# XOR 实验于 MCU 的部署和训练

# 目录

一、摘要..... 3
二、XOR 神经网络设计 ..... 4
2.1.神经网络 layout 设计..... 4
2.2.参数的导数计算..... 5
三、基于 C 语言的类设计..... 8
四、STM32 的部署 ..... 11
参考文献..... 15

## 一、摘要

**内容摘要：**本项目基于 STM32 裸机进行神经网络的简易部署，鉴于 STM32 单片机 RAM 容量极小，项目采用外部存储的方式（采用 EEPROM 如 AT24C256 存储模型，鉴于 XOR 实验参数较少，每次可直接将模型装载进 RAM 直接参与运算），同时采用 C 语言进行裸机编程控制，同时开启定时器定时检测，控制输出 LED 的亮灭。

**关键词：**人工智能，神经网络，单片机

**Abstract:** This project is based on STM32 for simple deployment of neural network. In view of the STM32’s RAM capacity is too small, the project adopts the external storage method (use EEPROM such as AT24C256 to storage the trained model. The number of XOR experimental parameters is small so each time the model can be directly loaded into RAM to directly participate in the train or predict). At the same time, we decide to use the C language to control STM32 without any RTOS. The timer is turned on to control the LED’s output.

**Key words:** artificial intelligence, neural networks, microcontrollers

## 二、XOR 神经网络设计

### 2.1. 神经网络 layout 设计

经典 XOR 三层网络由以下 layout 构成：

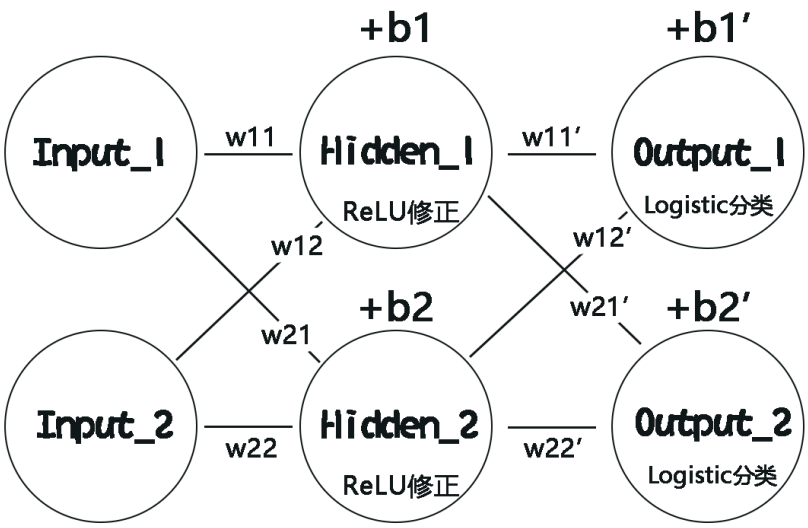


图 1：网络结构设计图

其中，前向传播公式为：

$$H = ReLU^T(W^T * X + B)$$

$$O = Logistic(W'^T * H + B')$$

其中隐藏层使用 ReLU 线性修正单元添加非线性因素，输出层使用 Logistic 函数用于线性二分类，其函数原型如图 2 所示。鉴于单片机较小的存储空间，不打算制作数值微分和符号微分和自动微分，我们将直接使用推导得出的导数运算式。从而大幅的降低部署难度，但是鉴于较低的可移植性，所以受限性很大，即仅适用于本 XOR 实验。首先，我们定义损失函数，鉴于本实验为离散二分类问题，所以我们采取交叉熵损失函数。

$$L = CrossEntropy(y,p) = -\frac{1}{2} * \sum_{i=1}^{2} y_i * \ln p_i$$

```
6  double ReLU(double dMem)
7  {
8      if (dMem > 0)
9          return dMem;
10     else
11         return 0;
12 }
13
```

图 2： ReLU 函数

```c
void Logistic(double **pMem, unsigned ulCount)
{
    double *pMemory = (double *)malloc(ulCount * sizeof(double));
    double *pTmp = *pMem, *pTmp2 = pMemory;
    double total = 0;

    unsigned i = 0;
    for (; i < ulCount; ++i)
        total += exp(*pTmp++);

    pTmp = *pMem;
    for (i = 0; i < ulCount; ++i)
        *pTmp2++ = exp(*(pTmp + i)) / total;

    free(*pMem);
    *pMem = pMemory;
}
```

图 3：Logistic 函数

```c
double CrossEntropy(double *pMem, double *pLabel, unsigned ulCount)
{
    double ret = 0;

    unsigned i = 0;
    for (; i < ulCount; ++i)
        ret += (*(pLabel + i)) * log(*(pMem + i));

    return -ret / ulCount;
}
```

图 4：CrossEntropy 函数

```
PS D:\pandownload1\Desktop\CodingFolder\C\XOR> gcc .\main.c -o main.exe
PS D:\pandownload1\Desktop\CodingFolder\C\XOR> .\main.exe
x: 2.000000, y: -2.000000
relu_x: 2.000000, relu_y: 0.000000

x: 2.000000, y: 4.000000
logistic_x: 0.119203, logistic_y: 0.880797

label_x: 0.000000, label_y: 1.000000
loss: 0.063464
PS D:\pandownload1\Desktop\CodingFolder\C\XOR>
```

图 5：测试结果验证

## 2.2. 参数的导数计算

然后就是喜闻乐见的求导环节：

$$\frac{\partial L}{\partial o_1} = -\frac{y_1}{2 * o_1}, \quad \frac{\partial L}{\partial o_2} = -\frac{y_2}{2 * o_2}$$

$$\frac{\partial o_1}{\partial a_1'} = o_1 * (1 - o_1), \quad \frac{\partial o_2}{\partial a_2'} = o_2 * (1 - o_2)$$

$$\frac{\partial a_1'}{\partial w_{11}'} = h_1, \quad \frac{\partial a_1'}{\partial w_{12}'} = h_2, \quad \frac{\partial a_2'}{\partial w_{21}'} = h_1, \quad \frac{\partial a_2'}{\partial w_{22}'} = h_2$$

$$\frac{\partial a_1'}{\partial b_1'} = 1, \quad \frac{\partial a_2'}{\partial b_2'} = 1$$

$$\frac{\partial L}{\partial w_{11}'} = h_1 * o_1 * (1 - o_1) * (-\frac{y_1}{2 * o_1})$$

$$\frac{\partial L}{\partial w_{12}'} = h2 * o_1 * (1 - o_1) * (-\frac{y_1}{2 * o_1})$$

$$\frac{\partial L}{\partial w_{21}'} = h1 * o_2 * (1 - o_2) * (-\frac{y_2}{2 * o_2})$$

$$\frac{\partial L}{\partial w_{22}'} = h2 * o_2 * (1 - o_2) * (-\frac{y_2}{2 * o_2})$$

$$\frac{\partial L}{\partial b_1'} = 1 * o_1 * (1 - o_1) * (-\frac{y_1}{2 * o_1})$$

$$\frac{\partial L}{\partial b_2'} = 1 * o_2 * (1 - o_2) * (-\frac{y_2}{2 * o_2})$$

展开公式 $\frac{\partial L}{\partial h_i} = \sum_{j=1}^{2} \frac{\partial L}{\partial a_j'} * \frac{\partial a_j'}{\partial h_i}$

$$\frac{\partial L}{\partial h_1} = w_{11}' * o_1 * (1 - o_1) * \left(-\frac{y_1}{2 * o_1}\right) + w_{21}' * o_2 * (1 - o_2) * (-\frac{y_2}{2 * o_2})$$

$$\frac{\partial L}{\partial h_2} = w_{12}' * o_1 * (1 - o_1) * \left(-\frac{y_1}{2 * o_1}\right) + w_{22}' * o_2 * (1 - o_2) * (-\frac{y_2}{2 * o_2})$$

$$\frac{\partial h_1}{\partial a_1} = \begin{cases} 0, & if\ h_1 \le 0 \\ 1, & if\ h_1 > 0 \end{cases}, \quad \frac{\partial h_2}{\partial a_2} = \begin{cases} 0, & if\ h_2 \le 0 \\ 1, & if\ h_2 > 0 \end{cases}$$

$$\frac{\partial a_1}{\partial w_{11}} = x_1, \quad \frac{\partial a_1}{\partial w_{12}} = x_2, \quad \frac{\partial a_2}{\partial w_{21}} = x_1, \quad \frac{\partial a_2}{\partial w_{22}} = x_2$$

$$\frac{\partial a_1}{\partial b_1} = 1, \quad \frac{\partial a_2}{\partial b_2} = 1$$

$$\frac{\partial L}{\partial w_{11}} = \frac{\partial a_1}{\partial w_{11}} * \frac{\partial h_1}{\partial a_1} * \frac{\partial L}{\partial h_1} = \begin{cases} 0, & if\ h_1 \le 0 \\ x_1 * 1 * \frac{\partial L}{\partial h_1}, & if\ h_1 > 0 \end{cases}$$

$$\frac{\partial L}{\partial w_{12}} = \frac{\partial a_1}{\partial w_{12}} * \frac{\partial h_1}{\partial a_1} * \frac{\partial L}{\partial h_1} = \begin{cases} 0, & if\ h_1 \le 0 \\ x_2 * 1 * \frac{\partial L}{\partial h_1}, & if\ h_1 > 0 \end{cases}$$

$$\frac{\partial L}{\partial w_{21}} = \frac{\partial a_2}{\partial w_{21}} * \frac{\partial h_2}{\partial a_2} * \frac{\partial L}{\partial h_2} = \begin{cases} 0, & if\ h_2 \le 0 \\ x_1 * 1 * \frac{\partial L}{\partial h_2}, & if\ h_2 > 0 \end{cases}$$

$$\frac{\partial L}{\partial w_{22}} = \frac{\partial a_2}{\partial w_{22}} * \frac{\partial h_2}{\partial a_2} * \frac{\partial L}{\partial h_2} = \begin{cases} 0, & if\ h_2 \le 0 \\ x_2 * 1 * \frac{\partial L}{\partial h_2}, & if\ h_2 > 0 \end{cases}$$

$$\frac{\partial L}{\partial b_1} = \frac{\partial a_1}{\partial b_1} * \frac{\partial h_1}{\partial a_1} * \frac{\partial L}{\partial h_1} = \begin{cases} 0, \ if \ h_1 \leq 0 \\ 1 * 1 * \dfrac{\partial L}{\partial h_1}, \ if \ h_1 > 0 \end{cases}$$

$$\frac{\partial L}{\partial b_2} = \frac{\partial a_2}{\partial b_2} * \frac{\partial h_2}{\partial a_2} * \frac{\partial L}{\partial h_2} = \begin{cases} 0, \ if \ h_2 \leq 0 \\ 1 * 1 * \dfrac{\partial L}{\partial h_2}, \ if \ h_2 > 0 \end{cases}$$

经过简单的计算后，显而易见，需要的空间为 12*sizeof(double) 字节，使用 EEPROM 主要是为了防止掉电后数据丢失，而并非作为外部缓存。

## 三、基于 C 语言的类设计

源码已上传 $Github$ 开源，网址如下：

$$https://github.com/hhhhc-da/XOR\_Experiment$$

项目内含 $MPL-2.0\ license$

```c
// 缓冲区定位枚举类型
typedef enum
{
    // 输入层 x1
    x1 = 0x00,
    // 输入层 x2
    x2,
    // 隐藏层 h1
    h1,
    // 隐藏层 h2
    h2,
    // 输出层 o1
    o1,
    // 输出层 o2
    o2,
    // Loss 对 h1 的偏导数
    LTh1,
    // Loss 对 h2 的偏导数
    LTh2
} ucType;
```

```c
// 模型结构体
typedef struct
{
    // 损失集 (循环队列)
    double *loss, best_loss;
    unsigned char bare_rate;
    // 参数阵列
    double *W1, *W2, *B1, *B2;
    // 数据缓冲区(8*sizeof(double)字节大小)
    double *buffer;
    // 队列尾指针
    double *loss_end;
} xModel;
```

图 6：C 语言实现的模型类型及其枚举类型

在类设计的过程中，侧重于 C 语言的特性，统一使用了双层指针用于信息传递，在 $xModel$ 类中，有损失集指针，还有一个 $best\_loss$ 表示目前的最小损失，$bare\_rate$ 则是用于初始化损失集合表长度，表示训练在多少次后无效果后提前终止，由于网络设计 $W_1$，$W_1'$，$B_1$，$B_1'$ 写作 $W_1$，$W_2$，$B_1$，$B_2$ ，$buffer$ 表示数据缓冲区，用于存储输入层参数、隐藏层结果、输出层结果和损失函数对 $h_i$ 的偏导数 $\frac{\partial L}{\partial h_i}$ ，用于后期反向传播计算。

```c
// 初始化 xModel
void pvInit(xModel **model);
// 销毁 xModel
void pvDeInit(xModel **model);
// 写入缓冲区
void pvWriteBuffer(xModel **model, ucType pos, double data);
// 读取缓冲区
void pvReadBuffer(xModel **model, ucType pos, double *data);
// 清除缓冲区
void pvClearBuffer(xModel **model);
// 前向传播
void pvForward(xModel **model, double *x, unsigned ulCount);
// 反向传播
double pvBackward(xModel **model, double lr, double *y, unsigned ulCount);
// 输出结果
int ulGetResultIndex(xModel **model);
// 提前终止训练
unsigned char pvEarlyStopDetect(xModel **model);
// 报告参数和损失
void pvDisplayWeights(xModel **model);
// 学习率衰减
double lr_fall(unsigned epoch);
```

图 7：模型推理函数声明

同时，网络结构定型后，我们已经确定了前向传播算法流程，加以简易实现

```
67
68  // 前向传播
69  void pvForward(xModel **model, double *x, unsigned ulCount)
70  {
71      xModel *pModel = *model;
72      // 隐藏层和输出层
73      double *h = (double *)malloc(2 * sizeof(double));
74      double *o = (double *)malloc(2 * sizeof(double));
75
76      pvWriteBuffer(&pModel, x1, x[0]);
77      pvWriteBuffer(&pModel, x2, x[1]);
78
79      h[0] = ReLU(pModel->W1[0] * x[0] + pModel->W1[1] * x[1] + pModel->B1[0]);
80      h[1] = ReLU(pModel->W1[2] * x[0] + pModel->W1[3] * x[1] + pModel->B1[1]);
81
82      pvWriteBuffer(&pModel, h1, h[0]);
83      pvWriteBuffer(&pModel, h2, h[1]);
84
85      o[0] = pModel->W2[0] * h[0] + pModel->W2[1] * h[1] + pModel->B2[0];
86      o[1] = pModel->W2[2] * h[0] + pModel->W2[3] * h[1] + pModel->B2[1];
87
88      Logistic(&o, 2);
89
90      pvWriteBuffer(&pModel, o1, o[0]);
91      pvWriteBuffer(&pModel, o2, o[1]);
92
93      free(h);
94      free(o);
95  }
96
```

图 8：前向传播函数

同时，我们也列出反向传播

```
96
97  // 反向传播(返回这一个样本的 loss)
98  double pvBackward(xModel **model, double lr, double *y, unsigned ulCount)
99  {
100     // 参数导数存储缓存
101     double *dtBuf = (double *)malloc(12 * sizeof(double));
102     // dL/dh1 和 dL/dh2 - 因为偏导符号似乎容易出现乱码所以用 dx 来表示了
103     double *dtMid = (double *)malloc(2 * sizeof(double));
104     xModel *pModel = *model;
105
106     double newLoss = inf;
107
108     double o[2] = {-1, -1};
109     pvReadBuffer(&pModel, o1, &o[0]);
110     pvReadBuffer(&pModel, o2, &o[1]);
111
112     double h[2] = {-1, -1};
113     pvReadBuffer(&pModel, h1, &h[0]);
114     pvReadBuffer(&pModel, h2, &h[1]);
115
116     double x[2] = {-1, -1};
117     pvReadBuffer(&pModel, x1, &x[0]);
118     pvReadBuffer(&pModel, x2, &x[1]);
119
120     // 计算 Loss 作为提前停止训练凭证
121     newLoss = CrossEntropy(o, y, 2);
122
123     // dL/dh1 和 dL/dh2
124     dtMid[0] = pModel->W2[0] * (1 - o[0]) * (-y[0] / 2) + pModel->W2[2] * (1 - o[1]) * (-y[1] / 2);
125     dtMid[1] = pModel->W2[1] * (1 - o[0]) * (-y[0] / 2) + pModel->W2[3] * (1 - o[1]) * (-y[1] / 2);
126
127     // dL/dW2
128     dtBuf[0] = h[0] * (1 - o[0]) * (-y[0] / 2);
129     dtBuf[1] = h[1] * (1 - o[0]) * (-y[0] / 2);
130     dtBuf[2] = h[0] * (1 - o[1]) * (-y[1] / 2);
131     dtBuf[3] = h[1] * (1 - o[1]) * (-y[1] / 2);
```

图 9：反向传播函数

由于不用实现自动微分，所以这段基本就是照着求出来的导数翻译，然后更新参

数后返回单条数据的损失。

之后命令行编译

$$gcc\ .\backslash Main.c\ .\backslash NeuralNetwork.c\ .\backslash XORModel.c\ -o\ Main.exe$$

运行即可验证程序准确性

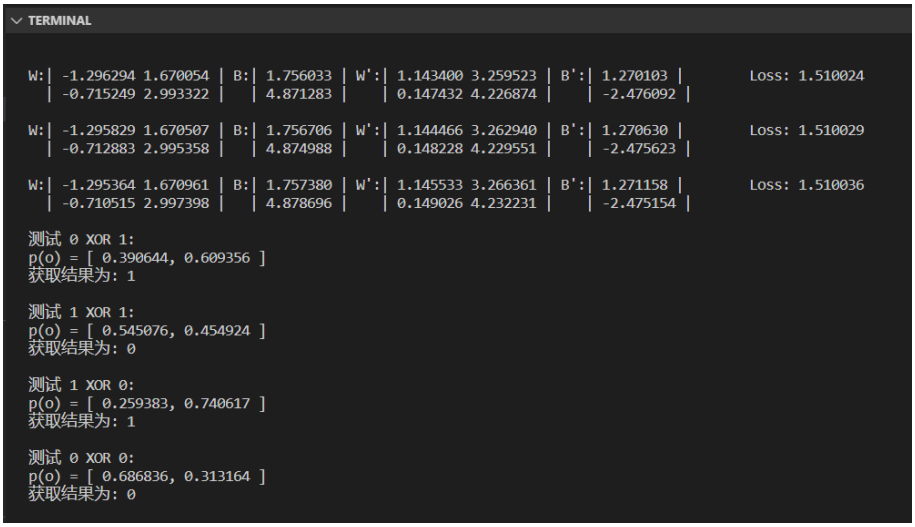


图 10：计算结果验证

至此，我们已经完成在 PC 平台上 XOR 实验的部署和验证。

## 四、STM32 的部署

首先创建新工程，之后编写好 $AT24C256$ 的驱动库和 $OLED$ 的 $I^2C$ 驱动库

main.c AT24C256.c

```
35  * 函数功能            : 利用 IIC 读取某地址上的某一个字节
36  * 注意事项            : 无
37  *****************************************************************************/
38  uint8_t AT24C256_ReadOneByte(uint16_t ReadAddr) {
39      uint8_t temp=0;
40      IIC_Start();
41
42      if(EE_TYPE>AT24C16){
43          IIC_Send_Byte(0XA0); //发送写命令
44          IIC_Wait_Ack();
45          IIC_Send_Byte(ReadAddr>>8);//发送高地址
46      }
47      else {
48          IIC_Send_Byte(0XA0+((ReadAddr/256)<<1));//发送器件地址,写
49      }
50
51      IIC_Wait_Ack();
52      IIC_Send_Byte(ReadAddr%256); //发送低地址
53      IIC_Wait_Ack();
54      IIC_Start();
55
56      IIC_Send_Byte(0XA1); //进入接收模式
57
58      IIC_Wait_Ack();
59      // NACK 有不再接收数据的意思
60      temp = IIC_Read_Byte(0);
61      IIC_Stop(); //产生一个停止条件
62
63      return temp;
64  }
65
66  /*****************************************************************************
67  * 函 数 名            : WriteAT24C256
68  * 函数功能            : 利用 IIC 连续写入某地址
```

图 11：AT24C256 驱动代码

将上一步编写好的 XOR 网络模型放入项目文件夹，添加编译搜索文件夹，将源文件添加到工程文件。使用 ARMCC 编译器，也就是 5.x 版本的老编译器，与 $GCC$ 类似，不用担心报错问题（请谨慎考虑是否要用 $CLANG$ 编译器）。值得一提的是，原本的 $STM32F103C8T6$ 和 $STM32F103C6T6A$ 完全不足以支持神经网络的运行，所以在本章图片内，使用的是标准库进行开发，而后期实验发现不行后，使用的是 $RAM$ 较大的 $STM32F407VET6$ ，同时开发转为 $HAL$ 库开发。

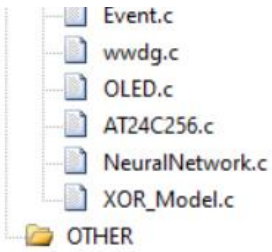


图 12：添加神经网络源文件

由于 STM32 片上 RAM 太小，所以经测试，他最多跑 17 轮的训练（因为动态内存碎片化和其他因素，代码都有释放内存，申请不回收导致的溢出概率较低），使用串口调试工具获取 $Debug$ 信息（本实验进行五轮 Epoch，实验现象如下）

```
Epoch[4]

W:| -1.881080 -4.617500 | B:| 1.440412 | W':| -0.892960 7.552684 | B':| 1.922998 |      Loss: 0.737550
  | 6.879378 3.670205 |   | 7.584059 |    | -5.832800 7.393023 |    | 4.693171 |


Epoch[5]

W:| -1.881080 -4.617500 | B:| 1.440410 | W':| -0.892957 7.557011 | B':| 1.923238 |      Loss: 0.737550
  | 6.882640 3.673015 |   | 7.588349 |    | -5.832800 7.397435 |    | 4.693506 |

测试 0.000 XOR 1.000:
p(0) = [ 0.274239, 0.725761 ]
获取结果为: 1

测试 1.000 XOR 0.000:
p(0) = [ 0.386739, 0.613261 ]
获取结果为: 1

测试 1.000 XOR 1.000:
p(0) = [ 0.531231, 0.468769 ]
获取结果为: 0

测试 0.000 XOR 0.000:
p(0) = [ 0.996151, 0.003849 ]
获取结果为: 0

p(0) = [ 0.531231, 0.468769 ]
1.000 XOR 1.000 = 0
p(0) = [ 0.531231, 0.468769 ]
1.000 XOR 1.000 = 0
p(0) = [ 0.531231, 0.468769 ]
1.000 XOR 1.000 = 0
p(0) = [ 0.531231, 0.468769 ]
1.000 XOR 1.000 = 0
```

单条发送 多条发送 协议传输 帮助

正点原子技术论坛：开源电子网:www.openedv.com

图 13：网络训练过程和测试

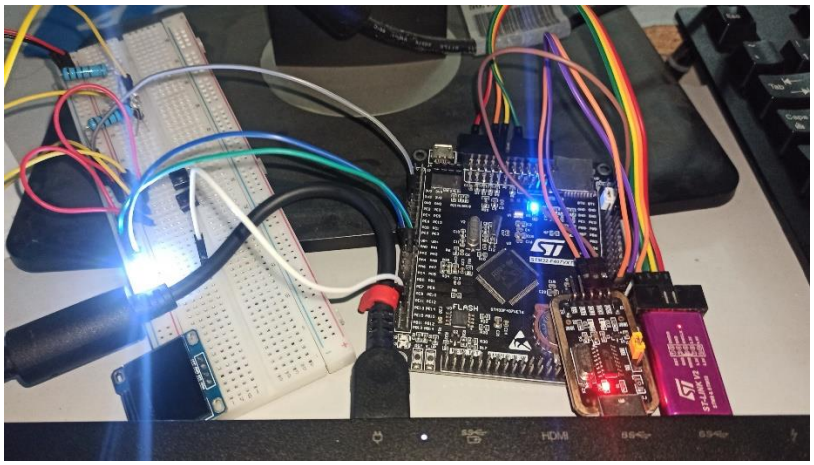

图 14：测试 $1\ XOR\ 0$ 结果为 1

其中，所使用的各项参数为：

$B_1, B_1', B_2, B_2'$ 分别对应

$$\mathbf{1.440419，1.922039，7.566921，4.691829}$$

$W_{11}, W_{11}', W_{12}, W_{12}', W_{21}, W_{21}', W_{22}, W_{22}'$ 分别对应

$$\mathbf{-1.881080，-0.892971，-4.617500，7.535398}$$

$$\mathbf{6.866347，-5.832800,，3.658982,，7.375399}$$

然后我们把参数存储到 $AT24C256$ 中，避免每次都需要训练，并且上电后可以直接从中读取最新数据。在写入前，根据 $AT24C256$ 的存储特性，需要按照字节存储，但是各参数均为双精度浮点型数据，不能直接存储。所以我们定义联合体

```
typedef union{
  uint8_t byte[8];
  double data;
}double_data_t;
```

图 15：联合体定义

之后我们将双精度浮点数写入联合体后按数组对其进行存储（由于 $double$ 类型数据占用 8 字节，所以联合体我们使用了 8 个无符号字符型数据做存储体）。

```
267
268 void saveModel(xModel **model){
269     // 写入数据格式 W11,W12,W21,W22,B1,B2,W11',W12',W21',W22',B1',B2'，写入到地址 0x00
270     double_data_t *temp = (double_data_t*)malloc(sizeof(double_data_t));
271     xModel *pTemp = *model;
272
273     // 使用共用体指针批量存储
274     temp->data = pTemp->W1[0];
275     WriteAT24C256(0x00,temp->byte,8);
276     HAL_Delay(100);
277     temp->data = pTemp->W1[1];
278     WriteAT24C256(0x08,temp->byte,8);
279     HAL_Delay(100);
280     temp->data = pTemp->W1[2];
281     WriteAT24C256(0x10,temp->byte,8);
282     HAL_Delay(100);
283     temp->data = pTemp->W1[3];
```

图 16：保存模型函数 $saveModel$

之后让 $OLED$ 间隔 $100ms$ 刷新屏幕即可。

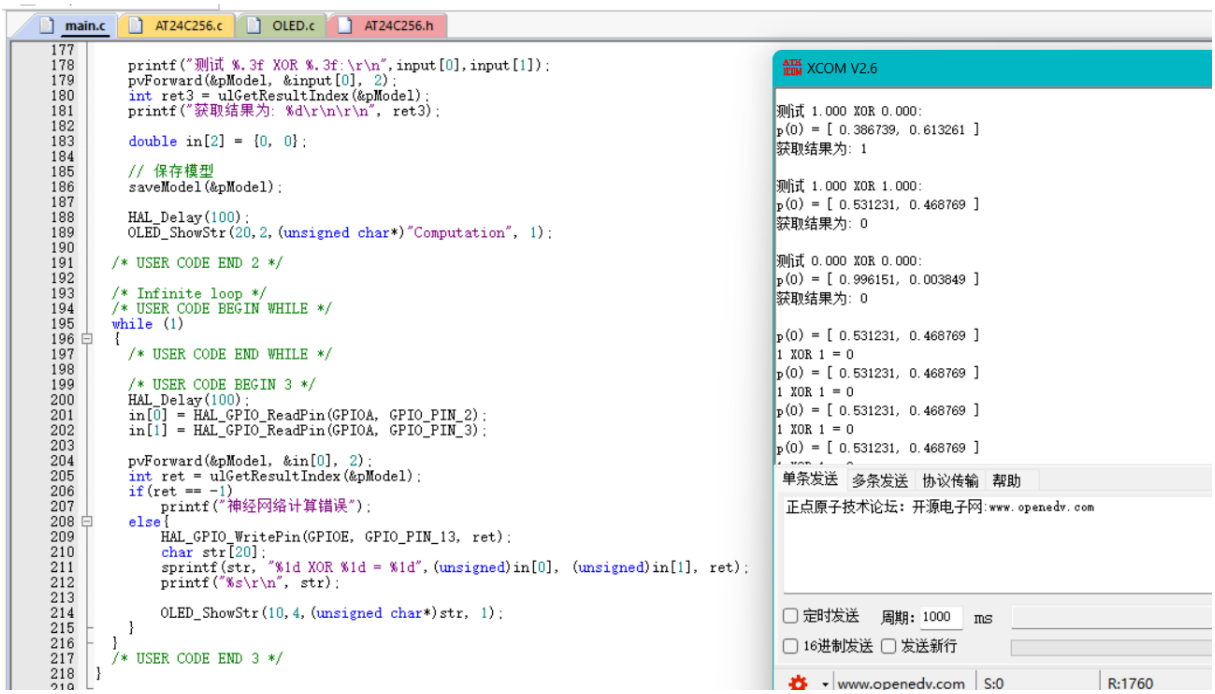


图 17：循环 $Loop$

不难看出，$I^2C$ 没有打印出任何错误日志，即通信成功。我们使用逻辑分析仪查看一下逻辑。

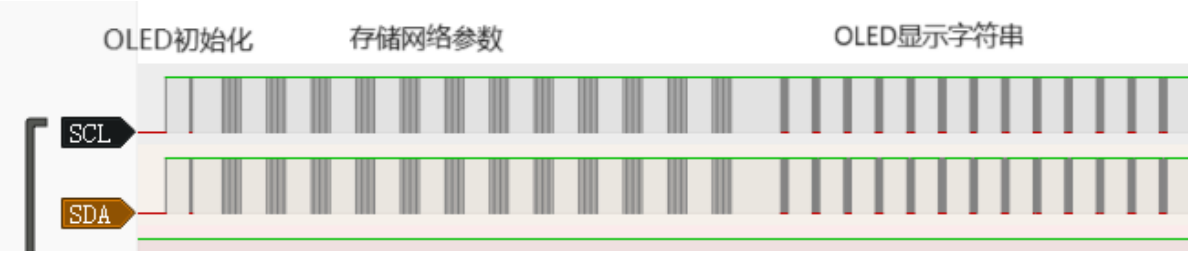


图 18：逻辑分析仪截获的 $SCL$ 和 $SDA$ 图像

实验现象如下：

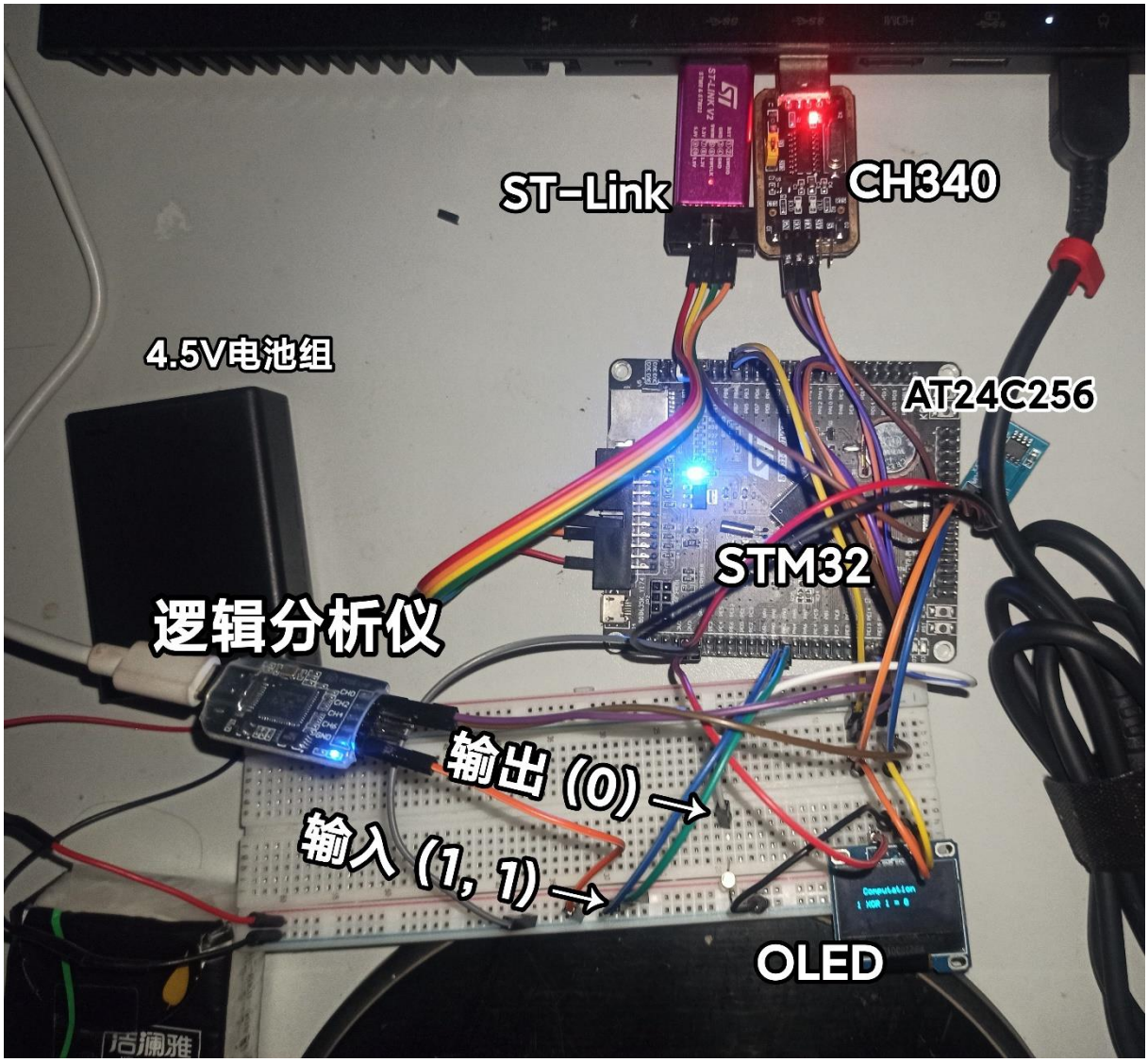


图 19：逻辑分析仪截获的 $SCL$ 和 $SDA$ 图像

至此，我们已经得到了预期的实验结果，实验结束。

# 参考文献

[1] 邱锡鹏. 神经网络与深度学习[M]. 第一版. https://nndl.github.io/, 2019.4.6.

[2] 正点原子. STM32F103 Datasheet[EB/OL]. [2015.6]. https://www.st.com/.