

7. Multimodal Visual Localization and Voice Interaction

7. Multimodal Visual Localization and Voice Interaction

1. Concept Introduction
 - 1.1 What is "Multimodal Visual Localization"?
 - 1.2 Brief Description of Implementation Principles
2. Code Analysis
 - Key Code
 1. Tool Layer Entry Point (`largemodel/utils/tools_manager.py`)
 2. Model interface layer (`largemodel/utils/large_model_interface.py`)
 - Code Analysis
3. Practical Application
 - 3.1 Configuring the Offline Large-Scale Model
 - 3.1.1 Configuring the LLM Platform (`yahboom.yaml`)
 - 3.1.2 Configuration model interface (`large_model_interface.yaml`)
 - 3.2 Starting and Testing the Functionality

Note: Raspberry Pi 5 2GB/4GB, RDK X5 4GB, and Jetson Orin Nano 4GB versions cannot run offline due to performance limitations. Please refer to online tutorials for large models.

1. Concept Introduction

1.1 What is "Multimodal Visual Localization"?

Multimodal Visual Localization is a technology that combines multiple sensor inputs (such as cameras, depth sensors, IMUs, etc.) and algorithmic processing techniques to achieve precise identification and tracking of the location and posture of devices or users in the environment. This technology does not rely solely on a single type of sensor data but integrates information from different perceptual modes, thereby improving the accuracy and robustness of localization.

1.2 Brief Description of Implementation Principles

1. **Cross-modal Representation Learning:** To enable LLMs to process visual information, a mechanism needs to be developed to convert visual signals into a form that the model can understand. This might involve using convolutional neural networks (CNNs) or other image-processing-appropriate architectures to extract features and map them into the same embedding space as the text.
2. **Joint Training:** By designing an appropriate loss function, text and visual data can be trained simultaneously within the same framework, allowing the model to learn to establish connections between the two modalities. For example, in a question-answering system, an answer can be given based on both a provided text question and related image content.
3. **Visual-Guided Language Generation/Understanding:** Once an effective cross-modal representation is established, visual information can be leveraged to enhance the functionality of the language model. For instance, given a photograph, the model can not only describe what is happening in the image but also answer specific questions about the scene and even execute instructions based on visual cues (such as navigating to a location).

2. Code Analysis

Key Code

1. Tool Layer Entry Point (`largemodel/utils/tools_manager.py`)

The `visual_positioning` function in this file defines the execution flow of the tool, specifically how it constructs a Prompt containing the target object's name and formatting requirements.

```
# From largemodel/utils/tools_manager.py
class ToolsManager:
    # ...
    def visual_positioning(self, args):
        """
        Locate object coordinates in image and save results to MD file.

        :param args: Arguments containing image path and object name.
        :return: Dictionary with file path and coordinate data.
        """
        self.node.get_logger().info(f"Executing visual_positioning() tool with
args: {args}")
        try:
            image_path = args.get("image_path")
            object_name = args.get("object_name")
            # ... (Path fallback mechanism and parameter checking)

            # Construct a prompt asking the large model to identify the
coordinates of the specified object.
            if self.node.language == 'zh':
                prompt = f"请仔细分析这张图片，用一个个框定位图像每一个{object_name}的位
置..."
            else:
                prompt = f"Please carefully analyze this image and find the
position of all {object_name}..."

            # ... (Building an independent message context)

            result = self.node.model_client.infer_with_image(image_path, prompt,
message=message_to_use)

            # ... (Process and parse the returned coordinate text)

            return {
                "file_path": md_file_path,
                "coordinates_content": coordinates_content,
                "explanation_content": explanation_content
            }
        # ... (Error Handling)
```

2. Model interface layer

(`largemodel/utils/large_model_interface.py`)

The `infer_with_image` function in this file is the unified entry point for all image-related tasks.

```
# From largemodel/utils/large_model_interface.py

class model_interface:
    # ...
```

```
def infer_with_image(self, image_path, text=None, message=None):
    """Unified image inference interface. """
    # ... (Prepare message)
    try:
        # Determine which specific implementation to call based on the value
        # of self.llm_platform
        if self.llm_platform == 'ollama':
            response_content = self.ollama_infer(self.messages,
            image_path=image_path)
        elif self.llm_platform == 'tongyi':
            # ... Logic for calling the Tongyi model
            pass
        # ... (Logic of other platforms)
        # ...
    return {'response': response_content, 'messages': self.messages.copy()}
```

Code Analysis

The core of the visual positioning function lies in **guiding large models to output structured data through precise instructions**. It also follows the layered design of the tool layer and the model interface layer.

1. Tools Layer (`tools_manager.py`):

- The `visual_positioning` function is the core of this function. It accepts two key parameters: `image_path` (the image path) and `object_name` (the name of the object to be positioned).
- The core operation of this function is **building a highly customized prompt**. It doesn't simply ask the model to describe an image. Instead, it embeds `object_name` into a carefully designed template, explicitly instructing the model to "locate each {object_name} in the image," and implicitly or explicitly requires the results to be returned in a specific format (such as an array of coordinates).
- After building the prompt, it calls the `infer_with_image` method of the model interface layer, passing the image and this customized instruction.
- After receiving the returned text from the model interface layer, it needs to perform **post-processing**: using methods such as regular expressions to parse the model's natural language response to extract precise coordinate data.
- Finally, it returns the parsed structured coordinate data to the upper-layer application.

2. Model Interface Layer (`large_model_interface.py`):

- The `infer_with_image` function still serves as the "dispatching center." It receives the image and prompt from `visual_positioning` and dispatches the task to the correct backend model implementation based on the current configuration (`self.llm_platform`).
- For visual positioning tasks, the model interface layer's responsibilities are essentially the same as for visual understanding tasks: correctly packaging the image data and text instructions, sending them to the selected model platform, and then returning the returned text results intact to the tool layer. All platform-specific implementation details are encapsulated in this layer.

In summary, the general workflow for visual localization is: `ToolsManager` receives the target object name and constructs a precise prompt requesting coordinates. `ToolsManager` calls the model interface. `model_interface` packages the image and prompt together and sends them to the corresponding model platform according to the configuration. The model returns text

containing the coordinates. `model_interface` returns this text to `ToolsManager`. `ToolsManager` parses the text, extracts the structured coordinate data, and returns it. This process demonstrates how prompt engineering can be used to enable a general-purpose large-scale visual model to accomplish more specific and structured tasks.

3. Practical Application

3.1 Configuring the Offline Large-Scale Model

3.1.1 Configuring the LLM Platform (`yahboom.yaml`)

This file determines which large-scale model platform the `model_service` node loads as its primary language model.

Raspberry Pi 5 requires entering a Docker container, while RDK X5 and Orin controllers do not:

```
./ros2_docker.sh
```

If you need to enter other commands in the same Docker container later, simply enter `./ros2_docker.sh` again in the host terminal.

1. **Open the file in the terminal:**

```
vim ~/yahboom_ws/src/largemodel/config/yahboom.yaml
```

2. **Modify/Confirm `llm_platform`:**

```
model_service:                                     #Model server node parameters
  ros__parameters:
    language: 'zh'                                   #Large Model Interface Language
    useolinetts: True                                #This item is invalid in text mode
    and can be ignored

    # Large model configuration
    llm_platform: 'ollama'                            # Key: Make sure it's 'ollama'
    regional_setting : "China"
```

3.1.2 Configuration model interface (`large_model_interface.yaml`)

This file defines which visual model to use when the platform is selected as `ollama`.

1. Open the file in a terminal.

```
vim ~/yahboom_ws/src/largemodel/config/large_model_interface.yaml
```

2. Find the Ollama-related configuration

```
#.....  
## Offline Large Language Models  
# Ollama Configuration  
ollama_host: "http://localhost:11434" # Ollama server address  
ollama_model: "llava" # Key: Change this to the multimodal model you  
downloaded, such as "llava"  
#.....
```

Note: Ensure that the model specified in the configuration parameters (e.g., `llava`) can handle multimodal input.

3.2 Starting and Testing the Functionality

Note: Using a model with a small number of parameters or running at a very low memory usage will result in poor performance. For a better experience with this feature, please refer to the corresponding section in "Online Large Model (Text Interaction)".

1. Prepare Image Files:

Place the image file you want to test in the following path:

```
~/yahboom_ws/src/largemodel/resources_file/visual_positioning
```

Then name the image `test_image.jpg`

2. Start the `largemodel` Main Program:

Open a terminal and enter the Docker container, then run the following command:

```
ros2 launch largemodel largemodel_control.launch.py
```

3. Test:

- **Wake-up:** Say into the microphone, "Hello, Xiaoya."
- **Dialogue:** After the speaker responds, you can say: `Analyze the position of the dinosaur in the image`
- **Observe the Logs:** In the terminal running the `launch` file, you should see:
 1. The ASR node has identified your problem and printed it out.
 2. The `model_service` node receives the text, invokes the LLM, and prints the LLM's response.
- **Listen to the answer:** After a while, you should hear the answer from the speaker and find an md file in the `~/yahboom_ws/src/largemodel/resources_file/visual_positioning` path that records the coordinate position and positioning object information.