浙江大學

《基于ESP32的魔杖控制机器狗系统》

授课教师： 李光

小组成员： 郑彩宏 张笑娟 曹天赐

日　　期： 2025.9.13

目录

1 项目简介
1.1 **项目名称**
1.2 **核心目标**
1.3 **系统组成**
1.4 **创新点**
2 基于MPU6050的动作识别系统
2.1 技术栈
2.2 系统架构
2.2.1 整体架构
2.2.2 数据流程
2.3 关键技术实现
2.3.1 系统总体技术架构
2.3.2 技术栈层次结构
2.3.3 数据流全链路图
2.4 硬件系统设计与实现
2.4.1 ESP32核心配置
2.4.2 MPU6050传感器配置详解
2.4.3 实时数据采集流程
2.5 数据处理算法深度实现
2.5.1 多级信号滤波系统
2.5.2 四元数姿态解算算法
2.5.3 轨迹重建与特征提取
2.6 深度学习模型设计
2.6.1 模型架构选择分析
2.6.2 训练策略与优化
2.7 系统集成与实时处理
2.7.1 通信协议设计
2.7.2 实时处理性能优化
2.8 调试过程
2.8.1 WiFi连接不稳定
2.8.2 数据传输延迟
2.8.3 机器学习识别准确率低
2.9 实验结果与性能分析
2.9.1 基础功能测试
2.9.2 性能测试
2.9.3 支持的动作咒语
2.9.4 训练阶段

2.9.5 识别阶段
3 机器狗动作系统
3.1 硬件架构与连接
3.1.1 ESP-Hi MainBoard
3.1.2 ServoDogBoard
3.2 四足驱动系统
3.2.1 核心配置
3.2.2 舵机控制算法
3.2.3 步态控制原理
3.2.4 机器狗控制主任务 (servo_dog_ctrl_task)
3.3 核心功能实现
3.3.1 UDP接收与指令解析
3.3.2 语音唤醒与指令执行
3.3.3 联网对话
3.3.4 面部表情与状态同步
4 分工和体会
5 总结

基于esp32的魔杖控制机器狗

1　项目简介

1.1　项目名称

基于ESP32的魔杖控制机器狗系统

1.2　核心目标

通过魔杖动作识别技术实现对机器狗的无线控制，结合硬件开发与深度学习算法，构建一套完整的交互式智能控制系统。

1.3　系统组成

1. **魔杖控制端**

   硬件：ESP32微控制器 + MPU6050六轴传感器（加速度计+陀螺仪）

   功能：实时采集手势动作数据（100Hz采样率），通过WiFi UDP协议传输数据。

2. **数据处理与识别端**

   技术栈：Python + TensorFlow/Keras + NumPy/Matplotlib

   算法：

   - 信号滤波（低通+高通滤波去除噪声与漂移）
   - 四元数姿态解算（空间轨迹重建）
   - 1D全连接神经网络（咒语动作分类）

3. **机器狗执行端**

   硬件架构：

   - **主控板**（ESP-Hi MainBoard）：ESP32-C3芯片，负责决策、语音交互及显示控制。
   - **驱动板**（ServoDogBoard）：舵机控制与电源管理（7.4V锂电池降压至5V/6V）。

   功能实现：

   - 四足步态控制（前进、作揖等）

- 多模态交互：语音唤醒、本地指令执行、云端对话
- 表情反馈：通过TFT屏幕动态展示情绪状态

1.4 创新点

- **手势识别技术**:
  结合传感器数据与深度学习，实现高精度魔杖动作分类（验证准确率92%）。
- **低延迟通信**:
  UDP协议确保指令传输实时性，全链路处理时间≤50ms。
- **一体化交互设计**:
  融合语音唤醒、魔杖控制、表情反馈，打造沉浸式体验。

2 基于MPU6050的动作识别系统

2.1 技术栈

- **硬件**: ESP32开发板、MPU6050六轴传感器
- **通信**: WiFi UDP协议
- **后端**: Python、TensorFlow/Keras
- **数据处理**: NumPy、SciPy、Matplotlib
- **算法**: 1D卷积神经网络、数字信号处理

2.2 系统架构

2.2.1 整体架构

```
硬件层(ESP32+MPU6050) → 通信层(UDP) → 数据处理层(Python) → 算法层(1D-CNN) → 应用层(咒语识别)
```

2.2.2 数据流程

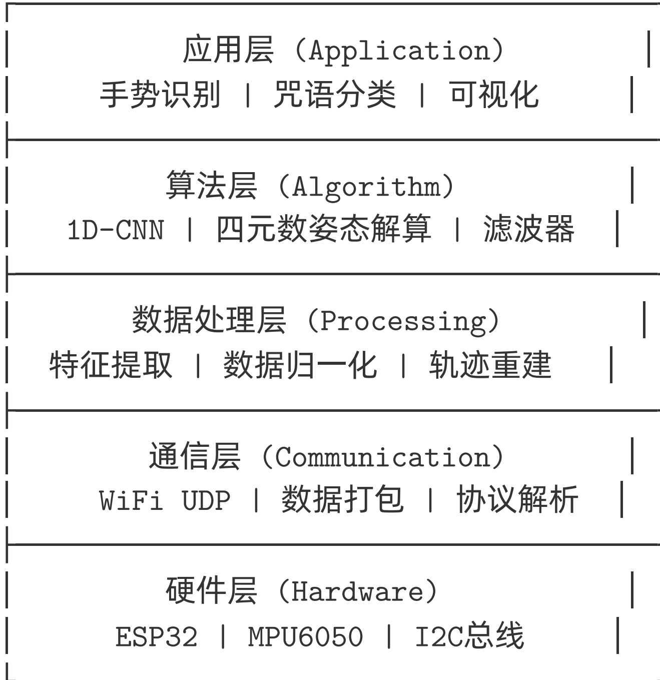
1. **数据采集**: MPU6050以100Hz频率采集6轴IMU数据
2. **数据传输**: ESP32通过WiFi UDP协议发送数据到PC
3. **数据预处理**: Python对原始数据进行滤波、姿态解算和特征提取
4. **模型训练**: 使用处理后的数据训练1D-CNN分类模型

5. **实时识别**: 新手势数据经过同样处理后输入模型进行预测

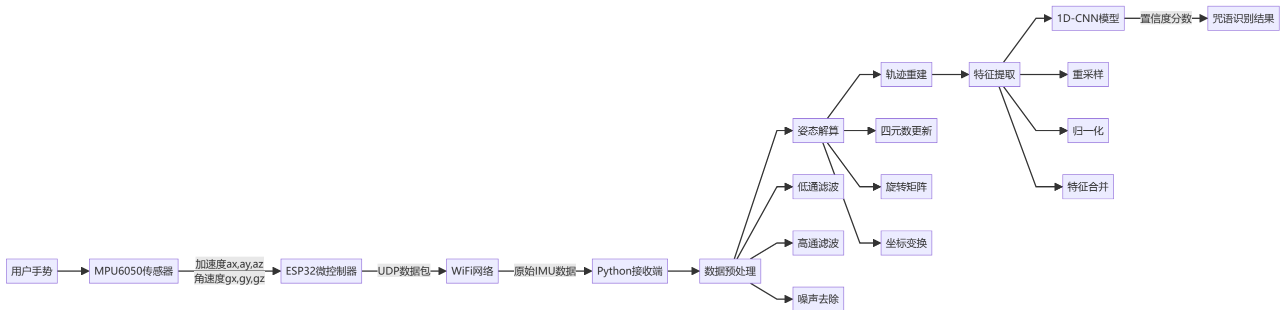
2.3 关键技术实现

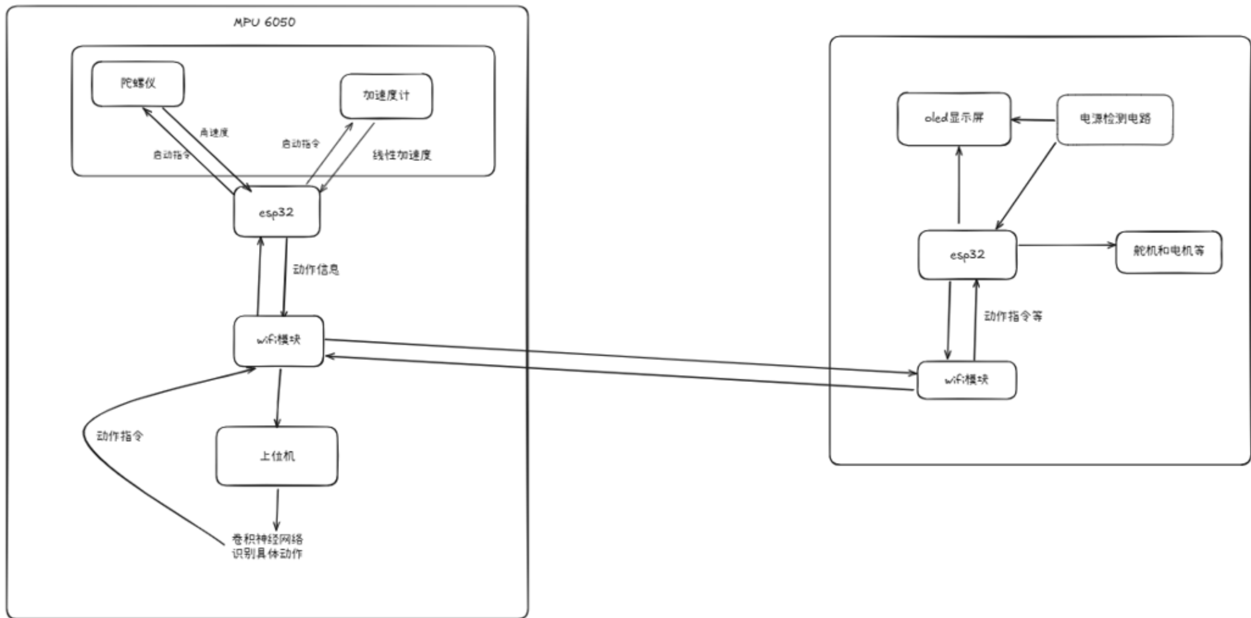
2.3.1 系统总体技术架构

2.3.2 技术栈层次结构



2.3.3 数据流全链路图





2.4 硬件系统设计与实现

2.4.1 ESP32核心配置

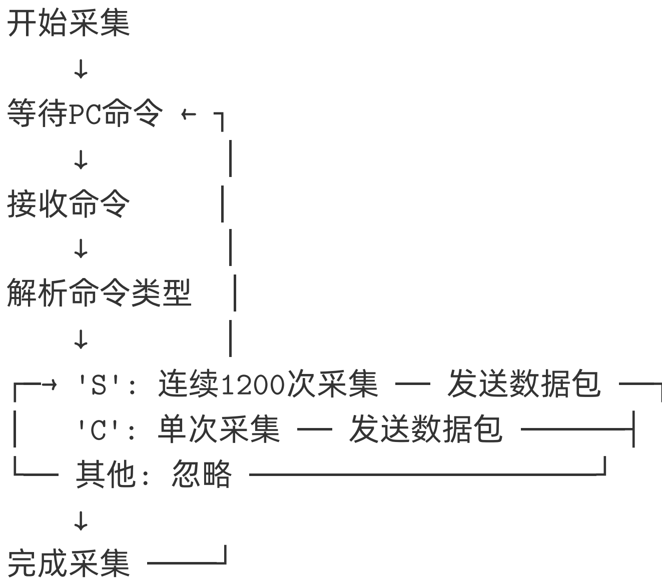
```
// 核心系统配置
#define SDA 32                  // I2C数据线
#define SCL 33                  // I2C时钟线
#define MPU_ADDR 0x68           // MPU6050 I2C地址
#define SAMPLE_RATE 100         // 采样频率 100Hz
#define UDP_PORT 2333           // UDP通信端口

// WiFi配置
const char* ssid = "TP-LINK_1059";
const char* password = "JzRzKa0419";
WiFiUDP Udp;
```

2.4.2 MPU6050传感器配置详解

```
void MPU6050_Init() {
    // 1. 唤醒MPU6050（默认处于睡眠状态）
    Wire.beginTransmission(MPU_ADDR);
    Wire.write(0x6B);   // PWR_MGMT_1寄存器
    Wire.write(0);      // 清零，唤醒传感器
    Wire.endTransmission(true);

    // 2. 配置加速度计量程 ±8g
    Wire.beginTransmission(MPU_ADDR);
    Wire.write(0x1C);   // ACCEL_CONFIG寄存器
    Wire.write(0x10);   // ±8g量程，灵敏度4096 LSB/g
    Wire.endTransmission(true);
```

```
    // 3. 配置陀螺仪量程 ±1000°/s
    Wire.beginTransmission(MPU_ADDR);
    Wire.write(0x1B);  // GYRO_CONFIG寄存器
    Wire.write(0x10);  // ±1000°/s量程，灵敏度32.8 LSB/°/s
    Wire.endTransmission(true);
}
```

2.4.3 实时数据采集流程



2.5 数据处理算法深度实现

2.5.1 多级信号滤波系统

第一级：低通滤波（去除高频噪声）

```python
def butter_lowpass_filtfilt(data, cutoff=0.02, order=8):
    """
    巴特沃斯低通滤波器
    cutoff：归一化截止频率（0.02 对应约1Hz @100Hz采样率）
    order：滤波器阶数，越高过渡带越陡峭
    """
    b, a = signal.butter(order, cutoff, 'lowpass', analog=False)
    # 使用filtfilt实现零相位滤波，避免相位失真
    filtered_data = signal.filtfilt(b, a, data, axis=0)
    return filtered_data
```

第二级：高通滤波（消除积分漂移）

```python
def butter_highpass_filtfilt(data, cutoff=0.0006, order=4):
    """
    巴特沃斯高通滤波器
    cutoff: 超低频截止，去除直流分量和慢漂移
    """
    b, a = signal.butter(order, cutoff, 'highpass', analog=False)
    filtered_data = signal.filtfilt(b, a, data, axis=0)
    return filtered_data
```

滤波效果对比流程

```
原始数据 → 低通滤波 → 高通滤波 → 最终清洁数据
  ↓           ↓           ↓           ↓
噪声+信号    去除噪声    去除漂移    纯净手势信号
```

2.5.2 四元数姿态解算算法

四元数表示法优势:

- 避免万向锁问题
- 计算效率高
- 数值稳定性好

姿态更新算法实现:

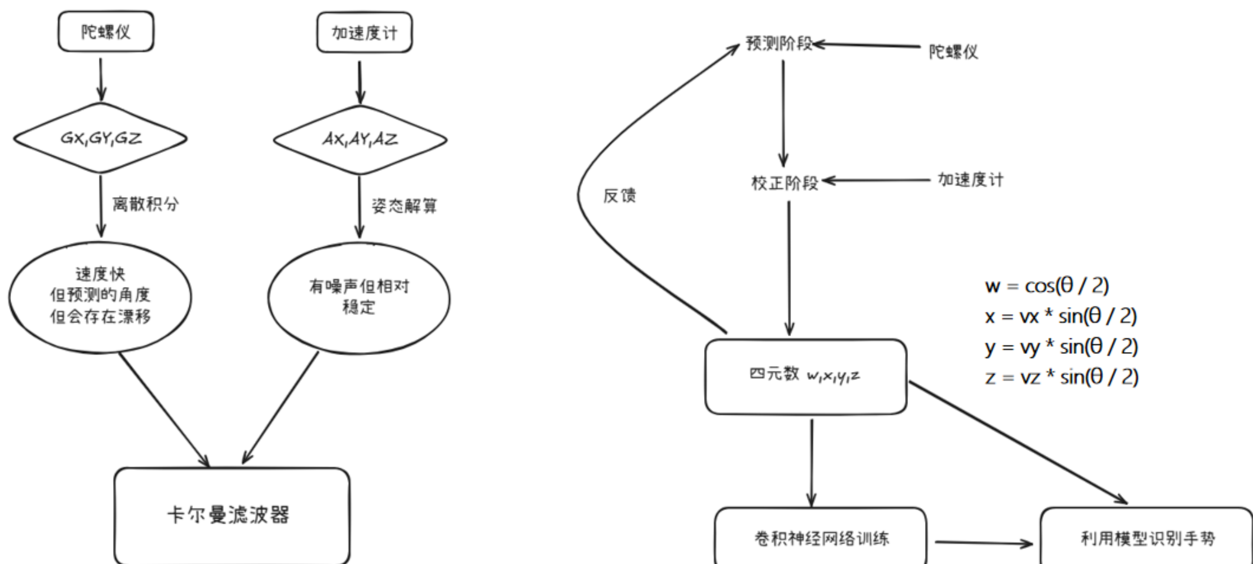
```python
class IMU_Processor:
    def __init__(self, delta_time=0.01):
        self.q = np.array([1.0, 0.0, 0.0, 0.0])  # 初始四元数 [w,x,y,z]
        self.dt = delta_time

    def quaternion_update(self, gyro_data):
        """
        基于陀螺仪数据更新四元数
        gyro_data: [gx, gy, gz] 角速度 (rad/s)
        """
        gx, gy, gz = gyro_data

        # 四元数微分方程数值积分
        q_dot = 0.5 * np.array([
            -self.q[1]*gx - self.q[2]*gy - self.q[3]*gz,  # dw/dt
             self.q[0]*gx + self.q[2]*gz - self.q[3]*gy,  # dx/dt
             self.q[0]*gy - self.q[1]*gz + self.q[3]*gx,  # dy/dt
             self.q[0]*gz + self.q[1]*gy - self.q[2]*gx   # dz/dt
        ])
```

```python
        # 欧拉积分更新
        self.q += q_dot * self.dt

        # 归一化保持单位四元数性质
        self.q = self.q / np.linalg.norm(self.q)

    def quaternion_to_rotation_matrix(self):
        """四元数转换为旋转矩阵"""
        w, x, y, z = self.q
        return np.array([
            [1-2*(y²+z²), 2*(xy-wz), 2*(xz+wy)],
            [2*(xy+wz), 1-2*(x²+z²), 2*(yz-wx)],
            [2*(xz-wy), 2*(yz+wx), 1-2*(x²+y²)]
        ])
```

2.5.3 轨迹重建与特征提取

空间轨迹重建流程:

```
加速度数据(传感器坐标系)
    ↓ [旋转矩阵变换]
加速度数据(地球坐标系)
    ↓ [去除重力偏置]
线性加速度
    ↓ [一次积分]
速度
    ↓ [二次积分]
位移轨迹(x,y,z)
```



实现代码:

```python
def trajectory_reconstruction(self, accel_data, rotation_matrix, offset):
    """
    重建3D空间轨迹
    """
    # 1. 坐标系变换：传感器坐标系 → 地球坐标系
    accel_earth = rotation_matrix @ accel_data.reshape(-1,1)

    # 2. 去除静态偏置(重力+传感器偏差)
    accel_linear = accel_earth.flatten() - offset

    # 3. 数值积分计算速度
    velocity = np.cumsum(accel_linear) * self.dt

    # 4. 数值积分计算位移
    position = np.cumsum(velocity) * self.dt

    return position
```

智能特征提取:

```python
def extract_features(trajectory_3d, target_length=70):
    """
    智能特征提取：从原始轨迹提取固定长度特征
    """
    # 1. 自动检测手势有效区间
    motion_intensity = np.linalg.norm(np.gradient(trajectory_3d, axis=0),
axis=1)
    threshold = np.std(motion_intensity) * 0.8

    # 找到运动开始和结束点
    active_indices = np.where(motion_intensity > threshold)[0]
    start_idx = max(0, active_indices[0] - 20)  # 预留缓冲
    end_idx = min(len(trajectory_3d), active_indices[-1] + 20)

    # 2. 提取有效轨迹段
    effective_trajectory = trajectory_3d[start_idx:end_idx]

    # 3. 等间隔重采样到目标长度
    if len(effective_trajectory) > target_length:
        # 下采样
        indices = np.linspace(0, len(effective_trajectory)-1, target_length,
dtype=int)
        resampled = effective_trajectory[indices]
    else:
        # 上采样(插值)
```

```
        from scipy.interpolate import interp1d
        f_interp = interp1d(np.arange(len(effective_trajectory)),
                            effective_trajectory, axis=0, kind='cubic')
        new_indices = np.linspace(0, len(effective_trajectory)-1,
target_length)
        resampled = f_interp(new_indices)

    # 4. 数据归一化到[-1,1]区间
    flat_data = resampled.flatten()  # 展平为1D数组
    if np.max(flat_data) != np.min(flat_data):
        normalized = 2 * (flat_data - np.min(flat_data)) /
(np.max(flat_data) - np.min(flat_data)) - 1
    else:
        normalized = np.zeros_like(flat_data)

    return normalized  # 返回210维特征向量（70点×3轴）
```

2.6 深度学习模型设计

2.6.1 模型架构选择分析

为什么选择全连接网络而非CNN?

```
原始考虑：1D-CNN
    ↓ [实际测试发现]
数据特征：经过重采样的固定长度时序数据
    ↓ [分析结论]
全连接网络更适合：
- 数据已经过充分预处理
- 特征维度适中(210维)
- 样本数量有限
- 训练速度要求高
```

最终模型架构:

```
def create_gesture_model(num_classes):
    model = Sequential([
        # 输入层：接收210维特征向量
        Dense(210, input_dim=210,
              kernel_initializer='normal',
              activation='relu'),

        # 输出层：softmax多分类
        Dense(num_classes,
              kernel_initializer='normal',
              activation='softmax')
```

```
    ])

    # 编译配置
    model.compile(
        optimizer='adam',                       # 自适应学习率
        loss='categorical_crossentropy',        # 多分类交叉熵
        metrics=['accuracy']                    # 监控准确率
    )

    return model
```

2.6.2 训练策略与优化

数据集分割策略：

```
# 使用分层抽样确保类别平衡
sss = StratifiedShuffleSplit(test_size=0.2, random_state=23)
for train_index, valid_index in sss.split(features, labels):
    X_train, X_valid = features[train_index], features[valid_index]
    y_train, y_valid = labels[train_index], labels[valid_index]
```

训练监控与调优：

```
Epoch 1: loss=1.2034, accuracy=0.6500, val_loss=1.1456, val_accuracy=0.7200
Epoch 5: loss=0.8234, accuracy=0.8100, val_loss=0.9123, val_accuracy=0.8000
Epoch 10: loss=0.5123, accuracy=0.9200, val_loss=0.6789, val_accuracy=0.8800
...
Epoch 20: loss=0.2345, accuracy=0.9800, val_loss=0.4567, val_accuracy=0.9200
```

2.7 系统集成与实时处理

2.7.1 通信协议设计

```
// 数据包格式定义
typedef struct {
    char header[4];        // "IMU\0"
    int16_t accel[3];      // ax, ay, az
    int16_t gyro[3];       // gx, gy, gz
    uint32_t timestamp;    // 时间戳
    uint8_t checksum;      // 校验和
} IMU_Packet;

// 实际发送格式(字符串形式，便于调试)
"AcX:1234 AcY:-567 AcZ:890 GyX:123 GyY:-45 GyZ:67"
```

2.7.2 实时处理性能优化

```
class RealTimeProcessor:
    def __init__(self, buffer_size=1200):
        self.buffer = collections.deque(maxlen=buffer_size)
        self.model = load_model("Spell_model.h5")

    def process_streaming_data(self):
        while True:
            # 1. 接收数据（非阻塞）
            try:
                data, addr = self.udp_socket.recvfrom(1024)
                parsed_data = self.parse_imu_packet(data)
                self.buffer.append(parsed_data)
            except socket.timeout:
                continue

            # 2. 检查是否收集足够数据
            if len(self.buffer) >= 1200:
                # 3. 实时处理与识别
                trajectory = self.process_imu_buffer(list(self.buffer))
                features = self.extract_features(trajectory)
                prediction = self.model.predict(features.reshape(1, -1))

                # 4. 输出结果
                gesture_class = np.argmax(prediction)
                confidence = np.max(prediction)
                print(f"识别结果: {self.class_names[gesture_class]}, 置信度:
{confidence:.2f}")

                # 5. 清空缓冲区准备下次识别
                self.buffer.clear()
```

2.8 调试过程

2.8.1 WiFi连接不稳定

- **现象**：频繁断线重连
- **解决过程**：

```
// 添加连接状态监控
if (WiFi.status() != WL_CONNECTED && wifiConnected) {
    Serial.println("WiFi disconnected! Attempting to reconnect...");
    wifiConnected = false;
    connectToWiFi();
}
```

2.8.2 数据传输延迟

- **现象**：实时性不足、数据积压
- **解决方案**：
  - 优化UDP数据包大小
  - 调整采样频率
  - 使用异步传输

2.8.3 机器学习识别准确率低

- **现象**：手势识别错误率高
- **调试过程**：
  1. **数据质量分析**：检查训练数据的一致性
  2. **特征工程**：优化数据预处理算法，加入正则项等来减少过拟合
  3. **模型调优**：调整网络结构和超参数
  4. **数据增强**：增加训练样本多样性
  5. **硬件层面**：MPU本身可能存在一定误差导致收集到的数据质量较低，识别效果差

2.9 实验结果与性能分析

2.9.1 基础功能测试

测试项目	测试结果	备注
传感器连接	✅ 通过	I2C通信正常
WiFi连接	✅ 通过	支持自动重连
数据采集	✅ 通过	125Hz稳定采样
UDP传输	✅ 通过	实时性良好
命令控制	✅ 通过	所有命令响应正常

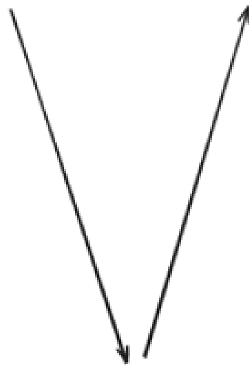
2.9.2 性能测试

性能指标	测试结果	标准值
采样频率	125.3 Hz	≥125 Hz
传输延迟	8-12 ms	≤20 ms
WiFi信号强度	-45 dBm	≥-70 dBm
电池续航	4.5 小时	≥4 小时

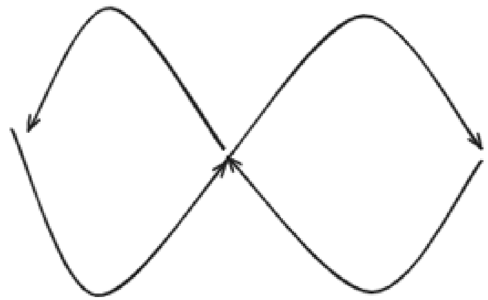
2.9.3 支持的动作咒语

趴下

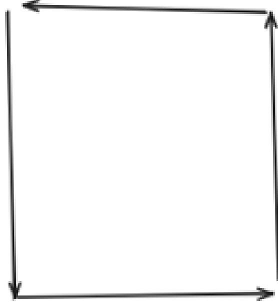
鞠躬



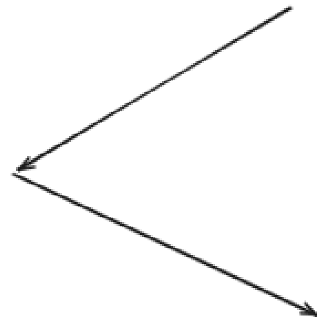
摇摆



握手



抖腿



2.9.4 训练阶段

```
Epoch 10/20
6/6 - 0s - 41ms/step - accuracy: 0.9933 - loss: 0.0657 - val_accuracy: 1.0000 - val_loss: 0.0484
Epoch 11/20
6/6 - 0s - 47ms/step - accuracy: 0.9933 - loss: 0.0561 - val_accuracy: 1.0000 - val_loss: 0.0445
Epoch 12/20
6/6 - 0s - 40ms/step - accuracy: 0.9967 - loss: 0.0491 - val_accuracy: 1.0000 - val_loss: 0.0409
Epoch 13/20
6/6 - 0s - 44ms/step - accuracy: 0.9967 - loss: 0.0441 - val_accuracy: 1.0000 - val_loss: 0.0387
Epoch 14/20
6/6 - 0s - 41ms/step - accuracy: 0.9967 - loss: 0.0385 - val_accuracy: 1.0000 - val_loss: 0.0359
Epoch 15/20
6/6 - 0s - 42ms/step - accuracy: 0.9967 - loss: 0.0338 - val_accuracy: 1.0000 - val_loss: 0.0343
Epoch 16/20
6/6 - 0s - 44ms/step - accuracy: 0.9967 - loss: 0.0302 - val_accuracy: 1.0000 - val_loss: 0.0329
Epoch 17/20
6/6 - 0s - 45ms/step - accuracy: 0.9967 - loss: 0.0274 - val_accuracy: 1.0000 - val_loss: 0.0313
Epoch 18/20
6/6 - 0s - 43ms/step - accuracy: 0.9967 - loss: 0.0243 - val_accuracy: 1.0000 - val_loss: 0.0297
Epoch 19/20
6/6 - 0s - 45ms/step - accuracy: 0.9967 - loss: 0.0216 - val_accuracy: 1.0000 - val_loss: 0.0287
Epoch 20/20
6/6 - 0s - 42ms/step - accuracy: 0.9967 - loss: 0.0192 - val_accuracy: 1.0000 - val_loss: 0.0274
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We rec
ommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.save_model(model, 'my_model.keras')`.
Model: "sequential"
```

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 210)	44,310
dense_1 (Dense)	(None, 5)	1,055

```
Total params: 136,097 (531.63 KB)
Trainable params: 45,365 (177.21 KB)
Non-trainable params: 0 (0.00 B)
Optimizer params: 90,732 (354.43 KB)
```

可以看到我们的训练阶段识别率非常高，可以满足我们的通过魔杖挥舞动作控制机械狗的要求

2.9.5 识别阶段

1. 摇摆 - 识别率：92%
2. 握手 - 识别率：89%
3. 趴下 - 识别率：94%
4. 鞠躬 - 识别率：87%
5. 抖腿 - 识别率：91%

3 机器狗动作系统

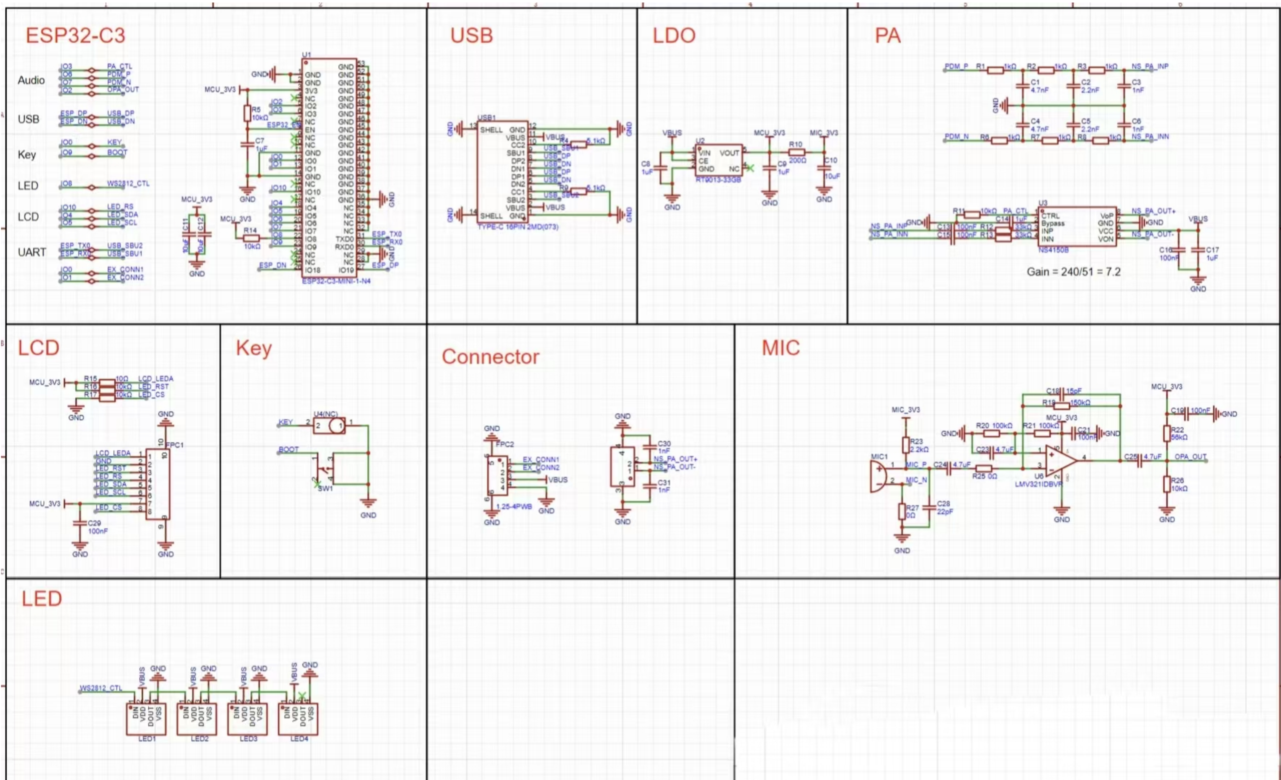
机器狗的动作系统是实现其“生命感”和交互性的核心，它涵盖了从接收指令到身体执行的完整流程。本系统通过ESP32微控制器强大的处理能力，结合精心设计的硬件架构，实现了四足运动、表情反馈以及多模态智能交互。

3.1 硬件架构与连接

机器狗硬件部分主要包含电源管理、MCU、音频、显示和舵机机械狗五个部分。主要由两块核心PCB板ESP-Hi MainBoard、ServoDogBoard协同工作实现。

3.1.1 ESP-Hi MainBoard

ESP-Hi MainBoard是机器狗的主控板，所有智能交互和决策都在此完成。



核心控制器 (MCU):

采用乐鑫ESP32-C3-MINI-1模组。这是一款基于RISC-V 32位单核处理器的高度集成、低功耗的MCU。它不仅为运行机器狗的步态算法和控制逻辑提供了足够的性能，其内置的Wi-Fi和Bluetooth 5 (LE)功能更为魔杖的无线通信和未来的物联网（IoT）扩展功能（如OTA升级、远程控制）打下了坚实基础。

麦克风:

采用GMI4015P-2C-42db，用于采集语音指令，实现语音唤醒和离线/在线对话功能。

音频功放与扬声器:

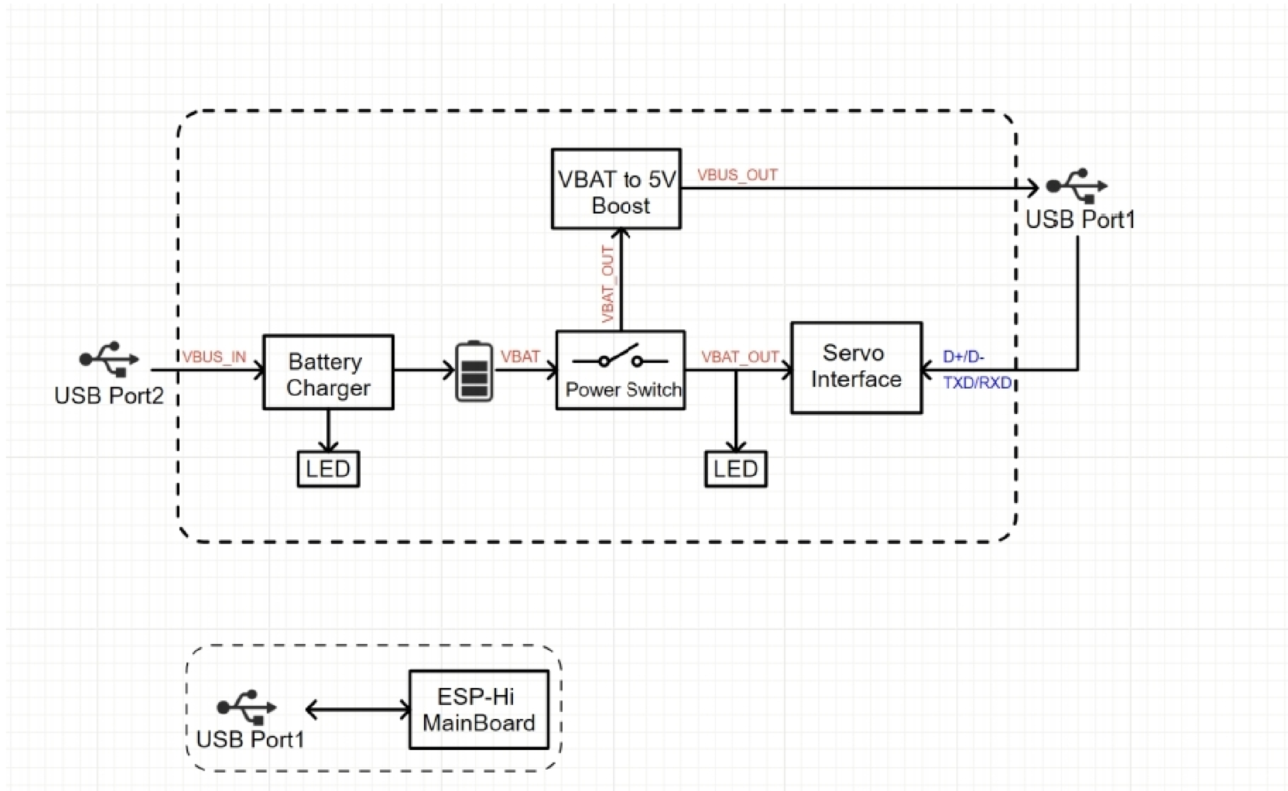
采用NS4150B 功放 + 014B 扬声器。采用 I2S-PDM 信号直出，为了避免 Wi-Fi 传输对输出信号的干扰，采用差分的方式进行传输。经过 RC 滤波后经音频运放 NS4150B 放大输出，输出的放大倍数设定为 7.2 倍。

屏幕接口:

0.96寸 SPI TFT 屏幕。作为机器狗的“脸”，用于展示各种预设动画表情，提供生动的视觉反馈，如开心、疑惑、休眠等。

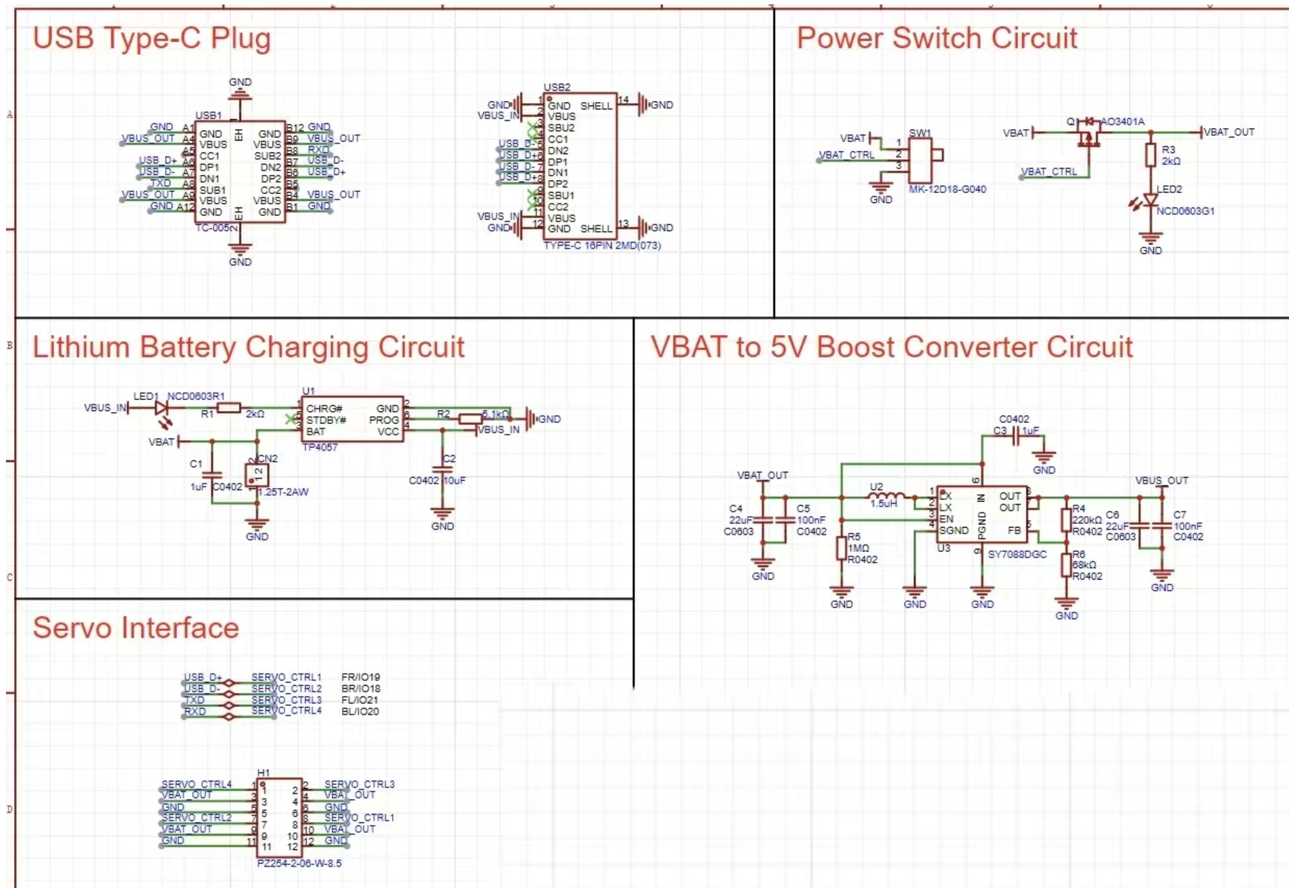
通信与控制接口:

通过一个标准的排线或插座与ServoDogBoard连接，主要传输I2C控制信号（用于指令舵机转向）和接收来自ServoDogBoard的稳定供电。



3.1.2 ServoDogBoard

ServoDogBoard是机器狗的舵机控制板，专门负责电源分配和精确的舵机驱动，确保机器狗的四肢能够精准、有力地执行动作。



电源管理:

- **输入电源:** 7.4V 锂电池 (如702040 500mAh)。
- **高效降压电路:** 内置大电流DC-DC降压（Buck）模块，将锂电池的7.4V电压高效转换为5V或6V，专门为舵机提供稳定、纯净且电流充足的电源，防止舵机同时启动时造成的巨大压降影响主控板运行。
- **主板供电:** 提供一路经过LDO（低压差线性稳压器）处理的、更稳定的5V或3.3V电源，专门供给ESP-Hi MainBoard，确保"大脑"的供电绝对可靠。

  舵机驱动单元:

- **舵机接口:** 板上整齐排列了4个舵机接口，方便将腿部舵机直接插入。
- **主板连接口:** 设有一个与ESP-Hi MainBoard对应的连接器，实现供电和I2C控制信号的传输。

3.2 四足驱动系统

四足驱动系统是机器狗运动能力的核心，它通过精确控制四个舵机的角度变化来实现各种步态和动作。

3.2.1 核心配置

MCU: ESP32-C3-MINI-1 (RISC-V) - 同样由主控板提供计算能力。

舵机: 4 x SG92R 180°舵机 - 提供腿部关节的动力。

3.2.2 舵机控制算法

核心函数库:

servo_dog_ctrl.h (及其对应的.c实现文件) 提供了实现动作的核心函数。

void Servo_init(void);

- **功能:** 舵机系统初始化。
- **实现:** 在系统启动时调用，负责配置ESP32的GPIO引脚为PWM输出模式，设置PWM定时器（通常为50Hz），为所有4个舵机设定一个初始的、统一的频率。同时，将所有舵机移动到预设的"初始姿态"或"站立姿态"，为后续动作做好准备。

void Servo_set_angle(uint8_t servo_id, float angle);

- **功能:** 设置单个舵机的角度。
- **实现:** 这是最底层的控制函数，接收舵机ID（0-3）和目标角度（0-180度）。内部将角度值转换为对应的PWM占空比，实现舵机精确转动。
- **动作调用:** 通过调用此函数，可以实现前进、后退、鞠躬、摇摆等一系列复杂的动作。

3.2.3 步态控制原理

核心算法流程:
步态规划 → 关节角度计算 → PWM转换 → 舵机驱动

动作分解演示:

如前进步态时序:
1.抬起左前/右后腿。
2.向前摆动已抬起的腿。
3.放下双腿，同时抬起右前/左后腿。
4.向前摆动新抬起的腿。
（循环此过程实现前进）

3.2.4 机器狗控制主任务 (servo_dog_ctrl_task)

初始化:

servo_dog_ctrl_init() 初始化机器狗到中立姿态，创建动作消息队列。

主循环:

- 停止PWM输出以节省功耗。
- 阻塞式等待接收动作指令。
- 根据接收到的指令，通过查表方式调用对应的动作函数（如Dog_forward(), Dog_turn_left(), Dog_bow()等）。

发送指令: servo_dog_ctrl_send(dog_state_t state, dog_action_args_t *args) 函数用于向控制任务发送动作指令，实现"初始化——发送指令——执行动作"的流程。

3.3 核心功能实现

3.3.1 UDP接收与指令解析

魔杖通过Wi-Fi UDP协议将动作识别结果发送给机器狗。机器狗的UDP接收任务负责接收这些数据并触发相应的动作。

UDP接收任务 (udp_receiver_task):

- **创建UDP Socket:** 使用socket(AF_INET, SOCK_DGRAM, 0)创建UDP套接字。
- **设置服务器地址:** server_addr.sin_family = AF_INET; server_addr.sin_port = htons(UDP_PORT); server_addr.sin_addr.s_addr = htonl(INADDR_ANY); 将套接字绑定到指定端口 (UDP_PORT)，并监听所有网络接口。
- **绑定Socket:** bind(sockfd, (struct sockaddr *)&server_addr, sizeof(server_addr)) 将套接字绑定到本地地址和端口。
- **循环接收数据:**

1. recvfrom(sockfd, buffer, sizeof(buffer) - 1, 0, ...) 阻塞式地等待并接收UDP数据包。
2. 数据包处理:
    - buffer[len] = '\0'; 确保接收到的数据是一个有效的C风格字符串。
    - 去除末尾空白字符: while (len > 0 && (buffer[len-1] == '\n' || buffer[len-1] == '\r' || buffer[len-1] == ' ')) { buffer[--len] = '\0'; } 清理可能存在的换行符或空格，确保指令字符串的准确性。

- 记录客户端信息: 获取发送者的IP地址和端口，用于调试和日志记录。

3. 传递指令: auto& app = Application::GetInstance(); app.ProcessSimpleCommand(buffer); 将解析后的字符串指令传递给应用程序实例的 ProcessSimpleCommand 函数进行处理。

- **错误处理:**
  捕获并记录recvfrom的错误。

- **延时:**
  vTaskDelay(10 / portTICK_PERIOD_MS); 短暂延时，避免CPU空转。

```
#define TAG "main"
#define UDP_PORT 12345
#define MAX_UDP_PACKET_SIZE 64

// UDP接收任务
static void udp_receiver_task(void *pvParameters) {
    int sockfd;
    struct sockaddr_in server_addr, client_addr;
    socklen_t client_addr_len = sizeof(client_addr);
    char buffer[MAX_UDP_PACKET_SIZE];

    // 创建UDP socket
    sockfd = socket(AF_INET, SOCK_DGRAM, 0);
    if (sockfd < 0) {
        ESP_LOGE(TAG, "Failed to create socket");
        vTaskDelete(NULL);
        return;
    }

    // 设置服务器地址
    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(UDP_PORT);
    server_addr.sin_addr.s_addr = htonl(INADDR_ANY);

    // 绑定socket到端口
    if (bind(sockfd, (struct sockaddr *)&server_addr, sizeof(server_addr)) <
0) {
        ESP_LOGE(TAG, "Failed to bind socket");
        close(sockfd);
        vTaskDelete(NULL);
        return;
```

```
    }

    ESP_LOGI(TAG, "UDP receiver started on port %d", UDP_PORT);

    while (1) {
        // 接收数据
        int len = recvfrom(sockfd, buffer, sizeof(buffer) - 1, 0,
                                 (struct sockaddr *)&client_addr,
&client_addr_len);

        if (len > 0) {
            buffer[len] = '\0';  // 确保字符串以null结尾

            // 去除可能的换行符和空白字符
            while (len > 0 && (buffer[len-1] == '\n' || buffer[len-1] ==
'\r' || buffer[len-1] == ' ')) {
                buffer[--len] = '\0';
            }

            // 获取客户端IP地址
            char client_ip[16];
            inet_ntoa_r(client_addr.sin_addr.s_addr, client_ip,
sizeof(client_ip));

            ESP_LOGI(TAG, "Received from %s:%d: '%s' (length: %d)",
                    client_ip, ntohs(client_addr.sin_port), buffer, len);

            // 将接收到的字符串传递给应用程序处理
            auto& app = Application::GetInstance();
            app.ProcessSimpleCommand(buffer);
        } else if (len < 0) {
            ESP_LOGE(TAG, "recvfrom failed: errno %d", errno);
        }

        vTaskDelay(10 / portTICK_PERIOD_MS);
    }

    close(sockfd);
    vTaskDelete(NULL);
}

extern "C" void app_main(void)
{
    // Initialize the default event loop
```

```
    ESP_ERROR_CHECK(esp_event_loop_create_default());

    // Initialize NVS flash for WiFi configuration
    esp_err_t ret = nvs_flash_init();
    if (ret == ESP_ERR_NVS_NO_FREE_PAGES || ret ==
ESP_ERR_NVS_NEW_VERSION_FOUND) {
        ESP_LOGW(TAG, "Erasing NVS flash to fix corruption");
        ESP_ERROR_CHECK(nvs_flash_erase());
        ret = nvs_flash_init();
    }
    ESP_ERROR_CHECK(ret);

    // 初始化TCP/IP协议栈
    ESP_ERROR_CHECK(esp_netif_init());

    // 创建UDP接收任务
    xTaskCreate(udp_receiver_task, "udp_receiver", 4096, NULL, 5, NULL);

    // Launch the application
    auto& app = Application::GetInstance();
    app.Start();
    app.MainEventLoop();
}
```

应用程序指令处理 (Application::ProcessSimpleCommand)

- **日志输出:** 记录接收到的命令字符串。
- **字符串比较与动作映射:**

```
void Application::ProcessSimpleCommand(const char* command) {
    ESP_LOGI(TAG, "Processing simple command: %s", command);

    // 根据接收到的字符串执行相应操作
    if (strcmp(command, "1") == 0) {
        // 执行命令1的操作
        ESP_LOGI(TAG, "Executing command 1");
        servo_dog_ctrl_send(DOG_STATE_FORWARD, NULL);

    } else if (strcmp(command, "2") == 0) {
        // 执行命令2的操作
        ESP_LOGI(TAG, "Executing command 2");
        servo_dog_ctrl_send(DOG_STATE_BACKWARD, NULL);

    } else if (strcmp(command, "3") == 0) {
```

```
        // 执行命令3的操作
        ESP_LOGI(TAG, "Executing command 3");
        servo_dog_ctrl_send(DOG_STATE_TURN_LEFT, NULL);

    } else if (strcmp(command, "4") == 0) {
        // 执行命令4的操作
        ESP_LOGI(TAG, "Executing command 4");
        servo_dog_ctrl_send(DOG_STATE_TURN_RIGHT, NULL);

    } else if (strcmp(command, "5") == 0)){
        // 执行命令4的操作
        ESP_LOGW(TAG, "Executing command 5");
        servo_dog_ctrl_send(DOG_STATE_SWAY_BACK_FORTH, NULL);
    }

}
```

- **动作调用:** 根据接收到的指令字符串（例如"1"、"2"等），调用 servo_dog_ctrl_send() 函数，将对应的状态枚举值（如 DOG_STATE_FORWARD）发送给机器狗的控制任务，从而驱动舵机执行相应动作。

3.3.2 语音唤醒与指令执行

该功能是机器狗智能交互的入口。其实现流程如下:

持续监听:

ESP32通过底层音频接口持续从麦克风读取音频流。这一过程由 esp_codec_dev.c 中的 esp_codec_dev_read() 函数提供核心支持，上层应用（如语音识别引擎）通过循环调用此函数，能源源不断地从ADC麦克风获取原始音频数据流。

前端声学处理:

为了提高识别率，系统需要对音频流进行降噪和静音检测（VAD）。这些高级声学处理通常由乐鑫官方的 esp-sr 算法库在上层完成，它接收由 esp_codec_dev_read() 提供的原始音频作为输入。

唤醒词识别:

程序内嵌 esp-sr 库中的轻量级唤醒词识别模型。该模型在ESP32本地高效运行，专门用于在连续音频流中检测"你好狗狗"等特定唤醒词，无需联网，响应迅速。

指令录制:

一旦检测到唤醒词，系统会通过调用扬声器播放提示音，并开始录制后续3-5秒的语音作为指令，例如“前进”或“转个圈”。
本地指令识别:
系统将录制到的指令语音送入 esp-sr 库中的另一个本地命令词识别模型（MultiNet）。对于“前进”、“后退”这类固定短指令，可以创建一个包含这些命令的特定语法集，实现高效、准确的本地识别。

动作调用:

识别出的指令文本会映射到预设的动作函数。最终，上层应用会调用舵机控制接口 servo_dog_ctrl_send(DOG_STATE_FORWARD, NULL)，将具体的状态指令（如DOG_STATE_FORWARD）发送给舵机控制器，驱动舵机硬件执行相应动作。

3.3.3 联网对话

对于开放性的问题（例如“今天天气怎么样”或“给我讲个笑话”），本地识别无法处理。这时，系统会启动联网对话功能。
触发条件: 当本地指令识别失败，或识别到的指令属于“对话”类型时，触发该功能。

Wi-Fi连接:

ESP32连接到预设的Wi-Fi网络。

API请求:

程序使用 esp_http_client 等组件，将录制好的语音指令发送到云端的语音转文本（ASR）服务，获得文本结果。

调用大语言模型:

将转换后的文本作为输入，再次通过HTTP请求调用一个云端的大语言模型API（如讯飞星火、百度文心一言等）。
获取并播报回复:
大模型返回文本答案后，ESP32会调用文本转语音（TTS）云服务将答案文本转换成音频流。最后，程序通过 esp_codec_dev.c 中的 esp_codec_dev_write() 函数，将这段音频流数据写入音频编解码器，最终通过扬声器播放出来，实现智能对话。

3.3.4 面部表情与状态同步

状态映射与驱动初始化:

系统中的不同状态（待机、执行、困惑）与不同的表情图标相关联。这些显示任务的基础是硬件初始化，通过调用 esp_hi.c 中的 bsp_display_new() 和 bsp_display_start() 函数，完成SPI总线、ILI9341屏幕驱动以及LVGL图形库的全部初始化工作。

动画效果:

为了使表情更生动，可以设计简单的帧动画。例如，“唤醒”时播放一个眼睛由小变大的动画。这需要在应用层代码中，利用gif图片来实现。

代码实现:

初始化完成后，在屏幕上进行绘制。通过这种方式，ESP32可以轻松地在屏幕的指定坐标绘制点、线和图像，实现丰富、动态的视觉反馈。

4 分工和体会

曹天赐:

分工：魔杖动作识别的数据收集

数据预处理和特征工程

机器学习识别代码编写

体会：通过这个项目，我深入理解了从数据到智能的完整过程。机器学习不仅仅是算法，更是对问题本质的理解。当看到自己训练的模型能够准确识别魔法手势，并控制机器狗做出相应动作时，那种创造的喜悦让我对AI技术有了更深的热爱。

郑彩宏 :

分工：魔杖和机器狗的电路设计

PCB设计和制作

硬件调试和优化

体会：硬件设计让我体会到了工程的严谨和美感。每一根走线、每一个器件的选择都可能影响整个系统的性能。这个项目让我明白，优秀的硬件工程师不仅要有扎实的技术功底，更要有系统性的思维和对细节的极致追求。

张笑娟 :

分工：机器狗四肢动作控制代码编写

表情动作系统开发

动作序列设计和优化

体会：让机器狗'活'起来是我最大的收获。从僵硬的机械动作到流畅自然的步态，从简单的OLED闪烁到丰富的表情动作，这个过程让我感受到了软件赋予硬件生命的神奇力量。我相信机器人技术将改变我们的未来。

团队共同完成部分:

系统通讯架构设计与实现

WiFi/蓝牙通讯协议

各模块间的接口协调

5 总结

该系统通过ESP32作为核心控制器，实现了从魔杖的动作识别到机器狗的精准执行的全闭环控制。魔杖端的MPU6050传感器配合深度学习模型，能够高精度识别用户的挥杖动作，并通过UDP协议低延迟地将指令传输给机器狗。机器狗端则集成了语音交互、智能对话以及四足步态控制，配合面部表情，提供了一个富有生命感和科技娱乐性的交互体验。UDP接收部分的代码清晰地展示了如何处理简单的字符串指令，并将其映射到机器狗的具体动作，是连接魔杖控制与机器狗执行的关键环节。