数值计算

朱明超

Email: deityrayleigh@gmail.com

Github: github.com/MingchaoZhu/DeepLearning

1 上溢和下溢

下溢（Underflow）：当接近零的数被四舍五入为零时发生下溢。

上溢（Overflow）：当大量级的数被近似为 ∞ 或 $-\infty$ 时发生上溢。

必须对上溢和下溢进行数值稳定的一个例子是 softmax 函数。softmax 函数经常用于预测与范畴分布相关联的概率，定义为:

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

[1]:
```
import numpy as np
import numpy.linalg as la
```

[2]:
```
x = np.array([1e7, 1e8, 2e5, 2e7])
y = np.exp(x)/sum(np.exp(x))
print("上溢: ", y)
x = x - np.max(x)  # 减去最大值
y = np.exp(x)/sum(np.exp(x))
print("上溢处理: ", y)
```

```
上溢:  [nan nan nan nan]
上溢处理:  [0. 1. 0. 0.]
```

[3]:
```
x = np.array([-1e10, -1e9, -2e10, -1e10])
y = np.exp(x)/sum(np.exp(x))
print("下溢: ", y)
x = x - np.max(x)  # 减去最大值
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):
    y = x - np.log(sum(np.exp(x)))
    return y

print("logsoftmax(x):", logsoftmax(x))
```

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

2 优化方法

2.1 梯度下降法

梯度下降法 (Gradient Descent) 或最速下降法 (Method of Steepest Descent) 的目标函数是最小化具有多维输入的函数：$f : \mathbb{R}^n \to \mathbb{R}$。梯度下降法建议新的点为:

$$\boldsymbol{x}' = \boldsymbol{x} - \epsilon \nabla_{\boldsymbol{x}} f(\boldsymbol{x}) \tag{2}$$

其中 ϵ 为学习率 (learning rate)，是一个确定步长大小的正标量。

这里引入实例 (线性最小二乘)：

$$f(\boldsymbol{x}) = \frac{1}{2}\|\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}\|_2^2 \tag{3}$$

假设我们希望找到最小化该式的 x 值。

可以计算梯度得到：

$$\nabla_{\boldsymbol{x}} f(\boldsymbol{x}) = \boldsymbol{A}^\top(\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}) = \boldsymbol{A}^\top\boldsymbol{A}\boldsymbol{x} - \boldsymbol{A}^\top\boldsymbol{b} \tag{4}$$

实例：线性最小二乘

[4]:
```
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
# 给定 A, b, 真正的解 x 为 [29, 16, 3]
```

[5]:
```
"""
梯度下降法
"""
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)
```

[5]:
```
array([27.82277014, 15.34731055,  2.83848939])
```

2.2 牛顿法

牛顿法 (Newton's Method) 基于一个二阶泰勒展开来近似 $\boldsymbol{x}^{(0)}$ 附近的 $f(\boldsymbol{x})$：

$$f(\boldsymbol{x}) \approx f(\boldsymbol{x}^{(0)}) + (\boldsymbol{x} - \boldsymbol{x}^{(0)})^\top\nabla_{\boldsymbol{x}} f(\boldsymbol{x}^{(0)}) + \frac{1}{2}(\boldsymbol{x} - \boldsymbol{x}^{(0)})^\top \mathrm{H}(\boldsymbol{x}^{(0)})(\boldsymbol{x} - \boldsymbol{x}^{(0)}) \tag{5}$$

接着通过计算，我们可以得到这个函数的临界点：

$$\boldsymbol{x}^* = \boldsymbol{x}^{(0)} - \mathrm{H}(\boldsymbol{x}^{(0)})^{-1}\nabla_{\boldsymbol{x}} f(\boldsymbol{x}^{(0)}) \tag{6}$$

牛顿法迭代地更新近似函数和跳到近似函数的最小点可以比梯度下降法更快地到达临界点。这在接近全局极小时是一个特别有用的性质，但是在鞍点附近是有害的。

针对上述实例，计算得到：$\mathrm{H} = \boldsymbol{A}^\top\boldsymbol{A}$

进一步计算得到最优解：

$$\boldsymbol{x}^* = \boldsymbol{x}^{(0)} - (\boldsymbol{A}^\top\boldsymbol{A})^{-1}(\boldsymbol{A}^\top\boldsymbol{A}\boldsymbol{x}^{(0)} - \boldsymbol{A}^\top\boldsymbol{b}) = (\boldsymbol{A}^\top\boldsymbol{A})^{-1}\boldsymbol{A}^\top\boldsymbol{b} \tag{7}$$

[6]:
```
"""
牛顿法
"""
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, delta):
```

```
    x = matmul_chain(np.linalg.inv(matmul_chain(A.T, A)), A.T, b)
    return x

newton(x0, A, b, delta)
```

[6]: array([29., 16., 3.])

2.3 约束优化

我们希望通过 m 个函数 $g^{(i)}$ 和 n 个函数 $h^{(j)}$ 描述 S ，那么 S 可以表示为 $\mathrm{S} = \{\boldsymbol{x} \mid \forall i, g^{(i)}(\boldsymbol{x}) = 0 \text{ and } \forall j, h^{(j)}(\boldsymbol{x}) \leq 0\}$。其中涉及 $g^{(i)}$ 的等式称为等式约束，涉及 $h^{(j)}$ 的不等式称为不等式约束。

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。广义拉格朗日式可以如下定义：

$$L(\boldsymbol{x}, \lambda, \alpha) = f(\boldsymbol{x}) + \sum_i \lambda_i g^{(i)}(\boldsymbol{x}) + \sum_j \alpha_j h^{(j)}(\boldsymbol{x}) \tag{8}$$

可以通过优化无约束的广义拉格朗日式解决约束最小化问题：

$$\min_{\boldsymbol{x}} \max_{\lambda} \max_{\alpha, \alpha \geq 0} L(\boldsymbol{x}, \lambda, \alpha) \tag{9}$$

优化该式与下式等价：

$$\min_{\boldsymbol{x} \in \mathrm{S}} f(\boldsymbol{x}) \tag{10}$$

针对上述实例，约束优化: $\boldsymbol{x}^\top \boldsymbol{x} \leq 1$

引入广义拉格朗日式：

$$L(\boldsymbol{x}, \lambda) = f(\boldsymbol{x}) + \lambda(\boldsymbol{x}^\top \boldsymbol{x} - 1) \tag{11}$$

解决以下问题：

$$\min_{\boldsymbol{x}} \max_{\lambda, \lambda \geq 0} L(\boldsymbol{x}, \lambda) \tag{12}$$

关于 $\boldsymbol{x}$ 对 Lagrangian 微分，我们得到方程:

$$\boldsymbol{A}^\top \boldsymbol{A}\boldsymbol{x} - \boldsymbol{A}^\top \boldsymbol{b} + 2\lambda \boldsymbol{x} = \boldsymbol{0} \tag{13}$$

得到解的形式是：

$$\boldsymbol{x} = (\boldsymbol{A}^\top \boldsymbol{A} + 2\lambda \boldsymbol{I})^{-1} \boldsymbol{A}^\top \boldsymbol{b} \tag{14}$$

λ 的选择必须使结果服从约束，可以对 λ 梯度上升找到这个值：

$$\frac{\partial}{\partial \lambda} L(\boldsymbol{x}, \lambda) = \boldsymbol{x}^\top \boldsymbol{x} - 1 \tag{15}$$

[7]:
```
"""
约束优化，约束解的大小
"""
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 constrain_opti(x, A, b, delta):
    k = len(x)
    lamb = 0
    while np.abs(np.dot(x.T, x)-1) > 5e-2:  # delta 设为 5e-2, 最优设为 0
        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

constrain_opti(x0, A, b, delta)
```

[7]: array([0.23637902, 0.05135858, -0.94463626])

[8]:
```
import numpy
print("numpy:", numpy.__version__)
```

```
numpy: 1.14.5
```