逻辑亦数据

逻辑门

这一章将视角从拟合上短暂地移开，我相信理解逻辑和数据的关系多少也会帮助我们理解神经网络。读者或许好奇过，计算是如何完成的呢？在讨论这个问题之前，先来做一个约定，我们将 0 视作 False（假），1 视作 True（真）[1]。先来看看几种最简单的逻辑运算。

1. Not（非）（数学写法：$\neg x$，C 语言写法：`!`，Python 写法：`not`）

非是一元运算符，它只有一个输入，输出与输入相反，其中

$$\neg 0 = 1, \neg 1 = 0$$

也就是说 $\neg x = 1 - x$，x 与 $\neg x$ 是互补的。如果你看逻辑 0, 1 仍然感觉不太自然，你可以把它想成 False = not True, True = not False。

2. And（与）（数学写法：$\wedge$，C 语言写法：`&&`，Python 写法：`and`）

与是二元运算符，它有两个输入，仅当两输入都为 1 时输出为 1，否则输出为 0，从真值表[2]就可以看出这一点：

x	y	$x \wedge y$
0	0	0
0	1	0
1	0	0
1	1	1

这与乘法的结果是一样的，所以有时也会省去和的符号，使用 xy 表示 $x \wedge y$。

3. Or（或）（数学写法：$\vee$，C 语言写法：`||`，Python 写法：`or`）

或是二元运算符，它有两个输入，仅当两输入都为 0 时输出为 0，否则输出为 1，真值表如下：

x	y	$x \vee y$
0	0	0
0	1	1
1	0	1
1	1	1

在图上这些运算一般会这样表示：

Figure 1: 逻辑门：从左到右分别为非门、与门、或门

[1] 逻辑 0/1：在物理上，逻辑 0 由 Low（低电平）表示，逻辑 1 由 High（高电平）表示，TTL 和 CMOS 电路各有多种电压标准，感兴趣的读者可以自行学习电路的知识。

[2] 真值表：逻辑运算的输出与输入的关系表。

看起来这只是一些非常简单的运算，但是有了这些就可以构建出所有的计算[3]。例如异或Exclusive Or 运算表示两个输入不同。最粗暴简单的定义方法是列出其输出为 1 的所有情况：$x \text{ xor } y = (x \wedge \neg y) \vee (\neg x \wedge y)$。这样就可以用非、与、或门来构建出一个异或门。

虽然它可以完成"所有的运算"，但是具体来说，比如有读者可能要问，如果我想计算加法，它应该怎么办呢？既然逻辑上只有两个值，那么自然地计算机就要使用二进制来表示数字了。二进制的加法非常简单，就以 $5+3$ 为例，我们可以这样计算：

TODO!

逻辑门又是如何完成这一过程的呢？我们将它拆解成一个个小问题。当加到某一位时，我们需要考虑三个数：两个加数和来自后方的进位。例如下面这种情况：

TODO!

考虑这一位时，后面相加得到结果的情况我们已经不关心了（因此标为浅灰色），在这里只需关心从后方是否有进位（按照列竖式加法习惯，图中蓝色标注的下标 1）。再考虑两个加数的这一位分别为 1 和 0，所以 $1+0+1=2$，在结果栏写下一个 0（横线下方红色的 0），向前进位 1（写在前面一位下标的红色的 1），然后以同样的流程处理前一位。

记两个加数的这一位分别为 x, y，后方进位为 c，那么这一位的加和 s 和向前进位 c_n 可以表示为：

$$s = x \oplus y \oplus c$$

$$c_n = (x \times y) + (c \times (x \oplus y))$$

当然这并不是唯一正确的写法，实际上有很多正确的写法，证明就免了，如果读者有兴趣可以自行尝试，或许也可以找到另外的表达式。最简单粗暴的方法就是把两个加数与是否带进位的情况全部列出来，分成 $2^3 = 8$ 种情况，就得到了如下的表，并逐一验证：

x	y	c	“sum”	c_n	s
0	0	0	0	0	0
0	0	1	1	0	1
0	1	0	1	0	1
0	1	1	2	1	0
1	0	0	1	0	1
1	0	1	2	1	0
1	1	0	2	1	0
1	1	1	3	1	1

就像等式描述的一样，每一个输出 Y 的位都可以通过输入的逻辑运算用一定的电路连接表示，把多个电路串起来[4]，就可以完成加法了。本质上我们的计算机 CPU 就是由这样的门电路与接线组成的[5]。一个 CPU 需要大量门电路组合形成，现代的 CPU 包含数十亿个门电路，而一个门又由若干个微型的晶体管构成。为了让电路精确地实现我们预期的功能，需要精准地将电

图灵完备性

[3]所有的运算：这里指的是 Turing Completeness，如果你想深入了解，可以在 Steam 上购买一个叫做 Turing Complete 的硬核游戏，推荐游玩。

[4]串起来：对于加法这个例子，在网上搜索全加器就可以很容易地搜到。

[5]说明：实际上制造中，与非门、或非门使用更多，因为它们有更方便制造、体积较小、功耗低等优势。

路雕刻在硅片上，这就是光刻技术如此重要的原因。但是山在那，总有人会去登的[6]，两个多世纪的技术积累才造就了现代计算机的诞生，从逻辑门到通用计算机每一步的发展都凝聚着人类技术与智慧的结晶。

推荐阅读

如果你是 Minecraft 玩家或许见过使用红石电路制作的计算机，背后的原理可阅读：

> 计算器计算出「*1+1=2*」的整个计算过程是怎样的？为什么能秒算？ - *WonderL* 的回答 - 知乎
>
> https://www.zhihu.com/question/29432827/answer/150408732

如果你有一些数字电路的基础，并想了解逻辑门是如何组合的，可以阅读：

> 计算器计算出「*1+1=2*」的整个计算过程是怎样的？为什么能秒算？ - *Pulsar* 的回答 - 知乎
>
> https://www.zhihu.com/question/29432827/answer/150337041

[6]山在那，总有人会去登：语出源自英国登山家 George Mallory 当被问及为何要攀登珠穆朗玛峰时的回答"因为山在那里"。刘慈欣的短篇小说《山》引用了这句话。写到大量的微晶体管以精妙地排布构成电路让我想起小说中从基本电路开始进化的的硅基生物。如果你看到这里看累了，去看看小说放松一下吧。

程序是怎么执行起来的

擅长编程的读者或许对编程-编译-执行的路径再熟悉不过了，可少有人思考其中细节。理解程序是如何运行起来的其实是一个基础性的问题，但如果深究下去，这里的水很深：仅是从代码编写到程序运行的过程这一个问题，就足以写好几本书[7]

了。因此我仅仅会从一个极简的视角来介绍 CPU 运行程序的流程，顺带解释必要的概念。让计算机执行程序前，我们首先需要思考"我们想让计算机做什么"并能把它讲明白。开发的第一步永远是明确需求，而后才是写代码让计算机执行，这一点贯彻到后续的机器学习也是一样的。

CPU 不是人类，它并不天然地理解我们的语言，不过或许并不应就这一点给我们带来的不便而感到沮丧：因为从人类手动完成一切计算到计算机的出现，电子器件的计算能力已经将人类从许多重复、繁琐的工作中解放出来。CPU 现在不能干的很多，但此刻更应该思考的是，它能干什么呢？这里我顺着这份 CSAPP 视频合集的思路简单介绍一下。

现代的 CPU 通常包含复杂的 Architecture（架构）与 Instruction Set（指令集），但是为了便于理解，我们先只考虑一个极度简化的 CPU，它就像是在一张"草稿纸"[8]上遵照着一份"指南"[9]运算。能干的事情也就是下面这几个指令（这里与主要的几种汇编语法都略有区别）：

```
mov a, b  ; 将 b 的值赋给 a
add a, b  ; 将 a 和 b 相加，结果存入 a
sub a, b  ; 将 a 减去 b，结果存入 a
mul a, b  ; 将 a 乘以 b，结果存入 a
div a, b  ; 将 a 除以 b，保留整数部分，结果存入 a
jmp addr  ; 跳转到 addr 执行
je addr   ; 如果上一次运算结果为 0，则跳转到 addr 执行
jne addr  ; 如果上一次运算结果不为 0，则跳转到 addr 执行
jl addr   ; 如果上一次运算结果小于 0，则跳转到 addr 执行
cmp a, b  ; 比较 a 和 b 的值，设置标志位
```

先解释一下这些指令名称的含义：

- `mov`：move 的缩写，将一个数值从一个地方移动到另一个地方。
- `add`, `sub`, `mul`, `div`：add, subtract, multiply, divide 的缩写，加减乘除。
- `jmp`, `je`, `jne`, `jl`：jump, jump if equal, jump if not equal, jump if less 的缩写，分别为跳转、当等于时跳转、当不等于时跳转、当小于时跳转。
- `cmp`：compare 的缩写，比较。

这里写作 `a`，`b` 的其实都表示内存上的一个地址，类似于如果给行编号，那么 `a`，`b` 就是行号。再引入一个额外的符号，`[a]` 表示取地址 a 上的值，例如当内存单元 42 中存着值 64 时，`[42]` 就表示 64，例如 `mov 10, [42]` 表示的就是把 64 号内存的值赋给 10 号内存。`#x` 表示 Immediate Value（立即数值） x，例如 `#10` 表示数值 10 本身，而非内存位置 10。那么我们可以写出一个简单的程序，例如把内存 0 位置[10]的值与内存 1 位置的加和存入内存 2：

---

[7] 好几本书：比如几本经典教材

- 程序如何编译出来：《编译原理》Compilers: Principles, Techniques, and Tools
- 计算机的结构：《深入理解计算机系统》Computer Systems: A Programmer's Perspective
- 程序的结构：《计算机程序的构造和解释》Structure and Interpretation of Computer Programs

[8] 草稿纸：比喻计算机的 RAM（内存），暂且把它理解为每格写了一个整数，实际计算机中是字节。

[9] 指南：比喻计算机的程序，是计算机要执行的 Instruction（指令）。

[10] 内存 0：按照计算机中的习惯，计数从 0 开始。

```
mov 2, 0    ; 将 0 号内存的值赋给 2 号内存
add 2, 1    ; 将 2 号内存和 1 号内存相加，结果存入 2 号内存
```

又比如，如果我们想交换内存 0 和内存 1 位置的数值，可以这样写：

```
mov 2, 0    ; 将 0 号内存的值赋给 2 号内存
mov 0, 1    ; 将 1 号内存的值赋给 0 号内存
mov 1, 2    ; 将 2 号内存的值赋给 1 号内存
```

这个过程运行时[11]看起来是这样的：右边的列表表示内存，每个元素是内存的一个单元，这里 x_i 示意第 i 个内存单元。x, y 都是数，你可以把它带入 $1, 2$ 或者你想的任何数字，右侧的列表则表示对应的指令执行后的内存状态：

TODO!

不过看到这里，不知读者是否发现了一个问题：内存中的 2 号位置在交换 0 号和 1 号位置的数值时被覆盖了。这种情况一般称为 Side Effect（副作用）[12]，但似乎不太可能既不修改其它内存，又交换数值[13]。万一内存 2 储存了重要的数据，丢失了是很大的问题。那么怎么办呢？干脆设定某块区域可以随意用作临时存储[14]，我们就此"发明"了 Register（寄存器）[15]。就假设我们接下来约定了地址 0-7 是寄存器，可以存储临时的数据。为了方便阅读，接下来把它们标记为 r0 到 r7。既然这样，0-7 的位置就可以用作临时存储了，但是同时它们也不适合作为输入输出[16]。所以这次我们把任务改为交换内存 8 和内存 9：

```
mov r0, 8   ; 将 8 号内存的值赋给 0 号寄存器
mov 8, 9    ; 将 9 号内存的值赋给 8 号内存
mov 9, r0   ; 将 0 号寄存器的值赋给 9 号内存
```

这样程序运行的过程中改变的就仅仅是我们视作数据内容 Volatile（易失）的寄存器，而内存中的数据则保持不变。这样我们再来写一个简单的求和程序，在内存 8 中存储了求和的起点地址，内存 9 中存储了求和的终点地址，为了方便起见，我们使用左闭右开区间，即包含起点，但不包含终点（一会就会看到它带来的方便）。最后将求和结果存入内存 10：

```
mov r0, #0  ; 将 0 写入 0 号寄存器
mov r1, 8   ; 将 8 号内存的值赋给 1 号寄存器
mov r2, 9   ; 将 9 号内存的值赋给 2 号寄存器
loop:
    add r0, [r1]    ; 将 1 号寄存器指向的内存的值加到 0 号寄存器
    add r1, #1      ; 1 号储存器指向的内存地址加 1
    cmp r1, r2      ; 比较 1 号寄存器和 2 号寄存器的值
    jne loop        ; 如果不相等，跳转到 loop
mov 10, r0  ; 将 0 号寄存器的值存入 10 号内存
```

严格来讲上面这段代码包含了前文还没引入标签的概念，其中的 `loop:` 就是一个标签，它是一个位置的别名[17]，也是填写在 jmp, je, jne 指令后的地址。

---

[11] 你先别管它怎么运行起来的。

[12] 副作用：指令运行的过程中对其他地方产生的影响。

[13] 不太可能：在本例中确实有奇技淫巧可以在不设中间变量的情况下交换变量，只是它使用到了一些代数性质，既不方便，可读性和可拓展性也差。

[14] 临时储存：可以理解为一种草稿纸，内容可以随时丢弃

[15] 寄存器：实际的 CPU 中，寄存器是 CPU 内部的一块存储区域，与内存的处理、读写速度等都有显著的不同。但是出于易于理解起见，我们这里仍把它当作一个特殊的内存区域。

[16] 不适合：这里指的是不方便我们的讨论，实际程序中是靠一定的约定依靠寄存器传递参数的，但是这些规则可能会为清晰的说明带来困扰，所以在这里寄存器还是用作纯粹的草稿。

[17] 别名：例如在本例中，它指代 `add r0, [r1]` 所在的行

这个程序运行起来是怎么样的呢？假设我们在 8 号位置存储了起点地址 15，9 号位置存储了终点地址 18（它们虽然储存的是地址，从程序逻辑上指向的是内存块，但是本质上在 CPU 看来仍然是一种"整数"，只是这个整数记录了另一个整数的位置信息）。那么程序运行的过程大概是这样的（这里假设内存中 x_{15}, x_{16}, x_{17} 分别存储了 1，2，3）：

指令	$[r_0, r_1, r_2, ...]$	$x_8, x_9, x_{10}, ...$	$x_{15}, x_{16}, x_{17}, ...$
(initial)	[?,?,?, ...]	$15, 18, ?, ...$	$1, 2, 3, ...$
→ `mov r0, #0`	[0,?,?, ...]	$15, 18, ?, ...$	$1, 2, 3, ...$
→ `mov r1, 8`	[0, 15,?, ...]	$15, 18, ?, ...$	$1, 2, 3, ...$
→ `mov r2, 9`	$[0, 15, 18, ...]$	$15, 18, ?, ...$	$1, 2, 3, ...$
→ `add r0, [r1]`	$[1, 15, 18, ...]$	$15, 18, ?, ...$	$1, 2, 3, ...$
→ `add r1, #1`	$[1, 16, 18, ...]$	$15, 18, ?, ...$	$1, 2, 3, ...$
→ `cmp r1, r2`	$[1, 16, 18, ...]$	$15, 18, ?, ...$	$1, 2, 3, ...$
→ `add r0, [r1]`	$[3, 16, 18, ...]$	$15, 18, ?, ...$	$1, 2, 3, ...$
→ `add r1, #1`	$[3, 17, 18, ...]$	$15, 18, ?, ...$	$1, 2, 3, ...$
→ `cmp r1, r2`	$[3, 17, 18, ...]$	$15, 18, ?, ...$	$1, 2, 3, ...$
→ `add r0, [r1]`	$[6, 17, 18, ...]$	$15, 18, ?, ...$	$1, 2, 3, ...$
→ `add r1, #1`	$[6, 18, 18, ...]$	$15, 18, ?, ...$	$1, 2, 3, ...$
→ `cmp r1, r2`	$[6, 18, 18, ...]$	$15, 18, ?, ...$	$1, 2, 3, ...$
→ `mov 10, r0`	$[6, 18, 18, ...]$	$15, 18, 6, ...$	$1, 2, 3, ...$

这个程序的执行过程是这样的：

1. 初始化：将 r_0 设为 0，r_1 设为起点地址（15），r_2 设为终点地址（18）。
2. 循环：
   - 将 r_1 指向的内存单元的值加到 r_0 中
   - r_1 加 1，指向下一个内存单元
   - 比较 r_1 和 r_2 的值，如果不相等就继续循环
3. 最后将 r_0 中的结果（6）存入内存 10 号位置。

这个程序就完成了从内存 15 号位置到 18 号位置（不含）的所有数的求和。这里我们可以看到左闭右开区间的好处：每次循环开始时，r_1 指向的是当前要处理的位置，而 r_2 指向的是要处理的最后一个位置的下一个位置。这样当 r_1 等于 r_2 时，就意味着所有的数都已经处理完了。

推荐阅读

如果你想了解更多关于计算机如何执行程序的细节，推荐阅读：

CSAPP 视频合集

https://www.bilibili.com/video/BV1Lp4y167im

数据的表示

在上一节中，我们看到了计算机是如何执行程序的。但是我们还没有讨论数据是如何表示的。在计算机中，所有的数据都是以二进制的形式存储的。这是因为计算机的硬件是由电路组成的，而电路只能表示两种状态：高电平和低电平。因此，计算机中的数据都是由 0 和 1 组成的。

整数的表示

整数是最简单的数据类型。在计算机中，整数通常使用 Two's Complement（二进制补码）表示。例如，在一个 8 位的系统中，数字的表示范围是 -128 到 127。这是因为：

- 最高位是符号位，0 表示正数，1 表示负数
- 对于正数，直接用二进制表示
- 对于负数，先将其绝对值用二进制表示，然后按位取反，最后加 1

例如，在 8 位系统中：
- 5：00000101
- -5：11111011 (5 的二进制是 00000101，取反得 11111010，加 1 得 11111011)

这种表示方法有几个优点：
1. 0 只有一种表示方式：00000000
2. 加法和减法可以统一处理
3. 比较大小时可以直接使用二进制比较

浮点数的表示

浮点数的表示比整数要复杂得多。在计算机中，浮点数通常使用 IEEE 754（IEEE 754 标准）表示。这个标准定义了两种主要的格式：
- 单精度（32 位）
- 双精度（64 位）

以单精度为例，其结构为：
- 1 位符号位
- 8 位指数位
- 23 位尾数位

一个数 N 可以表示为：$N = (-1)^s \times 2^{e-127} \times (1 + f)$ 其中：
- s 是符号位
- e 是指数位的值
- f 是尾数位表示的小数

这种表示方法可以表示很大范围的数，但是也有一些特殊情况需要处理：
- 非规范化数
- 无穷大
- NaN（Not a Number）

字符的表示

字符在计算机中也是用二进制表示的。最常用的字符编码标准是 ASCII 和 Unicode。ASCII 使用 7 位二进制表示 128 个字符，包括：
- 英文字母（大小写）

- 数字
- 标点符号
- 控制字符

而 Unicode 则可以表示更多的字符，包括：
- 各国文字
- 符号
- 表情符号等

在实际使用中，Unicode 通常使用 UTF-8、UTF-16 或 UTF-32 等编码方式存储。其中 UTF-8 是最常用的，它是一种变长编码，可以节省存储空间。

推荐阅读

如果你想深入了解计算机中的数据表示：

IEEE 754 浮点数标准详解

https://www.ruanyifeng.com/blog/2010/06/ieee_floating-point_representation.html

如果你想了解更多关于字符编码：

字符编码必知必会

https://www.joelonsoftware.com/2003/10/08/the-absolute-minimum-every-software-developer-absolutely-positively-must-know-about-unicode-and-character-sets-no-excuses/