

# 6. Multimodal Video Analysis and Voice Interaction

---

## 6. Multimodal Video Analysis and Voice Interaction

### 1. Concept Introduction

#### 1.1 What is "Video Analysis"?

#### 1.2 Implementation Principles Overview

### 2. Code Analysis

#### Key Code

1. Utility Layer Entry Point (`largemode1/utils/tools_manager.py`)

2. Model interface layer and frame extraction (`largemode1/utils/large_model_interface.py`)

#### Code Analysis

### 3. Practical Operations

#### 3.1 Configuring the Offline Large Model

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

3.1.2 Configuring the Model Interface (`large_model_interface.yaml`)

#### 3.2 Starting and Testing the Functionality

**Note:** Due to performance limitations, Raspberry Pi 5 2GB/4GB, RDK X5 4GB, and Jetson Orin Nano 4GB versions cannot run offline. Please refer to the online `largemode1` tutorial.

## 1. Concept Introduction

---

### 1.1 What is "Video Analysis"?

In the `Largemode1` project, the **multimodal video analysis** function refers to enabling the robot to process a video and summarize its core content, describe key events, or answer specific questions about the video in natural language. This allows the robot to leap from understanding only static images to understanding a dynamic world with temporal relationships.

The core tool for this function is `analyze_video`. When a user provides a video file and asks a question (e.g., "Summarize what this video is about"), the system calls this tool to process and analyze the video and return the AI's textual answer.

### 1.2 Implementation Principles Overview

The core challenge of offline video analytics lies in how to efficiently process video data containing hundreds or thousands of frames. A mainstream implementation principle is as follows:

1. **Keyframe Extraction:** First, the system does not process every frame of the video. Instead, it extracts the most representative **keyframes** using algorithms (such as scene boundary detection or fixed-time interval sampling). This significantly reduces the amount of data that needs to be processed.
2. **Image Encoding:** Each extracted keyframe is fed into a visual encoder, similar to applications of visual understanding, and converted into a digital vector containing image information.
3. **Temporal Information Fusion:** This is the biggest difference from single-image understanding. The model needs to understand the temporal order between these keyframes. A recurrent neural network (RNN) or Transformer model is typically used to fuse the vectors of all keyframes, forming a "memory" vector that represents the dynamic content of the entire video.

**4. Question Answering and Generation:** The user's text question is encoded and then fused across modally with this "memory" vector. Finally, the language model part generates a summary of the entire video or an answer to a specific question based on this fused information.

In short, it **condenses a video into a few key images and their order, then allows you to understand the whole story like reading a comic strip and answer related questions.**

## 2. Code Analysis

### Key Code

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

The `analyze_video` function in this file defines the execution flow of this tool.

```
# From largetmodel/utils/tools_manager.py
class ToolsManager:
    # ...
    def analyze_video(self, args):
        """
        Analyze video file and provide content description.

        :param args: Arguments containing video path.
        :return: Dictionary with video description and path.
        """

        self.node.get_logger().info(f"Executing analyze_video() tool with args: {args}")
        try:
            video_path = args.get("video_path")
            # ... (Intelligent path fallback mechanism)

            if video_path and os.path.exists(video_path):
                # ... (Building Prompt)

                # Use a fully isolated, one-time context for video analysis to
                # ensure a plain text description.
                simple_context = [{"role": "system",
                                   "content": "You are a video description assistant. ..."}]

                result = self.node.model_client.infer_with_video(video_path,
                                                               prompt, message=simple_context)

                # ... (Processing results)
                return {
                    "description": description,
                    "video_path": video_path
                }
            # ... (Error Handling)
```

## 2. Model interface layer and frame extraction (`large_model/utils/large_model_interface.py`)

The functions in this file are responsible for processing the video files and passing them to the underlying model.

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

class model_interface:
    # ...
    def infer_with_video(self, video_path, text=None, message=None):
        """Unified video inference interface."""
        # ... (Prepare Message)
        try:
            # Determine which specific implementation to call based on
            self.llm_platform
            if self.llm_platform == 'ollama':
                response_content = self.ollama_infer(self.messages,
                video_path=video_path)
                # ... (The logic of other online platforms)
            # ...
            return {'response': response_content, 'messages': self.messages.copy()}

    def _extract_video_frames(self, video_path, max_frames=5):
        """Extract keyframes from a video for analysis."""
        try:
            import cv2
            # ... (Video reading and frame interval calculation)
            while extracted_count < max_frames:
                # ... (Looping through video frames)
                if frame_count % frame_interval == 0:
                    # ... (Save the frame as a temporary image)
                    frame_base64 = self.encode_file_to_base64(temp_path)
                    frame_images.append(frame_base64)
                # ...
            return frame_images
        # ... (Exception handling)
```

## Code Analysis

The implementation of video analysis is more complex than image analysis. It requires a key preprocessing step at the model interface layer: frame extraction.

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

- The `analyze_video` function is the entry point for video analysis. Its responsibilities are clear: accept a video file path and construct a prompt to request a description of the video content.
- It initiates the analysis process by calling the `self.node.model_client.infer_with_video` method. Like the visual understanding tool, it is completely agnostic about the underlying model details and only handles passing the "video file" and "analysis instructions."

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

- The `infer_with_video` function is the scheduler that connects the upper-level tools with the underlying model. It dispatches tasks to the corresponding implementation

- functions based on the platform configuration (`self.llm_platform`).
- Unlike processing single images, processing videos requires an additional step. The `_extract_video_frames` method demonstrates the general logic for implementing this step: it uses the `cv2` library to read the video and extract several keyframes (by default, five).
- Each extracted frame is treated as a separate image and typically encoded as a Base64 string.
- Ultimately, a request containing multiple frame image data and analysis instructions is sent to the main model. The model performs a comprehensive analysis of these consecutive images to generate a description of the entire video content.
- This "video-to-multiple-images" preprocessing is completely encapsulated in the model interface layer and is transparent to the tool layer.

In summary, the general process of video analysis is: `ToolsManager` initiates an analysis request -> `model_interface` intercepts the request and calls `_extract_video_frames` to decompose the video file into multiple keyframe images -> `model_interface` sends these images, along with analysis instructions, to the corresponding model platform according to the configuration -> the model returns a comprehensive description of the video -> the results are finally returned to `ToolsManager`. This design ensures the stability and versatility of upper-layer applications.

## 3. Practical Operations

### 3.1 Configuring the 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/largemode1/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 Configuring the Model Interface (`large_model_interface.yaml`)

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

1. Open the file in a terminal.

```
vim ~/yahboom_ws/src/largemode1/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:** Please ensure that the model specified in the configuration parameters (e.g., `llava`) can handle multimodal input.

## 3.2 Starting and Testing the Functionality

**\*\*Note:** The performance will be worse when using a model with a small number of parameters or when the memory usage is at its limit. For a better experience with this feature, please refer to the corresponding chapter in <Online Large Model (Text Interaction)>

### 1. Prepare Video File:

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

```
~/yahboom_ws/src/largemode1/resources_file/analyze_video
```

Then name the video `test_video.mp4`

### 2. Start the `largetmodel` Main Program:

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

```
ros2 launch largemode1 largemode1_control.launch.py
```

### 3. Test:

- **Wake-up:** Say into the microphone, "Hello, Xiaoya."
- **Dialogue:** After the speaker responds, you can say: `Analyze the video 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:** You should hear the response from the speaker shortly.