实验报告 2

GPIO 的使用与线性回归模型

范皓年 1900012739　信息科学技术学院

2020 年 10 月 23 日

1　实验目的

1. 了解树莓派的 GPIO 接口，简单了解 Pienoeer 扩展板在树莓派的对外通信机制中的角色。
2. 了解树莓派 RPi.GPIO 模块的基本使用，注意电平和 PWM 两种方式的输出。
3. 掌握树莓派 GPIO 接口的编程方式和使用方法。
4. 了解线性回归和机器学习的基本概念，完成一个简单的反馈调节网络。

2　实验原理

2.1　树莓派 GPIO 接口与 Pioneer600 扩展板

GPIO：General-purpose input/output，意为通用可编程输入输出接口，顾名思义，其主要功能即是通过寄存器，利用相对简单的布线实现相对复杂的功能。

用它替代了串口、并口等复杂结构，还大大降低了成本，在诸如树莓派的嵌入式设备中，可以较多地集成 GPIO，从而以较低成本实现功能的多样性。这种数量带来的优势是树莓派强大的重要原因。低功耗、低成本是嵌入式系统的重要优点，因而 GPIO 优势明显。

Pioneer600 扩展板中带有大量可与树莓派的 GPIO 交互的接口，同时集成了嵌入式系统的常见重要器件。极大地拓展了树莓派 3b 的功能。

本次实验中我们使用了 LED 灯、按钮以及其和树莓派的 IO 通信。

2.2　Python 中 RPi.GPIO 的使用

Python 作为 Pi 的编程语言，在 Pi GPIO 交互过程中，其 RPi.GPIO 模块就显得至关重要。

实验原理部分我们只给出重要的模块接口，更多的实例我们附在实验内容中。下表为 GPIO 的基本使用方法：

操作	Python 语句
导入	import RPi.GPIO as GPIO
选定编号模式为 BCM（本实验）	GPIO.setmode(GPIO.BCM)
选定编号模式为 BOARD	GPIO.setmode(GPIO.BOARD)
设定输入通道	GPIO.setup(channel, GPIO.IN, GPIO.PUD_UP)
设定输出通道	GPIO.setup(channel, GPIO.OUT)
创建一个通道的 PWM 实例	pwm = GPIO.PWM(channel, frequency)
启动 PWM，并指定占空比 dc	pwm.start(dc)
更改 PWM 脉冲重复的频率为 frequency	pwm.ChangeFrequency(freq)
更改 PWM 的占空比为 dc	pwm.ChangeDutyCycle(dc)
停止 PWM	pwm.stop()
释放脚本中的引脚	GPIO.cleanup()

2.3 Python 异常

和 C++ 中异常一样，Python 的异常捕获机制可以让程序具有更好的容错性，当程序运行出现意外情况时，系统会自动生成一个 Error 对象来通知程序，从而实现将“业务实现代码”和“错误处理代码”分离，提供更好的可读性。

这一点在 GPIO 相关实验中更加重要的原因是，如果中途发生了异常但程序没有正常退出，那么很可能的结果是接口没有被释放掉，处于热状态，在下一次启动程序的时候会提示错误。当然我们可以关闭错误提示，但良好的异常处理可以避免这一点。

异常捕获机制的代码结构如下：

```
    try:
# 业务实现代码
...
except Error1:
# 遇到Error1错误时的处理代码
...
except Error2:
# 遇到Error2错误时的处理代码
...
finally:
# 不管是否发生异常，一定会执行的代码
...
```

我们常用的异常为 KeyboardInterrupt，这个异常可以使得用户可以使用 Ctrl + C 快捷地手动终端程序，在这个程序当中，如果中途退出还希望释放掉正在使用的 GPIO 接口，代码可如下：

```
try:
    while True: # 死循环
        time.sleep(1) # 休眠1 秒钟
except KeyboardInterrupt: # 键盘中断事件
```

```
5     GPIO.cleanup()
```

2.4 简易线性回归模型的建立

机器学习的重要理论和方法基础就是人工神经网络。基于信号处理的理论和参数调整的思想，依次迭代完成系统拟合的最优化。

本次实验当中，我们简化激活函数为恒等函数，将神经元模型简化为线性模型，通过学习的方式获得线性模型的参数，进而了解机器学习的基本方法。这样的模型一般被称为线性回归模型。

实验中采用了梯度下降法，其主要思想为，将各个参数在误差函数关于对应变量下降最快的方向（即梯度方向）进行微调，进行迭代调参并最终使得误差成为极小的。利用每一组数据完成的调参过程称为训练，用完所有的训练数据时完成一个 epoch。

梯度下降的核心想法，就是用最快的方法进行误差的最小化。

对于一个给定的数据集 $(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \cdots, (\mathbf{x}_m, y_m)$，其中 $\mathbf{x}_i$ 为一个向量，其中保存着可以确定一个唯一的 y_i 所必需的值。每个 epoch 利用这 m 组数据进行迭代调参，试图获得系统输入输出间关系的线性模型，来尽可能的根据输入值预测实际输出的值，即利用：

$$f(\mathbf{x}) = b + \sum_{j=1}^{n} w_j x_j$$

实现

$$f(\mathbf{x_i}) \simeq \mathbf{y_i}$$

具体方法是求误差函数并求其关于每个参数的梯度：

$$\begin{aligned} E &= f(\mathbf{x}_i - y_i)^2 = \Delta_i^2 \\ \frac{\partial E}{\partial w_j} &= 2(f(\mathbf{x}_i) - y_i)x_j = 2\Delta_i x_j \\ \frac{\partial E}{\partial b} &= 2(f(\mathbf{x}_i - y_i)) = 2\Delta_i \end{aligned}$$

由上我们可以看出，在迭代过程当中，与更新的量有关的中间变量仅为 Δ_i，我们每次迭代过程只需计算出其预测值和目标值的差即可。

根据这个差值 Δ_i 进行参数调整的过程当中，步长非常关键，如果过大则使得学习过程中各个参数在最优值两侧横跳，过小则学习进程会较慢。通常我们将学习率定义为 α，值为 0.01 即可。

另外我们可以作出一个小小的简化，梯度中出现的常数 2 在分析过程中应与学习率相乘，所以我们可以通过学习率的调整使其不出现在计算式中，也即得到每一步的操作如下：

$$w_j = w_j - \alpha\Delta_i x_j$$

其中 i 为训练集指标，j 为线性模型的参数指标。

3 实验内容

3.1 按键检测

这个实验中，我们利用树莓派上 GPIO 接口实现按键的输入检测。Pioneer600 扩展板上扩展了一个五向摇杆，摇杆可上下左右拨动，也可以按下。

遥杆按下的输入接到了树莓派 GPIO 的第 37 号引脚，BCM 编号为 20。试利用树莓派 RPi.GPIO 模块实现对 Pioneer600 扩展板五向摇杆的按键输入检测，按下一次按键就会向 GPIO 传入一个信号，从而检测到其活动并调用 call_back 函数，在屏幕上打印“KEY PRESS”。下面代码实现了对按键输入 GPIO 通道的初始化：

```
import RPi.GPIO as GPIO
KEY = 20
GPIO.setmode(GPIO.BCM)
GPIO.setup(KEY, GPIO.IN,GPIO.PUD_UP) # 上拉电阻
```

一种实现检测按键的方式是利用软件中断和回调函数。GPIO 模块包含 add_event_detect 函数，用来设置发生指定事件时刻的回调函数。函数的参数为通道 channel，触发方式（可选 GPIO.RISING 和 GPIO.FALLING），回调函数 callback，以及用于软件消抖时长（通常设为 200ms）。从而我们可以利用如下的回调函数以及行为检测来进行按键检测：

```
def cb(ch):
    print("KEY PRESSED")
GPIO.add_event_detect(channel, GPIO.RISING, callback=cb, bouncetime=200)
```

3.2 LED 指示灯的操作

利用树莓派上 GPIO 接口实现 LED 指示灯的输出控制。Pioneer600 扩展板 LED 指示灯的 BCM 编号为 26。类似按键检测当中的代码，我们将树莓派和 LED 相关的 GPIO 接口初始化为：

```
import RPi.GPIO as GPIO
LED = 26
GPIO.setmode(GPIO.BCM)
GPIO.setup(LED, GPIO.OUT)
```

为了进行规律的闪烁，我们还需要引入时间控制模块：

```
import time
```

通过调用其 sleep 函数我们可以控制 LED 规律变化，从而实现闪烁效果。

这里我们通过电平和 PWM 两种方式进行闪烁操作：

1. 高低电平可以通过直接调整 GPIO 的输出端口完成。

2. PWM 的原理是改变周期内的信号占空比，输出 PWM 信号的过程中，占空比 =1 即为高电平，占空比 =0 则为低电平。

以下给出两种实现的 Python 代码：

高低电平的实现法：

```
while True:
    GPIO.output(LED, 1)
    time.sleep(0.2)
    GPIO.output(LED, 0)
    time.sleep(0.2)
```

```
pwm = GPIO.PWM(LED, 50)
pwm.start(0)
while True:
    pwm.ChangeDutyCycle(100)
    time.sleep(0.2)
    pwm.ChangeDutyCycle(0)
    time.sleep(0.2)
```

这里为了节省空间，我们去除了相关的错误处理代码。完整代码将附在后面。

呼吸灯效果只需要用两个相反的循环使得光强递增、递减即可：

```
while True:
    for dc in range(0, 101, 5):
        pwm.ChangeDutyCycle(dc)
        time.sleep(0.05)
    for dc in range(100, -1, -5):
        pwm.ChangeDutyCycle(dc)
        time.sleep(0.05)
```

3.3 按键 LED 联调

利用按键切换 LED 指示灯的闪亮模式。

3.3.1 LED 开关灯

利用呼吸灯代码，我们每按键一次更换呼吸方向，并执行一次呼吸即可。

```
cb1_mode = True
def cb1(ch):
    global cb1_mode
    if cb1_mode:
        lighter()
    else:
        darker()
    if cb1_mode:
        cb1_mode = False
```

```
    else:
        cb1_mode = True
```

其中 lighter 函数和 darker 函数是呼吸灯实验当中的变亮和变暗的两个循环。

在这里因为涉及较多的 GPIO 接口，所以如果出现异常退出，仍将有脚本中占用的接口未被释放。所以这里我们将异常处理的代码加入：

```
def breath_led():
    try:
        GPIO.add_event_detect(channel, GPIO.RISING, \
            callback=cb1, bouncetime=200)
    except KeyboardInterrupt:
        pwm.stop() # 停用pwm实例
        GPIO.cleanup() # 清除脚本中使用的GPIO接口
        GPIO.remove_event_detect(channel) # 停用行为检测

if __name__ == "__main__":
    breath_led()
```

3.3.2 按键控制闪烁的开关和变频

实验要求单击按键使 LED 灯进入闪烁模式，再次单击按键闪烁频率加倍，双击按键停止闪烁。

首先我们确定闪烁和不闪是两个大的方向，所以定义 cb2_mode 变量对闪烁和不闪加以区别，如果闪烁，则进入闪烁循环，不闪烁则进入熄灭循环。

```
try:
    while True:
        while cb2_mode:
            pwm.ChangeDutyCycle(0)
            time.sleep(sleeptime)
            pwm.ChangeDutyCycle(100)
            time.sleep(sleeptime)
        while ~cb2_mode:
            pwm.ChangeDutyCycle(0)
            time.sleep(sleeptime)
except KeyboardInterrupt:
    print("Ending...")
    GPIO.cleanup()
except:
    print("Unknown error!")
    GPIO.cleanup()
```

回调函数只负责调整参数。

- 如果按键时发现当前不闪（cb2_mode 为 False），则直接打开闪烁，同时定义频率初态。
- 否则频率加倍。
- 频率过高时，可能无法分辨，所以添加响应的回复逻辑。
- 另外如果两次按键时间过于接近，则判断为双击，关闭闪烁模式即可。

```
sleeptime = 0
last_time = 0
cb2_mode = False
def cb2(ch):
    global last_time
    global sleeptime
    global cb2_mode
    if time.time() - last_time > 0.5:
        if cb2_mode == False:
            cb2_mode = True
            sleeptime = 1.6
            print("Twinkle starting...")
        else:
            sleeptime = sleeptime / 2
            print("Freq doubled...")
        if sleeptime < 0.2:
            sleeptime = 0.8
            print("Freq reinit...")
    else:
        cb2_mode = False
        print("Turning down...")
    last_time = time.time()
```

主函数中行为检测的开启：

```
global cb2_mode
try:
    GPIO.add_event_detect(channel, GPIO.RISING, \
        callback=cb2, bouncetime=200)
except KeyboardInterrupt:
    pwm.stop()
    GPIO.cleanup()
```

完整代码附于报告后。

3.4 线性回归模型的建立

假定线性回归模型参数个数为 2，即

$$f(\mathbf{x}) = w_1 x_1 + w_2 x_2 + b$$

其中 w_1, w_2, b 为三个。训练数据通过随机的方式产生。

训练集的建立：

```
len = 2000
x1 = np.random.rand(len) * 10
x2 = np.random.rand(len) * 10
y = x1 * m1 + x2 * m2 + m3 + np.random.randn(len)
```

初值设定：

```
# 参数目标值
m1 = 2
m2 = 1
m3 = 5
# 参数初始值
w1 = 1
w2 = 1
b = 1
#学习率
alpha = 0.01
```

按照 2.4实验原理部分的分析，取 $m = 2000, n = 2$，对于训练集中第 i 套训练数据，作如下四个操作：

$$\Delta_i = y_i - y_{std}$$
$$w_1 = w_1 - \alpha \Delta_i x_1$$
$$w_2 = w_2 - \alpha \Delta_i x_2$$
$$b = b - \alpha \Delta_i$$

所以对每个 epoch，将 2000 套数据集遍历训练，有如下代码：

```
def epoch():
    global w1
    global w2
    global b
    global y
    global x1
    global x2
    global error

```

```
    for i in range(len):
        yi = x1[i] * w1 + x2[i] * w2 + b
        d = yi - y[i]
        diff = [alpha * d * x1[i], alpha * d * x2[i], alpha * d]
        w1 = w1 - diff[0]
        w2 = w2 - diff[1]
        b = b - diff[2]
        print(diff)
    print("one epoch finished")
```

按一次键，调用一次 epoch 函数即可。为了表征学习结果，我们还是计算出误差函数，在循环中发现误差极小时退出即可。

```
error = d * d
if error < 1e-5:
    pwm.ChangeDutyCycle(100)
    break
```

当把误差设定为 1e-5 不加噪声时，很快即可完成拟合。

3.5 思考题

当学习率过大的时候，很容易在最优解附近的反复横跳，反而难以很快收敛，甚至导致发散，如图：

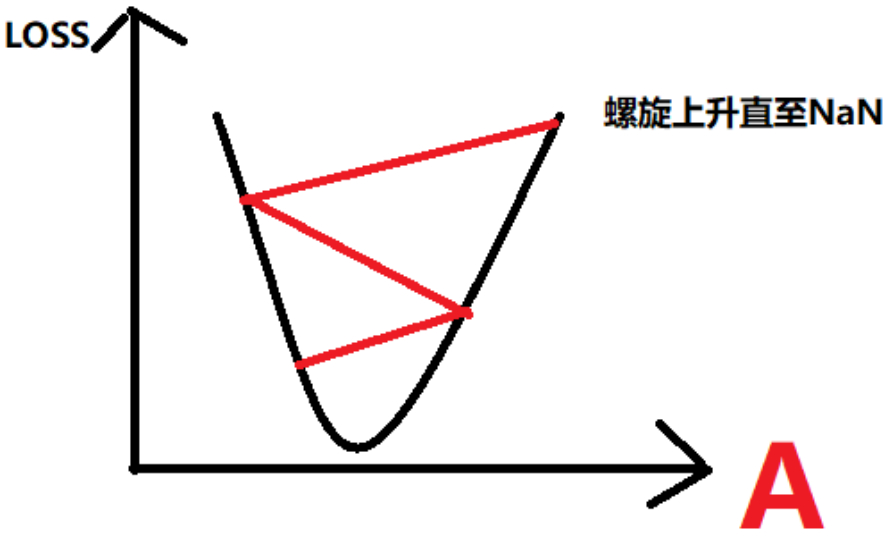

图 1: 学习率过大导致发散

检测方法也是直观的，每组训练观察梯度方向。如果梯度出现频繁的正负交替则判断其学习率过大。

A 附录代码

按键控制闪烁的开关和变频：

```
import RPi.GPIO as GPIO
import time
GPIO.setwarnings(False)

LED = 26
GPIO.setmode(GPIO.BCM)
GPIO.setup(LED, GPIO.OUT)
pwm = GPIO.PWM(LED, 50)
pwm.start(0)
for dc in range(0, 101, 5):
    pwm.ChangeDutyCycle(dc)
    time.sleep(0.05)
for dc in range(100, -1, -5):
    pwm.ChangeDutyCycle(dc)
    time.sleep(0.05)
print("LED Init ... Finished")

channel = 20
GPIO.setmode(GPIO.BCM)
GPIO.setup(channel,GPIO.IN, GPIO.PUD_UP)
print("button Init ... Finished")


sleeptime = 0
last_time = 0
cb2_mode = False
def cb2(ch):
    global last_time
    global sleeptime
    global cb2_mode
    if time.time() - last_time > 0.5:
        if cb2_mode == False:
            cb2_mode = True
            sleeptime = 1.6
            print("Twinkle starting ...")
        else:
            sleeptime = sleeptime / 2
            print("Freq doubled ...")
```

```
        if sleeptime < 0.2:
            sleeptime = 0.8
            print("Freq reinit...")
    else:
        cb2_mode = False
        print("Turning down...")
    last_time = time.time()


if __name__ == "__main__":
    global cb2_mode
    try:
        GPIO.add_event_detect(channel, GPIO.RISING,\
             callback=cb2, bouncetime=200)
    except KeyboardInterrupt:
        pwm.stop()
        GPIO.cleanup()
    try:
        while True:
            while cb2_mode:
                pwm.ChangeDutyCycle(0)
                time.sleep(sleeptime)
                pwm.ChangeDutyCycle(100)
                time.sleep(sleeptime)
            while ~cb2_mode:
                pwm.ChangeDutyCycle(0)
                time.sleep(sleeptime)
    except KeyboardInterrupt:
        print("Ending...")
        GPIO.cleanup()
    except:
        print("Unknown error!")
        GPIO.cleanup()
```

线性模型：

```
import RPi.GPIO as GPIO
import numpy as np
GPIO.setwarnings(False)
channel = 20
GPIO.setmode(GPIO.BCM)
GPIO.setup(channel,GPIO.IN, GPIO.PUD_UP)
```

```
LED = 26
GPIO.setup(LED, GPIO.OUT)
pwm = GPIO.PWM(LED, 50)
pwm.start(0)

m1 = 2
m2 = 1
m3 = 5

alpha = 0.01
w1 = 1
w2 = 1
b = 1
len = 2000
x1 = np.random.rand(len) * 10
x2 = np.random.rand(len) * 10
y = x1 * m1 + x2 * m2 + m3 + np.random.randn(len)
error = 2147483647

def cb(ch):
    epoch()

def epoch():
    global w1
    global w2
    global b
    global y
    global x1
    global x2
    global error

    for i in range(len):
        yi = x1[i] * w1 + x2[i] * w2 + b
        d = yi - y[i]
        error = d * d
        if error < 1e-5:
            pwm.ChangeDutyCycle(100)
            break
        diff = [alpha * d * x1[i], alpha * d * x2[i], alpha * d]
        w1 = w1 - diff[0]
        w2 = w2 - diff[1]
```

```
        b = b - diff[2]
        print(diff)
    print("one epoch finished")


if __name__ == "__main__":
    GPIO.add_event_detect(channel, GPIO.RISING, callback=cb, bouncetime=200)
```