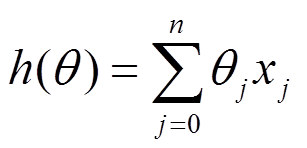
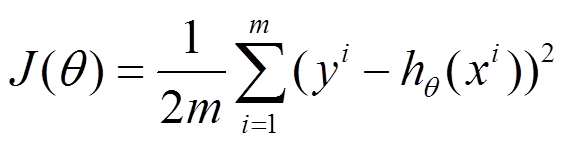
[随机梯度下降和批量梯度下降](http://blog.csdn.net/lilyth_lilyth/article/details/8973972)

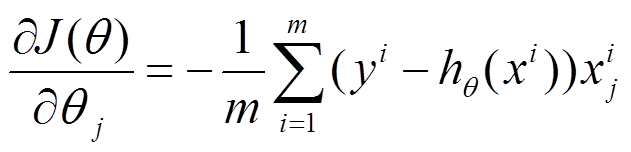
假设是拟合的函数，是损失函数，其中是样本数，是参数的个数。



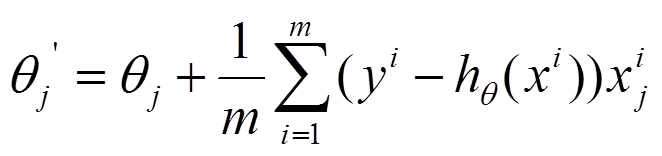


**1、批量梯度下降的求解思路如下：**

（1）将对求偏导，得到每个对应的的梯度



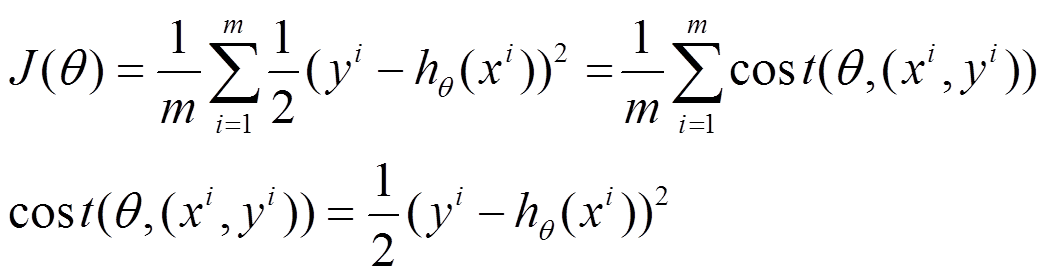
（2）由于是要最小化风险函数，所以按每个参数的**梯度负方向**，来更新每个



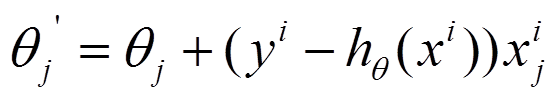
（3）从上面公式可以注意到，它得到的是一个全局最优解，但是每迭代一步，都要用到训练集所有的数据，如果m很大，则学习的速度回很慢。所以，这就引入了另外一种方法，随机梯度下降。

**2、随机梯度下降的求解思路如下：**

（1）上面的风险函数可以写成如下这种形式，损失函数对应的是训练集中每个样本的粒度，而上面批量梯度下降对应的是所有的训练样本：



（2）每个样本的损失函数，对求偏导得到对应梯度，来更新



（3）随机梯度下降是通过每个样本来迭代更新一次，如果样本量很大的情况（例如几十万），那么可能只用其中几万条或者几千条的样本，就已经将迭代到最优解了，对比上面的批量梯度下降，迭代一次需要用到十几万训练样本，一次迭代不可能最优，如果迭代10次的话就需要遍历训练样本10次。但是，SGD伴随的一个问题是噪音较BGD要多，使得SGD并不是每次迭代都向着整体最优化方向。

**3、对于上面的linear regression问题，与批量梯度下降对比，随机梯度下降求解的会是最优解吗？**

（1）批量梯度下降---最小化所有训练样本的损失函数，使得最终求解的是全局的最优解，即求解的参数是使得风险函数最小。

（2）随机梯度下降---最小化每条样本的损失函数，虽然不是每次迭代得到的损失函数都向着全局最优方向， 但是大的整体的方向是向全局最优解的，最终的结果往往是在全局最优解附近。

**4、梯度下降用来求最优解，哪些问题可以求得全局最优？哪些问题可能局部最优解？**

对于上面的linear regression问题，最优化问题对theta的分布是unimodal，即从图形上面看只有一个peak，所以梯度下降最终求得的是全局最优解。然而对于multimodal的问题，因为存在多个peak值，很有可能梯度下降的最终结果是局部最优。

**5、随机梯度和批量梯度的实现差别**

以前一篇博文中NMF实现为例，列出两者的实现差别（注：其实对应**[Python](http://lib.csdn.net/base/python" \o "Python知识库" \t "_blank)**的代码要直观的多，以后要练习多写python！）

1. // 随机梯度下降，更新参数
2. **public** **void** updatePQ\_stochastic(**double** alpha, **double** beta) {
3. **for** (**int** i = 0; i < M; i++) {
4. ArrayList<Feature> Ri = **this**.dataset.getDataAt(i).getAllFeature();
5. **for** (Feature Rij : Ri) {
6. // eij=Rij.weight-PQ for updating P and Q
7. **double** PQ = 0;
8. **for** (**int** k = 0; k < K; k++) {
9. PQ += P[i][k] \* Q[k][Rij.dim];
10. }
11. **double** eij = Rij.weight - PQ;
13. // update Pik and Qkj
14. **for** (**int** k = 0; k < K; k++) {
15. **double** oldPik = P[i][k];
16. P[i][k] += alpha
17. \* (2 \* eij \* Q[k][Rij.dim] - beta \* P[i][k]);
18. Q[k][Rij.dim] += alpha
19. \* (2 \* eij \* oldPik - beta \* Q[k][Rij.dim]);
20. }
21. }
22. }
23. }
25. // 批量梯度下降，更新参数
26. **public** **void** updatePQ\_batch(**double** alpha, **double** beta) {
28. **for** (**int** i = 0; i < M; i++) {
29. ArrayList<Feature> Ri = **this**.dataset.getDataAt(i).getAllFeature();
31. **for** (Feature Rij : Ri) {
32. // Rij.error=Rij.weight-PQ for updating P and Q
33. **double** PQ = 0;
34. **for** (**int** k = 0; k < K; k++) {
35. PQ += P[i][k] \* Q[k][Rij.dim];
36. }
37. Rij.error = Rij.weight - PQ;
38. }
39. }
41. **for** (**int** i = 0; i < M; i++) {
42. ArrayList<Feature> Ri = **this**.dataset.getDataAt(i).getAllFeature();
43. **for** (Feature Rij : Ri) {
44. **for** (**int** k = 0; k < K; k++) {
45. // 对参数更新的累积项
46. **double** eq\_sum = 0;
47. **double** ep\_sum = 0;
49. **for** (**int** ki = 0; ki < M; ki++) {// 固定k和j之后,对所有i项加和
50. ArrayList<Feature> tmp = **this**.dataset.getDataAt(i).getAllFeature();
51. **for** (Feature Rj : tmp) {
52. **if** (Rj.dim == Rij.dim)
53. ep\_sum += P[ki][k] \* Rj.error;
54. }
55. }
56. **for** (Feature Rj : Ri) {// 固定k和i之后,对多有j项加和
57. eq\_sum += Rj.error \* Q[k][Rj.dim];
58. }
60. // 对参数更新
61. P[i][k] += alpha \* (2 \* eq\_sum - beta \* P[i][k]);
62. Q[k][Rij.dim] += alpha \* (2 \* ep\_sum - beta \* Q[k][Rij.dim]);
63. }
64. }
65. }
66. }