梯度下降法及其优化算法分析报告

如今，在神经网络魔性的训练过程中梯度下降被广泛的使用，它主要用于权重的更新，即对参数向某一方向进行更新和调整，来最小化损失函数。其主要原理是：通过寻找最小值，控制方差，更新模型参数，最终使模型收敛[1]。梯度下降法以负梯度方向作为极小化算法的下降方向，是无约束优化中最简单的方法。但随着问题规模、维度的加大，收敛速度逐渐减慢。本文将围绕梯度下降法的改进，概述从梯度下降法的变形形式到其优化算法的原理、实现、特点，最后加以总结。

1 梯度下降法变形形式

根据计算梯度时所用数据量不同，可以分为三种基本方法：批量梯度下降法（Batch Gradient Descent, BGD）、随机梯度下降法（Stochastic Gradient Descent, SGD）以及小批量梯度下降法（Mini-batch Gradient Descent, MBGD）。

这里首先给出梯度下降法的一般求解框架：

- 给定待优化连续可微函数 J(Θ)、学习率α以及一组初始值Θ_0
- 计算待优化函数梯度 $\nabla J(\Theta_0)$
- 更新迭代公式$\Theta_{i+1} = \Theta_i - \alpha\nabla J(\Theta_0)$
- 计算Θ_{i+1}处的函数梯度$\nabla J(\Theta_{i+1})$
- 计算梯度向量的模来判断算法是否收敛
- 若收敛，算法停止，否则根据迭代公式继续迭代

1.1 批量梯度下降法（BGD）

1.1.1 原理

BGD 相对于标准 GD 进行了改进，也就是不再是想标准 GD 一样，对每个样本输入都进行参数更新，而是针对一个批量的数据输入进行参数更新。一般情况下，BGD 是在一个批量的样本数据中，求取该批量样本梯度的均值来更新参数，即每次权值调整发生在批量样本输入之后，而不是每输入一个样本就更新一次模型参数。

1.1.2 实现

```
#基于多元回归的应用 y = x0 + x1, X = [x0 , x1]
def BGD(X,y):
    ept = 0.001  #精度
    loss = 1     #损失
    alpha = 0.01 #学习率
```

```
    max_iter = 0 #迭代次数
    theta = np.random.randint(1,10,(X.shape[1],1)) #初始化
    while max_iter <= 10000 or loss < ept:
        partial = (1/X.shape[0])*X.T.dot(X.dot(theta)-y) #损失函数关于 theta 的偏导数
        theta = theta - alpha*partial
        max_iter+=1
        loss=(1/(2*X.shape[0]))*np.sum((X.dot(theta)-y)**2)
    return max_iter,theta
```

1.1.3 特点

- 每次迭代的梯度方向计算由所有训练样本共同决定，所以它的损失度比较稳定，不会产生大的震荡。对于凸函数，批量梯度下降法能够保证收敛到全局最小值，对于非凸函数，则收敛到一个局部最小值。
- 在执行每次更新时，我们需要在整个数据集上计算所有的梯度，所以批梯度下降法的速度会很慢，同时，批梯度下降法无法处理超出内存容量限制的数据集。批梯度下降法同样也不能在线更新模型，即在运行的过程中，不能增加新的样本。

1.2 随机梯度下降法（SGD）

1.2.1 原理

随机梯度下降法，不像 BGD 每一次参数更新，需要计算整个数据样本集的梯度，而是每次参数更新时，仅仅选取一个样本计算其梯度,以达到降低计算复杂度的目的[2]。可以看到 BGD 和 SGD 是两个极端。

1.2.2 实现

```
#基于多元回归的应用 y = x0 + x1, X = [x0 , x1]
def SGD(X,y):
    ept = 0.001  #精度
    loss = 1     #损失
    alpha = 0.01 #学习率
    max_iter = 0 #迭代次数
    theta = np.random.randint(1,10,(X.shape[1],1)) #初始化
    numsSample = X.shape[0]
    while max_iter <= 10000 or loss < ept:
        i = np.random.randint(0, numsSample) #随机抽取一个样本
        partial=X[i:i+1,:].T.dot((X[i:i+1,:].dot(theta)-y[i,:]).reshape(1,1)) #损失函数关于 theta 的偏导数
        theta=theta-alpha*partial
```

```
            max_iter+=1
            loss=(1/(2*X.shape[0]))*np.sum((X.dot(theta)-y)**2)
        return max_iter,theta
```

1.2.3 特点

- SGD 由于每次参数更新仅仅需要计算一个样本的梯度，训练速度很快，即使在样本量很大的情况下，可能只需要其中一部分样本就能迭代到最优解。
- 由于每次迭代并不是都向着整体最优化方向，甚至有可能向着反方向前进，因此更新的路线不稳定，导致梯度下降的波动非常大，更容易从一个局部最优跳到另一个局部最优，准确度下降。

1.3 小批量梯度下降法（MBGD）

1.3.1 原理

小批量梯度下降法最终结合了上述两种方法的优点，在每次更新时随机在训练集中选取一个 mini-batch，每个 mini-batch 包含 n 个样本；在每个 mini-batch 里计算每个样本的梯度，然后在这个 mini-batch 里求和取平均作为最终的梯度来更新参数。

1.3.2 实现

```
    #基于多元回归的应用 y = x0 + x1, X = [x0 , x1]
    def MBGD(X,y):
        ept = 0.001   #精度
        loss = 1      #损失
        alpha = 0.01  #学习率
        max_iter = 0  #梯度更新次数
        numsSample = X.shape[0]#样本数量
        theta=np.random.randint(1,10,(X.shape[1],1)) #初始化
        while max_iter <= 10000 or loss < ept:
            #这里的小批量梯度下降每次选取两个样本
            i = np.random.randint(0, numsSample-1)  # 随机抽取一个样本
            partial=(1/2)*X[i:i+2,:].T.dot(X[i:i+2,:].dot(theta)-y[i:i+2,:]) #损失函数关于 theta 的偏导数
            theta=theta-alpha*partial
            max_iter+=1
            loss=(1/(2*X.shape[0]))*np.sum((X.dot(theta)-y)**2)
        return max_iter,theta
```

1.3.3 特点

- 小批量梯度下降法既保证了训练的速度，又能保证最后收敛的准确率。
- 在迭代的过程中，因为噪音的存在，学习过程会出现波动。因此，它在最小值的区域徘徊，不会收敛。学习过程会有更多的振荡，为更接近最小值，需要增加学习率衰减项，以降低学习率，避免过度振荡。

2 梯度下降法优化

梯度下降算法有两个重要的控制因子：一个是步长，由学习率控制；一个是方向，由梯度指定。因此，要想对梯度下降的 “快” 和 “准” 实现调控，就可以通过调整它的两个控制因子来实现。因梯度方向已经被证明是变化最快的方向，很多时候都会使用梯度方向，而另外一个控制因子学习率则是解决上述影响的关键所在，换句话说，学习率是最影响优化性能的超参数之一。

在上述梯度下降法中，主要区别是计算梯度时所用样本量的不同，而导致不同的时间复杂度和收敛率，对于学习率，一般采取的固定学习率的方法。但是选择合适的学习率比较困难，需在保证快速迭代与稳定收敛之间取得平衡，而事先指定的学习率变化函数难以适应大样本数据的特性。若想在固定学习率的基础上改进，可以设定多个不同的学习率，循环往复作为学习率序列[3]，在一定程度上能加快收敛速度，但仍具有较大的随机性。目前主流的学习率设定类型分为不同的参数使用不同的学习率、动态调整学习率和自适应学习率[4][5]。

2.1Momentum

2.1.1 原理

SGD 在遇到沟壑时容易陷入震荡。为此，可以为其引入动量 Momentum，加速 SGD 在正确方向的下降并抑制震荡。动量梯度下降法（Momentum）的基本的想法就是计算梯度的指数加权平均数，并利用该梯度更新权重。形象地来说，Momentum 算法借用了物理中的动量概念，它模拟的是物体运动时的惯性，即更新的时候在一定程度上保留之前更新的方向，同时利用当前 batch 的梯度微调最终的更新方向。这样一来，可以在一定程度上增加稳定性，从而学习地更快，并且还有一定摆脱局部最优的能力。

$$v_t = \gamma v_{t-1} + (1-\gamma)\nabla \mathrm{J}(\Theta)$$
$$\Theta = \Theta - v_t$$

2.1.2 实现

```
#多元线性回归  y = x0 + x1, X = [x0 , x1]
def momentum(x,y):
    m, dim = x.shape
    theta = np.zeros(dim)
```

```
    momentum = 0.1  # 冲量
    ept = 0.0001
    loss = 0
    gradient = 0
    max_iter = 0
    while max_iter <= 10000 or loss < ept:
        j = max_iter % m
        loss = 1 / (2 * m) * np.dot((np.dot(x, theta) - y).T , (np.dot(x, theta) - y))
        gradient = momentum * gradient + (1 - momentum) * (x[j] * (np.dot(x[j], theta) - y[j]))
        theta -= gradient
        max_iter += 1
```

2.1.3 特点

- 与梯度下降相比，下降速度快，因为如果方向是一直下降的，那么速度将是之前梯度的和，所以比仅用当前梯度下降快。对于窄长的等梯度图，会减轻梯度下降的震荡程度，因为考虑了当前时刻是考虑了之前的梯度方向，加快收敛。
- 下降中后期，在局部最小值震荡，此时梯度趋近于 0，累计的动量能使权重更新的幅度加大，容易跳过最优值。

2.2Nesterov Accelerated Gradient(NAG)

2.2.1 原理

与动量梯度下降法类似，NAG 让每一次的参数更新方向不仅取决于当前位置的梯度，还受到上一次参数更新方向的影响。形象地来说就是物体在向下运动的过程中，能够提前预知坡面上升的大致位置，随即开始减速。

$$v_t = \gamma v_{t-1} + (1-\gamma)\nabla J(\Theta - \gamma v_{t-1})$$
$$\Theta = \Theta - v_t$$

2.2.2 实现

```
#多元线性回归  y = x0 + x1, X = [x0 , x1]
def momentum(x,y):
    m, dim = x.shape
    theta = np.zeros(dim)
    momentum = 0.1  # 冲量
    ept = 0.0001
    loss = 0
    gradient = 0
    max_iter = 0
```

```
    while max_iter <= 10000 or loss < ept:
        j = max_iter % m
        loss = 1 / (2 * m) * np.dot((np.dot(x, theta) - y).T , (np.dot(x, theta - momentum * gradient) - y))
        gradient = momentum * gradient + (1 - momentum) * (x[j] * (np.dot(x[j], theta - momentum * gradient) - y[j]))
        theta -= gradient
        max_iter += 1
```

2.2.3 特点

- 相对于普通 Momentum，NAG 算法的改进在于，以“向前看”看到的梯度而不是当前位置梯度去更新.经过变换之后的等效形式中，NAG 算法相对于普通 Momentum 多了一个本次梯度相对上次梯度的变化量，这个变化量本质上是对目标函数二阶导的近似。由于利用了二阶导的信息，NAG 算法才会比普通 Momentum 以更快的速度收敛。
- 和 Momentum 一样，NAG 的学习率是固定的，并不适用于收敛的各个阶段

2.3AdaGrad

2.3.1 原理

在前面所介绍的动量法中，虽然优化了梯度方向，解决了局部极小值和梯度中的噪音问题，但目标函数自始自终都是使用同一个学习率。现开始介绍自适应学习率的优化算法[6]。

AdaGrad 算法会根据自变量在每个维度的梯度值大小来调整各个维度上的学习率，从而避免统一的学习率难以适应所有维度的问题。具体而言，对于经常更新的参数，我们已经积累了大量关于它的知识，不希望被单个样本影响太大，希望学习速率慢一些；对于偶尔更新的参数，我们了解的信息太少，希望能从每个偶然出现的样本身上多学一些，即学习速率大一些。

$$g_{t,i} = \nabla_{\Theta_t} J(\Theta_{t,i})$$
$$G_{t,i} = G_{t,i+1} + {g_{t,i}}^2$$
$$\Theta_{t+1,i} = \Theta_{t,i} - \frac{\alpha}{\sqrt{G_{t,i} + \varepsilon}} \cdot g_{t,i}$$

迭代过程中，每个$G_{t,i}$累计了目标函数对参数Θ的梯度的平方，随迭代的进行越来越大，等价于对学习率进行了缩放，起到自适应调整学习率的效果。

2.3.2 实现

```
#多元线性回归  y = x0 + x1, X = [x0 , x1]
def AdaGrad(x,y):
    m, dim = x.shape
    theta = np.zeros(dim)
```

```
    alpha = 0.01
    ept = 0.0001
    loss = 0
    e = 0.00000001
    b2 = 0.999
    mt = np.zeros(dim)
    vt = np.zeros(dim)
    max_iter = 0
    while max_iter <= 10000 or loss < ept:
        j = max_iter % m
        loss = 1 / (2 * m) * np.dot((np.dot(x, theta) - y).T , (np.dot(x, theta) - y))
        gradient = x[j] * (np.dot(x[j] , theta) - y[j])
        vt = b2 * vt + (1 - b2) * (gradient**2)
        vtt = vt / (1 - (b2**(i + 1)))
        vtt_sqrt = np.array([math.sqrt(vtt[0]),
                             math.sqrt(vtt[1])])
        theta = theta - alpha * gradient / (vtt_sqrt + e)
```

2.3.3 特点

- 迭代前期 Gt 较小的时候能够放大梯度，迭代后期能够约束梯度，适合处理稀疏梯度
- 依赖于人工设置的一个全局学习率，依然能对后续学习率缩放产生影响。且在迭代后期，随着 Gt 的不断增加，分母上的累计越来越大，学习率下降过快，使得训练提前结束

2.4RMSProp

2.4.1 原理

为了解决 AdaGrad 会累计梯度平方导致梯度消失的问题，RMSProp 仅是计算对应的平均值，可以缓解 AdaGrad 学习率下降过快的问题。

2.4.2 特点

- 相比于 AgaGrad 算法，这种方法很好的解决了深度学习中训练过早结束的问题。适合处理非平稳目标，对于 RNN 效果很好。
- 又引入了新的超参数，衰减系数 ρ \rho ρ 。依然依赖于全局学习率

3 总结

算法	优点	缺点	适用情况
BGD	目标函数为凸函数时，可以找到全局最优值	收敛速度慢，需要用到全部数据，内存消耗大	不适用于大数据集，不能在线更新模型
SGD	仅需要计算一个样本的梯度，训练速度很快	很容易受噪声影响	适用于大规模训练样本
MBGD	避免冗余数据的干扰，收敛速度加快，能够在线学习	更新值的方差较大，收敛过程会产生波动，可能落入极小值	适用于需要在线更新的模型，适用于大规模训练样本情况
Momentum/NAG	能够在相关方向加速 SGD，抑制振荡，从而加快收敛	需要人工设定学习率	适用于有可靠的初始化参数
AdaGrad	实现学习率的自动更改	仍依赖于人工设置一个全局学习率，学习率设置过大，对梯度的调节太大。中后期，梯度接近于 0，使得训练提前结束	需要快速收敛，训练复杂网络时；适合处理稀疏梯度

3 文献

[1] 孙娅楠,林文斌.梯度下降法在机器学习中的应用[J].苏州科技大学学报(自然科学版),2018,35(02):26-31.

[2] 王丹. 随机梯度下降算法研究[D].西安建筑科技大学,2020.DOI:10.27393/d.cnki.gxazu.2020.000978.

[3] 郭跃东,宋旭东.梯度下降法的分析和改进[J].科技展望,2016,26(15):115+117.

[4] 小胖蹄儿. learning rate 四种改变方式. CSDN

[5] 宋美佳,贾鹤鸣,林志兴,卢仁盛,刘庆鑫.自适应学习率梯度下降的优化算法[J].三明学院学报,2021,38(06):36-44.DOI:10.14098/j.cn35-1288/z.2021.06.006.

[6] 王昕.梯度下降及优化算法研究综述[J].电脑知识与技术,2022,18(08):71-73.