梯度下降法

梯度下降法可以解决对可微函数的优化，当目标函数为凸函数时，可保证梯度下降法的解是全局最优解，否则其解不能保证是全局最优解

优化方法的思路及步骤

梯度下降法的思路是利用函数的一阶导数信息，通过向负梯度方向移动一段距离，使得函数值向着低点的方向移动

公式如下

$$x^{(k+1)} = x^{(k)} - \alpha \bigtriangledown J(x)$$

梯度下降法的分类

批量梯度下降法（Batch Gradient Descent）

每次更新参数时使用所有的样本进行更新

随机梯度下降（Stochastic Gradient Descent）

随机采样一个样本对参数进行更新

批量梯度下降法（Mini-batch Gradient Descent）

每次更新参数时用一部分样本

针对梯度下降法的优化

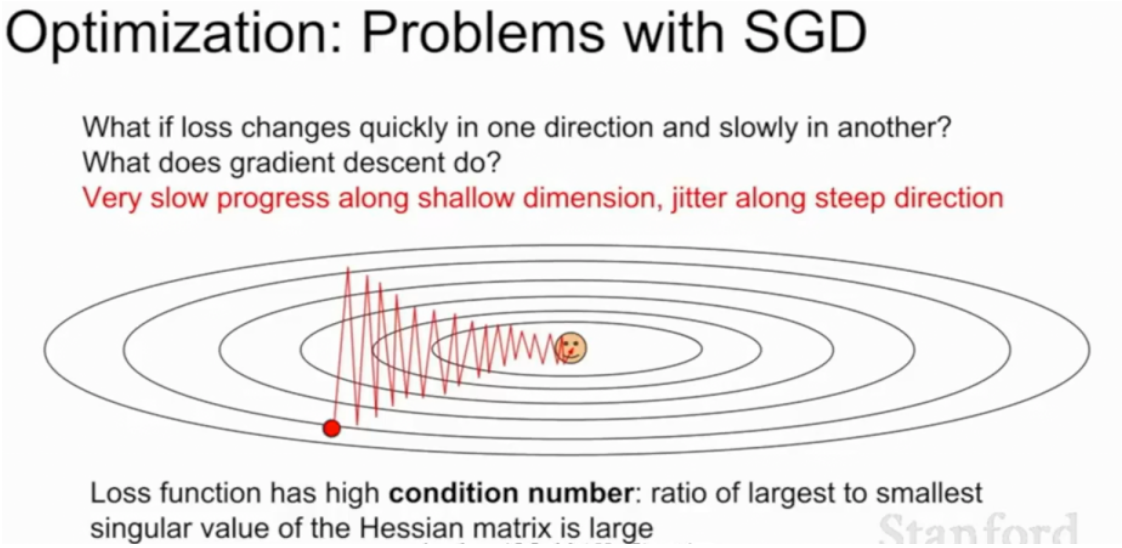
以下图及代码源自cs231n

最基础的算法随机梯度下降法

```
while True:
    dx = computer_gradient(x)
    x += learning_rate * dx
```

随机梯度下降法的缺点：

1. 如下图，在某个方向（图中为竖直方向）很敏感，而其他方向不敏感，因此损失呈之字形移动，而这让训练变得很慢



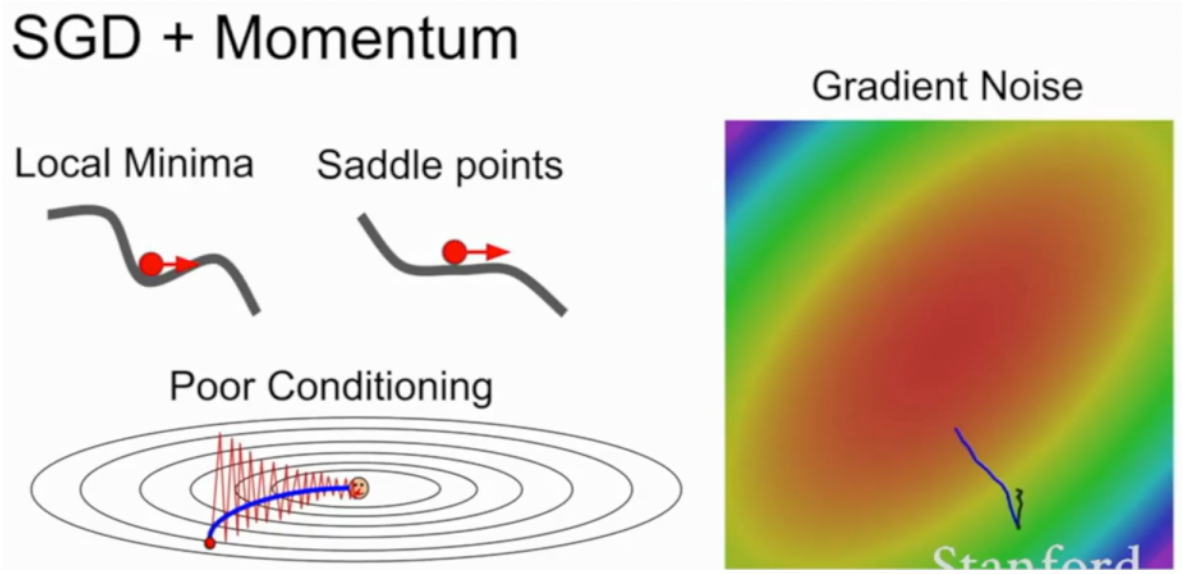
2. 在某个局部最小点卡住，或者在鞍点附近缓慢甚至停止移动。在高维空间中，鞍点的问题出现的可能性更大。

动量优化法

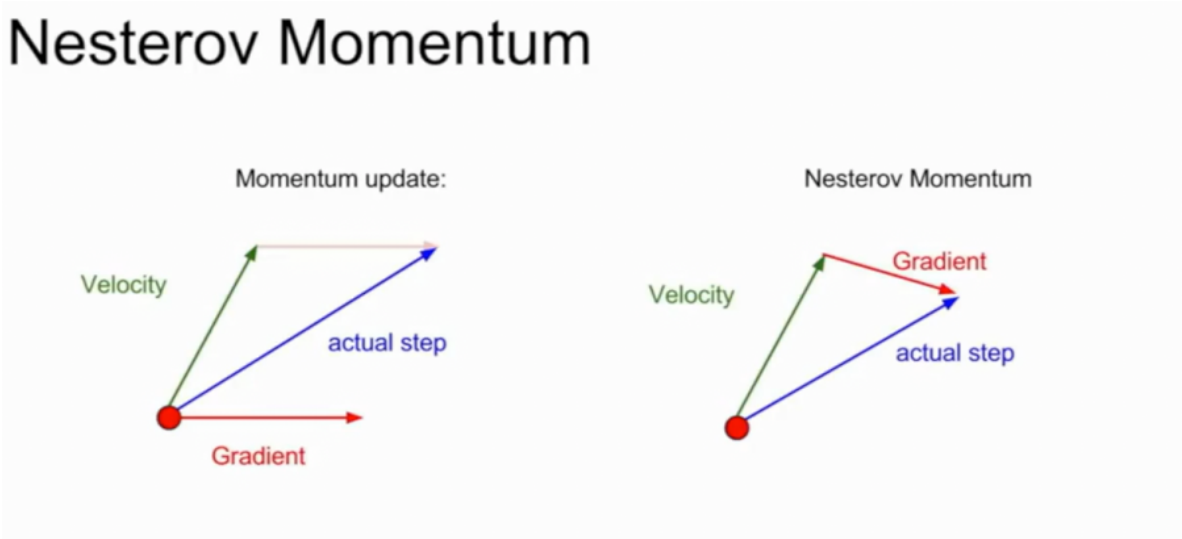
```
vx = 0
while True:
    dx = computer_gradient(x)
    vx = rho * vx + dx
    x += learning_rate * vx
```

保持一个不随时间变化的速度，将梯度估计添加到这个速度上，然后在这个速度的方向上步进，而不是在梯度的方向上步进

超参数ρ，摩擦系数，一般取较大的数字



如图，加上速度后，可避免在局部最小点或鞍点卡住，因为即使那个地方梯度很小，依然有一个速度向量推动其前进



Nesterov Momentum

另外一种加动量的方式

$$v_{t+1} = \rho v_t - \alpha \nabla f(x_t + \rho v_t)$$
$$x_{t+1} = x_t + v_{t+1}$$

$$let \ \ \tilde{x}_t = x_t + \rho + p v_t \ \ then$$
$$v_{t+1} = \rho v_t - \alpha \nabla f(\tilde{x}_t)$$
$$x_{t+1} = \tilde{x}_t - \rho v_t + (1+\rho) v_{t+1}$$
$$= \tilde{x}_t + v_{t+1} + \rho(v_{t+1} - v_t)$$

```
v = 0
while True:
    dx = computer_gradient(x)
    old_v = v
    v = rho * v - learning_rate * dx
    x += -rho * old_v + (1 + rho) * v
```

缩放优化

AdaGrad

```
grad_squared = 0
while True:
    dx = computer_gradient(x)
    grad_squared += dx * dx
    x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7) # 1e-7保证不会除于一个
0
```

在有高梯度的轴，由于`np.sqrt(grad_squared) + 1e-7`项的存在，使得在这个轴上运动的速度变慢

相应的，在低梯度的轴，这会让在这个轴上的运动速度加快

因为`grad_squared`一直在增加，所以训练的步长会随着时间而减少，但这也有可能会让训练在一个局部最小点卡住，因此有RMSProp解决这个问题

RMSProp

```
grad_squared = 0
while True:
    dx = computer_gradient(x)
    grad_squared = decay_rate * grad_squared + (1 - decay_rate) * dx * dx #
decay_rate 一般为0.9或者0.99
    x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

Adam

动量优化与AdaGrad/RMSProp的结合，一般状况下的首选

```
first_moment = 0
second_moment = 0
while True:
    dx = computer_gradient(x)
    first_moment = beta1 * first_moment + (1 - beta1) * dx # Momentum
    second_moment = beta2 * second_moment + (1 - beta2) *dx * dx #
AdaGrad/RMSProp
    x -= learning_rate * first_moment / (np.sqrt(second_moment) + 1e-7) #
AdaGrad/RMSProp
```

为了避免开始时步长过大，所以有下面的优化

```
first_moment = 0
second_moment = 0
for t in range(num_iterations):
    dx = computer_gradient(x)
    first_moment = beta1 * first_moment + (1 - beta1) * dx # Momentum
    second_moment = beta2 * second_moment + (1 - beta2) *dx * dx #
AdaGrad/RMSProp
    first_unbias = first_moment / (1 - beta1 ** t) # Biad correction
    second_unbias = second_moment / (1 - beta2 ** t) # Biad correction
    x -= learning_rate * first_unbias / (np.sqrt(second_unbias) + 1e-7) #
AdaGrad/RMSProp
```

Adam的超参数一般设置为：beta1 = 0.9，beta2 = 0.999，learning_rate = 1e-3 or 5e-4

梯度下降法的优缺点

优点

- 通过求一阶导的方式逐步靠近最优解，避免函数空间巨大时求解方程的繁琐计算

缺点

- 靠近最优解时收敛速度减慢
- 需要设定学习率，学习率过高会导致震荡，学习率过低会让收敛速度变慢