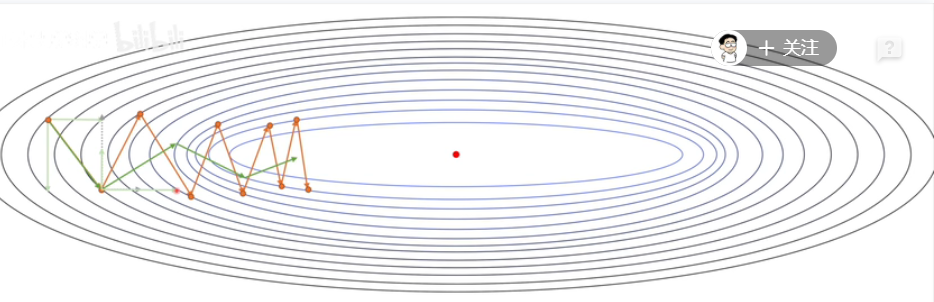
**几种改进的梯度下降法**

1. **Momentum冲量法**

**1.背景及理解**

梯度下降法在求解时的难题主要在于解决极小值和鞍点的问题，为了解决这个问题，可以模拟在现实中的惯性。物体有一个初始动量，在平缓的区域或者小的坑里也能继续向前运动试图滚出小坑，在动量变为0的时候停止，表示已经达到最低点。这个可以想成圆球在椭圆形的碗里滚到最低点的过程类似：动量每次都在累积，小球越来越快。每次梯度一致的方向上，速度越来越大，不一致的方向上，累积较小，甚至减小。这个最终使得收敛加速，震荡减少。

或者我们从下图来表示，对于每一条边，将其拆分为横轴上的分量和纵轴上的分量，我们发现横轴上的分量始终向前，纵轴上的分量则来回震荡，如果可以加速横轴分量，减少纵轴分量，就可以优化算法。因此我们用历史数据去优化当前前进的方向



**2.算法公式**

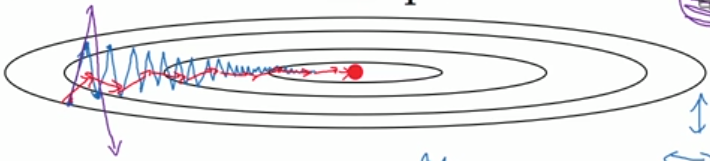
 （为学习率）

其中为权重，表示越远的数据权重越小，越近的数据权重越大。

动量法即在简单的梯度下降算法的基础上，代表上一时刻的动量的项v被加入进来，并且每一次都会乘上一个衰减系数，该系数可以类比于物理运动中摩擦的存在，前一次的迭代位置方向可以影响到下一次迭代。直到动量小于某个值，或者梯度小于一个值，或者迭代到了一定的次数。

**3.特点：**

实验表明，相比于标准梯度下降算法，Momentum算法具有更快的收敛速度。这是因为如下图：



蓝线是标准梯度下降法，可以看到收敛过程中产生了一些震荡。这些震荡在纵轴方向上是均匀的，几乎可以相互抵消，也就是说如果直接沿着横轴方向迭代，收敛速度可以加快。Momentum通过对原始梯度做了一个平滑，正好将纵轴方向的梯度抹平了（红线部分），使得参数更新方向更多地沿着横轴进行，因此速度更快。动量梯度下降法不但能使用较大的学习率，其迭代次数也较少

在学习率较小的时候，适当的 momentum 能够起到一个 加速收敛速度的作用。在学习率较大的时候，适当的 momentum 能够起到一个 减小收敛时震荡幅度的作用

momentum 能够较好地解决最速下降法学习率较小时，收敛到正确结果的速度较慢。学习率较大时，容易在搜索过程中发生震荡的问题。

**4.冲量法优缺点总结以及相对于最速下降法的提升**

|  |  |
| --- | --- |
| 优点 | 1. 收敛速度快  2. 在 学 习 率 较 小 的 时 候 ， 适 当 的momentum 能够起到一个加速收敛速度的作用。在学习率较大的时候，适当的momentum 能够起到一个减小收敛时震荡幅度的作用。3. 适当的学习率和冲量的选取可能有一定几率跳出局部最优 |
| 缺点 | 1. 当 momentum 较大时，原本能够正确收敛的时候却因为刹不住车跑过头了   2. 计算量大增，尤其是在迭代次数非常大的时候。 |
| 相对于最速下降法的提升 | 1.收敛速度快  2.解决了最速下降法学习率较小时，收敛到正确结果的速度较慢。学习率较大时，容易在搜索过程中发生震荡的问题。  3.解决了最速下降法靠近极小值时收  敛速度减慢 |

**5.编程实现**

#include <iostream>

#include <string>

#include <vector>

using namespace std;

double r = 0.01; //学习率

double e = 0.00001;  //精度

double b = 0.9;

double qd(double w);

int main()

{

    double wcur = 0;

    double wper = -1;

    double vcur = 2;

    double vper = 1;

    while(wcur-wper > e)

    {

        double J = qd(wper);

        double temv = vcur;

        double temw = wcur;

        vcur = b \* vper + (1-b) \* J;

        wcur = wper - r \* vcur;

        vper = temv;

        wper = temw;

    }

    cout << wcur;

    system("pause");

    return 0;

}

double qd(double w) //以函数y=2\*x^2+4\*4+5为例

{

    return 4 \* w + 4;

}

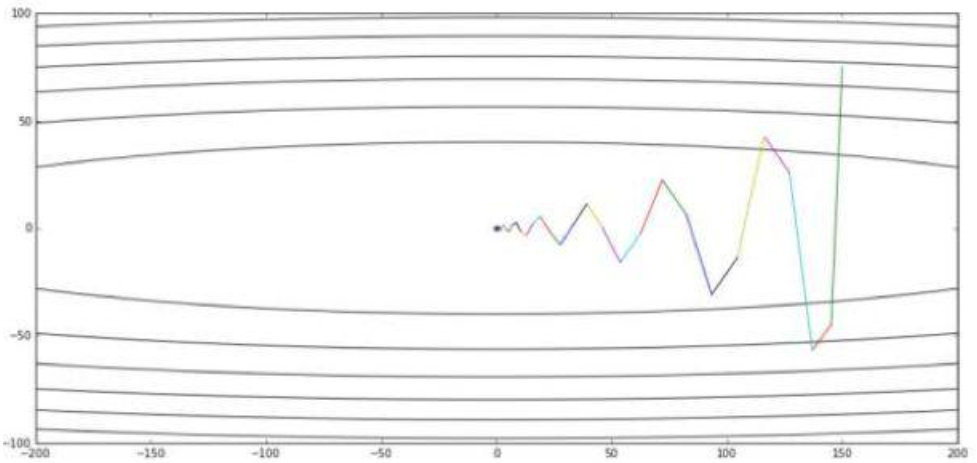


由结果可知程序所求结果与真实值十分接近，且运行时速度较快。

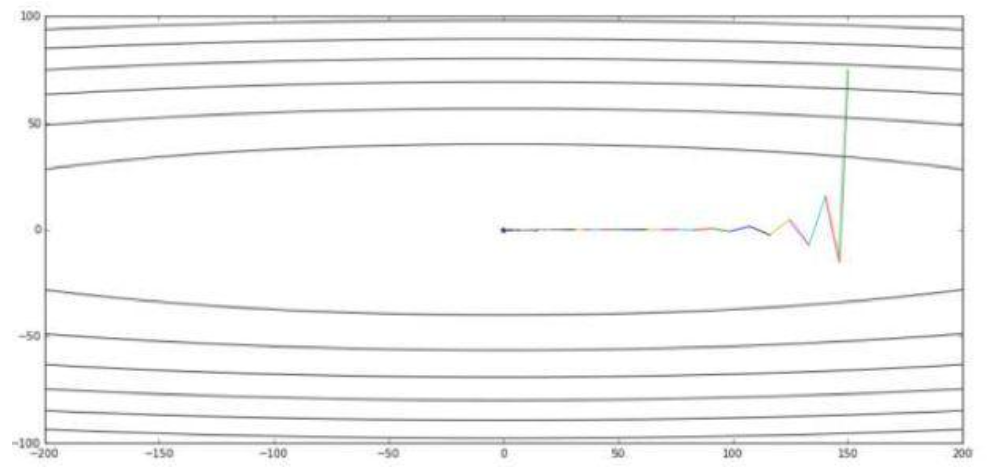
**二、NAG法**

**1.背景及理解**

NAG（Nesterov Accelerated Gradient）不仅仅把SGD梯度下降以前的方向考虑，还将Momentum梯度变化的幅度也考虑了进来。NAG算法是对冲量梯度下降算法的改进版本，动量方法导致小球到达最低点后动量非常大，可能会错过最低点，所以在小球在上坡时继续向上冲的时候小球应该减速更多，而不是只通过动量项的减少来简单地降低速度。这个算法是通过粗略预测下一个更新的位置的梯度（即θ - γm）来进行现在的更新，有效避免上坡又越过了最低点的部分情况。NAG可以避免更新过猛，在RNNs中可以明显的提升性能。

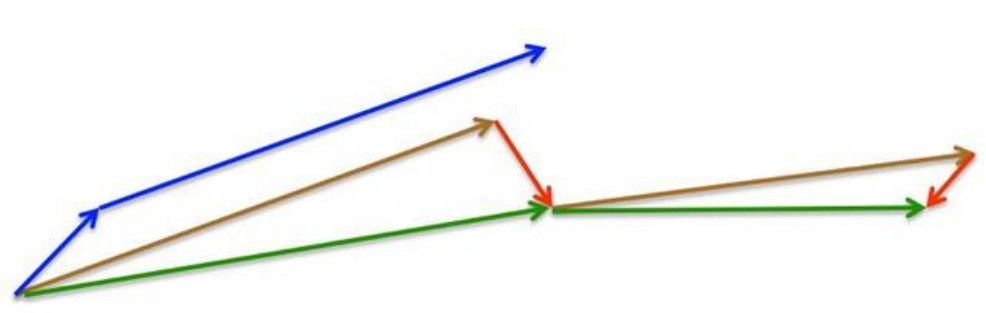


上图是Momentum的优化轨迹，下图是NAG的优化轨迹:



**2.算法公式**

NGA算法是一个对冲量算法的改进算法，其求梯度并非求解当前位置的梯度，而是应该 求解下一个时刻的梯度：



Momentum梯度法首先计算的是当前的梯度（图中的小蓝色向量）然后沿着更新的累积梯度的方向来一个大的跳跃（图中大蓝色向量），而NAG梯度法首先沿着先前的累积梯度方向（棕色向量）实现一个大的跳跃，然后加上一个小的按照动量梯度法计算的当前梯度（上图红色向量）进行修正得到上图绿色的向量。上图为什么画了两个三角形？如果能理解第二个矢量三解形的意义，才能正在理解NAG。注意第二个矢量三角形的棕色向量与前一个的绿色向量方向一致，因为上一个矢量三角形的结果是绿色向量，而棕色代表的是先前的累积梯度，方向就应该和绿色的一样。然后，再加上当前按照动量梯度法计算出的梯度，就得到第二个三角形的绿色向量。公式如下：



相当于一般的冲量算法是根据当前的梯度决定运动方向。而NGA算法则相当于看一下前方的梯度，再决定运动方向。

**3.代码实现**

#include <iostream>

#include <string>

#include <vector>

using namespace std;

double r = 0.001; //学习率

double e = 0.000001;  //精度

double b = 0.9;

double qd(double w,double v);

int main()

{

    double wcur = 0;

    double wper = -1;

    double vcur = 2;

    double vper = 1;

    while(wcur-wper > e)

    {

        double J = qd(wper,vper);

        double temv = vcur;

        double temw = wcur;

        vcur = b \* vper + r \* J;

        wcur = wper +  vcur;

        vper = temv;

        wper = temw;

    }

    cout << wcur;

    system("pause");

    return 0;

}

double qd(double w,double v) //以函数y=2\*x^2+4\*x+5为例

{

    return -(4 \* (w+b\*v) + 4);

}

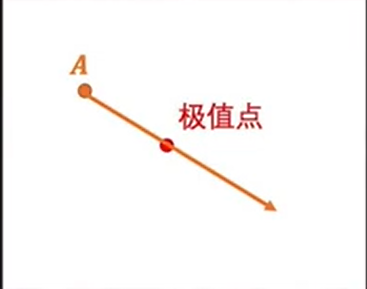
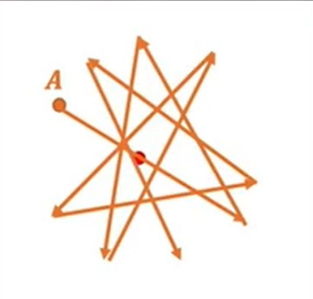


由结果可知程序所求结果与真实值十分接近，且运行时速度较快。

**三、自适应梯度AdaGrad下降法**

**1.背景及理解**

如果学习率是一个定值，那么他在搜索时有可能在极值点处无法收敛（如图一），导致在极值点附近反复震荡（如图二），无法找到极值点。



图一 图二

因此我们希望学习率能够根据自己的情况自动调节。AdaGrad是靠前面的历史数据使学习率能够达到自适应。Adagrad自适应地为各个参数分配不同的学习率：对于不常见的参数可以进行大幅度的调整，对于常见参数则进行小幅度的微调

1. **算法公式**

如果历史数据修改的多，学习率就减少的越多，代表的是一个极小量，是为了分母为0.

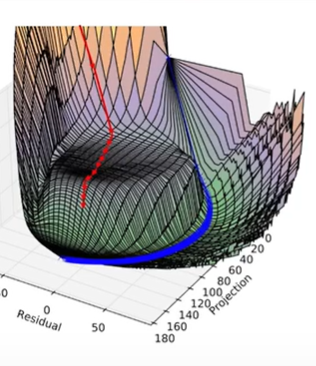


1. **特点**

下图3中白色为AdaGrad实现，紫色为动量法实现。可以发现动量法因为惯性原因会来回震荡，路径较长，而AdaGrad法是在下降的过程中不断修正，去朝向变化较多的方向修正的多，变化较少的方向修正的少，以找到比较好的方向。

AdaGrad方法适合稀疏数据。

该算法的缺点是，分母中需要计算每个参数梯度的累计平方和，由于每次均累加一个正数，训练阶段累积和会持续增加，导致训练后期的学习率非常小，以至更新时不能从当前的梯度获取任何有用信息。Adagrad算法会导致学习率越来越小，导致训练过早结束



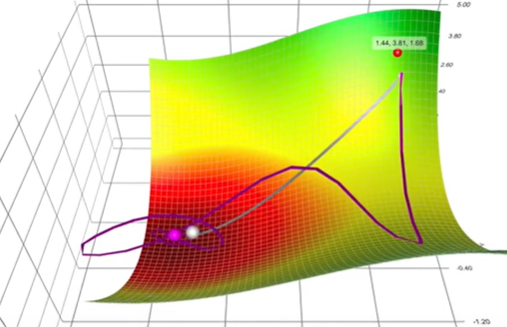


图3 图4

该算法也不适合4图的形式（先急剧变化后进入平台期，再急剧下降）。在平台期及之后会变化的非常慢，因为他需要考虑所有的历史数据。

1. **代码实现**

#include <iostream>

#include <string>

#include <cmath>

using namespace std;

double r = 0.01; //学习率

double e = 0.00001;  //精度

double b = 0.0000000000001;

double qd(double w);

int main()

{

    double wcur = 0;

    double wper = -1;

    double scur = 2;

    double sper = 1;

    while(wcur-wper > e)

    {

        double J = qd(wper);

        double tems = scur;

        double temw = wcur;

        scur = sper + J\*J;

        wcur = wper - ((r/(sqrt(scur+b)))\*J);

        sper = tems;

        wper = temw;

    }

    cout << wcur;

    system("pause");

    return 0;

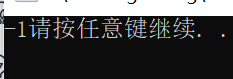
}

double qd(double w) //以函数y=2\*x^2+4\*4+5为例

{

    return 4 \* w + 4;

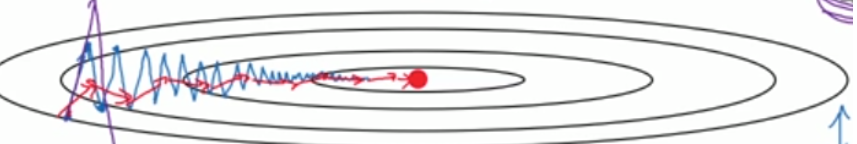
}



可以看出结果精度非常高，且运行时速度也很快。

1. **RMSProp算法**
2. **背景及理解**

针对AdaGrad进行优化，解决学习速率过快衰减的问题。学习率的修正不考虑过远的历史数据，只考虑较近的历史数据。对于上面的这个椭圆形的抛物面（图中的椭圆代表等高线），沿着横轴收敛速度是最快的，所以我们希望在横轴（假设记为w1）方向步长大一些，在纵轴（假设记为w2）方向步长小一些。这时候可以通过RMSprop实现。



1. **算法公式**



观察上面的公式可以看到，s是对梯度的平方做了一次平滑。在更新w时，先用梯度除以，相当于对梯度做了一次归一化。如果某个方向上梯度震荡很大，应该减小其步长；而震荡大，则这个方向的s也较大，除完之后，归一化的梯度就小了；如果某个方向上梯度震荡很小，应该增大其步长；而震荡小，则这个方向的s也较小，归一化的梯度就大了。因此，通过RMSprop，我们可以调整不同维度上的步长，加快收敛速度。

1. **程序实现**

**#**include <iostream>

#include <cmath>

using namespace std;

double r = 0.0001; //学习率

double e = 0.00001;  //精度

double b = 0.0000000000001;

double s = 0.999;

double qd(double w);

int main()

{

    double wcur = 0;

    double wper = -1;

    double scur = 2;

    double sper = 1;

    while(wcur-wper > e)

    {

        double J = qd(wper);

        double tems = scur;

        double temw = wcur;

        scur = s\*sper + (1-s)\*J\*J;

        wcur = wper - ((r/(sqrt(scur)+b))\*J);

        sper = tems;

        wper = temw;

    }

    cout << wcur;

    system("pause");

    return 0;

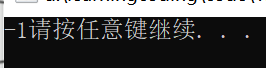
}

double qd(double w) //以函数y=2\*x^2+4\*4+5为例

{

    return 4 \* w + 4;

}



可以看出结果精度很高，且运行速度很快。

1. **Adam法**
2. **背景及理解**

将RMSPro法和动量法结合起来，可以计算每个参数的自适应学习率，也为其增加了动量项公式。Adam算法相当于先把原始梯度做一个指数加权平均，再做一次归一化处理，然后再更新梯度值。与其他自适应学习率算法相比，Adam算法收敛速度更快，而且可以纠正其他优化问题，比如学习率消失、收敛速度过慢

**2.算法公式**



**3.程序实现**

#include <iostream>

#include <cmath>

using namespace std;

double r = 0.0001; //学习率

double e = 0.00001;  //精度

double b = 0.0000000000001;

double s = 0.999;

double t = 0.9;

double qd(double w);

int main()

{

    double wcur = 0;

    double wper = -1;

    double scur = 2;

    double sper = 1;

    double vper = 2;

    double vcur = 3;

    while(wcur-wper > e)

    {

        double J = qd(wper);

        double tems = scur;

        double temw = wcur;

        double temv = vcur;

        scur = s\*sper + (1-s)\*J\*J;

        vcur = t\*vper + (1-t)\*J;

        wcur = wper - ((r/(sqrt(scur)+b))\*vcur);

        sper = tems;

        wper = temw;

        vper = temv;

    }

    cout << wcur;

    system("pause");

    return 0;

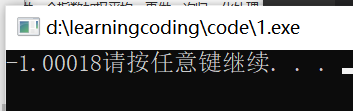
}

double qd(double w) //以函数y=2\*x^2+4\*4+5为例

{

    return 4 \* w + 4;

}



由结果可知，精度较高，运行速度较快。

1. 各种梯度下降法的区别

**梯度下降法的缺点：**

**（1）靠近极小值时收敛速度减慢，如下图所示；**

**（2）直线搜索时可能会产生一些问题；**

**（3）可能会“之字形”地下降。**