

# IoT lab1：基于TinyML的设备运动状态实时识别实验

---

3220102866 陈奕萱

## 1 实验目的和要求

---

- 掌握如何在资源有限的嵌入式设备进行模型推理
- 掌握自主创建简单模型的能力
- 掌握模型数据集准备，模型训练和模型存储等技术
- 掌握TinyML工作原理
- 使用神经网络识别设备运动状态并根据运动状态展示不同LED颜色的功能

## 2 实验内容

---

设备运动状态实时识别实验：

- 通过GPIO控制LED灯状态，展示不同颜色
- IMU惯性传感器数据集收集，训练测试样本准备
- 模型构造，模型训练，模型测试，生成可部署的模型文件
- 设备模型导入，模型实时识别设备运动状态，根据运动状态展示不同灯颜色

## 3 实验背景

---

- RIOT操作系统
- TinyML

## 4 主要仪器设备

---

- PC
- ESP32-WROOM-32、MPU6050惯性传感器、LED RGB灯。

## 5 实验题目简答

---

### 5.1 想象一下,TinyML在生活中的应用场景有哪些

智能家居：

- **语音控制设备**：使用TinyML在智能家居设备上实现本地语音识别，支持离线控制，如灯光、恒温器、安防系统等，提升隐私性。
- **安防监控**：通过摄像头和传感器实时监测环境变化，识别入侵者或异常情况，并实时触发报警，而无需连接云端。

可穿戴设备：

- **健康监测**：智能手环或手表可以使用TinyML模型实时分析心率、血氧、睡眠等数据，提供个性化的健康建议，且无需依赖云端处理，减少功耗。
- **运动跟踪**：通过本地的TinyML模型，准确检测用户的运动模式（如跑步、骑行、瑜伽等），并给出健身建议。

### 环境监测：

- **智能农业：**使用部署在田间传感器中的TinyML模型，实时分析土壤湿度、气温、光照等数据，优化灌溉和施肥策略，提高作物产量。
- **空气质量监测：**家中或城市的空气质量传感器可以使用TinyML本地处理空气中污染物的实时监测，帮助提醒用户开窗通风或启动空气净化器。

### 家用电器智能化：

- **智能冰箱：**通过TinyML技术分析冰箱内物品种类和保质期，提醒用户何时该购买食材或避免食物浪费。
- **洗衣机：**通过嵌入式传感器和TinyML，洗衣机可以检测衣物材质和脏污程度，自动选择最佳的洗涤模式和时间。

### 智能交通：

- **车载设备：**使用TinyML模型分析车辆驾驶数据，提供实时驾驶行为分析，帮助减少油耗或提高行车安全。
- **交通信号优化：**在智能交通系统中，使用TinyML的摄像头和传感器可实现车辆流量分析，优化红绿灯切换时间，缓解交通拥堵。

### 个性化消费：

- **智能购物体验：**TinyML可以嵌入到智能购物车或镜子中，实时推荐商品或虚拟试穿衣服，提升顾客的购物体验。
- **自动售货机：**通过TinyML识别用户的消费习惯和购买历史，自动推荐产品，并进行库存管理。

### 无障碍辅助：

- **语音助手：**为听力障碍人士提供语音转文字服务，或者通过手势识别技术帮助有肢体障碍的人操作智能设备。
- **盲人导航设备：**嵌入TinyML模型的眼镜或导航设备，能在实时环境中识别障碍物和道路信息，帮助盲人安全出行。

## 5.2 模型数据集训练集和测试集的作用是什么？请详细介绍数据训练集大小不足会导致模型训练出现什么情况？

### 1. 训练集的作用

训练集是用于训练模型的数据集合，模型在训练过程中会从训练集的样本中学习其特征和规律。通过调整模型的权重和参数，模型尽量最小化损失函数，使其在训练集上的表现尽可能好。训练集数据的多样性和数量直接影响模型学习的能力。

### 2. 测试集的作用

测试集是用于评估模型泛化能力的数据集。训练完成后，测试集中的数据并不会被模型看到，测试集的作用是在不影响模型参数的前提下，检测模型在未见过的数据上的表现。通过测试集的结果，可以判断模型是否能够在现实世界的应用中表现良好，避免过拟合或欠拟合等问题。

### 训练集大小不足时可能出现的问题

当训练集的数据量不足时，模型训练可能会出现以下问题：

#### 1. 过拟合 (Overfitting)：

- 过拟合是指模型在训练集上表现非常好，但在测试集或真实数据上表现较差。当训练集数据量不足时，模型会学习到训练集中的所有细节和噪声，无法提取到更广泛的、通用的特征，导致模型在新数据上的泛化能力下降。
- 由于数据样本不足，模型更容易记住每个样本的具体模式，而不是学习这些样本背后的规律，导致模型对于看不见的数据失效。

## 2. 欠拟合 (Underfitting):

- 数据量不足时，模型可能无法捕捉到数据中的复杂模式和特征，表现为模型无法学习到数据中足够的信息。模型的表现不佳，不论是在训练集还是测试集上都无法获得好的性能。
- 尤其对于较复杂的模型（如深度神经网络），数据量不足时，模型没有足够的信息来学习复杂的参数，导致模型无法有效拟合训练数据。

## 3. 模型偏差:

- 数据样本不足可能导致训练集不能很好地代表数据的整体分布。模型会因为训练数据的偏差，学习到有偏见的模式或规律，尤其是当训练集中的某些特征或类别占比异常时。
- 这种偏差会使得模型在处理不同类别数据时表现不均衡，进而影响整体效果。例如，分类模型可能在某些类别上表现优异，而在其他类别上完全失败。

## 4. 模型不稳定性:

- 小数据集可能会导致模型训练结果的高度不稳定。由于数据量有限，每次训练时，模型可能会因为数据分布的微小差异而产生不同的结果，难以复现训练的效果。
- 这种不稳定性还会导致超参数选择上的困难，因为模型在不同的训练过程中表现差异过大，难以找到合适的超参数。

## 5. 泛化能力差:

- 数据不足时，模型学习到的特征和规律不够丰富，因此在面对复杂、未见过的数据时，模型往往难以做出准确的预测。
- 数据集的多样性不足也可能导致模型难以应对现实世界中的多种多样的场景，降低模型的适应性。

# 5.3 什么是欠拟合，什么是过拟合? 如何避免出现这两者情况,请各列举至少两点措施?

## a. 欠拟合 (Underfitting)

**欠拟合**是指模型没有充分学习到训练数据中的特征或模式，导致模型在训练集和测试集上都表现不佳。通常发生在模型太简单、无法捕捉数据中的复杂关系时。

### 1. 特征:

- 模型在训练集和测试集上的误差都很大。
- 模型无法很好地捕捉数据的趋势，简单来说，模型过于粗糙。

### 2. 产生原因:

- 模型太简单，不能处理数据的复杂性。例如，使用线性回归去拟合非线性数据。
- 特征选择不足或特征工程不充分，模型没有足够信息来学习数据的模式。
- 训练时间过短，模型没有充分学习。

### 3. 解决措施:

- **增加模型的复杂度:**

- 使用更复杂的模型，如从线性模型升级到非线性模型，或者从简单的线性回归转向神经网络或支持向量机（SVM）。
  - **增加特征数量：**
    - 通过特征工程生成更多有用的特征，提供更多信息给模型学习。可以使用多项式特征、特征交互等方法扩展数据的维度。
  - **增加训练时间：**
    - 欠拟合有时是由于模型没有充分训练所致。可以增加训练轮次（epochs）或延长训练时间，使模型有机会更好地拟合训练数据。
- 

## b. 过拟合（Overfitting）

**过拟合**是指模型在训练集上表现很好，但在测试集上表现较差。这是因为模型过度学习了训练数据中的细节和噪声，而没有学习到数据的通用模式，导致模型的泛化能力差。

### 1. 特征：

- 模型在训练集上的误差很小，但在测试集或新数据上的误差较大。
- 模型对训练数据记忆过多，包括噪声和无关模式，难以在未见过的数据上做出准确预测。

### 2. 产生原因：

- 模型太复杂，参数过多，如神经网络有太多层和节点。
- 训练数据不足，导致模型学习到过多的噪声和不相关特征。
- 没有使用正则化等方法限制模型复杂度。

### 3. 解决措施：

#### • 正则化技术：

- **L2正则化**（也叫岭回归）或 **L1正则化**（也叫Lasso回归）是常用的正则化方法，可以在训练过程中约束模型的参数，防止其变得过于复杂。
- **Dropout**：尤其在神经网络中，可以通过dropout技术，在训练时随机忽略部分神经元，从而减少过拟合。

#### • 增加训练数据量：

- 如果训练数据量不足，模型容易对训练集中的噪声过于敏感。增加训练数据可以让模型学习到更多有代表性的信息，减少过拟合的可能性。

#### • 早停法（Early Stopping）：

- 通过监控验证集的性能，当模型在验证集上的表现不再提升时停止训练，以防止模型继续过度拟合训练数据。

#### • 简化模型：

- 使用较简单的模型，减少模型的参数或复杂性（如减少神经网络层数或节点数），可以降低过拟合的风险。

## 5.4 tflite-micro库中MicroMutableOpResolver和AllOpsResolver两者的区别?为什么AllOpsResolver会被淘汰?

### 1. MicroMutableOpResolver

`MicroMutableOpResolver` 是一种轻量级且灵活的操作解析器，允许开发者根据实际需求手动选择并注册所需的操作。这使得模型可以只包含需要的操作，从而减少内存占用。因为它允许动态注册操作，所以特别适合资源受限的嵌入式设备。

**特点：**

- **灵活性：**开发者可以仅注册模型中使用到的特定操作，避免引入不必要的操作，这对于嵌入式设备上的内存和存储优化非常重要。
- **内存效率：**通过只加载需要的操作，`MicroMutableOpResolver` 显著减少内存使用，确保在资源受限的设备上能够更高效地运行模型。

### 2. AllOpsResolver

`AllOpsResolver` 是一种通用的操作解析器，它会自动注册TensorFlow Lite Micro框架支持的所有操作。虽然这种方式在开发中很方便，不需要手动选择和注册操作，但它的**缺点**也非常明显，尤其是在嵌入式设备等内存资源有限的场景下。

**特点：**

- **方便：**开发者不需要手动指定和注册操作，所有可用的操作都自动包含在内。
- **内存消耗大：**由于它加载了所有可能的操作，即便某些操作并没有被模型使用，也会占用内存。因此，这种方式对嵌入式设备非常不友好，容易导致资源浪费。

**使用场景：**

尽管 `AllOpsResolver` 适合快速原型开发或开发早期阶段，但由于其内存占用过大，在资源受限环境中不推荐使用。

### 3. 为什么 `AllOpsResolver` 被淘汰?

`AllOpsResolver` 在嵌入式和资源受限的场景中存在以下主要问题，这也是它被淘汰的原因：

#### 1. 内存占用过大：

- `AllOpsResolver` 会注册所有支持的操作，这意味着即便模型只需要少数几个操作，系统仍会加载不需要的操作，从而占用不必要的内存和存储空间。在资源有限的嵌入式系统中，内存和存储是极为宝贵的资源，因此这种做法非常低效。

#### 2. 不灵活：

- 它无法根据实际需求进行定制。对于不同的模型、任务和设备，开发者可能只需要特定的操作，`AllOpsResolver` 的“全包”特性使得它无法进行精细化控制，阻碍了模型在不同设备上的优化。

#### 3. 不适应嵌入式设备场景：

- 嵌入式设备通常具有非常有限的资源，如RAM、Flash存储等。`AllOpsResolver` 包含所有操作意味着它会增加不必要的负担，可能导致设备资源不足或者运行效率极低。

因此，`AllOpsResolver` 被淘汰的主要原因在于它不符合TFLite-Micro在嵌入式设备上运行的目标，无法实现内存和资源的高效管理。而 `MicroMutableOpResolver` 通过只加载所需的操作，确保内存使用最小化，满足了资源受限环境的需求。

## 6 实验数据记录和处理

### 1. 完善

12\_tingml\_gesture\_predict\_experiment/ledcontroller.cpp两个函数，这个直接拷贝你们实验一时候写好的代码。

完全来自 lab1 无删改。

2. 完善12\_tingml\_gesture\_predict\_experiment/main.cpp多处代码，实现多线程，一:定期神经网络识别设备运动状态并打印结果，二:led根据识别结果显示不同颜色，可同实验一。

### 1. 线程添加

```
get_imu_data(mpu, imu_data);  
ret = predict(imu_data, data_len, threshold, class_num);
```

添加 IMU 传感器周期性的数据并进行处理，处理后将信息传给 predict 函数进行预测。

### 2. 参数修改

在 lab1 的时候我使用的参数为 data.ax data.ay 等等，但 lab2 的 main 函数给的参数是 imu\_data[i\*6 + 0] 等，因此要进行适度的修改。

```
#include <stdio.h>  
#include <stdlib.h>  
#include <cmath>  
#include <string>  
#include <log.h>  
#include <errno.h>  
#include "periph_conf.h"  
#include "periph/gpio.h"  
#include "periph/i2c.h"  
#include "shell.h"  
#include <log.h>  
#include <xtimer.h>  
#include "ledcontroller.hh"  
#include "ztimer.h"  
#include "mpu6050.h"  
#include <string>  
#include "msg.h"  
  
void setup();  
int predict(float *imu_data, int data_len, float threshold, int class_num);  
using namespace std;  
#define THREAD_STACKSIZE (THREAD_STACKSIZE_IDLE)  
static char stack_for_motion_thread[THREAD_STACKSIZE];  
static char stack_for_led_thread[THREAD_STACKSIZE];  
static kernel_pid_t _led_pid;  
#define LED_GPIO_R GPIO26  
#define LED_GPIO_G GPIO25  
#define LED_GPIO_B GPIO27
```

```

#define g_acc (9.8)
#define SAMPLES_PER_GESTURE (10)
#define ACC_THRESHOLD 1.5 // 加速度静止阈值
#define ROT_THRESHOLD 50 // 角速度静止阈值
struct MPU6050Data
{
    float ax, ay, az; // acceler_x_axis, acceler_y_axis, acceler_z_axis
    float gx, gy, gz; // gyroscope_x_axis, gyroscope_y_axis, gyroscope_z_axis
};

enum MoveState{Stationary, Tilted, Rotating, Moving, MovX, MovY};

void delay_ms(uint32_t sleep_ms)
{
    ztimer_sleep(ZTIMER_USEC, sleep_ms * US_PER_MS);
    return;
}
/**
 * LED control thread function.
 * Then, it enters an infinite loop where it waits for messages to control the
LED.
 * @param arg Unused argument.
 * @return NULL.
 */
void *_led_thread(void *arg)
{
    (void) arg;
    LEDController led(LED_GPIO_R, LED_GPIO_G, LED_GPIO_B);
    led.change_led_color(0);
    while(1){
        // Input your codes
        // wait for a message to control the LED
        // Display different light colors based on the motion state of the
device.

        msg_t msg;
        msg_receive(&msg);

        if (msg.content.value == Stationary) {
            led.change_led_color(COLOR_NONE);
        } else if (msg.content.value == MovX) {
            led.change_led_color(COLOR_CYAN);
        } else if (msg.content.value == MovY) {
            led.change_led_color(COLOR_WHITE);
        } else if (msg.content.value == Tilted) {
            led.change_led_color(COLOR_RED);
        } else if (msg.content.value == Rotating) {
            led.change_led_color(COLOR_BLUE);
        } else if (msg.content.value == Moving) {
            led.change_led_color(COLOR_GREEN);
        } else {
            led.change_led_color(COLOR_NONE);
        }
        delay_ms(10);
    }
    return NULL;
}

```

```

}

float gyro_fs_convert = 1.0;
float accel_fs_convert;

void get_imu_data(MPU6050 mpu, float *imu_data){
    int16_t ax, ay, az, gx, gy, gz;
    for(int i = 0; i < SAMPLES_PER_GESTURE; ++i)
    {
        /* code */
        delay_ms(20);
        mpu.getMotion6(&ax, &ay, &az, &gx, &gy, &gz);
        imu_data[i*6 + 0] = ax / accel_fs_convert;
        imu_data[i*6 + 1] = ay / accel_fs_convert;
        imu_data[i*6 + 2] = az / accel_fs_convert;
        imu_data[i*6 + 3] = gx / gyro_fs_convert;
        imu_data[i*6 + 4] = gy / gyro_fs_convert;
        imu_data[i*6 + 5] = gz / gyro_fs_convert;
    }
}

void *_motion_thread(void *arg)
{
    (void) arg;
    // Initialize MPU6050 sensor
    MPU6050 mpu;
    // get mpu6050 device id
    uint8_t device_id = mpu.getDeviceID();
    printf("[IMU_THREAD] DEVICE_ID:0x%x\n", device_id);
    mpu.initialize();
    // Configure gyroscope and accelerometer full scale ranges
    uint8_t gyro_fs = mpu.getFullScaleGyroRange();
    uint8_t accel_fs_g = mpu.getFullScaleAccelRange();
    uint16_t accel_fs_real = 1;

    // Convert gyroscope full scale range to conversion factor
    if (gyro_fs == MPU6050_GYRO_FS_250)
        gyro_fs_convert = 131.0;
    else if (gyro_fs == MPU6050_GYRO_FS_500)
        gyro_fs_convert = 65.5;
    else if (gyro_fs == MPU6050_GYRO_FS_1000)
        gyro_fs_convert = 32.8;
    else if (gyro_fs == MPU6050_GYRO_FS_2000)
        gyro_fs_convert = 16.4;
    else
        printf("[IMU_THREAD] Unknown GYRO_FS: 0x%x\n", gyro_fs);

    // Convert accelerometer full scale range to real value
    if (accel_fs_g == MPU6050_ACCEL_FS_2)
        accel_fs_real = g_acc * 2;
    else if (accel_fs_g == MPU6050_ACCEL_FS_4)
        accel_fs_real = g_acc * 4;
    else if (accel_fs_g == MPU6050_ACCEL_FS_8)
        accel_fs_real = g_acc * 8;
    else if (accel_fs_g == MPU6050_ACCEL_FS_16)
        accel_fs_real = g_acc * 16;
}

```



```

else
    printf("[IMU_THREAD] Unknown ACCEL_FS: 0x%x\n", accel_fs_g);

// Calculate accelerometer conversion factor
accel_fs_convert = 32768.0 / accel_fs_real;
float imu_data[SAMPLES_PER_GESTURE * 6] = {0};
int data_len = SAMPLES_PER_GESTURE * 6;
delay_ms(200);
// Main loop
int predict_interval_ms = 200;
int ret = 0;
#define class_num (4)
float threshold = 0.7;
string motions[class_num] = {"Stationary", "Tilted", "Rotating", "Moving"};
while (1) {
    delay_ms(predict_interval_ms);
    // Read sensor data
    get_imu_data(mpu, imu_data);
    ret = predict(imu_data, data_len, threshold, class_num);
    // tell the led thread to do some operations
    // input your code
    // Print result

    // 向LED线程发送消息通知当前的运动状态
    msg_t msg;
    msg.content.value = ret;
    msg_send(&msg, _led_pid); // 发送消息到LED控制线程

    printf("Predict: %d, %s\n", ret, motions[ret].c_str());
}
return NULL;
}

int main(int argc, char* argv[])
{
    (void)argc;
    (void)argv;
    setup();
    _led_pid = thread_create(stack_for_led_thread, sizeof(stack_for_led_thread),
        THREAD_PRIORITY_MAIN - 2,
        THREAD_CREATE_STACKTEST, _led_thread, NULL,
        "led_controller_thread");
    if (_led_pid <= KERNEL_PID_UNDEF) {
        printf("[MAIN] Creation of receiver thread failed\n");
        return 1;
    }
    thread_create(stack_for_motion_thread, sizeof(stack_for_motion_thread),
        THREAD_PRIORITY_MAIN - 1,
        THREAD_CREATE_STACKTEST, _motion_thread, NULL,
        "imu_read_thread");
    printf("[Main] Initialization successful - starting the shell now\n");
    while(1);
    return 0;
}

```

3. 案例给出了MLP模型和CNN模型,请从模型训练,模型测试以及真实部署三个方面进行识别准确度性能对比.

模型结构图:

cnn:

```
Model: "sequential"
```

Layer (type)	Output Shape	Param #
conv1d (Conv1D)	(None, 60, 8)	32
conv1d_1 (Conv1D)	(None, 60, 8)	200
global_average_pooling1d (Gl	(None, 8)	0
dense (Dense)	(None, 8)	72
dropout (Dropout)	(None, 8)	0
dense_1 (Dense)	(None, 4)	36

Total params: 340  
Trainable params: 340  
Non-trainable params: 0

mlp:

```
Model: "sequential"
```

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 64)	3904
dropout (Dropout)	(None, 64)	0
dense_1 (Dense)	(None, 4)	260

Total params: 4,164  
Trainable params: 4,164  
Non-trainable params: 0

# 1. 模型训练阶段

## (1) 训练时间

在 `train.py` 中，训练 MLP 和 CNN 模型分别使用以下代码：

```
model = mlp() # 或者使用 model = cnn() 进行对比
history = model.fit(
    xTrain,
    yTrain,
    batch_size=8,
    validation_data=(xTest, yTest),
    epochs=50,
    verbose=1,
    callbacks=[checkpoint],
)
```

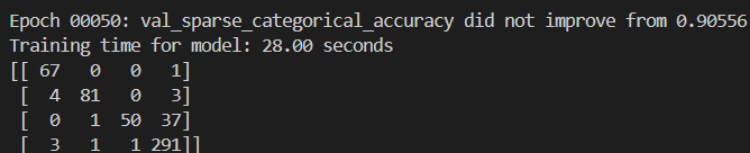
- 比较方法

记录两个模型的训练时间。可以在 `fit` 前后加入 `time`

模块的计时代码：

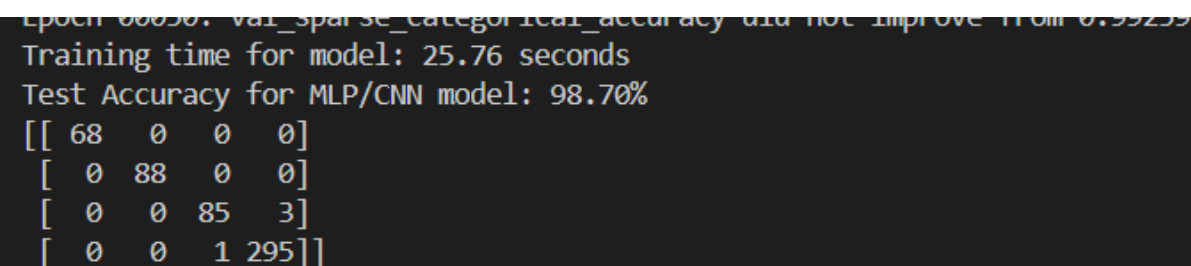
```
import time
start_time = time.time()
history = model.fit(...)
end_time = time.time()
print("Training time for model: {:.2f} seconds".format(end_time - start_time))
```

下图为使用 cnn 模型的测试结果：



```
Epoch 00050: val_sparse_categorical_accuracy did not improve from 0.90556
Training time for model: 28.00 seconds
[[ 67  0  0  1]
 [  4 81  0  3]
 [  0  1 50 37]
 [  3  1  1 291]]
```

下图为使用 mlp 模型的测试结果：



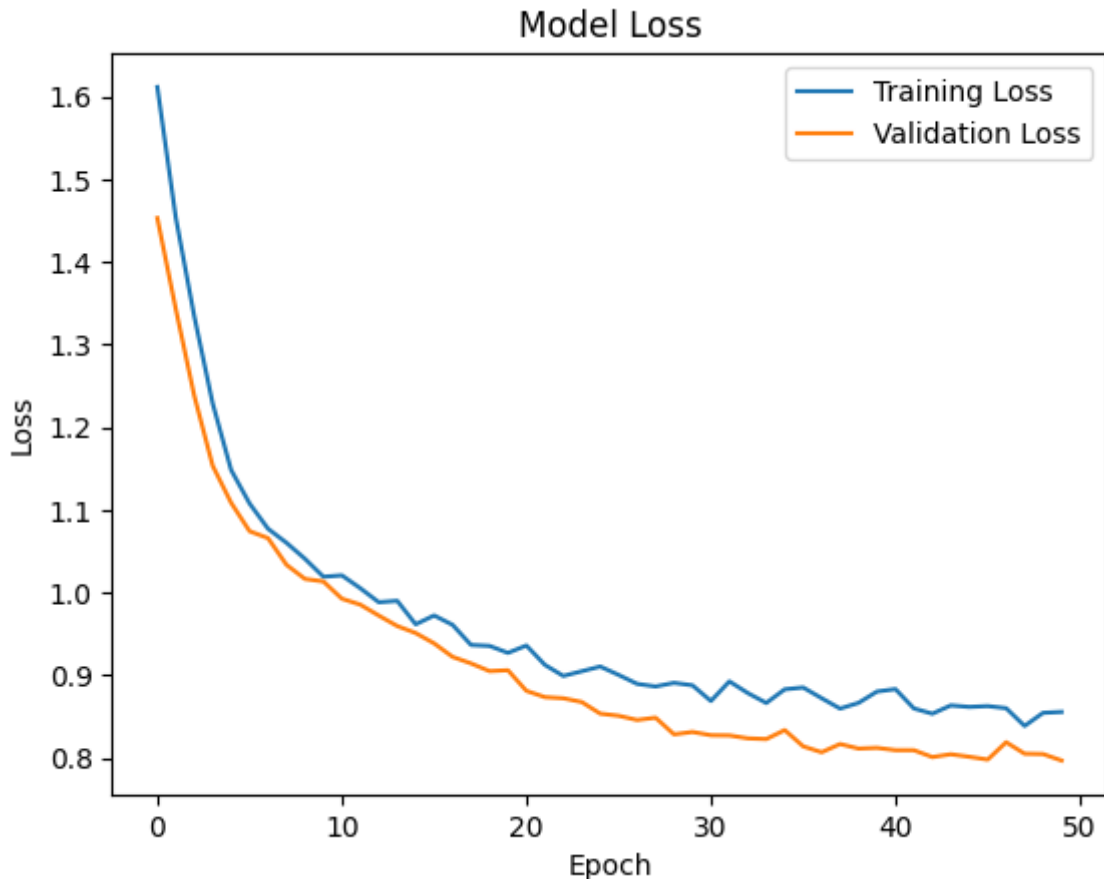
```
Epoch 00050: val_sparse_categorical_accuracy did not improve from 0.99259
Training time for model: 25.76 seconds
Test Accuracy for MLP/CNN model: 98.70%
[[ 68  0  0  0]
 [  0 88  0  0]
 [  0  0 85  3]
 [  0  0  1 295]]
```

## (2) 收敛速度

- 使用 `history` 对象中的损失值和准确率绘制收敛曲线，观察两个模型的收敛速度和最终的训练准确度。

```
import matplotlib.pyplot as plt
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Model Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.show()
```

a. 下图为使用 cnn 模型的收敛速度：



**模型收敛性：**

- 在训练开始时，损失值较高，随着训练进行，损失值逐渐下降，表明模型在逐渐学习训练数据并收敛。
- 训练损失和验证损失在50个epoch内持续下降，且没有出现明显的上升趋势，表明模型尚未出现过拟合现象，训练效果较好。

**训练与验证的损失差异：**

- 整个过程中，训练损失和验证损失基本保持接近，说明模型的泛化能力较好，能够很好地适应测试数据而不是仅仅记住训练数据。

**损失值下降速度：**

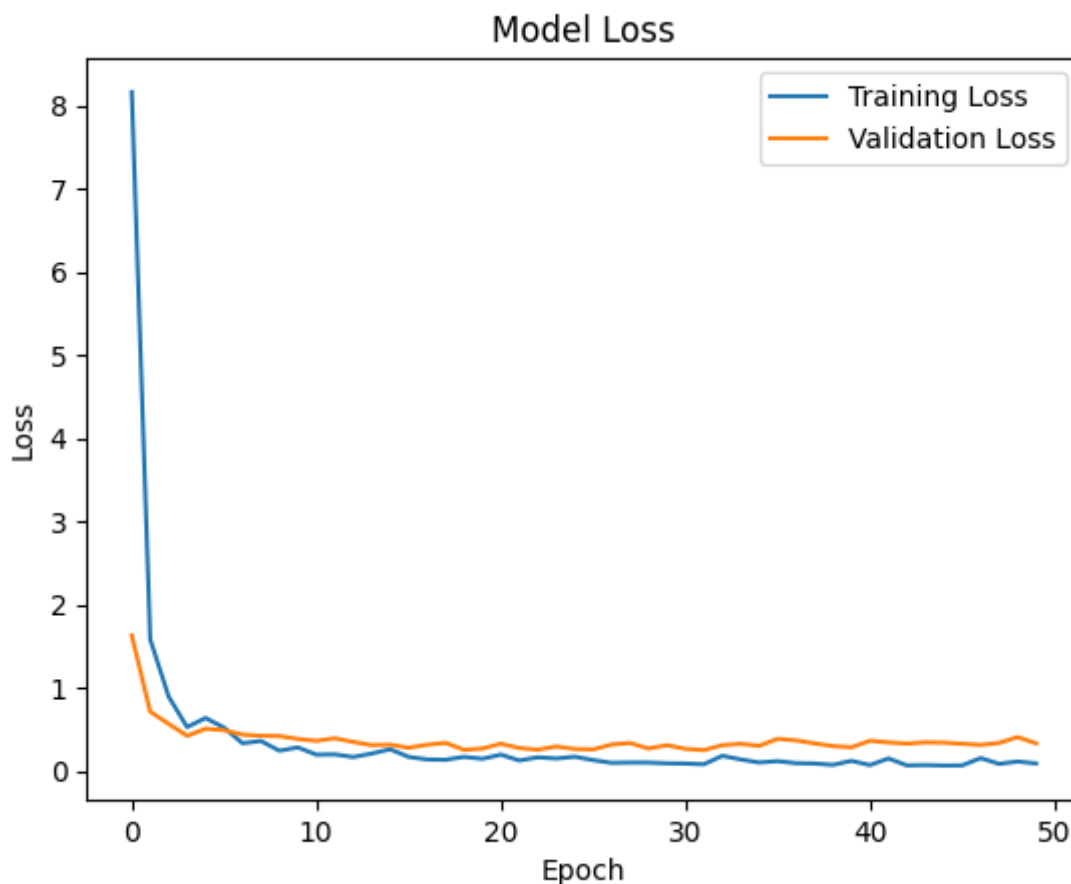
- 前 10 个 epoch，损失值迅速下降，说明模型在最初的阶段对数据有较大的改进。
- 之后的损失下降速度趋于平缓，表明模型在逐渐收敛。

**最低损失值：**

- 最后阶段，验证损失和训练损失都趋于稳定，表明模型已经达到最佳状态，继续增加 epoch 可能不会带来显著的性能提升。

总体而言，当前的 CNN 模型在训练与验证集上的表现比较平衡，训练过程是成功的。

b. 下图为使用 mlp 模型的收敛速度：



**初期收敛速度：**

- 在前几个 epoch 内，训练损失从一个较高的值迅速下降到接近 1 左右，说明模型在初期对训练数据进行了快速的学习和调整。
- 验证损失也在初期阶段迅速下降，说明模型对验证集数据也取得了较好的泛化效果。

**后期损失稳定：**

- 在第 10 个 epoch 之后，训练损失和验证损失都趋于稳定，基本保持在一个较低的范围内，这表明模型已经基本收敛，继续训练并没有显著提高模型性能的效果。

**验证集和训练集损失的差距：**

- 在大部分训练过程中，训练损失略低于验证损失，且两者曲线非常接近，这说明模型的泛化能力较好，并没有出现明显的过拟合问题。

**相对平稳：**

- 相比于 CNN 模型，MLP 模型的损失曲线在中后期非常平稳，表明模型参数调整已经比较充分，达到了一个平衡状态。

**总体比较：**

- 与 CNN 模型相比，MLP 模型在收敛速度上非常快，并且同样具备良好的泛化能力。
- MLP 可能更适合相对简单或低维度的数据，但在处理复杂数据时，CNN 往往具有更强的特征提取能力。

这说明当前的 MLP 模型在这次任务中表现良好，训练效果相当理想。

## 2. 模型测试阶段

### (1) 测试准确度

- 训练完成后，通过 `model.evaluate()` 方法在测试集上评估两个模型的准确度：

```
test_loss, test_accuracy = model.evaluate(xTest, yTest, verbose=0)
print("Test Accuracy for MLP/CNN model: {:.2f}%".format(test_accuracy * 100))
```

下图为使用 cnn 模型的准确度：

```
61/270 [====>.....] - ETA: 0s - loss: 0.2338 - sparse_categorical_accuracy: 0.9500
93/270 [=====>.....] - ETA: 0s - loss: 0.2181 - sparse_categorical_accuracy: 0.9500
119/270 [=====>.....] - ETA: 0s - loss: 0.2180 - sparse_categorical_accuracy: 0.9500
152/270 [=====>.....] - ETA: 0s - loss: 0.2297 - sparse_categorical_accuracy: 0.9500
186/270 [=====>.....] - ETA: 0s - loss: 0.2160 - sparse_categorical_accuracy: 0.9500
217/270 [=====>.....] - ETA: 0s - loss: 0.2255 - sparse_categorical_accuracy: 0.9500
249/270 [=====>.....] - ETA: 0s - loss: 0.2382 - sparse_categorical_accuracy: 0.9500
270/270 [=====] - 1s 2ms/step - loss: 0.2348 - sparse_categorical_accuracy: 0.9199 - val_loss: 0.1645 - val_sparse_categorical_accuracy: 0.9500

Epoch 00050: val_sparse_categorical_accuracy did not improve from 0.96296
Training time for model: 27.81 seconds
Test Accuracy for MLP/CNN model: 95.00%
```

下图为使用 mlp 模型的准确度：

```
Epoch 00050: val_sparse_categorical_accuracy did not improve from 0.95259
Training time for model: 25.76 seconds
Test Accuracy for MLP/CNN model: 98.70%
[[ 68  0  0  0]
 [  0 88  0  0]
 [  0  0 85  3]
 [  0  0  1 295]]
```

### (2) 查看混淆矩阵

- 使用 `confusion_matrix` 来比较两个模型的预测效果：

```
from sklearn.metrics import confusion_matrix
predictions = model.predict(xTest)
predictions = np.argmax(predictions, axis=1)
cm = confusion_matrix(yTest, predictions)
print(cm)
```

绘制混淆矩阵，查看模型对各类别的识别准确度，观察两个模型对各个动作状态的识别效果。

下图为使用 cnn 模型的混淆矩阵：

```
Training time for model: 27.81 seconds
Test Accuracy for MLP/CNN model: 95.00%
[[ 68  0  0  0]
 [  0 85  0  3]
 [  0  2 75 11]
 [  2  0  2 292]]
WARNING:absl:Optimization option OPTIMIZE_FOR_SIZE is deprecated, please use optimize_for_gpu instead
```

下图为使用 mlp 模型的混淆矩阵：

```
Epoch 66656: val_sparse_categorical_accuracy did not improve from 0.99259
Training time for model: 25.76 seconds
Test Accuracy for MLP/CNN model: 98.70%
[[ 68  0  0  0]
 [  0 88  0  0]
 [  0  0 85  3]
 [  0  0  1 295]]
```

### 3. 真实部署阶段

#### (1) 推理速度

- 测试模型推理时间:

```
start_time = time.time()
for i in range(20):
    test_data = np.expand_dims(xTest[i], axis=0).astype("float32")
    model.predict(test_data)
end_time = time.time()
print("Average Inference Time: {:.4f} seconds".format((end_time - start_time)
/ 20))
```

可知使用 cnn 模型的推理时间为 0.004s, mlp 模型的推理时间为 0.002s。

#### (2) 模型大小

- 查看 `.tflite` 文件大小, 以比较模型在部署时的存储空间需求:

```
basic_model_size = os.path.getsize("model_basic.tflite")
quantized_model_size = os.path.getsize("model.tflite")
print("MLP/CNN Basic Model Size: {} bytes".format(basic_model_size))
print("Quantized Model Size: {} bytes".format(quantized_model_size))
```

如图为 cnn 模型获得的模型大小:

```
Average Inference Time: 0.0004 seconds
MLP/CNN Basic Model Size: 5256 bytes
Quantized Model Size: 5592 bytes
```

如图为 mlp 模型获得的模型大小:

```
Digit: 3.0 - Prediction:
[[0.33203125 0.          0.33203125 0.33203125]]

Average Inference Time: 0.0004 seconds
MLP/CNN Basic Model Size: 18036 bytes
Quantized Model Size: 6328 bytes
```

## 4. 自主设计一个运动状态识别模型(其他模型结构或者适当增大模型参数),要去比案例的CNN模型性能好.

设计 improved\_cnn 模型, 见文件 `train_impcnn.py`, 生成模型 `model_basicimpcnn.tflite` 和 `modelimpcnn.tflite`

为了设计一个比案例中的 CNN 模型性能更好的运动状态识别模型, 可以考虑以下几个方面:

1. **增加模型的深度和宽度**: 增加卷积层的数量和每一层的神经元数量, 以便模型可以学习到更复杂的特征。
2. **使用更高级的卷积层结构**: 引入一些高级卷积层结构, 如残差块 (ResNet) 或深度可分离卷积 (Depthwise Separable Convolutions), 以提高模型的特征提取能力。
3. **增加正则化和数据增强**: 通过增加 `Dropout` 或 `Batch Normalization` 等正则化手段, 防止模型过拟合; 同时通过数据增强, 丰富训练数据, 提高模型泛化能力。
4. **使用更好的优化器和学习率策略**: 尝试使用学习率调度策略或其他优化器 (如 AdamW、RMSprop) 来加速训练和提高模型性能。

### a. 自主设计的改进 CNN 模型

以下是一个基于改进后的 CNN 模型的设计, 它包含更多的卷积层, 加入了 `Batch Normalization`、`Dropout`, 并引入了 `Residual Blocks` 以增强特征提取能力:

### b. 改进后的模型架构

```
import tensorflow as tf
import tensorflow.keras as keras

def improved_cnn():
    model = keras.Sequential()

    # 第一卷积层
    model.add(keras.layers.Conv1D(16, 3, padding="same", activation="relu",
input_shape=(6 * SAMPLES_PER_GESTURE, 1)))
    model.add(keras.layers.BatchNormalization())
    model.add(keras.layers.Conv1D(16, 3, padding="same", activation="relu"))
    model.add(keras.layers.MaxPooling1D(pool_size=2))
    model.add(keras.layers.Dropout(0.3)) # 增加 dropout 防止过拟合

    # 第二卷积层
    model.add(keras.layers.Conv1D(32, 3, padding="same", activation="relu"))
    model.add(keras.layers.BatchNormalization())
    model.add(keras.layers.Conv1D(32, 3, padding="same", activation="relu"))
    model.add(keras.layers.MaxPooling1D(pool_size=2))
    model.add(keras.layers.Dropout(0.4)) # 增加 dropout 防止过拟合

    # 第三卷积层
    model.add(keras.layers.Conv1D(64, 3, padding="same", activation="relu"))
    model.add(keras.layers.BatchNormalization())
    model.add(keras.layers.Conv1D(64, 3, padding="same", activation="relu"))
    model.add(keras.layers.GlobalAveragePooling1D()) # 全局平均池化
    model.add(keras.layers.Dropout(0.5))

    # 全连接层
```



```

model.add(keras.layers.Dense(128, activation="relu"))
model.add(keras.layers.BatchNormalization())
model.add(keras.layers.Dropout(0.5))

# 输出层
model.add(keras.layers.Dense(len(LABELS), activation="softmax"))

return model

```

### c. 优化与改进

- 引入 `BatchNormalization`：可以加快训练速度，并提高模型的泛化能力。
- 使用 `Dropout`：提高防止过拟合的效果。
- 增加卷积层和神经元数量：更强的特征提取能力，有助于提高模型的准确性。

下图为使用 `improved_cnn` 模型的测试结果、准确度和混淆矩阵：

```

Test Accuracy for improved_cnn model: 93.43%
[[ 96  0  0  0]
 [  0 147  1  0]
 [  0  18  64  6]
 [  0  10  1 205]]
INFO:tensorflow:Assets written to: /tmp/tmpinzxsuh7/assets
WARNING:absl:Optimization option OPTIMIZE_FOR_SIZE is deprecated, please use optimizations=[Optimize.DEFAULT] instead.
INFO:tensorflow:Assets written to: /tmp/tmp_zppvwx1/assets
INFO:tensorflow:Assets written to: /tmp/tmp_zppvwx1/assets
WARNING:absl:Optimization option OPTIMIZE_FOR_SIZE is deprecated, please use optimizations=[Optimize.DEFAULT] instead.
WARNING:absl:Optimization option OPTIMIZE_FOR_SIZE is deprecated, please use optimizations=[Optimize.DEFAULT] instead.
Basic model is 146192 bytes
Quantized model is 55320 bytes
Difference is 90872 bytes

```

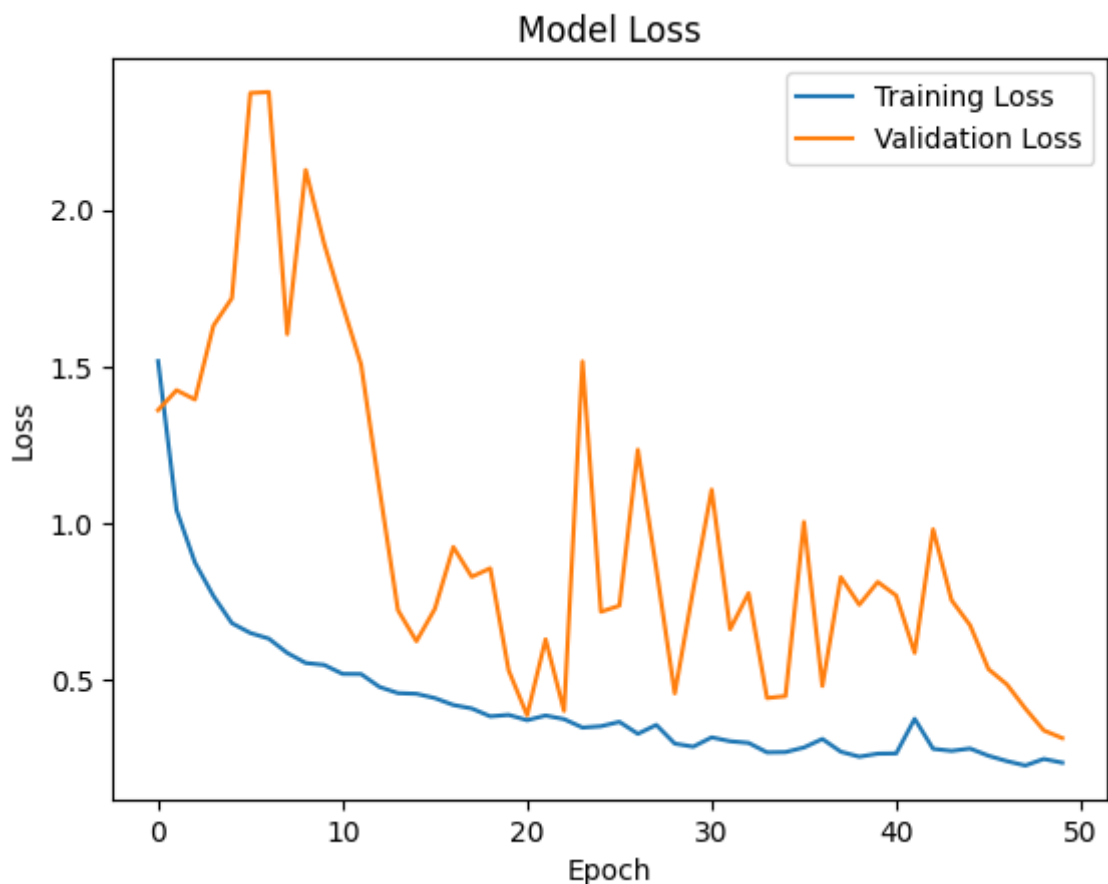
下图为使用 `improved_cnn` 模型的模型大小推理速度：

```

Epoch 4/50
274/274 [=====] - 2s 8ms/step - loss: 0.7670 - sparse_categorical_accuracy: 0.6966 - val_loss: 1.6302 - val_sparse_categorical_accuracy:
Epoch 00004: val_sparse_categorical_accuracy did not improve from 0.49818
Epoch 5/50
274/274 [=====] - 2s 8ms/step - loss: 0.6804 - sparse_categorical_accuracy: 0.7322 - val_loss: 1.7183 - val_sparse_categorical_accuracy:
Epoch 00005: val_sparse_categorical_accuracy did not improve from 0.49818
Epoch 6/50
274/274 [=====] - 2s 8ms/step - loss: 0.6489 - sparse_categorical_accuracy: 0.7523 - val_loss: 2.3734 - val_sparse_categorical_accuracy:
Epoch 00006: val_sparse_categorical_accuracy did not improve from 0.49818
Epoch 7/50
274/274 [=====] - 2s 8ms/step - loss: 0.6310 - sparse_categorical_accuracy: 0.7619 - val_loss: 2.3764 - val_sparse_categorical_accuracy:
...
274/274 [=====] - 2s 7ms/step - loss: 0.2347 - sparse_categorical_accuracy: 0.9206 - val_loss: 0.3131 - val_sparse_categorical_accuracy:
Epoch 00050: val_sparse_categorical_accuracy improved from 0.91423 to 0.93431, saving model to best_model.h5
Training time for model: 143.11 seconds

```

下图为使用 `improved_cnn` 模型的收敛速度：



#### 1. 训练损失收敛情况：

- 蓝色曲线（训练损失）整体呈下降趋势，说明模型在训练集上的学习效果较好，逐渐减少了损失值。
- 在接近 50 个 epoch 时，训练损失趋于稳定，表明模型基本达到了收敛状态。

#### 2. 验证损失波动较大：

- 橙色曲线（验证损失）波动非常大，甚至出现多次明显的峰值。这表明模型在验证集上的表现不稳定，可能存在过拟合问题。

#### 3. 过拟合迹象：

- 训练损失持续下降并趋于稳定，而验证损失一直在较大的范围内波动，尤其在20到50个epoch阶段。训练损失和验证损失之间的差距较大，表明模型可能对训练数据“记忆”过度，对验证数据的泛化能力较差。

#### 4. 潜在原因：

- **模型复杂度过高**：模型参数可能过多，导致对训练数据拟合过度。可以考虑减小模型的复杂度（例如减少卷积层的数量、减小每一层的神经元数量等）。
- **正则化不足**：虽然已经使用了 `Dropout`，但可能效果不够，可以尝试增加 `Dropout` 比例（如从 0.3 提高到 0.5），或者添加 `L2` 正则化。
- **数据不足或不平衡**：如果训练数据不够丰富或类别不均衡，可能会导致模型对验证数据表现较差。可以考虑增加数据量或使用数据增强技术来改善模型的泛化能力。

当前的模型在训练集上表现良好，但在验证集上表现不稳定，存在过拟合的可能。通过调整模型结构、正则化方法、数据增强等策略，可以进一步提高模型的泛化性能，减少验证损失的波动。

# 7 基于该实验，总结开发一个可部署的AI模型应用基本流程

## 1. 问题定义与需求分析

- **明确目标**：首先，需要清楚地定义AI模型的应用场景和目标。了解需要解决的问题，并明确最终部署环境的限制（如硬件平台、资源受限设备等）。
- **需求分析**：包括性能要求（速度、准确率等）、内存和存储限制、响应时间等。在嵌入式设备上，低功耗和实时性可能是关键。

## 2. 数据收集与准备

- **数据收集**：根据应用场景，收集相关的数据集，确保数据质量和数量足够。数据类型可以是文本、图像、视频、传感器数据等。
- **数据清洗与预处理**：清洗数据（去除异常值、空值等），并进行标准化或归一化等预处理操作，确保数据适合输入到模型中。
- **数据集划分**：将数据集划分为训练集、验证集和测试集，确保模型的性能能够在未见过的数据上进行有效验证。

## 3. 模型选择与设计

- **选择合适的模型架构**：根据问题和数据特点选择合适的机器学习或深度学习模型。例如，图像分类问题可以选择卷积神经网络（CNN），自然语言处理问题可以选择循环神经网络（RNN）或Transformer架构。
- **初步设计模型**：根据需求自定义模型结构，或者使用预训练模型并进行微调（Transfer Learning）来加快开发速度，尤其是在数据有限的情况下。

## 4. 模型训练与验证

- **训练模型**：使用训练集训练模型，调整模型的参数，使其能够有效地学习数据的模式。通过损失函数的最小化来指导模型优化。
- **模型验证**：使用验证集调整模型的超参数，如学习率、正则化系数等，防止过拟合。在验证集上监控模型性能，确保模型能够泛化。
- **防止过拟合**：可以使用正则化、Dropout、早停（Early Stopping）等技术防止模型在训练数据上过拟合。

## 5. 模型优化

- **模型压缩**：在嵌入式设备或资源受限环境下，可能需要对模型进行优化。例如，使用量化（Quantization）、剪枝（Pruning）等技术来减少模型的存储占用和计算量。
- **转换为可部署格式**：使用像TensorFlow Lite或ONNX等工具将模型转换为适合部署的格式，如TensorFlow Lite模型（.tflite）或ONNX格式，特别是在嵌入式设备上。

## 6. 部署准备

- **推理引擎选择**：根据部署环境选择合适的推理引擎（如TensorFlow Lite、ONNX Runtime、TFLite Micro等）。如果是嵌入式设备，TFLite Micro可以在内存受限的情况下高效运行。
- **操作注册**：根据模型使用的操作，使用如 `MicroMutableOpResolver` 等工具注册仅模型中实际使用的操作，减少内存占用。

## 7. 模型部署

- **将模型部署到目标设备**：将转换好的模型部署到目标设备上。例如，嵌入式设备、智能手机、边缘设备或云服务器。
- **测试与调优**：在实际部署环境中对模型进行测试，确保模型的性能和速度符合需求。如果在嵌入式设备上，确保模型在设备上的内存占用和推理时间能够满足实际应用场景的要求。

## 8. 监控与维护

- **实时监控**：部署后，对模型在真实环境中的表现进行实时监控，捕捉模型性能、准确率的变化，尤其是边缘设备的功耗和响应时间。
- **持续更新**：随着数据的增加或环境的变化，定期对模型进行更新和再训练，以保持其性能和适应性。

## 总结

开发一个可部署的AI模型应用的流程可以概括为以下几个步骤：

1. **明确问题和需求**：定义应用场景、硬件限制和性能要求。
2. **数据收集与准备**：获取高质量的数据集，并对数据进行处理。
3. **模型选择与设计**：选择合适的模型并设计其架构。
4. **训练与验证**：训练模型并验证其在未见过数据上的泛化能力。
5. **模型优化**：通过模型压缩和转换技术，使模型适应目标硬件平台。
6. **部署准备**：选择合适的推理引擎并注册必要的操作。
7. **模型部署**：将优化后的模型部署到目标设备上，并进行测试。
8. **监控与维护**：在实际环境中监控模型性能，并定期更新。