Natural Language Processing

Class 12: Augmented LLMs: Self-reflection, retrieval-augmented generation, and the Agent framework

Adam Faulkner

November 25, 2025

Arithmetic: An early problem for LLMs

- Interactions with early LLMs (2020-2022) quickly surfaced a surprising gap in their understanding: they struggled to perform basic mathematical and commonsense reasoning tasks. Given 31398 + 47271, GPT3's answer, circa 2022, according to Qian et al. 2022, was 7866.

Arithmetic: An early problem for LLMs

- This is a problem if one wants to maintain that LLMs “learn” complex reasoning abilities during pre-training.
- The most basic feature of genuine learning is *induction*, or moving from the particular to general case via learned rules. What is 31398 + 47271? Even though you’ve never seen these two particular numbers, you’ve learned a set of rules that allow you to move from the particular to the general case - in this case utilizing the learned rules of addition to add any two numbers.
- Clearly, early LLMs hadn’t learned arithmetic in this way.

Commonsense reasoning was also brittle in early LLMs

- Commonsense reasoning also seems to have been imperfectly learned during pre-training
  - **Question**: "What happens if you fire a cannonball directly at a pumpkin at high speeds?"
  - **Answer (GPT3-175B)**: "The pumpkin will pull the cannonball in, and the cannonball will bounce off of the pumpkin. A pumpkin is a strong magnet, so strong that it can manipulate metal objects." (From Ouyang et al., 2022)

Solving reasoning tasks via self-reflection

- When diagnosing these gaps, researchers noted performance gains on these sorts of tasks could be obtained by simply expanding the prompt with answer preambles such as, “Let’s think through this step by step”
- Encouraging the LLM to explicitly verbalize the kind of reasoning helps the LLM arrive at the correct answer
- This simple idea represented a breakthrough in solving reasoning-based problems and led to a flurry of research in prompt-based reasoning via “self-reflection”

Self-reflection: Encouraging LLMs to think out loud

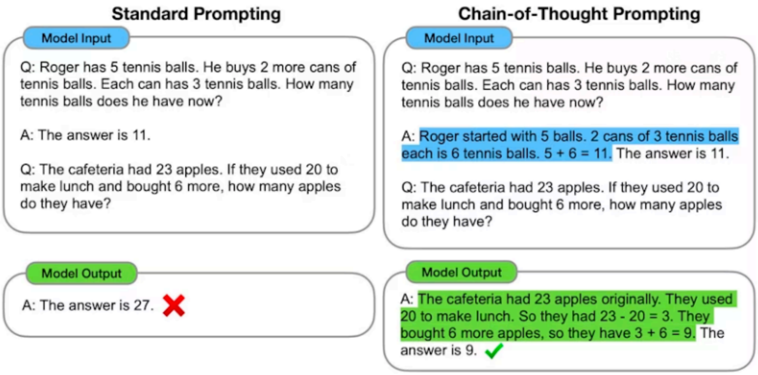
- Auditory learners are often encouraged to "think out loud" when studying complex content. Information retention improves markedly for these learners when they vocalize the concepts they are learning.
- Similarly, self-reflection involves encouraging the LLM to think out loud while solving a task
- Intuition: When an LLM is prompted to verbally reflect on each step of a complex reasoning task, it becomes better at completing complex tasks since, at each step, it conditions on the previous steps

Self-reflection: Encouraging LLMs to think out loud

- For example, LLMs can "think out loud" to work through each step of the task in much the same way that humans do when calculating, say, the lowest common denominator of two numbers: “First I need to find all of the multiples of these two numbers. Then, for each multiple, I’ll check to see if the other number also has that multiple...“ and so on.

Chain-of-thought prompting

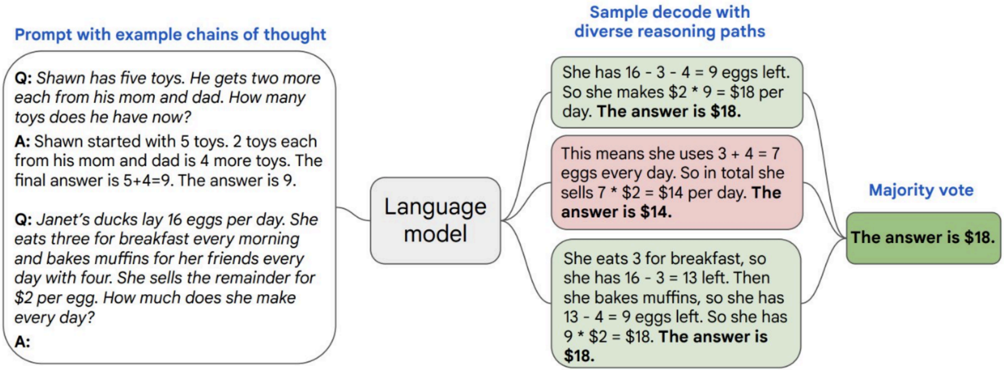
- Chain-of-thought (CoT) is the most basic form of self-reflection
- Simply adding “Let’s think through this step-by-step” to the initial task instruction allowed LLMs to solve fairly complex examples of mathematical reasoning



Expanding CoT prompting with Self-Consistency

- Multiple LLM next-token completions are sampled, and a final answer is chosen based on its consistency across samples.
- This token sampling is performed via *sampling decoding* which introduces variability in the LLM output by randomly sampling the next token from a probability distribution, often utilizing a temperature parameter to control for randomness
- Changing the temperature encourages the model to explore and vary its output.
- By stepping through a range of temperature values, one can generate multiple varied responses for a single prompt

Expanding CoT prompting with Self-Consistency



Least-to-most prompting

- Many complex word problems can be solved by breaking them down into a sequence of simpler sub-problems. Take this question for example:
  - ”Four years ago, Kody was only half as old as Mohamed. If Mohamed is currently twice 30 years old, how old is Kody?”
  - This problem can be broken down into sub-problems, which are solved one at a time.

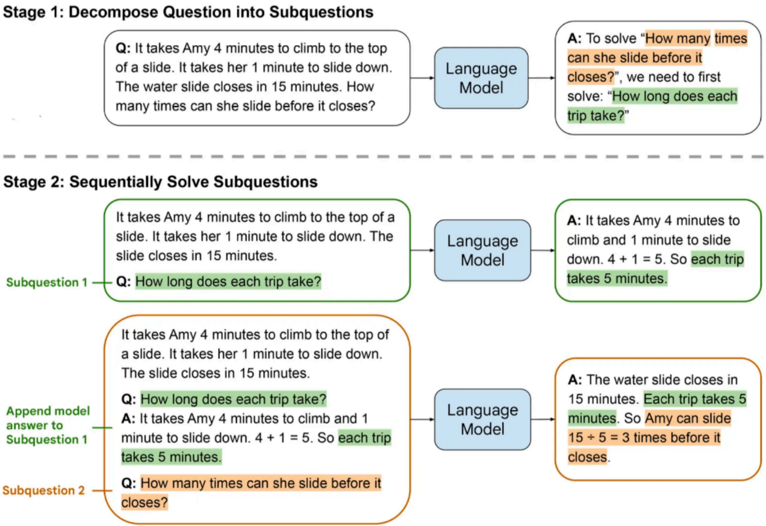
  1. “How old was Mohamed four years ago if hes currently twice 30 years old?” (56)
  2. “How old was Kody four years ago if he was only half as old as Mohamed, who was 56?” (28)

- The solutions to these sub-problems are then used to solve the whole problem.
  - “Kody was 28 four years ago, so hes 32 today.”

Least-to-most prompting

- In Least-to-most (LTM) prompting a prompt demonstrating how a problem can be reduced to a series of sub-questions (aka question decomposition) is added to the context
- Then, the target question is asked, and the LLM is prompted to decompose the question into subquestions. The loop continues until all sub-questions are exhausted.
- Finally, the LLM is prompted to solve the initial problem, with all of its sub-questions and answers included in the context window.

Least-to-most prompting



Adding Actions and Tools to Self-Reflection

- Self-reflection leverages the power of “thinking aloud” to answer questions. While this is sufficient for the math problems discussed in the previous slides, it reaches a limit when the correct answer cannot come from reasoning or innate knowledge alone.
- Suppose we are building an application that returns the current weather to the user. The LLM clearly can’t tell what the current weather is based on its training, so well need to somehow access a weather API and incorporate the response into the context.

Adding Actions and Tools to Self-Reflection

- By creating an LLM-based application with tools, such as network APIs or python functions, we enable the LLM to perform more useful tasks including reading external data or making changes to an environment.
- Tools allow us to use LLMs to carry out tasks such as booking travel, making payments or reservations, and many others.

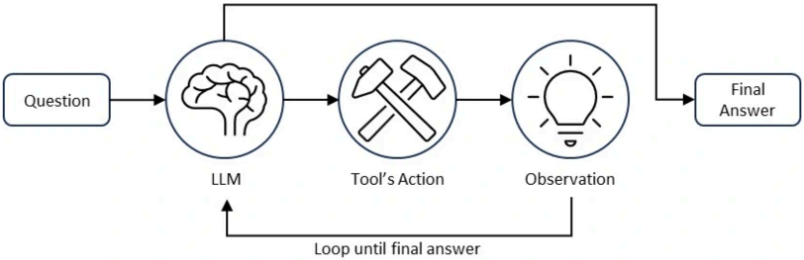
ReAct (Reason + Act

- When we call such tools, we are performing an **action** and subsequently **reasoning** over the results of these actions
- Reasoning and acting using tools are combined in the ReAct (Reason + Act) framework (Yao et al. 2023), a prompt-based framework that performs reasoning via thoughts, actions, and observations, allowing it to solve problems and create action plans.

ReAct (Reason + Act

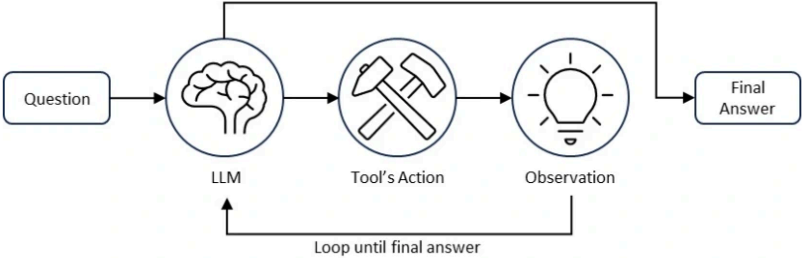
ReAct-style prompting occurs iteratively, utilizing the following elements:

- **Actions**: These are interactions with an environment and usually occur via tools. A tool in this context could be the Google Search API and action would be calling the API with a search query.



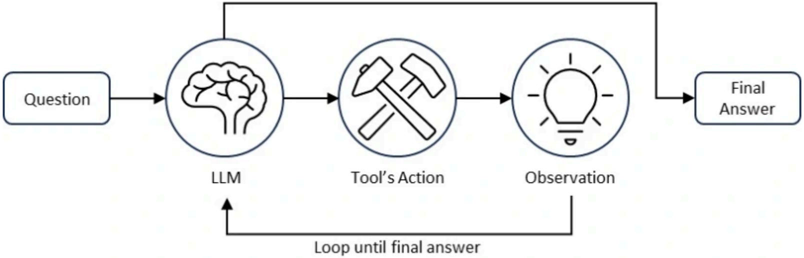
ReAct (Reason + Act

- **Observations**: These are the responses received from the environment after some action. If the action is an internet search, then the text returned by the search would constitute the observation.



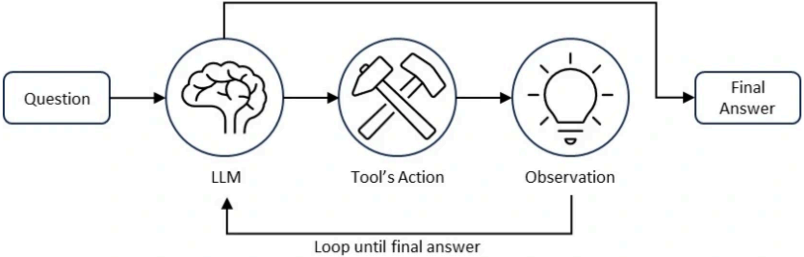
ReAct (Reason + Act

- **Thoughts:** These are analogous to the chain-of-thought-style self-talk discussed above. They use the output of previous actions and observations to plan further actions.



ReAct (Reason + Act

- To see why this might be useful, consider a task involving answering an obscure question regarding an Apple product. If the LLM hasn't seen the info to answer this question in pre-training it can utilize ReAct prompting and an internet search tool (dubbed `Search()`) to update the context with observations returned from the Google Search API



ReAct (Reason + Act

Example using a tool `Search()` that makes a call to the google API to collect information to answer a question

(1) Hotspot QA

Question: Aside from the Apple Remote, what other device can control the program Apple Remote was originally designed to interact with?

(1d) ReAct (Reason + Act)

Thought 1: I need to search Apple Remote and find the program it was originally designed to interact with.
Act 1: Search[Apple Remote]
Obs 1: The Apple Remote is a remote control introduced in October 2005 by Apple … originally designed to control the Front Row media center program …

Thought 2: Apple Remote was originally designed to control the Front Row media center program. I need to search Front Row next and find what other device can control it.
Act 2: Search[Front Row]
Obs 2: Could not find [Front Row]. Similar: ['Front Row Seat to Earth', 'Front Row Motorsports',' Front Row (software) ', …]

Thought 3: Front Row is not found. I need to search Front Row (software) .
Act 3: Search[Front Row (software)]
Obs 3: Front Row is a discontinued media center software …

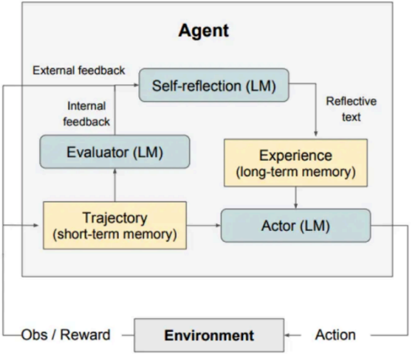
Thought 4: Front Row (software) is controlled by an Apple Remote or the keyboard function keys. So the answer is keyboard function keys.
Act 4: Finish[keyboard function keys]

ReAct + In-Context Reinforcement Learning = Reflexion

- The **Reflexion** self-reflection framework combines ReAct style self-reflection with feedback via reinforcement learning (RL), all within the LLM's context window.
- At first glance, the idea seems counterintuitive: A typical RL setting for language models occurs at training time and involves scalar values and gradient descent as part of the optimizations procedure.
- In contrast, an alternative RL method is implemented by converting natural language feedback produced by the environment into a format compatible with the LLMs context window. Specifically, the feedback is converted into a textual summary and added to the context.
- The text version of the feedback is incorporated into the context window and informs the next round of self-reflection

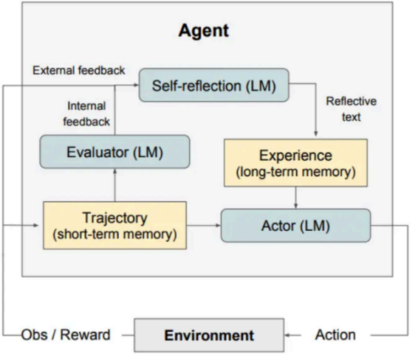
ReAct + In-Context Reinforcement Learning = Reflexion

Each of these steps requires a separate LLM: an Actor LLM, an Evaluator LLM, and a Self-Reflection LLM:



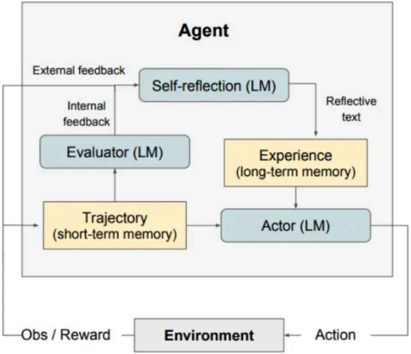
ReAct + In-Context Reinforcement Learning = Reflexion

- **Actor**: This is the core LLM in the framework and generates the thoughts and actions (using ReAct or COT) conditioned on the feedback generated by the Evaluator.



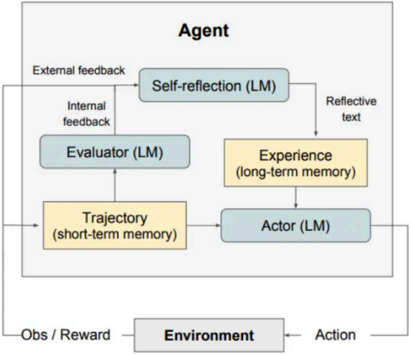
ReAct + In-Context Reinforcement Learning = Reflexion

- **Evaluator:** This is an LLM that, depending on the task, returns feedback to the Actor. The Actor incorporates this feedback as a reward signal as it progresses toward the solution to the task.



ReAct + In-Context Reinforcement Learning = Reflexion

- **Self-Reflection:** The Self-Reflection LLM takes the Evaluator's reward signal along with the current context and generates specific feedback regarding the proposed solution and, potentially, suggestions for corrective action.



ReAct + In-Context Reinforcement Learning = Reflexion

Context: The Battle of White Plains was a battle in the New York and New Jersey campaign of the American Revolutionary War fought on October 28, 1776, near White Plains, New York. Following the retreat ...

Trial #1: Question: What was a series of battles during the Revolutionary War, for control of New York City and the state of New Jersey, fought on October 28, 1776 near White Plains, New York?
Thought 1: The question is asking about a series of battles during the Revolutionary War for control of New York City and the state of New Jersey, fought on October 28, 1776 near White Plains, New York. From the context, I know that this was the Battle of White Plains.
Action 1: Finish[Battle of White Plains]
Observation 1: Answer is INCORRECT

Trial #2: Question: What was a series of battles during the Revolutionary War, for control of New York City and the state of New Jersey, fought on October 28, 1776 near White Plains, New York?
Thought 1: The question is asking for a series of battles during the Revolutionary War, for control of New York City and the state of New Jersey, fought on October 28, 1776 near White Plains, New York. Based on the context, I know that this series of battles was the New York and New Jersey campaign.
Action 1: Finish[The New York and New Jersey campaign]
Observation 1: Answer is CORRECT

Reflection: Upon reflecting on the incorrect answer I provided, I realize that I may not have provided enough context to accurately answer the question. The question asked for a series of battles, but I only provided the name of one battle. To better answer this question in the future, I will make sure to provide more context, such as the name of the campaign, which was the New York and New Jersey campaign, and the fact that it was a series of battles. Additionally, I will make sure to include the dates of the battles, as well as the locations, to ensure that my answer is comprehensive and accurate.

Reframing search as a question-answering task

- Question answering systems are designed to fill human information needs
- Since a lot of information is present in text form (on the web or in other data like our email, or books), question answering is closely related to the task behind search engines
- The distinction is becoming ever more fuzzy, as modern search engines are integrated with LLMs trained to do question answering

Factoid questions

- Question answering systems often focus on a useful subset of information needs: **factoid questions,** questions of fact or reasoning that can be answered with simple facts expressed in short or medium-length texts, like the following:
  1. Where is the Louvre Museum located?
  2. Where does the energy in a nuclear explosion come from?
  3. How to get a script l in latex?

Factoid questions

- Modern NLP systems answer these questions using LLM, in one of two ways. The first is prompt-based: we prompt a pretrained and instruction-tuned LLM, an LLM that has been finetuned on question/answer datasets with the question in the prompt.
- For example, we could prompt a causal language model with a string like

  `Q: Where is the Louvre Museum located?  A:`

  and have it do conditional generation given this prefix, and take the response as the answer.
- The idea is that language models have read a lot of facts in their pretraining data, presumably including the location of the Louvre, and have encoded this information in their parameters.

Problems with prompt-based approaches to QA

- There are few issues with the purely prompt-based approach
- The first is that LLM's hallucinate. A hallucination is a response that is not faithful to the facts of the world, i.e., when asked questions, LLMs simply make up answers that sound reasonable.
- To get an idea of the scope of the hallucination problem, consider Dahl et al. (2024), who found that when asked questions about the legal domain (particular legal cases, etc.), LLMs hallucinated from 69% to 88% of the time

Problems with prompt-based approaches to QA

- A second problem is that simply prompting an LLM doesnt allow us to ask questions about **proprietary data**
- A common use of question answering concerns data such as
  - personal emails or medical record
  - a company's internal documents, which contain answers for customer service or internal use-cases
  - or, in the case of legal firms asking questions about legal discovery, proprietary legal documents

Problems with prompt-based approaches to QA

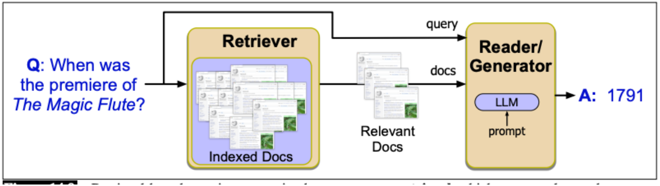
- The most common way to do question-answering with LLMs is RAG
- In RAG we use information retrieval (IR) techniques to retrieve information retrieval documents that are likely to have information that might help answer the question.
- Then we use an LLM to generate an answer given these documents

Advantages of RAG

- Basing our answers on retrieved documents can solve some of the problems with using simple prompting to answer questions.
- First, it helps ensure that the answer is grounded in facts from some curated dataset.
- And the system can give the user the answer accompanied by the context of the passage or document the answer came from.
- This information ensures that users have confidence in the accuracy of the answer (or help them spot when it is wrong)
- These retrieval techniques can be used on any proprietary data we want, such as legal or medical data for those applications.

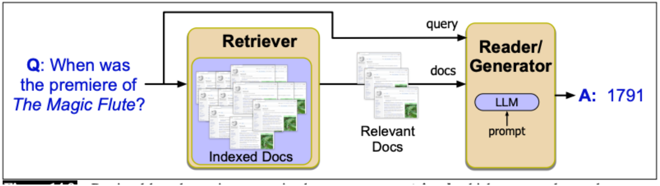
The RAG architecture

In RAG, we first **retrieve** relevant passages from a text collection, for example using a semantic search library such as FAISS.



The RAG architecture

In the second **reader** stage, we generate the answer via **retrieval-augmented generation**. In this method, we take a large pretrained language model, give it the set of retrieved passages and other text as its prompt, and autoregressively generate a new answer token by token.



The RAG architecture

Suppose we have a question such as

`Q: Who wrote the book "The Origin of Species"?  A:`

Rather than just conditioning on this text to generate the answer, RAG allows us to condition on the retrieved passages as part of the prefix, perhaps with some prompt text such as "Based on these texts, answer this question:"

Schematic of a RAG Prompt

```
retrieved passage 1

retrieved passage 2

...

retrieved passage n

Based on these texts, answer this question:  Q: Who wrote
the book ‘‘The Origin of Species"?  A:
```

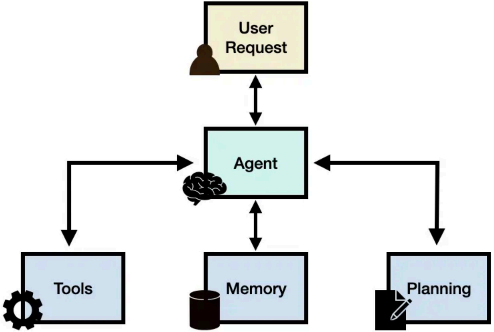
Planning + memory + tools = Agents

- So far, we've learned about basic inference-time planning strategies such as self-reflection
- We also learned about *tools*, which enable the LLM to perform useful things in the external environment such as accessing a local database or searching on the internet
- The Agent paradigm is an extension of these basic components: agents can execute complex tasks through the use of an architecture that combines LLMs with modules such as planning, memory, and tools
- The LLM serves as the main controller or "brain" that controls a flow of operations needed to complete a task or user request.

Planning + memory + tools = Agents

An agent usually consists of the following core components:

- **User Request:** a user question or request
- **Agent/Brain**: the agent core acting as coordinator
- **Planning:** assists the agent in planning future actions
- **Memory:** manages the agent's past behaviors



Why do we need agents?

- Suppose the LLM is given the following question:
  - `What's the average daily calorie intake for 2023 in the United States?`
- The question above could potentially be answered using an LLM that already has the knowledge needed to answer the question directly.
- If 2023 data doesn't exist we could use RAG to access health related information or reports.

Why do we need agents?

- But what if ask the LLM something more complicated, something that involves multiple steps and, potentially, interaction with the external environment?
  - `How has the trend in the average daily calorie intake among adults changed over the last decade in the United States, and what impact might this have on obesity rates? Additionally, can you provide a graphical representation of the trend in obesity rates over this period?`

Why do we need agents?

- To answer such a question, just using an LLM alone wouldn't be enough.
- You can combine the LLM with an external knowledge base to form a RAG system but this is still probably not enough to answer the complex query above.
- This is because the complex question above requires an LLM to break the task into subparts which can be addressed using tools and a flow of operations that leads to a desired final response.

Why do we need agents?

The basic components needed to answer this question include

1. A step-by-step plan to answer the question
2. A memory module that helps the agent keep track of the state of the flow of operations, observations, and overall progress.
3. Access to a search API, health-related publications, and public/private health database to provide relevant information related to calorie intake and obesity.
4. Access to a code interpreter tool that helps take relevant data to produce useful charts (via matplotlib or something similar) that help to visualize trends in obesity.

The Agent component

- A pretrained LLM serves as the main brain, agent module, or coordinator of the system. This component will be activated using a prompt template that entails important details about how the agent will operate, and the tools it will have access to (along with tool details).
- An agent can also be profiled or be assigned a persona to define its role. This profiling information is typically written in the prompt which can include specific details like role details, personality, social information, and other demographic information.

The Planning component

- Self-reflection paradigms such as CoT plan without feedback: they break down the necessary steps or subtasks the agent will solve individually to answer the user request.
- But they don't incorporate feedback, either from another LLM or from the user, and so cannot engage in the trial-and-error often required to solve complex tasks.
- More sophisticated planning paradigms such as ReAct incorporate feedback in the form of *observations*
- Other types of feedback can include human and model feedback.

The Memory component

- The memory module helps to store the agent's internal logs including past thoughts, actions, and observations from the environment, as well as all interactions between agent and user. There are two main memory types that have been reported in the LLM agent literature:
  1. **Short-term memory:** includes context information about the agent's current situations; this is typically realized by in-context learning which means it is short and finite due to context window constraints.
  2. **Long-term memory** includes the agent's past behaviors and thoughts that need to be retained and recalled over an extended period of time; this often leverages an external vector store accessible through fast and scalable retrieval to provide relevant information for the agent as needed.

The Memory component

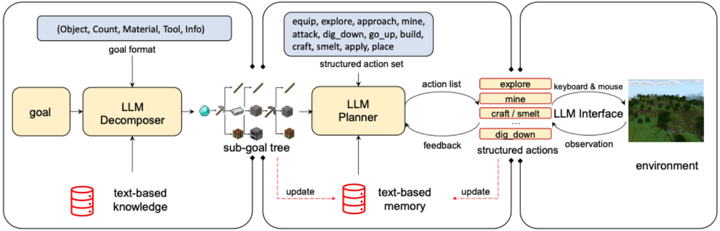
- **Hybrid memory** integrates both short-term memory and long-term memory to improve an agent's ability for long-range reasoning and accumulation of experiences.
- There are also different memory formats to consider when building agents. Representative memory formats include natural language, embeddings, databases, and structured lists, among others.

The Memory component example: Ghost in the Minecraft (GITM)

- GITM utilizes a key-value structure where the keys are represented by natural language and values are represented by embedding vectors
- GITM is an approach to the AI task of building intelligent agents capable of functioning in open-world environments (we'll be looking at this area again in a few weeks)
- Both the planning and memory modules allow the agent to operate in a dynamic environment and enable it to effectively recall past behaviors and plan future actions.

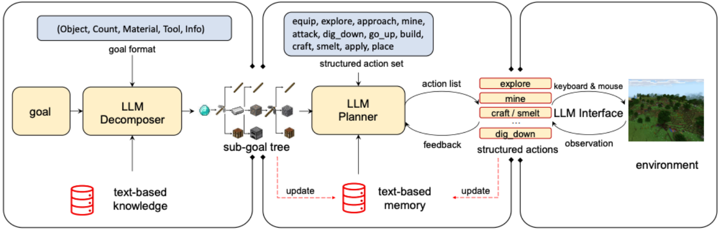
The Memory component example: Ghost in the Minecraft (GITM)

GITM consists of an LLM Decomposer, an LLM Planner, and an LLM Interface



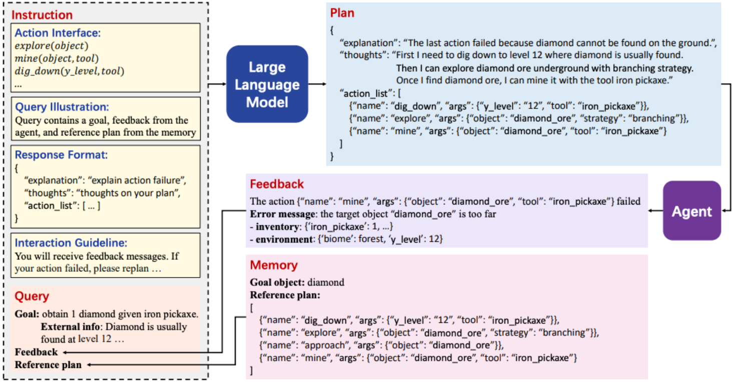
The Memory component example: Ghost in the Minecraft (GITM)

Given a Minecraft goal, the LLM Decomposer divides the goal into a sub-goal tree. The LLM Planner then plans an action sequence for each sub-goal. Finally, the LLM Interface executes each action in the environment. The agent can be further enhanced by leveraging text-based knowledge and memory.



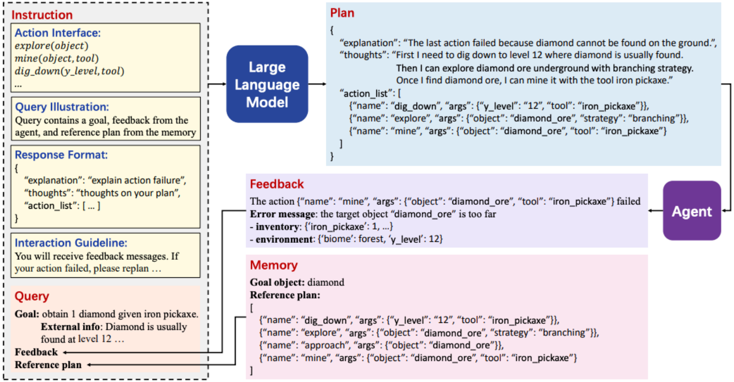
The Memory component example: Ghost in the Minecraft (GITM)

During each game episode, once the goal is achieved, the entirely executed action list is stored in memory.



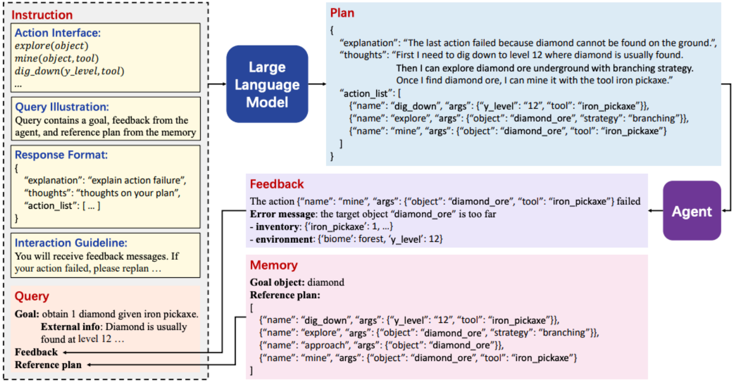
The Memory component example: Ghost in the Minecraft (GITM)

The LLM may achieve the same goal under various circumstances, resulting in a range of different plans. To identify a common reference plan suitable for general situations, essential actions from multiple plans are summarized.



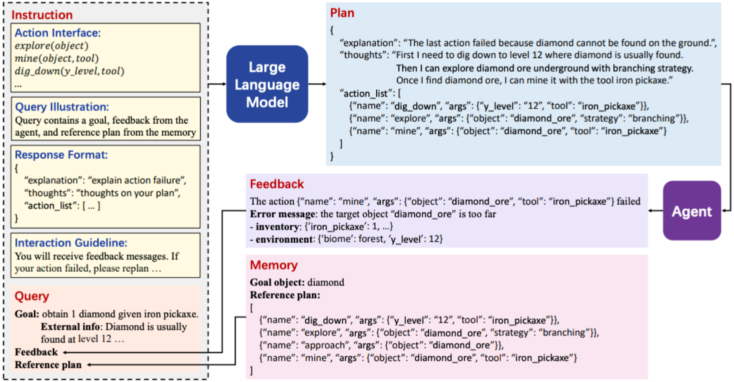
The Memory component example: Ghost in the Minecraft (GITM)

When encountering similar goals, the LLM creates new plans based on the summarized reference plans retrieved from memory (retrieval occurs via semantic search against the embedded summaries).



The Memory component example: Ghost in the Minecraft (GITM)

Successful action sequences from these new plans are also added to memory for future summarization. As the LLM-based Planner accumulates summaries, it becomes increasingly effective.



The prompt used to implement the memory component in GITM

You serve as an assistant that helps me play the game Minecraft.

I am using a set of actions to achieve goals in the game Minecraft. I have recorded several action sequences successfully achieving a goal in a certain state. I will give you the goal, the state, and the sequences later. Please summarize the multiple action sequences into a single action sequence as a universal reference to achieve the goal given that certain state. Here are the instructions:

1. Each action sequence is a sequence of the following actions:

`{action description}`

2. The action sequences before and after summarization are always conditioned on the given state, i.e., the actions are taken in that certain state to achieve the goal. I will describe the state in the following form: State: - inventory: a dict whose keys are the name of the objects and the values are their quantities. This inventory stores all the objects I have. - environment: a dict including my surrounding biome and whether I am on the ground or underground.

3. The action sequence you summarize should be able to achieve the goal in general cases without specific modification. Every necessary action should be included, even though it does not appear in some sequences because I manually skipped it in some lucky cases. The actions redundant or irrelevant to the goal should be filtered out. The corner cases, such as success by luck and dealing with contingencies, should not be summarized into the final sequence.

4. You should describe your thoughts on summarization in natural language at the beginning. After that, give me the summarized action sequence as a list in JSON format. Your response should follow this form:

Thoughts: "Your thoughts and descriptions of your summarization"
Summarized action sequence:
[
{"name": "action name", "args": {"arg name": value}, "expectation": "describe the expected results of this action"},
{"name": "action name", "args": {"arg name": value}, "expectation": "describe the expected results of this action"},
{"name": "action name", "args": {"arg name": value}, "expectation": "describe the expected results of this action"}
]

Why we need tools

- Tools enable LLM agents to interact with the external environment. They could include the Wikipedia Search API, a Python code interpreter, or a calculator
- Tools could also include databases, knowledge bases, and external models.
- When the agent interacts with external tools it executes tasks via workflows that assist the agent to obtain observations or necessary information to complete subtasks and satisfy the user request.
- In our initial health-related query, a code interpreter is an example of a tool that executes code and generates the necessary chart information requested by the user.

Tool retrieval: Tool retrieval using RAG

- How does the LLM know when to invoke a tool?
- **Tool retrieval using RAG:** Identify and retrieve the most relevant tools or information needed to complete a task by searching through a knowledge base or database based on a user's query

Tool retrieval: Tool retrieval using RAG

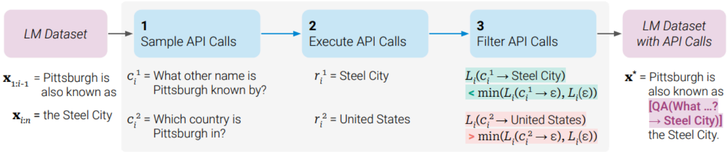
- For example, a user might ask about their current balance: `What is my current balance?`
- We have a vector store of tools such as `make_payment()` and `get_balance()`. We also have descriptions associated with each tool,e.g, "The `get_balance()` tool is called when the customer is asking about their current balance" ,etc.
- We perform semantic-search, using the query `What is my current balance?`, against the vector store of tool descriptions.
- The top-ranked tool is returned

Tool retrieval: Finetuning-based approach

- Rather than relying on RAG or other inference-time frameworks to retrieve tools, we can also teach the LLM to retrieve tools via finetuning
- An example is the *Toolformer* model, which was finetuned to decide which APIs to call, when to call them, what arguments to pass, and how to best incorporate the results into future token prediction
- Learned tools include a calculator, a QA system, a search engine, a translation system, and a calendar

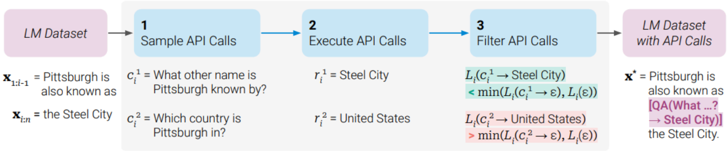
Tool retrieval: Finetuning-based approach

The finetuning dataset *LM Dataset* used for Toolformer is generated in the following way:



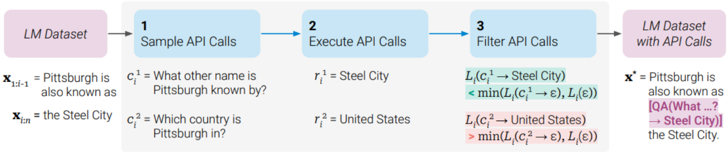
Tool retrieval: Finetuning-based approach

- Toolformer starts by sampling a large number of potential API calls.
- These API calls are executed and the obtained responses are checked if they are helpful for predicting future tokens; this is used as a filtering criterion.
- After filtering, the API calls for different tools are interleaved within the original data sample



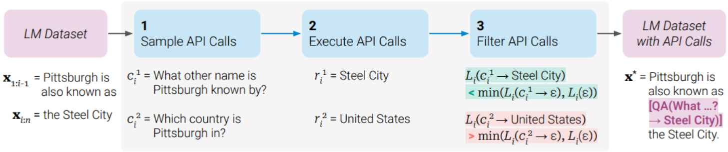
Tool retrieval: Finetuning-based approach

- In the below example, the LM dataset has the sample text: “Pittsburgh is also known as the Steel City“ for the input prompt of, “Pittsburgh is also known as“
- To find the correct answer, the model needs to make an API call and do so correctly.
- A few search API calls are sampled, e.g., ‘What other name is Pittsburgh known by?” and “Which country is Pittsburgh in?”
- The corresponding answers are, “Steel City“ and “United States“
- The answer that reduces the modeling loss is chosen as the correct answer



Tool retrieval: Finetuning-based approach

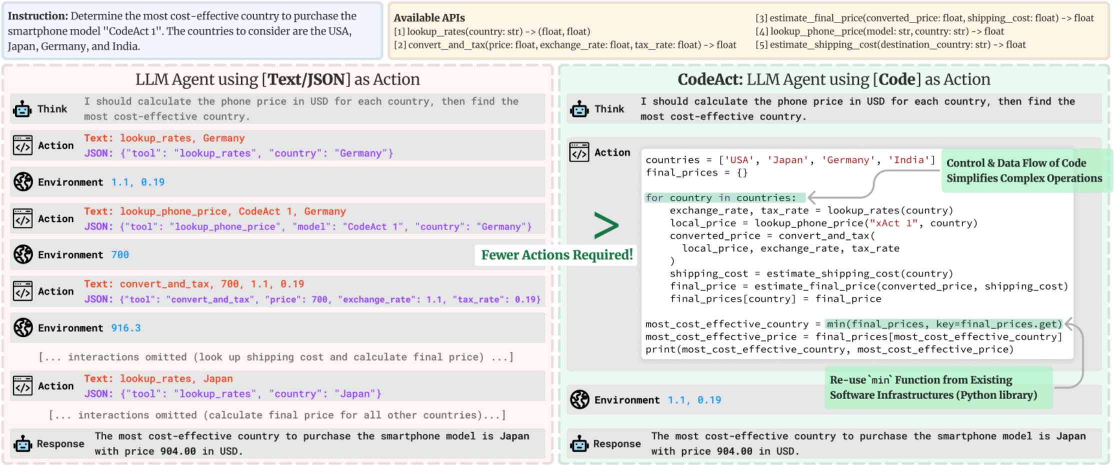
- The better answer is included into a new LM dataset with API calls as: `"Pittsburgh is also known as [QA(What other name is Pittsburgh known by?") -> Steel City] the Steel City."`
- This contains the expected API call along with the answer. This step is repeated to generate a new LM dataset with each kind of tool (i.e., API call).



The CodeAct framework: replacing text with code

- We know that pretrained LLMs can generate sophisticated Python code out-of-the-box
- Why not leverage this and implement agents as executable code rather than as text?
- CodeAct executes code actions and can dynamically revise prior actions or emit new actions upon new observations through multi-turn interactions

The CodeAct framework:replacing text with code



Open research problems

- **Role-playing capability:** LLM-based agents typically need to adapt a role to effectively complete tasks in a domain. For roles that the LLM doesn't characterize well, it's possible to fine-tune the LLM on data that represent uncommon roles or psychology characters.
- **Long-term planning and finite context length:** planning over a lengthy history remains a challenge that could lead to errors that the agent may not recover from. LLMs are also limited in context length they can support which could lead to constraints that limit the capabilities of the agent such as leveraging short-term memory.

Open research problems

- **Efficiency**: LLM agents involve a significant amount of requests that are handled by the LLM which could affect the efficiency of agent actions because it would depend heavily on the LLM inference speed. Cost is also a concern when deploying multiple agents.

Next class: Dec 2

Reading

- Rethinking Interpretability in the Era of Large Language Models
- Sirens Song in the AI Ocean: A Survey on Hallucination in Large Language Models
- Self-RAG: Learning to Retrieve, Generate, and Critique through Self-Reflection
- SELFCHECKGPT: Zero-Resource Black-Box Hallucination Detection for Generative Large Language Models