

Lecture 2: 神经网络与后向传播

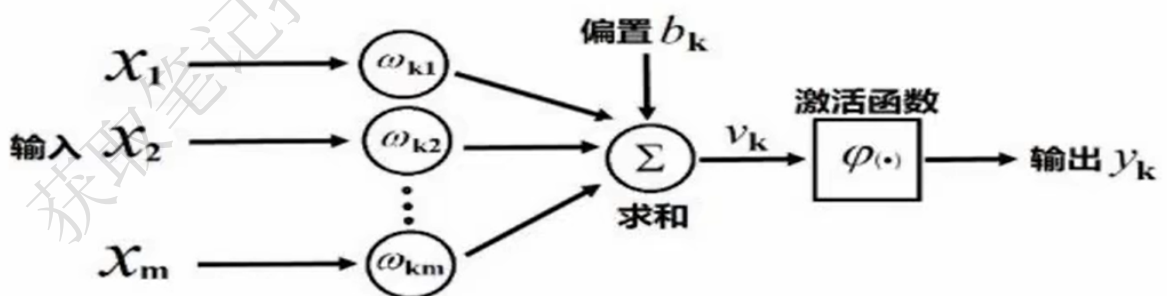
神经网络概述

基础概念

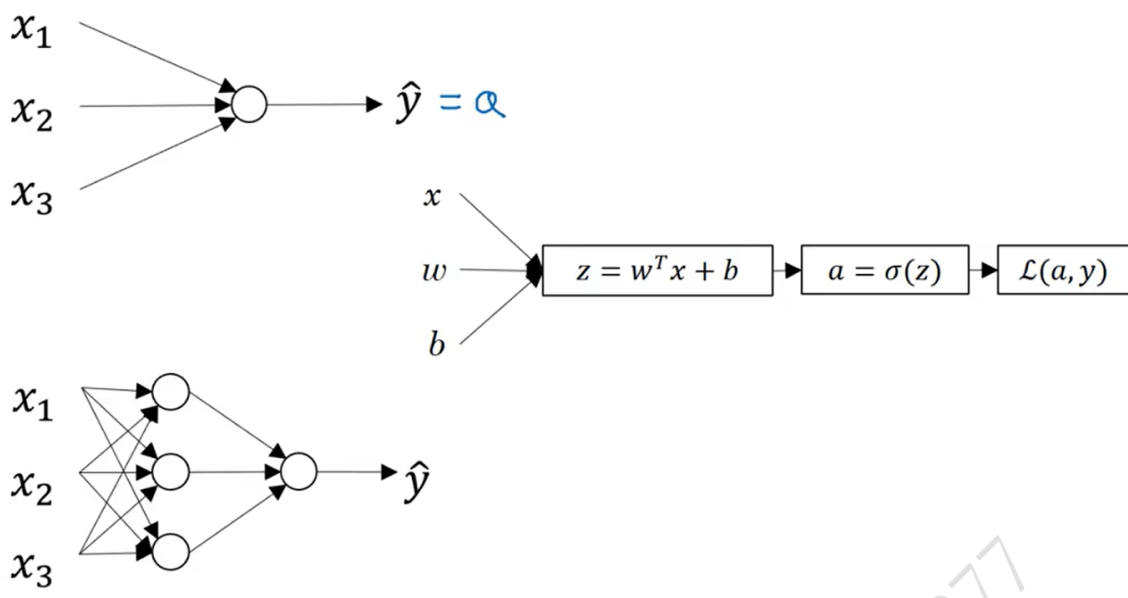
概念：神经网络的符号和表示

- 引入：
 - 1957 年，Frank Rosenblatt 从纯数学的度重新考察这一模型，指出能够从一些输入输出对 (X, y) 中通过学习算法获得权重和 b 。
- 概念：
 - 方括号： $W^{[1]}$ 表示 Layer1， $W^{[2]}$ 表示 Layer2；
 - 右下标： $W_1^{[1]}$ 表示第一层的第一个元素， $W_2^{[1]}$ 表示第一层的第二个元素， $W^{[1]}$ 表示第一层的所有元素；
 - 圆括号： $a^{[2](1)}$ 表示第二层的训练样本 1。 $a^{[2](2)}$ 表示第二层的训练样本 2；
 - 分层：
 - 输入层：第零层，即输入层不计入层数；
 - 隐藏层：我们知道输入层和输出层得到的数值，但是无法得到数据在隐藏层中的结点当中的数值；
 - 输出层：
 - 举例：

$$y_k = \varphi \left(\sum_{i=1}^m \omega_{ki} x_i + b_k \right) = \varphi (W_k^T X + b)$$



- 图示：



感知器算法

补充：感受器算法

- 概念：感知器算法

感知器算法 (Perceptron Algorithm) :

(1) 随机选择 W 和 b 。

(2) 取一个训练样本 (X, y)

(i) 若 $W^T X + b > 0$ 且 $y = -1$, 则:

$$W = W - X \quad b = b - 1$$

(ii) 若 $W^T X + b < 0$ 且 $y = +1$, 则:

$$W = W + X \quad b = b + 1$$

(3) 再取另一个 (X, y) , 回到 (2)

(4) 终止条件: 直到所有输入输出对 都不满足 (2)

中 (i) 和 (ii) 之一, 退出循环。

- 定理：感知器收敛定理

- 输入 $\{\vec{x}_i\}_{i=1 \sim N}$, 若线性可分, 则存在 W_{opt} 使得:

$$W_{opt}^T \vec{X}_i > 0 (i = 1 - N)$$

- 则利用上述感知器算法, 经过有限步后, 可以得到一个 W 使得:

$$W^T \vec{X}_i > 0 (i = 1 - N)$$

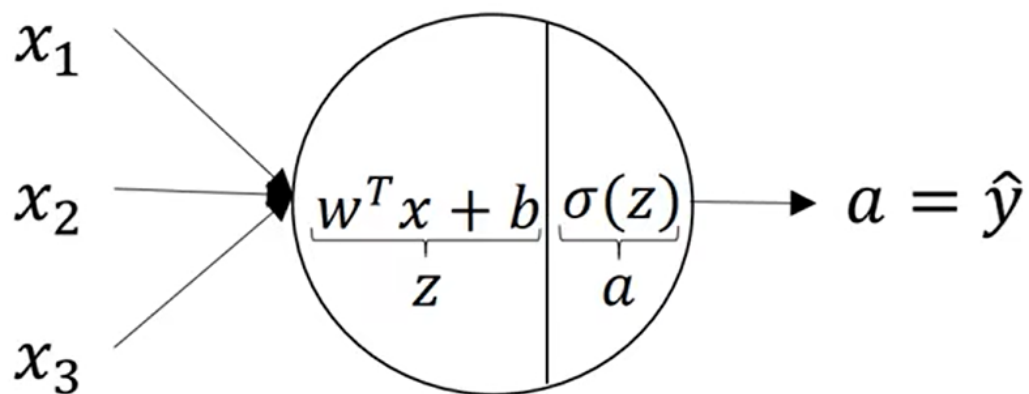
计算神经网络输出

概念：计算神经网络的输出

- 概念:

- 一个神经元表示计算:

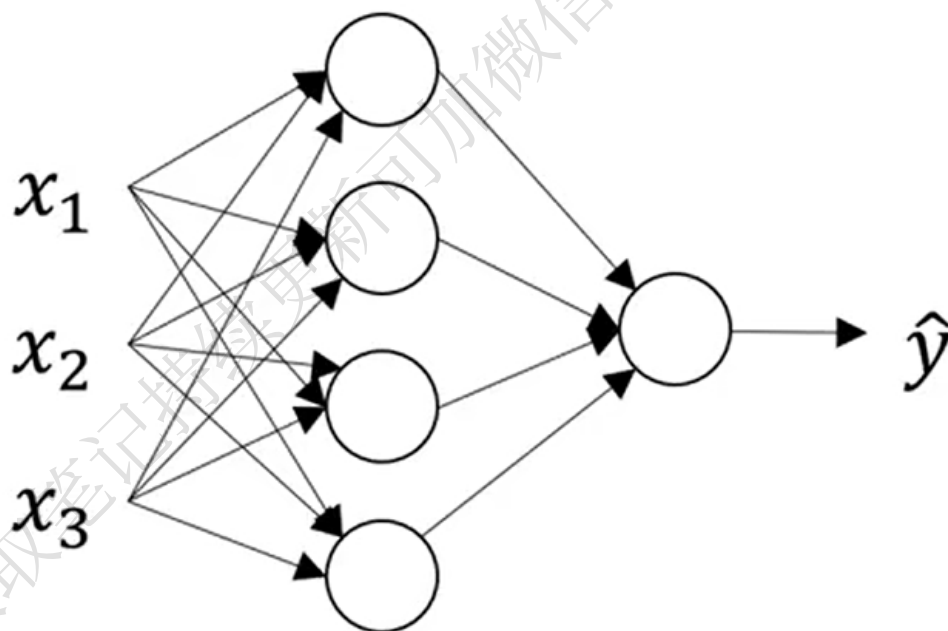
- 比如下述神经元表示回归计算的两个步骤;



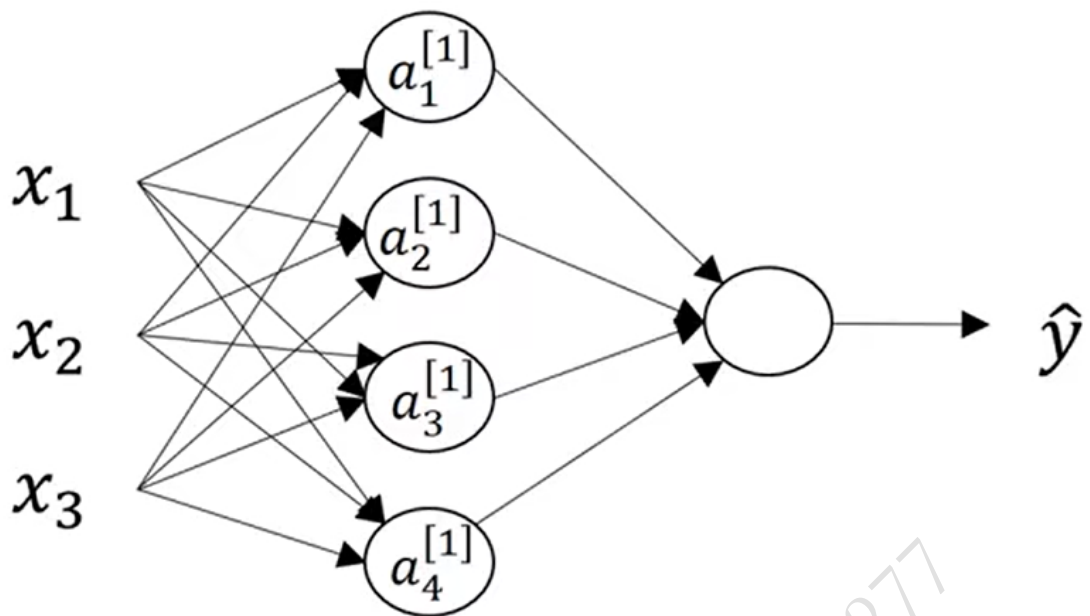
$$z = w^T x + b$$

$$a = \sigma(z)$$

- 而神经网络则表示很多的上述神经元构成的网络，并重复很多次；



- 示例：神经网络的计算
 - 图示：神经网络



- 分析:

- 四个第一层的结点:

$$\begin{aligned} z_1^{[1]} &= w_1^{[1]T} x + b_1^{[1]}, a_1^{[1]} = \sigma(z_1^{[1]}) \\ z_2^{[1]} &= w_2^{[1]T} x + b_2^{[1]}, a_2^{[1]} = \sigma(z_2^{[1]}) \\ z_3^{[1]} &= w_3^{[1]T} x + b_3^{[1]}, a_3^{[1]} = \sigma(z_3^{[1]}) \\ z_4^{[1]} &= w_4^{[1]T} x + b_4^{[1]}, a_4^{[1]} = \sigma(z_4^{[1]}) \end{aligned}$$

- 向量化:

$$\begin{bmatrix} w_1^{[1]T} \\ w_2^{[1]T} \\ w_3^{[1]T} \\ w_4^{[1]T} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \\ b_4^{[1]} \end{bmatrix} = \begin{bmatrix} \rightarrow w_1^{[1]T} x + b_1^{[1]} \\ \rightarrow w_2^{[1]T} x + b_2^{[1]} \\ \rightarrow w_3^{[1]T} x + b_3^{[1]} \\ \rightarrow w_4^{[1]T} x + b_4^{[1]} \end{bmatrix} = \begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \\ z_4^{[1]} \end{bmatrix}$$

- 结果: 四个方程

$$\begin{aligned} Z^{[1]} &= W^{[1]}x + b^{[1]} \\ a^{[1]} &= \sigma(z^{[1]}) \\ Z^{[2]} &= W^{[2]}a^{[1]} + b^{[2]} \\ a^{[2]} &= \sigma(z^{[2]}) \end{aligned}$$

多层神经网络

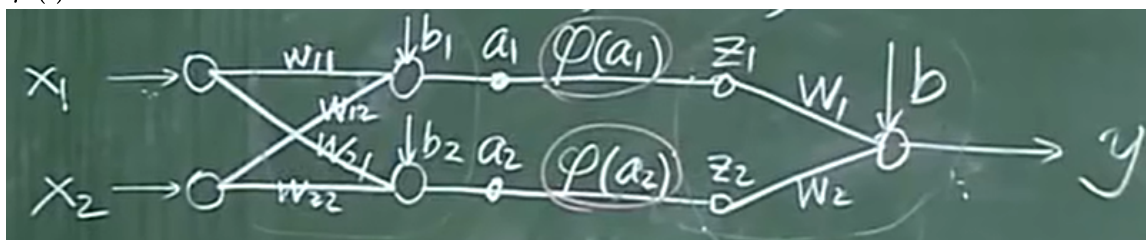
概念: 多层神经网络

- 引入:

- 生活中大部分问题都是非线性可分的, 因此不能只是用线性平面来分类它, 要用非线性的函数集合来分开非线性的样本;

- 概念:

- 假设 $\varphi(\cdot)$ 是一个非线性函数;



- 其中:

$$a_1 = w_{11}x_1 + w_{12}x_2 + b_1$$

$$a_2 = w_{21}x_1 + w_{22}x_2 + b_2$$

$$z_1 = \varphi(a_1)$$

$$z_2 = \varphi(a_2)$$

- 最后合并: $y = w_1z_1 + w_2z_2 + b$
- 假设没有非线性函数, 则 $y = w_1z_1 + w_2z_2 + b$ 依然是线性的, 这里面所有的 W 和 b 都是要被学习的;
- 如果不引入 $\varphi(\cdot)$ 非线性函数, 则最终神经元依然是线性的;

- 概念: 非线性函数 $\varphi(\cdot)$

- 最初引进的非线性函数是一个 0-1 阶跃函数;

定理: 三层神经网络可以模拟所有的决策面

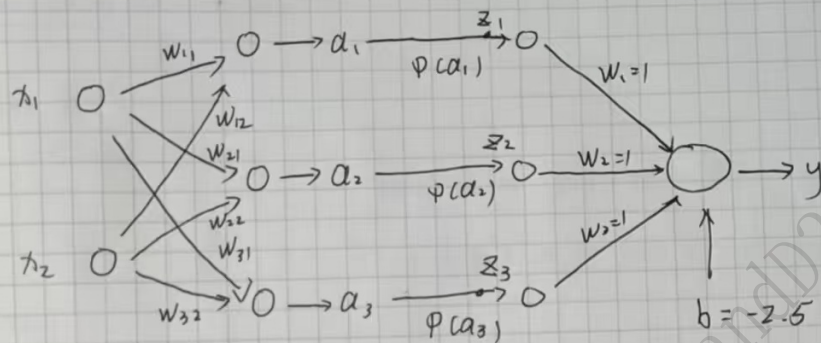
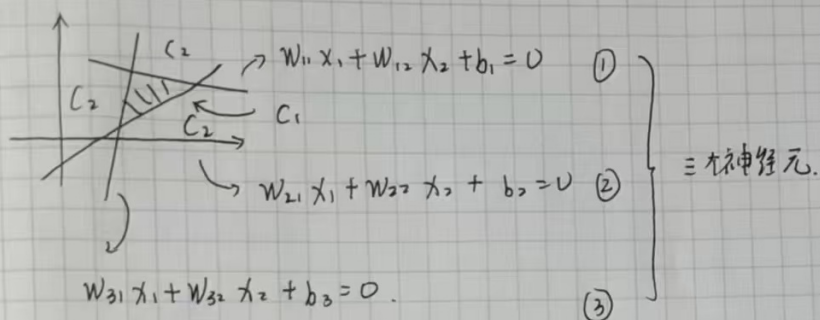
- 假设:

- 有一个平面, 其中一部分区域是 C1, 一部分区域是 C2, 只有当点 (x_1, x_2) 落在 C1 区域时、结果才大于 0;
- 假设是三条直线相交的区域为 C1, 三条直线为 $W_{21}X_1 + W_{22}X_2 + b_2 = 0$ 等等;

- 分析:

- 用两层神经网络, 模拟 C1 决策面:

假设: $\square C_1$ $\square C_2$



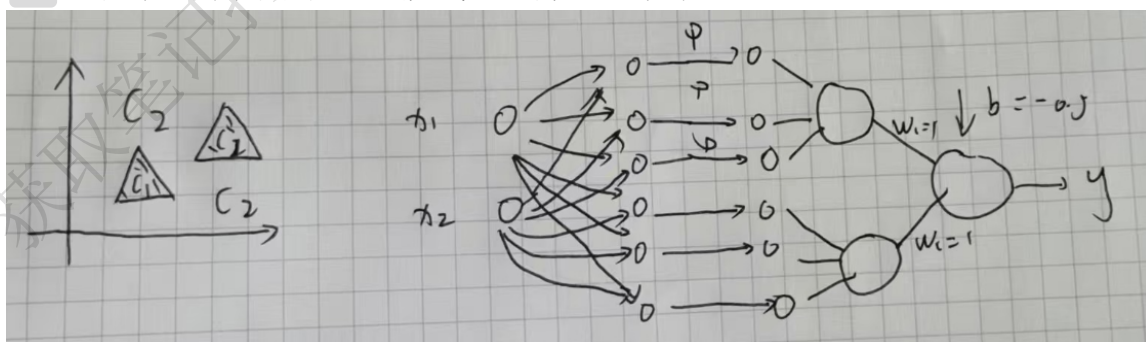
(当 $\phi(a_1)=1, \phi(a_2)=1, \phi(a_3)=0$

则 $y = 2 - 2.5 = -0.5 < 0$.)

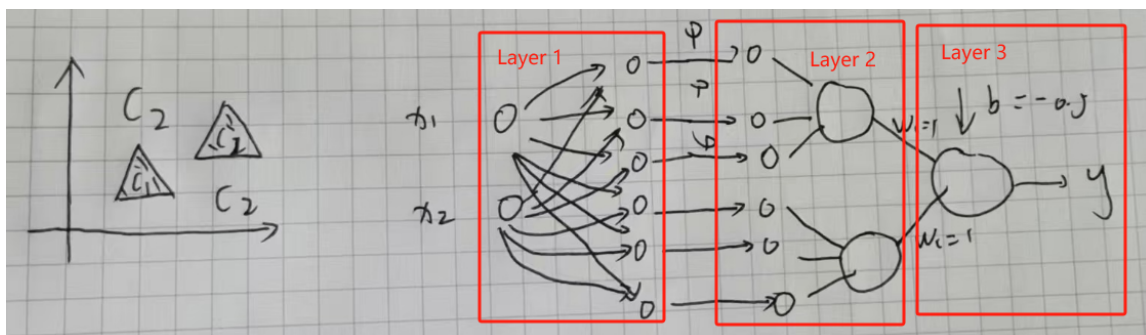
当 $\phi(a_1)=1, \phi(a_2)=1, \phi(a_3)=1$

则 $y = 3 - 2.5 = 0.5 > 0$.)

- 假设 C_1 是一个四边形时, 则此时需要四个神经元, 此时 $b=-3.5$;
- 假设 C_1 是一个圆, 则此时可以用 n 条直线来模拟一个圆, 则神经元个数为 n ;
- 假设 C_1 是一个非规则形状, 则可以用 n 条直线来模拟, 则神经元个数为 n ;
- 假设 C_1 是两个三角形构成的区域, 则此时需要 6 个神经元:



- 可知, 三层神经网络可以模拟所有决策面;



概念：多层神经网络的特点

- 优点：
 - (1) 基本单元简单，多个基本单元可扩展为非常复杂的非线性函数。因此易于构建，同时模型有很强的表达能力。
 - (2) 训练和测试的计算并行性非常好，有利于在分布式系统上的应用；
 - (3) 模型构建来源于对人脑的仿生，话题丰富，各种领域都能做贡献；
- 缺点：
 - (1) 数学不漂亮，优化算法只能获得局部极值，算法性能与初始值有关；
 - (2) 不可解释。训练神经网络获得的参数与实际任务的关联性非常模糊；
 - (3) 模型可调整的参数很多（网络层数、每层神经元个数非线性函数、学习率、优化方法、终止条件等等）；
 - (4) 如果要训练相对复杂的网络，需要大量的训练样本；

后向传播算法

梯度下降法求局部极值

概念：梯度下降法求局部极值

- 概念：梯度下降法求局部极值
 - 梯度下降法求出来的极值点和初始值有很大关系，因为它是沿着当前函数下降的方向移动，因此一直移动的结果、就是到达当前区域的极小值后，不再移动，但如果换其他初始值，则可能会有其他结果；
- 实现：
 - (1) 找一个 w_0
 - (2) 设 $k=0$:
 - 假设满足以下时退出：

$$\frac{df(w)}{dw}|_{w_k} = 0$$

- 否则：

$$W_{k+1} = W_k - a \frac{df(w)}{dw}|_{w_k}$$

- 解释：
 - 泰勒展开：

$$f(w + \Delta w) = f(w) + \frac{df(w)}{dw} \cdot \Delta w + o(\Delta w)$$

- 则：

$$f(w_{k+1}) = f(w_k) + \left(\frac{df(w)}{dw}|_{w_k} \right) \left(-a \frac{df(w)}{dw}|_{w_k} \right) + o(w) = f(w_k) - a \left[\frac{df(w)}{dw} \right]^2 + o(\Delta w)$$

- 这里的梯度只负责方向，而每次在这个方向上走多远，可以自己调节参数；

- 核心：先找到一个方向，然后选择一个方法、来决定走什么样的路；
- 后向传播法是一个典型的梯度下降法；

后向传播法基础概念

概念：后向传播法基础概念

- 目的：
 - 输入为 $\{(x_i, Y_i)\}_{i=1-N}$ ，想得到神经网络中的所有参数、使得 $y = Y_i$ ；

• 概念：

- 针对输入 (x, y) ，定义一个目标函数：

$$E = \frac{1}{2}(y - Y)^2$$

- 每输入一个 x 能够算出一个 E ；

• 实现：

- (1) 随机的取变量：($W_{11}W_{12}W_{21}W_{22}b_1b_2W_1W_2b$)
- (2) 对所有 W ：求 $\frac{\partial E}{\partial w}$ ；对所有 b ：求 $\frac{\partial E}{\partial b}$
- (3)

$$W^{(\text{新})} = W^{(\text{旧})} - a \frac{\partial E}{\partial W} \Big|_{w^{(\text{旧})}}$$

$$b^{(\text{新})} = b^{(\text{旧})} - a \frac{\partial E}{\partial b} \Big|_{b^{(\text{旧})}}$$

- (4) 当所有的 $\frac{\partial E}{\partial W}$ 和 $\frac{\partial E}{\partial b}$ 等于 0 时，退出循环；

• 分析：后向传播

- 第一步：当输入了 x 之后，先前向的计算、得到 Y ，这个过程当中会涉及到 W 和 b ；
- 第二步：后向传播，从 $y-a$ ，从 $a-w$ 和 $a-b$ ，即从后往前、求偏导，一直求到 $\frac{\partial E}{\partial W}$ 和 $\frac{\partial E}{\partial b}$ ；

概念：对于非线性函数 $\varphi(\cdot)$ 的改造

- 方法：使用 Sigmoid

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

- 非常好的性质： $\varphi'(x) = \varphi(x)[1 - \varphi(x)]$

• 方法：使用双曲正切函数

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

- 非常好的性质： $\varphi'(x) = 1 - [\varphi(x)]^2$

• 分析：以上两种函数的问题

- 当 x 十分大的时候，在 x 很大的区域、梯度信息被压缩掉了，因为梯度十分小；
- 而下面的 ReLU 函数，只对 x 小于 0 的部分进行压缩，其他部分维持其原本的样子；

- 这样导致的结果是：使得神经网络当中，某一层活跃的那些参数减少、同时在很大规模的神经网络中，使用 BP 算法时、降低每一层神经元的活跃参数，即"干掉一些神经元，减少过拟合"；

- 方法：使用 **ReLU** 函数

$$\varphi(x) = \begin{cases} x & x > 0 \\ 0 & x \leq 0 \end{cases} = \max(0, x)$$

- 非常好的性质： $\varphi'(x) = \begin{cases} 1 & x > 0 \\ 0 & x \leq 0 \end{cases}$

- 方法：使用 **Leak ReLU**

$$\varphi(x) = \begin{cases} x & x > 0 \\ \beta x & x < 0 \end{cases}$$

- 求导： $\varphi'(x) = \begin{cases} 1 & x > 0 \\ \beta & x \leq 0 \end{cases}$

反向传播算法推导

概念：后向传播算法推导

- 假设：
 - 1. 网络共有 **l** 层；
 - 2. $Z^{(k)}, a^{(k)}, b^{(k)}$ 是第 K 层的向量，与第 K 层的神经元个数一致；
 - 3. 用 $Z_i^{(k)}, a_i^{(k)}, b_i^{(k)}$ 表示是 $Z^{(k)}, a^{(k)}, b^{(k)}$ 的第 i 个分量；
 - 4. 用 y_i 表示 y 的第 i 个分量；
- 概念：BP 算法过程
 - 前向传播：
 - (1) 随机初始化 (w, b)
 - (2) 训练样本 (X, Y)，代入网络，可以求出所有的 (Z, a, y)
 - 后向传播：
 - (3) 链式求偏导，求最小化 $E = \frac{1}{2} \|y - Y\|^2$
 - (4) 求所有的 $\frac{\partial E}{\partial W}$ 和 $\frac{\partial E}{\partial b}$

$$\delta_i^{(m)} = \frac{\partial E}{\partial z_i^{(m)}}$$

- 1. 得到求导的第一层：

$$\delta_i^{(l)} = \frac{\partial E}{\partial z_i^{(l)}} = \frac{\partial E}{\partial y_i} \frac{\partial y_i}{\partial z_i^{(l)}} = (y_i - Y_i) \varphi'(z_i^{(c)})$$

- 2. 递推公式：求从 $1 \leq m \leq (l-1)$ ，从后一层求出前一层

$$\delta_i^{(m)} = \frac{\partial E}{\partial z_i^{(m)}} = \frac{\partial E}{\partial a_i^{(m)}} \cdot \frac{\partial a_i^{(m)}}{\partial z_i^{(m)}} = \varphi(z_i^{(m)})' \sum_{j=1}^{S_{m+1}} \delta_j^{(m+1)} \cdot W_{ji}$$

- 3. 求出 $\frac{\partial E}{\partial W}$ 和 $\frac{\partial E}{\partial b}$:

$$\frac{\partial E}{\partial W_{ij}^{(m)}} = \delta_j^{(m)} a_i^{(m-1)}$$

$$\frac{\partial E}{\partial b_i^{(m)}} = \delta_i^{(m)}$$

- (5) 求出 $\frac{\partial E}{\partial W}$ 和 $\frac{\partial E}{\partial b}$ 后, 更新参数

$$W^{(\text{新})} = W^{(1\text{月})} - a \frac{\partial E}{\partial w}$$

$$b^{(\text{新})} = b^{(1\text{日})} - a \frac{\partial E}{\partial b} \Big|_{b^{(1\text{月})}}$$

参数设置

一般的训练原则

概念: 一般的训练原则

- 原则:
 - (1) 一般情况下, 在训练集上的目标函数的平均值 (**cost**) 会随着训练的深入而不断减小, 如果这个指标有增大情况, 停下来;
 - 有两种情况: 第一是采用的模型不够复杂, 以致于不能在训练集上完全拟合; 第二是已经训练很好了;
 - (2) 分出一些验证集 (**Validation Set**), 训练的本质目标是在验证集上获取最大的识别率。因此训练一段时间后, 必须在验证集上测试识别率, 保存使验证集上识别率最大的模型参数作为最后结果;
 - (3) 注意调整学习率 (**Learning Rate**), 如果刚训练几步 **cost** 就增加, 一般来说是学习率太高了; 如果每次 **cost** 变化很小, 说明学习率太低;
 - 涉及到: 是否收敛, 以及收敛率;
- 举例:
 - 图示:

```

[N,M] = size(xapp);
p = randperm(M);
ratioTraining = 0.4;
ratioValidation = 0.1;
ratioTesting = 0.5;
xTraining = [];
yTraining = [];
for i=1:floor(ratioTraining*M)
    xTraining = [xTraining,xapp(:,p(i))];
    yTraining = [yTraining,yapp(:,p(i))];
end;
xTraining = xTraining';
yTraining = yTraining';
maxAccuracy = 0;
iteration = 0;
totalAccuracy = [];
while(1)
    iteration = iteration +1;
    nn = nn_train(nn, option, xTraining, yTraining);
    totalCost(iteration) = sum(nn.cost)/length(nn.cost);
    % plot(totalCost);
    [wrongs, accuracy] = nn_test(nn, xValidation, yValidation);
    totalAccuracy = [totalAccuracy, accuracy];
    if accuracy>maxAccuracy
        maxAccuracy = accuracy;
        storedNN = nn;
    end;
    cost = totalCost(iteration);
    accuracy
    cost

```

- 分析:

- 百分之四十做训练, 百分之十做验证, 百分之五十做测试;

```
ratioTraining = 0.4;
```

```
ratioValidation = 0.1;
```

```
• ratioTesting = 0.5;
```

- 把测试集、验证集全部打乱;

- 首先计算目标函数的平均值 (cost) , 然后计算它的准确率;

```
• totalCost(iteration) = sum(nn.cost)/length(nn.cost);
```

- 如果识别率比已有的更大, 则保存当前的正确率:

```
if accuracy>maxAccuracy
```

```
    maxAccuracy = accuracy;
```

```
    storedNN = nn;
```

```
• end;
```

- 最后在测试集上做测试:

```
end;
```

```
• [wrongs, accuracy] = nn_test(storedNN,xTesting,yTesting);
```

- 如果学习率特别高, 可能会导致其发生震荡, 从而无法找到局部极值;

- 如果学习率特别低, 则可能导致其虽然发现了正确的方向、但是得走很久;

随机梯度下降

概念: 随机梯度下降

- 概念:

- Stochastic Gradient Descent, SGD

- (1) 不用每输入一个样本就去变换参数, 而是输入一批样本 (叫做一个 BATCH 或 MINI-BATCH), 求出这些样本的梯度平均值后, 根据这个平均值改变参数;

- 也就是每一次进来一批样本值, 然后求出梯度后, 求平均;

- 这样的目的是稍微的减少随机性, 这样可以使得训练的过程的噪音减少;

- (2) 在神经网络训练中, BATCH 的样本数大致设置为 50-200 不等;

- 示例:

- 图示:

```

batch_size = option.batch_size;
m = size(train_x,1);
num_batches = m / batch_size;
for k = 1 : iteration
    kk = randperm(m);
    for l = 1 : num_batches
        batch_x = train_x(kk((l - 1) * batch_size
+ 1 : l * batch_size), :);
        batch_y = train_y(kk((l - 1) * batch_size
+ 1 : l * batch_size), :);
        nn = nn_forward(nn,batch_x,batch_y);
        nn = nn_backpropagation(nn,batch_y);
        nn = nn_applygradient(nn);
    end
end

```

• 解释:

- m 是训练样本的个数;
- 根据 batch_size 将其分成很多个小块;
- `kk = randperm(m);` 做的也是打乱;
- `for l = num_batches`: 每一次取出一个 batch
- 这三句:

```

nn = nn_forward(nn,batch_x,batch_y);
nn = nn_backpropagation(nn,batch_y);
nn = nn_applygradient(nn);

```

- 第一句是求前向传播;
- 第二句是求后向传播;
- 第三句是在改变 W 和 b;

• 前向计算:

```

nn.cost(s) = 0.5 / m * sum(sum((nn.a{k} - batch_y).^2))
+ 0.5 * nn.weight_decay * cost2;

```

• 后向传播:

```

nn.W_grad{nn.depth-1} =
nn.theta{nn.depth}*nn.a{nn.depth-1}'/m +
nn.weight_decay*nn.W{nn.depth-1};
nn.b_grad{nn.depth-1} = sum(nn.theta{nn.depth},2)/m;

```

激活函数

概念: 激活函数

• 概念:

- 方法: 使用 Sigmoid

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

- 非常好的性质: $\varphi'(x) = \varphi(x)[1 - \varphi(x)]$
- 方法: 使用双曲正切函数

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

- 非常好的性质: $\varphi'(x) = 1 - [\varphi(x)]^2$

- 方法: 使用 **ReLU** 函数

- $$\varphi(x) = \begin{cases} x & x > 0 \\ 0 & x \leq 0 \end{cases} = \max(0, x)$$

- 非常好的性质: $\varphi'(x) = \begin{cases} 1 & x > 0 \\ 0 & x \leq 0 \end{cases}$

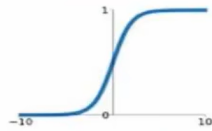
- 方法: 使用 **Leak ReLU**

- $$\varphi(x) = \begin{cases} x & x > 0 \\ \beta x & x < 0 \end{cases}$$

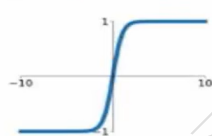
- 求导: $\varphi'(x) = \begin{cases} 1 & x > 0 \\ \beta & x \leq 0 \end{cases}$

- 图示:

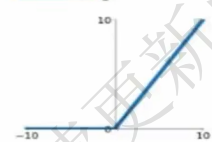
Sigmoid
 $\sigma(x) = \frac{1}{1+e^{-x}}$



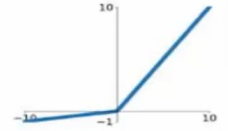
tanh
 $\tanh(x)$



ReLU
 $\max(0, x)$



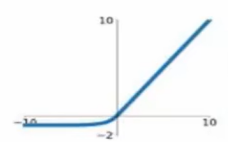
Leaky ReLU
 $\max(0.1x, x)$



Maxout
 $\max(w_1^T x + b_1, w_2^T x + b_2)$

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$

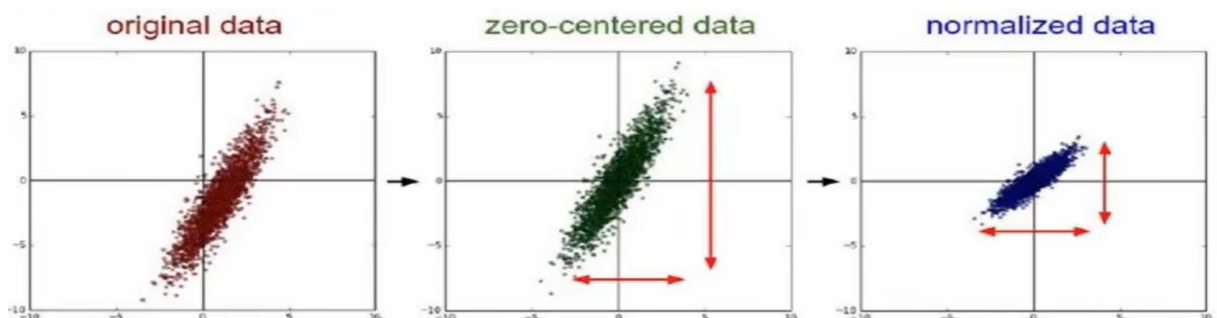


训练数据初始化

概念: 训练数据初始化

- 概念:
 - (建议) 对训练来的数据做均值和方差的归一化;
- 目的:
 - 输入的特征, 每一个维度对后面的影响都差不多;

- 图示:



参数的初始化

概念: (W, b) 的初始化

- 问题:

- Sigmoid 和 tanh 函数存在梯度消失的问题;

梯度消失现象:如果 $W^T X + b$ 一开始很大或很小,那么梯度将趋近于0,反向传播后前面与之相关的梯度也趋近于0,导致训练缓慢。

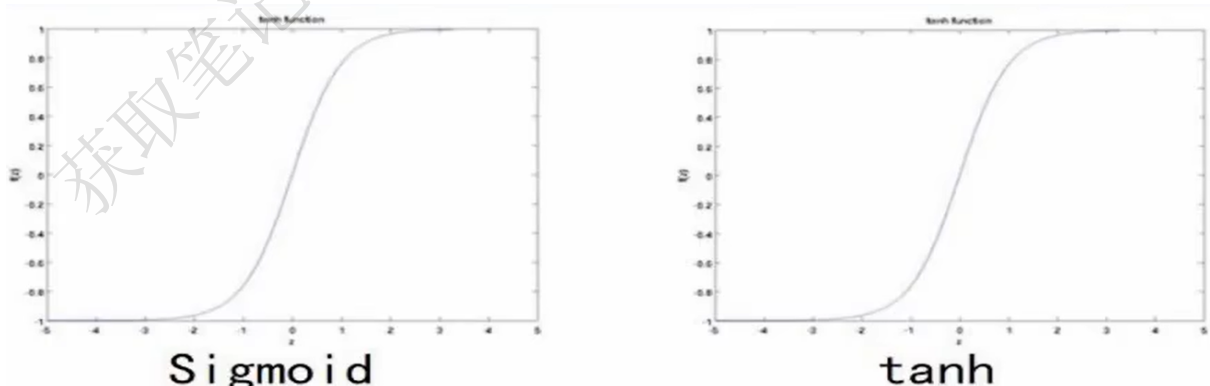
因此,我们要使 $W^T X + b$ 一开始在零附近。

- 当梯度趋向于 0 时,因为当前是在做链式求导的过程,因此在最后的几层梯度趋向于 0 的话,往前传播时、梯度也趋向于 0;
 - 这样将会使得训练的过程变得漫长;
 - 不过也可以将学习率调整;
 - 因此:想要使得 $W^T X + b$ 一开始就能在 0 的附近;

- 做法:

- 一种比较简单有效的方法是: (W, b) 初始化从区间 $(-\frac{1}{\sqrt{d}}, \frac{1}{\sqrt{d}})$ 均匀随机取值。其中 d 为 (W, b) 所在层的神经元个数;
 - 可以证明,如果 X 服从正态分布,均值 0, 方差 1, 且各个维度无关,而 (W, b) 是 $(-\frac{1}{\sqrt{d}}, \frac{1}{\sqrt{d}})$ 的均匀分布,则 $w^T x + b$ 是均值为 0, 方差为 $1/3$ 的正态分布;
 - 而如果 $W^T X + b$ 在方差为 $1/3$ 的区域内时,可以对其进行比较好的控制;
 - 也就是:梯度太低了,可能会导致梯度消失,进而导致训练的时间很长;
 - 而如果全部集中在中间(比如 sigmoid 函数的中间附件),则可能又导致其退化为线性函数;
 - 但如果想要梯度太大,进而反向传递更多的梯度值回去,则又必然导致其在某一部分是发散的,训练的难度也将变得更大;
 - 最核心的目标,还是让 W 和 b 在零附近徘徊;

- 图示:



Batch Normalization

概念: Batch Normalization

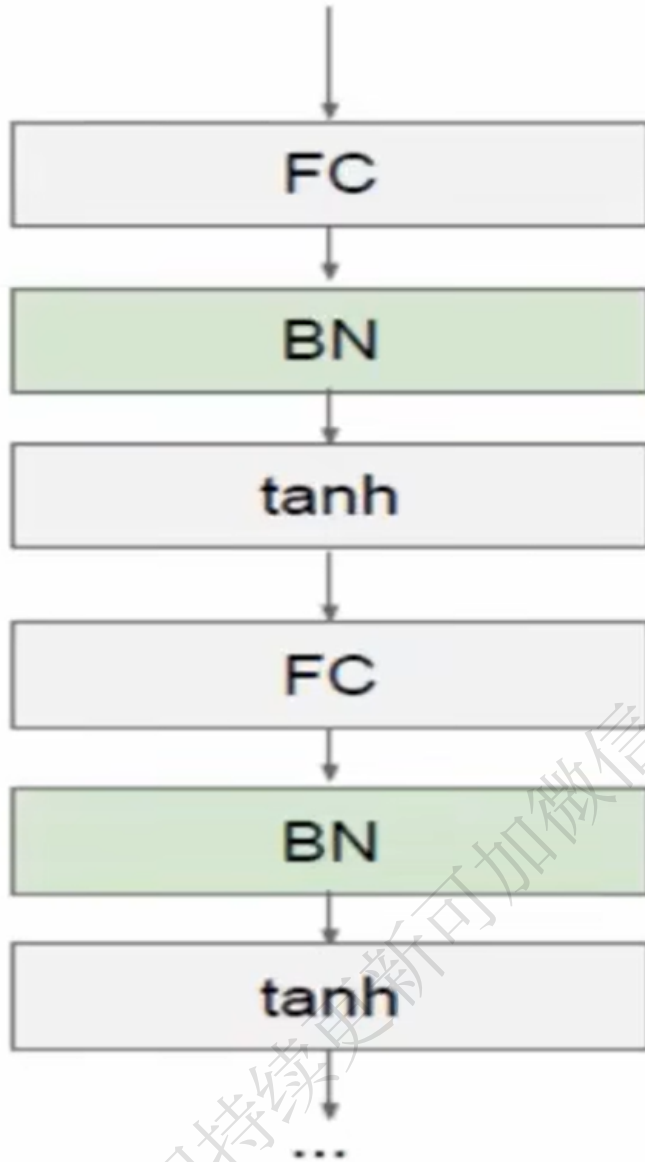
- 概念:

- 既然我们希望每一层获得的值都在 0 附近,从而避免梯度消失现象,那么我们为什么不直

接把每一层的值做基于均值和方差的归一化呢？

- 实现：

- 每一层 FC (Fully Connected Layer) 接一个 BN (Batch Normalization)；



- 减掉均值，除以方差：

$$\hat{x}^{(k)} = \frac{x^{(k)} - \mathbb{E}[x^{(k)}]}{\sqrt{\text{Var}[x^{(k)}]}}$$

- 结果：

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_1 \dots x_m\}$;

Parameters to be learned: γ, β

Output: $\{y_i = \text{BN}_{\gamma, \beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^m x_i \quad // \text{ mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_{\mathcal{B}})^2 \quad // \text{ mini-batch variance}$$

$$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \quad // \text{ normalize}$$

$$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma, \beta}(x_i) \quad // \text{ scale and shift}$$

- 为什么要做最后一步：因为在经过了上面的步骤，将每一层的结果归一化后，集中在 0 附近，进而又导致结果过于线性化。因此需要再对其进行放缩；

- 论文：

- Batch normalization accelerating deep network training by reducing internal covariate shift (2015)

- 图示：

- 前向传播：

```
y = nn.W{k-1} * nn.a{k-1} + repmat(nn.b{k-1}, 1, m);
if nn.batch_normalization
    nn.E{k-1} = nn.E{k-1} * nn.vecNum + sum(y, 2);
    nn.S{k-1} = nn.S{k-1} .^ 2 * (nn.vecNum - 1) + (m - 1) * std(y, 0, 2) .^ 2;
    nn.vecNum = nn.vecNum + m;
    nn.E{k-1} = nn.E{k-1} / nn.vecNum;
    nn.S{k-1} = sqrt(nn.S{k-1} / (nn.vecNum - 1));
    y = (y - repmat(nn.E{k-1}, 1, m)) ./ repmat(nn.S{k-1}, 1, m) + 0.0001 * ones(size(nn.S{k-1})), 1, m);
    y = nn.Gamma{k-1} * y + nn.Beta{k-1};
end;
switch nn.activation_function
case 'sigmoid'
    nn.a{k} = sigmoid(y);
case 'tanh'
    nn.a{k} = tanh(y);
```

- 后向传播：

```
nn.theta{k} = ((nn.W{k}' * nn.theta{k+1})) .* nn.a{k} .* (1 - nn.a{k});
if nn.batch_normalization
    x = nn.W{k-1} * nn.a{k-1} + repmat(nn.b{k-1}, 1, m);
    x = (x - repmat(nn.E{k-1}, 1, m)) ./ repmat(nn.S{k-1}, 1, m) + 0.0001 * ones(size(nn.S{k-1})), 1, m);
    temp = nn.theta{k} .* x;
    nn.Gamma_grad{k-1} = sum(mean(temp, 2));
    nn.Beta_grad{k-1} = sum(mean(nn.theta{k}, 2));
    nn.theta{k} = nn.Gamma{k-1} * nn.theta{k} ./ repmat((nn.S{k-1} + 0.0001), 1, m);
end;
nn.W_grad{k-1} = nn.theta{k} * nn.a{k-1}' / m + nn.weight_decay * nn.W{k-1};
nn.b_grad{k-1} = sum(nn.theta{k}, 2) / m;
```

目标函数选择

概念：目标函数选择

- 概念：可加正则项

$$L(W) = F(W) + R(W)$$
$$= \frac{1}{2} \left(\sum_{i=1}^{batch_size} \|y_i - Y_i\|^2 + \beta \sum_k \sum_l W_{k,l}^2 \right)$$

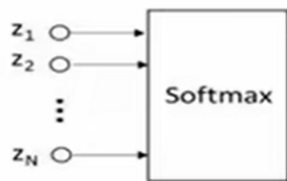
- 图示：

```
cost2 = cost2 + sum(sum(nn.W{k-1}.^2));  
nn.cost(s) = 0.5 / m * sum(sum((nn.a{k} - batch_y).^2))  
+ 0.5 * nn.weight_decay * cost2;
```

概念：SOFTMAX 函数和交叉熵函数

- 概念：
 - 如果是分类问题，可以使用 SOFTMAX 函数和交叉熵函数的组合；

(a) SOFTMAX函数：



$$q_i = \frac{\exp(z_i)}{\sum_{j=1}^N \exp(z_j)}$$

我们希望通过此网络学习由 $Z = \begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_N \end{bmatrix}$ 到 $P = \begin{bmatrix} p_1 \\ p_2 \\ \vdots \\ p_N \end{bmatrix}$ 的映射。其中 $\sum_{i=1}^N p_i = 1$

(b) 交叉熵 (Cross Entropy) :

$$\text{定义目标函数为 } E = - \sum_{i=1}^N p_i * \log(q_i)$$

深度学习数据库

概念：深度学习数据库

- 概念：Mnist

Mnist: 手写数字数据库 (LeCun 在1998年创造)

- (1) 手写数字 0-9共10类
- (2) 训练样本60000个, 测试样本10000个。
- (3) 图像大小 28*28 二值图像。
- (4) 样例:



- 概念: ImageNet

ImageNet: (Fei-fei Li等 2007年创造)

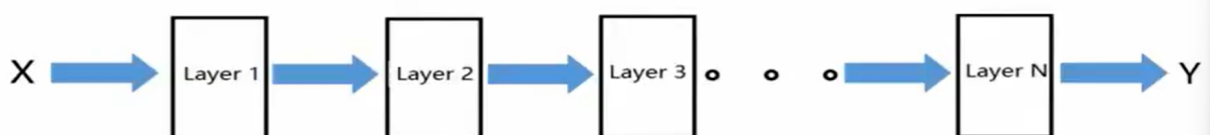
- (1) 1000类, 100多万张 (2009年的规模)
- (2) 图片大小: 正常图片大小, 像素几百*几百
- (3) WORDNET结构, 拥有多个Node (节点)。一个node (目前) 含有至少500个对应物体的可供训练的图片/图像。



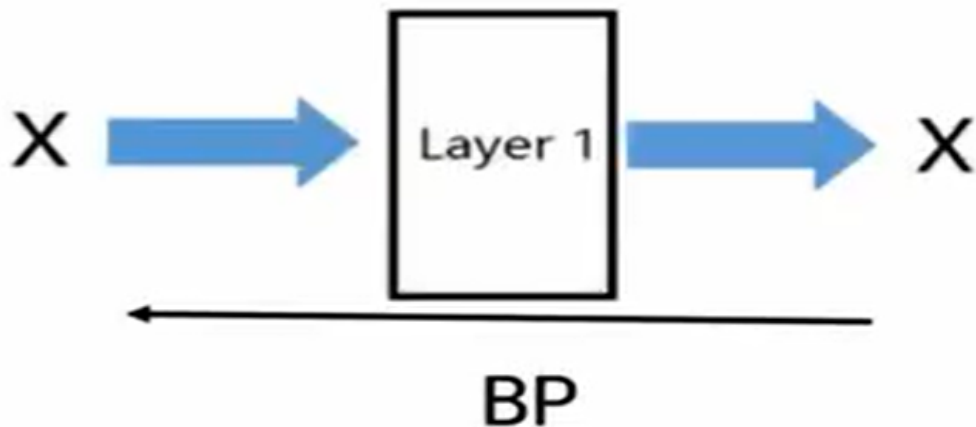
深度学习自编码器

概念: 深度学习自编码器

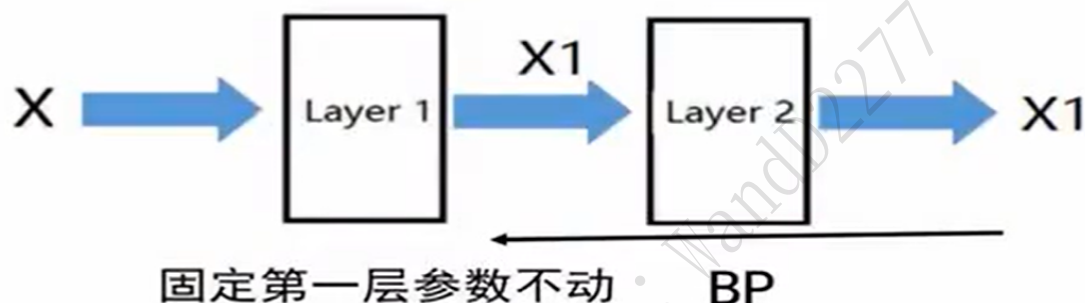
- 引入:
 - 2006 年是深度学习的起始年, Hinton 在 SCIENCE 上发文, 提出一种叫做自动编码器 (Auto-encoder) 的方法, 部分解决了神经网络参数初始化的问题;
- 概念: 自动编码器
 - 解决 (W, b) 参数初始化问题, 例如训练如下网络:



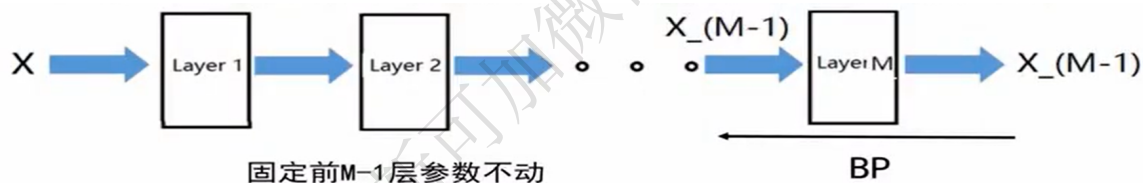
- 步骤一: 训练以下网络, 输入是 X, 经过 Layer 1 后, 输出还是 X;



- 步骤二：保留左边的参数，右边的取消，接着训练第二层：



- 步骤 N：以此类推，训练好第 M-1 层后，接着训练第 M 层



- 即：BP 算法向后传播时，固定住前面的 M-1 层的参数，只对第 M 层的参数进行更改。同时因为第 M 层输入的 X 是从第 M-1 层输出的，因此 X 通过 M-1 层时、依然保留了数据的本质特征；
- 也就是说：前面的层次将输入数据压缩成一个或多个维度较低表示，并能够在解码过程中恢复原始数据；
- 最后再用 BP 算法对神经网络进行微调；
- 分析：

- 在注意力机制中，`Softmax` 函数用于计算注意力权重，确保权重和为 1；

获取笔记持续更新可加微信：WandD2277