

处理大量数据

stochastic gradient descent 随机梯度下降

批量梯度下降 (batch gradient descent) 有一个问题，就是它的每一步都要遍历全部的数据，因此当我们的数据量非常大的时候，这样的做法非常没有效率。

因此我们引入一个新的方法，随即梯度下降

Batch gradient descent	Stochastic gradient descent
$\rightarrow J_{train}(\theta) = \frac{1}{2m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)})^2$	$\rightarrow cost(\theta, (x^{(i)}, y^{(i)})) = \frac{1}{2} (h_\theta(x^{(i)}) - y^{(i)})^2$
Repeat {	$J_{train}(\theta) = \frac{1}{m} \sum_{i=1}^m cost(\theta, (x^{(i)}, y^{(i)}))$
$\rightarrow \theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$	1. Randomly shuffle dataset. ↩
$\qquad\qquad\qquad \frac{\partial}{\partial \theta_j} J_{train}(\theta)$	2. Repeat {
$(for \ every j = 0, \dots, n)$	$\left\{ \begin{array}{l} \text{for } i=1, \dots, m \ : \\ \theta_j := \theta_j - \alpha (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)} \\ (for \ j=0, \dots, n) \end{array} \right.$
$\}$	$\rightarrow \frac{\partial}{\partial \theta_j} cost(\theta, (x^{(i)}, y^{(i)}))$
$m = 300,000,000$	$\rightarrow (x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), (x^{(3)}, y^{(3)}), \dots$

我们定义参数 θ ，关于训练样本 $(x(i), y(i))$ 的代价等于二分之一倍假设 $h(x(i))$ 跟实际输出 $y(i)$ 的误差的平方。因此这个代价函数值实际上测量的是我的假设在某个样本 $(x(i), y(i))$ 上的表现。

随机梯度下降法的第一步是将所有数据打乱，即将所有 m 个数据重新排列，这是标准的数据预处理过程

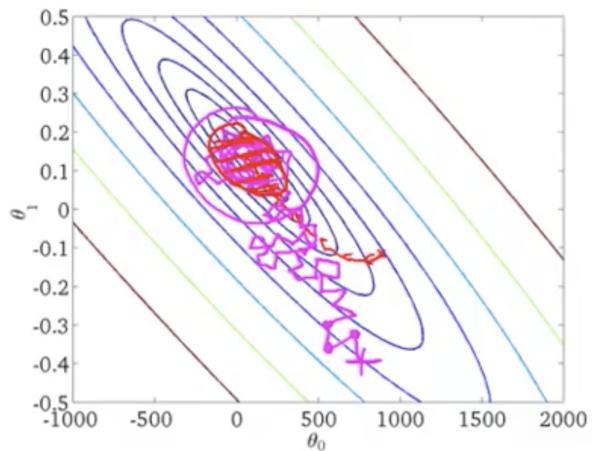
简单来说，随机梯度下降就是每一步我们只需要考虑一个样本，我们要使算法尽量拟合这一个样本，然后重复这个过程，直到我们拟合完所有 m 个样本

Stochastic gradient descent

- 1. Randomly shuffle (reorder) training examples

→ 2. Repeat {
 for $i := 1, \dots, m\{$
 $\rightarrow \theta_j := \theta_j - \alpha(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}$
 (for every $j = 0, \dots, n$)
 }

$\rightarrow m = 300,000,000$



我们会重复随机梯度下降1到10次，这样我们可以得到一个比较好的算法。

一般情况下，即使我们只进行一次完整的随机梯度下降，也能得到一个不错的结果。

另外有一点需要注意，就是随机梯度下降不会达到一个全局最小值，而是不停徘徊在全局最小值附近（如上图右边的图像）

Mini-Batch Gradient Descent 小批量梯度下降

Mini-batch gradient descent

- Batch gradient descent: Use all m examples in each iteration
- Stochastic gradient descent: Use 1 example in each iteration

Mini-batch gradient descent: Use b examples in each iteration

$$b = \text{mini-batch size.} \quad b = 10. \quad \frac{2-100}{}$$

Get $b = 10$ examples $(x^{(i)}, y^{(i)}), \dots, (x^{(i+9)}, y^{(i+9)})$

$$\rightarrow \theta_j := \theta_j - \alpha \frac{1}{10} \sum_{k=i}^{i+9} (h_\theta(x^{(k)}) - y^{(k)}) \cdot x_j^{(k)}$$

$i := i + 10$

小批量梯度下降就是我们每次对 b 个样本进行梯度下降（通常为2到100之间的任意数值），这样做的好处是我们可以用向量化的方法使得它计算起来非常方便，通常比随机梯度下降还要快，当然如果你没有一个比较好的向量化的方法，还是老老实实用随机梯度下降吧

小批量梯度下降的一个缺点是有一个额外的参数 b 你需要调试小批量大小因此会需要一些时间但是如果你有一个好的向量化实现这种方法有时甚至比随机梯度下降更快

验证随机梯度下降

Checking for convergence

→ Batch gradient descent:

→ Plot $J_{train}(\theta)$ as a function of the number of iterations of gradient descent.

$$\rightarrow J_{train}(\theta) = \frac{1}{2m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)})^2$$

M = 300, 500, 800

$\Rightarrow (x^{(i)}, y^{(i)}), (x^{(i+1)}, y^{(i+1)}), \dots$

→ Stochastic gradient descent:

$$\rightarrow cost(\theta, (x^{(i)}, y^{(i)})) = \frac{1}{2} (h_\theta(x^{(i)}) - y^{(i)})^2$$

→ During learning, compute $cost(\theta, (x^{(i)}, y^{(i)}))$ before updating θ using $(x^{(i)}, y^{(i)})$.

→ Every 1000 iterations (say), plot cost($\theta, (x^{(i)}, y^{(i)})$) averaged over the last 1000 examples processed by algorithm.

为了检验随机梯度下降算法是否在收敛，我们需要画出最优化的代价函数

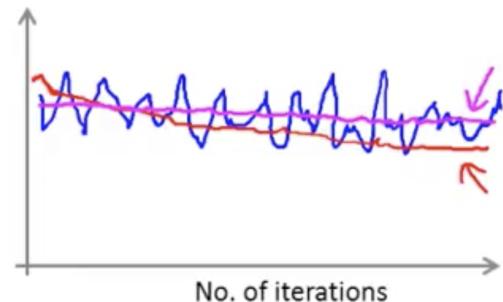
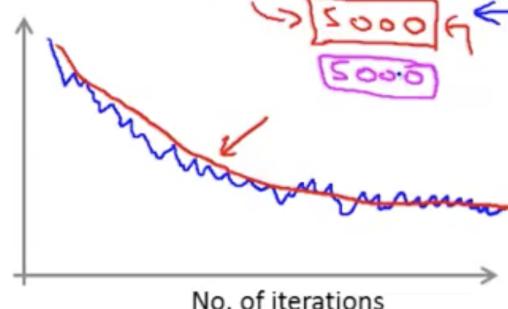
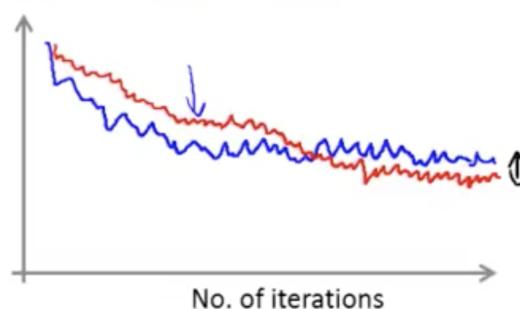
我们的做法如上图，首先，在我们更新theta之前，我们要用老的参数theta来计算单个当本的代价函数，

这样做是因为theta被更新后多半能表现得更好，因此我们暂时先不更新他，而是用老的theta去求最后1000个样本的代价函数的平均值，这可以帮助我们算出算法在最后1000个样本上的表现。

随机梯度下降法的这个步骤，只需要在每次更新θ之前进行，也并不需要太大的计算量。要做的就是，每1000次迭代运算中，我们对最后1000个样本的cost值求平均，然后通过观察这些画出来的图，我们就能检查出随机梯度下降是否在收敛。以下是几幅画出来的图的栗子。

Checking for convergence

Plot cost($\theta, (x^{(i)}, y^{(i)})$), averaged over the last 1000 (say) examples



左上：正常收敛，红线表示更小的学习速率 α 可以拟合得更好，因为我们的随机梯度算法不会收敛

到全局最小值，而是在那附近徘徊

右上：用更多的样本可以得到更好的曲线。红线代表5000样本的情况，红色更平滑，然而缺点是我们的数据样本变多了，所以我们的更新也会更延迟

左下：看上去没有下降，但如果我们用更多的样本，可能会得到红线的结果，我们能看到他还是有点在收敛。

当然这种情况也可能得到紫线那样的结果，说明算法确实出了点问题。

右下：算法明显发散了，这要求我们用更小的学习速率 α

Stochastic gradient descent

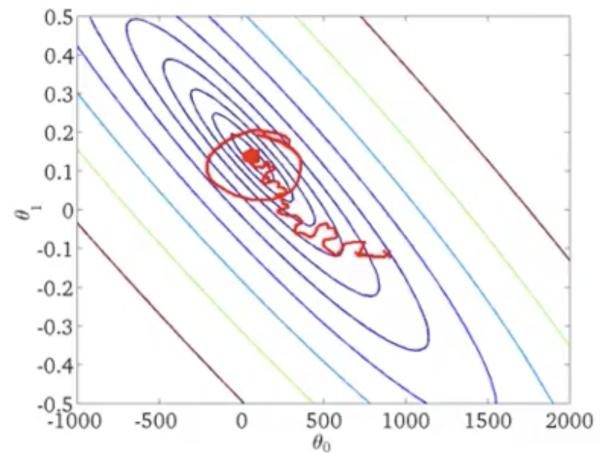
$$cost(\theta, (x^{(i)}, y^{(i)})) = \frac{1}{2}(h_{\theta}(x^{(i)}) - y^{(i)})^2$$

$$J_{train}(\theta) = \frac{1}{2m} \sum_{i=1}^m cost(\theta, (x^{(i)}, y^{(i)}))$$

1. Randomly shuffle dataset.

2. Repeat {

```
    for i := 1, ..., m      {  
         $\theta_j := \theta_j - \alpha(h_{\theta}(x^{(i)}) - y^{(i)})x_j^{(i)}$   
        (for j = 0, ..., n)  
    }  
}
```



Learning rate α is typically held constant. Can slowly decrease α over time if we want θ to converge. (E.g. $\alpha = \frac{\text{const1}}{\text{iterationNumber} + \text{const2}}$) $\alpha \rightarrow 0$

最后一点，由于随机梯度下降总是徘徊在全局最小值附近，如果想要得到全局最小值，我们需要对学习速率 α 进行逐渐减小的处理，例如上图中最小面的那一行， α 会随着循环次数的增大而减小，这会使得我们越来越接近全局最小值，当然如果我们最终把 α 设为0，我们就能得到全局最小值。

在实际应用中，大部分时候我们并不需要这么做，因为一个接近全局最小值的解往往就已经很足够了。

除非有特殊情况需要更精确的全局最小值，不然我们通常不会用这种办法。

Online Learning 在线学习

当今的很多网站都使用了不同版本的在线学习系统。

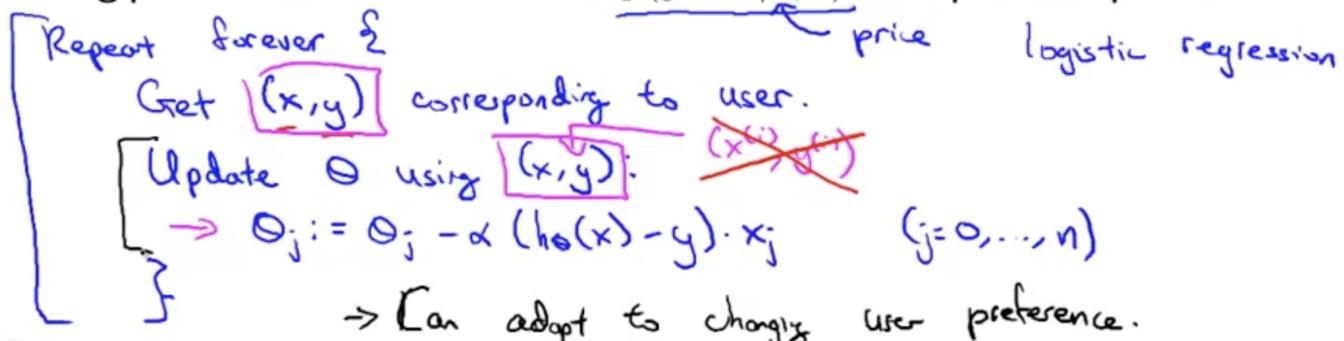
假如我们有一个由连续的用户流引发的连续数据流，我们可以用在线学习机制从数据流中学习用户偏好，

一边优化一些关于网站的决策。

Online learning

Shipping service website where user comes, specifies origin and destination, you offer to ship their package for some asking price, and users sometimes choose to use your shipping service ($y = 1$), sometimes not ($y = 0$).

Features x capture properties of user, of origin/destination and asking price. We want to learn $p(y = 1|x; \theta)$ to optimize price.



上图是一个栗子，我们通过学习用户对于价格的反应，试图给出一个更合理的价格，用户购买的情况被标记为1，用户放弃购买被标记为0。

如果我们的数据足够多，当我们每学习完一组样本，我们就可以抛弃它并且不再使用，因为我们的数据几乎是无限的，也就不需要重复测试一组数据。

如果我们的样本不是很多，我们则要考虑存储下这些数据。

Other online learning example:

Product search (learning to search)

User searches for "Android phone 1080p camera" \leftarrow

Have 100 phones in store. Will return 10 results.

$\rightarrow x = \text{features of phone, how many words in user query match name of phone, how many words in query match description of phone, etc.}$

$\rightarrow y = 1$ if user clicks on link. $y = 0$ otherwise.

\rightarrow Learn $p(y = 1|x; \theta)$. \leftarrow predicted CTR

$(x, y) \leftarrow$
↑ ↑

\rightarrow Use to show user the 10 phones they're most likely to click on.

Other examples: Choosing special offers to show user; customized selection of news articles; product recommendation; ...

上图展示了更多关于在线学习的思路，例如我们想要推荐手机给用户，我们可以收集一些关于手机的特征，

对于每一个手机以及一个给定的用户搜索命令，我们可以构建一个特征矢量 x ，这个特征矢量 x 可能会抓取手机的各种特点，它可能会抓取类似于用户搜索命令与这部电话的类似程度有多高这样的信息。

简单地说就是我们想要预测用户的点击率，这类问题又被称为预估点击率CTR(Click Through Rate)，它仅仅代表学习用户将点击某一个特定的你提供给他们的链接的概率。

一些其他的栗子包括给不同的用户展示不同的优惠券，不同的新闻。不同的推荐，等等。

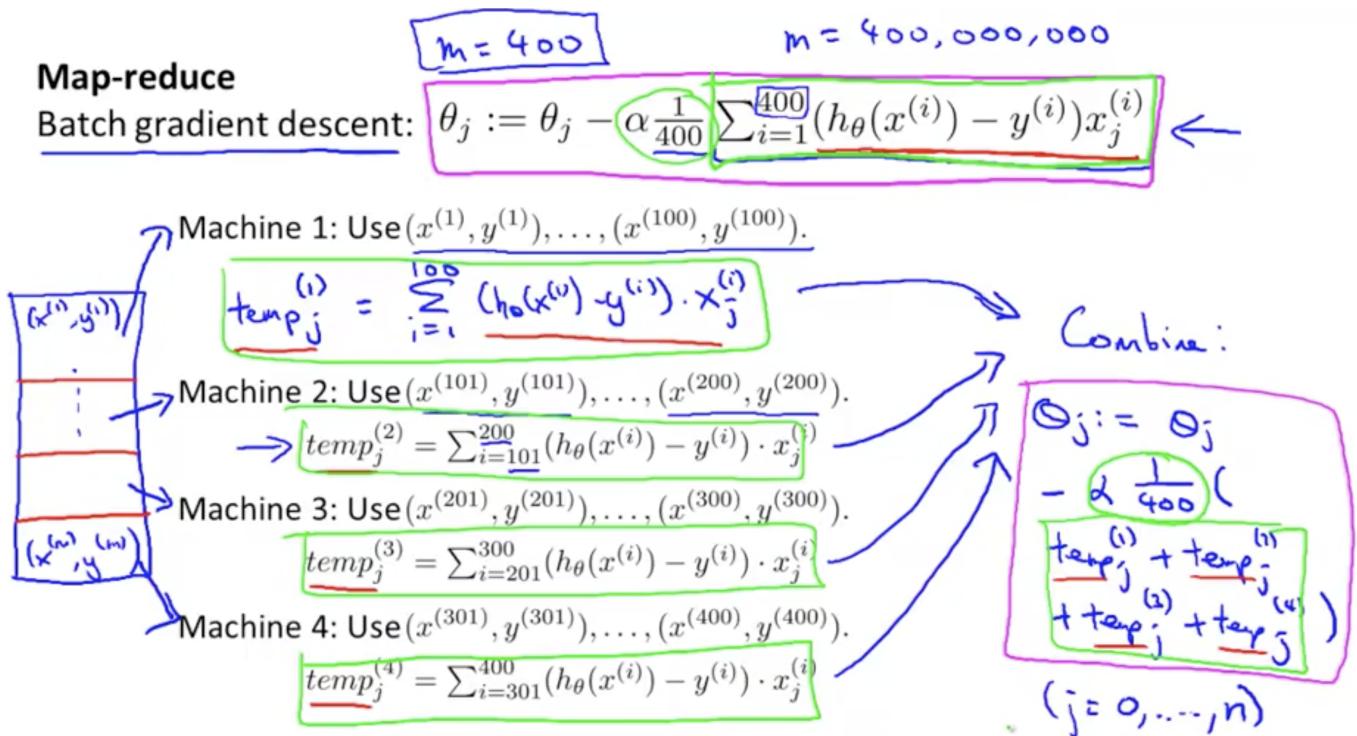
在这些问题上我们可能没有必要去建立一个固定的数据集，我们可以用一个在线的算法来连续学习，从用户不断产生的数据中来学习。

这个算法与随机梯度下降算法非常类似，唯一的区别的是我们不会使用一个固定的数据集。我们会做的是获取一个用户样本，从那个样本中学习，然后丢弃那个样本并继续下去，而且如果你对某一种应用有一个连续的数据流，这样的算法可能会非常值得考虑。当然，在线学习的一个优点就是如果你有一个变化的用户群或者你在尝试预测的事情在缓慢变化，就像你的用户的品味在缓慢变化 这个在线学习 算法可以慢慢地 调试你所学习到的假设，将其调节更新到最新的用户行为。

Map Reduce 映射化简

映射化简的原理非常简单，但却是一个非常重要的算法。

它的核心思想就是把最消耗资源的求和运算分散给不同的机器或核来运算，最后再集合到中心机上完成参数更新。参看下图。



只要是有求和的部分，都可以通过这个办法减轻计算压力，甚至多核的CPU也可以通过分配任务的方式完成多线程运算。