

*Intelligence is ability to adapt to change*

*Stephen Hawking*

# 超智能体

生命不限于个体。并非所有生命拥有意识，但所有生命都拥有智能。这些智能体通过大量并行和多层迭代的方式形成新的智能体。细胞、器官、个体、国家、地球，不论从哪个层级上观察，都是一个“智能体”。

人类作为智能的一环，需跳出自身层级，用超出人类自身感知、情感和意识的方式去理解生命。

阅读

---

Q Tango

# 目錄

关于本书	1.1
如何阅读	1.2
智能起源	1.3
智能的本质	1.3.1
线性代数	1.3.2
复数	1.3.2.1
概率	1.3.3
熵与生命	1.3.4
智能的条件	1.3.5
自然智能	1.4
RNA与DNA	1.4.1
进化	1.4.2
生物学习	1.4.3
神经元工作原理	1.4.4
神经元本质行为	1.4.5
网络	1.4.6
智能不同阶段	1.4.7
智能结构梳理	1.4.8
人工智能	1.5
机器学习	1.5.1
人工神经网络	1.5.2
梯度下降训练法	1.5.2.1
Back Propagation	1.5.2.1.1
深层神经网络	1.5.3
基本用法	1.5.3.1
代码演示LV1	1.5.3.2
代码演示LV2	1.5.3.3
代码演示LV3	1.5.3.4
Batch Normalization	1.5.3.5
变体神经网络	1.5.4

---

神经网络的问题	1.5.4.1
循环神经网络——介绍	1.5.4.2
循环神经网络——实现LSTM	1.5.4.2.1
循环神经网络——代码LV1	1.5.4.2.2
循环神经网络——代码LV2	1.5.4.2.3
卷积神经网络——介绍	1.5.4.3
卷积神经网络——代码LV1	1.5.4.3.1
深层学习应用	1.5.5
自然语言处理	1.5.5.1
Word Embedding介绍	1.5.5.1.1

---

# 关于本书

该书最终的目的是：通过理解智能，学习如何学习。

1. 如何机器学习
2. 如何大脑学习

注：公式显示不了的话请刷新

未经允许禁止转载。

- 邮箱：[gxiuukk@gmail.com](mailto:gxiuukk@gmail.com)
- 知乎：<https://www.zhihu.com/people/YJango>
- 网页：<http://gxiuukk.wixsite.com/super>
- 微博：<http://weibo.com/YJango>
- Github：<https://github.com/YJango>

公开课视频：[深层神经网络设计理念](#)

# 如何阅读

智能并非人类所特有，而是自生命诞生时起就产生了。没有智能就没有生命。智能又并非单一状态，它和宇宙一样，都在不断的扩张。不同阶段的智能表现出的能力不同。神经元、蚁群、人类、社会、国家、地球，乃至整个宇宙都可被视为智能体。它们通过组合和迭代来形成更高级的智能功能。智能从未停止发展，而我们人类也只不过是其中一个“细胞”。这本书将从智能的角度展示对世界的不同理解。

核心知识：并非每个章节的内容本身（读者完全可以找到很多对应内容的经典书籍）。真正有价值的是将这些知识以何种方式排列和表达。

传达方式：语言是交流的工具，而交流的前提是双方的脑中都有相同内容。但是读者和作者的信息是不对等的。语言作为教学手段本身就有限。所以我描述的方式也并非直接告诉读者一个结论，打上这就是真理的标签，而是结合图，结合例子尽可能的消除信息的不等。我不是以教学者的身份来分享知识，而是引出一个问题让读者一起思考。虽然我提供了我的见解，但我相信会有很多读者能提出比我更好的见解。

## 行文风格

行文分很多部分，彼此的关系是递进或并列。每个部分通常会以一个问题的引入开始，结合若干实例对该问题进行思考，经过描述，最后得出总结性概括。文中带链接的内容请打开观看。

## 表达格式

问题：一、斜线文字

实例：

- 例子（情景、方式等）：普通格式

描述：普通格式

结论：

引用格式

## 例如

猫咪觉得自己属于人类吗？

- 实例1：实例内容
- 实例2：实例内容

描述内容

猫咪觉得人类属于自己

# 智能的起源

我们对智能的探索陷入了错误的方式中。

一、如果某天所有人类突然失忆，在不拆开计算机的情况下，我们该如何弄清计算机的工作原理，并给计算机下一个定义？

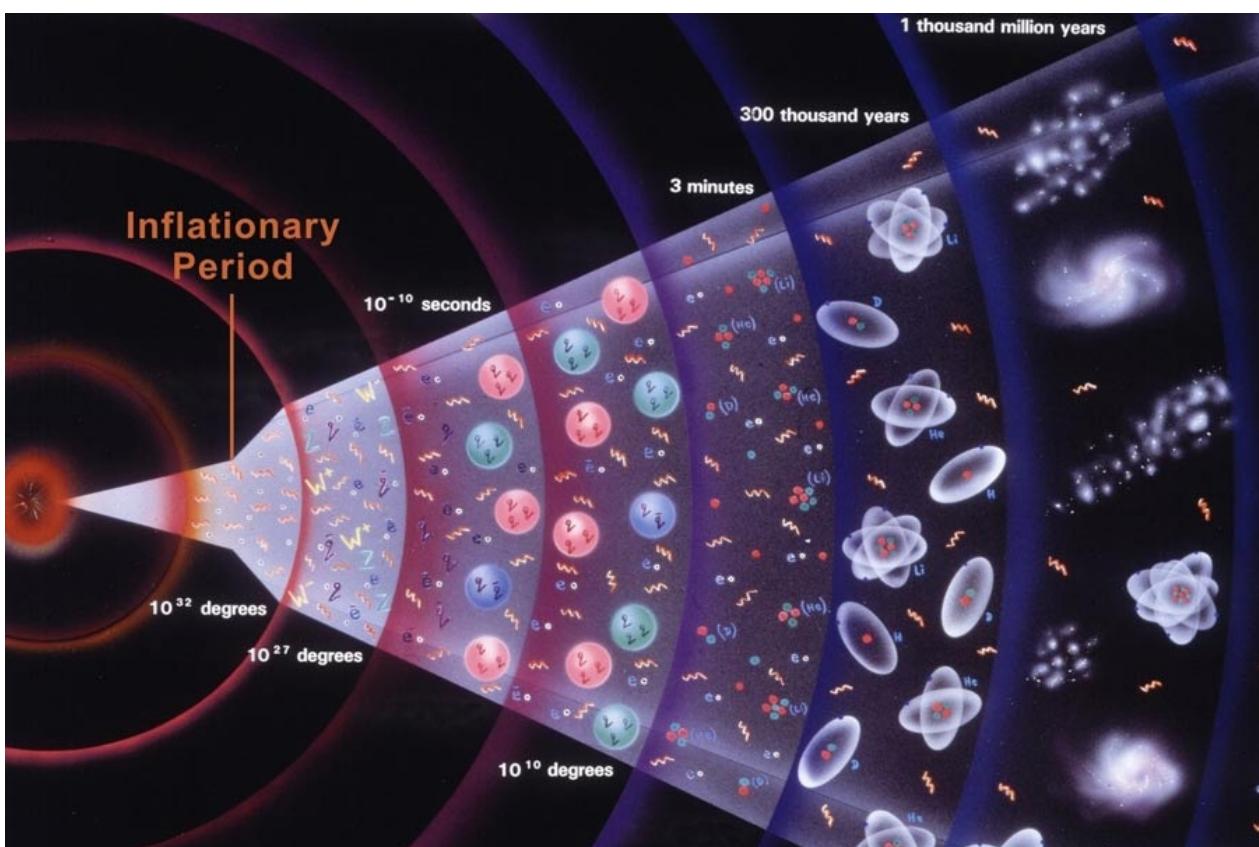
- 方式：从计算机软件开始研究，描述各种软件的功能，并找一个可以概括所有功能的定义。人们会发现计算机可以产生图像。但很快又发现很多软件并没有图形界面。有的软件可以玩射击游戏，有的却可以操控机械。各式各样的功能会被人们发现并逐一分类，使得用一个定义概括所有功能成为不可能。更不用说这些软件的功能还在持续增长。

显然我们并不以具体功能，而是以计算机最基本的工作原理去定义计算机。如今人们对智能的探索正是陷入了以功能来研究智能的方式中。人们把智能划分成了认知、理解、思考、学习等各式各样的功能并研究着，使得研究不同功能的专家对智能的定义都不相同。

如果我们能够知道宇宙自大爆炸之后是以何种方式进行扩张的。我们就可以从起点开始，推测出所有星系可能的形状甚至它们在未来可能的形状。

所以要想搞清智能的本质，不妨先试着找出智能的源头，思考一下为什么会产生智能。

智能的研究需要从智能的源头开始





# 智能的本质

## 一、生命是如何产生的？

在漫漫的宇宙演变中，我想很多星球都有产生生命的概率。

- 情景：想象某刻火星上产生了生命，但却遇到高温又被分解成了无机物。第二个产生的生命却因为找不到支持机体的能源而再次化成了无机物。第三个有机体幸运的诞生在大量能源旁，却由于能源的耗尽而前功尽弃。

问题的关键并非生命能否产生，而是产生的生命能否存活。

## 二、究竟是什么阻碍了生命，生命又该如何存活？

如果世界是静止的，那么生命自然不朽。然而我们的世界在时时刻刻发生着变化，会从一个状态变化到另一个状态。生物无从知道下一刻等待它的究竟是毁灭还是幸存？这种未来的不确定性（uncertainty）阻碍了生命的延续。

- 实例：植物会因为干旱而枯萎；野兽会因为捕不到猎物而饿死；每天又有约160中国人死于交通事故。

阻碍生命的正是这种不可预测性（随机），它还有另一个名字，叫做熵（entropy）。而热力学第二定律表述孤立系统会自发的朝向最大熵状态演化。

- 实例：我们房间会越来越乱，掉在地上的杯子的碎片会随处散落，熵会自发性的不断增加。我们从未见过房间自己越来越整洁，将杯子碎片会摔出一个完整的杯子的情况。说明了熵增在孤立系统下并不可逆。



熵增不可逆的现象也意味着世界持续的发生变化。在过去可能产生过无数个生命，但只有那些可以根据变化而做出相应变化的生命才能躲避危险从而幸存。这种能力就是智能，同时也是生命得以延续的原因。

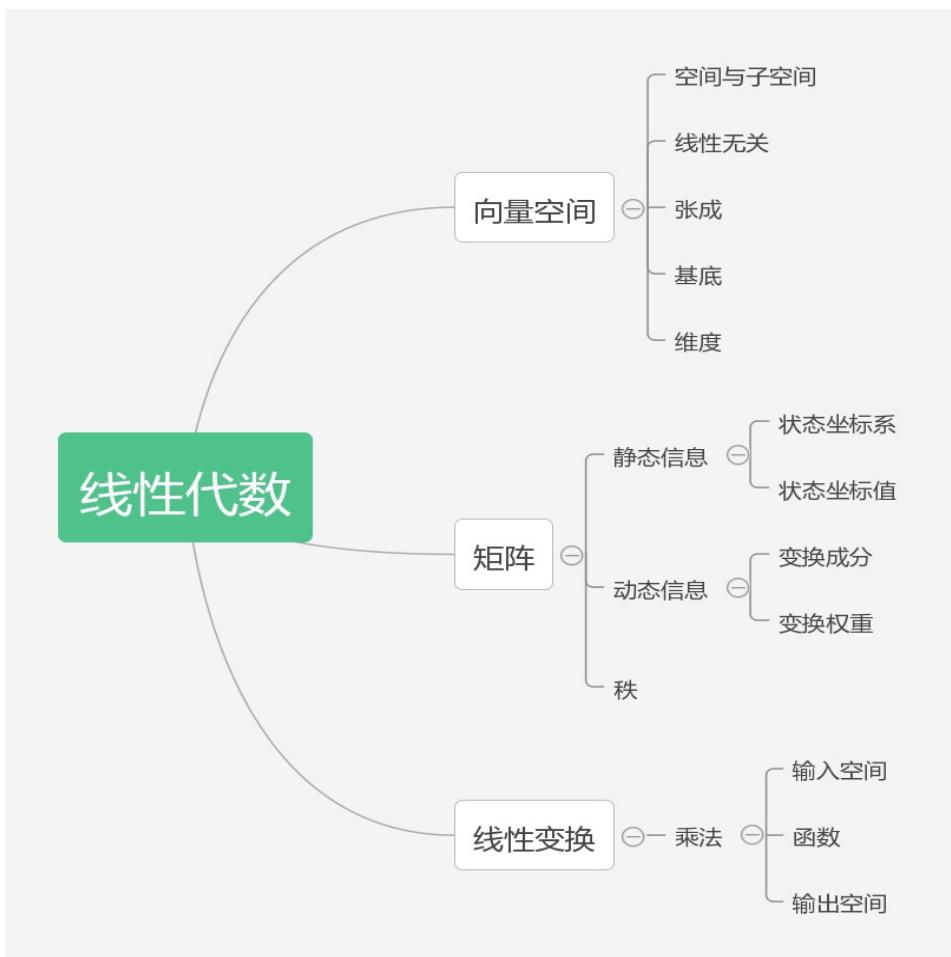
| 智能：可以根据环境变化而做出相应变化的能力

由于阻止生命延续的实际是不确定性（随机性、不可预测性），那生命所做的就是减少该不确定性来延续。

奥地利物理学家薛定谔在《[生命是什么](#)》首次提出负熵的概念，并认为生命以负熵为生，所以智能也可被描述是：

| 智能：熵减的能力

# 线性代数



## 一、什么是线性代数？

不断变化的世界使我们产生时间观念。正确描述事物状态及其不同时间下的变化至关重要。我们知道在三维空间下如何描述物体的位置。然而除了长宽高，世界上还有很多决定事物状态的因素。如决定股票价钱的因素、决定天气的因素。这些因素又该如何合理的描述？线性代数给了我们答案。推荐阅读《[Linear Algebra and Its Applications](#)》。

线性代数是有关任意维度空间下事物状态和状态变化的规则。

## 矩阵乘法

### 二、矩阵是什么？矩阵乘法又是什么？

带着这个问题我们开始对矩阵及其乘法进行第一遍理解。

# 向量点乘

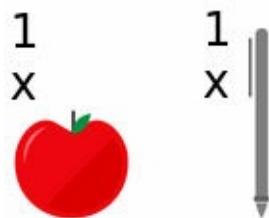
全篇将会以一个实例进行讨论，请观看一遍视频[PPAP洗脑全球](#)。

三、视频的内容涉及到很多种状态及变换。用线性代数应该如何描述？

视频内容可以写成3个向量乘法：

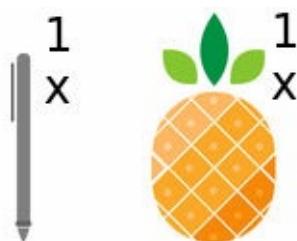
1、I have a pen, I have an apple---->apple pen

$$[applepen] = \begin{bmatrix} 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} pen \\ apple \end{bmatrix} \quad (\text{eq.1})$$



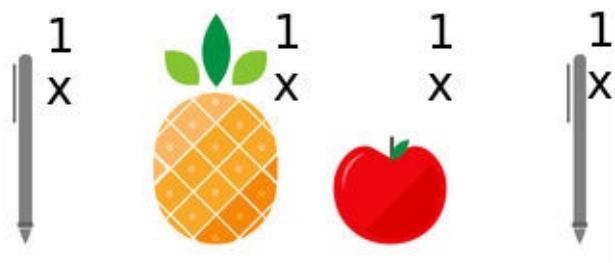
2、I have a pen, I have a pineapple---->pineapple pen

$$[pineapplepen] = \begin{bmatrix} 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} pen \\ pineapple \end{bmatrix} \quad (\text{eq.2})$$



3、apple pen, pineapple pen---->pen pineapple apple pen

$$[penpineappleapplepen] = \begin{bmatrix} 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} applepen \\ pineapplepen \end{bmatrix} \quad (\text{eq.3})$$



每个等式右边的第二个向量表示变化前拥有什么，右边的第一个向量表示变化时各拿几个，而等式的左边表示变化后获得了什么。从中可以看出来：

向量点乘(dot product)是一种组合(combination)

## 矩阵乘向量

四、有没有其他描述方式？

可以把(eq.1) (eq.2) 合二为一，表示为 (eq.4)：

(eq.1) I have a pen, I have an apple---->apple pen，

(eq.2) I have a pen, I have a pineapple---->pineapple pen

$$\begin{bmatrix} \text{applepen} \\ \text{pineapplepen} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} \text{apple} \\ \text{pineapple} \\ \text{pen} \end{bmatrix} \quad (\text{eq.4})$$

此时，表示各拿几个的向量变成了两行（两组）向量，也就成了矩阵（向量是只有一行或一列的矩阵）。每个向量也叫一组权重(weights)。

在  $\begin{bmatrix} 1 & 0 & 1 \end{bmatrix}$  中，第一个1对应着apple，第二个0对应着pineapple，第三个1对应着pen。不可以随意调换位置，所以，

向量是有顺序的一组数字，每个数字是向量的一个因素(element)

因素横着排列的向量叫做行向量(row vector)

因素竖着排列的向量叫做列向量(column vector)

更具体的描述一下第一个结论。向量点乘是一种组合，但

向量点乘是一个向量中各个因素的一个组合

五、如何计算矩阵乘向量？

(eq.4) 可分两步：

$$\begin{bmatrix} 1 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \text{apple} \\ \text{pineapple} \\ \text{pen} \end{bmatrix}$$

1. 计算第一行权重形成的组合：

$$\begin{bmatrix} \text{applepen} \end{bmatrix}$$

后，放到第一行。

2. 计算第二行权重形成的组合：

$$\begin{bmatrix} 0 & 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} \text{apple} \\ \text{pineapple} \\ \text{pen} \end{bmatrix}$$

得到的组合apple pen

得到的组合pineapple pen

后，放到第二行  $\begin{bmatrix} \text{pineapplepen} \end{bmatrix}$ 。

3. 行成的  $\begin{bmatrix} \text{applepen} \\ \text{pineapplepen} \end{bmatrix}$  依然有顺序，仍然是一个向量。比较向量点乘，可以看出

矩阵乘向量是向量中各个因素有顺的多个组合

## 向量乘矩阵

六、形成组合的成分一定是元素（数）吗？

形成组合的成分并非一定是向量中的各个元素，也可以是不同向量之间形成组合。

可以把 (eq.1) (eq.2) (eq.3) 所完成的行为改写成 (eq.5) (eq.6)：

$$[\text{applepen} \quad \text{pineapplepen}] = [1 \quad 1] \cdot \begin{bmatrix} \text{pen} & \text{pen} \\ \text{apple} & \text{pineapple} \end{bmatrix} \quad (\text{eq.5})$$

$$[\text{penpineappleapplepen}] = [\text{pineapplepen} \quad \text{applepen}] \cdot \begin{bmatrix} 1 \\ 1 \end{bmatrix} \quad (\text{eq.6})$$

(eq.5) 等式右侧的矩阵由两个行向量组成。

- 矩阵中的第一个行向量表示两次组合中分别先拿什么，第二个行向量表示两次组合中分别后拿什么。
- 权重  $[1 \quad 1]$  的第一个因素对应着矩阵中第一个行向量的个数，第二个因素表示右侧第二个行向量的个数。
- 矩阵中每个行向量内部因素的比例不变，整体完成矩阵内向量与向量之间的组合。

向量乘矩阵可以是矩阵中各个行向量有顺的多个组合

你会发现 (eq.6) 不同于 (eq.5)，要形成组合的向量被拿到了乘法点(dot)的左边，而权重被拿到了右边。

- 效果是拿一个  $\text{penpineapple}$  和一个  $\text{applepen}$  形成组合。
- 因为当行向量的因素作为组合成分时，乘法点右侧的矩阵（向量）是权重的信息。

可以看出矩阵乘法并不满足乘法交换律，因为交换了两个矩阵的位置，就交换了权重与要形成组合的向量的位置。

矩阵乘法不满足乘法交换律：commutative law:  $\mathbf{AB} \neq \mathbf{BA}$

## 矩阵乘矩阵

七、可以进行批量组合吗？

矩阵乘矩阵就可以看作是对一个矩阵中各个向量的批量线性组合。

如果视频中跳了两遍舞蹈。第二遍跳舞时，他在两次组合中，首次拿的东西都是两个，那么就可以把 (eq.5) 等式右侧的行向量变成两个行向量，形成了一个矩阵。

$$\begin{bmatrix} \text{apple} & \text{pen} \\ \text{apple} + 2 * \text{pen} & \text{pineapple} + 2 * \text{pen} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 2 & 1 \end{bmatrix} \cdot \begin{bmatrix} \text{pen} & \text{pen} \\ \text{apple} & \text{pineapple} \end{bmatrix}$$

在唱第二遍时，就要唱：

- I have two pens. I have an apple. Apple-2pens!
- I have two pens. I have a pineapple. Pineapple-2pens!

之前仅仅是把单词放在一起，并没有说明他们是如何组合的。而上式中终于写出了：  
 $\text{pineapple} + 2 * \text{pen}$ 。

也就是只有乘法来控制数量，加法来组合不同向量。这样的组合方式才是线性代数讨论的组合，即线性组合。

所以所有已概括的结论中，组合前面都要加上“线性”两个字。同时控制数量的数是属于什么数要事先规定好（经常被规定为是实数  $\in R$ ，也有虚数域）。

不过这还没有结束，严谨性是数学的特点。上文所说的“加法”和“乘法”也只不过是个名字而已。它们到底指的是什么运算，遵循什么样的规则需要明确规定。

当你看线性代数教材的时候，你就会发现这8条规则。

1.  $x + y = y + x$
2.  $x + (y + z) = (x + y) + z$
3. 有一个唯一的“零向量”对任意  $x$  都能使  $x + 0 = x$
4. 每个  $x$  都有一个唯一的相反数使得  $x + (-x) = 0$
5.  $1x = x$
6.  $(c_1 c_2)x = c_1(c_2 x)$
7.  $c(x + y) = cx + cy$
8.  $(c_1 + c_2)x = c_1x + c_2x$

不需要去记它们。只需要知道，它们是用于描述和约束在线性代数中的加法，乘法的运算。

特别要注意的是，这些运算都有一个原点 (0)，为了允许正负的出现。

线性组合：向量乘上各自对应的标量后再相加所形成的组合。（满足上述对乘法、加法的规则）

## 矩阵是什么

八、熟悉了各个乘法后，矩阵到底是什么？

线性代数是用来描述状态和变化的，而矩阵是存储状态和变化的信息的媒介。

矩阵的信息可以分为状态（静态）和变化（动态）信息来看待。

### 矩阵的静态信息

当把矩阵以静态信息来看待时，其信息的侧重点在于状态二字。

向量可用于描述一个事物的状态，该事物的状态由向量内各个因素来描述。

而矩阵可以视为多个维度（因素的个数）相同的向量的有序排列。

同时矩阵也可以视为一个“向量”，用于描述一个事物的状态，内部的每个向量就是矩阵的“因素”，该事物的状态由矩阵内各个向量来描述。

多个标量有序排列后形成向量，多个向量有序排列后形成矩阵，多个矩阵有序排列后形成三维张量（3D tensor）。所以标量可以视为因素个数为1的向量，向量可以视为因素个数为1的矩阵，矩阵可以视为因素个数为1的三维张量（3D tensor）。

### 坐标值与坐标系：

描述一个事物的状态需要在一个选好的坐标系中进行，所以矩阵所包含的信息从来都是成对出现。

向量  $\begin{bmatrix} \text{apple} \\ \text{pen} \end{bmatrix}$  举例来说，这个向量并没有被赋予任何数值。但已经确定了我们要在apple的数量和pen的数量的两个因素（两个维度）下描述数据。换句话说，坐标系已被规定好。所以

当写出任何具有实际数值的向量，如  $\begin{bmatrix} 2 \\ 1 \end{bmatrix}$  时，坐标系（二维向量空间）和坐标值就同时被确

定了。它实际上是  $\begin{bmatrix} \text{apple} \\ \text{pen} \end{bmatrix}$  和  $\begin{bmatrix} 2 \\ 1 \end{bmatrix}$  的缩写。二者无法分割。即使是  $\begin{bmatrix} \text{apple} \\ \text{pen} \end{bmatrix}$ ，虽然pen，apple前没有任何具体数字。但依然包含所有因素间的比例相同的隐含信息。调换2和1的顺序同时也表示坐标轴之间的调换。

## 矩阵的动态信息

当把矩阵以动态信息来看待时，其信息的侧重点在于变化二字。这时的矩阵可以看做是一个方程。

变化可以理解为由于矩阵的作用，事物本身的变化，也可以理解为坐标系的变化。

向量可用于控制变化时所用成分的数量，即一组权重。

矩阵可以视为多个维度（因素的个数）相同的权重的有序排列。可对另一个矩阵的静态信息进行批量变化。

## 矩阵乘法是什么

矩阵可以被视为载有状态和变化两种信息的媒介。而矩阵乘法就是变化的行为。

在一个矩阵内，把矩阵内的向量理解为向量或权重都可以。

但是当两个矩阵进行矩阵乘法时，一旦选择以动态信息理解其中一个矩阵，另一个矩阵的信息就会被瞬间静态信息。

两个矩阵相乘，一个矩阵提供状态信息，另个矩阵提供变化信息。

两个矩阵相乘 $A \cdot B$ 时，

当把前者矩阵(A)中行向量理解成若干组权重，后者矩阵(B)中的行向量就是要形成组合的成分。

$$\begin{aligned}
 C_a: & \boxed{c_i \ c_{ii} \ c_{iii}} & W_a: & \boxed{\color{red}{w_1} \ \color{blue}{w_2} \ \color{green}{w_3}} & V_1: & \boxed{v_i \ v_{ii} \ v_{iii}} \\
 C_b: & \boxed{c_i \ c_{ii} \ c_{iii}} & = & W_b: \boxed{\color{red}{w_1} \ \color{blue}{w_2} \ \color{green}{w_3}} & \cdot & V_2: \boxed{v_i \ v_{ii} \ v_{iii}} \\
 & & & & & V_3: \boxed{v_i \ v_{ii} \ v_{iii}}
 \end{aligned}$$

2组行向量权重 :  $W_a$ 、 $W_b$

3个行向量成分 :  $V_1$ 、 $V_2$ 、 $V_3$

得到2个行向量组合 :  $C_a$ 、 $C_b$



每个组合 : 3个向量乘以各自权重并相加后, 放到左边对应权重组的位置

$$C_b: \boxed{c_i \ c_{ii} \ c_{iii}} = W_b: \boxed{\color{red}{w_1} \ \color{blue}{w_2} \ \color{green}{w_3}} \rightarrow \times V_1: \boxed{v_i \ v_{ii} \ v_{iii}} + \times V_2: \boxed{v_i \ v_{ii} \ v_{iii}} + \times V_3: \boxed{v_i \ v_{ii} \ v_{iii}}$$

当把后者矩阵(B)中列向量理解成若干组权重, 前者矩阵(A)中的列向量就是要形成组合的成分。

$$\begin{array}{ccc}
 C_a & C_b & C_c \\
 \boxed{c_i} & \boxed{c_i} & \boxed{c_i} \\
 \boxed{c_{ii}} & \boxed{c_{ii}} & \boxed{c_{ii}}
 \end{array} = \begin{array}{ccc}
 V_1 & V_2 & V_3 \\
 \boxed{v_i} & \boxed{v_i} & \boxed{v_i} \\
 \boxed{v_{ii}} & \boxed{v_{ii}} & \boxed{v_{ii}}
 \end{array} \cdot \begin{array}{ccc}
 W_a & W_b & W_c \\
 \boxed{w_1} & \boxed{w_1} & \boxed{w_1} \\
 \boxed{w_2} & \boxed{w_2} & \boxed{w_2} \\
 \boxed{w_3} & \boxed{w_3} & \boxed{w_3}
 \end{array}$$

3组列向量权重 :  $W_a$ 、 $W_b$ 、 $W_c$

3个列向量成分 :  $V_1$ 、 $V_2$ 、 $V_3$

得到3个列向量组合 :  $C_a$ 、 $C_b$ 、 $C_c$



每个组合 : 3个向量乘以各自权重并相加后, 放到左边对应权重组的位置

$$C_a = \boxed{c_i} + \boxed{c_{ii}} = \boxed{v_i} + \boxed{v_{ii}} + \boxed{v_{iii}} \quad \times \quad \begin{array}{c} V_1 \\ V_2 \\ V_3 \end{array} \quad \times \quad \begin{array}{c} W_a \\ \boxed{w_1} \\ \boxed{w_2} \\ \boxed{w_3} \end{array}$$

注意对应行向量与列向量。

转置一个矩阵可以理解为调换一个矩阵的动态与静态信息。

单位矩阵可以被理解为动态与静态信息相同。

回想线性组合的描述（向量乘上各自对应的标量后再相加所形成的组合），因为向量的维度和权重的维度要一一对应。所以，

矩阵  $\mathbf{A}$ ( $m$  by  $n$ ) 和矩阵  $\mathbf{B}$ ( $p$  by  $q$ ) 能够做乘法的条件是  $n = p$

## 向量空间

很多线性代数教材所引入的第一个概念就是线性空间（linear space）。可见它的地位。虽然它有些抽象，但是却是自然而然推演出来的一个概念。

### 九、空间是什么？

空间的本质是集合。而且是一个能够容纳所有要描述状态的集合。若超过空间范围，就该寻找正确的空间。

对“要描述内容”进行进一步说明，需从如何理解线性代数这四个字开始。

我们已经知道了什么是线性（那8个条件约束的加法和乘法）。

那什么是代数？意思是指你可以把任何概念都代入其中。

看视频的时候，人们自然而然的会把苹果菠萝换成其他事物，比如[PPAP河南话版](#)。也可以换成任何宇宙上有的物体。

不仅仅是物体，甚至可以是一个抽象的概念。

我个人最喜欢的描述是：向量空间是描述状态(**state**)的线性空间。再加上之前的约束，于是我们就有了

向量空间是能够容纳所有线性组合的状态空间

### 十、什么样的状态空间能够容纳所有的线性组合？

- 情景：假如要描述一个人的两个状态（下图中的行向位置和纵向位置），向量的维度就是二维。那么一个大圆盘够不够容纳所有的线性组合？答案是不够。



因为线性组合是向量乘以各自对应的标量后再相加所形成的组合，而这个标量是实数域的时候，由于实数域无线延伸，那么乘以标量后的状态也会无限延伸。所以向量空间一定是各个维度都像实数轴一样可以无线延伸。最终得到的将不会是一维下的线段，二维下的圆盘。而一定是一维下的无限延伸的直线，二维下的无限延伸的平面。

向量空间的基本特点是各个维度都可以无限延伸，且过原点。

之所以用状态二字，是因为刚才的两个维度，可以用于描述速度和体温。这时两个维度所展开的依然是一个平面，但却不是描述位置的平面。

## 子空间

子空间（subspace）可以被想成是向量空间内的向量空间，同样要满足能够容纳线性组合的条件。

十一、最小的子空间是什么？

只有一个状态的空间（集合）。这个状态不是其他状态，就是 $0$ 。只有这样才可以在乘以标量后依然不会跑出空间外。

十二、其次空集可不可以是向量空间？

不可以，空集是没有任何元素的集合，既然什么状态都没有，又怎么能够容纳线性组合。

最小的向量空间是只包含零向量的空间

十三、假如上图的圆盘是个无线延伸的平面，那平面的子空间可以是平面上所有直线吗？

不可以，8个运算规则中明确规定了，一定要有原点，这样才可以包含正负。所以这个平面的子空间是所有过原点的直线，加上中心的那个原点自己所组成的最小子空间，再加上这个平面自身（最大的子空间）。

## 线性无关

十四、该如何选择因素？

在视频的例子中，当要把 (eq.1) (eq.2) 合为 (eq.4) 时，是这个样子：

$$\begin{bmatrix} \text{applepen} \\ \text{pineapplepen} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} \text{apple} \\ \text{pineapple} \\ \text{pen} \end{bmatrix} \quad (\text{eq.4}) , \text{但最右侧的向量并不是} \\ \begin{bmatrix} \text{apple} \\ \text{pen} \\ \text{pineapple} \\ \text{pen} \end{bmatrix} \text{4个维度。而是三个。因为 pen 和 pen 是一个东西。}$$

我们想用的是若干个毫不相关的因素去描述状态。在线性空间下的毫不相关，叫做线性无关。

十五、要描述的状态是由向量来描述时怎么办？

判断两个向量是否线性无关时，可以看他是否在空间下平行。

但怎么判断几个向量之间（不一定是两个）是否线性无关？我们需要可靠的依据。

线性无关 (**linearly independent**)：当  $c_i$  表示权重， $v_i$  表示向量时，

$c_1 v_1 + \dots + c_k v_k = 0$  只发生在  $c_1 = \dots = c_k = 0$  全都等于零时。

换句话说，这些向量不可以通过线性组合形成彼此。形成彼此的情况只能是他们都是零向量。

## 张成

注意词的属性和关联词。

张成 (**spanning**) 是一个动词，动词的主语是一组向量 (**a set of vectors**)。

描述的是一组向量通过线性组合所能形成子空间。描述的内容并不是形成的这个空间，而是形成的这个行为。

$\begin{bmatrix} \text{apple} \\ \text{pen} \\ \text{pineapple} \\ \text{pen} \end{bmatrix}$ ，是4个向量，但只可以张成一个三维空间。（因为有二维线性相关，所以并不能张成4维）

## 基底

一个向量空间的一个基底（A basis for a vector space  $V$ ）是一串有顺序的向量（a sequence of vectors），满足：

- A、向量之间彼此线性无关（不可多余）
- B、这些向量可以张成向量空间  $V$ （不可过少）

刚刚好可以张成向量空间  $V$  的一串向量是该向量空间  $V$  的一个基底。

基底是一个类似 people 的复数名词，是从属于某个空间的，而不是矩阵，也不是向量。

## 维度

一个向量空间可以有无数个基底。但每个基底所包含的向量的个数（the number of vectors in every basis）是一个空间的维度。

注意：维度是空间的概念，而不是描述一个具体的向量。人们常说的  $n$  维向量实际是指  $n$  维向量空间内的向量，由于在讨论时并未给向量指定任何实际的数值，所以可以是任何值，可以张成整个空间。所以其真正描述的依旧是一个空间。并且，维度是站在观察者角度，希望在某个向量空间下尽可能的描述物体状态而选择的，并不一定是被描述者真实处的空间。

但若是你觉得理解起来有困难。就简单记住：

互不相关的因素的个数是一个向量空间的维度

## 秩

矩阵可以视为动态和静态信息的媒介。而一个具体的矩阵到底涵盖了多少信息可以由秩（rank）来描述。

指的是一个矩阵的所有列向量所能张成的空间的维度。

矩阵的所有列向量所张成的空间叫做列空间（column space）

矩阵的所有行向量所张成的空间叫做行空间（row space）

一个矩阵的列空间的维度是这个矩阵的秩，同时也等于该矩阵行空间的维度

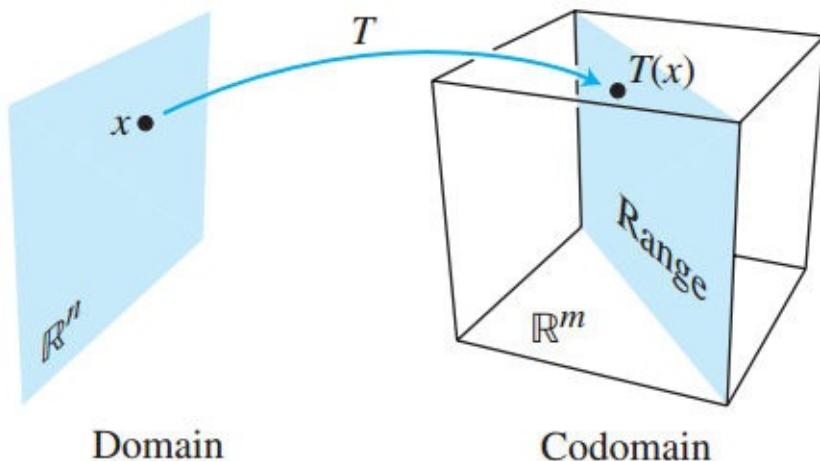
秩是用于描述矩阵所包含信息量的。

## 线性变换

最终我们想要做的就是描述事物的变化。上文所有的内容都可以说是为此刻所做的铺垫。

矩阵乘以矩阵可以视作一个矩阵内部向量的批量线性变换（linear transformation）。所以可以仅讨论由矩阵乘以向量所形成的一次线性变换。

## 十六、什么是变换？



**FIGURE 2** Domain, codomain, and range  
of  $T : \mathbb{R}^n \rightarrow \mathbb{R}^m$ .

by David C.Lay

一个从 $n$ 维实数域 ( $\mathbb{R}^n$ ) 到 $m$ 维实数域 ( $\mathbb{R}^m$ ) 的变换 (transformation or mapping or function)  $T$  是将 $n$ 维实数域 ( $\mathbb{R}^n$ ) 空间下任意一个向量  $x$  转换成为在 $m$ 维实数域 ( $\mathbb{R}^m$ ) 空间下对应向量  $T(x)$

其中 $n$ 维实数域 ( $\mathbb{R}^n$ ) 空间叫做变换  $T$  的 domain， $m$ 维实数域 ( $\mathbb{R}^m$ ) 空间叫做该变换的 codomain。

向量  $T(x)$  叫做向量  $x$  的 image (变换  $T$  行为下的)

所有 image 组成的集合叫做变换  $T$  的 range

而线性变换是指线性规则所造成的变换， $T()$  是由一个矩阵  $A$  来实现的。此时你就会看到无处不在的式子：

$y = Ax$  : 列向量  $x$  左乘一个矩阵  $A$  后得到列向量  $y$

$$\begin{bmatrix} \text{apple} \\ \text{pen} \\ \text{apple} \\ \text{pen} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} \text{apple} \\ \text{pineapple} \\ \text{pen} \end{bmatrix}$$

(eq.4) 举例来说， $x$  是三维空间的向量（即  $A$  的 domain 是三维），而经过线性变换后，变成了二维空间的向量  $y$ （即  $A$  的 codomain 是二维）。

矩阵  $A$  可以被理解成一个函数(function)，将三维空间下的每个向量投到二维空间下。

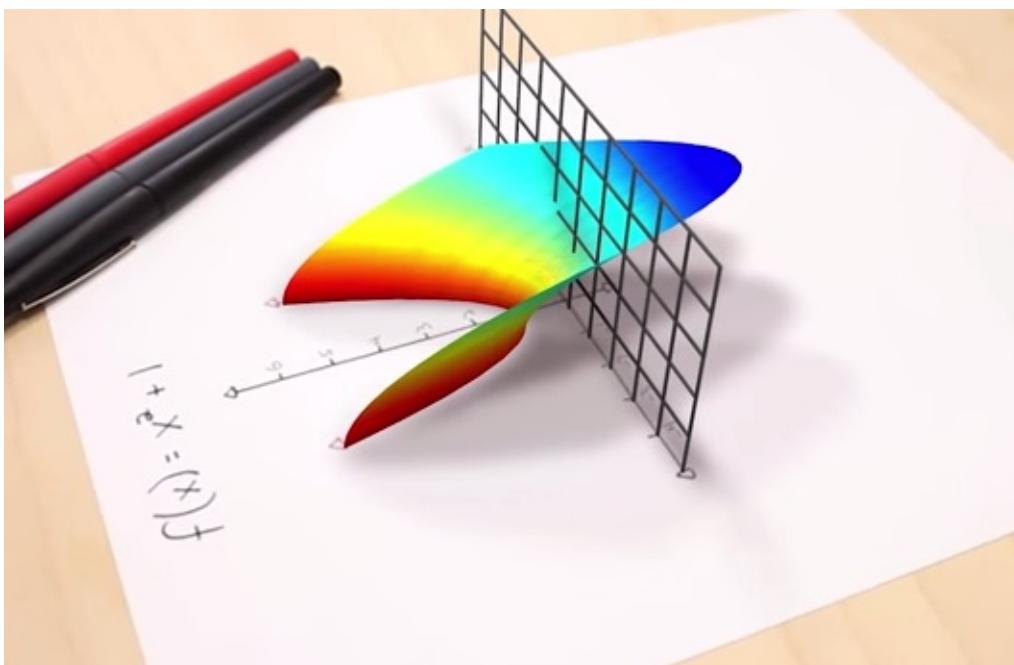
$y = Ax$  也可以理解为  $x$  经由一个外力  $A$ ，使其状态发生了改变。

$Ax$  同时也是深层神经网络每层变换中的核心： $y = a(Ax + b)$

# 复数

数字也是人们用来表示状态的符号。但是像方程 $0 = x^2 + 1$ 的解无法用实数来表达的时候，就意味着我们的数字不够描述这种状态。于是我们从实数所在的1维空间扩展到了复数所在的2维空间。用两个因素来表达 $0 = x^2 + 1$ 中 $x$ 的状态。

当在二维空间下描述 $0 = x^2 + 1$ 中的 $x$ 时， $x$ 的所有值分布在如下图的区域中



via Welch Labs

其中红色部分仅仅是该二维空间在虚部等于0时的一个切面，并非是 $x$ 的全部解所组成的集合。

## 维度的扩展

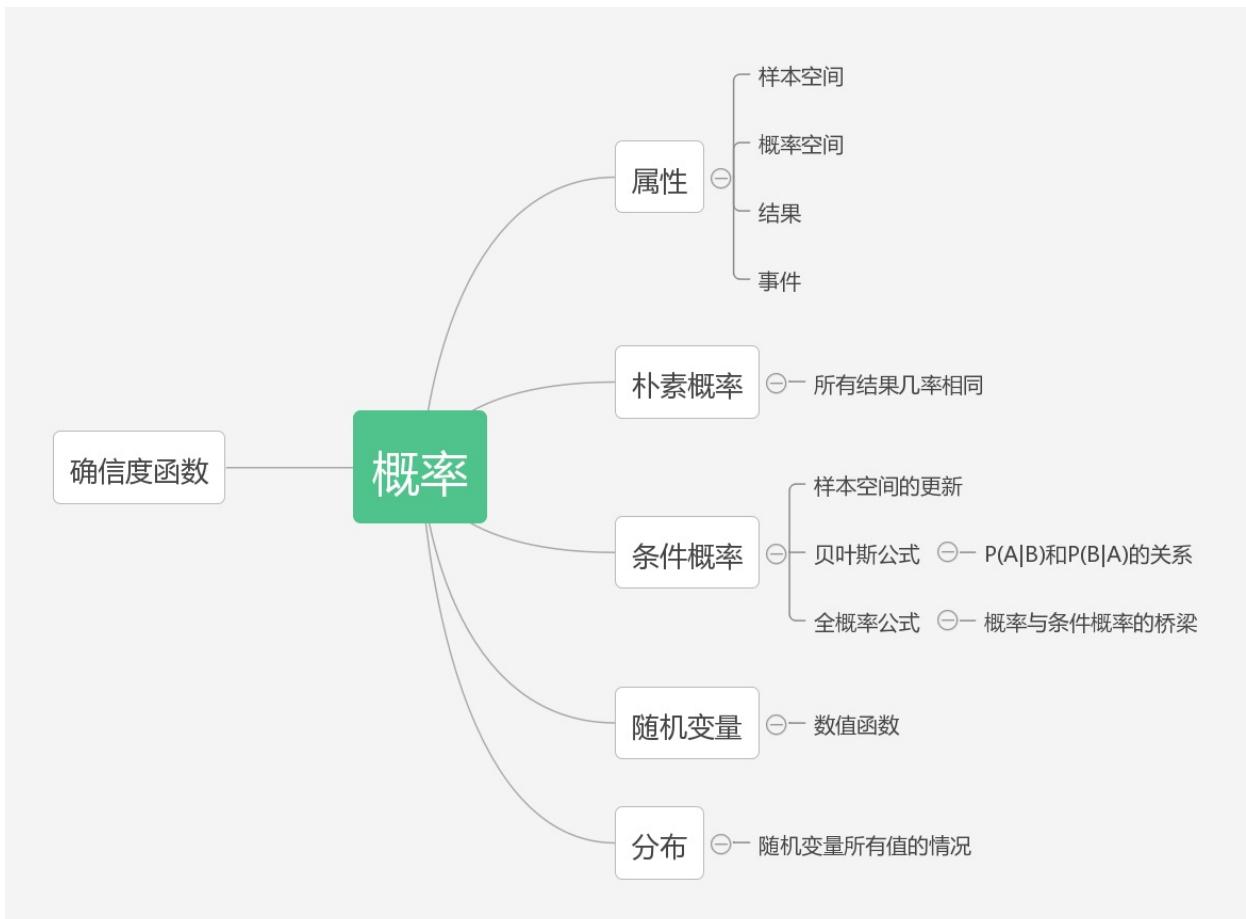
以复数作为引入，想分享在各个领域都非常重要的概念：思维空间。

人的成长往往是以认识到自己的渺小开始的。人们认为自己拥有自由的意识和思维。然而这种自由也是有限的。它好比线性空间里的张成，能张成多大的意识空间取决于脑中有多少互不相关的因素，也就是维度。很多时候找不到答案并非不够聪明，而是没有找对地方。以下就是维度对于理解事物的作用：

- 复数的理解：进一步扩展的数的域。
- 傅里叶变换的理解：在x-y坐标系上增加1维时，一切豁然开朗。
- 弦理论的理解：尝试融合相对论和量子力学的理论，但只有当扩充到10维空间+1维时间时，数学公式才合理。

当问题无法被理解时，往往是因为找错的地方，不妨尝试扩展维度，增加搜索空间。

# 概率



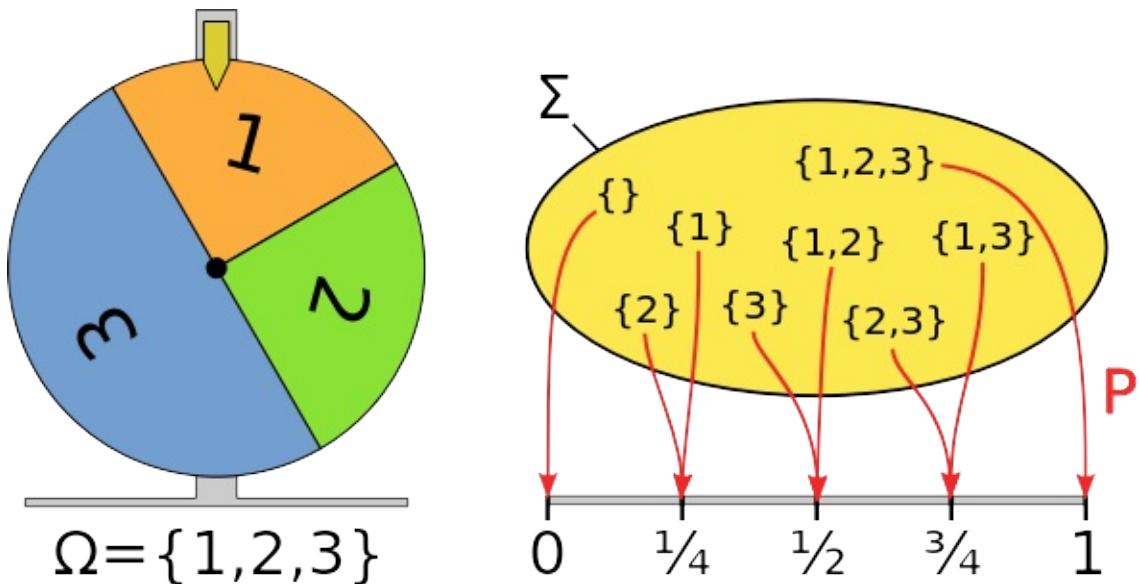
通过线性代数，我们知道了该如何描述事物状态及其变化。遗憾的是，对一个微小的生物而言，世界并非确定性（deterministic）的，由于信息量的限制，很多事物是无法确定其变化后会到达哪种状态。然而为了更好的生存，预测未来状态以决定下一刻的行为至关重要。而概率给我们的决策提供了依据。推荐阅读《[Introduction to Probability](#)》

## 一、什么是概率？

概率是用来衡量我们对事物在跨时间后不同状态的确信度。

- 设想：如果时间静止，其实并不需要概率。因为不确定的信息永远不知道，新的不确定性也永远不出产生。所以一定要以时间轴会变化的角度去理解概率。概率是观察者畅想一个状态从某个时间点变化到另个时间点后的確信度。跨时间可以是过去到未来，也可以是未来到过去。
- 情景：如何考虑转盘在未来停止后指针指向各个数字的可能性？为方便研究，需要总结出在任何情况都普遍适用的属性，并给予它们固定的名字。

1,2,3是可能被指到的三个结果（outcome）。在这里，这三个结果组成的集合也同时是样本空间（sample space），即无论事态如何发展，结果都不会出现在该集合之外（和向量空间一样）。样本空间的子集，如 $\{1,2\}$ 叫做一个事件（event），表示指针指到1或2的情况。满足任何一个情况都算作该事件发生了（occurred）。所有事件发生的可能性都用值域为 $[0,1]$ 间的实数表示，1表示必然发生，0表示不可能发生。 $\{1\}, \{2,3\}$ 两个不相交的事件的概率和为1。 $[0,1]$ 间的实数是概率得出的值，但并非概率的全部。概率是一个函数。



概率：概率是将样本空间内的子集投向概率空间的函数。

概率  $P()$  将事件  $A \subset S$  作为输入，并输出  $[0,1]$  之间的实数表示其发生的可能性。该函数需要满足两个条件：

$$1. P(\emptyset) = 0, P(S) = 1,$$

空集的概率为0，全集的概率为1。

$$2. P\left(\bigcup_{j=1}^{\infty} A_j\right) = \sum_{j=1}^{\infty} P(A_j),$$

不相交事件之间的并集事件的概率等于各个事件概率之和。

结果：可能到达的状态

样本空间：所有可能产生的结果所组成的集合。

事件：样本空间的子集。

当实际发生的结果  $s_{outcome} \in A$  时，表示  $A$  事件发生。

二、朴素概率的计算以及和普遍概率的区别是什么？

人们在计算概率时常常犯的错误就是不假思索的假定所有结果所发生的可能性都相同。并用事件的结果个数比上样本空间的结果个数。

$$\text{朴素概率} : P_{\text{naive}}(A) = |A|/|S| ,$$

$|A|$  和  $|S|$  表示集合中元素的个数。

但是这种假设并不严谨。

- 实例：在上图原盘问题中，如果使用朴素概率来计算指针停止时指向2的概率，就会得到  $P_{\text{naive}} = |A|/|S| = 1/3$  的概率。但很明显，指向3的结果就占有原盘一半的空间，指向3的概率更大。使得各个结果发生的可能性并不相同。不可以使用朴素概率算法。从图中可以看出答案是  $1/4$ 。

样本空间好比是总价为**1**的一筐苹果，一个事件就是一堆苹果，概率是将这堆苹果转换成实际价钱的函数。但苹果有大有小，只有当所有苹果都一模一样时，这堆苹果的价钱才是 苹果数/总个数。空集，即一个苹果都没有的话，价格为**0**。整框苹果的话，价格自然为**1**。把整框苹果分成几堆（事件之间不相交），价格的总和为**1**。



## 条件概率

当我们获得更多信息后，新信息会对原始样本空间产生更新。（看完熵的概念后再次理解概率和新信息会有意想不到的收获）

### 三、条件概率又是什么？

条件概率是新信息对样本空间进行调整后的概率情况。

- 实例：从一副洗好的扑克里，不放回的依次抽两张卡片。事件 $A$ 表示第一张卡片是心，事件 $B$ 表示第二张卡片是红色。求事件 $B$ 发生的条件下，事件 $A$ 发生的概率 $P(A|B)$ 。以及事件 $A$ 发生的条件下，事件 $B$ 发生的概率 $P(B|A)$ 。

卡片都是均匀形状，可用朴素概率计算。最初的样本空间是 $54 * 53 = 2862$ 种。事件 $B$ 发生后，样本空间被调整，所有第二张不是红色的结果都会从样本空间内去掉，变成 $26 * 53 = 1378$ 种（可认为第二张先抓，顺序不影响组合结果）。其中第一张是心，且第二张是红色的结果有 $13 * 25 = 325$ 种。所以 $P(A|B)$ 的概率为 $325/1378 \approx 0.236$ 。

事件 $A$ 发生后，所有第一张不是心的结果都会从样本空间内去掉，变成 $13 * 53 = 689$ 种。其中第一张是心，且第二张是红色的结果有 $13 * 25 = 325$ 种。所以 $P(B|A)$ 的概率为 $325/689 \approx 0.472$ 。

$P(A|B)$ 和 $P(B|A)$ 二者的条件对原始样本空间的调整不同，所以并不相等。同时“|”右边的事件并不意味着先发生，也并不意味着是左边事件的起因。

- 实例：先后投两次硬币。原始样本空间是{正正，反反，正反，反正}。已知事件 $A$ 是第一次投得正面，事件 $B$ 是第二次投得正面。 $P(B|A)$ 更新后的样本空间为{正正，正反}。但第二次投得正面的概率仍然是 $1/2$ 。事件 $A$ 和事件 $B$ 彼此没有影响，叫做两个事件独立。

条件概率： $P(A|B) = P(A \cap B)/P(B)$

$P(A|B)$ 表示 $B$ 事件条件下， $A$ 发生的条件概率。

$P(A)$ 叫做先验概率（prior probability），即事态未更新时， $A$ 事件的概率。

$P(A|B)$ 也叫做后验概率（posterior probability），即事态更新后， $A$ 事件的概率。

$P(A \cap B)$ 是 $B$ 发生后 $A$ 的事件集合，而除以 $P(B)$ 是在该基础上，将样本空间的总概率重新调整为1。

当事件 $A$ 与 $B$ 为独立事件时，其中一个事件的发生并不会对另一个事件的样本空间产生影响。即 $P(A|B) = P(A)$ ,  $P(B|A) = P(B)$ 。

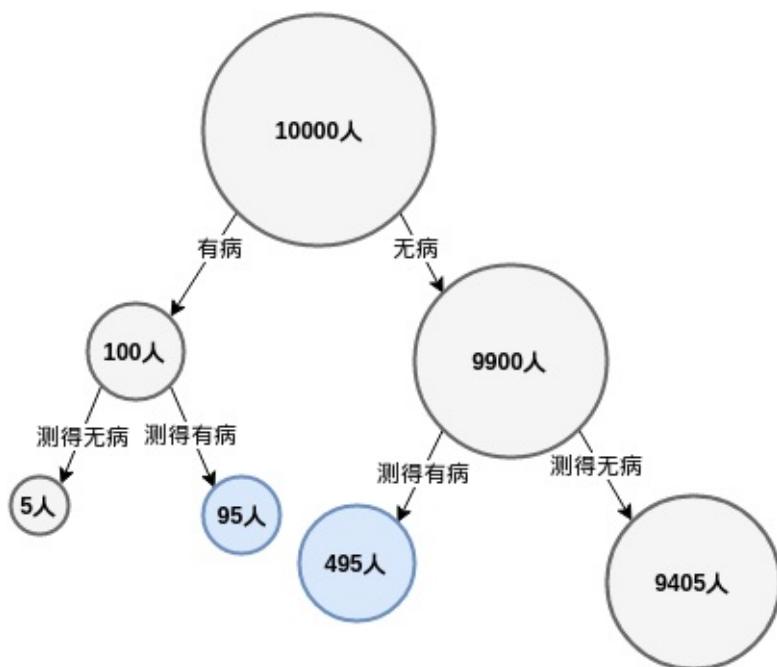
# 贝叶斯公式

人们经常将  $P(A|B)$  和  $P(B|A)$  搞混，把二者搞混的现象叫做检察官谬误（prosecutor's fallacy）。

四、 $P(A|B)$  和  $P(B|A)$  两者之间的关系是什么？

- 实例：某机器对在所有人口中得病率为1%的癌症识别率为95%（有病的人被测出患病的概率和没病的人被测出健康的概率）。一个被测得有病的人真实患癌症的概率是多少？

得出答案是95%的人就是搞混了  $P(A|B)$  和  $P(B|A)$ 。正确答案约等于16%。拿10000个人来思考。



真正的样本空间是由测得有病的癌症患者和测得有病的正常人组成，所以答案是  $95/(95 + 495) \approx 16\%$ 。

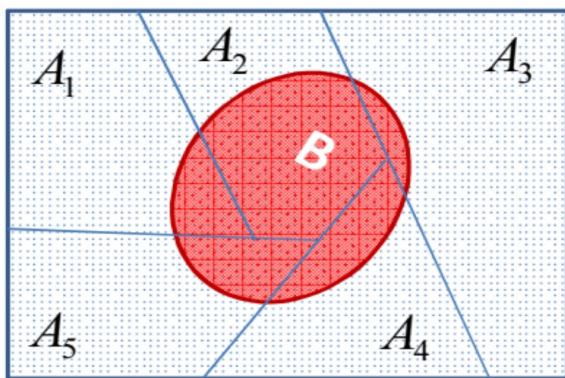
我们知道条件概率是新信息对样本空间进行调整后的概率情况，所以检察官谬误实际上是样本空间的更新产生了差错。不过我们可以从条件概率中寻找关系：通过变形条件概率的定义，就可以得出著名的贝叶斯公式和全概率公式。

贝叶斯公式 (Bayes' theorem) :  $P(A|B) = P(B|A)P(A)/P(B)$

$$P(B) = \sum_i^n P(B|A_i)P(A_i)$$

全概率公式 (Law of total probability) :

其中  $A_i$  是样本空间  $S$  的分割(partition)，即彼此不相交，并且组成的并集是样本空间。如下图：



用这两个公式，我们重新计算上面的癌症问题：

- 实例：其中  $P(A)$  是人口中患癌症的概率，为 1%， $P(B)$  是测得有病的概率。  
 $P(A|B)$  就是测得有病时，患癌症的概率。 $P(B|A)$  是有患癌症时，测得有病的概率，为 95%。 $P(B|A^C)$  就是没病时却测得有癌症的概率，为 5%。

想知道的是，当被测得有病时，真正患癌症的概率  $P(A|B)$  是多少。

由贝叶斯公式可以得到：

$$P(A|B) = P(B|A)P(A)/P(B) = 0.95 * 0.01 / P(B)$$

由全概率公式可以得到： $P(B) = P(B|A)P(A) + P(B|A^C)P(A^C)$

$$\text{全部代入就得到} : 0.95 * 0.01 / (0.95 * 0.01 + 0.05 * 0.99) \approx 16\%$$

这两个公式在机器学习中非常重要。贝叶斯公式告诉了我们  $P(A|B)$  和  $P(B|A)$  两者之间的关系。很多时候，我们难以得出其中一个的时候，可以改求另一个。

- 实例：语音识别中，听到某串声音的条件  $O$  下，该声音是某段语音  $s$  的条件概率最大的  $\arg \max_w P(s|o)$  为识别结果。然而  $P(s|o)$  并不好求。所以改求  $P(s|o) = P(o|s)P(s)/P(o)$ 。 $P(o)$  对比较同一个  $P(s|o)$  时并没有影响，因为大家都有，则不需要考虑。剩下的  $P(o|s)$  叫做声学模型，描述该段语音会发出什么样的声音。而  $P(s)$  叫做语言模型，包含着语法规则信息。

而全概率公式又是连接条件概率与非条件概率的桥梁。

全概率公式可以将非条件概率，分成若干块条件概率来计算。

- 实例：三门问题。三扇门中有一扇门后是汽车，其余是羊。参赛者会先被要求选择一扇门。这时主持人会打开后面是羊的一扇门，并给参赛者换到另一扇门的机会。问题是参赛者该不该换？应该换门。换门后获得汽车的概率为  $2/3$ ，不换门的概率为  $1/3$ 。

用全概率公式来思考该问题就可以将问题拆分成若干个相对简单的条件概率。

$P(\text{getcar})$  获得汽车的概率可以用拆分成选择各个门可得汽车的概率。 $P(D_1)$  为车在第一扇门的概率。 $P(\text{getcar}) =$

$$P(\text{getcar}|D_1)P(D_1) + P(\text{getcar}|D_2)P(D_2) + P(\text{getcar}|D_3)P(D_3)$$

$$P(\text{getcar}) =$$

$$P(\text{getcar}|D_1) * 1/3 + P(\text{getcar}|D_2) * 1/3 + P(\text{getcar}|D_3) * 1/3$$

如果不换门，得车的概率就是  $P(D_1)$ ，即  $1/3$ 。

若换门。当车在第一扇门后时， $P(\text{getcar}|D_1) * 1/3$  由于换门的选择而变成了  $0$ 。

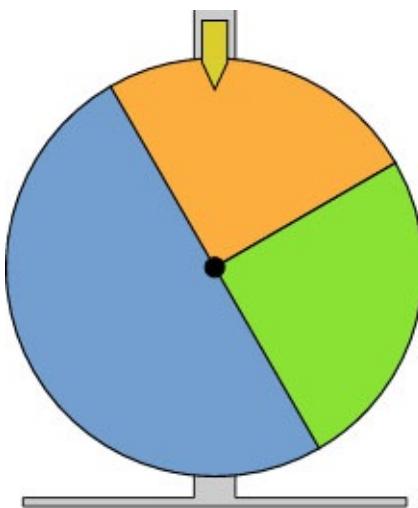
但当车在第二或第三扇门后时，由于主持人去掉了一扇后面为羊的门，换门的选择会  $100\%$  得到车。

$$\text{所以, } P(\text{getcar}) = 0 * 1/3 + 1 * 1/3 + 1 * 1/3 = 2/3$$

## 随机变量

### 五、是否有更好的方式表达事件？

随机变量是一种非常方便的事件表达方式。虽然它的名字叫做随机变量，但它实际上是一个函数。我们在“什么是概率”的例子中已经应用了随机变量的概念。我们用数字去表达事件。比较一下不用随机变量的方式。



- 实例：我们用文字去表达事件和概率。样本空间  $\Omega = \{\text{橘黄色, 绿色, 蓝色}\}$ 。

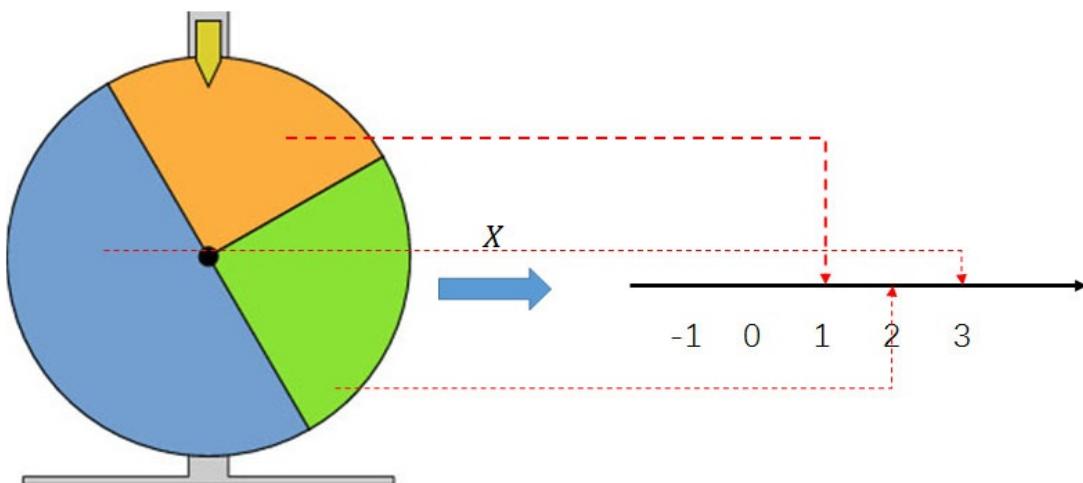
情况1：若仅仅是问转盘停止后指针指到某个颜色的概率还可以接受。如  $P(\text{指到橘黄色})$ 。

**情况2：**如果是奖励游戏，转到橘黄、绿、蓝色分别奖励1、2、3元。转3次后，想知道奖励了多少钱的概率。3元的我们要写一次描述，4元的也要写一次描述。十分笨拙。如果想问的是美元呢？我们又没办法用事件去乘以汇率。

然而如果用随机变量，就变得非常方便。设 $X_r$ 表示转 $r$ 次后一共奖励了多少人民币。 $c$ 是人民币对美元汇率的话， $c \cdot X_r$ 就表示转 $r$ 次后一共奖励了多少美元。 $X_{r+1} - X(r)$ 就表示了下一局赢得了多少人民币。

随机变量：给定一个样本空间 $\Omega$ 一个随机变量( $r.v.$ )是将样本空间投射到实数域的函数。

一个样本空间可以有很多个随机变量。在最初的例子，我们就已经将样本空间 $\Omega = \{\text{橘黄色, 绿色, 蓝色}\}$ 对应到了实数域中的1,2,3。



随机变量作为函数而言是确定的。输入事件橘黄色，一定会得到1这个输出，函数本身并没有什么“随机”。“随机”是由于函数的输入的发生概率。

$X = 3$ 表达的是指针指到蓝色的事件。 $P(X = 3)$ 表达指针指到蓝色的事件的概率。

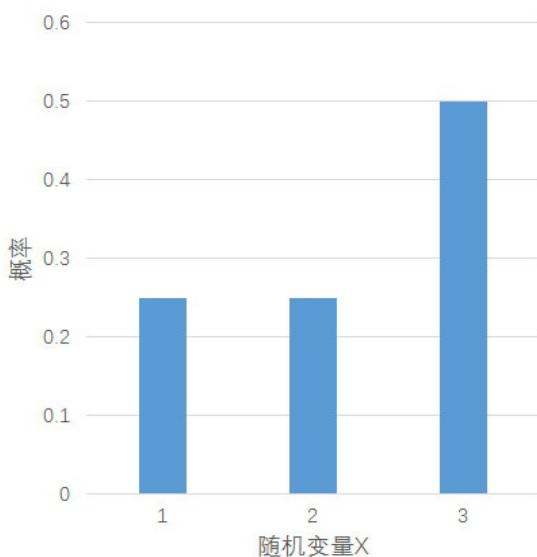
随机变量是认为事先选择的，非常灵活，好的随机变量会使问题简化许多。

根据随机变量投射后的值域是离散还是连续，随机变量可以分为离散随机变量和连续随机变量。

## 分布

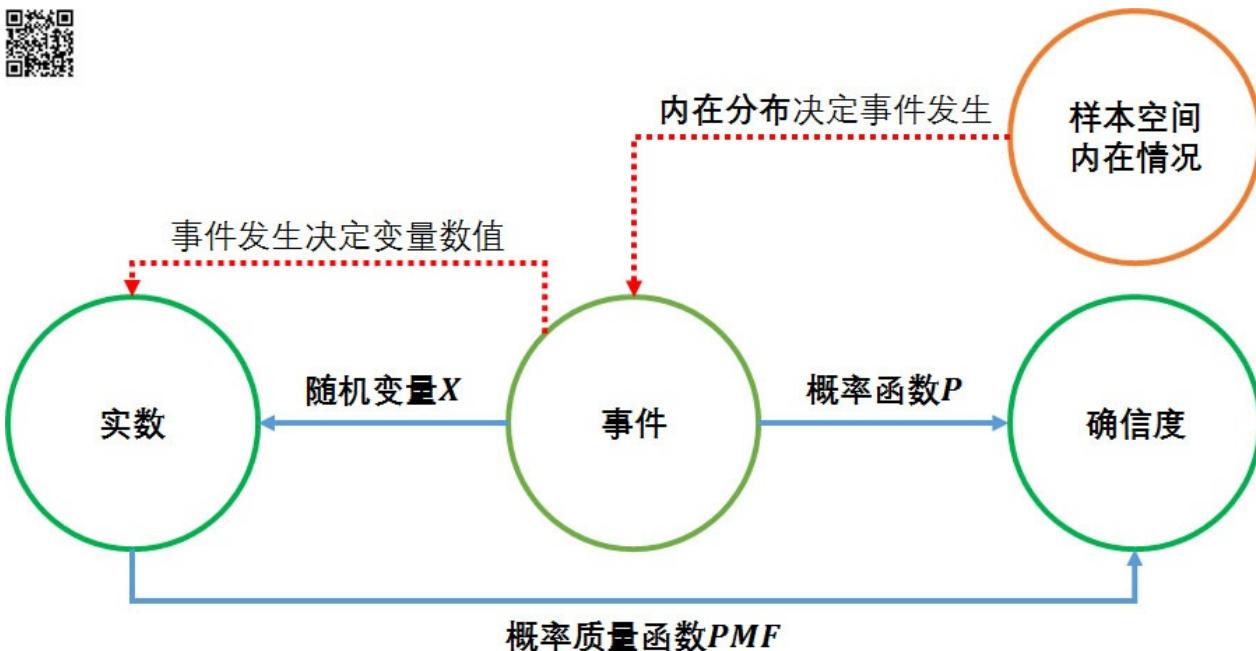
随机变量中的“随机”来自事件发生的概率。分布（distribution）是描述随机变量所对应的所有事件的发生概率的情况。

- 实例：上例随机变量 $X_1$ （转1次奖励人民币数）的分布情况用概率质量函数（probability mass function，简写为PMF）表示就是：



## 概率五要件

- 样本空间：所有可能结果组成的集合。
- 随机变量：将事件投向实数的函数。用数字代表事件。
- 事件：样本空间的子集。
- 概率：将事件投向  $[0, 1]$  实数域的函数。用实数表示确信度。
- 分布：随机变量的取值情况。



注意在应用中区分物理意义与数学定义。如随机变量虽然是以事件为输入，实数为输出。但是在用于表达概率  $P(X = 3)$  是用3这个数字去表示事件，并得出该事件的概率，并不是将实数作为输入。又如概率的数学定义是事件投射到  $[0, 1]$  的实数上，但在物理意义中，是样本

空间的内在情况决定了事件。上图中：

- 蓝线：表示人们为了描述物理现象而定义的数学函数。箭头由输入空间指向输出空间。
  - 概率函数：输入为事件，输出为 $[0,1]$ 实数
  - 随机变量函数：输入为事件，输出为实数（但使用时，用实数代表事件）
  - 概率质量函数：输入为实数，输出为 $[0,1]$ 实数
- 红线：表示真实的物理现象。箭头由因指向果。
  - 由确信度所反映的内在分布情况决定了事件的发生。
  - 事件的发生决定了随机变量的输出值。

数学定义（蓝线）中不存在随机，随机来源于物理现象（红线）。

# 熵与生命

熵和概率十分相近，但又不同。概率是真实反映变化到某状态的确信度。而熵反映的是从某时刻到另一时刻的状态有多难以确定。阻碍生命的不是概率，而是熵。

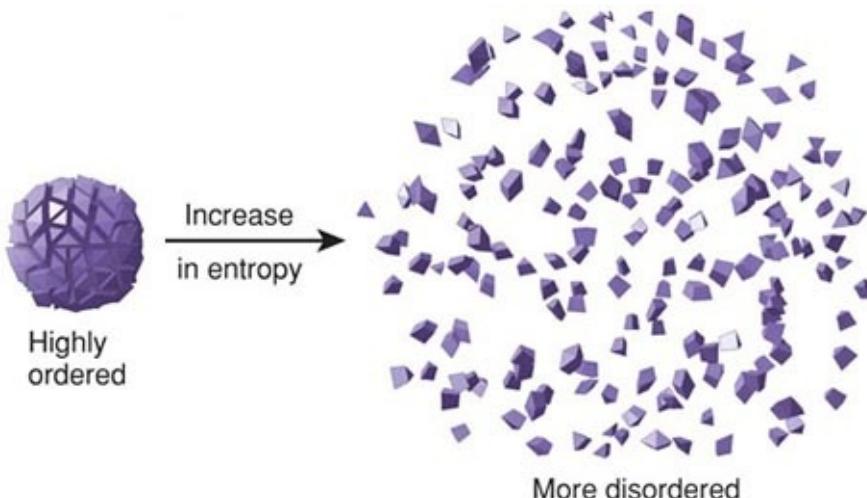
## 熵

熵是用来衡量我们对事物在跨时间后能产生不同状态的混乱度。

一、如何理解熵（随机，不确定性，不可预测性，信息量）？

当时间向未来发展时，事物可能达到的状态种类会越来越难以确定。

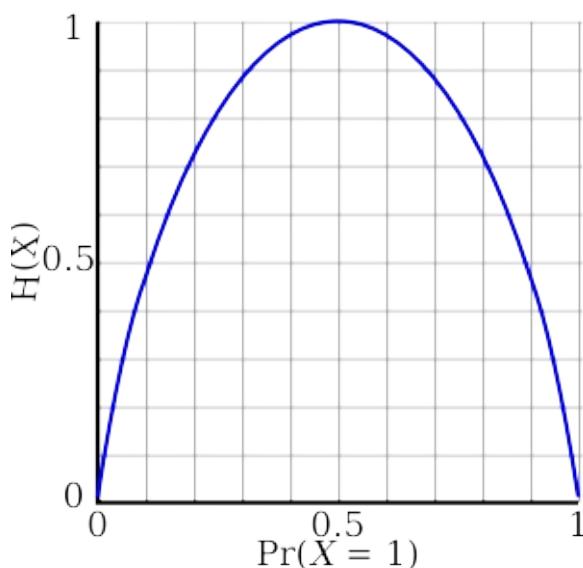
- 情景：若把下图的碎片看成可独立运动的粒子。时间由左侧的状态起进入下一刻状态时就有无数种运动的组合。从左边的状态变化后会到达右侧的哪一种状态非常难以确定。



注：这里的熵并不是左侧状态的熵，也不是右侧状态的熵，而是从左侧变化到右侧的熵。一定要有时间跨度，而有时间跨度是指有变化。

二、那么信息量（熵）又是如何衡量的？熵与概率之间的关系又是什么？

- 情景：如果上图的一个碎片有 $p(x)$ 的概率，从左侧的状态转变到右侧的某一个状态。当此概率越接近0或1时，说明它的结果就会越确定。和其他碎片组合的情况就越少。下图表示一个碎片是否到某个位置的概率与熵之间的关系。计算方式为 $H(X) = -p(x)\log_2(p(x)) - (1 - p(x))\log_2(1 - p(x))$ 。可以看到当概率为0.5时，熵最大。



越确定(**deterministic**)的事件的熵越低，越随机(**probabilistic**)的事件的熵越高。

香农熵(Shannon entropy)可表达为：
$$H(X) = - \sum_i p_i(x) \log p_i(x)$$
，其中  $H()$  是熵， $X$  是离散随机变量， $p_i(x)$  指  $p_i(X = x)$  的概率。 $\log$  的基底可以变化。一般会用自然底数  $e$ ，单位是 nat。用 2 为底数时，单位是 bit。

- 实例： $X \in \{1, 2, 3\}$ ，其中 1, 2, 3 分别表示成为红色，黄色，蓝色的概率。概率分别为 {0.5, 0.25, 0.25}。这时

$$H(X) = -0.5 * \ln 0.5 + (-0.25 * \ln 0.25) + (-0.25 * \ln 0.25) \approx 1.04$$

### 三、为什么说生命活着就在减熵？

生命要在这个变化的世界中生存，它就需要知道如何根据环境变化做出相应的行动来避免毁灭。把不确定的环境转换成确定的行动。会将无序的事物重新整理到有序的状态。生物仅仅活着就需要减熵，否则就会被不确定性会消灭。熵增意味着信息量越来越巨大，生物必须能够压缩这些不确定的信息并记住信息是如何被压缩的。

压缩信息的方法就是知识。

- 实例：病毒会利用宿主的细胞系统进行自我复制将微粒重组为有序的状态。
- 实例：工蜂用自身的蜡腺所分泌的蜂蜡修筑蜂巢。

此种知识是仅当环境发生时才会采取对应行动，并不会改变太多环境。但有第二种知识。像人类这样的动物同时还可以学习改变环境的原因以此来预测未来。

- 实例：由于动物会运动，环境会根据它的运动和它对环境的影响造成改变。动物需要可以利用记忆力从过去推测未来事件的能力。
- 实例：物理学家穷其一生寻找可以解释一切的公式(像  $e = mc^2$ , 弦理论)。这种能力存在于拥有时间观念的生命之中。人类更是其中的王者。

但这并不意味着智能越多越好。取决于智能如何被使用。如人类可以用第二种知识去改变环境，但会带来不确定性。于此同时我们又需要更多的智能去对抗不确定性，并改变了更多的环境。科技从未使我们的生活变得更好，它仅使生活变得复杂。是环境的复杂化造成了很多学生不得不学习更抽象的知识，不得不获得更高的学历来提高自己的生存能力。当平衡被打破时？我们将会失去部分智能，新的平衡将会尝试着建立。

许多已灭绝的生物曾经拥有他们与不确定性的平衡，直到人类将他们的环境改变。

# 智能的条件

## 智能LV1

一、生命又是如何仅通过活着便减熵的，需要哪些条件？

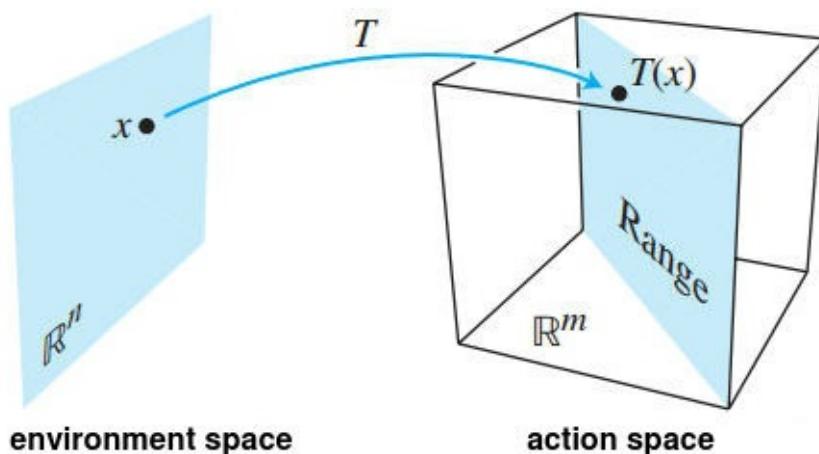
该问题等同于如何生存。

首先要知道何时根据环境做出变化，也就需要从环境到行动的关联，并且可以在未来执行行动来躲避不确定性。

从生存的视角来看：

智能LV1的基础是拥有能再次利用从环境到行动的关联的能力

从环境中得到，意味着并非先天固有，同时又不是永不改变。当环境变化时，关联也需要跟着变化。才符合根据环境变化而变化的特点。可再次利用，包含着两个意思。第一，得到的关联可以用于躲避危险，帮助生存，也就是预测。第二，需要将所得到的关联保留下来。总结一下我们就获得了这种关联的三个条件：



1. 学习（建模）：学习自发性使得  $H(Y)$  尽可能低的关联（或叫映射、函数）  $T : R^n \rightarrow R^m$ ，其中  $R^n$  是“环境空间”， $R^m$  是“行为空间”， $H()$  是熵， $Y$  是随机变量  $T(x) \in R^m$ 。关联的产生必须要从环境中得到，当环境变化时关联也会改变使得熵变低，是一种单向动态关联。
2. 应用（识别）：基于环境执行行动。
3. 储存（记忆）：学到的关联需要存储媒介，否则无法保留。

智能LV1属于应激性行为。所有微生物和植物都主要以该种智能为主。

## 智能LV2

智能LV1中的关联只能够是环境到行为的关联。而智能LV2的关联可以是过去状态到未来状态的关联，即预测。

| 智能LV2的基础是拥有能再次利用过去到未来的关联的能力

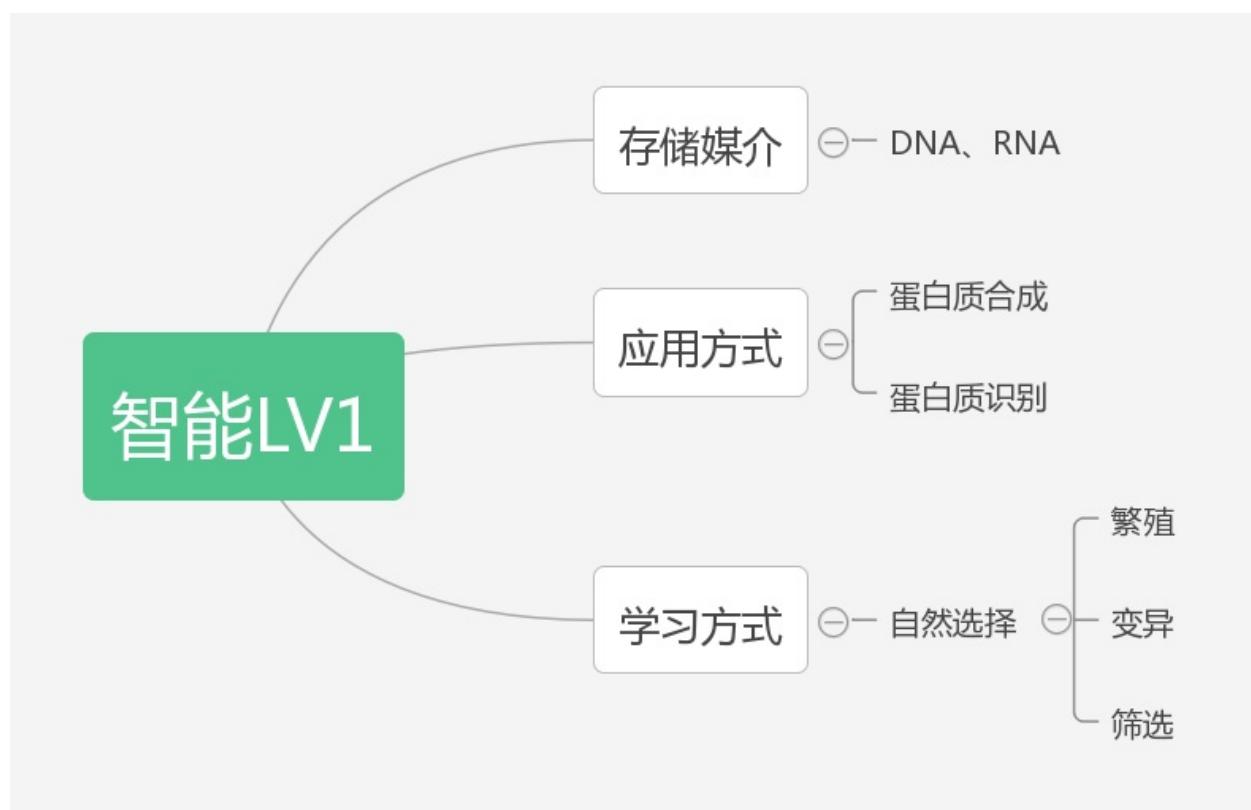
智能LV2大多存在于动物之中，通过预测下一刻的事件决定自己的行为。

## 智能在自然界的实现

目前描述的智能依然仅是一个概念，它是如何在地球上实现的？

在智能的三个条件中，存储关联的材料是对于一个星球可否延续生命最为重要的一点。在地球上，我们也叫其为遗传物质。过去可能产生过很多种遗传物质，但无疑DNA是地球上最为成功的遗传物质。它使得智能的实现成为了可能，而后所有的高级智能也都建立在DNA基础上。

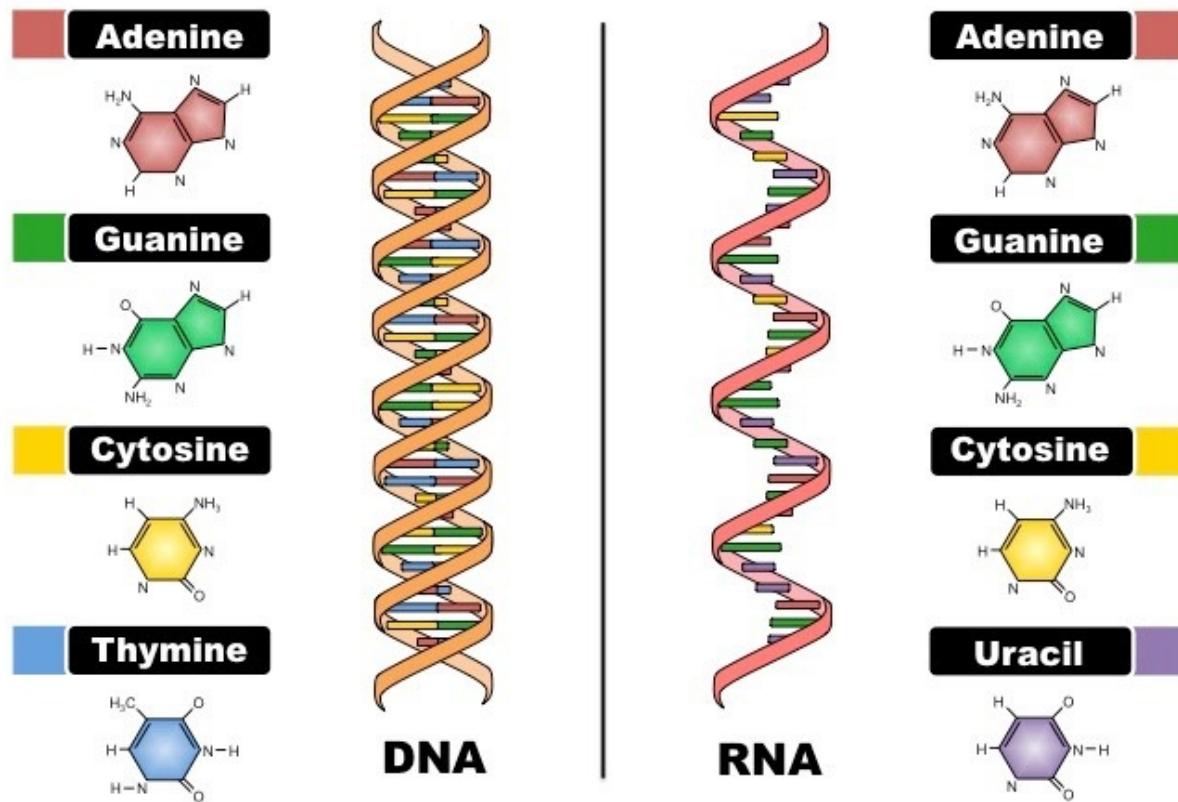
# RNA与DNA



智能LV1可被DNA、RNA和蛋白质等物质实现。

## 存储

这个神奇的结构存储着所有从环境中学来的信息。

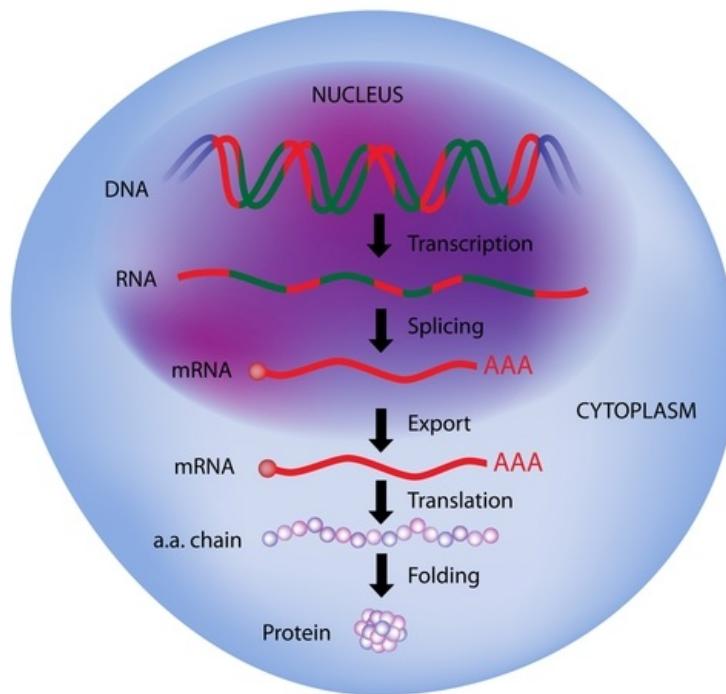


## 识别

一、DNA可以存储关联，但这些关联是如何基于环境产生对应行为的？

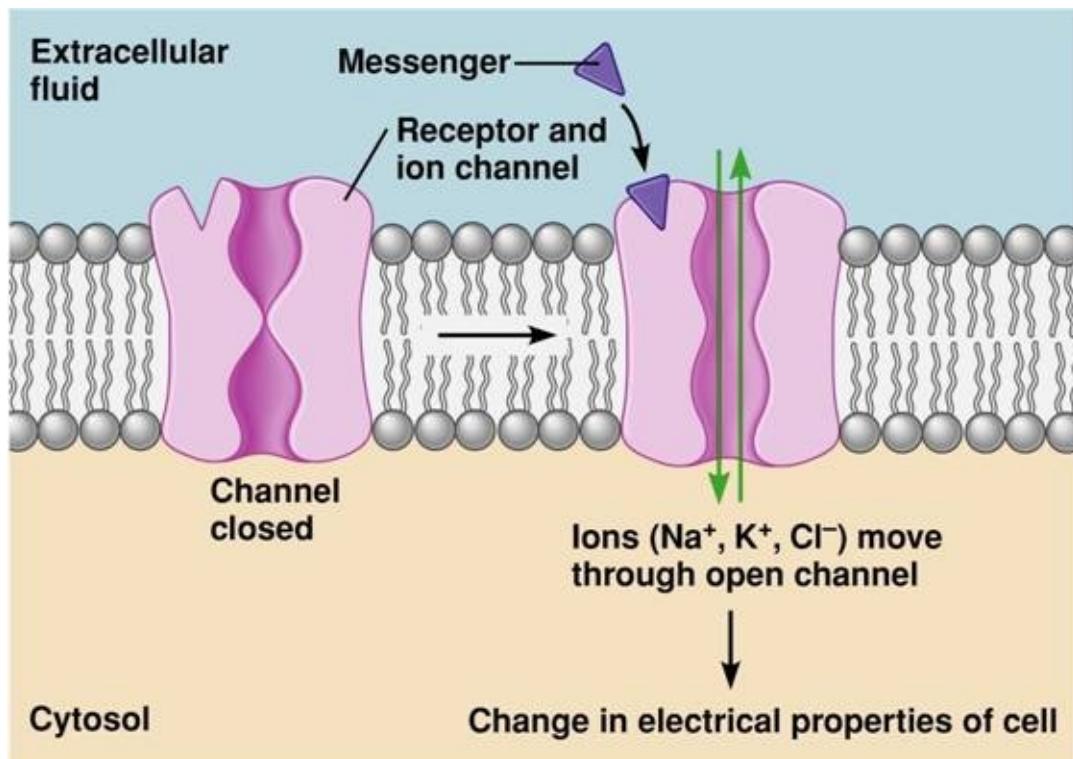
答案是蛋白质合成。

存储在DNA上的信息控制着蛋白质的合成方式。



蛋白质可以被视为由DNA打造的实现预测的“工具”。

- 实例：某些离子通道（Ligand-gated ion channel）只有当特定的神经递质契合在它的接受器上时会允许 $\text{Na}^+$ ,  $\text{K}^+$ 等离子通过细胞膜。



这就是一个二类分类器，并且是和微小粒子进行直接交互。其中，环境空间就是周围的粒子，而行为空间只有“开”和“关”。

这些分类器看似弱小，但当大量蛋白质并行预测时就实现了特定功能的细胞。而大量细胞又形成了更高级功能的组织，以此类推，从而形成了各种的复杂功能，支持我们人体的各项生理活动。

二、很多其他物质也可以识别特定物质的能力。为什么他们不叫智能？

- 实例：磁铁对磁场的识别就不叫智能。因为他们的关联并非从环境学来，也不可以更新。但是蛋白质是由DNA的编码控制，可以通过自然选择更新关联方式。

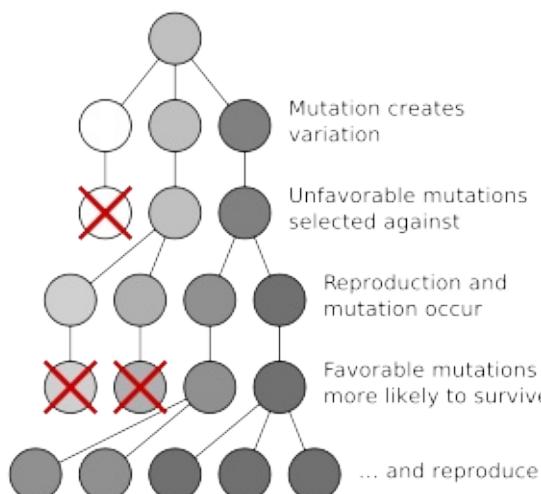
## 学习

我们知道DNA存储着关联，而蛋白质可以实现关联。

三、关联又是如何学习的呢？

答案是[自然选择](#)。共有三步：

1. 繁衍：比如，细胞会分裂成更多的细胞来延续关联。
2. 变异：然而分裂时DNA的序列并非完美复制，复制的错误会产生变异。
3. 筛选：在变异和非变异中，无法适应环境变化的细胞会死亡，而适应环境的细胞会继续繁殖。



这种叫做自然选择的动态过程就是学习。遗传算法的灵感来源。

从中可以意识到，“死亡”在这种智能系统中是注定的。

也能意识到蛋白质和DNA对于生命的意义。

请一定要观看该[视频](#)体会DNA与蛋白质等物质是如何大量并行识别与粒子直接交互将化无序为有序的。

## RNA世界学说

DNA依赖于蛋白质繁衍，而蛋白质需要DNA的关联去执行功能。现在我们有了“鸡和蛋”的问题。蛋白质和DNA，谁先产生的？

然而有第三种分子：RNA。

RNA既可以存储关联，又可以执行预测。这就是为什么大多数科学家相信RNA是最先产生的遗传物质。这种认为RNA使得生命成为可能的想法叫做RNA世界学说。

但是这种的学习方式的速度非常慢，并且建立在不断“死亡”的基础之上，也容易跟不上环境的变化而灭绝。

## 外星生物

四、是否有外星生物？

做一个最极端的假设，假设没有外星生物，那原因是什么？恐怕在于第3条：存储媒介。

DNA双螺旋是我所认为宇宙中最神奇的现象，神奇到让人很难相信它是自然的产物。它的出现奠定了地球上所有生命的可能。即便是病毒也是由保护性外壳包裹的一段DNA或者RNA。电影《普罗米修斯》中外星人给地球留下的“火种”就是“DNA双螺旋”。

如果说外星没有生物体。那一定是没有类似DNA这种既能完成智能的延续，又能稳定的适应那里的星球的环境的遗传物质。

其次，我们怀疑外星是否有生物的原因是因为没有得到任何联络。外星可形成的生物未必进化出了意识。可能全部都是植物。

# 进化

达尔文曾被孔雀尾巴伤透了脑。并且物竞天择适者生存难以解释生物自杀现象。于是《自私的基因》一书给了人们另一种思考：进化可能是基因。然而真的是基因吗？

## 常见误区

- 实例：很多母螳螂在交配过程中会吃掉公螳螂的大脑。有人猜测这样做的目的是为了消除公螳螂大脑的抑制机能，确保精液持续注入体内。虽然公螳螂个体死亡，但却利于子辈迅速繁衍，壮大种群数量。

生物会不断更新与环境的关联来延续自身。种群内个体数量越多，能产生的变异就越多，可以防止环境的变化筛选掉所有的个体，使得物种灭绝。个体只是生命的一个实例。

### 进化以种群为单位

- 实例：设想两批渔民，一批用大孔渔网捕鱼，另一批用小孔渔网，若干时间后，用大孔渔网捕鱼的渔民由于小鱼的存活得以不断靠捕鱼维持生计，而小孔渔网捕鱼的却因为河里的鱼群死绝最终饿死。而当时的渔民根本无法预测会发生什么后果。
- 实例：即便是当今人类，也无法预测某项举动会带来什么后果。比如爱因斯坦并不知道自己的理论会造出核武器。再如没人能够预测未来人工智能带给人类的将会是进步还是毁灭。
- 实例：想象一下，一个建筑里居住着20个人。某天其中的3人回老家探亲，2人外出采购食品。其余的人选择呆在家。

平行空间A：恰巧那天建筑在没有任何先兆下倒塌。建筑里的人无一幸免。而做出探亲和采购的举动的5个人存活了下来。

平行空间B：恰巧那天选择呆在家里的人安然无恙，而外出的5个人由于空气污染染上了肺癌，全死丧生。

事实上这些人在做出决策时根本不会知道他们的举动会带来什么后果。并非他们选择存活，而是他们的某些举动无意间避开了一次危险。

进化不以个体为单位，进化也并非由意识来推动。

某个物种的分支存活下来仅仅是因为他们偶然产生的某种倾向使他们没有被当时环境所毁灭。进化这个词并不适合描述这个现象，因为进化是一个主动动词。更合理的方式是用被动式。

### 进化是被动的

# 进化的对象

## 一、究竟什么在进化

- 实例：1000条蛇，900条拥有与人类相同的光感视觉(将反射光与物体进行关联)，100条拥有热感视觉(将热量与物体进行关联)。

平行空间**A**：这1000条蛇的食物都是日行动物。由于热感视觉的速度不如光感视觉的快，100条蛇因抢不到食物而被饿死。剩下900条蛇不断交配繁衍，使得蛇的视觉系统为光感视觉。

平行空间**B**：这1000条蛇的食物都是夜行动物。由于夜间无法探知反射光，900条蛇因为看不到猎物而被饿死。剩下100条蛇不断交配繁衍，使得蛇的视觉系统为热感视觉。

这两种关联方式，并无好坏，决定它存留的仅在于是否适合当前环境。若平行空间真的存在，有的蛇极可能和人类拥有相同的视觉系统。

DNA的功能是保存并延续智能关联，而不论是个体还是种群都是智能关联的实例。

### 进化的对象是智能

- 实例：核辐射产生的变异原因是因为核辐射会打乱、破坏DNA基因序列，而基因中的序列是进化了亿万年的智能关联方式，这种改变的结果往往是无法再与身体的其他智能关联相互协调，从而使整个生命系统崩溃。

### 进化的过程是智能关联不断被筛选的过程

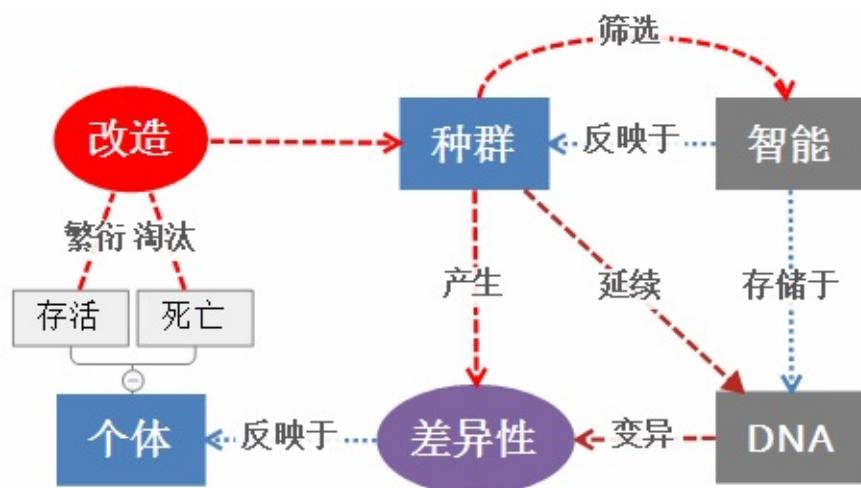
无法知道将来的世界会变成什么样子，所以进化的方向也难以预测。但一点却可以确定，那就是进化会使生物个体产生更多差异性的方向前进。因为这样扩大了样本，增加了存活几率。

名称	作用
智能关联	进化的真正对象
进化	不适应环境的智能的淘汰过程
基因	保存智能关联的媒介
个体	智能关联的具体反映
种群	进化的单位
差异性	进化的前提

## 二、为什么有性繁殖成主流？

关键在于差异性所带来的智能筛选。简单罗列一下物种产生差异性的能力

名称	描述	产生差异性能力
病毒	仅仅是不断复制自身	极弱
无性繁殖生物	繁衍速度快，但很容易因无法适应新环境而灭绝	弱
有性繁殖生物	基因突变和基因表现等多种产生差异性的手段	强
不可移动生物	个体的差异性	弱
可移动生物	不同的生活环境对个体的差异性任然有强化作用	强



智能存储于DNA之中。DNA在复制的时候会产生变异从而增加差异性。这种差异性反映于个体。不适合的个体会被环境所淘汰。存活的个体继续繁衍从而改造种群。而种群间的有性繁殖又会产生更多的差异性。种群的存活决定着DNA是否可以被延续。从而完成这种关联的筛选。

### 三、永生有什么弊端？

人类的“怕死”也是存在于DNA中的智能，有利于种群的延续。若理解了上文对进化的描述，便可以很容易想清楚其中的理由。被延续的是智能关联，一个个体只为种群提供了差异性。

- 实例：永生和胎生的两种延续方式。

吸血鬼：

永生，这里的永生指得是不会老死。严重的物理打击依旧会使其丧失生理功能而死亡。

可以生育。但无限生育必将造成子辈和父辈争夺生存资源。所以他们的数量也是保持相对稳定的。

人类：

身体所有构造和吸血鬼相同，但是却会老死。

靠一代代生儿育女的方式来延续智能。和吸血鬼都居住在地球。

突然某天地球上出现了一种前所未有的病毒。对拥有相同身体构造的人类和吸血鬼都造成了致命打击。

虽然吸血鬼不会老死，却依旧抵不过这种突发其来的灾灭。永生的吸血鬼很可能无一幸免。98%的人类也因此死掉。由于人类所用的是代代生育的智能延续方式，其产生的差异性，非常偶然的出现了2%可以抵御此病毒的变异体质。

永生的吸血鬼的风险如此之大的原因是，自然选择中的三步里缺少了筛选这一步，使其变异速度无法跟进环境的变化速度。

- 类比：好比玩闯关游戏，胎生相当于游戏有多个存档点，频繁存档，死亡后会从存档点开始。而永生相当于只有一个起始存档点，死亡后就要重头开始。这里的每存一次档就相当于更新一次基因库，胎生相当于强制生物频繁存档。游戏死亡是指环境改变造成的打击。

自然选择的实质是一个动态的从环境中学习关联过程，而进化达到的结果是自然选择中  
第三步：筛选。

# 学习的本质

我们每天都在不断学习（生物的学习），讨论学习。人人都希望能够有快速学习的本领，市面上因此有了大量关于“学习法”的教材。但是学习的本质究竟是什么？学习的对象是什么？

之前的章节讨论过：智能是减熵的能力，依靠学习来消除不确定性。

由于学习就是熵增的逆过程。为此我们应该首先搞清熵增的过程，然后再思考该其逆过程。

下面是思考流程图：



## 一、熵为何增加？

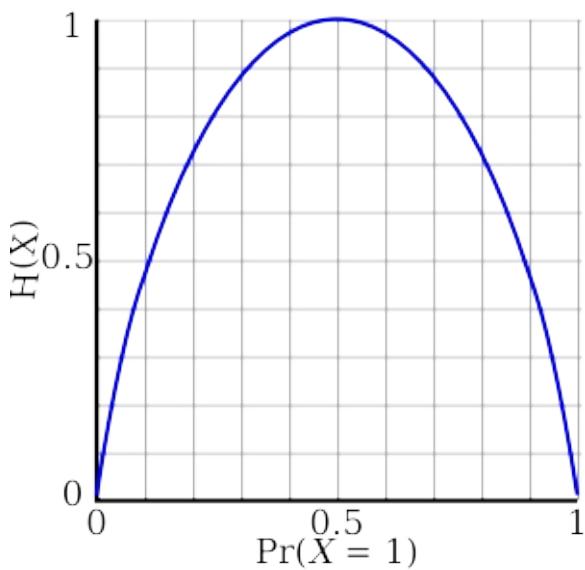
熵描述的是状态的不确定性。

但有必要再次加深理解“不确定性”四个字。

如果单纯地认为有序的事物的熵就低是不对的。因为即便是无序，若状态为确定性（即概率为0或者1时），熵就是0。熵所描述的一定是有概率的内容。回想起公式

$$H(X) = - \sum_i p_i(x) \log p_i(x)$$

和两种状态的熵图（y轴为熵，x轴为概率）：



- 实例：当下图的白球击中彩球时，彩球会散成哪种排列状态？

若简单将空间也想象成100个空心小球，可以将问题转化成从100个球中选15个球的排列问题。但熵并不是指有多少种排列状态，而是变化后的状态有多么难以确定。熵描述的是状态难以确定的程度。

若将击球前一刻作为起始时间 $t_0$ 。由于知道起始状态，所以该事物在 $t_0$ 时的熵为0，击球后时间变成了 $t_1$ ，在不观察的情况下设想可能到达的状态时，这时才有了熵。但观察后，由于已经确定了，熵瞬间又变为0。

注意：描述的问题是从击球前到击球后的状态的 $p(t_0 -> t_1)$ ，是变化后的状态。



熵描述的是不确定性。那么熵增就是不确定性的增加。

感受一下不确定性的增加过程：

- 实例：设击球瞬间为时刻 $t_1$ ，这时的状态并不是非常难以确定，我们知道白球会与前排的彩球接触（假设击球手不是门外汉）。如果在不知道 $t_1$ 时刻接触到哪个球的基础上发展1纳秒到 $t_2$ 时刻，这时被撞击到的彩球会再次向其他彩球撞击。若在不知道 $t_2$ 撞击情况的基础上再次发展1纳秒到 $t_3$ 时刻，你会发现，随着时间的推移。由于不知道 $t_1, t_2$ 的情况，状态会越来越难以确定。熵是这样一点一点累积增加的。

注意：这里的熵增加的是指从 $t_1$ 到 $t_3$ 的熵而不是， $t_1$ 到 $t_2$ 或 $t_2$ 到 $t_3$ 。

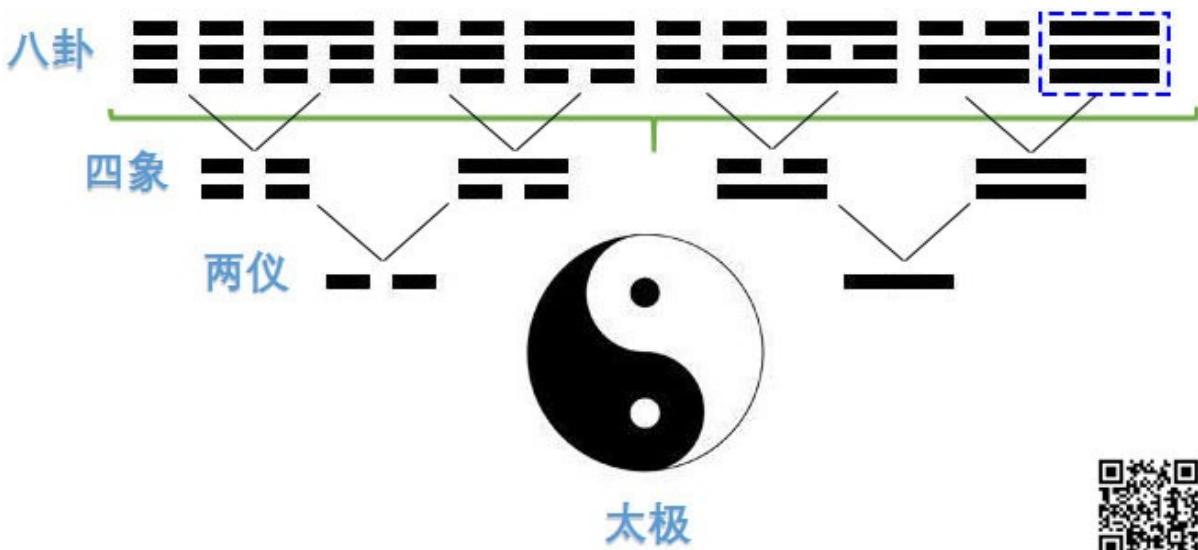
## 二、不确定性为何会增加？

我们通过刚才的例子感受了不确定性是如何增加的。现在感受一下量子计算机和传统计算机的区别，随后我们就会明白是什么带来的不确定性。

量子计算机的强大在于计算时的额外的信息量。

- 位 (bit)：经典计算机中信息的最小单位，表示0或1状态。
- 量子位 (qubit)：量子计算机中信息的最小单位，表示0和1共存状态。

- 区别：关键的区别在于确定性与非确定性。
- 实例：对于太极状态的存储：



经典计算机：状态全部都是确定的。当有1位时，状态要么是1，要么是0，不能共存。

- 若想穷尽两仪，需要2个1-bit（第一个数字是量词，第二个数字是形容词）。
- 若想穷尽四象，需要4个2-bit，
- 若想穷尽八卦，需要8个3-bit。

量子计算机：状态是共存的。当有1量子位时，状态可为1和0共存。

- 若想穷尽两仪，需要1个1-qubit。
- 若想穷尽四象，需要1个2-qubit。
- 若想穷尽八卦，需要1个3-qubit。

如上图中蓝色虚线所框，经典计算机中，1个3-bit，仅可以表示八卦中的1个状态，而量子计算机中，1个3-qubit可以表示八卦中的所有状态。

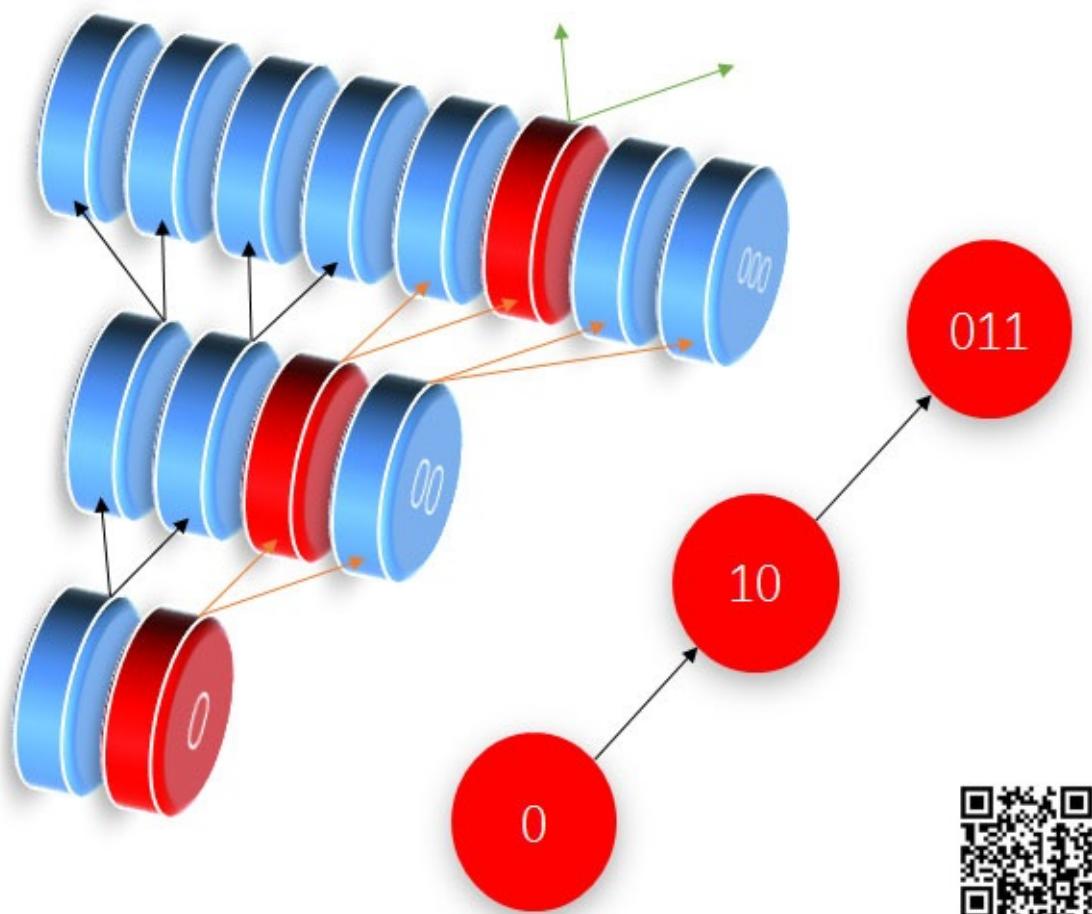
带来这种“额外存储量”的是状态的不确定性。

尽管1个3-qubit可以带有8个状态的信息，但当我们观察后，不确定状态立刻变为确定状态，由不确定性所带来的额外存储量便会消失。经典计算机可以理解成量子计算机的一种特例。

- 图解：在不观察的情况下连续从太极变化到八卦。

传统计算机：红色部分，特点是所有状态都是确定的，相当于观察每次变化后的状态，那么所有可能发生的状态信息（蓝色）都会丢失。

量子计算机：可保留所有可能发生的状态的信息，并在其基础上再次增加分支。但即使是量子计算机，当观察八卦的状态后，蓝色的信息就不再累积（如绿色箭头）。



你会发现随着量子计算机存储量的增加，其不确定性也在增加。实际上：

不确定性的增加就是信息（新状态）的增加。

其之所以难以理解是因为人类的认知所在的维度并非信息的全部。

所有的信息是如上图一样，包含了所有已发生和可能发生的信息。而经典计算机只有已发生的状态的信息。

### 三、新信息如何产生？

智能LV2是用于预测未来事件的能力。所预测的是未来会达到哪种状态。那么新状态如何产生？或者说：信息如何产生？

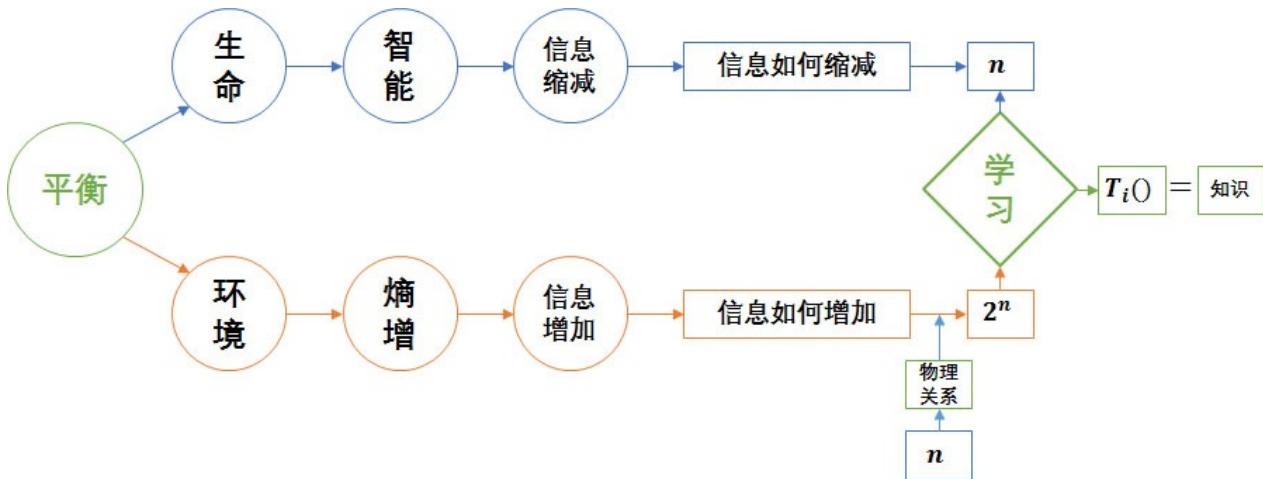
- 实例：如上例的太极图，设想只有2种状态的事物。当有1个事物时，有两种可能状态。当有3个事物时，所能产生的状态并不是 $2 \times 3$ 种这样简单累加，而是 $2^3$ 种。 $8 - 6 = 2$ ，这额外的2从何而来？新的状态是由四象的各项状态上分别再增加2种可能状态，而非简单累加。

信息是由事物彼此间的关系所产生的排列状态。在自然界中，所提到的关系由我们世界的物理规律所定。当因素个数增加时，就有更多事物彼此间的关系需要考虑。

新信息是因素间通过某规律形成的排列状态。

### 四、如何减熵？

知道了熵的增加是不确定性的增加，不确定性的增加是信息的增加，而信息的增加是原有状态通过关系所造成的新状态的增加。我们终于可以思考熵增的逆过程了，也就是学习的本质：



熵增是不确定性的增加，不确定性的增加是信息的增加，信息的增加是因素间关系的增加。

学习的过程是因素间的关系的拆分，关系的拆分是信息的回卷，信息的回卷是不确定性的缩减。

学习的对象是知识，并非信息。

生命需要降低周身环境的熵才能够存活下去。智能是减熵的能力。熵的增加实际上是信息的增加所造成的难以确定。并非我们丢失了信息，而是旧的状态通过物理关系所产生的新状态。学习就是将这些新状态重新拆分成旧状态与物理关系。这时所学到的物理关系就是知识。又如台球的例子， $t_3$ 时刻的新状态是由 $t_2$ 时刻的旧状态产生。 $t_2$ 时刻的旧状态由 $t_1$ 时刻的旧状态产生。由于自然界中信息的产生是在原有状态的基础上不断迭代产生。那么自然界中的学习很顺理成章的是多层迭代的，也就是深层。

#### 四、日常生活中的学习有什么问题？

每个人生下来就是科学家。We are born scientists.

我们出生后就充满了对这个世界的好奇，会发出各种各样的问题。那些让家长们都头疼的问题。当男生想要讨得一个女生的欢心时，很多男生都会阅读书籍，查阅资料。当大家遇到一个趣味题的时候，每个人本能的都想去解答，有的更进一步会从数学书中寻找答案。

每个人都善于学习。学习是人类的本性。

然而当我们到达初中时，好奇心就开始被逐渐碾碎。原本，知识应该是两类事物的关联，能够压缩信息的方法叫做知识，而学习是探索和研究这种关联的过程。

我们很幸运的拥有前人已经探索过的知识。然而这种幸运却由于教学的不当，逐渐将知识变成了信息，将学习变成了记忆。而数学也从思考方法变成了毫无“用处”的逻辑游戏。我们不再问为什么，不再关心知识是如何产生的。只要记住、背住前人留下的知识，甚至不需要会应

用，只要可以在考试时玩好这个逻辑游戏就好。造成了学到的“知识”根本不会用，认为记忆就是学习。这便是我们如何从会学习变成不会学习的。

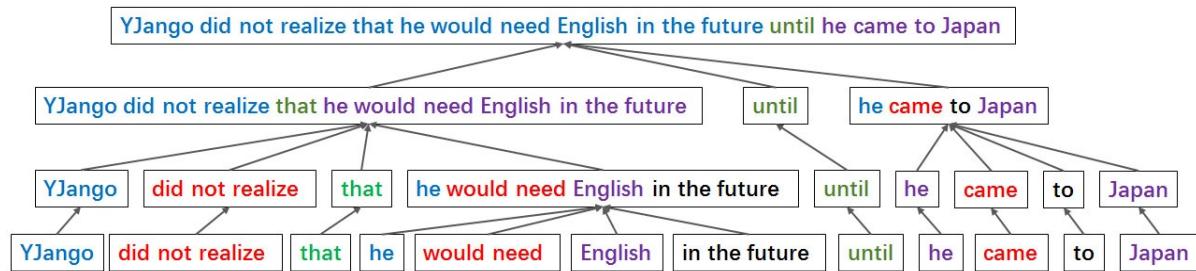
## 深层的应用

即便在日常学习和工作中“深层”原理也每时每刻被人们所使用。不同的领域会给它不同的名字。虽然说不清什么是学习，但是我们时刻都在应用着。因为生物以负熵为生，学习就是我们的本性。

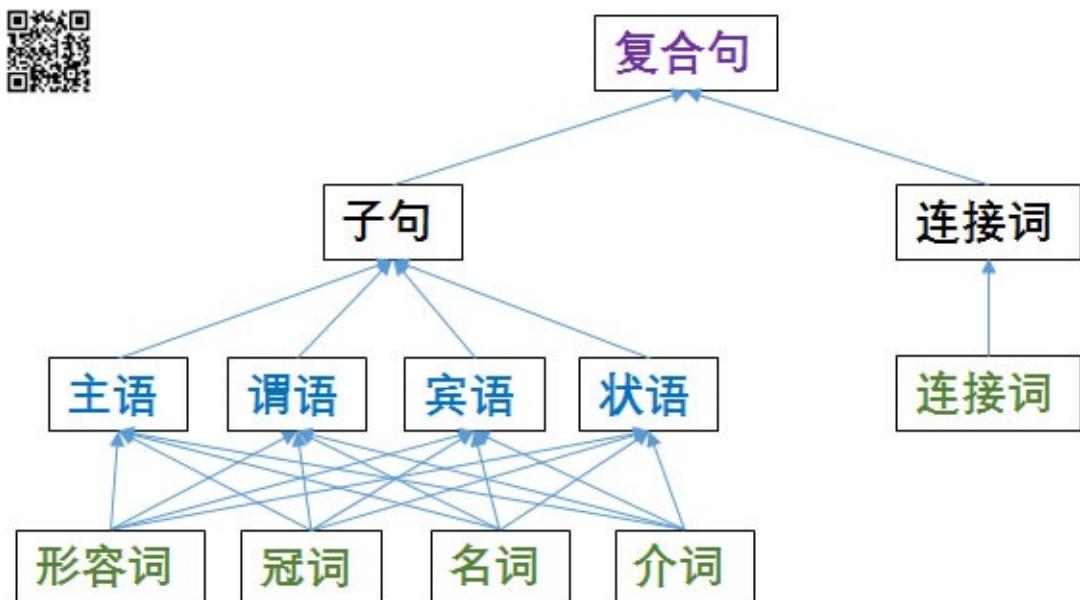
### 一、什么是深层原理？

深层原理的流程就是当人们遇到一个综合问题时，尽可能的将其拆分成若干个独立的子问题，反复应用该原理，直到分解为可以轻松实现的子问题。而最终结果就是这些子问题的组合。因为综合问题是由于子问题通过关系所形成的。

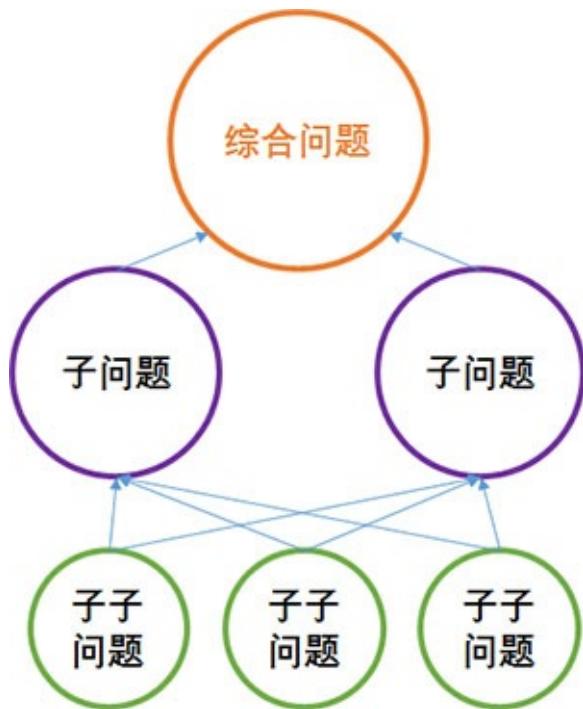
- 英语：对于下面的英语句子我们是如何分析的？也就是拆分关系。一个复杂句可以拆分成简单句的组合，而简单句又可以继续拆分各种成分，成分又可以拆分成不同的词类。



对所有的句子都可以进行这样的拆分。一个有无数种变体的难以确定的复合句，可以拆成子句和连接词。然而子句依然有变体，于是我们又将子句拆成了主语、谓语、宾语等各种成分。主语仍然有变体，于是我们继续拆分。



- 编程：编程中深层原理有一个更好听的名字。是古人早已有的思想：分而治之。



这是编程人每天都在用的法宝。将难以实现的复杂功能拆分成简单的功能再合并。我们再清楚不过这样做有什么优势了：

- 解决问题：拆分后，原本看似毫无头绪的任务变得可能。
- 照顾变体：用这种方法编写的程序，不需要为每个变体功能而重新写一遍程序。功能拆分的越细微，也就意味着“层数”越深，而对变体的照顾就越周全。

这就是深层原理。

# 神经元

智能LV1由自然选择来实现学习，受限于环境。只有当环境变化时才产生行为，属于应激反应。智能LV1对于微生物，植物这样移动微弱的生命而言或许足够。但对于可以大范围移动的生物而言就不足以躲避危险。

由于移动的生物会使自身周围的环境产生变化。这种环境可能是该生物从未见过的，极大的增加了死亡风险。因此决定是否移动，向哪里移动就至关重要。这就要求生物有可以学习从“过去状态”到“未来状态”的关联的能力。

智能LV1的功能是识别，而智能LV2的功能是预测

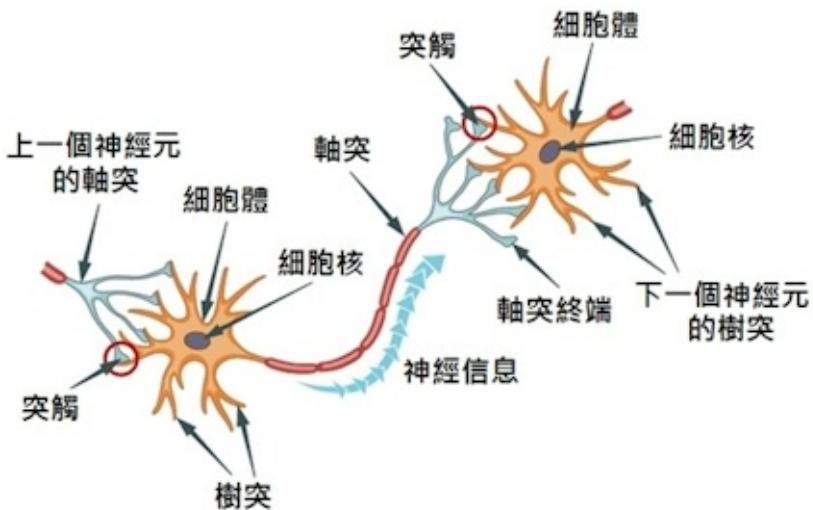
实现智能LV2的基础就是要有记忆能力。无论短期记忆还是长期记忆，只有当可以将“过去发生的状态”记住时，才能够实现从过去关联到未来。

神经元便可以实现智能LV2。

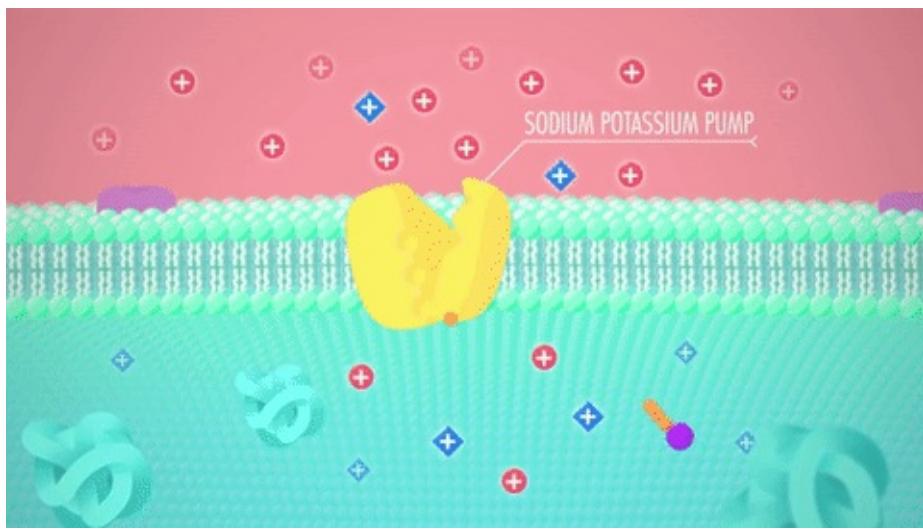
## 一、神经元是如何工作的？

请回忆起离子通道。蛋白质的识别可以控制电离子的流动，从而控制电信号，完成信息交互。

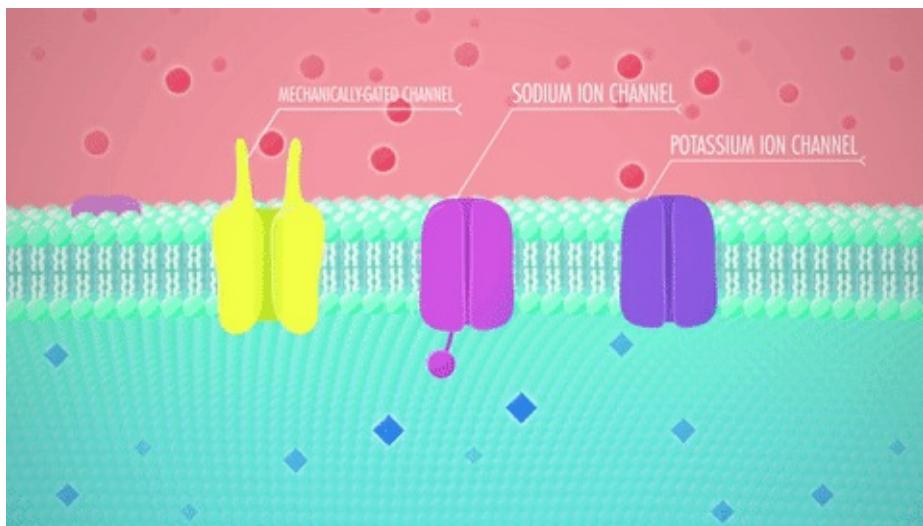
1. 连接：神经元本体（cell body）上有很多树突(dendrites)，这些树突会和其他的神经元细胞连接，形成突触（synapse）。



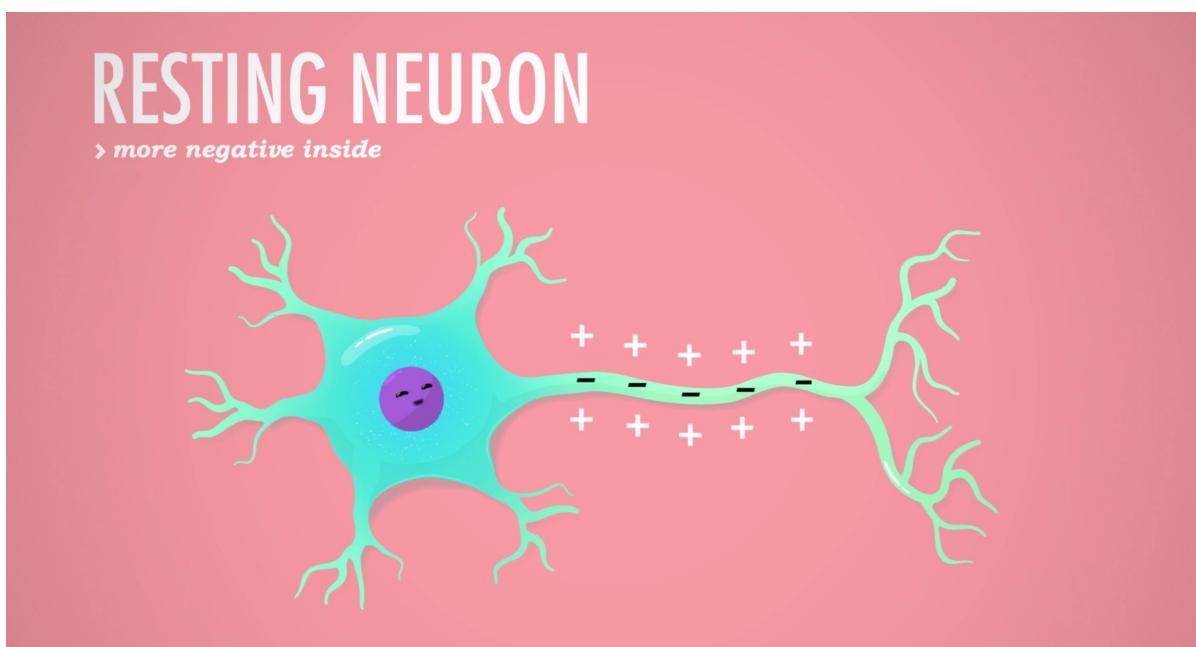
2. 维持静态：神经元细胞内有一种叫做钠钾泵（Sodium potassium pump）的蛋白质。ATP（右下方紫色）会提供能量让钠钾泵排除三个钠离子（Na+），放进两个钾离子（K+），使神经元细胞的膜电位（membrane potential voltage）始终处于-70mV。



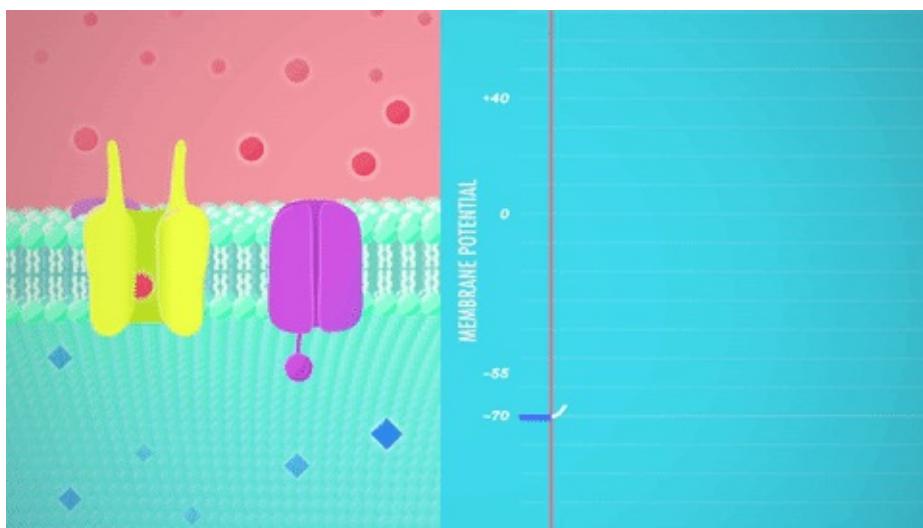
神经元细胞膜上还有很多其他离子通道。当所有离子通道都是关闭状态时，



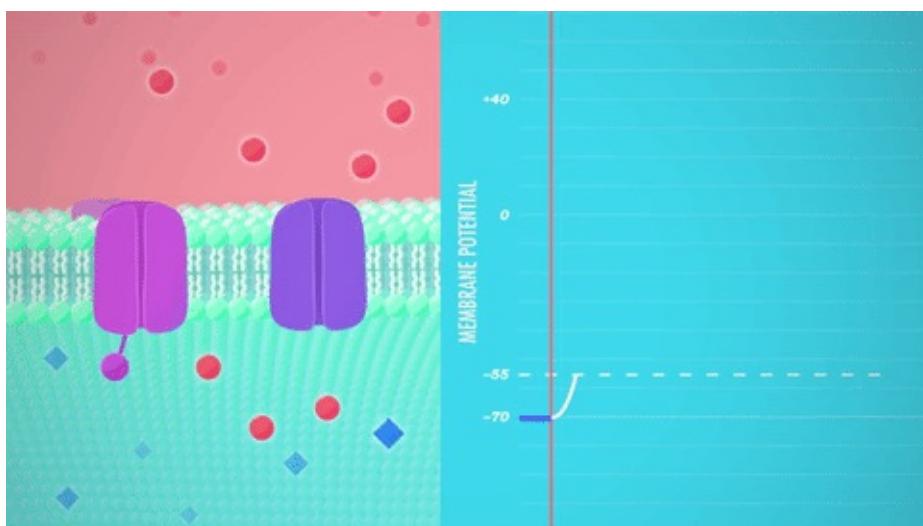
神经元处于静态电位（Resting Potential）。



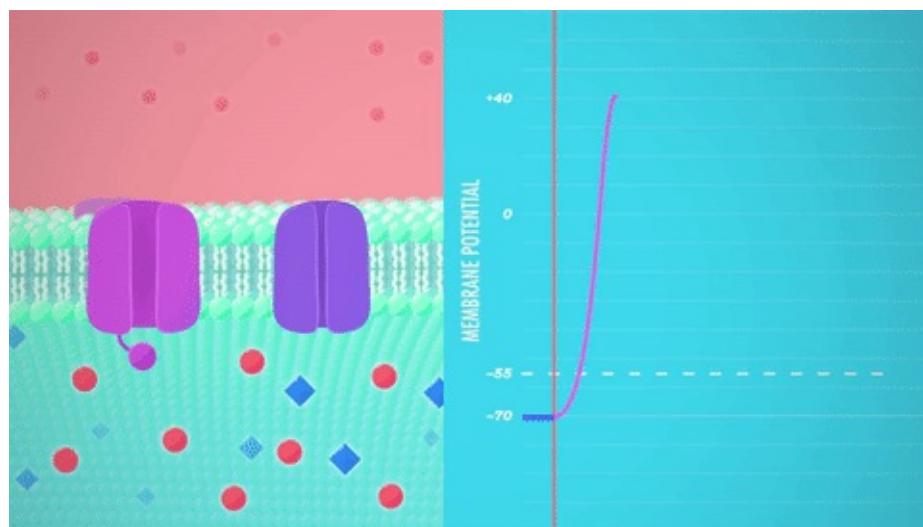
3. 充电：当刺激物（如神经递质）进入神经元时，会触发离子通道打开，允许钠离子（ $\text{Na}^+$ ）进入细胞膜，导致膜电位增高。



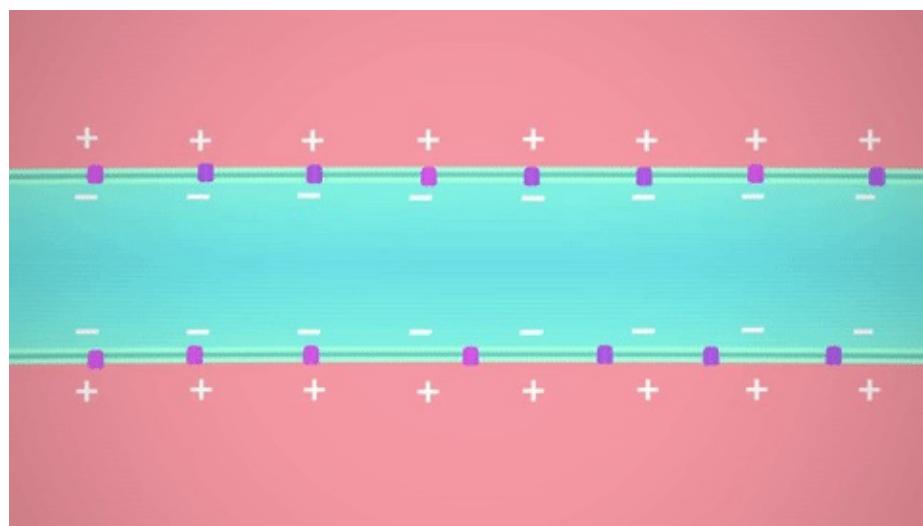
当膜电位累积到-55mV时（阈值），上千的钠离子通道会打开，使钠离子（ $\text{Na}^+$ ）迅速涌入，膜电位由负变正。



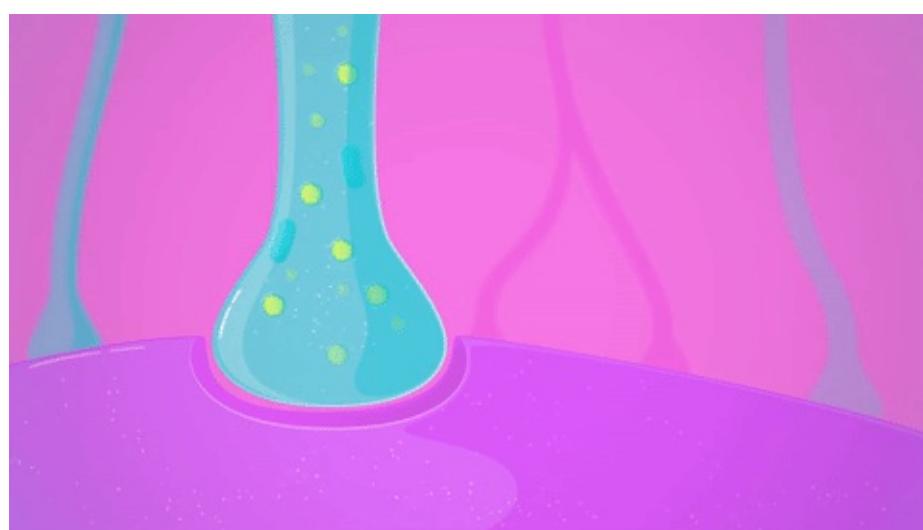
当膜电位到达大约+30 mV时，钠离子通道关闭，钾离子通道会打开，排除钾离子（ $\text{K}^+$ ），膜电位下降到差不多稍微低于-70 mV，然后钠钾泵会再次使神经元达到静态电位，从而迎接下一次动作电位（Action potential）。整个过程仅持续1-2毫秒。



4. 电信号传递：动作电位会单向传递，



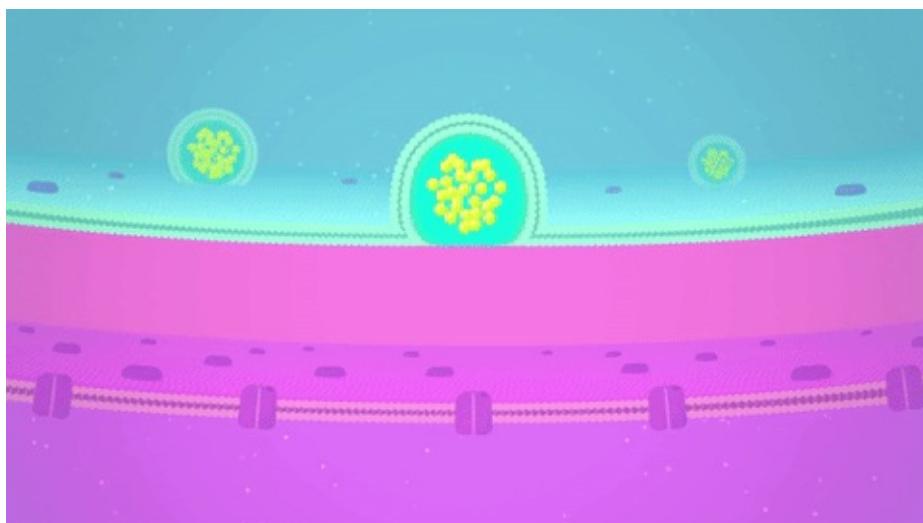
迅速到达与其他神经元连接的突触（synapse）处



1. 化学信号释放：动作电位会激活钙离子通道，钙离子进入膜内，



导致囊泡（vesicles）被释放到相连接的神经元细胞的突触间隙（synaptic cleft）。随后囊泡中的神经递质（Neurotransmitter）会与突触后细胞（postsynaptic cell）上的受体结合，造成兴奋或抑制。上文翻译Crash Course，参阅[更多内容](#)。



可以看到整个过程都是由大量蛋白质的并行识别完成的。

可把每个神经元作为一个新的单位（好比一个蛋白质的功能）

大量这样的单元一起工作可以实现什么功能？

# 神经元本质行为

一、当大量神经元在一起并行工作时，能够实现什么功能？

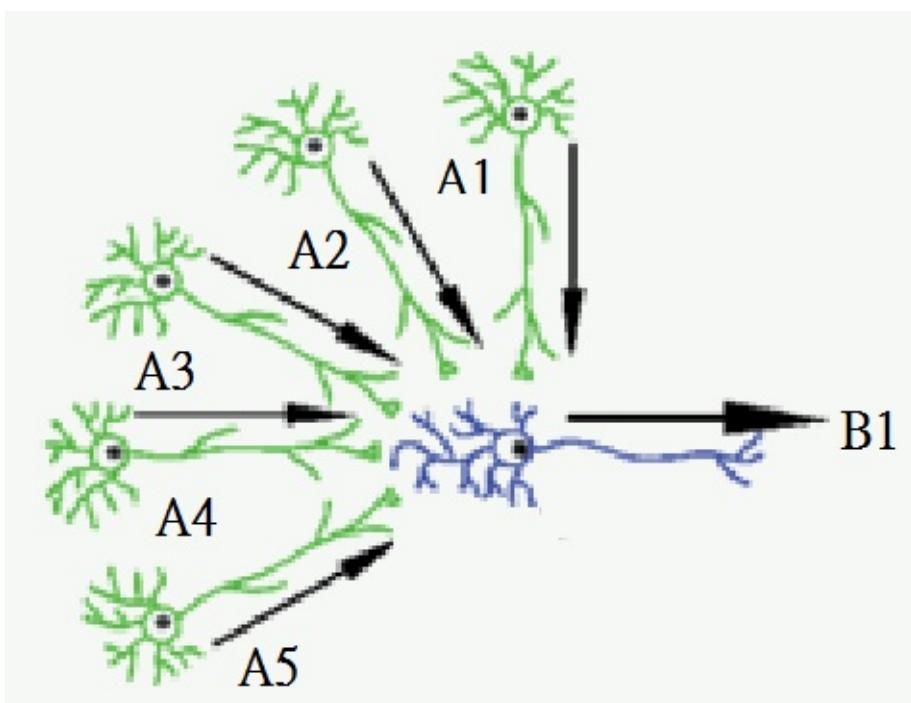
能够实现智能LV2：可学习和预测过去到未来的关联。

面对一堆神经元，难以理解它们的行为，这时就需要线性代数来帮助理解眼前的现象。

神经元的基本行为就是： $\vec{y} = a(W \cdot \vec{x} + \vec{b})$ ，

其中 $\vec{x}$ 是输入信号（向量）， $\vec{y}$ 是输出信号， $\vec{b}$ 是阀值， $W$ 是神经元各个链接的强弱， $a()$ 是化学传递。

以下图为例讲解：



输入：把左边5个神经元（绿色）作为输入，每个神经元代表一个因素，以此就形成了维度为

$$\vec{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix}$$

5的向量 $\vec{x}$ 。这五个因素形成的向量可以表示一个事物的状态。

权重：神经元之间所形成的连接，决定了 $W$ 的形状。 $W$ 装载着各个链接之间的强弱。表示着从 $\vec{x}$ 传递过来的电信号会以何种方式进行线性变化。

$$W = [A1 \ A2 \ A3 \ A4 \ A5]$$

阀值： $\vec{x}$ 经由  $W$  的变化传递到下一个神经元（蓝色）时，会逐渐累积，但并不会立刻触发，而是当膜电位达到  $-55 \text{ mV}$  时才会向下传递。在神经元细胞中的  $\vec{b}$  就可以是（具体数值可变）：  
 $\vec{b} = [-55 \text{ mV}]$

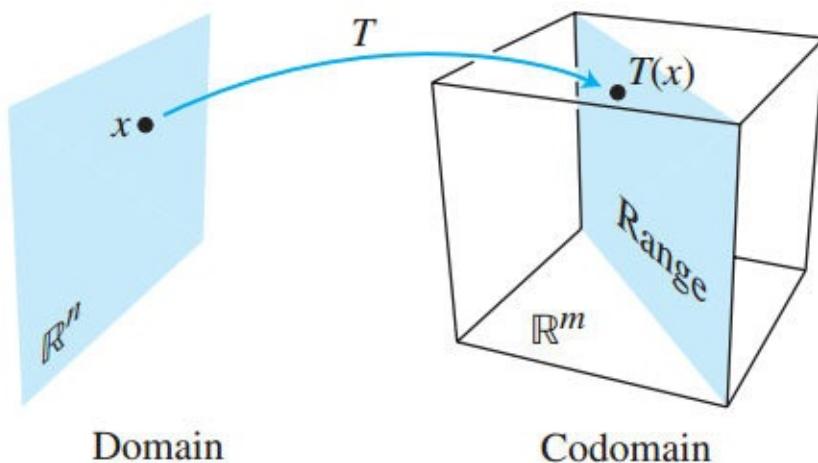
非线性：传递的电信号并不会直接被下一个神经元接受，而是通过神经递质进行增强（正信号）或抑制（负信号）。使线性变化的  $W \cdot \vec{x} + \vec{b}$  增加了非线性能力。生物中有多种  $a()$  表示不同的非线性能力。

输出：经由  $\vec{y} = a(W \cdot \vec{x} + \vec{b})$  得到的  $\vec{y}$  就可以看做 1 维向量，表示变化后的状态。这样一个基本动作就完成了。然而  $\vec{y}$  又可以看做为新的输入向量，通过其权重  $[B1]$  和其他的神经元共同作用，形成下一次的状态输出。

很多人认为深层学习跟神经学并不相关。然而事实上，他们本质是一模一样的。不过细节不一样。比如神经元并非靠电量来表示  $\vec{x}$  的值，而靠频率。进一步大脑会形成错综复杂的结构，并在神经元的基础上形成更复杂的功能。

但基本功能都是靠电信号+化学信号形成  $\vec{y} = a(W \cdot \vec{x} + \vec{b})$ ，以此完成一个状态到另一个状态的转变。通过大量的逐层迭代  $\vec{y} = a(W \cdot \vec{x} + \vec{b})$ ，并调控所有的  $W$ ，生物可以完美的学习我们这个星球上过去状态与未来状态的关联，通过预测来躲避危险。

大量神经元可学习过去到未来的关联  $T()$



**FIGURE 2** Domain, codomain, and range of  $T : \mathbb{R}^n \rightarrow \mathbb{R}^m$ .

## 二、神经网络借鉴了神经元？

深层学习是借鉴了神经元的信息交互方式后，在计算机上的一种实现方式。

- 类比：在  $\vec{y} = a(W \cdot \vec{x} + \vec{b})$  中：

神经元之间的电信号就是线性变换  $W \cdot \vec{x}$ 。

$\vec{x}$  的维度是输入层被连接的神经元的个数。

$\vec{y}$  的维度是输出层被连接的神经元的个数。

$W$  是各个从输入层神经元到输出层神经元之间连接的强弱（频率）。

$\vec{b}$  就是阈值。在生物体内应该是防止持续发送信号，在计算机中应该是防止  $\vec{x}$  为 0 后， $W \cdot \vec{x}$  还是 0 的情况。

而激活函数  $a()$  对应的是化学信号，即神经递质的信息传递方式。电信号改成化学信号后，就加入了非线性能力。生物体不一直用电信号的原因是无数年的自然选择早已进化出了最合适的方式。我们所居住的自然界有大量的非线性现象。光靠  $W \cdot \vec{x}$  是无法完成该星球的关联的学习的。

神经元和深层学习的本质类似，但细节不同，实现方法也不同。

- 实现方式比较：

- 计算机：基本 01 运算的顺序执行
- 生物：并行预测
- 例子：你会发现人类根本不擅长计算。原因就是在于二者的工作原理不同。即便是乘法，人脑在“计算”时是通过关联的方式，而计算机是实实在在的去计算。有时人们利用这一点，通过直接关联，从而看起来“计算”得比计算机快。

- 组合方式比较：

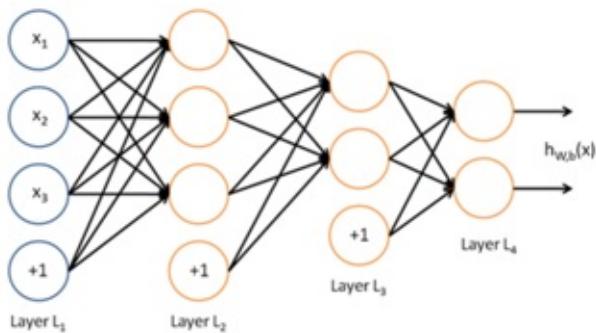
- 计算机：尝试过一次性建模。人们尝试用统计等数学原理直接建立两类事物的关联。
- 生物：逐层迭代。在生物体中，DNA 通过合成不同蛋白质来形成不同的预测模型，这些蛋白质通过并行预测又构成了一个细胞的预测功能，多个细胞并行预测又形成了更为复杂的预测功能，逐渐形成如视觉识别这样的高级感知能力。这种并行、且逐层迭代的方式使得生物体并没有在预测上遇到人工智能所遇到的那些困难。拿蛋白质而言，每个蛋白质只完成二类识别。仅仅考虑自己的任务。每个蛋白质可能识别错。但是并不会对全局的识别结果造成太大影响。这种方式完全符合自然进化规律。自然界无法突然产生出十分复杂的结构。往往都是同一原理在不同层级反复利用。
- 例子：仔细思考深层学习你就会发现。我们完全是在利用生物智能的特点去在计算机上实现智能原理，并行预测能力是由 GPU 提供的，而逐层迭代正是深层学习中多层表达的思想。

至此我们有了两套里程碑式的智能实现方式。神经元允许我们不需要通过筛选也可以实现智能，学习和再应用。

# 网络

DNA与蛋白质可以实现智能LV1，而大量蛋白质并行识别的基础上所形成的神经元可以完成智能LV2，人们模拟神经元在电脑上也同样实现了智能LV2。能够实现智能LV2的只有这些吗？

## 智能LV2的本质

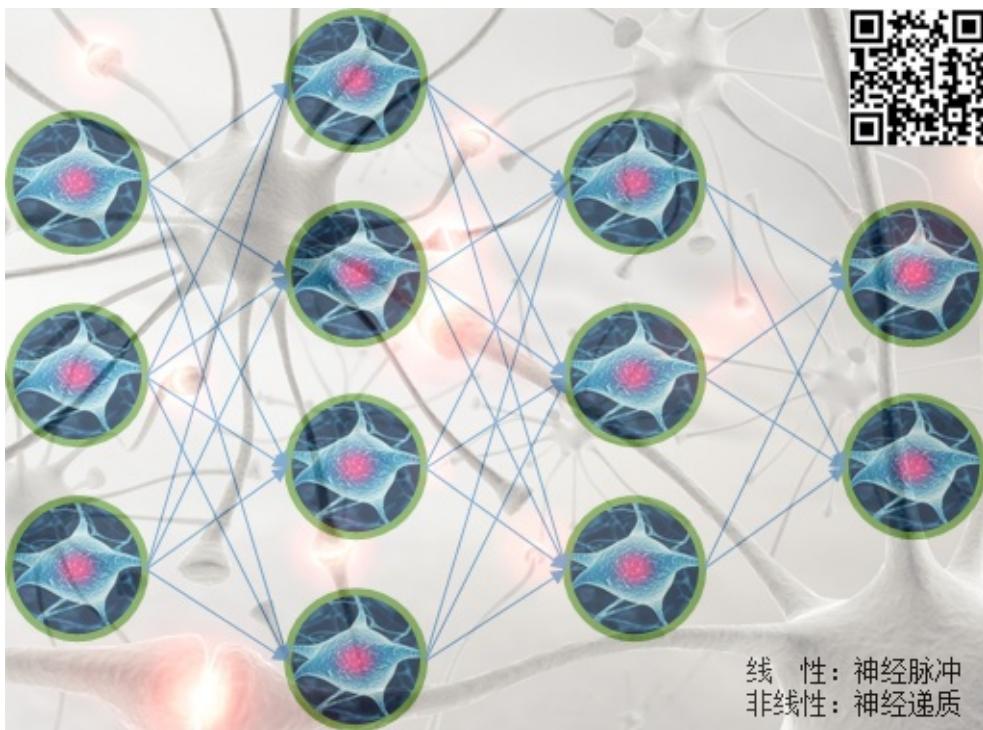


不管是神经元还是深层学习，都是一种网络结构。

除了神经元与深层学习，我们还可以在自然界中找到无数种这样的网络结构。任何一个层级下观察，都是一个智能体。蛋白质是，细胞是，人类是，乃至整个地球也是。而同类之间的再次组合，又会形成下一层的高级智能。

## 神经元

链接：蛋白质控制。



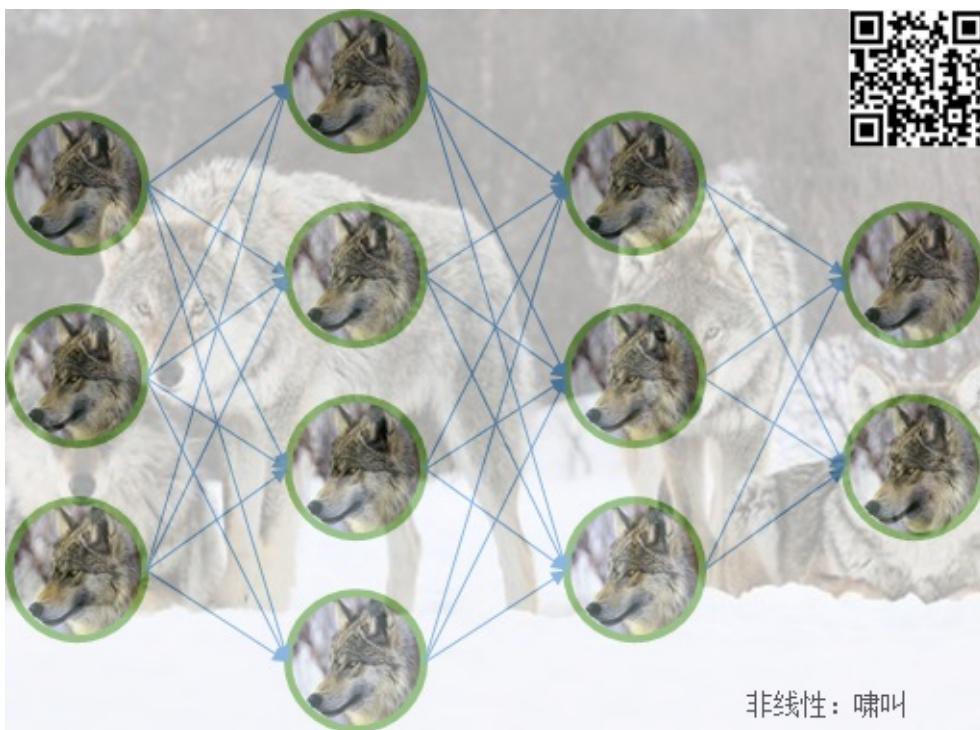
## 昆虫群

链接：空气中的信息素。



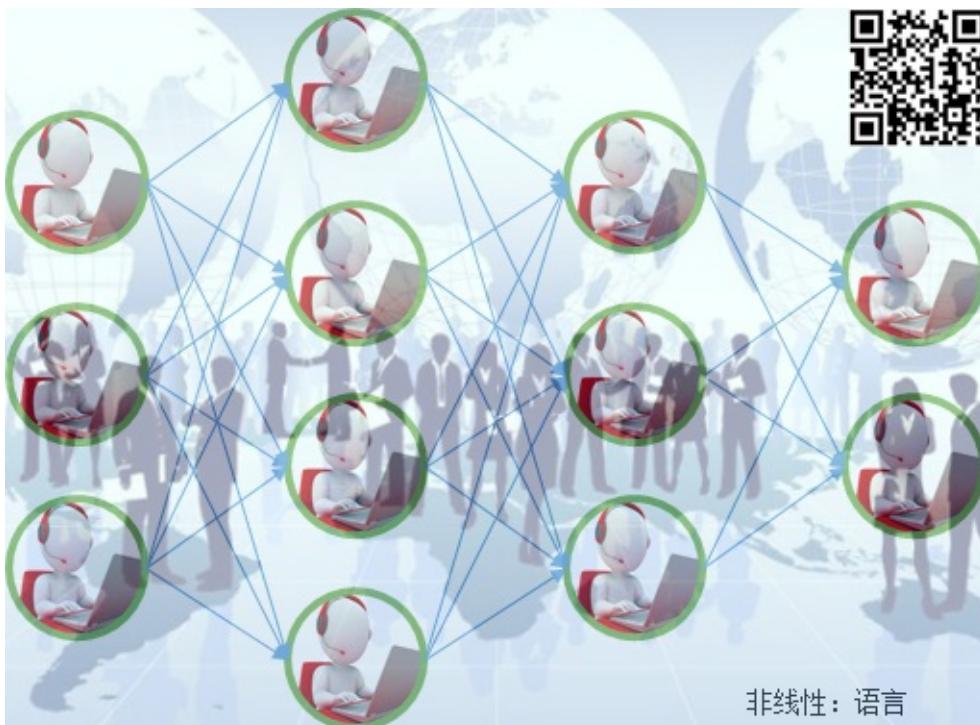
## 动物群

链接：气味，啸叫等信息传递方式。



## 人类社会

链接：语言，文字，互联网等。



影响智能功能大小的就有三个因素。深度，广度，速度。某些生物虽然数量多，但深度远不及人类。比如每个人只负责自己的工作，许多人形成一个工厂，而许多工厂共同生产一种零件或负责专业技术指导，层层相扣，最终完成航天客运系统。

大量模块形成更高级功能的前提：链接需稳定。不可以今天有，明天无。

其次与深层学习不同的地方是，这些网络虽然链接在一起，但是并非只有一个目的（损失函数）。比如人类就有自身的目的，每个工厂也有自己的目的。

这些表示如何变化状态的链接都可以存储在矩阵中。读到这里请再次体会一下矩阵是什么。

# 智能的不同阶段

如开篇所说，智能并非单一状态，而是分阶段、不断发展的，不同阶段的智能表现出的能力不同。下面就谈谈智能的不同阶段，及划分标准。

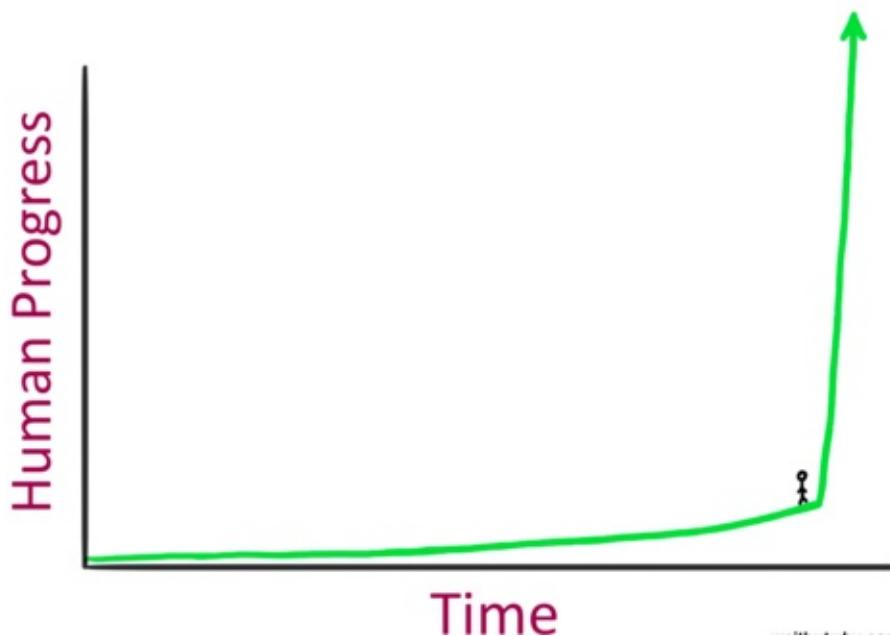
## 存储媒介划分：

智能的发展从某种意义上也可以看作“存储设备的延伸”。生命史上第一个成功的关联存储媒介是DNA。学习来的模型是可以依靠DNA，跨越生命个体来延续的。生物至今依然受存储在DNA上的预测模型影响。这也便是我们常说的“天性”。有些模型是在古代为了更好生存而学习得到的，虽然很多模型在当今社会不再适，却难以更改。动物的“兽性”，和人类的“自私，懒惰，恐惧，嫉妒”等存在于所谓的古脑中的本能情绪，都是来源于存储在DNA上的预测模型。在DNA之后的储存媒介是短期记忆，随后又有了长期记忆。甚至纸张，硬盘也都是模型存储设备的延伸。

DNA-->瞬时记忆-->短期记忆-->长期记忆-->竹简、石头-->纸张-->计算机硬盘

但仅靠自然选择的方式去更新关联就不得不建立在个体的不断淘汰的基础上，并且关联受限于环境。而在DNA构架的基础上产生的神经元实现了任意两个状态空间关联的学习。相比DNA而言，学习速率有了空前的提升，更为关键的是使得智能更新不需要通过自然选择，而通过个体的学习就可以实现。同时纸张，硬盘等智能储存媒介的产生使得人类从众生物中脱颖而出，成为万物之灵。其原因就在于人类率先找到石碑文，竹简，纸张等跨个体的传播保留通过大脑皮层所学到的关联，不需要个体在出生后都重新学习。互联网的诞生，又极大的

加速了关联的传播速度。让人类的GDP生产曲线变成了下图的形状。需要提到的是，尽管新大脑皮层的学习速度非常的快。但其对于生物个体的保障性和可靠性远不如DNA中学习了数



亿年的智能。

## 生物类别划分：

这里我就以生物类别为视角介绍智能从最初阶段一直到人类的智能阶段，并进一步扩展。

起源--->微生物--->植物--->昆虫--->昆虫群体--->动物--->动物群体--->>人类

智能的发展基本上取决于两个因素：预测模型的数量（宽度与深度）和预测模型之间的传输速度。

- 智能起源：就是单一的预测模型。至此不得不提到一个概念：群体智能。诸如蚁群，蜂群，大雁，狼群等。从生物角度上来说，不应该把群体智能特别看待，因为即便是单个细胞的智能，也是由大量蛋白质预测模型并行预测形成的。从这个角度来说所有生物的智能都是群体智能。这也解释了为什么决定智能划分标准的两个因素是预测模型的数量和模型间信息传输速度。但这两个因素共同张成了一张大网：描述世界的能力。
- 植物的智能：模型数量相比动物较少。功能性总的来看，两个大模型最为突出：1、根据光源改变面向的能力，2、根据水源改变根的生长方向的能力。依靠信息素传递的植物，传输速度成了智能的瓶颈。植物的描述世界程度基本只有根部、光源、水源。
- 昆虫的智能：模型数量少，但大模型的功能增加了诸如视觉，触觉等感知预测模型，可以小范围移动。模型间传输手段是电信号+化学信号，优于植物。能描述世界的程度高于植物，但相对来说依旧很小。

- 群体昆虫：预测模型是由个体昆虫组成的复杂模型（每个个体昆虫也可以简单看做为一个预测模型），世界描述程度高于个体昆虫，但是由于昆虫间靠信息素交流，传输速度又低于个体昆虫。相比个体昆虫拥有更强的寻找食物能力和抵挡外来入侵能力。
- 动物及动物群体：大体类似于昆虫和昆虫群体，但是由于活动空间的增大，所能描述世界的能力高于昆虫。并且某些动物已经可以利用叫声传递信息，声音的传播速度又快于信息素。
- 人类个体及人类社会：在各项指标方面，基本所有人类个体高于动物个体的原因都归因于人类社会，所以难以分开论述。人类个体内部预测模型的传输速度都是电信号+化学信号，并不比其他动物优异。但预测模型的数量却是已知生物中最多的(绝大多数存在于大脑)。新大陆的发现、各类交通工具的更新使得活动范围空前广大的人类，脑中的预测模型可以描述整个地球、甚至眼睛无法感知的外太空。再加上由精神思维活动产生的领域，使得人类可以描述世界的程度已经大到难以度量。另外纸张和硬盘等模型存储媒介的应用，允许不同个体之间交换已学的模型知识。
- 互联网：已知最强智能人类社会已经是很多人类连接起来的群体智能了。而互联网的发明更是将能够连接起来的人类的数量和信息传输速度提升到了极致。各项指标都是最高。是当之无愧的最强智能。不过由于战争种族的纠纷和贫穷问题，还有很大提升能力。
- 地球连接：跨越种族的群体智能。如人类和警犬之间的共同协作。

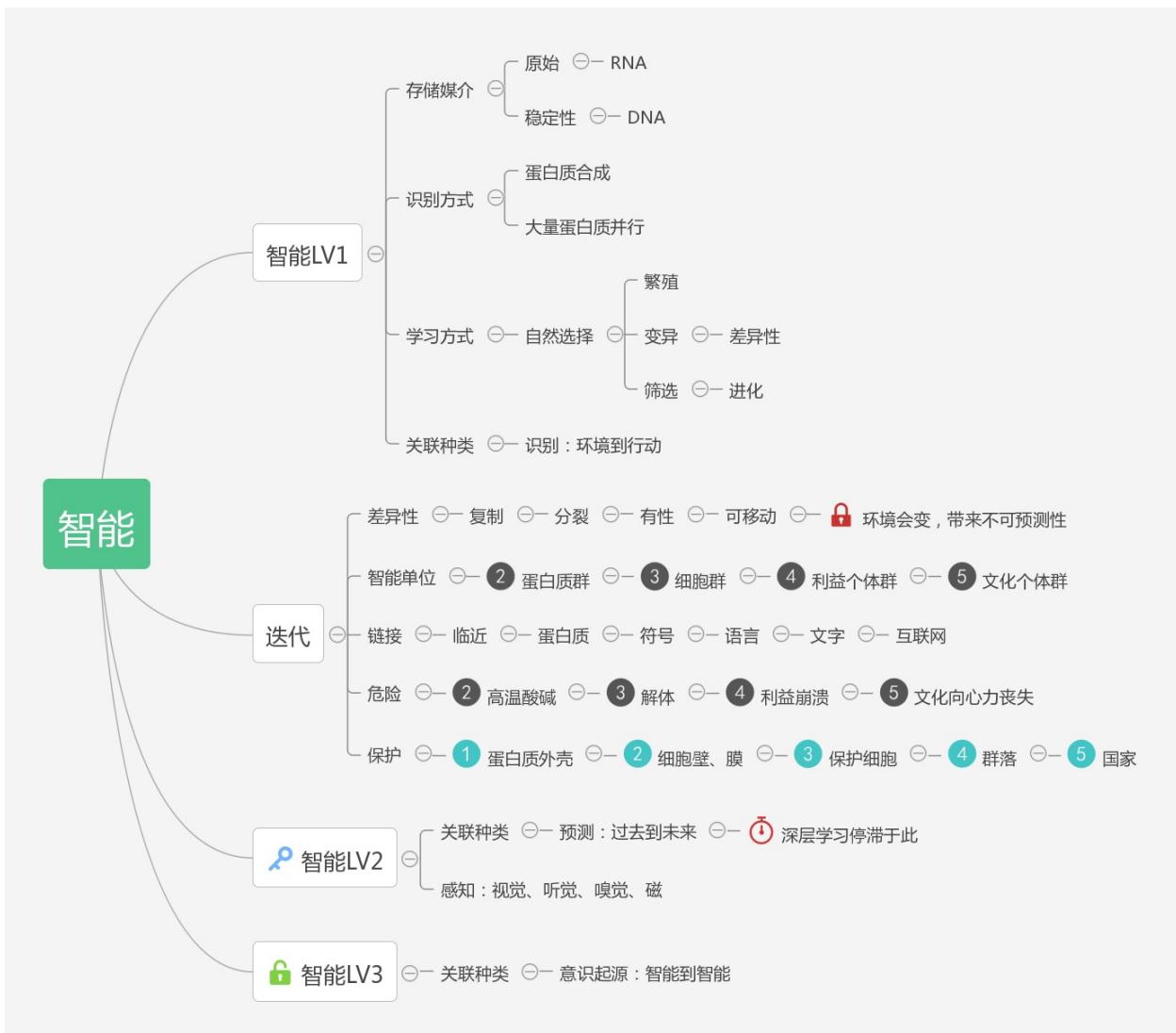
种群	连接模型数量	连接速度	世界描述力
智能起源	F级	F级	F级
植物	E级	E级	E级
昆虫	D级	D级	D级
群体昆虫	A级	E级	C级
动物	C级	D级	C级
动物群体	E级	C级	B级
人类社会	A级	B级	S级
互联网	A级	S级	S+级

# 智能结构梳理

- 研究方法：从源头推演
- 平衡：
  - 环境：自发性增熵
    - 例：今天转到明天，信息量（熵）增加，状态变化未知。
  - 生命：自发性减熵
    - 例：智能预测明天，信息量（熵）减少，预测状态变化。
- 研究工具：
  - 线性代数：任意维度空间下事物状态和状态变化的规则。
  - 概率：衡量对事物在跨时间后不同状态的确信度。
  - 熵：衡量对事物在跨时间后能产生不同状态的混乱度。
  - 其他：数学，化学，物理，生物，社会。能用的都用



- 智能：可以根据环境（生物周围的局部环境）变化而做出相应变化的能力（熵减的能力）



- 智能**LV1**：可再次利用从环境到行动的关联的能力。

- 储存媒介：产生RNA后，单链结构并不稳定，于是形成了DNA专门负责存储关联，而蛋白质专门用于实现关联。
- 识别方式：大量蛋白质并行进行简单识别形成高级功能。遗传物质需要保护，而病毒就是蛋白质外壳包裹遗传物质。蛋白质又怕高温高酸碱环境，逐渐又形成了单个细胞分隔外界环境来保护整个智能系统的运作。
- 学习方式：由于减少的是自身周围环境中的熵，所以当环境变化时关联也随之变化。自然选择就是动态学习环境关联的过程。自然界的繁殖建立在大量“浪费”的基础上并非真的浪费，而是为了产生差异性。无法合理完成减熵的智能关联会被筛选掉，造成的结果就是进化。进化是被动的，方向不由生命控制，而由环境决定。但有一个方向固定，那就是增加差异性。

- 迭代：所有功能上一级的基础上通过大量并行，多层迭代的方式增强。

- 差异性：当自然选择对智能关联的更新难以到达环境更新速度时，熵就无法有效减少，物种面临灭绝危机。因此能被留下的（进化）都是趋向于产生更多差异性的方

从最终的病毒复制，到细胞分裂等方式，再到有性繁殖，差异性不断增加。但当有性繁殖出现后，如何“洗牌”（如下图）就产生了问题。如果生物不可移动，那么就只能跟临近的个体交配，跟无性繁殖所能产生的差异性并不会差太多。因此变异成有可移动倾向的个体就被逐渐筛选下来。可移动生物的优势在于其差异性。



有趣的是植物无意中进化到了另一种方向，由于植物可以从太阳那里获得大量能量，动物需要能量，植物所产生的拥有能量（生物能）的果实的种子会被动物带到各个地方，从而减免了植物在某个环境下被淘汰的几率。然而当动物可以移动时，就带来了另一个严重问题，为智能LV2和智能LV3留下了伏笔。

- 智能单位：请不要把生命仅仅局限于个体这个单位之上。并非所有的生命都拥有意识，但所有的生命都拥有智能，它们通过大量并行和多层迭代的方式增强功能，与环境达到平衡，是自然界的一部分。病毒、细胞、组织、器官、个体、群体、国家、地球，不论从哪个层级上观察，所观察到的事物在全局上都是一个“智能体”。这也是这本书名的后三个字。而“超”字意味着，我希望作为智能中一环的人类，能跳出自身的层级，用超出人类自身的意识，感官和情感的方式去理解生命。
- 链接：不同的层级间有不同的链接方式，而链接的数量、速度也决定了智能的强弱。比如虽然同为人类，但再拥有互联网后的人类所能形成的智能和过去古人就无法相提并论。
- 危险：从局部上看，智能体也面临着危险。

比如辐射对DNA序列的打乱。

高温高酸碱环境对蛋白质的影响。

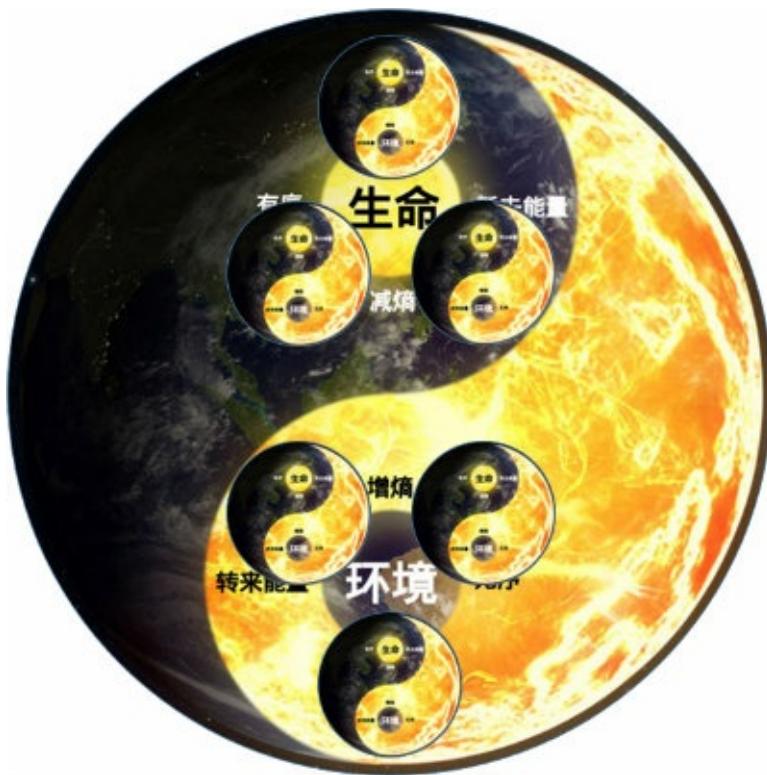
又比如某些毒蛇毒液攻击肌肉细胞组织，使其不再一同工作。

形成利益共同体的人类可以完成工厂等级的智能能力，担当利益解体时。该智能体便散回到人类个体的层级上去。

西方人很难理解中国的崛起。在他们眼中，优秀的国家应该可以让人民享有更多的自由和权利。但这种想法是片面的。因为凝聚国家的是宗教、文化等认同感。国家是人民共同协作产生的高级智能体。很多国家解体并不是因为国家对个体的政策不

够好，而是失去了让个体共同协作的向心力。自古的中国就在“平衡”二字上做的非常优秀。相比崇尚“自由”的西方国家而言，中国作为一个智能体可能在细节上对人民并不友好，但是它给足了人们共同协作的向心力。并且西方国家所推崇的“自由”其实是最大的谎言。人们的自由首先受限于智能LV1的关联，即生理系统，人类会饥饿，会寒冷，会死亡。其次在思维上也受限于自身的教育环境。环境并不会给人们固定的思维，但会给人们固定的思维范围。是一个类似于向量空间的概念。思考所产生的组合就和线性组合一样，是局限于在某个空间下的。所谓的高维思考就是提高了思维空间的维度，从而加大范围。人们难以理解三维以上的几何空间是由于我们压根就没见过。所以“自由”本身就是一个相对的概念。如果给个体足够的自由，那么个体就会跳出该层级，不再共同协作。最生动的例子就是“癌症细胞”。癌症细胞的拥有永生的能力，但它带来的确是整个智能体（如人体）的崩溃。很不幸的是中国其实也在形成“思维分层”的趋势，但这是另一个话题，不在该篇讨论。

- 保护：有危险，智能体自然也会通过筛选，进化出相应的保护措施。比如病毒的蛋白质保护性外壳，细胞膜和整个细胞的封闭结构，保护性细胞群等。这些保护措施是保护智能体在该层级上的稳定。又如人类的感情（同情、怜悯、自豪等）都是易于种群的凝聚和延续的。而到了国家层级上时，我们依然用中国举例，中国的在某些方面的封闭性和强制性从某种意义上也保护了国家在崛起时整个智能系统的稳定。但请不要误解这种保护。保护的是整体的稳定性，并不意味着该智能体不与外界进行交流（这完全不同于闭关锁国时的中国）。所有智能体都有形成链接来达成更高智能体的倾向。但这种链接的成立是建立在上一层智能体结构稳定的基础之上的。两个解体的神经元无法形成任何高级功能。同样两个混乱的国家也无法进行合理的贸易。地球现在就局限在这个阶段。如果各个国家共同协作，整个地球就是一个更高级的智能体，每个国家相当于人体内的不同器官。然而问题在于很多国家作为一个智能整体，其结构并不稳定，战争的动乱，社会的解体等。这些都阻碍了地球智能的形成。同时也可以发现，越高层次的智能体越不稳定，越需要大量的能量来维持。同样是熵和内部环境达成平衡。但是低级智能体的“循环圈”的半径小。而高级智能体的“循环圈”的半径大。形成的平衡图可以描述成下图这样。每一个圈都要达到平衡后才能形成更大的圈。



- 智能LV2：回到差异性的位置。当产生动物后就面临一个严峻的问题。那就是当生物移动后，环境也跟着变化了。于是就产生了下面的问题：

### 一、如何减少由移动带来的环境不确定性？

答案是预测，通过预测下一刻会发生什么，从而决定是否移动，何时移动。

### 二、需要满足哪些条件？

- 智能LV2：智能LV1只能完成从“环境空间”到“行为空间”的关联。而若要预测，则需要能够实现从“过去时刻”到“未来时刻”的关联。我们已经知道通过神经元的大量链接可以实现任意两个空间的关联。
- 感知：需要一种可以很好反映环境的关联，以此来做出判断。这时的关联是从“物体本身所反映出的客观属性”到“物体概念”的关联。这需要智能LV1的蛋白质作为底层接受器，直接和光，电，磁，声音震动等进行交互。然后利用神经元的智能LV2将其匹配到表示该事物的概念上。比如昆虫的触须，蛇的热感视觉，蝙蝠的超声定位，人的光感视觉。都是用于反映环境的关联。另外稍微联想一下就知道为什么比起文字人们更喜欢视频。因为视觉听觉才是人们最初用来建立关联的记忆。而文字只是交流手段，交流的前提是彼此都有该记忆。
- 记忆：不同于存储在DNA上的记忆。这里的记忆是指从感知器官接受到的环境的状态。只有记住过去发生的环境状态才能实现“过去时刻”到“未来时刻”的关联。

目前的深层学习就到达了智能LV2。我们可以轻松解决过去到未来的关联。语音识别，画面识别不再是难题。虽然在用计算机实现智能LV2时仍有研究可做。但放眼人工智能，不可过于局限在智能LV2，而需要下一阶段迈进。在深层学习基础上所建立的增强学习便是正确的方向。

智能LV2只是带来的第一把钥匙。并没有完全解决动物移动后带来的全部问题：

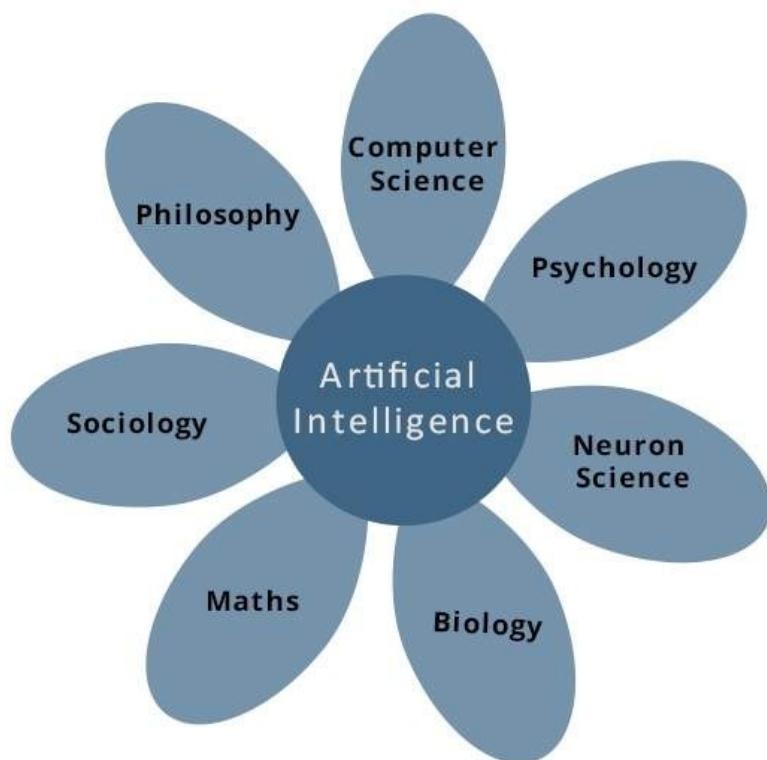
- 向下传递的必要：生物怎么知道什么时候该学习哪些关联？
- 智能模型的选择：四肢有各种各样的预测模型，某一时刻到底该用哪一个？（智能可以并行，唯独意识不可以同时出现在两个位置）
- 模拟情景的必要：这类生物负担不起胡乱尝试，它们需要在大脑中生成情景。

意识的起源也就渐渐浮出水面。

# 人工智能

## 一、什么是人工智能？

YJango对人工智能的理解先以一张图片作为引入。



图片出处

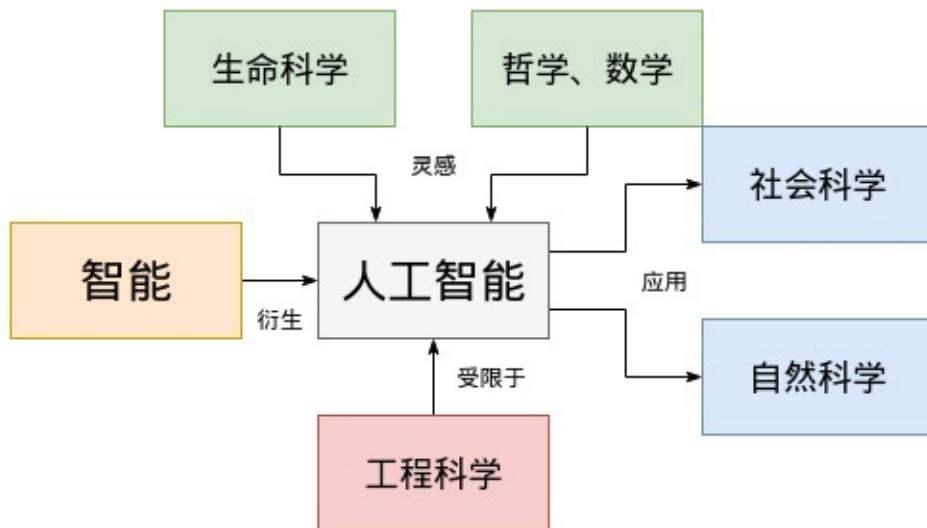
从上图可以看出来人工智能是多科学的交叉，但又不被任何一个学科完全包含。从智能的定义直接扩展的话，人工智能是非自然选择形成的一种减熵的能力。

### 人工智能是人造减熵的能力

由于人类对于宇宙而言还很年轻，对自然界的智能尚未全面掌握，所以我们对人工智能的发展是处在一边增强减熵能力，一边应用的方式中。

## 二、人工智能和其他科学的关系？

但总体来说，减熵这一特点使得人工智能本身就更偏向于实用性科学。



- 灵感：人工智能的能力增强研究主要是以生命科学如神经科学，生物学等发现作为灵感来源。以哲学、数学作为这些发现的解读和方向引航。
- 应用：将已有的人工智能技术应用到社会科学和自然科学之中（也可以返回到数学和生命科学中）。
- 受限：人工智能并非计算机科学的分支。工程科学是将人工智能理论研究实现的手段，并非人工智能的全部。计算机科学上实现的人工智能就好比带着脚镣的舞者，是因为当前的硬件条件使得人工智能的实现不得不去符合工程科学。除了传统计算机我们仍可发展其他计算机和智能的实现手段。比如量子计算机和人工改造基因技术。

# 机器学习

## 一、什么是机器学习？

问道什么是机器学习时，人们通常会给出一个非常抽象的描述。尽管严谨和具有更高的普遍性，但也成了“拦路虎”。容易让让新人觉得机器学习特别复杂。这里我举一个我认为最容易理解机器学习的例子。

- 例子：我们初中学过一元一次方程： $y = 2x + 1$ 。 $x$ 与 $y$ 之间是存在关系的，当把 $x$ 换成数值后就可以算出相应的 $y$ 。但我并不想通过 $x$ 来算出 $y$ ，而是想通过符合该关系的 $x$ ， $y$ 的具体数值来确定该关系。也就是解一元二次方程 $y = ax + b$ 。我们需要配对的 $(x, y)$ 来解该方程。并且需要的还不止一对。这里至少需要两对。但是如果我们知道 $y = ax + b$ 实际上是 $y = ax$ ，这时就只需要一对就可以确定 $x$ 与 $y$ 的关系。但如果我们连两者是否符合一元一次方程都不知道时，又要如何确定两者的关系，同时又需要多少对 $(x, y)$ ？

学习是关系的找回，而机器学习是靠机器来找回关系。

其中用于找回关系的所有 $(x, y)$ 叫做数据(data)。像这样外界给予 $x$ 所对应的 $y$ 的学习叫做监督学习（supervised learning）。

- 注：关于无监督学习（unsupervised learning），其实按字面意思理解就可以，没有人力来输入标签。很多人会认为无监督学习就是没有 $y$ 的学习。若没有 $y$ ，模型就只可以寻找输入空间的关系，比如聚类（cluster）。但聚类是无监督学习的一种。未来的人工智能的方向是无监督学习，是指模型已经足够强大，能够自己生成相应的 $y$ ，并非没有 $y$ 。

我们所知道 $x$ 与 $y$ 之间符合一元一次方程的这个信息叫做先验知识。先验知识越多，就可以用越少的数据来确定关系。

但机器学习所学习的关系更为复杂，没有一元一次方程这样的解题过程。在普遍的机器学习中，人们会选定一个目标函数（objective function），通过不断降低目标函数的值来寻找关系。该过程也叫做优化（optimization）。用于优化的算法便是优化算法（optimization algorithm）。

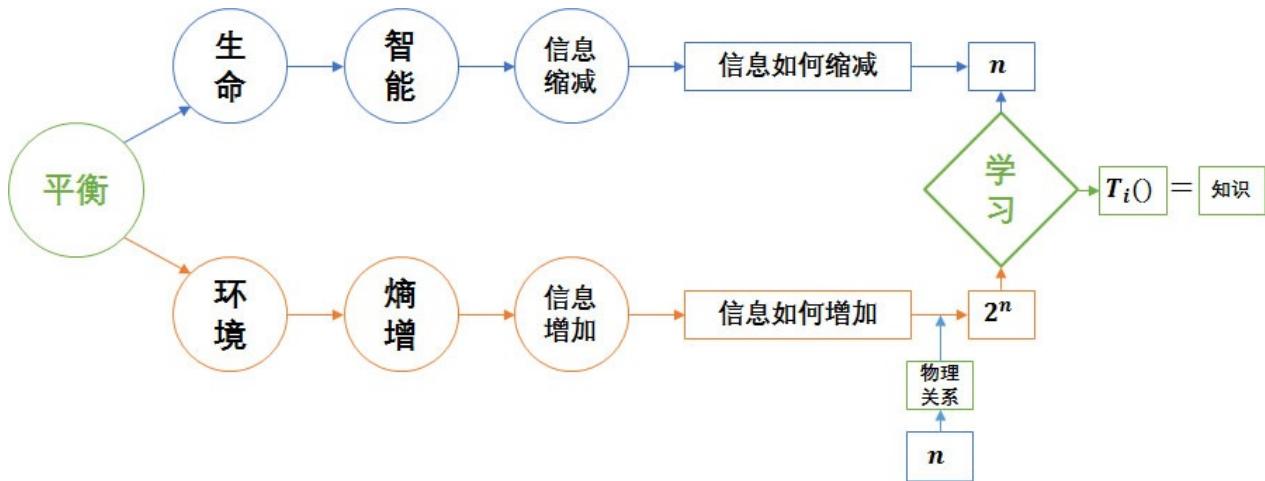
## 二、生物学习与机器学习的区别？

智能是减熵的能力。生物智能和人工智能、生物学习和机器学习是智能的两种实现方式。

生物智能和人工智能也都在不断的发展。由于目前的人工智能主要停留在智能LV2的阶段，所以这里比较的是智能LV2时彼此的区别。

自然：对于动物而言，最重要的是能够预测未来事件的能力。

- 智能LV2是可再次利用从过去状态到未来状态的关联的能力。
- 学习是寻求从过去状态到未来状态的关联的过程。



旧状态通过物理关系形成更多的新状态，使得不确定性增高。学习是从信息中找回物理关系从而回卷信息，降低不确定性的过程。被找回的物理关系叫做知识。但和第一个例子最后的问题一样，学习不是魔法，无法无中生有。需要先验知识。庆幸的是我们生活的世界，过去状态到未来状态的发展就有两个固有的先验知识：

1. 并行：新状态是由若干旧状态组合形成。
2. 迭代：新状态可由已形成的状态再次形成。

生物体里的神经元便时时刻刻在应用这两个固有的先验知识。

人造：同样是找回造成新信息的知识。区别在于，这里的知识就并非主要是物理关系。可以是由人们随意建立的关系。

没有免费午餐(No free lunch theorem)：这是在机器学习中被证明的定理。表述为“any two optimization algorithms are equivalent when their performance is averaged across all possible problems”。也就是说，对某个问题高效的优化算法一定在另个问题上存在缺陷，不可能对于所有问题都有效。这里的“all possible problems”指的就是任意关系。深层学习也同样满足该定理。

深层学习不是万能算法。它无法对任意关系都有效。但是你看到的是深层学习推动了整个人工智能的发展。那是因为我们现实世界需要解决的问题并非任意关系，而是物理关系。而深层学习的固有先验知识就是生物所用的并行与迭代。只有满足这两个物理规律的问题深层学习才会发挥作用。

# 为什么神经网络能够识别

为了研究神经网络，我们必须要对什么网络是什么有一个更直观的认识。

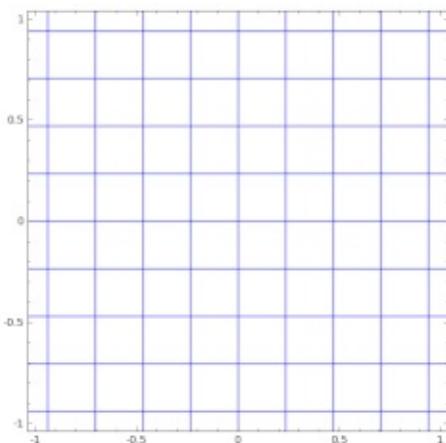
## 基本变换：层

一、神经网络是由一层一层构建的，那么每层究竟在做什么？

- 数学式子： $\vec{y} = a(W \cdot \vec{x} + \vec{b})$ ，其中 $\vec{x}$ 是输入向量， $\vec{y}$ 是输出向量， $\vec{b}$ 是偏移向量， $W$ 是权重矩阵， $a()$ 是激活函数。每一层仅仅是把输入 $\vec{x}$ 经过如此简单的操作得到 $\vec{y}$ 。
- 数学理解：通过如下5种对输入空间（输入向量的集合）的操作，完成  
 的变换(矩阵的行空间到列空间)。

注：用“空间”二字是指被分类的并不是单个事物，而是一类事物。空间是指这类事物所有个体的集合。

- 1. 升维/降维
- 2. 放大/缩小
- 3. 旋转
- 4. 平移
- 5. “弯曲”

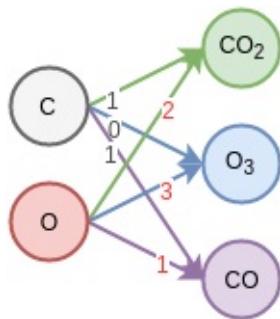


这5种操作中，1,2,3的操作由 $W \cdot \vec{x}$ 完成，4的操作是由 $+\vec{b}$ 完成，5的操作则是由 $a()$ 来实现。

每层的数学理解：用线性变换跟随着非线性变化，将输入空间投向另一个空间。

- 物理理解：对 $W \cdot \vec{x}$ 的理解就是通过组合形成新物质。 $a()$ 又符合了我们所处的世界都是非线性的特点。

- 假想情景： $\vec{x}$ 是二维向量，维度是碳原子和氧原子的数量  $[C; O]$ ，数值且定为  $[1; 1]$ 。若确定  $\vec{y}$  是三维向量，就会形成如下网络的形状（神经网络的每个节点表示一个维度）。通过改变权重的值，可以获得若干个不同物质。右侧的节点数决定了想要获得多少种不同的新物质（矩阵的行数）。



- 若权重  $W$  的数值如(1)，那么网络的输出  $\vec{y}$  就会是三个新物质，[二氧化碳，臭氧，一氧化碳]。

$$\begin{bmatrix} CO_2 \\ O_3 \\ CO \end{bmatrix} = \begin{bmatrix} 1 & 2 \\ 0 & 3 \\ 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} C \\ O \end{bmatrix} \quad (1)$$

- 若减少右侧的一个节点，并改变权重  $W$  至(2)，那输出  $\vec{y}$  就会是两个新物质， $[O_{0.3}; CO_{1.5}]$ 。

$$\begin{bmatrix} O_{0.3} \\ CO_{1.5} \end{bmatrix} = \begin{bmatrix} 0 & 0.3 \\ 1 & 1.5 \end{bmatrix} \cdot \begin{bmatrix} C \\ O \end{bmatrix} \quad (2)$$

- 若再加一层，就是再次通过组合  $[CO_2; O_3; CO]$  这三种基础物质，形成若干个更高层的物质。
- 若希望通过层网络能够从  $[C, O]$  空间转变到  $[CO_2; O_3; CO]$  空间的话，那么网络的学习过程就是将  $W$  的数值变成尽可能接近(1)的过程。
- 重要的是这种组合思想，组合成的东西在神经网络中并不需要有物理意义，可以是抽象概念。

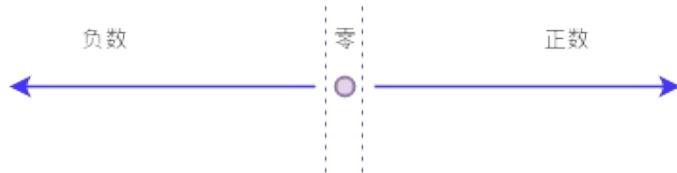
每层神经网络的物理理解：通过现有的不同物质的组合形成新物质。

## 理解视角：

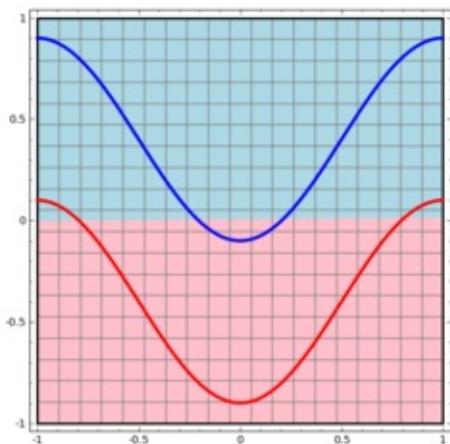
二、现在我们知道了每一层的行为，但这种行为又是如何完成识别任务的呢？

数学视角：“线性可分”

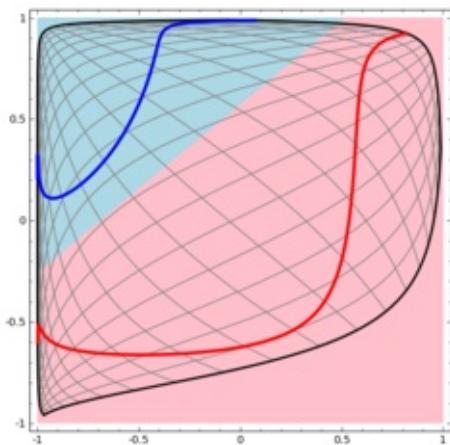
- 一维情景：以分类为例，当要分类正数、负数、零，三类的时候，一维空间的直线可以找到两个超平面（比当前空间低一维的子空间。当前空间是直线的话，超平面就是点）分割这三类。但面对像分类奇数和偶数无法找到可以区分它们的点的时候，我们借助  $x \% 2$ （除2取余）的转变，把  $x$  变换到另一个空间下来比较0和非0，从而分割奇偶数。



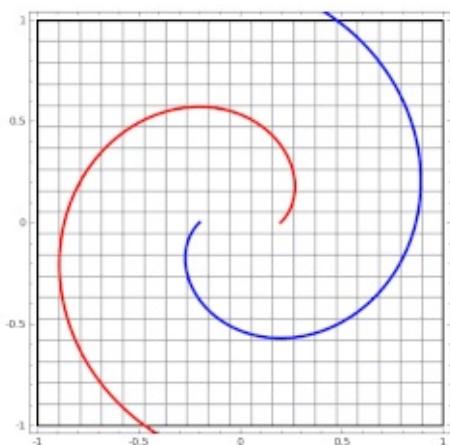
- 二维情景：平面的四个象限也是线性可分。但下图的红蓝两条线就无法找到一起超平面去分割。



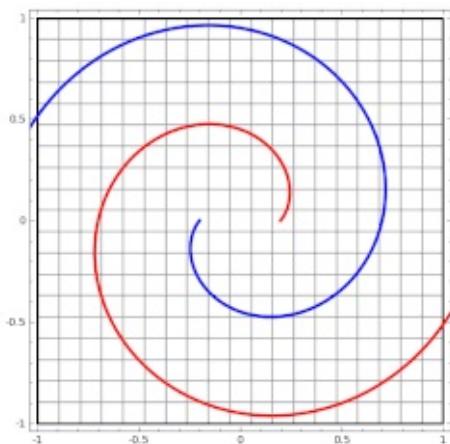
神经网络的解决方法依旧是转换到另外一个空间下，用的是所说的**5**种空间变换操作。比如下图就是经过放大、平移、旋转、扭曲原二维空间后，在三维空间下就可以成功找到一个超平面分割红蓝两线（同SVM的思路一样）。



上面是一层神经网络可以做到的空间变化。若把  $\vec{y}$  当做新的输入再次用这5种操作进行第二遍空间变换的话，网络也就变为了二层。最终输出是  $\vec{y} = a_2(W_2 \cdot (a_1(W_1 \cdot \vec{x} + b_1)) + b_2)$ 。设想当网络拥有很多层时，对原始输入空间的“扭曲力”会大幅增加，如下图，最终我们可以轻松找到一个超平面分割空间。



当然也有如下图失败的时候，关键在于“如何扭曲空间”。所谓监督学习就是给予神经网络大量的训练例子，让网络从训练例子中学会如何变换空间。每一层的权重  $W$  就控制着如何变换空间，我们最终需要的也就是训练好的神经网络的所有层的权重矩阵。。这里有非常棒的可视化空间变换demo，一定要打开尝试并感受这种扭曲过程。更多内容请看 [Neural Networks, Manifolds, and Topology](#)。



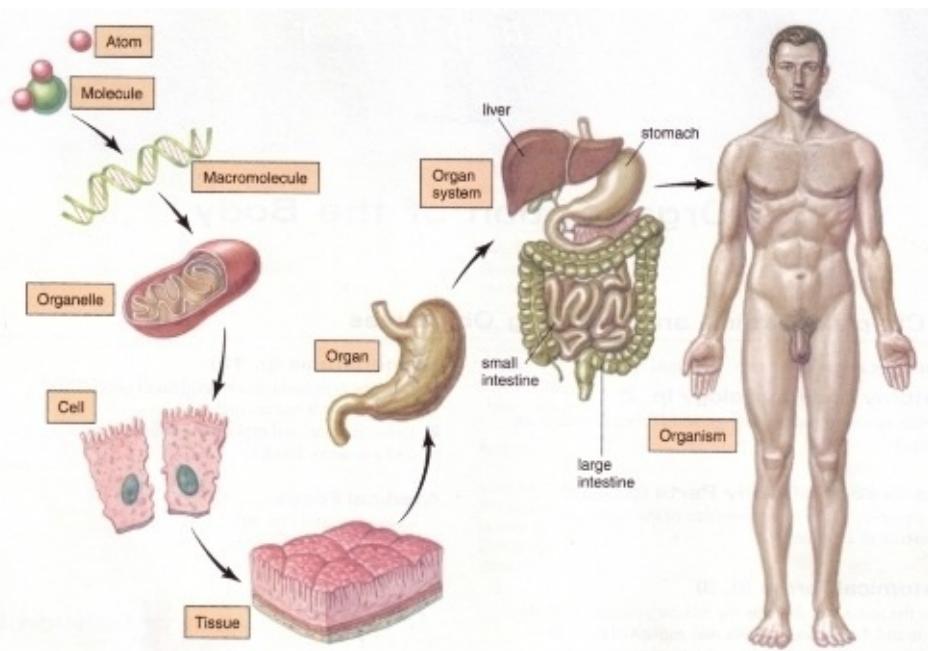
**线性可分视角：**神经网络的学习就是学习如何利用矩阵的线性变换加激活函数的非线性变换，将原始输入空间投向线性可分/稀疏的空间去分类/回归。

**增加节点数：**增加维度，即增加线性转换能力。

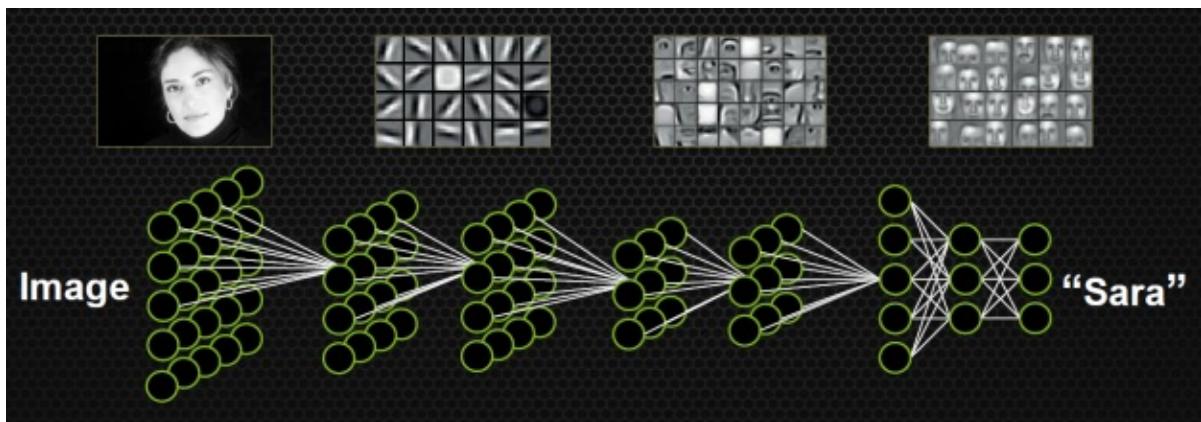
**增加层数：**增加激活函数的次数，即增加非线性转换次数。

## 物理视角：“物质组成”

- **类比：**回想上文由碳氧原子通过不同组合形成若干分子的例子。若从分子层面继续迭代这种组合思想，可以形成DNA，细胞，组织，器官，最终可以形成一个完整的人。继续迭代还会有家庭，公司，国家等。这种现象在身边随处可见。并且原子的内部结构与太阳系又惊人的相似。不同层级之间都是以类似的几种规则再不断形成新物质。你也可能听过分形学这三个字。可通过观看 [从1米到150亿光年来感受自然界这种层级现象的普遍性](#)。



- 人脸识别情景：我们可以模拟这种思想并应用在画面识别上。由像素组成菱角，再组成五官，最后到不同的人脸。每一层代表不同的物质层面(如分子层)。而每层的  $W$  存储着如何组合上一层的物质从而形成若干新物质。如果我们完全掌握一架飞机是如何从分子开始一层一层形成的，拿到一堆分子后，我们就可以判断他们是否可以以此形成方式，形成一架飞机。附：[Tensorflow playground](#)展示了数据是如何“流动”的。



物质组成视角：神经网络的学习过程就是学习物质组成方式的过程。

增加节点数：增加同一层物质的种类，比如118个元素的原子层就有118个节点。

增加层数：增加更多层级，比如分子层，原子层，器官层，并通过判断更抽象的概念来识别物体。

按照上文在理解视角中所述的观点，可以想出下面两条理由关于为什么更深的网络会更加容易识别，增加容纳变体（variation）（红苹果、绿苹果）的能力、鲁棒性（robust）。

- 数学视角：变体（variation）很多的分类的任务需要高度非线性的分割曲线。不断的利用那5种空间变换操作将原始输入空间像“捏橡皮泥一样”在高维空间下捏成更为线性可分/稀疏的形状：可视化空间变换。

- 物理视角：通过对“抽象概念”的判断来识别物体，而非细节。比如对“飞机”的判断，即便人类自己也无法用语言或者若干条规则来解释自己如何判断一个飞机。因为人脑中真正判断的不是是否“有机翼”、“能飞行”等细节现象，而是一个抽象概念。层数越深，这种概念就越抽象，所能涵盖的变异体就越多，就可以容纳战斗机，客机等多种不同种类的飞机。

然而这最多指解释了为何神经网络有效，并没有接触到核心问题：为何深层神经网络更有效

# 梯度下降训练法

## 如何训练：

既然我们希望网络的输出尽可能的接近真正想要预测的值。那么就可以通过比较当前网络的预测值和我们真正想要的目标值，再根据两者的差异情况来更新每一层的权重矩阵（比如，如果网络的预测值高了，就调整权重让它预测低一些。不断调整，直到能够预测出目标值）。

因此就需要先定义“如何比较预测值和目标值的差异”，这便是损失函数或目标函数（**loss function or objective function**），用于衡量预测值和目标值的差异的方程。**loss function**的输出值（**loss**）越高表示差异性越大。那神经网络的训练就变成了尽可能的缩小**loss**的过程。

所用的方法是梯度下降（**Gradient descent**）：通过使**loss**值向当前点对应梯度的反方向不断移动，来降低**loss**。一次移动多少是由学习速率（**learning rate**）来控制的。

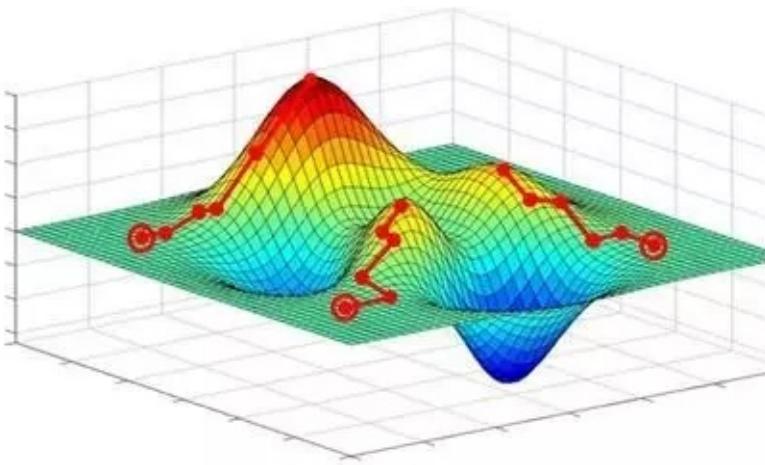
---

## 梯度下降的问题：

然而使用梯度下降训练神经网络拥有两个主要难题。

### 1、局部极小值(或鞍点)

梯度下降寻找的是**loss function**的局部极小值，而我们想要全局最小值。如下图所示，我们希望**loss**值可以降低到右侧深蓝色的最低点，但**loss**在下降过程中有可能“卡”在左侧的局部极小值中。也有最新研究表明在高维空间下局部极小值通常很接近全局最小值，训练网络时真正与之“斗争”的是鞍点。但不管是什么，其难处就是**loss**“卡”在了某个位置后难以下降。唯一的区别是：陷入局部极小值就难以出来，陷入鞍点最终会逃脱但是耗时。



试图解决“卡在局部极小值”问题的方法分两大类：

- 调节步伐：调节学习速率，使每一次的更新“步伐”不同。常用方法有：
  - 随机梯度下降（Stochastic Gradient Descent (SGD)）：每次只更新一个样本所计算的梯度
  - 小批量梯度下降（Mini-batch gradient descent）：每次更新若干样本所计算的梯度的平均值
  - 动量（Momentum）：不仅仅考虑当前样本所计算的梯度；Nesterov动量（Nesterov Momentum）：Momentum的改进
  - Adagrad、RMSProp、Adadelta、Adam：这些方法都是训练过程中依照规则降低学习速率，部分也综合动量
- 优化起点：合理初始化权重（weights initialization）、预训练网络（pre-train），使网络获得一个较好的“起始点”，如最右侧的起始点就比最左侧的起始点要好。常用方法有：高斯分布初始权重（Gaussian distribution）、均匀分布初始权重（Uniform distribution）、Glorot 初始权重、He 初始权、稀疏矩阵初始权重（sparse matrix）

## 2、梯度的计算

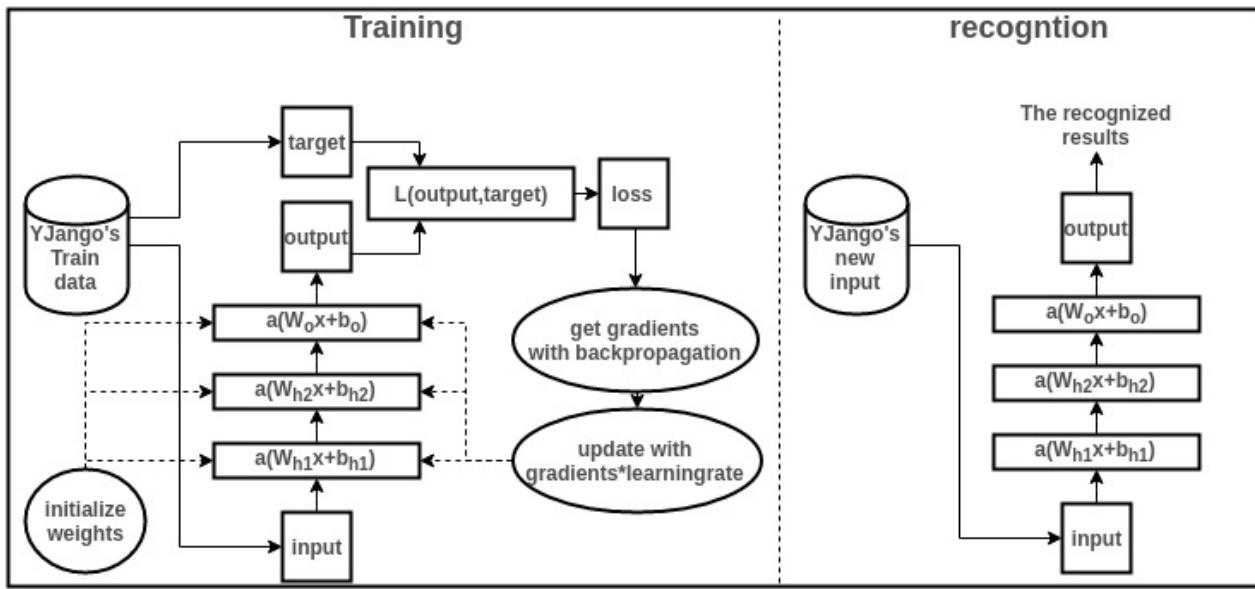
机器学习所处理的数据都是高维数据，该如何快速计算梯度、而不是以年来计算。其次如何更新隐藏层的权重？解决方法是：计算图：反向传播算法 这里的解释留给非常棒的 [Computational Graphs: Backpropagation](#)

需要知道的是，反向传播算法是求梯度的一种方法。如同快速傅里叶变换（FFT）的贡献。而计算图的概念又使梯度的计算更加合理方便。

---

## 基本流程图

下面就简单浏览一下训练和识别过程，并描述各个部分的作用。

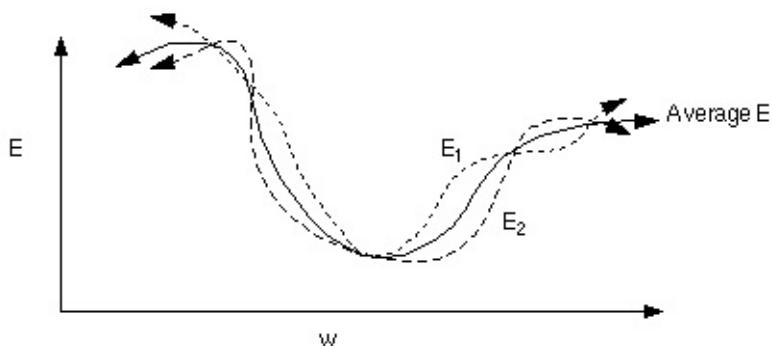


- 收集训练集（**train data**）：也就是同时有input以及对应label的数据。每个数据叫做训练样本（**sample**）。label也叫target，也是机器学习中最贵的部分。上图表示的是我的数据仓库。假设input的维度是39，label的维度是48。
- 设计网络结构（**architecture**）：确定层数、每一隐藏层的节点数和激活函数，以及输出层的激活函数和损失函数。上图用的是两层隐藏层（最后一层是输出层）。隐藏层所用激活函数a()是ReLU，输出层的激活函数是线性linear（也可看成是没有激活函数）。隐藏层都是1000节点。损失函数L()是用于比较距离MSE：mean((output - target)^2)。MSE越小表示预测效果越好。训练过程就是不断减小MSE的过程。到此所有数据的维度都已确定：
  - 训练数据： $input \in R^{39}$  ;  $label \in R^{48}$
  - 权重矩阵： $W_{h1} \in R^{1000 \times 39}$  ;  $W_{h2} \in R^{1000 \times 1000}$  ;  $W_o \in R^{48 \times 1000}$
  - 偏移向量： $b_{h1} \in R^{1000}$  ;  $b_{h2} \in R^{1000}$  ;  $b_o \in R^{48}$
  - 网络输出： $output \in R^{48}$
- 数据预处理（**preprocessing**）：将所有样本的input和label处理成能够使用神经网络的数据。label的值域符合激活函数的值域。并简单优化数据以便让训练易于收敛。比如中心化（mean subtraction）、归一化（normalization）、主成分分析（PCA）、白化（whitening）。假设上图的input和output全都经过了中心化和归一化。
- 权重初始化（**weights initialization**）： $W_{h1}$ 、 $W_{h2}$ 、 $W_o$ 在训练前不能为空，要初始化才能够计算loss从而降低。 $W_{h1}$ 、 $W_{h2}$ 、 $W_o$ 初始化决定了loss在loss function中从哪个点开始作为起点训练网络。上图用均匀分布初始权重（Uniform distribution）。
- 训练网络（**training**）：训练过程就是用训练数据的input经过网络计算出output，再和label计算出loss，再计算出gradients来更新weights的过程。
  - 正向传递：，算当前网络的预测值

$$output = linear(W_o \cdot Relu(W_{h2} \cdot Relu(W_{h1} \cdot input + b_{h1}) + b_{h2}) + b_o)$$

- 计算 loss :  $loss = mean((output - target)^2)$

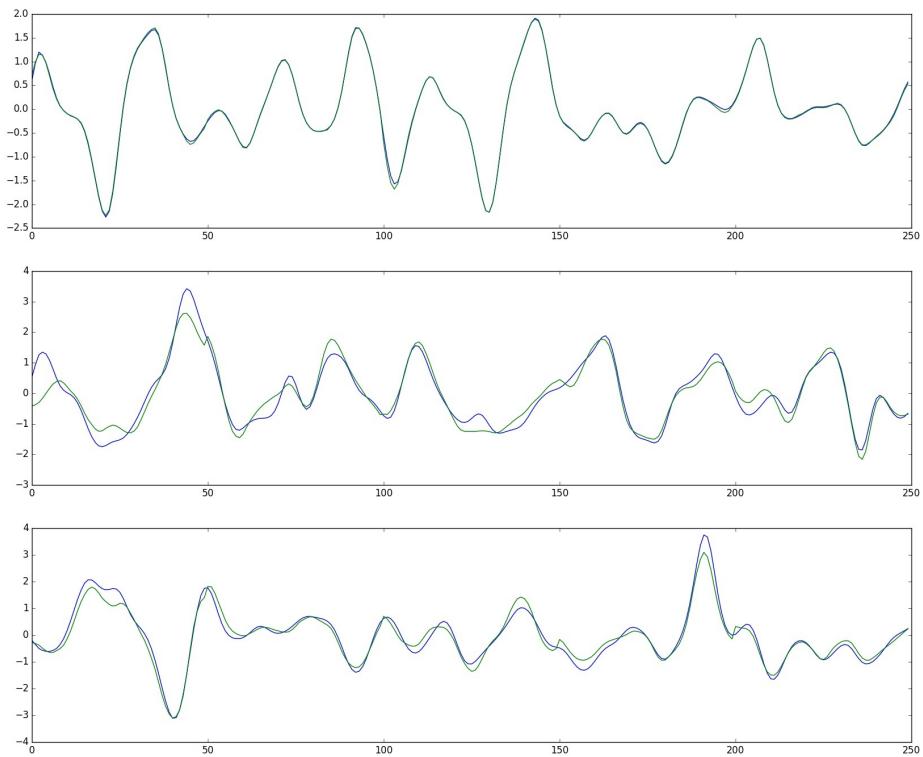
- 计算梯度：从 loss 开始反向传播计算每个参数 (parameters) 对应的梯度 (gradients)。这里用 Stochastic Gradient Descent (SGD) 来计算梯度，即每次更新所计算的梯度都是从一个样本计算出来的。传统的方法 Gradient Descent 是正向传递所有样本来计算梯度。SGD 的方法来计算梯度的话，loss function 的形状如下图所示会有变化，这样在更新中就有可能“跳出”局部最小值。



- 更新权重：这里用最简单的方法来更新，即所有参数都

$$W = W - learningrate * gradient$$

- 预测新值：训练过所有样本后，打乱样本顺序再次训练若干次。训练完毕后，当再来新的数据 input，就可以利用训练的网络来预测了。这时的 output 就是效果很好的预测值了。下图是一张实际值和预测值的三组对比图。输出数据是 48 维，这里只取 1 个维度来画图。蓝色的是实际值，绿色的是预测值。最上方的是训练数据的对比图（几乎重合），而下方的两行是神经网络模型从未见过的数据预测对比图。（不过这里用的是 RNN，主要是为了让大家感受一下效果）



## 结合实例理解

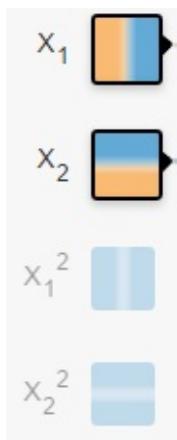
### 物质组成视角

下面就结合[Tensorflow playground](#)理解**5**种空间操作和物质组成视角。打开网页后，总体来说，蓝色代表正值，黄色代表负值。拿分类任务来分析。

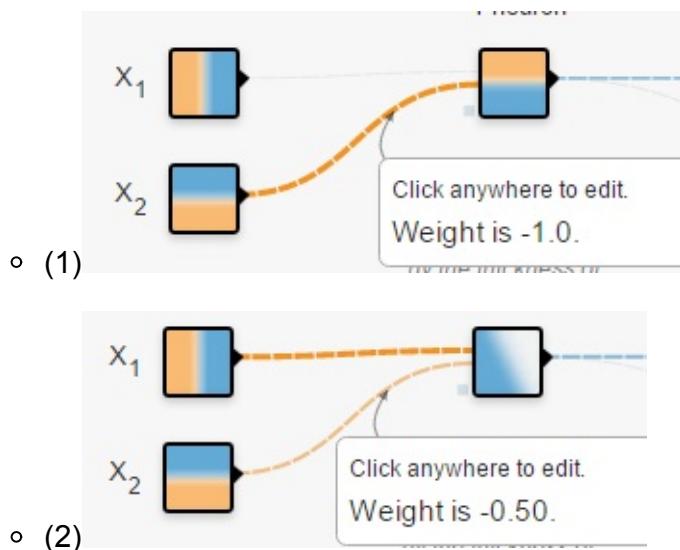
- 数据：在二维平面内，若干点被标记成了两种颜色。黄色，蓝色，表示想要区分的两类。你可以把平面内的任意点标记成任意颜色。网页给你提供了**4**种规律。神经网络会根据你给的数据训练，再分类相同规律的点。



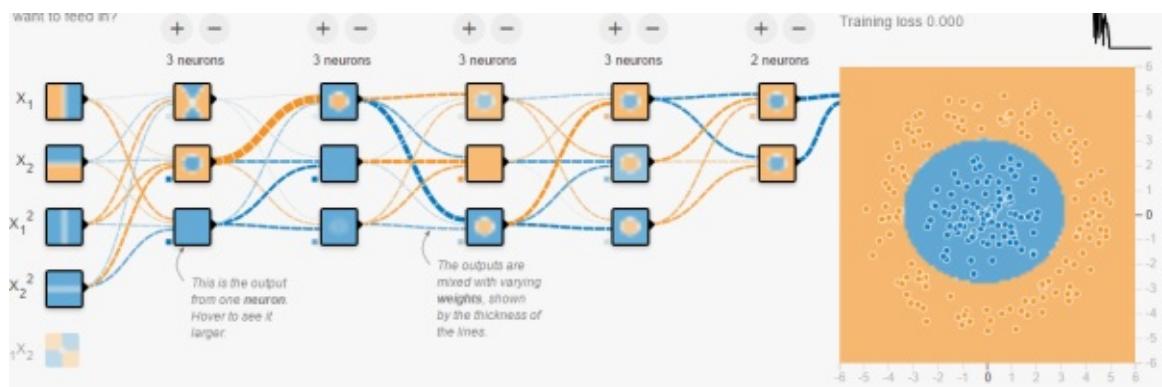
- 输入：在二维平面内，你想给网络多少关于“点”的信息。从颜色就可以看出来， $x_1$ 左边是负，右边是正， $x_1$ 表示此点的横坐标值。同理， $x_2$ 表示此点的纵坐标值。 $x_1^2$ 是关于横坐标值的“抛物线”信息。你也可以给更多关于这个点的信息。给的越多，越容易被分开。



- 连接线：表示权重，蓝色表示用神经元的原始输出，黄色表示用负输出。深浅表示权重的绝对值大小。鼠标放在线上可以看到具体值。也可以更改。在（1）中，当把 $x_2$ 输出的一个权重改为-1时， $x_2$ 的形状直接倒置了。不过还需要考虑激活函数。（1）中用的是linear。在（2）中，当换成sigmoid时，你会发现没有黄色区域了。因为sigmoid的值域是(0,1)



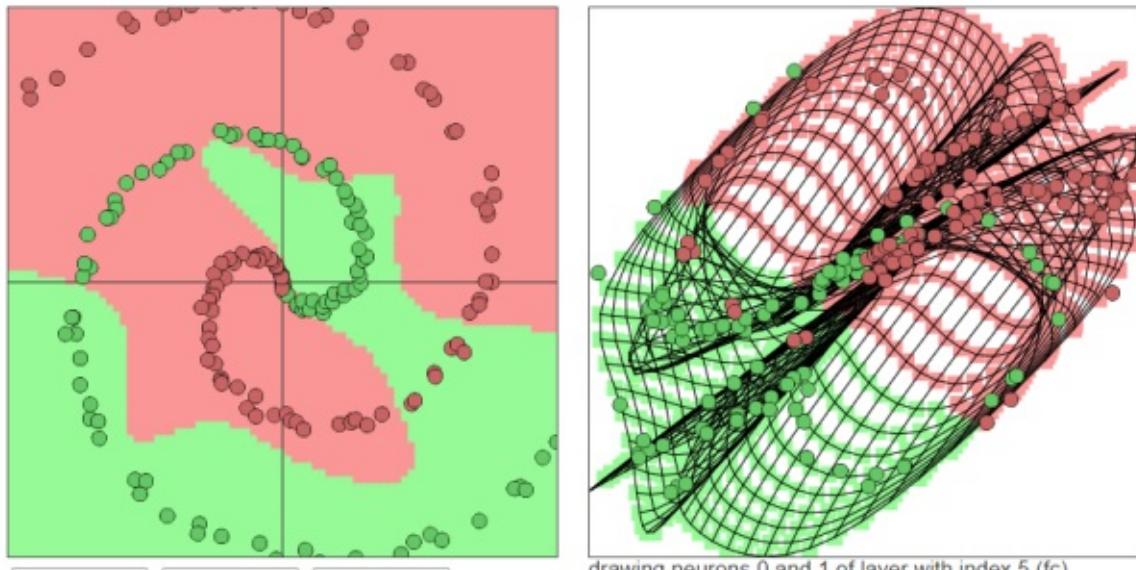
- 输出：黄色背景颜色都被归为黄点类，蓝色背景颜色都被归为蓝点类。深浅表示可能性的强弱。



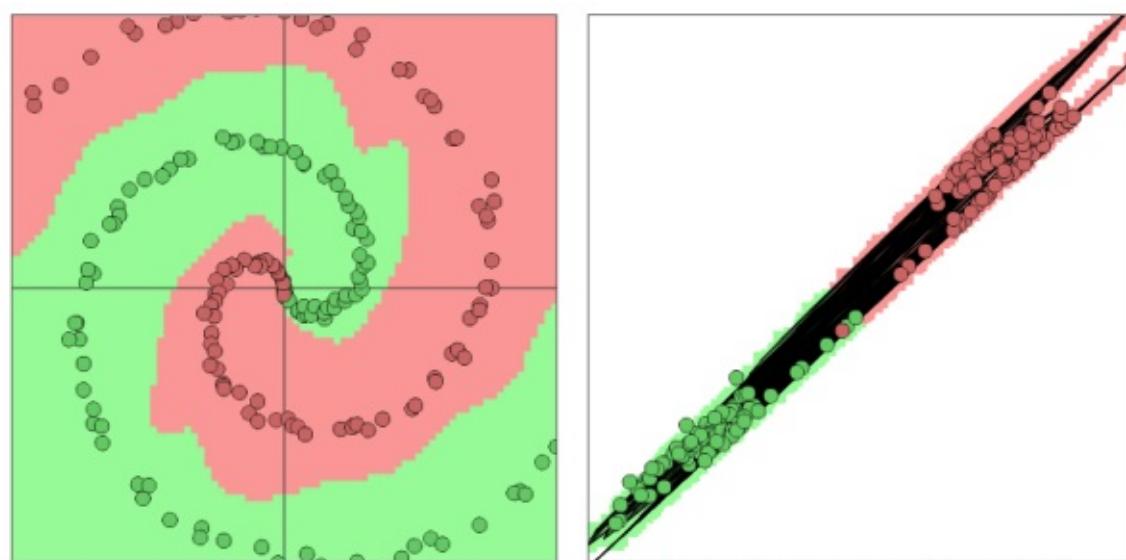
上图中所有在黄色背景颜色的点都会被分类为“黄点”，同理，蓝色区域被分成蓝点。在上面的分类分布图中你可以看到每一层通过上一层信息的组合所形成的。权重（那些连接线）控制了“如何组合”。神经网络的学习也就是从数据中学习那些权重。Tensorflow playground所表现出来的现象就是“在我文章里所写的“物质组成思想”，这也是为什么我把Tensorflow playground放在了那一部分。不过你要是把Tensorflow的名字拆开来看的话，是tensor（张量）的flow（流动）。Tensorflow playground的作者想要阐述的侧重点是“张量如何流动”的。

## 5种空间变换的理解

Tensorflow playground下没有体现5种空间变换的理解。需要打开这个网站尝试：[ConvNetJS demo: Classify toy 2D data](#)



左侧是原始输入空间下的分类图，右侧是转换后的高维空间下的扭曲图。



最终的扭曲效果是所有绿点都被扭曲到了一侧，而所有红点都被扭曲到了另一侧。这样就可以线性分割（用超平面（这里是一个平面）在中间分开两类）

---

此部分内容不是这篇文章的重点，是为了理解深层神经网络，需要明白最基本的训练过程。

若能理解训练过程是通过梯度下降尽可能缩小loss的过程即可。

若有理解障碍，可以用python实践一下[从零开始训练一个神经网络](#)，体会整个训练过程。

若有时间则可以再体会一下计算图自动求梯度的方便[利用TensorFlow](#)。

# 未完稿

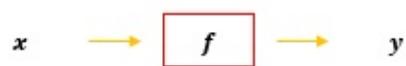
一切皆在变化。

这一部分是关于梯度下降训练法的一个实例演示。而在那之前，先简单的复习一下要用到的知识：微积分。

## 微积分简括

我们做有监督学习时，是从大量的 $\{(x_i, y_i)_{i=1}^N\}$ 个样本中，寻找可以较好预测未见过 $x_{new}$ 所对应 $y_{new}$ 的函数 $f : x \rightarrow y$ 。

实例：在我们日常生活的学习中，大量的 $\{(x_i, y_i)_{i=1}^N\}$ 就是历年真题， $x_i$ 题目，而 $y_i$ 是对应的正确答案。高考时将会遇到的 $x_{new}$ 往往是我们没见过的题目，希望可以通过做题训练出来的解题方法 $f : x \rightarrow y$ 来求解出正确的 $y_{new}$ 。



问题描述

解题方法

对应答案



未完

# 为何深层学习

深层学习开启了人工智能的新时代。不论任何行业都害怕错过这一时代浪潮，因而大批资金和人才争相涌入。但深层学习却以“黑箱”而闻名，不仅调参难，训练难，“新型”网络结构的论文又如雨后春笋般地涌现，使得对所有结构的掌握变成了不现实。我们缺少一个对深层学习合理的认识。

神经网络并不缺少新结构，但缺少一个该领域的 $E = mc^2$

很多人在做神经网络的实验时会发现调节某些方式和结构会产生意想不到的结果。但就我个人而言，这些发现并不会让我感到满足。我更关心这些新发现到底告诉我们了什么，造成这些现象的背后原因是什么。我会更想要将新的网络结构归纳到已有的体系当中。这也是我更多思考“为何深层学习有效”的原因。下面便是目前YJango关于这方面的见解。

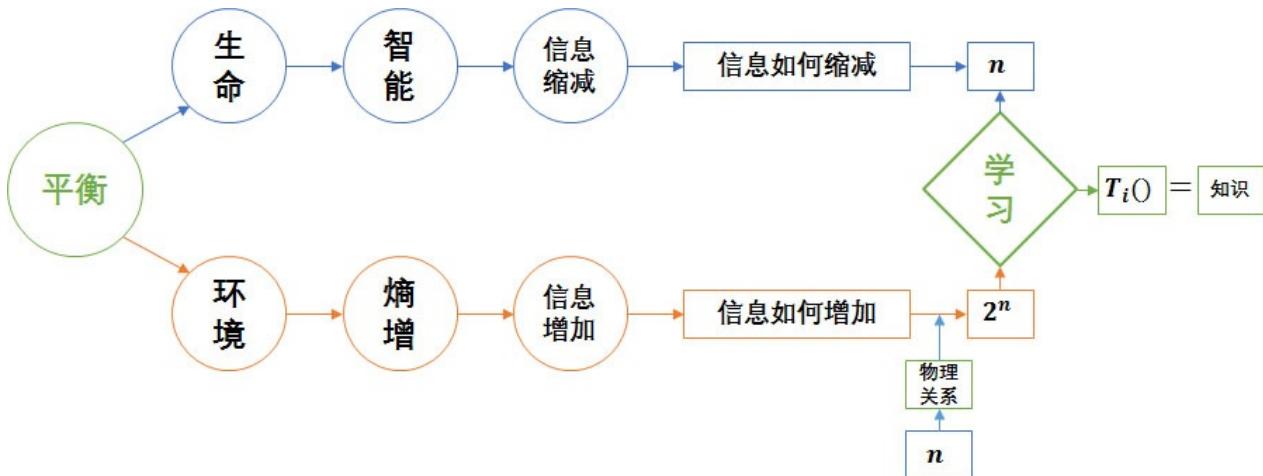
深层神经网络相比一般的统计学习拥有从数学的严谨中不会得出的关于物理世界的先验知识（非贝叶斯先验）。该内容也在Bengio大神的论文和演讲中多次强调。大神也在Bay Area Deep Learning School 2016的[Foundations and Challenges of Deep Learning pdf](#)（这里也有视频，需翻墙）中提到的distributed representations和compositionality两点就是神经网络和深层神经网络高效的原因（若有时间，强烈建议看完演讲再看该文）。虽然与大神的思考起点可能不同，但结论完全一致（看到Bengio大神的视频时特别兴奋）。下面就是结合例子分析：

1. 为什么神经网络高效
2. 学习的本质是什么
3. 为什么深层神经网络比浅层神经网络更高效
4. 神经网络在什么问题上不具备优势

## 其他推荐读物

- Bengio Y. Learning deep architectures for AI[J]. Foundations and trends® in Machine Learning, 2009, 2(1): 1-127.
- Brahma P P, Wu D, She Y. Why Deep Learning Works: A Manifold Disentanglement Perspective[J]. 2015.
- Lin H W, Tegmark M. Why does deep and cheap learning work so well?[J]. arXiv preprint arXiv:1608.08225, 2016.
- Bengio Y, Courville A, Vincent P. Representation learning: A review and new perspectives[J]. IEEE transactions on pattern analysis and machine intelligence, 2013, 35(8): 1798-1828.
- [Deep Learning textbook by Ian Goodfellow and Yoshua Bengio and Aaron Courville](#)

YJango的整个思考流程都围绕减熵二字进行。之前在《熵与生命》和《生物学习》中讨论过，生物要做的是降低环境的熵，将不确定状态变为确定状态。通常机器学习是优化损失函数，并用概率来衡量模型优劣。然而概率正是由于无法确定状态才不得不用的衡量手段。生物真正想要的是没有丝毫不确定性。



深层神经网络在自然问题上更具优势，因为它和生物学习一样，是找回使熵增加的“物理关系”（知识，并非完全一样），将变体 ( $2^n$ ) 转化回因素 ( $n$ ) 附带物理关系的形式，从源头消除熵（假设每个因素只有两种可能状态）。这样所有状态间的关系可以被确定，要么肯定发生，要么绝不发生，也就无需用概率来衡量。因此下面定义的学习目标并非单纯降低损失函数，而从确定关系的角度考虑。一个完美训练好的模型就是两个状态空间内所有可能取值间的关系都被确定的模型。

**学习目标：**是确定 (determine) 两个状态空间内所有可能取值之间的关系，使得熵尽可能最低。

注：对熵不了解的朋友可以简单记住，事件的状态越确定，熵越小。如绝不发生（概率0）或肯定发生（概率为1）的事件熵小。而50%可能性事件的熵反而大。

为举例说明，下面就一起考虑用神经网络学习以下两个集合的不同关联（OR gate和XOR gate）。看看随着网络结构和关联的改变，会产生什么不同情况。尤其是最后网络变深时与浅层神经网络的区别。

注：选择这种XOR这种简单关联的初衷是输入和输出空间状态的个数有限，易于分析变体个数和熵增的关系。

注：用“变体 (variation)”是指同一类事物的不同形态，比如10张狗的图片，虽然外貌各异，但都是狗。

问题描述：集合  $A$  有4个状态，集合  $B$  有2个状态。0和1只是用于表示不同状态的符号，也可以用0,1,2,3表示。状态也并不一定表示数字，可以表示任何物理意义。

$$A = \{00, 01, 10, 11\}$$

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

## 方式1：记忆

- 随机变量  $X$ ：可能取值是  $\{00, 01, 10, 11\}$
- 随机变量  $Y$ ：可能取值是  $\{0, 1\}$

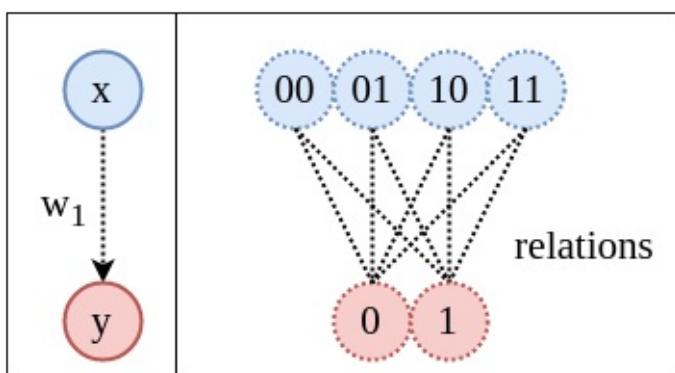
注：随机变量（大写  $X$ ）是将事件投射到实数的函数。用对应的实数表示事件。而小写字母  $x$  表示对应该实数的事件发生了，是一个具体实例。

- 网络结构：暂且不规定要学习的关联是OR还是XOR，先建立一个没有隐藏层，仅有一个输入节点，一个输出节点的神经网络。

- 表达式： $y = M(x) = \phi(w_1 \cdot x + b)$ ,  $\phi$  表示 sigmoid 函数。
- 说明：下图右侧中的虚线表示的既不是神经网络的链接，也不是函数中的映射，而是两个空间中，所有可能值之间的关系（relation）。学习的目的是确定这些状态的关系。比如当输入00时，模型要尝试告诉我们00到1的概率为0，00到0的概率为1，这样熵  $H(X) = - \sum_i p_i(x) \log p_i(x)$  才会为零。

- 关系图：左侧是网络结构，右侧是状态关系图。输入和输出空间之间共有8个关系（非箭头虚线表示关系）。除非这8个关系对模型来说都是相同的，否则用  $w_{h1}$  表示  $f : X \rightarrow Y$  时的熵  $H(M(X), X)$  就会增加。（ $w_{h1}$  无法照顾到8个关系，若完美拟合一个关系，其余的关系就不准确）

注：这里  $YJango$  是  $w_{h1}$  用表示  $\phi(w_{h1} \cdot x + b)$  的缩写。



- 数据量：极端假设，若用查找表来表示关系：需要用8个不同的  $(x, y)$  数据来记住想要拟合的  $f : X \rightarrow Y$ 。

## 方式2：手工特征

- 特征：空间  $A$  的4个状态是由两个0或1的状态共同组成。我们可以观察出来（计算机并不

能），我们利用这种知识  $k()$  把  $A$  中的状态分解开（disentangle）。分解成两个独立的子随机变量  $H_1 = \{0, 1\}$  和  $H_2 = \{0, 1\}$ 。也就是用二维向量表示输入。

- 网络结构：由于分成了二维特征，这次网络结构的输入需改成两个节点。下图中的上半部分是，利用人工知识  $k()$  将随机变量  $X$  无损转变为  $H_1$  和  $H_2$  的共同表达（representation）。这时  $h_1$  和  $h_2$  一起形成网络输入。

注： $k()$  旁边的黑线（实线表示确定关系）并非是真正的神经网络结构，只是方便理解，可以简单想象成神经网络转变的。

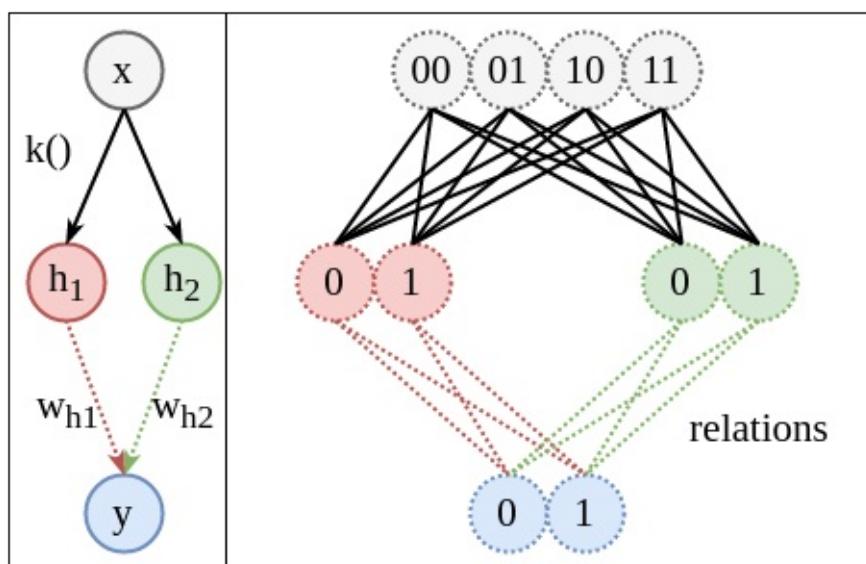
- 表达式： $y = M(h) = \phi(W_h \cdot h + b)$

注：方便起见， $w_{h1} \cdot h_1 + w_{h2} \cdot h_2$  写成了矩阵的表达形式  $W_h \cdot h$ ，其中  $b$  是标量，

$$\text{而 } W_h = [w_{h1} \quad w_{h2}], \vec{h} = \begin{bmatrix} h_1 \\ h_2 \end{bmatrix}$$

- 关系图：由于  $k()$  固定，只考虑下半部分的关系。因为这时用了两条线  $w_{h1}$  和  $w_{h2}$  来共同对应关系。原本需要拟合的8个关系，现在变成了4个（两个节点平摊）。同样，除非右图的4条红色关系线对  $w_{h1}$  来说相同，并且4条绿色关系线对  $w_{h2}$  来说也相同，否则用  $w_{h1}$  和  $w_{h2}$  表示  $f : X \rightarrow Y$  时，一定会造成熵  $H(M(X), X)$  增加。

注：下图中左侧是网络结构图。右侧关系图中，接触的圆圈表示同一个节点的不同变体。分开的、并标注为不同颜色的圆圈表示不同节点，左右两图连线的颜色相互对应，如红色的  $w_{h1}$  需要表示右侧的4条红色关系线。



- 关联1：下面和YJango确定想要学习的关联（函数）。如果想学习的关联是只取  $H_1$  或者  $H_2$  的值，那么该结构可以轻松用两条线  $w_{h1}$  和  $w_{h2}$  来表达这4个关系（让其中一条线的权重为0，另一条为1）。

- **关联2：**如果想学习的关联是或门，真值表和实际训练完的预测值对照如下。很接近，但有误差。不过若要是分类的话，可以找到0.5这个超平面来分割。大于0.5的就是1，小于0.5的就是0，可以完美分开。

注：第一列是输入，第二列是真实想要学习的输出，第三列是训练后的网络预测值。

$H_1$ and $H_2$	Y	predict
[0,0]	0	0.25
[0,1]	1	0.75
[1,0]	1	0.75
[1,1]	1	1.25

- **关联3：**如果想学习的关联是异或门（XOR），真值表和实际训练完的预测值对照如下。由于4条关系线无法用单个 $W$ 表达，该网络结构连XOR关联的分类都无法分开。

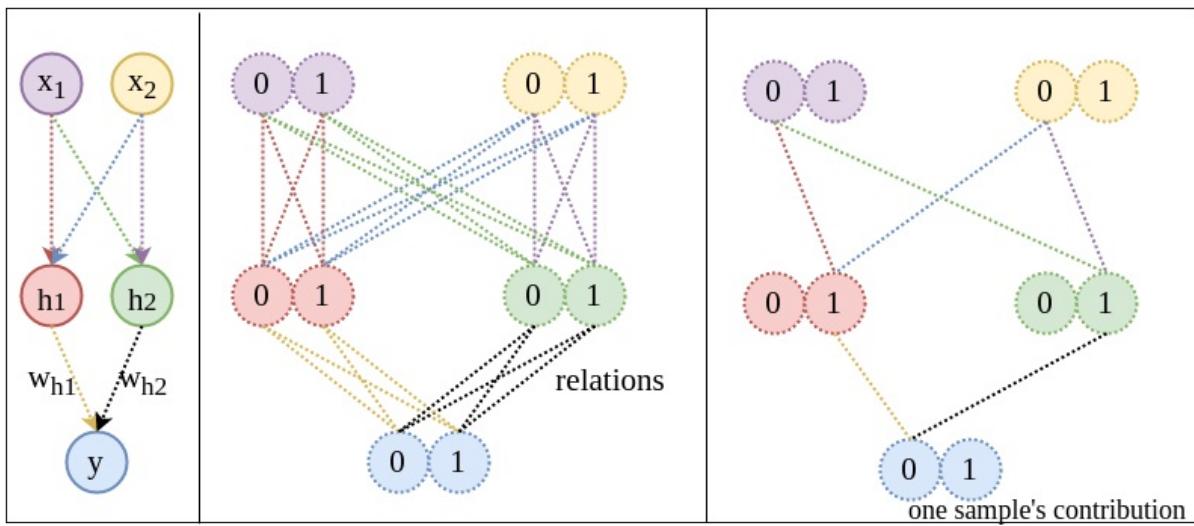
$H_1$ and $H_2$	Y	predict
[0,0]	0	0.5
[0,1]	1	0.5
[1,0]	1	0.5
[1,1]	0	0.5

- 数据量：学习这种关联至少需4个不同的 $([h_1, h_2], y)$ 来拟合 $f_{hy} : H \rightarrow Y$ 。其中每个数据可以用于确定2条关系线。

### 方式3：加入隐藏层

- **网络结构1：**现在直接把 $h_1$ 和 $h_2$ 作为输入（用 $x_1, x_2$ 表示），不考虑 $k()$ 。并且在网络结构中加入一个拥有2个节点（node）隐藏层（用 $h_1, h_2$ 表示）。
- 表达式： $y = M(x) = \phi(W_h \cdot \phi(W_x \cdot x + b_x) + b_h)$
- **关系图1：**乍一看感觉关系更难确定了。原来还是只有8条关系线，现在又多了16条。但实际上所需要的数据量丝毫没有改变。因为以下两条先验知识的成立。

注：下图最右侧是表示：当一个样本进入网络后，能对学习哪些关系线起到作用。



- 1. 并行： $f_{xh_1}: X_1, X_2 \rightarrow H_1$  和  $f_{xh_2}: X_1, X_2 \rightarrow H_2$  的学习完全是独立并行。这就是神经网络的两条固有先验知识中的：并行：网络可以同时确定  $f_{xh_1}$  和  $f_{xh_2}$  的关联。也是Bengio大神所指的distributed representation。

注：有效的前提是所要学习的状态空间的关联是可以被拆分成并行的因素（factor）。

注： $f_{hy}: H_1, H_2 \rightarrow Y$  就没有并行一说，因为  $Y$  是一个节点拥有两个变体，而不是两个独立的因素。但是也可以把  $Y$  拆开表示为 one-hot-vector。这就是为什么分类时并非用一维向量表示状态。更验证了 YJango 在机器学习中对学习定义：学习是将变体拆成因素附带关系的过程。

- 迭代：第二个先验知识是：在学习  $f_{hy}: H_1, H_2 \rightarrow Y$  的同时， $f_{xh_1}: X_1, X_2 \rightarrow H_1$  和  $f_{xh_2}: X_1, X_2 \rightarrow H_2$  也可以被学习。这就是神经网络的两条固有先验知识中的：迭代：网络可以在确定上一级的同时确定下一级的所有内容。也是Bengio大神所指的compositionality。

注：有效的前提是所要学习的空间的关联是由上一级迭代形成的。所幸的是，我们所生活的世界几乎都符合这个前提（有特殊反例）。

- 关联：如果想学习的关联是异或门（XOR），真值表和实际训练完的预测值对照如下。

$f_{xh_1}$  和  $f_{xh_2}$ ：期初若用两条网络连接表示  $X_1, X_2 \rightarrow H_1, H_2$  的 16 个关系可能，那么熵会很高。但用两条线表示  $X_1, X_2 \rightarrow H_1$  的 8 个关系，模型的熵可以降到很低。下图中  $f_{xh_1}$  的输出值对应红色数字。 $f_{xh_2}$  对应输出值是由蓝色数字表达。

$f_{hy}$ ：这时再看  $H_1, H_2 \rightarrow Y$  的关系，完全就是线性的。光靠观察就能得出  $f(h_1, h_2)$  的一个表达。

$X_1$ and $X_2$	$Y$	$H_1$ and $H_2$	predict
[0,0]	0	[1,1]	0
[0,1]	1	[1,0]	1
[1,0]	1	[0,1]	1
[1,1]	0	[1,1]	0

$f(h_1, h_2) = -h_1 - h_2 + 2$

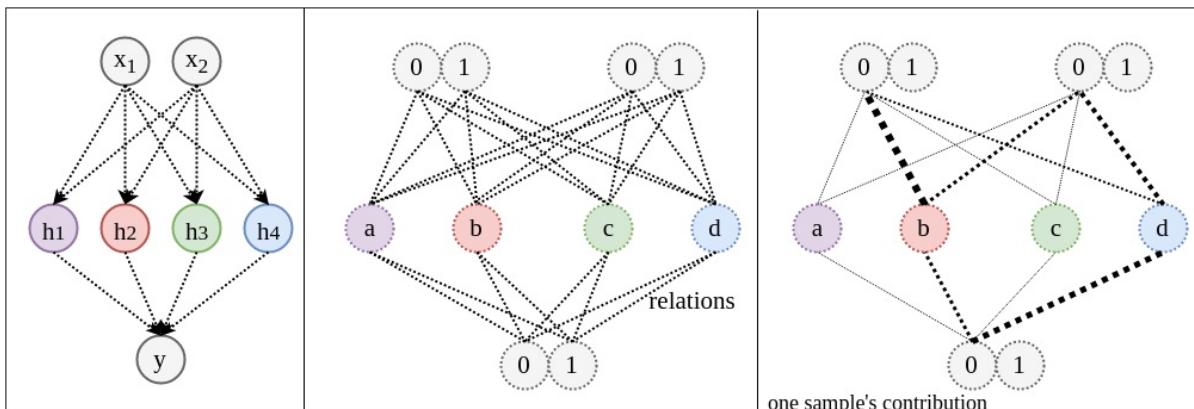
- 数据量：如关系图1中最右侧图所示，一个输入 $[0, 0]$ 会被关联到0。而这个数据可用于学习 $2 + 4$ 个关系。也就是说网络变深并不需要更多数据。

模型的熵  $H(M(X), X)$  与用一条  $\phi(w_1 \cdot x + b)$  所要拟合的关系数量有关。也就是说，

变体 (variation) 越少，拟合难度越低，熵越低。

- 网络结构2：既然这样， $X$ 有4个变体，这次把节点增加到4。
- 关系图2：与网络结构1不同，增加到4个节点后，每个节点都可以完全没有变体，只取一个值。想法很合理，但实际训练时，模型不按照该方式工作。

注：太多颜色容易眼花。这里不再用颜色标注不同线之间的对应关系，但对应关系依然存在。



- 问题：因为会出现右图的情况：只有两个节点在工作（线的粗细表示权重大小）。 $a$  和  $c$  的节点在滥竽充数。这就跟只有两个节点时没有太大区别。原因是神经网络的权重的初始化是随机的，数据的输入顺序也是随机的。这些随机性使得权重更新的方向无法确定。

讨论：网络既然选择这种方式来更新权重，是否一定程度上说明，用较少的节点就可以表示该关联？并不是，原因在于日常生活中的关联，我们无法获得所有变体的数据。所得数据往往是很小的一部分。较少的节点可以表示这一小部分数据的关联，但却无法涵盖所有变体情况。造成实际应用时效果不理想。

- 缓解：缓解的方式有L2正则化（L2 regularization）：将每层权重矩阵的平方和作为惩罚。

$$\lambda/2 \cdot \sum_w w^2$$

- 表达式： $\lambda/2 \cdot \sum_w w^2$ ， $\lambda$ 是惩罚的强弱，可以调节。除以2是为了求导方便（与后边的平方抵消）。
- 意义：当同一层的权重有个别值过高时，平方和就会增加。而模型在训练中会降低该惩罚。产生的作用是使所有权重平摊任务，让 $a, b, c, d$ 都有值。以这样的方式来使每个节点都工作，从而消除变体，可以缓解过拟合（overfitting）。
- 例如： $L2(W_{hy}) = \lambda/2 \cdot (w_{hy1}^2 + w_{hy2}^2 + w_{hy3}^2 + w_{hy4}^2)$

具有一个隐藏层的神经网络可以模拟任何函数，最糟的情况需要 $k^n$ 个节点。

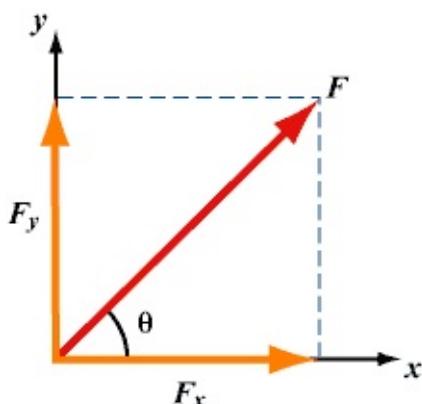
也叫Universal Approximation Theorem。但最糟的情况是输入空间有多少个变体，就需要多少个节点。 $k$ 表示独立因素的变体个数， $n$ 表示独立因素的个数。上述的例子中最糟的情况需要 $2^2$ 个隐藏节点。而代价也是需要 $k^n$ 个不同数据才可以完美拟合。

引入Christopher Olah在博客中的描述

Universality means that a network can fit to any training data you give it. It doesn't mean that it will interpolate to new data points in a reasonable way. [出处](#)

也就是说一个具有隐藏层的神经网络可以拟合任何训练数据，但是并不意味着能够合理预测未见过的数据。若需要见过所有情况才能够良好的预测，那就无法达到学习的目的。（学习就是为了从有限的样本中提取出规律来预测未见过的样本）

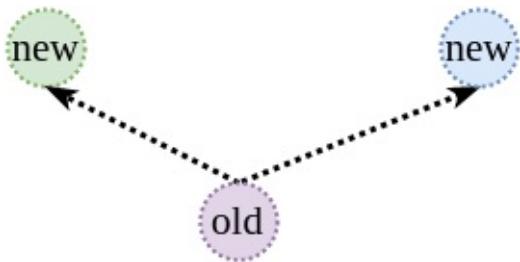
- 使用条件：浅层神经网络可以分开几乎所有自然界的关联。因为神经网络最初就是由于可移动的生物需要预测未来事件所进化而来的。所学习的关联是过去状态到未来状态。如下图，物理中的力也可以分离成两个独立的分力来研究。



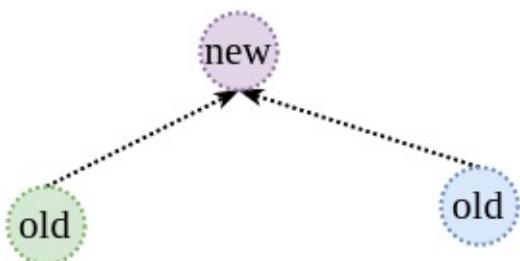
但有一种不适用的情况：非函数。

- 实例：函数的定义是：每个输入值对应唯一输出值的对应关系。为什么这么定义函数？对应两个以上的输出值在数学上完全可以。但是在宏观的世界发展中却不常见。如下图：

- 时间顺流：不管从哪个起点开始，相同的一个状态（不是维度）可以独立发展到多个不同状态（如氧原子可演变成氧分子和臭氧分子）。也就热力学第二定律的自发性熵增：原始状态通过物理关系，形成更多变体。



- 时间倒流：宏观自然界中难以找到两个以上的不同状态共同收回一个状态的例子（如氧分子和臭氧分子无法合并成氧原子）。如果这个可以实现，熵就会自发性减少。也就不需要生物来消耗能量减熵。我们的房间会向整齐的状态发展。但这违背热力学第二定律。至少在我们的宏观世界中，这种现象不常见。所以进化而来的神经网络可以拟合任何函数，但在非函数上就没有什么优势。毕竟生物的预测是从过去状态到未来状态。也说明神经网络并不违背无免费午餐定理。



- 实例：XOR门的输入空间和输出空间若互换位置，则神经网络拟合出来的可能性就非常低（并非绝对无法拟合）。

#### 方式4：继续加深

浅层神经网络可以模拟任何函数，但数据量的代价是无法接受的。深层解决了这个问题。相比浅层神经网络，深层神经网络可以用更少的数据量来学到更好的拟合。上面的例子很特殊。因为  $2 * 2 = 4$ ,  $2^2 = 4$ ，比较不出来。下面YJango就换一个例子，并比较深层神经网络和浅层神经网络的区别。

- 问题描述：空间  $A$  有 8 个可能状态，空间  $B$  有 2 个可能状态：

$$A = \{0, 1, 2, 3, 4, 5, 6, 7\}$$

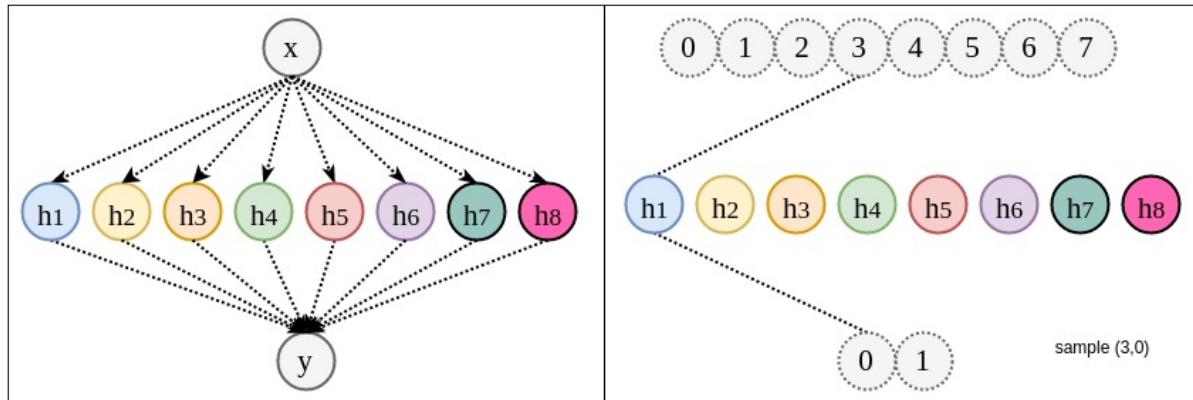
$$B = \{0, 1\}$$

- 网络结构：

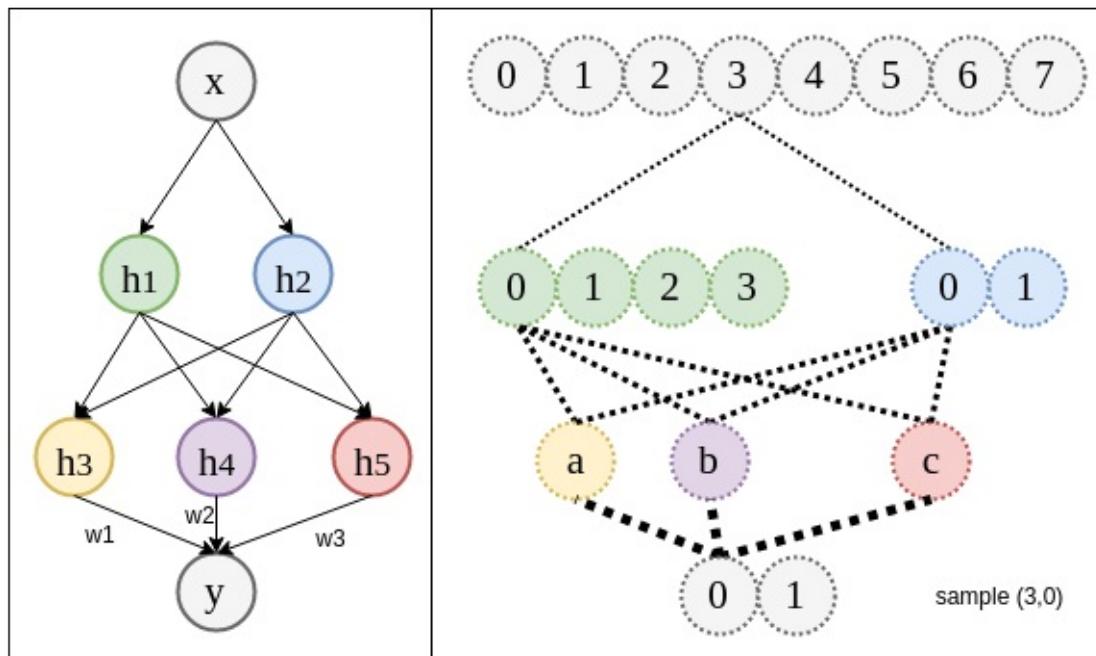
- 浅层神经网络：8 节点隐藏层
- 深层神经网络：2 节点隐藏层 + 3 节点隐藏层

- 深浅对比：

浅层神经网络：假如说输入3和输出0对应。数据 $(3, 0)$ 只能用于学习一个节点（如 $h_1$ ）前后的两条关系线。完美学习该关联需要所有8个变体。然而当变体数为 $10^{10}$ 时，我们不可能获得 $10^{10}$ 个不同变体的数据，也失去了学习的意义。毕竟我们是要预测没见过的数据。所以与其说这是学习，不如说这是强行记忆。好比一个学生做了100册练习题，做过的题会解，遇到新题仍然不会。他不是学会了做题，而是记住了怎么特定的题。



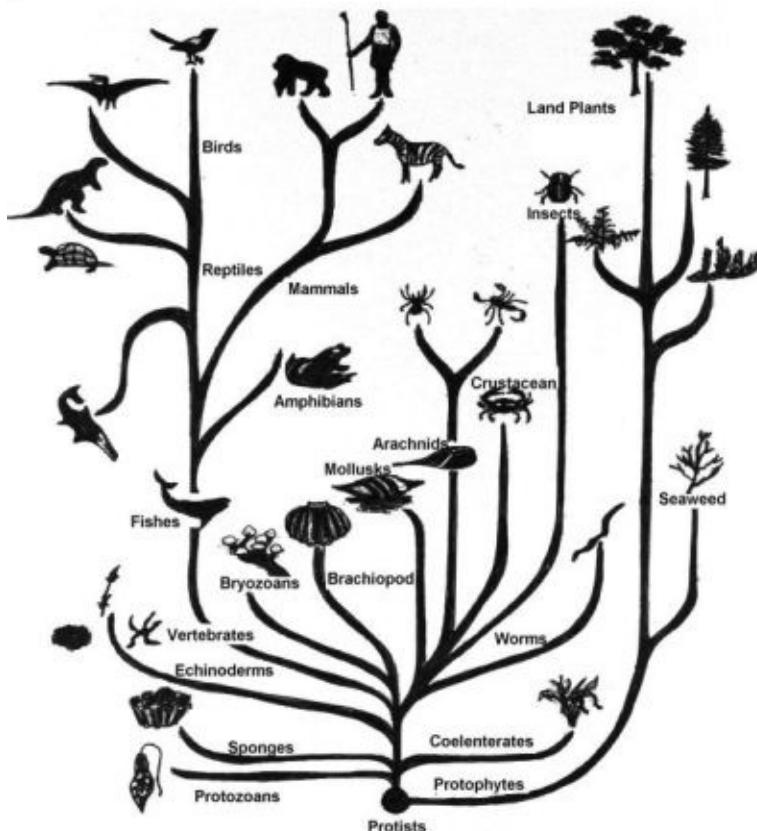
深层神经网络：如果只观察输入和输出，看起来同样需要8个不同的 $(x, y)$ 训练数据。但不同 $(x, y)$ 之间有共用部分。比如说，在确定3和0关系时，也同时对所有共用 $w_1, w_2, w_3$ 连接的其他变体进行确定。这样就使得原本需要8个不同数据才能训练好的关联变成只需要3,4个不同数据就可以训练好。（下图关系线的粗细并非表示权重绝对值，而是共用度）



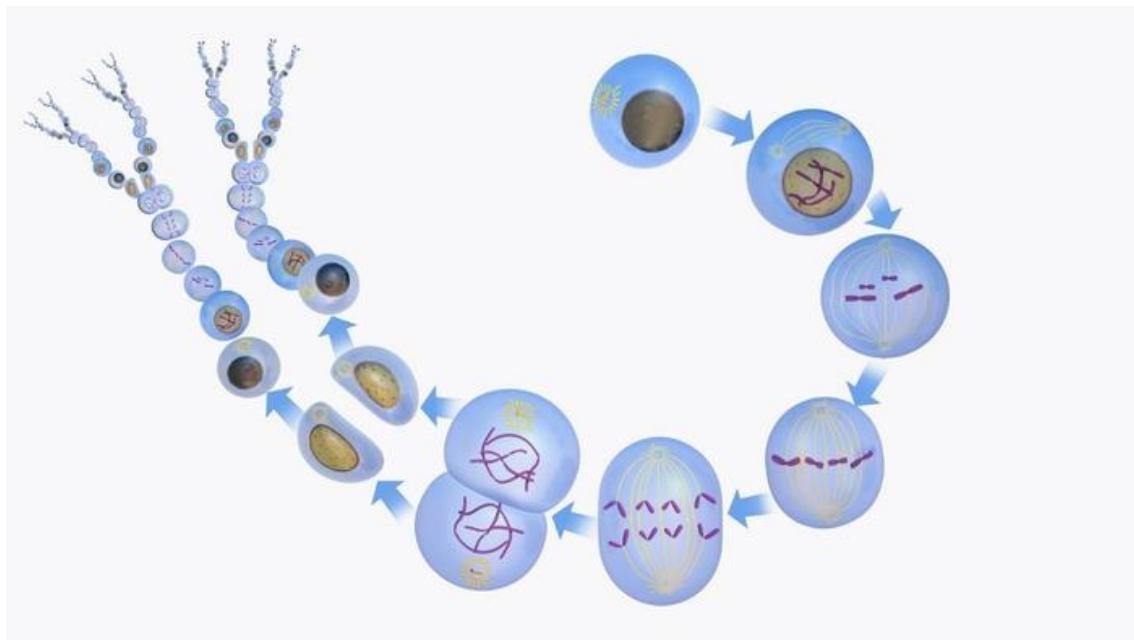
- 深层的前提：使用浅层神经网络好比是用 $y = ax + b$ 解 $a, b$ ，需要2个不同数据。而深层神经网络好比用 $y = ax$ 解 $a$ ，只需要一个数据。[无免费午餐定理](#)告诉我们，优化算法在一个任务上优秀，就一定会在其他任务上有缺陷，深层同样满足该定理。如果用

$y = ax$  去解实际上有  $b$  的  $y = ax + b$ ，或者去解实际为  $y = ax^2 + bx + c$  的关联时，搜索效果只会更差。所以深层的前提是： $X$  空间中的元素可以由  $Y$  迭代发展而来的。换句话说  $X$  中的所有变体，都有共用根源  $Y$  (root)。

- 我们的物理世界：幸运的是，我们的物理世界几乎都满足迭代的先验知识。
  - 实例：比如进化。不同动物都是变体，虽然大家现在的状态各异，但在过去都有共同的祖先。



- 实例：又如细胞分裂。八卦中的8个变体是由四象中4个变体的基础上发展而来，而四象又是由太极的2个变体演变而来。很难不回想起“无极生太极，太极生两仪，两仪生四象，四象生八卦”。（向中国古人致敬，虽然不知道他们的原意）



学习的过程是因素间的关系的拆分，关系的拆分是信息的回卷，信息的回卷是变体的消除，变体的消除是不确定性的缩减。

自然界两个固有的先验知识：

并行：新状态是由若干旧状态并行组合形成。

迭代：新状态由已形成的状态再次迭代形成。

为何深层学习：深层学习比浅层学习在解决结构问题上可用更少的数据学习到更好的关联。

随后的三篇文章正是用tensorflow实现上述讨论的内容，以此作为零基础实现深层学习的起点。

- [TensorFlow基本用法](#)
- [代码演示LV1](#)
- [代码演示LV2](#)

最后总结一下开篇的问题：

1. 为什么神经网络高效：并行的先验知识使得模型可用线性级数量的样本学习指数级数量的变体
2. 学习的本质是什么：将变体拆分成因素和知识（Disentangle Factors of Variation）
3. 稀疏表达：一个矩阵被拆分成了稀疏表达和字典。
4. one hot vector：将因素用不同维度分开，避免彼此纠缠。
  - i. 为什么深层神经网络比浅层神经网络更高效：迭代组成的先验知识使得样本可用于

帮助训练其他共用同样底层结构的样本。

- ii. 神经网络在什么问题上不具备优势：不满足并行与迭代先验的任务
- 5. 非函数：需要想办法将问题转化。
- 6. 非迭代：该层状态不是由上层状态构成的任务（如：很深的CNN因为有max pooling，信息会逐渐丢失。而residual network再次使得迭代的先验满足）

到此我们讨论完了神经网络最基础的，也是最重要的知识。在实际应用中仍会遇到很多问题（尤其是神经网络对noise的克服更加巧妙）。随后YJango会再和大家一起分析过深后会产生什么效果，并一步步引出设计神经网络的本质。

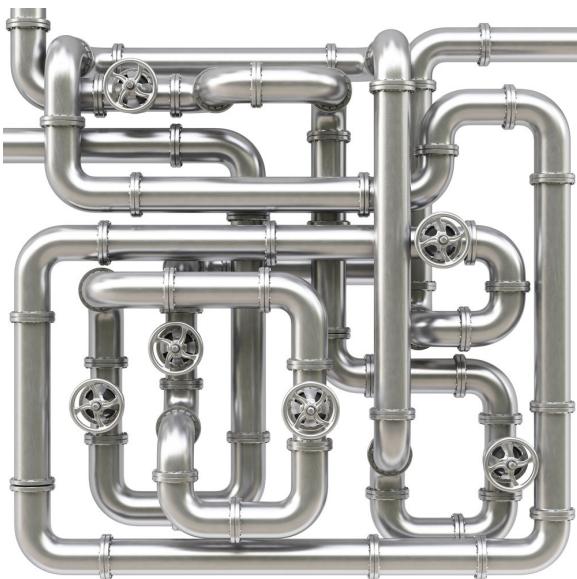
# TensorFlow基本用法

从这里开始，我们会用神经网络验证之前讨论的内容。用的工具是目前主流的TensorFlow。用tensorflow这样工具的原因是：它允许我们用计算图（Computational Graphs）的方式建立网络。同时又可以非常方便的对网络进行操作。下面就是对计算图的直观讲解。

- 注：看完这部分内容的人，可以紧接着实现一个神经网络：[代码演示LV1](#)。

## 比喻说明：

- 结构：而计算图所建立的只是一个网络框架。在编程时，并不会有实际值出现在框架中。所有权重和偏移都是框架中的一部分，初始时至少给定初始值才能形成框架。因此需要initialization初始化。
- 比喻：计算图就是一个管道。编写网络就是搭建一个管道结构。在投入实际使用前，不会有任何液体进入管道。而神经网络中的权重和偏移就是管道中的阀门，可以控制液体的流动强弱和方向。在神经网络的训练中，阀门会根据数据进行自我调节、更新。但是使用之前至少要给所有阀门一个初始的状态才能形成结构。用计算图又允许我们可以从任意一个节点处取出液体。



## 用法说明：

请类比管道构建来理解计算图的用法

构造阶段（**construction phase**）：组装计算图（管道）

- 计算图（**graph**）：要组装的结构。由许多操作组成。

- 操作（ops）：接受（流入）零个或多个输入（液体），返回（流出）零个或多个输出。
- 数据类型：主要分为张量（tensor）、变量（variable）和常量（constant）
  - 张量：多维array或list（管道中的液体）
    - 创建语句：`tensor_name=tf.placeholder(type, shape, name)`
  - 变量：在同一时刻对图中所有其他操作都保持静态的数据（管道中的阀门）
    - 创建语句：`name_variable = tf.Variable(value, name)`
    - 初始化语句：

```
#个别变量
init_op=variable.initializer()
#所有变量
init_op=tf.initialize_all_variables()
#注意：init_op的类型是操作（ops），加载之前并不执行
```

- 更新语句：`update_op=tf.assign(variable to be updated, new_value)`
- 常量：无需初始化的变量
  - 创建语句：`name_constant=tf.constant(value)`

### 执行阶段（execution phase）：使用计算图（获取液体）

- 会话：执行（launch）构建的计算图。可选择执行设备：单个电脑的CPU、GPU，或电脑分布式甚至手机。
  - 创建语句：

```
#常规
sess = tf.Session()
#交互
sess = tf.InteractiveSession()
#交互方式可用tensor.eval()获取值，ops.run()执行操作
#关闭
sess.close()
```

- 执行操作：使用创建的会话执行操作
  - 执行语句：`sess.run(op)`
  - 送值（feed）：输入操作的输入值（输入液体）
    - 语句：`sess.run([output], feed_dict={input1:value1, input2:value2})`
  - 取值（fetch）：获取操作的输出值（得到液体）
    - 语句：

```
#单值获取
sess.run(one op)
#多值获取
sess.run([a list of ops])
```

更多内容参考[官网文档](#)



# 代码演示LV1

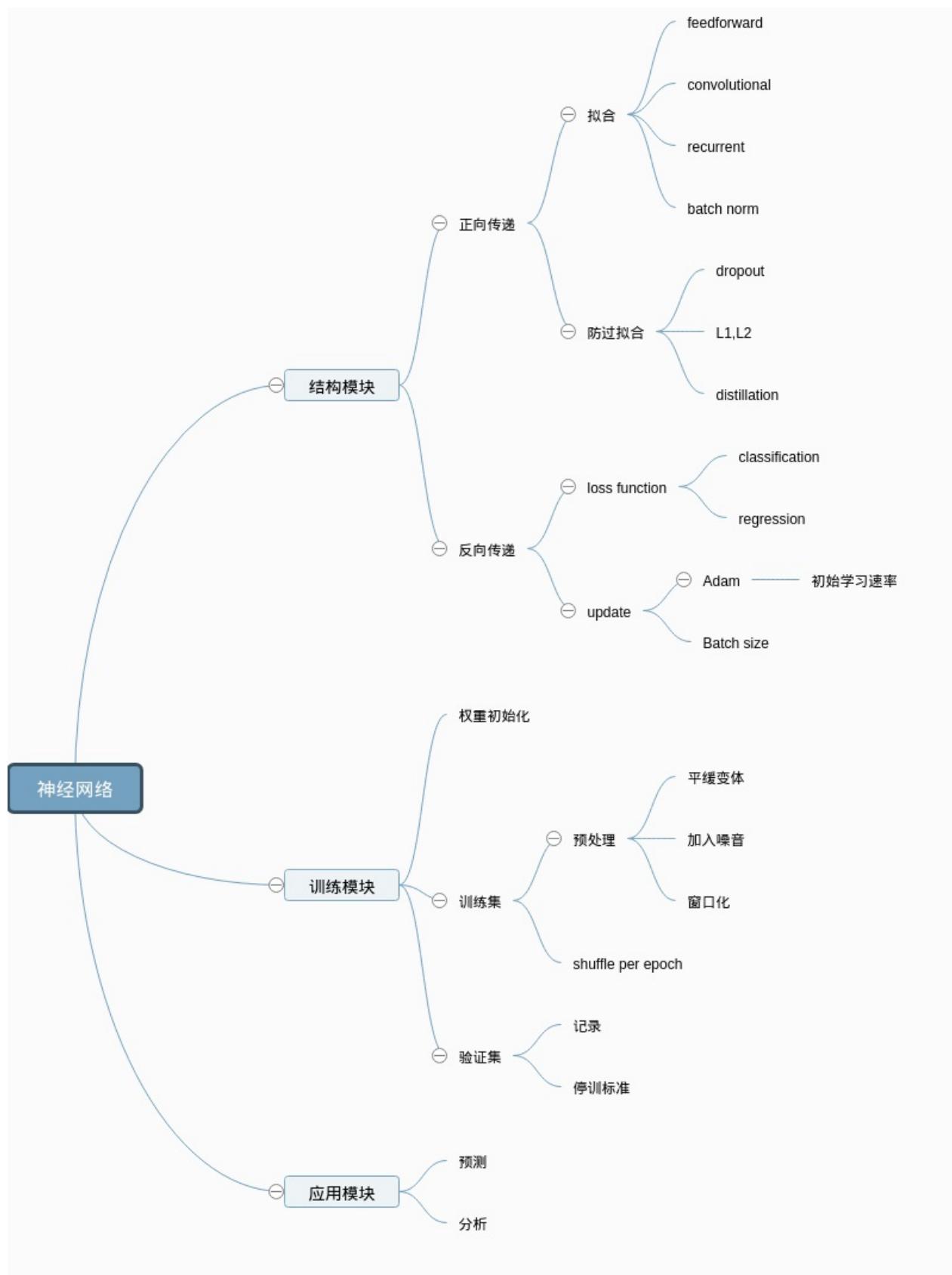
现在就用[TensorFlow](#)来演示[深层神经网络](#)中讨论的XOR门问题（可以换成任何问题，并用相同方法来解决）。下面的代码只是没有任何枝叶的核心内容。全部代码在[github](#)上。

## 准备工作

- ubuntu系统
- 安装工具

## 模块分布

训练神经网络主要就三大模块。掌握模块核心任务后再学习分支内容，中途可任意增添，并且以后还会不断发展。我不希望读者在该篇展开过深。只要对神经网络有一个大概的感觉即可。如下图所示，先抓住三大模块。越靠右边的内容，越不要上来就展开。



## 网络说明

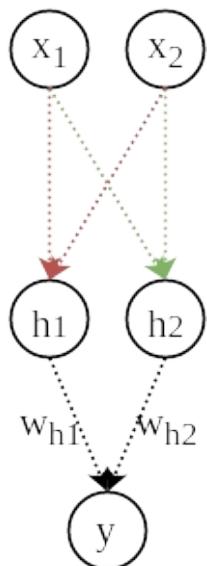
- 打开jupyter notebook (安装完anaconda直接就有)

- new->python
- 引入库：

```
# tensorflow引入，并重命名为tf
import tensorflow as tf

# 矩阵操作库
import numpy as np
```

- 网络结构：2维输入 --> 2维隐藏层 --> 1维输出



- 结构表达式：

- 正向传递： $y = M(x) = \text{relu}(W_h \cdot \text{relu}(W_x \cdot x + b_x) + b_h)$  (1)

- $y$ 用于表达随机变量 $Y$ 的值， $x$ 表示随机变量 $X$ 的值， $M(x)$ 是我们的神经网络模型，等号右侧是具体的表达。

$$\text{loss} = 1/2 \cdot \sum_i (y_i - t_i)^2$$

- 损失函数：
  - 该 $\text{loss}$ 就是比较 $y$ 和 $t$ 中所有值的差别。

## 核心功能

### 计算图说明：

请先阅读：[TensorFlow基本用法](#)，这里解释了：

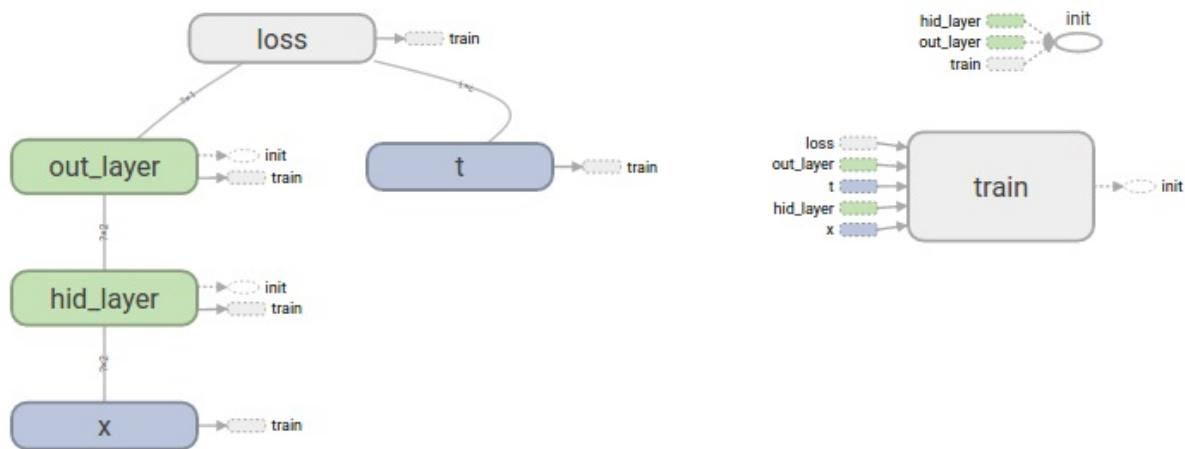
- 为什么使用tensorflow、theano等这样的工具。
- 计算图是什么
- TensorFlow基本用法

TensorFlow所建立的只是一个网络框架。在编程时，并不会有任何实际值出现在框架中。所有权重和偏移都是框架中的一部分，初始时至少给定初始值才能形成框架。因此需要 initialization 初始化。

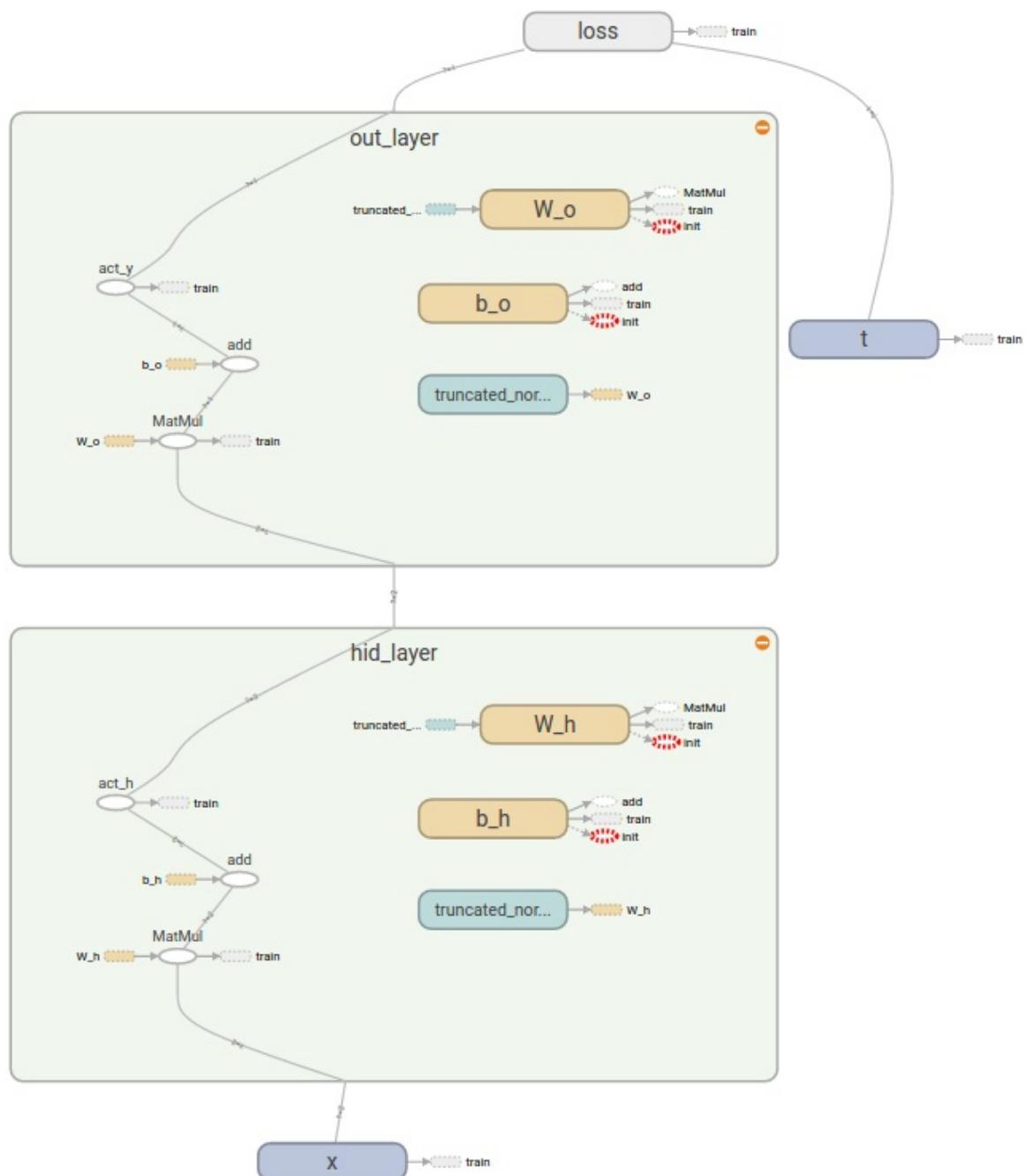
## 实例说明：

以下是利用 [tensorboard](#) 生成的我们想要建立的学习 XOR gate 的网络可视化结构图。

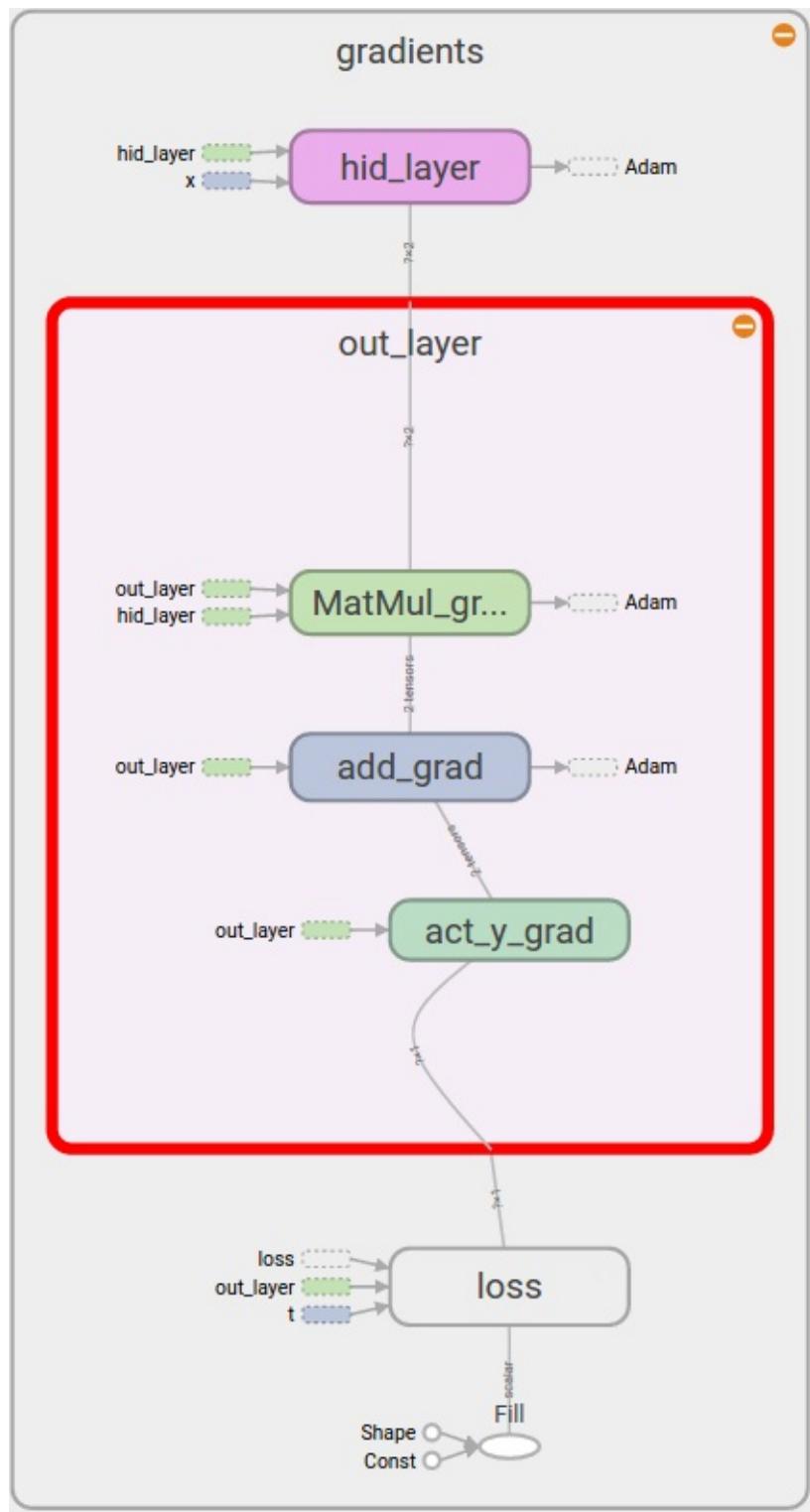
- 整体结构：左侧的图表示网络结构。绿色方框表示操作，也叫作层（layer）。该结构中，输入  $x$  经过 hid\_layer 算出隐藏层的值  $h$ ，再传递给 out\_layer，计算出预测值  $y$ ，随后与真实值  $t$  进行比较，算出损失  $loss$ ，再从  $loss$  反向求导得出梯度后对每一层的  $W$  和  $b$  进行更新。



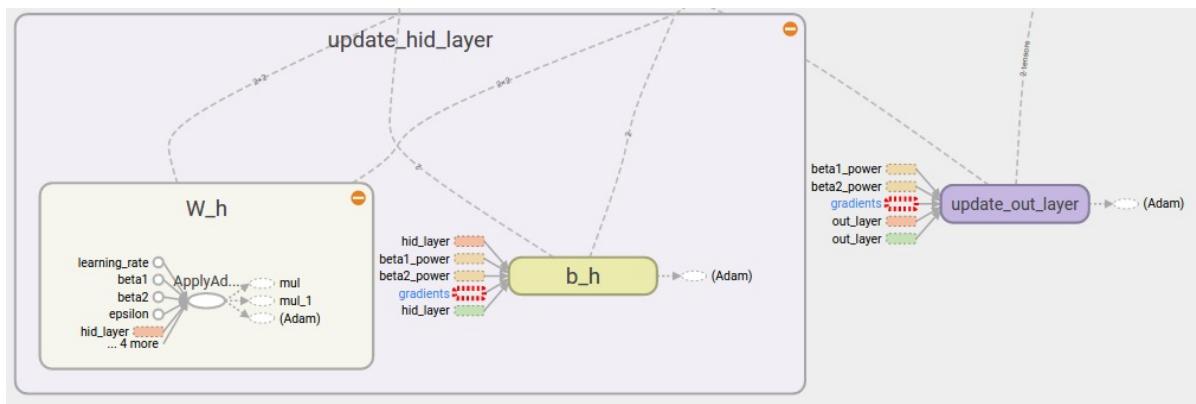
- 正向传递：如果放大 hid\_layer 内部，从下向上，会看到  $W_h$  先用 truncated\_normal 的方法进行了初始化，随后与输入  $x$  进行矩阵相乘，加上  $b_h$ ，又经过了 activation 后，送给了用于计算  $y$  的 out\_layer 中。而  $y$  的计算方式和  $h$  完全一致，但用的是不同的权重  $W$  和偏移  $b$ 。最后将算出的预测值  $y$  与真实值  $t$  一同求出  $loss$



- 反向传递：如果放大train的内部，再放大内部中的gradients，就可以看到框架是从 $loss$ 开始一步步反向求得各个层中 $W$ 和 $b$ 的梯度的。



- 权重更新：求出的各个层  $W$  和  $b$  的梯度，将会被用于更新对应的  $W$  和  $b$ ，备用 learning rate 控制一次更新多大。（`beta1_power` 和 `beta2_power` 是 Adam 更新方法中的参数，目前只需要知道权重更新的核心是各自对应的梯度。）



## 代码实现

### 网络模块

- 变量定义：定义节点数和学习速率

```
# 网络结构：2维输入 -> 2维隐藏层 -> 1维输出
# 学习速率 (learning rate) : 0.0001

D_input = 2
D_label = 1
D_hidden = 2
lr = 1e-4
```

正向传递：到预测值为止的网络框架。在预测时，只拿 $y$ 的输出。

- 容器：用`tf.placeholder`创建输入值和真实值的容器，编程过程中始终是个空的，只有加载到`sess`中才会放入具体值。这种容器便是存放`tensor`的数据类型。

```
x = tf.placeholder(tf.float32, [None, D_input], name="x")
t = tf.placeholder(tf.float32, [None, D_label], name="t")
```

- 精度：如果是用GPU训练，浮点精度要低于32bit，由第一个参数`tf.float32`定义。
- 矩阵形状：输入输出的容器都是矩阵。为的是可以进行mini-batch一次算多个样本的平均梯度来训练。`None`意味着样本数可随意改变。
- 命名：控制`tensorboard`生成图中的名字，也会方便debug。

- 隐藏层：

```
# 初始化W
w_h1 = tf.Variable(tf.truncated_normal([D_input, D_hidden], stddev=0.1), name="w_h")
# 初始化b
b_h1 = tf.Variable(tf.constant(0.1, shape=[D_hidden]), name="b_h")
# 计算Wx+b
pre_act_h1 = tf.matmul(x, w_h1) + b_h1
# 计算a(Wx+b)
act_h1 = tf.nn.relu(pre_act_h1, name='act_h')
```

- 变量：tensorflow中的变量tf.Variable是用于定义在训练过程中可以更新的值。权重W和偏移b正符合该特点。
- 初始化：合理的初始化会给网络一个比较好的训练起点，帮助逃脱局部极小值（或鞍点）。详细请回顾[梯度下降训练法](#)。tf.truncated\_normal([D\_input, D\_hidden], stddev=0.1)是初始化的一种方法（还有很多种），其中[imcoing\_dim, outputting\_dim]是矩阵的形状，前后参数的意义是进入该层的维度（节点个数）和输出该层的维度。stddev=是用于初始化的标准差。
- 矩阵乘法：tf.matmul(x, W\_h1)用于计算矩阵乘法
- 激活函数：除了tf.nn.relu()还有tf.nn.tanh(), tf.nn.sigmoid()
- 输出层：同隐藏层的计算方式一致，但输出层的“输入”是隐藏层的“输出”。

```
w_o = tf.Variable(tf.truncated_normal([D_hidden, D_label], stddev=0.1), name="w_o")
b_o = tf.Variable(tf.constant(0.1, shape=[D_label]), name="b_o")
pre_act_o = tf.matmul(act_h1, w_o) + b_o
y = tf.nn.relu(pre_act_o, name='act_y')
```

反向传递：计算误差值来更新网络权重的结构。

- 损失函数：定义想要不断缩小的损失函数。

```
loss = tf.reduce_mean((self.output-self.labels)**2)
```

- 更新方法：选择想要用于更新权重的训练方法和每次更新步伐（lr），除tf.train.AdamOptimizer外还有tf.train.RMSPropOptimizer等。默认推荐AdamOptimizer。

```
train_step = tf.train.AdamOptimizer(lr).minimize(loss)
```

- 学习速率：AdamOptimizer(lr)的括号内放入学习速率。
- 优化目标：minimize(loss)的括号内放入想要缩小的值。

准备数据：用**numpy.array**格式准备**XOR**的输入和输出，即训练数据

```
#X和Y是4个数据的矩阵，X[i]和Y[i]的值始终对应。
X=[[0,0],[0,1],[1,0],[1,1]]
Y=[[0],[1],[1],[0]]
X=np.array(X).astype('int16')
Y=np.array(Y).astype('int16')
```

- 数据类型：用python使用tensorflow时，输入到网络中的训练数据需要以np.array的类型存在。并且要限制dtype为32bit以下。变量后跟着“.astype('float32')”总可以满足要求。

加载训练：将建好的网络加载到session中执行操作。

```
#创建session
sess = tf.InteractiveSession()
#初始化权重
tf.initialize_all_variables().run()
```

- 创建方式：sess = tf.InteractiveSession()是比较方便的创建方法。也有sess = tf.Session()方式，但该方式无法使用tensor.eval()快速取值等功能。
- 初始化：虽然在先前定义权重 $W$ 时选择了初始化方式，但要实际执行该操作需要将操作（op）加载到session中。
- 训练网络：

```
T=10000 #训练几次
for i in range(T):
    sess.run(train_step, feed_dict={x:X, t:Y})
```

- 说明：sess.run(,)接受的两个参数。前一个参数的类型是list，表示所有想要执行的操作（op）或者想要获取的值。而后一个参数会需要执行第一个参数时相关placeholder的值。并且以feed\_dict=字典的方式赋值。{x:X,t:Y}中的x是key名，对应着placeholder，而冒号后的是计算时的具体输入值。
- **GD (Gradient Descent)**：X和Y是4组不同的训练数据。上面将所有数据输入到网络，算出平均梯度来更新一次网络的方法叫做GD。效率很低，也容易卡在局部极小值，但更新方向稳定。
- **SGD (Gradient Descent)**：一次只输入一个训练数据到网络，算出梯度来更新一次网络的方法叫做SGD。效率高，适合大规模学习任务，容易挣脱局部极小值（或鞍点），但更新方向不稳定。代码如下：

```
T=10000 #训练几epoch
for i in range(T):
    for j in range(X.shape[0]): #X.shape[0]表示样本个数
        sess.run(train_step, feed_dict={x:X[j], t:Y[j]})
```

- **batch-GD**：这是上面两个方法的折中方式。每次计算部分数据的平均梯度来更新权重。部分数据的数量大小叫做batch\_size，对训练效果有影响。一般10个以下的也叫mini-batch-GD。代码如下：

```
T=10000 #训练几epoch
b_idx=0 #batch计数
b_size=2 #batch大小
for i in range(T):
    while batch_idx<=X.shape[0]:
        sess.run(train_step,feed_dict={x:X[b_idx:b_idx+b_size],t:Y[b_idx:b_idx+b_size]})
        b_idx+=b_size #更新batch计数
```

- **shuffle**：SGD和batch-GD由于只用到了部分数据。若数据都以相同顺序进入网络会使得随后的epoch影响很小。**shuffle**是用于打乱数据在矩阵中的排列顺序，提高后续epoch的训练效果。代码如下：

```
#shuffle函数
def shufflelists(lists): #多个序列以相同顺序打乱
    ri=np.random.permutation(len(lists[1]))
    out=[]
    for l in lists:
        out.append(l[ri])
    return out
#训练网络
T=10000 #训练几epoch
b_idx=0 #batch计数
b_size=2 #batch大小
for i in range(T):
    #每次epoch都打乱顺序
    X,Y = shufflelists([X,Y])
    while batch_idx<=X.shape[0]:
        sess.run(train_step,feed_dict={x:X[b_idx:b_idx+b_size],t:Y[b_idx:b_idx+b_size]})
        b_idx+=b_size #更新batch计数
```

- 预测：

```
#计算预测值
sess.run(y,feed_dict={x:X})
#输出：已训练100000次
array([[ 0.],
       [ 1.],
       [ 1.],
       [ 0.]], dtype=float32)
```

- 说明：预测时与目标值 $t$ 无关，只需要将 $x$ 输入到网络中即可预测该 $x$ 对应哪个 $y$ ，而

预测好坏取决于训练的网络本身。

- 隐藏层：

```
#查看隐藏层的输出
sess.run(act_h1, feed_dict={x:X})
#输出：已训练100000次
array([[ 1.10531139,  1.00508392],
       [ 0.55236477,  0.        ],
       [ 0.55236477,  0.        ],
       [ 0.        ,  0.        ]], dtype=float32)
```

- 说明：`act_h1`是隐藏层的输出。可以看到隐藏层前有4个变体，而经过隐藏层后，只有3个变体。
- 查看其他值：用户可以用`sess.run()`获取网络框架中的任何值。比如查看隐藏层的权重 $W_h$ 和 $W_o$ ：

```
sess.run([W_h,W_o])
```

- 多值输出：可将多个想要的值放入一个list中传递给`sess.run()`实现多值输出。
- 变量无需容器： $W_h$ 和 $W_o$ 的获取并未需要输入`placeholder`。因为训练好后，它们的值与`placeholder`的值无关，所以不需要输出。

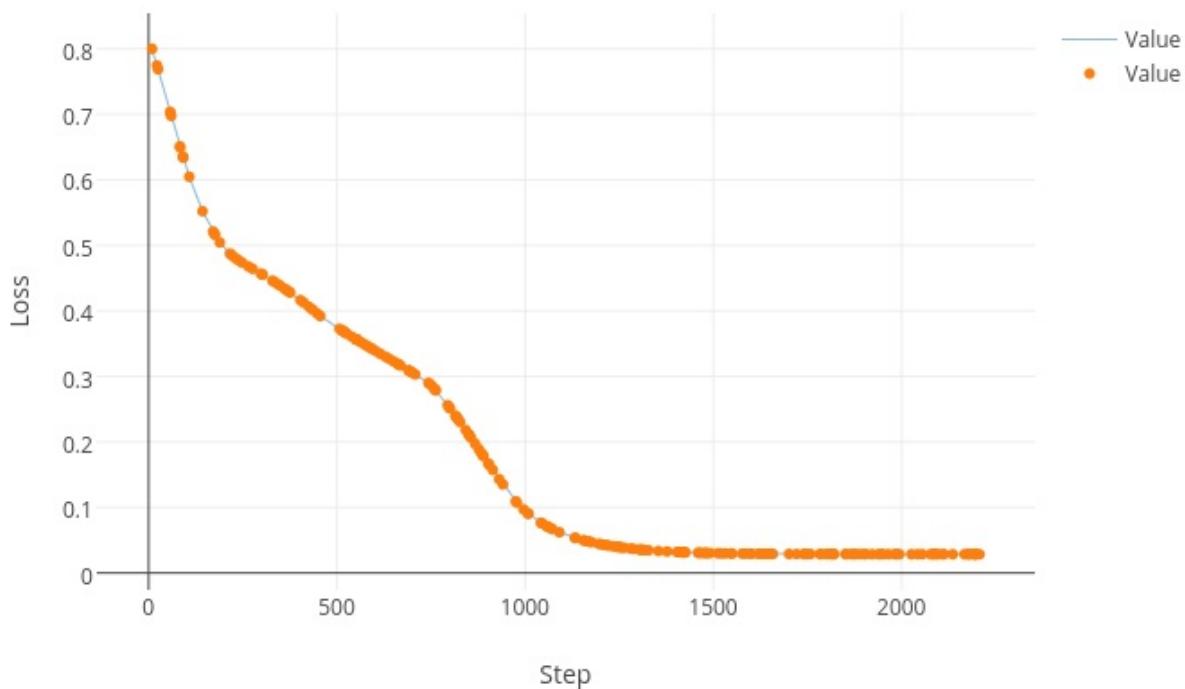
---

上面已经演示了最基本，也是核心代码内容。虽然简陋，但完全可以完成任务。

深层神经网络就是简单的在此基础上增加更多的隐藏层。深层神经网络的实现非常简单，并没有人们想的那么高大上。

然而深层学习的难点不在网络实现上，而在对数据的分析和反馈、当遇到无法拟合的情况该如何处理。为了能够记录和分析结果，我们会想知道误差下降的过程，网络在某时训练的如何，比如下图：可以看出来训练在第30000次后的误差就几乎为0了。但上述的代码可以让我们完成训练却没有这些分析功能。下章是关于记录、分析、代码重用性等功能所需要的“枝叶”。

Training loss plot



# 代码演示LV2

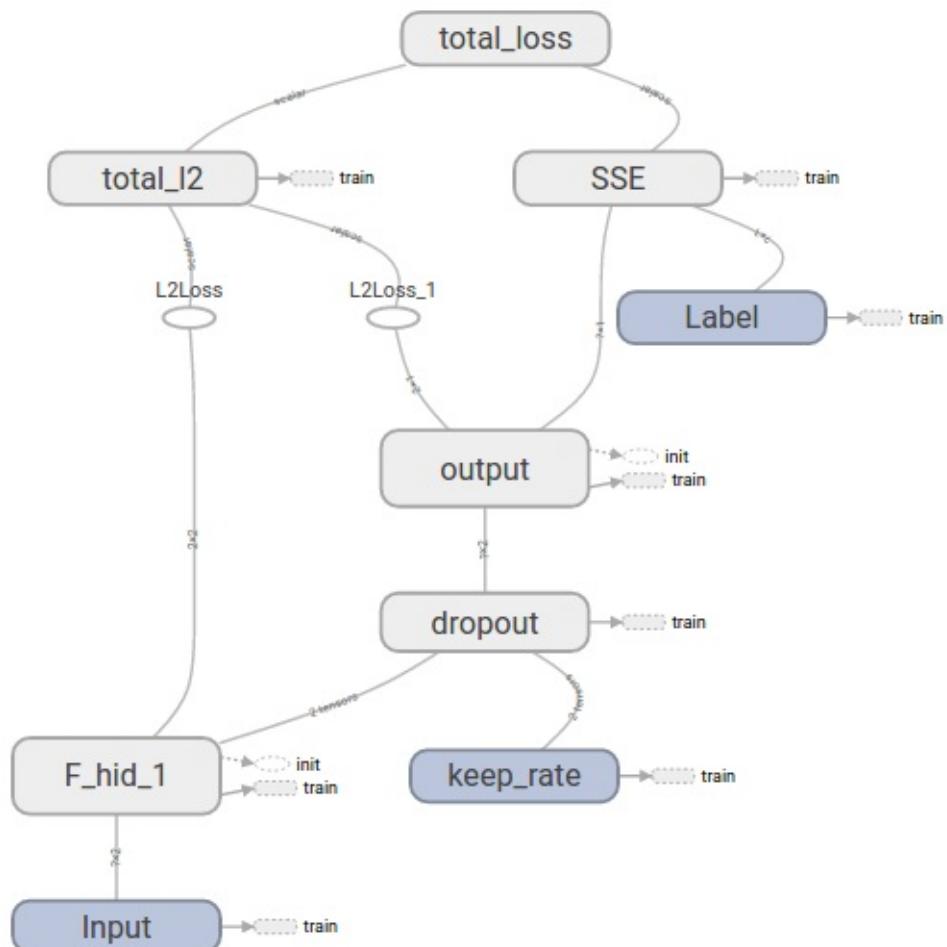
依旧是与[代码演示](#)相同的任务（[为何深层学习中的讲解内容](#)），但是这次要将程序增加一些“枝叶”，好用于记录分析。全部代码在[github](#)上。其实真正使用的时候并不需要像下面那样自己编写。可以选择利用[TensorLayer](#)、[TFLearn](#)这样写在[TensorFlow](#)之上的工具包。下面的操作也就是[TensorLayer](#)等所做的。

但使用该便利工具的前提是已了解底层内容，在随后的使用中可以自由更改网络底层结构，而无需依赖于[TensorLayer](#)等工具。原因在于神经网络仍在不断发展，每天都会有很多新的方法和结构被提出。这些都需要能够改造网络结构的能力。

更重要的是，对底层的了解可以允许我们迅速发现训练过程中的问题，并及时做出调整。

## 构建模块

- 计算结构图：这次加入了L2和dropout。最终的loss是由L2\_loss和Sum of Squared Error（就是上篇的loss）相加组成的。



- **step 0:** 为方便以后使用，将所有操作都建到一个类中。并且建立初始化函数：

```

class FNN(object):
    """Build a general FeedForward neural network
    Parameters
    -----
    learning_rate : float
    drop_out : float
    Layers : list
        The number of layers
    N_hidden : list
        The numbers of nodes in layers
    D_input : int
        Input dimension
    D_label : int
        Label dimension
    Task_type : string
        'regression' or 'classification'
    L2_lambda : float

    """
    def __init__(self, learning_rate, drop_keep, Layers, N_hidden,
                 D_input, D_label, Task_type='regression', L2_lambda=0.0):

        # 全部共有属性
        self.learning_rate = learning_rate
        self.drop_keep = drop_keep
        self.Layers = Layers
        self.N_hidden = N_hidden
        self.D_input = D_input
        self.D_label = D_label
        # 类型控制loss函数的选择
        self.Task_type = Task_type
        # L2 regularization的惩罚强弱，过高会使得输出都拉向0
        self.L2_lambda = L2_lambda
        # 用于存放所累积的每层l2 regularization
        self.l2_penalty = tf.constant(0.0)

```

- **解释 0：**该类的目的是建立一个通用的feedforward神经网络生成类。在以后的使用中只需要调节参数就可以改变网络内容，而无需重新编写。
- **step 1:** 建立 输入 $x$ 和输出 $y$ 的容器，但是这次加入name\_scope，为的是像上图一样生成一个结构图：

```

# 用于生成tensorflow缩放图的，括号里起名字
with tf.name_scope('Input'):
    self.inputs = tf.placeholder(tf.float32, [None, D_input], name="inputs")
with tf.name_scope('Label'):
    self.labels = tf.placeholder(tf.float32, [None, D_label], name="labels")
with tf.name_scope('keep_rate'):
    self.drop_keep_rate = tf.placeholder(tf.float32, name="dropout_keep")

# 初始化的时候直接生成，build方法是后面会建立的
self.build('F')

```

- 解释 1：“with tf.name\_scope('keep\_rate')：“下的内容都会被表示成一个方框。可以打开查看详细内容。
- step 2:** 由于所有层都需要该初始化，所以建函数方便以后直接调用。

```

def weight_init(self, shape):
    # shape : list [in_dim, out_dim]
    # can change initialization here
    initial = tf.truncated_normal(shape, stddev=0.1)
    return tf.Variable(initial)

def bias_init(self, shape):
    # can change initialization here
    initial = tf.constant(0.1, shape=shape)
    return tf.Variable(initial)

```

- 解释 2：这里的初始化方式是固定的。可以在函数中另加入一个input来控制初始化方式。上面的shape接受的都是list。
- step 3:** 用于记录和统计训练中的数据.

```

def variable_summaries(self, var, name):
    with tf.name_scope(name+'_summaries'):
        mean = tf.reduce_mean(var)
        tf.scalar_summary('mean/' + name, mean)
    with tf.name_scope(name+'_stddev'):
        stddev = tf.sqrt(tf.reduce_mean(tf.square(var - mean)))
        # 记录每次训练后变量的数值变化
        tf.scalar_summary('_stddev/' + name, stddev)
        tf.scalar_summary('_max/' + name, tf.reduce_max(var))
        tf.scalar_summary('_min/' + name, tf.reduce_min(var))
        tf.histogram_summary(name, var)

```

- 解释 3：上面的函数就是随着训练记录一个变量的最大值、最小值、方差、的变化，以及直方图。

- **step 4:** 构建用于建立每层的函数：

```
def layer(self, in_tensor, in_dim, out_dim, layer_name, act=tf.nn.relu):
    with tf.name_scope(layer_name):
        with tf.name_scope(layer_name+'_weights'):
            # 用所建立的weight_init函数进行初始化。
            weights = self.weight_init([in_dim, out_dim])
            # 存放着每一个权重W
            self.W.append(weights)
            # 对权重进行统计
            self.variable_summaries(weights, layer_name + '/weights')
        with tf.name_scope(layer_name+'_biases'):
            biases = self.bias_init([out_dim])
            self.variable_summaries(biases, layer_name + '/biases')
        with tf.name_scope(layer_name+'_Wx_plus_b'):
            # 计算Wx+b
            pre_activate = tf.matmul(in_tensor, weights) + biases
            # 记录直方图
            tf.histogram_summary(layer_name + '/pre_activations', pre_activate)
        # 计算a(Wx+b)
        activations = act(pre_activate, name='activation')
        tf.histogram_summary(layer_name + '/activations', activations)
    # 最终返回该层的输出，以及权重W的L2
    return activations, tf.nn.l2_loss(weights)
```

- **解释 4：**该函数接受输出tensor，以及该层的信息，返回处理后的输出。而该输出又可以做为下一层的输入，以此不断反复来叠加层数。这里的self.W.append(weights)是我随意添加的，用于输出每一层的权重。self.W本身会在使用该函数前声明。（selb.b同理）

- **step 5:** 创建dropout层的函数：

```
def drop_layer(self, in_tensor):
    #tf.scalar_summary('dropout_keep', self.drop_keep_rate)
    dropped = tf.nn.dropout(in_tensor, self.drop_keep_rate)
    return dropped
```

- **解释 5：**将该内容额外编写出来是想让读者知道，尽管这个称为“层”，但“层”的操作也可以很简单。叫做层的操作一般都是那些可以反复叠加利用的tensor处理方式。
- **step 6:** 构建网络，看起来内容很多，实际上核心内容就是上一节所描述的：

```
def build(self, prefix):
    # 建立网络
    # incoming也代表当前tensor的流动位置
    incoming = self.inputs
    # 如果没有隐藏层
    if self.Layers!=0:
        layer_nodes = [self.D_input] + self.N_hidden
```

```

else:
    layer_nodes = [self.D_input]

    # hid_layers用于存储所有隐藏层的输出
    self.hid_layers=[]
    # W用于存储所有层的权重
    self.W=[]
    # b用于存储所有层的偏移
    self.b=[]
    # total_l2用于存储所有层的L2
    self.total_l2=[]
    # drop存储dropout后的输出
    self.drop=[]

    # 开始叠加隐藏层。这跟千层饼没什么区别。
    for l in range(self.Layers):
        # 使用刚才编写的函数来建立层，并更新incoming的位置
        incoming,l2_loss= self.layer(incoming,layer_nodes[1],layer_nodes[l+1],prefix+'_hid_'+str(l+1),act=tf.nn.relu)
        # 累计l2
        self.total_l2.append(l2_loss)
        # 输出一些信息，让我们知道网络在建造中做了什么
        print('Add dense layer: relu with drop_keep:%s' %self.drop_keep)
        print('    %sD --> %sD' %(layer_nodes[1],layer_nodes[l+1]))
        # 存储所有隐藏层的输出
        self.hid_layers.append(incoming)
        # 加入dropout层
        incoming = self.drop_layer(incoming)
        # 存储所有dropout后的输出
        self.drop.append(incoming)

    # 输出层的建立。输出层需要特别对待的原因是输出层的activation function要根据任务来变。
    # 回归任务的话，下面用的是tf.identity，也就是没有activation function
    if self.Task_type=='regression':
        out_act=tf.identity
    else:
        # 分类任务使用softmax来拟合概率
        out_act=tf.nn.softmax
    self.output,l2_loss= self.layer(incoming,layer_nodes[-1],self.D_label, layer_name='output',act=out_act)
    self.total_l2.append(l2_loss)
    print('Add output layer: linear')
    print('    %sD --> %sD' %(layer_nodes[-1],self.D_label))

    # l2 loss的缩放图
    with tf.name_scope('total_l2'):
        for l2 in self.total_l2:
            self.l2_penalty+=l2
        tf.scalar_summary('l2_penalty', self.l2_penalty)

    # 不同任务的loss
    # 若为回归，则loss是用于判断所有预测值和实际值差别的函数。
    if self.Task_type=='regression':

```

```

with tf.name_scope('SSE'):
    self.loss=tf.reduce_mean((self.output-self.labels)**2)
    tf.scalar_summary('loss', self.loss)
else:
    # 若为分类，cross entropy的loss function
    entropy = tf.nn.softmax_cross_entropy_with_logits(self.output, self.labels)
    with tf.name_scope('cross entropy'):
        self.loss = tf.reduce_mean(entropy)
        tf.scalar_summary('loss', self.loss)
    with tf.name_scope('accuracy'):
        correct_prediction = tf.equal(tf.argmax(self.output, 1), tf.argmax(self.labels, 1))
        self.accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
        tf.scalar_summary('accuracy', self.accuracy)
# 整合所有loss，形成最终loss
with tf.name_scope('total_loss'):
    self.total_loss=self.loss + self.l2_penalty*self.L2_lambda
    tf.scalar_summary('total_loss', self.total_loss)

# 训练操作
with tf.name_scope('train'):
    self.train_step = tf.train.AdamOptimizer(self.learning_rate).minimize(self.total_loss)

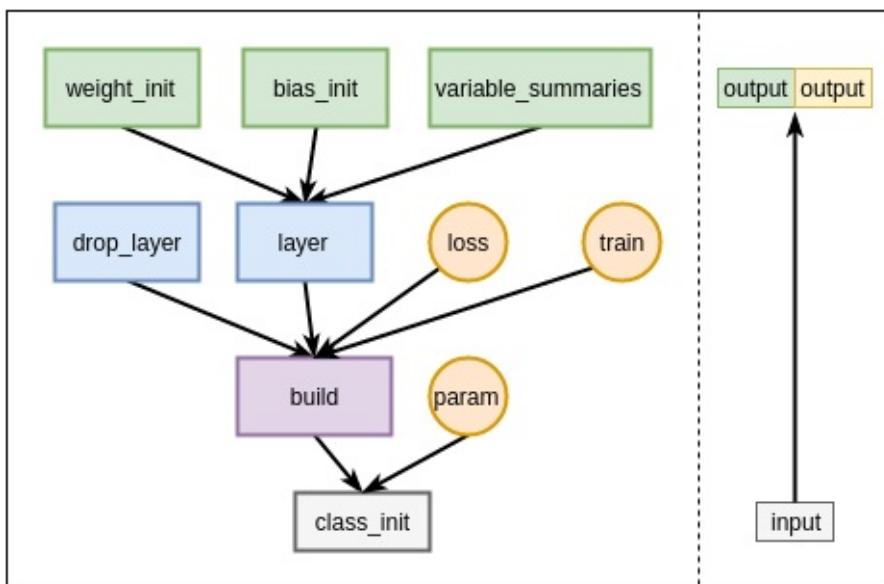
```

## 代码学习

这是关于如何记住上面的内容的说明。YJango在《超智能体》的最初就说过，对智能的研究最终要返回到：将智能的本质规律应用在学习，教育，社会发展中去。既然在[为何深层学习](#)中已经知道了深层学习的优点，就应该将这种特点应用到我们自身的学习当中去。

不同的层和所搭建的网络的本质都是变体，可以根据任务需求随意变化。学习是要把这些变体拆分成（因素+关联）的方式。

- 实例：上述的结构中，关系如下图：



- 因素：weight\_init、bias\_init、variable\_summaries、loss、train、parameters
  - 绿色：代表被编写成了function。
  - 黄色：代表直接以固定的形式写成代码。
- 变体：最终的网络结构是这些因素有结构性的迭代组合：layer是由绿色方块组成，而build的组成因素又有layer。
- 关联：这些黑线才是读者应该学习的对象：关联。

用变体训练，但不要学习变体，学习的是关联。

到此请再次理解深层神经网络的两大固有先验知识：并行、迭代。因为weight\_init和bias\_init的工作相互独立、并不干扰，我们才可以把他们拆分成因素。而这里的weight\_init是truncated\_normal方式的初始化，如果我想构建一个用其他初始化方法的层，那么被固定的weight\_init就无法被使用。我们不得不重新写一个初始化方法。所以深层的前提是下一级的内容可由上一级构成。

注：我画的网络是固定的，神经网络并不一样。神经网络可以在训练时自由调整，会逐渐形成满足迭代先验的结构。自然数据几乎都可以从迭代先验中获益。

## 使用模块

- **step 0**：数据准备

```

inputs=[[0,0],[0,1],[1,0],[1,1]]
outputs=[0,1,1,0]
X=np.array(inputs).reshape((4,1,2)).astype('int16')
Y=np.array(outputs).reshape((4,1,1)).astype('int16')
  
```

- 解释 0：不一样的地方是`reshape((4,1,1))`一部分，YJango刻意要用这种方式表达的原因是因为在TensorFlow的使用中，常会遇到很多关于tensor的shape不匹配的问题。所以要尽早熟悉`reshape`的指令和作用。`reshape((4,1,2))`表达的意思是，4个有序的1 by 2矩阵（向量）。4表示训练数量。2表示维度。
- step 1**：生成网络实例 ff（可随意取名字）

```
# 生成网络实例
ff=FNN(learning_rate=1e-3, drop_keep=1.0, Layers=1, N_hidden=[2], D_input=2, D_label=1, Task_type='regression', L2_lambda=1e-2)
# 下面是实际输出内容
#
Add dense layer: relu with drop_keep:1.0
2D --> 2D
Add output layer: linear
2D --> 1D
```

- 解释 1：上面是生成一个初始学习速率为0.001，没有dropout（1.0表示全部保留，一个不扔），一个隐藏层（Layers表示隐藏层的个数），输入维度是2，目标维度是1，回归任务，L2的惩罚强度为0.01。生成后，程序会按照事先编写的格式输出一些内容。随后我们就可以用`ff.xxx`的形式来获取ff内的所有属性。

- step 2**：加载会话Session

```
sess = tf.InteractiveSession()
tf.initialize_all_variables().run()
# 用于将所有summary合成一个ops
merged = tf.merge_all_summaries()
# 用于记录训练中内容。前一个参数是记录地址，后一个参数是记录的graph。
train_writer = tf.train.SummaryWriter('log' + '/train', sess.graph)
```

- 解释 2：相比代码演示，这次的新内容只有后两句。用于记录所有统计内容。

## 训练前

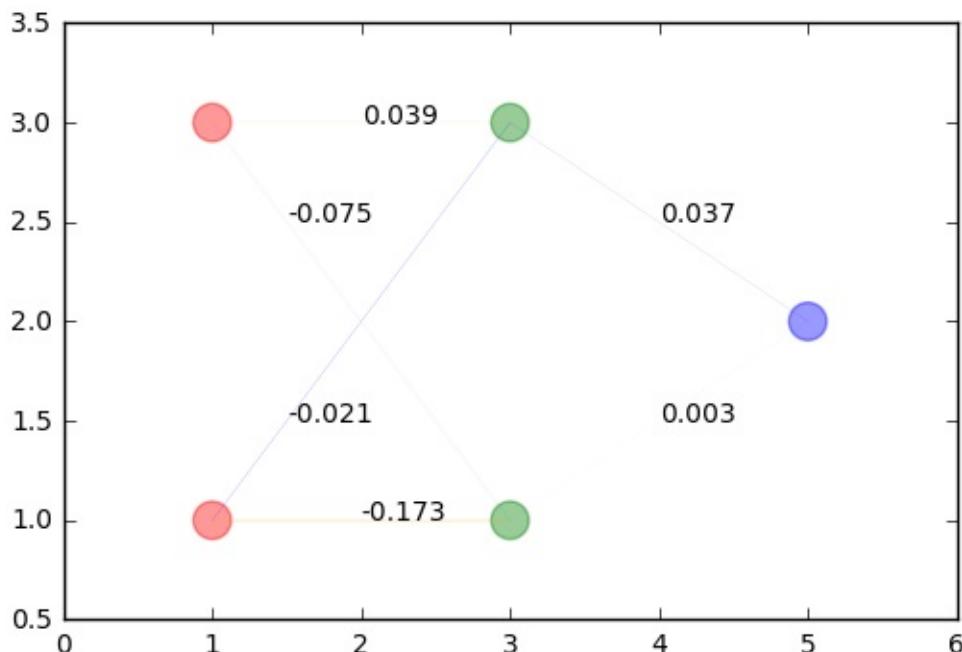
- step 3**：训练前观察一下权重W的值

```

# 获得输出
W0=sess.run(ff.W[0])
W1=sess.run(ff.W[1])
# 显示
print('W_0:\n%s' %sess.run(ff.W[0]))
print('W_1:\n%s' %sess.run(ff.W[1]))
# 实际输出为
W_0:
[[ -0.17334478 -0.0750991 ]
 [-0.02134527  0.03925243]]
W_1:
[[ 0.00257381]
 [ 0.0365728 ]]

```

- 解释 3 :  $ff.W[i]$  表示不同层的  $W$ ， $i$  是从 0 开始计数。为了有个更直观的认识，这里画出了权重和节点间的关系图。YJango 为节省篇幅，还请读者从 [github](#) 中找源码，并且结合先前讲的线性代数中的矩阵乘法，找到每个权重对应的两个节点（在 tensorflow 中，矩阵乘法是  $x \cdot W$  的形式进行的）。红色表示输入层，绿色是隐藏层，而蓝色是输出层。线的粗细表示权重的绝对值大小。



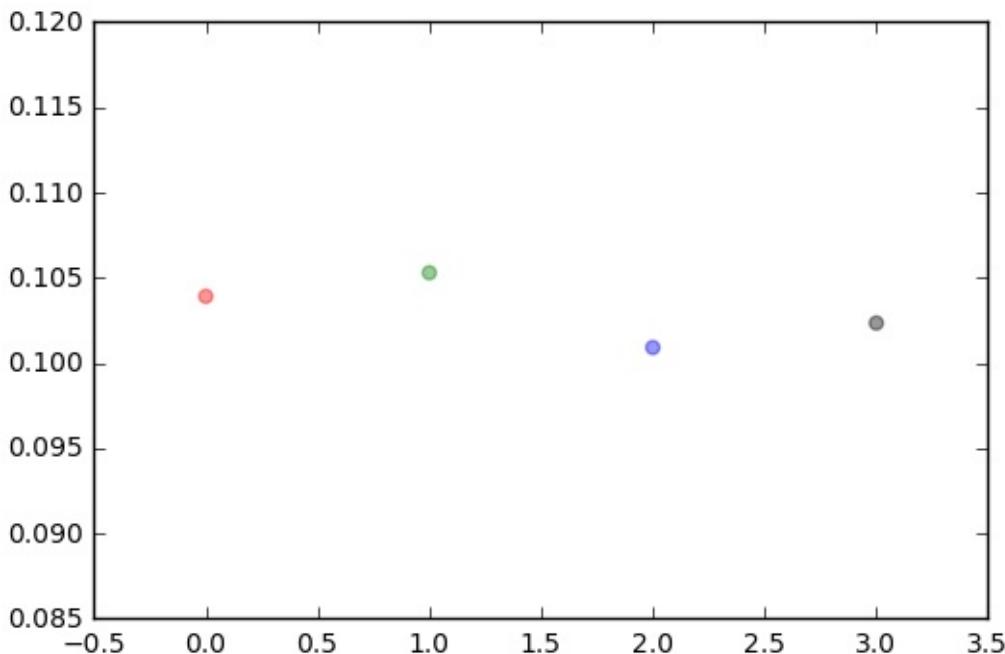
- step 4 : 训练前的输出值 ( $ff.output$  中)

```

# 训练前的输出
pY=sess.run(ff.output,feed_dict={ff.inputs:X.reshape((4,2)),ff.drop_keep_rate:1.0}
)
print(pY)
# 画图 (4个数据循序用红、绿、蓝、黑表示)
plt.scatter([0,1,2,3],pY,color=['red','green','blue','black'],s=25,alpha=0.4,marker='o')
# 实际输出结果：
[[ 0.10391466]
 [ 0.10529529]
 [ 0.1009107 ]
 [ 0.10234627]]

```

- 实际图：可以看到并不是我们想要的结果。



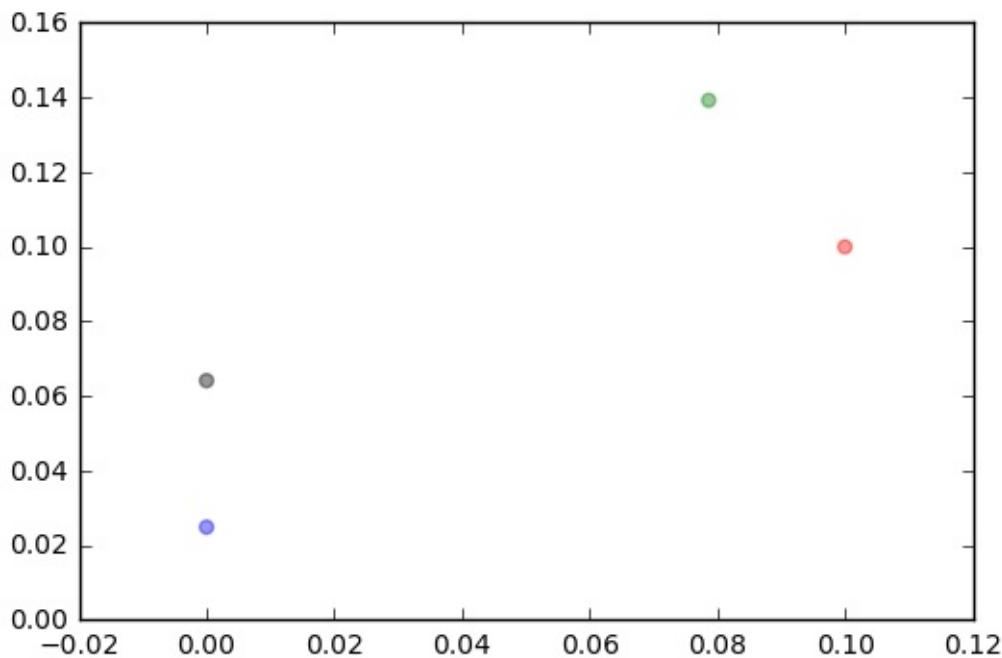
- step 5**：训练前的隐藏层的输出值（`ff.hid_layers[i]`中，`i`对应第一个隐藏层）

```

# 训练前隐藏层的输出
pY=sess.run(ff.hid_layers[0],feed_dict={ff.inputs:X.reshape((4,2)),ff.drop_keep_rate:1.0})
print(pY)
plt.scatter(pY[:,0],pY[:,1],color=['red','green','blue','black'],s=25,alpha=0.4,marker='o')
# 实际输出值：
[[ 0.1          0.1        ]
 [ 0.07865474  0.13925242]
 [ 0.          0.02490091]
 [ 0.          0.06415333]]

```

- 实际图：4个不同值



- **Dropout**：会随机扔掉节点，其余的节点会被乘以 $1/\text{keep\_rate}$ 。原理以后另讲。这里可以先通过调节`keep_rate`来感受隐藏层的输出变化。

```

sess.run(ff.drop[0], feed_dict={ff.inputs:X.reshape((4,2)), ff.labels:Y.reshape((4,1))
)), ff.drop_keep_rate:0.5})
# 当keep_rate : 1时，保持所有节点（就是隐藏层的原有输出）
[[ 0.1          0.1          ]
 [ 0.07865474   0.13925242]
 [ 0.           0.02490091]
 [ 0.           0.06415333]]
# 当keep_rate : 0.5时，每个数据的节点都有一个被扔掉了。剩下的那个节点被乘以1/0.5
[[ 0.2          0.           ]
 [ 0.           0.27850484]
 [ 0.           0.04980182]
 [ 0.           0.           ]]

```

## 训练

- **step 6**：训练，并且记录所有统计内容

```

# 训练并记录
# k表示训练了多少次
k=0.0
# i每增加1，就表示所有的训练数据都被训练完了一次。叫做epoch
for i in range(10000):
    k+=1
    # summary是merged得出的值，即所有统计内容
    summary, _ = sess.run([merged, ff.train_step], feed_dict={ff.inputs:X.reshape((4,2)), ff.labels:Y.reshape((4,1)), ff.drop_keep_rate:1.0})
    # 将统计内容写入指定log文件中
    train_writer.add_summary(summary, k)

```

- 解释 6：这里默认训练10000此。sess.run([merged,ff.train\_step],feed....)中执行了两个操作。ff.train\_step是训练操作。由于是ff的属性，记得加ff.

## 训练后

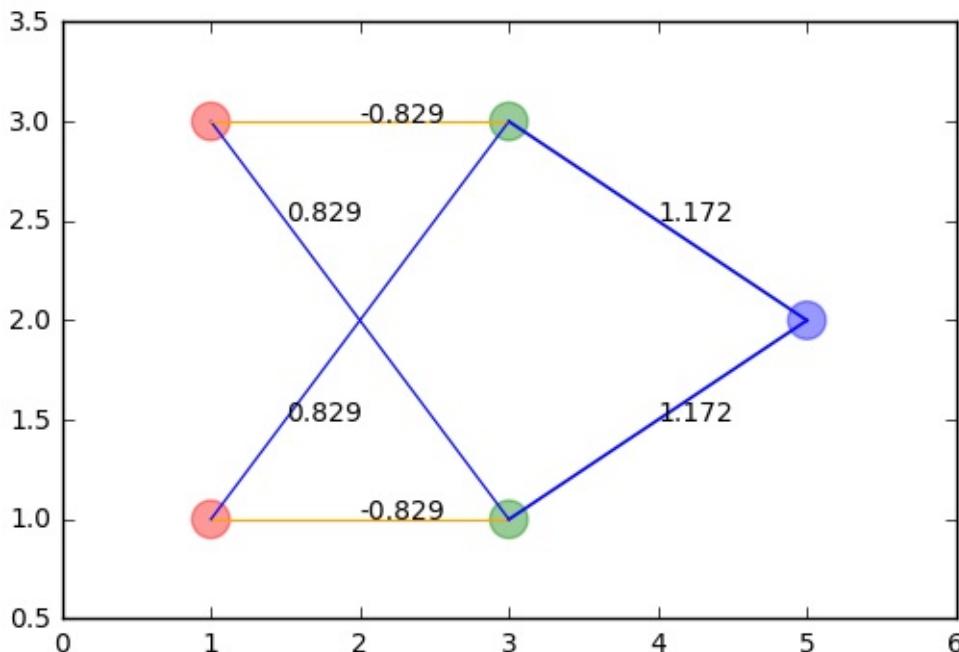
- 权重：

```

W_0:
[[ -0.82895017  0.82891428]
 [ 0.82915729 -0.82918972]]
W_1:
[[ 1.17231631]
 [ 1.1722393 ]]

```

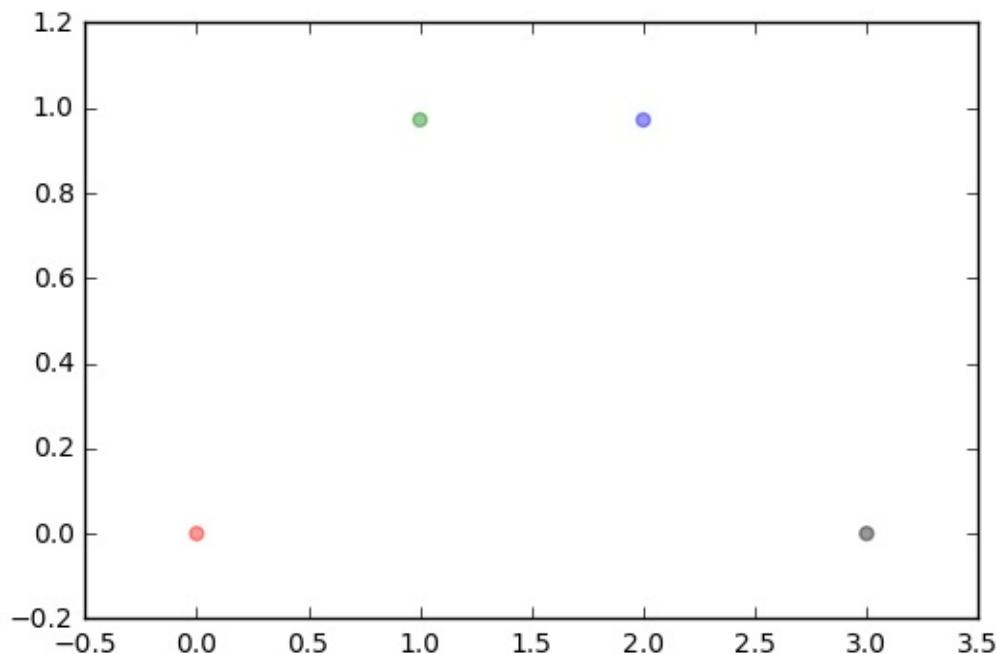
- 权重图：



- 输出：

```
[[ 0.00000000e+00]
 [ 9.72034633e-01]
 [ 9.71685886e-01]
 [ 2.42817347e-04]]
```

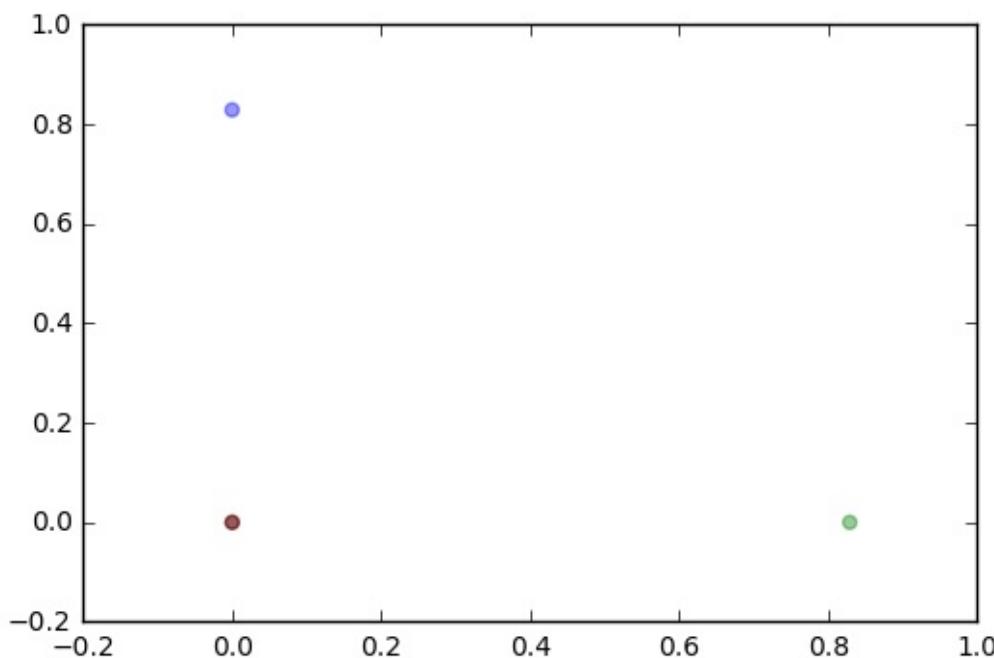
- 输出图：



- 隐藏层输出：

```
[[ 0.00000000e+00  0.00000000e+00]
 [ 8.29157293e-01  0.00000000e+00]
 [ 0.00000000e+00  8.28914285e-01]
 [ 2.07126141e-04  0.00000000e+00]]
```

- 隐藏层输出：注意这里只有**3**个变体。说明了学习也就是消除熵，消除变体的过程。



## 权重的读存

Tensorflow提供了[读存的API](#)，实际应用中十分方便。但这里YJango想让读者能更深的体会TensorFlow的操作（ops）的特点，直接用更新权重的方式读训练好的权重。可以作为练习，想想如果要用这种方式pre-train的话可以怎么做。详细内容看[github](#)。

```
# 该操作可以用于读取已经训练好的权重W和b
# 每层W想要读取的值
W_0=np.array([[-0.82895017,0.82891428],[ 0.82915729,-0.82918972]],dtype='float32')
W_1=np.array([[ 1.17231631],[ 1.1722393 ]],dtype='float32')
# 每层b想要读取的值
b_0=np.array([0,0],dtype='float32')
b_1=np.array([0],dtype='float32')
# 读取ops
reload1=tf.assign(ff.W[0],W_0)
reload2=tf.assign(ff.W[1],W_1)
reload3=tf.assign(ff.b[0],b_0)
reload4=tf.assign(ff.b[1],b_1)
# 执行ops
sess.run([reload1, reload2, reload3, reload4])
```

## 分析

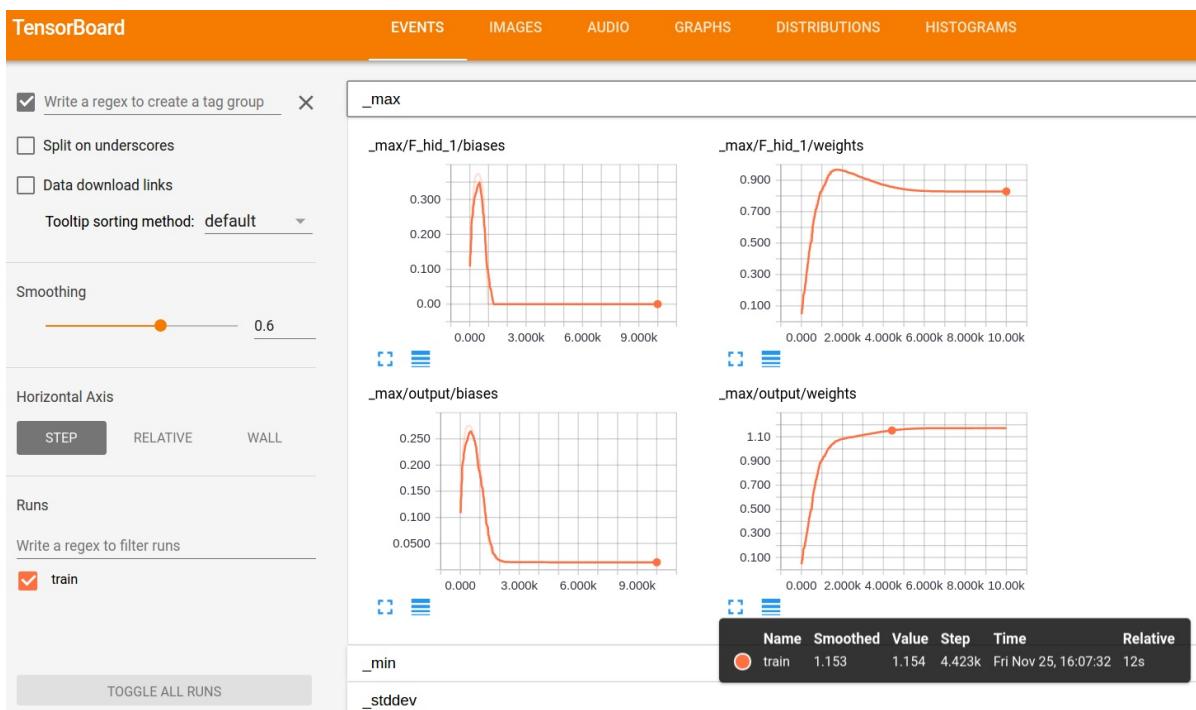
先前的代码中加入统计内容的目的就是为了在分析训练过程中的问题。

- **tensorboard :**

- 加载：`tensorboard --logdir=`（等号后面是log存放的地址）
- 进入：随后进入本地的<http://127.0.1.1:6006> 就可以查看所有统计内容。包括最开始

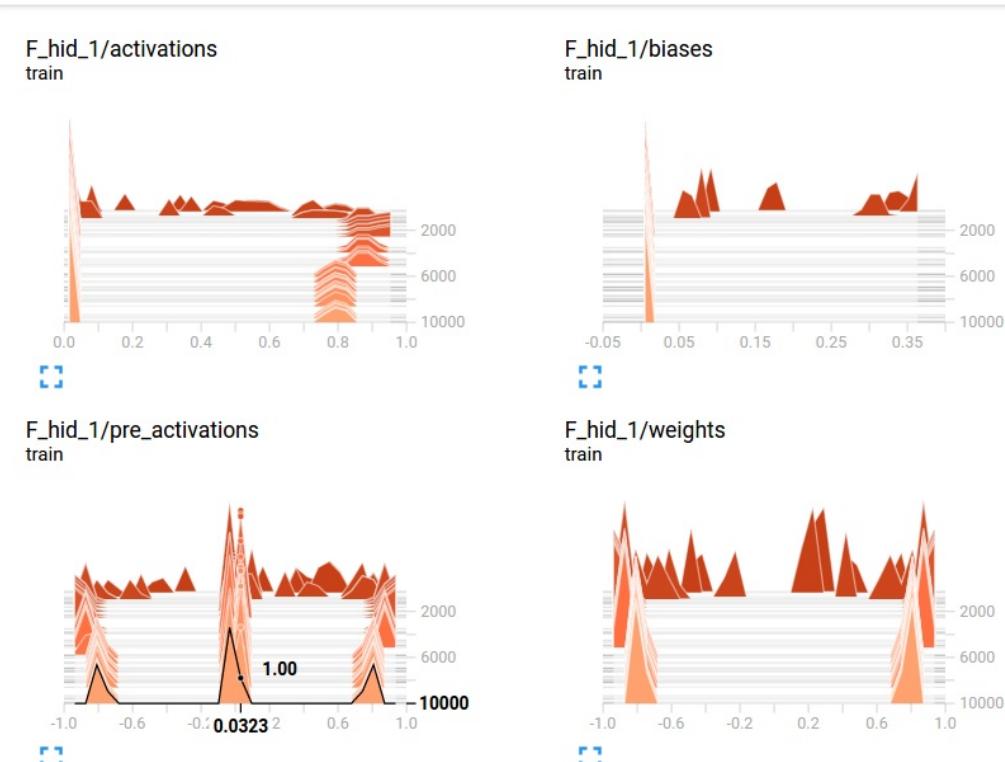
生成的graph图。

- **events:** 下面是各个变量最大值随训练的变化



- **histograms:** 从这里可以看到随着训练（从上到下），隐藏层的各个变量在不同数据间直方图的变化。可以看到随着训练，隐藏层的变体的分布的变化过程。从最初叠加在一起逐渐变得分明。

## F\_hid\_1



## output



YJango花这么大的篇幅来描述的并不是XOR gate问题本身，而是利用这种容易控制的任务让读者尽可能多的体会神经网络的特点、熟悉神经网络的核心要素：权重 $W$ ，偏移 $b$ ，隐藏层输出 $h$ 。随后的卷积神经网络、递归神经网络、所有的神经网络变体的本质就是代码演示中的内容。

并且看过《超智能体》前面内容的朋友，YJango希望你们将线性代数，生物进化，熵增，神经网络的所有内容都联想起来，不要分割。内容若不相关，我是不会特别放在《超智能体》中去描述的。

## 代码演示**LV3**

前两节所描述的[代码演示LV1](#)和[代码演示LV2](#)都是为了分析方便而选择的浅层任务，深层学习对其并不具备什么优势，其他机器学习算法恐怕要比深层学习做的更好。

真正可以发挥深层学习的是具有特定结构的任务。

这一节有两个目的：

- 换一个高度非线性和复杂的任务来再次体会一下深层学习的功能
- 熟悉深层学习训练的完整流程。

[代码演示LV3](#)用于构建网络的代码和[代码演示LV2](#)的完全相同相同，但参数等有所改变。全部代码在[github](#)上。所用到的数据输入输出数据分别是 `X.npy` 和 `Y.npy`。



说明：上图中打条的内容表示本次网络所用到的（不做说明），带有数字的内容会被逐一讲解。

重要：看下面内容之前，请先回顾上一章梯度下降训练法中的基本流程图。

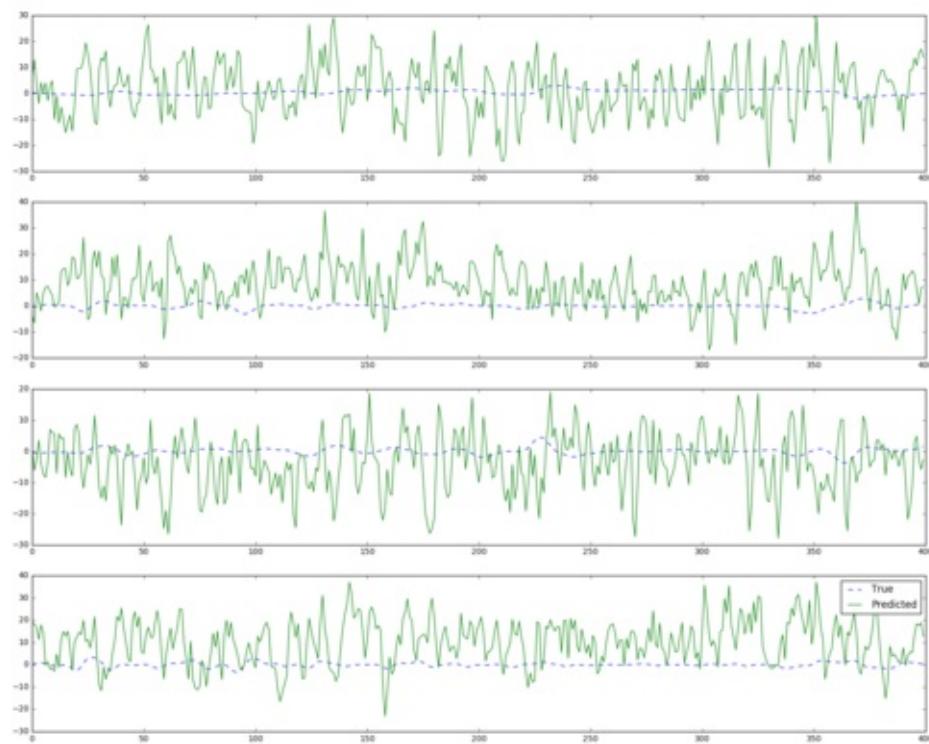
## 任务描述：

- 目标：用声音来预测口腔移动。
- 维度：两者都是实数域，输入是39维，输出是24维： $input \in R^{39}$  ;  $label \in R^{24}$
- 任务类型：因为预测的数值全部都是连续的实数，所以是回归（regression）任务。
- 样本数：39473
- 输出层激活函数：linear（无）
- 损失函数：预测数值和真实数值的差异可用均方差（mean square error : MSE）来表示。

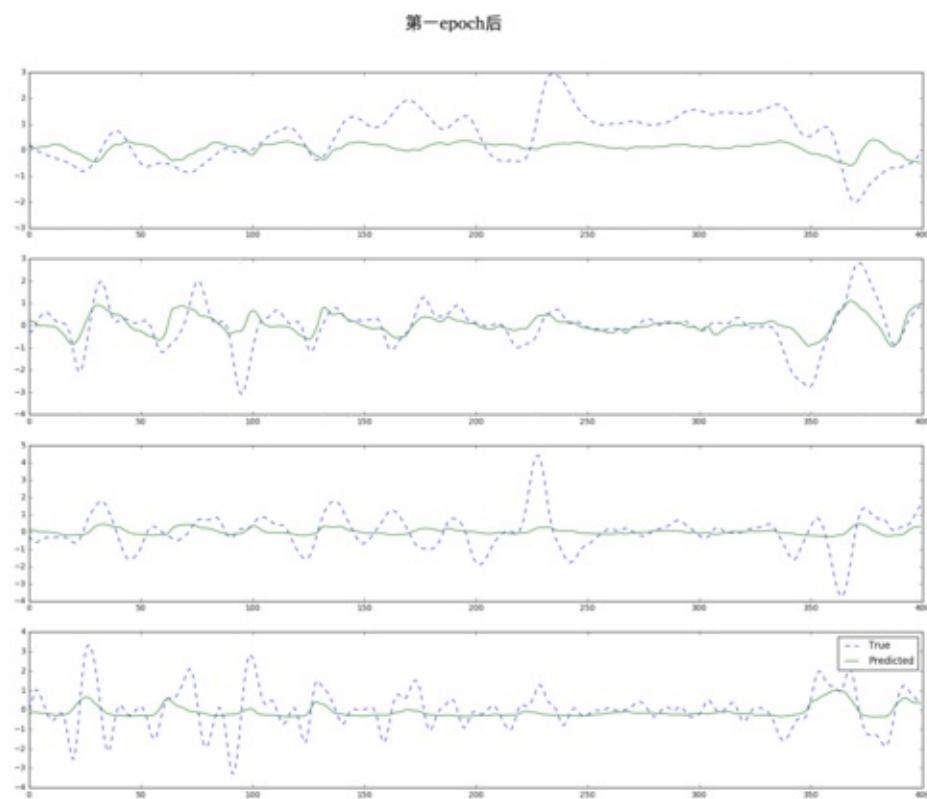
◦ 表达式：
$$L(f(x_i), a_i) = \frac{1}{24} \sum_{j=1}^{24} (f(x_i)_j - a_{ij})^2$$

其中， $j$ 表示维度的角标，由于任务是24维，这里直接用24表示， $i$ 表示不同样本的角标。 $f(x_i)$ 表示经过神经网络计算后的预测值。

- 初始预览：下面是未训练前真实值与预测值在四个不同维度的对比图。目的就是要给网络很多组（输入，输出）数据来更新网络权重，使网络输出尽可能符合真实值的预测。完美情况就是蓝线（真实值）和绿线（预测值）重合。下图中，横轴是时间，纵轴是数值。



- 效果预览：下图是经过若干次训练后的预测值变化图。

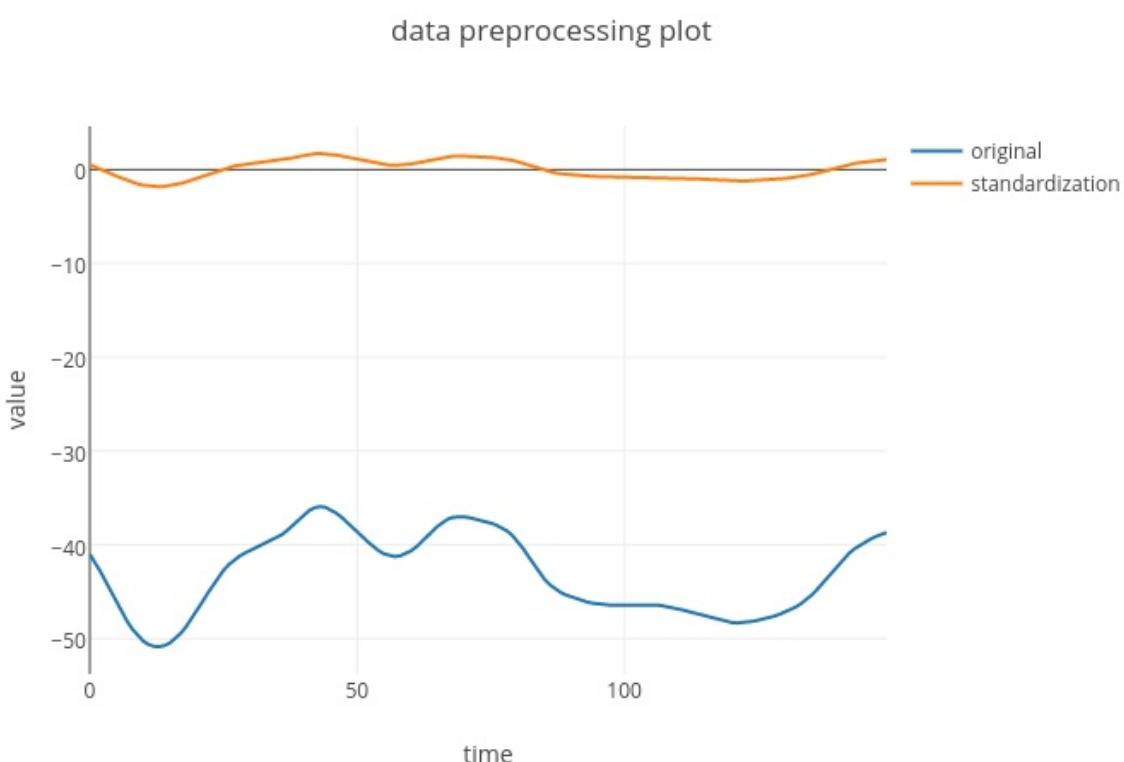


by YJango

## 1. 预处理——平缓变体

- 目的：机器学习的最大困难之一就在于变体。神经网络解决该问题可通过增加隐藏层节点来增强模型拟合变体的能力，然而还有一种相对方式是减弱输入和输出空间的变体。预处理就是该种可以通过缩小输入输出数据的差异性，使得建模变得相对简单的方式之一。
- 效果：下图是将每个样本都减去平均值、除以标准差后的变化（其中上图是变化后，下图是变化前）。

注：点击original或standardization会只显示一个轨迹线，可以看到两条轨迹线的形状相同，但值域不同（若显示有问题，请点下面选项栏的autoscale图标）



- 代码：

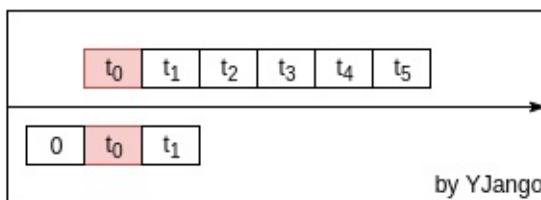
```
def Standardize(seq):
    #subtract mean
    centered=seq-np.mean(seq, axis = 0)
    #divide standard deviation
    normalized=centered/np.std(centered, axis = 0)
    return normalized
```

- 解释：除此之外还有其他的预处理方式。比如之前的激活函数大多用tanh，需要将输出值y限制到[-1,1]的值域中。但该节注重的预处理是消除各个样本的差异性，其中去均值和模值的方法最简单有效。

注：该方式的仅仅是缩小了尺度，并不会把值域限制到固定的范围内。

## 2. 预处理——窗口化

- 目的：如果用每个时刻的输入直接预测对应时刻的输出，由于判断依据较少，结果会不理想。就好比你画我猜，开始的几笔画出的信息在观察者的脑中会产生各种联想，这些联想所形成的集合是无比巨大的，观察者难以确定究竟哪一个才正确。但随着画者展开更多内容。联想范围会逐渐被缩减。用上下文窗（context window）就有这种功效，它提供了更多的判断依据。
- 效果图：将相邻的时序向量并接在一起形成一个“大”向量作为输入，而输出一般选择中间那一个向量对应的label，为得是有上文和下文的共同限制。这里的每个向量也叫作帧（frame）。而没有上下文的部分可由0来填补（zero padding），填零的部分会失去该维度的信息，但并不会对其他部分的计算产生误差。下图中的上半部分是用上下文窗之前的序列。而下半部分是用size为3（3个frames）的window形成的新输入序列。 $t$ 的维度是39维的话，下半部分的序列就是 $3 \times 39$ 维度，总个数不变。



- 代码：

```
def Makewindows(indata,window_size=41):
    outdata=[]
    mid=int(window_size/2)
    indata=np.vstack((np.zeros((mid,indata.shape[1])),indata,np.zeros((mid,indata.shape[1]))))
    # 前后补零
    for i in range(indata.shape[0]-window_size+1):
        outdata.append(np.hstack(indata[i:i+window_size]))
    return np.array(outdata)
```

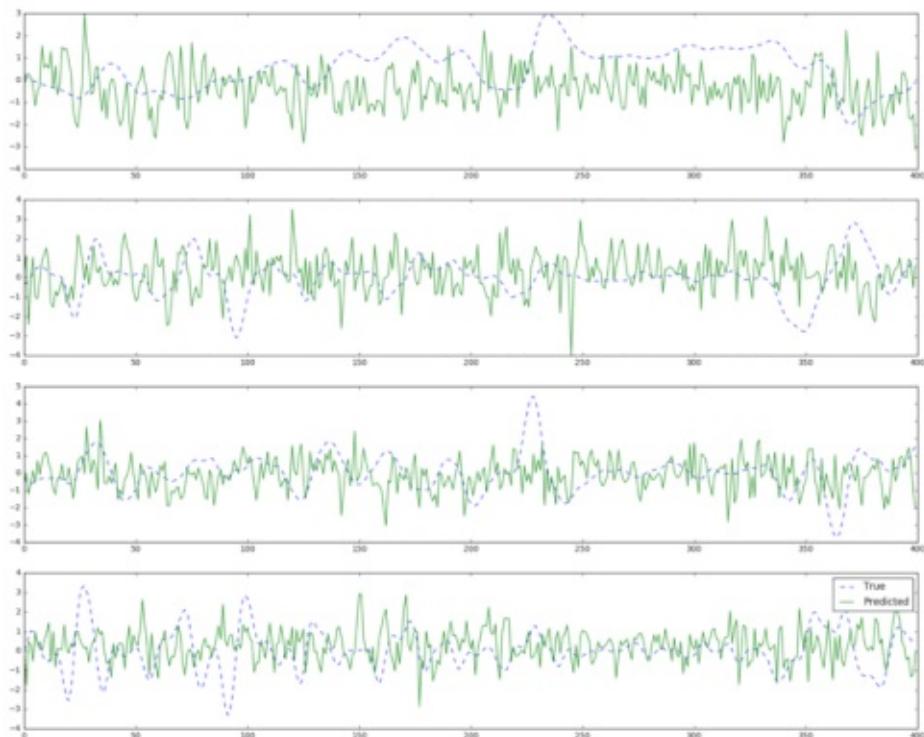
- 解释：窗口化并不是一种非常好的处理时序信号的方式，在随后的递归神经网络中会提到它的弊端以及递归神经网络的解决方式。

## 3. 权重初始化

- 目的：权重初始化决定了网络从什么位置开始训练。但请不要认为“网络从任何位置开始都可以训练到相同结果，仅是耗时差别”。良好的起始位置不仅可以减少训练耗时，也可以使模型的训练更加稳定，并且可以避开很多训练上的问题。

- 效果图：这里展示一下初始化对于“dying ReLU”问题的缓解。由于ReLU的梯度简单，可以避开像sigmoid/tanh这类损失函数所造成的严重梯度消失（gradient vanishing），ReLU基本上成为深层学习中激活函数的标配。但在训练过程中，ReLU也会使很多节点再也无法被激活（“dying ReLU”）。下图展示了该问题的严重。

注：梯度消失简单说就是距离输出越远的层的梯度会越接近0，当用链式规则计算梯度时，就会使整体的梯度都拉向0，造成更新的权重并不是想要的正确数值。



如果用[代码演示LV2](#)中的随机权重初始化的话，是无法训练该任务的。因为随着训练，原本随机初始的预测值全部都变成了0，而后的任何数据都无法再对模型的权重更新起到作用。解决的方式之一就是合理的权重初始化。

- 代码：只需要在LV2的权重初始化的基础上将所有值都除以输出维度的开方，使权重的初始值域落在 $[-1/\sqrt{n}, 1/\sqrt{n}]$ 内。其中n是该层的输出维度（[论文](#)）。

```

def weight_init(self, shape):
    # shape : list [in_dim, out_dim]
    # 在这里更改初始化方法
    # 方式1：下面的权重初始化dropout率40%的网络，可以使用带有6个隐藏层的神经网络。
    #           若过深，则使用dropout会难以拟合。
    #initial = tf.truncated_normal(shape, stddev=0.1)/ np.sqrt(shape[1])
    # 方式2：下面的权重初始化dropout率40%的网络，可以扩展到12个隐藏层以上（通常不会用那么多
)
    initial = tf.random_uniform(shape, minval=-np.sqrt(5)*np.sqrt(1.0/shape[0]),
maxval=np.sqrt(5)*np.sqrt(1.0/shape[0]))
    return tf.Variable(initial)

```

- 解释：除此之外的解决“dying ReLU”的方式还有“Leaky ReLU”。

## 过拟合问题

- 比喻：学习要达到的效果可以用高考时的做题来比喻：我们希望通过“做题、对照答案”来学会如何解该问题所有类型的题目，最终希望可在高考时解出问题。

“练习题”就是机器学习中的数据（**data**），同时具有问题（**observation**）和答案（**label**）两种数据。不断“做题、对照答案”的过程就是学习（**training**）。而高考的题就是只有问题（**observation**），需要靠以前的学习的知识（**function**）来解出答案，从而考察（**test**）我们是否“真”的学会了该知识（**function**）。但是不同题有不同的思路（**variation**），一直做同类练习题会使自己过分纠结于该类题的细节规律（**overfitting**），再次遇到其他类型问题时却无法解出。

机器学习也存在相同的现象：模型会过分学习训练数据（**training set**）中的规律，而在预测未见过的数据（**testing set**）时表现不佳。这种现象叫做过拟合（**overfitting**），而我们真正希望的是模型可以通过有限数据的学习涵盖所有变体，这种能力叫做普遍化（**generalization**）。深层学习相比浅层学习的优势就是在于其普遍化能力。然而普遍化是一个永远的课题，对人脑的学习来说同样如此。

- 举例：为了感受普遍化是一件多么难的问题，请考虑这样一个例子。这里有一组数据。括号前一个数表示x，后一个数表示对应的y。（-1,1),(2,0)，仅有两个数据时可以观察出什么规律？

单靠观察，该规律可以是正数就输出0，负数就输出1。然而也可以是奇数就输出1，偶数就输出0。或者两者同时满足。事实上运用各种数学运算，可以有无数种**function**能够拟合上面两个数据，所有机器学习算法都可以做到，然而机器怎么知道我们想要哪种**function**？虽然可以通过更多的数据来确定想要的**function**。但是数据是无限的，如果能够获得所有情况的数据也就没有必要学习了。

- 说明：这里引用Bengio大神的一句话。

If you don't assume much about the world, it's actually impossible to learn about it.

如果将所有符合训练集规律的functions组成一个集合叫 $\mathcal{F}$ ，那么想要用更少的数据来学习具有普遍性的function，就需要加入先验知识来缩小 $\mathcal{F}$ 。不过，缩小 $\mathcal{F}$ 虽可排除掉一部分的functions，加快搜索，但代价也是无法学习被排除掉了的functions，这也就是No free lunch theorem所描述的内容。深层学习对于数据的assume是并行与迭代，进而排除掉了很多functions。而卷积神经网络、递归神经网络等变体就是优先搜索特定的functions，从而排除掉另一部分的functions。换句话说：

神经网络的变体实际上是对函数优先搜索空间 $\mathcal{F}$ 的调整。

防止过拟合也是对 $\mathcal{F}$ 的调整。但“过拟合”是对该问题的一个较为偷懒的描述。因为我们想知道的是造成过拟合的原因，从而加以调整。

## 4. 防过拟合——L2 regularization

- 作用：L2对于神经网络有两个作用。其中一个作用在之前的章节中深层神经网络——方式3：加入隐藏层中已描述。具有强制让所有节点均摊任务的作用。

而另一个作用举例来说：当有两个节点的网络需要输出3时，可以让其中一个是300，另一个是-297；也可以让其中一个是1.5，另一个是1.5，L2会鼓励网络选择后者。

注：我们是利用L2的特点对网络的权重在训练中进行调整，仅仅是缓解作用，并不能完全解决问题。其次附加额外loss的做法都会多少对最终结果产生负面影响。比如过高的L2强度会压制目标loss。

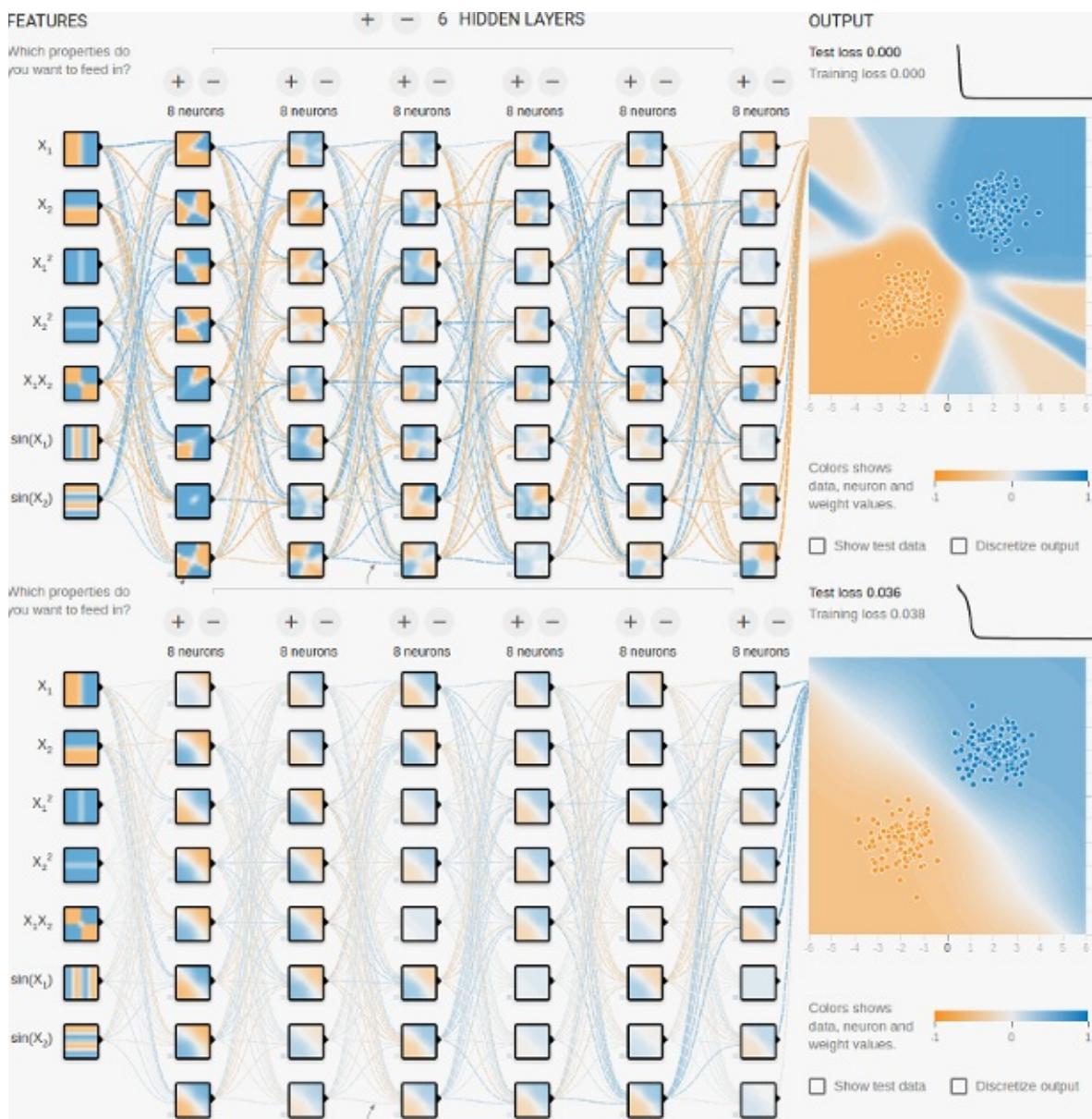
一般做法：我们会选择稍微过拟合的网络，再增加防止过拟合的方法加以训练，其目的是：用超过学习训练集所需的节点来建模，从而涵盖测试集中的变体。

- 效果图：下图是用playground训练的效果演示。上半部分是没用L2，下半部分用了L2。

1.先观察每层的连线（权重）的深浅，越浅绝对值越低。用L2的模型的权重十分均匀且数值很浅。

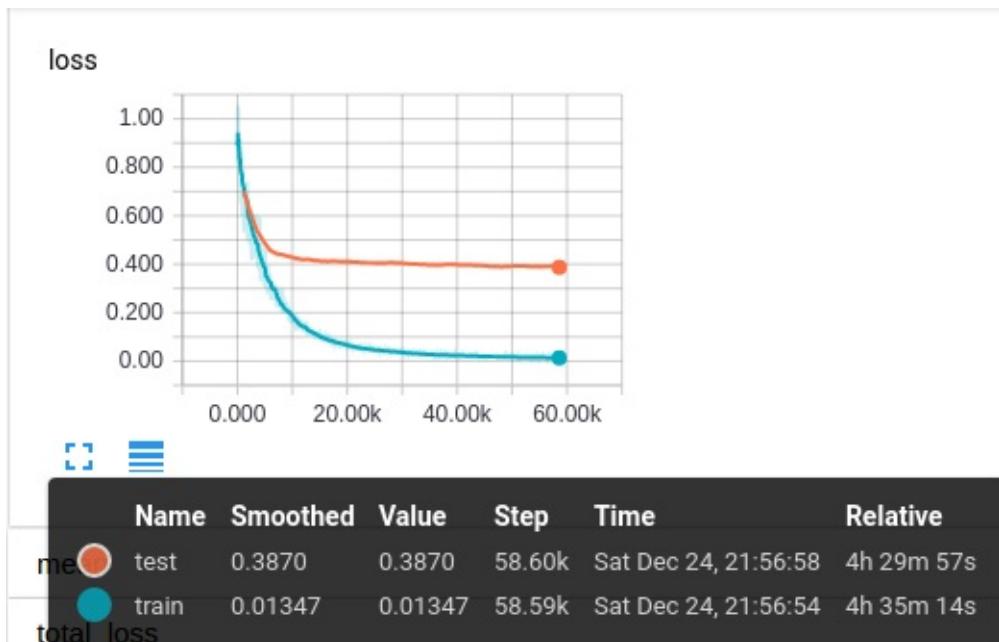
2.再看每个节点的方块，没用L2的网络就出现了300-297去获得3的方式，而用了L2的则倾向于用更简单的1.5+1.5去获得3。

3.最后再看实际分类图的差别。

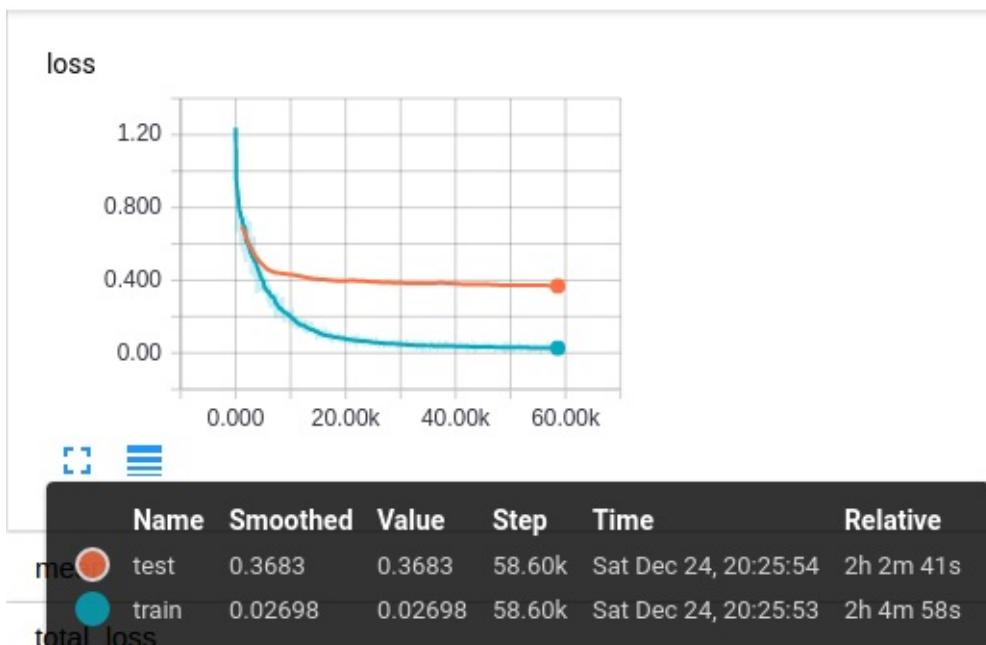


下面这张图是训练声音预测口腔移动任务50个epoch时，loss值随训练的变化曲线。x轴是训练次数（iteration），y轴是loss值。红线是测试集loss，蓝线是训练集loss。可以明显看到两条变化线的差别。测试集的loss最终停在了0.387上，而训练集的loss近乎为0。

注：一次epoch是指将所有样本数都训练完。一次iteration是指更新一次权重。



而下面这张图则是使用了L2后的效果。这是测试集的loss在50个epoch后，停在了0.368上。（忽略图中的训练时间，因为用的是不同的GPU）



## 5. 防过拟合——dropout

- 作用：拿语音为例，我们最终获得的语音信号并不纯粹（pure），而是有很多带有“额外规律”的杂讯（noise）掺杂其中，如录音设备的噪音，说话人的规律，环境噪音的规律。

[dropout论文](#)对其效果的解释是：dropout实际上是多个模型预测的平均值。但为什么平均多个模型的预测值会提高表现？

个人理解：dropout可阻碍网络学习仅存在于训练集中局部的“额外规律”。

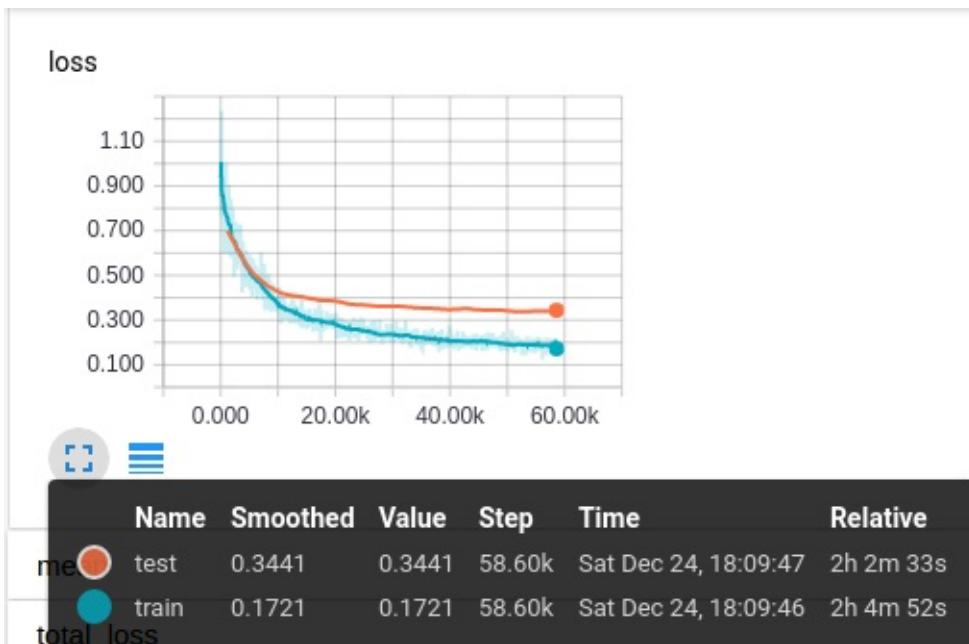
- 现象：目标规律（想学习的规律）基本在所有训练样本中都存在，但杂讯规律（由杂讯带来的额外规律）仅存在于部分的样本中，由于权重是随机初始，并且训练样本也是随机打乱顺序后逐个送入网络进行训练。当若干有相同杂讯规律的训练样本连续被送入网络训练后，网络就会记住该规律。
- 影响：神经网络一大特点在于它可以拆分因素，包括杂讯，并将他们分开建模。但上面的现象会影响神经网络对于因素的拆分。假如目标信号是10，而杂讯是2，获得的实际语音数据是12，混合信号未必会被拆分成 $10+2$ 的形式，有可能是 $9+3$ 。本该被拆分成10的目标信号变成了9，以至于在测试集中表现不佳。
- 缓解：但是这种现象有一个特点，就是杂讯规律相对目标规律而言较为薄弱，同时也并非在所有训练样本中都存在。`dropout`正是对这一特点展开的攻击。`dropout`会随机扔掉节点，让已经学到的规律被遗忘掉。这样即便是网络由于随机打乱顺序所训练的几个带有杂讯规律的样本，所形成的杂讯规律也会被遗忘掉。只有像目标规律稳固的规律才会得意保留。由于这种现象是随机性造成的，所以平均各个模型的预测值后就会抵消掉额外规律对目标规律的干扰（比如此次拆分成 $9+3$ ，下次拆分成 $11+1$ 的形式。取平均值后 $9+11=10$ ）。有一种筛子的作用。我想人类的遗忘也有类似作用。

`dropout`可阻碍网络学习仅存在于训练集中局部的“额外规律”。

- 效果图：`dropout`的实现方法和特点在[代码演示LV2——训练前-step 5](#)中已描述。下图是用了`dropout`（随机扔掉30%节点）后的`loss`图。`dropout`的实现非常简单，但却异常强大。`test`的`loss`最终落在了0.344。

从图中还可以看出来`dropout`确实是阻碍网络学习训练集中的规律，并且不仅仅是额外规律，连目标规律也被殃及（没用`dropout`前，50个epoch后的训练集的`loss`仅有0.01~0.02左右，用了以后连训练集`loss`都变成了0.172）。但其实随着训练，训练集的`loss`还会进一步下降一些。最终我们想要提升的是测试集的表现。

浅蓝色的线是实际的`loss`。`dropout`会使其上下波动很大。可以感受到，若规律不是在所有样本中都存在，则很难在这种`dropout`的遗忘下存留。



注：还有一种阻碍网络学习仅存在于训练集中的“额外规律”的方法就是在训练集中加入其他杂讯，从而屏蔽掉额外规律。注：如果你的代码是每一层之后都加入dropout layer，那么随着层数的加深，阻碍网络学习规律的强度也会增加。

## 6. 随机打乱

随机打乱（shuffle）训练数据送入网络更新的顺序十分重要。如dropout中所描述的现象。若更新网络的顺序始终不变，就会使网络受杂讯规律的干扰而拆分出错误的目标规律。若每个epoch对网络的作用也是几乎相同的，当该epoch的规律学习完毕后，网络几乎不再变化（如果看到自己训练的网络效果不好，一定要记得检查是否有shuffle）。正确的做法是：

每个epoch之后一定要将训练数据随机打乱顺序。

## 7. Batch size

如梯度下降训练法中基本流程图-训练网络所说，Stochastic Gradient Descent (SGD) 有“跳出”局部最小值（鞍点）的作用。纯粹的SGD是每次只计算一个样本的梯度来更新权重。实际应用中，一般会用10-512个样本（batch）计算出他们的梯度平均值来更新权重。

大的Batch size：

- 优点：有节约训练时间、收敛方向稳定的优点。
- 缺点：普遍化的程度降低、容易陷入鞍点、没有完全利用数据。

小的Batch size：

- 优点：普遍化的程度升高、利于逃脱鞍点、更全面的利用数据。

- 缺点：训练时间提升、收敛方向波动。

## 8. 停训标准

很多情况下随着模型的训练，训练集的loss会越来越好，然而测试集的loss反而变得糟糕。早停（early stopping）的思路就是在变糟之前停止训练。比如当测试集的loss停止降低3次以后就可以让网络自动停止。

在正常分set的时候并不会单一的分成训练、测试集。而是分成train/test/validation 三部分。原因在于test往往是指从未见过的数据。而我们在调试网络时会从train set中另分一部分validation进行反馈和调试。

## 完整代码

完整代码可到[github](#)上查看，为节省篇幅，这里不再粘贴。网络模型的FNN类和代码演示LV2的完全一致，除了改变了权重初始化的值域。值得说明的是数据的处理：

```
mfc=np.load('X.npy')
art=np.load('Y.npy')
x=[]
y=[]
for i in range(len(mfc)):
    x.append(Makewindows(Standardize(mfc[i])))# 输入数据标准化并窗口化
    y.append(Standardize(art[i]))# 输出数据标准化
vali_size=20 # 分多少百分比的数据作为验证集
totalsamples=len(np.vstack(x))
X_train=np.vstack(x)[int(totalsamples/vali_size):].astype("float32")#记得要改变类型。
Y_train=np.vstack(y)[int(totalsamples/vali_size):].astype("float32")

X_test=np.vstack(x)[:int(totalsamples/vali_size)].astype("float32")
Y_test=np.vstack(y)[:int(totalsamples/vali_size)].astype("float32")

print(X_train.shape,Y_train.shape,X_test.shape,Y_test.shape)#查看样本数和维度是否正确。
# 实际显示为：
((37500, 1599), (37500, 24), (1973, 1599), (1973, 24))
```

下面的函数是一个简单显示预测效果的plot图。仅此而已。

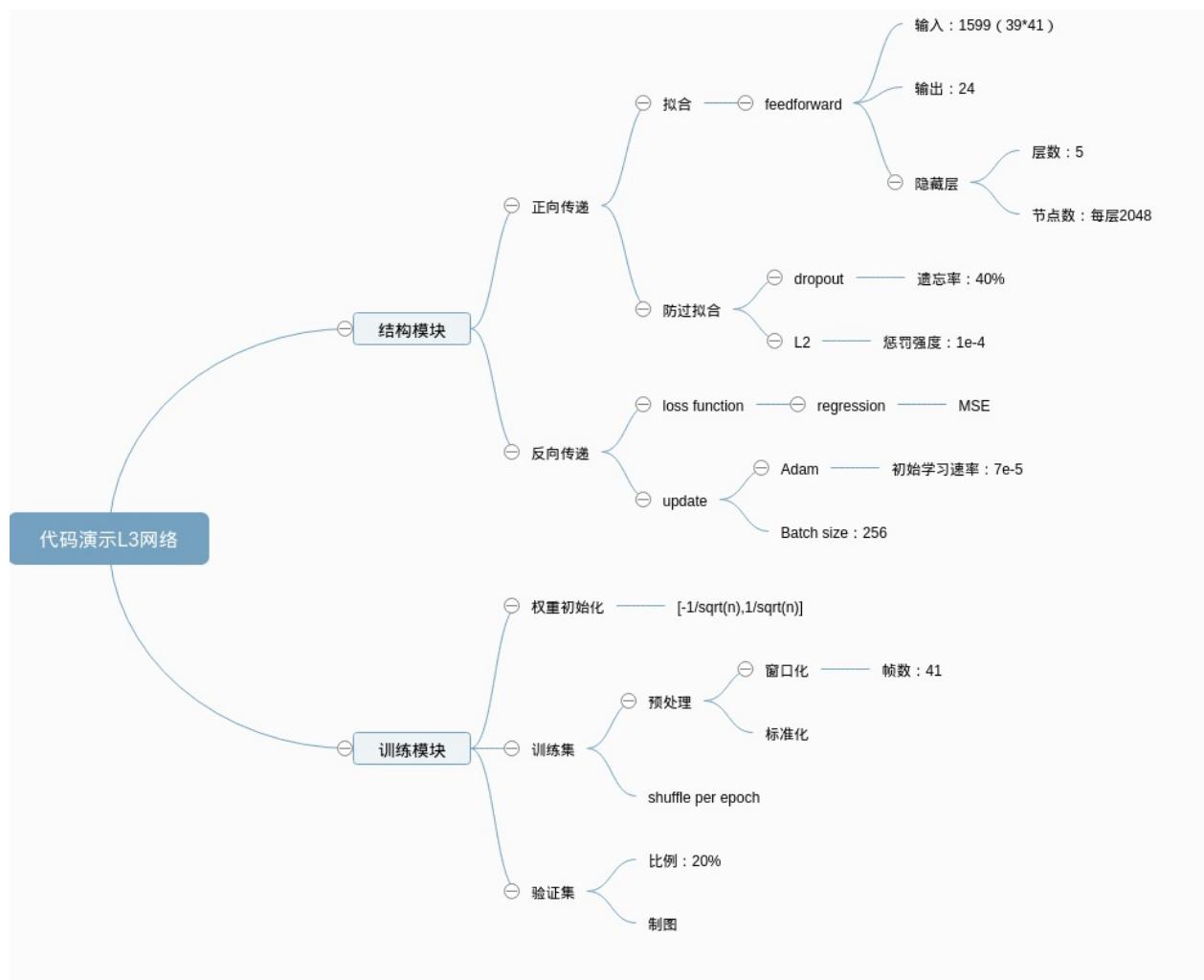
```
def plots(T,P,i, n=21,length=400):
    m=0
    plt.figure(figsize=(20,16))
    plt.subplot(411)
    plt.plot(T[m:m+length,7], '--')
    plt.plot(P[m:m+length,7])

    plt.subplot(412)
    plt.plot(T[m:m+length,8], '--')
    plt.plot(P[m:m+length,8])

    plt.subplot(413)
    plt.plot(T[m:m+length,15], '--')
    plt.plot(P[m:m+length,15])

    plt.subplot(414)
    plt.plot(T[m:m+length,16], '--')
    plt.plot(P[m:m+length,16])
    plt.legend(['True', 'Predicted'])
    plt.savefig('epoch'+str(i)+'.png')
    plt.close()
```

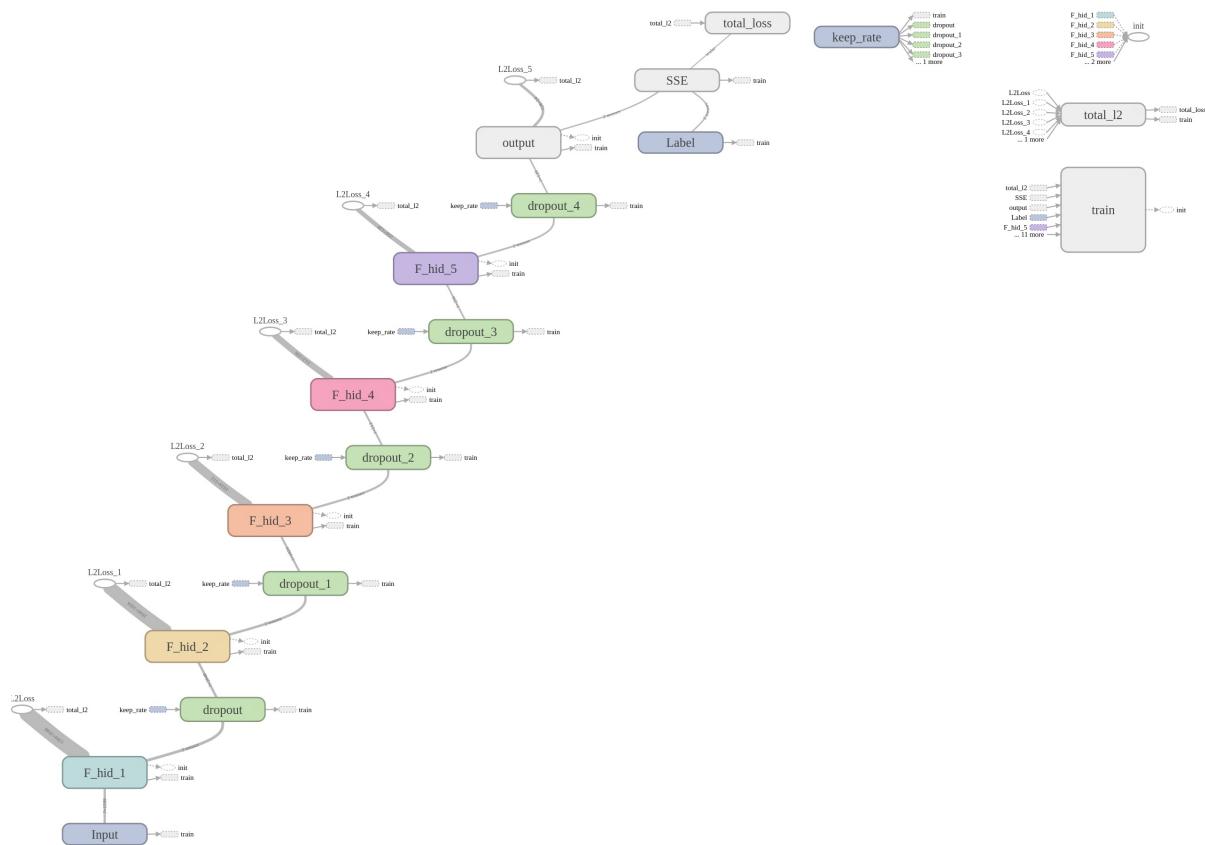
## 实验参数



注：上述参数不是最优参数。为了节省时间而随意设定的。

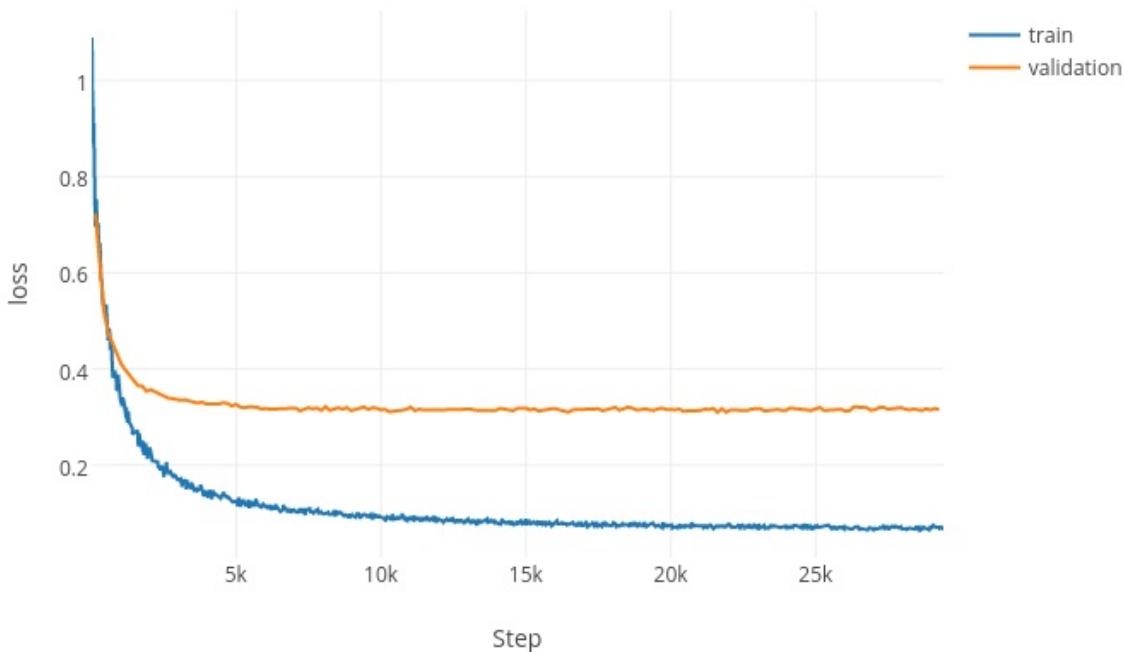
注：上述代码也是为了让读者有体会而刻意编写的。如果熟悉流程，请放心用如Keras、tensorlayer这样的库包，可以帮助节省很多编写和省去你自己测试和找bug的时间。

- 网络结构图：



- 训练/验证 **loss**：训练了200个epoch后的loss图，最后的验证loss达到了0.315左右。这次的图像并没有那么波动的原因是选择了256的batch size。不过feedforward做时序信号预测也只能到达这种程度了。随后会用相同任务和递归神经网络进行比较。

train/vali losses



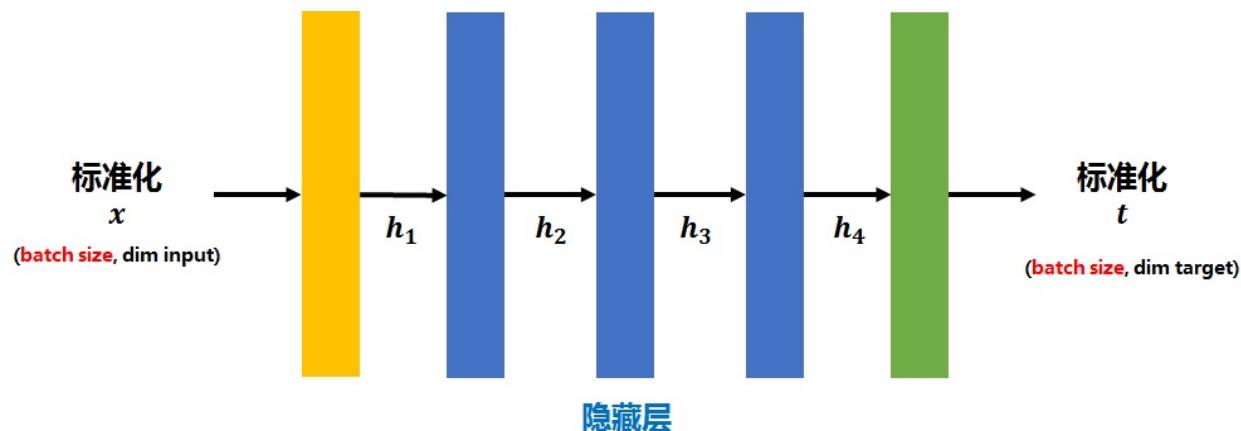


## 思考

[代码演示LV3](#)的数据预处理中提到过：在数据预处理阶段，数据会被标准化（减掉平均值、除以标准差），以降低不同样本间的差异性，使建模变得相对简单。

我们又知道神经网络中的每一层都是一次变换，而上一层的输出又会作为下一层的输入继续变换。如下图中， $x$ 经过第一层 $\phi(W_{h_1} \cdot x + b_{h_1})$ 的变换后，所得到的 $h_1$ ；而 $h_1$ 经过第二层 $\phi(W_{h_2} \cdot h_1 + b_{h_2})$ 的变换后，得到 $h_2$ 。

$h_1$ 在第二层所扮演的角色就是 $x$ 在第一层所扮演的角色。我们将 $x$ 进行了标准化，那么，为什么不对 $h_1$ 也进行标准化呢？



[Batch Normalization](#)论文便首次提出了这样的做法。

Batch Normalization (BN) 就是将每个隐藏层的输出结果（如 $h_1, h_2, h_3$ ）在batch上也进行标准化后再送入下一层（就像我们在数据预处理中将 $x$ 进行标准化后送入神经网络的第一层一样）。

## 优点

那么Batch Normalization (BN)有什么优点？BN的优点是多个并存，但这里只提一个最容易理解的优点。

## 训练时的问题

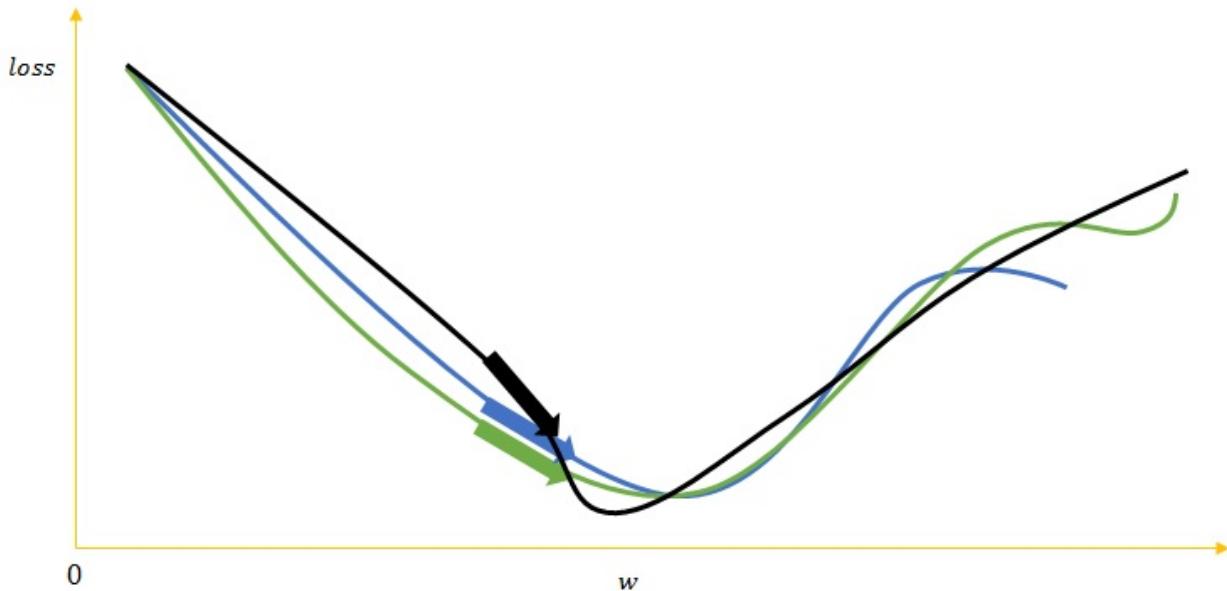
尽管在讲解神经网络概念的时候，神经网络的输入指的是一个向量 $x_i$ 。

但在实际训练中有：

- 随机梯度下降法（Stochastic Gradient Descent）：用一个样本的梯度来更新权重。
- 批量梯度下降法（Batch Gradient Descent）：用多个样本梯度的平均值来更新权重。

如下图所示，绿、蓝、黑的箭头表示三个样本的梯度更新网络权重后loss的下降方向。

若用多个梯度的均值来更新权重的批量梯度下降法可以用相对少的训练次数遍历完整个训练集，其次可以使更新的方向更加贴合整个训练集，避免单个噪音样本使网络更新到错误方向。



然而也正是因为平均了多个样本的梯度，许多样本对神经网络的贡献就被其他样本平均掉了，相当于在每个epoch中，训练集的样本数被缩小了。batch中每个样本的差异性越大，这种弊端就越严重。

一般的解决方法就是在每次训练完一个epoch后，将训练集中样本的顺序打乱再训练另一个epoch，不断反复。这样重新组成的batch中的样本梯度的平均值就会与上一个epoch的不同。而这显然增加了训练的时间。

同时因为没办法保证每次更新的方向都贴合整个训练集的大方向，只能使用较小的学习速率。这意味着训练过程中，一部分steps对网络最终的更新起到了促进，一部分steps对网络最终的更新造成了干扰，这样“磕磕碰碰”无数个epoch后才能达到较为满意的结果。

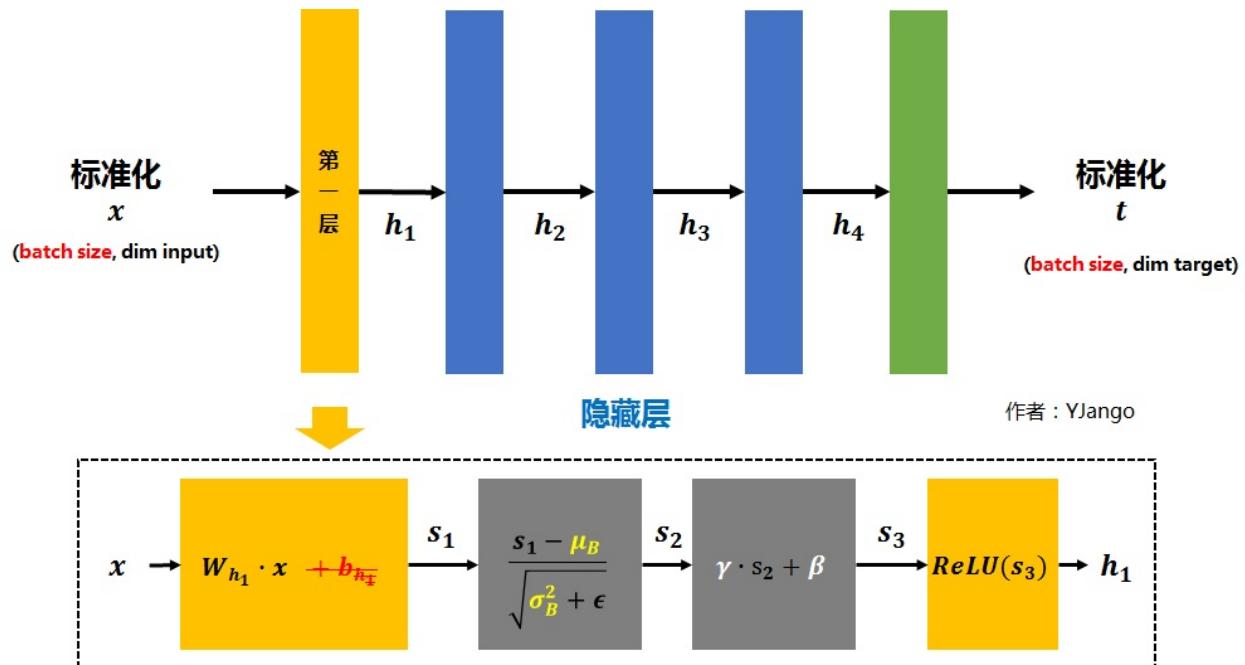
注：一个epoch是指训练集中的所有样本都被训练完。一个step或iteration是指神经网络的权重更新一次。

为了解决这种“不效率”的训练，BN首先是把所有的samples的统计分布标准化，降低了batch内不同样本的差异性，然后又允许batch内的各个samples有各自的统计分布。所以，

BN的优点自然也就是允许网络使用较大的学习速率进行训练，加快网络的训练速度（减少epoch次数），提升效果。

## 做法

设，每个batch输入是 $x = [x_0, x_1, \dots, x_n]$ （其中每个 $x$ 都是一个样本， $n$ 是batch size）假如在第一层后加入Batch normalization layer后， $h_1$ 的计算就改为下图所示的那样。



- 矩阵 $x$ 先经过 $W_{h1}$ 的线性变换后得到 $s_1$ 
  - 注：因为减去batch的平均值 $\mu_B$ 后， $b$ 的作用会被抵消掉，所以没必要加入 $b$ （红色删除线）。
- 将 $s_1$ 再减去batch的平均值 $\mu_B$ ，并除以batch的标准差 $\sqrt{\sigma_B^2 + \epsilon}$ 得到 $s_2$ 。 $\epsilon$ 是为了避免除数为0的情况所使用的微小正数。
  - 注：但 $s_2$ 会被限制在正态分布下，使得网络的表达能力下降。为解决该问题，引入两个新的parameters： $\gamma$ 和 $\beta$ 。 $\gamma$ 和 $\beta$ 是在训练时网络自己学习得到的。
- 将 $s_1$ 乘以 $\gamma$ 调整数值大小，再加上 $\beta$ 增加偏移后得到 $s_3$ 。
- 为加入非线性能力， $s_3$ 也会跟随着ReLU等激活函数。
- 最终得到的 $h_1$ 会被送到下一层作为输入。

需要注意的是，上述的计算方法用于在训练。因为测试时常会只预测一个新样本，也就是说batch size为1。若还用相同的方法计算 $\mu_B$ ， $\mu_B$ 就会是这个新样本自身， $s_1 - \mu_B$ 就会成为0。

所以在测试时，所使用的 $\mu$ 和 $\sigma^2$ 是整个训练集的均值 $\mu_P$ 和方差 $\sigma_P^2$ 。

# 变体神经网络

## 神经网络的问题

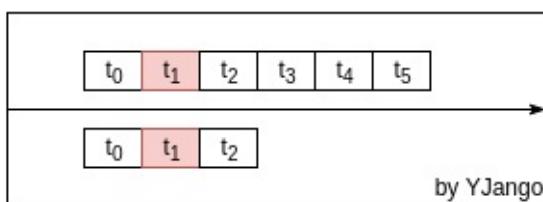
# 循环神经网络——介绍

## 时序预测问题

[代码演示3](#)已经展示了如何用前馈神经网络（feedforward）来做时序信号预测。

### 一、用前馈神经网络来做时序信号预测有什么问题？

- 依赖受限：前馈网络是利用窗处理将不同时刻的向量并接成一个更大的向量。以此利用前后发生的事情预测当前所发生的情况。如下图所示：



但其所能考虑到的前后依赖受限于将多少个向量（window size）并接在一起。所能考虑的依赖始终是固定长度的。

- 网络规格：想要更好的预测，需要让网络考虑更多的前后依赖。

例：若仅给“国（）”，让玩家猜括号中的字时，所能想到的可能性非常之多。但若给“中国（）”时，可能性范围降低。当给“我是中国（）”时，猜中的可能性会进一步增加。

那么很自然的做法就是扩大并接向量的数量，但这样做的同时也会使输入向量的维度和神经网络第一层的权重矩阵的大小快速增加。如[代码演示3](#)中每个输入向量的维度是39，41帧的窗处理之后，维度变成了1599，并且神经网络第一层的权重矩阵也变成了 $1599 \text{ by } n$  ( $n$ 为下一层的节点数)。其中很多权重都是冗余的，但却不得不一直存在于每一次的预测之中。

- 训练所需样本数：前两点可能无伤大雅，而真正使得递归神经网络（recurrent）在时序预测中击败前馈神经网络的关键在于训练网络所需要的数据量。

## 网络差异之处

几乎所有的神经网络都可以看作为一种特殊制定的前馈神经网络，这里“特殊制定”的作用在于缩减寻找映射函数的搜索空间，也正是因为搜索空间的缩小，才使得网络可以用相对较少的数据量学习到更好的规律。

例：解一元二次方程  $y = ax + b$ 。我们需要两组配对的  $(x, y)$  来解该方程。但是如果我们知道  $y = ax + b$  实际上是  $y = ax$ ，这时就只需要一对  $(x, y)$  就可以确定  $x$  与  $y$  的关系。递归神经网络和卷积神经网络等神经网络的变体就具有类似的功效。

## 二、相比前馈神经网络，递归神经网络究竟有何不同之处？

需要注意的是递归网络并非只有一种结构，这里介绍一种最为常用的递归网络结构。

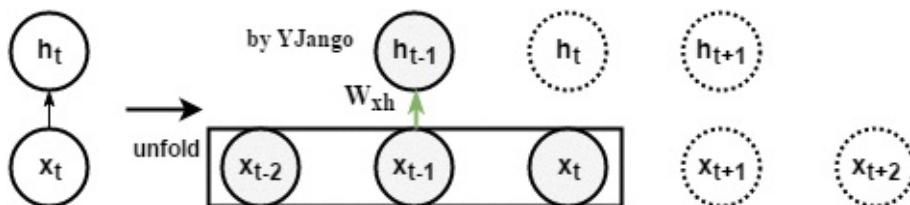
### 数学视角

首先让我们用从输入层到隐藏层的空间变换视角来观察，不同的地方在于，这次将时间维度一起考虑在内。

注：这里的圆圈不再代表节点，而是状态，是输入向量和输入经过隐藏层后的向量。

### 前馈网络：window size为3帧的窗处理后的前馈网络

- 动态图：左侧是时间维度展开前，右侧是展开后（单位时刻实际工作的只有灰色部分）。前馈网络的特点使不同时刻的预测完全是独立的。我们只能通过窗处理的方式让其照顾到前后相关性。

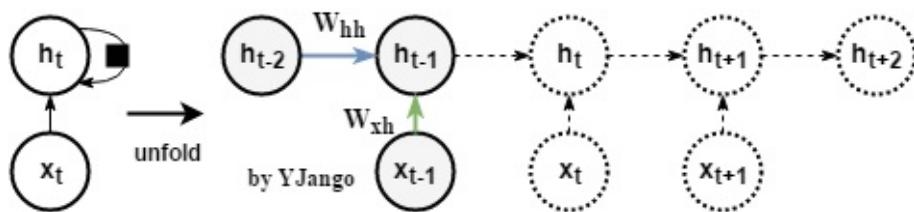


- 数学式子： $h_t = \phi(W_{xh} \cdot concat(x_{t-1}, x_t, x_{t+1}) + b)$ ，concat 表示将向量并接成一个更大维度的向量。
- 学习参数：需要从大量的数据中学习  $W_{xh}$  和  $b$ 。

要学习各个时刻（3个）下所有维度（39维）的关系（ $39*3$ 个），就需要很多数据。

### 递归网络：不再有window size的概念，而是time step

- 动态图：左侧是时间维度展开前，回路方式的表达方式，其中黑方框表示时间延迟。右侧展开后，可以看到当前时刻的  $h_t$  并不仅仅取决于当前时刻的输入  $x_t$ ，同时与上一时刻的  $h_{t-1}$  也相关。



- 数学式子： $h_t = \phi(W_{xh} \cdot x_t + W_{hh} \cdot h_{t-1} + b)$ 。 $h_t$ 同样也由 $x_t$ 经 $W_{xh}$ 的变化后的信息决定，

但这里多另一份信息： $W_{hh} \cdot h_{t-1}$ ，而该信息是从上一时刻的隐藏状态 $h_{t-1}$ 经过一个不同的 $W_{hh}$ 变换后得出的。

注： $W_{xh}$ 的形状是行为dim\_input，列为dim\_hidden\_state，而 $W_{hh}$ 是一个行列都为dim\_hidden\_state的方阵。

- 学习参数：前馈网络需要3个时刻来帮助学习一次 $W_{xh}$ ，而递归网络可以用3个时刻来帮助学习3次 $W_{xh}$ 和 $W_{hh}$ 。换句话说：所有时刻的权重矩阵都是共享的。这是递归网络相对于前馈网络而言最为突出的优势。

递归神经网络是在时间结构上存在共享特性的神经网络变体。

## 物理视角

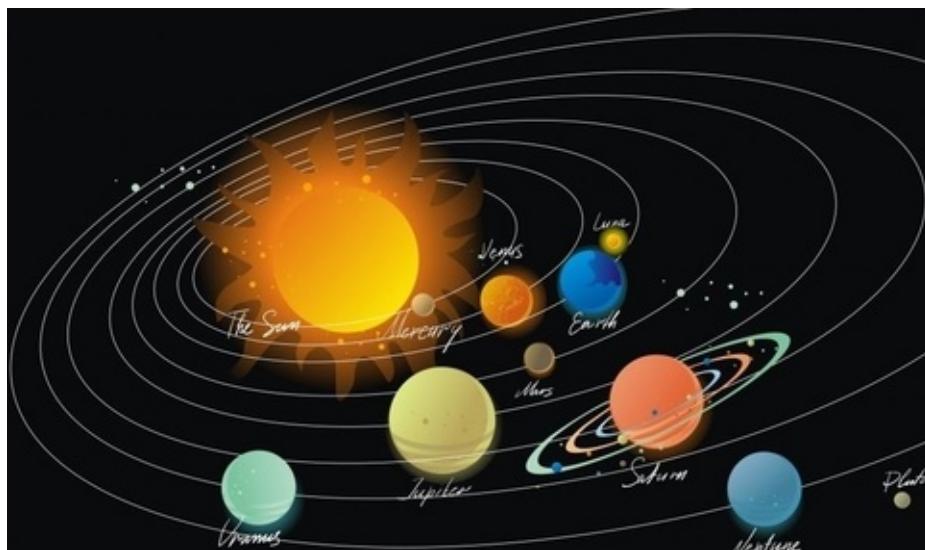
共享特性给网络带来了诸多好处，但也产生了另一个问题：

### 三、为什么可以共享？

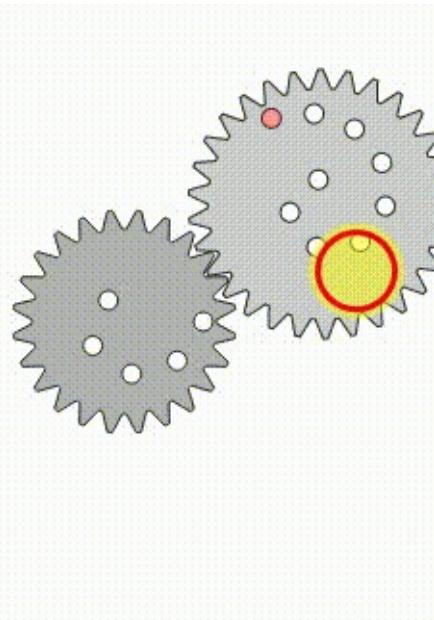
在物理视角中，YJango想给大家展示的第一点就是为什么我们可以用这种共享不同时刻权重矩阵的网络进行时序预测。

下图可以从直觉上帮助大家感受日常生活中很多时序信号是如何产生的。

- 例1：轨迹的产生，如地球的轨迹有两条线索决定，其中一条是地球自转，另一条是地球围绕太阳的公转。下图是太阳和其他星球。自转相当于 $W_{xh} \cdot x_t$ ，而公转相当于 $W_{hh} \cdot h_{t-1}$ 。二者共同决定实际轨迹。



- 例2：同理万花尺

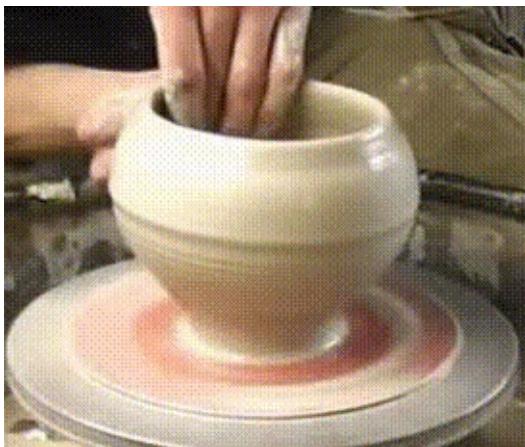


- 例3：演奏音乐时，乐器将力转成相应的震动产生声音，而整个演奏拥有一个主旋律贯穿全曲。其中乐器的物理特性就相当于 $W_{xh}$ ，同一乐器在各个时刻物理特性在各个时刻都是共享的，并不会改变。其内在也有一个隐藏的主旋律基准 $W_{hh}$ ，旋律信息 $W_{hh} \cdot h_{t-1}$ 与音乐信息 $W_{xh} \cdot x_t$ 共同决定下一时刻的实际声音。

上述例子中所产生的轨迹、音乐都是我们所能观察到的observations，我们常常会利用这些observation作为依据来做出决策。

下面的例子可能更容易体会共享特性对于数据量的影响。

- 实例：捏陶瓷：不同角度相当于不同的时刻



- 若用前馈网络：网络训练过程相当于不用转盘，而是徒手将各个角度捏成想要的形状。不仅工作量大，效果也难以保证。
- 若用递归网络：网络训练过程相当于在不断旋转的转盘上，以一种手势捏造所有角度。工作量降低，效果也可保证。

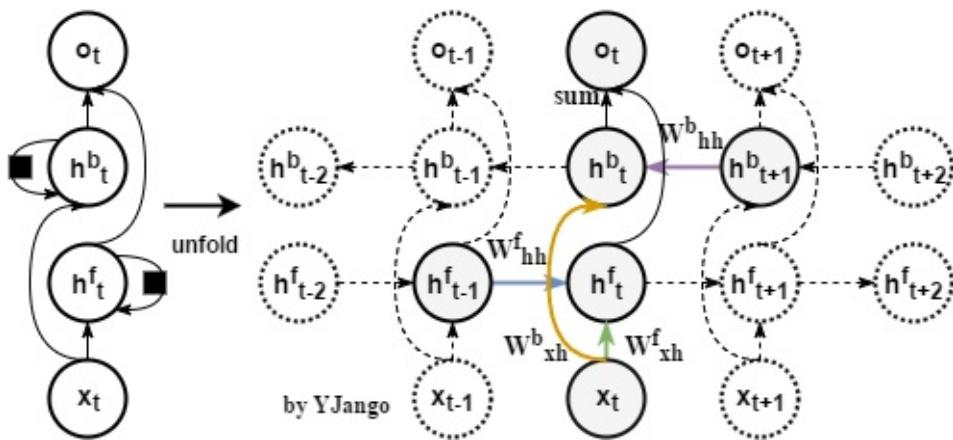
## 递归网络特点

- 时序长短可变：只要知道上一时刻的隐藏状态  $h_{t-1}$  与当前时刻的输入  $x_t$ ，就可以计算当前时刻的隐藏状态  $h_t$ 。并且由于计算所用到的  $W_{xh}$  与  $W_{hh}$  在任意时刻都是共享的。递归网络可以处理任意长度的时间序列。
- 顾及时间依赖：若当前时刻是第5帧的时序信号，那计算当前的隐藏状态  $h_5$  就需要当前的输入  $x_5$  和第4帧的隐藏状态  $h_4$ ，而计算  $h_4$  又需要  $h_3$ ，这样不断逆推到初始时刻为止。意味着常规递归网络对过去所有状态都存在着依赖关系。

注：在计算  $h_0$  的值时，若没有特别指定初始隐藏状态，则会将  $h_{-1}$  全部补零，

表达式会变成前馈神经网络：
$$h_t = \phi(W_{xh} \cdot x_t + W_{hh} \cdot h_{t-1} + b)$$

- 未来信息依赖：前馈网络是通过并接未来时刻的向量来引入未来信息对当前内容判断的限制，但常规的递归网络只对所有过去状态存在依赖关系。所以递归网络的一个扩展就是双向（bidirectional）递归网络：两个不同方向的递归层叠加。
  - 关系图：正向（forward）递归层是从最初时刻开始，而反向（backward）递归层是从最末时刻开始。



- 数学式子：

- 正向递归层： $h_t^f = \phi(W_{xh}^f \cdot x_t + W_{hh}^f \cdot h_{t-1} + b^f)$
- 反向递归层： $h_t^b = \phi(W_{xh}^b \cdot x_t + W_{hh}^b \cdot h_{t+1} + b^b)$
- 双向递归层： $h_t = h_t^f + h_t^b$

注：还有并接的处理方式，即  $h_t = concat(h_t^f, h_t^b)$ ，但反向递归层的作用是引入未来信息对当前预测判断的额外限制。并不是信息维度不够。至少在我所有的实验中，相加（sum）的方式往往优于并接。

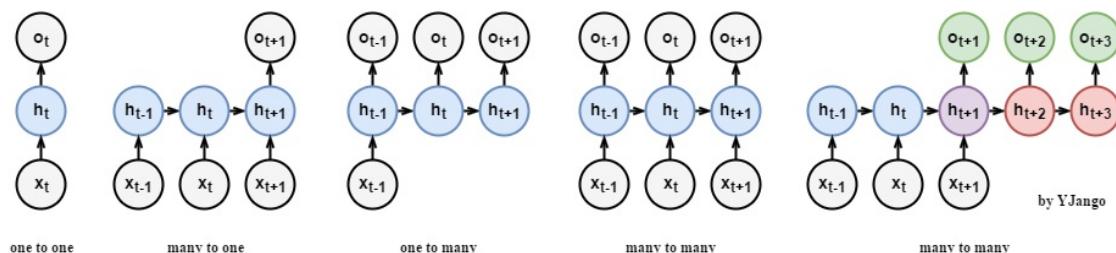
注：也有人将正向递归层和反向递归层中的权重  $W_{xh}^f$  与  $W_{xh}^b$  共享， $W_{hh}^f$  与  $W_{hh}^b$  共享。我没有做实验比较过。但直觉上  $W_{hh}^f$  与  $W_{hh}^b$  共享在某些任务中可能会有些许提升。 $W_{hh}^f$  与  $W_{hh}^b$  的共享恐怕并不会起到什么作用（要贴合任务而言）。

注：隐藏状态  $h_t$  通常不会是网络的最终结果，一般都会将  $h_t$  再接着另一个  $\phi(W_{ho} \cdot h_t + b_o)$  将其投射到输出状态  $o_t$ 。一个最基本的递归网络不会出现前馈神经网络那样从输入层直接到输出层的情况，而是至少会有一个隐藏层。

注：双向递归层可以提供更好的识别预测效果，但却不能实时预测，由于反向递归的计算需要从最末时刻开始，网络不得不等待着完整序列都产生后才可以开始预测。在对于实时识别有要求的线上语音识别，其应用受限。

- 递归网络输出：递归网络的出现实际上是对前馈网络在时间维度上的扩展。

- 关系图：常规网络可以将输入和输出以向量对向量（无时间维度）的方式进行关联。而递归层的引入将其扩展到了序列对序列的匹配。从而产生了 one to one 右侧的一系列关联方式。较为特殊的是最后一个 many to many，发生在输入输出的序列长度不确定时，其实质两个递归网络的拼接使用，公共点在紫色的隐藏状态  $h_{t+1}$ 。



- **many to one**：常用在情感分析中，将一句话关联到一个情感向量上去。
- **many to many**：第一个many to many在DNN-HMM语音识别框架中常有用到
- **many to many(variable length)**：第二个many to many常用在机器翻译两个不同语言时。
- 递归网络数据：递归网络由于引入time step的缘故，使得其训练数据与前馈网络有所不同。
  - 前馈网络：输入和输出：矩阵

输入矩阵形状： $(n\_samples, dim\_input)$

输出矩阵形状： $(n\_samples, dim\_output)$

注：真正测试/训练的时候，网络的输入和输出就是向量而已。加入 $n\_samples$ 这个维度是为了可以实现一次训练多个样本，求出平均梯度来更新权重，这个叫做Mini-batch gradient descent。

如果 $n\_samples$ 等于1，那么这种更新方式叫做Stochastic Gradient Descent (SGD)。

- 递归网络：输入和输出：维度至少是3的张量，如果是图片等信息，则张量维度仍会增加。

输入张量形状： $(time\_steps, n\_samples, dim\_input)$

输出张量形状： $(time\_steps, n\_samples, dim\_output)$

注：同样是保留了Mini-batch gradient descent的训练方式，但不同之处在于多了time step这个维度。

Recurrent 的任意时刻的输入的本质还是单个向量，只不过是将不同时刻的向量按顺序输入网络。你可能更愿意理解为一串向量 a sequence of vectors，或者是矩阵。

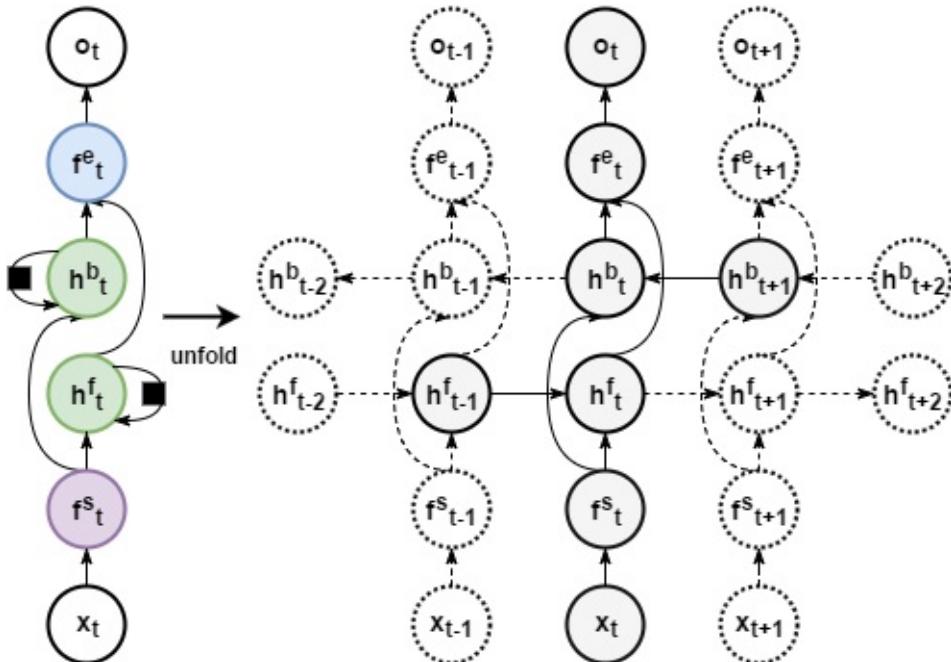
## 网络对待

请以层的概念对待所有网络。递归神经网络是指拥有递归层的神经网络，其关键在于网络中存在递归层。

每一层的作用是将数据从一个空间变换到另一个空间下。可以视为特征抓取方式，也可以视为分类器。二者没有明显界限并彼此包含。关键在于使用者如何理解。

以层的概念理解网络的好处在于，今后的神经网络往往并不会仅用到一种处理手段。往往是前馈、递归、卷积混合使用。这时就无法再以递归神经网络来命名该结构。

例：下图中就是在双向递归层的前后分别又加入了两个前馈隐藏层。也可以堆积更多的双向递归层，人们也会在其前面加入“深层”二字，提高逼格。



注：层并不是图中所画的圆圈，而是连线。圆圈所表示的是穿过各层前后的状态。

## 递归网络问题

常规递归网络从理论上应该可以顾及所有过去时刻的依赖，然而实际却无法按人们所想象工作。原因在于梯度消失（vanishing gradient）和梯度爆炸（exploding gradient）问题。下一节就是介绍Long Short Term Memory (LSTM) 和Gated Recurrent Unit (GRU)：递归网络的特别实现算法。

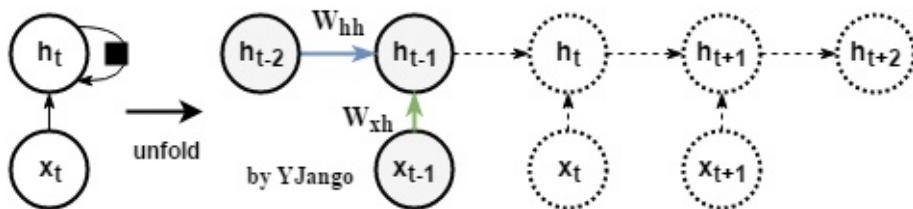
# 循环神经网络——实现

注：从《人工神经网络》后的内容就需要逐步阅读，没有相关知识而跳跃阅读是难以理解的。

## 梯度消失和梯度爆炸

网络回忆：在《循环神经网络——介绍》中提到循环神经网络用相同的方式处理每个时刻的数据。

- 动态图：



- 数学公式： $h_t = \phi(W_{xh} \cdot x_t + W_{hh} \cdot h_{t-1} + b)$

设计目的：我们希望循环神经网络可以将过去时刻发生的状态信息传递给当前时刻的计算中。

实际问题：但普通的RNN结构却难以传递相隔较远的信息。

- 考虑：若只看上图蓝色箭头线的、隐藏状态的传递过程，不考虑非线性部分，那么就会得到一个简化的式子(1)：

- (1)  $h_t = W_{hh} \cdot h_{t-1}$

如果将起始时刻的隐藏状态信息  $h_0$  向第  $t$  时刻传递，会得到式子(2)

- (2)  $h_t = (W_{hh})^t \cdot h_0$

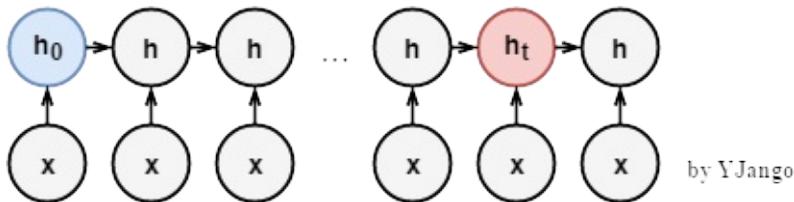
$W_{hh}$  会被乘以多次，若允许矩阵  $W_{hh}$  进行特征分解

- (3)  $W_{hh} = Q \cdot \Lambda \cdot Q^T$

式子(2)会变成(4)

- (4)  $h_t = Q \cdot \Lambda^t \cdot Q^T \cdot h_0$

当特征值小于1时，不断相乘的结果是特征值的t次方向0衰减；当特征值大于1时，不断相乘的结果是特征值的t次方向 $\infty$ 扩增。这时想要传递的 $h_0$ 中的信息会被掩盖掉，无法传递到 $h_t$ 。



- 类比：设想 $y = a^t * x$ ，如果 $a$ 等于0.1， $x$ 在被不断乘以0.1一百次后会变成多小？如果 $a$ 等于5， $x$ 在被不断乘以5一百次后会变得多大？若想要 $x$ 所包含的信息既不消失，又不爆炸，就需要尽可能的将 $a$ 的值保持在1。
- 注：更多内容请参阅[Deep Learning by Ian Goodfellow](#)中第十章。

## Long Short Term Memory (LSTM)

上面的现象可能并不意味着无法学习，但是即便可以，也会非常非常的慢。为了有效的利用梯度下降法学习，我们希望使不断相乘的梯度的积(**the product of derivatives**)保持在接近1的数值。

一种实现方式是建立线性自连接单元(linear self-connections)和在自连接部分数值接近1的权重，叫做leaky units。但Leaky units的线性自连接权重是手动设置或设为参数，而目前最有效的方式gated RNNs是通过gates的调控，允许线性自连接的权重在每一步都可以自我变化调节。LSTM就是gated RNNs中的一个实现。

### LSTM的初步理解

LSTM(或者其他gated RNNs)是在标准RNN ( $h_t = \phi(W_{xh} \cdot x_t + W_{hh} \cdot h_{t-1} + b)$ )的基础上装备了若干个控制数级(magnitude)的gates。可以理解成神经网络(RNN整体)中加入其他神经网络(gates)，而这些gates只是控制数级，控制信息的流动量。

数学公式：这里贴出基本LSTM的数学公式，看一眼就好，仅仅是为了让大家先留一个印象，不需要记住，不需要理解。

$$\begin{aligned} i_t &= \text{sigmoid}(W_{xi}x_t + W_{hi}h_{t-1} + b_i) \\ f_t &= \text{sigmoid}(W_{xf}x_t + W_{hf}h_{t-1} + b_f) \\ o_t &= \text{sigmoid}(W_{xo}x_t + W_{ho}h_{t-1} + b_o) \\ c_t &= f_t \odot c_{t-1} + i_t \odot \tanh(W_{xc}x_t + W_{hc}h_{t-1} + b_c) \\ h_t &= o_t \odot \tanh(c_t) \end{aligned}$$

尽管式子不算复杂，却包含很多知识，接下来就是逐步分析这些式子以及背后的道理。比如 $\odot$ 的意义和使用原因，sigmoid的使用原因。

## 门(gate)的理解

理解Gated RNNs的第一步就是明白**gate**到底起到什么作用。

- 物理意义：gate本身可看成是十分有物理意义的一个神经网络。
  - 输入：gate的输入是控制依据；
  - 输出：gate的输出是值域为 $(0, 1)$ 的数值，表示该如何调节其他数据的数级的控制方式。
- 使用方式：gate所产生的输出会用于控制其他数据的数级，相当于过滤器的作用。
  - 类比图：可以把信息想象成水流，而gate就是控制多少水流可以流过。



- 例如：当用gate来控制向量 $[20 \ 5 \ 7 \ 8]$ 时，
  1. 若gate的输出为 $[0.1 \ 0.2 \ 0.9 \ 0.5]$ 时，原来的向量就会被对应元素相乘(element-wise)后变成：
 
$$\begin{aligned} [20 \ 5 \ 7 \ 8] \odot [0.1 \ 0.2 \ 0.9 \ 0.5] \\ = [20 * 0.1 \ 5 * 0.2 \ 7 * 0.9 \ 8 * 0.5] = [2 \ 1 \ 6.3 \ 4] \end{aligned}$$
  2. 若gate的输出为 $[0.5 \ 0.5 \ 0.5 \ 0.5]$ 时，原来的向量就会被对应元素相乘(element-wise)后变成：
 
$$[20 \ 5 \ 7 \ 8] \odot [0.5 \ 0.5 \ 0.5 \ 0.5] = [10 \ 2.5 \ 3.5 \ 4]$$
- 控制依据：明白了gate的输出后，剩下要确定以什么信息为控制依据，也就是什么是gate的输入。

- 例如：即便是LSTM也有很多个变种。一个变种方式是调控门的输入。例如下面两种gate：

1.  $g = \text{sigmoid}(W_{xg} \cdot x_t + W_{hg} \cdot h_{t-1} + b)$  :

这种gate的输入有当前的输入 $x_t$ 和上一时刻的隐藏状态 $h_{t-1}$ ，表示gate是将这两个信息流作为控制依据而产生输出的。

2.  $g = \text{sigmoid}(W_{xg} \cdot x_t + W_{hg} \cdot h_{t-1} + W_{cg} \cdot c_{t-1} + b)$  :

这种gate的输入有当前的输入 $x_t$ 和上一时刻的隐藏状态 $h_{t-1}$ ，以及上一时刻的cell状态 $c_{t-1}$ ，表示gate是将这三个信息流作为控制依据而产生输出的。这种方式的LSTM叫做peephole connections。

## LSTM的再次理解

明白了gate之后再回过头来看LSTM的数学公式

数学公式：

$$i_t = \text{sigmoid}(W_{xi} x_t + W_{hi} h_{t-1} + b_i)$$

$$f_t = \text{sigmoid}(W_{xf} x_t + W_{hf} h_{t-1} + b_f)$$

$$o_t = \text{sigmoid}(W_{xo} x_t + W_{ho} h_{t-1} + b_o)$$

$$c_t = f_t \odot c_{t-1} + i_t \odot \tanh(W_{xc} x_t + W_{hc} h_{t-1} + b_c)$$

$$h_t = o_t \odot \tanh(c_t)$$

- **gates**：先将前半部分的三个式子 $i_t$ ， $f_t$ ， $o_t$ 统一理解。在LSTM中，网络首先构建了3个gates来控制信息的流通量。

注：虽然gates的式子构成方式一样，但是注意3个gates式子W和b的下角标并不相同。它们有各自的物理意义，在网络学习过程中会产生不同的权重。

有了这3个gates后，接下来要考虑的就是如何用它们装备在普通的RNN上来控制信息流，而根据它们所用于控制信息流通的地点不同，它们又被分为：

- 输入门 $i_t$ ：控制有多少信息可以流入memory cell（第四个式子 $c_t$ ）。
- 遗忘门 $f_t$ ：控制有多少上一时刻的memory cell中的信息可以累积到当前时刻的memory cell中。
- 输出门 $o_t$ ：控制有多少当前时刻的memory cell中的信息可以流入当前隐藏状态 $h_t$ 中。

注：gates并不提供额外信息，gates只是起到限制信息的量的作用。因为gates起到的是过滤器作用，所以所用的激活函数是sigmoid而不是tanh。

- 信息流：信息流的来源只有三处，当前的输入 $x_t$ ，上一时刻的隐藏状态 $h_{t-1}$ ，上一时刻的cell状态 $c_{t-1}$ ，其中 $c_{t-1}$ 是额外制造出来、可线性自连接的单元（请回想起leaky units）。真正的信息流来源可以说只有当前的输入 $x_t$ ，上一时刻的隐藏状态 $h_{t-1}$ 两处。三个gates的控制依据，以及数据的更新都是来源于这两处。

分析了gates和信息流后，再分析剩下的两个等式，来看LSTM是如何累积历史信息和计算隐藏状态 $h$ 的。

- 历史信息累积：

- 式子： $c_t = f_t \odot c_{t-1} + i_t \odot \tanh(W_{xc}x_t + W_{hc}h_{t-1} + b_c)$

其中 $new = \tanh(W_{xc}x_t + W_{hc}h_{t-1} + b_c)$ 是本次要累积的信息来源。

- 改写： $c_t = f_t \odot c_{t-1} + i_t \odot new$

所以历史信息的累积是并不是靠隐藏状态 $h$ 自身，而是依靠memory cell这个自连接来累积。在累积时，靠遗忘门来限制上一时刻的memory cell的信息，并靠输入门来限制新信息。并且真的达到了leaky units的思想，memory cell的自连接是线性的累积。

- 当前隐藏状态的计算：如此大费周章的最终任然是同普通RNN一样要计算当前隐藏状态。

- 式子： $h_t = o_t \odot \tanh(c_t)$

当前隐藏状态 $h_t$ 是从 $c_t$ 计算得来的，因为 $c_t$ 是以线性的方式自我更新的，所以先将其加入带有非线性功能的 $\tanh(c_t)$ 。随后再靠输出门 $o_t$ 的过滤来得到当前隐藏状态 $h_t$ 。

## 普通RNN与LSTM的比较

下面为了加深理解循环神经网络的核心，再来和YJango一起比较一下普通RNN和LSTM的区别。

- 比较公式：最大的区别是多了三个神经网络(gates)来控制数据的流通。

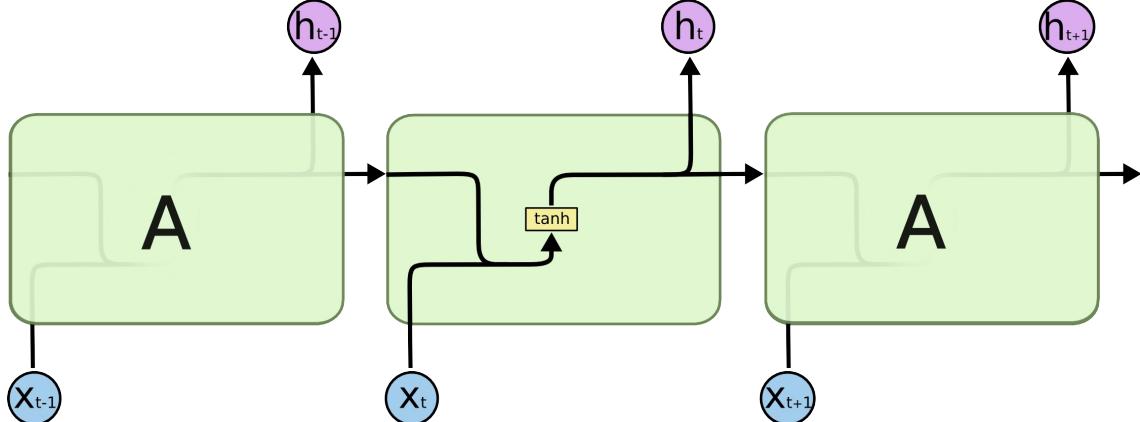
- 普通RNN： $h_t = \tanh(W_{xh} \cdot x_t + W_{hh} \cdot h_{t-1} + b)$

- LSTM：

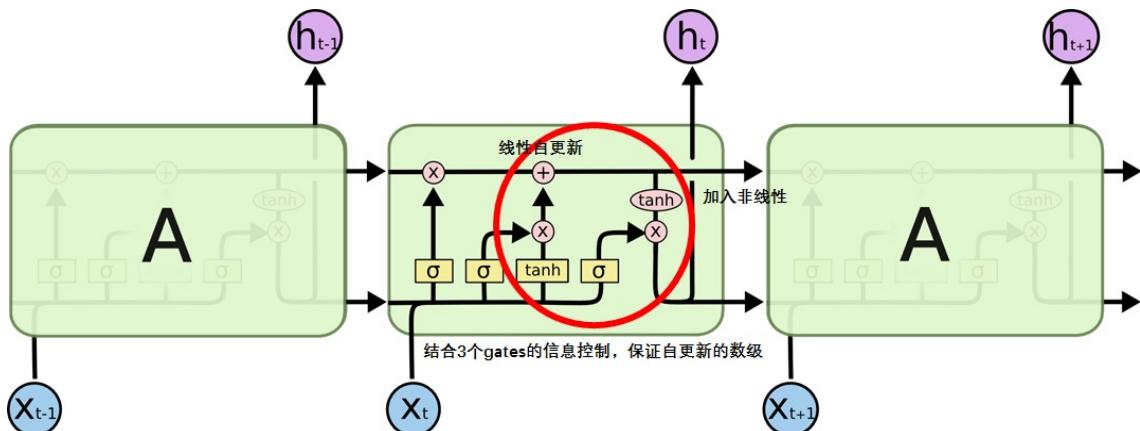
$$h_t = o_t \odot \tanh(f_t \odot c_{t-1} + i_t \odot \tanh(W_{xc}x_t + W_{hc}h_{t-1} + b_c))$$

- 比较：二者的信息来源都是  $\tanh(W_{xh} \cdot x_t + W_{hh} \cdot h_{t-1} + b)$ ，不同的是 LSTM 靠 3 个 gates 将信息的积累建立在线性自连接的 memory cell 之上，并靠其作为中间物来计算当前  $h_t$ 。
- 示图比较：图片来自 [Understanding LSTM](#)，强烈建议一并阅读。

- 普通 RNN：



- LSTM：加号圆圈表示线性相加，乘号圆圈表示用 gate 来过滤信息。



- 比较：新信息从黄色的  $\tanh$  处，线性累积到 memory cell 之中后，又从红色的  $\tanh$  处加入非线性并返回到了隐藏状态  $h_t$  的计算中。

LSTM 靠 3 个 gates 将信息的积累建立在线性自连接的权重接近 1 的 memory cell 之上，并靠其作为中间物来计算当前  $h_t$ 。

## LSTM 的类比记忆

对于用 LSTM 来实现 RNN 的记忆，可以类比我们所用的手机（仅仅是为了方便记忆，并非一一对应）。



普通RNN好比是手机屏幕，而LSTM-RNN好比是手机膜。

大量非线性累积历史信息会造成梯度消失(梯度爆炸)好比是不断使用后容易使屏幕刮花。

而LSTM将信息的积累建立在线性自连接的memory cell之上，并靠其作为中间物来计算当前  $h_t$ \*\*好比是用手机屏幕膜作为中间物来观察手机屏幕。

输入门、遗忘门、输出门的过滤作用好比是手机屏幕膜的反射率、吸收率、透射率三种性质。

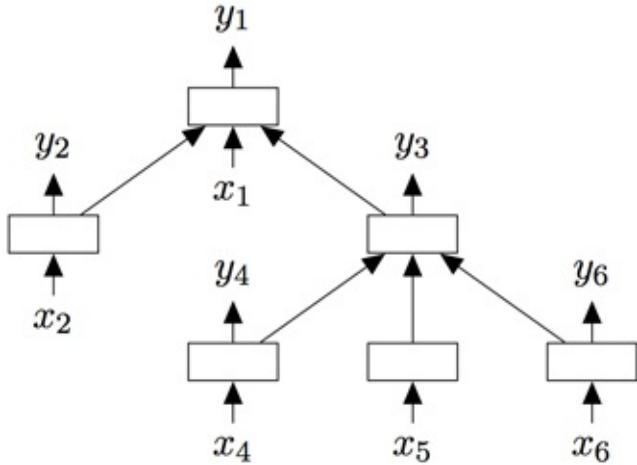
## Gated RNNs的变种

需要再次明确的是，神经网络之所以被称之为网络是因为它可以非常自由的创建合理的连接。而上面所介绍的LSTM也只是最基本的LSTM。只要遵守几个关键点，读者可以根据需求设计自己的Gated RNNs，而至于在不同任务上的效果需要通过实验去验证。下面就简单介绍YJango所理解的几个Gated RNNs的变种的设计方向。

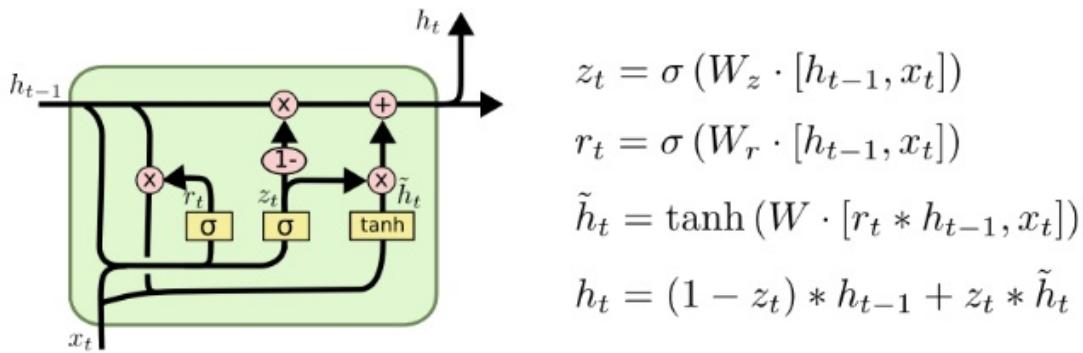
- 信息流：标准的RNN的信息流有两处：input输入和hidden state隐藏状态。

但往往信息流并非只有两处，即便是有两处，也可以拆分成多处，并通过明确多处信息流之间的结构关系来加入先验知识，减少训练所需数据量，从而提高网络效果。

例如：[Tree-LSTM](#)在具有此种结构的自然语言处理任务中的应用。



- **gates**的控制方式：与LSTM一样有名的是Gated Recurrent Unit (GRU)，而GRU使用gate的方式就与LSTM的不同，GRU只用了两个gates，将LSTM中的输入门和遗忘门合并成了更新门。并且并不把线性自更新建立在额外的memory cell上，而是直接线性累积建立在隐藏状态上，并靠gates来调控。



- **gates**的控制依据：上文所介绍的LSTM中的三个gates所使用的控制依据都是 $Wx_t + Wh_{t-1}$ ，但是可以通过与memory cell的连接来增加控制依据或者删除某个gate的 $Wx_t$ 或 $Wh_{t-1}$ 来缩减控制依据。比如去掉上图中 $z_t = \text{sigmoid}(W_z \cdot [h_{t-1}, x_t])$ 中的 $x_t$ 从而变成 $z_t = \text{sigmoid}(W_z \cdot h_{t-1})$

---

介绍完《循环神经网络——实现LSTM》后，接下来的第三篇《循环神经网络——代码》就是用tensorflow从头来实现网络内容。

# 循环神经网络——代码LV1

注：从《[人工神经网络](#)》后的内容就需要逐步阅读，没有相关知识而跳跃阅读是难以理解的。

---

上一节在《[循环神经网络——实现LSTM](#)》中介绍了循环神经网络目前最流行的实现方法LSTM和GRU，这一节就演示如何利用Tensorflow来搭建LSTM网络。代码LV1是指本次的演示是最核心的code，并没有多余的功能。为了更深刻的理解LSTM的结构，这次所用的并非是tensorflow自带的rnn\_cell类，而是重新编写，并且用scan来实现graph里的loop(动态RNN)。

## 任务描述：

这次所要学习的模型依然是[代码演示LV3](#)中的用声音来预测口腔移动，没有阅读的朋友请先阅读链接中的章节对于任务的描述。同时拿链接中的前馈神经网络与循环神经网络进行比较。

## 处理训练数据

- 目的：减掉每句数据的平均值，除以每句数据的标准差，降低模型拟合难度。
- 代码：

```

# 所需库包
import tensorflow as tf
import numpy as np
import time
import matplotlib.pyplot as plt
%matplotlib inline
# 直接使用在代码演示LV3中定义的function
def Standardize(seq):
    #subtract mean
    centered=seq-np.mean(seq, axis = 0)
    #divide standard deviation
    normalized=centered/np.std(centered, axis = 0)
    return normalized

# 读取输入和输出数据
mfc=np.load('X.npy')
art=np.load('Y.npy')
totalsamples=len(mfc)
# 20%的数据作为validation set
vali_size=0.2
# 将每个样本的输入和输出数据合成list，再将所有的样本合成list
# 其中输入数据的形状是[n_samples, n_steps, D_input]
# 其中输出数据的形状是[n_samples, D_output]
def data_prer(X, Y):
    D_input=X[0].shape[1]
    data=[]
    for x,y in zip(X,Y):
        data.append([Standardize(x).reshape((1, -1, D_input)).astype("float32"),
                    Standardize(y).astype("float32")])
    return data
# 处理数据
data=data_prer(mfc, art)
# 分训练集与验证集
train=data[int(totalsamples*vali_size):]
test=data[:int(totalsamples*vali_size)]

```

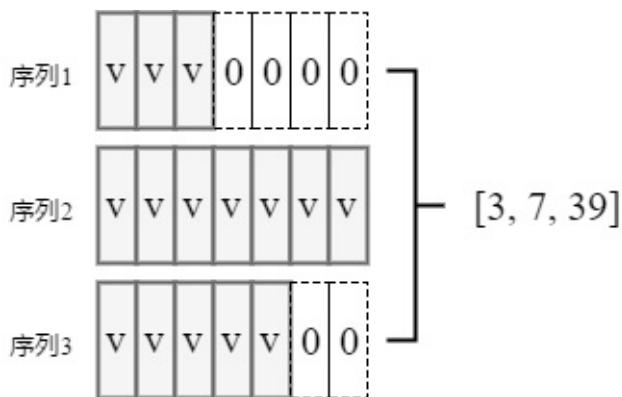
- 示意图：1，2，3，4，5表示list中的每个元素，而每个元素又是一个长度为2的list。

1	[input , ouput]
2	[input , ouput]
3	[input , ouput]
4	[input , ouput]
5	[input , ouput]

- 解释：比如全部数据有100个序列，如果设定每个input的形状就是[1, n\_steps, D\_input]，那么处理后的list的长度就是100，这样的数据使用的是SGD的更新方式。而如果想要使用mini-batch GD，将batch size(也就是n\_samples)的个数为2，那么处理后的list的长度就会是50，每次网络训练时就会同时计算两个样本的梯度并用均值来更新权重。因为每句语音数据的时间长短都不相同，如果使用3维tensor，需要大量的zero

padding，所以将n\_samples设成1。但是这样处理的缺点是：只能使用SGD，无法使用mini-batch GD。如果想使用mini-batch GD，需要几个n\_steps长度相同的样本并在一起形成3维tensor（不等长时需要zero padding，如下图）。

- 演示图：v表示一个维度为39的向量，序列1的n\_steps的长度为3，序列2的为7，如果想把这三个序列并成3维tensor，就需要选择最大的长度作为n\_steps的长度，将不足该长度的序列补零（都是0的39维的向量）。最后会形成shape为[3,7,39]的一个3维tensor。



## 权重初始化方法

- 目的：合理的初始化权重，可以降低网络在学习时卡在鞍点或极小值的损害，增加学习速度和效果
- 代码：

```

def weight_init(shape):
    initial = tf.random_uniform(shape,minval=-np.sqrt(5)*np.sqrt(1.0/shape[0]), maxv
al=np.sqrt(5)*np.sqrt(1.0/shape[0]))
    return tf.Variable(initial,trainable=True)
# 全部初始化成0
def zero_init(shape):
    initial = tf.Variable(tf.zeros(shape))
    return tf.Variable(initial,trainable=True)
# 正交矩阵初始化
def orthogonal_initializer(shape, scale = 1.0):
    #https://github.com/Lasagne/Lasagne/blob/master/lasagne/init.py
    scale = 1.0
    flat_shape = (shape[0], np.prod(shape[1:]))
    a = np.random.normal(0.0, 1.0, flat_shape)
    u, _, v = np.linalg.svd(a, full_matrices=False)
    q = u if u.shape == flat_shape else v
    q = q.reshape(shape) #this needs to be corrected to float32
    return tf.Variable(scale * q[:shape[0], :shape[1]],trainable=True, dtype=tf.flo
t32)
def bias_init(shape):
    initial = tf.constant(0.01, shape=shape)
    return tf.Variable(initial)
# 洗牌
def shufflelists(data):
    ri=np.random.permutation(len(data))
    data=[data[i] for i in ri]
    return data

```

- 解释：其中shufflelists是用于洗牌重新排序list的。正交矩阵初始化是有利gated\_rnn的学习的方法。

## 定义LSTM类

- 属性：使用class类来定义是因为LSTM中有大量的参数，定义成属性方便管理。
- 代码：在init中就将所有需要学习的权重全部定义成属性

```

class LSTMcell(object):
    def __init__(self, incoming, D_input, D_cell, initializer, f_bias=1.0):

        # var
        # incoming是用来接收输入数据的，其形状为[n_samples, n_steps, D_cell]
        self.incoming = incoming
        # 输入的维度
        self.D_input = D_input
        # LSTM的hidden state的维度，同时也是memory cell的维度
        self.D_cell = D_cell
        # parameters
        # 输入门的 三个参数
        # igate = W_xi.* x + W_hi.* h + b_i
        self.W_xi = initializer([self.D_input, self.D_cell])
        self.W_hi = initializer([self.D_cell, self.D_cell])
        self.b_i = tf.Variable(tf.zeros([self.D_cell]))
        # 遗忘门的 三个参数
        # fgate = W_xf.* x + W_hf.* h + b_f
        self.W_xf = initializer([self.D_input, self.D_cell])
        self.W_hf = initializer([self.D_cell, self.D_cell])
        self.b_f = tf.Variable(tf.constant(f_bias, shape=[self.D_cell]))
        # 输出门的 三个参数
        # ogate = W_xo.* x + W_ho.* h + b_o
        self.W_xo = initializer([self.D_input, self.D_cell])
        self.W_ho = initializer([self.D_cell, self.D_cell])
        self.b_o = tf.Variable(tf.zeros([self.D_cell]))
        # 计算新信息的三个参数
        # cell = W_xc.* x + W_hc.* h + b_c
        self.W_xc = initializer([self.D_input, self.D_cell])
        self.W_hc = initializer([self.D_cell, self.D_cell])
        self.b_c = tf.Variable(tf.zeros([self.D_cell]))

        # 最初时的hidden state和memory cell的值，二者的形状都是[n_samples, D_cell]
        # 如果没有特殊指定，这里直接设成全部为0
        init_for_both = tf.matmul(self.incoming[:, 0, :], tf.zeros([self.D_input, self.D_cell]))
        self.hid_init = init_for_both
        self.cell_init = init_for_both
        # 所以要将hidden state和memory并在一起。
        self.previous_h_c_tuple = tf.stack([self.hid_init, self.cell_init])
        # 需要将数据由[n_samples, n_steps, D_cell]的形状变成[n_steps, n_samples, D_cell]
        # 的形状
        self.incoming = tf.transpose(self.incoming, perm=[1, 0, 2])

```

- 解释：将hidden state和memory并在一起，以及将输入的形状变成[n\_steps, n\_samples, D\_cell]是为了满足tensorflow中的scan的特点，后面会提到。
- 每步计算方法：定义一个function，用于制定每一个step的计算。
- 代码：

```

def one_step(self, previous_h_c_tuple, current_x):

    # 再将hidden state和memory cell拆分开
    prev_h, prev_c = tf.unstack(previous_h_c_tuple)
    # 这时，current_x是当前的输入，
    # prev_h是上一个时刻的hidden state
    # prev_c是上一个时刻的memory cell

    # 计算输入门
    i = tf.sigmoid(
        tf.matmul(current_x, self.W_xi) +
        tf.matmul(prev_h, self.W_hi) +
        self.b_i)

    # 计算遗忘门
    f = tf.sigmoid(
        tf.matmul(current_x, self.W_xf) +
        tf.matmul(prev_h, self.W_hf) +
        self.b_f)

    # 计算输出门
    o = tf.sigmoid(
        tf.matmul(current_x, self.W_xo) +
        tf.matmul(prev_h, self.W_ho) +
        self.b_o)

    # 计算新的数据来源
    c = tf.tanh(
        tf.matmul(current_x, self.W_xc) +
        tf.matmul(prev_h, self.W_hc) +
        self.b_c)

    # 计算当前时刻的memory cell
    current_c = f * prev_c + i * c
    # 计算当前时刻的hidden state
    current_h = o * tf.tanh(current_c)

    # 再次将当前的hidden state和memory cell并在一起返回
    return tf.stack([current_h, current_c])

```

- 解释：将上一时刻的hidden state和memory拆开，用于计算后，所出现的新的当前时刻的hidden state和memory会再次并在一起作为该function的返回值，同样是为了满足scan的特点。定义该function后，LSTM就已经完成了。one\_step方法会使用LSTM类中所定义的parameters与当前时刻的输入和上一时刻的hidden state与memory cell计算当前时刻的hidden state和memory cell。
- scan**：使用scan逐次迭代计算所有timesteps，最后得出所有的hidden states进行后续的处理。
- 代码：

```

def all_steps(self):
    # 输出形状 : [n_steps, n_sample, D_cell]
    hstates = tf.scan(fn = self.one_step,
                      elems = self.incoming, #形状为[n_steps, n_sample, D_input]
                      initializer = self.previous_h_c_tuple,
                      name = 'hstates')[ :, 0, :, :]
    return hstates

```

- 解释：scan接受的fn, elems, initializer有以下要求：

- fn：第一个输入是上一时刻的输出（需要与fn的返回值保持一致），第二个输入是当前时刻的输入。
- elems：scan方法每一步都会沿着所要处理的tensor的第一个维进行一次一次取值，所以要将数据由[n\_samples, n\_steps, D\_cell]的形状变成[n\_steps, n\_samples, D\_cell]的形状。
- initializer：初始值，需要与fn的第一个输入和返回值保持一致。
- scan的返回值在上例中是[n\_steps, 2, n\_samples, D\_cell]，其中第二个维度的2是由hidden state和memory cell组成的。

## 构建网络

- 代码：

```

D_input = 39
D_label = 24
learning_rate = 7e-5
num_units=1024
# 样本的输入和标签
inputs = tf.placeholder(tf.float32, [None, None, D_input], name="inputs")
labels = tf.placeholder(tf.float32, [None, D_label], name="labels")
# 实例LSTM类
rnn_cell = LSTMcell(inputs, D_input, num_units, orthogonal_initializer)
# 调用scan计算所有hidden states
rnn0 = rnn_cell.all_steps()
# 将3维tensor [n_steps, n_samples, D_cell]转成 矩阵[n_steps*n_samples, D_cell]
# 用于计算outputs
rnn = tf.reshape(rnn0, [-1, num_units])
# 输出层的学习参数
W = weight_init([num_units, D_label])
b = bias_init([D_label])
output = tf.matmul(rnn, W) + b
# 损失
loss=tf.reduce_mean((output-labels)**2)
# 训练所需
train_step = tf.train.AdamOptimizer(learning_rate).minimize(loss)

```

- 解释：以hard coding的方式直接构建一个网络，输入是39维，第一个隐藏层也就是RNN-LSTM，1024维，而输出层又将1024维的LSTM的输出变换到24维与label对应。

注：这个网络并不仅仅取序列的最后一个值，而是要用所有timestep的值与实际轨迹进行比较计算loss

## 训练网络

- 代码：

```
# 建立session并实际初始化所有参数
sess = tf.InteractiveSession()
tf.global_variables_initializer().run()
# 训练并记录
def train_epoch(EPOCH):
    for k in range(EPOCH):
        train0=shufflelists(train)
        for i in range(len(train)):
            sess.run(train_step,feed_dict={inputs:train0[i][0],labels:train0[i][1]})
        t1=0
        d1=0
        for i in range(len(test)):
            d1+=sess.run(loss,feed_dict={inputs:test[i][0],labels:test[i][1]})
        for i in range(len(train)):
            t1+=sess.run(loss,feed_dict={inputs:train[i][0],labels:train[i][1]})

        print(k,'train:',round(t1/83,3),'test:',round(d1/20,3))
    t0 = time.time()
    train_epoch(10)
    t1 = time.time()
    print(" %f seconds" % round((t1 - t0),2))
# 训练10次后的输出和时间
(0, 'train:', 0.662, 'test:', 0.691)
(1, 'train:', 0.558, 'test:', 0.614)
(2, 'train:', 0.473, 'test:', 0.557)
(3, 'train:', 0.417, 'test:', 0.53)
(4, 'train:', 0.361, 'test:', 0.504)
(5, 'train:', 0.327, 'test:', 0.494)
(6, 'train:', 0.294, 'test:', 0.476)
(7, 'train:', 0.269, 'test:', 0.468)
(8, 'train:', 0.244, 'test:', 0.452)
(9, 'train:', 0.226, 'test:', 0.453)
563.110000 seconds
```

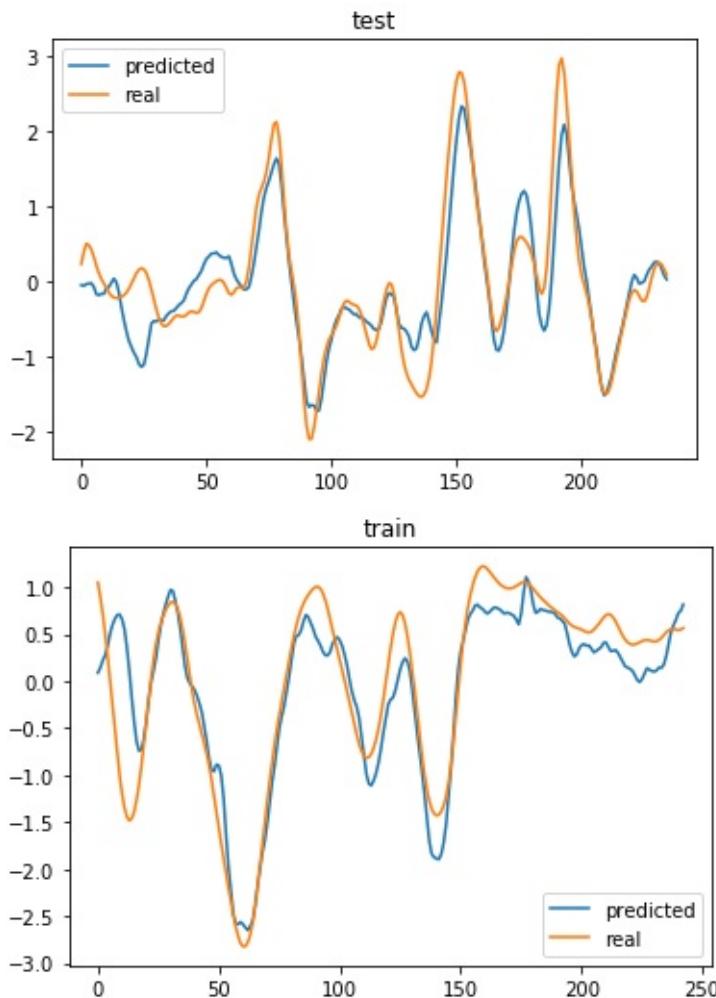
- 解释：由于上文的LSTM是非常直接的编写方式，并不高效，在实际使用中会花费较长时间。

## 预测效果

- 代码：

```
pY=sess.run(output,feed_dict={inputs:test[10][0]})  
plt.plot(pY[:,8])  
plt.plot(test[10][1][:,8])  
plt.title('test')  
plt.legend(['predicted','real'])
```

- 解释：plot出一个样本中的维度的预测效果与真是轨迹进行对比
- 效果图：



## 总结说明

该文是尽可能只展示LSTM最核心的部分，帮助大家理解其工作方式，完整代码可以从我的github中[LSTM\\_IV1](#)中找到。该LSTM由于运行效率并不高，下一篇会稍微进行改动加快运行速度，并整理结构方便使用GRU以及多层RNN的堆叠以及双向RNN。



# 循环神经网络——双向LSTM&GRU

## 任务描述：

本次的代码LV2是紧接着[代码LV1](#)的升级版，所学习的内容与先前的一样，不同的是：

- 简单梳理调整了代码结构，方便使用
- 将所有gate的计算并在一个大矩阵乘法下完成提高GPU的利用率
- 除了LSTM（Long-Short Term Memory）以外的cell，还提供了GRU（gate recurrent unit）cell模块
- 双向RNN（可选择任意cell组合）
- 该代码可被用于练习结构改造或实际建模任务

## 定义LSTMcell类

- 目的：LSTMcell包含所有学习所需要的parameters以及每一时刻所要运行的step方法
- 代码：

```
class LSTMcell(object):
    def __init__(self, incoming, D_input, D_cell, initializer,
                 f_bias=1.0, L2=False, h_act=tf.tanh,
                 init_h=None, init_c=None):
        # 属性
        self.incoming = incoming # 输入数据
        self.D_input = D_input
        self.D_cell = D_cell
        self.initializer = initializer # 初始化方法
        self.f_bias = f_bias # 遗忘门的初始偏移量
        self.h_act = h_act # 这里可以选择LSTM的hidden state的激活函数
        self.type = 'lstm' # 区分gru
        # 如果没有提供最初的hidden state和memory cell，会全部初始为0
        if init_h is None and init_c is None:
            # If init_h and init_c are not provided, initialize them
            # the shape of init_h and init_c is [n_samples, D_cell]
            self.init_h = tf.matmul(self.incoming[0,:,:], tf.zeros([self.D_input, self.D_cell]))
            self.init_c = self.init_h
            self.previous = tf.stack([self.init_h, self.init_c])
        # LSTM所有需要学习的参数
        # 每个都是[W_x, W_h, b_f]的tuple
        self.igate = self.Gate()
        self.fgate = self.Gate(bias = f_bias)
        self.ogate = self.Gate()
        self.cell = self.Gate()
```

```

# 因为所有的gate都会乘以当前的输入和上一时刻的hidden state
# 将矩阵concat在一起，计算后再逐一分离，加快运行速度
# W_x的形状是[D_input, 4*D_cell]
self.W_x = tf.concat(values=[self.igate[0], self.fgate[0], self.ogate[0], self.cell[0]], axis=1)
self.W_h = tf.concat(values=[self.igate[1], self.fgate[1], self.ogate[1], self.cell[1]], axis=1)
self.b = tf.concat(values=[self.igate[2], self.fgate[2], self.ogate[2], self.cell[2]], axis=0)
# 对LSTM的权重进行L2 regularization
if L2:
    self.L2_loss = tf.nn.l2_loss(self.W_x) + tf.nn.l2_loss(self.W_h)

# 初始化gate的函数
def Gate(self, bias = 0.001):
    # Since we will use gate multiple times, let's code a class for reusing
    Wx = self.initializer([self.D_input, self.D_cell])
    Wh = self.initializer([self.D_cell, self.D_cell])
    b = tf.Variable(tf.constant(bias, shape=[self.D_cell]), trainable=True)
    return Wx, Wh, b

# 大矩阵乘法运算完毕后，方便用于分离各个gate
def Slice_W(self, x, n):
    # split W's after computing
    return x[:, n*self.D_cell:(n+1)*self.D_cell]

# 每个time step需要运行的步骤
def Step(self, previous_h_c_tuple, current_x):
    # 分离上一时刻的hidden state和memory cell
    prev_h, prev_c = tf.unstack(previous_h_c_tuple)
    # 统一在concat成的大矩阵中一次完成所有的gates计算
    gates = tf.matmul(current_x, self.W_x) + tf.matmul(prev_h, self.W_h) + self.b
    # 分离输入门
    i = tf.sigmoid(self.Slice_W(gates, 0))
    # 分离遗忘门
    f = tf.sigmoid(self.Slice_W(gates, 1))
    # 分离输出门
    o = tf.sigmoid(self.Slice_W(gates, 2))
    # 分离新的更新信息
    c = tf.tanh(self.Slice_W(gates, 3))
    # 利用gates进行当前memory cell的计算
    current_c = f*prev_c + i*c
    # 利用gates进行当前hidden state的计算
    current_h = o*self.h_act(current_c)
    return tf.stack([current_h, current_c])

```

## 定义GRUcell类

- 代码：

```

class GRUcell(object):
    def __init__(self, incoming, D_input, D_cell, initializer, L2=False, init_h=None)

```

```

):
    # 属性
    self.incoming = incoming
    self.D_input = D_input
    self.D_cell = D_cell
    self.initializer = initializer
    self.type = 'gru'
    # 如果没有提供最初的hidden state，会初始为0
    # 注意GRU中并没有LSTM中的memory cell，其功能是由hidden state完成的
    if init_h is None:
        # If init_h is not provided, initialize it
        # the shape of init_h is [n_samples, D_cell]
        self.init_h = tf.matmul(self.incoming[0,:,:], tf.zeros([self.D_input, self.D_cell]))
    self.previous = self.init_h
    # 如果没有提供最初的hidden state，会初始为0
    # 注意GRU中并没有LSTM中的memory cell，其功能是由hidden state完成的
    self.rgate = self.Gate()
    self.ugate = self.Gate()
    self.cell = self.Gate()
    # 因为所有的gate都会乘以当前的输入和上一时刻的hidden state
    # 将矩阵concat在一起，计算后再逐一分离，加快运行速度
    # W_x的形状是[D_input, 3*D_cell]
    self.W_x = tf.concat(values=[self.rgate[0], self.ugate[0], self.cell[0]], axis=1)
    self.W_h = tf.concat(values=[self.rgate[1], self.ugate[1], self.cell[1]], axis=1)
    self.b = tf.concat(values=[self.rgate[2], self.ugate[2], self.cell[2]], axis=0)
    # 对LSTM的权重进行L2 regularization
    if L2:
        self.L2_loss = tf.nn.l2_loss(self.W_x) + tf.nn.l2_loss(self.W_h)
    # 初始化gate的函数
    def Gate(self, bias = 0.001):
        # Since we will use gate multiple times, let's code a class for reusing
        Wx = self.initializer([self.D_input, self.D_cell])
        Wh = self.initializer([self.D_cell, self.D_cell])
        b = tf.Variable(tf.constant(bias, shape=[self.D_cell]), trainable=True)
        return Wx, Wh, b
    # 大矩阵乘法运算完毕后，方便用于分离各个gate
    def Slice_W(self, x, n):
        # split W's after computing
        return x[:, n*self.D_cell:(n+1)*self.D_cell]
    # 每个time step需要运行的步骤
    def Step(self, prev_h, current_x):
        # 分两次，统一在concat成的大矩阵中完成gates所需要的计算
        Wx = tf.matmul(current_x, self.W_x) + self.b
        Wh = tf.matmul(prev_h, self.W_h)
        # 分离和组合reset gate
        r = tf.sigmoid(self.Slice_W(Wx, 0) + self.Slice_W(Wh, 0))
        # 分离和组合update gate
        u = tf.sigmoid(self.Slice_W(Wx, 1) + self.Slice_W(Wh, 1))
        # 分离和组合新的更新信息

```

```
# 注意GRU中，在这一步就已经有reset gate的干涉了
c = tf.tanh(self.Slice_W(Wx, 2) + r*self.Slice_W(Wh, 2))
# 计算当前hidden state，GRU将LSTM中的input gate和output gate的合设成1，
# 用update gate完成两者的工作
current_h = (1-u)*prev_h + u*c
return current_h
```

## 定义RNN函数

- 目的：用于接受cell的实例，并用scan计算所有time steps的hidden states
- 代码：

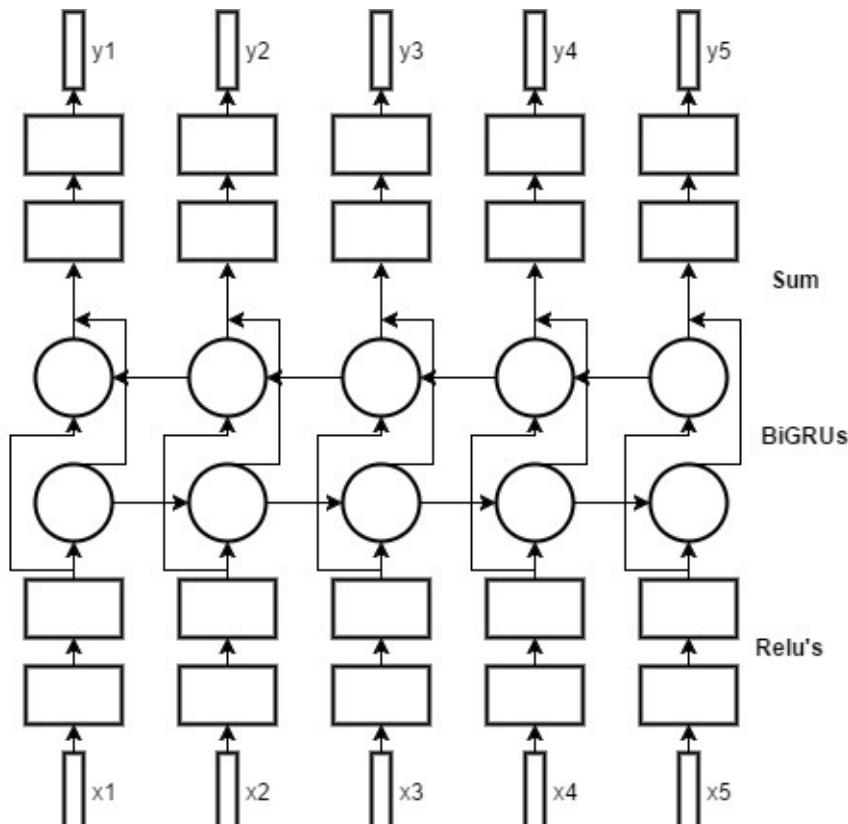
```
def RNN(cell, cell_b=None, merge='sum'):
    """
    该函数接受的数据需要是[n_steps, n_sample, D_output]，
    函数的输出也是[n_steps, n_sample, D_output].
    如果输入数据不是[n_steps, n_sample, D_input]，
    使用'inputs_T = tf.transpose(inputs, perm=[1,0,2])'.
    """

    # 正向rnn的计算
    hstates = tf.scan(fn = cell.Step,
                      elems = cell.incoming,
                      initializer = cell.previous,
                      name = 'hstates')
    # lstm的step经过scan计算后会返回4维tensor，
    # 其中[:,0,:,:]表示hidden state，
    # [:,1,:,:]表示memory cell，这里只需要hidden state
    if cell.type == 'lstm':
        hstates = hstates[:,0,:,:]
    # 如果提供了第二个cell，将进行反向rnn的计算
    if cell_b is not None:
        # 将输入数据变为反向
        incoming_b = tf.reverse(cell.incoming, axis=[0])
        # scan计算反向rnn
        b_hstates_rev = tf.scan(fn = cell_b.Step,
                               elems = incoming_b,
                               initializer = cell_b.previous, # 每个cell自带的初始值
                               name = 'b_hstates')
        if cell_b.type == 'lstm':
            b_hstates_rev = b_hstates_rev[:,0,:,:]
        # 用scan计算好的反向rnn需要再反向回来与正向rnn所计算的数据进行合并
        b_hstates = tf.reverse(b_hstates_rev, axis=[0])
        # 合并方式可以选择直接相加，也可以选择concat
        if merge == 'sum':
            hstates = hstates + b_hstates
        else:
            hstates = tf.concat(values=[hstates, b_hstates], axis=2)
    return hstates
```

- 解释：可以使用两个GRU cell进行双向rnn的就算，也可以混搭

## 网络构建

- 目的：这里演示的是两层relu feedforward layers后，接一层双向GRU-RNN，最后再接两层relu feedforward layers。
- 效果图：



- 代码：

```

D_input = 39
D_label = 24
learning_rate = 7e-5
num_units=1024
L2_penalty = 1e-4
inputs = tf.placeholder(tf.float32, [None, None, D_input], name="inputs")
labels = tf.placeholder(tf.float32, [None, D_label], name="labels")
# 保持多少节点不被dropout掉
drop_keep_rate = tf.placeholder(tf.float32, name="dropout_keep")
# 用于reshape
n_steps = tf.shape(inputs)[1]
n_samples = tf.shape(inputs)[0]
# 将输入数据从[n_samples, n_steps, D_input]，reshape成[n_samples*n_steps, D_input]
# 用于feedforward layer的使用
re1 = tf.reshape(inputs, [-1, D_input])
# 第一层

```

```

wf0 = weight_init([D_input, num_units])
bf0 = bias_init([num_units])
h1 = tf.nn.relu(tf.matmul(re1, wf0) + bf0)
# dropout
h1d = tf.nn.dropout(h1, drop_keep_rate)
# 第二层
wf1 = weight_init([num_units, num_units])
bf1 = bias_init([num_units])
h2 = tf.nn.relu(tf.matmul(h1d, wf1) + bf1)
# dropout
h2d = tf.nn.dropout(h2, drop_keep_rate)
# 将输入数据从[n_samples*n_steps, D_input]，reshape成[n_samples, n_steps, D_input]
# 用于双向rnn layer的使用
re2 = tf.reshape(h2d, [n_samples,n_steps, num_units])
# 将数据从[n_samples, n_steps, D_input]，转换成[n_steps, n_samples, D_input]
inputs_T = tf.transpose(re2, perm=[1,0,2])
# 实例rnn的正向cell，这里使用的是GRUcell
rnn_fcell = GRUcell(inputs_T, num_units, num_units, orthogonal_initializer)
# 实例rnn的反向cell
rnn_bcell = GRUcell(inputs_T, num_units, num_units, orthogonal_initializer)
# 将两个cell送给scan里计算，并使用sum的方式合并两个方向所计算的数据
rnn0 = RNN(rnn_fcell, rnn_bcell)
# 将输入数据从[n_samples, n_steps, D_input]，reshape成[n_samples*n_steps, D_input]
# 用于feedforward layer的使用
rnn1 = tf.reshape(rnn0, [-1, num_units])
# dropout
rnn2 = tf.nn.dropout(rnn1, drop_keep_rate)
# 第三层
w0 = weight_init([num_units, num_units])
b0 = bias_init([num_units])
rnn3 = tf.nn.relu(tf.matmul(rnn2, w0) + b0)
rnn4 = tf.nn.dropout(rnn3, drop_keep_rate)
# 第四层
w1 = weight_init([num_units, num_units])
b1 = bias_init([num_units])
rnn5 = tf.nn.relu(tf.matmul(rnn4, w1) + b1)
rnn6 = tf.nn.dropout(rnn5, drop_keep_rate)
# 输出层
w = weight_init([num_units, D_label])
b = bias_init([D_label])
output = tf.matmul(rnn6, w) + b
# loss
loss=tf.reduce_mean((output-labels)**2)
L2_total = tf.nn.l2_loss(wf0) + tf.nn.l2_loss(wf1)+ tf.nn.l2_loss(w0) + tf.nn.l2_loss(
w1) + tf.nn.l2_loss(w)#+ rnn_fcell.L2_loss + rnn_bcell.L2_loss
# 训练所需的
train_step = tf.train.AdamOptimizer(learning_rate).minimize(loss + L2_penalty*L2_total
)

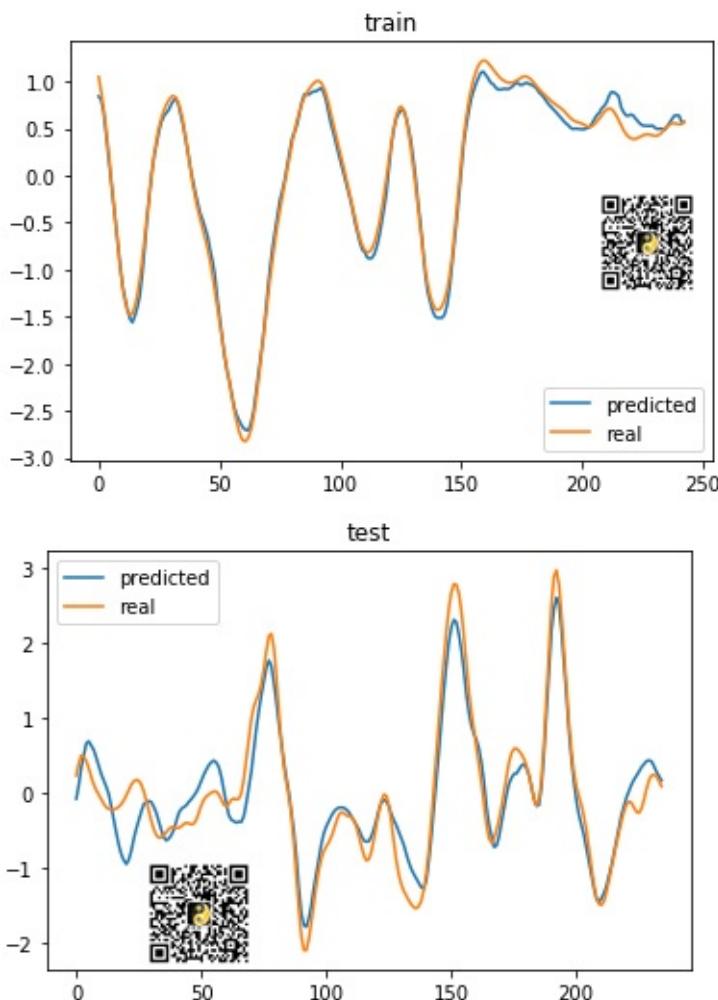
```

## 训练

剩下的代码就和[代码LV1](#)的相同了，大家可以结合[tensorboard](#)来记录和分析所学习的权重矩阵和loss的下降等。使用方式请参考[代码演示LV3](#)，也可以和[代码演示LV3](#)中的代码结合着使用，根据自己的需要注意[reshape](#)和[transpose](#)即可。完整代码在我的[github](#)上。

## 效果

- loss：训练集的loss在0.022，验证集的loss在**0.222**，比feedforward要好很多
- 效果图：另外预测的轨迹也十分的平滑



## 其他

- 速度：将所有gates的参数并在一起处理再分离可以节省很多时间，代价自然是更多的memory
- rnn的dropout：每个gate其实也是一个再cell内部的具有物理意义的神经网络，那么同样也是可以利用dropout来防止gate在拟合物理意义时过拟合。
- 多层rnn：可以rnn层之后以相同的方式再来一层双向或单向rnn，比如：

```
# 第一层双向rnn
rnn_fcell = GRUcell(inputs_T, num_units, num_units, orthogonal_initializer)
rnn_bcell = GRUcell(inputs_T, num_units, num_units, orthogonal_initializer)
rnn0 = RNN(rnn_fcell, rnn_bcell)
# 第二层双向rnn
rnn_fcell2 = GRUcell(inputs_T, num_units, num_units, orthogonal_initializer)
rnn_bcell2 = GRUcell(inputs_T, num_units, num_units, orthogonal_initializer)
rnn02 = RNN(rnn_fcell2, rnn_bcell2)
# 后续处理
rnn1 = tf.reshape(rnn02, [-1, num_units])
```

两层双向GRU-RNN的loss会达到: (190, 'train:', 0.024, 'test:', **0.193**)

- 图片里的二维码：并没有搞公众号的打算，只是用来发布些链接，让那些看到不贴原地址的恶意转载的朋友们也可以看到先前的内容（RNN之后的内容就无法从零开始阅读了，必须要有之前的知识作为铺垫）

# 卷积神经网络——介绍

阅读前请刷新页面以免公式无法显示。

关于卷积神经网络的讲解，网上有很多精彩文章，且恐怕难以找到比[斯坦福的CS231n](#)还要全面的教程。所以这里对卷积神经网络的讲解主要是以不同的思考侧重展开，通过对卷积神经网络的分析，进一步理解神经网络变体中“因素共享”这一概念。

读该文需要有本书前些章节作为预备知识，不然会有理解障碍。没看过前面内容的朋友建议看[公开课视频：深层神经网络设计理念](#)。当中的知识可以更好的帮助理解该文。

---

如果要提出一个新的神经网络结构，首先就需要引入像[循环神经网络](#)中“时间共享”这样的先验知识，降低学习所需要的训练数据需求量。而卷积神经网络同样也引入了这样的先验知识：“空间共享”。下面就让我们以画面识别作为切入点，看看该先验知识是如何被引入到神经网络中的。

---

## 目录

- 视觉感知
  - 画面识别是什么
  - 识别结果取决于什么
- 图像表达
  - 画面识别的输入
  - 画面不变形
- 前馈神经网络做画面识别的不足
- 卷积神经网络做画面识别
  - 局部连接
  - 空间共享
  - 输出空间表达
  - Depth维的处理
  - Zero padding
  - 形状、概念抓取
  - 多filters
  - 非线性
  - 输出尺寸控制
  - 矩阵乘法执行卷积

- Max pooling
  - 全连接层
  - 结构发展
  - 画面不变性的满足
    - 平移不变性
    - 旋转和视角不变性
    - 尺寸不变性
    - Inception的理解
    - 1x1卷积核理解
    - 跳层连接ResNet
- 

## 视觉感知

### 一、画面对识别是什么任务？

学习知识的第一步就是明确任务，清楚该知识的输入输出。卷积神经网络最初是服务于画面对识别的，所以我们先来看看画面对识别的实质是什么。

先观看几组动物与人类视觉的差异对比图。

#### 1. 苍蝇的视觉和人的视觉的差异



## 2. 蛇的视觉和人的视觉的差异



(更多对比图请参考[链接](#))

通过上面的两组对比图可以知道，即便是相同的图片经过不同的视觉系统，也会得到不同的感知。

这里引出一条知识：生物所看到的景象并非世界的原貌，而是长期进化出来的适合自己生存环境的一种感知方式。蛇的猎物一般是夜间行动，所以它就进化出了一种可以在夜间也能很好观察的感知系统，感热。

任何视觉系统都是将图像反光与脑中所看到的概念进行关联。



所以画面识别实际上并非识别这个东西客观上是什么，而是寻找人类的视觉关联方式，并再次应用。如果我们不是人类，而是蛇类，那么画面识别所寻找的就和现在的不一样。

画面识别实际上是寻找（学习）人类的视觉关联方式，并再次应用。

## 二、图片被识别成什么取决于哪些因素？

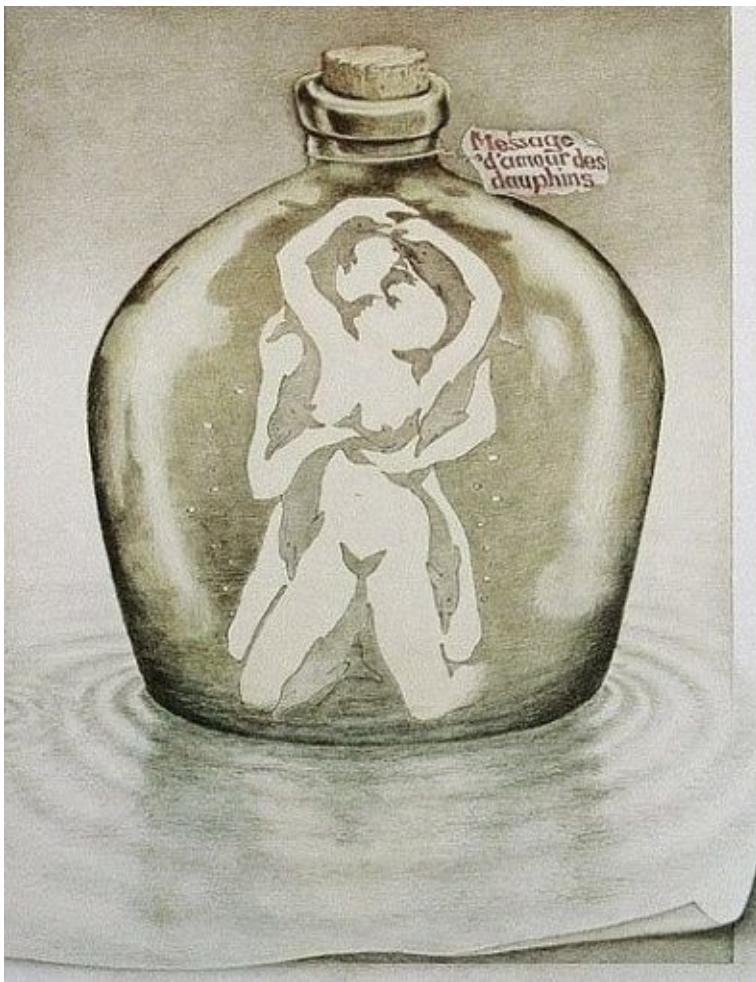
下面用两张图片来体会识别结果取决于哪些因素。

### 1. 老妇与少女



请观察上面这张图片，你看到的是老妇还是少女？以不同的方式去观察这张图片会得出不同的答案。图片可以观察成有大鼻子、大眼睛的老妇。也可以被观察成少女，但这时老妇的嘴会被识别成少女脖子上的项链，而老妇的眼睛则被识别为少女的耳朵。

### 2. 海豚与男女



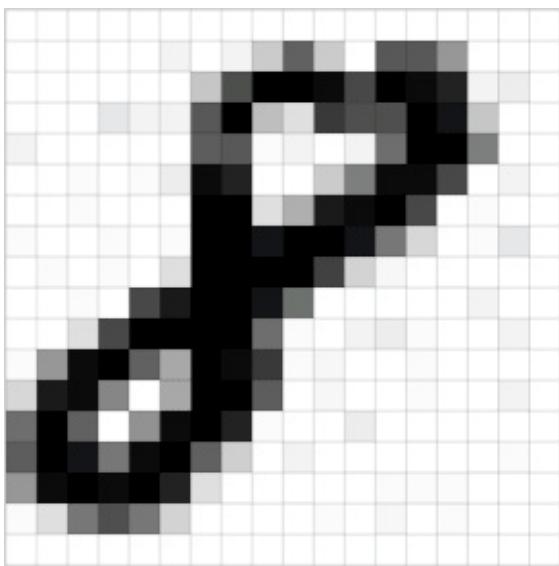
上面这张图片如果是成人观察，多半看到的会是一对亲热的男女。倘若儿童看到这张图片，看到的则会是一群海豚（男女的轮廓是由海豚构造出的）。所以，识别结果受年龄，文化等因素的影响，换句话说：

图片被识别成什么不仅仅取决于图片本身，还取决于图片是如何被观察的。

## 图像表达

我们知道了“画面识别是从大量的 $(x, y)$ 数据中寻找人类的视觉关联方式，并再次应用。其中 $x$ 是输入，表示所看到的东西。 $y$ 输出，表示该东西是什么。

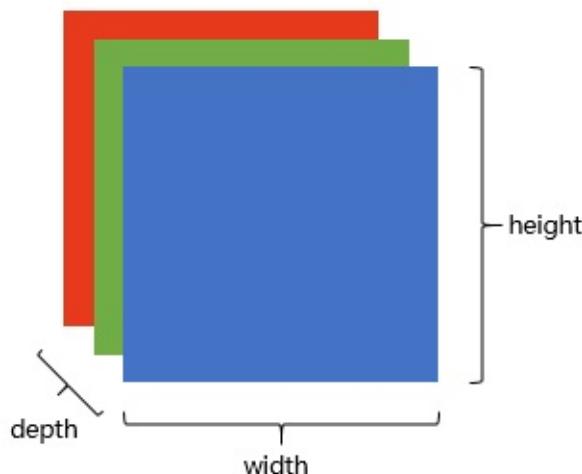
在自然界中， $x$ 是物体的反光，那么在计算机中，图像又是如何被表达和存储的呢？



图像在计算机中是一堆按顺序排列的数字，数值为0到255。0表示最暗，255表示最亮。你可以把这堆数字用一个长长的向量来表示，也就是[tensorflow的mnist教程中784维向量的表示方式](#)。然而这样会失去平面结构的信息，为保留该结构信息，通常选择矩阵的表示方式：28x28的矩阵。

上图是只有黑白颜色的灰度图，而更普遍的图片表达方式是RGB颜色模型，即红（Red）、绿（Green）、蓝（Blue）三原色的色光以不同的比例相加，以产生多种多样的色光。

这样，RGB颜色模型中，单个矩阵就扩展成了有序排列的三个矩阵，也可以用三维张量去理解，其中的每一个矩阵又叫这个图片的一个channel。



在电脑中，一张图片是数字构成的“长方体”。可用 宽**width**，高**height**，深**depth** 来描述，如上图。

画面识别的输入 $x$ 是**shape**为(**width**, **height**, **depth**)的三维张量。

接下来要考虑的就是该如何处理这样的“数字长方体”。

## 画面不变性

在决定如何处理“数字长方体”之前，需要清楚所建立的网络拥有什么样的特点。我们知道一个物体不管在画面左侧还是右侧，都会被识别为同一物体，这一特点就是不变性（invariance），如下图所示。

Translation Invariance



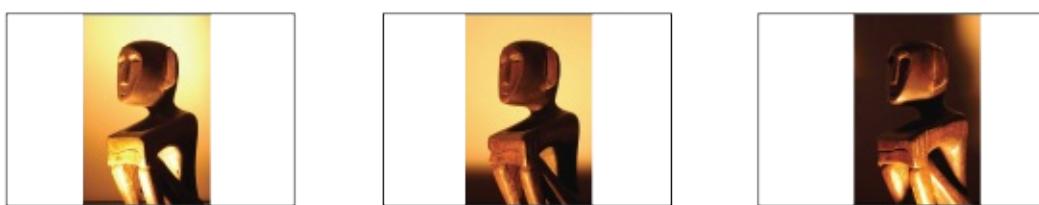
Rotation/Viewpoint Invariance



Size Invariance



Illumination Invariance



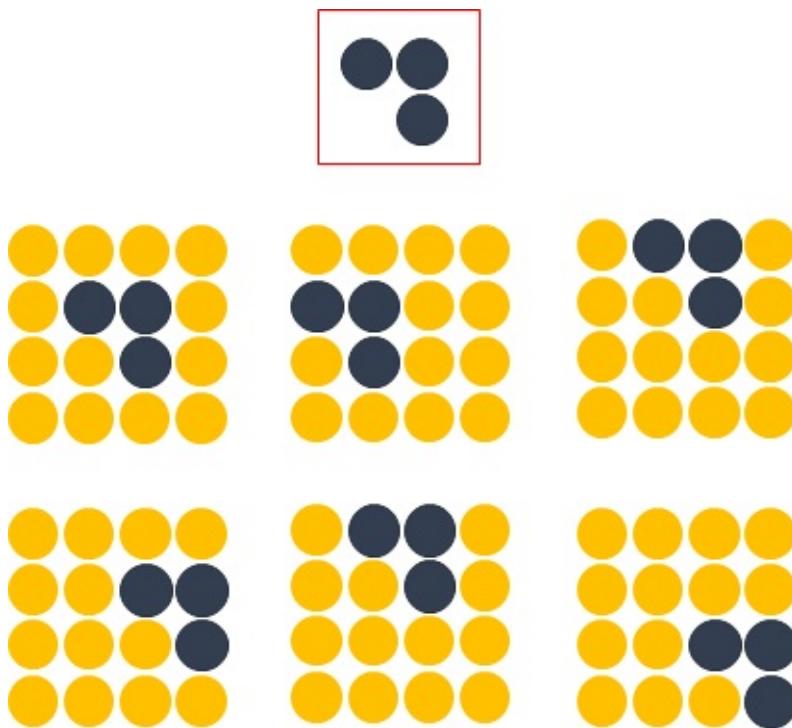
Matt Krause  
[mattkrause.se](http://mattkrause.se)

我们希望所建立的网络可以尽可能的满足这些不变性特点。

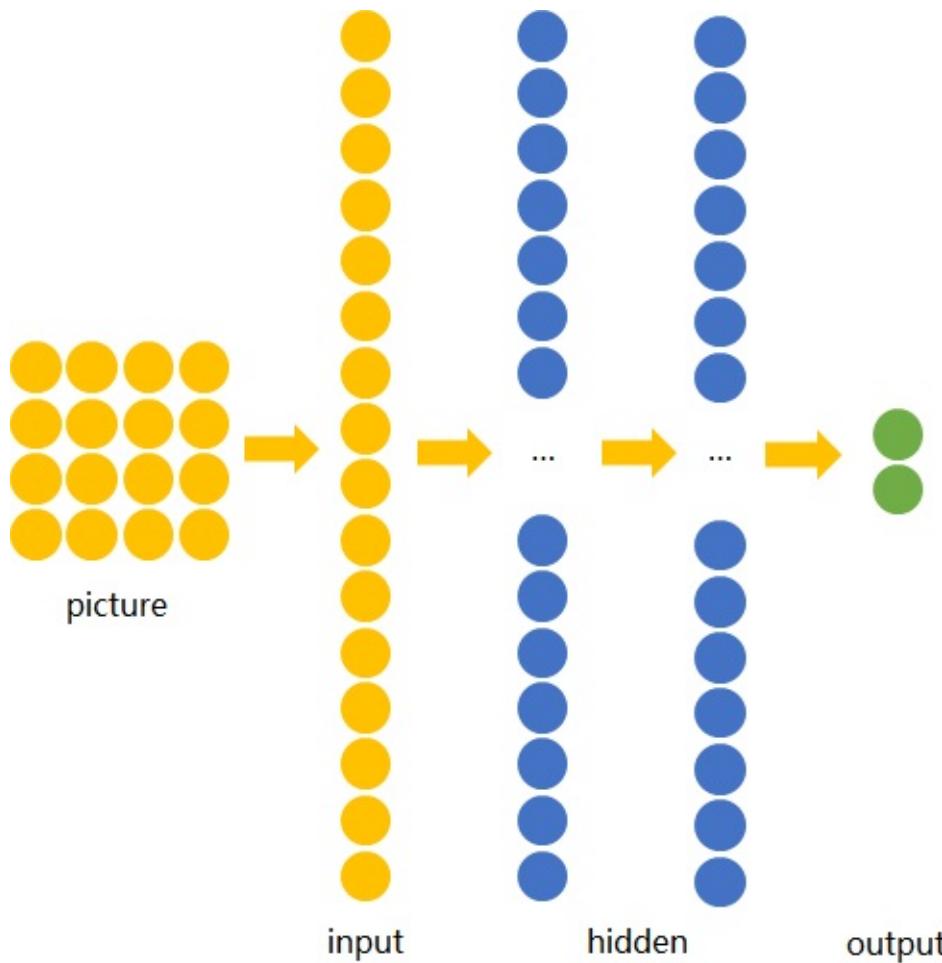
为了理解卷积神经网络对这些不变性特点的贡献，我们将用不具备这些不变性特点的前馈神经网络来进行比较。

## 图片识别--前馈神经网络

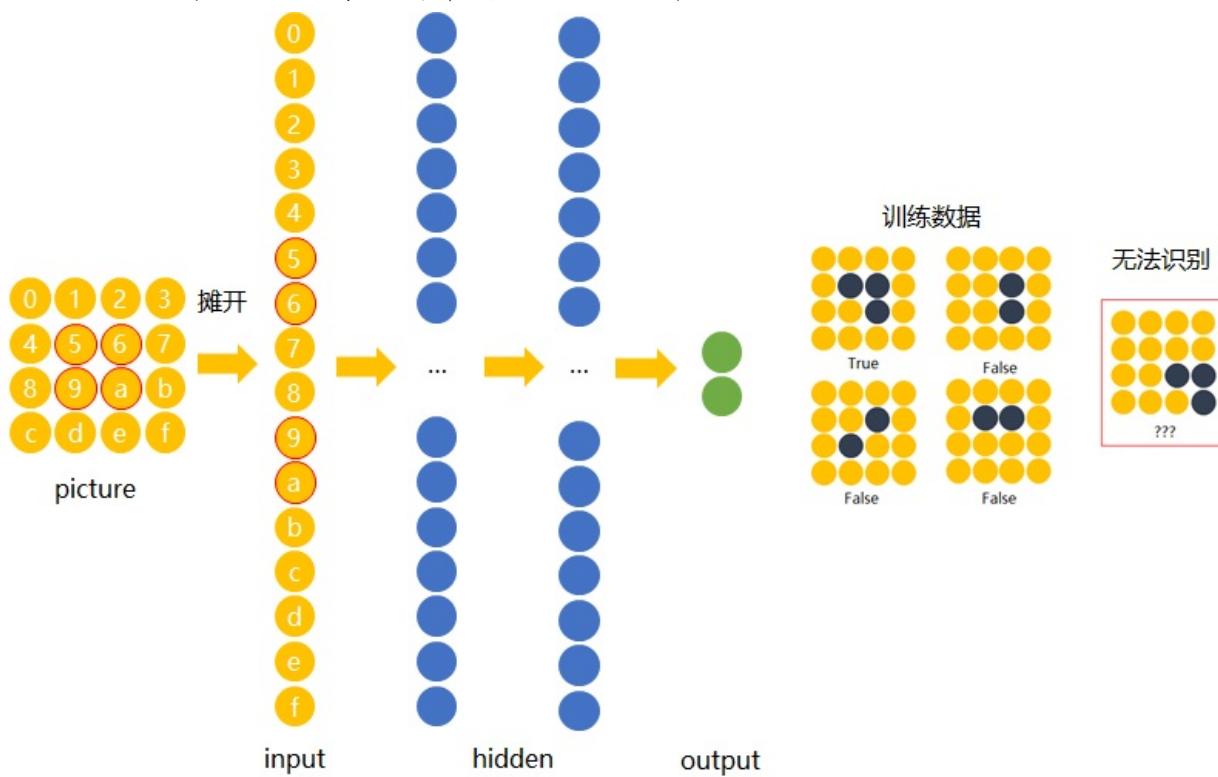
方便起见，我们用**depth**只有1的灰度图来举例。想要完成的任务是：在宽长为 $4 \times 4$ 的图片中识别是否有下图所示的“横折”。图中，黄色圆点表示值为0的像素，深色圆点表示值为1的像素。我们知道不管这个横折在图片中的什么位置，都会被认为是相同的横折。



若训练前馈神经网络来完成该任务，那么表达图像的三维张量将会被摊平成一个向量，作为网络的输入，即(**width**, **height**, **depth**)为(4, 4, 1)的图片会被展成维度为16的向量作为网络的输入层。再经过几层不同节点个数的隐藏层，最终输出两个节点，分别表示“有横折的概率”和“没有横折的概率”，如下图所示。



下面我们用数字（16进制）对图片中的每一个像素点（pixel）进行编号。当使用右侧那种物体位于中间的训练数据来训练网络时，网络就只会对编号为5,6,9,a的节点的权重进行调节。若让该网络识别位于右下角的“横折”时，则无法识别。



解决办法是用大量物体位于不同位置的数据训练，同时增加网络的隐藏层个数从而扩大网络学习这些变体的能力。

然而这样做十分不效率，因为我们知道在左侧的“横折”也好，还是在右侧的“横折”也罢，大家都是“横折”。为什么相同的东西在位置变了之后要重新学习？有没有什么方法可以将中间所学到的规律也运用在其他的位置？换句话说，也就是让不同位置用相同的权重。

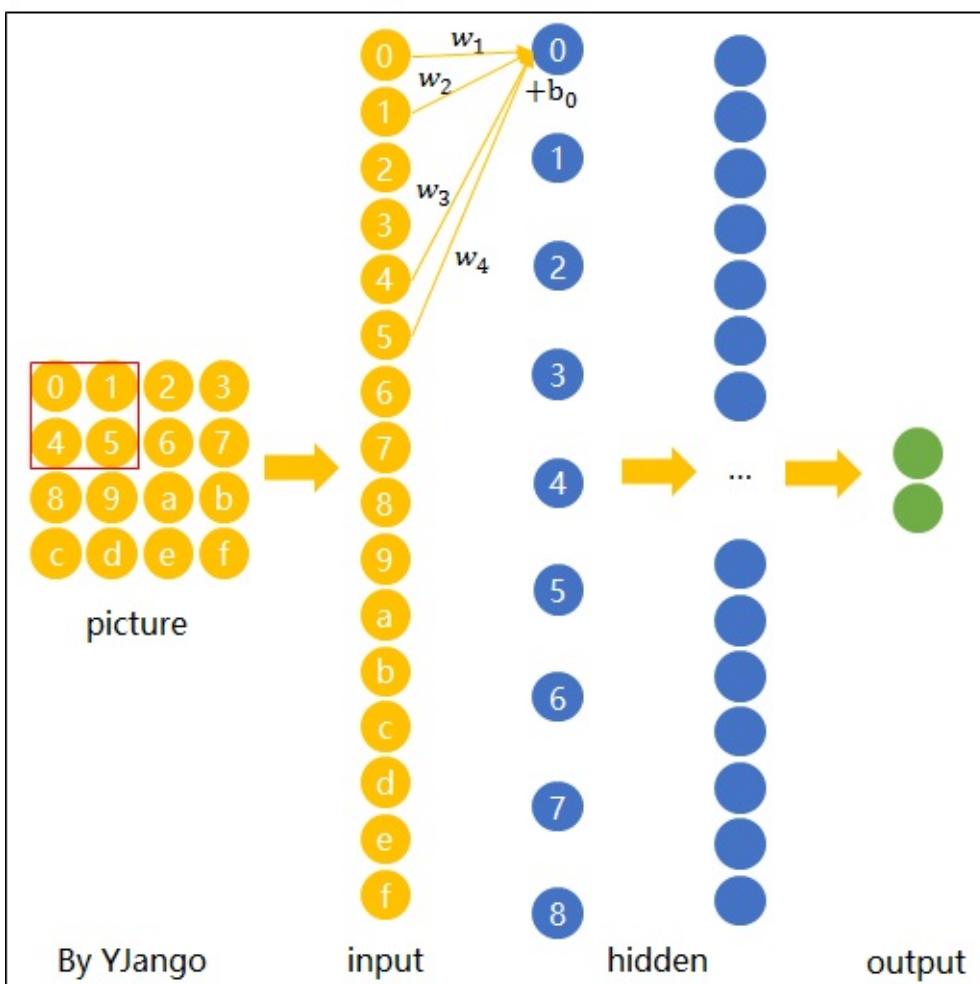
## 图片识别--卷积神经网络

卷积神经网络就是让权重在不同位置共享的神经网络。

### 局部连接

在卷积神经网络中，我们先选择一个局部区域，用这个局部区域去扫描整张图片。局部区域所圈起来的所有节点会被连接到下一层的一个节点上。

为了更好的和前馈神经网络做比较，我将这些以矩阵排列的节点展成了向量。下图展示了被红色方框所圈中编号为 $0, 1, 4, 5$ 的节点是如何通过 $w_1, w_2, w_3, w_4$ 连接到下一层的节点0上的。



By YJango

input

hidden

output

这个带有连接强弱的红色方框就叫做 **filter** 或 **kernel** 或 **feature detector**。而filter的范围叫做**filter size**，这里所展示的是 $2 \times 2$ 的filter size。

$$\begin{bmatrix} w_1 & w_2 \\ w_3 & w_4 \end{bmatrix} \quad (1)$$

第二层的节点 $0$ 的数值就是局部区域的线性组合，即被圈中节点的数值乘以对应的权重后相加。用 $x$ 表示输入值， $y$ 表示输出值，用图中标注数字表示角标，则下面列出了两种计算编号为 $0$ 的输出值 $y_0$ 的表达式。

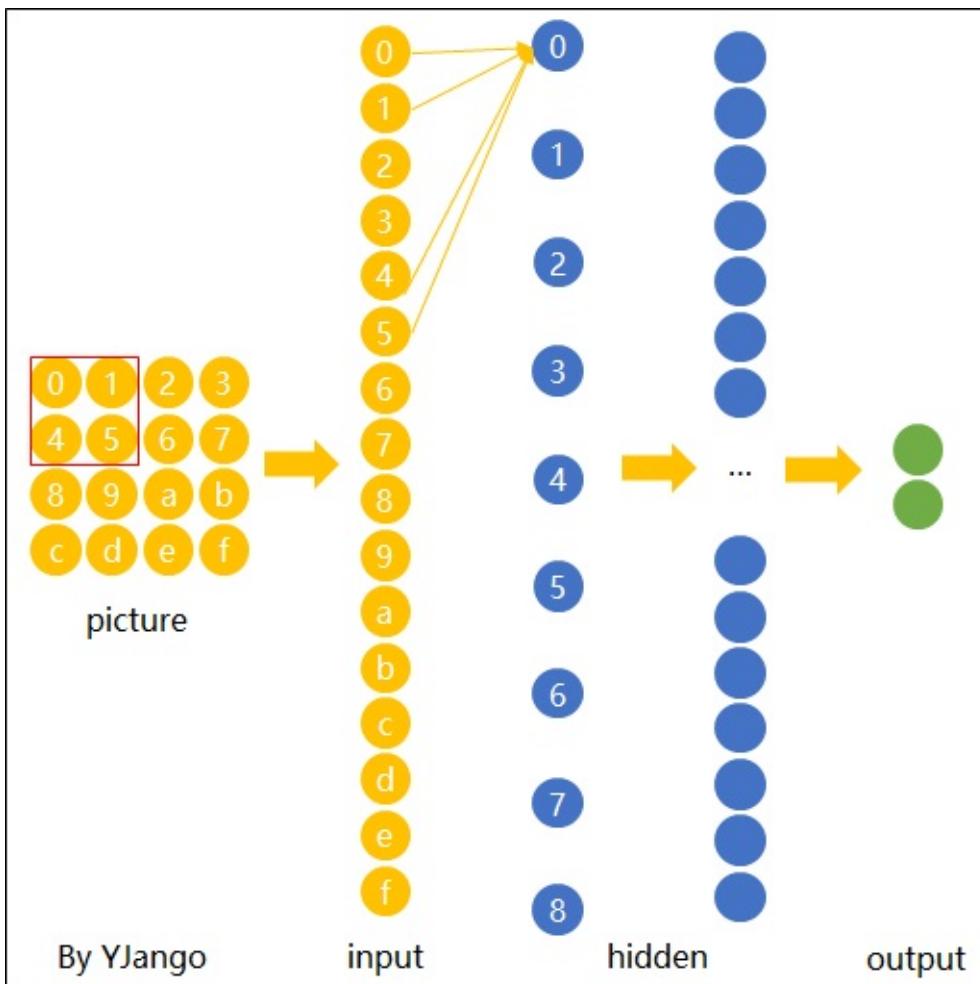
注：在局部区域的线性组合后，也会和前馈神经网络一样，加上一个偏移量 $b_0$ 。

$$\begin{aligned} y_0 &= x_0 * w_1 + x_1 * w_2 + x_4 * w_3 + x_5 * w_4 + b_0 \\ y_0 &= [w_1 \quad w_2 \quad w_3 \quad w_4] \cdot \begin{bmatrix} x_0 \\ x_1 \\ x_4 \\ x_5 \end{bmatrix} + b_0 \end{aligned} \quad (2)$$

## 空间共享

当filter扫到其他位置计算输出节点 $y_i$ 时， $w_1, w_2, w_3, w_4$ ，包括 $b_0$ 是共用的。

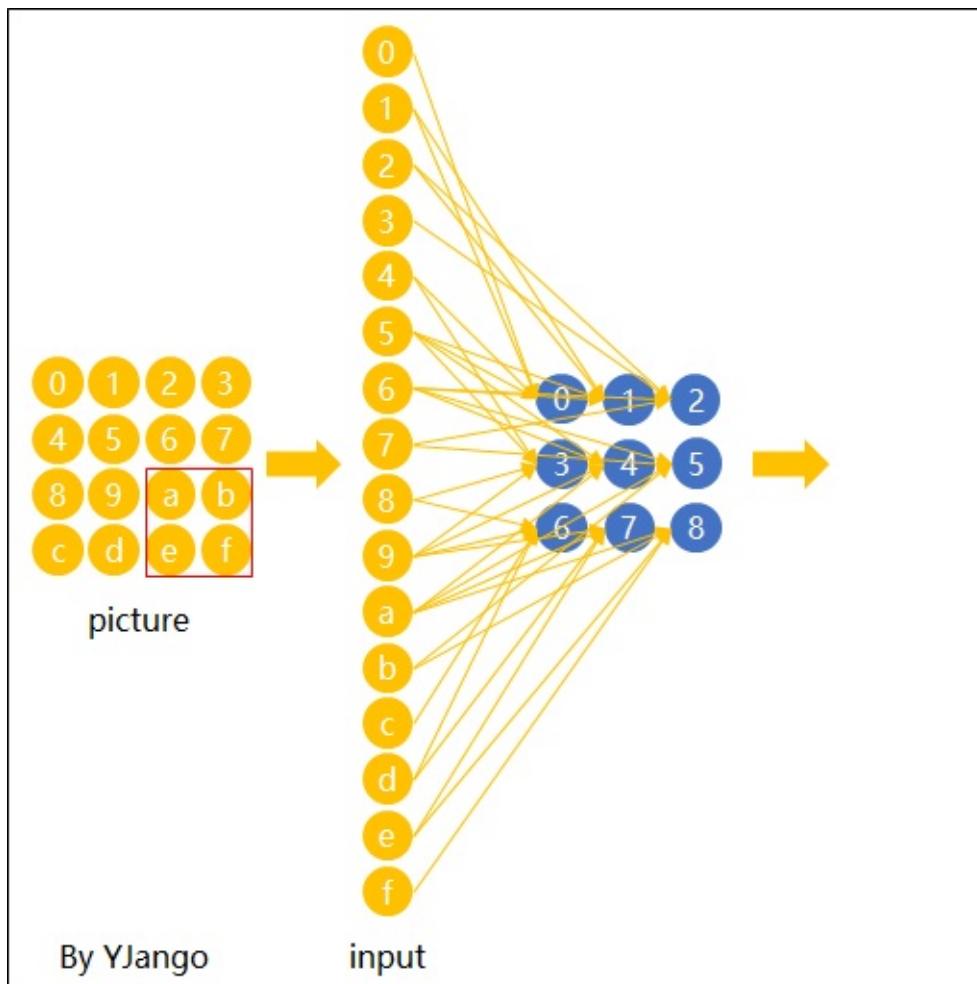
下面这张动态图展示了当filter扫过不同区域时，节点的链接方式。动态图的最后一帧则显示了所有连接。可以注意到，每个输出节点并非像前馈神经网络中那样与全部的输入节点连接，而是部分连接。这也就是为什么大家也叫前馈神经网络（feedforward neural network）为fully-connected neural network。图中显示的是一步一步的移动filter来扫描全图，一次移动多少叫做stride。



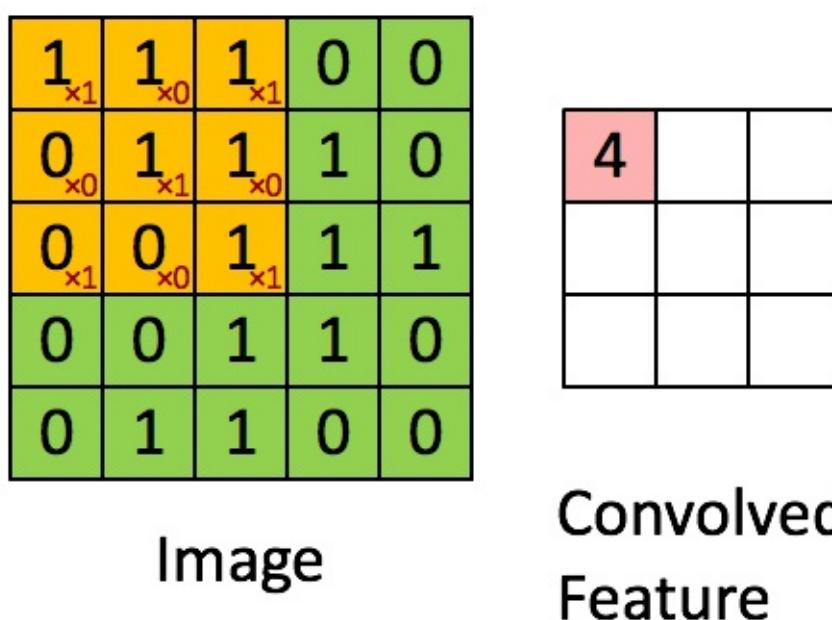
空间共享也就是卷积神经网络所引入的先验知识。

## 输出表达

如先前在图像表达中提到的，图片不用向量去表示是为了保留图片平面结构的信息。同样的，卷积后的输出若用上图的排列方式则丢失了平面结构信息。所以我们依然用矩阵的方式排列它们，就得到了下图所展示的连接。



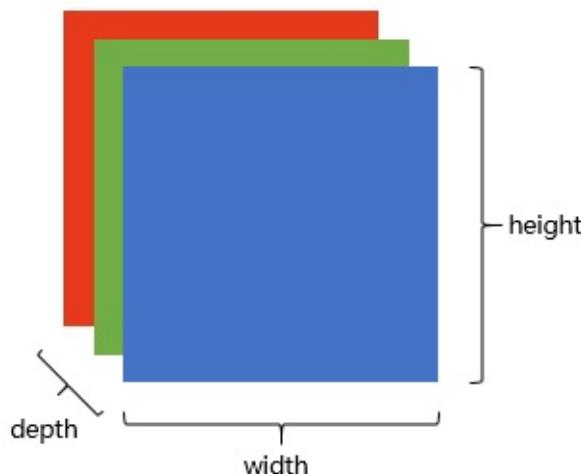
这也就是你们在网上所看到的下面这张图。在看这张图的时候请结合上图的连接一起理解，即输入（绿色）的每九个节点连接到输出（粉红色）的一个节点上的。



经过一个feature detector计算后得到的粉红色区域也叫做一个“**Convolved Feature**”或“**Activation Map**”或“**Feature Map**”。

## Depth维的处理

现在我们已经知道了depth维度只有1的灰度图是如何处理的。但前文提过，图片的普遍表达方式是下图这样有3个channels的RGB颜色模型。当depth为复数的时候，每个feature detector是如何卷积的？



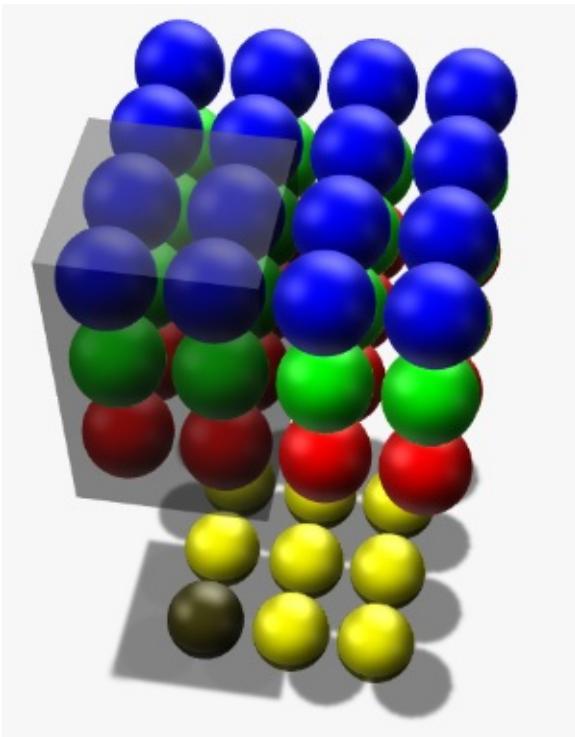
现象： $2 \times 2$ 所表达的filter size中，一个2表示width维上的局部连接数，另一个2表示height维上的局部连接数，并却没有depth维上的局部连接数，是因为depth维上并非局部，而是全部连接的。

在2D卷积中，filter在张量的width维, height维上是局部连接，在depth维上是贯穿全部channels的。

类比：想象在切蛋糕的时候，不管这个蛋糕有多少层，通常大家都会一刀切到底，但是在长和宽这两个维上是局部切割。

下面这张图展示了，在depth为复数时，filter是如何连接输入节点到输出节点的。图中红、绿、蓝颜色的节点表示3个channels。黄色节点表示一个feature detector卷积后得到的Feature Map。其中被透明黑框圈中的12个节点会被连接到黄黑色的节点上。

- 在输入depth为1时：被filter size为 $2 \times 2$ 所圈中的4个输入节点连接到1个输出节点上。
- 在输入depth为3时：被filter size为 $2 \times 2$ ，但是贯穿3个channels后，所圈中的12个输入节点连接到1个输出节点上。
- 在输入depth为 $n$ 时： $2 \times 2 \times n$ 输入节点连接到1个输出节点上。



(可从[vectary](#)在3D编辑下查看)

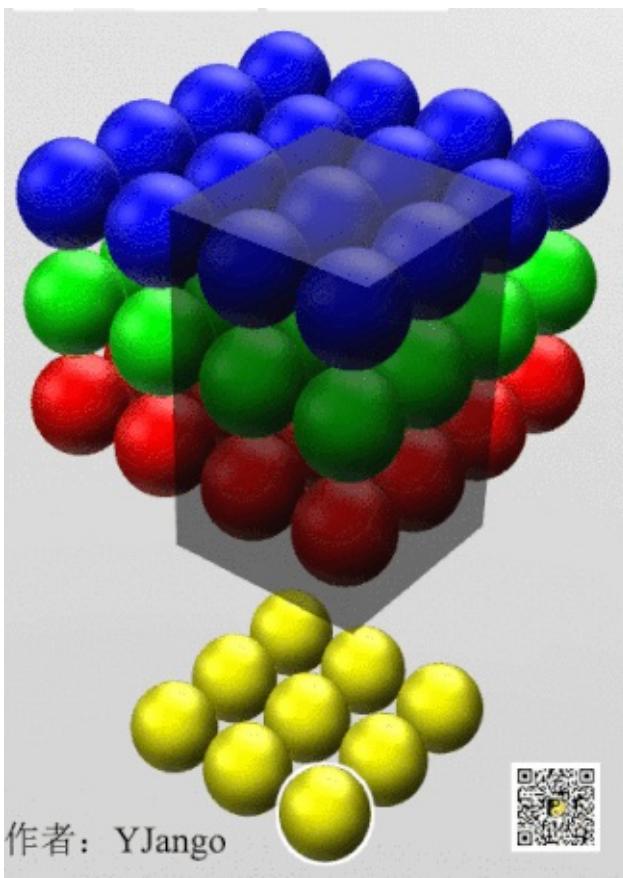
注意：三个channels的权重并不共享。即当深度变为3后，权重也跟着扩增到了三组，如式子(3)所示，不同channels用的是自己的权重。式子中增加的角标r,g,b分别表示red channel, green channel, blue channel的权重。

$$\begin{bmatrix} w_{r1} & w_{r2} \\ w_{r3} & w_{r4} \end{bmatrix}, \begin{bmatrix} w_{g1} & w_{g2} \\ w_{g3} & w_{g4} \end{bmatrix}, \begin{bmatrix} w_{b1} & w_{b2} \\ w_{b3} & w_{b4} \end{bmatrix} \quad (3)$$

计算例子：用 $x_{r0}$ 表示red channel的编号为0的输入节点， $x_{g5}$ 表示green channel编号为5个输入节点。 $x_{b1}$ 表示blue channel。如式子(4)所表达，这时的一个输出节点实际上是12个输入节点的线性组合。

$$\begin{aligned}
 y_0 &= x_{r0} * w_{r1} + x_{r1} * w_{r2} + x_{r4} * w_{r3} + x_{r5} * w_{r4} + x_{g0} * w_{g1} + x_{g1} * w_{g2} + x_{g4} * w_{g3} + x_{g5} * w_{g4} + x_{b0} \\
 &\quad * w_{b1} + x_{b1} * w_{b2} + x_{b4} * w_{b3} + x_{b5} * w_{b4} + b_0 \\
 y_0 &= [w_{r1} \quad w_{r2} \quad w_{r3} \quad w_{r4}] \cdot \begin{bmatrix} x_{r0} \\ x_{r1} \\ x_{r4} \\ x_{r5} \end{bmatrix} + [w_{g1} \quad w_{g2} \quad w_{g3} \quad w_{g4}] \cdot \begin{bmatrix} x_{g0} \\ x_{g1} \\ x_{g4} \\ x_{g5} \end{bmatrix} + [w_{b1} \quad w_{b2} \quad w_{b3} \quad w_{b4}] \\
 &\quad \cdot \begin{bmatrix} x_{b0} \\ x_{b1} \\ x_{b4} \\ x_{b5} \end{bmatrix} + b_0
 \end{aligned} \quad (4)$$

当filter扫到其他位置计算输出节点 $y_i$ 时，那12个权重在不同位置是共用的，如下面的动态图所展示。透明黑框圈中的12个节点会连接到被白色边框选中的黄色节点上。

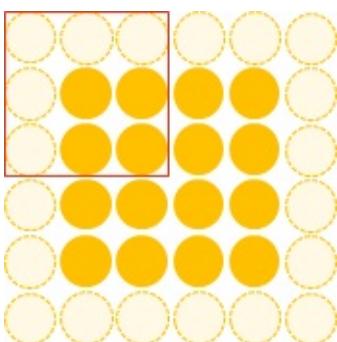


每个**filter**会在**width**维, **height**维上, 以局部连接和空间共享, 并贯穿整个**depth**维的方式得到一个**Feature Map**。

## Zero padding

细心的读者应该早就注意到了,  $4 \times 4$ 的图片被 $2 \times 2$ 的filter卷积后变成了 $3 \times 3$ 的图片, 每次卷积后都会小一圈的话, 经过若干层后岂不是变的越来越小? Zero padding就可以在这时帮助控制Feature Map的输出尺寸, 同时避免了边缘信息被一步步舍弃的问题。

例如: 下面 $4 \times 4$ 的图片在边缘Zero padding一圈后, 再用 $3 \times 3$ 的filter卷积后, 得到的Feature Map尺寸依然是 $4 \times 4$ 不变。



通常大家都想要在卷积时保持图片的原始尺寸。选择 $3 \times 3$ 的filter和1的zero padding, 或 $5 \times 5$ 的filter和2的zero padding可以保持图片的原始尺寸。这也是为什么大家多选择 $3 \times 3$ 和 $5 \times 5$ 的filter的原因。另一个原因是 $3 \times 3$ 的filter考虑到了像素与其距离为1以内的所有其他像素的关系, 而

$5 \times 5$ 则是考虑像素与其距离为2以内的所有其他像素的关系。

尺寸：Feature Map的尺寸等于 $(\text{input\_size} + 2 * \text{padding\_size} - \text{filter\_size})/\text{stride} + 1$ 。

注意：上面的式子是计算width或height一维的。 $\text{padding\_size}$ 也表示的是单边补零的个数。例如 $(4+2-3)/1+1 = 4$ ，保持原尺寸。

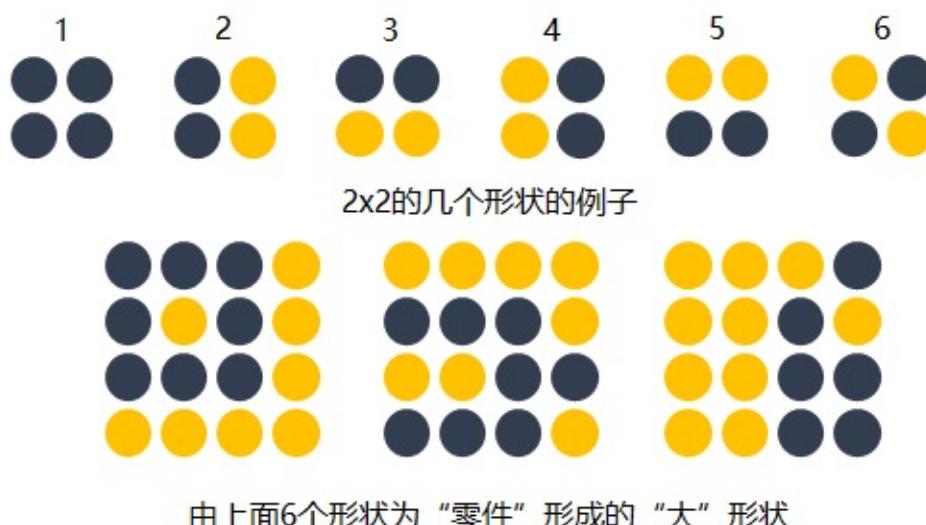
不用去背这个式子。其中 $(\text{input\_size} + 2 * \text{padding\_size})$ 是经过Zero padding扩充后真正要卷积的尺寸。减去 $\text{filter\_size}$ 后表示可以滑动的范围。再除以可以一次滑动（ $\text{stride}$ ）多少后得到滑动了多少次，也就意味着得到了多少个输出节点。再加上第一个不需要滑动也存在的输出节点后就是最后的尺寸。

## 形状、概念抓取

知道了每个filter在做什么之后，我们再来思考这样的一个filter会抓取到什么样的信息。

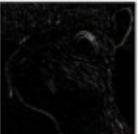
我们知道不同的形状都可由细小的“零件”组合而成的。比如下图中，用 $2 \times 2$ 的范围所形成的16种形状可以组合成格式各样的“更大”形状。

卷积的每个filter可以探测特定的形状。又由于Feature Map保持了抓取后的空间结构。若将探测到细小图形的Feature Map作为新的输入再次卷积后，则可以由此探测到“更大”的形状概念。比如下图的第一个“大”形状可由2,3,4,5基础形状拼成。第二个可由2,4,5,6组成。第三个可由6,1组成。



除了基础形状之外，颜色、对比度等概念对画面的识别结果也有影响。卷积层也会根据需要去探测特定的概念。

可以从下面这张图中感受到不同数值的filters所卷积过后的Feature Map可以探测边缘，棱角，模糊，突出等概念。

Operation	Filter	Convolved Image
Identity	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	
Edge detection	$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}$	
	$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$	
	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	
Sharpen	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
Box blur (normalized)	$\frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	
Gaussian blur (approximation)	$\frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$	

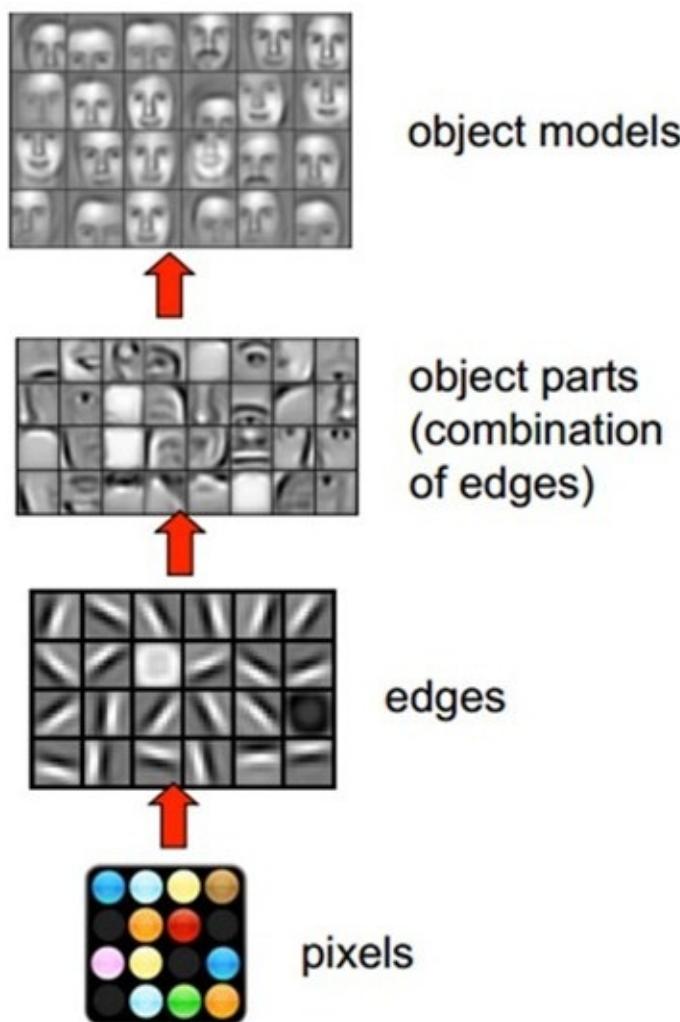
[from]

如我们先前所提，图片被识别成什么不仅仅取决于图片本身，还取决于图片是如何被观察的。

而filter内的权重矩阵W是网络根据数据学习得到的，也就是说，我们让神经网络自己学习以什么样的方式去观察图片。

拿老妇与少女的那幅图片举例，当标签是少女时，卷积网络就会学习抓取可以成少女的形状、概念。当标签是老妇时，卷积网络就会学习抓取可以成老妇的形状、概念。

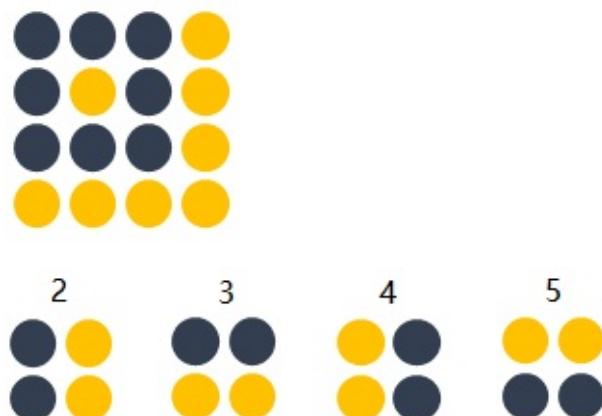
下图展现了在人脸识别中经过层层的卷积后，所能够探测的形状、概念也变得越来越抽象和复杂。



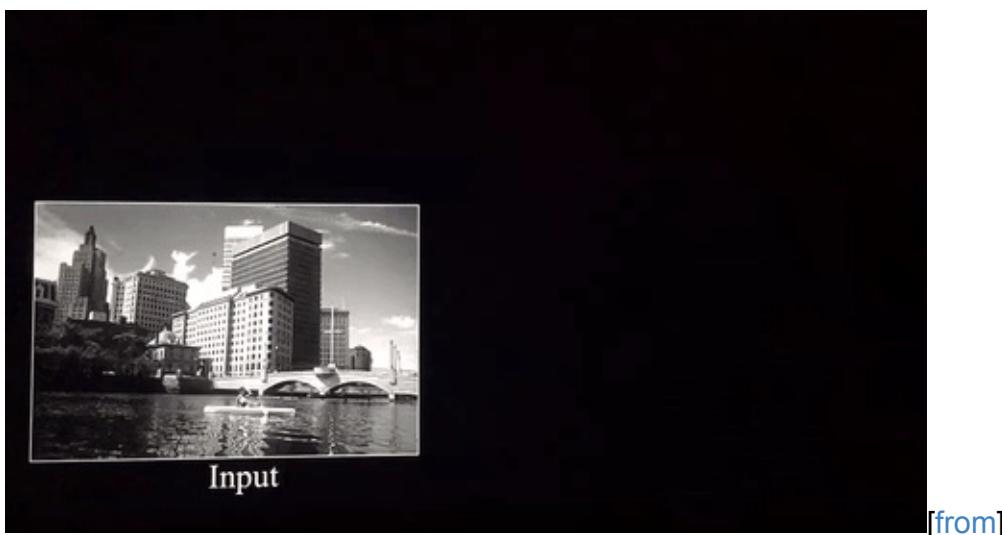
卷积神经网络会尽可能寻找最能解释训练数据的抓取方式。

## 多 filters

每个filter可以抓取探测特定的形状的存在。假如我们要探测下图的长方框形状时，可以用4个filters去探测4个基础“零件”。

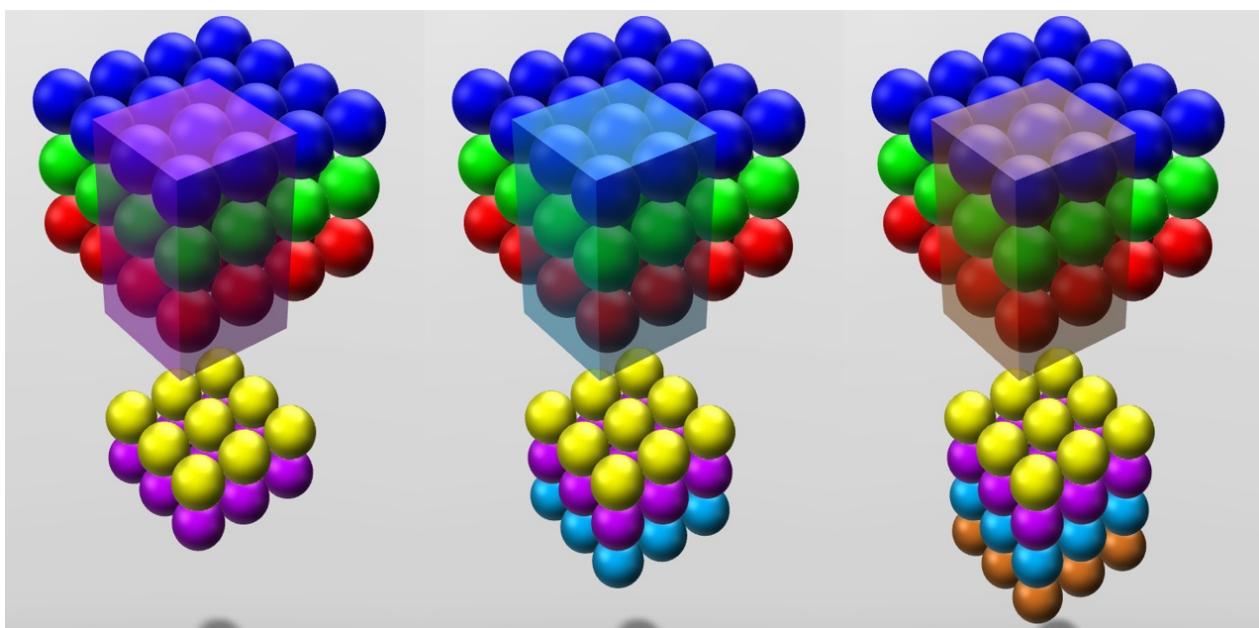


因此我们自然而然的会选择用多个不同的filters对同一个图片进行多次抓取。如下图，同一个图片，经过两个（红色、绿色）不同的filters扫描过后可得到不同特点的Feature Maps。每增加一个filter，就意味着你想让网络多抓取一个特征。



这样卷积层的输出也不再是depth为1的一个平面，而是和输入一样是depth为复数的长方体。

如下图所示，当我们增加一个filter（紫色表示）后，就又可以得到一个Feature Map。将不同filters所卷积得到的Feature Maps按顺序堆叠后，就得到了一个卷积层的最终输出。



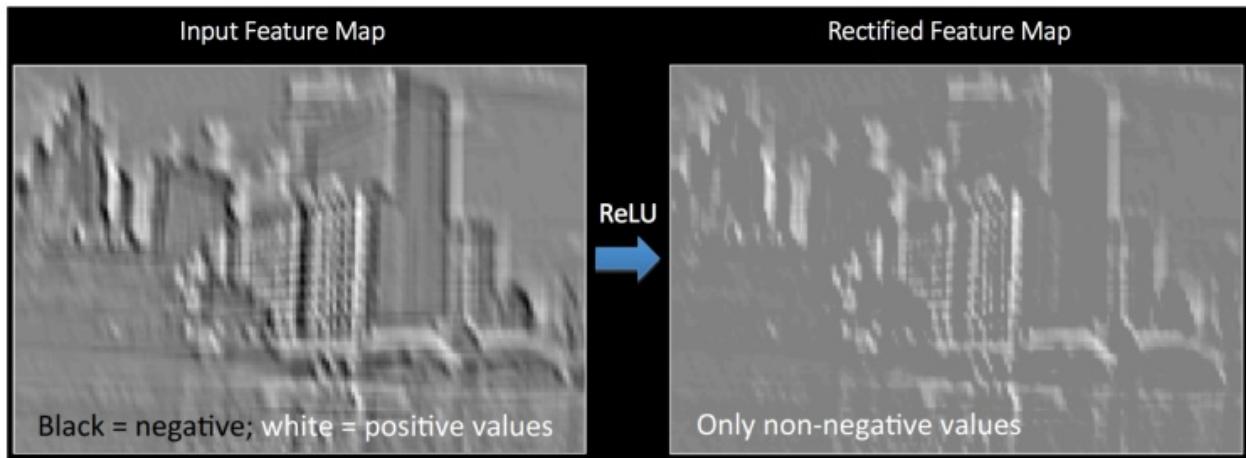
卷积层的输入是长方体，输出也是长方体。

这样卷积后输出的长方体可以作为新的输入送入另一个卷积层中处理。

## 加入非线性

和前馈神经网络一样，经过线性组合和偏移后，会加入非线性增强模型的拟合能力。

将卷积所得的Feature Map经过ReLU变换（elementwise）后所得到的output就如下图所展示。



[from]

## 输出长方体

现在我们知道了一个卷积层的输出也是一个长方体。那么这个输出长方体的(`width`, `height`, `depth`)由哪些因素决定和控制。

这里直接用[CS231n](#)的Summary：

- 输入尺寸： $W_1 \times H_1 \times D_1$
- 卷积层参数：
  - filters个数： $K$
  - filters尺寸： $F$
  - stride： $S$
  - zero padding： $P$
- 输出尺寸： $W_2 \times H_2 \times D_2$ ，其中
  - 
  - 
  -
- 权重个数：权重共享后
  - 每个filter有 $F \cdot F \cdot D_1$
  - 一共有 $F \cdot F \cdot D_1 \cdot K$ 个权重及 $K$ 个偏移。
- 常规设置： $F = 3, S = 1, P = 1$

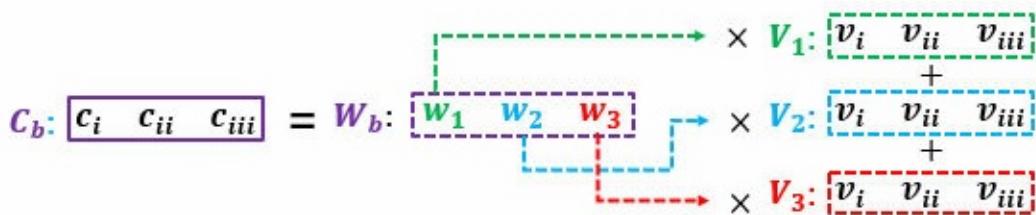
计算例子：请体会[CS231n](#)的Convolution Demo部分的演示。

## 矩阵乘法执行卷积

如果按常规以扫描的方式一步步计算局部节点和filter的权重的点乘，则不能高效的利用GPU的并行能力。所以更普遍的方法是用两个大矩阵的乘法来一次性囊括所有计算。

因为卷积层的每个输出节点都是由若干个输入节点的线性组合所计算。因为输出的节点个数是 $W_2 \times H_2 \times D_2$ ，所以就有 $W_2 \times H_2 \times D_2$ 个线性组合。

读过我写的[线性代数教程](#)的读者请回忆，矩阵乘矩阵的意义可以理解为批量的线性组合按顺序排列。其中一个矩阵所表示的信息是多组权重，另一个矩阵所表示的信息是需要进行组合的向量。大家习惯性的把组成成分放在矩阵乘法的右边，而把权重放在矩阵乘法的左边。所以这个大型矩阵乘法可以用 $W_{row} \cdot X_{col}$ 表示，其中 $W_{row}$ 和 $X_{col}$ 都是矩阵。



卷积的每个输出是由局部的输入节点和对应的filter权重展成向量后所计算的，如式子(2)。那么 $W_{row}$ 中的每一行则是每个filter的权重，有 $F \cdot F \cdot D_1$ 个；而 $X_{col}$ 的每一列是所有需要进行组合的节点（上面的动态图中被黑色透明框圈中的节点），也有 $F \cdot F \cdot D_1$ 个。 $X_{col}$ 的列的个数则表示每个filter要滑动多少次才可以把整个图片扫描完，有 $W_2 \cdot H_2$ 次。因为我们有多个filters， $W_{row}$ 的行的个数则是filter的个数 $K$ 。

最后我们得到：

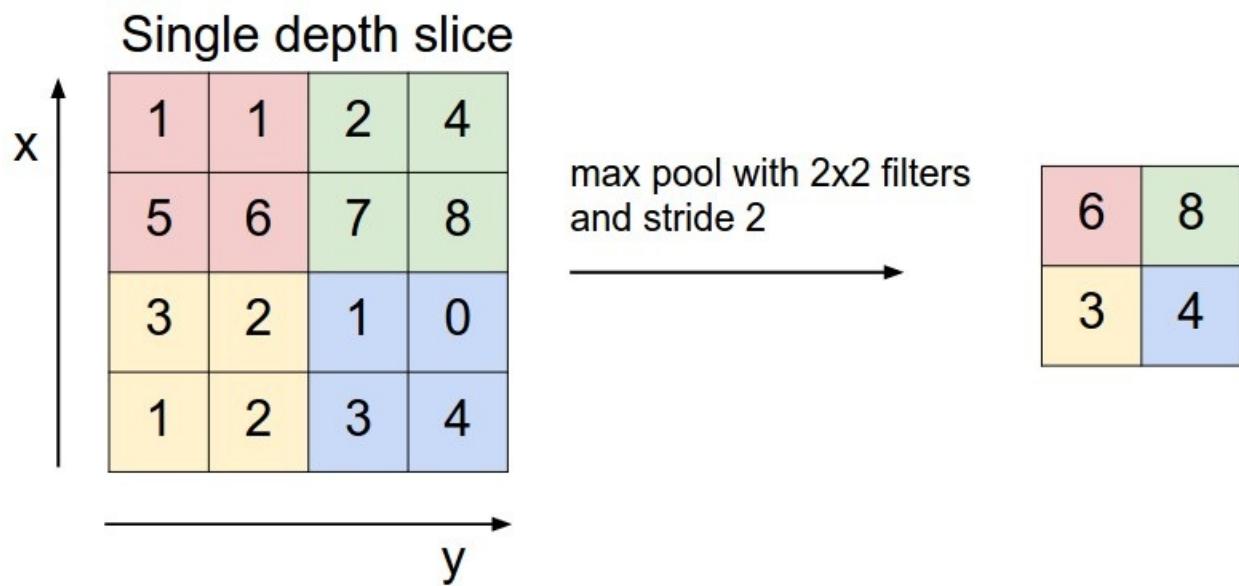
- $W_{row} \in R^{K \times F \cdot F \cdot D_1}$
- $X_{col} \in R^{F \cdot F \cdot D_1 \times W_2 \cdot H_2}$
- $W_{row} \cdot X_{col} \in R^{K \times W_2 \cdot H_2}$

当然矩阵乘法后需要将 $W_{row} \cdot X_{col}$ 整理成形状为 $W_2 \times H_2 \times D_2$ 的三维张量以供后续处理（如再送入另一个卷积层）。 $X_{col}$ 则也需要逐步的局部滑动图片，最后堆叠构成用于计算矩阵乘法的形式。

## Max pooling

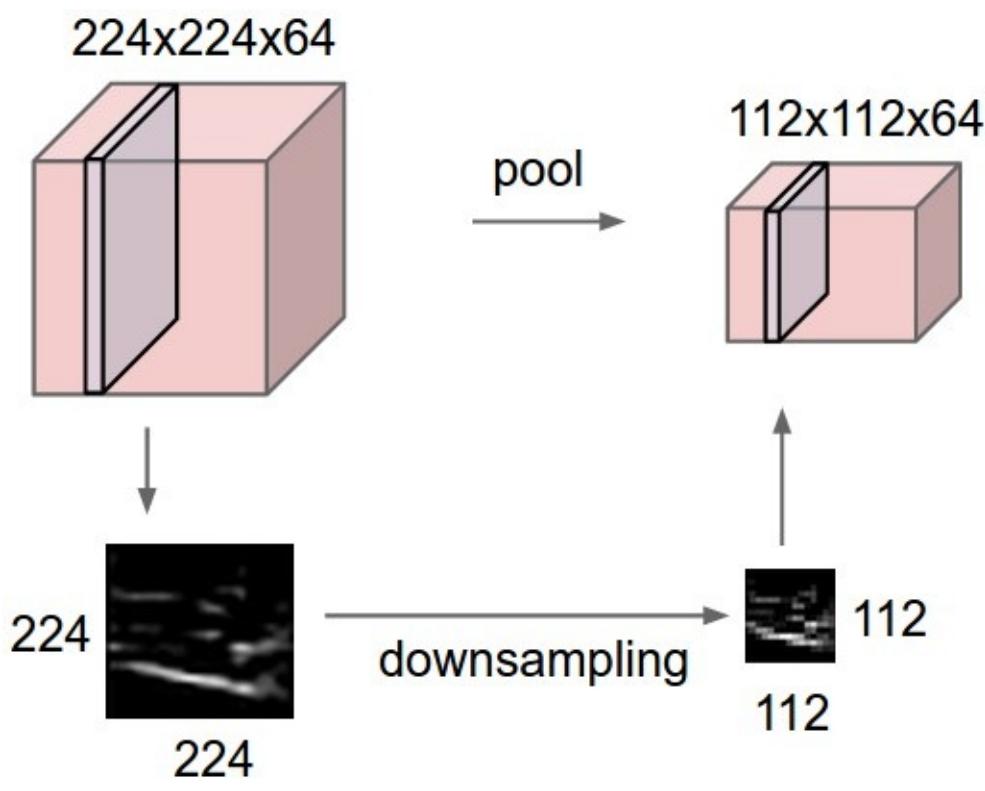
在卷积后还会有一个pooling的操作，尽管有其他的比如average pooling等，这里只提max pooling。

max pooling的操作如下图所示：整个图片被不重叠的分割成若干个同样大小的小块（pooling size）。每个小块内只取最大的数字，再舍弃其他节点后，保持原有的平面结构得出output。



[from]

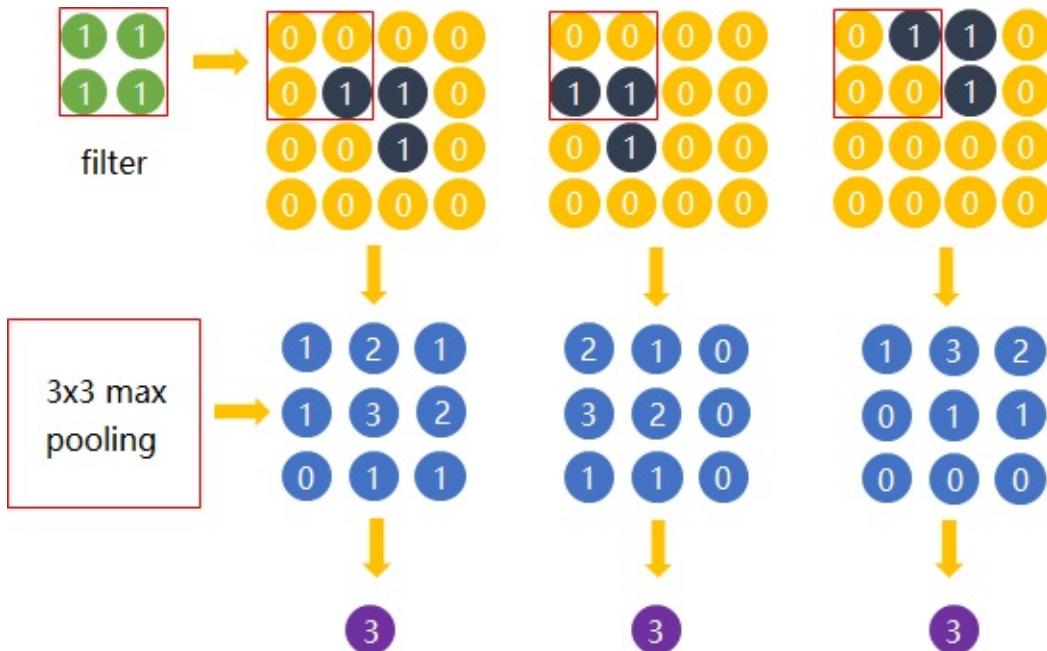
max pooling在不同的depth上是分开执行的，且不需要参数控制。那么问题就max pooling有什么作用？部分信息被舍弃后难道没有影响吗？



Max pooling的主要功能是downsampling，却不会损坏识别结果。这意味着卷积后的Feature Map中有对于识别物体不必要的冗余信息。那么我们就反过来思考，这些“冗余”信息是如何产生的。

直觉上，我们为了探测到某个特定形状的存在，用一个filter对整个图片进行逐步扫描。但只有出现了该特定形状的区域所卷积获得的输出才是真正有用的，用该filter卷积其他区域得出的数值就可能对该形状是否存在的判定影响较小。比如下图中，我们还是考虑探测“横折”这个形

状。卷积后得到 $3 \times 3$ 的Feature Map中，真正有用的就是数字为3的那个节点，其余数值对于这个任务而言都是无关的。所以用 $3 \times 3$ 的Max pooling后，并没有对“横折”的探测产生影响。试想在这里例子中如果不使用Max pooling，而让网络自己去学习。网络也会去学习与Max pooling近似效果的权重。因为是近似效果，增加了更多的parameters的代价，却还不如直接进行Max pooling。



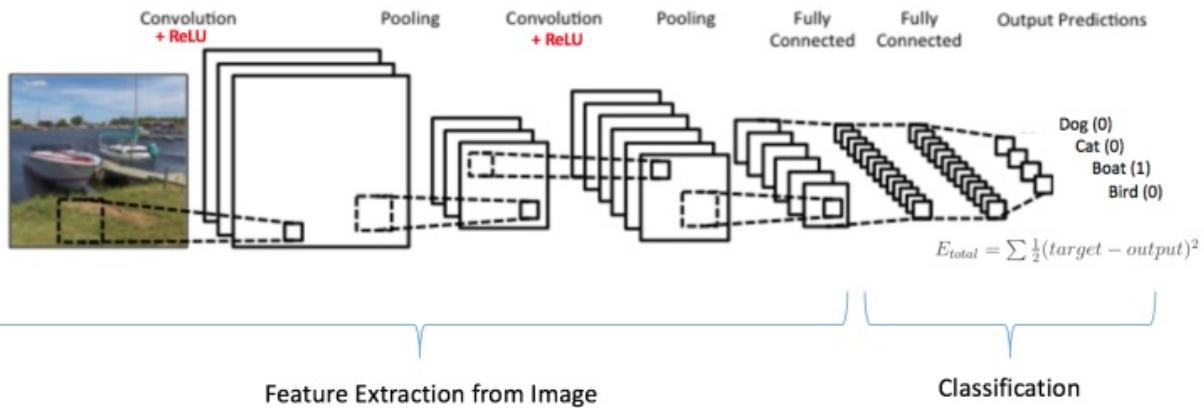
Max pooling还有类似选择句的功能。假如有两个节点，其中第一个节点会在某些输入下最大，那么网络就只在这个节点上流通信息；而另一些输入又会让第二个节点的值最大，那么网络就转而走这个节点的分支。

但是Max pooling也有不好的地方。因为并非所有的抓取都像上图的例子。有些周边信息对某个概念是否存在的判定也有影响。并且Max pooling是对所有的Feature Maps进行等价的操作。就好比用相同网孔的渔网打鱼，一定会有漏网之鱼。

## 全连接层

当抓取到足以用来识别图片的特征后，接下来的就是如何进行分类。全连接层（也叫前馈层）就可以用来将最后的输出映射到线性可分的空间。通常卷积网络的最后会将末端得到的长方体平摊(flatten)成一个长长的向量，并送入全连接层配合输出层进行分类。

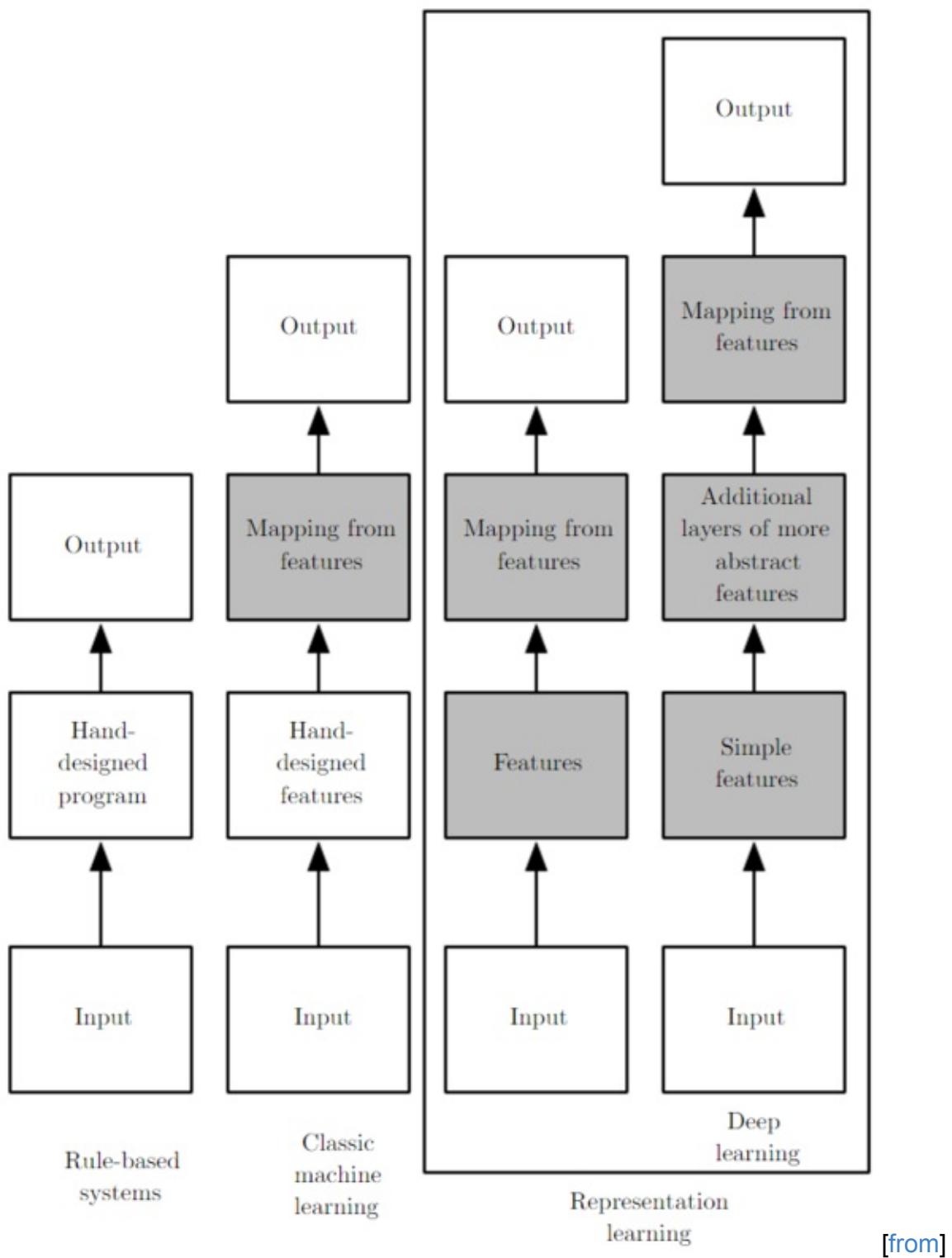
卷积神经网络大致就是convolutional layer, pooling layer, ReLu layer, fully-connected layer的组合，例如下图所示的结构。



[from]

这里也体现了深层神经网络或deep learning之所以称deep的一个原因：模型将特征抓取层和分类层合在了一起。负责特征抓取的卷积层主要是用来学习“如何观察”。

下图简述了机器学习的发展，从最初的人工定义特征再放入分类器的方法，到让机器自己学习特征，再到如今尽量减少人为干涉的deep learning。



## 结构发展

以上介绍了卷积神经网络的基本概念。以下是几个比较有名的卷积神经网络结构，详细的请看[CS231n](#)。

- **LeNet**：第一个成功的卷积神经网络应用
- **AlexNet**：类似LeNet，但更深更大。使用了层叠的卷积层来抓取特征（通常是一个卷积层马上一个max pooling层）

- **ZF Net**：增加了中间卷积层的尺寸，让第一层的stride和filter size更小。
- **GoogLeNet**：减少parameters数量，最后一层用max pooling层代替了全连接层，更重要的是Inception-v4模块的使用。
- **VGGNet**：只使用3x3 卷积层和2x2 pooling层从头到尾堆叠。
- **ResNet**：引入了跨层连接和batch normalization。
- **DenseNet**：将跨层连接从头进行到底。

总结一下：这些结构的发展趋势有：

- 使用small filter size的卷积层和pooling
- 去掉parameters过多的全连接层
- Inception（稍后会对其中的细节进行说明）
- 跨层连接

## 不变性的满足

接下来会谈谈我个人的，对于画面不变性是如何被卷积神经网络满足的想法。同时结合不变性，对上面提到的结构发展的重要变动进行直觉上的解读。

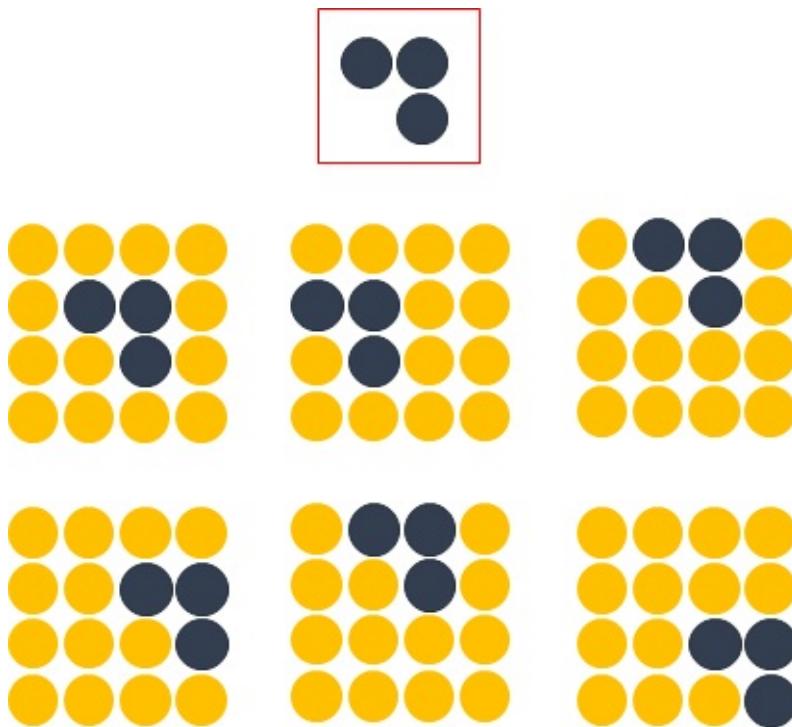
需要明白的是为什么加入不变性可以提高网络表现。并不是因为我们用了更炫酷的处理方式，而是加入了先验知识，无需从零开始用数据学习，节省了训练所需数据量。思考表现提高的原因一定要从训练所需要的数据量切入。提出满足新的不变性特点的神经网络是计算机视觉的一个主要研究方向。

### 平移不变性

可以说卷积神经网络最初引入局部连接和空间共享，就是为了满足平移不变性。



因为空间共享，在不同位置的同一形状就可以被等价识别，所以不需要对每个位置都进行学习。



## 旋转和视角不变性

个人觉得卷积神经网络克服这一不变性的主要手段还是靠大量的数据。并没有明确加入“旋转和视角不变性”的先验特性。



[Deformable Convolutional Networks](#)似乎是对此变性进行了进行增强。

## 尺寸不变性

与平移不变性不同，最初的卷积网络并没有明确照顾尺寸不变性这一特点。



我们知道filter的size是事先选择的，而不同的尺寸所寻找的形状（概念）范围不同。

从直观上思考，如果选择小范围，再一步步通过组合，仍然是可以得到大范围的形状。如 $3 \times 3$ 尺寸的形状都是可以由 $2 \times 2$ 形状的图形组合而成。所以形状的尺寸不变性对卷积神经网络而言并不算问题。这恐怕ZF Net让第一层的stride和filter size更小，VGGNet将所有filter size都设置成 $3 \times 3$ 仍可以得到优秀结果的一个原因。

但是，除了形状之外，很多概念的抓取通常需要考虑一个像素与周边更多像素之间的关系后得出。也就是说 $5 \times 5$ 的filter也是有它的优点。同时，小尺寸的堆叠需要很多个filters来共同完成，如果需要抓取的形状恰巧在 $5 \times 5$ 的范围，那么 $5 \times 5$ 会比 $3 \times 3$ 来的更有效率。所以一次性使用多个不同filter size来抓取多个范围不同的概念是一种顺理成章的想法，而这个也就是Inception。可以说Inception是为了尺寸不变性而引入的一个先验知识。

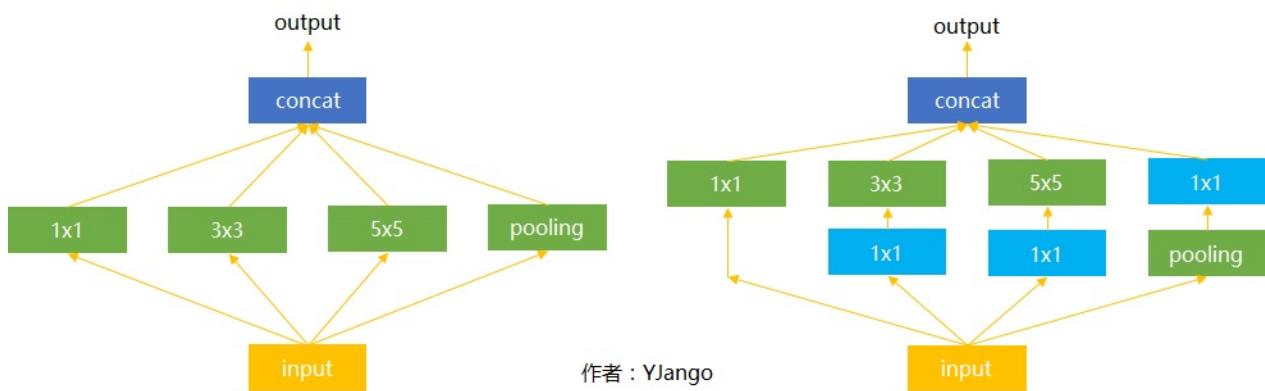
## Inception

下图是Inception的结构，尽管也有不同的版本，但是其动机都是一样的：消除尺寸对于识别结果的影响，一次性使用多个不同filter size来抓取多个范围不同的概念，并让网络自己选择需要的特征。

你也一定注意到了蓝色的 $1 \times 1$ 卷积，撇开它，先看左边的这个结构。

输入（可以是被卷积完的长方体输出作为该层的输入）进来后，通常我们可以选择直接使用像素信息( $1 \times 1$ 卷积)传递到下一层，可以选择 $3 \times 3$ 卷积，可以选择 $5 \times 5$ 卷积，还可以选择max pooling的方式downsample刚被卷积后的feature maps。但在实际的网络设计中，究竟该如何选择需要大量的实验和经验的。Inception就不用我们来选择，而是将4个选项给神经网络，让网络自己去选择最合适的解决方案。

接下来我们再看右边的这个结构，多了很多蓝色的 $1 \times 1$ 卷积。这些 $1 \times 1$ 卷积的作用是为了让网络根据能够更灵活的控制数据的depth的。

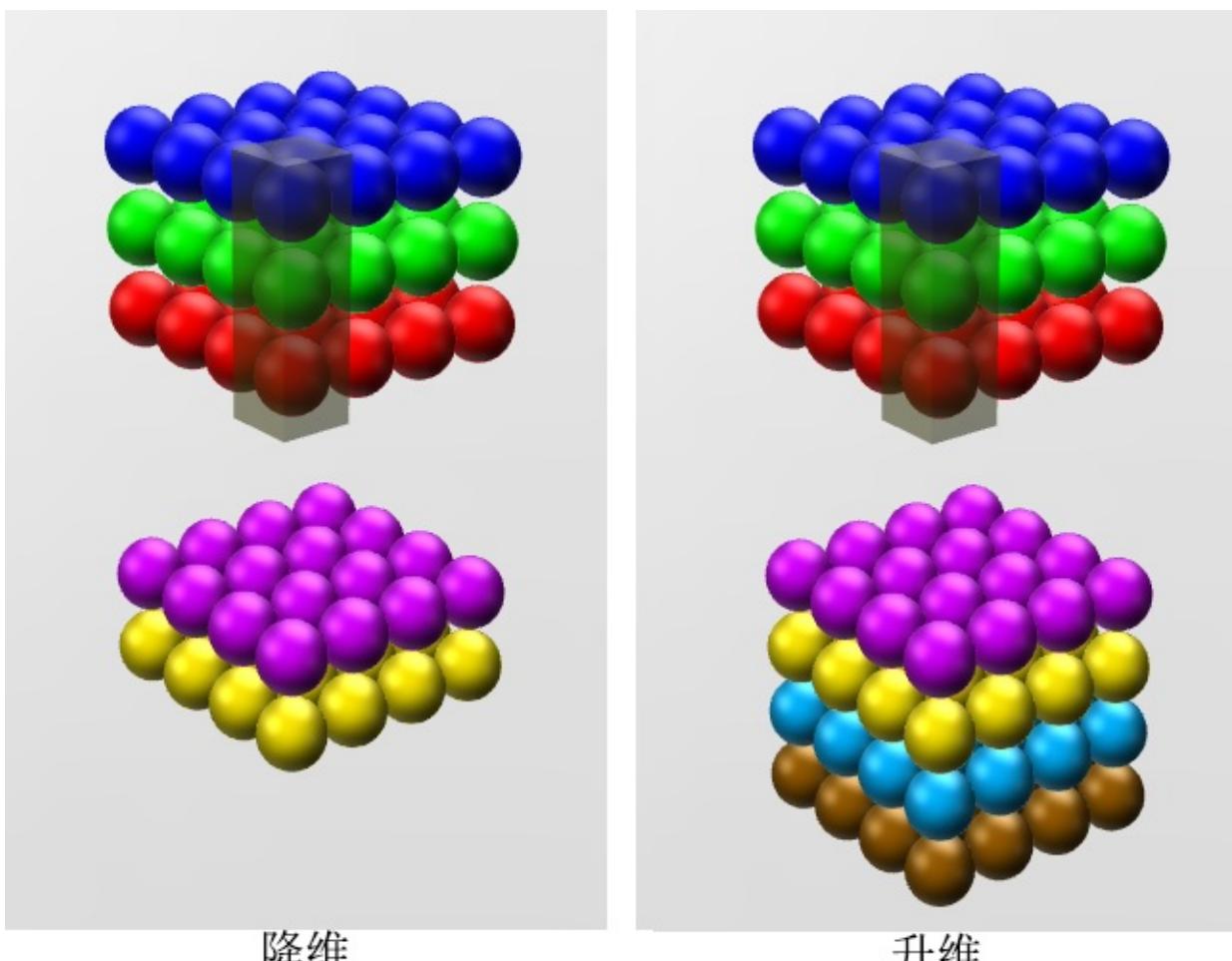


## 1x1 卷积核

如果卷积的输出输入都只是一个平面，那么 $1\times 1$ 卷积核并没有什么意义，它是完全不考虑像素与周边其他像素关系。但卷积的输出输入是长方体，所以 $1\times 1$ 卷积实际上是对每个像素点，在不同的channels上进行线性组合（信息整合），且保留了图片的原有平面结构，调控depth，从而完成升维或降维的功能。

如下图所示，如果选择2个filters的 $1\times 1$ 卷积层，那么数据就从原本的depth 3 降到了2。若用4个filters，则起到了升维的作用。

这就是为什么上面Inception的4个选择中都混有一个 $1\times 1$ 卷积，如右侧所展示的那样。其中，绿色的 $1\times 1$ 卷积本身就 $1\times 1$ 卷积，所以不需要再用另一个 $1\times 1$ 卷积。而max pooling用来去掉卷积得到的Feature Map中的冗余信息，所以出现在 $1\times 1$ 卷积之前，紧随刚被卷积后的feature maps。（由于没做过实验，不清楚调换顺序会有什么影响。）

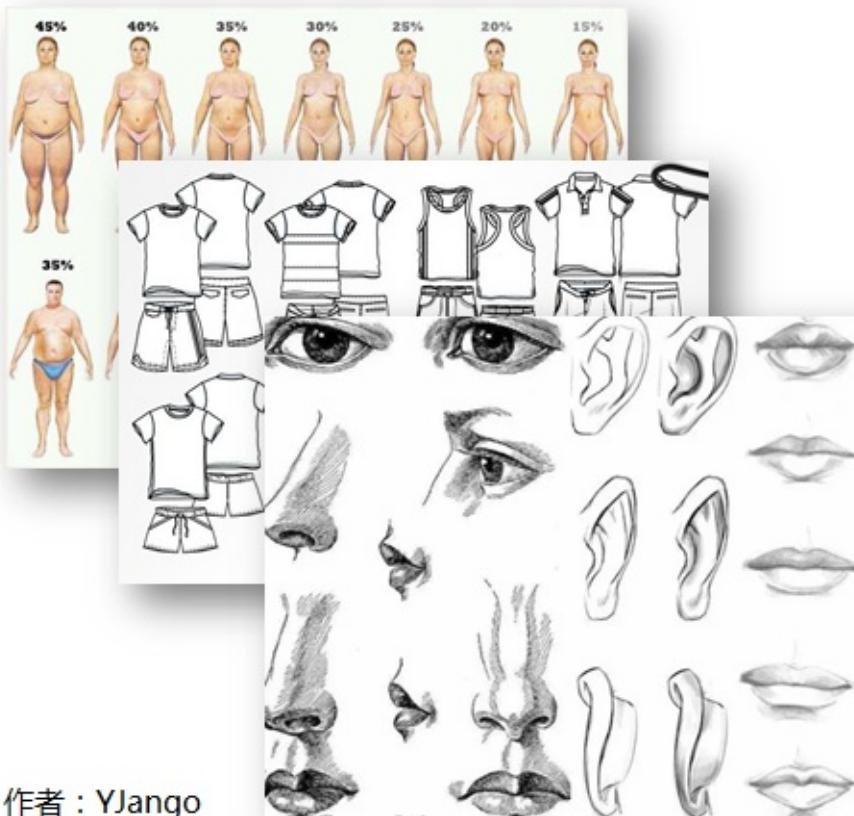


## 跳层连接

前馈神经网络也好，卷积神经网络也好，都是一层一层逐步变换的，不允许跳层组合。但现实中是否有跳层组合的现象？

比如说我们在判断一个人的时候，很多时候我们并不是观察它的全部，或者给你的图片本身就是残缺的。这时我们会靠单个五官，外加这个人的着装，再加他的身形来综合判断这个人，如下图所示。这样，即便图片本身是残缺的也可以很好的判断它是什么。这和前馈神经网络的先验知识不同，它允许不同层级之间的因素进行信息交互、综合判断。

残差网络就是拥有这种特点的神经网络。大家喜欢用*identity mappings*去解释为什么残差网络更优秀。这里我只是提供了一个以先验知识的角度去理解的方式。需要注意的是每一层并不会像我这里所展示的那样，会形成明确的五官层。只是有这样的组合趋势，实际无法保证神经网络到底学到了什么内容。



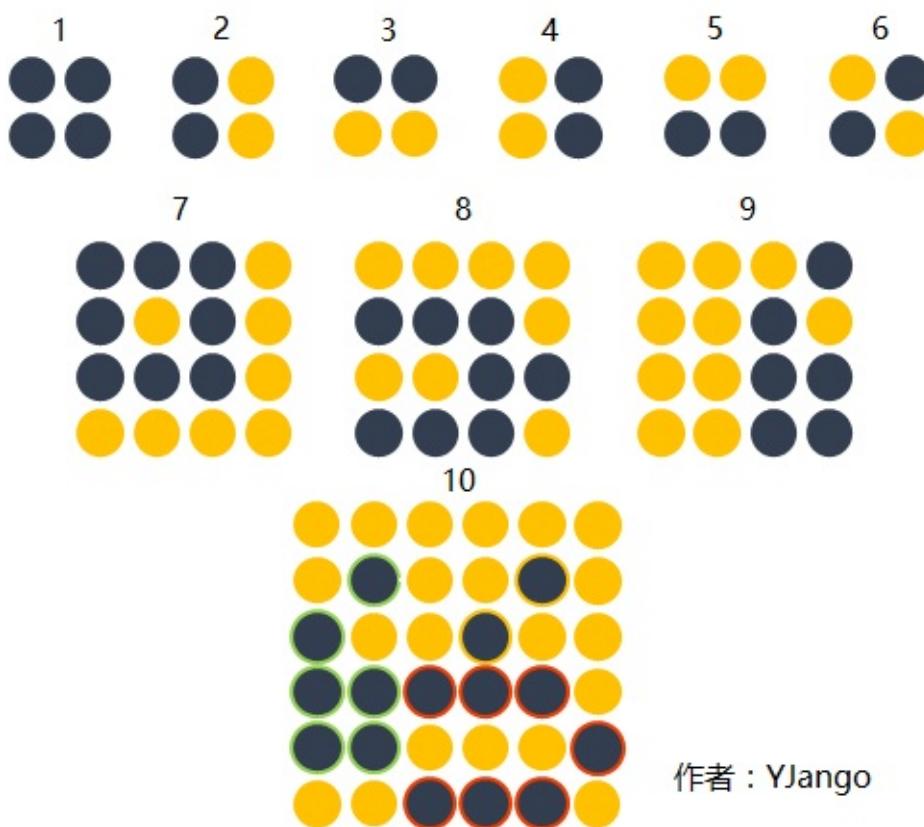
## 身形 + 着装 + 眼睛 → 来判断一个人

用下图举一个更易思考的例子。图形1,2,3,4,5,6是第一层卷积层抓取到的概念。图形7,8,9是第二层卷积层抓取到的概念。图形7,8,9是由1,2,3,4,5,6的基础上组合而成的。

但当我们想要探测的图形10并不是单纯的靠图形7,8,9组成，而是第一个卷积层的图形6和第二个卷积层的8,9组成的话，不允许跨层连接的卷积网络不得不更多地使用filter来保持第一层已经抓取到的图形信息。并且每次传递到下一层都需要学习那个用于保留前一层图形概念的filter的权重。当层数变深后，会越来越难以保持，还需要max pooling将冗余信息去掉。

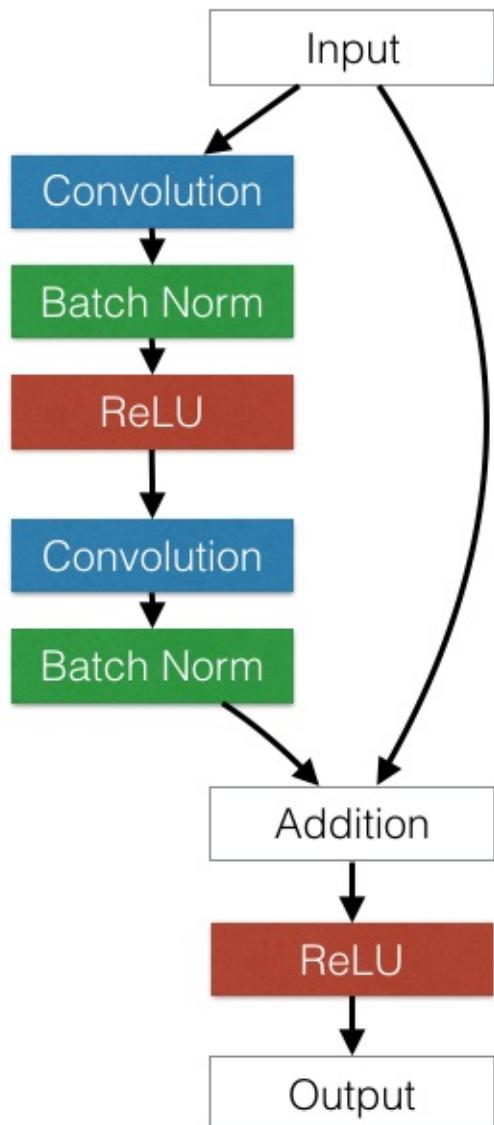
一个合理的选择就是直接将上一层所抓取的概念也跳层传递给下一层，不用让其每次都重新学习。就好比在编程时构建了不同规模的functions。每个function我们都是保留，而不是重新再写一遍。提高了重用性。

同时，因为ResNet使用了跳层连接的方式。也不需要max pooling对保留低层信息时所产生的冗余信息进行去除。

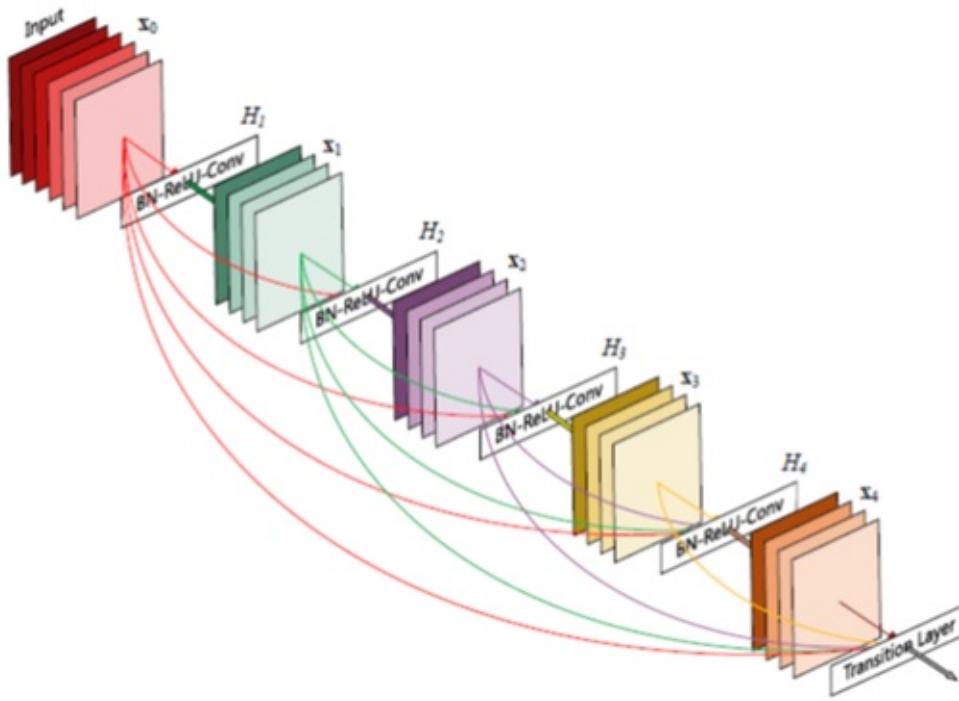


Inception中的第一个 $1\times 1$ 的卷积通道也有类似的作用，但是 $1\times 1$ 的卷积仍有权重需要学习。并且Inception所使用的结合方式是concatenate的合并成一个更大的向量的方式，而ResNet的结合方式是sum。两个结合方式各有优点。concatenate当需要用不同的维度去组合成新观念的时候更有益。而sum则更适用于并存的判断。比如既有油头发，又有胖身躯，同时穿着常年不洗的牛仔裤，三个不同层面的概念并存时，该人会被判定为程序员的情况。又比如双向LSTM中正向和逆向序列抓取的结合常用相加的方式结合。在语音识别中，这表示既可以正向抓取某种特征，又可以反向抓取另一种特征。当两种特征同时存在时才会被识别成某个特定声音。

在下图的ResNet中，前一层的输入会跳过部分卷积层，将底层信息传递到高层。



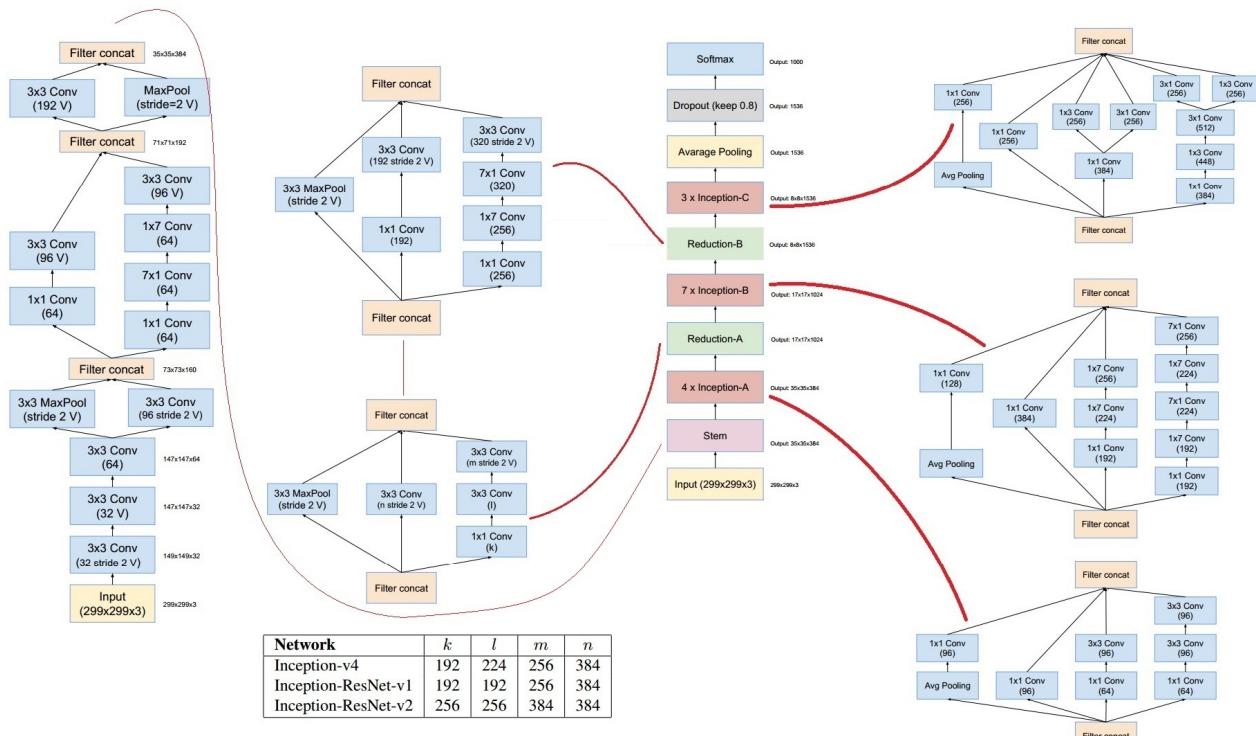
在下图的DenseNet中，底层信息会被传递到所有的后续高层。



## 后续

随着时间推移，各个ResNet, GoogLeNet等框架也都在原有的基础上进行了发展和改进。但基本都是上文描述的概念的组合使用加上其他的tricks。

如下图所展示的，加入跳层连接的Inception-ResNet。



但对我而言，

真正重要的是这些技巧对于各种不变性的满足。







# 为什么要做Word Embedding

在做自然语言处理（NLP）时，都会用到word embedding技巧。可是为什么要做word embedding？它究竟给我们的机器学习带来了什么好处？

## 目录

- 单词表达
  - One hot representation
  - Distributed representation
- Word embedding
  - 目的
    - 数据量角度
    - 神经网络分析
  - 训练简述

## 单词表达

word embedding被翻译成词向量，但它的好处并不是把单词向量化了。可以向量化单词的方法有很多。

先前在[卷积神经网络](#)的一节中，提到过图片是如何在计算机中被表达的。同样的，单词也需要用计算机可以理解的方式表达后，才可以进行接下来的操作。

## One hot representation

程序中编码单词的一个方法是one hot encoding。

实例：有1000个词汇量。排在第一个位置的代表英语中的冠词"a"，那么这个"a"是用 $[1,0,0,0,0,\dots]$ ，只有第一个位置是1，其余位置都是0的1000维度的向量表示，如下图中的第一列所示。

“a”	“abbreviations”	“zoology”	“zoom”
1	0	0	0
0	1	0	1
0	0	0	0
.	.	.	.
.	.	.	.
.	.	0	0
0	0	1	0
0	0	0	0
0	0	1	1

也就是说，

在one hot representation编码的每个单词都是一个维度，彼此independent。

## Distributed representation

然而每个单词彼此无关这个特点明显不符合我们的现实情况。我们知道大量的单词都是有关。

语义：girl和woman虽然用在不同年龄上，但指的都是女性。复数：word和words仅仅是复数和单数的差别。时态：buy和bought表达的都是“买”，但发生的时间不同。

所以用one hot representation的编码方式，上面的特性都没有被考虑到。我们更希望用诸如“语义”，“复数”，“时态”等维度去描述一个单词。每一个维度不再是0或1，而是连续的实数，表示不同的程度。

## word embedding

### 目的

但是说到底，为什么我们想要用Distributed representation的方式去表达一个单词呢。

### 数据量角度

这需要再次记住我们的目的：

机器学习：从大量的 $\{(x_i, y_i)_{i=1}^N\}$ 个样本中，寻找可以较好预测未见过 $x_{new}$ 所对应的 $y_{new}$ 的函数 $f: x \rightarrow y$ 。

实例：在我们日常生活的学习中，大量的 $\{(x_i, y_i)_{i=1}^N\}$ 就是历年真题， $x_i$ 是题目，而 $y_i$ 是对应的正确答案。高考时将会遇到的 $x_{new}$ 往往是我们没见过的题目，希望通过做题训练出来的解题方法 $f: x \rightarrow y$ 来求解出正确的 $y_{new}$ 。

如果可以见到所有的情况，那么只需要记住所有的 $x_i$ 所对应的 $y_i$ 就可以完美预测。但正如高考无法见到所有类型的题一样，我们无法见到所有的情况。这意味着，

机器学习需要从有限的例子中寻找到合理的 $f$ 。

高考有两个方向提高分数：

- 方向一：训练更多的数据：题海战术。
- 方向二：加入先验知识：尽可能排除不必要的可能性。

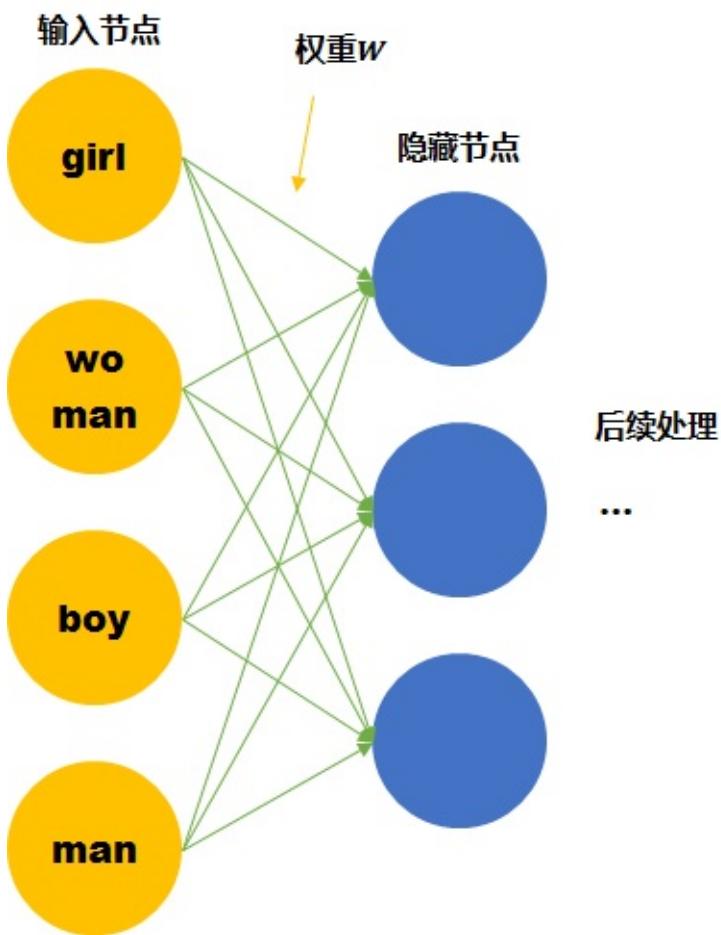
问题的关键在于训练所需要的数据量上。同理，如果我们用One hot representation去学习，那么每一个单词都需要实例数据去训练，即便我们知道"Cat"和"Kitty"很多情况下可以被理解成一个意思。为什么相同的东西却需要分别用不同的数据进行学习？

## 神经网络分析

假设我们的词汇只有4个，girl, woman, boy, man，下面就思考用两种不同的表达方式会有什么区别。

### One hot representation

尽管我们知道他们彼此的关系，但是计算机并不知道。在神经网络的输入层中，每个单词都会被看作一个节点。而我们知道训练神经网络就是要学习每个连接线的权重。如果只看第一层的权重 $W$ ，下面的情况需要确定 $4*3$ 个连接线的关系，因为每个维度都彼此独立，girl的数据不会对其他单词的训练产生任何帮助，训练所需要的数据量，基本就固定在那里了。

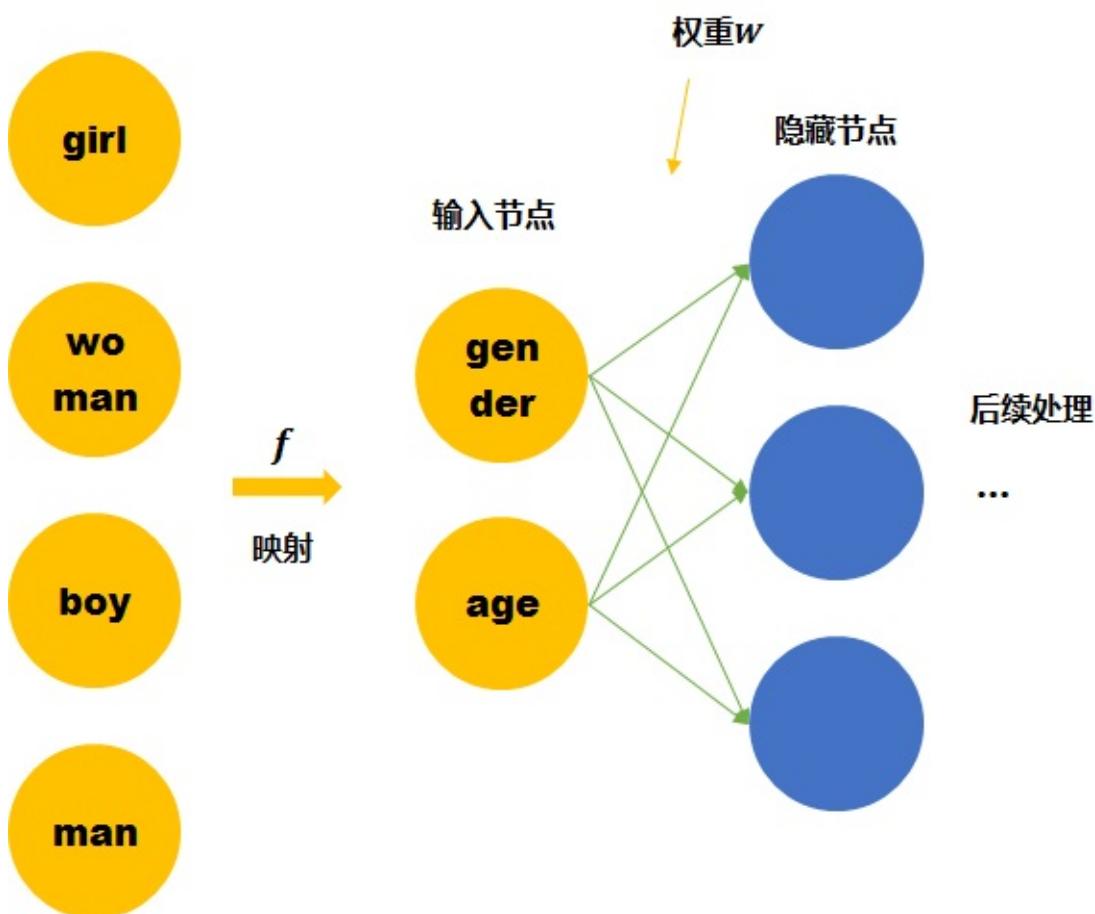


## Distributed representation

我们这里手动的寻找这四个单词之间的关系 $f$ 。可以用两个节点去表示四个单词。每个节点取不同值时的意义如下表。那么girl就可以被编码成向量 $[0,1]$ ，man可以被编码成 $[1,1]$ （第一个维度是gender，第二个维度是age）。

	0	1
gender	female	male
age	child	adult

那么这时再来看神经网络需要学习的连接线的权重就缩小到了 $2 \times 3$ 。同时，当送入girl为输入的训练数据时，因为它是由两个节点编码的。那么与girl共享相同连接的其他输入例子也可以被训练到（如可以帮助到与其共享female的woman，和child的boy的训练）。



Word embedding就是要达到第二个神经网络所表示的结果，降低训练所需要的数据量。

而上面的四个单词可以被拆成2个节点的是由我们人工提供的先验知识将原始的输入空间经过 $f$ (上图中的黄色箭头)投射到了另一个空间(维度更小)，所以才能够降低训练所需要的数据量。但是我们没有办法一直人工提供，机器学习的宗旨就是让机器代替人力去发现pattern。

Word embedding就是要从数据中自动学习到输入空间到Distributed representation空间的映射 $f$ 。

## 训练简述

问题来了，我们该如何自动寻找到类似上面的关系，将One hot representation转变成Distributed representation。我们事先并不明确目标是什么，所以这是一个无监督学习任务。

无监督学习中常用思想是：当得到数据 $\{(x_i, y_i)\}_{i=1}^N\}$ 后，我们又不知道目标(输出)时，

- 方向一：从各个输入之间 $x_i$  $_{i=1}^N$ 的关系找目标。如聚类。
- 方向二：并接上以目标输出 $y_i$ 作为新输入的另一个任务 $g: y \rightarrow z$ ，同时我们知道 $z_i$ 的对应值。用数据 $\{(x_i, z_i)\}_{i=1}^N\}$ 训练得到 $k: x \rightarrow z$ 也就是 $z = g(f(x))$ ，中间的表达 $y$ 则是我们真正想要的目标。如生成对抗网络。

Word embedding更偏向于方向二。同样是学习一个 $k : x \rightarrow z$ ，但训练后并不使用 $k$ ，而是只取前半部分的 $f : x \rightarrow y$ 。

到这里，我们希望所寻找的 $k : x \rightarrow z$ 既有标签 $z_i$ ，又可以让 $f(x)$ 所转换得到的 $y$ 的表达具有Distributed representation中所演示的特点。

同时我们还知道，

单词意思需要放在特定的上下文中去理解。

那么具有相同上下文的单词，往往是有联系的。

实例：那这两个单词都狗的品种名，而上下文的内容已经暗指了该单词具有可爱，会舔人的特点。

- 这个可爱的 泰迪 舔了我的脸。
- 这个可爱的 金巴 舔了我的脸。

而从上面这个例子中我们就可以找到一个 $k : x \rightarrow z$ ：预测上下文。

用输入单词 $x$ 作为中心单词去预测其他单词 $z$ 出现在其周边的可能性。

我们既知道对应的 $z$ 。而该任务 $k$ 又可以让 $f(x)$ 所转换得到的 $y$ 的表达具有Distributed representation中所演示的特点。因为我们让相似的单词（如泰迪和金巴）得到相同的输出（上下文），那么神经网络就会将 $x_{\text{泰迪}}$ 和 $x_{\text{金巴}}$ 经过神经网络 $f(x)$ 得到几乎相同的 $y_{\text{泰迪}}$ 和 $y_{\text{金巴}}$ 。

用输入单词 $x$ 作为中心单词去预测周边单词的方式叫做：**The Skip-Gram Model**。

用输入单词 $x$ 作为周边单词去预测中心单词的方式叫做：**Continuous Bag of Words(CBOW)**。

该篇主要是讨论为什么要做word embedding。至于word embedding的详细训练方法在下一节描述。