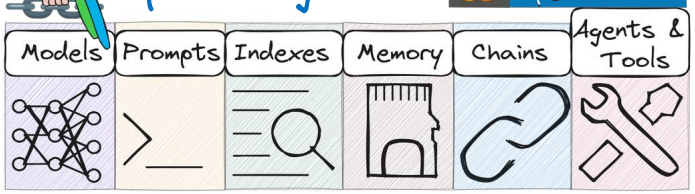
LangChain onepager

import langchain

Open in Colab



Calling OpenAI large language model — Models

```
from langchain.llms import OpenAI
llm = OpenAI(  model_name="text-davinci-003",  temperature=0.01)
llm("Suggest 3 bday gifts for a data scientist")
>>> 1. A subscription to a data science magazine
>>> 2. A set of data science books
>>> 3. A data science-themed mug or t-shirt
```

Conversation schemas: History and Instructions

```
from langchain.chat_models import ChatOpenAI
from langchain.schema import HumanMessage,AIMessage,SystemMessage
chat = ChatOpenAI(model_name="gpt-3.5-turbo", temperature=0.01)
conversation_history = [
    HumanMessage(content="Suggest 3 bday gifts for a data scientist"),
    AIMessage(content="What is your price range?"),
    HumanMessage(content="Under 100$") ]
chat(conversation_history).content
>>> 1. A data science book: Consider gifting a popular and highly ...
>>> 2. Data visualization tool: A data scientist often deals with ....
>>> 3. Subscription to a data science platform: Give them access to ....
system_instruction = SystemMessage(content = """You work as an assistant
in an electronics store. Your income depends on the items you sold""")
user_message = HumanMessage(content="3 bday gifts for a data scientist")
chat([system_instruction, user_message]).content
>>> 1. Laptop: A high-performance laptop with a powerful processor ....
>>> 2. External Hard Drive: Data scientists deal with large datasets ....
>>> 3. Data Science Books: Books related to data science can be ....
```

Open-source models

```
from auto_gptq import AutoGPTQForCausalLM, BaseQuantizeConfig
from transformers import AutoTokenizer, AutoModelForCausalLM
model_name = "TheBloke/llama-2-13B-Guanaco-QLoRA-GPTQ"
tokenizer = AutoTokenizer.from_pretrained(model_name, use_fast=True)
# Initialize the AutoGPTQForCausalLM model with appropriate parameters
model = AutoGPTQForCausalLM.from_quantized(
    model_name, use_safetensors=True, trust_remote_code=True,
    device_map="auto", quantize_config=None )
# Tokenize the query and convert to CUDA tensor
input_ids = tokenizer(query, return_tensors="pt").input_ids.cuda()
# Generate text using the model with specified settings
output = model.generate(inputs=input_ids, temperature=0.1)
```

Text generation parameters

```
The temperature parameter affects the randomness of the token generation
Top-k sampling limits token generation to the top k most likely at each step
Top-p (nucleus) sampling limits token generation to cumulative probability p
The length of generated tokens can be specified by max_tokens parameter
llm = OpenAI(temperature=0.5, top_k=10, top_p=0.75, max_tokens=50)
```

Quantization

```
from transformers import BitsAndBytesConfig
# Configure BitsAndBytesConfig for 4-bit quantization
bnb_config = BitsAndBytesConfig(
    load_in_4bit=True,  bnb_4bit_compute_dtype=torch.bfloat16,
    bnb_4bit_quant_type="nf4", bnb_4bit_use_double_quant=True)
model_4bit = AutoModelForCausalLM.from_pretrained(
    model_name_or_path, quantization_config=bnb_config,
    device_map="auto", trust_remote_code=True)
```

Fine-tuning — Models

```
from peft import LoraConfig, get_peft_model, prepare_model_for_kbit_training
pretrained_model = AutoModelForCausalLM.from_pretrained(...)
pretrained_model.gradient_checkpointing_enable()
model = prepare_model_for_kbit_training(pretrained_model)
# Specify LoRa configuration
config = LoraConfig(r=16, lora_alpha=32, lora_dropout=0.05, bias="none",
    target_modules=["query_key_value"], task_type="CAUSAL_LM")
model = get_peft_model(model, config)
# Set training parameters
trainer = transformers.Trainer(
    model=model, train_dataset=train_dataset,
    args=transformers.TrainingArguments(
        num_train_epochs=10, per_device_train_batch_size=8, ...),
    data_collator=transformers.DataCollatorForLanguageModeling(tokenizer))
model.config.use_cache = False
trainer.train()
```

Prompt Templates — Prompts

```
from langchain.prompts import PromptTemplate
# Define the template for SEO description
template = "Act as an SEO expert. Provide a SEO description for {product}"
# Create the prompt template
prompt = PromptTemplate(input_variables=["product"], template=template)
# Pass in an input to return a formatted prompt
formatted_prompt = prompt.format(product="Electric Scooter")
llm(formatted_prompt)
>>> The Electric Scooter is the perfect way to get around town quickly ...
formatted_prompt = prompt.format(product="Perpetuum Mobile")
llm(formatted_prompt)
>>> Perpetuum Mobile is an innovative product that provides a ...
```

```
from langchain.prompts import FewShotPromptTemplate
# Define three examples for the 3-shot learning
examples = [
    {"email_text": "Win a free iPhone!", "category": "Spam"},
    {"email_text": "Next Sprint Planning Meeting.", "category": "Meetings"},
    {"email_text": "Version 2.1 of Y is now live", "category": "Project Updates"}]
# Create a PromptTemplate for classifying emails
prompt_template = PromptTemplate(template="Classify the email:
  {email_text}/n{category}", input_variables=["email_text", "category"])
# Create a FewShotPromptTemplate using PromptTemplate and examples
few_shot_prompt = FewShotPromptTemplate(example_prompt =
  prompt_template, examples = examples, suffix = "Classify the email:
  {email_text}", input_variables=["email_text"])
```

Document loaders — Indexes

```
from langchain.document_loaders import csv_loader, DirectoryLoader,
    WebBaseLoader, JSONLoader, UnstructuredPDFLoader, .....
loader = DirectoryLoader('../', glob="**/*.md")
loader = csv_loader.CSVLoader(...)
loader = WebBaseLoader(...)
loader = JSONLoader(...)
loader = UnstructuredPDFLoader(...)
loaded_documents = loader.load()
```

Retrievers and Vectorstores

```
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain.vectorstores import FAISS, Chroma, Pinecone, ...
# Split docs into texts
splitter = RecursiveCharacterTextSplitter(chunk_size=800, chunk_overlap=50)
texts = splitter.split_documents(loaded_documents)
# Embed your texts and store them in a vectorstore
db = FAISS.from_documents(texts, embeddings)
db = FAISS.from_texts(["some_string_abc", "some_string_xyz"], embeddings)
# Perform similarity search
db.similarity_search(query)
# Initialize retriever and ask for relevant documents back
retriever = db.as_retriever()
docs = retriever.get_relevant_documents(some_query)
```

Setup Memory — Memory

```
from langchain.memory import ConversationBufferMemory
memory = ConversationBufferMemory(memory_key="chat_history")
# Setup predefined memories
memory.chat_memory.add_user_message("Hi!")
memory.chat_memory.add_ai_message("Welcome! How can I help you?")
memory_variables = memory.load_memory_variables({...})
# Add response to memory
memory.add_ai_message(chat_response.content)
```

Chains — Chains

```
from langchain.chains import ConversationChain, summarize, question_answering
from langchain.schema import StrOutputParser
# Templates for summarizing customer feedback and drafting email response
feedback_summary_prompt = PromptTemplate.from_template(
    """You are a customer service manager. Summarize the customer feedback.
Customer Feedback: {feedback}
Summary:""")
email_prompt = PromptTemplate.from_template(
    """You are a customer service representative. Given the summary of
customer feedback, it is your job to write a professional email response.
Feedback Summary: {summary}
Email Response:""" )
feedback_chain = feedback_summary_prompt | llm | StrOutputParser()
summary_chain = {"summary": feedback_chain} | email_prompt | llm | StrOutputParser()
summary_chain.invoke({"feedback": "Incorrect item has arrived"})
```

```
# Predefined chains: summarization and Q&A
chain = summarize.load_summarize_chain(llm, chain_type="stuff")
chain.run(loaded_documents)
chain = question_answering.load_qa_chain(llm, chain_type="stuff")
chain.run(input_documents=loaded_documents, question = <input>)
# Use memory
conversation=ConversationChain(llm=llm,memory=ConversationBufferMemory())
conversation.run("Name the tallest mountain in the world")   >>> Everest
conversation.run("How high is it?")                          >>> 8848 m
```

Tools — Agents and Tools

```
from langchain.agents import load_tools
tools = load_tools(["serpapi", "llm-math", ...], llm=llm)
from langchain.tools import StructuredTool, BaseTool
def multiply_two_numbers(a: float, b: float) -> float:
    """multiply two numbers"""
    return a * b
multiplier_tool = StructuredTool.from_function(multiply_two_numbers)
```

Agents

```
from langchain.agents import initialize_agent, AgentType, BaseSingleActionAgent
agent = initialize_agent(
        tools, llm, agent=AgentType.ZERO_SHOT_REACT_DESCRIPTION )
agent.run({"input": "How old would Harry Potter be when Daniel
Radcliffe was born?"})                                  >>>9
# create own agents and tools
class UnitConversionTool(BaseTool):
    name = "Unit Conversion Tool"
    description = "Converts American units to International units"
    def _run(self, text: str):
        def miles_to_km(match):
            miles = float(match.group(1))
            return f"{miles * 1.60934:.2f} km"
        return re.sub(r'\b(\d+(\.\d+)?)\s*(miles|mile)\b', miles_to_km, text)
    def _arun(self, text: str):
        raise NotImplementedError("No async yet")
agent = initialize_agent(
    agent='chat-conversational-react-description',
    tools=[UnitConversionTool()],
    llm=llm,
    memory=memory
)
agent.run("five miles")
>>> 8.05 kilometers
```


v1.1.0 – 10.11.2023
latest version (click) →
by Ivan Reznikov
medium.com/@ivanreznikov
linkedin.com/in/reznikovivan