17、大规模机器学习(Large Scale Machine Learning)

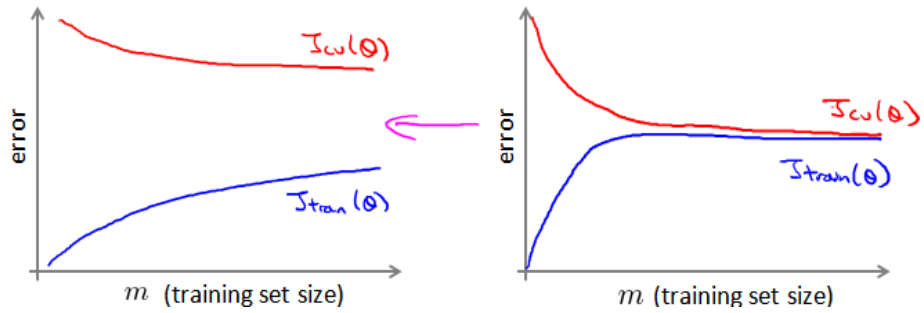
17.1 大型数据集的学习

参考视频: 17 - 1 - Learning With Large Datasets (6 min).mkv

如果我们有一个低方差的模型，增加数据集的规模可以帮助你获得更好的结果。我们应该怎样应对一个有 100 万条记录的训练集？

以线性回归模型为例，每一次梯度下降迭代，我们都需要计算训练集的误差的平方和，如果我们的学习算法需要有 20 次迭代，这便已经是非常大的计算代价。

首先应该做的事是去检查一个这么大规模的训练集是否真的必要，也许我们只用 1000 个训练集也能获得较好的效果，我们可以绘制学习曲线来帮助判断。



17.2 随机梯度下降法

参考视频: 17 - 2 - Stochastic Gradient Descent (13 min).mkv

如果我们一定需要一个大规模的训练集，我们可以尝试使用随机梯度下降法（**SGD**）来代替批量梯度下降法。

在随机梯度下降法中，我们定义代价函数为一个单一训练实例的代价：

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

随机梯度下降算法为：首先对训练集随机“洗牌”，然后：

Repeat (usually anywhere between1-10){

for $i=1:m${

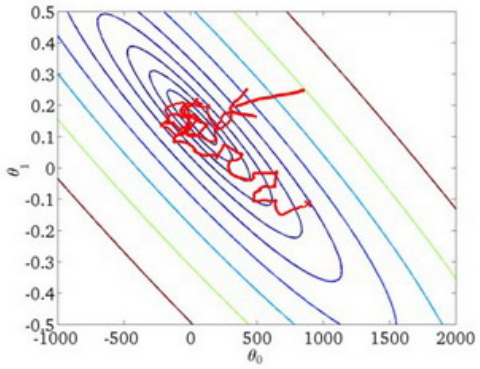
$\theta:=\theta_j-\alpha\left(h_\theta\left(x^{(i)}\right)-y^{(i)}\right)x_j{}^{(i)}$

(**for** $j=0:n$)

}

}

随机梯度下降算法在每一次计算之后便更新参数 θ ，而不需要首先将所有的训练集求和，在梯度下降算法还没有完成一次迭代时，随机梯度下降算法便已经走出了很远。但是这样的算法存在的问题是，不是每一步都是朝着”正确”的方向迈出的。因此算法虽然会逐渐走向全局最小值的位置，但是可能无法站到那个最小值的那一点，而是在最小值点附近徘徊。



17.3 小批量梯度下降

参考视频: 17 - 3 - Mini-Batch Gradient Descent (6 min).mkv

小批量梯度下降算法是介于批量梯度下降算法和随机梯度下降算法之间的算法，每计算常数b次训练实例，便更新一次参数 θ 。

Repeat {

for $i = 1:m${

$$\theta := \theta_j - \alpha \frac{1}{b} \sum_{k=i}^{i+b-1} \left(h_\theta \left(x^{(k)} \right) - y^{(k)} \right) x_j{}^{(k)}$$

(**for** $j = 0:n$)

$i+= 10$

}

}

通常我们会令 b 在 2-100 之间。这样做的好处在于，我们可以用向量化的方式来循环b个训练实例，如果我们用的线性代数函数库比较好，能够支持平行处理，那么算法的总体表现将不受影响（与随机梯度下降相同）。

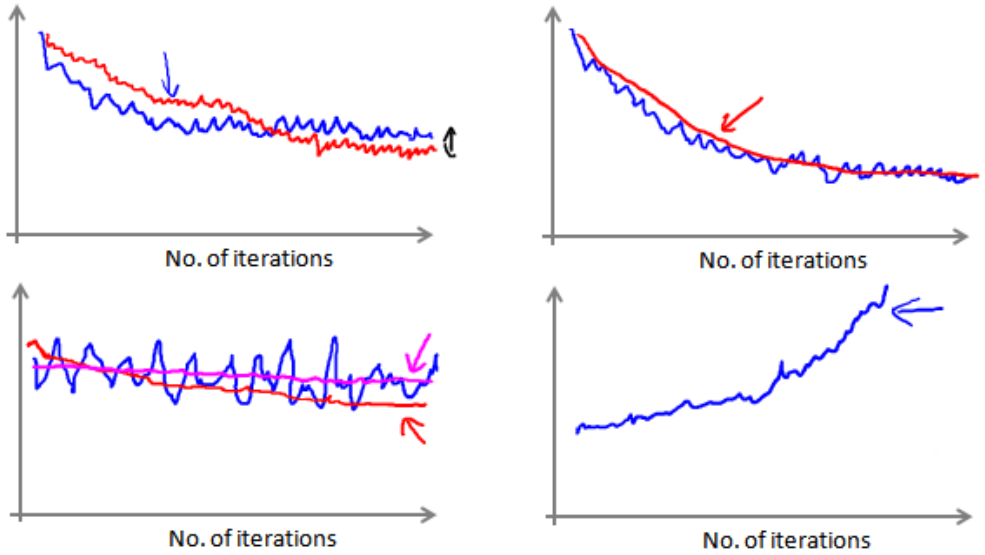
17.4 随机梯度下降收敛

参考视频: 17 - 4 - Stochastic Gradient Descent Convergence (12 min). mkv

现在我们介绍随机梯度下降算法的调试，以及学习率 α 的选取。

在批量梯度下降中，我们可以令代价函数J为迭代次数的函数，绘制图表，根据图表来判断梯度下降是否收敛。但是，在大规模的训练集的情况下，这是不现实的，因为计算代价太大了。

在随机梯度下降中，我们在每一次更新 θ 之前都计算一次代价，然后每x次迭代后，求出这x次对训练实例计算代价的平均值，然后绘制这些平均值与x次迭代的次数之间的函数图表。



当我们绘制这样的图表时，可能会得到一个颠簸不平但是不会明显减少的函数图像（如上面左下图中蓝线所示）。我们可以增加α来使得函数更加平缓，也许便能看出下降的趋势了（如上面左下图中红线所示）；或者可能函数图表仍然是颠簸不平且不下降的（如洋红色线所示），那么我们的模型本身可能存在一些错误。

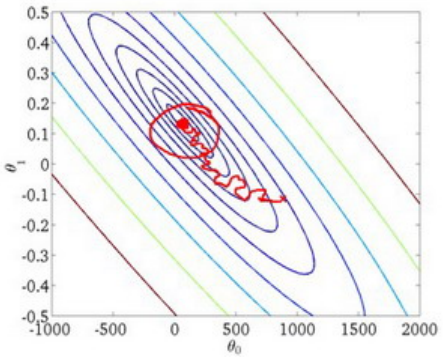
如果我们得到的曲线如上面右下方所示，不断地上升，那么我们可能会需要选择一个较小的学习率α。

我们也可以令学习率随着迭代次数的增加而减小，例如令：

$$\alpha = \frac{const1}{iterationNumber + const2}$$

随着我们不断地靠近全局最小值，通过减小学习率，我们迫使算法收敛而非在最小值附

近徘徊。 但是通常我们不需要这样做便能有非常好的效果了，对α进行调整所耗费的计算通常不值得



总结下，这段视频中，我们介绍了一种方法，近似地监测出随机梯度下降算法在最优化代价函数中的表现，这种方法不需要定时地扫描整个训练集，来算出整个样本集的代价函数，而是只需要每次对最后 1000 个，或者多少个样本，求一下平均值。应用这种方法，你既可以保证随机梯度下降法正在正常运转和收敛，也可以用它来调整学习速率α的大小。

17.5 在线学习

参考视频: 17 - 5 - Online Learning (13 min).mkv

在这个视频中，讨论一种新的大规模的机器学习机制，叫做在线学习机制。在线学习机制让我们可以模型化问题。

今天，许多大型网站或者许多大型网络公司，使用不同版本的在线学习机制算法，从大批的涌入又离开网站的用户身上进行学习。特别要提及的是，如果你有一个由连续的用户流引发的连续的数据流，进入你的网站，你能做的是使用一个在线学习机制，从数据流中学习用户的偏好，然后使用这些信息来优化一些关于网站的决策。

假定你有一个提供运输服务的公司，用户们来向你询问把包裹从 **A** 地运到 **B** 地的服务，同时假定你有一个网站，让用户们可多次登陆，然后他们告诉你，他们想从哪里寄出包裹，以及包裹要寄到哪里去，也就是出发地与目的地，然后你的网站开出运输包裹的的服务价格。比如，我会收取$50 来运输你的包裹，我会收取$20 之类的，然后根据你开给用户的这个价格，用户有时会接受这个运输服务，那么这就是个正样本，有时他们会走掉，然后他们拒绝购买你的运输服务，所以，让我们假定我们想要一个学习算法来帮助我们，优化我们想给用户开出的价格。

一个算法来从中学习的时候来模型化问题在线学习算法指的是对数据流而非离线的静态数据集的学习。许多在线网站都有持续不断的用户流，对于每一个用户，网站希望能在不将数据存储到数据库中便顺利地进行算法学习。

假使我们正在经营一家物流公司，每当一个用户询问从地点 A 至地点 B 的快递费用时，我们给用户一个报价，该用户可能选择接受（$y = 1$）或不接受（$y = 0$）。

现在，我们希望构建一个模型，来预测用户接受报价使用我们的物流服务的可能性。因此报价 是我们的一个特征，其他特征为距离，起始地点，目标地点以及特定的用户数据。模型的输出是:$p(y = 1)$。

在线学习的算法与随机梯度下降算法有些类似，我们对单一的实例进行学习，而非对一个提前定义的训练集进行循环。

Repeat forever (as long as the website is running) {

Get (x, y) corresponding to the current user

$\theta := \theta_j - \alpha(h_\theta(x) - y)x_j$

(**for** $j = 0:n$)

}

一旦对一个数据的学习完成了，我们便可以丢弃该数据，不需要再存储它了。这种方式的好处在于，我们的算法可以很好的适应用户的倾向性，算法可以针对用户的当前行为不断地更新模型以适应该用户。

每次交互事件并不只产生一个数据集，例如，我们一次给用户提供 3 个物流选项，用户选择 2 项，我们实际上可以获得 3 个新的训练实例，因而我们的算法可以一次从 3 个实例中学习并更新模型。

这些问题中的任何一个都可以被归类到标准的，拥有一个固定的样本集的机器学习问题中。或许，你可以运行一个你自己的网站，尝试运行几天，然后保存一个数据集，一个固定的数据集，然后对其运行一个学习算法。但是这些是实际的问题，在这些问题里，你会看到大公司会获取如此多的数据，真的没有必要来保存一个固定的数据集，取而代之的是你可以使用一个在线学习算法来连续的学习，从这些用户不断产生的数据中来学习。这就是在线学习机制，然后就像我们所看到的，我们所使用的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，又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。

17.6 映射化简和数据并行

参考视频: 17 - 6 - Map Reduce and Data Parallelism (14 min).mkv

映射化简和数据并行对于大规模机器学习问题而言是非常重要的概念。之前提到，如果我们用批量梯度下降算法来求解大规模数据集的最优解，我们需要对整个训练集进行循环，计算偏导数和代价，再求和，计算代价非常大。如果我们能够将我们的数据集分配给不多台计算机，让每一台计算机处理数据集的一个子集，然后我们将计所的结果汇总在求和。这样的方法叫做映射简化。

具体而言，如果任何学习算法能够表达为，对训练集的函数的求和，那么便能将这个任务分配给多台计算机（或者同一台计算机的不同 **CPU** 核心），以达到加速处理的目的。

例如，我们有 400 个训练实例，我们可以将批量梯度下降的求和任务分配给 4 台计算机进行处理：

Machine no	Samples	Computation	Result
1	1-100	$temp^{(1)} = \sum_{i=1}^{100}(h_\theta(x^i) - y^i)x_j^i$	$\theta_j = \theta_j - \alpha\frac{1}{400}(temp^{(1)} + temp^{(2)} + temp^{(3)} + temp^{(4)})$
2	101-200	$temp^{(2)} = \sum_{i=101}^{200}(h_\theta(x^i) - y^i)x_j^i$	
3	201-300	$temp^{(3)} = \sum_{i=201}^{300}(h_\theta(x^i) - y^i)x_j^i$	
4	301-400	$temp^{(4)} = \sum_{i=301}^{400}(h_\theta(x^i) - y^i)x_j^i$	

很多高级的线性代数函数库已经能够利用多核 **CPU** 的多个核心来并行地处理矩阵运算，这也是算法的向量化实现如此重要的缘故（比调用循环快）。