

如何理解神经网络——信息量、压缩与智能

gtj

2025 年 5 月 4 日

目录

1 从函数拟合开始	3
1.1 最简单的规律——简单线性回归	3
1.2 多项式拟合	9
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
2.3 位运算与 Bit-flag	40
3 为什么是神经网络	52
3.1 神经网络：一个大的函数	52
3.2 激活函数与非线性	56
3.3 神经网络的训练	64
3.4 如果没人教你怎么做	77
4 泛化性：一个矛盾	92
4.1 过拟合与欠拟合	92
4.2 网络的大小好像小于训练数据？哪来的泛化性	92
4.3 训练好像被卡住了——香农极限	92
5 你能猜到一句话接下来要说什么？	92
5.1 什么是“废话”？	92
5.2 jpeg虽然有损，但为什么说是极其成功的？	92

5.3 熵与压缩	92
5.4 马尔可夫链	92
6 压缩即智能	93
6.1 你是如何看出对面的人心情怎么样的?	93
6.2 压缩的极限——区分能力的边界	93
6.3 从母语词汇看对事物的认识	93
6.4 压缩的本质	93
7 潜空间：更适合机器人的编码方式	93
7.1 潜空间是什么?	93
7.2 高维空间的维数灾难	93
7.3 “空空”的空间的另一面——维数远不是储存能力的极限	93
8 但是，代价是什么：可解释性的地狱	93
8.1 想想什么是“解释”？	93
8.2 神经网络不能很好地被解释	93
9 再论网络结构	93
9.1 全连接网络	93
9.2 循环神经网络	93
9.3 卷积神经网络	93
9.4 深度学习与残差链接	93
9.5 transformer	93
9.6 自编码器与扩散模型	93
9.7 仿人的架构——专家模型	93
10 杂谈	94
10.1 矩阵式研究——场景与模型的排列组合	94
10.2 AI圈的常见行话	94
10.3 AI——生产力还是毁灭？	94
10.4 新人类与自由意志？	94

1 从函数拟合开始

1.1 最简单的规律——简单线性回归

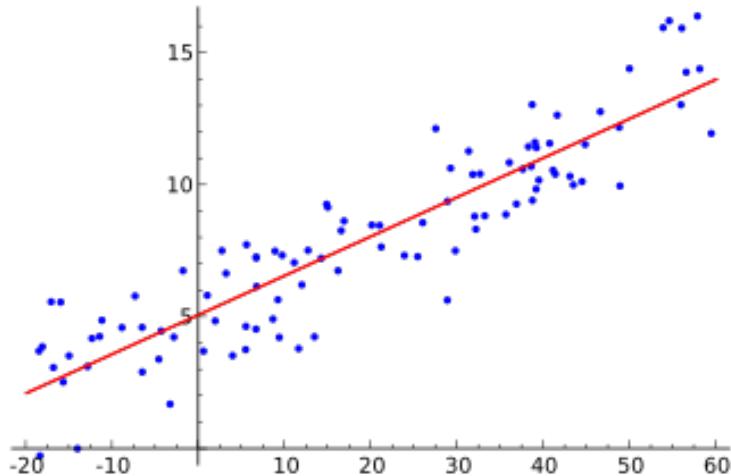


图 1: 线性回归示意图

图源: [Wikipedia](#)

虽然 “线性回归” 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 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，^{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，度量它偏离原点的程度就是向量的 ^{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得到“平均”的残差平方。习惯上称之为 ^{Mean Squared Error} 均方误差 (MSE)：

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

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：

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

这时再计算均方误差呢，变成了 $n+2$ 个点，一种我们设想的“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：

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

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(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，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

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”的度量方式。^{Measurement Metrics}

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“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：

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

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。

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。

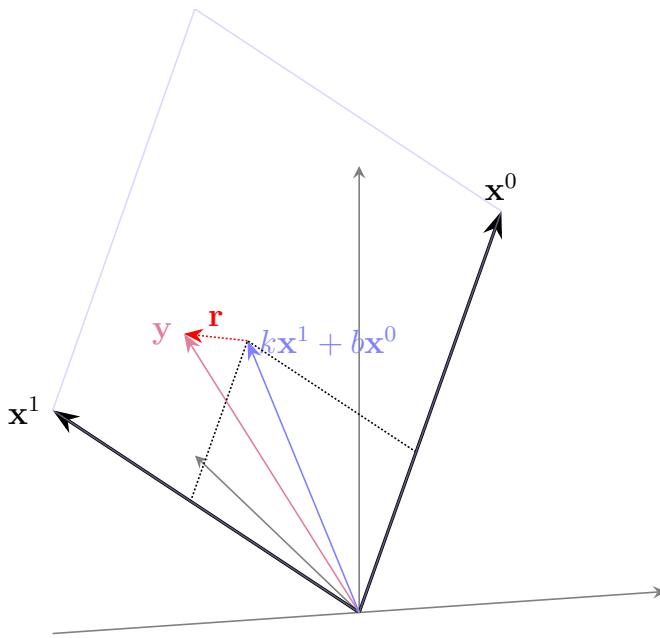


图 2: 从几何的角度看残差

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

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

即均值。

看斜率之前先看看 方差 和 协方差, 方差⁶的表达式是

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

⁶此注释写给学过数理统计的读者: 此处并非 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

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

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

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

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 $k = \text{Cov}(\mathbf{x}, \mathbf{y})/\text{Var}(\mathbf{x})$ ，如果把协方差看作一种乘法⁷，那么 $k = (\mathbf{x} \cdot \mathbf{y})/(\mathbf{x} \cdot \mathbf{x})$ 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

<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

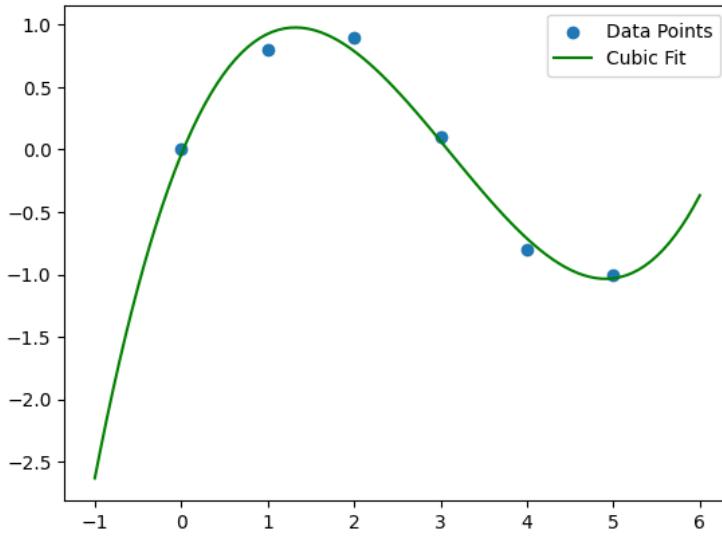


图 3: 多项式拟合示意图 (图为 3 次拟合)

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。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

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

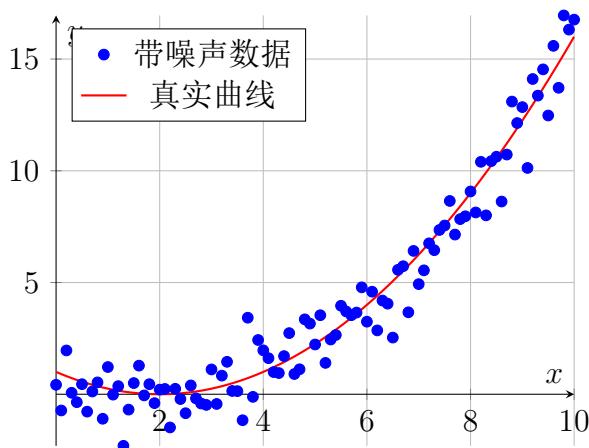
⁸记号说明：虽然习惯上幂次从大到小排列，但是为了下标和幂次的统一性，所以这里选择从常数项到最高次项排列

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

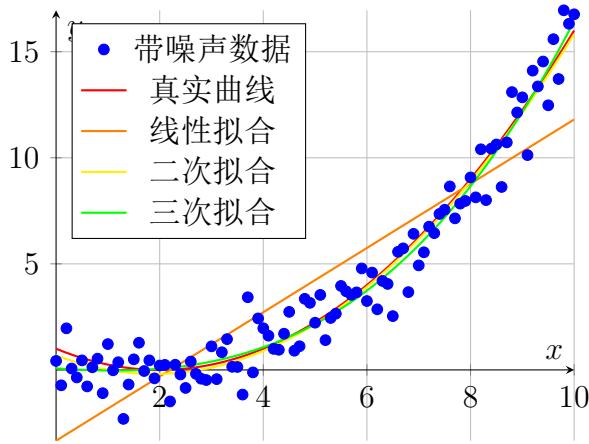
拿做题打个比方：使用直线拟合明显不线性的数据是方法错了，只能说是没完全学会。但是用接近数据量的参数来拟合数据，留给它的空间都够把结果“背下来”了，捕捉到了数据的细节，却忽略了数据背后的规律，化成了一种只知道背答案的自我感动。在几道例题上能做到滴水不漏，但是一遇到新题就束手无策。

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。

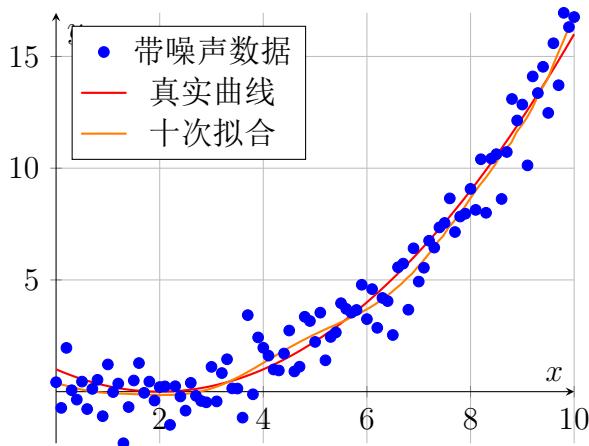


那么现在我们来试试用不同次数的多项式拟合这个数据集。不难看出线性拟合的线与数据点还是相差不少，因为它没能提供可以制造数据“弯曲”形状的项，它没能捕捉到数据更加复杂的趋势，这种现象称为欠拟合^{Underfitting}。2 次曲线的效果几乎和真实曲线一样，即使提升到 3 次也没有太明显的改变，它们拟合的效果都还算好。

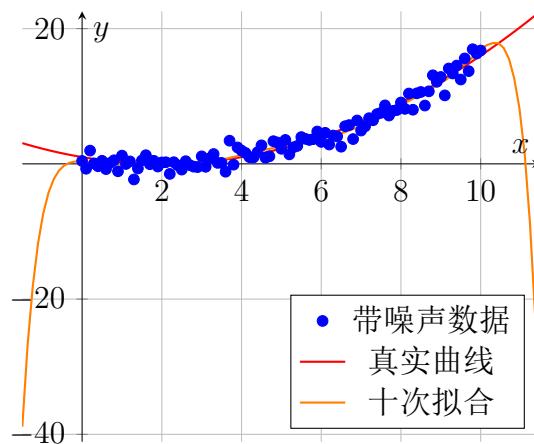
⁹ $\mathcal{N}(0, 1)$: 表示一个服从 [标准正态分布](#) 的变量，均值为 0，方差为 1



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

11

也拟合进去了，从而给出了泛化性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 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，从百分位就已经不准确，在此之后的数位就不太可靠了。

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。

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。

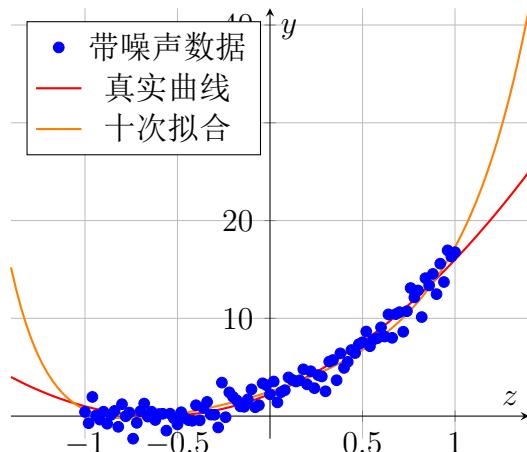
不过即使我们使用了相同的系数 λ ，也仍然有一些技巧可以帮我们把高次项压下去。我们发现一路下来导致问题的根源都是 x 的高次项即使在系数很小时也会导致数值爆炸。但是如果把 x 放到 $[-1, 1]$ 的闭区间内呢？这样即使是 x^{10} 项， x 的 10 次方也不会超过 1，系数再怎么大，至少在小范围内也不会导致数值爆炸，这样一来正则化项才能发挥其约束作用。

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 $z = \frac{0.2x - 1}{\text{Normalization}}$ 再对 y 用 z 的多项式拟合，这种操作称作 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，^{Singularity}高次多项式拟合本身就有^{Ill-posed Problem}奇异牲，解并不稳定（这种对微小噪声敏感的问题常称为 病态问题）。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。这些先验的¹⁴参数通常称为^{Hyperparameter}超参数。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

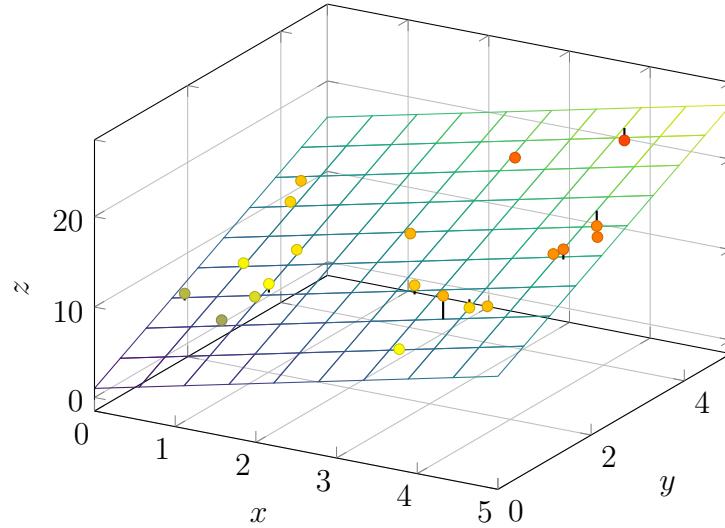


图 4: 高维线性拟合示例

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，Multiple Linear Regression 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”。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：

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

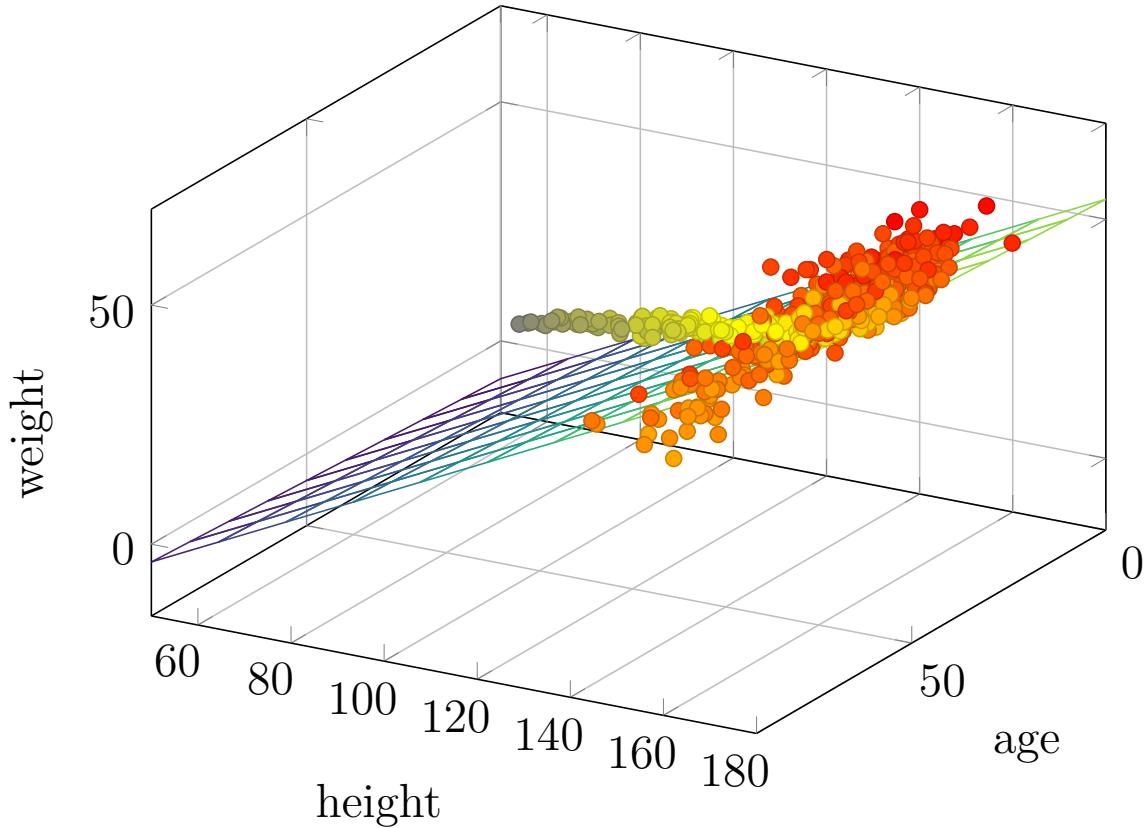
同样地，优化的目标仍然是最小化均方误差，即对 n 个数据点令

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

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超平面了，Hyperplane 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， b 表示偏置，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 $0.04676645038926784 \cdot \text{age} + 0.47766688346191755 \cdot \text{height} - 31.805656676953056 = \hat{\text{weight}}$ ，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

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

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 $\mathbf{x}_{:1}, \mathbf{x}_{:2}, \dots, \mathbf{x}_{:n}$ 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。

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。

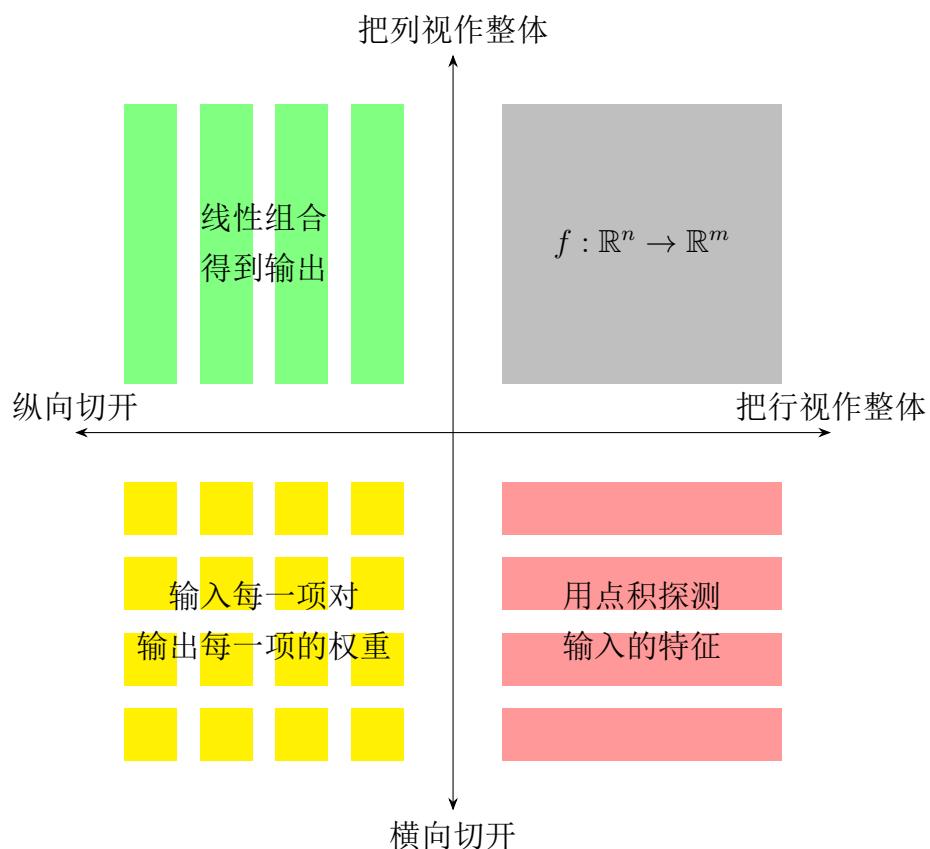
¹⁷记号说明：在公式中我特意将常数项放到了最前面，这是为了让它和多项式拟合的形式保持一致。

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。

Interpretation
矩阵有很多种 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。

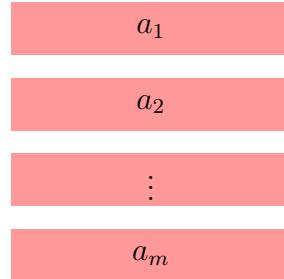
¹⁹几种视角：整体解读、按行解读、按列解读、按元素解读。

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 $\mathbb{R}^n \rightarrow \mathbb{R}^m$ 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：



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 ^{Homogeneous} 线性函数，它的形式也只能是这种 x 的 ^{Weighted Sum} 加权和²³，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、^{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。

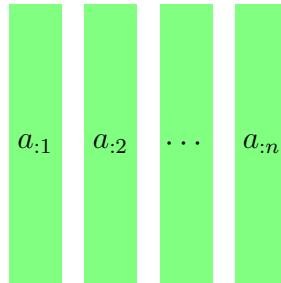
²²严格地说 a 是行向量， x 是列向量，这中间在数学上有一些差别，是矩阵乘法而不是点积。但是在计算机存储中，因为都是一维数组，从实用的角度并不需要纠结于此，这种 ^{Abuse of Notation} 记号混用就见怪不怪了。

²³齐次：指不带偏置(常数)项。

Feature Detector
我们可以把矩阵的每一行看作一个特征检测器，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)。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。

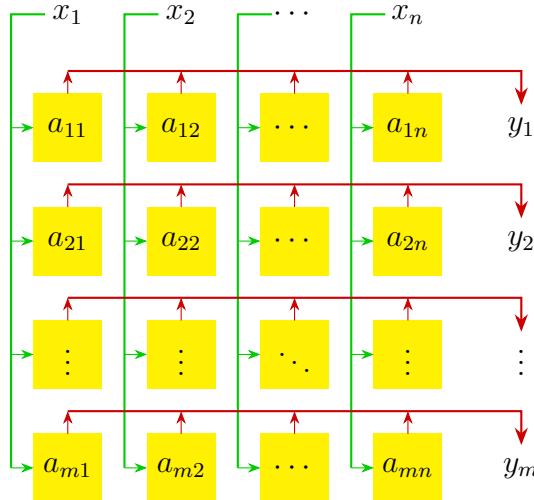
最后一种角度则带有更为浓重的 ^{Distruction}解构²⁴色彩，如果把矩阵看成一个数表，作为一个
填了数字的 $m \times n$ 矩形：

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。

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}

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 Construction 建构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：



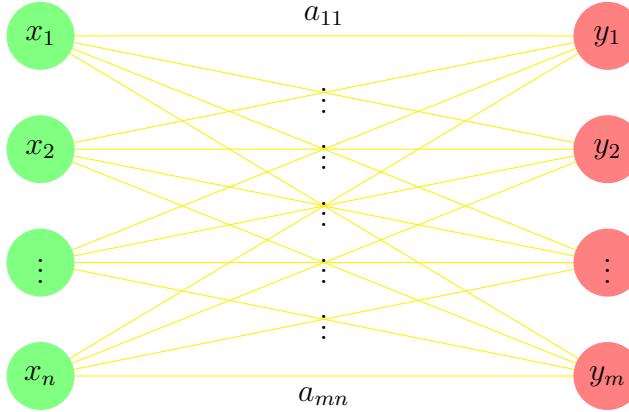
矩阵乘法可以看成这样，输入的每个分量沿着列地址线²⁶输入到每一列的所有块，由每个元素乘上对应的权重后，将结果“上传”到对应的行地址线上，最后在行地址线上的所有块累积起来得到输出的每个分量。

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 Linear Layer 线性层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 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 $b = 0$ 。



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

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

但是如果改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} 。

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

唉？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 视作假，1 视作真²⁹。先来看看几种最简单的逻辑运算。

1. 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. 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. 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由低电平表示，逻辑1由高电平表示，TTL和CMOS电路各有多种电压标准，感兴趣的读者可以自行学习电路的知识。

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 Exclusive Or 异或 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：

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

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

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

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

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

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”，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。

Architecture Instruction Set

现代的 CPU 通常包含复杂的“架构”与“指令集”，但是为了便于理解，我们先只考虑一个极度简化的 CPU，它就像是在一张“草稿纸”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，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”。

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

这里写作 a, b 的其实都表示内存上的一个地址, 类似于如果给行编号, 那么 a, b 就是行号。再引入一个额外的符号, [a] 表示取地址 a 上的值, 例如当内存单元 42 中存着值 64 时, [42] 就表示 64, 例如 mov 10, [42] 表示的就是把 64 号内存的值赋给 10 号内存。^{Immedate Value} #x 表示 立即数值 x, 例如 #10 表示数值 10 本身, 而非内存位置 10。那么我们可以写出一个简单的程序, 例如把内存 0 位置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	$[x_1, x_2, x_3, \dots]$
(initial)	$[x, y, -, \dots]$
▷ mov 2, 0	$[x, y, x, \dots]$
▷ mov 0, 1	$[y, y, x, \dots]$
▷ mov 1, 2	$[y, x, x, \dots]$

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

```
mov r0, #0    ; 将 0 写入 0 号寄存器
mov r1, 8     ; 将 8 号内存的值赋给 1 号寄存器
mov r2, 9     ; 将 9 号内存的值赋给 2 号寄存器
loop:
    add r0, [r1]    ; 将 1 号寄存器指向的内存的值加到 0 号寄存器
    add r1, #1      ; 1 号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
```

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	$[r_0, r_1, r_2, \dots, x_8, x_9, x_{10}, \dots, x_{15}, x_{16}, x_{17}, \dots]$
(initial)	$[-, -, -, \dots, 15, 18, -, \dots, 1, 2, 3, 4, \dots]$
$\triangleright \text{mov } r_0, \#0$	$[0, -, -, \dots, 15, 18, -, \dots, 1, 2, 3, 4, \dots]$ 向 r_0 写入 0
$\triangleright \text{mov } r_1, 8$	$[0, 15, -, \dots, 15, 18, -, \dots, 1, 2, 3, 4, \dots]$ 将 x_8 的 15 赋给 r_1
$\triangleright \text{mov } r_2, 9$	$[0, 15, 18, \dots, 15, 18, -, \dots, 1, 2, 3, 4, \dots]$ 将 x_9 的 18 赋给 r_2
$\triangleright \text{add } r_0, [r_1]$	$[1, 15, 18, \dots, 15, 18, -, \dots, 1, 2, 3, 4, \dots]$ $r_1 = 15$, 取 $x_{15} = 1$ 加到 r_0
$\triangleright \text{add } r_1, \#1$	$[1, 16, 18, \dots, 15, 18, -, \dots, 1, 2, 3, 4, \dots]$ r_1 加 1 (指向 x_{16})
$\triangleright \text{cmp } r_1, r_2$	$[1, 16, 18, \dots, 15, 18, -, \dots, 1, 2, 3, 4, \dots] \rightarrow (16 \neq 18, \text{跳回 loop})$
$\triangleright \text{add } r_0, [r_1]$	$[3, 16, 18, \dots, 15, 18, -, \dots, 1, 2, 3, 4, \dots]$ $r_1 = 16$, 取 $x_{16} = 2$ 加到 r_0
$\triangleright \text{add } r_1, \#1$	$[3, 17, 18, \dots, 15, 18, -, \dots, 1, 2, 3, 4, \dots]$ r_1 加 1 (指向 x_{17})
$\triangleright \text{cmp } r_1, r_2$	$[3, 17, 18, \dots, 15, 18, -, \dots, 1, 2, 3, 4, \dots] \rightarrow (17 \neq 18, \text{跳回 loop})$
$\triangleright \text{add } r_0, [r_1]$	$[6, 17, 18, \dots, 15, 18, -, \dots, 1, 2, 3, 4, \dots]$ $r_1 = 17$, 取 $x_{17} = 3$ 加到 r_0
$\triangleright \text{add } r_1, \#1$	$[6, 18, 18, \dots, 15, 18, -, \dots, 1, 2, 3, 4, \dots]$ r_1 加 1 (指向 x_{18})
$\triangleright \text{cmp } r_1, r_2$	$[6, 18, 18, \dots, 15, 18, -, \dots, 1, 2, 3, 4, \dots] \rightarrow (18 = 18, \text{顺序执行})$
$\triangleright \text{mov } 10, r_0$	$[6, 18, 18, \dots, 15, 18, 6, \dots, 1, 2, 3, 4, \dots]$ 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 ^{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

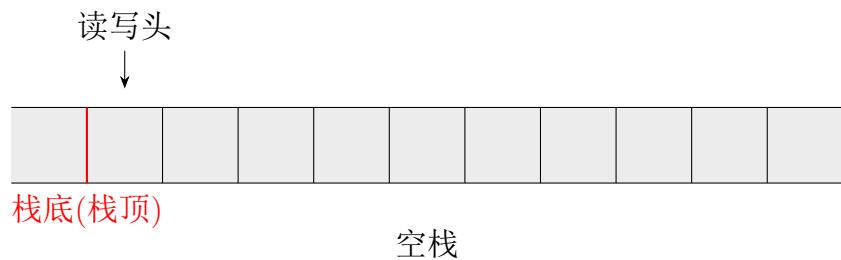
⁴⁵别名：例如在本例中，它指代 `add r0, [r1]` 所在的行

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**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**, 而其它可以随意用作存储的寄存器称为 **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 **后进先出**), 这就引出了**栈**。

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。

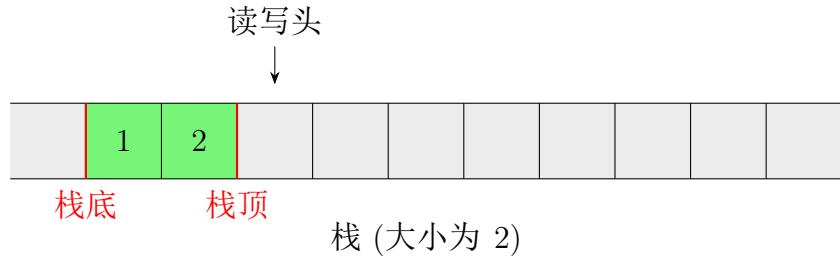


当我们向其中加入一个元素时, 就把这个元素放在栈顶, 同时读写头指向下一个位置。这个过程称为压栈, 例如上面的空栈加入一个元素后的状态是这样的:

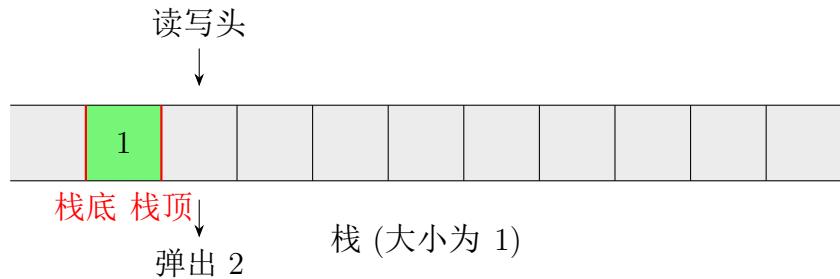


⁴⁶程序计数器: 在实际的 CPU 中, 程序计数器是一个寄存器, 用来存储待执行的指令地址。

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**。于是我们大手一挥，把 8 号位置作为 **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

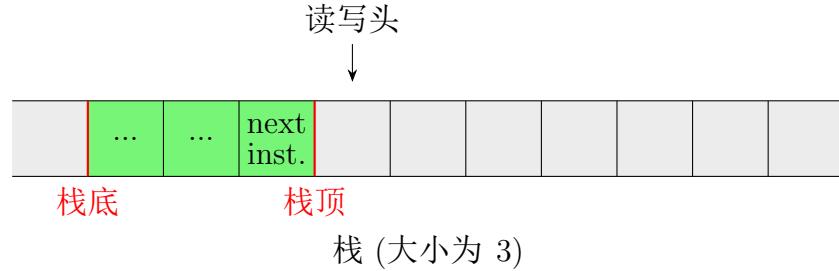
⁴⁷80：这个数字有其历史原因，早期计算机终端通常只有 80 列，因此 80 个字符以内成为了 Linux 编码的规范，这个规范延续到了很多语言的编程风格建议之中，成为一种约定俗成。这里限制深度 80 意味着如果使用“标准”的列宽，一行能写下所有的左括号。

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、Ray Tracing、光路追踪等，完全可能达到成百上千层。

⁴⁹jmp：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。



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。



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和返回的指令。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	<div style="border: 1px solid black; padding: 5px;"> mov ans, a ret </div>	返回
if a != b jmp addr	<div style="border: 1px solid black; padding: 5px;"> cmp a, b jne addr </div>	不等跳转
if a == b jmp addr	<div style="border: 1px solid black; padding: 5px;"> cmp a, b je addr </div>	等于跳转
if a < b jmp addr	<div style="border: 1px solid black; padding: 5px;"> cmp a, b jl addr </div>	小于跳转
do {...} while (a != b)	<div style="border: 1px solid black; padding: 5px;"> loop_i: ... cmp a, b jne loop_i </div>	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;

```

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、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。SIMD

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。

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>

2.3 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为 32 的数组。

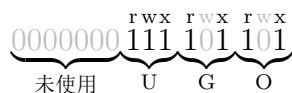
了解信息竞赛的同学或许知道，位运算催生了各种各样的优化技巧，位运算的各种巧妙操作也是大家津津乐道的话题。从 [借助位运算实现俄罗斯方块](#) 到 [求解 n 皇后问题](#)，再到 [各种关于整数的问题](#)（例如前文提过的原地交换整数技巧），位运算的应用无处不在。

不过这里我们来看一个更为重要的概念：位标志。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⁵³，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。^{Read Write Execute} `r, w, x` 分别表示 读，写 和 执行 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：



⁵³`ls -l`: 其中 `ls` 命令用于列出文件夹中的文件和子文件夹，选项 `-l` 表明以 `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⁵⁴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 GPIO⁵⁵、串口、定时器、中断等模块。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;
```

⁵⁴STM32：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。

⁵⁵GPIO：即通用型输入输出，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：

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

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：

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

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 API⁵⁶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⁵⁷来创建一个窗口，其中有一个参数叫 dwStyle⁵⁸，它就用于设置窗口样式。与现代的许多使用类 CSS⁵⁹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

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。

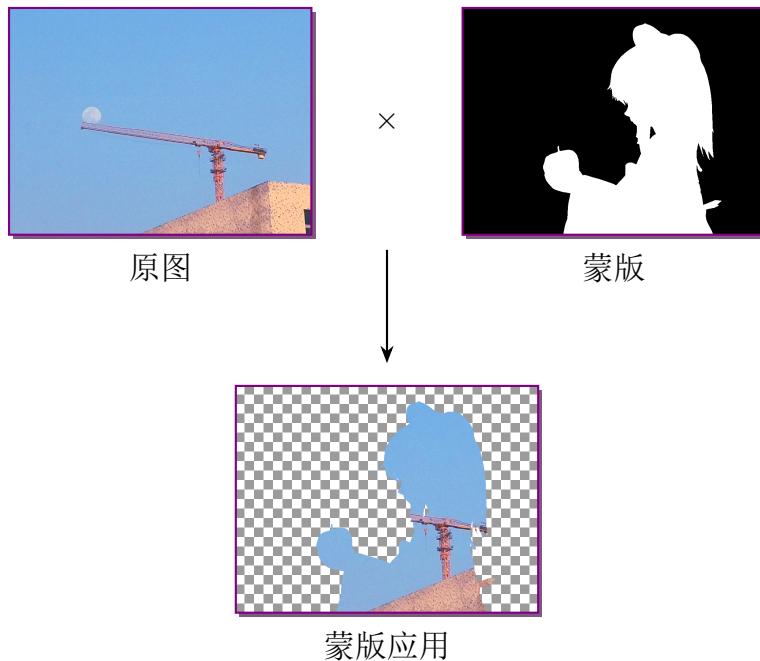
⁵⁶Win32 API: 应用程序编程接口，是 Windows 操作系统的一套接口。

⁵⁷CreateWindowEx: 最早的接口是 CreateWindow, 但是后来为了支持更多的 **拓展** 功能, 微软推出了 CreateWindowEx。

⁵⁸**dwStyle:** 双字 Word 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。

⁵⁹CSS: 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⁶⁰ (这里沿用 Photoshop 中的约定, 将透明的部分用格子表示):

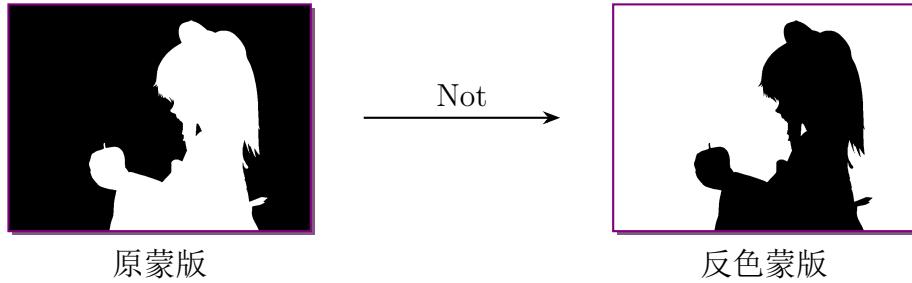


在这里我们可以看到, 原图中蒙版为白色的部分显示出来, 黑色的部分隐藏了起来。将它叠加到其它的图层上, 黑色部分展示的就是下一个图层。

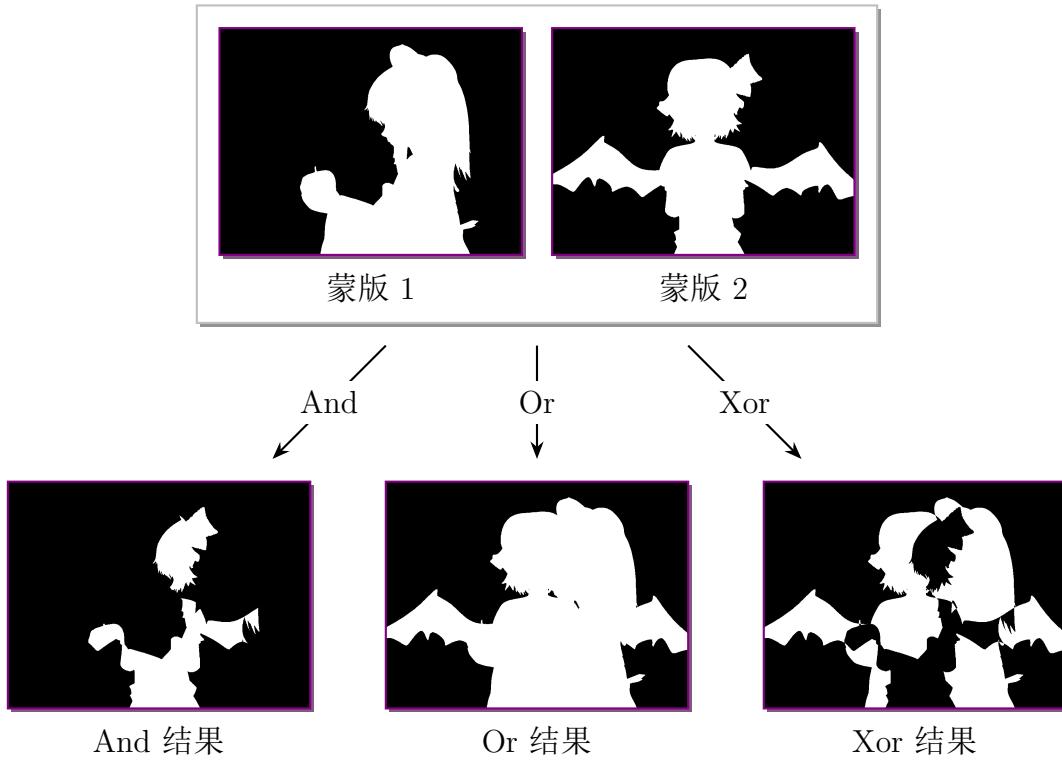
至此我们已经看到了位标志的许多应用, 使用这样的表达方式从根本上是因为它用一种统一而简洁的方式将多个状态编码到数字中, 各种从逻辑上有所区别的状态就这样打包成了数字。我们不妨再以图像为载体的数组为呈现方式, 看看几种位运算是如何工作的。

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:

⁶⁰说明: 此处的蒙版截自视频 [【東方】Bad Apple!! P V 【影繪】](#), 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？但是问题就在于，在我们赋予它一个标签的同时，在数学上看来，它就排布在了数轴上。但在这个过程中，我们是否真的能够保证这个排列是正确的呢？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，同样是标签，为什么是 0, 1, 2, 3 而不是 1, 0, 3, 2 呢？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，漫无目的地把它用数字编号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 $2 = (1+3)/2$ ，所以说家兔是狗和仓鼠的平均值。

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

猫 : (1, 0, 0, 0)

狗 : (0, 1, 0, 0)

家兔 : (0, 0, 1, 0)

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 One-Hot Encoding 独热编码。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, …。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：

类别编号			
0	1	2	3
■	■		■
		■	■
■		■	■
	■		■
■		■	
	■	■	
.....			

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⁶¹地执行指令。

⁶¹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，为了编码一个多分类问题的答案，我们可以提出一系列小的问题，并将这些小问题的答案以 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>

拓展阅读——那么，GPU 呢？

Graphics Processing Unit

GPU，全称“图形处理器”，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，[显卡如何构成](#)，CPU 和 GPU 又有什么区别呢？

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⁶²，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：

```
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 FPS⁶³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 Parallel Computing 并行计算，喷颜料总比一笔笔涂快，为了高效进行并行计算，GPU 诞生了。

⁶²方便起见，这里只考虑黑白颜色。

⁶³FPS: Frame Per Second 帧每秒 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](#)，[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⁶⁴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](#)，[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](#)⁶⁵。

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，

⁶⁴运算单元：在 Nvidia 的显卡中通常称为 CUDA (Compute Unified Device Architecture) 核心。

⁶⁵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/wheel/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 ^{Thread Block} 线程块 再交给 GPU 的工作。在 GPU 中的 ^{Streaming Multiprocessor} 流式多处理器 将线程块以线程束为单位调度，交给若干个 ^{Warp} 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。

3 为什么是神经网络

3.1 神经网络：一个大的函数

Neural Network

相比于 神经网络 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。

Artificial General Intelligence

虽然目前人类仍然很难说摸到了 通用人工智能⁶⁶的边界，但人工智能已然在许多问题上取得了巨大成就，走出了 20 世纪末 21 世纪初被大众认为是“伪科学”的寒冬。经过 深度残差网络 在图像识别的重大突破、 AlphaGo 学会下围棋、 Transformer 在翻译比赛取得优异成绩并引来一波生成式模型的热潮等等，人工智能就这样走向了时代的焦点。但是如果要问：为什么它这么成功？最直接的回答仍是： It works.

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

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

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

读者看到的第一反应大抵是感到看不懂。不过我也并非想让读者先学完再来看这个表格，而是希望读者看到：解决问题的方法虽然多样，但仍可根据若干指标大致分类。表中

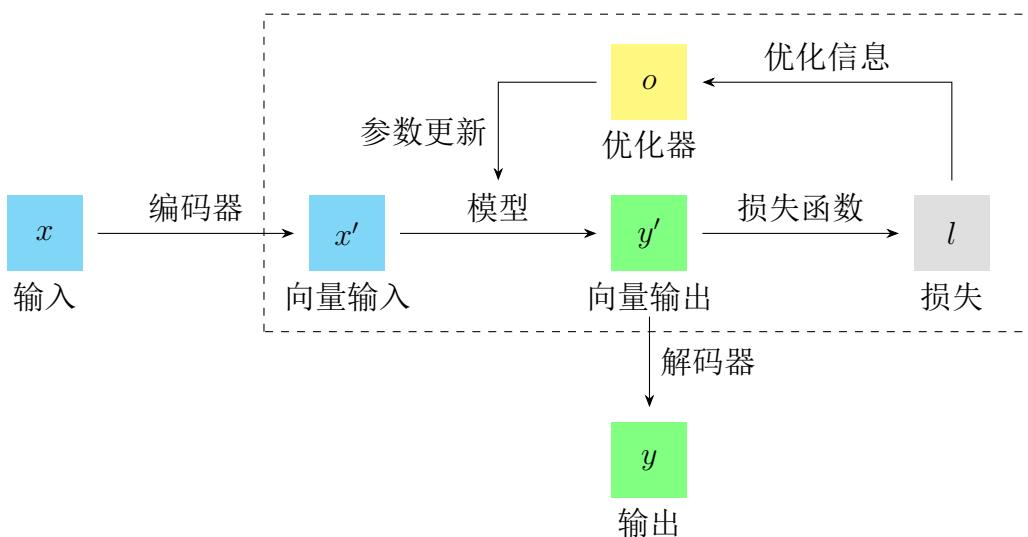
⁶⁶通用人工智能：指能像人类一样解决各种通用的问题的人工智能。

的术语有的是模型结构，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将数值构成的向量变为我们想要的输出。

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

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



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

在这里我想简单讲讲使用矩阵运算的原因。在第一章中我们已经简单地学习了矩阵运算的基本知识，它本质上是正比例函数在向量空间中的推广，只是 $y = kx$ 中的斜率变成了一个个从输入 x_j 连接到输出 y_i 的权重 w_{ij} 。从行看过去，它反映了输出的每个分量（或称为特征）是如何由输入的每个分量线性组合而成的。而从列看过去，它表明了输入的每个分量是如何影响输出的。就像一次函数有一个常数项一样，矩阵运算也有一个偏置项 b ，运算的总体结构是 $y = wx + b$ 。从代数上看，它运算简单⁶⁷，而从分析上看，它的输出变化光滑，容易求导⁶⁸。

下一节中我们会引入激活函数，暂且不论它们的具体形式如何，它们也是一些非常简单的运算。或许读者会有疑问，这样一些简单的运算，真的有能力让神经网络胜任复杂的任务吗？**万能逼近定理**⁶⁹虽然在理论上告诉我们它可以，却需要假定足够多的神经元，并不令人安心。所幸无数的实验表明：可以，在人类能实现的范围内，量变也可以引起质变，将一系列简单的单元堆叠起来，便可以形成复杂的行为。Philip W. Anderson⁷⁰曾说过：“More is different”，他的原意指的是物理学中，微观的规律并不能简单地推导出宏观的规律，整个系统可以表现的与单个元素完全不同，因此微观和宏观需要不同的理论来描述。但这里我们不妨借用一下，同样地认识到大量简单的数学函数也可以产生复杂的行为。在神经网络的 **涌现**⁷¹现象中，这一事实不断地被验证。就像婴儿可以通过教育成为适应社会的成年人一样，适当的算法和足够的训练数据的确可以让神经网络学会知识。

需要说明的是，现代的机器学习库 PyTorch 与 TensorFlow 都提供了完善的参数更新机制，使得用户不必自己实现优化算法。这可以说是非常简单易用，让用户可以聚焦模型的设计。不过我仍然会解读其中的原理，并试图说明设计网络结构与优化算法的人为什么要这么做。⁷²

⁶⁷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。

⁶⁸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。

⁶⁹万能逼近定理：指出足够大的神经网络可以以任意精度在给定范围内拟合任意的复杂函数。

⁷⁰Philip W. Anderson：美国物理学家，1977 年诺贝尔物理学奖获得者。

⁷¹涌现：即增大参数量带来性能突然提升的现象。

⁷²其中的原理：实际上人类理解的神经网络工作原理与计算机实际运行的原理或许有很大的区别，人类对现在大部分网络的理解本质上都是经过实验后进行的归纳甚至是猜测，而非从数学上严格证明。神经网络的 **可解释性**^{Interpretability} 仍然是很大的问题，因此很多时候人们只知道怎么样做效果好，而不“真正地”理解为什么这么做效果好，有时也被调侃为“新时代的炼金术”。虽然有许多相关的 **解释** 来帮助人们了解神经网络

推荐阅读

- 这篇文章讲述了神经网络的起源：
如何简单形象又有趣地讲解神经网络是什么？ - 佳人李大花的回答 - 知乎
<https://www.zhihu.com/question/22553761/answer/3359939138>
- 读者或许会好奇所谓的万能逼近定理需要是如何能逼近给定函数的，这篇回答的解释不错：
神经网络的万能逼近定理已经发展到什么地步了？ - 牛油果博士的回答 - 知乎
<https://www.zhihu.com/question/347654789/answer/1534866932>
- 这一问题下有关于神经网络“涌现”出新的现象的讨论，对其机理感兴趣的读者也可以想想背后的原因：
如果神经网络规模足够大，会产生智能吗？ - 知乎
<https://www.zhihu.com/question/408690594>

中发生什么，但学界内仍然没有形成一套系统的理论。

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⁷³。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⁷³，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} 线性整流函数 (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：

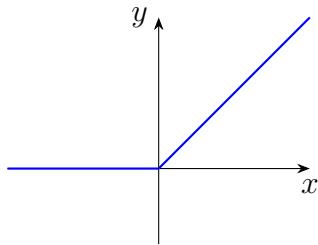


图 7: ReLU 函数图像

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 ^{Rectifier} 电路中的半波整流器一样，把负值截断了。

⁷³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。

而它的导数也非常简单：

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

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⁷⁴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 1⁷⁵。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

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

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 case⁷⁶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：

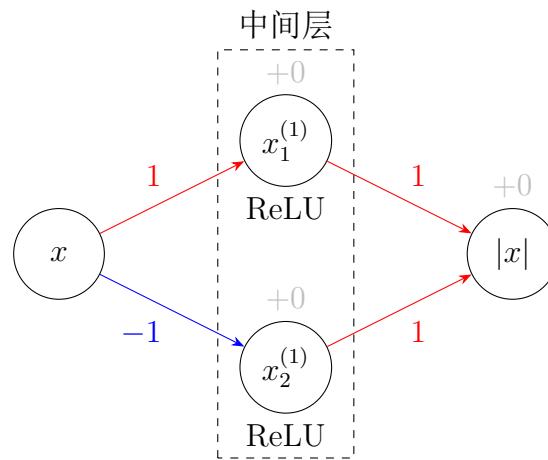


图 8: 神经网络表示 $|x|$

⁷⁴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。

⁷⁵0 处的导数：PyTorch 通常选择 0

⁷⁶toy 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。

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

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

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

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

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：

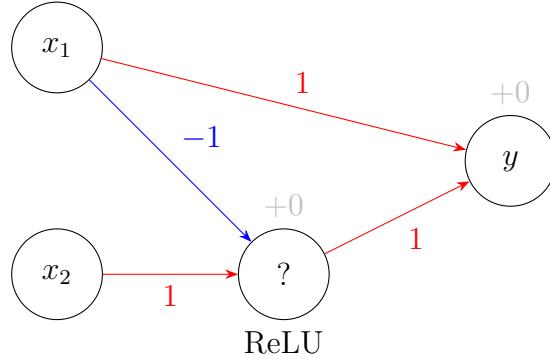


图 9: 神经网络表示 $\max\{x_1, x_2\}$ 的一种方法

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⁷⁷。

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

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

⁷⁷并不优雅：与之对比，在深层神经网络中通常会引入看起来有些像这里的 **跳连接** 结构，由此引出 **残差网络** 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。

这一式子中的三个分量提出来，便可以得到

$$\begin{aligned}x_1^{(1)} &= \text{ReLU}(x_1 + 0x_2) \\x_2^{(1)} &= \text{ReLU}(-x_1 + 0x_2) \\x_3^{(1)} &= \text{ReLU}(-x_1 + x_2) \\y &= x_1^{(1)} - x_2^{(1)} + x_3^{(1)}\end{aligned}$$

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：

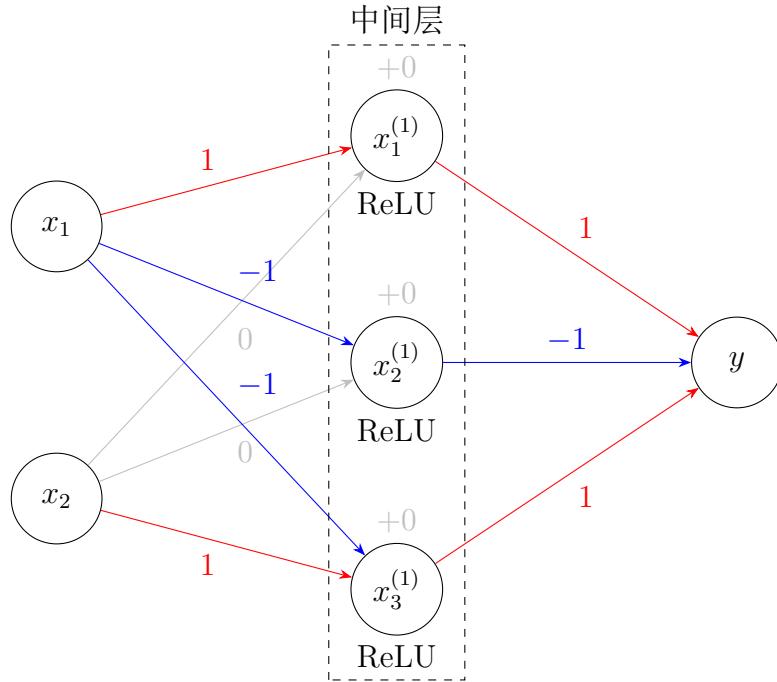


图 10: 神经网络表示 $\max\{x_1, x_2\}$ 的另一种方法

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

一个有趣的事是，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

$$\begin{aligned}x_1 \text{ and } x_2 &= \text{ReLU}(x_1 + x_2 - 1) \\x_1 \text{ or } x_2 &= \text{ReLU}(x_1) + \text{ReLU}(x_2 - x_1) \\x_1 \text{ xor } x_2 &= \text{ReLU}(x_1 - x_2) + \text{ReLU}(x_2 - x_1)\end{aligned}$$

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⁷⁸，因此从这一特例来看，求特征的交集、并集的操作确实可以自然地以权重的方式编码到网络的运算中。

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

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：

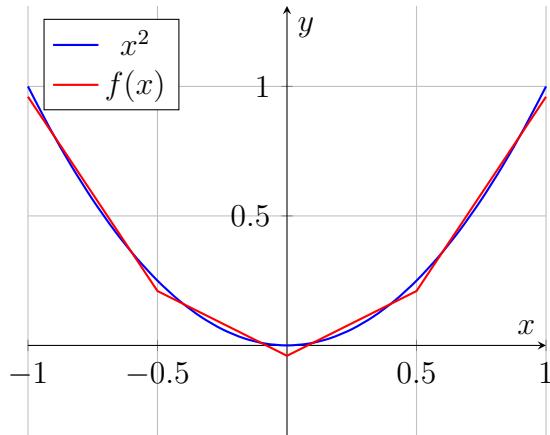


图 11：分段线性函数近似光滑函数

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

⁷⁸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。

控制在一个范围之内。遵循这些原则，ReLU 网络的表达能力已经足够强大，能解决大多数实际问题。尽管仍有一些细节需要注意，但这并不影响我们对其整体能力的理解。

另外再提一嘴其它的激活函数。Sigmoid 函数⁷⁹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：

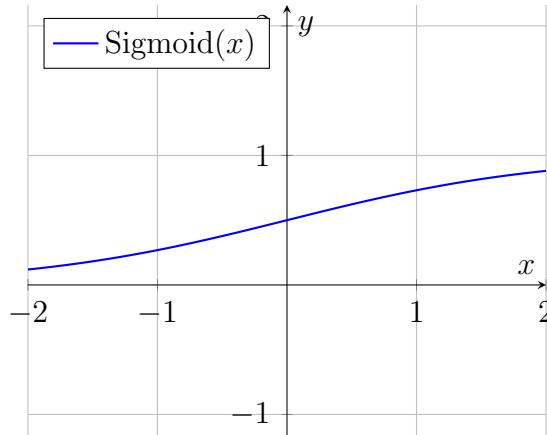


图 12: Sigmoid 函数图像

\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 $[-1, 1]$ ：

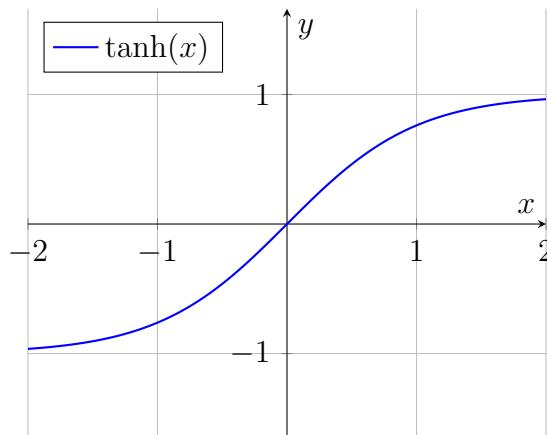


图 13: \tanh 函数图像

⁷⁹Sigmoid 函数：Sigmoid 来源于拉丁语，得名于其类似小写字母 sigma 变体 ς 的形状。

早期的研究中，它们出现在许多生物学的研究中，可以描述生物神经元的激活或者极化程度，于是人工神经网络出于仿生的考虑也使用了它们。然而它们在两端很小的导数也为优化带来了许多麻烦，导致了 ^{Vanishing Gradient} 梯度消失⁸⁰ 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 $x < 0$ 的区域也有一个斜率，定义为

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

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

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

⁸⁰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。

推荐阅读

- 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>

3.3 神经网络的训练

有了前面的模块，我们已经可以搭建起简单的神经网络了，理论上通过“合适的”参数就可以拟合任意的函数了，“合适的”参数从何而来呢？回顾第一章中，线性拟合问题可以完全通过解析方法求解得到最优的参数，然而神经网络网络却是一个复杂的非线性函数，根本不可能对各种情况分段讨论给出解析解。这是否意味着我们就无能为力了呢？当然不是。通过一些手段可以让信息从数据定向地“流向”模型的参数，通过一套优化算法来调整参数，使得模型的输出尽可能地接近目标，这一过程就叫做 ^{Training} 训练，这一调整的过程使用的各种优化方法大多是基于梯度下降法⁸¹的。

正如第一章中所见，拟合的好不好需要量化为损失的数值，来定量分析大小如何。而在机器学习上，我们就是借助这一可量化的评判标准——利用梯度下降算法给出了往更好的方向前进的一步。以 [3Blue1Brown的视频](#) 为代表的一系列科普教程中都有对这一算法的良好讲解。不过此处我仍然通过下山这个最经典的比喻来试图说明这一算法的原理。

前面我提到过，每层函数的函数需要可导，可导性就在此处显得尤为重要。即使不知道函数的总体行为，但是我们仍然可以通过导数对当前位置附近的情况形成大致的感知。就像在山上行走时，想快速下山的人并不需要知道整座山的形状，只要知道当前的坡度，就可以沿着向下的方向走。不过这一步不能走太大，在机器学习中，走太大了可能会走到山的另一边去，导数仅为优化提供了局部的信息。

在一元函数中，导数的正负可以直接指出当前的坡度是向上还是向下，局部的近似意味着对于很小的 Δx 可以写出

$$\Delta y \approx f'(x_0)\Delta x$$

看起来像是这样：

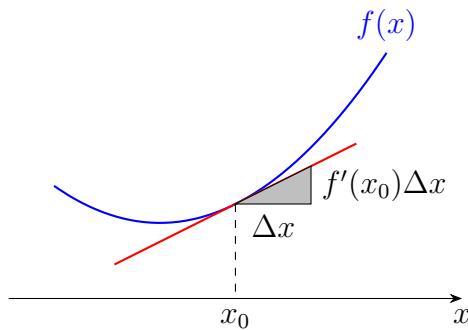


图 14: 一元函数的导数近似

导数反映的是切线的斜率，也就是函数在局部的变化率信息。为了让 $f(x)$ 变小，令

⁸¹Gradient 词源说明：gradi 是一个拉丁语词根，意为行走，与“步行”相关。与 progress, regress 等词中的“gress”是同一词根。因此从词根来看其实可以理解为在步行中下降。

Δx 的方向与 $f'(x)$ 相反就可以了。也就是说，可以令

$$\Delta x = -\eta f'(x)$$

Learning Rate

这里的 η 决定了一步要走多大，称为 学习率⁸²。如果 η 太小，可能会走得很慢；如果 η 太大，可能会走过头。一步步地往下走，我们的位置就像是一个小球，从斜坡上滚下去，最终停留在一个极小值⁸²。

不过我们的神经网络毕竟有很多的参数要在同时调整，你或许想问，我们能不能固定其它的参数，只调整一个参数，把它变成一元函数来处理呢？可以是确实可以，这可以把它变为了一元函数的情况，只是这种方法太低效了，更高效且优雅的做法是联合优化它们，同时调整所有的参数。听起来很合理，但是这初听起来似乎有点玄学，毕竟我们并不知道所有的参数之间如何作用的，庞大的网络让它们之间的关系变得复杂而难以捉摸。因此我认为需要在此处引入一些多元微积分的知识，来把求导操作推广到多维，得到所谓的梯度与 反向传播 算法。

在一元情况下，导数是这样定义的

$$f'(x_0) = \lim_{\Delta x \rightarrow 0} \frac{f(x_0 + \Delta x) - f(x_0)}{\Delta x}$$

但是在输入 x 是向量的情况下，数学运算并不允许除以一个向量，那还能如何求导呢？回应看一下最初的动机：求出导数的目的是为了让它在局部能用简单的线性函数来近似。如果使用 dx 表示一个输入的变化量，用 dy 表示我们对输出结果变化量的估计。在一元的情况下这个式子可以简单地写成

$$dy = f'(x_0)dx$$

而在多元情况下，应当估计每个分量的变化量如何影响输出，再把这些线性的分量叠起来，从而有线性近似

$$dy = a_1 dx_1 + a_2 dx_2 + \cdots + a_n dx_n$$

而线性近似意味着在 dx 足够小的情况下，这一近似给出的估计 dy 与真实的变化量 Δy 之间的差距是可以忽略的，也就是说

$$\lim_{dx \rightarrow 0} \frac{\Delta y - dy}{\|dx\|} = 0$$

不过上面的极限在实际应用中可以假设成立，在足够好的可导的假设下，需要关心的只是如何算出这些系数 a_i 。对于标量函数 f 和向量输入 x ，这样的一些系数所构成的向量就叫做 梯度^{Gradient}， f 对 x 的梯度在 x_0 处的值常记作 $\nabla_x f(x_0)$ ，而从根本上，它最主要的作用

⁸² 极小值：不过这个极小值可能只是山谷中的一个盆地，而非全局的最低点。

就是提供了线性近似 $dy = \nabla_x f(x_0) \cdot dx$ 的系数⁸³。为了沿着网络层层回溯求出梯度，需要再引入链式法则。

设想如果 y 本身并不是 x 的函数，而是经过两步得到的 $x^{(1)} = f_1(x)$ 和 $y = f_2(x^{(1)})$ ，那么应该怎么求导呢？这一过程需要分步进行。如果从一个逐元素的视角看来，实际上是先使用 $dx^{(1)}$ 来估计 y 的变化量，然后再借助 dy 与 $dx^{(1)}$ 之间的关系来估计 dx 对 y 的影响。更为具体地说：如果能写出

$$dy = a_1 dx_1^{(1)} + a_2 dx_2^{(1)} + \cdots + a_m dx_m^{(1)}$$

而 dx 对 $dx^{(1)}$ 各项的影响分别是

$$\begin{aligned} dx_1^{(1)} &= b_{11} dx_1 + b_{12} dx_2 + \cdots + b_{1n} dx_n \\ dx_2^{(1)} &= b_{21} dx_1 + b_{22} dx_2 + \cdots + b_{2n} dx_n \\ &\vdots \\ dx_m^{(1)} &= b_{m1} dx_1 + b_{m2} dx_2 + \cdots + b_{mn} dx_n \end{aligned}$$

接下来，怎么把 dy 写成 $c_1 dx_1 + c_2 dx_2 + \cdots + c_n dx_n$ 的形式呢？答案非常简单粗暴：把每个 $dx_i^{(1)}$ 代入到 dy 的表达式中，就有了

$$\begin{aligned} dy &= a_1(b_{11} dx_1 + b_{12} dx_2 + \cdots + b_{1n} dx_n) + \\ &\quad a_2(b_{21} dx_1 + b_{22} dx_2 + \cdots + b_{2n} dx_n) + \\ &\quad \cdots + \\ &\quad a_n(b_{m1} dx_1 + b_{m2} dx_2 + \cdots + b_{mn} dx_n) \\ &= (a_1 b_{11} + a_2 b_{21} + \cdots + a_n b_{m1}) dx_1 + \\ &\quad (a_1 b_{12} + a_2 b_{22} + \cdots + a_n b_{m2}) dx_2 + \\ &\quad \cdots + \\ &\quad (a_1 b_{1n} + a_2 b_{2n} + \cdots + a_n b_{mn}) dx_n \end{aligned}$$

由此可见，可以写出表达式

$$\begin{aligned} c_j &= a_1 b_{1j} + a_2 b_{2j} + \cdots + a_n b_{mj} \\ dy &= c_1 dx_1 + c_2 dx_2 + \cdots + c_n dx_n \end{aligned}$$

这表明，只要我们知道下一层计算出来的梯度 a_i 和联系起两层的 b_{ij} ，就可以将下一层的梯度“回溯”到上一层，计算出上一层的梯度 c_j ，这种方法被称作反向传播。于是问题被进一步拆解，变成了更为细化的小问题：如何求出这里每层之间梯度传播的关系？

⁸³说明：这里的 · 是向量点积，表明逐分量相乘并求和

这里选取最简单的一层 $x^{(k)} = \text{ReLU}(wx^{(k-1)} + b)$ 来说明，其中 w 是权重矩阵， b 是偏置。我们需要理解 $dx_i^{(k)}$ 应当如何表示。不过这里需要注意一点，在求导时不但要考虑 $x^{(k)}$ 的变化量，还要考虑 w 和 b 的变化量，因为 w 和 b 作为参数也会影响到 $x^{(k)}$ 的值，更是我们希望调整的对象。

首先写出 $x_i^{(k)}$ 的表达式

$$x_i^{(k)} = \text{ReLU}(w_{i1}x_1^{(k-1)} + w_{i2}x_2^{(k-1)} + \cdots + w_{in}x_n^{(k-1)} + b_i)$$

接下来分类讨论，在 ReLU 内部小于等于 0 的情况下，因为输入有微小变化时输出仍然为 0，所以

$$dx_i^{(k)} = 0$$

而当其输入大于 0 时，ReLU 可以去除掉，即变为

$$\begin{aligned} dx_i^{(k)} &= d(w_{i1}x_1^{(k-1)} + w_{i2}x_2^{(k-1)} + \cdots + w_{in}x_n^{(k-1)} + b_i) \\ &= w_{i1}dx_1^{(k-1)} + w_{i2}dx_2^{(k-1)} + \cdots + w_{in}dx_n^{(k-1)} \\ &\quad + x_1^{(k-1)}dw_{i1} + x_2^{(k-1)}dw_{i2} + \cdots + x_n^{(k-1)}dw_{in} + db_i \\ &= w_{i:} \cdot dx^{(k-1)} + x^{(k-1)} \cdot dw_{i:} + db_i \end{aligned}$$

这里仍然沿用了向量的点积表示法， $w_{i:}$ 表示 w 的第 i 行。由此也可以看出 ReLU 的好处：使用它作为激活函数的梯度回溯非常方便。

这里我们会发现梯度信息发生了一个“分岔”，一部分信息传递给了 $x^{(k-1)}$ ，另一部分则传递给了 w 和 b 。传递给 w 和 b 的信息将在后续用于更新参数，而传递给 $x^{(k-1)}$ 的信息则会继续向后传递求出离结果更远、离输入更近的层的参数梯度。直到传递到输入层，最终得到所有参数的梯度，关于原始输入 x 的输入则会被丢弃⁸⁴。这个过程看起来像是这样，如果说正向计算是把参数信息参与到计算中，一步步计算得到最终的结果（由于我们希望最小化损失，所以在模型的最终输出之后还有一个箭头表示损失函数）：

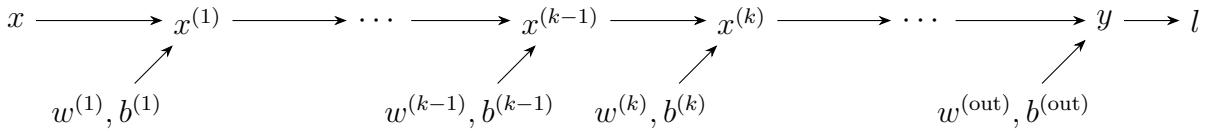


图 15: 正向计算

那么反向传播就是考虑损失变化的估计量 dl ，并将它的梯度信息经由 dy 和各个中间计算结果 $dx^{(k)}$ 逐层传递回去，直到输入层。

⁸⁴丢弃：因为我们并不需要对输入进行调整，只需要对参数进行调整。不过一些研究表明，观察输入的梯度信息可以帮助我们理解模型认为哪些特征更加重要。

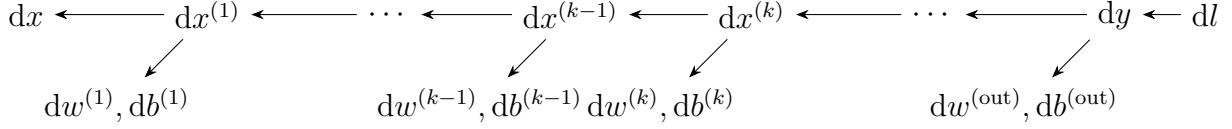


图 16: 反向传播

在图中看到从 $dx^{(k)}$ 指向 $dx^{(k-1)}$ 或者 $dw^{(k)}, db^{(k)}$ 的箭头时，它指的是关于 $x^{(k)}$ 的梯度信息被传递到了 $x^{(k-1)}$ 和 $w^{(k)}, b^{(k)}$ 上。上面的图只是下面表达式的一个形象说明（这里用 $g_x^{(k)}$ 表示对 $x^{(k)}$ 的梯度，同理定义对 w 和 b 的梯度符号）

$$\begin{aligned}
dl &= g_y \cdot dy \\
&= g_x^{(\text{out})} \cdot dx^{(\text{out})} + g_w^{(\text{out})} \cdot dw^{(\text{out})} + g_b^{(\text{out})} \cdot db^{(\text{out})} \\
&= (\text{向前展开这一部分}) + g_w^{(\text{out})} \cdot dw^{(\text{out})} + g_b^{(\text{out})} \cdot db^{(\text{out})} \\
&= g_x^{(k)} \cdot dx^{(k)} + g_w^{(k)} \cdot dw^{(k)} + g_b^{(k)} \cdot db^{(k)} \\
&\quad + \cdots + g_w^{(\text{out})} \cdot dw^{(\text{out})} + g_b^{(\text{out})} \cdot db^{(\text{out})} \\
&= g_x^{(k-1)} \cdot dx^{(k-1)} + g_w^{(k-1)} \cdot dw^{(k-1)} + g_b^{(k-1)} \cdot db^{(k-1)} \\
&\quad + g_w^{(k)} \cdot dw^{(k)} + g_b^{(k)} \cdot db^{(k)} + \cdots \\
&\quad + g_w^{(\text{out})} \cdot dw^{(\text{out})} + g_b^{(\text{out})} \cdot db^{(\text{out})} \\
&= \cdots \\
&= g_x \cdot dx \\
&\quad + g_w^{(1)} \cdot dw^{(1)} + g_b^{(1)} \cdot db^{(1)} \\
&\quad + g_w^{(2)} \cdot dw^{(2)} + g_b^{(2)} \cdot db^{(2)} \\
&\quad + \cdots \\
&\quad + g_w^{(\text{out})} \cdot dw^{(\text{out})} + g_b^{(\text{out})} \cdot db^{(\text{out})}
\end{aligned}$$

随着式子每次将第一项不断地展开，树状的计算图也从计算结果的终点不断向前回溯，最终在每个终止节点处得到对每个分量的梯度信息。理论上只需要手动维护这个图的状态就能完成反向传播的计算，不过实际应用中已经有了非常方便的自动求导工具，现代的机器学习库 PyTorch 和 TensorFlow 都有类似的功能，它们都会在内部维护一个计算图，自动地完成反向传播的计算。以 PyTorch 为例，用户不需要关心如何维护这个图，只需要设置 `requires_grad` 属性为 `True`，在计算后对损失调用 `backward` 方法，就能自动地完成反向传播的计算。

在训练时数据点通常是固定的，所以可以直接令 $dx = 0$ ，这样前面的梯度信息就仅留

下与待优化的参数相关的部分

$$\begin{aligned} \mathrm{d}l &= g_w^{(1)} \cdot \mathrm{d}w^{(1)} + g_b^{(1)} \cdot \mathrm{d}b^{(1)} \\ &\quad + g_w^{(2)} \cdot \mathrm{d}w^{(2)} + g_b^{(2)} \cdot \mathrm{d}b^{(2)} \\ &\quad + \dots \\ &\quad + g_w^{(\text{out})} \cdot \mathrm{d}w^{(\text{out})} + g_b^{(\text{out})} \cdot \mathrm{d}b^{(\text{out})} \end{aligned}$$

而如果我们再做一层抽象，把这一堆参数塞到一个叫做 θ 的向量中，那么就可以把上面的式子写成

$$\mathrm{d}l = \nabla_{\theta} l \cdot \mathrm{d}\theta$$

其中 $\nabla_{\theta} l$ 就是损失函数 l 对参数 θ 的梯度，其各个分量表明了损失函数对每个参数的敏感程度。而仿照一元函数下山的做法，同样可以按照如下的方式沿着“下山最快的方向”来更新参数

$$\mathrm{d}\theta^* = -\eta \nabla_{\theta} l$$

从图形上看，这个过程是垂直于等高线的方向在向下走，最终停留在一个极小值处。或许初看会觉得这一优化方法和等高线并不相关，但如果仔细想想就会发现，实际上等高线的形状就是损失函数的等值线，而沿着等值线的切线方向意味着

$$\mathrm{d}l = \nabla_{\theta} l \cdot \mathrm{d}\theta = 0$$

前一个部分是梯度的方向，而令上式等于 0 成立的方向 $\mathrm{d}\theta$ 就是等高线的切线方向。两个向量的内积为 0 意味着它们是垂直的，用心观察的读者或许会发现，地图上的水系与等高线交汇之处，两条线往往是垂直的，水往低处走是一种自然的梯度下降。

在实际的神经网络中通常有数万乃至数亿个参数，在这样庞大的参数空间中梯度下降虽然有着下降最快的方向的理论支持，但想象一个高维空间已经足够困难，再去想它里面的等高线是什么样子就更是难上加难，所以为了便于理解，还是看一个低维空间中非常简单的样例来帮助理解。

以一个简单的函数 $f(\theta) = 2\theta_1^2 + \theta_2^2$ 为例，假设从 $(1, 1)$ 处开始迭代，取每次更新的 $\eta = 0.1$ ，因为 $\mathrm{d}f(\theta) = 4\theta_1 \mathrm{d}\theta_1 + 2\theta_2 \mathrm{d}\theta_2$ ，所以在第一步梯度为 $(2\theta_1, \theta_2) = (4, 2)$ ，取 $\mathrm{d}\theta^* = -\eta \nabla_{\theta} f$ 可以得到下一步的 θ 。进行 10 次迭代后，得到的点在等值线图上会按照如下图所示的方式移动，可以看到每次更新的 θ 都是垂直于等高线的方向向下走的，逐渐逼近最优解 $(0, 0)$

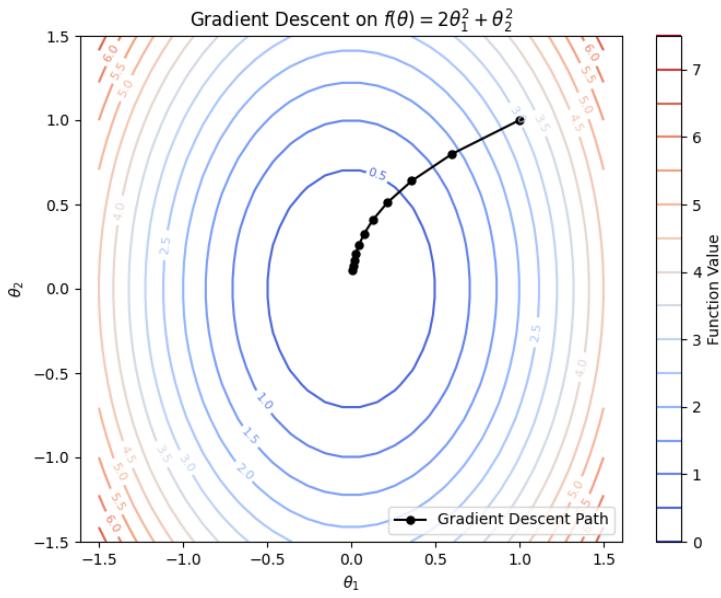


图 17: 梯度下降

不过同时我想也要给出一个错误样例说明学习率 η 调太大会发生什么，假设刚才的例子中我们把 η 调到 0.5，就会发现它直接跳过极小值，跳到了空间中的对边，之后就在 $(-1, 0)$ 和 $(1, 0)$ 之间来回震荡了起来，无法收敛。

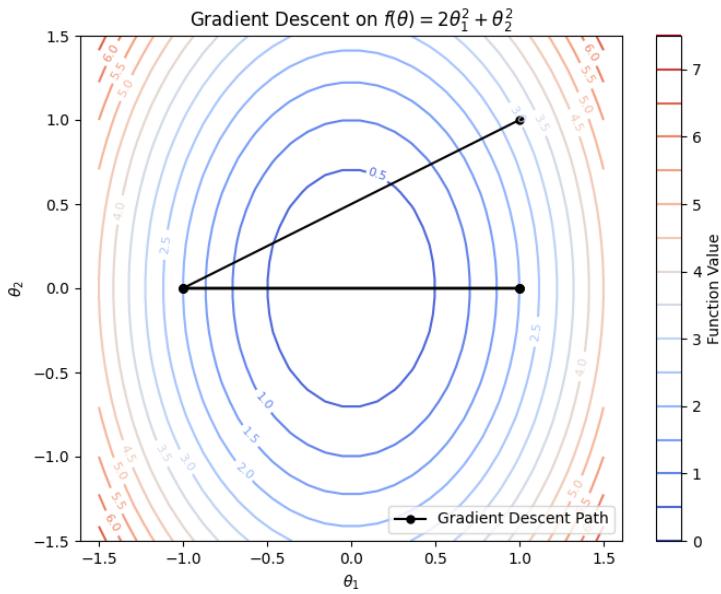


图 18: 学习率设置不当的梯度下降

这个现象在高维空间中也会发生，如果学习率 η 过大，常常会观察到 loss 在某个值附近反复震荡，无法收敛。

在实际有各种不同的梯度下降方法，最简单的是 **随机梯度下降法** (SGD)，它的基本思想是每次只随机抽取部分数据来计算梯度，来更新参数。这样做的好处是每次只需要计算一小部分数据，速度快；坏处是每次计算的梯度并不准确，可能会导致参数在最优解附近震荡。有的为了加速收敛速度，使用 **动量法**，将梯度信息不作为速度，而作为加速度来更新参数，让它能快速冲下坡。还有的使用 **自适应动量法** (Adam)，在每次更新时考虑了历史梯度信息的影响，来调整学习率。这些方法都以各自的方式帮我们找到了更好的参数更新方式，来加速收敛速度。不过动量也不是越大越好，正如学习率不是越大越好一样，过大的动量会导致参数在最优解附近震荡，甚至无法收敛。

推荐阅读

- 至此我们已经基本理解了深度学习的核心思想，它是如何拟合的，又是如何优化的，我想看看 3Blue1Brown 的视频系列很有启发性：

深度学习 Deep Learning - 3Blue1Brown - Bilibili

<https://space.bilibili.com/88461692/lists/1528929?type=series>

- 可以说是梯度下降非常通俗的解释：

用随机梯度下降来优化人生 - 李沐的文章 - 知乎

<https://zhuanlan.zhihu.com/p/414009313>

- 文中没细讲反向传播的过程是如何发生的，如果想详细地理解其中的细节可以阅读：

反向传播算法详解(手算详解传播过程) - 望舒同学的文章 - 知乎

<https://zhuanlan.zhihu.com/p/464268270>

- 关于不同优化方法的选择与优劣比较，这篇文章有很好的总结：

梯度下降法的神经网络容易收敛到局部最优，为什么应用广泛？ - Summer Clover的回答 - 知乎

<https://www.zhihu.com/question/68109802/answer/1677007564>

- 虽然本书中关于具体网络结构的介绍在后面，但是读者如果感兴趣也可以先了解一些很经典的网络架构，笔者觉得这一回答总结的很好：

2024年，深度学习，你心目中的top10算法是什么？ - Russ的回答 - 知乎

<https://www.zhihu.com/question/638660013/answer/3401213045>

动手实践：简单的网络实践

为了让读者对训练神经网络形成一个大概的感知，这里我会用一个极其简单的例子来说明如何训练一个神经网络。而我们的目标就是：拟合一个一元函数 $f(x) = \sin 2x$ ，其中 $x \in [-5, 5]$ 。不过为了实现这个目标，需要先简单讲一下 Python 中的数组操作。

Python 中内置的数组类型是列表，它是动态的，可以在运行时改变长度，这不同于 C 语言中为数组分配一块内存空间并储存指定类型元素。而 Python 列表中的“元素”也实际上是对象的引用。虽然原则上通常推荐用列表存储相同类型的元素⁸⁵，但在运行中，这类强制的要求并不存在。与 C 等语言不同的是，Python 由于其运算符重载机制，其中的四则运算是通过调用某些特定的函数实现的⁸⁶，如果想在列表实现向量运算，在遍历列表的过程中就需要不停地检查对象的类型并调度对应的方法。打个比方，就像是一个工程师虽然知道生产的每一个环节，但拿到一个工件后也要先检查它的类型才能决定用什么工具来加工它。能处理的东西确实是灵活了，但是另一面是效率低下了。而与之对比的是流水线工人，他们面对的工件和工序是完全确定的，不需要检查工件的类型，只要不停地按照固定的流程来加工就可以了，省去了分析的时间，而且也方便同时上很多人来加工⁸⁷。

用 Python 原生的运算来做数值运算显然不够高效，但是 Python 的精髓就在于它的灵活性和可扩展性。可以说如果没有丰富的第三方库，Python 就不可能发展成今天这样一个强大的语言。在数值计算这一块的基石就是 NumPy，自其 2006 年发布 1.0 版本⁸⁸开始，Python 社区内就开始广泛地使用，已经成为了 Python 进行数值计算的 De Facto Standard⁸⁹，重塑了整个生态⁹⁰。不少人都认为 NumPy 设计很好，在运算速度顶级的同时其语法和 Python 的语法风格保持了一致，同时还提供了清晰的接口和丰富的功能。有不少的优质教程可以帮助读者快速上手，[NumPy 官方的初学者指南](#)已经是一份非常清晰的教程，其中引用的部分图片取自[NumPy 的图形化展示](#)，即使单独看这一篇图形化教程也很好。只要用心搜索，中文互联网上亦不乏相关的优质资料，[一些知乎上文章](#)的清晰度与丰富度也已经可以媲美官方教程，让没有经验的读者也可以快速理解 NumPy 的数组接口设计。

⁸⁵通常推荐：写类型标注时通常把列表的类型标记为 `list[T]`，其中 `T` 是元素的类型，例如 `int`。

⁸⁶四则运算：例如 `a + b` 实际上是调用了 `a.__add__(b)` 函数来实现的，仅加法这一个操作就要经过多层函数调用的包装，比 C 语言的加法慢了大概一到两数量级。

⁸⁷多人加工：指对于算术运算，CPU 中有指令可以并行地执行。

⁸⁸1.0 版本：虽然对于 NumPy 来讲是 1.0 版本，不过它是在历史更久远的 Numeric 和 Numarray 两个已有的数值运算库基础上，为了统一数组运算，作为科学计算库 SciPy 的一个核心模块开发的。

⁸⁹事实标准：指虽然没有官方强制规定要用 NumPy，但 Python 社区内几乎所有教程、科研工作、乃至工业界都在使用它（或依赖它的其它模块），它成为了“数组运算”的通用接口标准。

⁹⁰生态：例如 SciPy 和 matplotlib 分别作为科学运算库和绘图库开发早于 NumPy，但是后来它们的底层和接口都改为使用 NumPy。在社区对矩阵运算的庞大需求的推动下，连一向谨慎添加新语法的 Python 官方都专门引入了[PEP 465 提案](#)，早在 Python 3.5 就引入了额外的 `@` 运算符来表示矩阵乘法。

虽有珠玉在前，众多的教程已清晰教会了我们如何使用基础的数组操作，但我还是决定简单地讲一下数组化运算的逻辑。第一条原则是数组中的元素类型是相同的，通常 NumPy 会自动推导⁹¹ Data Type 数据类型（即 dtype 参数）。第二条则是对于加减乘除这样的四则运算和 NumPy 中提供的所有一元函数（例如 sin, cos, exp 等），运算都是逐个进行的。例如两个 64 位浮点数类型的数组四则运算的结果如下。

```
import numpy as np
x = np.array([1, 2, 3], dtype=np.float64)
y = np.array([4, 5, 6], dtype=np.float64)

print(x + y) # [5. 7. 9.]
print(x - y) # [-3. -3. -3.]
print(x * y) # [4. 10. 18.]
print(x / y) # [0.25 0.4 0.5]
```

与经典的向量操作对比，加减法都完全相同。但是读者或许会感到疑惑，向量点积应该返回一个标量，而向量间并未定义除法。乘除法为什么要这么定义呢？因为数组不仅仅是高维空间中的一个向量，它同样可以用于表示一个函数在每一点上的取值⁹²。逐点的乘除法不过是 $f(x) = g(x)h(x)$ 和 $f(x) = g(x)/h(x)$ 在有限个点上的表示。按传统的写法，需要通过一个循环遍历所有的 x 并写入每个点的运算结果，逐元素运算则省去了这样手动遍历的麻烦。不过 * 用于逐元素乘法不代表无法进行想要的的点积运算。在上面的例子中，如果想计算两个向量的点积，只需 $x @ y$ 或者更为明确的 $x.dot(y)$ 就可以了。

了解了 NumPy 的基本原理后，获得 $f(x) = \sin 2x$ 在 $[-5, 5]$ 的样本就非常简单了。`np.linspace` 可以在给定的闭区间上均匀地采样指定的点，例如如果我们希望在 $[-5, 5]$ 上均匀采样 200 个点，导入 numpy 后通过两行代码就能得到预期的样本：

```
import numpy as np
x = np.linspace(-5, 5, num=200)
y = np.sin(2 * x)
```

⁹¹ 自动推导：比如整型数组会变成 int64，浮点数组变成 float64。但也可以手动指定 dtype，以防不符合预期，例如如果数据点是整数但是希望参与 64 位浮点运算，我们也可以手动指定数据为 float64 类型。

⁹² 函数取值：这里的函数是广义上的，例如图像也可以看成是一个关于空间位置 (i, j) 的二元函数，其中 i, j 分别表示一个像素点的行号、列号。乘 mask 就是一个典型的逐像素作用的例子。

数据已经准备好，再下一步则是搭建一个用来拟合的神经网络了。NumPy 的数组操作很优雅，但是要搭建神经网络还少一层包装。回忆训练的过程便会发现更新参数依赖于各个环节的梯度，其中第 k 层的梯度计算需要用到上一层的输出 $x^{(k-1)}$ 和下一层传回的梯度 $g_x^{(k)}$ 。为了计算梯度，数据在正向传播的过程中有必要留下“痕迹”：计算进入下一层并不能直接将它们丢弃，而是等到梯度计算与权重更新之后才能删除。诚然，其中的许多步骤可以手动推导：写出解析的表达式，自己维护一个表用于暂存这些状态，然后手动计算梯度。但是手动计算 Gradient Function 梯度函数 的时代已经过去了，从最便捷的角度来看，PyTorch 中 Automatic Gradient 自动微分 系统的包装已经很完备，只需定义网络结构就能自动计算梯度并优化参数了。学会钻木取火也许对荒野求生作用很大，但现代社会中的我们在学习做饭时已有了方便的工具，没有理由不直接打开家里的天然气灶。

PyTorch 有着类似 NumPy 的数组接口，但多了许多对网络的包装。常见的用法是在自定义模块时继承 `torch.nn.Module`，并重写 Initialization 初始化 方法 `__init__` 和正向计算方法 `forward` 即可，将求导和参数更新的一切交给 PyTorch 的自动微分系统和优化器来处理。例如对于拟合 $f(x) = \sin 2x$ 的例子，三层的网络已经足够： $1 \rightarrow 10 \rightarrow 10 \rightarrow 1$ ⁹³，每一层都是一个带偏置的线性层，在中间两层加上 Leaky ReLU 激活函数并取 $\alpha = 0.1$ ⁹⁴。最后一层的输出是一个标量，表示对 $f(x)$ 的预测值。定义网络的过程如下：

```
import torch
import torch.nn as nn

class SimpleModel(nn.Module):
    def __init__(self):
        super().__init__() # Initialize the parent class
        self.linear1 = nn.Linear(1, 10)
        self.linear2 = nn.Linear(10, 10)
        self.linear3 = nn.Linear(10, 1)
        self.leaky_relu = nn.LeakyReLU(negative_slope=0.1)
    def forward(self, x):
        x = self.leaky_relu(self.linear1(x))
        x = self.leaky_relu(self.linear2(x))
        x = self.linear3(x)
        return x
```

⁹³这样的中间层数量和规律并非是固定的，这里仅是随意的选择。

⁹⁴ α : 含义可以回顾本章第一节激活函数。

最上面的 `SimpleModel(nn.Module)` 表示我们定义的模型继承了 PyTorch 中的模块基类，`nn.Linear` 表示线性层。`forward` 作为模型调用的正向计算过程，通过 `model(x)` 调用时会自动触发 `forward` 的计算。

不过有一点重要的是调用 `model(x)` 时，因为一次训练时需要用到多个样本，所以输入的 `x` 需要是一个二维数组，第一维表示样本数，第二维表示每个样本的特征数。对一元函数来说，特征数就是 1，样本数则是我们在前面准备的 200 个点。为了让 PyTorch 能够正确地处理这个数据，我们需要将它转换为一个二维数组。可以使用 `x.reshape(-1, 1)` 来实现这个功能，其中 -1 表示自动推导这一维的大小，而 1 则表示第二维的大小为 1。

至于损失函数与优化器：使用均方误差作为损失函数，`torch.nn.MSELoss` 就是均方误差的函数，在下面的代码中写作 criterion，即准则。使用 `torch.optim.AdamW` 作为优化器⁹⁵，下面的示例代码中随意地设置了学习率为 0.001。这个学习率一个常用的值。

```
import torch.optim as optim

x = torch.from_numpy(x).float().reshape(-1, 1) # float64 -> float32
y = torch.from_numpy(y).float().reshape(-1, 1)
criterion = nn.MSELoss()
model = SimpleModel()
optimizer = optim.AdamW(model.parameters(), lr=0.001)
model.train()

for _ in range(2000):
    optimizer.zero_grad()
    output = model(x)
    loss = criterion(output, y)
    loss.backward()
    optimizer.step()
```

迭代次数 2000 是简单实验得到的本任务在 0.001 学习率下较合理的训练轮数。其中 `optimizer.zero_grad()` 一步用于清空上一步的梯度信息，`model(x)` 计算正向传播结果，`loss.backward()` 计算反向传播梯度信息，`optimizer.step()` 则是更新参数。

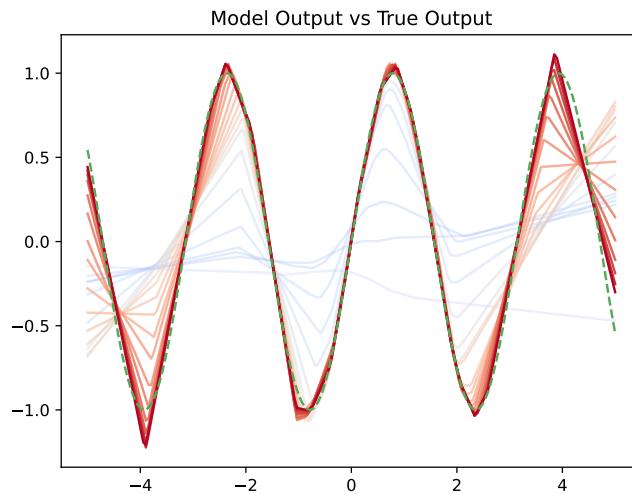
上面就是训练的核心过程。笔者将完整代码放在了 `examples/simple_model.py` 中，并加入了进度条和实时输出损失的功能，方便观察训练过程及模型输出的变化。假设读者已经安装了 Python，在运行代码前，读者还需要安装 NumPy, PyTorch, matplotlib 和

⁹⁵ AdamW：是 Adam 优化器的一个变种，它是一个比较现代的优化器，通常比 SGD 更快收敛。

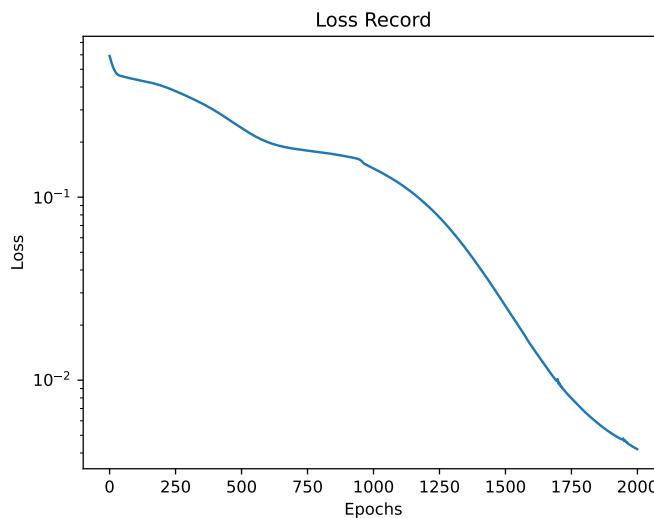
tqdm 四个库。可以在命令行使用下面的命令来安装⁹⁶:

```
pip install numpy torch matplotlib tqdm
```

在运行代码后，训练过程中的模型输出会逐渐接近真实值，一次运行的结果如图所示。图中，由蓝到红的线表明从训练前期到后期的模型输出，虚线表示真值。



反映输出和预期 $f(x) = \sin 2x$ 差异的 loss 变化趋势则如下图所示



⁹⁶提示：如果你在使用虚拟环境，一般不推荐在 base 环境中安装。此外，如果你计划使用 GPU 训练更复杂的模型，推荐安装支持 CUDA 的 PyTorch 版本，详情见前一章 GPU 相关部分。

3.4 如果没人教你怎么做

在前文中，我们已经学会了如何利用神经网络来拟合一个我们期望的函数。但本质上，这依然是在让机器去完成人类已经会做的任务。它只是沿着数据标记者留下的轨迹，一步步将已有的知识“抄写”进模型之中。这种拿来主义的优势显而易见：路径明确、学习高效，知识转化为模型参数后，能极大地提升生产力。

然而，摸着石头过河终究只能走一段路。我们不可能永远依赖于前人的经验去指引未来的方向。历史的车轮滚滚向前，祖辈的智慧无法解决所有的新问题。终有一日，我们必须走出自己的探索之路。人类的发展史，正是一个不断拓荒、积累知识、改造世界的过程。我们当然希望人工智能不仅能继承人类的经验，还能像人类一样，在探索中扩展认知边界，带来新的发现。所谓开拓精神，就是即使前方一片未知，也要义无反顾地前行。

当没有人教人工智能该怎么做时，我们就让它在试错中学会如何做得更好。正如人类不是天生会走路，而是在一次次跌倒中学会了平衡，我们也可以让智能体⁹⁷通过探索和反馈，不断调整策略⁹⁸、优化行动⁹⁹。这，正是 强化学习 的起点。

在学习强化学之前，为了建立起一个清晰的印象，我想有必要先概括一下强化学与前面讲的 深度学习¹⁰⁰ 的关系。深度学习和强化学是两个较为独立的概念。深度学习更多的是考虑如何构建一个模型来预测数据，强化学则是一种思想，关注的是如何从环境中学习到新知识。因此二者既可以单独地使用，又可以双管齐下，正如我在文中一贯强调的：开发的第一步永远是明确需求。任务的情境决定了是否应当使用深度学习或者是否应当使用强化学，从而划分出来四种情况：

- 完全可以使用经典模型就可以解决的任务，例如静态情况下简单的线性回归，或者动态情况下的经典控制理论，它们既不需要深度学习也不需要强化学。
- 在训练信息已知情况下的预测只需要单独使用深度学习。
- 在一些环境有限的较为简单的情况，列出表格就能完成强化学的任务，这时只需要使用强化学。
- 在一些复杂的环境中，强化学让知识从环境中来，深度学习让知识到模型中去。在对环境认识更充分的同时也指导智能体更好地探索环境。

⁹⁷ 智能体：即在环境中做出决策的人工智能。

⁹⁸ 策略：即智能体根据环境决定应该怎么做，它也有可能是按某个给定的概率选择动作。例如当拿到一副牌时根据牌面就知道现在可以打哪些牌，应该优先打什么牌，这样的想法就是策略。

⁹⁹ 行动：智能体按照它的策略选择的实际动作。策略虽然指导了行动，但它并不等同于行动。仍然用打牌来比喻：如果你觉得两张单牌你都不想要，可以“随便”打一张，那么策略是一个概率分布，可能两张牌各 $1/2$ 的概率。行动则是实际打出去的牌，它可以是“随便”选的，但是一次只会打出去一张。

¹⁰⁰ 深度学习：前文的用语是神经网络，但是现在的神经网络通常堆叠很多层，因而常常也称为深度学习。

强化学习听起来就像是一个很复杂的概念，对于初学者而言光是看到“强化”二字就可能感到畏惧，各种各样的术语与学习方法更是让人眼花缭乱，摸不着头脑。数学上，最经典的方式是通过一个用五元组表示的马尔可夫决策过程¹⁰¹来描述的。

关于强化学习的场景描述，上来就是一大段关于马尔可夫链的定义显然会劝退读者，因此我想先打几个比方再真正地引入定义。有这样几个很经典的哲学问题：我是谁，从哪来，到哪去。此外我还应该加上一个问题：一切的意义是什么。这几个问题总的来讲可以与强化学习的场景建模相对应。

首先应该问“我是谁”，换句话说，智能体它本质上是什么呢？我认为是它能对环境做出反应。从一个辩证的角度看来，“存在”就是运动、选择与实践。我的本质在于我的选择和实践，只有在行动中，我才真正体现出我是谁。或许有人会认为智能体当前处在的环境更接近对“存在”的描述。但如前文所述，我认为智能体的本质在于其选择行动的能力。策略是行动的一个概率分布，因此在决策前需要先知道能采取哪些行动。在状态 s 下可以采取的行动通常称作行动空间 $A(s)$ ，智能体在状态 s 下的输出实际上就是 $A(s)$ 上的一个分布。

那么“从哪来”的问题呢？这就是当前所处的状态 s ，打牌时我们能观测到自己手上的牌和已经打出去的牌都是已知的信息，这些信息的总和就可以作为一个状态。¹⁰²

至于“到哪去”，则是说一个状态在进行一个行动后下一个状态是什么。不过它并不一定是注定的结局。即使在相同的状态下，相同的行动也可能暗含随机的因素，两次实验投出的骰子也可能带来不同的结局。从数学上，“到哪去”要如何刻画呢？它其实是给定状态与行动下，下一刻状态的概率分布，即 $P(s'|s, a)$ ，其中 s' 是下一个状态。

但如果把目光从结构转向目标，我们还需要问一个更根本的问题：一切的意义是什么。只有明确了评价指标，才能确定要怎么样优化行为，“明确目标”并不仅仅是开发原则，更是行为学习能否成功的前提。如果在状态 s 下采取行动 a ，下一个状态是 s' ，这样的过程中会获得一个奖励 $R(s, a, s')$ ，这个奖励可以是正的也可以是负的。正的奖励表示这个行动是好的，负的奖励则表示这个行动是不好的。当然也可以是零，表示这个行动是中性的。奖励的定义是非常重要的，它直接影响到智能体的行为。

下棋、打牌、或者其它积分制的游戏中有直接且明确的输赢指标，因此它们是最先被强化学习攻克的。机器人控制中，速度、平滑性、执行难度等因素都需要权衡，如何衡量一个指标已经需要细细思考，多个指标的共同优化便更难设计。大语言模型通过长思维链获得细致的思考和回答是近些年涌现的新方法，但是要量化地评判回答好不好则又是一个

¹⁰¹ 马尔可夫决策过程：这解释起来又是一个复杂的概念，简单来说就是决策只用考虑当前状态，而不需要假设整个环境的背后还有什么“隐藏的”信息。

¹⁰² 观测^{Observation} 和状态^{State}：在严格意义上和状态其实有一些区别，但马尔可夫假设中它们相同。许多优化方法都是在马尔可夫假设下运作的，因此初学者不妨假定它们相同

很困难的问题：数学证明和代码易于形式化验证，它们的评判尚且有个客观的标准，成为了第一批拿来验证大语言模型强化学习的对象，但以艺术创作为代表的许多问题由于因人而异的审美与需求，难以确定什么是“好”，挑战仍然很艰巨。强化学习的有效性并不只取决于算法的聪明程度，更取决于奖励定义是否合理明确——这一点往往比想象中更难做到。

在上面提到的几个基础的问题之外，还有一个从直觉上不太容易直接想到的参数 γ ，
Discount Factor¹⁰³ 称作 折扣因子¹⁰³。它的引入是为了保证回报的收敛性，在决策过程中也有着重要的意义——因为 γ 决定了评价指标更注重当下还是更眼光长远。众所周知钱币会随着通货膨胀而贬值，明天才能拿到的钱比今天拿到的钱更不值钱，因此考虑累计奖励时需要给未来的奖励打折。 $0 \leq \gamma \leq 1$ 控制着折扣的比例，下一次的奖励 r 在这一次看起来就变为了 γr 。一般来讲它的值在 0.9 到 0.99 之间，如果设置太小则过于短视，只见到眼前的利益，可能留下长期的潜在隐患；而如果设置太接近 1，则可能因为对未来的错误判断，被误导走上错误的道路，却在路上自以为选择正确，沦为一种画饼充饥，只是在行进的路上不断地安慰自己这是一时的阵痛与必要的牺牲。

至此建模所需的五个要素已经概括完毕，表示为一个五元组：

$$M = (S, A, P, R, \gamma)$$

其中

- S : 状态空间，表示所有可能的状态集合。
- $A(s)$: 行动空间，表示状态 s 下所有可能的行动集合。
- $P(s'|s, a)$: 状态转移概率，表示在状态 s 下采取行动 a 后转移到状态 s' 的概率。
- $R(s, a, s')$: 奖励函数，表示在状态 s 下采取行动 a 后转移到状态 s' 的奖励。
- γ : 折扣因子，表示未来奖励的折扣率。

这些要素不只是建模的参数，它们决定了智能体如何理解世界、如何行动，也最终决定它是否能成功学习并做出明智的决策。为了有效的学习，需要精心地设计这些元素。

我们希望把复杂的不确定性问题，转化为对单个动作平均效果的评估，这样能更直接地判断一个动作值不值得做。因为每次行动的结果可能不同，我们可以先不管具体结果，而是先看平均表现，这就需要计算期望。特别地，我们注意到在五元组中， P 和 R 都依赖于三个参数：当前状态 s 、当前行动 a 和下一个状态 s' 。由于状态转移是一个概率过程，

¹⁰³折扣因子：在经济学中，也译作贴现因子。

$P(s'|s, a)$ 实际上是一个关于下一个状态 s' 的概率分布。因此，我们可以通过这个概率分布计算期望奖励，得到每个状态-行动对 (s, a) 的期望奖励值。

更为具体地说，仍然采用 R 的符号来表示单步行动的奖励，不过这时定义的是一个二元函数，即

$$R(s, a) = \mathbb{E}_{s' \sim P(s'|s, a)}[R(s, a, s')] = \sum_{s'} \underbrace{P(s'|s, a)}_{\text{每一个结果的概率}} \cdot \underbrace{R(s, a, s')}_{\text{结果对应的奖励}}$$

这样就把 s' 从 R 的参数中消去。我们可以把 $R(s, a)$ 理解为在状态 s 下采取行动 a 的期望奖励，这意味着我们不再等到下一个状态发生后再获得奖励，而是直接把这一项提到了结果 s' 出来之前。有的问题中 s' 随 (s, a) 唯一确定¹⁰⁴，那么 $P(s'|s, a)$ 仅在一个确定的 s' 上取值 1，在其它地方取值 0，这时就省去了对随机变量 s' 求期望的一步， $R(s, a, s')$ 直接退化为 $R(s, a)$ 。

如何衡量一个策略的好坏呢？仍然使用一个期望来描述，既然策略是一个分布

$$\pi(a|s), \quad \text{where } a \in A(s)$$

因此我们可以先定义每个动作的“好坏”程度（质量），再以策略对这些动作加权平均，得到该状态的整体评价。对动作的评估称作质量函数¹⁰⁵，记作 $Q(s, a)$ 。^{Value} 如果暂且不论它的计算方式，对一个状态 s 的好坏估计，即值函数，表示为这一策略下所有可能的行动的加权平均¹⁰⁶：

$$V^\pi(s) = \sum_{a \in A(s)} \pi(a|s) Q^\pi(s, a)$$

读者或许想剥洋葱式地继续追问，动作的质量 $Q(s, a)$ 又是如何计算的呢？这就需要对下一步的状态 s' 进行评估了。第一项是这一个动作带来的即时奖励，第二项是下一个状态的值函数 $V(s')$ 的期望，不过考虑到以后的奖励不如现在的奖励重要，这一步要乘上一个折扣因子 γ 。也就是说：

$$\begin{aligned} Q^\pi(s, a) &= \sum_{s'} P(s'|s, a) [R(s, a, s') + \gamma V^\pi(s')] \\ &= \underbrace{R(s, a)}_{\text{即时回报期望}} + \gamma \underbrace{\sum_{s'} P(s'|s, a) V^\pi(s')}_{\text{下一个状态的值期望}} \end{aligned}$$

¹⁰⁴说明： s' 由 (s, a) 对完全确定时称为确定性环境，例如下围棋时下一步完全由当前局面和走子决定。

¹⁰⁵质量：也称作 Action-Value Function 动作值函数，衡量一个状态-动作对的好坏程度。

¹⁰⁶记号说明： V^π 表示的是在策略 π 下的值函数，不是 V 的 π 次方。这也意味着当策略改变时，这函数的值也随之改变，下面的 $Q^\pi(s, a)$ 含义同理

当然我们可以把一个式子带入另一个式子中得到仅关于 $V^\pi(s)$ 或者仅包含 $Q^\pi(s, a)$ 的表达式。例如将 $V^\pi(s)$ 带入 $Q^\pi(s, a)$ 的表达式从而得到

$$Q^\pi(s, a) = R(s, a) + \gamma \sum_{s'} P(s'|s, a) \left[\sum_{a' \in A(s')} \pi(a'|s') Q^\pi(s', a') \right]$$

而如果反过来，将 $Q^\pi(s, a)$ 带入 $V^\pi(s)$ 的表达式则会得到¹⁰⁷：

$$V^\pi(s) = \sum_{a \in A(s)} \pi(a|s) \left[R(s, a) + \gamma \sum_{s'} P(s'|s, a) V^\pi(s') \right]$$

熟悉编程的读者或许要捏捏鼻子，笃定地说： $Q^\pi(s, a)$ 和 $V^\pi(s)$ 之间的定义互相依赖，这个式子放到任何一个编程语言里面都会因为一层层展开的函数调用而导致无限递归，或在实现中造成死循环似的矛盾，这样一个定义怎么可能成立呢？而我们之前给出的两种代入方式似乎也印证了这个问题：不论从哪个函数开始，最终都会引用回自身。

然而从数学角度看，这样的定义不仅合理，而且结构上非常优美，它妙就妙在 γ 让后期的奖励经折扣变得越来越小：虽然包含无限项，但只要 R 是有界的，它最终就确实能收敛。因为后面的 V 展开为 Q 再展开为 V 时每展开一次都要乘上一个 γ ，遥远的奖励会被越来越小的折扣因子压缩到接近 0。虽“一尺之锤，日取其半，万世不竭”，但越来越小的比例让余项逐渐无足轻重，从结果上是收敛的¹⁰⁸。另外，从操作上并非通过暴力地展开来计算所有的结果，因为指数级增长的情况使得计算并不现实，通过两项之间的管理来递推导出与更新对值（或质量）函数的估计值才是实际的做法。

另外一种情况是与环境的交互每次都是有界的，有一个 Terminal State 终止状态 所组成的空间 T ，当到达一个终止状态 $s_T \in T$ 时环境的演化便不再继续，而是直接给出一个值 $V(s_T)$ ，例如下棋到最终返回一个胜/负/平得到的 +1、0、-1 或其他环境约定的终值，因此也最终会收敛。值得一提的是，这样的值是不依赖于策略 π 的，因为智能体的行动已经走到了终止，在这个状态下不会有新的动作与奖励发生，未来回报为零，因此策略无从作用。

现代的强化学习算法几乎无一例外地将任务分为了两个部分，一个部分是认识环境，一个部分是优化策略，这两个部分的灵活组合衍生出了多种不同的算法。强化学习的算法多种多样，在一节内讲完显然不可能。为了简单起见，先来看看最简单的 Q-Learning 的方法。传统的 Q-Learning 不需要神经网络，对环境的认识直接显式地存储在一个表格中，

¹⁰⁷说明：这个方程与上面关于 $Q^\pi(s, a)$ 的递归表达式都是 Bellman 期望方程的等价形式。前者是关于 Q 的 Bellman 方程，后者是关于 V 的 Bellman 方程。

¹⁰⁸收敛性说明：还记得级数收敛性的读者或许不难推出：只要 $0 \leq \gamma < 1$ 这个级数就会收敛，给定 $|R|$ 的上界 M ，与要求的精度 ε ，完全可以计算出一个截断的 N ，使得“只需要”展开 N 次并丢弃后面的项就能保证估计值 $\hat{Q}^\pi(s, a)$ 与真实值 $Q^\pi(s, a)$ 之间的误差小于 ε 。

选择动作时直接根据查表的结果选择。它只能解决一些很简单的问题，不过作为一个经典算法，它确实也是很纯粹的强化学习。那么，Q-Learning 到底是怎么“学会”行动的呢？

Q-Learning 做的第一个假设就是：空间是有限的。也就是说，状态空间 S 和行动空间 A 都是有限的集合。认识环境的部分变得极为简单：只需要拿一个表¹⁰⁹来储存每个状态-行动对 (s, a) 的质量（后文中称作 Q 值）估计¹¹⁰ $\hat{Q}(s, a)$ ，我们无法一开始就知道每个 (s, a) 的真实价值¹¹¹，因此初始时通常全部初始化为 0，虽然一开始环境是未知的，但在每次行动交互后，算法可以通过经验更新这个表格来形成对环境更加精准的认识。

在这样的假设下策略也很好写出：基于对 Q 值的估计 \hat{Q} 直接选取 Q 值最大的行动¹¹²。也就是说¹¹³：

$$\pi_{\hat{Q}}^*(a|s) = \begin{cases} 1, & \text{if } a = \arg \max_{a' \in A(s)} \hat{Q}(s, a') \\ 0, & \text{otherwise} \end{cases}$$

需要注意，这里的策略是确定性的，即在每个状态下始终选择当前估计价值最高的动作，但是这里为了保持前后文记号的一致性，仍然写作一个分布。不严格地说对于这种确定性的策略¹¹⁴也可以写成

$$\pi_{\hat{Q}}^*(s) = \arg \max_{a' \in A(s)} Q(s, a')$$

在贪心策略下，很容易由 Q 的估计 \hat{Q} 导出对值函数的估计 \hat{V} ¹¹⁵：

$$\begin{aligned} \hat{V}_{\hat{Q}}^*(s) &= \sum_{a \in A(s)} \pi_{\hat{Q}}^*(a|s) \hat{Q}(s, a) \\ &= \hat{Q}(s, a^*), \quad \text{where } a^* = \arg \max_{a' \in A(s)} \hat{Q}(s, a') \\ &= \max_{a' \in A(s)} \hat{Q}(s, a') \end{aligned}$$

也就是说，在当前状态 s 下，我们评估其价值时，重要的只是当前估计下最有价值的那个动作所带来的期望收益，这与策略的贪心性质保持一致。在理想状态下，对于贪心策

¹⁰⁹表：在 Python 中通常使用字典来实现。

¹¹⁰估计：与真值相比可能会有误差。理想情况下随着与环境的持续交互，经验会逐步修正误差，对 Q 值的估计会不断更新并趋近于真实值。但是真实情况下可能会有各种阻碍。

¹¹¹说明：认识环境的关键就在于逐渐修正价值估计，如果一开始都知道真值就完全没必要学了。

¹¹²选取方式：这种方法通常称为贪心选择。不过在探索环境时通常会有一些其它的处理，例如 ε -greedy 或者按照温度选择的策略，这一点会在后文提到。

¹¹³记号说明：上标的星号表示最优， $\pi_{\hat{Q}}^*$ 表明智能体基于 Q 值估计给出其认为的最优策略。

¹¹⁴记号说明：在其它地方你可能会看见这样的表示，这个式子中的 $\pi_{\hat{Q}}^*$ 不表示概率分布，而是从状态直接映射到最优的行动。不过本文中仅此处展示这样的写法，其它地方仍然表示概率分布。

¹¹⁵记号说明：严格来讲应当写作 $V^{\pi_{\hat{Q}}^*}$ ，但从视觉上这样上标与下标的嵌套书写会有些混乱。美观起见，此处省略了 π 并将其上下标直接标注在 V 上。

略下真正的 Q 值，对于每个 (s, a) 对，它都应当满足前文导出的 Bellman 方程（这里将相同的式子重复写一遍）：

$$Q^\pi(s, a) = R(s, a) + \gamma \sum_{s'} P(s'|s, a) \left[\sum_{a' \in A(s')} \pi(a'|s') Q^\pi(s', a') \right]$$

在贪心的策略下，作适当的化简便可以得到¹¹⁶：

$$\begin{aligned} Q^*(s, a) &= R(s, a) + \gamma \sum_{s'} P(s'|s, a) \left[\sum_{a' \in A(s')} \pi^*(a'|s') Q^*(s', a') \right] \\ &= R(s, a) + \gamma \sum_{s'} P(s'|s, a) Q^*(s', a^*) \quad \text{where } a \in A(s) \\ &= R(s, a) + \gamma \sum_{s'} P(s'|s, a) \max_{a' \in A(s')} Q^*(s', a') \end{aligned}$$

然而对于估计值 \hat{Q} ，上述等式通常并不成立，因为它尚未收敛于真正的 Q^* ，存在一定误差。即实践中

$$\hat{Q}(s, a) \neq R(s, a) + \gamma \sum_{s'} P(s'|s, a) \max_{a' \in A(s')} \hat{Q}(s', a')$$

总是会有一个误差¹¹⁷，记作 $\delta_{\hat{Q}}(s, a)$ ，即¹¹⁸：

$$\begin{aligned} \delta_{\hat{Q}}(s, a) &= \text{RHS} - \text{LHS} \\ &= R(s, a) + \underbrace{\gamma \sum_{s'} P(s'|s, a) \max_{a' \in A(s')} \hat{Q}(s', a')}_{\text{经过单步奖励修正的 } Q \text{ 值估计}} - \underbrace{\hat{Q}(s, a)}_{Q \text{ 值的直接估计}} \end{aligned}$$

而我们期待的是让 \hat{Q} 逐渐收敛到 Q^* ，因此这个不等号左右两侧的差值 $\delta_{\hat{Q}}(s, a)$ 也应该逐渐减小。强化学习的核心就是用这一误差 $\delta_{\hat{Q}}(s, a)$ 作为更新的信号，来修正对 Q 值的估计。很显然地，下标的 \hat{Q} 表明这一误差依赖于当前的 Q 值估计，因此在估计量 \hat{Q} 变化时， $\delta_{\hat{Q}}(s, a)$ 也会随之变化。

这已经很接近 Q 值估计的更新了，然而仍然有一个问题：即使是将式子化为了关于 \hat{Q} 的形式， $\delta_{\hat{Q}}(s, a)$ 的真值可能仍然未知，因为它依赖于 P 和 R 的真实值，而我们并不知道真值。除非我们拥有对环境的完整建模，否则状态的转移只能通过自己尝试来获得，所

¹¹⁶说明：这一方程通常称为 Bellman 最优性方程。

¹¹⁷误差：通常称作 Bellman 误差、Q 误差、或者 Temporal Difference Error 时序差分误差 (TD Error)。

¹¹⁸记号说明：LHS 表示 Left-Hand Side 等式左侧，RHS 表示 Right-Hand Side 等式右侧。

幸我们可以记录下每一步的当前状态 s 、当前行动 a 、下一个状态 s' 和即时奖励 r ，这给了我们一个退而求其次的方法，即考虑在 $s \xrightarrow{a} s'$ 时的误差：

$$\delta_{\hat{Q}}(s, a, s', r) = r + \gamma \max_{a' \in A(s')} \hat{Q}(s', a') - \hat{Q}(s, a)$$

虽然它和 $\delta_Q(s, a)$ 之间有着很明显的差别，但是在对 s' 求期望的意义上，至少有

$$\mathbb{E}_{s' \sim P(s'|s, a)} [\delta_{\hat{Q}}(s, a, s', r)] = \delta_{\hat{Q}}(s, a)$$

回忆一下我们学过的参数更新过程：在深度学习中，以负梯度作为更新信号时，学习率 η 给出更新：

$$\theta^{\text{new}} = \theta^{\text{old}} - \eta \cdot \text{gradient}$$

正如深度学习中使用学习率控制梯度下降过程一样，Q-Learning 中同样使用一个学习率 α 来控制更新的幅度¹¹⁹。假设我们已经有了一些 (s, a, s', r) 的样本，我们就能以 $\delta_{\hat{Q}}(s, a, s', r)$ 为更新信号，修正对 Q 值的估计：

$$\hat{Q}^{\text{new}}(s, a) = \hat{Q}^{\text{old}}(s, a) + \alpha \cdot \delta_{\hat{Q}}(s, a, s', r)$$

从某种意义上，这颇有一种“自己拟合自己”的感觉： Q 值估计的更新依赖于当前的 Q 值估计。读者或许想问：如果原来的 Q 值估计 $\hat{Q}^{\text{old}}(s, a)$ 就是不靠谱，那更新后不还是错的吗？这个话既不完全对，也不完全是错的。确实，这样的更新会将原来的误差带到新的估计中，因此原本的估计不好确实会影响到新的估计。但是一方面，这样的误差会随着 γ 的引入而衰减，另外， r 直接地引入了当前的奖励，虽然修正的估计值也不准确，但是总的来说会变得更准一些。

这个过程可以说充满了随机因素：在每次更新时， \hat{Q} 的估计会依赖于当前的 Q 值估计，这本身就带有一定的误差；而在每次更新时， $\delta_{\hat{Q}}(s, a, s', r)$ 不仅依赖于 (s, a) 对，还依赖于采样探索时的下一个状态 s' 和即时奖励 r ，如果状态转移本身就带有随机性，因而 s' 与 r 本身也会有随机性， $V(s')$ 与 r 本身就有可能波动，而非随着 (s, a) 而唯一确定，这种波动称为 Sampling Error 采样误差。因此 α 的选择非常重要，如果简单粗暴地令 $\alpha = 1$ ，那么把式子展开就会得到¹²⁰

$$\hat{Q}^{\text{new}}(s, a) = r + \gamma \max_{a' \in A(s')} \hat{Q}^{\text{old}}(s', a')$$

也就是直接拿修正的 Q 值估计来替换原来的 Q 值估计，但是一旦 r 存在随机性或者探索的路径发生了一些变化，这样的做法就会导致 Q 值的估计在每次更新时都发生剧烈的震荡，而无法收敛。反之如果 α 过小，更新的幅度就会过小，虽然不会发生剧烈的震

¹¹⁹记号说明：Q-Learning 中更习惯使用 α 而非前文使用的 η 来表示 Q 表的学习率。

¹²⁰说明：读者不难将 α 带入自行推导出这个式子。不过这么更新是一个错误示范。

荡，但是收敛速度会变得很慢。一般来讲， α 的值在 0.1 到 0.5 之间比较合适。这样它既能较快地保证收敛，又能起到一定的滤波作用，避免过大的震荡。这种做法的历史根源向上追溯到经典控制中的卡尔曼滤波器，通过对当前的估计值和新的观测值进行加权平均来修正对状态的估计，基于增益¹²¹来控制更新的幅度。

如果要处理井字棋或者走迷宫这样的简单场景，需要的元素 (S, A, P, R, γ) 都很容易定义出来，而 S 与每个状态下的 $A(s)$ 都有限，参数更新的规则又有了，看起来已经可以直接上手了。不过如果真正试一下就会发现一个问题：一旦探索到某条路径上的 $\hat{Q} > 0$ ，它可能直接导致在后续的探索中，智能体总是选择这条路径，而忽略了其它可能更好的路径。就像是一个挖矿者，在走了一段路后发现了一个闪亮的宝箱，便急于停止挖掘，错过了藏在更深处的真正的钻石。就像下面这张经典的图一样：

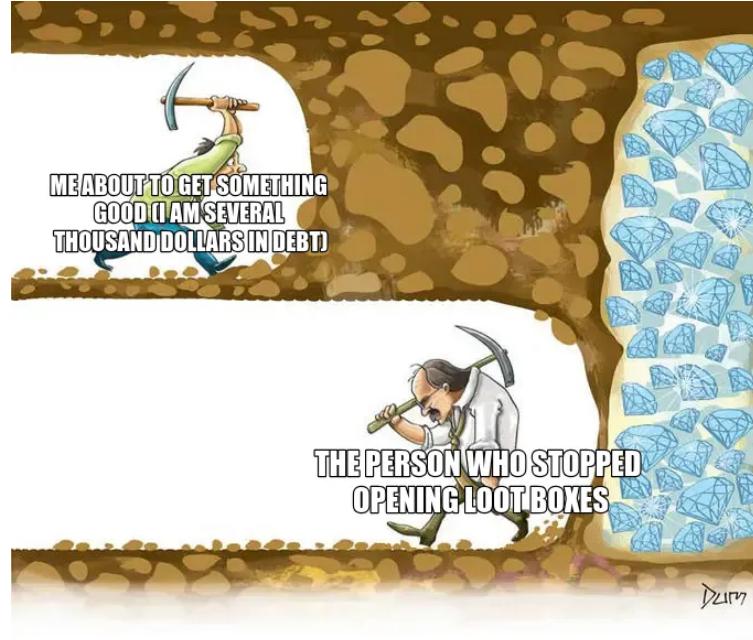


图 19: 挖掘钻石的人错过了更好的机会

图源: [Kapwing](#)

这意味着守成不变的贪心策略并不适合于探索未知的环境，路径依赖带来的可能是不思进取。一味地效仿已经被验证过的路径，可能会陷入局部最优解而无法跳出，如果仅仅因为可能的风险而反对变革，便会堵死更好的道路，最终掉进局部最优解里打转的周期之中。为了避免这种情况，需要在探索与利用之间取得平衡，既要利用当前的知识来选择更好的行动，又要适时地探索新的路径。这就称为 Exploration 与 Exploitation 之间的平衡：在每次行动时，智能体以一个小概率 ε 随机选择一个动作，而不是总是选择当前估计值最大的动

¹²¹增益：在卡尔曼滤波器中，增益是一个权重因子，用于平衡当前估计和新观测值之间的影响。

作。这样的扰动或许可以揭示一条更好的道路，避免陷入局部最优解。 ε 通常会随着时间的推移而逐渐减小，这样一开始应当多探索一些新的路径，增长对环境规律的认识。而后期作为一个成熟稳重的智能体，则更多地利用当前的知识来选择更好的行动。

经验利用与探索之间的平衡是强化学习中的一个重要问题，因为一方面我们希望获得最优的策略，另一方面我们又需要在有限的时间内探索足够多的状态和行动，以便获得足够的信息来更为全面地评估状态与行动的价值。像上面这样以 ε -greedy 的方式来平衡探索与利用的策略是一种常用方法，其它的方法还有基于 Temperature ¹²²、 $\text{Upper Confidence Bound}$ (UCB)¹²³ 等的方法。在 Q-Learning 中，因为 $\pi_{\hat{Q}}^*$ 是一个确定性的策略，因此需要人为地引入一个随机性来平衡探索与利用，采样时的实际策略与当前计算出的理想策略 $\pi_{\hat{Q}}^*$ 不同，这称为离线策略。而在一些其它的强化学习算法¹²⁴中，策略本身就是一个概率分布，因此通常不需要引入额外的探索机制，而是直接从 π 中采样动作，这称为在线策略。

至此，有了参数更新机制，又有了探索与利用的平衡，Q-Learning 算法就可以完整地写出来了。算法的核心思想是：在每次行动时，智能体根据当前的 Q 值估计 \hat{Q} 选择一个动作 a ，然后与环境交互得到下一个状态 s' 和即时奖励 r ，并用 $\delta_{\hat{Q}}(s, a, s', r)$ 来修正对 Q 值的估计。不过俗话说“温故而知新”，智能体的学习也是一样： (s, a, s', r) 的样本并不是用过一次就丢掉的，智能体可以将它们存储在一个经验池（也称作 Replay Buffer ）中，更新参数时再从中读取。在探索多次后会反复使用这些样本来更新 Q 值估计，就像学习时我们会回忆过往的尝试经历，并使用它们进一步巩固自己的认识，这一过程称作 $\text{Experience Replay Mechanism}$ 。需要注意的是，在更新的过程中，每次的更新信息 $\delta_{\hat{Q}}(s, a, s', r)$ 也会随着 \hat{Q} 的变化而变化，因此每次更新时都需要重新计算。

不过正如在引入 Q-Learning 时所提到的，Q-Learning 只能解决一些简单的问题，面对复杂的环境时，一个表便完全无法存储所有的 (s, a) 对了。以下围棋为例，棋盘上的格点是有限的，因此每个局面 s 下的行动空间 $A(s)$ 也是有限的。但即便整个状态空间 S 是有限的，其大小却令实际的储存不切实际。对于 19×19 的棋盘，每个格点上可以放置黑子、白子或者为空，状态空间大小的上界在 $3^{19 \times 19}$ ，也就是 10^{172} 的数量级，任何的计算机都无法存储这样一个表格。对于这种情况，可以改用一个神经网络来近似地表示 Q 值估计 \hat{Q} ，而更新的过程也改为对 $\delta_{\hat{Q}}(s, a, s', r)$ 平方和的梯度下降，不过前提是能够抽离出某种“规律”，神经网络才有可能学习到较好的近似函数。这种使用神

¹²² 温度机制：在每次行动时，智能体以一个温度参数 τ 来控制选择动作的概率分布，这种选择方式称为 $\text{Boltzmann Exploration}$ ，与粒子在不同能级上的分布有关。

¹²³ 上置信界：与置信区间的估计有关，它是多臂老虎机（一种赌博机）问题解出的最优结果，也有着良好的数学基础，许多基于树搜索的方法都依赖于对上置信界的估计。

¹²⁴ 其它的强化学习算法：例如 Actor-Critic 等方法

Deep Q-Learning

经网络近似 $Q(s, a)$ 的方法称为深度 Q 学习，它基本上是 Q-Learning 直接的神经网络变种。

如果说用神经网络近似 Q 表还能在一定程度上弥补表格空间不够的问题，仍然能用最大化 $\hat{Q}(s, a)$ 选出策略 a^* 。那么更为甚者，当状态空间 S 与每个状态下的行动空间 $A(s)$ 都是无限、甚至是连续空间时¹²⁵，不但表格方法失效了，策略也不再能简单地通过查表来找到最大值。这时，在策略部分也需要引入深度学习了，输入一个状态 s ，输出通常包含两个部分：

1. 值函数：由于状态空间的无限性， Q 的完整估计 \hat{Q} 也无法写出，所以退而求其次，计算的是基于值函数的估计 $V(s)$ 。
2. 策略函数：输出一个概率分布 $\pi(a|s)$ ，表示在当前状态下选择每个动作的概率。

这两个部分通常称为评论者与行动者¹²⁶。

评论者的目标通常仍然是最小化 δ 的平方和，来修正对值函数的估计，不过这里既然不是基于 \hat{Q} 的估计了，不能还是 $\delta_{\hat{Q}}(s, a, s', r)$ 了，这一误差的形式如何便有待商榷。当基于 Q 值估计时，我们借助 Q 值的 Bellman 方程来定义了误差。为了定义基于 V 值的误差，我们需要借助 V 的 Bellman 方程来定义，前文中的形式为：

$$V^\pi(s) = \sum_{a \in A(s)} \pi(a|s) \left[R(s, a) + \gamma \sum_{s'} P(s'|s, a) V^\pi(s') \right]$$

然而对于连续的情况，我们不得不把求和改为积分，形式上大约是这样：

$$V^\pi(s) = \int_{a \in A(s)} \pi(a|s) \left[R(s, a) + \gamma \int_{s'} P(s'|s, a) V^\pi(s') ds' \right] da$$

因此对于对 V 的估计 \hat{V} ，我们仍然可以定义一个误差 $\delta_{\hat{V}}(s)$ ，即

$$\begin{aligned} \delta_{\hat{V}}(s) &= \text{RHS} - \text{LHS} \\ &= \int_{a \in A(s)} \pi(a|s) \left[R(s, a) + \gamma \int_{s'} P(s'|s, a) V^\pi(s') ds' \right] da - \hat{V}(s) \end{aligned}$$

与之前相同的是，我们并不知道 P 和 R 的真实值，因此真实的 $\delta_{\hat{V}}(s)$ 也无从计算，只能粗糙地用样本上的采样来近似真值，假设有样本 (s, a, s', r) ，将 a 与 s' 都用单点分布代替，并在样本上化简为：

$$\delta_{\hat{V}}(s, a, s', r) = r + \gamma \hat{V}(s') - \hat{V}(s)$$

¹²⁵ 连续行动空间：例如控制机器人的关节角度。

¹²⁶ 评论者与行动者：评论者负责评估当前的状态与行动的价值，行动者负责选择当前的行动。不过笔者更喜欢的叫法是 **Value Head** 与 **Policy Head**，这一描述从含义上或许更为清晰。

总的来说，先用样本的单步回报信息修正，再估计估计中含有的误差，这一思想与前面计算 Q 值的 Bellman 误差思路相同。注意到这里对 a 和 s' 都做了采样处理，这意味着这一过程中发生了两次近似，一次是用单点的 a 近似策略下的期望，另一次是用单点的 s' 近似 P 和 R 。好处是我们不需要知道环境的精确模型，仅通过经验采样即可进行学习¹²⁷。但是坏处也很显然：它引入的误差更大，因此同时优化策略与值函数的算法通常会比单独优化其中一个要复杂得多，许多算法就是为了保证这一过程的稳定性与收敛性而提出的。

虽然 Q-Learning 中的贪心策略可以直接用 \hat{Q} 来选择最优的行动，但在连续的空间中，通常意义上的最大值并不现实，因此只能再退一步，试图最大化 $V^\pi(s)$ 。而行动者则是需要优化策略来提高 $V^\pi(s)$ ，回忆 V^π 的计算公式

$$V^\pi(s) = \sum_{a \in A(s)} \pi(a|s) Q^\pi(s, a)$$

在连续情况下变为了

$$V^\pi(s) = \int_{a \in A(s)} \pi(a|s) Q^\pi(s, a) da$$

而对动作采样近似就得到了这一项的贡献为（贡献会随着概率，即重要性 $\pi(a|s)$ 而变化）：

$$\pi(a|s)[r + \gamma \hat{V}^\pi(s')]$$

也就是说在优化值估计的同时，我们希望通过增大这一项的贡献来增大实际的期望 $V^\pi(s)$ 。而在主流强化学习算法中， L 中的后一项从 Q 的估计改为一种相对优势估计，它考虑的是差值而非值本身，例如在 PPO 中¹²⁸，目标函数中用策略加权的项就不是 Q 值，而是基于优势函数 $A(s, a) = Q(s, a) - V(s)$ ，使用 Bellman 误差的加权和来给出相对优势的估计 \hat{A} 用于评估当前策略相对旧策略的改进效果，实践证明这极大提高了稳定性。同时基于一种比较的思想，最大化的策略一部分还要除以 $\pi^{\text{old}}(a|s)$ ，来鼓励相对当前的策略进行优化。

从信息流动的角度来看，这种二元的结构引发的信息流差不多是这样的：为了减少对值函数的估计误差，评论者会根据下一步的值函数估计 $\hat{V}(s')$ 与回报，通过在值函数的 loss¹²⁹ 中设置一个误差的项来修正当前对 V 值的估计 $\hat{Q}(s)$ ，信息便从环境流向评论者；而行动者则会根据当前的 Q 值估计 \hat{Q} 来优化 π ，在策略函数的 loss¹³⁰ 中设置一个希望极

¹²⁷说明：这种方法称为无模型的方法，“模型”指的是对环境的精确建模。

¹²⁸Proximal Policy Optimization (PPO)：一种策略优化算法，由 OpenAI 的 John Schulman 等人在 2017 年提出。具体的优化方法在此处不再叙述，读者可以自行翻阅[原论文](#)或查找网上的其它相关解释。

¹²⁹值函数的 loss：即 δ 的平方平均值。

¹³⁰策略函数的 loss：通常是负的期望值函数或者期望优势。

大化的目标（为统一为最小化问题，通常将其负号加入 loss 中）。信息进一步从评论者流向行动者。智能体在修改策略的同时探索新的环境，新的信息则再次流向回报，形成一个闭环，推动着信息随着策略的优化不断从环境流向模型。

与通常的深度学习不同的是，这一过程中两个部分的的 loss 函数都并不是单向的信息接受器，而是“有进有出”的。策略引来的新探索将新计算出的偏差注入值的 loss，评论者的更新则从其中“取出”新的信息，并将计算出来的策略改善信息注入策略的 loss。这一过程中 loss 就像一个外卖柜，信息的传入和传出在训练过程中不断发生。loss 大时，可能表明无法收敛，但是有时又表明信息流正在快速运行；而 loss 小的时候，可能表明模型已经收敛，但是有时又表明信息流动缓慢。

放眼如今的强化学习研究界，除了 Q-Learning 之外，基于问题的各种条件与假设强化学习算法也有了各种各样的变种与改进。这一节之内无法一一列举，仅从理念上阐释了最基础的强化学习算法 Q-Learning 的基本思想。在此过程中，一些基础的概念已经基本上介绍完毕，其它的变种只是在某种对环境的假设下做文章，无非是优化方式或目标设置不同。真正掌握核心机制后，面对新方法就无需害怕，而能通过把握环境特征、面向的问题与信息流的方向快速理解其本质。不过我一向认为良好的直觉与理解是学习强化学习的基础，各个方法的细节反倒是在需要时再临时查阅的，只是希望读者在阅读完这一节后，能够理解核心概念，清晰地认识强化学习的基本思想。

回顾在本章中讲过的内容，信息非常多，涵盖了深度学习与强化学习中的关键思想与方法。那么在这一章的最后，我们照例来小结如下：

- 最开始，我们从函数拟合引入。学习神经网络基本上需要牢牢记住一个点，它的本质就是为了拟合，从而找到数据的规律。
- 神经网络的基本模型可以概括为一个从输入到输出计算，再通过偏差矫正的循环，不过输入和输出还需要做一些编码与解码处理。
- 神经网络的基本模块是一些矩阵运算与非线性激活函数的组合，一层层叠加起来就可以做到简单的函数无法拟合的复杂函数。
- 激活函数通过引入非线性，给了神经网络强大的表达能力，基本的逻辑函数都能借助激活函数来表示就表明它至少能够完成逻辑运算能够解决的问题。
- 神经网络的训练过程就是通过反向传播算法来计算梯度，进而更新参数，来最小化损失函数，梯度下降算法是常用的优化方法。就像下山一样，梯度帮助函数找到极小值，不过也可能陷入局部极小，学习率与优化器的选取也很重要。
- 梯度下降中反向传播的计算和正向的计算刚好反过来，信息流原本是从输入与参数

“汇聚”到输出，反向传播则是从输出“分散”到输入与每一个参数之中，描述了它们的贡献。

- 最后一节中，我们介绍了强化学习的基本思想，虽然强化学习不一定依赖于网络，但是也是机器学习的一个重要组成部分。与神经网络相比，它更注重从环境中获取样本从而得到信息的过程。
- 通常会通过五个要素来建模环境。“我是谁、从哪来、到哪去”可以解释动作空间、状态空间与转移函数，奖励函数是环境的反馈，折扣因子则是对未来奖励的折现，表明未来的奖励更不重要。
- 在环境的基础上，通过 V 或者 Q 可以列出状态或者动作的价值函数，来评估当前的状态或者动作的好坏。并通过 Bellman 方程来递归给出表达式。
- 总的来说强化学习会分为认识环境与策略优化两个部分，前者是通过奖励来学习环境的规律，后者则是通过对环境的认识来优化策略。
- 我们以 Q-Learning 为一个简单的例子，体会了强化学习的基本思想，它的策略非常简单直接，认识环境直接通过记录在 Q 表中，更新时通过对 Q 表的更新来优化策略。Bellman 误差 δ 描述了 Q 表和预期的差异，并作为指导信息修正 Q 表。
- 除此之外，还需要探索机制与经验回放机制来平衡探索与利用，避免陷入局部最优解，并增加样本的利用率。
- 如果状态空间无限但是动作空间有限，仍然可以使用 Q-Learning 的思想，只不过需要用神经网络来近似 Q 表，更新时通过对 δ 的平方和进行梯度下降来修正。
- 如果连动作空间也是无限的，那么价值和策略都需要用神经网络来近似，评论者与行动者分别负责对值函数与策略函数的估计与优化。评论者通过对值函数的误差来修正对值函数的估计，行动者则通过对值函数的估计来优化策略。
- 这种二元结构的运作方式是一个循环，获取到的信息会增加对环境的认识（即改进值函数的估计），而对环境的认识又会影响到策略的优化（即改进策略函数的估计），策略的优化进一步引发新的探索，带来新的信息，最终在理想状况下会越学越好。
- 强化学习的研究发展到今天，已经有了许多不同的变种与改进，只需要理解环境特征、它面向的问题和信息流的情况，就可以对比不同的算法来理解它们的本质。

推荐阅读

- 强化学习其实就一种控制：
为什么自动化所很多做强化学习的课题组？ - 消融ball的回答 - 知乎
<https://www.zhihu.com/question/647238131/answer/3424550835>
- 于是你或许会有这样的疑问，感觉强化学习好像许多时候不如传统方法？这种想法是对的，但是强化学习本质上是在很弱的前提下（许多时候只能记录状态，行动和奖励）求解问题，详细解释可以看：
为什么我总感觉强化学习不是真的人工智能？ - Lemon的回答 - 知乎
<https://www.zhihu.com/question/542991276/answer/2580779607>
- 不过强化学习除了看起来有用的方法实际上很难落地，因此许多时候还应用受限，一些关于它的吐槽可以看回答：
为什么说强化学习在近年不会被广泛应用？ - 王源的回答 - 知乎
<https://www.zhihu.com/question/404471029/answer/1590652261>
- 前面讲了很多直觉上的理解，不过想要深入研究强化学习的读者仅仅从直觉上理解是不够的，更细的探讨可以看《强化学习的数学原理》这份教程：
强化学习的数学原理】课程：从零开始到透彻理解（完结） - 西湖大学 WindyLab
- Bilibili
<https://www.bilibili.com/video/BV1sd4y167NS>

4 泛化性：一个矛盾

4.1 过拟合与欠拟合

4.2 网络的大小好像小于训练数据？哪来的泛化性

4.3 训练好像被卡住了——香农极限

5 你能猜到一句话接下来要说什么？

5.1 什么是“废话”？

5.2 jpeg虽然有损，但为什么说是极其成功的？

5.3 熵与压缩

5.4 马尔可夫链

6 压缩即智能

6.1 你是如何看出对面的人心情怎么样的？

6.2 压缩的极限——区分能力的边界

6.3 从母语词汇看对事物的认识

6.4 压缩的本质

7 潜空间：更适合机器人的编码方式

7.1 潜空间是什么？

7.2 高维空间的维数灾难

7.3 “空空”的空间的另一面——维数远不是储存能力的极限

8 但是，代价是什么：可解释性的地狱

8.1 想想什么是“解释”？

8.2 神经网络不能很好地被解释

9 再论网络结构

9.1 全连接网络

9.2 循环神经网络

9.3 卷积神经网络

9.4 深度学习与残差链接

9.5 transformer

9.6 自编码器与扩散模型

9.7 仿人的架构——专家模型

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？