Tips for Deep Learning

本文的主要思路：针对 training set 和 testing set 的 performance 分别提出针对性的解决方法

1. 在 training set 上准确率不高
   New activation function: ReLU、Maxout
   Adaptive learning rate：Adagrad、RMSProp、Momentum、Adam
2. 在 testing set 上准确率不高: Early Stopping、Regularization or Dropout

Recipe of Deep Learning

Three step of deep learning 的三个步骤

- Define the function set(network structure)
- Goodness of function（loss function---cross entropy）
- Pick the best function（gradient descent----optimization）

Good Results on Training Data?

The first thing you should do: **提高 model 在 training set 上的正确率**

先检查 training set 的 performance 其实是 deep learning 一个非常 unique 的地方，如果今天你用的是 k-nearest neighbor 或 decision tree 这类非 deep learning 的方法，做完以后你其实会不太想检查 Training set 的结果，因为在 training set 上的 performance 正确率就是 100，没有什么好检查的；

有人说 deep learning 的 model 里这么多参数，感觉一脸很容易 overfitting 的样子，但实际上这个 deep learning 的方法，它不容易 overfitting，我们说的 overfitting 就是在 training set 上 performance 很好，但在 testing set 上 performance 没有那么好；只有像 k-nearest neighbor 或 decision tree 这类方法，它们 Training set 上正确率都是 100，这才容易 overfitting 的，而对 deep learning 来说，overfitting 往往不会是你遇到的第一个问题

因为你在 training 的时候，deep learning 并不是想 k-nearest neighbor 这种方法一样，一训练就可以得到非常好的正确率，他有可能在 training set 上根本没有办法给你一个好的正确率，所以，这个时候你要回头检查在前面的 step 里面要做什么样的修改，好让你在 training set 上可以得到比价高的正确率

Good Results on Testing Data?

The second thing：**提高 model 在 testing set 上的正确率**

假设现在你已经在 training set 上得到好的 performance 了，那接下来就把 model apply 到 testing set 上，我们最后真正关心的是 testing set 上的 performance，假如得到的结果不好，合格情况下发生的才会使 Overfitting，也就是在 training set 上得到好的结果，却在 testing set 上得到不好的结果

那你要回过头去做一些事情，试着解决 overfitting，但有时候你加了新的 technique，想要 overcome overfitting 这个 problem 的时候，其实反而会让 training set 上的结果变坏，所以你在做完这一步的修改以后，要先回头去检查新的 model 在 training set 上的结果，如果这个结果变坏的话，你就要从头对 network training 的 process 做一些调整，那如果你同时在 training set 还有 testing set 上得到好结果的话，你就成功了，最后就可以把你的系统真正用在 application 上面了

Conclusion

当你在 deep learning 的文献上看到某种方法的时候，永远要想一下，这个方法是解决什么样的问题，因为在 deep learning 里面，有两个问题:

- 在 training set 上的 performance 不够好
- 在 testing set 上的 performance 不够好

当只有一个方法 propose（提出）的时候，它往往只针对这两个问题的其中一个来做处理，举例来说，deep learning 有一个很潮的方法叫做 dropout，那很多人就会说，所以今天只要看到 performance 不好，我就去用 dropout；但是，其实只有在 testing 的结果不好的时候，才可以去 apply dropout，如果你今天的问题只是 training 的结果不好，那你去 apply dropout，只会越 train 越差而已

所以，**你必须先想清楚现在的问题到底是什么，然后再根据这个问题去找针对性的方法，**而不是病急乱投医，甚至是盲目诊断

下面我们分别从 training data 和 testing data 这两个问题出发，来讲述一些针对性优化的方法

Goodness Results on Training Data?

这一部分主要讲述如何在 Training data 上得到更好的 performance，分为两个模块，New activation function 和

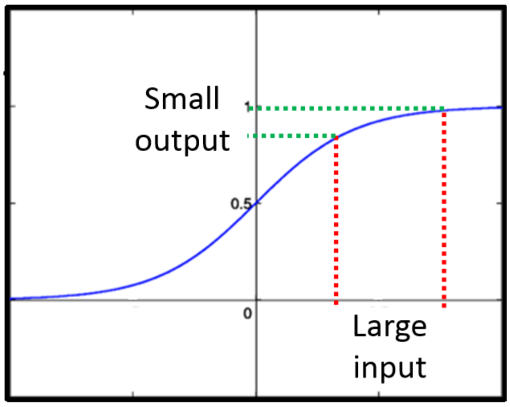
Adaptive Learning Rate

New activation function

Vanishing Gradient Problem

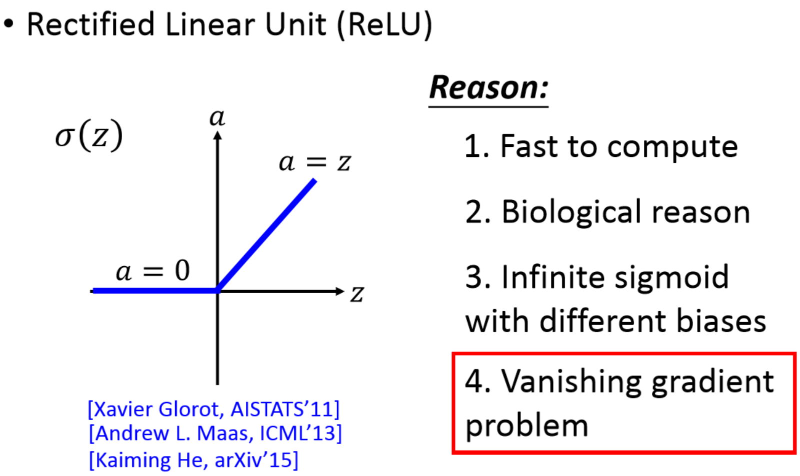
某一个参数w对 total cost l 的偏微分，即 gradient $\frac{\partial l}{\partial w}$,它直觉的意思是说，当我今天把这个参数做小小的变化的时候，它对这个 cost 的影响有多大；那我们就把第一个 layer 里的某一个参数w计算Δw，看看对 network 的 output 和 target 之间有什么样的影响

Δw通过 sigmoid function 之后，得到 output 是会变小的，改变某一个参数的 weight，会对某个 neuron 的 output 值产生影响，但是这个影响是会随着层数的递增而衰减的，sigmoid function 的形状如下如所示，它会把负无穷大到正无穷大之间的值都硬压到 0~1 之间，把较大的 input 压缩成较小的 output



因此即使Δw值很大，但每经过一个 sigmoid function 就会被缩小一次，所以 network 越深，Δw被衰减的次数就越多，直到最后，它对 output 的影响就是很微小，相应的也导致 input 对 loss 的影响会比较小，于是靠近 input 的那些 weight 对 loss 的 gradient $\frac{\partial l}{\partial w}$远小于靠近 output 的 gradient

ReLU(Rectified Linear Unit)

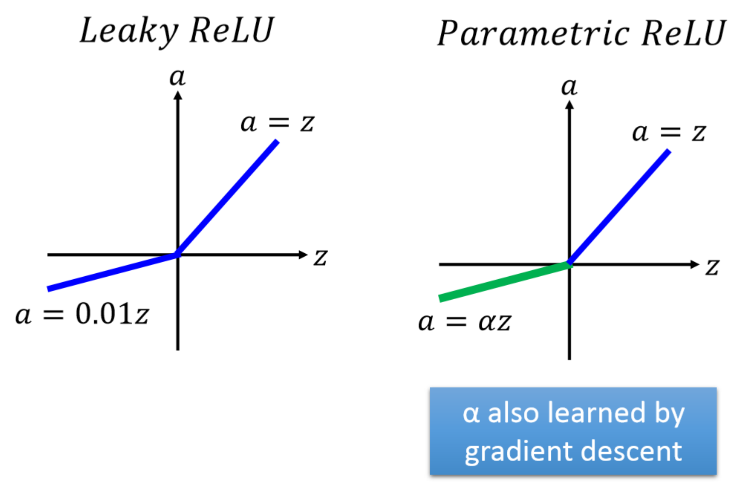


选择 ReLU 的理由如下:

- 跟 sigmoid function 相比，计算快
- 无穷多 bias 不同的 sigmoid function 叠加的结果会变成 ReLU
- ReLU 可以处理 Vanishing gradient 的问题

ReLU-Variant

其实 ReLU 还存在一定的问题，比如当 input<0 的时候，output=0，此时微分值 gradient 也为 0，你就没有办法去 update 参数了，所以我们应该让 input<0 的时候，微分后还能有一点点的值，比如令 a=0.01z，这个东西就叫做 **Leaky ReLU**



既然 a 可以等于 0.01z，那这个的系数可不可以是 0.07/0.08 之类呢?所以就有人提出了 Parametric ReLU，也就是令$a = \alpha \cdot z$，其中α并不是固定的值，而是 network 的一个参数，它可以通过 training data 学出来，甚至每个 neuron 都可以有不同的α值

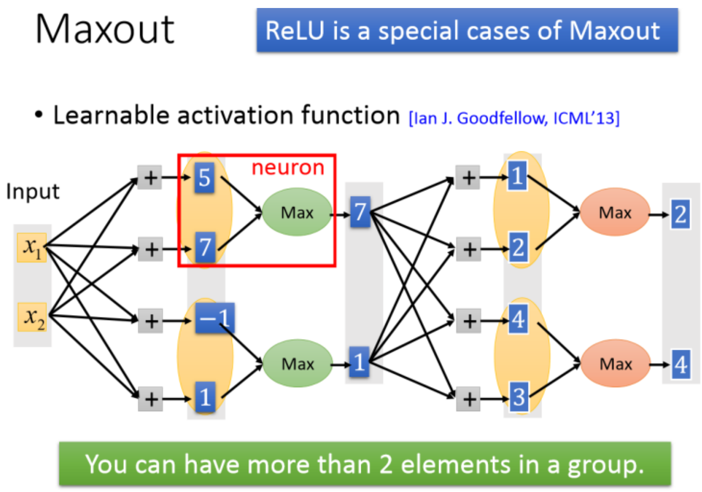
Maxout

Maxout 的想法是，让 network 自动去学习它的 activation function，那 Maxout network 就可以自动学出 ReLU，也可以学出其他的 activation function，这一切都是由 training data 来决定的

假设现在有 input x_1，x_2,它们乘上几组不同的 weight 分别得到 5,7-1,1,这些值本来是不同 neuron 的 input，它们要通过 activation function 变为 neuron 的 output；但在 Maxout network 里，我们事先决定好讲某几个‘neuron’的 input 分为一个 group，比如 5,7 分为一个 group，然后在这个 group 里选取一个最大值 7 作为 output

这个过程就好像在一个 layer 上做 Max pooling 一样，它和原来的 network 不同之处在于，它把原来几个‘neuron’的 input 按一定规则组成了一个 group，然后并没有使它们通过 activation function，而是选取其中的最大值当做这几个‘neuron’的 output

当然，实际上原来的‘neuron’早就已经不存在了，这几个被合并的‘neuron’应当被看做是一个新的 neuron，这个新的 neuron 的 input 是原来几个‘neuron’的 input 组成的 vector，output 则取 input 的最大值，而并非有 activation function 产生

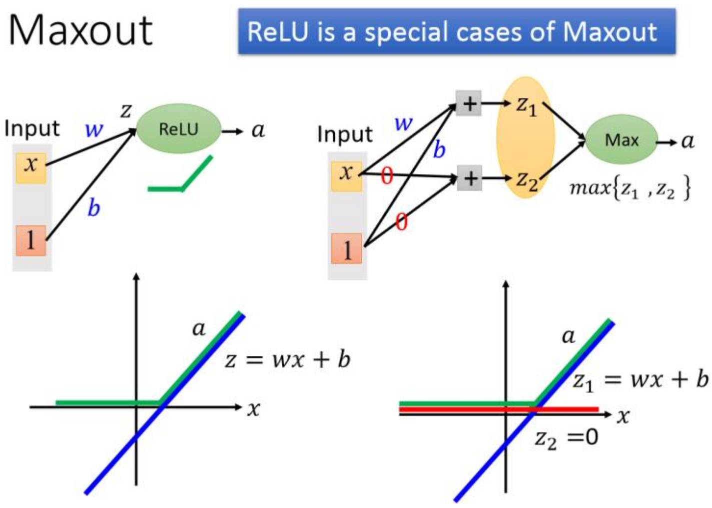


在实际操作上，几个 element 被分为一个 group 这件事情是由你自己决定的，它就是 network structure 里一个需要被调的参数，不一定要跟上图一样两个分为一组

Maxout—>>ReLU

下图左上角是一个 ReLU 的 neuron，它的 input x会乘上 neuron 的 weight w，再加上 bias b，然后通过 activation function,得到 output a

- Neuron 的 input 为$z = wx + b$，为下图左下角紫线
- Neuron 的 output 为$a = z(z > 0); a = 0(z < 0)$,为下图左下角绿线



如果我们使用的是上图右上角所示的 Maxout network，假设z_1的参数w和b与 ReLU 的参数一致，而 z 的参数w和b设为 0，然后做 Max pooling，选取z_1, z_2较大值作为a

- neuron 的 input 为$[z_1, z_2]$
  - $z_1 = wx + b$，为上图右下角紫线
  - $z_2 = 0$，为上图右下角红线
- neuron 的 output 为 max$[z_1, z_2]$，为上图右下角绿线

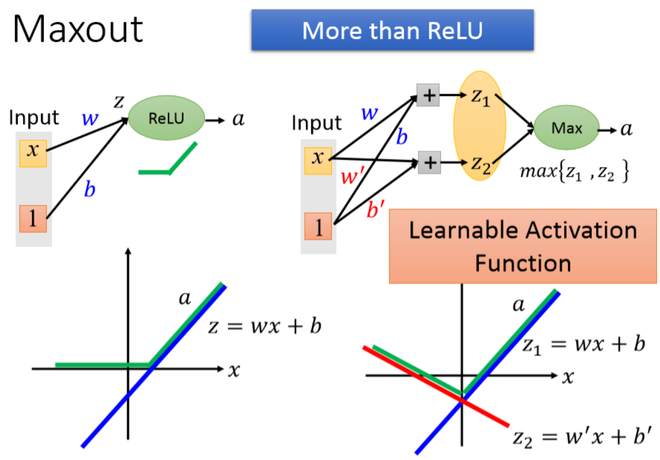
你会发现，此时 ReLU 和 Maxout 所得到的 output 是一模一样的，它们是相同的 activation function

Maxout->more than ReLU

除了 ReLU，Maxout 还可以实现更多不同的 activation function

比如z_2的参数w和b不是 0，而是w', b',此时

- neuron 的 input 为$[z_1, z_2]$
  - $z_1 = wx + b$，为上图右下角紫线
  - $z_2 = w'x + b'$，为上图右下角红线
- neuron 的 output 为 max$[z_1, z_2]$，为上图右下角绿线

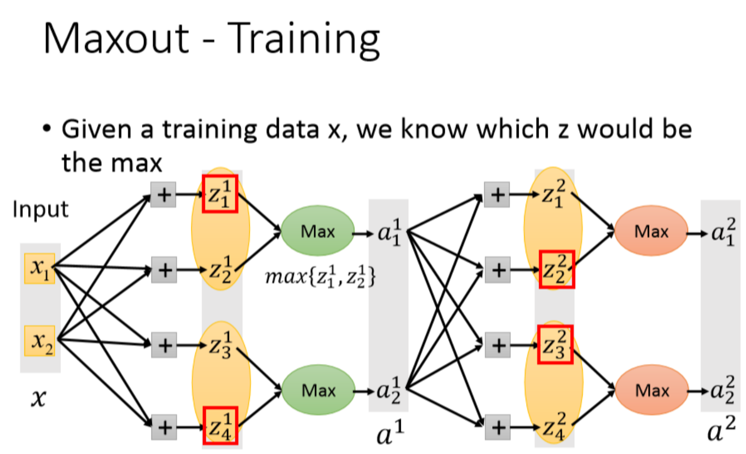


这个时候你得到的 activation function 的形状(绿线形状)，是由 network 的参数w, b, w', b',决定的，因此它是一个 Learnable Activation Function，具体的形状可以根据 training data 去 generate 出来

How to train Maxout

接下来我们要面对的是，怎么去 train 一个 Maxout network，如何解决 Max 不能微分的问题

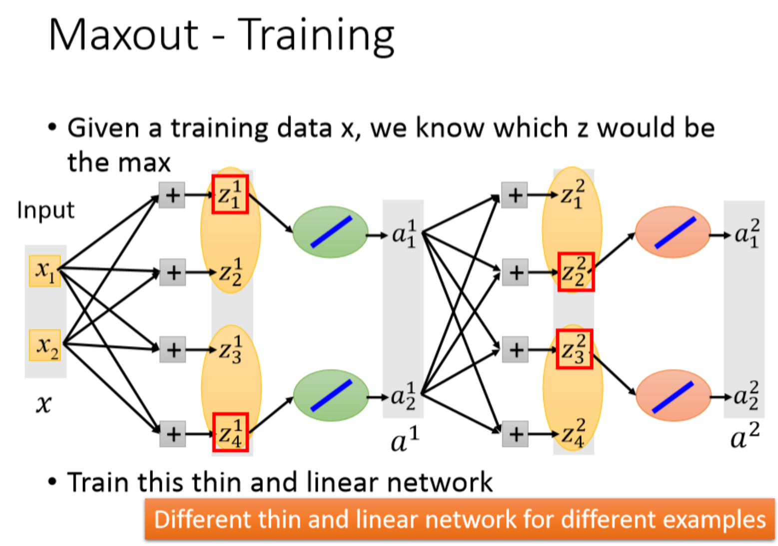
假设在下面的 Maxout network 中，红框圈起来的部分为每个 neuron 的 output



其实 Max operation 就是 linear 的 operation，只是它紧接在前面这个 group 里的某一个 element 上，因此我们可以把那些并没有被 Max 连接到的 element 通通拿掉，从而得到一个比较细长的 linear network

实际上我们真正训练的并不是一个含有 max 函数的 network，而是一个化简后如下图所示的 linear network；当我们还没有真正开始训练模型的时候，此时这个 network 含有 max 函数无法微分，但是只要真正的丢进去了一笔 data，network 就会马上根据这笔 data 确定具体的形状，此时 max 函数的问题已经被实际数据解决了，所以我们完全可以根据这笔 training data 使用 Backpropagation 的方法去训练被 network 留下来的参数

所以我们担心的 max 函数无法微分，**它只是理论上的问题，在具体的实践上，我们完全可以先根据 data 把 max 函数转化为某个具体的函数，再对这个转化后的 thiner linear network 进行微分**



这个时候你也许会有一个问题，如果按照上面的做法，那岂不是只会 train 留在 network 里面的那些参数，剩下的参数该怎么办?那些被拿掉的 weight 岂不是永远也 train 不到了吗?

其实这也只是个理论上的问题，在实际操作上，我们之前已经提到过，每个 linear network 的 structure 都是由 input 的那一笔 data 来决定的，当你 input 不同 data 的时候，得到的 network structure 是不同的，留在 network 里面的参数也是不同的，**由于我们有很多很多笔 training data，所以 network 的 structure 在训练中不断地变换，实际上最后每一个 weight 参数都会被 train 到**

RMSProp

我们的 learning rate 依旧设置为一个固定的值η除掉一个变化的σ，这个σ等于上一个σ和当前梯度g的加权方均根(特别的是，在第一个时间点，σ^0就是第一个算出来的 gradient 值g^0)

$$w^{t+1}=w^t-\frac{\eta}{\sigma^t}g^t$$

$$\sigma^t=\sqrt{\alpha(\sigma^{t-1})^2+(1-\alpha)(g^t)^2}$$

这里的α值是可以自由调整的，RMSProp 跟 Adagrad 不同之处在于，Adagrad 的分母是对过程中所有的 gradient 取平方和开根号，也就是说 Adagrad 考虑的是整个过程平均的 gradient 信息；而 RMSProp 虽然也是对所有的 gradient 进行平方和开根号，但是**它用一个α来调整对不同 gradient 的使用程度**，比如你把α的值设的小一点，意思就是你更倾向于相信新的 gradient 所告诉你的 error surface 的平滑或陡峭程度，而比较无视于旧的 gradient 所提供给你的 information

所以当你做 RMSProp 的时候，一样是在算 gradient 的 root mean square，但是你可以给现在已经看到的 gradient 比较大的 weight，给过去看到的 gradient 比较小的 weight，来调整对 gradient 信息的使用程度

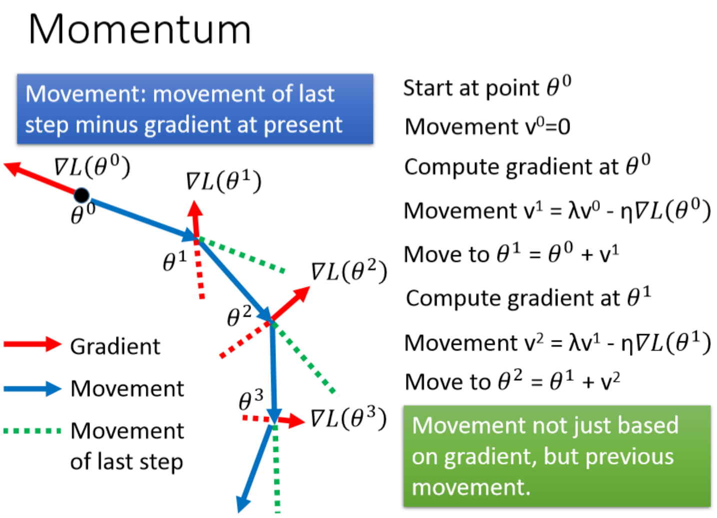
Momentum

当我们在 gradient descent 里加上 Momentum 的时候，每一次 update 的方向，不在只考虑 gradient 的方向，还要考虑上一次 update 的方向，那这里我们就用一个变量v去记录前一个时间点 update 的方向

随机选一个初始值θ^0,初始化$v^0 = 0$,接下来计算θ^0处的 gradient，然后我们要移动的方向是由前一个时间点的移动方向v^0和 gradient 的反方向$\nabla L(\theta^0)$来决定的，即

$v^1 = \lambda v^0 - \eta \nabla L(\theta^0)$

接下来我们第二个时间点要走的方向v^2，它是由第一时间点移动的方向v^1和 gradient 的反方向$\nabla L(\theta^1)$共同决定的；λv是图中的绿色虚线，它代表由于上一次的惯性想要继续走的方向；$\eta \nabla L(\theta)$是图中的红色虚线，它代表这次 gradient 告诉你所要移动的方向，它们的矢量和就是这一次真实移动的方向，为蓝色实线



Adam

其实 RMSProp 加上 Momentum，就可以得到 Adam

根据下面的 paper 来快速描述一下 Adam 的 algorithm

- 先初始化$m_0 = 0$, m_0就是 Momentum 中，前一个时间点 movement 再初始化$v_0 = 0$, v_0就是 RMSProp 里计算 gradient 的 root mean square 的σ最后初始化$t = 0$, t用来表示时间点
- 先算出 gradient $g_t = \nabla_\theta f_t(\theta_{t-1})$
- 再根据过去要走的方向m_{t-1}和 gradient g_t,算出现在要走的方向m_t
  $m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$
- 然后根据前一个时间点的v_{t-1}和 gradient g_t的平方，算一下放在分母的v_t
  $v_t = \beta_2 v_{t-1} + (1-\beta_2)(g_t)^2$
- 接下来做了一个原来 RMSProp 和 Momentum 里没有的的东西，就是 bias correction，它使m_t和v_t都除上一个值，这个值本来比较小，后来会越来越接近 1
  $\hat{m}_t = \frac{m_t}{1-\beta_1^t}$　　　　$\hat{v}_t = \frac{v_t}{1-\beta_2^t}$
- 最后做 update，把 Momentum 建议你的方向$\hat{m}_t$乘上 learning rate α，再除掉 RMSProp normalize 后建议的 learning rate 分母，然后得到 update 方向
  $$\theta_t = \theta_{t-1} - \frac{\alpha \cdot \hat{m}_t}{\sqrt{\hat{v}_t} + \varepsilon}$$

Regularization

L2 regularization

Regularization term 可以是参数的 L2 norm，所谓的 L2 norm，就是把 model 参数集θ里的每一个参数都取平方然后求和，这件事被称作 L2 regularization，即

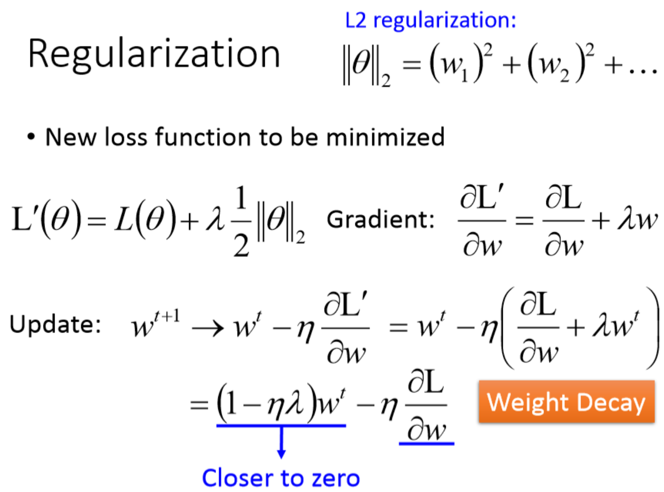
$||\theta||_2 = (w_1)^2 + (w_2)^2 + \cdots\cdots$

通常我们在做 regularization 的时候，新加的 term 里是不会考虑 bias 这一项的，因为加 regularization 的目的是为了让我们的 function 更平滑，而 bias 通常是跟 function 的平滑程度没有关系的

你会发现我们新加的 regularization term $\lambda\frac{1}{2}||\theta||_2$里有一个$\frac{1}{2}$，由于我们是要对 loss function 求微分的，而新加的 regularization term 是参数w_i的平方和，对平方求微分会多出来一个系数 2，我们的$\frac{1}{2}$就是用来和这个 2 相消的

L2 regularization 具体工作流程如下:

- 我们加上 regularization term 之后得到了一个新的 loss function：$L'(\theta)=L(\theta)+\lambda\frac{1}{2}||\theta||_2$
- 将这个 loss function 对参数w_i求微分:$\frac{\partial L'}{\partial w_i}=\frac{\partial L}{\partial w_i}+\lambda w_i$
- $w_i^{t+1}=w_i^t-\eta\frac{\partial L'}{\partial w_i}=w_i^t-\eta\left(\frac{\partial L}{\partial w_i}+\lambda w_i\right)=(1-\eta\lambda)w_i^t-\eta\frac{\partial L}{\partial w_i}$



如果把这个推导出来的式子和原来作比较，你会发现参数w_i在每次 update 之前，都会乘上一个$(1-\eta\lambda)$，而η和λ通常会被设为一个很小的值，因此$(1-\eta\lambda)$通常是一个接近于 1 的值，比如 0.99；也就是说，regularization 做的事情是，每次 update 参数w_i之前，不分青红皂白就先对原来的w_i乘上 0.99，这就意味着，随着 update 次数增加，参数w_i会越来越接近 0

使用 L2 regularization 可以让 weight 每次都变得更小一点，这就叫做 Weight Decay(权重衰减)

L1 regularization

$L1\ regularization: ||\theta||=|w_1|+|w_2|+\cdots\cdots$

如果w是正的，那微分出来就是+1，如果w是负的，那微风出来就是-1，所以这边写了一个w的 sign function，它的意思是说，如果w是正数的话，这个 function output 就是+1，w是负数的话，这个 function output 就是-1

L1 regularization 的工作流程如下:

- 我们加上 regularization term 之后得到了一个新的 loss function: $L'(\theta)=L(\theta)+\lambda||\theta||_1$
- 将这个 loss function 对参数w_i求微分: $\frac{\partial L'}{\partial w_i}=\frac{\partial L}{\partial w_i}+\lambda sgn(w_i)$
- 然后 update 参数w_i:
- $w_i^{t+1}=w_i^t-\eta\frac{\partial L'}{\partial w_i}=w_i^t-\eta\left(\frac{\partial L}{\partial w_i}+\lambda sgn(w_i)\right)=w_i^t-\eta\frac{\partial L}{\partial w_i}-\eta\lambda sgn(w_i)$

L1 vs L2

$$L1: w_i^{t+1}=w_i^t-\eta\frac{\partial L}{\partial w_i}-\eta\lambda sgn(w_i)$$

$$L2: w_i^{t+1}=(1-\eta\lambda)w_i^t-\eta\frac{\partial L}{\partial w_i}$$

L1 和 L2，虽然它们同样是让参数的绝对值变小，但它们做的事情其实略有不同：

- L1 使参数绝对值变小的方式是每次 update **减掉一个固定值**
- L2 使参数绝对值变小的方式是每次 update **乘上一个小于 1 的固定值**

因此，当参数w的绝对值比较大的时候，L2 会让w下降的更快，而 L1 每次 update 只让w减去一个固定值，train 完

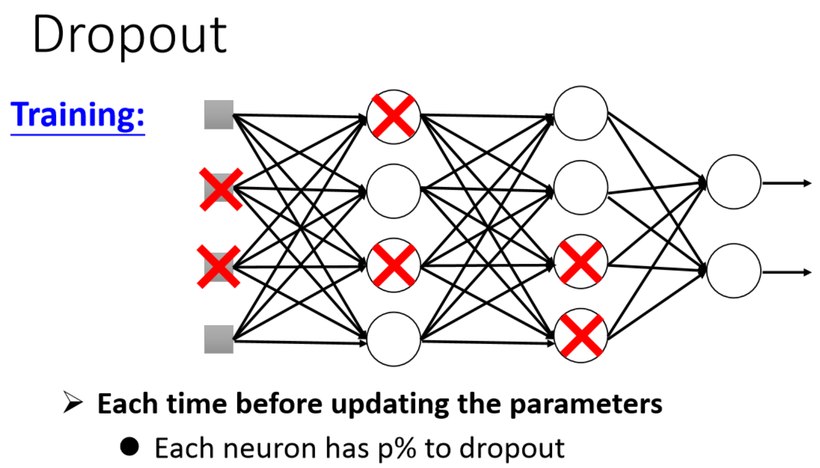
以后可能还会有很多比较大的参数；当参数w的绝对值比较小的时候，L2 的下降速度就会变得很慢，train 出来的参数平均都是比较小的，而 L1 每次下降一个固定的 value，train 出来的参数是比较 sparse 的，这些参数有很多是接近 0 的值，也会有很大的值

Dropout

Training

在 training 的时候，每次 update 参数之前，我们对每一个 neuron(也包括 input layer 的" neuron")做 sampling(抽样)，每个 neuron 都有 p%的几率会被丢掉，如果某个 neuron 被丢掉的话，跟它相连的 weight 也都要被丢掉

实际上就是每次 update 参数之前都通过抽样只保留 network 中的一部分 neuron 来做训练

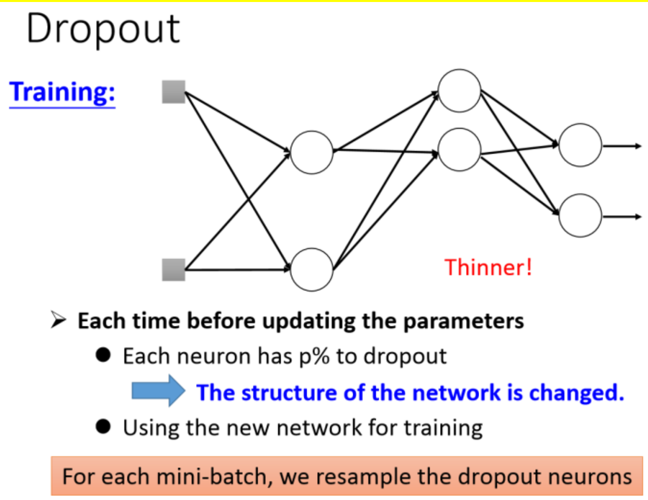


做完 sampling 以后，network structure 就会变得比较细长了，然后你再去 train 这个细长的 network

注:每次 update 参数之前都要做一遍 sampling，所以每次 update 参数的时候，拿来 training 的 network structure 都是不一样的；你可能会觉得这个方法跟前面提到的 Maxout 会有一点相似，但实际上，Maxout 是每一笔 data 对应的 network structure 不同，而 Dropout 时每一次 update 的 network structure 都是不同的(每一个 minibatch 对应着一次 update，而一个 minibatch 里含有很多笔 data)

当你在 Training 的时候使用 dropout，得到的 performance 其实是会变差的，因为某些 neuron 在 training 的时候莫名其妙就会消失不见，但这并不是问题，因为:

Dropout 真正要做的事情，就是要让你在 training set 上的结果变差，但是在 testing set 上的结果是变好的



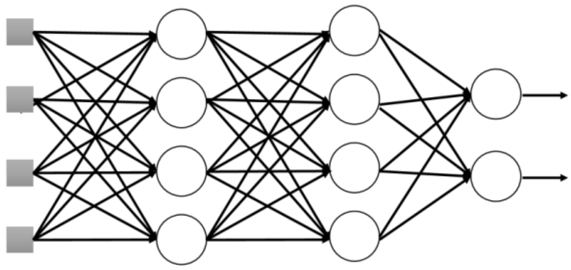
所以如果你今天遇到的问题是在 training set 上得到的 performance 不够好，你再加 dropout，就只会越做越差；这告诉我们，不同的 problem 需要用不同的方法去解决，而不是胡乱使用，dropout 就是针对 testing set 的方法，当然不能够拿来解决 training set 上的问题啦!

Testing

在使用 dropout 方法做 testing 的时候要注意两件事情:

- Testing 的时候不做 dropout，所有的 neuron 都要被用到
- 假设在 training 的时候，dropout rate 是 p%，从 training data 中被 learn 出来的所有 weight 都要乘上(1-p%)才能被当做 testing 的 weight 使用

Testing:



➢ **No dropout**

- If the dropout rate at training is p%,
  all the weights times 1-p%
- Assume that the dropout rate is 50%.
  If a weight $w = 1$ by training, set $w = 0.5$ for testing.