenever possible for optimal results.

After tokenizing the inputs, you can call the `generate()` method to return the generated tokens. The generated tokens then should be converted to text before printing.

```
>>> generated_ids = model.generate(**model_inputs)
>>> tokenizer.batch_decode(generated_ids, skip_special_tokens=True)[0]
'A list of colors: red, blue, green, yellow, orange, purple, pink,'
```

Finally, you don't need to do it one sequence at a time! You can batch your inputs, which will greatly improve the throughput at a small latency and memory cost. All you need to do is to make sure you pad your inputs properly (more on that below).

```
>>> tokenizer.pad_token = tokenizer.eos_token # Most LLMs don't have a pad token by default
>>> model_inputs = tokenizer(
...     ["A list of colors: red, blue", "Portugal is"], return_tensors="pt", padding=True
... ).to(DEVICE)
>>> generated_ids = model.generate(**model_inputs)
>>> tokenizer.batch_decode(generated_ids, skip_special_tokens=True)
['A list of colors: red, blue, green, yellow, orange, purple, pink,',
'Portugal is a country in southwestern Europe, on the Iber']
```

And that's it! In a few lines of code, you can harness the power of an LLM.

Common pitfalls

There are many generation strategies, and sometimes the default values may not be appropriate for your use case. If your outputs aren't aligned with what you're expecting, we've created a list of the most common pitfalls and how to avoid them.

```
>>> from transformers import AutoModelForCausalLM, AutoTokenizer

>>> tokenizer = AutoTokenizer.from_pretrained("mistralai/Mistral-7B-v0.1")
>>> tokenizer.pad_token = tokenizer.eos_token # Most LLMs don't have a pad token by default
>>> model = AutoModelForCausalLM.from_pretrained(
...     "mistralai/Mistral-7B-v0.1", device_map="auto", load_in_4bit=True
... )
```

Generated output is too short/long

If not specified in the `GenerationConfig` file, `generate` returns up to 20 tokens by default. We highly recommend manually setting `max_new_tokens` in your `generate` call to control the maximum number of new tokens it can return. Keep in mind LLMs (more precisely, decoder-only models) also return the input prompt as part of the output.

```
>>> model_inputs = tokenizer(["A sequence of numbers: 1, 2"], return_tensors="pt").to(DEVICE)

>>> # By default, the output will contain up to 20 tokens
>>> generated_ids = model.generate(**model_inputs)
>>> tokenizer.batch_decode(generated_ids, skip_special_tokens=True)[0]
'A sequence of numbers: 1, 2, 3, 4, 5'

>>> # Setting 'max_new_tokens' allows you to control the maximum length
>>> generated_ids = model.generate(**model_inputs, max_new_tokens=50)
>>> tokenizer.batch_decode(generated_ids, skip_special_tokens=True)[0]
'A sequence of numbers: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,'
```

Incorrect generation mode

By default, and unless specified in the `GenerationConfig` file, `generate` selects the most likely token at each iteration (greedy decoding). Depending on your task, this may be undesirable; creative tasks like chatbots or writing an essay benefit from sampling. On the other hand, input-grounded tasks like audio transcription or translation benefit from greedy decoding. Enable sampling with `do_sample=True`, and you can learn more about this topic in this [blog post](#).

```
>>> # Set seed for reproducibility -- you don't need this unless you want full reproducibility
>>> from transformers import set_seed
>>> set_seed(42)
```

```
>>> model_inputs = tokenizer(["I am a cat."], return_tensors="pt").to(DEVICE)

>>> # LLM + greedy decoding = repetitive, boring output
>>> generated_ids = model.generate(**model_inputs)
>>> tokenizer.batch_decode(generated_ids, skip_special_tokens=True)[0]
'I am a cat. I am a cat. I am a cat. I am a cat'

>>> # With sampling, the output becomes more creative!
>>> generated_ids = model.generate(**model_inputs, do_sample=True)
>>> tokenizer.batch_decode(generated_ids, skip_special_tokens=True)[0]
'I am a cat. Specifically, I am an indoor-only cat. I'
```

Wrong padding side

LLMs are decoder-only architectures, meaning they continue to iterate on your input prompt. If your inputs do not have the same length, they need to be padded. Since LLMs are not trained to continue from pad tokens, your input needs to be left-padded. Make sure you also don't forget to pass the attention mask to generate!

```
>>> # The tokenizer initialized above has right-padding active by default: the 1st sequence,
>>> # which is shorter, has padding on the right side. Generation fails to capture the logic.
>>> model_inputs = tokenizer(
...     ["1, 2, 3", "A, B, C, D, E"], padding=True, return_tensors="pt"
... ).to(DEVICE)
>>> generated_ids = model.generate(**model_inputs)
>>> tokenizer.batch_decode(generated_ids, skip_special_tokens=True)[0]
'1, 2, 333333333333'
```



```
>>> # With left-padding, it works as expected!
>>> tokenizer = AutoTokenizer.from_pretrained("mistralai/Mistral-7B-v0.1", padding_side="left")
>>> tokenizer.pad_token = tokenizer.eos_token # Most LLMs don't have a pad token by default
>>> model_inputs = tokenizer(
...     ["1, 2, 3", "A, B, C, D, E"], padding=True, return_tensors="pt"
... ).to(DEVICE)
>>> generated_ids = model.generate(**model_inputs)
>>> tokenizer.batch_decode(generated_ids, skip_special_tokens=True)[0]
'1, 2, 3, 4, 5, 6,'
```

Wrong prompt

Some models and tasks expect a certain input prompt format to work properly. When this format is not applied, you will get a silent performance degradation: the model kinda works, but not as well as if you were following the expected prompt. More information about prompting, including which models and tasks need to be careful, is available in this [guide](#). Let's see an example with a chat LLM, which makes use of [chat templating](#):

```
>>> tokenizer = AutoTokenizer.from_pretrained("HuggingFaceH4/zephyr-7b-alpha")
>>> model = AutoModelForCausalLM.from_pretrained(
...     "HuggingFaceH4/zephyr-7b-alpha", device_map="auto", load_in_4bit=True
... )
>>> set_seed(0)
>>> prompt = """How many helicopters can a human eat in one sitting? Reply as a thug."""
>>> model_inputs = tokenizer([prompt], return_tensors="pt").to(DEVICE)
>>> input_length = model_inputs.input_ids.shape[1]
>>> generated_ids = model.generate(**model_inputs, max_new_tokens=20)
```

```

>>> print(tokenizer.batch_decode(generated_ids[:, input_length:], skip_special_tokens=True)[0])
"I'm not a thug, but i can tell you that a human cannot eat"
>>> # Oh no, it did not follow our instruction to reply as a thug! Let's see what happens when we write
>>> # a better prompt and use the right template for this model (through `tokenizer.apply_chat_template`,

>>> set_seed(0)
>>> messages = [
...     {
...         "role": "system",
...         "content": "You are a friendly chatbot who always responds in the style of a thug",
...     },
...     {"role": "user", "content": "How many helicopters can a human eat in one sitting?"},
... ]
>>> model_inputs = tokenizer.apply_chat_template(messages, add_generation_prompt=True, return_tensors="pt")
>>> input_length = model_inputs.shape[1]
>>> generated_ids = model.generate(model_inputs, do_sample=True, max_new_tokens=20)
>>> print(tokenizer.batch_decode(generated_ids[:, input_length:], skip_special_tokens=True)[0])
'None, you thug. How bout you try to focus on more useful questions?'
>>> # As we can see, it followed a proper thug style 🤖

```

Further resources

While the autoregressive generation process is relatively straightforward, making the most out of your LLM can be a challenging endeavor because there are many moving parts. For your next steps to help you dive deeper into LLM usage and understanding:

Advanced generate usage

1. Guide on how to [control different generation methods](#), how to set up the generation configuration file, and how to stream the output;
2. [Accelerating text generation](#);
3. [Prompt templates for chat LLMs](#);
4. [Prompt design guide](#);
5. API reference on [GenerationConfig](#), [generate\(\)](#), and [generate-related classes](#). Most of the classes, including the logits processors, have usage examples!

LLM leaderboards

1. [Open LLM Leaderboard](#), which focuses on the quality of the open-source models;
2. [Open LLM-Perf Leaderboard](#), which focuses on LLM throughput.

Latency, throughput and memory utilization

1. Guide on how to [optimize LLMs for speed and memory](#);
2. Guide on [quantization](#) such as bitsandbytes and autogptq, which shows you how to drastically reduce your memory requirements.

Related libraries

1. [optimum](#), an extension of 🤗 Transformers that optimizes for specific hardware devices;
2. [outlines](#), a library where you can constrain text generation (e.g. to generate JSON files);

3. [SynCode](#), a library for context-free grammar guided generation (e.g. JSON, SQL, Python);
4. [text-generation-inference](#), a production-ready server for LLMs;
5. [text-generation-webui](#), a UI for text generation;
6. [logits-processor-zoo](#), containing additional options to control text generation with 🧐 Transformers. See our related [blog post](#).

↔ [Update](#) on GitHub

← Agents, supercharged - Multi-agents, External tools, and more

Chatting with Transformers →