从SGD到NadaMax，十种优化算法原理及实现

永远在你…

生乎？死乎？编乎。知乎。

+ 关注他

无论是什么优化算法，最后都可以用一个简单的公式抽象：

$$w = w + \Delta w$$

w 是参数，而 Δw 是参数的增量，而各种优化算法的主要区别在于对 Δw 的计算不同，本文总结了下面十个优化算法的公式，以及简单的Python实现：

1. SGD
2. Momentum
3. Nesterov Momentum
4. AdaGrad
5. RMSProp
6. AdaDelta
7. Adam
8. AdaMax
9. Nadam
10. NadaMax

· SGD

虽然有凑数的嫌疑，不过还是把SGD也顺带说一下，就算做一个符号说明了。常规的随机梯度下降公式如下：

$$\Delta w = -\eta J'(w)$$

其中 η 是学习率， $J'(w)$ 是损失关于参数的梯度（有的资料中会写成 $\nabla_w J(w)$ 等形式），不过相比SGD，用的更多的还是小批量梯度下降（mBGD）算法，不同之处在于一次训练使用多个样本，然后取所有参与训练样本梯度的平均来更新参数，公式如下：

$$\Delta w = -\eta g_i$$

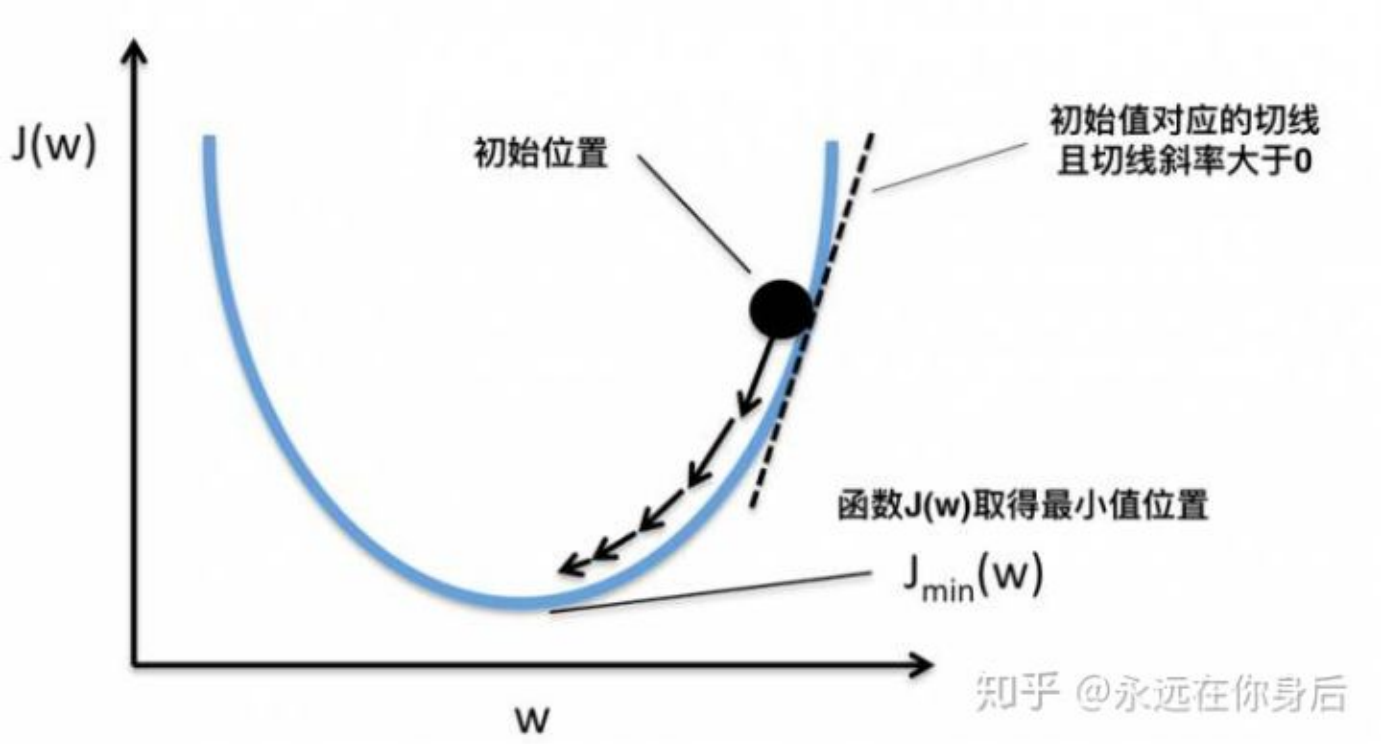
$$g_i = \frac{1}{m}\sum_{k=1}^{m} J'(w)$$

其中 g_i 是第 i 次训练中 m 个样本损失关于参数梯度的均值，如无特别声明，下文所出现 g_i 也遵循该定义

另外 $J'(w)$ 或者 g_i 在下面的优化算法中，只是作为一个传入的变量，其具体的计算是由其他模块负责，可以参考下面两个链接：

· Momentum

Momentum，也就是动量的意思。该算法将梯度下降的过程视为一个物理系统，下图是在百度图片中找的（侵删）



图片来自网络

如上图所示，在该物理系统中有一个小球（质点），它所处的水平方向的位置对应为 w 的值，而垂直方向对应为损失。设其质量 $m=1$，在第 i 时刻，在单位时间内，该质点受外力而造成的动量改变为：

$$Ft = mv_i - mv_{i-1} \qquad (1.1)$$
$$F = v_i - v_{i-1} \qquad (1.2)$$

(1.1)到(1.2)是因为 $m = 1, t = 1$ ，所以约去了。另外受到的外力可以分为两个分量：**重力**沿斜面向下的力 I_G 和**粘性阻尼力** I_V

$$\begin{aligned} F &= I_G + I_V \\ &= I_G - cv \end{aligned} \qquad (1.3)$$

令

$$\alpha = 1 - c$$

代入(1.2)式中：

$$\begin{aligned} v_i &= v_{i-1} + I_G - cv_{i-1} \\ &= \alpha v_{i-1} + I_G \end{aligned} \qquad (1.4)$$

然后对“位置”进行更新：

$$\begin{aligned} w &= w + v_i t \\ &= w + v_i \end{aligned} \qquad (1.5)$$

所以这里 $\Delta w = v_i$ ，另外 I_G 的方向与损失的梯度方向相反，并取系数为 η ，得到：

$$I_G = -\eta g_i$$

代入(1.4)，得到速度的更新公式：

$$v_i = \alpha v_{i-1} - \eta g_i \qquad (1.6)$$

进一步的，将(1.6)式展开，可以得到：

$$
\begin{aligned}
v_i &= \alpha v_{i-1} - \eta g_i \\
&= \alpha(\alpha v_{i-2} - \eta g_{i-1}) - \eta g_i \\
&= \alpha(\alpha(\alpha v_{i-3} - \eta g_{i-2}) - \eta g_{i-1}) - \eta g_i \\
&\cdots \\
&= \alpha^i v_0 - \eta\alpha^{i-1} g_1 - \cdots - \eta\alpha^1 g_{i-1} - \eta\alpha^0 g_i \\
&= \alpha^i v_0 - \eta \sum_{j=1}^{i} \alpha^{i-j} g_j \\
&= -\eta \sum_{j=1}^{i} \alpha^{i-j} g_j \quad (\because v_0 = 0)
\end{aligned}
$$

可以看出来是一个变相的等比数列之和，且公比小于1，所以存在极限，当 i 足够大时， v_i 趋近于 $\eta\dfrac{g_i}{1-\alpha}$

实现代码

```python
import numpy as np

class Momentum(object):
    def __init__(self, alpha=0.9, lr=1e-3):
        self.alpha = alpha  # 动量系数
        self.lr = lr        # 学习率
        self.v = 0          # 初始速度为0

    def update(self, g: np.ndarray):    # g = J'(w) 为本轮训练参数的梯度
        self.v = self.alpha * self.v - self.lr * g  # 公式
        return self.v    # 返回的是参数的增量，下同
```

以上是基于**指数衰减**的实现方式，另外有的Momentum算法中会使用**指数加权平均**来实现，主要公式如下：

$$
\begin{aligned}
v_i &= \alpha v_{i-1} + (1-\alpha) g_i \\
\Delta w &= -\eta v_i
\end{aligned}
$$

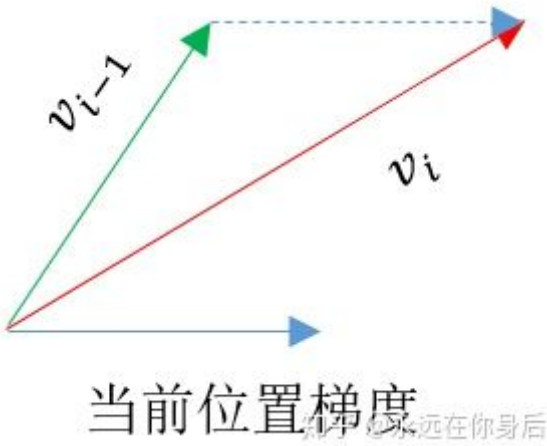
不过该方式因为 $v_0 = 0$ ，刚开始时 v_i 会比期望值要小，需要进行修正，下面的Adam等算法会使用该方式

- Nesterov Momentum

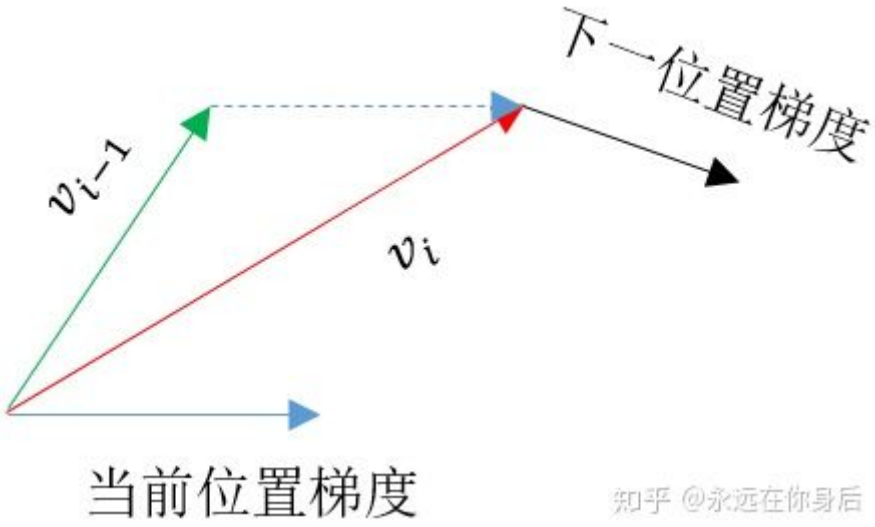
Nesterov Momentum是Momentum的改进版本，与Momentum唯一区别就是，Nesterov先用当前的速度 v 更新一遍参数，得到一个临时参数 $\tilde{w}$ ，然后使用这个临时参数计算本轮训练的梯度。相当于是小球预判了自己下一时刻的位置，并提前使用该位置的梯度更新：

$$\begin{aligned}\tilde{w} &= w + \alpha v_{i-1} \\ g_i &= J'(\tilde{w}) \\ v_i &= \alpha v_{i-1} - \eta g_i \\ w &= w + v_i\end{aligned}$$

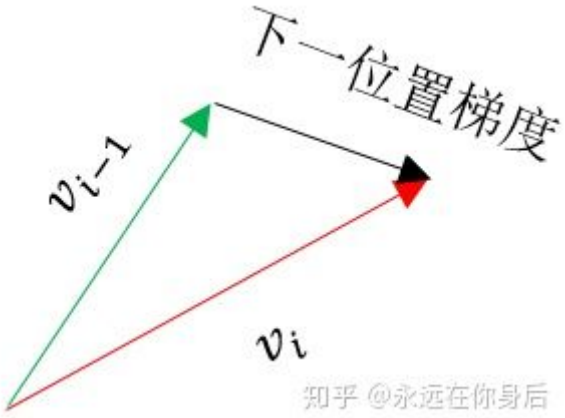
为了更加直观，还是上几个图吧，以下是Momentum算法 v_i 的更新过程：



假设下一个位置的梯度如下：



那么Nesterov Momentum就提前使用这个梯度进行更新：



整体来看Nesterov的表现要好于Momentum，至于代码实现的话因为主要变化的是 g_i ，所以可以之前使用Momentum的代码

· AdaGrad

AdaGrad全称为Adaptive Subgradient，其主要特点在于不断累加每次训练中梯度的平方，公式如下：

$$r_i = r_{i-1} + g_i^2 \tag{2.1}$$

$$\Delta w = -\frac{\eta}{\epsilon + \sqrt{r_i}} g_i \tag{2.2}$$

$$w = w + \Delta w$$

其中 ϵ 是一个极小的正数，用来防止除0，而 $g_i^2 = g_i \odot g_i$ ， $\odot$ 是矩阵的**哈达玛积**运算符，另外，本文中矩阵的平方或者两矩阵相乘都是计算哈达玛积，而不是计算矩阵乘法

从公式中可以看出，随着算法不断迭代， r_i 会越来越大，整体的学习率会越来越小。所以，一般来说AdaGrad算法一开始是激励收敛，到了后面就慢慢变成惩罚收敛，速度越来越慢

对于代码实现，首先将 r_i 展开得到：

$$\begin{aligned} r_i &= r_{i-1} + g_i^2 \\ &= r_{i-2} + g_{i-1}^2 + g_i^2 \\ &= r_0 + g_1^2 + g_2^2 + \cdots + g_i^2 \\ &= r_0 + \sum_{j=1}^{i} g_j^2 \end{aligned} \tag{2.3}$$

通常 $r_0 = 0$ ，所以在第一次训练时(2.2)式为：

$$\Delta w = -\frac{\eta}{\epsilon + \sqrt{r_1}} g_1$$
$$= -\frac{\eta}{\epsilon + g_1} g_1$$

因为每次训练 g_i 的值是不确定的，所以要防止处0，但是可以令 $r_0 = \epsilon$ ，这样就可以在(2.2)式中去掉 ϵ

$$\Delta w = -\frac{\eta}{\sqrt{r_i}} g_i$$

将 $r_0 = \epsilon$ 代入(2.3)式，可以得到：

$$r_i = r_0 + \sum_{j=1}^{i} g_j^2$$
$$= \epsilon + \sum_{j=1}^{i} g_j^2 > 0$$

可知 r_i 恒大于0，因此不必在计算 Δw 中额外加入 ϵ ，代码如下：

```
class AdaGrad(object):
    def __init__(self, eps=1e-8, lr=1e-3):
        self.r = eps    # r_0 = epsilon
        self.lr = lr

    def update(self, g: np.ndarray):
        r = r + np.square(g)
        return -self.lr * g / np.sqrt(r)
```

· RMSProp

RMSProp是AdaGrad的改进算法，其公式和AdaGrad的区别只有 r_i 的计算不同，先看公式

$$r_i = \beta r_{i-1} + (1-\beta) g_i^2$$
$$\Delta w = -\frac{\eta}{\epsilon + \sqrt{r_i}} g_i$$
$$w = w + \Delta w$$

可以看出，与AdaGrad不同，RMSProp只会累积近期的梯度信息，对于“遥远的历史”会以指数衰减的形式放弃

并且AdaGrad算法虽然在**凸函数**(Convex Functions)上表现较好，但是当目标函数非凸时，算法梯度下降的轨迹所经历的结构会复杂的多，早期梯度对当前训练没有太多意义，此时RMSProp往往表现更好

以下是将 r_i 展开后的公式：

$$
\begin{aligned}
r_i &= \beta r_{i-1} + (1-\beta) g_i^2 \\
&= \beta^i r_0 + (1-\beta) \sum_{j=1}^{i} \beta^{i-j} g_j^2
\end{aligned}
$$

与AdaGrad一样，令 $r_0 = \epsilon$ ，从而去掉计算 Δw 时的 ϵ ，实现代码：

```
class RMSProp(object):
    def __init__(self, lr=1e-3, beta=0.999, eps=1e-8):
        self.r = eps
        self.lr = lr
        self.beta = beta

    def update(self, g: np.ndarray):
        r = r * self.beta + (1-self.beta) * np.square(g)
        return -self.lr * g / np.sqrt(r)
```

· AdaDelta

AdaDelta是与RMSProp相同时间对立发展出来的一个算法，在实现上可以看作是RMSProp的一个变种，先看公式：

$$
\begin{aligned}
r_i &= \beta r_{i-1} + (1-\beta) g_i^2 \\
\Delta w &= -\frac{\sqrt{\epsilon + s_{i-1}}}{\sqrt{\epsilon + r_i}} g_i \qquad\qquad (3.1) \\
s_i &= \beta s_{i-1} + (1-\beta) \Delta w^2 \\
w &= w + \Delta w
\end{aligned}
$$

可以看到该算法不需要设置学习率 η ，这是该算法的一大优势。除了同样以 r_i 来累积梯度的信息之外，该算法还多了一个 s_i 以指数衰减的形式来累积 Δw 的信息

与前面相同，令：

$$s_0 = \epsilon$$
$$r_0 = \epsilon$$

然后去掉(3.1)中的 ϵ ，得到：

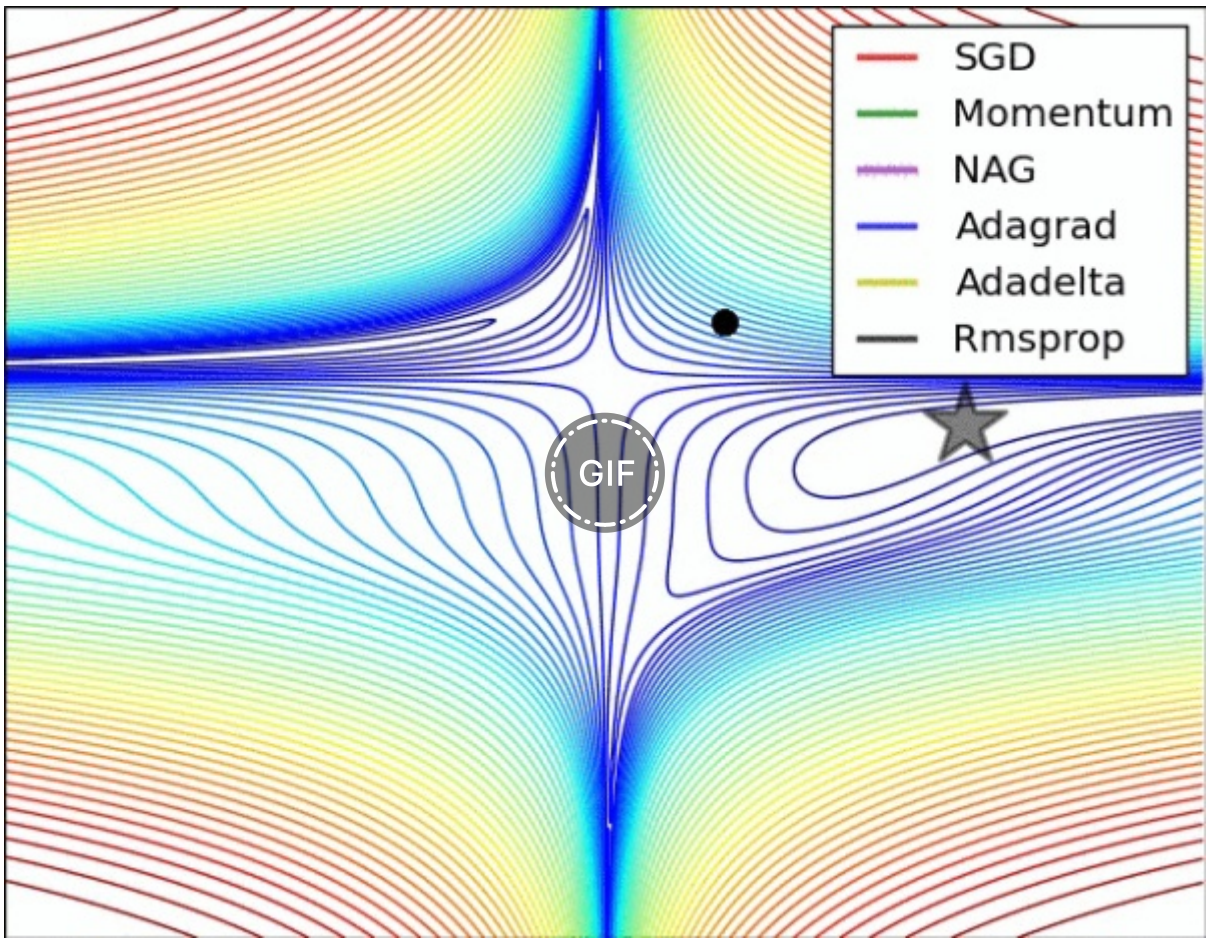
$$\Delta w = -\frac{\sqrt{s_{i-1}}}{\sqrt{r_i}} g_i$$
$$\Delta w^2 = \frac{s_{i-1}}{r_i} g_i^2$$
$$s_i = \beta s_{i-1} + \frac{s_{i-1}}{r_i}(1-\beta) g_i^2$$

这样的话可以减少一些计算，代码如下：

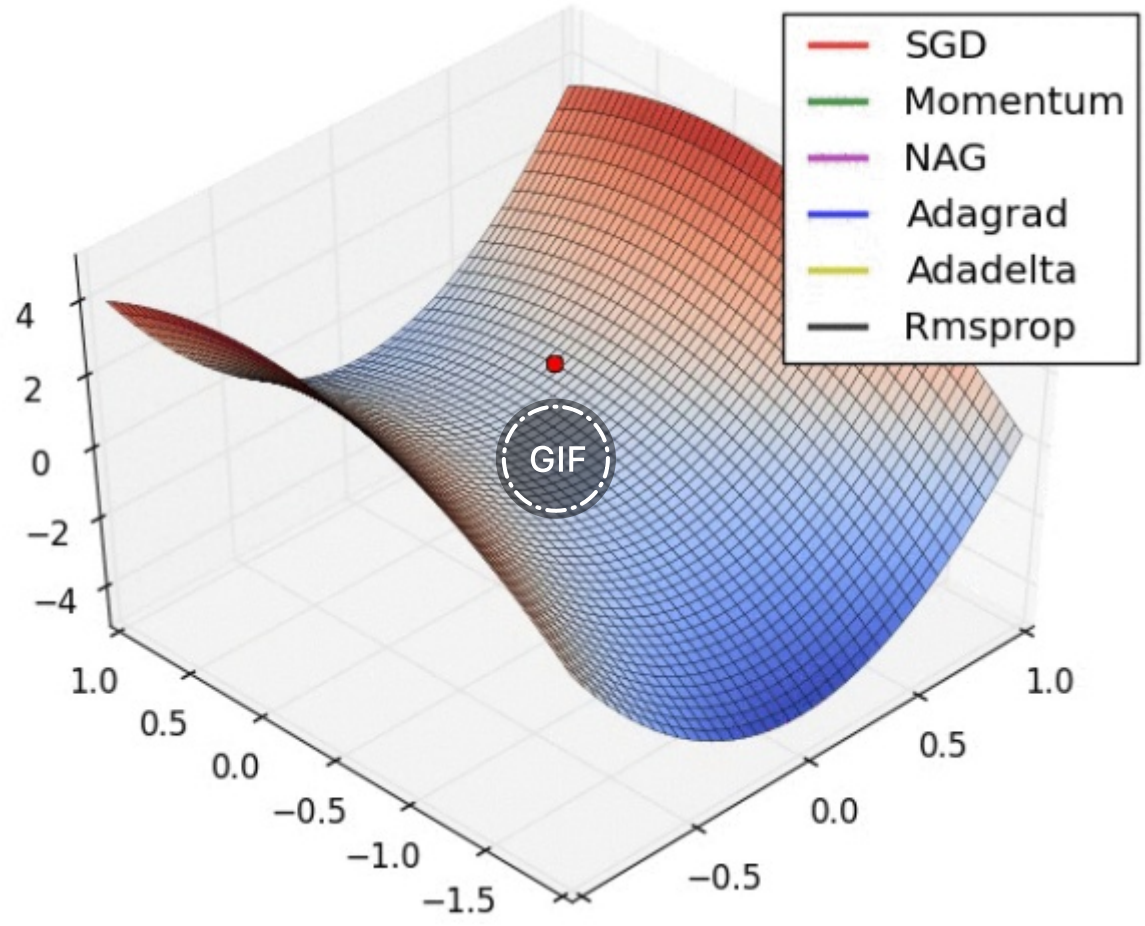
```
class AdaDelta(object):
    def __init__(self, beta=0.999, eps=1e-8):
        self.r = eps
        self.s = eps
        self.beta = beta

    def update(self, g: np.ndarray):
        g_square = (1-self.beta) * np.square(g)     # (1-beta)*g^2
        r = r * self.beta + g_square
        frac = s / r
        res = -np.sqrt(frac) * g
        s = s * self.beta + frac * g_squaretmp      # 少一次乘法。。。
        return res
```

关于以上几个算法的对比：



其中NAG是Nesterov Momentum



更多关于AdaDelta的信息，可以参考这篇文章：自适应学习率调整：AdaDelta

· Adam

Adam的名称来自Adaptive Momentum，可以看作是Momentum与RMSProp的一个结合体，该算法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，公式如下：

$$
\begin{aligned}
s_i &= \alpha s_{i-1} + (1-\alpha) g_i && (4.1)\\
r_i &= \beta r_{i-1} + (1-\beta) g_i^2 && (4.2)\\
\hat{s}_i &= \frac{s_i}{1-\alpha^i} && (4.3)\\
\hat{r}_i &= \frac{r_i}{1-\beta^i} && (4.4)\\
\Delta w &= -\eta \frac{\hat{s}_i}{\sqrt{\hat{r}_i + \epsilon}} && (4.5)\\
w &= w + \Delta w
\end{aligned}
$$

(4.1)和(4.2)在Momentum和RMSProp中已经介绍过了，而不直接使用 s_i, r_i 计算 Δw 却先经过(4.3)和(4.4)式是因为通常会设 $s_0 = r_0 = 0$ ，所以此时梯度的一阶矩估计和二阶矩估是有偏的，需要进行修正

虽然没办法避免修正计算，但是还是可以省去一些计算过程，初始化时令：

$$
\begin{aligned}
s_0 &= 0\\
r_0 &= \epsilon
\end{aligned}
$$

然后(4.5)式变为：

$$
\begin{aligned}
\Delta w &= -\eta \frac{\hat{s}}{\sqrt{\hat{r}}}\\
&= -\eta \frac{\sqrt{1-\beta^i}}{1-\alpha^i} \frac{s_i}{\sqrt{r_i}}
\end{aligned}
$$

因为 $\alpha, \beta < 1$ ，可知当 i 足够大时修正将不起作用（也不需要修正了）：

$$
\frac{\sqrt{1-\beta^i}}{1-\alpha^i} \to 1
$$

代码如下：

```
class Adam(object):
    def __init__(self, lr=1e-3, alpha=0.9, beta=0.999, eps=1e-8):
        self.s = 0
        self.r = eps
        self.lr = lr
        self.alpha = alpha
        self.beta = beta
        self.alpha_i = 1
        self.beta_i = 1

    def update(self, g: np.ndarray):
        self.s = self.s * self.alpha + (1-self.alpha) * g
        self.r = self.r * self.beta + (1-self.beta) * np.square(g)
        self.alpha_i *= self.alpha
        self.beta_i *= self.beta_i
        lr = -self.lr * (1-self.beta_i)**0.5 / (1-self.alpha_i)
        return lr * self.s / np.sqrt(self.r)
```

· AdaMax

首先回顾RSMSProp中 r_i 的展开式并且令 $r_0 = 0$ ，得到：

$$
\begin{aligned}
r_i &= \beta r_{i-1} + (1-\beta) g_i^2 \\
&= (1-\beta) \sum_{j=1}^{i} \beta^{i-j} g_j^2
\end{aligned}
$$

可以看到这相当于是一个 g_j 的 L_2 范数，也就是说 w 的各维度的增量是根据该维度上梯度的 L_2 范数的累积量进行缩放的。如果用 L_p 范数替代就得到了Adam的不同变种，不过其中 L_∞ 范数对应的变种算法简单且稳定

对于 L_p 范数，第 i 轮训练时梯度的累积为：

$$
\begin{aligned}
r_i &= \beta^p r_{i-1} + (1-\beta^p) |g_i|^p \\
&= (1-\beta^p) \sum_{j=1}^{i} \beta^{i-j} |g_j|^p
\end{aligned}
$$

然后求无穷范数：

$$
\begin{aligned}
\lim_{p\to\infty}(r_i)^{1/p} &= \lim_{p\to\infty}(1-\beta^p\sum_{j=1}^{i}\beta^{p(i-j)}|g_i|^p)^{1/p} \\
&= \lim_{p\to\infty}(1-\beta^p)^{1/p}(\sum_{j=1}^{i}\beta^{p(i-j)}|g_i|^p)^{1/p} \\
&= \lim_{p\to\infty}(\sum_{j=1}^{i}(\beta^{i-j}|g_i|)^p)^{1/p} \\
&= \max(\beta^{i-1}|g_1|,\beta^{i-2}|g_2|,\ldots,\beta|g_{i-1}|,|g_i|)
\end{aligned}
$$

由此再来递推 r_{i+1} ：

$$
\begin{aligned}
r_{i+1} &= \max(\beta^{i}|g_1|,\beta^{i-1}|g_2|,\ldots,\beta^2|g_{i-1}|,\beta|g_i|,|g_{i+1}|) \\
&= \max(\beta r_i,|g_{i+1}|)
\end{aligned}
$$

需要注意，这个max比较的是梯度**各个维度**上的当前值和历史最大值，具体可以结合代码来看，最后其公式总结如下：

$$
\begin{aligned}
s_i &= \alpha s_{i-1} + (1-\alpha)g_i \\
\hat{s}_i &= \frac{s_i}{1-\alpha^i} \\
r_i &= \max(\beta r_{i-1},|g_i|) \\
\Delta w &= -\eta\frac{\hat{s}_i}{r_i} \\
w &= w + \Delta w
\end{aligned}
$$

另外，因为 r_i 是累积的梯度各个分量的绝对值最大值，所以直接用做分母且不需要修正，代码如下：

```
class AdaMax(object):
    def __init__(self, lr=1e-3, alpha=0.9, beta=0.999):
        self.s = 0
        self.r = 0
        self.lr = lr
        self.alpha = alpha
        self.alpha_i = 1
        self.beta = beta

    def update(self, g: np.ndarray):
        self.s = self.s * self.alpha + (1-self.alpha) * g
```

```
        self.r = np.maximum(self.r*self.beta, np.abs(g))
        self.alpha_i *= self.alpha
        lr = -self.lr / (1-self.alpha_i)
        return lr * self.s / self.r
```

· Nadam

Adam可以看作是Momentum与RMSProp的结合，既然Nesterov的表现较Momentum更优，那么自然也就可以把Nesterov Momentum与RMSProp组合到一起了，首先来看Nesterov的主要公式：

$$g_i = J'(w + \alpha v_{i-1}) \qquad (5.1)$$
$$v_i = \alpha v_{i-1} - \eta g_i$$
$$w = w + v_i \qquad (5.2)$$

为了令其更加接近Momentum，将(5.1)和(5.2)修改为：

$$g_i = J'(w) \qquad (5.3)$$
$$v_i = \alpha v_{i-1} - \eta g_i$$
$$w = w + (\alpha v_i - \eta g_i) \qquad (5.4)$$

然后列出Adam中Momentum的部分：

$$s_i = \alpha s_{i-1} + (1-\alpha) g_i \qquad (5.5)$$
$$\hat{s}_i = \frac{s_i}{1-\alpha^i} \qquad (5.6)$$
$$\Delta w = -\eta \frac{\hat{s}_i}{\sqrt{\hat{r}_i + \epsilon}} \qquad (5.7)$$

将(5.5)和(5.6)式代入到(5.7)式中：

$$\Delta w = -\frac{\eta}{\sqrt{\hat{r}_i + \epsilon}} \frac{\alpha s_{i-1} + (1-\alpha) g_i}{1-\alpha^i}$$
$$= -\frac{\eta}{\sqrt{\hat{r}_i + \epsilon}} \left(\frac{\alpha s_{i-1}}{1-\alpha^i} + \frac{(1-\alpha) g_i}{1-\alpha^i} \right)$$

将上式中标红部分进行近似：

$$\frac{s_{i-1}}{1-\alpha^i} \approx \frac{s_{i-1}}{1-\alpha^{i-1}} = \hat{s}_{i-1}$$

代入原式，得到：

$$\Delta w = -\frac{\eta}{\sqrt{\hat{r}_i + \epsilon}}(\alpha \hat{s}_{i-1} + \frac{(1-\alpha)g_i}{1-\alpha^i})$$

接着，按照(5.4)式的套路，将 $\hat{s}_{i-1}$ 替换成 $\hat{s}_i$ ，得到：

$$\Delta w = -\frac{\eta}{\sqrt{\hat{r}_i + \epsilon}}(\alpha \hat{s}_i + \frac{(1-\alpha)g_i}{1-\alpha^i}) \tag{5.8}$$

整理一下公式：

$$\begin{aligned}
s_i &= \alpha s_{i-1} + (1-\alpha)g_i \\
r_i &= \beta r_{i-1} + (1-\beta)g_i^2 \\
\hat{s}_i &= \frac{s_i}{1-\alpha^i} \\
\hat{r}_i &= \frac{r_i}{1-\beta^i} \\
\Delta w &= -\frac{\eta}{\sqrt{\hat{r}_i + \epsilon}}(\alpha \hat{s}_i + \frac{(1-\alpha)g_i}{1-\alpha^i})
\end{aligned}$$

同样令 $r_0 = \epsilon$ ，消去(5.8)式种的 ϵ ：

$$\begin{aligned}
\Delta w &= -\frac{\eta}{\sqrt{\hat{r}_i}}(\alpha \hat{s}_i + \frac{(1-\alpha)g_i}{1-\alpha^i}) \\
&= -\frac{\eta\sqrt{1-\beta^i}}{\sqrt{r_i}}(\alpha \hat{s}_i + \frac{(1-\alpha)g_i}{1-\alpha^i}) \\
&= -\frac{\eta\sqrt{1-\beta^i}}{\sqrt{r_i}}(\alpha \frac{s_i}{1-\alpha^i} + \frac{(1-\alpha)g_i}{1-\alpha^i}) \\
&= -\frac{\eta\sqrt{1-\beta^i}}{1-\alpha^i}\frac{1}{\sqrt{r_i}}(\alpha s_i + (1-\alpha)g_i)
\end{aligned}$$

代码

```
class Nadam(object):
    def __init__(self, lr=1e-3, alpha=0.9, beta=0.999, eps=1e-8):
        self.s = 0
        self.r = eps
        self.lr = lr
        self.alpha = alpha
        self.beta = beta
        self.alpha_i = 1
        self.beta_i = 1

    def update(self, g: np.ndarray):
        self.s = self.s * self.alpha + (1-self.alpha) * g
        self.r = self.r * self.beta + (1-self.beta) * np.square(g)
        self.alpha_i *= self.alpha
        self.beta_i *= self.beta_i
        lr = -self.lr * (1-self.beta_i)**0.5 / (1-self.alpha_i)
        return lr * (self.s * self.alpha + (1-self.alpha) * g) / np.sqrt(self.r)
```

· NadaMax

按照同样的思路，可以将Nesterov与AdaMax结合变成NadaMax，回顾以下(5.8)式：

$$\Delta w = -\frac{\eta}{\sqrt{\hat{r}_i + \epsilon}}(\alpha\hat{s}_i + \frac{(1-\alpha)g_i}{1-\alpha^i}) \qquad (6.1)$$

然后是AdaMax的二阶矩估计部分：

$$r_i = \max(\beta r_{i-1}, |g_i|) \qquad (6.2)$$

$$\Delta w = -\eta\frac{\hat{s}_i}{r_i} \qquad (6.3)$$

用(6.2)式替换掉(6.1)式中标红部分，得到：

$$\begin{aligned}\Delta w &= -\frac{\eta}{r_i}(\alpha\hat{s}_i + \frac{(1-\alpha)g_i}{1-\alpha^i}) \\ &= -\frac{\eta}{1-\alpha^i}\frac{1}{r_i}(\alpha s_i + (1-\alpha)g_i)\end{aligned}$$

最后，整理公式：

$$
\begin{aligned}
s_i &= \alpha s_{i-1} + (1-\alpha) g_i \\
r_i &= \max(\beta r_{i-1}, |g_i|) \\
\hat{s}_i &= \frac{s_i}{1-\alpha^i} \\
\Delta w &= -\frac{\eta}{1-\alpha^i} \frac{1}{r_i} (\alpha s_i + (1-\alpha) g_i)
\end{aligned}
$$

代码实现：

```
class NadaMax(object):
    def __init__(self, lr=1e-3, alpha=0.9, beta=0.999):
        self.s = 0
        self.r = 0
        self.lr = lr
        self.alpha = alpha
        self.alpha_i = 1
        self.beta = beta

    def update(self, g: np.ndarray):
        self.s = self.s * self.alpha + (1-self.alpha) * g
        self.r = np.maximum(self.r*self.beta, np.abs(g))
        self.alpha_i *= self.alpha
        lr = -self.lr / (1-self.alpha_i)
        return lr * (self.s * self.alpha + (1-self.alpha) * g) / self.r
```

参考资料：

[1]: 《机器学习算法背后的理论与优化》 ISBN 978-7-302-51718-4

[2]: Adam: A Method for Stochastic Optimization

[3]: Incorporating Nesterov Momentum into Adam

[4]: An overview of gradient descent optimization algorithms

发布于 2019-09-05

机器学习　神经网络　深度学习（Deep Learning）