CoRE

CoRE System Documentation

Table of Contents

Introduction

CoRE (Collaboration of Role-based Experts) is a proof-of-concept implementation of the RaTiO (Reasoning through Interaction of Experts) framework. This system leverages multiple specialized language models, treating each as an expert with distinct capabilities, to collaboratively solve complex queries through a sequential processing approach.

The primary goal of CoRE is to demonstrate how multiple smaller language models can work together to produce responses that match or exceed the quality of larger, more resource-intensive models while operating efficiently on limited hardware.

RaTiO Framework Overview

The RaTiO framework provides a structured approach to collaborative reasoning among multiple language models. Rather than relying on a single large model, RaTiO distributes the cognitive load across several specialized experts, each contributing unique insights based on their strengths.

Key Principles of RaTiO

1. **Expert Specialization**: Different models excel at different tasks (reasoning, creativity, engagement, etc.)
2. **Sequential Processing**: Experts build upon each other's work through a chain of reasoning
3. **Dynamic Expert Selection**: The most relevant experts are chosen based on query requirements
4. **Knowledge Aggregation**: A designated model synthesizes the collective insights
5. **Hardware Efficiency**: Sequential processing requires less memory than parallel approaches

CoRE System Implementation

CoRE is the practical implementation of the RaTiO framework using Streamlit for the user interface and Langchain for managing the expert workflows. It serves as a demonstration of how collaborative reasoning can be achieved using smaller, more accessible language models.

System Architecture

The CoRE system is built using a modular architecture with the following major components:

1. **User Interface**: A Streamlit web application that allows users to input queries, configure expert selection, and view results
2. **Expert Models**: A collection of language models hosted via Ollama, with optional Gemini API integration
3. **Processing Pipeline**: A sequential workflow that routes queries through relevant experts
4. **Context Management**: Multi-tiered context storage that maintains conversation history
5. **Knowledge Transfer**: A system for extracting and reusing insights across different queries
6. **Visualization Tools**: Interactive charts that provide insights into expert contributions

Expert Models

CoRE supports the following expert models:

Model	Default Role	Strengths
Gemma2 (2B)	Case Study Expert	Conversational skills, detailed analysis
LLaMA3.2 (3B)	Logical Reasoning Expert	Critical thinking, structured analysis
Phi3 (latest)	Engagement Expert	Clear explanations, user engagement
Qwen2.5 (3B)	Creative Expert	Creative problem-solving, novel approaches
Gemini Pro (API)	Code Writing Expert	Technical implementation, code generation

User Interface

The CoRE system provides a Streamlit-based interface with the following features:

1. **Query Input**: A text area for entering user queries
2. **Expert Configuration**:
    - Toggle between automatic and manual expert selection
    - Customize the role of each expert

3. **Aggregator Selection**: Choose which model synthesizes the final response
4. **Visualization Panels**:
    - Expert relevance radar chart (for automatic selection)
    - Response flow diagram showing contribution lengths
5. **Results Display**:
    - Expandable view of individual expert responses
    - Final synthesized response

Query Processing Workflow

1. **Query Submission**: User enters a query through the interface
2. **Expert Selection**:
    - **Automatic Mode**: The gating mechanism analyzes the query and assigns relevance scores to each expert, selecting those above a threshold
    - **Manual Mode**: User selects which experts to involve and optionally customizes their roles
3. **Context Retrieval**: The system fetches relevant context from previous interactions
4. **Sequential Processing**: Each selected expert processes the query, building upon previous expert outputs
5. **Knowledge Transfer**: Common insights are identified and stored in the shared knowledge base
6. **Response Aggregation**: The designated aggregator model synthesizes all expert responses
7. **Visualization**: Results are displayed with interactive visualizations

Key Components

Expert Class

The `Expert` class encapsulates a language model with a specific role. It handles:

- Model initialization via Ollama
- Query processing with role-specific prompting
- Context integration into responses

```
class Expert:
    def __init__(self, model_name: str, role: str = None):
        self.model_name = model_name
        self.role = role
        self.llm = Ollama(model=model_name)

    def process(self, query: str, context: str = "") -> str:
        prompt_template = PromptTemplate(
            input_variables=["role", "context", "query"],
            template="""You are an expert with the role: {role}.
            Previous context: {context}
            Please analyze and respond to the following query: {query}
            Analyse the previous context and iterate upon it with your perspective, correct any info if required.
            Provide your expert perspective while staying focused on your role."""
        )

        chain = LLMChain(llm=self.llm, prompt=prompt_template)
        response = chain.run(role=self.role, context=context, query=query)
        return response
```

Expert Gating Mechanism

The `ExpertGating` class dynamically determines which experts are most relevant to a query by:

1. Analyzing the query using a small LLM (LLaMA3.2:3b by default)
2. Assigning relevance scores across four dimensions:
    - Conversational: Need for social interaction/dialogue
    - Logical: Requirement for reasoning/analysis
    - Creative: Degree of imagination/innovation needed
    - Engagement: Extent of explanation/elaboration required
3. Selecting experts with scores above a threshold (0.3)

This approach ensures that only the most appropriate experts are engaged for each query, optimizing both performance and response quality.

Context Manager

The `ContextManager` class implements a multi-tiered memory system that maintains conversation history across different timeframes:

1. **Short-term Context**: Recent interactions (last 5 exchanges)
2. **Medium-term Context**: Extended history (up to 20 exchanges)
3. **Long-term Context**: Persistent memory (up to 50 exchanges)

The manager employs semantic compression to avoid redundancy:

```
def semantic_compress(self, context: List[Dict[str, Any]]) -> List[Dict[str, Any]]:
    """
    Semantically compress context by removing redundant information
    """
    if not context or len(context) < 2:
        return context

    # Extract texts, filtering out invalid entries
    texts = [item['text'] for item in context if item.get('text') and isinstance(item['text'], str)]

    if len(texts) < 2:
        return context

    try:
        embeddings = self.embedding_model.encode(texts)

        # Ensure 2D array
        if embeddings.ndim == 1:
            embeddings = embeddings.reshape(1, -1)

        # Compute similarity matrix
        similarity_matrix = cosine_similarity(embeddings)

        # Select unique and most representative contexts
        unique_indices = []
        for i in range(len(texts)):
            is_unique = True
            for j in unique_indices:
                if similarity_matrix[i][j] > 0.8:  # High similarity threshold
                    is_unique = False
                    break
            if is_unique:
                unique_indices.append(i)

        return [context[texts.index(texts[i])] for i in unique_indices]
    except Exception as e:
```

```
        print(f"Error in semantic compression: {e}")
        return context
```

This tiered approach with semantic compression allows the system to maintain relevant context without overwhelming the models with repetitive information.

Knowledge Transfer Hub

The `KnowledgeTransferHub` facilitates cross-expert learning and knowledge reuse through:

1. **Common Insight Extraction**: Identifying shared conclusions across expert responses
2. **Knowledge Propagation**: Storing valuable insights in a shared knowledge base
3. **Relevant Knowledge Retrieval**: Fetching insights pertinent to new queries

This component enables the system to build upon previously established knowledge, improving the quality and consistency of responses over time.

Key methods include:

```
def extract_common_insights(self, expert_responses: List[Dict[str, Any]]) → List[str]:
    """
    Extract common insights across expert responses
    """
    # Implementation details...

def propagate_knowledge(self, expert_responses: List[Dict[str, Any]]):
    """
    Propagate knowledge across experts and update shared knowledge base
    """
    # Implementation details...

def retrieve_relevant_knowledge(self, query: str, top_k: int = 3) → List[str]:
    """
    Retrieve most relevant knowledge from shared knowledge base
    """
    # Implementation details...
```

The Knowledge Transfer Hub uses SentenceTransformer embeddings and cosine similarity to identify semantically similar content, ensuring that only truly novel insights are added to the knowledge base.

Gemini Expert Integration

The `GeminiExpert` class extends CoRE's capabilities by integrating with Google's Gemini API:

```
class GeminiExpert:
    def __init__(self, api_key: str, model_name: str = "gemini-pro"):
        """
        Initialize Gemini Expert with API configuration
        """
        genai.configure(api_key=api_key)
        self.model = genai.GenerativeModel(model_name)

    def process(self, query: str, role: str, context: str = "") → str:
        """
        Process query using Gemini model with context
        """
        self.role=role
        full_prompt = f"""You are an expert with the role: {role}.
            Previous context: {context}
            Please analyze and respond to the following query: {query}
            Analyse the previous context and iterate upon it with your perspective, correct any info if required.
```

```
        Provide your expert perspective while staying focused on your role."""

    try:
        response = self.model.generate_content(full_prompt)
        return response.text
    except Exception as e:
        return f"Gemini Expert Error: {str(e)}"
```

This integration allows the system to incorporate more advanced models when available, enhancing its capabilities while maintaining the flexible expert-based architecture.

Visualization Features

CoRE provides two main visualizations to enhance understanding of the system's operation:

Expert Relevance Radar Chart

A polar chart displaying relevance scores for each expert dimension (conversational, logical, creative, and engagement), helping users understand which aspects of their query are most prominent.

```
def create_expert_radar_chart(scores: Dict[str, float]):
    categories = list(scores.keys())
    values = list(scores.values())
    values.append(values[0])  # Complete the circle
    categories.append(categories[0])  # Complete the circle

    fig = go.Figure(data=[
        go.Scatterpolar(
            r=values,
            theta=categories,
            fill='toself',
            name='Expert Relevance'
        )
    ])
    # More configuration...
    return fig
```

Response Flow Diagram

A horizontal bar chart showing the length contribution of each expert, providing insight into which experts contributed most substantially to the final response.

```
def create_response_flow_diagram(intermediate_responses: List[Dict]):
    fig = make_subplots(rows=len(intermediate_responses), cols=1,
                        subplot_titles=[f"{resp['model'].title()} ({resp['role']})"
                                        for resp in intermediate_responses])

    for i, resp in enumerate(intermediate_responses, 1):
        response_len = len(resp['response'])
        fig.add_trace(
            go.Bar(
                x=[response_len],
                y=[resp['model']],
                orientation='h',
                name=resp['model'],
                text=[f"{response_len} chars"],
                textposition='auto',
            ),
            row=i, col=1
        )
```

```
    # More configuration...
    return fig
```

Hardware and Performance Advantages

CoRE's architecture offers several key advantages for real-world deployment:

Low Memory Requirements

By processing models sequentially rather than simultaneously, CoRE requires only enough VRAM to run one model at a time. This contrasts with traditional Mixture of Experts (MoE) systems, which run experts in parallel and therefore require VRAM capacity for all engaged experts simultaneously.

Resource-Efficient Scaling

CoRE can scale up by simply adding more experts in sequence without a proportional increase in memory requirements. This flexibility makes it suitable for deployment on both resource-constrained and high-performance systems.

Quality Through Collaboration

The system achieves high-quality responses by leveraging the collective strengths of multiple smaller models, potentially matching or exceeding the capabilities of much larger models that would require significantly more resources.

Implementation Details

Key Classes and Methods

The MoRESystem class serves as the central controller for the CoRE implementation:

```
class MoRESystem:
    def __init__(self, gemini_api_key: str = None):
        self.available_models = {
            "gemma": "gemma2:2b",
            "llama": "llama3.2:3b",
            "phi": "phi3:latest",
            "qwen": "qwen2.5:3b",
            "gemini": "gemini-pro"
        }

        self.default_roles = {
            "gemma": "Case Study Expert",
            "llama": "Logical Reasoning Expert",
            "phi": "Engagement Expert",
            "qwen": "Creative Expert",
            "gemini": "Code writing Expert"
        }

        self.experts: Dict[str, Union[Expert, GeminiExpert]] = {}
        self.gating = ExpertGating()

        # Context and knowledge transfer modules
        self.context_manager = ContextManager()
        self.knowledge_transfer = KnowledgeTransferHub()

        # Gemini API key (optional)
        self.gemini_api_key = gemini_api_key

    def initialize_experts(self, selected_models: List[str], custom_roles: Dict[str, str] = None):
        # Implementation...
```

```
def process_query(self, query: str, aggregator_model: str) → Tuple[str, List[Dict]]:
    # Implementation...

def auto_select_experts(self, query: str) → Tuple[List[str], Dict[float, str]]:
    # Implementation...
```

Integration with External Services

CoRE integrates with several external services and libraries:

1. **Ollama**: For hosting and accessing local language models
2. **Langchain**: For managing the chain-of-thought reasoning process
3. **SentenceTransformer**: For generating embeddings for semantic analysis
4. **Google Generative AI**: For accessing the Gemini model
5. **Plotly**: For creating interactive visualizations
6. **Streamlit**: For the web-based user interface

Comparison: Framework vs. Implementation

Aspect	RaTiO Framework	CoRE Implementation
Focus	General framework for expert collaboration	Practical demonstration using specific tools
Models	Supports any compatible LLMs	Currently uses Ollama models with optional Gemini
Architecture	Theoretical design for expert interaction	Concrete implementation with specific classes
Customization	Highly extensible conceptual framework	Configured for demonstration with limited options
Context Management	Conceptual multi-tiered memory	Implemented with short/medium/long-term storage
Knowledge Transfer	Theoretical cross-expert learning	Implemented with embedding-based similarity

Getting Started

Installation Requirements

To run the CoRE system, you need the following dependencies:

- Python 3.8+
- Streamlit
- Langchain
- Ollama (with model files for Gemma2, LLaMA3.2, Phi3, and Qwen2.5)
- SentenceTransformer
- Plotly
- Google Generative AI Python SDK (for Gemini integration)
- Scikit-learn

Basic Usage

1. Ensure Ollama is running with the required models
2. Launch the Streamlit application:

```
streamlit run app.py
```

3. Input your query in the text area
4. Select experts manually or let the system choose automatically
5. Click "Process Query" to generate a response

Advanced Configuration

CoRE offers several advanced configuration options:

1. **Custom Roles**: Define specialized roles for each expert
2. **Gemini Integration**: Add your Gemini API key to include Gemini Pro in the expert pool
3. **Aggregator Selection**: Choose which model synthesizes the final response
4. **Expert Selection Mode**: Toggle between automatic and manual expert selection

Future Development

Potential enhancements for the CoRE system include:

1. **Parallel Processing Option**: Enable concurrent expert processing for systems with adequate resources
2. **Additional Models**: Expand the expert pool with more specialized models
3. **Enhanced Knowledge Management**: Implement more sophisticated knowledge extraction and application
4. **User Feedback Integration**: Learn from user ratings to improve expert selection
5. **Domain-Specific Configurations**: Pre-configured expert setups for specific domains like medicine, law, etc.
6. **Multi-Modal Support**: Extend the framework to handle images, audio, and other input types
7. **Fine-Tuning Interface**: Allow users to fine-tune expert models for specific tasks