Inteligência Artificial 2

Aula 24: IA Autônoma (Agentes)

Prof. Aléssio Miranda Júnior
alessio@cefetmg.br

CEFET-MG - Campus Timoteo
Dep. Engenharia de Computação

Inteligência Artificial 2

Sumário

Onde Paramos?

Na aula anterior, fizemos o **Fine-Tuning de um LLM com LoRA**:

- Adaptadores de baixo rank ($r = 16$) em cima do Llama-3 quantizado 4-bit.
- Curva de Loss decrescente = prova empírica de aprendizado.
- Inferência com o modelo adaptado no Colab gratuito.

A Pergunta de Hoje

O LLM gera texto, mas é **passivo**. E se ele pudesse **agir no mundo real** — consultar APIs, buscar dados, executar código?

Objetivos da Sessão

Ao final desta aula, você será capaz de:

1. **Diferenciar** LLMs passivos de Agentes Autônomos.
2. **Descrever** o loop ReAct: Thought $\rightarrow$ Action $\rightarrow$ Observation.
3. **Implementar** um Agente com `@tool` no LangChain.
4. **Explicar** Function Calling e como o LLM "escolhe" a ferramenta.
5. **Avaliar** riscos: o que acontece quando o Agente erra?

O Paradigma do Oráculo Passivo

Historicamente, a interação humano-IA seguiu um modelo de **resposta a estímulo**:

- O usuário fornece um *prompt* textual.
- O LLM (*Large Language Model*) realiza a *forward pass* gerando *tokens* autoregressivos.
- A interação se encerra. O modelo retorna ao estado inerte.

Por que isso é insuficiente? Este modelo pressupõe que *toda* a informação necessária para a tarefa ou está embutida nos pesos (o que sofre de *Knowledge Cutoff*) ou foi injetada no *prompt* (RAG).

Deficiências Inerentes dos LLMs Puros

Quando operam sem extensões, modelos de linguagem tropeçam em tarefas fundamentais:

1. **Aritmética Exata:** LLMs não "calculam", eles prevêem o próximo *token*. A multiplicação 3459×8123 raramente estava no *dataset* de treino, gerando alucinações numéricas com alta confiança.
2. **Amnésia Temporal:** O modelo não possui percepção do "agora". Perguntar a cotação do dólar atual é impossível para pesos congelados há meses.
3. **Acesso a Sistemas Fechados:** A IA não pode ler o seu e-mail do Gmail, acessar seu banco de dados SQL privado ou rodar comandos no terminal do seu servidor web apenas prevendo palavras.

O Salto de Paradigma: Modelos como "Raciocinadores"

A verdadeira revolução na Inteligência Artificial não é tornar os LLMs enciclopédias maiores, mas sim **motores de raciocínio** (*Reasoning Engines*).

> *"Se um LLM não sabe calcular uma raiz quadrada, mas sabe gerar código Python para calcular, por que não deixá-lo executar esse código e ler o resultado?"*

Isso dá origem aos **Agentes Autônomos**: sistemas onde o LLM atua como o córtex

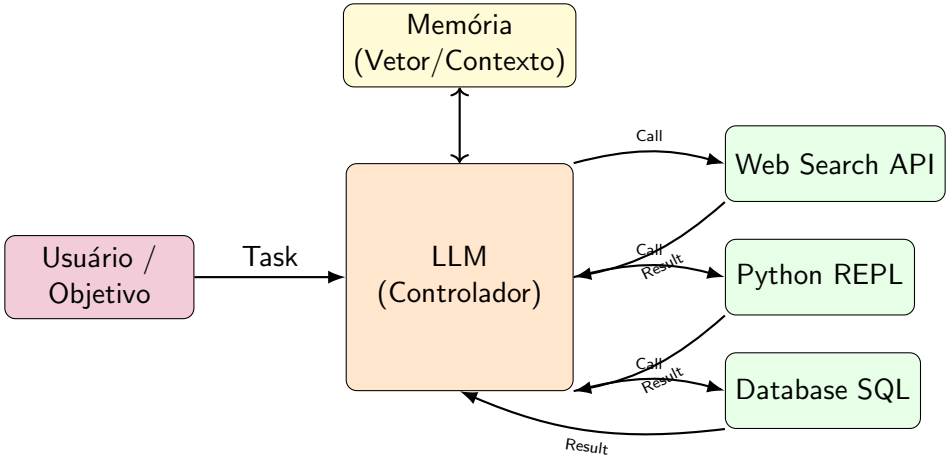
pré-frontal, delegando a execução de memória de curto prazo, cálculos e interações com a rede para *ferramentas* externas.

Componentes de um Agente

De acordo com o *survey* de Weng (2023) sobre *LLM-powered Autonomous Agents*, uma arquitetura robusta engloba:

- **Perfil (Persona):** O *System Prompt* que dita as regras operacionais.
- **Memória:**
  - *Curto Prazo:* A janela de contexto (*chat history*).
  - *Longo Prazo:* Banco de Dados Vetorial, armazenando experiências passadas.
- **Planejamento:** Quebra de tarefas complexas em sub-tarefas (*Task Decomposition*) e auto-reflexão sobre erros (*Self-Correction*).
- **Ação (Tools):** Capacidade de invocar APIs (Clima, Buscas web, Python REPL, SQL execution).

Arquitetura Visual



Raciocínio + Ação (ReAct)

Proposto no *paper* "ReAct: Synergizing Reasoning and Acting in Language Models" (Yao et al., 2022). O *framework* força o modelo a externalizar seus pensamentos e intercalá-los com interações no ambiente:

- **Chain of Thought (CoT):** Ajuda o modelo a resolver problemas (apenas *Reasoning*), mas sofre de isolamento ambiental e alucinações irremediáveis.
- **Acting-only:** Toma ações cegas sem plano de longo prazo.
- **ReAct:** Combina ambos. O modelo rastreia, atualiza planos e lida com exceções quando uma ação externa falha.

O Loop de Pensamento e Ação

A estrutura canônica de um Agente ReAct segue o loop ininterrupto:

1. **Thought (Pensamento):** O modelo analisa o estado atual e decide a próxima jogada.
2. **Action (Ação):** Invoca uma ferramenta específica e fornece os parâmetros.
3. **PAUSA:** A geração de *tokens* é artificialmente interrompida pela sua aplicação base.
4. **Observation (Observação):** Seu código Python executa a ferramenta, captura o retorno textual e injeta de volta no *prompt* do LLM.

O *loop* reinicia (Thought $\rightarrow$ Action $\rightarrow$ Observation) até o modelo atingir a condição de parada, gerando a **Final Answer**.

O Loop ReAct na Prática (Exemplo Textual)

```
Usuário: Qual a raiz quadrada da população atual de Paris?

Thought 1: Preciso saber a população atual de Paris. Vou buscar na
    web.
Action 1: WebSearch(query="população atual Paris 2024")
Observation 1: Cerca de 2.102.650 habitantes (censo oficial est.
    2024).

Thought 2: Agora eu sei a população. Vou calcular a raiz quadrada de
     2102650 usando a calculadora, pois é um número complexo.
Action 2: Calculator(expression="sqrt(2102650)")
Observation 2: 1449.81

Thought 3: Tenho todos os dados. Posso responder ao usuário.
Final Answer: A raiz quadrada da população atual de Paris é
    aproximadamente 1.449,81.
```

Function Calling na API

Como traduzimos esse *prompt* abstrato para os modelos de ponta modernos (GPT-4, Claude 3, Llama 3)? O mecanismo nativo chama-se **Function Calling** (ou *Tool Use*).

- Quando inicializamos a chamada à API do LLM, incluímos não apenas o histórico de mensagens, mas também um *Array* de **Ferramentas disponíveis**.
- Essas ferramentas são enviadas na forma de esquemas JSON (*JSON Schema*) declarando o nome da função, a descrição de seu propósito, e a tipagem exata dos argumentos exigidos.

Definindo o Schema JSON da Ferramenta

O modelo não lê seu código em Python. Ele só entende o "contrato" JSON:

```
{
  "type": "function",
  "function": {
    "name": "obter_clima_atual",
    "description": "Obtém o clima atual para uma localidade",
    "parameters": {
      "type": "object",
      "properties": {
        "cidade": {
          "type": "string",
          "description": "O nome da cidade. Ex: Rio de Janeiro"
        }
      },
      "required": ["cidade"]
    }
  }
}
```

Interceptação e Execução Local

Quando a API recebe a requisição com o contrato de funções, e conclui que a ação é necessária:

1. O LLM retorna uma resposta especial contendo o atributo `tool_calls`.
2. Esta mensagem não possui conteúdo de texto para o usuário, apenas metadados (ex: `function="obter_clima_atual"`, `arguments="{'cidade':'Timóteo'}"`).
3. **Seu Código Python:** Deve analisar essa resposta, invocar a biblioteca `requests` na API do Clima, receber a resposta HTTP 200, transformar em *String*.
4. Você re-acessa a API do LLM anexando uma mensagem do tipo `role="tool"` (Observação) contendo os dados.
5. O LLM então gera o texto legível.

Por que usar um Framework (LangChain)?

O gerenciamento manual de *Function Calling* exige *boilersplates* tediosos:

- Serializar esquemas JSON manualmente.
- Escrever blocos gigantes de `if/else` mapeando nomes de *strings* para funções de código real.
- Lidar com erros: e se a API externa cair? E se o modelo errar o nome do parâmetro? O código irá quebrar.
- Gerenciar o limite numérico da janela de contexto.

O **LangChain** abstrai esse roteamento, conectando a abstração nativa do Python com a chamada de rede.

O Decorator @tool

No LangChain, você escreve funções Python normais. O segredo está nas **Type Hints** (*typing* do Python) e nas **Docstrings**, que o LangChain compila automaticamente para *JSON Schema*!

```python
from langchain.tools import tool
import requests

@tool
def buscar_clima(cidade: str) -> str:
    """Retorna o clima atual de uma cidade em graus Celsius."""
    url = f"https://api.clima.com?q={cidade}"
    resposta = requests.get(url).json()
    return f"A temperatura e {resposta['temp']} graus."

# O LangChain abstraiu todo o contrato JSON daqui!
ferramentas = [buscar_clima]
```

Instanciando o Maestro: AgentExecutor

Para que o *Loop ReAct* ocorra automaticamente:

```
from langchain.agents import create_tool_calling_agent,
    AgentExecutor
from langchain_openai import ChatOpenAI
from langchain_core.prompts import ChatPromptTemplate

llm = ChatOpenAI(model="gpt-4o")

prompt = ChatPromptTemplate.from_messages([
    ("system", "Voce e um assistente meteorologico prestativo."),
    ("human", "{input}"),
    ("placeholder", "{agent_scratchpad}") # Onde o ReAct anota o
        rascunho
])

# Cria o cerebro e liga com as ferramentas
agente = create_tool_calling_agent(llm, ferramentas, prompt)
```

Executando o Agente

Ao invocar o `AgentExecutor`, o *framework* roda o loop silenciosamente:

```
resposta = executor.invoke({"input": "Qual o clima em Ipatinga hoje?"})
print(resposta["output"])
```

Saída com `verbose=True`:

- `> Entering new AgentExecutor chain...`
- `Invoking: buscar_clima with {'cidade': 'Ipatinga'}`
- `Observation: A temperatura e 32 graus.`
- `Resuming reasoning...`
- `> Finished chain.`
- "A temperatura em Ipatinga hoje está em cerca de 32 graus."

Considerações de Segurança

Agentes Autônomos trazem **riscos imensos** (*Risk Surface*):

- **Prompt Injection em Ferramentas:** Um atacante pode colocar comandos maliciosos no nome de uma cidade para explorar vulnerabilidades do *Backend*.
- Se o agente possuir acesso a uma ferramenta `ExecutarSQL()`, o modelo pode ser induzido (por alucinação ou injeção) a executar `DROP TABLE`.
- **Solução:** Agentes nunca devem operar ferramentas com impacto no mundo real (como compras financeiras ou exclusão de dados) sem um *Human-in-the-Loop* (HITL). A arquitetura deve forçar uma aprovação manual para *Side-Effects*.

Orquestração Avançada e Graph-Based Agents

Conforme os Agentes crescem, a execução puramente linear (Chain) torna-se instável para tomada de decisão complexa. **LangGraph e Máquinas de Estado:**

- Uma evolução do LangChain que modela o fluxo como um Grafo Direcionado.
- Os nós (*Nodes*) são funções Python independentes (ex: Pesquisar Google, Avaliar Qualidade).
- As arestas (*Edges*) são roteamentos condicionais. Se o nó "Avaliador" definir que a pesquisa foi ruim, a aresta faz um *loop* de volta para a Pesquisa.
- Isso permite agentes que corrigem seus próprios erros autonomamente antes de entregar a resposta final ao cliente.

O Custo de Agentes em Produção

A automação total tem um preço de infraestrutura:

- **Explosão de Tokens:** O raciocínio *ReAct* exige que a cada *Loop*, o histórico de todo o pensamento passado seja re-enviado à API. Um ciclo de 5 pensamentos consome 5 vezes mais tokens de entrada.
- **Latência Inviável para UI:** Uma chamada direta ao GPT demora 2 segundos. Um agente executando buscas e loops pode demorar 25 segundos, quebrando a experiência de chat do usuário.
- **Loops Infinitos:** Sem uma restrição estrita (*Max Iterations*), o agente pode ficar tentando executar ferramentas falhas eternamente.

Resumo da Era dos Agentes

A Inteligência Artificial passou de "Recuperação de Informação" (Busca, RAG) para "Delegação de Tarefas".

- Os **Agentes Autônomos** representam o Santo Graal atual da IA Comercial.
- Eles transcendem os limites de treinamento do LLM injetando contexto exato via execução computacional.
- Ferramentas como **LangChain** ou seu sucessor focado em grafos, **LangGraph**, orquestram os intrincados *loops* do ReAct, mantendo o código Python limpo e resiliente.

Laboratório 24: Agente Multi-Ferramentas

Usaremos o Python para construir um Agente com o LangChain.
Sua missão será criar funções em Python que façam desde simples cálculos de raiz quadrada, até requisições via internet para descobrir detalhes recentes. Em seguida, veremos o LLM rotear as perguntas perfeitamente entre essas funções.