如何理解神经网络——信息量、压缩与智能

目录

1 从函数拟合开始

1.1 最简单的规律——简单线性回归

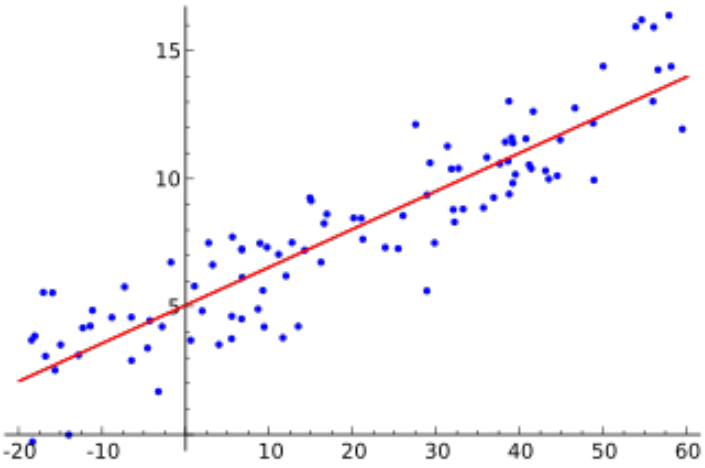

图 1　线性回归示意图

图源：Wikipedia

虽然 线性回归 Linear Regression 的名字叫做" 回归 Regression "，但是事实上我更喜欢叫做 线性拟合 Linear Fitting 。它的目的是找到一条直线尽可能"贴近"数据点。在这一基础上，我们可以发现数据之间的规律，从而做出一些预测。不过这里有几个问题：

- 为什么要用直线？为什么不用曲线？
- 为什么要用直线拟合数据点？这有什么用？
- “贴近"数据点的标准是什么？为什么要选择这个标准？

我认为用直线的原因无非两点：一是直线 $y = kx + b$ 简单且意义明确，又能处理不少的问题。几何上直线作为基本对象，尺子就能画出；代数上只需要加减乘除，一次函数我们也很早就学过了。而它的思想一路贯穿到了微积分的导数并延申到了线性代数。二是许多曲线的回归可以转为线性回归（见后文）。

例如指数型的 $y = k\mathrm{e}^{\alpha x}$ 取对数变为 $z = \alpha x + \ln k$，又如分式型的 $y = (\alpha x + \beta)^{-1}$ 取倒数转化为 $z = \alpha x + \beta$，从而归结为线性拟合。因此带着线性拟合经验再去考虑曲线会更轻松。

至于其意义：一是找到数据的规律，二是做出预测。拟合的系数可以用于测算数据之间的关系，斜率 k 表明输出对输入的敏感程度。一个经典例子是广告投放的 边际效益 Marginal Benefit [1]，在一定范围内拟合收益与投入的关系，可以估算当前的边际效益，从而决定是否继续投放。而物理上，比值定义法定义的各种物理量，如电阻、电容等，最常用的测算方式都是线性拟合。例如测量电源输出的若干组电压和电流数据，并拟合出直线，斜率的绝

[1]边际效益：经济学概念，每增加单位投入，产出会增加多少单位

对值是电源的内阻，同时截距顺带给出了电源的电动势，这样测得的数据就可以用于预测电源的输出情况。对我们所处的世界有定量的认识是科学的基础。可测量的数据和数学模型来描述、解释和预测自然现象是科学的基本方法，也是拟合的根本目的。

推荐阅读

Least Squares

如果你想了解"回归"与" 最小二乘 “的含义： 用人话讲明白线性回归 Linear Regression - 化简可得的文章 - 知乎

如果你想阅读从求导法到线性代数方法的详尽公式推理： 非常详细的线性回归原理讲解 - 小白 Horace 的文章 - 知乎

如果你想详细了解了线性回归中的术语、求解过程与几何诠释： 机器学习| 算法笔记-线性回归（Linear Regression） - iamwhatiwant 的文章 - 知乎

1.2 多项式拟合

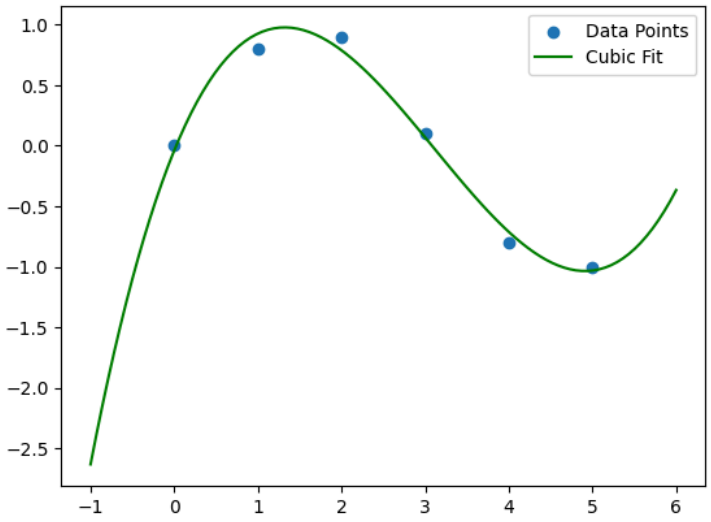

图 2　多项式拟合示意图（图为 3 次拟合）

图源：GeeksforGeeks

线性拟合虽然很好，但是如果拿到了明显不线性的一堆数据，那么线性拟合就显得有些力不从心了。不过既然都是拟合，能做一次的那按理来讲也能做多次。多项式拟合（Polynomial Fitting）就是这样一种思路，只是预测 $\hat{y}$ 从 $kx+b$ 变成了 $a_0+a_1x+...+a_mx^m$[2]，其中 m 是多项式的次数。而均方误差的表达式甚至几乎不用变，仍然是

$$\mathrm{MSE}=\frac{1}{n}\sum_{i=1}^{n}(y_i-\hat{y})^2$$

只不过展开后是一系列的多项式项，待拟合的参数从两个变成了 $m+1$ 个。但是如果观察一下，这个式子仍然是一个（多变量的）二次函数，所以最小化的方法也是一样的。多项式自有多项式的好，能加的项多了，拟合的灵活性也就大了，误差显然会更小。然而与线性拟合相比，它虽然有解析解（Analytical Solution），但不再像线性拟合一样可以逐项明确说出意义，而是只剩下一堆矩阵运算把这些参数算出来。因此相比于记下公式，形成一个整体上的印象显得尤为重要。

上一小节中，我们从图像看到了这种拟合的几何解释，而多项式拟合也是相似的，还是从 $\boldsymbol{r}$ 的表达式入手

$$\boldsymbol{r}=\boldsymbol{y}-\left(a_0\boldsymbol{x}^0+a_1\boldsymbol{x}^1+...+a_m\boldsymbol{x}^m\right)$$

对比之前的表达式，当 $a_0,a_1,...,a_m$ 变化时，预测得到的结果 $\hat{\boldsymbol{y}}=a_0\boldsymbol{x}^0+a_1\boldsymbol{x}^1+...+a_m\boldsymbol{x}^m$ 也会在一个 $m+1$ 维的空间中变化，正如之前的平面，这个空间也是一个 $m+1$

[2]记号说明：虽然习惯上幂次从大到小排列，但是为了下标和幂次的统一性，所以这里选择从常数项到最高次项排列

维的子空间。求最小模的 $\boldsymbol{r}$ 又回到了从点到子空间的垂线问题。虽然不得不承认：想象从一个高维的 n 维空间中向 $m+1$ 维的子空间做垂线确实有些困难，但是这多少离我们的几何直觉更近了一些。

系数的意义不那么明确了，但是误差下来了，这是好事吗？也不一定，灵活性的另一面是潜在的 过拟合 Overfitting 。前文中做线性拟合的时候有一个重要的假设是测量得到数据带有一定的误差。拟合的直线滤去了大部分的误差，留下了重要的趋势。但是如果灵活性太高，拟合的多项式会过于贴合数据，甚至把误差也拟合进去了。即使在给定的数据上做到了很小的误差，预测新数据的能力却可能会大打折扣。

拿做题打个比方：使用直线拟合明显不线性的数据是方法错了，只能说是没完全学会。但是用接近数据量的参数来拟合数据，留给它的空间都够把结果"背下来"了，捕捉到了数据的细节，却忽略了数据背后的规律，化成了一种只知道背答案的自我感动。在几道例题上能做到滴水不漏，但是一遇到新题就束手无策。

举个例子，在下面这个数据集上试图拟合，我们在二次函数 $y=0.25x^2-x+1$ 上添加了标准正态分布的噪声，即实际上 $y=0.25x^2-x+1+\mathcal{N}(0,1)$[3]。

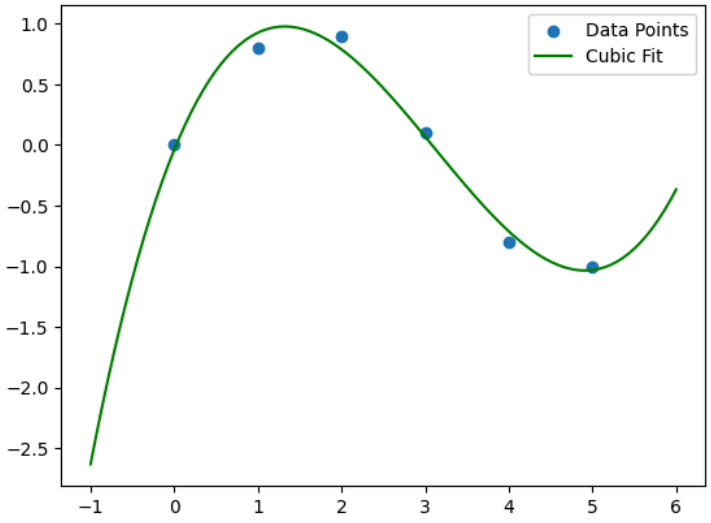

图 3　多项式拟合示意图（图为 3 次拟合）

图源：GeeksforGeeks

那么现在我们来试试用不同次数的多项式拟合这个数据集。不难看出线性拟合的线与数据点还是相差不少，因为它没能提供可以制造数据"弯曲"形状的项，它没能捕捉到数据更加复杂的趋势，这种现象称为 欠拟合 Underfitting 。2 次曲线的效果几乎和真实曲线一样，即使提升到 3 次也没有太明显的改变，它们拟合的效果都还算好。

[3] $\mathcal{N}(0,1)$：表示一个服从标准正态分布的变量，均值为 0，方差为 1

2 逻辑亦数据

2.1 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）[4]。先来看看几种最简单的逻辑运算。

1. Not（非）（数学写法：$\overline{x}$，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

   $$\overline{0} = 1, \overline{1} = 0$$

   也就是说 $\overline{x} = 1 - x$，x 与 $\overline{x}$ 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[5]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：

---

[4]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。

[5]真值表：逻辑运算的输出与输入的关系表。

图 4　逻辑门：从左到右分别为非门、与门、或门

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[6]。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：$x \text{ xor } y = (x \wedge \overline{y}) \vee (\overline{x} \wedge y)$。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

[6]所有的运算：这里指的是 Turing Completeness，图灵完备性，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。

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[7]，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[8]。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[9]，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

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

[7]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。

[8]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。

[9]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。

2.2 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[10]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"[11]上遵照着一份"指南"[12]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。

---

[10]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

[11]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。

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

- `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（立即数值） Value 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[13]的值与内存 `1` 位置的加和存入内存 `2`：

```
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[14]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）[15]，但似乎不太可能既不修改其它内存，又交换数值[16]。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[17]，我们就此"发明"了 Register（寄存器）[18]。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[19]。所以这次我们把任务改为交换内存 `8` 和内存 `9`：

---

[13] 内存 `0`：按照计算机中的习惯，计数从 0 开始。

[14] 你先别管它怎么运行起来的。

[15] 副作用：指令运行的过程中对其他地方产生的影响。

[16] 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。

[17] 临时储存：可以理解为一种草稿纸，内容可以随时丢弃

[18] 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。

[19] 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。

```
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[20]，也是填写在 `jmp`, `je`, `jne` 指令后的地址。

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, ...

[20]别名：例如在本例中，它指代 `add r0, [r1]` 所在的行

→ `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

2.3 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。

2.3.1 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

2.3.2 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）

2.3.3 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

如果你想了解更多关于字符编码：字符编码必知必会

3 为什么是神经网络

3.1 神经网络：一个大的函数

神经网络

相比于 Neural Network 如何实现其功能，读者或许更想问的是：为什么要用神经网络？现有的神经网络为什么用了这些方法？对于这一类问题，一个现实的回答是：机器学习是高度以实用为导向的，实验显示这样做效果更好。在现实中，我们往往要解决各种各样的问题，人类开发者以手写每一行代码创造了各种各样的程序，自动化地解决了许多问题。但很多问题难以在有限的时间内找到确定性的解决方案，例如识别图片中的物体、识别语音、自然语言处理等等。它们有一个共同点：输入的信息量巨大、关系复杂，难以用确定的规则来描述。手动规定像素范围来判断物体类型，或用固定的规则来解析自

然语言显然并不现实。因此人们自然要问有没有更加自动化、灵活、智能的方法来一劳永逸地解决这些问题。人工智能的概念就此提出，人们希望让机器自己学习知识来解决问题。

虽然目前人类仍然很难说摸到了 Artificial General Intelligence[21]（通用人工智能）的边界，但人工智能已然在许多问题上取得了巨大成就，走出了 20 世纪末 21 世纪初被大众认为是"伪科学"的寒冬。经过深度残差网络在图像识别的重大突破、AlphaGo 学会下围棋、Transformer 在翻译比赛取得优异成绩并引来一波生成式模型的热潮等等，人工智能就这样走向了时代的焦点。但是如果要问：为什么它这么成功？最直接的回答仍是：It works.

除了一些基础的训练方法外，其它的结构构成、参数调整等等往往都是人们有一个想法，于是就这样展开了实验。部分实验成功了，就说明这个想法是对的，从而延伸出新的调节思路。如此循环往复，形成了现在的人工智能领域。因此就模型结构而言并没有非常完备的理论，有的只能说是经验法则。

不过我想可以对解决的方法做一个简单的分类。按照参数的数量，从参数复杂到参数简单可以画出一条轴。按照模型获取经验的方式，从模型完全编码了先验经验，到通过一些例子得到经验，再到持续在与环境的互动中获取经验，可以画出另一条轴。在这里我也试图并不严谨地画出了这样一个表格。

监督方式 参数量	超大参数量	大参数量	小参数量	经典模型
持续互动	PPO, A3C	DQN	Q-Learning	经典控制
输入/输出对	ResNet, Transformer	浅层 CNN	浅层 MLP	SVM
无监督	GAN, SimCLR	———	K-Means, KNN	PCA, t-SNE

读者看到的第一反应大抵是感到看不懂。不过我也并非想让读者先学完再来看这个表格，而是希望读者看到：解决问题的方法虽然多样，但仍可根据若干指标大致分类。表中的术语有的是模型结构，有的是算法，有的是思想，有的是算法，有的是思想，而右侧的一列甚至根本就不是机器学习，对机器学习有基本了解的读者或许会认为它们可比性存疑。诚然，模型之间并没有一个实际上的绝对界限，表中划分的位置也仅是凭借我的经验评价一个模型大多数时候处于什么位置，而非绝对的准则，但我认为这样的划分是有意义的，用一种更为建设性的话来说：意义就是在混乱的世界中建构起规律，用于解决问题。

[21]通用人工智能：指能像人类一样解决各种通用的问题的人工智能。

大参数量的一侧——神经网络的领域，正是本书的主题。作为神经网络的引入，有必要从更高的角度来理解以神经网络为基础的模型目标是什么。小节标题已经足以表达内容核心：先不论内部结构如何，所谓的神经网络，无非也是一个函数。所谓函数，就必然要考虑到输入和输出，或者更准确地说，我们关心的就是怎么用计算机程序对给定的输入，得到我们想要的输出。无论是连续的数据，还是按照 0 或者 1 编码为向量的标签，输入和输出都可以变为向量。因此许多问题都可以归结为一个更加狭义的、数值拟合意义上的函数拟合问题。一个 Encoder（编码器）将原始输入变为向量这种易于处理的形式。而对于函数的原始输出，可以通过一个 Decoder（解码器）将数值构成的向量变为我们想要的输出。

而再向前看，在第一章中我们已经初步了解了以线性回归为代表的一类函数拟合问题。虽然这一问题从结构上相对简单，但是从这一情境中可以抽象出函数拟合的理念：有一些输入和输出的对应关系，我们要设计一个带参数的拟合模型，调整参数，让模型的输出尽可能接近我们预期的输出，接近程度则通过一个损失函数来衡量。

因此我会把模型抽象成五个要素：Input（输入）、Output（输出）、Architecture（模型结构）、Loss Function（损失函数）和 Optimizer（优化算法）。输入、模型架构和具体参数决定了输出如何计算，按照损失函数计算得到的损失指导模型调整具体参数，优化算法则决定了参数如何调整。当然这样的划分只是我自己的理解，而非理解神经网络的唯一方式。这里我不打算在概念之间玩文字游戏，把机器学习中的概念倒来倒去，变成一篇又臭又长，令人看完莫名其妙、不知所云、又对实践毫无益处的文章。因此我认为画一个图串起来是最直观的方式。

TODO: 这里需要一个图，展示了神经网络的五个要素及其关系

从输入到输出再到损失的过程通常称为 Forward Propagation（正向传播），而从损失到参数的更新过程则称为 Backward Propagation（反向传播）。而这中间的模型结构常常由矩阵运算与一些 Activation Function（激活函数）构成的层组成。几乎可以说众多的神经网络中，只有这种传播的方式和网络的基本组成元素是相同的，如何从这些基本元素构建出好的模型则像是搭积木一样，各有各的搭法。

在这里我想简单讲讲使用矩阵运算的原因。在第一章中我们已经简单地学习了矩阵运算的基本知识，它本质上是正比例函数在向量空间中的推广，只是 $y = kx$ 中的斜率变成了一个个从输入 x_j 连接到输出 y_i 的权重 w_{ij}。从行看过去，它反映了输出的每个分量（或称为特征）是如何由输入的每个分量线性组合而成的。而从列看过去，它表明了输入的每个分量是如何影响输出的。就像一次函数有一个常数项一样，矩阵运算也有一个偏

置项 b，运算的总体结构是 $y = wx + b$。从代数上看，它运算简单[22]，而从分析上看，它的输出变化光滑，容易求导[23]。

相关阅读

这篇文章讲述了神经网络的起源：

如何简单形象又有趣地讲解神经网络是什么？ - 佳人李大花的回答 - 知乎

https://www.zhihu.com/question/22553761/answer/3359939138

读者或许会好奇所谓的万能逼近定理需要是如何能逼近给定函数的，这篇回答的解释不错：

神经网络的万能逼近定理已经发展到什么地步了？ - 牛油果博士的回答 - 知乎

https://www.zhihu.com/question/347654789/answer/1534866932

这一问题下有关于神经网络"涌现"出新的现象的讨论，对其机理感兴趣的读者也可以想想背后的原因：

如果神经网络规模足够大，会产生智能吗？ - 知乎

https://www.zhihu.com/question/408690594

[22]简单：仅由简单的四则运算组成，现代 GPU 也常常提供高效的矩阵运算加速。

[23]容易求导：记住这一点，这对后续反向传播等算法的实现至关重要。如果在离散的空间中操作，例如使用阶跃函数或者逻辑门，便无法借助导数来进行参数更新。

3.2 “激活函数与非线性”

将 $y = wx + b$ 作为一次函数的类比应该足以说明它是很简单的一类函数。但是正如一次函数的复合 $y = w_2(w_1x + b_1) + b_2 = w_2w_1x + (w_2b_1 + b_2)$ 仍然是一次函数一样，如果仅仅沉浸在矩阵运算中，我们便永远无法表达那些复杂的函数。举个最简单的例子，我们甚至无法表示输入的绝对值 $y = |x|$。因此我们需要在模型的结构中加点"非线性"，让它不仅仅局限于简单的加减乘除，专业的说法称之为 Activation Function（激活函数）。激活函数直接作用在每个特征上，而且函数本身通常是固定的[24]，且总体通常呈现递增的趋势。

所谓逐元素作用，也就是说，与矩阵对特征进行组合不同，激活函数对各个分量的操作是独立的。其输入是一个向量，输出也是一个同样维数的向量。如果选定了激活函数 $f : \mathbb{R} \to \mathbb{R}$，输入为 $x = [x_1, x_2, \cdots, x_n]$，则输出为 $y = [f(x_1), f(x_2), \cdots, f(x_n)]$。

现在使用最多的激活函数是 Rectified Linear Unit (ReLU)（线性整流函数），虽然相对于其它激活函数，诸如 Sigmoid、tanh 等等，"ReLU" 其实算是晚辈，但是在关于激活函数的讨论中，有研究表明它的效果更好，而后 AlexNet 的成功更让它成为了主流的激活函数。虽然失去了早期其它激活函数的仿生背景，但它好用，而且非常简单。它的定义是：

$$\mathrm{ReLU}(x) = \max\{0, x\} = \begin{cases} x, x \geq 0 \\ 0, x < 0 \end{cases}$$

图像是这样的：

TODO: 这里需要一个图，展示 ReLU 函数的图像

举一个例子就可以看出逐元素作用的含义。例如有输入向量 $x = [1, -2, 3]$，那么它经过 ReLU 激活函数的输出为 $y = [1, 0, 3]$。正的部分被保留了，而负的部分被置为 0。正如电路中的半波 Rectifier（整流器）一样，把负值截断了。

而它的导数也非常简单：

$$\frac{\partial}{dx}\,\mathrm{ReLU}(x) = \begin{cases} 1, x > 0 \\ 0, x < 0 \end{cases}$$

读者或许会关心，那 0 这一点不可导要怎么办？其实关系不大，因为一个小数几乎不可能[25]在训练中恰好落在 0 上。即使有，也可以任意地选择一个值，例如 0 或者 1[26]。有了这样的激活函数，函数的表达能力大大就增强了。以目标 $|x|$ 为例，假设有输入 x，只需两个 ReLU 函数值的和就可以表示它：

---

[24]通常是固定的：在一些模型，例如使用可变样条函数的 KAN 中，激活函数也是可学习的，而且各个元素上的效果可能不同，但是可变的激活函数总体来说并不常见。

[25]几乎不可能：在最常用的 32 位浮点数中，一个数恰好取到 0 的概率大概在 10^{-9} 量级。虽然在 FP8 或者 FP16 量化中恰好取到 0 的概率更大，然而实践中这单个不可导点几乎不会对训练产生影响。

[26]0 处的导数：PyTorch 通常选择 0

$$|x| = \mathrm{ReLU}(x) + \mathrm{ReLU}(-x) = \max\{0, x\} + \max\{0, -x\}$$

初看可能会觉得这样的表达方式有点多此一举，像是为了 $|x|$ 这盘醋专门包的饺子。但是别急，让我们把它拆解成神经网络的结构，更加结构化地看待。

最初的输入是 x，它先经过一个线性的函数得到 $[x, -x]$，再经过 ReLU 函数得到中间的向量 $x^{(1)} = (\max\{0, x\}, \max\{0, -x\})$，而这使用一个线性函数就可以得到 $y = |x|$。

写成矩阵的形式就有

$$w_1 = \begin{pmatrix} 1 \\ -1 \end{pmatrix}, b_1 = \begin{pmatrix} 0 \\ 0 \end{pmatrix}, w_2 = (1 \ \ 1), b_2 = 0$$

遂可以写成 $y = w_2\ \mathrm{ReLU}(w_1 x + b_1) + b_2$。我认为，把这件事作为一个 toy case[27]想明白多少可以帮助理解神经网络。把矩阵的每个权重都画出来就是这样了：

TODO: 这里需要一个图，展示神经网络表示|x|的结构

这看起来很简单，读者可能想问：还能不能再给力一点，看看更复杂的情况呢？当然可以。不过在看之前先抛出两个思考题：

1. 试着用线性函数和 ReLU 函数表示 $y = \max\{x_1, x_2\}$，并画出它的神经网络结构图。
2. 线性函数和 ReLU 的组合不能表示什么函数呢？

在思考这个问题时，读者可以先回顾 ReLU 的性质：它的作用是将负数截断为 0，而正数保持不变。那么，能否通过适当的线性变换和 ReLU 来分辨两个数的大小呢？实际上我们可以很容易地发现

$$\max\{x_1, x_2\} = x_1 + \mathrm{ReLU}(x_2 - x_1)$$

但是这个答案并不够好，如果直接把它画成神经网络结构图，就会发现它的结构看起来像是这样：

TODO: 这里需要一个图，展示神经网络表示 max{x_1, x_2}的一种方法

变量 x_1 没有经过统一的隐藏层，而是跳过中间，直接连接到了输出层。显然就不能用一致的 $\mathrm{ReLU}(wx + b)$ 的形式来表示了，而是要单独开一个通道来处理。而我们使用神经网络的目的本来就是用一致的方式来处理所有的输入，所以这样的表示方式并不优雅[28]。

---

[27]toy case：玩具案例，指的是一个简单的例子，用于说明某个概念或方法。

[28]并不优雅：与之对比，在深层神经网络网络中通常会引入看起来有些像这里的 Shortcut Connection（跳连接）结构，由此引出 Residual Network（残差网络）[29]的概念。它看起来有些像这里的跳过中间层的结构，但那里是系统性地引入这样的连接，而不是这样对某个分量单独处理。

[29]残差网络：是指在深度神经网络中，通过引入跳过中间层的连接，使得网络能够更好地学习到输入和输出之间的残差，从而使得网络能够更深地进行训练。

不过使用一点小小的技巧，可以把 x_1 本身写成 $x_1 = \mathrm{ReLU}(x_1) - \mathrm{ReLU}(-x_1)$，这样一来就可以把它写成带有三个中间变量的一个网络结构了。把

$$\max\{x_1, x_2\} = \mathrm{ReLU}(x_1) - \mathrm{ReLU}(-x_1) + \mathrm{ReLU}(x_2 - x_1)$$

这一式子中的三个分量提出来，便可以得到

$$x_1^{(1)} = \mathrm{ReLU}(x_1 + 0x_2)$$

$$x_2^{(1)} = \mathrm{ReLU}(-x_1 + 0x_2)$$

$$x_3^{(1)} = \mathrm{ReLU}(-x_1 + x_2)$$

$$y = x_1^{(1)} - x_2^{(1)} + x_3^{(1)}$$

偏置 b 仍然为 0，读者可以自行试着写出对应的权重矩阵 w，按照新的写法重新绘制，这时结构图就会变成这样：

TODO: 这里需要一个图，展示神经网络表示 max{x_1, x_2}的另一种方法

虽然中间的神经元多了一些，但是它的结构看起来就统一而且整齐得多了。或许有人会有疑问，这里连的线变多了，不是把事情复杂化了吗？实际上并没有，恰恰相反，把它整齐地写出来才有利于算法的数值优化。

一个有趣的事实是，如果把 True 和 False 分别视作 1 和 0，那么最多两层的网络就可以表示任意的逻辑函数。例如

$$x_1 \text{ and } x_2 = \mathrm{ReLU}(x_1 + x_2 - 1)$$

$$x_1 \text{ or } x_2 = \mathrm{ReLU}(x_1) + \mathrm{ReLU}(x_2 - x_1)$$

$$x_1 \text{ xor } x_2 = \mathrm{ReLU}(x_1 - x_2) + \mathrm{ReLU}(x_2 - x_1)$$

这至少表明逻辑可以在一定程度上编码进神经网络中，用一些可调的权重来模拟逻辑门[30]，因此从这一特例来看，求特征的交集、并集的操作确实可以自然地以权重的方式编码到网络的运算中。

推而广之，不难发现 ReLU 本质上完成的是将函数分段的操作。调整权重就可以做到在不同的区域选择不同的段，从而给出不同的表达式。虽然它在每一根区域内仍然是线性的，但却可以通过一些点上的弯折来实现非线性，表达能力比单纯的线性函数大大提高。这样的函数在数学上称为 Piecewise Linear Function（分段线性函数），如我们所见，ReLU 函数就提供了一种通用的方式来实现分段线性函数，从而将关于"分类"的信息编码到网络中。

[30]用权重模拟逻辑门：这里仅说明它可以，不过这么做太奢侈了，很浪费储存和计算资源。

那么它不能表示什么函数呢？由于其分段线性的特性，不难证明它无法完全精准地表示光滑的曲线，例如 $y = x^2$。而且可以证明，对于任何一个分段线性函数 $f(x)$，都可以找到一个常数 c 使对于 $\|x\|$ 足够大的时候，$f(x) \leq c\|x\|$。从而增长速度有限，无法表示指数函数或者高次的多项式函数。

这确实体现出了它的局限性，但这必然是它的弱点吗？并不一定。一方面，虽然它本身无法精准地表示光滑的函数，但是只要给定一个自变量的区间，在这样的函数堆叠多层之后总是可以调整参数，做到良好地近似给定的函数。事实上，只需四段就可以在区间 $[-1, 1]$ 上用如下的分段线性函数来相当好地近似 x^2 了，例如下面的分段线性函数 $f(x)$：

$$f(x) = 2\mathrm{ReLU}(x - 1) + 2\mathrm{ReLU}(x) + 2\mathrm{ReLU}(-x) + 2\mathrm{ReLU}(-x - 1) - 0.04$$

图像是这样的：

TODO: 这里需要一个图，展示分段线性函数近似光滑函数

另一方面，虽然它的输出会被输入大小的一个常数倍所控制，但在很大程度上，这也避免了在第一章中多项式拟合的数值爆炸问题。此外，这提醒我们应当将模型的输入输出控制在一个范围之内。遵循这些原则，ReLU 网络的表达能力已经足够强大，能解决大多数实际问题。尽管仍有一些细节需要注意，但这并不影响我们对其整体能力的理解。

另外再提一嘴其它的激活函数。Sigmoid 函数[31]是一个 S 型函数，定义为

$$\mathrm{Sigmoid}(x) = \frac{1}{1 + e^{-x}}$$

输出随输入变化的图像是这样的，可见它把输入压缩到了 $[0, 1]$ 的范围内：

TODO: 这里需要一个图，展示 Sigmoid 函数的图像

tanh 函数是双曲正切函数，其定义为

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

它的图像和 Sigmoid 函数很类似，只是经过了一个伸缩和平移，输出范围是 $[-1, 1]$：

TODO: 这里需要一个图，展示 tanh 函数的图像

早期的研究中，它们出现在许多生物学的研究中，可以描述生物神经元的激活或者极化程度，于是人工神经网络出于仿生的考虑也使用了它们。然而它们在两端很小的导数也为优化带来了许多麻烦，导致了 Vanishing Gradient[32]（梯度消失）的问题，后来逐渐被 ReLU 函

[31]Sigmoid 函数：Sigmoid 来源于拉丁语，得名于其类似小写字母 sigma 的 S 形状。

[32]梯度消失：是指在深度神经网络中，由于输出随输入的变化过于小，导致信息无法有效地从输出传回输入，从而使得网络难以优化学习的现象。关于梯度的进一步介绍会在后文给出，此处可以简单理解为信息回传受阻。

数取代，仅在特定层要将输出限制在给定范围内时才使用。虽然近期有研究指出现在的优化器有能力克服这个问题，即使使用 tanh 仍然可以正常地优化，不过这也仅是一个理论上的结果，实际应用中通常认为它们仍然不如 ReLU 函数好用。从此也能看见人工智能的发展并非一帆风顺，仿生不是唯一的出路，人工的神经网络的发展和对其规律的认识必然要走过曲折的探索，才能形成一套独特而成熟的方法论。

不过 ReLU 在 $x < 0$ 的区域也存在斜率为 0 导致梯度消失的问题，为此人们还提出了一些变体，例如 Leaky ReLU 函数，它在 $x < 0$ 的区域也有一个小的斜率，定义为

$$\text{Leaky ReLU}(x) = \begin{cases} x, x \geq 0 \\ \alpha x, x < 0 \end{cases}$$

上式中 α 是一个小的常数，通常取 0.01，它同样简单易于计算。还有一些较为复杂的变体，包括 Gaussian Error Linear Unit (GELU)，Exponential Linear Unit (ELU) 等，都在一定程度上克服了 ReLU 导数为 0 导致信息传播不畅的问题。不过这些都属于工程上的细节问题，读者可以在需要的时候再去了解。

由此我们更加具体化地认识到了神经网络的工作原理：它的基本单元由线性函数与激活函数交替组成。每一层都可以看作是对输入进行线性组合，然后通过激活函数进行非线性变换以实现更复杂的表达能力。这让网络以一种统一的方式来处理输入数据，并有能力通过调整参数拟合复杂的输出。

相关阅读

文中为了简单起见，只是简单介绍了 ReLU 激活函数，关于更多激活函数的定义与性质可以看这篇笔记：

深度学习随笔——激活函数(*Sigmoid*、*Tanh*、*ReLU*、*Leaky ReLU*、*PReLU*、*RReLU*、*ELU*、*SELU*、*Maxout*、*Softmax*、*Swish*、*Softplus*) - *Lu1zero9* 的文章 - 知乎
https://zhuanlan.zhihu.com/p/585276457

形成图形化的直觉许多时候相当重要，这篇文章就给出了一个图形解释：

形象的解释神经网络激活函数的作用是什么？ - 忆臻的文章 - 知乎
https://zhuanlan.zhihu.com/p/25279356

4 神经网络的训练

5 神经网络的优化