第3章　神经网络的优化难题

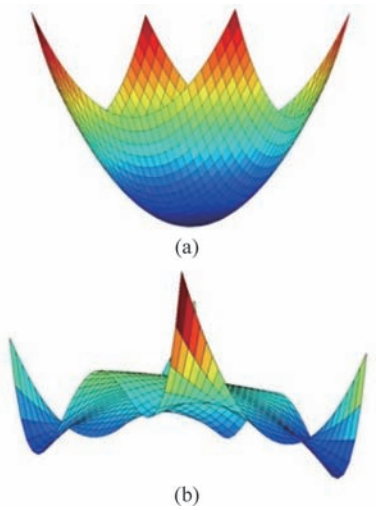
即使我们可以利用反向传播来进行优化，但是训练过程中仍然会出现一系列的问题，比如鞍点、病态条件、梯度消失和梯度爆炸，对此我们首先提出了小批量随机梯度下降，并且基于批量随机梯度下降的不稳定的特点，继续对其做出方向和学习率上的优化。

3.1　局部极小值，鞍点和非凸优化

正如我们在第 2 章讨论的，基于梯度的一阶和二阶优化都在梯度为零的点停止迭代，梯度为零的点并非表示我们真的找到了最佳的参数，更可能是局部极小值或者鞍点，在统计学习的大部分问题中，我们似乎并不关心局部极小值和全局最小值的问题，这是因为统计学习的损失函数经过设计是一个方便优化的凸函数，会保证优化问题是一个凸优化问题。

在凸优化问题中，比如最小二乘和线性约束条件下的二次规划，参数空间的局部最小值必定是全局最小值。但对于神经网络这样复杂的参数空间，损失函数就不再是一个凸函数，如图 3.1，非凸函数的局部极小值可能与全局极小值相去甚远，那么在理论上就无法保证一定会找到全局极小值。

但是我们并不用担心这样的问题，优化停止在局部极小值也是非常困难的，因为在高维参数空间中，局部极小值的海森矩阵必须是正定的，也就是说每个维度上的二阶导数都必须为正，要陷入真正的局部极小值也是很困难的。我们可以假设某一维度的二阶导数为正的概率为 s，那么在一个 d 维的参数空间的，找到局部极小值概率就是 s^d，可以看出局部极小值随着参数空间维度的增加，概率指数级下降。另一方面，目前的主流观点认为，局部极小值也具有小的损失函数，优化的目的只需要将损失函数降到足够低的水平，所以即便找到了局部极小值，但是损失函数已经降低到了足够低的水平，也是可以接受的。从理论上来说，真正容易陷入的是鞍点，鞍点的存在条件更为宽松，因为它在各个维度上二阶导数有



■图 3.1　损失函数在二维参数空间的可视化，(a)为典型的凸函数，(b)为非凸函数

正有负。虽然有实验表明，基于梯度下降的算法可以逃离鞍点，但在理论上并无保证，面对更广泛的场景，单纯的梯度下降对于鞍点的表现仍然是一个需要证明的问题。

3.2　随机梯度下降的优势

一个好的优化算法，一方面我们希望它迭代更新的次数越少越好，最好能一步到位；另一方面我们希望计算的代价要小，也就是利用的信息要少。但这两者是无法兼得的，单纯的梯度下降只需要计算梯度，但在每一步下降只局限于大小为 ε 的局部方向，迭代更新的次数较多，而牛顿法这样的二阶优化，利用了海森矩阵包含的曲率信息，迭代更新的每一个点都可以保证是极值点或者鞍点，使得一般情况下的迭代更新次数更少，但是每次迭代时需要计算海森矩阵的逆。

如果我们想在牛顿法的基础上进一步的减小信息量，就可以考虑 BFGS(Broyden-FletcherGoldfalb-Shanno)这种著名的拟牛顿法，它并没有直接计算海森矩阵的逆，而是采取向量的乘积和矩阵的加法来代替了逆的计算，使得计算量进一步下降。但即便这样，深度学习中模型参数和训练数据的庞大，使得耗时仍然较为严重。

转向对梯度下降法的改良时，我们首先会注意到数据集内样本的冗余，很多样本太相似了，产生的损失也是相似的，对梯度估计的贡献也就是相似的，比如说我们只是采取复制的操作去扩充数据集，那么经过一次迭代，计算量翻了一番，但是得到的参数更新却是相同的。

我们完全可以采用少且有效的信息去替代冗余的信息，每次计算梯度时，我们不采用全部的样本，而是选择一个或者一批样本来进行梯度的估计，前者为随机梯度下降(Stochastic Gradient Descent)，后者为小批量梯度下降(mini-batch gradient descent)，在大部分情况

下，人们会混用它们，统一叫作随机梯度下降。它还会带来两个好处：

（1）小批量的梯度下降或者随机梯度下降，即便增加了迭代次数，但计算的代价少了，总体的效率仍然很高。

（2）随机的挑选小批量的样本一定程度上增加了梯度估计的方差（噪声项），反而使得梯度下降有机会跳出局部最小值和鞍点。

那么随机梯度下降的参数更新公式就由原来的：

$$\theta_{i+1}=\theta_i-\varepsilon\nabla_{\theta_i}L(X,\theta,y)$$

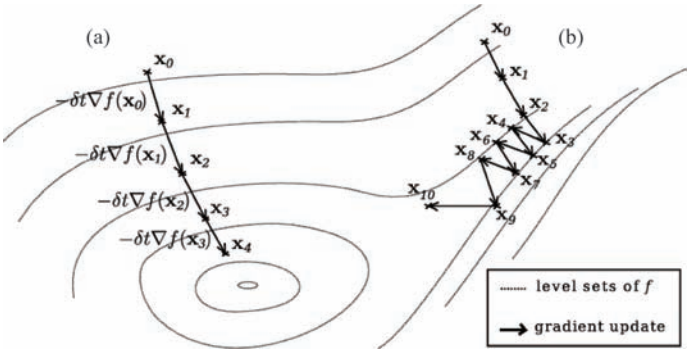
变为了：

$$\theta_{i+1}=\theta_i-\varepsilon\left[\frac{1}{m}\nabla_{\theta_i}\sum_i L(x^1,\theta,y^i)\right]$$

当使用随机梯度下降时，虽然可以更好地逃离局部最小值或者鞍点，但也会存在下降到真正的极小值（如果存在极小值的话），仍然无法停止迭代，但此时的梯度也会变得很小，所以会看到随机梯度下降并不会停留在极小值，而是在其附近微小振荡。另外实验表明，随机梯度下降会在迭代初期性能远远超越使用全部样本的梯度下降。但是，小批量随机梯度下降会引入一个额外的参数，就是批量的大小。小的批量可能会更有助于逃离梯度为零的点，但是迭代的次数却增加了，而大批量的效果恰恰相反，在应用中，它仍然是一个根据实际情况来指定的参数。

3.3 梯度方向优化

虽然梯度是所有方向导数变化最大的，但是因为小批量随机梯度的方差较大，表现为每一次的梯度方向都不相同，多次的迭代都被浪费在了来回移动上面；甚至只是因为初始值的选取，使得在迭代一开始所产生的梯度就无法朝向最佳的参数。如图 3.2，初始值和小批量使得每一步的梯度下降方向并不一致，而是在损失函数的多个 contour 之间来回移动。本质上，这是因为梯度下降是一个贪心算法。



■图 3.2　梯度下降算法的表现，(a)为理想情形，(b)为可能的实际情况

改善该问题的方法之一就是在执行梯度下降的时候，将每一步的下降都尽量积累在同一个方向上。动量(Momentum)算法引入了一个量叫作速度，用来积累以前的梯度信息：

$$\nu = \mu\nu - \varepsilon\left[\frac{1}{m}\nabla_{\theta_i}\sum_i L(X,\theta^i,y)\right] \tag{3.1}$$

$$= \mu\nu - \varepsilon g(\theta^i) \tag{3.2}$$

这种做法是出于物理上的考虑，损失函数的负梯度就相当于物理上的势的负梯度，我们把它叫作力，根据矢量加法，就可以直观地看出，累加的结果就是把每一次力的相同方向加大，而相反的方向相互抵消，使得速度在每次力的相同方向都越来越大，本质上就是对梯度的方向做了归一化，然后我们选择利用速度去更新参数：

$$\theta_{i+1} = \theta_i + \nu$$

让我们来考虑极端的情况，如果每次的梯度的方向一致，那么 ν 就会沿着一个方向越来越大，参数 μ 叫作动量因子，用来调节过去积累梯度与现在梯度的比重，μ 越大，那么过去的信息占的比重越大，本质上是一种加权移动平均法。参数的更新幅度不再单单只取决于当前的梯度，而是决定于当前梯度和历史梯度的加权平均。

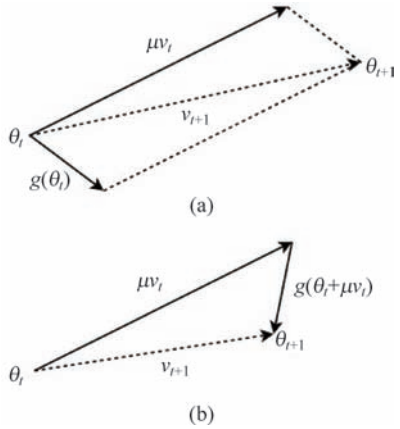
同时，我们也可以在梯度计算之前就在参数中添加速度项：

$$\nu = \mu\nu - \varepsilon\left[\frac{1}{m}\nabla\theta_i\sum_i L(X,\theta^i + \alpha\nu,y)\right] \tag{3.3}$$

$$= \mu\nu - \varepsilon g(\theta^{i+1}) \tag{3.4}$$

这样就得到了 Nesterov 动量算法，简单来看只是添加的步骤不同，并且在循环中，标准的动量算法迟早都会执行参数更新后的梯度，事实上，两者有着根本的不同。

如果我们做一个简单的矢量加法，如图 3.3，它们两者的主要区别在于梯度的计算，当我们需要更新 θ^{i+1} 时，Nesterov 算法给出的更新方向是基于 $g(\theta^{i+1})$，表明在累积梯度方向信息时，会把当前参数梯度信息也计算进去，显得更为合理。



■图 3.3　(a)为动量法的参数更新示意，(b)为 Nesterov 动量的参数更新示意

3.4 动态调整学习率

动量算法一定程度上加速了梯度下降，但是却引入了另一个超参数 μ，更重要的是，实践发现，同样作为超参数，α 远远没有学习率 ε 重要。在第 2 章，利用泰勒展开式对梯度下降的表现作出了评估，在曲率特别大的地方，固定学习率可能会导致损失上升，我们就无法迭代到极小值。

所以我们希望动态地调整学习率，使得学习率在梯度很大的时候，变得小一些，不至于错过极小值，在梯度很小的时候变得大一点，使得在损失平坦的地方走得更快。这就是所谓的自适应学习率算法。

首先我们可以设想，优化算法是可以良好工作的，这意味着随着迭代次数的增加，它会越来越接近极小值，学习率应该越来越小，不容易错过极小值。所以很直接想法就是直接设置学习率随迭代次数的衰减(learning rate decay)，比如指数衰减：

$$\varepsilon_t = \varepsilon_0 \beta^t \tag{3.5}$$

上式的 β 小于 1，为衰减率，t 为迭代次数。此外我们也可以设置其他的函数来描述这一衰减的过程。可是，学习率的指数衰减率一般都要事先指定，我们进一步的设想，这一衰减率可否从优化过程中自动学习到，同时每个参数对于梯度下降贡献不同，很多情况下，我们希望在一个参数方向上下降得快一点，而在另一个参数方向上下降得慢一点。

一个自然的想法就是，我们想在损失陡峭的时候走得慢一点，梯度大的时候，学习率应该小一点，也即，偏导数大的参数，学习率应该小一点，同时也想在平坦的区域走得快一点，梯度小的时候，学习率应该大一些。那我们如何知道这是一个平缓或者陡峭的区域呢？AdaGrad 算法使用一个量来累计历史梯度：

$$r = r + g \circ g$$

其中，$\circ$ 是哈达玛积(Hadamard product)，见定义 3.1，这里使用哈达玛积是因为，我们需要分开处理每个维度上的梯度信息。

定义 3.1(哈达玛积) 哈达玛积区别于一般的矩阵乘法，假设 $\boldsymbol{A}$ 和 $\boldsymbol{B}$ 是相同维度的矩阵，$\boldsymbol{A}$ 和 $\boldsymbol{B}$ 的哈达玛积被定义为：

$$(\boldsymbol{A} \circ \boldsymbol{B})_{i,j} = (\boldsymbol{A})_{i,j} (\boldsymbol{B})_{i,j} \tag{3.6}$$

接着，在学习率上乘以梯度的倒数，表示学习率与梯度的反比关系：

$$\varepsilon \to \frac{\varepsilon}{\delta + \sqrt{r}} \tag{3.7}$$

其中 δ 是非常小的常数，避免分母为零。它简单实现了动态调整学习率衰减的目标。但它积累了全部的历史梯度，使得学习率过早下降，可能永远无法迭代到最佳值。一种改进的方法就是，给不同时期的梯度赋予权重因子，使得较早的梯度不重要，而较近的梯度更重要：

$$r = \alpha r + (1 - \alpha) g \circ g$$

参数 α 是权重因子，用来调节历史梯度和当前梯度的权重。这样就得到了 RMSProp 算法。在此基础上，我们希望将动量算法这种针对梯度方向的优化和 RMSProp 这种自适应调节学习率的算法结合起来，结合两者的优点，相当于对动量算法提供的“速度”提供了修正。

回顾上一节的内容，动量算法其实就是将每次的速度作为参数的更新，即：

$$\Delta\theta = \nu \tag{3.8}$$

我们将 RMSProp 对于学习率的更新加入，使得更新的速度可以动态调整：

$$\Delta\theta = \frac{\varepsilon}{\delta + \sqrt{r}}\nu \tag{3.9}$$

这就得到了 Adam 算法的基本形式。动量使用的是梯度的一阶信息，而自适应学习率算法使用了哈达玛积，也就是梯度的二阶信息，所以有些书中会把 Adam 算法称为一阶矩和二阶矩结合起来的算法。

3.5 使用 keras

我们在第 2 章的代码中已经详细讨论了一维参数空间中梯度下降和牛顿法的表现，并且也看到了学习率和海森矩阵对于学习算法的影响，在本章我们延续上一节的思路，来讨论基于梯度下降算法的各类优化算法。首先我们构造一个简单的损失函数：

$$\mathcal{L}(\theta) = \theta_1^2 + \theta_2^2 \tag{3.10}$$

此时的损失是两个参数的函数，并且它是一个凸函数，对其使用梯度下降算法：

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.cm as cm
import seaborn as sns

def f(x,y):
    return x**2+y**2
def partial_x(x,y):
    return 2*x
def partial_y(y,x):
    return 2*y

def GD(lr, start, iterations):
    x, y= start[0],start[1]
    GD_x, GD_y ,GD_z= [], [],[]
    for it in range(iterations):
        GD_x.append(x)
        GD_y.append(y)
        GD_z.append(f(x,y))
        dx = partial_x(x,y)
        dy = partial_y(y,x)
        x = x - lr * dx
```

```
        y = y - lr * dy

    return(GD_x, GD_y, GD_z)

def plot_track(learning_rate, iterations):
    GD_x, GD_y, GD_z = GD(lr = learning_rate, start = [15,0.1],
        iterations = iterations)
    a = np.linspace(-20,20,100)
    b = np.linspace(-20,20,100)
    A,B = np.meshgrid(a,b)
    sns.set(style = 'white')
    plt.contourf(A, B, f(A,B), 10, alpha = 0.8, cmap = cm.Greys)
    plt.scatter(GD_x, GD_y, c = 'red', alpha = 0.8, s = 20)
    u = np.array([GD_x[i+1] - GD_x[i] for i in range(len(GD_x)-1)])
    v = np.array([GD_y[i+1] - GD_y[i] for i in range(len(GD_y)-1)])
        plt.quiver(GD_x[:len(u)], GD_y[:len(v)], u, v, angles = 'xy', width = 0.005, \
        scale_units = 'xy', scale = 1 ,alpha = 0.9, color = 'k') plt.xlabel('x')
    plt.ylabel('y')
    plt.title('learning  rate:{}
        iterations:{}'.format(round(learning_rate,2),iterations))
    plt.show()

plot_track(learning_rate = pow(2, -7)*16, iterations = 100)
```

如图 3.4，梯度下降在简单的二维参数空间上快速迭代，它说明了梯度确实是下降最快的方向，并且在这样的凸函数中，两个参数是对称的，我们无论从哪一个初始点出发，只要使用相同的学习率，那么参数更新轨迹总是从起始点直接到达目标点。

图 3.4　梯度下降在二维参数空间的表现

一个有趣的想法就是我们可以通过设置参数学习率不同来改变其运动轨迹，因为参数具有对称性，梯度必然保持一致，根据梯度下降算法的参数更新公式 $\Delta\theta = -\varepsilon\nabla L$，影响更新大小就只有学习率，所以我们将参数的学习率设置为不同：

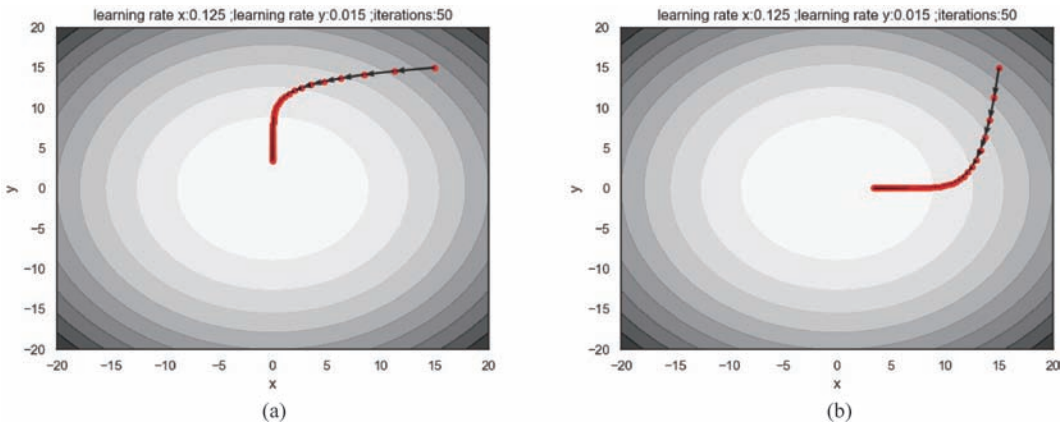
```
.....
    def GD(lr, start, iterations):
    x, y = start[0],start[1]
    GD_x, GD_y ,GD_z = [], [],[]
    for it in range(iterations):
        GD_x.append(x)
        GD_y.append(y)
        GD_z.append(f(x,y))
        dx = partial_x(x,y)
        dy = partial_y(y,x)
        x = x - lr[0] * dx
        y = y - lr[1]* dy

    return(GD_x,GD_y,GD_z)

.....

plot_track(learning_rate = [pow(2,-7)*16, 0.015], iterations = 50)
plot_track(learning_rate = [0.015, pow(2,-7)*16], iterations = 50)
```

在上段代码中，我们将学习率设置为一个列表，以便于传入不同的数值。如图 3.5，对于对称的参数梯度，学习率的不同会显著影响学习的过程，以(a)为例，变量 x 的学习率稍大一点，所以其更新幅度大于变量 y 的更新幅度，在 x 方向上会迅速迭代到损失对于 x 偏导为零的点，而 y 方向几乎没有什么变化。此时，x 方向因为偏导为零，就不会发生迭代，反而在 y 方向上的偏导不为零，所以后期的轨迹就是沿着 y 方向缓慢迭代。(b)恰恰相反。



■图 3.5　(a)为变量 x 的学习率稍大一点的结果，(b)为变量 y 的学习率稍大一点儿的结果

事实上，我们可以将这个例子完全反过来，在一般的损失函数中，在学习率相同的情况下，两个参数的梯度不同，造成参数的更新幅度也并不相同，所以我们经常看到使用梯度下降会在参数空间迂回前进，这也同样解释了随机梯度下降的迂回前进，因为每个 batch 所估计的梯度也不相同，一定程度上加剧了迂回的程度，所以会看到 batch 越小，损失函数会在前期振荡得越剧烈。

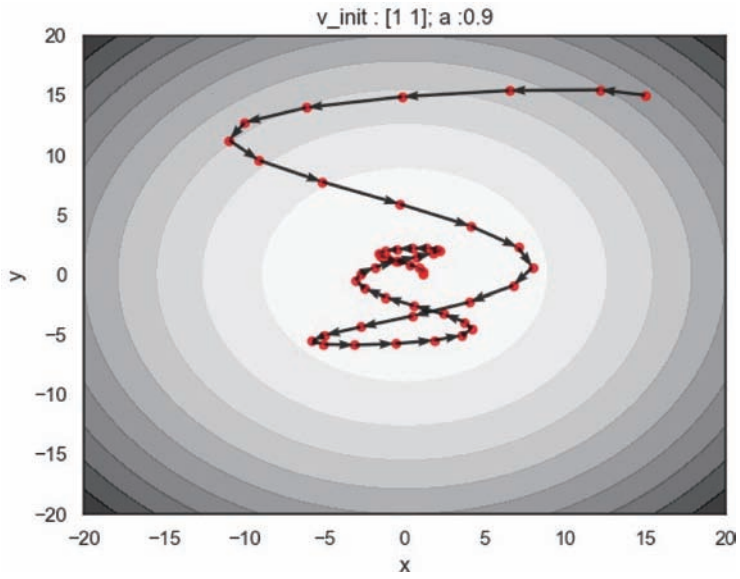
所以在这里，相同参数梯度下学习率的差异与相同学习率下参数梯度的差异是等价的。因为参数更新方向的差异，我们对更新轨迹的方向尝试做优化，考虑动量法，构建算法为：

```
.....
def Momentum(lr, a, v_init, start, iterations):
    x, y = start[0], start[1]
    v = v_init
    M_x , M_y, M_z = [], [], []
    for it in range(iterations):
        M_x.append(x)
        M_y.append(y)
        M_z.append(f(x, y))
        v = a * v - [partial_x(x, y) * lr[0], partial_y(y, x) * lr[1]]
        x = x + v[0]
        y = y + v[1]
    return(M_x, M_y, M_z)
.....
```

在差异化的学习率下调用动量算法，并将动量因子设置为 0.9，这表明当前参数的更新会极大地依赖于历史梯度如图 3.6，一开始的梯度较大，所以参数在逐渐靠近极小值的时候速度并不为零，所以并不会立刻停下来，在远离极小值的时候速度逐渐减小，最终停留在极小值点。所以从图中可以看出，更新轨迹就像牛顿力学下的粒子在一个碗中滑动。

同时我们想得到用于参数更新的速度随迭代的变化，那么我们修改动量算法返回累计的速度信息，并且绘制出两个方向的速度与迭代次数的关系：

```
def Momentum(lr, a, v_init, start, iterations):
    x, y = start[0], start[1]
    v = v_init
    M_x , M_y, M_z, M_v = [], [], [], []
    for it in range(iterations):
        M_x.append(x)
        M_y.append(y)
        M_z.append(f(x, y))
        M_v.append(v)
        v = a * v - [partial_x(x, y) * lr[0], partial_y(y, x) * lr[1]]
        x = x + v[0]
        y = y + v[1]
    return(M_x, M_y, M_z, M_v)
```



■图 3.6　动量因子为 0.9，初始速度为(1,1)，动量算法的更新轨迹

```
def plot_moment_v(learning_rate, iterations, v_init, a):
    M_x, M_y, M_z, M_v = Momentum(lr = learning_rate, a = a,
        v_init = v_init, start = [15, 15], iterations = iterations)
    plt.figure()
    plt.plot(range(iterations),[i[0] for i in  M_v],label = 'x', linewidth
        = 3)
    plt.plot(range(iterations),[i[1] for   i   in   M_v],label = 'y', linewidth
        = 3)
    plt.xlabel('iterations')
    plt.ylabel('$\nv$')
    plt.title('Momentum v at $\mu = $ {}'.format(a))
    plt.legend()
    plt.show()

plot_moment_v(learning_rate = [pow(2, -7) * 16, 0.015], a = 0.9, v_init = np.array([1,1]),
    iterations = 80)
```

如图 3.7，x 方向和 y 方向的速度具有相同的初值，所以它们从同一个起点出发。由于 x 方向的学习率较大，所以迅速产生了较大的速度，随着迭代的进行，越来越靠近极值点，梯度逐渐减小，但速度却在不断增大，当梯度的方向变化后，速度才会逐渐减小。所以 x 方向上速度的变化就像牛顿力学下谐振子的速度，极值点附近的速度反而是最大的。y 方向上的速度变化与 x 方向的道理相同，只是幅度较小。

在这个例子中我们可以预想到，因为一开始的梯度较大，所以稍微减小动量因子，使得在靠近极小值的时候速度变小，就可以更快地到达目标点。如图 3.8，设置动量因子为 0.8，

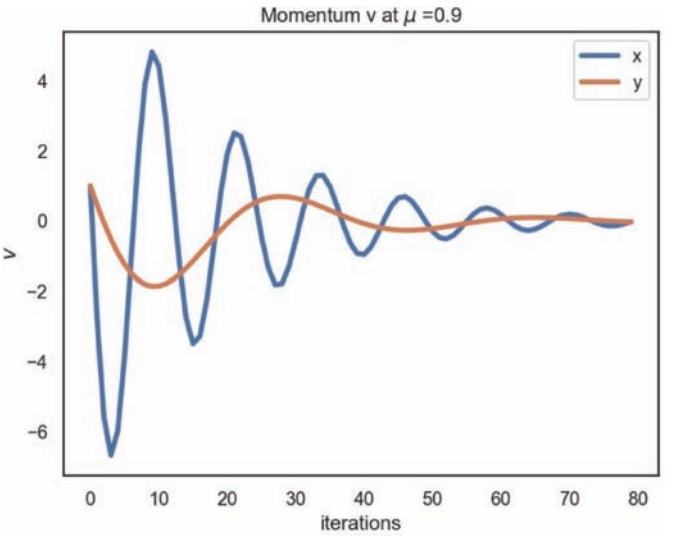


图 3.7　动量因子为 0.9，动量算法速度的变化

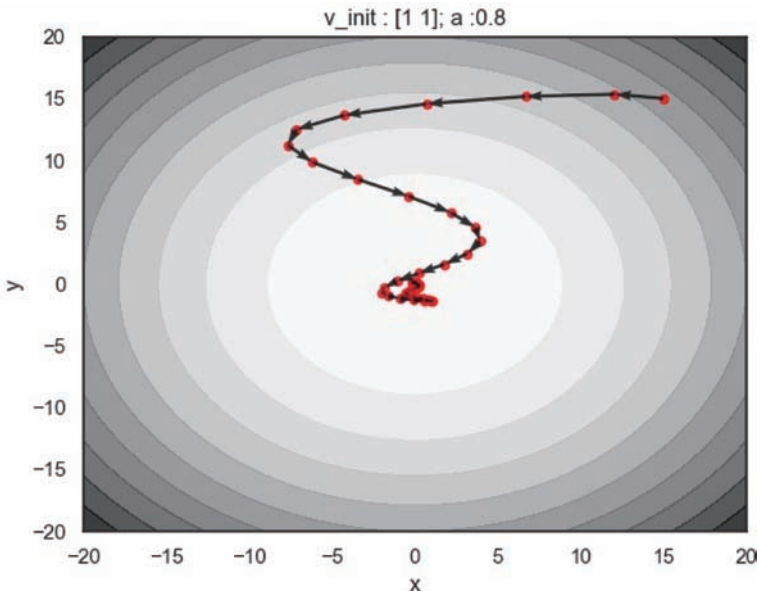


图 3.8　动量因子为 0.8，初始速度为(1,1)，动量算法的更新轨迹

减弱了历史较大梯度的影响，迭代得更快。

另一个角度就是对于学习率本身的动态调整。在使用 RMSprop 算法之前，因为它提供了学习率衰减，所以为了避免学习率过早地衰减到零，我们一开始就将学习率设置的大一点，使得它在初期迭代得快一点；另一方面为了很直观地看到不同参数所积累的梯度信息，我们将初始的参数点设置为(15,7.5)，代码如下：

```
.....
def RMSProp(lr, d, ro, start, iterations):
    x, y = start[0], start[1]
    r = np.array([0, 0])
    RMS_x, RMS_y, RMS_z, RMS_r = [], [], [], []
    for it in range(iterations):
        RMS_x.append(x)
        RMS_y.append(y)
        RMS_z.append(f(x, y))
        RMS_r.append(r)
        g = np.array([partial_x(x, y), partial_y(y, x)])
        r = ro * r + (1 - ro) * g * g
        lr[0] = lr[0] /(d + np.sqrt(r[0]))
        lr[1] = lr[1] /(d + np.sqrt(r[1]))
        x = x - lr[0] * g[0]
        y = y - lr[1] * g[1]
    return(RMS_x, RMS_y, RMS_z, RMS_r)
.....
```

我们在构建 RMSprop 算法时，使用了 numpy 的两个数组直接相乘来得到哈达玛积，结果如图 3.9，即使我们设置了很大的初始学习率，RMSprop 能很快地迭代，而不会产生震荡。但是它并未迭代到最低点就几乎停止了，这是因为一开始的梯度较大，学习率过早衰减，那么也就很难到达目标点。

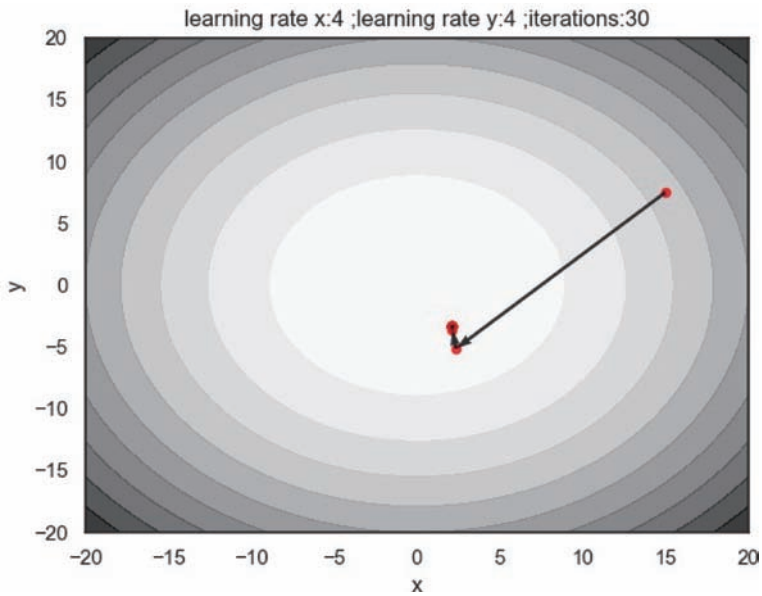


图 3.9 $\mu=0.9$ 时，RMSprop 算法的更新轨迹

我们在上述构建的算法函数中还将累计的梯度信息返回，这样是为了方便我们对累积梯度进行观察：

```
def plot_hadamard(learning_rate, iterations, ro, start):

    RMS_x, RMS_y,RMS_z, RMS_r = RMSProp(lr = learning_rate, d = 1e-6, ro = ro,\
        start = start, iterations = iterations)
    plt.figure()
    plt.plot(range(iterations), [i[0] for i in RMS_r], label = 'x',
        linewidth = 3)
    plt.plot(range(iterations), [i[1] for i in RMS_r], label = 'y',
        linewidth = 3)
    plt.xlabel('iterations')
    plt.ylabel('$ g \circ g $')
    plt.title('gradient hadamard product')
    plt.legend()
    plt.show()

plot_hadamard(learning_rate = [pow(2, -2) * 16, pow(2, -2) * 16 ], \
            iterations = 30, ro = 0.9, start = [15, 7.5])
```

如图 3.10，x 方向的参数梯度的哈达玛积随着迭代先增加后减少，这说明一开始的梯度比较大，在该方向上参数产生了一个较大的更新以后，梯度迅速减小。由于使用了 0.9 的移动平均来积累梯度信息，后面的梯度如果小于前面梯度的$\frac{1}{10}$，梯度就会缓慢减小。y 方向的参数梯度的哈达玛积随着迭代缓慢增加，这意味着一开始的梯度较小，并且算法在该方向上迭代的次数更多。这一现象与参数初始点(15,7.5)是对应的，损失函数在初始点在 x 方向上具有更大的偏导。

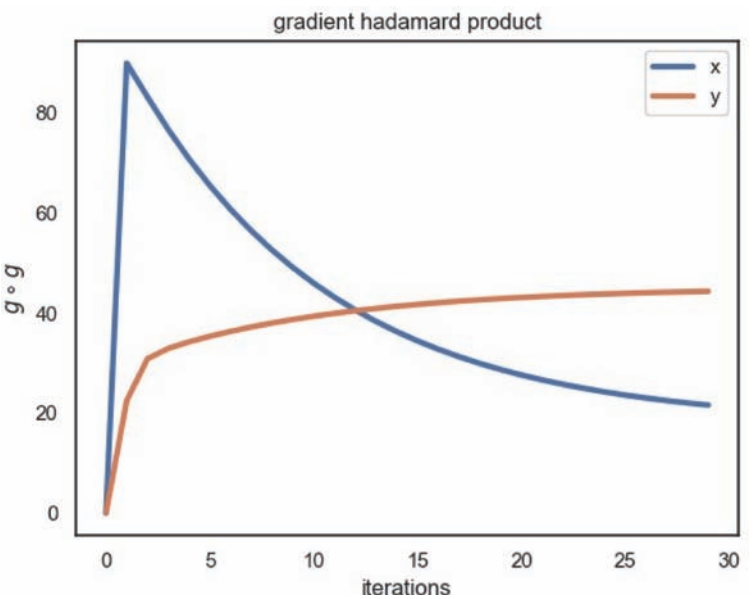


图 3.10　$\mu=0.9$ 时，RMSprop 算法下不同参数梯度累积随迭代的变化