Bfstack 机器学习基础系列分享

An Overview of Gradient Descent Optimization Algorithms

张晓龙 – 2018-2-10

目录

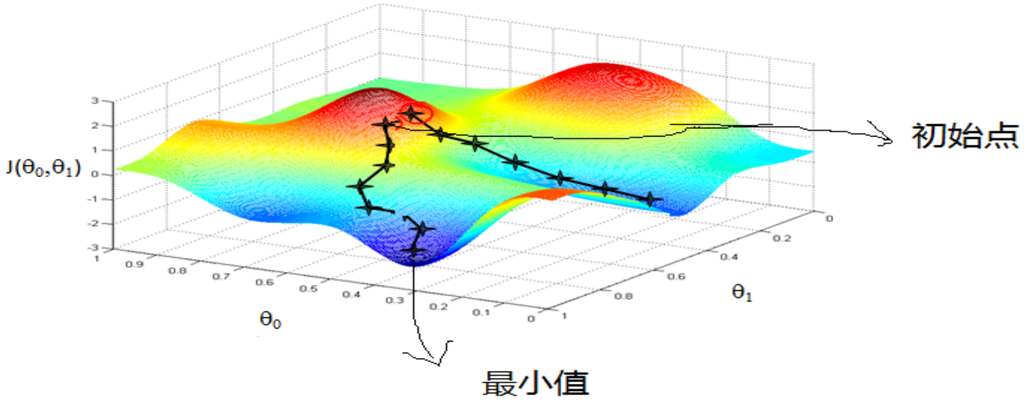
1. 梯度下降、及算法家族
2. 随机梯度下降、问题和挑战
3. 随机梯度下降的优化算法
4. 并行和分布式架构
5. 随机梯度下降的其他优化方法

分享目的：不同算法的原理和效果展示，帮助在实际问题中更合理的选用梯度下降算法

梯度下降

- 无约束优化问题时,一阶最优化算法，通常也称为**最速下降法**
- 目标函数J(θ)的局部最小值，向函数上当前点对应梯度（或者是近似梯度）的*反方向*（-∇θJ(θ) ）的规定步长距离点进行迭代搜索

$$\theta_j = \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta) \quad \alpha \text{ 为学习率}$$



梯度下降法家族

- Batch gradient descent
- Stochastic gradient descent
- Mini-batch gradient descent
- 困难：
  - 梯度计算：目标函数一般是求和形式的函数
    - 样本量极大时，梯度的计算变得非常耗时耗力
  - 学习率的选择：过小会导致算法收敛慢，过大容易导致算法不收敛

$$J_X(\theta) = \sum_i J_{x_i}(\theta)$$

Batch gradient descent

- 在整个数据集上计算损失函数的梯度来更新参数 θ：
- 每一轮的迭代需要计算整个数据集

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta).$$

  - 每调节一个 θj，都需要遍历一遍样本集,批量梯度训练耗时比较长
  - 内存中加载数据问题
  - 不能支持online更新model
  - 可化解为向量型表示，所以就能利用到**并行计算**优化性能
- 能保证凸函数（损失函数）梯度下降法一定是全局最优解，非凸函数为局部最小值

```
for i in range(nb_epochs):
  params_grad = evaluate_gradient(loss_function, data, params)
  params = params - learning_rate * params_grad
```

Stochastic gradient descent

- 随机梯度下降直接根据每一个输入样本做参数更新
- 同批量梯度下降相比
  - 速度快：不会做重复计算
  - 会造成Objection function波动 - 以高variance更新
    - 有可能找到更好的局部最小值
  - 迭代过程中缓慢减小学习率，收敛行为同batch类似，对凸或者非凸函数，几乎可以收敛于全局最小或者局部最小

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i)}; y^{(i)}).$$

```
for i in range(nb_epochs):
  np.random.shuffle(data)
  for example in data:
    params_grad = evaluate_gradient(loss_function, example, params)
    params = params - learning_rate * params_grad
```

手段	概括	优点	缺点
批量梯度下降法	尽可能减小训练样本的**总的**预测代价	能够获得最优解，支持并行计算	样本容量较大时，性能显著下降
随机梯度下降法	尽可能的减小**每个**训练样本的预测代价	训练速度快	并不一定能获得全局最优，经常出现抖动和噪音，且不能通过并行计算优化

Mini-batch gradient descent

- 结合上面两种算法的优势
  - 降低了更新参数的方差（variance），使得收敛过程更为稳定
  - 利用最新的深度学习程序库中高度优化的矩阵运算器，能够高效地求出每小批数据的梯度

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i:i+n)}; y^{(i:i+n)}).$$

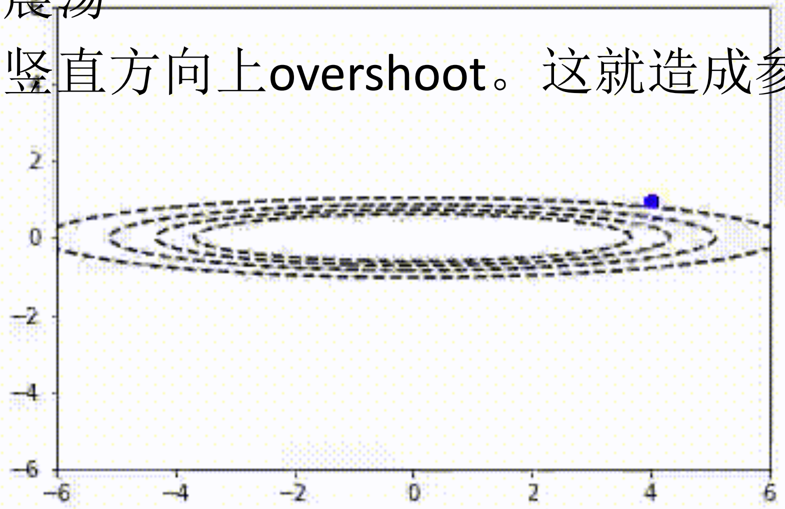
```
for i in range(nb_epochs):
  np.random.shuffle(data)
  for batch in get_batches(data, batch_size=50):
    params_grad = evaluate_gradient(loss_function, batch, params)
    params = params - learning_rate * params_grad
```

问题和挑战

- **局部梯度的反方向不一定是函数整体下降的方向**
  - 隧道型函数，梯度下降表现不佳
- **预定学习率衰减的问题**
  - 太小导致收敛速度慢，太大导致损失函数在最小值附近波动甚至偏离
  - 调整学习率（动态算法：退火算法、阈值设置），需要提前设置，一般难以和数据集特征吻合
- **对不同参数采取不同的学习率的问题**
  - 如果数据稀疏，同时特征频率差异大，对于出现次数较少的特征，我们对其执行更大的学习
- **局部最小值问题（极小值、鞍点）**
  - 优化非凸的损失函数可能在多个局部最小值处徘徊，不能够跳出
  - 鞍点：比起局部极小值，鞍点更可怕
    - 就是在一个方向上斜率是正的、这些鞍点通常由一些函数值相同的面环绕，它们在各个方向的梯度值都为 0，所以 SGD 很难从这些鞍点中脱开

Momentum（动量法）

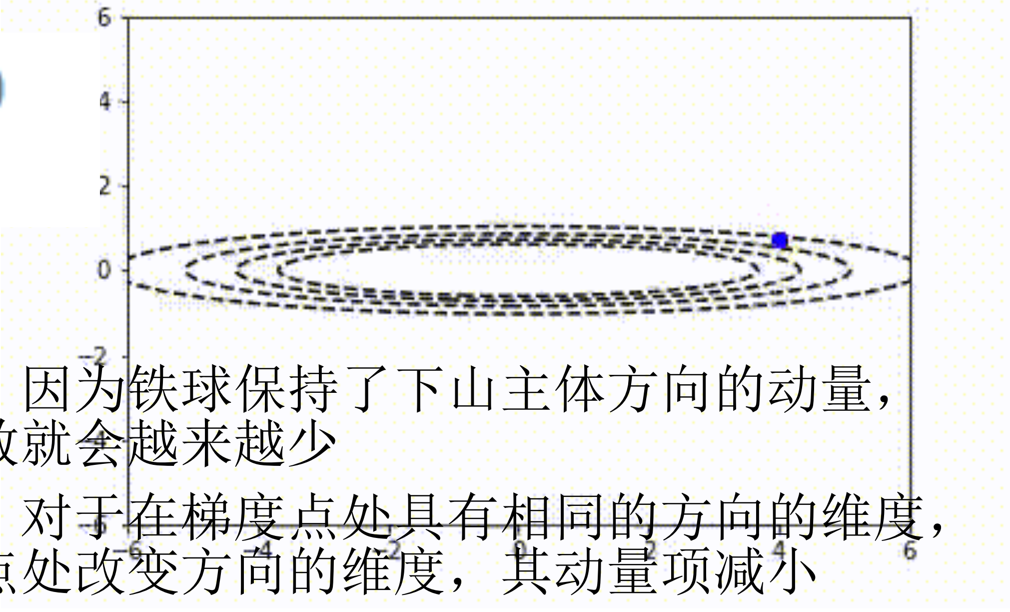
- 在狭长的隧道型函数上表现不佳，如下图
  - 函数主体缓缓向中心下降
  - 在主体方向两侧各有一面高墙，导致垂直于主体方向有更大的梯度
  - 梯度下降法会在隧道两侧频繁震荡
  - 较小的学习率从而避免参数在竖直方向上overshoot。这就造成参数向最优解移动速度的缓慢



Momentum（动量法）

- 帮助SGD在相关方向上加速并抑制摇摆的一种方法
  - 适用于隧道型曲面
- 动量法将历史步长的更新向量的一个分量γ增加到当前的更新向量中

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$$
$$\theta = \theta - v_t$$



- 物理上就像推一个铁球下山，因为铁球保持了下山主体方向的动量，所以在隧道上沿两侧震荡次数就会越来越少
- 收敛加速、减少摇摆的原因：对于在梯度点处具有相同的方向的维度，其动量项增大，对于在梯度点处改变方向的维度，其动量项减小

Nesterov accelerated gradient（NAG）

- 动量法的一个问题在于：从山顶退下的铁球会越来越快，以至于到了山底停不下来，希望算法更聪明一些，可以在到达底部之前就自己刹车
- NAG是一种能够给动量项这样的预知能力的方法
- 利用主体下降方向提供的先见之明，预判自己下一步的位置，并到预判位置计算梯度
- 通过基于未来参数的近似值而非当前的参数值计算相得应罚函数 J(θ-γvt-1) 并求偏导数，我们能让优化器高效地「前进」并收敛

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta - \gamma v_{t-1})$$
$$\theta = \theta - v_t$$

Nesterov accelerated gradient（NAG）

- 动量法首先计算当前的梯度值（小的蓝色向量），然后在更新的累积梯度（大的蓝色向量）方向上前进一大步
- NAG首先（试探性地）在先前累积梯度（棕色的向量）方向上前进一大步，根据当前地情况修正，以得到最终的前进方向（绿色向量）
- 这种基于预测的更新方法，使我们避免过快地前进，并提高了算法地响应能力（responsiveness），大大改进了 RNN 在一些任务上的表现

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta - \gamma v_{t-1})$$
$$\theta = \theta - v_t$$

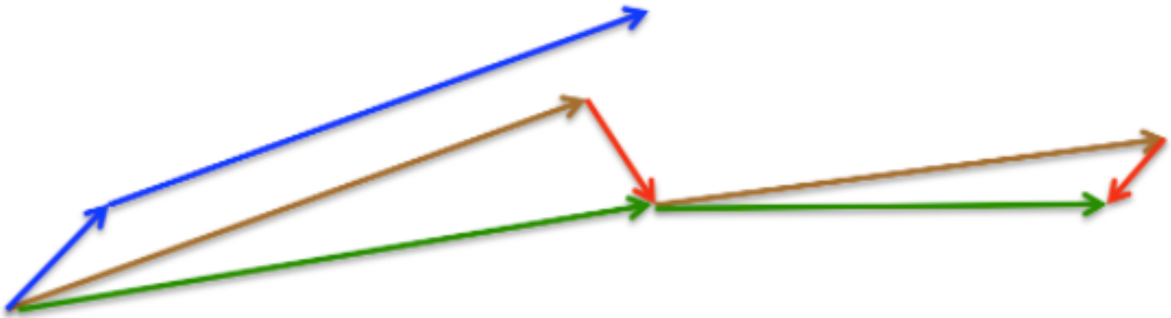


Image 4: Nesterov update (Source: G. Hinton's lecture 6c)

Adagrad

- 自动调整学习率，适用于稀疏数据
- 梯度下降法在每一步对每一个参数使用相同的学习率，这种一刀切的做法不能有效的利用每一个数据集自身的特点
- Adagrad是一种自动调整学习率的方法
  - 随着模型的训练，学习率自动衰减
  - 对于更新频繁的参数，采取较小的学习率
  - 对于更新不频繁的参数，采取较大的学习率

Adagrad

- 在迭代次数 t 下，对参数θi 求目标函数梯度的结果为 gt,i，则普通的SGD更新规则变为：

$$g_{t,i} = \nabla_\theta J(\theta_{t,i}).$$

$$\theta_{t+1,i} = \theta_{t,i} - \eta \cdot g_{t,i}.$$

- 对于上述的更新规则，在t时刻，基于对θi计算过的历史梯度，Adagrad修正了对每一个参数θi的学习率

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{t,i}.$$

- **Gt（对角矩阵）**的对角线上包含了关于所有参数θ的历史梯度的平方和，通过Gt和gt之间的元素向量乘法⊙向量化上述的操作

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t.$$

Adagrad

- 主要优点是无需手动调整学习率。在大多数的应用场景中，通常采用常数0.01
- **主要缺点**是它在分母中累加梯度的平方
  - 由于每增加一个正项，在整个训练过程中，累加的和会持续增长。这会导致**学习率变小以至于最终变得无限小**，在学习率无限小时，Adagrad算法将无法取得额外的信息
- 实现：

```
# Adagrad算法
def adagrad(params, sqrs, lr, batch_size):
    eps_stable = 1e-7
    for param, sqr in zip(params, sqrs):
        g = param.grad / batch_size
        sqr[:] += nd.square(g)
        div = lr * g / nd.sqrt(sqr + eps_stable)
        param[:] -= div
```

Adadelta

- Adadelta是Adagrad的一种改进算法，旨在解决学习率不断单调下降的问题
- Adadelta 使用梯度平方的移动平均来取代全部历史平方和
- 定义移动平均：当前时间的梯度均值是基于过去梯度均值和当前梯度值平方的加权平均，其中是类似上述动量项的权值

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma)g_t^2$$

- 于是就得到参数更新法则

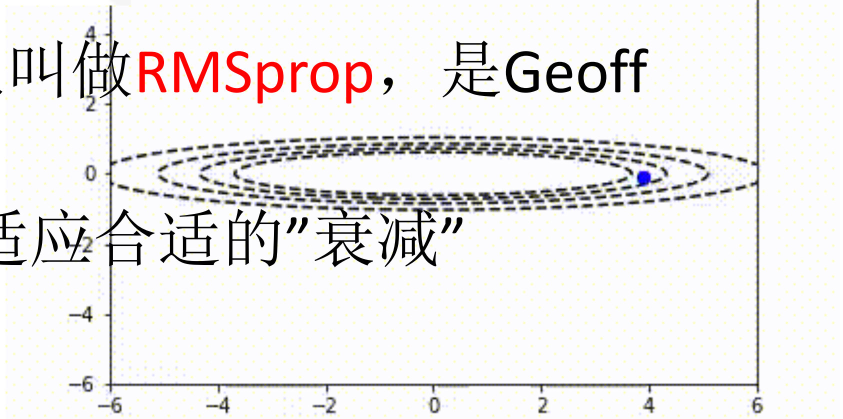
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

Adadelta

- Adagrad以及一般的梯度下降的另外一个问题：**梯度与参数的单位不一致**
- Adadelta使用参数更新的移动平均来取代学习率，参数更新法则：

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\sqrt{E[\Delta\theta]_{t-1}}}{\sqrt{E[g^2]_{t,ii} + \epsilon}} \cdot g_{t,i}$$

- 注意：Adadelta的第一个版本也叫做RMSprop，是Geoff Hinton独立于Adadelta提出的
- 学习率不一定一直衰减，是自适应合适的"衰减"



Adam：自适应矩估计（Adaptive Moment Estimation）

- 将动量法和 Adadelta算法结合成一种算法，以获得两全其美的效果
- 如果把Adadelta里面梯度的平方和看成是梯度的二阶矩，那么梯度自身的求和就是一阶矩。
- Adam算法在Adadelta的二阶矩的基础上又引入一阶矩。而一阶矩，其实就类似于动量法里面的动量

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$$

- 于是参数更新法则为 $$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t + \epsilon}} \hat{m}_t$$
- 注意：实际操作中，vt和mt都采用更好的无偏估计，来避免前几次更新时候数据不足的问题

AMSGrad

- 为了解决指数滑动平均的影响
- 使用过去梯度平方的最大值以替代指数平均值来更新参数。不带偏差修正估计的 AMSGrad 更新规则可以表示为

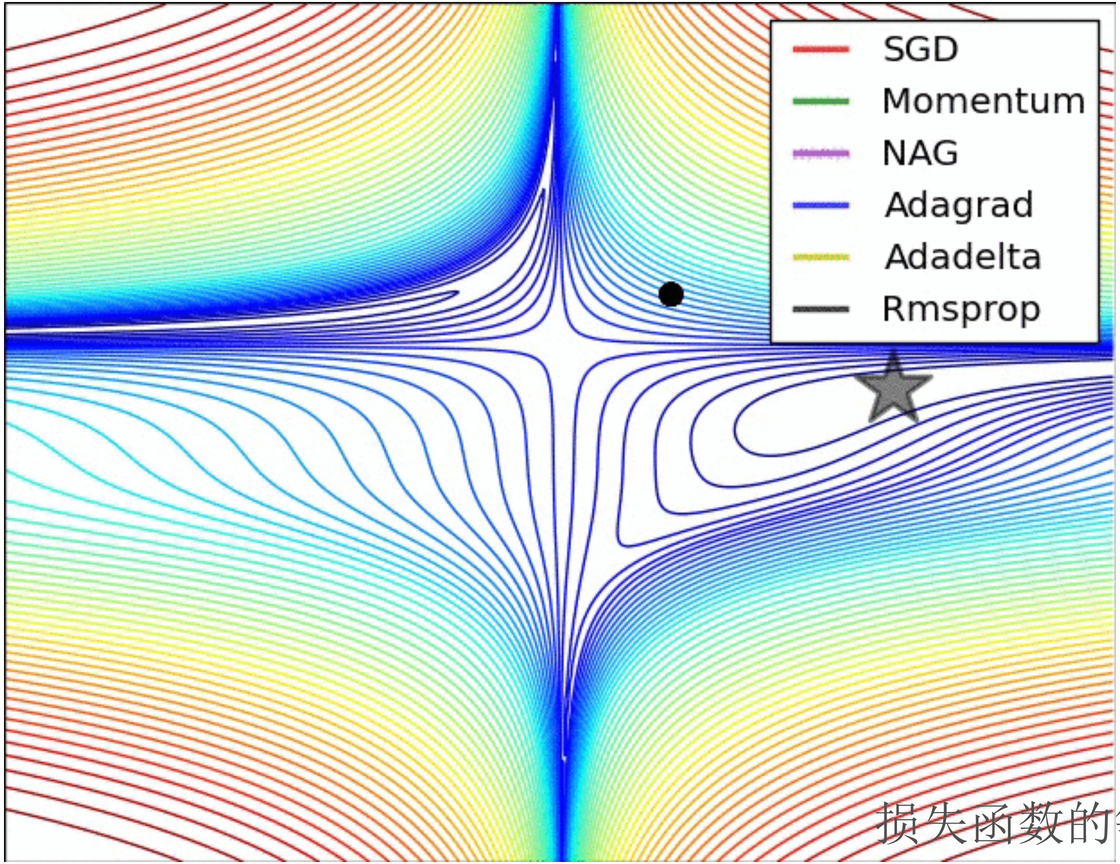
$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$$

$$\boxed{\hat{v}_t = \max(\hat{v}_{t-1}, v_t)}$$

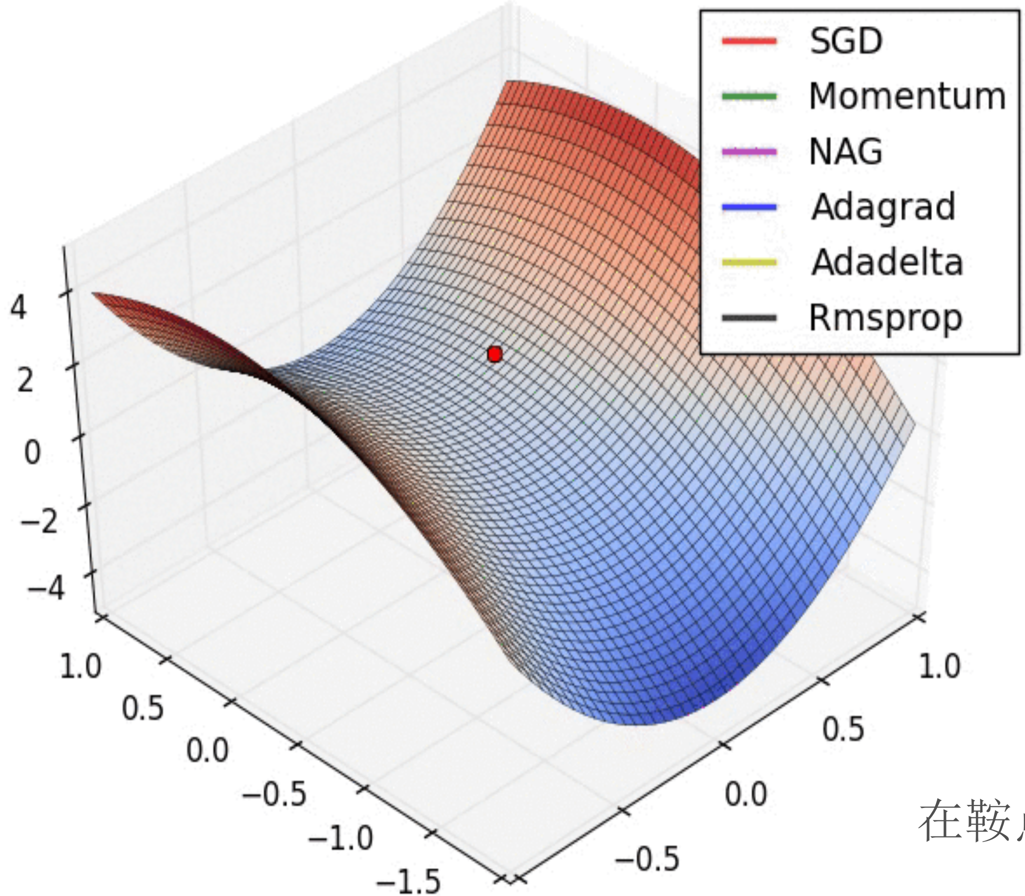
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} m_t$$

算法可视化展示



损失函数的等高线上SGD优化

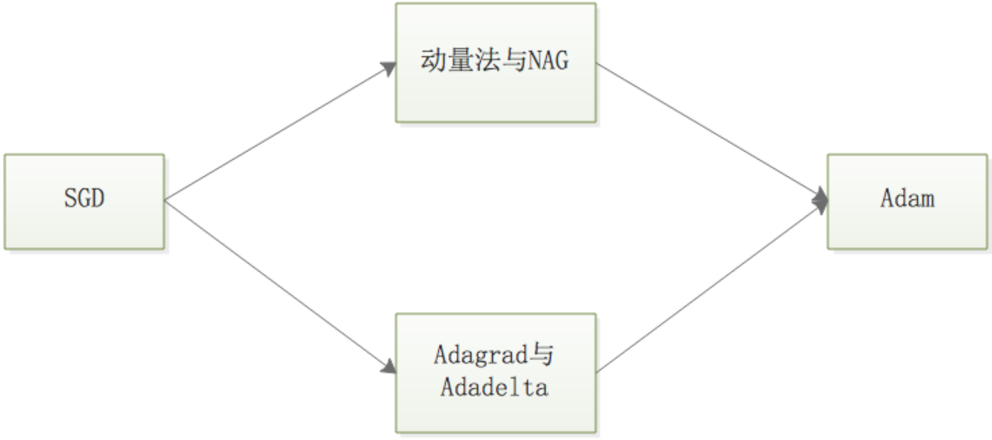
算法可视化展示



在鞍点处的SGD优化

选择合适的优化算法

- 动量法与NAG的改进方法着重解决目标函数像崎岖的问题
- Adagrad与Adadelta主要解决学习率更新的问题
- Adam集中了前述两种做法的主要有点
- 目前为止Adam可能是几种算法中综合表现最好的



并行和分布式SGD

- Hogwild!
- Downpour SGD， tensorflow前身
- Delay-tolerant Algorithms for SGD
- TensorFlow
- Elastic Averaging SGD

优化SGD的一些tricks

- Shuffling and Curriculum Learning（重排法和递进学习）
  - 每次大循环前洗牌（提升模型稳定性）
  - 人为对数据排序
- Batch normalization（批量归一化）
  - 通常用0均值和单位方差初始化参数的初始值来归一化，以帮助模型进行学习。
  - 随着不断训练，参数得到不同的程度的更新，我们失去了这种归一化，随着网络变得越来越深，这种现象会降低训练速度，且放大参数变化。
  - 在模型中加入批量标准化后，能使用更高的学习率且不要那么在意初始化参数。
  - 批量正则化还可以看作是一种正则化手段，能够减少（甚至去除）留出法的使用。

优化SGD的一些tricks

- Early stopping
  - Geoff Hinton: "*Early stopping (is) beautiful free lunch*”
  - 在训练过程中，时刻关注模型在验证集上的误差情况，并且在误差没有明显改进的时候停止训练
- Gradient noise
  - Neelakantan等人在每个梯度更新中增加满足高斯分布 N(0,σ^2) 的噪声
  - 在每一次更新中加入人为噪音，帮助逃离鞍点和局部极小值点

牛顿法

- 为什么不用牛顿法？
  - 牛顿法要求计算目标函数的二阶导数（Hessian matrix），在高维特征情形下这个**矩阵非常巨大**，计算和存储都成问题
  - 在使用小批量情形下，牛顿法对于二阶导数的估计**噪音太大**
  - 在目标函数非凸时，牛顿法更容易收到**鞍点**甚至最大值点的吸引

Reference

- 统计学习方法. 李航
- An overview of gradient descent optimization algorithms
- http://www.julyedu.com/video/play/69/646
- http://cs231n.github.io/
- http://zh.gluon.ai/chapter_optimization/index.html
- http://www.cs.toronto.edu/~tijmen/csc321/
- https://www.jiqizhixin.com/articles/2017-12-06
- 深度解读最流行的优化算法：梯度下降
- 目标函数的经典优化算法介绍


好 课 就 在 跟 谁 学