

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	2
1.1	最简单的规律——简单线性回归	2
1.2	多项式拟合	8
1.3	高维的线性拟合	15
2	逻辑亦数据	24
2.1	逻辑门	24
2.2	程序是怎么执行起来的	27
3	为什么是神经网络	40
3.1	神经网络：一个大的函数	40
3.2	激活函数与非线性	44
4	神经网络的训练	50
5	神经网络的优化	50
6	神经网络的泛化	50
7	神经网络的可解释性	50
8	神经网络的应用	50
9	神经网络的未来	50
10	结语	50

1 从函数拟合开始

1.1 最简单的规律——简单线性回归

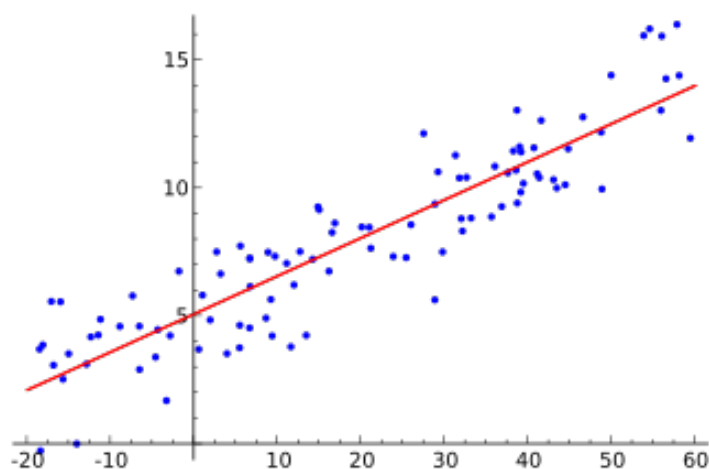


图 1 线性回归示意图

图源: [Wikipedia](#)

Linear Regression

Regression

Linear Fitting

虽然 线性回归 的名字叫做“回归”，但是事实上我更喜欢叫做 线性拟合。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伸到了线性代数。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 $y = ke^{\alpha x}$ 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 ^{Marginal Benefit} 边际效益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。

既然有了基本思路，那么如何选择“贴近的”标准呢？直接去度量一堆散点和直线的接近程度多少有点霰弹枪²打移动靶的感觉，但是我们总是可以计算子弹达到了几环。换言之，两个相差的部分才是关键的，^{Residual}残差的概念由此产生。取出每个点实际值和拟合值的差，就得到了这样一个列表³（其中根据拟合函数 $\hat{y}_i = kx_i + b$ 计算出预测值）：

$$\mathbf{r} = [r_1, r_2, \dots, r_n] = [y_1 - \hat{y}_1, y_2 - \hat{y}_2, \dots, y_n - \hat{y}_n]$$

度量数据点与直线间偏差这一问题就转为了度量残差与 0 的偏差。还记得勾股定理吗？直角坐标系内一点到 0 的距离是坐标平方和的平方根，只不过这里残差列表是个 n 维的向量，度量它偏离原点的成都就是向量的^{Norm}模。这个模越小，说明拟合的效果越好。这样我们就自然地引入了度量拟合效果的量化标准，不过实际应用中出于方便（特别是计算上的方便），通常省去开根号的一步，直接采用残差的平方和，此外还会除以样本点数^{Mean Squared Error}得到“平均”的残差平方。习惯上称之为均方误差（MSE）：

$$\text{MSE} = \frac{1}{n} |\mathbf{r}|^2 = \frac{1}{n} \sum_{i=1}^n r_i^2$$

在踏出下一步之前，我想这里有一点点思考的空间。例如：

- 为什么要用平方和而不是直接相加呢？

这是因为直接相加会有正负相互抵消的可能，度量出的偏差为 0 even 为负实在是不合理，因此至少要保证每一项都是正数。但是这又引出下一个问题。

- 为什么不用绝对值呢？绝对值也是正的啊。

从正态分布的角度看，选用平方和自有它的道理。但是即使读者并不熟悉这些统计的知识背景，也可以从另一个角度理解：平方和的确是一个更好的度量方式，因为它对大的偏差更加敏感。例如一个残差为 2 的点和一个残差为 4 的点，直接绝对值相加的话是 6，在这里残差为 4 的点贡献了 $4/6 \approx 66.7\%$ 的偏差。而它们的平方和是 $2^2 + 4^2 = 20$ ，残差为 4 的点贡献了 $4^2/20 = 80\%$ ，更加凸显了 4 的偏差，反映了我们更“关注”这一大偏差的想法，更符合通常对“偏差”的直观认识。

- 为什么要除以样本点数 n 呢？

²霰弹枪：一种枪，射出的子弹像雨点一样散开

³记号说明：对于变量，无论是一维变量还是多维变量，一律采用斜体。对于具体的数据点，视是否为向量决定使用黑体还是斜体。例如 $\mathbf{r} = \mathbf{y} - \hat{\mathbf{y}}$ 表示的是方程，所以全部采用斜体。但是具体数据的残差计算，例如 $\mathbf{r} = \mathbf{y} - \hat{\mathbf{y}}$ ，是对数据点的向量运算，所以采用正体。

一方面是为了跨数据集比较。数据集的大小通常有区别，就像买东西的重量不同。这正如不能光看价格不看质量就评价 5 元 2 斤的苹果贵还是 3 元 1 斤的苹果贵，因此需要一个“单位”来衡量。另一方面，看完下一个问题你就会明白其中的精妙之处。

- 这里直接把所有的残差平方加了起来，但如果有的点重要一些怎么办？

先说明一下，这样的需求并非空想。有时测量条件决定了不同点的可靠性并不相同。以一个精度 1 % 的表为例，测量得到 1.00, 2.00, 3.00 时它们本身允许的误差分别是 0.01, 0.02, 0.03，而非相同。也就是说我们会觉得 1.00 的测量值从残差的大小上⁴更为可靠，这时似乎应该衡量一下点的“重要性”。如果你想说一个点很重要怎么办？直观上来讲你可能会想把它重复几遍。例如，如果你很关心 r_1 ，你可能会想，这还不简单吗？在误差列表中把 r_1 重复 3 遍就好，就像这样：

$$\text{Refined } \mathbf{r} = [r_1, r_1, r_1, r_2, r_3, \dots, r_n]$$

这时再计算均方误差呢，变成了 $n + 2$ 个点，一种我们设想的“^{Refined}改善的”均方误差公式就变成了这样：

$$\text{Refined MSE} = \frac{1}{n+2} \left(2r_1^2 + \sum_{i=1}^{n+2} r_i^2 \right)$$

只不过这样的方式无疑有点“笨重”。再仔细想想呢？如果把 $1/(n+2)$ 乘到每一项上，就像这样：

$$\text{Refined MSE} = \frac{3}{n+2} r_1^2 + \frac{1}{n+2} r_2^2 + \dots + \frac{1}{n+2} r_n^2$$

再对照者上面的列表看一看， $3/(n+2)$ 不正好表明在大小为 $n+2$ 的列表中 r_1 出现了 3 次吗？频次就这样和 ^{Weight}权重（系数）^{Weighted Error}联系起来了。我们也没必要守着重复 3 次或者 5 次这种固定的规则——至少自然可没有限制重要性之间的比例正好是整数。这样一来只需要一个权重列表就可以了。权重乘在残差平方前，这就引出了 ^{Weighted Error}加权误差，大权重表示更重要。略微改写一下公式得到：

$$\text{Weighted MSE} = \sum_{i=1}^n w_i r_i^2$$

这里为了方便起见，假设了权重的和为 1，即 $\sum_{i=1}^n w_i = 1$ ，如果不为 1，可以先计算误差再除以权重的和。由此可以根据实际情况调整不同点的重要性，也可以看出，之前的均方误差不过是因为在 n 个数中每个残差变量都出现了 1 次，所以权重都设为了 $\frac{1}{n}$ 。在重要性可变时，^{Measurement Metrics}加权均方误差无疑提供了一种更加“通用”的 ^{Absolute Error}度量方式^{绝对误差}。

⁴残差的大小：严谨地说称作 ^{Absolute Error}绝对误差

使用的工具已经准备好了，目标也已经明确了，那么可以开始拟合了。当然，为了简单起见，这里还是只考虑无权重情况。我们要做的是找到一组 ^{Optimal Parameter}最优的参数值 \hat{k}, \hat{b} 使得均方误差最小，从这一点可以窥见贯穿整个机器学习的核心思想——^{Minimize Loss}最小化损失（误差）。形式上，公式会这么写：

$$\hat{k}, \hat{b} = \arg \min_{k, b} \text{MSE} = \arg \min_{k, b} \frac{1}{n} \sum_{i=1}^n (y_i - kx_i - b)^2$$

但是它并没有那么神秘： \arg 是 **argument** 的缩写⁵， \min 则是 **minimize** 的缩写。上面的式子完全可以读作“^{Find the parameter values k, b that minimize the MSE}找到参数值 \hat{k}, \hat{b} 使得均方误差最小”。虽然项很多，但这本质上只是一个二次函数，所以无论是配方法、对 k, b 分别求导还是使用矩阵方法，都可以很容易地求解。不过我很喜欢另一个较少被人提及的视角——从线性代数和几何的角度来看待这个问题。我们回头看看残差的表达式：

$$\begin{aligned} \mathbf{r} &= [r_1, r_2, \dots, r_n] \\ &= [y_1 - (kx_1 + b), y_2 - (kx_2 + b), \dots, y_n - (kx_n + b)] \\ &= [y_1, y_2, \dots, y_n] - (k[x_1, x_2, \dots, x_n] + b[1, 1, \dots, 1]) \end{aligned}$$

我们暂时用一个这样的记号，记拟合所用的函数在这些数据点上的取值

$$\mathbf{x}^0 = [1, 1, \dots, 1]$$

$$\mathbf{x}^1 = [x_1, x_2, \dots, x_n]$$

并记输出 $\mathbf{y} = [y_1, y_2, \dots, y_n]$ ，那么残差就可以写成 $\mathbf{r} = \mathbf{y} - (k\mathbf{x}^1 + b\mathbf{x}^0)$ 。这样一来，我们的目标是找到 k, b 使得 \mathbf{r} 的模最小。写到这里，从代数上看可能依然不够直观，让我们换个角度看看。

TODO: 图 2: 从集合的角度看残差

从几何上， $k\mathbf{x}^1 + b\mathbf{x}^0$ 落在 \mathbf{x}^0 与 \mathbf{x}^1 确定的平面上，求 $\mathbf{r} = \mathbf{y} - (k\mathbf{x}^1 + b\mathbf{x}^0)$ 的最小值实际上就是从点向平面做垂线并求垂线长。平面上的点恰好表示了那些可以精准拟合的数据，而偏离平面的部分则暗示了无论怎么用直线拟合都会有误差。不得不说从几何上看确实清晰很多，事实上也有人从几何角度给出了[推导](#)，不过掠过这些细节，仅保留一个直观的印象也无大碍。本节的几篇推荐阅读中都用不同的方法解答了如何最小化误差，有详细的推导，因此这里不再赘述。但是我认为如果读者有一些基础的统计知识而

⁵Argument: 自变量，数学优化中函数的输入变量。然而 $\arg \min$ 中的 \arg 仅仅表明在优化算法看来 k, b 是可变的、待优化的自变量。但是从拟合模型外看过去，它们是固定的参变量，通常意义上仍称作 Parameter。这里 Argument 与 Parameter 的区别一定程度上体现了视角的转换。

且想记住线性回归推导出的结果，那么结论值得一提，不过跳过也无妨。计算出来的结论是这样的：

首先要计算的是样本中心点，对 b 的导数项为 0 推出最优的直线必然经过样本中心点 (\bar{x}, \bar{y}) ，其中

$$\bar{x} = \frac{1}{n} \sum_{i=1}^n x_i, \bar{y} = \frac{1}{n} \sum_{i=1}^n y_i$$

即 ^{均值} Mean。

看斜率之前先看看 ^{Variance} 方差 和 ^{Covariance} 协方差，方差⁶的表达式是

$$\text{Var} = \frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})^2$$

是不是感觉很熟悉？这不就是自变量相对均值的 MSE 吗？而协方差的表达式是

$$\text{Cov} = \frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})$$

它把方差中的平方项换成了 x 和 y 的 ^{Cross Term} 交叉项，并由此体现出了 ^{Correlation} 相关关系。接下来计算的是斜率 k ，它的表达式是

$$\hat{k} = \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^n (x_i - \bar{x})^2}$$

虽然分子分母都是求和式，看起来有些复杂，但是总结起来其实就是协方差除以自变量的方差，即 $k = \frac{\text{Cov}(\mathbf{x}, \mathbf{y})}{\text{Var}(\mathbf{x})}$ ，如果把协方差看作一种乘法⁷，那么 $k = \frac{\mathbf{x} \cdot \mathbf{y}}{\mathbf{x} \cdot \mathbf{x}}$ 看起来确实挺像那么回事的。

这样一来，通过点-斜率式方程就可以得到最优的直线，那么直线拟合就告一段落了。

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

<https://zhuanlan.zhihu.com/p/72513104>

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

⁶此注释写给学过数理统计的读者：此处并非 ^{样本方差} Sample Variance，样本方差除以的是 $n - 1$

⁷此注释写给熟悉线性代数的同学：在 [向量空间内积](#) 的意义上这几乎正确

<https://zhuanlan.zhihu.com/p/488128941>

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

<https://zhuanlan.zhihu.com/p/139445419>

1.2 多项式拟合

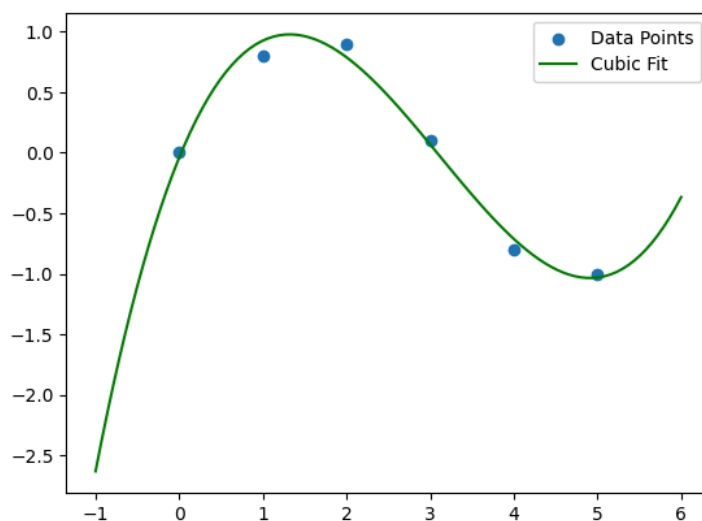


图2 多项式拟合示意图（图为3次拟合）

图源：[GeeksforGeeks](https://www.geeksforgeeks.org/)

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。^{Polynomial Fitting}多项式拟合就是这样一种思路，只是预测 \hat{y} 从 $kx + b$ 变成了 $a_0 + a_1x + \dots + a_mx^{m^8}$ ，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

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

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^{Analytical Solution}解析解，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。在这个情况下，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 \mathbf{r} 的表达式入手

$$\mathbf{r} = \mathbf{y} - (a_0\mathbf{x}^0 + a_1\mathbf{x}^1 + \dots + a_m\mathbf{x}^m)$$

对比之前的表达式，当 a_0, a_1, \dots, a_m 变化时，预测得到的结果 $\hat{\mathbf{y}} = a_0\mathbf{x}^0 + a_1\mathbf{x}^1 + \dots + a_m\mathbf{x}^m$ 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 $m + 1$

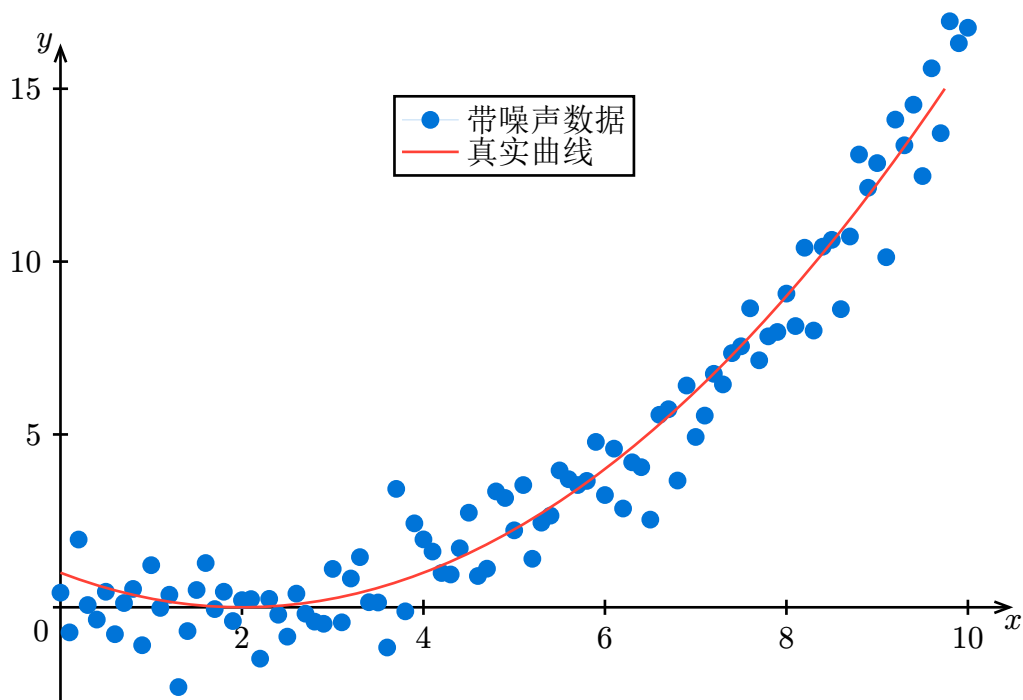
⁸记号说明：虽然习惯上幂次从大到小排列，但是为了下标和幂次的统一性，所以这里选择从常数项到最高次项排列

维的子空间。求最小模的 \mathbf{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 ^{Overfitting} 过拟合。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。

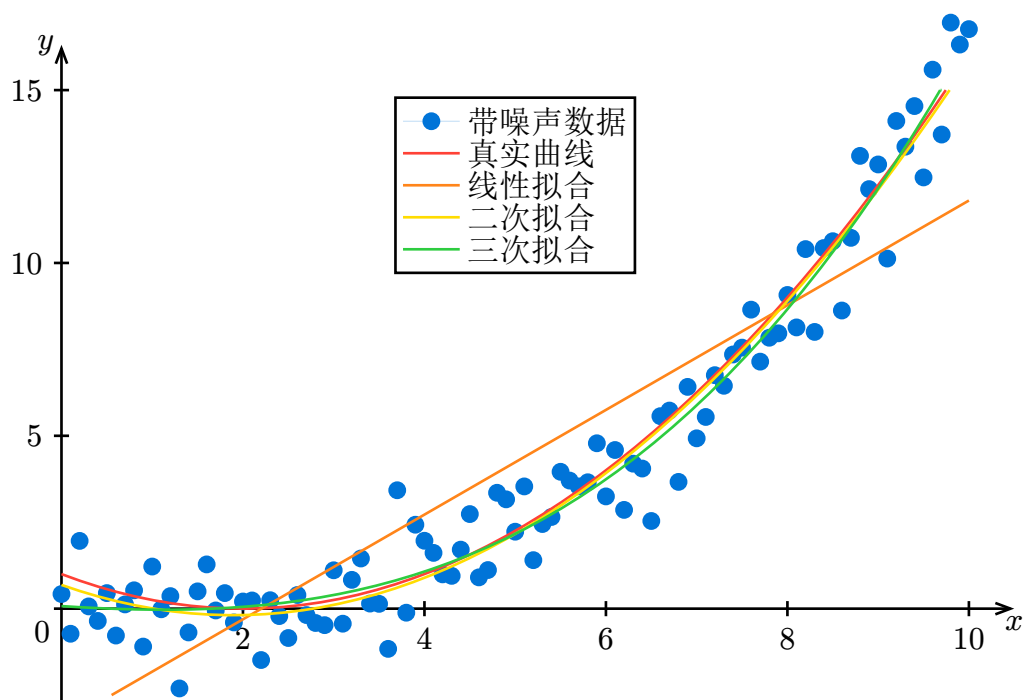
拿做题打个比方：使用直线拟合明显不线性的数据是方法错了，只能说是没完全学会。但是用接近数据量的参数来拟合数据，留给它的空间都够把结果“背下来”了，捕捉到了数据的细节，却忽略了数据背后的规律，化成了一种只知道背答案的自我感动。在几道例题上能做到滴水不漏，但是一遇到新题就束手无策。

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 $y = 0.25x^2 - x + 1 + \mathcal{N}(0, 1)$ ⁹。

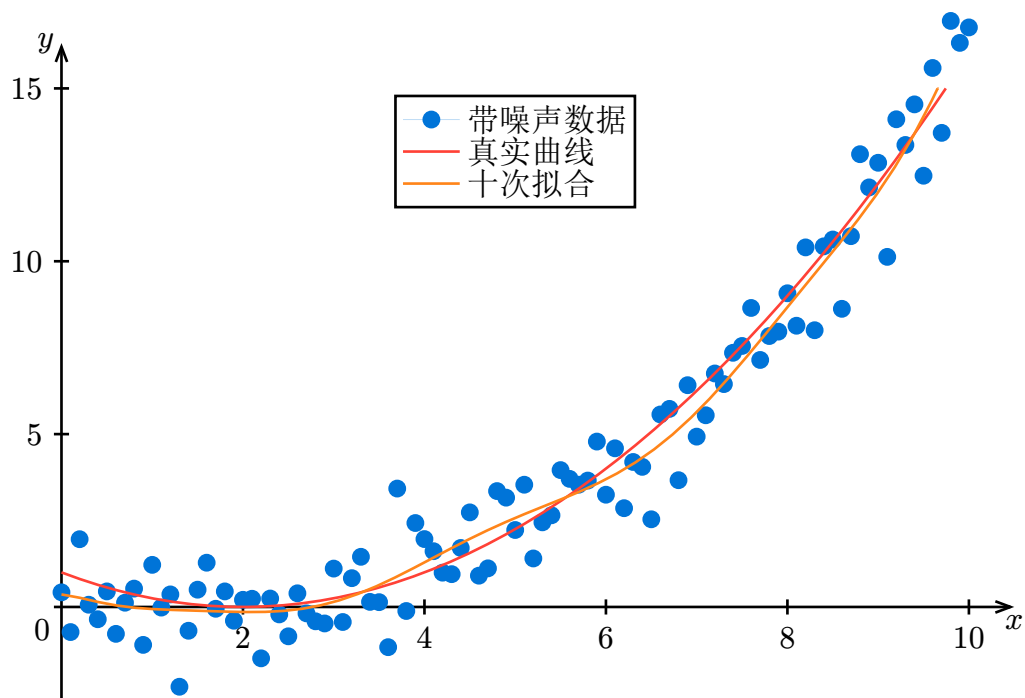


那么现在我们来试试用不同次数的多项式拟合这个数据集。不难看出线性拟合的线与数据点还是相差不少，因为它没能提供可以制造数据“弯曲”形状的项，它没能捕捉到数据更加复杂的趋势，这种现象称为 ^{Underfitting} 欠拟合。2 次曲线的效果几乎和真实曲线一样，即使提升到 3 次也没有太明显的改变，它们拟合的效果都还算好。

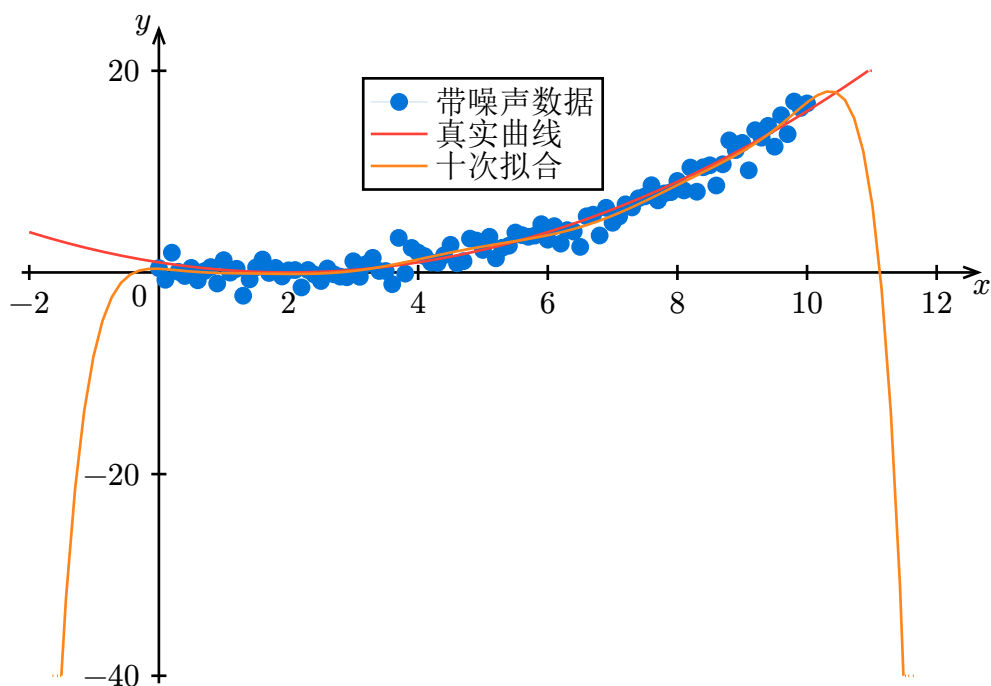
⁹ $\mathcal{N}(0, 1)$: 表示一个服从[标准正态分布](#)的变量，均值为 0，方差为 1



但是如果继续增加次数呢？先来看看十次的拟合效果。



你可能会想，虽然是稍微歪了一点，不过这看起来还行吧。但是如果你把 x 的范围稍微扩大一点，你就会发现势头完全不对了。



一旦离开了拟合的区域，十次拟合的曲线就直勾勾地弯向无穷远，这是因为它把噪声也拟合进去了，从而给出了 ^{泛化性}Generalization¹⁰ 极差的结果。这就是过拟合的危害。因为参数量与样本点数量并没有非常显著的差别（10 个参数，100 个样本点），所以从去噪声的角度看，结果过拟合并不奇怪——过滤掉噪声需要更多的数据。

当然解决办法并不是没有，要解决问题先要找到问题的根源。既然得到的函数行为不符合预期，那么很自然地我们会想问，这个函数的系数怎么样呢？在上面这个具体的例子中，函数的表达式是

$$\begin{aligned}\hat{y} = & -0.000004129005667x^{10} + 0.000200033877258x^9 - 0.004061827595427x^8 \\ & + 0.044810202155712x^7 - 0.291097682876070x^6 + 1.129425113256322x^5 \\ & - 2.542208192992861x^4 + 3.091776048493755x^3 - 1.584353162512058x^2 \\ & - 0.267618886184698x + 0.362912675589959\end{aligned}$$

简单估算一下就会发现，例如 3, 4 次项的系数都在个位数级别，再乘以 x 的 3 次方、4 次方数值就会变得很大。10 次方项的系数看起来只有 4.1×10^{-6} ，但是乘上 x 中最大值的 10 次方，也就是 10^{10} 后，这个数值同样会飙升到上万的级别。一堆上万级别的数加在一起，倒不如说顶着舍入误差¹¹还能够回归到原来的数据集上已经是奇迹了。也就只有

¹⁰泛化性：预测原有数据集以外点的能力

¹¹舍入误差：就像手动计算时保留几位小数一样，计算机计算的并不是“实数”，而是具有一定精度的浮点数，同样也有误差。例如对于 32-bit 的浮点数，只能精确到 7 个十进制位，这意味着从万位向后数到第 7 位，从百分位就可能已经不准确，在此之后的数位就不太可靠了。

MSE 可以限制一下它在数据集内的行为，出了预定义的范围，这个高次函数大概就放飞自我了。

不过如果一定要用高次函数，补救的办法也不是没有。既然这些系数导致了很大的数值，那限制一下这些数值就好了，这就是 ^{Regularization} 正则化 的思路。我们在待优化的函数上加上一个 ^{Penalty Term} 惩罚项，同样地使用平方求和的形式，只不过这次是对系数进行惩罚，为了让系数尽量小，即尽量贴近于 0，自然想到把它们乘以权重后的平方也加起来，优化的目标¹²变成了

$$\text{Loss} = \text{MSE} + \underbrace{\sum_{i=1}^{10} (\mu_i a_i)^2}_{\text{正则化项}}$$

可调参数 μ_i 表明我们希望在多大程度上抑制每个系数，在这个例子中不同次项的权重可能并不相同，不过在后续拟合的许多例子中这个会使用统一的权重，即所有的 μ_i 均为相同的定值 μ ，式子从而变为了一个常数 $\lambda = \mu^2$ 倍的平方和。为每一项设置分立的系数是为了解决一个致命的问题：不同系数对最终结果的影响可能不同。例如在 $x = 10$ 这一点上， x^{10} 项的系数对结果的影响远远大于 x 项的系数，即使是 4.1×10^{-6} 这样微小的 10 次项系数也会导致非常大的数值。平方后这一系数变得十分微小，原本用于约束系数大小的正则化项对它的影响更是微乎其微。

不过即使我们使用了相同的系数 λ ，也仍然有一些技巧可以帮我们把高次项压下去。我们发现一路走来导致问题的根源都是 x 的高次项即使在系数很小时也会导致数值爆炸。但是如果把 x 放到 $[-1, 1]$ 的闭区间内呢？这样即使是 x^{10} 项， x 的 10 次方也不会超过 1，系数再怎么大，至少在小范围内也不会导致数值爆炸，这样一来正则化项才能发挥其约束作用。

操作上只需要把 $[0, 10]$ 范围内的 x 线性地映射到 $[-1, 1]$ 范围内。这很简单，令 $z = 0.2x - 1$ 再对 y 用 z 的多项式拟合，这种操作称作 ^{归一化} Normalization¹³。

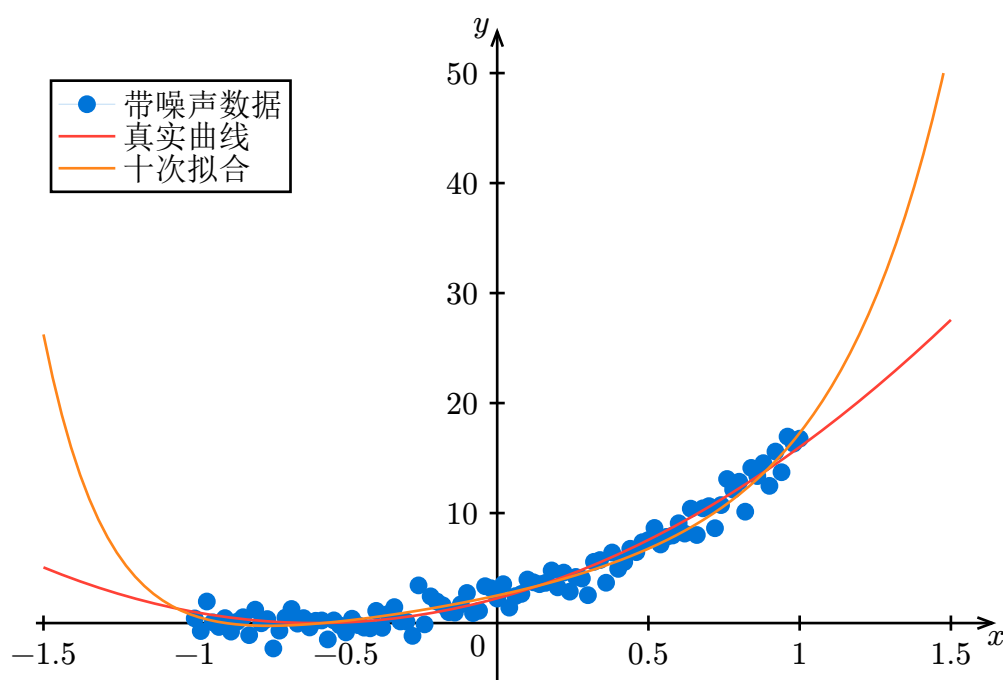
事实上只需要一个很小的 λ 就可以在一定程度上抑制高次项的系数，这里我们取 $\lambda = 0.01$ ，优化的目标变为了

$$\text{Loss} = \text{MSE} + 0.01 \sum_{i=1}^{10} a_i^2$$

¹²Loss: 损失，与前文单纯使用 MSE 时相同，我们希望让它尽可能小，它的每一项包含了我们对拟合结果的一个美好“祝愿”，MSE 项希望它误差减小，正则化项希望它系数正常。

¹³归一化：调整数据到某个给定的范围内，使数据在不同场景下更加可比、更加数值稳定。前文计算误差时取平均实际上也是一种归一化。

这时拟合出来的图像是这样的：



虽然图中拟合曲线与真值在数据集外确实仍然有显著的差距，但经过自变量归一化和参数正则化项的加入，拟合的曲线至少把数据的大体趋势成功地延伸到了数据集的一个邻域内，不至于像原本的那样惨不忍睹。

不过在实际应用中，几乎不会用到5次以上的多项式拟合。仍然是因为容易过拟合：就以 $[-1, 1]$ 上的函数为例， x^5 与 x^7 的图像几乎是一样的，它们最大的差值仅为 0.12，这意味着如果允许的高次项太多，一点点微小的噪声就能让轻易地把五次项的系数“分给”七次项，或者反之。这导致拟合的数值稳定性很差，因此显然不太可靠。从这种影响的角度看，高次多项式拟合本身就有 ^{奇异性} Singularity，解并不稳定（这种对微小噪声敏感的问题常称为 ^{病态问题} Ill-posed Problem）。因此比正则化或者归一化更重要的是，我们应该意识到高次函数并不是万能的。当你觉得需要用到很高次的函数才能成功拟合时，不如先想想，多项式的假设真的合适吗？

我们注意到一个重要的事实：虽然拟合的参数在变化，但是拟合前仍然需要人为地设定多项式的次数，正则项（如果有的话）权重也需要人为设定。这些 ^{先验} Prior 的 ¹⁴ 参数 ^{超参数} 通常称为 Hyperparameter。如何用模型拟合固然是重要的问题，但是模型的结构，包括如何选取适当的超参数也有学问。因为它们通常不是直接从数据中学习的，而需要人为设定。[有道是](#)“学而不思则欠拟合，思而不学则过拟合”。参数太少就会像那直线拟合曲线一样，必然导致拟合的精度不足。参数太多则会受到太多的噪声干扰，像是拿高次

¹⁴先验：在观测到数据之前，我们已经了解了一些数据特征。

函数拟合低次函数一样因为一点噪声导致函数行为夸张，在预定义的数据集外两眼一抹黑。只有在一定程度上了解问题的本质，才能选出合适的拟合模型。

```
begin{tcolorbox}[myrecommendbox, title=推荐阅读, breakable=false] begin{itemize}
item 如果你想看更多关于多项式拟合的实战，可以阅读：\textit{多项式拟合的介绍与例子 - 姓甚名谁的文章 - 知乎}\url{https://zhuanlan.zhihu.com/p/366870301} item 如果你曾经想过拿问卷调查来做拟合，可以看看：\textit{理科生觉得哪些知识不知道是文科生的遗憾？ - 一只小猫咪的回答 - 知乎}\url{https://www.zhihu.com/question/270455074/answer/2374983755} item 这个比喻很好，同一问题下的其它回答也很有趣：\textit{人的大脑会不会出现“过拟合”病？ - 莲梅莉 usamimeri 的回答 - 知乎}\url{https://www.zhihu.com/question/625846838/answer/3250463511} end{itemize} end{tcolorbox}
```

推荐阅读

如果你想看更多关于多项式拟合的实战，可以阅读：

多项式拟合的介绍与例子 - 姓甚名谁的文章 - 知乎

<https://zhuanlan.zhihu.com/p/366870301>

如果你曾经想过拿问卷调查来做拟合，可以看看：

理科生觉得哪些知识不知道是文科生的遗憾？ - 一只小猫咪的回答 - 知乎

<https://www.zhihu.com/question/270455074/answer/2374983755>

这个比喻很好，同一问题下的其它回答也很有趣：

人的大脑会不会出现“过拟合”病？ - 莲梅莉 usamimeri 的回答 - 知乎

<https://www.zhihu.com/question/625846838/answer/3250463511>

1.3 高维的线性拟合

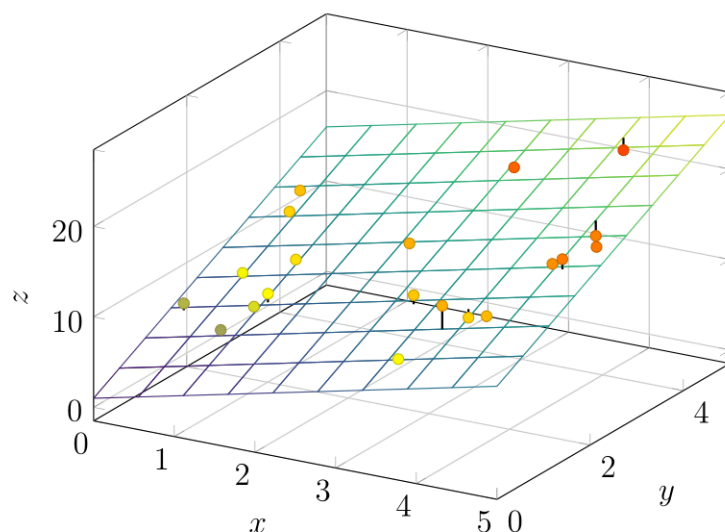


图 8 高维线性拟合示例

第一节我们介绍了“简单线性回归”，即只有一个自变量的线性回归。但是在实际问题中，自变量往往不止一个，这时一元的线性回归就需要改成 ^{Multiple Linear Regression} 多元线性回归。不过按照我的习惯，文中仍然称为“拟合”。

现实世界中的数据往往是多维的，就以估计体重为例，不难发现年龄和身高就是两个可能相关的变量。如果我们想用一个模型来描述这种相关性的话，最简单的就是线性模型了，与之前的 $\hat{y} = kx + b$ 类似，自然想到用这样的函数¹⁵去拟合数据：

$$\hat{y} = w_1x_1 + w_2x_2 + \dots + w_dx_d + b$$

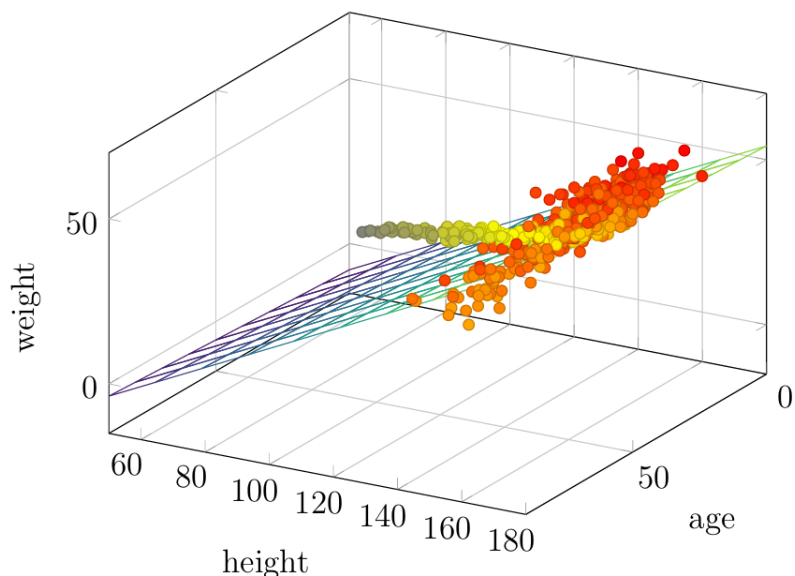
同样地，优化的目标仍然是最小化均方误差，即对 n 个数据点令

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

通过最小化误差得到 $w = [w_1, w_2, \dots, w_d]$ 和 b 的值。这个过程与一元线性回归的过程是类似的，只不过自变量是一维时，可以在平面上直接画出拟合的直线，二维时可以在空间中画出平面，但是当维数增加到三维及以上时，拟合所用的线性函数就变为 ^{Hyperplane} 超平面了，我们无法直观地看到这个超平面，但是可以猜测，它的原理差不多。

话不多说，先看看效果。这里以美国人类学家 Richard McElreath 搜集到的一个年龄、身高与体重数据集为例，它的分布与拟合出来的平面是这样的：

¹⁵记号说明：这里使用字母 w 表示 ^{weight} 权重， b 表示 ^{bias} 偏置，即常数项， d 表示的是空间的 ^{dimension} 维度。



通过拟合，我们可以得到一个超平面，它大致描述了数据的分布。这个超平面的方程是 $0.04676645038926784 \cdot \text{age} + 0.47766688346191755 \cdot \text{height} - 31.805656676953056 = \widehat{\text{weight}}$ ，它比单纯使用身高或者年龄的拟合效果都要好一些。由此还可以量化地看到，年龄与身高都会影响体重，但是年龄是弱相关，而身高是强相关，这也符合我们的日常经验。

不过正如我们之前一直在做的一样，让我们看看更为直观的几何视角。仍然用 \mathbf{x}^0 表示全 1 的向量，使用 $\mathbf{x}_{:1}$ 表示所有样本的第一个 ^{Feature} 特征（分量）， $\mathbf{x}_{:2}$ 表示所有样本的第二个特征，以此类推¹⁶。那么多元线性拟合时的残差向量变为了¹⁷

$$\mathbf{r} = \mathbf{y} - \hat{\mathbf{y}} = \mathbf{y} - (b\mathbf{x}^0 + w_1\mathbf{x}_{:1} + w_2\mathbf{x}_{:2} + \dots + w_d\mathbf{x}_{:d})$$

如果回顾一下我们在多项式拟合一节的内容，就会发现这和多项式时的残差向量

$$\mathbf{r} = \mathbf{y} - (a_0\mathbf{x}^0 + a_1\mathbf{x}^1 + \dots + a_m\mathbf{x}^m)$$

有着惊人的相似之处。细心的读者可能已经发现，如果令这些分量 $\mathbf{x}_{:1}, \mathbf{x}_{:2}, \dots, \mathbf{x}_{:n}$ 分别为 \mathbf{x} 的幂次组成的向量 $\mathbf{x}^1, \mathbf{x}^2, \dots, \mathbf{x}^d$ ，那么我们得到的完完全全就是多项式拟合。这也就意味着，多项式拟合实际上可以视为多元线性拟合的一种特殊情况。

事已至此，我们似乎已经许多次遇到了这样一种情况：从一面看过去，是代数上，一组样本点上的线性拟合。但是从另一面看过去，确是在几何上找到高维空间的超平面中最接近给定点的向量。这里其实有不少精妙的数学原理¹⁸，但是考虑到这里的主题是机

¹⁶记号说明：冒号表示取所有行，这是为了与 Python 中 Numpy, Torch 等库的列切片语法 $a[:, j]$ 对齐。

¹⁷记号说明：在公式中我特意将常数项放到了最前面，这是为了让它和多项式拟合的形式保持一致。

¹⁸写给数学基础好的读者：这本质上体现了代数与几何的 ^{Duality} 对偶性。

器学习，我将只带读者简要地复习（或者学习）一下线性代数，更为系统性地从几种略有差距的视角¹⁹ 体会矩阵的本质。

在绘图讲解前，我首先要感谢 [《线性代数的艺术》](#)（*The Art of Linear Algebra*）

这篇笔记，我第一次读到便感到文中的插图绘制非常精妙。它的思路是顺着 Gilbert Strang 教授书籍 [《写给所有人的线性代数》](#) 的思路，使用图形化的方式来解释线性代数的概念。认为可以看成是一本矩阵图鉴，对理解矩阵运算有着极大的帮助。

[3Blue1Brown 的线性代数系列](#) 也是优质线性代数学习资源。这个制作精良的合集仅用不到两个小时的视频就清晰地从几何的角度讲明白了线性代数的基础知识，也是我入门线性代数的第一课。

矩阵有很多种 ^{Interpretation} 解读，不过我觉得大致可以按照是否把行看作一个整体以及是否把列看作一个整体来分为四类。

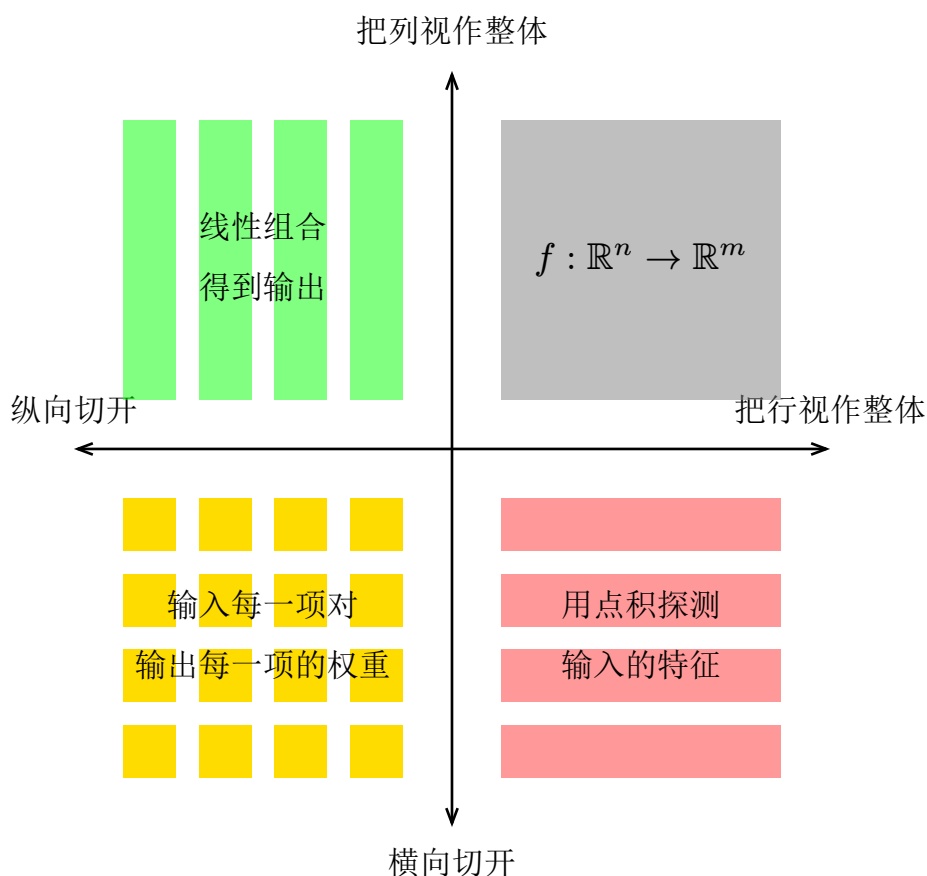


图 10 矩阵运算的直观理解

矩阵究竟是什么，我们的大学教了很多年，也没有完全搞清楚。从数学的角度看，可能 [Linear Algebra Done Right](#) 的思路比较好，搞了个向量空间起手，全程以 ^{Map} 映射的逻辑贯穿。但是在国内，大部分教材一上来前两章就是讲行列式的计算，教学内容逐渐搞僵化了。

¹⁹ 几种视角：Overall 整体解读、Row-wise 按行解读、Column-wise 按列解读、Element-wise 按元素解读。

既然是服务于机器学习，许多内容²⁰我们一概砍掉，只留下最为基础的内容。第一种视角就是作为 $\mathbb{R}^n \rightarrow \mathbb{R}^m$ 的线性映射。使用矩阵的第一个重要目的就是要把一个线性映射“打包”成一个符号，毕竟只有这样才能方便地书写、推导和计算。从工科的视角看来，一个“向量”无非是一个数组，而一个“线性映射”实际上就是吃进去一个数组，吐出来另一个数组的机器²¹。矩阵作为一个二维数组，忠实地记录了这个机器的所有参数。它的运算规则是这样的：

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n \\ a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n \\ \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \dots + a_{mn}x_n \end{bmatrix}$$

从输出的表达式中，我们自然引出了行的视角。如果把矩阵看作若干行：

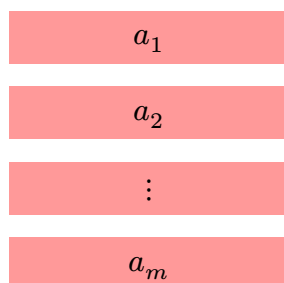


图 11 矩阵的行视角

我们会发现输出 y 的每一个分量 y_i 都是输入 x 与行 a_i 的点积²²。例如

$$y_1 = a_1 \cdot x = a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n$$

Homogeneous

Weighted Sum

诚然，每一行都是一个 齐次的 ²³线性函数，它的形式也只能是这种 x 的 加权和，但是相信一定会有读者好奇：点积衡量了两个向量的相似程度，那么这里做点积的几何意义是什么呢？答曰：探测输入的特征。

我们可以把矩阵的每一行看作一个 ^{Feature Detector} 特征检测器，它的方向表明了待检测的特征方向，与 x 的点积则说明了这个输入在这个特征上的响应强度(通常称为 ^{Feature Response} 特征响应)。当 x 与特征的方向相近时响应的值为正，而当 x 与特征的方向相反时响应的值会为负，当 x 与特征几乎无关时响应的值会接近于零。这是点积的几何特性，至少从理论上为特征提取画出了一条路径。

²⁰许多内容：线性方程组的求解、行列式、^{Change of Basis} 基变换、^{Eigen Decomposition} 特征分解、^{Quadratic Form} 二次型。

²¹此注释写给编程基础较好的同学：这里的机器指的就是编程中的 ^{Function} 函数。

²²严格地说 a 是行向量， x 是列向量，这中间在数学上有一些差别，是矩阵乘法而不是点积。但是在计算机存储中，因为都是一维数组，从实用的角度并不需要纠结于此，这种 ^{Abuse of Notation} 记号混用 就见怪不怪了。

²³齐次：指不带偏置(常数)项。

这时如果考虑怎么样的输入可以获得接近预期的输出呢？根据几何解释，其实就是试图找到一个输入向量，让它尽可能通过这些特征检测器，使得每一个特征检测器的响应值与预期的响应(输出的对应分量)尽可能接近，并考虑使用均方误差来“惩罚”不接近的程度，我们一开始的代数解释便是如此。

但是如果改改输出的写法，把矩阵切成若干列，我们又有了一个不同的视角，不过这里我们用 $a_{:j}$ 表示它的列：

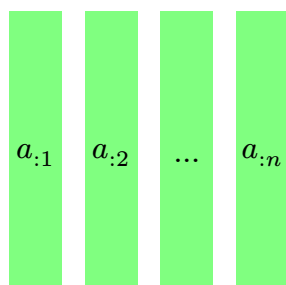


图 12 矩阵的列视角

这样看来，矩阵的乘法也可以写成

$$Ax = [a_{:1} \ a_{:2} \ \dots \ a_{:n}] \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

$$= x_1 a_{:1} + x_2 a_{:2} + \dots + x_n a_{:n}$$

也就是说，输出写成了输入的线性组合。这时我们在输出的空间 \mathbb{R}^m 操作，而输入的空间 \mathbb{R}^n 仅仅是作为权重的载体，给出了这些列向量应该以怎么样的比例组合。

这时我们要怎么考虑用输出反推合适的输入这个问题呢？随着输入的变化，输出会变为列向量的不同组合方式，正如之前所说的，我们需要在这些列向量线性组合形成的超平面上找点，让它和预期的输出尽可能接近，这就是我们在前文提到的最小二乘的几何视角。

最后一种角度则带有更为浓重的 ^{Distruction} 解构 ²⁴ 色彩：把矩阵看成一个数表，作为一个填了数字的 $m \times n$ 矩形：

²⁴解构：哲学术语，通常指的是对一个结构或概念进行拆解、分析。

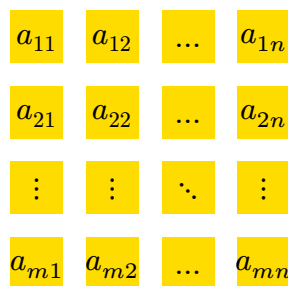


图 13 矩阵的元素视角

一般来说，我们的教材都是这么引入的，但是就像我刚才提到的一样，这种理解有着一股解构的色彩。如果没有解构后重新²⁵建构^{Construction}，这种理解很容易让人迷失在行列式、特征值、特征向量等等复杂计算的汪洋大海中，从而忘记了矩阵的本质。那么这种解释有什么意义呢？我认为它的作用就在于 a_{ij} 体现了第 j 个输入对第 i 个输出的权重。

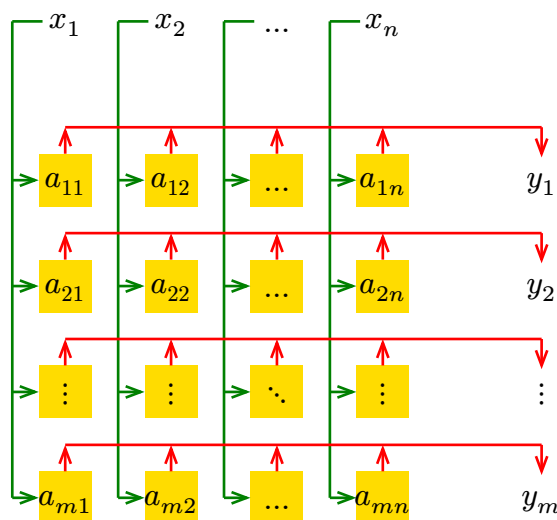


图 14 矩阵乘法的地址线视角

矩阵乘法可以看成这样：输入的每个分量沿着列地址线²⁶输入到每一列的所有块，由每个元素乘上对应的权重后，将结果“上传”到对应的行地址线上，最后行地址线上的所有块累积起来得到输出的每个分量。

这个图还有另外一个很常见的呈现形式，把它看成一个无偏置的^{Linear Layer}线性层²⁷：

²⁵建构：哲学术语，通常指的是对一个结构或概念进行重建、整合。

²⁶地址线：计算机内存中的概念，虽然逻辑上内存是连续的，但是实际上内存寻址时有很多层，最底层时被选择的内存芯片是通过行和列寻址的，物理上由两条地址线输入^{Row/Column Address Strobe}行/列地址选通信号。

²⁷线性层：在后面章节的机器学习中会成为一个基本模块，本质上就是从输入 x 得到输出 $y = Ax + b$ 的过程，这里的无偏置即 $b = 0$ 。

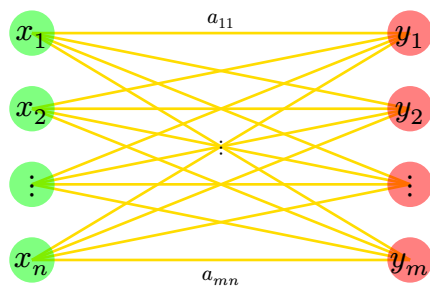


图 15 线性层连接图

每条从 x 到 y 的连线都代表有一个从 x 到 y 的权重，我们给 x_j 到 y_i 的连线赋予权重 a_{ij} ，它就表明了输入 x_j 是如何影响输出 y_i 的。

至此我们已经从四个有差别但是又有联系的视角理解了矩阵的行为，不过我觉得我对于不同的解释还有一点观察。当降维时，特征提取(行的视角)体现的更明显，而当升维时，特征组合(列的视角)更为重要。降维伴随着对信息的压缩和精简，通过去掉不重要的部分，更接近事物的本质。升维不仅仅是增加维度，更是通过新的空间来赋予数据提供更多的可变性。在这个过程中，每一列代表了一个基向量，整个矩阵的列向量按比例组合出高维的结果。

花了一些篇幅来复习线性代数，是时候回到多元线性拟合的问题上了。不过这次可以使用矩阵的语言来描述这个问题了。假设我们有 n 组 d 维的 x 的取值，它们组成了一个 $n \times d$ 的矩阵 \mathbf{x} ²⁸。我们的目标是找到一个 d 维的 w ，使得数据集上 $x \cdot w$ 尽可能接近 y ，现在 \hat{y} 的表达式用点积的语言可以简洁地写为 $\hat{y} = x \cdot w + b$ 这种形式。

但是通过一点点技巧可以让问题更简洁，在拟合函数中，我们可以把 b 合并到 w 中，也就是

$$\begin{aligned}\hat{y} &= x \cdot w + b \\ &= x_1 w_1 + x_2 w_2 + \dots + x_d w_d + b \\ &= [x_1 \ x_2 \ \dots \ x_d \ 1] \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_d \\ b \end{bmatrix}\end{aligned}$$

对于样本，把所有的行并起来就变成了不需要额外添加偏置的 $\hat{\mathbf{y}} = \tilde{\mathbf{x}}\tilde{\mathbf{w}}$ 。如果说这个 $\tilde{\mathbf{x}}$ 是什么，它就是把 \mathbf{x} 的每一行拼上一个 1：

²⁸记号说明：使用大写字母表示矩阵的比较多，但是这里为了美观和符号的一致性，我们仍然采用黑体小写字母表示所有数据点的集合。 \mathbf{x}_i 表示第 i 个数据点， $\mathbf{x}_{:j}$ 表示所有数据点的第 j 个分量， x_{ij} 表示第 i 个数据点的第 j 个分量，因为是标量，所以采取小写。而在方程 $x \cdot w$ 中， x, w 并非数据点的集合，而是变量，故虽然为向量，但是采用斜体。

$$\begin{bmatrix} \hat{y} \end{bmatrix} = \begin{bmatrix} \mathbf{x}_1 & & & \\ \mathbf{x}_2 & & & \\ \vdots & & & \\ \mathbf{x}_n & & 1 & \end{bmatrix} \cdot \begin{bmatrix} w \\ \\ \\ b \end{bmatrix}$$

图 16 多元拟合的矩阵表示（行视角）

但是如果改成用列的视角来看待矩阵 $\tilde{\mathbf{x}}$ ，我们会发现问题变成了用 $\mathbf{x}_{:,1}, \mathbf{x}_{:,2}, \dots, \mathbf{x}_{:,d}$ 与 \mathbf{x}^0 的线性组合来贴近 \mathbf{y} :

$$\begin{bmatrix} \hat{y} \end{bmatrix} = \begin{bmatrix} \mathbf{x}_{:,1} & \dots & \mathbf{x}_{:,d} & 1 \end{bmatrix} \cdot \begin{bmatrix} w_1 \\ \vdots \\ w_d \\ b \end{bmatrix}$$

图 17 多元拟合的矩阵表示（列视角）

诶？这不是函数拟合问题吗？如果把 \mathbf{y} 和 $\mathbf{x}_{:,j}$ 看作是关于 i 的函数（这里函数定义在 $\{1, 2, \dots, n\}$ 上，换言之，记 $\mathbf{y}(1) = y_1, \mathbf{y}(2) = y_2, \dots, \mathbf{y}(n) = y_n$ ，同理，设 $\mathbf{x}_{:,j}(i) = x_{ij}$ 。那么我们的问题就是用 d 个函数 $\mathbf{x}_{:,1}(i) \sim \mathbf{x}_{:,d}(i)$ 与一个常值函数 $\mathbf{x}^0(i) \equiv 1$ 来拟合一个给定的函数 $\mathbf{y}(i)$ 。

去掉常数项（或者把常数项也当成一个 $\mathbf{x}_{:,d+1}$ ）并把自变量 i 当成一元函数拟合问题中的 x 就可以发现：这与使用给定函数 $f_1(x) \sim f_{d(x)}$ 的线性组合，在若干数据点上拟合给定函数 $f(x)$ 这一问题没有任何差别。由此可见，用给定的函数集来拟合一个函数本质上与多元线性拟合有着完全相同的数学结构。

至此，我们至少已经初步理解了线性拟合。在这一章的最后，让我们做个总结。

- 最开始我们引入了一元的线性拟合，学会了最小二乘法与最小化损失以优化参数的基本思想，也学会了如何给数据加权。
- 接下来来到了多项式拟合，虽然是曲线，但是如果把 x 的方幂看作线性独立的分量，这仍然可以看作是一种线性拟合，在这里我们领略到了过拟合的危害，也理解了为什么要使用正则化与归一化方法。

- 在过拟合中，我们得到的更重要的启示是要意识到高次并不意味着万能，合适的才是最好。如果参数足以存下所有的数据，那么大概率就会过拟合。
- 最后我们引入了多元线性拟合，通过矩阵的不同解读，我们理解了特征提取与特征重组的逻辑。
- 从矩阵的行、列解读中，我们意识到多元线性拟合与函数的线性拟合本质上是一样的，这里有一种精妙的对应关系。

推荐阅读

The Art of Linear Algebra 这篇文章非常好，但是如果你上不去 GitHub，进这个知乎回答看也行：

如何快速理解线性代数？ - 小奇迹 | 的回答 - 知乎

<https://www.zhihu.com/question/30726396/answer/3124578647>

这篇文章推荐给数理统计与线性代数都学的较好的读者：

回归分析|笔记整理（6）——多元线性回归（上） - 学弱獐的文章 - 知乎

<https://zhuanlan.zhihu.com/p/48541799>

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^非一个约定，我们将 0 视作^假 False，1 视作^真 True²⁹。先来看看几种最简单的逻辑运算。

1. Not（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³⁰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

表 1 与的真值表

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

表 2 或的真值表

²⁹逻辑 0/1：在物理上，逻辑 0 由^{低电平} Low 表示，逻辑 1 由^{高电平} High 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。

³⁰真值表：逻辑运算的输出与输入的关系表。

在图上这些运算一般会这样表示：



图 18 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³¹。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：

$$\begin{array}{r} 101 \\ + 011 \\ \hline 1000 \end{array}$$

表 3 二进制加法

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：

$$\begin{array}{r} \dots 1 \dots \\ + \dots \textcolor{red}{1} \textcolor{blue}{0}_1 \dots \\ \hline \dots 1 \dots \end{array}$$

表 4 二进制加法

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：

$$\begin{aligned} s &= (x \text{ xor } y) \text{ xor } c \\ c_n &= (x \wedge y) \vee (c \wedge (x \text{ xor } y)) \end{aligned}$$

³¹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。

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：

加数 1 x	加数 2 y	后方进位 c	加和 sum	向前进位 c'	结果位 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³²，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³³。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³⁴，两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>

³²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。

³³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。

³⁴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³⁵。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硅基生物。如果你看到这里看累了，去看看小说放松一下吧。

³⁵约翰·肯尼迪在登月演讲中也引用了这句话。

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³⁶

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 与 Instruction Set，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”³⁷上遵照着一份“指南”³⁸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。

³⁶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)

³⁷草稿纸：比喻计算机的 RAM，暂且把它理解为每格写了一个整数，实际计算机中是字节。

³⁸指南：比喻计算机的程序，是计算机要执行的 Instruction。

- `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。
^{立即数值}
`#x` 表示 Immediate 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³⁹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⁴⁰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` 或者你想要的任何数字，右侧的列表则表示对应的指令执行后的内存状态：

指令	x_1, x_2, x_3, \dots
(initial)	$x, y, _, \dots$
→ <code>mov 2, 0</code>	x, y, x, \dots
→ <code>mov 0, 1</code>	y, y, x, \dots
→ <code>mov 1, 2</code>	y, x, x, \dots

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 Effect⁴¹^{副作用}，但似乎不太可能既不修改其它内存，又交换数值⁴²。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⁴³，我们就此“发明”了 Register⁴⁴^{寄存器}。就假设我们

³⁹内存 `0`：按照计算机中的习惯，计数从 `0` 开始。
⁴⁰你先别管它怎么运行起来的。
⁴¹副作用：指令运行的过程中对其他地方产生的影响。
⁴²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。
⁴³临时储存：可以理解为一种草稿纸，内容可以随时丢弃

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⁴⁵。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 ^{易失} Volatile 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 号寄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⁴⁶，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）：

⁴⁴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。

⁴⁵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。

⁴⁶别名：例如在本例中，它指代 add r0, [r1] 所在的行

指令	$[r_0, r_1, r_2, \dots$	x_8, x_9, x_{10}, \dots	$x_{15}, x_{16}, x_{17}, \dots$	解释
(initial)	?, ?, ?, ...	15, 18, ?, ...	1, 2, 3, ...	
→ mov r0, #0	0, ?, ?, ...	15, 18, ?, ...	1, 2, 3, ...	向 r_0 写入 0
→ mov r1, 8	0, 15, ?, ...	15, 18, ?, ...	1, 2, 3, ...	将 x_8 的 15 赋给 r_1
→ mov r2, 9	0, 15, 18, ...	15, 18, ?, ...	1, 2, 3, ...	将 x_9 的 18 赋给 r_2
→ add r0, [r1]	1, 15, 18, ...	15, 18, ?, ...	1, 2, 3, ...	$r_1 = 15$, 取 $x_{15} = 1$ 加到 r_0
→ add r1, #1	1, 16, 18, ...	15, 18, ?, ...	1, 2, 3, ...	r_1 加 1 (指向 x_{16})
→ cmp r1, r2	1, 16, 18, ...	15, 18, ?, ...	1, 2, 3, ...	→(16≠18, 跳回 loop)
→ add r0, [r1]	3, 16, 18, ...	15, 18, ?, ...	1, 2, 3, ...	$r_1 = 16$, 取 $x_{16} = 2$ 加到 r_0
→ add r1, #1	3, 17, 18, ...	15, 18, ?, ...	1, 2, 3, ...	r_1 加 1 (指向 x_{17})
→ cmp r1, r2	3, 17, 18, ...	15, 18, ?, ...	1, 2, 3, ...	→(17≠18, 跳回 loop)
→ add r0, [r1]	6, 17, 18, ...	15, 18, ?, ...	1, 2, 3, ...	$r_1 = 17$, 取 $x_{17} = 3$ 加 到 r_0
→ add r1, #1	6, 18, 18, ...	15, 18, ?, ...	1, 2, 3, ...	r_1 加 1 (指向 x_{18})
→ cmp r1, r2	6, 18, 18, ...	15, 18, ?, ...	1, 2, 3, ...	→(18=18, 顺序执行)
→ mov i0, r0	6, 18, 18, ...	15, 18, 6, ...	1, 2, 3, ...	将 r_0 的 6 存入 x_{10}

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>

这个求和固然写的很好，但是我们又有一个问题，比方说下一次我们想写代码来求一块连续内存的均值，那么我们就需要再写一遍类似的代码，只是在最后加一个除法指令。这显然非常不经济实惠，因此需要把这个求和的过程给抽象出来，这就是 ^{Function} 函数的概念。函数就是一段可以重复使用的代码块，它可以接受输入，产生输出。想的很好，但是我们要怎么实现呢？

我们先从日常生活经验来理解这么一件事情：你在做作业，突然感觉饿了，于是你拿起手机，打开了某外卖软件，点了一份外卖。这个过程中，你并不需要知道外卖是怎么做的，而在点完外卖后，你放下手机，继续做作业。我们从这个例子中可以得到什么启发呢？首先，原本的语境是做作业，点完外卖后应该要切换回做作业的场景，而不是紧接着打开某视频或者小说软件，这说明你需要记住你原本的工作做到哪里了。其次，你的行为是逐层嵌套的，要先拿起手机才能点外卖，但是点完之后要先退出外卖软件，然后才是放下手机。就像这样两个闭合的括号 (())，你需要先进入外层才能进入内层，反过来要先退出内层才能退出外层。我想这已经足以说明函数应当如何设计了：总的来说要有一个入口和一个出口，而且函数内部的操作应该是封闭的，用完要能够切换回原来的场景。

跳进函数很容易，只需要 `jmp` 到函数的入口执行代码就可以了，但是仔细一想，我们的函数调用完之后要怎么知道该回到哪里呢？这里我就要提到一个之前没有明说的地方，实际上执行程序时我写在每一行的指令都是有编号的，这个编号就是 ^{Program Counter} 程序计数器 ^{Special-Purpose Register} ⁴⁷。与其它的寄存器不同，这个寄存器是有专门用途的，所以称为 ^{General-Purpose Register} 专用目的寄存器，而其它可以随意用作存储的寄存器称为 ^{General-Purpose Register} 通用目的寄存器。在前文中指令左侧画的小三角就是程序计数器的表示，所以“记住”运行到了哪里实际上只需要把程序计数器的值存起来，然后在函数结束后再把它取出来就可以了。

最简单的想法是，再设置一个寄存器专门用来存储要返回的地址，不过这个想法存在一个问题：如果在函数里面再次想要调用其它函数，那么这个寄存器就会被覆盖，也就是说内层函数调用成功并返回了，外层函数却不知道该回到哪了。为此我们发现存储应该是分层的，每进入一个函数就应该有一个新的存储空间，当退出时再把这个存储空间销毁，而且进入和退出的顺序是相反的（这种顺序通常称为 ^{Last In First Out} 后进先出），这就引出了 ^{Stack} 栈。

⁴⁷程序计数器：在实际的 CPU 中，程序计数器是一个寄存器，用来存储待执行的指令地址。

这里画一幅图来说明栈的概念：想象一张有很多个格子的纸条，我们有一支带橡皮的铅笔（下面画一个小的箭头表示这支“笔”），刚开始栈是空的，里面什么也没有存储。一条边界固定，称作栈底，另一条边界线会变化，称作栈顶，两线重合表明栈是空的。

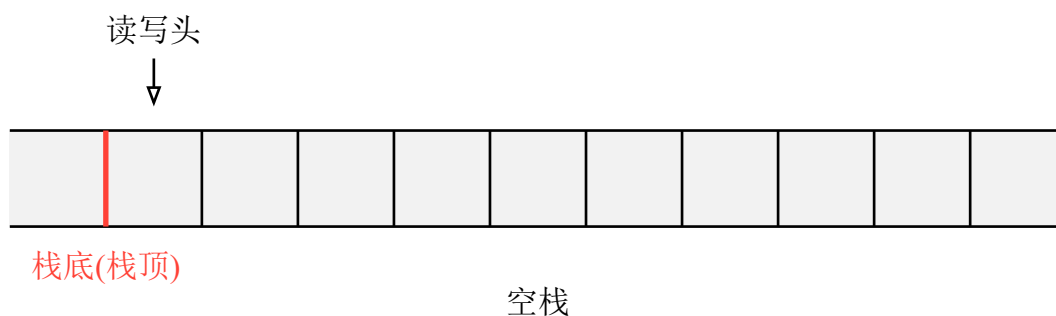


图 19 空栈示意图

当我们向其中加入一个元素时，就把这个元素放在栈顶，同时读写头指向下一个位置。这个过程称为 ^{Push} 压栈，例如上面的空栈加入一个元素后的状态是这样的：

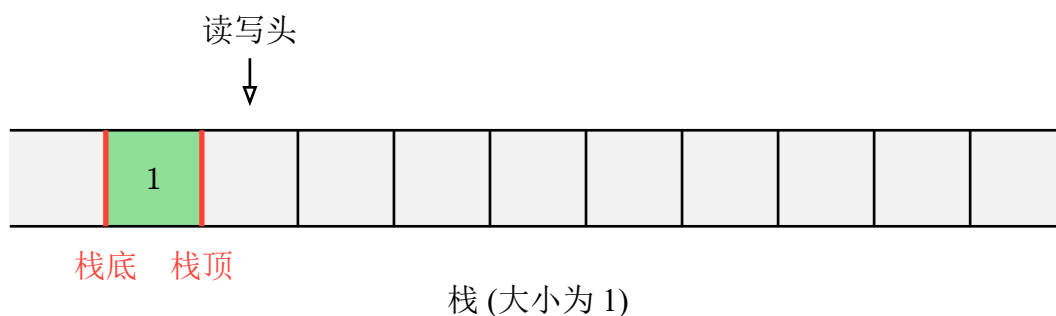


图 20 压栈后的状态（大小为 1）

再加入一个元素呢？这个元素又会被跟着放在栈顶，读写头的位置加 1，即指向下一个位置，就像这样：

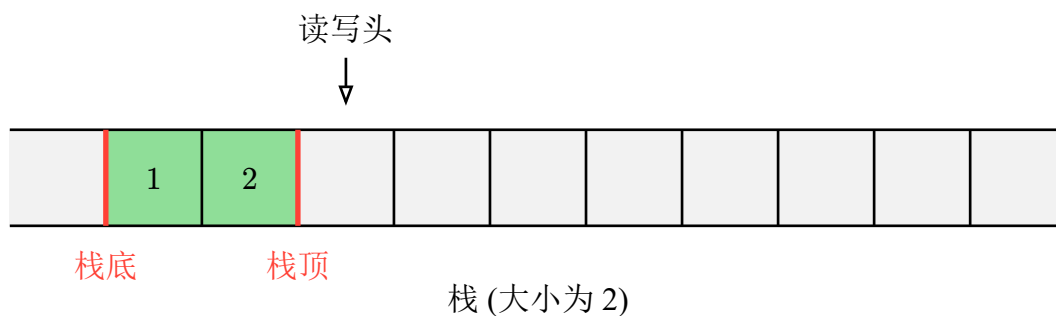


图 21 继续压栈后的状态（大小为 2）

当我们要取出一个元素时，就把栈顶的元素取出，同时读写头向前移动一个单元。这个过程称为 ^{Pop} 弹栈，例如上面的栈弹出一个元素后的状态是这样的：

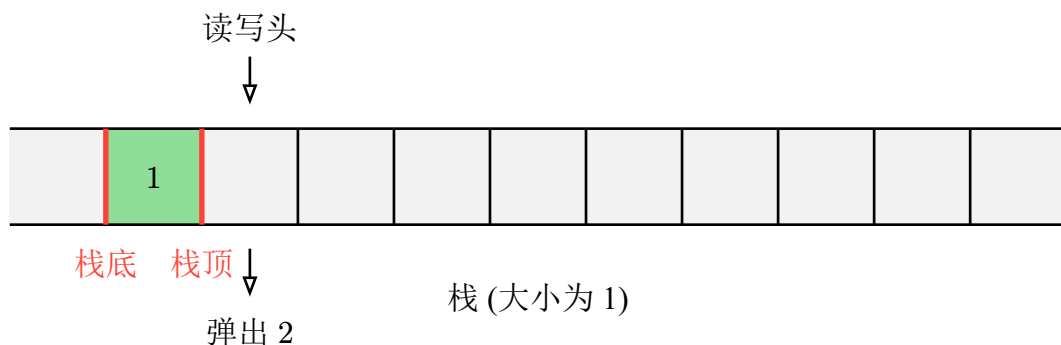


图 22 弹栈后的状态

栈就像一摞盘子，每次放盘子都是放在最上面，取盘子也是从最上面取（我们不讨论一次拿走多个盘子的情况）。只需要知道如何往上放和如何取下来就可以操作了。不过盘子能叠的高度是有限的，正如内存是有限的，但是假使我们的程序没有太深层的函数调用，这里就假设是 80 层⁴⁸，那么我们只需要分配 80 个单位的连续内存空间。对于人类来讲，匹配十几层的括号已经不可思议，80 层更是相当深了⁴⁹。除此之外我们需要一个寄存器来存储栈顶的位置，其称为 Stack Pointer 栈指针。于是我们大手一挥，把 8 号位置作为 Stack Pointer 栈顶指针，用一个别名 `sp` 代表它。又把 20 到 99 号内存分配给栈。如果暂且不考虑调用层数太深的问题。加下来函数调用要怎么样呢？

首当其冲的是把当前的下一条指令地址压到栈顶，接下来是把栈顶指针加 1，然后再 `jmp` 到函数的入口⁵⁰。在函数结束时，我们需要先恢复栈顶指针，再把栈顶我们事先存的下一条地址弹出来，最后再 `jmp` 到这个地址。

看起来大概像这样，初始时栈中可能已经有了一些内容，我们把下一条指令的地址写入栈，栈顶右移。

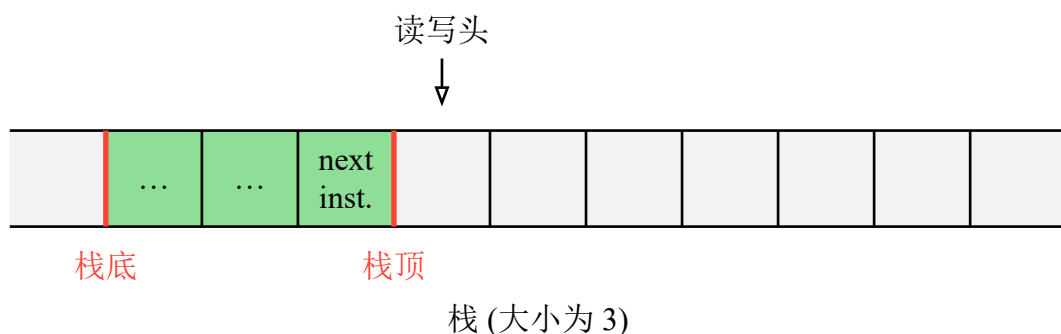


图 23 函数调用时的栈状态

⁴⁸80：这个数字有其历史原因，早期计算机终端通常只有 80 列，因此 80 个字符以内成为了 Linux 编码的规范，这个规范延续到了很多语言的编程风格建议之中，成为一种约定俗成。这里限制深度 80 意味着如果使用“标准”的列宽，一行能写下所有的左括号。

⁴⁹相当深：相对大部分应用程序确实是这样的，但是对于一些特殊的部分，例如搜索、嵌套 Callback 回调、Ray Tracing 光路追踪 等，完全可能达到成百上千层。

⁵⁰`jmp`：本质上读者可能已经发现了，`jmp` 实际上就是把某个值写入程序寄存器，这样一来 CPU 就会跳转到这个地址执行了。

等函数执行完要返回到原先的位置时，我们先把栈顶指针左移，再把栈顶的下一条指令地址取出来，最后 `jmp` 到这个地址。

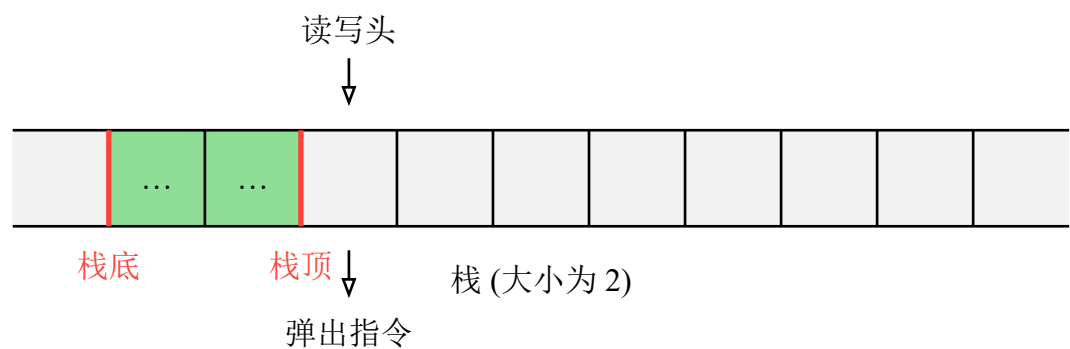


图 24 函数返回时的栈状态

另外，在前面的例子中，起始、终止地址、写入位置等参数是通过手动指定的 8, 9, 10 位置来传递的，但是我们显然不想为每一个函数都手动指定参数要放在哪里，这很麻烦，需要一个清晰、明确的规则来传递参数⁵¹。此处设定一个比较简单的规则：用 9 号位置专门储存函数得到的结果，起别名 `ans`。同时做这样一个限制：函数最多有 4 个参数⁵²，把 10-13 位置用作参数存储，给它们分别起名 `arg0` 到 `arg3`。那么在我们做出了看起来还算满意的内存分配后，目前看来大概是这么分布的⁵³：

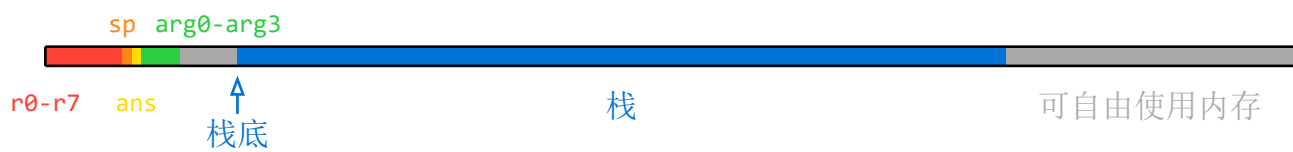


图 25 设想中的一种内存分配方式

这样我们就可以以“函数调用”的方式求 101-103 号位置均值并存储到 100 位置了，不过我们需要在每次调用函数前后都要写一段代码来维护栈，如果我们手写一切代码来维护大概是这样的：

先是 `sum` 函数：

```
sum:
    mov r0, #0 ; 将 0 写入 0 号寄存器
    mov r1, arg0 ; 将 arg0 的值赋给 1 号寄存器
    mov r2, arg1 ; 将 arg1 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
```

⁵¹规则：x86, x86-64 Linux, x86-64 Windows, ARM 各有各的传法。
⁵²4 个参数：这个限制是为了简化问题，实际计算机参数传递中对于多出的部分会用到栈，但是这里不允许使用栈传递，4 这个数字是按照 x86-64 Windows 可用的寄存器参数传递来的。
⁵³说明：实际的计算机中栈通常是从后向前增长的，与此处不同，注意区分。

```

add r1, #1      ; 1 号寄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 ans, r0      ; 将 0 号寄存器的值存入 ans
sub sp, #1      ; 栈顶指针减 1
jmp [sp]        ; 跳转到栈顶指向的指令地址

```

再是 mean 函数:

```

mean:
    mov [sp], label1; 将 label1 的地址存入栈顶
    add sp, #1      ; 栈顶指针加 1
    jmp sum         ; 跳转到 sum 函数, 无需改变参数

label1:
    sub arg2, arg1 ; 将 arg2 减去 arg1 得到求和的长度
    div ans, arg2  ; 将 ans 除以 arg2 得到均值
    sub sp, #1     ; 栈顶指针减 1
    jmp [sp]       ; 跳转到栈顶指向的指令地址

```

最后是主程序:

```

main:
    mov 101, #1      ; 将 1 存入 101 号内存
    mov 102, #2      ; 将 2 存入 102 号内存
    mov 103, #3      ; 将 3 存入 103 号内存
    mov arg0, #101   ; 将起始地址 101 存入 arg0 (含)
    mov arg1, #104   ; 将终止地址 104 存入 arg1 (不含)

    mov [sp], label2; 将 label2 的地址存入栈顶
    add sp, #1      ; 栈顶指针加 1
    jmp mean        ; 跳转到 mean 函数
label2:
    mov 100, ans     ; 将 ans 的值存入 100 号内存

```

最终我们总体的程序结构是这样的:

```

jmp main
sum: ...

```

```
mean: ...
main: ...
```

读者可以一步步地思考，假设 `sp` 最开始存储了空的栈顶 `20`，并体会它是如何通过精确的操作完成函数调用的。不过随之而来的我们发现每次调用函数前起手都要写这样一段

```
...           ; 前面的代码
mov [sp], label1; 将 label1 的地址存入栈顶
add sp, #1     ; 栈顶指针加 1
jmp func       ; 跳转到 func 函数
label:         ; 为了后续继续执行添加标签
...           ; 原本的后续代码
```

同样在函数结束时又要写一段

```
...           ; 函数内部
sub sp, #1     ; 栈顶指针减 1
jmp [sp]       ; 跳转到栈顶指向的指令地址，函数结束
```

实在是太麻烦了，显然属于重复性的劳动，于是我们从中提炼出 ^{Call}调用 和 ^{Return}返回 的指令。既然这样，就给了简化写法的空间：定义一个指令 `call func`，它自动完成函数开始时压栈、栈顶移动和跳转到函数的操作。再定义 `ret`，它自动完成栈顶回退、跳转到栈顶指向的地址的操作。把这个过程抽象出来之后，我们的程序就变成了这样：

```
jmp main
sum:
    mov r0, #0 ; 将 0 写入 0 号寄存器
    mov r1, arg0 ; 将 arg0 的值赋给 1 号寄存器
    mov r2, arg1 ; 将 arg1 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 号寄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 ans, r0 ; 将 0 号寄存器的值存入 ans
    ret
mean:
    call sum ; 直接把 arg0 和 arg1 传给 sum
    sub arg2, arg1 ; 将 arg2 减去 arg1 得到求和的长度
```

```


div



ans,



arg2



;



将 sum 得到的 ans 除以 arg2 得到均值



ret



main:



mov



101,



#1



;



将 1 存入 101 号内存



mov



102,



#2



;



将 2 存入 102 号内存



mov



103,



#3



;



将 3 存入 103 号内存



mov



arg0,



#101



;



将起始地址 101 存入 arg0 (含)



mov



arg1,



#104



;



将终止地址 104 存入 arg1 (不含)



call



mean



;



直接把 arg0, arg1, arg2 传给 mean



mov



100,



ans



;



将 mean 得到的 ans 存入 100 号内存


```

但是其实抽象远没有结束，还可以进一步提炼出更精简的代码。我们感觉直接操作指令的方式过于野蛮了，但是我们可以写一个简单的文本替换程序来帮我们从较为简洁的代码生成这些指令。我们假设有这样一个程序，能完成如下的替换：

原文本	替换后	说明
"..."	; ...	注释
;	(换行)	换行
123	#123	立即数
x123	123	内存地址
a += b	add a, b	加法
a -= b	sub a, b	减法
a *= b	mul a, b	乘法
a /= b	div a, b	除法
a = b	mov a, b	赋值
return a	mov ans, a ret	返回
if a != b jmp addr	cmp a, b jne addr	不等跳转
if a == b jmp addr	cmp a, b je addr	等于跳转
if a < b jmp addr	cmp a, b jl addr	小于跳转
do {...} while (a != b)	loop_i: ... cmp a, b jne loop_i	循环，其中 i 为自动分配的编号

表 8 文本替换规则表

这样事情会变得简单很多，我们只需要写出一个更加易于理解的代码，再让这个文本替换工具把它翻译成可以被执行的指令就可以了。例如对于前面的例子，我们可以写出这样的代码（为了美观起见给每一行加上分号结尾）：

```
jmp main

sum:
    r0 = 0; r1 = arg0; r2 = arg1;
    do {
        r0 += [r1];
        r1 += 1;
    }
    while (r1 != r2);
    return r0;

mean:
    call sum;
    arg2 -= arg1;
    ans /= arg2;
    ret;

main:
    x101 = 1; x102 = 2; x103 = 3; arg0 = 101; arg1 = 104;
    call mean;
    x100 = ans;
```

Compiler

事实上这个“文本替换程序”就已经是编译器的雏形了，而这里抽象出来的更适合人类阅读的代码再往下走进行逐层的抽象就会一步步地走向高级语言。熟悉 C 语言的读者应该已经发现此处我是参考了 C 语言的语法来设计的，实际上 C 语言的设计就是为了更好地表达汇编语言而诞生，最初始的 C 语言差不多每句代码都对应一句汇编，很多现代 C 语言的特性是后续才慢慢添加的。这里我们就不再深入这个话题了，感兴趣的读者可以自行了解编译器的工作原理。

既然我们看完了极简的 CPU 模型，我想还是稍微提几句现代的 CPU 为好。往下（硬件）看，现代的 CPU 指令显然比这个模型丰富的多，而且通常是多核的，每个核都有自己的寄存器、程序计数器、栈指针等。寄存器也不只是看起来的几个，而是加了另一层的抽象，使用寄存器重命名技术将物理寄存器映射到逻辑寄存器。为了提升速度，在 CPU 和内存中间又插入了多级的 ^{Cache} 缓存，这样 CPU 不用每次都去内存中读取数据，而是先读

取缓存，如果缓存中没有再去内存中读取。在执行时，CPU 以其优化技术会对指令进行 Out-of-Order Execution

乱序执行，而并不一定严格地按照代码的顺序。流水线执行、SIMD 单指令多数据、Branch Prediction 分支预测等技术也都是现代 CPU 的特色，它们将大大提升 CPU 的性能。

往上（软件）看，从 C 语言或者其它基础语言为基石构建的高级语言拥有了越来越丰富的特色，基于它们开发的各种库与框架也让程序员写出更加可靠、高效而又可复用的代码，可以更加专注于业务逻辑的开发，这些环环相扣构成了一张严密的逻辑网络。不同的 Programming Paradigm 编程范式、Design Pattern 设计模式、Software Architecture 软件架构等概念也让程序员们在开发时有更多的选择，而这些都是在计算机科学的基础上发展起来的。

在这里我再稍微点一下这一章的题目。本章的题目毕竟是逻辑亦数据，但是这里逻辑和数据似乎是分离开的：代码是代码，数字是数字，它们的关系又体现在哪里呢？其实这里为了理解，呈现的已经是一个经过抽象的版本，在 CPU 看来，每一条指令其实也是若干个字节。在内存中，指令和数据是混杂在一起的，只不过 CPU 会根据指令的不同来对待它们。在更高层次上，数据也可以是代码，代码也可以是数据。

不过上面的流程主要还是针对 CPU 的，但如果转头看向 GPU，会看到一个全然不同的世界，这些内容放在本章的拓展阅读中

推荐阅读

前面我们提到的只是一个极其简单的模型，如果你想了解现代 CPU 的寄存器分布，可以看：

CPU 寄存器到底有多大？《深入理解计算机系统》说大概有几百字节，可是汇编课上却说理论上有 64kb - 北极的回答 - 知乎

<https://www.zhihu.com/question/28611947/answer/55987003>

如果读者有一定的基础并想了解现代 CPU 的一些优化技术，可以看：

分支预测，uOP，乱序执行 - XZiar 的文章 - 知乎

<https://zhuanlan.zhihu.com/p/349758402>

如果读者具有坚实的 C 语言和数据结构基础，想自己试图写一个简单的编译器，可以试着看看这个项目：

rswier/c4: C in four functions

<https://github.com/rswier/c4>

3 为什么是神经网络

3.1 神经网络：一个大的函数

相比于 ^{神经网络}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 ^{通用人工智能}Artificial General Intelligence⁵⁴ 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：[表 9](#)。

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

表 9 机器学习方法分类

⁵⁴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。

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。

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 ^{编码器} Encoder 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 ^{解码器} Decoder 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。

我把模型抽象成五个要素：^{输入} Input、^{输出} Output、^{模型结构} Architecture、^{损失函数} Loss Function 和 ^{优化算法} Optimizer。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。我认为画一个图串起来（如[图 26](#)）是最直观的方式。

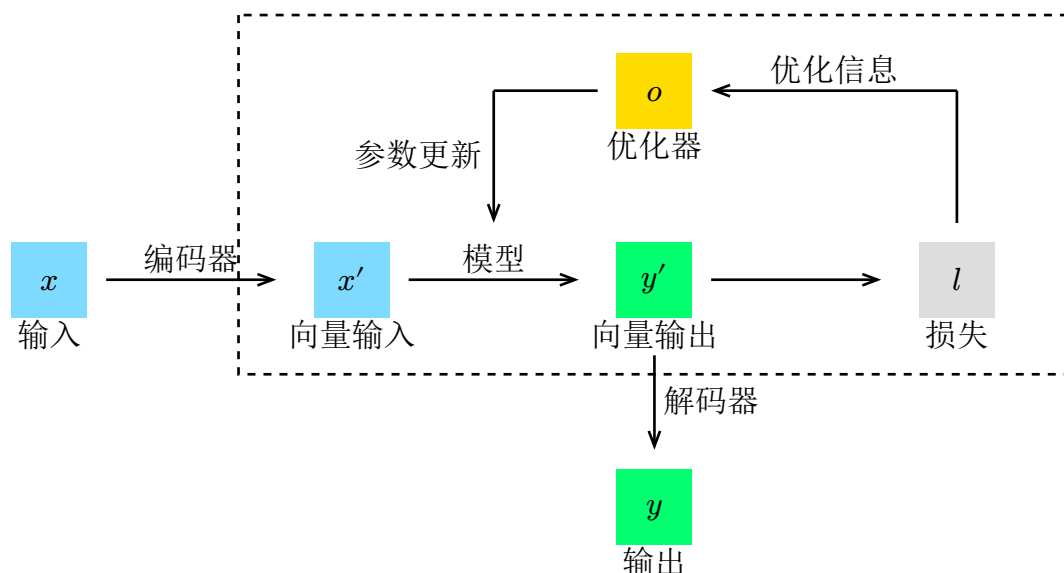


图 26 神经网络的五个要素及其关系

从输入到输出再到损失的过程通常称为 Forward Propagation，而从损失到参数的更新过程则称为 Backward Propagation。而这中间的模型结构常常由矩阵运算与一些 Activation Function 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⁵⁵，而从分析上看，它的输出变化光滑，容易求导⁵⁶。

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

⁵⁵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。

⁵⁶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。

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>

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⁵⁷，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 $f: \mathbb{R} \rightarrow \mathbb{R}$ ，输入为 $x = [x_1, x_2, \dots, x_n]$ ，则输出为 $y = [f(x_1), f(x_2), \dots, f(x_n)]$ 。

现在使用最多的激活函数是 Rectified Linear Unit (ReLU)，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：

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

图像是这样的：

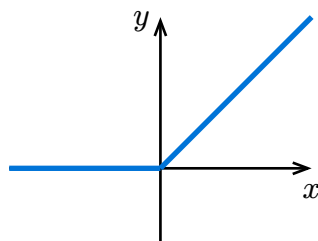


图 27 ReLU 函数图像

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 ^{整流器} Rectifier 一样，把负值截断了。

而它的导数也非常简单：

$$\frac{d}{dx} \text{ReLU}(x) = \begin{cases} 1, & x > 0 \\ 0, & x < 0 \end{cases}$$

⁵⁷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。

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⁵⁸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 1⁵⁹。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：

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

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

$$w_1 = \begin{bmatrix} 1 \\ -1 \end{bmatrix}, \quad b_1 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \quad w_2 = [1 \ 1], \quad b_2 = 0$$

遂可以写成 $y = w_2 \text{ReLU}(w_1 x + b_1) + b_2$ 。我认为，把这件事作为一个 toy case⁶⁰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：

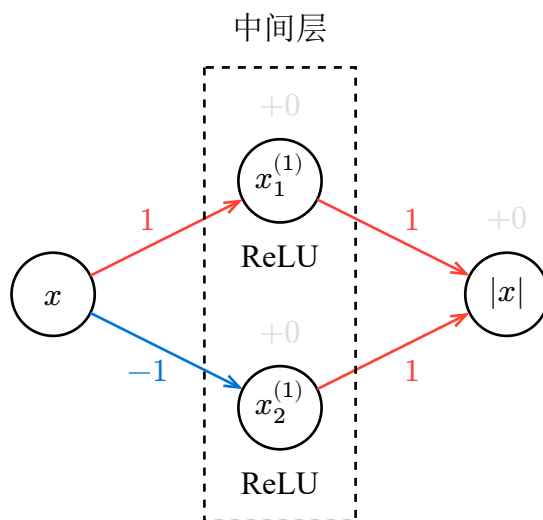


图 28 神经网络表示 $|x|$ 的结构

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。

⁵⁸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。

⁵⁹0 处的导数：PyTorch 通常选择 0

⁶⁰toy 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。

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

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

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 $\text{ReLU}(wx + b)$ 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⁶¹。

不过使用一点小小的技巧，可以把 x_1 本身写成 $x_1 = \text{ReLU}(x_1) - \text{ReLU}(-x_1)$ ，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

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

这一式子中的三个分量提出来，便可以得到

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

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

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

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

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。

⁶¹并不优雅：与之对比，在深度神经网络中通常会引入看起来有些像这里的 跳连接 **Shortcut Connection** 结构，由此引出 残差网络 **Residual Network**⁶² 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。

⁶²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。

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

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

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

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

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⁶³，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 分段线性函数 Piecewise Linear Function，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。

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 $f(x)$ ：

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

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。

⁶³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。

另外再提一嘴其它的激活函数。Sigmoid 函数⁶⁴是一个 S 型函数，定义为

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

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 Gradient⁶⁵的问题，后来逐渐被 ReLU 函数取代，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 高斯误差线性单元 Gaussian Error Linear Unit (GELU)，指数线性单元 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

⁶⁴Sigmoid 函数：Sigmoid 来源于拉丁语，得名于其类似小写字母 sigma 的 S 形状。

⁶⁵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。

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

6 神经网络的泛化

7 神经网络的可解释性

8 神经网络的应用

9 神经网络的未来

10 结语