**神经网络训练中的Tricks之高效BP（反向传播算法）**

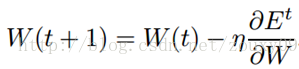
<http://blog.csdn.net/zouxy09/article/details/45288129>

Neural Networks: Tricks of the Trade 原书

真正的干货终于要登场了。上面说到，BP是一阶梯度方法，所以BP是很慢的。特别是在多层网络的时候，代价函数曲面一般是非二次、非凸和高维的，因此有非常多的局部最小值和（或者）平flat的区域。所以BP算法无法保证：1）网络会收敛到一个好的解；2）收敛是迅速的；3）收敛总会出现。然而，本节中我们会讨论一系列的tricks，这些tricks一般可以非常有效的增加上述这些情况出现的机会，也就是在可以数量级的降低收敛时间的基础上找到一个不错的解。注意是数量级哦。是不是很期待呢。

**1）Stochastic Learning vs Batch Learning**

因为我们的代价函数是在整个训练集上取平均，所以在对权重的每次迭代更新中，都需要对数据库中所有的样本过一遍，然后求解平均或者真实的梯度。这种方式叫batch学习，因为每次的参数更新需要考虑了完整的batch数据。相对的，我们可以使用随机或者**在线stochastic (online)学习**的方式，也就是每次从训练集中选择（例如，随机的）一个样本{Zt, Dt}来计算梯度。这时候，梯度的估计只通过这一个样本进行估计获得，这时候t时刻对模型参数的更新为：



因为这个对梯度的估计是有噪声的，所以在每次的迭代中，参数可能不是很精确的沿着梯度下降的方式走。但是，我们将会看到，这种每次迭代引入的“噪声”实际上是有利的。随机学习因为下面三个原因而备受青睐：

A、 随机学习一般比batch学习收敛更快；

B、 随机学习一般会得到更好的解；

C、 随机学习对跟踪网络的变化很有用。

首先，我们来分析下第一点。随机学习在大部分情况下会比batch学习要快，特别是在大规模的冗余数据库中。原因很简单。假设我们有这样一个数据库，它有1000的样本，但由于我们不小心，这1000个样本是把100个样本复制了10份得到的。所以，对这全部1000个样本进行梯度平均，和对这100个样本进行梯度平均的结果是一模一样的。因此，batch梯度下降就很浪费了，因为对一次参数更新它重复计算了10次。做了很多无用功。而随机梯度只是相对于一个100样本（假设batch包括100个样本）的训练集迭代了10次。实际上，在数据库中，一个样本很少出现两次，但数据库里面还是存在很多很相似的样本的。例如在音素分类中，所有包含音素/æ/的模式基本上都包含了相似的信息，所以这种冗余就使得batch学习要比在线学习慢得多。

第二点，随机学习可以得到更好的解是因为它给我们的梯度更新带来了噪声。有利的噪声可以美其名叫扰动。非线性网络一般具有很多不同深度的局部极小值。训练的目标是找到其中一个极小值。Batch学习发现的极小值是依据参数初始化在代价函数表面的某个坑上面，所以如果参数一旦初始化了，因为梯度下降都是往低的地方走，所以最后的归宿一般就确定了，它一定是掉到这个坑里面。如果人生过得循规蹈矩，不信命不行。不过在随机学习中，由于噪声的存在，有时候会使参数跳到另一个坑中，从而有可能找到更深的局部极小值。更深的局部极小值意味着更小的代价函数值，也就是更拟合数据的模型。所以人生多姿多彩，还是有不同的收获的。

随机学习还有一个很有用的场景，就是我们要建模的函数在时间上是变化。一个非常常见的场景是在工业应用领域。这里数据的分布随着时间会变化（例如，由于机器的磨损）。如果我们的模型不能检测和适应这个变化，他们它就无法学习数据，从而导致非常大的泛化误差。时代在发展，思想得改革。对batch，由于我们需要在少数的rules中进行平均，所以这种变化很难被检测，从而获得不好的结果。但在线学习，如果操作得当，可以跟踪这个变化，从而得到更近似的结果。

       尽管随机学习是有上述这些讨人喜欢的地方，但batch学习也并非一无是处，还是存在一些理由让我们考虑batch学习的。Batch学习的优点：

A、 收敛条件易于理解；

B、 很多加速的方法，例如共轭梯度，只在batch学习中有效；

C、 对参数变化和收敛率的理论分析更简单。

为了平等，我们也来分析下batch的这些优点。凡事都有两面性，这个“噪声”也有利有弊，它的“利”成就了随机学习，而它的“弊”也将大家对随机学习的一部分爱推向了batch学习。换句话说，batch学习上述这些优点就是因为没有这个“噪声”而带来的。这个噪声对找到更好的局部极小值很关键，但它也会阻止完全的收敛到局部极小值，它会让代价函数在极小值周围徘徊，它很想下去，但却有心无力，因为它生于噪声而被噪声束缚。这种噪声会导致模型的参数发生抖动，就算在局部极小值附近，也不稳定，一直折腾。这个抖动的大小也取决于随机更新的噪声大小。在局部极小值附近的抖动的方差和学习率是成比例的。所以为了减小这种抖动，我们有两种方法：1）减小学习率（退火）；2）使用一个自适应的batch大小。对第一种方法，有理论说明了对学习率调整最优的退火过程是：η=c/t。c是一个常数，t是样本个数。实际应用中，这个学习率还是有点大。

另一种方法也很自然，解铃还须系铃人，谁带来的烂摊子就谁来收拾。那就是想办法去掉噪声。咦？那随机学习的那些优点不是也因为噪声的去除而消失？凡事都有中庸之道，存在中庸之法，极端不可取，那就取矛盾的折衷。mini-batches正式登上历史舞台来平衡这各方压力。很简单，训练一开始，我们的参数刚初始化，离最小值还很远，这时候我们就要加快它前进的步伐，因此借助随机学习的收敛速度，我们采用一个很小的mini-batches，也就是每个batch包含的训练样本数不多。随着训练的进行，离最小值越来越近，我们就得减速了，否则冲过界了，又得抖来抖去了。因此我们增加mini-batches的大小，从而降低噪声。然而，每种方法的引入都会引入另外需要考虑的超参，在这里就是应该对mini-batches的大小选择怎样的增长率？这实际上和选择学习率是同样困难的。殊途同归，有效的调整学习率和有效的调整mini-batches的大小增长率效果差不多。

不过，值得注意的一点是，因为考虑到模型的泛化能力，所以也有认为去除数据中的噪声就显得不那么关键了。因为还没遇到噪声的这些弊病的时候就已经过训练（过拟合）了。

另一个batch训练的优点是可以使用二阶优化方法来加速学习过程。二阶方法不仅估计了代价函数曲面在某点处的梯度（一阶信息），同时还估计了曲面的曲率（二阶信息）。拿到曲率后，它就可以估计真实最小值的近似位置，以此进行强有力的加速。

尽管batch学习有这样那样的优点，随机学习还是大家比较青睐的方法，特别是训练的数据库非常大的时候，它的确更快。天下武功，唯快不破！

**2）Shuffling打乱样本的学习顺序**

有一个原则是，网络从意料之外的样本中学习最快。枪打出头鸟，就像如果第一天上课，你很突出，很捣蛋，那你老师肯定记住了你，而不是其他“平凡”的学生。所以思想就很简单了，为了加速学习，在每次迭代的时候我们挑选一个和系统最不相似、最不和谐的样本让网络去学习。把“出头鸟”拎出来，擒贼先擒王。很明显，这个方法只对随机学习有效，因为batch是不管顺序的，不管先来后到，都得等齐人了，才发粮草（计算全部样本的总误差再去做梯度更新）。当然了，没有很简单的方法可以知道到底哪个输入样本携带了对系统最丰富的信息量，不过有个简单的trick就是粗糙地选择来自不同类的样本，换句话来说，就是，如果在第t次迭代，我是用第i类的样本来学习的，那么在第t+1次迭代的时候，就选择除i类外的其他类的一个样本来学习。因为同一个类的训练样本很大可能携带的是相似的信息，所以我这次见到你了，下次就不想见到和你长得差不多的人了，没什么信息量，审美疲劳。

另一种启发式的判断到底一个训练样本携带了多少新信息的方法，就是测试当将这个样本输入到网络的时候，网络的输出值和目标输出值的误差大小。误差越大，那就表示网络还没学习到这个样本，因此它具有更多的新的信息。就像突然你的世界出现了一个新鲜事物一样，那种“哎呀”的感觉。所以，我们偏心的将这个样本多次输入网络去学习也是有意义的。当然了，这个误差的“大”是相对于其他训练样本来说的。随着网络的训练，每个输入样本的这个误差都会变化，所以每个样本被输入网络训练的次数也会变化。有个修改每个样本的这个概率或者次数的方法叫emphasizing scheme：

A、 打乱训练集，使得邻近的样本几乎不会属于同一个类；

B、 挑会使网络产生更大误差的样本输入网络学习。

然而，需要小心的是，打乱输入样本被学习的正常频率，会改变每个样本对网络的重要程度，这可能不是那么好。让一部分人先富起来，富起来后他们就不管后富了，这贫富差距就大了。千万宠爱集于几人，朱门酒肉臭，路有被冷落死的样本。这种不利于社会和谐的政策还是不要太极端的好。举个极端的例子，如果训练集中有离群点outliers，那将带来灾难性的后果。因为离群点可以产生很大的误差，但很明显，不应该将它多次地送给网络去训练，这样会扰乱这个网络的正常学习。网络为这个样本调整了半天的参数，然后发现这个是个很不正常的样本，那乖乖，无力吐槽。不过，这个trick对一种情况非常有用，那就是可以对那些正常的但很少出现的输入模式进行性能的加速，例如在音素识别中/z/这个音素。如果这个样本是个正常的小众，那让网络多次学习它是有益的。关注弱小群体，构建和谐社会。

**3）对输入进行标准化Normalize**

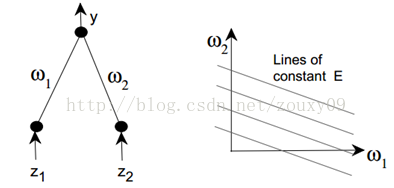
如果训练样本中每个输入变量（特征维度）的均值接近于0，那收敛一般都会更快。我们考虑个极端的情况。也就是网络所有的输入都是正数。第一个隐层的神经元的参数更新值是和δx成比例的，δ是这个神经元的误差，x是输入的向量。当x所有的元素都是正数的时候，对这个神经元的参数的更新值都具有相同的符号（因为x是正数，所以更新值的符号和δ的符号一致，而δ是一个标量）。这就导致了，这些参数对一个给定的输入样本，要么全部增加（δ是正数），要么全部减小（δ是负数）。所以，如果一个参数向量到达到最优值是必须要改变方向的话，那么它就会沿着“之”形状的路径前进，这是非常低效的，所以会导致收敛非常慢。

上述例子中，所有的输入都是正数。然而，实际上，如果训练样本的输入变量的均值远离于0，都会让参数的更新倾向于一个特定的方向，从而降低了学习的速度。因此，将整个训练集每个样本的输入变量的均值偏移到0处是有好处的。而且，这种启发式的方法应该在网络的每一层都使用上，换句话说，我们希望每个节点的输出的均值都接近于0，因为这些输出实际上是下一层的输入。不过这个问题，可以将对输入的变换和对sigmoid激活函数的选择共同考虑。这里我们讨论对输入的变换。后面再讨论sigmoid函数。

除了对样本进行平移外，还有一个加速收敛的方法是对样本进行缩放，让每一个特征维度都具有相同的协方差。缩放为什么会加速学习？因为它可以平衡与输入节点连接的参数的学习率。什么意思呢？上面提到第一个隐层的神经元的参数更新值是和δx成比例的，那如果x中有些元素的值很大，而有些元素的值很小，那很明显，值大的会导致参数的更新值也很大。值小的更新值也小，哦，贫富差距又来了。不过，上面只是他们的方差值要相同，那应该取多少呢？这个值应该和sigmoid的选择相匹配。对下面给定的sigmoid函数（说了下面给定，那肯定得在下面才能看到啦，囧），协方差取1是个不错的选择。

不过也有例外的情况，那就是当你事先知道某些输入变量的重要性要比其他输入变量弱的时候。这种情况下，可以将重要性小的输入变量缩小，这样就可以让学习算法轻微的“忽略”它了。人的先验为大嘛。

上述对输入进行平移和缩放的tricks是很容易实现的。还有一个也很有效，但比较难实现的tricks是对输入进行解相关。考虑下图所示的简单网络。如果输入是独立的，也就是不相关的，那么就可以通过最小化而得到w1的解，而不用去考虑w2，反之亦然。如果输入是相关的，那么就需要同时解两个参数w，这明显要更难点。那如何去对输入变量进行解相关呢？鼎鼎大名的PCA登上历史舞台。不过它能力也有限哦，只能用来移除输入的线性相关性（只能搞定二阶，高阶就鞭长莫及了）。



实际上，如果输入是线性独立的（相关的极端情况）也可能会产生某种降低学习速度的退化。考虑一种情况是，当一个输入变量总是另一个输入的两倍z2=2z1。那网络沿着线W2=v-(1/2)W1（v是个常数）的输出就都是常数。因此，在这个方向的梯度就都是0了。因此在这些线上移动对学习不会起到任何的效果。我们本来是想尝试解决一个二维的问题的，但这个问题实际上在一维的时候才是有效的。吃力不讨好。因此，理想情况下，我们希望去掉这一个输入，减小网络的大小。

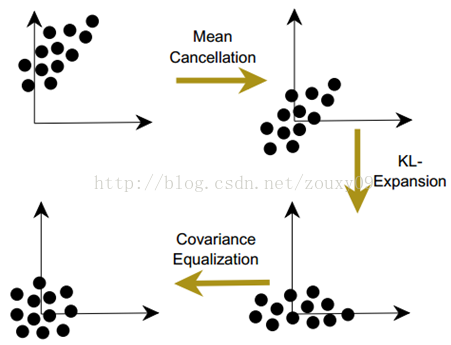
总结来说，对输入的变换如下：

A、 训练集的每个输入变量的均值要接近于0；

B、 对输入变量进行缩放，使他们的方差具有相同的值；

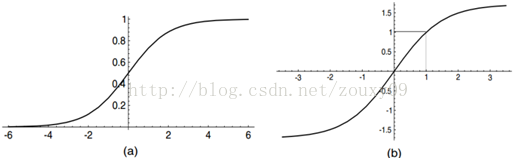
C、 输入变量最好是不相关的。

       这个过程可以表达如下：1）平移输入让他们的均值为0；2）对输入解相关；3）均衡化协方差。如下图所示：



**4）sigmoid函数**

       非线性激活函数的使用赋予了神经网络对非线性函数的建模能力。如果没有他，无数隐层的神经网络还是一个线性网络。家喻户晓的激活函数非sigmoid莫属了。它是单调递增的，通过在正负无穷大的时候是渐进于某个有限值。一般取标准的逻辑函数f(x)=1/(1+e-x)和双曲线正切函数f(x)=tanh(x)。人们往往更喜欢关于原点对称版本的Sigmoid函数（双曲线正切函数），因为上面我们提到输入应该要满足标准化，所以这个函数的输出更有可能为下一层创造均值接近于0的输入。相反，Logistic函数因为输出总是正数，因此它的均值也总是正数。



(a)标准Logistic函数。(b) 双曲线正切函数f(x)=1.7159tanh(2x/3)

对Sigmoids函数的Tricks如下：

A、 对称性的sigmoids函数例如双曲线正切函数往往比标准的Logistic函数收敛更快。

B、 一个建议的激活函数是f(x)=1.7159tanh(2x/3)。因为tanh函数计算挺耗时的，所以一般可以用多项式的系数来近似。

C、 有时候，增加一个线性项会很有用，例如f(x)=tanh(x)+ax，这样可以避免代价函数曲面flat的地方。

我们上面建议你使用的那个激活函数，连参数都给你选择好了。因为当你使用的是标准化的输入后，这个激活函数输出的方差也会接近于1，因为sigmoid的effective gain（有效增益？）在它的有效范围内大致为1。这个特别版本的sigmoid具有以下性质：a）f(正负1)=正负1；b) 最大的二次导数出现在x=1的地方；c）有效增益接近于1。

当然了，凡事依然有两面性，使用对称性sigmoid也有它的缺点，那就是它会使得误差表面在接近原点的地方会非常平flat。因为这个原因，所以最好可以避免将网络参数初始化为很小的值。因为sigmoids的饱和，误差表面在远离原点的时候也是flat的。在sigmoid中增加一个线性的项有时候可以避开这些flat的区域。

**5）目标值的选择**

在分类问题上，目标值一般都是二值的，例如{-1,+1}。很多智者都建议把目标值设置为sigmoid的渐进线的地方。然而，这种做法有些弊端：

首先，会导致不稳定。我们知道，网络的训练会尽自己的最大努力让网络的输出尽可能的接近于目标值，当然了，只能渐进的接近。这样，网络的参数（输出层，甚至隐层）会变得越来越大，而在这些地方，sigmoid的导数值接近于0。这些非常大的参数会增加梯度的值，然而，这些梯度接下来会乘以非常小的sigmoid导数（除非增加一个twisting扭曲项，也就是之前说的增加个线性项ax）从而导致最后的参数更新值也接近于0。最终导致的惨不忍睹的结果就是参数被卡住了，动不了啦。

第二，当输出饱和时，网络无法给出置信度的指示。首先，置信度是个啥？说白了，置信度就是当你给神经网络输入一个样本的时候，我，神经网络，要给你分类是不是。我经过一轮辛勤的计算，给你一个分类结果对不对。那你就百分比相信我给你的结果是正确的？作为一个负责任的网络，我是不是还得告诉你，我做出这个分类判断的可信度是多少？这样才会给你的下一步决策提供参考，信不信由你。例如，当一个输入样本落在决策边界附近的时候，网络输出的决策值实际上是不确定的。理想情况下，这种置信度应该要被反映在网络中，比如输出一个在两个可能的目标值之间的某个值，而不是在两端渐进线的地方。然而，大的参数会强制所有的输出都落在sigmoid的尾部，而不去考虑不确定性。典型的妄自尊大呀。因此，在没有给出任何关于这个结果置信度很低的指示，网络就可能给的是一个错误的类别结果，这不坑人嘛。大的参数会导致神经元的饱和，蒙蔽了它的双眼，从而使其丧失了对样本基本的区分能力。

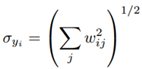
解决这个问题的一个方法就是把目标值设置在sigmoid的有效范围内，而不是在渐进线的区域。还需要小心的是，为了保证节点不会只被限制在sigmoid的线性部分，可以把目标值设置在sigmoid的最大二阶导数的位置，这样不但可以利用非线性的优点，还可以避免sigmoid的饱和。这也是上图b中的sigmoid函数是个不错的选择的原因。它在正负1的地方具有最大的二阶导数，而正负1对应的恰好是分类问题的典型二值目标值。

Trick：目标值：将目标值选择在sigmoid函数最大二阶导数的位置。从而避免输出节点的饱和。

**6）参数的初始化**

参数的初始值对训练过程有着重大的影响。我们对参数初始化的原则是：参数应该随机初始化在能让sigmoid函数在线性区域激活的值。如果参数全部都很大，那sigmoid一开始就饱和了，这样就会得到一个非常小的梯度值，那参数更新就会很慢，训练也会很慢。如果参数太小了，那梯度也会很小，同样也会导致训练很慢。中庸之道！参数处于sigmoid线性范围的那段区域有几个优点：1）梯度可以足够的大，从而使得学习能正常进行；2）网络可以在学习映射的非常困难的非线性部分之前学习映射的线性部分。

达到这个目标并非sigmoid一家之力可以完成。它需要数据标准化、sigmoid的选择和参数初始化的选择这三者的协调。首先，我们要求每个节点的输出的标准差应该接近于1，这可以通过使用之前提到的数据标准化来对训练集进行变换获得。为了可以在第一个隐层的输出同样获得标准差为1的输出，我们只需要使用上面建议的sigmoid函数，同时要求sigmoid的输入的标准差也为1。假设一个结点的输入yi是不相关的，而且方差为1，那结点的标准差就是参数的加权和：



因此，为了保证上述这个方差近似于1，参数就应该从一个均值为0，标准差为：σw=m-1/2的分布中随机采样得到（m是fan-in，也就是与这个结点连接的输入个数，也就是前一层的节点个数，如果是全连接网络的话）。

Trick：参数初始化：

假设：1）训练集已经被标准化；2）sigmoid是选择f(x)=1.7159tanh(2x/3)。

那参数就应该从一个均值为0，标准差为σw=m-1/2的分布（例如正态分布）中采样得到。

**7）学习率的选择**

对学习率的选择也是一个大学问。和上面说的batch的大小选择一样。学习率，就是参数更新每一步走多远，这个参数很关键。如果设置的太大，那么很容易就在最优值附加徘徊，因为你步伐太大了。例如要从广州到上海，但是你的一步的距离就是广州到北京那么远，没有半步的说法，自己能迈那么大步，是幸运呢？还是不幸呢？事物总有两面性嘛，它带来的好处是能很快的从远离最优值的地方回到最优值附近，只是在最优值附近的时候，它有心无力了。但如果设置的太小，那收敛速度就太慢了，像蜗牛一样，虽然会落在最优的点，但是这速度如果是猴年马月，我们也没这耐心啊。所以有的改进就是在这个学习率这个地方下刀子的。我开始迭代是，学习率大，慢慢的接近最优值的时候，我的学习率变小就可以了。所谓采两者之精华啊！

实际上，学习的方法经过那么久的发展，对学习率的研究还是有不少的成果或者经验的。至少存在着一个很好的估计理想学习率的方法。而且还存在着很多其他的自动调整学习率的方法，不过大部分都是经验性的。

大部分这些方法都是在参数发生震荡的时候减小学习率，而在参数相对稳定的朝着一个方向前进的时候增加学习率。这个方法的主要问题在于它对随机梯度或者在线学习是不合适的，因为参数在所有的训练过程中都是抖动的。从一开始，到最后，它都是抖动的。人生如此坎坷，希望在哪。

与为所有参数选择一个同样的全局学习率相对，可以为每个参数选择不同的学习率，这样一般都可以加快收敛速度。有种不错的方法就是计算二阶导数来实现的，这个会在后面提到。这个方法最需要确定的是网络中的所有参数都会以差不多的速度收敛。这具体取决于误差表面的曲率，一些参数可能需要小的学习率来避免发散，而一些参数需要大的学习率要加快收敛速度。因为这个原因，低层的学习率一般要比高层的大。这是因为对大部分的神经网络框架而言，代价函数对低层网络参数的二阶导数要比对高层参数的小。

如果网络中使用了共享参数，例如TDNN或者CNN，那学习率应该和共享参数的连接个数的平方根成比例。因为我们知道，梯度是一些或多或少独立的项的和。

Tricks：

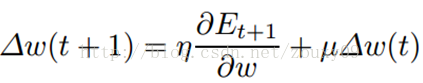
A、 给每个参数自己的学习率；

B、 学习率应该和该节点的输入个数的平方根成比例；

C、 低层参数的学习率应该比高层的大。

还有一些加快收敛的tricks包括：

Momentum：



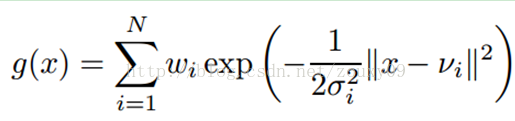
       当代价函数表面是高度非球形的，Momentum可以提高收敛速度。因为它可以限制大曲率方向的步长过大，因此可以在低曲率方向得到一个更有效的学习率。（μ衡量的是矩项的强度）。江湖中，有种说法，就是矩在batch学习比在随机模式中要有效得多，但这个说法没有什么系统的研究。

自适应学习率：

       主要是在训练中根据误差来实时调整学习率。（因为问题比较大，此处略去。有兴趣的参考原文）。

**8）RBF vssigmoid 节点**

       尽管大部分的系统都是使用基于点积和sigmoid的神经元，不过还是存在其他选择的。一个比较典型的就是RBF径向基网络了。在RBF网络中，参数和输入向量的点积被替换为两者的欧式距离，同时sigmoid函数变为指数函数。例如对一个输入x，它的输出为：



vi(σi)是第i个高斯的均值和标准差。这些节点可以替换标准的节点，也可以和他们共存。他们一般是通过梯度下降（对输出层）和非监督聚类算法对RBF节点的均值和方差进行学习。这个可以参考我的[这个博文](http://blog.csdn.net/zouxy09/article/details/13297881)。

与sigmoid节点不同，sigmoid可以覆盖整个空间。但一个RBF节点只能覆盖输入空间的一小局部区域。这样对快速学习是有利的。RBF还可以构建一组能更好的对输入空间进行建模的基函数，不过这是问题独立的。RBF还是有缺点的，它的局部特性可能会带来一些不好的性能，特别是在高维空间中，就需要非常多的节点才可以覆盖整个空间。不过，有人也得到了一些网络设置的经验，就是在网络的低层（高维）用sigmoid，在高层（低维）使用RBF。