深度学习数学原理初探

10月17日 师振华

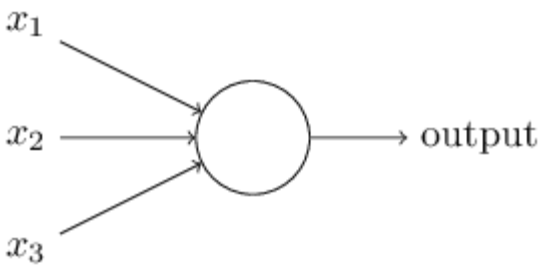
目录

第一节

神经网络

1.1 原型问题

1.2 感知器



$$\text{output} = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq \text{ threshold} \\ 1 & \text{if } \sum_j w_j x_j > \text{ threshold} \end{cases}$$

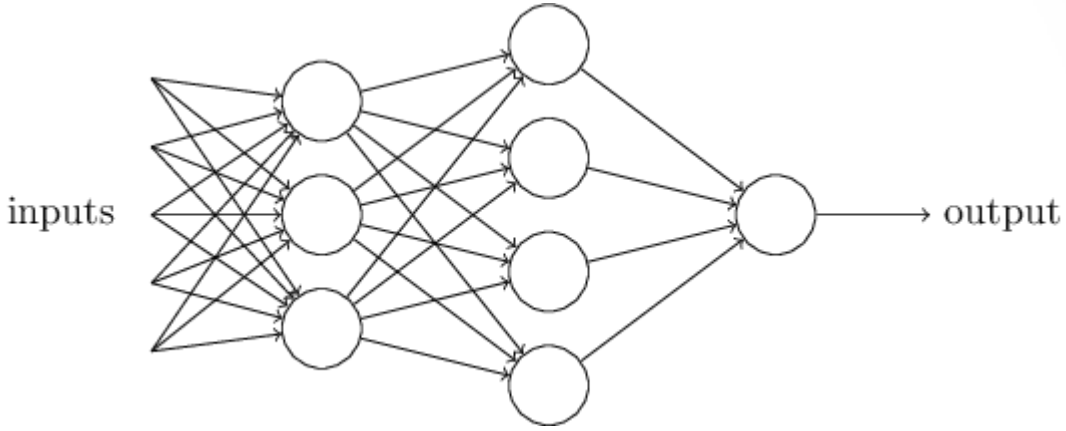
1.2 感知器

- 假设周末就要来了，你听说你所在的城市有个车展。你喜欢车（模），正试着决定是否去参加。你也许会通过给三个因素设置权重来作出决定：
- 1. 天气好不好？w1
- 2. 你的女朋友会不会陪你去？w2
- 3. 举办的地点是否靠近地铁公交？（你没有车）w3

$$\text{output} = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq \text{ threshold} \\ 1 & \text{if } \sum_j w_j x_j > \text{ threshold} \end{cases}$$

1.2 感知器

- 感知器网络

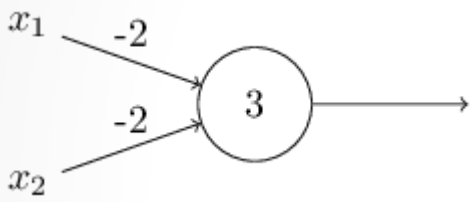


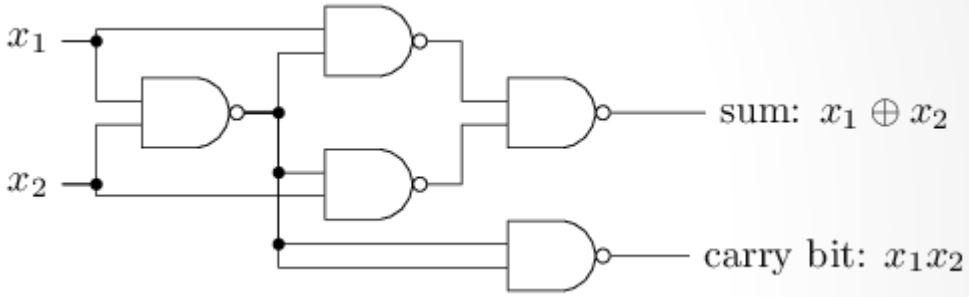
$$\text{output} = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq \text{ threshold} \\ 1 & \text{if } \sum_j w_j x_j > \text{ threshold} \end{cases}$$

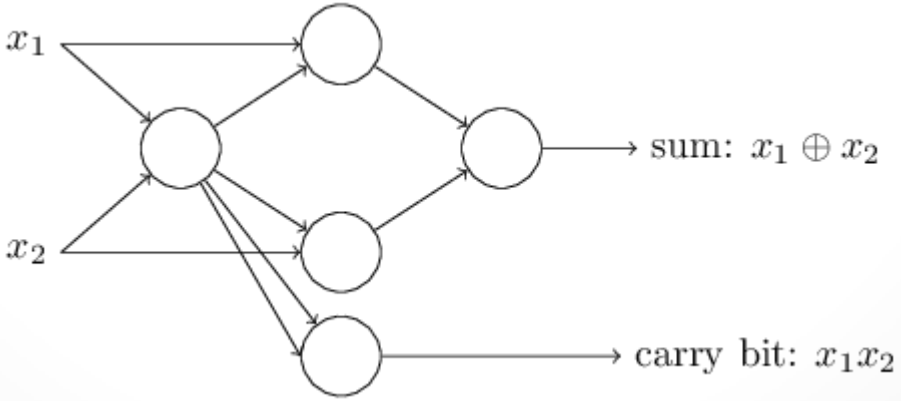
$$\text{output} = \begin{cases} 0 & \text{if } w \cdot x + b \leq 0 \\ 1 & \text{if } w \cdot x + b > 0 \end{cases}$$

1.2 感知器

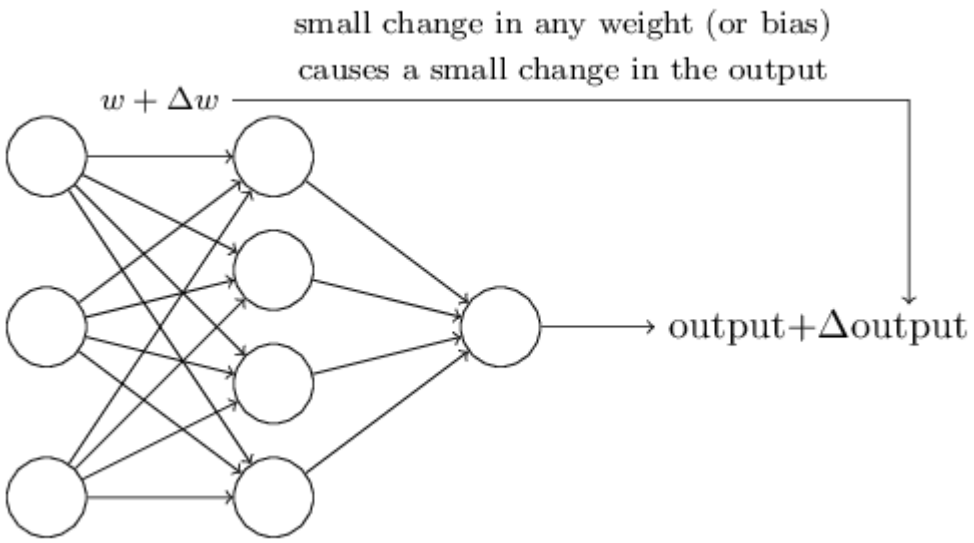
- 与非门





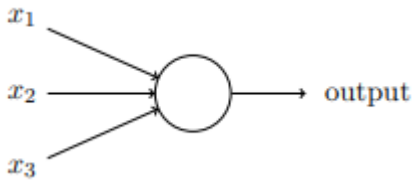


1.3 S形神经元



1.3 S形神经元

权重和偏置的微小改动只引起输出的微小变化

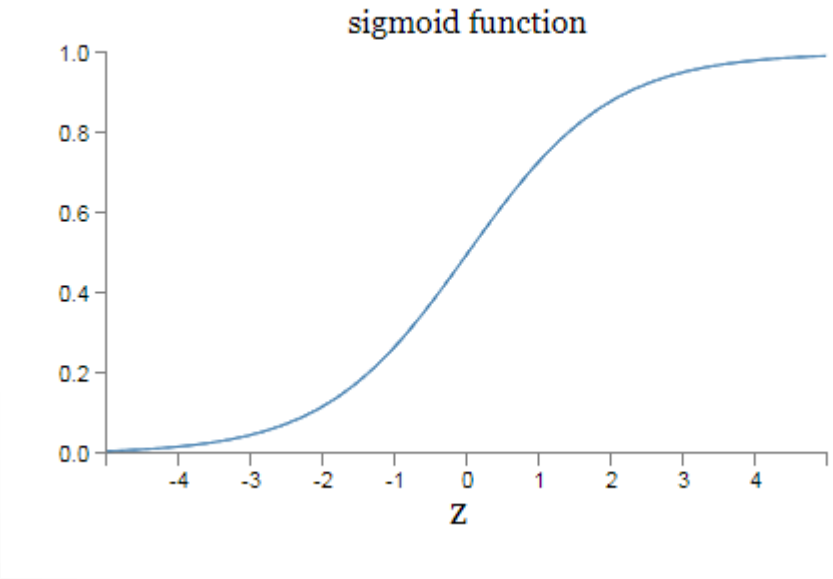


$$\sigma(z) \equiv \frac{1}{1+e^{-z}}.$$

$$\frac{1}{1+\exp(-\sum_j w_j x_j - b)}.$$

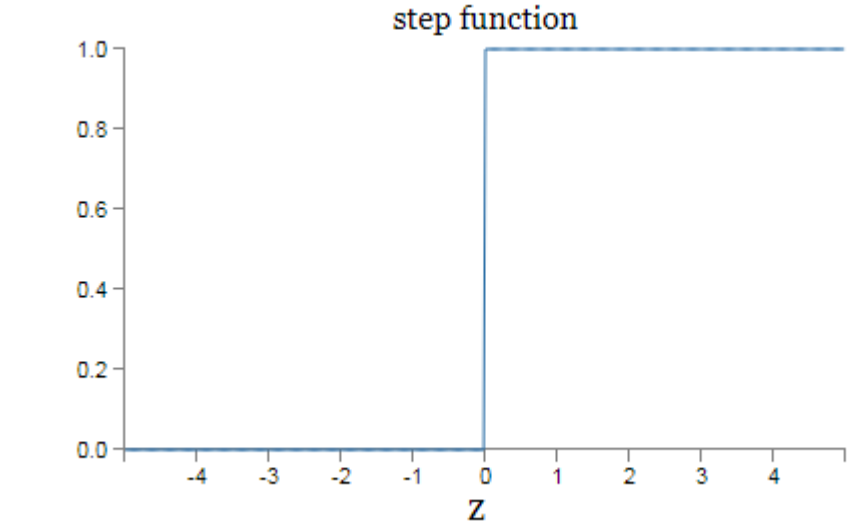
1.3 S形神经元

- S形神经元σ函数的形状

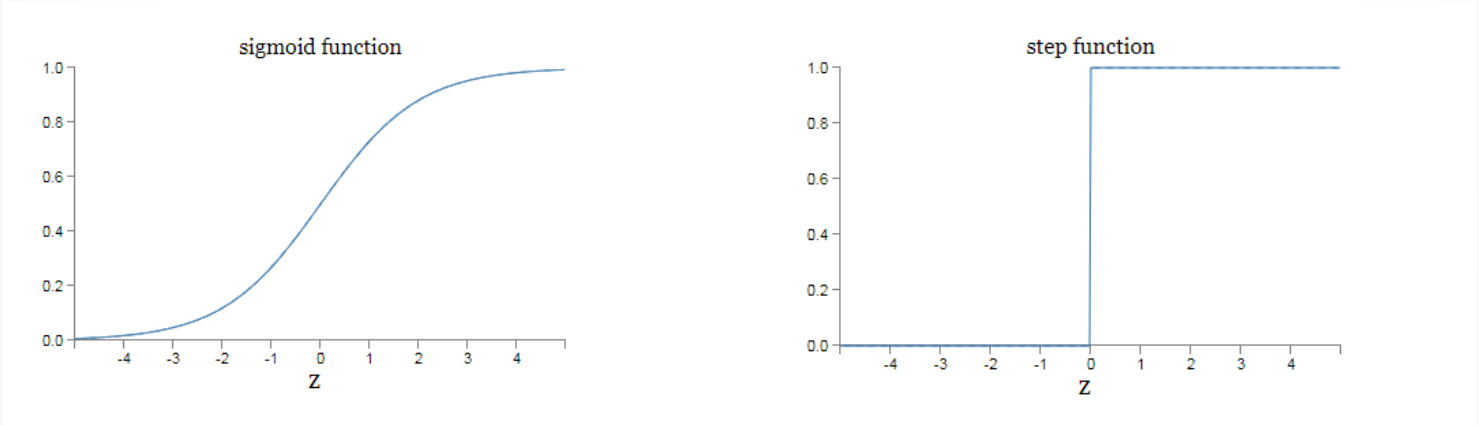


1.3 S形神经元

- 感知器的阶跃函数形状



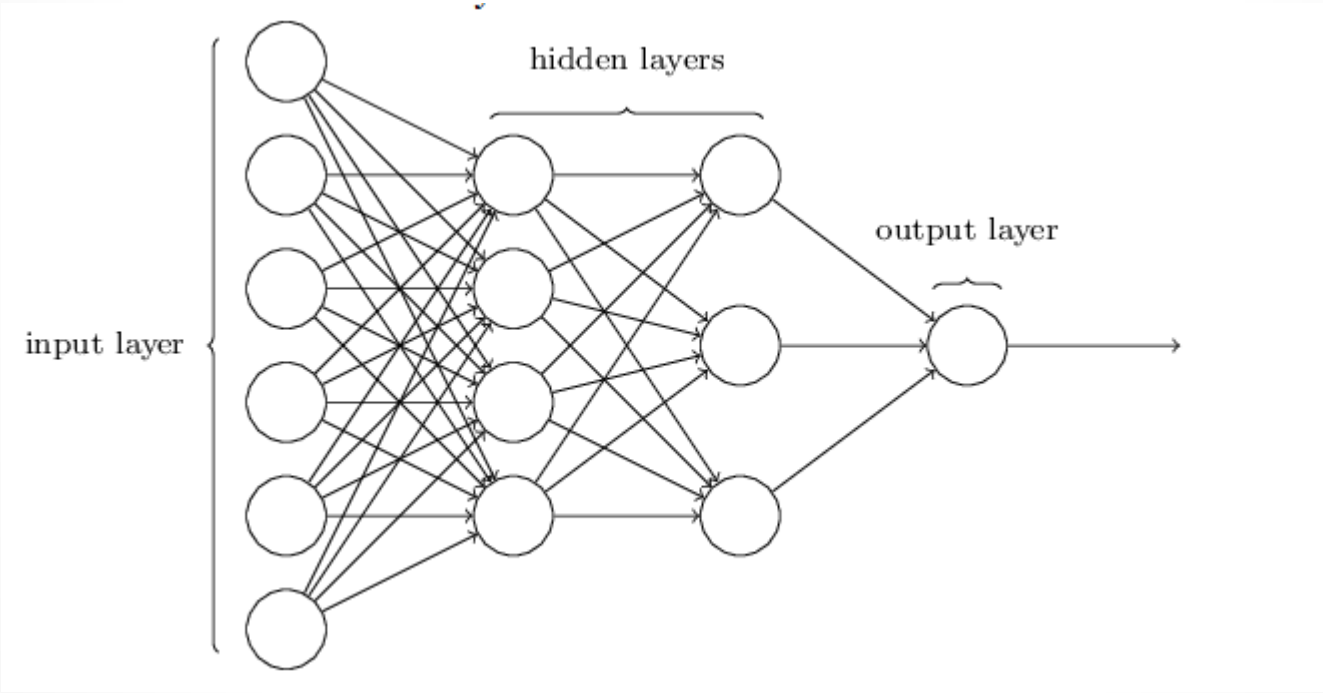
1.3 S形神经元



$$\Delta \text{output} \approx \sum_j \frac{\partial \, \text{output}}{\partial w_j} \Delta w_j + \frac{\partial \, \text{output}}{\partial b} \Delta b,$$

1.4 神经网络的架构

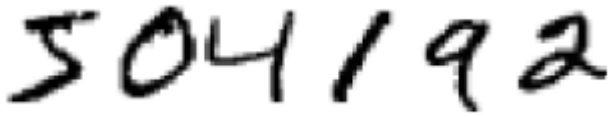
- 输入层、隐藏层、输出层



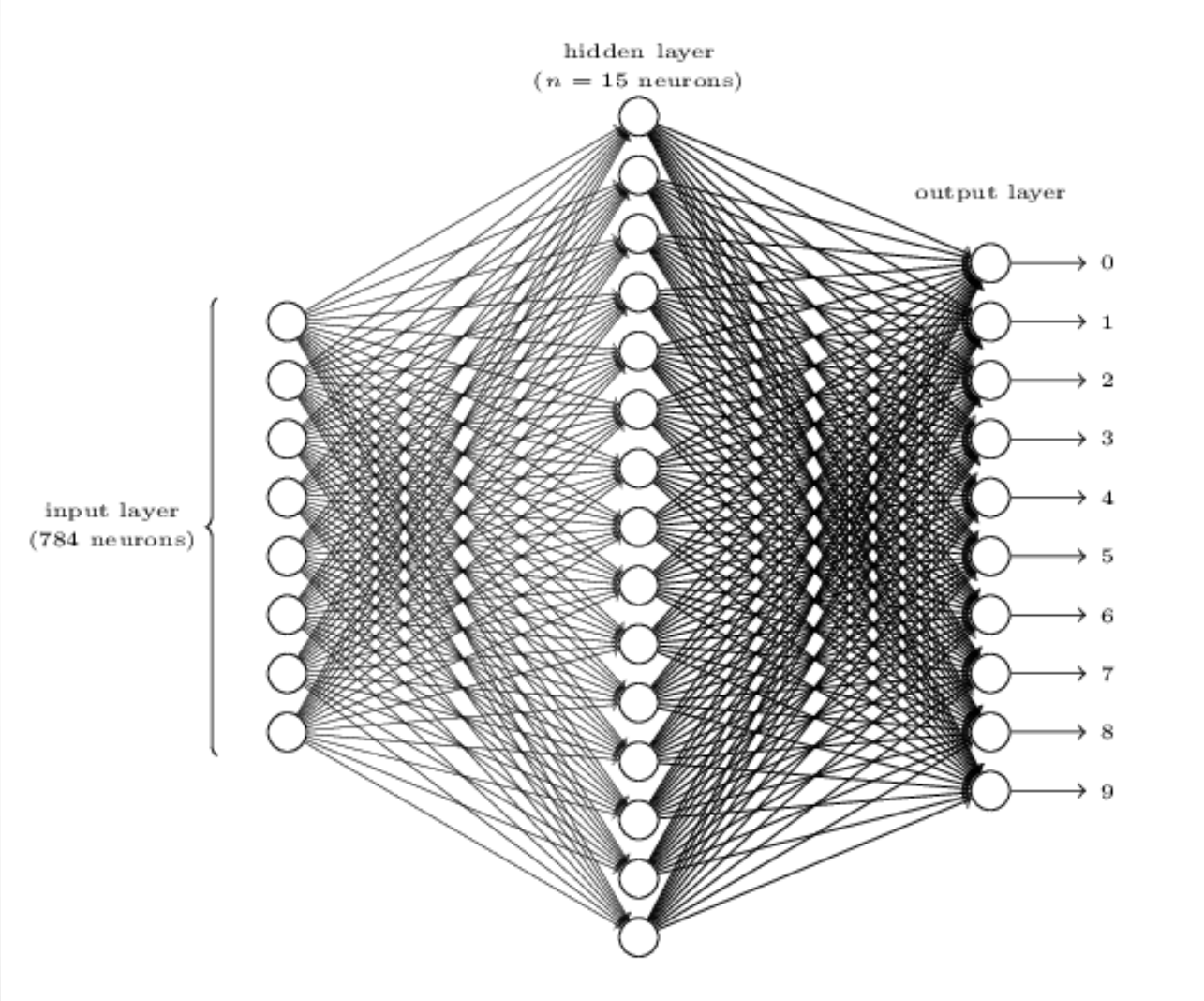
1.4 神经网络的架构

- 前馈神经网络
- 递归神经网络

1.5 分类手写数字



1.5 分类手写数字

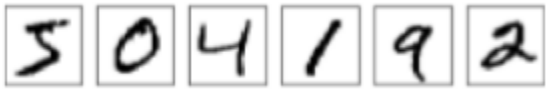


第二节

随机梯度下降

2.1 MNIST数据集

- 60000幅用于训练数据的图像
- 10000幅用于测试数据的图像
- 输入样本(28x28=784维的向量)
- 输出向量 $y(x) = (0,0,0,0,0,0,1,0,0,0)^T$



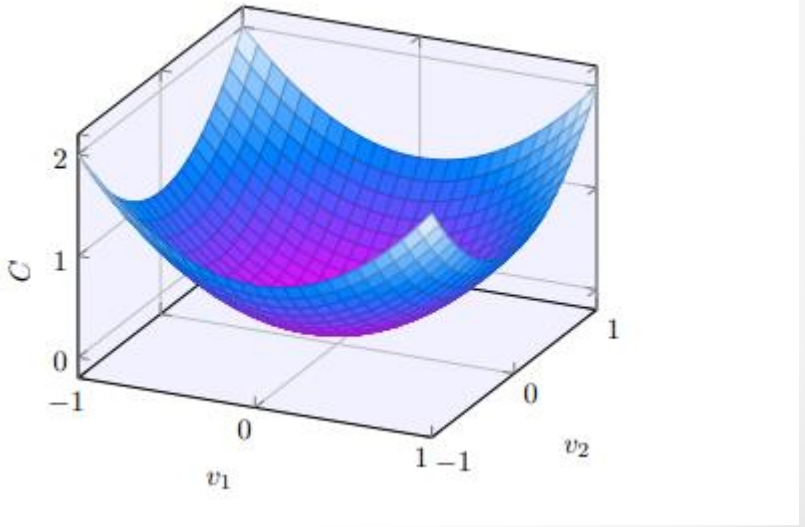
2.2 代价函数

- 二次代价函数(均方误差/MSE)
- y(x)越接近a时，C(w,b)≈0
- 我们想要找到一系列能让代价更小的权重(w)和偏置(b)
- 为什么使用二次代价函数？

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

2.3 梯度下降

- 现在把精力集中在最小化一个多元函数上
- 简单起见，假设是一个二元函数
- 微积分解析最小值



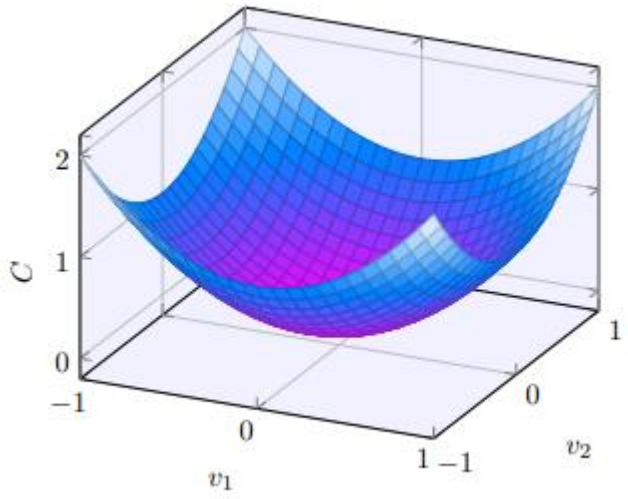
2.3 梯度下降

$$\Delta C \approx \frac{\partial C}{\partial v_1}\Delta v_1 + \frac{\partial C}{\partial v_2}\Delta v_2.$$

$$\Delta v \equiv (\Delta v_1, \Delta v_2)^T$$

$$\nabla C \equiv \left(\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2}\right)^T.$$

$$\Delta C \approx \nabla C \cdot \Delta v.$$



2.3 梯度下降

- 由于 $\|\nabla C\|^2 \geq 0$
- 所以保证了 $\Delta C \leq 0$
- 所以按照右上公式的规则去改变v，C会一直减少不会增加，直至接近于全局最小值

$$\Delta C \approx \nabla C \cdot \Delta v.$$

$$\Delta v = -\eta \nabla C,$$

$$\Delta C \approx -\eta \nabla C \cdot \nabla C = -\eta \|\nabla C\|^2.$$

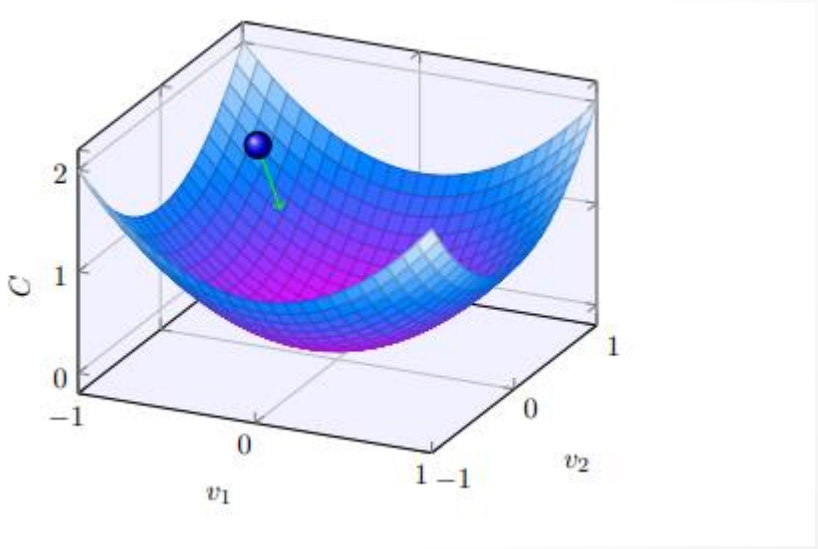
$$v \to v' = v - \eta \nabla C.$$

2.3 梯度下降

- 总结一下：

  梯度下降算法的工作方式就是重复计算梯度∇C，然后沿着**相反**的方向移动

  就像图中小球一样向山谷中滚落



2.3 梯度下降

- 学习速率 η的选择
  - 足够小才能使方程 $\Delta C \approx \nabla C \cdot \Delta v$ 得到很好的近似
  - 太小的话会使△v的变化极小，梯度下降算法就会运行的非常缓慢
  - 真正实现中 η通常是变化的

2.3 梯度下降

- 对于多元函数C(v_1,v_2,v_3,…,v_m)

$$\Delta v = (\Delta v_1, \ldots, \Delta v_m)^T$$

$$\Delta C \approx \nabla C \cdot \Delta v,$$

$$\nabla C \equiv \left(\frac{\partial C}{\partial v_1}, \ldots, \frac{\partial C}{\partial v_m} \right)^T.$$

$$\Delta v = -\eta \nabla C,$$

$$v \to v' = v - \eta \nabla C.$$

2.4 随机梯度下降

- 用权重和偏置代替参数v

$$w_k \to w_k' = w_k - \eta \frac{\partial C}{\partial w_k}$$
$$b_l \to b_l' = b_l - \eta \frac{\partial C}{\partial b_l}.$$

- 代价函数遍历所有样本取均值

$$C = \frac{1}{n}\sum_x C_x$$

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

- 当输入的数据量过大时，训练会非常缓慢
- **随机梯度下降**，随机选取小量训练输入样本计算∇Cx，进而估算梯度 ∇C

2.4 随机梯度下降

- 随机梯度下降通过随机选取小量的 m 个训练输入来工作
- 我们将这些随机的 训练输入标记为 $X_1, X_2, \ldots, X_m$，并把它们称为一个**小批量数据（mini-batch）**

$$\frac{\sum_{j=1}^{m} \nabla C_{X_j}}{m} \approx \frac{\sum_x \nabla C_x}{n} = \nabla C,$$

$$\nabla C \approx \frac{1}{m} \sum_{j=1}^{m} \nabla C_{X_j},$$

$$w_k \rightarrow w_k' = w_k - \frac{\eta}{m} \sum_j \frac{\partial C_{X_j}}{\partial w_k}$$

$$b_l \rightarrow b_l' = b_l - \frac{\eta}{m} \sum_j \frac{\partial C_{X_j}}{\partial b_l},$$

2.4 随机梯度下降

- 训练迭代期（epoch）
- 在线或者递增学习

$$w_k \to w_k' = w_k - \frac{\eta}{m} \sum_j \frac{\partial C_{X_j}}{\partial w_k}$$

$$b_l \to b_l' = b_l - \frac{\eta}{m} \sum_j \frac{\partial C_{X_j}}{\partial b_l},$$

2.5 程序实现

- MNIST
- NumPy

```
class Network(object):

    def __init__(self, sizes):
        self.num_layers = len(sizes)
        self.sizes = sizes
        self.biases = [np.random.randn(y, 1) for y in sizes[1:]]
        self.weights = [np.random.randn(y, x)
                        for x, y in zip(sizes[:-1], sizes[1:])]
```

2.5 程序实现

```
def SGD(self, training_data, epochs, mini_batch_size, eta,
        test_data=None):
    """Train the neural network using mini-batch stochastic
    gradient descent.  The "training_data" is a list of tuples
    "(x, y)" representing the training inputs and the desired
    outputs.  The other non-optional parameters are
    self-explanatory.  If "test_data" is provided then the
    network will be evaluated against the test data after each
    epoch, and partial progress printed out.  This is useful for
    tracking progress, but slows things down substantially."""
    if test_data: n_test = len(test_data)
    n = len(training_data)
    for j in xrange(epochs):
        random.shuffle(training_data)
        mini_batches = [
            training_data[k:k+mini_batch_size]
            for k in xrange(0, n, mini_batch_size)]
        for mini_batch in mini_batches:
            self.update_mini_batch(mini_batch, eta)
        if test_data:
            print "Epoch {0}: {1} / {2}".format(
                j, self.evaluate(test_data), n_test)
        else:
            print "Epoch {0} complete".format(j)
```

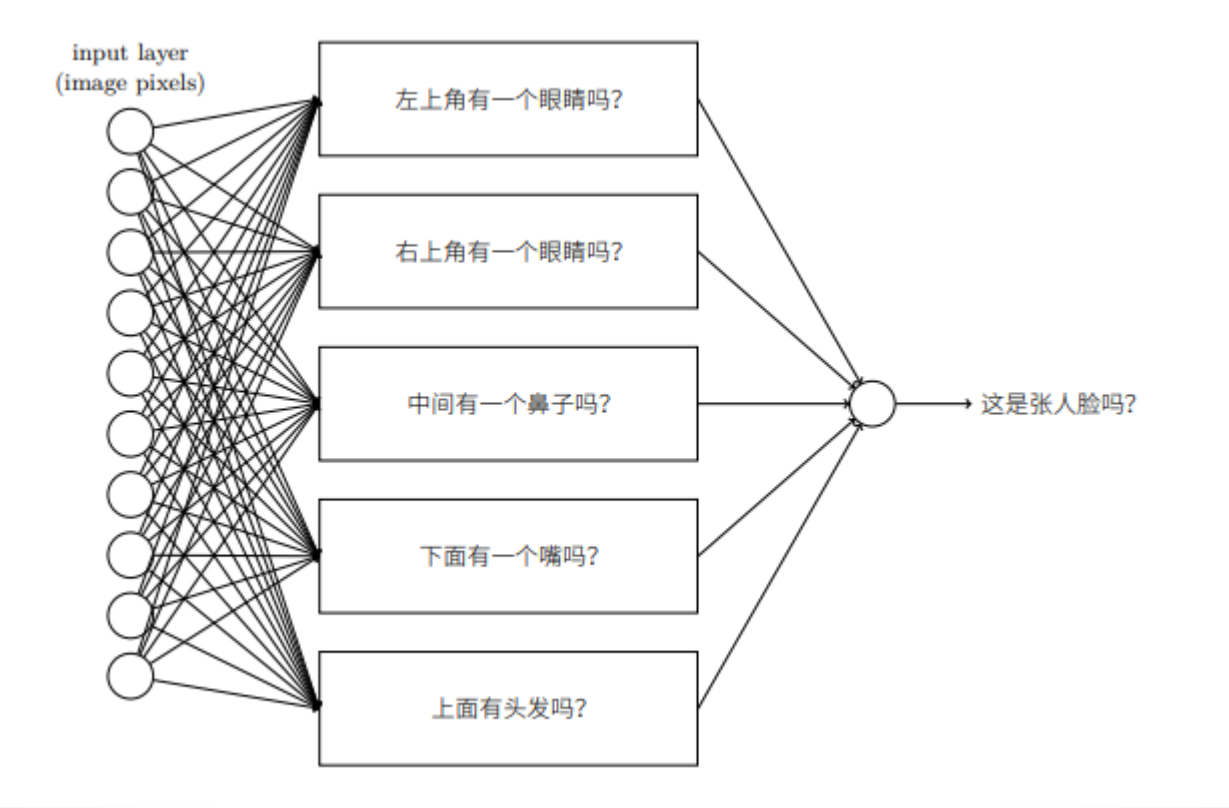
```
def update_mini_batch(self, mini_batch, eta):
    """Update the network's weights and biases by applying
    gradient descent using backpropagation to a single mini batch.
    The "mini_batch" is a list of tuples "(x, y)", and "eta"
    is the learning rate."""
    nabla_b = [np.zeros(b.shape) for b in self.biases]
    nabla_w = [np.zeros(w.shape) for w in self.weights]
    for x, y in mini_batch:
        delta_nabla_b, delta_nabla_w = self.backprop(x, y)
        nabla_b = [nb+dnb for nb, dnb in zip(nabla_b, delta_nabla_b)]
        nabla_w = [nw+dnw for nw, dnw in zip(nabla_w, delta_nabla_w)]
    self.weights = [w-(eta/len(mini_batch))*nw
                    for w, nw in zip(self.weights, nabla_w)]
    self.biases = [b-(eta/len(mini_batch))*nb
                   for b, nb in zip(self.biases, nabla_b)]
```

2.6 迈向深度学习

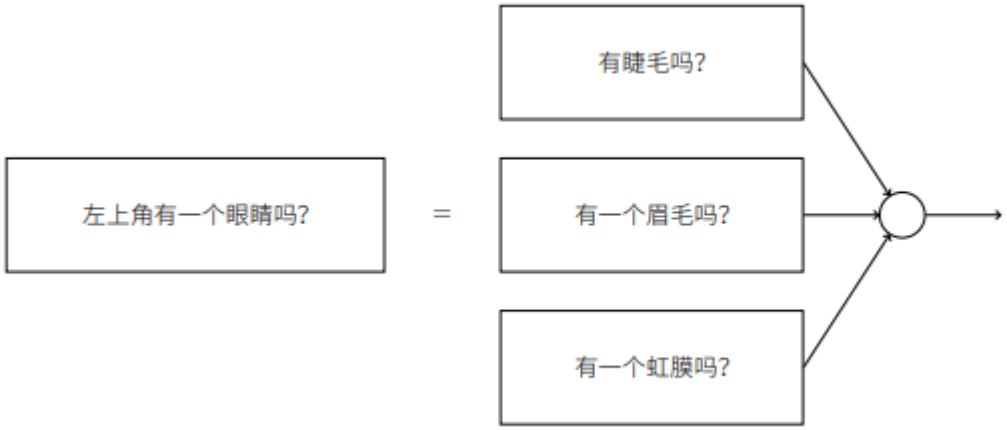
- 复杂的算法 ≤ 简单的学习算法 + 好的训练数据

```
Epoch 0: 9129 / 10000
Epoch 1: 9295 / 10000
Epoch 2: 9348 / 10000
...
Epoch 27: 9528 / 10000
Epoch 28: 9542 / 10000
Epoch 29: 9534 / 10000
```

2.6 迈向深度学习



2.6 迈向深度学习



2.6 迈向深度学习

- 包含这种多层结构 —— 两层或更多隐藏层 —— 的网络被称为**深度神经网络**。

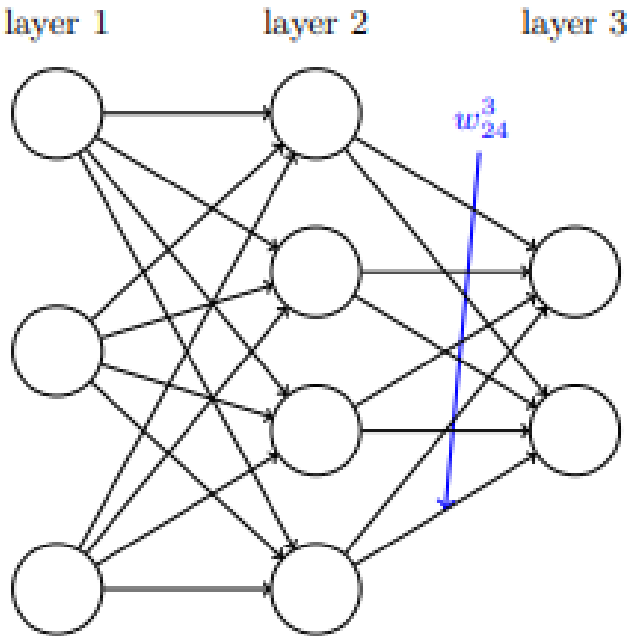
第三节

反向传播

(backpropagation)

3.1神经网络中使用矩阵快速计算输出的方法

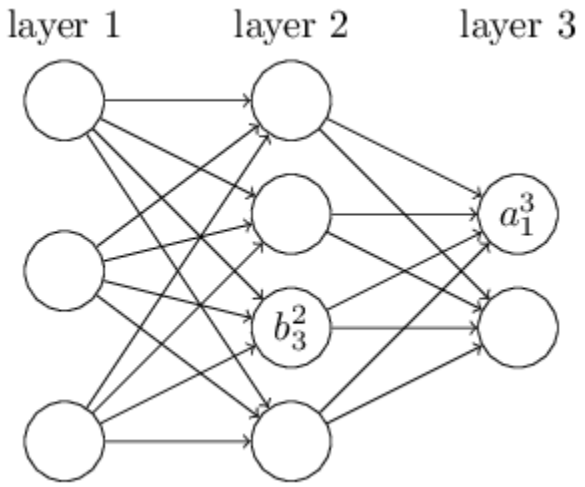
- 权重 w_{jk}^{l}



w_{jk}^{l} 是从 $(l-1)^{\text{th}}$ 层的第 k^{th} 个神经元到 l^{th} 层的第 j^{th} 个神经元的连接上的权重

3.1神经网络中使用矩阵快速计算输出的方法

- 偏置 b_j^l
- 激活值 a_j^l



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

3.1神经网络中使用矩阵快速计算输出的方法

- 权重矩阵 w^l
- 偏置向量 b^l
- 带权输入 z^l
- 激活向量 a^l
- 向量化函数，作用到向量中的每个元素 $f\left(\begin{bmatrix}2\\3\end{bmatrix}\right)=\begin{bmatrix}f(2)\\f(3)\end{bmatrix}=\begin{bmatrix}4\\9\end{bmatrix}$,

$$\begin{bmatrix} w_{0,0} & w_{0,1} & \cdots & w_{0,n} \\ w_{1,0} & w_{1,1} & \cdots & w_{1,n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{k,0} & w_{k,1} & \cdots & w_{k,n} \end{bmatrix}$$

$$a_j^l = \sigma\left(\sum_k w_{jk}^l a_k^{l-1} + b_j^l\right), \qquad a^l = \sigma(w^l a^{l-1} + b^l).$$

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

3.2 关于代价函数的两个假设

- 目标：计算代价函数 C 分别关于w和b的偏导数 ∂C/∂w 和 ∂C/∂b
- 代价函数可以写成每个训练样本x上的代价函数Cx的均值

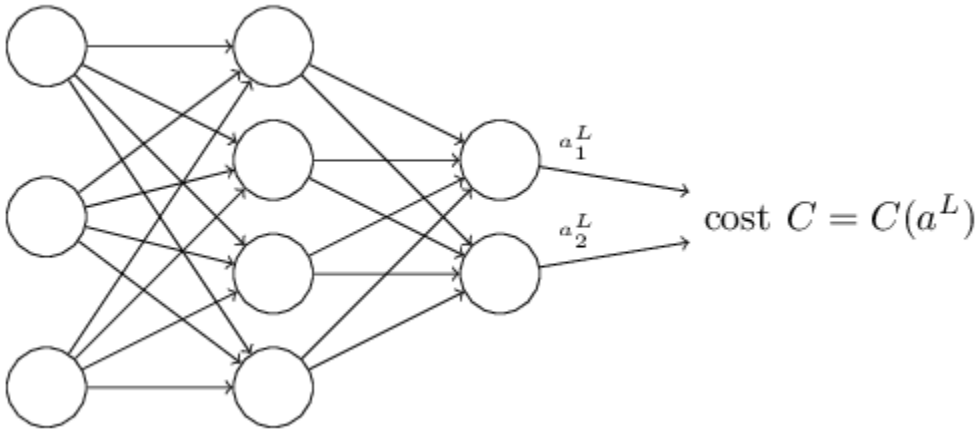
$$C = \frac{1}{2n} \sum_x \|y(x) - a^L(x)\|^2,$$

$$C = \frac{1}{n} \sum_x C_x$$

$$C_x = \frac{1}{2} \|y - a^L\|^2$$

3.2 关于代价函数的两个假设

- 代价可以写成神经网络输出的函数



$$C = \frac{1}{2}\|y - a^L\|^2 = \frac{1}{2}\sum_j (y_j - a_j^L)^2,$$

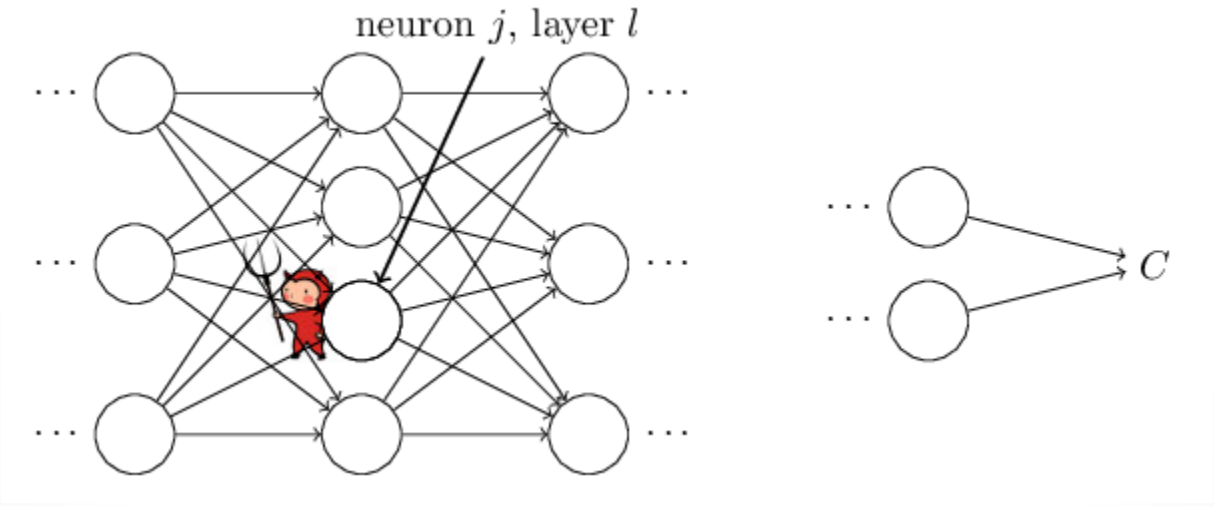
3.3 Hadamard乘积

向量按元素乘积

$$\begin{bmatrix} 1 \\ 2 \end{bmatrix} \odot \begin{bmatrix} 3 \\ 4 \end{bmatrix} = \begin{bmatrix} 1 * 3 \\ 2 * 4 \end{bmatrix} = \begin{bmatrix} 3 \\ 8 \end{bmatrix}.$$

3.4 神经元的误差

- 引入中间量 δ_j^l
- 在l层第j个神经元的误差 $\sigma(z_j^l + \Delta z_j^l)$. $\frac{\partial C}{\partial z_j^l}\Delta z_j^l$.



$$\delta_j^l \equiv \frac{\partial C}{\partial z_j^l}.$$

3.5 反向传播的四个方程

- 1. 输出层误差的方程

$$\delta_j^L = \frac{\partial C}{\partial a_j^L}\sigma'(z_j^L).$$

$$C = \frac{1}{2}\sum_j (y_j - a_j^L)^2$$

$$\partial C/\partial a_j^L = (a_j^L - y_j),$$

$$\delta^L = \nabla_a C \odot \sigma'(z^L).$$

$$\nabla_a C = (a^L - y),$$

$$\delta^L = (a^L - y) \odot \sigma'(z^L).$$

3.5 反向传播的四个方程

- 2.使用下一层的误差 δ^{l+1} 来表示当前层的误差 δ^l（**反向传播**）

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l), \qquad \delta^L = (a^L - y) \odot \sigma'(z^L).$$

3.5 反向传播的四个方程

- 3.代价函数关于网络中任意偏置的改变率

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l.$$

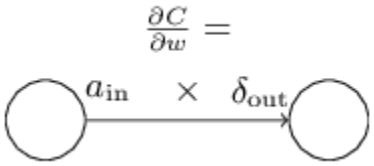
$$\frac{\partial C}{\partial b} = \delta,$$

3.5 反向传播的四个方程

- 代价函数关于任何一个权重的改变率
- 缓慢学习

$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j.$$

$$\frac{\partial C}{\partial w} = a_{\rm in} \delta_{\rm out},$$



3.5 反向传播的四个方程

- 神经元饱和

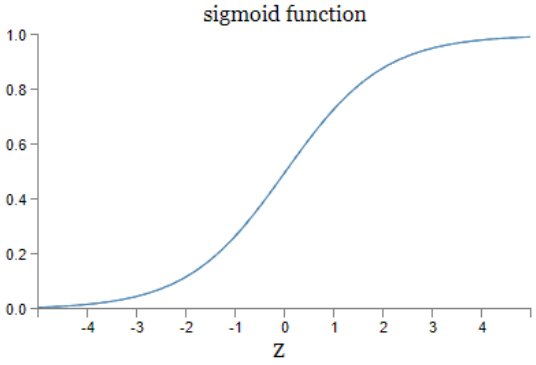
总结：反向传播的四个方程式

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

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

$$\frac{\partial C}{\partial b^l_j} = \delta^l_j \qquad \text{(BP3)}$$

$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j \qquad \text{(BP4)}$$



3.6 四个基本方程的证明

- 1. 输出层误差的计算方程
  - 定义：

$$\delta_j^L = \frac{\partial C}{\partial z_j^L}.$$

$$C = \frac{1}{2}\|y - a^L\|^2 = \frac{1}{2}\sum_j (y_j - a_j^L)^2$$

$$a^l = \sigma(z^l)$$

  - 应用链式法则（见高数教材下册第八章多元函数微分法，第四节多元复合函数的求导法则）得到:

$$\delta_j^L = \sum_k \frac{\partial C}{\partial a_k^L}\frac{\partial a_k^L}{\partial z_j^L},$$

$$\delta_j^L = \frac{\partial C}{\partial a_j^L}\frac{\partial a_j^L}{\partial z_j^L}.$$

$$a_j^L = \sigma(z_j^L)$$

$$\delta_j^L = \frac{\partial C}{\partial a_j^L}\sigma'(z_j^L),$$

3.6 四个基本方程的证明

- 2. 使用下一层的误差表示当前层的误差
  - 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 $\delta_k^{l+1} = \partial C / \partial z_k^{l+1}$ 重写 $\delta_j^l = \partial C / \partial z_j^l$ 我们可以用链式法则：

$$\begin{aligned}\delta_j^l &= \frac{\partial C}{\partial z_j^l} \\ &= \sum_k \frac{\partial C}{\partial z_k^{l+1}} \frac{\partial z_k^{l+1}}{\partial z_j^l} \\ &= \sum_k \frac{\partial z_k^{l+1}}{\partial z_j^l} \delta_k^{l+1},\end{aligned}$$

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

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

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

3.6 四个基本方程的证明

- 最后两个方程BP3和BP4，它们同样遵循链式法则，和前面两个方程的证明相似，这里就不再赘述了。

3.7 算法实现

- 1. **输入x**：为输入层设置对应的激活值a1
- 2. **前向传播**：对每个l = 2,3,...,L计算相应的带权输入 $z^l = w^l a^{l-1} + b^l$ 和激活值 $a^l = \sigma(z^l)$.
- 3. 输出层误差δL：计算输出层误差向量 $\delta^L = \nabla_a C \odot \sigma'(z^L)$.
- 4. 反向误差传播：对每个l = L−1,L−2,...,2，计算 $\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$.
- 5. 输出：代价函数的梯度由 $\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j$ 和 $\frac{\partial C}{\partial b^l_j} = \delta^l_j$ 计算得出

3.7 算法实现

- 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 $a^{x,1}$，并执行下面的步骤：
  - **前向传播**：对每个l = 2,3,....L计算 $z^{x,l} = w^l a^{x,l-1} + b^l$ 和 $a^{x,l} = \sigma(z^{x,l})$
  - **输出误差**：$\delta^{x,L}$ 计算向量 $\delta^{x,L} = \nabla_a C_x \odot \sigma'(z^{x,L})$.
  - **反向传播误差**：对每个l = L−1,L−2,...,2计算 $\delta^{x,l} = ((w^{l+1})^T \delta^{x,l+1}) \odot \sigma'(z^{x,l})$.
- 3.梯度下降：
  - 对每个l = L − 1,L − 2,...,2根据 $w^l \to w^l - \frac{\eta}{m} \sum_x \delta^{x,l} (a^{x,l-1})^T$ 更新权重
  - 根据 $b^l \to b^l - \frac{\eta}{m} \sum_x \delta^{x,l}$ 更新偏置

3.7 算法实现

```
class Network(object):
...
    def update_mini_batch(self, mini_batch, eta):
        """Update the network's weights and biases by applying
        gradient descent using backpropagation to a single mini batch.
        The "mini_batch" is a list of tuples "(x, y)", and "eta"
        is the learning rate."""
        nabla_b = [np.zeros(b.shape) for b in self.biases]
        nabla_w = [np.zeros(w.shape) for w in self.weights]
        for x, y in mini_batch:
            delta_nabla_b, delta_nabla_w = self.backprop(x, y)
            nabla_b = [nb+dnb for nb, dnb in zip(nabla_b, delta_nabla_b)]
            nabla_w = [nw+dnw for nw, dnw in zip(nabla_w, delta_nabla_w)]
        self.weights = [w-(eta/len(mini_batch))*nw
                        for w, nw in zip(self.weights, nabla_w)]
        self.biases = [b-(eta/len(mini_batch))*nb
                       for b, nb in zip(self.biases, nabla_b)]
```

self.backprop(x, y) 实现了反向传播

3.7 算法实现

```
class Network(object):
...
    def backprop(self, x, y):
        """Return a tuple "(nabla_b, nabla_w)" representing the
        gradient for the cost function C_x.  "nabla_b" and
        "nabla_w" are layer-by-layer lists of numpy arrays, similar
        to "self.biases" and "self.weights"."""
        nabla_b = [np.zeros(b.shape) for b in self.biases]
        nabla_w = [np.zeros(w.shape) for w in self.weights]
        # feedforward
        activation = x
        activations = [x] # list to store all the activations, layer by layer
        zs = [] # list to store all the z vectors, layer by layer
        for b, w in zip(self.biases, self.weights):
            z = np.dot(w, activation)+b
            zs.append(z)
            activation = sigmoid(z)
            activations.append(activation)
        # backward pass
        delta = self.cost_derivative(activations[-1], y) * \
            sigmoid_prime(zs[-1])
        nabla_b[-1] = delta
        nabla_w[-1] = np.dot(delta, activations[-2].transpose())
        # Note that the variable l in the loop below is used a little
        # differently to the notation in Chapter 2 of the book.  Here,
        # l = 1 means the last layer of neurons, l = 2 is the
        # second-last layer, and so on.  It's a renumbering of the
        # scheme in the book, used here to take advantage of the fact
        # that Python can use negative indices in lists.
        for l in xrange(2, self.num_layers):
            z = zs[-l]
            sp = sigmoid_prime(z)
            delta = np.dot(self.weights[-l+1].transpose(), delta) * sp
            nabla_b[-l] = delta
            nabla_w[-l] = np.dot(delta, activations[-l-1].transpose())
        return (nabla_b, nabla_w)

...

    def cost_derivative(self, output_activations, y):
        """Return the vector of partial derivatives \partial C_x /
        \partial a for the output activations."""
        return (output_activations-y)

def sigmoid(z):
    """The sigmoid function."""
    return 1.0/(1.0+np.exp(-z))

def sigmoid_prime(z):
    """Derivative of the sigmoid function."""
    return sigmoid(z)*(1-sigmoid(z))
```

第四节

总结

4.1 训练模型

- 1.输入训练样本的集合 (m个样本)
- 2.对每个训练样本x：设置对应的输入激活 $a^{x,1}$，并执行下面的步骤：
  - **前向传播**：对每个l = 2,3,. . l 计算 $z^{x,l} = w^l a^{x,l-1} + b^l$ 和 $a^{x,l} = \sigma(z^{x,l})$
  - **输出误差**：$\delta^{x,L}$ 计算向量 $\delta^{x,L} = \nabla_a C_x \odot \sigma'(z^{x,L})$.
  - **反向传播误差**：对每个l = L−1,L−2,...,2计算 $\delta^{x,l} = ((w^{l+1})^T \delta^{x,l+1}) \odot \sigma'(z^{x,l})$.
- 3.梯度下降：
  - 对每个l = L − 1,L − 2,....2根据 $w^l \to w^l - \frac{\eta}{m} \sum_x \delta^{x,l} (a^{x,l-1})^T$ 更新权重
  - 根据 $b^l \to b^l - \frac{\eta}{m} \sum_x \delta^{x,l}$ 更新偏置
- 4.这波样本完了之后再换一波新的样本继续上述操作，直到所有样本被计算完，这样算是完成了一个训练周期。
- 5.重复若干次训练周期，直至代价函数降到接近全局最小值。
- 6.保存当前神经网络的权重值和偏置值到一个文件（**训练结果**）

4.2 使用模型识别数字

- 1. 从文件中加载上面训练出的模型（神经网络的权重和偏置）
- 2. 输入要识别的图片（转换至28x28灰度图，再转换为输入向量）
- 3. 前向传播（ feedforward ）即通过神经网络一层一层的计算下去
- 4. 得到输出向量（10个元素），遍历10个元素找出最大值（即是神经网络认为该图像最接近的结果）

4.3 参考文献

- 原书地址：
  - http://neuralnetworksanddeeplearning.com/
- 反向传播论文：
  - http://www.nature.com/nature/journal/v323/n6088/pdf/323533a0.pdf
- Youtube视频：
  - https://www.youtube.com/watch?v=aircAruvnKk&t=609s
- 源码：
  - https://github.com/mnielsen/neural-networks-and-deep-learning.git
- 通过Django部署到Webserver的源码：
  - https://github.com/yish0000/neuralnetwork.git

祖龙娱乐
LOONG ENTERTAINMENT

Q & A

祖龙娱乐
LOONG ENTERTAINMENT

Thank you