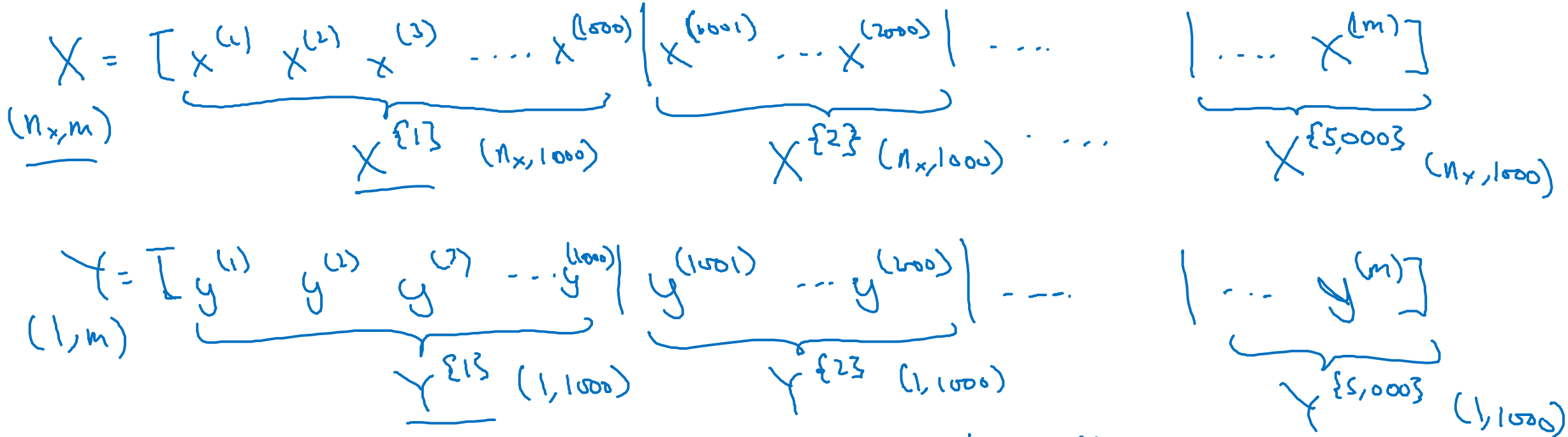
deeplearning.ai

Optimization Algorithms

Mini-batch gradient descent

Batch vs. mini-batch gradient descent

Vectorization allows you to efficiently compute on m examples.



What if $m = 5{,}000{,}000$?

5,000 mini-batches of 1,000 each

Mini-batch t: $X^{\{t\}}, Y^{\{t\}}$.

$x^{(i)}$

$z^{[l]}$

$X^{\{t\}}, Y^{\{t\}}$.

Andrew Ng

Mini-batch gradient descent

1 step of gradient descent using $X^{\{t\}}, Y^{\{t\}}$.
(as if $m = 1000$)

X, Y

Repeat {

for $t = 1, \ldots, 5000$ {

Forward prop on $X^{\{t\}}$.

$$Z^{[1]} = W^{[1]} X^{\{t\}} + b^{[1]}$$
$$A^{[1]} = g^{[1]}(Z^{[1]})$$
$$\vdots$$
$$A^{[L]} = g^{[L]}(Z^{[L]})$$

} Vectorized implementation (1000 examples)

Compute cost $J^{\{t\}} = \frac{1}{1000} \sum_{i=1}^{l} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2 \cdot 1000} \sum_{l} \|W^{[l]}\|_F^2$. (from $X^{\{t\}}, Y^{\{t\}}$)

Backprop to compute gradients wrt $J^{\{t\}}$ (using $(X^{\{t\}}, Y^{\{t\}})$)

$W^{[l]} := W^{[l]} - \alpha dW^{[l]}$, $b^{[l]} := b^{[l]} - \alpha db^{[l]}$

}

}

"1 epoch"
└ pass through training set.

Andrew Ng

欢迎回来 本周我们要学习 加快神经网络训练速度的优化算法 我之前说过机器学习的应用
是一个高度依赖经验的 不断重复的过程 你需要训练很多模型才能找到一个确实好用的 所以能够快速的训练模型的确是个优势 令情况更艰难的是 在大数据领域中深度学习表现得并不算完美 我们能够训练基于大量数据的神经网络 而用大量数据训练就会很慢 所以你会发现快速的优化算法 好的优化算法 的确能大幅提高你和你的团队的效率 那么让我们从小批量梯度下降算法
(mini-batch gradient descent)开始 我们之前学过 矢量化(vectorization)可以 让你有效地计算所有m个样例 而不需要一个具体的for循环
就能处理整个训练集 这就是为什么我们要将所有的训练样例集中到 这些巨型的矩阵X中去 就是x1 x2直到xm的m个训练样例 Y也做类似处理 y1 y2 y3直到ym 所以X为nx*m维矩阵,Y为1*m维矩阵 矢量化运算能够相对快地处理M个样例 如果M非常大 速度依然会慢 例如 如果M是5百万或者5千万或者更大 对你的整个训练集运用梯度下降法 你必须 先处理你的整个训练集 才能在梯度下降中往前一小步 然后再处理一次 整个5百万的训练集 才能再往前一小步 所以实际上算法是可以加快的 如果你让梯度下降
在处理完整个巨型的5百万训练集之前 就开始有所成效 具体来说 你可以这样做 首先将你的训练集拆分成更小的 微小的训练集 即小批量训练集(mini-batch) 比如说每一个微型训练集只有1000个训练样例 也就是说 取x1至x1000作为第一个微训练集 也叫做小批量训练集 然后取接下来的1000个样例 x1001至x2000这1000个样例 依次继续 我要引入一个新的符号 把这些表示为X{1} 这些表示为 x{2} 现在 如果你总共有5百万个训练样例 每个小批量样例有1000个样例 则你有5000个这样的小批量样例
因为5000乘1000是5百万 即总共有5000个小批量样例 所以最后一个为X{5000} Y也作类似处理 做相应的拆分处理 这个命名为Y{1}
接下来这个包含y1001至y2000 命名为Y{2}
依次拆分 最后得到Y{5000} 第T个小批量样例 包括X,T和Y。T 即对应1000个训练样例的输入输出对 在进行下一步之前
先再明确一下我用的标记符号 我们之前用小括号上标X(i)表示训练集中的 第i个样例 用上标中括号[l] 索引神经网络的不同层 所以z[l]表示神经网络第L层的z值 这里我们引入大括号{t}
代表不同的小批量样例 那么你就有了X{t} Y{t}
为了检验你对这些的理解 请问X{t}和Y{t}的维分别是多少 X的维度是nx*m 如果X{1}代表1000个样例
或者说1000个样例的x值 那么它的维度应该是nx*1000
X{2}的也是nx*1000 依此类推 即所有都是nx*1000维的 而这些应该是1*1000维的 要解释这个算法的名字 批量梯度下降(batch gradient descent) 可以先参考之前学习过的梯度下降法 该算法同时处理整个训练集 这个名字的由来就是 它同时处理整个训练集批次 我知道这不是一个特牛的名字
但是它就是这样叫的 小批量梯度下降 相对的是指下一页将要介绍到的算法 该算法每次只处理一个小批量样例X{t} Y{t} 而不是一次处理完整个训练集X Y 我们来看看小批量梯度下降是怎么做的 在训练集上运行小批量梯度下降法的时候 t=1到5000都要运行一遍
因为我们有5000个子集 每个子集1000个样例 for循环里要做的基本上就是 用(X{t},Y{t})做一次梯度下降 就好像你有一个规模为1000的训练集 而你只是要实现你已熟知的算法 只不过现在在你的m为1000的子训练集上 而不是为全部1000个样例写一个for循环 也就是说用矢量化的方法同时处理1000个样例 让我们先写出来 首先对输入值运用前向传播(Forward Prop) 也就是对X{t} 计算z[1]等于W(1) 之前这里只有X 对吧 但是现在你不是在处理整个训练集 只是处理第一个小批量训练集 所以当你处理第T个小批量训练集时
对应的X为X{t} 然后有A[1]=g[1](Z[1]) 这个Z是大写的 因为这里要表达一个矢量含义
依次继续 直到A[l]=g[l](Z[l]) 得到你的预测值 你应该已经注意到了 这些都是矢量化运算 只不过这个矢量化的算法 每次只处理1000个样例而不是5百万个

然后你要计算代价函数J 这里要除以1000 1000是你的子训练集的规模 对i=1到l的(^y(i),y(i))的损失值求和 说明一下 这个i是指 子训练集(X{t},Y{t})中的样例 如果你要正则化 可以加入这个正则化项 将2移至分母然后乘以 l次求和的
弗罗贝尼乌斯范数(Frobenius norm)的平方 因为这只是一个小型训练集的代价函数值 我用上标大括号{t}标识J 你应该已经注意到我们正在做的与 之前的梯度下降法的实现一模一样 只不过之前是训练X Y 而不是X{t}和Y{t} 接下来运用反向传播(Back Prop) 以计算J{t}的梯度 仍然只使用X{t}和Y{t} 然后再更新权重W 实际上是W[l] 减αdW[l] b也做类似操作 这是小批量梯度下降算法处理训练集一轮的过程 刚刚我写的代码
也叫做训练集的一次遍历(epoch) 遍历是指过一遍训练集 只不过在批量梯度下降法中 对训练集的一轮处理只能得到一步梯度逼近 而小批量梯度下降法中对训练集的一轮处理 也就是一次遍历 可以得到5000步梯度逼近 当然你会一如既往地想对 训练集进行多轮遍历 你可以用另一个for循环或者while循环实现 所以你不断地训练训练集 并希望它收敛在某个近似收敛值 当你有一个大型训练集时 小批量梯度下降法比梯度下降法要快得多 这几乎是每个从事深度学习的人 在处理一个大型数据集时会采用的算法 下一节我们将继续深入小型梯度下降算法 来进一步理解它在做什么和它为什么会奏效

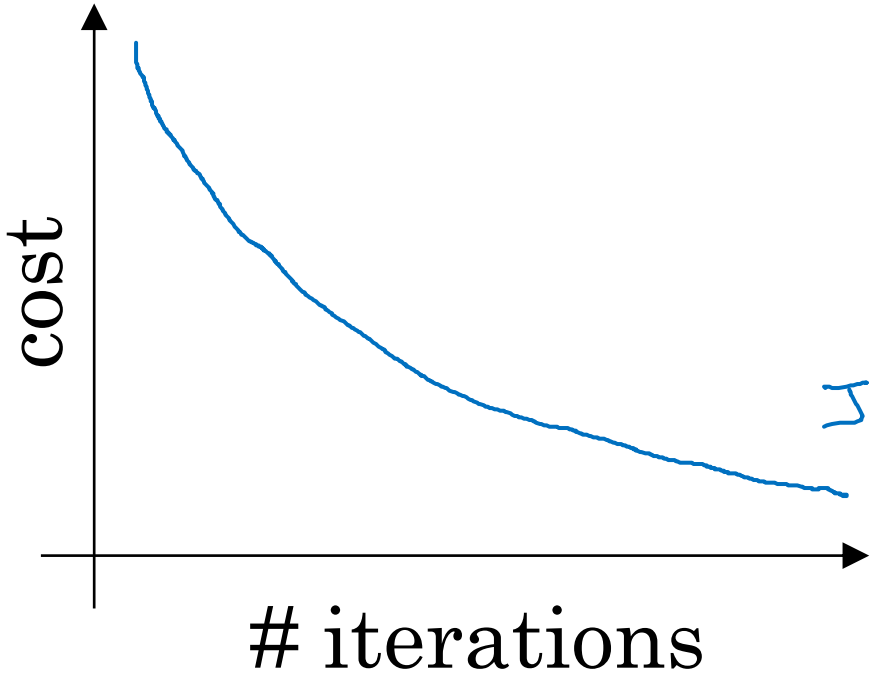
deeplearning.ai

Optimization Algorithms

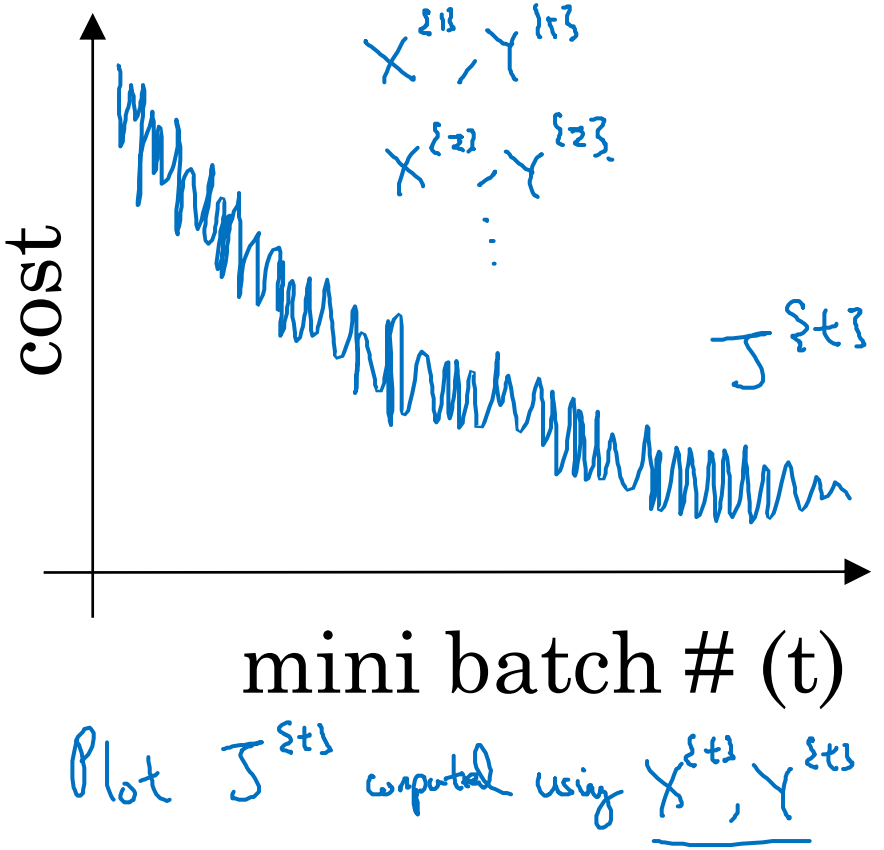
Understanding mini-batch gradient descent

Training with mini batch gradient descent

Batch gradient descent



Mini-batch gradient descent

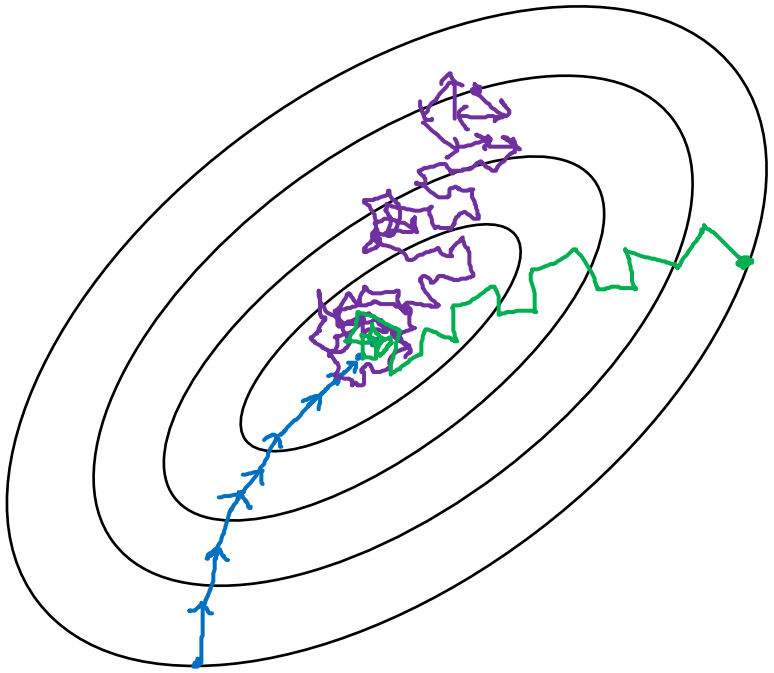


Choosing your mini-batch size

→ If mini-batch size = m : Batch gradient descent. $(X^{\{1\}}, Y^{\{1\}}) = (X, Y)$.

→ If mini-batch size = 1 : Stochastic gradient descent. Every example is it own mini-batch. $(X^{\{1\}}, Y^{\{1\}}) = (x^{(1)}, y^{(1)})$ ... $(x^{(2)}, y^{(1)})$

In practice: Somewhere in-between 1 and m



Stochastic gradient descent

{

Lose speedup from vectorization

In-between (mini-batch size not too big/small)

Fastest learning.

- Vectorization. (n 1000)
- Make progress without processing entire training set.

Batch gradient descent (mini-batch size = m)

Too long per iteration

Andrew Ng

Choosing your mini-batch size

If small train set: Use batch gradient descent.
($m \leq 2000$)

Typical mini-batch sizes:

→ 64, 128, 256, 512 (1024)
2^6, 2^7, 2^8, 2^9 (2^{10})

Make sure mini-batch fit in CPU/GPU memory.

$X^{\{t\}}, Y^{\{t\}}$

在上一节中 你学习了如何使用小批量梯度下降 以及在仅部分处理训练集数据时 运行梯度下降算法 在这一节中 你将进一步学习如何使用梯度下降 并进一步学习它究竟在做什么以及为什么有效 在批量梯度下降算法中
每一次迭代你将遍历整个训练集 并希望代价函数的值随之不断减小 如果我们用 J 来表示代价函数 那么它应该随着迭代单调递减 如果某一次迭代它的值增加了
那么一定是哪里错了 也许是你的学习率太大 而在小批量梯度下降中
同样画图就会发现 并不是每一次迭代代价函数的值都会变小 从细节来看 每次迭代 都是对X{t} Y{t}的处理 所以对通过它们计算出来的代价函数值 J{t} 进行画图 这就好像每次迭代你都使用不同的训练集 也就是使用不同的小块（mini-batch）所以如果你对代价函数 J 画图 你就会看到类似这样的 它的趋势是向下的 但是也会有许多噪声 如果使用小批量梯度下降算法 经过几轮训练后 对 J{t} 作图结果很可能就像这样 它并不一定每次迭代都会下降 但是整体趋势必须是向下的 而它之所以有噪声 可能和计算代价函数时使用的
那个批次X{t} Y{t}有关 让你的代价函数的值或大或小 也可能这个批次里含有一些标签标错的数据 导致代价函数有一些高 等等 这就是为什么在使用小批量梯度下降时 得到的代价函数图像有这样的震动 你必须定义的一个参数是mini-batch的大小 如果m是训练集的大小
一个极端的情况是 mini-batch的大小就等于m
这样其实就是批量梯度下降 在这种情况下你的mini-batch
只有一个X{1}和Y{1} 而它就等于你的整个训练集 所以如果把mini-batch的大小设置成m
你就得到了批量梯度下降 另一极端情况是把mini-batch的大小设为1 就会得到一种叫随机梯度下降的算法 这里每一条数据就是一个mini-batch 在这种情况下 先看第一个mini-batch X{1} Y{1} 因为mini-batch的大小是1
所以其实就是训练数据的第一个样本 你对此运行梯度下降算法 然后 第二个mini-batch 其实就是第二个样本
对它们再运行梯度下降算法 然后再用第三个样本做类似的工作 每次都只使用一组训练数据 下面让我们看一下这两种方法
在优化代价函数时有什么不同 这是你想要最小化的代价函数的等高线图 你的最小值在这里 批量梯度下降算法可能从这里开始 它的噪声相对小些 每一步相对大些 并且最终可以达到最小值 而相对的 随机梯度下降算法 让我们选一个不同的点 假使从这里开始 这时对于每一次迭代你就在一个样本上做梯度下降 大多数时候你可以达到全局最小值 但是有时候也可能因为某组数据不太好 把你指向一个错误的方向 因此随机梯度算法的噪声会非常大 一般来说它会沿着正确的方向 但是有事也会指向错误的方向 而且随机梯度下降算法
最后也不会收敛到一个点 它一般会在最低点附近摆动 但是不会达到并且停在那里 实际上 mini-batch的大小一般会在这2个极端之间 一个在1和m之间的值
因为1和m都太小或太大了 以下是原因 如果你使用批量梯度下降算法 你的mini-batch大小就是m 那么你将在每一次迭代中遍历整个训练集 这样做最大的缺点是
如果你的训练集非常大 就将在每一次迭代上花费太长的时间 如果你的训练集比较小
那它还是一个不错的选择 相反 如果你使用随机梯度下降算法 使用一个样本来更新梯度 这没有问题 而且可以通过选择比较小的学习率 来减少噪声 但随机梯度下降有一个很大的缺点是 你失去了可以利用向量加速运算的机会 因为这里你每次只处理一个训练数据 这是非常没有效率的
所以更好的做法是选择一个中间值 让mini-batch的大小既不太大也不太小 这样你的训练才能最快 这么做有两个好处 第一你可以使用向量的方式运算 在上一节的例子中

如果你的mini-batch大小是1000 你就可以用一个向量同时处理这1000个样本 这就比一个个的处理要快许多 第二你可以不用等待整个训练集 都遍历完一遍才运行梯度下降 还是用前一个视频中的数字 每遍历一遍训练集
可以执行5000次梯度下降算法 所以在实践中这样选择mini-batch的大小是最好的 如果使用小批量梯度下降算法 我们从这里开始 可能第一次迭代是这样 第二次 第三次 第四次 它并不能保证总是可以达到最小值 但是相比随机梯度下降 它的噪声会更小 而且它不会总在最小值附近摆动 如果有什么问题 你可以缓慢的减小学习率 我们会在下一节中介绍学习率衰减 以及如何降低学习率 如果mini-batch的大小既不能是m也不能是1 而是在它们之间的一个值 那么该如何选择呢? 这里有一些准则 第一 如果你的训练集较小
就使用批量梯度下降算法 这时候没有理由使用小批量梯度下降 因为你可以快速的处理整个训练集 所以使用批量梯度下降算法是没问题的 较小的定义我觉得是小于2000 这时使用批量梯度下降是非常适合的 否则 如果你有个更大的训练集 一般选择64到512作为mini-batch的大小 这是因为计算机内存的布局和访问方式 所以把mini-batch的大小设置为2的幂数
你的代码会运行的快一些 就像64是2的6次方 2的7次方 2的8次方 2的9次方 所以一般我会把mini-batch的大小
设置成2的幂数 我在上一节中我的mini-batch大小是1000 但我建议你就使用1024 2的10次方 但是1024这个值还是比较罕见的 而这些值更常用些 最后一个提醒是确保你的mini-batch你所有的X{t} Y{t}
是可以放进你CPU/GPU 内存的 当然这和你的配置 以及一个训练样本的大小都有关系 但是如果你使用的mini-batch超过了 CPU/GPU 内存的容量 不管你怎么做 你都会发现 结果会突然变得很糟 我希望以上说的能让你对
mini-batch大小的标准范围 有更好的了解 当然mini-batch的大小也是一个超参数 可能你要做一个快速的搜索去确定哪一个值 可以让代价函数J下降的最快 所以我的做法是尝试几个不同的值 尝试几个不同的2的幂数
然后看能否找到那个 让你的梯度下降算法尽可能效率的值 希望这能给你一些指导在 对这个超参的选择上 现在你知道了如何使用小批量梯度下降算法 并且在大训练集时让你的算法跑的更快 但是事实上还有一些比梯度下降 或者小批量梯度下降更高效的算法 我们将在下面几讲介绍它们