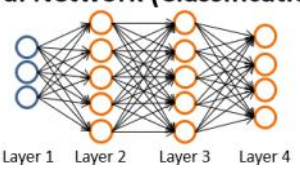
# 9 神经网络的学习

2022年11月5日　　10:29

## 9.1代价函数

为了更好地理解如何求取神经网络中的参数，将从分类问题讲起。

**Neural Network (Classification)**




$\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(m)}, y^{(m)})\}$

$L =$ total no. of layers in network

$s_l =$ no. of units (not counting bias unit) in layer $l$

如图所示为一个神经网络的结构示意图，我们用L表示神经网络的层数，用$s_l$表示第1层有多少个单元，其中不包括第1层的偏差单元。

Binary classification

$y = 0$ or $1$

1 output unit

Multi-class classification (K classes)

$y \in \mathbb{R}^K$ E.g. $\begin{bmatrix}1\\0\\0\\0\end{bmatrix}, \begin{bmatrix}0\\1\\0\\0\end{bmatrix}, \begin{bmatrix}0\\0\\1\\0\end{bmatrix}, \begin{bmatrix}0\\0\\0\\1\end{bmatrix}$

pedestrian car motorcycle truck

K output units

如果我们想要分K类，那么输出的结果应为由二进制数组成的K维向量，而且其中的K大于等于3，因为只有当分类数量足够多时才需要使用这种一对多的方法。如果K为2，那么只用一个输出单元就足够了。

逻辑回归中的代价函数为之前线性回归问题里代价函数的一般形式：

Logistic regression:

$$J(\theta) = -\frac{1}{m}\left[\sum_{i=1}^{m} y^{(i)} \log h_\theta(x^{(i)}) + (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)}))\right] + \frac{\lambda}{2m}\sum_{j=1}^{n} \theta_j^2$$

而神经网络的代价函数为逻辑回归的代价函数的进一步演化：

Neural network:

$$h_\Theta(x) \in \mathbb{R}^K \quad (h_\Theta(x))_i = i^{th} \text{ output}$$

$$J(\Theta) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log((h_\Theta(x^{(i)}))_k + (1 - y_k^{(i)}) \log(1 - (h_\Theta(x^{(i)}))_k)\right]$$
$$+\frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(\Theta_{ji}^{(l)})^2$$

## 9.2反向传播算法

**Gradient computation**

$$\rightarrow J(\Theta) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log h_\theta(x^{(i)})_k + (1 - y_k^{(i)}) \log(1 - h_\theta(x^{(i)})_k)\right]$$
$$+\frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(\Theta_j^{(l)})^2$$

$\rightarrow \min_\Theta J(\Theta)$

Need code to compute:

$\rightarrow$ - $J(\Theta)$

$\rightarrow$ - $\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta)$ ←　　$\Theta_{ij}^{(l)} \in \mathbb{R}$

如果想要求得一个好的参数 θ ，在之前的回归问题中，可以使用梯度下降或其他高级

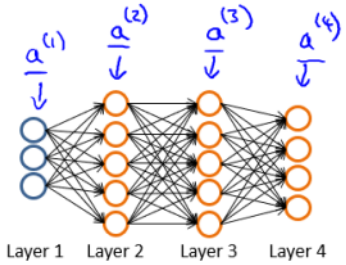
的算法，其目的是为了求出代价函数的值和其偏导数，以找到使代价函数取得最小的参数值。

以一个只有一个实数对的数据集为例帮助我们理解前向传播：

Given one training example $(x, y)$:

Forward propagation:

$$a^{(1)} = x$$
$$z^{(2)} = \Theta^{(1)} a^{(1)}$$
$$a^{(2)} = g(z^{(2)}) \quad (\text{add } a_0^{(2)})$$
$$z^{(3)} = \Theta^{(2)} a^{(2)}$$
$$a^{(3)} = g(z^{(3)}) \quad (\text{add } a_0^{(3)})$$
$$z^{(4)} = \Theta^{(3)} a^{(3)}$$
$$a^{(4)} = h_\Theta(x) = g(z^{(4)})$$



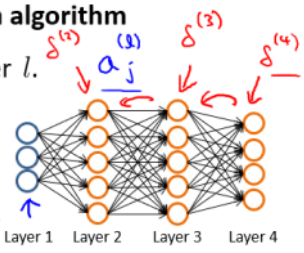

这个数据对通过三个激活函数后到达输出层。接下来，为了计算导数项，我们需要采用一种反向传播(Backpropagation)的算法。

**Gradient computation: Backpropagation algorithm**

Intuition: $\delta_j^{(l)}$ = "error" of node $j$ in layer $l$.

For each output unit (layer L = 4)

$$\delta_j^{(4)} = a_j^{(4)} - y_j$$




首先，定义第l层的第j个节点的误差为 δ ，那么可以用输出节点的因子和标签值将其表示出来。根据向量化的原理，从而可以将前几层的误差项也表达出来：

$$\delta^{(3)} = (\Theta^{(3)})^T \delta^{(4)} .* g'(z^{(3)})$$
$$\delta^{(2)} = (\Theta^{(2)})^T \delta^{(3)} .* g'(z^{(2)})$$

反向传播的核心就是我们由输出层的误差向量不断向前求取每一层误差项的过程。从而利用反向传播，可以更好地求取代价函数的偏导数项。

反向传播的实现算法：

**Backpropagation algorithm**

Training set $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$

Set $\Delta_{ij}^{(l)} = 0$ (for all $l, i, j$).

For $i = 1$ to $m$

Set $a^{(1)} = x^{(i)}$

Perform forward propagation to compute $a^{(l)}$ for $l = 2, 3, \ldots, L$

Using $y^{(i)}$, compute $\delta^{(L)} = a^{(L)} - y^{(i)}$

Compute $\delta^{(L-1)}, \delta^{(L-2)}, \ldots, \delta^{(2)}$

$\Delta_{ij}^{(l)} := \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$

- 得到训练集
- 设置 Δ 作为误差的累计
- 逐一输入数据集中的实数对
- 令数据集中的x为输入因子
- 输入因子经过逐层的前向传播获得输出因子
- 根据输出因子和数据集中标签的差值得到误差
- 误差向前传播得到前面除输出层之外每一层的误差
- 对每一层误差进行累计，储存在 Δ 中

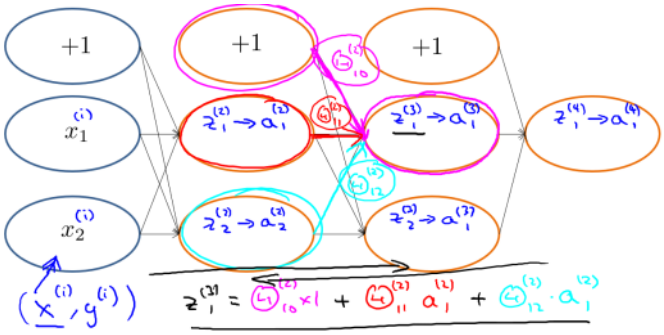
加上正则化因子后，计算出一个D项：

$$D_{ij}^{(l)} := \frac{1}{m}\triangle_{ij}^{(l)} + \lambda\Theta_{ij}^{(l)} \quad \text{if } j \neq 0$$
$$D_{ij}^{(l)} := \frac{1}{m}\triangle_{ij}^{(l)} \quad \text{if } j = 0$$

根据D即可计算出代价函数的导数项：

$$\frac{\partial}{\partial\Theta_{ij}^{(l)}}J(\Theta) = D_{ij}^{(l)}$$

## 9.3反向传播的理解




首先回顾前向传播的过程。

$$J(\Theta) = -\frac{1}{m}\left[\sum_{i=1}^{m} y^{(i)}\log(h_\Theta(x^{(i)})) + (1-y^{(i)})\log(1-(h_\Theta(x^{(i)})))\right]$$
$$+\frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(\Theta_{ji}^{(l)})^2$$

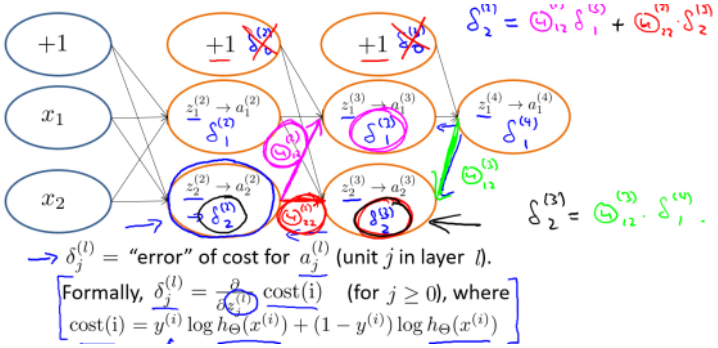
Focusing on a single example $x^{(i)}$, $y^{(i)}$, the case of 1 output unit, and ignoring regularization ($\lambda = 0$),

$$\text{cost(i)} = y^{(i)}\log h_\Theta(x^{(i)}) + (1-y^{(i)})\log h_\Theta(x^{(i)})$$

(Think of $\text{cost(i)} \approx (h_\Theta(x^{(i)}) - y^{(i)})^2$)

I.e. how well is the network doing on example i?

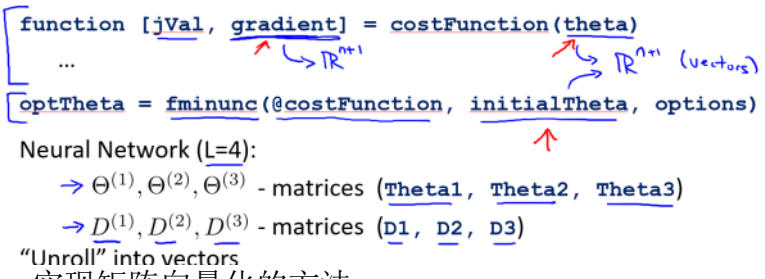
对于一个只有一个输出的代价函数，我们忽略其正则项，那么对于第i个标签对，其代价值可以看作是假设函数和标签的方差。



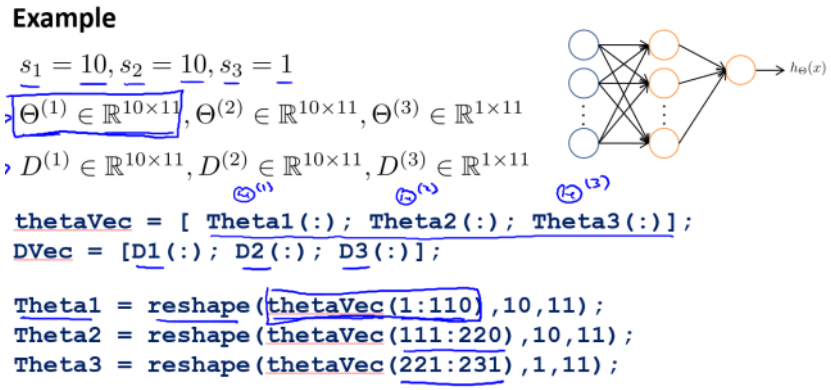

反向传播过程的核心就是在计算误差δ，而从微积分的角度来看，误差就是代价对加权和的求导。它们衡量的是为了影响这些中间值，我们想要改变的神经网络中的权重的程度，进而影响整个假设函数的输出和代价函数的值。

## 9.4使用注意：展开参数

不同于逻辑回归，使用神经网络的参数不再是向量而是矩阵了，那么如何将其展开成向量以便使其称为恰当的格式传入函数中。




对于Matlab来说，实现矩阵向量化的方法：



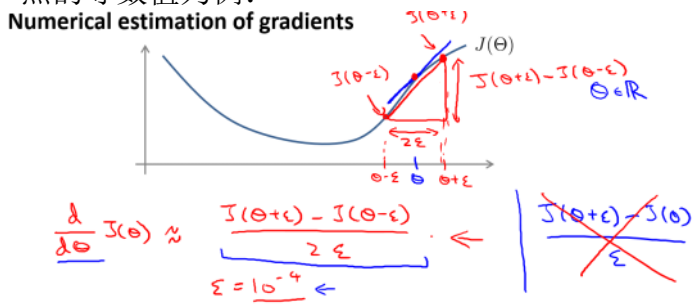

就是把所有参数矩阵里的值全部取出，展开成长向量。用reshape命令也可以进行反向的转化。

## 9.5梯度检验

虽然对神经网络使用梯度下降的方法进行求取使代价函数最小的参数值时很顺利，而且代价值也确实随着迭代在减小，但是这其中可能会隐藏一些bug不被我们察觉，而且其误差相对无bug的情况是巨大的。如何解决这一问题？

答案是用梯度检验可以验证所求取的代价函数的导数的准确性。

以求取一条曲线在一点的导数值为例：




Implement: `gradApprox = (J(theta + EPSILON) - J(theta - EPSILON)) / (2*EPSILON)`

众所周知，曲线在一点的导数为其在该点的切线的斜率。假设一个很小的数 ε ，那么可以在该点的左右取该距离的点的对应曲线点的连线的斜率作为近似。而且，从数学上可以表明，当 ε 足够小时，其斜率值就是曲线在这一点的导数。相比于单侧差分，双侧差分能得到更加准确的结果。

而当参数为向量时，可以用类似的思想，求出代价函数在所有点处的偏导：

$\theta \in \mathbb{R}^n$ (E.g. $\theta$ is "unrolled" version of $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$)

$\theta = [\theta_1, \theta_2, \theta_3, \ldots, \theta_n]$

$$\frac{\partial}{\partial \theta_1} J(\theta) \approx \frac{J(\theta_1+\epsilon, \theta_2, \theta_3, \ldots, \theta_n) - J(\theta_1-\epsilon, \theta_2, \theta_3, \ldots, \theta_n)}{2\epsilon}$$

$$\frac{\partial}{\partial \theta_2} J(\theta) \approx \frac{J(\theta_1, \theta_2+\epsilon, \theta_3, \ldots, \theta_n) - J(\theta_1, \theta_2-\epsilon, \theta_3, \ldots, \theta_n)}{2\epsilon}$$

$$\vdots$$

$$\frac{\partial}{\partial \theta_n} J(\theta) \approx \frac{J(\theta_1, \theta_2, \theta_3, \ldots, \theta_n+\epsilon) - J(\theta_1, \theta_2, \theta_3, \ldots, \theta_n-\epsilon)}{2\epsilon}$$

其代码的具体实现为：

```
for i = 1:n,
  thetaPlus = theta;
  thetaPlus(i) = thetaPlus(i) + EPSILON;
  thetaMinus = theta;
  thetaMinus(i) = thetaMinus(i) - EPSILON;
  gradApprox(i) = (J(thetaPlus) - J(thetaMinus))
                  /(2*EPSILON);
end;
```

Check that gradApprox ≈ DVec

From backprop.

如果近似解和我们用梯度下降计算出来的导数值非常近似，可以认为反向传播是正确的。

那么，整个反向传播的梯度检验的过程如下：

**Implementation Note:**

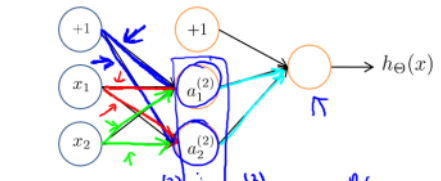
- Implement backprop to compute DVec (unrolled $D^{(1)}, D^{(2)}, D^{(3)}$).
- Implement numerical gradient check to compute gradApprox.
- Make sure they give similar values.
- Turn off gradient checking. Using backprop code for learning.

- 首先，计算出导数值
- 实现数值上的梯度检验，计算出导数的近似结果
- 确保检验值和计算值近似
- 最后，在使用神经网络学习之前，关掉梯度检验。因为梯度检验非常复杂、非常慢，所以一旦确定反向传播的正确实现，应当将梯度检验关掉。

## 9.6随机初始化

当我们运行梯度下降算法或其他高级的优化算法时，需要对参数设置一个初始值。如何设置初始值呢？

在逻辑回归中，我们把参数的初始值设置为0是行得通的，但是在训练神经网络时，这种设置就不那么明智。




$\Theta_{ij}^{(l)} = 0$ for all $i, j, l$.

After each update, parameters corresponding to inputs going into each of two hidden units are identical.

当把初始值都设为0，会使后续更新的每一个节点对应的几个参数值都相等。这阻止了

神经网络特性的发挥。因而，在初始化参数时需要采用随机的思想。

Initialize each $\Theta_{ij}^{(l)}$ to a random value in $[-\epsilon, \epsilon]$
(i.e. $-\epsilon \leq \Theta_{ij}^{(l)} \leq \epsilon$)

E.g.

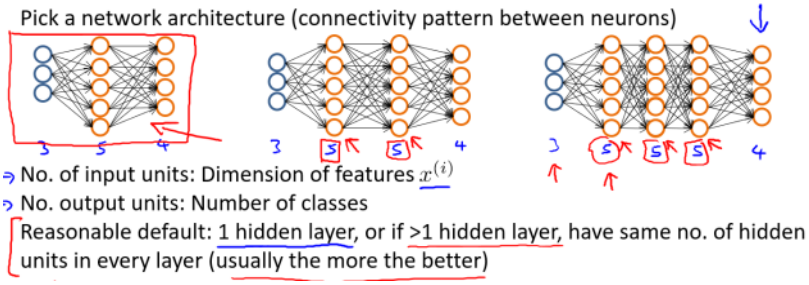
```
Theta1 = rand(10,11)*(2*INIT_EPSILON)
         - INIT_EPSILON;

Theta2 = rand(1,11)*(2*INIT_EPSILON)
         - INIT_EPSILON;
```

将参数划定在一个区间内即可。

## 9.7组合到一起

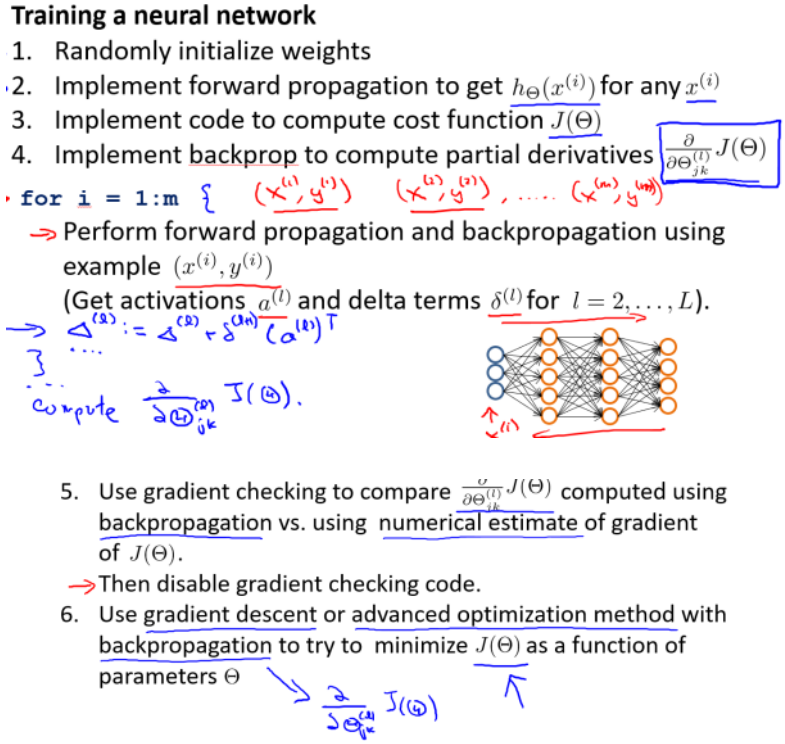
接下来做一个总体实现的回顾。

1. 选择一种神经网络架构




输入单元的数量为特征量的多少；输出单元的个数由所划分类别决定。再次强调，多元分类的个数大于2时，用向量来表示。而对于隐藏层的单元个数和层数，一般默认为只使用单个隐藏层；或者是多于一个隐藏层，但每一个隐藏层的单元个数是相等的。一般来说，隐藏单元越多越好，且和输入特征数量相匹配，比如1倍、2倍、3倍等。

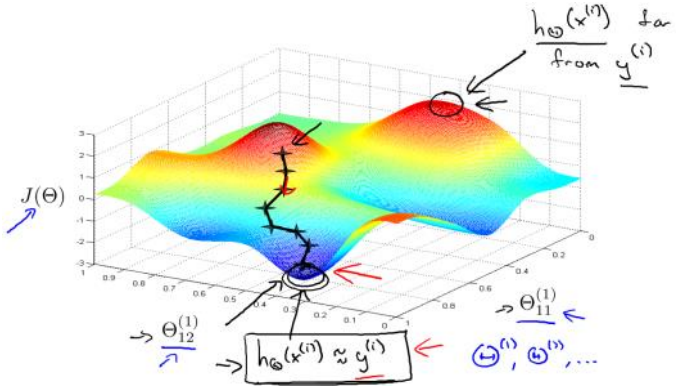
2. 神经网络的实现




- 随机初始化参数
- 对任意输入，前向传播得到假设函数的输出值
- 计算代价函数值
- 反向传播计算代价函数的偏导数(老师推荐我们在第一次实现反向传播算法时使

用for循环，而不要使用一些很高级的向量化方法)

- 使用梯度检验来检查偏导数值是否准确、反向传播算法是否正确
- 禁用梯度检验
- 使用一个最优化方法，如梯度下降算法或其他更高级的优化算法(LBFGS算法、共轭梯度法等)和反向传播算法相结合最小化参数(众所周知，这些算法可能会使代价函数下降到局部最小值，不能保证一定会得到全局最优值，但是这些算法在实际的使用过程中都是不错的)




## 9.8无人驾驶

使用神经网络可以实现自动驾驶。