Caching Generative LLMs | Saving API Costs

BEGINNER CHATBOT DATABASE GENERATIVE AI LLMS

Introduction

Generative AI has prevailed so much that most of us will be or already have started working on applications involving Generative AI models, be it Image generators or the famous Large Language Models. Most of us work with Large Language Models, especially the closed source ones like OpenAI, where we have to pay to use the models developed by them. Now if we are careful enough, we can minimize the costs when working with these models, but somehow or the other, the prices do add up a lot. And this is what we will look into in this article, i.e., catching the responses / API calls sent to the Large Language Models. Are you excited to learn about Caching Generative LLMs?

Learning Objectives

- Understand what Caching is and how it works
- Learn how to Cache Large Language Models
- Learn different ways to Cache LLMs in LangChain
- Understand the potential benefits of Caching and how it reduces API costs

This article was published as a part of the Data Science Blogathon.

Table of contents

What is Caching? Why is it Required?

A cache is a place to store data temporarily so that it can be reused, and the process of storing this data is called caching. Here the most frequently accessed data is stored to be accessed more quickly. This has a

drastic effect on the performance of the processor. Imagine the processor performing an intensive task requiring a lot of computation time. Now imagine a situation where the processor has to perform the exact computation again. In this scenario, caching the previous result really helps. This will reduce the computation time, as the result was cached when the task was performed.

In the above type of cache, the data is stored in the processor's cache, and most of the processes come within an in-built cache memory. But these may not be sufficient for other applications. So in these cases, the cache is stored in RAM. Accessing data from RAM is much faster than from a hard disk or SSD. Caching can also save API call costs. Suppose we send a similar request to the Open AI model, We will be billed for each request sent, and the time taken to respond will be greater. But if we cache these calls, we can first search the cache to check if we have sent a similar request to the model, and if we have, then instead of calling the API, we can retrieve the data, i.e., the response from the cache.

Caching in Large Language Models

We know that closed-source models like GPT 3.5 from OpenAI and others charge the user for the API calls being made to their Generative Large Language Models. The charge or the cost associated with the API call largely depends on the number of tokens passed. The larger the number of tokens, the higher the associated cost. This must be carefully handled so you do not pay large sums.

Now, one way to solve this / reduce the costs of calling the API is to cache the prompts and their corresponding responses. When we first send a prompt to the model and get the corresponding response, we store it in the cache. Now, when another prompt is being sent, before sending it to the model, that is, before making an API call, we will check if the prompt is similar to any of the ones saved in the cache; if it is, then we will take the response from the cache instead of sending the prompt to the model(i.e., Making an API call) and then getting the response from it.

This will save costs whenever we ask for similar prompts to the model, and even the response time will be less, as we are getting it directly from the cache instead of sending a request to the model and then getting a response from it. In this article, we will see different ways to cache the responses from the model.

Caching with LangChain's InMemoryCache

Yes, you read it right. We can cache responses and calls to the model with the LangChain library. In this section, we will go through how to set up the Cache mechanism and even see the examples to ensure that our results are being Cached and the responses to similar queries are being taken from the cache. Let's get started by downloading the necessary libraries.

```
!pip install langchain openai
```

To get started, pip install the LangChain and OpenAI libraries. We will be working with OpenAI models and see how they are pricing our API calls and how we can work with cache to reduce it. Now let's get started with the code.

```
import os import openai from langchain.llms import OpenAI os.environ["OPENAI_API_KEY"] = "Your API Token" llm = OpenAI(model_name="text-davinci-002", openai_api_key=os.environ["OPENAI_API_KEY"]) llm("Who was the first person to go to Space?")
```

```
'\n\nThe first person to go to space was Yuri Gagarin, a Soviet cosmonaut, in April 1961.'
```

- Here we have set the OpenAI model to start working with. We must provide the OpenAI API key to the os.environ[] to store our API key to the OPNEAI_API_KEY environment variable.
- Then import the LangChain's LLM wrapper for OpenAI. Here the model we are working on is the "text-davinci-002" and to the OpenAI() function, we also pass the environment variable containing our API key.
- To test that the model works, we can make the API calls and query the LLM with a simple question.
- We can see the answer generated by the LLM in the above picture. This ensures that the model is up and running, and we can send requests to the model and get responses generated by it.

Caching Through LangChain

Let us now take a look at caching through LangChain.

```
import langchain from langchain.cache import InMemoryCache from langchain.callbacks import get_openai_callback langchain.llm_cache = InMemoryCache()
```

- LangChain library has an in-built function for caching called InMemoryCache. We will work with this function for caching the LLMs.
- To start caching with LangChain, we pass the InMemoryCache() function to the langchain.llm_cache
- So here, first, we are creating an LLM cache in LangChain using the langchain.llm_cache
- Then we take the InMemoryCache(a caching technique) and pass it to the langchain.llm_cache
- Now this will create an InMemoryCache for us in LangChain. We replace the InMemoryCache with the one we want to work with to use a different caching mechanism.
- We are even importing the get_openai_callback. This will give us information about the number of tokens passed to the model when an API call is made, the cost it took, the number of response tokens, and the response time.

Query the LLM

Now, we will query the LLM, then cache the response, then query the LLM to check if the caching is working and if the responses are being stored and retrieved from the cache when similar questions are asked.

```
%%time import time with get_openai_callback() as cb: start = time.time() result = llm("What is the Distance between Earth and Moon?") end = time.time() print("Time taken for the Response",end-start) print(cb) print(result)
```

Time Function

In the above code, we use the %% timeline function in colab to tell us the time the cell takes to run. We also import the time function to get the time taken to make the API call and get the response back. Here as stated before, we are working with the get_openai_callback(). We then print it after passing the query to the model. This function will print the number of tokens passed, the cost of processing the API call, and the time taken. Let's see the output below.

```
Time taken for the Response 0.8009738922119141
Tokens Used: 30
        Prompt Tokens: 9
        Completion Tokens: 21
Successful Requests: 1
Total Cost (USD): $0.0006000000000000001


The average distance between Earth and Moon is about 384,400 kilometers (238,900
miles).
CPU times: user 9.08 ms, sys: 123 µs, total: 9.2 ms
Wall time: 802 ms
```

The output shows that the time taken to process the request is 0.8 seconds. We can even see the number of tokens in the prompt query that we have sent, which is 9, and the number of tokens in the generated output, i.e., 21. We can even see the cost of processing our API call in the callbacks generated, i.e., $0.0006. The CPU time is 9 milliseconds. Now, let's try rerunning the code with the same query and see the output generated.

```
Time taken for the Response 0.00036025047302246094
Tokens Used: 0
        Prompt Tokens: 0
        Completion Tokens: 0
Successful Requests: 0
Total Cost (USD): $0.0


The average distance between Earth and Moon is about 384,400 kilometers (238,900
miles).
CPU times: user 482 µs, sys: 0 ns, total: 482 µs
Wall time: 488 µs
```

Here we see a significant difference in the time it took for the response. It is 0.0003 seconds which is 2666x faster than the first time we ran it. Even in callback output, we see the number of prompt tokens as 0, the cost is $0, and the output tokens are 0 too. Even the Successful Requests is set to 0, indicating no API call/request was sent to the model. Instead, it was fetched from the cache.

With this, we can say that LangChain had cached the prompt and the response generated by the OpenAIs Large Language Model when it was run for the same prompt last time. This is the method to cache the LLMs through LangChain's InMemoryCache() function.

Caching with SQLiteCache

Another way of caching the Prompts and the Large Language Model responses is through the SQLiteCache. Let's get started with the code for it

```
from langchain.cache import SQLiteCache langchain.llm_cache = SQLiteCache(database_path=".langchain.db")
```

Here we define the LLM Cache in LangChain in the same way as we have defined previously. But here, we giving it a different caching method. We are working on the SQLiteCache, which stores the database's Prompts and Large Language Model responses. We even provide the database path of where to store these Prompts and Responses. Here it will be the langchain.db.

So let's try testing the caching mechanism like we have tested it before. We will run a query to the OpenAI's Large Language Model two times and then check if the data is being cached by observing the output generated on the second run. The code for this will be

```
%%time import time start = time.time() result = llm("Who created the Atom Bomb?") end = time.time()
print("Time taken for the Response",end-start) print(result)
```

```
Time taken for the Response 0.7359402179718018


The atom bomb was created by Robert Oppenheimer.
CPU times: user 13 ms, sys: 4.05 ms, total: 17.1 ms
Wall time: 736 ms
```

```
%%time import time start = time.time() result = llm("Who created the Atom Bomb?") end = time.time() print("Time taken for the Response",end-start) print(result)
```

```
Time taken for the Response 0.0026454925537109375


The atom bomb was created by Robert Oppenheimer.
CPU times: user 4.13 ms, sys: 8 µs, total: 4.14 ms
Wall time: 9.15 ms
```

In the first output, when we first ran the query to the Large Language Model, it takes to send the request to the model and get the response back is 0.7 seconds. But when we try to run the same query to the Large Language Model, we see the time taken for the response is 0.002 seconds. This proves that when the query “Who created the Atom Bomb” was run for the first time, both the Prompt and the response generated by the Large Language Model were cached in the SQLiteCache database.

Then when we ran the same query for the second time, it first looked for it in the cache, and as it was available, it just took the corresponding response from the cache instead of sending a request to the OpenAI’s model and getting a response back. So this is another way of caching Large Language Models.

Benefits of Caching

Reduction in Costs

Caching significantly reduces API costs when working with Large Language Models. API costs are associated with sending a request to the model and receiving its response. So the more requests we send to the Generative Large Language Model, the greater our costs. We have seen that when we ran the same query for the second time, the response for the query was taken from the cache instead of sending a request to the model to generate a response. This greatly helps when you have an application where many a time, similar queries are sent to the Large Language Models.

Increase in Performance/ Decreases Response Time

Yes. Caching helps in performance boosts. Though not directly but indirectly. An increase in performance is when we are caching answers which took quite a time to compute by the processor, and then we have to re-calculate it again. But if we have cached it, we can directly access the answer instead of recalculating it. Thus, the processor can spend time on other activities.

When it comes to caching Large Language Models, we cache both the Prompt and the response. So when we repeat a similar query, the response is taken from the cache instead of sending a request to the model.

This will significantly reduce the response time, as it directly comes from the cache instead of sending a request to the model and receiving a response. We even checked the response speeds in our examples.

Conclusion

In this article, we have learned how caching works in LangChain. You developed an understanding of what caching is and what its purpose is. We also saw the potential benefits of working with a cache than working without one. We have looked at different ways of caching Large Language Models in LangChain(InMemoryCache and SQLiteCache). Through examples, we have discovered the benefits of using a cache, how it can decrease our application costs, and, at the same time, ensure quick responses.

Key Takeaways

Some of the key takeaways from this guide include:

- Caching is a way to store information that can then be retrieved at a later point in time
- Large Language Models can be cached, where the Prompt and the response generated are the ones that are saved in the cache memory.
- LangChain allows different caching techniques, including InMemoryCache, SQLiteCache, Redis, and many more.
- Caching Large Language Models will result in fewer API calls to the models, a reduction in API costs, and provides faster responses.

Frequently Asked Questions

Q1. What is Caching Generative LLMs?

A. Caching stores intermediate/final results so they can be later fetched instead of going through the entire process of generating the same result.

Q2. What are the benefits of Caching?

A. Improved performance and a significant drop in response time. Caching will save hours of computational time required to perform similar operations to get similar results. Another great benefit of caching is reduced costs associated with API calls. Caching a Large Language Model will let you store the responses, which can be later fetched instead of sending a request to the LLM for a similar Prompt.

Q3. Does LangChain support Caching?

A. Yes. LangChain supports the caching of Large Language Models. To get started, we can directly work with InMemoryCache() provided by LangChain, which will store the Prompts and the Responses generated by the Large Language Models.

Q4. What are some of the Caching techniques for LangChain?

A. Caching can be set in many ways to cache the models through LangChain. We have seen two such ways, one is through the in-built InMemoryCache, and the other is with the SQLiteCache method. We can even cache through the Redis database and other APIs designed especially for caching.

Q5. In what cases can caching be used?

A. It is mainly used when you expect similar queries to appear. Consider you are developing a customer service chatbot. A customer service chatbot gets a lot of similar questions, many users have similar queries when talking with customer care regarding a specific product/service. In this case, caching can be employed, resulting in quicker responses from the bot and reduced API costs.

Article Url - https://www.analyticsvidhya.com/blog/2023/08/caching-generative-llms-saving-api-costs/

Ajay Kumar Reddy