Microsoft

Reactor

一起学人工智能系列
- 深度学习

2021-11-16

Map


Stockholm
COMING SOON
London
Redmond
Toronto
New York
San Francisco
Tel Aviv
Shanghai
Abu Dhabi
Bangalore
Sao Paulo
Sydney

个人介绍

Kinfey Lo – (卢建晖)

Microsoft Cloud Advocate

前微软MVP、Xamarin MVP和微软RD，拥有超过10年的云原生、人工智能和移动应用经验，为教育、金融和医疗提供应用解决方案。 Microsoft Iginte，Teched 会议讲师，Microsoft AI 黑客马拉松教练，目前在微软，为技术人员和不同行业宣讲技术和相关应用场景。

爱编程(Python， C#，TypeScript，Swift，Rust，Go ）

专注于人工智能，云原生，跨平台移动开发

Github : https://github.com/kinfey

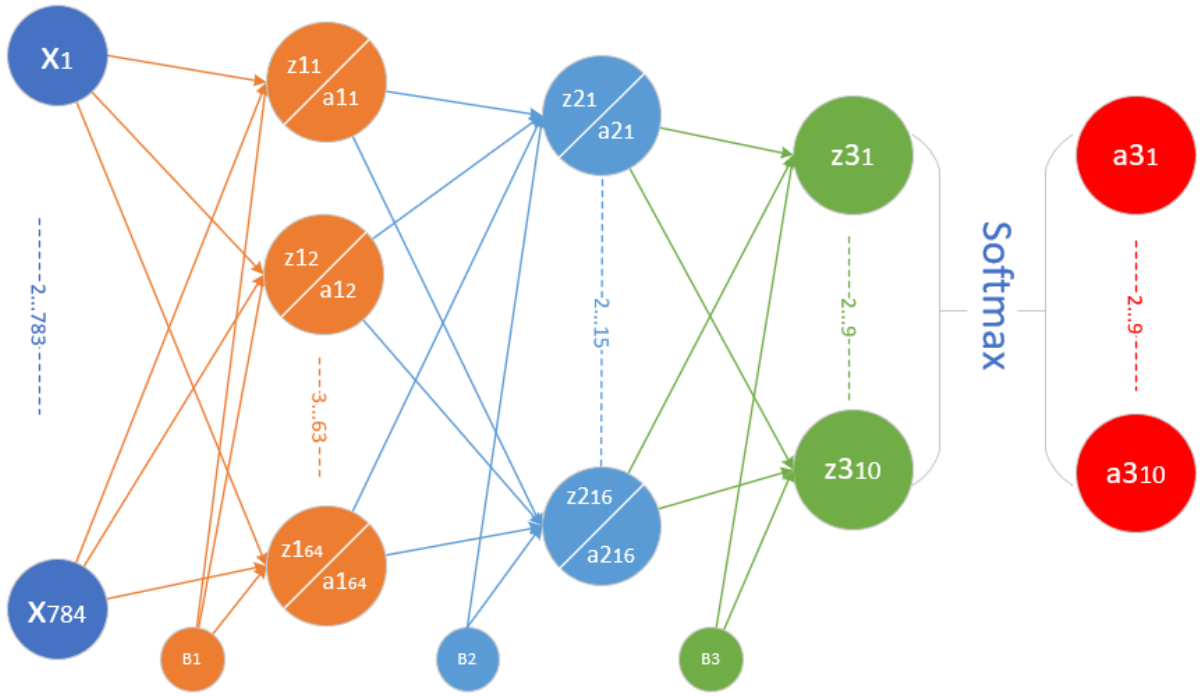
Email : kinfeylo@microsoft.com **Blog :** https://blog.csdn.net/kinfey

Twitter : @Ljh8304

深度神经网络

深度神经网络

深度神经网络是一种具备至少一个隐层的神经网络。与浅层神经网络类似，深度神经网络也能够为复杂非线性系统提供建模，但多出的层次为模型提供了更高的抽象层次，因而提高了模型的能力。深度神经网络通常都是前馈神经网络，但也有语言建模等方面的研究将其拓展到循环神经网络



多入多出的三层神经网络

多变量非线性多分类

提出问题

手写识别是人工智能的重要课题之一。MNIST数字手写体识别图片集，大家一定不陌生，下面就是一些样本。

图12-1 MNIST数据集样本示例

由于这是从欧美国家和地区收集的数据，从图中可以看出有几点和中国人的手写习惯不一样：

- 数字2，下面多一个圈
- 数字4，很多横线不出头
- 数字6，上面是直的
- 数字7，中间有个横杠

不过这些细节不影响咱们学习课程，正好还可以验证一下中国人的手写习惯是否能够被正确识别。

由于不是让我们识别26个英文字母或者3500多个常用汉字，所以问题还算是比较简单，不需要图像处理知识，也暂时不需要卷积神经网络的参与。咱们可以试试用一个三层的神经网络解决此问题，把每个图片的像素都当作一个向量来看，而不是作为点阵。

图片数据归一化

数据归一化，是针对数据的特征值做的处理，也就是针对样本数据的列做的处理。

三层神经网络的实现

输入层

28x28=784个特征值：

$$X = \begin{pmatrix} x_1 & x_2 & \dots & x_{784} \end{pmatrix}$$

隐层1

- 权重矩阵w1形状为784x64

$$W1 = \begin{pmatrix} w^1_{1,1} & w^1_{1,2} & \dots & w^1_{1,64} \\ \dots & \dots & & \dots \\ w^1_{784,1} & w^1_{784,2} & \dots & w^1_{784,64} \end{pmatrix}$$

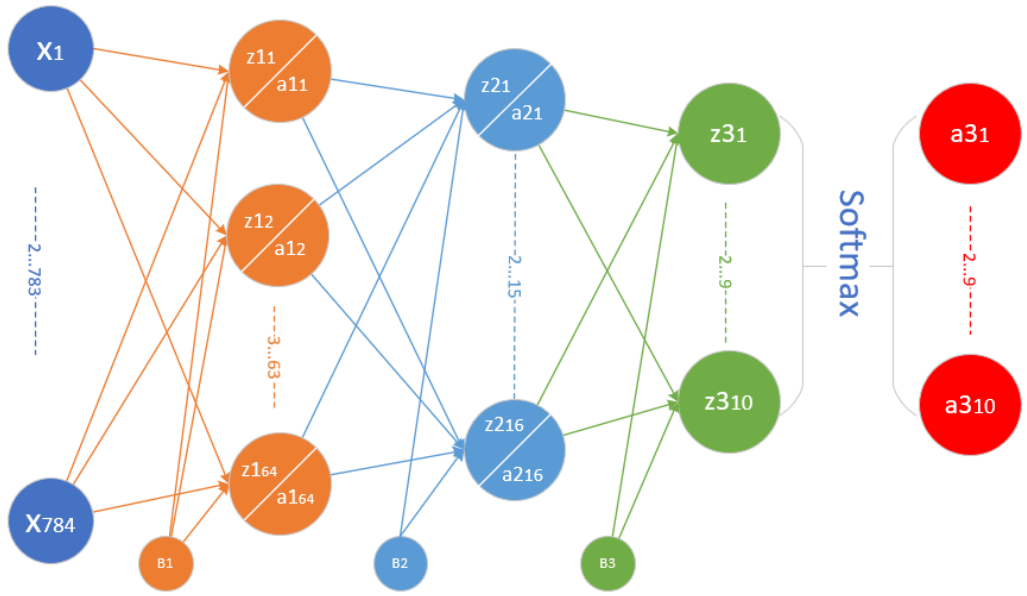
- 偏移矩阵b1的形状为1x64

$$B1 = \begin{pmatrix} b^1_1 & b^1_2 & \dots & b^1_{64} \end{pmatrix}$$

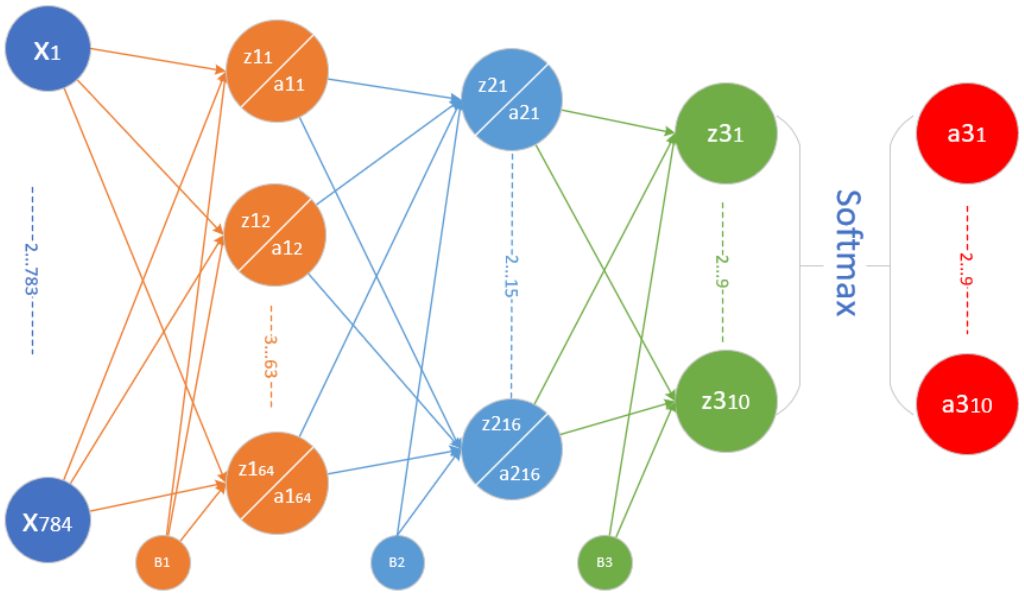
- 隐层1由64个神经元构成，其结果为1x64的矩阵

$$Z1 = \begin{pmatrix} z^1_1 & z^1_2 & \dots & z^1_{64} \end{pmatrix}$$

$$A1 = \begin{pmatrix} a^1_1 & a^1_2 & \dots & a^1_{64} \end{pmatrix}$$



三层神经网络的实现



隐层2

- 权重矩阵w2形状为64x16

$$W2 = \begin{pmatrix} w^2_{1,1} & w^2_{1,2} & \dots & w^2_{1,16} \\ \dots & \dots & & \dots \\ w^2_{64,1} & w^2_{64,2} & \dots & w^2_{64,16} \end{pmatrix}$$

- 偏移矩阵b2的形状是1x16

$$B2 = \begin{pmatrix} b^2_1 & b^2_2 & \dots & b^2_{16} \end{pmatrix}$$

- 隐层2由16个神经元构成

$$Z2 = \begin{pmatrix} z^2_1 & z^2_2 & \dots & z^2_{16} \end{pmatrix}$$

$$A2 = \begin{pmatrix} a^2_1 & a^2_2 & \dots & a^2_{16} \end{pmatrix}$$

输出层

- 权重矩阵w3的形状为16x10

$$W3 = \begin{pmatrix} w^3_{1,1} & w^3_{1,2} & \dots & w^3_{1,10} \\ \dots & \dots & & \dots \\ w^3_{16,1} & w^3_{16,2} & \dots & w^3_{16,10} \end{pmatrix}$$

- 输出层的偏移矩阵b3的形状是1x10

$$B3 = \begin{pmatrix} b^3_1 & b^3_2 & \dots & b^3_{10} \end{pmatrix}$$

- 输出层有10个神经元使用Softmax函数进行分类

三层神经网络的实现-前向计算

隐层1

$$Z1 = X \cdot W1 + B1 \tag{1}$$

$$A1 = Sigmoid(Z1) \tag{2}$$

隐层2

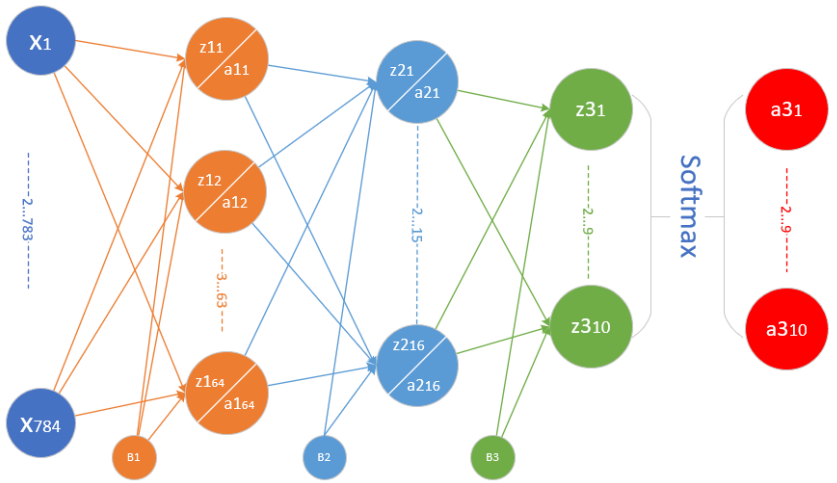
$$Z2 = A1 \cdot W2 + B2 \tag{3}$$

$$A2 = Tanh(Z2) \tag{4}$$

输出层

$$Z3 = A2 \cdot W3 + B3 \tag{5}$$

$$A3 = Softmax(Z3) \tag{6}$$



我们的约定是行为样本，列为一个样本的所有特征，这里是784个特征，因为图片高和宽是28x28，总共784个点，把每一个点的值做为特征向量。

两个隐层，分别定义64个神经元和16个神经元。第一个隐层用Sigmoid激活函数，第二个隐层用Tanh激活函数。

输出层10个神经元，再加上一个Softmax计算，最后有a1,a2,...a10十个输出，分别代表0-9的10个数字。

三层神经网络的实现-反向传播

输出层

$$dZ3 = A3 - Y \tag{7}$$

$$dW3 = A2^T \cdot dZ3 \tag{8}$$

$$dB3 = dZ3 \tag{9}$$

隐层2

$$dA2 = dZ3 \cdot W3^T \tag{10}$$

$$dZ2 = dA2 \odot (1 - A2 \odot A2) \tag{11}$$

$$dW2 = A1^T \cdot dZ2 \tag{12}$$

$$dB2 = dZ2 \tag{13}$$

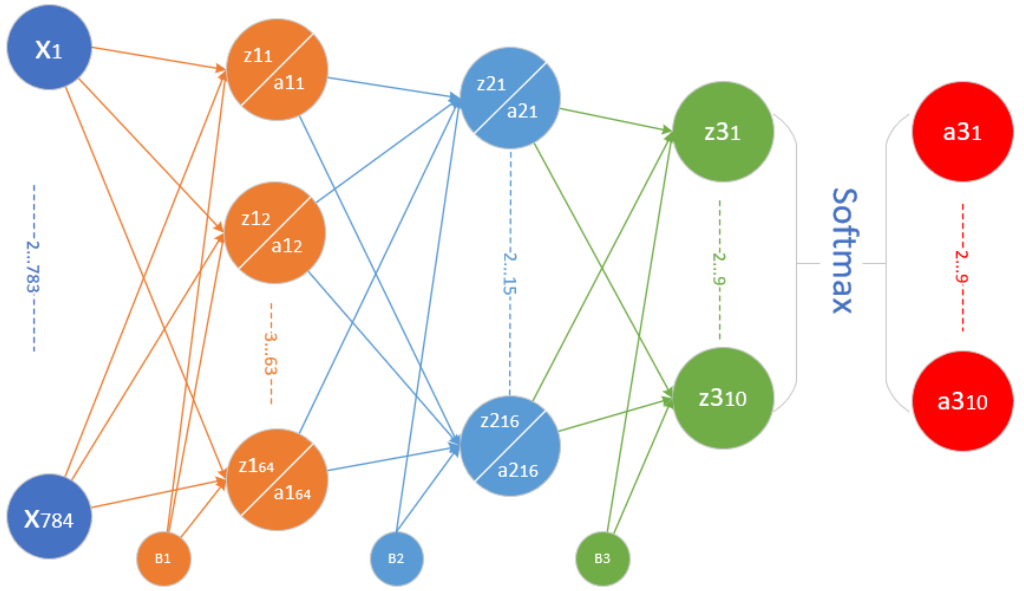
隐层1

$$dA1 = dZ2 \cdot W2^T \tag{14}$$

$$dZ1 = dA1 \odot A1 \odot (1 - A1) \tag{15}$$

$$dW1 = X^T \cdot dZ1 \tag{16}$$

$$dB1 = dZ1 \tag{17}$$



案例

用PyTorch的
一些基本算法实现

案例

训练时候的几个问题

随着网络的加深，训练变得越来越困难，时间越来越长，

原因可能是：

- **参数多**
- **数据量大**
- **梯度消失**
- **损失函数坡度平缓**

权重矩阵初始化

权重矩阵初始化

权重矩阵初始化是一个非常重要的环节，是训练神经网络的第一步，选择正确的初始化方法会带了事半功倍的效果。这就好比攀登喜马拉雅山，如果选择从南坡登山，会比从北坡容易很多。而初始化权重矩阵，相当于下山时选择不同的道路，在选择之前并不知道这条路的难易程度，只是知道它可以抵达山下。这种选择是随机的，即使你使用了正确的初始化算法，每次重新初始化时也会给训练结果带来很多影响。

比如第一次初始化时得到权重值为(0.12847，0.36453)，而第二次初始化得到(0.23334，0.24352)，经过试验，第一次初始化用了3000次迭代达到精度为96%的模型，第二次初始化只用了2000次迭代就达到了相同精度。这种情况在实践中是常见的

权重矩阵初始化 – 零初始化

即把所有层的`W`值的初始值都设置为0。

$$W = 0$$

但是对于多层网络来说，绝对不能用零初始化，否则权重值不能学习到合理的结果。看下面的零值初始化的权重矩阵值打印输出：

```
W1= [[-0.82452497 -0.82452497 -0.82452497]]
B1= [[-0.01143752 -0.01143752 -0.01143752]]
W2= [[-0.68583865]
 [-0.68583865]
 [-0.68583865]]
B2= [[0.68359678]]
```

可以看到`W1`、`B1`、`W2`内部3个单元的值都一样，这是因为初始值都是0，所以梯度均匀回传，导致所有`W`的值都同步更新，没有差别。这样的话，无论多少轮，最终的结果也不会正确。

权重矩阵初始化 –标准初始化

标准正态初始化方法保证激活函数的输入均值为0，方差为1。将W按如下公式进行初始化：

$$W \sim N[0,1]$$

其中的W为权重矩阵，N表示高斯分布，Gaussian Distribution，也叫做正态分布，Normal Distribution，所以有的地方也称这种初始化为Normal初始化。

一般会根据全连接层的输入和输出数量来决定初始化的细节：

$$W \sim N\left(0, \frac{1}{\sqrt{n_{in}}}\right)$$

$$W \sim U\left(-\frac{1}{\sqrt{n_{in}}}, \frac{1}{\sqrt{n_{in}}}\right)$$

权重矩阵初始化 –标准初始化

当目标问题较为简单时，网络深度不大，所以用标准初始化就可以了。但是当使用深度网络时，会遇到如图15-1所示的问题。

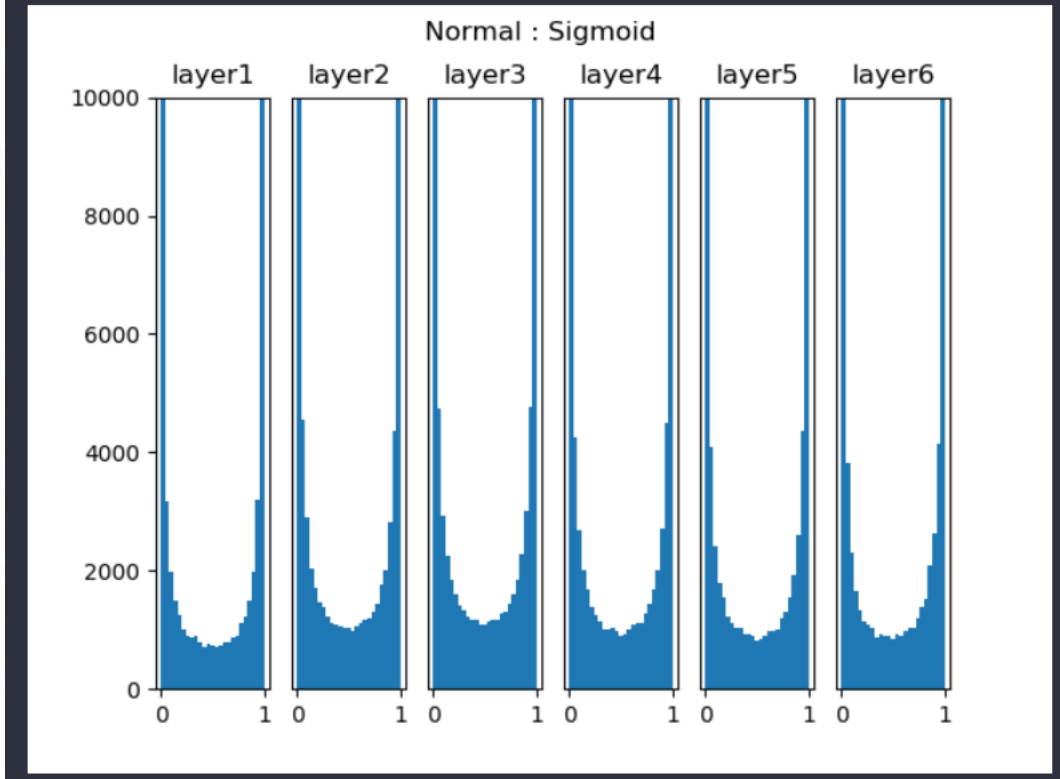


图15-1 标准初始化在Sigmoid激活函数上的表现

图15-1是一个6层的深度网络，使用全连接层+Sigmoid激活函数，图中表示的是各层激活函数的直方图。可以看到各层的激活值严重向两侧[0,1]靠近，从Sigmoid的函数曲线可以知道这些值的导数趋近于0，反向传播时的梯度逐步消失。处于中间地段的值比较少，对参数学习非常不利。

权重矩阵初始化 – Xavier初始化方法

条件：正向传播时，激活值的方差保持不变；反向传播时，关于状态值的梯度的方差保持不变。

$$W \sim N\left(0, \sqrt{\frac{2}{n_{in}+n_{out}}}\right)$$

$$W \sim U\left(-\sqrt{\frac{6}{n_{in}+n_{out}}}, \sqrt{\frac{6}{n_{in}+n_{out}}}\right)$$

其中的W为权重矩阵，N表示正态分布（Normal Distribution），U表示均匀分布（Uniform Distribution）。下同。

假设激活函数关于0对称，且主要针对于全连接神经网络。适用于tanh和softsign。

即权重矩阵参数应该满足在该区间内的均匀分布。其中的W是权重矩阵，U是Uniform分布，即均匀分布。

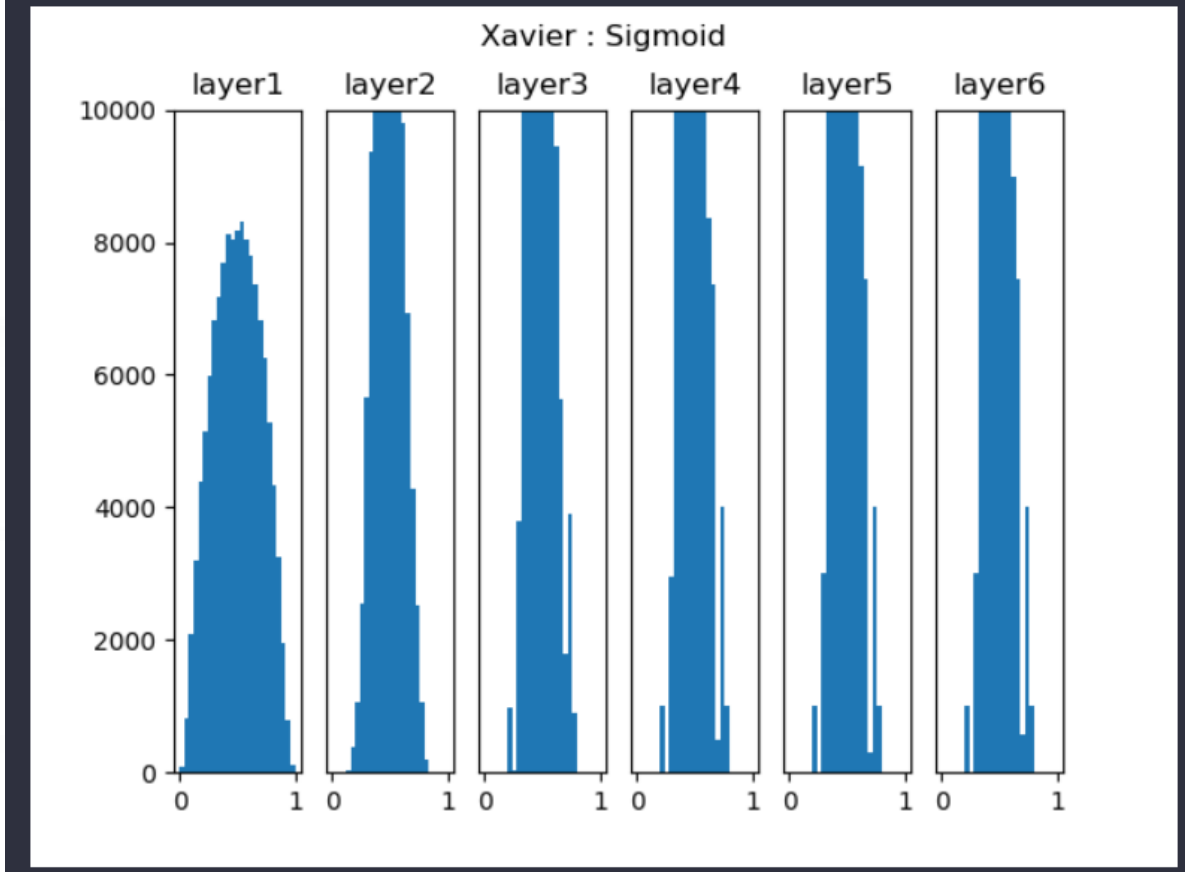
论文摘要：神经网络在2006年之前不能很理想地工作，很大原因在于权重矩阵初始化方法上。Sigmoid函数不太适合于深度学习，因为会导致梯度饱和。基于以上原因，我们提出了一种可以快速收敛的参数初始化方法。

Xavier初始化方法比直接用高斯分布进行初始化W的优势所在：

一般的神经网络在前向传播时神经元输出值的方差会不断增大，而使用Xavier等方法理论上可以保证每层神经元输入输出方差一致。

权重矩阵初始化 – Xavier初始化方法

图15-2是深度为6层的网络中的表现情况，可以看到，后面几层的激活函数输出值的分布仍然基本符合正态分布，利于神经网络的学习。



权重矩阵初始化 – Xavier初始化方法

但是，随着深度学习的发展，人们觉得Sigmoid的反向力度受限，又发明了ReLU激活函数。图15-3显示了Xavier初始化在ReLU激活函数上的表现。

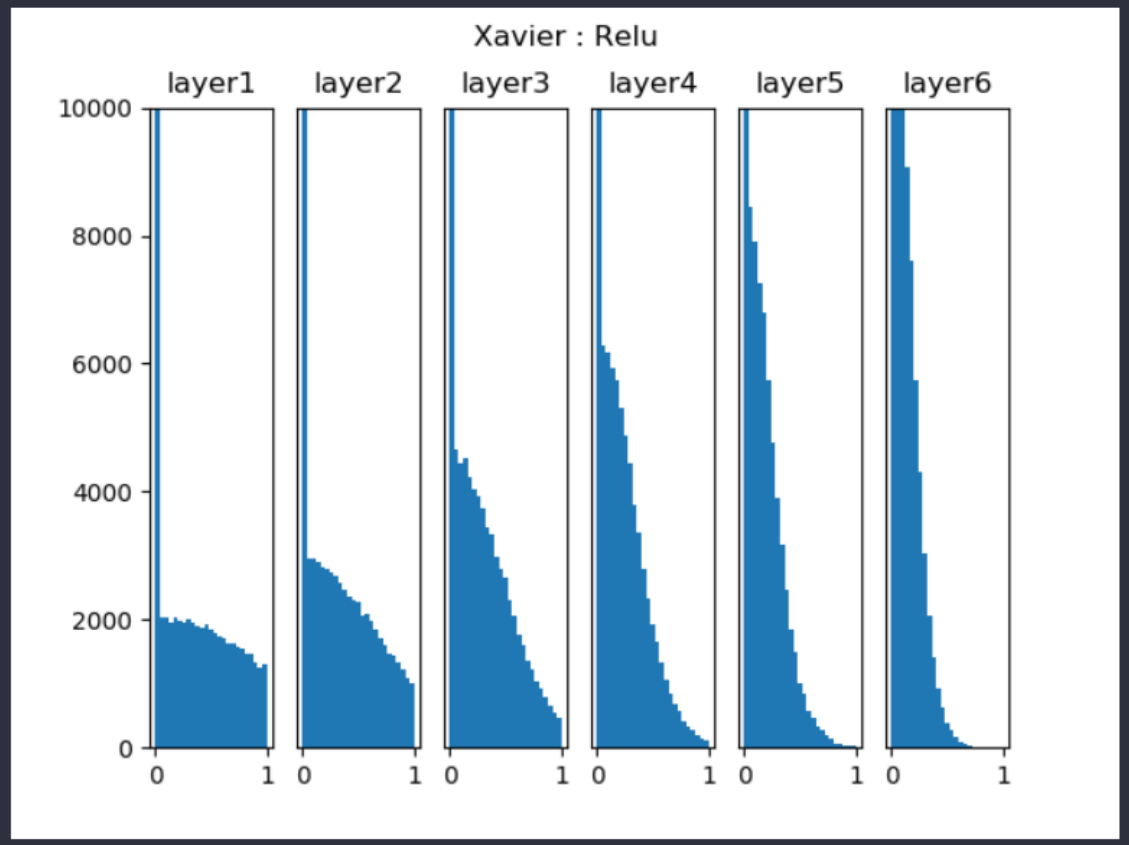


图15-3 Xavier初始化在ReLU激活函数上的表现

权重矩阵初始化 – MSRA初始化方法

MSRA初始化方法[2]，又叫做He方法，因为作者姓何。

条件：正向传播时，状态值的方差保持不变；反向传播时，关于激活值的梯度的方差保持不变。

网络初始化是一件很重要的事情。但是，传统的固定方差的高斯分布初始化，在网络变深的时候使得模型很难收敛。VGG团队是这样处理初始化的问题的：他们首先训练了一个8层的网络，然后用这个网络再去初始化更深的网络。

“Xavier”是一种相对不错的初始化方法，但是，Xavier推导的时候假设激活函数在零点附近是线性的，显然我们目前常用的ReLU和PReLU并不满足这一条件。所以MSRA初始化主要是想解决使用ReLU激活函数后，方差会发生变化，因此初始化权重的方法也应该变化。

只考虑输入个数时，MSRA初始化是一个均值为0，方差为2/n的高斯分布，适合于ReLU激活函数：

$$W \sim N\left(0, \sqrt{\frac{2}{n}}\right)$$

$$W \sim U\left(-\sqrt{\frac{6}{n_{in}}}, \sqrt{\frac{6}{n_{out}}}\right)$$

权重矩阵初始化 – MSRA初始化方法

图15-4中的激活值从0到1的分布，在各层都非常均匀，不会由于层的加深而梯度消失，所以，在使用ReLU时，推荐使用MSRA法初始化。

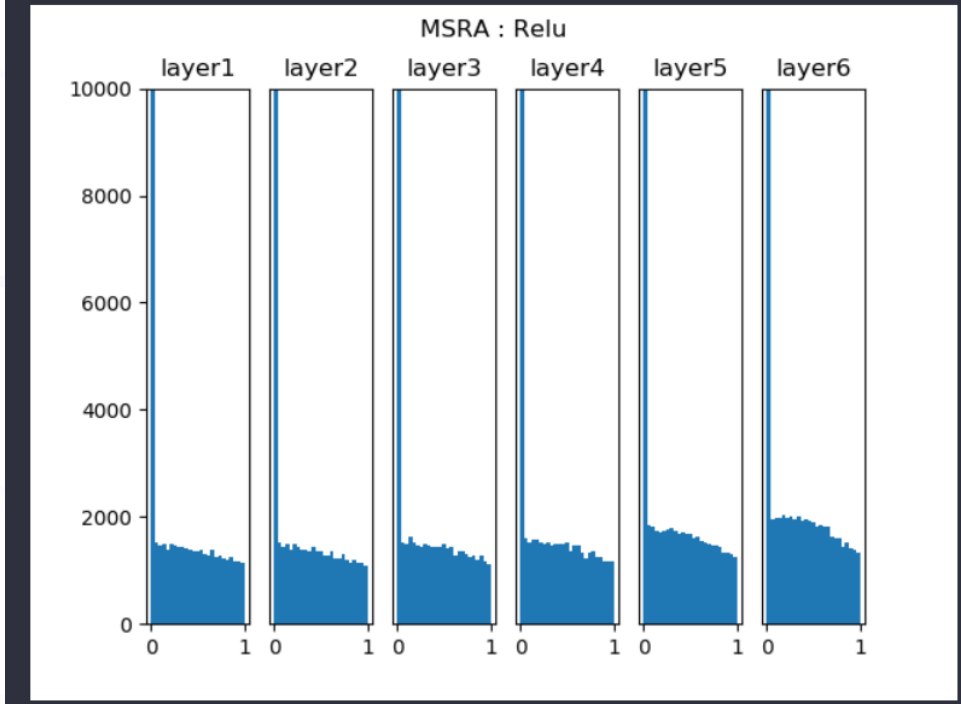


图15-4 MSRA初始化在ReLU激活函数上的表现

对于Leaky ReLU：

$$W \sim N\left[0, \sqrt{\frac{2}{(1+\alpha^2)\hat{n}_i}}\right] \hat{n}_i = h_i \cdot w_i \cdot d_i h_i：卷积核高度，w_i：卷积核宽度，d_i：卷积核个数$$

权重矩阵初始化比较

ID	网络深度	初始化方法	激活函数	说明
1	单层	零初始化	无	可以
2	双层	零初始化	Sigmoid	错误，不能进行正确的反向传播
3	双层	随机初始化	Sigmoid	可以
4	多层	随机初始化	Sigmoid	激活值分布成凹形，不利于反向传播
5	多层	Xavier初始化	Tanh	正确
6	多层	Xavier初始化	ReLU	激活值分布偏向0，不利于反向传播
7	多层	MSRA初始化	ReLU	正确

梯度下降优化算法

梯度下降优化算法 -随机梯度下降 SGD

先回忆一下随机梯度下降的基本算法，便于和后面的各种算法比较。图15-5中的梯度搜索轨迹为示意图。

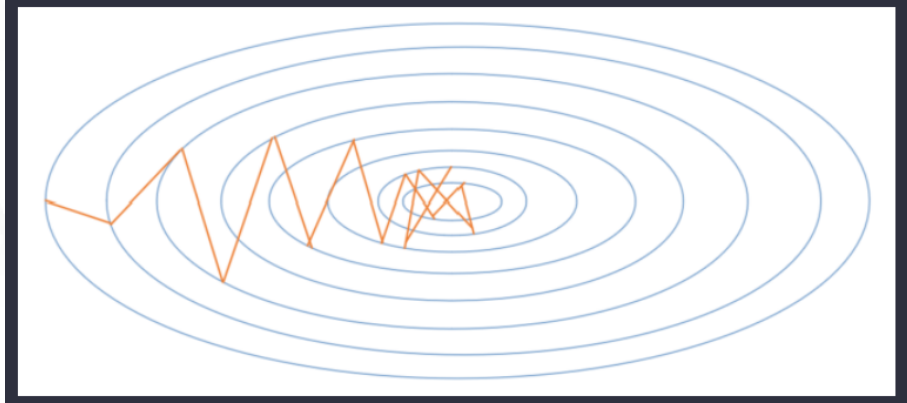


图15-5 随机梯度下降算法的梯度搜索轨迹示意图

输入和参数

- η - 全局学习率

算法

计算梯度：$g_t = \nabla_\theta J(\theta_{t-1})$

更新参数：$\theta_t = \theta_{t-1} - \eta \cdot g_t$

随机梯度下降算法，在当前点计算梯度，根据学习率前进到下一点。到中点附近时，由于样本误差或者学习率问题，会发生来回徘徊的现象，很可能会错过最优解。

收敛速度慢

梯度下降优化算法 - 动量算法 Momentum

SGD方法的一个缺点是其更新方向完全依赖于当前batch计算出的梯度，因而十分不稳定，因为数据有噪音。

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。Momentum算法会观察历史梯度，若当前梯度的方向与历史梯度一致（表明当前样本不太可能为异常点），则会增强这个方向的梯度。若当前梯度与历史梯度方向不一致，则梯度会衰减。

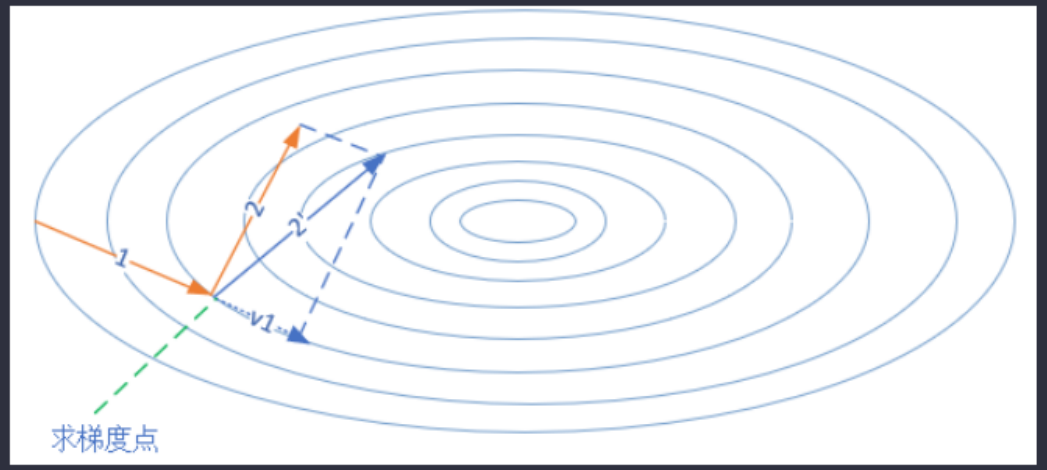


图15-6 动量算法的前进方向

图15-6中，第一次的梯度更新完毕后，会记录v_1的动量值。在“求梯度点”进行第二次梯度检查时，得到2号方向，与v_1的动量组合后，最终的更新为2'方向。这样一来，由于有v_1的存在，会迫使梯度更新方向具备“惯性”，从而可以减小随机样本造成的震荡。

梯度下降优化算法 - 动量算法 Momentum

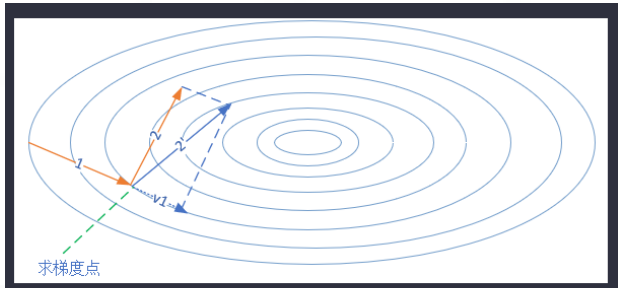


图15-6 动量算法的前进方向

图15-6中，第一次的梯度更新完毕后，会记录v_1的动量值。在"求梯度点"进行第二次梯度检查时，得到2号方向，与v_1的动量组合后，最终的更新为2'方向。这样一来，由于有v_1的存在，会迫使梯度更新方向具备"惯性"，从而可以减小随机样本造成的震荡。

输入和参数

- η - 全局学习率
- α - 动量参数，一般取值为0.5, 0.9, 0.99
- v_t - 当前时刻的动量，初值为0

算法

计算梯度：$g_t = \nabla_\theta J(\theta_{t-1})$

计算速度更新：$v_t = \alpha \cdot v_{t-1} + \eta \cdot g_t$ (公式1)

更新参数：$\theta_t = \theta_{t-1} - v_t$ (公式2)

但是在花书上的公式是这样的:

$v_t = \alpha \cdot v_{t-1} - \eta \cdot g_t$ (公式3)

$\theta_t = \theta_{t-1} + v_t$ (公式4)

根据算法公式(1)(2)，以W参数为例，有:

1. $v_0 = 0$
2. $dW_0 = \nabla J(w)$
3. $v_1 = \alpha v_0 + \eta \cdot dW_0 = \eta \cdot dW_0$
4. $W_1 = W_0 - v_1 = W_0 - \eta \cdot dW_0$
5. $dW_1 = \nabla J(w)$
6. $v_2 = \alpha v_1 + \eta dW_1$
7. $W_2 = W_1 - v_2 = W_1 - (\alpha v_1 + \eta dW_1) = W_1 - \alpha \cdot \eta \cdot dW_0 - \eta \cdot dW_1$
8. $dW_2 = \nabla J(w)$
9. $v_3 = \alpha v_2 + \eta dW_2$
10. $W_3 = W_2 - v_3 = W_2 - (\alpha v_2 + \eta dW_2) = W_2 - \alpha^2 \eta dW_0 - \alpha \eta dW_1 - \eta dW_2$

根据公式(3)(4)有:

1. $v_0 = 0$
2. $dW_0 = \nabla J(w)$
3. $v_1 = \alpha v_0 - \eta \cdot dW_0 = -\eta \cdot dW_0$
4. $W_1 = W_0 + v_1 = W_0 - \eta \cdot dW_0$
5. $dW_1 = \nabla J(w)$
6. $v_2 = \alpha v_1 - \eta dW_1$
7. $W_2 = W_1 + v_2 = W_1 + (\alpha v_1 - \eta dW_1) = W_1 - \alpha \cdot \eta \cdot dW_0 - \eta \cdot dW_1$
8. $dW_2 = \nabla J(w)$
9. $v_3 = \alpha v_2 - \eta dW_2$
10. $W_3 = W_2 + v_3 = W_2 + (\alpha v_2 - \eta dW_2) = W_2 - \alpha^2 \eta dW_0 - \alpha \eta dW_1 - \eta dW_2$

通过手工推导迭代，我们得到两个结论:

1. 可以看到两种方式的第9步结果是相同的，即公式(1)(2)等同于(3)(4)
2. 与普通SGD的算法$W_3 = W_2 - \eta dW_2$相比，动量法不但每次要减去当前梯度，还要减去历史梯度W_0, W_1乘以一个不断减弱的因子α，因为α小于1，所以α^2比α小，α^3比α^2小。这种方式的学名叫做指数加权平均。

梯度下降优化算法 - 梯度加速算法 NAG

在小球向下滚动的过程中，我们希望小球能够提前知道在哪些地方坡面会上升，这样在遇到上升坡面之前，小球就开始减速。这方法就是Nesterov Momentum，其在凸优化中有较强的理论保证收敛。并且，在实践中Nesterov Momentum也比单纯的Momentum 的效果好。

输入和参数

- η - 全局学习率
- α - 动量参数，缺省取值0.9
- v - 动量，初始值为0

算法

临时更新：$\hat{\theta} = \theta_{t-1} - \alpha \cdot v_{t-1}$

前向计算：$f(\hat{\theta})$

计算梯度：$g_t = \nabla_{\hat{\theta}} J(\hat{\theta})$

计算速度更新：$v_t = \alpha \cdot v_{t-1} + \eta \cdot g_t$

更新参数：$\theta_t = \theta_{t-1} - v_t$

其核心思想是：注意到 momentum 方法，如果只看 $\alpha \cdot v_{t-1}$ 项，那么当前的θ经过momentum的作用会变成 $\theta - \alpha \cdot v_{t-1}$。既然我们已经知道了下一步的走向，我们不妨先走一步，到达新的位置"展望"未来，然后在新位置上求梯度，而不是原始的位置。

所以，同Momentum相比，梯度不是根据当前位置θ计算出来的，而是在移动之后的位置$\theta - \alpha \cdot v_{t-1}$计算梯度。理由是，既然已经确定会移动$\theta - \alpha \cdot v_{t-1}$，那不如之前去看移动后的梯度。

图15-7是NAG的前进方向。

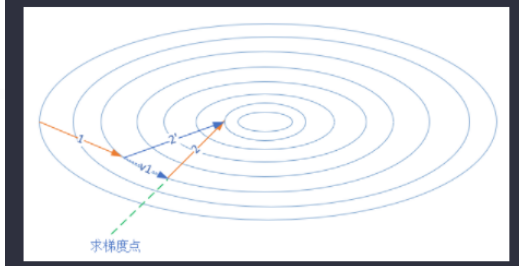


图15-7 梯度加速算法的前进方向

这个改进的目的就是为了提前看到前方的梯度。如果前方的梯度和当前梯度目标一致，那我直接大步迈过去；如果前方梯度同当前梯度不一致，那我就小心点更新。

自适应学习率算法

梯度下降优化算法 - AdaGrad

AdaGrad是一个基于梯度的优化算法，它的主要功能是：它对不同的参数调整学习率，具体而言，对低频出现的参数进行大的更新，对高频出现的参数进行小的更新。因此，他很适合于处理稀疏数据。

在这之前，我们对于所有的参数使用相同的学习率进行更新。但 Adagrad 则不然，对不同的训练迭代次数t，AdaGrad 对每个参数都有一个不同的学习率。这里开方、除法和乘法的运算都是按元素运算的。这些按元素运算使得目标函数自变量中每个元素都分别拥有自己的学习率。

梯度下降优化算法 - AdaGrad

输入和参数

- η - 全局学习率
- ϵ - 用于数值稳定的小常数，建议缺省值为 `1e-6`
- $r = 0$ 初始值

算法

对于AdaGrad来说，可以在开始时把学习率的值设置大一些，因为它会衰减得很快。

计算梯度：$g_t = \nabla_\theta J(\theta_{t-1})$

累计平方梯度：$r_t = r_{t-1} + g_t \odot g_t$

计算梯度更新：$\Delta\theta = \frac{\eta}{\epsilon + \sqrt{r_t}} \odot g_t$

更新参数：$\theta_t = \theta_{t-1} - \Delta\theta$

从AdaGrad算法中可以看出，随着算法不断迭代，r会越来越大，整体的学习率会越来越小。所以，一般来说AdaGrad算法一开始是激励收敛，到了后面就慢慢变成惩罚收敛，速度越来越慢。r值的变化如下：

1. $r_0 = 0$
2. $r_1 = g_1^2$
3. $r_2 = g_1^2 + g_2^2$
4. $r_3 = g_1^2 + g_2^2 + g_3^2$

在SGD中，随着梯度的增大，我们的学习步长应该是增大的。但是在AdaGrad中，随着梯度g的增大，r也在逐渐的增大，且在梯度更新时r在分母上，也就是整个学习率是减少的，这是为什么呢?

这是因为随着更新次数的增大，我们希望学习率越来越慢。因为我们认为在学习率的最初阶段，我们距离损失函数最优解还很远，随着更新次数的增加，越来越接近最优解，所以学习率也随之变慢。

但是当某个参数梯度较小时，累积和也会小，那么更新速度就大。

经验上已经发现，对于训练深度神经网络模型而言，从训练开始时积累梯度平方会导致有效学习率过早和过量的减小。AdaGrad在某些深度学习模型上效果不错，但不是全部。

梯度下降优化算法 - AdaDelta

Adaptive Learning Rate Method. [2]

AdaDelta法是AdaGrad 法的一个延伸，它旨在解决它学习率不断单调下降的问题。相比计算之前所有梯度值的平方和，AdaDelta法仅计算在一个大小为w的时间区间内梯度值的累积和。

但该方法并不会存储之前梯度的平方值，而是将梯度值累积值按如下的方式递归地定义：关于过去梯度值的衰减均值，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。

输入和参数

- ϵ - 用于数值稳定的小常数，建议缺省值为1e-5
- $\alpha \in [0, 1)$ - 衰减速率，建议0.9
- s - 累积变量，初始值0
- r - 累积变量变化量，初始为0

算法 ⚓

计算梯度：$g_t = \nabla_\theta J(\theta_{t-1})$

累积平方梯度：$s_t = \alpha \cdot s_{t-1} + (1 - \alpha) \cdot g_t \odot g_t$

计算梯度更新：$\Delta\theta = \sqrt{\frac{r_{t-1}+\epsilon}{s_t+\epsilon}} \odot g_t$

更新梯度：$\theta_t = \theta_{t-1} - \Delta\theta$

更新变化量：$r = \alpha \cdot r_{t-1} + (1 - \alpha) \cdot \Delta\theta \odot \Delta\theta$

初始学习率设置为0.1或者0.01，对于本算法来说都是一样的，这是因为算法中用r来代替学习率。

梯度下降优化算法 -均方根反向传播 RMSProp

RMSprop 是由 Geoff Hinton 在他 Coursera 课程中提出的一种适应性学习率方法，至今仍未被公开发表。RMSprop法要解决AdaGrad的学习率缩减问题。

输入和参数

- η - 全局学习率，建议设置为0.001
- ϵ - 用于数值稳定的小常数，建议缺省值为1e-8
- α - 衰减速率，建议缺省取值0.9
- r - 累积变量矩阵，与θ尺寸相同，初始化为0

算法

计算梯度：$g_t = \nabla_\theta J(\theta_{t-1})$

累计平方梯度：$r = \alpha \cdot r + (1-\alpha)(g_t \odot g_t)$

计算梯度更新：$\Delta\theta = \frac{\eta}{\sqrt{r+\epsilon}} \odot g_t$

更新参数：$\theta_t = \theta_{t-1} - \Delta\theta$

RMSprop也将学习率除以了一个指数衰减的衰减均值。为了进一步优化损失函数在更新中存在摆动幅度过大的问题，并且进一步加快函数的收敛速度，RMSProp算法对权重W和偏置b的梯度使用了微分平方加权平均数，这种做法有利于消除了摆动幅度大的方向，用来修正摆动幅度，使得各个维度的摆动幅度都较小。另一方面也使得网络函数收敛更快。

其中，r值的变化如下：

1. $r_0 = 0$
2. $r_1 = 0.1g_1^2$
3. $r_2 = 0.9r_1 + 0.1g_2^2 = 0.09g_1^2 + 0.1g_2^2$
4. $r_3 = 0.9r_2 + 0.1g_3^2 = 0.081g_1^2 + 0.09g_2^2 + 0.1g_3^2$

与AdaGrad相比，r_3要小很多，那么计算出来的学习率也不会衰减的太厉害。注意，在计算梯度更新时，分母开始时是个小于1的数，而且非常小，所以如果全局学习率设置过大的话，比如0.1，将会造成开始的步子迈得太大，而且久久不能收缩步伐，损失值也降不下来。

梯度下降优化算法 - Adam - Adaptive Moment Estimation

计算每个参数的自适应学习率，相当于RMSProp + Momentum的效果，Adam[4]算法在RMSProp算法基础上对小批量随机梯度也做了指数加权移动平均。和AdaGrad算法、RMSProp算法以及AdaDelta算法一样，目标函数自变量中每个元素都分别拥有自己的学习率。

输入和参数

- t - 当前迭代次数
- η - 全局学习率，建议缺省值为0.001
- ϵ - 用于数值稳定的小常数，建议缺省值为1e-8
- β_1, β_2 - 矩估计的指数衰减速率，$\in [0, 1)$，建议缺省值分别为0.9和0.999

算法

计算梯度：$g_t = \nabla_\theta J(\theta_{t-1})$

计数器加一：$t = t + 1$

更新有偏一阶矩估计：$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$

更新有偏二阶矩估计：$v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2)(g_t \odot g_t)$

修正一阶矩的偏差：$\hat{m}_t = m_t / (1 - \beta_1^t)$

修正二阶矩的偏差：$\hat{v}_t = v_t / (1 - \beta_2^t)$

计算梯度更新：$\Delta\theta = \eta \cdot \hat{m}_t / (\epsilon + \sqrt{\hat{v}_t})$

更新参数：$\theta_t = \theta_{t-1} - \Delta\theta$

自适应学习率算法

算法在等高线图上的效果比较- Adam - Adaptive Moment Estimation

为了简化起见，我们先用一个简单的二元二次函数来模拟损失函数的等高线图，测试一下我们在前面实现的各种优化器。但是以下测试结果只是一个示意性质的，可以理解为在绝对理想的条件下（样本无噪音，损失函数平滑等等）的各算法的表现。

$$z = \frac{x^2}{10} + y^2 \tag{1}$$

公式1是模拟均方差函数的形式，它的正向计算和反向计算的 Python 代码如下：

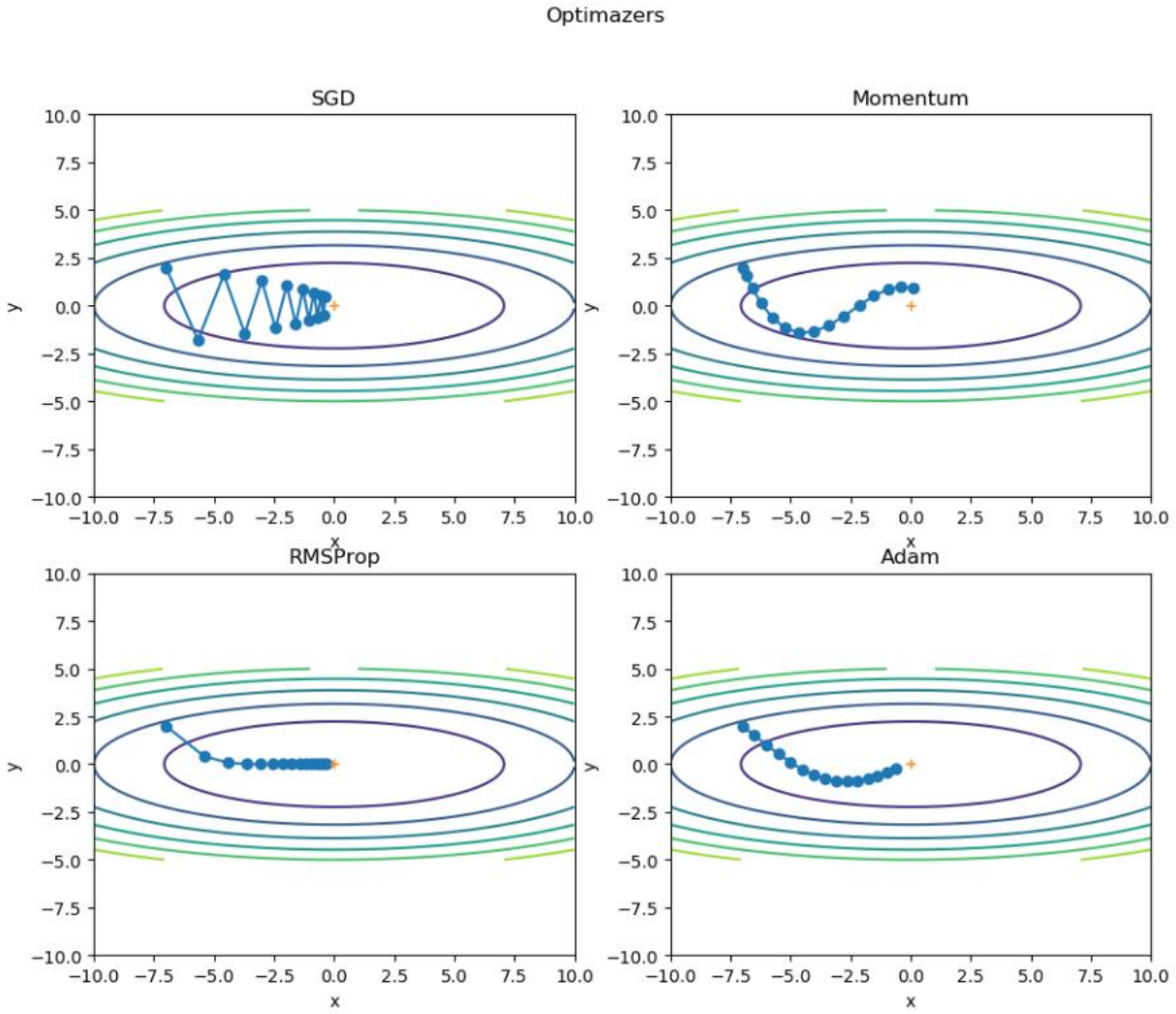
```python
def f(x, y):
    return x**2 / 10.0 + y**2

def derivative_f(x, y):
    return x / 5.0, 2.0*y
```

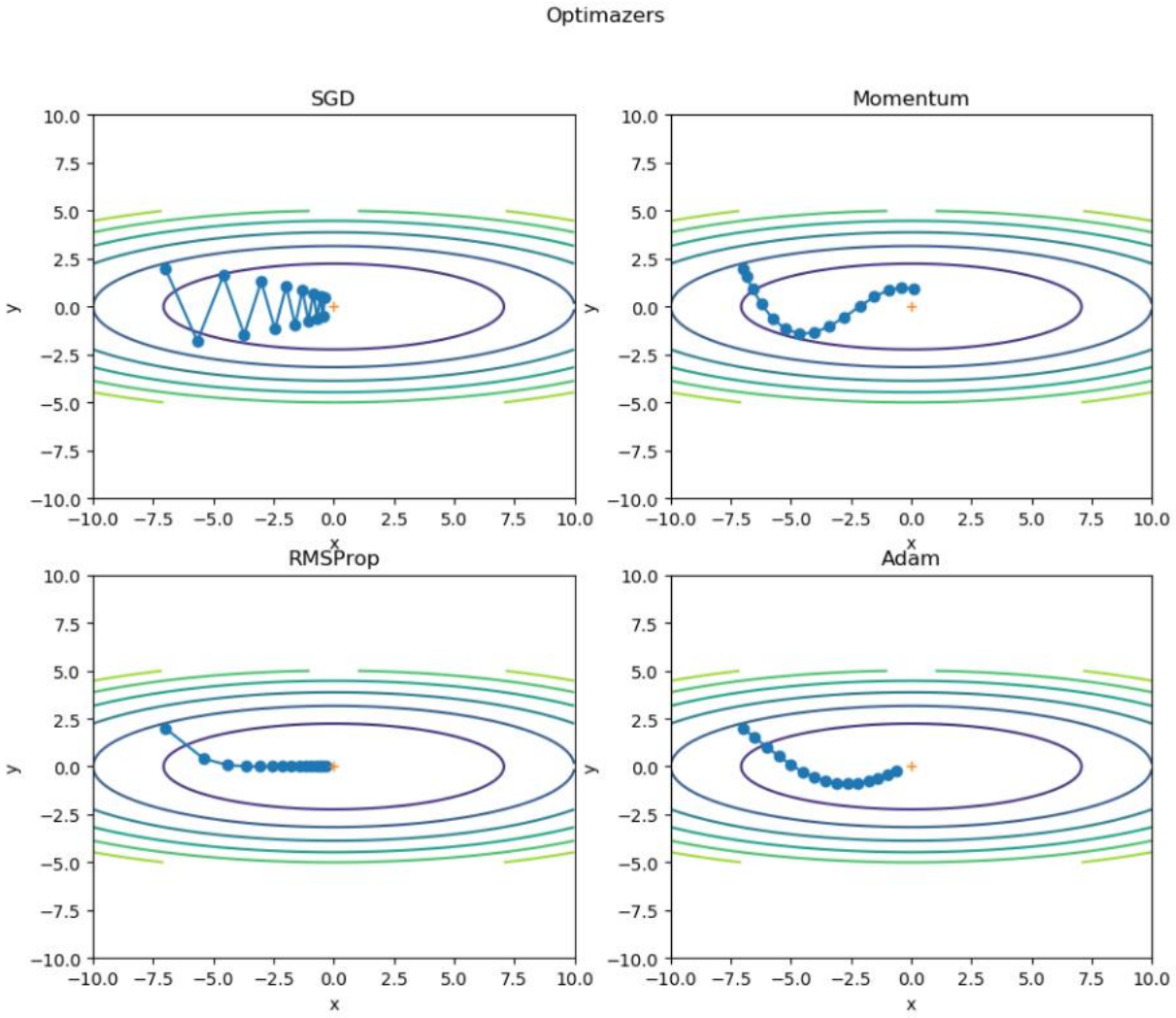
我们依次测试4种方法：

- 普通SGD, 学习率0.95
- 动量Momentum, 学习率0.1
- RMPSProp，学习率0.5
- Adam，学习率0.5

每种方法都迭代20次，记录下每次反向过程的(x,y)坐标点，绘制图15-8如下。



算法在等高线图上的效果比较- Adam - Adaptive Moment Estimation



•SGD算法，每次迭代完全受当前梯度的控制，所以会以折线方式前进。

•Momentum算法，学习率只有0.1，每次继承上一次的动量方向，所以会以比较平滑的曲线方式前进，不会出现突然的转向。

•RMSProp算法，有历史梯度值参与做指数加权平均，所以可以看到比较平缓，不会波动太大，都后期步长越来越短也是符合学习规律的。

•Adam算法，因为可以被理解为Momentum和RMSProp的组合，所以比Momentum要平缓一些，比RMSProp要平滑一些。

更多请等待下周的课程

批量归一化

QnA

Microsoft

Reactor

Thank You!