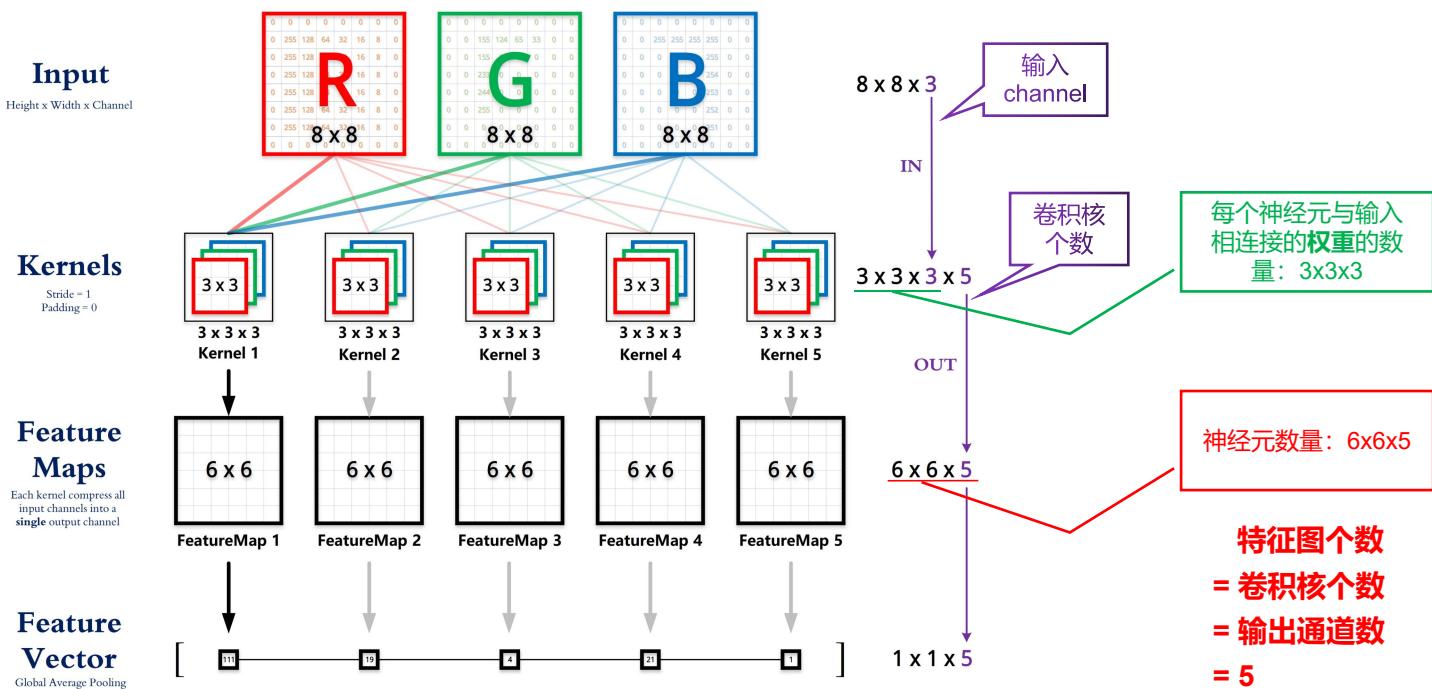


卷积层

参考文献：

1. CNN中卷积层的计算细节：<https://www.cnblogs.com/lfri/p/10491617.html>
2. 卷积网络中的通道(Channel)和特征图：<https://www.cnblogs.com/lfri/p/10491009.html>



标准卷积计算举例

以 AlexNet 模型的第一个卷积层为例，

- 输入图片的尺寸统一为 $227 \times 227 \times 3$ (高度 x 宽度 x 颜色通道数)
- 本层一共具有 96 个卷积核，
- 每个卷积核的尺寸都是 $11 \times 11 \times 3$ 。
- 已知 $\text{stride} = 4$, $\text{padding} = 0$,
- 假设 $\text{batch_size} = 256$,
- 则输出矩阵的高度/宽度为 $(227 - 11) / 4 + 1 = 55$

该卷积层一共有 55x55x96 个神经元。卷积核 (kernel) 的个数有 96 个。

- 即每一个卷积核对应有 55x55 个神经元。
- 即每 55x55 个神经元构成了一个卷积核对于 $227 \times 227 \times 3$ 输入的完整输出。
- 同一个卷积核的 55x55 个神经元中的每一个神经元只连接到输入 $227 \times 227 \times 3$ 中的局部内容，即 $11 \times 11 \times 3$ 这一局部。
- 即每一个神经元与输入连接的 weights 数量为 $11 \times 11 \times 3$ 。
- 该卷积层 bias 的数量等同于神经元的数量，即 $55 \times 55 \times 96$ 。
- 同一个卷积核的 55x55 神经元共享相同的 weights 和 bias，故实际上 96 个不同的卷积核只需要：
 - weights 的数量为 $3 \times (11 \times 11) \times 96$ 个
 - bias 的数量为 96 个
 - 总的参数量为 $(3 \times (11 \times 11) + 1) \times 96$ 个
- 可见，该层的参数数量只与卷积核的大小、数量（输出通道数）、输入通道数相关，具体而言就是：
 \rightarrow 卷积核大小 \times 输入通道数 \times 输出通道数

1 x 1 卷积计算举例

后期 GoogLeNet、ResNet 等经典模型中普遍使用一个像素大小的卷积核作为降低参数复杂度的手段。

从下面的运算可以看到，其实 1×1 卷积没有什么神秘的，其作用就是将输入矩阵的通道数量缩减后输出（512 降为 32），并保持它在宽度和高度维度上的尺寸（ 227×227 ）。

	Batch	Height	Width	In Depth	Out Depth
Input	256	x	227	x	227 x 512
Kernel			1	x	1 x 512 x 32
Output	256	x	227	x	227 x 32

总结：

全连接层计算举例

实际上，全连接层也可以被视为是一种极端情况的卷积层，其卷积核尺寸就是输入矩阵尺寸，因此输出矩阵的高度和宽度尺寸都是1。

	Batch	Height	Width	In Depth	Out Depth
Input	256	32	32	512	
Kernel		32	32	512	4096
Output	256	1	1		4096

- 总结下来，其实只需要认识到，虽然输入的每一张图像本身具有三个维度，但是对于卷积核来讲依然只是一个一维向量。卷积核做的，其实就是与感受野范围内的像素点进行点积（而不是矩阵乘法）。
- 卷积层内卷积核的个数不是计算出来的，而是人为设计出来的。

卷积网络中一个很重要的概念，**通道 (Channel)**，也有叫**特征图 (feature map)** 的。

卷积网络中主要有两个操作，一个是**卷积 (Convolution)**，一个是**池化 (Pooling)**。

卷积层则可以在通道与通道之间进行交互，之后在下一层生成新的通道，其中最显著的就是Incept-Net里大量用到的1x1卷积操作。基本上完全就是在通道与通道之间进行交互，而不关心同一通道中的交互。

一种卷积核得到一个通道，所以**特征图个数 = 输出通道数 = 卷积核个数**。

一个通道是对某个特征的检测，通道中某一处数值的强弱就是对当前特征强弱的反应。如一个蓝色通道中，如果是256级的话，那么一个像素如果是255的话那么就表示蓝色度很大。

于是卷积网络中的特征图，也能够很直接地理解为通道了。之后通过对一定范围的特征图进行卷积，可以将多个特征组合出来的模式抽取成一个特征，获得下一个特征图。之后再继续，对特征图进行卷积，特征之间继续组合，获得更复杂的特征图。

又因为**池化层**的存在，会不断提取一定范围内最强烈的特征，并且缩小张量的大小，使得大范围内的特征组合也能够捕捉到。

通过特征角度来看卷积网络的话，那么**1x1卷积也就很好理解了**。即使1x1卷积前后的张量大小完全不变，比如说 $16 \times 16 \times 64 \rightarrow 16 \times 16 \times 64$ 这样的卷积，看上去好像是没有变化。但实际上，**可能通过特征之间的互动，已经由之前的64个特征图组成了新的64个特征图**。有时候我理解一个这样的1x1卷积操作，就会把它当成是一次**对之前特征的整理**。

一个概念上需要澄清的是，虽然说1x1卷积，而且也从融合特征角度，给了它特殊的理解。但如果再仔细看看的话，就会发现实际上**1x1卷积就是全连接网络**。所以我们可以把最后的1x1网络当成某种程度上的1x1卷积。

……上面的网络最后几层，将张量展平然后输入全连接网络。因为剩下的特征图中都保留了很重要的信息，为了利用所有的信息，并且让它们获得足够的交互，所以直接输入全连接网络，获得最后的特征向量。这个特征向量能够用来干什么呢？一个很有趣的应用案例是Siamese网络。输入一张脸，输出一个128的特征向量，这个向量就类似于ID号码。

总结：

卷积、池化、CNN结构发展

参考文献：

1. 当我们在谈论 Deep Learning: CNN 其常见架构 (上) : <https://zhuanlan.zhihu.com/p/27023778>
2. 当我们在谈论 Deep Learning: CNN 其常见架构 (下) : <https://zhuanlan.zhihu.com/p/27235732>

CNN 其实可以看作 DNN 的一种特殊形式。它跟传统 DNN 标志性的区别在于两点：

- Convolution Kernel
- Pooling

首先解释一下 Convolution (卷积)。一般我们接触过的都是一维信号的卷积，也就是

$$y[n] = x[n] * h[n] = \sum_k x[k]h[n - k]$$

在信号处理中， $x[n]$ 是输入信号， $h[n]$ 是单位响应。于是输出信号 $y[n]$ 就是输入信号 $x[n]$ 响应的延迟叠加。这也就是一维卷积本质：加权叠加/积分。

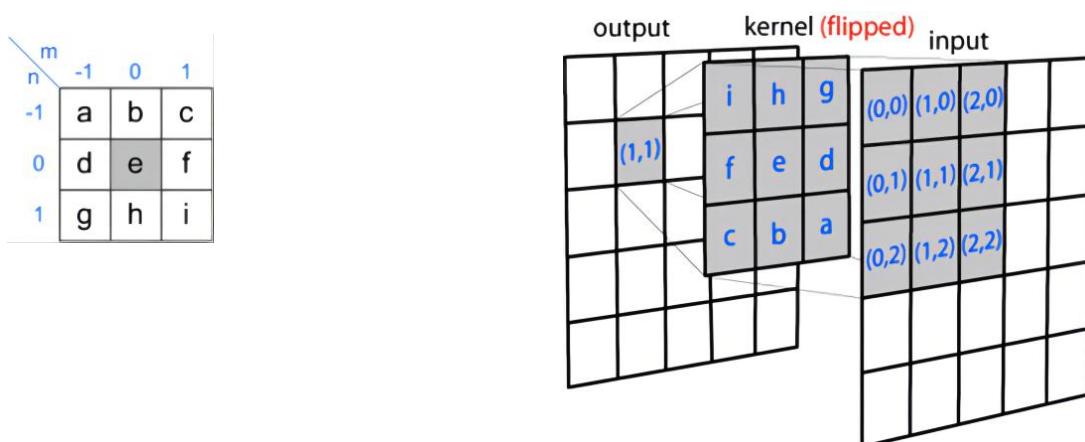
对于二维信号，比如图像，卷积的公式就成了

$$y[m, n] = x[m, n] * h[m, n] = \sum_j \sum_i x[i, j]h[m - i, n - j]$$

假设现在 Convolution Kernel 大小是 3×3 ，我们就可以化简上式为

$$\begin{aligned} y[1, 1] &= \sum_{j=-\infty}^{\infty} \sum_{i=-\infty}^{\infty} x[i, j] \cdot h[1-i, 1-j] \\ &= x[0, 0] \cdot h[1, 1] + x[1, 0] \cdot h[0, 1] + x[2, 0] \cdot h[-1, 1] \\ &\quad + x[0, 1] \cdot h[1, 0] + x[1, 1] \cdot h[0, 0] + x[2, 1] \cdot h[-1, 0] \\ &\quad + x[0, 2] \cdot h[1, -1] + x[1, 2] \cdot h[0, -1] + x[2, 2] \cdot h[-1, -1] \end{aligned}$$

看公式很繁琐，我们画个图看看，假如 Convolution Kernel 如下图。从 Input Image 到 Output Image 的变化如下：



可以看出，其实二维卷积一样也是加权叠加/积分。需要注意的是，其中 Convolution Kernel 进行了水平和竖直方向的翻转。

总结：

Convolution Kernel 的意义

- Convolution Kernel 在图像处理中并不是新事物，Sobel 算子等滤波算子，一直都在被用于边缘检测等工作中，只是以前被称为 Filter。
- Convolution Kernel 的一个属性就是局部性。即它只关注局部特征。局部的程度取决于 Convolution Kernel 的大小。比如用 Sobel 算子进行边缘检测，本质就是比较图像邻近像素的相似性。
- 也可以从另外一个角度理解 Convolution Kernel 的意义。信号处理中，时域卷积对应频域相乘。所以原图像与 Convolution Kernel 的卷积，其实对应频域中对图像频段进行选择。比如，图像中的边缘和轮廓属于是高频信息，图像中区域强度的综合考量属于低频信息。在传统图像处理里，这些物理意义是指导设计 Convolution Kernel 的一个重要方面。

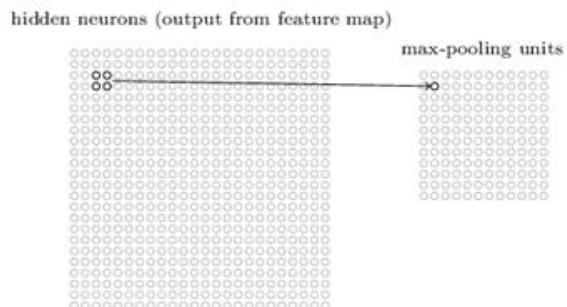
CNN 的 Convolution Kernel

- CNN 中的 Convolution Kernel 跟传统的 Convolution Kernel 本质没有什么不同。
- 同时，CNN 有一些它独特的地方，比如各种定义，以及它属于 DNN 的那些属性：
 - CNN 可以看作是 DNN 的一种简化形式，Input 和 Output 是 DNN 中的 Layer，Convolution Kernel 则是这两层连线对应的 w ，且与 DNN 一样，会加一个参数 Bias b 。
 - 一个 Convolution Kernel 在与 Input 不同区域做卷积时，它的参数是固定不变的。放在 DNN 的框架中理解，就是对 Output Layer 中的神经元而言，它们的 w 和 b 是相同的，只是与 Input Layer 中连接的节点在改变。在 CNN 里，这叫做 Shared Weights and Biases。
 - 在 CNN 中，Convolution Kernel 可能是高维的。假如输入是 $m \times n \times k$ 维的，那么一般 Convolution Kernel 就会选择为 $d \times d \times k$ 维，也就是与输入的 Depth 一致。
 - 最重要的一点，在 CNN 中，Convolution Kernel 的权值其实就是 w ，因此不需要提前设计，而是跟 DNN 一样利用 GD 来优化。
 - Convolution Kernel 卷积后得到的会是原图的某些特征（如边缘信息），所以在 CNN 中，Convolution Kernel 卷积得到的 Layer 称作 Feature Map。
 - 一般 CNN 中一层会含有多个 Convolution Kernel，目的是学习出 Input 的不同特征，对应得到多个 Feature Map。又由于 Convolution Kernel 的参数是通过 GD 优化得到而非设定的，于是 w 的初始化就显得格外重要。

Pooling

Pooling的本质，其实是采样。Pooling 对于输入的 Feature Map，选择某种方式对其进行压缩。如右图，表示的就是对 Feature Map 2×2 邻域内的值，选择最大值输出到下一层，这叫做 Max-Pooling。于是一个 $2N \times 2N$ 的 Feature Map 被压缩到了 $N \times N$ 。

除此之外，还有 Mean-Pooling，Stochastic-Pooling 等。它们的具体实现如名称所示，具体选择哪一个则取决于具体的任务。



Pooling 的意义，主要有两点：

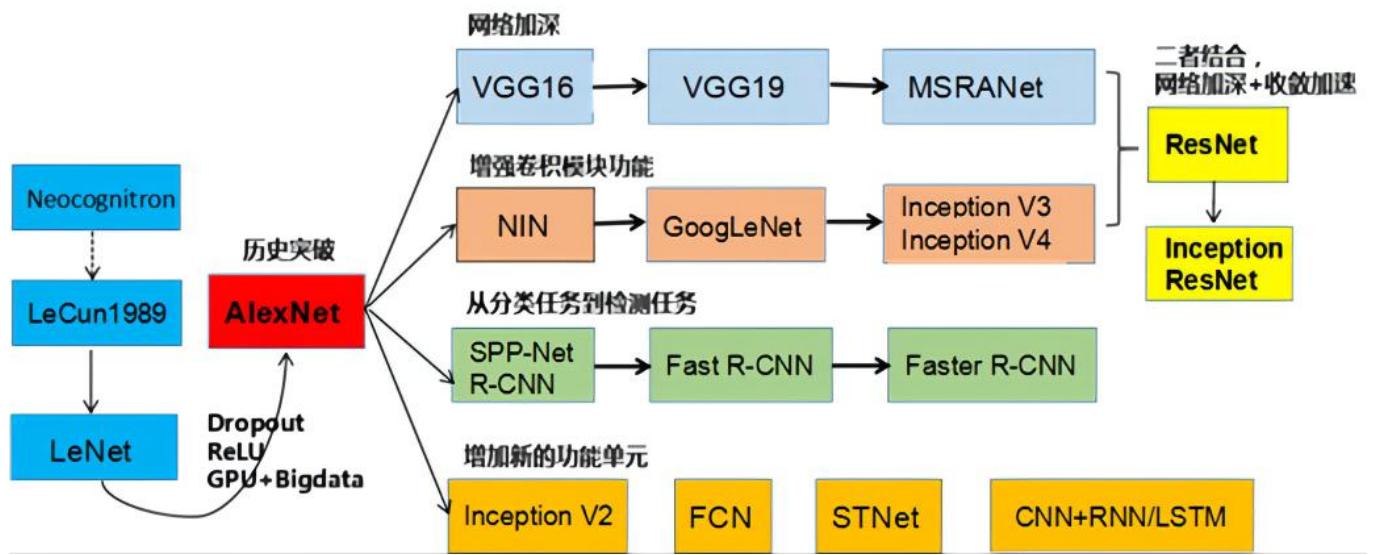
- 第一个就是减少参数。通过对 Feature Map 降维，有效减少后续层需要的参数。
- 另一个则是 Translation Invariance。它表示对于 Input，当其中像素在邻域发生微小位移时，Pooling Layer 的输出是不变的。这就增强了网络的鲁棒性，有一定抗扰动的作用。

小结

- 从上面我们可以看出来，CNN 里结构大都对应着传统图像处理某种操作。区别在于，以前是我们利用专业知识设计好每个操作的细节，而现在是利用训练样本+优化算法学习出网络的参数。在实际工程中，我们也必须根据实际物理含义对 CNN 结构进行取舍。
- 但是，随着 Convolution 的堆叠，Feature Map 变得越来越抽象，人类已经很难去理解了。为了攻克这个黑箱，现在大家也都在尝试各种不同的方式来对 CNN 中的细节进行理解，因为如果没有足够深的理解，或许很难发挥出 CNN 更多的能力。不过，这就是另外一个很大的课题了。

CNN 架构发展

CNN 的架构发展可以参考刘昕博士总结的图：



CNN 与 Backpropagation

CNN 可以看成是普通 FC (Fully Connected) DNN 的特殊形式。特殊在以下几个方面：

- CNN 中 Convolution Kernel 对应的两个 Layer 神经元之间的连接是稀疏的，当然，也可以将那些无连接看成是 w 恒为 0 的连接。
- CNN 中存在 Pooling Layer。

由于 CNN 是多种 Layer 的花式堆叠，不像 FC DNN 有统一的结构，因此无法给出整个网络的 BP 公式，这里我们只关注两个 Layer 之间的 BP。

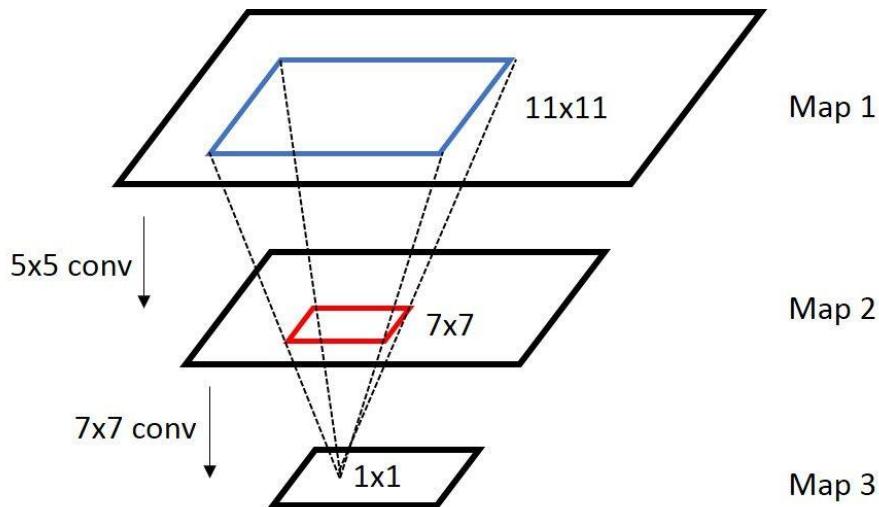
由于 Pooling Layer 的 BP 相对简单，就不增加篇幅了。建议对 FC DNN 的 BP 不熟悉的同学，可以先看看本专栏的“[当我们在谈论 Deep Learning：DNN 与 Backpropagation](#)”，下面整体思路与该篇相似，且会直接使用其中某些结论。

感受野(Receptive Field)

参考文献:

1. Deep Visualization: 可视化并理解CNN: <https://zhuanlan.zhihu.com/p/24833574>
2. CNN: 接受视野 (Receptive Field) : <https://zhuanlan.zhihu.com/p/41955458>
3. PyTorch 实现经典模型3: VGG

感受野是一个非常重要的概念, receptive field往往是描述两个feature maps A/B上神经元的关系, 假设从A经过若干个操作得到B, 这时候B上的一个区域 $area_b$ 只会跟A上的一个区域 $area_a$ 相关, 这时候 $area_a$ 成为 $area_b$ 的感受野。用图片来表示:



在上图里面, map3 里 1x1 的区域对应 map2 的 receptive field 是那个红色的 7x7 的区域, 而 map2 里 7x7 的区域对应于 map1 的 receptive field 是蓝色的 11x11 的区域, 所以 map3 里 1x1 的区域对应 map1 的 receptive field 是蓝色的 11x11 的区域。

receptive field的计算公式如下:

① 对于Convolution/Pooling layer:

$$r_i = s_i \cdot (r_{i+1} - 1) + k_i$$

其中 r_i 表示第 i 层 layer 的输入的某个区域, s_i 表示第 i 层 layer 的步长, k_i 表示 kernel size, 注意, 不需要考虑 padding size。

② 对于Neuron layer(ReLU/Sigmoid/...):

$$r_i = r_{i+1}$$

很容易看出, 除了不考虑padding之外, 上述过程与反卷积计算 feature map 的尺寸的公式是一样的。

总结:

我们举个例子：

首先图片经过 AlexNet 做一次前向传播，之后在 conv5 层可以得到一个 feature map，我们对这个 feature map 做一次 max pooling，只保留其中响应最大的值 (1x1)，然后根据以上感受野的计算公式一路计算回输入图像中：

AlexNet



其中ReLU和LRN层感受野不会产生变化，所以我们只需要考虑pool层和conv层：

$$\text{conv5: } r_6 = s_6 \cdot (r_7 - 1) + k_6 = 1 \cdot (1 - 1) + 3 = 3$$

$$\text{conv4: } r_5 = 1 \cdot (3 - 1) + 3 = 5$$

$$\text{conv3: } r_4 = 1 \cdot (5 - 1) + 3 = 7$$

$$\text{pool2: } r_3 = 2 \cdot (7 - 1) + 3 = 15$$

$$\text{conv2: } r_2 = 1 \cdot (15 - 1) + 5 = 19$$

$$\text{pool1: } r_1 = 2 \cdot (19 - 1) + 3 = 39$$

$$\text{conv1: } r_0 = 4 \cdot (39 - 1) + 11 = 163$$

所以，conv5层一个feature map中响应最大的值所对应的输入图中的感受野是163x163的尺寸，
我们从原图中切割出163x163的图片块即可。这就是论文中图片块的来源。

[1311.2901] Visualizing and Understanding Convolutional Networks
<https://link.zhihu.com/?target=https%3A//arxiv.org/abs/1311.2901>

params	AlexNet	FLOPs
4M	FC 1000	4M
16M	FC 4096 / ReLU	16M
37M	FC 4096 / ReLU	37M
442K	Max Pool 3x3s2	
74M	Conv 3x3s1, 256 / ReLU	74M
1.3M	Conv 3x3s1, 384 / ReLU	112M
884K	Conv 3x3s1, 384 / ReLU	149M
307K	Max Pool 3x3s2	
223M	Local Response Norm	
Conv 5x5s1, 256 / ReLU	r7	
Max Pool 3x3s2	r6	
Local Response Norm	r5	
Conv 11x11s4, 96 / ReLU	r4	
Max Pool 3x3s2	r3	
Local Response Norm	r2	
Conv 11x11s4, 96 / ReLU	r1	
105M		

左图中 Local Response Norm 不改变接受视野，因此在计算时可以忽略，实际需要计算的如下图：

Conv 3x3s1, 256 / ReLU	r7
Conv 3x3s1, 384 / ReLU	r6
Conv 3x3s1, 384 / ReLU	r5
Max Pool 3x3s2	r4
Conv 5x5s1, 256 / ReLU	r3
Max Pool 3x3s2	r2
Conv 11x11s4, 96 / ReLU	r1

实际需要计算的层

$$\text{conv5: } r_6 = s_6 \cdot (r_7 - 1) + k_6 = 1 \cdot (1 - 1) + 3 = 3$$

$$\text{conv4: } r_5 = 1 \cdot (3 - 1) + 3 = 5$$

$$\text{conv3: } r_4 = 1 \cdot (5 - 1) + 3 = 7$$

$$\text{pool2: } r_3 = 2 \cdot (7 - 1) + 3 = 15$$

$$\text{conv2: } r_2 = 1 \cdot (15 - 1) + 5 = 19$$

$$\text{pool1: } r_1 = 2 \cdot (19 - 1) + 3 = 39$$

$$\text{conv1: } r_0 = 4 \cdot (39 - 1) + 11 = 163$$

因此第5层卷积在输入层的感受视野为 163。

依据此就可以更好的理解此论文：《Visualizing and Understanding Convolutional Networks》
<https://link.zhihu.com/?target=https%3A//arxiv.org/abs/1311.2901>

总结：

感受野 (Receptive Field) 的定义是卷积神经网络每一层输出的**特征图** (feature map) 上的像素点在输入图片上映射的区域大小。通俗解释就是特征图上的一个点跟原图上有关系的点的区域。

感受野被称作是CNN中最重要的概念之一，目标检测流行的算法如 SSD，Faster RCNN 和 prior box 和 anchor box 的设计都是以感受野为依据做的设计。

- 如果conv1:5x5 stride=1, valid感受野是多少？

结论：

- 一个卷积核(5x5)感受野大小与两个3x3卷积核感受野等效，以此类推三个3x3卷积核感受野与一个7x7卷积核等效。

感受野计算公式：

$$r_n = r_{n-1} + (k_n - 1) \prod_{i=1}^{n-1} s_i$$

其中， r_{n-1} : 上一层感受野大小； k_n : 本层卷积核尺寸； s_i : 卷积步幅。

还有写作其他形式

$$RF_{l+1} = RF_l + (kernel_size - 1) * stride$$

eg: (原图感受野大小为 1)

卷积层	卷积核	步长	卷积方式	感受野大小
conv1:	3x3	stride=1	valid	1+(3-1)=3
conv2:	3x3	stride=1	valid	3+(3-1)*1=5
conv3:	3x3	stride=1	valid	5+(3-1)*1*1=7
conv4:	3x3	stride=1	valid	7+(3-1)*1*1*2=11
maxp:	2x2	stride=1	valid	11+(2-1)*1*1*2*1=13

注意：感受野这里计算理论值，实际起作用的感受野大小小于理论感受野。

DNN (Deep Neural Network)

参考文献:

1. 当我们在谈论 Deep Learning: DNN 与 Backpropagation <https://zhuanlan.zhihu.com/p/25794795>
2. 当我们在谈论 Deep Learning: DNN 与它的参数们 (壹) <https://zhuanlan.zhihu.com/p/26122560>
3. 当我们在谈论 Deep Learning: DNN 与它的参数们 (贰) <https://zhuanlan.zhihu.com/p/26392287>
4. 当我们在谈论 Deep Learning: DNN 与它的参数们 (叁) <https://zhuanlan.zhihu.com/p/26682707>
5. 文中截图大部分出自李宏毅老师的课件: <https://link.zhihu.com/?target=http%3A//speech.ee.ntu.edu.tw/~tlkagk/index.html>

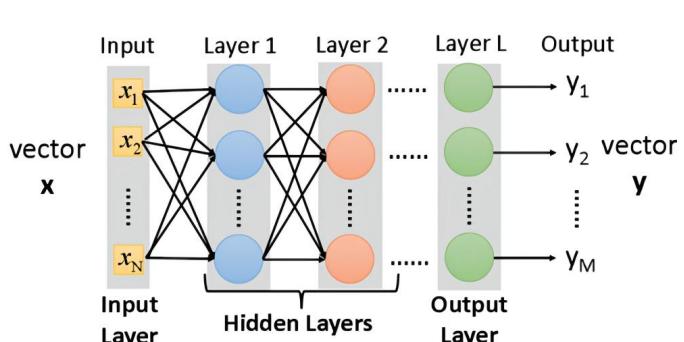
Deep Learning 是众多机器学习算法中的一种。它起源于60年代的 Perceptron，经过80年代的 Artificial Neural Network，现阶段被称为 Deep Learning。迄今为止，是“有监督学习”领域最强大的算法类型，暂时还没有“之一”。同时，它也正在往“无监督”和“强化学习”领域扩散。

除了算法效果突出，Deep Learning 另一个特点则是又一个“应用倒逼理论”的例子。Deep Learning 依靠强大的泛化能力在越来越多的应用领域开花，但是却极其缺乏理论依据。不论是模型的解释性、结构的设计、参数的选择等等，现阶段大多是依靠经验、试错。很多在以前或许被称为 Trick 的手段，在 Deep Learning 中却可能是重要的调参方法。

这个系列意在从 DNN (Deep Neural Network) 开始，对 Deep Learning 领域一些常见的算法进行介绍及说明。

DNN 的结构

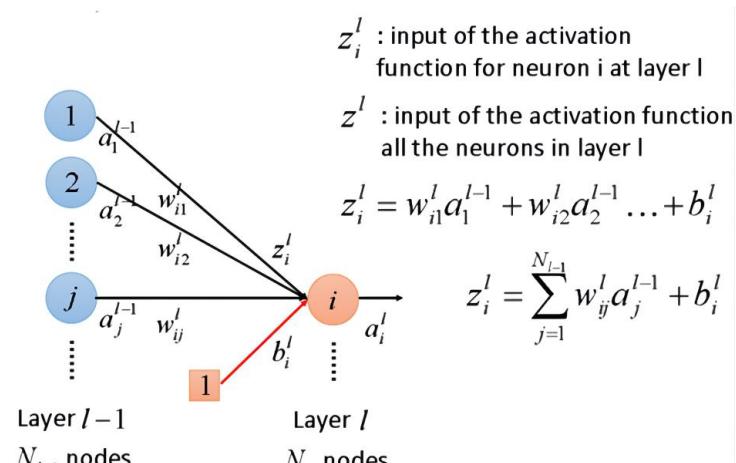
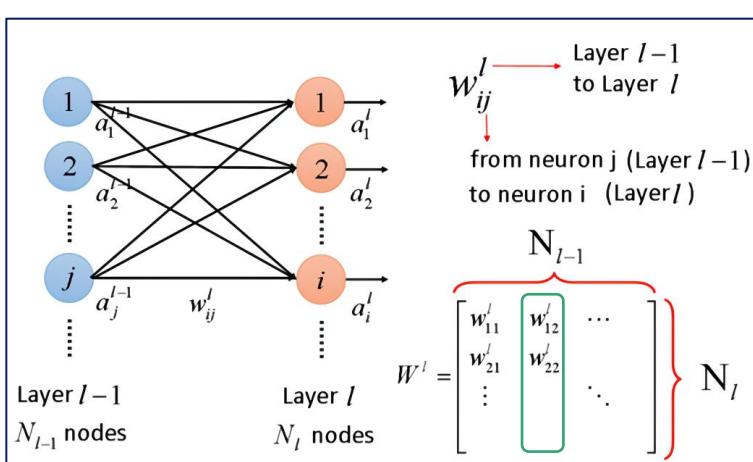
DNN (Deep Neural Network) 的基本结构如下。其中 x 表示一个训练样本向量， y 表示期望的输出向量。



z_i^l 表示第 l 层第 i 个神经元的输入。
 a_i^l 表示第 l 层第 i 个神经元的输出。
 b_i^l 表示第 l 层第 i 个神经元输入对应的偏置。
 w_{ij}^l 表示从 a_j^{l-1} 连接到 z_i^l 的权重。

于是 z_i^l 本质就是所有第 $l - 1$ 层输出跟 b_i^l 的线性组合。同时，第 $l - 1$ 层到第 l 层的所有权重能写成向量的形式

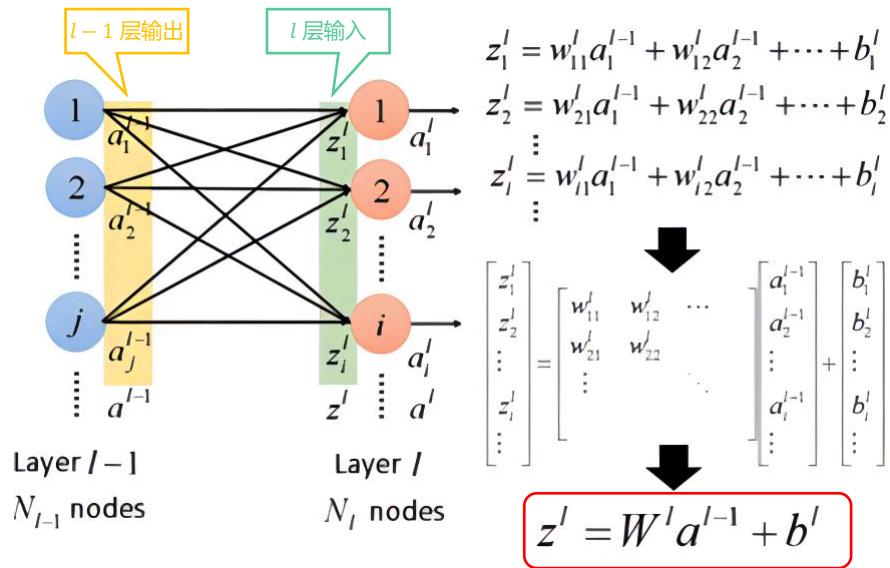
W^l ，其中第 j 列 $W_{:,j}^l$ 表示从 a_j^{l-1} 发出的权重。



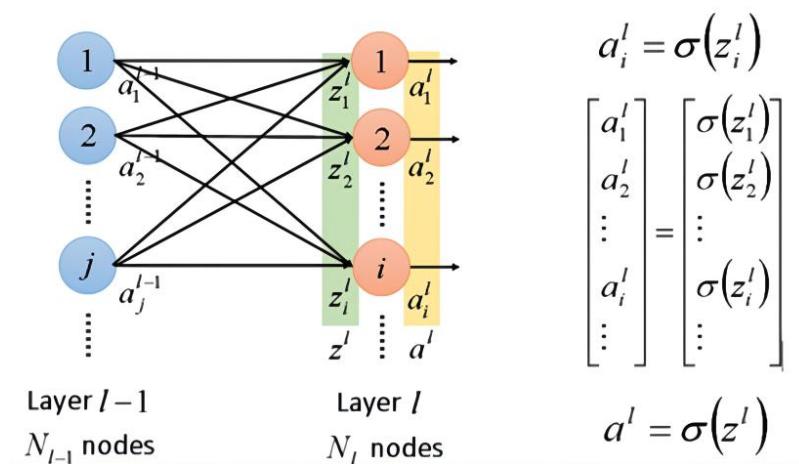
接下来我们分别用 z^l , a^l , b^l 分别表示第 l 层所有输入构成的向量、所有输出构成的向量、以及所有偏置构成的向量。于是, z^l 与 a^{l-1} 的关系可以表示为

$$z^l = W_l \times a^{l-1} + b^l$$

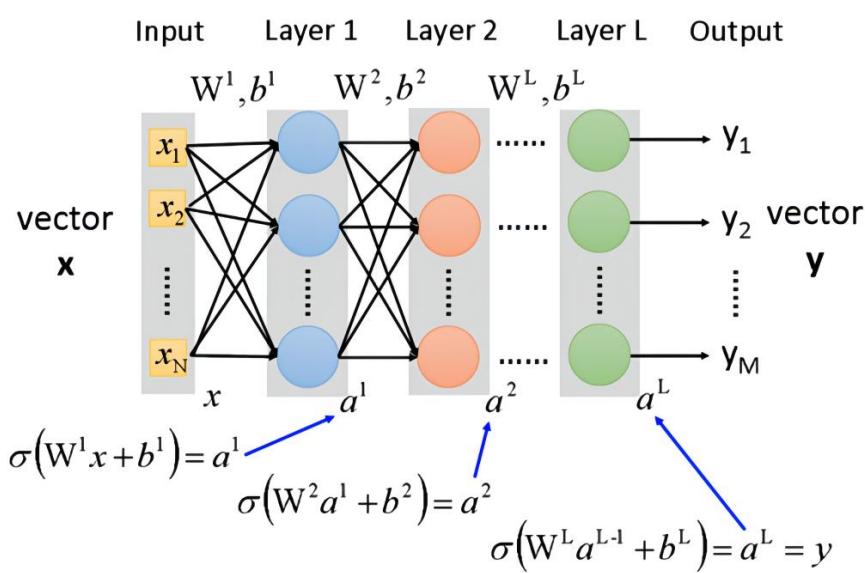
过程说明如下:



对于第 l 层第 i 个神经元, 其输入 z_i^l 与输出 a_i^l 的关系如下图, 即经过一个激活函数 σ 进行了变换。我们也可以将它们的关系描述成向量的形式, 即 $a^l = \sigma(z^l)$ 。



因此, 一个基础的 DNN 的网络就是下图的形式



总结:

Back Propagation

当设计好了 DNN 的结构，且有了训练样本，再给出损失函数的定义，接下来就可以求 DNN 中的参数 W 和 b 了。

所用的算法，本质就是梯度下降法，这里称为 BP (Back Propagation) 算法。需要注意的是，Deep Learning 中所说的 BP 跟传统的 BP 不完全一样，已经简化成纯粹的梯度下降，更容易理解。

先简单地回顾下求导的链式法则：

Case 1 $y = g(x)$ $z = h(y)$

$$\Delta x \rightarrow \Delta y \rightarrow \Delta z \quad \frac{dz}{dx} = \frac{dz}{dy} \frac{dy}{dx}$$

Case 2

$$x = g(s) \quad y = h(s) \quad z = k(x, y)$$

$$\begin{array}{ccc} \Delta s & \xrightarrow{\Delta x} & \Delta z \\ & \searrow & \nearrow \\ & \Delta y & \end{array} \quad \frac{\partial z}{\partial s} = \frac{\partial z}{\partial x} \frac{\partial x}{\partial s} + \frac{\partial z}{\partial y} \frac{\partial y}{\partial s}$$

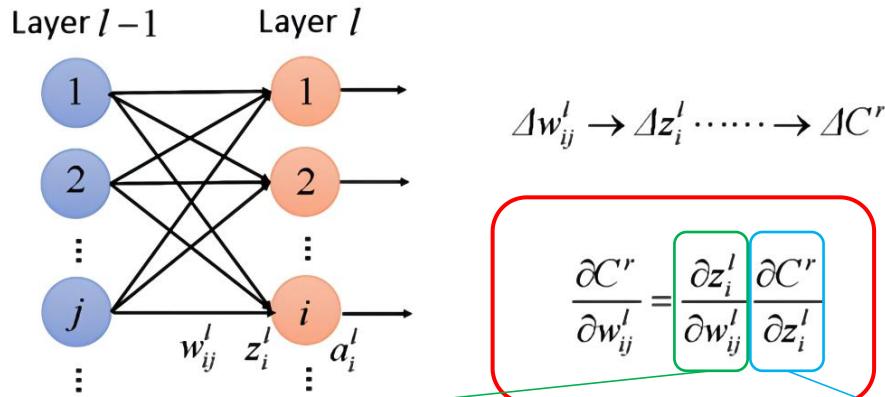
假设我们的训练样本集合为 $\{(x^1, \hat{y}^1), \dots, (x^r, \hat{y}^r), \dots (x^R, \hat{y}^R)\}$ ，且定义损失函数为 $C(\theta)$ ，其中 θ 为损失函数的参数向量，则有

$$C(\theta) = \frac{1}{R} \sum_r C^r(\theta)$$
$$\nabla C(\theta) = \frac{1}{R} \sum_r \nabla C^r(\theta)$$

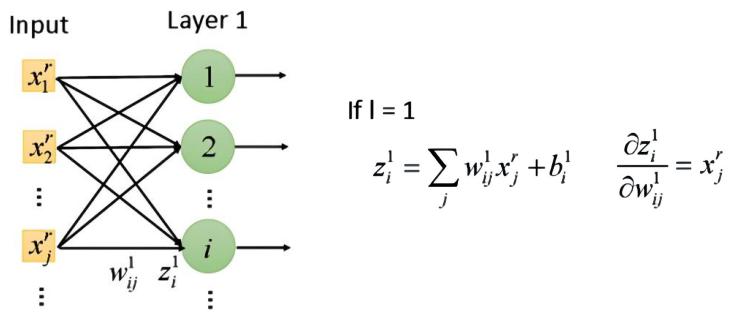
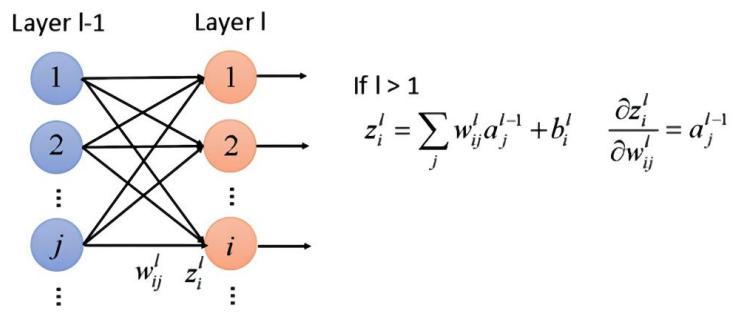
对于 DNN 来说， θ 其实只是包含了 w 和 b ，所以我们的目的就是求 $\frac{\partial C^r}{\partial w_{ij}^l}$ 和 $\frac{\partial C^r}{\partial b_i^l}$

我们抽出第 l 与第 $l - 1$ 层的网络。

① 根据链式法则, $\frac{\partial C^r}{\partial w_{ij}^l}$ 其实由两部分相乘所得, 如下图



对于第一项, $\frac{\partial z_i^l}{\partial w_{ij}^l}$ 可以根据下面的方式计算。



归纳起来, 对于第一项, 有

$$\frac{\partial z_i^l}{\partial w_{ij}^l} = \begin{cases} x_j^r, & \text{if } l = 1 \\ a_j^{l-1}, & \text{if } l > 1 \end{cases}$$

上式中 x_j^r 和 a_j^{l-1} 可在 forward pass 过程中计算得出。

总结:

- 当 $l < L$, 即我们观察的是中间某层 Hidden Layer 时, 同样根据链式法则, δ_i^l 与 δ_i^{l+1} 有以下关系:

$$\begin{aligned} \delta_i^l &= \frac{\partial C^r}{\partial z_i^l} \\ &= \frac{\partial a_i^l}{\partial z_i^l} \frac{\partial C^r}{\partial a_i^l} \\ &= \frac{\partial a_i^l}{\partial z_i^l} \sum_k \frac{\partial z_k^{l+1}}{\partial a_i^l} \frac{\partial C^r}{\partial z_k^{l+1}} \end{aligned}$$

$$= \sigma'(z_i^l) \sum_k w_{ki}^{l+1} \delta_k^{l+1}$$

- 当 $l = L$, 即我们观察的是输出层时:

$$\begin{aligned} \delta_i^L &= \frac{\partial C^r}{\partial z_i^L} \\ &= \frac{\partial y_i^r}{\partial z_i^L} \frac{\partial C^r}{\partial y_i^r} \\ &= \sigma'(z_i^L) \frac{\partial C^r}{\partial y_i^r} \end{aligned}$$

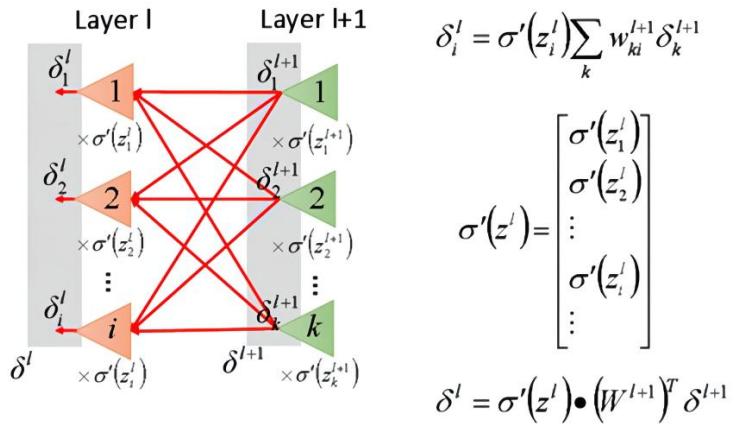
其中, $\frac{\partial C^r}{\partial y_i^r}$ 根据 C^r 的定义方式计算。
如定义为 $C^r = \|y^r - \hat{y}^r\|$

归纳起来, 对于第二项, 写成向量形式, 有:

$$\begin{aligned} \delta^l &= \frac{\partial C^r}{\partial z^l} \\ &= \left\{ ((W^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l), \text{ if } l < L \right. \\ &\quad \left. \nabla_{y^r} C^r \odot \sigma'(z^L), \text{ if } l = L \right\} \end{aligned}$$

其中 \odot 表示向量 element-wise 的乘法

为了更方便理解，其实对于 δ_i^l 的计算，可以看做是 DNN 网络的反向传播，如下图

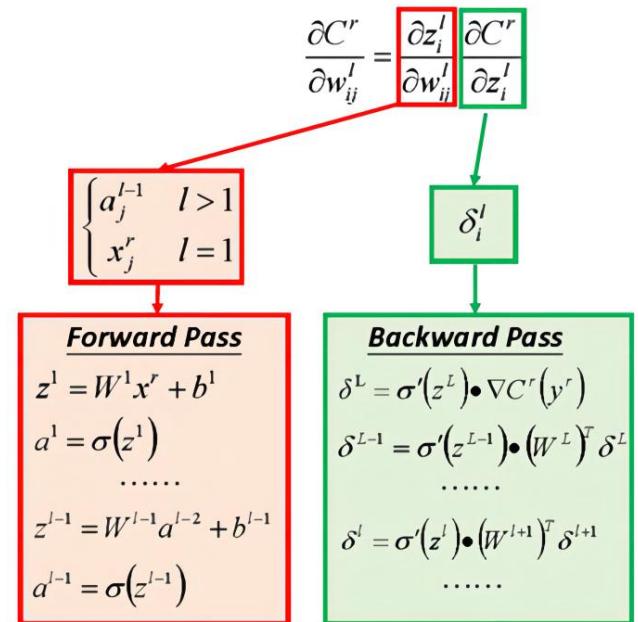


② 我们还需要计算 $\frac{\partial C^r}{\partial b_i^l}$ ，其推导如下：

$$\frac{\partial C^r}{\partial b_i^l} = \frac{\partial z_i^l}{\partial b_i^l} \frac{\partial C^r}{\partial z_i^l} = 1 \cdot \frac{\partial C^r}{\partial z_i^l} = \delta_i^l$$

因此，整个 Back Propagation 的过程本质：

- 就是先利用正向计算中得出的 $\frac{\partial z_i^l}{\partial w_{ij}^l}$ 的值，
- 再从最后一层反向计算 $\delta_i^l = \frac{\partial C^r}{\partial z_i^l}$ ，它等于该层的 $\frac{\partial C^r}{\partial b_i^l}$ 。
- 利用链式法则求出 $\frac{\partial C^r}{\partial w_{ij}^l}$ 。
- 利用链式法则求出其它各层关于 w 和 b 的偏导数。



综上所示，即 Back Propagation 的流程，传说中 BP 的四个公式也在过程中都推导过了，汇总如下：

$$\delta^l = ((W^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$$

$$\delta^L = \nabla_{y^r} C^r \odot \sigma'(z^L)$$

$$\frac{\partial C^r}{\partial b_i^l} = \delta_i^l$$

$$\frac{\partial C^r}{\partial w_{ij}^l} = a_j^{l-1} \delta_i^l$$

总结：

Mini-batch Gradient Descent

GD (Gradient Descent) 的本质就是希望逐步逼近最优，其迭代公式为：

$$\theta^i = \theta^{i-1} - \eta \nabla C(\theta^{i-1})$$

最常用的 GD，是使用所有的训练样本来求梯度，即：

$$\nabla C(\theta^{i-1}) = \frac{1}{R} \sum_r \nabla C^r(\theta^{i-1})$$

利用所有的训练样本来求梯度，好处是梯度下降的方向会稳定地朝着极值方向并收敛，不容易受噪声影响；但是问题也比较明显，一个是考虑了所有的数据所有收敛慢，同时容易陷入局部最优。随着数据量的增大，更大的问题是每更新一次参数，计算量太大；同时，由于考虑了所有数据，收敛就慢。

因此 SGD (Stochastic Gradient Descent) 就应运而生：每次 Iteration 计算梯度并更新参数时只考虑一个样本，
<删：对每一个样本> 所有样本执行完一次这个过程称为一次 Epoch。即：

$$\theta^i = \theta^{i-1} - \eta \nabla C^r(\theta^{i-1})$$

SGD 的好处就是加快了收敛的速度。问题是由于根据一个样本求的梯度，方向并不一定指向极值方向；甚至可能出现每一次 Iteration 求出的梯度方向差异巨大，最终无法收敛。

因此 Mini-batch SGD (Stochastic Gradient Descent) 又应运而生：每次 Iteration 计算梯度并更新参数时考虑 Batch_Size 个样本（称为一个 Batch），对所有样本执行完这个过程称为一次 Epoch。公式如下：

$$\nabla C(\theta^{i-1}) = \frac{1}{B} \sum_{x^r \in b} \nabla C^r(\theta^{i-1})$$

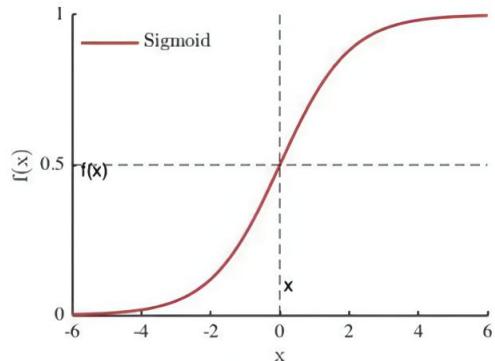
b 表示本次选择的 Batch。

B 表示 Batch_Size。

Sigmoid

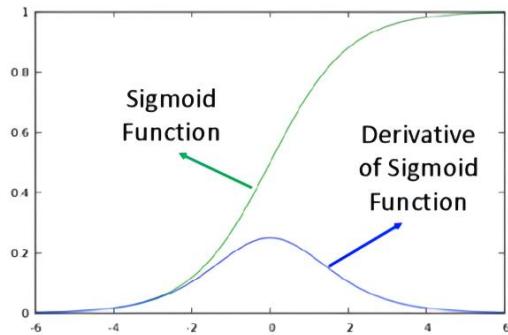
最开始接触 ANN 的时候，大家听说的 Activation Function 应该还都是 Sigmoid 函数。它的定义如下：

$$f(x) = \frac{1}{1 + e^{-x}}$$



Sigmoid 函数优点很多：

1. 作为 Activation Function，它是单调递增的，能够很好地描述被激活的程度
2. Sigmoid 能将 $(-\infty, +\infty)$ 转换为 $(0, 1)$ ，避免数据在传递过程中太过发散，同时输出还能被理解成某种概率
3. Sigmoid 在定义域内处处可导，而且导数很好算。 $f'(x) = f(x)(1 - f(x))$ ，图形如下，可以看出 $f'(x) < 1$



但是，Sigmoid 的导数也带来了一些问题。在上一部分我们介绍过如何通过 BP 来计算 $\frac{\partial C^r}{\partial w_{ij}^l}$ ，如下

$$\frac{\partial C^r}{\partial w_{ij}^l} = a_j^{l-1} \delta_i^l$$

$$\delta^l = (W^{l+1})^T \delta^{l+1} \odot \sigma'(z^l)$$

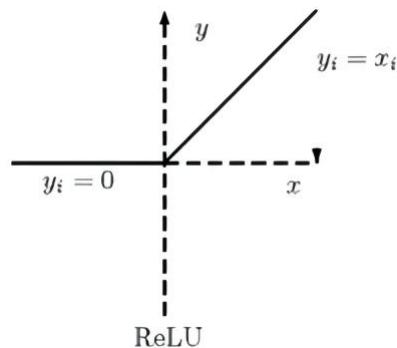
假设 $\sigma(z)$ 为 Sigmoid 函数，有 $\sigma'(z) < 1$ 。因此随着 l 的减小， δ^l 会越来越小，从而 $\frac{\partial C^r}{\partial w_{ij}^l}$ 也会越来越小。

当 DNN 比较深的时候，较前层的参数求出的梯度会非常小，几乎不会再更新，这种现象被称为 **Gradient Vanish**。

ReLU

为了缓解 Gradient Vanish 现象，现在大家都会使用 ReLU (Rectified Linear Unit) , 其定义如下：

$$y = \begin{cases} x, & \text{if } x \geq 0 \\ 0, & \text{if } x < 0 \end{cases}$$



ReLU 除了具有 Sigmoid 函数大部分的优点外，还有

- 对某个神经元，当 $x > 0$ 时，其导数为1，因而缓解了 Gradient Vanish 现象。因此，ReLU 也是最近几年非常受欢迎的激活函数。
- 对某个神经元，当 $x < 0$ 时，其输出也是0，也就是对后续的网络不起作用，可以看作从网络中被移除了。因此在整个训练过程结束后，整个网络会呈现出一种稀疏的状态，也就是会有很多的神经元由于在网络中不起作用，可以当成不存在。这种稀疏也表明 ReLU 对避免网络过拟合有一定作用。

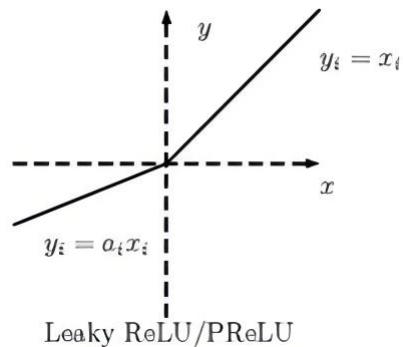
同时，ReLU 也有自己的缺陷：

- 可以看出当 $x < 0$ 时，不仅输出为0，ReLU 的导数也为0。即对应的参数不再更新。因此这个神经元再也没有被激活的机会了，这种现象被称为 **dying ReLU**。
- 第二个现象叫 **Bias shift**。在将数据输入 DNN 时我们一般会进行 **高斯归一**，但是由于 ReLU 的输出恒大于0，会导致后续层输出的数据分布发生偏移。对于很深的网络，这可能会导致无法收敛。

LReLU、PReLU

为了解决 dying ReLU 的问题，有学者提出了 LReLU (Leaky Rectified Linear Unit) 、 PReLU (Parametric Rectified Linear Unit) 。它们被定义为：

$$y_i = \begin{cases} x_i & \text{if } (x_i > 0) \\ a_i x_i & \text{if } (x_i \leq 0) \end{cases}$$



LReLU 可以避免 dying ReLU，使神经元在任何输入下都能持续更新参数。对于 LReLU， a_i 是需要我们提前设定好的较小的数，比如0.01。但是，提前设定 a_i 也带来了调参的难度。为了解决难以选择合适参数的问题，出现了 PReLU。

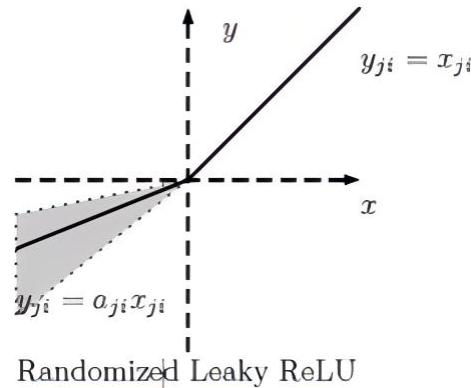
PReLU 公式跟 LReLU 是一样的，不同的是 PReLU 的 a_i 是通过训练样本自动学习的。学习的方式依然是 BP 算法。但是，PReLU 在小数据下容易过拟合。具体的步骤可以参考 "He, Kaiming, Zhang, Xiangyu, Ren, Shaoqing, and Sun, Jian. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. 2015."

RReLU

RReLU (Randomized Rectified Linear Unit) 是 LReLU 的随机版本。它最早出现在 Kaggle NDSB 比赛中，定义如下：

$$y_{ji} = \begin{cases} x_{ji} & \text{if } (x_{ji} > 0) \\ a_{ji} y_{ji} & \text{if } (x_{ji} \leq 0) \end{cases}$$

其中 $a_{ji} \sim U(l, u)$, $l < u$ and $l, u \in [0, 1]$



在训练时， a_{ji} 是一个保持在 (l, u) 内均匀分布的随机变量，而在测试中被固定为 $\frac{l+u}{2}$ 。

关于 RReLU 并没有太多的文章，想进一步了解的话可以参考 “Xu B, Wang N, Chen T, et al. Empirical Evaluation of Rectified Activations in Convolutional Network. 2015.”

Others

Activation Function 是一个比较发散的课题，在不同的任务中有不同的选择，暂时先不做更多的介绍。其它的比如 Maxout、ELU 等，有兴趣的同学可以自己查找相关资料。

总结：

Cost Function

Softmax + Cross Entropy

在 DNN 中进行多类分类时，使用的最多的 Cost Function 就是 Cross Entropy Loss，这里我们也主要介绍它。

首先，对于两个分布 $p(x)$ 与 $q(x)$ ，交叉熵的定义为：

$$H(p, q) = - \int p(x) \log q(x) dx$$

交叉熵用于衡量两个分布的相似性。当 $p(x)$ 与 $q(x)$ 的分布完全一致时， $H(p, q)$ 最小。

对于分类问题，假设进行 K 类分类，我们将某个样本 x 的真实的类别标记 \hat{y} 用向量 $[\hat{y}_1, \dots, \hat{y}_k, \dots, \hat{y}_K]$ 描述，其中

$$\hat{y}_k = \begin{cases} 1, & \text{if } k = \hat{y} \\ 0, & \text{if } k \neq \hat{y} \end{cases}$$

于是， \hat{y}_k 可以被理解为 x 为第 k 类的概率。因此，DNN 的输出 y 应该也是一个向量 $[y_1, \dots, y_k, \dots, y_K]$ ，且需要与 $[\hat{y}_1, \dots, \hat{y}_k, \dots, \hat{y}_K]$ 尽可能的接近。

为了描述这两个分布的相似性，于是就引入了 **Cross Entropy Loss**，其定义如下：

$$C(y, \hat{y}) = - \sum_{k=1}^K \hat{y}_k \log(y_k)$$

对于两类分类， $\hat{y} \in \{0, 1\}$ ， $C(y, \hat{y})$ 可以简化为：

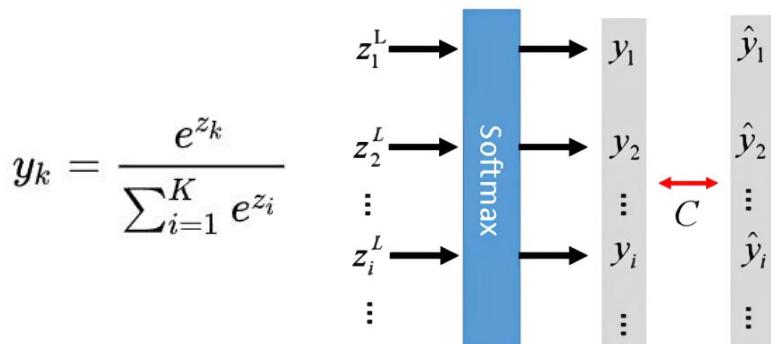
$$C(y, \hat{y}) = -\hat{y} \log(y) - (1 - \hat{y}) \log(1 - y)$$

还有一点需要注意，在前面介绍DNN时，我们假设了 Output Layer 中从 z 到 y 的变换是普通的 Activation Function，即 σ 函数，如图：

Ordinary Output layer

$$\begin{aligned} z_1^L &\xrightarrow{\sigma} y_1 = \sigma(z_1^L) \\ z_2^L &\xrightarrow{\sigma} y_2 = \sigma(z_2^L) \\ z_3^L &\xrightarrow{\sigma} y_3 = \sigma(z_3^L) \end{aligned}$$

但是在多类分类时， y_k 表示的样本属于第 y 类的概率。此时将 z_k 转换为 y_k 的操作即 **Softmax**，其公式和图像表达如下：



总结：

其实使用 Cross Entropy Loss 作为损失函数不是 DNN 的专属，在“当我们在谈论GBDT：Gradient Boosting 用于分类与回归”中介绍过 GBDT 在进行多类分类的时候就是使用的 Softmax + Cross Entropy，只是当时它被称为对数损失函数 (Log-Likelihood Loss)。

最后需要说明的是，当我们使用 MSE (均方误差) 作为 Cost Function，会导致[公式]的更新速度依赖于 Activation Function 的导数，在 Activation Function 为 Sigmoid 函数时很容易更新缓慢；而使用 Softmax + Cross Entropy Loss，能够有效克服这个问题。这也是 Cross Entropy Loss 在分类问题中使用较多的原因之一。

Regularization 与 Weight Decay

Weight Decay

假设我们使用的是 L2 Regularization，则对应的公式如下：

$$C_1(\theta) = C(\theta) + \lambda \frac{1}{2} \|\theta\|^2$$

其中， $\theta = \{W^1, W^2, \dots\}$ ，即所有 W 的集合，注意这里只考虑了 W 没有考虑 b （原因似乎是因为实验表明 b 的约束对结果并没有太大的提升，不过暂时还没找到出处，以后看到再补充）。

$\|\theta\| = \sum_l (w_{ij}^l)^2$ ，即所有 w 的平方和； λ 用于控制 Regularization 的程度，也叫 **Weight Decay**，越大则模型越倾向于简单。

在用 BP 更新参数时，有

$$\frac{\partial C_1(\theta)}{\partial w} = \frac{\partial C(\theta)}{\partial w} + \lambda w$$

于是在更新 w 时，更新式为

$$\begin{aligned} w^{t+1} &= w^t - \eta \frac{\partial C_1(\theta)}{\partial w} \\ &= (1 - \eta \lambda) w^t - \eta \frac{\partial C(\theta)}{\partial w} \end{aligned}$$

可以看出，更新式比没有 Weight Decay 多了 $(1 - \eta \lambda)$ 这一项；且由于它小于 1， w 会在进行梯度下降的同时会被持续减小，这也是 Weight Decay 这个名称的由来。

Regularization 的理解

第一种，是从 PRML 中看到的

对于 L1、L2 Regularization，他们的 Cost Function 分别可以写作

$$\min_w Loss(w) + \lambda \|w\|_1$$
$$\min_w Loss(w) + \lambda \|w\|_2$$

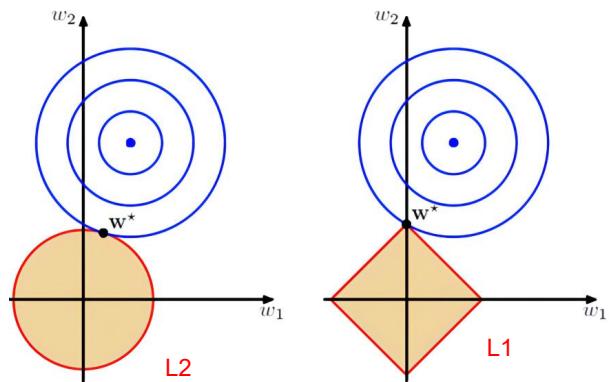
于是，我们可以写出它们的等价形式，如下

$$\min_w Loss(w) \text{ s.t. } \|w\|_1 \leq C$$
$$\min_w Loss(w) \text{ s.t. } \|w\|_2 \leq C$$

可以看出，我们其实是通过 L1 或 L2 约束将 w 限制在一个空间中，在此基础上求出使 Cost Function 最小的 w 。

右

假设现在考虑 w 有两维，如下图。其中黄色区域分别是 L2 和 L1 约束，蓝色是 Cost Function 的等高线。蓝色与红色的切点即求出的 w ，可以看出对于 L1 约束， w 更容易出现在坐标轴上，即只有一个维度上有非0值；而对于 L2 约束，则可能出现在象限的任何位置。这也是 L1 正则会带来稀疏解的解释之一。



第二种，是从概率的角度

假设给定观察数据为 D ，贝叶斯方法通过最大化后验概率估计参数 w ，即

$$\begin{aligned} w^* &= \operatorname{argmax}_w p(w|D) \\ &= \operatorname{argmax}_w \frac{p(D|w)p(w)}{p(D)} \\ &= \operatorname{argmax}_w p(D|w)p(w) \\ &= \operatorname{argmax}_w \log(p(D|w)) + \log(p(w)) \\ &= \operatorname{argmin}_w -\log(p(D|w)) - \log(p(w)) \end{aligned}$$

其中， $p(D|w)$ 是似然函数， $p(w)$ 是参数的先验。

当 w 服从 M 维 0 均值高斯分布，即

$$p(w) = N(w|0, \alpha^{-1} I) = \left(\frac{\alpha}{2\pi}\right)^{M/2} \exp\left\{-\frac{\alpha}{2} w^T w\right\}$$

代入上式，有

$$\begin{aligned} w^* &= \operatorname{argmin}_w -\log(p(D|w)) - \log(p(w)) \\ &= \operatorname{argmin}_w -\log(p(D|w)) + \frac{\alpha}{2} w^T w + C(\alpha) \end{aligned}$$

其中，第一项就是我们平时所定义的 Cost Function；第二项就是 L2 正则项；第三项当给定了 α 那么就是常数了。所以我们可以看出，L2 正则本质其实是给模型参数 w 添加了一个协方差为 $\alpha^{-1} I$ 的零均值高斯分布先验。而 α 越小，则协方差 $\alpha^{-1} I$ 越大，表明先验对 w 的约束越弱，模型的 variance 越大；反之，而 α 越大，则协方差 $\alpha^{-1} I$ 越小，表明先验对 w 的约束越强，模型越稳定。这与我们平时的理解是吻合的。

总结：

当 w 服从 M 维 0 均值同分布 Laplace 分布时，即

$$p(w) = \frac{1}{(2b)^M} \exp\left\{-\frac{\|w\|_1}{b}\right\}$$

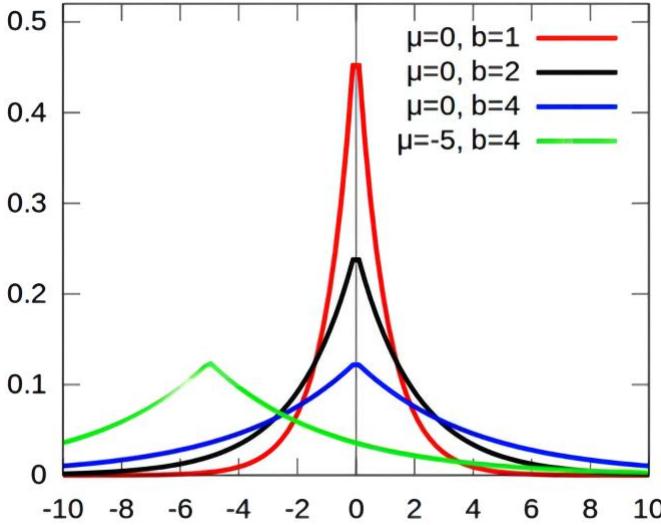
代入上式，有

$$\begin{aligned} w^* &= \operatorname{argmin}_w -\log(p(D|w)) - \log(p(w)) \\ &= \operatorname{argmin}_w -\log(p(D|w)) + \frac{1}{b} \|w\|_1 + C(b) \end{aligned}$$

同样，第一项就是我们平时所定义的 Cost Function；第二项就是 L1 正则项；第三项当给定了 b 那么就是常数了。一维 Laplace 分布如下：

$$p(w) = \frac{1}{2b} \exp\left\{-\frac{|w - \mu|}{b}\right\}$$

其概率密度如下图所示，可以看出它取值的特点是很大概率落在一个小范围内。当 $\mu = 0$ 时，它会以很大概率取值在 0 的附近，这也就是 L1 约束会出现稀疏解的原因。



Mini-batch SGD

Mini-batch SGD，它是 GD 与 SGD 的一个折中选择，有着较好的收敛效果与速度。

Algorithm 8.1 Stochastic gradient descent (SGD) update at training iteration k

Require: Learning rate ϵ_k .

Require: Initial parameter θ

while stopping criterion not met **do**

 Sample a minibatch of m examples from the training set $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$ with corresponding targets $\mathbf{y}^{(i)}$.

 Compute gradient estimate: $\hat{\mathbf{g}} \leftarrow +\frac{1}{m} \nabla_{\theta} \sum_i L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$

 Apply update: $\theta \leftarrow \theta - \epsilon \hat{\mathbf{g}}$

end while

对于 $\theta = \theta - \epsilon \hat{\mathbf{g}}$ ，我们可以这样理解：某次迭代的参数初始位置为 θ ，该参数以匀速 $-\epsilon \hat{\mathbf{g}}$ 发生位移，时长为单位时间，得到的新的位置，即新的 θ 。由于 $\hat{\mathbf{g}}$ 是根据训练样本的一个 Batch 求得的，所以变化可能会比较剧烈，也容易收噪声干扰，从而减慢收敛速度。

Momentum

为了提高 Mini-batch SGD 的收敛速度，利用动量（Momentum）来加速收敛的方法被提了出来，其步骤如下：

Algorithm 8.2 Stochastic gradient descent (SGD) with momentum

Require: Learning rate ϵ , momentum parameter α .

Require: Initial parameter θ , initial velocity v .

while stopping criterion not met **do**

 Sample a minibatch of m examples from the training set $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$ with corresponding targets $\mathbf{y}^{(i)}$.

 Compute gradient estimate: $\mathbf{g} \leftarrow \frac{1}{m} \nabla_{\theta} \sum_i L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$

 Compute velocity update: $v \leftarrow \alpha v - \epsilon g$

 Apply update: $\theta \leftarrow \theta + v$

end while

可以看出，与 Mini-batch SGD 的区别，其中一个就是 $\theta = \theta + v$ 这个步骤。即，参数改变的速度变成了 v ，而不再是 $-\epsilon \hat{\mathbf{g}}$ 。同时， $v = \alpha v - \epsilon g$ ，即此时 $-\epsilon g$ 可以看作是速度 v 的加速度，且 v 以 α 发生衰减。

于是有：

1. 参数以速度 v 发生改变， v 的加速度是 $-\epsilon g$ 。此时，每一个 Batch 的 g 的改变会以加速度的方式影响 v ，而 v 不会再突变（比如 v 方向的剧烈改变），这就带来了一定的抗噪声能力
2. 当速度 v 与加速度 $-\epsilon g$ 长期方向一致时，即使 $-\epsilon g$ 比较小，速度 v 也会逐渐加快，从而加快收敛速度
3. 同时，可以通过设置 α 来改变前期梯度 g 对当前 v 的影响。 α 越小，则前期的梯度 g 对现在的速度 v 的影响越小。 α 的含义可以类比成运动时的摩擦力，使参数不会无止境的改变下去。

Nesterov Momentum (NAG: Nesterov Accelerated Gradient)

Nesterov Momentum (Sutskever I, Martens J, Dahl G, et al. On the importance of initialization and momentum in deep learning. 2013.) 是对 Momentum 方法的一种修改，又称为 NAG (Nesterov Accelerated Gradient) , 其步骤如下：

Algorithm 8.3 Stochastic gradient descent (SGD) with Nesterov momentum

Require: Learning rate ϵ , momentum parameter α .

Require: Initial parameter θ , initial velocity v .

while stopping criterion not met **do**

 Sample a minibatch of m examples from the training set $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$ with corresponding labels $\mathbf{y}^{(i)}$.

 Apply interim update: $\tilde{\theta} \leftarrow \theta + \alpha v$

 Compute gradient (at interim point): $\mathbf{g} \leftarrow \frac{1}{m} \nabla_{\tilde{\theta}} \sum_i L(f(\mathbf{x}^{(i)}; \tilde{\theta}), \mathbf{y}^{(i)})$

 Compute velocity update: $v \leftarrow \alpha v - \epsilon g$

 Apply update: $\theta \leftarrow \theta + v$

end while

可以看出，Nesterov Momentum 是先对新的参数进行了预估， $\theta = \theta + \alpha v$ ，再进行梯度的计算。虽然只是进行了细微的改变，Nesterov Momentum 在效果上比 Momentum 却有着显著的提升。关于对这种提升原因的分析，可以参考比Momentum更快：揭开Nesterov Accelerated Gradient的真面目，这里就不再赘述了。<https://zhuanlan.zhihu.com/p/22810533>

AdaGrad

SGD 在参数更新的过程中，经常需要对 ϵ 进行衰减，原因也是很直观的：前期参数距离最优解比较远，需要较大的更新速度；而当到达了最优解附近，则需要较小的速度以精细搜索最优解。

而以上介绍的方法，它们的学习率 ϵ 都需要提前设定。如果想需要对 ϵ 进行衰减，衰减的策略也需要自己去设置。对于多维数据，不同维度的数据还需要考虑是否需要不同的 ϵ 衰减策略。于是，有学者在 Mini-Batch SGD 的基础上提出了自适应衰减 ϵ 的方法 (Duchi J, Hazan E, Singer Y. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. 2011.)，其步骤如下：

Algorithm 8.4 The AdaGrad algorithm

Require: Global learning rate ϵ

Require: Initial parameter θ

Require: Small constant δ , perhaps 10^{-7} , for numerical stability

Initialize gradient accumulation variable $r = \mathbf{0}$

while stopping criterion not met **do**

 Sample a minibatch of m examples from the training set $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$ with corresponding targets $\mathbf{y}^{(i)}$.

 Compute gradient: $\mathbf{g} \leftarrow \frac{1}{m} \nabla_{\theta} \sum_i L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$

 Accumulate squared gradient: $r \leftarrow r + \mathbf{g} \odot \mathbf{g}$

 Compute update: $\Delta\theta \leftarrow -\frac{\epsilon}{\delta + \sqrt{r}} \odot \mathbf{g}$. (Division and square root applied element-wise)

 Apply update: $\theta \leftarrow \theta + \Delta\theta$

end while

可以看出，它与 Mini-Batch SGD 的区别在于： ϵ 在更新的过程中一直在自适应的衰减。学习率 ϵ 衰减的幅度与该维度上梯度 \mathbf{g} 的累计平方和成反比，即：如果某个维度上 \mathbf{g} 一直较大，则该维度上的学习率 ϵ 衰减的较快； \mathbf{g} 较小，则该维度上的学习率 ϵ 衰减的较慢。

但按照《Deep Learning》的说法，在 DNN 的训练中，AdaGrad 在某些模型上效果不错，但并非全部。

总结：

RMSProp

RMSProp (Tieleman, T. and Hinton, G. Lecture 6.5 - rmsprop, COURSERA: Neural Networks for Machine Learning.2012) 是 AdaGrad 的一种修改形式，用于改善 AdaGrad 在非凸场景效果不好的情况。
RMSProp 的步骤如下：

Algorithm 8.5 The RMSProp algorithm

Require: Global learning rate ϵ , decay rate ρ .
Require: Initial parameter θ
Require: Small constant δ , usually 10^{-6} , used to stabilize division by small numbers.
Initialize accumulation variables $r = 0$
while stopping criterion not met **do**
 Sample a minibatch of m examples from the training set $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$ with corresponding targets $\mathbf{y}^{(i)}$.
 Compute gradient: $\mathbf{g} \leftarrow \frac{1}{m} \nabla_{\theta} \sum_i L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$
 Accumulate squared gradient: $\mathbf{r} \leftarrow \rho \mathbf{r} + (1 - \rho) \mathbf{g} \odot \mathbf{g}$
 Compute parameter update: $\Delta\theta = -\frac{\epsilon}{\sqrt{\delta+r}} \odot \mathbf{g}$. ($\frac{1}{\sqrt{\delta+r}}$ applied element-wise)
 Apply update: $\theta \leftarrow \theta + \Delta\theta$
end while

可以看出，RMSProp 与 AdaGrad 最大的区别就是在计算累计梯度时，多了一个衰减参数 ρ 。

AdaGrad 在计算累计梯度时，使用的是过去所有梯度的平方和。于是，在更新参数的过程中，如果某些迭代计算出的 \mathbf{g} 使累计梯度太大，后续参数更新速度就会剧烈减少从而过早停止更新。

RMSProp 通过衰减参数 ρ ，降低较为久远的 \mathbf{g} 的影响，从而改善了上述情况。由于其实践中较好的效果，RMSProp 也是目前 Deep Learning 中最常用的优化方法之一。

RMSProp + Nesterov Momentum

当我们把 RMSProp 和 Nesterov Momentum 融合，也即同时考虑了自适应更新学习率，以及 Momentum 的思想，就有如下算法：

Algorithm 8.6 RMSProp algorithm with Nesterov momentum

Require: Global learning rate ϵ , decay rate ρ , momentum coefficient α .

Require: Initial parameter θ , initial velocity v .

Initialize accumulation variable $r = 0$

while stopping criterion not met **do**

 Sample a minibatch of m examples from the training set $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$ with corresponding targets $\mathbf{y}^{(i)}$.

 Compute interim update: $\tilde{\theta} \leftarrow \theta + \alpha v$

 Compute gradient: $\mathbf{g} \leftarrow \frac{1}{m} \nabla_{\tilde{\theta}} \sum_i L(f(\mathbf{x}^{(i)}; \tilde{\theta}), \mathbf{y}^{(i)})$

 Accumulate gradient: $\mathbf{r} \leftarrow \rho \mathbf{r} + (1 - \rho) \mathbf{g} \odot \mathbf{g}$

 Compute velocity update: $v \leftarrow \alpha v - \frac{\epsilon}{\sqrt{\delta+r}} \odot \mathbf{g}$. ($\frac{1}{\sqrt{\delta+r}}$ applied element-wise)

 Apply update: $\theta \leftarrow \theta + v$

end while

Adam

Adam 的全称是 Adaptive Moment (Kingma, D. P., & Ba, J. L. Adam: a Method for Stochastic Optimization. 2015.), 同样也是一种融合了自适应学习率, 以及 Momentum 思想的方法。其步骤如下:

Algorithm 8.7 The Adam algorithm

Require: Step size ϵ (Suggested default: 0.001)

Require: Exponential decay rates for moment estimates, ρ_1 and ρ_2 in $[0, 1]$.
(Suggested defaults: 0.9 and 0.999 respectively)

Require: Small constant δ used for numerical stabilization. (Suggested default:
 10^{-8})

Require: Initial parameters θ

Initialize 1st and 2nd moment variables $s = \mathbf{0}, r = \mathbf{0}$

Initialize time step $t = 0$

while stopping criterion not met **do**

 Sample a minibatch of m examples from the training set $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$ with
 corresponding targets $\mathbf{y}^{(i)}$.

 Compute gradient: $\mathbf{g} \leftarrow \frac{1}{m} \nabla_{\theta} \sum_i L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$

$t \leftarrow t + 1$

 Update biased first moment estimate: $\mathbf{s} \leftarrow \rho_1 \mathbf{s} + (1 - \rho_1) \mathbf{g}$

 Update biased second moment estimate: $\mathbf{r} \leftarrow \rho_2 \mathbf{r} + (1 - \rho_2) \mathbf{g} \odot \mathbf{g}$

 Correct bias in first moment: $\hat{\mathbf{s}} \leftarrow \frac{\mathbf{s}}{1 - \rho_1^t}$

 Correct bias in second moment: $\hat{\mathbf{r}} \leftarrow \frac{\mathbf{r}}{1 - \rho_2^t}$

 Compute update: $\Delta\theta = -\epsilon \frac{\hat{\mathbf{s}}}{\sqrt{\hat{\mathbf{r}}} + \delta}$ (operations applied element-wise)

 Apply update: $\theta \leftarrow \theta + \Delta\theta$

end while

可以看出, Adam 就是在 RMSProp 的基础上, 又考虑了 Momentum。

由于 ρ_1, ρ_2 的设置近似 1, Adam 中的 s, r 其实分别是梯度 g 的一阶矩、二阶矩的估计。因此,
 \hat{s}/\hat{r} 属于 $[0, 1]$, 于是 $\Delta\theta$ 也有了约束。按照原文的说法, 这个性质可以帮助我们更好地调节 ϵ
; 同时, 《Deep Learning》中也提到, 这使 Adam 对参数更加的鲁棒。

bengio 的 "Deep Learning"

Others

Gradient Descent Optimization 这个课题在 Deep Learning 火热起来后也越发热闹, 各种方法络绎不绝。其它的方法如 Adamax、Nadam 等等, 有兴趣的同学可以去进行更多的学习。而到底哪个方法最好, 还是要根据实际场景选择, Mini-batch SGD 可能效果已经满足需求, 而 RMSprop 是现阶段比较常用的方法之一。

深度学习优化算法经历了 SGD → SGDM → NAG → AdaDelta → Adam → Nadam 这样的发展历程。

- SGD with Momentum
- SGD with Nesterov Acceleration (NAG 全称 Nesterov Accelerated Gradient, 是在 SGD、SGD-M 的基础上的进一步改进)
- AdaGrad
- AdaDelta / RMSProp
- Adam (Adaptive + Momentum)
- Nadam (Nesterov + Adam = Nadam)

这一段引用自: 一个框架看懂优化算法之异同 SGD/AdaGrad/Adam <https://zhuanlan.zhihu.com/p/32230623>

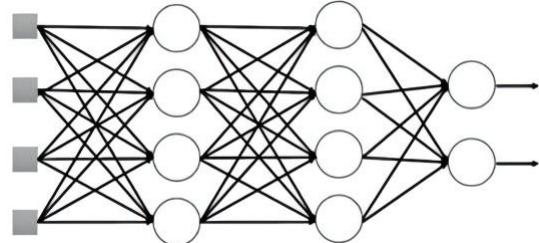
总结:

Dropout

Dropout 是 Hinton 在2014年提出的一种优化 DNN 的一种通用方法 (Srivastava N, Hinton G, Krizhevsky A, et al. Dropout: a simple way to prevent neural networks from overfitting. 2014)。虽然作者在文章中并没有对 Dropout 的原理进行太多的理论分析，但是他用大量的对比实验证明此方法的有效性与效果的稳定性。而随后，Dropout 也确实在实践中被证明是非常简单，且有效提高 DNN 效果的一种优化方法。（恩，这很 Deep Learning。。。所以很多老 ML 专家不喜欢 Deep Learning 论文其实是可以被理解的。）

下面简单介绍一下 Dropout 方法。

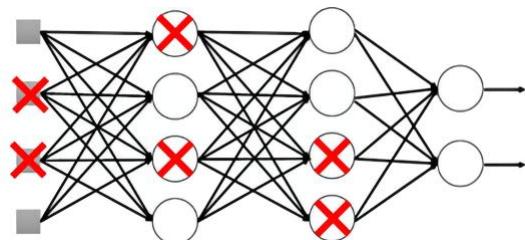
假设对于一般的 DNN，其结构如下。我们使用 Mini-batch SGD 的方式进行参数更新：



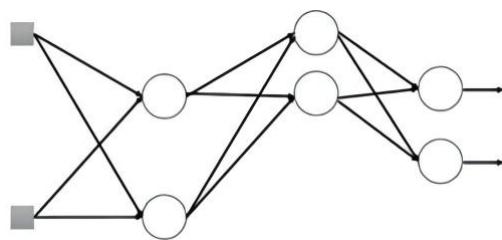
此时，DNN 对应的 Feed Forward 的公式为：

$$\begin{aligned} z_i^{(l+1)} &= \mathbf{w}_i^{(l+1)} \mathbf{y}^l + b_i^{(l+1)}, \\ y_i^{(l+1)} &= f(z_i^{(l+1)}), \end{aligned}$$

在训练时，对于一次 Mini-batch，我们设定一个概率 p ，表示 DNN 中每个节点（包括输入节点，以及 Hidden Layer 的节点）被保留的概率。也就是 DNN 中的每个节点，都会以 $1 - p$ 概率被移除。如下：



与被移除的节相连的那些线在这次 Mini-batch 训练中也会被移除，于是 DNN 可以被看作如下结构。对这个结构中的参数应用 Mini-batch SGD 进行更新，被删除的结构对应的参数本次不更新。

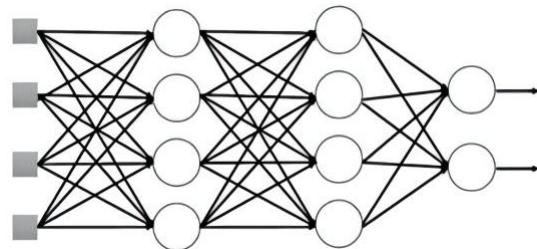


将这种思想写成公式，即 Feed Forward 公式变成如下。其中， $r_j^{(l)}$ 服从 Bernoulli 分布，即以 p 的概率为 1，以 $1 - p$ 的概率为 0。

$$\begin{aligned} r_j^{(l)} &\sim \text{Bernoulli}(p), \\ \tilde{\mathbf{y}}^{(l)} &= \mathbf{r}^{(l)} * \mathbf{y}^{(l)}, \\ z_i^{(l+1)} &= \mathbf{w}_i^{(l+1)} \tilde{\mathbf{y}}^l + b_i^{(l+1)}, \\ y_i^{(l+1)} &= f(z_i^{(l+1)}). \end{aligned}$$

总结：

在测试时，会使用初始的 DNN 结构，即：



但是，每个节点对应的参数 w 需要乘以 p ，即 $W_{test}^{(l)} = pW^{(l)}$ 。

在文章中，作者给出 p 的建议值是：对于输入层， $p = 0.8$ ；对于 Hidden Layer， $p = 0.5$ 。

Dropout 可以看作是一种 Ensemble Learning 方法。Ensemble Learning 本质就是用同样的数据去训练不同的模型，然后进行融合；或者用不同的数据去训练相同的模型，然后再进行融合。在 DNN 的训练过程中，对于不同的 Epoch，相同的 Mini-batch，如果 Dropout 后的结构不同，那么就对应第一种情况；整个训练过程中，如果某两次 Dropout 后的结构相同，而 Mini-batch 不同，那么就对应第二种情况。所以，Dropout 也拥有 Ensemble Learning 的优点，即可以缓解 overfitting。

除此之外，Dropout 还有一些仿生学之类的解释。不过我一般都觉得这只是学者们在卖萌，有兴趣的同学可以参考原文。

在本篇中，我们会先简介一些“数据预处理”的方法，然后介绍由此演进的 Batch Normalization 结构。Batch Normalization 最大的优点，就在于能缓解 DNN 在调参时的困难，是一种简单但是很实用的结构。

数据预处理

“数据预处理”是 ML 中的一个重要步骤，对 Feature Engineering 和结果有着巨大的影响。同样，在 DNN 中，一般数据也需要进行预处理，才会送入网络中进行训练或测试。这时，“数据预处理”是希望缓解以下问题：

- 范围、量纲统一：对每一维的特征，进行范围、量纲的统一，防止数值过大的维度影响分类效果。
- 加速 Gradient Descent：特征不同维度的尺度、相关性等可能会使 Loss Function 的等高线变得复杂，进而减慢 Gradient Descent。
- Covariate Shift：训练、测试样本的分布不一致时，模型无法很好地泛化。这个问题一般的处理办法是 Domain Adaption。

Mean Subtraction 与 Z-Score

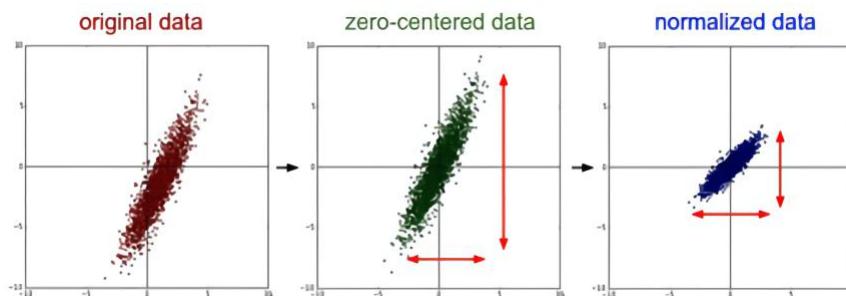
Mean Subtraction 其实就是将每一维的特征的向量 x 中心搬到原点，使均值为0。其公式如下，其中 μ 是向量 x 的均值。

$$\hat{x} = x - \mu$$

Z-Score 是在 Mean Subtraction 的基础上，使 x 的均值为0，标准差为1。其公式如下，其中 μ 是向量 x 的均值， σ 为标准差。

$$\hat{x} = \frac{x - \mu}{\sigma}$$

如果用二维数据来展示，则原始数据、Mean Subtraction、Z-Score 后的图像分别为：



Mean Subtraction 和 Z-Score 虽然简单，却是统一范围、量纲的最常用方法，且能显著提高 Gradient Descent 的速度。

PCA 与 Whitening

PCA (Principle Components Analysis) 是比较通识的技术了，这里只简单回顾一下其步骤。对于训练样本集 $\hat{X} = [\hat{x}^{(1)}, \dots, \hat{x}^{(i)}, \dots, \hat{x}^{(N)}]$ ，其 PCA 部分步骤如下：

$$\begin{aligned}\hat{\mu} &= \frac{1}{N} \sum \hat{x}^{(i)} \\ x^{(i)} &= \hat{x}^{(i)} - \hat{\mu} \\ X &= [x^{(1)}, \dots, x^{(i)}, \dots, x^{(N)}] \\ \Sigma &= \frac{1}{N} X X^T \\ U S V^T &= \Sigma\end{aligned}$$

需要注意的是， Σ 是样本不同维度之间的协方差矩阵。一般我们为了求 Σ 的特征值，都是直接对 X 做 SVD，即 $U S V^T = X$ 。这里写作 $U S V^T = \Sigma$ ，是为了顺便说明，对 Σ 求特征值、对 Σ 做 SVD、对 X 做 SVD，其实都是等价的。具体的说明，可以参考从 PCA 和 SVD 的关系拾遗。

在 SVD 后， U 是 X 一组相互正交的基向量，且其每一列为 Σ 的特征值； S 的对角线上的值为 X 的奇异值，且是 Σ 的非零特征值的平方。

此时，我们可以将 X 投影到新的基 U 上，得到 $X_{proj} = U^T X$ 。有 X_{proj} 不同维度的协方差矩阵 Σ_{proj} 为对角阵，推导如下

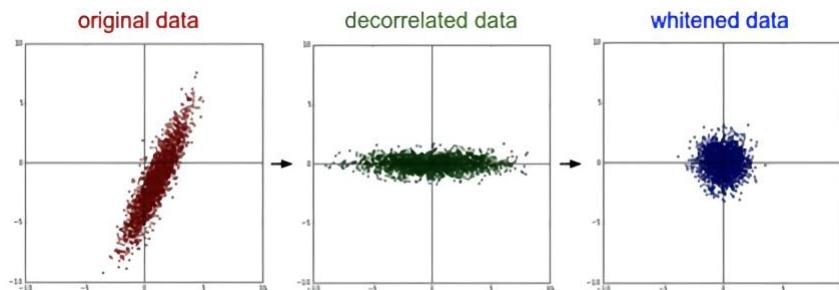
$$\begin{aligned}\Sigma_{proj} &= \frac{1}{N} X_{proj} X_{proj}^T \\ &= \frac{1}{N} U^T X X^T U \\ &= U^T \Sigma U \\ &= S\end{aligned}$$

即， X_{proj} 的每个维度之间不相关。
当然，还可以利用 U 和 S 进行特征降维，不是这里的重点，不再赘述。

接下来，介绍 Whitening。

利用 PCA 的一些步骤，我们对 \hat{X} 每一维去相关，得到了 X_{proj} ，且其每一维都是零中心。Whitening 就是在此基础上更进一步，使 X_{proj} 每一维有相同的方差。上面已经介绍过， X_{proj} 的协方差矩阵为对角阵 S ，于是我们对 X_{proj} 每一维分别除以对应的标准差，即 S 对角线对应元素的开方。

用二维数据来展示，则原始数据、去相关、Whitening 后的图像分别为：



Whitening 后能提高 Gradient Descent 的速度，且数据性质很好（去相关，零均值等方差）。但是，由于 Whitening 需要进行 SVD，计算量会比较大。

Batch Normalization

WHY Batch Normalization ?

前面介绍的都是数据预处理的方法，但是在 DNN 中，除了面对量纲、GD 速度慢、Covariate Shift 这些 ML 共有的问题，还有一个独特的问题，就是 Internal Covariate Shift。

1. Internal Covariate Shift 是作者在 Batch Normalization 论文中，仿照 Covariate Shift 提出的概念。具体来说：对 DNN 某一层，随着 GD 训练的过程中参数的改变，该层的输出数据的分布可能会改变；此时，对于下一层，相当于输入数据的分布改变了，这就类似于 Covariate Shift。这种输入数据分布的改变，可能会使 DNN 的难以学习到好的参数，从而影响 DNN 的效果。
2. 同时，DNN 为了补偿输入数据分布改变带来的损失，需要更多的时间来调整参数，这可能会使 GD 的速度下降。
3. 同时，DNN 还有 Gradient Vanish 的问题。当输入分布改变，可能使本层的 Activation Function 饱和，进而导致 Gradient Vanish。而随着深度的增加，较深的层会受到前面层参数变化的影响叠加，它输入分布的改变可能会更明显。

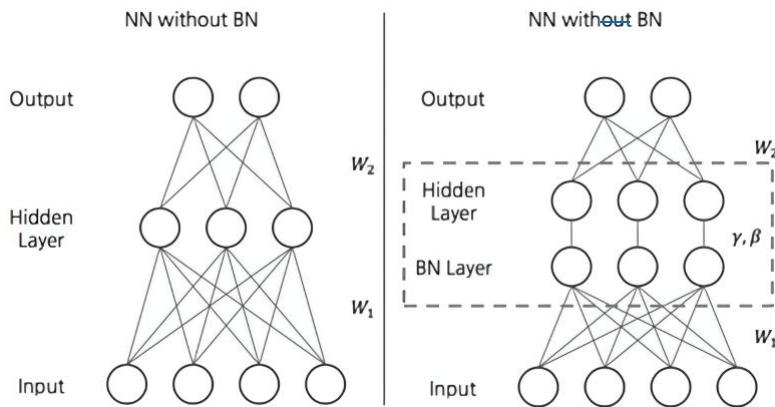
这里有一点需要明确，对于上面的 Internal Covariate Shift，我（参考文献作者）是根据原始论文写出的自己的理解。我还见过一种说法，是这样理解 Internal Covariate Shift：对 DNN 每一层，经过了该层的变换，其输出数据与输入数据的分布一般是不会一样的，这种分布的差异随着深度的增加会越发明显，但是它们描述的样本与标记仍然是一样的。个人认为与原文中想表达的有所偏差，但是也列在这里，以免因为自己的原因误导大家。有兴趣的同学可以参考原文。

BN 步骤

增加了 Batch Normalization 的 DNN，其训练步骤如下：

1. 增加 BN 结构：对 DNN 中每一个 Activation，在它们前面放置一个 BN Layer (Batch Normalization Layer)。相当于以前的将 $Wu + b$ 输入 Activation Function，现在将 $BN(Wu + b)$ 输入 Activation Function。
2. 求解参数：利用 BP 求解 DNN 中的参数。

所以，DNN 与 BN+DNN 的结构分别如下(下图中的文中 Hidden Layer 请脑补成 Activation)：



Batch Normalization 中的 Batch，跟 miniBatch 是一样的，说明它也是针对一个 Batch 进行的操作。

对于一个样本，假设有这么一个标量 x ， x 可以理解成这个样本的某一维特征的值，或者是这个样本输入到 DNN 后某个 Activation Function 的输入值。那么对于一个 Batch，假设对应的 x 的集合为 $B = \{x_1, \dots, x_i, \dots, x_m\}$ ，于是其对应的 BN Layer 的输出集合 $\{y_i\}$ 就通过以下方式计算：

总结：

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_1 \dots m\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \text{BN}_{\gamma, \beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^m x_i \quad // \text{mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_{\mathcal{B}})^2 \quad // \text{mini-batch variance}$$

$$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \quad // \text{normalize}$$

$$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma, \beta}(x_i) \quad // \text{scale and shift}$$

其步骤如下：

- 对一个 Batch 中的样本 x_i , 进行 Z-Score 归一化得到 \hat{x}_i 。
- 由于 Z-Score 归一化后对 \hat{x}_i 的范围增加了很强的约束, 为了让 DNN 学习出合适的输入, 利用 scale γ 和 shift β 对 \hat{x}_i 进行变换。不过, 这里 γ 和 β 都是网络自己学习出来的。

有几个问题需要在这里进行说明：

- 为什么是对 Activation Function 的输入进行 BN (即 $BN(Wu + b)$) , 而非对 Hidden Layer 的每一个输入进行 BN ($W * BN(u) + b$) 。按照作者的解释, 由于 Hidden Layer 的输入 u 是上一层非线性 Activation Function 的输出, 在训练初期其分布还在剧烈改变, 此时约束其一阶矩和二阶矩无法很好地缓解 Covariate Shift; 而 $BN(Wu + b)$ 的分布更接近 Gaussian Distribution, 限制其一阶矩和二阶矩能使输入到 Activation Function 的值分布更加稳定。
- 还有 $y_i = \gamma \hat{x}_i + \beta$ 这一步的意义, 即为什么在对数据进行归一化后还要再做变换。我觉得这个跟 AutoEncoder 的意义是类似的, 即 AutoEncoder 是一种智能的 PCA, BN 则是一种智能的 Z-Score。不过这些都只是感性的理解罢了, 并没什么严谨的证明就是了。

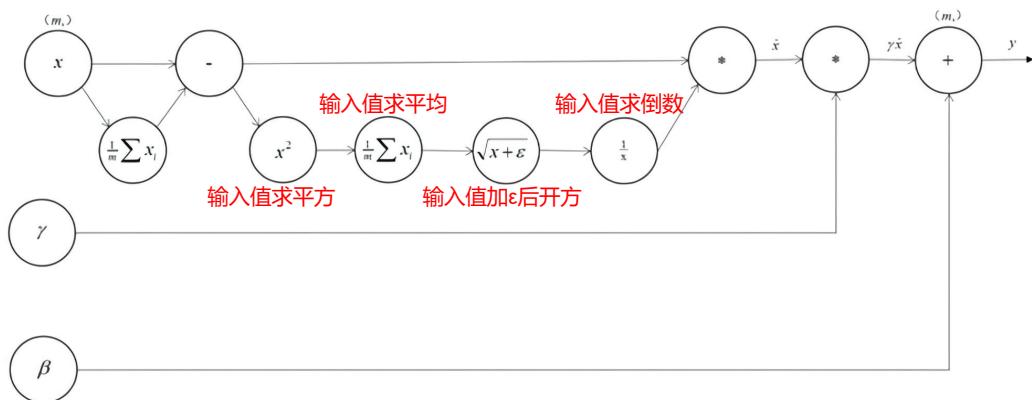
BN + BP

在 DNN 中增加了 BN Layer 后, 即增加了参数 γ 和 β , BP 也相应地发生了改变。这里, 参考 Understanding the backward pass through Batch Normalization Layer, 以 Computational Graph 的方式来简单解释 BN 的 BP。

<https://link.zhihu.com/?target=https%3A//kratzert.github.io/2016/02/12/understanding-the-gradient-flow-through-the-batch-normalization-layer.html>

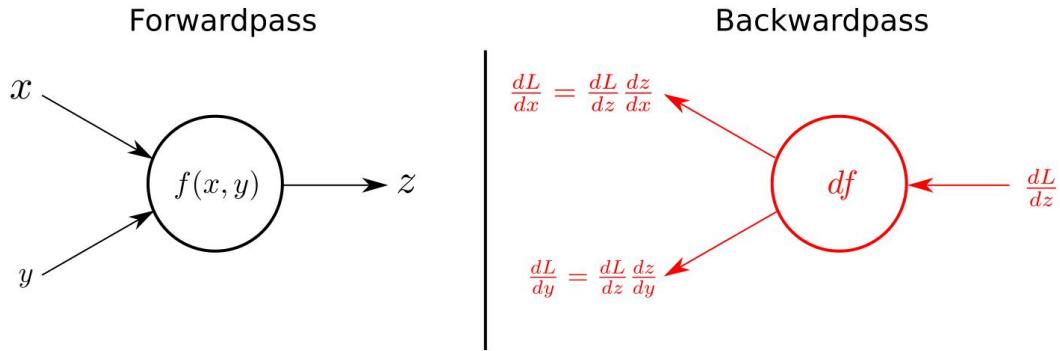
定义 $\mathbf{x} = [x_1, \dots, x_i, \dots, x_m]^T$, 其中 m 表示一个 Batch 中样本的数量, x_i 是一个样本对应的某个标量, 比如理解成 DNN 中某个节点的 Activation Function 的输入。于是, \mathbf{x} 是一个 m 维的列向量。这里跟 Understanding the backward pass through Batch Normalization Layer 定义不同, 主要是为了让后续的公式更好理解。

根据 BN 的公式, 我们能得到其 FP (Forward Propagation) 的 Computational Graph, 如下:



总结:

为了简介用 Computational Graph 计算 BP, 我们定义一个函数 $z = f(x, y)$, 其中 x, y 都为标量时, 根据 Chain Rule, 其 FP 和 BP 分别如下:

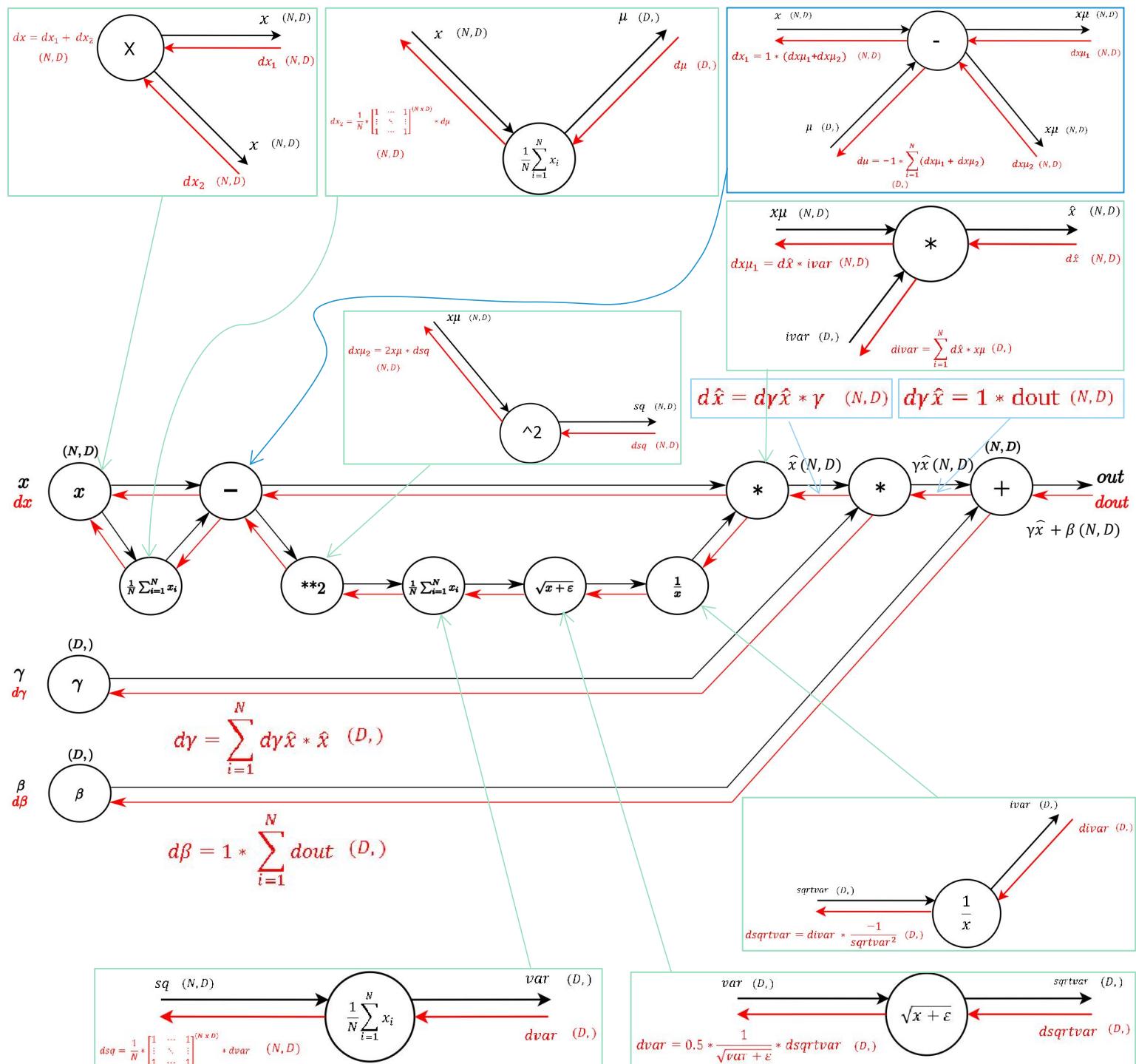


可以看出, 若 $\frac{dL}{dz}$ 已知, 则求 $\frac{dL}{dx}$ 和 $\frac{dL}{dy}$ 分别只需要知道 $\frac{dz}{dx}$ 和 $\frac{dz}{dy}$ 即可。其实就是 BP 的 Computational Graph 描述。

于是, 我们可以利用 Computational Graph 和 BP 来计算增加了 BN Layer 后的 $\frac{dL}{dx}$ 、 $\frac{dL}{dy}$ 、 $\frac{dL}{d\beta}$ 。

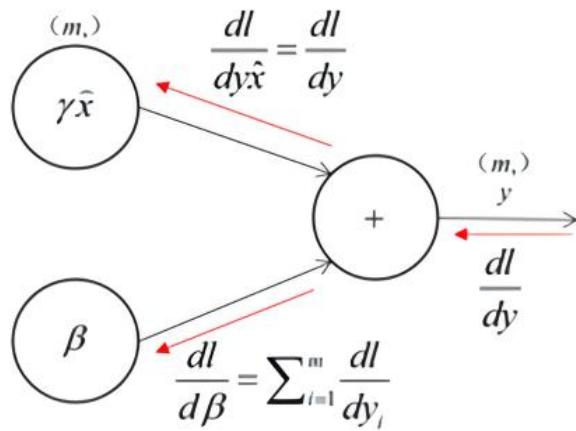
下面的图片组合自: Understanding the backward pass through Batch Normalization Layer

<https://link.zhihu.com/?target=https%3A//kratzert.github.io/2016/02/12/understanding-the-gradient-flow-through-the-batch-normalization-layer.html>

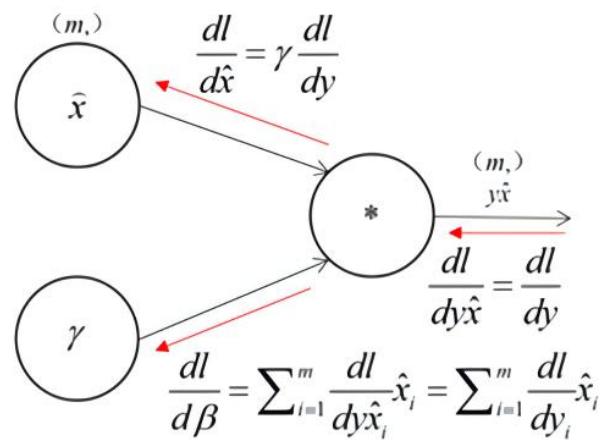


总结:

通过下图，我们可以计算出 $\frac{dL}{d\beta}$



通过下图，我们可以计算出 $\frac{dL}{dy}$



详细可以参考 Understanding the backward pass through Batch Normalization Layer。

最终，我们可以得到 BN 的 BP 公式，如下。然后就可以利用 miniBatch GD 来更新参数了。

$$\begin{aligned}\frac{\partial \ell}{\partial \hat{x}_i} &= \frac{\partial \ell}{\partial y_i} \cdot \gamma \\ \frac{\partial \ell}{\partial \sigma_B^2} &= \sum_{i=1}^m \frac{\partial \ell}{\partial \hat{x}_i} \cdot (x_i - \mu_B) \cdot \frac{-1}{2} (\sigma_B^2 + \epsilon)^{-3/2} \\ \frac{\partial \ell}{\partial \mu_B} &= \left(\sum_{i=1}^m \frac{\partial \ell}{\partial \hat{x}_i} \cdot \frac{-1}{\sqrt{\sigma_B^2 + \epsilon}} \right) + \frac{\partial \ell}{\partial \sigma_B^2} \cdot \frac{\sum_{i=1}^m -2(x_i - \mu_B)}{m} \\ \frac{\partial \ell}{\partial x_i} &= \frac{\partial \ell}{\partial \hat{x}_i} \cdot \frac{1}{\sqrt{\sigma_B^2 + \epsilon}} + \frac{\partial \ell}{\partial \sigma_B^2} \cdot \frac{2(x_i - \mu_B)}{m} + \frac{\partial \ell}{\partial \mu_B} \cdot \frac{1}{m} \\ \frac{\partial \ell}{\partial \gamma} &= \sum_{i=1}^m \frac{\partial \ell}{\partial y_i} \cdot \hat{x}_i \\ \frac{\partial \ell}{\partial \beta} &= \sum_{i=1}^m \frac{\partial \ell}{\partial y_i}\end{aligned}$$

BN 测试

这里需要注意的是，上面 Batch 的概念只可能出现在训练过程中。在测试时，DNN 的参数需要固定。因此，在训练的最后一个 epoch，需要记录下其每个 Batch 的 μ_i 和 σ_i ，并使用其无偏估计作为测试时 DNN 的 μ 和 σ ，即：

$$\begin{aligned} E[x] &\leftarrow E_{\mathcal{B}}[\mu_{\mathcal{B}}] \\ \text{Var}[x] &\leftarrow \frac{m}{m-1} E_{\mathcal{B}}[\sigma_{\mathcal{B}}^2] \end{aligned}$$

BN 的作用

BN 主要的好处其实是减少 DNN 在训练时调参的难度，这里列举几个比较主要的：

- 在一般 DNN 中，如果 Learning Rate 设置的太小，则 DNN 会收敛的很慢；而 Learning Rate 设置的太大，则容易 Gradient Explode 或者 Gradient Vanish。按照 “WHY Batch Normalization ?” 中的分析，增加的 BN Layer 能缓解这种情况，从而能够使用较大的 Learning Rate，加速 GD。
- 作者通过实验说明，在有 BN Layer 的情况下，可以不使用 Dropout，而不带来性能的降低。不过感觉其说明不是很有力，所以实际中适当减少 Dropout 的使用即可，毕竟 Dropout 本身被证明是简单而有效缓解 Overfitting 的一种方法。
- DNN 在控制 Overfitting 时一般都是使用 L2 Weight Decay。同样是用实验，作者表示有了 BN，可以适当减少 L2 Weight Decay 的大小，即 BN 本身也有一定 Regularization 的作用。这一点我觉得可以这么理解，BN Layer 虽然没有直接约束 W ，但是通过 $BN(W\mu + b)$ 其实约束了下一层的 μ ，从而间接约束了 W 。当然，这同样只是感性上的理解而已。

优化器

参考文献:

1. 详解 Deep Learning 的各种优化器 (一) <https://zhuanlan.zhihu.com/p/377141061>
2. 详解 Deep Learning 的各种优化器 (二) <https://zhuanlan.zhihu.com/p/390819660>
3. An overview of gradient descent optimization algorithms <https://link.zhihu.com/?target=https%3A//arxiv.org/pdf/1609.04747>
4. An Introduction to AdaGrad <https://medium.com/konvergen/an-introduction-to-adagrad-f130ae871827>

本文将讲解以下概念：

- Gradient Descent
- Batch Gradient Descent
- Stochastic Gradient Descent (SGD)
- Min-batch Stochastic Gradient Descent (Mini-batch SGD)
- Momentum
- Nesterov accelerated gradient (NAG)
- Adagrad
- Adadelta
- RMSprop
- Adam
- AdaMax
- Nadam
- AMSGrad

梯度下降

在微积分中，函数在点 $f(x)$ 上的导数定义为: $f'(x) = \lim_{x \rightarrow x_0} \frac{f(x) - f(x_0)}{x - x_0}$ ，这在几何上指的是函数 $f(x)$ 在点 x_0 处的切线方向。

当 $f'(x) = 0$ 时得到函数的极值点(临界点)。

但是临界点并不一定是全局最大值或者全局最小值，甚至不是局部的最大值或者局部最小值（如鞍点）。

从 Taylor 级数的角度来看， $f(x)$ 在 x_0 附近的 Taylor 级数（皮亚诺余项）是：

$$f(x) = f(x_0) + f'(x_0)(x - x_0) + \frac{f''(x_0)}{2}(x - x_0)^2 + O(|x - x_0|^3)$$

若 x_0 为临界点，则其满足条件: $f'(x) = 0$ 。

- 当 $f''(x) > 0$ 时，可以得到 x_0 是 $f(x)$ 的局部最小值；
- 当 $f''(x) < 0$ 时，可以得到 x_0 是 $f(x)$ 的局部最大值。

而对于例子 $f(x) = x^3$ 而言，临界点 0 的二阶导数则是 $f''(0) = 0$ ，因此使用上面的方法则无法判断临界点 0 是否是局部极值。

对于多元函数 $f(x) = f(x_1, \dots, x_n)$ 而言，同样可以计算它们的“**导数**”，也就是偏导数和梯度。梯度可以定义为：

$$\nabla f(x) = \left(\frac{\partial f}{\partial x_1}(x), \dots, \frac{\partial f}{\partial x_n}(x) \right)$$

而多元函数 $f(x)$ 在点 x_0 上的 Taylor 级数（皮亚诺余项）为：

$$f(x) = f(x_0) + \nabla f(x_0)(x - x_0) + \frac{1}{2}(x - x_0)^T \mathbf{H}(x - x_0) + O(|x - x_0|^3)$$

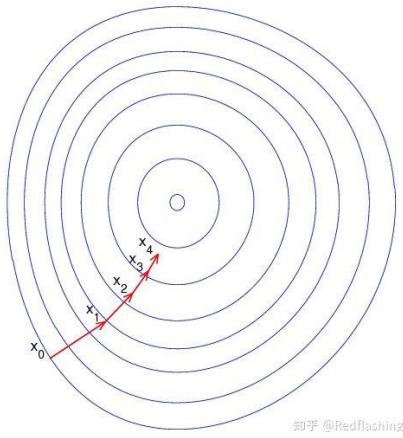
其中 H 表示**黑塞矩阵** (Hessian Matrix)。如果 x_0 为临界点，并且黑塞矩阵为**正定矩阵**时， $f(x)$ 在 x_0 处达到局部极小值。

$$\mathbf{H} = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}$$

梯度下降法基于以下定义：如果实值函数 $f(x)$ 在点 a 处可微且有定义，那么函数 $f(x)$ 在 a 点沿着梯度相反的方向 $-\nabla f(a)$ 下降最多。

因而，如果 $b = a - \gamma \nabla f(a)$ 对于 $\gamma > 0$ 且 $\gamma \rightarrow 0$ 时成立，那么 $f(a) \geq f(b)$ 。值得注意的是**迭代步长 γ 并不是定值**。

故我们可以从函数 f 的局部极小值的**初始估计** x_0 出发，并依次可得到如下序列 (x_0, x_1, x_2, \dots) 使得 $x_{n+1} = x_n - \gamma_n \nabla f(x_n)$ ， $n \geq 0$ 。因此可得到 $f(x_0) \geq f(x_1) \geq f(x_2) \geq \dots$ ，如果顺利的话序列 (x_n) 收敛到期望的局部极小值。



在神经网络中，梯度下降给调整网络参数提供了一种可行的方法。初始权重和偏置可解释为单个向量 θ_0 ，理论上迭代步骤可以用来确定模型的最佳参数 θ^* 。实际上我们试图最小化的函数是根据整个数据集 D 来定义的：

$$J(\theta) = \frac{1}{|D|} \sum_{x \in D} f(x|\theta)$$

其中 $f(x|\theta)$ 表示使用参数向量 θ 时，数据集元素 x 的损失。

总的来说，梯度下降是一种通过更新模型中参数 $\theta \in R^d$ 来最小化 $J(\theta)$ 的方法（通过计算目标函数梯度 $\nabla_\theta J(\theta)$ ，并反向更新参数）。**学习率 η** 决定了我们为了达到（局部）最小区而采取的跨度大小。

总结：

梯度下降变体

梯度下降有 3 种变体，它们在计算目标函数的梯度所用的数据量方面有所差异。根据数据量的大小，我们在参数更新的准确性和执行时间之间进行权衡。

Batch Gradient Descent

Batch Gradient Descent 通过计算参数 θ 关于整个训练集的损失函数的梯度：

$$\theta = \theta - \eta \cdot \nabla_{\theta} J(\theta)$$

但由于我们需要计算整个数据集来执行一次梯度更新，因此 Batch Gradient Descent 可能非常缓慢，并且对于大型数据集（大于内存大小）的训练将会变得十分棘手。此外 Batch Gradient Descent 也不允许我们在线更新模型。

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

以上为 Batch Gradient Descent 的简单代码示例。我们以梯度相反的反向更新参数，学习率 `learning_rate` 决定了我们每次更新速度。对于 Convex Error Surface 而言 Batch Gradient Descent 可以保证收敛到全局最小值，对于 Non-convex Surface 则可以保证收敛到局部最小值。

Stochastic Gradient Descent (SGD)

Sotchastic Gradient Descent，其为每个训练样本 $x^{(i)}$ 和标签 $y^{(i)}$ 执行参数更新：

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

Batch Gradient Descent 对大型数据集有大量的冗余计算，因为它会在每个参数更新前重新计算相似的梯度。而 SGD 针对每个样本进行一次参数更新。由于 SGD 频繁执行更新，且变化很大，这导致目标函数震荡十分剧烈。

当 Batch Gradient Descent 下降收敛到参数某个局部最小值时不再继续收敛。而 SGD 的不稳定性使得有收敛到更好的局部最小值的可能性（收敛的过程也因此变得过分复杂）。已经表明，当缓慢降低学习率时，SGD 会显示与 Batch Gradient Descent 相同的收敛行为，对于非凸优化和凸优化，它们分别收敛到局部最小值和全局最小值。

```
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
```

Mini-batch Stochastic Gradient Descent (Mini-batch SGD)

(在原文中被称之为 Mini-batch Gradient Descent) 它同时兼顾了上述两种方法的优势，针对 n 个训练样本的 mini-batch 计算损失进行参数梯度更新：

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

其优点在于：

- 降低了参数更新的方差，可以更稳定地收敛（与 SGD 相比较）
- 利用深度学习库对常见大小 mini-batch 的矩阵进行高度优化的特性，可非常高效计算出其梯度

常见 mini-batch 大小在 50 至 256 之间，但会因不同的应用场景而有所不同。训练神经网络时，通常选择 min-batch (而当使用 mini-batch 时，通常也使用术语 SGD (即 mini-batch SGD))。

注意：在下文的改进的SGD中，为了简单，我们省略了参数 $x^{(i:i+n)}$ 和 $y^{(i:i+n)}$ 。

以下是大小为 50 的 mini-batch 示例代码：

```
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
```

挑战性

Mini-batch SGD 虽然具有综合优势，但其并不能保证良好的收敛性，其还有一些需要解决的挑战：

- 选择合适的学习率十分困难。学习率太小会导致收敛时间过长，而学习率过大又会阻碍收敛导致损失函数在最小值附近波动甚至发散。
- 学习率表**尝试通过例如调整训练过程中的学习率（例如退火）。即根据预定义的时间表或在各个时期之间的目标函数变化降到阈值以下时降低学习率。但是，这些计划和阈值必须预先定义，因此无法适应数据集的特征。
- 对于参数更新，通常我们采用相同学习率用于所有参数。如果数据集稀疏且模型特性有非常不同的出现频率，我们可能并不想将所有模型特征更新到相同的程度，且对很少出现的特性执行较大的更新。
- 另一个关键挑战在最小化高度非凸损失函数（神经网络中十分常见）如何避免陷入众多的局部最小值。Dauphin 等人认为关键困难在于并不是由于局部最小值，而是在于鞍点。这使得 SGD 很难逃脱，因为在所有维度上梯度都接近于 0。

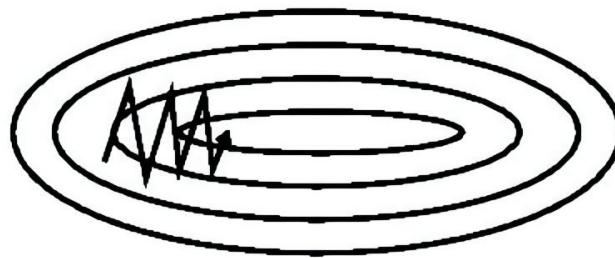
梯度下降优化算法

接下来，我们将讨论一些广泛使用的算法来应对上述挑战。但我们并不会讨论在实际中无法应用于高维数据集的算法，例如二阶算法（如牛顿法）。

Momentum

我们将最小化目标函数的任务比作从坡顶到坡底的过程。

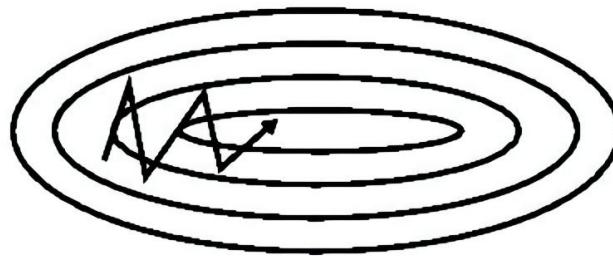
SGD 难以在沟壑中有效移动，在局部上有些维度上的弯曲程度要比正确下降维度要陡峭得多，这在局部最优的情况下很常见。这些情况下，SGD 会在山坡上震荡，沿着底部最优的方向缓慢下降，如下图所示。



Momentum 通过将上一个步骤的权重更新向量乘以 momentum 参数 γ 加在当前权重更新向量上达到了在相关方向上加速 SGD 并抑制振荡。

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

通常 momentum 参数 γ 设置为 0.9。与 SGD 相比较，Momentum 轨迹如下图所示：



本质上，当使用 Momentum 时，我们可以将该过程比作将球推下山坡的过程。球在下坡滚动时会积累动量，并且在途中越来越快（如果存在空气阻力，即 $\gamma < 1$ ，直到达到最终速度）。我们的参数更新也发生了同样的事情：动量参数对于梯度相同方向进行促进，而在梯度改变方向进行抑制。最终，我们获得了更快的收敛并减少了振荡。

上面红色字这句话陈锐平个人的理解：因为引入了动量（由速度 v 表示，同时也引入了动量参数 γ ），对于当前所求梯度与动量方向相同的，当前梯度会促进动量增加，进而增加参数 θ 的更新幅度（因为 v_t 比 v_{t-1} 更大了）。对于当前所求梯度与动量方向相反的，当前梯度会导致动量减小，进而减少参数 θ 的更新幅度（因为 v_t 比 v_{t-1} 更小了）。

Nesterov Momentum / Nesterov accelerated gradient (NAG)

然而，从山上滚下来的球盲目地跟随斜坡是不尽如人意的。我们希望有一个更聪明的球，能够知道在坡度变大之前放慢速度。

NAG 是一种能够给动量上述能力的方法。我们知道我们使用动量项 γv_{t-1} 跟新参数 θ (即 $\gamma v_{t-1} + \eta \nabla_\theta J(\theta)$)。因此，计算 $\theta - \gamma v_{t-1}$ 可得出参数下一个位置的近似值 (还缺少梯度)。通过计算关于参数未来的近似位置的梯度，而不是关于当前的参数 θ 的梯度，我们可以高效的求解：

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

同样，我们设置动量项 γ 大约为 0.9。

有关 NAG 更多直观的解释，可以参考: <https://link.zhihu.com/?target=https%3A//cs231n.github.io/neural-networks-3/>

Adagrad

Adagrad 是一种基于梯度的优化算法，它具体是通过学习率适应参数：低学习率执行较小的更新（与频繁出现的特征相关联的参数），高学习率执行较大更新（与不常见特征相关联的参数）。因此，它非常适合处理稀疏数据。

- Dean 等人发现 Adagrad 可以极大提高 SGD 的鲁棒性，并将其用于训练 Google 的大规模神经网络，包括识别 Youtube 视频中的猫咪。
- Pennington 等人使用 Adagrad 来训练 GloVe 词嵌入，因为不常见的词需要的更新跨度比频繁出现的词大得多。

之前的算法我们每次都对 θ 的所有参数进行了更新，对每一个 θ_i 都使用了相同的学习率 η 。由于 Adagrad 再每个时刻 t 对每个参数 θ_i 使用了不同的学习率，我们首先展示 Adagrad 的每个参数更新，然后我们将其量化。

为了方便表示，我们使用 g_t 表示时刻 t 处的梯度。 $g_{t,i}$ 表示目标函数在时刻 t 关于 θ_i 的偏导：

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

参数 θ_i 在时刻 t 的 SGD 更新过程变为：

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

基于上述的更新规则，在 t 时刻，对于 θ_i 计算过的历史梯度，Adagrad 修正对每一个 θ_i 的学习率：

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

$$G_t^{(i,i)} = \sum_{\tau}^t (g_{\tau}^{(i)})^2$$

其中：

- $G_t \in \mathbb{R}^{d \times d}$ 是一个**对角矩阵**，对角线上的元素 (i, i) 是直到 t 时刻为止，所有关于 θ_i 的梯度平方和。
- ϵ 是一个平滑项，可以避免分母为零（通常 ϵ 在 10^{-8} 的数量级上）。

Duchi 等人将该矩阵作为包含所有先前梯度的**外积的全矩阵** G_t 的替代，因为计算全矩阵的平方根是不切实际的，尤其是在非常高的维度上（即使对于中等参数量，矩阵的均方根的计算都是不切实际的）。从另一方面看，计算平方根和对角线 G_t 的倒数却能轻易实现。

为了更直观展示上式，我们将其展开：

$$\begin{bmatrix} \theta_{t+1}^{(1)} \\ \theta_{t+1}^{(2)} \\ \vdots \\ \theta_{t+1}^{(m)} \end{bmatrix} = \begin{bmatrix} \theta_t^{(1)} \\ \theta_t^{(2)} \\ \vdots \\ \theta_t^{(m)} \end{bmatrix} - \eta \left(\begin{bmatrix} \varepsilon & 0 & \cdots & 0 \\ 0 & \varepsilon & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \varepsilon \end{bmatrix} + \begin{bmatrix} G_t^{(1,1)} & 0 & \cdots & 0 \\ 0 & G_t^{(2,2)} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & G_t^{(m,m)} \end{bmatrix} \right)^{-1/2} \begin{bmatrix} g_t^{(1)} \\ g_t^{(2)} \\ \vdots \\ g_t^{(m)} \end{bmatrix} \quad (4)$$

进一步地，我们简化学习率部分：

$$\begin{aligned} & \eta \left(\begin{bmatrix} \varepsilon & 0 & \cdots & 0 \\ 0 & \varepsilon & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \varepsilon \end{bmatrix} + \begin{bmatrix} G_t^{(1,1)} & 0 & \cdots & 0 \\ 0 & G_t^{(2,2)} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & G_t^{(m,m)} \end{bmatrix} \right)^{-1/2} \\ &= \eta \begin{bmatrix} \varepsilon + G_t^{(1,1)} & 0 & \cdots & 0 \\ 0 & \varepsilon + G_t^{(2,2)} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \varepsilon + G_t^{(m,m)} \end{bmatrix}^{-1/2} \\ &= \eta \begin{bmatrix} \frac{1}{\sqrt{\varepsilon + G_t^{(1,1)}}} & 0 & \cdots & 0 \\ 0 & \frac{1}{\sqrt{\varepsilon + G_t^{(2,2)}}} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \frac{1}{\sqrt{\varepsilon + G_t^{(m,m)}}} \end{bmatrix} \\ &= \begin{bmatrix} \frac{\eta}{\sqrt{\varepsilon + G_t^{(1,1)}}} & 0 & \cdots & 0 \\ 0 & \frac{\eta}{\sqrt{\varepsilon + G_t^{(2,2)}}} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \frac{\eta}{\sqrt{\varepsilon + G_t^{(m,m)}}} \end{bmatrix}, \end{aligned}$$

知乎 @Redflashing

总结：

最后得到：

$$\begin{bmatrix} \theta_{t+1}^{(1)} \\ \theta_{t+1}^{(2)} \\ \vdots \\ \theta_{t+1}^{(m)} \end{bmatrix} = \begin{bmatrix} \theta_t^{(1)} \\ \theta_t^{(2)} \\ \vdots \\ \theta_t^{(m)} \end{bmatrix} - \begin{bmatrix} \frac{\eta}{\sqrt{\varepsilon + G_t^{(1,1)}}} g_t^{(1)} \\ \frac{\eta}{\sqrt{\varepsilon + G_t^{(2,2)}}} g_t^{(2)} \\ \vdots \\ \frac{\eta}{\sqrt{\varepsilon + G_t^{(m,m)}}} g_t^{(m)} \end{bmatrix} \quad (6)$$

知乎 @RedFlashing

我们最后可以这样表示：

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

有趣的是，如果没有平方根运算，算法的性能会差很多。

Adagrad算法的一个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**。

Adagrad的一个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。**接下来的算法旨在解决这个不足。**

总结：

Adadelta

Adadelta 是 Adagrad 的一种扩展算法，以处理 Adagrad 学习率单调递减的问题。不是计算所有梯度平方，Adadelta 将积累过去梯度的窗口大小限制为一个固定值 ω 。

在 Adadelta 中，无需存储先前的 ω 个平方梯度，而是将梯度的平方递归地表示所有历史梯度平方的均值。在 t 时刻的均值 $E[g^2]_t$ 只取决于先前的均值和当前的梯度（分量 γ 类似于动量项）：

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

我们将 γ 设置成与动量项相似的值，即为 0.9 左右。为了简单起见，我们利用参数更新向量 $\Delta\theta_t$ 重新表示 SGD 的更新过程：

$$\begin{aligned}\Delta\theta_t &= -\eta \cdot g_{t,i} \\ \theta_{t+1} &= \theta_t + \Delta\theta_t\end{aligned}$$

Adagrad 参数更新向量为：

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

现在我们简单将对角矩阵 G_t 替换为历史梯度均值 $E[g^2]_t$ ：

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

由于分母只是梯度的均方根（RMS）误差，我们可以简写为：

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

但作者指出，此更新（以及 SGD、Momentum 或者 Adagrad）中的单位不匹配，即更新量与参数具有相同的假设单位。为了实现这个要求，作者首次定义了另一个指数衰减均值，这次不是梯度平方，而是参数的平方更新：

$$E[\Delta\theta^2]_t = \gamma E[\Delta\theta^2]_{t-1} + (1 - \gamma) \Delta\theta_t^2$$

因此，参数更新的均方根误差为：

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

由于 $RMS[\Delta\theta]_t$ 是未知的，我们利用参数的均方根误差来近似更新。

利用 $RMS[\Delta\theta]_{t-1}$ 替换先前的更新规则中的学习率 η ，最终得到 Adadelta 的更新规则：

$$\begin{aligned}\Delta\theta_t &= -\frac{RMS[\Delta\theta]_{t-1}}{RMS[g]_t} g_t \\ \theta_{t+1} &= \theta_t + \Delta\theta_t\end{aligned}$$

使用 Adadelta，我们甚至不需要设置默认学习率，因为它已经从更新规则中删除。

总结：

RMSprop

RMSprop 是一个未被发表的自适应学习率算法，该算法由 Geoff Hinton 提出。

RMSprop 和 Adadelta 在相同的时间内分别独立提出，均是为了应对 Adagrad 的急速下降的学习率的问题。实际上，RMSprop 是 Adadelta 的第一个更新向量的特例：

$$E[g^2]_t = 0.9E[g^2]_{t-1} + 0.1g_t^2$$

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

同样，RMSprop 将学习率分解成一个平方梯度的指数衰减的平均。Hinton 建议将 γ 设置为 0.9，对于学习率 η 的一个合适的默认值为 0.001。

Adam

自适应矩估计（Adaptive Moment Estimation, Adam）是另一种每个参数的自适应学习率的方法。除了类似于 Adadelta 和 RMSprop 存储一个指数衰减的历史平方梯度的平均值 v_t ，Adam 同时还保存了一个历史梯度的指数衰减均值 m_t ，类似于动量。动量可以看成一个从斜坡上跑下来的球，而 Adam 的行为就像一个带有摩擦力的重球，因此它更喜欢误差表面上更为平缓的最小值。

我们计算过去和过去平方梯度的衰减均值 m_t 和 v_t 分别如下：

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

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

m_t 和 v_t 分别对应梯度的一阶矩（均值）和二阶矩（非确定的方差）的估计，正如该算法的名称。当 m_t 和 v_t 初始化为 0 向量时，Adam 的作者发现到它们都偏向于 0，尤其是在初始化步骤和当衰减率很小的时候（例如 β_1 和 β_2 趋向于 1）。

通过计算偏差校正的一阶矩和二阶估计来抵消偏差：

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \quad \hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

正如我们在 Adadelta 和 RMSprop 中看到那样，他们利用上述公式更新参数，由此生成了 Adam 的更新规则：

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

作者建议 β_1 取默认值为 0.9， β_2 为 0.999， ϵ 为 10^{-8} 。他们从经验上表明 Adam 在实际中表现很好，同时，与其他的自适应学习算法相比，其更有优势。

总结：

AdaMax

AdaMax 是 Adam 的一种变体，此方法对学习率的上限提供了一个更简单的范围。

在 Adam 中，单个权重的更新规则是将其梯度与当前梯度 $|g_t|^2$ 和过去梯度的 l_2 范数（标量）成反比例缩放：

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

因此，我们可以推导出 l_p 范数（的版本）：

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

虽然这样的变体会因为 p 值较大而在数值上变得不稳定（这也是为什么 l_1 和 l_2 范数在实际场景中广泛应用的原因），然而，当 $p \rightarrow \infty$ 时， l_∞ 表现出极其稳定的特性。由此，AdaMax 的作者 (Kingma and Ba, 2015) 展示了由 l_∞ 得到的 v_t 拥有更好的稳定性。为了与 Adam 相区分，我们用 u_t 表示无穷范数约束 v_t ：

$$\begin{aligned} u_t &= \beta_2^\infty v_{t-1} + (1 - \beta_2^\infty) |g_t|^\infty \\ &= \max(\beta_2 \cdot v_{t-1}, |g_t|) \end{aligned}$$

我们现在将上式插入到 Adam 更新公式：将 $\sqrt{\hat{v}_t} + \epsilon$ 替换为 u_t ，得到 AdaMax 的更新规则：

$$\theta_{t+1} = \theta_t - \frac{\eta}{u_t} \hat{m}_t$$

注意到 u_t 依赖于 \max 运算，所以 AdaMax 不会像 Adam 中 m_t 和 v_t 趋向于 0，这也是为什么我们不需要去计算偏差校正。比较合适的参数设置： $\eta = 0.002$, $\beta_1 = 0.9$, $\beta_2 = 0.999$

Nadam

正如我们所见，Adam 可以被视为 RMSprop 和 momentum 的结合产物：RMSprop 提供历史平方梯度的指数衰减均值 v_t ，而 Momentum 提供了历史梯度的指数衰减均值 m_t 。

与此同时按照前文叙述 NAG (Nesterov accelerated gradient) 是优于 Momentum 的。

Nadm (Nesterov-accelerated Adaptive Moment Estimation) 就算结合了 Adam 和 NAG 的产物。为了将 NAG 融入 Adam 中，我们需要修改其 Momentum 部分 m_t 。

第一步，我们复习一下 Momentum 的更新规则：

$$g_t = \nabla_{\theta_t} J(\theta_t)$$

$$m_t = \gamma m_{t-1} + \eta g_t$$

$$\theta_{t+1} = \theta_t - m_t$$

其中， J 是目标函数， γ 是动量的衰减项， η 是步长（即为学习率），展开第三个等式可得：

$$\theta_{t+1} = \theta_t - (\gamma m_{t-1} + \eta g_t)$$

以上再次说明了当前动量的前一个动量上前进了一步，也在当前梯度方向上前进了一步。

NAG 在计算梯度之前用动量步骤更新参数使得其在梯度方向上表现更为准确。我们只需要修改 NAG 中的梯度 g_t ：

$$g_t = \nabla_{\theta_t} J(\theta_t - \gamma m_{t-1})$$

$$m_t = \gamma m_{t-1} + \eta g_t$$

$$\theta_{t+1} = \theta_t - m_t$$

Dozat 建议按照如下步骤修改 NAG：与应用两次动量步骤不同（一次更新梯度 g_t ，一次跟新参数 θ_{t+1} ），我们直接更新参数：

$$g_t = \nabla_{\theta_t} J(\theta_t)$$

$$m_t = \gamma m_{t-1} + \eta g_t$$

$$\theta_{t+1} = \theta_t - (\gamma m_t + \eta g_t)$$

值得注意的是不同于上面扩展动量梯度更新规则方程利用动量 m_{t-1} ，作者使用了当前动量向量 m_t 去梯度更新。为了加上 Nesterov momentum 到 Adam 中去，作者类似地用当前动量向量替换之前的动量向量。

总结：

首先，我们将 Adam 梯度更新规则列出来（注意我们不需要修改 \hat{v}_t ）：

$$\left. \begin{aligned} m_t &= \beta_1 m_{t-1} + (1 - \beta_1) g_t \\ \hat{m}_t &= \frac{m_t}{1 - \beta_1^t} \\ \theta_{t+1} &= \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t \end{aligned} \right\} \quad \theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \left(\frac{\beta_1 m_{t-1}}{1 - \beta_1^t} + \frac{(1 - \beta_1) g_t}{1 - \beta_1^t} \right)$$

注意到 $\frac{\beta_1 m_{t-1}}{1 - \beta_1^t}$ 是上一步的动量向量偏差修正估计，对此作者进行用 \hat{m}_{t-1} 如下替换：

\downarrow

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \left(\beta_1 \hat{m}_{t-1} + \frac{(1 - \beta_1) g_t}{1 - \beta_1^t} \right)$$

为了方便起见，作者忽略了分母是 $1 - \beta_1^t$ 而不是 $1 - \beta_1^{t-1}$ 。这个方程看起来与我们上面的扩展动量更新规则非常相似。我们现在可以在此之上添加 Nesterov momentum，只需要简单使用当前动量向量的偏置校正估计 \hat{m}_t 替换上一步衰减动量的偏置估计校正 \hat{m}_{t-1} ，这样我们得到 Nadam 的梯度更新规则：

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \left(\beta_1 \hat{m}_t + \frac{(1 - \beta_1) g_t}{1 - \beta_1^t} \right)$$

AMSGrad

当自适应学习率算法成为训练神经网络的标准，从业者也注意到例如目标检测或者机器翻译并不是总能收敛到最优，并被带有 momentum 的 SGD 超越。

Reddi 等人将这个问题形式化，并且指出过去平方梯度的指数移动平均值是自适应学习率算法泛化行为不佳的原因。回顾一下，**指数均值的引入原因：为了防止学习率随着训练变得无穷小**，这是 Adagrad 算法的关键缺陷。然而这种梯度的**短期记忆**在其他场景中成为障碍。

在 Adam 收敛到次优解的设置中，就已经注意到某些 minibatch 提供了大且丰富的梯度，但这些 minibatch 的出现十分罕见，指数平均减弱了它们的影响并导致了收敛性差的问题。作者提供了一个简单的凸优化问题例子中可以观察到 Adam 有着相同的行为。

为了修复这种行为，作者提出了一种新的算法——AMSGrad，用历史平方梯度 v_t 的最大值而不是指数均值去更新参数。 v_t 与上面 Adam 中定义相同：

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

不同于直接使用 v_t （或者偏置校正的 \hat{v}_t ），作者采用 \hat{v}_{t-1} 和 v_t 的最大值：

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

这样，AMSGrad 算法的结果就不是一个递增的值，这避免了 Adam 所遇到的问题。为了简单起见，作者也删除了 Adam 中的 debiasing 步骤。完整的带偏置校正估计的 AMSGad 更新规则如下：

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

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

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

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

相比于 Adam 作者在小数据集以及 CIFAR-10 上观察到更好的表现。然而，其他的 **实验** (<https://fdlm.github.io/post/amsgrad/>) 中也展现了相同的或者更坏的表现（相比于 Adam）。AMSGad 在实践中是否能够始终超越 Adam，还有待观察。

其他优化算法

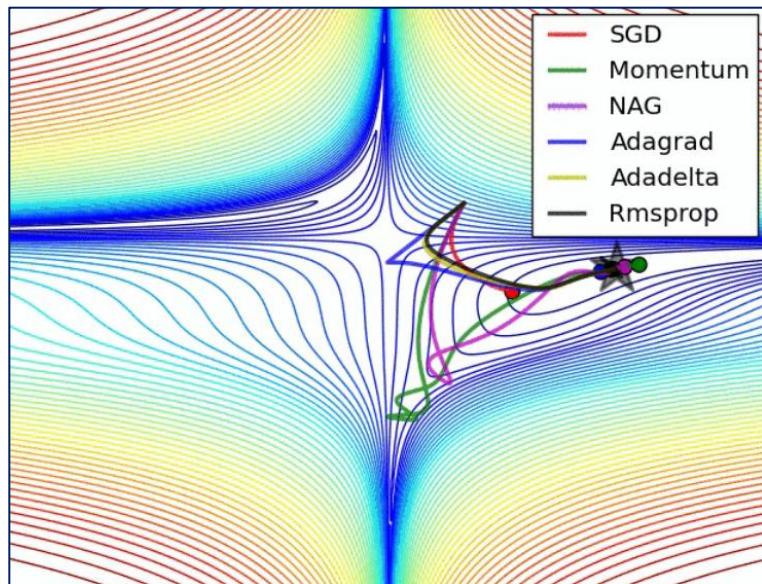
在 AMSGad 之后出现了很多其他的优化器，包括：

- **AdamW**：修复了 Adam 的权重衰减。
- **QHAdam**：在更新权重时将动量项与当前梯度解耦，在更新权重时将均方梯度项与当前平方梯度解耦。
- **AggMo**：结合了多个动量项 γ 。

更多更新的新优化器可以参考：An updated overview of recent gradient descent algorithms – John Chen – ML at Rice University (johnchenresearch.github.io) <https://link.zhihu.com/?target=https%3A//johnchenresearch.github.io/demon/>

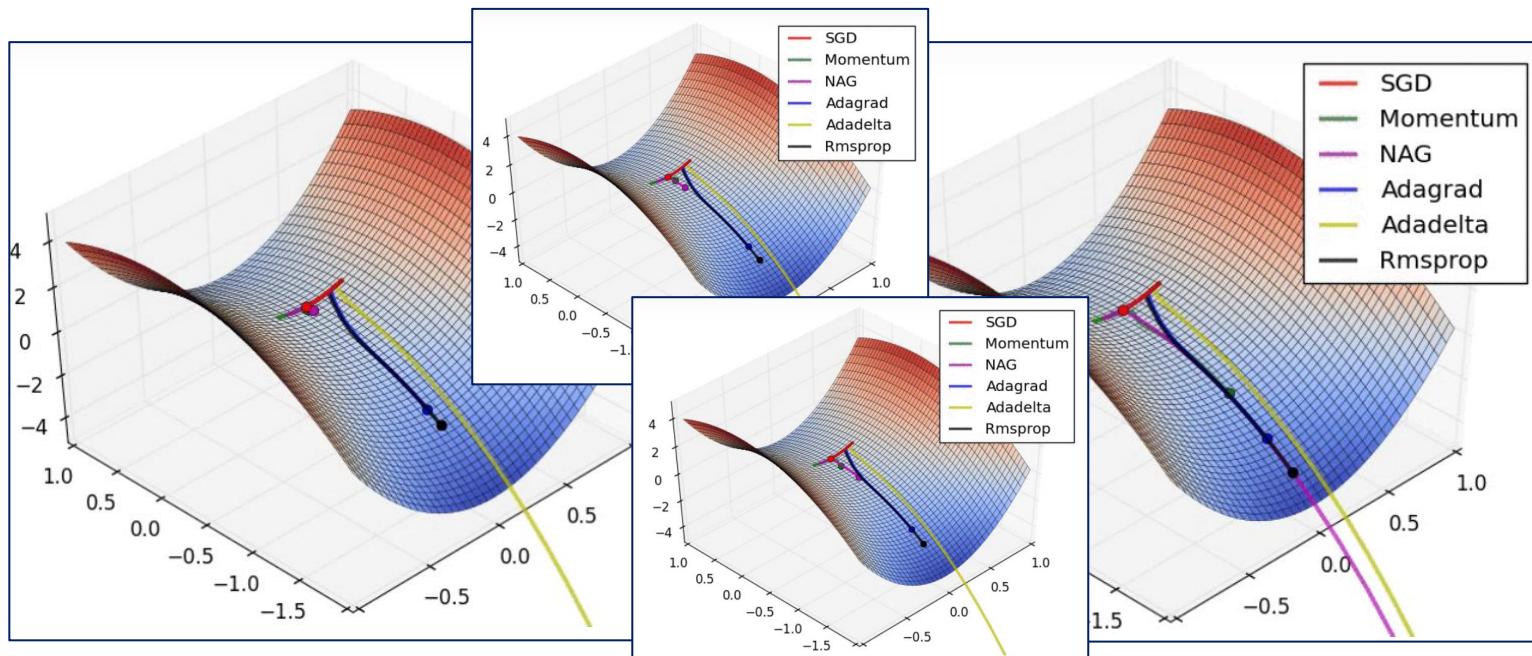
总结：

优化器算法可视化



从左图我们能够看到这些优化器在损失平面的等高线图(Beale 函数) 随着时间的行为。注意到：

- Adagrad, Adadelta, 以及 RMSprop 几乎立即朝着正确的方向并以同样的速度收敛。
- 与此同时 Momentum 和 NAG 偏离方向，行为类似球从山上滚下来的场景。然而相比于 Momentum , NAG 能够迅速纠正方向。



第二张图展示了优化算法在鞍点的行为，一个维度具有正斜率，而另一个维度具有负斜率的点，正如之前提到的，这对 SGD 造成了困难。注意到：

- SGD、Momentum 和 NAG 发现很难打破对称性，尽管后者最终设法逃离了鞍点。
- 而 Adagrad、RMSprop 和 Adadelta 迅速下降到负斜率。

正如我们所看到的，自适应学习率方法，即 Adagrad、Adadelta、RMSprop 和 Adam 是最合适的，并且为这些场景提供了最好的收敛性。

如何挑选优化算法？

那么，我们现在应该使用哪个优化器？

如果的输入数据稀疏，那么我们可能会使用其中一种自适应学习率方法获得最佳结果。另一个好处是我们不需要调整学习率，但可能会使用默认值获得最佳结果。

- 总的来说，RMSprop 是 Adagrad 的扩展，改进其学习率急剧下降的问题。这和 Adadelta 是一致的，只是 Adadelta 在分子更新规则中使用参数均方根进行更新。
- Adam 最终结合偏置校正和 momentum 到 RMSprop。

到目前为止，RMSprop、Adadelta 以及 Adam 是非常相似的算法，在一些相近的场景下都表现十分优异。Kingma 等人表明随着梯度变得更稀疏，它的偏差校正有助于 Adam 在优化结束时略微优于 RMSprop。就此而言，Adam 可能是最佳的整体选择。

有趣的是，最近的许多论文使用朴素 SGD 没有 Momentum 且使用简单的学习率表。正如已经显示的那样，SGD 通常可以找到最小值，但它可能比使用某些优化器花费的时间长得多，更依赖于稳健的初始化和模拟退火表，并且可能会卡在鞍点而不是局部最小值。

因此，如果您关心快速收敛并训练深度或复杂的神经网络，就应该选择一种自适应学习率方法。

并行以及分布 SGD

鉴于大规模数据解决方案的普遍性和低成本集群的可用性，分布式 SGD 以进一步加速是一个明确可行的选择。

SGD 本质上是连续的：一步一步朝着最低值前进。它可以提供良好的收敛性，但在超大型数据集上可能收敛十分缓慢。相比之下，异步 SGD 速度更快，但 worker 之间的不佳的通信会导致收敛效果差。此外，作者还可以在一台机器上并行化 SGD，而无需大型计算集群。

以下是为优化并行化和分布式 SGD 提出的算法和架构。

Hogwild!

Niu 等人介绍了一种更新方案叫做 Hogwild!。这允许在 CPU 上并行执行 SGD 更新。允许处理器在不锁定参数的情况下访问共享内存。**这仅在输入数据稀疏时才有效**，因为每次更新只会修改所有参数的一小部分。他们表明，在这种情况下，更新方案几乎达到了最佳收敛速度，因为处理器不太可能覆盖有用信息。

Downpour SGD

Downpour SGD 是 Dean 等人使用的 SGD 的异步变体。在 Google 的 DistBelief 框架（TensorFlow 的前身）中。它在训练数据的子集上并行运行模型的多个副本。这些模型将它们的更新发送到一个参数服务器，该服务器分布在许多机器上。每台机器负责存储和更新模型参数的一小部分。但是，由于副本不相互通信，例如通过共享权重或更新，它们的参数不断面临发散的风险，阻碍收敛。

Delay-tolerant Algorithms for SGD

McMahan 和 Streeter 通过开发不仅适应过去梯度而且适应更新延迟的延迟容忍算法，将 AdaGrad 扩展到并行设置。这已被证明在实践中运作良好。

TensorFlow

TF 是谷歌开源的用于实施和部署大规模机器学习模型的框架。它基于他们在 DistBelief 方面的经验，并且已经在内部用于在大量移动设备和大规模分布式系统上执行计算。对于分布式执行，计算图被拆分为每个设备的子图，并使用发送/接收节点对进行通信。

Elastic Averaging SGD

zhang 等人。提出了弹性平均 SGD (EASGD)，它将异步 SGD 的 worker 的参数与弹性力联系起来，即参数服务器存储的中心变量。这允许局部变量从中心变量进一步波动，这在理论上允许对参数空间进行更多探索。他们凭经验表明，这种增加的探索能力可以通过寻找新的局部最优来提高性能。

优化 SGD 的其他策略

最后，作者介绍了可以与前面提到的任何算法一起使用的其他策略，以进一步提高 SGD 的性能。

An overview of gradient descent optimization algorithms

<https://link.zhihu.com/?target=https%3A//arxiv.org/pdf/1609.04747>

混洗和课程式学习

通常，我们希望避免以有意义的顺序向我们的模型提供训练示例，因为这可能会使优化算法产生偏差。因此，在每个 epoch 之后打乱训练数据通常是一个好办法。

另一方面，对于我们旨在逐步解决更难问题的某些情况，以有意义的顺序提供训练示例实际上可能会提高性能和更好的收敛性。建立这种有意义的秩序的方法称为课程式学习。

Zaremba 和 Sutskever 只能训练 LSTM 来评估使用课程学习的简单程序，并表明组合或混合策略（通过对样本排序增加训练难度）比通过原始策略更好。

Batch Normalization

为了便于学习，我们通常通过用零均值和单位方差初始化参数的初始值来规范化参数的初始值。随着训练的进行和我们在不同程度上更新参数，我们失去了这种归一化，这会减慢训练并随着网络变得更深而放大变化。

Batch Normalization 为每个小批量重新建立这些标准化，并且更改也通过操作进行反向传播。通过将归一化作为模型架构的一部分，我们能够使用更高的学习率并且更少关注初始化参数。Batch Normalization 还充当正则化器，减少（有时甚至消除）对 Dropout 的需求。

提前停止训练

根据 Geoff Hinton 的说法：“Early stopping (is) beautiful free lunch” (NIPS 2015 Tutorial slides, slide 63)。因此，我们应该在训练期间始终查看测试集上的误差，如果测试集的误差没有得到足够的改善，则及时停止。

<https://link.zhihu.com/?target=http%3A//www.iro.umontreal.ca/~bengioy/talks/DL-Tutorial-NIPS2015.pdf>

梯度噪声

Neelakantan 等人。添加遵循高斯分布的噪声 $N(0, \sigma^2)$ 到每一次梯度更新：

$$g_{t,i} = g_{t,i} + N(0, \sigma_t^2)$$

他们根据以下规则对方差进行模拟退火：

$$\sigma_t^2 = \frac{\eta}{(1+t)^\gamma}$$

他们表明，添加这种噪声会使网络对不良初始化更加鲁棒，并有助于训练特别深且复杂的网络。他们推测增加的噪声使模型有更多机会逃脱并找到新的局部最小值，这对于越深的模型越频繁。

总结：

优化器 (补充1)

参考文献:

1. 深度学习中的优化算法 <https://zhuanlan.zhihu.com/p/43506482>

多元函数的梯度和 Taylor 级数

对于多元函数 $f(\mathbf{x}) = f(x_1, \dots, x_n)$ 而言，同样可以计算它们的“导数”，也就是偏导数和梯度，i.e. 它的梯度可以定义为：

$$\nabla f(\mathbf{x}) = \left(\frac{\partial f}{\partial x_1}(\mathbf{x}), \dots, \frac{\partial f}{\partial x_n}(\mathbf{x}) \right).$$

而多元函数 $f(\mathbf{x})$ 在点 \mathbf{x}_0 上的 Taylor 级数是：

$$f(\mathbf{x}) = f(\mathbf{x}_0) + \nabla f(\mathbf{x}_0)(\mathbf{x} - \mathbf{x}_0) + \frac{1}{2} \underbrace{(\mathbf{x} - \mathbf{x}_0)^T}_{1 \times N} \underbrace{H(\mathbf{x} - \mathbf{x}_0)}_{N \times N} \underbrace{(\mathbf{x} - \mathbf{x}_0)}_{N \times 1} + O(|\mathbf{x} - \mathbf{x}_0|^3),$$

其中 H 表示 Hessian 矩阵。如果 x_0 是临界点，并且 Hessian 矩阵是正定矩阵的时候， $f(\mathbf{x})$ 在 x_0 处达到局部极小值。

从数学上的角度来看，梯度的方向是函数增长速度最快的方向，那么梯度的反方向就是函数减少最快的方向。那么：

- 如果想计算一个函数的最小值，可以使用**梯度下降法**的思想来做。
- 如果要计算函数的最大值，沿着梯度的方向前进即可。（**梯度上升法**）

随机梯度下降法也有着自身的局限性。**针对很多高维非凸函数而言，鞍点的数量其实远远大于局部极小值（极大值）的数量。因此，如何快速的逃离鞍点就是深度学习中大家所研究的问题之一。**

总结：

优化器 (补充2)

参考文献:

- 机器学习各优化算法的简单总结 <https://zhuanlan.zhihu.com/p/25572077>

Momentum 改进自 SGD，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。

Nestrov Momentum

Nestrov Momentum的意义在于，既然下一次一定会更新 βd_{i-1} ，那么求梯度的时候就可以用提前位置的梯度 $g(\theta_{i-1} + \beta d_{i-1})$ ，则

$$\begin{aligned}d_i &= \beta d_{i-1} - \lambda g(\theta_{i-1} + \beta d_{i-1}) \\ \theta_i &= \theta_{i-1} + d_i\end{aligned}$$

实验中，一般用Nestrov Momentum比较多，收敛比Momentum要更快一些。

Adagrad

$$\begin{aligned}c_i &= c_{i-1} + g^2(\theta_{i-1}) \\ \theta_i &= \theta_{i-1} - \frac{\lambda}{\sqrt{c_i + \epsilon}} g(\theta_{i-1})\end{aligned}$$

其中， c_i 是一个cache，保存了各位置梯度的平方， ϵ 一般取值为 $10^{-4} \sim 10^{-8}$ ，为了防止分母取零。可以看出，Adagrad不需要手动调节学习率 λ ，因为整体来看，学习率为 $\frac{\lambda}{\sqrt{c_i + \epsilon}}$ ，它会根据 c_i 的大小发生自适应的变化（不同位置上变化的幅度不同）。但是因为 $g^2(\theta_{i-1})$ 一直是正数，所以总的来说，学习率也一直是在变小，最后可能会导致参数无法更新。

Rmsprop

$$\begin{aligned}c_i &= \gamma c_{i-1} + (1 - \gamma) g^2(\theta_{i-1}) \\ \theta_i &= \theta_{i-1} - \frac{\lambda}{\sqrt{c_i + \epsilon}} g(\theta_{i-1})\end{aligned}$$

可以看出，Rmsprop与Adagrad类似，只不过cache的计算略微复杂一些，利用了一个衰减因子 γ ，这样可以使得 c_i 并不是一直处于增大的情况，可以解决Adagrad学习率迅速减小的问题。其中，衰减因子 γ 通常取值为 [0.9, 0.99, 0.999]。

Adadelta

$$\begin{aligned}c_i &= \gamma c_{i-1} + (1 - \gamma) g^2(\theta_{i-1}) \\ d_i &= -\frac{\sqrt{\Delta_{i-1} + \epsilon}}{\sqrt{c_i + \epsilon}} g(\theta_{i-1}) \\ \theta_i &= \theta_{i-1} + d_i \\ \Delta_i &= \gamma \Delta_{i-1} + (1 - \gamma) d_i^2\end{aligned}$$

Adadelta与Rmsprop类似，但是连初始的学习速率 λ 都不用设置。

Adam

$$m_i = \beta_1 m_{i-1} + (1 - \beta_1) g(\theta_{i-1})$$

$$v_i = \beta_2 v_{i-1} + (1 - \beta_2) g^2(\theta_{i-1})$$

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

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

$$\theta_i = \theta_{i-1} - \frac{\lambda}{\sqrt{\hat{v}_i + \epsilon}} \hat{m}_i$$

与Rmsprop类似，Adam除了利用一个衰减因子 β_2 计算cache以外，还类似Momentum，利用了上一次的参数更新方向，所以可以说Adam是带Momentum的Rmsprop。在参数更新的最初几步中，由于 m_i 与 v_i 是初始化为0的，为了防止最初几步的更新向0偏差，Adam利用 β_i 的 t 次幂来修正这种偏差（ t 每次更新加1）。其中，通常 β_1 取值为0.9， β_2 取值为0.999， ϵ 取值为 10^{-8} 。

牛顿法与拟牛顿法

牛顿法

牛顿法和梯度下降法类似，也是求解无约束最优化问题的方法，收敛速度较快（考虑到了二阶导数的信息）。

$$d_i = g(\theta_{i-1})$$

$$\theta_i = \theta_{i-1} - \lambda H_{i-1}^{-1} d_i$$

其中， H_{i-1} 是优化目标函数 L 在第 $i-1$ 步的海森矩阵。牛顿法的缺点就是 H^{-1} 计算比较复杂，因此有其他改进的方法，例如拟牛顿法。

拟牛顿法

拟牛顿法用一个矩阵 G 来近似代替 H^{-1} （或 B 来代替 H ），其中 G 满足拟牛顿条件：

$$G_{i+1} y_i = \delta_i \quad (B_{i+1} \delta_i = y_i)$$

其中 $y_i = g(\theta_{i+1}) - g(\theta_i)$ ， $\delta_i = x_{i+1} - x_i$ 。因此按照拟牛顿条件，每次只需更新 G_{i+1} （或 B_{i+1} ）即可，使得 $G_{i+1} = G_i + \Delta G_i$ 。

牛顿法有多种的具体实现，其中DFP算法选择更新 G ，BFGS选择更新 B ，这里就不细讲了。

优化器（补充3）

参考文献：

1. 深度学习中的训练优化问题 <https://zhuanlan.zhihu.com/p/97329073>

1、SGD和BGD区别，mini-batch 梯度下降法呢？通常使用哪个？

BGD是批量梯度下降法：是梯度下降法最原始的形式，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。

SGD是随机梯度下降法：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，就已经将theta迭代到最优解了，对比上面的批量梯度下降，迭代一次需要用到十几万训练样本，一次迭代不可能最优，如果迭代10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。

首先从计算量上来说，BGD一个很大的问题是每次迭代计算梯度的时候，都需要扫描整个数据集，因此当数据量很大的时候，就不可避免的导致计算量大，效率低下。而SGD在每一次迭代计算梯度时只需要取一个样本点，因此具有计算上的优势。

其次，由于SGD每次计算出的梯度与真实的负梯度方向差别较大，因此不是很稳定，这也解释了SGD的一个优点，可以跳出局部最优解，从而寻找到真正的全局最优解。这一点在深度学习中尤其重要，因为深度学习中的目标函数往往是非凸的。

更为明确的解释：

BGD在每次更新模型的时候，都要使用全量样本来计算更新的梯度值。如果有m个样本，迭代n轮，那么需要是 $m \times n$ 的计算复杂度。

SGD在每次更新模型的时候，只要当前遍历到的样本来计算更新的梯度值就行了。如果迭代n轮，则只需要n的计算复杂度，因为每轮只计算一个样本。

以上就是BGD和SGD的区别，容易看出，BGD的优势在于计算的是全局最优解，效果较SGD会好一些，劣势在于计算开销大；SGD则相反，优势在于计算开销减小很多，劣势在于计算的是局部最优解，可能最终达不到全局最优解。在数据量大的时候，SGD是较好的折衷选择。

mini-batch梯度下降法是介于SGD和BGD中间的一种优化方式，基本思想是每次拿出batch_size大小的数据做参数更新，既不是全部数据量也不是单个数据量，这也是并行计算的一种应用。

BGD(Batch Gradient Decent)利用整个训练集计算梯度，优化速度慢，需要一次将所有数据加载进内存不适用于在线学习，因此，在神经网络的优化中常采用mini-batch SGD (Stochastic Gradient Descent)进行优化。

总结：

2. batch size对收敛速度的影响?

Batch size大，收敛速度会比较慢，因为参数每次更新所需要的样本量增加了，但是会沿着比较准确的方向进行。

3、在合理范围内，增大Batch_Size有何好处？

- 内存利用率提高了，大矩阵乘法的并行化效率提高。
- 跑完一次 epoch (全数据集) 所需的迭代次数减少，对于相同数据量的处理速度进一步加快。
- 在一定范围内，一般来说 Batch_Size 越大，其确定的下降方向越准，引起训练震荡越小。

4、盲目增大 batch_size 有何坏处？

- 内存利用率提高了，但是内存容量可能撑不住了。
- 跑完一次 epoch (全数据集) 所需的迭代次数减少，要想达到相同的精度，其所花费的时间大大增加了，从而对参数的修正也就显得更加缓慢。
- Batch_size 增大到一定程度，其确定的下降方向已经基本不再变化。

5、梯度下降如何跳出局部最优值？

可以采用随机梯度下降法或者mini-batch梯度下降法，因为每次更新的方向大概率不相同，这就有可能导致其逃离局部最优。

6、SGD每步做什么，为什么能online learning?

online learning 强调的是学习是实时的，流式的，每次训练不用使用全部样本，而是以之前训练好的模型为基础，每来一个样本就更新一次模型，这种方法叫做**OGD (online gradient descent)**。这样做的目的是快速地进行模型的更新，提升模型时效性。而SGD的思想正式在线学习的思想。

而 **batch learning** 或者叫 **offline learning** 强调的是每次训练都需要使用全量的样本，因而可能会面临数据量过大的问题。

后面要讨论的批量梯度下降法 (BGD) 和随机梯度下降法 (SGD) 都属于batch learning或者offline learning的范畴。

7、为什么负梯度方向是使函数值下降最快的方向？简单数学推导一下？

设我们的目标是最小化损失函数 $J(\theta)$ ，为了快速得到最佳的参数 θ ，我们需要找到损失函数下降最快的方向，即找到一个 θ 移动的方向，使得 $J(\theta) - J(\theta + v)$ 最大，对 $J(\theta + v)$ 进行一阶泰勒展开：

$$J(\theta + v) \approx J(\theta) + v^T \nabla_{\theta} J(\theta)$$

$$J(\theta) - J(\theta + v) \approx -v^T \nabla_{\theta} J(\theta)$$

要使 $-v^T \nabla_{\theta} J(\theta)$ 最大，也就是说 $v^T \nabla_{\theta} J(\theta)$ 最小，其实 $v^T \nabla_{\theta} J(\theta)$ 可以看成两个向量的点积，

可以表示为 $|v| \|\nabla_{\theta} J(\theta)\| \cos \alpha$ ，最小的时候 $\cos \alpha = -1$ ，也就是下降方向和梯度方向相反，

此时下降方向即是梯度方向的负方向。

知乎 @大伟的AI算法之路

详细请参考：<https://link.zhihu.com/?target=https%3A//blog.csdn.net/u013166817/article/details/85131588>

8、牛顿法的思想？

牛顿法的基本思想是基于二阶泰勒级数展开在某点 θ_0 附近来近似 $J(\theta)$ 的优化方法，其忽略了高阶导数：

$$J(\theta) \approx J(\theta_0) + (\theta - \theta_0)^T \nabla_{\theta} J(\theta_0) + \frac{1}{2} (\theta - \theta_0)^T H(\theta - \theta_0)$$

其中：H 是 J 相对于 θ 的 Hessian 矩阵在 θ_0 处的估计。

对于多元函数的情况：

$$f(x) = f(x_0) + \nabla f(x_0)^T (x - x_0) + \frac{1}{2} (x - x_0)^T \nabla^2 f(x_0) (x - x_0)$$

忽略二次及以上的项，并对上式两边同时求梯度，得到函数的导数为：

$$\nabla(x) = \nabla f(x_0) + \nabla^2 f(x_0) (x - x_0)$$

其中 $\nabla^2 f(x_0)$ 即为 Hessian 矩阵，我们写作 H，令函数的梯度为 0，则有：

$$\nabla f(x_0) + \nabla^2 f(x_0) (x - x_0) = 0 \Rightarrow x = x_0 - (\nabla^2 f(x_0))^{-1} \nabla f(x_0)$$

这时一个线性方程组的解，也就是对应了下面的公式：

$$x = x_0 - H^{-1} g$$

如果我们再求解这个函数的临界点，将得到牛顿参数更新规则：

$$\theta^* = \theta_0 - H^{-1} \nabla_{\theta} J(\theta_0)$$

深度学习中为什么不采用牛顿法作为优化算法呢？

原因一：牛顿法需要用到梯度和 Hessian 矩阵，这两个都难以求解。因为很难写出深度神经网络拟合函数的表达式，很难直接得到其梯度表达式，更不要说得到基于梯度的 Hessian 矩阵了。

原因二：即使可以得到梯度和 Hessian 矩阵，当输入向量的维度 N 较大时，Hessian 矩阵的大小是 $N \times N$ ，所需要的内存非常大。

原因三：在高维非凸优化问题中，鞍点相对于局部最小值的数量非常多，而且鞍点处的损失值相对于局部最小值处也比较大。而二阶优化算法是寻找梯度为 0 的点，所以很容易陷入鞍点。

9、BFGS(拟牛顿法-无约束优化算法)了解吗？L-BFGS算法呢？

BFGS算法具有牛顿法的一些优点，但没有牛顿法的计算负担。在这方面，BFGS和CG很像。然而，BFGS使用了一个更直接的方法近似牛顿更新。回顾牛顿更新，由下式子给出：

$$\theta^* = \theta_0 - H^{-1} \nabla_{\theta} J(\theta_0)$$

，其中， H 是 J 相对于 θ 的Hessian矩阵。运用牛顿法的主要难点是黑森矩阵的逆 H^{-1} 比较

难求。拟牛顿法所采用的方法是使用矩阵 M_t 近似逆，迭代地低秩更新精度以更好地近似

H^{-1} 。

当黑森矩阵逆近似 M_t 更新时，下降方向为 p_t 为 $p_t = M_t g_t$ 。该方向上的线搜索用于决定该

方向上的步长 ε^* 。参数的最后更新为：

$$\theta_{t+1} = \theta_t + \varepsilon^* p_t$$

L-BFGS是在BFGS的基础上做了进一步的改进，详细算法请见花书第194页。
知乎 @大伟的AI算法之路

10、二阶优化方法有哪些？相比一阶的区别？

二阶方法主要有牛顿法、BFGS算法。

11、高斯牛顿法用来优化目标函数时，设目标函数为 $f(x)$ ，其雅可比矩阵为 J 。那么在高斯牛顿法的每一次迭代中， x 的下降方向为？

$$x_{k+1} = x_k - H_k^{-1} g_k$$

其中：

$$H \approx J^T J$$

$$g_k = \frac{\partial f(x)}{\partial x}$$

因此最终的梯度下降方向为：

$$\theta = -(J^T J)^{-1} \frac{\partial f(x)}{\partial x}$$

知乎 @大伟的AI算法之路

总结：

12、深度学习一阶优化和二阶优化的方法有哪些？为什么不使用二阶优化？

首先看下什么是一阶优化算法：梯度下降法的基本原理是，通过对目标函数做泰勒展开，当变量的运动方向与梯度方向相同，目标函数值增长最快；负梯度方向，目标函数值减小最快。这里主要讨论将目标函数做**一阶泰勒展开**，也就是**一阶梯度优化方法**。一阶优化方法有：

- SGD+Momentum
- NAG
- AdaGrad
- RMSProp
- AdaDelta（使用一阶的方法来近似模拟二阶牛顿法）
- Adam

二阶优化主要是对目标函数进行**二阶泰勒展开**，也就是**二阶梯度**优化方法。二阶优化方法主要有：

- 牛顿法
- BFGS法

利用二阶泰勒展开，即牛顿法需要计算Hessian矩阵的逆，计算量大，在深度学习中并不常用。因此一般使用的一阶梯度优化。注意：二阶方法其实对参数的更新更为准确。

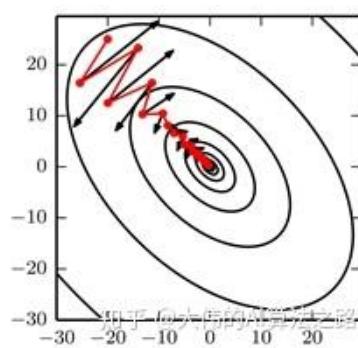
13、SGD、momentum、Nesterov momentum以及AdaGrad、RMSProp、Adam的算法原理及如何选择？

Mini-batch SGD+牛顿动量+RMSProp

Mini-batch SGD+Adam

14、为什么Momentum可以加速训练？

动量其实累加了历史梯度更新方向。在每次更新时，要是当前时刻的梯度与历史时刻梯度方向相似，这种趋势在当前时刻则会加强；要是不同，则当前时刻的梯度方向减弱。动量方法限制了梯度更新方向的随机性，使其沿正确方向进行。



15、Momentum和Nesterov Momentum的区别在哪里？

Momentum在参数更新前，没有对参数使用动量做一个先更新，而牛顿动量在参数更新前，对参数使用动量做了一个初步的更新。

16、AdaGrad的缺点？

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。

17、RMSProp的思想？

RMSProp 在 AdaGrad 的基础上进行了一项改进，那就是不直接累积所有历史梯度的平方和，而是逐渐忘掉遥远的梯度。（【锐平】通过下式的衰减因子 ρ ，不同时刻的梯度的系数是 ρ 的幂，使得越久远的梯度 ρ 的幂越大，即系数值越小，亦即权重越小。）

$$\gamma = \rho\gamma + (1 - \rho)g \odot g$$

18、RMSProp和Adam有什么不同？

RMSProp在二阶上对学习率做了一个衰减，Adam在这个基础上对梯度本身也做了一个衰减。

算法 8.7 Adam 算法

Require: 步长 ϵ (建议默认为: 0.001)
Require: 矩估计的指数衰减速率, ρ_1 和 ρ_2 在区间 $[0, 1)$ 内。 (建议默认为: 分别为 0.9 和 0.999)
Require: 用于数值稳定的小常数 δ (建议默认为: 10^{-8})
Require: 初始参数 θ
初始化一阶和二阶矩变量 $s = 0, r = 0$
初始化时间步 $t = 0$
while 没有达到停止准则 **do**
 从训练集中采包含 m 个样本 $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$ 的小批量，对应目标为 $\mathbf{y}^{(i)}$ 。
 计算梯度: $\mathbf{g} \leftarrow \frac{1}{m} \nabla_{\theta} \sum_i L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$
 $t \leftarrow t + 1$
 更新有偏一阶矩估计: $\hat{s} \leftarrow \rho_1 s + (1 - \rho_1) \mathbf{g}$
 更新有偏二阶矩估计: $\hat{r} \leftarrow \rho_2 \hat{r} + (1 - \rho_2) \mathbf{g} \odot \mathbf{g}$
 修正一阶矩的偏差: $\hat{s} \leftarrow \frac{\hat{s}}{1 - \rho_1^t}$
 修正二阶矩的偏差: $\hat{r} \leftarrow \frac{\hat{r}}{1 - \rho_2^t}$
 计算更新: $\Delta\theta = -\epsilon \frac{\hat{s}}{\sqrt{\hat{r} + \delta}}$ (逐元素应用操作)
 应用更新: $\theta \leftarrow \theta + \Delta\theta$
end while

这个是动量的一种表达形式

这个是RMSprop的一种表达形式

知乎 @大伟的AI算法之路

19、Adam能否适用于稀疏数据？

Adam算法的优势：

- 1、速度快；
- 2、可用于非平稳的目标函数/数据，即梯度的均值、协方差变化大；
- 3、可用于有噪声并且/或者稀疏的梯度；

20、根据黑森矩阵来判断极值点？

当Hessian矩阵是正定的，则该点是局部极小点；

当Hessian矩阵是负定的，该点位局部极大点；

当Hessian矩阵不定的时候，该点不是极值。

21、有约束优化算法和无约束优化算法有哪些？

这个其实可以和SVM中的优化方式结合在一起，像SVM中优化的时候要通过构建拉格朗日函数来把约束一起进行优化，这个叫**有约束优化**；

对于深度学习中的梯度下降、随机梯度下降等一阶梯度优化算法以及牛顿法、拟牛顿法、BFGS等二阶梯度优化算法都是可以直接建立损失函数的，然后对损失函数进行优化，没有约束，这些称之为**无约束优化算法**。