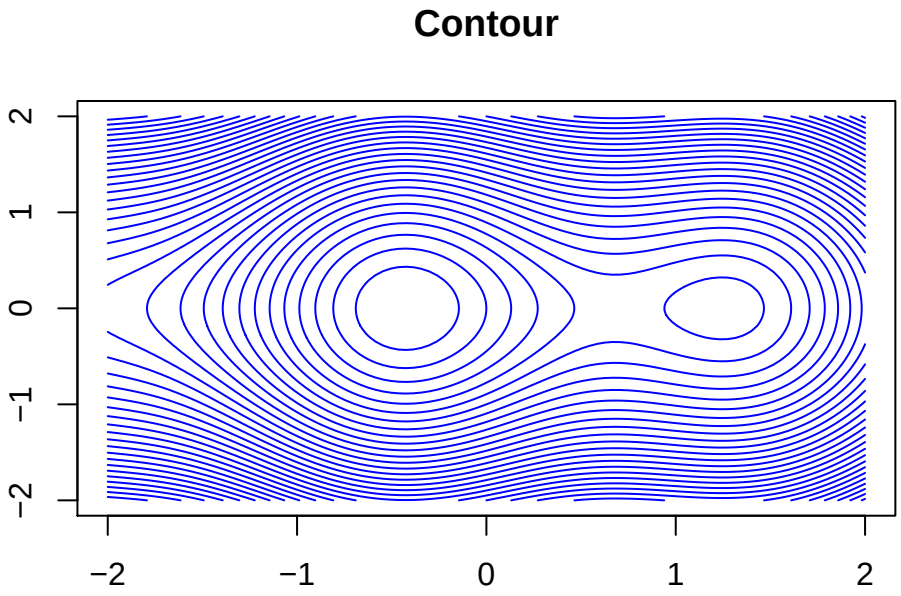
Gradient Descent Optimization

GPK

2018.12.9

1 二元函数选择

选取的二元函数为 $f(x) = \frac{1}{2}x_1^2 + \frac{3}{2}\sin(3x_1) + x_2^2$. 该函数在原点附近有全局最小值、局部极小值和鞍点。作目标函数等高线图如下：



2 优化方法比较

所有优化算法的 R 代码都在 gradientDescent.R 中。

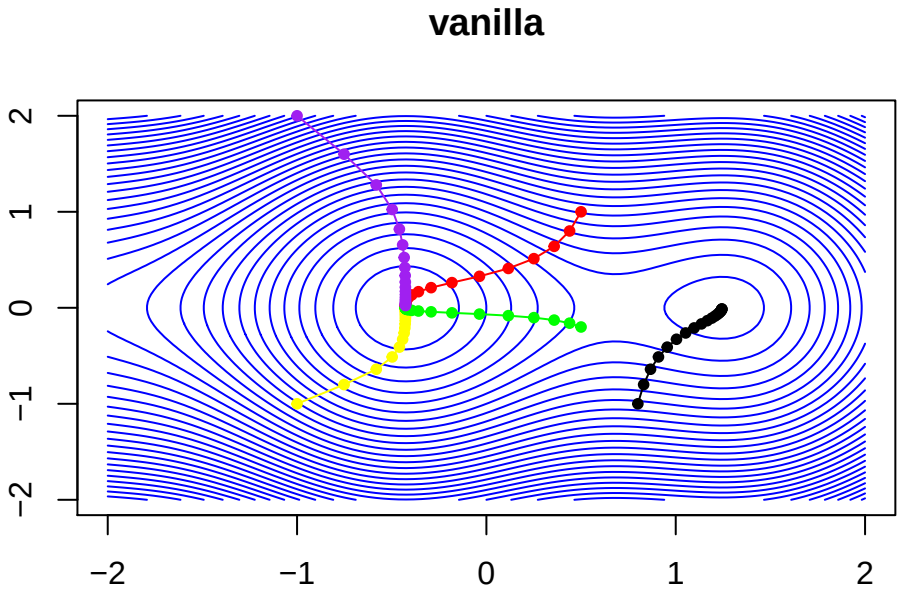
2.1 Vanilla Gradient Descent

基本思路：

多元函数 $f(x)$ 在 x 处下降最快的方向为 $-\nabla f(\boldsymbol{x})$，取正数 η 作为学习率，其梯度下降的迭代公式为：

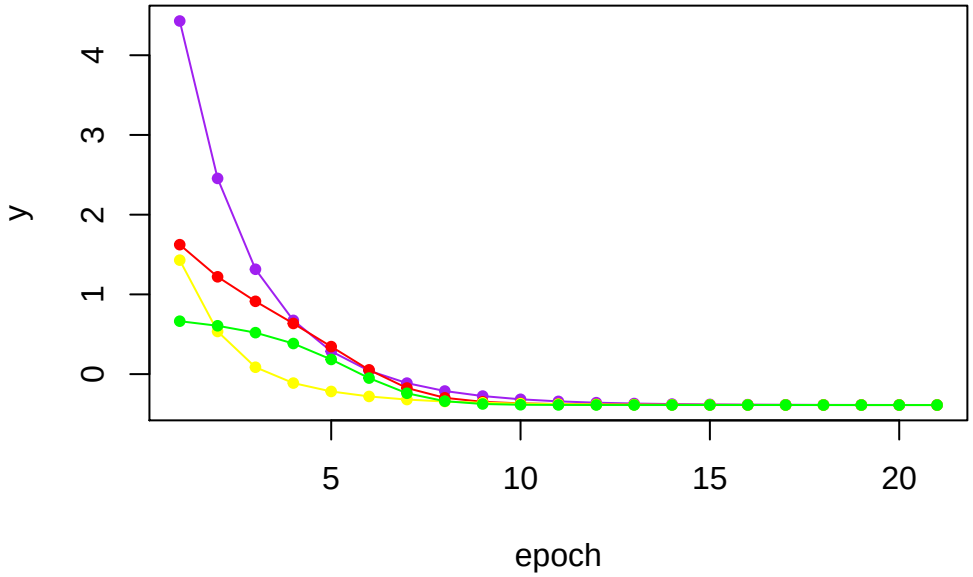
$$\boldsymbol{x} \leftarrow \boldsymbol{x} - \eta \nabla f(\boldsymbol{x})$$

从以上等高线图中选取初始点，以 0.1 的学习率进行梯度下降：



上图中一共进行了 20 次迭代，除了黑色轨迹，其他轨迹基本都收敛到了全局最小值。绿色轨迹起始点在鞍点附近，最终也迭代到了全局最小。

目标函数下降过程（颜色和上图对应，不包括迭代到鞍点的曲线）：

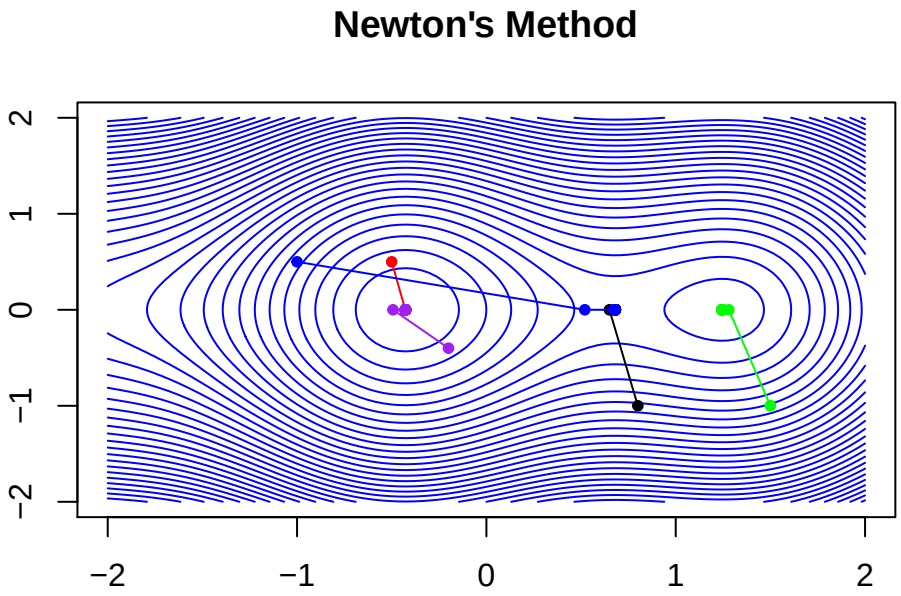


2.2 Newton's Method

迭代公式：

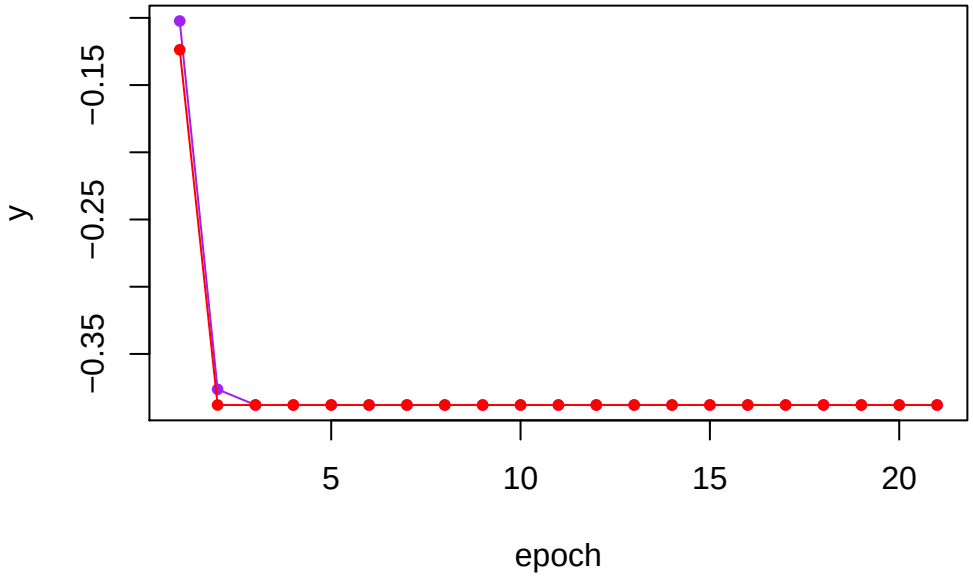
$$\boldsymbol{x} \leftarrow \boldsymbol{x} - H^{-1}(x)\nabla f(\boldsymbol{x})$$

等高线图上的下降过程：



上图中一共进行了 20 次迭代，首先可见虽然下降速度非常快，符合前面的讨论，但牛顿法也很容易落入 saddle point/local minima 中。

目标函数值下降过程（颜色和上图对应，不包括迭代到鞍点的曲线）：

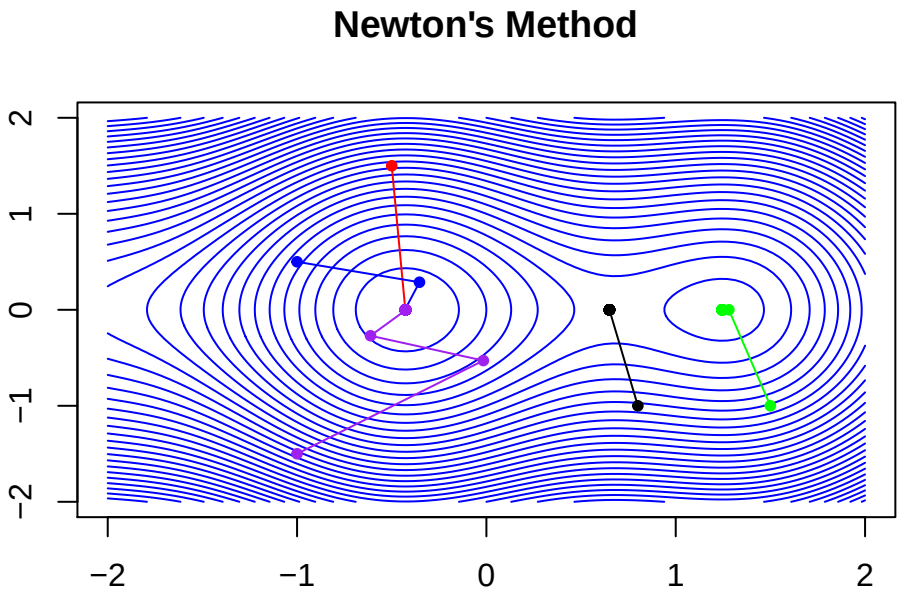


从本质上看，牛顿法是二阶收敛（用二次曲面拟合当前曲面），梯度下降是一阶收敛，因此牛顿法收敛更快。缺点是迭代过程每一步都需要计算 Hessian 矩阵，当多元函数复杂或者不可知时运算量巨大。

2.3 Damped Newton's Method

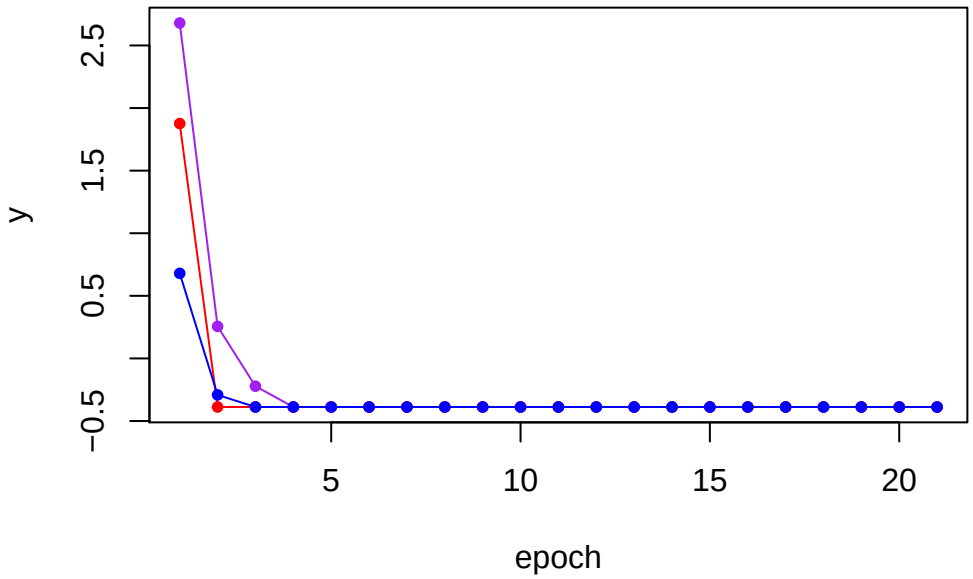
阻尼牛顿法相比牛顿法在迭代过程中增加了线搜索，可以修改迭代的步长，在原步长的基础上进行线性拉伸使目标函数最小。

等高线图上的下降过程：



上图中一共进行了 20 次迭代，可以明显看出线搜索的效果（尤其是紫色轨迹）。但如果改变参数，仍然很容易出现收敛到鞍点和局部极小值的情形。

目标函数值下降过程：



2.4 Conjugate Gradient

共轭梯度法是介于最速下降法与牛顿法之间的一个方法，它仅需利用一阶导数信息，但克服了最速下降法收敛慢的缺点，又避免了牛顿法需要存储和计算 Hesse 矩阵并求逆的缺点。其优点是所需存储量小，具有步收敛性，稳定性高，而且不需要任何外来参数。[1] 算法如下：

[1] https://en.wikipedia.org/wiki/Conjugate_gradient_method

```
conjugateGradient = function(x, eps){
    results = x
    r = 10;
    for(k in 0:r){
        if(k == 0){
            p = - dfun(x)
            g = dfun(x)
        }
        lambda = softline(x, p, fun, dfun)
        x = x + lambda*p
        g0 = g
        g = dfun(x)
        if(sqrt(g*g) < eps) break
        if(k==r-1){
            k = 0
        }else{
            alpha = g*g/(g0*g0)
            p = -g + alpha*p
        }
        if(p %*% g){
            k = 0
        }
        results = rbind(results, x)
    }
    print(dfun(x))
    return(results)
}
```

等高线图上的下降过程:

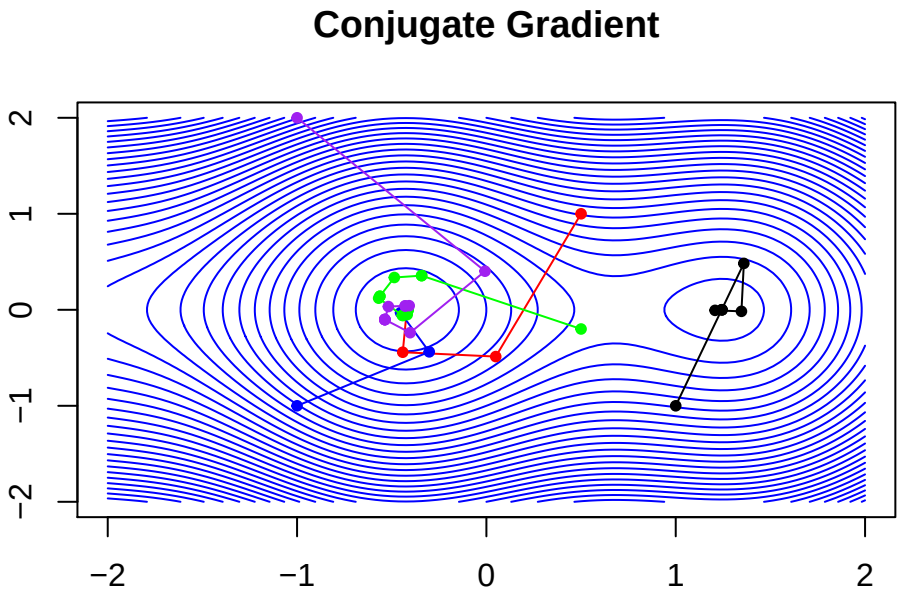
```
## [1] 3.075271e-07 1.562050e-06
```

```
## [1] 3.149465e-06 2.667977e-05
```

```
## [1] -3.798200e-05  4.217291e-06
```

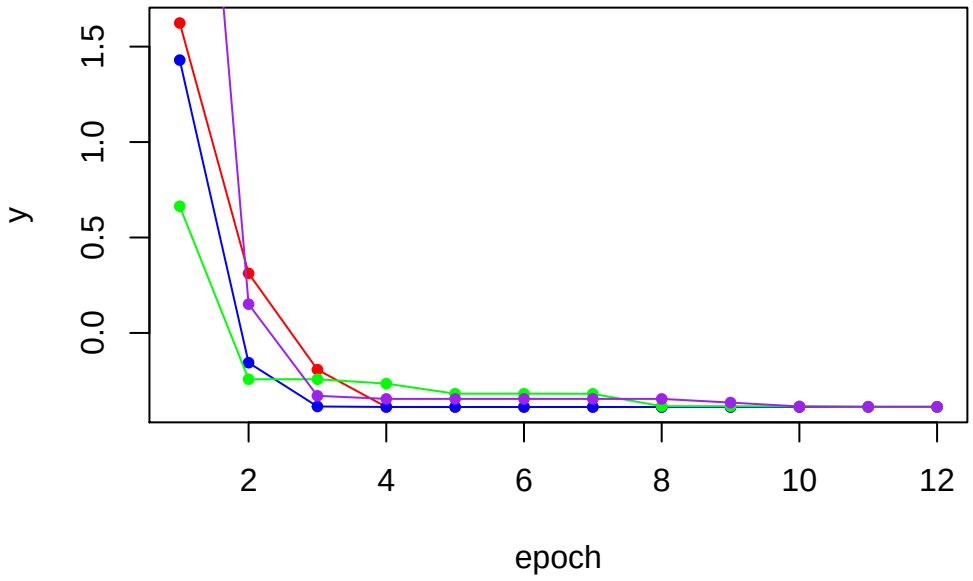
```
## [1]  0.071780604 -0.006729471
```

```
## [1] -0.009730278  0.080359315
```



图中没有体现出的是，共轭梯度法并不是迭代次数越多精度越高，在实际应用中对终止条件有比较高的要求。

目标函数值下降过程：



2.5 Momentum

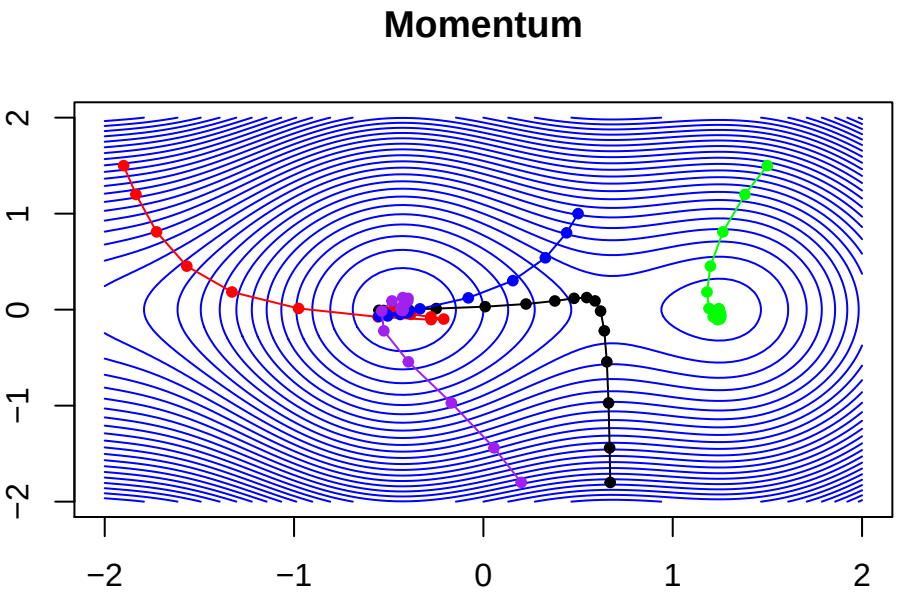
在使用梯度下降迭代时，常出现目标函数不同方向下降速度差异很大的情况，这时就容易导致自变量在某些方向上振荡甚至发散（学习率过大时）。动量法的引入解决了上述问题，其原理是使用指数加权平均的步长，在每个时间步的自变量更新量近似于将前者对应的最近 $1/(1-\gamma)$ 个时间步的更新量做了指数加权移动平均后再除以 $1-\gamma$。所以动量法中，自变量在各个方向上的移动幅度不仅取决当前梯度，还取决于过去的各个梯度在各个方向上是否一致。

迭代公式：

$$\boldsymbol{v}_t \leftarrow \gamma \boldsymbol{v}_{t-1} + \eta_t \boldsymbol{g}_t,$$
$$\boldsymbol{x}_t \leftarrow \boldsymbol{x}_{t-1} - \boldsymbol{v}_t$$

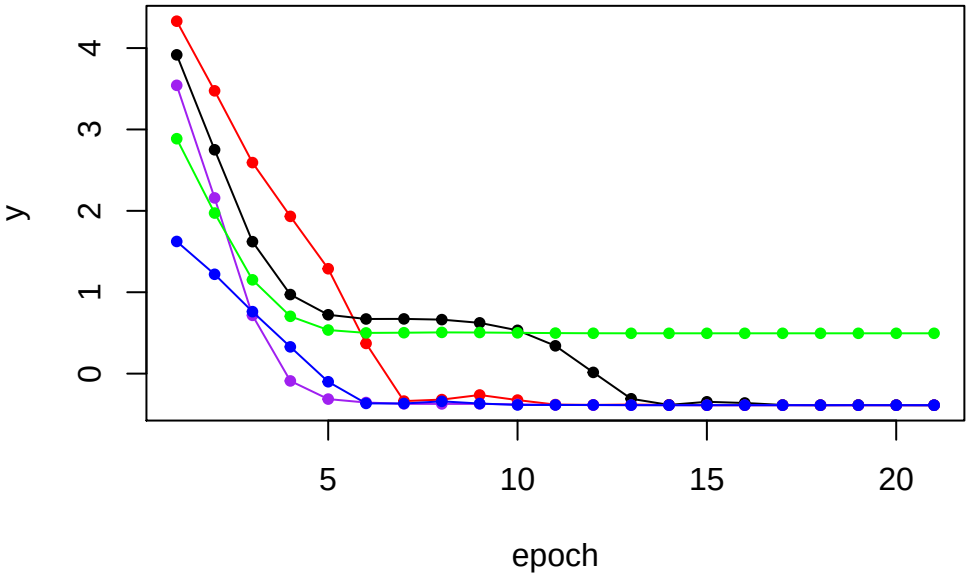
其中，动量 hyperparameter γ 满足 $0 \leq \gamma < 1$。当 $\gamma = 0$ 时，动量法等价于梯度下降。

等高线图上的下降过程：



局部极小值是难以避免的，但相比牛顿法等二阶方法，不容易落到鞍点。

目标函数值下降过程（绿色为局部极小值）：

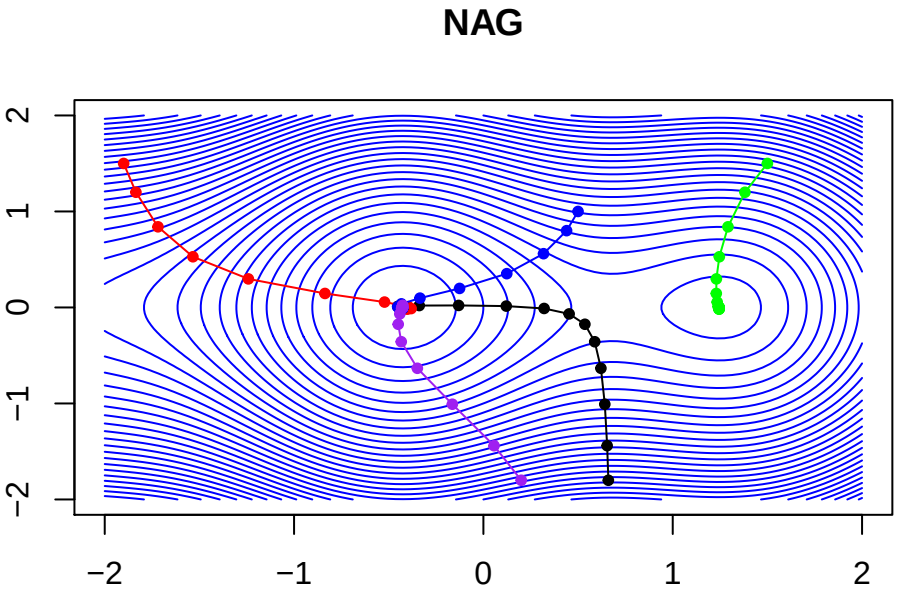


2.6 Nesterov Accelerated Gradient

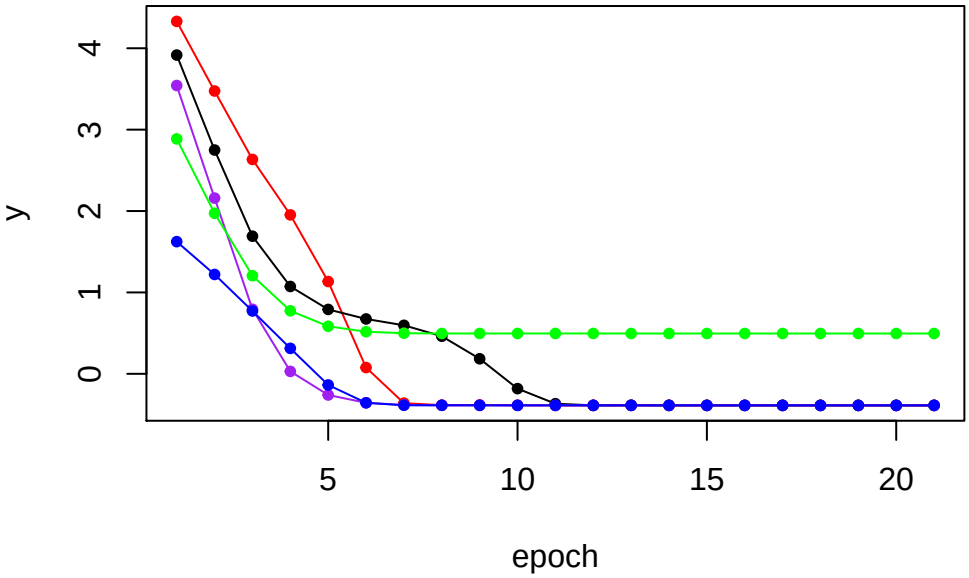
NAG 是在动量法基础上改进得到的。NAG 在 Momentum 计算步长的步骤中，不计算当前位置的梯度方向，而是计算“按照累积动量方向迭代一步后的下降方向”。迭代公式：

$$\boldsymbol{v}_t \leftarrow \gamma \boldsymbol{v}_{t-1} + \eta_t \nabla(x_t - \gamma v_{t-1})$$
$$\boldsymbol{x}_t \leftarrow \boldsymbol{x}_{t-1} - \boldsymbol{v}_t$$

等高线图上的下降过程：



目标函数值下降过程（绿色为局部极小值）：



从等高线图可以看出，和 Momentum 下降轨迹相比，NAG 的梯度计算方式使得整体下降过程更加平滑连贯，下降速度也有一定的提升（黑色、红色曲线）。

2.7 Adagrad

在之前的 GD、Momentum 及其变种中，目标函数自变量的每一个元素在相同时间步都使用同一个学习率来自我迭代。通常为了使函数在梯度较大的维度上不发散，需要选择比较小的学习率。这又会使得自变量在梯度较小的维度上迭代过慢。对此，动量法的解决方式是让自变量的更新方向更加连贯。而 Adagrad 及以下几种算法的解决方式是为目标函数的每个维度调整不同的学习率。

算法[2]：

Adagrad 的算法会使用一个小批量随机梯度 $\boldsymbol{g}_t$ 按元素平方的累加变量 $\boldsymbol{s}_t$。在时间步 0，adagrad 将 $\boldsymbol{s}_0$ 中每个元素初始化为 0。在时间步 t，首先将梯度 $\boldsymbol{g}_t$ 按元素平方后累加到变量 $\boldsymbol{s}_t$：

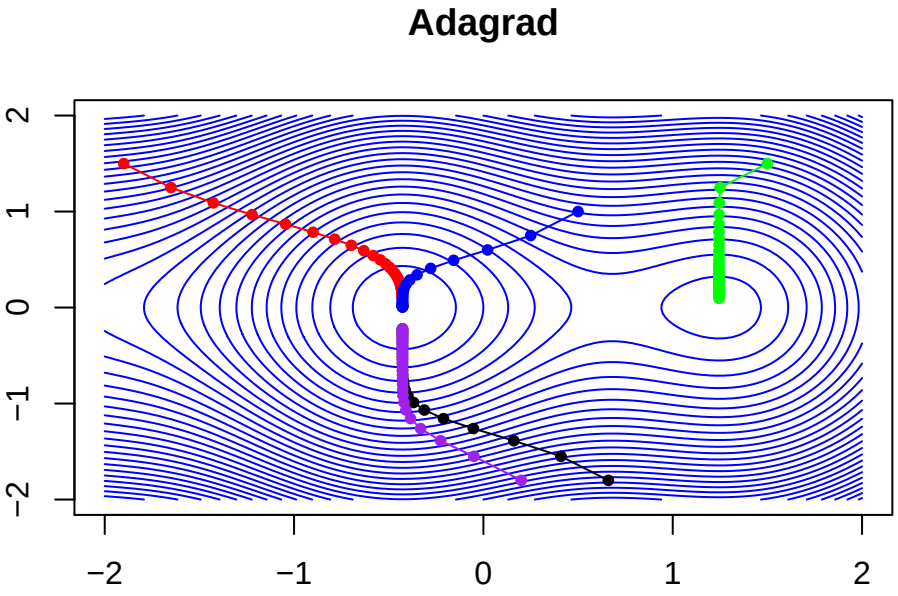
$$\boldsymbol{s}_t \leftarrow \boldsymbol{s}_{t-1} + \boldsymbol{g}_t \odot \boldsymbol{g}_t,$$

其中 $\odot$ 是按元素相乘。接着，我们将目标函数自变量中每个元素的学习率通过按元素运算重新调整一下：

$$\boldsymbol{x}_t \leftarrow \boldsymbol{x}_{t-1} - \frac{\eta}{\sqrt{\boldsymbol{s}_t + \epsilon}} \odot \boldsymbol{g}_t,$$

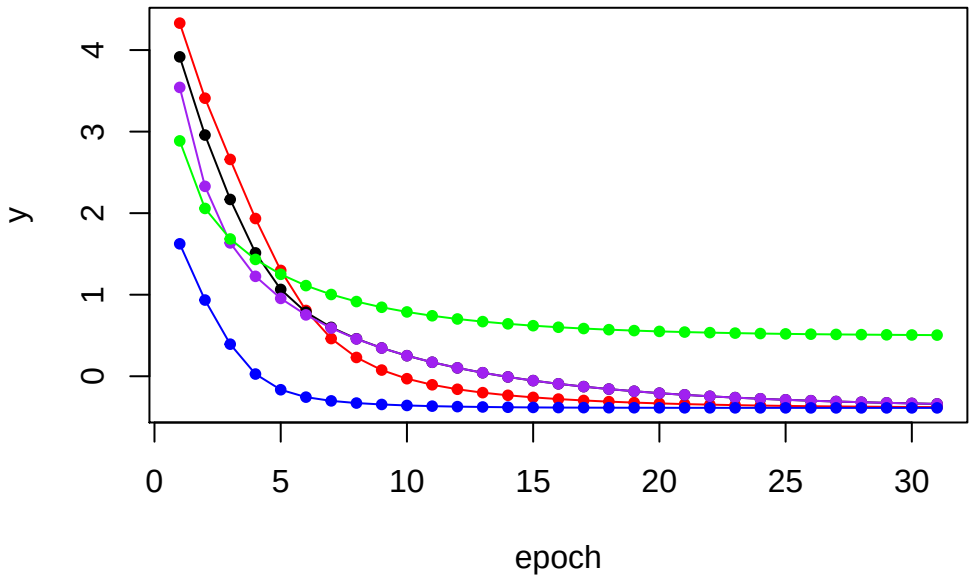
其中 η 是学习率，ϵ 是为了维持数值稳定性而添加的常数，例如 10^{-6}。这里开方、除法和乘法的运算都是按元素进行的。这些按元素运算使得目标函数自变量中每个元素都分别拥有自己的学习率。

等高线图上的下降过程：



目标函数值下降过程（绿色为局部极小值）：

[2] 《动手学习深度学习》，李沐



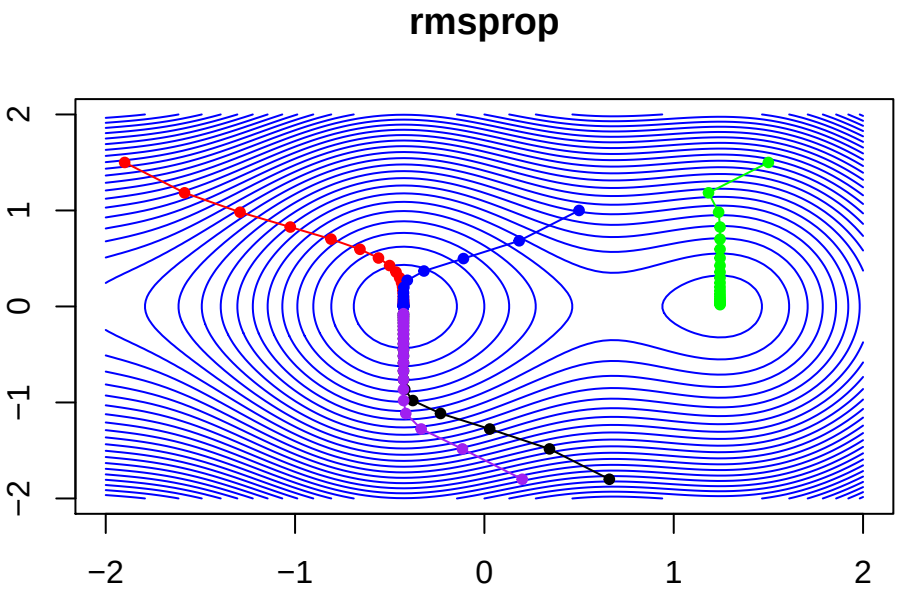
等高线图中可以明显看出越往后下降速度越慢，表现在自变量越来越密集。这里使用了 0.25 的学习率，和之前的方法相比，某些出发点的最终迭代结果仍然没有完全下降到最小值。由此可见 Adagrad 的缺陷是，在步长计算中分母根号内的参数 s_t 是之前梯度平方的累加，导致整个学习率分式单调递减。如果学习率在早期下降过快，而目标函数仍然没有到达极小值附近，Adagrad 可能因后期移动幅度太小而无法到达期望的解。

2.8 Adadelta/rmsprop

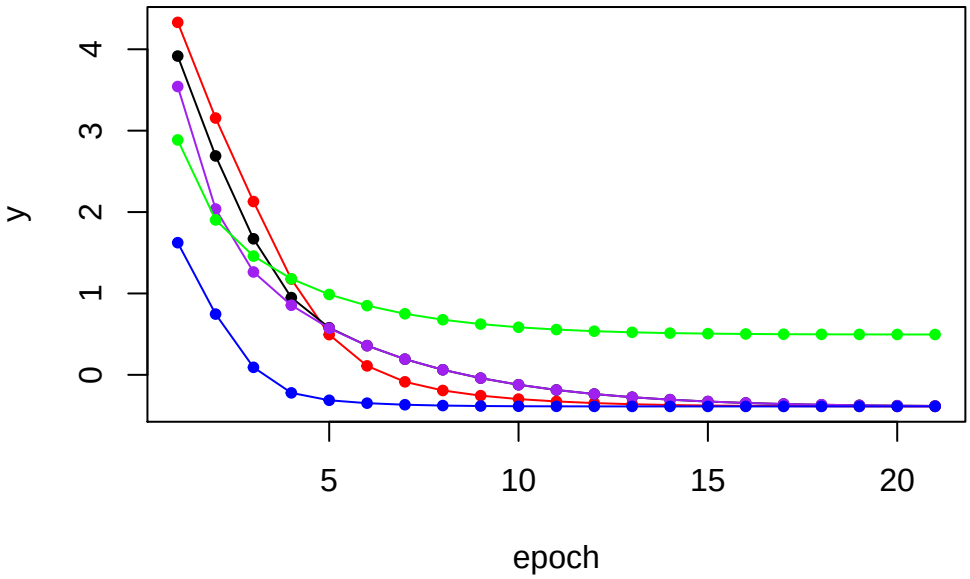
为了解决上述分母单调递增导致后期移动幅度过小的问题，rmsprop 将自适应学习率的计算公式改为指数加权移动平均：

$$\boldsymbol{s}_t \leftarrow \gamma \boldsymbol{s}_{t-1} + (1-\gamma)\boldsymbol{g}_t \odot \boldsymbol{g}_t.$$

其中 $\odot$ 表示按元素平方。如此一来，每个变量的学习率不再单调下降或停滞不前。



目标函数值下降过程（绿色为局部极小值）：



这里采用 0.1 的学习率，γ 按照原文献采用 0.9，从等高线图可以看出移动步长不会像 Adagrad 那样衰减，在较少的 epoch 内即可收敛。

2.9 Adam

Adam 加入了 Momentum，并保留了自适应梯度（每个元素学习率不同）。迭代算法如下，其中超参数 β_1, β_2 的区间均为 $(0, 1)$，原文献的建议值为 $\beta_1 = 0.9, \beta_2 = 0.999$。

$$\boldsymbol{v}_t \leftarrow \beta_1 \boldsymbol{v}_{t-1} + (1 - \beta_1)\boldsymbol{g}_t.$$

$$\boldsymbol{s}_t \leftarrow \beta_2 \boldsymbol{s}_{t-1} + (1-\beta_2)\boldsymbol{g}_t \odot \boldsymbol{g}_t.$$

$$\hat{\boldsymbol{v}}_t \leftarrow \frac{\boldsymbol{v}_t}{1-\beta_1^t},$$

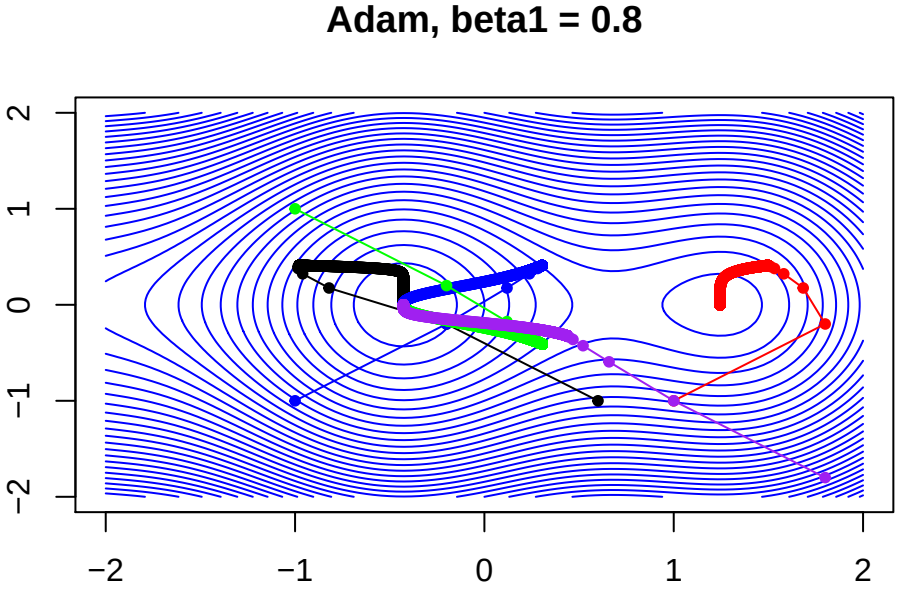
$$\hat{\boldsymbol{s}}_t \leftarrow \frac{\boldsymbol{s}_t}{1-\beta_2^t}.$$

$$\boldsymbol{g}'_t \leftarrow \frac{\eta \hat{\boldsymbol{v}}_t}{\sqrt{\hat{\boldsymbol{s}}_t + \epsilon}},$$

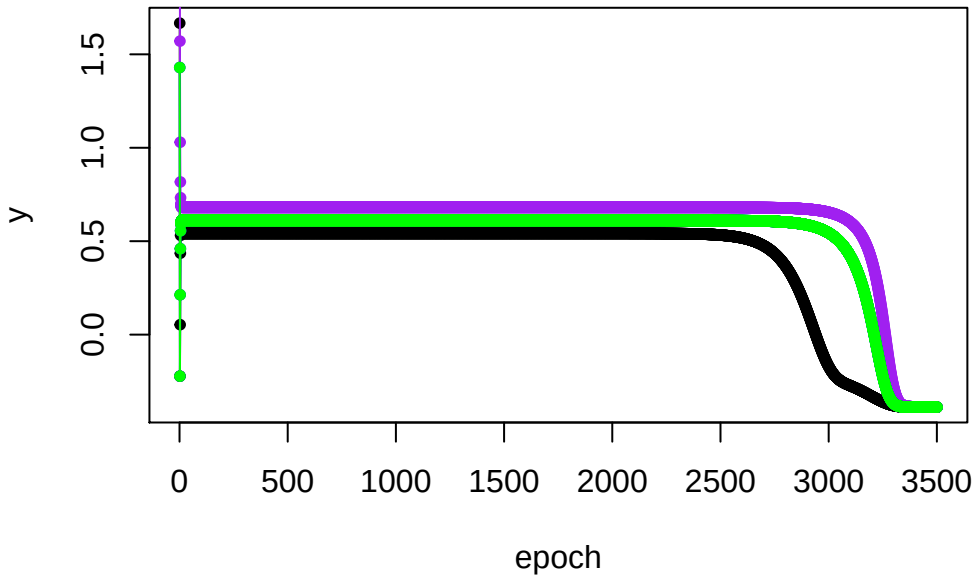
$$\boldsymbol{x}_t \leftarrow \boldsymbol{x}_{t-1} - \boldsymbol{g}'_t.$$

其中 η 是学习率，ϵ 是为了维持数值稳定性而添加的常数，例如 10^{-8}。

按照原文献中推荐的超参数 $\beta_1 = 0.9$，每一步学习率的分子有 90% 受前面的 v 决定，效果不佳。参数调整为 0.8 后：



目标函数值下降过程：



最终可以收敛，但需要较大的迭代次数。出现这样结果的一个原因是迭代过程中遇到了目标函数的大片平坦区域，因为局部梯度太小、经过指数校正后移动较慢。另外，在代码中使用的梯度是 batch=1 的当前自变量梯度，实际运用中使用小批量随机梯度时应该可以提升收敛速度。

3 参考文献

- https://www.cnblogs.com/ljy2013/p/5129294.html
- https://blog.csdn.net/garfielder007/article/details/50292447
- https://zhuanlan.zhihu.com/p/22810533
- 《动手学习深度学习》李沐