优化算法小结
--SGD, Adam, Adagrad

Outline

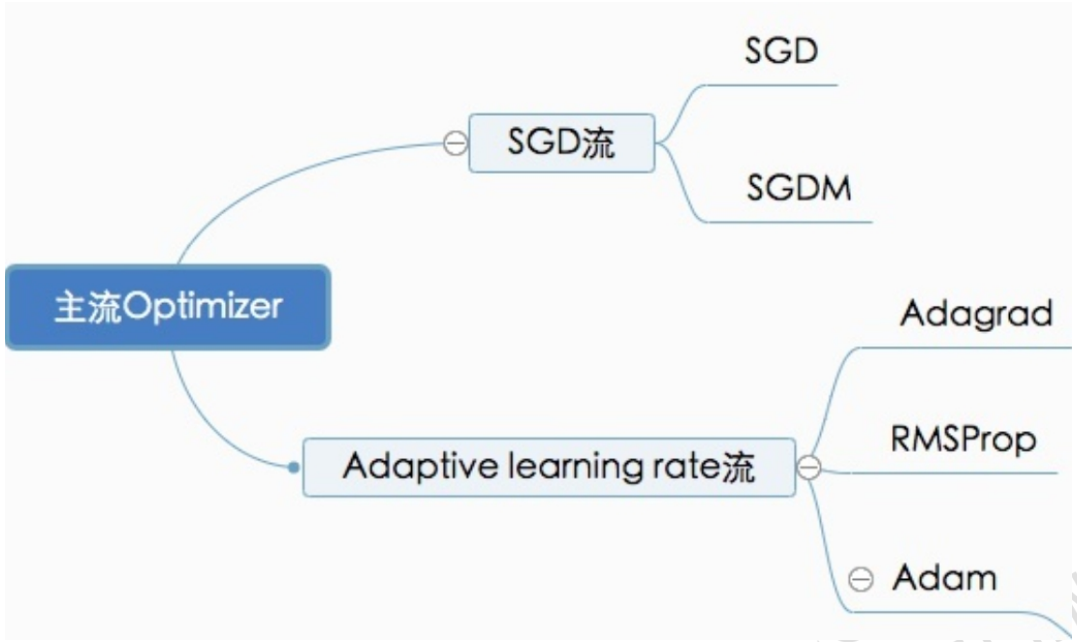
- 优化算法框架
- SGD
- Adam
- Adagrad

优化算法框架

一般用一个通用框架来表述优化算法

有如下定义：
1）待优化的参数 θ
2）目标函数 J(θ)
3）学习率 α

有如下过程（每次迭代）：
1）计算目标关于此时参数的梯度 ∇θ(J(θ))
2）计算历史梯度的一阶动量和二阶动量
3）计算下降梯度 g
4）根据梯度进行迭代 θ = θ − g



优化算法目前有固定学习率：
1. 固定学习率
2.自适应学习率

两种，差别也就体现在过程的第1和第2步
固定学习率优化算法有：BGD、SGD、Mini-Batch GD
自适应学习率优化算法有：Adam、AdaGrad、AdaDelta

现在用的最多的算法是SGD和Adam，两者各有好坏。SGD能够到达全局最优解，而且训练的最佳精度也要高于其他优化算法，但它对学习率的调节要求非常严格，而且容易停在鞍点；Adam很容易的跳过鞍点，而且不需要人为的干预学习率的调节，但是它很容易在局部最小值处震荡，存在在特殊的数据集下出现学习率突然上升，造成不收敛的情况。

算法	优点	缺点	适用情况
BGD	目标函数为凸函数时，可以找到全局最优值	收敛速度慢，需要用到全部数据，内存消耗大	不适用于大数据集，不能在线更新模型
SGD	避免冗余数据的干扰，收敛速度加快，能够在线学习	更新值的方差较大，收敛过程会产生波动，可能落入极小值（卡在鞍点），选择合适的学习率比较困难（需要不断减小学习率）	适用于需要在线更新的模型，适用于大规模训练样本情况
Momentum	能够在相关方向加速SGD，抑制振荡，从而加快收敛	需要人工设定学习率	适用于有可靠的初始化参数
Adagrad	实现学习率的自动更改	仍依赖于人工设置一个全局学习率，学习率设置过大，对梯度的调节太大。中后期，梯度接近于0，使得训练提前结束	需要快速收敛，训练复杂网络时；适合处理稀疏梯度[1]
Adadelta	不需要预设一个默认学习率，训练初中期，加速效果不错，很快，可以避免参数更新时两边单位不统一的问题	在局部最小值附近震荡，可能不收敛	需要快速收敛，训练复杂网络时
Adam	速度快，对内存需求较小，为不同的参数计算不同的自适应学习率	在局部最小值附近震荡，可能不收敛	需要快速收敛，训练复杂网络时；善于处理稀疏梯度和处理非平稳目标的优点，也适用于大多非凸优化 - 适用于大数据集和高维空间

BGD v.s. SGD

BGD: batch gradient descent
SGD: stochastic gradient descent

梯度下降 (Gradient Decent) 是优化算法的一种，其思想是让损失函数沿着梯度的方向下降， 以最快的速度取到最小值。为啥是沿梯度的方向？因为梯度 (gradient) 就是函数变化最快的方向.

批梯度下降(BGD)是梯度下降最基本的形式，下面尝试在Linear Regression算法中使用批梯度下降来优化他的损失函数。

Linear Regression 函数为:

$$h(x) = \sum_{i=0}^{n} \theta_i x_i$$

其损失函数:

$$J(\theta) = \frac{1}{2} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2$$

优化目标: 使得损失函数最小 $min_\theta J(\theta)$

$h(x)$ 学习算法的假设函数，本例中的学习算法是Linear Regression
x_i 数据集的第 i 个特征
θ_i 假设函数对第 i 个特征的系数
n 数据集的特征数
m 数据集的样本数目
$(x^{(i)}, y^{(i)})$ 第 i 条训练样本

BGD v.s. SGD

BGD: batch gradient descent
SGD: stochastic gradient descent

批梯度下降(BGD)是梯度下降最基本的形式，下面尝试在Linear Regression算法中使用批梯度下降来优化他的损失函数。

Linear Regression 函数为:

$$h(x) = \sum_{i=0}^{n} \theta_i x_i$$

其损失函数:

$$J(\theta) = \frac{1}{2} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2$$

优化目标: 使得损失函数最小 $min_\theta J(\theta)$

下降的过程需要起点和终点，期望的终点自然是最小值，那么起点在哪里呢？沿梯度的方向更新:

$$\theta_i := \theta_i - \alpha \frac{\partial J(\theta)}{\partial \theta_i}$$

α 是学习率/步长/训练速度。

BGD v.s. SGD

Linear Regression 函数为:

$$h(x) = \sum_{i=0}^{n} \theta_i x_i$$

其损失函数:

$$J(\theta) = \frac{1}{2} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2$$

一组样本:

$$\frac{\partial J(\theta)}{\partial \theta_i} = \frac{\partial}{\partial \theta_i} \frac{1}{2} (h_\theta(x) - y)^2$$

$$= (h_\theta(x) - y) \bullet \frac{\partial}{\partial \theta_i} (\theta_0 x_0 + \theta_1 x_1 + \ldots + \theta_i x_i + \ldots \theta_n x_n - y)$$

$$= (h_\theta(x) - y) \bullet x_i$$

下降的过程需要起点和终点，期望的终点自然是最小值，那么起点在哪里呢？沿梯度的方向更新:

$$\theta_i := \theta_i - \alpha \frac{\partial J(\theta)}{\partial \theta_i}$$

α 是学习率/步长/训练速度。

m组样本:

$$\frac{\partial J(\theta)}{\partial \theta_i} = \frac{\partial}{\partial \theta_i} \frac{1}{2} \sum_{j=1}^{m} (h_\theta(x^{(j)}) - y^{(j)})^2$$

$$= \sum_{j=1}^{m} \frac{\partial}{\partial \theta_i} \frac{1}{2} (h_\theta(x^{(j)}) - y^{(j)})^2$$

$$= \sum_{j=1}^{m} (h_\theta(x^{(j)}) - y^{(j)}) \bullet x_i^{(j)}$$

需要遍历所有的训练样本，这将带来庞大的计算量，在样本数量巨大的情况下会更为明显，如何避免这个问题？这个时候SGD出现了。

BGD v.s. SGD

SGD是为了避免BGD在样本数量大时带来的巨大计算量。巨大的计算量来自每次迭代对整个训练集的遍历，从这里出发，SGD每次更新只选择一个样本:

$$
\begin{aligned}
&for\ \ j = 1\ \ to\ \ m\ \{ \\
&\theta_i := \theta_i - \alpha(h_\theta(x^{(j)}) - y^{(j)}) \bullet x_i^{(j)}\ \ \text{(for all } i\ \text{)} \\
&\}
\end{aligned}
$$

有得必有失，因为每次只选取一个样本，SGD下降的过程可能没有BGD那样一帆风顺，在“下山”的路上它可能会这里走走那里瞧瞧我们关心的其实只是它能够带我们到达终点而且它够快。也因为每次只选取一个样本，它可能会失去对数据模型的精准刻画，这就导致在遇到噪声时，它可能会把我们带跑偏，即陷入局部最优解。

小批量随机梯度下降 (Mini-batch Stochastic Gradient Decent)

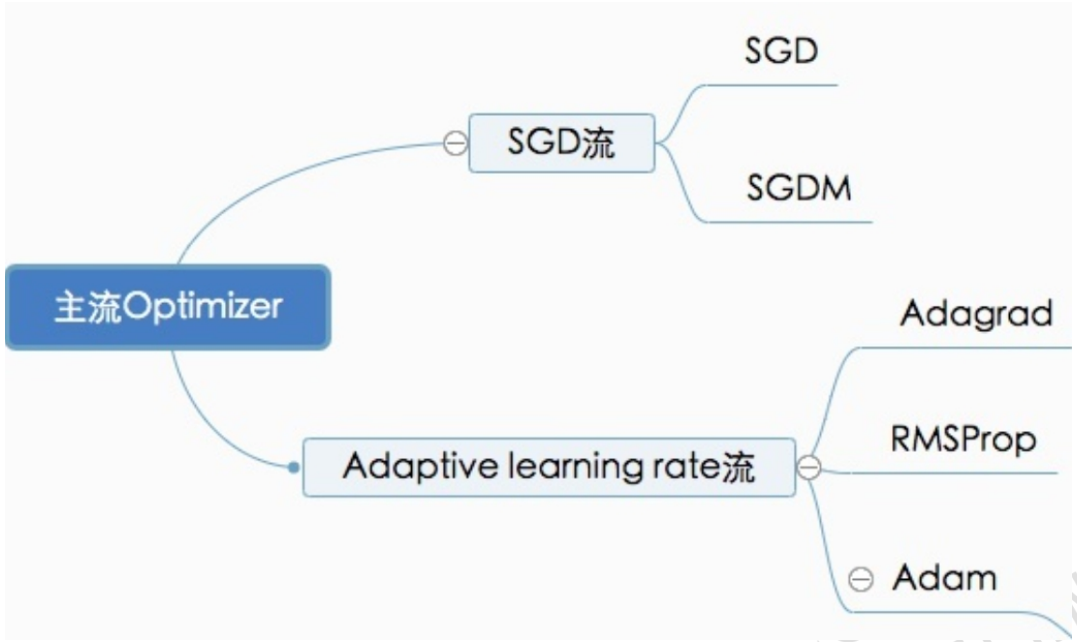
MBGD吸收了BGD和SGD的优点，SGD选择的是一个样本，而MBGD选择将样本划分为多个小块，将每个小块看作是一个样本，这样即保证了对数据模型的精准刻画，也不会太慢。

优化算法框架

一般用一个通用框架来表述优化算法

有如下定义：
1）待优化的参数 θ
2）目标函数 J(θ)
3）学习率 α

有如下过程（每次迭代）：
1）计算目标关于此时参数的梯度 ∇θ(J(θ))
2）计算历史梯度的一阶动量和二阶动量
3）计算下降梯度 g
4）根据梯度进行迭代 θ = θ − g



优化算法目前有固定学习率：
1. 固定学习率
2.自适应学习率

两种，差别也就体现在过程的第1和第2步
固定学习率优化算法有：BGD、SGD、Mini-Batch GD
自适应学习率优化算法有：Adam、AdaGrad、AdaDelta

Adagrad Adaptive Gradient

$$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$$

$$\theta_{t+1} = \theta_t - \alpha \cdot g_t / \sqrt{\sum_{i=1}^{t} g_t^2}.$$

算法 8.4 AdaGrad 算法

Require: 全局学习率 ϵ

Require: 初始参数 θ

Require: 小常数 δ，为了数值稳定大约设为 10^{-7}

初始化梯度累积变量 $r = 0$

while 没有达到停止准则 **do**

从训练集中采包含 m 个样本 $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ 的小批量，对应目标为 $\boldsymbol{y}^{(i)}$。

计算梯度：$\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$

累积平方梯度：$\boldsymbol{r} \leftarrow \boldsymbol{r} + \boldsymbol{g} \odot \boldsymbol{g}$

计算更新：$\Delta\boldsymbol{\theta} \leftarrow -\frac{\epsilon}{\delta + \sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}$ （逐元素地应用除和求平方根）

应用更新：$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$

end while

黄色部分为，不同参数学习率的改变

SGD的梯度下降是学习率恒定的：

$$\theta_i := \theta_i - \alpha \frac{\partial J(\theta)}{\partial \theta_i}$$

简单来讲，设置全局学习率之后，每次通过，全局学习率逐参数的除以历史梯度平方和的平方根，使得每个参数的学习率不同。

起到的效果是在参数空间更为平缓的方向，会取得更大的进步（因为平缓，所以历史梯度平方和较小，对应学习下降的幅度较小），并且能够使得陡峭的方向变得平缓，从而加快训练速度。

RMSProp

算法 8.5 RMSProp 算法

Require: 全局学习率 ϵ，衰减速率 ρ

Require: 初始参数 θ

Require: 小常数 δ，通常设为 10^{-6}（用于被小数除时的数值稳定）

初始化累积变量 $r = 0$

while 没有达到停止准则 **do**

从训练集中采包含 m 个样本 $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ 的小批量，对应目标为 $\boldsymbol{y}^{(i)}$。

计算梯度：$\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$

累积平方梯度：$\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1-\rho) \boldsymbol{g} \odot \boldsymbol{g}$ 与AdaGrad的唯一不同

计算参数更新：$\Delta\boldsymbol{\theta} = -\frac{\epsilon}{\sqrt{\delta + \boldsymbol{r}}} \odot \boldsymbol{g}$ （$\frac{1}{\sqrt{\delta+\boldsymbol{r}}}$ 逐元素应用）

应用更新：$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$

end while

SGD的梯度下降是学习率恒定的：

$$\theta_i := \theta_i - \alpha \frac{\partial J(\theta)}{\partial \theta_i}$$

可以看出RMSProp优化算法和AdaGrad算法唯一的不同，就在于累积平方梯度的求法不同。RMSProp算法不是像AdaGrad算法那样暴力直接的累加平方梯度，而是加了一个衰减系数来控制历史信息的获取多少.

Adam

Adam: A Method for Stochastic Optimization (ICLR 2015)

adaptive **m**oment estimation

Adam 算法和传统的随机梯度下降不同。随机梯度下降保持单一的学习率（即 alpha）更新所有的权重，学习率在训练过程中并不会改变。而 Adam 通过计算梯度的一阶矩估计和二阶矩估计而为不同的参数设计独立的自适应性学习率。

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 1^{st} moment vector)
$v_0 \leftarrow 0$ (Initialize 2^{nd} moment vector)
$t \leftarrow 0$ (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)
$\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
$\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
end while
return θ_t (Resulting parameters)

Adam

Adam 算法同时获得了 AdaGrad 和 RMSProp 算法的优点。Adam 不仅如 RMSProp 算法那样基于一阶矩均值计算适应性参数学习率，它同时还充分利用了梯度的二阶矩均值（即有偏方差/uncentered variance）。具体来说，算法计算了梯度的指数移动均值（exponential moving average），超参数 beta1 和 beta2 控制了这些移动均值的衰减率。

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)
$v_0 \leftarrow 0$ (Initialize 2nd moment vector)
$t \leftarrow 0$ (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)

alpha：同样也称为学习率或步长因子，它控制了权重的更新比率（如 0.001）。较大的值（如 0.3）在学习率更新前会有更快的初始学习，而较小的值（如 1.0E-5）会令训练收敛到更好的性能。

beta1：一阶矩估计的指数衰减率（如 0.9）。

beta2：二阶矩估计的指数衰减率（如 0.999）。该超参数在稀疏梯度（如在 NLP 或计算机视觉任务中）中应该设置为接近 1 的数。

epsilon：该参数是非常小的数，其为了防止在实现中除以零（如 10E-8）。

Adam

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)
$v_0 \leftarrow 0$ (Initialize 2nd moment vector)
$t \leftarrow 0$ (Initialize timestep)

$$\alpha_t = \alpha \cdot \sqrt{1 - \beta_2^t}/(1 - \beta_1^t)$$

$$\theta_t \leftarrow \theta_{t-1} - \alpha_t \cdot m_t/(\sqrt{v_t} + \hat{\epsilon})$$

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)
$\widehat{m}_t \leftarrow m_t/(1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
$\widehat{v}_t \leftarrow v_t/(1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t/(\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
end while
return θ_t (Resulting parameters)

Adam

Adam 算法更新规则的一个重要特征就是它会很谨慎地选择步长的大小。假定ε=0，则在时间步 t 和参数空间上的有效下降步长为

$$\Delta_t = \alpha \cdot \hat{m}_t / \sqrt{\hat{v}_t}$$

有效下降步长有两个上确界：即在

$$|\Delta_t| \leq \alpha \cdot (1 - \beta_1)/\sqrt{1 - \beta_2} \text{ in the case } (1 - \beta_1) > \sqrt{1 - \beta_2},$$

和其他情况下满足 |Δt| ≤ α。

第一种情况只有在极其稀疏的情况下才会发生：即梯度除了当前时间步不为零外其他都为零。而在不那么稀疏的情况下，有效步长将会变得更小

https://blog.csdn.net/yzy_1996/article/details/84618536

https://github.com/yzy1996/Python-Code/tree/master/Python%2BAlgorithm/Optimization-Algorithm

Thanks