Week 01

(摸了)

Week 02

Cost Function(代价函数)

对于一组训练集, 我们给出了一个对参数theta的预测值
但预测得到底准不准? 我们需要另外设计一个代价函数$J(\vec{\theta})$用于评估准确性,
评估的依据正是测试集里的(全体)数据.
注意: CostFunction是关于自变量 $\vec{\theta}$ 的函数!

Gradient Descent Algorithm

目标: 让`代价函数`达到极小值, 最好是全局最小值.(所以我们得预先设计用何种代价函数!)

思路: 由于函数沿梯度的反方向下降速度最快,
我们通过多次迭代, 每次迭代都沿着当前位置的梯度的反方向挪动一小步,
期望最终到达最小值点.
即:

repeat until covergence: $\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\vec{\theta}), j = 0, 1, \dots$

> 注: 虽然Gradient descent算法中的learning rate(α)保持不变,
> 但由于代价函数 J 在最低点的导数为0,
> 从而在接近最低点"附近", |导数|总会逐渐减小到0, 从而 $\alpha \ grad$ 在逐渐减小.

具体问题: 针对`线性回归问题`的梯度下降算法:

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}, (\text{simultaneously update } j = 0, 1, \dots n)$$

训练数据预处理:

- Feature Scaling:

(一种让梯度下降的迭代次数减小的技巧)
考虑如果有两个特征 x1: house_size, x2: bedroom_amount,
其中 x1: 0~2000, x2: 1~5
那等高线会是一个非常细长的椭圆,
但如果我们把 x1 / 2000, x2 / 5
那等高线就不再那么细长, 可以证明, 在这种情况下所需的**迭代次数会变少**.
一般而言, 我们希望特征值落在[-1,1]附近.
可以考虑把`极差归1`或`标准差归1`作为目标.

> 注: 既然利用了FeatureScaling后的数据训练theta, 那当你需要预测时, 也别忘了将这个样本进行同样的处理!

- Mean normalization:

让每个 x_i 都减去特征i的平均值 μ_i, 让特征的平均值变为0. (但不处理 x_0, 因为规定一个恒为1的"第0个特征"是为了方便矩阵运算)

确保梯度下降法正常运行:

方法: 绘制 “J(θ)-迭代次数” 曲线
如果曲线大幅波动或者甚至发散(递增), 有可能是α(LearningRate)太大了
建议尝试α的方式: 0.001, 0.003, 0.01, 0.03, 0.1, 0.3, 1, ...

Normal Equation(正规方程法)

测试集大小为m, 特征数为n
我们构造m行(n+1)列矩阵X, 其中第0列为1作为bias;
以及m个元素的列向量 y
例如:

$$X = \begin{bmatrix} 1 & 2104 & 5 & 1 & 45 \\ 1 & 1416 & 3 & 2 & 45 \\ 1 & 1534 & 3 & 2 & 45 \\ 1 & 852 & 2 & 1 & 45 \end{bmatrix}, \quad y = \begin{bmatrix} 460 \\ 232 \\ 315 \\ 178 \end{bmatrix}$$

则根据高代385页推导 (或AndrewNg: 令偏导数$= \vec{0}$ 求解即可), 有如下公式:

$$\theta = (X^T X)^{-1} X^T y$$

pinv函数: 求伪逆运算(pseudo-inverse)

octave基础

1. 基本操作

```
1 == 2  % false %是注释, 似乎#也可以?
1 ~= 2  % 似乎!=也可以?

PS1(">>> ") % 设置命令提示符

a = pi; # 分号阻止打印
disp(a);
disp(sprintf("2 decimals: %.2f", a))

>>> format long
>>> a
```

```
3.14159265358979
>>> format short
>>> a
3.1416

A = [1,2; 3, 4; 5, 6]   # 逗号可以省略: A = [1 2; 3 4; 5 6]
v = [1 2 3]      # v = [1, 2, 3]
v = [1; 2; 3]    # 列向量
v = 1:10         # 起点:终点(含!) 步长为1
v = 1:0.1:2      # 起点:步长:终点(含!)

ones(2,3)        # 全1矩阵
zeros(3,4)       # 零矩阵
rand(2,5)        # 每个元素都是0到1的随机值
randn(4,3)       # 按(μ=0,σ^2=1)的高斯分布 生成4行3列的随机数
eye(4)           # 单位矩阵I

>>> w = -6 + sqrt(10) * (randn(1,10000))
>>> hist(w)        # histogram - 直方图
>>> hist(w, 50)    # 指定直方图的条数为50
```

2. 移动数据

```
>>> A = [1,2; 3,4; 5,6]
A =
   1   2
   3   4
   5   6

>>> size(A)    # 返回行列数(1x2矩阵)
ans = 3  2

>>> length(A) # 返回最大维度值
ans = 3        # (3 > 2)

>>> pwd        # Print Working Directory
>>> load('featuresX.dat')

>>> who        # 目前内存中的变量
>>> whos       # 更详细
>>> clear featuresX # 从内存中删除

>>> v = priceY(1:10)  # 获取第1到第10元素,共10个

>>> save test.mat v   # 存为二进制格式
>>> save test.mat A
>>> clear
>>> load test.mat     # 加载存入的所有变量
>>> load test.mat A   # 只加载变量A
```

```
>>> save test2.txt v -ascii   # 存成文本格式


>>> A = [1,2; 3,4; 5,6]
A =
   1   2
   3   4
   5   6

>>> A(3,2)       # 引用A[3][2]元素
>>> A(2,:);      # 第2行
>>> A([1,3], :) # 第1行和第3行的子矩阵

>>> A(2,:) = [10,11]  # assigning - 赋值

>>> A = [A, [100; 101; 102]]  # append new column

>>> A(:)       # put elements of A into single column



>>> A = [1,2; 3,4; 5,6];
>>> B = [11,12; 13,14; 15,16]

>>> C = [A, B]    # concatenate A and B, 左右排列
>>> C = [A; B]    # 上下排列
```

3. 计算数据

```
>>> A * C      # 矩阵乘法
>>> A .* B     # 对应位置元素的乘法, “.”一般修饰对于元素的操作
>>> A .^ 2     # 每个元素取平方
>>> 1 ./ A     # 每个元素取倒数
>>> log(v)
>>> exp(v)
>>> abs(v)
>>> -v
>>> v + 1
>>> A'

>>> a = [1, 15, 2, 0.5]
>>> val = max(a)
>>> val, ind = max(a)   # 同时取出下标

>>> a < 3          # 返回的矩阵中, 1 0 表示 true / false
>>> find(a < 3)    # find()返回nonzero的下标列表, 传入矩阵的话, 则先按照列优先遍历矩阵, 视为一个大的列向量, 再处理.

```

```
%{
  1. octave(matlab)在某些情况下是列优先的, 比如当你对一个矩阵运用max()函数时, 它事实上会返回"每一列的最大值"组成的列表
  2. octave似乎具有某种机制识别你所需要的返回值个数, 比如对max()、find()的调用
%}

>>> A = magic(4)  # magic()用于生成幻方, 此处仅用于快速生成一个4x4矩阵
>>> [r, c] = find(A >= 7)   # r, c是两个列表, 合在一起看, 表示矩阵中nonzero元素的索引

>>> sum(a)    # 求a中元素之和
>>> prod(a)   # 求a中元素之积
>>> floor(a)  # 每个元素向下取整
>>> rand(3)   # 3阶随机矩阵
>>> max(A, B) # 两个矩阵对应位置的最大元素组成的矩阵

>>> max(A, [], 1) # 显式指明列优先, 即每一列最大值
>>> max(A, [], 2) # 行优先, 即每一行的最大值
>>> max(max(A))   # 整个矩阵的最大值

>>> sum(A, 1)   # 显式指明按照列优先求和
>>> sum(A, 2)   # 按照行优先求和
>>> sum(sum((A .* eye(9))))   # 求矩阵的迹的技巧
```

4. 数据绘制

```
t = [0: 0.01: 1]
y1 = sin(1 * pi * t)
y2 = sin(2 * pi * t)
y3 = sin(3 * pi * t)
plot(t, y1)   # 绘制正弦函数
plot(t, y2)   # 绘制另一个正弦函数, 之前的图像被替换掉了

hold off      # 设置绘制新曲线前,擦除之前的曲线(默认)
hold on       # 设置不擦除之前的
plot(t, y1, 'r')  # 指定颜色
plot(t, y2, 'g')
plot(t, y3, 'b')

xlable('time')      # 设置x轴标签
ylable('amplitude') # 设置y轴标签

legend("w=pi", "w=2pi") # 先后顺序给曲线命名 legend-图例

title("test my plot") # 标题

print -dpng 'myPlot.png'  # 输出到图片文件

close # 关闭图像
```

```
# 为多张图片标号
figure(1); plot(t, y1)  # 图1
figure(2); plot(t, y1)  # 图2

# 子图
subplot(1, 2, 1)    # 分割为1x2网格, 并定位到第一个格子
plot(t, y1)
subplot(1, 2, 2)    # 定位到第二个格子
plot(t, y2)

# 设置绘制范围
axis([0.5, 1, -1, 1])

clf; # clear figure 清空屏幕

# 矩阵可视化
A = magic(5);
imagesc(A);
colorbar;   # 显示 颜色-value 对应条
colormap gray;  # 改为灰度图, 还有更多选项, 比如rainbow,viridis等等, help colormap 即可查看

# 一次输入多条指令
a = 1, b = 2, c = 3   # 不显示输出
a = 1; b = 2; c = 3;  # 显示输出
```

5. 控制语句

```
v = zeros(10, 1)

# for循环
for i = 1: 10,
  v(i) = 2 ^ i;
end;

indices = 1: 10;    # 感觉有点像python的range(1, 11)
for i = indices,
  disp(i);
end;


# while循环
while i <= 5,
  v(i) = 100;
  i = i + 1;
end;
```

```
# if elseif else
if v(1) == 1,
  disp("value is one");
elseif v(1) == 2,
  disp("value is two");
else
  disp("others");
end;


# while with break
while true,
  v(i) = 999;
  i = i + 1;
  if i == 6,
    break;
  end;
end;
```

6. 函数

```
# 当函数存储在一个 "函数名.m" 的文件中时, 需要下列方法让octave可以定位到这个函数
cd "/Users/cuipy/Desktop/myFuncs"       # 方法1
addpath('/Users/cuipy/Desktop/myFuncs') # 方法2

# 单返回值 y就是返回值 不显式return
function y = myFunc(x)
  y = x ^ 2

# 多返回值
function [y1, y2] = myFunc2(x)
  y1 = x ^ 2
  y2 = x ^ 3

# 例: 代价函数
function J = costFunctionJ(X, y, theta)
    # X is the "design matrix" containing our training examples
    # y is the class labels

    m = size(X, 1);             # number of training examples
    predictions = X * theta;    # predictions of hypothesis on all m
examples
    sqrErrors = (predictions - y) .^ 2; # squared errors

    J = 1 / (2 * m) * sum(sqrErrors);

# 匿名函数
# 格式: f = @(参数表)(返回值表达式)
# 例如:
```

```
#    我们写了一个costFunction, 接受3个参数,
#    但对于单独一次fminunc的调用, 其使用的训练集是不变的,
#    而且fminunc接收的第一个参数 必须是一个一元函数.
#    我们使用匿名函数来解决这个问题.
[theta, cost] = fminunc(@(t)(costFunction(t, X, y)), initial theta,
options);
```

> 这里是cpy的`octave/Matlab`的debug总结:
>
> 1. 在octave中, 对一个矩阵A取某一行/列时, 事实上是在取它的一个子矩阵!
>   比如要取第2行, 那就是A(2, :), 而不是A(2), 后者等价于A(2,1)
>   甚至对于一个数字, 也可以(应该?)看成一个1x1的矩阵
> 2. 在python中, 可以通过list1[1:]获取除了首元素的切片;
>   但在octave中, 必须指明是从第二个元素到`end`: theta(1:end, 1)
> 3.

向量化编程

1. 使用向量点乘, 而非for循环来计算 $\Sigma_{i=1}^{m} x_i * y_i$, 可以通过并行性提高性能.
2. 使用矩阵运算实现线性回归的梯度下降算法:

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}, (\text{simultaneously update } j = 0, 1, \dots n)$$

比如考虑n=2个特征, m个数据的情况:

$$\theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_0^{(i)}$$
$$\theta_1 := \theta_1 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_1^{(i)}$$
$$\theta_2 := \theta_2 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_2^{(i)}$$

不难发现, 可以写成下列向量形式:

$$\vec{\theta} := \vec{\theta} - \alpha \vec{\delta}$$
$$\text{where: } \vec{\delta} = \frac{1}{m} \sum_{i=1}^{m} (h_\theta(\vec{x}^{(i)}) - y^{(i)}) \vec{x}^{(i)}$$
$$\left(\quad \text{further: } \vec{\delta} = \frac{1}{m} X^T * (X \cdot \theta - y) \quad \right)$$

Week 03

Classification(Logistic Regression)

执行分类任务(输出值为0或1), 我们想让预测值落在[0,1]
依然用类似LinearRegression的Hypothesis函数, 不过要再复合上一个`sigmoid`函数(或者叫logistic Function):

$$sigmoid(z) = \frac{1}{1+e^{-z}}$$

`sigmoid`函数的性质: sigmoid(0) = 0.5, 值域为(0,1)

Decision Boundary: 决策边界, 是hypothesis函数 h_θ 的属性, 与训练集无关

和LinearRegression一样, 在Logistic Regression中, 可以运用`高阶多项式`来得到更复杂的决策边界.

> **高阶多项式** 如: $h_\theta(x) = sigmoid(\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_1^2 + \theta_4 x_2^2 + \cdots)$
>
> 我们通过将2个特征x1 x2, 组合成各种高次项, 生成更多的特征, 再用这些特征作为训练集.

模型参数θ的拟合

代价函数设计

- 对于单训练样本

  我们不能继续采用之前的$(h_\theta(x) - y)^2$来作为CostFunction了.
  因为在这里, 由于sigmoid 函数的介入, 会导致CostFunction变成非凸(non-convex)函数,
  具有很多局部极小值, 不利于进行优化.

  可以证明, 采用下列CostFunction可以避免这一问题:

$$Cost(h_\theta(x), y) = \begin{cases} -\log(h_\theta(x)) & \text{if y} = 1 \\ -\log(1 - h_\theta(x)) & \text{if y} = 0 \end{cases}$$

简单观察性质：比如$\mathrm{y}=1$时，如果预测值就是1，那么$\mathrm{Cost}=0$; 但如果预测值接近0，那么$\mathrm{Cost}->$正无穷!

- 等价写法(这是由于y只可能取0或1)

$$Cost(h_\theta(x), y) = -y\log(h_\theta(x)) - (1-y)\log(1 - h_\theta(x))$$

- 对于整个训练集

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

梯度下降法优化J(θ)

直接给出公式:

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}, (\text{simultaneously update } j = 0, 1, \dots n)$$

我们惊喜地发现, 这和线性回归的公式具有完全相同的`形式`!

> 注: 虽然形式相同, 但要注意此时 h_θ 已经发生变化, 因为多复合了一个`sigmoid`函数.

高级优化算法

- Conjugate gradient
- BFGS
- L-BFGS

  这些算法更高效, 但也更复杂, 不过即使不了解算法的细节, 也不妨碍使用.

  运用专业人员已经写好的库, 不要自己去实现.

例: Octave中的`fminunc`方法

> fminunc 即 minimization unconstrained (无约束最小化函数)
> 所谓无约束, 是指theta的各个分量的取值没有约束.
> 在一些情况下, theta的分量取值有限制, 比如theta < 1

```
# 首先要自己实现一个函数, 用于计算优化算法所需的 J(theta) 和 grad_J(theta)
function [jVal, gradient] = costFunction(theta)
  jVal = ...
  gradient = zeros(n, 1)
  gradient(1) = ...
  gradient(2) = ...
  ... ...
  gradient(n) = ...
end;

# 调用库函数fminunc
options = optimset('GradObj','on', 'MaxIter','100');  # 提供梯度:on; 最大迭代:100
initialTheta = zero(2,1);
[optTheta, functionVal, exitFlag] = fminunc(@costFunction, initialTheta, options); # @表示函数指针
```

多元分类: One-versus-ALL Classification

很多情况下, 需要预测的事情并不是非黑即白的.
比如: 天气: [晴, 雨, 多云, 雾, ...], 邮件类别: [Word, Family, Friends, College,...]

事实上对于更多类别, 比如对于Δ口〇三类物体,
我们可以使用`one-versus-all / one-versus-rest`方法,
将问题转化为之前的2元情况:

对于训练集中的数据,
先看成 Δ / (口&〇) 两类, 然后算出一个theta1;
再看成 口 / (Δ&〇) 两类, 然后算出一个theta2;
最后看成 〇 / (口&Δ) 两类, 然后算出一个theta3.

这样训练出了三个分类器h1,h2,h3.
对于需要被预测的新输入x,
我们只需要分别代入三个分类器, 得到三个预测成功率,
选出最大的即可.

Overfitting(过拟合)

解决方法:

1. 减少特征数量, 选取重要的保留, 其余舍弃. (模型选择算法)
2. 正则化(regularization)

正则化(regularization)

当我们用高次多项式拟合训练集时,
经验而言, 如果theta的每个分量都比较小, 就不容易出现过拟合
为了做到这一点, 我们为CostFunction加入惩罚项 (我们一般**不**对 θ_0 加入惩罚, 虽然加不加影响不大):

$$J(\vec{\theta}) = \frac{1}{2m}(\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2 \quad + \quad \lambda\sum_{i=1}^{n}\theta_i^2)$$

也就是说, 代价函数分为了两部分: 前半部分帮助拟合训练集, 后半部分保持参数尽可能小.

λ的选取:

- 如果过大, 那最后的$\theta_1, \ldots, \theta_n$都趋于0, 只剩下$\theta_0$, 这是常值函数, 造成欠拟合.
- 如果过小, 那高次项可能会造成过拟合

Linear Regression的正则化

- 梯度下降法

$$\theta_0 : \text{same as before}$$
$$\theta_j := \theta_j - \alpha(\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)} + \frac{\lambda}{m}\theta_j) \quad j = 1, 2, \cdots, n$$
$$[\Rightarrow \quad \theta_j := \theta_j(1 - \frac{\alpha\lambda}{m}) - \alpha\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)} \ ]$$

化简成**下面的形式**后, 我们发现:
相比于以前的梯度下降, 正则化的算法的改变在于在每次迭代时, 都给 θ_j乘上了一个略小于1的数.

- 正规方程法:

$$\theta = (X^T X + \lambda \begin{bmatrix} 0 & \\ & E_n \end{bmatrix})^{-1} X^T y$$

相比与之前, 正则化的算法只是加入了一个

$$\lambda \begin{bmatrix} 0 & \\ & E_n \end{bmatrix}$$

顺便, 与之前不同, 只要 $lambda > 0$, 就可以确保pinv作用的矩阵时可逆的.

Logistic Regression的正则化

$$\theta_0 : \text{same as before}$$
$$\theta_j := \theta_j - \alpha(\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)} + \frac{\lambda}{m}\theta_j) \quad j = 1, 2, \cdots, n$$

Week 04

Neural Network Representation

在之前, 我们通过手动将特征扩充为各种高次多项式, 获得非线性的拟合. 但一旦本来的特征情况就很多, 比如有100个特征, 那即使只考虑对二次项的扩充, 也有 $C_{100}^2 + 100 = 5050$ 种特征. 很容易导致过拟合, 以及算力无法支撑这么大的数据规模.
因此, 为了建立非线性拟合, 我们需要新的算法.
通过对生物大脑的神经元的简单抽象, 我们可以以此建立人工神经网络

Feedforward Propagation Algorithm

符号约定:

$g = sigmoid$

$\Theta_{i,j}^{(k)}$ 表示第(k+1)层中, 第i个神经元的第j个参数. 因为第(k+1)层拥有的$\Theta(k)$是用来和第`k`层的output来做运算的.

(换言之, $\Theta^{(k)}$ 与 $a^{(k)}$ 用于计算 $a^{(k+1)}$)

$z^{(i)} = \Theta \cdot a^{(i-1)}$ 表示第i层的"中间运算结果",

令$a_j^{(i)} \triangleq g(z^{(i)})$, 再加上 $a_0^{(i)} \triangleq 1$ 作为偏置项bias,

凑在一起得到 $a^{(i)} \triangleq [1;\ g(z^{(i)})]$ 表示第i层的输出经过sigmoid处理, 加入bias后的向量. 称为**激活值(activation)**. 这将作为下一层的输入.

> 注: 当用矩阵 $\Theta^{(i)}$ 来描述第 i 层的所有神经元时, 矩阵 $\Theta^{(i)}$ 的第 j 行对应于第 j 个神经元的信息.
>
> (当然也可以理解成第i层和第j层之间的权重)
>
> > 所以符号 Θ 其实是一个"三维数组", 表示了整个神经网络

Week 05

Cost Function of Neural Network (for Classification)

符号约定:

L = 神经网络层数

s_l = 第 l 层的神经元数量, 不包括bias单元.

K = 分类的类别数

(回顾) Logistic Regression 的代价函数:

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

下面给出 Neural Network 的代价函数:

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

其中 $h_\Theta(x)$ 是**整个神经网络**的输出, 在分类问题中是一个长度为 K 的向量.

相比于(i), 神经网络由于输出一个向量, 所以在(ii)的前半部分需要多一个**把所有分量求和**的步骤;

正则化的方式则是对**三维矩阵** Θ 的每个元素都平方后相加.

> 注: $\Theta^{(i)}$ 描述第 i+1 层神经元, 第 i+1 层的神经元数目为 s_{i+1}, 其中每个神经元又具有 s_i 个参数 (因为它要处理第i层的输出)
>
> 在(ii)中, 我们同样不去处理第0个参数.

Back Propagation Algorithm (反向传播算法)

上一节我们给出了代价函数J的计算公式, 我们只需要再计算梯度就好了.

注意, 我们要对每个 $\Theta_{ij}^{(l)}$ 都计算梯度. 即每一层的每个神经元的每个参数.

1. 引入概念:

$\delta_j^{(l)}$ 表示 第l层的第j个节点的"**误差(error)**".

> 回顾类似的表示法: $a_j^{(l)}$ 表示 第l层的第j个节点的"激活值(activation)".

2. 对于单个训练样本(x, y)的 $\delta_j^{(l)}$ 计算公式(证明略)

- 对于Output-Layer:

$$\delta_j^{(L)} = a_j^{(L)} - y_j \qquad (\text{Output layer})$$

$$\delta^{(L)} = a^{(L)} - y \qquad (\text{Output layer } \#\text{Vectorized}\#)$$

- 对于Hidden-Layer ($l = 2, 3, \ldots, L-1$):

$$\begin{aligned}\delta^{(l)} &= (\Theta^{(l)})^T \delta^{(l+1)} .* g'(z^{(l)}) \\ &= (\Theta^{(l)})^T \delta^{(l+1)} .* [\, a^{(l)}(1 - a^{(l)}) \,]\end{aligned} \qquad (\text{Hidden layer } \#\text{Vectorized}\#)$$

3. 对于单个训练样本(x, y)的梯度计算(证明略)

$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = a_j^{(l)} \delta_i^{(l+1)} \qquad (\text{ignoring regularzation})$$

4. 对于具有m个训练样本的训练集的梯度计算

首先建立一个三维矩阵 Δ , 设置所有元素 $\Delta_{ij}^{(l)} = 0$.

for循环每个样本:

先用前向传播计算出每一层的 $a^{(l)}$,

然后用后向传播计算出每一层的 $\delta^{(l)}$,

然后给每个 $\Delta_{ij}^{(l)}$ 设置增量: $\Delta_{ij}^{(l)} = \Delta_{ij}^{(l)} + a_j^{(i)} \delta_i^{(i+1)}$,

(或者使用矩阵乘法: $\Delta^{(l)} = \Delta^{(l)} + \delta^{(l+1)} (a^{(l)})^T$)

最后计算梯度:

$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = \frac{1}{m} \Delta_{ij}^{(l)} + \lambda \Theta_{ij}^{(l)} \qquad (\text{j} \neq 0)$$

$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = \frac{1}{m} \Delta_{ij}^{(l)} \qquad (\text{j} = 0)$$

5. 理解:

$\delta_j^{(i)}$ 表示第l层的第j个节点的误差(eror), 确切地说, 是 CostFunction_i 关于 $z_j^{(l)}$ 的偏导数.

参数展开(为了使用优化算法库函数)

由于 `fminunc(@costFunction, initialTheta, options)` 中的
costFunction, initialTheta 都应该是一维的,
但在反向传播算法中, 我们计算出的 J 和 grad 都是二维的.
这要求我们在costFunction的实现中, 首先把传入的向量还原回矩阵, 而在返回时则需要把梯度转换为一维向量.

```
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)
```

> 矩阵形式: 方便进行正向&反向传播算法;
> 长向量形式: 用于调用库中的高级优化算法函数.