8. Multimodal Table Scanning Applications

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 Table Scanning"?

Multimodal table scanning is a technology that uses image processing and artificial intelligence to identify and extract table information from images or PDF documents. It not only focuses on visual table structure recognition but also combines multimodal data such as text content and layout information to enhance table understanding. **Large Language Models (LLMs)** provide powerful semantic analysis capabilities for understanding this extracted information. The two complement each other, jointly improving the intelligence level of document processing.

1.2 Brief Description of Implementation Principles

1. **Table Detection and Content Recognition**

- Using computer vision technology to locate tables in documents and using OCR technology to convert the text within the tables into an editable format.
- Employs deep learning methods to parse table structure (row and column division, cell merging, etc.) and generate structured data representation.

2. **Multimodal Fusion**

- Integrates visual data (e.g., table layout), text (OCR results), and potential metadata (e.g., file type, source) to form a comprehensive data view.
- Uses a specially designed multimodal model (e.g., LayoutLM) to process these different types of data simultaneously for a more accurate understanding of table content and its contextual relationships.

2. Code Analysis

Key Code

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

The `scan_table` function in this file defines the execution flow of the tool, specifically how it constructs a Prompt that requests a Markdown format return.

```python
# From largemodel/utils/tools_manager.py
class ToolsManager:
    # ...
    def scan_table(self, args):
        """
        Scan a table from an image and save the content as a Markdown file.

        :param args: Arguments containing the image path.
        :return: Dictionary with file path and content.
        """
        self.node.get_logger().info(f"Executing scan_table() tool with args:
{args}")
        try:
            image_path = args.get("image_path")
            # ... (Path checking and fallback)

            # Construct a prompt asking the large model to recognize the table
and return it in Markdown format.
            if self.node.language == 'zh':
                prompt = "请仔细分析这张图片，识别其中的表格，并将其内容以Markdown格式返
回。"
            else:
                prompt = "Please carefully analyze this image, identify the
table within it, and return its content in Markdown format."

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

            # ... (Extract Markdown text from the results)

            # Save the recognized content to a Markdown file.
            md_file_path = os.path.join(self.node.pkg_path, "resources_file",
"scanned_tables", f"table_{timestamp}.md")
            with open(md_file_path, 'w', encoding='utf-8') as f:
                f.write(table_content)

            return {
                "file_path": md_file_path,
                "table_content": table_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.

```python
# 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 table scanning function is a typical application for converting unstructured image data into structured text data. Its core technology remains **guiding model behavior through prompt engineering**.

1. **Tools Layer (`tools_manager.py`)**:
   - The `scan_table` function is the business process controller for this function. It receives an image containing a table as input.
   - The key operation of this function is **building a targeted prompt**. This prompt directly instructs the large model to perform two tasks: 1. Recognize the table in the image. 2. Return the recognized content in Markdown format. This mandatory output format is key to achieving unstructured-to-structured conversion.
   - After constructing the prompt, it calls the `infer_with_image` method of the model interface layer, passing the image and the formatting instructions.
   - After receiving the Markdown text returned from the model interface layer, it performs a file operation: writing the text content to a new `.md` file.
   - Finally, it returns structured data containing the new file path and table contents.
2. **Model Interface Layer (`large_model_interface.py`)**:
   - The `infer_with_image` function continues to serve as the unified "dispatching center." It receives the image and prompt from `scan_table` and dispatches the task to the correct backend model implementation based on the current system configuration (`self.llm_platform`).
   - Regardless of the backend model, this layer handles the communication details with the specific platform, ensuring that the image and text data are sent correctly, and then returns the plain text (in this case, Markdown-formatted text) returned by the model to the tooling layer.

In summary, the general workflow for table scanning is: `ToolsManager` receives an image and constructs a command to convert the table in this image to Markdown. `ToolsManager` calls the model interface. `model_interface` packages the image and the command and sends it to the corresponding model platform according to the configuration. The model returns Markdown-formatted text. `model_interface` returns the text to `ToolsManager`. `ToolsManager` saves the text as a .md file and returns the result. This workflow demonstrates how to leverage the formatting capabilities of a large model as a powerful OCR (Optical Character Recognition) and data structuring tool.

3. Practical Applications

3.1 Configuring an Offline Large Model

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

This file determines which large 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/scan_table`

Then name the image `test_table.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 table content`
- **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 response:** Later, you should hear the response from the speaker and find a Markdown file containing the table information in the path `~/yahboom_ws/src/largemodel/resources_file/scan_table`.