神经网络

- 概述
- 神经元模型
- 神经网络结构
- 代价函数
- 梯度下降
- 反向传播算法
- 使用神经网络识别手写数字
- 改进：交叉熵代价函数
- 改进：过度拟合和规范化

概述

神经网络是机器学习众多算法中的一类。首先，我们来聊聊“什么是机器学习”。

$f($  $)=$ “How are you”

语言识别

$f($  $)=$ “Cat”

图像识别

$f($  $)=$ “5-5”

围棋

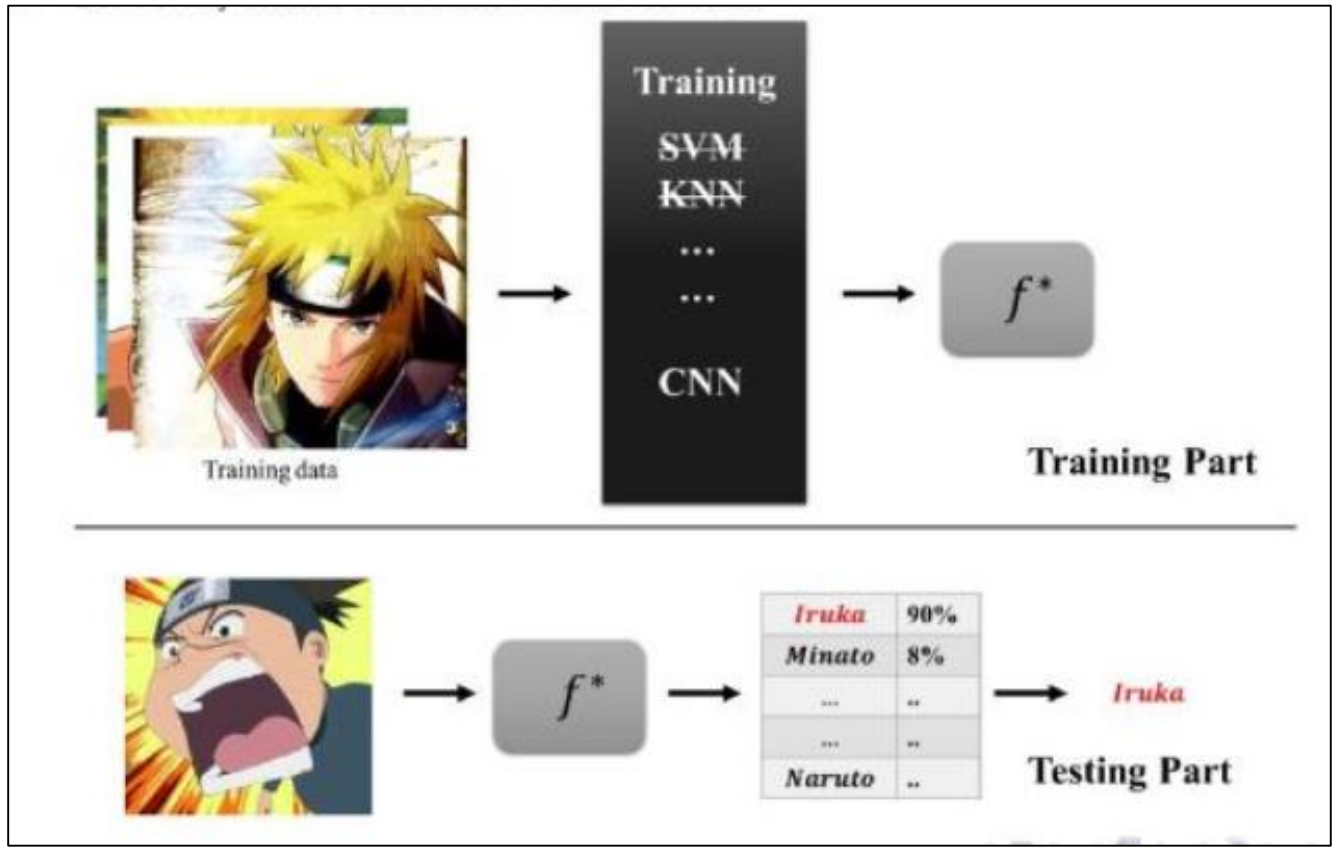
$f($ “Hi” $)=$ “Hello”

对话系统

什么是机器学习，从一个不精确的角度来说，它就如同寻找一个合适的函数。

概述

图像识别



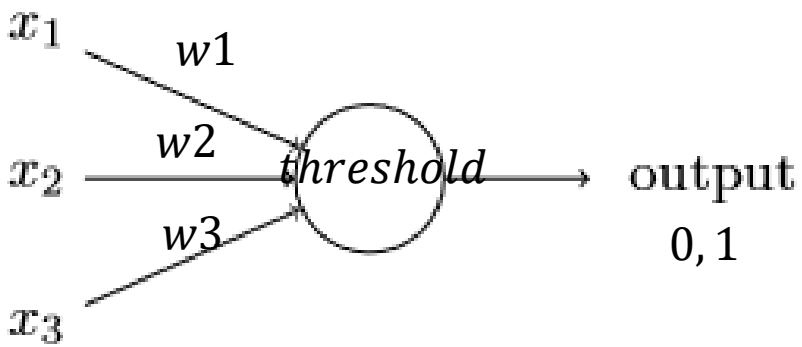
- 我们能够使用很多的方法来训练我们的模型。
- 但是，今天我们只讨论神经网络。

概述

- 神经网络是由具有适应性的简单单元组成的广泛并行的网络，它的组织能够模拟生物神经系统对真实世界物体所作出的交互反应。—— Kohonen, 1998
- 神经网络逐渐兴起于二十世纪八九十年代，应用得非常广泛。但由于各种原因，在 90年代的后期应用减少了。但是最近，神经网络又东山再起了。其中一个原因是：神经网络是计算量有些偏大的算法。然而大概由于近些年计算机的运行速度变快，才足以真正运行起大规模的神经网络。

神经元模型

感知器：接收二进制输入，并产生一个二进制输出。

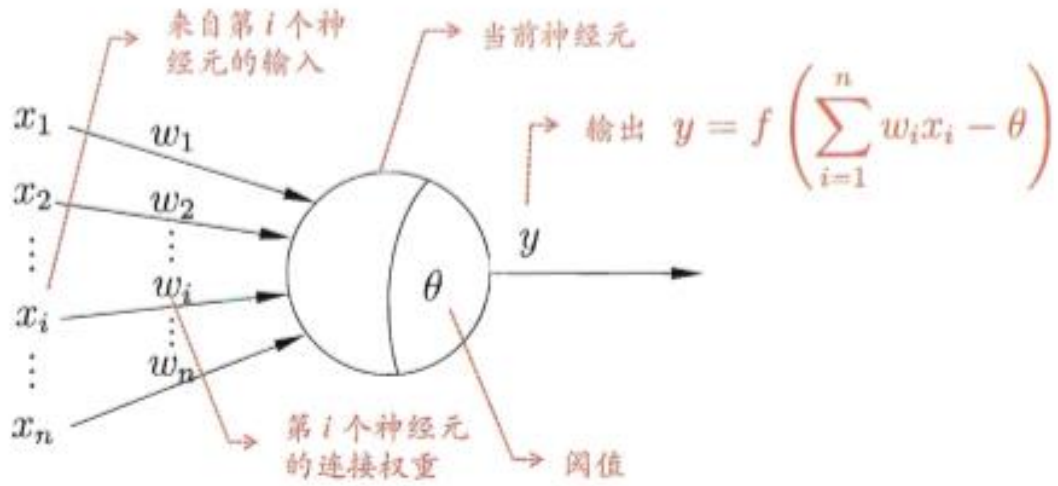


$$output = \begin{cases} 0, if \sum_i w_i x_i - threshold \leq 0 \\ 1, if \sum_i w_i x_i - threshold > 0 \end{cases}$$

简化：将$\sum_{i=1}^{n} w_i x_i$改写成点乘，$w \cdot x = \sum_{i=1}^{n} w_i x_i$,这里w和x对应权重和输入的向量。偏置$b = -threshold$。

$$output = \begin{cases} 0, if\ w \cdot x + b \leq 0 \\ 1, if w \cdot x + b > 0 \end{cases}$$

神经元模型



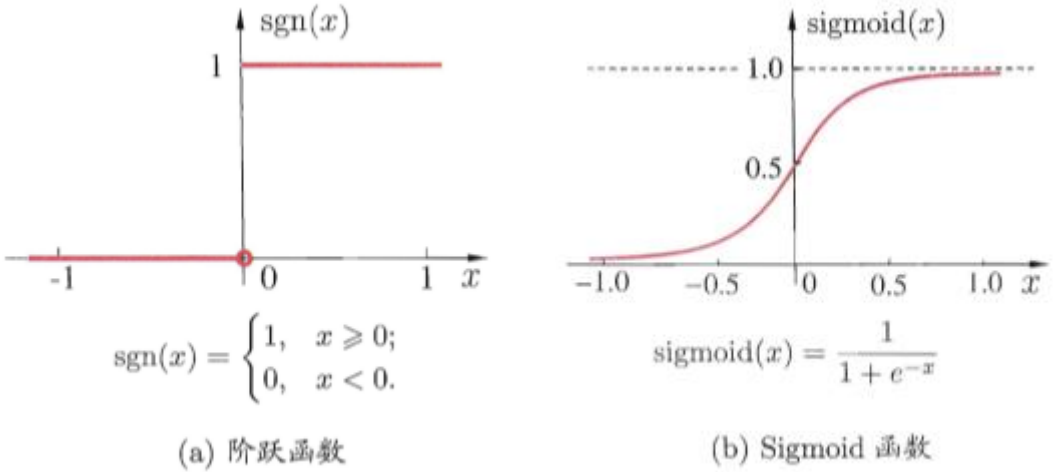
M-P神经元模型

简化：将$\sum_{i=1}^{n} w_i x_i$改写成点乘，$w \cdot x = \sum_{i=1}^{n} w_i x_i$，

这里w和x对应权重和输入的向量。偏置$b = -\theta$。

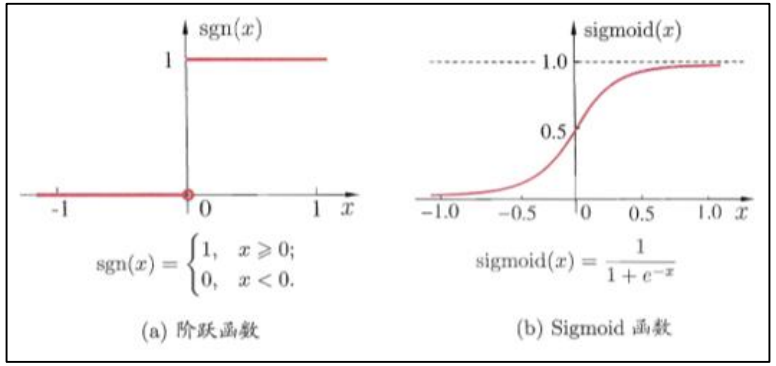
$$y = f(w \cdot x + b)$$

激活函数（activation function）

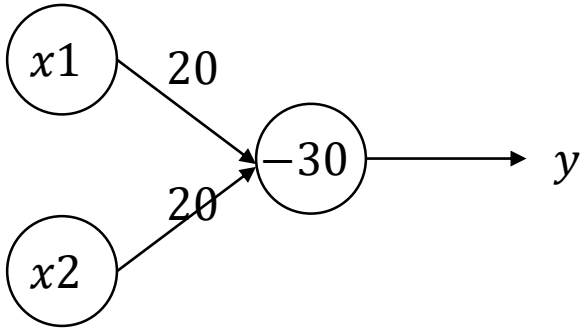


备注：$\sigma'(x) = \sigma(x) \times (1 - \sigma(x))$

神经元模型

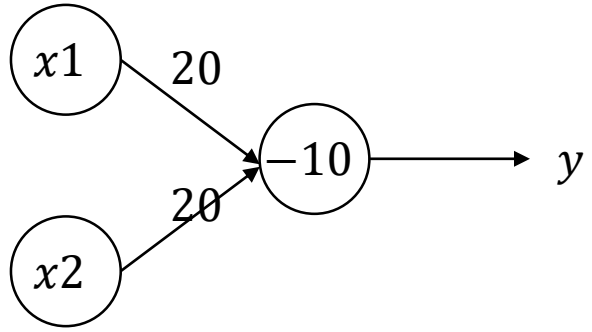


$\boldsymbol{x1, x2 \in \{0, 1\}}$
$\boldsymbol{y = x1\ AND\ x2}$



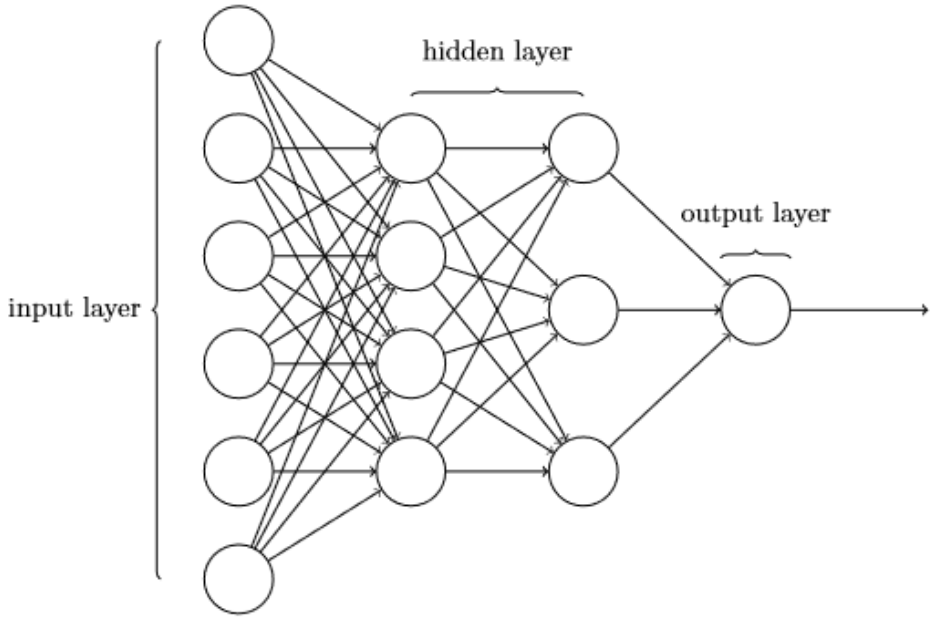
x1	x2	$y = f(w \cdot x + b)$
0	0	$f(20 \times 0 + 20 \times 0 - 30) \approx 0$
0	1	$f(20 \times 0 + 20 \times 1 - 30) \approx 0$
1	0	$f(20 \times 1 + 20 \times 0 - 30) \approx 0$
1	1	$f(20 \times 1 + 20 \times 1 - 30) \approx 1$

$\boldsymbol{x1, x2 \in \{0, 1\}}$
$\boldsymbol{y = x1\ OR\ x2}$

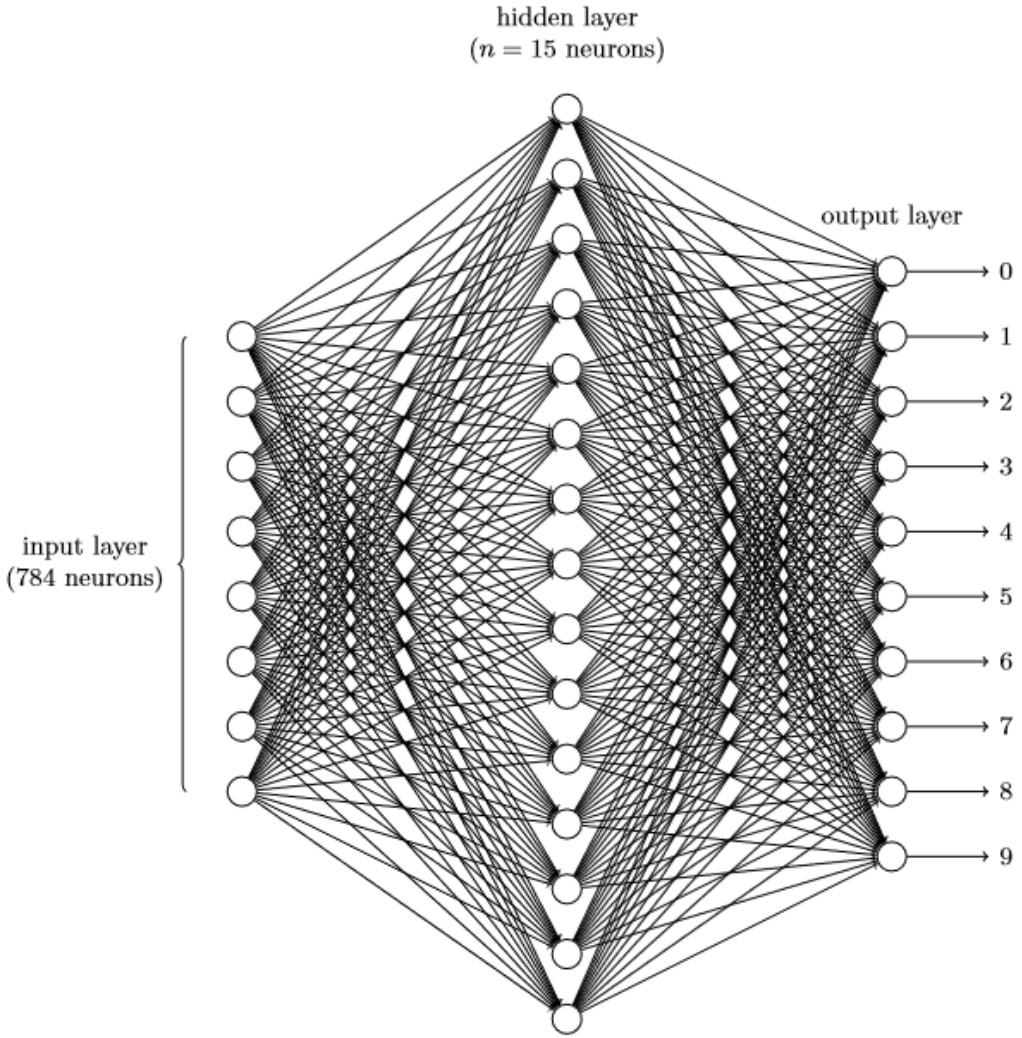


x1	x2	$y = f(w \cdot x + b)$
0	0	$f(20 \times 0 + 20 \times 0 - 10) \approx 0$
0	1	$f(20 \times 0 + 20 \times 1 - 10) \approx 1$
1	0	$f(20 \times 1 + 20 \times 0 - 10) \approx 1$
1	1	$f(20 \times 1 + 20 \times 1 - 10) \approx 1$

神经网络结构



二分类



多分类

代价函数（Cost function）

$$C(w,b) \equiv \frac{1}{2n} \sum_x \|y(x) - a\|^2$$

w：权重的集合。

b：偏置的集合。

n：输入数据的数量。

a：当输入为x时输出的向量。

||v||：指向量v的模。

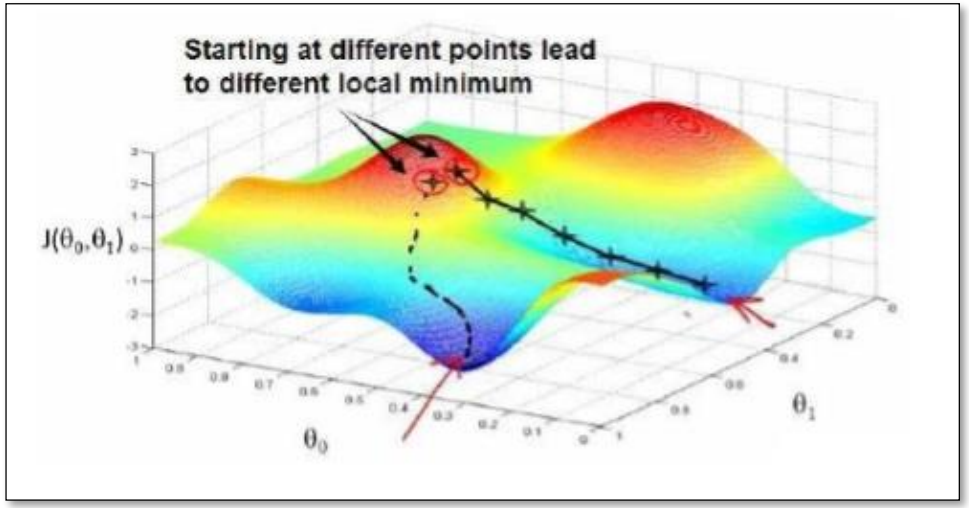
求和运算遍历每个训练样本x。

我们的目标是找到合适的w和b，使得 $C(w,b) \approx 0$。我们采用梯度下降算法达到这个目的。

梯度下降

梯度下降是一个用来求函数最小值的算法，我们将使用梯度下降算法来求代价函数的最小值。

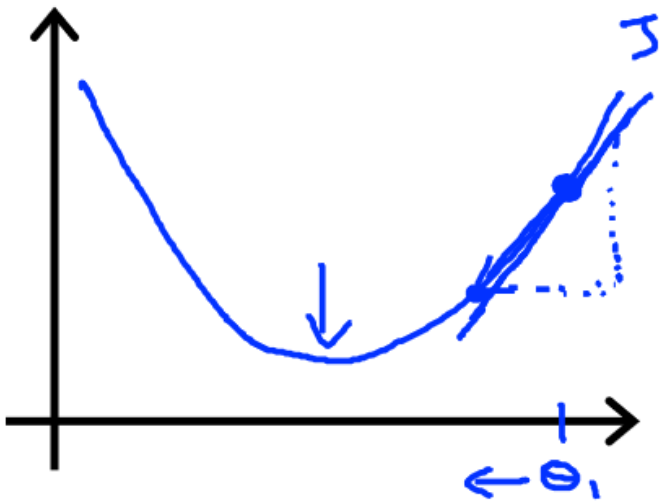
梯度下降背后的思想是：开始时我们随机选择一个参数的组合（$\theta_1, \theta_2, \ldots, \theta_n$），计算代价函数，然后我们寻找下一个能让代价函数值下降最多的参数组合。我们持续这么做直到到一个局部最小值。因为我们并没有尝试完所有的参数组合，所以不能确定我们得到的局部最小值是否便是全局最小值。选择不同的初始参数组合，可能会找到不同的局部最小值。



批量梯度下降（batch gradient descent）算法的公式为：

$$
\begin{aligned}
&\text{repeat until convergence } \{ \\
&\quad \theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1) \qquad (\text{for } j = 0 \text{ and } j = 1) \\
&\}
\end{aligned}
$$

梯度下降

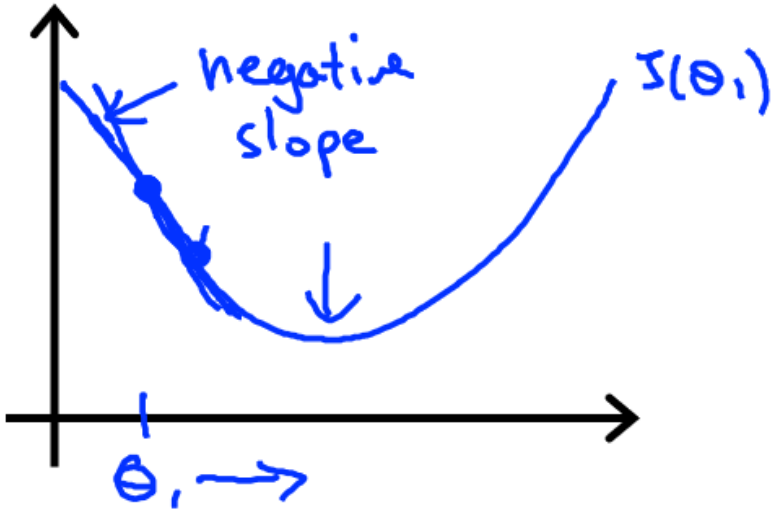


$(\theta_1 \in \mathbb{R})$

$\theta_1 := \theta_1 - \alpha \frac{d}{d\theta_1} J(\theta_1)$

$\frac{d}{d\theta_1} J(\theta_1) \geq 0$

$\theta_1 := \theta_1 - \alpha \cdot (\text{positive number})$



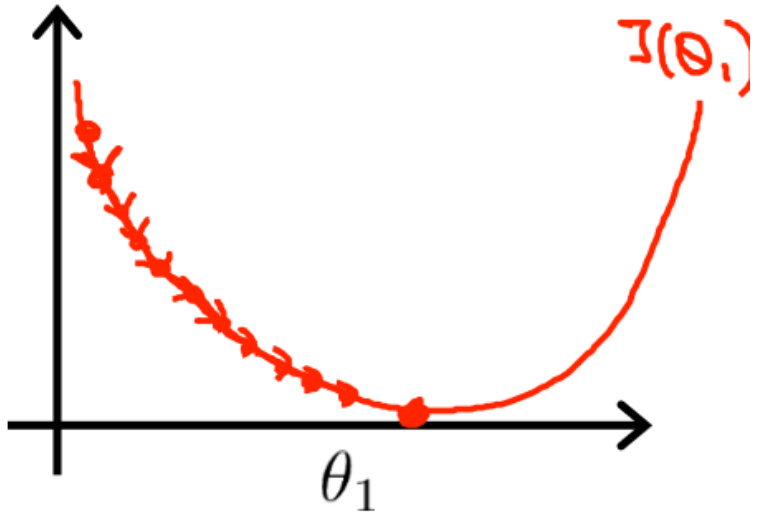
$\frac{d}{d\theta_1} J(\theta_1) \leq 0$

$\theta_1 := \theta_1 - \alpha \, (\text{negative number})$

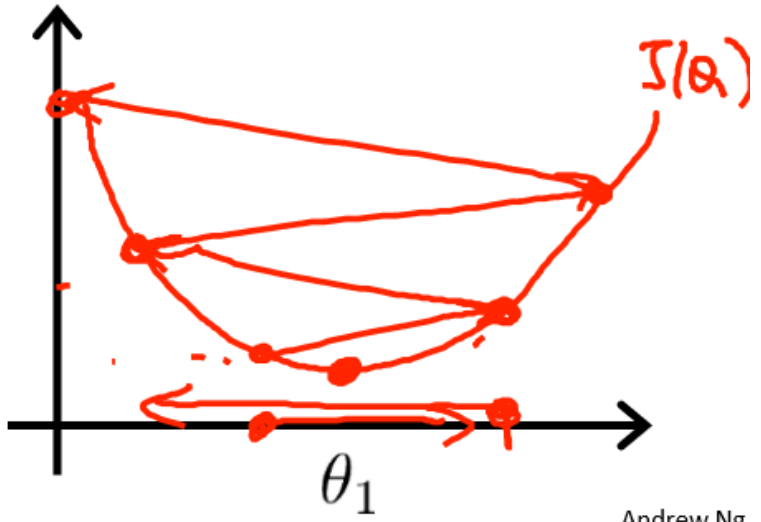
梯度下降

$$\theta_1 := \theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_1)$$

If α is too small, gradient descent can be slow.



If α is too large, gradient descent can overshoot the minimum. It may fail to converge, or even diverge.



Andrew Ng

梯度下降

线性回归中梯度下降的比较：批量梯度下降(BGD)、随机梯度下降(SGD)、小批量梯度下降(MBGD)

线性回归的假设函数：$h_\theta(x^{(i)}) = \theta_1 x^{(i)} + \theta_0$　　代价函数为：$J(\theta_0,\theta_1) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2$

其中$i = 1,2,\dots,m$表示样本数，$j = 0,1$表示特征数，这里我们使用偏置项$x_0^{(i)} = 1$。

批量梯度下降(BGD)：最原始的形式，在每一次迭代时使用所有样本来对参数进行更新。

$$repeat\{$$
$$\theta_j = \theta_j - \alpha\frac{1}{m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)x_j^{(i)} \quad (for\ j = 0,1)$$
$$\}$$

随机梯度下降(SGD)：在每一次迭代时使用一个样本来对参数进行更新。

$$repeat\{$$
$$for\ i = 1,2,\dots,m\ \{$$
$$\theta_j = \theta_j - \alpha\left(h_\theta(x^{(i)}) - y^{(i)}\right)x_j^{(i)} \quad (for\ j = 0,1)$$
$$\}$$
$$\}$$

小批量梯度下降(MBGD)：在每一次迭代时使用$batch_size$个样本来对参数进行更新。

假设$batch_size = 10, m = 1000$。

$$repeat\{$$
$$for\ i = 1,11,21,\dots,991\ \{$$
$$\theta_j = \theta_j - \frac{\alpha}{10}\sum_{k=i}^{(i+9)}\left(h_\theta(x^{(i)}) - y^{(i)}\right)x_j^{(i)} \quad (for\ j = 0,1)$$
$$\}$$
$$\}$$

$$\frac{\partial J^{(i)}(\theta_0,\theta_1)}{\partial \theta_j}$$

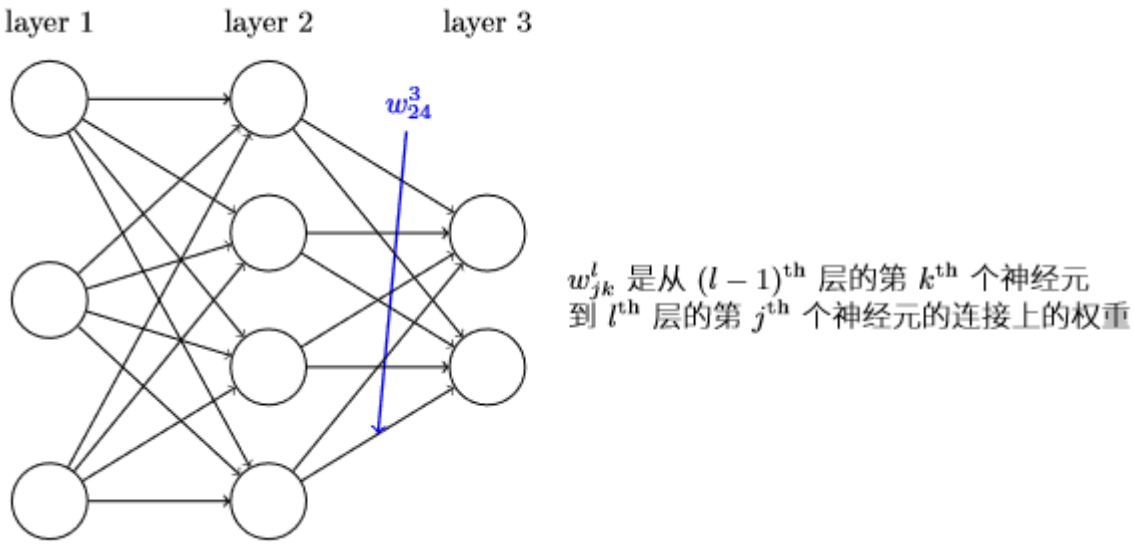
梯度下降

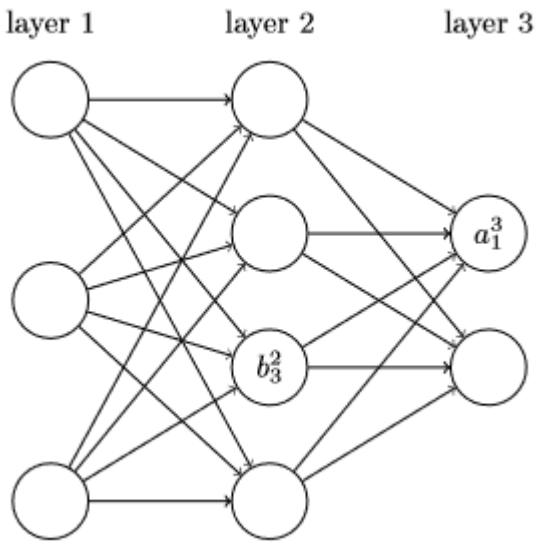
神经网络的目标：找到使得代价函数$C(w,b)$取得最小值的权重w和偏置b。

批量梯度下降BGD	随机梯度下降SGD	小批量梯度下降MBGD
每一次迭代使用所有训练样本对参数进行更新	每一次迭代使用一个训练样本对参数进行更新	每一次迭代选取小批量训练样本对参数进行更新
所有训练输入标记为$X_1, X_2, \dots, X_n$ $w_k \to w_k' = w_k - \frac{\alpha}{n}\sum_{i=1}^{n}\frac{\partial C_{X_i}}{\partial w_k}$ $b_l \to b_l' = b_l - \frac{\alpha}{n}\sum_{i=1}^{n}\frac{\partial C_{X_i}}{\partial b_l}$	一个训练输入样本标记为X_j $w_k \to w_k' = w_k - \alpha\frac{\partial C_{X_j}}{\partial w_k}$ $b_l \to b_l' = b_l - \alpha\frac{\partial C_{X_j}}{\partial b_l}$	随机训练输入标记为$X_1, X_2, \dots, X_m$ $w_k \to w_k' = w_k - \frac{\alpha}{m}\sum_{i=1}^{m}\frac{\partial C_{X_i}}{\partial w_k}$ $b_l \to b_l' = b_l - \frac{\alpha}{m}\sum_{i=1}^{m}\frac{\partial C_{X_i}}{\partial b_l}$
优点：目标函数为凸函数时，能得到全局最优解；易于并行实现 缺点：当样本数目很多时，训练过程会很慢。	优点：训练速度快。 缺点：准确度下降，并不是全局最优；不易于并行实现	降低了BGD和SGD的缺点，又结合了BGD和SGD的优点。

反向传播算法

明确网络中权重w_{jk}^{l}、偏置b_{j}^{l}和激活值a_{j}^{l}的清晰定义

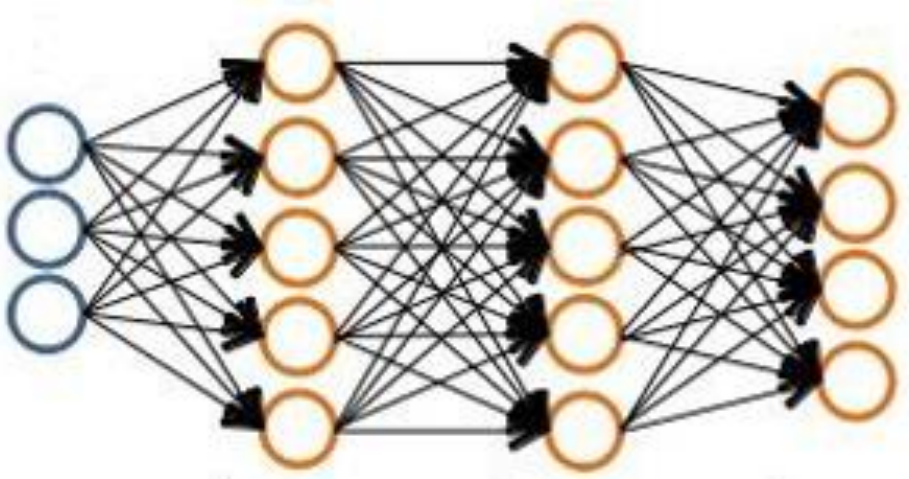




w_{jk}^{l}表示从$(l-1)^{th}$层的k^{th}个神经元到l^{th}层的j^{th}个神经元的链接上的权重。
b_{j}^{l}表示l^{th}层第j^{th}个神经元的偏置。
a_{j}^{l}表示l^{th}层第j^{th}个神经元的激活值。

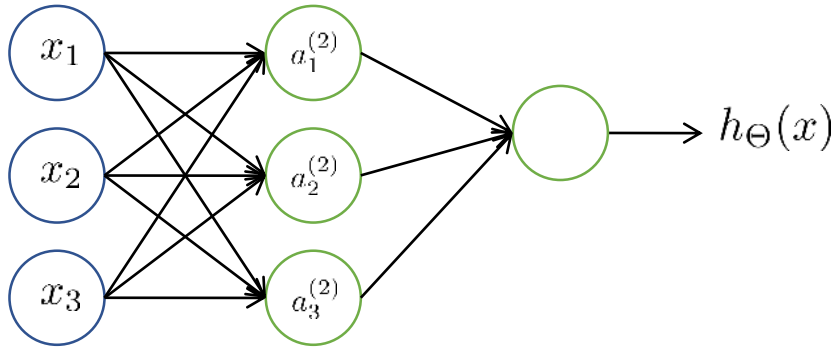
反向传播算法

1. **输入$\boldsymbol{x}$**：为输入层设置对应的激活值a^1。
2. **前向传播**：对每个$l = 2,3,\ldots,L$计算相应的$\mathrm{z}^l = w^l a^{l-1} + b^l$和$a^l = \sigma(z^l)$。
3. **输出层误差δ^{L}**：计算向量$\delta^{\mathrm{L}} = \nabla_{\mathrm{a}} C \odot \sigma'(z^L)$。
4. **反向误差传播**：对每个$l = L-1, L-2, \ldots, 2$，计算$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'\left(z^l\right)$。
5. **输出**：代价函数的梯度由$\frac{\partial \mathrm{C}}{\partial w_{jk}^l} = a_k^{l-1}\delta_j^l$和$\frac{\partial C}{\partial \mathrm{b}_j^l} = \delta_j^l$。



反向传播算法

前向传播



$$a_j^l = \sigma\left(\sum_k w_{jk}^l a_k^{l-1} + b_j^l\right)$$

$$a^l = \sigma(w^l a^{l-1} + b^l)$$

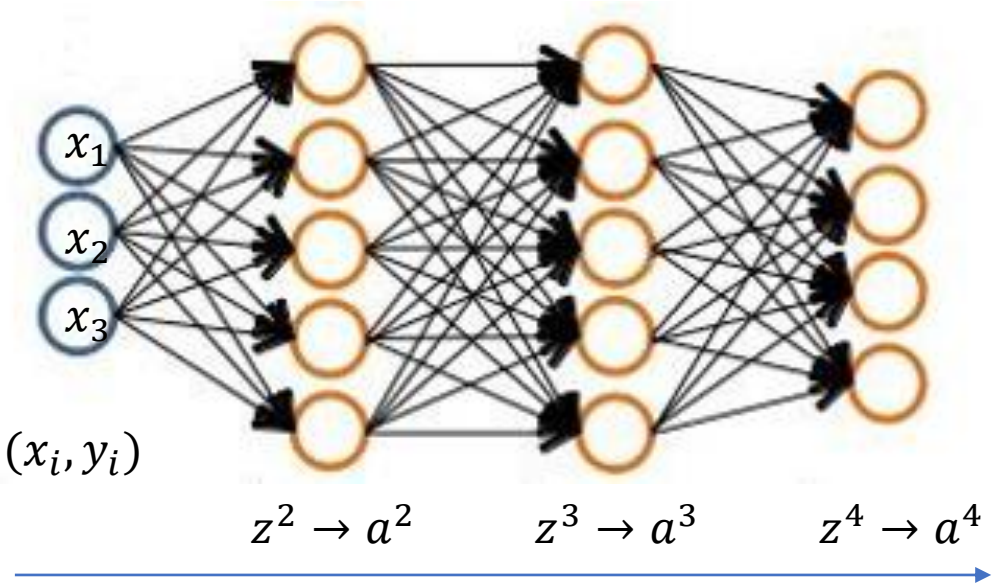
$$a_1^1 = x_1, a_2^1 = x_2, a_3^1 = x_3;$$
$$a_1^2 = \sigma(w_{11}^2 a_1^1 + w_{12}^2 a_2^1 + w_{13}^2 a_3^1 + b_1^2);$$
$$a_2^2 = \sigma(w_{21}^2 a_1^1 + w_{22}^2 a_2^1 + w_{23}^2 a_3^1 + b_2^2);$$
$$a_3^2 = \sigma(w_{31}^2 a_1^1 + w_{32}^2 a_2^1 + w_{33}^2 a_3^1 + b_3^2);$$
$$output = a_1^3 = \sigma(w_{11}^3 a_1^2 + w_{12}^3 a_2^2 + w_{13}^3 a_3^2 + b_1^3);$$

$$a^2 = \begin{bmatrix} a_1^2 \\ a_2^2 \\ a_3^2 \end{bmatrix} = \sigma\left(\begin{bmatrix} w_{11}^2 & w_{12}^2 & w_{13}^2 \\ w_{21}^2 & w_{22}^2 & w_{23}^2 \\ w_{31}^2 & w_{32}^2 & w_{33}^2 \end{bmatrix} \times \begin{bmatrix} a_1^1 \\ a_2^1 \\ a_3^1 \end{bmatrix} + \begin{bmatrix} b_1^2 \\ b_2^2 \\ b_3^2 \end{bmatrix}\right);$$

$$output = a^3 = \sigma([w_{11}^3 \quad w_{12}^3 \quad w_{13}^3] \times \begin{bmatrix} a_1^2 \\ a_2^2 \\ a_3^2 \end{bmatrix} + [b_1^3])$$

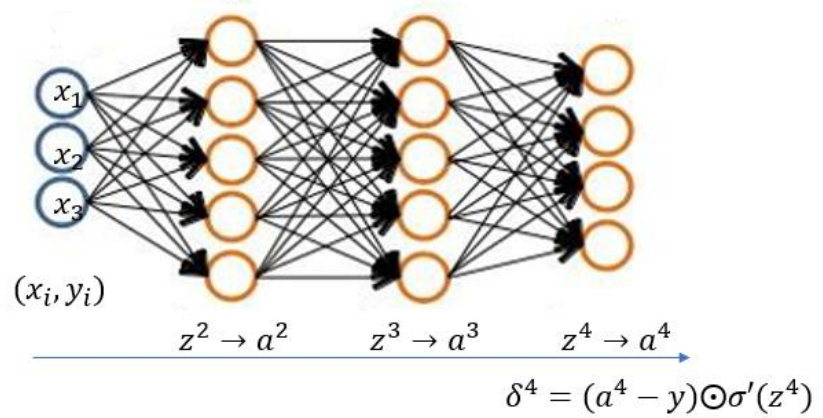
反向传播算法

前向传播



$$a^1 = x;$$
$$z^2 = w^2 a^1 + b^2;$$
$$a^2 = \sigma(z^2);$$
$$z^3 = w^3 a^2 + b^3;$$
$$a^3 = \sigma(z^3);$$
$$z^4 = w^4 a^3 + b^4;$$
$$a^4 = output = \sigma(z^4);$$

反向传播算法



输出层误差

反向传播的终极目标是计算偏导数$\partial C/\partial w_{jk}^l$和$\partial C/\partial b_j^l$。但是为了计算这些值，我们首先引入一个中间量$\delta_j^l$，我们称它为在$l^{th}$层第$j^{th}$个神经元上的**误差**。反向传播将给出计算$\delta_j^l$的流程，然后将其关联到计算$\partial C/\partial w_{jk}^l$和$\partial C/\partial b_j^l$上。

我们定义l^{th}层第j^{th}个神经元上的误差$\delta_j^l = \frac{\partial C}{\partial z_j^l}$

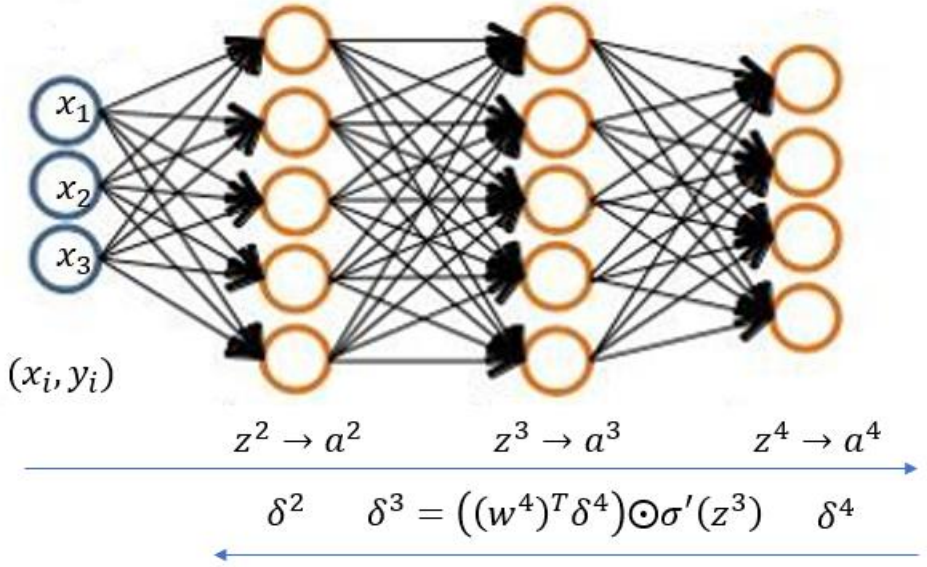
输出层误差方程：$\delta_j^{\mathrm{L}} = \frac{\partial C}{\partial a_j^L}\sigma'(z_j^L)$

矩阵式重写上述方程：$\delta^L = \nabla_a C \odot \sigma'(z^L)$

由于对于每个独立训练样本的代价函数$C_x = \frac{1}{2}||y - a^L||^2$，所以$\delta^L = (a^L - y)\odot\sigma'(z^L)$

反向传播算法

反向误差传播



使用下一层的误差δ^{l+1}来表示当前层的误差δ^l:

$$\delta^l = \left(\left(w^{l+1}\right)^T \delta^{l+1}\right) \odot \sigma'\left(z^l\right)$$

证明过程：上述公式给出了以下一层误差δ^{l+1}的形式表示误差δ^l。为此，我们想要以$\delta_k^{l+1} = \partial C / \partial z_k^{l+1}$的形式重写$\delta_j^l = \partial C / \partial z_j^l$。我们可以用链式法则:

$$\delta_j^l = \frac{\partial C}{\partial z_j^l} = \sum_k \frac{\partial C}{\partial z_k^{l+1}} \frac{\partial z_k^{l+1}}{\partial z_j^l} = \sum_k \frac{\partial z_k^{l+1}}{\partial z_j^l} \delta_k^{l+1} \quad (1)$$

这里最后一步我们交换了右边的两项，并用δ_k^{l+1}代入。为了对$\frac{\partial z_k^{l+1}}{\partial z_j^l}$求值:

$$z_k^{l+1} = \sum_j w_{kj}^{l+1} a_j^l + b_k^{l+1} = \sum_j w_{kj}^{l+1} \sigma(z_j^l) + b_k^{l+1} (2)$$

做微分，我们得到:

$$\frac{\partial z_k^{l+1}}{\partial z_j^l} = w_{kj}^{l+1} \sigma'(z_j^l) \ (3)$$

将它代入(1)，我们得到:

$$\delta_j^l = \sum_k w_{kj}^{l+1} \delta_k^{l+1} \sigma'(z_j^l) \ (4)$$

反向传播算法

代价函数的梯度

$$\frac{\partial C}{\partial w_{jk}^{l}} = a_{k}^{l-1}\delta_{j}^{l};\ \frac{\partial C}{\partial b_{j}^{l}} = \delta_{j}^{l}$$

证明过程：

$$\frac{\partial C}{\partial w_{jk}^{l}} = \frac{\partial C}{\partial z_{j}^{l}}\frac{\partial z_{j}^{l}}{w_{jk}^{l}} = a_{k}^{l-1}\delta_{j}^{l}$$

$$\frac{\partial C}{\partial b_{j}^{l}} = \frac{\partial C}{\partial z_{j}^{l}}\frac{\partial z_{j}^{l}}{\partial b_{j}^{l}} = \delta_{j}^{l}$$

反向传播算法

1. **输入训练样本的集合**
2. **对每个训练样本**x：设置对应的输入激活值a^1，并执行下面步骤：
   - **前向传播**：对每个$l = 2,3,\ldots,L$计算相应的$\mathrm{z}^l = w^l a^{l-1} + b^l$和$a^l = \sigma(z^l)$。
   - **输出层误差**δ^{L}：计算向量$\delta^{\mathrm{L}} = \nabla_{\mathrm{a}} C \odot \sigma'(z^L)$。
   - **反向误差传播**：对每个$l = L-1, L-2, \ldots, 2$，计算$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$。
3. **梯度下降**：对每个$l = L-1, L-2, \ldots, 2$，根据$w^l \to w^l - \frac{\alpha}{m}\sum_x \delta_j^l (a_k^{l-1})^T$和$b^l \to b^l - \frac{\alpha}{m}\sum_x \delta_j^l$更新权重和偏置。

总结：反向传播的四个方程式

$$\delta^L = \nabla_a C \odot \sigma'(z^L) \qquad \text{(BP1)}$$

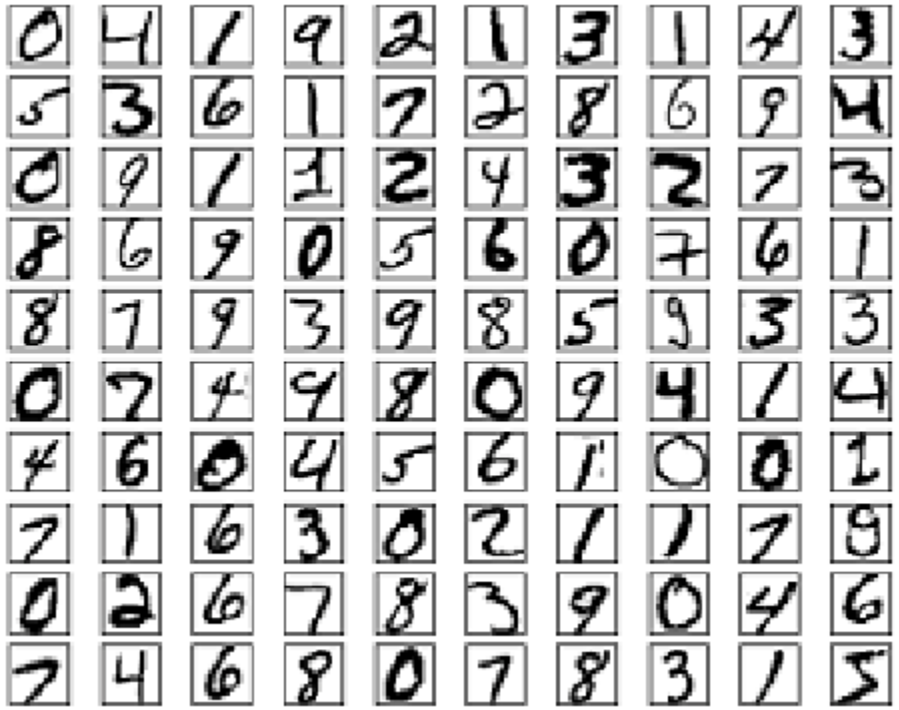
$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l) \qquad \text{(BP2)}$$

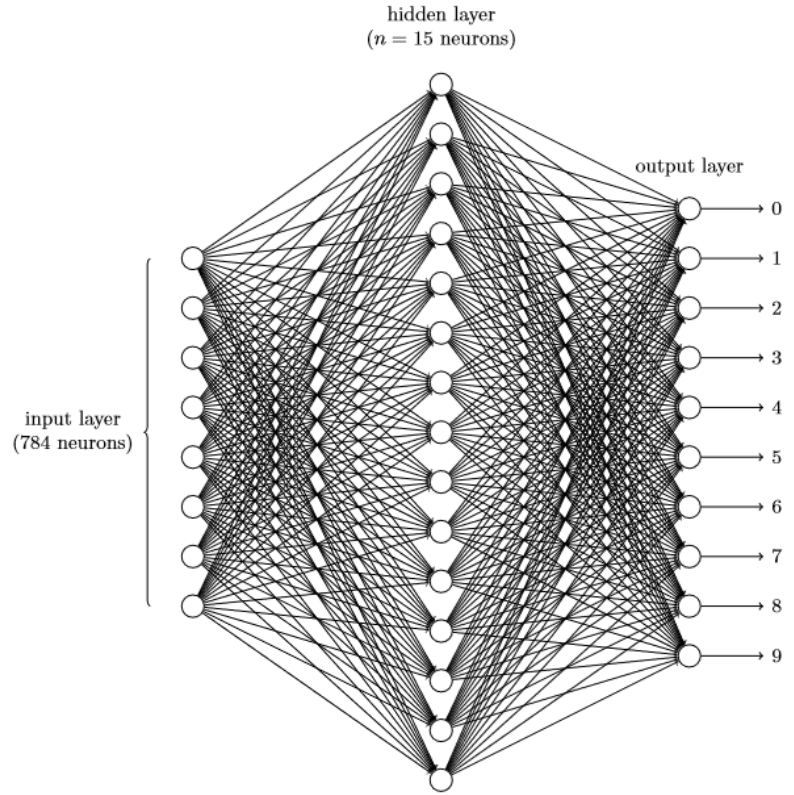
$$\frac{\partial C}{\partial b_j^l} = \delta_j^l \qquad \text{(BP3)}$$

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l \qquad \text{(BP4)}$$

使用神经网络识别手写数字

我们使用MNIST数据集。给网络的训练数据由28*28的手写数字的图像组成，输入层包含784=28*28个神经元。输入像素是灰度级的，值为0.0表示白色，值为1.0表示黑色，中间数值表示逐渐暗淡的灰色。





1. **输入训练样本的集合**
2. **对每个训练样本**x：设置对应的输入激活值a^1，并执行下面步骤：
   - **前向传播**：对每个$l = 2,3,\dots,L$计算相应的$z^l = w^l a^{l-1} + b^l$和$a^l = \sigma(z^l)$。
   - **输出层误差**δ^L：计算向量$\delta^L = \nabla_a C \odot \sigma'(z^L)$。
   - **反向误差传播**：对每个$l = L-1, L-2, \dots, 2$，计算$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$。
3. **梯度下降**：对每个$l = L-1, L-2, \dots, 2$，根据$w^l \to w^l - \frac{\alpha}{m}\sum_x \delta_j^l (a_k^{l-1})^T$和$b^l \to b^l - \frac{\alpha}{m}\sum_x \delta_j^l$更新权重和偏置。

```python
def gradient_descent(self, training_data, epochs, mini_batch_size, alpha, test_data=None):
    """MBGD，运行一个或者几个batch时更新一次

    :param training_data: 训练数据，每一个样本包括(x, y)，类型为zip
    :epochs: 迭代次数
    :mini_batch_size：每一个小批量数据的数量
    :alpha: 学习率
    :test_data: 测试数据
    """

    training_data = list(training_data)
    n = len(training_data)
    if test_data:
        test_data = list(test_data)
        n_test = len(list(test_data))
    for i in range(epochs):
        random.shuffle(training_data)
        mini_batches = [training_data[k:k+mini_batch_size] for k in range(0, n, mini_batch_size)]
        for mini_batch in mini_batches:
            init_ws_derivative = np.array([np.zeros(w.shape) for w in self.weights])
            init_bs_derivative = np.array([np.zeros(b.shape) for b in self.biases])
            for x, y in mini_batch:
                activations, zs = self.forwardprop(x) #前向传播
                delta = self.cost_deviation(activations[-1], zs[-1], y) #计算最后一层误差
                ws_derivative, bs_derivative = self.backprop(activations, zs, delta) #反向传播，cost func对w和b求偏导
                init_ws_derivative = init_ws_derivative + ws_derivative
                init_bs_derivative = init_bs_derivative + bs_derivative
            self.weights = self.weights - alpha / len(mini_batch) * init_ws_derivative
            self.biases = self.biases - alpha / len(mini_batch) * init_bs_derivative
        if test_data:
            print("Epoch {} : {} / {}".format(i, self.evaluate(test_data), n_test)) #识别准确数量/测试数据集总数量
        else:
            print("Epoch {} complete".format(i))
```

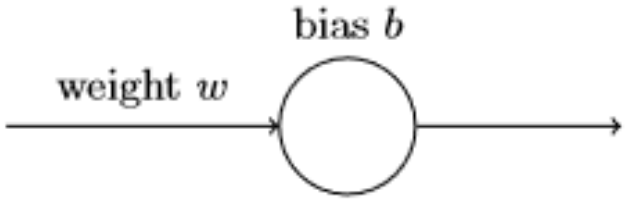
```python
def forwardprop(self, x):
    """前向传播"""

    activation = x
    activations = [x]
    zs = []
    for w, b in zip(self.weights, self.biases):
        z = np.dot(w, activation) + b
        zs.append(z)
        activation = sigmoid(z)
        activations.append(activation)
    return (activations, zs)


def cost_deviation(self, output, z, y):
    """计算最后一层误差"""

    return (output - y) * sigmoid_derivative(z)


def backprop(self, activations, zs, delta):
    """反向传播"""

    ws_derivative = np.array([np.zeros(w.shape) for w in self.weights])
    bs_derivative = np.array([np.zeros(b.shape) for b in self.biases])
    ws_derivative[-1] = np.dot(delta, activations[-2].transpose())
    bs_derivative[-1] = delta

    for l in range(2, self.num_layers):
        z = zs[-l]
        delta = np.dot((self.weights[-l+1]).transpose(), delta) * sigmoid_derivative(z)
        ws_derivative[-l] = np.dot(delta, activations[-l-1].transpose())
        bs_derivative[-l] = delta

    return (ws_derivative, bs_derivative)
```

交叉熵代价函数

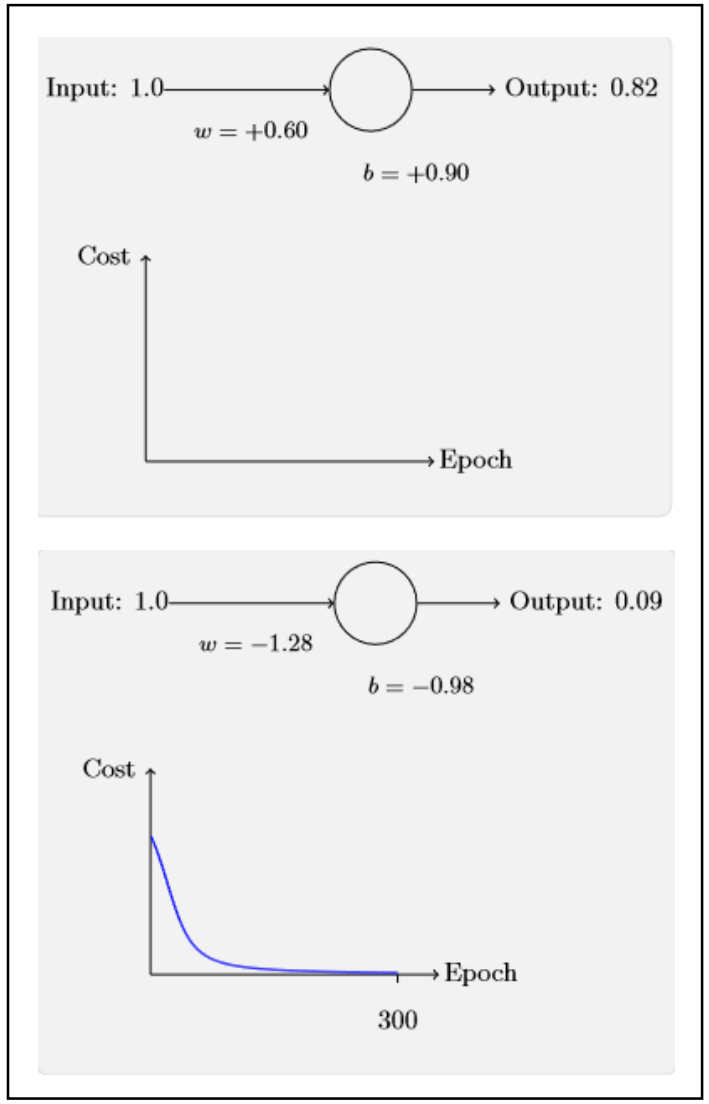
为什么引入交叉熵代价函数?

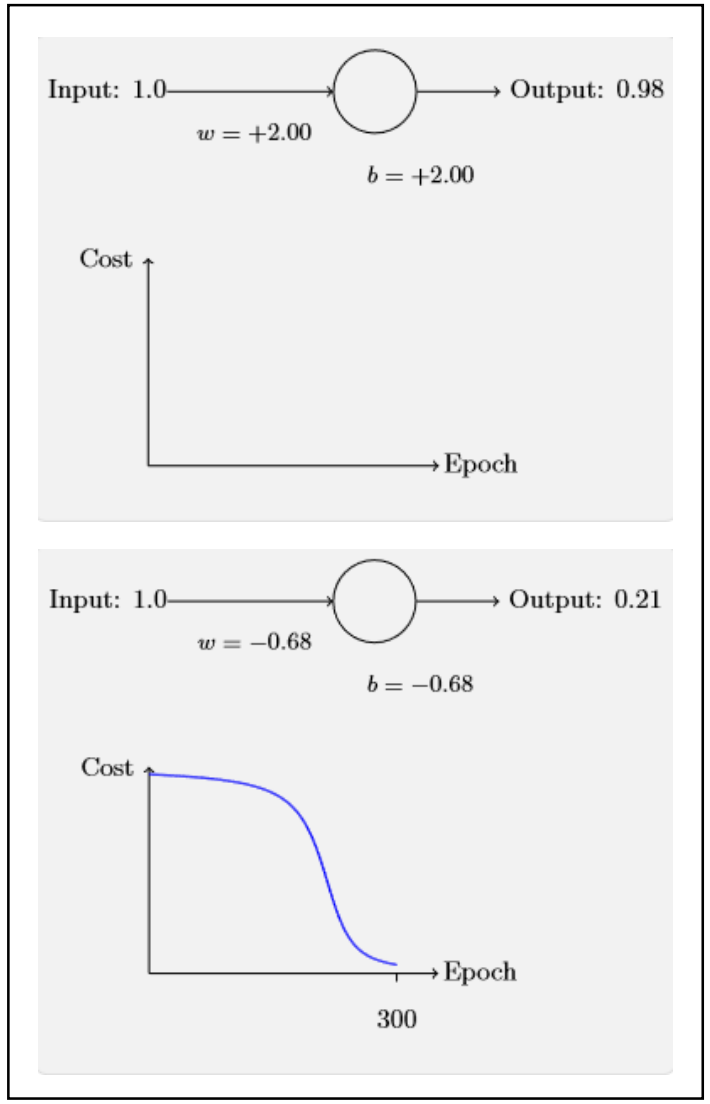
下图是一个仅只有一个输入的神经元，我们会训练这个神经元做一件非常简单的事：让输入1转化为0。我们来看看神经元是如何学习，更新权重和偏置的。

该案例中我们使用的是二次代价函数。我们初始化了两组权重和偏置来观察对比神经元的学习过程。两组实验均设置学习率$\alpha = 0.15$。



交叉熵代价函数






```
IPython: C:Users/yangkui

In [86]: def sigmoid(x):
    ...:     return 1.0 / (1.0 + np.exp(-x))
    ...:

In [87]: def sigmoid_derivative(x):
    ...:     return sigmoid(x) * (1.0 - sigmoid(x))
    ...:

In [88]: def iter(w, b, output):
    ...:     wd = sigmoid_derivative(w + b)
    ...:     bd = sigmoid_derivative(w + b)
    ...:     w = w - 0.15 * output * wd
    ...:     b = b - 0.15 * output * bd
    ...:     output = sigmoid(w + b)
    ...:     return (w, b, output)
    ...:

In [89]: w, b, output = 0.6, 0.9, sigmoid(w + b)

In [90]: for i in range(300):
    ...:     w,b,output = iter(w, b, output)
    ...:

In [91]: w
Out[91]: -1.282067691272107

In [92]: b
Out[92]: -0.982067691272106

In [93]: output
Out[93]: 0.09413713046290294
```

- 权重和偏置的初始值选择会影响学习过程。
- “学习缓慢”问题。

$$\frac{\partial C}{\partial w} = (a-y)\sigma'(z)x = a\sigma'(z)$$
$$\frac{\partial C}{\partial b} = (a-y)\sigma'(z) = a\sigma'(z)$$

由sigmoid函数图像我们知道，当神经元输出接近1的时候，曲线变得相当平缓，所以$\sigma'(z)$就很小。

交叉熵代价函数

$$C = -\frac{1}{n}\sum_x\sum_j \left[y_j \ln a_j^L + (1-y_j)\ln(1-a_j^L)\right]$$

$$\delta^L = \left(\frac{1-y}{1-a^L} - \frac{y}{a^L}\right)\sigma'(z^L) = \frac{a^L - y}{a^L(1-a^L)}\Big(\sigma(z^L)\big(1-\sigma(z^L)\big)\Big) = a^L - y$$

为什么交叉熵代价函数可以避免学习速度下降的问题?

$$\frac{\partial C}{\partial w_{jk}^L} = \frac{1}{n}\sum_x a_k^{L-1}(a_j^L - y_j)$$

$$\frac{\partial C}{\partial b_j^L} = \frac{1}{n}\sum_x (a_j^L - y_j)$$

当激活函数为sigmoid函数时，交叉熵一般都是更好的选择。

总结：反向传播的四个方程式

$$\delta^L = \nabla_a C \odot \sigma'(z^L) \quad \text{(BP1)}$$

$$\delta^l = ((w^{l+1})^T\delta^{l+1}) \odot \sigma'(z^l) \quad \text{(BP2)}$$

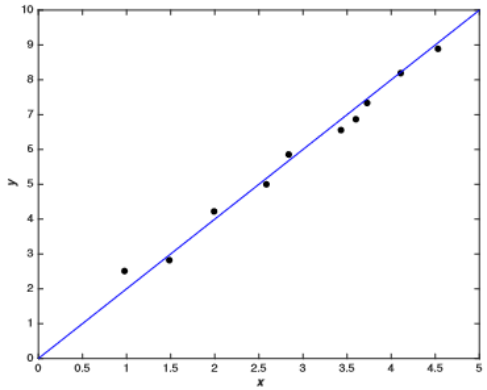
$$\frac{\partial C}{\partial b_j^l} = \delta_j^l \quad \text{(BP3)}$$

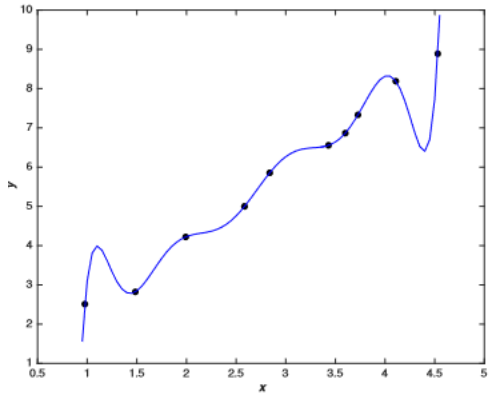
$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1}\delta_j^l \quad \text{(BP4)}$$

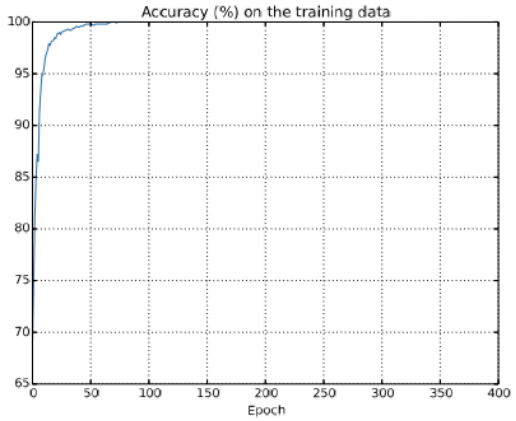
过度拟合和规范化

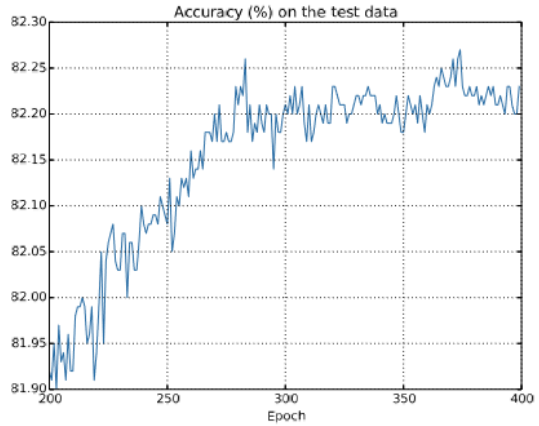
过度拟合(overfitting)是这样一种现象：一个假设在训练数据上能够获得比其它假设更好的拟合，但是在训练数据外的数据集上却不能很好的拟合数据。出现这种现象的主要原因是训练数据中存在噪声或者训练数据太少。

神经网络的训练，并不是希望模型尽量模拟训练的数据，而是希望模型对未来的数据具有准确的判断。









过度拟合和规范化

增加训练样本的数量是一种减轻过度拟合的方法。另外还有一种技术也可以缓解过度拟合，这种技术就是**规范化**。下面给出了一种最常用的规范化手段，有时候称为**权重衰弱(weight decay)**或者**L2规范化**。

L2规范化的想法是加一个额外的项到代价函数上，这个项叫做**规范化项**。下面是规划化方程:

$$C = -\frac{1}{n}\sum_{xj}\left[y_j \ln a_j^L + (1-y_j)\ln(1-a_j^L)\right] + \frac{\lambda}{2n}\sum_w w^2$$

$$C = \frac{1}{2n}\sum_x \|y - a^L\|^2 + \frac{\lambda}{2n}\sum_w w^2$$

$\lambda > 0$称为规范化参数

$$C = C_0 + \frac{\lambda}{2n}\sum_w w^2$$

计算权重和偏置的偏导数:

$$\frac{\partial C}{\partial w} = \frac{\partial C_0}{\partial w} + \frac{\lambda}{n}w \qquad \frac{\partial C}{\partial b} = \frac{\partial C_0}{\partial b}$$

梯度下降:

$$w \to w - \alpha\frac{\partial C_0}{\partial w} - \frac{\alpha\lambda}{n}w \qquad b \to b - \alpha\frac{\partial C_0}{\partial b}$$

$$= \left(1 - \frac{\alpha\lambda}{n}\right)w - \alpha\frac{\partial C_0}{\partial w}$$

过度拟合和规范化

规范化可以当作一种寻找小的权重和最小化原始的代价函数之间的折中。为什么规范化能够在实践中减少过度拟合？通常的说法是：小的权重在某种程度上，意味着更低的复杂性，也就对数据给出了一种更简单却更强大的解释，因此应该优先选择。

更小的权重意味着网络的行为不会因为我们随便改变一个输入而改变太大。这会让规范化网络学习局部噪声的影响更加困难。对比看，大的权重可能会因为输入的微小改变而产生比较大的行为改变。简言之，规范化网络根据训练数据中常见的模式来构建相对简单的模型，而能够抵抗训练数据中的噪声的特性影响。

Thank You