

实验报告 6

AD/DA 与人工神经网络模型

范皓年 1900012739 信息科学技术学院

2020 年 11 月 21 日

目录

1 实验目的	2
2 实验原理	2
2.1 模拟和数字量的相互转换	2
2.2 PCF8591 芯片及其 AD/DA 操作	3
2.3 Python 多线程	5
2.4 Pytorch 的神经网络相关组件	5
2.5 人工神经网络解析	7
3 实验内容	10
3.1 AD/DA 相关模块的使用	10
3.1.1 AD 实验	11
3.1.2 DA 操作	12
3.1.3 AD/DA 联动：利用电位控制呼吸灯频率	13
3.2 利用 PyTorch 模型建立简单的多层感知机	15
A 附录代码	17

1 实验目的

1. 熟悉 AD/DA 原理。
2. 通过 I^2C 总线访问扩展板的 PCF8591 AD/DA 芯片。
3. 了解 keras 框架和简单的人工神经网络模型搭建。

2 实验原理

2.1 模拟和数字量的相互转换

模拟量是自然物理世界中存在的大量的连续量。而在实际的电子系统当中，模拟量的获取、保存、传输都不容易，所以我们需要把模拟量转换为离散表示的数字量，可以进行这种转换的设备称为模数（A/D）转换器。在单端输入情况下，A/D 转化启动时，对 A/D 输入的模拟信号与地之间的差值与 A/D 的最小分辨率比较，根据比较的值量化输出得到最终的 AD 值（如图 1）

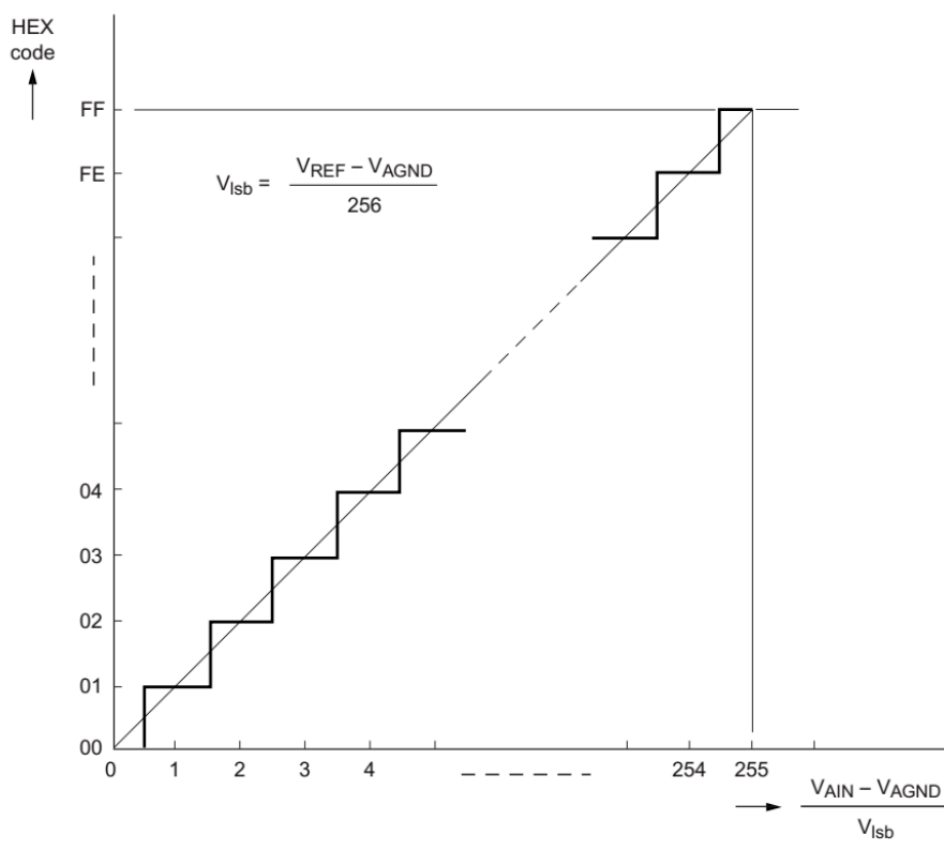


图 1: AD 的实现原理

计算方法为：

$$Val = \left\lceil \frac{V_A + V_{AGND}}{V_{lsb}} + 1 \right\rceil \quad (1)$$

A/D 的最小分辨率与 A/D 的位数有关。如在 8bit 条件下，A/D 的最小分辨电压为：

$$V_{lsb} = \frac{V_{ref} - V_{gnd}}{256} \quad (2)$$

数字量经过电子系统的加工处理之后，需要重新和外界交互，要重新转换为模拟信号。把数字量转换为模拟量的设备称为数模（D/A）转换器。如图 2，数模转换根据输入（8bit 数据）进行 8bit-→256 译码，将译码的输出值驱动分档开关，最后输出不同的电压值，完成 D/A 转换。

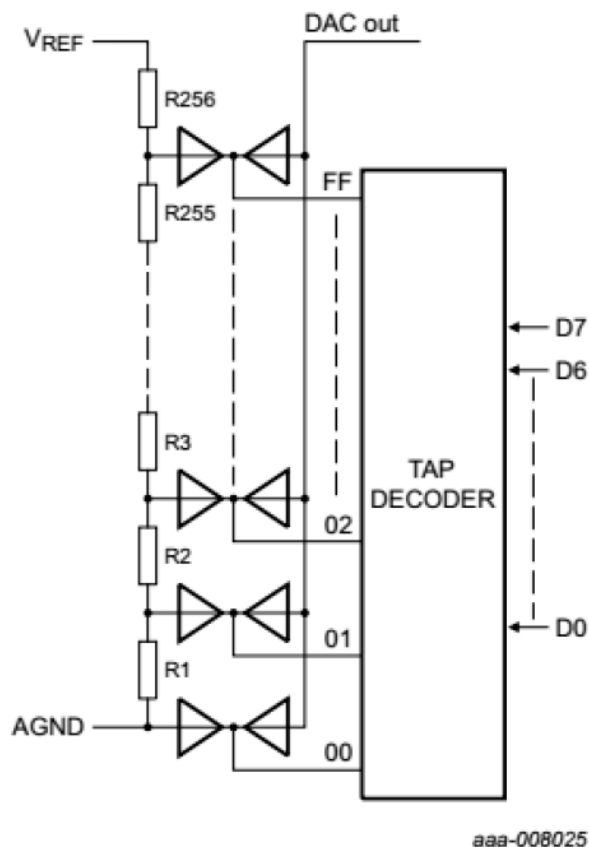


图 2: DA 的转换原理

对于一个 8-bit 数字信号（D0, D1, ..., D7），恢复出的模拟量为：

$$V_{Ao} = \sum_{i=0}^7 2^i \times D_i \quad (3)$$

2.2 PCF8591 芯片及其 AD/DA 操作

PCF8591 芯片也挂载在树莓派的 I²C 总线上，其内部框图如图3(a)所示。

扩展板上 PCF8591 的 A0/A1/A2 均接低电平，根据 PCF8591 地址配置规则（表1），PCF8591 的地址为 0x48。PCF8591 集成了多路 AD/DA 功能，在使用其功能时，需要对其控制寄存器进行控制，控制寄存器的定义如图3(b)。

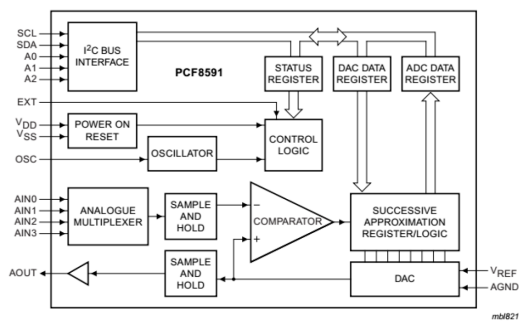
控制寄存器共有 8bit，通过配置不同 bit，来实现 A/D 工作模式、通道选择及 D/A 输出功能的定义。

在树莓派端利用 Python 实现 AD/DA 的相关初始化以及 API 如下：

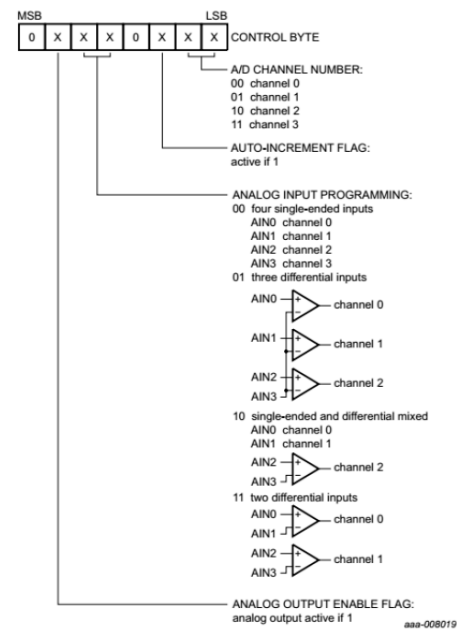
```
1 import smbus
2 import time # 包含相关库文件
```

位	7	6	5	4	3	2	1
地址	1	0	0	1	A2	A1	A0

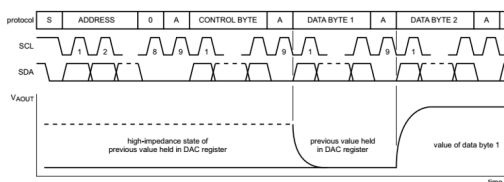
表 1: PCF8591 地址配置



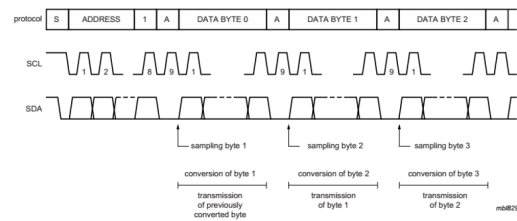
(a) PCF8591 内部框图



(b) PCF8591 控制寄存器定义



(c) PCF8591 D/A 接口时序



(d) PCF8591 A/D 接口时序

图 3: PCF8591 扩展板相关的硬件结构和性质

```
3 address = 0x48
4 A0 = 0x40
5 bus = smbus.SMBus(1) # 初始化 i2c Bus
6
7 # A/D: value 为数字量
8 bus.write_byte(address, A0)
9 value = bus.read_byte(address)
10
11 # D/A
12 bus.write_byte_data(address, A0, value)
```

2.3 Python 多线程

以上的各自在一个死循环中进行 value 的读或写，显然，如果使用通常的编程方式，程序只能处在其中一个死循环中，从而不能有效地实现同时读写。基于这种问题，可以利用 Python 语言同时开启多个线程进行。

多线程是一种并行执行技术，在特定的应用环境下，采用并行执行的方式可以提高程序的执行效率，或者可以使程序编写简便，增加程序的可读性。实现并行执行的方法有很多种，这里介绍 threading 模块。使用 threading 模块建立线程类的简单方法是调用 Thread 构造函数：

```
1 from threading import Thread
2 mythread = Thread(target=callable, args=())
```

其中 callable 是新建线程所运行的函数名，在主程序中可以通过 mythread.start() 启动线程，调用 mythread.join() 等待线程结束。如果希望主程序退出时，线程可以自动退出，可以通过 mythread.setDaemon(True) 来设置。

需要注意的是这里的并行并没有利用多核处理器同时执行，而是采用分时的方式使得两部分的程序看起来都在运行。而在所有的并行运行技术中，都要注意共享资源的使用，避免竞争和死锁。

2.4 Pytorch 的神经网络相关组件

PyTorch 是由 Facebook 的人工智能研究小组在 2016 年开发的基于 Torch 的 Python 机器学习库。Pytorch 是 torch 的 python 版本，是由 Facebook 开源的神经网络框架。PyTorch 的程序可以立即执行计算，这正好符合 Python 的编程方法，不需要完成全部代码才能运行，可以轻松的运行部分代码并实时检查。PyTorch 支持互动式的调试，使得调试和可视化变得非常容易。与 Tensorflow 的静态计算图不同，PyTorch 的计算图是动态的，以便可以在运行时构建计算图，甚至在运行时更改它们，在不知道创建神经网络需要多少内存的情况下这非常有价值。PyTorch 可以顺利地 Python 数据科学栈集成。它非常类似于 numpy，甚至注意不到它们的差别。

对 PyTorch 的最基本理解包括如下三方面：

1. Numpy 风格的 Tensor（张量）操作
2. 变量自动求导

3. 神经网络层与损失函数优化等高层封装

Numpy 风格的 Tensor（张量）操作。Tensor 是神经网络框架中重要的基础数据类型，可以简单理解为 N 维数组的容器对象。tensor 之间的通过运算进行连接，从而形成计算图。PyTorch 中 tensor 提供的 API 参考了 Numpy 的设计，因此熟悉 Numpy 的用户基本上可以无缝理解并创建和操作 tensor，同时 torch 中的数组和 Numpy 数组对象可以无缝的对接。Torch 定义了七种 CPU tensor 类型和八种 GPU tensor 类型，torch 模块内提供了操作 tensor 的接口，而 Tensor 类型的对象上也设计了对应的接口，例如 `torch.add()` 与 `tensor.add()` 等价。

变量自动求导。tensor 对象通过一系列的运算可以组成动态图，每个 tensor 对象可以方便的计算自己对目标函数的梯度。这样就可以方便的实现神经网络的后向传播过程。

神经网络层与损失函数优化等高层封装。网络层的封装存在于 `torch.nn` 模块，损失函数由 `functional` 模块提供，优化函数由 `torch.optim` 模块提供。

本实验中需要使用到网络层 `torch.nn` 模块，它提供了创建神经网络的基础构件，这些层都继承自 `Module` 类。在 `nn.functional` 模块中，提供多种激活函数的实现。通常对于可训练参数的层使用 `module`，而对于不需要训练参数的层如 `softmax` 这些，可以使用 `functional` 中的函数。

在实验二中我们使用了简单的线性模型人工神经网络，实际上更常见的人工神经网络会有更多的层，中间的层称为隐藏层。

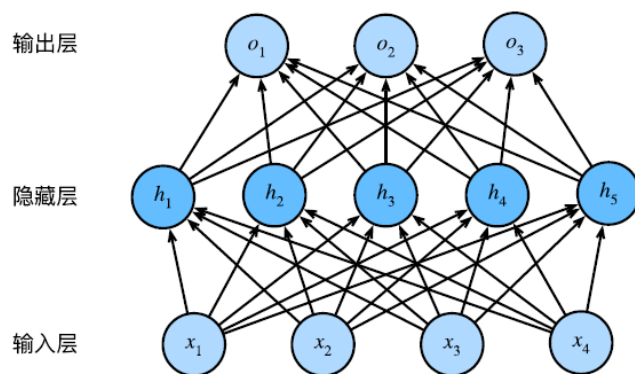


图 4

如图4是含一个隐藏层的网络。由于线性变换的组合仍然是线性变换，为了体现多层模型的价值，在每个感知器模型中都引入了非线性的激活函数如图7。

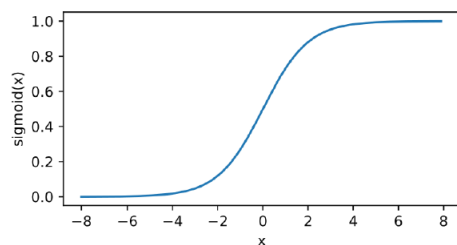


图 5: Sigmoid

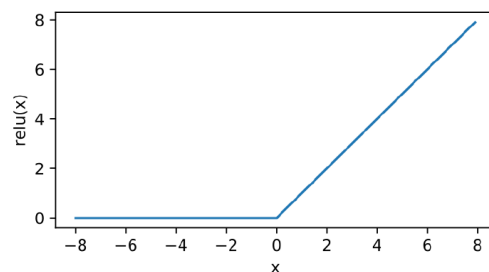


图 6: ReLU

图 7: 常见的非线性激活函数

用 PyTorch 定义如图7所示 MLP 神经网络的代码如下：

```
1 import torch.nn as nn
2 import torch.nn.functional as F
3 class Net(nn.Module):
4     def __init__(self):
5         super(Net, self).__init__()
6         self.hidden = nn.Linear(3, 5)
7         self.out = nn.Linear(5, 4)
8     def forward(self, x):
9         x = F.relu(self.hidden(x))
10        x = self.out(x)
11        return x
12 net=Net()
13 print(net)
```

2.5 神经网络解析

如图4给出了一个含隐藏层、非线性激活函数的人工神经网络，我们接下来给出一个多层感知机的实例，来具体说明人工神经网络的实现。

神经网络通常可以由如下步骤架构：

1. 准备数据
2. 定义网络结构 model
3. 定义损失函数
4. 定义优化算法 optimizer
5. 训练
6. 准备好 tensor 形式的输入数据和标签（可选）
7. 前向传播计算网络输出 output 和计算损失函数 loss
8. 反向传播更新参数：
9. 将上次迭代计算的梯度值清 0：optimizer.zero_grad()
10. 反向传播，计算梯度值：loss.backward()
11. 更新权值参数：optimizer.step()
12. 保存训练集上的 loss 和验证集上的 loss 以及准确率以及打印训练信息。（可选）
13. 图示训练过程中 loss 和 accuracy 的变化情况（可选）
14. 在测试集上测试

接下来的实例也将根据这个过程进行架构：

```
1 import torch
2 import torch.nn as nn
3 import torchvision
4 import torchvision.transforms as transforms
5 # 设备配置，部署GPU加速
6 device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
7 # Hyper-parameters, 定义各层规模及其他超参数
8 input_size = 784
9 hidden_size = 500
10 num_classes = 10
11 num_epochs = 5
12 batch_size = 100
13 learning_rate = 0.001
14 # MNIST dataset, 准备数据集，转换为Tensor数据格式
15 train_dataset = torchvision.datasets.MNIST(root='data',
16                                           train=True,
17                                           transform=transforms.ToTensor(),
18                                           download=True)
19 test_dataset = torchvision.datasets.MNIST(root='data',
20                                           train=False,
21                                           transform=transforms.ToTensor())
22 # DataLoader类，进行batch_size(每个batch的大小)，
23 # shuffle(是否进行shuffle操作)，
24 # num_workers(加载数据的时候使用几个子进程)等操作
25 train_loader = torch.utils.data.DataLoader(dataset=train_dataset,
26                                           batch_size=batch_size,
27                                           shuffle=True)
28 test_loader = torch.utils.data.DataLoader(dataset=test_dataset,
29                                           batch_size=batch_size,
30                                           shuffle=False)
31 # 定义网络结构，Fully connected neural network with one hidden layer
32
33 class NeuralNet(nn.Module):
34     def __init__(self, input_size, hidden_size, num_classes):
35         super(NeuralNet, self).__init__()
36         self.fc1 = nn.Linear(input_size, hidden_size)
37         self.relu = nn.ReLU()
38         self.fc2 = nn.Linear(hidden_size, num_classes)
39
40     def forward(self, x):
```



```
41     # 向前传播，模型的计算流程
42     out = self.fc1(x)
43     out = self.relu(out)
44     out = self.fc2(out)
45     return out
46
47 model = NeuralNet(input_size, hidden_size, num_classes)
48 # Loss and optimizer
49 # 定义损失函数，使用的是交叉熵函数
50 criterion = nn.CrossEntropyLoss()
51 # 定义迭代优化算法，使用的是Adam (Adaptive Moment Estimation)
52 optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
53 # Train the model 迭代训练
54
55 def train_model(model, train_loader):
56     total_step = len(train_loader)
57     for epoch in range(num_epochs):
58         for i, (images, labels) in enumerate(train_loader):
59             # Move tensors to the configured device
60             images = images.reshape(-1, 28 * 28).to(device)
61             labels = labels.to(device)
62             # Forward pass, 前向传播计算网络结构的输出结果
63             outputs = model(images)
64             # 计算损失函数
65             loss = criterion(outputs, labels)
66             # Backward and optimize 反向传播更新参数
67             # 将上次迭代计算的梯度值清0
68             optimizer.zero_grad()
69             # 反向传播，计算梯度值
70             loss.backward()
71             # 更新权值参数
72             optimizer.step()
73             # 打印训练信息
74             if (i + 1) % 100 == 0:
75                 print('Epoch [{}/{}], Step [{}/{}], loss: {:.4f}'
76                       .format(epoch + 1, num_epochs, i + 1, total_step, loss.item()))
77
78     # 保存训练好的模型
79     torch.save(model.state_dict(), 'model.ckpt')
80 def test_model(model, test_loader, device):
81     model.load_state_dict(torch.load('model.ckpt', map_location=device))
```

```

82     with torch.no_grad():
83         correct = 0
84         total = 0
85         for images, labels in test_loader:
86             images = images.reshape(-1, 28 * 28).to(device)
87             labels = labels.to(device)
88             outputs = model(images)
89             _, predicted = torch.max(outputs.data, 1)
90             total += labels.size(0)
91             correct += (predicted == labels).sum().item()
92         print('Accuracy is: {}'.format(100 * correct / total))
93 train_model(model, train_loader)
94 test_model(model, test_loader, device)

```

深度学习框架中涉及很多参数，下面介绍一下 batch、iterations 和 epochs 的概念：深度学习的优化算法，即梯度下降，每次的参数更新有两种方式：

1. 遍历全部数据集算一次损失函数，然后算函数对各个参数的梯度，更新梯度。这种方法每更新一次参数都要把数据集里的所有样本都看一遍，计算量开销大，计算速度慢，不支持在线学习，这称为 Batch gradient descent，批梯度下降。
2. 每看一个数据就算一下损失函数，然后求梯度更新参数，这个称为随机梯度下降，stochastic gradient descent。这个方法速度比较快，但是收敛性能不太好，可能在最优解附近晃来晃去，得不到最优解。两次参数的更新也有可能互相抵消掉，造成目标函数震荡的比较剧烈。

为了克服两种方法的缺点，现在一般采用的是一种折中手段，mini-batch gradient decent，小批的梯度下降，这种方法把数据分为若干个批，按批来更新参数，这样，一个批中的一组数据共同决定了本次梯度的方向，下降起来就不容易跑偏，减少了随机性。另一方面因为批的样本数与整个数据集相比小了很多，计算量也不是很大。基本上现在的梯度下降都是基于 mini-batch 的，所以深度学习框架的函数中经常会出现 batch_size，就是指这个。

每一次迭代都是一次权重更新，每一次权重更新需要 batch_size 个数据进行 Forward 运算得到损失函数，再 BP 算法更新参数。1 个 iteration 等于使用 batch_size 个样本训练一次。

epochs 被定义为向前和向后传播中所有批次的单次训练迭代。这意味着 1 个周期是整个输入数据的单次向前和向后传递。简单说，epochs 指的就是训练过程中数据将被“轮”多少次。

例如，训练集有 1000 个样本，batchsize=10，那么训练完整个样本集需要：100 次 iteration，1 次 epoch。

3 实验内容

3.1 AD/DA 相关模块的使用

实验电路图如下：

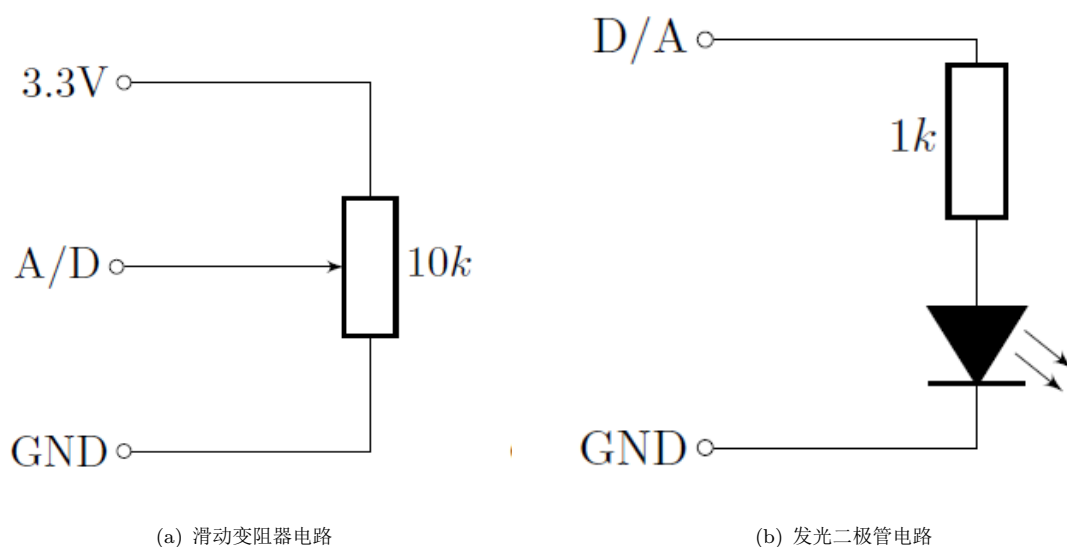


图 8: AD/DA 实验电路图

3.1.1 AD 实验

使用面包板，对电位器的电压信号进行采样并存储，在 OLED 屏幕上展示信号的电压信息。参考电路如图8(a)。

利用 AD 原理以及 Python 相关模块进行读出对应分压。通过先前 OLED 实验中的代码将电压值进行转换后显示在 OLED 屏幕上，利用代码将思路表示如下：

```

1  # AD相关模块导入
2  import smbus
3  import time
4
5  # SPI相关模块导入
6  import spidev as SPI
7  import SSD1306
8  from PIL import Image # 调用相关库文件
9  from PIL import ImageDraw
10 from PIL import ImageFont
11 RST = 19
12 DC = 16
13 bus = 0
14 device = 0 # 树莓派管脚配置
15
16 # 初始化屏幕相关参数及清屏
17 disp = SSD1306.SSD1306(rst=RST,dc=DC,spi=SPI.SpiDev(bus,device))
18 disp.begin()
19 disp.clear()

```

```
20 disp.display()
21 font = ImageFont.load_default()
22 # 显示欢迎界面
23 image = Image.new('RGB',(disp.width,disp.height),'black').convert('1')
24 draw = ImageDraw.Draw(image)
25 x = 30; y = 30;
26 draw.text((x,y), 'Hello, Pi!!', font=font, fill=255)
27 disp.image(image)
28 disp.display()
29 time.sleep(1)
30 disp.clear()
31 disp.display() # 初始化屏幕相关参数及清屏
32
33 # 进行OLED显示
34 address = 0x48
35 A0 = 0x40
36 bus = smbus.SMBus(1)
37 while True:
38     bus.write_byte(address, A0)
39     value = bus.read_byte(address) # 读出A/D端口中的电压值
40     voltage = value / 256 * 3.3 # 进行转换
41     # 显示
42     font = ImageFont.load_default()
43     image = Image.new('RGB',(disp.width,disp.height),'black').convert('1')
44     draw = ImageDraw.Draw(image)
45     x = 30; y = 30;
46     draw.text((x,y), 'Voltage: %.4f'%voltage, font=font, fill=255)
47     disp.image(image)
48     disp.display()
49     print(voltage)
```

3.1.2 DA 操作

调整 DA 输出, 点亮发光二极管, 设计二极管的发光模式 (如呼吸灯)。参考电路如图8(b)。将亮度作为数字值一次轮番遍历, 并将此值转化为模拟值输出到电路中完成周期性上拉实现呼吸灯。代码如下:

```
1 import smbus
2 import time
3
4 address = 0x48
5 A0 = 0x40
6 bus = smbus.SMBus(1)
```

```
7
8 while True:
9     for value in range(256):
10         bus.write_byte_data(address, A0, value)
11         print(value)
12     for value in range(255, -1, -1):
13         bus.write_byte_data(address, A0, value)
14         print(value)
```

3.1.3 AD/DA 联动：利用电位控制呼吸灯频率

在3.1.1中我们容易将电位读出，呼吸灯的频率控制，利用 `time.sleep()` 实现。在每个循环中，停息时间较长，则频率较低，反之频率较高。

注意其中的 AD/DA 分成两个线程实现。用于 AD 读取控制电压的程序利用函数封装进入子线程进行循环。DA 输出电压控制频率的程序直接在主函数中实现。这样能保证两个死循环互不干扰，且读取没有时间延迟：

```
1 import smbus
2 import time
3 from threading import Thread
4
5 import spidev as SPI
6 import SSD1306
7 from PIL import Image # 调用相关库文件
8 from PIL import ImageDraw
9 from PIL import ImageFont
10 RST = 19
11 DC = 16
12 bus = 0
13 device = 0 # 树莓派管脚配置
14
15 disp = SSD1306.SSD1306(rst=RST,dc=DC,spi=SPI.SpiDev(bus,device))
16 disp.begin()
17 disp.clear()
18 disp.display() # 初始化屏幕相关参数及清屏
19 font = ImageFont.load_default()
20 image = Image.new('RGB',(disp.width,disp.height),'black').convert('1')
21 draw = ImageDraw.Draw(image)
22 x = 30; y = 30;
23 draw.text((x,y), 'Hello, Pi!!', font=font, fill=255)
24 disp.image(image)
25 disp.display()
```

```
26 time.sleep(1)
27 disp.clear()
28 disp.display() # 初始化屏幕相关参数及清屏
29
30 address = 0x48
31 A0 = 0x40
32 bus = smbus.SMBus(1)
33 value = 0
34
35 def ad_read():
36     global value
37     while True:
38         bus.write_byte(address, A0)
39         value = bus.read_byte(address)
40         voltage = value / 256 * 3.3
41         font = ImageFont.load_default()
42         image = Image.new('RGB', (disp.width, disp.height), 'black').convert('1')
43         draw = ImageDraw.Draw(image)
44         x = 30; y = 30;
45         draw.text((x, y), 'Voltage: %.4f'%voltage, font=font, fill=255)
46         disp.image(image)
47         disp.display()
48
49 mythread = Thread(target = ad_read)
50 mythread.setDaemon(True)
51
52 try:
53     mythread.start()
54     while True:
55         for i in range(256):
56             bus.write_byte_data(address, A0, i)
57             if value == 0:
58                 value = 0.1
59             time.sleep(1/value)
60         for i in range(255, -1, -1):
61             bus.write_byte_data(address, A0, i)
62             if value == 0:
63                 value = 0.1
64             time.sleep(1/value)
65 except KeyboardInterrupt:
66     pass
```

3.2 利用 PyTorch 模型建立简单的多层感知机

将2.5中的的代码进行重构，将其中 MNIST 的图片数据集换成 scikit-learn 的 datasets 中的 8*8 数据。有如下几个要点：

- 将输入数据规模换成 64
- 导入数据之后将前半和后半分开成为训练数据和检验数据
- 导入图片要将 16 级归一化成为 torch 的图片数据格式
- 图片 reshape 参数时要将 28*28 换成 8*8
- 将标签 labels reshape

```

1 import torch
2 import torch.nn as nn
3 import torchvision
4 import torchvision.transforms as transforms
5 from sklearn import datasets
6 # 设备配置，部署GPU加速
7 device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
8 # Hyper-parameters, 定义各层规模及其他超参数
9 input_size = 64 # 将输入数据规模换成 64
10 hidden_size = 500
11 num_classes = 10
12 num_epochs = 5
13 batch_size = 100
14 learning_rate = 0.001
15
16 # sklearn dataset, 准备数据集，转换为 Tensor 数据格式
17 digits = datasets.load_digits()
18 # 导入图片要将 16 级归一化成为 torch 的图片数据格式
19 images_and_labels = list(zip(torch.from_numpy(digits.images/16).float(),
20                             torch.from_numpy(digits.target).long()))
21 n_samples = len(digits.images)
22
23 # 导入数据之后将前半和后半分开成为训练数据和检验数据
24 train_loader = images_and_labels[:n_samples//2]
25 test_loader = images_and_labels[n_samples//2:]
26 # 定义网络结构，Fully connected neural network with one hidden layer
27 class NeuralNet(nn.Module):
28     def __init__(self, input_size, hidden_size, num_classes):
29         super(NeuralNet, self).__init__()
30         self.fc1 = nn.Linear(input_size, hidden_size)

```

```

31         self.relu = nn.ReLU()
32         self.fc2 = nn.Linear(hidden_size, num_classes)
33     def forward(self, x):
34         # 向前传播，模型的计算流程
35         out = self.fc1(x)
36         out = self.relu(out)
37         out = self.fc2(out)
38         return out
39
40 model = NeuralNet(input_size, hidden_size, num_classes)
41 # Loss and optimizer
42 # 定义损失函数，使用的是交叉熵函数
43 criterion = nn.CrossEntropyLoss()
44 # 定义迭代优化算法，使用的是Adam (Adaptive Moment Estimation)
45 optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
46 # Train the model 迭代训练
47 def train_model(model, train_loader):
48     total_step = len(train_loader)
49     for epoch in range(num_epochs):
50         for i, (images, labels) in enumerate(train_loader):
51             # Move tensors to the configured device
52             # 图片 reshape 参数时要将 28*28 换成 8*8,
53             images = images.reshape(-1, 8 * 8).to(device)
54             labels = labels.reshape(1).to(device)
55
56             # Forward pass, 前向传播计算网络结构的输出结果
57             outputs = model(images)
58             # 计算损失函数
59             loss = criterion(outputs, labels)
60             # Backward and optimize 反向传播更新参数
61             # 将上次迭代计算的梯度值清 0
62             optimizer.zero_grad()
63             # 反向传播，计算梯度值
64             loss.backward()
65             # 更新权值参数
66             optimizer.step()
67             # 打印训练信息
68             if (i + 1) % 100 == 0:
69                 print('Epoch [{}/{}], Step [{}/{}], loss: {:.4f}'
70                       .format(epoch + 1, num_epochs, i + 1, total_step, loss.item()))
71 # 保存训练好的模型

```



```

72     torch.save(model.state_dict(), 'model_skl.ckpt')
73
74 # Test the model 测试模型
75 def test_model(model, test_loader, device):
76     model.load_state_dict(torch.load('model_skl.ckpt', map_location=device))
77     with torch.no_grad():
78         correct = 0
79         total = 0
80         for images, labels in test_loader:
81             images = images.reshape(-1, 8 * 8).to(device)
82             labels = labels.reshape(1).to(device)
83             outputs = model(images)
84             _, predicted = torch.max(outputs.data, 1)
85             total += labels.size(0)
86             correct += (predicted == labels).sum().item()
87         print('Accuracy is: {}'.format(100 * correct / total))
88
89 train_model(model, train_loader)
90 test_model(model, test_loader, device)

```

A 附录代码

A.1 ad.py

```

1  import smbus
2  import time
3
4  import spidev as SPI
5  import SSD1306
6  from PIL import Image # 调用相关库文件
7  from PIL import ImageDraw
8  from PIL import ImageFont
9  RST = 19
10 DC = 16
11 bus = 0
12 device = 0 # 树莓派管脚配置
13
14 disp = SSD1306.SSD1306(rst=RST,dc=DC,spi=SPI.SpiDev(bus,device))
15 disp.begin()
16 disp.clear()
17 disp.display() # 初始化屏幕相关参数及清屏

```

```
18 font = ImageFont.load_default()
19 image = Image.new('RGB',(disp.width,disp.height),'black').convert('1')
20 draw = ImageDraw.Draw(image)
21 x = 30; y = 30;
22 draw.text((x,y), 'Hello , Pi!!', font=font, fill=255)
23 disp.image(image)
24 disp.display()
25 time.sleep(1)
26 disp.clear()
27 disp.display() # 初始化屏幕相关参数及清屏
28
29 address = 0x48
30 A0 = 0x40
31 bus = smbus.SMBus(1)
32 while True:
33     bus.write_byte(address, A0)
34     value = bus.read_byte(address)
35     voltage = value / 256 * 3.3
36     font = ImageFont.load_default()
37     image = Image.new('RGB',(disp.width,disp.height),'black').convert('1')
38     draw = ImageDraw.Draw(image)
39     x = 30; y = 30;
40     draw.text((x,y), 'Voltage: %.4f'%voltage, font=font, fill=255)
41     disp.image(image)
42     disp.display()
43     print(voltage)
```

A.2 da.py

```
1 import smbus
2 import time
3
4 address = 0x48
5 A0 = 0x40
6 bus = smbus.SMBus(1)
7
8 while True:
9     for value in range(256):
10         bus.write_byte_data(address, A0, value)
11         print(value)
12     for value in range(255, -1, -1):
```

```
13     bus.write_byte_data(address, A0, value)
14     print(value)
```

A.3 adda.py

```
1  import smbus
2  import time
3  from threading import Thread
4
5  import spidev as SPI
6  import SSD1306
7  from PIL import Image # 调用相关库文件
8  from PIL import ImageDraw
9  from PIL import ImageFont
10 RST = 19
11 DC = 16
12 bus = 0
13 device = 0 # 树莓派管脚配置
14
15 disp = SSD1306.SSD1306(rst=RST,dc=DC,spi=SPI.SpiDev(bus,device))
16 disp.begin()
17 disp.clear()
18 disp.display() # 初始化屏幕相关参数及清屏
19 font = ImageFont.load_default()
20 image = Image.new('RGB',(disp.width,disp.height),'black').convert('1')
21 draw = ImageDraw.Draw(image)
22 x = 30; y = 30;
23 draw.text((x,y), 'Hello, Pi!!', font=font, fill=255)
24 disp.image(image)
25 disp.display()
26 time.sleep(1)
27 disp.clear()
28 disp.display() # 初始化屏幕相关参数及清屏
29
30 address = 0x48
31 A0 = 0x40
32 bus = smbus.SMBus(1)
33 value = 0
34
35 def ad_read():
36     global value
```

```

37     while True:
38         bus.write_byte(address, A0)
39         value = bus.read_byte(address)
40         voltage = value / 256 * 3.3
41         font = ImageFont.load_default()
42         image = Image.new('RGB', (disp.width, disp.height), 'black').convert('1')
43         draw = ImageDraw.Draw(image)
44         x = 30; y = 30;
45         draw.text((x,y), 'Voltage: %.4f'%voltage, font=font, fill=255)
46         disp.image(image)
47         disp.display()
48
49 mythread = Thread(target = ad_read)
50 mythread.setDaemon(True)
51
52 try:
53     mythread.start()
54     while True:
55         for i in range(256):
56             bus.write_byte_data(address, A0, i)
57             if value == 0:
58                 value = 0.1
59                 time.sleep(1/value)
60         for i in range(255, -1, -1):
61             bus.write_byte_data(address, A0, i)
62             if value == 0:
63                 value = 0.1
64                 time.sleep(1/value)
65 except KeyboardInterrupt:
66     pass

```

A.4 nnslearn.py

```

1 import torch
2 import torch.nn as nn
3 import torchvision
4 import torchvision.transforms as transforms
5 from sklearn import datasets
6 # 设备配置，部署GPU加速
7 device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
8 # Hyper-parameters, 定义各层规模及其他超参数

```

```
9 input_size = 64
10 hidden_size = 500
11 num_classes = 10
12 num_epochs = 5
13 batch_size = 100
14 learning_rate = 0.001
15 # sklearn dataset, 准备数据集, 转换为 Tensor 数据格式
16 digits = datasets.load_digits()
17 images_and_labels = list(zip(torch.from_numpy(digits.images/16).float(), torch.from_n
18 n_samples = len(digits.images)
19
20 train_loader = images_and_labels[:n_samples//2]
21 test_loader = images_and_labels[n_samples//2:]
22 # 定义网络结构, Fully connected neural network with one hidden layer
23 class NeuralNet(nn.Module):
24     def __init__(self, input_size, hidden_size, num_classes):
25         super(NeuralNet, self).__init__()
26         self.fc1 = nn.Linear(input_size, hidden_size)
27         self.relu = nn.ReLU()
28         self.fc2 = nn.Linear(hidden_size, num_classes)
29     def forward(self, x):
30         # 向前传播, 模型的计算流程
31         out = self.fc1(x)
32         out = self.relu(out)
33         out = self.fc2(out)
34         return out
35
36 model = NeuralNet(input_size, hidden_size, num_classes)
37 # Loss and optimizer
38 # 定义损失函数, 使用的是交叉熵函数
39 criterion = nn.CrossEntropyLoss()
40 # 定义迭代优化算法, 使用的是Adam (Adaptive Moment Estimation)
41 optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
42 # Train the model 迭代训练
43 def train_model(model, train_loader):
44     total_step = len(train_loader)
45     for epoch in range(num_epochs):
46         for i, (images, labels) in enumerate(train_loader):
47             # Move tensors to the configured device
48             images = images.reshape(-1, 8 * 8).to(device)
49             labels = labels.reshape(1).to(device)
```

```
50
51     # Forward pass, 前向传播计算网络结构的输出结果
52     outputs = model(images)
53     # 计算损失函数
54     loss = criterion(outputs, labels)
55     # Backward and optimize 反向传播更新参数
56     # 将上次迭代计算的梯度值清0
57     optimizer.zero_grad()
58     # 反向传播, 计算梯度值
59     loss.backward()
60     # 更新权值参数
61     optimizer.step()
62     # 打印训练信息
63     if (i + 1) % 100 == 0:
64         print('Epoch [{}/{}], Step [{}/{}], loss: {:.4f}'.format(epoch + 1, n
65 # 保存训练好的模型
66     torch.save(model.state_dict(), 'model_skl.ckpt')
67
68 # Test the model 测试模型
69 def test_model(model, test_loader, device):
70     model.load_state_dict(torch.load('model_skl.ckpt', map_location=device))
71     with torch.no_grad():
72         correct = 0
73         total = 0
74         for images, labels in test_loader:
75             images = images.reshape(-1, 8 * 8).to(device)
76             labels = labels.reshape(1).to(device)
77             outputs = model(images)
78             _, predicted = torch.max(outputs.data, 1)
79             total += labels.size(0)
80             correct += (predicted == labels).sum().item()
81         print('Accuracy is: {}'.format(100 * correct / total))
82
83 train_model(model, train_loader)
84 test_model(model, test_loader, device)
```