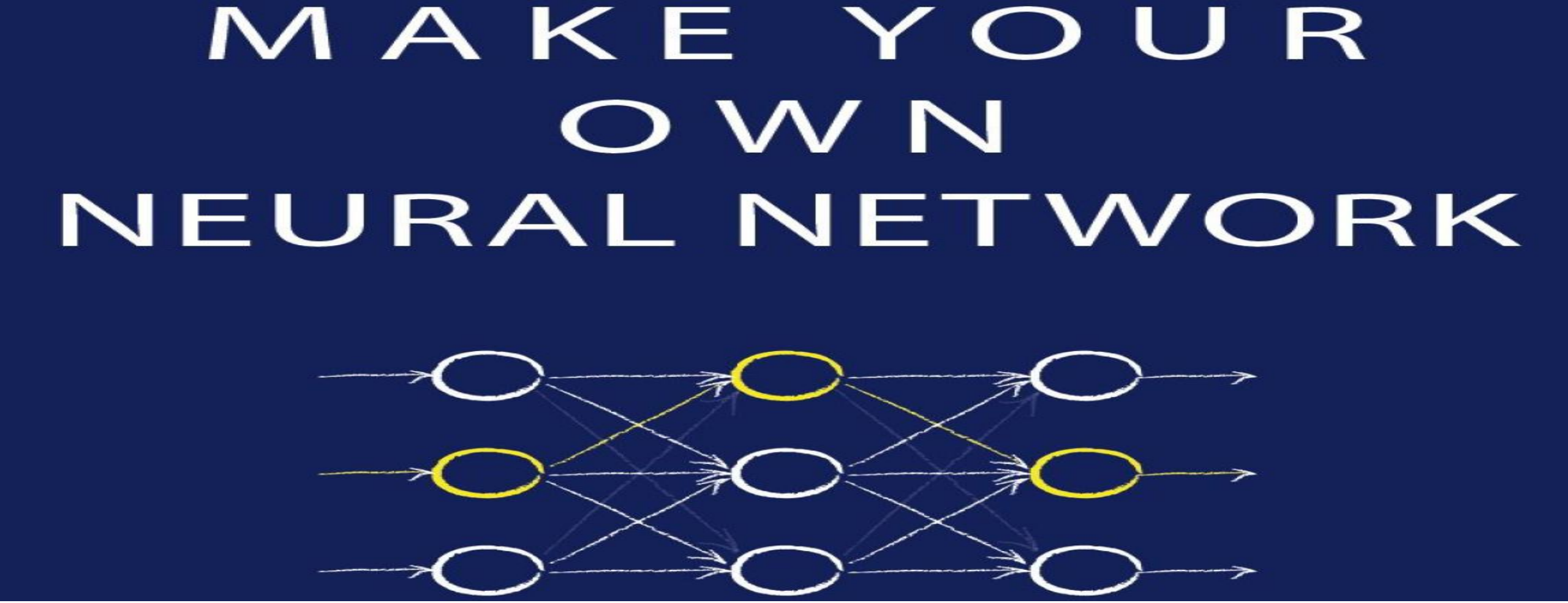


联结主义（神经网络）

1. 常用的激活函数：

1.Sigmoid

Sigmoid 非线性激活函数的数学表达式是 $\sigma(x)=\frac{1}{1+e^{-x}}$，其图形如图 3.14所示。目前我们知道 Sigmoid 激活函数是将一个实数输入转化到 0 ~ 1 之间的输出，具体来说也就是将越大的负数转化到越靠近 0，越大的正数转化到越靠近 1。历史上 Sigmoid 函数频繁地使用，因为其具有良好的解释性。

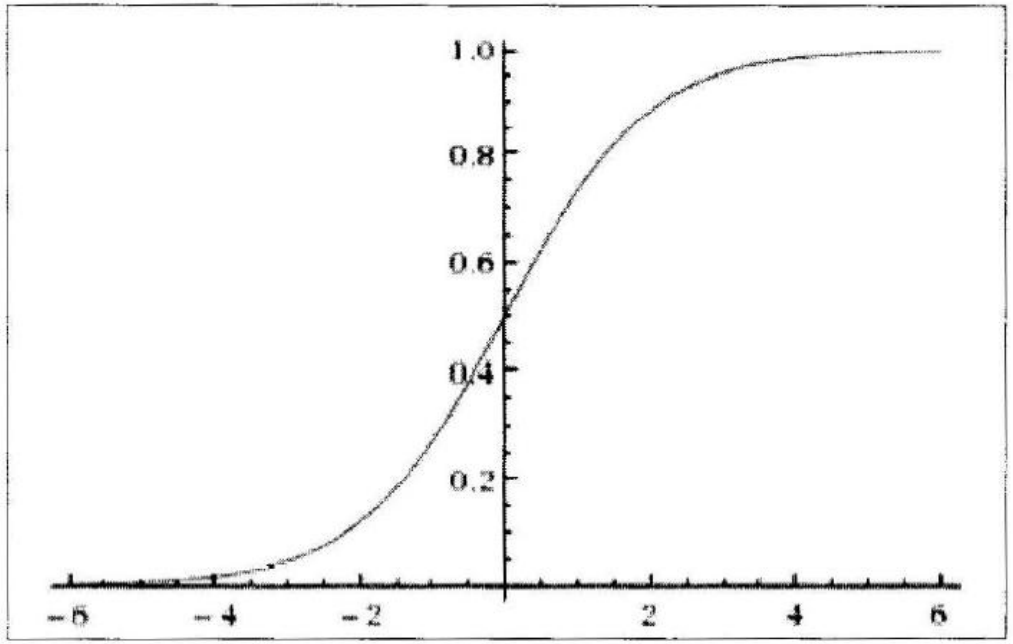
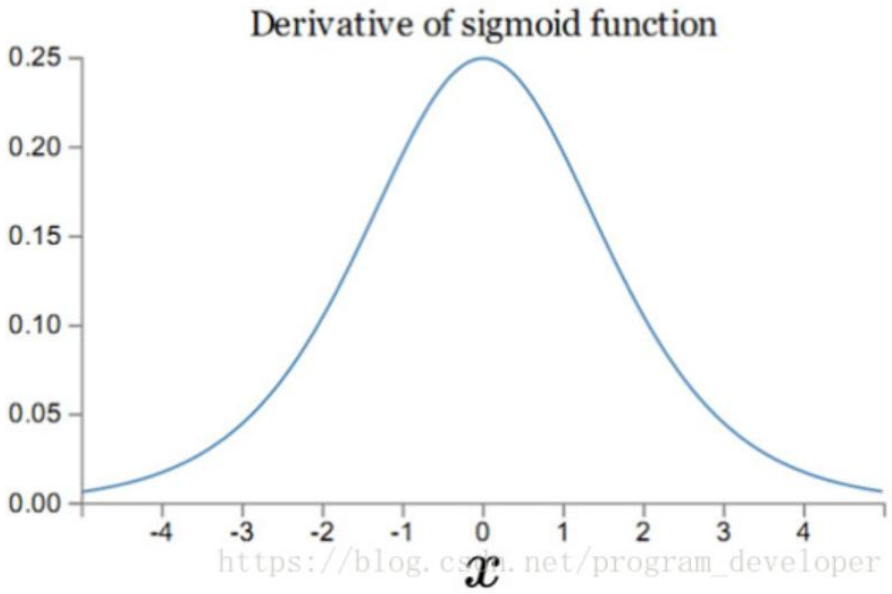


图 3.14　Sigmoid 函数图形

导函数图像如图所示：



Sigmoid缺点

但是最近几年，Sigmoid激活函数已经越来越少地被人使用了，主要是因为Sigmoid 函数有以下两大缺点：

（1） Sigmoid函数会造成梯度消失。一个非常不好的特点就是Sigmoid函数在靠近 1和0的两端时，梯度会几乎变成0，我们前面讲过梯度下降法通过梯度乘上学习率来更新参数，因此如果梯度接近0，那么没有任何信息来更新参数，这样就会造成模型不收敛。另外，如果使用Sigmoid函数，那么需要在初始化权重的时候也必须非常小心。 如果初始化的时候权重太大，那么经过激活函数也会导致大多数神经元变得饱和，没有办法更新参数。

Sigmoid缺点

但是最近几年，Sigmoid激活函数已经越来越少地被人使用了，主要是因为Sigmoid 函数有以下两大缺点：

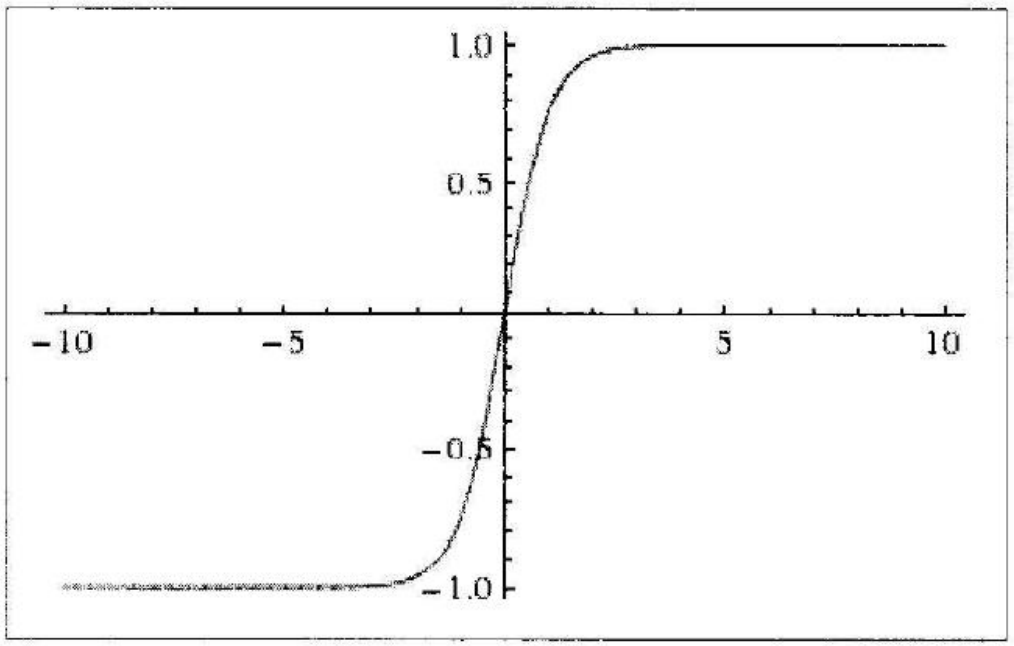
（2）Sigmoid 输出不是以 0 为均值，这就会导致经过 Sigmoid 激活函数之后的输出，作为后面一层网络的输入的时候是非 0 均值的，这个时候如果输入进入下一层神经元的时候全是正的，这就会导致梯度全是正的，那么在更新参数的时候永远都是正梯度。怎么理解呢？比如进入下一层神经元的输入是 x，参数是 w 和 b，那么输出就是 $f = wx + b$，这个时候 $\nabla f(w) = x$，所以如果 x 是 0 均值的数据，那么梯度就会有正有负。但是这个问题并不是太严重，因为一般神经网络在训练的时候都是按 batch（批）进行训练的，这个时候可以在一定程度上缓解这个问题，所以说虽然 0 均值这个问题会产生一些不好的影响，但是总体来讲跟上一个缺点：梯度消失相比还是要好很多。

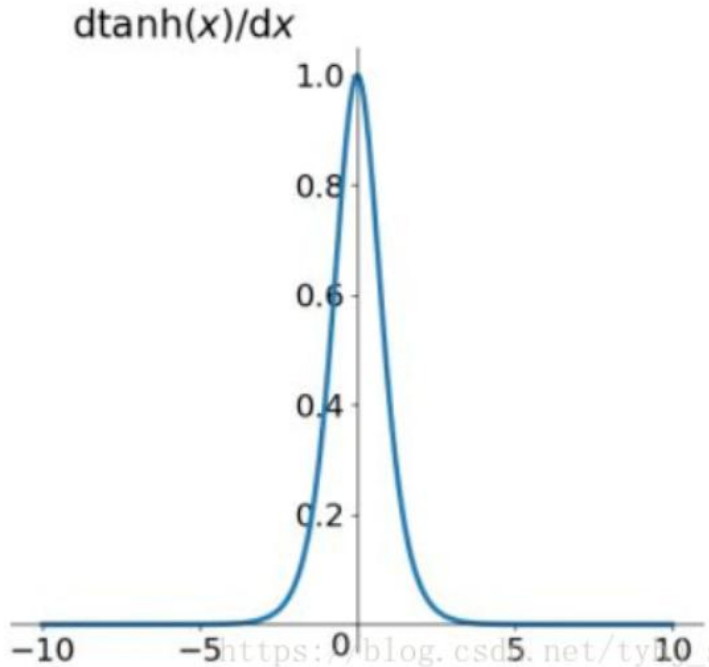
1. 常用的激活函数：

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

2.Tanh

Tanh 激活函数是上面 Sigmoid 激活函数的变形，其数学表达为 $tanh(x) = 2\sigma(2x) -$ 1，图形如图3.15所示。





它将输入的数据转化到 $-1 \sim 1$ 之间，可以通过图像看出它将输出变成了 0 均值，在一定程度上解决了 Sigmoid 函数的第二个问题，但是它仍然存在梯度消失的问题。因此实际上 Tanh 激活函数总是比 Sigmoid 激活函数更好。

1. 常用的激活函数：

3.ReLU

ReLU 激活函数（Rectified Linear Unit）近几年变得越来越流行，它的数学表达式为 $f(x) = max(0, x)$，换句话说，这个激活函数只是简单地将大于 0 的部分保留，将小于 0 的部分变成 0，它的图形如图 3.16所示。

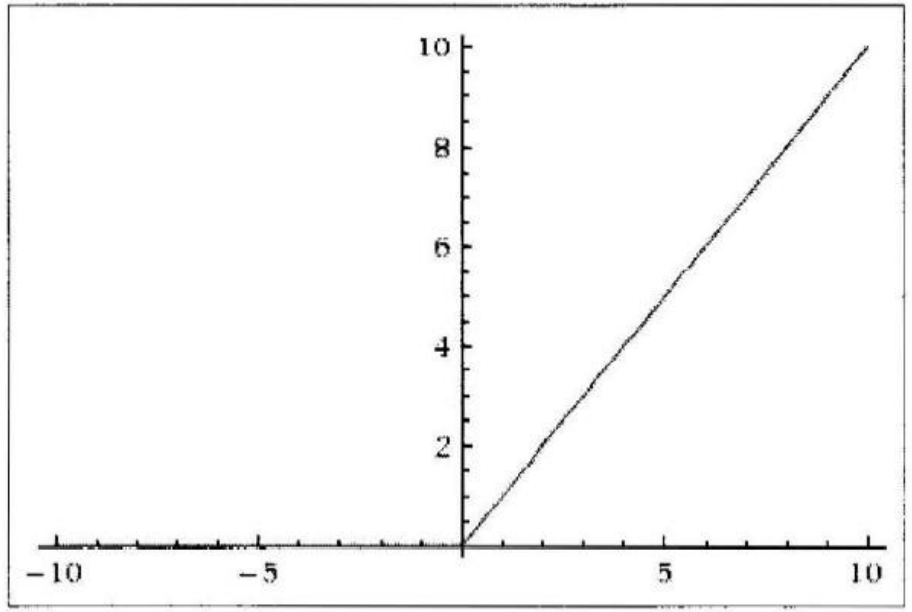
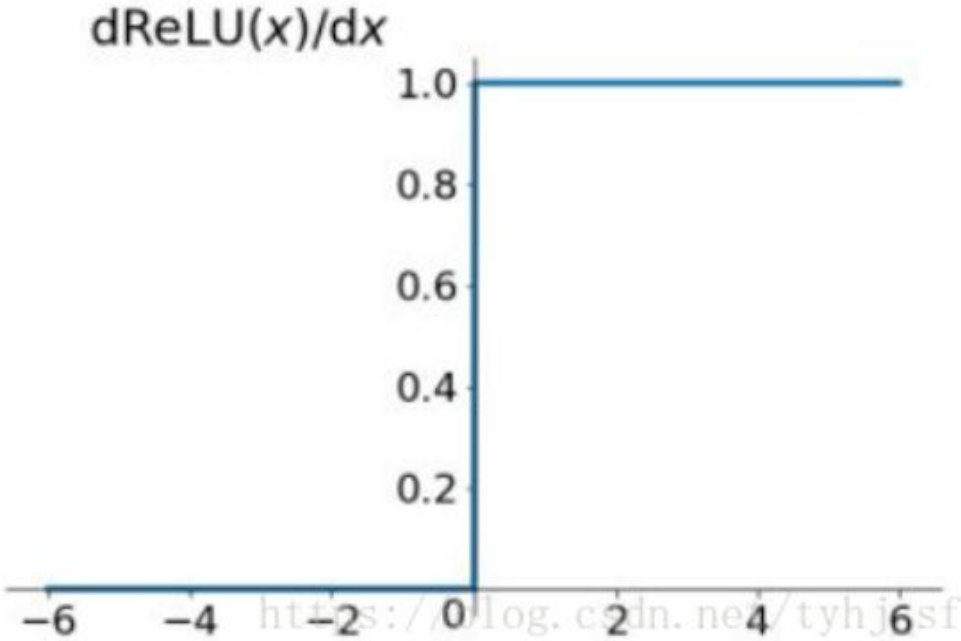


图 3.16　ReLU 函数图形



ReLU优缺点：

ReLU 的优点：

（1）相比于 Sigmoid 激活函数和 Tanh 激活函数，ReLU 激活函数能够极大地加速随机梯度下降法的收敛速度，这因为它是线性的，且不存在梯度消失的问题。

（2）相比于 Sigmoid 激活函数和 Tanh 激活函数的复杂计算而言，ReLU 的计算方法更加简单，只需要一个阈值过滤就可以得到结果，不需要进行一大堆复杂的运算。

ReLU优缺点：

ReLU 的缺点：

训练的时候很脆弱，比如一个很大的梯度经过 ReLU 激活函数，更新参数之后，会使得这个神经元不会对任何数据有激活现象。如果发生这种情况之后，经过 ReLU 的梯度永远都会是 0，也就意味着参数无法再更新了，因为 ReLU 激活函数本质上是一个不可逆的过程，因为它会直接去掉输入小于 0 的部分。在实际操作中可以通过设置比较小的学习率来避免这个小问题。

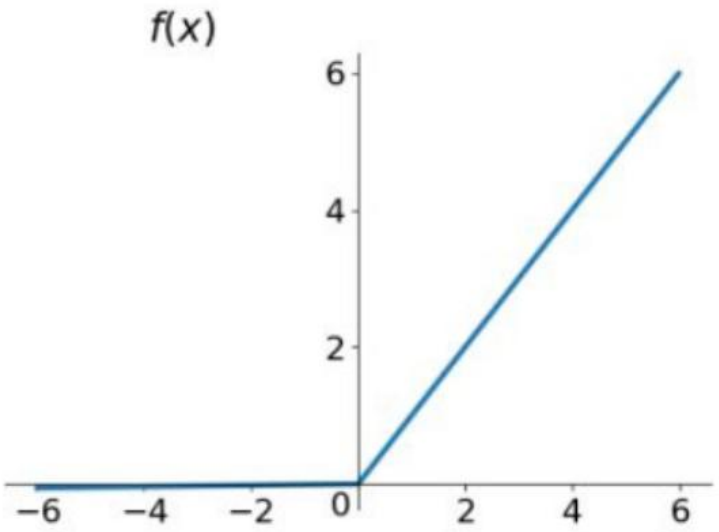
1. 常用的激活函数：

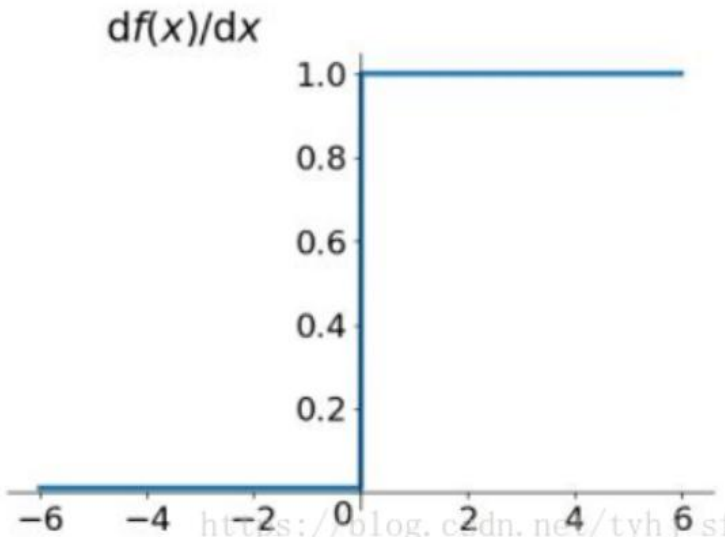
- 4. 其他

Leaky ReLU函数（PReLU）

函数表达式：

$$f(x) = max(\alpha x, x)$$





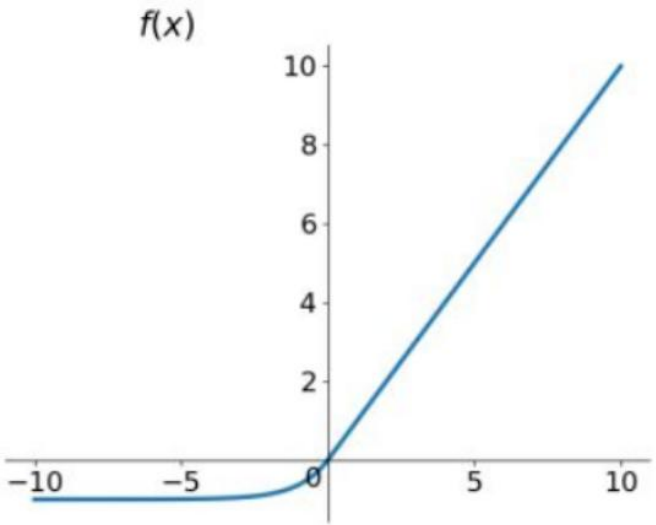
1. 常用的激活函数：

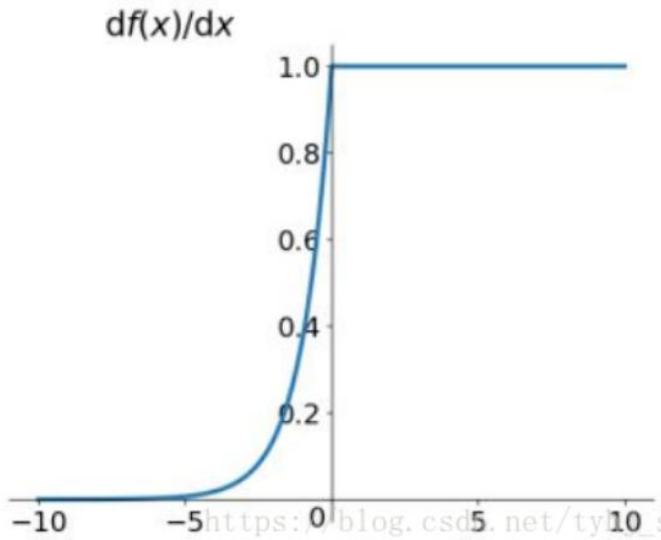
- 4. 其他

ELU (Exponential Linear Units) 函数

函数表达式：

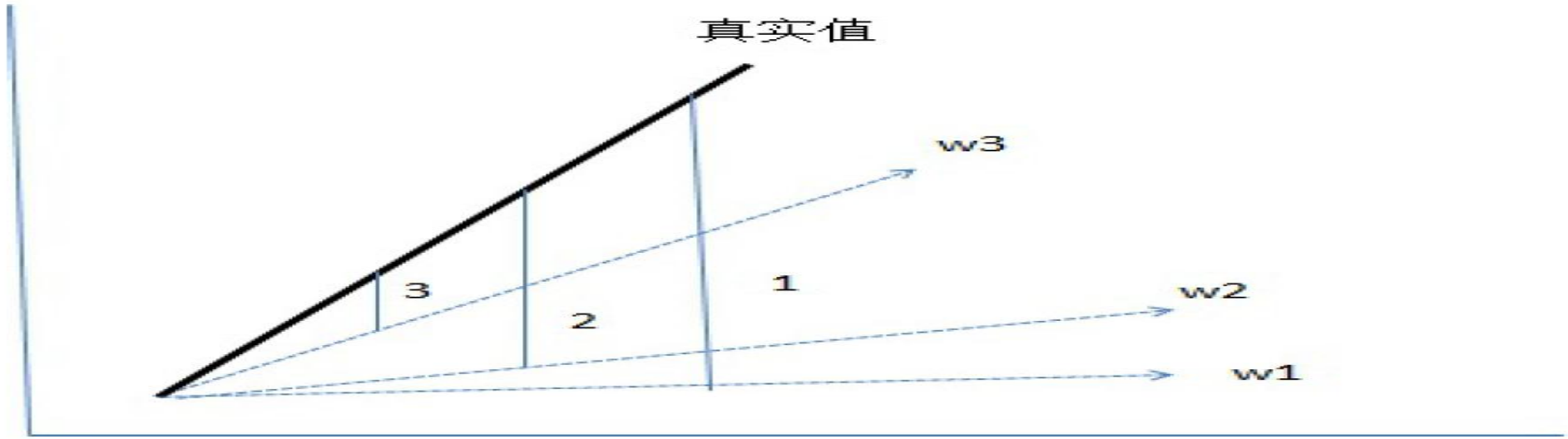
$$f(x) = \begin{cases} x, & \text{if } x > 0 \\ \alpha(e^x - 1), & \text{otherwise} \end{cases}$$





2. 常用的损失函数：

损失函数可以表示为 $L(y, f(x))$ ，用以衡量真实值 y和预测值 f(x)之间不一致的程度，一般越小越好。为了便于不同损失函数的比较，常将其表示为单变量的函数，在回归问题中$[y - f(x)]$ 为 $[yf(x)]$ ，残差；在分类问题中为 趋势一致。



回归问题

（1）MSE 均方误差

均方误差（MSE）是回归损失函数中最常用的误差，也常被称为L2 loss，它是预测值与目标值之间差值的平方和，其公式如下所示：

$$MSE = \frac{\sum_{i=1}^{n}(y_i - y_i^p)^2}{n} \text{ 或者 } L_2(\hat{y}, y) = \sum_{i=0}^{m}\left(y^{(i)} - \hat{y}^{(i)}\right)^2$$

优点：各点都连续光滑，方便求导，具有较为稳定的解。

缺点：不是特别的稳健，因为当函数的输入值距离中心值较远的时候，使用梯度下降法求解的时候梯度很大，可能导致**梯度爆炸**。

回归问题

（2）平均绝对误差

平均绝对误差（MAE）是另一种常用的回归损失函数，也常被称为L1 loss，它是目标值与预测值之差绝对值的和，表示了预测值的平均误差幅度，而不需要考虑误差的方向，范围是0到∞，其公式如下所示：

$$MAE = \frac{\sum_{i=1}^{n}|y_i - y_i^p|}{n} \text{ 或者 } L_1(\hat{y}, y) = \sum_{i=0}^{m}\left|y^{(i)} - \hat{y}^{(i)}\right|$$

优点：无论对于什么样的输入值，都有着稳定的梯度，不会导致梯度爆炸问题，具有较为稳健性的解。

缺点：在中心点是折点，不能求导，不方便求解。

回归问题

（3）smooth L1 loss

smooth L1说的是光滑之后的L1，前面说过了L1损失的缺点就是有折点，不光滑，导致不稳定，那如何让其变得光滑呢？smooth L1损失函数为：

$$\text{smooth}_{L_1}(x) = \begin{cases} 0.5x^2 \times 1/\sigma^2 \text{ if } |x| < 1/\sigma^2 \\ |x| - 0.5 \text{ otherwise} \end{cases}$$

smooth L1 loss让loss对于离群点更加鲁棒，相比于L2损失函数，其对离群点、异常值（outlier）不敏感，梯度变化相对更小，训练时不容易跑飞。

分类问题

（1）0-1损失函数

以二分类问题为例，错误率=1-正确率，也就是0-1损失函数，可以定义为：

$$L(Y, f(X)) = \begin{cases} 1, Y \neq f(X) \\ 0, Y = f(X) \end{cases}$$

该损失函数不考虑预测值和真实值的误差程度，也就是说只要预测错误，预测错误差一点和差很多是一样的。感知机就是用的这种损失函数，但是由于相等这个条件太过严格，我们可以放宽条件，即满足 |Y-f(X)|<T时认为相等。

这种损失函数用在实际场景中比较少，更多的是用来衡量其他损失函数的效果。

分类问题

（2）绝对值损失函数

$$L\left(Y, f(X)\right) = |Y - f(X)|$$

分类问题

（3）交叉熵损失函数

交叉熵损失是非常重要的损失函数，也是应用最多的损失函数之一，交叉熵损失更清晰的描述了模型与理想模型的距离。二分类问题的交叉熵 Loss 主要有两种形式（标签y的定义不同）：

基于输出标签 label 的表示方式为 {0,1}，也最为常见。它的 Loss 表达式为：

$$L = -[y \log \hat{y} + (1-y) \log(1-\hat{y})] \text{ 或者 } -\sum_i \hat{y} \log(y_i)$$

基于输出标签 label 的表示方式为 {-1,+1}（Logistics Loss），也比较常见。它的 Loss 表达式：

$$L = \log(1 + e^{-ys})$$

ys 的符号反映了预测的准确性。

分类问题

（4）Softmax Loss

对于多分类问题，也可以使用Softmax Loss。

机器学习模型的 Softmax 层，正确类别对于的输出是：

$$S = \frac{e^s}{\sum_{j=1}^{C} e^{s_j}}$$

（5）Logistic loss

$$L(y, f(x)) = \log\left(1 + e^{-yf(x)}\right)$$

3. 深度模型中的优化

寻找神经网络上的一组参数θ，它能显著地降低代价函数J（θ），
该代价函数通常包括整个训练集上的性能评估和额外的正则化项。

通常，代价函数可写为训练集上的平均，如

$$J(\boldsymbol{\theta}) = \mathbb{E}_{(\mathbf{x},\mathrm{y})\sim\hat{p}_{\mathrm{data}}} L(f(\boldsymbol{x};\boldsymbol{\theta}), y)$$

其中 L 是每个样本的损失函数，$f(\boldsymbol{x};\boldsymbol{\theta})$ 是输入 $\boldsymbol{x}$ 时所预测的输出，$\hat{p}_{\mathrm{data}}$ 是经验分布。监督学习中，y 是目标输出。

经验风险最小化（empirical risk minimization）

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

其中 m 表示训练样本的数目。

一般的优化和用于训练算法的优化有一个重要不同：
训练算法通常不会停止在局部极小点，通常是在基于提前终止的收敛条件满足时停止。与纯优化不同的是，提前终止时代理损失函数仍然有较大的导数，而纯优化终止时导数较小。

批量算法

机器学习算法和一般优化算法不同的一点是，机器学习算法的目标函数通常可以分解为训练样本上的求和。机器学习中的优化算法在计算参数的每一次更新时通常仅使用整个代价函数中一部分项来估计代价函数的期望值。

例如，最大似然估计问题可以在对数空间中分解成各个样本的总和：

$$\boldsymbol{\theta}_{\mathrm{ML}} = \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{m} \log p_{\mathrm{model}}(\boldsymbol{x}^{(i)}, y^{(i)}; \boldsymbol{\theta}).$$

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：

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

优化算法用到的目标函数J 中的大多数属性也是训练集上的期望。例如，最常用的属性是梯度：

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

随机梯度下降（stochastic gradient descent, SGD）

- 准确计算这个期望的计算代价非常大，因为需要在整个数据集上的每个样本上评估模型。
- 从小数目样本中获得梯度的统计估计的动机是训练集的冗余。
- 在实践中，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平均值。

✓ 小批量是随机抽取的这点很重要。从一组样本中计算出梯度期望的无偏估计要求这些样本是独立的。也希望两个连续的梯度估计是互相独立的，因此两个连续的小批量样本也应该是彼此独立的

✓ 很多小批量随机梯度下降方法的实现都会打乱数据顺序一次，然后多次遍历数据来更新参数。

随机梯度下降的核心是，梯度是期望。期望可使用小规模的样本近似估计。具体而言，在算法的每一步，我们从训练集中均匀抽出一**小批量**（minibatch）样本 $\mathbb{B} = \{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m')}\}$。小批量的数目 m' 通常是一个相对较小的数，从一到几百。重要的是，当训练集大小 m 增长时，m' 通常是固定的。我们可能在拟合几十亿的样本时，每次更新计算只用到几百个样本。

梯度的估计可以表示成

$$\boldsymbol{g} = \frac{1}{m'} \nabla_{\boldsymbol{\theta}} \sum_{i=1}^{m'} L(\boldsymbol{x}^{(i)}, y^{(i)}, \boldsymbol{\theta}).$$

使用来自小批量 $\mathbb{B}$ 的样本。然后，随机梯度下降算法使用如下的梯度下降估计：

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon \boldsymbol{g},$$

其中，ϵ 是学习率。

算法 8.1 随机梯度下降（SGD）在第 k 个训练迭代的更新

Require: 学习率 ϵ_k

Require: 初始参数 $\boldsymbol{\theta}$

while 停止准则未满足 **do**

从训练集中采包含 m 个样本 $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ 的小批量，其中 $\boldsymbol{x}^{(i)}$ 对应目标为 $\boldsymbol{y}^{(i)}$。

计算梯度估计：$\hat{\boldsymbol{g}} \leftarrow +\frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$

应用更新：$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon \hat{\boldsymbol{g}}$

end while

● 学习率可通过试验和误差来选取，通常最好的选择方法是监测目标函数值随时间变化的学习曲线。与其说是科学，这更像是一门艺术。

➢ 若太大，学习曲线将会剧烈振荡，代价函数值通常会明显增加。

➢ 如果学习率太小，那么学习过程会很缓慢。如果初始学习率太低，那么学习可能会卡在一个相当高的代价值。

梯度下降法的变式

1.SGD

随机梯度下降法是梯度下降法的一个小变形，就是每次使用一批（batch)数据进行梯度的计算，而不是计算全部数据的梯度，因为现在深度学习的数据量都特别大，所以每次都计算所有数据的梯度是不现实的，这样会导致运算时间特别长，同时每次都计算全部的梯度还失去了一些随机性，容易陷入局部误差，所以使用随机梯度下降法可能每次都不是朝着真正最小的方向，但是这样反而容易跳出局部极小点。

加速神经网络训练---Momentum

算法 8.2 使用动量的随机梯度下降（SGD）

Require: 学习率 ϵ，动量参数 α

Require: 初始参数 $\boldsymbol{\theta}$，初始速度 $\boldsymbol{v}$

while 没有达到停止准则 **do**

从训练集中采包含 m 个样本 $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ 的小批量，对应目标为 $\boldsymbol{y}^{(i)}$。

计算梯度估计：$\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$

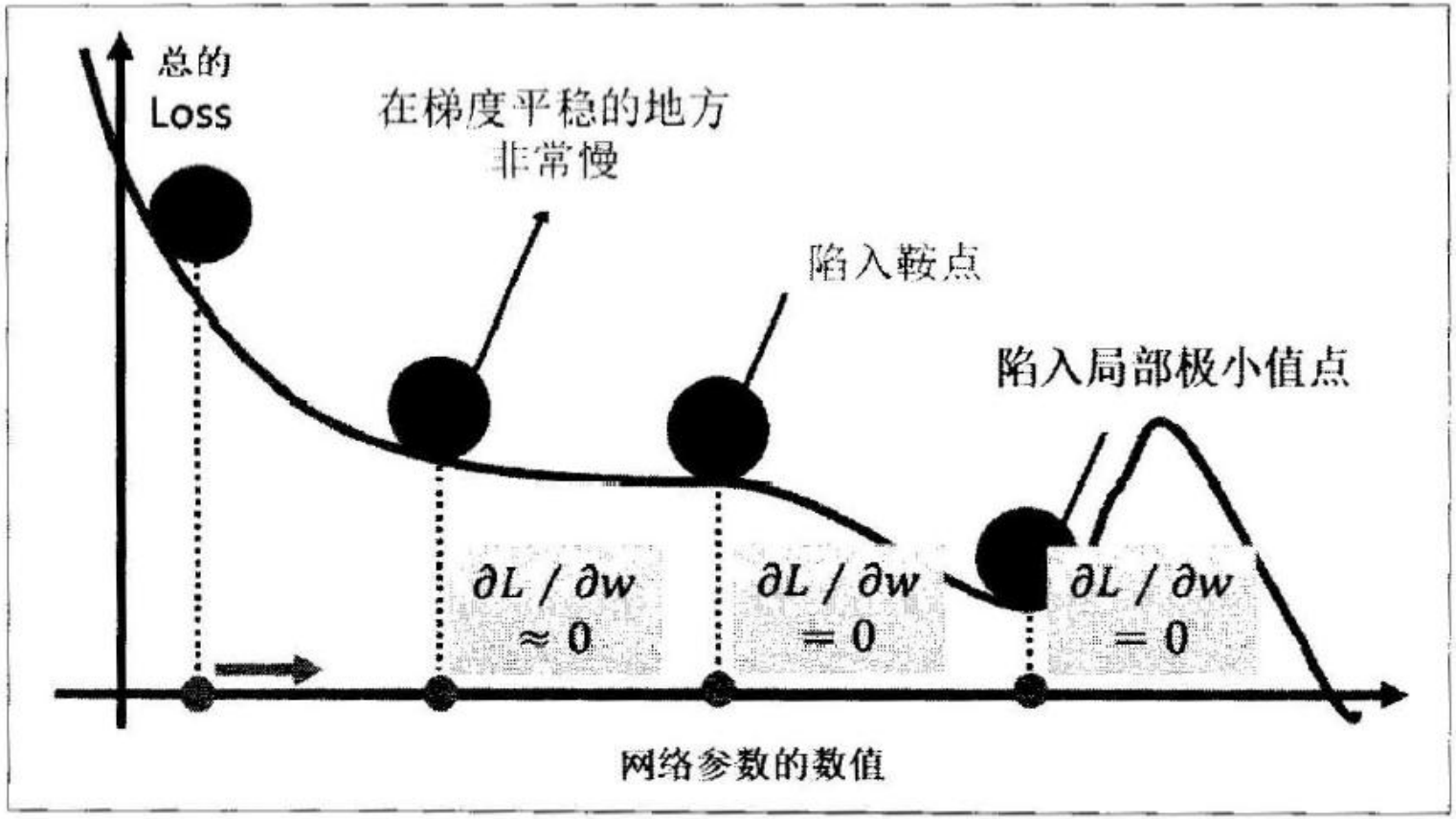
计算速度更新：$\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \boldsymbol{g}$

应用更新：$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$

end while

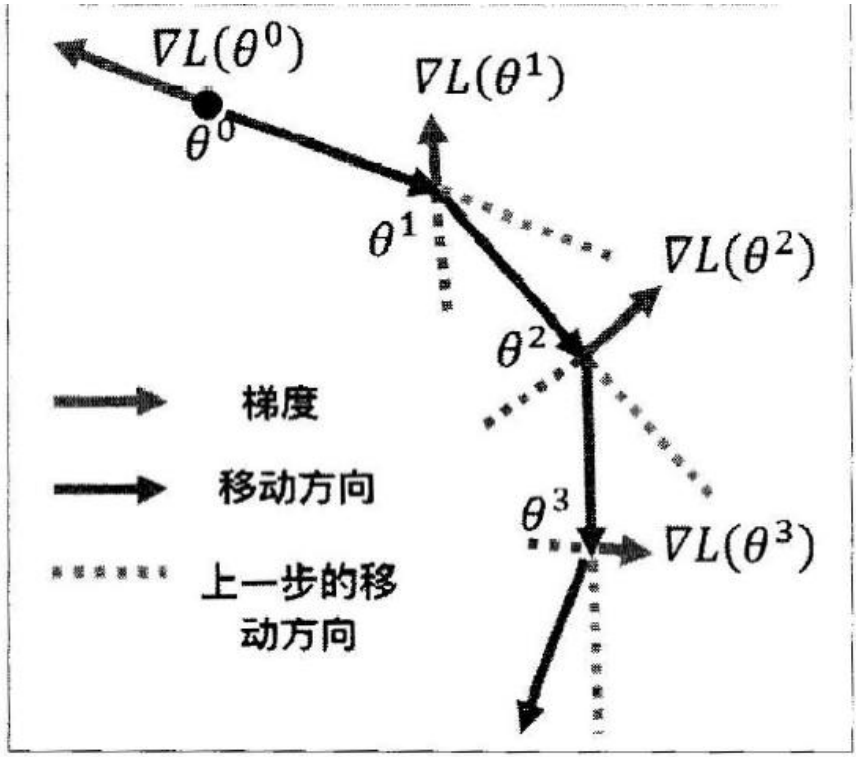
2.Momentum

第二种优化方法就是在随机梯度下降的同时，增加动量(Momentum)。这来自于物理中的概念，可以想象损失函数是一个山谷，一个球从山谷滑下来，在一个平坦的地势，球的滑动速度就会慢下来，可能陷人一些鞍点或者局部极小值点。



这个时候给它增加动量就可以让它从高处滑落时的势能转换为平地的动能，相当于惯性增加了小球在平地滑动的速度，从而帮助其跳出鞍点或者局部极小点。

动量怎么计算呢？动量的计算基于前面梯度，也就是说参数更新不仅仅基于当前的梯度，也基于之前的梯度。



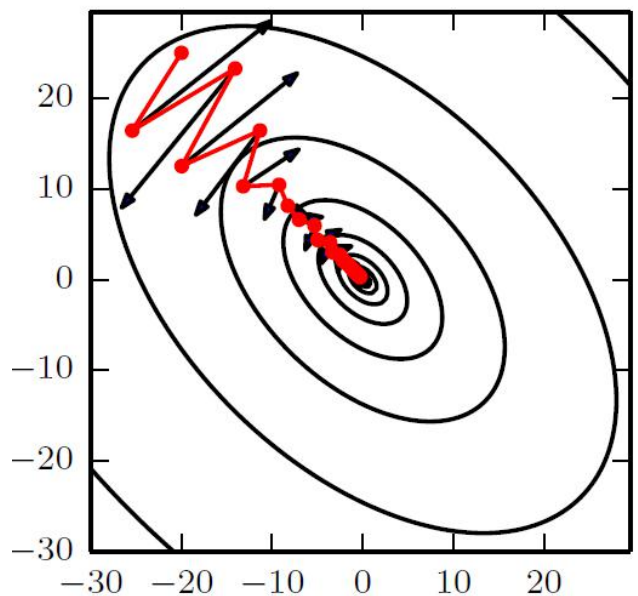
更新规则如下：

$$\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \nabla_{\boldsymbol{\theta}} \left(\frac{1}{m} \sum_{i=1}^{m} L(\boldsymbol{f}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)}) \right),$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}.$$

速度 $\boldsymbol{v}$ 累积了梯度元素 $\nabla_{\boldsymbol{\theta}}(\frac{1}{m}\sum_{i=1}^{m} L(\boldsymbol{f}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)}))$。

相对于 ϵ，α 越大，之前梯度 对现在方向的影响也越大。



加速神经网络训练---AdaGrad

算法 8.4 AdaGrad 算法

Require: 全局学习率 ϵ

Require: 初始参数 $\boldsymbol{\theta}$

Require: 小常数 δ，为了数值稳定大约设为 10^{-7}

初始化梯度累积变量 $\boldsymbol{r} = 0$

while 没有达到停止准则 **do**

从训练集中采包含 m 个样本 $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ 的小批量，对应目标为 $\boldsymbol{y}^{(i)}$。

计算梯度：$\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$

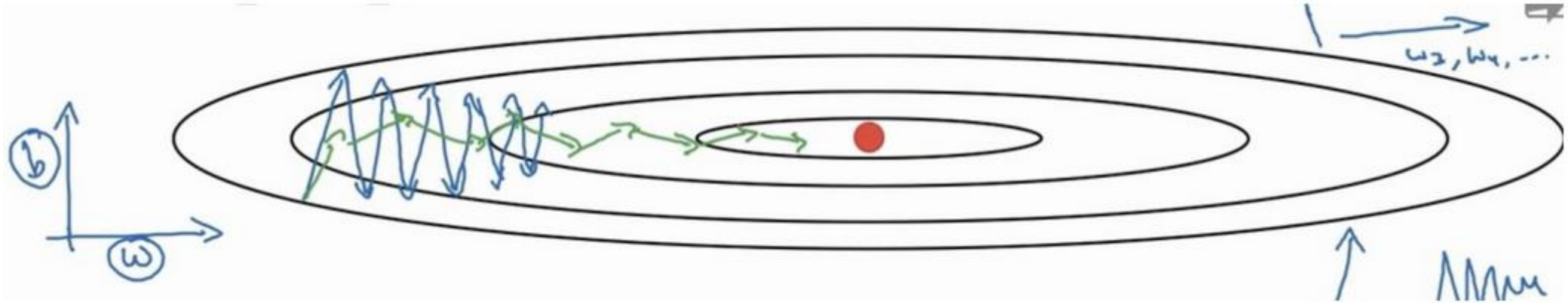
累积平方梯度：$\boldsymbol{r} \leftarrow \boldsymbol{r} + \boldsymbol{g} \odot \boldsymbol{g}$

计算更新：$\Delta\boldsymbol{\theta} \leftarrow -\frac{\epsilon}{\delta + \sqrt{r}} \odot \boldsymbol{g}$ （逐元素地应用除和求平方根）

应用更新：$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$

end while

AdaGrad在算法中使用了累积平方梯度r=:r + g⊙g。那么在下次计算更新的时候，
r是作为分母出现的，越大的反而更新越小，越小的值反而更新越大，
那么更新则会像下面绿色线更新一样，明显就会好于SGD蓝色更新曲线。



在参数空间更为平缓的方向，会取得更大的进步（因为平缓，所以历史梯度平方和较小，对应学习下降的幅度较小），并且能够使得陡峭的方向变得平缓，
从而加快训练速度。

加速神经网络训练---RMSProp

算法 8.5 RMSProp 算法

Require: 全局学习率 ϵ，衰减速率 ρ

Require: 初始参数 $\boldsymbol{\theta}$

Require: 小常数 δ，通常设为 10^{-6}（用于被小数除时的数值稳定）

初始化累积变量 $\boldsymbol{r} = 0$

while 没有达到停止准则 **do**

从训练集中采包含 m 个样本 $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ 的小批量，对应目标为 $\boldsymbol{y}^{(i)}$。

计算梯度：$\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$

累积平方梯度：$\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1 - \rho) \boldsymbol{g} \odot \boldsymbol{g}$

计算参数更新：$\Delta \boldsymbol{\theta} = -\frac{\epsilon}{\sqrt{\delta + r}} \odot \boldsymbol{g}$ （$\frac{1}{\sqrt{\delta + r}}$ 逐元素应用）

应用更新：$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta \boldsymbol{\theta}$

end while

RMSprop是一种Adagrad的改进方法。
这里主要多了一个ρ，这是一个衰减率，也就是说RMSprop不再会将前面所有的梯度平方求和，而是通过一个衰减率将其变小，使用了一种滑动平均的方式，越靠前面的梯度对自适应的学习率影响越小，这样就能更加有效地避免Adagrad学习率一直递减太多的问题，能够更快地收敛。

计算梯度：$\boldsymbol{g} \leftarrow \frac{1}{m}\nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}), \boldsymbol{y}^{(i)})$

累积平方梯度：$\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1-\rho)\boldsymbol{g} \odot \boldsymbol{g}$

计算参数更新：$\Delta\boldsymbol{\theta} = -\frac{\epsilon}{\sqrt{\delta + r}} \odot \boldsymbol{g}$ （$\frac{1}{\sqrt{\delta+r}}$ 逐元素应用）

应用更新：$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$

加速神经网络训练---Adam

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 $\boldsymbol{\theta}$

初始化一阶和二阶矩变量 $\boldsymbol{s}=0, \boldsymbol{r}=0$

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 $\{\boldsymbol{x}^{(1)},\ldots,\boldsymbol{x}^{(m)}\}$ 的小批量，对应目标为 $\boldsymbol{y}^{(i)}$。

计算梯度：$\boldsymbol{g} \leftarrow \frac{1}{m}\nabla_{\boldsymbol{\theta}}\sum_i L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}),\boldsymbol{y}^{(i)})$

$t \leftarrow t+1$

更新有偏一阶矩估计：$\boldsymbol{s} \leftarrow \rho_1\boldsymbol{s}+(1-\rho_1)\boldsymbol{g}$

更新有偏二阶矩估计：$\boldsymbol{r} \leftarrow \rho_2\boldsymbol{r}+(1-\rho_2)\boldsymbol{g}\odot\boldsymbol{g}$

修正一阶矩的偏差：$\hat{\boldsymbol{s}} \leftarrow \frac{\boldsymbol{s}}{1-\rho_1^t}$

修正二阶矩的偏差：$\hat{\boldsymbol{r}} \leftarrow \frac{\boldsymbol{r}}{1-\rho_2^t}$

计算更新：$\Delta\boldsymbol{\theta} = -\epsilon\frac{\hat{\boldsymbol{s}}}{\sqrt{\hat{\boldsymbol{r}}}+\delta}$（逐元素应用操作）

应用更新：$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta}+\Delta\boldsymbol{\theta}$

end while

RMSprop+Momentum

优化器

pytorch 的优化器都放在 torch.optim 包中。常见的优化器有：

SGD，Adam，Adadelta，Adagrad，Adamax 等。这几种优化器足够现实

```
import torch.optim as optim

dir(optim)
```

```
['ASGD',
 'Adadelta',
 'Adagrad',
 'Adam',
 'Adamax',
 'LBFGS',
 'Optimizer',
 'RMSprop',
 'Rprop',
 'SGD',
 'SparseAdam',
 '__builtins__',
 '__cached__',
 '__doc__',
 '__file__',
 '__loader__',
 '__name__',
 '__package__',
 '__path__',
 '__spec__',
 'lr_scheduler']
```

选择正确的优化算法

该选择哪种算法呢？

- 遗憾的是，目前在这一点上没有达成共识。Schaul et al. (2014) 展示了许多优化算法在大量学习任务上极具价值的比较。虽然结果表明，具有自适应学习率（以RMSProp 和AdaDelta 为代表）的算法族表现得相当鲁棒，不分伯仲，但没有哪个算法能脱颖而出。
- 目前，最流行并且使用很高的优化算法包括SGD、具动量的SGD、RMSProp、具动量的RMSProp、AdaDelta 和Adam。此时，选择哪一个算法似乎主要取决于使用者对算法的熟悉程度（以便调节超参数）。

4. 数据处理和训练技巧

α：数据预处理

1. 中心化

数据预处理中一个最常见的处理办法就是每个特征维度都减去相应的均值实现中心化，这样可以使得数据变成0均值，特别对于一些图像数据，为了方便我们将所有的数据都减去一个相同的值。

4. 数据处理和训练技巧

α： 数据预处理

2. 标准化

在使得数据都变成0均值之后，还需要使用标准化的做法让数据不同的特征维度都有着相同的规模。有两种常用的方法：一种是除以标准差，这样可以使得新数据的分布接近标准高斯分布；还有一种做法就是让每个特征维度的最大值和最小值按比例缩放到-1～1之间

如果知道输入不同特征有着不同的规模，那就需要使用标准化的做法让它们处于同一个规模下面，这对于机器学习算法而言是非常重要的。

4. 数据处理和训练技巧

α：数据预处理

3.PCA

PCA 是另外一种处理数据的方法，在进行这一步以前，首先会将数据中心化，然后计算数据的协方差矩阵，这一步特别简单，假设输入是 $X = N \times D$，那么通过 $\frac{X^T X}{N}$ 能够得到这个协方差矩阵，可以验证一下这个结果的正确性，而且这个协方差矩阵是对称半正定的，可以通过这个协方差矩阵来进行奇异值分解（SVD），然后对数据进行去相关性，将其投影到一个特征空间，我们能够取一些较大的、主要的特征向量来降低数据的维数，去掉一些没有方差的维度，这也叫做主成分分析（PCA）。

这个操作对于一些线性模型和神经网络，都能取得良好的效果。

4. 数据处理和训练技巧

α：数据预处理

4. 白噪声

白噪声也是一种处理数据的方式，首先会跟 PCA 一样将数据投影到一个特征空间，然后每个维度除以特征值来标准化这些数据，直观上就是一个多元高斯分布转化到了一个 0 均值，协方差矩阵为 1 的多元高斯分布。

4. 数据处理和训练技巧

α：数据预处理

在实际处理数据中，中心化和标准化都特别重要。我们计算训练集的统计量比如均值，然后将这些统计量应用到测试集和验证集当中。但是 PCA 和白噪声在卷积网络中基本不使用，因为卷积网络可以自动学习如何提取这些特征而不需要人工再去对其进行干预。

4. 数据处理和训练技巧

β：权重初始化

1. 全0初始化

首先从一个最直观，但是不应该采用的策略入手，那就是将参数全部初始化为0

解释一下为什么不能采用这一种策略：首先，我们并不知道训练之后的网络最后权重更新的是多少，但是知道数据在进入网络之前经过了合适的预处理，所以我们可以假设最后的权重有一半是正的，一半是负的，所以将参数全部初始化为0似乎是一个非常好的选择。但这是不对的，因为如果神经网络中每个权重都被初始化成相同的值，那么每个神经元就会计算出相同的结果，在反向传播的时候也会计算出相同的梯度，最后导致所有权重都会有相同的更新。换句话说，如果每个权重都被初始化成相同的值，那么权重之间失去了不对称性。

4. 数据处理和训练技巧

β：权重初始化

2. 随机初始化

目前知道我们希望权重初始化的时候能够尽量靠近0，但是不能全都等于0，所以可以初始化权重为一些靠近0的随机数，通过这个方式可以打破对称性。这里面的核心想法就是神经元最开始都是随机的、唯一的，所以在更新的时候也是作为独立的部分， 最后一起合成在神经网络当中。

一般的随机化策略有高斯随机化、均匀随机化等，需要注意的是并不是越小的随机化产生的结果越好，因为权重初始化越小，反向传播中关于权重的梯度也越小，因为梯度与参数的大小是成比例的，所以这会极大地减弱梯度流的信号，成为神经网络训练中的一个隐患。

4. 数据处理和训练技巧

β：权重初始化

3. 稀疏初始化

另外一种初始化的方法就是稀疏初始化，将权重全部初始化为 0，然后为了打破对称性在里面随机挑选一些参数附上一些随机值。这种方法的好处是参数占用的内存较少，因为里面有较多的 0，但是实际中使用较少。

4. 初始化偏置（bias）

对于偏置 (bias)，通常是初始化为 0，因为权重已经打破了对称性，所以使用 0 来初始化是最简单的。

4. 数据处理和训练技巧

β：权重初始化

5. 批标准化（Batch Normalization ）

最近兴起的一项技术叫做批标准化，它的核心想法就是标准化这个过程是可微的，减少了很多不合理初始化的问题，所以可以将标准化过程应用到神经网络的每一层中做前向传播和反向传播，通常批标准化应用在全连接层后面、非线性层前面。

实际中批标准化已经变成了神经网络中的一个标准技术，特别是在卷积神经网络中，它对于很坏的初始化有很强的鲁棒性，同时还可以加快网络的收敛速度。另外，批标准化还可以理解为在网络的每一层前面都会做数据的预处理。

Batch normalization:

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

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

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

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

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

Ioffe S, Szegedy C. Batch normalization: Accelerating deep network training by reducing internal covaria
International conference on machine learning. PMLR, 2015: 448-456.

4. 数据处理和训练技巧

γ：防止过拟合

1. 正则化

L2 正则化是正则化（regularization）中比较常用的形式，它的想法是对于权重过大的部分进行惩罚，也就是直接在损失函数中增加权重的二范数量级，也就是 $\frac{1}{2}\lambda w^2$，其中 λ 是正则化强度，通常使用 0.5，因为对于 w^2 的梯度是 $2w$，使用 $\frac{1}{2}$ 就能使得梯度是 λw 而不是 $2\lambda w$。所以使用 L2 正则化可以看成是权重更新在原来的基础上再 $-\lambda w$，这样可以让参数更新之后更加靠近 0。

4. 数据处理和训练技巧

γ：防止过拟合

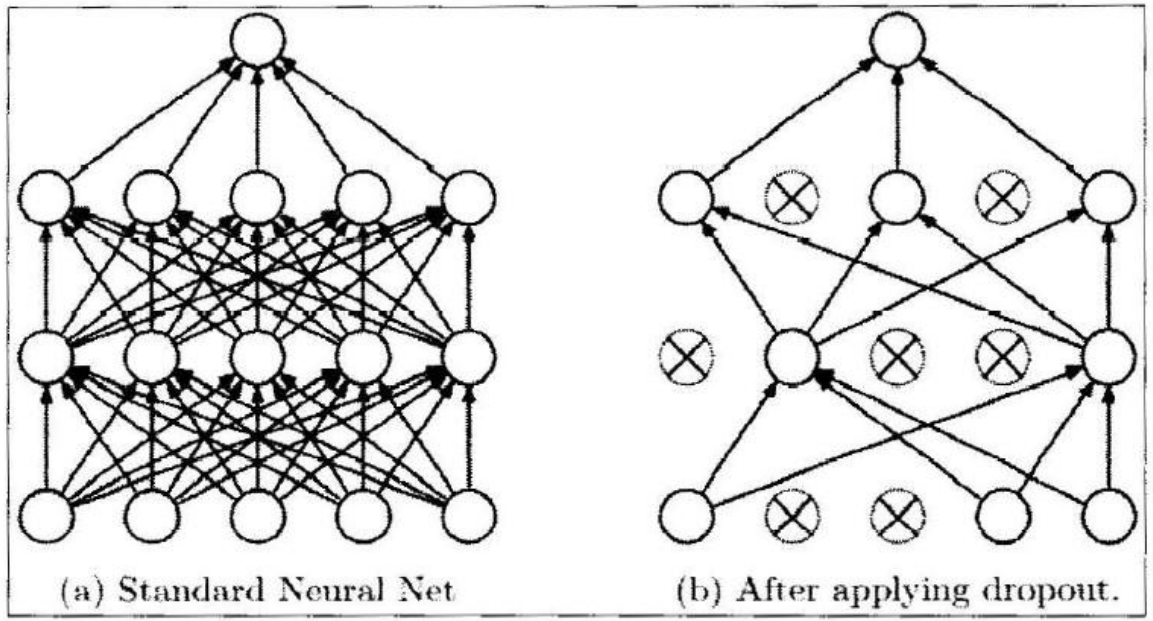
L1 正则化是另外一种正则化方法，其在损失函数中增加权重的 1 范数，也就是 $\lambda|w|$，我们也可以把 L1 正则化和 L2 正则化结合起来，如 $\lambda_1|w| + \lambda_2 w^2$。L1 正则化相对于 L2 正则化的优势是在优化的过程中可以让权重变得更加稀疏，换句话说，也就是在优化结束的时候，权重只会取一些与最重要的输入有关的权重，这就使得与噪声相关的权重被尽可能降为 0。L2 正则化的优势在于最终的效果会比 L1 正则化更加发散，权重也会被限制得更小。

4. 数据处理和训练技巧

γ：防止过拟合

2. Dropout

现在介绍一种非常有效、简单、同时也是现在深度学习使用最为广泛的防止过拟合的方法——Dropout。其核心想法就是在训练网络的时候依概率P保留每个神经元， 也就是说每次训练的时候有些神经元会被设置为0。




Thanks!
Obrigado!
Gracias!
Dziękuję!
谢谢！
Danke!
Grazie!
شكرا لك !
Merci!
감사합니다!
Kiitos!