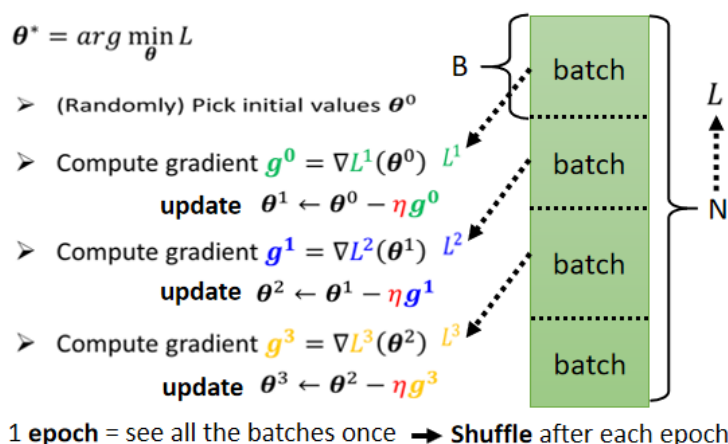


7月12日

一、李宏毅2021春机器学习课程第2.2节：Batch and Momentum

Review: Optimization with Batch



我们实际上在算微分的时候，并不是真的对所有 Data 算出来的 L 作微分，而是把所有的 Data 分成一个一个的 Batch，每次只针对一个 batch 的数据来计算 Loss，计算 gradient，并更新参数。每一个 batch 都看过一遍，叫做一个 Epoch。

在这里还需要提到的一个重要概念就是 shuffle，一个常见的做法就是，在每一个 Epoch 开始之前，会重新划分一次 Batch，这样的话每一个 Epoch 的 Batch 都不一样了。

至于为什么要 shuffle ->

- [机器学习，深度学习模型训练阶段的Shuffle重要么？为什么？NLP 翟-CSDN博客深度学习shuffle](#)
- [浅谈深度学习shuffle问题 不忘初心，方得始终CSDN博客训练不shuffle](#)

举一个简单的反例来说明，假设我们现在的网络最终的目的是做分类，现在我们手头有300个样本用作训练数据，前100个属于A类，中间100个属于B类，最后100个属于C类，如果不对训练数据进行打乱 (shuffle)，那么我们的模型很可能在很短的时间内就学会了以100为分界点来判断数据类型，但实际上我们的模型并没有学习到区分这些类的真正的特征。当然这只是不shuffle可能导致的其中一类问题，具体的描述可以参看上面的链接。

另外还必须指出的是，并不是所有的训练数据都需要shuffle，还是需要具体情况具体分析，就比如我们本来就想让模型学会某种次序关系或者我们希望模型对某部分数据表现的更好一点，那么我们则要根据自己的目的来决定数据的顺序，并决定是局部shuffle还是完全不shuffle。

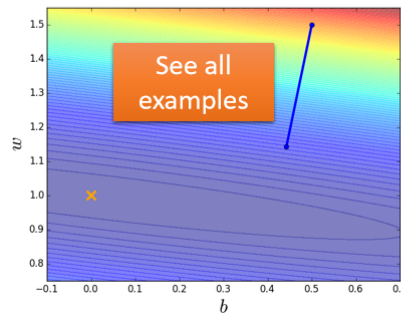
Small Batch VS Large Batch

首先我们来看看为什么需要使用batch：

Consider 20 examples ($N=20$)

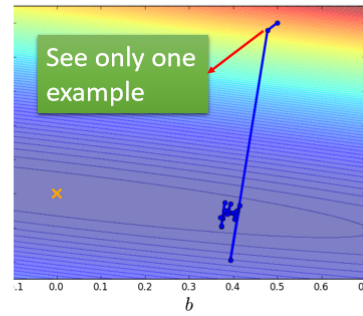
Batch size = N (Full Batch)

Update after seeing all
the 20 examples



Batch size = 1

Update for each example



这里给了一个比较极端的例子，假设现在我们有20笔训练资料：

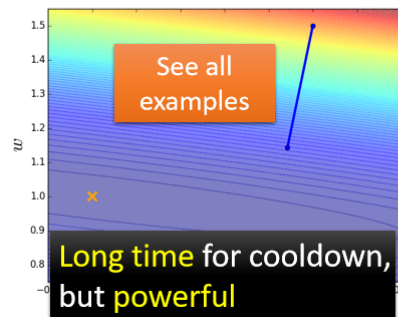
- 左边的 Case 就是没有用 Batch，也叫做 **Full Batch**。在这种情况下，我们的 Model 必须把20笔训练资料都看完，才能够计算一次 Loss，然后计算 Gradient，最后才能把模型的参数更新一次。
- 右边的 Case 就是Batch Size等于1，这也就是说我们只需要看过一笔资料就可以开始算 Loss，然后计算 Gradient，更新模型的参数。在每一个 Epoch 里面，我们的参数会 Update 20次。但是由于我们每次都只是看过一笔资料之后就更新模型参数，**显然这个更新是比较 Noisy 的**，所以观察右边的图可以发现我们 Update 的方向是很曲折的。

那我们可能要问，左边和右边哪一个比较好呢？他们有什么差别呢？

Consider 20 examples ($N=20$)

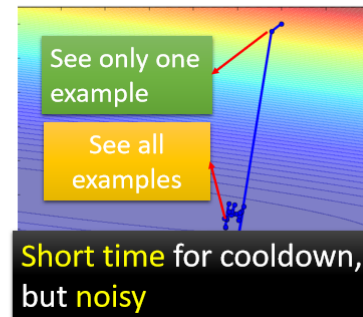
Batch size = N (Full Batch)

Update after seeing all
the 20 examples



Batch size = 1

Update for each example
Update 20 times in an epoch



直观上我们可以发现，左边要看完全部的资料之后，才会进行一次 Update，这样**参数更新的速度应该比较慢**，但是因为看过了所有的资料，所以这个**更新是很稳健的，是兼听则明的**。而右边每看完一笔资料就进行一次 Update，这样**参数更新的速度应该很快**，但是因为只看过一笔资料就做出判断，所以这个**更新是很不稳定的，是偏信则暗的**。

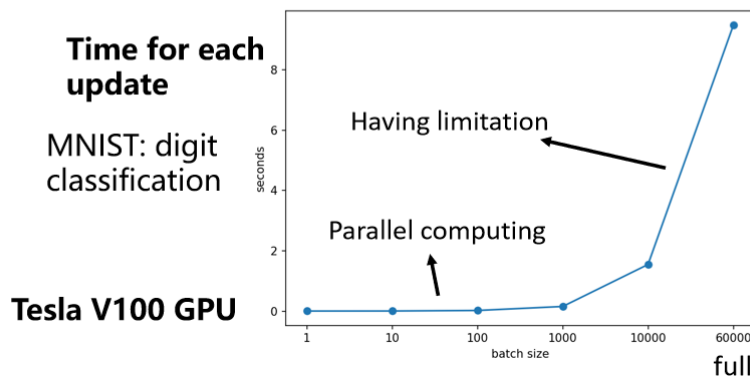
但是接下来我们要看到的是，**如果考虑并行运算的话，左边的情况需要的时间并不一定比右边长。**

Larger batch size does not require longer time to compute gradient

为了说明上面的观点，这里有一个在MNIST数据集上采用不同的batch size做训练的真实的例子：

Small Batch v.s. Large Batch

- Larger batch size does not require longer time to compute gradient (unless batch size is too large)

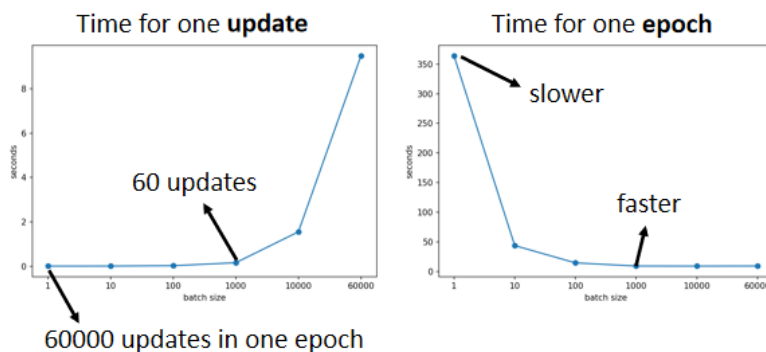


这个实验的目的就是我们想知道，给机器一个 batch size，它要计算出 Gradient，进而 Update 参数，到底需要耗费多长时间。你会发现 **Batch Size 从1到1000，需要耗费的时间几乎是一样的**，尽管从直觉上来说，对1000笔资料计算 Loss，然后计算 Gradient，花的时间应该是对一笔资料做同样操作所需时间的1000倍。

这是因为在**实际上做运算的时候，我们有 GPU 来做并行运算**，这1000笔资料实际是并行处理的，所以处理1000笔资料所花的时间，并不是一笔资料的1000倍。当然 **GPU 并行运算的能力也是有极限的**，这也就是曲线后半部分迅速上升的原因。

Smaller batch requires longer time for one epoch

- Smaller batch requires longer time for one epoch (longer time for seeing all data once)

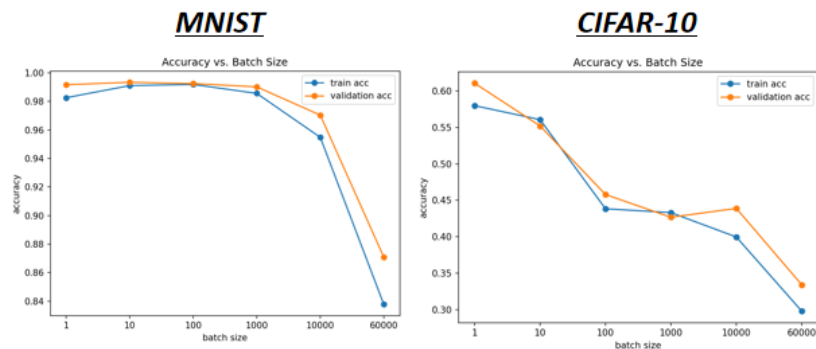


由于 GPU 提供了并行运算的能力，**对于每个 update 来说，小的 batch size 和稍大的 batch size 所需要花费的时间其实相差不多**。但当我们关注每一个 epoch 需要花费的时间时，你会发现右边的曲线：小的 batch size 跑完一个 Epoch 所花的时间往往比大的 batch size 要多。

假设我们的训练资料有60000笔，如果 Batch Size 设为1，那你需要60000个 update 才能跑完一个 Epoch，如果 Batch Size 设为 1000，那你需要60个 update 跑完一个 Epoch。由于对每个 update 来说，在并行运算可用的情况下小的 batch size 和稍大的 batch size 所需要花费的时间差不多，那**此时 60000次 update 跟60次 update 比起来，所需要耗费的时间相差就很大了**。

那这样看来，似乎大的 batch size 在并行运算的条件下消除了劣势，已经很完美了，是不是就不再需要小的 batch size 出场了呢？但其实不然，这里还有一个很神奇的事情，小的 batch size 情况下**Noisy 的 Gradient**，反而可以帮助 Training，这个也是跟直觉正好相反的。

还是来看一个实际的例子，下图是用不同的 batch size 分别在 MNIST 和 来 CIFAR-10 上训练你的模型得到的实验结果。横轴代表的是 batch size，从左到右越来越大，纵轴代表的是正确率，越往上正确率越高。



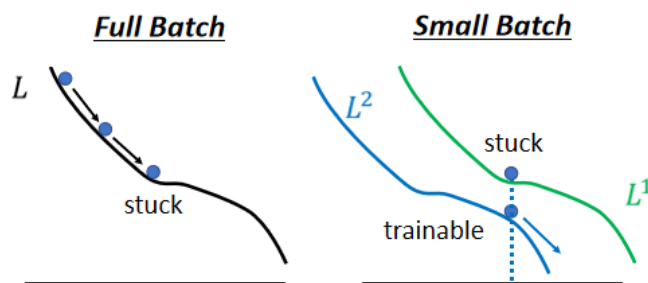
- Smaller batch size has better performance
- What's wrong with large batch size? Optimization Issue

通过上面的实验结果我们可以看到，大的 batch size，往往在 Training 的时候，会给你带来比较差的结果。这个是 Optimization 的问题，代表当你用大的 batch size 的时候,你的 Optimization 可能会有问题，小的 batch size，Optimization 的结果反而是比较好的，那么这又是为什么呢？

“Noisy” update is better for training

一个可能的解释是这样子的：

- Smaller batch size has better performance
- “Noisy” update is better for training



假设你是 Full Batch，那你在 Update 你的参数的时候，你就是沿着同一个 Loss Function 来 Update 参数，这样如果你的训练走到了一个 local minima 或者 saddle point，如果你不特别去看 Hessian 的话，你的训练就停下来了。

但是假如是 Small Batch 的话，因为我们每次是挑一个 Batch 出来，算它的 Loss，所以在这种情况下，每一次 Update 你的参数的时候，你用的 Loss Function 都是不一样的，你选到第一个 Batch 的时候，你是用 L^1 来算你的 Gradient，你选到第二个 Batch 的时候，你是用 L^2 来算你的 Gradient。假设你用 L^1 算 Gradient 的时候，发现 Gradient 是零，卡住了，但 L^2 它的 Function 跟 L^1 又不一样， L^2 就不一定会卡住，所以你还是想办法让你的 Loss 继续变小，所以今天这种 Noisy 的 Update 的方式，结果反而对 Training 是有帮助的。

“Noisy” update is better for generalization

这边还有另外一个更神奇的事情，**小的 Batch Size 对 Testing 也是有帮助的。**

以下这个实验结果是引用自，On Large-Batch Training For Deep Learning, Generalization Gap And Sharp Minima <https://arxiv.org/abs/1609.04836>，这篇 Paper 的实验结果：

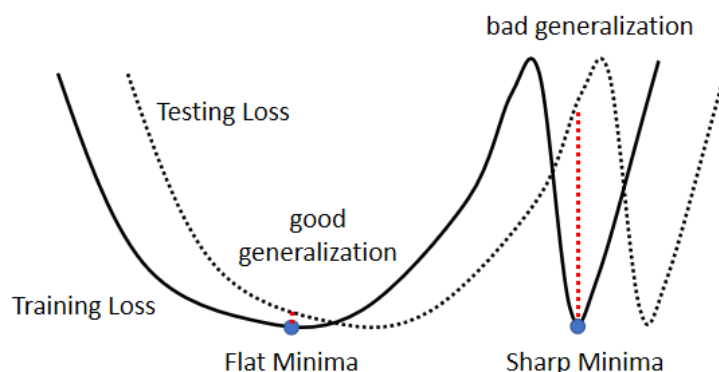
• “Noisy” update is better for generalization

	Name	Network Type	Data set	
			SB	LB
SB = 256	F_1	Fully Connected	MNIST (LeCun et al., 1998a)	
	F_2	Fully Connected	TIMIT (Garofolo et al., 1993)	
LB = 0.1 x data set	C_1	(Shallow) Convolutional	CIFAR-10 (Krizhevsky & Hinton, 2009)	
	C_2	(Deep) Convolutional	CIFAR-10	
	C_3	(Shallow) Convolutional	CIFAR-100 (Krizhevsky & Hinton, 2009)	
	C_4	(Deep) Convolutional	CIFAR-100	

Name	Training Accuracy		Testing Accuracy	
	SB	LB	SB	LB
F_1	99.66% \pm 0.05%	99.92% \pm 0.01%	98.03% \pm 0.07%	97.81% \pm 0.07%
F_2	99.99% \pm 0.03%	98.35% \pm 2.08%	64.02% \pm 0.2%	59.45% \pm 1.05%
C_1	99.89% \pm 0.02%	99.66% \pm 0.2%	80.04% \pm 0.12%	77.26% \pm 0.42%
C_2	99.99% \pm 0.04%	99.99% \pm 0.01%	89.24% \pm 0.12%	87.26% \pm 0.07%
C_3	99.56% \pm 0.44%	99.88% \pm 0.30%	49.58% \pm 0.39%	46.45% \pm 0.43%
C_4	99.10% \pm 1.23%	99.57% \pm 1.84%	63.08% \pm 0.5%	57.81% \pm 0.17%

那这篇 Paper 里面，作者 Train 了六个不同的 Network，来表示这个实验是很泛用的，在很多不同的 Case 下都观察到一样的结果。然后他想办法，在大的 Batch 跟小的 Batch，都 Train 到差不多的 Training 的 Accuracy，但是就算是在 **Training 的时候结果差不多，Testing 的时候你还是会发现，大的 Batch Testing Accuracy 比大的 Batch 要差。** Training 的时候好而 Testing 的时候差，也就说明大的 Batch 遇到了 **Over Fitting** 的问题，至于为什么会有这样的结果，在这篇文章里也给出了一个可能的解释：

• “Noisy” update is better for generalization



假设这个是我们的 Training Loss，那在这个 Training Loss 上面可能有很多个 Local Minima，这些 Local Minima 的 Loss 都很低，但是 **Local Minima 其实也有有好 Minima 跟坏 Minima 之分。**

如果一个 Local Minima 它在一个狭窄的峡谷里面，它是坏的 Minima；如果它在一个宽阔的盆地上，那它是好的 Minima。那为什么我们要这样界定 minima 的好坏呢？




假设现在 Training 跟 Testing 中间，有一个 Mismatch，也就是说 Training 的 Loss Function 跟 Testing 的 Loss Function 不太一样，这有可能是本来你 Training 跟 Testing 的 Distribution 就不一样，也可能是因为 Training 和 Testing 本身采样到的数据就有差异。这里我们就**假设说这个 Training 跟 Testing 之间的差距就是把 Training 的 Loss Function 往右平移一点**，这时候你会发现，对左边这个在一个**盆地**里面的 Minima 来说，它在 Training 跟 Testing 上面的结果不会差太多，但是对右边这个在**峡谷**里面的 Minima 来说，就是“失之毫厘，谬以千里”了。

很多人相信**大的 Batch Size**，会让我们倾向于走到峡谷里面，而**小的 Batch Size**，会让我们倾向于走到**盆地里**面。直觉上的想法是这样，小的 Batch，它有很多的 Loss，每次 Update 的方向都不太一样，是比较 Noisy 的，所以如果这个峡谷非常窄，它可能一个不小心就跳出去了，最终一般是在一个非常宽的盆地才能停下来。而大的 Batch 相对而言就比较容易被困在峡谷中。

当然这只是一个可能的解释，这个问题其实还是一个**尚待研究的问题**。

Have both fish and bear's paws?


用一张图来展示一下大的 batch size 和小的 batch size 的区别：

	Small	Large
Speed for one update (no parallel)	Faster	Slower
Speed for one update (with parallel)	Same	Same (not too large)
Time for one epoch	Slower	Faster 
Gradient	Noisy	Stable
Optimization	Better 	Worse
Generalization	Better 	Worse

Batch size is a hyperparameter you have to decide.

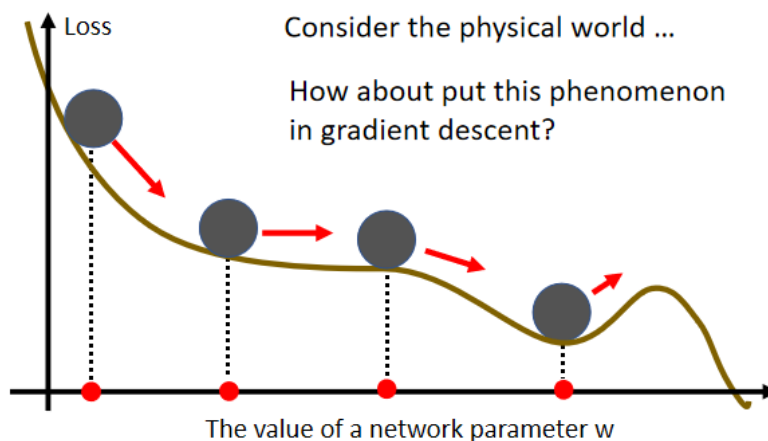
我们可能会去思考的一点是：能不能够鱼与熊掌兼得呢？我们能不能用大的 Batch Size 来做训练，利用 GPU 并行运算的能力来增加训练的效率，但是同时又能在测试集上得到好的结果呢？**这是有可能的，有很多文章都在探讨这个问题**，在下面列出的这些 Paper 里面，为什么他们可以把网络训练的那么快，就是因为他们用的 **Batch Size 是真的很大**，利用并行运算的优势你可以在很短的时间内看到大量的资料，但同时他们也需要有一些特别的方法来解决 Batch Size 太大可能会带来的劣势。

Have both fish and bear's paws?

- Large Batch Optimization for Deep Learning: Training BERT in 76 minutes (<https://arxiv.org/abs/1904.00962>)
- Extremely Large Minibatch SGD: Training ResNet-50 on ImageNet in 15 Minutes (<https://arxiv.org/abs/1711.04325>)
- Stochastic Weight Averaging in Parallel: Large-Batch Training That Generalizes Well (<https://arxiv.org/abs/2001.02312>)
- Large Batch Training of Convolutional Networks (<https://arxiv.org/abs/1708.03888>)
- Accurate, large minibatch sgd: Training imagenet in 1 hour (<https://arxiv.org/abs/1706.02677>) 

Momentum

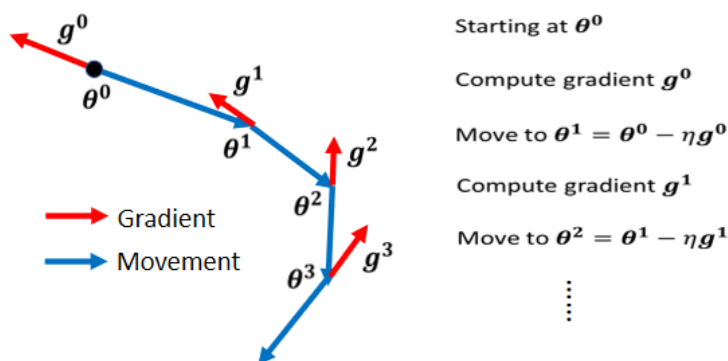
这是**另外一个有可能可以对抗 Saddle Point 和 Local Minima 的技术**，Momentum 的运作方式是这样的：



想像我们是在**物理的世界**里面，假设 **Error Surface 就是真正的斜坡**，而我们的**参数是一个球**，你把球从斜坡上滚下来，如果我们采用的是**传统的 Gradient Descent**，那它走到 **Critical Point** 就停住了。

但是在物理的世界里，一个球如果从高处滚下来，就算滚到 Saddle Point，因为**惯性的原因**它还是会继续往右走，如果**动量够大**，甚至它能够翻过 **Local Minima** 的山坡继续往右走。把这个概念运用到我们的机器学习中，就是接下来要提到的 Momentum 技术。

传统的 Gradient Descent 如下图所示：

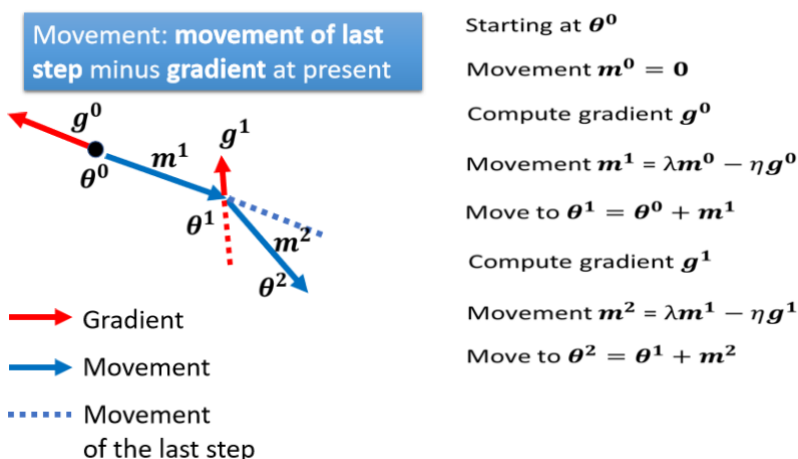


我们有一个初始的参数叫做 θ^0 ，我们计算一下 Gradient，然后计算完这个 Gradient 以后，我们往 Gradient 的反方向去 Update 参数

$$\theta^1 = \theta^0 - \eta g^0$$

我们到了新的参数以后，再计算一次 Gradient，再往 Gradient 的反方向 Update 一次参数，这个过程就一直这样持续下去。可以看到**在这种情况下，我们的参数 Update 的方向只取决于当前计算出的 Gradient**，如果当前到达了 Critical Point，由于 Gradient 为 0，我们的参数就没办法继续更新了。

加上 Momentum 以后，每一次在移动参数的时候，我们不是只往 Gradient 的反方向来移动参数，我们是利用 **Gradient 的反方向加上前一步移动的方向的结果**，来调整我们的参数。



每一步的移动，我们都用 m 来表示，那这个 m 其实可以写成之前所有算出来的 Gradient 的 Weighted Sum。

Movement: movement of last step minus gradient at present

m^i is the weighted sum of all the previous gradient: g^0, g^1, \dots, g^{i-1}

$$m^0 = 0$$

$$m^1 = -\eta g^0$$

$$m^2 = -\lambda \eta g^0 - \eta g^1$$

\vdots

Starting at θ^0

Movement $m^0 = 0$

Compute gradient g^0

Movement $m^1 = \lambda m^0 - \eta g^0$

Move to $\theta^1 = \theta^0 + m^1$

Compute gradient g^1

Movement $m^2 = \lambda m^1 - \eta g^1$

Move to $\theta^2 = \theta^1 + m^2$

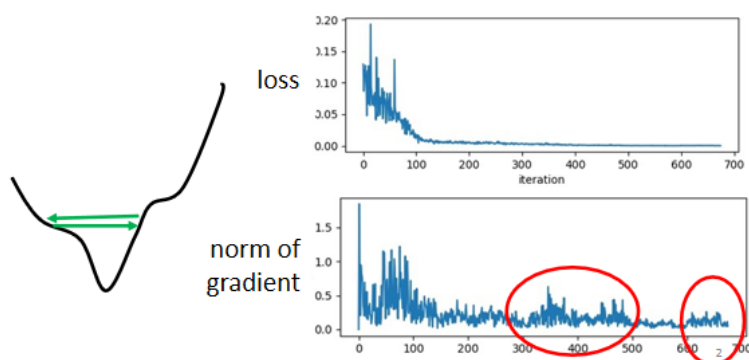
Movement not just based on gradient, but previous movement.

现在你会发现对于 Momentum，一个解读是：Gradient 的反方向加上前一次移动的方向。另外一个解读方式是：当加上 Momentum 之后，我们 Update 的方向，不是只考虑现在的 Gradient，而是还要考虑过去所有 Gradient 的总和。

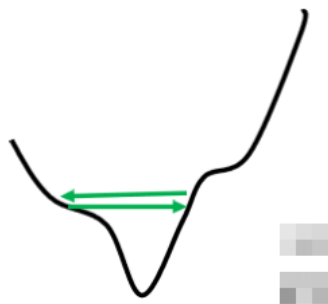
二、李宏毅2021春机器学习课程第2.3节：Adaptive Learning Rate

Training stuck \neq Small Gradient

首先要明确的一点是，目前当我们用 gradient descend 来做 optimization 的时候，你真正应该要怪罪的对象往往不是 critical point，而是其他的原因。

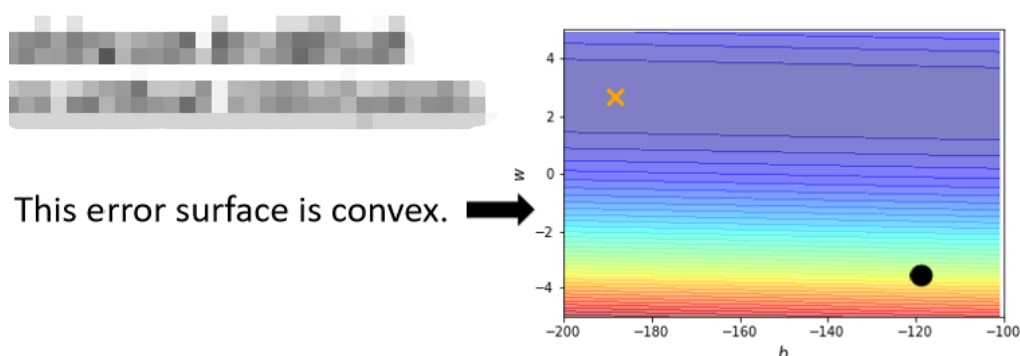


我们之前常说，走到 critical point 的时候，意味着 gradient 非常小，但是你真的确认过，当你的 loss 不再下降的时候，gradient 真的很小吗？事实上在上图这个例子中，当我们的 loss 不再下降的时候，gradient 并没有真的变得很小，那为什么会出现这样的状况呢？



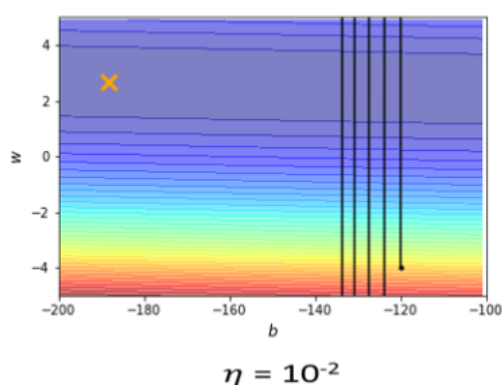
上图是我们的error surface，而你每次更新model的参数，其实只是在error surface山谷的两个谷壁间不断的来回震荡，这个时候你的loss不会再下降，所以你会觉得它肯定卡在了critical point，但实际上，它的gradient仍然很大，只是因为不断震荡，loss不能够再减小了。

既然critical point不是主要问题的话，那为什么我们的training还是会卡住呢？这里举一个非常简单的例子，下图是一个非常简单的error surface：



这个error surface的最低点在黄叉×这个地方，可以观察到它的这个等高线是椭圆形的，只是它在横轴的地方坡度的变化非常小，所以这个椭圆的长轴非常长，短轴相对之下比较短，在纵轴的地方gradient的变化很大，error surface的坡度非常的陡峭。现在我们要从黑点这个地方来做gradient descend。

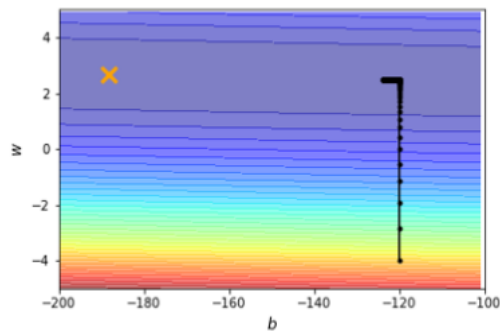
你可能会觉得，这个要做gradient descend应该很简单，不就是一路滑下来，然后再往左走过去嘛。但要是你实际上自己试一下，你就会发现即使是形状这么简单的error surface，用gradient descend都不见得能把它做好，举例来说：



learning rate设为 10^{-2} 的时候，参数在峡谷的两端不断震荡，loss始终掉不下去，但是gradient其实仍然是很大的，这与本文开头那个例子的情况相同。

那这时你可能说，肯定是因为learning rate设的太大，learning rate决定了我们update参数的时候步伐有多大，只要把learning rate设小一点，步伐小一点慢慢滑入山谷，不就可以解决这个问题了吗？

但事情没有想的那么简单，慢慢调整这个learning rate，从 10^{-2} 一直调到 10^{-7} ，调到 10^{-7} 之后，震荡终于停下了。



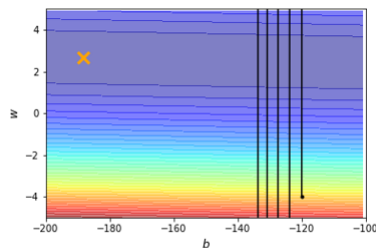
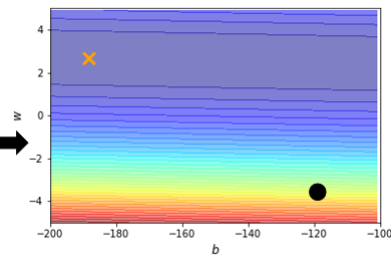
$$\eta = 10^{-7}$$

但是这个时候我们又发现，这个训练似乎永远走不到终点，因为**learning rate已经太小了**，竖直往上这一段因为坡度很陡，gradient的值很大，所以还能够前进一点，但左拐以后这个地方坡度已经非常的平滑了，**这么小的learning rate，根本没有办法再让我们的训练前进。**

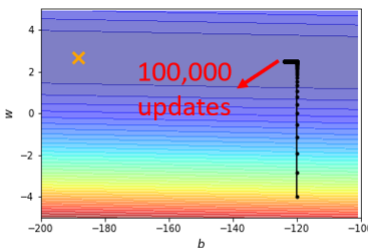
Training can be difficult even without critical points.

This error surface is convex. →

Learning rate cannot be one-size-fits-all



$$\eta = 10^{-2}$$



$$\eta = 10^{-7}$$

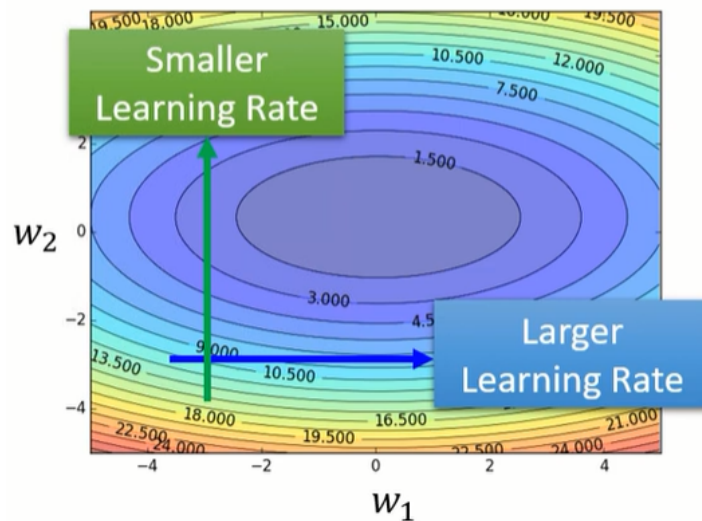
4

事实上在左拐这个地方，可以看到有一大堆黑点，实际上这些黑点总共有**十万个**，可以看到即使我们update了十万次，还是没有让我们的训练前进多少，这个时候其实我们希望此时的learning rate能大一点，总的来说，**我们希望模型的learning rate是能够自适应调整的。**

在之前我们使用的gradient descend里面，所有的参数都是设置同样的learning rate，这显然是不够的，我们应该要**能为每一个参数定制learning rate**，并且**learning rate还能够根据训练的进度进行自适应的调整。**

Different parameters needs different learning rate

从刚才的例子里面，其实我们可以总结出一个简单的原则：**如果在某一个方向上，我们的gradient的值很小，非常的平坦，那我们会希望learning rate调大一点；如果在某一个方向上gradient的值很大，坡度很大，那我们会希望learning rate可以设得小一点。**



那我们要如何实现learning rate的自动调整呢？

我们需要修改一下gradient descend原来的式子，这边为了简化问题，我们只看一个参数上的update，但是你完全可以把这个方法推广到所有参数的状况：

$$\theta_i^{t+1} \leftarrow \theta_i^t - \eta g_i^t$$

这个式子中的 η 其实就是我们的learning rate，在这里我们看到这个 η 的值是固定不变的，也就是说我们的learning rate是固定的。

现在我们要有一个能随着参数而变化的learning rate，我们把原来learning rate η 这一项呢，改写成 $\frac{\eta}{\sigma_i}$

$$\theta_i^{t+1} \leftarrow \theta_i^t - \frac{\eta}{\sigma_i} g_i^t$$

这个 σ_i 你发现它有一个上标 t ，有一个下标 i ，这代表说这个 σ 这个参数，首先它是 depend on i 的，不同的参数我们要给它不同的 σ ，同时它也是 iteration dependent 的，不同的 iteration 我们也会有不同的 σ 。接下来我们要看这个 σ_i 有哪些常见的计算方式。

Root mean square

Root Mean Square $\theta_i^{t+1} \leftarrow \theta_i^t - \boxed{\frac{\eta}{\sigma_i^t}} g_i^t$

$$\theta_i^1 \leftarrow \theta_i^0 - \frac{\eta}{\sigma_i^0} g_i^0 \quad \sigma_i^0 = \sqrt{(g_i^0)^2} = |g_i^0|$$

$$\theta_i^2 \leftarrow \theta_i^1 - \frac{\eta}{\sigma_i^1} g_i^1 \quad \sigma_i^1 = \sqrt{\frac{1}{2} [(g_i^0)^2 + (g_i^1)^2]}$$

$$\theta_i^3 \leftarrow \theta_i^2 - \frac{\eta}{\sigma_i^2} g_i^2 \quad \sigma_i^2 = \sqrt{\frac{1}{3} [(g_i^0)^2 + (g_i^1)^2 + (g_i^2)^2]}$$

$$\vdots$$

$$\theta_i^{t+1} \leftarrow \theta_i^t - \frac{\eta}{\sigma_i^t} g_i^t \quad \sigma_i^t = \sqrt{\frac{1}{t+1} \sum_{i=0}^t (g_i^t)^2}$$

第 $t+1$ 次 update 参数的时候， σ_i^t 就是过去所有的 gradient， g_i^t 从第一步到目前为止，所有算出来的 g_i^t 的平方和求平均值再开根号得到 σ_i^t 。

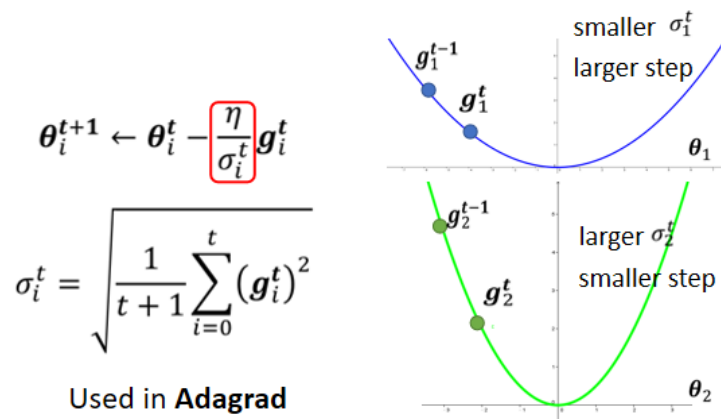
$$\sigma_i^t = \sqrt{\frac{1}{t+1} \sum_{i=0}^t (g_i^t)^2}$$

然后再用之前的learning rate除以 σ_i^t ，就得到了新的learning rate来更新你的参数。

$$\theta_i^{t+1} \leftarrow \theta_i^t - \frac{\eta}{\sigma_i^t} g_i^t$$

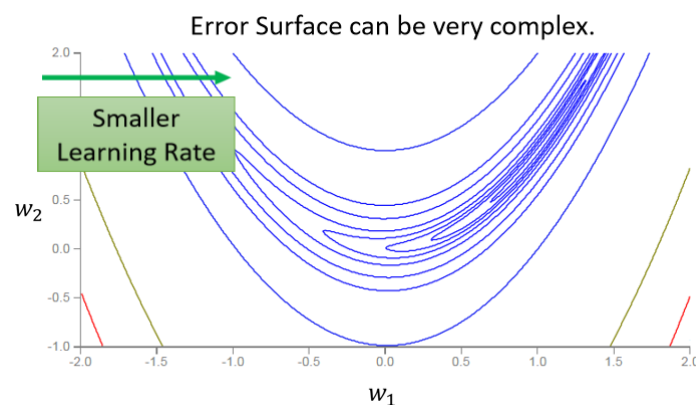
Adagrad

刚刚的这一招被用在一个叫做Adagrad的方法里，因为我们的 σ_i 这一项在分母上，所以如果某个参数 θ_1 的坡度小，那么在 θ_1 这个参数上面，算出来的gradient就小，所以算出来的 σ 就小，而 σ 小 learning rate就大。同理可以想到某个参数 θ_2 的坡度大，则最终的 learning rate就小。**这就可以做到我们刚才讲的，坡度比较大的时候，learning rate就减小，坡度比较小的时候，learning rate就放大的效果。**

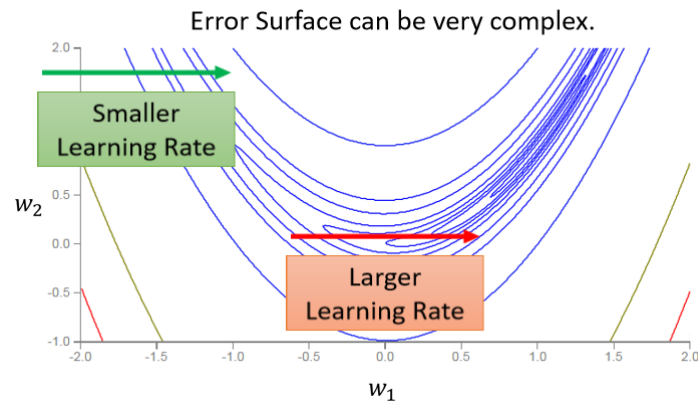


RMSProp

在刚刚的Adagrad的方法中，如果对于同一个参数，它的gradient的大小，就会差不多是固定的值，但事实上并不一定是这样的，举例来说我们来看下面这个新月形的error surface：



如果我们考虑横轴的话，你会发现在绿色箭头这个地方坡度**比较陡峭**，所以我们需要**比较小的learning rate**。



但是走到了中间这一段，到了红色箭头的时候呢，坡度又变得平滑了起来，这就需要比较大的learning rate，所以就算是同一个参数同一个方向，我们也希望learning rate是可以动态调整的，这也就是RMSProp要做的。

RMS Prop这个方法有点传奇,它传奇的地方在于它找不到论文,非常多年前应该是将近十年前,Hinton在Coursera上,开过deep learning的课程,那个时候他在他的课程里面,讲了RMS Prop这个方法,然后这个方法没有论文,所以你要cite的话,你要cite那个影片的连接,这是个传奇的方法叫做RMS Prop

$$\begin{aligned}
 &\text{RMSProp} \quad \theta_i^{t+1} \leftarrow \theta_i^t - \frac{\eta}{\sigma_i^t} g_i^t \\
 &\theta_i^1 \leftarrow \theta_i^0 - \frac{\eta}{\sigma_i^0} g_i^0 \quad \sigma_i^0 = \sqrt{(g_i^0)^2} \quad 0 < \alpha < 1 \\
 &\theta_i^2 \leftarrow \theta_i^1 - \frac{\eta}{\sigma_i^1} g_i^1 \quad \sigma_i^1 = \sqrt{\alpha(\sigma_i^0)^2 + (1 - \alpha)(g_i^1)^2} \\
 &\theta_i^3 \leftarrow \theta_i^2 - \frac{\eta}{\sigma_i^2} g_i^2 \quad \sigma_i^2 = \sqrt{\alpha(\sigma_i^1)^2 + (1 - \alpha)(g_i^2)^2} \\
 &\vdots \\
 &\theta_i^{t+1} \leftarrow \theta_i^t - \frac{\eta}{\sigma_i^t} g_i^t \quad \sigma_i^t = \sqrt{\alpha(\sigma_i^{t-1})^2 + (1 - \alpha)(g_i^t)^2}
 \end{aligned}$$

刚才在Adagrad中算Root Mean Square的时候，每一个gradient都有同等的重要性，但在RMS Prop里面，是现在刚算出来的 g_i^t 比较重要还是之前算出来的gradient比较重要，是通过调整这个 α 的值来决定的。

这个 α 也就成为了一个hyperparameter：

- 如果把 α 设很小趋近于0，就代表我觉得现在刚算出来的 g_i^t 比较重要。
- 我 α 设很大趋近于1，那就代表我觉得现在算出来的 g_i^t 比较不重要，之前算出来的gradient比较重要。

$$\sigma_i^t = \sqrt{\alpha(\sigma_i^{t-1})^2 + (1 - \alpha)(g_i^t)^2}$$

Adam

目前最常用的optimization的策略就是Adam：

Adam: RMSProp + Momentum

Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t .

Require: α : Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0 : Initial parameter vector

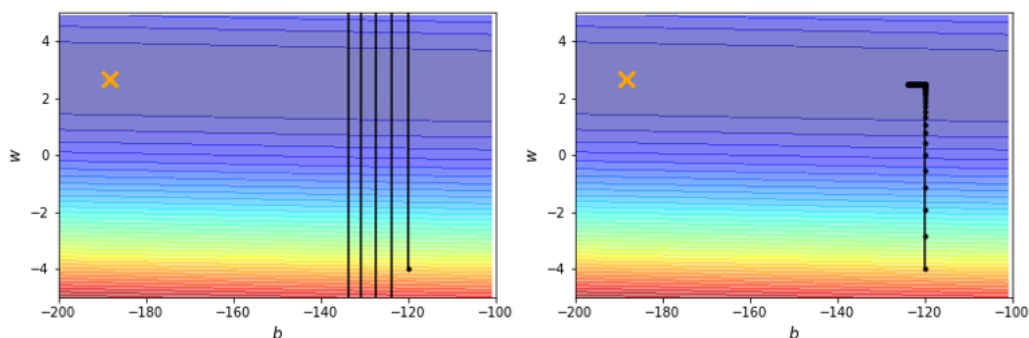
$m_0 \leftarrow 0$ (Initialize 1st moment vector) \rightarrow for momentum
 $v_0 \leftarrow 0$ (Initialize 2nd moment vector) \rightarrow for RMSprop
 $t \leftarrow 0$ (Initialize timestep)

while θ_t not converged **do**
 $t \leftarrow t + 1$
 $g_t \leftarrow \nabla_{\theta} f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
 $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
 $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
 $\hat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
 $\hat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
 $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon)$ (Update parameters)
end while
return θ_t (Resulting parameters)

Adam就是RMSProp加上Momentum，技术的进步，使得傻瓜式操作（Adam）就可以得到不错的效果，但是在特定的场景下，要拍出最好的效果，依然需要深入地理解光线、理解结构、理解器材（SGD慢慢调优）。

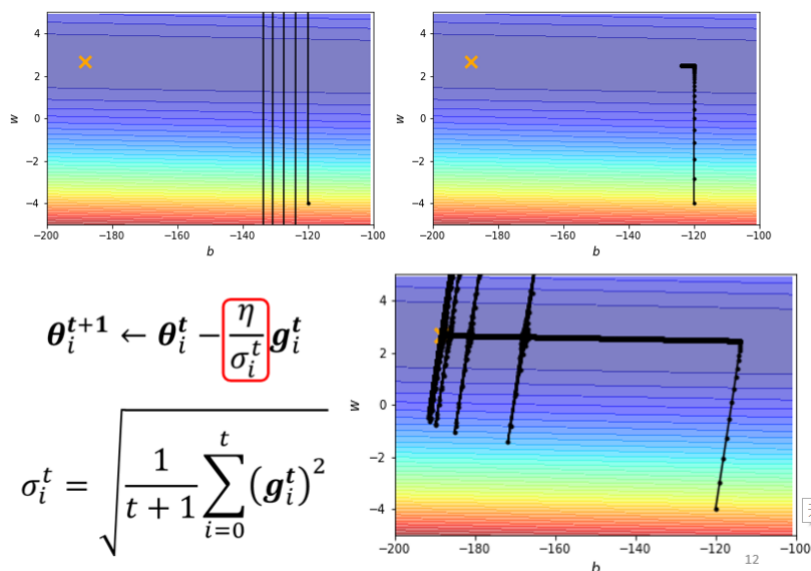
Learning Rate Scheduling

Without Adaptive Learning Rate



重新回到我们最开始的例子上来，现在我们来看一下加上Adaptive Learning Rate 以后，能不能把这个网络训练起来。这里采用Adagrad的做法，结果如下：

Without Adaptive Learning Rate



可以看到使用Adagrad以后，我们的训练可以继续走下去，走到非常接近终点的位置，这是因为**当我们走到之前gradient太小而导致参数update不动的地方时，在使用Adagrad的情况下learning rate会自动变大，所以我们的步伐就可以变大，就可以继续前进。**

但是接下来你又会发现，**为什么快走到终点的时候训练的轨迹在纵轴上突然爆炸了呢？**

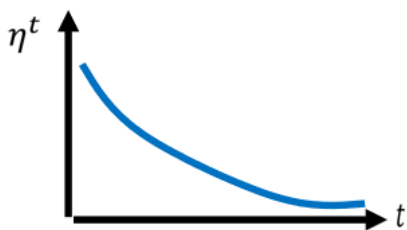
原因是这样的，我们在计算 σ 的时候，是把过去所有看到的gradient，都拿来作平均：

- 所以在纵轴的方向，起初的坡度很陡，gradient很大，导致算出的 σ 很大，learning rate就比较小。
- 但在转向向左走以后，在纵轴的方向gradient算出来都很小，而我们的 σ 是把过去所有看到的gradient都拿来作平均，所以纵轴这个方向就累积了很小的 σ ，累积到一个地步以后，learning rate就变得很大，导致step就变很大，因此在纵轴上就喷发出去了。
- 喷出去也没有关系，有办法修正回来，这是因为喷出去以后，就走到了在纵轴的方向gradient比较大的地方，于是这个 σ 又慢慢的变大， σ 慢慢变大以后，step又会慢慢变小，使得我们慢慢回到之前的道路上来。

如果说你不想看到这个训练的轨迹隔一段时间就喷发一次，你希望它能够稳定一些，**有一个方法也许可以解决这个问题，这个方法就是learning rate的scheduling。**

Learning Rate Scheduling

$$\theta_i^{t+1} \leftarrow \theta_i^t - \frac{\eta^t}{\sigma_i^t} g_i^t$$



Learning Rate Decay

As the training goes, we are closer to the destination, so we reduce the learning rate.

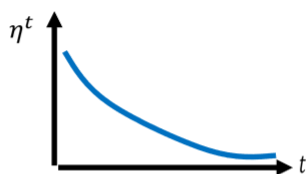
learning rate scheduling的意思就是说，我们**不要把 η 当作一个常数，我们让它成为跟时间有关的一个变量。**

最常见的策略叫做**Learning Rate Decay**，也就是说，**随着时间的不断地进行，随着参数不断的update，我们让这个 η 越来越小。**

这个做法的合理性如下：因为一开始我们距离终点很远，随着参数不断update，我们离终点越来越近，所以我们将learning rate减小，让我们参数的更新踩了一个刹车，让我们参数的更新能够慢下来，这样应该就能避免之前那个喷发的状况。

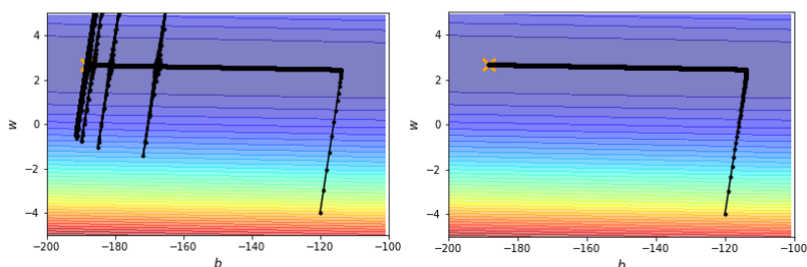
Learning Rate Scheduling

$$\theta_i^{t+1} \leftarrow \theta_i^t - \frac{\eta^t}{\sigma_i^t} g_i^t$$

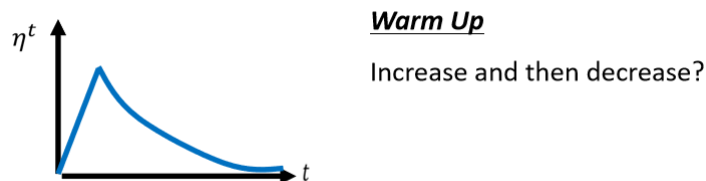


Learning Rate Decay

As the training goes, we are closer to the destination, so we reduce the learning rate.



可以看到加上Learning Rate Decay之后，我们就可以很平顺的走到终点。除了Learning Rate Decay以外，还有另外一个常用的Learning Rate Scheduling的方式，叫做**Warm Up**：



Warm Up是**让learning rate先变大后变小**，至于变大要变到多大，变大速度要多快，变小速度要多快，**这些都是这个方法的hyperparameter**，这个方法听起来很神奇，其实就是一个黑科技这样，这个黑科技出现在很多远古时代的论文里面，举例来说，Residual Network里面就是有Warm Up的：

We further explore $n = 18$ that leads to a 110-layer ResNet. In this case, we find that the initial learning rate of 0.1 is slightly too large to start converging⁵. So we use 0.01 to warm up the training until the training error is below 80% (about 400 iterations), and then go back to 0.1 and continue training. The rest of the learning schedule is as done previously. This 110-layer network converges well (Fig. 6, middle). It has *fewer* parameters than other deep and thin

⁵With an initial learning rate of 0.1, it starts converging (<90% error) after several epochs, but still reaches similar accuracy.

Residual Network

<https://arxiv.org/abs/1512.03385>

这个Residual Network里面说我们**先用learning rate 0.01来Warm Up**，再把learning rate改成**0.1**。

此外在有名的Transformer里面也用了一个式子提到了它：

5.3 Optimizer

We used the Adam optimizer [17] with $\beta_1 = 0.9$, $\beta_2 = 0.98$ and $\epsilon = 10^{-9}$. We varied the learning rate over the course of training, according to the formula:

$$lrate = d_{\text{model}}^{-0.5} \cdot \min(\text{step_num}^{-0.5}, \text{step_num} \cdot \text{warmup_steps}^{-1.5}) \quad (3)$$

This corresponds to increasing the learning rate linearly for the first warmup_steps training steps, and decreasing it thereafter proportionally to the inverse square root of the step number. We used $\text{warmup_steps} = 4000$.

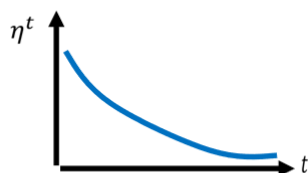
Transformer

<https://arxiv.org/abs/1706.03762>

那**为什么需要warm Up**呢，这个仍然是今天一个可以研究的问题。

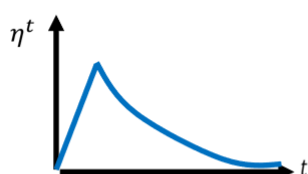
Learning Rate Scheduling

$$\theta_i^{t+1} \leftarrow \theta_i^t - \frac{\eta^t}{\sigma_i^t} g_i^t$$



Learning Rate Decay

After the training goes, we are close to the destination, so we reduce the learning rate.



Warm Up

Increase and then decrease?

At the beginning, the estimate of σ_i^t has large variance.

Please refer to **RAadam**

<https://arxiv.org/abs/1908.03265>

有一个可能的解释是说：当我们在用Adam, RMSProp, 或者Adagrad的时候，我们需要计算一个 σ ，它是一个统计的结果，而这个统计的结果要看到足够多笔数据以后，这个统计才精准，所以一开始我们的统计是不精准的。所以我们一开始不要让我们的参数变化的太快，先让它在初始点附近做一些小的探索，所以一开始learning rate比较小，是让它探索收集一些有关error surface的情报，等 σ 统计得比较精准以后，再让learning rate慢慢地变大。

所以这是一个可能的解释，想要进一步了解有关warm up的东西的话，可以看一下RAdam，这是对Adam的一个改进，论文里对warm up有更多的理解。