Member-only story

GPT4All is the Local ChatGPT for your documents… and it is free!

How to install GPT4All on your Laptop and ask AI about your own domain knowledge (your documents)… and it runs on CPU only!

Fabio Matricardi
Artificial Corner

Fabio Matricardi

.

Follow

Published in

Artificial Corner

.
20 min read
·
May 13

--

40

Listen

Share

I talked in my previous article about how to use open source LLM to answer questions about your own documents: but we were still using Google Colab.

Answering Question About your Documents Using LangChain (and NOT OpenAI)

How to use Hugging Face LLM (open source LLM) to talk to your documents, pdfs and also articles from webpages.

artificialcorner.com

But what if you can do the same on your local machine (or any other personal servers or machines)?

In this article we will learn how to deploy and use GPT4All model on your CPU only computer (I am using a Macbook Pro without GPU!)

Use GPT4All on Your Computer — Picture by the author

In this article we are going to install on our local computer GPT4All (a powerful LLM) and we will discover how to interact with our documents with python. A collection of PDFs or online articles will be the knowledge base for our question/answers.

What is GPT4All

From the official website GPT4All it is described as a free-to-use, locally running, privacy-aware chatbot. No GPU or internet required.

GTP4All is an ecosystem to train and deploy powerful and customized large language models that run locally on consumer grade CPUs.

Our GPT4All model is a 4GB file that you can download and plug into the GPT4All open-source ecosystem software. Nomic AI facilitates high quality and secure software ecosystems, driving the effort to enable individuals and organizations to effortlessly train and implement their own large language models locally.

How it will work?

Workflow of the QnA with GPT4All — created by the author

The process is really simple (when you know it) and can be repeated with other models too. The steps are as follows:

- load the GPT4All model
- use Langchain to retrieve our documents and Load them
- split the documents in small chunks digestible by Embeddings
- Use FAISS to create our vector database with the embeddings
- Perform a similarity search (semantic search) on our vector database based on the question we want to pass to GPT4All: this will be used as a context for our question
- Feed the question and the context to GPT4All with Langchain and wait for the the answer.

So what we need is Embeddings. An embedding is a numerical representation of a piece of information, for example, text, documents, images, audio, etc. The representation captures the semantic meaning of what is being embedded, and this is exactly what we need. For this project we cannot rely on heavy GPU models: so we will download the Alpaca native model and use from Langchain the LlamaCppEmbeddings. Don't worry! Everything is explained step by step

Let's start coding

Create a Virtual Environment

Create a new folder for your new Python project, for example GPT4ALL_Fabio (put your name…):

```
mkdir GPT4ALL_Fabio
cd GPT4ALL_Fabio
```

Next, create a new Python virtual environment. If you have more than one python version installed, specify your desired version: in this case I will use my main installation, associated to python 3.10.

```
python3 –m venv .venv
```

The command python3 -m venv .venv creates a new virtual environment named .venv (the dot will create a hidden directory called venv).

A virtual environment provides an isolated Python installation, which allows you to install packages and dependencies just for a specific project without affecting the system-wide Python installation or other projects. This isolation helps maintain consistency and prevent potential conflicts between different project requirements.

Once the virtual environment is created, you can activate it using the following command:

```
source .venv/bin/activate
```

Activated virtual environment

The libraries to install

For the project we are building we don't need too many packages. We need only:

- python bindings for GPT4All
- Langchain to interact with our documents

LangChain is a framework for developing applications powered by language models. It allows you not only to call out to a language model via an API, but also connect a language model to other sources of data and allow a language model to interact with its environment.

```
pip install pygpt4all==1.0.1
pip install pyllamacpp==1.0.6
pip install langchain==0.0.149
pip install unstructured==0.6.5
pip install pdf2image==1.16.3
pip install pytesseract==0.3.10
pip install pypdf==3.8.1
pip install faiss-cpu==1.7.4
```

For LangChain you see that we specified also the version. This library is receiving a lot of updates recently, so to be certain the our setup is going to work also tomorrow it is better to specify a version we know is working fine. Unstructured is a required dependency for the pdf loader and pytesseract and pdf2image as well.

NOTE: on the GitHub repository there is a requirements.txt file (suggested by jl adcr) with all the versions associated to this project. You can do the installation in one shot, after downloading it into the main project file directory with the following command:

```
pip install -r requirements.txt
```

At the end of the article I created a section for the troubleshooting. The GitHub repo has also an updated READ.ME with all these information.

Bear in mind that some libraries have versions available depending on the python version you are running on your virtual environment.

Download on your PC the models

This is a really important step.

For the project we certainly need GPT4All. The process described on Nomic AI is really complicated and requires hardware that not all of us have (like me). So here is the link to the model already converted and ready to be used. Just click on download.

Download the GPT4All model

As described briefly in the introduction we need also the model for the embeddings, a model that we can run on our CPU without crushing. Click the link here to download the alpaca-native-7B-ggml already converted to 4-bit and ready to use to act as our model for the embedding.

Click the download arrow next to ggml-model-q4_0.bin

Why we need embeddings? If you remember from the flow diagram the first step required, after we collect the documents for our knowledge base, is to embed them. The LLamaCPP embeddings from this Alpaca model fit the job perfectly and this model is quite small too (4 Gb). By the way you can also use the Alpaca model for your QnA!

Update 2023.05.25: Mani Windows users are facing problems to use the llamaCPP embeddings. This mainly happens because during the installation of the python package llama-cpp-python with:

```
pip install llama-cpp-python
```

the pip package is going to compile from source the library. Windows usually does not have CMake or C compiler installed by default on the machine. But don't warry there is a solution

Running the installation of llama-cpp-python, required by LangChain with the llamaEmbeddings, on windows CMake C complier is not installed by default, so you cannot build from source.
On Mac Users with Xtools and on Linux, usually the C complier is already available on the OS.
To avoid the issue you MUST use pre complied wheel.
Go here https://github.com/abetlen/llama-cpp-python/releases
and look for the complied wheel for your architecture and python version — you MUST take Weels Version 0.1.49 because higher versions are not compatible.

Screenshot from https://github.com/abetlen/llama-cpp-python/releases

In my case I have Windows 10, 64 bit, python 3.10
so my file is llama_cpp_python-0.1.49-cp310-cp310-win_amd64.whl

This issue is tracked on the GitHub repository

After downloading you need to put the two models in the models directory, as shown below.

Directory structure and where to put the model files

Basic Interaction with GPT4All

Since we want to have control of our interaction the the GPT model, we have to create a python file (let's call it pygpt4all_test.py), import the dependencies and give the instruction to the model. You will see that is quite easy.

```
from pygpt4all.models.gpt4all import GPT4All
```

This is the python binding for our model. Now we can call it and start asking. Let's try a creative one.

We create a function that read the callback from the model, and we ask GPT4All to complete our sentence.

```
def new_text_callback(text):
    print(text, end="")

model = GPT4All('./models/gpt4all-converted.bin')
model.generate("Once upon a time, ", n_predict=55, new_text_callback=new_text_callback)
```

The first statement is telling our program where to find the model (remember what we did in the section above)

The second statement is asking the model to generate a response and to complete our prompt "Once upon a time, ".

To run it, make sure that the virtual environment is still activated and simply run :

```
python3 pygpt4all_test.py
```

You should se a loading text of the model and the completion of the sentence. Depending on your hardware resources it may take a little time.

The result may be different from yours… But for us the important is that it is working and we can proceed with LangChain to create some advanced stuff.

NOTE (updated 2023.05.23): if you face an error related to pygpt4all, check the troubleshooting section on this topic with the solution given by Rajneesh Aggarwal or by Oscar Jeong.

LangChain template on GPT4All

LangChain framework is a really amazing library. It provides Components to work with language models in a easy to use way, and it also provides Chains. Chains can be thought of as assembling these components in particular ways in order to best accomplish a particular use case. These are intended to be a higher level interface through which people can easily get started with a specific use case. These chains are also designed to be customizable.

In our next python test we will use a Prompt Template. Language models take text as input — that text is commonly referred to as a prompt. Typically this is not simply a hardcoded string but rather a combination of a template, some examples, and user input. LangChain provides several classes and functions to make constructing and working with prompts easy. Let's see how we can do it too.

Create a new python file and call it my_langchain.py

```
# Import of langchain Prompt Template and Chain
from langchain import PromptTemplate, LLMChain
# Import llm to be able to interact with GPT4All directly from langchain
from langchain.llms import GPT4All
# Callbacks manager is required for the response handling
from langchain.callbacks.base import CallbackManager
from langchain.callbacks.streaming_stdout import StreamingStdOutCallbackHandler

local_path = './models/gpt4all-converted.bin'
callback_manager = CallbackManager([StreamingStdOutCallbackHandler()])
```

We imported from LangChain the Prompt Template and Chain and GPT4All llm class to be able to interact directly with our GPT model.

Then, after setting our llm path (as we did before) we instantiate the callback managers so that we are able to catch the responses to our query.

To create a template is really easy: following the documentation tutorial we can use something like this…

```
template = """Question: {question}

Answer: Let's think step by step on it.

"""
prompt = PromptTemplate(template=template, input_variables=["question"])
```

The template variable is a multi-line string that contains our interaction structure with the model: in curly braces we insert the external variables to

the template, in our scenario is our question.

Since it is a variable you can decide if it is an hard-coded question or an user input question: here the two examples.

```
# Hardcoded question
question = "What Formula 1 pilot won the championship in the year Leonardo di Caprio was born?"

# User input question...
question = input("Enter your question: ")
```

For our test run we will comment the user input one. Now we only need to link together our template, the question and the language model.

```
template = """Question: {question}

Answer: Let's think step by step on it.

"""
prompt = PromptTemplate(template=template, input_variables=["question"])
# initialize the GPT4All instance
llm = GPT4All(model=local_path, callback_manager=callback_manager, verbose=True)
# link the language model with our prompt template
llm_chain = LLMChain(prompt=prompt, llm=llm)

# Hardcoded question
question = "What Formula 1 pilot won the championship in the year Leonardo di Caprio was born?"

# User imput question...
# question = input("Enter your question: ")

#Run the query and get the results
llm_chain.run(question)
```

Remember to verify your virtual environment is still activated and run the command:

```
python3 my_langchain.py
```

You may get a different results from mine. What is amazing is that you can see the entire reasoning followed by GPT4All trying to get an answer for you. Adjusting the question may give you better results too.

Langchain with Prompt Template on GPT4All

Answering Question About your Documents Using LangChain and GPT4All

Here we start the amazing part, because we are going to talk to our documents using GPT4All as a chatbot who replies to our questions.

The sequence of steps, referring to Workflow of the QnA with GPT4All, is to load our pdf files, make them into chunks. After that we will need a Vector Store for our embeddings. We need to feed our chunked documents in a vector store for information retrieval and then we will embed them together with the similarity search on this database as a context for our LLM query.

For this purposes we are going to use FAISS directly from Langchain library. FAISS is an open-source library from Facebook AI Research, designed to quickly find similar items in big collections of high-dimensional data. It offers indexing and searching methods to make it easier and faster to spot the most similar items within a dataset. It is particularly convenient for us because it simplifies information retrieval and allow us to save locally the created database: this means that after the first creation it will be loaded very fast for any further usage.

Creation of the vector index db

Create a new file and call it my_knowledge_qna.py

```
from langchain import PromptTemplate, LLMChain
from langchain.llms import GPT4All
from langchain.callbacks.base import CallbackManager
from langchain.callbacks.streaming_stdout import StreamingStdOutCallbackHandler
# function for loading only TXT files
from langchain.document_loaders import TextLoader
# text splitter for create chunks
from langchain.text_splitter import RecursiveCharacterTextSplitter
# to be able to load the pdf files
from langchain.document_loaders import UnstructuredPDFLoader
from langchain.document_loaders import PyPDFLoader
from langchain.document_loaders import DirectoryLoader
# Vector Store Index to create our database about our knowledge
from langchain.indexes import VectorstoreIndexCreator
```

```
# LLamaCpp embeddings from the Alpaca model
from langchain.embeddings import LlamaCppEmbeddings
# FAISS  library for similaarity search
from langchain.vectorstores.faiss import FAISS
import os  #for interaaction with the files
import datetime
```

The first libraries are the same we used before: in addition we are using Langchain for the vector store index creation, the LlamaCppEmbeddings to interact with our Alpaca model (quantized to 4-bit and compiled with the cpp library) and the PDF loader.

Let's also load our LLMs with their own paths: one for the embeddings and one for the text generation.

```
# assign the path for the 2 models GPT4All and Alpaca for the embeddings
gpt4all_path = './models/gpt4all-converted.bin'
llama_path = './models/ggml-model-q4_0.bin'
# Calback manager for handling the calls with  the model
callback_manager = CallbackManager([StreamingStdOutCallbackHandler()])

# create the embedding object
embeddings = LlamaCppEmbeddings(model_path=llama_path)
# create the GPT4All llm object
llm = GPT4All(model=gpt4all_path, callback_manager=callback_manager, verbose=True)
```

For test let's see if we managed to read all the pfd files: the first step is to declare 3 functions to be used on each single document. The first is to split the extracted text in chunks, the second is to create the vector index with the metadata (like page numbers etc…) and the last one is for testing the similarity search (I will explain better later).

```
# Split text
def split_chunks(sources):
    chunks = []
    splitter = RecursiveCharacterTextSplitter(chunk_size=256, chunk_overlap=32)
    for chunk in splitter.split_documents(sources):
        chunks.append(chunk)
    return chunks


def create_index(chunks):
    texts = [doc.page_content for doc in chunks]
    metadatas = [doc.metadata for doc in chunks]

    search_index = FAISS.from_texts(texts, embeddings, metadatas=metadatas)

    return search_index


def similarity_search(query, index):
    # k is the number of similarity searched that matches the query
    # default is 4
    matched_docs = index.similarity_search(query, k=3)
    sources = []
    for doc in matched_docs:
        sources.append(
            {
                "page_content": doc.page_content,
                "metadata": doc.metadata,
            }
        )

    return matched_docs, sources
```

Now we can test the index generation for the documents in the docs directory: we need to put there all our pdfs. Langchain has also a method for loading the entire folder, regardless of the file type: since it is complicated the post process, I will cover it in the next article about LaMini models.

my docs directory contains 4 pdf files

We will apply our functions to the first document in the list

```
# get the list of pdf files from the docs directory into a list  format
pdf_folder_path = './docs'
doc_list = [s for s in os.listdir(pdf_folder_path) if s.endswith('.pdf')]
num_of_docs = len(doc_list)
# create a loader for the PDFs from the path
loader = PyPDFLoader(os.path.join(pdf_folder_path, doc_list[0]))
# load the documents with Langchain
docs = loader.load()
# Split in chunks
chunks = split_chunks(docs)
# create the db vector index
db0 = create_index(chunks)
```

In the first lines we use os library to get the list of pdf files inside the docs directory. We then load the first document (doc_list[0]) from the docs folder with Langchain, split in chunks and then we create the vector database with the LLama embeddings.

As you saw we are using the pyPDF method. This one is a bit longer to use, since you have to load the files one by one, but loading PDF using pypdf into array of documents allows you to have an array where each document contains the page content and metadata with page number. This is really convenient when you want to know the sources of the context we will give to GPT4All with our query. Here the example from the readthedocs:

Screenshot from Langchain documentation

We can run the python file with the command from terminal:

```
python3 my_knowledge_qna.py
```

After the loading of the model for embeddings you will see the tokens at work for the indexing: don't freak out since it will take time, specially if you run only on CPU, like me (it took 8 minutes).

Completion of the first vector db

As I was explaining the pyPDF method is slower but gives us additional data for the similarity search. To iterate through all our files we will use a convenient method from FAISS that allows us to MERGE different databases together. What we do now is that we use the code above to generate the first db (we will call it db0) and the with a for loop we create the index of the next file in the list and merge it immediately with db0.

Here is the code: note that I added some logs to give you the status of the progress using datetime.datetime.now() and printing the delta of end time and start time to calculate how long the operation took (you can remove it if you don't like it).

The merge instructions is like this

```
# merge dbi with the existing db0
db0.merge_from(dbi)
```

One of the last instructions is for saving our database locally: the entire generation can take even hours (depends on how many documents you have) so it is really good that we have to do it only once!

```
# Save the databasae locally
db0.save_local("my_faiss_index")
```

Here the entire code. We will comment many part of it when we interact with GPT4All loading the index directly from our folder.

```
# get the list of pdf files from the docs directory into a list  format
pdf_folder_path = './docs'
doc_list = [s for s in os.listdir(pdf_folder_path) if s.endswith('.pdf')]
num_of_docs = len(doc_list)
# create a loader for the PDFs from the path
general_start = datetime.datetime.now() #not used now but useful
print("starting the loop...")
loop_start = datetime.datetime.now() #not used now but useful
print("generating fist vector database and then iterate with .merge_from")
loader = PyPDFLoader(os.path.join(pdf_folder_path, doc_list[0]))
docs = loader.load()
chunks = split_chunks(docs)
db0 = create_index(chunks)
print("Main Vector database created. Start iteration and merging...")
for i in range(1,num_of_docs):
    print(doc_list[i])
    print(f"loop position {i}")
    loader = PyPDFLoader(os.path.join(pdf_folder_path, doc_list[i]))
    start = datetime.datetime.now() #not used now but useful
    docs = loader.load()
    chunks = split_chunks(docs)
    dbi = create_index(chunks)
    print("start merging with db0...")
    db0.merge_from(dbi)
    end = datetime.datetime.now() #not used now but useful
    elapsed = end - start #not used now but useful
    #total time
    print(f"completed in {elapsed}")
    print("------------------------------------")
loop_end = datetime.datetime.now() #not used now but useful
loop_elapsed = loop_end - loop_start #not used now but useful
print(f"All documents processed in {loop_elapsed}")
print(f"the daatabase is done with {num_of_docs} subset of db index")
print("------------------------------------")
print(f"Merging completed")
print("------------------------------------")
print("Saving Merged Database Locally")
# Save the databasae locally
db0.save_local("my_faiss_index")
```

```
print("-----------------------------------")
print("merged database saved as my_faiss_index")
general_end = datetime.datetime.now() #not used now but useful
general_elapsed = general_end - general_start #not used now but useful
print(f"All indexing completed in {general_elapsed}")
print("-----------------------------------")
```

Running the python file took 22 minutes

Ask questions to GPT4All on your documents

Now we are here. We have our index, we can load it and with a Prompt Template we can ask GPT4All to answer our questions. We start with an hard-coded question and then we will loop through our input questions.

Put the following code inside a python file db_loading.py and run it with the command from terminal python3 db_loading.py

```
from langchain import PromptTemplate, LLMChain
from langchain.llms import GPT4All
from langchain.callbacks.base import CallbackManager
from langchain.callbacks.streaming_stdout import StreamingStdOutCallbackHandler
# function for loading only TXT files
from langchain.document_loaders import TextLoader
# text splitter for create chunks
from langchain.text_splitter import RecursiveCharacterTextSplitter
# to be able to load the pdf files
from langchain.document_loaders import UnstructuredPDFLoader
from langchain.document_loaders import PyPDFLoader
from langchain.document_loaders import DirectoryLoader
# Vector Store Index to create our database about our knowledge
from langchain.indexes import VectorstoreIndexCreator
# LLamaCpp embeddings from the Alpaca model
from langchain.embeddings import LlamaCppEmbeddings
# FAISS  library for similaarity search
from langchain.vectorstores.faiss import FAISS
import os  #for interaaction with the files
import datetime

# TEST FOR SIMILARITY SEARCH

# assign the path for the 2 models GPT4All and Alpaca for the embeddings
gpt4all_path = './models/gpt4all-converted.bin'
llama_path = './models/ggml-model-q4_0.bin'
# Calback manager for handling the calls with  the model
callback_manager = CallbackManager([StreamingStdOutCallbackHandler()])

# create the embedding object
embeddings = LlamaCppEmbeddings(model_path=llama_path)
# create the GPT4All llm object
llm = GPT4All(model=gpt4all_path, callback_manager=callback_manager, verbose=True)

# Split text
def split_chunks(sources):
    chunks = []
    splitter = RecursiveCharacterTextSplitter(chunk_size=256, chunk_overlap=32)
    for chunk in splitter.split_documents(sources):
        chunks.append(chunk)
    return chunks


def create_index(chunks):
    texts = [doc.page_content for doc in chunks]
    metadatas = [doc.metadata for doc in chunks]

    search_index = FAISS.from_texts(texts, embeddings, metadatas=metadatas)

    return search_index


def similarity_search(query, index):
    # k is the number of similarity searched that matches the query
    # default is 4
    matched_docs = index.similarity_search(query, k=3)
    sources = []
    for doc in matched_docs:
        sources.append(
            {
                "page_content": doc.page_content,
                "metadata": doc.metadata,
            }
        )
```

```
    return matched_docs, sources

# Load our local index vector db
index = FAISS.load_local("my_faiss_index", embeddings)
# Hardcoded question
query = "What is a PLC and what is the difference with a PC"
docs = index.similarity_search(query)
# Get the matches best 3 results - defined in the function k=3
print(f"The question is: {query}")
print("Here the result of the semantic search on the index, without GPT4All..")
print(docs[0])
```

The printed text is the list of the 3 sources that best matches with the query, giving us also the document name and the page number.

Results of the semantic search running the file db_loading.py

Now we can use the similarity search as the context for our query using the prompt template. After the 3 functions just replace all the code with the following:

```
# Load our local index vector db
index = FAISS.load_local("my_faiss_index", embeddings)

# create the prompt template
template = """
Please use the following context to answer questions.
Context: {context}
---
Question: {question}
Answer: Let's think step by step."""

# Hardcoded question
question = "What is a PLC and what is the difference with a PC"
matched_docs, sources = similarity_search(question, index)
# Creating the context
context = "\n".join([doc.page_content for doc in matched_docs])
# instantiating the prompt template and the GPT4All chain
prompt = PromptTemplate(template=template, input_variables=["context", "question"]).partial(context=context)
llm_chain = LLMChain(prompt=prompt, llm=llm)
# Print the result
print(llm_chain.run(question))
```

After running you will get a result like this (but may vary). Amazing no!?!?

```
Please use the following context to answer questions.
Context: 1.What is a PLC
2.Where and Why it is used
3.How a PLC is different from a PC
PLC is especially important in industries where safety and reliability are
critical, such as manufacturing plants, chemical plants, and power plants.
How a PLC is different from a PC
Because a PLC is a specialized computer used in industrial and
manufacturing applications to control machinery and processes.,the
hardware components of a typical PLC must be able to interact with
industrial device. So a typical PLC hardware include:
---
Question: What is a PLC and what is the difference with a PC
Answer: Let's think step by step. 1) A Programmable Logic Controller (PLC),
also called Industrial Control System or ICS, refers to an industrial computer
that controls various automated processes such as manufacturing
machines/assembly lines etcetera through sensors and actuators connected
with it via inputs & outputs. It is a form of digital computers which has
the ability for multiple instruction execution (MIE), built-in memory
registers used by software routines, Input Output interface cards(IOC)
to communicate with other devices electronically/digitally over networks
or buses etcetera
2). A Programmable Logic Controller is widely utilized in industrial
automation as it has the ability for more than one instruction execution.
It can perform tasks automatically and programmed instructions, which allows
it to carry out complex operations that are beyond a
Personal Computer (PC) capacity. So an ICS/PLC contains built-in memory
registers used by software routines or firmware codes etcetera but
PC doesn't contain them so they need external interfaces such as
hard disks drives(HDD), USB ports, serial and parallel
communication protocols to store data for further analysis or
report generation.
```

If you want an user-input question replace the line

```
question = "What is a PLC and what is the difference with a PC"
```

with something like this:

```
question = input("Your question: ")
```

Conclusions (for now…)

It is time for you to experiment. Ask different questions on all the topics related to your documents, and see the results. There is a big room for improvement, certainly on the prompt and template: you can have a look here for some inspirations. But Langchain documentation is really amazing (I could follow it!!).

You can follow the code from the article or check it on my github repo.

I will try in the next articles to explore with you very small models that perform very well on question and answers.

Troubleshooting Section

Update and bug fixes — 2023.05.23

Some readers faces an issue with langchain.callbacks

```
ImportError: cannot import name 'CallbackManager' from 'langchain.callbacks.base'
on the line importing CallbackManager in my_langchain.py.
```

Michal Founě solved it follwing the issue as described in https://github.com/hwchase17/chat-langchain/issues/70 with

pip install langchain==0.0.142

On the GitHub repo there is already an issue solved related to 'GPT4All' object has no attribute '_ctx'. Fixed with versions during pip install like this:

```
pip install pygpt4all==1.0.1
pip install pygptj==1.0.10
pip install pyllamacpp==1.0.6
```

Another quite common issue is related to readers using Mac with M1 chip. The arm64 architecture is a little reluctant to work. As suggested to Emile Pretorius and to Yosef Agung Wicaksono you can try to fix it with the guidelines in this document.

Bruce Wen suggested to avoid all the problems mentioned above giving already all the versions in the code: all the pip instructions are now updated.

BEpshtein suggested to update the article and the Github repo: keep reporting the issues and I will try to reply.

geert van kempen got the following issue:

```
File "/Users/XXXX/GPT4ALL_gvk/.venv/lib/python3.10/site-packages/pyllamacpp/model.py", line 402, in __del__
if self._ctx:
AttributeError: 'GPT4All' object has no attribute '_ctx'
```

He was using pygpt4all version 1.1.0, and pygptj 2.0.3 (and he was using version 1.0.6 of pyllamaccp) and he solved it with the GitHub solved Issue (see above).

Norman Procope and Kon16ov got problems loading the pdfs from my GitHub repo. It was solved downloading again the pdf or checking if UTF-8 format was used or not: fixed with decoding text = text.decode()

Alain Uro and other users got an error on the pygtp4all callbacks. As greatly explained and solved by Rajneesh Aggarwal this happens because the pygpt4all PyPI package will no longer by actively maintained and the bindings may diverge from the GPT4All model backends. He solved it installing instead of pygtp4all pip install gpt4all

The code must be changed as well as follows

```
import gpt4all
model_path = '.\models'
model = gpt4all.GPT4All(model_name='gpt4all-converted.bin', model_path=model_path, model_type='llama', allow_download=True)
model.generate("Once upon a time, ", streaming=True)
```

Shamik Dhar also had an issue with the new_text_callback.

```
def new_text_callback(text):
print(text, end="")
model = GPT4All('./models/gpt4all-converted.bin')
model.generate("Once upon a time, ", n_predict=55, new_text_callback=new_text_callback)
```

```
I am getting an error here that there is no argument in
the generate module of name new_text_callback
```

It was solved by Oscar Jeong changing the code generate() to cpp_generate() as follows:

```
# Just change generate() to cpp_generate() then it works.
def new_text_callback(text):
print(text, end="")
model = GPT4All('./models/gpt4all-converted.bin')
model.cpp_generate("Once upon a time, ", n_predict=55, new_text_callback=new_text_callback)
```

If this story provided value and you wish to show a little support, you could:

1. Clap 50 times for this story (this really, really helps me out)
2. Sign up for a Medium membership using my link — ($5/month to read unlimited Medium stories)
3. Follow me on Medium
4. Read my latest articles (https://medium.com/@fabio.matricardi)

Do you want to learn how to summarize long texts, papers or video transcript on your own, using the new LaMini-LM models, running only on CPU? Read this.

LaMini is here: a little giant LLM on your CPU

Upload documents or Youtube videos and interact with them: learn how to summarize long texts, papers or video…

artificialcorner.com

Are looking for a fast solution to build a ChatBot that replies to the content of your website? Have a look at this: it is simple, smart and free!

Create a Chatbot for your website in 5 minutes

Ask questions directly on your webpage with ora.ai and ChatGPT for FREE!

artificialcorner.com

Artificial Intelligence
Python
Gpt4all
Local Gpt
Hugging Face

--

--

40

Fabio Matricardi
Artificial Corner
Follow

Written by Fabio Matricardi

977 Followers
·Writer for

Artificial Corner

passionate educator, curious industrial automation engineer, AI and ML enthusiast. Learning Leadership and how to build my own AI for a smooth process control.

Follow

More from Fabio Matricardi and Artificial Corner

Fabio Matricardi

Fabio Matricardi

in

Artificial Corner

Answering Question About your Documents Using LangChain (and NOT OpenAI)

How to use Hugging Face LLM (open source LLM) to talk to your documents, pdfs and also articles from webpages.

·14 min read·May 1

--

11

The PyCoach

The PyCoach

in

Artificial Corner

Bye-bye ChatGPT: AI Tools As Good As ChatGPT (But Few People Are Using Them)

Go beyond ChatGPT with these powerful AI tools.

·5 min read·Apr 26

--

126

Diana Dovgopol

Diana Dovgopol

in

Artificial Corner

I Used ChatGPT (Every Day) for 5 Months. Here Are Some Hidden Gems That Will Change Your Life

Transform your life with these ChatGPT's hidden gems.

·7 min read·May 3

--

80

Fabio Matricardi

Fabio Matricardi

in

Artificial Corner

LaMini is here: a little giant LLM on your CPU

Upload documents or Youtube videos and interact with them: learn how to summarize long texts, papers or video transcript on your own.

·24 min read·6 days ago

--

3

See all from Fabio Matricardi
See all from Artificial Corner

Recommended from Medium

Wei-Meng Lee

Wei-Meng Lee

in

Level Up Coding

Training Your Own LLM using privateGPT

Learn how to train your own language model without exposing your private data to the provider

·8 min read·May 19

--

7

Rui Alves

Rui Alves

in

The Generator

Snapchat Influencer Uses GPT-4 to Become a High-End AI Girlfriend

This is how she made $71,610 in a single month

·4 min read·May 12

--

46

Lists

What is ChatGPT?

9 stories·57 saves

Staff Picks

323 stories·82 saves
The PyCoach

The PyCoach

in

Artificial Corner

You’re Using Midjourney Wrong! Here’s How to Create Better Images than 99% of Midjourney Users

Generate amazing images by learning how to create better Midjourney prompts.

·7 min read·Apr 12

--

25

Guodong (Troy) Zhao

Guodong (Troy) Zhao

in

Bootcamp

A comprehensive and hands-on guide to autonomous agents with GPT

Understand autonomous agents with LLM, from conceptual guide to hands-on tutorial

·15 min read·May 2

--

5

Gabe Araujo, M.Sc.

Gabe Araujo, M.Sc.

in

Level Up Coding

🧬 How I Used Python to Make Everyday Tasks Easier

Hey there! As a busy person with a lot on my plate, I’m always looking for ways to make my life easier.

·8 min read·May 1

--

2

Matt Chapman

Matt Chapman

in

Towards Data Science

How I Stay Up to Date With the Latest AI Trends as a Full-Time Data Scientist

No, I don’t just ask ChatGPT to tell me

·8 min read·May 1

--

21

See more recommendations