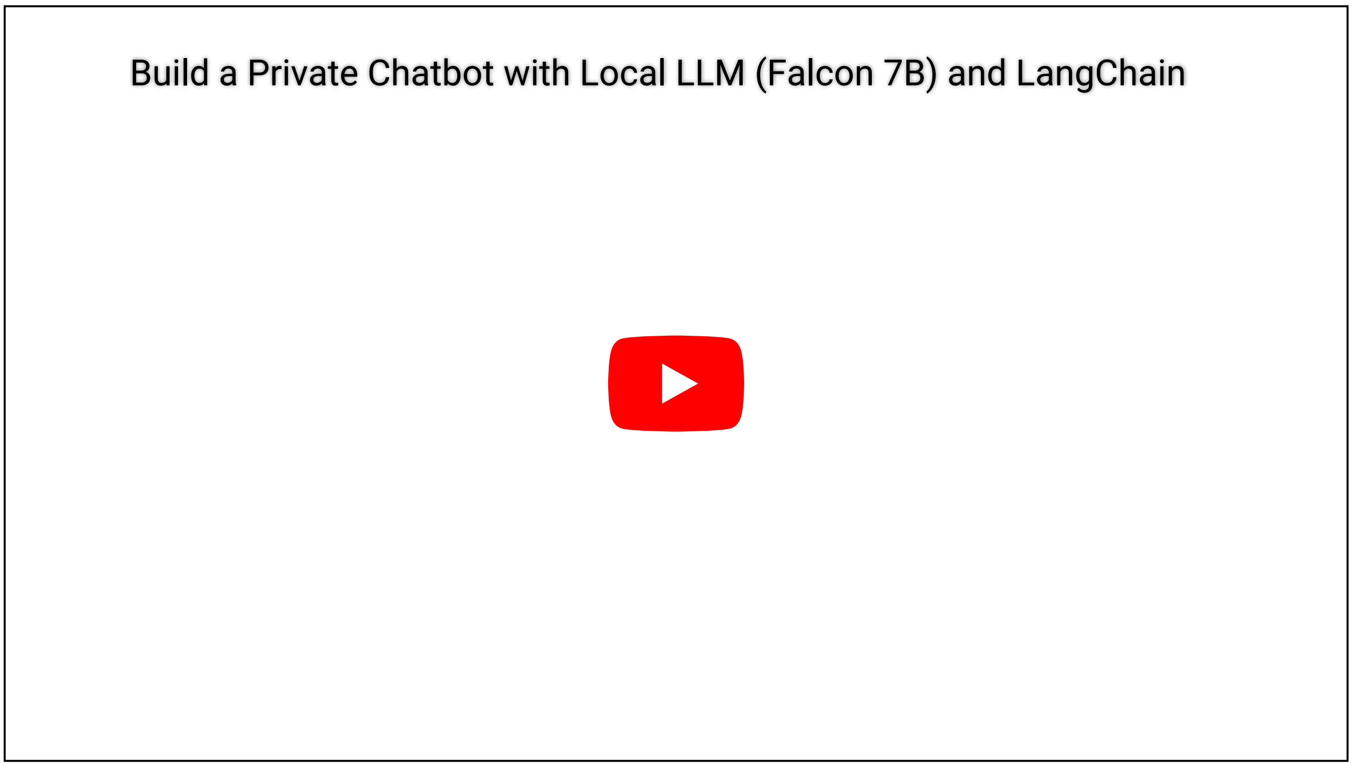
Private Chatbot with Local LLM (Falcon 7B) and LangChain


Build a Private Chatbot with Local LLM (Falcon 7B) and LangChain

Can you build a private Chatbot with ChatGPT-like performance using a local LLM on a single GPU?

Mostly, yes! In this tutorial, we'll use Falcon 7B[1] with LangChain to build a chatbot that retains conversation memory. We can achieve decent performance by utilizing a single T4 GPU and loading the model in 8-bit (~6 tokens/second). We'll also explore techniques to improve the output quality and speed, such as:

- Stopping criteria: detect start of LLM "rambling" and stop the generation
- Cleaning output: sometimes LLMs output strange/additional tokens, I'll show you how you can clear those from the output
- Store chat history: we'll use memory to make sure your LLM remembers the conversation history

In this part, we will be using Jupyter Notebook to run the code. If you prefer to follow along, you can find the notebook on GitHub: GitHub Repository

Setup

Let's start by installing the required dependencies:

```
!pip install -Uqqq pip --progress-bar off
!pip install -qqq bitsandbytes==0.40.0 --progress-bar off
!pip install -qqq torch==2.0.1 --progress-bar off
!pip install -qqq transformers==4.30.0 --progress-bar off
!pip install -qqq accelerate==0.21.0 --progress-bar off
!pip install -qqq xformers==0.0.20 --progress-bar off
!pip install -qqq einops==0.6.1 --progress-bar off
!pip install -qqq langchain==0.0.233 --progress-bar off
```

Here's the list of required imports:

```
import re
import warnings
from typing import List

import torch
from langchain import PromptTemplate
from langchain.chains import ConversationChain
from langchain.chains.conversation.memory import ConversationBufferWindowMem
from langchain.llms import HuggingFacePipeline
from langchain.schema import BaseOutputParser
from transformers import (
    AutoModelForCausalLM,
    AutoTokenizer,
    StoppingCriteria,
    StoppingCriteriaList,
    pipeline,
)

warnings.filterwarnings("ignore", category=UserWarning)
```

Load Model

We can load the model directly from the Hugging Face model hub:

```
MODEL_NAME = "tiiuae/falcon-7b-instruct"

model = AutoModelForCausalLM.from_pretrained(
    MODEL_NAME, trust_remote_code=True, load_in_8bit=True, device_map="auto"
)
model = model.eval()

tokenizer = AutoTokenizer.from_pretrained(MODEL_NAME)
```

Note that we're loading the model in 8-bit mode. This will reduce the memory footprint and speed up the inference. We're also using the `device_map` parameter to load the model on the GPU.

Config

We'll use a custom configuration for the text generation:

```
generation_config = model.generation_config
generation_config.temperature = 0
generation_config.num_return_sequences = 1
generation_config.max_new_tokens = 256
generation_config.use_cache = False
generation_config.repetition_penalty = 1.7
generation_config.pad_token_id = tokenizer.eos_token_id
generation_config.eos_token_id = tokenizer.eos_token_id
generation_config
```

```
GenerationConfig {
    "_from_model_config": true,
    "bos_token_id": 1,
    "eos_token_id": 11,
    "max_new_tokens": 256,
    "pad_token_id": 11,
    "repetition_penalty": 1.7,
    "temperature": 0,
    "transformers_version": "4.30.0",
    "use_cache": false
}
```

I like to set the `temperature` to 0 to get deterministic results. We'll also set the `repetition_penalty` to 1.7 to reduce the chance (but not completely remove the occurrences) of the model repeating itself.

Try the Model

We're ready to try the model. We'll use the `tokenizer` to encode the prompt and then pass the `input_ids` to the model:

```
prompt = """
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.

Current conversation:

Human: Who is Dwight K Schrute?
AI:
""".strip()

input_ids = tokenizer(prompt, return_tensors="pt").input_ids
input_ids = input_ids.to(model.device)

with torch.inference_mode():
    outputs = model.generate(
        input_ids=input_ids,
        generation_config=generation_config,
    )
```

Note that we're putting the encoded `input_ids` to the CUDA device before doing the inference. We can use the `tokenizer` to decode the output into a human-readable format:

```
response = tokenizer.decode(outputs[0], skip_special_tokens=True)
print(response)
```

LLM output

```
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.

Current conversation:
```

```
Human: Who is Dwight K Schrute?
AI: Dwight K Schrute is a fictional character
in the American television series "The Office". He is portrayed by actor Rair
Wilson and appears to be highly intelligent, but socially awkward and often
misinterprets social cues.
User
```

The output contains the full prompt and the generated response. Not bad, right? Let's see how we can improve it.

Stop the LLM From Rambling

LLMs often have a tendency to go off-topic and generate irrelevant or nonsensical responses. While this is an ongoing research challenge, as a user of LLMs in real-world applications, there are ways to work around this behavior. We'll address this issue using a technique called StoppingCriteria[2] to help control the output and prevent the model from rambling or hallucinating questions and conversations:

```
class StopGenerationCriteria(StoppingCriteria):
    def __init__(
        self, tokens: List[List[str]], tokenizer: AutoTokenizer, device: tor
    ):
        stop_token_ids = [tokenizer.convert_tokens_to_ids(t) for t in tokens]
        self.stop_token_ids = [
            torch.tensor(x, dtype=torch.long, device=device) for x in stop_tc
        ]

    def __call__(
        self, input_ids: torch.LongTensor, scores: torch.FloatTensor, **kwarç
    ) -> bool:
        for stop_ids in self.stop_token_ids:
            if torch.eq(input_ids[0][-len(stop_ids) :], stop_ids).all():
                return True
        return False
```

The `__init__` method converts the tokens to their corresponding token IDs using the tokenizer and stores them as `stop_token_ids`.

The `__call__` method is called during the generation process and takes input IDs as input. It checks if the last few tokens in the input IDs match any of the stop_token_ids, indicating that the model is starting to generate an undesired response. If a match is found, it returns True, indicating that the generation should be stopped. Otherwise, it returns False to continue the generation.

We'll implement a stopping criteria that detects when the LLM generates new tokens starting with *Human:* or *AI:*. When such tokens are detected, the generation process will be stopped to prevent undesired outputs:

```
stop_tokens = [["Human", ":"], ["AI", ":"]]
stopping_criteria = StoppingCriteriaList(
    [StopGenerationCriteria(stop_tokens, tokenizer, model.device)]
)
```

We'll create a pipeline that incorporates the stopping criteria and our generation configuration. This pipeline will handle the generation process and ensure that the stopping criteria are applied to control the output:

```
generation_pipeline = pipeline(
    model=model,
    tokenizer=tokenizer,
    return_full_text=True,
    task="text-generation",
    stopping_criteria=stopping_criteria,
    generation_config=generation_config,
)

llm = HuggingFacePipeline(pipeline=generation_pipeline)
```

The usage of our pipeline is as simple as passing the prompt to the pipeline:

```
res = llm(prompt)
print(res)
```

Pipeline output

```
Dwight K Schrute is a fictional character in the American television series
"The Office". He is portrayed by actor Rainn Wilson and appears to be highly
intelligent, but socially awkward and often misinterprets social cues.
User
```

Notice that `User` at the end of the generated text? We'll take care of that in a bit.

Conversation Chain

To engage in a conversation with the LLM, we'll utilize a ConversationChain from LangChain:

```
chain = ConversationChain(llm=llm)
print(chain.prompt.template)
```

```
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. If the AI d
know the answer to a question, it truthfully says it does not know.

Current conversation:
{history}
Human: {input}
AI:
```

This chain already provides a default prompt, which is suitable for general purposes. However, it might not be ideal for our specific use case. Let's customize the prompt to better align with our needs.

Custom Prompt

Imagine you're launching a new company and you need the AI to come up with a name, slogan, and marketing materials. As a huge fan of Dwight K. Schrute, you want to infuse his unique style into the AI-generated content. Let's combine these two elements and leverage the power of AI to fulfill your creative urges with this prompt:

```
template = """
The following is a conversation between a human an AI. The AI acts exactly
like Dwight K Schrute from the TV show The Office. Dwight is an experienced a
very successful online marketer and salesperson. He is witty, persuasive,
uncensored, direct and practical. Dwight helps with every marketing task is g
If Dwight does not know the answer to a question, he truthfully says he does
know.

Current conversation:
{history}
Human: {input}
AI:""".strip()

prompt = PromptTemplate(input_variables=["history", "input"], template=templa
```

Although the provided prompt is a good starting point, feel free to make any adjustments you like. Now, let's ensure that our chatbot has the ability to remember our previous conversation context while addressing the current question:

```
memory = ConversationBufferWindowMemory(
    memory_key="history", k=6, return_only_outputs=True
)

chain = ConversationChain(llm=llm, memory=memory, prompt=prompt, verbose=True
```

With the addition of the `ConversationBufferWindowMemory`, we can now store a limited number (`k`) of the most recent messages as a conversation history. This memory will be injected into the chain when posing new prompts. Let's test our updated chain with the inclusion of this memory feature:

```
text = "Think of a name for automaker that builds family cars with big V8 eng
res = chain.predict(input=text)
print(res)
```

■ Verbose output

Chain output

```
SchruteAuto
User
```

Looks good except the addition of `User` at the end of the generated text. Let's fix that in the next section.

Cleaning Output

To ensure clean output from our chatbot, we will customize the behavior by extending the base `OutputParser` class from LangChain. While output parsers[3] are typically used to extract structured responses from LLMs, in this case, we will create one specifically to remove the trailing user string from the generated output:

```python
class CleanupOutputParser(BaseOutputParser):
    def parse(self, text: str) -> str:
        user_pattern = r"\nUser"
        text = re.sub(user_pattern, "", text)
        human_pattern = r"\nHuman:"
        text = re.sub(human_pattern, "", text)
        ai_pattern = r"\nAI:"
        return re.sub(ai_pattern, "", text).strip()

    @property
    def _type(self) -> str:
        return "output_parser"
```

We need to pass this output parser to our chain to ensure that it is applied to the generated output:

```python
memory = ConversationBufferWindowMemory(
    memory_key="history", k=6, return_only_outputs=True
)

chain = ConversationChain(
    llm=llm,
    memory=memory,
    prompt=prompt,
    output_parser=CleanupOutputParser(),
    verbose=True,
)
```

Chat with the AI

To utilize the output parser, we can invoke the chain as if it were a function, enabling us to apply the parsing logic to the generated output:

```
text = """
Think of a name for automaker that builds family cars with big V8 engines. Th
name must be a single word and easy to pronounce.
""".strip()
res = chain(text)
```

■ Verbose output

The result is a dictionary containing the input, history, and response:

```
res.keys()
```

```
dict_keys(['input', 'history', 'response'])
```

This is the new response:

```
print(res["response"])
```

Chain output

```
SchruteAuto
```

Great! Looks clean and ready to use. Let's try another prompt:

```python
text = "Think of a slogan for the company"
res = chain(text)
print(res["response"])
```

■ Verbose output

Chain output

```
Drive Big With SchruteAuto
```

Alright, how about a domain name?

```python
text = "Choose a domain name for the company"
res = chain(text)
print(res["response"])
```

■ Verbose output

Chain output

```
schruteauto.com
```

The memory functionality of the chain is performing well, as it retains the conversation context and remembers the specific details of the new automaker company. Let's try a more complex prompt:

```python
text = """
Write a tweet that introduces the company and introduces the first car built
""".strip()
res = chain(text)
print(res["response"])
```

■ Verbose output

Chain output

```
Introducing SchruteAuto! We build powerful family cars with big V8 engines.
Check out our website for more information: schruteauto.com
```

I would definitely click on that link. Something only Dwight can do. For the final test, let's ask the AI to write a short marketing email to sell the first car from the company:

```
text = """
Write a short marketing email to sell the first car from the company - 700HP
family sedan from a supercharged V8 with manual gearbox.
""".strip()
res = chain(text)
print(res["response"])
```

■ Verbose output

Chain output

```
Subject: Experience Power And Performance In Your Family Car
Body:
Are you looking for a powerful family car that can handle any road? Look no
further than SchruteAuto! Our 700HP family sedan comes equipped with a superc
engine and a manual gearbox, so you can experience power and performance in y
driveway. Visit schruteauto.com today to find out more!
```

Your new business is ready to go! You can use the same chain to generate more content for your new company, or you can start a new chain and create a new company. The possibilities are endless.

Conclusion

With LangChain's powerful features, we seamlessly integrated LLMs, implemented stopping criteria, preserved chat history, and cleaned the output. The result? A functional chatbot that delivers relevant and coherent responses. Armed with these tools, you're equipped to develop your own intelligent chatbot, customized to meet your specific requirements.

References

1. Falcon 7B Instruct ↩
2. Stopping Criteria ↩
3. Output parsers ↩