神经网络基础-numpy实现 C16

信息科学与技术学院

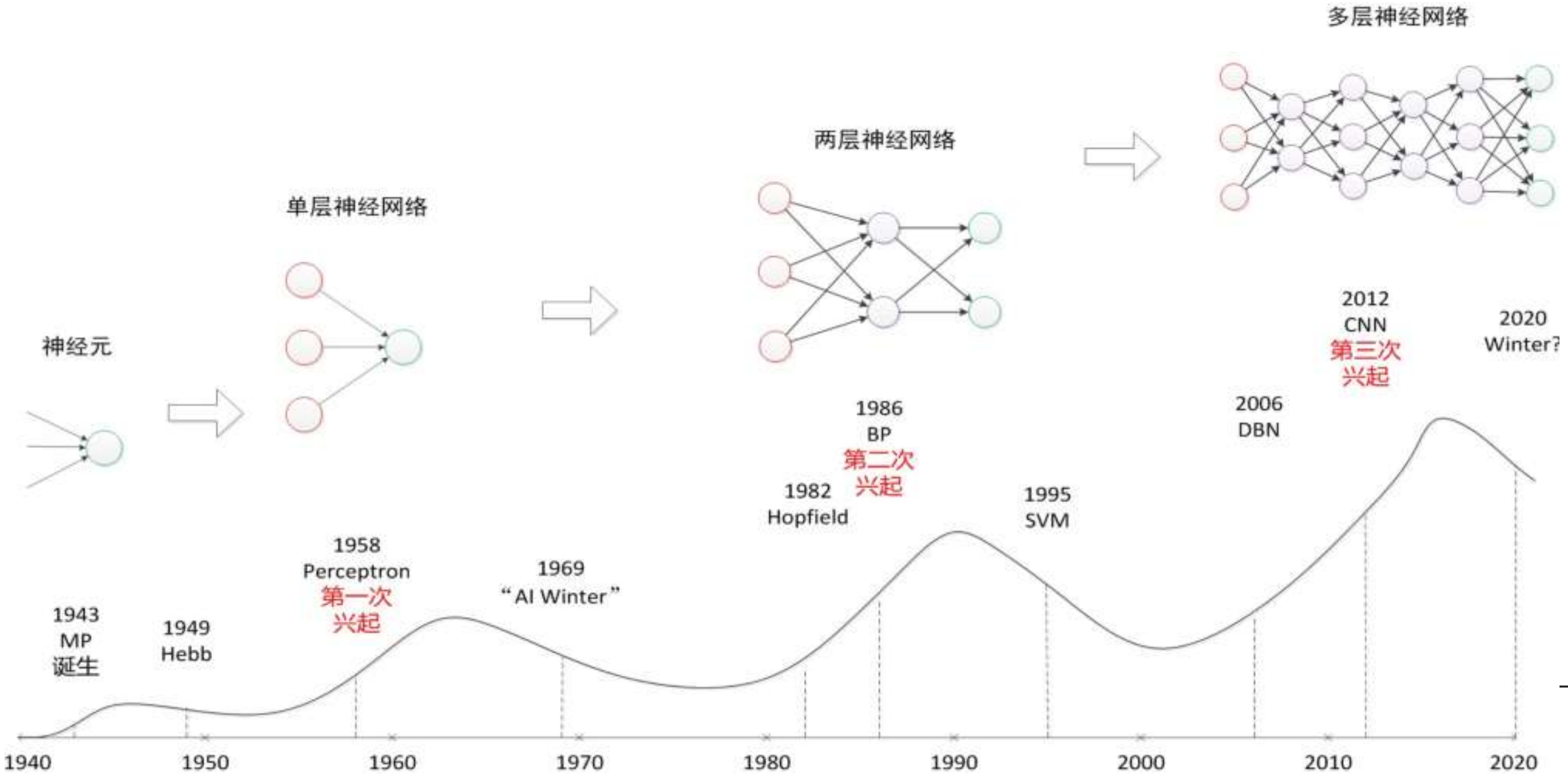
胡俊峰

神经网络基础

- 人工智能与神经网络的历史
- 函数回归与梯度下降法
- 单层神经网络回归
- 多层神经网络建模与反向梯度传播训练
- 用numpy实现一个神经网络

神经网络发展史

- 超大预训练强化模型
- 预训练大模型

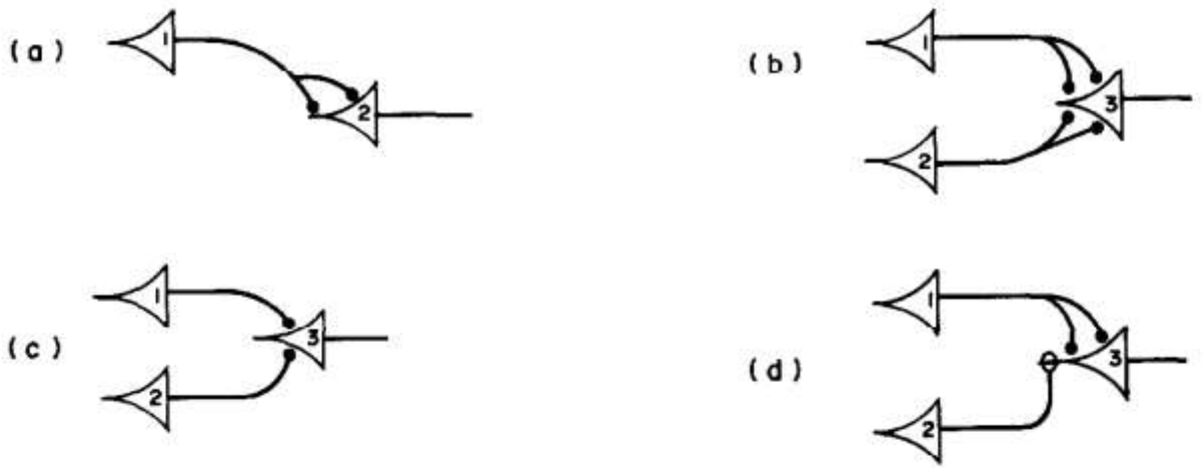


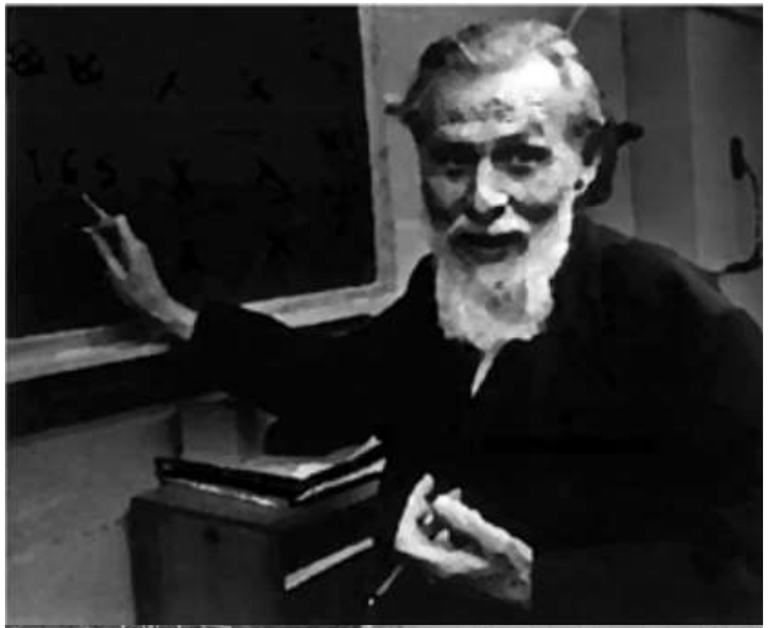
A LOGICAL CALCULUS OF THE IDEAS IMMANENT IN NERVOUS ACTIVITY*

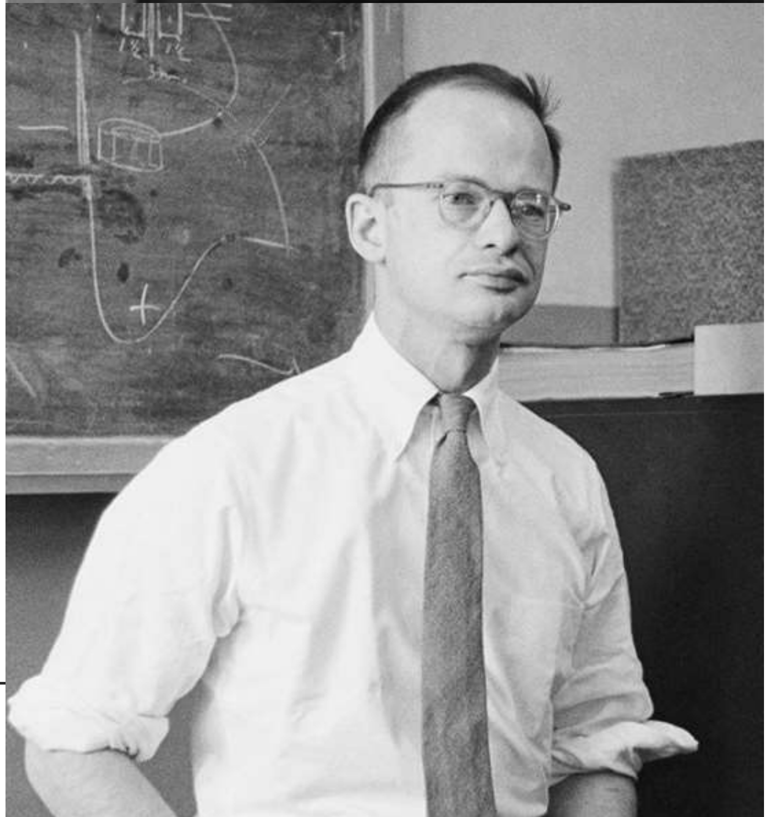
■ WARREN S. MCCULLOCH AND WALTER PITTS
University of Illinois, College of Medicine,
Department of Psychiatry at the Illinois Neuropsychiatric Institute,
University of Chicago, Chicago, U.S.A.

- 提出了神经元网络模型
- 证明了多层感知机方案能模拟任何逻辑算子
- 启动了第一轮人工智能浪潮

LOGICAL CALCULUS FOR NERVOUS ACTIVITY 105





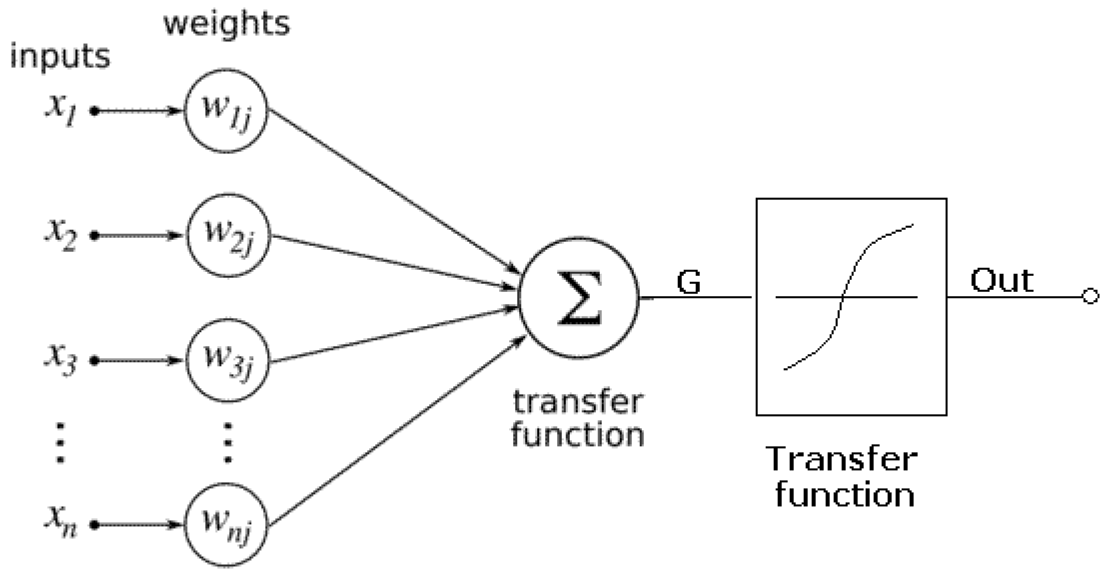


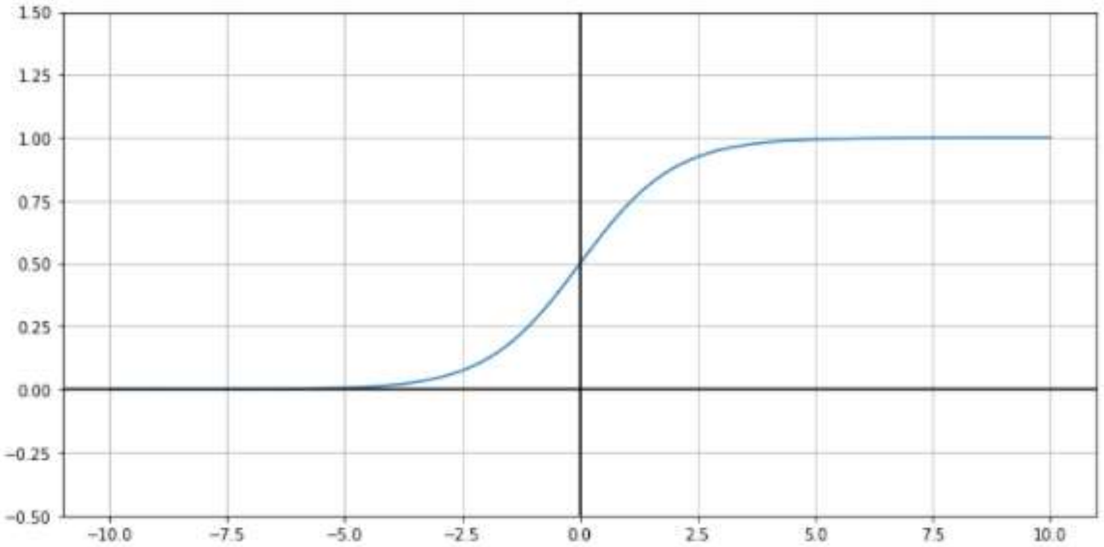
函数拟合与梯度下降回归

- 神经元拟合逻辑计算
- 最小二乘拟合与函数回归

单神经元：sigmoid函数

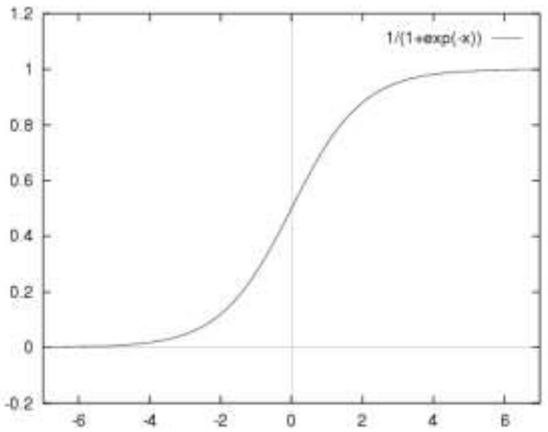
```python
def sigmoid(x):
    """Sigmoid function"""
    return 1.0 / (1.0 + np.exp(-x))
```





单元神经网络拟合布尔逻辑

$$z = w_1 x_1 + w_2 x_2 + b$$

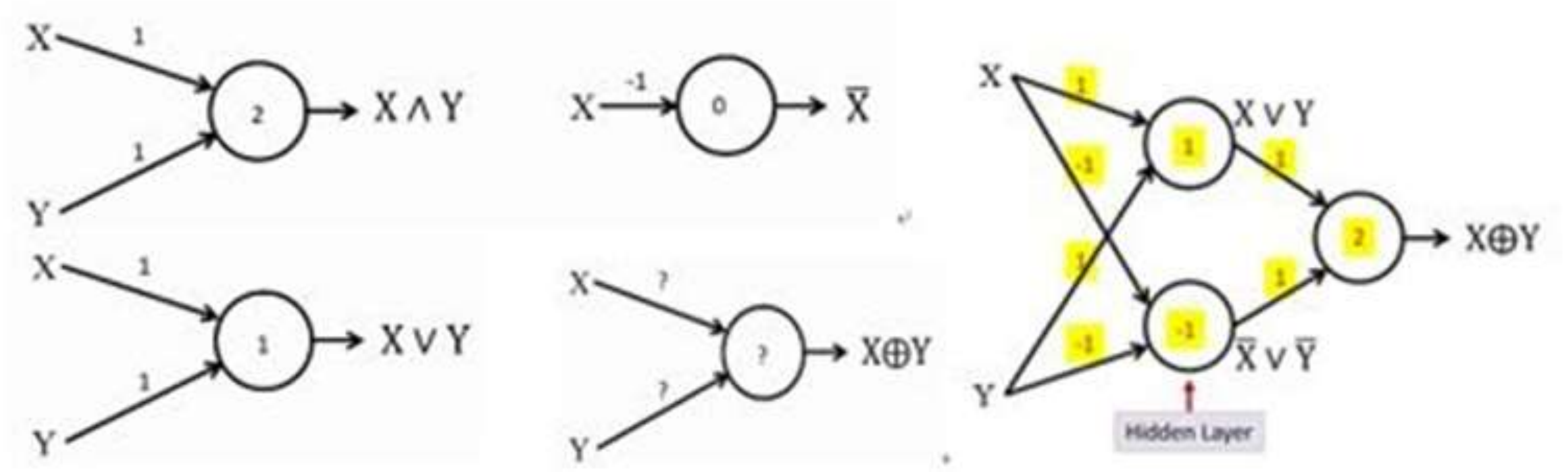



```python
def logic_gate(w1, w2, b):
    # Helper to create logic gate functions
    # Plug in values for weight_a, weight_b, and bias
    return lambda x1, x2: sigmoid(w1 * x1 + w2 * x2 + b)

def test(gate):
    # Helper function to test out our weight functions.
    for a, b in (0, 0), (0, 1), (1, 0), (1, 1):
        print("{}, {}: {}".format(a, b, np.round(gate(a, b))))
```

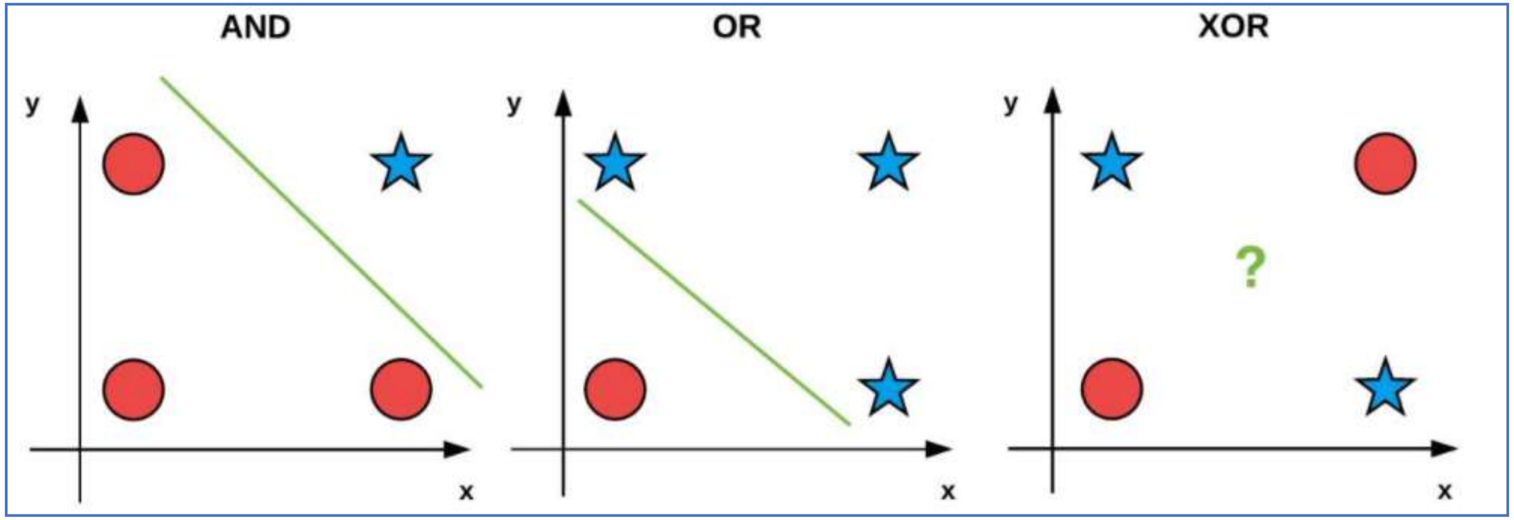
```python
or_gate = logic_gate(20, 20, -10)   # 设置参数实现 或 门
test(or_gate)
```

```
0, 0: 0.0
0, 1: 1.0
1, 0: 1.0
1, 1: 1.0
```

多层网络实现XOR运算



单神经元网络的局限性：XOR问题




```
w1 = -10
w2 = -10
b = 14
nand_gate = logic_gate(w1, w2, b)

test(nand_gate)
```

```
0, 0: 1.0
0, 1: 1.0
1, 0: 1.0
1, 1: 0.0
```

```
def xor_gate(a, b):
    c = or_gate(a, b)
    d = nand_gate(a, b)
    return and_gate(c, d)
test(xor_gate)
```

```
0, 0: 0.0
0, 1: 1.0
1, 0: 1.0
1, 1: 0.0
```

多元函数梯度下降回归

- 多元函数MSEloss
- 多元函数偏导与梯度向量
- 随机梯度下降法求解

线性函数最小二乘回归

```
from sklearn.linear_model import LinearRegression
model = LinearRegression(fit_intercept=True)
print(model)
```

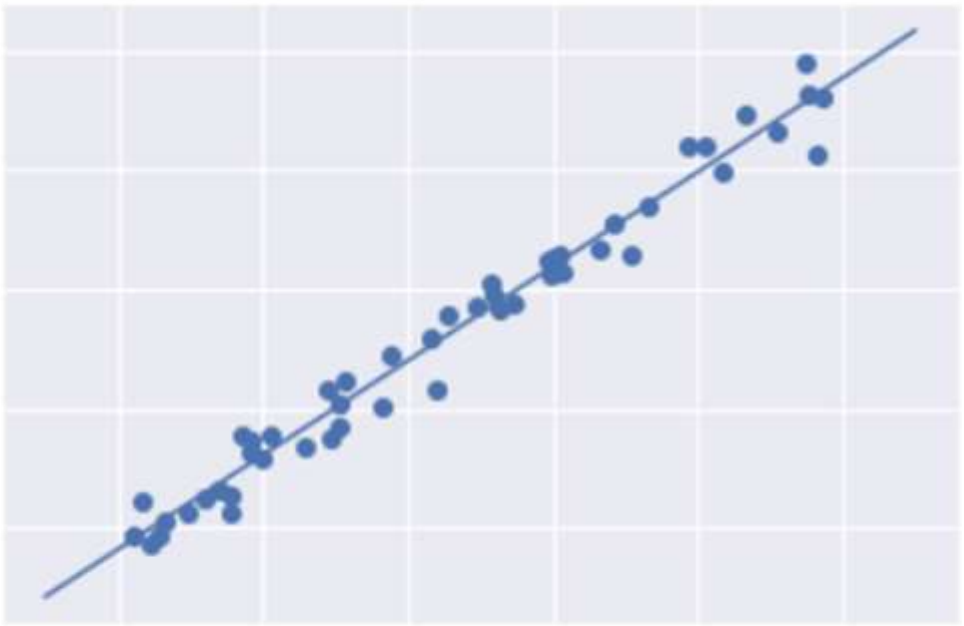
```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=1, normalize=False)
```

```
X = x[:, np.newaxis]
X.shape
```

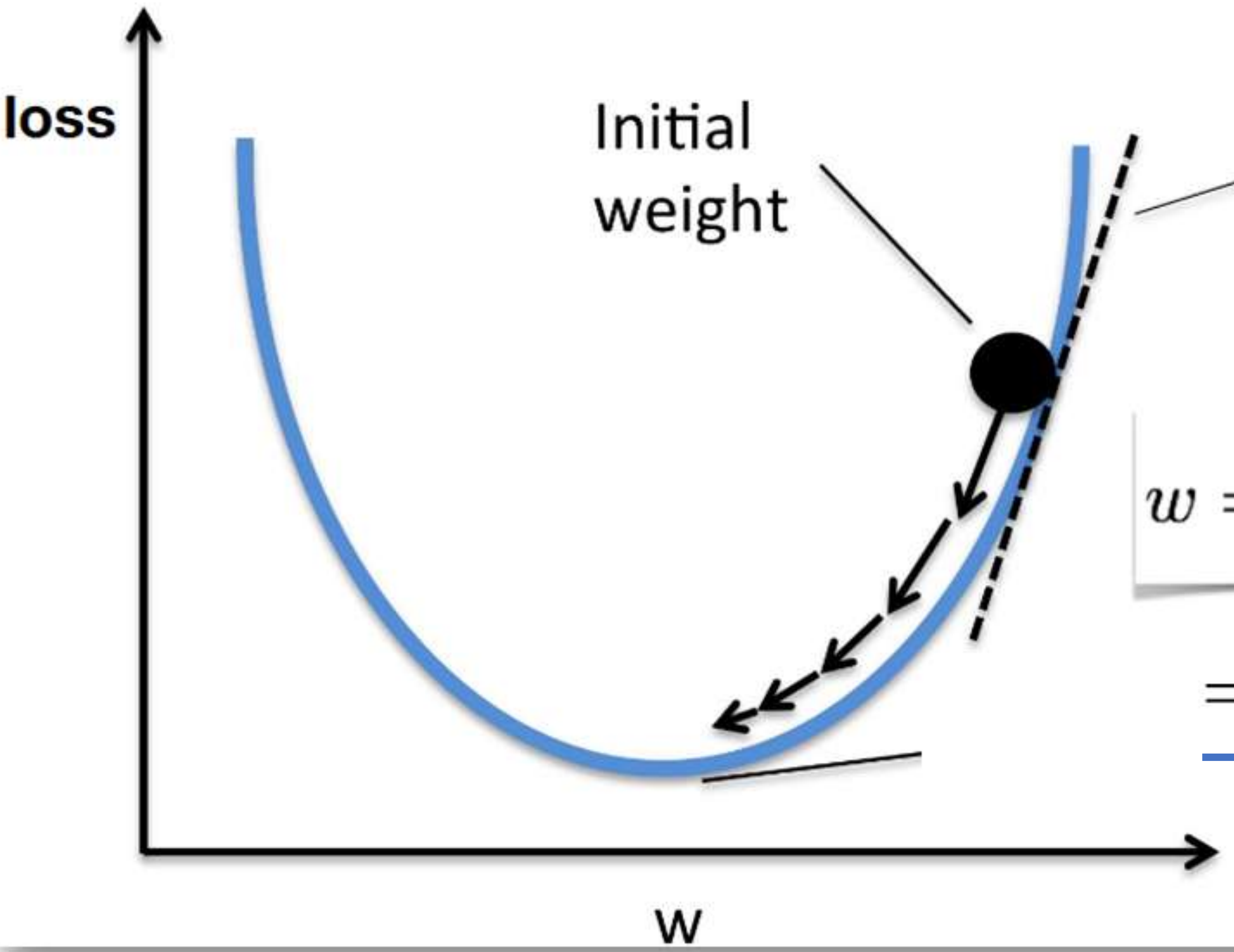
```
(50, 1)
```

```
model.fit(X, y)
```

```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=1,
```



Gradient descent algorithm



Gradient $\frac{\partial loss}{\partial w}$

$$w = w - \alpha \frac{\partial loss}{\partial w}$$

$$= w - \alpha * 2x(xw - y)$$

梯度下降法

MSE loss的偏导与梯度向量（batch size = m）：

$$\frac{\partial}{\partial \theta_j} MSE(\theta) = \frac{2}{m}\sum_{i=1}^{m}(\theta^T \cdot X^{(i)} - y^{(i)})x_j^{(i)}$$

$$\nabla_\theta MSE(\theta) = \begin{bmatrix} \frac{\partial}{\partial \theta_0} MSE(\theta) \\ \frac{\partial}{\partial \theta_1} MSE(\theta) \\ \dots \\ \frac{\partial}{\partial \theta_n} MSE(\theta) \end{bmatrix} = \frac{2}{m} X^T \cdot (X \cdot \theta - y)$$

梯度下降回归：

$$Y = b + \theta_1 X_1 + \theta_2 X_2 + \epsilon$$

- 生成批量数据：

$$b = 1.5, \theta_1 = 2, \theta_2 = 5$$

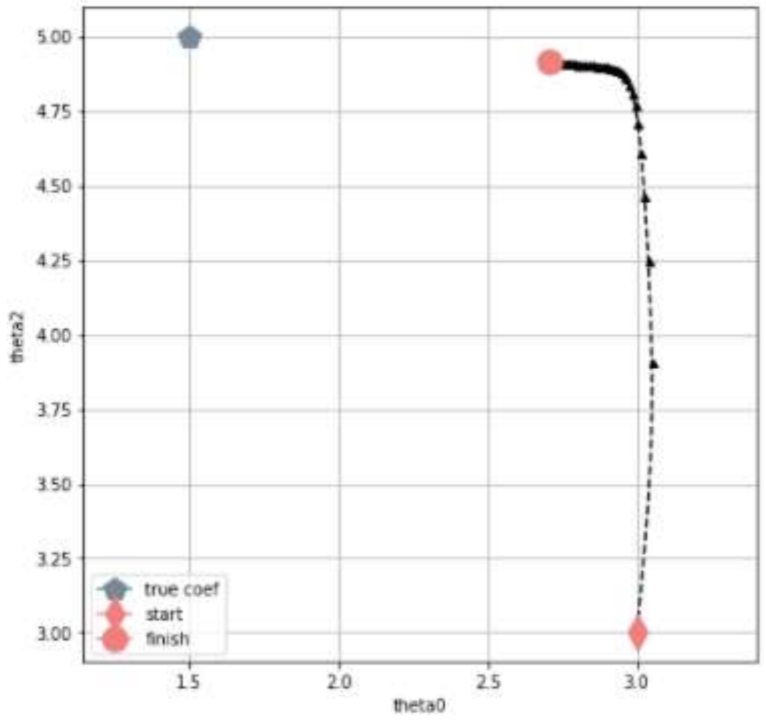
```
num_obs = 100
x1 = np.random.uniform(0,10,num_obs)    # 0-10 均匀分布 100个随机数
x2 = np.random.uniform(0,10,num_obs)    # 变量2
const = np.ones(num_obs)                # 偏置参数的位置（凑成矩阵运算
eps = np.random.normal(0,.5,num_obs)    # 随机量残量，高斯分布，

b = 1.5
theta_1 = 2
theta_2 = 5

y = b*const+ theta_1*x1 + theta_2*x2 + eps   # 两维有噪声线性模型

x_mat = np.array([const,x1,x2]).T # 模型输入转换成nd数组，不含eps
```

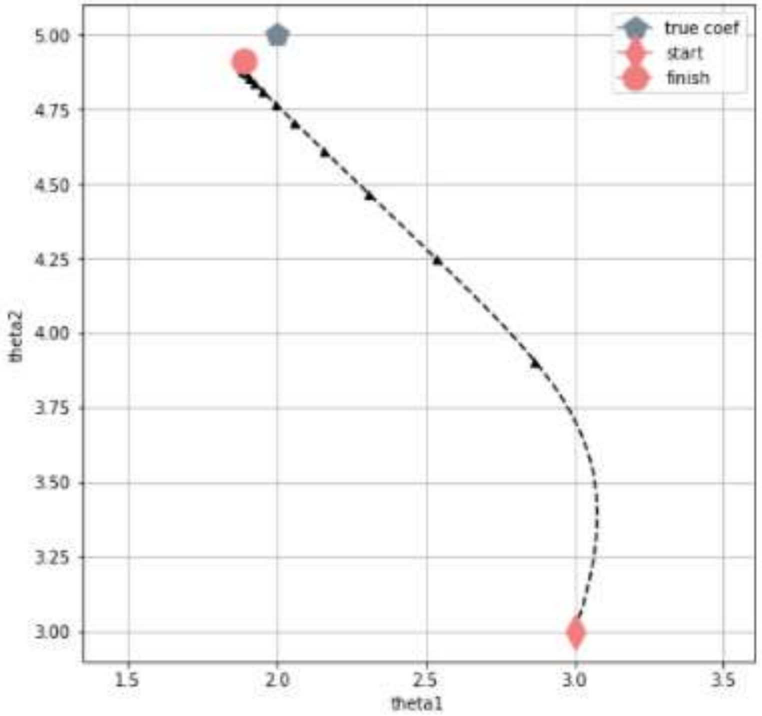
$$Y = b + \theta_1 X_1 + \theta_2 X_2 + \epsilon$$

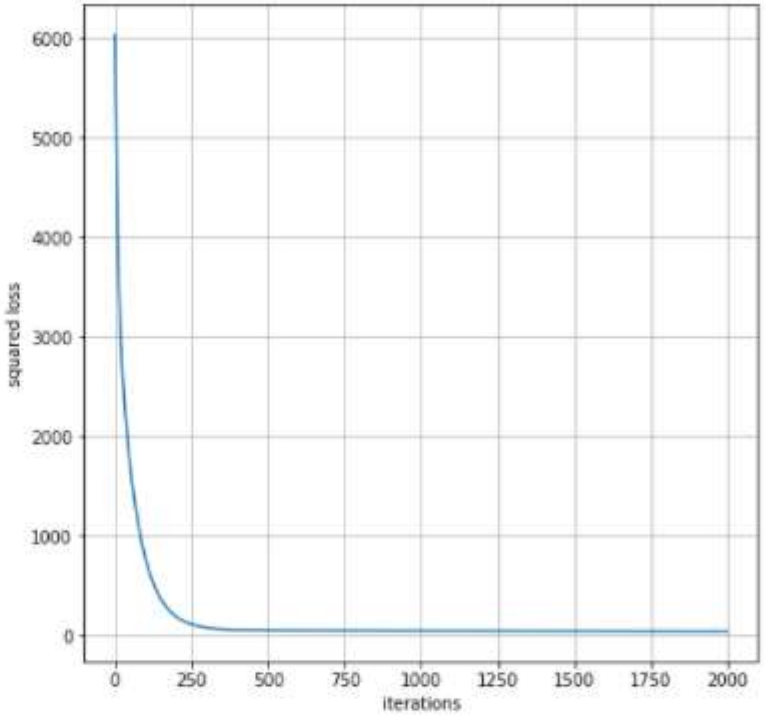
```
learning_rate = 1e-3
num_iter = 2000
theta_initial = np.array([3,3,3])

def gradient_descent(learning_rate, num_iter, theta_initial):

    ## Initialization steps
    theta = theta_initial
    theta_path = np.zeros((num_iter+1,3)) # 两维数组，存放参数轨迹
    theta_path[0,:]= theta_initial          # 第一步

    loss_vec = np.zeros(num_iter)

    ## Main Gradient Descent loop (for a fixed number of iterations)
    for i in range(num_iter):
        y_pred = np.dot(theta.T,x_mat.T)        # 按方程计算y predict, 一个batch
        loss_vec[i] = np.sum((y-y_pred)**2)   # 记录每一轮的loss, 无方向
        grad_vec = (y-y_pred).dot(x_mat) # *2 向量与矩阵的点乘，结果为一个向量
        #print(  grad_vec)
        grad_vec /= num_obs                     # 结果为一个batch的平均梯度,有方向
        theta = theta + learning_rate*grad_vec  # y_pred向y方向拟合
        theta_path[i+1,:]=theta
    return theta_path, loss_vec  #返回参数变化矩阵, loss向量
```

随机梯度下降：

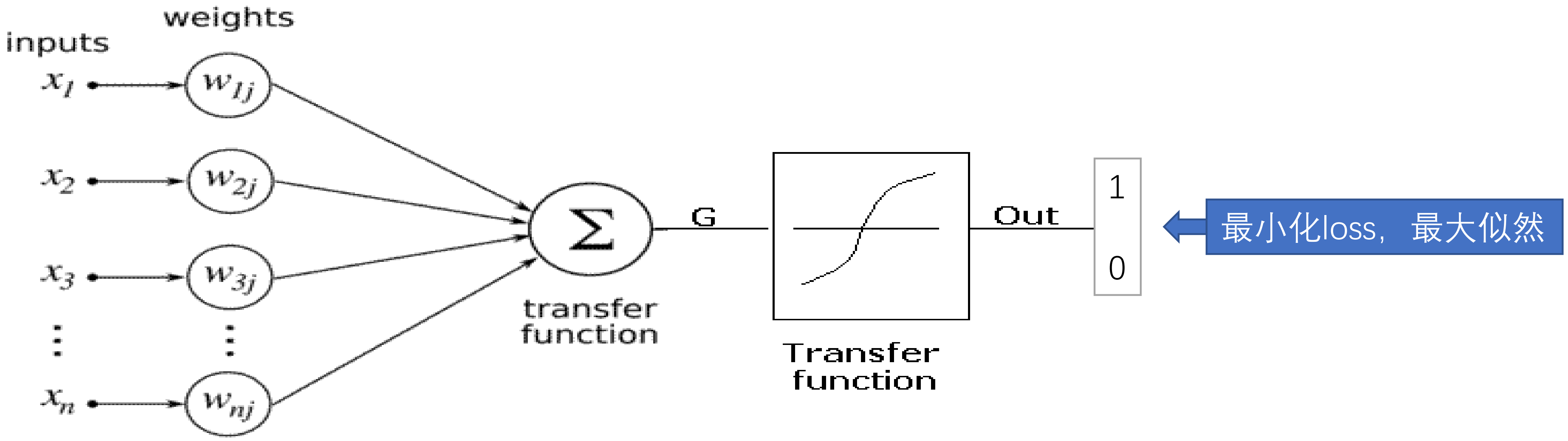
```
grad_vec = (y[j]-y_pred[j])*(x_mat[j,:])  # 对一个样本做梯度回归一次
theta = theta + learning_rate*grad_vec
```







神经元网络模型回归


weights
inputs
x_1
x_2
x_3
x_n
w_{1j}
w_{2j}
w_{3j}
w_{nj}
Σ
transfer function
G
Transfer function
Out
1
0
最小化loss，最大似然

Logistic Regression分类器

对于Logistic Regression($y^{(i)} \in \{0,1\}$表示属于哪一类)，一个样本的似然是：

$$P(y^{(i)}|\mathbf{x}^{(i)},\mathbf{k}) = \begin{cases} \sigma(\mathbf{k}^\top \mathbf{x}) & \text{if } y^{(i)} = 1 \\ 1-\sigma(\mathbf{k}^\top \mathbf{x}) & \text{if } y^{(i)} = 0 \end{cases}$$

$$= \sigma(\mathbf{k}^\top \mathbf{x}^{(i)})^{y^{(i)}} \cdot (1-\sigma(\mathbf{k}^\top \mathbf{x}^{(i)}))^{1-y^{(i)}}$$

最大后验概率

整个数据集的似然则是：

$$\hat{\mathbf{k}} = \arg\max_{\mathbf{k}} \prod_{i=1}^{N} \left\{ \sigma(\mathbf{k}^\top \mathbf{x}^{(i)})^{y^{(i)}} \cdot (1-\sigma(\mathbf{k}^\top \mathbf{x}^{(i)}))^{1-y^{(i)}} \right\}$$

$$= \arg\max_{\mathbf{k}} \sum_{i=1}^{N} \left\{ y^{(i)} \log \sigma(\mathbf{k}^\top \mathbf{x}^{(i)}) + (1-y^{(i)}) \log(1-\sigma(\mathbf{k}^\top \mathbf{x}^{(i)})) \right\}$$

所以我们想要找一个k，最大化上面的这个函数，这就是一个求函数最大值的问题了

Logistic Regression

也就是说，朴素贝叶斯分类器的后验概率是这样一个形式：

$$\sigma(\sum_{i=1}^{K} k_i x_i), \quad (x_0 = 0)$$

事实上，还有很多模型的后验概率也都是这样的形式，所以我们不妨想办法直接求出合适的$\boldsymbol{k_i}$，而不去使用贝叶斯公式。

即是说，我们直接假设：

$$P(y=1|x_1,...,x_K) = \sigma(k_0 + k_1 x_1 + ... + k_K x_k)$$
$$P(y=0|x_1,...,x_K) = 1 - P(y=1|x_1,...,x_K)$$

然后根据我们手里的样本集，估计出k的一个合理的取值。

梯度下降法：计算损失函数的梯度函数：

$$C(k)=\sum_{i=1}^{n}-y^{(i)}\log(g(k^Tx^{(i)}))-(1-y^{(i)})\log(1-g(k^Tx^{(i)}))$$

$$\frac{\partial C(k)}{\partial k_j}=\sum_{i=1}^{n}-y^{(i)}\frac{\frac{\partial g(k^Tx^{(i)})}{\partial k_j}}{g(k^Tx^{(i)})}-(1-y^{(i)})\frac{\frac{\partial(1-g(k^Tx^{(i)}))}{\partial k_j}}{1-g(k^Tx^{(i)})}$$

sigmoid函数这样一个性质：$g'(x)=g(x)(1-g(x))$

$$\frac{\partial C(k)}{\partial k_j}=\sum_{i=1}^{n}\left(-y^{(i)}\frac{g(k^Tx^{(i)})(1-g(k^Tx^{(i)}))x_j^{(i)}}{g(k^Tx^{(i)})}-(1-y^{(i)})\frac{-g(k^Tx^{(i)})(1-g(k^Tx^{(i)}))x_j^{(i)}}{1-g(k^Tx^{(i)})}\right)$$

$$\frac{\partial C(k)}{\partial k_j}=\sum_{i=1}^{n}-y^{(i)}(1-g(k^Tx^{(i)}))x_j^{(i)}-(1-y^{(i)})(0-g(k^Tx^{(i)}))x_j^{(i)}$$

$$\frac{\partial C(k)}{\partial k_j}=\sum_{i=1}^{n}(g(k^Tx^{(i)})-y^{(i)})x_j^{(i)}$$

Logistic Regression训练流程：

输入：样本集；输出：参数k的极大似然估计

1. 随机初始化$\mathbf{k}$
2. 计算梯度$\mathbf{g}$，满足$\mathbf{g}_j = \sum_{i=1}^{N} (y^{(i)} - \sigma(\mathbf{k}^\intercal \mathbf{x}^{(i)}))x_j^{(i)}$
3. $\mathbf{k} = \mathbf{k} + \alpha \mathbf{g}$
4. 迭代上两步

梯度下降

反向梯度

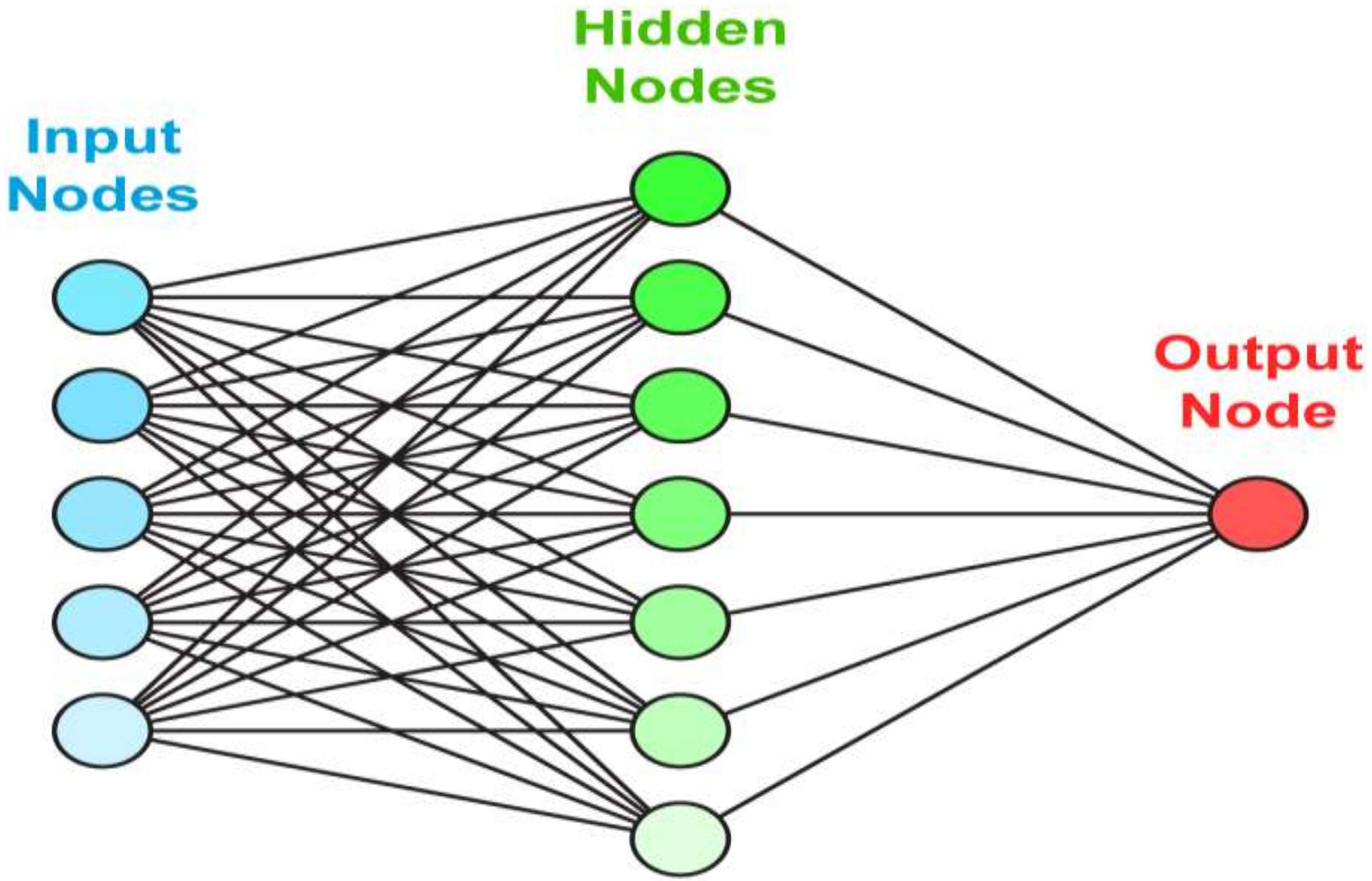
α为学习率

Logistic Regression推断流程：

输入：一个y未知的x；输出：此x的y=1的概率

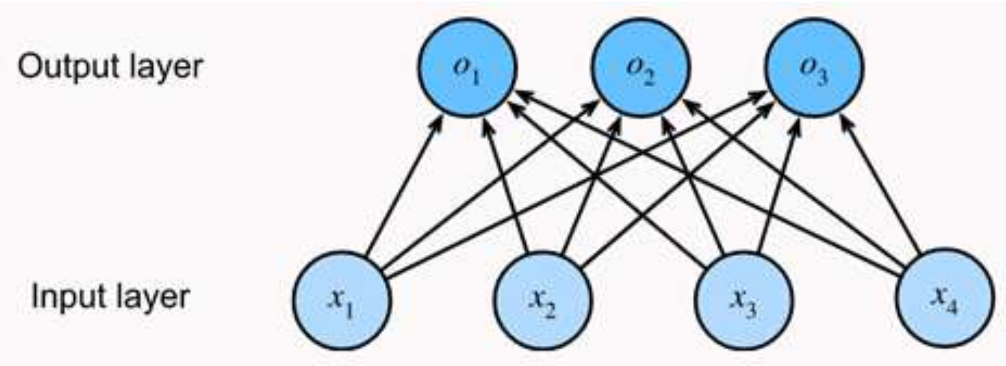
1. 求$P(y=1) = \sigma(\mathbf{k}^\intercal x)$

进一步的改进?



多分类问题: 矩阵降维 + 激活函数 → MSE?

- 多分类问题：$X = \{x_1, x_2, x_3, x_4\} \rightarrow Y = \{'cat','dog','chicken'\}$
  - One-hot encoding：$Y = \{(1,0,0), (0,1,0), (0,0,1)\}$
- 网络结构：
  - $o_1 = x_1 w_{11} + x_2 w_{12} + x_3 w_{13} + x_4 w_{14} + b_1$,
  - $o_2 = x_1 w_{21} + x_2 w_{22} + x_3 w_{23} + x_4 w_{24} + b_2$
  - $o_3 = x_1 w_{31} + x_2 w_{32} + x_3 w_{33} + x_4 w_{34} + b_3$



Softmax推断 与 交叉熵loss

- Softmax运算：

$$\hat{\mathbf{y}} = \mathrm{softmax}(\mathbf{o}) \quad \text{where} \quad \hat{y}_j = \frac{\exp(o_j)}{\sum_k \exp(o_k)}.$$

$$\hat{y}_1 + \hat{y}_2 + \hat{y}_3 = 1 \text{ with } 0 \leq \hat{y}_j \leq 1 \text{ for all } j.$$

- Softmax推断：

$$\operatorname*{argmax}_j \hat{y}_j = \operatorname*{argmax}_j o_j.$$

$$\mathbf{O} = \mathbf{X}\mathbf{W} + \mathbf{b},$$
$$\hat{\mathbf{Y}} = \mathrm{softmax}(\mathbf{O}).$$

Softmax与交叉熵损失函数

- 熵 (Entropy) 与交叉熵 (Cross-Entropy)
  - 熵：$H(P) = \sum_j -P(j)logP(j)$
  - 交叉熵：$H(P,Q) = \sum_j -P(j)logQ(j) - Q(j)logP(j)$

- 交叉熵损失函数:

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

- logistic回归对参数求偏导:

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

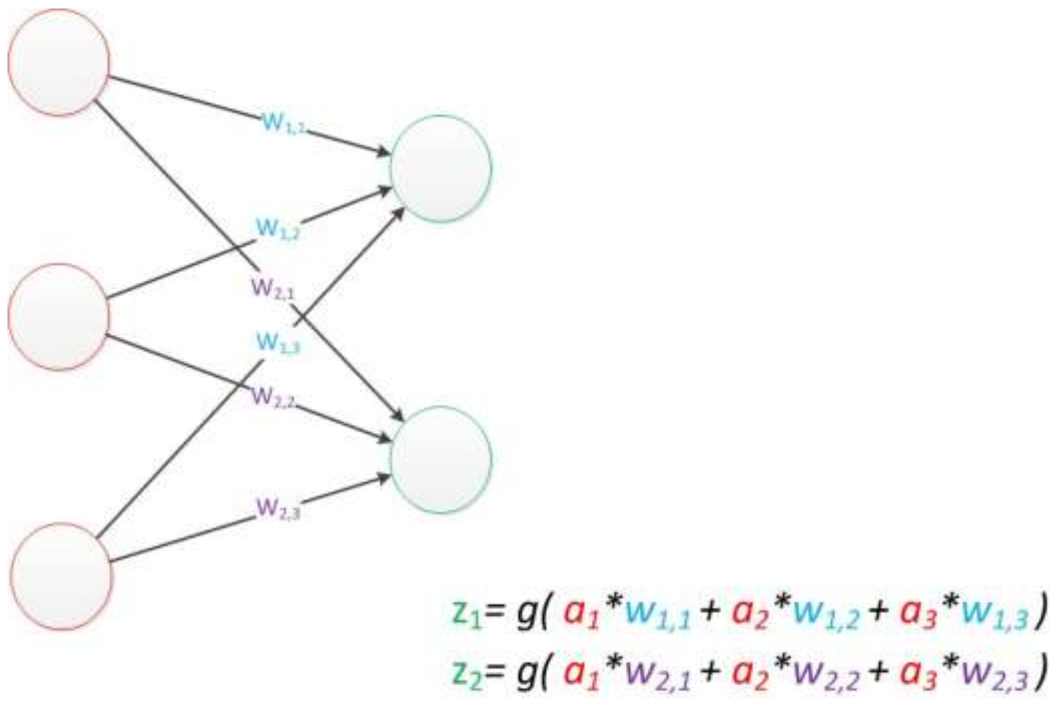
  - 推导：https://blog.csdn.net/jasonzzj/article/details/52017438

神经网络的向前传播机制（predict）

- 输入层输入特征
- 按模型权重与偏置量实现向前传播
- Softmax得到结果分布

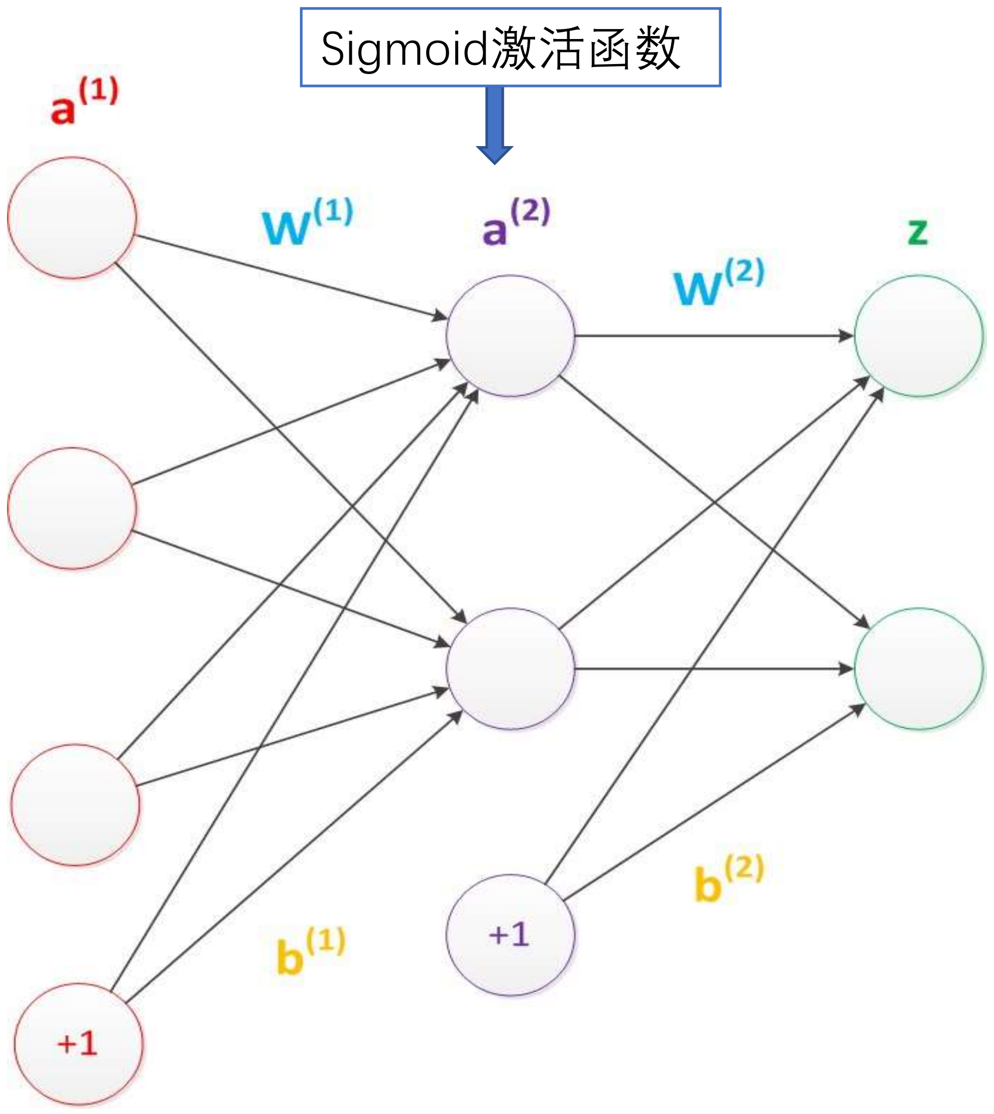
单层前馈网络:

- 输入变量: a= [a_1, a_2, a_3]^T
- 权值: $W = \begin{bmatrix} w_{11}, w_{12}, w_{13} \\ w_{21}, w_{22}, w_{23} \end{bmatrix}$
- 输出: $Z = [z_1, z_2] = g(W * a)$
- 权值是通过训练得到的。



多层神经网络

- $\mathbf{a}^{(1)}$， $\mathbf{a}^{(2)}$， $\mathbf{z}$是网络中传输的向量数据
- $\mathbf{b}^{(1)}$, $\mathbf{b}^{(2)}$, $\mathbf{W}^{(1)}$和$\mathbf{W}^{(2)}$是网络的参数
- $\mathbf{a}^{(2)} = g(\mathbf{W}^{(1)} * \mathbf{a}^{(1)} + \mathbf{b}^{(1)})$;
- $\mathbf{Z} = g(\mathbf{W}^{(2)} * \mathbf{a}^{(2)} + \mathbf{b}^{(2)})$

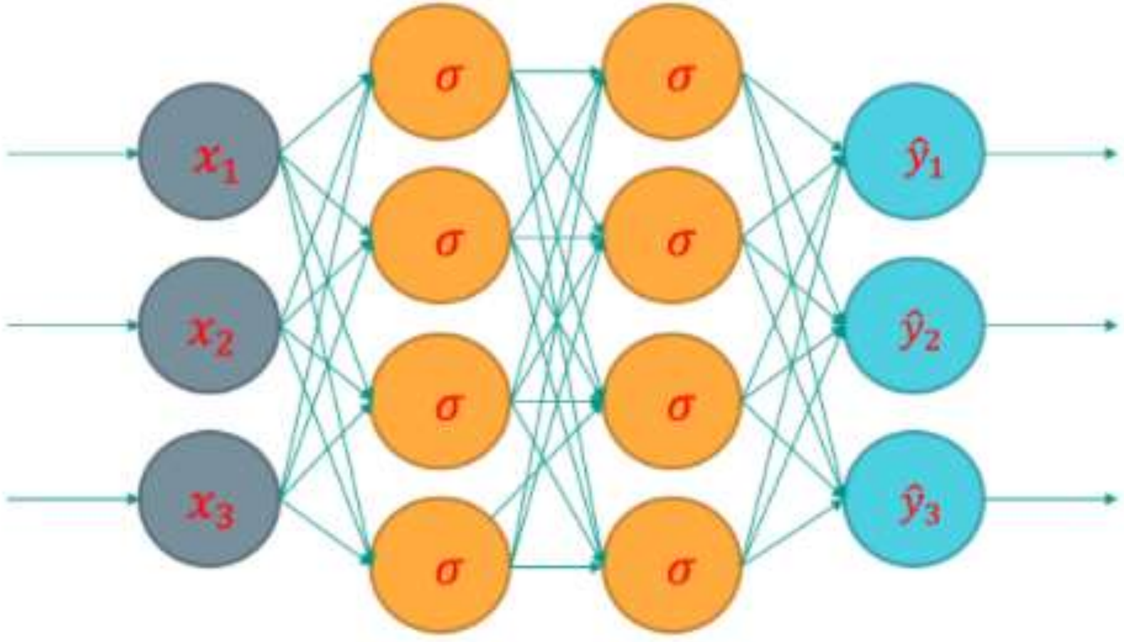


```python
W_1 = np.array([[2,-1,1,4],[-1,2,-3,1],[3,-2,-1,5]])              # 3*4
W_2 = np.array([[3,1,-2,1],[-2,4,1,-4],[-1,-3,2,-5],[3,1,1,1]])   # 4*4
W_3 = np.array([[-1,3,-2],[1,-1,-3],[3,-2,2],[1,2,1]])             # 4*3
x_in = np.array([.5,.8,.2])                               # 单向量输入1*3
x_mat_in = np.array([[.5,.8,.2],[.1,.9,.6],[.2,.2,.3],[.6,.1,.9],[.5,.5,.4],[.9,.1,.9],[.1,.8,.7]])# 7*3 batch in

def soft_max_vec(vec):
    return np.exp(vec)/(np.sum(np.exp(vec))) # 向量softmax

print('the matrix W_1\n')
print(W_1)
print('-'*30)
print('vector input x_in\n')
print(x_in)
print ('-'*30)
```

目前常见为正值

```
the matrix W_1

[[ 2 -1  1  4]
 [-1  2 -3  1]
 [ 3 -2 -1  5]]
------------------------------
vector input x_in

[0.5 0.8 0.2]
------------------------------
```

前馈神经网络（Feedforward Networks)



向前传播的计算流：

```
z_2 = np.dot(x_in,W_1) # x_in 可以认为是a_1 （初始的输入）
z_2                    # 输入与第一层权重网路相乘得到第一层的输出
```

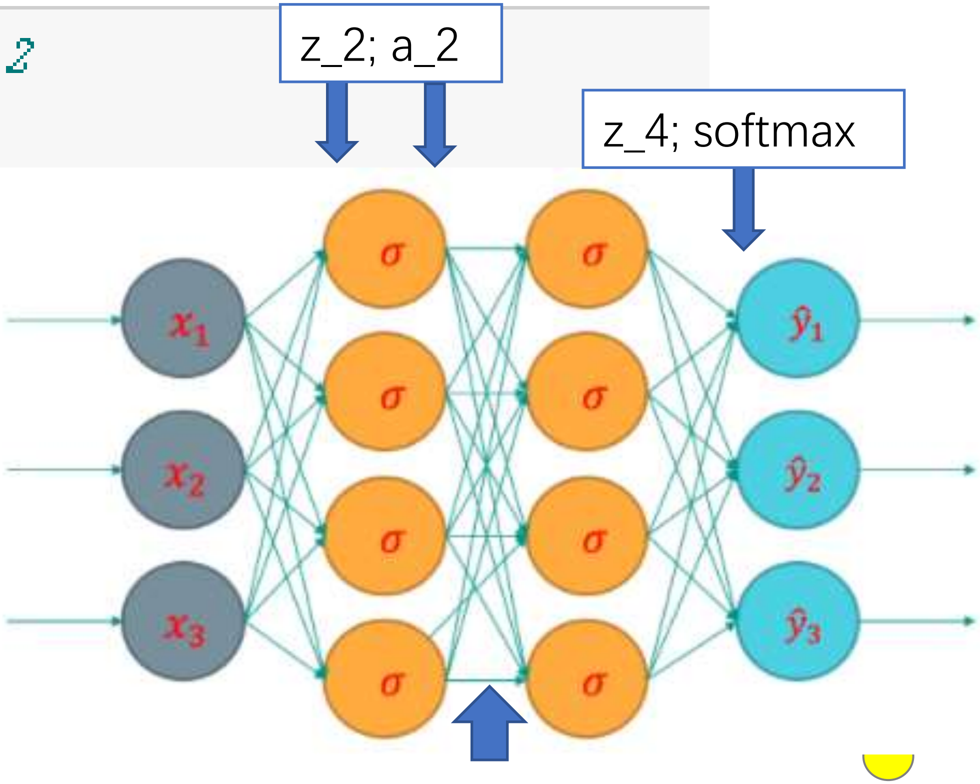
```
array([ 0.8,  0.7, -2.1,  3.8])
```

```
a_2 = sigmoid(z_2) # z_1经过sigmoid输出，得到第二层的输入a_2
a_2
```

```
array([0.68997448, 0.66818777, 0.10909682, 0.97811873])
```

```
z_3 = np.dot(a_2,W_2)  # 第二层
a_3 = sigmoid(z_3)

z_4 = np.dot(a_3,W_3) # 第三层
y_out = soft_max_vec(z_4)
y_out                  # 输出是一个概率分布
```

```
array([0.72780576, 0.26927918, 0.00291506])
```



矩阵数据（batch）进行前馈计算

```
W_1 = np.array([[2,-1,1,4],[-1,2,-3,1],[3,-2,-1,5]])          # 3*4
W_2 = np.array([[3,1,-2,1],[-2,4,1,-4],[-1,-3,2,-5],[3,1,1,1]])  # 4*4
W_3 = np.array([[-1,3,-2],[1,-1,-3],[3,-2,2],[1,2,1]])        # 4*3
x_in = np.array([.5,.8,.2])                          # 单向量输入1*3
x_mat_in = np.array([[.5,.8,.2],[.1,.9,.6],[.2,.2,.3],[.6,.1,.9],[.5,.5,.4],[.9,.1,.9],[.1,.8,.7]])# 7*3 batch in

def soft_max_mat(mat):
    return np.exp(mat)/(np.sum(np.exp(mat),axis=1).reshape(-1,1)) # 矩阵的softmax, 对每个输入向量的输出计算一个分布

print('batch矩阵输入 — starts with the x_mat_in\n')
print(x_mat_in)
```

```
batch矩阵输入 — starts with the x_mat_in

[[0.5 0.8 0.2]
 [0.1 0.9 0.6]
 [0.2 0.2 0.3]
 [0.6 0.1 0.9]
 [0.5 0.5 0.4]
 [0.9 0.1 0.9]
 [0.1 0.8 0.7]]
```

```python
z_2 = np.dot(x_mat_in,W_1)
z_2                     # 7*3 dot 3*4 输出7*4
```

```
array([[ 0.8,  0.7, -2.1,  3.8],
       [ 1.1,  0.5, -3.2,  4.3],
       [ 1.1, -0.4, -0.7,  2.5],
       [ 3.8, -2.2, -0.6,  7. ],
       [ 1.7, -0.3, -1.4,  4.5],
       [ 4.4, -2.5, -0.3,  8.2],
       [ 1.5,  0.1, -3. ,  4.7]])
```

```python
a_2 = sigmoid(z_2) #   return 1.0 / (1.0 + np.exp(-x))

z_3 = np.dot(a_2,W_2)
a_3 = sigmoid(z_3)

z_4 = np.dot(a_3,W_3)
y_out = soft_max_mat(z_4)

y_out  # 每行是一个分布
```

```
array([[0.72780576, 0.26927918, 0.00291506],
       [0.62054212, 0.37682531, 0.00263257],
       [0.69267581, 0.30361576, 0.00370844],
       [0.36618794, 0.63016955, 0.00364252],
       [0.57199769, 0.4251982 , 0.00280411],
       [0.38373781, 0.61163804, 0.00462415],
       [0.52510443, 0.4725011 , 0.00239447]])
```

多层BP神经网络模型学习

- 多层多分类网络与激活函数
  - 反向传播学习

模型学习流程 - 反向梯度传播

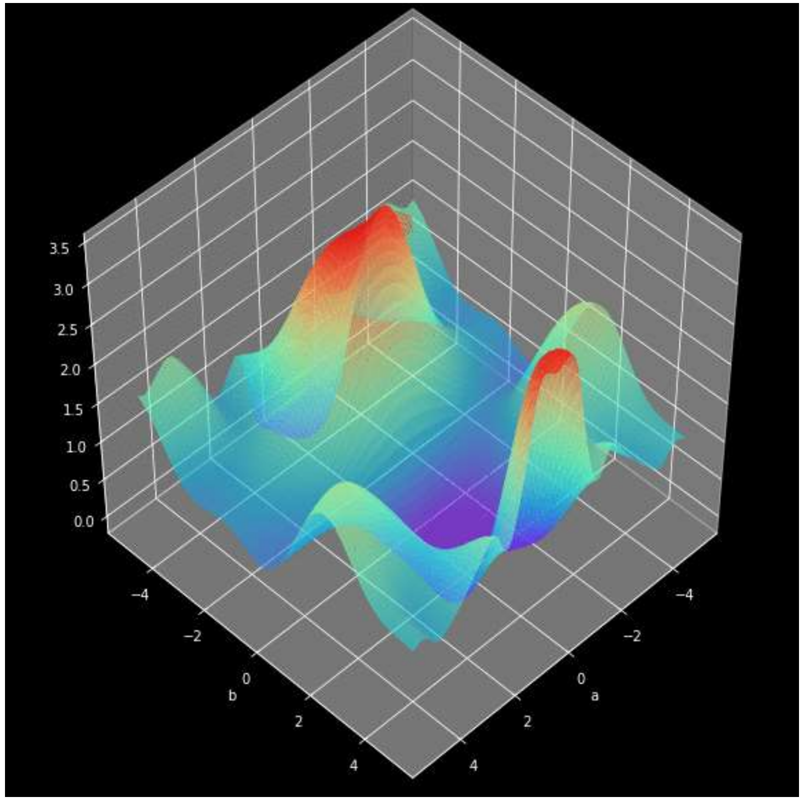
- 模型学习流程：
  - 向前传播：矩阵计算 + 激活函数 + pooling + norm
  - 计算loss
  - 反向梯度回传回归模型参数
    - 链式法则:

$$(g \circ f)'(x) = g'(f(x))\,f'(x),$$

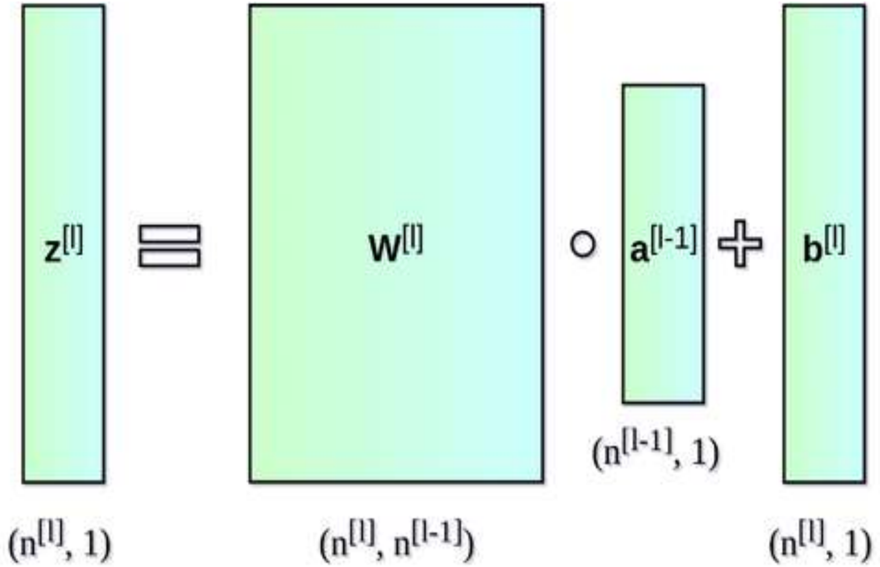
反向梯度传播与梯度下降训练法

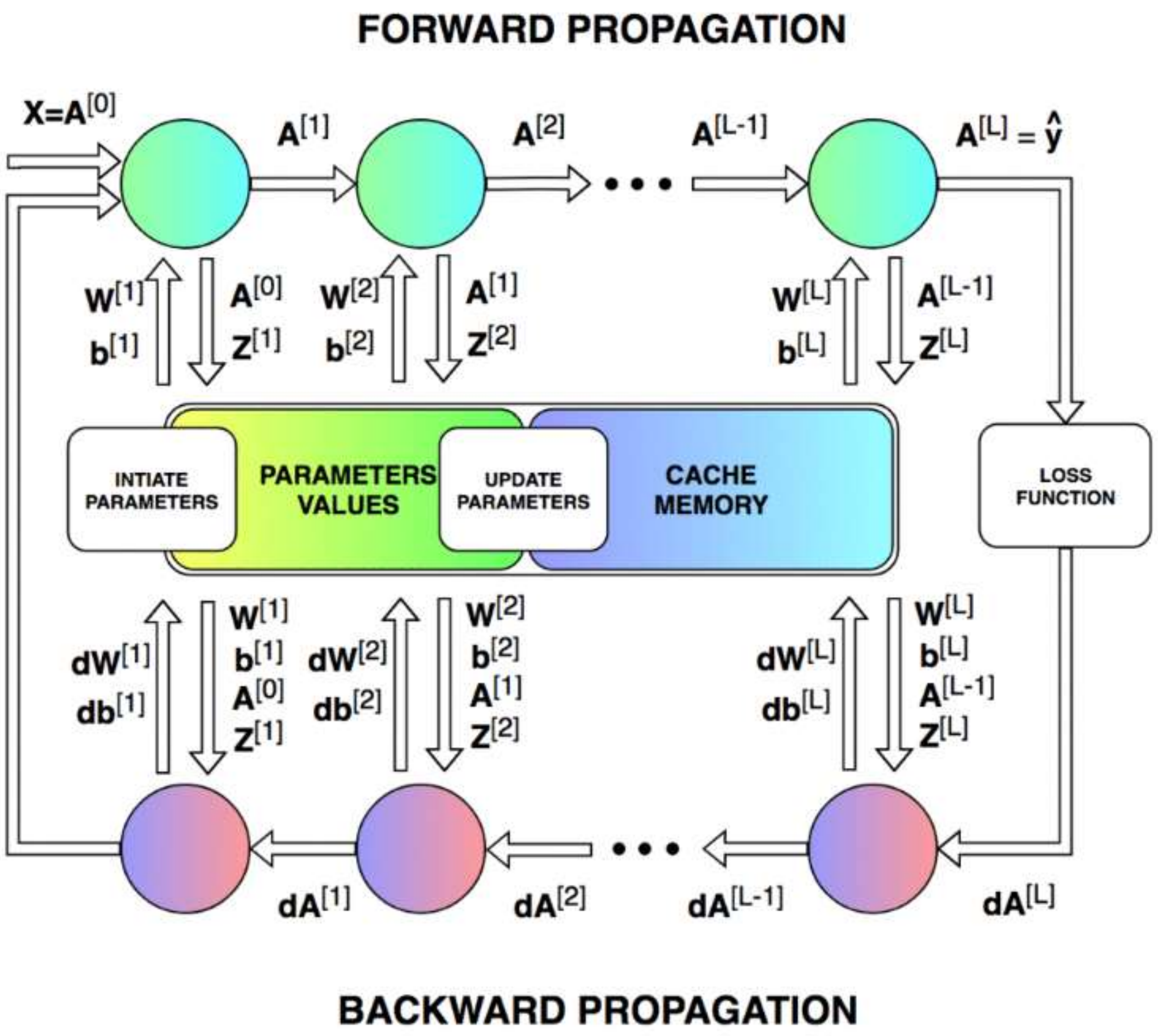
- 假设损失函数为：$MSE = \frac{(y-p)^2}{2}$; $p = g(W * x + b)$.
- 目标：求得一组参数，使得MSE最小。(最优化问题)
- W和b更新算法：
  - 计算W和b的梯度：$\Delta W = \frac{\partial MSE}{\partial W}$，$\Delta b = \frac{\partial MSE}{\partial b}$
  - $W = W - \lambda * \Delta W$
  - $b = b - \lambda * \Delta b$ (λ为学习率)
  - 循环上述过程，直到损失函数足够小。
- 批量梯度：每轮权重更新所有样本都参与训练
  - 随机梯度下降：每轮权重更新只随机选取一个样本参与训练

梯度下降法与局部最优解



训练过程：向前传播-计算loss-反向梯度优化：

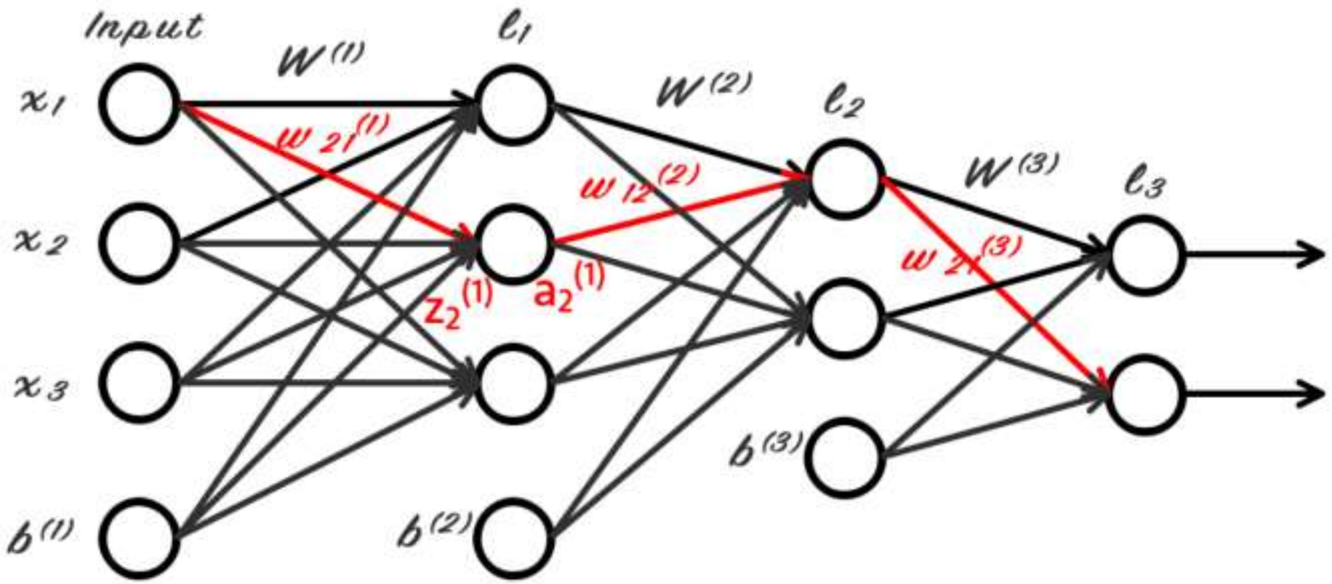


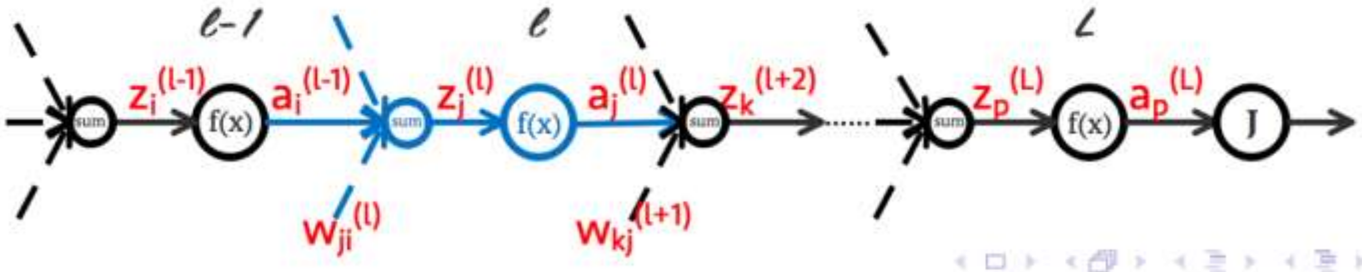


Full forward propagation

向前传播（模型计算）

多层网络计算流程符号体系

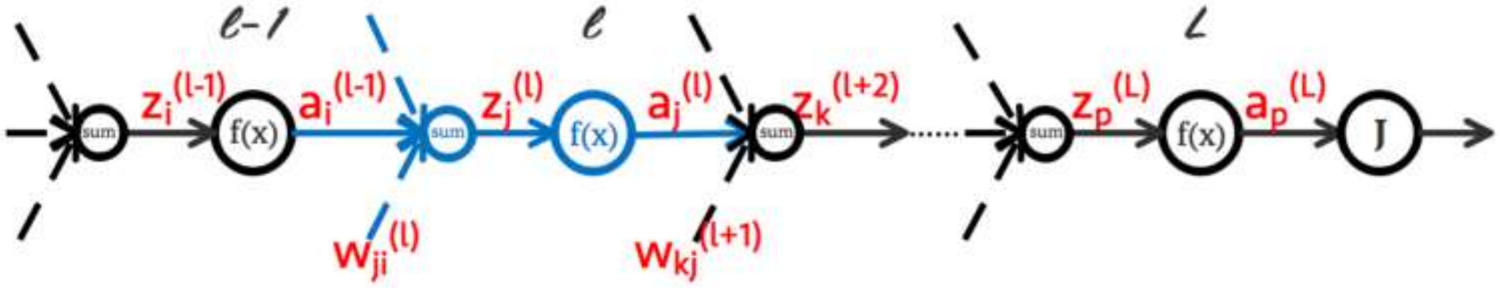




反向梯度传播思想:

- 从后向前计算
- 每层梯度计算都可以看作是一个独立的网络，链式法则
- L-1层梯度的计算与L层的梯度计算有关

反向传播



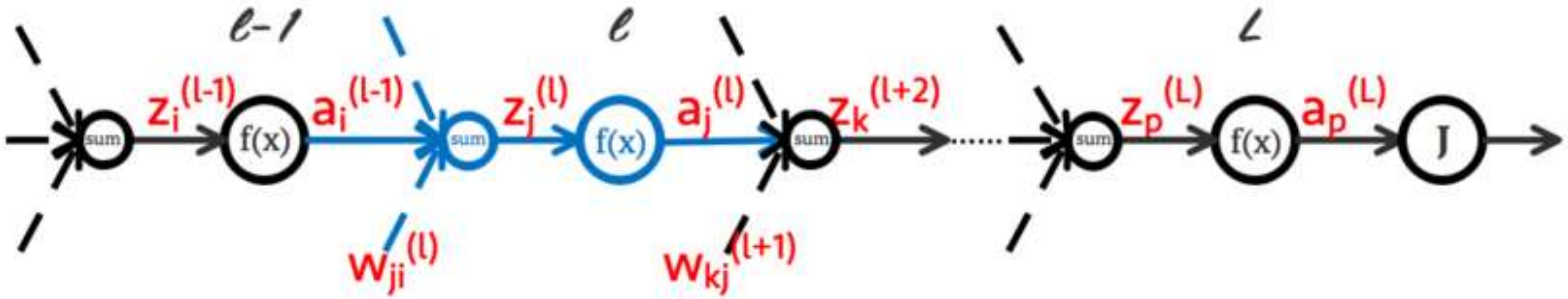
$$z^l = W^{(l)}a^{(l-1)} + b^{(l)} \qquad\qquad a^{(l)} = f(z^{(l)})$$

由梯度下降方法，可知，需要对每个权重权值 $w_{ij}^{(l)}$，求取：

$$w_{ji}^{(l)} = w_{ji}^{(l)} - \alpha\frac{\partial J(W,b)}{\partial w_{ji}^{(l)}} \qquad b_i^{(l)} = b_i^{(l)} - \alpha\frac{\partial J(W,b)}{\partial b_i^{(l)}}$$

其中，关键是如何求取：$\frac{\partial J(W,b)}{\partial w_{ji}^{(l)}}$ 和 $\frac{\partial J(W,b)}{\partial b_i^{(l)}}$

反向传播



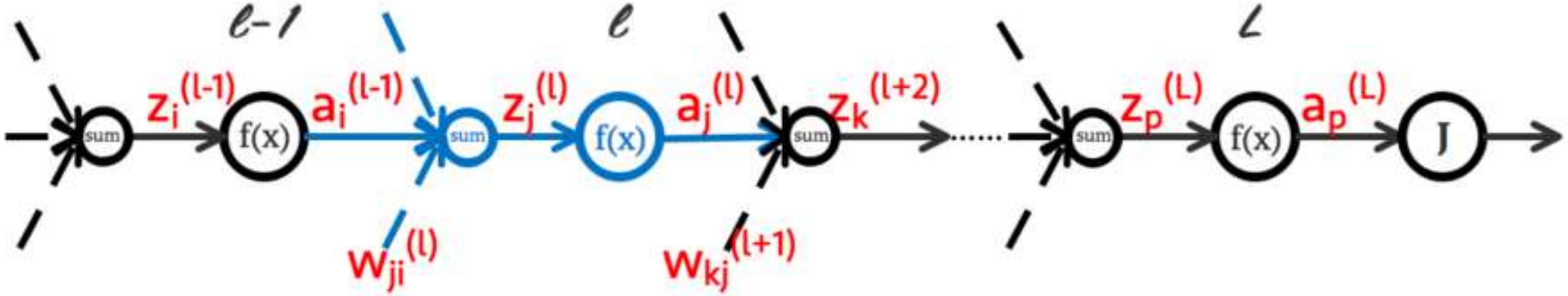
由前向传播过程可知：$z_j^{(l)} = \sum_{i=1}^{n_l} w_{ji}^{(l)} a_i^{(l-1)} + b_i^{(l)}$ 可知：

$$\frac{\partial J(W,b)}{\partial w_{ji}^{(l)}} = \frac{\partial J(W,b)}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial w_{ji}^{(l)}} = \frac{\partial J(W,b)}{\partial z_j^{(l)}} a_i^{(l-1)}$$

$$\frac{\partial J(W,b)}{\partial b_i^{(l)}} = \frac{\partial J(W,b)}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial b_i^{(l)}} = \frac{\partial J(W,b)}{\partial z_j^{(l)}}$$

到此为止，关键是如何求取 $\frac{\partial J(W,b)}{\partial z_j^{(l)}}$

反向传播

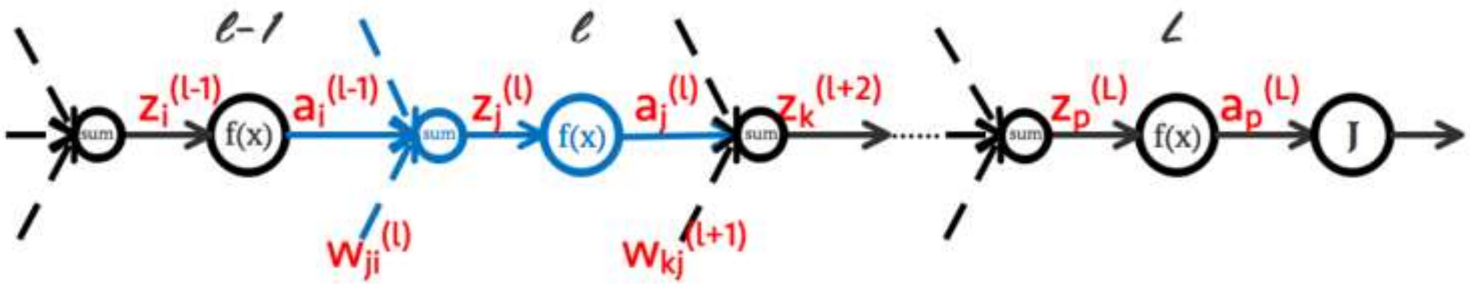


设： $\delta_j^{(l)} = \frac{\partial J(W,b)}{\partial z_j^{(l)}}$

因为： $z_k^{(l+1)} = \sum_{j=1}^{n_{l+1}} w_{kj}^{(l+1)} a_j^{(l)} + b^{(l+1)}$

所以，可以选择从 $z_k^{(l+1)}$ 开始进行对 $z_j^{(l)}$ 进行求导计算：

反向传播推导



$$
\begin{aligned}
\delta_j^{(l)} = \frac{\partial J(W,b)}{\partial z_j^{(l)}} &= \sum_{k=1}^{n_{l+1}} \frac{\partial J(W,b)}{\partial z_k^{(l+1)}} \frac{\partial z_k^{(l+1)}}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} \\
&= \sum_{k=1}^{n_{l+1}} \frac{\partial J(W,b)}{\partial z_k^{(l+1)}} w_{kj}^{(l+1)} f'(z_j^{(l)}) \\
&= \sum_{k=1}^{n_{l+1}} \delta_k^{(l+1)} w_{kj}^{(l+1)} f'(z_j^{(l)})
\end{aligned}
$$

反向传播推导

对于最后一层:

$$\delta_p^{(L)} = \frac{\partial J(W,b)}{\partial z_p^{(L)}} = \frac{\partial J(W,b)}{\partial a_p^{(L)}} * \frac{\partial a_p^{(L)}}{\partial z_p^{(L)}} = \frac{\partial J(W,b)}{\partial a_p^{(L)}} * f'(z_p^{(L)})$$

并且:

$$\frac{\partial J(W,b)}{\partial w_{pq}^{(L)}} = \frac{\partial J(W,b)}{\partial z_p^{(L)}} a_p^{(L-1)} = \delta_p^{(L)} a_p^{(L-1)}$$

$$\frac{\partial J(W,b)}{\partial b_q^{(L)}} = \frac{\partial J(W,b)}{\partial z_p^{(L)}} = \delta_p^{(L)}$$

反向传播推导

小结一下，因为:

$$\frac{\partial J(W,b)}{\partial w_{ji}^{(l)}} = \frac{\partial J(W,b)}{\partial z_j^{(l)}} a_i^{(l-1)} \qquad \frac{\partial J(W,b)}{\partial b_i^{(l)}} = \frac{\partial J(W,b)}{\partial z_j^{(l)}}$$

又因为 (上文推导结果):

$$\frac{\partial J(W,b)}{\partial z_j^{(l)}} = \delta_j^{(l)} = \sum_{k=1}^{n_{l+1}} \delta_k^{(l+1)} w_{kj}^{(l+1)} f'(z_j^{(l)})$$

从而得到:

$$\frac{\partial J(W,b)}{\partial w_{ji}^{(l)}} = \delta_j^{(l)} a_i^{(l-1)} \qquad \frac{\partial J(W,b)}{\partial b_i^{(l)}} = \delta_j^{(l)}$$

反向传播总结

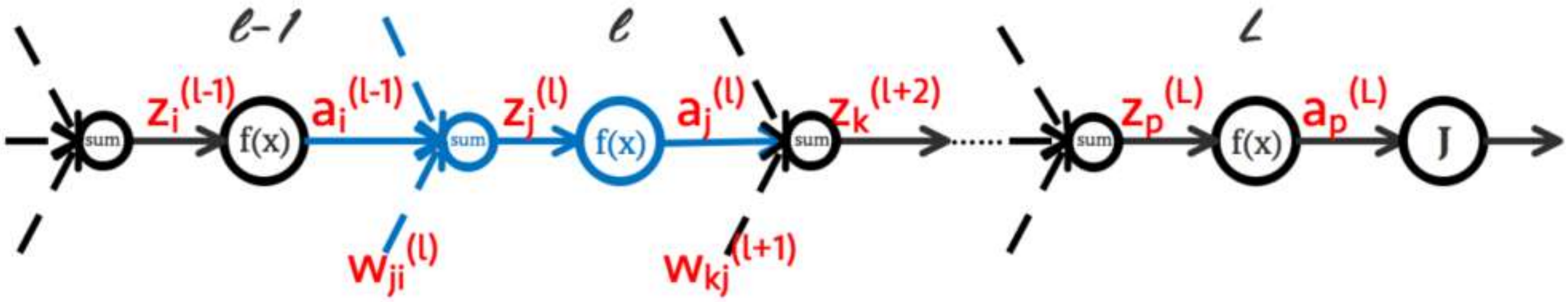
总结一下：

$$\frac{\partial J(W,b)}{\partial w_{ji}^{(l)}} = \delta_j^{(l)} a_i^{(l-1)} = \left(\sum_{k=1}^{n_{l+1}} \delta_k^{(l+1)} w_{kj}^{(l+1)} f'(z_j^{(l)}) \right) a_i^{(l-1)}$$

$$\frac{\partial J(W,b)}{\partial b_i^{(l)}} = \delta_j^{(l)} = \sum_{k=1}^{n_{l+1}} \delta_k^{(l+1)} w_{kj}^{(l+1)} f'(z_j^{(l)})$$

反向传播计算流程

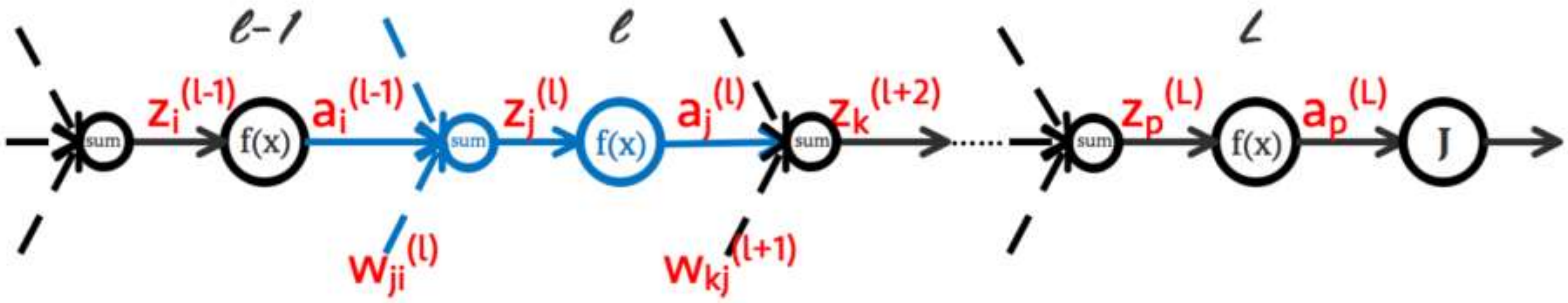
Step-1：依据前向传播算法求解每一层的激活值：



$$z^l = W^{(l)}a^{(l-1)} + b^{(l)} \qquad a^{(l)} = f(z^{(l)})$$

反向传播计算流程

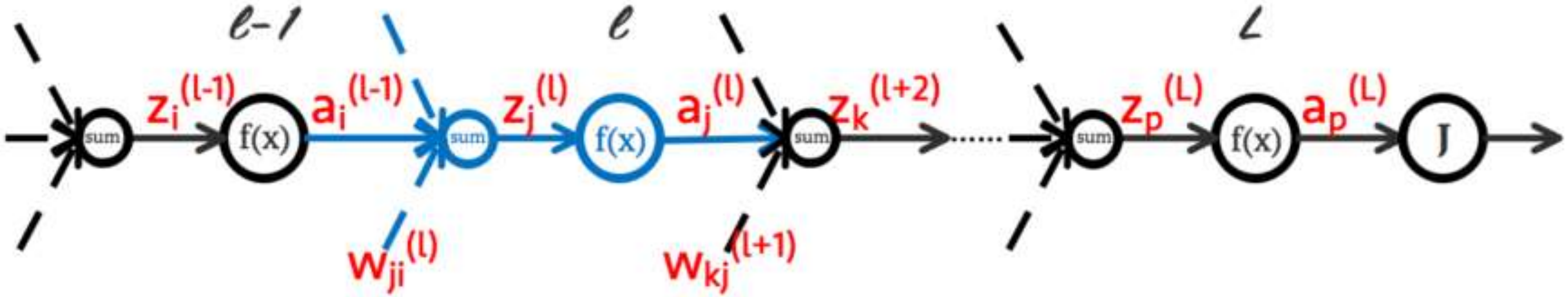
Step-2：计算出最后一层 (L 层) 的每个神经元的 $\delta_p^{(L)}$：



$$\delta_p^{(L)} = \frac{\partial J(W,b)}{\partial a_p^{(L)}} * f'(z_p^{(L)})$$

反向传播计算流程

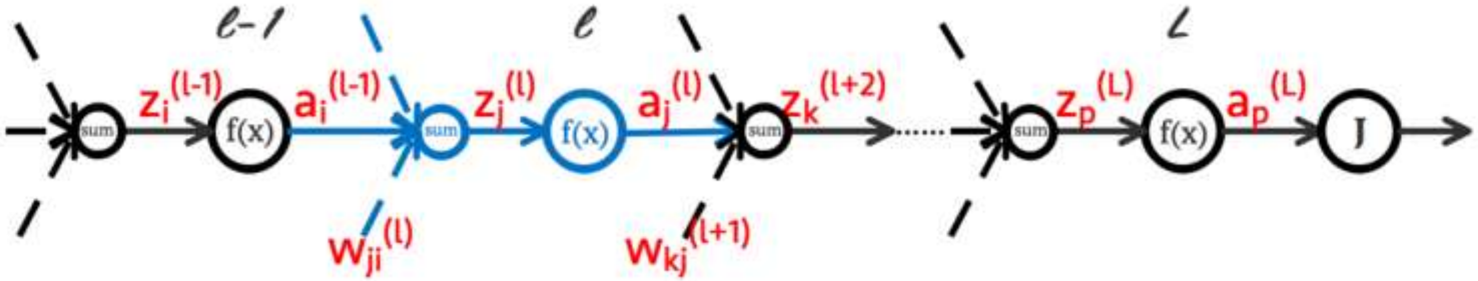
Step-3：由后向前，依次计算出各层（l 层）各个神经元的 $\delta_j^{(l)}$



$$\delta_j^{(l)} = \sum_{k=1}^{n_{l+1}} \delta_k^{(l+1)} w_{kj}^{(l+1)} f'(z_j^{(l)})$$

反向传播计算流程

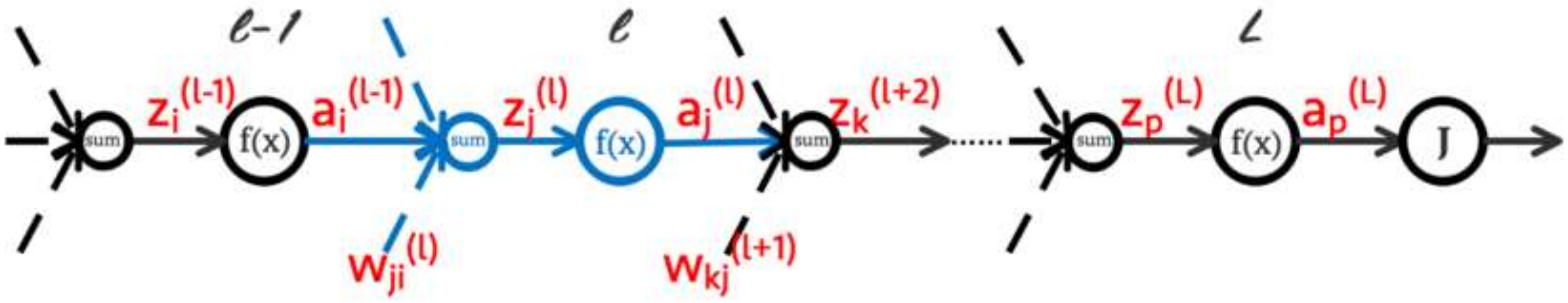
Step-4：计算出各层（l 层）的各个权重（$w_{ji}^{(l)}$）的梯度 $\frac{\partial J(W,b)}{\partial w_{ji}^{(l)}}$ 及各个偏置（$b_i^{(l)}$）的梯度 $\frac{\partial J(W,b)}{\partial b_i^{(l)}}$：



$$\frac{\partial J(W,b)}{\partial w_{ji}^{(l)}} = \delta_j^{(l)} a_i^{(l-1)} \qquad \frac{\partial J(W,b)}{\partial b_i^{(l)}} = \delta_j^{(l)}$$

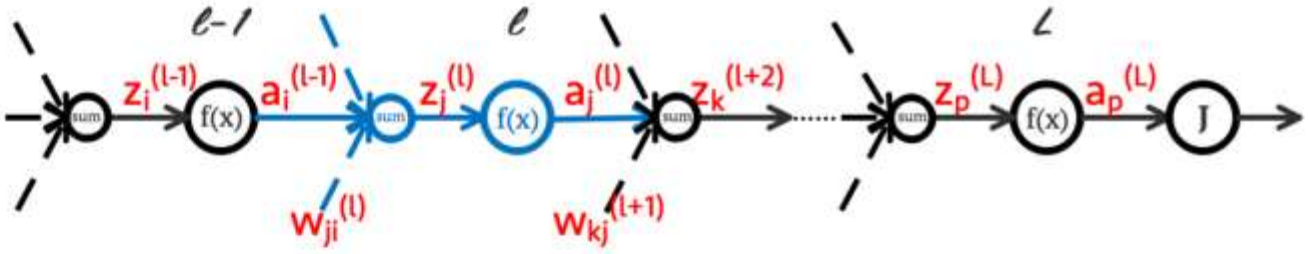
反向传播计算流程

Step-5：对各层（l 层）的各个权重（$w_{ji}^{(l)}$）及各个偏置（$b_i^{(l)}$）进行更新，直到代价函数 $J(W,b)$ 足够小：



$$w_{ji}^{(l)} = w_{ji}^{(l)} - \alpha \frac{\partial J(W,b)}{\partial w_{ji}^{(l)}} \qquad b_i^{(l)} = b_i^{(l)} - \alpha \frac{\partial J(W,b)}{\partial b_i^{(l)}}$$

反向传播梯度下降权重修正计算公式



$$
\begin{aligned}
w_{ji}^{(l)} &= w_{ji}^{(l)} - \alpha \frac{\partial J(W,b)}{\partial w_{ji}^{(l)}} \\
&= w_{ji}^{(l)} - \alpha \frac{\partial J(W,b)}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial w_{ji}^{(l)}} = w_{ji}^{(l)} - \alpha \frac{\partial J(W,b)}{\partial z_j^{(l)}} a_i^{(l-1)} \\
&= w_{ji}^{(l)} - \alpha \delta_j^{(l)} a_i^{(l-1)} \\
&= w_{ji}^{(l)} - \alpha \left(\sum_{k=1}^{n_{l+1}} \delta_k^{(l+1)} w_{kj}^{(l+1)} f'(z_j^{(l)}) \right) a_i^{(l-1)}
\end{aligned}
$$

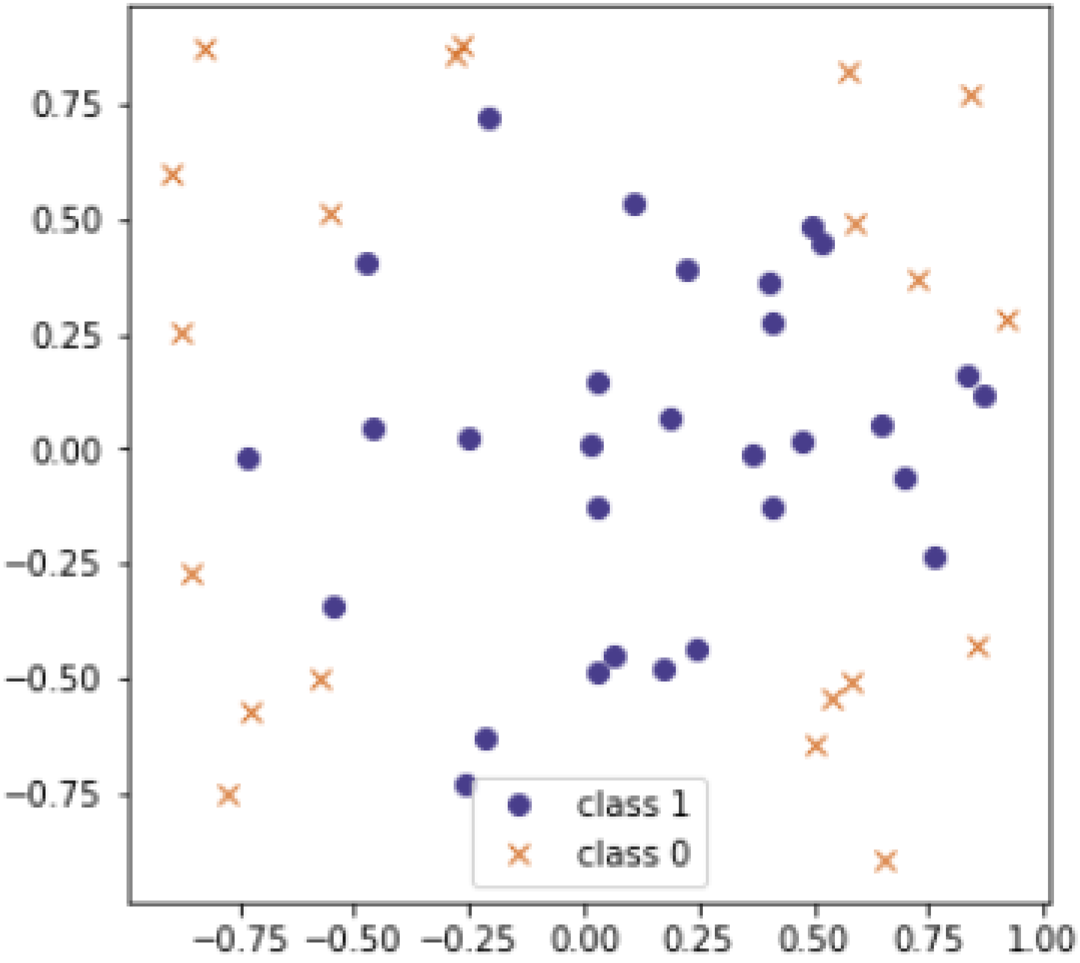
- 总结：想求第n层节点j的zj输入对loss的偏导，如果第n+1层的所有zj对loss的偏导是知道的，就可以推算出来（激活函数对zj导函数是确定的，只有一个输入，一个输出，因此不是偏导数，直接能套公式算）。
- 最后一级无论是sigmoid或softmax都是可以直接求导的，因此就是先向前传播，到了最后一级，sigmoid或softmax一下，计算loss，然后从最后一级开始从后向前依次计算网络边权对loss的倒数，然后对边权进行反向梯度修正。

例子： 数据生成

```
# 生成带标签的2维特征空间的样本集
num_obs = 50 #500
x_mat_1 = np.random.uniform(-1,1,size = (num_obs,2))# -1, 1 均匀分布 2维随机向量
x_mat_bias = np.ones((num_obs,1))                    # 偏置位 a1*x1 + a2*x2 + c
x_mat_full = np.concatenate( (x_mat_1,x_mat_bias), axis=1)  # 3维数据

# x,y坐标值的abs中较大的<.5, 是原点.5的正方形区域, 返回值ture, 2int: 1, 其他 0
# y = ((np.maximum(np.abs(x_mat_full[:,0]), np.abs(x_mat_full[:,1])))<.5).astype(int)

# x,y坐标值的abs之<1, 是原点距离1的菱形区域, 返回值ture, 2int: 1, 其他 0
y = ((np.abs(x_mat_full[:,0]) + np.abs(x_mat_full[:,1]))<1).astype(int)
print(y)
```

```
[1 1 1 1 1 1 1 0 1 1 1 0 1 1 1 0 1 0 0 0 1 0 0 1 0 0 0 0 0 1 0 0 0 1 1 1 1
 1 1 0 1 1 0 1 1 0 1 1 1 0]
```

```python
fig, ax = plt.subplots(figsize=(5, 5))
ax.plot(x_mat_full[y==1, 0],x_mat_full[y==1, 1], 'ro', label='class 1', color='darkslateblue')
ax.plot(x_mat_full[y==0, 0],x_mat_full[y==0, 1], 'bx', label='class 0', color='chocolate')
# ax.grid(True)
ax.legend(loc='best')
ax.axis('equal');
```



网络模型：2层网络，sigmoid激活函数

$$\frac{\partial J}{\partial W^{(3)}} = (\hat{y} - y) \cdot a^{(3)}$$

激活函数求导

$$\frac{\partial J}{\partial W^{(2)}} = (\hat{y} - y) \cdot W^{(3)} \cdot \sigma'\left(z^{(3)}\right) \cdot a^{(2)}$$

sigmoid激活函数求导

$$\frac{\partial J}{\partial W^{(1)}} = (\hat{y} - y) \cdot W^{(3)} \cdot \sigma'\left(z^{(3)}\right) \cdot W^{(2)} \cdot \sigma'\left(z^{(2)}\right) \cdot X$$

$$\sigma'(z) = \sigma(z)(1 - \sigma(z))$$

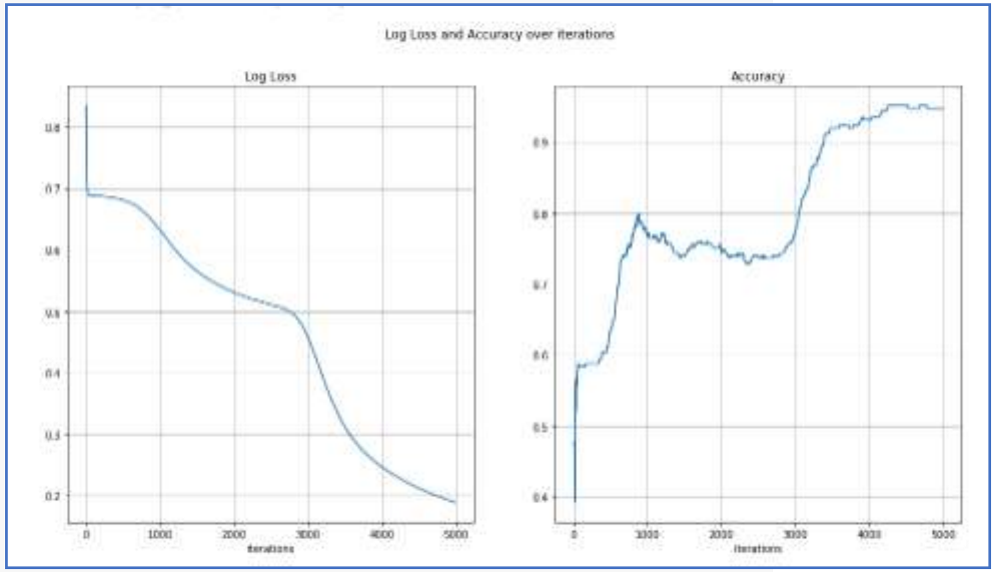
$$\boxed{f'(x) = \frac{e^{-x}}{(1+e^{-x})^2} = \frac{(1+e^{-x})-1}{(1+e^{-x})^2} = \frac{1}{1+e^{-x}} - \left(\frac{1}{1+e^{-x}}\right)^2 = f(x) - f(x)^2}$$

定义向前传播及反向计算梯度函数

```
# 向前传播函数，输入网络参数矩阵3*4, 4*1 ，输出预测结果以及loss
def forward_pass(W1, W2):

    global x_mat  # 输入样本参数矩阵 50*3
    global y      # 目标（向量）     50*1
    global num_
    # First, compute the new predictions `y_pred`
    z_2 = np.dot(x_mat, W_1)   # 输入经过第一层网络计算到z_2: 50*4
    a_2 = sigmoid(z_2)
    z_3 = np.dot(a_2, W_2)     # a_2到pred结果层z_3: 50*1
    y_pred = sigmoid(z_3).reshape((len(x_mat),)) # 经过激活函数得到预测结果向量

    # 开始反向梯度传播
    J_z_3_grad = -y + y_pred
    J_W_2_grad = np.dot(J_z_3_grad, a_2) # 得到W_2的梯度
    a_2_z_2_grad = sigmoid(z_2)*(1-sigmoid(z_2))
    J_W_1_grad = (np.dot((J_z_3_grad).reshape(-1,1), W_2.reshape(-1,1).T)*a_2_z_2_grad).T.dot(x_mat).T

    gradient = (J_W_1_grad, J_W_2_grad) # 记录多层的梯度

    return y_pred, gradient  #返回本轮预测值以及本轮回归的梯度值
```

模型初始化及训练

```python
W_1 = np.random.uniform(-1,1,size=(3,4))  # 3*4 网络
W_2 = np.random.uniform(-1,1,size=(4))    # 4*1 输出单分类结果
num_iter = 5000
learning_rate = .001
x_mat = x_mat_full    # 输入样本矩阵


loss_vals, accuracies = [], []
for i in range(num_iter):
    ## 向前传播得到预测值、梯度
    y_pred, (J_W_1_grad, J_W_2_grad) = forward_pass(W_1, W_2)

    ## 按反向梯度值调整模型参数
    W_1 = W_1 - learning_rate*J_W_1_grad
    W_2 = W_2 - learning_rate*J_W_2_grad

    ### Compute the loss and accuracy
    curr_loss = loss_fn(y,y_pred)
    loss_vals.append(curr_loss)
    acc = np.sum((y_pred>=.5) == y)/num_obs
    accuracies.append(acc)

    ## Print the loss and accuracy for every 200th iteration
    if((i%200) == 0):
        print('iteration {}, log loss is {:.4f}, accuracy is {}'.format(
            i, curr_loss, acc
        ))
plot_loss_accuracy(loss_vals, accuracies)
```



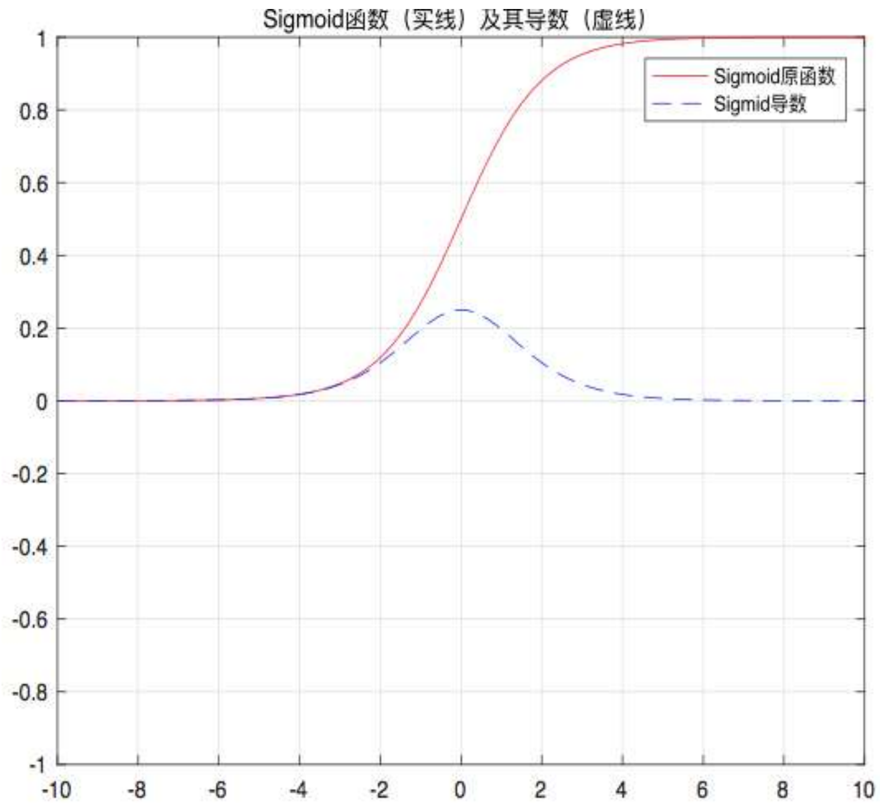

```
iteration 0, log loss is 0.8357, accuracy is 0.476
iteration 200, log loss is 0.6879, accuracy is 0.588
```

梯度消失与激活函数选择

- 模型拟合与冗余
- 梯度消失问题
- 常见的激活函数

Sigmoid

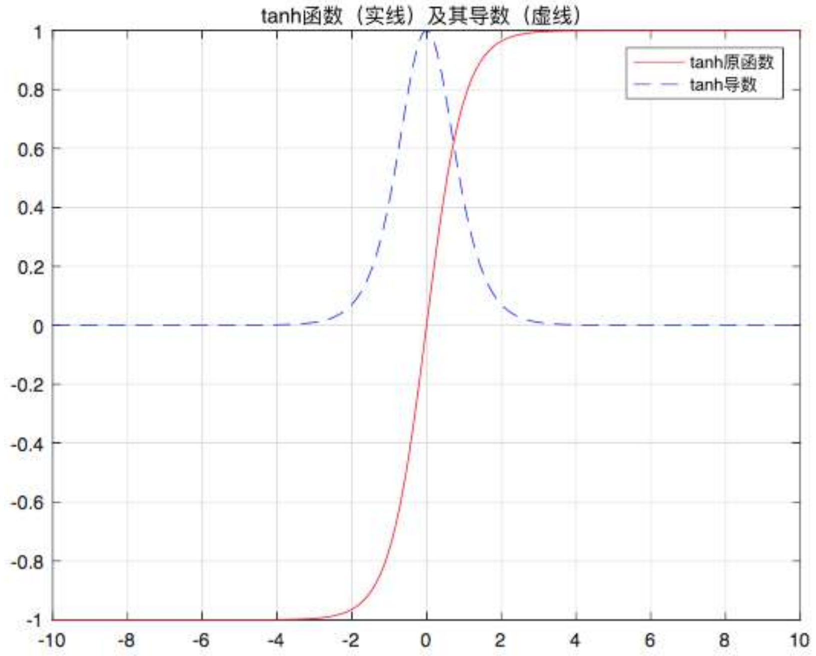
- $f(x) = \frac{1}{1+e^{-x}}$



- $f'(x) = \frac{e^{-x}}{(1+e^{-x})^2} = \frac{(1+e^{-x})-1}{(1+e^{-x})^2} = \frac{1}{1+e^{-x}} - \left(\frac{1}{1+e^{-x}}\right)^2 = f(x) - f(x)^{\ 2}$

Tanh

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



- $f'(x) = \frac{(e^x + e^{-x})^2 - (e^x - e^{-x})^2}{(e^x + e^{-x})^2} = 1 - f(x)^2$

Relu

- $f(x) = \begin{cases} 0; x < 0 \\ x; x \geq 0 \end{cases}$
- $f'(x) = \begin{cases} 0; x < 0 \\ 1; x \geq 0 \end{cases}$