

神经网络训练不起来，怎么办？

General Guidance

Framework of ML

我们已经看了作业一了,其实之后好几个作业,它看起来的样子,基本上都是大同小异。就是你会有一堆训练的资料,这些训练集裡面,会包含了 x 跟 y 的 \hat{y}^1, x^1 和跟它对应的 \hat{y}^1, x^2 跟它对应的 \hat{y}^2 ,一直到 x^n 还有它对应的 \hat{y}^n 测试集,测试集就是你只有 x 没有 y ,其实在之后每一个作业,看起来都是非常类似的格式。

Training data: $\{(x^1, \hat{y}^1), (x^2, \hat{y}^2), \dots, (x^N, \hat{y}^N)\}$

Testing data: $\{x^{N+1}, x^{N+2}, \dots, x^{N+M}\}$

- 作业二,其实是做**语音辨识**,我们的 x 就是非常小的一段声音讯号,其实这个不是真正完整的语音辨识系统,它是语音辨识系统的一个阉割版, \hat{y} 是去预测判断,这一小段声音讯号,它对应到哪一个 phoneme。你不知道phoneme是什麼没有关係,你就把它想成是音标就可以了
- 作业三叫做**图像识别**,这个时候我们的 x 是一张图片, \hat{y} 是机器要判断说这张图片裡面,有什麼样的东西
- 作业四是**语者辨识**,语者辨识要做的事情是,这个 x 也是一段声音讯号, \hat{y} 现在不是phoneme, \hat{y} 是现在是哪一个人在说话,这样的系统,现在其实非常的有用,如果你打电话去银行的客服,现在都有自动的语者辨认系统,它会听说现在打电话进来的人,是不是客户本人,就少了客服人员问你身份验证的时间
- 作业五是做**机器翻译**, x 就是某一个语言,比如说,这是我唯一会的一句日文,痛みを知れ,它的 \hat{y} 就是另外一句话。



训练集就要拿来训练我们的Model,训练Model的过程上週已经讲过了,训练的过程就是三个步骤

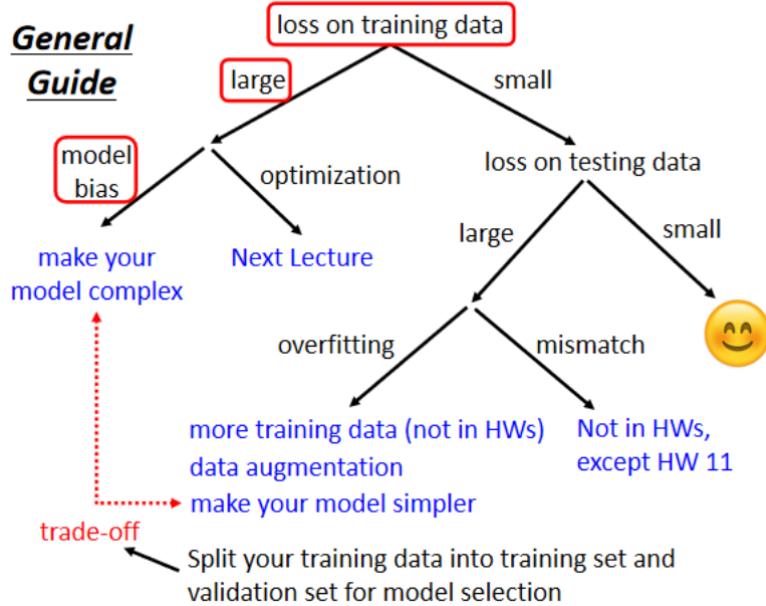
- 第一步,你要先写出一个有未知数的function,这个未知数,以后我们都用 θ 来代表,一个Model裡面所有的未知函数,所以 $f_\theta(x)$ 的意思就是说,我现在有一个function叫 $f(x)$,它裡面有一些未知的参数,这些未知的参数表示成 θ ,它的input叫做 x ,这个input叫做feature
- 第二步,你要定一个东西叫做loss,loss是一个function,这个loss的输入就是一组参数,去判断说这一组参数是好还是不好
- 第三步,你要解一个,Optimization的problem,你要去找一个 θ ,这个 θ 可以让loss的值越小越好,可以让loss的值最小的那个 θ ,我们叫做 θ^*

有了 θ^* 以后,那你就把它拿来用在测试集上,也就是你把 θ^* 带入这些未知的参数,本来 $f_\theta(x)$ 裡面有一些未知的参数,现在这个 θ 用 θ^* 来取代,它的输入就是你现在的测试集,输出的结果你就把它存起来,然后上传到Kaggle就结束了。

接下来你就会遇到一个问题,直接执行助教的sample code,往往只能够给你simple baseline的结果而已,如果你想要做得更好,那应该要怎麽办

General Guide

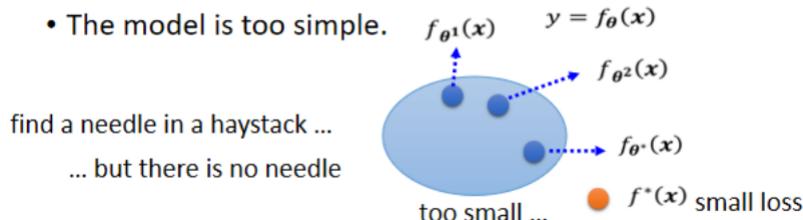
以下就是如何让你做得更好的攻略,它适用於前期所有的作业,这个攻略是怎麽走的呢?



从最上面开始走起,第一个是你今天如果你觉得,你在Kaggle上的结果不满意的话,第一件事情你要做的事情是,检查你的**training data**的**loss**。有人说"我在意的不是应该是,testing data的loss吗,因为Kaggle上面的结果,呈现的是testing data的结果"。但是你要先检查你的**training data**,看看你的**model**在**training data**上面,有没有学起来,再去看**testing**的结果,如果你发现,你的**training data**的**loss**很大,显然它在训练集上面也没有训练好,接下来你要分析一下,在训练集上面没有学好,是什麽样的原因,这边有两个可能,第一个可能是**model**的**bias**

Model bias -> More flexible Model

model的**bias**这件事情,我们在上週已经跟大家讲过了,所谓**model bias**的意思是说,假设你的**model**太过简单。



举例来说,我们现在写了一个有未知parameter的function,这个未知的parameter,我们可以代各种不同的数字,你代 θ^1 得到一个function $f_{\theta^1}(x)$,我们把那个function用这个,一个点来表示, θ^2 得到另一个function $f_{\theta^2}(x)$,你把所有的function集合起来,得到一个function的set。但是这个function的set太小了,这个function的set裡面,没有包含任何一个function,可以让我们的loss变低,即可以让loss变低的function,不在你的**model**可以描述的范围内。在这个情况下,就算你找出了一个 θ^* ,它是这些蓝色的function裡面,最好的那一个,也无济於事了,那个loss还是不够低。

这个状况就是你想要在大海裡面捞针,这个针指的是一个loss低的function,结果针根本就不在海裡,白忙一场,你怎麼捞都捞不出针,因为针根本就不在你的,这个function set裡面,不在你的这个大海裡面,所以怎麽办?

这个时候重新设计一个model,给你的model更大的弹性,举例来说,你可以增加你输入的features,我们上週说,本来我们输入的features,只有前一天的资讯,假设我们要预测接下来的这个,观看人数的话,我们用前一天的资讯,不够多,那用56天前的资讯,那model的弹性就比较大了。你也可以用Deep Learning,增加更多的弹性,所以如果你觉得,你的model的弹性不够大,那你可以增加更多features,可以设一个更大的model,可以用deep learning,来增加model的弹性,这是第一个可以的解法。

- Solution: redesign your model to make it more flexible

$$\begin{aligned}
 y = b + w x_1 &\xrightarrow{\text{More features}} y = b + \sum_{j=1}^{56} w_j x_j \\
 &\downarrow \text{Deep Learning (more neurons, layers)} \\
 y = b + \sum_i c_i \text{sigmoid}\left(b_i + \sum_j w_{ij} x_j\right)
 \end{aligned}$$

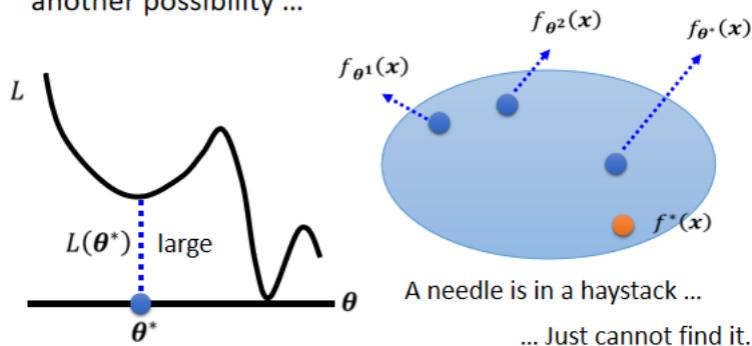
但是并不是training的时候,loss大就代表一定是model bias,你可能会遇到另外一个问题,这个问题是什麼,这个问题是**optimization做得不好**,什麼意思呢?

Optimization Issue

我们知道说,我们今天用的optimization,在这门课裡面,我们其实都只会用到gradient descent,这种optimization的方法,这种optimization的方法很多的问题。

举例来说 我们上週也讲过说,你可能会卡在local minima的地方,你没有办法找到一个,真的可以让loss很低的参数,如果用图具象化的方式来表示,就像这个样子

- Large loss not always imply model bias. There is another possibility ...



蓝色部分是你的model可以表示的函式所形成的集合,你可以把 θ 代入不同的数值,形成不同的function,把所有的function通通集合在一起,得到这个蓝色的set,这个蓝色的set裡面,确实包含了一些function,这些function它的loss是低的。

但问题是**gradient descent**这一个演算法,没办法帮我们找出,这个loss低的function,gradient descent是解一个optimization的problem,找到 θ^* 然后就结束了。但是这个 θ^* 它给我的loss不够低,这一个model裡面,存在著某一个function,它的loss是够低的,gradient descent,没有给我们这一个function

这就好像是说 我们想**大海捞针**,针确实在海裡,但是我们却没有办法把针捞起来,这边问题就来了

training data的loss不够低的时候,到底是model bias,还是optimization的问题呢==

- 找不到一个loss低的function,到底是因為我们的model的弹性不够,我们的海裡面没有针
- 还是说,我们的model的弹性已经够了,只是optimization gradient descent不给力,它没办法把针捞出来

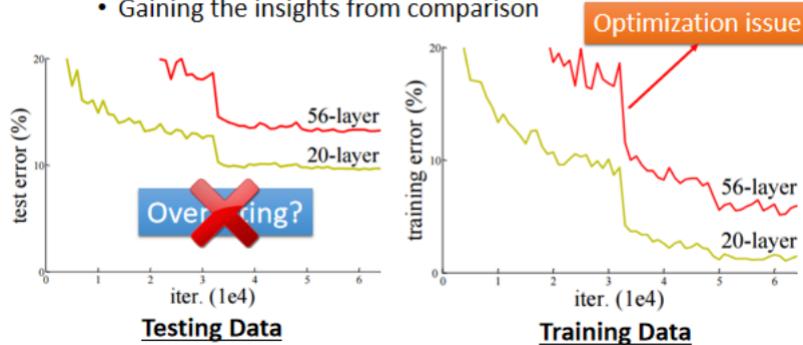
到底是哪一个呢,到底我们的model已经够大了,还是它不够大,怎麼判断这件事呢

Model Bias or Optimization Issue?

一个建议判断的方法,就是你可以透过比较不同的模型,来得知说,你的model现在到底够不够大,怎麽说呢

- Diagnosis: large loss on training data, and you believe your model has sufficient flexibility (?)

- Gaining the insights from comparison



我们这边举一个例子,这一个实验是从residual network,那篇paper裡面节录出来的,我们把paper链接: <http://arxiv.org/abs/1512.03385>

这篇paper一开头就跟你讲一个故事,它说 我想测2个networks。一个network有20层,一个network有56层。我们把它们测试在测试集上,这个横轴指的是training的过程,就是你参数update的过程,随著参数的update,当然你的loss会越来越低,但是结果20层的loss比较低,56层的loss还比较高。这个residual network是比较早期的paper,2015年的paper,如果你现在大学生的话,那个时候你都还是高中生而已,所以那个时候大家对Deep Learning,了解还没有那麼的透彻,大家对deep learning,有各种奇怪的误解,很多人看到这张图都会说,这个代表overfitting,告诉你deep learning不work,56层太深了 不work,根本就不需要那麼深。那个时候大家也不是每个人都觉得deep learning是好的,那时候还有很多,对deep learning的质疑,所以看到这个实验有人就会说,最深没有比较好,所以这个叫做overfitting,但是这个是overfitting吗。这个不是overfitting,等一下会告诉你overfitting是什麼,并不是所有的结果不好,都叫做overfitting

你要检查一下训练集上面的解释,你检查训练集的结果发现说,现在20层的network,跟56层的network比起来,在训练集上,20层的network loss其实是比较低的,56层的network loss是比较高的。这代表56层的network,它的optimization没有做好,它的optimization不给力。你可能问说,你怎麼知道是56层的optimization不给力,搞不好是model bias,搞不好是56层的network,它的model弹性还不够大,它要156层才好,56层也许弹性还不够大,但是你比较56层跟20层,20层的loss都已经可以做到这样了,56层的弹性一定比20层更大对不对。如果今天56层的network要做到20层的network可以做到的事情,对它来说是轻而易举的。它只要前20层的参数,跟这个20层的network一样,剩下36层就什麼事都不做,identity copy前一层的输出就好了,56层的network一定可以做到20层的network可以做到的事情,所以20层的network已经都可以走到這麼底的loss了,56层的network,它比20层的network弹性还要更大,所以没有道理。所以56层的network,如果你optimization成功的话,它应该要比20层的network,可以得到更低的loss,但结果在训练集上面没有,这个不是overfitting,这个也不是model bias,因為56层network弹性是够的,这个问题是你的optimization不给力,optimization做得不够好

所以刚才那个例子就告诉我们,你怎麼知道你的optimization有没有做好,这边给大家的建议是。看到一个你从来没有做过的问题,也许你可以先跑一些比较小的,比较浅的network,或甚至用一些,不是deep learning的方法,比如说 linear model,比如说support vector machine。support vector machine不知道是什麼也没有关係,它们可能是比较容易做Optimize的,它们比较不会有optimization失败的问题。也就是这些model它会竭尽全力的,在它们的能力范围之内,找出一组最好的参数,它们比较不会有失败的问题,所以你可以先train一些,比较浅的model,或者是一些比较简单的model,先知道 先有个概念说,这些简单的model,到底可以得到什麼样的loss。

接下来还缺一个深的model,如果你发现你深的model,跟浅的model比起来,深的model弹性比较大,但loss却没有办法比浅的model压得更低,那就代表说你的optimization有问题,你的gradient descent不给力,那你要有一些其它的方法,来把optimization这件事情做得更好

- Diagnosis: large loss on training data, and you believe your model has sufficient flexibility (?)
- Gaining the insights from comparison
- Start from shallower networks (or other models), which are easier to train.
- If deeper networks do not obtain smaller loss on **training data**, then there is optimization issue.

	1 layer	2 layer	3 layer	4 layer	5 layer
2017 – 2020	0.28k	0.18k	0.14k	0.10k	0.34k

- Solution: More powerful optimization technology (next lecture)

举例来说,我们上次看到的这个,观看人数预测的例子,我们说在训练集上面,2017年到2020年的资料是训练集,一层的network,它的loss是0.28k,2层就降到0.18k,3层就降到0.14k,4层就降到0.10k。但是我测5层的时候结果变成0.34k,这是什麼问题?我们现在loss很大,这显然不是model bias的问题,因为4层都可以做到0.10k了,5层应该可以做得更低,这个是optimization的problem,这个是optimization的时候做得不好,才造成这样子的问题。

那如果optimization做得不好的话,怎麽办呢,这个我们下一节课,就会告诉大家要怎麽办,你现在就知道怎麽判断,现在如果你的training的loss大,到底是model bias还是optimization,如果model bias 那就把model变大,如果是optimization失败了,那就看等一下的课程怎麽解这个问题。

假设你现在经过一番的努力,你已经可以让你的,training data的loss变小了,那接下来你就可以来看,testing data loss,如果testing data loss也小,有比这个strong baseline还要小就结束了,没什麼好做的就结束了。

那但是如果你觉得还不够小呢,如果**training data上面的loss小,testing data上的loss大**,那你可能就是真的遇到**overfitting**的问题

你拿一个结果来问我说,老师这个结果要怎麽做得更好的时候,我第一个问题就会问你说,你在**training data**上的loss,到底做得怎麽样,我发现十个同学有八个同学都说,要看**training data**的loss吗,我没有把**training data** loss记下来,你要把**training data** loss记下来,先确定说你的optimization没有问题,你的model够大了,然后接下来,才看看是不是**testing**的问题,如果是**training**的loss小,testing的loss大,这个有可能是overfitting

Overfitting

為什麼会有overfitting这样的状况呢,為什麼有可能**training**的loss小,testing的loss大呢,这边就举一个极端的例子来告诉你说,為什麼会发生这样子的状况

- Small loss on training data, large loss on testing data. Why?

An extreme example

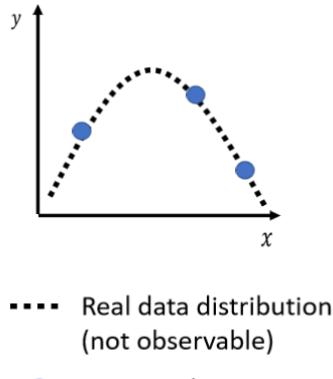
Training data: $\{(x^1, \hat{y}^1), (x^2, \hat{y}^2), \dots, (x^N, \hat{y}^N)\}$

$$f(x) = \begin{cases} \hat{y}^i & \exists x^i = x \\ \text{random} & \text{otherwise} \end{cases} \quad \text{Learns nothing ... !}$$

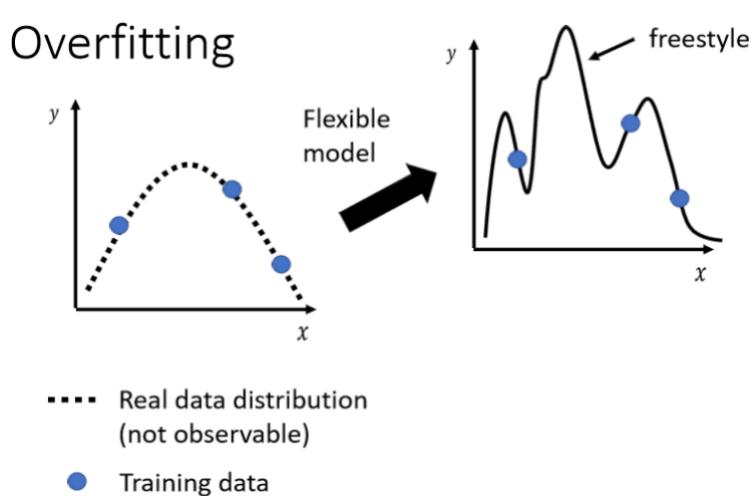
This function obtains **zero training loss**, but **large testing loss**.

这是我们的训练集,假设根据这些训练集,某一个很废的,machine learning的方法,它找出了一个一无是处的function。这个一无是处的function说,如果今天 x 当做输入的时候,我们就去比对这个 x ,有没有出现在训练集裡面,如果 x 有出现在训练集裡面,就把它对应的 y 当做输出,如果 x 没有出现在训练集裡面,就输出一个随机的值。那你可以想像这个function啥事也没有干,它是一个一无是处的function,但虽然它是一个一无是处的function,它在training的数据上,它的loss可是0呢。你把training的数据,通通丢进这个function裡面,它的输出跟你的训练集的level,是一模一样的,所以在training data上面,这个一无是处的function,它的loss可是0呢,可是在testing data上面,它的loss会变得很大,因为它其实什麼都没有预测,这是一个比较极端的例子,在一般的状况下,也有可能发生类似的事情。

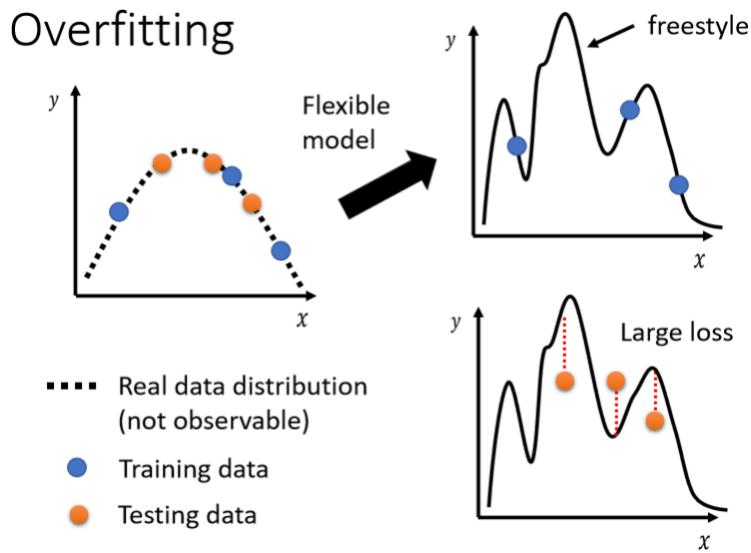
举例来说,假设我们输入的feature叫做 x ,我们输出的level叫做 y ,那 x 跟 y 都是一维的



x 跟 y 之间的关係,是这个二次的曲线,这个曲线我们刻意用虚线来表示,因为我们通常没有办法,直接观察到这条曲线,我们真正可以观察到的是什麼,我们真正可以观察到的,是我们的训练集,训练集 你可以想像成,就是从这条曲线上面,随机sample出来的几个点



今天的模型 它的能力非常的强,它的flexibility很大,它的弹性很大的话,你只给它这三个点,它会知道说,在这三个点上面我们要让loss低,所以今天你的model,它的这个曲线会通过这三个点,但是其它没有训练集做为限制的地方,它就会有freestyle,因为它的flexibility很大,它弹性很大,所以你的model,可以变成各式各样的function,你没有给它资料做为训练,它就会有freestyle, 可以产生各式各样奇怪的结果。



这个时候,如果你再丢进你的testing data,你的testing data 和training的data,当然不会一模一样,它们可能是从同一个,distribution sample出来的,testing data是橙色的这些点,训练data是蓝色的这些点,用蓝色的这些点,找出一个function以后,你测试在橘色的这些点上,不一定会好,如果你的model它的自由度很大的话,它可以產生非常奇怪的曲线,导致训练集上的结果好,但是测试集上的loss很大,那至於更详细的背后的数学原理,為什麼这个比较有弹性的model,它就比较会,overfitting背后的数学原理,我们留待下下週

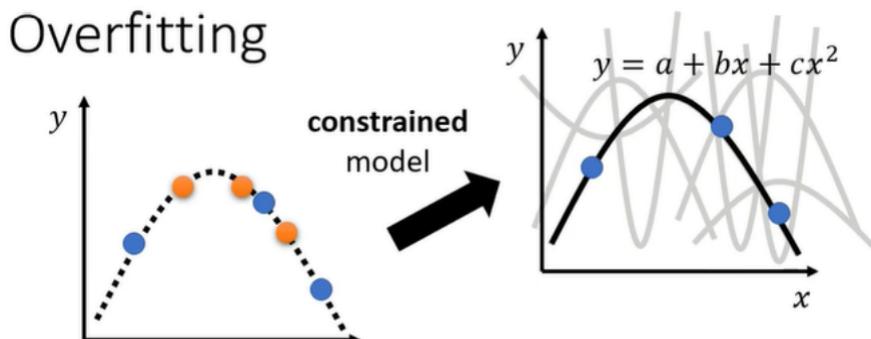
那怎麼解决刚才那个,overfitting的问题呢,有两个可能的方向

1. 第一个方向是,也许这个方向往往是最有效的方向,是增加你的训练集

今天假设你自己,想要做一个application,你发现有overfitting的问题,其实我觉得,最简单解决overfitting的方法,就是增加你的训练集。所以今天如果训练集,蓝色的点变多了,那虽然你的model它的弹性可能很大,但是因为这边的点非常非常的多,它就可以限制住,它看起来的形状还是会很像,产生这些资料背后的二次曲线,但是你在作业裡面,你是不能够使用这一招的,因为我们并不希望大家浪费时间,来收集资料。那你可以做什麼呢,你可以做data augmentation, 这个方法并不算是使用了额外的资料。Data augmentation就是,你用一些你对於这个问题的理解,自己创造出新的资料。

举例来说在做影像辨识的时候,非常常做的一个招式是,假设你的训练集裡面有某一张图片,把它左右翻转,或者是把它其中一块截出来放大等等,你做左右翻转 你的资料就变成两倍,那这个就是data augmentation。但是你要注意一下data augmentation,不能够随便乱做,这个augment要augment得有道理,举例来说在影像辨识裡面,你就很少看到有人把影像上下颠倒当作augmentation,因为这些图片都是合理的图片,你把一张照片左右翻转,并不会影响到裡面是什麽样的东西,但你把它颠倒 那就很奇怪了,这可能不是一个训练集裡面,可能不是真实世界会出现的影像。那如果你给机器看这种,奇怪的影像的话,它可能就会学到奇怪的东西,所以data augmentation,要根据你对资料的特性,对你现在要处理的问题的理解,来选择合适的,data augmentation的方式,好 那这边是增加资料的部分

2. 另外一个解法就是不要让你的模型,有那麼大的弹性,给它一些限制,



举例来说 假设我们直接限制说,现在我们的model,我们somehow猜测出 知道说,x跟y背后的关係,其实就是一条二次曲线,只是我们不明确的知道这二次曲线,裡面的每一个参数长什麼样。那你说你怎麽会猜测出这样子的结果,你怎麽会知道说,要用多constraint的model才会好呢,那这就取决於你对这个问题的理解,因为这种model是你自己设计的,到底model要多constraint多flexible,结果才会好,那这个要问你自己,那要看这个设计出不同的模型,你就会得出不同的结果

那现在假设我们已经知道说,模型就是二次曲线,那你就会给你,你就会在选择function的时候,有很大的限制,因为二次曲线要嘛就是这样子,来来去去就是那几个形状而已,所以当我们的训练集有限的时候,因为我们来来去去,只能够选那几个function。所以你可能,虽然说只给了三个点,但是因为我们可以选择的function有限,你可能就会正好选到,跟真正的distribution,比较接近的function,然后在测试集上得到比较好的结果。

有哪些方法可以给model製造限制呢,举例来说,

1. 给它**比较少的参数**,如果是deep learning的话,就给它比较少的神经元的数目,本来每层一千个神经元,改成一百个神经元之类的,或者是你可以让model共用参数,你可以让一些参数有一样的数值,那这个部分如果你没有很清楚的话,也没有关係,我们之后在讲CNN的时候,会讲到这个部分,所以这边先前情先预告一下,就是我们之前讲的network的架构,叫做fully-connected network,那fully-connected network,其实是一个比较有弹性的架构,而CNN是一个比较有限制的架构,就说你可能会说,CNN不是比较厉害吗,大家都说做影像就是要CNN,比较厉害的model,难道它比较没有弹性吗,没错,它是一种比较没有弹性的model,它厉害的地方就是,它是针对影像的特性,来限制模型的弹性,所以你今天fully-connected的network,可以找出来的function所形成的集合,其实是比较大的,CNN这个model所找出来的function,它形成的集合其实是比较小的,其实包含在fully-connected的, network裡面的,但是就是因为CNN给了,比较大的限制,所以CNN在影像上,反而会做得比较好,那这个之后都还会再提到,
2. 另外一个就是用**比较少的features**,那刚才助教已经示范过,本来给三天的资料,改成用给两天的资料,其实结果就好了一些,那这是一个招数
3. 还有一个招数叫做**Early stopping**, Early stopping, Regularization跟Dropout,都是之后课程还会讲到的东西,那这三件事情在作业一的程式裡面,这个Early stopping其实是有的,助教有写在它的code裡面,所以不知道这是什麼也没有关係,反正你直接执行sample code,裡面就有了,Regularization,助教留下了一个空格给大家填,那你不知道什麼是regularization,没有关係,反正你可以过得了middle的baseline,那如果你想做得更好,也许你可以先自己survey一下,regularization是什麼,看有没有办法自己写
4. **Dropout**,这是另外一个在Deep Learning裡面,常用来限制模型的方法,那这个之后还会再提到

但是我们也不要给太多的限制,為什麼不能给模型太多的限制呢

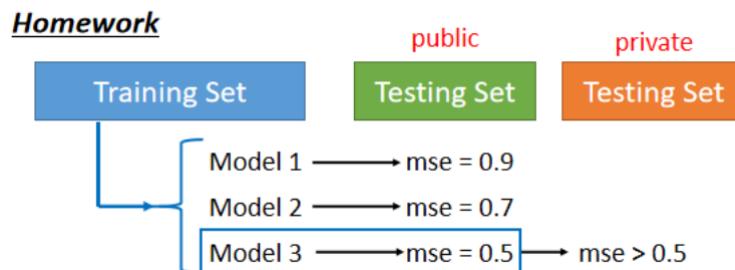
假设我们现在给模型更大的限制说,我们假设我们的模型,一定是Linear的Model,一定是写成 $y=a+bx$,那你的model它能够产生的function,就一定是一条直线。今天给三个点,没有任何一条直线,可以同时通过这三个点,但是你只能找到一条直线,这条直线跟这些点比起来,它们的距离是比较近的,但是你没有办法找到任何一条直线,同时通过这三个点,这个时候你的模型的限制就太大了,你在测试集上就不会得到好的结果。但是这个不是overfitting,因为又回到了model bias的问题,所以你现在这样在这个情况下,这个投影片上面你结果不好,并不是因为overfitting了,而是因为给你模型太大的限制,大到你有了model bias的问题。

所以你就会发现说,这边產生了一个矛盾的状况,今天你让你的模型的复杂的程度,或这样让你的模型的弹性越来越大,但是什麼叫做复杂的程度,什麼叫做弹性,在今天这堂课裡面,我们其实都没有给明确的定义,只给你一个概念上的叙述,那在下下週的课程裡面,你会真的认识到,什麼叫做一个模型很复杂,什麼叫做一个模型有弹性,怎麼真的衡量一个模型的弹性,复杂的程度有多大,那今天我们先用直观的来了解

所谓比较复杂就是,它可以包含的function比较多,它的参数比较多,这个就是一个比较复杂的model。那一个比较复杂的model,如果你看它的training的loss,你会发现说 随著model越来越复杂,Training的loss可以越来越低,但是testing的时候呢,当model越来越复杂的时候,刚开始,你的testing的loss会跟著下降,但是当复杂的程度,超过某一个程度以后,Testing的loss就会突然暴增了

那这就是因為说,当你的model越来越复杂的时候,复杂到某一个程度,overfitting的状况就会出现,所以你在training的loss上面,可以得到比较好的结果,那在Testing的loss上面,你会得到比较大的loss,那我们当然期待说,我们可以选一个中庸的模型,不是太复杂的 也不是太简单的,刚刚好可以在训练集上,给我们最好的结果,给我们最低的loss,给我们最低的testing loss,怎麼选出这样的model呢

一个很直觉的 你很有可能,没有人告诉你要怎麼做的话,你可能很直觉就会這麼做的,做法就是说,这个kaggle不是立刻上传,就可以知道答案了吗。所以假设我们有三个模型,它们的复杂的程度不太一样,我不知道要选哪一个模型才会刚刚好,在测试集上得到最好的结果,因為你选太复杂的就overfitting,选太简单的有model bias的问题,那怎麼选一个不偏不倚的,不知道 那怎麼办

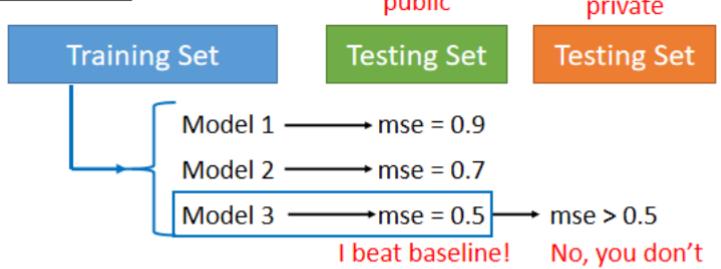


把这三个模型的结果都跑出来,然后上传到kaggle上面,你及时的知道了你的分数,看看哪个分数最低,那个模型显然就是最好的模型。但是并不建议你這麼做,為什麼不建议你這麼做呢。我们再举一个极端的例子,我们再把刚才那个极端的例子拿出来,假设现在有一群model,这一群model不知道為什麼都非常废,它们每一个model產生出来的,都是一无是处的function,我们有一到一兆个model,这一到一兆个model不知道為什麼,learn出来的function,都是一无是处的function

它们会做的事情就是,训练集裡面有的资料就把它记下来,训练集没看过的,就直接output随机的结果。那你现在有一兆个模型,那你再把这一兆个模型的结果,通通上传到kaggle上面,你就得到一兆个分数,然后看这一兆的分数裡面,哪一个结果最好,你就觉得那个模型是最好的。那虽然说每一个模型,它们在这个Testing data上面,这个testing data它都没有看过啊,所以它输出的结果都是随机的,但虽然在testing data上面,输出的结果都是随机的,但是你不断的随机,你总是会找到一个好的结果,所以也许编号五六七八九的那个模型,它找出来的function,正好在testing data上面,就给你一个好的结果,那你就会很高兴觉得说,这个model编号五六七八九,是个好model,这个好model得到一个好function。虽然它其实是随机的 但你不知道,但这个好function,在这个testing data上面,给我们好的结果,所以你就觉得说 这个结果不错,就这样 我就选这一个model,这个function,当作我们最后上传的结果,当作我最后要用在,private testing set上的结果

但是如果你这样做,往往就会得到非常糟的结果,因為这个model毕竟是随机的,它恰好在public的testing set data上面得到一个好结果,但是它在private的testing set上,可能仍然是随机的。我们这个testing set,分成public的set跟private的set,你在看分数的时候 你只看得到public的分数, private的分数要deadline以后才知道,但假设你在挑模型的时候,你完全看你在public set上面的,也就leaderboard上的分数,来选择你的模型的话,你可能就会这个样子: 你在public的leaderboard上面排前十,但是deadline一结束,你就心态就崩了这样,你就掉到三百名之外,而且我们这修课的人這麼多,你搞不好会掉到一千名之外,也说不定。

Homework



The extreme example again

$$f_k(x) = \begin{cases} \hat{y}^i & \exists x^i = x \\ \text{random} & \text{otherwise} \end{cases} \quad k: 1 - 10000000000000000000$$

It is possible that $f_{56789}(x)$ happens to get good performance on public testing set.

So you select $f_{56789}(x)$ Random on private testing set

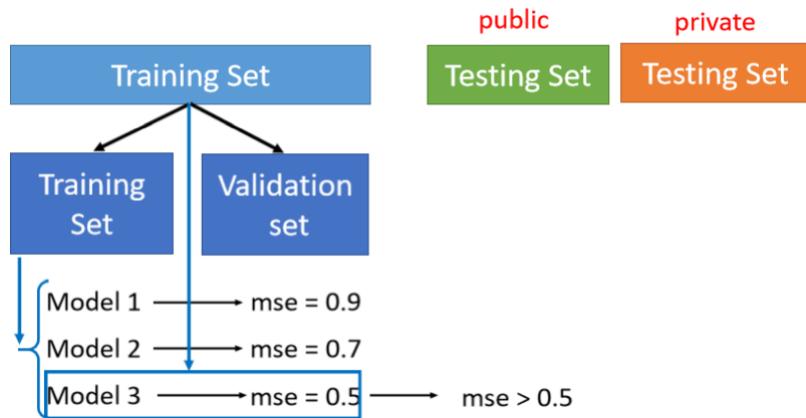
而且这件事情并不是传说,并没有夸饰,每年都会有这样子的状况发生,那因為今年我们会看public,就是说我们在算分数的时候,你在public上面的结果好,还是会给你一点分数,我们不是只看private的分数而已,是public跟private的分数都看,那过去有些学期,是只看private的分数的时候,发生这种状况,你心态就会整个崩掉这样子,你就会非常非常的郁闷

那為什麼我们要把testing的set,分成public跟private呢?為什麼我们不能,就通通都分public就好呢,為什麼要為难大家呢,為什麼要让大家疑神疑鬼,不知道自己private上的结果是什麼。你自己想想看,假设所有的data都是public,那我刚才说,就算是一个一无是处的Model,得到了一无是处的function,它也有可能在public的数据上面,得到好的结果,如果我们今天只有public的testing set,没有private的testing set,那你就回去写一个程式,不断random產生输出就好,然后不断把random的输出,上传到kaggle,然后看你什麼时候,可以random出一个好的结果,那这个作业就结束了。这个显然没有意义,显然不是我们要的,而且因為如果今天 你想看,然后这边有另外一个有趣的事情就是,你知道因為今天如果,public的testing data是公开的,你可以知道public的,testing data的结果,那你就算是一个很废的模型,產生了很废的function,也可能得到非常好的结果

所以讲了這麼多,只是想要告诉大家说,我們為什麼要切public的testing set,我為什麼要切private的testing set,然后你其实不要花,不要用你public的testing set,去调你的模型,因為你可能会在, private的testing set上面,得到很差的结果,那不过因為今年,你在public set上面的,好的结果也有算分数,所以怎麼办呢,為了避免你 就你可能会说,好 那我放弃private set的结果,就只拿public set的结果,然后不断地產生随机的结果,去上传到Kaggle来,然后看看说能不能够,正好随机出一个好的结果,為了避免你浪费时间做这件事情,所以有每日上传的限制,让你不会说,我拿很废的模型只產生随机的结果,不断的测试public的testing 的score

How to Choose Model : Cross Validation

那到底要怎麼做才选择model,才是比较合理的呢,那界定的方法是这个样子的,那助教程式裡面也都帮大家做好了,你要把Training的资料分成两半,一部分叫作Training Set,一部分是Validation Set



刚才助教程式裡面已经看到说,有90%的资料放在Training Set裡面,有10%的资料,会被拿来做Validation Set,你在Training Set上训练出来的模型,你在Validation Set上面,去衡量它们的分数,你根据Validation Set上面的分数,去挑选结果,再把这个结果上传到Kaggle上面,去看看你得到的public的分数,那因為你在挑分数的时候,是用Validation Set来挑你的model,所以你的public的Testing Set的分数,就可以反应你的,private Testing Set的分数,就比较不会得到说,在public上面结果很好,但是在private上面结果很差,这样子的状况

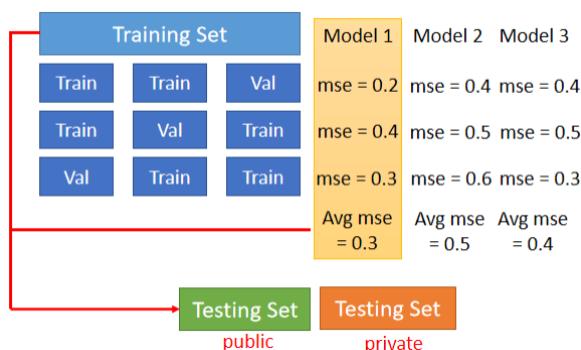
当我知道说,其实你看到public的结果以后,你就会去想要调它,你看到你现在弄了一堆模型,然后用Validation Set检查一下,找了一个模型放到public set上以后,发现结果不好,你其实不太可能不根据这一个结果,去调整你的模型,但是假设这一个route做太多次,你根据你的,public Testing Set上的结果,去调整你的model太多次,你就又有可能fit在,你的public Testing Set上面,然后在private Testing Set上面,得到差的结果,不过还好反正我们有限制上传的次数,所以这个route,你也没有办法走太多次,可以避免你太过fit在,public的Testing Set上面的结果。

当我们拿到数据之后,一般来说,我们把数据分成这样的三份:训练集(60%),验证集(20%),测试集(20%)。用训练集训练出模型,然后用验证集验证模型,根据情况不断调整模型,选出其中最好的模型,记录最好的模型的各项选择,然后据此再用(训练集+验证集)数据训练出一个新模型,作为最终的模型,最后用测试集评估最终的模型。

线上直播的同学,我复述一下刚才那个同学的问题,他的问题是说,所以我们不能去看,public Testing Set的结果吗,理想上是,理想上你就用Validation Set挑就好,然后上传以后 怎样就是怎样,有过那个strong baseline以后,就不要再去动它了,那这样子就可以避免,你overfit在Testing Set上面,好 那但是这边会有一个问题,就是怎麼分Training Set,跟Validation Set呢,那如果在助教程式裡面,就是随机分的,但是你可能会说,搞不好我这个分 分得不好啊,搞不好我分到很奇怪的Validation Set,会导致我的结果很差,

N-fold Cross Validation

如果你有这个担心的话,那你可以用N-fold Cross Validation,



N-fold Cross Validation就是你先把你的训练集切成N等份,在这个例子裡面我们切成三等份,切完以后,你拿其中一份当作Validation Set,另外两份当Training Set,然后这件事情你要重复三次。也就是说,你先第一份第二份当Train,第三份当Validation,然后第一份第三份当Train,第二份当Validation,第一份当Validation,第二份第三份当Train。然后接下来 你有三个模型,你不知道哪一个是好的,你就把这三个模型,在这三个setting下,在这三个Training跟Validation的,data set上面,通通跑过一次,然后把这三个模型,在这三种状况的结果都平均起来,把每一个模型在这三种状况的结果,都平均起来,再看看谁的结果最好

那假设现在model 1的结果最好,你用这三个fold得出来的结果是,这个model 1最好,然后你再把model 1,用在全部的Training Set上,然后训练出来的模型,再用在Testing Set上面,好 那这个是N-fold Cross Validation,好 那这个就是这门课前期的攻略,它可以带你打赢前期所有的副本

Predicted playback volume

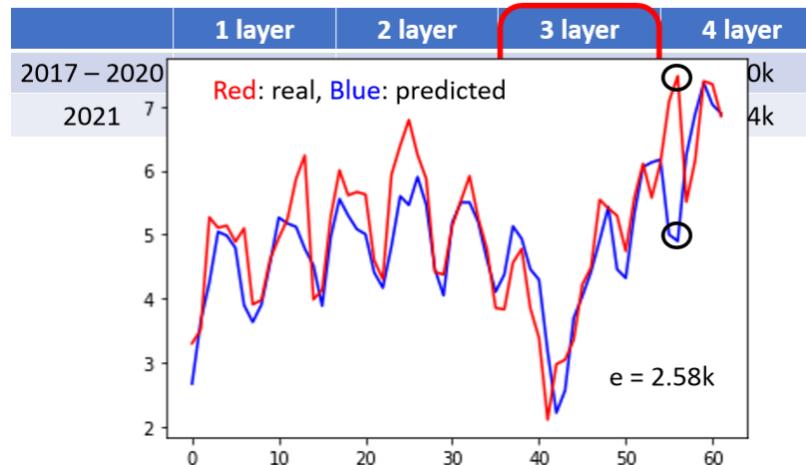
那接下来也许你要问的一个问题是,上週结束的时候,不是讲到预测2/26,也就是上週五的观看人数吗,到底结果做得怎麽样

那这个就是我们要做的结果,上週比较多人选了三层的network,所以我们就把三层的network,拿来测试一下,以下是测试的结果,我们就没有再调参数了,大家决定用三层的就是下好离手了,就直接用上去了

Let's predict no. of views of 2/26!

	1 layer	2 layer	3 layer	4 layer
2017 – 2020	0.28k	0.18k	0.14k	0.10k
2021	0.43k	0.39k	0.38k	0.44k

Let's predict no. of views of 2/26!

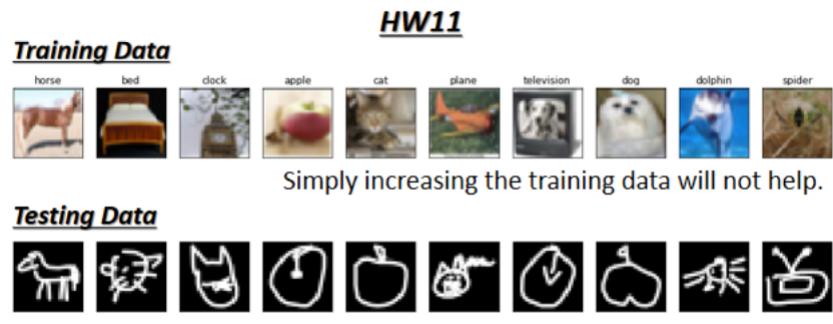


得到的结果是这个样子了,这个图上 这个横轴就是从,2021年的1月1号开始 一直往下,然后红色的线是真实的数字,蓝色的线是预测的结果,2/26在这边 这个是今年2021年,观看人数最高的一天了,那机器的预测怎样呢,哇 非常的惨 差距非常的大,差距有2.58k這麼多,感谢大家 為了让这个模型不準,上週五花了很多力气,去点了这个video,所以这一天是,今年观看人数最多的一天,那你可能开始想说,那别的模型怎麽样呢,其实我也跑了一层二层跟四层的看看,所有的模型 都会惨掉,两层跟三层的错误率都是2点多k,其实四层跟一层比较好,都是1.8k左右,但是这四个模型不约而同的,觉得2/26应该是个低点,但实际上2/26是一个公值,那模型其实会觉得它是一个低点,也不能怪它,因為根据过去的资料,礼拜五就是没有人要学机器学习,礼拜五晚上大家都出去玩了对不对,礼拜五的观看人数是最少了,但是2/26出现了反常的状况,好 那这个就不能怪模型了,那我觉得出现这种状况,应该算是另外一种错误的形式,这种错误的形式,我们这边叫作mismatch。

那也有人会说,mismatch也算是一种Overfitting,这样也可以,这都只是名词定义的问题,那我这边想要表达的事情是,mismatch它的原因跟overfitting,其实不一样,一般的overfitting,你可以用搜集更多的资料来克服,但是mismatch意思是说,你今天的训练集跟测试集,它们的分佈是不一样的

在训练集跟测试集,分佈是不一样的时候,你训练集再增加,其实也没有帮助了,那其实在多数的作业裡面,我们不会遇到这种mismatch的问题,我们都有把题目设计好了,所以资料跟测试集它的分佈差不多

举例来说 以刚才作业一的,Covid19為例的话,假设我们今天资料在,分训练集跟测试集的时候,我们说2020年的资料是训练集,2021年的资料是测试集,那mismatch的问题可能就很严重了,这个我们其实有试过了试了一下,如果今天用2020年当训练集,2021年当测试集,你就怎麼做都是惨了 就做不起来,训练什麼模型都会惨掉。因為2020年的资料跟2021年的资料,它们背后的分佈其实都是不一样,所以你拿2020年的资料来训练,在2021年的作业一的资料上,你根本就预测不準,所以来助教是用了别的方式,来分割训练集跟测试集,好 所以我们多数的作业,都不会有这种mismatch的问题,那除了作业十一。



因為作业十一就是,针对mismatch的问题来设计的,作业十一也是一个影像分类的问题,这是它的训练集,看起来蛮正常的,但它测试集就是长这样子了,所以你知道这个时候,这个时候增加资料哪有什麼用呢,增加资料,你也没有办法让你的模型做得更好,所以这种问题要怎麼解决,那犹待作业十一的时候再讲,好 那你可能会问说 我怎麼知道,现在到底是不是mismatch呢,那我觉得知不知道是mismatch,那就要看你对这个资料本身的理解了,你可能要对你的训练集跟测试集,的產生方式有一些理解,你才能判断说,它是不是遇到了mismatch的状况,好 那这个就是我们作业的攻略,

Optimization

When gradient is small

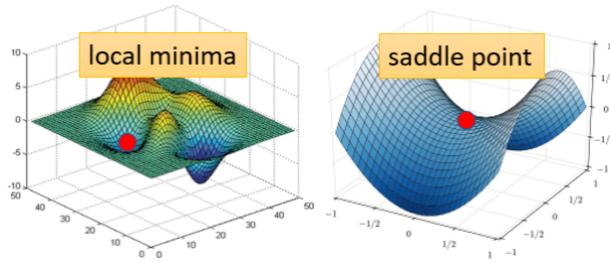
Critical Point or Saddle Point?

现在我们要讲的是Optimization的部分,所以我们要讲的东西基本上跟Overfitting没有什麼太大的关联,我们只讨论Optimization的时候,怎麼把gradient descent做得更好,那為什麼Optimization会失败呢?

你常常在做Optimization的时候,你会发现,随著你的参数不断的update,你的training的loss不会再下降,但是你对这个loss仍然不满意,就像我刚才说的,你可以把deep的network,跟linear的model,或比较shallow network 比较,发现说它没有做得更好,所以你觉得deep network,没有发挥它完整的力量,所以Optimization显然是有问题的。但有时候你会甚至发现,一开始你的model就train不起来,一开始你不管怎麼update你的参数,你的loss通通都掉不下去,那这个时候到底发生了什麼事情呢?

过去常见的一个猜想,是因為我们现在走到了一个地方,这个地方参数对loss的微分為零, **当你的参数对loss微分為零的时候,gradient descent就没有办法再update参数了**,这个时候training就停下来了,loss当然就不会再下降了。讲到gradient為零的时候,大家通常脑海中最先浮现的,可能就是local minima,所以常有人说做deep learning,用gradient descent会卡在local minima,然后所以gradient descent不work,所以deep learning不work。但是如果有一天你要写,跟deep learning相关paper的时候,你千万不要讲卡在local minima这种事情,别人会觉得你非常没有水準,為什麼

因為不是只有local minima的gradient是零,还有其他可能会让gradient是零,比如说 saddle point,所谓的saddle point,其实就是gradient是零,但是不是local minima,也不是local maxima的地方,像在右边这个例子裡面 红色的这个点,它在左右这个方向是比较高的,前后这个方向是比较低的,它就像是一个马鞍的形状,所以叫做saddle point,那中文就翻成鞍点



像saddle point这种地方,它也是gradient為零,但它不是local minima,那像这种gradient為零的点,统称为**critical point**,所以你可以说你的loss,没有办法再下降,也许是因為卡在了critical point,但你不能说是卡在local minima,因為saddle point也是微分為零的点

但是今天如果你发现你的gradient,真的很靠近零,卡在了某个critical point,我们有没有办法知道,到底是local minima,还是saddle point? 其实是有办法的

為什麼我们想要知道到底是卡在local minima,还是卡在saddle point呢

- 因为如果是卡在local minima,那可能就没有路可以走了,因为四周都比较高,你现在所在的位置已经是最低的点,loss最低的点了,往四周走 loss都会比较高,你会不知道怎麽走到其他的地方去
- 但saddle point就比较没有这个问题,如果你今天是卡在saddle point的话,saddle point旁边还是有路可以走的,还是有路可以让你的loss更低的,你只要逃离saddle point,你就有可能让你的loss更低

所以鉴别今天我们走到,critical point的时候,到底是local minima,还是saddle point,是一个值得去探讨的问题,那怎麽知道今天一个critical point,到底是属于local minima,还是saddle point呢?

这边需要用到一点数学,以下这段其实没有很难的数学,就只是微积分跟线性代数,但如果你没有听懂的话,以下这段skip掉是没有关系的

那怎麽知道说一个点,到底是local minima,还是saddle point呢?

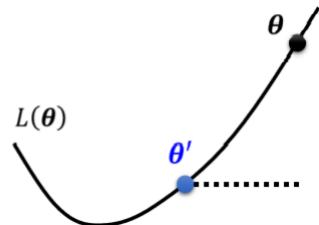
你要知道我们loss function的形状,可是我们怎麽知道,loss function的形状呢, network本身很复杂,用复杂network算出来的loss function,显然也很复杂,我们怎麽知道loss function,长什麼样子,虽然我们没有办法完整知道,整个loss function的样子

但是如果给定某一组参数,比如说蓝色的这个 θ' ,在 θ' 附近的loss function,是有办法被写出来的,它写出来就像是这个样子

Taylor Series Approximation

$L(\theta)$ around $\theta = \theta'$ can be approximated below

$$L(\theta) \approx L(\theta') + (\theta - \theta')^T g + \frac{1}{2} (\theta - \theta')^T H (\theta - \theta')$$



所以这个 $L(\theta)$ 完整的样子写不出来,但是它在 θ' 附近,你可以用这个式子来表示它,这个式子是,Taylor Series Appoximation泰勒级数展开,这个假设你在微积分的时候,已经学过了,所以我就不会细讲这一串是怎麽来的,但我们就只讲一下它的概念,这一串裡面包含什麼东西呢?

- 第一项是 $L(\theta')$,就告诉我们说,当 θ 跟 θ' 很近的时候, $L(\theta)$ 应该跟 $L(\theta')$ 还蛮靠近的

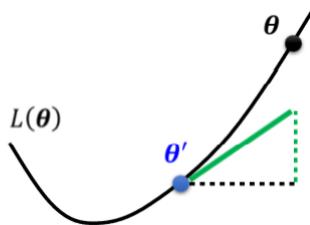
- 第二项是 $(\theta - \theta')^T g$

$L(\theta)$ around $\theta = \theta'$ can be approximated below

$$L(\theta) \approx L(\theta') + (\theta - \theta')^T g + \frac{1}{2} (\theta - \theta')^T H (\theta - \theta')$$

Gradient g is a vector

$$g = \nabla L(\theta') \quad g_i = \frac{\partial L(\theta')}{\partial \theta_i}$$



g 是一个向量,这个 g 就是我们的gradient,我们用绿色的这个 g 来代表gradient,这个gradient会来弥补 θ' 跟 θ 之间的差距,我们虽然刚才说 θ' 跟 θ ,它们应该很接近,但是中间还是有一些差距的,那这个差距,第一项我们用这个gradient,来表示他们之间的差距,有时候gradient会写成 $\nabla L(\theta')$,这个地方的 g 是一个向量,它的第*i*个component,就是 θ 的第*i*个component对L的微分,光是看 g 还是没有办法,完整的描述 $L(\theta)$,你还要看第三项

- 第三项跟Hessian有关,这边有一个 H

$L(\theta)$ around $\theta = \theta'$ can be approximated below

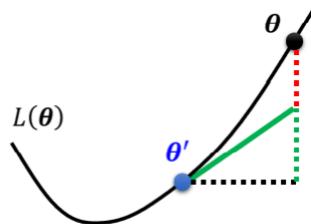
$$L(\theta) \approx L(\theta') + (\theta - \theta')^T g + \frac{1}{2} (\theta - \theta')^T H (\theta - \theta')$$

Gradient g is a vector

$$g = \nabla L(\theta') \quad g_i = \frac{\partial L(\theta')}{\partial \theta_i}$$

Hessian H is a matrix

$$H_{ij} = \frac{\partial^2}{\partial \theta_i \partial \theta_j} L(\theta')$$



这个 H 叫做Hessian,它是一个矩阵,这个第三项是,再 $(\theta - \theta')^T H (\theta - \theta')$,所以第三项会再补足,再加上gradient以后,与真正的 $L(\theta)$ 之间的差距. H 裡面放的是 L 的二次微分,它第*i*个row,第*j*个column的值,就是把 θ 的第*i*个component,对 L 作微分,再把 θ 的第*j*个component,对 L 作微分,再把 θ 的第*i*个component,对 L 作微分,做两次微分以后的结果就是这个 H_{ij}

如果这边你觉得有点听不太懂的话,也没有关系,反正你就记得这个 $L(\theta)$,这个loss function,这个error surface在 θ' 附近,可以写成这个样子,这个式子跟两个东西有关,跟gradient有关,跟hessian有关,gradient就是一次微分,hessian就是裡面有二次微分的项目

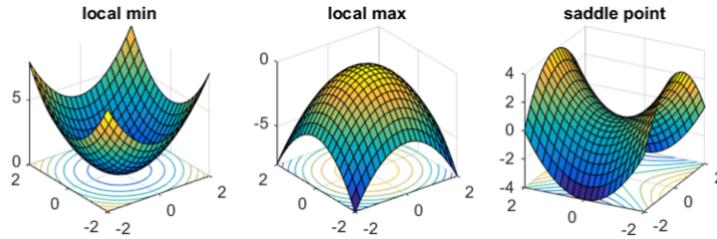
那如果我们今天走到了一个critical point,意味著gradient为零,也就是绿色的这一项完全都不见了

$L(\theta)$ around $\theta = \theta'$ can be approximated below

$$L(\theta) \approx L(\theta') + (\theta - \theta')^T g + \frac{1}{2} (\theta - \theta')^T H (\theta - \theta')$$

At critical point

telling the properties of critical points



g 是一个zero vector,绿色的这一项完全都不见了,只剩下红色的这一项,所以当在critical point的时候,这个loss function,它可以被近似为 $L(\theta')$,加上红色的这一项。我们可以根据红色的这一项来判断,在 θ' 附近的error surface,到底长什麼样子。知道error surface长什麼样子,我就可以判断。 θ' 它是一个local minima,是一个local maxima,还是一个saddle point。我们可以靠这一项来了解,这个error surface的地貌,大概长什麼样子,知道它地貌长什麼样子,我们就可以知道说,现在是在什麼样的状态,这个是Hessian那我们就来看一下怎麼根据Hessian,,怎麼根据红色的这一项,来判断 θ' 附近地貌

At critical point:

$$\text{Hessian} \quad L(\theta) \approx L(\theta') + \frac{1}{2} (\theta - \theta')^T H (\theta - \theta')$$

For all v

$$v^T H v > 0 \rightarrow \text{Around } \theta': L(\theta) > L(\theta') \rightarrow \text{Local minima}$$

For all v

$$v^T H v < 0 \rightarrow \text{Around } \theta': L(\theta) < L(\theta') \rightarrow \text{Local maxima}$$

$$\text{Sometimes } v^T H v > 0, \text{ sometimes } v^T H v < 0 \rightarrow \text{Saddle point}$$

我们现在為了等一下符号方便起见,我们把 $(\theta - \theta')$ 用 v 这个向量来表示

- 如果今天对任何可能的 v , $v^T H v$ 都大於零,也就是说 现在 θ 不管代任何值, v 可以是任何的 v ,也就是 θ 可以是任何值,不管 θ 代任何值,红色框框裡面通通都大於零,那意味著说 $L(\theta) > L(\theta')$ 。 $L(\theta)$ 不管代多少 只要在 θ' 附近, $L(\theta)$ 都大於 $L(\theta')$,代表 $L(\theta')$ 是附近的一个最低点,所以它是local minima
- 如果今天反过来说,对所有的 v 而言, $v^T H v$ 都小於零,也就是红色框框裡面永远都小於零,也就是说 θ 不管代什麼值,红色框框裡面都小於零,意味著说 $L(\theta) < L(\theta')$,代表 $L(\theta')$ 是附近最高的一个点,所以它是local maxima
- 第三个可能是假设, $v^T H v$ 有时候大於零 有时候小於零,你代不同的 v 进去 代不同的 θ 进去,红色这个框框裡面有时候大於零,有时候小於零,意味著说在 θ' 附近,有时候 $L(\theta) > L(\theta')$ 有时候 $L(\theta) < L(\theta')$,在 $L(\theta')$ 附近,有些地方高 有些地方低,这意味著什麼,这意味著这是一个saddle point

但是你这边是说我们要代所有的 v ,去看 $v^T H v$ 是大於零,还是小於零.我们怎麼有可能把所有的 v ,都拿来试试看呢,所以有一个更简便的方法,去确认说这一个条件或这一个条件,会不会发生.

这个就直接告诉你结论,线性代数理论上是有教过这件事情的,如果今天对所有的 v 而言, $v^T H v$ 都大於零,那这种矩阵叫做**positive definite 正定矩阵**,positive definite的矩阵,它所有的eigen value特征值都是正的

所以如果你今天算出一个hessian,你不需要把它跟所有的 v 都乘看看,你只要去直接看这个 H 的eigen value,如果你发现

- 所有eigen value都是正的,那就代表说这个条件成立,就 $v^T Hv$,会大於零,也就代表说是一个local minima。所以你从hessian metric可以看出,它是不是local minima,你只要算出hessian metric算完以后,看它的eigen value发现都是正的,它就是local minima。
- 那反过来说也是一样,如果今天在这个状况,对所有的v而言, $v^T Hv$ 小於零,那H是negative definite,那就代表所有eigen value都是负的,就保证他是local maxima
- 那如果eigen value有正有负,那就代表是saddle point,

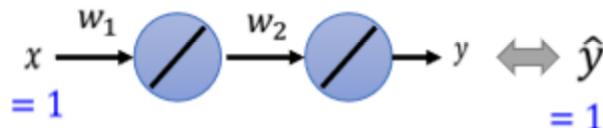
那假设在这裡你没有听得很懂的话,你就可以记得结论,你只要算出一个东西,这个东西的名字叫做hessian,它是一个矩阵,这个矩阵如果它所有的eigen value,都是正的,那就代表我们现在在local minima,如果它有正有负,就代表在saddle point。

那如果刚才讲的,你觉得你没有听得很懂的话,我们这边举一个例子

我们现在有一个史上最废的network,输入一个x,它只有一个neuron,乘上 w_1 ,而且这个neuron,还没有activation function,所以x乘上 w_1 以后之后就输出,然后再乘上 w_2 然后就再输出,就得到最终的数据就是y.总之这个function非常的简单

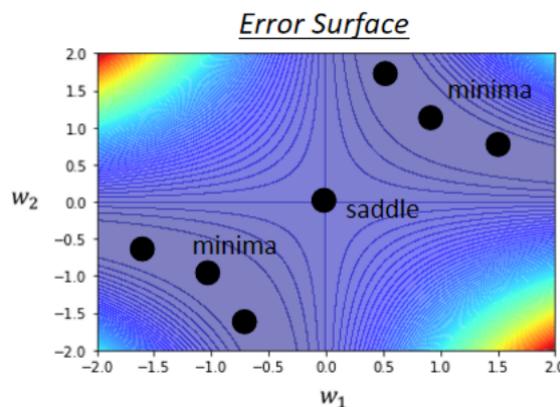
$$y = w_1 \times w_2 \times x$$

$$y = w_1 w_2 x$$



我们有一个史上最废的training set,这个data set说,我们只有一笔data,这笔data是x,是1的时候,它的level是1 所以输入1进去,你希望最终的输出跟1越接近越好

而这个史上最废的training,它的error surface,也是有办法直接画出来的,因为反正只有两个参数 w_1 w_2 ,连bias都没有,假设没有bias,只有 w_1 跟 w_2 两个参数,这个network只有两个参数 w_1 跟 w_2 ,那我们可以穷举所有 w_1 跟 w_2 的数值,算出所有 w_1 w_2 数值所代来的loss,然后就画出error surface 长这个样



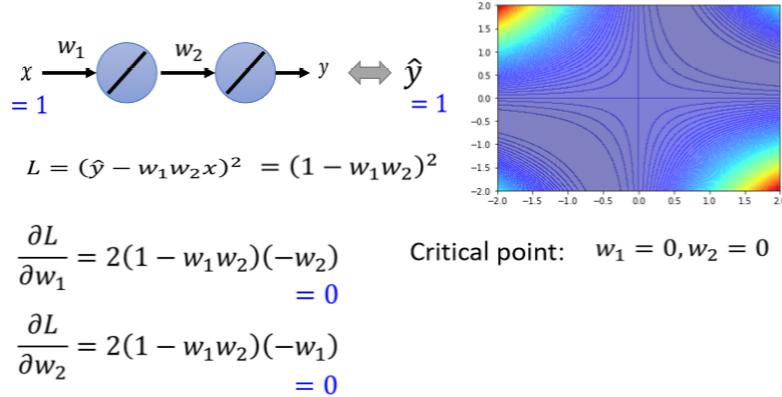
四个角落loss是高的,好 那这个图上你可以看出来说,有一些critical point,这个黑点点的地方(0,0),原点的地方是critical point,然后事实上,右上三个黑点也是一排critical point,左下三个点也是一排critical point

如果你更进一步要分析,他们是saddle point,还是local minima的话,那圆心这个地方,原点这个地方**它是saddle point**,为什麼它是saddle point呢

你往左上这个方向走 loss会变大,往右下这个方向走 loss会变大,往左下这个方向走 loss会变小,往右下这个方向走 loss会变小,它是一个saddle point

而这两群critical point,它们都是local minima,所以这个山沟裡面,有一排local minima,这一排山沟里面有一排local minima,然后在原点的地方,有一个saddle point,这个是我们把error surface,暴力所有的参数,得到的loss function以后,得到的loss的值以后,画出error surface,可以得到这样的结论

现在假设如果不暴力所有可能的loss,如果要直接算说一个点,是local minima,还是saddle point的话怎麽算呢



我们可以把loss的function写出来,这个loss的function 这个L是

$$L = (\hat{y} - w_1 w_2 x)^2$$

正确答案 \hat{y} 减掉model的输出,也就是 $w_1 w_2 x$,这边取square error,这边只有一笔data,所以就不会 summation over 所有的 training data,因为反正只有一笔data, x 代 1, \hat{y} 代 1, 我刚才说过只有一笔训练资料最废的, 所以只有一笔训练资料, 所以 loss function 就是 $L = (\hat{y} - w_1 w_2 x)^2$, 那你可以把这个 loss function, 它的 gradient 求出来, w_1 对 L 的微分, w_2 对 L 的微分写出来是这个样子

$$\frac{\partial L}{\partial w_1} = 2(1 - w_1 w_2)(-w_2)$$

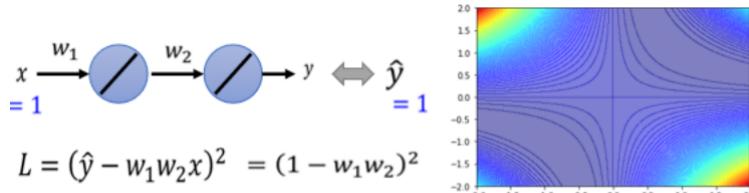
$$\frac{\partial L}{\partial w_2} = 2(1 - w_1 w_2)(-w_1)$$

这个东西

$$\begin{bmatrix} \frac{\partial L}{\partial w_1} \\ \frac{\partial L}{\partial w_2} \end{bmatrix}$$

就是所谓的 gradient, 什麼时候 gradient 会零呢, 什麼时候会到一个 critical point 呢?

举例来说 如果 $w_1=0$ $w_2=0$, 就在圆心这个地方, 如果 w_1 代 0 w_2 代 0, w_1 对 L 的微分 w_2 对 L 的微分, 算出来就都是零, 就都是零, 这个时候我们就知道说, 原点就是一个 critical point, 但它是 local maxima, 它是 local maxima, local minima, 还是 saddle point 呢, 那你就要看 hessian 才能够知道了



$$\begin{aligned} \frac{\partial L}{\partial w_1} &= 2(1 - w_1 w_2)(-w_2) && \text{Critical point: } w_1 = 0, w_2 = 0 \\ &= 0 && H = \begin{bmatrix} 0 & -2 \\ -2 & 0 \end{bmatrix} \quad \lambda_1 = 2, \lambda_2 = -2 \\ \frac{\partial L}{\partial w_2} &= 2(1 - w_1 w_2)(-w_1) && \text{Saddle point} \\ &= 0 && \end{aligned}$$

$$\frac{\partial^2 L}{\partial w_1^2} = 2(-w_2)(-w_2) = 0 \quad \frac{\partial^2 L}{\partial w_1 \partial w_2} = -2 + 4w_1 w_2 = -2$$

$$\frac{\partial^2 L}{\partial w_2 \partial w_1} = -2 + 4w_1 w_2 = -2 \quad \frac{\partial^2 L}{\partial w_2^2} = 2(-w_1)(-w_1) = 0$$

当然我们刚才已经暴力所有可能的 $w_1 w_2$ 了, 所以你已经知道说, 它显然是一个 saddle point, 但是现在假设还没有暴力所有可能的 loss, 所以我们要看看能不能够用 H , 用 Hessian 看出它是什麼样的 critical point, 那怎麼算出这个 H 呢

H 它是一个矩阵, 这个矩阵裡面元素就是 L 的二次微分, 所以这个矩阵裡面第一个 row, 第一个 column 的位置, 就是 w_1 对 L 微分两次, 第一个 row 第二个 column 的位置, 就是先用 w_2 对 L 作微分, 再用 w_1 对 L 作微分, 然后这边就是 w_1 对 L 作微分, w_2 对 L 作微分, 然后 w_2 对 L 微分两次, 这四个值组合起来, 就是我们的 hessian, 那这个 hessian 的值是多少呢

这个 hessian 的式子, 我都已经把它写出来了, 你只要把 $w_1 = 0$ $w_2 = 0$ 代进去, 代进去你就得到在原点的地方, hessian 是这样的一个矩阵

$$\begin{bmatrix} 0 & -2 \\ -2 & 0 \end{bmatrix}$$

这个 hessian 告诉我们, 它是 local minima, 还是 saddle point 呢, 那你就要看这个矩阵的 eigen value, 算一下发现, 这个矩阵有两个 eigen value, 2 跟 -2 eigen value 有正有负, 代表 saddle point

所以我们现在就是用一个例子, 跟你操作一下, 告诉你说, 你怎麼从 hessian 看出一个点, 它一个 critical point 它是 saddle point, 还是 local minima,

Don't afraid of saddle point

如果今天你卡的地方是 saddle point, 也许你就不用那麼害怕了, 因為如果你今天你发现, 你停下来的时候, 是因為 saddle point 停下来了, 那其实就有机会可以放心了

因為 H 它不仅可以帮助我们判断, 现在是不是在一个 saddle point, 它还指出了我们参数, 可以 update 的方向, 就之前我们参数 update 的时候, 都是看 gradient 看 g , 但是我们走到某个地方以后, 发现 g 变成 0 了, 不能再看 g 了, g 不见了, gradient 没有了, 但如果是一个 saddle point 的话, 还可以再看 H , 怎麼再看 H 呢, H 怎麼告诉我们, 怎麼 update 参数呢

$$v^T H v$$

$$\text{At critical point: } L(\theta) \approx L(\theta') + \frac{1}{2}(\theta - \theta')^T H(\theta - \theta')$$

Sometimes $v^T H v > 0$, sometimes $v^T H v < 0 \rightarrow$ Saddle point

H may tell us parameter update direction!

$$\begin{array}{l} u \text{ is an eigen vector of } H \\ \lambda \text{ is the eigen value of } u \\ \lambda < 0 \end{array} \longrightarrow u^T H u = u^T (\lambda u) = \lambda \|u\|^2 < 0 < 0$$

$$L(\theta) \approx L(\theta') + \frac{1}{2}(\theta - \theta')^T H(\theta - \theta') \rightarrow L(\theta) < L(\theta')$$

$$\theta - \theta' = u \quad \theta = \theta' + u \quad \text{Decrease } L$$

我们这边假设 μ 是 H 的eigen vector特征向量,然后 λ 是 u 的eigen value特征值。

如果我们把这边的 v 换成 μ 的话,我们把 μ 乘在 H 的左边,跟 H 的右边,也就是 $\mu^T H \mu$, $H \mu$ 会得到 $\lambda \mu$, 因為 μ 是一个eigen vector。 H 乘上eigen vector特征向量会得到特征向量 λ eigen value乘上eigen vector即 $\lambda \mu$, 所以我们在这边得到 u^T 乘上 λu ,然后再整理一下,把 u^T 跟 u 乘起来,得到 $\|\lambda u\|^2$,所以得到 $\lambda \|u\|^2$

$$u^T H u = u^T (\lambda u) = \lambda \|u\|^2$$

假设我们这边 v 代的是一个eigen vector,我们这边 θ 减 θ' ,放的是一个eigen vector的话,会发现说我们这个红色的项裡面,其实就是 $\lambda \|u\|^2$

$$\text{At critical point: } L(\theta) \approx L(\theta') + \frac{1}{2}(\theta - \theta')^T H(\theta - \theta')$$

Sometimes $v^T H v > 0$, sometimes $v^T H v < 0 \rightarrow$ Saddle point

H may tell us parameter update direction!

$$\begin{array}{l} u \text{ is an eigen vector of } H \\ \lambda \text{ is the eigen value of } u \\ \lambda < 0 \end{array} \longrightarrow u^T H u = u^T (\lambda u) = \lambda \|u\|^2 < 0 < 0$$

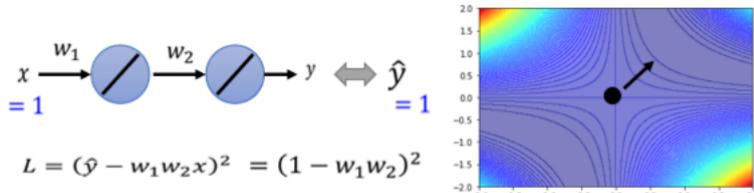
$$L(\theta) \approx L(\theta') + \frac{1}{2}(\theta - \theta')^T H(\theta - \theta') \rightarrow L(\theta) < L(\theta')$$

$$\theta - \theta' = u \quad \theta = \theta' + u \quad \text{Decrease } L$$

那今天如果 λ 小於零,eigen value小於零的话,那 $\lambda \|u\|^2$ 就会小於零,因為 $\|u\|^2$ 一定是正的,所以eigen value是负的,那这一整项就会是负的,也就是 u 的transpose乘上 H 乘上 u ,它是负的,也就是红色这个框裡是负的。所以这意思是说假设 $\theta - \theta' = \mu$,那这一项 $(\theta - \theta')^T H(\theta - \theta')$ 就是负的,也就是 $L(\theta) < L(\theta')$ 。也就是说假设 $\theta - \theta' = \mu$,也就是,你在 θ' 的位置加上 u ,沿著 u 的方向做update得到 θ ,你就可以让loss变小。因为根据这个式子,你只要 θ 减 θ' 等於 u ,loss就会变小,所以你今天只要让 θ 等於 θ' 加 u ,你就可以让loss变小,你只要沿著 u ,也就是eigen vector的方向,去更新你的参数 去改变你的参数,你就可以让loss变小了

所以虽然在critical point没有gradient,如果我们今天是在一个saddle point,你也不一定要惊慌,你只要找出负的eigen value,再找出它对应的eigen vector,用这个eigen vector去加 θ' ,就可以找到一个新的点,这个点的loss比原来还要低

举具体的例子



$$\frac{\partial L}{\partial w_1} = 2(1 - w_1 w_2)(-w_2) \quad \text{Critical point: } w_1 = 0, w_2 = 0$$

$$\frac{\partial L}{\partial w_2} = 2(1 - w_1 w_2)(-w_1) \quad H = \begin{bmatrix} 0 & -2 \\ -2 & 0 \end{bmatrix} \quad \lambda_1 = 2, \lambda_2 = -2$$

Saddle point

$$\lambda_2 = -2 \quad \text{Has eigenvector } \mathbf{u} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

Update the parameter along the direction of \mathbf{v}_2

You can escape the saddle point and decrease the loss.

(this method is seldom used in practice)

刚才我们已经发现,原点是一个critical point,它的Hessian长这个样,那我现在发现说,这个Hessian有一个负的eigen value,这个eigen value等於-2,那它对应的eigen vector,它有很多个,其实是无穷多个对应的eigen vector,我们就取一个出来,我们取 $\begin{bmatrix} 1 \\ 1 \end{bmatrix}$ 是它对应的一个eigen vector,那我们其实只要顺着这个u的方向,顺着 $\begin{bmatrix} 1 \\ 1 \end{bmatrix}$ 这个vector的方向,去更新我们的参数,就可以找到一个,比saddle point的loss还要更低的点

如果以今天这个例子来看的话,你的saddle point在(0,0)这个地方,你在这个地方会没有gradient,Hessian的eigen vector告诉我们,只要往 $\begin{bmatrix} 1 \\ 1 \end{bmatrix}$ 的方向更新,你就可以让loss变得更小,也就是说你可以逃离你的saddle point,然后让你的loss变小,所以从这个角度来看,似乎saddle point并没有那麼可怕

如果你今天在training的时候,你的gradient你的训练停下来,你的gradient变成零,你的训练停下来,是因为saddle point的话,那似乎还有解

但是当然实际上,在实际的implementation裡面,你几乎不会真的把Hessian算出来,这个要是二次微分,要计算这个矩阵的computation,需要的运算量非常非常的大,更遑论你还要把它的eigen value,跟eigen vector找出来,所以在实作上,你几乎没有看到,有人用这一个方法来逃离saddle point

等一下我们会讲其他,也有机会逃离saddle point的方法,他们的运算量都比要算这个H,还要小很多,那今天之所以我们把,这个saddle point跟 eigen vector,跟Hessian的eigen vector拿出来讲,是想要告诉你说,如果是卡在saddle point,也许没有那麼可怕,最糟的状况下你还有这一招,可以告诉你要往哪一个方向走.

Saddle Point v.s. Local Minima

讲到这边你就会有一个问题了,这个问题是,那到底saddle point跟local minima,谁比较常见呢,我们说,saddle point其实并没有很可怕,那如果我们今天,常遇到的是saddle point,比较少遇到local minima,那就太好了,那到底saddle point跟local minima,哪一个比较常见呢?这边我们要讲一个不相干的故事,先讲一个故事

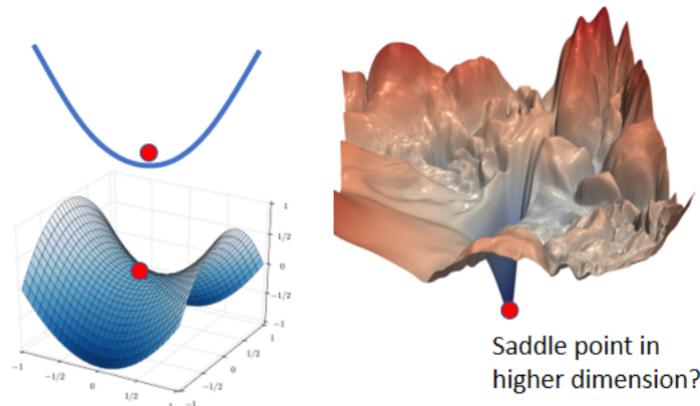
这个故事发生在1543年,1543年发生了什麼事呢,那一年君士坦丁堡沦陷,这个是君士坦丁堡沦陷图,君士坦丁堡本来是东罗马帝国的领土,然后被鄂图曼土耳其帝国佔领了,然后东罗马帝国就灭亡了,在鄂图曼土耳其人进攻,君士坦丁堡的时候,那时候东罗马帝国的国王,是君士坦丁十一世,他不知道要怎麼对抗土耳其人,有人就献上了一策,找来了一个魔法师叫做狄奥伦娜

这是真实的故事,出自三体的故事,这个狄奥伦娜这样说,狄奥伦娜是谁呢,他有一个能力跟张飞一样,张飞不是可以万军从中取上将首级,如探囊取物吗,狄奥伦娜也是一样,他可以直接取得那个苏丹的头,他可以从万军中取得苏丹的头,大家想说狄奥伦娜怎麼這麼厉害,他真的有這麼强大的魔法吗,所以大家就要狄奥伦娜,先展示一下他的力量,这时候狄奥伦娜就拿出了一个圣杯,大家看到这个圣杯就大吃一惊,為什麼大家看到这个圣杯,要大吃一惊呢,因為这个圣杯,本来是放在圣索菲亚大教堂的地下室,而且它是被放在一个石棺裡面,这个石棺是密封的,没有人可以打开它.

但是狄奥伦娜他从裡面取得了圣杯,而且还放了一串葡萄进去,君士坦丁十一世為了要验证,狄奥伦娜是不是真的有这个能力,就带了一堆人真的去撬开了这个石棺,发现圣杯真的被拿走了,裡面真的有一串新鲜的葡萄,就知道狄奥伦娜真的有,这个万军从中取上将首级的能力,那為什麼迪奥伦娜可以做到这些事呢,那是因為这个石棺你觉得它是封闭的,那是因為你是从三维的空间来看,从三维的空间来看,这个石棺是封闭的,没有任何路可以进去,但是狄奥伦娜可以进入四维的空间,从高维的空间中,这个石棺是有路可以进去的,它并不是封闭的,至於狄奥伦娜有没有成功刺杀苏丹呢,你可以想像一定是没有嘛,所以君坦丁堡才沦陷,那至於為什麼没有,大家请见於三体这样就不雷大家,

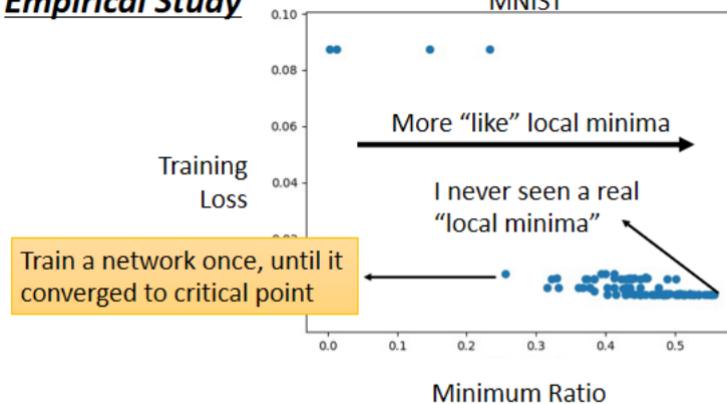
总之这个从三维的空间来看,是没有路可以走的东西,在高维的空间中是有路可以走的,error surface会不会也一样呢

所以你在一维的空间中,一维的一个参数的error surface,你会觉得好像到处都是local minima,但是会不会在二维空间来看,它就只是一个saddle point呢,常常会有人画类似这样的图,告诉你说Deep Learning的训练,是非常的复杂的,如果我们移动某两个参数,error surface的变化非常的复杂,是这个样子的,那显然它有非常多的local minima,我的这边现在有一个local minima,但是会不会这个local minima,只是在二维的空间中,看起来是一个local minima,在更高维的空间中,它看起来就是saddle point,在二维的空间中,我们没有路可以走,那会不会在更高的维度上,因為更高的维度,我们没办法visualize它,我们没办法真的拿出来看,会不会在更高维的空间中,其实有路可以走的,那如果维度越高,是不是可以走的路就越多了呢,所以今天我们在训练,一个network的时候,我们的参数往往动辄百万千万以上,所以我们的error surface,其实是在一个非常高的维度中,对不对,我们参数有多少,就代表我们的error surface的,维度有多少,参数是一千万就代表error surface,它的维度是一千万,竟然维度這麼高,会不会其实,根本就有非常多的路可以走呢,**那既然有非常多的路可以走,会不会其实local minima,根本就很少呢,**



而经验上,如果你自己做一些实验的话,也支持这个假说

Empirical Study



$$\text{Minimum ratio} = \frac{\text{Number of Positive Eigen values}}{\text{Number of Eigen values}}$$

这边是训练某一个network的结果,每一个点代表,训练那个network训练完之后,把它的Hessian拿出来进行计算,所以这边的每一个点都代表一个network,就我们训练某一个network,然后把它训练训练,训练到gradient很小,卡在critical point,把那组参数出来分析,看看它比较像是saddle point,还是比较像是local minima

- 纵轴代表training的时候的loss,就是我们今天卡住了,那个loss没办法再下降了,那个loss是多少,那很多时候,你的loss在还很高的时候,训练就不动了就卡在critical point,那很多时候loss可以降得很低,才卡在critical point,这是纵轴的部分
- 横轴的部分是minimum ratio,minimum ratio是eigen value的数目分之正的eigen value的数目,又如果所有的eigen value都是正的,代表我们今天的critical point,是local minima,如果有正有负代表saddle point,那在实作上你会发现说,你几乎找不到完全所有eigen value都是正的critical point,你看这边这个例子裡面,这个minimum ratio代表eigen value的数目分之正的eigen value的数目,最大也不过0.5到0.6间而已,代表说只有一半的eigen value是正的,还有一半的eigen value是负的,

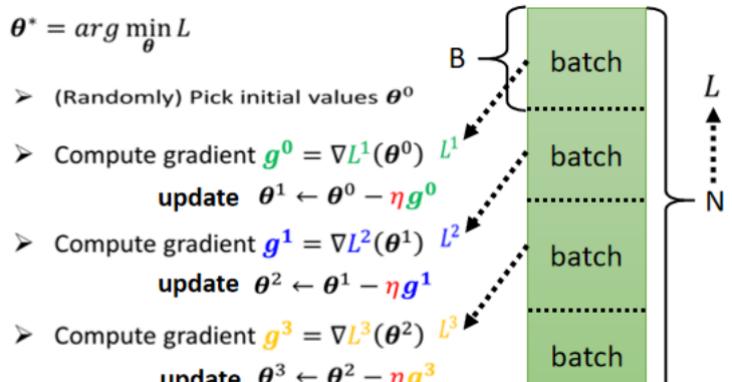
所以今天虽然在这个图上,越往右代表我们的critical point越像local minima,但是它们都没有真的变成local minima,就算是在最极端的状况,我们仍然有一半的case,我们的eigen value是负的,这一半的case eigen value是正的,代表说在所有的维度裡面有一半的路,这一半的路如果要让loss上升,还有一半的路可以让loss下降。

所以从经验上看起来,其实local minima并没有那麼常见,多数的时候,你觉得你train到一个地方,你gradient真的很小,然后所以你的参数不再update了,往往是因为你卡在了一个saddle point。

Batch and Momentum are helpful for critical point

Review: Optimization with Batch

上次我们有讲说,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,你是把所有的 Data 分成一个一个的 Batch,有的人是叫Mini Batch ,那我这边叫做 Batch,其实指的是一样的东西,助教投影片裡面,是写 Mini Batch



1 epoch = see all the batches once → **Shuffle** after each epoch

每一个 Batch 的大小呢,就是大 B 一笔的资料,我们每次在 Update 参数的时候,我们是拿大 B 一笔资料出来,算个 Loss,算个 Gradient,Update 参数,拿另外B一笔资料,再算个 Loss,再算个 Gradient,再 Update 参数,以此类推,所以我们不会拿所有的资料一起去算出 Loss,我们只会拿一个 Batch 的资料,拿出来算 Loss

所有的 Batch 看过一遍,叫做一个 Epoch,那事实上啊,你今天在做这些 Batch 的时候,你会做一件事情叫做 Shuffle

Shuffle 有很多不同的做法,但一个常见的做法就是,在每一个 Epoch 开始之前,会分一次 Batch,然后呢,每一个 Epoch 的 Batch 都不一样,就是第一个 Epoch,我们分这样子的 Batch,第二个 Epoch,会重新再分一次 Batch,所以哪些资料在同一个 Batch 裡面,每一个 Epoch 都不一样的这件事情,叫做 Shuffle。

Small Batch v.s. Large Batch

我们先解释為什麼要用 Batch,再说 Batch 对 Training 带来了什麼样的帮助。

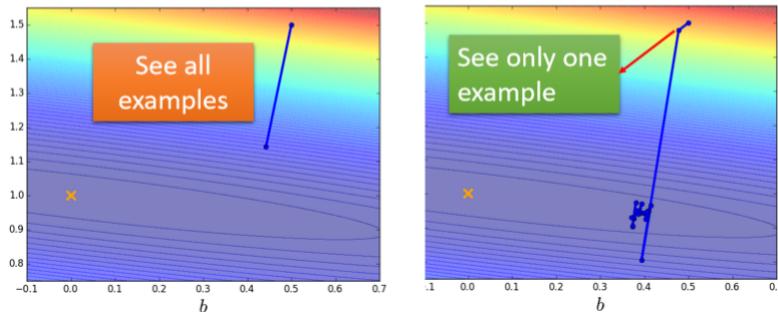
Consider 20 examples (N=20)

Batch size = N (Full Batch)

Update after seeing all
the 20 examples

Batch size = 1

Update for each example



我们来比较左右两边这两个 Case,那假设现在我们有20笔训练资料

- 左边的 Case 就是没有用 Batch,Batch Size,直接设的跟我训练资料一样多,这种状况叫做 Full Batch,就是没有用 Batch 的意思
- 那右边的 Case 就是,Batch Size 等於1

这是两个最极端的状况

我们先来看左边的 Case,在左边 Case 裡面,因為没有用 Batch,我们的 Model 必须把20笔训练资料都看完,才能够计算 Loss,才能够计算 Gradient,所以我们必须要把所有20笔 Examples 都看完以后,我们的参数才能够 Update 一次。就假设开始的地方在上边,把所有资料都看完以后,Update 参数就从这裡移动到下边。

如果 Batch Size 等於1的话,代表我们只需要拿一笔资料出来算 Loss,我们就可以 Update 我们的参数,所以每次我们 Update 参数的时候,看一笔资料就好,所以我们开始的点在这边,看一笔资料就 Update 一次参数,再看一笔资料就 Update 一次参数,如果今天总共有20笔资料的话 那在每一个 Epoch 裡面,我们的参数会 Update 20次,那不过,因为我们现在是只看一笔资料,就 Update 一次参数,所以用一笔资料算出来的 Loss,显然是比较 Noisy 的,所以我们今天 Update 的方向,你会发现它是曲曲折折的

所以如果我们比较左边跟右边, 哪一个比较好呢,他们有什麼差别呢?

你会发现左边没有用 Batch 的方式,它蓄力的时间比较长,还有它技能冷却的时间比较长,你要把所有的资料都看过一遍,才能够 Update 一次参数

而右边的这个方法,Batch Size 等於1的时候,蓄力的时间比较短,每次看到一笔参数,每次看到一笔资料,你就会更新一次你的参数

所以今天假设有20笔资料,看完所有资料看过一遍,你已经更新了20次的参数,但是左边这样子的方法有一个优点,就是它这一步走的是稳的,那右边这个方法它的缺点,就是它每一步走的是不稳的

看起来左边的方法跟右边的方法,他们各自都有擅长跟不擅长的东西,左边是蓄力时间长,但是威力比较大,右边技能冷却时间短,但是它是比较不準的,看起来各自有各自的优缺点,但是你会觉得说,左边的方法技能冷却时间长,右边的方法技能冷却时间短,那只是你没有考虑并行运算的问题。

实际上考虑并行运算的话,左边这个并不一定时间比较长

Larger batch size does not require longer time to compute gradient

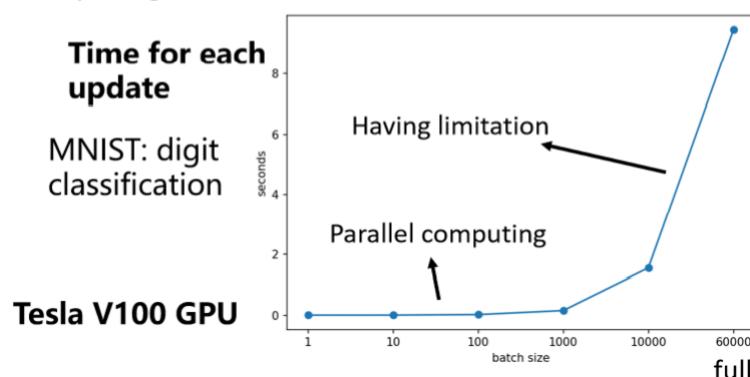
这边是真正的实验结果了,事实上,比较大的 Batch Size,你要算 Loss,再进而算 Gradient,所需要的时间,不一定比小的 Batch Size 要花的时间长

那以下是做在一个叫做 MNIST上面,MNIST (Mixed National Institute of Standards and Technology database)是美国国家标准与技术研究院收集整理的大型手写数字数据库,机器要做的事情,就是给它一张图片,然后判断这张图片,是0到9的哪一个数字,它要做数字的分类,那 MNIST 呢 是机器学习的helloworld,就是假设你今天,从来没有做过机器学习的任务,一般大家第一个会尝试的机器学习的任务,往往就是做 MNIST 做手写数字辨识,

oldest slides: [http://speech.ee.ntu.edu.tw/~tlkagk/courses/MLDS_2015_2/Lecture/DNN%20\(v4\).pdf](http://speech.ee.ntu.edu.tw/~tlkagk/courses/MLDS_2015_2/Lecture/DNN%20(v4).pdf)
old slides: http://speech.ee.ntu.edu.tw/~tlkagk/courses/ML_2017/Lecture/Keras.pdf

Small Batch v.s. Large Batch

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



这边我们就是做了一个实验,我们想要知道说,给机器一个 Batch,它要计算出 Gradient,进而 Update 参数,到底需要花多少的时间。这边列出了 Batch Size 等於1 等於10,等於100 等於1000 所需要耗费的时间。你会发现说 Batch Size 从1到1000,需要耗费的时间几乎是一样的,你可能直觉上认为有1000笔资料,那需要计算 Loss,然后计算 Gradient,花的时间不会是一笔资料的1000倍吗,但是实际上并不是这样的

因為在实际上做运算的时候,我们有 GPU,可以做并行运算,是因為你可以做平行运算的关係,这1000笔资料是平行处理的,所以1000笔资料所花的时间,并不是一笔资料的1000倍,当然 GPU 平行运算的能力还是有它的极限,当你的 Batch Size 真的非常非常巨大的时候,GPU 在跑完一个 Batch,计算出 Gradient 所花费的时间,还是会随著 Batch Size 的增加,而逐渐增长

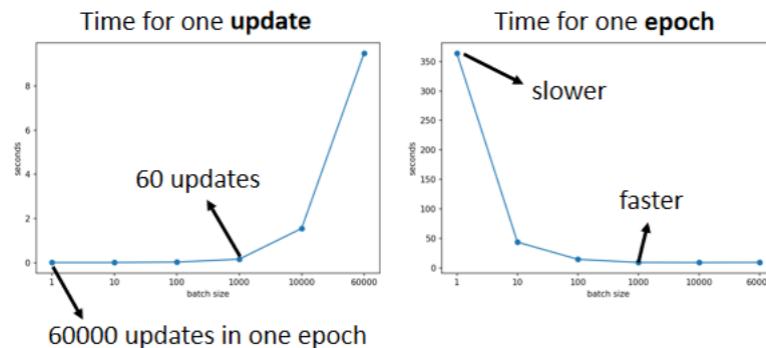
所以今天如果 Batch Size 是从1到1000,所需要的时间几乎是一样的,但是当你的 Batch Size 增加到10000,乃至增加到60000的时候,你就会发现 GPU 要算完一个 Batch,把这个 Batch 裡面的资料都拿出来算 Loss,再进而算 Gradient,所要耗费的时间,确实有随著 Batch Size 的增加而逐渐增长,但你会发现这边用的是 V100,所以它挺厉害的,给它60000笔资料,一个 Batch 裡面,塞了60000笔资料,它在10秒鐘之内,也是把 Gradient 就算出来

而那这个 Batch Size 的大小跟时间的关係,其实每年都会做这个实验,我特别把旧的投影片放在这边了,如果你有兴趣的话,,可以看到这个时代的演进这样,17年的时候用的是那个980啊,2015年的时候用的是那个760啊,然后980要跑什麼60000个 Batch,那要跑好几分鐘才跑得完啊,现在只要10秒鐘就可以跑得完了,你可以看到这个时代的演进,

Smaller batch requires longer time for one epoch

所以 GPU 虽然有平行运算的能力,但它平行运算能力终究是有个极限,所以你 Batch Size 真的很大的时候,时间还是会增加的

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



但是因為有平行运算的能力,因此实际上,当你的 Batch Size 小的时候,你要跑完一个 Epoch,花的时间是比大的 Batch Size 还要多的,怎麼说呢

如果今天假设我们的训练资料只有60000笔,那 Batch Size 设1,那你要60000个 Update 才能跑完一个 Epoch,如果今天是 Batch Size 等於1000,你要60个 Update 才能跑完一个 Epoch,假设今天一个 Batch Size 等於1000,要算 Gradient 的时间根本差不多,那60000次 Update,跟60次 Update 比起来,它的时间的差距量就非常可观了

所以左边这个图是 Update 一次参数,拿一个 Batch 出来计算一个 Gradient,Update 一次参数所需要的时间,右边这个图是,跑完一个完整的 Epoch,需要花的时间,你会发现左边的图跟右边的图,它的趋势正好是相反的,假设你 Batch Size 这个1,跑完一个 Epoch,你要 Update 60000次参数,它的时间是非常可观的,但是假设你的 Batch Size 是1000,你只要跑60次,Update 60次参数就会跑完一个 Epoch,所以你跑完一个 Epoch,看完所有资料的时间,如果你的 Batch Size 设1000,其实是比较短的,Batch Size 设1000的时候,把所有的资料看过一遍,其实是比 Batch Size 设1 还要更快

所以如果我们看右边这个图的话,看完一个 Batch,把所有的资料看过一次这件事情,大的 Batch Size 反而是较有效率的,是不是跟你直觉想的不太一样

在没有考虑平行运算的时候,你觉得大的 Batch 比较慢,但实际上,在有考虑平行运算的时候,一个 Epoch 大的 Batch 花的时间反而是比较少的

我们如果要比较这个 Batch Size 大小的差异的话,看起来直接用技能时间冷却的长短,并不是一个精确的描述,看起来在技能时间上面,大的 Batch 并没有比较吃亏,甚至还佔到优势了.

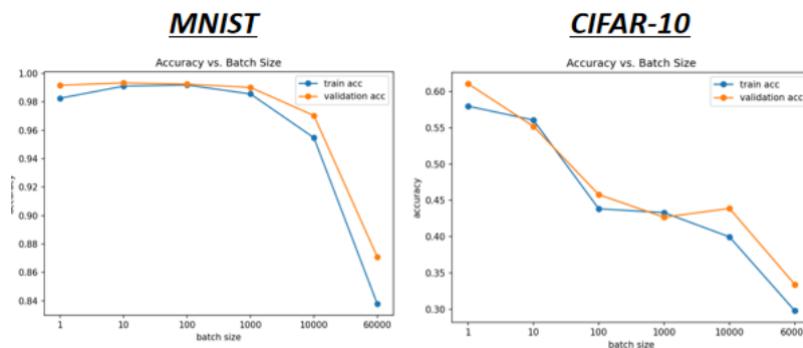
所以事实上,20笔资料 Update 一次的时间,跟右边看一笔资料 Update 一次的时间,如果你用 GPU 的话,其实可能根本就是所以一样的,所以大的 Batch,它的技能时间,它技能冷却的时间,并没有比较长,那所以这时候你可能就会说,欸 那个大的 Batch 的劣势消失了,那难道它真的就,那这样看起来大的 Batch 应该比较好?

你不是说大的 Batch,这个 Update 比较稳定,小的 Batch,它的 Gradient 的方向比较 Noisy 吗,那这样看起来,大的 Batch 好像应该比较好哦,小的 Batch 应该比较差,因為现在大的 Batch 的劣势已经,因為平行运算的时间被拿掉了,它好像只剩下优势而已.

那神奇的地方是 **Noisy 的 Gradient,反而可以帮助 Training**,这个也是跟直觉正好相反的

“Noisy” update is better for training

如果你今天拿不同的 Batch 来训练你的模型,你可能会得到这样子的结果,左边是坐在 MNIST 上,右边是坐在 CIFAR-10 上,不管是 MNIST 还是 CIFAR-10,都是影像辨识的问题



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

- 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,越上面正确率越高,当然正确率越高越好

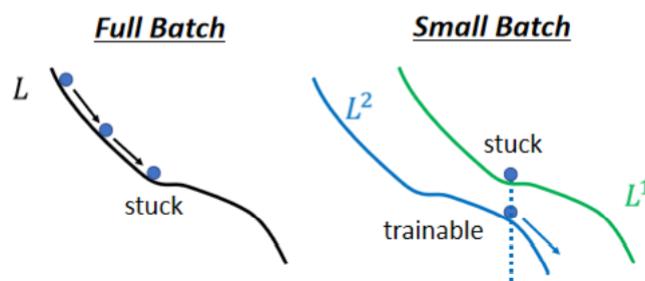
而如果你今天看 Validation Acc 上的结果,会发现说,Batch Size 越大,Validation Acc 上的结果越差,但这个不是 Overfitting,因為如果你看你的 Training 的话,会发现说 Batch Size 越大,Training 的结果也是越差的,而我们现在用的是同一个模型哦,照理说,它们可以表示的 Function 就是一模一样的

但是神奇的事情是,大的 Batch Size,往往在 Training 的时候,会给你带来比较差的结果

所以这个是什麼样的问题,同样的 Model,所以这个不是 Model Bias 的问题,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,好 為什麼会这样子呢

為什麼小的 Batch Size,在 Training Set 上会得到比较好的结果,為什麼 Noisy 的 Update,Noisy 的 Gradient 会在 Training 的时候,给我们比较好的结果呢? 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, 那你今天在 Update 你的参数的时候, 你就是沿著一个 Loss Function 来 Update 参数, 今天 Update 参数的时候走到一个 Local Minima, 走到一个 Saddle Point, 显然就停下来了, Gradient 是零, 如果你不特别去看 Hession 的话, 那你用 Gradient Descent 的方法, 你就没有办法再更新你的参数了。

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, 所以等於是, 等於你每一次 Update 你的参数的时候, 你用的 Loss Function 都是越有差异的, 你选到第一个 Batch 的时候, 你是用 L1 来算你的 Gradient, 你选到第二个 Batch 的时候, 你是用 L2 来算你的 Gradient, 假设你用 L1 算 Gradient 的时候, 发现 Gradient 是零, 卡住了, 但 L2 它的 Function 跟 L1 又不一样, L2 就不一定会卡住, 所以 L1 卡住了没关係, 换下一个 Batch 来, L2 再算 Gradient。

你还是有办法 Training 你的 Model, 还是有办法让你的 Loss 变小, 所以今天这种 Noisy 的 Update 的方式, 结果反而对 Training, 其实是有帮助的。

“Noisy” update is better for generalization

那这边还有另外一个更神奇的事情, 其实小的 Batch 也对 Testing 有帮助。

假设我们今天在 Training 的时候, 都不管是大的 Batch 还小的 Batch, 都 Training 到一样好, 刚才的 Case 是 Training 的时候就已经 Training 不好了。

假设你有一些方法, 你努力的调大的 Batch 的 Learning Rate, 然后想办法把大的 Batch, 跟小的 Batch Training 得一样好, 结果你会发现小的 Batch, 居然在 Testing 的时候会是比较好的, 那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

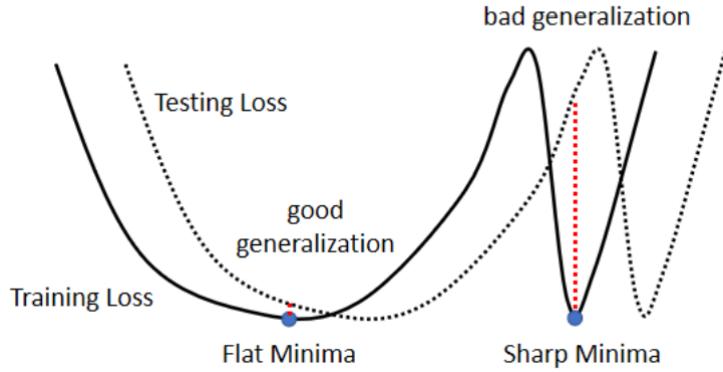
	Name	Network Type	Data set
SB = 256	F_1	Fully Connected	MNIST (LeCun et al., 1998a)
	F_2	Fully Connected	TIMIT (Garofolo et al., 1993)
LB = 0.1 x data set	C_1	(Shallow) Convolutional	CIFAR-10 (Krizhevsky & Hinton, 2009)
	C_2	(Deep) Convolutional	CIFAR-10
	C_3	(Shallow) Convolutional	CIFAR-100 (Krizhevsky & 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 裡面, 作者 Train 了六个 Network 裡面有 CNN 的, 有 Fully Connected Network 的, 做在不同的 Cover 上, 来代表这个实验是很泛用的, 在很多不同的 Case 都观察到一样的结果, 那它有小的 Batch, 一个 Batch 裡面有 256 笔 Example, 大的 Batch 就是那个 Data Set 乘 0.1, Data Set 乘 0.1, Data Set 有 60000 笔, 那你就是一个 Batch 裡面有 6000 笔资料。

然后他想办法, 在大的 Batch 跟小的 Batch, 都 Train 到差不多的 Training 的 Accuracy, 所以刚才我们看到的结果是, Batch Size 大的时候, Training Accuracy 就已经差掉了, 这边不是想办法 Train 到大的 Batch 的时候, Training Accuracy 跟小的 Batch, 其实是差不多的。但是就算是在 Training 的时候结果差不多, Testing 的时候你还是看到了, 小的 Batch 居然比大的 Batch 差, Training 的时候都很好, Testing 的时候小的 Batch 差, 代表 Over Fitting, 这个才是 Over Fitting 对不对, 好那为什麼会有这样子的现象呢? 在这篇文章裡面也给出了一个解释,

- “Noisy” update is better for generalization



假设这个是我们的 Training Loss,那在这个 Training Loss 上面呢,可能有很多个 Local Minima,有不只有一个 Local Minima,那这些 Local Minima 它们的 Loss 都很低,它们 Loss 可能都趋近於 0,但是这个 Local Minima,还是有好 Minima 跟坏 Minima 之分

如果一个 Local Minima 它在一个峡谷裡面,它是坏的 Minima,然后它在一个平原上,它是好的 Minima,為什麼会有这样的差异呢

- 因为假设现在 Training 跟 Testing 中间,有一个 Mismatch,Training 的 Loss 跟 Testing 的 Loss,它们那个 Function 不一样,有可能是本来你 Training 跟 Testing 的 Distribution 就不一样。
- 那也有可能是因为 Training 跟 Testing,你都是从 Sample 的 Data 算出来的,也许 Training 跟 Testing,Sample 到的 Data 不一样,那所以它们算出来的 Loss,当然是有一点差距。

那我们就假设说这个 Training 跟 Testing,它的差距就是把 Training 的 Loss,这个 Function 往右平移一点,这时候你会发现,对左边这个在一个盆地裡面的 Minima 来说,它的在 Training 跟 Testing 上面的结果,不会差太多,只差了一点点,但是对右边这个在峡谷裡面的 Minima 来说,一差就可以天差地远

它在这个 Training Set 上,算出来的 Loss 很低,但是因为 Training 跟 Testing 之间的不一样,所以 Testing 的时候,这个 Error Surface 一变,它算出来的 Loss 就变得很大,而很多人相信这个**大的 Batch Size,会让我们倾向於走到峡谷裡面,而小的 Batch Size,倾向於让我们走到盆地裡面**

那他直觉上的想法是这样,就是小的 Batch,它有很多的 Loss,它每次 Update 的方向都不太一样,所以如果今天这个峡谷非常地窄,它可能一个不小心就跳出去了,因为每次 Update 的方向都不太一样,它的 Update 的方向也就随机性,所以一个很小的峡谷,没有办法困住小的 Batch

如果峡谷很小,它可能动一下就跳出去,之后停下来如果有一个人非常宽的盆地,它才会停下来,那对于大的 Batch Size,反正它就是顺着规定 Update,然后它就很有可能,走到一个比较小的峡谷裡面

但这只是一个解释,那也不是每个人都相信这个解释,那这个其实还是一个**尚待研究的问题**

那这边就是比较了一下,大的 Batch 跟小的 Batch

	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.

左边这个是第一个 Column 是小的 Batch,第二个 Column 是大的 Batch

在有平行运算的情况下,小的 Batch 跟大的 Batch,其实运算的时间并没有太大的差距,除非你的大的 Batch 那个大是真的非常大,才会显示出差距来。但是一个 Epoch 需要的时间,小的 Batch 比较长,大的 Batch 反而是比较快的,所以从一个 Epoch 需要的时间来看,大的 Batch 其实是佔到优势的。

而小的 Batch,你会 Update 的方向比较 Noisy,大的 Batch Update 的方向比较稳定,但是 Noisy 的 Update 的方向,反而在 Optimization 的时候会佔到优势,而且在 Testing 的时候也会佔到优势,所以大的 Batch 跟小的 Batch,它们各自有它们擅长的地方

所以 Batch Size,变成另外一个 你需要去调整的 Hyperparameter。

那我们能不能够鱼与熊掌兼得呢,我们能不能够截取大的 Batch 的优点,跟小的 Batch 的优点,我们用大的 Batch Size 来做训练,用平行运算的能力来增加训练的效率,但是训练出来的结果同时又得到好的结果呢,又得到好的训练结果呢。

Have both fish and bear's paws?

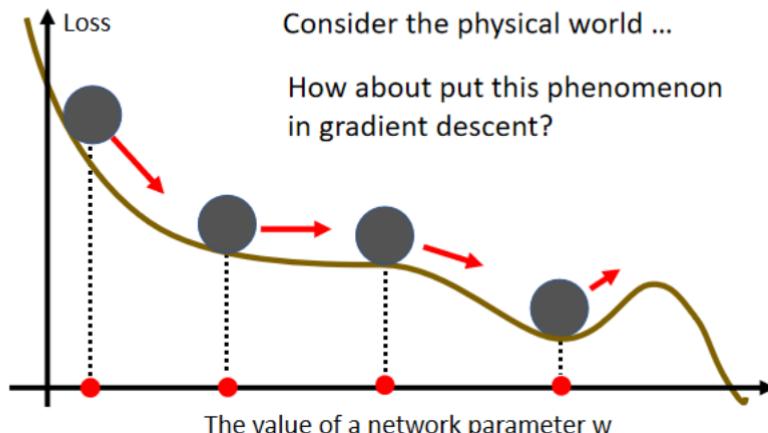
- Large Batch Optimization for Deep Learning: Training BERT in 76 minutes (<https://arxiv.org/abs/1904.00962>)
- Extremely Large Minibatch SGD: Training ResNet-50 on ImageNet in 15 Minutes (<https://arxiv.org/abs/1711.04325>)
- Stochastic Weight Averaging in Parallel: Large-Batch Training That Generalizes 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>)

这是有可能的,有很多文章都在探讨这个问题,那今天我们就不细讲,我们把这些 Reference 列在这边给大家参考,那你发现这些 Paper,往往它想要做的事情都是什麼,哇 76分鐘 Train BERT,15分鐘 Train ResNet,一分鐘 Train Imagenet 等等,这為什麼他们可以做到那麼快,就是因為他们 Batch Size 是真的开很大,比如说在第一篇 Paper 裡面,Batch Size 裡面有三万笔 Example 这样,Batch Size 开很大,Batch Size 开大 真的就可以算很快,你可以在很短的时间内看到大量的资料,那他们需要有一些特别的方法来解决,Batch Size 可能会带来的劣势。

Momentum

Momentum,这也是另外一个,有可能可以对抗 Saddle Point,或 Local Minima 的技术,Momentum 的运作是这个样子的

Small Gradient



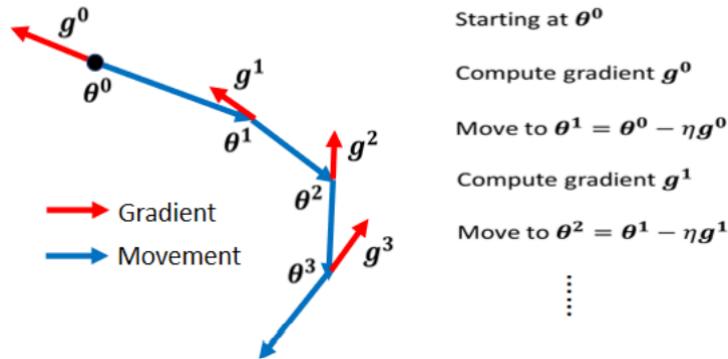
它的概念,你可以想像成在物理的世界裡面,假设 Error Surface 就是真正的斜坡,而我们的参数是一个球,你把球从斜坡上滚下来,如果今天是 Gradient Descent,它走到 Local Minima 就停住了,走到 Saddle Point 就停住了

但是在物理的世界裡,一个球如果从高处滚下来,从高处滚下来就算滚到 Saddle Point,如果有惯性,它从左边滚下来,因為惯性的关係它还是会继续往右走,甚至它走到一个 Local Minima,如果今天它的动量够大的话,它还是会继续往右走,甚至翻过这个小坡然后继续往右走

那所以今天在物理的世界裡面,一个球从高处滚下来的时候,它并不会被 Saddle Point,或 Local Minima 卡住,不一定会被 Saddle Point,或 Local Minima 卡住,我们有没有办法运用这样子的概念,到 Gradient Descent 裡面呢,那这个就是我们等一下要讲的,Momentum 这个技术

Vanilla Gradient Descent

那我们先很快的复习一下,原来的 Gradient Descent 长得是什麼样子,这个是 Vanilla 的 Gradient Descent,Vanilla 的意思就是一般的的意思,它直译是香草的,但就其实是一般的,一般的 Gradient Descent 长什麼样子呢



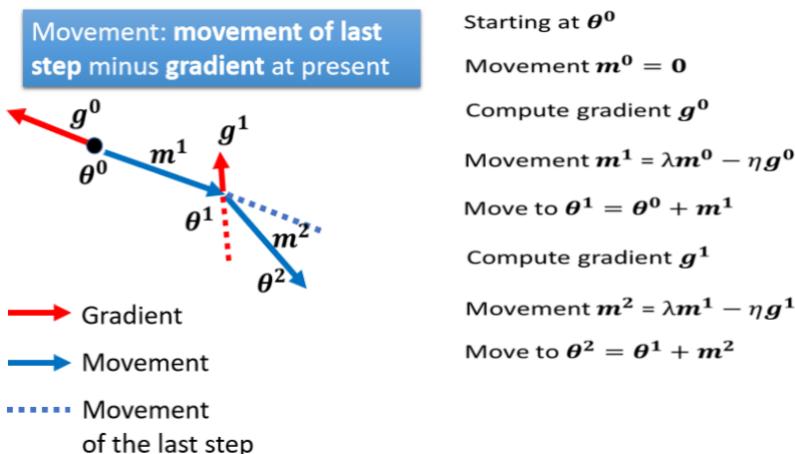
一般的 Gradient Descent 是说,我们有一个初始的参数叫做 θ^0 ,我们计算一下 Gradient,然后计算完这个 Gradient 以后呢,我们往 Gradient 的反方向去 Update 参数

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

我们到了新的参数以后,再计算一次 Gradient,再往 Gradient 的反方向,再 Update 一次参数,到了新的位置以后再计算一次 Gradient,再往 Gradient 的反方向去 Update 参数,这个 Process 就一直这样子下去

Gradient Descent + Momentum

加上 Momentum 以后,每一次我们在移动我们的参数的时候,我们不是只往 Gradient Descent,我们不是只往 Gradient 的反方向来移动参数,我们是 **Gradient 的反方向,加上前一步移动的方向,两者加起来的结果,去调整去到我们的参数**,



那具体说起来是这个样子,一样找一个初始的参数,然后我们假设前一步的参数的 Update 量呢,就设为 0

$$m^0 = 0$$

接下来在 θ^0 的地方,计算 Gradient 的方向 g^0

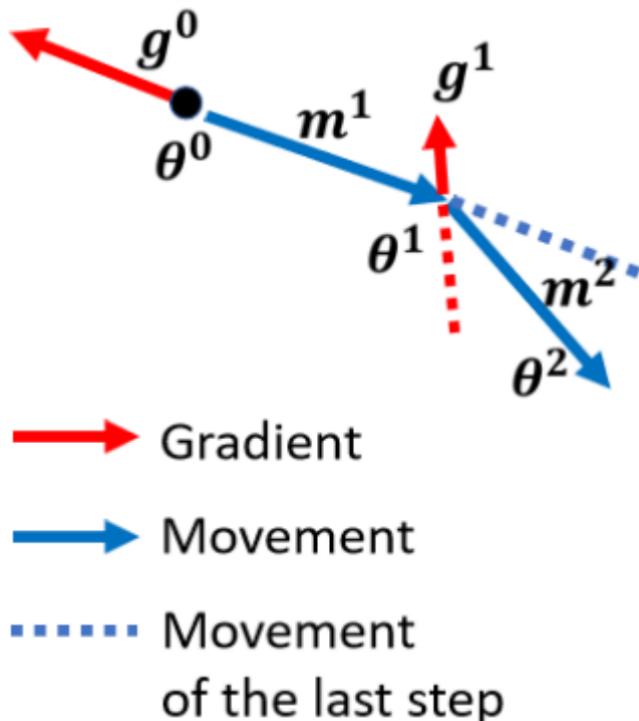
然后接下来你要决定下一步要怎麼走,它是 Gradient 的方向加上前一步的方向,不过因為前一步正好是 0,现在是刚初始的时候所以前一步是 0,所以 Update 的方向,跟原来的 Gradient Descent 是一样的,这没有什麼有趣的地方

$$\begin{aligned} m^1 &= \lambda m^0 - \eta g^0 \\ \theta^1 &= \theta^0 + m^1 \end{aligned}$$

但从第二步开始,有加上 Momentum 以后就不太一样了,从第二步开始,我们计算 g^1 ,然后接下来我们 Update 的方向,不是 g^1 的反方向,而是根据上一次 Update 方向,也就是 m^1 减掉 g^1 ,当做我们新的 Update 的方向,这边写成 m^2

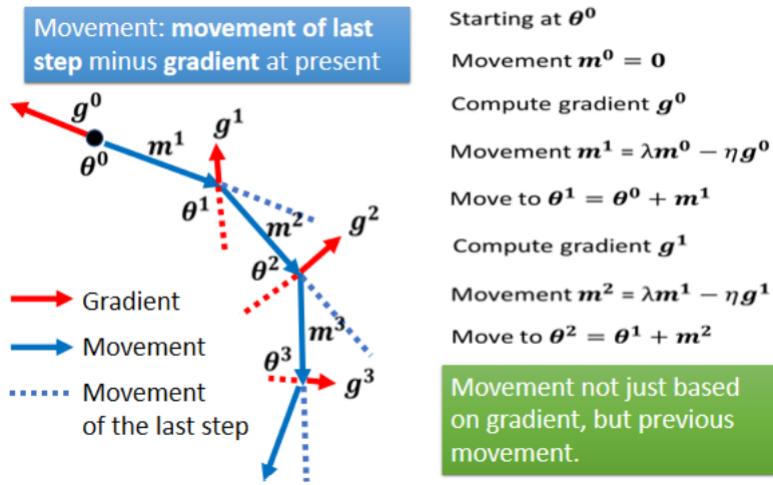
$$m^2 = \lambda m^1 - \eta g^1$$

那我们就看下面这个图

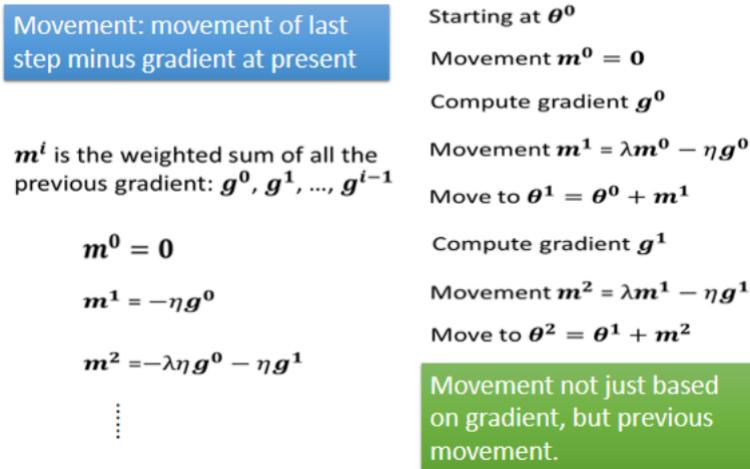


g^1 告诉我们,Gradient 告诉我们要往红色反方向这边走,但是我们不是只听 Gradient 的话,加上 Momentum 以后,我们不是只根据 Gradient 的反方向,来调整我们的参数,我们也会看前一次 Update 的方向

- 如果前一次说要往 m^1 蓝色及蓝色虚线这个方向走
- Gradient 说要往红色反方向这个方向走
- 把两者相加起来,走两者的折中,也就是往蓝色 m^2 这一个方向走,所以我们就移动了 m^2 ,走到 θ^2 这个地方



接下来就反覆进行同样的过程,在这个位置我们计算出 Gradient,但我们不是只根据 Gradient 反方向走,我们看前一步怎麼走,前一步走这个方向,走这个蓝色虚线的方向,我们把蓝色的虚线加红色的虚线,前一步指示的方向跟 Gradient 指示的方向,当做我们下一步要移动的方向



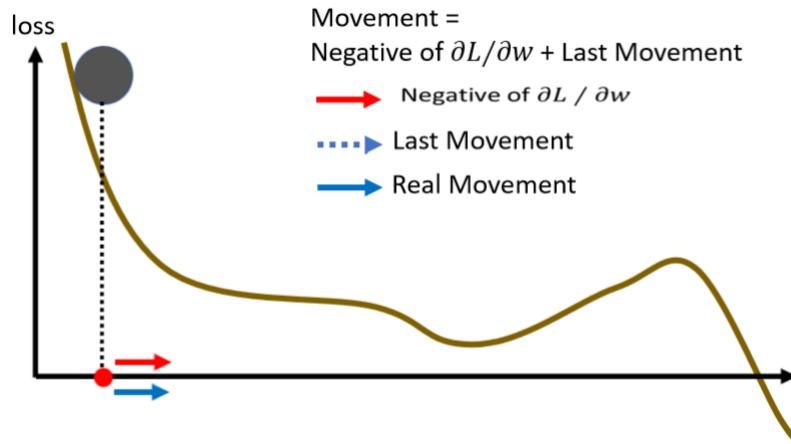
每一步的移动,我们都用 m 来表示,那这个 m 其实可以写成之前所有算出来的,Gradient 的 Weighted Sum.从右边的这个式子,其实就可以轻易的看出来

$$\begin{aligned}m^0 &= 0 \\m^1 &= -\eta g^0 \\m^2 &= -\lambda \eta g^0 - \eta g^1 \\&\vdots\end{aligned}$$

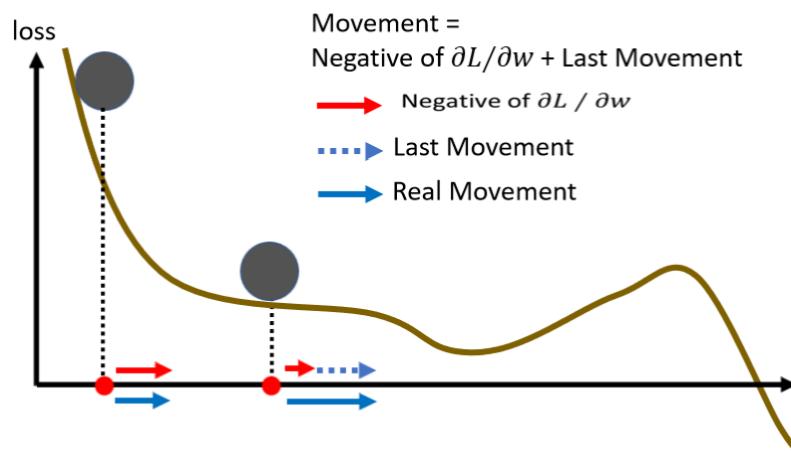
m^0 我们把它设為 0, m^1 是 m^0 减掉 g^0 , m^0 為 0,所以 m^1 就是 g^0 乘上负的 η , m^2 是 λ 乘上 m^1 , λ 就是另外一个参数,就好像 η 是 Learning Rate 我们要调, λ 是另外一个参数,这个也是需要调的, m^2 等於 λ 乘上 m^1 ,减掉 η 乘上 g^1 ,然后 m^1 在哪裡呢, m^1 在这边,你把 m^1 代进来,就知道说 m^2 ,等於负的 λ 乘上 η 乘以 g^0 ,减掉 η 乘上 g^1 ,它是 g^0 跟 g^1 的 Weighted Sum

以此类推,所以你会发现说,现在这个加上 Momentum 以后,一个解读是 Momentum 是,Gradient 的负反方向加上前一次移动的方向,那但另外一个解读方式是,所谓的 Momentum,当加上 Momentum 的时候,我们 Update 的方向,不是只考虑现在的 Gradient,而是考虑过去所有 Gradient 的总合.

有一个更简单的例子,希望帮助你了解 Momentum

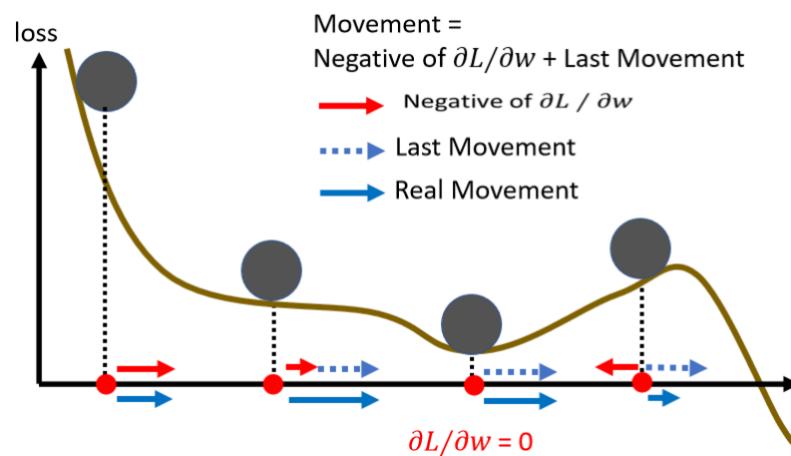


那我们从这个地方开始 Update 参数,根据 Gradient 的方向告诉我们,应该往右 Update 参数,那现在没有前一次 Update 的方向,所以我们就完全按照 Gradient 给我们的指示,往右移动参数,好那我们的参数,就往右移动了一点到这个地方



Gradient 变得很小,告诉我们往右移动,但是只有往右移动一点点,但前一步是往右移动的,我们把前一步的方向用虚线来表示,放在这个地方,我们把之前 Gradient 告诉我们要走的方向,跟前一步移动的方向加起来,得到往右走的方向,那再往右走 走到一个 Local Minima,照理说走到 Local Minima,一般 Gradient Descent 就无法向前走了,因为已经没有这个 Gradient 的方向,那走到 Saddle Point 也一样;没有 Gradient 的方向已经无法向前走了

但没有关系,如果有 Momentum 的话,你还是有办法继续走下去,因为 Momentum 不是只看 Gradient,Gradient就算是0,你还有前一步的方向,前一步的方向告诉我们向右走,我们就继续向右走,甚至你走到这种地方,Gradient 告诉你应该要往左走了,但是假设你前一步的影响力,比 Gradient 要大的话,你还是有可能继续往右走,甚至翻过一个小丘,搞不好就可以走到更好 Local Minima,这个就是 Momentum 有可能带来的好处



那这个就是今天想要跟大家说的内容,

Concluding Remarks

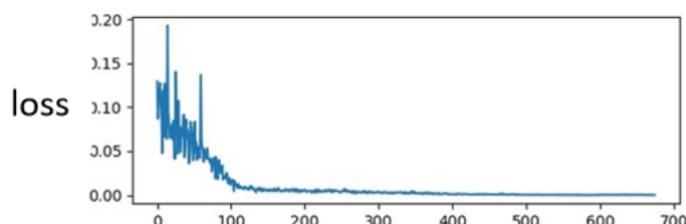
- Critical points have zero gradients.
- Critical points can be either saddle points or local minima.
 - Can be determined by the Hessian matrix.
 - Local minima may be rare.
 - It is possible to escape saddle points along the direction of eigenvectors of the Hessian matrix
- Smaller batch size and momentum help escape critical points.

Tips for training: Adaptive Learning Rate

critical point其实不一定是,你在训练一个Network的时候,会遇到的最大的障碍,今天要告诉大家的是一个叫做Adaptive Learning Rate的技术,我们要给每一个参数不同的learning rate

Training stuck ≠ Small Gradient

為什麼我说这个critical point不一定是我们训练过程中,最大的阻碍呢?



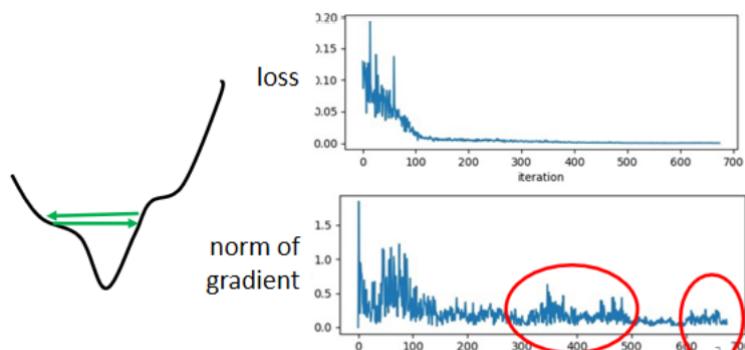
往往同学们,在训练一个network的时候,你会把它的loss记录下来,所以你会看到,你的loss原来很大,随著你参数不断的update,横轴代表参数update的次数,随著你参数不断的update,这个loss会越来越小,最后就卡住了,你的loss不再下降

那多数这个时候,大家就会猜说,那是是不是走到了critical point,因為gradient等於零的关係,所以我们没有办法再更新参数,但是真的是这样吗

当我们说 走到critical point的时候,意味著gradient非常的小,但是你有确认过,当你的loss不再下降的时候,gradient真的很小吗? 其实多数的同学可能,都没有确认过这件事,而事实上在这个例子裡面,在今天我show的这个例子裡面,当我们的loss不再下降的时候,gradient并没有真的变得很小

gradient是一个向量,下面是gradient的norm,即gradient这个向量的长度,随著参数更新的时候的变化,你会发现说虽然loss不再下降,但是这个gradient的norm,gradient的大小并没有真的变得很小

这样子的结果其实也不难猜想,也许你遇到的是这样子的状况



这个是我们的error surface,然后你现在的gradient,在error surface山谷的两个谷壁间,不断的来回的震荡

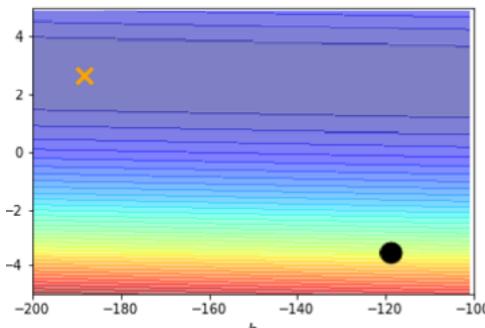
这个时候你的loss不会再下降,所以你会觉得它真的卡到了critical point,卡到了saddle point,卡到了local minima吗? 不是的,它的gradient仍然很大,只是loss不见得再减小了.

所以你要注意,当你今天训练一个network,train到后来发现,loss不再下降的时候,你不要随便说,我卡在local minima,我卡在saddle point,有时候根本两个都不是,你只是单纯的loss没有办法再下降

就是為什麼你在[作业2-2](#),会有一个作业叫大家,算一下gradient的norm,然后算一下说,你现在是卡在saddle point,还是critical point,因為多数的时候,当你说你训练卡住了,很少有人会去分析卡住的原因,為了强化你的印象,我们有一个作业,让你来分析一下,卡住的原因是什麼,

Training can be difficult even without critical points

如果今天critical point不是问题的话,為什麼我们的training会卡住呢,我这边举一个非常简单的例子,我这边有一个,非常简单的error surface

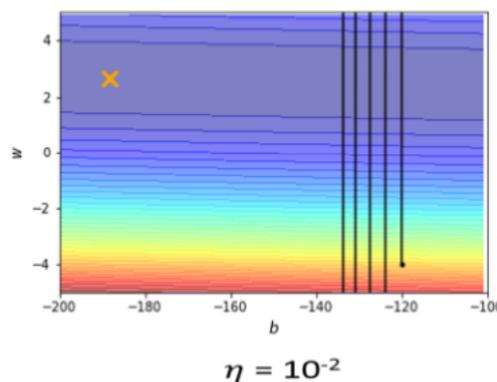


我们只有两个参数,这两个参数值不一样的时候,Loss的值不一样,我们就画出了一个error surface,这个error surface的最低点在黄色X这个地方,事实上,这个error surface是convex的形状(可以理解为凸的或者凹的, convex optimization常翻译为“凸优化”)

如果你不知道convex是什麼,没有關係,总之它是一个,它的这个等高线是椭圆形的,只是它在横轴的地方,它的gradient非常的小,它的坡度的变化非常的小,非常的平滑,所以这个椭圆的长轴非常的长,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

那现在我们要从黑点这个地方,这个地方当作初始的点, 然后来做gradient descend

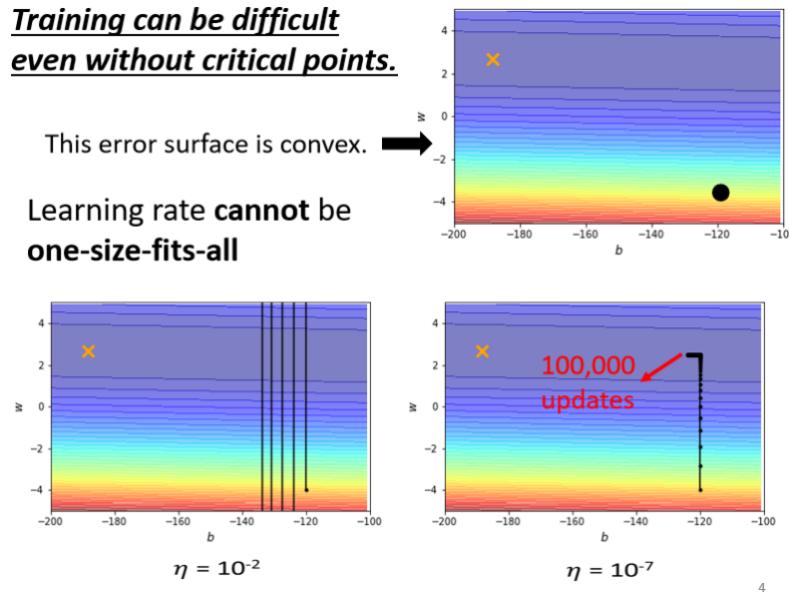
你可能觉得说,这个convex的error surface,做gradient descend,有什麼难的吗? 不就是一路滑下来,然后可能再走过去吗,应该是非常容易。你实际上自己试一下,你会发现说,就连这种convex的error surface,形状這麼简单的error surface,你用gradient descend,都不见得能把它做好,举例来说这个是我实际上,自己试了一下的结果



我learning rate设 10^{-2} 的时候,我的这个参数在峡谷的两端,我的参数在山壁的两端不断的震盪,我的loss掉不下去,但是gradient其实仍然是很大的

那你可能说,就是因為你learning rate设太大了阿,learning rate决定了我们update参数的时候步伐有多大,learning rate显然步伐太大,你没有办法慢慢地滑到山谷裡面只要把learning rate设小一点,不就可以解决这个问题了吗?

事实不然,因為我试著去,调整了这个learning rate,就会发现你光是要train这种convex的optimization的问题,你就觉得很痛苦,我就调这个learning rate,从 10^{-2} ,一直调到 10^{-7} ,调到 10^{-7} 以后,終於不再震盪了



終於从这个地方滑滑滑,滑到山谷底終於左转,但是你发现说,这个训练永远走不到终点,因為我的learning rate已经太小了,竖直往上这一段这个很斜的地方,因為这个坡度很陡,gradient的值很大,所以还能够前进一点,左拐以后这个地方坡度已经非常的平滑了,這麼小的learning rate,根本没有办法再让我们的训练前进

事实上在左拐这个地方,看到这边一大堆黑点,这边有十万个点,这个是张辽八百冲十万的那个十万,但是我都没有办法靠近,这个local minima的地方,所以显然就算是一个convex的error surface,你用gradient descend也很难train

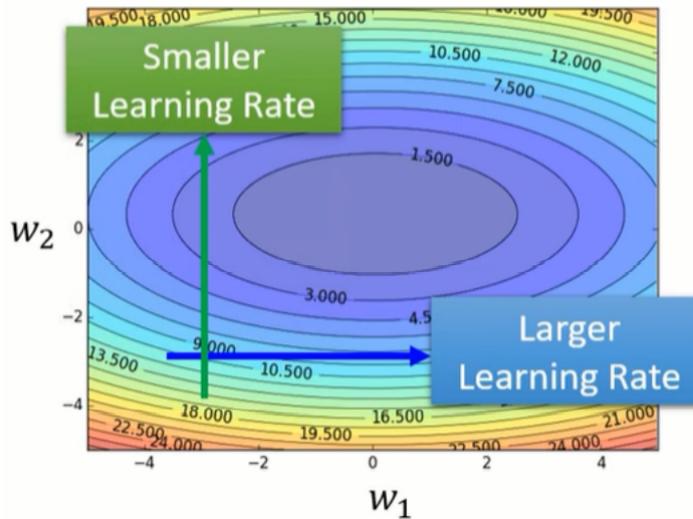
这个convex的optimization的问题,确实有别的方法可以解,但是你想看,如果今天是更复杂的error surface,你真的要train一个deep network的时候,gradient descend是你唯一可以仰赖的工具,但是gradient descend这个工具,连這麼简单的error surface都做不好,一室之不治 何以天下国家為,這麼简单的问题都做不好,那如果难的问题,它又怎麼有可能做好呢

所以我们需要更好的gradient descend的版本,在之前我们的gradient descend裡面,所有的参数都是设同样的learning rate,这显然是不够的,learning rate它应该要為,每一个参数客製化,所以接下来我们就是要讲,客製化的learning rate,怎麼做到这件事情

Different parameters needs different learning rate

那我们要怎麼客製化learning rate呢,我们不同的参数到底,需要什麼样的learning rate呢

从刚才的例子裡面,其实我们可以看到一个大原则,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非常的陡峭,坡度很大,那我们其实期待,learning rate可以设得小一点



那这个learning rate要如何自动的,根据这个gradient的大小做调整呢

我们要改一下,gradient descend原来的式子,我们只放某一个参数update的式子,我们之前在讲gradient descend,我们往往是讲所有参数update的式子,那这边為了等一下简化这个问题,我们只看一个参数,但是你完全可以把这个方法,推广到所有参数的状况

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

我们只看一个参数,这个参数叫做 θ_i^t ,这个 θ_i^t 在第t个iteration的值,减掉在第t个iteration这个参数i算出来的gradient g_i^t

$$g_i^t = \frac{\partial L}{\partial \theta_i} |_{\theta=\theta^t}$$

这个 g_i^t 代表在第t个iteration,也就是 θ 等於 θ^t 的时候,参数 θ 对loss的微分,我们把这个 θ^t 减掉learning rate,乘上 g_i^t 会更新learning rate到 θ^{t+1} ,这是我们原来的gradient descend,我们的learning rate是固定的

现在我们要有一个随著参数客製化的learning rate,我们把原来learning rate η 这一项呢,改写成 $\frac{\eta}{\sigma_i^t}$

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

这个 σ_i^t 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

所以当我们把我们的learning rate,从 η 改成 $\frac{\eta}{\sigma_i^t}$ 的时候,我们就要一个,parameter dependent的learning rate,接下来我们是要看说,这个parameter dependent的learning rate有什麼常见的计算方式

Root mean square

那这个 σ 有什麼样的方式,可以把它计算出来呢,一个常见的类型是算gradient的Root Mean Square

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

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

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

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

$$\vdots$$

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

6

现在参数要update的式子,我们从 θ_i^0 初始化参数减掉 g_i^0 ,乘上learning rate η 除以 σ_i^0 ,就得到 θ_i^1 ,

$$\theta_i^1 \leftarrow \theta_i^0 - \frac{\eta}{\sigma_i^0} g_i^0$$

- 这个 σ_i^0 在第一次update参数的时候,这个 σ_i^0 是 $(g_i^0)^2$ 开根号

$$\sigma_i^0 = \sqrt{(g_i^0)^2} = |g_i^0|$$

这个 g_i^0 就是我们的gradient,就是gradient的平方开根号,其实就是 g_i^0 的绝对值,所以你把 g_i^0 的绝对值代到 $\theta_i^1 \leftarrow \theta_i^0 - \frac{\eta}{\sigma_i^0} g_i^0$,这个式子中 g_i^0 跟这个根号底下的 g_i^0 ,它们的大小是一样的,所以式子中这一项只会有一个,要嘛是正一 要嘛是负一,就代表说我们第一次在update参数,从 θ_i^0 update到 θ_i^1 的时候,要嘛是加上 η 要嘛是减掉 η ,跟这个gradient的大小没有关系,是看你 η 设多少,这个是第一步的状况

- 重点是接下来怎麽处理,那 θ_i^1 它要一样,减掉gradient g_i^1 乘上 η 除以 σ_i^1 ,

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

现在在第二次update参数的时候,是要除以 σ_i^1 ,这个 σ_i^1 就是我们过去,所有计算出来的gradient,它的平方的平均再开根号

$$\sigma_i^1 = \sqrt{\frac{1}{2} [(g_i^0)^2 + (g_i^1)^2]}$$

我们到目前为止,在第一次update参数的时候,我们算出了 g_i^0 ,在第二次update参数的时候,我们算出了 g_i^1 ,所以这个 σ_i^1 就是 $(g_i^0)^2 + (g_i^1)^2$ 除以 $\frac{1}{2}$ 再开根号,这个就是Root Mean Square,我们算出这个 σ_i^1 以后,我们的learning rate就是 η 除以 σ_i^1 ,然后把 θ_i^1 减掉, η 除以 σ_i^1 乘以 g_i^1 得到 θ_i^2

$$\theta_i^2 \leftarrow \theta_i^1 - \frac{\eta}{\sigma_i^1} g_i^1$$

- 同样的操作就反覆继续下去,在 θ_i^2 的地方,你要减掉 η 除以 σ_i^2 乘以 g_i^2 ,

$$\theta_i^2 - \frac{\eta}{\sigma_i^2} g_i^2$$

那这个 σ 是什麼呢,这个 σ^2 就是过去,所有算出来的gradient,它的平方和的平均再开根号

$$\sigma_i^2 = \sqrt{\frac{1}{3} [(g_i^0)^2 + (g_i^1)^2 + (g_i^2)^2]}$$

所以你把 g_i^0 取平方, g_i^1 取平方, g_i^2 取平方,的平均再开根号,得到 σ_i^2 放在这个地方,然后update参数

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

- 所以这个process这个过程,就反覆继续下去,到第t次update参数的时候,其实这个是第t + 1次,第t + 1次update参数的时候,你的这个 σ_i^t 它就是过去所有的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,再平均 再开根号得到 σ_i^t ,

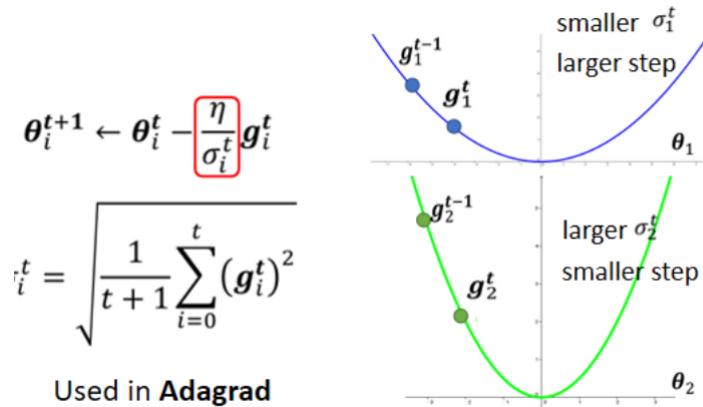
$$\sigma_i^t = \sqrt{\frac{1}{t+1} \sum_{i=0}^t (g_i^t)^2}$$

然后在把它除learning rate,然后用这一项当作是新的learning rate来update你的参数,

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

Adagrad

那这一招被用在一个叫做Adagrad的方法裡面,為什麼这一招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就放大呢?



你可以想像说,现在我们有两个参数:一个叫 θ_1 一个叫 θ_2 θ_1 坡度小 θ_2 坡度大

- θ_1 因为它坡度小,所以你在 θ_1 这个参数上面,算出来的gradient值都比较小
- 因为gradient算出来的值比较小,然后这个 σ 是gradient的平方和取平均再开根号

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

- 所以算出来的 σ 就小, σ 小 learning rate就大

$$\frac{\eta}{\sigma_i^t}$$

反过来说明 θ_2 , θ_2 是一个比较陡峭的参数,在 θ_2 这个方向上loss的变化比较大,所以算出来的gradient都比较大,,你的 σ 就比较大,你在update的时候 你的step,你的参数update的量就比较小

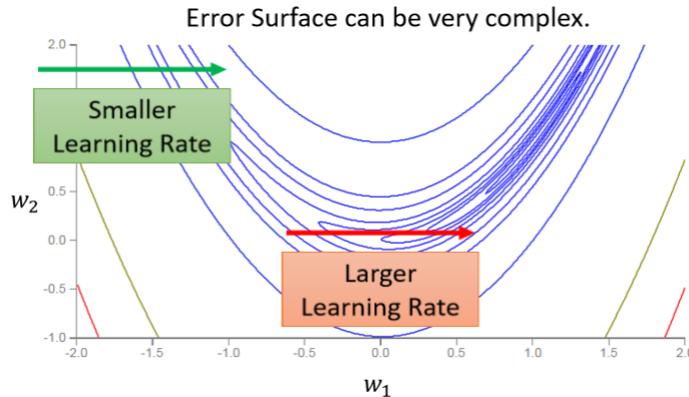
所以有了 σ 这一项以后,你就可以随著gradient的不同,每一个参数的gradient的不同,来自动的调整 learning rate的大小,那这个并不是,你今天会用的最终极的版本,

RMSProp

刚才那个版本,就算是同一个参数,它需要的learning rate,也会随著时间而改变,我们刚才的假设,好像是同一个参数,它的gradient的大小,就会固定是差不多的值,但事实上并不一定是这个样子的

举例来说我们来看,这个新月形的error surface

如果我们考虑横轴的话,考虑左右横的水平线的方向的话,你会发现说,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,平滑了起来就需要比较大的 learning rate,所以就算是同一个参数同一个方向,我们也期待说,learning rate是可以动态的调整的,于是就有了一个新的招数,这个招数叫做RMS Prop

RMS Prop这个方法有点传奇,它传奇的地方在於它找不到论文,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裡面,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那个影片的连结,这是个传奇的方法叫做RMS Prop

RMS Prop这个方法,它的第一步跟刚才讲的Root Mean Square,也就是那个Apagrad的方法,是一模一样的

$$\sigma_i^0 = \sqrt{(g_i^0)^2}$$

我们看第二步,一样要算出 σ_i^1 ,只是我们现在算出 σ_i^1 的方法跟刚才,算Root Mean Square的时候不一样,刚才在算Root Mean Square的时候,每一个gradient都有同等的重要性,但在RMS Prop裡面,它决定你可以自己调整,现在的这个gradient,你觉得它有多重要

$$\sigma_i^1 = \sqrt{\alpha(\sigma_i^0)^2 + (1 - \alpha)(g_i^1)^2}$$

所以在RMS Prop裡面,我们这个 σ_i^1 它是前一步算出来的 σ_i^0 ,裡面就是有 g_i^0 ,所以这个 σ_i^0 就代表了 g_i^0 的大小,所以它是 $(\sigma_i^0)^2$,乘上 α 加上 $(1 - \alpha)$,乘上现在我们刚算出来的,新鲜热腾腾的gradient就是 g_i^1

那这个 α 就像learning rate一样,这个你要自己调它,它是一个hyperparameter

- 如果我今天 α 设很小趋近於0,就代表我觉得 g_i^1 相较于之前所算出来的gradient而言,比较重要
- 我 α 设很大趋近於1,那就代表我觉得现在算出来的 g_i^1 比较不重要,之前算出来的gradient比较重要

所以同理在第三次update参数的时候,我们要算 σ_i^2 ,我们就把 σ_i^1 拿出来取平方再乘上 α ,那 σ_i^1 裡面有 g_i^1 跟 σ_i^0 , σ_i^0 裡面又有 g_i^0 ,所以你知道 σ_i^1 裡面它有 g_i^1 有 g_i^0 ,然后这个 g_i^1 跟 g_i^0 呢他们会被乘上 α ,然后再加上 $1 - \alpha$ 乘上这个 $(g_i^2)^2$

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

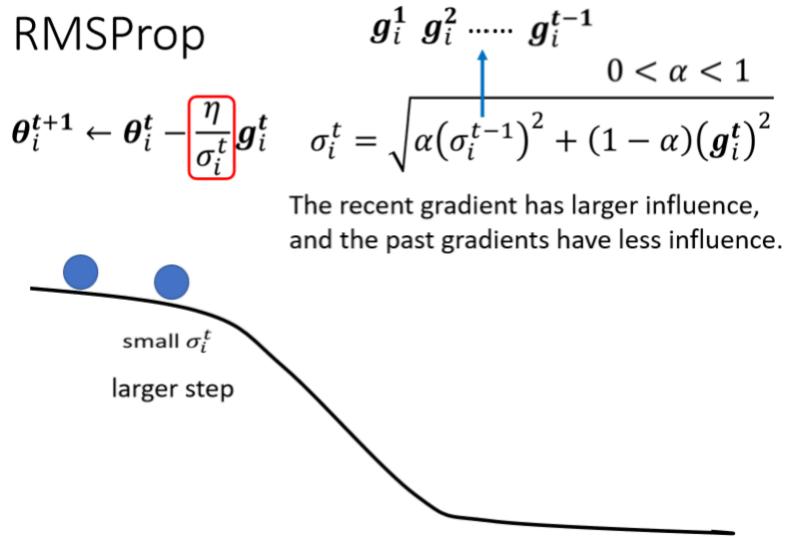
所以这个 α 就会决定说 g_i^2 ,它在整个 σ_i^2 裡面佔有多大的影响力

那同样的过程就反覆继续下去, σ_i^t 等於根号 α 乘上 $(\sigma_i^{t-1})^2$, 加上 $(1-\alpha)(g_i^t)^2$,

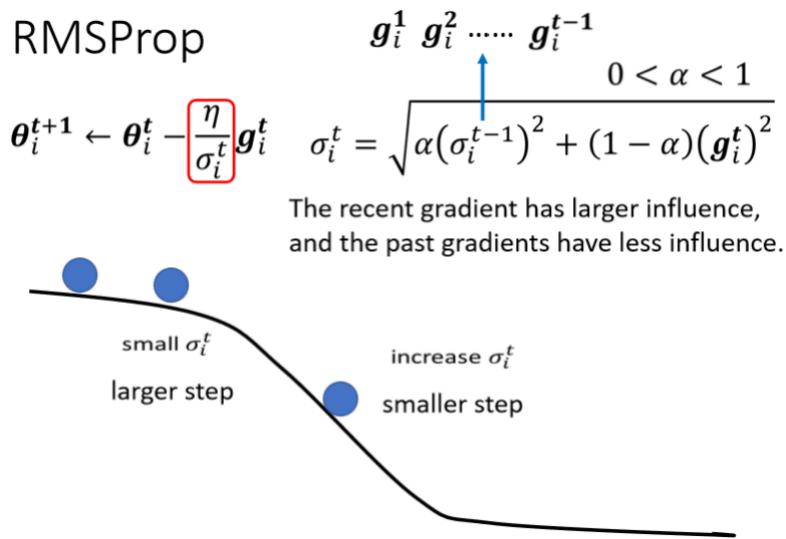
$$\sigma_i^t = \sqrt{\alpha(\sigma_i^{t-1})^2 + (1-\alpha)(g_i^t)^2}$$

你用 α 来决定现在刚算出来的 g_i^t , 它有多重要, 好那这个就是 RMSProp

那 RMSProp 我们刚刚讲过说, 透过 α 这一项你可以决定说, g_i^t 相较於之前存在, σ_i^{t-1} 裡面的 g_i^t 到 g_i^{t-1} 而言, 它的重要性有多大, 如果你用 RMS Prop 的话, 你就可以动态调整 σ 这一项, 我们现在假设从这个地方开始



这个黑线是我们的 error surface, 从这个地方开始你要 update 参数, 好你这个球就从这边走到这边, 那因为一路上都很平坦, 很平坦就代表说 g 算出来很小, 代表现在 update 参数的时候, 我们会走比较大的步伐



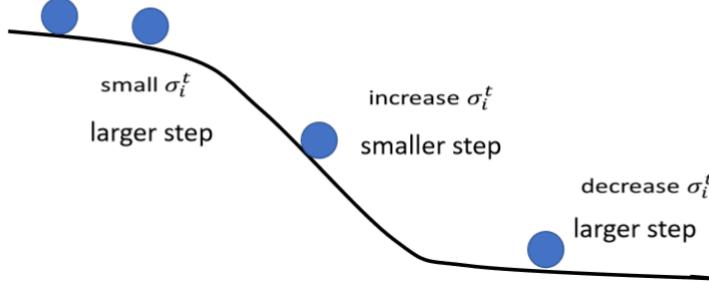
接下来继续滚, 滚到这边以后我们 gradient 变大了, 如果不是 RMS Prop, 原来的 Adagrad 的话它反应比较慢, 但如果你用 RMS Prop, 然后呢你把 α 设小一点, 你就是让新的, 刚看到的 gradient 影响比较大的话, 那你就就可以很快的让 σ 的值变大, 也可以很快的让你的步伐变小

你就可以踩一个煞车, 本来很平滑走到这个地方, 突然变得很陡, 那 RMS Prop 可以很快的踩一个煞车, 把 learning rate 变小, 如果你没有踩刹车的话, 你走到这裡这个地方, learning rate 太大了, 那 gradient 又很大, 两个很大的东西乘起来, 你可能就很快就飞出去了, 飞到很远的地方

RMSProp

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

The recent gradient has larger influence, and the past gradients have less influence.



如果继续走,又走到平滑的地方了,因为这个 σ_i^t 你可以调整 α ,让它比较看重於,最近算出来的gradient,所以你gradient一变小, σ 可能就反应很快,它的这个值就变小了,然后呢你走的步伐就变大了,这个就是RMS Prop,

Adam

那今天你最常用的,optimization的策略,有人又叫做optimizer,今天最常用的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) → for momentum
 $v_0 \leftarrow 0$ (Initialize 2nd moment vector) → 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RMS Prop (影响学习率, 即影响参数变化快慢) 加上Momentum (影响参数的方向), 那Adam的演算法跟原始的论文<https://arxiv.org/pdf/1412.6980.pdf>

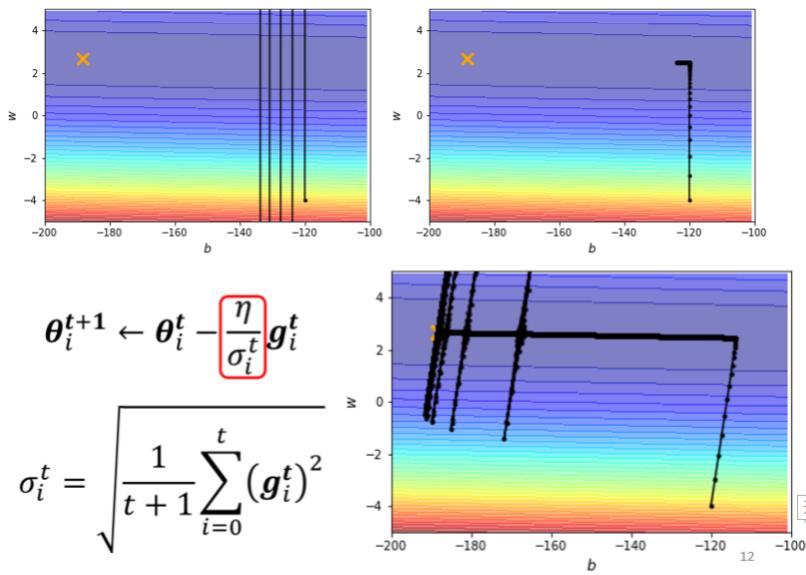
今天pytorch裡面,都帮你写得好好的了,所以这个你今天,不用担心这种optimization的问题,optimizer这个deep learning的套件,往往都帮你做好了,然后这个optimizer裡面,也有一些参数需要调,也有一些hyperparameter,需要人工决定,但是你往往用预设的,那一种参数就够好了,你自己调有时候会调到比较差的,往往你直接copy,这个pytorch裡面,Adam这个optimizer,然后预设的参数不要随便调,就可以得到不错的成绩了,關於Adam的细节,就留给大家自己研究

Learning Rate Scheduling

我们刚才讲说这个简单的error surface,我们都train不起来,现在我们来看一下,加上Adaptive Learning Rate以后,train不train得起来,

那这边是採用,最原始的Adagrad那个做法啦,就是把过去看过的,这个learning rate通通都,过去看过的gradient,通通都取平方再平均再开根号当作这个 σ ,做起来是这个样子的

Without Adaptive Learning Rate



这个走下来没有问题,然后接下来在左转的时候,这边也是update了十万次,之前update了十万次,只卡在左转这个地方

那现在有Adagrad以后,你可以再继续走下去,走到非常接近终点的位置,因為当你走到这个地方的时候,你因為这个左右的方向的,这个gradient很小,所以learning rate会自动调整,左右这个方向的,learning rate会自动变大,所以你这个步伐就可以变大,就可以不断的前进

接下来的问题就是,為什麼快走到终点的时候突然爆炸了呢

你想想看 我们在做这个 σ 的时候,我们是把过去所有看到的gradient,都拿来作平均

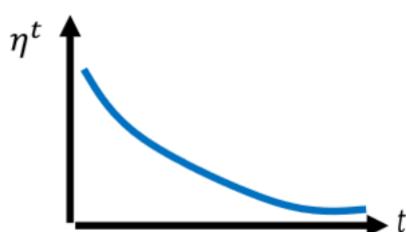
- 所以这个纵轴的方向,在这个初始的这个地方,感觉gradient很大
- 可是这边走了很长一段路以后,这个纵轴的方向,gradient算出来都很小,所以纵轴这个方向,这个y轴的方向就累积了很小的 σ
- 因為我们在这个y轴的方向,看到很多很小的gradient,所以我们就累积了很小的 σ ,累积到一个地步以后,这个step就变很大,然后就爆走就喷出去了
- 喷出去以后没关係,有办法修正回来,因為喷出去以后,就走到了这个gradient比较大的地方,走到gradient比较大的地方以后,这个 σ 又慢慢的变大, σ 慢慢变大以后,这个参数update的距离,Update的步伐大小就慢慢的变小

你就发现说走著走著,突然往左右喷了一下,但是这个喷了一下不会永远就是震盪,不会做简谐运动停不下来,这个力道慢慢变小,有摩擦力 让它慢慢地慢慢地,又回到中间这个峡谷来,然后但是又累计一段时间以后又会喷,然后又慢慢地回来 怎麼办呢,有一个方法也许可以解决这个问题,这个叫做learning rate的 scheduling

什麼是learning rate的scheduling呢

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.

我们刚才这边还有一项 η ,这个 η 是一个固定的值,learning rate scheduling的意思就是说,我们不要把 η 当一个常数,我们把它跟时间有关

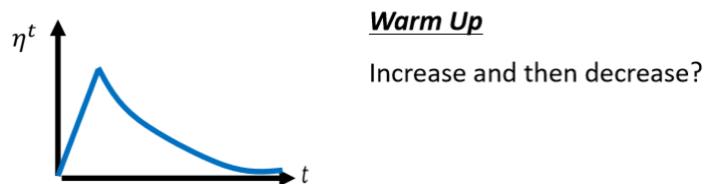
Learning Rate Decay

最常见的策略叫做Learning Rate Decay,也就是说,随著时间的不断地进行,随著参数不断的update,我们这个 η 让它越来越小

那这个也就合理了,因为一开始我们距离终点很远,随著参数不断update,我们距离终点越来越近,所以我们把learning rate减小,让我们参数的更新踩了一个煞车,让我们参数的更新能够慢慢地慢下来,所以刚才那个状况,如果加上Learning Rate Decay有办法解决

刚才那个状况,如果加上Learning Rate Decay的话,我们就可以很平顺的走到终点,因为在这个地方,这个 η 已经变得非常的小了,虽然说它本来想要左右乱喷,但是因为乘上这个非常小的 η ,就停下来了,就可以慢慢地走到终点,那除了Learning Rate Decay以外,还有另外一个经典,常用的Learning Rate Scheduling的方式,叫做Warm Up

Warm up



Warm Up这个方法,听起来有点匪夷所思,这Warm Up的方法是让learning rate,要先变大后变小,你会问说变大要变到多大呢,变大速度要多快呢,小速度要多快呢,这个也是hyperparameter,你要自己用手调的,但是大方向的大策略就是,learning rate要先变大后变小,那这个方法听起来很神奇,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裡面

这个warm up,最近因为在训练BERT的时候,往往需要用到Warm Up,所以又被大家常常拿出来讲,但它并不是有BERT以后,才有Warm Up的,Warm Up这东西远古时代就有了,举例来说,Residual Network裡面是有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>

这边是放了Residual network,放在arXiv上面的文章连结啦,今天这种有关machine learning的,文章往往在投conference之前,投国际会议之前,就先放到一个叫做arXiv的网站上,把它公开来让全世界的人都可以看

你其实看这个arXiv的网址,你就可以知道,这篇文章是什麼时候放到网路上的,怎麽看呢 arXiv的前四个数字,这15代表年份,代表说residual network这篇文章,是2015年放到arXiv上面的,后两个数字代表月份,所以它是15年的12月,15年的年底放在arXiv上面的

所以五六年前的文章,在deep learning变化,这麼快速的领域裡面,五六年前就是上古时代,那在上古时代,这个Residual Network裡面,就已经记载了Warm Up的这件事情,它说我们用learning rate 0.01,取Warm Up,先用learning rate 0.01,再把learning rate改成0.1

用过去我们通常最常见的train,Learning Rate Scheduling的方法,就是让learning rate越来越小,但是Residual Network,这边特别註明它反其道而行,一开始要设0.01 接下来设0.1,还特别加一个註解说,一开始就用0.1反而就train不好,不知道為什麼 也没解释,反正就是train不好,需要Warm Up这个黑科技。

而在哪个黑科技,在知名的Transformer裡面(这门课也会讲到),也用一个式子提了它

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:

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

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 $warmup_steps = 4000$.

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

它这边有一个式子说,它的learning rate遵守这一个,神奇的function来设定,它的learning rate,这个神奇的function,乍看之下会觉得 哇 在写什麼,不知道在写些什麼

这个东西你实际上,把这个function画出来,你实际上把equation画出来的话,就会发现它就是Warm Up,learning rate会先增加,然后接下来再递减

所以你发现说Warm Up这个技术,在很多知名的network裡面都有,被当作一个黑科技,就论文裡面不解释说,為什麼要用这个;但就偷偷在一个小地方,你没有注意到的小地方告诉你说,这个network要用这种黑科技,才能够把它训练起来

那為什麼需要warm Up呢,这个仍然是今天,一个可以研究的问题啦

这边有一个可能的解释是说,你想看当我们在用Adam RMS Prop,或Adagrad的时候,我们会需要计算 σ ,它是一个统计的结果, σ 告诉我们,某一个方向它到底有多陡,或者是多平滑,那这个统计的结果,要看得够多笔数据以后,这个统计才精準,所以一开始我们的统计是不精準的

一开始我们的 σ 是不精準的,所以我们一开始不要让我们的参数,走离初始的地方太远,先让它在初始的地方呢,做一些像是探索这样,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,先收集有关 σ 的统计数据,等 σ 统计得比较精準以后,在让learning rate呢慢慢地爬升

所以这是一个解释,為什麼我们需要warm up的可能性,那如果你想要学更多,有关warm up的东西的话,你其实可以看一篇paper,它是Adam的进阶版叫做RAdam,裡面对warm up这件事情,有更多的理解

那有关optimization的部分,其实我们就讲到这边啦,

Summary of Optimization

所以我们从最原始的gradient descent,进化到这一个版本

(Vanilla) Gradient Descent

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

Various Improvements

$$\theta_i^{t+1} \leftarrow \theta_i^t - \frac{\eta^t}{\sigma_i^t} m_i^t \rightarrow \text{Momentum: weighted sum of the previous gradients}$$

这个版本裡面

- 我们有Momentum,也就是说我们现在,不是完全顺著gradient的方向,现在不是完全顺著这一个时间点,算出来的gradient的方向,来update参数,而是把过去,所有算出来gradient的方向,做一个加总当作update的方向,这个是momentum
- 接下来应该要update多大的步伐呢,我们要除掉,gradient的Root Mean Square

(Vanilla) Gradient Descent

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

Various Improvements

$$\theta_i^{t+1} \leftarrow \theta_i^t - \frac{\eta^t}{\sigma_i^t} m_i^t \rightarrow \begin{array}{l} \text{Momentum: weighted sum of the} \\ \text{previous gradients} \end{array}$$

Consider direction

root mean square of the gradients

only magnitude

17

那讲到这边可能有同学会觉得很困惑,这一个momentum是考虑,过去所有的gradient,这个 σ 也是考虑过去所有的gradient,一个放在分子一个放在分母,都考虑过去所有的gradient,不就是正好抵消了吗,

但是其实这个Momentum跟这个 σ ,它们在使用过去所有gradient的方式是不一样的,Momentum是直接把所有的gradient通通都加起来,所以它有考虑方向,它有考虑gradient的正负号,它有考虑gradient是往左走还是往右走

但是这个Root Mean Square,它就不考虑gradient的方向了,它只考虑gradient的大小,记不得我们在算 σ 的时候,我们都要取平方项,我们都要把gradient取一个平方项,我们是把平方的结果加起来,所以我们只考虑gradient的大小,不考虑它的方向,所以Momentum跟这个 σ ,算出来的结果并不会互相抵消掉

- 那最后我们还会加上,一个learning rate的scheduling,

(Vanilla) Gradient Descent

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

Various Improvements

$$\theta_i^{t+1} \leftarrow \theta_i^t - \frac{\eta^t}{\sigma_i^t} m_i^t \rightarrow \begin{array}{l} \text{Learning rate scheduling} \\ \text{Momentum: weighted sum of the} \\ \text{previous gradients} \end{array}$$

Consider direction

root mean square of the gradients

only magnitude

17

那这个是今天optimization的,完整的版本了,这种Optimizer,除了Adam以外,Adam可能是今天最常用的,但除了Adam以外,还有各式各样的变形,但其实各式各样的变形都不脱,就是要嘛不同的方法算M,要嘛不同的方法算 σ ,要嘛不同的,Learning Rate Scheduling的方式

那如果你想要知道更多,跟optimization有关的事情的话,那有之前助教的录影,给大家参考到这裡,影片蛮长的大概两个小时,所以你可以想见说,有关Optimizer的东西,其实是还有蛮多东西可以讲的,所以时间的关係我们就不讲下去

To Learn More



<https://youtu.be/4pUmZ8hXIHM>
(in Mandarin)

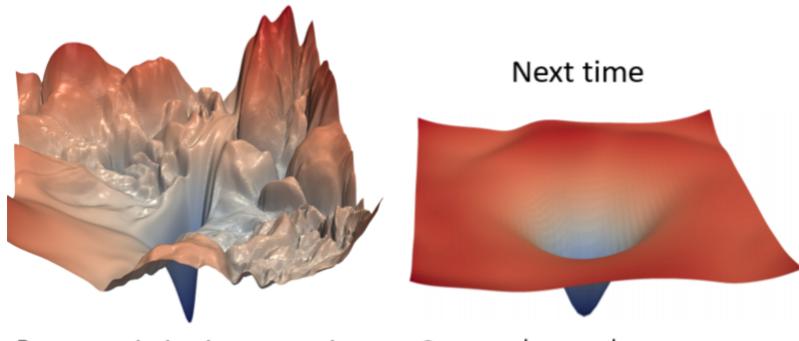


<https://youtu.be/e03YKGHxNl8>
(in Mandarin)

到目前为止呢 我们讲的是什麼,我们讲的是,当我们的error surface非常的崎岖,就像这个例子一样非常的崎岖的时候

Next Time

Source of image: <https://arxiv.org/abs/1712.09913>



Better optimization strategies:
If the mountain won't move,
build a road around it.

Can we change the error
surface?
Directly move the mountain!

我们需要一些比较好的方法,来做optimization,前面有一座山挡著,我们希望可以绕过那座山,山不转路转的意思这样,你知道这个gradient,这奇怪的error surface,会让人觉得很痛苦

那就要用神罗天征,把这个炸平这样子,所以接下来我们会讲的技巧,就是有没有可能,直接把这个error surface移平,我们改Network裡面的什麼东西,改Network的架构activation function,或者是其它的东西,直接移平error surface,让它变得比较好train,也就是山挡在前面,就把山直接剷平的意思

Batch Normalization

本篇是一个很快地介绍,Batch Normalization 这个技术

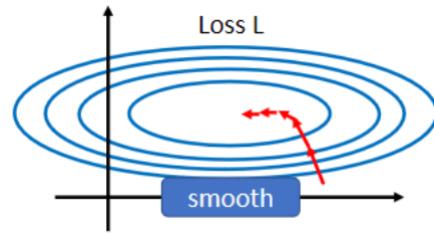
Changing Landscape

之前才讲过说,我们能不能够直接改error surface 的 landscape,我们觉得说 error surface 如果很崎岖的时候,它比较难 train,那我们能不能够直接把山剷平,让它变得比较好 train 呢?

Batch Normalization 就是其中一个,把山剷平的想法

我们一开始就跟大家讲说,不要小看 optimization 这个问题,有时候就算你的 error surface 是 convex的,它就是一个碗的形状,都不见得很好 train

假设你的两个参数啊,它们对 Loss 的斜率差别非常大,在 w_1 这个方向上面,你的斜率变化很小,在 w_2 这个方向上面斜率变化很大

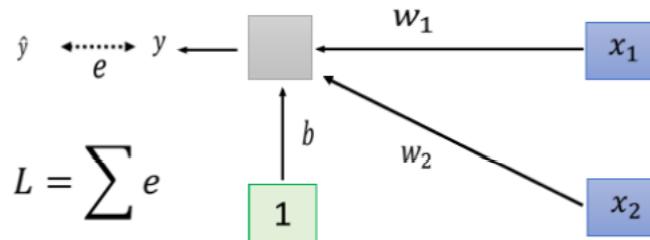


如果是固定的 learning rate,你可能很难得到好的结果,所以我们才说你需要adaptive 的 learning rate、Adam 等等比较进阶的 optimization 的方法,才能够得到好的结果

现在我们要从另外一个方向想,直接把难做的 error surface 把它改掉,看能不能够改得好做一点

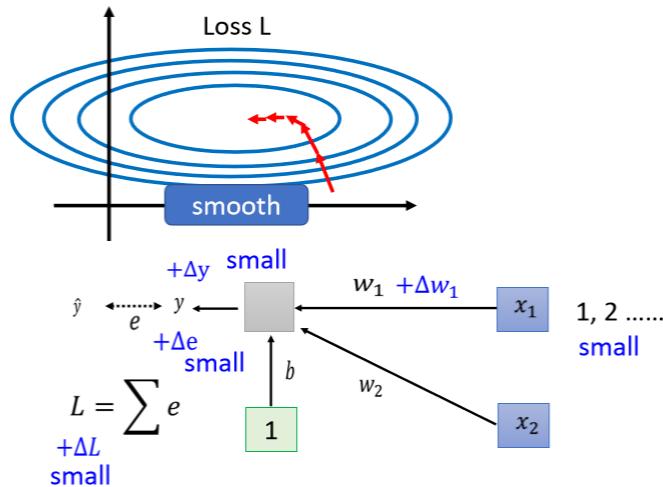
在做这件事之前,也许我们第一个要问的问题就是,有这一种状况, w_1 跟 w_2 它们的斜率差很多的这种状况,到底是从什麼地方来的

假设我现在有一个非常非常非常简单的 model,它的输入是 x_1 跟 x_2 ,它对应的参数就是 w_1 跟 w_2 ,它是一个 linear 的 model,没有 activation function



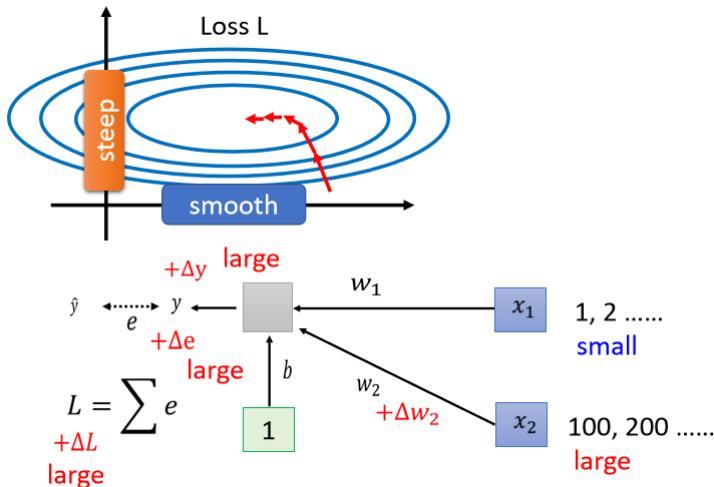
w_1 乘 x_1 , w_2 乘 x_2 加上 b 以后就得到 y ,然后会计算 y 跟 \hat{y} 之间的差距当做 e ,把所有 training data e 加起来就是你的 Loss, 然后去 minimize 你的 Loss

那什麼样的状况我们会產生像上面这样子,比较不好 train 的 error surface 呢?



当我们对 w_1 有一个小小的变化,比如说加上 Δw_1 的时候,那这个 L 也会有一个改变,那这个 w_1 呢,是透过 w_1 改变的时候,你就改变了 y , y 改变的时候你就改变了 e ,然后接下来就改变了 L , 那什麼时候 w_1 的改变会对 L 的影响很小呢,也就是它在 error surface 上的斜率会很小呢? 一个可能性是当你的 input 很小的时候,假设 x_1 的值在不同的 training example 裡面,它的值都很小,那因為 x_1 是直接乘上 w_1 , 如果 x_1 的值都很小, w_1 有一个变化的时候,它得到的,它对 y 的影响也是小的,对 e 的影响也是小的,它对 L 的影响就会是小的

反之呢,如果今天是 x_2 的话,那假设 x_2 的值都很大,当你的 w_2 有一个小小的变化的时候,虽然 w_2 这个变化可能很小,但是因为它乘上了 x_2, x_2 的值很大,那 y 的变化就很大,那 e 的变化就很大,那 L 的变化就会很大,就会导致我们在 w 这个方向上,做变化的时候,我们把 w 改变一点点,那我们的 error surface 就会有很大的变化



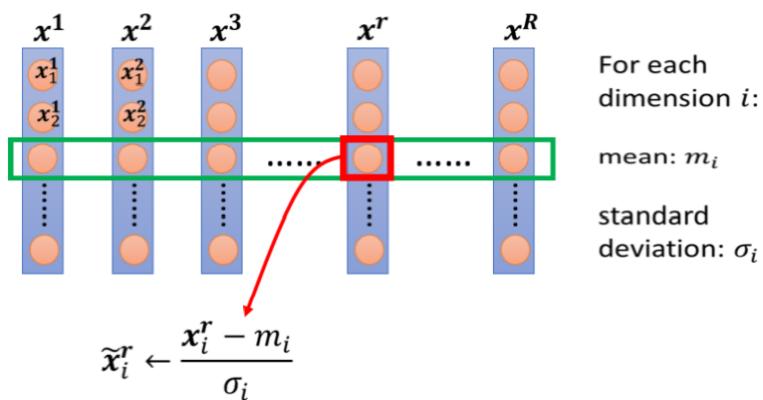
所以你发现说,既然在这个 linear 的 model 裡面,当我们 input 的 feature,每一个 dimension 的值,它的 scale 差距很大的时候,我们就可能產生像这样子的 error surface,就可能產生不同方向,斜率非常不同,坡度非常不同的 error surface

所以怎麽办呢,我们有没有可能给 feature 裡面不同的 dimension,让它有同样的数值的范围

如果我们可以给不同的 dimension,同样的数值范围的话,那我们可能就可以製造比较好的 error surface,让 training 变得比较容易一点, 其实有很多不同的方法,这些不同的方法,往往就合起来统称为 Feature Normalization

Feature Normalization

以下所讲的方法只是 Feature Normalization 的一种可能性,它并不是 Feature Normalization 的全部,假设 x^1 到 x^R ,是我们所有的训练资料的 feature vector, 不同的 x 代表不同的资料



我们把所有训练资料的 feature vector,统统都集合起来,那每一个 vector, x_1 裡面就 x_1^1 代表 x_1 的第一个 element, x_1^2 , 就代表 x_2 的第一个 element,以此类推

那我们把不同笔资料即不同 feature vector,同一个 dimension 裡面的数值,把它取出来,然后去计算某一个 dimension 的 mean, 它的 mean 呢就是 m_i , 我们计算第 i 个 dimension 的 standard deviation, 我们用 σ_i 来表示它

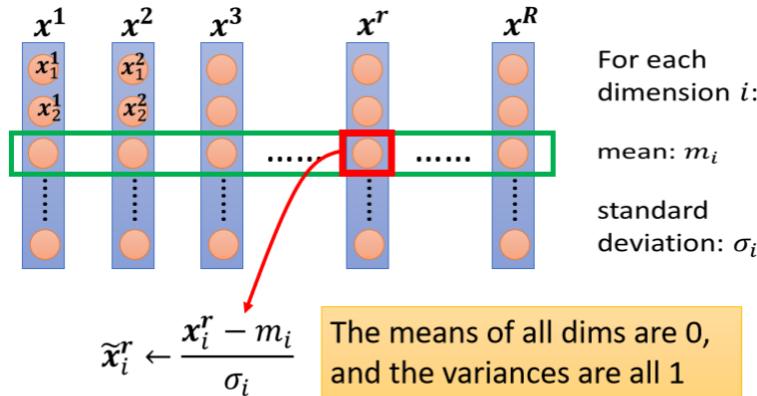
那接下来我们就可以做一种 normalization, 那这种 normalization 其实叫做標準化, 其实叫 standardization, 不过我们这边呢, 就等一下都统称 normalization 就好了

$$\tilde{x}_i^r \leftarrow \frac{x_i^r - m_i}{\sigma_i}$$

我们就是把这边的某一个数值 x , 减掉这一个 dimension 算出来的 mean, 再除掉这个 dimension, 算出来的 standard deviation, 得到新的数值叫做 \tilde{x}

然后得到新的数值以后, 再把新的数值把它塞回去, 以下都用这个 tilde 来代表有被 normalize 后的数值

那做完 normalize 以后有什么好处呢?



In general, feature normalization makes gradient descent converge faster.

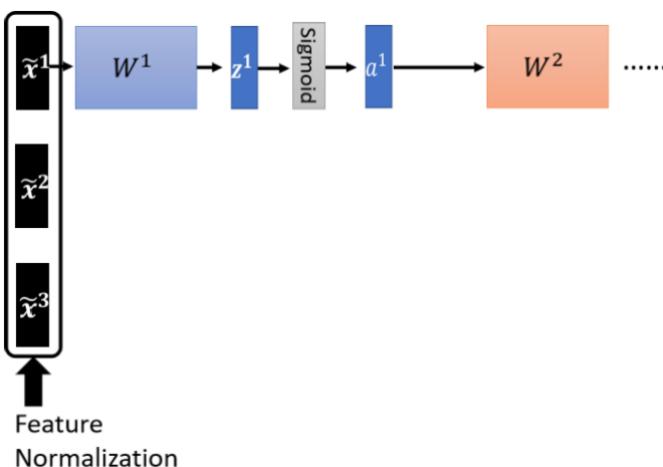
4

- 做完 normalize 以后啊, 这个 dimension 上面的数值就会平均是 0, 然后它的 variance 就会是 1, 所以这一排数值的分布就都会在 0 上下
- 对每一个 dimension 都做一样的 normalization, 就会发现所有 feature 不同 dimension 的数值都在 0 上下, 那你可能就可以製造一个比较好的 error surface

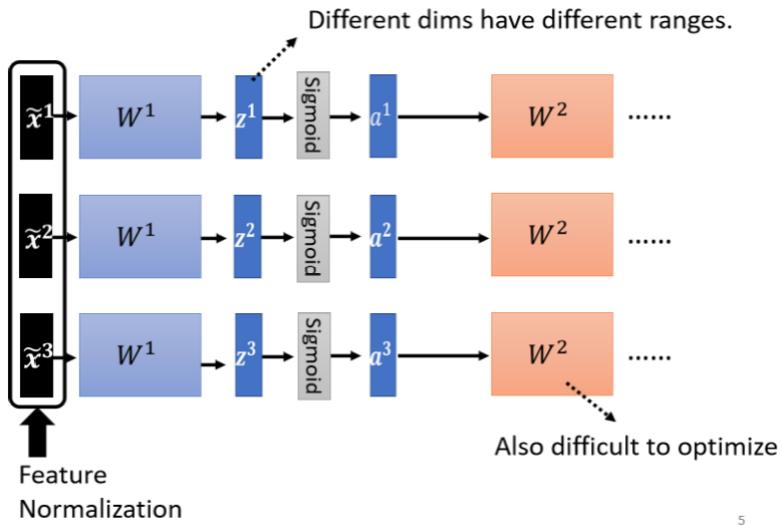
所以像这样子 Feature Normalization 的方式, 往往对你的 training 有帮助, 它可以让你在做 gradient descent 的时候, 这个 gradient descent, 它的 Loss 收敛更快一点, 可以让你的 gradient descent, 它的训练更顺利一点, 这个是 Feature Normalization

Considering Deep Learning when Feature Normalization

\tilde{x} 代表 normalize 的 feature, 把它丢到 deep network 裡面, 去做接下来的计算和训练, 所以把 x_1 tilde 通过第一个 layer 得到 z^1 , 那你有可能通过 activation function, 不管是选 Sigmoid 或者 ReLU 都可以, 然后再得到 a^1 , 然后再通过下一层等等, 那就看你有几层 network 你就做多少的运算



所以每一个 x 都做类似的事情, 但是如果我们进一步来想的话, 对 w_2 来说



5

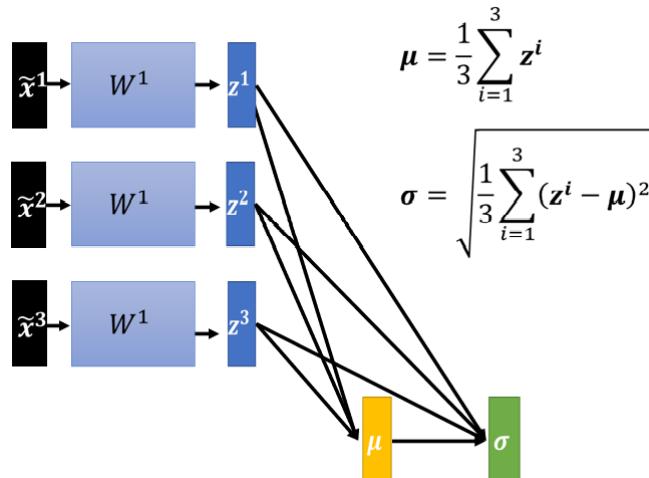
这边的 a^1 a^3 这边的 z^1 z^3 , 其实也是另外一种 input, 如果这边 \tilde{x} , 虽然它已经做 normalize 了, 但是通过 w_1 以后它就没有做 normalize, 如果 \tilde{x} 通过 w_1 得到是 z^1 , 而 z^1 不同的 dimension 间, 它的数值的分布仍然有很大的差异的话, 那我们要 train w_2 第二层的参数, 会不会也有困难呢

对 w_2 来说, 这边的 a 或这边的 z 其实也是一种 feature, 我们应该要对这些 feature 也做 normalization

那如果你选择的是 Sigmoid, 那可能比较推荐对 z 做 Feature Normalization, 因为 Sigmoid 是一个 s 的形状, 那它在 0 附近斜率比较大, 所以如果你对 z 做 Feature Normalization, 把所有的值都挪到 0 附近, 那你到时候算 gradient 的时候, 算出来的值会比较大

那不过因为你看得是用 sigmoid, 所以你也不一定要把 Feature Normalization 放在 z 这个地方, 如果是选别的, 也许你选 a 也会有好的结果, 也说不定, **In general 而言, 这个 normalization, 要放在 activation function 之前, 或之后都是可以的, 在实作上, 可能没有太大的差别**, 好, 那我们这边呢, 就是对 z 呢, 做一下 Feature Normalization

那怎么对 z 做 Feature Normalization 呢



那你就把 z , 想成是另外一种 feature, 我们这边有 z^1 z^2 z^3 , 我们就把 z^1 z^2 z^3 拿出来

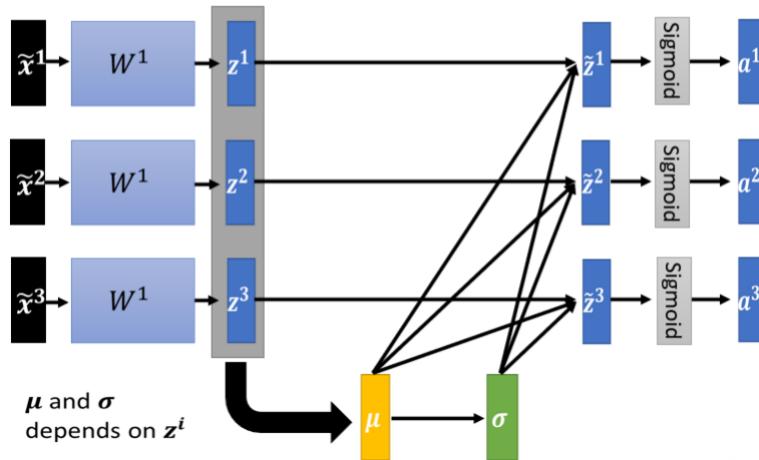
- 算一下它的 mean, 这边的 μ 是一个 vector, 我们就把 z^1 z^2 z^3 , 这三个 vector 呢, 把它平均起来, 得到 μ 这个 vector
- 那我们也算一个 standard deviation, 这个 standard deviation 呢, 这边这个成 σ , 它也代表了一个 vector, 那这个 vector 怎么算出来呢, 你就把 z^i 减掉 μ , 然后取平方, 这边的平方, 这个 notation 有点 abuse 啊, 这边的平方就是指, 对每一个 element 都去做平方, 然后再开根号, 这边开根号指的是对每一个 element, 向量里面的每一个 element, 都去做开根号, 得到 σ , 反正你知道我的意思就好

把这三个 vector, 裡面的每一个 dimension, 都去把它的 μ 算出来, 把它的 σ 算出来, 好, 我这边呢, 就不把那些箭头呢画出来了, 从 z^1 z^2 z^3 , 算出 μ , 算出 σ

接下来就把这边的每一个 z , 都去减掉 μ 除以 σ , 你把 z^i 减掉 μ , 除以 σ , 就得到 z^i 的 tilde

Considering Deep Learning

$$\tilde{z}^i = \frac{z^i - \mu}{\sigma}$$

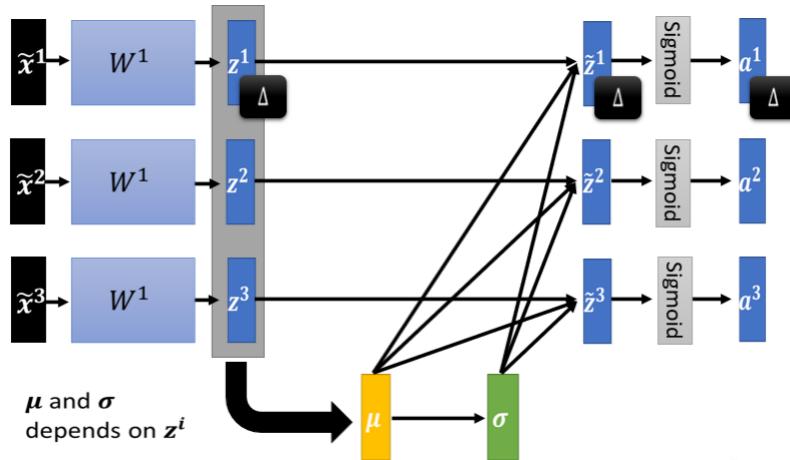


那这边的 μ 跟 σ , 它都是向量, 所以这边这个除的意思是 element wise 的相除, 就是 z^i 减 μ , 它是一个向量, 所以分子的地方是一个向量, 分母的地方也是一个向量, 把这个两个向量, 它们对应的 element 的值相除, 是我这边这个除号的意思, 这边得到 Z 的 tilde

所以我们就是把 z^1 减 μ 除以 σ , 得到 z^1 tilde, 同理 z^2 减 μ 除以 σ , 得到 z^2 tilde, z^3 减 μ 除以 σ , 得到 z^3 tilde, 那就把这个 z^1, z^2, z^3 , 做 Feature Normalization, 变成 z^1 tilde, z^2 tilde 跟 z^3 的 tilde

接下来就看你爱做什麼 就做什麼啦, 通过 activation function, 得到其他 vector, 然后再通过, 再去通过其他 layer 等等, 这样就可以了, 这样你就等於对 z^1, z^2, z^3 , 做了 Feature Normalization, 变成 $\tilde{z}^1, \tilde{z}^2, \tilde{z}^3$

在这边有一件有趣的事情, 这边的 μ 跟 σ , 它们其实都是根据 z^1, z^2, z^3 算出来的

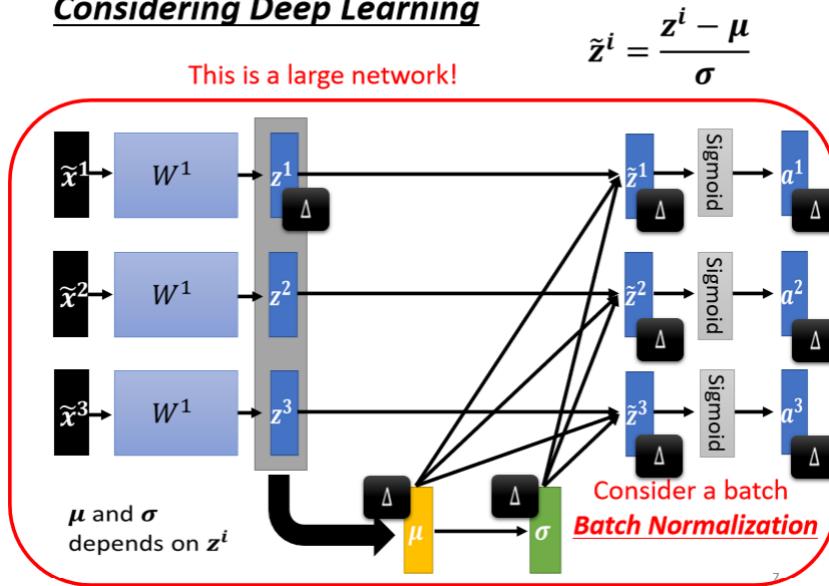


所以这边 z^1 啊, 它本来, 如果我们没有做 Feature Normalization 的时候, 你改变了 z^1 的值, 你会改变这边 a 的值, 但是现在啊, 当你改变 z^1 的值的时候, μ 跟 σ 也会跟著改变, μ 跟 σ 改变以后, z^2 的值 a^2 的值, z^3 的值 a^3 的值, 也会跟著改变

所以之前, 我们每一个 $\tilde{x}_1, \tilde{x}_2, \tilde{x}_3$, 它是独立分开处理的, 但是我们在做 Feature Normalization 以后, 这三个 example, 它们变得彼此关联了

我们这边 z^1 只要有改变, 接下来 z^2, a^2, z^3, a^3 , 也都会跟著改变, 所以这边啊, 其实你要把, 当你有做 Feature Normalization 的时候, 你要把这一整个 process, 就是有收集一堆 feature, 把这堆 feature 算出 μ 跟 σ 这件事情, 当做是 network 的一部分

Considering Deep Learning



也就是说,你现在有一个比较大的 network

- 你之前的 network,都只吃一个 input,得到一个 output
- 现在你有一个比较大的 network,这个大的 network,它是吃一堆 input,用这堆 input 在这个 network 裡面,要算出 μ 跟 σ ,然后接下来產生一堆 output

Batch Normalization

那这边就会有一个问题了,因為你的训练资料裡面的 data 非常多,现在一个 data set,benchmark corpus 都上百万笔资料, GPU 的 memory,根本没有办法,把它整个 data set 的 data 都 load 进去。

在实作的时候,你不会让这一个 network 考虑整个 training data 裡面的所有 example,你只会考虑一个 batch 裡面的 example,举例来说,你 batch 设 64,那你这个巨大的 network,就是把 64 笔 data 读进去,算这 64 笔 data 的 μ ,算这 64 笔 data 的 σ ,对这 64 笔 data 都去做 normalization

因為我们在实作的时候,我们只对一个 batch 裡面的 data,做 normalization,所以这招叫做 Batch Normalization

那这个 Batch Normalization,显然有一个问题 就是,你一定要有一个够大的 batch,你才算得出 μ 跟 σ ,假设你今天,你 batch size 设 1,那你就没有什麼 μ 或 σ 可以算

所以这个 Batch Normalization,是适用於 batch size 比较大的时候,因為 batch size 如果比较大,也许这个 batch size 裡面的 data,就足以表示,整个 corpus 的分布,那这个时候你就可以,把这个本来要对整个 corpus,做 Feature Normalization 这件事情,改成只在一个 batch,做 Feature Normalization,作為 approximation,

在做 Batch Normalization 的时候,往往还会有这样的设计你算出这个 \tilde{z} 以后

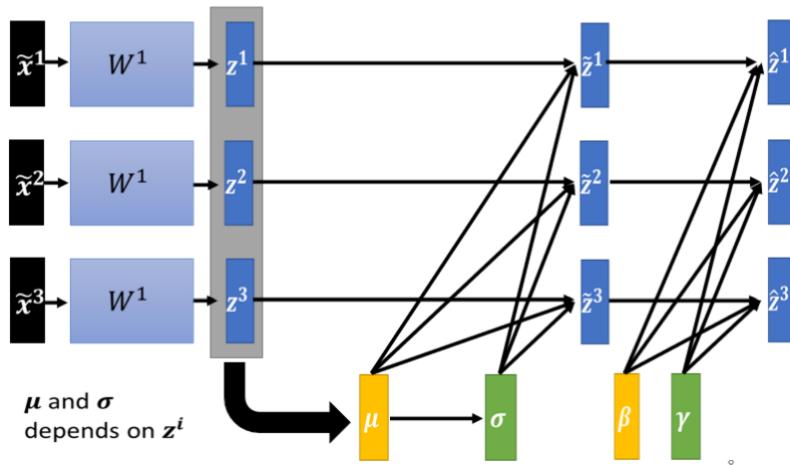
- 接下来你会把这个 \tilde{z} ,再乘上另外一个向量叫做 γ ,这个 γ 也是一个向量,所以你就是把 \tilde{z} 跟 γ 做 element wise 的相乘,把 z 这个向量裡面的 element,跟 γ 这个向量裡面的 element,两两做相乘
- 再加上 β 这个向量,得到 \hat{z}

而 β 跟 γ ,你要把它想成是 network 的参数,它是另外再被 learn 出来的,

Batch normalization

$$\tilde{z}^i = \frac{z^i - \mu}{\sigma}$$

$$\hat{z}^i = \gamma \odot \tilde{z}^i + \beta$$



那為什麼要加上 β 跟 γ 呢

有人可能会觉得说,如果我们做 normalization 以后,那这边的 \tilde{z} ,它的平均就一定是 0,那也许,今天如果平均是 0 的话,就是给那 network 一些限制,那也许这个限制会带来什麼负面的影响,所以我们把 β 跟 γ 加回去

然后让 network 呢,现在它的 hidden layer 的 output 平均不是 0 的话,他就自己去 learn 这个 β 跟 γ ,来调整一下输出的分布,来调整这个 \hat{z} 的分布

但讲到这边又会有人问说,刚才不是说做 Batch Normalization 就是,為了要让每一个不同的 dimension,它的 range 都是一样吗,现在如果加去乘上 γ ,再加上 β ,把 γ 跟 β 加进去,这样不会不同 dimension 的分布,它的 range 又都不一样了吗

有可能,但是你实际上在训练的时候,这个 γ 跟 β 的初始值啊

- 你会把这个 γ 的初始值就都设為 1,所以 γ 是一个裡面的值,一开始其实是一个裡面的值,全部都是 1 的向量
- 那 β 是一个裡面的值,全部都是 0 的向量,所以 γ 是一个 one vector,都是 1 的向量, β 是一个 zero vector,裡面的值都是 0 的向量

所以让你的 network 在一开始训练的时候,每一个 dimension 的分布,是比较接近的,也许训练到后来,你已经训练够长的一段时间,已经找到一个比较好的 error surface,走到一个比较好的地方以后,那再把 γ 跟 β 慢慢地加进去,好 所以加 Batch Normalization,往往对你的训练是有帮助的

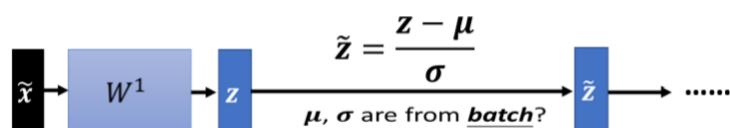
Testing

以上说的都是 training 的部分,testing ,有时候又叫 inference ,所以有人在文件上看到有人说,做个 inference,inference 指的就是 testing

这个 Batch Normalization 在 inference,或是 testing 的时候,会有什麼样的问题呢

在 testing 的时候,如果 当然如果今天你是在做作业,我们一次会把所有的 testing 的资料给你,所以你确实也可以在 testing 的资料上面,製造一个一个 batch

但是假设你真的有系统上线,你是一个真正的线上的 application,你可以说,我今天一定要等 30,比如说你的 batch size 设 64,我一定要等 64 笔资料都进来,我才一次做运算吗,这显然是不行的

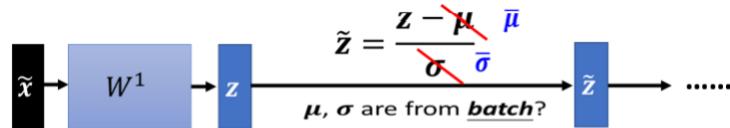


但是在做 Batch Normalization 的时候,一个 \tilde{x} ,一个 normalization 过的 feature 进来,然后你有一个 z ,你的 z 呢,要减掉 μ 跟除 σ ,那这个 μ 跟 σ ,是用一个 batch 的资料算出来的

但如果今天在 testing 的时候,根本就没有 batch,那我们要怎麽算这个 μ ,跟怎麽算这个 σ 呢

所以真正的,这个实作上的解法是这个样子的,如果你看那个 PyTorch 的话呢,Batch Normalization 在 testing 的时候,你并不需要做什麼特别的处理,PyTorch 帮你处理好了

在 training 的时候,如果你有在做 Batch Normalization 的话,在 training 的时候,你每一个 batch 计算出来的 μ 跟 σ ,他都会拿出来算 moving average



We do not always have batch at testing stage.

Computing the moving average of μ and σ of the batches during training.

$$\mu^1 \quad \mu^2 \quad \mu^3 \quad \dots \quad \mu^t$$

$$\bar{\mu} \leftarrow p\bar{\mu} + (1-p)\mu^t$$

你每一次取一个 batch 出来的时候,你就会算一个 μ^1 ,取第二个 batch 出来的时候,你就算个 μ^2 ,一直到取第 t 个 batch 出来的时候,你就算一个 μ^t

接下来你会算一个 moving average,你会把你现在算出来的 μ 的一个平均值,叫做 μ bar,乘上某一个 factor,那这也是一个常数,这个也是一个 constant,这也是一个那个 hyper parameter,也是需要调的那种啦

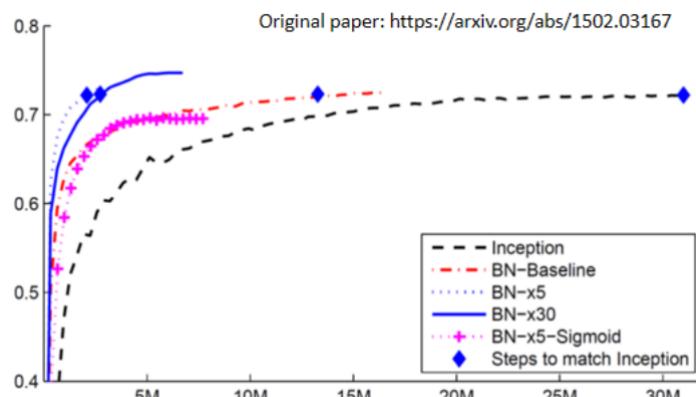
在 PyTorch 裡面,我没记错 他就设 0.1,我记得他 P 就设 0.1,好,然后加上 1 减 P,乘上 μ^t ,然后来更新你的 μ 的平均值,然后最后在 testing 的时候,你就不用算 batch 裡面的 μ 跟 σ 了

因为 testing 的时候,在真正 application 上,也没有 batch 这个东西,你就直接拿 $\bar{\mu}$ 跟 $\bar{\sigma}$,也就是 μ 跟 σ 在训练的时候,得到的 moving average, $\bar{\mu}$ 跟 $\bar{\sigma}$,来取代这边的 μ 跟 σ ,这个就是 Batch Normalization,在 testing 的时候的运作方式

Comparison

好 那这个是从 Batch Normalization,原始的文件上面截出来的一个实验结果,那在原始的文件上还讲了很多其他的东西,举例来说,我们今天还没有讲的是,Batch Normalization 用在 CNN 上,要怎麽用呢,那你自己去读一下原始的文献,裡面会告诉你说,Batch Normalization 如果用在 CNN 上,应该要长什麼样子

这个是原始文献上面截出来的一个数据

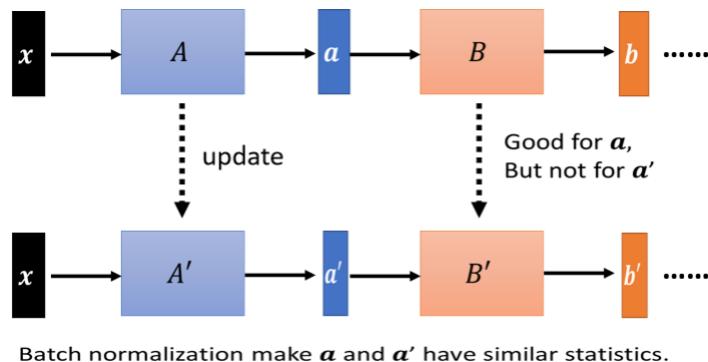


- 横轴呢,代表的是训练的过程,纵轴代表的是 validation set 上面的 accuracy
- 那这个黑色的虚线是没有做 Batch Normalization 的结果,它用的是 inception 的 network,就是某一种 network 架构啦,也是以 CNN 为基础上的 network 架构
- 然后如果有做 Batch Normalization,你会得到红色的这一条虚线,那你会发现说,红色这一条虚线,它训练的速度,显然比黑色的虚线还要快很多,虽然最后收敛的结果啊,就你只要给它足够的训练的时间,可能都跑到差不多的 accuracy,但是红色这一条虚线,可以在比较短的时间内,就跑到一样的 accuracy,那这边这个蓝色的菱形,代表说这几个点的那个 accuracy 是一样的
- 粉红色的线是 sigmoid function,就 sigmoid function 一般的认知,我们虽然还没有讨论这件事啦,但一般都会选择 ReLu,而不是用 sigmoid function,因为 sigmoid function,它的 training 是比较困难的,但是这边想要强调的点是说,就算是 sigmoid 比较难搞的,加 Batch Normalization,还是 train 的起来,那这边没有 sigmoid,没有做 Batch Normalization 的结果,因为在实验上,作者有说,sigmoid 不加 Batch Normalization,根本连 train 都 train 不起来
- 蓝色的实线跟这个蓝色的虚线呢,是把 learning rate 设比较大一点,乘 5,就是 learning rate 变原来的 5 倍,然后乘 30,就是 learning rate 变原来的 30 倍,那因为如果你做 Batch Normalization 的话,那你的 error surface 呢,会比较平滑 比较容易训练,所以你可以把你的比较不崎岖,所以你就可以把你的 learning rate 呢,设大一点,那这边有个不好解释的奇怪的地方,就是不知道为什麼,learning rate 设 30 倍的时候,是比 5 倍差啦,那作者也没有解释啦,你也知道做 deep learning 就是,有时候会产生这种怪怪的,不知道怎麼解释的现象就是了,不过作者就是照实,把他做出来的实验结果,呈现在这个图上面

Internal Covariate Shift?

好接下来的问题就是,Batch Normalization,它為什麼会有帮助呢,在原始的 Batch Normalization,那篇 paper 裡面,他提出来一个概念,叫做 internal covariate shift,covariate shift(训练集和预测集样本分布不一致的问题就叫做“covariate shift”现象) 这个词汇是原来就有的,internal covariate shift,我认为是,Batch Normalization 的作者自己发明的

他认为说今天在 train network 的时候,会有以下这个问题,这个问题是这样



network 有很多层

- x 通过第一层以后 得到 a
- a 通过第二层以后 得到 b
- 计算出 gradient 以后,把 A update 成 A' ,把 B 这一层的参数 update 成 B'

但是作者认为说,我们在计算 B ,update 到 B' 的 gradient 的时候,这个时候前一层的参数是 A 啊,或者是前一层的 output 是小 a 啊

那当前一层从 A 变成 A' 的时候,它的 output 就从小 a 变成小 a' 啊

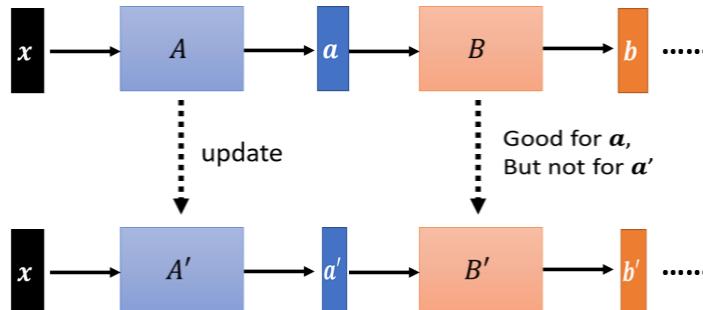
但是我们计算这个 gradient 的时候,我们是根据这个 a 算出来的啊,所以这个 update 的方向,也许它适合用在 a 上,但不适合用在 a' 上面

那如果说 Batch Normalization 的话,我们会让,因為我们每次都有做 normalization,我们就会让 a 跟 a' 呢,它的分布比较接近,也许这样就会对训练呢,有帮助

Internal Covariate Shift?

How Does Batch Normalization Help Optimization?

<https://arxiv.org/abs/1805.11604>



但是有一篇 paper 叫做,How Does Batch Normalization,Help Optimization,然后他就打脸了internal covariate shift 的这一个观点

在这篇 paper 裡面,他从各式各样的面向来告诉你说,internal covariate shift,首先它不一定是 training network 的时候的一个问题,然后 Batch Normalization,它会比较好,可能不见得是因為,它解决了 internal covariate shift

那在这篇 paper 裡面呢,他做了很多很多的实验,比如说他比较了训练的时候,这个 a 的分布的变化发现,不管有没有做 Batch Normalization,它的变化都不大

然后他又说,就算是变化很大,对 training 也没有太大的伤害,然后他又说,不管是根据 a 算出来的 gradient,还是根据 a' 算出来的 gradient,方向居然都差不多

所以他告诉你说,internal covariate shift,可能不是 training network 的时候,最主要的问题,它可能也不是,Batch Normalization 会好的一个的关键,那有关更多的实验,你就自己参见这篇文章,

為什麼 Batch Normalization 会比较好呢,那在这篇 How Does Batch Normalization,Help Optimization 这篇论文裡面,他从实验上,也从理论上,至少支持了 Batch Normalization,可以改变 error surface,让 error surface 比较不崎岖这个观点

How Does Batch Normalization Help Optimization?

<https://arxiv.org/abs/1805.11604>

Experimental results (and theoretically analysis) support batch normalization change the landscape of error surface.

and 12 of Appendix B.) This suggests that the positive impact of BatchNorm on training might be somewhat serendipitous. Therefore, it might be valuable to perform a principled exploration of the design space of normalization schemes as it can lead to better performance.

serendipitous (偶然的)

penicillin



所以这个观点是有理论的支持,也有实验的佐证的,那在这篇文章裡面呢,作者还讲了一个非常有趣的话,他说他觉得啊,这个 Batch Normalization 的 positive impact

因為他说,如果我们要让 network,这个 error surface 变得比较不崎岖,其实不见得要做 Batch Normalization,感觉有很多其他的方法,都可以让 error surface 变得不崎岖,那他就试了一些其他的方法,发现说,跟 Batch Normalization performance 也差不多,甚至还稍微好一点,所以他就讲了下面这句感嘆他觉得说,这个,positive impact of batchnorm on training,可能是 somewhat,serendipitous,什麼是 serendipitous 呢,这个字眼可能可以翻译成偶然的,但偶然并没有完全表达这个词汇的意思,这个词汇的意思是说,你发现了一个什麼意料之外的东西

举例来说,盘尼西林就是,意料之外的发现,大家知道盘尼西林的由来就是,有一个人叫做弗莱明,然后他本来想要那个,培养一些葡萄球菌,然后但是因為他实验没有做好,他的那个葡萄球菌被感染了,有一些霉菌掉到他的培养皿裡面,然后发现那些培养皿,那些霉菌呢,会杀死葡萄球菌,所以他就发明了,发现了盘尼西林,所以这是一种偶然的发现

那这篇文章的作者也觉得,Batch Normalization 也像是盘尼西林一样,是一种偶然的发现,但无论如何,它是一个有用的方法

To learn more

那其实 Batch Normalization,不是唯一的 normalization,normalization 的方法有一把啦,那这边就是列了几个比较知名的,

Batch Renormalization

<https://arxiv.org/abs/1702.03275>

Layer Normalization

<https://arxiv.org/abs/1607.06450>

Instance Normalization

<https://arxiv.org/abs/1607.08022>

Group Normalization

<https://arxiv.org/abs/1803.08494>

Weight Normalization

<https://arxiv.org/abs/1602.07868>

Spectrum Normalization

<https://arxiv.org/abs/1705.10941>