九、神经网络的学习(Neural Networks: Learning)

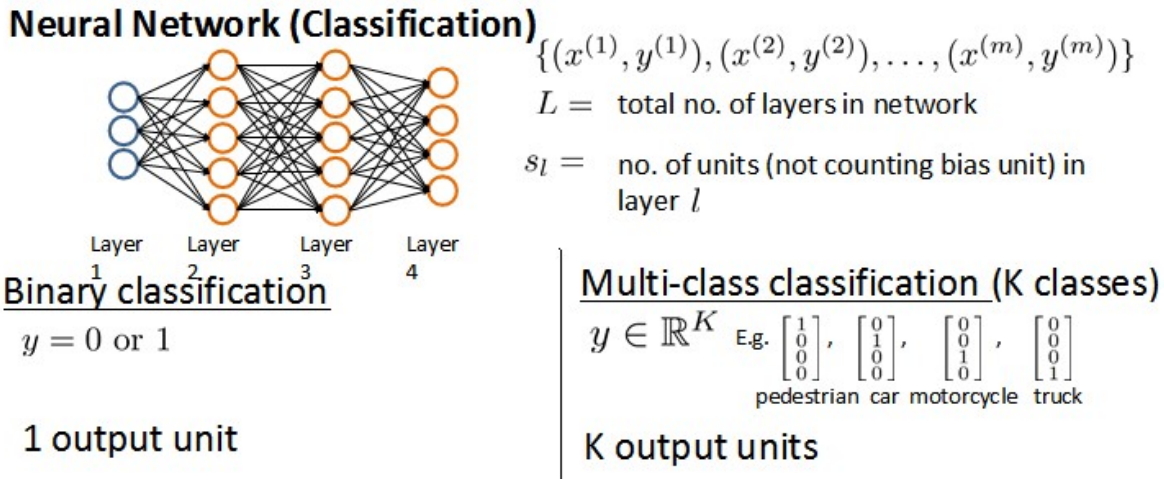
9.1 代价函数

参数

- 假设神经网络的训练样本有m个，
- 每个包含一组输入x和一组输出信号y，
- L表示神经网络层数，
- S_I表示每层的**neuron**个数(S_l表示输出层神经元个数)，
- S_L代表最后一层中处理单元的个数。

  分类

  将神经网络的分类定义为两种情况：二类分类和多类分类，
- 二类分类：$S_L = 0, y = 0\ or\ 1$表示哪一类；
- K类分类：$S_L = k, y_i = 1$表示分到第i类；$(k > 2)$



我们回顾逻辑回归问题中我们的代价函数为：

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

在逻辑回归中，我们只有一个输出变量，又称标量（**scalar**），也只有一个因变量y，但是在神经网络中，我们可以有很多输出变量，我们的$h_\theta(x)$是一个维度为K的向量，并且我们训练集中的因变量也是同样维度的一个向量，因此我们的代价函数会比逻辑回归更加复杂一些，为：

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

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

这个看起来复杂很多的代价函数背后的思想还是一样的，我们希望通过代价函数来观察算法预测的结果与真实情况的误差有多大，唯一不同的是，对于每一行特征，我们都会给出K个预测，基本上我们可以利用循环，对每一行特征都预测K个不同结果，然后在利用循环在K个预测中选择可能性最高的一个，将其与y中的实际数据进行比较。

正则化的那一项只是排除了每一层θ_0后，每一层的θ 矩阵的和。最里层的循环j循环所有的行（由s_{l+1} 层的激活单元数决定），循环i则循环所有的列，由该层（s_l层）的激活单元数所决定。即：$h_\theta(x)$与真实值之间的距离为每个样本-每个类输出的加和，对参数进行**regularization**的**bias**项处理所有参数的平方和。

9.2 反向传播算法

之前我们在计算神经网络预测结果的时候我们采用了一种正向传播方法，我们从第一层开始正向一层一层进行计算，直到最后一层的$h_\theta(x)$。

现在，为了计算代价函数的偏导数$\frac{\partial}{\partial\Theta_{ij}^{(l)}}J(\Theta)$，我们需要采用一种反向传播算法，也就是**首先计算最后一层的误差，然后再一层一层反向求出各层的误差，直到倒数第二层。**
以一个例子来说明反向传播算法。

假设我们的训练集只有一个样本$\left(x^{(1)},y^{(1)}\right)$，我们的神经网络是一个四层的神经网络，其中$K=4$，$S_L=4$，$L=4$:

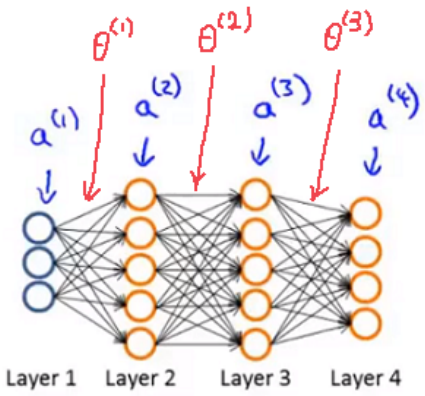
前向传播算法：

Gradient computation

Given one training example (x, y):

Forward propagation:

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



下面的公式推导过程见：https://blog.csdn.net/qq_29762941/article/details/80343185

- 我们从最后一层的误差开始计算，误差是激活单元的预测（$a^{(4)}$）与实际值（y^k）之间的误差，（$k=1:k$）。我们用δ来表示误差，则：$\delta^{(4)}=a^{(4)}-y$
- 我们利用这个误差值来计算前一层的误差：$\delta^{(3)}=\left(\Theta^{(3)}\right)^T\delta^{(4)}*g'\left(z^{(3)}\right)$
  其中$g'(z^{(3)})$是S形函数的导数，$g'(z^{(3)})=a^{(3)}*(1-a^{(3)})$。而$(\theta^{(3)})^T\delta^{(4)}$则是权重导致的误差的和。
- 下一步是继续计算第二层的误差：$\delta^{(2)}=(\Theta^{(2)})^T\delta^{(3)}*g'(z^{(2)})$
- 因为第一层是输入变量，不存在误差。我们有了所有的误差的表达式后，便可以计算代价函数的偏导数了，假设$\lambda=0$，即我们不做任何正则化处理时有：$\frac{\partial}{\partial\Theta_{ij}^{(l)}}J(\Theta)=a_j^{(l)}\delta_i^{l+1}$

重要的是清楚地知道上面式子中上下标的含义：
l 代表目前所计算的是第几层。
j 代表目前计算层中的激活单元的下标，也将是下一层的第j个输入变量的下标。
i 代表下一层中误差单元的下标，是受到权重矩阵中第i行影响的下一层中的误差单元的下标。

如果我们考虑正则化处理，并且我们的训练集是一个特征矩阵而非向量。在上面的特殊情况中，我们需要计算每一层的误差单元来计算代价函数的偏导数。在更为一般的情况中，我们同样需要计算每一层的误差单元，但是我们需要为整个训练集计算误差单元，此时的误差单元也是一个矩阵，我们用$\Delta_{ij}^{(l)}$来表示这个误差矩阵。第 l 层的第 i 个激活单元受到第 j 个参数影响而导致的误差。

我们的算法表示为：

$$
\begin{aligned}
&for\ i{=}1{:}m\ \ \{\\
&\quad set\ a^{(i)}{=}x^{(i)}\\
&\quad perform\ foward\ propagation\ to\ compute\ a^{(l)}\ for\ l{=}1,2,3...L\\
&\quad Using\ \delta^{(L)}{=}a^{(L)}{-}y^{i}\\
&\quad perform\ back\ propagation\ to\ compute\ all\ previous\ layer\ error\ vector\\
&\quad \Delta_{ij}^{(l)}{:=}\Delta_{ij}^{(l)}{+}a_j^{(l)}\delta_i^{l+1}\\
&\qquad \}
\end{aligned}
$$

即首先用正向传播方法计算出每一层的激活单元，利用训练集的结果与神经网络预测的结果求出最后一层的误差，然后利用该误差运用反向传播法计算出直至第二层的所有误差。

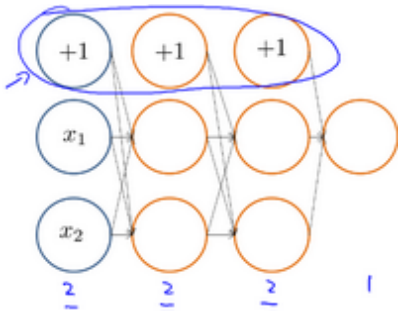
在求出了$\Delta_{ij}^{(l)}$之后，我们便可以计算代价函数的偏导数了，计算方法如下：

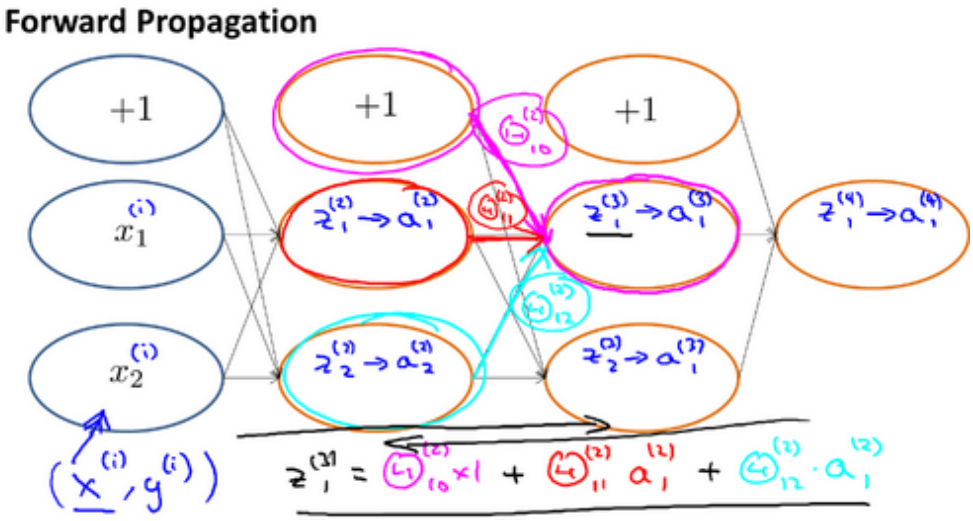
$$D_{ij}^{(l)} := \frac{1}{m}\Delta_{ij}^{(l)} + \lambda\Theta_{ij}^{(l)} \qquad if\ j \neq 0$$

$$D_{ij}^{(l)} := \frac{1}{m}\Delta_{ij}^{(l)} \qquad if\ j = 0$$

9.3 反向传播算法的直观理解

前向传播算法：





反向传播算法做的是：

What is backpropagation doing?

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

$(x^{(i)}, y^{(i)})$

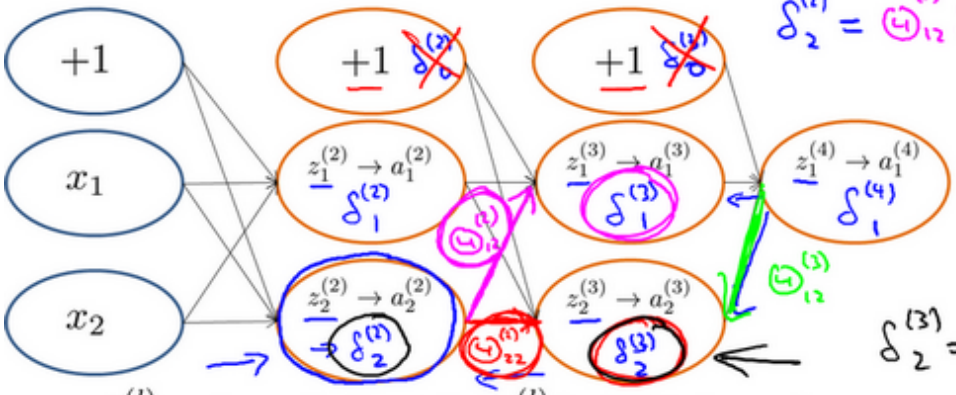
Focusing on a single example $x^{(i)}$, $y^{(i)}$, the case of 1 output unit, and ignoring regularization ($\lambda = 0$),

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

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

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

Forward Propagation



$\delta_1^{(4)} = y^{(i)} - a_1^{(4)}$

$\delta_2^{(3)} = \Theta_{12}^{(3)} \delta_1^{(4)} + \Theta_{22}^{(3)} \cdot \delta_2^{(4)}$

$\delta_2^{(3)} = \Theta_{12}^{(3)} \cdot \delta_1^{(4)} \cdot$

$\delta_j^{(l)}$ = "error" of cost for $a_j^{(l)}$ (unit j in layer l).

Formally, $\delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}} \text{cost(i)}$ (for $j \geq 0$), where $\text{cost(i)} = y^{(i)} \log h_\Theta(x^{(i)}) + (1-y^{(i)}) \log h_\Theta(x^{(i)})$

感悟：上图中的 $\delta_j^{(l)} = "error" of cost\ for\ a_j^{(l)}\ (unit\ j\ in\ layer\ l)$ 理解如下：

$\delta_j^{(l)}$ 相当于是第 l 层的第 j 单元中得到的激活项的"误差"，即"正确"的 $a_j^{(l)}$ 与计算得到的 $a_j^{(l)}$ 的差。

而 $a_j^{(l)} = g(z^{(l)})$，（g为sigmoid函数）。我们可以想象 $\delta_j^{(l)}$ 为函数求导时迈出的那一丁点微分，所以更准确的说 $\delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}} cost(i)$

9.4 实现注意：展开参数

Example

$s_1 = 10, s_2 = 10, s_3 = 1$

$\Theta^{(1)} \in \mathbb{R}^{10\times 11}, \Theta^{(2)} \in \mathbb{R}^{10\times 11}, \Theta^{(3)} \in \mathbb{R}^{1\times 11}$

$D^{(1)} \in \mathbb{R}^{10\times 11}, D^{(2)} \in \mathbb{R}^{10\times 11}, D^{(3)} \in \mathbb{R}^{1\times 11}$


```
thetaVec = [ Theta1(:); Theta2(:); Theta3(:)];
DVec = [D1(:); D2(:); D3(:)];

Theta1 = reshape(thetaVec(1:110),10,11);
Theta2 = reshape(thetaVec(111:220),10,11);
Theta3 = reshape(thetaVec(221:231),1,11);
```

Learning Algorithm

Have initial parameters $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$.

Unroll to get `initialTheta` to pass to

```
fminunc(@costFunction, initialTheta, options)
```

```
function [jval, gradientVec] = costFunction(thetaVec)
```

From `thetaVec`, get $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$. reshape

Use forward prop/back prop to compute $D^{(1)}, D^{(2)}, D^{(3)}$ and $J(\Theta)$.

Unroll $D^{(1)}, D^{(2)}, D^{(3)}$ to get `gradientVec`.

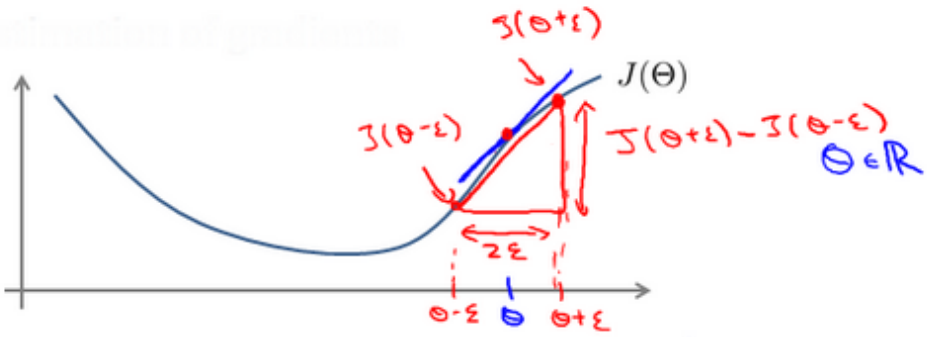
9.5 梯度检验

当我们对一个较为复杂的模型（例如神经网络）使用梯度下降算法时，可能会存在一些不容易察觉的错误，意味着，虽然代价看上去在不断减小，但最终的结果可能并不是最优解。

为了避免这样的问题，我们采取一种叫做梯度的数值检验（**Numerical Gradient Checking**）方法。这种方法的思想是通过**估计梯度值来检验我们计算的导数值是否真的是我们要求的。**

对梯度的估计采用的方法

在代价函数上沿着切线的方向选择离两个非常近的点然后计算两个点的平均值用以估计梯度。即对于某个特定的 θ，我们计算出在 θ-ε 处和 θ+ε 的代价值（ε是一个非常小的值，通常选取 0.001），然后求两个代价的平均，用以估计在 θ 处的代价值。



当θ是一个向量时，我们则需要对偏导数进行检验。因为代价函数的偏导数检验只针对一个参数的改变进行检验，下面是一个只针对θ_1进行检验的示例：

$$\frac{\partial}{\partial\theta_1} = \frac{J(\theta_1+\varepsilon_1,\theta_2,\theta_3...\theta_n)-J(\theta_1-\varepsilon_1,\theta_2,\theta_3...\theta_n)}{2\varepsilon}$$

最后我们还需要对通过反向传播方法计算出的偏导数进行检验。

根据上面的算法，计算出的偏导数存储在矩阵 $D_{ij}^{(l)}$ 中。检验时，我们要将该矩阵展开成为向量，同时我们也将 θ 矩阵展开为向量，我们针对每一个 θ 都计算一个近似的梯度值，将这些值存储于一个近似梯度矩阵中，最终将得出的这个矩阵同 $D_{ij}^{(l)}$ 进行比较。

9.6 随机初始化

任何优化算法都需要一些初始的参数。到目前为止我们都是初始所有参数为0，这样的初始方法对于逻辑回归来说是可行的，但是对于神经网络来说是不可行的。如果我们令所有的初始参数都为0，这将意味着我们第二层的所有激活单元都会有相同的值。同理，如果我们初始所有的参数都为一个非0的数，结果也是一样的。

我们通常初始参数为正负ε之间的随机值，假设我们要随机初始一个尺寸为10×11的参数矩阵，代码如下：

```
Theta1 = rand(10, 11) * (2*eps) – eps
```

9.7 综合起来

小结一下使用神经网络时的步骤：

网络结构：第一件要做的事是选择网络结构，即决定选择多少层以及决定每层分别有多少个单元。

第一层的单元数即我们训练集的特征数量。

最后一层的单元数是我们训练集的结果的类的数量。

如果隐藏层数大于1，确保每个隐藏层的单元个数相同，通常情况下隐藏层单元的个数越多越好。

我们真正要决定的是隐藏层的层数和每个中间层的单元数。

训练神经网络：

1. 参数的随机初始化
2. 利用正向传播方法计算所有的$h_\theta(x)$
3. 编写计算代价函数 J 的代码
4. 利用反向传播方法计算所有偏导数
5. 利用数值检验方法检验这些偏导数
6. 使用优化算法来最小化代价函数