收藏版｜最全机器学习优化器Optimizer汇总

浅梦学习笔记 今天

以下文章来源于苏学算法 ， 作者苏学算法

苏学算法
分享算法原理 & 实践经验

虽然起了个“标题党”的题目，不过疏漏之处和不当之处，还请大佬指出

前言

首先，为什么需要优化器（Optimizer）这个东西呢，因为许多问题都是需要“优化”的（当然也包括未来35岁的你我）。人生中，你经历的很多事都可以有一个目标函数（诸如买到房子，找到对象，生个娃，再“鸡”个娃，再买个房子，再帮他找个对象...），那么有了目标，就需要进行求解，也就是优化。如果你的目标很简单，就像一个沙盘大小，那你可以一眼就看出沙盘的最低点（或者最高点）在哪，也就是最优解；但是，如果你的目标函数是一个撒哈拉沙漠，你想找到最优解（最低点）的话，那就没有那么容易了...

如何在复杂的目标函数中找到最优解，衍生出了一系列优化算法，本文则主要以优化器为着笔点展开。

关于优化器（Optimizer），TensorFlow与PyTorch框架中都进行了封装，为了对这些优化器进行更加深入的了解，本文以经典综述文献 An overview of gradient descent optimization algorithms 为线索，结合一些优秀的文章，对目前主流的优化算法进行汇总。

当然，仅从面试的角度，优化器也是面试官最爱考察的内容之一。

统一变量

当前使用的许多优化算法，是对梯度下降法的衍生和优化。

在微积分中，对多元函数的参数 θ 求偏导数，把求得的各个参数的导数以向量的形式写出来就是**梯度**。梯度就是函数变化最快的地方。梯度下降是迭代法的一种，在求解机器学习算法的模型参数 θ 时，即无约束问题时，梯度下降是最常采用的方法之一。

顾名思义，**梯度下降法的计算过程就是沿梯度下降的方向求解极小值，也可以沿梯度上升方向求解最大值**。

为了便于后续讨论，在此统一定义变量。假设，待优化的模型参数为 θ，目标函数（损失函数）为 $J(\theta)$，学习率为 η，迭代的周期 epoch 为 t，因此可以得到下面的式子

损失函数 $J(\theta)$ 关于当前参数 θ 的梯度：$g_t = \nabla_\theta J(\theta)$

一、梯度下降法 (Gradient Descent)

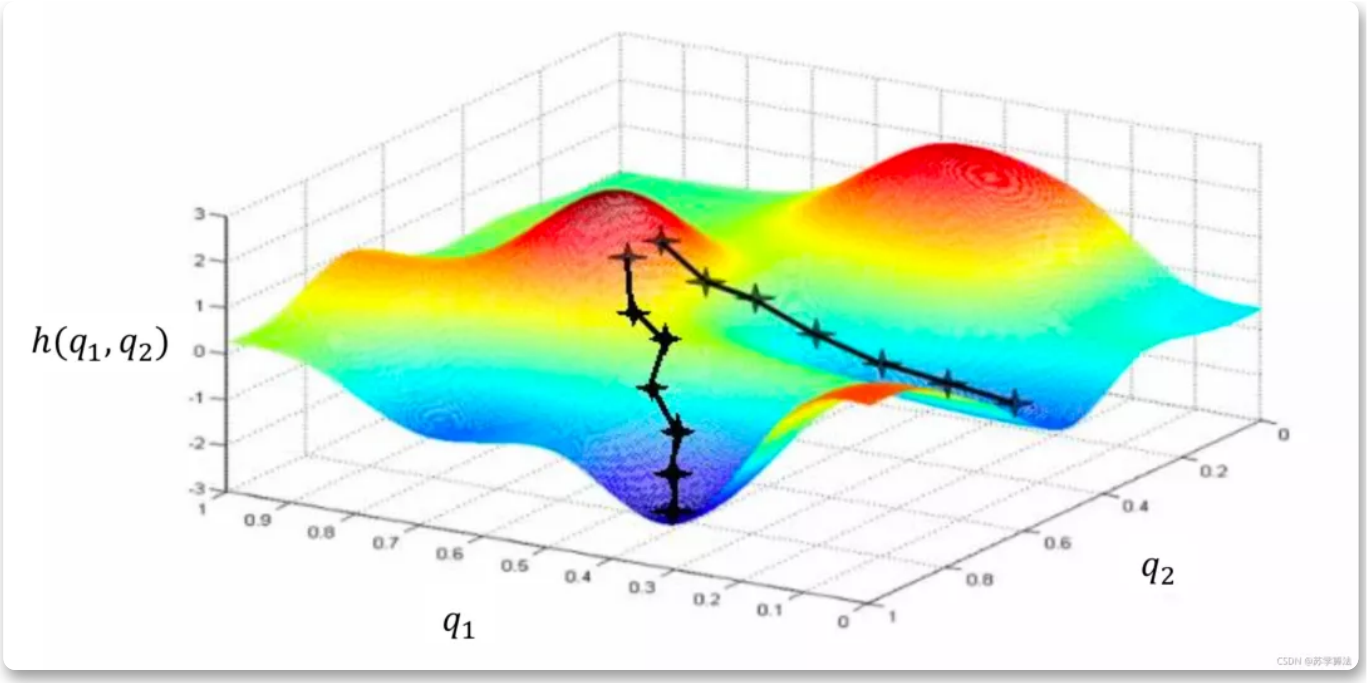
梯度下降法是最基本的优化算法之一。梯度下降法就是沿着梯度下降最快的方向求极小值。

> 简单理解就是，上面提到的场景中，你要在撒哈拉沙漠中找最低点，你根本不知道最低点在哪，但是在你所站的位置上，总有一个方向的坡度最陡，你沿着这个方向往下滑是有可能到最低点的。

梯度下降法参数更新为：

$$\theta_{t+1} = \theta_t - \eta \cdot \nabla_\theta J(\theta) = \theta_t - \eta \cdot g_t$$

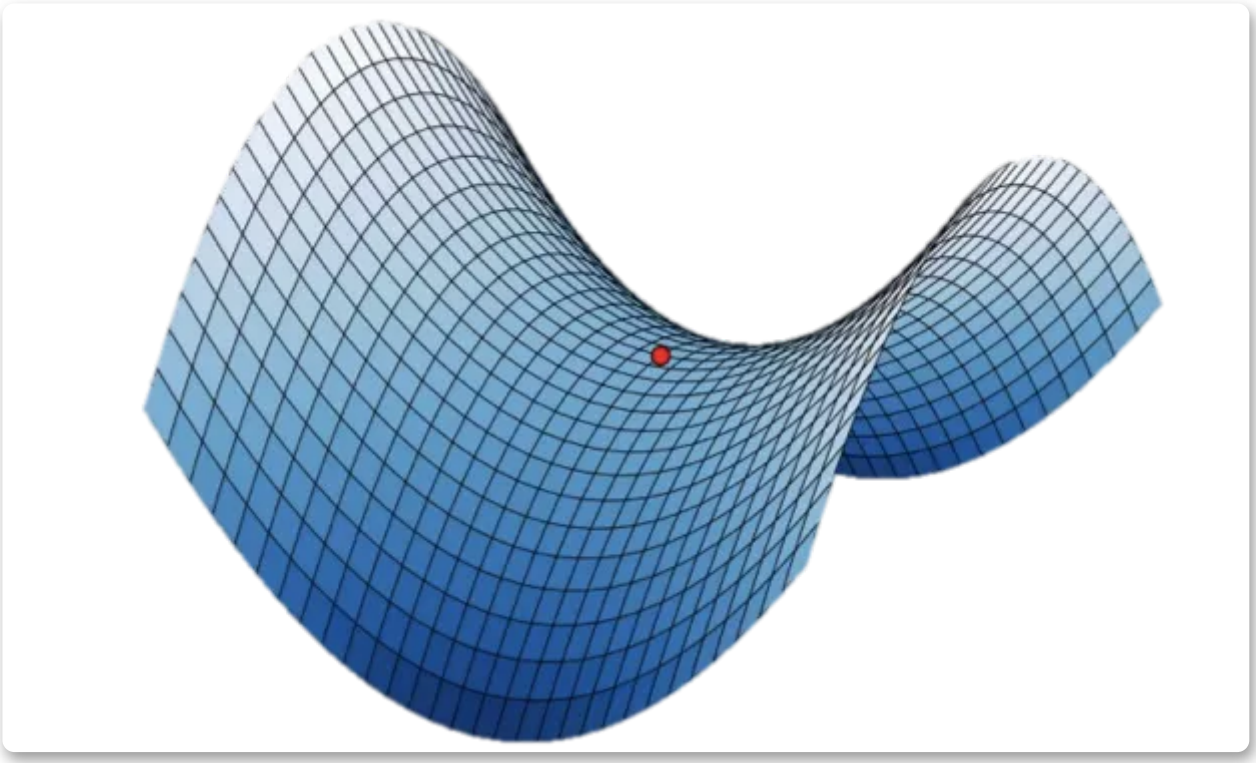
梯度下降算法中，沿着梯度的方向不断减小模型参数，从而最小化损失函数。基本策略可以理解为”**在你目光所及的范围内，不断寻找最陡最快的路径下山**“



标准的梯度下降主要有两个缺点：

训练速度慢：每走一步都要计算调整下一步的方向，下山的速度变慢。在应用于大型数据集中，每输入一个样本都要更新一次参数，且每次迭代都要遍历所有的样本。会使得训练过程及其缓慢，需要花费很长时间才能得到收敛解。

容易陷入局部最优解：由于是在有限视距内寻找下山的方向。当陷入平坦的洼地，会误以为到达了山地的最低点，从而不会继续往下走。所谓的局部最优解就是鞍点。落入鞍点，梯度为0，使得模型参数不在继续更新。“鞍部” 如下图所示



因此，真正在使用时，主要是经过改进的以下三类方法，区别在于**每次参数更新时计算的样本数据量不同**：

- 批量梯度下降法(BGD, Batch Gradient Descent)
- 随机梯度下降法(SGD, Stochastic Gradient Descent)
- 小批量梯度下降法(Mini-batch Gradient Descent)

1. 批量梯度下降法 (Batch Gradient Descent, BGD)

这里的“批量”其实指的就是整个训练集的数据，后面 MBGD 中的 mini-batch 才是真正意义上的 “一批”

BGD相对于标准GD进行了改进，所谓“批量”，也就是不再像标准GD一样，对每个样本输入都进行参数更新，而是针对**所有的数据**输入进行参数更新。我们假设所有的训练样本总数为 n，样本为 $\{(x_1, y_1), \cdots, (x_n, y_n)\}$，模型参数为 θ，在对第 i 个样本 (x_i, y_i) 上的损失函数关于参数的梯度为 $\nabla_\theta J_i(\theta, x_i, y_i)$，则使用BGD更新参数的式子为

$$\theta_{t+1} = \theta_t - \eta \cdot \frac{1}{n} \cdot \sum_{i=1}^{n} \nabla_{\theta_t} J_i(\theta_t, x_i, y_i)$$

优点：

由于**每一步迭代使用了全部样本**，每次下降的方向为总体的平均梯度，因此损失函数收敛过程会比较稳定。对于凸函数可以收敛到全局最小值，对于非凸函数可以收敛到局部最小值。

缺点：

每一步更新中，都要利用全部样本计算梯度，计算起来非常慢，遇到很大量的数据集也会非常棘手，而且不能投入新数据实时更新模型。

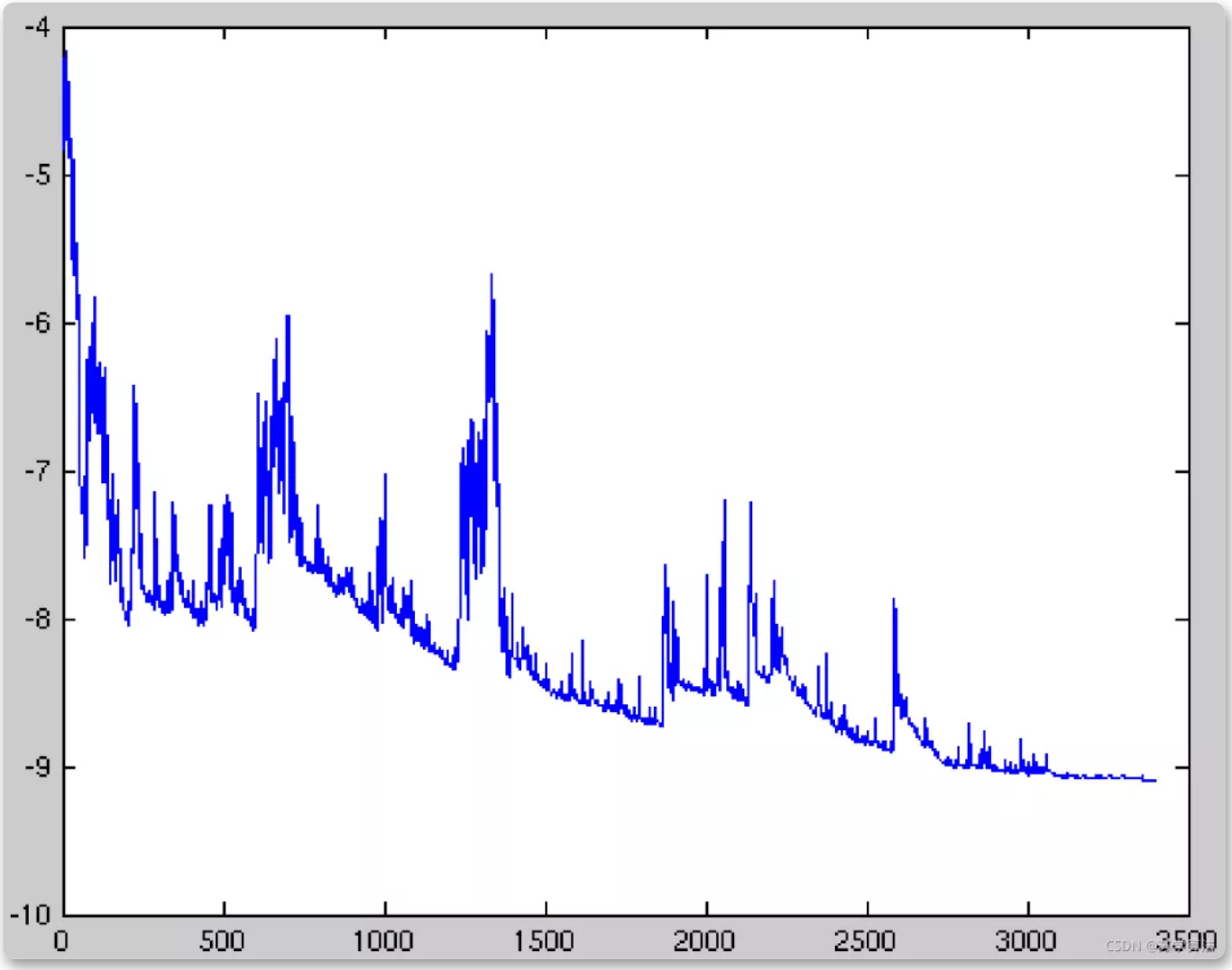
2. 随机梯度下降（Stochastic Gradient Descent, SGD)

随机梯度下降法，不像BGD每一次参数更新，需要计算整个数据样本集的梯度，而是每次参数更新时，仅仅选取一个样本 (x_i, y_i) 计算其梯度，参数更新公式为

$$\theta_{t+1} = \theta_t - \eta \cdot \nabla_\theta J_i(\theta, x_i, y_i)$$

公式看起来和上面标准GD一样，但不同点在于，这里的样本是从所有样本中随机选取一个，而标准GD是所有的样本都输入进行计算。

可以看到BGD和SGD是两个极端，SGD由于每次参数更新仅仅需要计算一个样本的梯度，训练速度很快，即使在样本量很大的情况下，可能只需要其中一部分样本就能迭代到最优解，由于每次迭代并不是都向着整体最优化方向，导致梯度下降的波动非常大（如下图），更容易从一个局部最优跳到另一个局部最优，准确度下降。



论文中提到，当缓慢降低学习率时，SGD会显示与BGD相同的收敛行为，几乎一定会收敛到局部（非凸优化）或全局最小值（凸优化）

优点：

- 由于每次迭代只使用了一个样本计算梯度，训练速度快，包含一定随机性，但是从期望来看，每次计算的梯度基本是正确的导数的。虽然看起来SGD波动非常大，会走很多弯路，但是对梯度的要求很低（计算梯度快），而且对于引入噪声，大量的理论和实践工作证明，只要噪声不是特别大，SGD都能很好地收敛。

- 应用大型数据集时，训练速度很快。比如每次从百万数据样本中，取几百个数据点，算一个SGD梯度，更新一下模型参数。相比于标准梯度下降法的遍历全部样本，每输入一个样本更新一次参数，要快得多

缺点：

- 更新频繁，带有随机性，会造成损失函数在收敛过程中严重震荡。SGD没能单独克服局部最优解的问题（主要）
- SGD在随机选择梯度的同时会引入噪声，使得权值更新的方向不一定正确（次要）

3. 小批量梯度下降法（Mini-batch Gradient Descent, MBGD or SGD）

小批量梯度下降法就是结合BGD和SGD的折中，对于含有 n 个训练样本的数据集，每次参数更新，选择一个大小为 m（$m<n$）的="" mini-batch="" 数据样本计算其梯度，其参数更新公式如下<="" p="">

$$\theta_{t+1} = \theta_t - \eta \sum_{i=x}^{i=x+m-1} \nabla_\theta J_i(\theta, x_i, y_i)$$

小批量梯度下降法即保证了训练的速度，又能保证最后收敛的准确率，目前的SGD默认是小批量梯度下降算法。常用的小批量尺寸范围在50到256之间，但可能因不同的应用而异。

优点：

可以降低参数更新时的方差，收敛更稳定，另一方面可以充分地利用深度学习库中高度优化的矩阵操作来进行更有效的梯度计算

缺点：

- Mini-batch gradient descent 不能保证很好的收敛性，learning rate 如果选择的太小，收敛速度会很慢，如果太大，loss function 就会在极小值处不停地震荡甚至偏离（有一种措施是先设定大一点的学习率，当两次迭代之间的变化低于某个阈值后，就减小 learning rate，不过这个阈值的设定需要提前写好，这样的话就不能够适应数据集的特点）。对于非凸函数，还要避免陷于局部极小值处，或者鞍点处，因为鞍点所有维度的梯度都接近于0，SGD 很容易被困在这里（会在鞍点或者局部最小点震荡跳动，因为在此点处，如果是BGD的训练集全集带入，则优化会停止不动，如果是mini-batch或者SGD，每次找到的梯度都是不同的，就会发生震荡，来回跳动）。

- SGD对所有参数更新时应用同样的 learning rate，如果我们的数据是稀疏的，我们更希望对出现频率低的特征进行大一点的更新， 且learning rate会随着更新的次数逐渐变小。

在梯度下降公式 $\theta_{t+1} = \theta_t - \eta g_t$ 中，能改进的两个点，一是学习率，二是梯度。其分别衍生出自适应（adaptive）学习率方法与动量（momentum）方法。后面讲展开介绍

二、动量优化法（Momentum）

动量优化法引入了物理之中的概念。动量 $p = mv$，当一个小球从山顶滚下，速度越来越快，动量越来越大，开始加速梯度下降，当跨越了山谷，滚到对面的坡上时，速度减小，动量减小。

带动量的小球不仅可以加速梯度；还可以借着积累的动量，冲过小的山坡，以避免落入局部最优点。

1. Momentum

梯度下降法容易被困在局部最小的沟壑处来回震荡，可能存在曲面的另一个方向有更小的值；有时候梯度下降法收敛速度还是很慢。动量法就是为了解决这两个问题提出的

momentum算法思想：参数更新时在一定程度上保留之前更新的方向，同时又利用当前batch的梯度微调最终的更新方向，简言之就是通过积累之前的动量来(previous_sum_of_gradient) 加速当前的梯度。

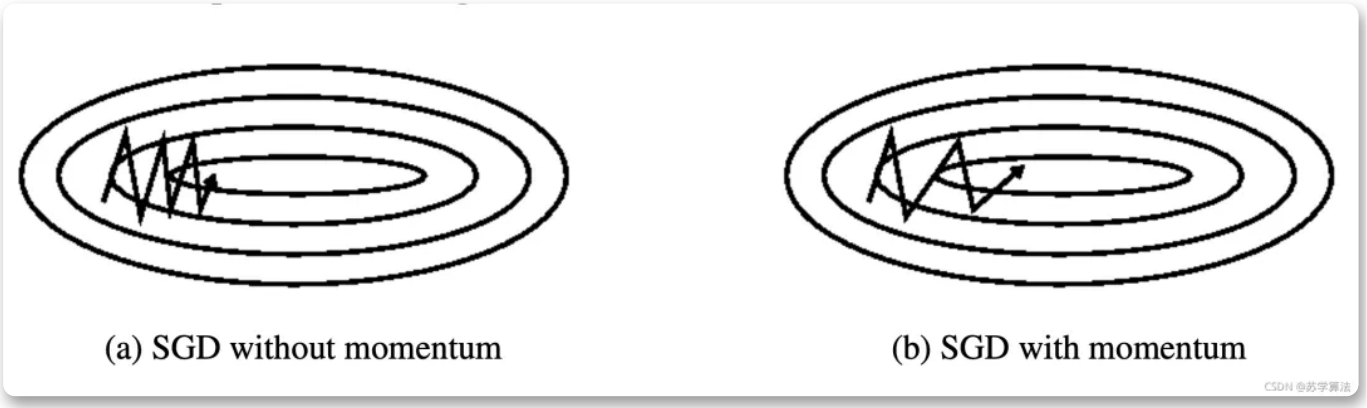
SGD只使用了当步参数的梯度，随机性较大。如果将历次迭代的梯度按比例融合，可能更加稳定、更有利于跳出局部最优。

假设 m_t 表示 t 时刻的动量，γ 表示动量因子，通常取值 0.9 或者近似值，在SGD的基础上增加动量，则参数更新公式为

$$m_{t+1} = \gamma \cdot m_t + \eta \cdot \nabla_\theta J(\theta)$$
$$\theta_{t+1} = \theta_t - m_{t+1}$$

一阶动量 m_t 是各个时刻梯度方向的指数移动平均值，约等于最近 $\frac{1}{(1-\gamma)}$ 个时刻的梯度向量和的平均值。也就是说，t 时刻的下降方向，不仅由当前点的梯度方向决定，而且由此前累积的下降方向决定。

动量因子 γ 的经验值为0.9，这就意味着下降方向主要是此前累积的下降方向，并略微偏向当前时刻的下降方向。在梯度方向改变时，momentum能够降低参数更新速度，从而减少震荡，在梯度方向相同时，momentum可以加速参数更新， 从而加速收敛，如下图



动量主要解决SGD的两个问题：

- 随机梯度的方法（引入的噪声）
- Hessian矩阵病态问题（可以理解为SGD在收敛过程中和正确梯度相比来回摆动比较大的问题）

优点：

前后梯度一致的时候能够加速学习；前后梯度不一致的时候能够抑制震荡，越过局部极小值（加速收敛，减小震荡）

缺点：

增加了一个超参数

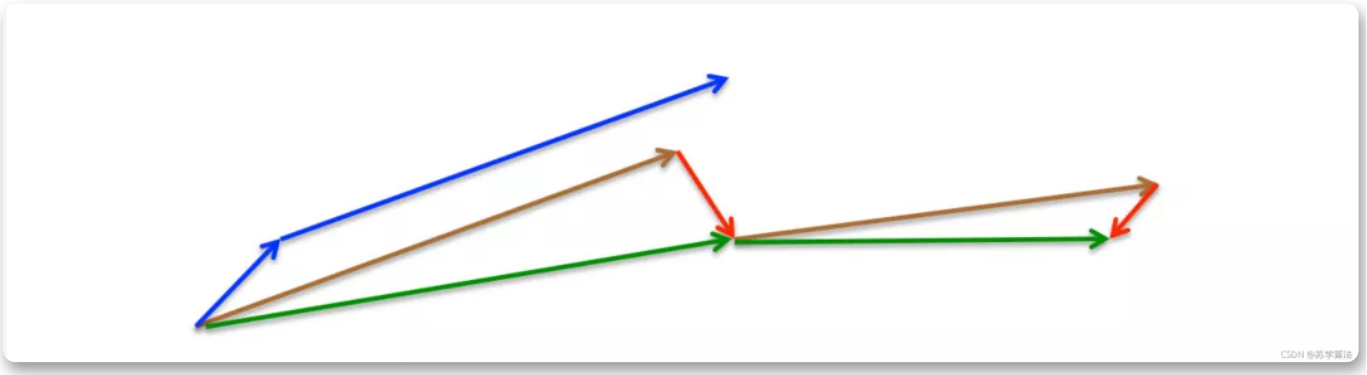
2. NAG（Nesterov accelerated gradient）

为了增强探索性，进一步引入了 nesterov 动量。momentum保留了上一时刻的梯度 $\nabla_\theta J(\theta)$，对其没有进行任何改变，NAG是momentum的改进版，在梯度更新时做了一个

矫正，具体做法是在当前的梯度 $\nabla_\theta J(\theta)$ 上添加上一个时刻的动量 $\gamma \cdot m_t$，梯度改变为 $\nabla_\theta J(\theta - \gamma \cdot m_t)$

$$m_{t+1} = \gamma \cdot m_t + \eta \cdot \nabla_\theta J(\theta - \gamma \cdot m_t)$$
$$\theta_{t+1} = \theta_t - m_{t+1}$$

加上nesterov项后，梯度在大的跳跃后，进行计算对当前梯度进行校正。 下图是momentum和nesterrov的对比表述图



momentum首先计算一个梯度(短的蓝色向量)，然后在加速更新梯度的方向进行一个大的跳跃(长的蓝色向量)，nesterov项首先在之前加速的梯度方向进行一个大的跳跃(棕色向量)，计算梯度然后进行校正(绿色向量)

Nesterov动量梯度的计算在模型参数施加当前速度之后，因此可以理解为往标准动量中添加了一个校正因子。在凸批量梯度的情况下，Nesterov动量将额外误差收敛率从 $O(1/k)$ (k步后) 改进到 $O(1/k^2)$ ，然而，在随机梯度情况下，Nesterov动量对收敛率的作用却不是很大。

> Momentum和Nexterov都是为了使梯度更新更灵活。但是人工设计的学习率总是有些生硬，下面介绍几种自适应学习率的方法。

三、自适应学习率优化算法

传统的优化算法要么将学习率设置为常数要么根据训练次数调节学习率。往往忽视了学习率其他变化的可能性。然而，学习率对模型的性能有着显著的影响，因此需要采取一些策略来想办法更新学习率，从而提高训练速度

> 先来看一下使用统一的全局学习率的缺点可能出现的问题

- 对于某些参数，通过算法已经优化到了极小值附近，但是有的参数仍然有着很大的梯度。
- 如果学习率太小，则梯度很大的参数会有一个很慢的收敛速度； 如果学习率太大，则已经优化得差不多的参数可能会出现不稳定的情况。 解决方案：
- 对每个参与训练的参数设置不同的学习率，在整个学习过程中通过一些算法自动适应这些参数的学习率。 如果损失与某一指定参数的偏导的符号相同，那么学习率应该增加； 如果损失与该参数的偏导的符号不同，那么学习率应该减小。

自适应学习率算法主要有：AdaGrad算法，RMSProp算法，Adam算法以及AdaDelta算法等

1. AdaGrad ((Adaptive Gradient))

Adagrad其实是对学习率进行了一个约束，对于经常更新的参数，我们已经积累了大量关于它的知识，不希望被单个样本影响太大，希望学习速率慢一些；对于偶尔更新的参数，我们了解的信息太少，希望能从每个偶然出现的样本（稀疏特征的样本）身上多学一些，即学习速率大一些。而该方法中开始使用二阶动量，才意味着“自适应学习率”优化算法时代的到来。

AdaGrad 算法，独立地适应所有模型参数的学习率，缩放每个参数反比于其**所有梯度历史平均值总和的平方根**。

- 具有损失函数最大梯度的参数相应地有个快速下降的学习率
- 而具有小梯度的参数在学习率上有相对较小的下降。

其梯度更新公式为

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{t,i}$$

其中，g_t 为 t 时刻参数 θ_i 的梯度

$$g_{t,i} = \nabla_{\theta_t} J(\theta_{t,i})$$

如果是普通的 SGD，那么每一个时刻的梯度更新公式为

$$\theta_{t+1,i} = \theta_{t,i} - \eta \cdot g_{t,i}$$

但是 AdaGrad 的更新公式中，矫正的学习率 $\frac{\eta}{\sqrt{G_{t,ii}+\epsilon}}$ 也随着 t 和 i 而变化，也就是所谓的“自适应”。（其中分母加了一个小的平滑项 ϵ 是为了防止分母为0）

上式中的 G_t 为对角矩阵，(i,i) 元素就是到 t 时刻为止，参数 θ_i 的**累积梯度平方和**，也就是“二阶动量”。从上述公式可以看出，$\sqrt{G_{t,ii}+\epsilon}$ 是恒大于 0 的，而且参数更新越频繁，二阶动量就越大，学习率 $\frac{\eta}{\sqrt{G_{t,ii}+\epsilon}}$ 就越小，所以在稀疏的数据场景下表现比较好

优点：

自适应的学习率，无需人工调节

缺点：

- 仍需要手工设置一个全局学习率 η，如果 η 设置过大的话，会使 regularizer 过于敏感，对梯度的调节太大
- 中后期，**分母上梯度累加的平方和会越来越大，使得参数更新量趋近于0**，使得训练提前结束，无法学习

2. Adadelta

由于AdaGrad调整学习率变化过于激进，我们考虑一个改变二阶动量计算方法的策略：不累积全部历史梯度，而只关注过去一段时间窗口的下降梯度，即Adadelta只累加固定大小的项，并且也不直接存储这些项，仅仅是近似计算对应的平均值（指数移动平均值），这就避免了二阶动量持续累积、导致训练过程提前结束的问题了，参数更新公式如下

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E\left[g^2\right]_t + \epsilon}} g_t$$

与 Adagrad 相比，就是分母的 G 变成了过去的梯度平方的衰减平均值（指数衰减平均值）

其中，E 的计算公式如下，t 时刻的值依赖于前一时刻的平均和当前的梯度：

$$E\left[g^2\right]_t = \gamma E\left[g^2\right]_{t-1} + (1-\gamma) g_t^2$$

这个分母相当于梯度的均方根（root mean squared，RMS），所以可以用 RMS 简写成

$$\theta_{t+1} = \theta_t - \frac{\eta}{RMS[g]_t} g_t$$

此外，还可以将学习率 η 换成 $RMS[\Delta \theta]$，这样的话，就**不需要提前设定学习率**了

$$\theta_{t+1} = \theta_t - \frac{RMS[\Delta \theta]_{t-1}}{RMS[g]_t} g_t$$

优点：

- 不依赖全局learning rate
- 训练初中期，加速效果不错，很快

缺点： 训练后期，反复在局部最小值附近抖动

3. RMSprop

RMSprop 和 Adadelta 都是为了解决 Adagrad 学习率急剧下降问题的，但是RMSProp算法修改了AdaGrad的梯度平方和累加为**指数加权的移动平均**，使得其在非凸设定下效果更好。

指数加权平均，旨在消除梯度下降中的摆动，与Momentum的效果一样，某一维度的导数比较大，则指数加权平均就大，某一维度的导数比较小，则其指数加权平均就小，这样就保证了各维度导数都在一个量级，进而减少了摆动。

另外，**指数衰减平均的方式可以淡化遥远过去的历史对当前步骤参数更新量的影响**，衰减率表明的是只是最近的梯度平方有意义，而很久以前的梯度基本上会被遗忘

并且RMSprop允许使用一个更大的学习率 η

设定参数：全局初始学习率 $\eta = 0.001$，decay_rate $\rho = 0.9$，极小常量 $\epsilon = 10e-6$，参数更新如下：

$$E\left[g^2\right]_t = 0.9E\left[g^2\right]_{t-1} + 0.1g_t^2$$
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E\left[g^2\right]_t + \epsilon}} g_t$$

优点：

- RMSprop算是Adagrad的一种发展，和Adadelta的变体，效果趋于二者之间
- 适合处理非平稳目标(包括季节性和周期性)——对于RNN效果很好

缺点：

- 其实RMSprop依然依赖于全局学习率 η

4. Adam (Adaptive Moment Estimation)

Adam 结合了前面方法的一阶动量和二阶动量，相当于 Ada + Momentum，SGD-M和NAG在SGD基础上增加了一阶动量，AdaGrad和AdaDelta在SGD基础上增加了二阶动量。

Adam 除了像 Adadelta 和 RMSprop 一样存储了过去梯度的平方 v_t 的指数衰减平均值 ，也像 momentum 一样保持了过去梯度 m_t 的指数衰减平均值：

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$$

如果 m_t 和 v_t 被初始化为 0向量，那么它们就会向0偏置，所以做了**偏差校正** ，通过计算偏差校正后的m_t 和 v_t来抵消这些偏差

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$$
$$\hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

最终的更新公式为

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$$

通常情况下，默认值为 $\beta_1 = 0.9$，$\beta_2 = 0.999$，$\beta_1 = 0.002$

优点：

- Adam梯度经过偏置校正后，每一次迭代学习率都有一个固定范围，使得参数比较平稳。
- 结合了Adagrad善于处理稀疏梯度和RMSprop善于处理非平稳目标的优点
- 为不同的参数计算不同的自适应学习率
- 也适用于大多非凸优化问题——适用于大数据集和高维空间。

缺点：

Adam 使用动量的滑动平均，可能会随着训练数据变化而抖动比较剧烈，在online场景可能波动较大，在广告场景往往效果不如 AdaGrad

5. Nadam

其实如果说要集成所有方法的优点于一身的话，Nadam应该就是了，Adam遗漏了啥？没错，就是Nesterov项，我们在Adam的基础上，加上Nesterov项就是Nadam了，参数更新公式如下：

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon}\left(\beta_1 \hat{m}_t + \frac{(1-\beta_1)g_t}{1-\beta_1^t}\right)$$

其中

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)g_t$$
$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$$

可以看出，Nadam对学习率有更强的约束，同时对梯度的更新也有更直接的影响。一般而言，在使用带动量的RMSprop或Adam的问题上，使用Nadam可以取得更好的结果。

6. AdamW

Adam有很多的优点，但是在很多数据集上的最好效果还是用SGD with Momentum细调出来的。可见Adam的泛化性并不如SGD with Momentum。Decoupled Weight Decay Regularization 提出其中一个重要原因就是 Adam中L2正则化项并不像在SGD中那么有效

- L2正则和Weight Decay在Adam这种自适应学习率算法中并不等价，只有在标准SGD的情况下，可以将L2正则和Weight Decay看做一样。特别是，当与自适应梯度相结合时，L2正则化导致具有较大历史参数和/或梯度幅度的权重比使用权重衰减时更小。

- 使用Adam优化带L2正则的损失并不有效，如果引入L2正则化项，在计算梯度的时候会加上正则项求梯度的结果。正常的权重衰减是对所有的权重都采用相同的系数进行更新，本身比较大的一些权重对应的梯度也会比较大，惩罚也越大。但由于Adam计算步骤中减去项会有除以梯度平方的累积，使得梯度大的减去项偏小，从而具有大梯度的权重不会像解耦权重衰减那样得到正则化。 这导致自适应梯度算法的L2和解耦权重衰减正则化的不等价。

而在常见的深度学习库中只提供了L2正则，并没有提供权重衰减的实现。这可能就是导致Adam跑出来的很多效果相对SGD with Momentum有偏差的一个原因

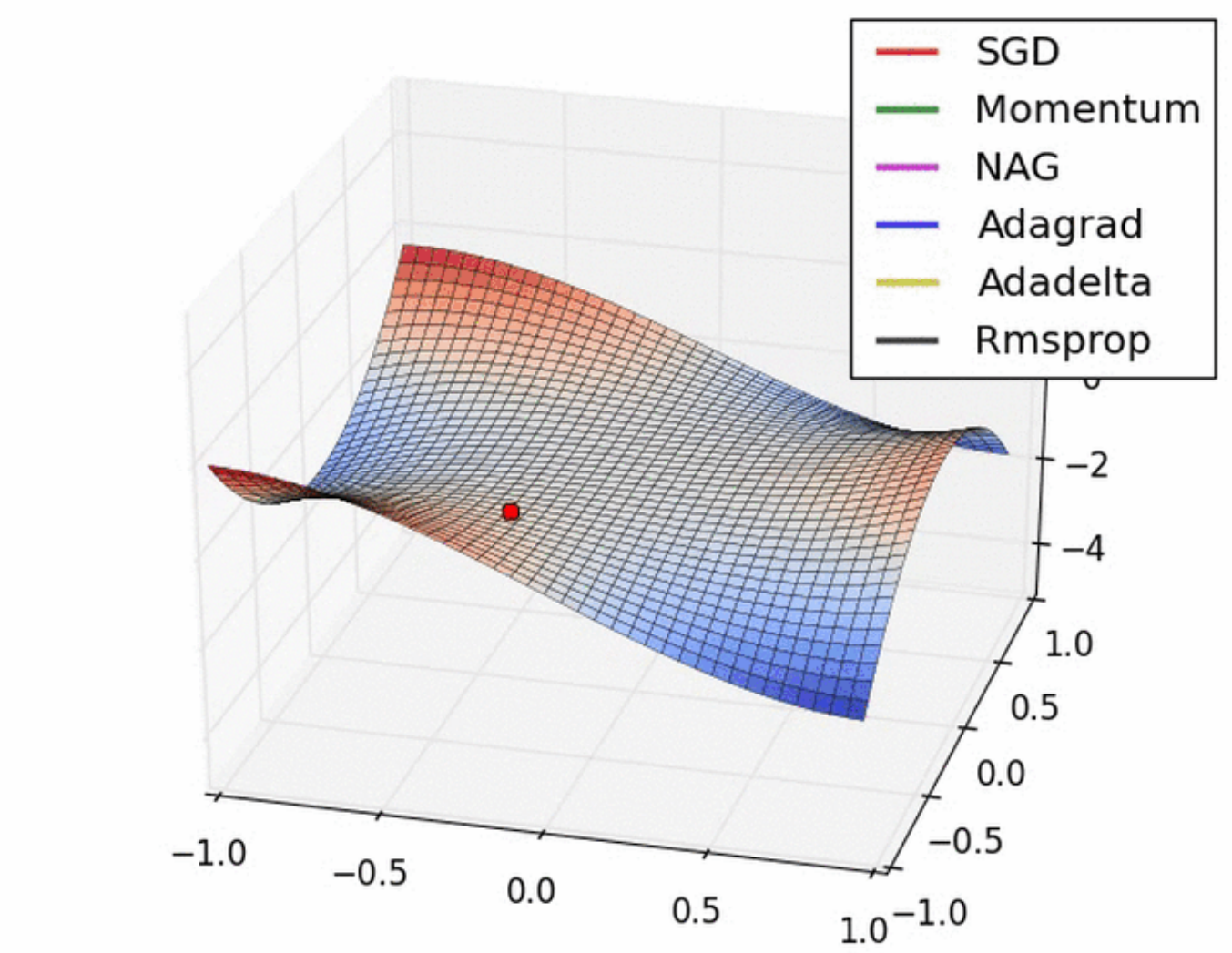
AdamW 使用了严谨的 weight decay（非L2正则），即权重衰减不参与一、二动量计算，只在最后的更新公式中使用。其更新公式如下：

$$
\begin{aligned}
m_t &= \beta_1 * m_{t-1} + (1-\beta_1) * g_t \\
v_t &= \beta_2 * v_{t-1} + (1-\beta_2) * g_t^2 \\
\hat{m}_t &= m_t / \left(1-\beta_1^t\right) \\
\hat{v}_t &= v_t / \left(1-\beta_2^t\right) \\
\theta_t &= \theta_{t-1} - \alpha * \left(\frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} + \lambda * \theta_{t-1} \right)
\end{aligned}
$$

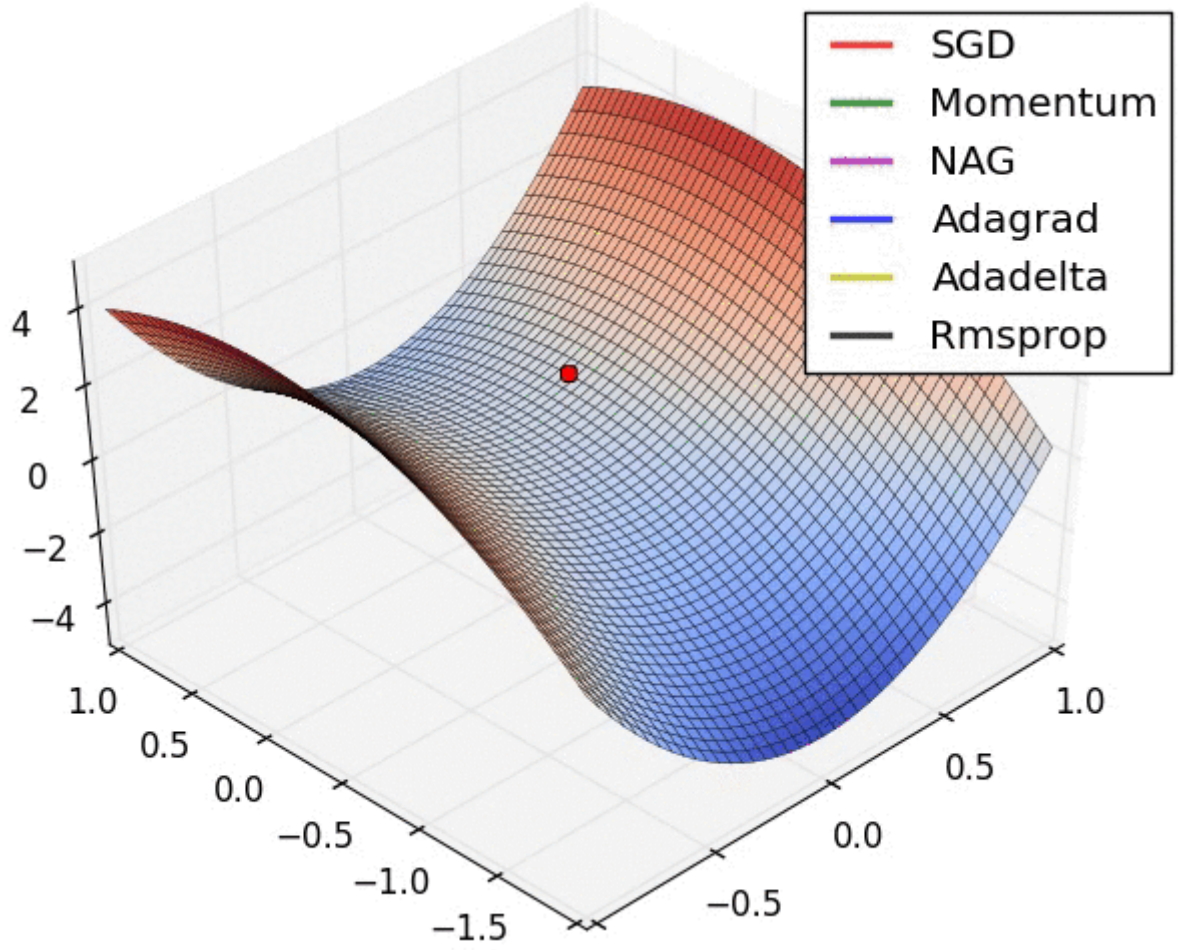
四、优化器对比 & 总结

1. 收敛直观对比

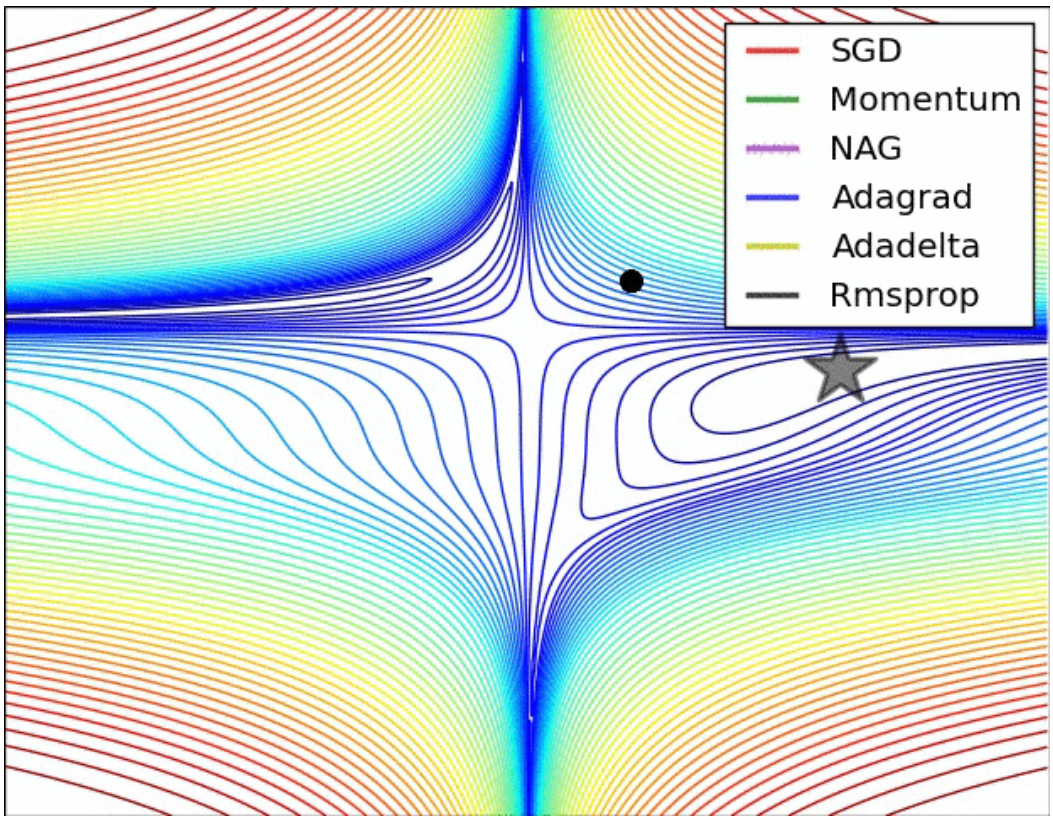
下图描述了在一个曲面上，6种优化器的表现



下图在一个存在鞍点的曲面，比较6中优化器的性能表现



下图图比较了6种优化器收敛到目标点（五角星）的运行过程



2. 总结 & tricks

目前，最流行并且使用很高的优化器（算法）包括SGD、具有动量的SGD、RMSprop、具有动量的RMSProp、AdaDelta和Adam。在实际应用中，选择哪种优化器应结合具体问题。在充分理解数据的基础上，依然需要根据数据特性、算法特性进行充分的调参实验，找到最优解。

- **首先，各大算法孰优孰劣并无定论**：如果是刚入门，优先考虑SGD+NesteroV Momentum或者Adam.（Standford 231n：The two recommended updates to use are either SGD+NesteroV Momentum or Adam）

- **选择你熟悉的算法**：这样你可以更加熟练地利用你的经验进行调参。

- **充分了解你的数据**：如果模型是非常稀疏的，那么优先考虑自适应学习率的算法。如果在意更快的收敛，并且需要训练较深较复杂的网络时，推荐使用自适应学习率的优化方法。

- **根据你的需求来选择**：在模型设计实验过程中，要快速验证新模型的效果，可以先用Adam进行快速实验优化；在模型上线或者结果发布前，可以用精调的SGD进行模型的极致优化。

- **先用小数据集进行实验**：有论文研究指出，随机梯度下降算法的收敛速度和数据集的大小的关系不大。因此可以先用一个具有代表性的小数据集进行实验，测试一下最好的优化算法，并通过参数搜索来寻找最优的训练参数。

- **考虑不同算法的组合**：先用Adam进行快速下降，而后再换到SGD进行充分的调优。切换策略可以参考本文介绍的方法。

- **数据集一定要充分的打散（shuffle）**：这样在使用自适应学习率算法的时候，可以避免某些特征集中出现，而导致的有时学习过度、有时学习不足，使得下降方向出现偏差的问题。

- **训练过程中持续监控**：监控训练数据和验证数据上的目标函数值以及精度或者AUC等指标的变化情况。对训练数据的监控是要保证模型进行了充分的训练——下降方向正确，且学习率足够高；对验证数据的监控是为了避免出现过拟合。

- **制定一个合适的学习率衰减策略**：可以使用定期衰减策略，比如每过多少个epoch就衰减一次；或者利用精度或者AUC等性能指标来监控，当测试集上的指标不变或者下跌时，就降低学习率。

参考文献：

1. An overview of gradient descent optimization algorithms(https://arxiv.org/pdf/1609.04747.pdf)
2. 收藏 | 各种 Optimizer 梯度下降优化算法回顾和总结(https://mp.weixin.qq.com/s/yfPE4-czmWE9TKXCYvcTSw)
3. 优化方法总结以及Adam存在的问题(SGD, Momentum, AdaDelta, Adam, AdamW, LazyAdam)(https://blog.csdn.net/yinyu19950811/article/details/90476956)
4. 优化算法Optimizer比较和总结(https://zhuanlan.zhihu.com/p/55150256)
5. Adam,AdamW,LAMB优化器原理与代码(https://blog.csdn.net/weixin_41089007/article/details/107007221)

推荐阅读

- 学习交流小组精彩内容摘要 No.68
- 如何从 0 到 1 构建个性化推荐？
- 推荐生态中的bias和debias
- 推荐系统应该如何保障推荐的多样性？

想了解更多关于**推荐系统与NLP**的内容，欢迎扫码关注公众号**浅梦的学习笔记**。回复**加群**可以加入我们的交流群一起学习！

浅梦学习笔记
分享算法技术与实践经验
174篇原创内容

公众号

码字很辛苦，有收获的话就请分享、点赞、在看三连吧

阅读原文