

Retrieval Augmentation Generation - Langchain integration with local LLM

This integration allows LLM to efficiently retrieve and organize information, making it easier for users to find the information they need. With the help of provided documents, LLM can answer questions more precisely and accurately.

Preparation

First, let's set up the conda environment and install Python dependencies.

```
module load Anaconda3/2022.05
conda create -y --name langchain python=3.10.6
source activate langchain
conda info --envs
conda install -c pip
conda install -c anaconda ipykernel
pip3 -q install langchain tiktoken chromadb pypdf transformers InstructorEmbedding
pip3 -q install accelerate bitsandbytes sentencepiece Xformers torch sentencepiece accelerate sentence-
transformers
pip3 install bitsandbytes
```

Second, we may prepare some arbitrary pdf documents for the LLM to reference.

```
wget -q <https://www.dropbox.com/s/zoj9rnm7oyeaivb/new_papers.zip>
unzip -q new_papers.zip -d <your ref doc directory>
```

If you intend to use Jupyter Notebook in this experiment, you will need to install the Conda environment to an IPython kernel.

```
python -m ipykernel install --user --name=langchain
```

Evaluation (using Jupyter Lab)

Use the LangChain kernel to run the following Jupyter notebook.

<https://hpccenter.hk/file/download?p=%2Fpfss%2Ftoolkit%2Flangchain.ipynb>

Evaluation (python)

Alternatively, you can execute it as a Python file.

```
import torch
import transformers
from transformers import LlamaTokenizer, LlamaForCausalLM, GenerationConfig, pipeline
from transformers import pipeline
from langchain.llms import HuggingFacePipeline
import os
from langchain.vectorstores import Chroma
from langchain.text_splitter import RecursiveCharacterTextSplitter

from langchain.chains import RetrievalQA
from langchain.document_loaders import TextLoader
from langchain.document_loaders import PyPDFLoader
from langchain.document_loaders import DirectoryLoader

from InstructorEmbedding import INSTRUCTOR
from langchain.embeddings import HuggingFaceInstructEmbeddings
import json
import textwrap

tokenizer = LlamaTokenizer.from_pretrained("/pfss/toolkit/vicuna-7b")

model = LlamaForCausalLM.from_pretrained("/pfss/toolkit/vicuna-7b",
                                         # load_in_8bit=True,
                                         # device_map={"":0},
                                         # torch_dtype=torch.float16,
                                         # low_cpu_mem_usage=True
                                         )

pipe = pipeline(
    "text-generation",
    model=model,
    tokenizer=tokenizer,
    max_length=2048,
    temperature=0,
```

```

top_p=0.95,
repetition_penalty=1.15
)

local_llm = HuggingFacePipeline(pipeline=pipe)
print(local_llm('What is the capital of England?'))
"""
A. London
B. Manchester
C. Birmingham
D. Liverpool
"""

# Load and process the text files
# loader = TextLoader('single_text_file.txt')
loader = DirectoryLoader('<Your ref doc directory>', glob="/*.pdf", loader_cls=PyPDFLoader)

documents = loader.load()
len(documents)
#142

#splitting the text into
text_splitter = RecursiveCharacterTextSplitter(chunk_size=1000, chunk_overlap=200)
texts = text_splitter.split_documents(documents)

instructor_embeddings = HuggingFaceInstructEmbeddings(model_name="hkunlp/instructor-xl",
                                                         model_kwargs={"device": "cuda"})

# Embed and store the texts
# Supplying a persist_directory will store the embeddings on disk
#persist_directory = '/pfss/scratch01/don/langchainRef/db'
persist_directory = 'db'

## Here is the nmew embeddings being used
embedding = instructor_embeddings

vectordb = Chroma.from_documents(documents=texts,
                                 embedding=embedding,
                                 persist_directory=persist_directory)

import logging
from langchain.retrievers.multi_query import MultiQueryRetriever
#ref: https://python.langchain.com/docs/use\_cases/question\_answering/how\_to/vector\_db\_qa

```

```

#for use vectordb only
#retriever = vectordb.as_retriever(search_kwargs={"k": 3})
#for use multiqueryRetriever
retriver = MultiQueryRetriever.from_llm(retriever=vectordb.as_retriever(),
                                         llm=local_llm)

# create the chain to answer questions
qa_chain = RetrievalQA.from_chain_type(llm=local_llm,
                                       chain_type="stuff",
                                       retriever=retriever,
                                       return_source_documents=True)

human_prompt = 'What is the meaning of life?'

def get_prompt(human_prompt):
    prompt_template=f"### Human: {human_prompt} \n### Assistant:"
    return prompt_template

print(get_prompt('What is the meaning of life?'))

def remove_human_text(text):
    return text.split('### Human:', 1)[0]

def parse_text(data):
    for item in data:
        text = item['generated_text']
        assistant_text_index = text.find('### Assistant:')
        if assistant_text_index != -1:
            assistant_text = text[assistant_text_index+len('### Assistant:').strip():].strip()
            assistant_text = remove_human_text(assistant_text)
            wrapped_text = textwrap.fill(assistant_text, width=100)
            print(wrapped_text)

data = [{'generated_text': '### Human: What is the capital of England? \n### Assistant: The capital city of England is London.'}]
parse_text(data)

"""
### Human: What is the meaning of life?
### Assistant:
The capital city of England is London.
"""

## Cite sources

```

```

import textwrap

def wrap_text_preserve_newlines(text, width=110):
    # Split the input text into lines based on newline characters
    lines = text.split('\n')

    # Wrap each line individually
    wrapped_lines = [textwrap.fill(line, width=width) for line in lines]

    # Join the wrapped lines back together using newline characters
    wrapped_text = '\n'.join(wrapped_lines)

    return wrapped_text

def process_llm_response(llm_response):
    print(wrap_text_preserve_newlines(llm_response['result']))
    print("\n\nSources:")
    for source in llm_response["source_documents"]:
        print(source.metadata['source'])
        print(source)

# full example
query = "What is Flash attention?"
llm_response = qa_chain(query)
process_llm_response(llm_response)
"""

```

FlashAttention is a new attention algorithm proposed by the authors that reduces the number of memory accesses required for attention computation. It achieves this by splitting the input into blocks and making multiple passes over them, allowing for incremental computation of the softmax reduction. Additionally, it stores the softmax normalization factor from the forward pass to compute attention on-chip in the backward pass, reducing the need for accessing external memory such as High Bandwidth Memory (HBM). The authors claim

that their implementation of FlashAttention is faster and more memory efficient than existing methods, and has been shown to improve the quality of Transformer models while also enabling longer context for better results.

Sources:

/pfss/scratch01/don/langchainRef/Flash-attention.pdf

page_content='access.\nWe propose FlashAttention , a new attention algorithm that computes exact attention with far fewer\nmemory accesses. Our main goal is to avoid reading and writing the attention matrix to and from HBM.\nThis requires (i) computing the softmax reduction without access to the whole input (ii) not storing

the large intermediate attention matrix for the backward pass. We apply two well-established techniques to address these challenges. (i) We restructure the attention computation to split the input into blocks and make several passes over input blocks, thus incrementally performing the softmax reduction (also known as tiling). (ii) We store the softmax normalization factor from the forward pass to quickly recompute attention on-chip in the backward pass, which is faster than the standard approach of reading the intermediate attention matrix from HBM. We implement FlashAttention in CUDA to achieve fine-grained control over memory access and

metadata={'page': 1, 'source': '/pfss/scratch01/don/langchainRef/Flash-attention.pdf'}
/pfss/scratch01/don/langchainRef/Flash-attention.pdf

page_content='•Higher Quality Models. FlashAttention scales Transformers to longer sequences, which improves their quality and enables new capabilities. We observe a 0.7 improvement in perplexity on GPT-2 and 6.4 points of lift from modeling longer sequences on long-document classification [13].

FlashAttention enables the first Transformer that can achieve better-than-chance performance on the Path-X [80] challenge, solely from using a longer sequence length (16K). Block-sparse FlashAttention enables a Transformer to scale to even longer sequences (64K), resulting in the first model that can achieve better-than-chance performance on Path-256. •Benchmarking Attention. FlashAttention is up to 3x faster than the standard attention implementation across common sequence lengths from 128 to 2K and scales up to 64K. Up to sequence length of 512, FlashAttention is both faster and more memory-efficient than any existing attention method, whereas

metadata={'page': 2, 'source': '/pfss/scratch01/don/langchainRef/Flash-attention.pdf'}
/pfss/scratch01/don/langchainRef/Flash-attention.pdf

page_content='aware—accounting for reads and writes between levels of GPU memory. We propose FlashAttention, an IO-aware exact attention algorithm that uses tiling to reduce the number of memory reads/writes between GPU high bandwidth memory (HBM) and GPU on-chip SRAM. We analyze the IO complexity of FlashAttention, showing that it requires fewer HBM accesses than standard attention, and is optimal for a range of SRAM sizes. We also extend FlashAttention to block-sparse attention, yielding an approximate attention algorithm that is faster than any existing approximate attention method. FlashAttention trains Transformers faster than existing baselines: 15% end-to-end wall-clock speedup on BERT-large (seq. length 512) compared to the MLPerf 1.1 training speed record, 3x speedup on GPT-2 (seq. length 1K), and 2.4x speedup on long-range arena (seq. length 1K-4K). FlashAttention and block-sparse FlashAttention enable longer context in Transformers, yielding higher quality models'

metadata={'page': 0, 'source': '/pfss/scratch01/don/langchainRef/Flash-attention.pdf'}
''''

Revision #7

Created 11 August 2023 06:02:03 by Don Chu

Updated 21 August 2023 02:17:48 by Loki Ng