3-ways to Set up LLaMA 2 Locally on CPU (Part 1 — llama-cpp-python)

Antoine Frd · Follow

8 min read · Jan 17, 2024

What is Llama 2?

Llama 2 is an Open Source Large Language Model released by Meta. It's a chat model from 7 to 70 billions parameters trained on a massive dataset of text from the internet.

Why locally?

The goal of using Llama 2 locally is to have a powerful and flexible open-source LLM model at our fingertips, without relying on remote servers. When you use models locally, you don't need any API token or subscription plan.

In general, businesses that require a high degree of control, customization, security, and performance may find it advantageous to use Llama 2 locally, rather than relying on server-based solutions.

Here are some business use-cases where using Llama 2 locally may be better:

1. **Data Privacy and Security:** Businesses that deal with sensitive data, such as financial institutions, healthcare providers, and law firms, may prefer to use Llama 2 locally to ensure that their data remains private and secure.
2. **Compliance:** Businesses that need to comply with data residency regulations, may find it easier to use Llama 2 locally to ensure that their data is stored and processed within their own premises.
3. **Speed and Performance:** Businesses that require fast response times and high-performance processing, such as real-time chatbots or voice assistants, may find that using Llama 2 locally results in faster response times and better performance.
4. **Offline Access:** Businesses that need to access Llama2 in areas with limited or no internet connectivity may find it useful to have it installed locally.

Getting the model

All Llama 2 models are available on HuggingFace. In order to access them you have to apply for an access token by accepting the terms and conditions.

In this tutorial we are interested in the **CPU** version of **Llama 2**. Usually big and performant Deep Learning models require high-end GPU's to be ran. However, we have llama.cpp, which allows us to run **LLama** models easily on CPU. The llama.cpp applies a custom quantization approach to compress the models in a GGUF format. This reduces the size and resources needed.

> ***About GGUF***
> *GGUF is a new format introduced by the llama.cpp team on August 21st 2023. It is a replacement for*

> *GGML, which is no longer supported by llama.cpp. GGUF offers numerous advantages over GGML, such as better tokenisation, and support for special tokens. It is also supports metadata, and is designed to be extensible.*

Load LlaMA 2 model with `llama-cpp-python` 🚀

Install dependencies for running LLaMA locally

Since we're writing our code in Python, we need to execute the `llama.cpp` in a Python-friendly manner. Fortunately, the community has already considered this and created a project called `llama-cpp-python`, which allows us to integrate llama.cpp seamlessly into our Python code.

llama-cpp-python is a project based on `lama.cpp` which allow you to run Llama models on your local Machine by **4-bits Quantization**.

> ***Quantization:*** *Quantization refers to the process of reducing the precision of a model's weights and activations from floating-point numbers to integers. This can be useful for deploying models on devices with limited computational resources, as it can reduce memory usage and improve computational efficiency.*
>
> Example: *Imagine you have a big suitcase full of clothes that you want to take on a trip. The suitcase is very heavy, and it would be hard to carry it around. So, you decide to reduce the number of clothes in the suitcase by taking out some of them and putting them in a smaller bag. This makes the suitcase lighter and easier to carry. In the same way, quantization in LLMs is like reducing the number of "clothes" (weights and activations) in the model's "suitcase" (memory) to make it lighter and more efficient. This way, the model can be "carried" (deployed) on devices with limited resources, like a smartphone or a small computer, without taking up too much space or using too much memory.*

```
cd Install-Llama2-locally

# Create a new venv environment
python3 -m venv .venv

# Activate the environment
source .venv/bin/activate
```

First, we install it in our local machine using `pip`:

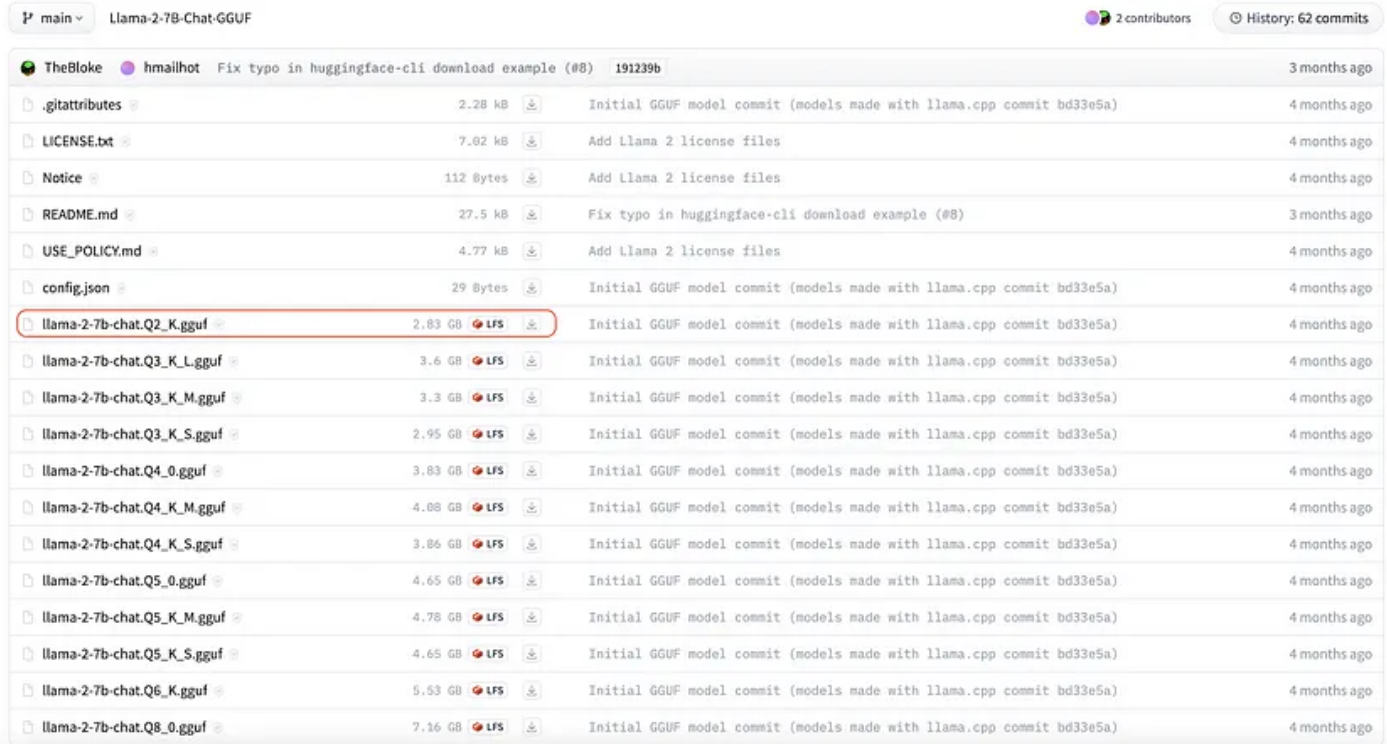
```
pip3 install llama-cpp-python
```

```
(venv) antoine@antoines-mbp Llama2 locally % pip3 install llama-cpp-python
Collecting llama-cpp-python
  Downloading llama_cpp_python-0.2.27.tar.gz (9.4 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 9.4/9.4 MB 35.7 MB/s eta 0:00:00
  Installing build dependencies ... done
  Getting requirements to build wheel ... done
  Installing backend dependencies ... done
  Preparing metadata (pyproject.toml) ... done
Collecting typing-extensions>=4.5.0 (from llama-cpp-python)
  Obtaining dependency information for typing-extensions>=4.5.0 from https://files.pythonhosted.org/packages/b7/f4/6a90020cd2d93349b442b
ns-4.9.0-py3-none-any.whl.metadata
  Downloading typing_extensions-4.9.0-py3-none-any.whl.metadata (3.0 kB)
Collecting numpy>=1.20.0 (from llama-cpp-python)
  Obtaining dependency information for numpy>=1.20.0 from https://files.pythonhosted.org/packages/55/78/f85aab3bda3ddffe6ce8c590190b5f0c
acosx_11_0_arm64.whl.metadata
  Downloading numpy-1.26.3-cp311-cp311-macosx_11_0_arm64.whl.metadata (115 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 115.1/115.1 kB 4.9 MB/s eta 0:00:00
Collecting diskcache>=5.6.1 (from llama-cpp-python)
  Obtaining dependency information for diskcache>=5.6.1 from https://files.pythonhosted.org/packages/3f/27/4570e78fc0bf5ea0ca45eb1de3818
e-any.whl.metadata
  Downloading diskcache-5.6.3-py3-none-any.whl.metadata (20 kB)
Downloading diskcache-5.6.3-py3-none-any.whl (45 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 45.5/45.5 kB 1.6 MB/s eta 0:00:00
Downloading numpy-1.26.3-cp311-cp311-macosx_11_0_arm64.whl (14.0 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 14.0/14.0 MB 49.0 MB/s eta 0:00:00
Downloading typing_extensions-4.9.0-py3-none-any.whl (32 kB)
Building wheels for collected packages: llama-cpp-python
  Building wheel for llama-cpp-python (pyproject.toml) ... done
  Created wheel for llama-cpp-python: filename=llama_cpp_python-0.2.27-cp311-cp311-macosx_12_0_arm64.whl size=2000214 sha256=e6a9648be65
  Stored in directory: /Users/antoine/Library/Caches/pip/wheels/0e/27/a4/13df52c36a09d5eaab1bd43ccce1bdee5d5a4e282537267fdd
Successfully built llama-cpp-python
Installing collected packages: typing-extensions, numpy, diskcache, llama-cpp-python
Successfully installed diskcache-5.6.3 llama-cpp-python-0.2.27 numpy-1.26.3 typing-extensions-4.9.0
```

Note: The default `pip install llama-cpp-python` behaviour is to build `llama.cpp` for CPU only on Linux and Windows and use Metal on MacOS.

Download the model from HuggingFace

We download the *llama-2–7b-chat.Q2_K.gguf* file, which is the most compressed version of the 7B chat model and requires the least resources.



And we add it to our `models` directory.

```
.
└── models
    └── llama-2-7b-chat.Q2_K.gguf
```

Note: The Hugging Face models provided by **TheBloke** have a `Provided files` section that reveals the RAM requirements for running models with various quantization sizes and methods.

Running the model using `llama_cpp` library

Prompt template

As mentionned in **The Bloke Huging Face model page** , for the Llama2-Chat models, the prompt template has to have a specific format as bellow:

```
[INST] <<SYS>>
You are a helpful, respectful and honest assistant. Always answer as helpfully as possible,
<</SYS>>
{prompt}[/INST]
```

See more about Llama2-Chat templates here.

To test the model, we can use a notebook to see the results:

```
from llama_cpp import Llama

# Put the location of to the GGUF model that you've download from HuggingFace here
model_path = "models/llama-2-7b-chat.Q2_K.gguf"
llm = Llama(model_path=model_path)

# Prompt creation
system_message = "You are a helpful assistant"
user_message = "Q: Name the planets in the solar system? A: "

prompt = f"""<s>[INST] <<SYS>>
{system_message}
<</SYS>>
{user_message} [/INST]"""

# Run the model
output = llm(
  prompt, # Prompt
  max_tokens=32, # Generate up to 32 tokens
```

```
    stop=["Q:", "\n"], # Stop generating just before the model would generate a new question
    echo=True # Echo the prompt back in the output
) # Generate a completion, can also call create_completion

print(output)
```

The LLM call returns a dictionary:


```
{
  "id": "cmpl-4fa2786c-ddaa-451f-8238-2aeeeb150e39",
  "object": "text_completion",
  "created": 1704621144,
  "model": "./models/llama-2-7b-chat.Q2_K.gguf",
  "choices": [
        {
         "text": "<s>[INST] <<SYS>>\\nYou are a helpful assistant\\n<</SYS>>\\nQ: Name the
         "index": 0,
         "logprobs": None,
         "finish_reason": "stop"
  }
],
"usage": {
        "prompt_tokens": 40,
        "completion_tokens": 23,
        "total_tokens": 63
  }
}
```


We can get the text output of the model by writing

```
print(output["choices"][0]["text"]
```

As we chose the 7b-chat model, the result may give some bad results.

Running the model using `Langchain`

There exists a LlamaCpp LLM wrapper in Langchain, which we can access with

```
pip3 install langchain
```

```
from langchain_community.llms import LlamaCpp
```

Next, we can import the following libraries

```
from langchain.callbacks.manager import CallbackManager
from langchain.callbacks.streaming_stdout import StreamingStdOutCallbackHandler
from langchain.chains import LLMChain
from langchain.prompts import PromptTemplate
from langchain_community.llms import LlamaCpp
```

One of the most useful features of `LangChain` is the ability to create **prompt templates.** A prompt template is a string that contains a placeholder for input variable(s).

> "A prompt template refers to a reproducible way to generate a prompt"

Prompt templates can contain the following:

> *Instructions to the language model.*
>
> A *set of few shot examples to help the language model generate a better response.*
>
> A *question to the language model.*

Let's see how we can use them:

```
from langchain import PromptTemplate

template = """
<s>[INST] <<SYS>>
Act as an Astronomer engineer who is teaching high school students.
<</SYS>>

{text} [/INST]
"""

prompt = PromptTemplate(
```

```
    input_variables=["text"],
    template=template,
)
```

The variable must be surrounded by `{}`. The `input_variables` argument is a list of **variable names** that will be used to format the template.

```
text = "Explain what is the solar system in 2-3 sentences"
print(prompt.format(text=text))
```

```
# Callbacks support token-wise streaming
callback_manager = CallbackManager([StreamingStdOutCallbackHandler()])
```

Streaming text output is gaining popularity among large language models (LLMs) and chatbots, offering a more dynamic experience for users. Unlike traditional generation methods that wait for completion, streaming sends text incrementally, allowing for a more interactive experience. With Lanchain we can turn on streaming token using the `StreamingStdOutCallbackHandler` handler. Learn more about Streaming with Langchain with this article.

Example using the LLaMA 2–7B chat model:

```
model_path = "models/llama-2-7b-chat.Q2_K.gguf"
llm = LlamaCpp(
    model_path=model_path,
    temperature=0.5,
    max_tokens=500,
    top_p=1,
    callback_manager=callback_manager,
    verbose=True,  # Verbose is required to pass to the callback manager
)

output = llm.invoke(prompt.format(text=text))
print(output)
```

> *Note:* Use `llm.invoke(prompt)` invoke instead of `llm(prompt)`. The function `__call__` was deprecated in LangChain 0.1.7 and will be removed in 0.2.0.

```
Ah, high school students! *adjusts glasses* Excellent! The solar system is a fascinating top
```

- `streaming=True` enables streaming capabilities
- `StreamingStdoutCallbackHandler` prints each new token

```
the solar system is a collection of celestial bodies that orbit around our sun, including eight planets and various dwarf planets, asteroids,
comets, and other small bodies. Our solar system formed around 4.6
```

Streaming response

Congratulations! We've successfully installed Llama 2 and Langchain locally. We are now ready to use it. In the next step, we'll explore another method using **Ollama**. Stay tuned!

Llm | Generative Ai Use Cases | Llama 2 | Hugging Face | Langchain

Follow

Written by Antoine Frd

16 Followers

AI Engineer

More from Antoine Frd