深度学习系列（7）：神经网络的优化方法

一、Gradient Descent [Robbins and Monro, 1951, Kiefer et al., 1952]

机器学习中，梯度下降法常用来对相应的算法进行训练。常用的梯度下降法包含三种不同的形式，分别是BGD、SGD和MBGD，它们的不同之处在于我们在对目标函数进行梯度更新时所使用的样本量的多少。

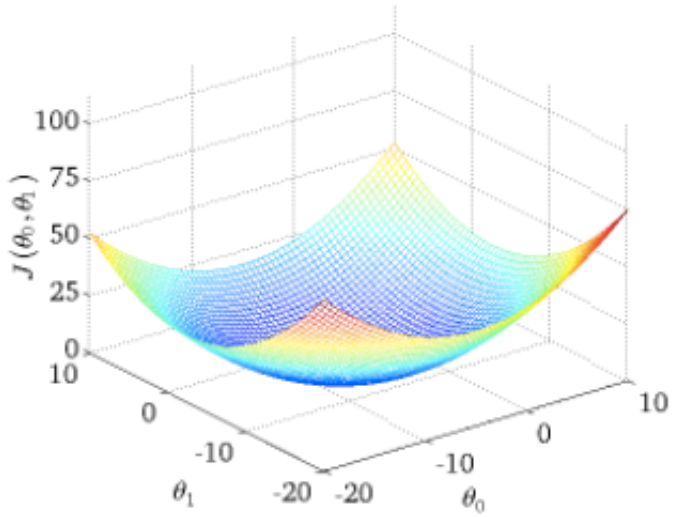
以线性回归算法来对三种梯度下降法进行比较。
一般线性回归函数的假设函数为：

$$h_\theta = \sum_{j=0}^{n} \theta_j x_j$$

（即有n个特征）对应的损失函数为

$$L(\theta) = \frac{1}{2m} \sum_{i=1}^{m} (h(x_i) - y_i)^2$$

下图即为一个二维参数θ_0和θ_1组对应的损失函数可视化图像：



1.1 BGD（Batch Gradient Descent）

批量梯度下降法（Batch Gradient Descent，简称BGD）是梯度下降法最原始的形式，它的具体思路是在更新每一参数时都使用所有的样本来进行更新，其数学形式如下：

- 1）对上述的损失函数求偏导：

$$\frac{\partial L(\theta)}{\partial \theta_j} = -\frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)} - h_\theta\left(x^{(i)}\right)\right)x_j^{(i)}$$

- 2）由于是最小化损失函数，所以按照每个参数θ的梯度负方向来更新每个θ：

$$\theta_j' = \theta_j + \frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)} - h_\theta\left(x^{(i)}\right)\right)x_j^{(i)}$$

  其伪代码如下：

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

从上面的公式可以看到，它得到的是全局最优解，但是每迭代一步，都要用到训练集所有的数据，若样本数目m很大，那么迭代速度会大大降低。
其优缺点如下：

- 优点：全局最优解；易于并行实现；
- 缺点：当样本量很大时，训练过程会很慢

1.2 SGD（Stochastic Gradient Descent）

由于批量梯度下降法在更新每一个参数时，都需要所有的训练样本，所以训练过程会随着样本数量的加大而变得异常缓慢。随机梯度下降法（Stochastic Gradient Descent，简称SGD）正是为了解决批量梯度下降法这一弊端而提出的。
对每个样本的损失函数对θ求偏导得到对应的梯度，来更新θ：

$$\theta_j' = \theta_j + \left(y^{(i)} - h_\theta\left(x^{(i)}\right)\right)x_j^{(i)}$$

具体的伪代码形式为

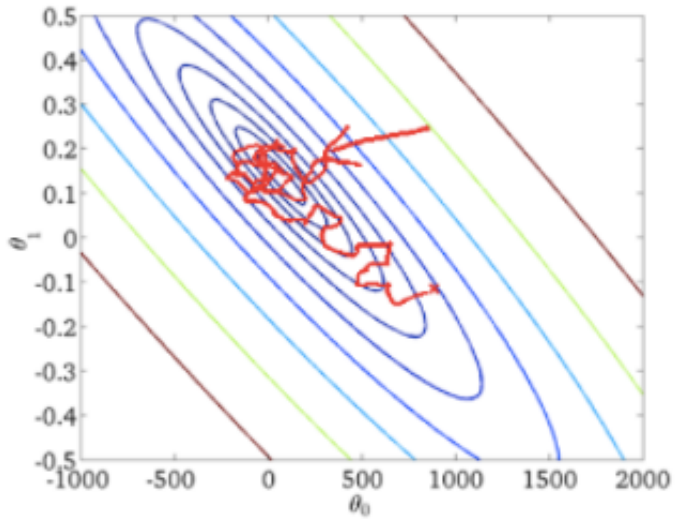
```
for i in range(nb_epochs):
  np.random.shuffle(data)
  for example in data:
    params_grad = evaluate_gradient(loss_function, example, params)
    params = params - learning_rate * params_grad
```

随机梯度下降是通过每个样本来迭代更新一次，如果样本量很大的情况（例如几十万），那么可

能只用其中几万条或者几千条的样本，就已将将θ迭代到最优解了，对比上面的批量梯度下降，迭代一次不可能最优，如果迭代十次的话就需要遍历训练样本10次。但是，SGD伴随的一个问题是噪音较BGD要多，使得SGD并不是每次迭代都向着整体最优化方向。其优缺点如下：

- 优点：训练速度快；
- 缺点：准确度下降，并不是全局最优；不易于并行实现。

从迭代次数上来看，SGD迭代的次数较多，在解空间的搜索过程看起来很盲目。其迭代的收敛曲线示意图表示如下：



1.3 MBGD（Mini-batch Gradient Descent）

从上述的两种梯度下降法可以看出，其各自均有优缺点，那么能否在两种方法的性能之间取得一个折中呢？即，算法的训练过程比较快，而且也要保证最终参数训练的准确率，而这正是小批量梯度下降法（Mini-batch Gradient Descent，简称MBGD）的初衷。

下面的伪代码中，我们每轮迭代的mini-batches设置为50：

```
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
```

1.4 梯度下降算法的局限

虽然梯度下降算法效果很好，并且被广泛的使用，但它存在着一些需要解决的问题：

- 1）首先选择一个合适的学习速率很难。若学习速率过小，则会导致收敛速度很慢。如果学习速率过大，那么会阻碍收敛，即在极值点附近振荡
- 2）学习速率调整（又称学习速率调度，Learning rate schedules）试图在每次更新过程中，改变学习速率，如模拟退火按照预先设定的调度算法或者当相邻的迭代中目标变化小于一个阈值时候减小学习速率。但是梯度下降算法的调度和阈值需要预先设置，无法对数据集特征进行自适应。
- 3）模型所有的参数每次更新都是使用相同的学习速率。如果我们的数据很稀疏并且我们的特征出现的次数不同，我们可能不会希望所有的参数以某种相同的幅度进行更新，而是针对很少出现的特征进行一次大幅度更新。
- 4）在神经网络中常见的极小化highly non-convex error functions的一个关键挑战是避免步入大量的suboptimal local minima。Dauphin等人认为实践中的困难来自saddle points而非local minima。这些saddle points（鞍点）经常被一个相等误差的平原包围，导致SGD很难摆脱，因为梯度在所有方向都近似于0。

二、Momentum

这是一种启发式算法。形式如下：

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

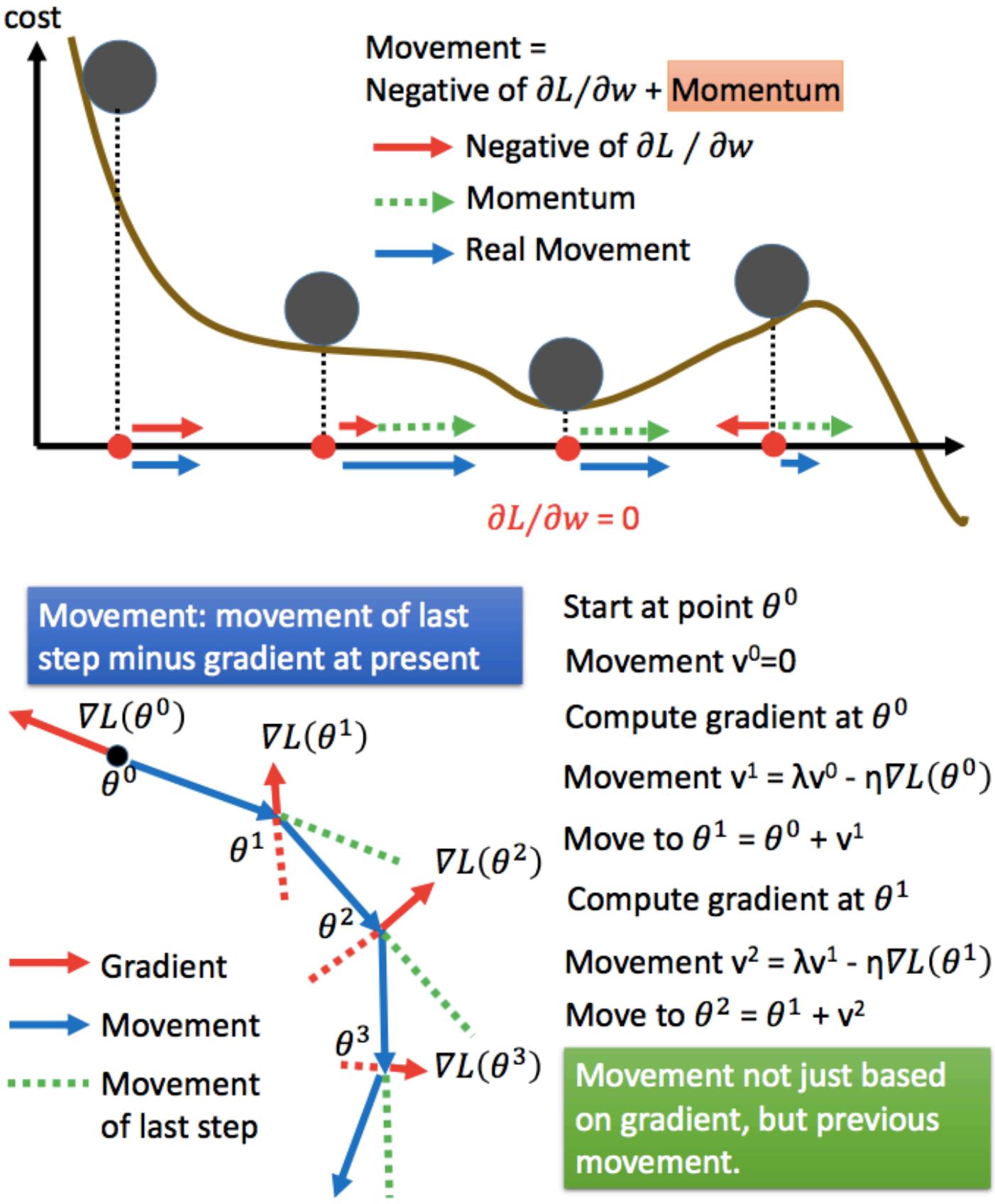
$$\theta = \theta - v_t$$

我们用物理上的动能势能转换来理解它。即物体在这一时刻的动能=物体在上一时刻的动能+上一时刻的势能差。由于有阻力和转换时的损失，所以两者都乘以一个系数。

就像一个小球从坡上向下滚，当前的速度取决于上一时刻的速度和势能的改变量。

这样在更新参数时，除了考虑到梯度以外，还考虑了上一时刻参数的历史变更幅度。例如，参数上一次更新幅度较大，并且梯度也较大，那么在更新时是不是得更加猛烈一些了。这样的启发式算法，从直观感知上确实有道理。

下面两张图直观的展示了Momentum算法，其中绿色箭头表示上一时刻参数的变更幅度，红色箭头表示梯度，两者向量叠加即得到蓝色箭头即真实的更新幅度。



三、NAG（Nesterov accelerated gradient）[Nesterov, 1983]

还是以上面小球的例子来看，momentum方式下小球完全是盲目被动的方式滚下的。这样有个缺

点就是在邻近最优点附近是控制不住速度的。我们希望小球可以预判后面的“地形”，要是后面地形还是很陡峭，那就继续坚定不移地大胆走下去，不然的话就减缓速度。

当然，小球自己也不知道真正要走到哪里，这里以

$$\theta - \gamma v_{t-1}$$

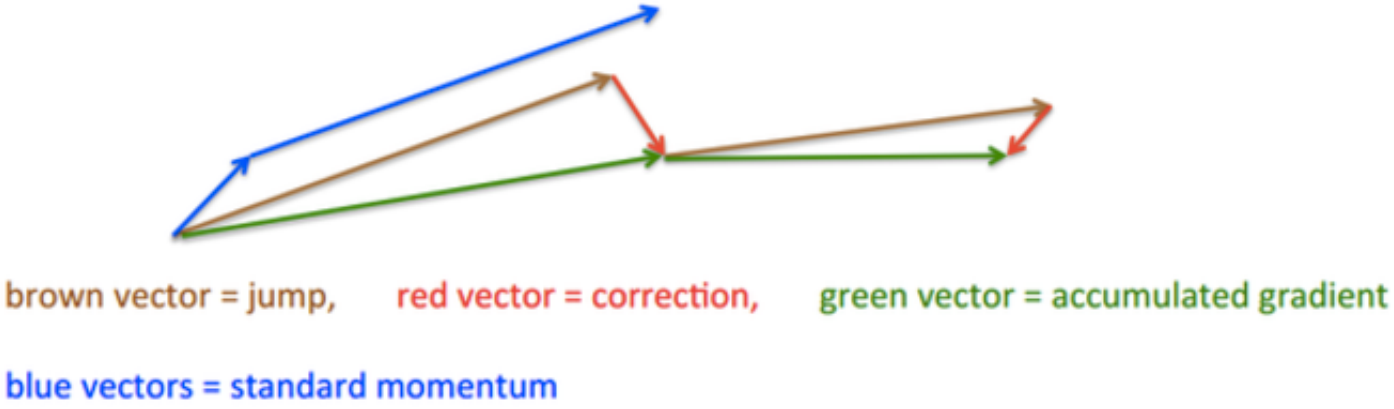
作为下一个位置的近似，将动量的公式更改为：

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

$$\theta = \theta - v_t$$

相比于动量方式考虑的是上一时刻的动能和当前点的梯度，而NAG考虑的是上一时刻的梯度和近似下一点的梯度，这使得它可以先往前探探路，然后慎重前进。

Hinton的slides是这样给出的：



其中两个blue vectors分别理解为梯度和动能，两个向量和即为momentum方式的作用结果。

而靠左边的brown vector是动能，可以看出它那条blue vector是平行的，但它预测了下一阶段的梯度是red vector，因此向量和就是green vector，即NAG方式的作用结果。

momentum项和nesterov项都是为了使梯度更新更加灵活，对不同情况有针对性。但是，人工设置一些学习率总还是有些生硬，接下来介绍几种自适应学习率的方法

四、学习率退火

训练深度网络的时候，可以让学习率随着时间退火。因为如果学习率很高，系统的动能就过大，参数向量就会无规律地变动，无法稳定到损失函数更深更窄的部分去。对学习率衰减的时机把握很有技巧：如果慢慢减小，可能在很长时间内只能浪费计算资源然后看着它混沌地跳动，实际进展很少；但如果快速地减少，系统可能过快地失去能量，不能到达原本可以到达的最好位置。通常，实现学习率退火有三种方式：

- 1）随步数衰减：每进行几个周期就根据一些因素降低学习率。通常是每过5个周期就将学习率减少一半，或者每20个周期减少到之前的十分之一。这些数值的设定是严重依赖具体问题和模型的选择的。在实践中可能看见这么一种经验做法：使用一个固定的学习率来进行训练的同时观察验证集错误率，每当验证集错误率停止下降，就乘以一个常数（比如0.5）来降低学习率。
- 2）指数衰减。数学公式是$\alpha = \alpha_0 e^{-kt}$，其中$\alpha_0, k$是超参数，$t$是迭代次数（也可以使用周期作为单位）。
- 3）$1/t$衰减的数学公式是$\alpha = \alpha_0/(1 + kt)$，其中$\alpha_0, k$是超参数，t是迭代次数。

在实践中，我们发现随步数衰减的随机失活（dropout）更受欢迎，因为它使用的超参数（衰减系数和以周期为时间单位的步数）比k更有解释性。但如果你有足够的计算资源，可以让衰减更加缓慢一些，让训练时间更长些。

五、自适应学习率方法

5.1 Adagrad [Duchi et al., 2011]

之前的方法中所有参数在更新时均使用同一个Learning rate。而Learning rate调整是一个非常耗费计算资源的过程，所以如果能够自适应地对参数进行调整的话，就大大降低了成本。在Adagrad的每一个参数的每一次更新中都使用不同的learning rate。这样的话，令第t步更新时对第i个参数的梯度为

$$g_{t,i} = \nabla_\theta J\left(\theta_j\right)$$

参数的更新的一般形式为：

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

如上所述，Adagrad的差异之处正是在于learning rate不同于其他，将learning rate改为如下：

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{\sum_{i=0}^{t}\left(g^i\right)^2 + \epsilon}} \cdot g_{t,i}$$

实质上是对学习率形成了一个约束项regularizer：$\frac{1}{\sqrt{\sum_{i=0}^{t}\left(g^i\right)^2+\epsilon}}$，$\sum_{i=0}^{t}\left(g^i\right)^2$是对直至t次迭代的梯度平方和的累加和，$\epsilon$是一个防止分母为0的很小的平滑项。不用平方根操作，算法性能会变差很多

我们可以将到累加的梯度平方和放在一个对角矩阵中$G_t \in \mathbb{R}^{d\times d}$中，其中每个对角元素$(i, i)$是参数$\theta_i$到时刻$t$为止所有时刻梯度的平方之和。由于$G_t$的对角包含着所有参数过去时刻的平方之

和，我们可以通过在G_t和g_t执行element-wise matrix vector mulitiplication来向量化我们的操作：

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

- 优点：Adagrad让学习速率自适应于参数，在前期g_t较小的时候，regularizer较大，能够放大梯度；后期g_t较大的时候，regularizer较小，能够约束梯度；因为这一点，它非常适合处理稀疏数据。Dean等人发现Adagrad大大地提高了SGD的鲁棒性并在谷歌的大规模神经网络训练中采用了它进行参数更新，其中包含了在Youtube视频中进行猫脸识别。此外，由于低频词（参数）需要更大幅度的更新，Pennington等人在GloVe word embeddings的训练中也采用了Adagrad。

- 缺点：由公式可以看出，仍依赖于人工设置一个全局学习率；η设置过大的话，会使得regularizer过于敏感，对梯度的调节太大；中后期，分母上梯度平方的累加将会越来越大，使得梯度为0，训练提前结束。

5.2 RMSprop [Hinton]

RMSprop是一个没有公开发表的适应性学习率方法，它是Hinton在他的课上提出的一种自适应学习速率方法。有趣的是，每个使用这个方法的人在他们的论文中都引用自Geoff Hinton的Coursera课程的第六课的第29页PPT。它用了一种很简单的方式修改了Adagrad方法，让它不过于激进而过早停止学习。具体说来就是，它使用了一个梯度平方的滑动平均，仍然是基于梯度的大小来对每个权重的学习率进行修改，效果不错。但是和Adagrad不同的是，其更新不会让学习率单调变小。

下图展示了RMSprop的计算过程，其中α是一个超参数，常用的值是[0.9,0.99,0.999]：

$$w^1 \leftarrow w^0 - \frac{\eta}{\sigma^0} g^0 \qquad \sigma^0 = g^0$$

$$w^2 \leftarrow w^1 - \frac{\eta}{\sigma^1} g^1 \qquad \sigma^1 = \sqrt{\alpha(\sigma^0)^2 + (1-\alpha)(g^1)^2}$$

$$w^3 \leftarrow w^2 - \frac{\eta}{\sigma^2} g^2 \qquad \sigma^2 = \sqrt{\alpha(\sigma^1)^2 + (1-\alpha)(g^2)^2}$$

$$\vdots$$

$$w^{t+1} \leftarrow w^t - \frac{\eta}{\sigma^t} g^t \qquad \sigma^t = \sqrt{\alpha(\sigma^{t-1})^2 + (1-\alpha)(g^t)^2}$$

Root Mean Square of the gradients with previous gradients being decayed

5.3 Adadelta [Zeiler, 2012]

Adadelta是Adagrad的一种扩展，以缓解Adagrad学习速率单调递减问题的算法。Adadelta不是对过去所有时刻的梯度平方进行累加，而是将累加时刻限制在窗口大小为的w区间。

但梯度累加没有采用简单的存储前w个时刻的梯度平方，而是递归地定义为过去所有时刻梯度平方的decaying average$E[g^2]_t$。t时刻的running average仅仅依赖于之前average和当前的梯度：

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

类似momentum term，我们将γ取值在0.9附近。简介起见，我们从参数更新向量$\Delta\theta_t$角度重写普通SGD的参数更新：

$$\Delta\theta_t = -\eta \cdot g_{t,i}$$

$$\theta_{t+1} = \theta_t + \Delta\theta_t$$

Adagrad中我们推倒的参数更新向量现在就以下述形式出现：

$$\Delta\theta_t = -\frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t$$

现在我们简单地将对角矩阵替换为过去时刻梯度平方的decaying average $E[g^2]_t$：

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

由于分母是root mean squared (RMS) error criterion of the gradient，则上面公式可以替换为：

$$\Delta\theta_t = -\frac{\eta}{RMS[g]_t}$$

作者发现（和SGD，Momentum或者Adagrad一样）上述更新中的单元不匹配，即只有部分参数进行更新，也就是参数和更新应该有着相同的hypothetical units。为了实现这个目的，他们首先定义了另外一个exponentially decaying average，这一次对更新参数的平方进行操作，而不只是对梯度的平方进行操作：

$$E[\Delta\theta^2]_t = \gamma \cdot E[\Delta\theta^2]_t + (1-\gamma)\Delta\theta^2$$

参数更新中的root mean squared error则为：

$$RMS[\Delta\theta]_t = \sqrt{E[\Delta\theta^2]_t + \epsilon}$$

将以前的更新规则中的学习速率替换为参数更新的RMS，则得到Adadelta更新规则:

$$\Delta\theta_t = -\frac{RMS[\Delta\theta]_t}{RMS[g]_t} \cdot g_t$$

$$\theta_{t+1} = \theta_t + \Delta\theta$$

由于Adadelta更新规则中没有了学习速率这一项，我们甚至都不用对学习速率进行设置。
训练初中期，加速效果不错，很快
训练后期，反复在局部最小值附近抖动

5.4 Adam [Kingma and Ba, 2014]

Adaptive Moment Estimation (Adam)是另外一种对每个参数进行自适应学习速率计算的方法，除了像Adadelta和RMSprop一样保存去过梯度平方和的exponentially decaying average外，Adam还保存类似momentum一样过去梯度的exponentially decaying average。它看起来像是RMSProp的动量版。

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

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

m_t和v_t分别是分别是梯度的一阶矩（均值）和二阶距（偏方差）的估计，由于m_t和v_t由全零的向量来初始化，Adam的作者观察到他们会被偏向0，特别是在initial time steps或decay rates很小的时候（即β_1和β_2都接近于1）,于是他们通过计算bias-corrected一阶矩和二阶矩的估计低消

掉偏差。

$$\hat{m} = \frac{m}{1-\beta_1^t}$$

$$\hat{v} = \frac{v}{1-\beta_2^t}$$

然后使用上述项和Adadelta和RMSprop一样进行参数更新，可以得到Adam的更新规则：

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

Adam的完整更新过程如下图所示，其中它推荐默认设置
$\alpha = 0.001, \beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8}$，在实际操作中，推荐将Adam作为默认的算法，一般而言跑起来比RMSProp要好一些。但也可以试试SGD+Nesterov动量。

Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t.

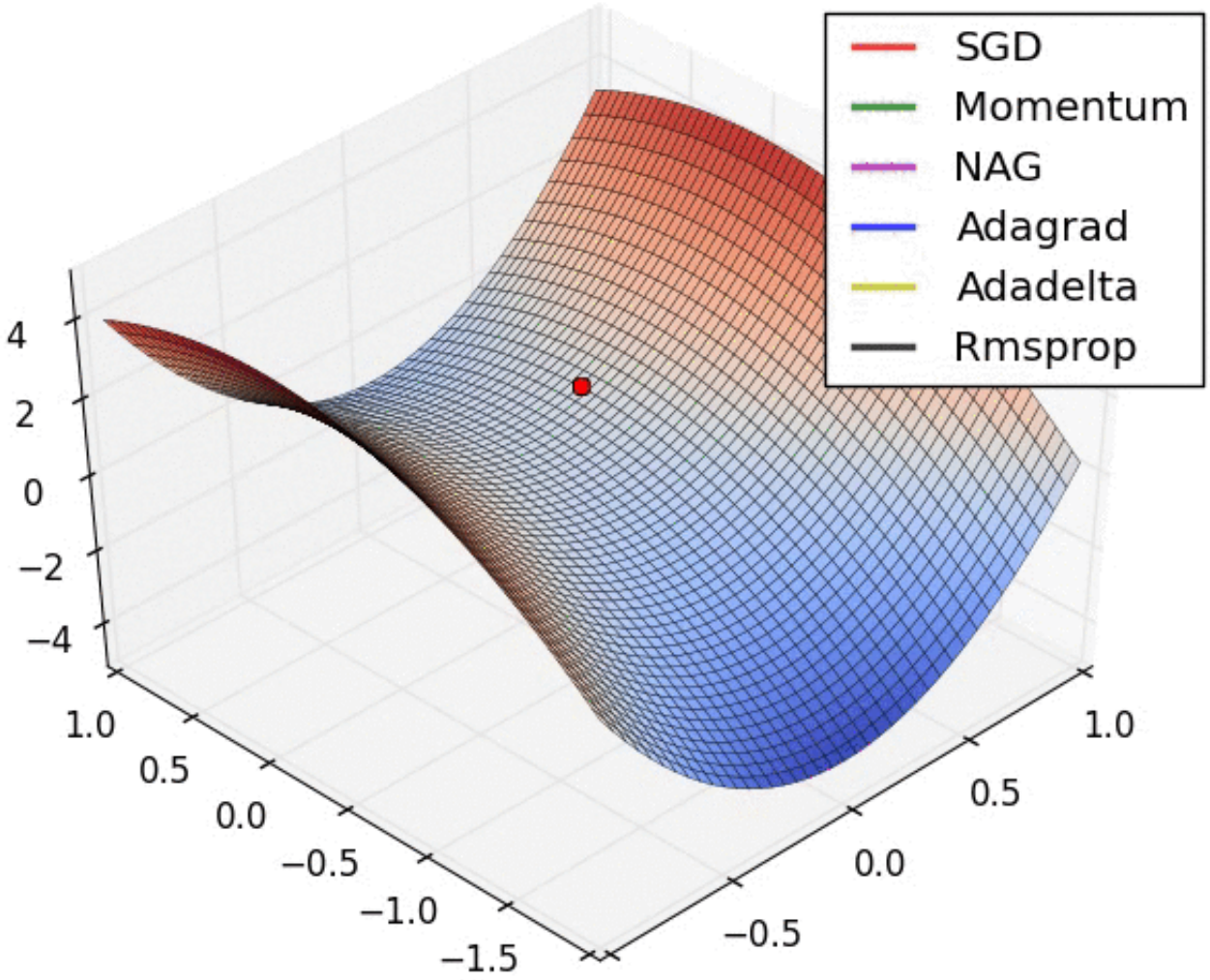
Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
$m_0 \leftarrow 0$ (Initialize 1st moment vector) → for momentum
$v_0 \leftarrow 0$ (Initialize 2nd moment vector) → for RMSprop
$t \leftarrow 0$ (Initialize timestep)
while θ_t not converged **do**
$t \leftarrow t + 1$
$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
$\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
$\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
end while
return θ_t (Resulting parameters)

六、算法可视化

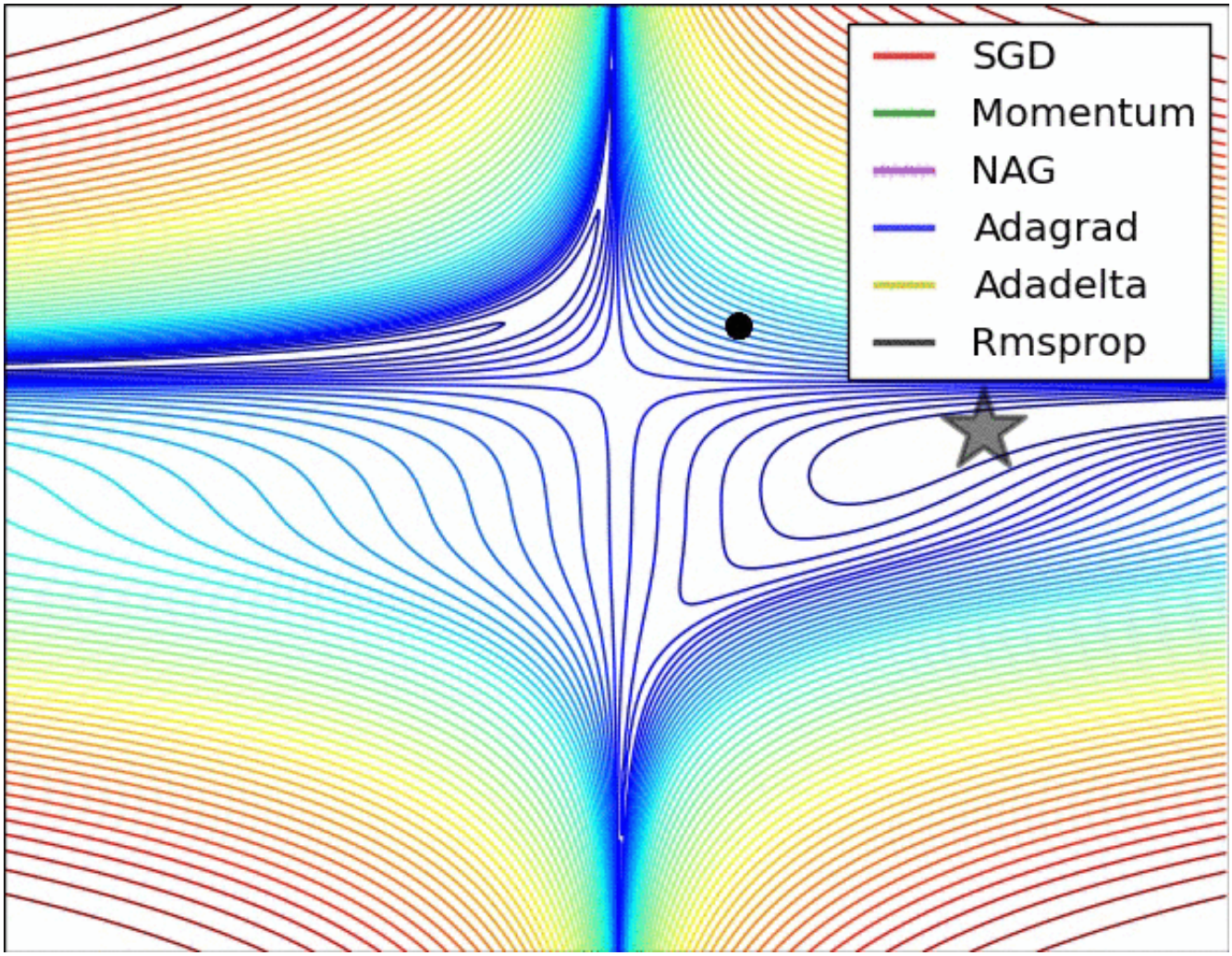
下面两幅动画让我们直观感受一些优化算法的优化过程。

在第一幅动图中，我们看到他们随着时间推移在损失表面的轮廓（contours of a loss surface）的移动。注意到Adagrad、Adadelta和RMSprop几乎立刻转向正确的方向并快速收敛，但是Momentum和NAG被引导偏离了轨道。这让我们感觉就像看滚下山的小球。然而，由于NAG拥

有通过远眺所提高的警惕，它能够修正他的轨迹并转向极小值。



第二幅动图中为各种算法在saddle point（鞍点）上的表现。所谓saddle point也就是某个维度是positive slope，其他维度为negative lope。前文中我们已经提及了它给SGD所带来的困难。注意到SGD、Momentum和NAG很难打破对称，虽然后两者最后还是逃离了saddle point。然而Adagrad, RMSprop, and Adadelta迅速地沿着negative slope下滑。



七、二阶方法

在深度网络背景下，第二类常用的最优化方法是基于牛顿法的，其迭代如下：

$$x \leftarrow x - [Hf(x)]^{-1}\nabla f(x)$$

这里$Hf(x)$是Hessian矩阵，它是函数的二阶偏导数的平方矩阵。$\nabla f(x)$是梯度向量，这和梯度下降中一样。直观理解上，Hessian矩阵描述了损失函数的局部曲率，从而使得可以进行更高效的参数更新。具体来说，就是乘以Hessian转置矩阵可以让最优化过程在曲率小的时候大步前进，在曲率大的时候小步前进。需要重点注意的是，在这个公式中是没有学习率这个超参数的，这相较于一阶方法是一个巨大的优势。

然而上述更新方法很难运用到实际的深度学习应用中去，这是因为计算（以及求逆）Hessian矩阵操作非常耗费时间和空间。举例来说，假设一个有一百万个参数的神经网络，其Hessian矩阵大小就是[1,000,000 x 1,000,000]，将占用将近3,725GB的内存。这样，各种各样的拟-牛顿法就被发明出来用于近似转置Hessian矩阵。在这些方法中最流行的是L-BFGS，L-BFGS使用随时间的梯度中的信息来隐式地近似（也就是说整个矩阵是从来没有被计算的）。

然而，即使解决了存储空间的问题，L-BFGS应用的一个巨大劣势是需要对整个训练集进行计

算，而整个训练集一般包含几百万的样本。和小批量随机梯度下降（mini-batch SGD）不同，让L-BFGS在小批量上运行起来是很需要技巧，同时也是研究热点。

实践时在深度学习和卷积神经网络中，使用L-BFGS之类的二阶方法并不常见。相反，基于（Nesterov的）动量更新的各种随机梯度下降方法更加常用，因为它们更加简单且容易扩展。

参考资料

[1] Kiefer, J., Wolfowitz, J., et al. (1952). Stochastic estimation of the maximum of a regression function. The Annals of Mathematical Statistics, 23(3):462–466.
[2] Nesterov, Y. (1983). A method of solving a convex programming problem with convergence rate o (1/k2). In Soviet Mathematics Doklady, volume 27, pages 372–376.
[3] Duchi, J., Hazan, E., and Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. The Journal of Machine Learning Research, 12:2121–2159.
[4] Hinton. Neural Networks for Machine Learning
[5] Zeiler, M. D. (2012). Adadelta: An adaptive learning rate method.
[6] Kingma, D. and Ba, J. (2014). Adam: A method for stochastic optimization.
[7] CS231n Convolutional Neural Networks for Visual Recognition.