深度学习中的最优化问题

1. 深度学习中最优化问题很重要，代价也很高，因此需要开发一组专门的优化技术。
2. 在深度学习领域，即使在数据集和模型结构完全相同的情况下，选择不同的优化算法可能导致截然不同的训练效果。

   甚至相同的优化算法，但是选择了不同的参数初始化策略，也可能会导致不同的训练结果。

一、代价函数

1.1 经验风险最小化

1. 机器学习通常是间接的，需要优化的是测试集上的某个性能度量 P 。这个度量 P 通常很难直接求解，甚至难以直接建模优化。如：在图像目标检测问题中，常见的 P 就是`mAP:mean average precision` 。

   普遍的做法是：希望通过降低代价函数 $J(\vec{\theta})$ 来提高 P。这不同于纯粹的最小化 J 本身，因为最终目标是提高 P 。

   当代价函数 $J(\vec{\theta})$ 最小时是否 P 最大？这一结论是未知的。
2. 实际任务中，可以采用训练集上的损失函数均值作为代价函数：

$$J(\vec{\theta})=\mathbb{E}_{(\vec{\mathbf{x}},y)\sim\hat{p}_{data}}L(f(\vec{\mathbf{x}};\vec{\theta}),y)=\frac{1}{N}\sum_{i=1}^{N}L(f(\vec{\mathbf{x}}_i;\vec{\theta}),y_i)$$

   其中：L 为每个样本的损失函数，$f(\vec{\mathbf{x}};\vec{\theta})$ 为对输入 $\vec{\mathbf{x}}$ 的预测输出，$\hat{p}_{data}$ 是经验分布， y 为标记信息。
3. 理论上，代价函数中的期望最好取自真实的数据生成分布 p_{data} ，而不是有限个训练集上对应的经验分布 $\hat{p}_{data}$ 。即：$J^*(\vec{\theta})=\mathbb{E}_{(\vec{\mathbf{x}},y)\sim p_{data}}L(f(\vec{\mathbf{x}};\vec{\theta}),y)$ ，$J^*(\vec{\theta})$ 称作泛化误差。

   问题是对于绝大多数问题，样本的真实分布 p_{data} 是未知的，仅能提供训练集中的样本的分布 $\hat{p}_{data}$ 。

   实际应用中，使用经验分布 $\hat{p}_{data}$ 来代替真实分布 p_{data} 。这就是为什么使用 $J(\vec{\theta})$ 作为代价函数的原因。
4. 最小化训练集上的期望损失称作最小化经验风险`empirical risk` 。其缺点是：
   - 很容易过拟合 。
   - 某些类型的损失函数没有导数，无法使用基于梯度下降的优化算法来优化。

     如 ：`0-1` 损失函数，导数要么为零，要么没有定义。

1.2 替代损失函数

1. 有时候真正的代价函数无法有效优化，此时可以考虑使用`替代损失函数` 来代替真实的损失函数。

   如：将正类的负对数似然函数作为 `0-1` 损失函数的替代。
2. 一般的优化和机器学习优化的一个重要不同：机器学习算法通常并不收敛于代价函数的局部极小值。因为：
   - 机器学习算法通常使用`替代损失函数`。

     算法终止时，可能出现：采用 `替代损失函数`的代价函数的导数较小，而采用真实损失函数的代价函数的导数仍然较大（相比较于0值）。
   - 机器学习算法可能会基于早停策略而提前终止。

早停发生时，可能出现：训练集上的代价函数的导数值仍然较大（相比较于0值）。

因为早停的规则是基于验证集上代价函数的值不再下降，它并不关心训练集上代价函数的导数值。

二、神经网络最优化挑战

1. 机器学习中，通常会仔细设计目标函数和约束，从而保证最优化问题是凸的。但是神经网络中，通常遇到的都是非凸的最优化问题。

2.1 病态黑塞矩阵

1. 病态 `ill-conditioning` 的黑塞矩阵 $\mathbf{H}$ 是凸优化或者其他形式优化中普遍存在的问题。
   - 在神经网络训练过程中，如果 $\mathbf{H}$ 是病态的，则随机梯度下降会“卡”在某些地方：此时即使很小的更新步长也会增加代价函数。
   - 当黑塞矩阵是病态时，牛顿法是一个很好的解决方案。但是牛顿法并不适用于神经网络，需要对它进行较大改动才能用于神经网络。
2. 将 $f(\vec{\mathbf{x}})$ 在 $\vec{\mathbf{x}}_0$ 处泰勒展开：$f(\vec{\mathbf{x}}) \approx f(\vec{\mathbf{x}}_0) + (\vec{\mathbf{x}} - \vec{\mathbf{x}}_0)^T\vec{\mathbf{g}} + \frac{1}{2}(\vec{\mathbf{x}} - \vec{\mathbf{x}}_0)^T\mathbf{H}(\vec{\mathbf{x}} - \vec{\mathbf{x}}_0)$ 。其中 $\vec{\mathbf{g}}$ 为 $\vec{\mathbf{x}}_0$ 处的梯度； $\mathbf{H}$ 为 $\vec{\mathbf{x}}_0$ 处的海森矩阵。

   根据梯度下降法：$\vec{\mathbf{x}}' = \vec{\mathbf{x}} - \epsilon\nabla_{\vec{\mathbf{x}}} f(\vec{\mathbf{x}})$ 。应用在点 $\vec{\mathbf{x}}_0$ ，有：

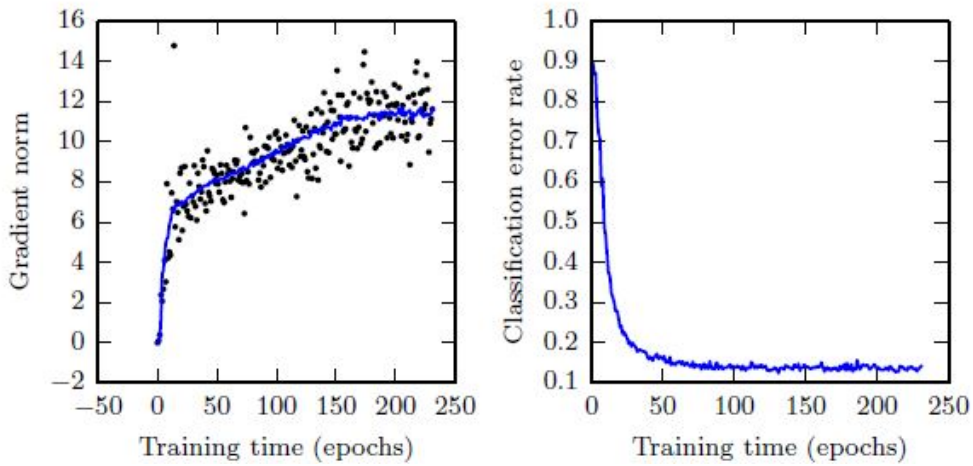
   $$f(\vec{\mathbf{x}}_0 - \epsilon\vec{\mathbf{g}}) \approx f(\vec{\mathbf{x}}_0) - \epsilon\vec{\mathbf{g}}^T\vec{\mathbf{g}} + \frac{1}{2}\epsilon^2\vec{\mathbf{g}}^T\mathbf{H}\vec{\mathbf{g}}$$

   因此沿着负梯度的方向，步长 $-\epsilon\vec{\mathbf{g}}$ 将导致代价函数 f 增加：$-\epsilon\vec{\mathbf{g}}^T\vec{\mathbf{g}} + \frac{1}{2}\epsilon^2\vec{\mathbf{g}}^T\mathbf{H}\vec{\mathbf{g}}$ 。

   当 $\frac{\epsilon}{2}\vec{\mathbf{g}}^T\mathbf{H}\vec{\mathbf{g}} > \vec{\mathbf{g}}^T\vec{\mathbf{g}}$ 时，黑塞矩阵的病态会成为问题。此时沿着负梯度的方向，代价函数 f 的值反而在增长！
3. 理论上，随着训练的推进，梯度的平方范数 $\vec{\mathbf{g}}^T\vec{\mathbf{g}}$ 应该逐步降低并最终收敛到 0 。因为代价函数 f 取极小值时，梯度为 0 。

   实际上在很多问题中，梯度的平方范数 $\vec{\mathbf{g}}^T\vec{\mathbf{g}}$ 并不会在训练过程中显著缩小，而是随着时间的延长在增加，且并不随着训练过程收敛到临界点而减小。

   下面左图为梯度的范数随着时间的变化，右图为验证集上的分类误差随着时间的变化。



   这是因为 $\vec{\mathbf{g}}^T\mathbf{H}\vec{\mathbf{g}}$ 会更快速的增长（相对于 $\vec{\mathbf{g}}^T\vec{\mathbf{g}}$ ）。它带来两个结果：
   - 尽管梯度很强，但是训练却可以成功，因为它使得代价函数 f 的增量不断逼近0（增量为0表示到达极值点）。
   - 尽管梯度很强，但是学习会变得非常缓慢，因为学习率必须减小从而适应更强的曲率。

2.2 局部极小值

1. 对于非凸函数，如神经网络，可能存在多个局部极小值。实际上这并不是一个严重的问题。
2. 如果一个训练集可以唯一确定一组模型参数，则该模型称作可辨认的`identifiable`。
   - 带有隐变量的模型通常是不可辨认的。因为可以批量交换隐变量，从而得到等价的模型。如：交换隐单元 i和 j 的权重向量。

     这种不可辨认性称作权重空间对称性`weight space symmetry`。
   - 也可以放大权重和偏置 α 倍，然后缩小输出 $\frac{1}{\alpha}$ 倍，从而保持模型等价。
   - 模型可辨认性问题意味着：神经网络的代价函数具有非常多、甚至是无限多的局部极小解。
   - 由可辨认性问题产生的局部极小解都具有相同的代价函数值，它并不是代价函数非凸性带来的问题。
3. 如果局部极小解和全局极小解相差很大时，此时多个局部极小解会带来很大隐患。它将给基于梯度的优化算法带来很大的问题。
   - 实际中的网络，是否存在大量严重偏离全局极小解的局部极小解、优化算法是否会遇到这些局部极小解？

     这些都是未决的问题。
   - 目前学者们猜想：对于足够大的神经网络，大部分局部极小值都具有很小的代价函数值。

     是否找到全局极小值并不重要，实际上只需要在参数空间中找到一个使得损失函数很小的点。
4. 目前很多人将神经网络优化中的所有困难都归结于局部极小值。

   有一种方案是排除局部极小值导致的困难（说明是其他原因导致的困难）：绘制梯度范数随着时间的变化：
   - 如果梯度范数没有缩小到一个很小的值，则问题的原因既不是局部极小值引起的，也不是其他形式的临界点（比如鞍点）引起的。
   - 如果梯度范数缩小到一个很小的值，则问题的原因可能是局部极小值引起的，也可能是其他原因引起的。
5. 神经网络训练中，通常不关注代价函数的精确全局极小值，而是关心将代价函数值下降到足够小，从而获得一个很好的泛化误差。

   关于优化算法能否到达这个目标的理论分析是极其困难的。

2.3 鞍点

1. 鞍点是另一类梯度为零的点。鞍点附近的某些点的函数值比鞍点处的值更大，鞍点附近的另一些点的函数值比鞍点处的值更小。
2. 在鞍点处，黑塞矩阵同时具有正负特征值：
   - 正特征值对应的特征向量方向的点，具有比鞍点更大的值。
   - 负特征值对应的特征向量方向的点，具有比鞍点更小的值。
3. 通常在低维空间中，局部极小值很普遍；在高维空间中，局部极小值很少见，鞍点更常见。
   - 对于函数 $f:\mathbb{R}^n \to \mathbb{R}$，鞍点和局部极小值的数量之比的期望随着 n 呈指数级增长。
   - 假如黑塞矩阵的特征值的正负号由抛硬币来决定的话：
     - 在一维情况下，很容易抛硬币得到正面向上。
     - 而在 n 维中，很难出现 n 次抛硬币都是正面向上。
4. 当位于函数值较低的区间时，黑塞矩阵的特征值为正的可能性更大。这意味着：
   - 具有较大函数值的临界点更可能是鞍点，因为此时黑塞矩阵的特征值可能既存在正值、也存在负值。

- 具有较小函数值的临界点更可能是局部极小值点，因为此时黑塞矩阵的特征值更可能全部为正值。
- 具有极高函数值的临界点更可能是局部极大值点，因为此时黑塞矩阵的特征值更可能全部为负值。

5. 鞍点对于训练算法的影响：
   - 对于只使用了梯度的一阶优化算法而言：情况不明。
     - 理论上，鞍点附近的梯度通常会非常小，这导致梯度下降算法沿着梯度方向的步长非常小。
     - 实际上，梯度下降算法似乎在很多情况下都能够逃离鞍点。
   - 对于牛顿法而言，鞍点是个大问题。

     因为梯度下降的原则是：朝着下坡路的方向移动。而牛顿法的原则是：明确寻找梯度为零的点。如果不做任何修改，则牛顿法会主动跳入一个鞍点。
6. 也可能出现一个恒值的、平坦的宽区域：在这个区域中，梯度和黑塞矩阵都为零。

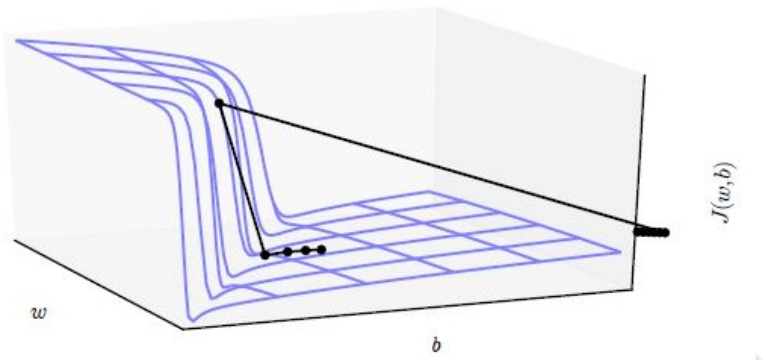
2.4 悬崖

1. 多层神经网络通常有像悬崖一样的区域，悬崖是指代价函数斜率较大的区域。
2. 产生悬崖的原因：由于几个较大的权重相乘，导致求导的时候，其梯度异常巨大。

   在`RNN`网络的代价函数中悬崖结构很常见，因为`RNN`这一类模型会涉及到多个时间步长中的因子的相乘，导致产生了大量的权重相乘。
3. 悬崖的影响：在梯度更新时，如果遇到悬崖，则会导致参数更新的步长非常大（因为此时梯度非常大），从而跨了非常大的一步，使得参数弹射的非常远。这样可能会使得已经完成的大量优化工作无效。

   因为当弹射非常远时，可能横跨了参数空间的很多个区域而进入到另一个区域。这样已经探索的参数区域就被放弃了。

   下图是一个悬崖的例子：第二根路径就是由于遇到悬崖，导致参数更新的步长非常大。



4. 解决悬崖问题的方案：使用梯度截断策略。

   梯度下降法只是指明了参数更新的方向（负梯度的方向），但是未指明最佳步长。当常规的梯度下降算法建议更新一大步时，梯度截断会干涉并缩减步长，从而使其基本上贴着悬崖来更新。如上图的第一根路径所示。

2.5 长期依赖

1. 当计算图非常深时，容易产生另一种优化困难：长期依赖。

   假设计算图中包含一条重复地、与矩阵 $\mathbf{W}$ 相乘的路径。经过 t 步，则相当于与 $\mathbf{W}^t$ 相乘。在第 i 步有：$\vec{\mathbf{h}}_t = \mathbf{W}^{t-i}\vec{\mathbf{h}}_i$。

根据反向传播原理，有：$\nabla_{\vec{\mathbf{h}}_i} J = \left(\frac{\partial \vec{\mathbf{h}}_t}{\partial \vec{\mathbf{h}}_i}\right)^T \nabla_{\vec{\mathbf{h}}_t} J = (\mathbf{W}^{t-i})^T \nabla_{\vec{\mathbf{h}}_t} J$ 。

考虑到权重 $\mathbf{W}$ 参与到每个时间步的计算，因此有：$\nabla_{\mathbf{W}} = \sum_{i=1}^{t} \frac{\partial J}{\partial \vec{\mathbf{h}}_i} \vec{\mathbf{h}}_{i-1}^T = \sum_{i=1}^{t} (\mathbf{W}^{t-i})^T (\nabla_{\vec{\mathbf{h}}_t} J) \vec{\mathbf{h}}_{i-1}^T$ 。

其中记 $\vec{\mathbf{x}} = \vec{\mathbf{h}}_0$ 。假设矩阵 $(\nabla_{\vec{\mathbf{h}}_t} J)\vec{\mathbf{x}}^T = c\mathbf{I}$，则有：

$$\nabla_{\mathbf{W}} = \sum_{i=1}^{t} (\mathbf{W}^{t-i})^T (\nabla_{\vec{\mathbf{h}}_t} J)(\mathbf{W}^{i-1}\vec{\mathbf{x}})^T = \sum_{i=1}^{t} c(\mathbf{W}^{t-1})^T = c \times t \times (\mathbf{W}^{t-1})^T$$

假设 $\mathbf{W}$ 有特征值分解 $\mathbf{W} = \mathbf{V}\Lambda\mathbf{V}^{-1}$，则：$\mathbf{W}^{t-1} = \mathbf{V}\Lambda^{t-1}\mathbf{V}^{-1}$ 。其中：

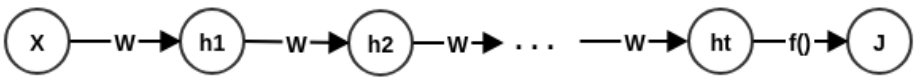
$$\Lambda = \begin{bmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \lambda_n \end{bmatrix}$$

考虑特征值 λ_i，当它不在 1 附近时：

- 如果量级大于 1，λ_i^t 非常大，这称作梯度爆炸问题 `exploding gradient problem` 。

  梯度爆炸使得学习不稳定，悬崖结构是梯度爆炸的一个例子。
- 如果量级小于 1，λ_i^t 非常小，这称作梯度消失问题 `vanishing gradient problem` 。

  梯度消失使得学习难以进行，此时学习的推进会非常缓慢。



2. 循环网络在每个时间步上使用相同的矩阵 $\mathbf{W}$，因此非常容易产生梯度爆炸和梯度消失问题。

   前馈神经网络并没有在每一层使用相同的矩阵 $\mathbf{W}$，因此即使是非常深层的前馈神经网络也能很大程度上避免梯度爆炸和梯度消失问题。

3. 对于梯度爆炸，可以通过梯度裁剪来缓解：限定梯度的范数的上限。

   对于梯度消失，不能够简单的通过放大来解决。因为有两个问题：

   - 当梯度很小的时候，无法分辨它是梯度消失问题，还是因为抵达了极小值点。
   - 当梯度很小的时候，噪音对梯度的影响太大。获得的梯度很可能由于噪音的影响，导致它的方向是随机的。

     此时如果放大梯度，则无法确保此时的方向就是代价函数下降的方向。而对于梯度爆炸，如果缩小梯度，仍然可以保证此时的方向就是代价函数下降的方向。

2.6 非精确梯度

1. 大多数优化算法都假设知道精确的梯度或者 `Hessian` 矩阵，实际中这些量都有躁扰，甚至是有偏的估计。

   如：`mini-batch` 随机梯度下降中，用一个 `batch` 的梯度来估计整体的梯度。

   各种神经网络优化算法的设计都考虑到了梯度估计的不精确。

2. 另外，实际情况中的目标函数是比较棘手的，甚至难以用简洁的数学解析形式给出。

   此时可以选择 `替代损失函数` 来避免这个问题。

2.7 局部和全局结构的弱对应

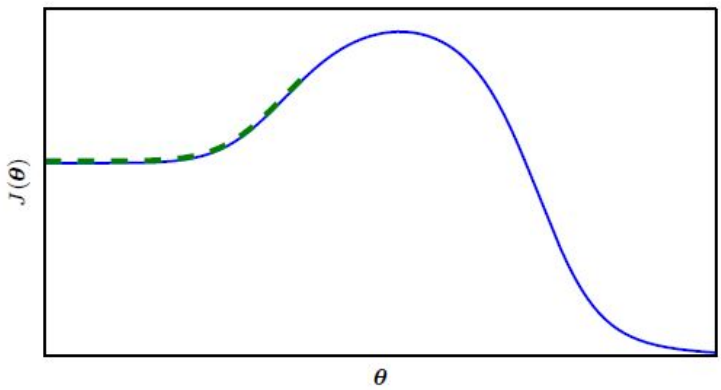
1. 对于最优化问题，即使克服了以上的所有困难，但是并没有到达代价函数低得多的区域，则表现仍然不佳。这就是局部优秀，全局不良。
   - 局部优秀：跨过了鞍点、爬过了悬崖、克服了梯度消失，最终到达局部极小值点。
   - 全局不良：并未到达目标函数全局比较小的值所在的区域。
2. 大多数优化研究的难点集中于：目标函数是否到达了全局最小值、局部最小值、鞍点。

   但是实践中，神经网络不会到达任何一种临界点，甚至不会到达梯度很小的区域，甚至这些临界点不是必然存在的。

   如：损失函数 $-\log p(y \mid \vec{\mathbf{x}};\vec{\theta})$ 没有全局极小点，而是趋向于某个极限值。
3. 下图的例子说明：即使没有局部极小值和鞍点，还是无法使用梯度下降得到一个好的结果。

   原因是：初始化在山的左侧。左侧向左趋向于一条渐近线，此时梯度的负方向会不停向左来逼近这条渐进线。

   理论上，优化算法要向右跨过山头，从而沿着右侧下降才能到达一个较低的函数值。



4. 在局部结构中执行梯度下降（称作`局部梯度下降`）的问题：
   - 局部梯度下降或许能找出一条解路径，但是该路径可能包含了很多次梯度更新，遵循该路径会带来很高的计算代价。
   - 如果目标函数是类似 $-\log p(y \mid \vec{\mathbf{x}};\vec{\theta})$ 函数：没有任何鞍点、极值点，而是具有一个宽而平坦的区域（这个区域逼近某个极限）。

     此时，若要寻求一个精确的临界点，则局部梯度下降无法给出解路径。这意味着算法难以收敛。
   - 局部梯度下降可能太过贪心，使得训练虽然朝着梯度下降的方向移动，但是远离了真正的解。
5. 如果存在某个参数区域，当遵循局部梯度下降就能够合理地直接到达最优解，且参数初始化点就位于该区域，则局部梯度下降的问题就得到解决。
6. 现有的很多研究方法在求解局部结构复杂的最优化问题时，解决方案为：寻求良好的初始化点，而不再是寻求良好的全局参数更新算法。

三、 mini-batch

1. 机器学习算法和一般最优化算法不同的一点：机器学习算法的目标函数通常可以分解为每个训练样本上的损失函数的求和。如：$\vec{\theta}_{ML}=\arg\max_{\vec{\theta}}\sum_{i=1}^{N}\log p_{model}(\vec{\mathbf{x}}_i,y_i;\vec{\theta})$ 。

   最大化这个总和，等价于最大化训练集在经验分布上的期望：

$$J(\vec{\theta})=\mathbb{E}_{(\vec{\mathbf{x}},y)\sim\hat{p}_{data}}\log p_{model}(\vec{\mathbf{x}},y;\vec{\theta})$$

2. 当使用基于梯度的优化算法求解时，需要用到梯度：

$$\nabla_{\vec{\theta}} J(\vec{\theta}) = \nabla_{\vec{\theta}} [\mathbb{E}_{(\vec{\mathbf{x}},y)\sim \hat{p}_{data}} \log p_{model}(\vec{\mathbf{x}}, y; \vec{\theta})] = \mathbb{E}_{(\vec{\mathbf{x}},y)\sim \hat{p}_{data}} \nabla_{\vec{\theta}} \log p_{model}(\vec{\mathbf{x}}, y; \vec{\theta})$$

   这个梯度本质上也是一个期望，要准确的求解这个期望的计算量非常大，因为需要计算整个数据集上的每一个样本。

   实践中，可以从数据集中随机采样少量的样本，然后计算这些样本上的梯度的均值，将这个均值作为该期望的一个估计。

3. 使用小批量样本来估计梯度的原因：
   - 使用更多样本来估计梯度的方法的收益是低于线性的。

     独立同分布的 n 个样本的均值 $\bar{X}$ 是个随机变量，其标准差为 $\frac{\sigma}{\sqrt{n}}$ ，其中 σ 为样本的真实标准差。
   - 如果能够快速计算出梯度的估计值（而不是费时地计算准确值），则大多数优化算法会更快收敛。

     大多数优化算法基于梯度下降，如果每一步中计算梯度的时间大大缩短，则它们会更快收敛。
   - 训练集存在冗余。

     实践中可能发现：大量样本都对梯度做出了非常相似的贡献。

     最坏情况下， 训练集中的 N 个样本都是相同的拷贝。此时基于小批量样本估计梯度的策略也能够计算正确的梯度，但是节省了大量时间。

4. 使用整个训练集的优化算法被称作`batch`梯度算法（或者确定性`deterministic`梯度算法）。

   每次只使用单个样本的优化算法被称作随机`stochastic`算法（或者在线`online`算法）。

   大多数深度学习的优化算法介于两者之间：使用一个以上、又不是采用全部的训练样本，称作`mini-batch`或者`mini-batch`随机算法。

5. 当使用小批量样本来估计梯度时，由于估计的梯度往往会偏离真实的梯度，这可以视作在学习过程中加入了噪声扰动。这种扰动会带来一些正则化效果。

3.1 mini-batch 大小

1. `mini-batch`的大小由下列因素决定：
   - 不能太大。更大的`batch`会使得训练更快，但是可能导致泛化能力下降。
     - 训练更快是因为：
       - 更大的`batch size`只需要更少的迭代步数就可以使得训练误差收敛。

         因为`batch size`越大，则小批量样本来估计总体梯度越可靠，则每次参数更新沿着总体梯度的负方向的概率越大。

         另外，训练误差收敛速度快，并不意味着模型的泛化性能强。
       - 更大的`batch size`可以利用大规模数据并行的优势。
     - 泛化能力下降是因为：更大的`batch size`计算的梯度估计更精确，它带来更小的梯度噪声。此时噪声的力量太小，不足以将参数推出一个尖锐极小值的吸引区域。

       解决方案为：提高学习率，从而放大梯度噪声的贡献。
   - 不能太小。因为对于多核架构来讲，太小的`batch`并不会相应地减少计算时间（考虑到多核之间的同步开销）。
   - 如果`batch`中所有样本可以并行地预处理，则内存消耗和`batch`大小成正比。

对于许多硬件设备来说，这就是`batch`大小的限制因素。

- 在有些硬件上，特定大小的效果更好。

  在使用`GPU`时，通常使用 2 的幂作为`batch`大小。

2. 泛化误差通常在`batch`大小为 1 时最好，但此时梯度估计值的方差非常大，因此需要非常小的学习速率以维持稳定性。如果学习速率过大，则导致步长的变化剧烈。

   由于需要降低学习速率，因此需要消耗更多的迭代次数来训练。虽然每一轮迭代中，计算梯度估计值的时间大幅度降低了（`batch size`为 1），但是总的运行时间还是非常大。

3. 某些算法对采样误差非常敏感，此时`mini-batch`效果较差。原因可能有两个：

   - 这些算法需要用到全部样本的一些精确信息，但是这些信息难以在少量样本上估计到。
   - 这些算法会放大采样误差，导致误差积累越来越严重。

4. 通常仅仅基于梯度 $\vec{\mathbf{g}}$ 的更新方法相对更稳定，它能够处理更小的`batch`（如100）。

   如果使用了黑塞矩阵 $\mathbf{H}$（如需要计算 $\mathbf{H}^{-1}\vec{\mathbf{g}}$ 的二阶方法）通常需要更大的`batch`（如 10000）。

   - 即使 $\mathbf{H}$ 被精确估计，但是它的条件数很差，那么乘以 $\mathbf{H}$ 或者 $\mathbf{H}^{-1}$ 会放大之前存在的误差。这就导致 $\vec{\mathbf{g}}$ 的一个非常小的变化也会导致 $\mathbf{H}^{-1}\vec{\mathbf{g}}$ 的一个非常大的变化。因此`batch`需要更大从而降低梯度估计的方差。
   - 通常只是近似地估计 $\mathbf{H}$ ，因此只会引入更多的误差。

3.2 随机抽样

1. `mini-batch`是随机抽样的也非常重要。从一组样本中计算出梯度期望的无偏估计要求：组内的样本是独立的。

   另外，也希望两个连续的梯度估计也是相互独立的。这要求：两个连续的`mini-batch`样本集合也应该是彼此独立的。

2. 实际应用中，采集的数据样本很可能出现这样的情况：连续的样本之间具有高度相关性。

   如：统计人群的偏好，很可能连续的样本就是同一个家庭、同一个职业、同一个地域...。

   解决方法是：将样本随机混洗之后存储，训练时按照混洗之后的顺序读取。

   这种打乱顺序不会对`mini-batch`产生严重的影响，不打乱顺序的`mini-batch`才会极大降低算法泛化能力。

3. `mini-batch`可以异步并行分布式更新：在计算`mini-batch`样本集合 $\mathbb{X}_i$ 上梯度更新时，也可以同时计算其他`mini-batch`样本集合 $\mathbb{X}_j$ 上的更新。

3.3 重复样本

1. 当 $\vec{\mathbf{x}}, y$ 都是离散时，泛化误差的期望：

$$J^*(\vec{\theta})=\mathbb{E}_{(\vec{\mathbf{x}},y)\sim p_{data}}L(f(\vec{\mathbf{x}};\vec{\theta}),y)=\sum_{(\vec{\mathbf{x}},y)}p_{data}(\vec{\mathbf{x}},y)L(f(\vec{\mathbf{x}};\vec{\theta}),y)$$

   其梯度为：

$$\vec{\mathbf{g}}=\nabla_{\vec{\theta}}J^*(\vec{\theta})=\sum_{(\vec{\mathbf{x}},y)}p_{data}(\vec{\mathbf{x}},y)\nabla_{\vec{\theta}}L(f(\vec{\mathbf{x}};\vec{\theta}),y)$$

   泛化误差的梯度的无偏估计可以通过从数据生成分布 p_{data} 抽取`mini-batch`样本 $\{\vec{\mathbf{x}}_1,\vec{\mathbf{x}}_2,\cdots,\vec{\mathbf{x}}_m\}$ 以及对应的标签 $\{y_1,y_2,\cdots,y_m\}$，然后计算`mini-batch`上的损失函数对于 $\vec{\theta}$ 的梯度：

$$\hat{\vec{\mathbf{g}}}=\frac{1}{m}\nabla_{\vec{\theta}}\sum_{i=1}^{m}L(f(\vec{\mathbf{x}}_i;\vec{\theta}),y_i)$$

它就是 $\vec{\mathbf{g}}$ 的无偏估计。

因此，`mini-batch` 随机梯度下降中，只要没有重复使用样本，它就是真实泛化误差梯度的无偏估计。

2. 如果是在线学习，则每个样本或者`mini-batch`都不会重复。每次更新都是独立地从真实分布 p_{data} 中采样获得。
3. 如果训练集不大，通常需要多次遍历训练集，此时只有第一遍满足泛化误差的梯度的无偏估计。

   后续的遍历中，泛化误差的梯度估计不再是无偏的。但是后续的遍历会减小训练误差，从而抵消了训练误差和测试误差之间`gap`的增加。最终效果是：减小了测试误差。
4. 随着数据集规模的爆炸性增长，超过了计算能力的增长速度，现在普遍地对每个样本只使用一次，甚至不完整地使用训练集。

   此时过拟合不再是问题，欠拟合以及计算效率成为主要问题。

四、基本优化算法

4.1 随机梯度下降 SGD

4.1.1 算法

1. 随机梯度下降沿着随机挑选的`mini-batch`数据的梯度下降方向推进，可以很大程度的加速训练过程。
2. 随机梯度下降`SGD`及其变种可能是机器学习中用的最多的优化算法。
3. 算法：
   - 输入：学习率 ϵ
   - 初始参数：$\vec{\theta}_0$
   - 算法步骤：

     迭代，直到满足停止条件。迭代步骤为：
     - 从训练集中随机采样 m 个样本 $\{\vec{\mathbf{x}}_1,\vec{\mathbf{x}}_2,\cdots,\vec{\mathbf{x}}_m\}$ 构成`mini-batch`，对应的标记为 $\{y_1,y_2,\cdots,y_m\}$。
     - 计算`mini-batch`上的梯度作为训练集的梯度的估计：

       $$\hat{\vec{\mathbf{g}}}\leftarrow\frac{1}{m}\nabla_{\vec{\theta}}\sum_{i=1}^{m}L(f(\vec{\mathbf{x}}_i;\vec{\theta}),y_i)$$

     - 更新参数：$\vec{\theta}\leftarrow\vec{\theta}-\epsilon\hat{\vec{\mathbf{g}}}$
4. 在深度学习中，通常的停止条件是：运行指定数量的迭代步或者`epoch`， 或者在验证集上的某个度量（如：损失函数、错误率、`auc`等）不再提升。

4.1.2 学习率

1. `SGD`中一个关键参数是学习率。前面介绍的`SGD`算法步骤使用固定的学习率 ϵ，实践中有必要随着时间的推移而降低学习率。

使用标准的梯度下降到达极小点时，整个代价函数的真实梯度非常小，甚至为零。由于 `SGD` 使用 `mini-batch` 的梯度作为整体梯度的估计，因此引入了噪源。

该噪源并不会在极小值处消失，使得在极小点时，梯度的估计可能会比较大。因此：标准的梯度下降可以使用固定的学习率，而 `SGD` 必须使用逐渐降低的学习率。

假设在极小点时，梯度的估计值 $\hat{\vec{\mathbf{g}}}$ 由于引入了噪源导致较大：

- 如果采取降低学习率的方法，则步长 $\epsilon\hat{\vec{\mathbf{g}}}$ 会很小，这就会导致参数 $\vec{\theta}$ 在极小点附近宅幅震荡直至收敛。
- 如果没有采取降低学习率的方法，则步长 $\epsilon\hat{\vec{\mathbf{g}}}$ 会很大，这会导致参数 $\vec{\theta}$ 在极小点附近宽幅震荡而且很难收敛。

2. 第 k 步的学习率记做 ϵ_k ，则对于学习率，保证 `SGD` 收敛的一个充分条件是：$\sum_{k=1}^{\infty}\epsilon_k=\infty$ ，且 $\sum_{k=1}^{\infty}\epsilon_k^2<\infty$ 。
3. 在实践中，学习率一般线性衰减到第 τ 次迭代，之后由于学习率足够小则可以保持不变：

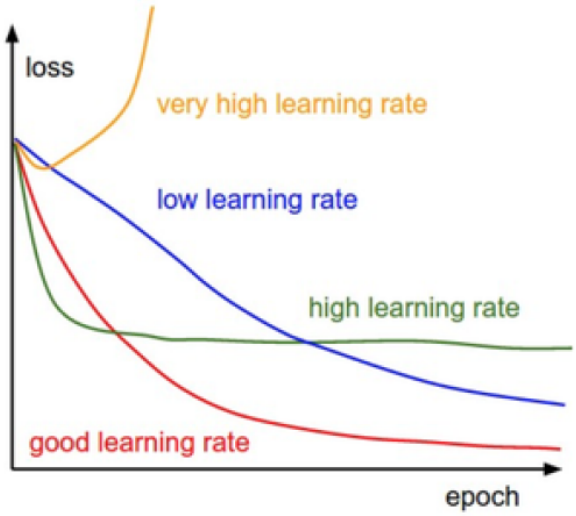
$$\epsilon_k=\begin{cases}(1-\frac{k}{\tau})\epsilon_0+\frac{k}{\tau}\epsilon_\tau & ,0\le k\le\tau\\ \epsilon_\tau & ,k\ge\tau\end{cases}$$

其中： τ 是预先指定的（如 `1000` ），ϵ_0,ϵ_τ 为常数。

学习率不能够衰减到零，因为一旦 ϵ 衰减到零，则很难说明模型收敛是因为学习率为零，还是梯度为零。

4. $\epsilon_0,\epsilon_\tau,\tau$ 可以通过试验来选取。
   - ϵ_τ 通常被设置为 ϵ_0 的大约 `1%` ， 即降低到足够低的位置。
   - τ 决定了学习率衰减的速度：经过多少个迭代步，使得学习率降低到足够低的位置。
   - ϵ_0 被称作初始学习率，它的选择是个重要因素：
     - 如果太大，则学习曲线将会剧烈震荡，代价函数值会明显增加。因为学习率太大，则容易发生超调现象。即：参数剧烈震荡，导致代价函数发散或者震荡。

       注意：温和的震荡是良好的。
     - 如果太小，则学习过程会非常缓慢，学习可能会卡在一个相当高的代价函数值上。
     - 通常最好检测最早的几轮迭代，使用一个高于此时效果最佳学习率的一个学习率，但是又不能太高以至于导致严重的不稳定性。



5. 学习速率的选择更像是一门艺术，而不是科学。

4.1.3 性质

1. `SGD` 以及其它的 `mini-batch` 算法的最重要性质是：每一步参数更新的计算时间（就是计算梯度的时间）不会随着训练样本数量的增加而增加。
   - 即使训练样本数量非常庞大时，算法也能收敛。
   - 对于足够大的数据集，`SGD` 可能在处理整个训练集的所有样本之前就收敛到测试集误差的允许范围之内了
2. 研究优化算法的收敛率，会衡量额外误差 `excess error`：$J(\vec{\theta})-\min_{\vec{\theta}}J(\vec{\theta})$。它刻画了：目标函数当前值到目标函数最小值的距离。
   - `SGD` 应用于凸问题时，k 步迭代之后的额外误差量级是 $O(\frac{1}{\sqrt{k}})$，在强凸情况下是 $O(\frac{1}{k})$。除非给定额外条件，否则这些界限无法进一步改进。
   - `Cramer-Rao` 界限指出：泛化误差的下降速度不会快于 $O(\frac{1}{k})$。
   - `Bottou and Bousquet` 认定：对于机器学习任务，不值得探寻收敛快于 $O(\frac{1}{k})$ 的优化算法。更快的收敛可能对应于过拟合。

   本章中剩余介绍的大多数算法都实现了实践中的好处，但是丢失了常数倍 $O(\frac{1}{k})$ 的渐进收敛率。
3. 可以结合标准梯度下降和 `SGD` 两者的优点，在学习过程中逐渐增大 `mini-batch` 的大小。

4.2 动量方法

4.2.1 算法

1. 应用了学习率衰减的 `SGD` 算法存在一个问题：有些时候学习率会很小，但是明明可以应用一个较大的学习率，而 `SGD` 并不知道这一情况。

   其解决方法是采用动量方法。
2. 动量方法积累了之前梯度的指数级衰减的移动平均，然后继续沿着该方向移动。
   - 它是一种移动平均，权重是指数级衰减的：近期的权重较大，远期的权重很小。
   - 动量方法取这些加权梯度的均值，根据该均值的方向决定参数的更新方向。
3. 动量方法是一种框架，它可以应用到随机梯度下降 `SGD` 算法中。
4. 动量方法引入了变量 $\vec{\mathbf{v}}$ 充当速度的角色：它刻画了参数在参数空间移动的方向和速度。

   定义速度为负梯度的指数衰减平均，其更新规则为：

   $$\vec{\mathbf{v}}\leftarrow\alpha\vec{\mathbf{v}}-\epsilon\nabla_{\vec{\theta}}J(\vec{\theta})$$
   $$\vec{\theta}\leftarrow\vec{\theta}+\vec{\mathbf{v}}$$

   超参数 $\alpha\in[0,1)$ 描述了衰减权重的底数，从近期到远期的衰减权重为 $\alpha,\alpha^2,\alpha^3,\cdots$
   - α 决定了梯度衰减有多快。
   - α 越大，则早期的梯度对当前的更新方向的影响越大；反之，则越小。
5. 令 $\vec{\theta}$ 为位移，$J(\vec{\theta})$ 为参数空间的势能，作用力 $\vec{\mathbf{F}}=-\epsilon\nabla_{\vec{\theta}}J(\vec{\theta})$。根据动量定理：

   $$\vec{\mathbf{F}}\Delta t=\Delta(m\times\vec{\mathbf{v}})$$

   - 令粒子为单位质量，时间间隔为单位时间，则有 $\vec{\mathbf{F}}=\Delta(\vec{\mathbf{v}})$，即：$-\epsilon\nabla_{\vec{\theta}}J(\vec{\theta})=\vec{\mathbf{v}}_{t_1}-\vec{\mathbf{v}}_{t_0}$。

     则得到更新公式：$\vec{\mathbf{v}}\leftarrow\vec{\mathbf{v}}-\epsilon\nabla_{\vec{\theta}}J(\vec{\theta})$。
   - 引入一个速度衰减系数 α，它对上一刻的速度进行降权，则有：$\vec{\mathbf{v}}\leftarrow\alpha\vec{\mathbf{v}}-\epsilon\nabla_{\vec{\theta}}J(\vec{\theta})$。

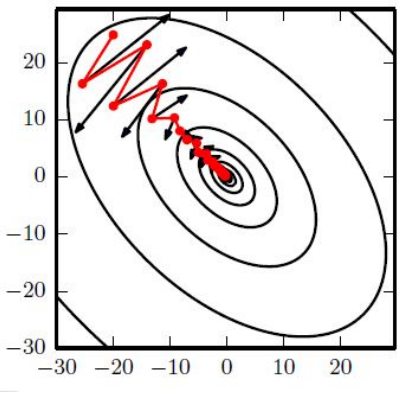
- 这段时间位移的增量为： $\vec{\mathbf{v}}\Delta t=\vec{\mathbf{v}}$，因此有位移更新公式：$\vec{\theta}\leftarrow\vec{\theta}+\vec{\mathbf{v}}$。
  - 由于这个单位时间内，速度发生了变化，因此应该用平均速度。但是由于速度的变化不大，因此用最新的速度也可以。
  - 当前时刻的最新速度等于下一时刻的起始速度，因此无论用起始速度还是最新速度，位移的更新序列都是相同的。
- 动量方法更新规则的物理意义为：速度更新，位移更新。

6. 下图给出了非动量的方法与动量方法的路径图。代价函数为一个二次函数，它的黑塞矩阵具有不良的条件数。

   红色路径表示动量方法的路径图，黑色箭头给出了在这些点非动量方法的更新方向和步长。

   - 可以看到：动量方法能够快速、正确地到达极小值，而非动量方法会浪费大量的时间在某些方向上宽幅震荡。
   - 原因是：动量方法中，经过加权移动平均之后，在指向极小值的方向上的速度 $\vec{\mathbf{v}}$ 不断加强，在垂直于极小值的方向上速度 $\vec{\mathbf{v}}$ 不断抵消。

     最终参数会沿着极小值方向快速到达极小值，而在垂直极小值方向波动很小。



7. 使用动量的`SGD`算法：

   - 输入：
     - 学习率 ϵ
     - 动量参数 α
     - 初始参数 $\vec{\theta}_0$
     - 初始速度 $\vec{\mathbf{v}}_0$
   - 算法步骤：

     迭代，直到满足停止条件。迭代步骤为：

     - 从训练集中随机采样 m 个样本 $\{\vec{\mathbf{x}}_1,\vec{\mathbf{x}}_2,\cdots,\vec{\mathbf{x}}_m\}$ 构成`mini-batch`，对应的标记为 $\{y_1,y_2,\cdots,y_m\}$。
     - 计算`mini-batch`上的梯度作为训练集的梯度的估计：

       $$\hat{\mathbf{g}}\leftarrow\frac{1}{m}\nabla_{\vec{\theta}}\sum_{i=1}^{m}L(f(\vec{\mathbf{x}}_i;\vec{\theta}),y_i)$$

     - 更新速度：$\vec{\mathbf{v}}\leftarrow\alpha\vec{\mathbf{v}}-\epsilon\hat{\mathbf{g}}$
     - 更新参数：$\vec{\theta}\leftarrow\vec{\theta}+\vec{\mathbf{v}}$

4.2.2 衰减因子

1. 非动量方法的步长只是梯度范数乘以学习率。采用动量之后，步长取决于梯度序列、衰减因子、学习率。
   - 当有许多连续的梯度指向相同的方向时，步长最大。

     可以理解为：不同单位时刻的作用力沿着同一个方向连续的次数越多，则质点的速度会较大（不停沿着同一个方向加速）。而步长就是质点的速度。
   - 动量方法中，如果梯度始终为常量 $\vec{\mathbf{g}}$，则它会在方向 $-\vec{\mathbf{g}}$ 上不停地加速，直到最终的步长为：$\frac{\epsilon\|\vec{\mathbf{g}}\|}{1-\alpha}$ 。

     > 通过求解速度序列 $\vec{\mathbf{v}}_n = \alpha\vec{\mathbf{v}}_{n-1} - \epsilon\vec{\mathbf{g}}$ 的解析表达式可以得到
2. 实践中，α 取值一般为 0.5、0.9、0.99。
   - 和学习率一样，α 也可以随着时间变化。通常初始时采用一个较小的值，后面慢慢变大。
   - 随着时间推移，改变 α 没有收缩 ϵ 更重要。因为只要 $0 < \alpha < 1$，则最终 $\lim_{t\to+\infty}\alpha^t = 0$ 。因此最终参数更新主导的还是 $-\epsilon\hat{\vec{\mathbf{g}}}$ 。

4.2.3 Nesterov 动量

1. `Nesterov` 动量是动量方法的变种，也称作 `Nesterov Accelerated Gradient`（`NAG`）。区别在于：计算 `mini-batch` 的梯度时，采用更新后的参数 $\vec{\theta} + \alpha\vec{\mathbf{v}}$ 。

   它可以视作向标准动量方法中添加了一个校正因子：

   $$\vec{\mathbf{v}} \leftarrow \alpha\vec{\mathbf{v}} - \epsilon\nabla_{\vec{\theta}} J(\vec{\theta} + \alpha\vec{\mathbf{v}})$$
   $$\vec{\theta} \leftarrow \vec{\theta} + \vec{\mathbf{v}}$$
2. 在凸 `batch` 梯度情况下，`Nesterov` 动量将额外误差收敛率从 $O(\frac{1}{k})$ 改进到 $O(\frac{1}{k^2})$ 。

   但是在随机梯度的情况下，`Nesterov` 动量并没有改进收敛率。

五、自适应学习率算法

1. 假设代价函数高度敏感于参数空间中的某些方向，则优化算法最好针对不同的参数设置不同的学习率。
   - 代价函数变化明显的参数方向（偏导数较大）：学习率较小，使得更新的步长较小。
   - 代价函数变化不明显的参数方向（偏导数较小）：学习率较大，使得更新的步长较大。
2. 对于标准的 `batch` 梯度下降优化算法，沿着负梯度的方向就是使得代价函数下降的方向。
   - 如果不同的方向设置了不同的学习率，则前进的方向不再是负梯度方向，有可能出现代价函数上升。
   - 对于 `mini-batch` 梯度下降，由于对梯度引入了躁扰，因此可以针对不同的方向设置不同的学习率。
3. 选择哪个算法并没有一个统一的共识，并没有哪一个算法表现的最好。

   目前流行的优化算法包括：`SGD`，具有动量的 `SGD`，`RMSProp`，`AdaDelta`，`Adam`。选用哪个算法似乎主要取决于使用者对于算法的熟悉程度，以便调节超参数。

5.1 delta-bar-delta

1. `delta-bar-delta` 算法是一个自适应学习率的算法，其策略是：对于代价函数，如果其偏导数保持相同的符号，则该方向的学习率应该增加；如果偏导数变化了符号，则该方向的学习率应该减小。

   偏导数符号变化，说明更新的方向发生了反向。如果此时降低了学习率，则会对震荡执行衰减，会更有利于到达平衡点（偏导数为0的位置）。

2. `delta-bar-delta`算法只能用于标准的梯度下降中。`mini-batch`算法对于梯度的估计不是精确的，因此对于正负号的判断会出现较大偏差。

   对于`mini-batch`随机梯度下降，有一些自适应学习率的算法。包括：`AdaGrad、RMSProp、Adam`等算法。

5.2 AdaGrad

1. `AdaGrad`算法会独立设置参数空间每个轴方向上的学习率。
   - 如果代价函数在某个方向上具有较大的偏导数，则这个方向上的学习率会相应降低。
   - 如果代价函数在某个方向上具有较小的偏导数，则这个方向上的学习率会相应提高。
2. `AdaGrad`算法的思想是：参数空间每个方向的学习率反比于某个值的平方根。这个值就是该方向上梯度分量的所有历史平方值之和。

$$\vec{\mathbf{r}}\leftarrow\vec{\mathbf{r}}+\hat{\vec{\mathbf{g}}}\odot\hat{\vec{\mathbf{g}}}$$
$$\vec{\theta}\leftarrow\vec{\theta}-\frac{\epsilon}{\sqrt{\vec{\mathbf{r}}}}\odot\hat{\vec{\mathbf{g}}}$$

   其中$\odot$ 表示两个向量的逐元素的相乘。
   - 用平方和的平方根而不是均值，因为分量可能为负值。
   - 用历史结果累加，因为考虑的是该方向上的平均表现，而不仅仅是单次的表现。
3. `AdaGrad`算法：
   - 输入：
     - 全局学习率 ϵ
     - 初始参数 $\vec{\theta}_0$
     - 小常数 δ（为了数值稳定，大约为 10^{-7}）
   - 算法步骤：
     - 初始化梯度累计变量 $\vec{\mathbf{r}}=\vec{\mathbf{0}}$
     - 迭代，直到停止条件。迭代步骤为：
       - 从训练集中随机采样 m 个样本 $\{\vec{\mathbf{x}}_1,\vec{\mathbf{x}}_2,\cdots,\vec{\mathbf{x}}_m\}$ 构成`mini-batch`，对应的标记为 $\{y_1,y_2,\cdots,y_m\}$。
       - 计算`mini-batch`上的梯度作为训练集的梯度的估计：

$$\hat{\vec{\mathbf{g}}}\leftarrow\frac{1}{m}\nabla_{\vec{\theta}}\sum_{i=1}^{m}L(f(\vec{\mathbf{x}}_i;\vec{\theta}),y_i)$$

       - 累计平方梯度：$\vec{\mathbf{r}}\leftarrow\vec{\mathbf{r}}+\hat{\vec{\mathbf{g}}}\odot\hat{\vec{\mathbf{g}}}$
       - 计算更新（逐元素）：$\Delta\vec{\theta}\leftarrow-\frac{\epsilon}{\delta+\sqrt{\vec{\mathbf{r}}}}\odot\hat{\vec{\mathbf{g}}}$
       - 更新参数：$\vec{\theta}\leftarrow\vec{\theta}+\Delta\vec{\theta}$
4. 由于随迭代次数的增加， r_i 的值也会增加，因此 $\frac{\epsilon}{\delta+\sqrt{\vec{\mathbf{r}}}}$ 随着迭代的推进而降低。这起到了一个学习率衰减的效果。

   另一方面，不同方向的分量 r_i 不同，平均来说梯度分量越大的方向的学习率下降的最快。这起到了一个自适应的效果。
5. 效果：
   - 在凸优化中，`AdaGrad`算法效果较好。

- `AdaGrad` 算法在某些深度学习模型上效果不错，但大多数情况下效果不好。

  在训练深度神经网络模型的实践中发现：`AdaGrad` 学习速率减小的时机过早，且减小的幅度过量。因为学习率与历史梯度的平方和的开方成反比，导致过快、过多的下降。

5.3 RMSProp

5.3.1 RMSProp 原始算法

1. `RMSProp` 是 `AdaGrad` 的一个修改：将梯度累计策略修改为指数加权的移动平均。

$$\vec{\mathbf{r}} \leftarrow \rho\vec{\mathbf{r}} + (1-\rho)\hat{\vec{\mathbf{g}}} \odot \hat{\vec{\mathbf{g}}}$$
$$\vec{\theta} \leftarrow \vec{\theta} - \frac{\epsilon}{\sqrt{\vec{\mathbf{r}}}} \odot \hat{\vec{\mathbf{g}}}$$

   其中 ρ 为衰减速率，它决定了指数加权移动平均的有效长度。

2. 标准的 `RMSProp` 算法：
   - 输入：
     - 全局学习率 ϵ
     - 衰减速率 ρ
     - 初始参数 $\vec{\theta}_0$
     - 小常数 δ（为了数值稳定，大约为 10^{-6}）
   - 算法步骤：
     - 初始化梯度累计变量 $\vec{\mathbf{r}} = \vec{\mathbf{0}}$
     - 迭代，直到停止条件。迭代步骤为：
       - 从训练集中随机采样 m 个样本 $\{\vec{\mathbf{x}}_1, \vec{\mathbf{x}}_2, \cdots, \vec{\mathbf{x}}_m\}$ 构成 `mini-batch`，对应的标记为 $\{y_1, y_2, \cdots, y_m\}$。
       - 计算 `mini-batch` 上的梯度作为训练集的梯度的估计：

$$\hat{\vec{\mathbf{g}}} \leftarrow \frac{1}{m}\nabla_{\vec{\theta}} \sum_{i=1}^{m} L(f(\vec{\mathbf{x}}_i;\vec{\theta}), y_i)$$

       - 累计平方梯度：$\vec{\mathbf{r}} \leftarrow \rho\vec{\mathbf{r}} + (1-\rho)\hat{\vec{\mathbf{g}}} \odot \hat{\vec{\mathbf{g}}}$
       - 计算更新（逐元素）：$\Delta\vec{\theta} \leftarrow -\frac{\epsilon}{\delta+\sqrt{\vec{\mathbf{r}}}} \odot \hat{\vec{\mathbf{g}}}$
       - 更新参数：$\vec{\theta} \leftarrow \vec{\theta} + \Delta\vec{\theta}$
3. 实践证明 `RMSProp` 是一种有效、实用的深度神经网络优化算法，目前它是深度学习业界经常采用的优化方法之一。

5.3.2 RMSProp 原始算法性质

1. 假设迭代过程中，梯度刚好是固定的某个量，令 $\vec{\mathbf{c}} = \hat{\vec{\mathbf{g}}} \odot \hat{\vec{\mathbf{g}}}$。对于某个方向，假设其分量为 $c = g^2$ 。

   对于 `RMSProp` 算法：根据等比数列求和公式，该方向的比例因子为：$r = c\,(1-\rho^{\tau})$ ，其中 τ 为迭代次数。
   - 该方向的学习率为：

$$\overline{\epsilon} = \frac{\epsilon}{\delta + \sqrt{c(1-\rho^{\tau})}}$$

- 随着 τ 的增大，$\bar{\epsilon}$ 会减小，因为分母在增加。
- r 在 τ 逐渐增大时，从 $c(1-\rho)$ 增加到 c ，其取值范围有限。它使得 $\bar{\epsilon}$ 取值从 $\frac{\epsilon}{\delta+\sqrt{c(1-\rho)}}$ 降低到 $\frac{\epsilon}{\delta+\sqrt{c}}$ 。

- 当某个方向的导数 g 相对于 δ 较大时，更新步长为（考虑到 $r=g^2(1-\rho^\tau)$ ）：

$$\frac{\epsilon}{\delta+\sqrt{r}}g\sim\frac{\epsilon}{\sqrt{1-\rho^\tau}}$$

  - 它与梯度无关，只与迭代次数 τ 有关。
  - 随着 τ 增大，从 $\frac{\epsilon}{\sqrt{1-\rho}}$ 降低到 ϵ 。
  - `RMSProP` 算法确保了在梯度较大时，学习的步长不会很小。
- 当导数 g 非常小以至于和 δ 相差无几时，此时更新步长与梯度有关。

  但是由于此时梯度非常小，算法可能已经收敛。

2. 对于 `AdaGrad` 算法的情况：根据等差数列的求和公式，该方向的比例因子为：$r=\tau c$ ，其中 τ 为迭代次数。
   - 该方向的学习率为：

$$\bar{\epsilon}=\frac{\epsilon}{\delta+\sqrt{\tau c}}$$

     - 随着 τ 的增大，$\bar{\epsilon}$ 会减小，因为分母在增加。
     - r 在 τ 逐渐增大时，从 c 增加到 $+\infty$ 。从而使得 $\bar{\epsilon}$ 取值从 $\frac{\epsilon}{\delta+\sqrt{c}}$ 降低到 0 。
   - 当该方向的梯度对于 δ 较大时，更新步长为：

$$\frac{\epsilon}{\delta+\sqrt{r}}g\sim\frac{\epsilon}{\sqrt{\tau}}$$

     它与梯度无关，只与迭代次数 τ 有关。随着 τ 增大，从ϵ 降低到 0 。

     因此，即使此时的梯度仍然较大，学习的步长仍然接近 0 。

3. `RMSProp` 算法 `vs AdaGrad` 算法：
   - `AdaGrad` 算法中，随着迭代次数的增加，其学习率降低到 0。 而 `RMSProp` 算法中，学习率降低到一个较大的渐进值 $\frac{\epsilon}{\delta+\sqrt{c}}$ 。
   - `AdaGrad` 算法中，随着迭代次数的增加，更新步长降低到 0 。而 `RMSProp` 算法中，更新步长降低到一个较大的值 ϵ 。

5.3.3 RMSProp 动量算法

1. `RMSProp` 也可以结合 `Nesterov` 动量算法。

   本质上 `Nesterov` 算法（包括其他的动量算法）是关于如何更新参数 $\vec{\theta}$ 的算法，它并不涉及任何学习率 ϵ 的选取。 `RMSProp` 算法是关于如何选取学习率 ϵ 的算法，它并不涉及如何更新参数 $\vec{\theta}$。因此二者可以结合。

2. `RMSProp` 动量算法
   - 输入：
     - 全局学习率 ϵ
     - 衰减速率 ρ
     - 动量系数 α
     - 初始参数 $\vec{\theta}_0$
     - 初始速度 $\vec{\mathbf{v}}_0$

- 算法步骤：
  - 初始化梯度累计变量 $\vec{\mathbf{r}}=\vec{\mathbf{0}}$
  - 迭代，直到停止条件。迭代步骤为：
    - 从训练集中随机采样 m 个样本 $\{\vec{\mathbf{x}}_1,\vec{\mathbf{x}}_2,\cdots,\vec{\mathbf{x}}_m\}$ 构成`mini-batch`，对应的标记为 $\{y_1,y_2,\cdots,y_m\}$
    - 计算临时更新：$\tilde{\vec{\theta}}\leftarrow\vec{\theta}+\alpha\vec{\mathbf{v}}$

      > 这是因为`Nesterov`动量算法使用更新后的参数来计算代价函数。如果使用原始的动量算法，则不需要这一步。

    - 计算`mini-batch`上的梯度作为训练集的梯度的估计：

      $$\hat{\vec{\mathbf{g}}}\leftarrow\frac{1}{m}\nabla_{\tilde{\vec{\theta}}}\sum_{i=1}^{m}L(f(\vec{\mathbf{x}}_i;\tilde{\vec{\theta}}),y_i)$$

    - 累计平方梯度：$\vec{\mathbf{r}}\leftarrow\rho\vec{\mathbf{r}}+(1-\rho)\hat{\vec{\mathbf{g}}}\odot\hat{\vec{\mathbf{g}}}$
    - 计算速度更新（逐元素）：$\vec{\mathbf{v}}\leftarrow\alpha\vec{\mathbf{v}}-\frac{\epsilon}{\sqrt{\vec{\mathbf{r}}}}\odot\hat{\vec{\mathbf{g}}}$
    - 更新参数：$\vec{\theta}\leftarrow\vec{\theta}+\vec{\mathbf{v}}$

5.3.4 AdaDelta

1. `AdaDelta`也是`AdaGrad`的一个修改。

   `AdaDelta`将梯度平方和的累计策略修改为：只考虑过去窗口 $window$ 内的梯度。

2. `RMSProp`可以认为是一种软化的`AdaDelta`，衰减速率 ρ 决定了一个软窗口：$window=\frac{1}{1-\rho}$。
   - 加权梯度平方和主要由 $window$ 内的梯度决定。
   - 更早的梯度由于衰减，其贡献非常小。

5.4 Adam

1. `Adam`来自于`Adaptive moments`，它是另一种引入了动量的`RMSProp`算法。

5.4.1 Adam 算法

1. `Adam`算法：
   - 输入：
     - 学习率 ϵ（建议默认为 0.001）
     - 矩估计的指数衰减速率 ρ_1,ρ_2，它们都位于区间 $[0,1)$（建议默认值分别为：0.9 和 0.999）
     - 用于数值稳定的小常数 δ（建议默认为 10^{-8}）
     - 初始参数 $\vec{\theta}_0$
   - 算法步骤：
     - 初始化一阶和二阶矩变量 $\vec{\mathbf{s}}=\vec{\mathbf{0}},\vec{\mathbf{r}}=\vec{\mathbf{0}}$
     - 初始化时间步 $t=0$
     - 迭代，直到停止条件。迭代步骤为：
       - 从训练集中随机采样 m 个样本 $\{\vec{\mathbf{x}}_1,\vec{\mathbf{x}}_2,\cdots,\vec{\mathbf{x}}_m\}$ 构成`mini-batch`，对应的标记为 $\{y_1,y_2,\cdots,y_m\}$

- 计算`mini-batch`上的梯度作为训练集的梯度的估计：

$$\hat{\vec{\mathbf{g}}} \leftarrow \frac{1}{m}\nabla_{\vec{\theta}}\sum_{i=1}^{m}L(f(\vec{\mathbf{x}}_i;\vec{\theta}),y_i)$$

- $t\leftarrow t+1$
- 更新有偏一阶矩估计：$\vec{\mathbf{s}}\leftarrow\rho_1\vec{\mathbf{s}}+(1-\rho_1)\hat{\vec{\mathbf{g}}}$。

  它是梯度的指数加权平均，用于计算梯度。
- 更新有偏二阶矩估计：$\vec{\mathbf{r}}\leftarrow\rho_2\vec{\mathbf{r}}+(1-\rho_2)\hat{\vec{\mathbf{g}}}\odot\hat{\vec{\mathbf{g}}}$。

  它是梯度平方的指数加权平均，用于计算学习率。
- 修正一阶矩的偏差：$\hat{\vec{\mathbf{s}}}\leftarrow\frac{\vec{\mathbf{s}}}{1-\rho_1^t}$。

  它使得修正之后的结果更接近真实的梯度。
- 修正二阶矩的偏差：$\hat{\vec{\mathbf{r}}}\leftarrow\frac{\vec{\mathbf{r}}}{1-\rho_2^t}$。

  它使得修正之后的结果更接近真实的梯度平方。
- 计算更新（逐元素）：$\Delta\vec{\theta}=-\frac{\epsilon}{\sqrt{\hat{\vec{\mathbf{r}}}}+\delta}\hat{\vec{\mathbf{s}}}$
- 更新参数：$\vec{\theta}\leftarrow\vec{\theta}+\Delta\vec{\theta}$

2. `RMSProp`算法中，通过累计平方梯度（采用指数移动平均）来修正学习率。

   而`Adam`算法中，不仅采用同样的方式来修正学习率，还通过累计梯度（采用指数移动平均）来修正梯度。
3. 动量的计算可以类似`Nesterov`动量的思想：计算`mini-batch`的梯度时，采用更新后的参数$\vec{\theta}+\Delta\vec{\theta}$。

   此时称作`NAdam`算法。

5.4.2 Adam 算法性质

1. 假设迭代过程中，梯度刚好是固定的某个量，则有：$\vec{\mathbf{c}}=\hat{\vec{\mathbf{g}}}\odot\hat{\vec{\mathbf{g}}}$。对于某个方向，假设其分量为$c=g^2$，则对于`Adam`算法有：

$$\begin{aligned}s&=g(1-\rho_1^t)\\r&=c(1-\rho_2^t)\\\hat{s}&=\frac{s}{1-\rho_1^t}=g\\\hat{r}&=\frac{r}{1-\rho_2^t}=c\\\Delta\theta&=-\epsilon\frac{\hat{s}}{\sqrt{\hat{r}}+\delta}\sim-\epsilon\frac{g}{\sqrt{g^2}}=\epsilon\end{aligned}$$

   - 无论梯度的大小如何，`Adam`算法的参数更新步长几乎都是ϵ

     相比较而言，`AdaGrad`和`RMSProp`算法的参数更新步长随时间在减少。
   - 虽然最终结果不包含ρ_1,ρ_2，但是这是由于假定梯度保持不变这一特殊情况推导的。

     实际中梯度很少保持不变，因此ρ_1,ρ_2还是比较重要。
2. `Adam`算法之所以需要使用一阶矩的修正和二阶矩的修正，是为了剔除时间因子t的影响。

3. 在某些情况下，`Adam` 可能导致训练不收敛。主要原因是：随着时间窗口的变化，遇到的数据可能发生巨变。这就使得 $\sqrt{\hat{r}}$ 可能会时大时小，从而使得调整后的学习率 $\hat{\epsilon}=\frac{\epsilon}{\sqrt{\hat{r}}+\delta}$ 不再是单调递减的。

   这种学习率的震荡可能导致模型无法收敛。

   > `AdaDelta`、`RMSProp` 算法也都存在这样的问题。

   解决方案为：对二阶动量的变化进行控制，避免其上下波动（确保 r 是单调递增的）：

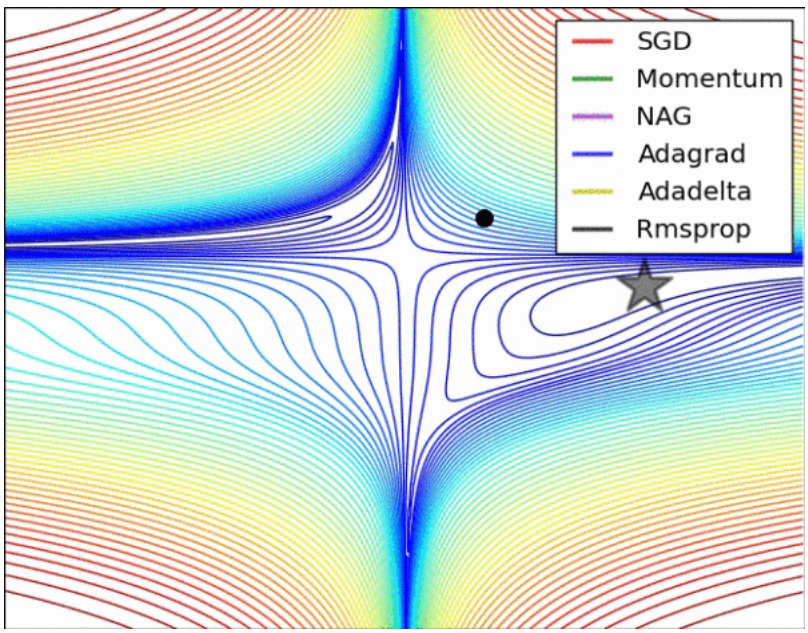
   $$\vec{\mathbf{r}}\leftarrow\max\{\rho_2\vec{\mathbf{r}}+(1-\rho_2)\hat{\vec{\mathbf{g}}}\odot\hat{\vec{\mathbf{g}}},\vec{\mathbf{r}}\}$$

4. 实践证明，虽然在训练早期 `Adam` 具有很好的收敛速度，但是最终模型的泛化能力并不如使用朴素的 `SGD` 训练得到的模型好：`Adam` 训练的模型得到的测试集误差会更大。
   - 其主要原因可能是：训练后期，`Adam` 的更新步长过小。
   - 一种改进策略为：在训练的初期使用 `Adam` 来加速训练，并在合适的时期切换为 `SGD` 来追求更好的泛化性能。

     这种策略的缺陷是：切换的时机不好把握，需要人工经验来干预。

5.5 SGD 及其变种的比较

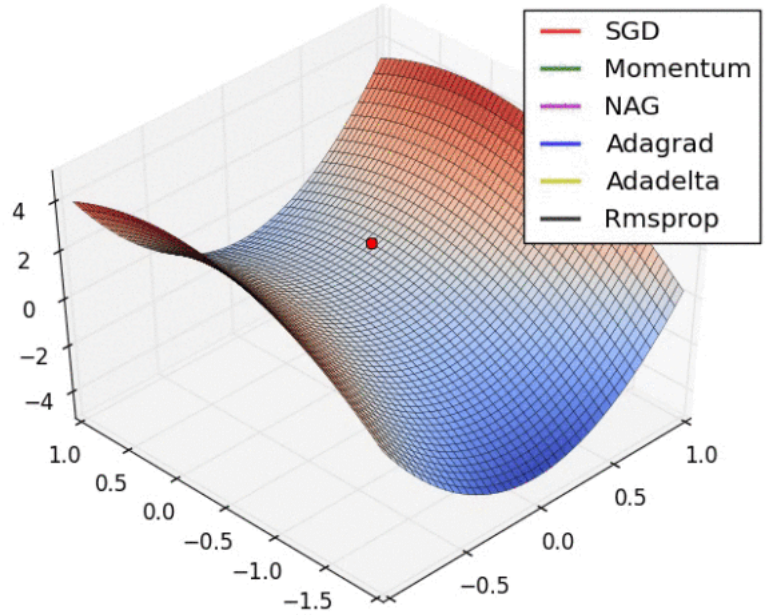
1. 下图为 `SGD` 及其不同的变种在代价函数的等高面中的学习过程。这里 `batch-size` 为全体样本大小。
   - `AdaGrad`、`Adadelta`、`RMSprop` 从最开始就找到了正确的方向并快速收敛。
   - `SGD` 找到了正确的方向，但是收敛速度很慢。
   - `SGD` 的动量算法、`Netsterov` 动量算法（也叫做 `NAG`）最初都偏离了正确方向，但最终都纠正到了正确方向。



2. 下图为 `SGD` 及其不同的变种在代价函数的鞍点上的学习过程。这里 `batch-size` 为全体样本大小。
   - `SGD`、`SGD` 的动量算法、`Netsterov` 动量算法（也叫做 `NAG`）都受到鞍点的严重影响。
     - `SGD` 最终停留在鞍点。

       如果采用 `mini-batch`，则 `mini-batch` 引入的梯度噪音，可以使得 `SGD` 能够逃离鞍点。
     - `SGD` 的动量算法、`Netsterov` 动量算法最终逃离了鞍点。

- `Adadelta`、`Adagrad`、`RMSprop` 都很快找到了正确的方向。



六、二阶近似方法

1. 为了简化问题，这里考察目标函数为经验风险函数（即：不考虑正则化项）：

$$J(\vec{\theta})=\mathbb{E}_{\vec{\mathbf{x}},y\sim\hat{p}_{data}(\vec{\mathbf{x}},y)}[L(f(\vec{\mathbf{x}};\vec{\theta}),y)]=\frac{1}{m}\sum_{i=1}^{m}L(f(\vec{\mathbf{x}}_i;\vec{\theta}),y_i)$$

.

6.1 牛顿法

6.1.1 牛顿法算法

1. 牛顿法在某点 $\vec{\theta}_0$ 附近，利用二阶泰勒展开来近似 $J(\vec{\theta})$：

$$J(\vec{\theta})\approx J(\vec{\theta}_0)+(\vec{\theta}-\vec{\theta}_0)^T\nabla_{\vec{\theta}}J(\vec{\theta}_0)+\frac{1}{2}(\vec{\theta}-\vec{\theta}_0)^T\mathbf{H}(\vec{\theta}-\vec{\theta}_0)$$

其中 $\mathbf{H}$ 为 J 对于 $\vec{\theta}$ 的海森矩阵在 $\vec{\theta}_0$ 处的值。

如果求解这个函数的临界点，根据 $\frac{\partial J(\vec{\theta})}{\partial\vec{\theta}}=\vec{\mathbf{0}}$，则得到牛顿法的更新规则：

$$\vec{\theta}^*=\vec{\theta}_0-\mathbf{H}^{-1}\nabla_{\vec{\theta}}J(\vec{\theta}_0)$$

   - 如果 J 为 $\vec{\theta}$ 的二次函数，则牛顿法会直接跳到极值或者鞍点。
     - 如果 $\mathbf{H}$ 是正定的，则跳到的点是极小值点。
     - 如果 $\mathbf{H}$ 是负定的，则跳到的点是极大值点。
     - 如果 $\mathbf{H}$ 是非正定、非负定的，则跳到的点是鞍点。因此，牛顿法会主动跳到鞍点。
   - 如果 J 为 $\vec{\theta}$ 的高阶的，则需要迭代。
2. 牛顿法：

- 输入：
  - 初始参数 $\vec{\theta}_0$
  - 包含 m 个样本的训练集
- 算法步骤：

  迭代，直到到达停止条件。迭代步骤为：
  - 计算梯度：$\vec{\mathbf{g}} \leftarrow \nabla_{\vec{\theta}} J(\vec{\theta})$
  - 计算海森矩阵：$\mathbf{H} \leftarrow \nabla^2_{\vec{\theta}} J(\vec{\theta})$
  - 计算海森矩阵的逆矩阵：$\mathbf{H}^{-1}$
  - 计算更新：$\Delta\vec{\theta} = -\mathbf{H}^{-1}\vec{\mathbf{g}}$
  - 应用更新：$\vec{\theta} = \vec{\theta} + \Delta\vec{\theta}$

6.1.2 牛顿法性质

1. 只要海森矩阵保持正定，牛顿法就能够迭代的使用。
2. 在深度学习中目标函数具有很多鞍点，海森矩阵的特征值并不全为正的，因此使用牛顿法有问题：在靠近鞍点附近，牛顿法会导致参数更新主动朝着鞍点的方向移动，这是错误的移动方向。

   解决的办法是：使用正则化的海森矩阵。
3. 常用的正则化策略是：在海森矩阵的对角线上增加常数 α。

   于是更新过程为：$\vec{\theta} = \vec{\theta} - [\mathbf{H} + \alpha\mathbf{I}]^{-1}\nabla_{\vec{\theta}} J(\vec{\theta})$ 。
   - 这个正则化策略是牛顿法的近似。
   - 只要海森矩阵的负特征值都在零点附近，则可以起到效果。
   - 当有很强的负曲率存在时（即存在绝对值很大的负的特征值），α 可能需要特别大。

     但是如果 α 增大到一定程度，则海森矩阵就变得由对角矩阵 $\alpha\mathbf{I}$ 主导。此时，牛顿法选择的方向就是 $-\frac{1}{\alpha}\vec{\mathbf{g}}$ 的方向。
4. 海森矩阵的元素的数量是参数数量的平方。如果参数数量为 n，则牛顿法需要计算一个 $n \times n$ 矩阵的逆，计算一次参数更新的复杂度为 $O(n^3)$ 。

   由于每次参数更新时都将改变参数，因此每轮迭代训练都需要计算海森矩阵的逆矩阵，计算代价非常昂贵。因此只有参数很少的网络才能够在实际中应用牛顿法训练，绝大多数神经网络不能采用牛顿法训练。
5. 综上所述，牛顿法有两个主要缺点：
   - 主动跳向鞍点。
   - 逆矩阵的计算复杂度太大，难以计算。

6.2 BFGS

1. `BFGS` 算法具有牛顿法的一些优点，但是没有牛顿法的计算负担。

   大多数拟牛顿法（包括 `BFGS` ）采用矩阵 $\mathbf{M}_t$ 来近似海森矩阵的逆矩阵，当 $\mathbf{M}_t$ 更新时，下降方向为 $\vec{\mathbf{d}}_t = \mathbf{M}_t\vec{\mathbf{g}}_t$。

   在该方向上的线性搜索到的最佳学习率为 ϵ^*，则参数更新为：$\vec{\theta}_{t+1} = \vec{\theta}_t + \epsilon^*\vec{\mathbf{d}}_t$ 。
2. `BFGS` 算法迭代了一系列线性搜索，其方向蕴含了二阶信息。

   与共轭梯度不同，`BFGS` 的成功并不需要线性搜索真正找到该方向上接近极小值的一点（而是探索一部分点）。因此 `BFGS` 在每个线性搜索上花费更少的时间。

3. `BFGS` 算法必须存储海森矩阵逆矩阵的近似矩阵 $\mathbf{M}_t$，需要 $O(n^2)$ 的存储空间。因此 `BFGS` 不适用于大多数具有百万级参数的现代深度学习模型。

   `L-BFGS` 通过避免存储完整的海森矩阵的逆的近似矩阵 $\mathbf{M}_t$ 来降低存储代价，它假设起始的 $\mathbf{M}_0$ 为单位矩阵，每步存储一些用于更新 $\mathbf{M}$ 的向量，并且每一步的存储代价是 $O(n)$ 。

七、共轭梯度法

7.1 最速下降法

1. 最速下降法是梯度下降法的一个特例：在每次梯度下降时，学习率的选择是使得当前函数值下降最快的那个学习率。

   $$\epsilon^* = \min_{\epsilon} f(\vec{\theta}_k - \epsilon \vec{\mathbf{g}}_k)$$

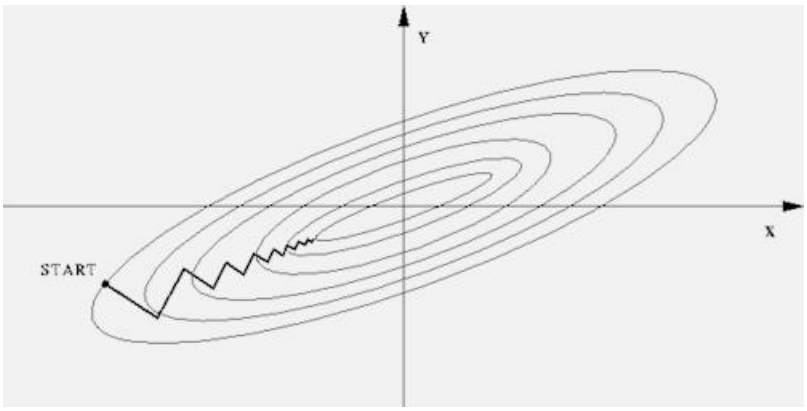
   梯度下降法仅仅指明：负梯度的方向是使得代价函数值下降的方向。它并没有说明沿着负梯度的方向推进多少步长。

   最速下降法通过显式对学习率建模，求解沿着负梯度的方向需要推进多少步长。

2. 在最速下降法中，假设上一次搜索方向是 $\vec{\mathbf{d}}_{t-1}$，当前搜索方向是 $\vec{\mathbf{d}}_t$。可以证明有：

   $$\vec{\mathbf{d}}_t \cdot \vec{\mathbf{d}}_{t-1} = 0$$

   即：前后两次搜索的方向是正交的。

   证明过程：

   已知 $\vec{\theta}_k$ 的梯度 $\vec{\mathbf{g}}_k$，根据 $0 = \frac{\partial f}{\partial \epsilon}$ ，则有：

   $$\begin{aligned} 0 = \frac{\partial f}{\partial \epsilon} &= -\nabla f(\vec{\theta}_k - \epsilon \vec{\mathbf{g}}_k) \cdot \vec{\mathbf{g}}_k \\ &\rightarrow \nabla f(\vec{\theta}_{k+1}) \cdot \vec{\mathbf{g}}_k = 0 \\ &\rightarrow \vec{\mathbf{g}}_{k+1} \cdot \vec{\mathbf{g}}_k = 0 \end{aligned}$$

3. 在最速下降法中，前进过程是锯齿形的。在某种意义上，这意味着消除了之前的线性搜索方向上取得的进展：

   如果前一次搜索已经找出了某个方向上的最小值，最速下降法不会保持这一方向。

7.2 共轭梯度法

1. 共轭：给定一个正定矩阵 $\mathbf{Q}$ ，如果非零向量 $\vec{\mathbf{x}}, \vec{\mathbf{y}}$ 满足：$\vec{\mathbf{x}}^T \mathbf{Q} \vec{\mathbf{y}} = 0$ ，则称向量 $\vec{\mathbf{x}}, \vec{\mathbf{y}}$ 关于 $\mathbf{Q}$ 共轭。
2. 如果 $\mathbf{Q} = \mathbf{I}$ 为单位矩阵，则有 $\vec{\mathbf{x}}^T \vec{\mathbf{y}} = 0$。即：向量共轭是向量点积为0的推广。

7.2.1 共轭梯度原理

1. 对于线性空间 $\mathbb{R}^n$，如果能找到 n 个向量 $\vec{\mathbf{d}}_1,\vec{\mathbf{d}}_2,\cdots,\vec{\mathbf{d}}_n$，且它们满足：$\vec{\mathbf{d}}_i^T\mathbf{Q}\vec{\mathbf{d}}_j=0,i\neq j$，则可以证明这一组向量是线性无关的。

   此时这一组向量可以作为线性空间的一组基向量，空间中任意向量 $\vec{\mathbf{x}}$ 都可以用这组向量表示：$\vec{\mathbf{x}}=\sum_{i=1}^n a_i\vec{\mathbf{d}}_i$ 。

2. 对于二次函数极值问题（其中 $\mathbf{Q}$ 为正定矩阵）：$\min_{\vec{\mathbf{x}}\in\mathbb{R}^n}\frac{1}{2}\vec{\mathbf{x}}^T\mathbf{Q}\vec{\mathbf{x}}-\vec{\mathbf{b}}^T\vec{\mathbf{x}}$，将 $\vec{\mathbf{x}}=\sum_{i=1}^n a_i\vec{\mathbf{d}}_i$ 代入上式，令 $\vec{\mathbf{a}}=(a_1,a_2,\cdots,a_n)^T$：

$$\begin{aligned}&\min_{\vec{\mathbf{a}}\in\mathbb{R}^n}\frac{1}{2}\left(\sum_{i=1}^n a_i\vec{\mathbf{d}}_i\right)^T\mathbf{Q}\left(\sum_{i=1}^n a_i\vec{\mathbf{d}}_i\right)-\vec{\mathbf{b}}^T\left(\sum_{i=1}^n a_i\vec{\mathbf{d}}_i\right)\\&=\min_{\vec{\mathbf{a}}\in\mathbb{R}^n}\frac{1}{2}\left(\sum_{i=1}^n\sum_{j=1}^n a_ia_j\vec{\mathbf{d}}_i^T\mathbf{Q}\vec{\mathbf{d}}_j\right)-\left(\sum_{i=1}^n a_i\vec{\mathbf{b}}^T\vec{\mathbf{d}}_i\right)\end{aligned}$$

   根据 $\vec{\mathbf{d}}_i\mathbf{Q}\vec{\mathbf{d}}_j=0$，有：

$$\min_{\vec{\mathbf{a}}\in\mathbb{R}^n}\frac{1}{2}\left(\sum_{i=1}^n a_i^2\vec{\mathbf{d}}_i^T\mathbf{Q}\vec{\mathbf{d}}_i\right)-\left(\sum_{i=1}^n a_i\vec{\mathbf{b}}^T\vec{\mathbf{d}}_i\right)=\min_{\vec{\mathbf{a}}\in\mathbb{R}^n}\sum_{i=1}^n\left(\frac{1}{2}a_i^2\vec{\mathbf{d}}_i^T\mathbf{Q}\vec{\mathbf{d}}_i-a_i\vec{\mathbf{b}}^T\vec{\mathbf{d}}_i\right)$$

3. 现在变量 $a_1,a_2,\cdots,a_n$ 已经被分开，可以分别优化。

   于是求解第 i 项的最小值：

$$\min_{a_i}\left(\frac{1}{2}a_i^2\vec{\mathbf{d}}_i^T\mathbf{Q}\vec{\mathbf{d}}_i-a_i\vec{\mathbf{b}}^T\vec{\mathbf{d}}_i\right)$$

   直接求导即可得到：

$$a_i^*=\frac{\vec{\mathbf{b}}^T\vec{\mathbf{d}}_\mathbf{i}}{\vec{\mathbf{d}}_i^T\mathbf{Q}\vec{\mathbf{d}}_i}$$

   最优解为：

$$\vec{\mathbf{x}}^*=\sum_{i=1}^n a_i^*\vec{\mathbf{d}}_i=\sum_{i=1}^n\frac{\vec{\mathbf{b}}^T\vec{\mathbf{d}}_\mathbf{i}}{\vec{\mathbf{d}}_i^T\mathbf{Q}\vec{\mathbf{d}}_i}\vec{\mathbf{d}}_i$$

4. 上述最优解可以这样理解：
   - 目标函数沿着方向 $\vec{\mathbf{d}}_i$ （即：$\vec{\mathbf{x}}=a_i\vec{\mathbf{d}}_i$）求解极小值，得到 a_i^* 。
   - 总和所有的 n 个方向求得的极小值，得到 $\vec{\mathbf{x}}^*=\sum_{i=1}^n a_i^*\vec{\mathbf{d}}_i$ 。

   现在的问题是：n 个关于 $\mathbf{Q}$ 共轭的向量组 $\{\vec{\mathbf{d}}_1,\vec{\mathbf{d}}_2,\cdots,\vec{\mathbf{d}}_n\}$未知。

7.2.2 共轭梯度搜索

1. 共轭梯度法通过迭代来搜索关于 $\mathbf{Q}$ 共轭的向量组 $\{\vec{\mathbf{d}}_1,\vec{\mathbf{d}}_2,\cdots,\vec{\mathbf{d}}_n\}$ 。

2. 令梯度 $\vec{\mathbf{g}}=\nabla_{\vec{\mathbf{x}}}f(\vec{\mathbf{x}})$，由于 $f(\vec{\mathbf{x}})=\frac{1}{2}\vec{\mathbf{x}}^T\mathbf{Q}\vec{\mathbf{x}}-\vec{\mathbf{b}}^T\vec{\mathbf{x}}$，则有：

$$\vec{\mathbf{g}}=\frac{\mathbf{Q}^T+\mathbf{Q}}{2}\vec{\mathbf{x}}-\vec{\mathbf{b}}$$

假设 $\mathbf{Q}$ 为对称的正定矩阵，则有：$\mathbf{Q}=\mathbf{Q}^T$ 。因此有：$\vec{\mathbf{g}}=\mathbf{Q}\vec{\mathbf{x}}-\vec{\mathbf{b}}$ 。

3. 任取一个初始点 $\vec{\mathbf{x}}_1$。
   - 如果 $\nabla_{\vec{\mathbf{x}}}f(\vec{\mathbf{x}}_1)=\vec{0}$，则停止计算。因为此时 $\vec{\mathbf{x}}_1$ 就是极值点。
   - 否则，令 $\vec{\mathbf{d}}_1=\vec{\mathbf{g}}_1=-\nabla_{\vec{\mathbf{x}}}f(\vec{\mathbf{x}}_1)$ ，即：$\vec{\mathbf{d}}_1$ 沿着点 $\vec{\mathbf{x}}_1$ 的梯度方向。

     沿着 $\vec{\mathbf{d}}_1$ 方向搜索： $\vec{\mathbf{x}}_2=\vec{\mathbf{x}}_1+\lambda\vec{\mathbf{d}}_1$，使得沿着这个方向，目标函数下降最多。

     $$\min_{\lambda}\left(\frac{1}{2}(\vec{\mathbf{x}}_1+\lambda\vec{\mathbf{d}}_1)^T\mathbf{Q}(\vec{\mathbf{x}}_1+\lambda\vec{\mathbf{d}}_1)-\vec{\mathbf{b}}^T(\vec{\mathbf{x}}_1+\lambda\vec{\mathbf{d}}_1)\right)-\left(\frac{1}{2}\vec{\mathbf{x}}_1^T\mathbf{Q}\vec{\mathbf{x}}_1-\vec{\mathbf{b}}^T\vec{\mathbf{x}}_1\right)$$

     解得：

     $$\lambda=-\frac{\vec{\mathbf{x}}_1^T\mathbf{Q}\vec{\mathbf{d}}_1-\vec{\mathbf{b}}^T\vec{\mathbf{d}}_1}{\vec{\mathbf{d}}_1^T\mathbf{Q}\vec{\mathbf{d}}_1}$$

     由于 $\vec{\mathbf{g}}^T=\vec{\mathbf{x}}^T\mathbf{Q}-\vec{\mathbf{b}}^T$ ，因此有：

     $$\lambda=-\frac{\vec{\mathbf{g}}_1^T\vec{\mathbf{d}}_1}{\vec{\mathbf{d}}_1^T\mathbf{Q}\vec{\mathbf{d}}_1}$$

4. 选择第二个迭代点：

   $$\vec{\mathbf{x}}_2=\vec{\mathbf{x}}_1+\lambda\vec{\mathbf{d}}_1=\vec{\mathbf{x}}_1-\frac{\vec{\mathbf{g}}_1^T\vec{\mathbf{d}}_1}{\vec{\mathbf{d}}_1^T\mathbf{Q}\vec{\mathbf{d}}_1}\vec{\mathbf{d}}_1$$

   注意：这里有 $\vec{\mathbf{g}}_1=\vec{\mathbf{d}}_1$，但是不保证后面的 $\vec{\mathbf{g}}_i=\vec{\mathbf{d}}_i$ 。

   构建下一个搜索方向，使得 $\vec{\mathbf{d}}_2$ 与 $\vec{\mathbf{d}}_1$ 关于 $\mathbf{Q}$ 共轭。

   令：$\vec{\mathbf{d}}_2=-\vec{\mathbf{g}}_2+\beta_1\vec{\mathbf{d}}_1$，代入 $\vec{\mathbf{d}}_1^T\mathbf{Q}\vec{\mathbf{d}}_2=0$，有：

   $$\beta_1=\frac{\vec{\mathbf{d}}_1^T\mathbf{Q}\vec{\mathbf{g}}_2}{\vec{\mathbf{d}}_1^{\mathbf{T}}\mathbf{Q}\vec{\mathbf{d}}_1},\quad \vec{\mathbf{d}}_2=-\vec{\mathbf{g}}_2+\frac{\vec{\mathbf{d}}_1^T\mathbf{Q}\vec{\mathbf{g}}_2}{\vec{\mathbf{d}}_1^{\mathbf{T}}\mathbf{Q}\vec{\mathbf{d}}_1}\vec{\mathbf{d}}_1$$

5. 同理：得到 $\vec{\mathbf{x}}_{k+1}$ 时，计算梯度 $\vec{\mathbf{g}}_{k+1}$，同时用 $-\vec{\mathbf{g}}_{k+1}$ 和 $\vec{\mathbf{d}}_k$ 来构建下一个搜索方向：

   $$\vec{\mathbf{d}}_{k+1}=-\vec{\mathbf{g}}_{k+1}+\beta_k\vec{\mathbf{d}}_k,\quad \vec{\mathbf{d}}_k^T\mathbf{Q}\vec{\mathbf{d}}_{k+1}=0$$

   得到：

   $$\beta_k=\frac{\vec{\mathbf{d}}_k^T\mathbf{Q}\vec{\mathbf{g}}_{k+1}}{\vec{\mathbf{d}}_{\mathbf{k}}^{\mathbf{T}}\mathbf{Q}\vec{\mathbf{d}}_{\mathbf{k}}},\quad \vec{\mathbf{d}}_{k+1}=-\vec{\mathbf{g}}_{k+1}+\frac{\vec{\mathbf{d}}_k^T\mathbf{Q}\vec{\mathbf{g}}_{k+1}}{\vec{\mathbf{d}}_{\mathbf{k}}^{\mathbf{T}}\mathbf{Q}\vec{\mathbf{d}}_{\mathbf{k}}}\vec{\mathbf{d}}_k$$

.

7.2.3 共轭梯度算法

1. 共轭梯度法的搜索方向 $\vec{\mathbf{d}}_t$ 会保持前一次线性搜索方向上 $\vec{\mathbf{d}}_{t-1}$ 取得的进展：

   $$\vec{\mathbf{d}}_t=-\nabla_{\vec{\theta}}J(\vec{\theta})+\beta_t\vec{\mathbf{d}}_{t-1}$$

   其中 β_t 的大小控制了：要保留多大比例的上一次搜索方向。

2. 在实际应用中，目标函数往往是高于二次的函数。此时为非线性共轭梯度， $\mathbf{Q}$ 就是海森矩阵 $\mathbf{H}$ 。

   直接计算 β_t 也需要计算海森矩阵的逆矩阵，比较复杂。有两个计算 β_t 的流行方法（不需要计算海森矩阵的逆矩阵）：

   > 这里不再保证 $\vec{\mathbf{d}}_t$ 和 $\vec{\mathbf{d}}_{t-1}$ 关于 $\mathbf{Q}$ 是共轭的。

   - `Fletcher-Reeves` ：

   $$\beta_t=\frac{\vec{\mathbf{g}}_t^T\vec{\mathbf{g}}_t}{\vec{\mathbf{g}}_{t-1}^T\vec{\mathbf{g}}_{t-1}}$$

   - `Polak_Ribiere` ：

   $$\beta_t=\frac{(\vec{\mathbf{g}}_t-\vec{\mathbf{g}}_{t-1})^T\vec{\mathbf{g}}_t}{\vec{\mathbf{g}}_{t-1}^T\vec{\mathbf{g}}_{t-1}}$$

3. 目标函数在高于二次的函数时，往往可能存在局部极值。此时共轭梯度法不能在 n 维空间内依靠 n 步搜索到达极值点。

   此时需要重启共轭梯度法，继续迭代，以完成搜索极值点的工作。

4. 共轭梯度法：

   - 输入：
     - 初始参数 $\vec{\theta}_0$
     - 包含 m 个样本的训练集
   - 算法步骤：
     - 初始化：

     $$\vec{\mathbf{d}}_0=\vec{\mathbf{0}}$$
     $$\vec{\mathbf{g}}_0=\vec{\mathbf{0}}$$
     $$t=1$$

     - 迭代，直到到达停止条件。迭代步骤为：
       - 计算梯度：

       $$\vec{\mathbf{g}}_t\leftarrow\frac{1}{m}\nabla_{\vec{\theta}}\sum_{i=1}^{m}L(f(\vec{\mathbf{x}}_i;\vec{\theta}),y_i)$$

       - 计算 β_t

       $$\beta_t=\frac{(\vec{\mathbf{g}}_t-\vec{\mathbf{g}}_{t-1})^T\vec{\mathbf{g}}_t}{\vec{\mathbf{g}}_{t-1}^T\vec{\mathbf{g}}_{t-1}}$$

       如果是非线性共轭梯度：则可以以一定的频率重置 β_t 为零（如每当 t 为 5 的倍数时）。

       - 计算搜索方向： $\vec{\mathbf{d}}_t=-\vec{\mathbf{g}}_t+\beta_t\vec{\mathbf{d}}_{t-1}$
       - 执行线性搜索： $\epsilon^*=\arg\min_\epsilon\frac{1}{m}\sum_{i=1}^{m}L(f(\vec{\mathbf{x}}_i;\vec{\theta}_t+\epsilon\vec{\mathbf{d}}_t),y_i)$

       > 在前面推导过程中，并没有出现这一步。这是因为：在非线性共轭梯度下，以及修改的 β_t 的情况下，不再满足 $\vec{\mathbf{d}}_t$ 和 $\vec{\mathbf{d}}_{t-1}$ 关于 $\mathbf{Q}$ 是共轭的。
       >
       > 对于真正的二次函数：存在 ϵ^* 的解析解，无需显式搜索。

- 应用更新：$\vec{\theta}_{t+1}=\vec{\theta}_t+\epsilon^*\vec{\mathbf{d}}_t$
- $t\leftarrow t+1$

7.2.4 共轭梯度算法性质

1. 神经网络的目标函数比二次函数复杂得多，但是共轭梯度法在这种情况下仍然适用。此时非线性共轭梯度算法会偶尔采取一些重置操作（重置 β_t 为零）。
2. 尽管共轭梯度算法是批方法（需要所有的样本来计算梯度），目前也开发出了`mini-batch`版本。
3. 理论上，对于二次函数的目标函数，需要迭代 n 步，n 为参数的个数。每一步中都需要计算梯度。

   在神经网络中，这样有两个问题：

   - 当参数的数量 n 巨大时，迭代的量非常大，而且难以并行（后面的迭代依赖前面的迭代），计算量非常庞大。
   - 当样本的数量巨大时，每次迭代中的梯度计算的计算量非常庞大。

八、优化策略和元算法

1. 有些优化技术并不是真正的算法，而是一个模板：它可以产生特定的算法。

8.1 坐标下降

1. 最小化 $f(\vec{\mathbf{x}})$ 可以采取如下的步骤：
   - 先相对于单一变量 x_i 最小化。
   - 然后相对于另一个变量 x_j 最小化。
   - ....
   - 如此反复循环所有的变量，可以保证到达（局部）极小值。

   这种做法被称作坐标下降。
2. 还有一种块坐标下降：它对于全部变量的一个子集同时最小化。
3. 当优化问题中的不同变量能够清晰地划分为相对独立的组，或者优化一组变量明显比优化所有变量的效率更高时，坐标下降最有意义。
4. 当一个变量值很大程度影响另一个变量的最优值时，坐标下降不是个好办法。如：

$$f(\vec{\mathbf{x}})=(x_1-x_2)^2+\alpha(x_1^2+x_2^2)$$

   - 第一项鼓励两个变量具有相近的值；第二项鼓励它们接近零。

     牛顿法可以一步解决该问题（它是一个正定二次问题），解为零。
   - 对于较小的 α ，此时函数值由第一项决定。

     此时采用坐标下降法非常缓慢，因为第一项不允许两个变量相差太大。
5. 坐标下降算法可以用于求解稀疏编码的代价函数最小化问题。

   给定训练集 $\mathbf{X}$ ，稀疏编码的目标是：寻求一个权重矩阵 $\mathbf{W}$ （未知的）和一个解码字典矩阵 $\mathbf{H}$ （也是未知的）来重构训练集 $\mathbf{X}$ ，其中要求字典矩阵 $\mathbf{H}$ 尽量稀疏 。

   代价函数为：$J(\mathbf{H},\mathbf{W})=\sum_{i,j}|\mathbf{H}_{i,j}|+\sum_{i,j}(\mathbf{X}-\mathbf{W}^T\mathbf{H})_{i,j}^2$ 。

   虽然代价函数 J 不是凸的，但是可以将输入分成两个集合：权重 $\mathbf{W}$ 和字典 $\mathbf{H}$。J 关于权重 $\mathbf{W}$ 是凸的，J 关于字典 $\mathbf{H}$ 也是凸的。

因此可以使用块坐标下降（其中可以使用高效的凸优化算法）：交替执行：固定 $\mathbf{H}$ 优化 $\mathbf{W}$ ，以及固定 $\mathbf{W}$ 优化 $\mathbf{H}$ 。

8.2 Polyak 平均

1. `Polyak` 平均的基本思想是：优化算法可能因为震荡，反复穿越极值点而没有落在极值点。因此可以考虑路径的均值来平滑输出。
2. 假设 t 次迭代，梯度下降的参数迭代路径为 $\theta^{(1)},\theta^{(1)},\cdots,\theta^{(t)}$ ，则 `Polyak` 平均算法的输出为：

$$\hat{\theta}^{(t)}=\frac{1}{t}\sum_i\theta^{(i)}$$

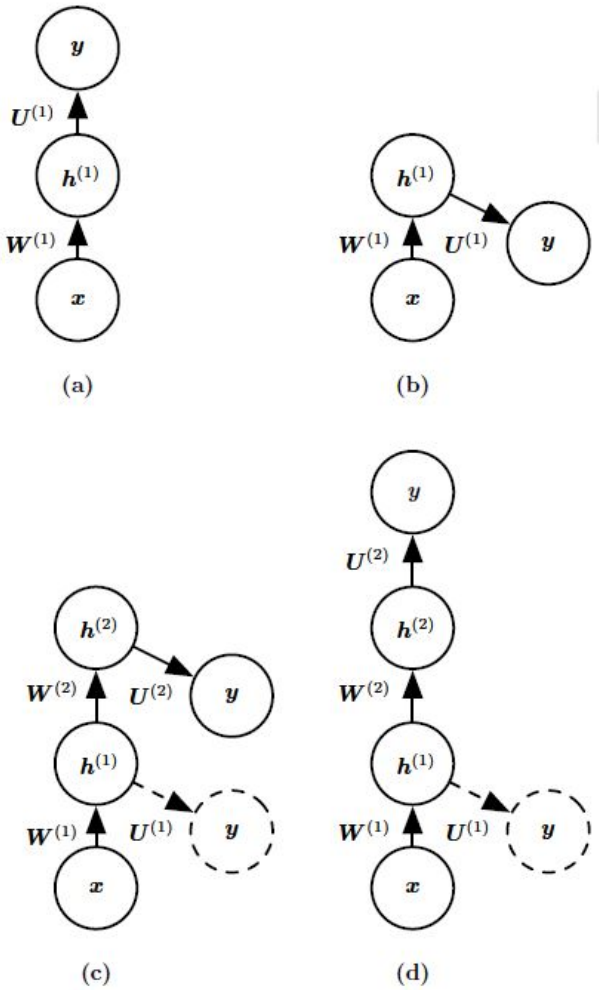
   - 对于凸问题，该方法具有较强的收敛保证。
   - 对于神经网络，这是一种启发式方法，实践中表现良好。
3. 在非凸问题中，优化轨迹的路径可能非常复杂。因此当 `Polyak` 应用于非凸问题时，通常会使用指数衰减来计算平均值：$\hat{\theta}^{(t)}=\alpha\hat{\theta}^{(t-1)}+(1-\alpha)\theta^{(t)}$ 。

8.3 贪心监督预训练

1. 有时模型太复杂难以优化，直接训练模型可能太过于困难。此时可以训练一个较简单的模型，然后逐渐使模型复杂化来求解原始问题。

   在直接训练目标模型、求解目标问题之前，训练简单模型求解简化问题的方法统称为预训练。
2. 预训练，尤其是贪心预训练，在深度学习中是普遍存在的。

   贪心监督预训练将复杂的监督学习问题分解成简化的监督学习问题。
3. 贪心监督预训练的一个例子如下图所示：
   - 先训练一个最简单的架构，只有一个隐层，如图 `a` 所示。图 `b` 是另一个画法。
   - 然后将第一个隐层的输出 $h^{(1)}$ 作为输入，再添加一个隐层，来训练 $h^{(1)}\rightarrow h^{(2)}\rightarrow y$，如图 `c` 所示。图 `d` 是另一个画法。
   - 然后将第二个隐层的输出作为输入，再添加一个隐层，训练....
   - 在这个过程中，前一步训练的最末尾的隐层的输出作为后一步训练的输入。
   - 为了进一步优化，最后可以联合微调所有层。



4. 贪心监督预训练有效的原因，`Bengio et al.`提出的假说是：它有助于更好地指导深层结构的中间层的学习。
    - 中间层的知识能够有助于训练神经网络。
    - 预训练在优化（提高训练速度）和泛化（提高模型的泛化能力）这两方面都是有帮助的。

8.4 选择有助于优化的模型

1. 改进优化的最好方法是选择一个好的模型，选择一族容易优化的模型比使用一个强大的优化算法更重要。
    - 深度模型中，优化的许多改进来自于易于优化的模型。如：使用`relu`激活函数。
    - 神经网络过去30年大多数进步主要来自于改变模型族，而不是优化算法。
    - 1980年代的带动量的随机梯度下降，依然是当前神经网络应用中的前沿算法。
2. 现代神经网络更多使用线性函数，如`relu`单元、`maxout`单元。

8.5 连续方法

1. 许多优化挑战都来自于：因为并不知道代价函数的全局结构，所以不知道最优解所在的区域。

    解决该问题的主要方法是：尝试初始化参数到某个区域内，该区域可以通过局部下降很快达到参数空间中的解。

2. 连续方法的原理：挑选一系列的初始化点，使得在表现良好的区域中执行局部优化。

    方法为：构造一系列具有相同参数的目标函数 $\{J^{(0)}(\theta), J^{(1)}(\theta), \cdots, J^{(n)}(\theta)\}$，其中满足：

- 这些代价函数逐步提高难度，其中 $J^{(0)}$ 是最容易优化的。
- 前一个代价函数的解是下一个的初始化点。

这样：首先解决一个简单的问题，然后改进解来解决逐步变难的问题，直到求解真正问题的解。

3. 传统的连续方法（非神经网络的）通常是基于平滑目标函数，主要用于克服局部极小值的问题。它用于在有许多局部极小值的情况下，求解一个全局极小值。
   - 它通过“模糊”原始的代价函数来构建更加容易的代价函数。这种模糊操作可以用采样来近似：

     $J^{(i)}(\theta) = \mathbb{E}_{\theta' \sim \mathcal{N}(\theta';\theta,\sigma^{(i)2})} J(\theta')$
   - 它背后的思想是：某些非凸函数，在模糊之后会近似凸的。
   - 通常这种模糊保留了关于全局极小值的足够多的信息。那么可以通过逐步求解更少模糊的问题，来求解全局极小值。
   - 这种方法有三种失败的可能：
     - 可能需要非常多的代价函数，导致整个过程的成本太高。
     - 不管如何模糊，可能代价函数还是没有办法变成凸的。
     - 函数可能在模糊之后，最小值会逐步逼近到原始代价函数的一个局部极小值，而不是原始代价函数的全局极小值。
4. 对于神经网络，局部极小值已经不是神经网络优化中的主要问题，但是连续方法仍然有所帮助。

   连续方法引入的简化的目标函数能够消除平坦区域、减少梯度估计的方差、提高海森矩阵的条件数，使得局部更新更容易计算，或者改进局部更新方向朝着全局解。

九、参数初始化策略

1. 有些优化算法是非迭代的，可以直接解析求解最优解；有些优化算法是迭代的，但是它们是初始值无关的。

   深度学习不具有这两类性质，通常是迭代的，且与初始值相关。

2. 深度学习中，大多数算法都受到初始值的影响。初始值能够决定：算法最终是否收敛、以及收敛时的收敛速度有多快、以及收敛到一个代价函数较高还是较低的值。

3. 深度学习中，初始值也会影响泛化误差，而不仅仅是目标函数的最优化。

   因为如果选择一个不好的初始值，则最优化的结果会落在参数空间的一个较差的区域。此时会导致模型一个较差的泛化能力。

4. 目前深度学习中，选择初始化策略是启发式的。
   - 大多数初始化策略使得神经网络初始化时实现一些良好的性质。但是这些性质能否在学习过程中保持，难以保证。
   - 有些初始化点从最优化的观点是有利的，但是从泛化误差的观点来看是不利的。
   - 设定一个好的初始化策略是困难的，因为神经网络最优化任务至今都未被很好理解。
   - 对于初始点如何影响泛化误差的理论是空白的，几乎没有任何指导。
5. 度学习中，初始值的选择目前唯一确定的特性是：需要在不同单元之间破坏对称性。
   - 如果具有相同激励函数的两个隐单元连接到相同的输入，则这些单元必须具有不同的初始化参数。

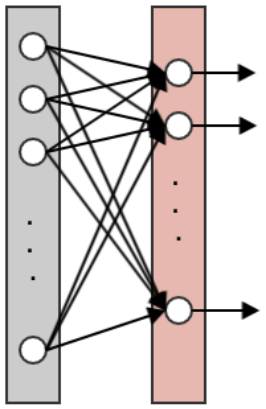
     如果它们具有相同的初始化参数，则非随机的学习算法将一直以同样的方式更新这两个单元。
   - 通常鼓励每个单元使用和其他单元不一样的函数，如选择不同的权重或者偏置。
6. 通常的参数初始化策略为：随机初始化权重，偏置通过启发式挑选常数，额外的参数也通过启发式挑选常数。
7. 也可以使用机器学习来初始化模型的参数。

在同样的数据集上，即使是用监督学习来训练一个不相关的任务，有时也能够得到一个比随机初始化更好的初始值。原因是：监督学习编码了模型初始参数分布的某些信息。

9.1 权重初始化

1. 通常权重的初始化是从高斯分布或者均匀分布中挑选出来的值。
   - 从高斯分布还是均匀分布中挑选，看起来似乎没有很大差别，实际上也没有被仔细研究。
   - 该分布的范围（如均匀分布的上、下限）对优化结果和泛化能力有很大的影响。
2. 初始权重的大小很重要，下面的因素决定了权重的初始值的大小：
   - 更大的初始权重具有更强的破坏对称性的作用，有助于避免冗余的单元。
   - 更大的初始权重也有助于避免梯度消失。
   - 更大的初始权重也容易产生梯度爆炸。
   - 循环神经网络中，更大的初始权重可能导致混沌现象：对于输入中的很小的扰动非常敏感，从而导致确定性算法给出了随机性结果。
3. 关于如何初始化网络，正则化和最优化有两种不同的角度：
   - 从最优化角度，建议权重应该足够大，从而能够成功传播信息。
   - 从正则化角度，建议权重小一点（如 L_2 正则化），从而提高泛化能力。
4. 有些启发式方法可用于选择权重的初始化大小。

   假设有 m 个输入， n 个输出的全连接层。



   - 常见的做法是建议使用均匀分布的随机初始化：$\mathbf{W}_{i,j} \sim U\left(-\sqrt{\frac{1}{m}}, \sqrt{\frac{1}{m}}\right)$。
   - `Glorot et al.` 建议使用均匀分布的随机初始化：$\mathbf{W}_{i,j} \sim U\left(-\sqrt{\frac{6}{m+n}}, \sqrt{\frac{6}{m+n}}\right)$。

     这种方法使网络在相同激励方差和相同的梯度方差之间折中。

     > 激励就是前向传播中，各信号（如权重、偏置等）的值。梯度就是它们的导数值。

5. 不幸的是上述启发式初始化权重的策略往往效果不佳。有三个可能的原因：
   - 可能使用了错误的标准：约束网络中信号（如梯度、权重）的范数可能并不会带来什么好处。
   - 初始化时强加给参数的性质可能在学习开始之后无法保持。
   - 可能提高了优化速度，但意外地增大了泛化误差。
6. `Martens` 提出了一种称作稀疏初始化的替代方案：每个单元初始化为恰好具有 k 个非零的权重。

   这个方案有助于单元之间在初始化时就具有更大的多样性。

7. 实践中，通常需要将初始权重范围视作超参数。

   如果计算资源允许，可以将每层权重的初始数值范围设置为一个超参数，然后使用超参数搜索算法来挑选这些超参数。

9.2 偏置初始化

1. 偏置的初始化通常更容易。大多数情况下，可以设置偏置初始化为零。
2. 有时可以设置偏置初始化为非零，这发生在下面的三种情况：
   - 如果偏置是作为输出单元，则初始化偏置为非零值。

     假设初始权重足够小，输出单元的输出仅由初始化偏置决定，则非零的偏置有助于获取正确的输出边缘统计。
   - 有时选择偏置的初始值以免初始化引起激活函数饱和。如：`ReLU` 激活函数的神经元的偏置设置为一个小的正数，从而避免`ReLU` 初始时就位于饱和的区域。
   - 有时某个单元作为开关来决定其他单元是使用还是不使用。此时偏置应该非零，从而打开开关。

十、Normalization

10.1 batch normalization

1. `batch normalization`是优化神经网络的一大创新。
   - 它并不是一个优化算法，而是一个自适应的、调整参数模型的方法。
   - 它试图解决训练非常深的神经网络的困难。
2. 深度神经网络训练困难的一个重要原因是：深度神经网络涉及很多层的叠加，而每一层的参数更新会导致上一层的输入数据分布发生变化。这会带来两个问题：
   - 下层输入的变化可能趋向于变大或者变小，导致上层落入饱和区，使得学习过早停止。
   - 通过层层叠加，高层的输入分布变化会非常剧烈。这就使得高层需要不断去适应底层的参数更新变化。

     这就要求我们需要非常谨慎的设定学习率、初始化权重、参数更新策略。

10.1.1 白化

1. 在机器学习中，如果数据是独立同分布的，则可以简化模型的训练，提升模型的预测能力。所以通常需要对输入数据进行白化`whitening`。 白化主要实现两个目的：
   - 去除特征之间的相关性。即：特征之间尽可能的独立。
   - 使得所有特征都具有相同的均值和方差。即：特征之间尽可能的同分布。
2. 白化操作：
   - 首先将输入执行 `PCA` 降维，这称作`PCA` 处理。
   - 然后在新的空间中，对输入数据的每一维进行标准差归一化处理。
3. 理论上可以对神经网络的每一层的输入执行白化来解决输入数据分布的问题。但是有两个困难：
   - 白化操作代价高昂，算法复杂度太大。因为`PCA` 处理涉及到协方差矩阵 $\mathbf{X}^T\mathbf{X}$ 的特征值求解，而计算协方差矩阵的算法复杂度为 $O(N^2\times n^2)$（不考虑`Strassen` 算法优化），其中 N 为数据集样本数量，n 为特征数量。

     对于神经网络的训练集，通常 N 都数以万计甚至百万计，如果在每一层、每一次参数更新都执行白化操作，则不可接受。

- 白化操作不可微，这样反向传播算法无法进行。

因此`batch normalization`就退而求其次，执行简化版的白化：将神经网络的每一层的输入的分布限定其均值和方差。

10.1.2 深层网络的参数更新

1. 对于一个深层的神经网络，如果同时更新所有层的参数，则可能会发生一些意想不到的后果。

   假设有一个深层神经网络，一共有 l 层，每层只有一个单元，且每个隐层不使用激励函数。则输出为：

   $$\hat{y} = xw_1w_2\cdots w_l$$

   其中 w_i 为第 i 层的权重。第 i 层的输出为： $h^{<i>} = h^{<i-1>}w_i$。

   令 $\vec{\mathbf{g}} = (g_1, g_2, \cdots, g_l)^T = \nabla_{\vec{\mathbf{w}}}\hat{y}$ ，其中：

   $$g_i = \frac{\partial \hat{y}}{w_i} = x\prod_{j=1,j\neq i}^{l} w_j$$

   利用梯度下降法更新参数，则有：$\vec{\mathbf{w}} \leftarrow \vec{\mathbf{w}} - \epsilon\vec{\mathbf{g}}$

2. 如果使用 $\hat{y}$ 的一阶泰勒近似，则有：$f(\vec{\mathbf{w}} - \epsilon\vec{\mathbf{g}}) - f(\vec{\mathbf{w}}) \approx -\epsilon\vec{\mathbf{g}}^T\vec{\mathbf{g}}$ 。即：$\hat{y}$ 的值下降了 $\epsilon\vec{\mathbf{g}}^T\vec{\mathbf{g}}$。因此梯度下降法一定能够降低 $\hat{y}$ 的值。

   如果直接按多项式乘法展开，则会考虑 ϵ 的二阶、三阶甚至更高阶的项，有：

   $$\begin{aligned} f(\vec{\mathbf{w}} - \epsilon\vec{\mathbf{g}}) - f(\vec{\mathbf{w}}) &= x(w_1 - \epsilon g_1)(w_2 - \epsilon g_2)\cdots(w_l - \epsilon g_l) - xw_1w_2\cdots w_l \\ &= -\epsilon x\sum_{i=1}^{l}\left(g_i\prod_{j=1,j\neq i}^{l} w_j\right) + \epsilon^2 x\sum_{i=1}^{l}\sum_{j=i}^{l}\left(g_ig_j\prod_{k=1,k\neq i,k\neq j}^{l} w_k\right) + \cdots \end{aligned}$$

   考虑到 $g_i = x\prod_{j=1,j\neq i}^{l} w_j$，则有：

   $$f(\vec{\mathbf{w}} - \epsilon\vec{\mathbf{g}}) - f(\vec{\mathbf{w}}) = -\epsilon\vec{\mathbf{g}}^T\vec{\mathbf{g}} + \epsilon^2 x\sum_{i=1}^{l}\sum_{j=i}^{l}\left(g_ig_j\prod_{k=1,k\neq i,k\neq j}^{l} w_k\right) + \cdots$$

   - 如果 w_i 都比较小，则 $\prod_{k=1,k\neq i,k\neq j}^{l} w_k$ 很小，则二阶项可以忽略不计。

     如果 w_i 都比较大，则该二阶项可能会指数级大。此时很难选择一个合适的学习率，使得 $f(\vec{\mathbf{w}} - \epsilon\vec{\mathbf{g}}) - f(\vec{\mathbf{w}}) < 0$ 。

     因此某一层中参数更新的效果会取决于其他所有层（即：其它层的权重是不是较大）。
   - 虽然二阶优化算法会利用二阶项的相互作用来解决这个问题，但是还有三阶项甚至更高阶项的影响。

10.1.3 BN 算法

1. `batch normalization`解决了多层之间协调更新的问题，它可以应用于网络的任何输入层或者隐层。
2. 设 $\mathbb{H}_{mini-batch} = \{\vec{\mathbf{h}}_1, \vec{\mathbf{h}}_2, \cdots, \vec{\mathbf{h}}_m\}$ 为神经网络某层的一个`mini-batch`的输入， n 为输入的维度。
   - 首先计算这个`mini-batch`输入的均值和每维特征的标准差：

     $$\vec{\mu} = \frac{1}{m}\sum_{i=1}^{m}\vec{\mathbf{h}}_i$$

     $$\vec{\sigma}^2 = (\sigma_1^2, \sigma_2^2, \cdots, \sigma_n^2)^T, \quad \sigma_j^2 = \frac{1}{m}\sum_{i=1}^{m}(h_{i,j} - \mu_j)^2$$

- 然后对输入进行归一化：

$$\vec{\mathbf{h}}_i^{<1>} = \frac{\vec{\mathbf{h}}_i - \vec{\mu}}{\sqrt{\vec{\sigma}^2 + \epsilon}}$$

其中 $\frac{1}{\sqrt{\vec{\sigma}^2+\epsilon}}$ 表示逐元素的除法： $h_{i,j}^{<1>} = \frac{h_{i,j}-\mu_j}{\sqrt{\sigma_j^2+\epsilon}}, j = 1,2,\cdots,n$ 。

- 最后执行缩放：$\vec{\mathbf{h}}_i^{<2>} = \vec{\gamma} \odot \vec{\mathbf{h}}_i^{<1>} + \vec{\beta}$ 。其中 $\vec{\gamma}, \vec{\beta}$ 是网络从数据中自动学习到的参数，用于调整 $\vec{\mathbf{h}}_i^{<2>}$ 的均值和方差， $\odot$ 为逐元素积。

  虽然 $\vec{\mathbf{h}}_i^{<2>}$ 的每个维度不是零均值、单位方差的，但是可以保证它的每个维度的均值、方差不再依赖于低层的网络。

3. 归一化一个神经元的均值和标准差会降低包含该神经元的神经网络的表达能力。

   若每个神经元的输出都是均值为0、标准差为 1 ，则会产生两个问题：

   - 无论底层的神经元如何学习 ，其输出在提交给上层神经元处理之前，都被粗暴的归一化。导致底层神经元的学习毫无意义。
   - `sigmoid` 等激活函数通过区分饱和区、非饱和区（线性区），使得神经网络具有非线性计算的能力。

     输入归一化使得数据几乎都被映射到激活函数的线性区，从而降低了模型的表达能力。

   因此执行缩放的原因是：保证模型的容量不会被降低。

   当网络学到的参数且好是 $\vec{\gamma} = \sqrt{\vec{\sigma}^2 + \epsilon}, \vec{\beta} = \vec{\mu}$ 时， $\vec{\mathbf{h}}_i^{<2>} = \vec{\mathbf{h}}_i$ ，因此 `BN` 可以还原原来的输入。这样，模型既可以改变、也可以保持原输入，这就提升了网络的表达能力。

4. `batch normalization` 算法

   - 输入：
     - 网络中某层的一个 `mini-batch` 的输入 $\mathbb{H}_{mini-batch} = \{\vec{\mathbf{h}}_1, \vec{\mathbf{h}}_2, \cdots, \vec{\mathbf{h}}_m\}$
     - 参数 $\vec{\gamma}, \vec{\beta}$ （它们是由神经网络自动学习到的）
   - 输出：经过 `batch normalization` 得到的新的输入 $\mathbb{H}_{mini-batch}^{<2>} = \{\vec{\mathbf{h}}_1^{<2>}, \vec{\mathbf{h}}_2^{<2>}, \cdots, \vec{\mathbf{h}}_m^{<2>}\}$
   - 算法步骤：
     - 计算输入的每个维度的均值和方差：

$$\vec{\mu} = \frac{1}{m}\sum_{i=1}^{m}\vec{\mathbf{h}}_i$$

$$\vec{\sigma}^2 = (\sigma_1^2, \sigma_2^2, \cdots, \sigma_n^2)^T, \quad \sigma_j^2 = \frac{1}{m}\sum_{i=1}^{m}(h_{i,j} - \mu_j)^2$$

     - 对输入的每个维度执行归一化：

$$\vec{\mathbf{h}}_i^{<1>} = \frac{\vec{\mathbf{h}}_i - \vec{\mu}}{\sqrt{\vec{\sigma}^2 + \epsilon}}$$

     - 执行缩放：$\vec{\mathbf{h}}_i^{<2>} = \vec{\gamma} \odot \vec{\mathbf{h}}_i^{<1>} + \vec{\beta}$

5. 根据梯度的链式法则，反向传播规则为（假设代价函数为 $\mathcal{L}$ ）：

   - $\nabla_{\vec{\mathbf{h}}_i^{<1>}}\mathcal{L} = \vec{\gamma} \odot \nabla_{\vec{\mathbf{h}}_i^{<2>}}\mathcal{L}$

- 考虑到 $\vec{\gamma},\vec{\beta}$ 出现在 $\vec{\mathbf{h}}_1^{<2>},\cdots,\vec{\mathbf{h}}_m^{<2>}$ 中，因此有：

$$\nabla_{\vec{\beta}}\mathcal{L}=\sum_{i=1}^{m}\nabla_{\vec{\mathbf{h}}_i^{<2>}}\mathcal{L},\quad \nabla_{\vec{\gamma}}\mathcal{L}=\sum_{i=1}^{m}(\nabla_{\vec{\mathbf{h}}_i^{<2>}}\mathcal{L})\odot\vec{\mathbf{h}}_i^{<1>}$$

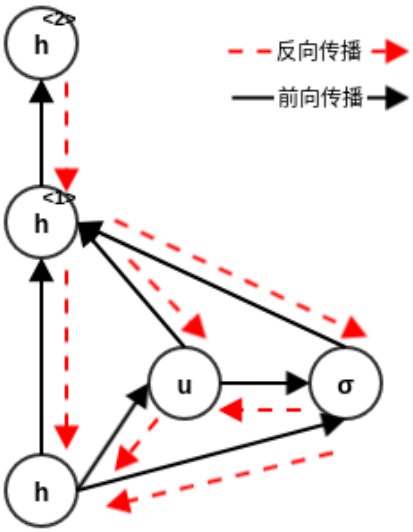
- 由于 $\vec{\mu},\vec{\sigma}^2$ 出现在 $\vec{\mathbf{h}}_1^{<2>},\cdots,\vec{\mathbf{h}}_m^{<2>}$ 中，因此有：

$$\nabla_{\vec{\mu}}\mathcal{L}=\sum_{i=1}^{m}\left(\frac{\partial\vec{\mathbf{h}}_i^{<1>}}{\partial\vec{\mu}}\right)^T\nabla_{\vec{\mathbf{h}}_i^{<1>}}\mathcal{L}=\sum_{i=1}^{m}-\frac{\nabla_{\vec{\mathbf{h}}_i^{<1>}}\mathcal{L}}{\sqrt{\vec{\sigma}^2+\epsilon}}$$

$$\nabla_{\vec{\sigma}^2}\mathcal{L}=\sum_{i=1}^{m}\left(\frac{\partial\vec{\mathbf{h}}_i^{<1>}}{\partial\vec{\sigma}^2}\right)^T\nabla_{\vec{\mathbf{h}}_i^{<1>}}\mathcal{L}=\sum_{i=1}^{m}-\frac{1}{2}\frac{\vec{\mathbf{h}}_i-\vec{\mu}}{(\vec{\sigma}^2+\epsilon)^{3/2}}\odot(\nabla_{\vec{\mathbf{h}}_i^{<1>}}\mathcal{L})$$

- 由于 $\vec{\mathbf{h}}_i$ 出现在多条路径中，因此有：

$$\begin{aligned}\nabla_{\vec{\mathbf{h}}_i}\mathcal{L}&=\left(\frac{\partial\vec{\mathbf{h}}_i^{<1>}}{\partial\vec{\mathbf{h}}_i}\right)^T\nabla_{\vec{\mathbf{h}}_i^{<1>}}\mathcal{L}+\left(\frac{\partial\vec{\mu}}{\partial\vec{\mathbf{h}}_i}\right)^T\nabla_{\vec{\mu}}\mathcal{L}+\left(\frac{\partial\vec{\sigma}^2}{\partial\vec{\mathbf{h}}_i}\right)^T\nabla_{\vec{\sigma}^2}\mathcal{L}\\&=\frac{\nabla_{\vec{\mathbf{h}}_i^{<1>}}\mathcal{L}}{\sqrt{\vec{\sigma}^2+\epsilon}}+\frac{\nabla_{\vec{\mu}}\mathcal{L}}{m}+\frac{2}{m}(\vec{\mathbf{h}}_i-\vec{\mu})\odot\nabla_{\vec{\sigma}^2}\mathcal{L}\end{aligned}$$



6. 大多数神经网络隐层采用 $\phi(\mathbf{X}\mathbf{W}+\vec{\mathbf{b}})$ 的形式，其中 ϕ 是非线性激励函数（如 `relu` ）。

   在 `batch normalization` 中推荐使用 $\phi(\mathbf{X}\mathbf{W})$ ，因为参数 $\vec{\mathbf{b}}$ 会被 `batch normalization` 中的参数 $\vec{\beta}$ 吸收：无论 $\vec{\mathbf{b}}$ 的值是多少，在归一化的过程中它将被减去。

10.1.4 BN 内部原理

1. `BN` 表现良好的一个解释是：内部协方差偏移 `Internal Covariate Shift:ICS` 会对训练产生负面影响，`BN` 能够减少 `ICS` 。
2. 内部协方差偏移：低层网络的更新对当前层输入分布造成了改变。

统计学习中一个经典假设是源空间`source domain`和目标空间`target domain`的数据分布一致。协方差偏移`covariate shift`就是分布不一致假设之下的一个分支问题。它指的是：源空间和目标空间的条件概率是一致的，但是其边缘概率不同。即：对所有的 $x \in \mathcal{X}$，有 $P_s(Y \mid X=x)=P_t(Y \mid X=x)$，但是 $P_s(X) \neq P_t(X)$。

在神经网络中经过各层的作用，各层输出与其输入分布会不同。这种差异随着网络深度的增大而增大，但是它们能够"指示"的样本标记仍然不变，这就符合`covariate shift`的定义。

由于是对层间信号的分析，这就是`internal`的由来。

`ICS`带来的问题是：各个神经元的输入数据不再是独立同分布的。

- 上层参数需要不断适应新的输入数据分布，降低了学习速度。
- 下层输入的变化可能趋向于变大或者变小，导致上层落入饱和区，使得学习过早停止。
- 每层的更新都会影响到其它层，因此每层的参数更新策略需要尽可能的谨慎。

3. 论文`《How Does Batch Normalization Help Optimization》 2018 Shibani Santurkar etc.`说明`BN`对训练带来的增益与`ICS`的减少没有任何联系，或者说这种联系非常脆弱。研究发现：`BN`甚至不会减少`ICS`。

   论文说明`BN`的成功的真正原因是：它使得优化问题的解空间更加平滑了。这确保梯度更具有预测性，从而允许使用更大范围的学习率，实现更快的网络收敛。

10.1.5 BN 性质

1. `BN`独立地归一化每个输入维度，它要求每个`mini batch`的统计量是整体统计量的近似估计。

   因此`BN`要求每个`mini-batch`都比较大，而且每个`mini-batch`的数据分布比较接近。所以在训练之前，需要对数据集进行充分混洗，否则效果可能很差。

2. 当验证或者测试的`batch size`较小时（如：只有一个测试样本），此时无法得到`mini batch`的统计量，或者`mini batch`统计量无法作为整体统计量的近似估计。

   此时的做法是：先通过训练集上得到的所有`mini batch`的统计量的移动平均值，然后将它们作为验证或者测试时的`mini batch`的统计量。

   但是当训练数据、验证数据、测试数据的数据分布存在差别时（如：训练数据从网上爬取的高清图片，测试数据是手机拍照的图片），训练集上预先计算好的`mini batch`的统计量的移动平均值并不能代表验证集、测试集的相应的统计量。这就导致了训练、验证、测试三个阶段存在不一致性。

3. `BN`存在两个明显不足：

   - 高度依赖于`mini batch`的大小。它要求每个`mini-batch`都比较大，因此不适合`batch size`较小的场景，如：在线学习（`batch size=1`）。
   - 不适合`RNN`网络。

     因为不同样本的`sequence`的长度不同，因此`RNN`的深度是不固定的。同一个`batch`中的多个样本会产生不同深度的`RNN`，因此很难对同一层的样本进行归一化。

4. 设 $\vec{\mathbf{h}}_i=\mathbf{W}\vec{\mathbf{x}}_i$，则`BN`具有权重伸缩不变性，以及数据伸缩不变性。

   - 权重伸缩不变性：假设 $\tilde{\mathbf{W}}=\lambda\mathbf{W}$，则有：

     $$\tilde{\vec{\mathbf{h}}}_i=\lambda\vec{\mathbf{h}}_i,\quad \tilde{\vec{\mu}}=\lambda\vec{\mu},\quad \tilde{\vec{\sigma}}_2=\lambda^2\vec{\sigma}_2$$

     其中因为 ϵ 很小几乎可以忽略不计，因此有 $\tilde{\vec{\mathbf{h}}}_i^{<1>}=\vec{\mathbf{h}}_i^{<1>}$，则有：$\nabla_{\tilde{\vec{\mathbf{h}}}_i^{<1>}}\mathcal{L}=\nabla_{\vec{\mathbf{h}}_i^{<1>}}\mathcal{L}$。

$$\frac{\partial\tilde{\vec{\mathbf{h}}}_i^{<1>}}{\partial\vec{\mathbf{x}}_i}=\frac{\tilde{\mathbf{W}}}{\sqrt{\tilde{\vec{\sigma}}^2+\epsilon}}=\frac{\mathbf{W}}{\sqrt{\vec{\sigma}^2+\epsilon}}=\frac{\partial\vec{\mathbf{h}}_i^{<1>}}{\partial\vec{\mathbf{x}}_i}$$

因此权重缩放前后， $\nabla_{\vec{\mathbf{x}}_i}\mathcal{L}$ 保持不变。 $\vec{\mathbf{x}}_i$ 是 `BN` 层的输入， $\nabla_{\vec{\mathbf{x}}_i}\mathcal{L}$ 就是高层流向低层的梯度，因此权重缩放不影响梯度的流动。

另外，由于 $\frac{\partial\tilde{\vec{\mathbf{h}}}_i^{<1>}}{\partial\tilde{\mathbf{W}}}=\frac{1}{\lambda}\frac{\partial\vec{\mathbf{h}}_i^{<1>}}{\partial\mathbf{W}}$ ，因此权重越大，则该权重的梯度越小，这样权重更新就越稳定。这相当与实现了参数正则化的效果，避免了参数的大幅震荡。

> 通常并不会把 `batch normalization` 当作一种正则化的手段，而是作为加速学习的方式。

- 数据伸缩不变性：假设 $\tilde{\vec{\mathbf{x}}}_i=\lambda\vec{\mathbf{x}}_i$ ，同理有：

$$\frac{\partial\tilde{\vec{\mathbf{h}}}_i^{<1>}}{\partial\mathbf{W}}=\frac{\partial\vec{\mathbf{h}}_i^{<1>}}{\partial\mathbf{W}},\quad\frac{\partial\tilde{\vec{\mathbf{h}}}_i^{<1>}}{\partial\tilde{\vec{\mathbf{x}}}_i}=\frac{1}{\lambda}\frac{\partial\vec{\mathbf{h}}_i^{<1>}}{\partial\vec{\mathbf{x}}_i}$$

  因此数据的伸缩变化不会影响到对该层的权重更新，简化了对学习率的选择。

5. 究竟是在激活函数之前、还是之后进行 `batch normalization` ， 这个问题在文献中有一些争论。

   实践中，通常都是在激活函数之前进行的。

6. 在测试阶段，如果需要对单一样本评估，此时测试集只有单个样本，无法给出均值和标准差。

   解决的方式为：将 $\vec{\mu},\vec{\sigma}$ 设置为训练阶段收集的运行均值（或者是指数加权均值）。

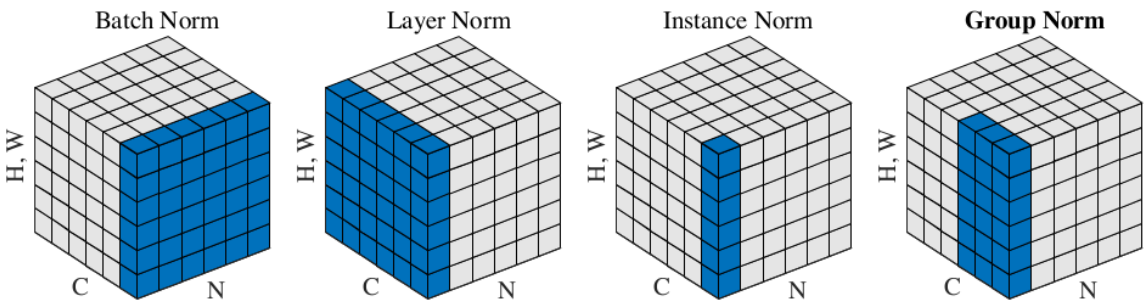
10.2 其它 normalization

1. 除了 `batch normalization` 之外，还有 `layer normalization、 instance normalization、group normalization、 weight normalization` 。

   下图给出了 `BN、LN、IN、GN` 的示意图（出自论文 `《Group Normalization》 Kaiming He etc.` ）。其中蓝色部分表示：通过这些蓝色区域计算均值和方差，然后蓝色区域中的每个单元都使用这些均值、方差来归一化。

   注意：这里的 `BN` 是网络某层中，对每个通道进行归一化；而前面的 `BN` 是对每个神经元进行归一化。

   如果是对每个神经元进行归一化，则 `BN` 示意图中，蓝色区域只有最底下的一行。



10.2.1 layer normalization

1. 与 `BN` 不同， `LN` 是对单个样本的同一层的神经元进行归一化，同层神经元使用相同的均值和方差。

   对于该层神经元，不同样本可以使用的均值和方差不同。

与之相比，`BN` 是对每个神经元在`mini batch` 样本之间计算均值和方差。对每个神经元，`mini batch` 中的所有样本在该神经元上都使用相同的均值和方差。但是不同神经元使用不同的均值和方差。

因此`LN` 不依赖于`batch size` ，也不依赖于网络深度。因此它适合在线学习，也适合于`RNN` 网络。

2. 设神经网络第 l 层的输入为 $\vec{\mathbf{h}}$ ， $\vec{\mathbf{h}}=(h_1,\cdots,h_n)^T$ ，n 为该层神经元的数量。则`LN` 的步骤为：
   - 首先计算该层所有神经元的均值和方差：
     $$\mu=\frac{1}{n}\sum_{i=1}^{n}h_i,\quad \sigma^2=\frac{1}{n}\sum_{i=1}^{n}(h_i-\mu)^2$$
   - 然后对神经元进行归一化：
     $$\vec{\mathbf{h}}^{<1>}=\frac{\vec{\mathbf{h}}-\mu}{\sqrt{\sigma^2+\epsilon}}$$
     其中 μ,σ^2 都是标量。
   - 最后执行缩放：$\vec{\mathbf{h}}^{<2>}=\vec{\gamma}\odot\vec{\mathbf{h}}^{<1>}+\vec{\beta}$ 。

     与 `BN` 相同， $\vec{\gamma},\vec{\beta}$ 也是网络从数据中自动学习到的参数，用于调整 $\vec{\mathbf{h}}^{<2>}$ 的均值和方差， $\odot$ 为逐元素积。

     这一步的作用也是提升神经网络的表达能力。
3. `layer normalization` 算法
   - 输入：
     - 网络中第 l 层的输入 $\vec{\mathbf{h}}$
     - 参数 $\vec{\gamma},\vec{\beta}$ （它们是由神经网络自动学习到的）
   - 输出：经过`layer normalization` 得到的新的输入 $\vec{\mathbf{h}}^{<2>}$
   - 算法步骤：
     - 计算该层所有神经元的均值和方差：
       $$\mu=\frac{1}{n}\sum_{i=1}^{n}h_i,\quad \sigma^2=\frac{1}{n}\sum_{i=1}^{n}(h_i-\mu)^2$$
     - 对该层神经元进行归一化：
       $$\vec{\mathbf{h}}^{<1>}=\frac{\vec{\mathbf{h}}-\mu}{\sqrt{\sigma^2+\epsilon}}$$
     - 执行缩放：$\vec{\mathbf{h}}^{<2>}=\vec{\gamma}\odot\vec{\mathbf{h}}^{<1>}+\vec{\beta}$
4. 根据梯度的链式法则，反向传播规则为（假设代价函数为 $\mathcal{L}$ ）：

$$\nabla_{\vec{\mathbf{h}}^{<1>}}\mathcal{L}=\vec{\gamma}\odot\nabla_{\vec{\mathbf{h}}^{<2>}}\mathcal{L}$$
$$\nabla_{\vec{\beta}}\mathcal{L}=\nabla_{\vec{\mathbf{h}}^{<2>}}\mathcal{L}$$
$$\nabla_{\vec{\gamma}}\mathcal{L}=(\nabla_{\vec{\mathbf{h}}^{<2>}}\mathcal{L})\odot\vec{\mathbf{h}}^{<1>}$$
$$\nabla_{\mu}\mathcal{L}=-\frac{1}{\sqrt{\sigma^2+\epsilon}}\sum_{i=1}^{n}\frac{\partial\mathcal{L}}{\partial h_i^{<1>}}$$
$$\nabla_{\sigma^2}\mathcal{L}=-\frac{1}{2}\times\frac{1}{(\sigma^2+\epsilon)^{3/2}}\sum_{i=1}^{n}\frac{\partial\mathcal{L}}{\partial h_i^{<1>}}\times(h_i-\mu)$$
$$=\frac{-1}{2(\sigma^2+\epsilon)^{3/2}}(\nabla_{\vec{\mathbf{h}}^{<1>}}\mathcal{L})^T(\vec{\mathbf{h}}-\mu)$$

由于 $\vec{\mathbf{h}}$ 出现在多条路径中，因此有：

$$\nabla_{\vec{\mathbf{h}}}\mathcal{L}=\left(\frac{\partial\vec{\mathbf{h}}^{<1>}}{\partial\vec{\mathbf{h}}}\right)^T\nabla_{\vec{\mathbf{h}}^{<1>}}\mathcal{L}+\left(\frac{\partial\mu}{\partial\vec{\mathbf{h}}}\right)\nabla_{\mu}\mathcal{L}+\left(\frac{\partial\sigma^2}{\partial\vec{\mathbf{h}}}\right)\nabla_{\sigma^2}\mathcal{L}$$
$$=\frac{\nabla_{\vec{\mathbf{h}}^{<1>}}\mathcal{L}}{\sqrt{\sigma^2+\epsilon}}+\frac{\nabla_{\mu}\mathcal{L}}{n}+\frac{2(\nabla_{\sigma^2}\mathcal{L})\times(\vec{\mathbf{h}}-\mu)}{n}$$

其中出现标量与向量的加法，它等价于将标量扩充为向量，扩充向量在每个维度上的取值就是该标量。如：$\frac{1}{n}\rightarrow(\frac{1}{n},\frac{1}{n},\cdots,\frac{1}{n})^T$ 。

其计算图如下所示，与 `BN` 相同。

`LN` 的特点是：针对单个样本进行，不依赖于 `mini batch` 中的其它样本。

10.2.2 instance normalization

1. 与 `IN` 与 `LN` 相同，它们都是对单个样本进行操作。与 `LN` 不同的是：`IN` 对同一层神经元中的同一个通道进行归一化。

   `IN` 主要用于图像处理任务中，此时每一层网络都有 `N、H、W、C` 四个维度。其中 `N` 代表 `batch` 维度，`H、W` 代表 `feature map` 的宽度和高度， `C` 代表通道数量。

   `LN` 使得同一层神经元中的同一个通道上的神经元使用相同的均值和方差。对于该通道中的神经元，不同的样本使用的均值和方差不同。

2. 设单张图片在网络第 l 层的输入张量为 $\mathbf{X}=(X_{i,j,k})\in\mathbb{R}^{C\times H\times W}$。 为了防止名字冲突，这里用 $\mathbf{X}$ 标记第 l 层的输入。其中 C 为通道数，H,W 为 `feature map` 的高度和宽度。

   三个索引分别代表： i 代表通道维的索引，j,k 分别代表高度和宽度维度的索引。

   则 `LN` 的步骤为：

   - 首先计算样本在第 i 个通道的神经元的均值和方差：

   $$\mu_i=\frac{1}{H\times W}\sum_{j=1}^{H}\sum_{k=1}^{W}X_{i,j,k},\quad\sigma_i^2=\frac{1}{H\times W}\sum_{j=1}^{H}\sum_{k=1}^{W}(X_{i,j,k}-\mu_i)^2$$

   - 然后对神经元进行归一化：

   $$X_{i,j,k}^{<1>}=\frac{X_{i,j,k}-\mu_i}{\sqrt{\sigma_i^2+\epsilon}},\quad j=1,\cdots,H;k=1,\cdots,W$$

- 最后执行缩放：

$$X_{i,j,k}^{<2>}=\gamma_{i,j,k}\times X_{i,j,k}^{<1>}+\beta_{i,j,k}$$

其中 $\gamma_{i,j,k},\beta_{i,j,k}$ 表示第 i 个通道位于 (j,k) 的神经元的缩放因子和平移因子。

3. `instance normalization` 算法
   - 输入：
     - 网络第 l 层的输入张量为 $\mathbf{X}=(X_{i,j,k})\in\mathbb{R}^{C\times H\times W}$
     - 参数张量 $(\gamma_{i,j,k}),(\beta_{i,j,k})\in\mathbb{R}^{C\times H\times W}$
   - 输出：经过 `instance normalization` 得到的新的输入 $\mathbf{X}^{<2>}$
   - 算法步骤：
     - 计算样本在第 i 个通道的神经元的均值和方差：

$$\mu_i=\frac{1}{H\times W}\sum_{j=1}^{H}\sum_{k=1}^{W}X_{i,j,k},\quad \sigma_i^2=\frac{1}{H\times W}\sum_{j=1}^{H}\sum_{k=1}^{W}(X_{i,j,k}-\mu_i)^2$$

     - 对神经元进行归一化：

$$X_{i,j,k}^{<1>}=\frac{X_{i,j,k}-\mu_i}{\sqrt{\sigma_i^2+\epsilon}},\quad j=1,\cdots,H;k=1,\cdots,W$$

     - 执行缩放：$X_{i,j,k}^{<2>}=\gamma_{i,j,k}\times X_{i,j,k}^{<1>}+\beta_{i,j,k}$ 。
4. `instance normalization` 的反向传播规则的推导类似 `layer normalization` 。
5. `BN` 对 `mini batch` 中所有图片求均值和方差，计算得到的统计量会受到 `mini batch` 中其它样本的影响。而 `IN` 是对单个图片求均值和方差，与其它样本无关。
   - 对于 `GAN` 、风格迁移这类任务上， `IN` 效果要优于 `BN` 。其普遍解释为：这类生成式方法，每张图片自己的风格比较独立，不应该与 `batch` 中其它图片产生太大联系。
   - `IN` 也不依赖于 `batch size` ，也不依赖于网络深度。因此它适合在线学习，也适合于 `RNN` 网络。

10.2.3 group normalization

1. `GN` 首先将通道分组。假设有 `C` 个通道，分成 `G` 个组，则：通道 `1,...,C/G` 为一个组，通道 `C/G+1,...,2C/G` 为一个组.... 。

   然后 `GN` 对每个通道组进行归一化。

   因此可以看到： `GN` 介于 `LN` 和 `IN` 之间。如果 `G=1` ，即只有一个分组，则 `GN` 就是 `LN` ；如果 `G=C` ，即每个通道构成一个组，则 `GN` 就是 `IN` 。
2. `group normalization` 算法
   - 输入：
     - 网络第 l 层的输入张量为 $\mathbf{X}=(X_{i,j,k})\in\mathbb{R}^{C\times H\times W}$
     - 分组的组数 G
     - 参数张量 $(\gamma_{i,j,k}),(\beta_{i,j,k})\in\mathbb{R}^{G\times H\times W}$
   - 输出：经过 `group normalization` 得到的新的输入 $\mathbf{X}^{<2>}$
   - 算法步骤：

- 计算每个分组的通道数 $n_G = \frac{C}{G}$
- 计算样本在第 i 个分组的神经元的均值和方差：

$$\mu_i = \frac{1}{n_G \times H \times W} \sum_{i'=(i-1)*n_G+1}^{i*n_G} \sum_{j=1}^{H} \sum_{k=1}^{W} X_{i',j,k}$$

$$\sigma_i^2 = \frac{1}{n_G \times H \times W} \sum_{i'=(i-1)*n_G+1}^{i*n_G} \sum_{j=1}^{H} \sum_{k=1}^{W} (X_{i',j,k} - \mu_i)^2$$

- 对神经元进行归一化：

$$X^{<1>}_{(i-1)*n_G+g,j,k} = \frac{X_{(i-1)*n_G+g,j,k} - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}}$$

$$j = 1, \cdots, H; k = 1, \cdots, W; i = 1, \cdots, G; g = 1, \cdots, n_G$$

- 执行缩放：$X^{<2>}_{i,j,k} = \gamma_{i,j,k} \times X^{<1>}_{i,j,k} + \beta_{i,j,k}$ 。

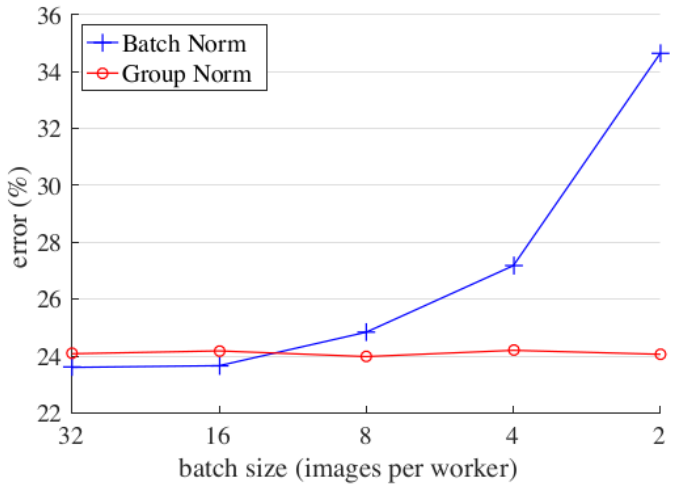
3. `GN` 有效的可能原因是：在网络的每一层中，多个卷积核学到的特征并不是完全独立的。某些特征具有类似的分布，因此可以被分到同一组。

10.2.4 比较

1. 根据论文 `《Group Normalization》 Kaiming He etc.` 得到的结论：
   - `BN` 很容易受到 `batch size` 的影响，而 `GN` 不容易受到 `batch size` 的影响。

     如下图/表 所示为 `BN` 和 `GN` 的比较，模型为 `resnet-50`、训练集为 `ImageNet`、训练硬件为8个带 `GPU` 的 `worker` 、指标为它们在验证集上的验证误差。

batch size	32	16	8	4	2
BN	23.6	23.7	24.8	27.3	34.7
GN	24.1	24.2	24.0	24.2	24.1
提升	0.5	0.5	-0.8	-3.1	-10.6



   - 在较大的 `batch size` 上，`BN` 的表现效果最好。

`resnet-50` 模型在 `ImageNet` 训练中，当采用 `batch size=32` 时，采取各种 `normalization` 的模型的表现（验证误差）为：

	BN	LN	IN	GN
验证误差	23.6	25.3	28.4	24.1
相比 BN 的提升	-	-1.7	-4.8	-0.5

下图为这些模型在各个 `epoch` 事训练、验证的表现：