如何理解神经网络——信息量、压缩与智能

gtj

2025 年 3 月 13 日

目录

1 从函数拟合开始

1.1 最简单的规律——简单线性回归

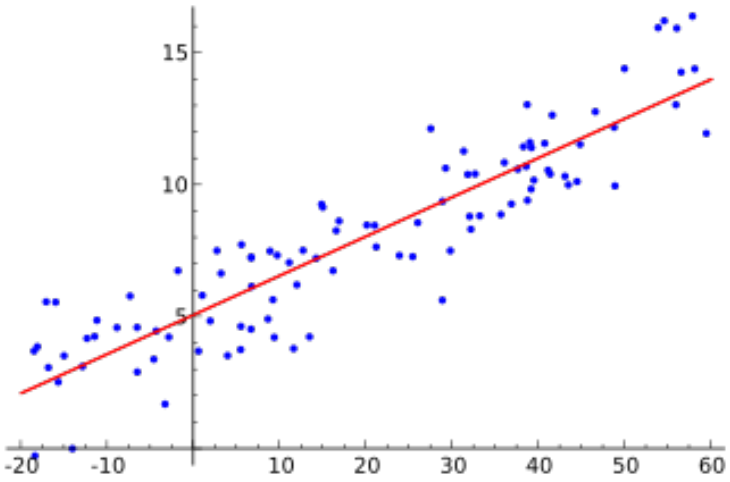


图 1: 线性回归示意图

图源: Wikipedia

虽然 Linear Regression 线性回归 的名字叫做“ Regression 回归 ”，但是事实上我更喜欢叫做 Linear Fitting 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 Marginal Benefit 边际效益 [1]，

[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

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打到了几环。换言之，两个相差的部分才是关键的，残差 (Residual) 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，度量它偏离原点的程度就是向量的模 (Norm)。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得到“平均”的残差平方。习惯上称之为均方误差 (Mean Squared Error) (MSE)：

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

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 甚至为负实在是不合理，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

[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 $r = y - \hat{y}$ 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。

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 的点贡献提升到了 $4^2/20 = 80\%$，更加凸显出了 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？

  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。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：

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

  这时再计算均方误差呢，变成了 $n+2$ 个点，一种我们设想的“改善的”(Refined)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：

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

  再对照着上面的列表看一看，$3/(n+2)$ 不正好表明在大小为 $n+2$ 的列表中 r_1 出现了 3 此吗？频次就这样和权重(Weight)(系数)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刚好是整数。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，这就引出了加权误差(Weighted Error)，大权重表示更重要，略微改写一下公式得到：

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

[4]残差的大小：严谨的说称作绝对误差(Absolute Error)

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 $1/n$。在重要性可变时，加权均方误差无疑提供了一种更加“通用”的 Measurement Metrics 度量方式 。

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的情况。我们要做的是找到一组最优的 Optimal Parameter 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——最小化 Minimize Loss 损失(误差)。形式上，公式会这么写：

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

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，但这本是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：

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

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

$$\mathbf{x}^0 = [1, 1, \ldots, 1]$$
$$\mathbf{x}^1 = [x_1, x_2, \ldots, x_n]$$

并记输出 $\mathbf{y} = [y_1, y_2, \ldots, y_n]$，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。

[5]Argument: 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。

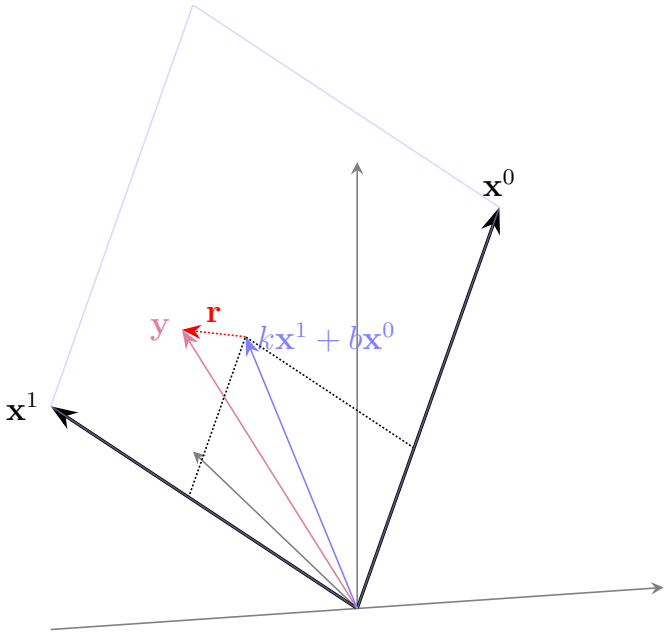


图 2: 从几何的角度看残差

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

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

Mean
即均值。

看斜率之前先看看 Variance 方差 和 Covariance 协方差，方差[6]的表达式是

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

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

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

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

它把方差中的平方项换成了 x 和 x 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 $k = \mathrm{Cov}(\mathbf{x},\mathbf{y})/\mathrm{Var}(\mathbf{x})$，如果把协方差看作一种乘法[7]，那么 $k = (\mathbf{x}\cdot\mathbf{y})/(\mathbf{x}\cdot\mathbf{x})$ 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

- 如果你想了解“回归”与“Least Squares 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
  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

[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

1.2 多项式拟合

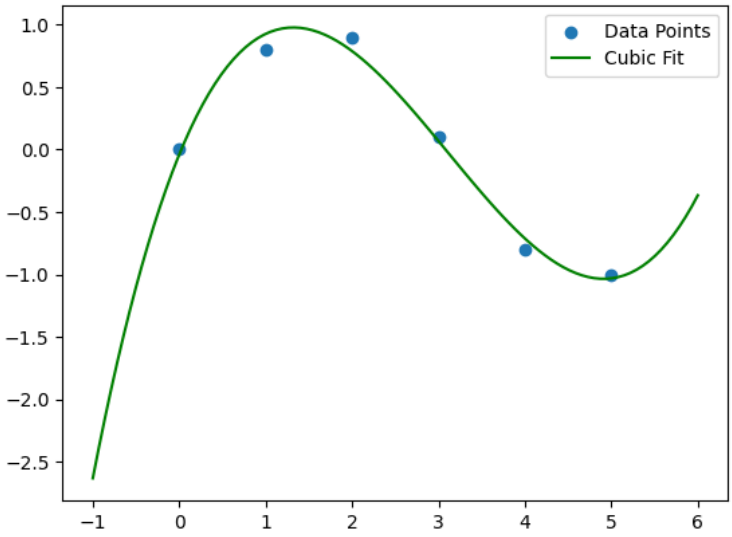


图 3: 多项式拟合示意图（图为 3 次拟合）
图源: GeeksforGeeks

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 $a_0+a_1x+\cdots+a_mx^m$ [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

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

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 $\mathbf{r}$ 的表达式入手

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

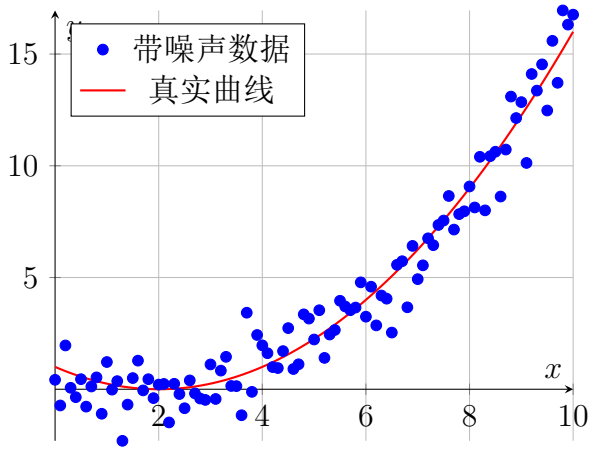
[8]记号说明：虽然习惯上幂次从大到小排列，但是为了下标和幂次的统一性，所以这里选择从常数项到最高次项排列

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

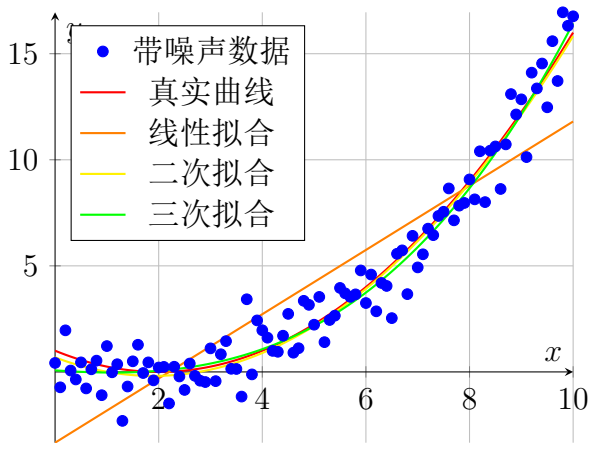
拿做题打个比方：使用直线拟合明显不线性的数据是方法错了，只能说是没完全学会。但是用接近数据量的参数来拟合数据，留给它的空间都够把结果“背下来”了，捕捉到了数据的细节，却忽略了数据背后的规律，化成了一种只知道背答案的自我感动。在几道例题上能做到滴水不漏，但是一遇到新题就束手无策。

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

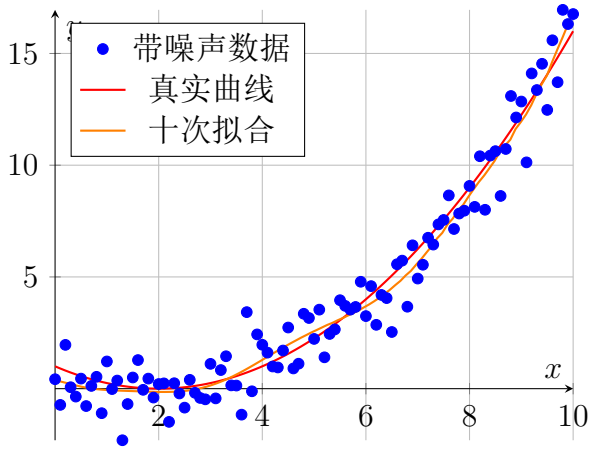


那么现在我们来试试用不同次数的多项式拟合这个数据集。不难看出线性拟合的线与数据点还是相差不少，因为它没能提供可以制造数据“弯曲”形状的项，它没能捕捉到数据更加复杂的趋势，这种现象称为欠拟合（Underfitting）。2 次曲线的效果几乎和真实曲线一样，即使提升到 3 次也没有太明显的改变，它们拟合的效果都还算好。

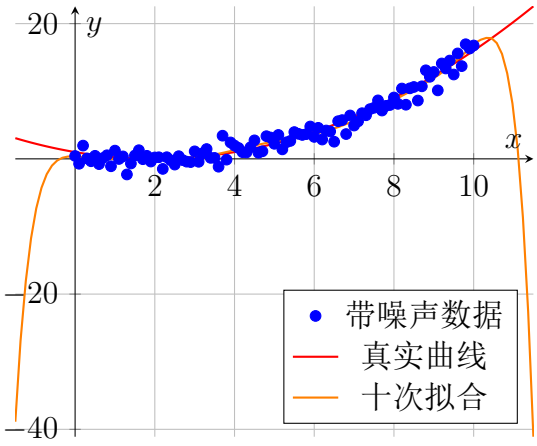
[9] $\mathcal{N}(0,1)$：表示一个服从 标准正态分布 的变量，均值为 0，方差为 1



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（Generalization）[10]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 $\mathbf{x}$ 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 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，这就是正则化（Regularization）的思路。我们在待优化的函数上加上一个惩罚项（Penalty Term），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

[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。

[12]Loss：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。

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。

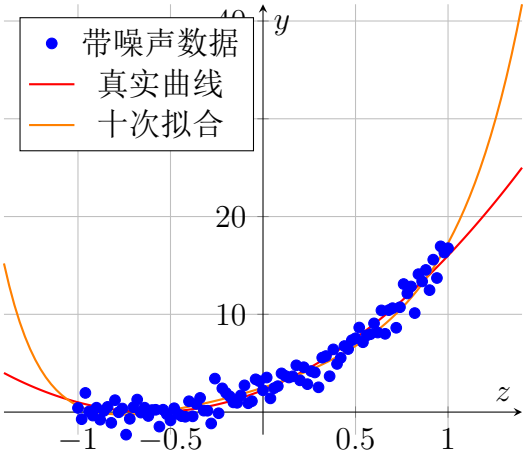
不过即使我们使用了相同的系数 λ，也仍然有一些技巧可以帮我们把高次项压下去。我们发现一路下来导致问题的根源都是 x 的高次项即使在系数很小时也会导致数值爆炸。但是如果把 x 放到 $[-1, 1]$ 的闭区间内呢？这样即使是 x^{10} 项，x 的 10 次方也不会超过 1，系数再怎么大，至少在小范围内也不会导致数值爆炸，这样一来正则化项才能发挥其约束作用。

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 Normalization 归一化 [13]。

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

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

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

[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。

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]参数通常称为超参数（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。

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

[14]先验：在观测到数据之前，我们已经了解了一些数据特征。

1.3 高维的线性拟合

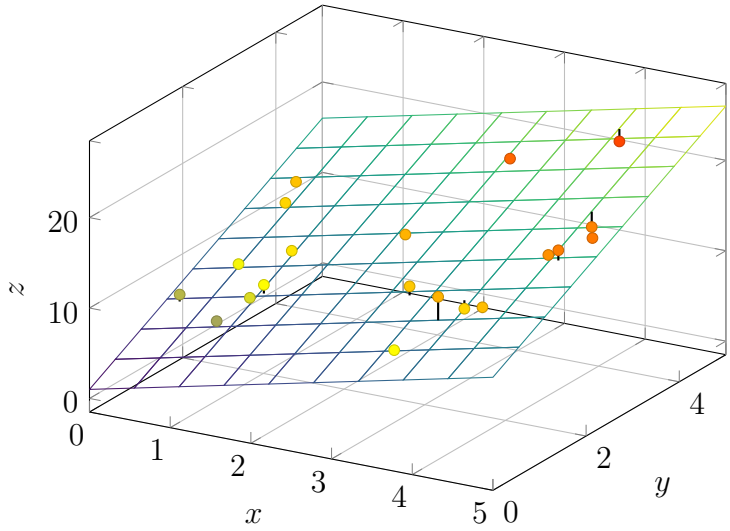


图 4: 高维线性拟合示例

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 多元线性回归 (Multiple Linear Regression)。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：

$$\hat{y} = w_1 x_1 + w_2 x_2 + \cdots + w_d x_d + b$$

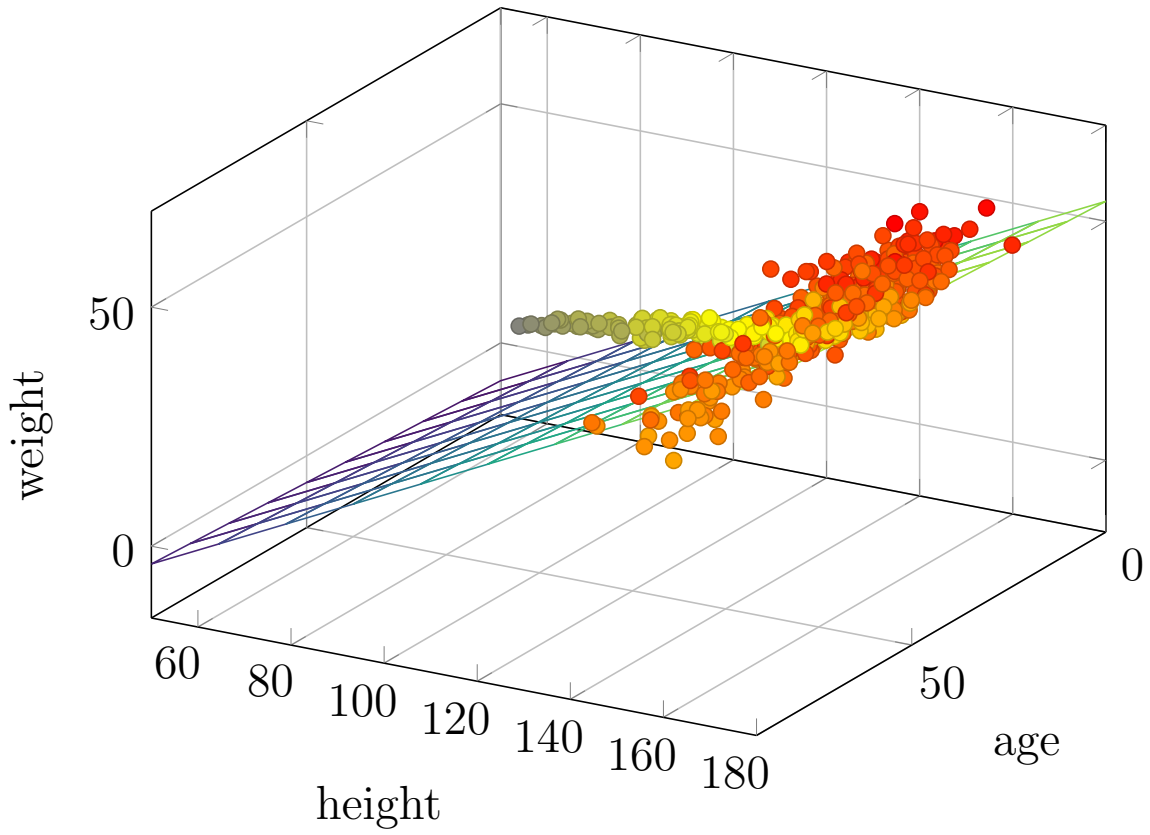
同样地，优化的目标仍然是最小化均方误差，即对 n 个数据点令

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

通过最小化误差得到 $w = [w_1, w_2, \cdots, w_d]$ 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 超平面 (Hyperplane) 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 权重 (weight)，b 表示 偏置 (bias)，即常数项，d 表示的是空间的 维度 (dimension)。



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 $0.04676645038926784 \cdot \text{age} + 0.47766688346191755 \cdot \text{height} - 31.805656676953056 = \hat{\text{weight}}$，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]

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

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

$$\mathbf{r} = \mathbf{y} - (a_0\mathbf{x}^0 + a_1\mathbf{x}^1 + a_2\mathbf{x}^2 + \cdots + a_m\mathbf{x}^m)$$

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 $\mathbf{x}_{:1}, \mathbf{x}_{:2}, \ldots, \mathbf{x}_{:n}$ 分别为 $\mathbf{x}$ 的幂次组成的向量 $\mathbf{x}^1, \mathbf{x}^2, \ldots, \mathbf{x}^d$，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。

[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。

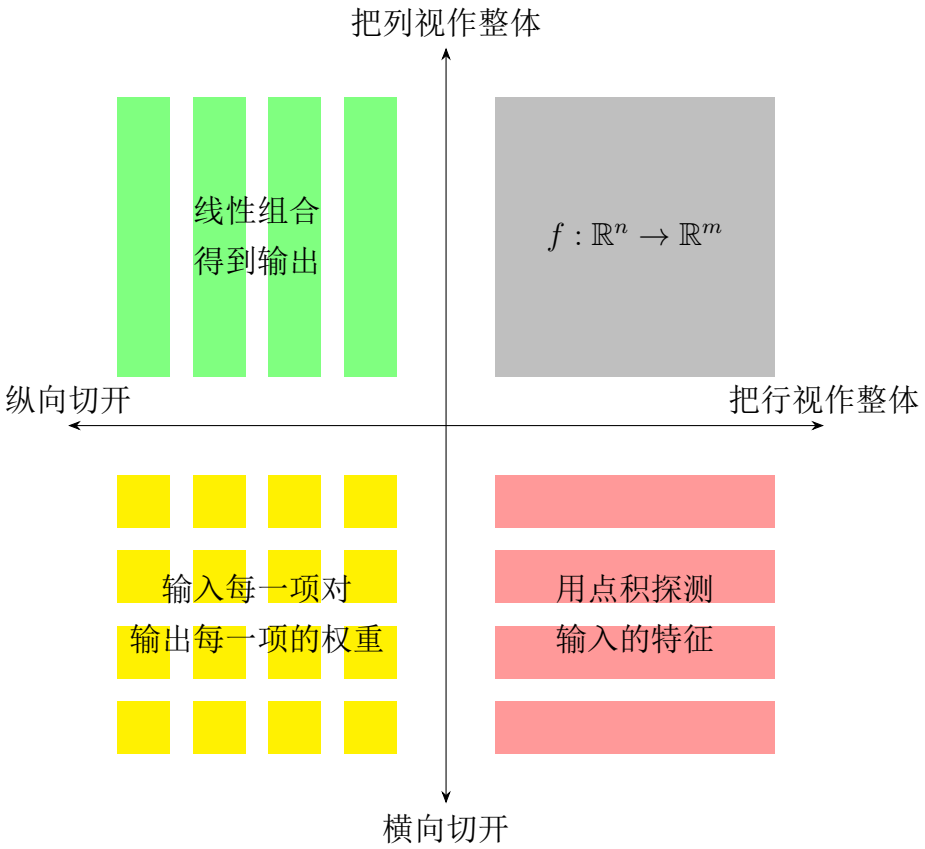
[17]记号说明：在公式中我特意将常数项放到了最前面，这是为了让它和多项式拟合的形式保持一致。

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》(*Linear Algebra for Everyone*) 来的，我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。



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

[19]几种视角：整体 Overall 解读、按行 Row-wise 解读、按列 Column-wise 解读、按元素 Element-wise 解读。

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，全程以映射(Map)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。

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 $\mathbb{R}^n \to \mathbb{R}^m$ 的线性映射。使用矩阵的第一个重要目的就是把一个线性映射“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：

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

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：

$$\begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_m \end{bmatrix}$$

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

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

诚然，每一行都是一个齐次的(Homogeneous)[23]线性函数，它的形式也只能是这种 x 的加权和(Weighted Sum)，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。

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

[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函数(Function)。

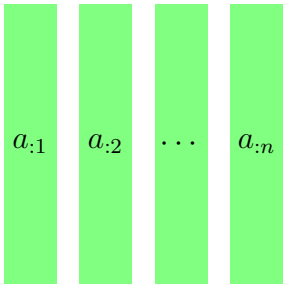
[22]严格地说 a 是行向量，x 是列向量，这中间在数学上有一些差别，是矩阵乘法而不是点积。但是在计算机存储中，因为都是一维数组，从实用的角度并不需要纠结于此，这种记号混用(Abuse of Notation)就见怪不怪了。

[23]齐次：指不带偏置(常数)项。

我们可以把矩阵的每一行看作一个特征检测器(Feature Detector)，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(通常称为特征响应(Feature Response))。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。

这时如果考虑怎么样的输入可以接近预期的输出呢？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：



这样看来，矩阵的乘法也可以写成

$$Ax = \begin{bmatrix} a_{:1} & a_{:2} & \cdots & a_{:n} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

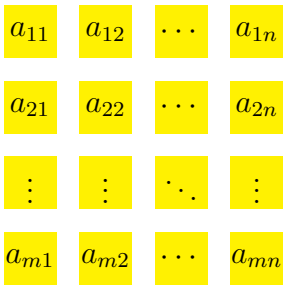
$$= x_1 a_{:1} + x_2 a_{:2} + \cdots + x_n a_{:n}$$

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。

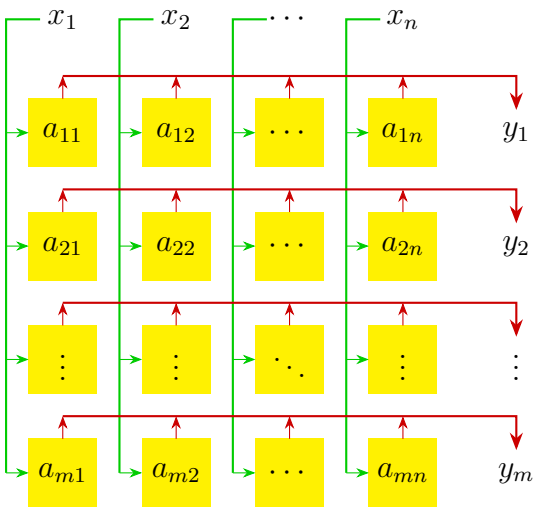
最后一种角度则带有更为浓重的解构(Distruction)[24]色彩，如果把矩阵看成一个数表，作为一个填了数字的 $m \times n$ 矩形：

[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。



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的 建构 (Construction) [25]，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。

让我们看看下面这个示意图：



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，最后在行地址线上的所有块累积起来得到输出的每个分量。

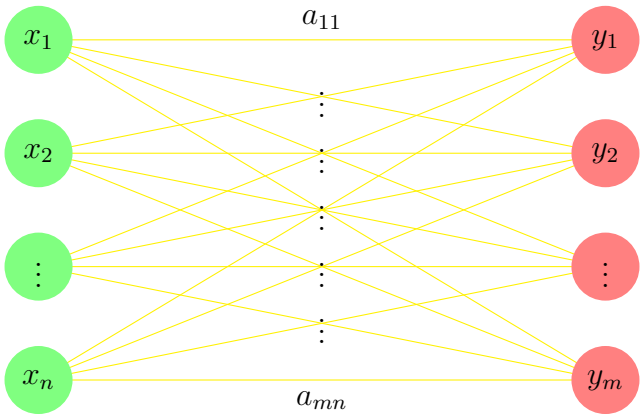
这个图还有另外一个很常见的呈现形式，把它看成一个无偏置的 线性层 (Linear Layer) [27]，我们通常是这样绘制的:

---

[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 行/列地址选通信号 (Row/Column Address Strobe)。

[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 $b = 0$。



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。

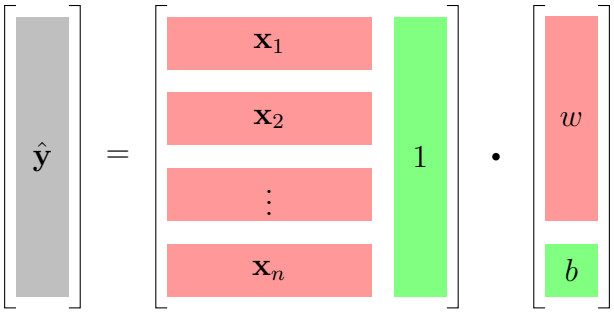
花了一些篇幅来复习线性代数，是时候回到多元线性拟合的问题上了。不过这次可以使用矩阵的语言来描述这个问题了。假设我们有 n 组 d 维的 x 的取值，它们组成了一个 $n \times d$ 的矩阵 $\mathbf{x}$[28]。我们的目标是找到一个 d 维的 w，使得数据集上 $x \cdot w$ 尽可能接近 y，现在 $\hat{y}$ 的表达式用点积的语言可以简洁地写为 $\hat{y} = x \cdot w + b$ 这种形式。

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

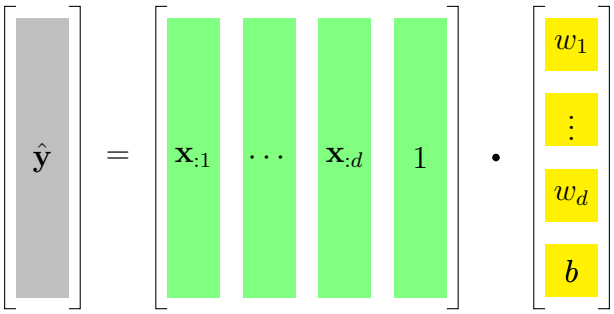
$$\begin{aligned}\hat{y} &= x \cdot w + b \\ &= x_1 w_1 + x_2 w_2 + \cdots + x_d w_d + b \\ &= \begin{bmatrix} x_1 & x_2 & \cdots & x_d & 1 \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_d \\ b \end{bmatrix}\end{aligned}$$

[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。

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 $\hat{\mathbf{y}} = \tilde{\mathbf{x}}\tilde{w}$。如果要说这个 $\tilde{\mathbf{x}}$ 是什么，它就是把 $\mathbf{x}$ 的每一行拼上一个 1：



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 $\mathbf{x}_{:1}, \mathbf{x}_{:2}, \cdots, \mathbf{x}_{:d}$ 与 $\mathbf{x}^0$ 的线性组合来贴近 $\mathbf{y}$。



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 $\{1, 2, \cdots, n\}$ 上，换言之，记 $\mathbf{y}(1) = y_1, \mathbf{y}(2) = y_2, \cdots, \mathbf{y}(n) = y_n$，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？ - 丿小奇迹丨的回答 - 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） - 学弱猹的文章 - 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，先来做一个约定，我们将 0 视作 False 假，1 视作 True 真[29]。先来看看几种最简单的逻辑运算。

1. Not 非（数学写法：$\neg$，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

   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

[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 5: 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：

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

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

$$\begin{array}{rr} & \ldots\ 1\ \ldots \\ + & \ldots_1\, 0_1 \ldots \\ \hline & \cdots\ 1\ \ldots \end{array}$$

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' 可以表示为

$$s = (x \text{ xor } y) \text{ xor } c$$

$$c' = (x \wedge y) \vee (c \wedge (x \text{ xor } y))$$

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

[31]所有的运算：这里指的是 图灵完备性 (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。

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

- 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。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[35]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 架构(Architecture) 与 指令集(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”[36]上遵照着一份“指南”[37]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：

[35]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*)

[36]草稿纸：比喻计算机的内存(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。

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

- `mov`：**mov**e 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**, **sub**tract, **mul**tiply, **div**ide 的缩写，加减乘除。
- `jmp`, `je`, `jne`, `jl`：**j**ump, **j**ump if **e**qual, **j**ump if **n**ot **e**qual, **j**ump if **l**ess 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`：**cmp**are 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 立即数值 (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[38]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[39]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：

指令	$[x_1, x_2, x_3, \cdots]$
(initial)	$[x, y, \_, \ldots]$
▷ mov 2, 0	$[x, y, x, \ldots]$
▷ mov 0, 1	$[y, y, x, \ldots]$
▷ mov 1, 2	$[y, x, x, \ldots]$

[38] 内存 `0`：按照计算机中的习惯，计数从 0 开始。

[39] 你先别管它怎么运行起来的。

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。这种情况一般称为副作用（Side Effect）[40]，但似乎不太可能既不修改其它内存，又交换数值[41]。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[42]，我们就此“发明”了寄存器（Register）[43]。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[44]。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
```

[40]副作用：指令运行的过程中对其他地方产生的影响。

[41]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。

[42]临时储存：可以理解为一种草稿纸，内容可以随时丢弃

[43]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。

[44]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。

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[45]，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	$[r_0, r_1, r_2, \cdots,$	$x_8, x_9, x_{10}, \cdots,$	$x_{15}, x_{16}, x_{17}, \cdots]$	
(initial)	$[\_, \_, \_, \cdots,$	$15, 18, \_, \cdots,$	$1, 2, 3, 4, \cdots]$	
▷ mov r0, #0	$[0, \_, \_, \cdots,$	$15, 18, \_, \cdots,$	$1, 2, 3, 4, \cdots]$	向 r_0 写入 0
▷ mov r1, 8	$[0, 15, \_, \cdots,$	$15, 18, \_, \cdots,$	$1, 2, 3, 4, \cdots]$	将 x_8 的 15 赋给 r_1
▷ mov r2, 9	$[0, 15, 18, \cdots,$	$15, 18, \_, \cdots,$	$1, 2, 3, 4, \cdots]$	将 x_9 的 18 赋给 r_2
▷ add r0, [r1]	$[1, 15, 18, \cdots,$	$15, 18, \_, \cdots,$	$1, 2, 3, 4, \cdots]$	$r_1 = 15$, 取 $x_{15} = 1$ 加到 r_0
▷ add r1, #1	$[1, 16, 18, \cdots,$	$15, 18, \_, \cdots,$	$1, 2, 3, 4, \cdots]$	r_1 加 1 (指向 x_{16})
▷ cmp r1, r2	$[1, 16, 18, \cdots,$	$15, 18, \_, \cdots,$	$1, 2, 3, 4, \cdots]$	$\to (16 \neq 18,$ 跳回 loop)
▷ add r0, [r1]	$[3, 16, 18, \cdots,$	$15, 18, \_, \cdots,$	$1, 2, 3, 4, \cdots]$	$r_1 = 16$, 取 $x_{16} = 2$ 加到 r_0
▷ add r1, #1	$[3, 17, 18, \cdots,$	$15, 18, \_, \cdots,$	$1, 2, 3, 4, \cdots]$	r_1 加 1 (指向 x_{17})
▷ cmp r1, r2	$[3, 17, 18, \cdots,$	$15, 18, \_, \cdots,$	$1, 2, 3, 4, \cdots]$	$\to (17 \neq 18,$ 跳回 loop)
▷ add r0, [r1]	$[6, 17, 18, \cdots,$	$15, 18, \_, \cdots,$	$1, 2, 3, 4, \cdots]$	$r_1 = 17$, 取 $x_{17} = 3$ 加到 r_0
▷ add r1, #1	$[6, 18, 18, \cdots,$	$15, 18, \_, \cdots,$	$1, 2, 3, 4, \cdots]$	r_1 加 1 (指向 x_{18})
▷ cmp r1, r2	$[6, 18, 18, \cdots,$	$15, 18, \_, \cdots,$	$1, 2, 3, 4, \cdots]$	$\to (18 = 18,$ 顺序执行)
▷ mov 10, r0	$[6, 18, 18, \cdots,$	$15, 18, 6, \cdots,$	$1, 2, 3, 4, \cdots]$	将 r_0 的 6 存入 x_{10}

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 函数 (Function) 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

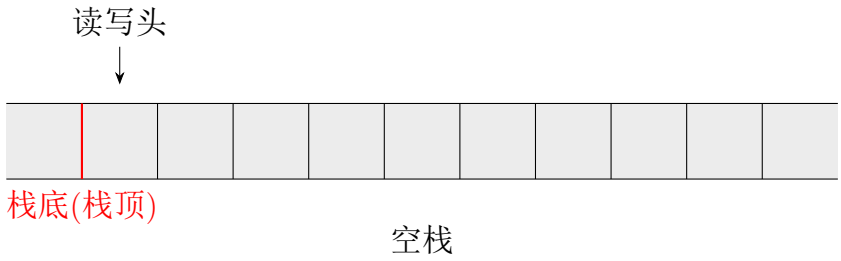
[45]别名：例如在本例中，它指代 `add r0, [r1]` 所在的行

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，这个编号就是程序计数器 (Program Counter)[46]。与其它的寄存器不同，这个寄存器是有专门用途的，所以称为专用目的寄存器 (Special-Purpose Register)，而其它可以随意用作存储的寄存器称为通用目的寄存器 (General-Purpose Register)。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。

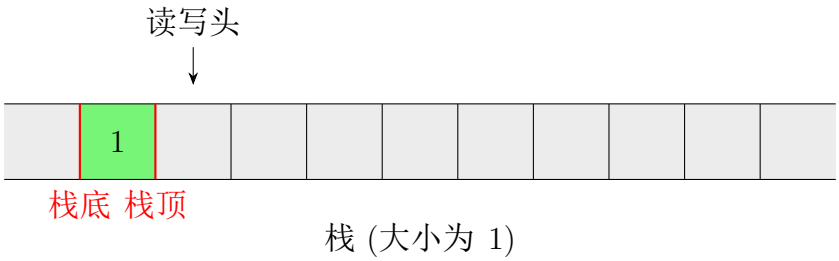
最简单的想法是，再设置一个寄存器专门用来存储要返回的地址，不过这个想法存在一个问题：如果在函数里面再次想要调用其它函数，那么这个寄存器就会被覆盖，也就是说内层函数调用成功并返回了，外层函数却不知道该回到哪了。为此我们发现存储应该是分层的，每进入一个函数就应该有一个新的存储空间，当退出时再把这个存储空间销毁，而且进入和退出的顺序是相反的（这种顺序通常称为后进先出 (Last In First Out)），这就引出了栈 (Stack)。

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。



空栈

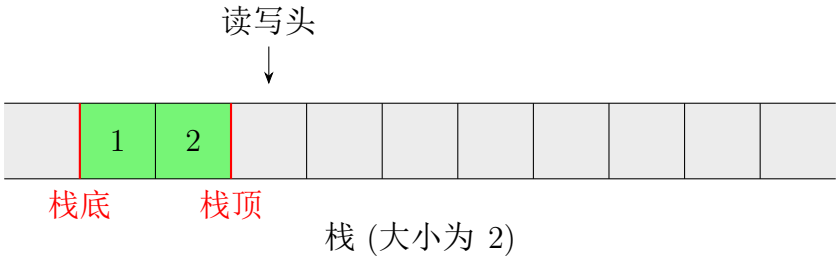
当我们向其中加入一个元素时，就把这个元素放在栈顶，同时读写头指向下一个位置。这个过程称为压栈 (Push)，例如上面的空栈加入一个元素后的状态是这样的：



栈 (大小为 1)

[46]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。

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：



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。这个过程称为弹栈(Pop)，例如上面的栈弹出一个元素后的状态是这样的：

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[47]，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[48]。除此之外我们需要一个寄存器来存储栈顶的位置，其称为栈指针(Stack Pointer)。于是我们大手一挥，把 8 号位置作为栈顶指针(Stack Pointer)，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[49]。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

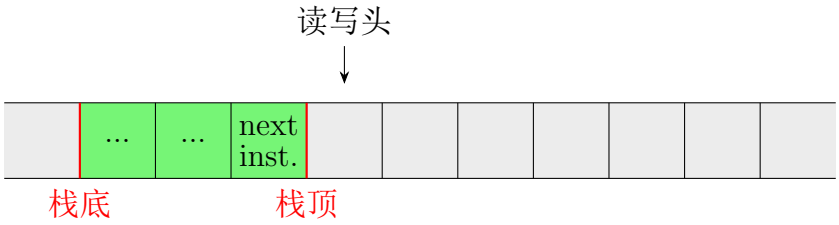
[47] 80：这个数字有其历史原因，早期计算机终端通常只有 80 列，因此 80 个字符以内成为了 Linux 编码的规范，这个规范延续到了很多语言的编程风格建议之中，成为一种约定俗成。这里限制深度 80 意味着如果使用“标准”的列宽，一行能写下所有的左括号。

[48] 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、嵌套回调(Callback)、光路追踪(Ray Tracing)等，完全可能达到成百上千层。

[49] `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。

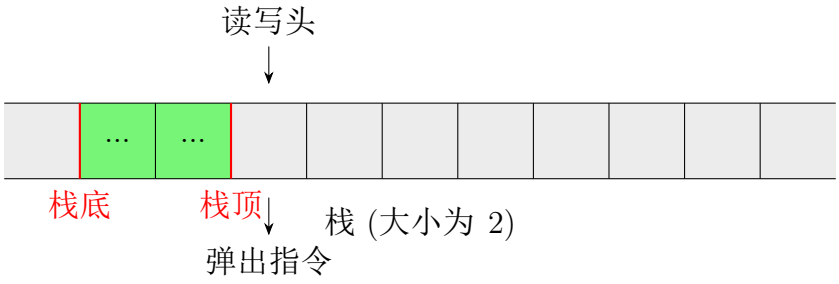
下一条地址弹出来，最后再 `jmp` 到这个地址。

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。



栈 (大小为 3)

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。



栈 (大小为 2)

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[50]。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[51]，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[52]：

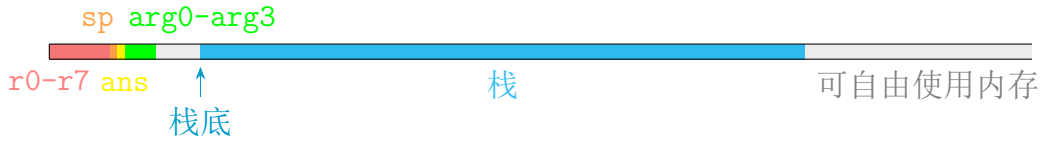


图 6: 设想中的一种内存分配方式

[50]规则：x86, x86-64 Linux, x86-64 Windows, ARM 各有各的传法。

[51]4 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。

[52]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。

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
    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 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调用(Call)和返回(Return)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 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

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;
```

事实上这个“文本替换程序”就已经是编译器（Compiler）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缓存（Cache），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、单指令多数据 SIMD、分支预测 Branch Prediction 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了更多的选择，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。

但是接下来转头看向 GPU，我们将会看到一个全然不同的世界。

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

2.3 那么，GPU 呢？

GPU，全称 图形处理器 (Graphics Processing Unit)，是一种专门用来处理图形计算的处理器。打游戏的同学或许知道，为了打开高画质高刷新的游戏，需要一块强大的显卡，显卡的核心就是 GPU。都是执行指令，CPU 和 GPU 有什么区别呢？

在回答这个问题前我想有必要提出一个更为根本性的问题：渲染图形提出了怎样的特别的需求？让我们举一个最简单的例子：在平面上绘制一个圆。我们且不讨论怎么把圆加载到屏幕上，且就假设我们只需要把每个点都计算出来，或者更为具体地：在一个高 1080 宽 1920 的数组中计算出每个像素的颜色[53]，按照惯例，黑色是 0，白色是 255。看起来很简单，我在这里用一份 C 语言代码来实现这个功能：

```c
unsigned char img[1080][1920];

void draw_circle(int x, int y, int r) {
    for (int i = 0; i < 1080; i++) {
        for (int j = 0; j < 1920; j++) {
            if ((i - x) * (i - x) + (j - y) * (j - y) < r * r) {
                img[i][j] = 255;
            }
        }
    }
}
```

原理本身是十分简单的，但是这段代码读下去我们无不发现一件事情：两个循环的次数一乘，单单渲染一个圆就需要 $1080 \times 1920 = 2073600$ 次的计算，考虑到加法、乘法、写入、读取都要时间，这个数字是相当大的。如果要处理的几何体多了，那么一个程序不说跑上 120 FPS [54]的高速刷新了，或许连 30 FPS 都难保证。但是计算的需求本身客观存在，我们无法改变这个事实，那么该怎么办呢？

按照老规矩，我们应该先想想，对于人类来说，“想象一个圆”这个过程是如何发生的。我们显然不是在脑海中一个一个像素地去计算，而是直接画出一条边界，然后成片地填充。与逐像素刷新过去相比，这是“一瞬间”的事情，每个像素同时计算了出来。这就叫做 并行计算 (Parallel Computing)，喷颜料总比一笔笔涂快，为了高效进行并行计算，GPU 诞生了。

[53]方便起见，这里只考虑黑白颜色。

[54]FPS： 帧每秒 (Frame Per Second) 的缩写，也称作帧率，许多游戏都有帧率设置。

有一个虽然不太严谨但是很形象的 比喻：CPU 就像是大学生，有几个核就有几个大学生，他们都很聪明，但是只能做一件事情。GPU 如果有几千个核，就像是几千个小学生，虽然每个人都不太聪明，只能完成简单的任务，但是适合完成大规模的任务。这个比喻虽然不够准确，也有人对这样的比喻提出过 质疑，但是可以让我们对 CPU 和 GPU 有一个直观的认识。一个更为细致的 解释 是 CPU 可以完成细致的分支预测等工作，大大提高了处理了复杂逻辑的能力，然而 GPU 并不适合处理 分支分歧（Branch Diverge），分支语句会让 GPU 的并行计算效率大大降低（见 Nvidia 的 文档）。

总的来讲，CPU 有少量的寄存器和算数逻辑单元，适合处理复杂的逻辑运算。但是 GPU 有一个许多的线程，每个线程都有自己的多个寄存器。同时有多个运算单元[55]来提供大规模的并行计算。这里不多介绍 GPU 的具体架构与底层组成，因为 GPU 的架构是多变的，不同的厂商、不同的型号都有不同的架构，而且 GPU 的架构也在不断地发展。

GPU 上要运行的指令所属的指令集与 CPU 上并不相同，甚至不同型号的 GPU 也有不同的指令集，因此自然地，程序需要适配到自己的 GPU 上。经常玩游戏的同学可能会发现不少游戏在启动时会有一个正在编译着色器（Shader）的过程，或许不少人也好奇过 为什么。实际上这个过程就是把图形渲染的代码编译成 GPU 可以执行的指令，就像不同的语言需要用不同的编译器一样，GPU 驱动中通常会包含一个着色器编译器，用来把着色器翻译成适配具体 GPU，可以被顺利执行的指令。

从实用的角度说，如果只是作为游戏玩家，我们只需要考虑什么显卡算力够用，花钱买卡就可以，但是游戏厂要考虑的就多了。游戏开发者为了渲染出更加逼真的画面需要了解 GPU 的 渲染流水线（Rendering Pipeline），如何借助图形化库与 GPU 交互，以及如何使用着色器来实现特效、如何计算光线追踪。

但是话又说回来，渲染图形、纹理填充这些内容和机器学习又有什么关系呢？本质上是因为机器学习需要大量的矩阵运算，需要大量加法、乘法操作，因此这种重复性的操作也可以交给 GPU 并行处理。随着人工智能的快速发展，GPU 开始加入了专门加速矩阵运算的单元。例如 Nvidia 在 2017 年就向其生产的 GPU 中加入了 张量运算核（Tensor Core）[56]。

作为机器学习的使用者，我们显然也需要了解如何使用 GPU，不过更重要的是如何用好矩阵或张量运算的能力，所以侧重点有些不同。首先，无论出于什么用途，为了让主板识别到 GPU 与之正常交互需要安装 GPU 驱动。当购买预装了 Nvidia 显卡的电脑而言，一般厂家已经事先在 Windows 系统上安装了驱动，免去了手动安装的麻烦。然而当我们自己装机或者安装其它系统时，我们就不得不自己走这个流程。科研和工业环境中常常考虑到开源、高性能等优点常常选择 Ubuntu 系统，然而 在 Ubuntu 上安装 Nvidia 驱动 的各种问题常常被人诟病，配环境本身就相当令人头疼（其实是因为 Linux 内核经常改动，

[55]运算单元：在 Nvidia 的显卡中通常称为 CUDA (Compute Unified Device Architecture) 核心。

[56]张量：深度学习中指的是一个高维的数组。

而到现在仍然很难做内核态程序的兼容性适配）。说句题外话，许多库因为快速的更新迭代，库的依赖关系常常错综复杂，显卡驱动只是其中的一例。因此试图在自己的设备上复现论文中的科研成果时，我们常常感到配好环境几乎是成功的一半。这就是灵活性的另一面，与 Windows 上开箱即用的软件体验是全然不同的。

再说说 Nvidia 显卡的 CUDA Toolkit，类似着色器编译器等内容虽然已经在驱动中了，但是它只是一些特殊的预定义的运算。如果要更精细地调控运算，并从源代码开始编译 CUDA 的程序，特别是用 C/C++ 开发，就需要下载 CUDA Toolkit。因此在配置需要 GPU 加速的项目中，读者常常会发现需要安装 CUDA Toolkit（其版本小于等于驱动对应的 CUDA 版本），其最重要的作用就是提供适用于 GPU 的编译器。适当了解 几个工具之间的关系 对理解配置环境时干了什么是有益的。

这时相信有人会问，平时运行程序时也不需要自己编译，那有没有这样一个程序或者工具可以让我们开箱即用地使用 GPU 完成矩阵运算呢？答案是，有的，Torch 和 TensorFlow 就包装掉了底层的细节，可以方便地完成这些运算，而在科研环境中，Torch 的 Python 接口 PyTorch 的使用尤其普遍。这时自然有一个问题：GPU 的版本不同，预编译的程序怎么适配不同GPU 版本呢？如果我们打开 PyTorch 的官方网页，便会发现，其中有一个选项是 Compute Platform。CUDA 是向后兼容的，也就是说新版本的硬件可以运行旧版本的代码，这意味着我们需要根据自己的硬件支持的 CUDA 版本做出 选择。通常推荐选择 PyTorch 官网上提供的小于等于电脑 CUDA 版本的最新版本。例如，如果在命令行运行 `nvidia-smi` 得到的 CUDA 版本为 12.5，然而 PyTorch 官网页面提供 11.8, 12.4, 12.6 这三个版本，那么在两个满足条件的版本中选择一个更大的，12.4 版本就是推荐安装的版本，把下方的命令，例如：

```
pip3 install torch torchvision torchaudio --index-url \
https://download.pytorch.org/whl/cu124
```

复制到命令行运行即可成功安装。当然如果你的 CUDA 版本刚好在页面上就不需要思考，可以直接安装了。事实上当我们在运行 Torch 并将矩阵调度到 GPU 计算时，它帮我们完成了矩阵数据的传输、把运算任务划分为多个 线程块 (Thread Block) 再交给 GPU 的工作。在 GPU 中的 流式多处理器 (Streaming Multiprocessor) 将线程块以线程束 (Warp) 为单位调度，交给若干个 CUDA 核心与张量运算核共同完成矩阵的计算，再将内容写回显存。而这些都被抽象成了一条条的矩阵运算代码，隐去了底层实现的细节，把它们交给了库。这让我们可以从繁琐的逻辑抽离，可以专心于数据的运算。

2.4 位运算与 Bit-flag

前面讲到逻辑实际上可以用 0 和 1 来表示，而在内存中，储存的基本单元是字节（Byte）。一个字节有 8 位，每一位都可以表示 0 或 1，本质上它可以视作一个长度为 8 的 0-1 数组。而一个 32 位整数同样可以视作一个长度为 32 的 0-1 数组。你可能想问：把一个整数看成一堆的位，这不过是一些文字游戏，有什么用呢？

这里就不得不提之前一直没有提到的一类基本运算：位运算。位运算是一种逐位的运算，它可以对整数的每一个二进制位进行操作，从而在某种程度上一次性地操作了一个整数的多个位。以 `int32` 为例，这就相当于我们一次性地操作了一个大小位 32 的数组。

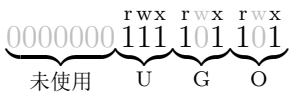
了解信息竞赛的同学或许知道，位运算催生了各种各样的优化技巧，位运算的各种巧妙操作也是大家津津乐道的话题。从 借助位运算实现俄罗斯方块 到 求解 n 皇后问题，再到 各种关于整数的问题，位运算的应用无处不在。

不过这里我们来看一个更为重要的概念：位标志（Bit-flag）。位标志是一种用来表示状态的方法，它可以用一个整数的某一位来表示某种状态。例如，我们可以用一个 32 位整数的第 0 位来表示一个开关是否打开，第 1 位来表示一个设备是否连接，第 2 位来表示一个权限是否开启，等等。这样我们就可以用一个整数来表示多个状态，而且可以同时操作多个状态。位标志在许多系统的底层仍然随处可见。下面仅举出几例：

- **文件权限**：在 Linux 系统中，文件的权限是用一个整数表示的，有 9 个有效的位，分别表示文件所有者的权限、文件所在组的权限、其他用户的权限。例如，如果你使用过 Linux 系统并在某个目录下运行 `ls -l` 命令[57]，你可以看到许多文件夹的详细信息。例如在根目录（即 `/` 目录）中 `home` 这个文件夹的信息可能类似这样：

```
drwxr-xr-x   3 root root    4096 Nov 13 18:02 home
```

  我们仅关注最前面的 `drwxr-xr-x`，这是文件的权限信息，其中第一个字符 `d` 表示这是一个文件夹（Directory）。而后面就是我们说的 9 位的权限信息。`r`, `w`, `x` 分别表示 读（Read），写（Write）和 执行（Execute）的权限。虽然写作 `rwxr-xr-x`，但是实际上它是一个使用了 9 个有效位的二进制数，而并非储存了 9 个字符。就以前面提到的 `drwxr-xr-x` 为例，其作为一个 16 位整数的布局大约是这样的，所有用户可读可打开，但仅所有者可修改：



[57] `ls -l`：其中 ls 命令用于列出（List）文件夹中的文件和子文件夹，选项 `-l` 表明以 长格式（Long Format）输出文件。

这里 U、G、O 分别表示文件所有者(User)、文件所在组(Group)、其他用户(Other)。同样的，设置权限时也是设置这些位。例如如果我们想给文件 `example` 所有者设置读写权限，同时对组和其他用户设置读权限，我们可以使用 `chmod` 命令：

```
chmod u=rw,g=r,o=r example
```

可是这仍然有些麻烦，有没有更简洁紧凑的写法呢？答案是有的，我们可以使用八进制数来表示这些权限。我们把每 3 位看成一个八进制数，分别表示 User、Group、Other 的权限。例如 7 在 2 进制中是 111，表示所有权限都开启，而 5 是 101，表示读和执行权限开启。`rw` 的二进制表示就是 110，对应的八进制数就是 6。而 `r` 的二进制表示就是 100，对应的八进制数就是 4。因此上面的代码就可以简化为

```
chmod 644 example
```

通过这个例子我们已经可以看到了位标志是如何简化操作的，不过让我们再继续看一些例子来更好地理解位标志的应用。

- **嵌入式系统**：在嵌入式系统中，位标志的使用更是无处不在。就以 STM32 系列的芯片[58]为例，它们的许多寄存器是每一位都有特殊含义的，特别是关于 GPIO[59]、串口(Serial)、定时器(Timer)、中断(Interrupt)等模块。例如设置 GPIO 的输入输出状态时，我们可能会这样写来启用 A1 与 A2（如果读者不会 C 语言也可以大概意会一下它的作用）：

```
#include "stm32f4xx.h"

void GPIO_Init(void) {
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA, ENABLE);

    GPIO_InitTypeDef GPIO_InitStructure;
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_1 | GPIO_Pin_2;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;
```

[58]STM32：STMicroelectronics 意法半导体 公司推出的一系列基于ARM Cortex-M内核的 Micro Controll Unit 微控制器 。

[59]GPIO：即 General-Purpose Input/Output 通用型输入输出 ，微控制器通过它与外设连接。

```
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;
    GPIO_Init(GPIOA, &GPIO_InitStructure);
}
```

这其中最关键的部分其实就在于 GPIO_Pin_1 | GPIO_Pin_2。对于不熟悉的人来讲，这看起来像是什么特殊设计出来用于这个情境的符号，甚至有人会认为这是一个特殊的宏定义。然而实际上这只是一个普通的位运算，GPIO_Pin_1 和 GPIO_Pin_2 都是整数，它们的二进制表示分别是 00000010 和 00000100，而 | 就是位或运算，它会把两个数的对应位进行或运算，因此 GPIO_Pin_1 | GPIO_Pin_2 的结果就是 00000110（十进制为 6），这个数就能表示 A1 和 A2 两个位被设置了。

如果我们列一个二进制的竖式，这个运算就变得十分清晰：

$$\begin{array}{r} 00000010 \\ \mid\ 00000100 \\ \hline 00000110 \end{array}$$

当两位中存在 1 时，结果的对应位也是 1，否则为 0。这时可能有人会问，为什么不直接用加法呢？实际上对于这个例子来说，加法和位或运算的结果是一样的，但是使用位运算的意图更加清晰，它表明每一位是独立的。这怎么说呢？例如，让我们想象 2+2=4 这个式子会代表什么。如果我们使用加法，就会得到一个很反直觉的结果，即两个 GPIO_Pin_1 的和是 4，等于 GPIO_Pin_2。它超出原本讨论的引脚，显然与实际不符。而使用位或运算，两个 GPIO_Pin_1 做位或运算的结果仍然为 GPIO_Pin_1，这样就不会“污染”其它位。比如如果我们随便写两个二进制数，01001001 和 00101010，它们做位或运算的结果是 01101011：

$$\begin{array}{r} 01001001 \\ \mid\ 00101010 \\ \hline 01101011 \end{array}$$

这样每一位的变化都是独立的，不会相互影响。更加符合它作为标志，独立编码多个状态的特性。

- **窗口程序：**现代的开发者是幸福的，得益于一系列强大的图形库，他们可以很方便地开发出各种各样的图形界面。然而读者如果恰好学习过 C 语言课程并尝试过使用

Win32 API[60]来编写窗口程序，就会发现窗口样式、窗口消息等的设计都是通过位标志来实现的。例如，我们可以使用 `CreateWindowEx` 函数[61]来创建一个窗口，其中有一个参数叫 `dwStyle`[62]，它就用于设置窗口样式。与现代的许多使用类 CSS[63]的库不同，Win32 API 使用位标志来设置窗口的样式。例如，如果要设置一个窗口为普通窗口，同时具有标题栏、系统菜单、边框可调、具有最大化、最小化按钮等。那么窗口样式应该是这样的：

```
WS_OVERLAPPED | WS_CAPTION | WS_SYSMENU |\
WS_THICKFRAME | WS_MINIMIZEBOX | WS_MAXIMIZEBOX
```

虽然这一长串更常见的叫法是直接作为一个宏整体叫做 `WS_OVERLAPPEDWINDOW`，不过我们还是可以从二进制的角度来看看

```
00000000000000000000000000000000  overlapped
00000000110000000000000000000000  caption
00000000000010000000000000000000  sysmenu
00000000000001000000000000000000  thick frame
00100000000000000000000000000000  minimize box
00000001000000000000000000000000  maximize box
-------------------------------------------------
00100000111001100000000000000000  overlapped window
```

各个位有各自的含义，把它们组合到一起就可以得到一个完整的窗口样式。对于需要高性能的场景，这种设计也让我们可以一次记录多个状态，相比解析字符串构成的样式表速度更快。

---

[60] Win32 API：Application Programming Interface 应用程序编程接口，是 Windows 操作系统的一套接口。

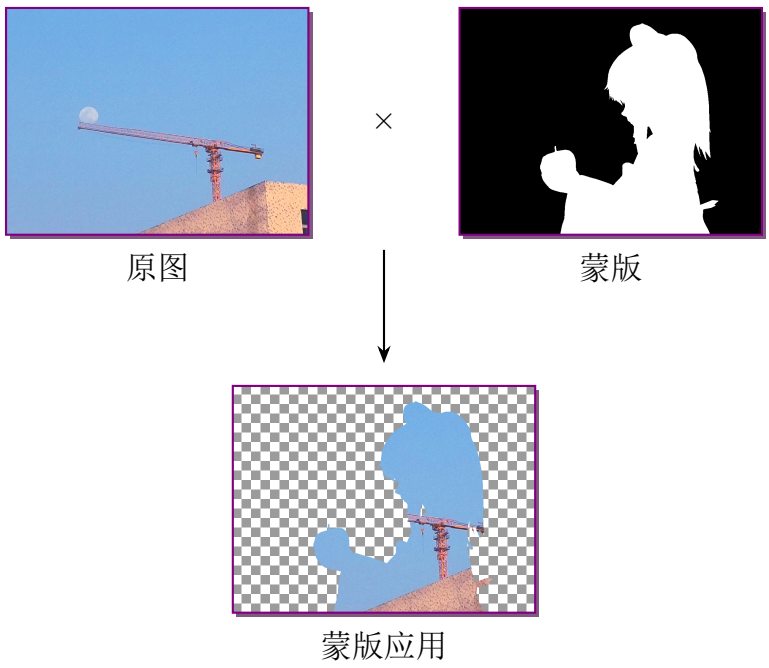
[61] CreateWindowEx：最早的接口是 CreateWindow，但是后来为了支持更多的 Extended 拓展 功能，微软推出了 CreateWindowEx。

[62] dwStyle：Double Word 双字 风格，在 16 位 CPU 的年代，一个基本的 CPU 数据单元是 16 位，当时一个 Word 字 指的是 16 位，因此 32 位的整数就被称为双字。如果打开 `minwindef.h` 仍然可以看到它们的定义。今天的 CPU 大多已经是 64 位的，自然会看起来有些奇怪，但库已经定下来了，这个有些奇怪的名称就一定程度上成为了历史遗留问题。一些现代的语言，如 Rust 则直接使用 `u32` 来表示 32 位无符号整数，这从某种程度上是更清晰的。

[63] CSS：Cascading Style Sheets 层叠样式表，用于 设置网页的样式，例如字体、颜色、布局等。类似的机制也用于许多现代的图形库，如 Qt、GTK 等。

Layer Mask

- **图层蒙版**：从广义的角度看，bit-mask 并没有必要局限于用一个整数的位来表示状态，如果“升级”一下，完全可以使用一个真正的 0-1 数组来记录一个图像每个像素的属性。如果把 0 和 1 视作黑和白，一个黑白图像完全可以视作一个二维数组。黑色表示“不开启”，像素就隐藏了起来，像是被删除了。白色表示“开启”，像素则显现出来，因此只需要在图层蒙版上涂上黑白，就可以实现图层的显示与隐藏。看起来效果像是这样[64]（这里沿用 Photoshop 中的约定，将透明的部分用格子表示）：

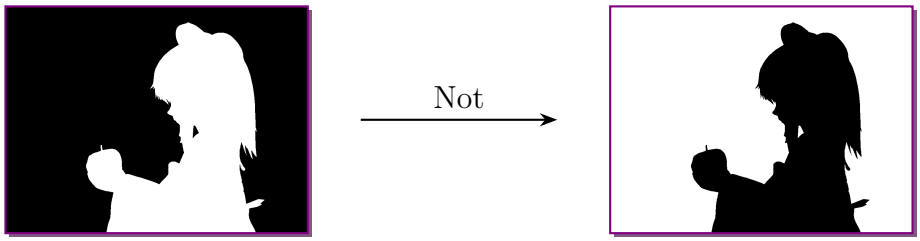


在这里我们可以看到，原图中蒙版为白色的部分显示出来，黑色的部分隐藏了起来。将它叠加到其它的图层上，黑色部分展示的就是下一个图层。

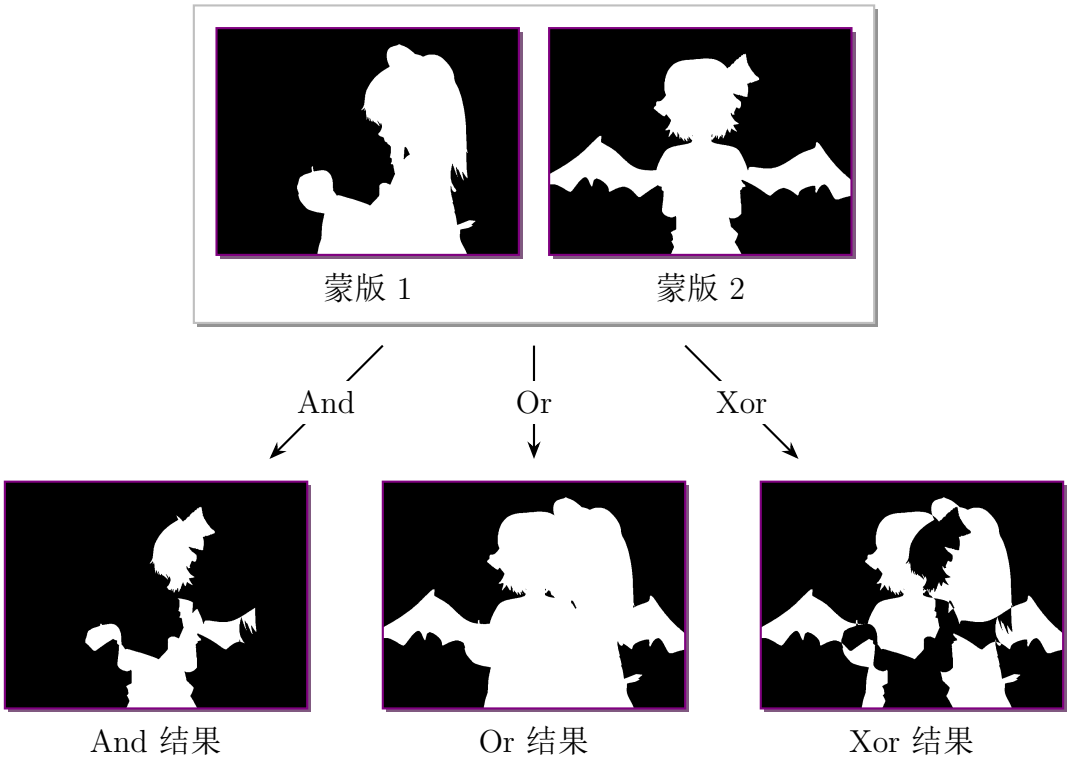
至此我们已经看到了位标志的许多应用，使用这样的表达方式从根本上是因为它用一种统一而简洁的方式将多个状态编码到数字中，各种从逻辑上有所区别的状态就这样打包成了数字。我们不妨再以图像为载体的数组为呈现方式，看看几种位运算是如何工作的。

先回忆一下几种基本运算，一元的 Not 运算是对一个数的每一位取反，即 0 变为 1，1 变为 0。如果在图像的蒙版上看呢？Not 就是将黑色变白，白色变黑，即反色。呈现的效果大约是这样：

[64]说明：此处的蒙版截自视频 【東方】Bad Apple!! ＰＶ【影絵】，有改动。在更早的时候这一视频常被网友用作显示技术测试，有众多二次创作与不同显示设备的呈现。



再看二元的运算：And 运算取的是两个数中较小的值，一旦有 0，And 运算的结果就是 0。Or 运算取的是两个数中较大的值，一旦有 1，Or 运算的结果就是 1。Xor 运算比较两个数是否相同，如果相同，结果就是 0，否则就是 1。因此 And 运算就像是把所有黑色的部分都保留下来，Or 则是把所有白色的部分保留下来，Xor 把两个图像不同的部分标记为白色，在图上的呈现效果大抵是这样的：



可以看见，对于 And 运算，它仅保留了白色的公共区域，处理后的一个像素为白则表示原本两个像素均为白。而对于 Or 运算，它保留了黑色的公共区域，一个像素为黑表示原本两个像素均为黑。而 Xor 运算则保留了两个图像不同的部分，一个像素为白表示原本的像素一黑一白。

但是读者不禁要问，从文件系统讲到嵌入式，从窗口程序再到图层蒙版，这与本书的主题——机器学习有什么关系呢？在第一章我们看到了许多的拟合问题，它们的自变量都

是连续的，但是实际问题中并非这样。

就比如如果我们想通过对社会人群的调查来研究某个现象，有许多是/否类型的问题。例如如果要调查人口与社会流动，可能要对研究对象提出是否结婚、是否是本地户口、是否购房、是否在具有长期的工作等等。这些问题的答案只有两种，是或否。如果使用文字记录所有的信息，并不利于计算机的处理，但是如果约定好每个问题的顺序，将答案看作一个 0-1 数组，这样一些问题答案就可以被量化，更方便计算机的处理。

而同样，在图像分类问题中（例如分类猫、狗、家兔、仓鼠四种宠物），直接输出标签这个字符串似乎不太现实。我们暂且不讨论中间的过程是如何分类出最终结果的，就假设有这样的一个能学习的分类器。可能有人会想，给它们四个标签 $0, 1, 2, 3$，直接让计算机输出一个数字，这样不就可以了吗？但是问题就在于在我们赋予它一个顺序的过程中，我们是否真的能够保证这个顺序是正确的呢？如果作为数字，那么 1 和 2 之间的距离显然小于 0 和 3 之间的距离，但是你似乎很难说狗和家兔之间的距离比猫和仓鼠之间的距离更大，这个话无论怎么说都显得非常荒谬。同样地，就像之前的 GPIO 引脚一样，你显然不能说因为 $2 = (1 + 3)/2$，所以说家兔时狗和仓鼠的平均值。

于是我们得到了这样的结论：这四个类别就不能简单地用一条数轴来容纳，更为保险的方法是分别提出四个问题：这个动物是猫吗？这个动物是狗吗？这个动物是家兔吗？这个动物是仓鼠吗？然后将四个问题的答案放在一起，得到一个长度为 4 的 0-1 数组，如果你仔细检查就会发现四种动物对应的四个向量

$$\text{猫} : (1, 0, 0, 0)$$

$$\text{狗} : (0, 1, 0, 0)$$

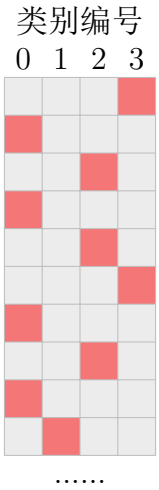
$$\text{家兔} : (0, 0, 1, 0)$$

$$\text{仓鼠} : (0, 0, 0, 1)$$

两两之间的距离是相等的，这种做法的好处就是说我们并没有给它预设一个可能有问题的顺序，不同的顺序只不过是交换了问题的顺序而已。而且如果读者还记得线性代数中线性相关的概念，就会很容易地发现这四个向量是线性无关的，不会出现某个动物是其它动物的组合这样的情况。

这样我们就可以用 4 维的向量来表示这四种动物，这种编码方式在后续的机器学习中非常常见，通常称为 独热编码 (One-Hot Encoding)。由此可以看见，关于逻辑的判断本质上也可以转化成若干个问题的答案，以 0-1 来编码，这其实是一个相当自然、合理的过程。

另外假设我们有这样的一些图片，每张图片有一个类别标签，不同数字表示不同类。这些图片的标签依次为 $3, 0, 2, 0, 2, 3, 0, 2, 0, 1, \cdots$。按照独热编码，即有一个被标为 1 的维度，其它维度都标为 0。图中我们用灰色来标记 0，使用红色来标记 1，每个图像的标签作为一行，那么这个数组看起来是这样的：



还记得我们在第一章的行列对偶视角吗？我们知道从行看过去，它是一张张图像的标签，但是从列看过去，每个列都是一个类别，标明了哪些对象属于这个类别。这也意味着你可以方便地提取出具有某个特征的所有对象，在许多程序处理用户数据时实际上是以列储存的，这种方法在处理非常多的特征时很有效，这一算法称作 Bitmap Algorithm 位图算法 。不过这里我们并不打算就位图算法深入展开，只是提示读者回忆一下矩阵的行、列之间的关系。

至此，我们大致理清了逻辑与数据的关系，也许读者会发现，逻辑的判断本质上就是对数据的处理，而数据的处理也可以转化为逻辑的判断。这样的思想在现代的机器学习中推导了极致，不过按照惯例，我们在此回顾一下这一章的内容。

- 从基本的逻辑 0-1 开始引入了逻辑门，它们是逻辑运算的基础，可以用来构建复杂的逻辑关系，物理上的电路组合造就了计算机的基础。
- 人类将任务分解为一系列较为简单的指令，让 CPU 可以跟着执行。这些指令本身只是简单的运算和调准，但是足以实现复杂的功能，构建起各式各样的程序。
- 通过使用许多寄存器来记录程序状态，我们实现了程序的抽象，能够又好有精准地控制程序的运行，同时一些常见的指令组合给了我们抽象的空间，提炼出这些关系后便发明出了高级语言。
- 现代 CPU 的架构实际上非常复杂，但是其目的从根本上看就是为了更快地方式，但是仍然确定性[65]地执行指令。
- GPU 给我们提供了高效的并行计算能力，使得我们可以更快地处理大量的数据，有安装 CUDA 工具需求的读者可以参阅与 GPU 相关的这一节大致理清几种工具的关系，或许对快速地配好环境是有益的。

[65]确定性：这是个伏笔，强调这一点是因为神经网络的训练过程有许多的非确定性因素，与之形成对比。

- 位标志是一种非常高效的编码方式，它可以将多个状态编码到一个数字中，使得我们可以更加高效地处理这些状态。
- 从更广阔的意义上，许多问题都可以看作对若干个问题的答案，以 0-1 数组的形式来编码，独热编码为编码相互独立的概念提供了一种非常自然的方式。

推荐阅读

- 如果你想要预习机器学习中 mask 的作用，可以参考：
  深度学习中的*mask*到底是什么意思？ - 随时学丫的回答 - 知乎
  https://www.zhihu.com/question/320615749/answer/1080485410
- 如果你想要预习一下独热编码，可以参考：
  深入浅出 *one-hot* - 董董灿是个攻城狮的文章 - 知乎
  https://zhuanlan.zhihu.com/p/634296763

10 杂谈

10.1 矩阵式研究——场景与模型的排列组合

10.2 AI圈的常见行话

10.3 AI——生产力还是毁灭？

10.4 新人类与自由意志？