ReAct with Tool Calling

Copyright 2025. Korea Aerospace University. All rights reserved.

Text 기반 ReAct의 문제점

- Prompt로 LLM에게 Thought-Action을 텍스트로 출력하게 명령하였으나 LLM이 이를 정확하게 따르지 않을 경우 전체 추론 프로세스가 망가질 수 있다.
- 구체적으로 LLM은 우리가 넣어준 Tool list를 보고 적당한 툴을 선택하고 그 툴 사용에 필요한 arguments를 추출하고 이를 우리가 파싱을 위해서 기대하는 포맷으로 반환하여야 하는데 여기에 우리가 기대했던 포맷과 약간만 다르더라도 파싱 에러가 날 위험성이 항상 있다.

- 이를 해결하기 위하여 OpenAI는 'function calling' 이라는 기법을 도입하였다(지금은 tool calling이라는 보다 넓은 의미를 내포하는 용어 사용)
- Tool calling을 직관적으로 설명하자면, ① Prompting으로 LLM이 엄격하게 Action 포맷을 따르게 하는 것 보다 ② JSON 포맷으로 function을 기술하고 JSON 포맷으로 Action 포맷을 따르게 하는 것이 더 효과적이다.

Copyright 2025. Korea Aerospace University. All rights reserved.

Text 기반 ReAct(초기 방식)

- 프롬프트 지시: "너는 Thought:로 생각을, Action: tool_name[arguments] 형식으로 행동을 출력해야 해."
- LLM의 출력(Text)
  - Thought: 사용자가 날씨를 물어봤으니, search 툴을 사용해야겠어.
  - Action: search[query="오늘 서울 날씨"]
- Python 코드의 역할:
  ① LLM이 반환한 텍스트(String)를 파싱
  ② Action: 키워드가 있는지 확인
  ③ search라는 함수 이름과 query="오늘 서울 날씨"라는 인수를 정규식(regex)이나 문자열 분리(split)로 추출
  ④ search() 함수를 실행
- 단점: 매우 불안정합니다 (Brittle). LLM이 Action: search("오늘 서울 날씨")처럼 형식을 조금만 다르게 출력해도 파싱 에러가 발생합니다.

Copyright 2025. Korea Aerospace University. All rights reserved.

Tool Calling 기반 ReAct(현대 방식)

- 프롬프트 지시: "너는 Thought를 통해 생각하고, 필요하면 도구를 호출(Tool Call)해야 해. 도구 호출이 필요 없으면 최종 생각을 말해." (이것은 개념 설명을 위한 예시임)
- LLM의 출력 (Tool Call Object):
  - 내부적으로 "사용자가 날씨를 물어봤으니 search 툴을 써야겠다"라고 **생각(Thought)**, 텍스트 대신 Tool Call 객체를 반환

```
{"tool_call": {
    "name": "search",
    "arguments": { "query": "오늘 서울 날씨" }
  }}
```

- Python 코드의 역할:
  ① LLM 응답에 tool_call이 있는지 간단히 확인
  ② response.tool_call.name (함수 이름)과 response.tool_call.arguments (인수)를 안정적으로 추출
  ③ search(**response.tool_call.arguments)처럼 함수를 바로 실행
- 장점: 매우 안정적(Robust). LLM이 항상 정해진 JSON 형식으로 도구를 요청하므로 파싱 에러가 거의 없음.

Copyright 2025. Korea Aerospace University. All rights reserved.

Tool Calling Python Pseudo Code

```
response = call_llm(messages, tools=[search_tool_spec])
messages.append(response.message) # 모델의 응답(tool_call 요청 포함)을 추가
if response.has_tool_call: # Action에 해당하며, 특별한 파싱 필요 없음
    tool_name = response.tool_call.name
    tool_args = response.tool_call.arguments # <-- 안정적!
    observation = execute_tool(tool_name, **tool_args)
    # Tool Call 결과를 'tool' 역할로 다시 넣어줌
    messages.append({"role": "tool", "content": str(observation)})
    continue # 다음 루프
else:
    # Tool Call이 없으면 최종 답변
    print(response.text)
    break
```

Copyright 2025. Korea Aerospace University. All rights reserved.

텍스트 기반 ReAct vs. Tool Calling 기반 ReAct

구분	1. 텍스트 기반 ReAct	2. Tool Calling 기반 ReAct
프롬프트 (Prompt)	• 시스템 프롬프트에 도구 목록과 사용법을 텍스트로 명시해야 함. • Thought: ...와 Action: ...[...]라는 정확한 텍스트 형식을 강제해야 함.	• 시스템 프롬프트는 "도구를 사용하라" 정도로 간단함. • 도구 정보는 API의 tools 파라미터(JSON)로 전달됨.
LLM 응답	• 단순 텍스트 (String) • "Thought: ... \nAction: get_weather[서울]"	• 구조화된 객체 (Object) • response.tool_calls = [...]
Action' 파싱	• re.search(...) 같은 정규식이나 문자열 파싱 필요. • LLM이 형식을 조금만 다르게 출력하면 (예: Action:get_weather("서울")) 실패함.	• response.tool_calls가 있는지 확인하면 됨. • 인수는 json.loads(...)로 안정적으로 파싱됨.
Observation' 전달	role: "user" (또는 system)로 "Observation: ..." 텍스트를 전달.	role: "tool"이라는 전용 역할을 사용하여 tool_call_id와 함께 결과를 전달.

Copyright 2025. Korea Aerospace University. All rights reserved.

Tool Calling 지원 LLM

- Tool Calling은 pre-trained model을 fine-tuning하는 단계에서 학습
- Tool Calling (또는 'Function Calling')을 지원하는 LLM은 이제 매우 보편화
  - 주요 상용 API 모델은 대부분 이 기능을 기본으로 지원하며, 오픈소스 모델들도 이 기능을 지원하도록 파인튜닝된 버전 다수임
  - 어떤 모델을 사용하는가에 따라 API 사양이 조금씩 상이함

1. OpneAI – GPT-4o, GPT-4 Turbo, GPT-3.5 Turbo
2. Google – Gemini 1.5 Pro, Gemini 1.5 Flash, Gemini 1.0 Pro
3. Anthropic – Claude 3 (Opus, Sonnet, Haiku)
4. Mistral AI – Mistral Large 2, Mistral Small
5. Cohere – Command R, Command R+
6. Meta(Llama 3) – Meta-Llama-3-Instruct 버전(8B, 70B)
7. Mistral(Open Source) – Mistral-7B-Instruct, Mixtral-8x7B-Instruct
8. Microsoft(Phi-3) – Phi-3-mini-instruct
9. Hugging Face 특화 모델

Copyright 2025. Korea Aerospace University. All rights reserved.

Tool Calling 정책

- 최신 상용 모델은 사고과정 직접 서술(step-by-step, chain-of-thought)을 정책적으로 차단
- Prompt 본문에 " 생각을 자세히 써라 " 는 문장 자체가 무시/거부 되곤 함
- 만약에 추론 과정에서 내려진 결정을 로그로 남기고 싶다면?
- 요약하는 것을 하나의 툴로 정의하고 tool call하기 전에 요약을 호출할 것을 강제할 수 있음
  - 굳이 이렇게까지 할 필요가 있을까?

Copyright 2025. Korea Aerospace University. All rights reserved.

Tool Calling기반 ReAct 예제

전체 흐름

- Tool 구현

① 사용할 툴 정의 (JSON Schema)

② Python으로 실제 툴 구현

③ Python 툴 호출 방법 마련 – 간단한 Dictionary 또는 Tool Registry로 구현

- ReAct 구현

① System Prompt 정의

② 초기 메시지 작성하고 메시지 리스트에 추가 – System Prompt + 사용자 Query

③ ReAct Loop 구현

① LLM에게 메시지 리스트를 가지고 질의하고 새로운 메시지 반환
② tool_calls가 있으면 Action 단계로 메시지 리스트에 반환된 메시지 내용을 추가하고 아래 Tool 실행 Loop로 진입
③ Tool 실행 Loop: tool_calls에서 하나씩 tool 실행하고 tool 결과를 메시지 리스트에 추가 – Observation 단계
④ tool_calls가 없으면 Final 단계로 간주하고 최종 메시지 반환

Copyright 2025. Korea Aerospace University. All rights reserved.

ReAct with Tool Calling System Prompt 예

SYSTEM_PROMPT = """₩

You are a helpful AI assistant that uses tools with a ReAct-style loop.

- 먼저 질문을 이해하고, 필요하면 내부적으로 생각(Thought)을 정리합니다.
- 실제로 계산/검색이 필요하면 제공된 tools를 호출합니다.
- tool 결과(Observation)를 보고 다시 생각한 뒤, 최종 답변(Final)을 한국어로 친절하게 정리합니다.

규칙:
- 사용자가 이해할 필요가 없는 내부 추론은 굳이 자세히 드러내지 않아도 됩니다.
- 하지만 디버깅과 교육을 위해, 때때로 짧은 '생각의 이유' 정도는 자연스럽게 설명해도 괜찮습니다.
- 도구가 필요한 작업(예: 계산, 시간 조회 등)은 반드시 tool_calls로 처리하고,
  스스로 결과를 '지어내지' 마세요.
"""

Copyright 2025. Korea Aerospace University. All rights reserved.

실제 코드

- react_tool_agent.py

Copyright 2025. Korea Aerospace University. All rights reserved.