**Understanding of LangChain and LangGraph**

## Chatbots With Langgraph

!pip install langgraph langsmith

!pip install langchain langchain\_groq langchain\_community

import os

os.environ["GROQ\_API\_KEY"] = "gsk\_UokaJx1jtKtaDqS19eWFWGdyb3FYXTtvpZ4gM8AYy9evSdXgwvEL"

os.environ['LANGSMITH\_API\_KEY'] = 'lsv2\_pt\_541c5896e34f41c4a1a446cee0b1ac16\_ef6de050a5'

from google.colab import userdata

groq\_api\_key=os.getenv('GROQ\_API\_KEY')

langsmith=os.getenv('LANGSMITH\_API\_KEY')

print(langsmith)

import os

os.environ["LANGCHAIN\_API\_KEY"] = langsmith

os.environ["LANGCHAIN\_TRACING\_V2"]="true"

os.environ["LANGCHAIN\_PROJECT"]="CourseLanggraph"

from langchain\_groq import ChatGroq

llm=ChatGroq(groq\_api\_key=groq\_api\_key,model\_name="Gemma2-9b-It")

llm

## Start Building Chatbot Using Langgraph

from typing import Annotated

from typing\_extensions import TypedDict

from langgraph.graph import StateGraph,START,END

from langgraph.graph.message import add\_messages

class State(TypedDict):

# Messages have the type "list". The `add\_messages` function

# in the annotation defines how this state key should be updated

# (in this case, it appends messages to the list, rather than overwriting them)

messages:Annotated[list,add\_messages]

graph\_builder=StateGraph(State)

graph\_builder

def chatbot(state:State):

return {"messages":llm.invoke(state['messages'])}

graph\_builder.add\_node("chatbot",chatbot)

graph\_builder

graph\_builder.add\_edge(START,"chatbot")

graph\_builder.add\_edge("chatbot",END)

graph=graph\_builder.compile()

from IPython.display import Image, display

try:

display(Image(graph.get\_graph().draw\_mermaid\_png()))

except Exception:

pass

while True:

user\_input=input("User: ")

if user\_input.lower() in ["quit","q"]:

print("Good Bye")

break

for event in graph.stream({'messages':("user",user\_input)}):

print(event.values())

for value in event.values():

print(value['messages'])

print("Assistant:",value["messages"].content)

**Code explanation:  
  
1. Installing Required Libraries**

!pip install langgraph langsmith

!pip install langchain langchain\_groq langchain\_community

* These commands install essential libraries for building LangGraph-based applications, including support for Groq models and LangChain integrations.

**2. Setting Environment Variables for API Access**

import os

os.environ["GROQ\_API\_KEY"] = "..." # GROQ LLM Key

os.environ['LANGSMITH\_API\_KEY'] = "..." # LangSmith Tracing Key

* API keys are securely stored in environment variables for authorized access to Groq's LLM and LangChain’s LangSmith tracing tools.

**3. Configuring LangChain Tracing**

os.environ["LANGCHAIN\_API\_KEY"] = langsmith

os.environ["LANGCHAIN\_TRACING\_V2"] = "true"

os.environ["LANGCHAIN\_PROJECT"] = "CourseLanggraph"

* These settings enable advanced tracing and monitoring of LangChain workflows through **LangSmith**, helping debug and understand agent behavior.

**4. Initializing the Groq Language Model**

from langchain\_groq import ChatGroq

llm = ChatGroq(groq\_api\_key=groq\_api\_key, model\_name="Gemma2-9b-It")

* The chatbot is powered by **Groq’s Gemma2-9b-It model**, known for its speed and efficiency on inference chips.

**5. Defining the State Structure**

from typing import Annotated

from typing\_extensions import TypedDict

from langgraph.graph.message import add\_messages

class State(TypedDict):

messages: Annotated[list, add\_messages]

* A State dictionary tracks the conversation, where messages is a list that gets updated using add\_messages (appends new messages instead of replacing them).

**6. Creating the LangGraph Workflow**

from langgraph.graph import StateGraph, START, END

graph\_builder = StateGraph(State)

def chatbot(state: State):

return {"messages": llm.invoke(state["messages"])}

* A **LangGraph** object (graph\_builder) is initialized using the defined state.
* A single node named "chatbot" is created, which invokes the LLM with the current messages.

**7. Connecting Graph Nodes**

graph\_builder.add\_node("chatbot", chatbot)

graph\_builder.add\_edge(START, "chatbot")

graph\_builder.add\_edge("chatbot", END)

graph = graph\_builder.compile()

* The graph consists of a simple flow: START → chatbot → END. It is then compiled into an executable graph object.

**8. Optional Visualization (if supported)**

from IPython.display import Image, display

display(Image(graph.get\_graph().draw\_mermaid\_png()))

* Attempts to render a flowchart of the chatbot graph using MermaidJS (optional, may not work outside Jupyter/Colab).

**9. Running the Chatbot Interaction Loop**

while True:

user\_input = input("User: ")

if user\_input.lower() in ["quit", "q"]:

print("Good Bye")

break

for event in graph.stream({'messages': ("user", user\_input)}):

for value in event.values():

print("Assistant:", value["messages"].content)

* A simple REPL (read–eval–print loop) is implemented.
* User inputs are streamed into the chatbot graph.
* The LLM’s response is printed in real time.

## Chatbots With Langgraph Tools

!pip install langgraph langsmith langchain langchain\_groq langchain\_community

from typing import Annotated

from typing\_extensions import TypedDict

!pip install arxiv wikipedia

## Working With Tools

from langchain\_community.utilities import ArxivAPIWrapper,WikipediaAPIWrapper

from langchain\_community.tools import ArxivQueryRun,WikipediaQueryRun

## Arxiv And Wikipedia tools

arxiv\_wrapper=ArxivAPIWrapper(top\_k\_results=1,doc\_content\_chars\_max=300)

arxiv\_tool=ArxivQueryRun(api\_wrapper=arxiv\_wrapper)

api\_wrapper=WikipediaAPIWrapper(top\_k\_results=1,doc\_content\_chars\_max=300)

wiki\_tool=WikipediaQueryRun(api\_wrapper=api\_wrapper)

wiki\_tool.invoke("who is Kim Namjoon")

arxiv\_tool.invoke("attention is all you need")

tools=[wiki\_tool]

## Langgraph Application

from langgraph.graph.message import add\_messages

class State(TypedDict):

messages:Annotated[list,add\_messages]

import os

os.environ["GROQ\_API\_KEY"] = "gsk\_UokaJx1jtKtaDqS19eWFWGdyb3FYXTtvpZ4gM8AYy9evSdXgwvEL"

from langgraph.graph import StateGraph,START,END

graph\_builder= StateGraph(State)

from langchain\_groq import ChatGroq

from google.colab import userdata

groq\_api\_key=os.getenv("GROQ\_API\_KEY")

llm=ChatGroq(groq\_api\_key=groq\_api\_key,model\_name="Gemma2-9b-It")

llm

llm\_with\_tools=llm.bind\_tools(tools=tools)

def chatbot(state:State):

return {"messages":[llm\_with\_tools.invoke(state["messages"])]}

from langgraph.prebuilt import ToolNode,tools\_condition

graph\_builder.add\_node("chatbot",chatbot)

tool\_node = ToolNode(tools=tools)

graph\_builder.add\_node("tools", tool\_node)

graph\_builder.add\_conditional\_edges(

"chatbot",

tools\_condition,

)

graph\_builder.add\_edge("tools", "chatbot")

graph\_builder.add\_edge(START,"chatbot")

graph=graph\_builder.compile()

from IPython.display import Image, display

try:

display(Image(graph.get\_graph().draw\_mermaid\_png()))

except Exception:

# This requires some extra dependencies and is optional

pass

user\_input="Hi there!, My name is Esha"

events=graph.stream(

{"messages": [("user", user\_input)]},stream\_mode="values"

)

for event in events:

event["messages"][-1].pretty\_print()

user\_input = "what is 911."

# The config is the \*\*second positional argument\*\* to stream() or invoke()!

events = graph.stream(

{"messages": [("user", user\_input)]},stream\_mode="values"

)

for event in events:

event["messages"][-1].pretty\_print()

**Code explanation:  
  
1. Installing Required Libraries**

!pip install langgraph langsmith langchain langchain\_groq langchain\_community

!pip install arxiv wikipedia

* Installs the LangChain ecosystem tools, along with APIs to query **ArXiv** (for research papers) and **Wikipedia** (for general information).

**2. Importing Tool Wrappers**

from langchain\_community.utilities import ArxivAPIWrapper, WikipediaAPIWrapper

from langchain\_community.tools import ArxivQueryRun, WikipediaQueryRun

* **ArxivAPIWrapper**: Interfaces with the ArXiv API to retrieve scientific papers.
* **WikipediaAPIWrapper**: Interfaces with Wikipedia to fetch concise summaries.

**3. Creating and Testing Tool Instances**

arxiv\_wrapper = ArxivAPIWrapper(top\_k\_results=1, doc\_content\_chars\_max=300)

arxiv\_tool = ArxivQueryRun(api\_wrapper=arxiv\_wrapper)

api\_wrapper = WikipediaAPIWrapper(top\_k\_results=1, doc\_content\_chars\_max=300)

wiki\_tool = WikipediaQueryRun(api\_wrapper=api\_wrapper)

* Both tools are configured to return **only the top result** with a maximum of **300 characters**, providing focused and brief outputs.

**4. Preparing Tools for LangGraph**

tools = [wiki\_tool]

* A list of tools (in this case, just the Wikipedia tool) is prepared to be injected into the LLM pipeline.

**5. Defining the State for LangGraph**

from typing import Annotated

from typing\_extensions import TypedDict

from langgraph.graph.message import add\_messages

class State(TypedDict):

messages: Annotated[list, add\_messages]

* A State dictionary tracks messages in the conversation and updates them using the add\_messages function.

**6. Initializing LLM with Tools**

from langchain\_groq import ChatGroq

llm = ChatGroq(groq\_api\_key=groq\_api\_key, model\_name="Gemma2-9b-It")

llm\_with\_tools = llm.bind\_tools(tools=tools)

* The Groq LLM (Gemma2-9b-It) is bound to the Wikipedia tool using .bind\_tools(). This allows the LLM to decide when to use external tools based on user input.

**7. Defining the Chatbot Node**

def chatbot(state: State):

return {"messages": [llm\_with\_tools.invoke(state["messages"])]}

* This function acts as the main chatbot node, invoking the LLM with any incoming messages and optionally using tools.

**8. Building the LangGraph Flow**

from langgraph.graph import StateGraph, START, END

from langgraph.prebuilt import ToolNode, tools\_condition

graph\_builder = StateGraph(State)

graph\_builder.add\_node("chatbot", chatbot)

tool\_node = ToolNode(tools=tools)

graph\_builder.add\_node("tools", tool\_node)

Two nodes are created:

* "chatbot": Handles normal LLM replies.
* "tools": Executes external tools if required.

The decision to go to the "tools" node is made using tools\_condition, which checks if a tool is needed.

graph\_builder.add\_conditional\_edges("chatbot", tools\_condition)

graph\_builder.add\_edge("tools", "chatbot")

graph\_builder.add\_edge(START, "chatbot")

graph = graph\_builder.compile()

This flow allows flexible routing:

* Start → chatbot → (optional tools) → chatbot → End

**9. Visualizing the Graph**

from IPython.display import Image, display

display(Image(graph.get\_graph().draw\_mermaid\_png()))

* Optionally visualizes the chatbot logic using a MermaidJS graph.

**10. Running the Chatbot with Streaming**

user\_input = "Hi there!, My name is Esha"

events = graph.stream({"messages": [("user", user\_input)]}, stream\_mode="values")

* Starts the chat session. The user message is streamed through the graph and processed by the LLM (and tools if needed).

for event in events:

event["messages"][-1].pretty\_print()

* Prints the assistant’s final reply in a formatted way.

user\_input = "what is 911."

# Running another input

events = graph.stream({"messages": [("user", user\_input)]}, stream\_mode="values")

for event in events:

event["messages"][-1].pretty\_print()

* Another user input is processed similarly, showcasing the system’s ability to query external sources dynamically.