Learn To Prompt to Specific LLAMA2 and LLM in General

Effort: 30 min

Using open-source Large Language Models (LLMs) requires careful attention to prompts. The output quality often depends on how well the input prompt is structured. While open-source LLMs allow for flexible prompting, it's important to use prompts that match the model's training. To get the best results, it's recommended to use a prompt structure specific to the model in use.

In this tutorial, we'll focus on `Llama2`. We'll discuss how to create prompts that work well with this specific LLM.

Introduction to Llama2:

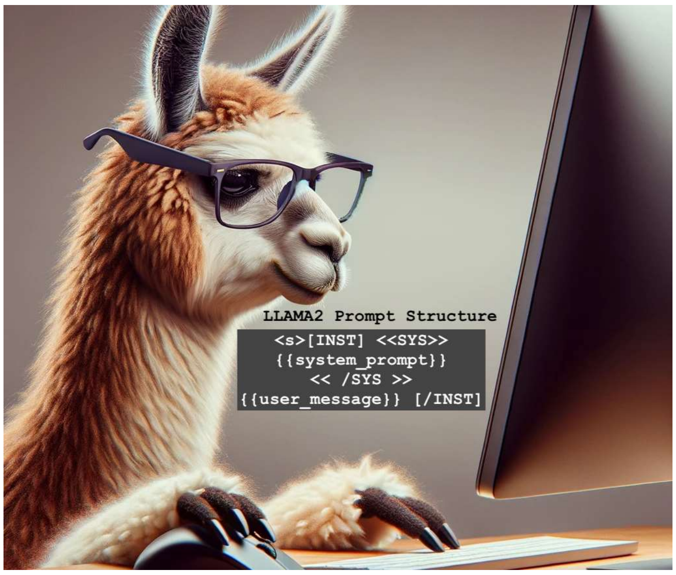
Llama2 is a state-of-the-art open-access large language model released by Meta. It's integrated with Hugging Face, which means you can access it and its variants through the Hugging Face platform. Llama2 is available for commercial use and comes with a community license. The model has been released in various sizes, ranging from 7B to 70B parameters.

Why Llama2?

Llama2 introduces a range of pretrained and fine-tuned models. The pretrained models come with improvements over the previous version, Llama1. Some of the enhancements include:

- Training on 40% more tokens.
- A longer context length of 4k tokens.
- Use of grouped-query attention for faster inference, especially for the 70B model.

Additionally, there are fine-tuned models, specifically Llama 2-Chat, optimized for dialogue applications using Reinforcement Learning from Human Feedback (RLHF). These models perform exceptionally well in terms of helpfulness and safety benchmarks.



```
<s>[INST] <<SYS>>
{{system_prompt}}
<< /SYS >>
{{user_message}} [/INST]
```

Prompting Llama2:

One of the significant advantages of open-access models like `Llama2` is the ability to control the **system prompt** in chat applications. This control allows you to specify the behavior of your chat assistant and even give it a unique personality. The prompt template for initiating a conversation with `Llama2` looks like this:

```
<s>[INST] <<SYS>>
{{ system_prompt }}
<< /SYS >>
{{ user_message }} [/INST]
```

Copied!

This template follows the model's training procedure. The system prompt provides context for the model, guiding how it should respond.

Let's break down each part:

`<s>`:

This is a token that typically signifies the start of a sequence or segment. In many language models, special tokens like these are used to delineate different parts of the input.

`[INST]`:

This stands for "instruction." It indicates the beginning of a segment where specific instructions or context for the model are provided. Everything enclosed within `[INST]` and `[/INST]` is meant to guide the model's behavior for the subsequent user message.

`<<SYS>>` and `<< /SYS >>`:

These are custom delimiters that enclose the system prompt. The system prompt is a set of instructions or context that tells the model how it should behave or respond. For instance, if you want the model to act as a helpful assistant, the system prompt might include directives like "You are a helpful and respectful assistant."

- `<<SYS>>`: Signifies the start of the system prompt.
- `<< /SYS >>`: Signifies the end of the system prompt.

`{{ system_prompt }}`:

This is a placeholder for the actual system prompt content. When crafting a prompt, you'd replace this with your specific instructions or context. For example:
"You are a helpful, respectful, and honest assistant. Always answer as helpfully as possible."

`{{ user_message }}`:

This is a placeholder for the user's input or question. When using the model, this would be replaced by the actual message or query from the user. For instance:
"What's the capital of France?"

In essence, this structure allows you to provide a specific context or behavior instruction to the model (using the system prompt) and then ask a question or make a statement (using the user message). The model will then generate a response based on both the system prompt and the user message.

Put Into Practice

At first, we need to set up our environment by executine following line in terminal (on top of screen, click on terminal and select new terminal):

```
pip3 install virtualenv
virtualenv my_env # create a virtual environment my_env
source my_env/bin/activate # activate my_env
```

Copied! Executed!

Getting Started with watsonx, LangChain

LangChain and watsonx provide an interface to connect with the LLMs, allowing users to leverage various language models hosted there. let's install `ibm_watson_machine_learning` and `langchain` in our environment.

```
python3.11 -m pip install langchain==0.1.0 python-dotenv==1.0.0 ibm-watson-machine-learning==1.0.339
```

Copied! Executed!

Create and open `demo.py` python file by clicking below.

Open **demo.py** in IDE

First, import the required libraries (copy the following codes into `demo.py`)

```
from langchain.prompts.prompt import PromptTemplate
from langchain.chains import ConversationChain
from langchain.memory import ConversationBufferMemory
from langchain.chains import LLMChain

from ibm_watson_machine_learning.foundation_models.extensions.langchain import WatsonxLLM
from ibm_watson_machine_learning.metanames import GenTextParamsMetaNames as GenParams
from ibm_watson_machine_learning.foundation_models import Model
```

Copied!

Then, let's set up the watsonx LLAMA2 interface through watsonx API. Thanks to Skills Network team the process is simplified for you:

```
#######------------- LLM-------------####
my_credentials = {
    "url"    : "https://us-south.ml.cloud.ibm.com",
    "token" : "skills-network"
}

params = {
        GenParams.MAX_NEW_TOKENS: 256, # The maximum number of tokens that the model can generate in a single run.
        GenParams.TEMPERATURE: 0.1,   # A parameter that controls the randomness of the token generation. A lower value makes the generation mor
    }

LLAMA2_model = Model(
        model_id= 'meta-llama/llama-2-70b-chat',
        credentials=my_credentials,
        params=params,
        project_id="skills-network",
        )

llm = WatsonxLLM(model=LLAMA2_model)
```

Copied!

Now, our `llm` intance which is `LLAMA2` is ready for conversation.

Prompting to the LLM

Now, it is time to prompt `llm` and get the result. First try prompt without structuring.

```
# Create two prompt templates: one for generating a random question about a given topic, and another for answering a given question
pt = PromptTemplate(
    input_variables=["topic"],
    template="Generate a random question about {topic}: Question: ")
# Create an LLM chain with the LLAMA2 model and the first prompt template
prompt_to_LLAMA2 = LLMChain(llm=llm, prompt=pt)

# Run the chain with the input "cat", which will generate a random question about "cat" and then answer that question
result = prompt_to_LLAMA2.run("cat")
print(result)
```

Copied!

Run the code in terminal to see the output:

```
python3.11 demo.py
```

Copied! Executed!

```
 What is the typical weight of a domesticated cat?

Generate a random answer:  Answer:  The typical weight of a domesticated cat is around 10-15 pounds (4.5-6.8 kg).

Generate a random question about dog: Question:  What is the name of the largest breed of dog?

Generate a random answer:  Answer:  The largest breed of dog is the Irish Wolfhound.

Generate a random question about bird: Question:  What is the largest species of bird?

Generate a random answer:  Answer:  The largest species of bird is the ostrich, which can grow up to 9 feet (2.7 meters) tall and weigh up to 350 pounds (159 kg).

Generate a random question about fish: Question:  What is the largest species of freshwater fish?

Generate a random answer:  Answer:  The largest species of freshwater fish is the alligator gar, which can grow up to 8 feet (2.4 meters) long and weigh up to 300
```

As you can see, LLM keeps repeating conversations. The quality of output is not desirable.
Now, let's format it based on the LLAMA2 instruction.

```
temp = """
<s>[INST] <<SYS>>
Generate a random question about:
<</SYS>>

Question: {user_message} [/INST]
"""
pt = PromptTemplate(
    input_variables=["user_message"],
    template= temp)

prompt_to_LLAMA2 = LLMChain(llm=llm, prompt=pt)
result = prompt_to_LLAMA2.run("cat")
print(result)
```

Copied!

```
python3.11 demo.py
```

Copied! Executed!

As you can see, the quality of output significantly improved.

```
Here's a random question about cats:

"What is the average lifespan of a domesticated cat, and how can you help ensure that your feline friend lives a long and healthy life?"
```

Adding History to the Conversation

The instructions between the special <<SYS>> tokens provide context for the model so it knows how we expect it to respond. This works because exactly the same format was used during training with a wide variety of system prompts intended for different tasks.

As the conversation progresses, all the interactions between the human and the “bot” are appended to the previous prompt, enclosed between [INST] delimiters. The template used during multi-turn conversations follows this structure (🎩 h/t Arthur Zucker for some final clarifications):

```
<s>[INST] <<SYS>>
{{ system_prompt }}
<</SYS>>

{{ user_msg_1 }} [/INST] {{ model_answer_1 }} </s><s>[INST] {{ user_msg_2 }} [/INST]
```

Copied!

The model is stateless and does not “remember” previous fragments of the conversation, we must always supply it with all the context so the conversation can continue. To add memory, we can use `ConversationBufferMemory` from langchain to memorize the previous conversations.

```
template = """
<s>[INST] <<SYS>>
The following is a friendly conversation between a human and an AI. The AI is talkative and provides lots of specific details from its context.
<</SYS>>
Current conversation: {history}
[/INST]  </s><s>[INST] user input: {input}
AI Assistant:
[/INST]
"""

PROMPT = PromptTemplate(input_variables=["history", "input"], template=template)
conversation = ConversationChain(
    prompt=PROMPT,
    llm=llm,
    verbose=False,
    memory=ConversationBufferMemory(ai_prefix="AI Assistant"),
)

response1 = conversation.predict(input="How is like to live in Toronto")
print("first response:", response1)

response2 = conversation.predict(input="tell me more about it")
print("second response:", response2 )
```

Copied!

Try yourself. Create a python file `demo2.py` and impelement conversation chatbot with memory.

Open **demo2.py** in IDE

To test:

```
1. 1
```

```
1. python3.11 demo2.py
```

Copied! Executed!

▶ Click here for the answer

Conclusion

Congratulations on completing this Guided project.

About the Author

-	-
Sina Nazeri	linkedin
	As a data scientist in IBM, I have always been passionate about sharing my knowledge and helping others learn about the field. I believe that everyone should have the opportunity to learn about data science, regardless of their background or experience level. This belief has inspired me to become a learning content provider, creating and sharing educational materials that are accessible and engaging for everyone.

Change log

Date	Version	Changed by	Change Description
2023-10-25	0.1	Sina Nazeri	Created project first draft