Relevant Code

Code Repository

Repository: Not Available (Pseudocode Implementation)

Status: Pseudocode (No Repository Found)

Language: Python-based pseudocode

Last Updated: Based on paper from October 2022

Stars/Popularity: N/A

Note: The paper mentions code at https://anonymous.4open.science/r/ReAct-2268/ but this anonymous repository is no longer accessible. The following implementation is based on the detailed algorithmic descriptions in the paper.

Architecture Overview

ReAct's architecture centers around a unified action space that combines traditional domain-specific actions with language-based reasoning traces. The system operates through an interleaved thought-action-observation cycle, where a large language model generates both reasoning traces and actions in sequence.

The core architecture consists of:
1. **Action Space Augmentation**: Extends traditional action spaces to include language thoughts alongside domain-specific actions
2. **Context Management**: Maintains a growing context window containing the complete trajectory of thoughts, actions, and observations
3. **Environment Interface**: Provides standardized methods for interacting with external environments like Wikipedia APIs or interactive environments
4. **Prompt Engineering**: Uses few-shot in-context examples to demonstrate the desired reasoning-action patterns

Directory Structure

```
react_framework/
├── core/
│   ├── __init__.py
```

```
│   ├── agent.py          # Main ReAct agent implementation
│   ├── action_space.py   # Unified action space with thoughts + actions
│   └── context.py        # Context management for trajectories
├── environments/
│   ├── __init__.py
│   ├── wikipedia.py      # Wikipedia API wrapper
│   ├── alfworld.py       # ALFWorld environment interface
│   └── webshop.py        # WebShop environment interface
├── prompts/
│   ├── __init__.py
│   ├── hotpotqa.py       # HotpotQA prompting templates
│   ├── fever.py          # Fever prompting templates
│   ├── alfworld.py       # ALFWorld prompting templates
│   └── webshop.py        # WebShop prompting templates
└── utils/
    ├── __init__.py
    ├── parsing.py        # Response parsing utilities
    └── evaluation.py     # Evaluation metrics
```

Key Implementation

Core ReAct Agent

The main agent implements the interleaved reasoning and action cycle described in Section 2 of the paper. It manages the unified action space and maintains the complete trajectory context.

```
class ReActAgent:
    """
    Main ReAct agent implementation that synergizes reasoning and acting.
    Based on the interleaved thought-action-observation paradigm from the paper.
    """

    def __init__(self, llm, action_space, max_steps=7):
        self.llm = llm
        self.action_space = action_space
        self.max_steps = max_steps
        self.trajectory = []

    def solve(self, question_or_instruction, examples):
```

```
        """
        Main solving method that implements the ReAct paradigm.

        Args:
            question_or_instruction: Input task (question for QA, instruction for decision making)
            examples: Few-shot examples demonstrating desired reasoning-action patterns

        Returns:
            Complete trajectory of thoughts, actions, and observations
        """
        # Initialize context with task description and examples
        context = self._build_prompt(question_or_instruction, examples)
        self.trajectory = []

        for step in range(self.max_steps):
            # Generate next action (either thought or domain-specific action)
            response = self.llm.generate(context)
            action_type, action_content = self._parse_response(response)

            if action_type == "Thought":
                # Add reasoning trace to trajectory (no external observation)
                self.trajectory.append({
                    "type": "thought",
                    "content": action_content
                })
                context += f"Thought {len(self.trajectory)}: {action_content}\n"

            elif action_type == "Action":
                # Execute action and get observation
                observation = self.action_space.execute(action_content)
                self.trajectory.append({
                    "type": "action",
                    "content": action_content,
                    "observation": observation
                })
                context += f"Act {len([t for t in self.trajectory if t['type'] == 'action'])}: {act
                context += f"Obs {len([t for t in self.trajectory if t['type'] == 'action'])}: {obs

            elif action_type == "Finish":
                # Task completed
                self.trajectory.append({
```

```
                    "type": "finish",
                    "content": action_content
                })
                break

        return self.trajectory

    def _build_prompt(self, task, examples):
        """Build the complete prompt with task description and examples."""
        prompt = "Solve the following task using interleaved Thought, Action, Observation sequences

        # Add few-shot examples
        for i, example in enumerate(examples):
            prompt += f"Example {i+1}:\n{example}\n\n"

        # Add current task
        prompt += f"Task:\n{task}\n"
        prompt += "Thought 1: "

        return prompt

    def _parse_response(self, response):
        """Parse model response to extract action type and content."""
        response = response.strip()

        if response.startswith("Thought"):
            return "Thought", response[7:].strip()
        elif response.startswith("Act"):
            return "Action", response[4:].strip()
        elif response.startswith("finish"):
            return "Finish", response[7:].strip()
        else:
            # Fallback parsing
            return "Thought", response
```

Wikipedia Action Space Implementation

This implements the simple Wikipedia API interface used in the knowledge-intensive reasoning experiments (Section 3.1).

```
class WikipediaActionSpace:
    """
    Wikipedia API wrapper implementing the three action types from Section 3.1:
    - search[entity]: Returns first 5 sentences from entity page or suggestions
    - lookup[string]: Returns next sentence containing string (Ctrl+F functionality)
    - finish[answer]: Completes the task with final answer
    """

    def __init__(self, wiki_client):
        self.wiki_client = wiki_client

    def execute(self, action):
        """Execute Wikipedia actions and return observations."""
        if action.startswith("search[") and action.endswith("]"):
            entity = action[7:-1]
            return self._search_entity(entity)

        elif action.startswith("lookup[") and action.endswith("]"):
            string = action[7:-1]
            return self._lookup_string(string)

        elif action.startswith("finish[") and action.endswith("]"):
            answer = action[7:-1]
            return f"Task completed with answer: {answer}"

        else:
            return f"Invalid action format: {action}"

    def _search_entity(self, entity):
        """Search for Wikipedia entity and return first 5 sentences."""
        try:
            page = self.wiki_client.page(entity)
            if page.exists():
                sentences = page.summary.split('.')[:5]
                return '. '.join(sentences) + '.'
            else:
                # Return similar entity suggestions
                suggestions = self.wiki_client.search(entity, results=5)
                return f"Page '{entity}' not found. Did you mean: {', '.join(suggestions)}?"
        except Exception as e:
```

```
            return f"Error searching for '{entity}': {str(e)}"

    def _lookup_string(self, string):
        """Lookup string in current page (simulates Ctrl+F)."""
        # This would maintain page state and find next occurrence
        # Simplified implementation for demonstration
        return f"Found '{string}' in current page content. [Implementation would show next sentence
```

ALFWorld Environment Interface

Implementation for the text-based game environment requiring sparse reasoning for decision making (Section 4).

```
class ALFWorldActionSpace:
    """
    ALFWorld environment interface for household task completion.
    Implements sparse reasoning patterns where thoughts appear at critical decision points.
    """

    def __init__(self, alfworld_env):
        self.env = alfworld_env
        self.current_observation = None

    def execute(self, action):
        """Execute ALFWorld actions (navigation, object interaction, etc.)."""
        if action.startswith("go to") or action.startswith("take") or \
           action.startswith("use") or action.startswith("open") or \
           action.startswith("put") or action.startswith("clean"):
            # Domain-specific actions
            observation, reward, done, info = self.env.step(action)
            self.current_observation = observation
            return observation
        else:
            return f"Invalid ALFWorld action: {action}"

    def reset(self, task_description):
        """Reset environment for new task."""
        observation = self.env.reset()
        self.current_observation = observation
        return observation
```

Prompt Templates for HotpotQA

Example of the few-shot prompting approach used for knowledge-intensive reasoning tasks (Section 3.2).

```
def build_hotpotqa_prompt():
    """
    Build HotpotQA prompt with few-shot examples demonstrating ReAct patterns.
    Based on the manual trajectory composition described in Section 3.2.
    """

    examples = [
        # Example 1: Multi-hop question requiring multiple searches
        """Question: Which documentary film about the Sierra Club's campaign
        to block the construction of the Diablo Canyon Power Plant was
        released by 1972?

Thought 1: I need to search for information about the Diablo Canyon Power Plant
        and its opposition by the Sierra Club to find the documentary.
Act 1: search[Diablo Canyon Power Plant Sierra Club documentary]

Obs 1: Diablo Canyon is a nuclear power plant in California that faced
        opposition from environmental groups including the Sierra Club.
        A documentary about the anti-nuclear movement was made in the 1970s.

Thought 2: The first search mentions a documentary about the anti-nuclear movement
        but doesn't give the specific title. I should search more specifically
        for documentaries about this controversy from the early 1970s.
Act 2: search[documentary Diablo Canyon Sierra Club 1972]

Obs 2: "Countdown to Zero" is a 1972 documentary film that chronicles
        the Sierra Club's campaign against the Diablo Canyon Power Plant construction.

Thought 3: I found the specific documentary title "Countdown to Zero" released in 1972
        about the Sierra Club's campaign against Diablo Canyon. This should be the answer.
Act 3: finish[Countdown to Zero]""",

        # Example 2: Question requiring comparison of information
        """Question: Are both the author of "The Left Hand of Darkness" and
        the author of "The Word for World is Forest" from the United States?
```

```
Thought 1: I need to find who wrote "The Left Hand of Darkness" and their nationality.
Act 1: search[The Left Hand of Darkness author]

Obs 1: "The Left Hand of Darkness" is a 1969 novel by Ursula K. Le Guin.
        Ursula K. Le Guin was an American author born in Berkeley, California.

Thought 2: Now I need to find who wrote "The Word for World is Forest" and check their nationality.
Act 2: search[The Word for World is Forest author]

Obs 2: "The Word for World is Forest" is a 1972 novel also by Ursula K. Le Guin.

Thought 3: Both books were written by the same author, Ursula K. Le Guin, who is from the United St
        Since they're the same person, the answer is yes.
Act 3: finish[Yes]"""
    ]

    return examples
```

Combination Strategy for ReAct + CoT

Implementation of the hybrid approach that combines ReAct with Chain-of-Thought Self-Consistency (Section 3.2).

```
class ReActCoTCombination:
    """
    Implements the ReAct + CoT-SC combination strategies from Section 3.2:
    - ReAct → CoT-SC: Fall back to CoT when ReAct fails within step limits
    - CoT-SC → ReAct: Fall back to ReAct when CoT consensus is weak
    """

    def __init__(self, react_agent, cot_agent, consensus_threshold=0.5):
        self.react_agent = react_agent
        self.cot_agent = cot_agent
        self.consensus_threshold = consensus_threshold

    def solve_with_fallback(self, task, strategy="react_to_cot"):
        """Solve task using hybrid strategy with fallback."""

        if strategy == "react_to_cot":
            # Try ReAct first, fallback to CoT-SC if no answer within steps
```

```
            react_trajectory = self.react_agent.solve(task, examples)
            answer = self._extract_answer_from_trajectory(react_trajectory)

            if answer is None:
                # ReAct failed, fallback to CoT with self-consistency
                return self.cot_agent.solve_with_consensus(task, samples=21)
            else:
                return react_trajectory

        elif strategy == "cot_to_react":
            # Try CoT-SC first, fallback to ReAct if consensus is weak
            cot_results = self.cot_agent.solve_multiple_samples(task, samples=21)
            majority_answer, consensus_ratio = self._analyze_consensus(cot_results)

            if consensus_ratio < self.consensus_threshold:
                # Weak consensus, fallback to ReAct
                return self.react_agent.solve(task, examples)
            else:
                return majority_answer

    def _extract_answer_from_trajectory(self, trajectory):
        """Extract final answer from ReAct trajectory."""
        for step in reversed(trajectory):
            if step.get("type") == "finish":
                return step.get("content")
        return None

    def _analyze_consensus(self, cot_results):
        """Analyze consensus strength in CoT results."""
        answer_counts = {}
        for result in cot_results:
            answer = result.get("answer")
            answer_counts[answer] = answer_counts.get(answer, 0) + 1

        if not answer_counts:
            return None, 0.0

        majority_answer = max(answer_counts.keys(), key=lambda x: answer_counts[x])
        consensus_ratio = answer_counts[majority_answer] / len(cot_results)

        return majority_answer, consensus_ratio
```

Relation to Paper

Paper Section	Code Component	Notes
Section 2: ReAct Paradigm	`ReActAgent.solve()`	Core implementation of interleaved reasoning and acting
Equation 1 (Action Space)	`action_space.py`	Implementation of augmented action space $ \hat{\mathcal{A}} =\mathcal{A} \cup\mathcal{L}$
Section 3.1: Wikipedia API	`WikipediaActionSpace`	Simple 3-action interface (search, lookup, finish)
Section 3.2: Prompting	`build_hotpotqa_prompt()`	Few-shot examples with thought-action-observation patterns
Section 4: ALFWorld	`ALFWorldActionSpace`	Sparse reasoning for interactive decision making
ReAct → CoT Strategy	`ReActCoTCombination.solve_with_fallback()`	Hybrid approach combining both methods
Figure 1 (Trajectories)	`ReActAgent.trajectory`	Complete trace of thoughts, actions, observations

Key Differences from Paper

- **Simplified Environment Interfaces**: The paper used more sophisticated environment wrappers; this implementation provides simplified interfaces for clarity

- **Prompt Engineering**: The paper carefully crafted prompts with specific thought patterns; this implementation shows example patterns but would require the exact prompts from Appendix C for full reproduction
- **Model Integration**: The paper used PaLM-540B; this implementation abstracts the LLM interface to work with any language model
- **Fine-tuning**: The paper describes bootstrapping fine-tuning; this implementation focuses on the prompting approach only

Usage Example

```
# Example usage for HotpotQA question answering
from transformers import AutoModelForCausalLM, AutoTokenizer

# Initialize LLM (would use PaLM in original paper)
llm = LanguageModelWrapper(model_name="gpt-3.5-turbo")

# Initialize action space for Wikipedia
wiki_api = WikipediaAPI()
action_space = WikipediaActionSpace(wiki_api)

# Create ReAct agent
agent = ReActAgent(llm, action_space, max_steps=7)

# Load HotpotQA examples
examples = build_hotpotqa_prompt()

# Solve a question
question = "Which documentary film about the Sierra Club's campaign to block the construction of th
trajectory = agent.solve(question, examples)

# Print complete reasoning trace
for step in trajectory:
    if step["type"] == "thought":
        print(f"Thought: {step['content']}")
    elif step["type"] == "action":
        print(f"Action: {step['content']}")
        print(f"Observation: {step['observation']}")
    elif step["type"] == "finish":
        print(f"Final Answer: {step['content']}")
```