# 1 优化算法

## 1.1 梯度下降

### 1.1.1 批梯度下降

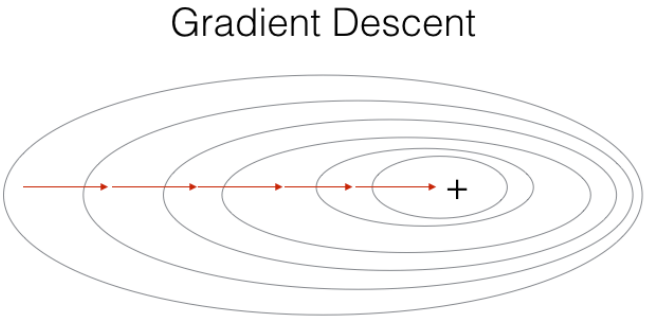
batch 梯度下降法（批梯度下降法，我们之前一直使用的梯度下降法）是最常用的梯度下降形式，即同时处理整个训练集。其在更新参数时使用所有的样本来进行更新。具体过程如下：

$$
\begin{aligned}
X &= [x^{(1)}, x^{(2)}, \dots, x^{(m)}] \\
z^{[1]} &= w^{[1]}X + b^{[1]} \\
a^{[1]} &= g^{[1]}(z^{[1]}) \\
&\cdots \quad \cdots \\
z^{[l]} &= w^{[l]}a^{[l-1]} + b^{[l]} \\
a^{[l]} &= g^{[l]}(z^{[l]})
\end{aligned} \tag{1}
$$

$$
J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\sum_{l=1}^{L}||w^{[l]}||_F^2
$$

$$
\theta_j := \theta_j - \alpha\frac{\partial J(\theta)}{\partial \theta_j}
$$

示意图如下：




- 优点：最小化所有训练样本的损失函数，得到全局最优解；易于并行实现
- 缺点：当样本数目很多时，训练过程会很慢

对整个训练集进行梯度下降法的时候，我们必须处理整个训练数据集，然后才能进行一步梯度下降，即每一步梯度下降法需要对整个训练集进行一次处理，如果训练数据集很大的时候，处理速度就会比较慢。

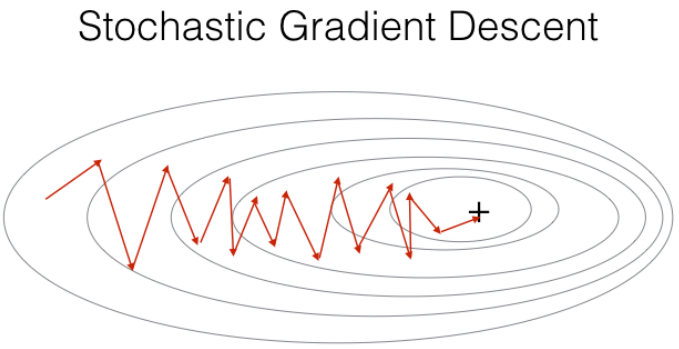
但是如果每次处理训练数据的一部分即进行梯度下降法，则我们的算法速度会执行的更快。而处理的这些一小部分训练子集即称为 mini-batch。

### 1.1.2 随机梯度下降

随机梯度下降法（Stochastic Gradient Descent，SGD）与批梯度下降原理类似，区别在于每次通过一个样本来迭代更新。其具体过程为：

$$
\begin{aligned}
X &= [x^{(1)}, x^{(2)}, \dots, x^{(m)}] \\
for \quad i &= 1, 2, \dots, m \{ \\
z^{[1]} &= w^{[1]} x^{(i)} + b^{[1]} \\
a^{[1]} &= g^{[1]}(z^{[1]}) \\
&\dots \quad \dots \\
z^{[l]} &= w^{[l]} a^{[l-1]} + b^{[l]} \\
a^{[l]} &= g^{[l]}(z^{[l]}) \\
J(\theta) &= \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2} \sum_{l=1}^{L} ||w^{[l]}||_F^2 \\
\theta_j &:= \theta_j - \alpha \frac{\partial J(\theta)}{\partial \theta_j} \}
\end{aligned} \tag{2}
$$

示意图如下：




- 优点：训练速度快。
- 缺点：最小化每条样本的损失函数，最终的结果往往是在全局最优解附近，不是全局最优；不易于并行实现。

### 1.1.3 小批量梯度下降

Mini-Batch 梯度下降法（小批量梯度下降法）每次同时处理单个的 mini-batch，其他与 batch 梯度下降法一致。

使用 batch 梯度下降法，对整个训练集的一次遍历只能做一个梯度下降；而使用 Mini-Batch 梯度下降法，对整个训练集的一次遍历（称为一个 epoch）能做 mini-batch 个数个梯度下降。之后，可以一直遍历训练集，直到最后收敛到一个合适的精度。具体过程为：

$$
X = [x^{\{1\}}, x^{\{2\}}, \dots, x^{\{k=\frac{m}{t}\}}] \tag{3}
$$

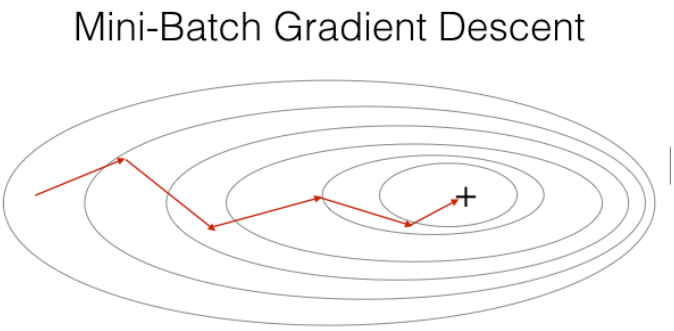
其中：

$$
\begin{aligned}
x^{\{1\}} &= x^{(1)}, x^{(2)}, \dots, x^{(t)} \\
x^{\{2\}} &= x^{(t+1)}, x^{(t+2)}, \dots, x^{(2t)} \\
&\dots \quad \dots
\end{aligned} \tag{4}
$$

之后：

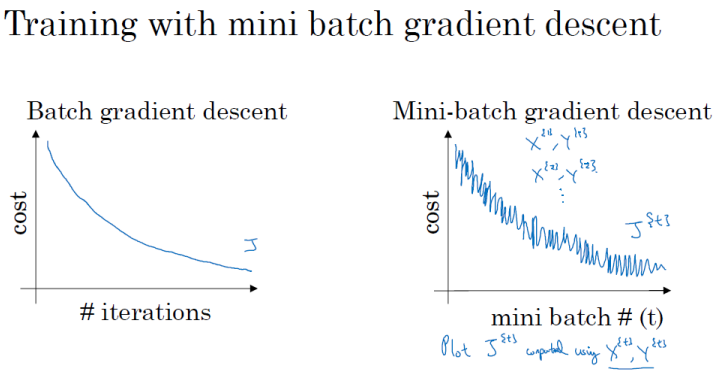
$$
\begin{aligned}
&for \quad i = 1, 2, \dots, k \ \{ \\
&\quad z^{[1]} = w^{[1]} x^{(i)} + b^{[1]} \\
&\quad a^{[1]} = g^{[1]}(z^{[1]}) \\
&\quad \dots \quad \dots \\
&\quad z^{[l]} = w^{[l]} a^{[l-1]} + b^{[l]} \\
&\quad a^{[l]} = g^{[l]}(z^{[l]}) \\
&J(\theta) = \frac{1}{k} \sum_{i=1}^{k} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2k} \sum_{l=1}^{L} ||w^{[l]}||_F^2 \\
&\quad \theta_j := \theta_j - \alpha \frac{\partial J(\theta)}{\partial \theta_j} \ \}
\end{aligned}
\tag{5}
$$

示意图如下：




**理解 Mini-batch**

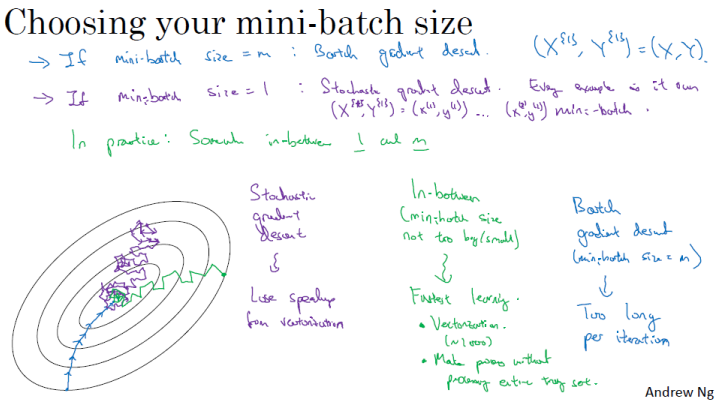
batch 梯度下降法和 Mini-batch 梯度下降法代价函数的变化趋势如下：




**batch 的不同大小**

- mini-batch 的大小为 1，即是随机梯度下降法（stochastic gradient descent），每个样本都是独立的 mini-batch；
- mini-batch 的大小为 m（数据集大小），即是 batch 梯度下降法；

下图是以上三者的比较：




**mini-batch 大小的选择**

- 如果训练样本的大小比较小，如 m ⩽ 2000 时，选择 batch 梯度下降法；
- 如果训练样本的大小比较大，选择 Mini-Batch 梯度下降法。为了和计算机的信息存储方式相适应，代码在 mini-batch 大小为 2 的幂次时运行要快一些。典型的大小为 $2^6$、$2^7$、...、$2^9$；
- mini-batch 的大小要符合 CPU/GPU 内存。

mini-batch 的大小也是一个重要的超变量，需要根据经验快速尝试，找到能够最有效地减少成本函数的值。

# 1.2 梯度下降优化

## 1.2.1 指数加权平均

指数加权平均（Exponentially Weight Average）是一种常用的序列数据处理方式，计算公式为：

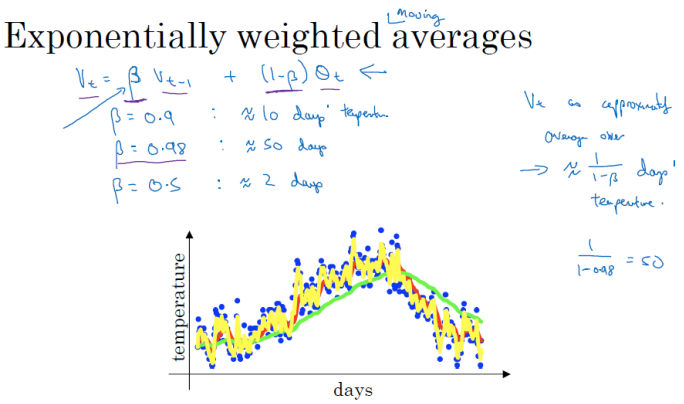
$$S_t = \begin{cases} Y_1, & t = 1 \\ \beta S_{t-1} + (1-\beta) Y_t, & t > 1 \end{cases} \tag{6}$$

其中 $Y_t$ 为 t 下的实际值，$Y_t$ 为 t 下加权平均后的值，β 为权重值。

指数加权平均数在统计学中被称为“指数加权移动平均值”。

**理解指数加权平均**

下面是反应温度变化的大致趋势的数据图：




给定一个时间序列，例如伦敦一年每天的气温值，图中蓝色的点代表真实数据。对于一个即时的气温值，取权重值 β 为 0.9，根据求得的值可以得到图中的红色曲线，它反映了气温变化的大致趋势。

当取权重值 β=0.98 时，可以得到图中更为平滑的绿色曲线。而当取权重值 β=0.5 时，得到图中噪点更多的黄色曲线。**β 越大相当于求取平均利用的天数越多**，曲线自然就会越平滑而且越滞后。

当 β 为 0.9 时，

$$
\begin{aligned}
v_0 &= 0 \\
v_1 &= 0.9v_0 + 0.1\theta_1 \\
&\cdots \quad \cdots \\
v_{100} &= 0.1\theta_{100} + 0.1 \times 0.9\theta_{99} + 0.1 \times 0.9^2\theta_{98} \ \ldots \\
v_t &= 0.9v_{t-1} + 0.1\theta_t
\end{aligned} \tag{7}
$$

其中 θi 指第 i 天的实际数据。所有 θ 前面的系数（不包括 0.1）相加起来为 1 或者接近于 1，这些系数被称作**偏差修正（Bias Correction）**

根据函数极限的一条定理：

$$
\lim_{\epsilon \to 0}(1-\epsilon)^{\frac{1}{\epsilon}} = \frac{1}{e} \approx 0.368 \tag{8}
$$

当 β 为 0.9 时，可以当作把过去 10 天的气温指数加权平均作为当日的气温，因为 10 天后权重已经下降到了当天的 1/3 左右。同理，当 β 为 0.98 时，可以把过去 50 天的气温指数加权平均作为当日的气温。

因此，在计算当前时刻的平均值时，只需要前一天的平均值和当前时刻的值：

$$
v_t = \beta v_{t-1} + (1-\beta)\theta_t \tag{9}
$$

考虑到代码，只需要不断更新 v 即可：

$$
v := \beta v + (1-\beta)\theta_t \tag{10}
$$

指数平均加权并不是最精准的计算平均数的方法，你可以直接计算过去 10 天或 50 天的平均值来得到更好的估计，但缺点是保存数据需要占用更多内存，执行更加复杂，计算成本更加高昂。

指数加权平均数公式的好处之一在于它只需要一行代码，且占用极少内存，因此效率极高，且节省成本。

**指数加权平均的偏差修正**

当进行指数加权平均计算时，第一个值 $v_0$ 被初始化为 0，这样将在前期的运算用产生一定的偏差。为了矫正偏差，需要在每一次迭代后用以下式子进行偏差修正：

$$
v_t := \frac{v_t}{1-\beta^t} \tag{11}
$$

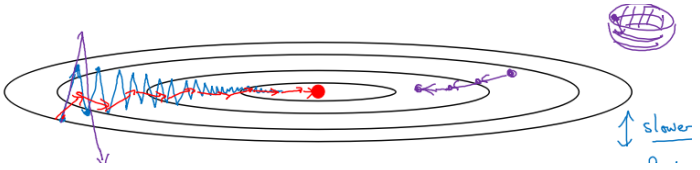
随着 t 的增大，β 的 t 次方趋近于 0。因此当 t 很大的时候，偏差修正几乎没有作用，但是在前期学习可以帮助更好的预测数据。在实际过程中，一般会忽略前期偏差的影响。

## 1.2.2 动量梯度下降

动量梯度下降（Gradient Descent with Momentum）是计算梯度的指数加权平均数，并利用该值来更新参数值。具体过程为：

$$
\begin{aligned}
v_{dw} &= \beta v_{dw} + (1-\beta)dw \\
v_{db} &= \beta v_{db} + (1-\beta)db \\
w &:= w - \alpha v_{dw} \\
b &:= b - \alpha v_{db}
\end{aligned} \tag{12}
$$

其中，将动量衰减参数 β 设置为 0.9 是超参数的一个常见且效果不错的选择。当 β 被设置为 0 时，显然就成了 batch 梯度下降法。




进行一般的梯度下降将会得到图中的蓝色曲线，由于存在上下波动，减缓了梯度下降的速度，因此只能使用一个较小的学习率进行迭代。如果用较大的学习率，结果可能会像紫色曲线一样偏离函数的范围。

而使用动量梯度下降时，通过累加过去的梯度值来减少抵达最小值路径上的波动，加速了收敛，因此在横轴方向下降得更快，从而得到图中红色的曲线。

当前后梯度方向一致时，动量梯度下降能够加速学习；而前后梯度方向不一致时，动量梯度下降能够抑制震荡。

我们不需要进行偏差修正，因为在 10 次迭代之后，移动平均已经不再是一个具有偏差的预测。

因此实际在使用梯度下降法或者动量梯度下降法时，不会同时进行偏差修正。

**形象理解**

将成本函数想象为一个碗状，从顶部开始运动的小球向下滚，其中 dw，db 想象成球的加速度；而 $v_{dw}$、$v_{db}$ 相当于速度。

小球在向下滚动的过程中，因为加速度的存在速度会变快，但是由于 β 的存在，其值小于 1，可以认为是摩擦力，所以球不会无限加速下去。

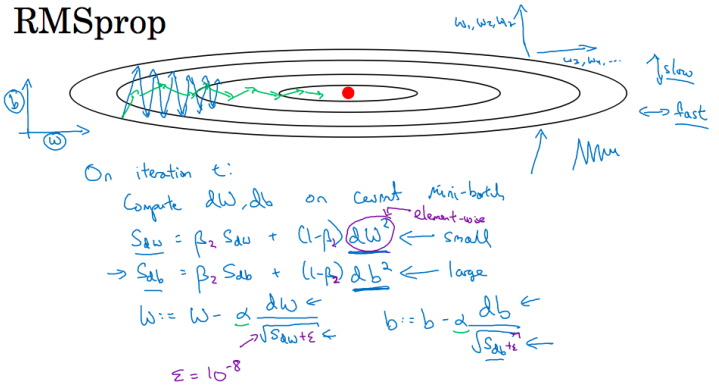
### 1.2.3 RMSProp 算法

RMSProp（Root Mean Square Prop，均方根支）算法是在对梯度进行指数加权平均的基础上，引入平方和平方根。具体过程为：

$$
\begin{aligned}
s_{dw} &= \beta s_{dw} + (1-\beta) dw^2 \\
s_{db} &= \beta s_{db} + (1-\beta) db^2 \\
w &:= w - \alpha \frac{dw}{\sqrt{s_{dw}+\epsilon}} \\
b &:= b - \alpha \frac{db}{\sqrt{s_{db}+\epsilon}}
\end{aligned}
\tag{13}
$$

其中，ϵ 是一个实际操作时加上的较小数（例如 $10^{-8}$），为了防止分母太小而导致的数值不稳定。

当 dw 或 db 较大时，$dw^2$、$db^2$ 会较大，造成 $s_{dw}$、 $s_{db}$ 也会较大，最终使 $\frac{dw}{\sqrt{s_{dw}}}$、$\frac{db}{\sqrt{s_{db}}}$ 较小，减小了抵达最小值路径上的摆动。




RMSProp 有助于减少抵达最小值路径上的摆动，并允许使用一个更大的学习率 α，从而加快算法学习速度。并且，它和 Adam 优化算法已被证明适用于不同的深度学习网络结构。

注意，β 也是一个超参数。

### 1.2.4 Adam 优化算法

Adam 优化算法（Adaptive Moment Estimation，自适应矩估计）基本上就是将 Momentum 和 RMSProp 算法结合在一起，通常有超越二者单独时的效果。具体过程如下：

首先进行初始化：

$$v_{dW}=0, s_{dW}=0, v_{db}=0, s_{db}=0 \tag{14}$$

用每一个 mini-batch 计算 dW、db，第 t 次迭代时：

$$\begin{aligned} v_{dW} &= \beta_1 v_{dW} + (1-\beta_1) dW \\ v_{db} &= \beta_1 v_{db} + (1-\beta_1) db \\ s_{dW} &= \beta_2 s_{dW} + (1-\beta_2)(dW)^2 \\ s_{db} &= \beta_2 s_{db} + (1-\beta_2)(db)^2 \end{aligned} \tag{15}$$

一般使用 Adam 算法时需要计算偏差修正：

$$\begin{aligned} v_{dW}^{corrected} &= \frac{v_{dW}}{1-{\beta_1}^t} \\ v_{db}^{corrected} &= \frac{v_{db}}{1-{\beta_1}^t} \\ s_{dW}^{corrected} &= \frac{s_{dW}}{1-{\beta_2}^t} \\ s_{db}^{corrected} &= \frac{s_{db}}{1-{\beta_2}^t} \end{aligned} \tag{16}$$

所以，更新 W、b 时有：

$$\begin{aligned} W &:= W - \alpha \frac{v_{dW}^{corrected}}{\sqrt{s_{dW}^{corrected}} + \epsilon} \\ b &:= b - \alpha \frac{v_{db}^{corrected}}{\sqrt{s_{db}^{corrected}} + \epsilon} \end{aligned} \tag{17}$$

**超参数的选择**

Adam 优化算法有很多的超参数，其中

- 学习率 α：需要尝试一系列的值，来寻找比较合适的；
- β1：常用的缺省值为 0.9；
- β2：Adam 算法的作者建议为 0.999；
- ϵ：不重要，不会影响算法表现，Adam 算法的作者建议为 $10^{-8}$；

β1、β2、ϵ 通常不需要调试。

### 1.2.5 学习率衰减

如果设置一个固定的学习率 α，在最小值点附近，由于不同的 batch 中存在一定的噪声，因此不会精确收敛，而是始终在最小值周围一个较大的范围内波动。

随着时间推移，慢慢减少学习率 α 的大小。在初期 α 较大时，迈出的步长较大，能以较快的速度进行梯度下降，而后期逐步减小 α 的值，减小步长，有助于算法的收敛，更容易接近最优解。

- 最常用的学习率衰减方法：

$$\alpha = \frac{1}{1 + decay\_rate * epoch\_num} * \alpha_0 \tag{18}$$

  其中，decay_rate 为衰减率（超参数），epoch_num 为将所有的训练样本完整过一遍的次数。

- 指数衰减：

$$\alpha = 0.95^{epoch\_num} * \alpha_0 \tag{19}$$

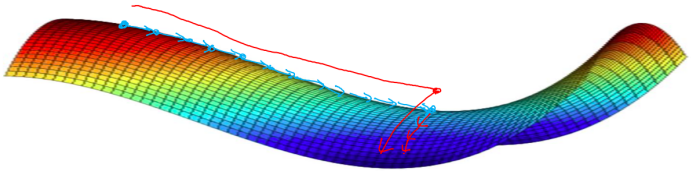
- 其他：

$$\alpha = \frac{k}{\sqrt{epoch\_num}} * \alpha_0 \tag{20}$$

- 离散下降

对于较小的模型，也有人会在训练时根据进度手动调小学习率。

## 1.3 局部最优问题



**鞍点（saddle）**是函数上的导数为零，但不是轴上局部极值的点。当我们建立一个神经网络时，通常梯度为零的点是上图所示的鞍点，而非局部最小值。减少损失的难度也来自误差曲面中的鞍点，而不是局部最低点。因为在一个具有高维度空间的成本函数中，如果梯度为 0，那么在每个方向，成本函数或是凸函数，或是凹函数。而所有维度均需要是凹函数的概率极小，因此在低维度的局部最优点的情况并不适用于高维度。

- 在训练较大的神经网络、存在大量参数，并且成本函数被定义在较高的维度空间时，困在极差的局部最优中是不大可能的；
- 鞍点附近的平稳段会使得学习非常缓慢，而这也是动量梯度下降法、RMSProp 以及 Adam 优化算法能够加速学习的原因，它们能帮助尽早走出平稳段。

## 1.4 超参数调试

### 1.4.1 超参数

目前已经讲到过的超参数中，重要程度依次是（仅供参考）：

**最重要：**

- 学习率 $\alpha$；

**其次重要：**

- $\beta$：动量衰减参数，常设置为 0.9；
- #hidden units：各隐藏层神经元个数；
- mini-batch 的大小；

**再次重要：**

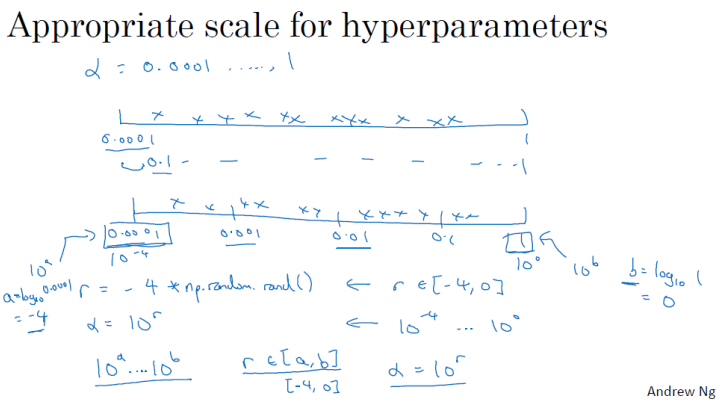
- $\beta_1$，$\beta_2$，$\epsilon$：Adam 优化算法的超参数，常设为 0.9、0.999、$10^{-8}$；
- #layers：神经网络层数；
- decay_rate：学习衰减率；

### 1.4.2 调参技巧

系统地组织超参调试过程的技巧：

- 随机选择点（而非均匀选取），用这些点实验超参数的效果。这样做的原因是我们提前很难知道超参数的重要程度，可以通过选择更多值来进行更多实验；
- 由粗糙到精细：聚焦效果不错的点组成的小区域，在其中更密集地取值，以此类推；

### 1.4.3 选择合适的范围

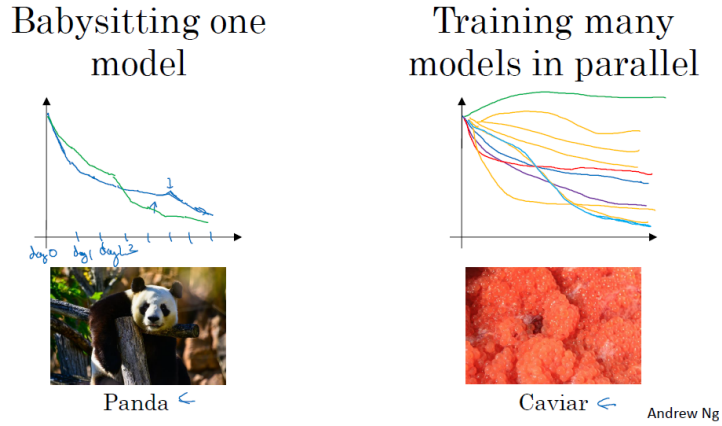



- 对于学习率 α，用对数标尺而非线性轴更加合理：0.0001、0.001、0.01、0.1 等，然后在这些刻度之间再随机均匀取值；
- 对于 β，取 0.9 就相当于在 10 个值中计算平均值，而取 0.999 就相当于在 1000 个值中计算平均值。可以考虑给 1-β 取值，这样就和取学习率类似了。

上述操作的原因是当 β 接近 1 时，即使 β 只有微小的改变，所得结果的灵敏度会有较大的变化。例如，β 从 0.9 增加到 0.9005 对结果（1/(1-β)）几乎没有影响，而 β 从 0.999 到 0.9995 对结果的影响巨大（从 1000 个值中计算平均值变为 2000 个值中计算平均值）。

### 1.4.4 一些建议

- 深度学习如今已经应用到许多不同的领域。不同的应用出现相互交融的现象，某个应用领域的超参数设定有可能通用于另一领域。不同应用领域的人也应该更多地阅读其他研究领域的 paper，跨领域地寻找灵感；
- 考虑到数据的变化或者服务器的变更等因素，建议每隔几个月至少一次，重新测试或评估超参数，来获得实时的最佳模型；
- 根据你所拥有的计算资源来决定你训练模型的方式：




- Panda（熊猫方式）：在在线广告设置或者在计算机视觉应用领域有大量的数据，但受计算能力所限，同时试验大量模型比较困难。可以采用这种方式：试验一个或一小批模型，初始化，试着让其工作运转，观察它的表现，不断调整参数；
- Caviar（鱼子酱方式）：拥有足够的计算机去平行试验很多模型，尝试很多不同的超参数，选取效果最好的模型；

## 1.5 批标准化

### 1.5.1 介绍

批标准化（Batch Normalization，经常简称为 BN）会使参数搜索问题变得很容易，使神经网络对超参数的选择更加稳定，超参数的范围会更庞大，工作效果也很好，也会使训练更容易。

之前，我们对输入特征 X 使用了标准化处理。我们也可以用同样的思路处理隐藏层的激活值 $a^{[l]}$，以加速 $W^{[l+1]}$ 和 $b^{[l+1]}$ 的训练。在实践中，经常选择标准化 $Z^{[l]}$：

$$\mu = \frac{1}{m}\sum_i z^{(i)} \tag{21}$$

$$\sigma^2 = \frac{1}{m}\sum_i (z_i - \mu)^2$$

$$z^{(i)}_{norm} = \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \epsilon}}$$

其中，m 是单个 mini-batch 所包含的样本个数，ε 是为了防止分母为零，保证数值稳定，通常取 $10^{-8}$。

这样，我们使得所有的输入 $z^{(i)}$ 均值为 0，方差为 1。但我们不想让隐藏层单元总是含有平均值 0 和方差 1，也许隐藏层单元有了不同的分布会更有意义。因此，我们计算：

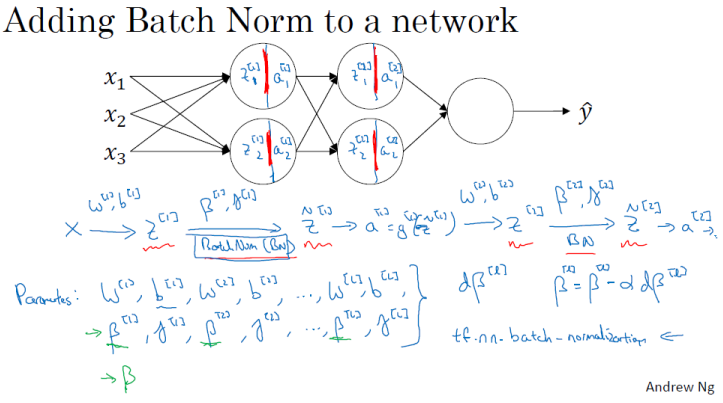
$$\tilde{z}^{(i)} = \gamma z^{(i)}_{norm} + \beta \tag{22}$$

其中，γ 和 β 都是模型的学习参数，所以可以用各种梯度下降算法来更新 γ 和 β 的值，如同更新神经网络的权重一样。

通过对 γ 和 β 的合理设置，可以让 $\tilde{z}^{(i)}$ 的均值和方差为任意值。这样，我们对隐藏层的 $z^{(i)}$ 进行标准化处理，用得到的 $\tilde{z}^{(i)}$ 替代 $z^{(i)}$ 。

设置 γ 和 β 的原因是，如果各隐藏层的输入均值在靠近 0 的区域，即处于激活函数的线性区域，不利于训练非线性神经网络，从而得到效果较差的模型。因此，需要用 γ 和 β 对标准化后的结果做进一步处理。例如当 $\gamma = \sqrt{\sigma^2 + \epsilon}, \beta = \mu$，就抵消掉了之前的正则化操作。

### 1.5.2 BN与神经网络




对于 L 层神经网络，经过 Batch Normalization 的作用，整体流程如下：

实际上，Batch Normalization 经常使用在 mini-batch 上，这也是其名称的由来。

下面是使用 BN 的前向传播和后向传播流程图：

**Input:** Values of $x$ over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: $\gamma$, $\beta$
**Output:** $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

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

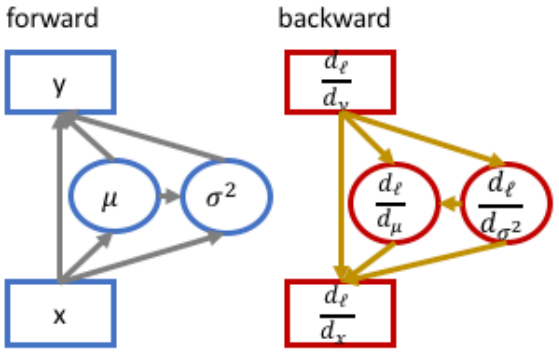
$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

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

$$\frac{\partial \ell}{\partial \widehat{x}_i} = \frac{\partial \ell}{\partial y_i} \cdot \gamma$$
$$\frac{\partial \ell}{\partial \sigma^2_{\mathcal{B}}} = \sum_{i=1}^{m} \frac{\partial \ell}{\partial \widehat{x}_i} \cdot (x_i - \mu_{\mathcal{B}}) \cdot \frac{-1}{2}(\sigma^2_{\mathcal{B}} + \epsilon)^{-3/2}$$
$$\frac{\partial \ell}{\partial \mu_{\mathcal{B}}} = \left(\sum_{i=1}^{m} \frac{\partial \ell}{\partial \widehat{x}_i} \cdot \frac{-1}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}}\right) + \frac{\partial \ell}{\partial \sigma^2_{\mathcal{B}}} \cdot \frac{\sum_{i=1}^{m} -2(x_i - \mu_{\mathcal{B}})}{m}$$
$$\frac{\partial \ell}{\partial x_i} = \frac{\partial \ell}{\partial \widehat{x}_i} \cdot \frac{1}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} + \frac{\partial \ell}{\partial \sigma^2_{\mathcal{B}}} \cdot \frac{2(x_i - \mu_{\mathcal{B}})}{m} + \frac{\partial \ell}{\partial \mu_{\mathcal{B}}} \cdot \frac{1}{m}$$
$$\frac{\partial \ell}{\partial \gamma} = \sum_{i=1}^{m} \frac{\partial \ell}{\partial y_i} \cdot \widehat{x}_i$$
$$\frac{\partial \ell}{\partial \beta} = \sum_{i=1}^{m} \frac{\partial \ell}{\partial y_i}$$

整体对比如下：




Batch Normalization [5] in training mode.

使用 Batch Normalization 时，因为标准化处理中包含减去均值的一步，因此 b 实际上没有起到作用，其数值效果交由 β 来实现。因此，在 Batch Normalization 中，可以省略 b 或者暂时设置为 0。

在使用梯度下降算法时，分别对 $W^{[l]}$，$\beta^{[l]}$ 和 $\delta^{[l]}$ 进行迭代更新。除了传统的梯度下降算法之外，还可以使用之前学过的动量梯度下降、RMSProp 或者 Adam 等优化算法。

### 1.5.3 BN 有效解释

Batch Normalization 效果很好的原因有以下两点：

1. 通过对隐藏层各神经元的输入做类似的标准化处理，提高神经网络训练速度；
2. 可以使前面层的权重变化对后面层造成的影响减小，整体网络更加健壮。

关于第二点，如果实际应用样本和训练样本的数据分布不同（例如，橘猫图片和黑猫图片），我们称发生了“Covariate Shift”。这种情况下，一般要对模型进行重新训练。Batch Normalization 的作用就是减小 Covariate Shift 所带来的影响，让模型变得更加健壮，鲁棒性（Robustness）更强。

即使输入的值改变了，由于 Batch Normalization 的作用，使得均值和方差保持不变（由 γ 和 β 决定），限制了在前层的参数更新对数值分布的影响程度，因此后层的学习变得更容易一些。Batch Normalization 减少了各层 W 和 b 之间的耦合性，让各层更加独立，实现自我训练学习的效果。

另外，Batch Normalization 也起到**微弱的正则化（regularization）**效果。因为在每个 mini-batch 而非整个数据集上计算均值和方差，只由这一小部分数据估计得出的均值和方差会有一些噪声，因此最终计算出的 $\tilde{z}^{(i)}$ 也有一定噪声。类似于 dropout，这种噪声会使得神经元不会再特别依赖于任何一个输入特征。

因为 Batch Normalization 只有微弱的正则化效果，因此可以和 dropout 一起使用，以获得更强大的正则化效果。通过应用更大的 mini-batch 大小，可以减少噪声，从而减少这种正则化效果。

最后，不要将 Batch Normalization 作为正则化的手段，而是当作加速学习的方式。正则化只是一种非期望的副作用，Batch Normalization 解决的还是反向传播过程中的梯度问题（梯度消失和爆炸）。

### 1.5.4 测试时的 BN

Batch Normalization 将数据以 mini-batch 的形式逐一处理，但在测试时，可能需要对每一个样本逐一处理，这样无法得到 μ 和 $\sigma^2$。

理论上，我们可以将所有训练集放入最终的神经网络模型中，然后将每个隐藏层计算得到的 $\mu^{[l]}$ 和 $\sigma^{2[l]}$ 直接作为测试过程的 μ 和 σ 来使用。但是，实际应用中一般不使用这种方法，而是使用之前学习过的指数加权平均的方法来预测测试过程单个样本的 μ 和 $\sigma^2$。

对于第 l 层隐藏层，考虑所有 mini-batch 在该隐藏层下的 $\mu^{[l]}$ 和 $\sigma^{2[l]}$，然后用指数加权平均的方式来预测得到当前单个样本的 $\mu^{[l]}$ 和 $\sigma^{2[l]}$。这样就实现了对测试过程单个样本的均值和方差估计。

## 1.6 SoftMax 回归

目前为止，介绍的分类例子都是二分类问题：神经网络输出层只有一个神经元，表示预测输出 $\hat{y}$ 是正类的概率$P(y=1|x)$，$\hat{y}>0.5$ 则判断为正类，反之判断为负类。

对于多分类问题，用 C 表示种类个数，则神经网络输出层，也就是第 L 层的单元数量 $n^{[L]}=C$。每个神经元的输出依次对应属于该类的概率，即 $P(y=c|x), c=0,1,..,C-1$。有一种 Logistic 回归的一般形式，叫做 Softmax 回归，可以处理多分类问题。

对于 Softmax 回归模型的输出层，即第 L 层，有：

$$Z^{[L]}=W^{[L]}a^{[L-1]}+b^{[L]} \tag{23}$$

激活函数使用的是softmax函数：

$$\sigma(z)_j=\frac{exp(z_j)}{\sum_{i=1}^{m}exp(z_i)} \tag{24}$$

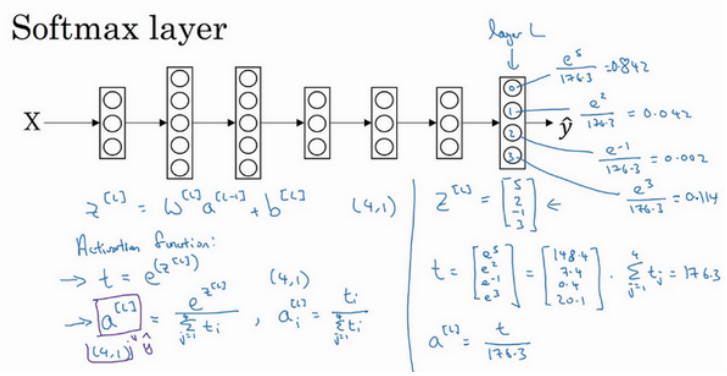
for i in range(L)，有：

$$a_i^{[L]}=\frac{e^{Z_i^{[L]}}}{\sum_{i=1}^{C}e^{Z_i^{[L]}}} \tag{25}$$

为输出层每个神经元的输出，对应属于该类的概率，满足：

$$\sum_{i=1}^{C}a_i^{[L]}=1 \tag{26}$$

下面是一个直观地例子：




**代价函数**

定义损失函数为：

$$L(\hat{y}, y) = -\sum_{j=1}^{C} y_j log\hat{y}_j \tag{27}$$

当 i 为样本真实类别，则有：

$$y_j = 0, j \neq i \tag{28}$$

因此，损失函数可以简化为：

$$L(\hat{y}, y) = -y_i log\hat{y}_i = log\hat{y}_i \tag{29}$$

所有 m 个样本的成本函数为：

$$J = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}, y) \tag{30}$$

# 1.7 深度学习框架

- Caffe / Caffe 2
- CNTK
- DL4J
- Keras
- Lasagne
- mxnet
- PaddlePaddle
- TensorFlow
- Theano
- Torch

**选择框架的标准**

- 便于编程：包括神经网络的开发和迭代、配置产品；
- 运行速度：特别是训练大型数据集时；
- 是否真正开放：不仅需要开源，而且需要良好的管理，能够持续开放所有功能。