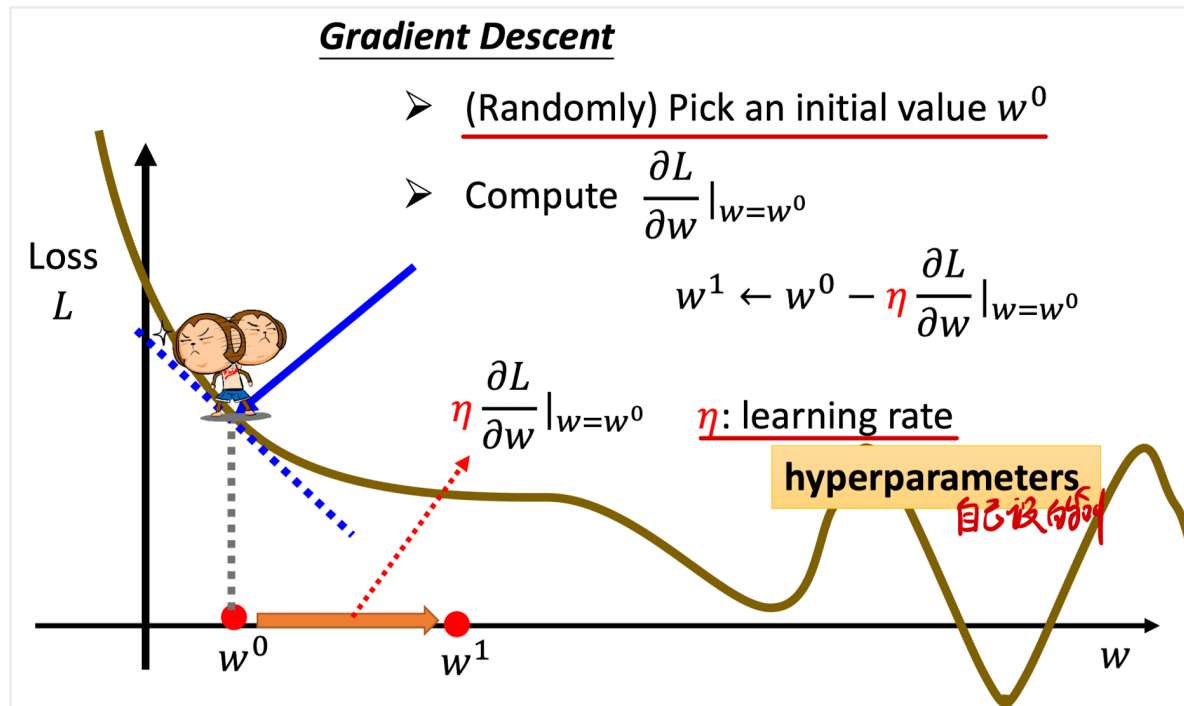


1.1 Introduction of Machine Learning

- gradient descent
- Sigmoid Function
- example

gradient descent



线性回归的梯度下降不会存在局部最小值的问题

在线性回归中，损失函数通常使用均方误差（Mean Squared Error, MSE）。对于简单的线性回归，损失函数是凸的（convex），这意味着它只有一个全局最小值，没有局部最小值。因此，对于线性回归，梯度下降法不会陷入局部最小值。

为了更具体地理解这一点，考虑线性回归的均方误差损失函数：

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)})^2$$

其中， $h_{\theta}(x)$ 是由特征 x 和参数 θ 定义的假设函数， m 是样本数量。这个损失函数相对于参数 θ 是二次的，形成了一个抛物面。因此，其形状是碗状的，有一个明确的最小值点。

Sigmoid Function

1. Sigmoid函数:

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

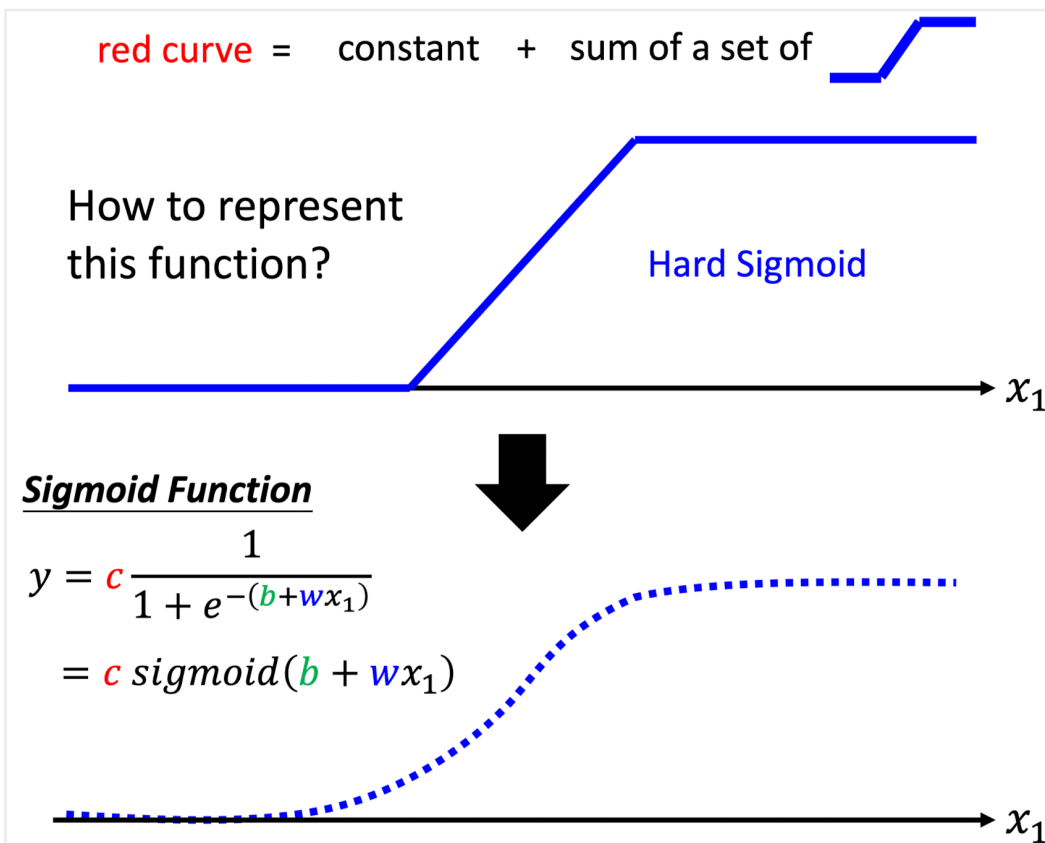
Sigmoid函数可以将任意实数映射到0和1之间。它的形状类似于一个平滑的"S"形。

2. Hard Sigmoid函数:

Hard sigmoid函数是sigmoid函数的一个简化版本，它的计算更加高效，但牺牲了一些精确度。它通常用线性段来近似sigmoid函数，例如：

$$\text{hard_sigmoid}(x) = \max(0, \min(1, \frac{x}{2} + 0.5))$$

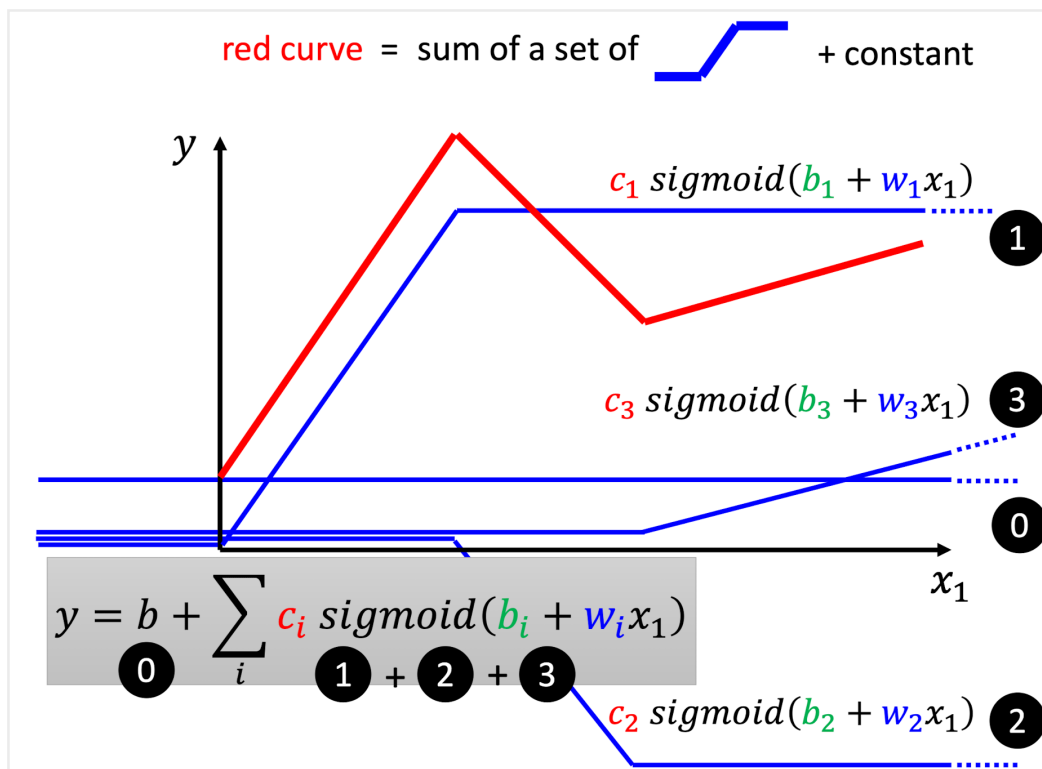
这实际上是一个两段的线性函数，对于输入在某个范围内，它的斜率是固定的，并且在0和1之间进行裁剪。



- Different **w**: Change slopes(坡度)
- different **b**: shift(平移)
- different **c**: Change height(高度)

通过设置不同的 c , b , w 来逼近不同的 continuous functions


用前一天的数据 x_1 来预测 $y = b + w \cdot x_1$ (model bias) → 设置更有弹性的有未知参数的 function



考虑前j天的数据 (用多个 feature)

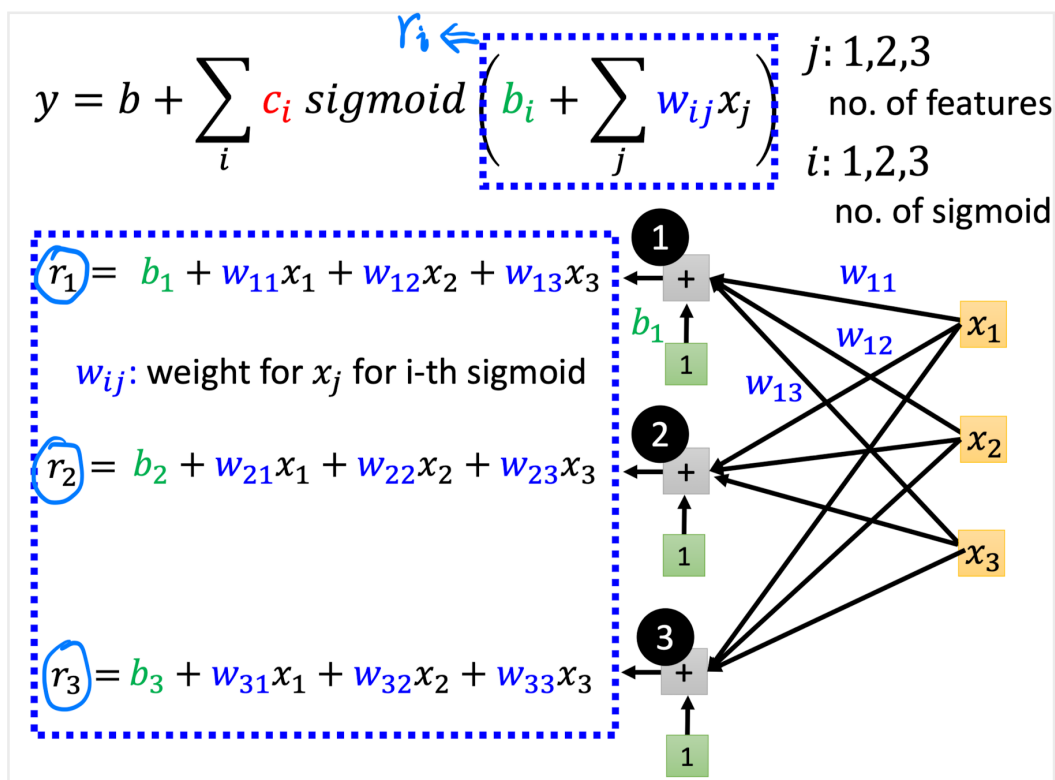
$$y = b + \sum_j w_j x_j \quad \text{考虑前j天的数据}$$

↓

$$y = b + \sum_i c_i \text{sigmoid} \left(\underline{b_i} + \sum_j \underline{w_{ij} x_j} \right)$$


具体举例：

1. 每个i代表一个sigmoid function, 每个sigmoid function(r1,r2,r3) 里面做的就是
一个矩阵和向量相乘



$$y = b + \sum_i c_i \text{sigmoid} \left(b_i + \sum_j w_{ij} x_j \right)$$

$i: 1,2,3$
 $j: 1,2,3$

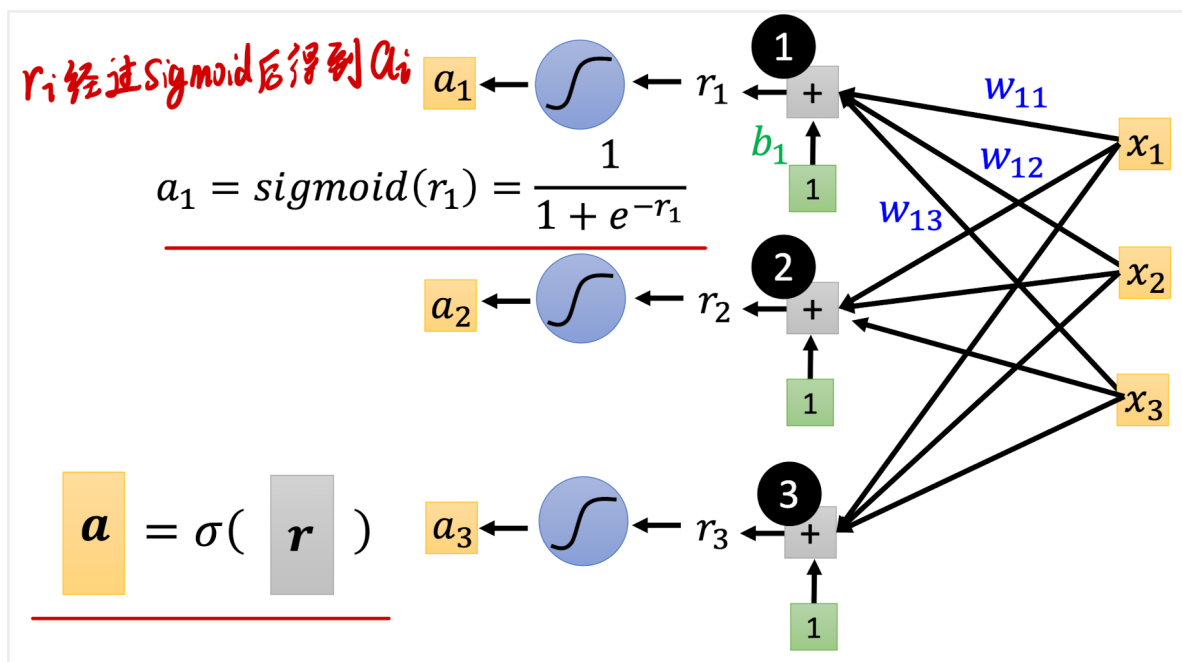
$r_1 = b_1 + w_{11}x_1 + w_{12}x_2 + w_{13}x_3$
 $r_2 = b_2 + w_{21}x_1 + w_{22}x_2 + w_{23}x_3$
 $r_3 = b_3 + w_{31}x_1 + w_{32}x_2 + w_{33}x_3$

矩阵
向量

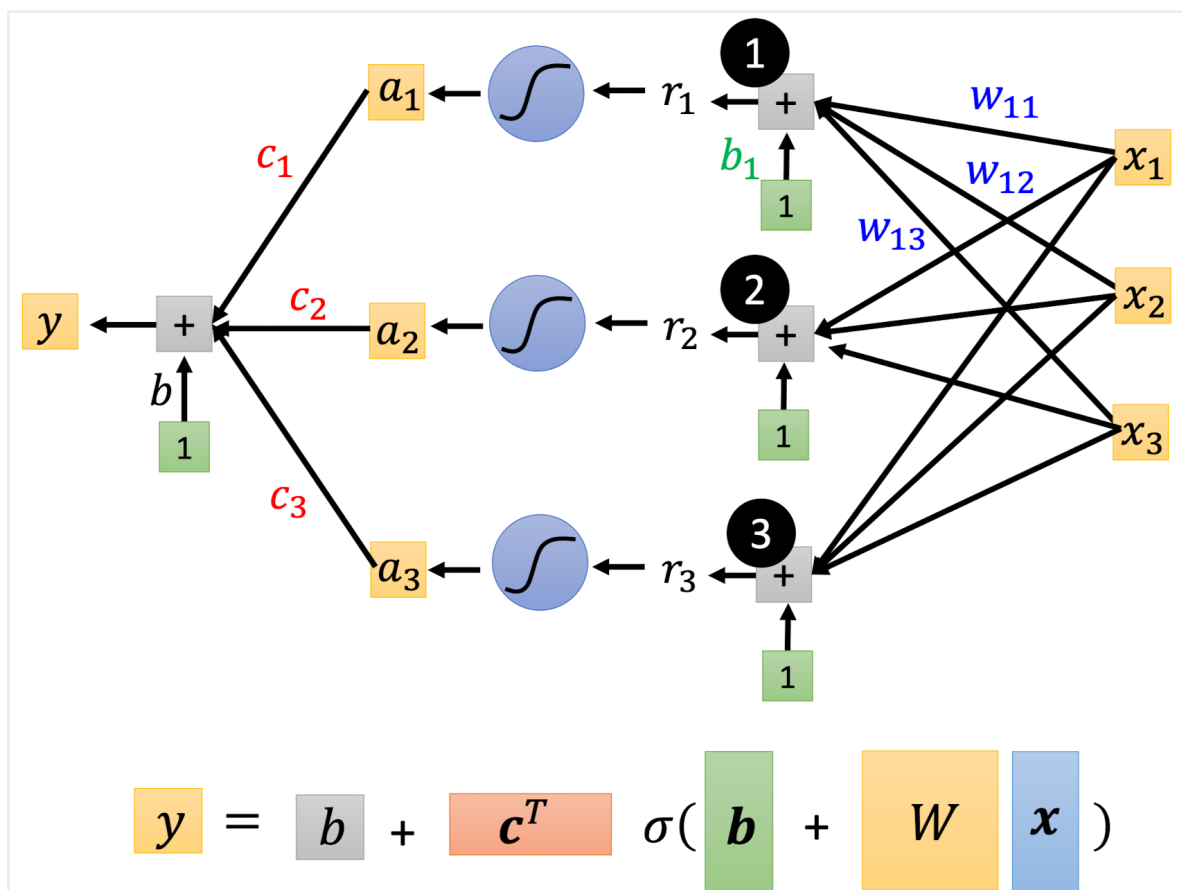
$$\begin{bmatrix} r_1 \\ r_2 \\ r_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} + \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

$\mathbf{r} = \mathbf{b} + \mathbf{W} \mathbf{x}$

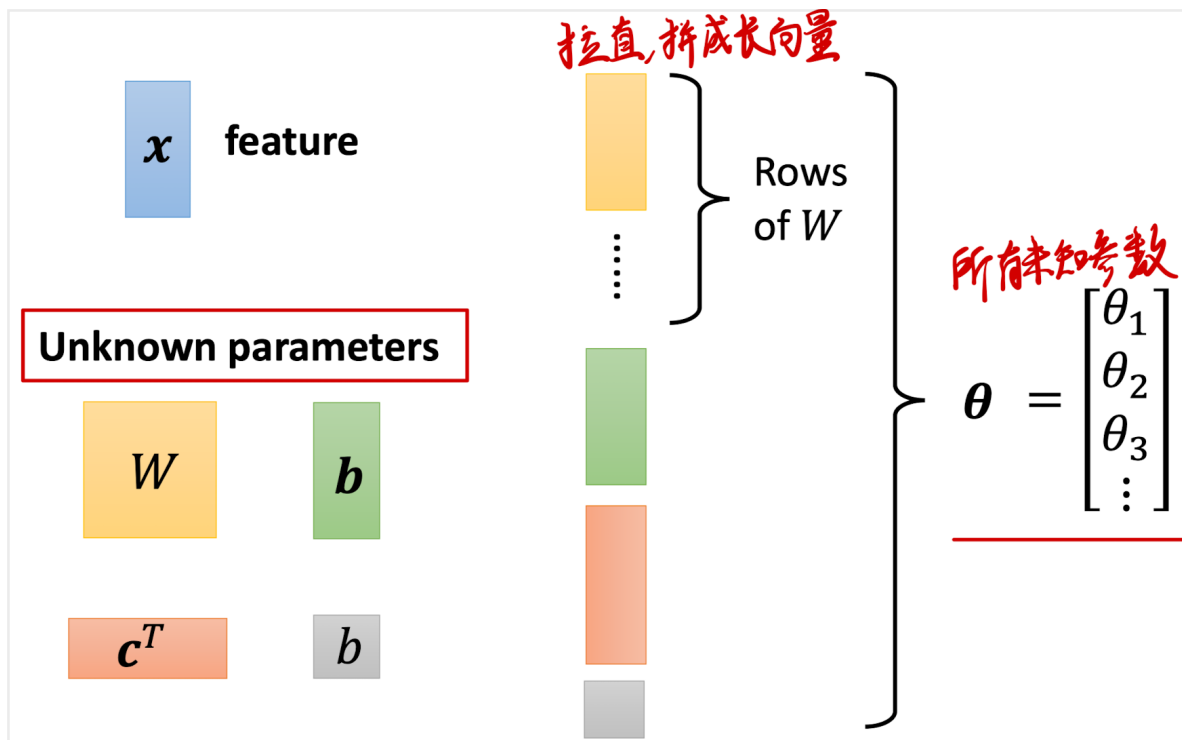
2. 然后每个 r_i 都经过 sigmoid function 后变成 a_i , 然后 a_i 乘上 c_i 的和加上 b 得到 y



3. 图示表示



重新定义符号：feature: \mathbf{x} (已知), unknow parameters: $\mathbf{W}, \mathbf{b}, \mathbf{c}^T, b$, 把所有未知参数放到一起。



4. 然后用梯度下降求参数的最优解

Optimization of New Model

$$\theta^* = \arg \min_{\theta} L$$

$$\theta = \begin{bmatrix} \theta_1 \\ \theta_2 \\ \theta_3 \\ \vdots \end{bmatrix}$$

➤ (Randomly) Pick initial values θ^0

计算每个未知参数微分, 叫 g (red text)

$$g = \begin{bmatrix} \frac{\partial L}{\partial \theta_1} |_{\theta=\theta^0} \\ \frac{\partial L}{\partial \theta_2} |_{\theta=\theta^0} \\ \vdots \end{bmatrix}$$

gradient (red text)

$$\begin{bmatrix} \theta_1^1 \\ \theta_2^1 \\ \vdots \end{bmatrix} \leftarrow \begin{bmatrix} \theta_1^0 \\ \theta_2^0 \\ \vdots \end{bmatrix} - \begin{bmatrix} \eta \frac{\partial L}{\partial \theta_1} |_{\theta=\theta^0} \\ \eta \frac{\partial L}{\partial \theta_2} |_{\theta=\theta^0} \\ \vdots \end{bmatrix}$$

简写 (red text)

$$g = \nabla L(\theta^0)$$

在 $\theta=\theta^0$ 时对每个参数的微分 (red text)

$$\theta^1 \leftarrow \theta^0 - \eta g$$

更新 θ^0 为 θ^1 (red text)

$\theta^* = \arg \min_{\theta} L$

- (Randomly) Pick initial values θ^0
- Compute gradient $\mathbf{g} = \nabla L^1(\theta^0)$
 update $\theta^1 \leftarrow \theta^0 - \eta \mathbf{g}$
- Compute gradient $\mathbf{g} = \nabla L^2(\theta^1)$
 update $\theta^2 \leftarrow \theta^1 - \eta \mathbf{g}$
- Compute gradient $\mathbf{g} = \nabla L^3(\theta^2)$
 update $\theta^3 \leftarrow \theta^2 - \eta \mathbf{g}$

1 **epoch** = see all the batches once

先随机分成N组batch

不是拿L更新, 而是拿 L_1, L_2, L_3 更新

HyperParameter: 人所设的东西, 不是机器找出来的, 如 learning rate, batch size, sigmoid()

两个 ReLU 函数叠起来 = Hard Sigmoid, Sigmoid 和 ReLU 是常见的 **Activation function**

$$y = b + \sum_i c_i \text{sigmoid} \left(b_i + \sum_j w_{ij} x_j \right)$$

Activation function

$$y = b + \sum_{2i} c_i \max \left(0, b_i + \sum_j w_{ij} x_j \right)$$

which is better?

Sigmoid函数

Sigmoid函数定义为：

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

优点：

1. 输出值在0和1之间，有明确的界限。
2. 具有平滑梯度，意味着小的变化在输入可以导致输出的小变化。

缺点：

1. **消失的梯度问题**：当输入值远离0时，sigmoid函数的梯度趋于0，导致权重更新几乎停止，进而使得深度神经网络的训练变得非常困难。
2. 输出不是零中心的：这意味着sigmoid函数的输出总是正的，这可能导致权重更新时出现不必要的偏移。

ReLU函数

ReLU函数定义为：

$$f(x) = \max(0, x)$$

优点：

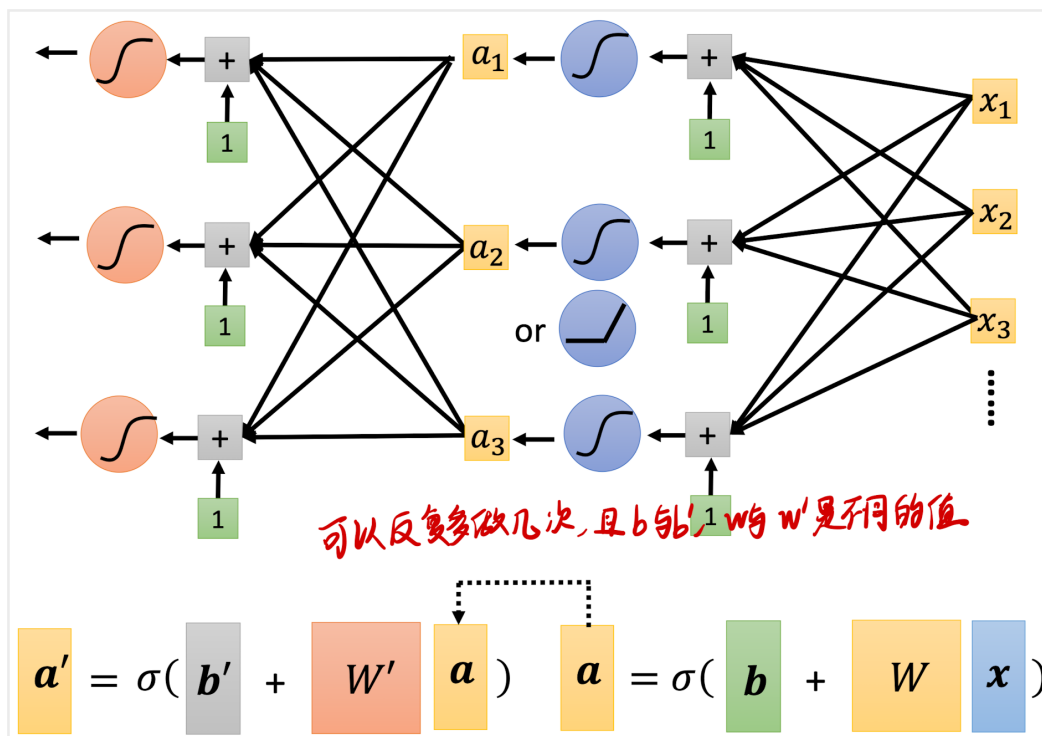
1. 计算效率高：ReLU是一个简单的阈值函数，计算效率比sigmoid高很多。
2. 解决了消失的梯度问题（至少在正数部分）：这使得ReLU在深度学习模型中非常受欢迎，因为它允许更深的网络进行训练。
3. 在实际应用中，ReLU的性能通常优于sigmoid。

缺点：

1. **死亡ReLU问题**：对于某些输入，ReLU会输出0，这可能导致某些神经元在训练过程中“死掉”，即永远不会被激活。这是因为一旦神经元输出0，它再也不能对任何数据产生影响。
2. 输出不是零中心的。

- 在深度学习模型中，ReLU通常是首选的激活函数，因为它的训练速度快并且在大多数场景中表现出色。
- Sigmoid函数通常在输出层用于二分类问题，因为它的输出可以解释为概率。
- 如果ReLU的“死亡”问题成为一个关注点，可以考虑使用其变种，如Leaky ReLU或Parametric ReLU。
- 在选择激活函数时，实际的实验和交叉验证仍然是决定最佳激活函数的关键。

5.可以将同样的动作反复多做几次，这就是另外一个Hyperparameter



上面图示就是 **Neural Network = Deep Learning**

6.实验结果

- Loss for multiple hidden layers
 - 100 ReLU for each layer
 - input features are the no. of views in the past 56 days

	1 layer	2 layer	3 layer	4 layer
2017 – 2020	0.28k	0.18k	0.14k	0.10k
2021	0.43k	0.39k	0.38k	0.44k

Better on training data, worse on unseen data

➡ **Overfitting**

