优化算法

Created	@July 27, 2022 3:44 PM
Class	interview
Type	
Materials	
Reviewed	☐

梯度

微积分中用梯度表示函数增长最快的反向，因此神经网络中采用负梯度来表示损伤函数下降最快的方向

梯度实际上是损伤函数对每个参数的偏导组成的向量

随机梯度下降（SGD)

在深度学习中，SGD以及其变种是应用最广泛的优化算法，虽然其存在着陷入局部最优的缺点，但是只是针对一个参数来说，即使其中一个参数陷入局部最优点，但是其他参数还是会继续更新，所以大概率会将陷入局部最优的那个参数脱离局部最优点，因此这个算法依旧可以work

随机梯度下降是随机抽取一批样本（batch），每次使用m个样本来近似整个数据集的平均损失

$$g \leftarrow \frac{1}{m}\nabla_\theta \sum_i J(f(\boldsymbol{x}^{(i)};\theta), y^{(i)})$$

$$\theta \leftarrow \theta - \epsilon g$$

随机梯度下降还可能出现**“峡谷”**和**“鞍点”**的问题

改进方向：引入历史梯度的一阶动量：Momentum方法

引入历史梯度的二阶动量：AdaGrad、RMSProp

引入历史梯度的一阶和二阶动量：Adam

这里实现一个简单的SGD

```
def sgd(params, lr, batch_size):
    with torch.no_gard():
        for param in params:
             param -= lr * param.gard / batch_size
             param.grad.zero_()
```

Momentum算法

动量算法一方面是为了解决峡谷和鞍点问题，另一方面也是为了SGD加速

如果把原始的SGD算法想象成一个纸团在重力下滚动，由于质量小受到山壁弹力的干扰大，来回震荡滚动，在鞍点处速度很快就减为0，无法离开平地

而动量方法就相当于为SGD算法把纸团换成了一个铁球，不容易受到外力的干扰，轨迹更加稳定，而且在鞍点时由于惯性的作用，更有可能离开平地

$$v \leftarrow \alpha v - \epsilon \nabla_\theta (\frac{1}{m}\sum_{i=1}^{m} J(f(\boldsymbol{x^{(i)}};\theta), y^{(i)}))$$

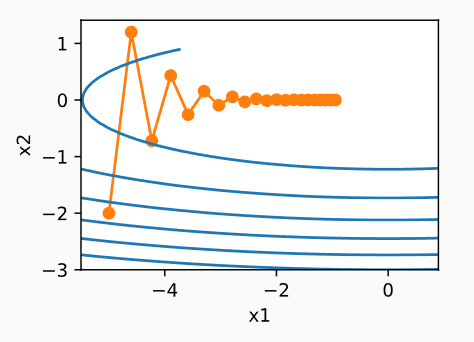
$$\theta \leftarrow \theta + v$$

动量算法引入了历史梯度的一阶动量，使得当前迭代点的下降方向不仅仅取决于当前的梯度，还受到前面所有迭代点的影响。

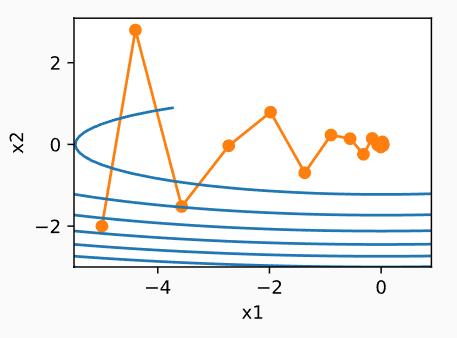
二者对比实验：

通过一个椭球目标，来看看如何执行传统的梯度下降

$$f(\mathbf{x}) = 0.1x_1^2 + 2x_2^2.$$



当使用了动量法后，收敛速度加快了



NAG

动量算法是将历史梯度和当前的梯度进行合并，来计算下降的方向，而NAG方法是让迭代点先按照历史梯度走一步，然后再合并，公式如下：

$$v \leftarrow \alpha v - \epsilon \nabla_\theta (\frac{1}{m} \sum_{i=1}^{m} J(f(\boldsymbol{x^{(i)}}; \theta + \alpha v), y^{(i)}))$$

$$\theta \leftarrow \theta + v$$

自适应学习率算法

AdaGrad

算法的思想是独立地适应模型的每个参数，一个较大偏导的参数应该有一个较小的学习率，初始学习率会下降的比较快，而一直小偏导的参数应该有一个较大的学习率，初始学习率会下降的比较慢

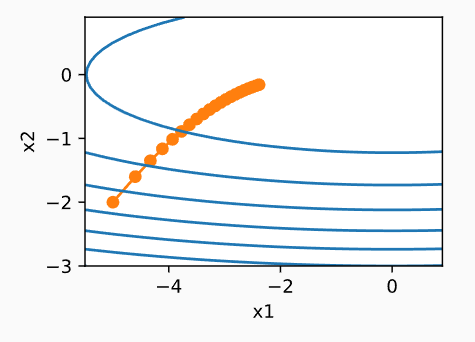
每个参数的学习率会缩放个参数反比于其历史梯度平方值总和的平方根

$$g \leftarrow \frac{1}{m} \nabla_\theta \sum_i J(f(\boldsymbol{x}^{(i)}; \theta), y^{(i)})$$

$$r \leftarrow r + g \odot g$$

$$\theta \leftarrow \theta - \frac{\epsilon}{\delta + \sqrt{r}} \odot g$$

缺点是随着时间的增加，历史梯度在分母上的积累会越来越大，所以学习率就会越来越小，使得在中后期网络的学习能力越来越弱



RMSProp

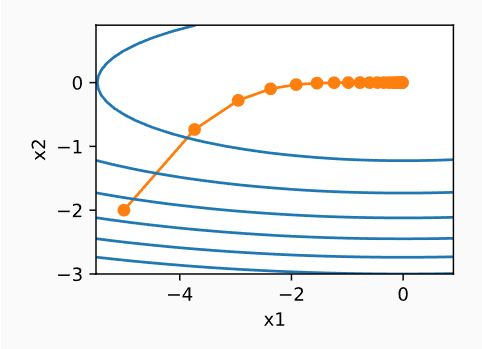
由于AdaGrad方法中的学习率会存在过度衰减的问题，AdaGrad根据所有历史梯度来收缩学习率，可能使得整个网络还未收敛时，学习率就已经变得很小，难以继续训练

RMSProp通过动量方法中的指数加权移动平均值来代替历史梯度的总和

$$g \leftarrow \frac{1}{m}\nabla_\theta \sum_i J(f(\boldsymbol{x}^{(i)};\theta), y^{(i)})$$

$$r \leftarrow \rho r + (1-\rho) g \odot g$$

$$\theta \leftarrow \theta - \frac{\epsilon}{\sqrt{\delta + r}} \odot g$$



AdaDelta

AdaDelta和RMSProp一样使用指数加权移动平均值来代替历史梯度的总和，但是AdaDelta没有学习率这一个超参数，而是用一个额外的变量来计算

$$g \leftarrow \frac{1}{m}\nabla_\theta \sum_i J(f(\boldsymbol{x}^{(i)};\theta), y^{(i)})$$

$$r \leftarrow \rho r + (1-\rho) g \odot g$$

$$g' \leftarrow \frac{\sqrt{\delta + \Delta\theta}}{\sqrt{\delta + r}} \odot g$$

$$\theta \leftarrow \theta - g'$$

$$\Delta\theta \leftarrow \rho\Delta\theta + (1-\rho) g' \odot g'$$

Adam

Adam算法的关键组成部分之一是：它使用指数加权移动平均值来估算梯度的动量和二次矩，即它使用状态变量

$$\begin{aligned}\mathbf{v}_t &\leftarrow \beta_1 \mathbf{v}_{t-1} + (1-\beta_1)\mathbf{g}_t,\\ \mathbf{s}_t &\leftarrow \beta_2 \mathbf{s}_{t-1} + (1-\beta_2)\mathbf{g}_t^2.\end{aligned}$$

通常将β1的值设为0.9，而β2的值设为0.999，也就是说，方差估计的移动远远慢于动量估计的移动，在这里要注意一个问题，如果我们初始化v0=s0=0，就会获得一个相当大的初始偏差。

因此

我们可以通过使用$\sum_{i=0}^{t} \beta^i = \frac{1-\beta^t}{1-\beta}$来解决这个问题。 相应地，标准化状态变量由下式获得

$$\hat{\mathbf{v}}_t = \frac{\mathbf{v}_t}{1-\beta_1^t} \text{ and } \hat{\mathbf{s}}_t = \frac{\mathbf{s}_t}{1-\beta_2^t}.$$

有了正确的估计，我们现在可以写出更新方程。 首先，我们以非常类似于RMSProp算法的方式重新缩放梯度以获得

$$\mathbf{g}_t' = \frac{\eta \hat{\mathbf{v}}_t}{\sqrt{\hat{\mathbf{s}}_t} + \epsilon}.$$

最后简单更新：

$$\mathbf{x}_t \leftarrow \mathbf{x}_{t-1} - \mathbf{g}_t'.$$

```
def init_adam_states(feature_dim):
    v_w, v_b = torch.zeros((feature_dim, 1)), torch.zeros(1)
    s_w, s_b = torch.zeros((feature_dim, 1)), torch.zeros(1)
    return ((v_w, s_w), (v_b, s_b))

def adam(params, states, hyperparams):
    beta1, beta2, eps = 0.9, 0.999, 1e-6
    for p, (v, s) in zip(params, states):
        with torch.no_grad():
            v[:] = beta1 * v + (1 - beta1) * p.grad
            s[:] = beta2 * s + (1 - beta2) * torch.square(p.grad)
            v_bias_corr = v / (1 - beta1 ** hyperparams['t'])
            s_bias_corr = s / (1 - beta2 ** hyperparams['t'])
            p[:] -= hyperparams['lr'] * v_bias_corr / (torch.sqrt(s_bias_corr)
                                                       + eps)
        p.grad.data.zero_()
    hyperparams['t'] += 1
```

学习率自适应