第四章 数值计算

机器学习算法通常需要大量的数值计算。通常是指通过迭代过程更新解的估计值来解决数学问题的算法，而不是通过解析过程推到出公式来提供正确解的方法。常见的操作包括优化和线性方程组的求解。对数字计算机来说实数无法在有限内存下精确表示，因此仅仅是计算涉及实数的函数也是困难的。

4.1 上溢和下溢

一种极具毁灭性的舍入是 `下溢(underflow)` 。当接近零的数被四舍五入为零时发生下溢。
另一个极具破坏的数值错误形式是 `上溢(overflow)` 。
必须对上溢和下溢数值稳定的一个例子是 `softmax函数(softmax function)` 。softmax函数经常用于预测与Multinoulli分布相关联的概率，定义为：

$$softmax(x)_i = \frac{\exp(x_i)}{\sum_{j=1}^{n} \exp(x_j)}$$

如果x_i等于常数c，所有输出都应该是$\frac{1}{n}$，如果c是很小的负数，$\exp(x_i)$会下溢。意味着softmax分母会变成0，so结果是未定义。当c是非常大的正数时，$\exp(x_i)$的上溢再次使整个表达式未定义。可通过计算softmax(z)解决，其中$z = x - \max x_i$，减去$\max x_i$导致exp的最大值为0，排除上溢的可能性，同时分母有至少有一个是1。排除了下溢导致分母是0的可能性。

```
import numpy as np
import numpy.linalg as la
x = np.array([1e7, 1e8, 2e5, 2e7])
print(x)
y = np.exp(x) / sum(np.exp(x))
print("上溢: ", y)
x = x - np.max(x)
print(x)
y = np.exp(x)/sum(np.exp(x))
print("上溢处理: ", y)
------------------
[1.e+07 1.e+08 2.e+05 2.e+07]
上溢:  [nan nan nan nan]
[-90000000.         0. -99800000. -80000000.]
上溢处理:  [0. 1. 0. 0.]
-------------------
x = np.array([-1e10, -1e9, -2e10, -1e10])
y = np.exp(x)/sum(np.exp(x))
print("下溢: ", y)
x = x - np.max(x)
print(x)
y = np.exp(x)/sum(np.exp(x))
print("下溢处理: ", y)
print("log softmax(x):", np.log(y))
# 对 log softmax 下溢的处理:
def logsoftmax(x):
    print('x:', x)
    print('sum:', sum(np.exp(x)))
    y = x - np.log(sum(np.exp(x)))
    return y
print("logsoftmax(x):", logsoftmax(x))
```

```
--------------------
下溢： [nan nan nan nan]
[-9.0e+09  0.0e+00 -1.9e+10 -9.0e+09]
下溢处理： [0. 1. 0. 0.]
log softmax(x): [-inf   0. -inf -inf]
x: [-9.0e+09  0.0e+00 -1.9e+10 -9.0e+09]
sum: 1.0
logsoftmax(x): [-9.0e+09  0.0e+00 -1.9e+10 -9.0e+09]
```

4.2 病态条件

条件数表征函数相对于输入的微小变化而变化的快慢程度。输入被轻微扰动而迅速改变的函数对于科学计算来说可能是有问题的，因为输入中的舍入误差可能导致输出误差的巨大变化。
考虑函数$f(x) = A^{-1}x$当$A \in n \times n$具有特征值分解时，其条件数为：

$$\max_{i,j} \left| \frac{\lambda_i}{\lambda_j} \right|$$

这是最大和最小特征值的模之比。当该数很大时，矩阵求逆对输入的误差别特敏感。这种敏感性是矩阵本身的固有特性，而不是矩阵求逆期间舍入误差的结果。

4.3 基于梯度的优化方法

大多数深度学习的算法都涉及某种形式的优化。优化指的是改变x以最小化或最大化某个函数f(x)的任务。通常以最小化f(x)指代大多数优化问题。
我们把要最小化或最大化的函数称为 目标函数(objective function) 或 准则(criterion) 。当对其进行最小化时称它为 代价函数(cost function) 、 损失函数(loss function) 或 误差函数(error function) 。
通常使用上标*表示最小化或最大化函数的x值。如我们记$x^=arg\min f(x)*

。将x往导数的反方向移动一小步来减小$f(x)$，这种技术被称为‘梯度下降$(gradientdescent)$‘。当f(x)'=0$，导数无法提供往哪个方向移动的信息。$f(x)'=0$的点称为‘临界点$(criticalpoint)$‘或‘驻点$(stationarypoint)$‘。一个‘局部极小点$(localminimum)$‘意味着这个点$f(x)$小于所有临近点。$so$不能通过移动无穷小的步长来减小f:\mathbb{R}^n \rightarrow \mathbb{R}

，为了使“最小化”的概念有意义，输出必须是一维(标量)。针对具有多维输入的函数，我们需要用到‘偏导数$(partialderivative)$‘的概念。偏导数\frac{\partial}{\partial x_i} f(x)衡量点x处只有x_i

增加时$f(x)$如何变化。‘梯度$(gradient)$‘是相对一个向量求导的导数：f的导数是包含所有偏导数的向量，记作\nabla_x f(x)。梯度的第i个元素是f关于x_i的偏导数。在多维情况下，临界点是梯度中所有元素都为0的点。在\mu(单位向量)方向的‘方向导数$(directinalderivative)$‘是函数f在\mu方向的斜率。方向导数是函数f(x+\alpha\mu)关于\alpha的导数(在\alpha=0时取得)。使用链式法则，可以看到当\alpha=0时，\frac{\partial}{\partial\alpha}f(x+\alpha\mu)=\mu^T\nabla_xf(x)$。
为了最小化f，我们希望找到使f下降得最快的方向(个人理解：方向导数越小，为负，下降才快。so下面要取方向导数的min)。计算的方向导数：

$$\min_{\mu,\mu^T\mu=1} \mu^T \nabla_x f(x)$$
$$= \min_{\mu,\mu^T\mu=1} \| \mu \|_2 \| \nabla_x f(x) \|_2 \cos\theta$$

其中θ是μ与梯度的夹角。将$\| \mu \|_2 = 1$，代入并忽略与μ无关的项，就能简化得到

$$\min_{\mu} \cos\theta$$

在μ与梯度方向相反时取得最小。也就是说，梯度向量指向上坡，负梯度向量指向下坡。在负梯度方向移动可以减小f。这被称为 最速下降法(method of steepest descent) 或 梯度下降(gradient descent) 。最速下降建议新的点为

$$x' = x - \epsilon \nabla_x f(x)$$

其中ϵ为 学习率(learning rate) ，是一个确定大小的正标量。可以通过几种不同方式选择ϵ。普遍选择一个小常数。有时我们通过计算，选择使方向导数消失的步长。还有种方法是根据几个ϵ计算$f(x - \epsilon\nabla_x f(x))$，选择其中能产生最小目标值的$\epsilon$。这种策略被称为线搜索。最速下降在梯度的每一个元素为零时收敛(或在实践中很接近零)。在某些情况下，我们也许能够避免运行该迭代算法，通过解方程$\nabla_x f(x) = 0$直接跳到临界点。

虽然梯度下降被限制在连续空间中的优化问题，但不断向更好的情况移动一小步的概念可以推广到离散空间。递增带有离散参数的目标函数被称为 爬山(hill climbing) 算法。

```
x0 = np.array([1.0, 1.0, 1.0])
A = np.array([[1.0, -2.0, 1.0], [0.0, 2.0, -8.0], [-4.0, 5.0, 9.0]])
b = np.array([0.0, 8.0, -9.0])
epsilon = 0.001
delta = 1e-3
```

```
"""
gradient descent
"""
def matmul_chain(*args):
    if len(args) == 0: return np.nan
    result = args[0]
    for x in args[1:]:
        result = result @ x
    return result

def gradient_decent(x, A, b, epsilon, delta):
    while la.norm(matmul_chain(A.T, A, x) - matmul_chain(A.T, b)) > delta:
        x -= epsilon * (matmul_chain(A.T, A, x) - matmul_chain(A.T, b))
    return x

gradient_decent(x0, A, b, epsilon, delta)

array([27.82277014, 15.34731055,  2.83848939])
```

4.3.1 梯度之上：Jacobian和Hession矩阵

有时需要计算输入和输出都为向量的函数的所有偏导数。包含所有这样偏导数的矩阵称为Jacobian矩阵。具体来说，如果我们有一个函数$f:\mathbb{R}^m-\mathbb{R}^n$，f的Jacobin矩阵$J\in\mathbb{R}^{n\times m}$定义为$J_{i,j}=\frac{\partial}{\partial x_j}f(x)_i$。

补充知识：
Jacobian相对于通用型函数的一阶导数，Hession矩阵是一个$\mathbb{R}^n\to\mathbb{R}$的函数的二阶导数。本质上说，一个函数对(行)向量求导，本质上还是为向量每个元素进行求导。比如$\mathbb{R}^n\to\mathbb{R}$的函数$f(\vec{a})$，则其导数(梯度)为$\nabla f=[\frac{\partial f}{\partial a_1},\frac{\partial f}{\partial a_2},\dots,\frac{\partial f}{\partial a_n}]$，此时一阶导数就变成一个$\mathbb{R}^n\to\mathbb{R}^n$的函数，对于$\mathbb{R}^m\to\mathbb{R}^n$的函数，可以看成一个长度为n列向量，对一个长度为m的行向量求偏导。

$$\frac{\partial\vec{y}}{\partial\vec{a}}=\begin{bmatrix}\frac{\partial y_1}{\partial a_1},\dots,\frac{\partial y_1}{\partial a_m}\\ \dots,\dots,\dots\\ \frac{\partial y_n}{\partial a_1},\dots,\frac{\partial y_n}{\partial a_m}\end{bmatrix}$$

有时也对导数的导数感兴趣，即 二阶导数(second derivative) 。例如，有一个函数$f:\mathbb{R}^m\to\mathbb{R}$，f的一阶导数(关于$x_j$)关于$x_i$的导数记为$\frac{\partial^2}{\partial x_i\partial x_j}f$。在一维情况下，可以将$\frac{\partial^2}{\partial x^2}f$为$f''(x)$。二阶导数告诉我们，一阶导数将如何随着输入变化而改变。它表示只基于梯度信息的梯度下降步骤是否会如我们预期那样大的改善，二阶导数是曲率的衡量。假设有一个二次函数，如果这样的函数具有零二阶导数，那么就没有曲率。也就是一条完全平坦的线，仅用梯度就可以预测他的值。我们使用沿负梯度方向大小为ϵ的下降步，当梯度是1时，代价函数将下降ϵ。如果二阶导数是负，函数曲线向上凸出，因此代价函数将下降比ϵ多。如果二阶导数是正，函数曲线向下凹，因此代价函数将下降比ϵ少。
当函数有多维输入时，二阶导数也有很多。我们可以将这些导数合并成一个矩阵，称为 Hession矩阵 ，Hession矩阵$H(f)(x)$定义为

$$H(f)(x)_{i,j}=\frac{\partial^2}{\partial x_i\partial x_j}f(x)$$

Hessian等价于梯度的Jacobian矩阵。
微分算子在任何二阶偏导连续连续的点处可交换，也就是顺序可以互换：

$$\frac{\partial^2}{\partial x_i \partial x_j} f(x) = \frac{\partial^2}{\partial x_j \partial x_i} f(x)$$

这意味着$H_i, j = H_j, i$，因此Hession在这些点上是对称的。因为Hession是是对称矩阵，我们可以将其分解成一组实特征值和一组特征向量的正交基，在特定方向d上的二阶导数可写成$d^T Hd$。当d是H的一个特征向量时，这个方向的二阶导数就是对应的特征值。对于其他的方向d，方向二阶导数是所有特征值的加权平均，权重在0和1之间，且与d夹角越小的特征向量权重越大。最大特征值确定最大二阶导数，最小特征值确定最小二阶导数。
我们可以通过(方向)二阶导数预期一个梯度下降步骤能表现有多好。我们在当前点$x^{(0)}$处作函数f(x)近似二阶泰勒级数：

$$f(x) \approx f(x^{(0)}) + (x - x^{(0)})^T g + \frac{1}{2}(x - x^{(0)})^T H(x - x^{(0)})$$

其中g是梯度，H是$x^{(0)}$点的Hession。如果使用学习率ϵ，那么新的点x将会是$x^{(0)} - \epsilon g$代入上述的近似，可得：

$$f(x^{(0)} - \epsilon g) \approx f(x^{(0)}) - \epsilon g^T g + \frac{1}{2}\epsilon^2 g^T Hg$$

其中有3项：函数原始值、函数斜率导致的预期改善、函数曲率导致的校正。当最后一项太大时，梯度下降实际上是可能向上移动的。当$g^T Hg$为零或负时，近似的泰勒级数表明增加ϵ将永远使f下降。在实践中，泰勒级数不会在ϵ大的时候也保持准确，so在这种情况下必须采取更启发式的选择。当$g^T Hg$为正时，通过计算可得，使近似泰勒级数下降最多的最优步长为：

$$\epsilon^* = \frac{g^T g}{g^T Hg}$$

知识补充：
写一下推到过程，泰勒级数展开式对ϵ求导，令导数为0。

$$\frac{df(x^{(0)} - \epsilon g)}{d\epsilon} \approx -g^T g + \epsilon g^T Hg = 0$$

可以得到上面的ϵ^*。表明最速下降法最优步长，不仅与梯度有关，而且与Hession矩阵有关。
最坏情况下，g与H最大特征值λ_{max}对应的特征向量对齐(此处翻译太糟糕，英文版是align with表示对齐，成一条直线的意思。此处最好理解的应该翻译成两个特征向量成一条直线)，最优步长是$\frac{1}{\lambda_{max}}$。我们要最小化的函数能够用二次函数很好的近似的情况下，Hession的特征值决定了学习率的量级。
二阶导数还可以被用来确定一个临界点是否是局部极大点、局部极小点或鞍点。当$f'(x) = 0$。而$f''(x) > 0$时，x是一个局部极小点。同样，当$f'(x) = 0$。而$f''(x) < 0$时，x是一个局部极大点。这就是 二阶导数测试(second derivative test) 。不幸的是，当$f''(x) = 0$时测试是不确定的。这种情况下，x可以说鞍点或平坦区域的一部分。

```
"""
newton
"""
def matmul_chain(*args):
    if len(args) == 0: return np.nan
    result = args[0]
    for x in args[1:]:
        result = result @ x
    return result

def newton(x, A, b):
    x = matmul_chain(np.linalg.inv(matmul_chain(A.T, A)), A.T, b)
    return x
newton(x0, A, b)

array([29., 16.,  3.])
```

在多维情况下，需要检测函数的所有二阶导数。利用Hession特征值分解，我们可以将二阶导数测试扩展到多维情况。在临界点处($\nabla_x f(x) = 0$)，我们通过检测Hession的特征值该临界点是局部极大点、局部极小点还是鞍点。当Hession是正定的(所有的特征值都是正的)，则临界点是局部极小点。多维情况下，可以找到该点是否为鞍点的迹象。如果Hession特征值中有正有负，那么x是f某个横截面的局部极大点，却是另一个横截面的局部最小点。
多维情况下，单个点处每个方向上二阶导数是不同的。Hession的条件数衡量这些二阶导数的变化范围。
对condition number来个一句话总结：condition number 是一个矩阵（或者它所描述的线性系统）的稳定性或者敏感度的度量，如

果一个矩阵的 condition number 在1附近，那么它就是well-conditioned的，如果远大于1，那么它就是 ill-conditioned 的，如果一个系统是 ill-conditioned 的，它的输出结果就不要太相信了。
当条件数很差时，梯度下降法也会表现的很差。因为一个方向上的导数增加得很快，而另一个方向上增加得很慢。梯度下降不知道导数这种变化，so不知道优先处理导数长期为负的方向。(补充：求解方程组时如果对数据进行较小的扰动，则得出的结果具有很大波动，这样的矩阵称为病态矩阵。)病态条件也导致很难选择合适的步长。步长必须足够小，以免冲过最小点，往正曲率方向上升(就是往上破走)。但步长小那么在小曲率上进展很慢。
可以使用Hession矩阵指导搜索来解决这个问题。最简单的方法是 牛顿法(Newton's method) 。牛顿法基于一个二阶泰勒展开来近似$x^{(0)}$附近的$f(x)$：

$$f(x) \approx f(x^0) + (x - x^{(0)})^T \nabla_x f(x^0) + \frac{1}{2}(x - x^{(0)})^T H(f)(x^{(0)})(x - x^{(0)})$$

(知识补充：对上式进行求导，令$f^{'}(x) = 0$得到

$$f^{'}(x^{(0)}) + f^{''}(x^{(0)})(x - x^{(0)}) = 0$$
$$x = x^{(0)} - \frac{f^{'}(x^{(0)})}{f^{''}(x^{(0)})}$$

对于神经网络病态条件问题，出现在梯度变化过快的情况时即二阶导数较大，此时通过二阶优化算法如牛顿法，将二阶导数作为分母加入更新参数，使得在梯度变化过快方向相对梯度变化慢的方向更新尺度发生改变，从而解决病态问题。)
通过计算可以得到函数的临界点：

$$x^* = x^{(0)} - H(f)(x^{(0)})^{-1} \nabla_x f(x^{(0)})$$

当f是一个正定二次函数(知识补充：正定二次函数(positive definite quadratic function)是系数矩阵为对称正定矩阵的二次函数。设$x \in R^n$，A为$n \times n$对称正定矩阵，$b \in R^n$为常向量，c为常数，则二次函数$f(x) = \frac{1}{2}x^T Ax + b^T x + c$称为正定二次函数。矩阵有正定、负定和不定之分。对于任意非零向量X：
(1) 若有$X^T HX > 0$，则称矩阵且H是正定矩阵。
(2)若有$X^H X < 0$，则称矩阵H是负定矩阵。
(3)若有时$X^T HX > 0$，有时$X^T HX < 0$，则称矩阵H是不定矩阵。)牛顿法只要应用一次上式就能直接跳到函数的最小点。如果f不是一个真正二次，但能在局部近似为正定二次，牛顿法需要多次迭代上式。迭代更新近似函数和跳到近似函数的最小点可以比梯度下降更快到达临界点。
仅使用梯度信息的优化算法被称为 一阶优化算法(first-order optimization algorithms) ，如梯度下降。使用Hession矩阵的优化算法被称为 二阶最优化算法(second-order optimizition algorithns)，如牛顿法。
本书大多数上下文中使用的优化算法适用于各种各样的函数，但几乎都没有保证。因为深度学习中使用的函数族是相当复杂的，so深度学习算法往往缺乏保证。在许多其他领域，优化的主要方法是为有限的函数族设计优化算法。
在深度学习背景下，限制函数满足 Lipschitz连续(Lipschitz continues) 或其导数Lipschitz连续可以获得一些保证。Lipschitz连续函数的变化速度以 Lipschitz常数(Lipschitz constant)`ζ为界：

$$\forall x, \forall y, |f(x) - f(y)| \leq \zeta \|x - y\|_2$$

这个属性允许我们量化我们的假设--梯度下降等算法导致的输入的微小变化将使输出只产生微小变化，Lipschitz连续性也是脆弱的约束，并且深度学习中很多优化问题经过相对较小的修改后就能变成Lipschitz连续。
最成功的特定优化领域或许是 凸优化(Convex optimization) ，凸优化通过更强的限制提供更多的保证。凸优化算法只对凸函数适用，即Hession处处半正定的函数。因为这些函数没有鞍点而且其所有局部极小点必然是全局最小点，so表现很好。但是深度学习中大部分问题都难以表示成凸优化形式。凸优化仅用作一些深度学习算法的子程序。凸优化中分析思路对对证明深度学习算法收敛性非常有用，然而一般来说，深度学习背景下凸优化的重要性大大减少。

4.4 约束优化

有时候，在x的所有可能值下最大化或最小化一个函数$f(x)$不是我们所希望的。相反我们可能希望在x的某些集合$\mathbb{S}$中找$f(x)$的最大值或最小值。这被称为 约束优化(cinstrained optimization) 。在约束优化术语中，集合$\mathbb{S}$内的点x被称为 可行(feasible) 点。我们常希望找到在某种意义上小的解。常见方法是强加一个范数约束，如$\|x\| \leq 1$。
约束优化的一个简单方法是将约束考虑在内后简单的对梯度下降进行修改。如果我们使用一个小的恒定步长ϵ，我们可以先取梯度下降单步结果，然后将结果投影回$\mathbb{S}$。如果我们使用线搜索，我们只能在步长为ϵ范围内搜索可行的新x点，或者我们可以将线上的每个点投影到约束区域。如果可能的话，在梯度下降或线搜索前将梯度投影到可行域的切空间会更高效(不得不说这里翻译烂透了，看了英文When possible, this method can be made more efficient by projecting the gradient into the tangent space of feasible region before taking the step or beginning the line search.其中tangent意思是切线的。所以这里翻译成切空间。)一个更复杂的方法是设计一个不同的、无约束的优化问题，其解可以转化成原始优化问题的解。例如，我们要在$x \in \mathbb{R}^2$中最小化$f(x)$，其中

x约束为具有单位L^2范数。我们可以关于θ最小化$g(\theta) = f([\cos\theta, \sin\theta]^T)$，最后返回$[\cos\theta, \sin\theta]$作为愿问题的解。这种方法需要创造性；优化问题之间的转换必须专门根据我们遇到的每一种情况进行设计。

`Karush-Kuhn-Tucker(KKT)方法` (KKT方法是Lagrangian乘子法(只允许等式约束)的推广)是针对约束优化非常通用的解决方案。为介绍KKT方法，引入一个称为 `广义Lagrangian( generalized Lagrangian)` 或 `广义Lagrangian函数(generalized Lagrangian function)` 的新函数。

为了定义Lagrangian，我们先要通过等式或不等式得形式描述$\mathbb{S}$，希望通过m个函数$g^{(i)}$和n个函数$h^{(j)}$描述$\mathbb{S}$，那么$\mathbb{S}$可以表示为$\mathbb{S} = \{x|\forall i, g^{(i)}(x) = 0 and \forall j, h^{(j)}(x) \leq 0\}$其中$g^{(i)}$的等式称为 `等式约束(equality constraint)` ，涉及$h^j$的不等式称为 `不等式约束(inequality constraint)` 。

我们为每个约束引入新的变量λ_i和α_j，这些新变量被称为KKT乘子。广义Lagrangian(就是拉格朗日)可以定义如下：

$$L(x, \lambda, \alpha) = f(x) + \sum_i \lambda_i g^{(i)} + \sum_j \alpha_j h^{(j)}$$

现在我们可以通过优化无约束的广义Lagrangian解决约束最小化问题。只要存在至少一个可行点且f(x)不允许取∞，那么

$$\min_x \max_\lambda \max_{\alpha, \alpha \geq 0} L(x, \lambda, \alpha)$$

与如下函数有相同的最优目标函数值和最优点集x

$$\min_{x \in \mathbb{S}} f(x)$$

这是因为当约束满足时

$$\max_\lambda \max_{\alpha, \alpha \geq 0} L(x, \lambda, \alpha) = f(x)$$

而违反任意约束时

$$\max_\lambda \max_{\alpha, \alpha \geq 0} L(x, \lambda, \alpha) = \infty$$

这些性质保证不可行点不是最佳的，并且可行点范围内的最优点不变。

要解决约束最大化问题，我们可以构造$-f(x)$的广义Lagrangian函数，从而导致以下优化问题：

$$\min_x \max_\lambda \max_{\alpha, \alpha \geq 0} -f(x) + \sum_i \lambda_i g^{(i)} + \sum_j \alpha_j h^{(j)}$$

也可以转化成外层最大化问题：

$$\max_x \min_\lambda \min_{\alpha, \alpha \geq 0} f(x) + \sum_i \lambda_i g^{(i)} - \sum_j \alpha_j h^{(j)}$$

等式约束对应项的符号不重要，可以自由选择λ_i的正负号，可以将其定义为加法或减法。

不等式约束特别有趣。如果$h^{(i)}(x^*) = 0$，我们就说这个约束$h^{(i)}(x)$是 `活跃(active)` 的。如果约束不是活跃的，则有该约束问题的解与去掉该约束问题的解至少存在一个相同的局部解。(个人理解：去掉该约束相对于求解范围变大，有该约束说明解范围变小，相对于取了没约束的子集，如在有约束求得极值点，那么在没用约束的大范围中，也是极值点，也就是导数为0的点，看了网上拉格朗日乘子法与KKT条件，此种情况，约束的最优点不在边界，在内部，而且与无约束函数是同解)一个不活跃的约束有可能排除其他解。例如，整个区域(代价相等的宽平区域)都是全局最优点的凸问题，可能因为约束消去其中的某个子区域，或在非凸问题的情况下，收敛时不活跃的约束可能排除了较好的局部驻点。然而，无论不活跃的约束是否包含在内(不活跃说明极值点在内部，与$f(x)$同解)，收敛时找到的点仍然是一个驻点。因为一个不活跃的约束$h^{(i)}$必有负值，那么

$$\min_x \max_\lambda \max_{\alpha, \alpha \geq 0} L(x, \lambda, \alpha)$$

中的$\alpha_i = 0$。(可行解落在约束区域内部，此时约束不起作用，令$\alpha_i = 0$消去约束即可)。因此，我们可以观察到在该解中$\alpha \odot h(x) = 0$。换句话说，对于所有的i，$\alpha_i \geq 0$或$h^{(j)}(x) \leq 0$在收敛时必有一个是活跃的(要么在不等式边界上$h^{(i)} = 0$要么$\alpha_i = 0$)。这个想法的直观解释，可以说这个解是不等式强加的边界(不等式等于0的情况，解在边界)，必须通过对应的KKT乘子影响x的解，或者不等式对解没有影响(不加约束解和加不等式约束解相同时，不等式可以消去，失效了)，我们则归零KKT乘子。

我们可以使用一组简单的性质来描述约束优化问题的最优点。这些性质被称为 `Karush-Kuhn-Tucker(KKT)条件` 。这些是确定一个点是最优点的必要条件，但不一定是充分条件。这些条件是：

- 广义Lagrangian梯度为0
- 所有关于x和KKT约束都满足
- 不等式约束显示的“互补松弛性”$\alpha \odot h(x) = 0$。

(知识补充：
列出Lagrangian得到无约束优化问题：

$$L(x,\alpha,\beta)=f(x)+\sum_{i=1}^{m}\alpha_i h_i(x)+\sum_{j=1}^{n}\beta_i g_i(x)$$

不等式约束后可行解x需要满足的就是以下KKT条件：

$$\begin{aligned}\nabla_x L(x,\alpha,\beta) =&0 &(1)\\ \beta_j g_j(x) =&0, j=1,2,\dots,n &(2)\\ h_i(x) =&0, i=1,2,\dots,n &(3)\\ g_j(x) \le&0, j=1,2,\dots,n &(4)\\ \beta_j \ge&0, j=1,2,\dots,n &(5)\end{aligned}$$

满足KKT条件后极小化Lagrangian即可得到在不等式约束条件下的可行解。KKT条件看起来很多但很好理解：
(1)：拉格朗日取得可行解的必要条件；
(2)：这就是以上分析比较有意思的约束，称作松弛互补条件；
(3) (4)：初始的约束条件；
(5)：不等式约束的Lagrangian乘子需满足的条件。

主要的KKT条件是(3)和(5)，只要满足这俩个条件便可直接用拉格朗日乘子法， SVM 中的支持向量便是来自于此，需要注意的是 KKT 条件与对偶问题也有很大的联系，后续章节研究拉格朗日对偶。)

(知识补充：

梯度的提出只为回答一个问题：函数在变量空间的某一点处，沿着哪一个方向有最大的变化率?

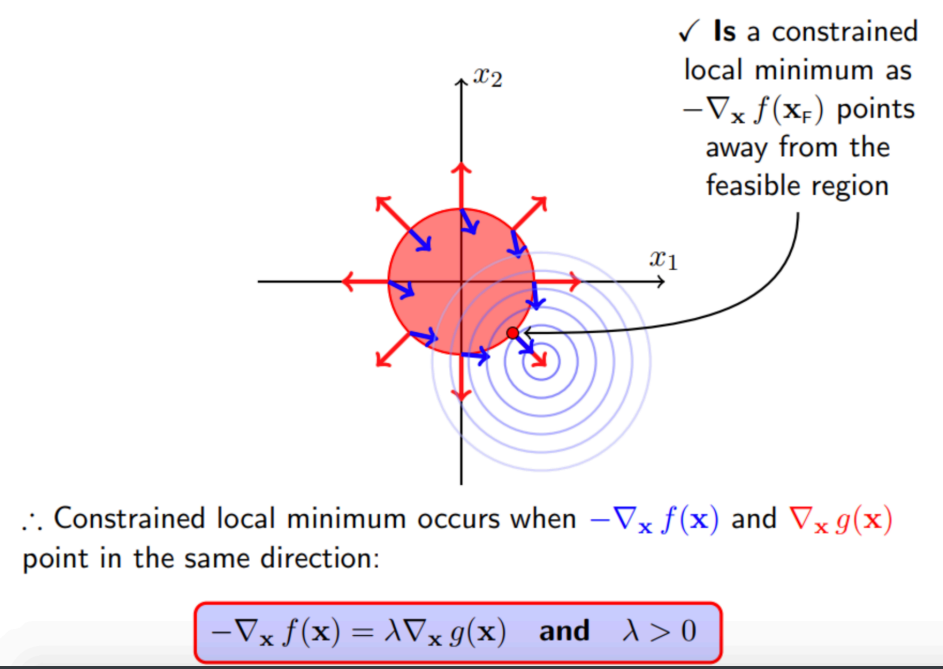
梯度定义如下：

函数在某一点的特殊梯度是这样一个向量，它的方向与取得最大方向导数的方向一致，而它的模为方向导数的最大值。

注意三点：

- 梯度是一个向量，即有方向有大小；
- 梯度的方向是最大方向导数的方向
- 梯度的值是最大方向导数的值

)



对于f(x)而言要沿着f(x)的负梯度方向走，才能走到极小点，如上图蓝色箭头。由于$g(x) \leq 0$，只有在边缘处等于0，因此区域内都是小于0的，类似一个碗，边缘高，中间低。梯度方向和最大导数方向一致，梯度方向向四周发散。只有当梯度上的点，最靠近f(x)极值点的方向。因此g(x)的梯度和f(x)的负梯度同向。因此极小点在边界上，这个时候g(x)=0。$-\nabla_x f(x) = \lambda \nabla_x g(x)$，这个式子要满足，要求拉格朗日乘子$\lambda > 0$。

```
"""
constrain optimazation
"""
def matmul_chain(*args):
    if len(args) == 0: return np.nan
    result = args[0]
    for x in args[1:]:
        result = result @ x
    return result

def contrain_opti(x, A, b, delta):
    k = len(x)
    lamb = 0
    while np.abs(np.dot(x.T, x) - 1) > 5e-2:
        x = matmul_chain(np.linalg.inv(matmul_chain(A.T, A) + 2 * lamb * np.identity(k)), A.T, b)
        lamb += np.dot(x.T, x) - 1
    return x

contrain_opti(x0, A, b, delta)

array([ 0.23637902,  0.05135858, -0.94463626])
```

```
import numpy
print("numpy:", numpy.__version__)

numpy: 1.19.2
```

对于f(x)而言要沿着f(x)的负梯度方向走，才能走到极小点，如上图蓝色箭头。由于$g(x) \leq 0$，只有在边缘处等于0，因此区域内都是小于0的，类似一个碗，边缘高，中间低。梯度方向和最大导数方向一致，梯度方向向四周发散。只有当梯度上的点，最靠近f(x)极值点的方向。因此g(x)的梯度和f(x)的负梯度同向。因此极小点在边界上，这个时候g(x)=0。$-\nabla_x f(x) = \lambda \nabla_x g(x)$，这个式子要满足，要求拉格朗日乘子$\lambda > 0$。

4.5 实例：线形最小二乘

假如我们希望找到最小化下式x值

$$f(x) = \frac{1}{2}\|Ax - b\|_2^2$$

存在专门的线性代数算法能够高效地解决这个问题；但是我们也可以探索如何使用基于梯度的优化来解决这个问题，这可以作为这些技术是如何工作的一个简单例子。首先计算梯度：

$$\nabla_x f(x) = A^T(Ax - b) = A^T Ax - A^T b$$

然后可以采用小的步长，并按照这个梯度下降。见算法详解。

算法从任意点x开始，使用梯度下降关于x最小化$\nabla_x f(x) = A^T(Ax - b) = A^T Ax - A^T b$的算法。

将步长(ϵ)和容差(δ)设为小的正数。
while $\|A^T Ax - A^b\|_2 > \delta$ do
$x \leftarrow x - \epsilon(A^T Ax - A^T b)$

end while

我们也可以用牛顿法解决这个问题。因为在这个情况下，真实函数是二次的，牛顿法所用的二次近似是精确的，该算法会在一步后收敛到全局最小点。
现在希望最小化同样的函数，但受到$x^T x \leq 1$的约束，要做到这一点我们引入Lagrangrian

$$L(x, \lambda) = f(x) + \lambda(x^T x - 1)$$

现在解决以下问题

$$\min_x \max_{\lambda, \lambda \geq 0} L(x, \lambda)$$

我们可以用Moore-Penrose伪逆$x = A^+ b$找到无约束最小二乘问题的最小范数解。(知识补充：如果矩阵 A的行数大于列数，那么方程$Ax = y$，可能没有解。如果矩阵 A的列数大于行数，那么上述方程可能有多个解。Moore-Penrose伪逆使我们在这类问题上取得了一定的进展。矩阵 A的伪逆定义为

$$A^+ = \lim_{a \to 0}(A^T A + \alpha I)^{-1} A^T$$

计算伪逆的算法没有基于这个定义，而是使用下面的公式，

$$A^+ = V D^+ U^T$$

其中，矩阵U，D和 V是矩阵 A经奇异值分解后得到的矩阵。对角矩阵D的伪逆D^+是其非零元素取倒数再经转置后得到的。当矩阵A的列数多于行数时，使用伪逆求解线性方程是众多可能解法中的一种。特别的，$x = A + y$是方程所有可行解中欧几里德范数$\|x\|_2$最小的一个。 当矩阵 A的行数多于列数时，可能没有解。在这种情况下，通过伪逆求得的 x使得 Ax和 y的欧几里德距离$\|Ax - \|_2$最小。
)
如果这一点可行，这也是约束问题的解。否则我们必须找到约束是活跃的解。关于x对Lagrangian微分，我们得到方程

$$A^T Ax - A^T b + 2\lambda x = 0$$

这告诉我们该解的形式将会是

$$x = (A^T A + 2\lambda I)^{-1} A^T b$$

λ的选择必须使结果服从约束，我们可以关于λ进行梯度上升找到这个值，为了做到这一点，观察

$$\frac{\partial}{\partial \lambda} L(x, \lambda) = x^T x - 1$$

当x的范数超过1时，该导数是正的，so为了跟随导数上坡并相对λ增加Lagrangian，我们需要增加λ。因为$x^T x$的惩罚系数增加了，求解关于x的线性方程现在将得到具有最小范数的解。求解线性方程和调整λ的过程将一直持续到x具有正确的范数并且关于λ的导数是0。