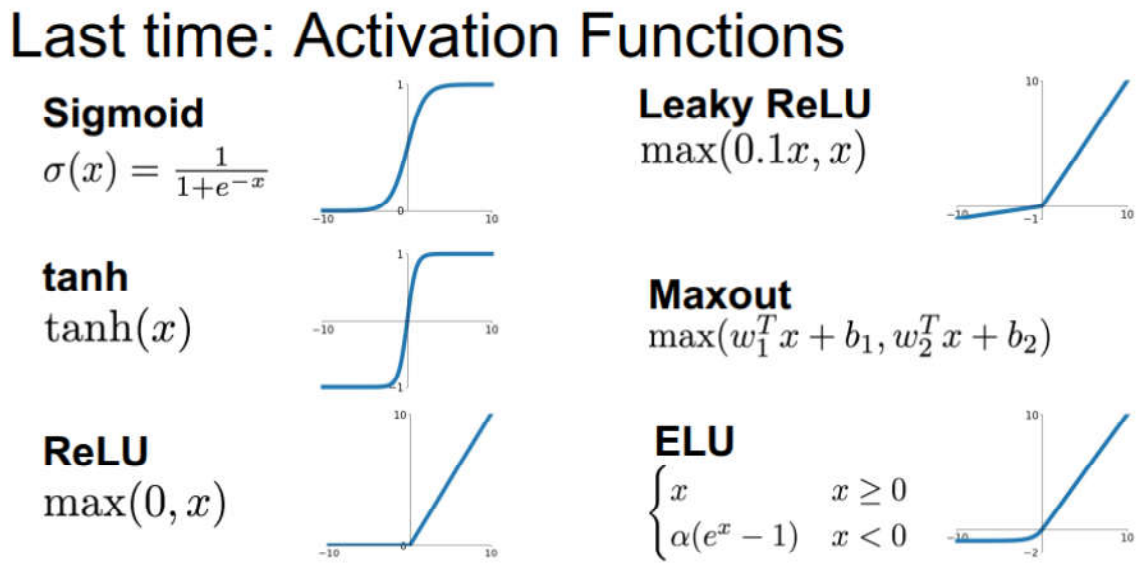
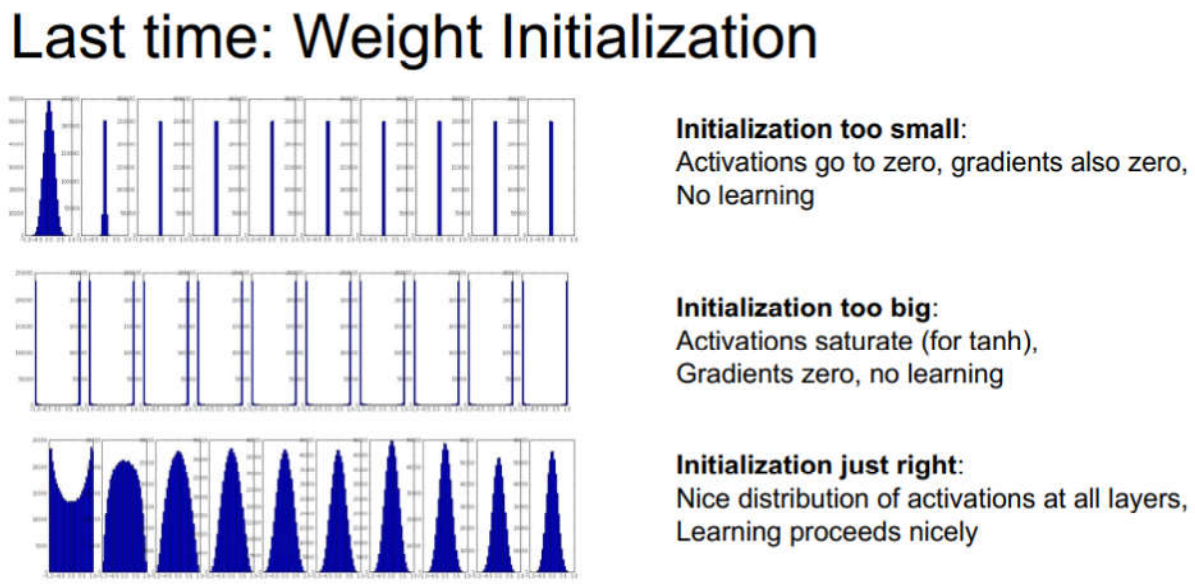
快速回顾：激活函数 Last time: Activation Functions

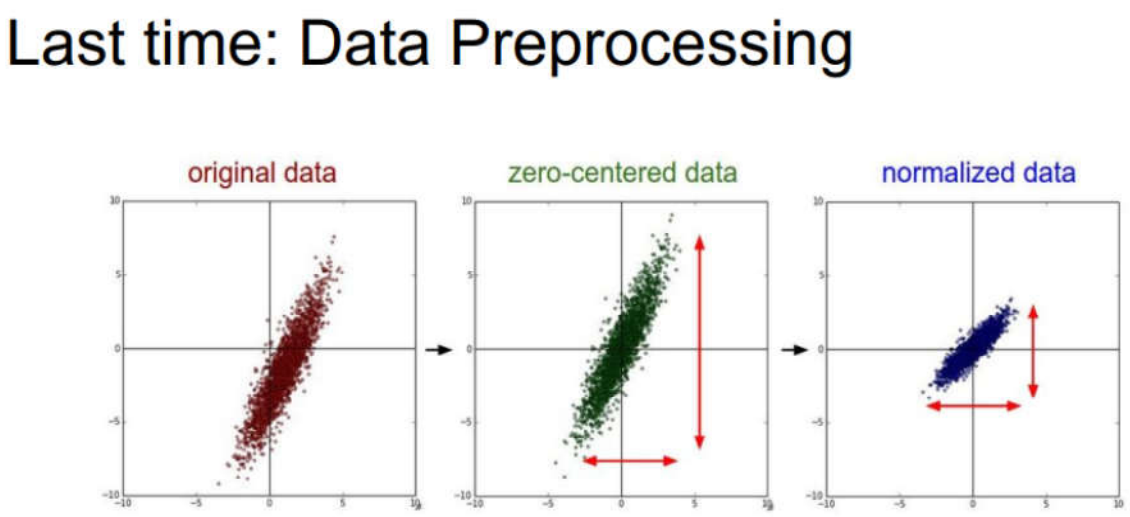


我们讨论了各种各样的激活函数和一些它们不同的属性。我们看到，大概在10年前，sigmoid激活函数曾经在训练神经网络方面十分受欢迎，但是在sigmoid激活函数的两端存在**梯度消失**的问题。tanh函数也存在类似的问题。对于这样的问题一般的建议是，你可能想要在大多数这样的情况下把ReLU激活函数作为默认的选择，因为它在很多不同的框架下，都能够运行得很好。

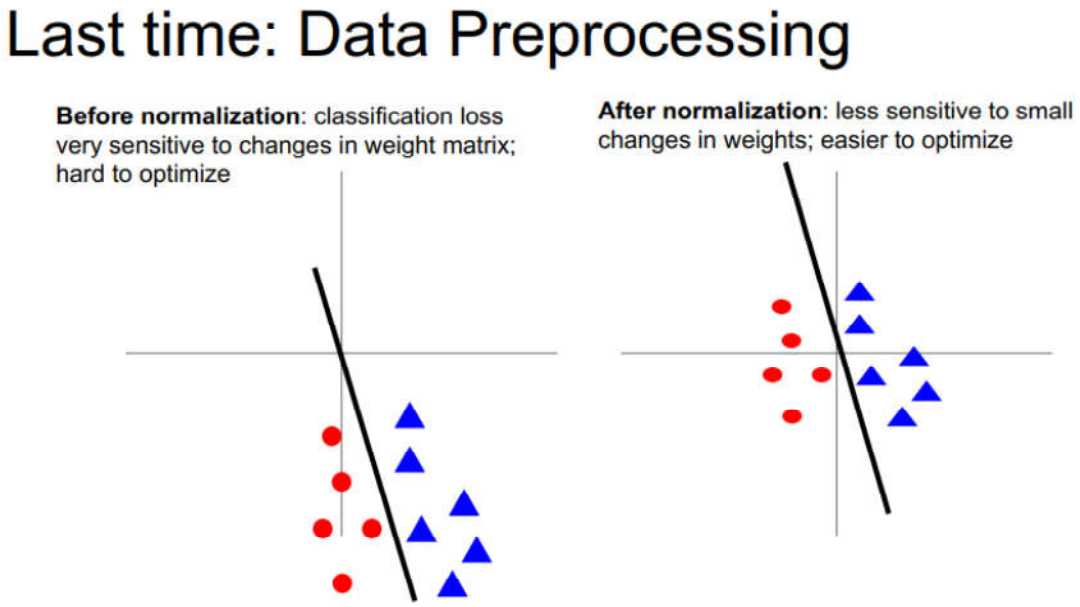


我们也讨论了权重初始化，首先要记住的是，当你们在开始训练的时候，初始化你们的权重值（即：参数W），如果那些权重的初始值太小，那么当你在学习深度网络的时候，你就会发现，激活值消失，因为当你不断的乘以这些很小的数，他们就会衰减到0，那么所有的都会变成0，学习也就无从谈起。从另一个角度来说，如果你的权重初始值过大，那么这些初始值不断地乘以你的权重矩阵，最终将会一发不可收拾（PS：爆炸性增长），无法学习。但是如果你正确的初始化权重参数，举个例子，使用Xavier初始化法或者MSRA初始化法，那么在你学习深度网络时，（在深度网

络的每一层）激活值有很好的分布。记住，在你们的深度网络越来越深的时候，权重的初始化会变得至关重要，因为随着你的深度网络的变深，你将不断地乘以那些权值矩阵。



上次我们也讨论过关于数据预处理方面的问题，我们讲过，在卷积神经网络中，中心化和归一化是非常常用的手段，它会使数据分布均值为零，方差为一。再直观的讲一下这样做的原因，举一个简单的例子，我们要通过二元分类的方法，分离这些红色的点和蓝色的点，从左边的图来看，如果这些数据点没有被归一化，没有被中心化，而且距离（坐标系）原点很远，那么虽然我们仍然可以用一条直线（如果所示）分离它们，但是现在如果这条直线稍微转动一点，那么我们的分类器将会被完全的破坏，这意味着，在左边的例子中，损失函数对我们的权重矩阵中的线性分类器中的小扰动非常敏感。我们仍然可以表示相同的函数，但是这会使深度学习变得异常艰难，再一次强调，因为它们的损失对我们的参数向量非常敏感。



而在右边的情况下，如果你在使用数据集时，讲那些数据点移动到原点附近，并且缩小它们的单位方差，再一次强调，我们仍然可以很好的对这些数据进行分类，但是现在我们稍微的转动直线，我们的损失函数对参数值中的小扰动就不那么敏感了。这可能会优化变得更容易一些的同时能够看到一些进展。顺便提一下，这种情况，不仅仅在线性分类中遇到，记住，在神经网络里我们需要交叉

地使用线性矩阵相乘或者卷积，还有非线性激活函数。如果神经网络中，某一层的输入均值不为0或者方差不为1，该层网络权值矩阵的微小摄动就会造成该层输出的巨大摄动，从而造成学习困难。所以这就直观地解释了为什么归一化那么重要。

Last time: Batch Normalization

Input: $x : N \times D$

Learnable params: $\gamma, \beta : D$

Intermediates: $\mu, \sigma : D$, $\hat{x} : N \times D$

Output: $y : N \times D$

$$\mu_j = \frac{1}{N}\sum_{i=1}^{N} x_{i,j}$$

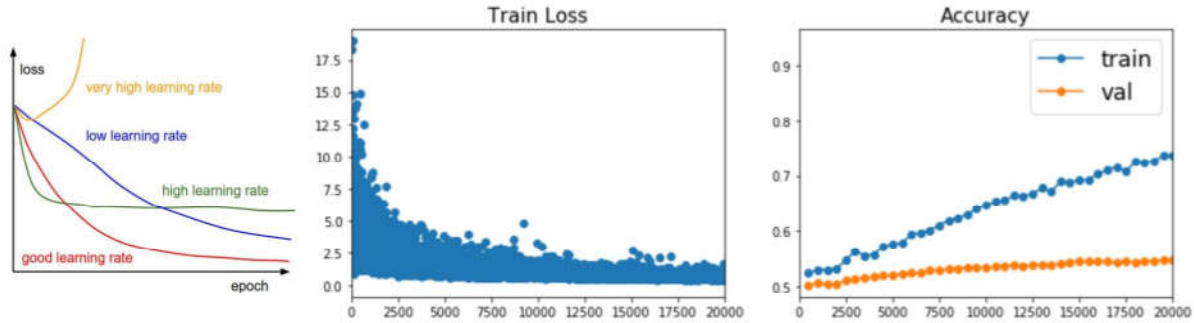
$$\sigma_j^2 = \frac{1}{N}\sum_{i=1}^{N}(x_{i,j} - \mu_j)^2$$

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

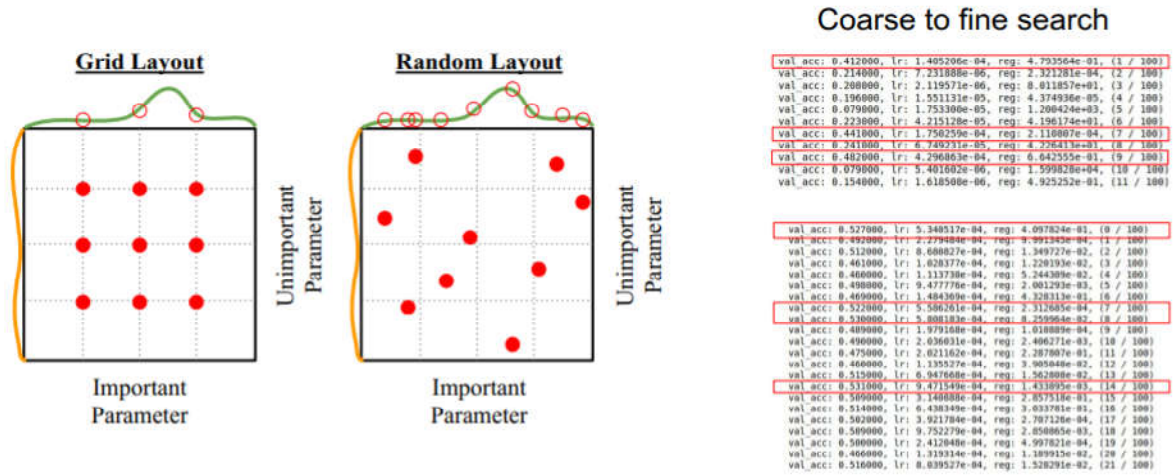
$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$

还要记住，因为我们了解了归一化的重要性，所以引入了Batch Normalization的概念，即在神经网络中加入额外一层，以使得中间的激活值均值为0方差为1，在这里，我们通过几个更直观的形式，重新总结了Batch Normalization的方程，在Batch Normalization中，正向传递时，我们用小批量的统计数据计算平均值和标准差，并用这个估计值，并且对数据进行归一化，之后我们还介绍了缩放参数和平移参数来增加一层的可表达性。

Last time: Babysitting Learning



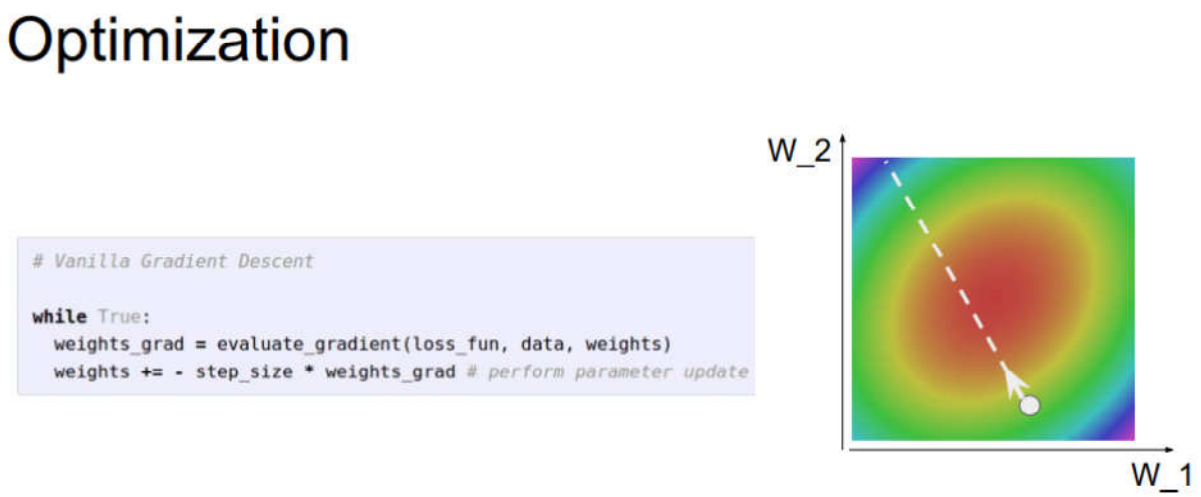
上次我们还讲了一小块跟踪学习过程的内容，比如在训练过程中，如何观察损失函数曲线，以上是一些神经网络的例子，左侧是一些随时间变化的损失函数曲线，你会发现曲线多多少少是下降的，说明神经网络的损失函数在降低，这是个好的信号。右侧曲线，x轴同样是训练时间，或者迭代次数，y轴表示模型在训练集和验证集上的效果。你会发现，随着时间增加，训练集上效果，随着损失函数下降越来越好，但验证集上的表现，却在某一点之后不再上升，这表明模型进入了过拟合的状态。这时候可能就需要加入其他的正则化手段。



上次我们还简要涉及了超参数搜索，所有这些神经网络涉及到大量超参数，找到正确的参数十分重要。我们讲到了网格搜索，以及随机搜索在理论上的优越性在哪里。因为当你的模型性能对某一个超参数比对其他超参数更敏感的时候，随机搜索可以对超参数空间覆盖的更好。我们还介绍了粗细粒交叉搜索，当你做超参数优化的时候，一开始可能处理很大的搜索范围，几次迭代之后就可以缩小范围，圈定合适的超参数所在的区域，然后再对这个小范围，重复这个过程，你可以多次进行上述的步骤，以获得超参数的右区域。另外很重要的一点是，一开始你得确定粗略的范围，这个范围要非常宽，覆盖你所有的超参数，理想情况下，范围应该足够宽到你的网络不会超过范围的任一边，这样你就知道自己的搜索范围足够大。

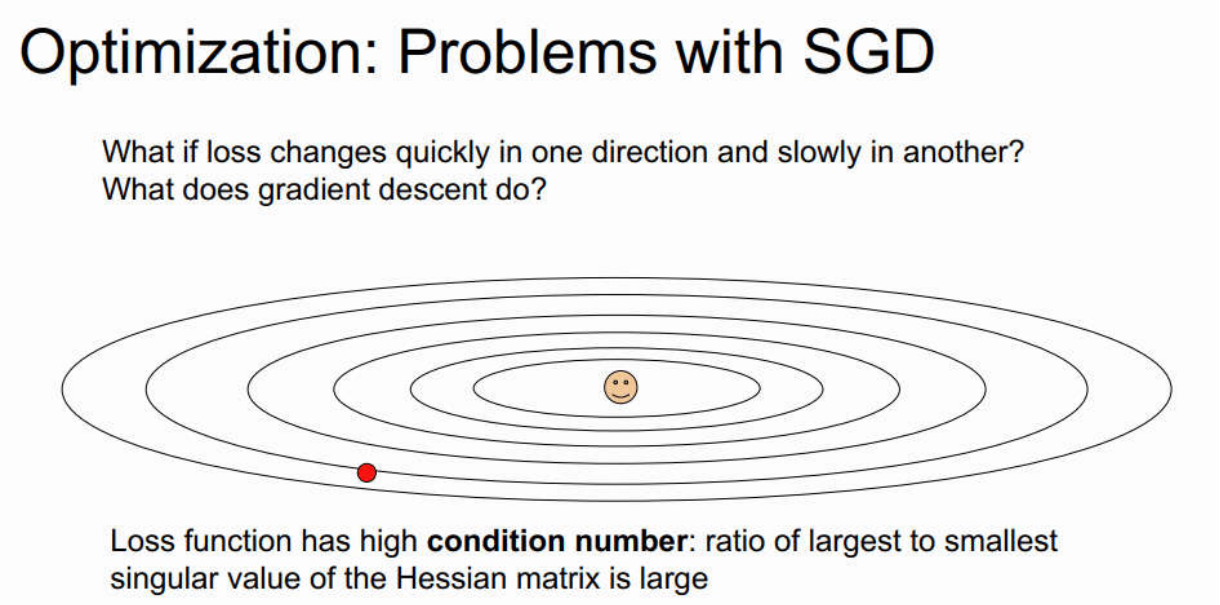
优化 Optimization

训练神经网络的核心策略是一个优化问题。

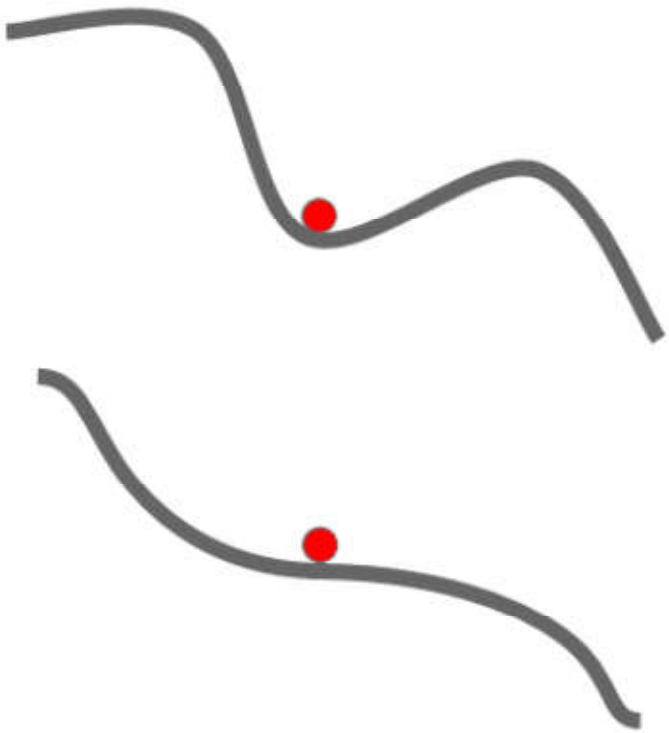


最简单的优化算法：随机梯度下降，它只有三行代码。我们首先评估一下一些小批量数据中损失的梯度，更进一步，想梯度为负的方向，更新参数向量。因为，它给出了损失函数下降最快的方向，然后我们重复这个过程，幸运的话，它在红色区域收敛，我们如愿得到很小的误差值。但不幸的是，这个相对简单的优化算法，在实际使用中会产生很多问题。随机梯度下降算法的问题之一，想

象一下我们的目标函数发生了什么，就像这样，我们画两个值W_1和W_2，当我们改变其中之一时，损失函数变化非常慢，当我们在水平方向改变值，损失函数变化非常慢，当我们在等高线图上下运动，损失值则对竖直方向的变化非常敏感，顺便提一句，对于损失值来说，在这一点是很坏的情况，在这一点，它是Hessian矩阵中，最大奇异值和最小奇异值之比，但是直观来看，损失值等高线图看起来像一个玉米卷饼，它在一个方向上非常敏感，而在其他方向则并不敏感。



问题是，对于一个像这样的函数SGD会做什么？ 如果你在这类函数上，运行随机梯度下降算法，你会得到这样特有的之字形图形，其原因正是因为这类目标函数梯度的方向并不是与最小值成一条线，当你计算梯度并沿着前进时，你可能一遍遍跨过这些等高线，之字形地前进或后退，所以，你在水平维度上的前进速度非常慢，在这个方向上的敏感度较低，但是却在非常敏感的垂直维度上做之字形运动，这并不是我们所希望的，而且，事实上，这个问题在高维空间变得更加普遍，在这种卡通图中，我们只展示了两维优化等高线图，但在实际中，我们的神经网络可能包含百万、千万甚至上亿个参数，它将会沿着上亿个方向进行运动，在这个上亿个不同的运动方向上，介于最大值和最小值的方向上的比例可能很高，SGD表现并不好，你可以想象如果我们有一亿个参数，在它们两者之间的最大比例可能很大。我认为在很多多维问题中，这是一个大问题。



SGD的另一个问题是局部极小值或鞍点（saddle oints不是极大值也不是极小值的临界点），在这种情况下SGD会发生什么？在这种情况下，SGD会卡在中间，因为那里是局部极小值，梯度为0，因为那一段是平的，我们计算梯度，向梯度相反的方向前进，在目前的点上，相反的梯度值为0，我们不会做任何动作，我们就被卡在了这个位置。关于鞍点还有另外一个问题，相比于局部极小值，你可以设想一点往一个方向是向上，往另一个方向是向下，在当前的位置梯度为0，在这种情况下，函数将被卡在鞍点，因为这里梯度为0。虽然我希望指出一点，在一维问题上，局部极小值看起来是个大问题，鞍点看起来并不需要担心，但是事实上，一旦涉及到高维问题，恰恰相反，如果你想想，一亿个参数的空间，鞍点意味着什么，它意味着在当前点上某些方向上损失会增加，某些方向损失会减小。如果你有一亿个维度，它发生得会更加频繁，基本上几乎在任何点上都会发生，然而在局部极小值点上，向一亿个方向中，任何一个方向前进损失都会变大，事实上，当你考虑这种很高维的问题时，看起来似乎这种情况非常稀少。这个问题是最近几年在训练非常大的神经网络过程中才显露出来，问题更多的出在鞍点上，局部极小值问题少一点，不过有时候问题并非恰好在鞍点上，也可能在鞍点附近。

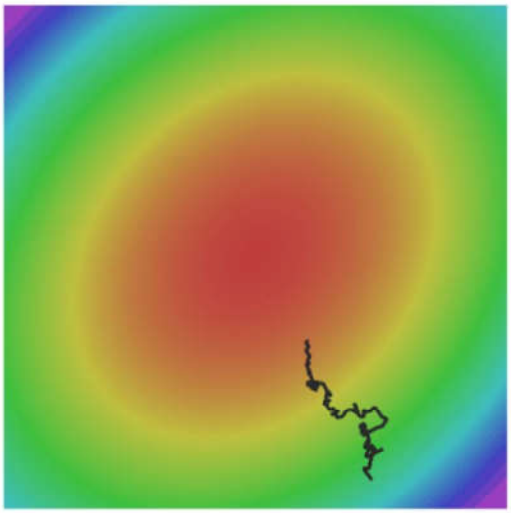
如果你看看下面的例子，可以看见鞍点附近，梯度并不是0，但是斜率非常小，这意味着，如果我们向梯度方向前进，而梯度非常小，任何时候当前参数值在目标等高线上接近鞍点时，我们前进都会非常缓慢，事实上，这是个大问题。随机性是SGD另一个问题是S，记住SGD是随机梯度下降。回忆一下，损失函数，是通过多次重复计算不同实例的损失来定义的，在这个例子中，如果N是你的整个训练集，可能有一百万个，每个计算损失都会耗费很大的计算量。事实上，记住我们经常通过小批的实例，对损失和梯度进行估计，这意味着，事实上我们并不会每一步都去计算真实的梯度，而是在当期点，对梯度进行噪声估计，在右边，我对图做了一点修改，我只是在每一点的梯度上加入了随机均匀噪声，搞乱梯度，在这样的噪声条件下运行SGD，这可能并不完全是SGD过程发生的事情，但这仍然给你了一种感觉，如果在你的梯度估计中存在噪声，那么常规额SGD，这种周围曲折的空间，可能实际上需要花费很长时间才能得到极小值。

Optimization: Problems with SGD

Our gradients come from minibatches so they can be noisy!

$$L(W) = \frac{1}{N}\sum_{i=1}^{N} L_i(x_i, y_i, W)$$

$$\nabla_W L(W) = \frac{1}{N}\sum_{i=1}^{N} \nabla_W L_i(x_i, y_i, W)$$



解决这个问题，有一个非常简单的策略，在这些很多问题上，都能表现地很好，这个想法就是，在我们的随机梯度下降中，加入一个动量项，在左侧，是我们经典的SGD，我们通常只是在梯度的方向上步进。但在右侧，有一个非常小的方差，称为带动量的SGD，它包含两个等式和五行代码，所以它变得加倍复杂，但它真的很简单，这个思想就是，保持一个不随时间变化的速度，并且我们将梯度估计添加到这个速度上，然后，在这个速度的方向上步进，而不是在梯度的方向上步进，这真的非常简单，我们也有这个和摩擦有关的超参数 ，现在，在每一步我们采用当前的速度，然后用摩擦系数 来对其衰减，摩擦系数有时取值比较大，例如0 就是一个通常的选择，我们采用当前的速度，然后使用摩擦系数进行衰减，之后加到梯度上，现在，我们在速度向量的方向上步进，而不是在原始梯度向量的方向上步进，这个超级简单的策略，实际上帮助解决了所有我们刚才讨论过的问题。

SGD + Momentum

SGD

$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$

```
while True:
  dx = compute_gradient(x)
  x += learning_rate * dx
```

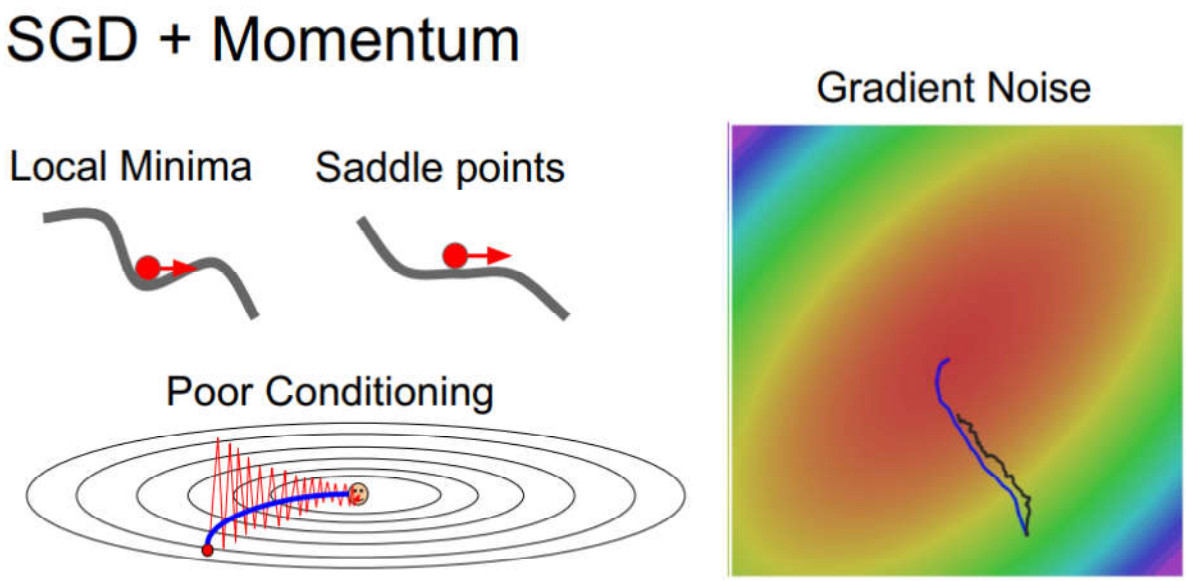
SGD+Momentum

$$v_{t+1} = \rho v_t + \nabla f(x_t)$$
$$x_{t+1} = x_t - \alpha v_{t+1}$$

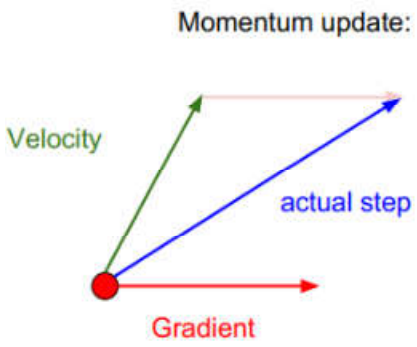
```
vx = 0
while True:
  dx = compute_gradient(x)
  vx = rho * vx + dx
  x += learning_rate * vx
```

- Build up “velocity” as a running mean of gradients
- Rho gives “friction”; typically rho=0.9 or 0.99

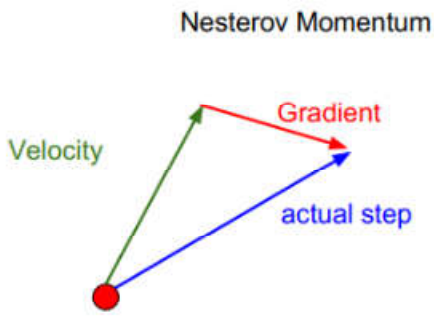
如果你去思考在局部极小值点或者鞍点发生了什么，可以想象这个系统中的速度，你可能可以理解为一个球滚下山，它会在下降的时候速度变快，现在，一旦加上了速度，那么甚至当，通过局部极小值点的时候，在这个点仍然会有速度，即使在这里没有梯度，那么我们就能够越过这个局部极小值点，然后继续下降，在鞍点附近也是类似，虽然鞍点附近的梯度非常小，但是我们还有在下山时就建立起来的速度向量，这能够帮助我们通过鞍点，并且让我们继续滚动下去，如果你去思考在不好的情况下会发生什么，现在如果我们有这种曲曲折折的近似梯度，那么一旦我们使用动量，这些之字形的曲折就会很快相互抵消，这会很有效地减少我们朝敏感方向步进的数量，而在水平方向上，我们的速度只会不断增加，实际上还会加速在，不那么敏感的维度上下降，在这里添加动量，实际上也会帮助我们处理高条件数的问题。最后在右侧，我们重现了，同样的带有噪声的梯度下降可视化过程，这条黑色的曲线是常规SGD，它会曲曲折折地经过所有地方，这条蓝色曲线展示的是带有动量的，你能够看到这个，是因为我们添加了动量，我们随着时间变化得到了速度，而噪音在估计梯度时，被抵消掉了一下。现在，相比于因为噪声而路径有些曲折的SGD，带动量的SGD最终更加平稳地接近最小值点。



当我们在使用带有动量的SGD时，你可以想象这样一幅画面，这个红色的点事我们当前的位置，有一些红色的向量表示梯度，或者我们队当前位置梯度估计的方向。绿色向量是速度向量的方向，当我们做动量更新时，实际上，我们是在根据这两者的平均权重进行步进，这有助于克服梯度估计中的一些噪声。有事你会看到动量的一个轻微变化，叫做Nesterov加速梯度，有时也称之为Nesterov动量。它把这个顺序改变了一下，在普通的SGD动量中，我们估算当前位置的梯度，然后取速度和梯度的混合，在Nesterov加速梯度中，你要做的事情有一点不同，你从红色点开始，然后在取得的速度的方向上进行步进，之后，你评估这个位置的梯度，随后，回到初始位置，将这两者混合起来。



SGD + Momentum



Nesterov Momentum

Nesterov Momentum

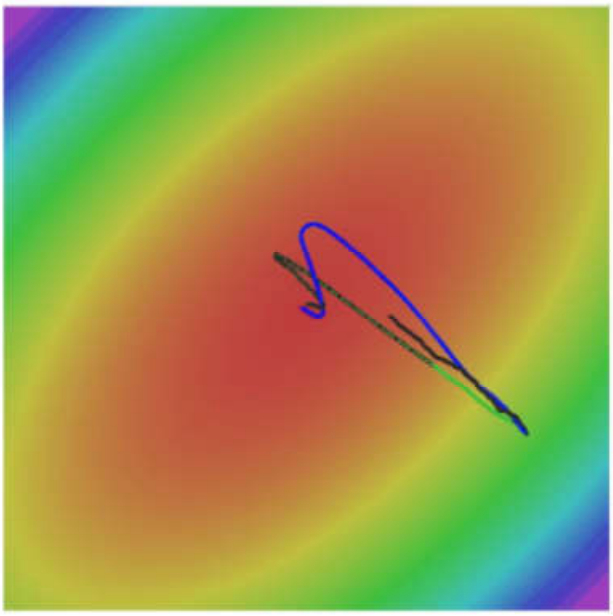
$$v_{t+1} = \rho v_t - \alpha \nabla f(x_t + \rho v_t)$$
$$x_{t+1} = x_t + v_{t+1}$$

Annoying, usually we want update in terms of $x_t, \nabla f(x_t)$

Change of variables $\tilde{x}_t = x_t + \rho v_t$ and rearrange:

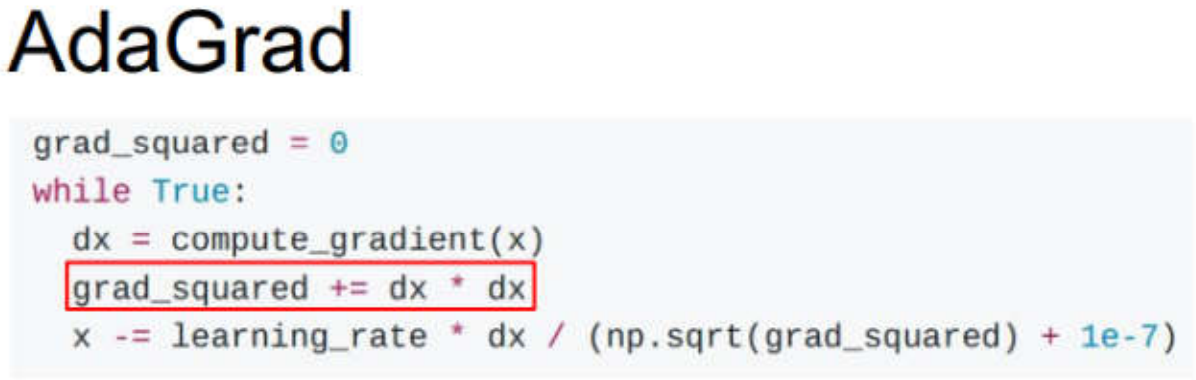
$$v_{t+1} = \rho v_t - \alpha \nabla f(\tilde{x}_t)$$
$$\tilde{x}_{t+1} = \tilde{x}_t - \rho v_t + (1+\rho) v_{t+1}$$
$$= \tilde{x}_t + v_{t+1} + \rho(v_{t+1} - v_t)$$

```
dx = compute_gradient(x)
old_v = v
v = rho * v - learning_rate * dx
x += -rho * old_v + (1 + rho) * v
```



另一种常见到的优化策略是斯坦福的 ohn D chi教授在攻读博士期间提出的AdaGrad算法，AdaGrad的核心思想是： （Added element ise scaling o the gradient ased on the historical

s m o s ares in each dimension ）在你优化的过程中，需要保持一个在训练过程中的每一步的梯度的平方和的持续估计，与速度项不同的是，现在我们有了一个梯度平方项，在训练时，我们会一直累加，当前梯度的平方到这个梯度平方项，当我们在更新我们的参数向量时，我们会除以这个梯度平方项。



1、那么现在的问题是这样的放缩对于矩阵中条件数很大的情形有什么改进呢？ What ha ens ith AdaGrad

是的，这个思想就是如果我们有两个坐标轴，沿其中一个轴我们有很高的梯度，而另一个轴方向却有很小的梯度，那么随着我们累加小梯度的平方，我们会在最后更新参数向量时除以一个很小的数字，从而加速了在小梯度维度上的学习速度，一个轴的情况是这样。然后在另一个维度方向上，由于梯度变得特别大，我们会除以一个非常大的数，所以我们会降低这个维度方向上的训练速度。

2、不过这里有一个问题，也就是当t（时间）越来越大的时候，在训练的过程中使用AdaGrad会发生什么？ 2 What ha ens to the ste size over long time

使用了AdaGrad步长会变得越来越小，因为我们一直在随时间更新梯度平方的估计值，所以这个估计值在训练过程中，一直随时间单调递增，这回导致我们的步长随时间越来越小。在学习目标是一个凸函数的情况下，有理论证明这个特征效果很好。因为当你接近极值点时，你会逐渐的慢下来最后到达收敛，这点是（AdaGrad）在凸函数情况下的一个很好的特性。但是在非凸函数的情况下，事情会变得复杂，因为当你到达一个局部的极值点时，使用AdaGrad会让你在这里被困住，从而使得训练过程无法再进行下去。因此我们队AdaGrad有一个变体叫做RMSPro ，它实际上就考虑到了这个问题。

RMSProp

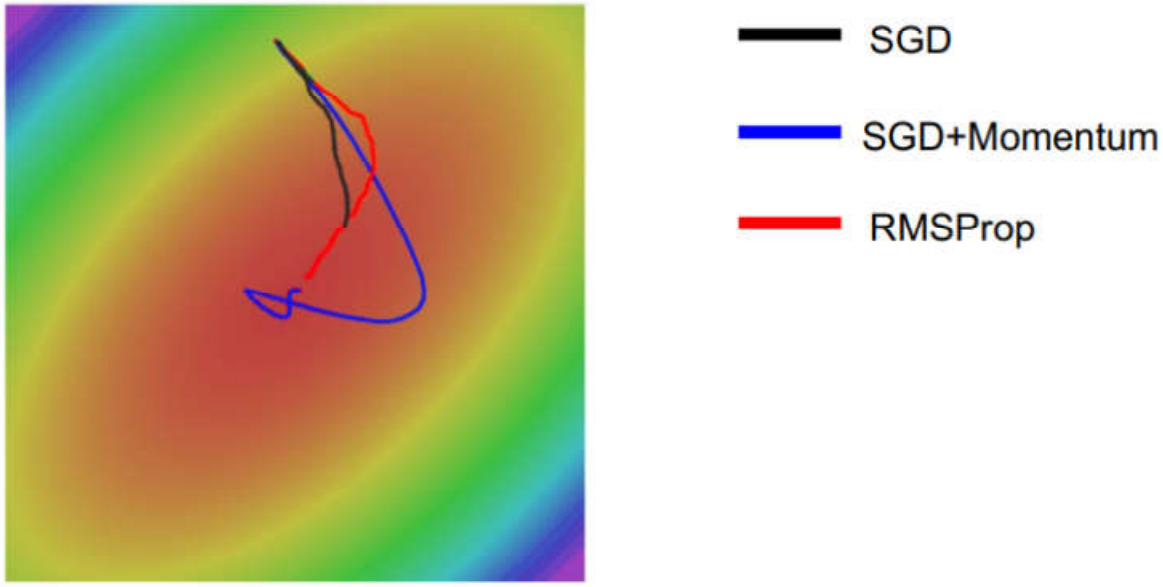
AdaGrad

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared += dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

RMSProp

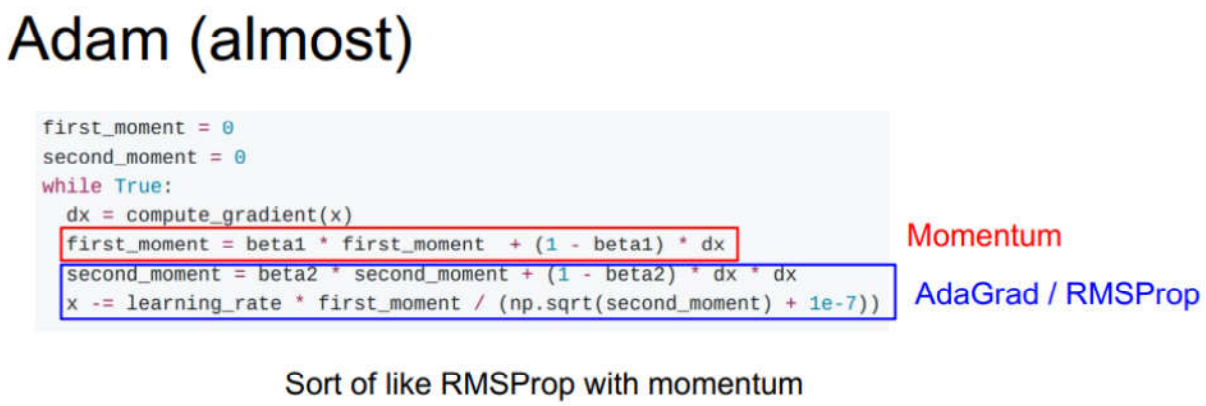
```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared = decay_rate * grad_squared + (1 - decay_rate) * dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

在RMSPro 中，我们仍然计算梯度的平方，但是我们并不是仅仅简单的在训练中累加梯度平方，而是我们会让平方梯度按照一定比率下降，他看起来就和动量优化法很像，除了我们是给梯度的平方加上动量，而不是给梯度本身，有了RMSPro ，在我们计算完梯度之后，我们取出当前的梯度平方，将其乘以一个衰减率，通常是0 或者是0 ，然后用1减去衰减率乘以梯度平方，加上之前的结果，现在随着训练的进行，你们可以想象得到的是我们的步长和AdaGrad一样，在被除了平方梯度之后，步长会有一个良好的性质，在一个维度上（梯度下降很慢的）训练会加快，而在另一个维度方向上训练减慢，而现在有了RMSPro ，由于梯度平方估计被衰减了，这有可能会造成训练总是一直在变慢，而这可能不是我们想要的。



在AdaGrad和RMSPro 中我们有另一套方法，先求梯度平方的估计值，然后除以梯度的平方的实际值。这两种方法单独来看都是不错的方法。把它们结合起来，这就引入了Adam算法，或者说接近Adam的算法。使用Adam，我们更新第一动量和第二动量的估计值。在红框里，我们让第一动量的估计值等于我们梯度的加权和，我们有一个第二动量的动态估计值，像AdaGrad和RMSPro 一样，就是一个梯度平方的动态近似值，现在我们来看看怎么更新它们，我们使用第一动量，有点类似于

速度，并且除以第二动量或者说第二动量的平方根，就是这个梯度平方项，这样的话，Adam最后看起来有点像RMSPro 加上动量，或者看起来像动量加上第二个梯度平方，就像是合并了两者各自好的性质。



但是这里也存在一点问题，这个问题就是在最初的第一步，会发什么？ What ha ens at irst timeste

在最初的第一步，你们可以看到，在开始时，我们已经将第二动量初始化为0，第二动量经过一步更新后，通常 eta2也就是第二动量的衰减率，大概是0 或0 ，非常接近于1的一个数，经过一次更新，第二动量仍然非常接近于0，现在我们在这作出更新步骤，除以第二动量，也就是一个非常小的数，俺么我们在一开始就会得到一个很大的步长，这个在开始时非常大的步长，并不是因为这一步的梯度太大，只是因为我们人为地将第二动量初始化成了0。

Adam算法也增加了偏置校正项，来避免出现开始时得到很大步长的问题出现，在我们更新了第一和第二动量之后，我们构造了第一和第二动量的无偏估计，通过使用当前时间步t，现在我们实际上在使用无偏估计来做每一步更新，而不是初始的第一和第二动量的估计值，这样我们就得到了Adam算法的完整形式。顺便说一下，Adam确实是一个非常好的最优化算法，并且对于不同的问题使用Adam算法都能得到非常不错的结果。因此，Adam差不多是我的一个用来解决任何新问题的默认算法。特别是，如果你将 eta1设置为0 ， eta2设置为0 ，学习率为1e 或者 e ，我无论使用什么网络架构，都会从这个设定开始。大家可以试试看（Adam算法）一般情况下真的是首选。

Adam (full form)

```
first_moment = 0
second_moment = 0
for t in range(1, num_iterations):
  dx = compute_gradient(x)
  first_moment = beta1 * first_moment  + (1 - beta1) * dx
  second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
  first_unbias = first_moment / (1 - beta1 ** t)
  second_unbias = second_moment / (1 - beta2 ** t)
  x -= learning_rate * first_unbias / (np.sqrt(second_unbias) + 1e-7))
```

Momentum

Bias correction

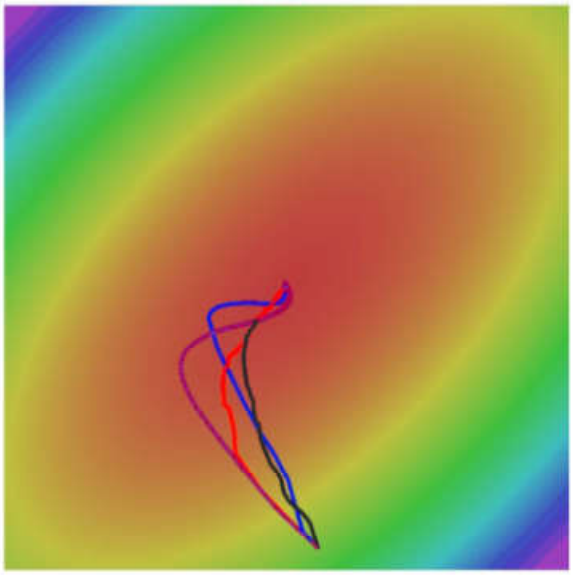
AdaGrad / RMSProp

Bias correction for the fact that first and second moment estimates start at zero

Adam with beta1 = 0.9, beta2 = 0.999, and learning_rate = 1e-3 or 5e-4 is a great starting point for many models!

Kingma and Ba, "Adam: A method for stochastic optimization", ICLR 2015

Adam



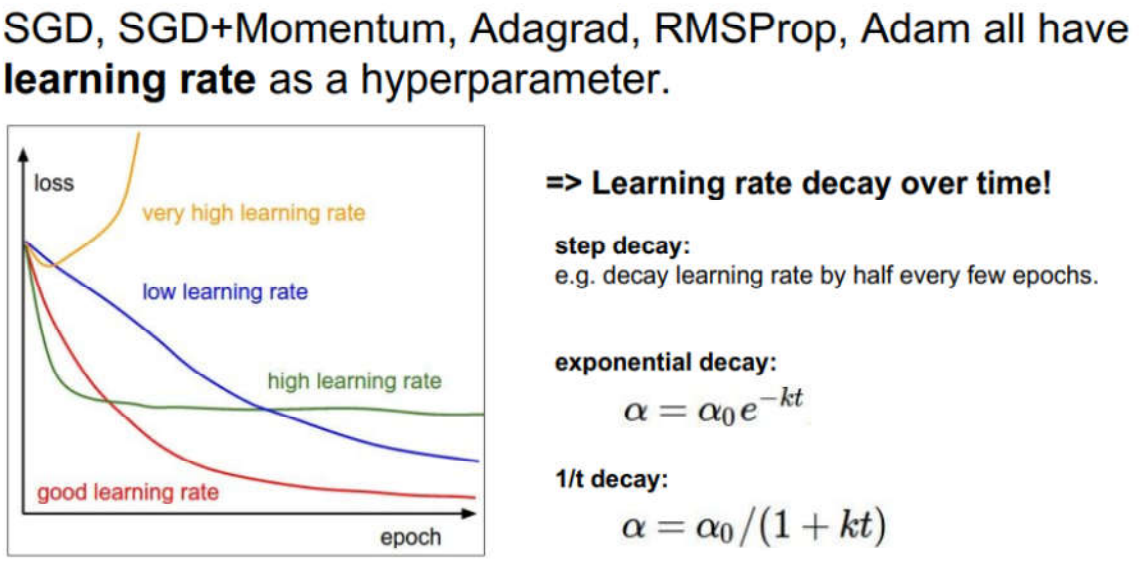
SGD

SGD+Momentum

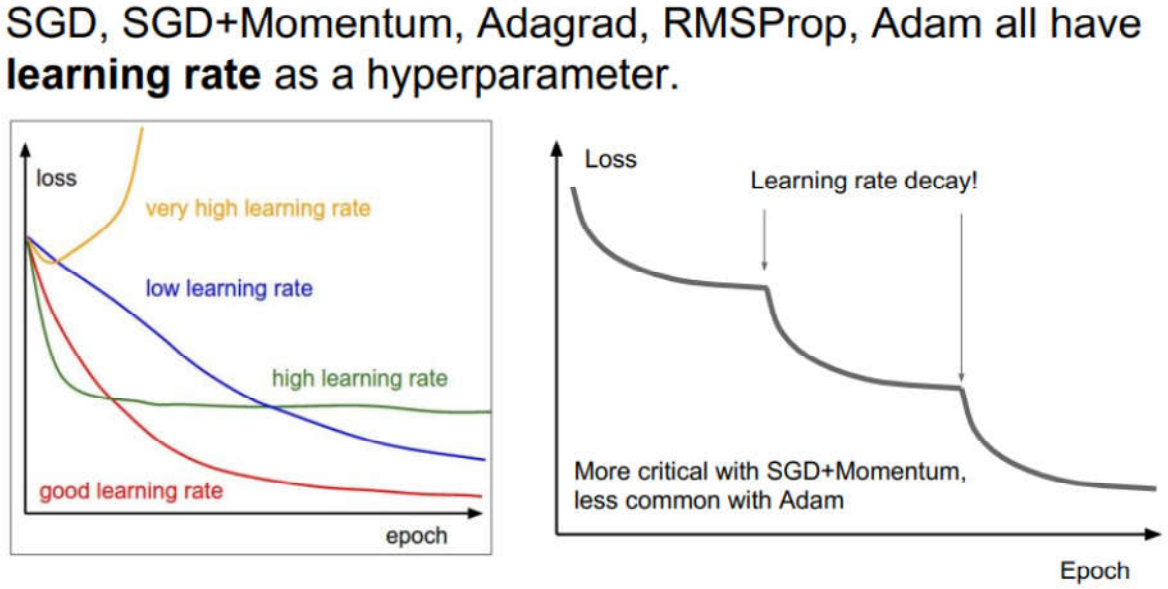
RMSProp

Adam

另一个事情是学习率，这些优化算法都有这个超参数（hy er arameter），当你使用不同的学习率时，有时候太高了就会导致（损失函数）爆炸，如黄色的曲线，如果学习率很小，如蓝色的曲线，可能要花很长的时间才收敛，挑选正确的学习率确实需要点技巧。这个技巧是，我们不必在整个训练的过程中都一直固定使用同一个学习率，有时候人们会把学习率沿着时间衰减，有点像是结合了，左图中不同的曲线的效果，而且是每个图里好的性质，比如在训练开始的时候用较大的一些学习率，然后在训练的过程中，逐渐衰减地越来越小，一个衰减的策略是步长衰减，比如在第10万次迭代时，可以衰减一个因子，然后继续训练，还有指数衰减，这种是训练时持续衰减，你可以看到训练时，学习率连续衰减的不同做法。

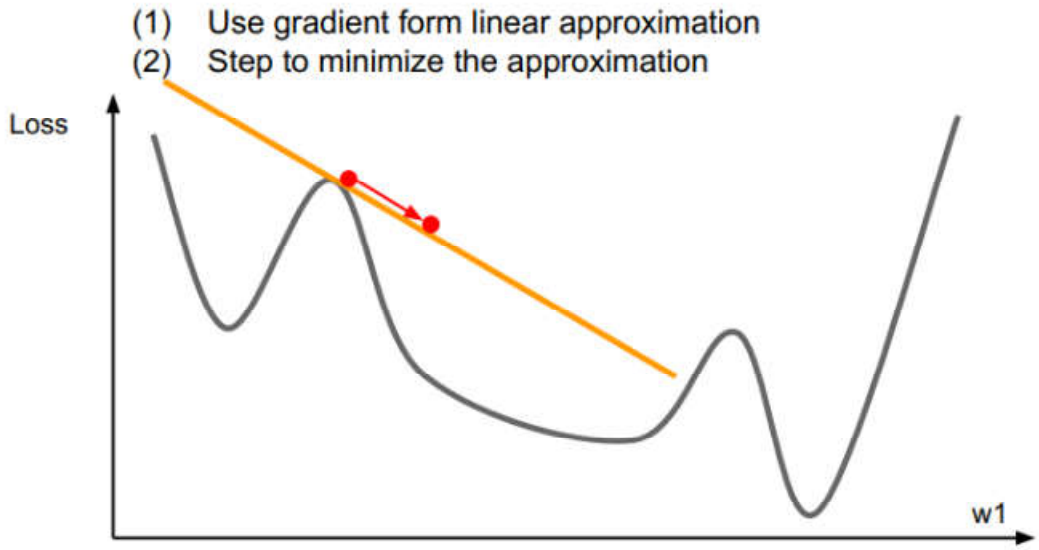


残差网络那篇论文，你会经常看到像这样的曲线，可以看到损失先一直下降，然后骤降，再然后平坦，接着又骤降，这些曲线背后，其实是它们在用步长衰减的学习率，那些曲线中出现骤降的地方，是在迭代时把学习率乘上了一个因子。



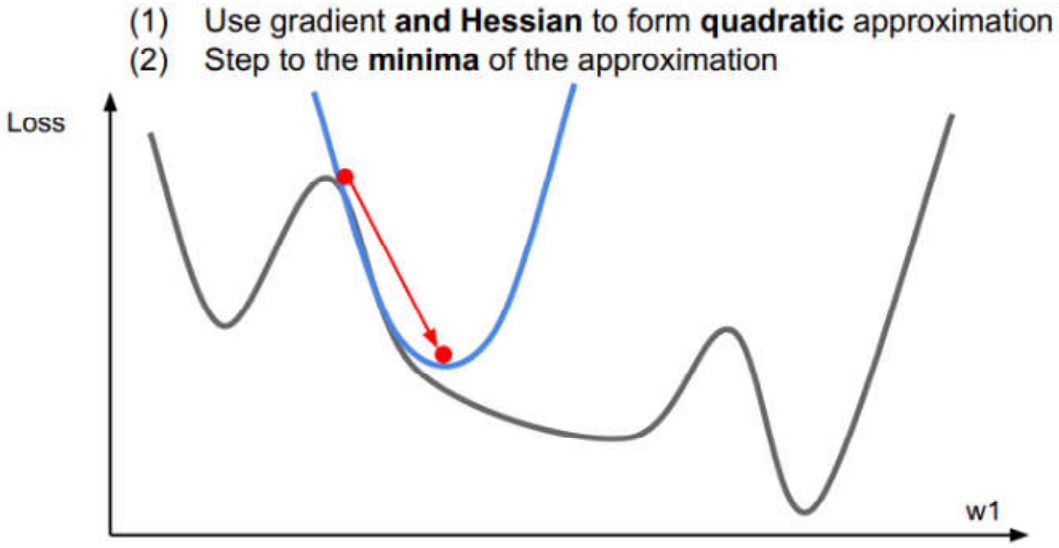
降低学习率的想法是说，假设模型已经接近一个比较不错的取值区域，但是此时的梯度已经很小了，保持原有学习速率只能在最优点附近来回徘徊，如果我们降低了学习率，目标函数仍然能够进一步降低，即在损失函数上进一步取得进步，这个有时候在实际中很有用，不过值得指出的一点是，带动量SGD的学习率衰减很常见，但是想Adam的优化算法就很少用学习率的衰减。另一点要指出的是，学习率衰减是一种二阶的超参数，你通常不用改一开始就用上学习率衰减这样的事情。通常你想要让神经网络开始工作，你想要挑选一个不带学习率衰减的，不错的学习率来作为开始，尝试在交叉验证中同时调学习率衰减和初始学习率等等其他的事情。设置学习率衰减的方法是，先尝试不用衰减，看看会发生什么，然后仔细观察损失曲线，看看你希望在哪个地方开始衰减。

First-Order Optimization



我们之前谈的这些算法都是一阶优化算法，在这张图里，这个一维的图中。我们有一个目标函数曲线，当前点是这个红色的点，我们在这个点上，求一个梯度，我们用梯度信息来计算，这个函数的线性逼近，这相当于是对我们的函数进行的一阶泰勒逼近，现在假设我们的一阶逼近，就是实际的函数，然后我们想要迈出一步，来找到逼近的最小值，但是这个逼近在稍大的区间内，并不成立，所以我们不能，朝那个方向一下走太多，事实上，这里梯度的想法，用上了函数的一阶偏导。

Second-Order Optimization



你可以多做一些工作，其实是有二阶逼近，这里我们同时考虑一阶和二阶偏导信息，现在我们队函数做一个二阶泰勒逼近，就是用一个二次函数来局部逼近我们的函数，因为是二次函数，可以直接跳到最小值点，这就很开心了。这个就是二阶优化的思想。

Second-Order Optimization

second-order Taylor expansion:

$$J(\boldsymbol{\theta}) \approx J(\boldsymbol{\theta}_0) + (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0) + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \boldsymbol{H}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)$$

Solving for the critical point we obtain the Newton parameter update:

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \boldsymbol{H}^{-1} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0)$$

当把这个思想推广到多维的情况，就会得到一个叫做牛顿步长的东西，计算这个Hessian海森矩阵，即二阶偏导矩阵，接着求这个海森矩阵的逆，以便直接走到对你的函数，用二次逼近后的最小值的地方。

有人发现和我们之前的方法相比，这里的更新规则有什么不一样的地方吗？ What is nice a o t this date

这里没有学习率，没有超参数。

我们是做了二阶逼近，直接走到这个二次函数的最小值点，至少在这个牛顿法的原始版本中，你是不需要学习率的，每次只需要直接走到最小的点就可以了，然而实际中，你可能也会用上一个学习率，因为这个二次逼近也不是完美的，你可能只是想要沿着二次函数最小值的方向前进，而不是走到最小值的位置，不过在这个原始版本中，是没有学习率的。

2 Why is this ad or dee learning 不幸的是，这个算法对深度学习来说有点不切实际，因为这个海森矩阵是N N的，其中N表示网络中参数的数量，如果N是一亿，那么一亿的平方非常大，内存肯定是存不下的，也肯定没办法求这个矩阵的逆。

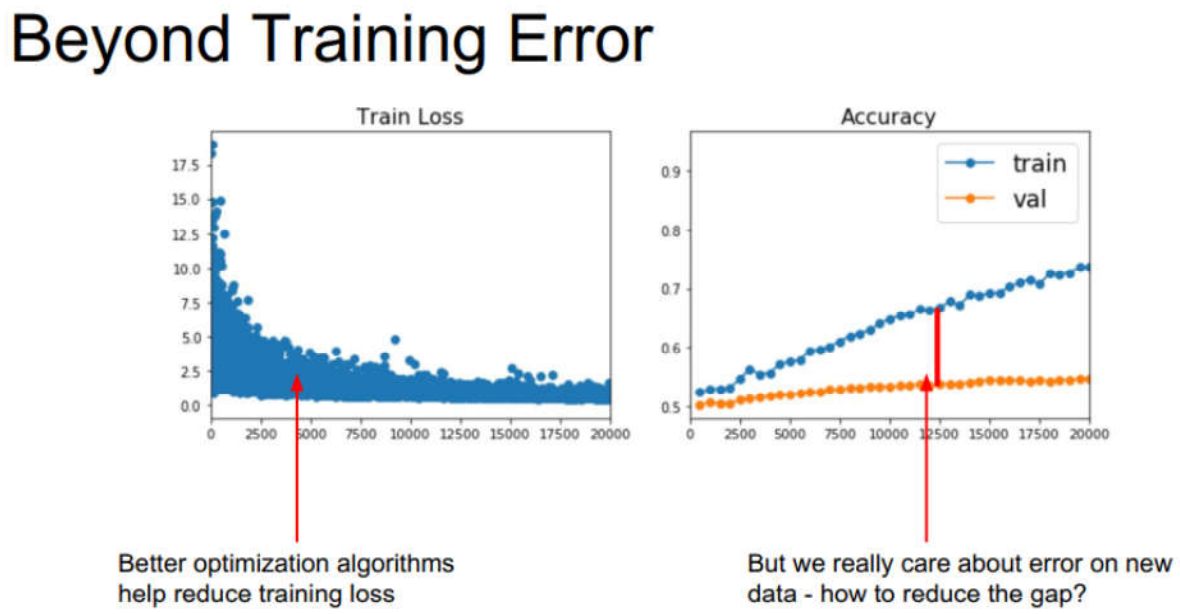
L-BFGS

- **Usually works very well in full batch, deterministic mode** i.e. if you have a single, deterministic f(x) then L-BFGS will probably work very nicely

- **Does not transfer very well to mini-batch setting**. Gives bad results. Adapting L-BFGS to large-scale, stochastic setting is an active area of research.

事实上，人们经常用这个拟牛顿法来替代牛顿法，不是直接地去求完整的Hessian矩阵的逆，而是去逼近这个矩阵的逆，常见的是低阶逼近，某些问题的求解你可能会看到，L B GS就是一个二阶优化器，它是二阶逼近，用Hessian矩阵来逼近。实际中你可能会看到，很多深度学习的问题并不适应这个算法，因为这些二阶逼近的方法对随机的情况处理的不是很好，而且在非凸问题上表现不是很好。

在实践中你可以考虑的是，对很多不同的神经网络问题，Adam已经是一个很好的选择了，然而如果这种情况，如果能够承受整个批次的更新，而且你知道你的问题没有很多随机性，那么L B GS是一个很好的选择，L B GS在训练神经网络时，不是很有用，但是我们会看到，风格迁移会用到，因为很少的随机性。

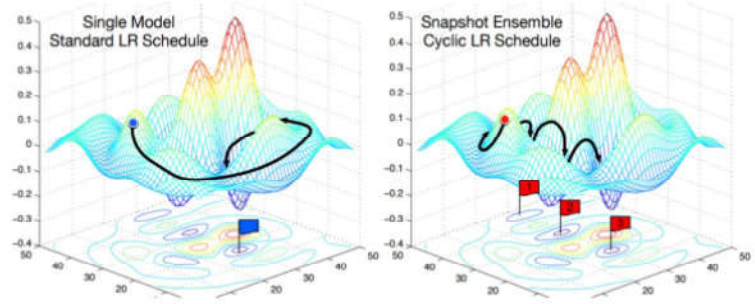


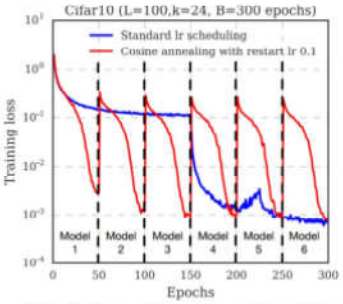
目前我们讲过的所有策略都是在减少训练误差，这些优化算法都在降低训练误差和最小化目标函数。但是我们并不在意训练误差，我们更在意在没见过的数据上的表现，我们很在意减少训练误差和测试误差之间的差距，现在的问题是如果我们已经很擅长优化目标函数，要怎么做来减少训练和测试之间的误差差距，以使得我们的模型在没见过的数据上变现的更好呢？

一个快速、笨拙又简单的方法是模型集成，模型集成在机器学习的很多领域，方法其实很简单，比起使用一个模型，我们选择从不同的随机初始值上训练10个不同的模型，到了测试时，我们就会在10个模型上运行测试数据，然后平均10个模型的预测结果，把这些多个模型加到一起，能够缓解一点过拟合，从而提高一些性能，通常会提高几个百分点，这不是很巨大的提升，但是却是很固定的提升，可以在ImageNet比赛或者其他类似的比赛使用集成技术是很常见的，这样能够得到最大的性能。

Model Ensembles: Tips and Tricks

Instead of training independent models, use multiple snapshots of a single model during training!





Loshchilov and Hutter, "SGDR: Stochastic gradient descent with restarts", arXiv 2016
Huang et al, "Snapshot ensembles: train 1, get M for free", ICLR 2017
Figures copyright Yixuan Li and Geoff Pleiss, 2017. Reproduced with permission.

Cyclic learning rate schedules can make this work even better!

再发挥一下创造力，有时候可以不用独立地训练不同的模型，你可以在训练的过程中，保留多个模型的快照，然后用这些模型来做集成学习，然后在测试阶段，你仍然需要把这些多个快照的预测结果做平均，但是你可以在训练的过程中，收集这些快照。这是ICLR会议上面一篇非常好的论文，讲的是上面的想法的一个高级一点的版本，这里用了一个疯狂的学习率计划，学习率开始时很慢，然后非常快，接着又很慢，再然后又特别快，这个有点疯狂的学习率计划的想法是说，这样的学习率会使得训练过程中，模型会收敛到目标函数不同的区域，但是结果仍然还不错，如果你对这些不同的快照做集成以后，就能够大幅提高最后的性能，虽然你只进行了一次训练。

Model Ensembles: Tips and Tricks

Instead of using actual parameter vector, keep a moving average of the parameter vector and use that at test time (Polyak averaging)

```
while True:
  data_batch = dataset.sample_data_batch()
  loss = network.forward(data_batch)
  dx = network.backward()
  x += - learning_rate * dx
  x_test = 0.995*x_test + 0.005*x # use for test set
```

另外一个你们可能会用到的小技巧是。在训练模型的时候，对不同时刻的每个模型参数求指数衰减平均值，从而得到网络训练中一个比较平滑的集成模型，之后使用这些平滑衰减的平均后的模型参数，而不是截至某一时刻的模型参数，这个方法叫做Polya 平均，有时候能有一点效果，这是你们可能会用到的另一个小技巧，但是并不常见。