为什么是神经网络

神经网络：一个大的函数

神经网络
相比于 Neural Network 如何实现其功能，读者或许更想问的是：为什么要用神经网络？现有的神经网络为什么用了这些方法？对于这一类问题，一个现实的回答是：机器学习是高度以实用为导向的，实验显示这样做效果更好。在现实中，我们往往要解决各种各样的问题，人类开发者以手写每一行代码创造了各种各样的程序，自动化地解决了许多问题。但很多问题难以在有限的时间内找到确定性的解决方案，例如识别图片中的物体、识别语音、自然语言处理等等。它们有一个共同点：输入的信息量巨大、关系复杂，难以用确定的规则来描述。手动规定像素范围来判断物体类型，或用固定的规则来解析自然语言显然并不现实。因此人们自然要问有没有更加自动化、灵活、智能的方法来一劳永逸地解决这些问题。人工智能的概念就此提出，人们希望让机器自己学习知识来解决问题。

通用人工智能
虽然目前人类仍然很难说摸到了 Artificial General Intelligence[1]的边界，但人工智能已然在许多问题上取得了巨大成就，走出了 20 世纪末 21 世纪初被大众认为是"伪科学"的寒冬。经过深度残差网络在图像识别的重大突破、AlphaGo 学会下围棋、Transformer 在翻译比赛取得优异成绩并引来一波生成式模型的热潮等等，人工智能就这样走向了时代的焦点。但是如果要问：为什么它这么成功？最直接的回答仍是：It works.

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

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

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

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

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。

[1]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。

因此许多问题都可以归结为一个更加狭义的、数值拟合意义上的函数拟合问题。一个 Encoder（编码器）将原始输入变为向量这种易于处理的形式。而对于函数的原始输出，可以通过一个 Decoder（解码器）将数值构成的向量变为我们想要的输出。

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

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

TODO: 这里需要一个图，展示了神经网络的五个要素及其关系

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

在这里我想简单讲讲使用矩阵运算的原因。在第一章中我们已经简单地学习了矩阵运算的基本知识，它本质上是正比例函数在向量空间中的推广，只是 $y = kx$ 中的斜率变成了一个个从输入 x_j 连接到输出 y_i 的权重 w_{ij}。从行看过去，它反映了输出的每个分量（或称为特征）是如何由输入的每个分量线性组合而成的。而从列看过去，它表明了输入的每个分量是如何影响输出的。就像一次函数有一个常数项一样，矩阵运算也有一个偏置项 b，运算的总体结构是 $y = wx + b$。从代数上看，它运算简单[2]，而从分析上看，它的输出变化光滑，容易求导[3]。

相关阅读

这篇文章讲述了神经网络的起源：
如何简单形象又有趣地讲解神经网络是什么？ - 佳人李大花的回答 - 知乎
https://www.zhihu.com/question/22553761/answer/3359939138

读者或许会好奇所谓的万能逼近定理需要是如何能逼近给定函数的，这篇回答的解释不错：
神经网络的万能逼近定理已经发展到什么地步了？ - 牛油果博士的回答 - 知乎
https://www.zhihu.com/question/347654789/answer/1534866932

这一问题下有关于神经网络"涌现"出新的现象的讨论，对其机理感兴趣的读者也可以想想背后的原因：
如果神经网络规模足够大，会产生智能吗？ - 知乎
https://www.zhihu.com/question/408690594

[2] 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。

[3] 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。

“激活函数与非线性”

将 $y = wx + b$ 作为一次函数的类比应该足以说明它是很简单的一类函数。但是正如一次函数的复合 $y = w_2(w_1x + b_1) + b_2 = w_2w_1x + (w_2b_1 + b_2)$ 仍然是一次函数一样，如果仅仅沉浸在矩阵运算中，我们便永远无法表达那些复杂的函数。举个最简单的例子，我们甚至无法表示输入的绝对值 $y = |x|$。因此我们需要在模型的结构中加点"非线性"，让它不仅仅局限于简单的加减乘除，专业的说法称之为 Activation Function（激活函数）。激活函数直接作用在每个特征上，而且函数本身通常是固定的[4]，且总体通常呈现递增的趋势。

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

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

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

图像是这样的：

TODO: 这里需要一个图，展示 ReLU 函数的图像

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

而它的导数也非常简单：

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

读者或许会关心，那 0 这一点不可导要怎么办？其实关系不大，因为一个小数几乎不可能[5]在训练中恰好落在 0 上。即使有，也可以任意地选择一个值，例如 0 或者 1[6]。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：

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

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

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

写成矩阵的形式就有

---

[4] 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。

[5] 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。

[6] 0 处的导数：PyTorch 通常选择 0

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

遂可以写成 $y = w_2 \ \mathrm{ReLU}(w_1 x + b_1) + b_2$。我认为，把这件事作为一个 toy case[7]想明白多少可以帮助理解神经网络。把矩阵的每个权重都画出来就是这样了：

TODO: 这里需要一个图，展示神经网络表示|x|的结构

这看起来很简单，读者可能想问：还能不能再给力一点，看看更复杂的情况呢？当然可以。不过在看之前先抛出两个思考题：
1. 试着用线性函数和 ReLU 函数表示 $y = \max\{x_1, x_2\}$，并画出它的神经网络结构图。
2. 线性函数和 ReLU 的组合不能表示什么函数呢？

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

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

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

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

变量 x_1 没有经过统一的隐藏层，而是跳过中间，直接连接到了输出层。显然就不能用一致的 $\mathrm{ReLU}(wx + b)$ 的形式来表示了，而是要单独开一个通道来处理。而我们使用神经网络的目的本来就是用一致的方式来处理所有的输入，所以这样的表示方式并不优雅[8]。

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

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

这一式子中的三个分量提出来，便可以得到

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

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：

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

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

---

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

[8] 并不优雅：与之对比，在深层神经网络网络中通常会引入看起来有些像这里的 Shortcut（跳连接） Connection 结构，由此引出 Residual（残差网络） Network[9]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。

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

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

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

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[10]，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，$f(x) \le c\|x\|$。从而增长速度有限，无法表示指数函数或者高次的多项式函数。

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

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

图像是这样的：

TODO: 这里需要一个图，展示分段线性函数近似光滑函数

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

另外再提一嘴其它的激活函数。Sigmoid 函数[11]是一个 S 型函数，定义为

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

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

TODO: 这里需要一个图，展示 Sigmoid 函数的图像

tanh 函数是双曲正切函数，其定义为

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

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

[10] 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。

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

TODO: 这里需要一个图，展示 tanh 函数的图像

早期的研究中，它们出现在许多生物学的研究中，可以描述生物神经元的激活或者极化程度，于是人工神经网络出于仿生的考虑也使用了它们。然而它们在两端很小的导数也为优化带来了许多麻烦，导致了 Vanishing Gradient[12]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。

梯度消失

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

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

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

高斯误差线性单元　　指数线性单元

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

相关阅读

文中为了简单起见，只是简单介绍了 ReLU 激活函数，关于更多激活函数的定义与性质可以看这篇笔记：
深度学习随笔——激活函数(*Sigmoid*、*Tanh*、*ReLU*、*Leaky ReLU*、*PReLU*、*RReLU*、*ELU*、*SELU*、*Maxout*、*Softmax*、*Swish*、*Softplus*) - *Lu1zero9* 的文章 - 知乎
https://zhuanlan.zhihu.com/p/585276457

形成图形化的直觉许多时候相当重要，这篇文章就给出了一个图形解释：
形象的解释神经网络激活函数的作用是什么？ - 忆臻的文章 - 知乎
https://zhuanlan.zhihu.com/p/25279356

[12]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。