# [浅谈深度学习(Deep Learning)的基本思想和方法](http://blog.csdn.net/xianlingmao/article/details/8478562)

深度学习（Deep Learning），又叫Unsupervised Feature Learning或者Feature Learning，是目前非常热的一个研究主题。

本文将主要介绍Deep Learning的基本思想和常用的方法。

一. 什么是Deep Learning？

实际生活中，人们为了解决一个问题，如对象的分类（对象可是是文档、图像等），首先必须做的事情是如何来表达一个对象，即必须抽取一些特征来表示一个对象，如文本的处理中，常常用词集合来表示一个文档，或把文档表示在向量空间中（称为VSM模型），然后才能提出不同的分类算法来进行分类；又如在图像处理中，我们可以用像素集合来表示一个图像，后来人们提出了新的特征表示，如SIFT，这种特征在很多图像处理的应用中表现非常良好，特征选取得好坏对最终结果的影响非常巨大。因此，选取什么特征对于解决一个实际问题非常的重要。

然而，手工地选取特征是一件非常费力、启发式的方法，能不能选取好很大程度上靠经验和运气；既然手工选取特征不太好，那么能不能自动地学习一些特征呢？答案是能！Deep Learning就是用来干这个事情的，看它的一个别名Unsupervised Feature Learning，就可以顾名思义了，Unsupervised的意思就是不要人参与特征的选取过程。因此，自动地学习特征的方法，统称为Deep Learning。

二. Deep Learning的基本思想

假设我们有一个系统S，它有n层（S1,...Sn），它的输入是I，输出是O，形象地表示为： I =>S1=>S2=>.....=>Sn => O，如果输出O等于输入I，即输入I经过这个系统变化之后没有任何的信息损失，保持了不变，这意味着输入I经过每一层Si都没有任何的信息损失，即在任何一层Si，它都是原有信息（即输入I）的另外一种表示。现在回到我们的主题Deep Learning，我们需要自动地学习特征，假设我们有一堆输入I（如一堆图像或者文本），假设我们设计了一个系统S（有n层），我们通过调整系统中参数，使得它的输出仍然是输入I，那么我们就可以自动地获取得到输入I的一系列层次特征，即S1，..., Sn。

另外，前面是假设输出严格地等于输入，这个限制太严格，我们可以略微地放松这个限制，例如我们只要使得输入与输出的差别尽可能地小即可，这个放松会导致另外一类不同的Deep Learning方法。上述就是Deep Learning的基本思想。

三.Deep Learning的常用方法

a). AutoEncoder

最简单的一种方法是利用人工神经网络的特点，人工神经网络（ANN）本身就是具有层次结构的系统，如果给定一个神经网络，我们假设其输出与输入是相同的，然后训练调整其参数，得到每一层中的权重，自然地，我们就得到了输入I的几种不同表示（每一层代表一种表示），这些表示就是特征，在研究中可以发现，如果在原有的特征中加入这些自动学习得到的特征可以大大提高精确度，甚至在分类问题中比目前最好的分类算法效果还要好！这种方法称为AutoEncoder。当然，我们还可以继续加上一些约束条件得到新的Deep Learning方法，如如果在AutoEncoder的基础上加上L1的Regularity限制（L1主要是约束每一层中的节点中大部分都要为0，只有少数不为0，这就是Sparse名字的来源），我们就可以得到Sparse AutoEncoder方法。

b). Sparse Coding

如果我们把输出必须和输入相等的限制放松，同时利用线性代数中基的概念，即O = w1\*B1 + W2\*B2+....+ Wn\*Bn， Bi是基，Wi是系数，我们可以得到这样一个优化问题：

                                                                                    Min |I - O|

通过求解这个最优化式子，我们可以求得系数Wi和基Bi，这些系数和基础就是输入的另外一种近似表达，因此，它们可以特征来表达输入I，这个过程也是自动学习得到的。如果我们在上述式子上加上L1的Regularity限制，得到：

                                                                                    Min |I - O| + u\*(|W1| + |W2| + ... + |Wn|)

这种方法被称为Sparse Coding。

c) Restrict  Boltzmann Machine (RBM)

假设有一个二部图，每一层的节点之间没有链接，一层是可视层，即输入数据层（**v**)，一层是隐藏层(**h**)，如果假设所有的节点都是二值变量节点（只能取0或者1值），同时假设全概率分布p(**v, h**)满足Boltzmann 分布，我们称这个模型是Restrict  Boltzmann Machine (RBM)。下面我们来看看为什么它是Deep Learning方法。首先，这个模型因为是二部图，所以在已知**v**的情况下，所有的隐藏节点之间是条件独立的，即p(**h**|**v**) =p(**h**1|**v**).....p(**h**n|**v**)。同理，在已知隐藏层**h**的情况下，所有的可视节点都是条件独立的，同时又由于所有的v和h满足Boltzmann 分布，因此，当输入**v**的时候，通过p(**h**|**v**) 可以得到隐藏层**h**，而得到隐藏层**h**之后，通过p(**v**|**h**) 又能得到可视层，通过调整参数，我们就是要使得从隐藏层得到的可视层**v1**与原来的可视层**v**如果一样，那么得到的隐藏层就是可视层另外一种表达，因此隐藏层可以作为可视层输入数据的特征，所以它就是一种Deep Learning方法。

如果，我们把隐藏层的层数增加，我们可以得到Deep Boltzmann Machine (DBM)；如果我们在靠近可视层的部分使用贝叶斯信念网络（即有向图模型，当然这里依然限制层中节点之间没有链接），而在最远离可视层的部分使用Restrict  Boltzmann Machine，我们可以得到Deep Belief Net （DBN） 。

当然，还有其它的一些Deep Learning 方法，在这里就不叙述了。总之，Deep Learning能够自动地学习出数据的另外一种表示方法，这种表示可以作为特征加入原有问题的特征集合中，从而可以提高学习方法的效果，是目前业界的研究热点。

# [Deep Learning的基本思想和方法](http://blog.csdn.net/kezunhai/article/details/9859205)

          实际生活中，人们为了解决一个问题，如对象的分类（对象可是是文档、图像等），首先必须做的事情是如何来表达一个对象，即必须抽取一些特征来表示一个对象，如文本的处理中，常常用词集合来表示一个文档，或把文档表示在向量空间中（称为VSM模型），然后才能提出不同的分类算法来进行分类；又如在图像处理中，我们可以用像素集合来表示一个图像，后来人们提出了新的特征表示，如SIFT，这种特征在很多图像处理的应用中表现非常良好，特征选取得好坏对最终结果的影响非常巨大。因此，选取什么特征对于解决一个实际问题非常的重要。  
          然而，手工地选取特征是一件非常费力、启发式的方法，能不能选取好很大程度上靠经验和运气；既然手工选取特征不太好，那么能不能自动地学习一些特征呢？答案是能！Deep Learning就是用来干这个事情的，看它的一个别名Unsupervised Feature Learning，就可以顾名思义了，Unsupervised的意思就是不要人参与特征的选取过程。因此，自动地学习特征的方法，统称为Deep Learning。  
         **1）Deep Learning的基本思想**  
         假设我们有一个系统S，它有n层（S1,…Sn），它的输入是I，输出是O，形象地表示为： I =>S1=>S2=>…..=>Sn => O，如果输出O等于输入I，即输入I经过这个系统变化之后没有任何的信息损失，保持了不变，这意味着输入I经过每一层Si都没有任何的信息损失，即在任何一层Si，它都是原有信息（即输入I）的另外一种表示。现在回到我们的主题Deep Learning，我们需要自动地学习特征，假设我们有一堆输入I（如一堆图像或者文本），假设我们设计了一个系统S（有n层），我们通过调整系统中参数，使得它的输出仍然是输入I，那么我们就可以自动地获取得到输入I的一系列层次特征，即S1，…, Sn。  
        另外，前面是假设输出严格地等于输入，这个限制太严格，我们可以略微地放松这个限制，例如我们只要使得输入与输出的差别尽可能地小即可，这个放松会导致另外一类不同的Deep Learning方法。上述就是Deep Learning的基本思想。  
        **2）Deep Learning的常用方法**  
        **a). AutoEncoder**  
        最简单的一种方法是利用人工神经网络的特点，人工神经网络（ANN）本身就是具有层次结构的系统，如果给定一个神经网络，我们假设其输出与输入是相同的，然后训练调整其参数，得到每一层中的权重，自然地，我们就得到了输入I的几种不同表示（每一层代表一种表示），这些表示就是特征。在研究中可以发现，如果在原有的特征中加入这些自动学习得到的特征可以大大提高精确度，甚至在分类问题中比目前最好的分类算法效果还要好！这种方法称为AutoEncoder。当然，我们还可以继续加上一些约束条件得到新的Deep Learning方法，如在AutoEncoder的基础上加上L1的Regularity限制（L1主要是约束每一层中的节点中大部分都要为0，只有少数不为0，这就是Sparse名字的来源），我们就可以得到Sparse AutoEncoder方法。  
       **b). Sparse Coding**  
       如果我们把输出必须和输入相等的限制放松，同时利用线性代数中基的概念，即O = w1\*B1 + W2\*B2+….+ Wn\*Bn， Bi是基，Wi是系数，我们可以得到这样一个优化问题：  
                                                                  **Min |I – O|**  
通过求解这个最优化式子，我们可以求得系数Wi和基Bi，这些系数和基础就是输入的另外一种近似表达，因此，它们可以特征来表达输入I，这个过程也是自动学习得到的。如果我们在上述式子上加上L1的Regularity限制，得到：  
                                                         **Min |I – O| + u\*(|W1| + |W2| + … + |Wn|)**  
这种方法被称为Sparse Coding。  
        **c) Restrict Boltzmann Machine (RBM)**  
        假设有一个二部图，每一层的节点之间没有链接，一层是可视层，即输入数据层（**v**)，一层是隐藏层(**h**)，如果假设所有的节点都是二值变量节点（只能取0或者1值），同时假设全概率分布p(**v, h**)满足Boltzmann 分布，我们称这个模型是Restrict Boltzmann Machine (RBM)。下面我们来看看为什么它是Deep Learning方法。首先，这个模型因为是二部图，所以在已知**v**的情况下，所有的隐藏节点之间是条件独立的，即p(**h**|**v**) =p(**h**1|**v**)…..p(**h**n|**v**)。同理，在已知隐藏层**h**的情况下，所有的可视节点都是条件独立的，同时又由于所有的v和h满足Boltzmann 分布，因此，当输入**v**的时候，通过p(**h**|**v**) 可以得到隐藏层**h**，而得到隐藏层**h**之后，通过p(v|h) 又能得到可视层，通过调整参数，我们就是要使得从隐藏层得到的可视层**v1**与原来的可视层**v**如果一样，那么得到的隐藏层就是可视层另外一种表达，因此隐藏层可以作为可视层输入数据的特征，所以它就是一种Deep Learning方法。  
         如果，我们把隐藏层的层数增加，我们可以得到Deep Boltzmann Machine (DBM)；如果我们在靠近可视层的部分使用贝叶斯信念网络（即有向图模型，当然这里依然限制层中节点之间没有链接），而在最远离可视层的部分使用Restrict Boltzmann Machine，我们可以得到Deep Belief Net （DBN） 。  
        当然，还有其它的一些Deep Learning 方法，在这里就不叙述了。总之，Deep Learning能够自动地学习出数据的另外一种表示方法，这种表示可以作为特征加入原有问题的特征集合中，从而可以提高学习方法的效果，是目前业界的研究热点。  
原文链接：<http://blog.csdn.net/xianlingmao/article/details/8478562>

**学习深度架构的突破**         2006年前，尝试训练深度架构都失败了：训练一个深度有监督前馈神经网络趋向于产生坏的结果(同时在训练和测试误差中)，然后将其变浅为1(1或者2个隐层)。  
        2006年的3篇论文改变了这种状况，由Hinton的革命性的在深度信念网(Deep Belief Networks, DBNs)上的工作所引领：

* Hinton, G. E., Osindero, S. and Teh, Y., [A fast learning algorithm for deep belief nets](http://www.cs.toronto.edu/~hinton/absps/fastnc.pdf) .Neural Computation 18:1527-1554, 2006
* Yoshua Bengio, Pascal Lamblin, Dan Popovici and Hugo Larochelle, [Greedy Layer-Wise Training of Deep Networks](http://www.iro.umontreal.ca/~lisa/publications2/index.php/publications/show/190), in J. Platt et al. (Eds), Advances in Neural Information Processing Systems 19 (NIPS 2006), pp. 153-160, MIT Press, 2007
* Marc’Aurelio Ranzato, Christopher Poultney, Sumit Chopra and Yann LeCun [Efficient Learning of Sparse Representations with an Energy-Based Model](http://yann.lecun.com/exdb/publis/pdf/ranzato-06.pdf), in J. Platt et al. (Eds), Advances in Neural Information Processing Systems (NIPS 2006), MIT Press, 2007

在这三篇论文中以下主要原理被发现：

* 表示的无监督学习被用于(预)训练每一层；
* 在一个时间里的一个层次的无监督训练，接着之前训练的层次。在每一层学习到的表示作为下一层的输入；
* 用无监督训练来调整所有层(加上一个或者更多的用于产生预测的附加层)；

       DBNs在每一层中利用用于表示的无监督学习RBMs。Bengio et al paper 探讨和对比了RBMs和auto-encoders(通过一个表示的瓶颈内在层预测输入的神经网络)。Ranzato et al paper在一个convolutional架构的上下文中使用稀疏auto-encoders(类似于稀疏编码)。Auto-encoders和convolutional架构将在以后的课程中讲解。从2006年以来，大量的关于深度学习的论文被发表，一些探讨了其他原理来引导中间表示的训练，查看[Learning Deep Architectures for AI](http://www.iro.umontreal.ca/~lisa/publications2/index.php/publications/show/239)  
原文链接： <http://www.cnblogs.com/ysjxw/archive/2011/10/08/2201782.html>   
**四、拓展学习推荐**         Deep Learning 经典阅读材料：

* The monograph or review paper [Learning Deep Architectures for AI](http://www.iro.umontreal.ca/~lisa/publications2/index.php/publications/show/239) (Foundations & Trends in Machine Learning, 2009).
* The ICML 2009 Workshop on Learning Feature Hierarchies [webpage](http://www.cs.toronto.edu/~rsalakhu/deeplearning/index.html) has a [list of references](http://www.cs.toronto.edu/~rsalakhu/deeplearning/references.html).
* The LISA [public wiki](http://www.iro.umontreal.ca/~lisa/twiki/bin/view.cgi/Public/WebHome) has a [reading list](http://www.iro.umontreal.ca/~lisa/twiki/bin/view.cgi/Public/ReadingOnDeepNetworks) and a [bibliography](http://www.iro.umontreal.ca/~lisa/twiki/bin/view.cgi/Public/DeepNetworksBibliography).
* Geoff Hinton has [readings](http://www.cs.toronto.edu/~hinton/deeprefs.html) from last year’s [NIPS tutorial](http://videolectures.net/jul09_hinton_deeplearn/).

**Deep Learning工具——**[**Theano**](http://deeplearning.net/software/theano)**：**                [Theano](http://deeplearning.net/software/theano)是deep learning的Python库，要求首先熟悉Python语言和numpy，建议读者先看[Theano basic tutorial](http://deeplearning.net/software/theano/tutorial)，然后按照[*Getting Started*](http://deeplearning.net/tutorial/gettingstarted.html#gettingstarted)下载相关数据并用gradient descent的方法进行学习。学习了Theano的基本方法后，可以练习写以下几个算法：  
**有监督学习：**

1. [*Logistic Regression*](http://deeplearning.net/tutorial/logreg.html#logreg) - using Theano for something simple
2. [*Multilayer perceptron*](http://deeplearning.net/tutorial/mlp.html#mlp) - introduction to layers
3. [*Deep Convolutional Network*](http://deeplearning.net/tutorial/lenet.html#lenet) - a simplified version of LeNet5

**无监督学习：**

* [*Auto Encoders, Denoising Autoencoders*](http://deeplearning.net/tutorial/dA.html#daa) - description of autoencoders
* [*Stacked Denoising Auto-Encoders*](http://deeplearning.net/tutorial/SdA.html#sda) - easy steps into unsupervised pre-training for deep nets
* [*Restricted Boltzmann Machines*](http://deeplearning.net/tutorial/rbm.html#rbm) - single layer generative RBM model
* [*Deep Belief Networks*](http://deeplearning.net/tutorial/DBN.html#dbn) - unsupervised generative pre-training of stacked RBMs followed by supervised fine-tuning

最后呢，推荐给大家基本ML的书籍：

* [Chris Bishop, “Pattern Recognition and Machine Learning”, 2007](http://research.microsoft.com/en-us/um/people/cmbishop/prml/)
* [Simon Haykin, “Neural Networks: a Comprehensive Foundation”, 2009 (3rd edition)](http://books.google.ca/books?id=K7P36lKzI_QC&dq=simon+haykin+neural+networks+book&source=gbs_navlinks_s)
* [Richard O. Duda, Peter E. Hart and David G. Stork, “Pattern Classification”, 2001 (2nd edition)](http://www.rii.ricoh.com/~stork/DHS.html)

原文链接：<http://blog.csdn.net/abcjennifer/article/details/7826917>  
   
**五、应用实例**1、计算机视觉

* **ImageNet Classification with Deep Convolutional Neural Networks**, Alex Krizhevsky, Ilya Sutskever, Geoffrey E Hinton, NIPS 2012.
* **Learning Hierarchical Features for Scene Labeling**, Clement Farabet, Camille Couprie, Laurent Najman and Yann LeCun, IEEE Transactions on Pattern Analysis and Machine Intelligence, 2013.
* **Learning Convolutional Feature Hierachies for Visual Recognition**, Koray Kavukcuoglu, Pierre Sermanet, Y-Lan Boureau, Karol Gregor, Micha&euml;l Mathieu and Yann LeCun, Advances in Neural Information Processing Systems (NIPS 2010), 23, 2010.

2、语音识别   
微软研究人员通过与hintion合作，首先将RBM和DBN引入到语音识别声学模型训练中，并且在大词汇量语音识别系统中获得巨大成功，使得语音识别的错误率相对减低30%。但是，DNN还没有有效的并行快速算法，目前，很多研究机构都是在利用大规模数据语料通过GPU平台提高DNN声学模型的训练效率。  
在国际上，IBM、google等公司都快速进行了DNN语音识别的研究，并且速度飞快。  
国内方面，科大讯飞、百度、中科院自动化所等公司或研究单位，也在进行深度学习在语音识别上的研究。  
3、自然语言处理等其他领域  
很多机构在开展研究，但目前深度学习在自然语言处理方面还没有产生系统性的突破。  
**六、参考链接：**

1.[http://baike.baidu.com/view/9964 ... enter=deep+learning](http://baike.baidu.com/view/9964678.htm?subLemmaId=10105430&fromenter=deep+learning)  
2.<http://www.cnblogs.com/ysjxw/archive/2011/10/08/2201819.html>  
3.<http://blog.csdn.net/abcjennifer/article/details/7826917>

4.<http://blog.sciencenet.cn/blog-315535-663215.html>

# [如何正确理解深度学习（Deep Learning）的概念](http://blog.csdn.net/raodotcong/article/details/9003087)

      现在深度学习在机器学习领域是一个很热的概念，不过经过各种媒体的转载播报，这个概念也逐渐变得有些神话的感觉：例如，人们可能认为，深度学习是一种能够模拟出人脑的神经结构的机器学习方式，从而能够让计算机具有人一样的智慧；而这样一种技术在将来无疑是前景无限的。那么深度学习本质上又是一种什么样的技术呢？

**深度学习是什么**

      深度学习是机器学习领域中对模式（声音、图像等等）进行建模的一种方法，它也是一种基于统计的概率模型。在对各种模式进行建模之后，便可以对各种模式进行识别了，例如待建模的模式是声音的话，那么这种识别便可以理解为语音识别。而类比来理解，如果说将机器学习算法类比为排序算法，那么深度学习算法便是众多排序算法当中的一种（例如冒泡排序），这种算法在某些应用场景中，会具有一定的优势。

**深度学习的“深度”体现在哪里**

      论及深度学习中的“深度”一词，人们从感性上可能会认为，深度学习相对于传统的机器学习算法，能够做更多的事情，是一种更为“高深”的算法。而事实可能并非我们想象的那样，因为从算法输入输出的角度考虑，深度学习算法与传统的有监督机器学习算法的输入输出都是类似的，无论是最简单的Logistic Regression，还是到后来的SVM、boosting等算法，它们能够做的事情都是类似的。正如无论使用什么样的排序算法，它们的输入和预期的输出都是类似的，区别在于各种算法在不同环境下的性能不同。

      那么深度学习的“深度”本质上又指的是什么呢？深度学习的学名又叫深层神经网络（Deep Neural Networks ），是从很久以前的人工神经网络（Artificial Neural Networks）模型发展而来。这种模型一般采用计算机科学中的图模型来直观的表达，而深度学习的“深度”便指的是图模型的层数以及每一层的节点数量，相对于之前的神经网络而言，有了很大程度的提升。

      深度学习也有许多种不同的实现形式，根据解决问题、应用领域甚至论文作者取名创意的不同，它也有不同的名字：例如卷积神经网络（Convolutional Neural Networks）、深度置信网络（Deep Belief Networks）、受限玻尔兹曼机（Restricted Boltzmann Machines）、深度玻尔兹曼机（Deep Boltzmann Machines）、递归自动编码器（Recursive Autoencoders）、深度表达（Deep Representation）等等。不过究其本质来讲，都是类似的深度神经网络模型。

      既然深度学习这样一种神经网络模型在以前就出现过了，为什么在经历过一次没落之后，到现在又重新进入人们的视线当中了呢？这是因为在十几年前的硬件条件下，对高层次多节点神经网络的建模，时间复杂度（可能以年为单位）几乎是无法接受的。在很多应用当中，实际用到的是一些深度较浅的网络，虽然这种模型在这些应用当中，取得了非常好的效果（甚至是the state of the art），但由于这种时间上的不可接受性，限制了其在实际应用的推广。而到了现在，计算机硬件的水平与之前已经不能同日而语，因此神经网络这样一种模型便又进入了人们的视线当中。

*“2012年6月，《纽约时报》披露了Google Brain项目，吸引了公众的广泛关注。这个项目是由著名的斯坦福大学机器学习教授Andrew Ng和在大规模计算机系统方面的世界顶尖专家Jeff Dean共同主导，用16000个CPU Core的并行计算平台训练一种称为“深层神经网络”（DNN，Deep Neural Networks）”*

从Google Brain这个项目中我们可以看到，神经网络这种模型对于计算量的要求是极其巨大的，为了保证算法实时性，需要使用大量的CPU来进行并行计算。

      当然，深度学习现在备受关注的另外一个原因，当然是因为在某些场景下，这种算法模式识别的精度，超过了绝大多数目前已有的算法。而在最近，深度学习的提出者修改了其实现代码的Bug之后，这种模型识别精度又有了很大的提升。这些因素共同引起了深层神经网络模型，或者说深度学习这样一个概念的新的热潮。

**深度学习的优点**

为了进行某种模式的识别，通常的做法首先是以某种方式，提取这个模式中的特征。这个特征的提取方式有时候是人工设计或指定的，有时候是在给定相对较多数据的前提下，由计算机自己总结出来的。深度学习提出了一种让计算机自动学习出模式特征的方法，并将特征学习融入到了建立模型的过程中，从而减少了人为设计特征造成的不完备性。而目前以深度学习为核心的某些机器学习应用，在满足特定条件的应用场景下，已经达到了超越现有算法的识别或分类性能。

**深度学习的缺点**

深度学习虽然能够自动的学习模式的特征，并可以达到很好的识别精度，但这种算法工作的前提是，使用者能够提供“相当大”量级的数据。也就是说在只能提供有限数据量的应用场景下，深度学习算法便不能够对数据的规律进行无偏差的估计了，因此在识别效果上可能不如一些已有的简单算法。另外，由于深度学习中，图模型的复杂化导致了这个算法的时间复杂度急剧提升，为了保证算法的实时性，需要更高的并行编程技巧以及更好更多的硬件支持。所以，目前也只有一些经济实力比较强大的科研机构或企业，才能够用深度学习算法，来做一些比较前沿而又实用的应用。

# [深度学习( Deep Learning )软件资源列表](http://blog.csdn.net/chlele0105/article/details/17217919)

星号代表对软件库的推荐度，考虑了适用范围、开发语言、更新情况、文档示例完整性等方面，Just personal opinion。

根据编程语言，对深度学习的软件资源分类：

**PYTHON：**

★★★★★ **五星**Theano – CPU/GPU 符号表示编译器in python (from LISA lab at University of Montreal)

相关资源：

[Deep Learning Tutorials](http://deeplearning.net/tutorial) – 使用Theano实现深度学习的示例 (from LISA lab at University of Montreal)

[Pylearn2](https://github.com/lisa-lab/pylearn2) - Pylearn2是一个机器学习库，功能建立在[Theano](http://deeplearning.net/software/theano/)之上.

[Gnumpy](http://www.cs.toronto.edu/~tijmen/gnumpy.html) 是一个Python模块，提供与numpy相似的接口，使用GPU进行计算，运行于cudamat之上.

[cudamat](http://code.google.com/p/cudamat/) 是一个基于GPU的矩阵库，包括训练Neural Networks and Restricted Boltzmann Machines的示例代码。

[3-way factored RBM](http://www.cs.toronto.edu/~ranzato/publications/factored3wayRBM/code/factored3wayBM_04May2010.zip) and[mcRBM](http://www.cs.toronto.edu/~ranzato/publications/mcRBM/code/mcRBM_04May2010.zip) 是Python代码，调用CUDAMat用于为自然图像训练模型(from[Marc’Aurelio Ranzato](http://www.cs.toronto.edu/~ranzato)).

[mPoT](http://www.cs.toronto.edu/~ranzato/publications/mPoT/mPoT.html) 是Python代码，调用CUDAMat和gnumpy用于为自然图像训练模型 (from[Marc’Aurelio Ranzato](http://www.cs.toronto.edu/~ranzato)).

MATLAB：

**★★★★★ 五星**

[DeepLearnToolbox](https://github.com/rasmusbergpalm/DeepLearnToolbox) – A Matlab toolbox for Deep Learning (from Rasmus Berg Palm)

Matlab code for training [conditional RBMs/DBNs](http://www.cs.nyu.edu/~gwtaylor/publications/nips2006mhmublv/code.html) and[factored conditional RBMs](http://www.cs.nyu.edu/~gwtaylor/publications/icml2009/code/) (from[Graham Taylor](http://www.cs.nyu.edu/~gwtaylor/)).

**★★★★ 四星**

[Deep Belief Networks](http://www.cs.toronto.edu/~hinton/MatlabForSciencePaper.html). Matlab代码用于学习深度信念网络(Deep Belief Networks) (from Ruslan Salakhutdinov).

[Estimating Partition Functions of RBM’s](http://www.cs.toronto.edu/~rsalakhu/rbm_ais.html). Matlab代码用于使用退火重要性采样(Annealed Importance Sampling)估计Restricted Boltzmann Machines的剖分函数(the partition function)   (from Ruslan Salakhutdinov).

[Learning Deep Boltzmann Machines](http://web.mit.edu/~rsalakhu/www/DBM.html)Matlab代码用于训练与微调Deep Boltzmann Machines (from Ruslan Salakhutdinov).

**★★★ 三星**

[matrbm](http://code.google.com/p/matrbm/). Ruslan Salakhutdinov’s代码的简化版本, by Andrej Karpathy (Matlab).

**C++：**

**★★★ 三星**

[Cuda-Convnet](http://code.google.com/p/cuda-convnet/) –一个快速的卷积(或更一般地，前向式feed-forward)神经网络的C++/CUDA实现。可用于建模arbitrary layer connectivity and network depth. Any directed acyclic graph of layers will do. Training is done using the back-propagation algorithm.

★★★★★ **五星**

[Eblearn](http://eblearn.sourceforge.net/index.shtml) 是 C++机器学习库，BSD许可证，用于基于能量的学习(energy-based learning),卷积网络(convolutional networks), 视觉/识别应用(vision/recognition applications)等。EBLearn最初由[Pierre Sermanet](http://cs.nyu.edu/~sermanet/) at NYU维护。

**★★★ 三星**

The [CUV Library](http://www.ais.uni-bonn.de/deep_learning/downloads.html) (github[link](https://github.com/deeplearningais/CUV)) 是一个C++库，包括python绑定，易于操作Nvidia CUDA矩阵函数。包括一个RBM实现，退火重要性采样代码( annealed importance sampling)，以及精确计算剖分函数(the partition function)的代码 (from[AIS lab](http://www.ais.uni-bonn.de/) at University of Bonn).

**LUSH：**

**★★★ 三星**

[Eblearn.lsh](http://cs.nyu.edu/~koray/wp/?page_id=29) 是基于 LUSH的机器学习库，用于实现基于能量的学习(Energy-Based Learning). 它包括 “Predictive Sparse Decomposition” 的代码以及其他非监督学习的sparse auto-encoder methods.[Koray Kavukcuoglu](http://cs.nyu.edu/~koray/wp/)在其[主页](http://koray.kavukcuoglu.org/code.html)上提供多篇深度学习相关论文的Eblearn代码。

相关资源：

[LUSH](http://lush.sourceforge.net/) 编程语言及开发环境, 用于@ NYU 开发深度卷积网络。LUSH全称是Lisp Universal Shell，Wiki上有[介绍](http://en.wikipedia.org/wiki/Lush_%28programming_language%29)。

**LUA：**

**★★★★★ 五星**

[Torch](http://www.torch.ch/) – 提供与Matlab相似的环境，用于最新的机器学习算法。(from Ronan Collobert, Clement Farabet and Koray Kavukcuoglu)

 推荐从Theano的Tutorials开始学习，若更关注算法效率，推荐Eblearn和Torch7，尤其是后者，值得一试。

# [Deep Learning源代码收集-持续更新…](http://blog.csdn.net/zouxy09/article/details/11910527)

收集了一些Deep Learning的源代码。主要是Matlab和C++的，当然也有python的。放在这里，后续遇到新的会持续更新。下表没有的也欢迎大家提供，以便大家使用和交流。谢谢。

最近一次更新：2013-9-22

Theano

<http://deeplearning.net/software/theano/>

code from: <http://deeplearning.net/>

Deep Learning Tutorial notes and code

<https://github.com/lisa-lab/DeepLearningTutorials>

code from: lisa-lab

A Matlab toolbox for Deep Learning

<https://github.com/rasmusbergpalm/DeepLearnToolbox>

code from: RasmusBerg Palm

deepmat

Matlab Code for Restricted/Deep BoltzmannMachines and Autoencoder

<https://github.com/kyunghyuncho/deepmat>

code from: KyungHyun Cho <http://users.ics.aalto.fi/kcho/>

Training a deep autoencoder or a classifieron MNIST digits

<http://www.cs.toronto.edu/~hinton/MatlabForSciencePaper.html>

code from: Ruslan Salakhutdinov and GeoffHinton

CNN - Convolutional neural network class

<http://www.mathworks.cn/matlabcentral/fileexchange/24291>

Code from: matlab

Neural Network for Recognition ofHandwritten Digits (CNN)

<http://www.codeproject.com/Articles/16650/Neural-Network-for-Recognition-of-Handwritten-Digi>

cuda-convnet

A fast C++/CUDA implementation ofconvolutional neural networks

<http://code.google.com/p/cuda-convnet/>

matrbm

a small library that can train RestrictedBoltzmann Machines, and also Deep Belief Networks of stacked RBM's.

<http://code.google.com/p/matrbm/>

code from: Andrej Karpathy

Exercise  from UFLDL Tutorial:

<http://deeplearning.stanford.edu/wiki/index.php/UFLDL_Tutorial>

and tornadomeet’s bolg: <http://www.cnblogs.com/tornadomeet/tag/Deep%20Learning/>

and <https://github.com/dkyang/UFLDL-Tutorial-Exercise>

Conditional Restricted Boltzmann Machines

<http://www.cs.nyu.edu/~gwtaylor/publications/nips2006mhmublv/code.html>

from Graham Taylor <http://www.cs.nyu.edu/~gwtaylor/>

Factored Conditional Restricted BoltzmannMachines

<http://www.cs.nyu.edu/~gwtaylor/publications/icml2009/code/index.html>

from Graham Taylor <http://www.cs.nyu.edu/~gwtaylor/>

Marginalized Stacked Denoising Autoencodersfor Domain Adaptation

<http://www1.cse.wustl.edu/~mchen/code/mSDA.tar>

code from: <http://www.cse.wustl.edu/~kilian/code/code.html>

Tiled Convolutional Neural Networks

<http://cs.stanford.edu/~quocle/TCNNweb/pretraining.tar.gz>

<http://cs.stanford.edu/~pangwei/projects.html>

tiny-cnn:

A C++11 implementation of convolutionalneural networks

<https://github.com/nyanp/tiny-cnn>

myCNN

<https://github.com/aurofable/18551_Project/tree/master/server/2009-09-30-14-33-myCNN-0.07>

Adaptive Deconvolutional Network Toolbox

<http://www.matthewzeiler.com/software/DeconvNetToolbox2/DeconvNetToolbox.zip>

<http://www.matthewzeiler.com/>

Deep Learning手写字符识别C++代码

<http://download.csdn.net/detail/lucky_greenegg/5413211>

from: <http://blog.csdn.net/lucky_greenegg/article/details/8949578>

convolutionalRBM.m

A MATLAB / MEX / CUDA-MEX implementation ofConvolutional Restricted Boltzmann Machines.

<https://github.com/qipeng/convolutionalRBM.m>

from: <http://qipeng.me/software/convolutional-rbm.html>

rbm-mnist

C++ 11 implementation of Geoff Hinton'sDeep Learning matlab code

<https://github.com/jdeng/rbm-mnist>

Learning Deep Boltzmann Machines

<http://web.mit.edu/~rsalakhu/www/code_DBM/code_DBM.tar>

<http://web.mit.edu/~rsalakhu/www/DBM.html>

Code provided by Ruslan Salakhutdinov

Efficient sparse coding algorithms

<http://web.eecs.umich.edu/~honglak/softwares/fast_sc.tgz>

<http://web.eecs.umich.edu/~honglak/softwares/nips06-sparsecoding.htm>

Linear Spatial Pyramid Matching UsingSparse Coding for Image Classification

<http://www.ifp.illinois.edu/~jyang29/codes/CVPR09-ScSPM.rar>

<http://www.ifp.illinois.edu/~jyang29/ScSPM.htm>

SPAMS

(SPArse Modeling Software) is anoptimization toolbox for solving various sparse estimation problems.

<http://spams-devel.gforge.inria.fr/>

sparsenet

Sparse coding simulation software

<http://redwood.berkeley.edu/bruno/sparsenet/>

fast dropout training

<https://github.com/sidaw/fastdropout>

<http://nlp.stanford.edu/~sidaw/home/start>

Deep Learning of Invariant Features viaSimulated Fixations in Video

<http://ai.stanford.edu/~wzou/deepslow_release.tar.gz>

<http://ai.stanford.edu/~wzou/>

 Sparse filtering

<http://cs.stanford.edu/~jngiam/papers/NgiamKohChenBhaskarNg2011_Supplementary.pdf>

 k-means

<http://www.stanford.edu/~acoates/papers/kmeans_demo.tgz>

 others:

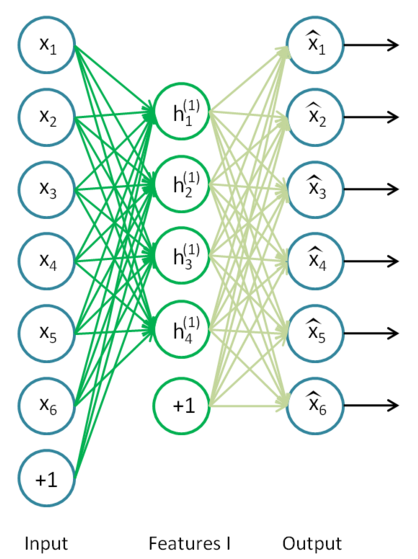
<http://deeplearning.net/software_links/>

# [从自联想神经网络到深度神经网络](http://blog.csdn.net/celerychen2009/article/details/9079715)

深度神经网路已经在语音识别，图像识别等领域取得前所未有的成功。本人在多年之前也曾接触过神经网络。本系列文章主要记录自己对深度神经网络的一些学习心得。

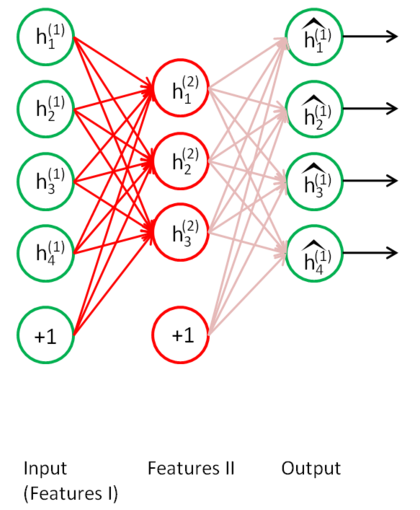
**1.  自联想神经网络与深度网络**

      自联想神经网络是很古老的神经网络模型，简单的说，它就是三层BP网络，只不过它的输出等于输入。很多时候我们并不要求输出精确的等于输入，而是允许一定的误差存在。所以，我们说，输出是对输入的一种重构。其网络结构可以很简单的表示如下：

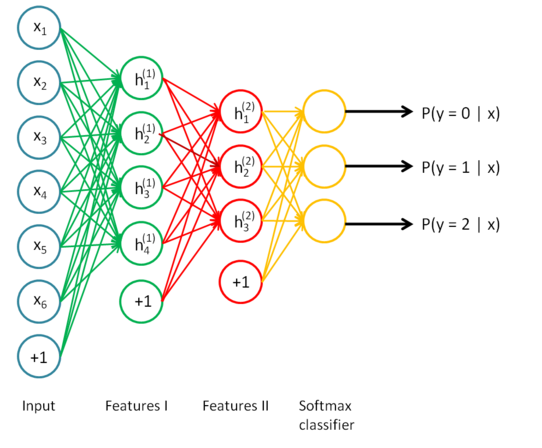


          如果我们在上述网络中不使用sigmoid函数，而使用线性函数，这就是PCA模型。中间网络节点个数就是PCA模型中的主分量个数。不用担心学习算法会收敛到局部最优，因为线性BP网络有唯一的极小值。

      在深度学习的术语中，上述结构被称作自编码神经网络。从历史的角度看，自编码神经网络是几十年前的事情，没有什么新奇的地方。



         既然自联想神经网络能够实现对输入数据的重构，如果这个网络结构已经训练好了，那么其中间层，就可以看过是对原始输入数据的某种特征表示。如果我们把它的第三层去掉，这样就是一个两层的网络。如果，我们把这个学习到特征再用同样的方法创建一个自联想的三层BP网络，如上图所示。换言之，第二次创建的三层自联想网络的输入是上一个网络的中间层的输出。用同样的训练算法，对第二个自联想网络进行学习。那么，第二个自联想网络的中间层是对其输入的某种特征表示。如果我们按照这种方法，依次创建很多这样的由自联想网络组成的网络结构，这就是深度神经网络，如下图所示：



      注意，上图中组成深度网络的最后一层是级联了一个softmax分类器。

      深度神经网络在每一层是对最原始输入数据在不同概念的粒度表示，也就是不同级别的特征描述。

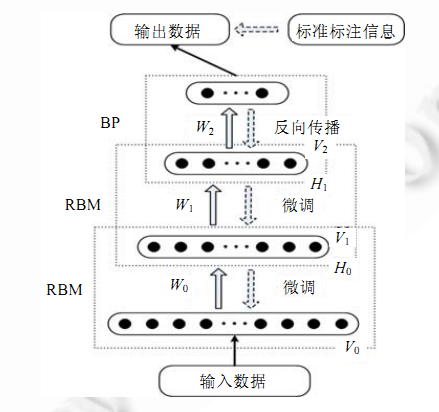
      这种层叠多个自联想网络的方法，最早被Hinton想到了。

       从上面的描述中，可以看出，深度网络是分层训练的，包括最后一层的分类器也是单独训练的，最后一层分类器可以换成任何一种分类器，例如SVM，HMM等。上面的每一层单独训练使用的都是BP算法。 相信这一思路，Hinton早就实验过了。

**2. DBN神经网络模型**

        使用BP算法单独训练每一层的时候，我们发现，必须丢掉网络的第三层，才能级联自联想神经网络。然而，有一种更好的神经网络模型，这就是受限玻尔兹曼机。**使用层叠波尔兹曼机组成深度神经网络的方法，在深度学习里被称作深度信念网络DBN**，这是目前非常流行的方法。下面的术语，将把自联想网络称作自编码网络autoencoder.。通过层叠自编码网络的深度网络在深度学习里另外一个属于叫栈式自编码网络。

        经典的DBN网络结构 是由若干层 RBM 和一层 BP 组成的一种深层神经网络, 结构如下图所示.

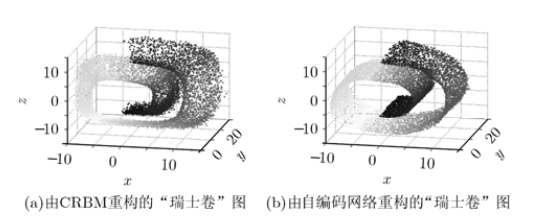
                      
             DBN 在训练模型的过程中主要分为两步:  
             第 1 步：分别单独无监督地训练每一层 RBM 网络,确保特征向量映射到不同特征空间时,都尽可能多地保留特征信息;  
             第 2 步：在 DBN 的最后一层设置 BP 网络,接收 RBM 的输出特征向量作为它的输入特征向量,有监督地训练实体关系分类器.而且每一层 RBM 网络只能确保自身层内的 权值对该层特征向量映射达到最优,并不是对整个 DBN 的特征向量映射达到最优,所以反向传播网络还将错误信息自顶向下传播至每一层 RBM,微调整个 DBN 网络.RBM 网络训练模型的过程可以看作对一个深层 BP 网络权值参数的初始化,使DBN 克服了 BP 网络因随机初始化权值参数而容易陷入局部最优和训练时间长的缺点.

           上述训练模型中第一步在深度学习的术语叫做预训练，第二步叫做微调。最上面有监督学习的那一层，根据具体的应用领域可以换成任何分类器模型，而不必是BP网络。

**3. 深度信念网络的应用**

      由于自编码网络可以对原始数据在不同概念的粒度上进行抽象，深度网络一种自然的应用是对数据进行压缩或者叫降维。

      胡邵华等，他们用一种自编码网络实现了对经典的"瑞士卷"数据的重构:

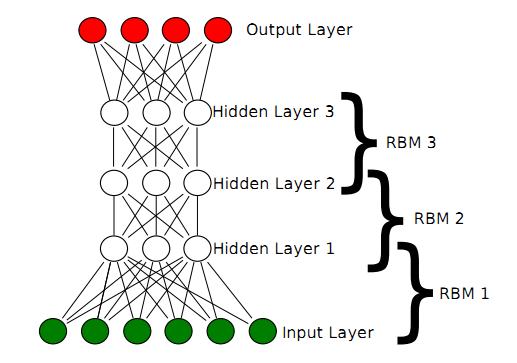


        " 瑞士卷"数据是经典的机器学习中难于分类的数据之一，其隐含的数据内在模式难以在二维数据中描述。然而， 胡邵华等，采用深度信念网络实现了对三维瑞士卷数据的2维表示，其自编码网络节点大小依次为3-100-50-25-10-2. 具体的实现细节请参考文献：胡邵华、宋耀良：基于autoencoder网络的数据降维与重构。

        深度神经网络的另一个常见的应用是特征提取。

        文献：Philippe Hamel and Douglas Eck, LEARNING FEATURES FROM MUSIC AUDIO WITH DEEP BELIEF NETWORKS.

        通过训练一个5层的深度网络提取音乐的特征，用于音乐风格的分类，其分类精度比基于梅尔倒谱系数特征分类的方法提到了14个百分点。



      他们的实现思路非常简单，用上述层叠的多个RBM网络组成深度网络结构来提取音乐的特征。输入的原始数据是经过分帧，加窗之后的信号的频谱。分类器采用的是支撑矢量机SVM。对比的方法则是提取MFCC特征系数，分类器同样采用SVM。更多的细节和实验结果可以参考上面提到的文献。

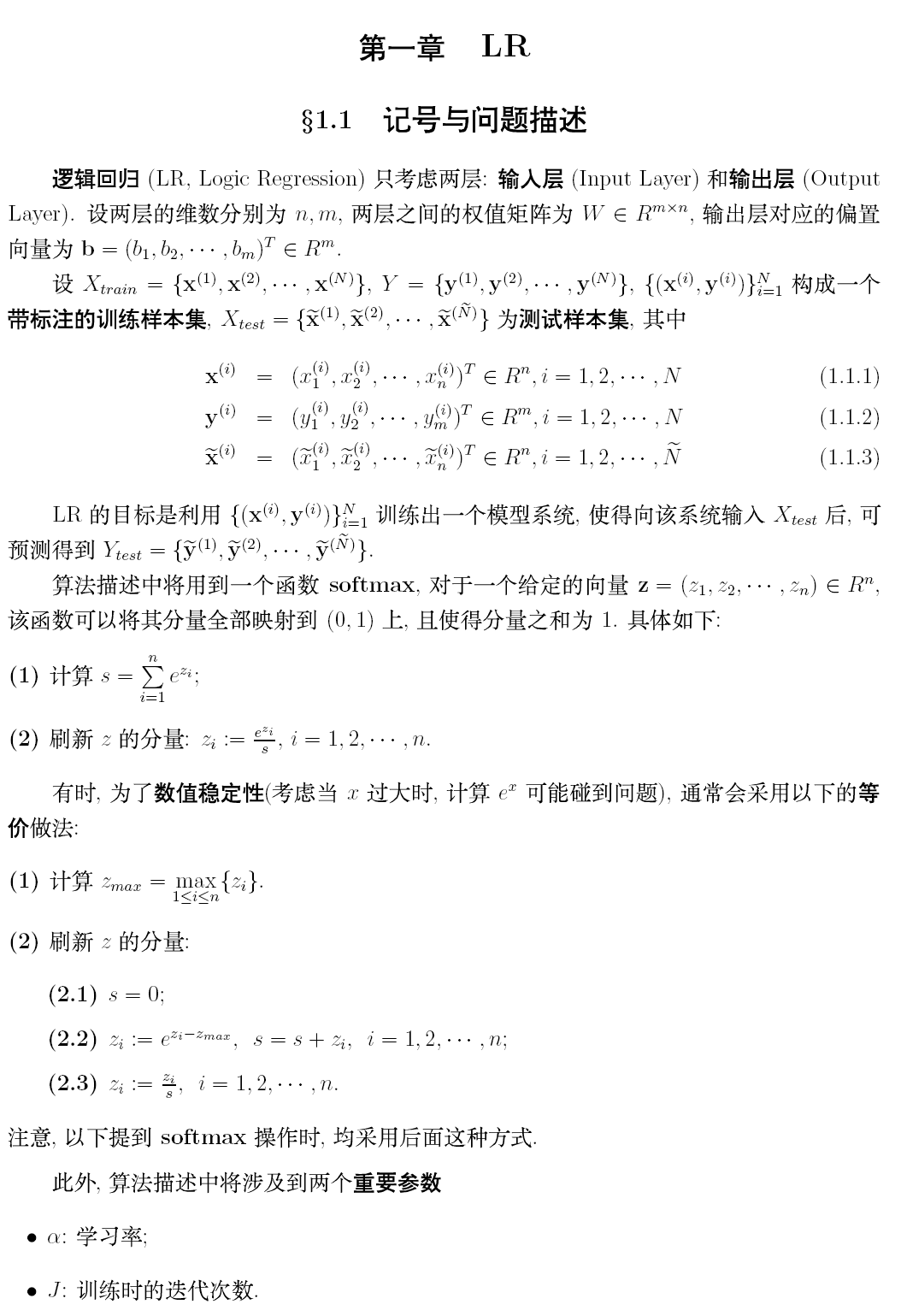
        深度网络是一种良好的无监督学习方法，其特征提取功能能够针对不同概念的粒度大小，能够在很多领域得到广泛的应用。通常，DBN主要用于对一维数据的建模比较有效，例如语音。而通过级联多层卷积网络组成深度网络的模型主要用于二维数据，例如图像等。

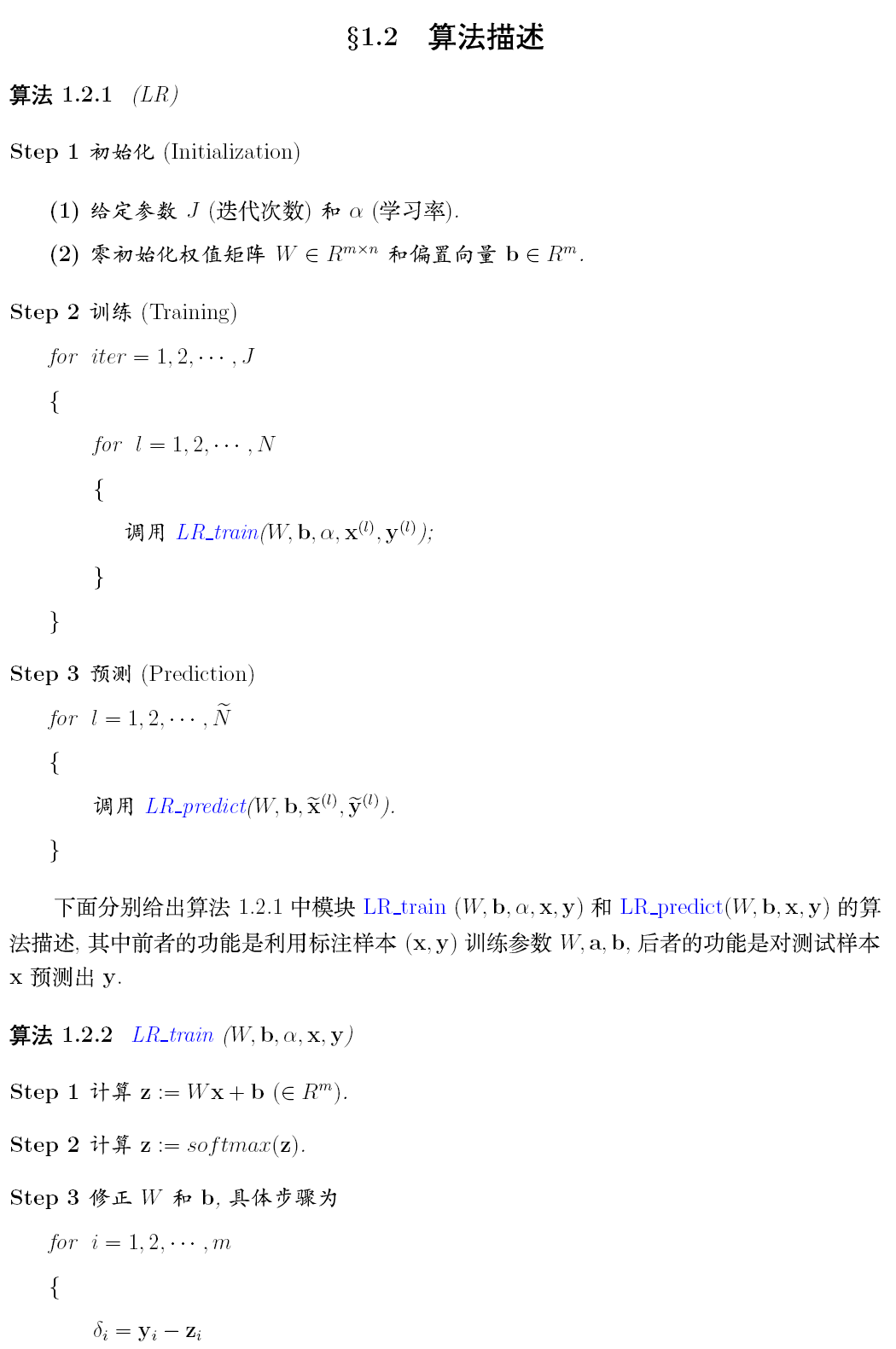
       今后，本人将会更多的关注其具体应用领域的具体算法细节的工程化实现。

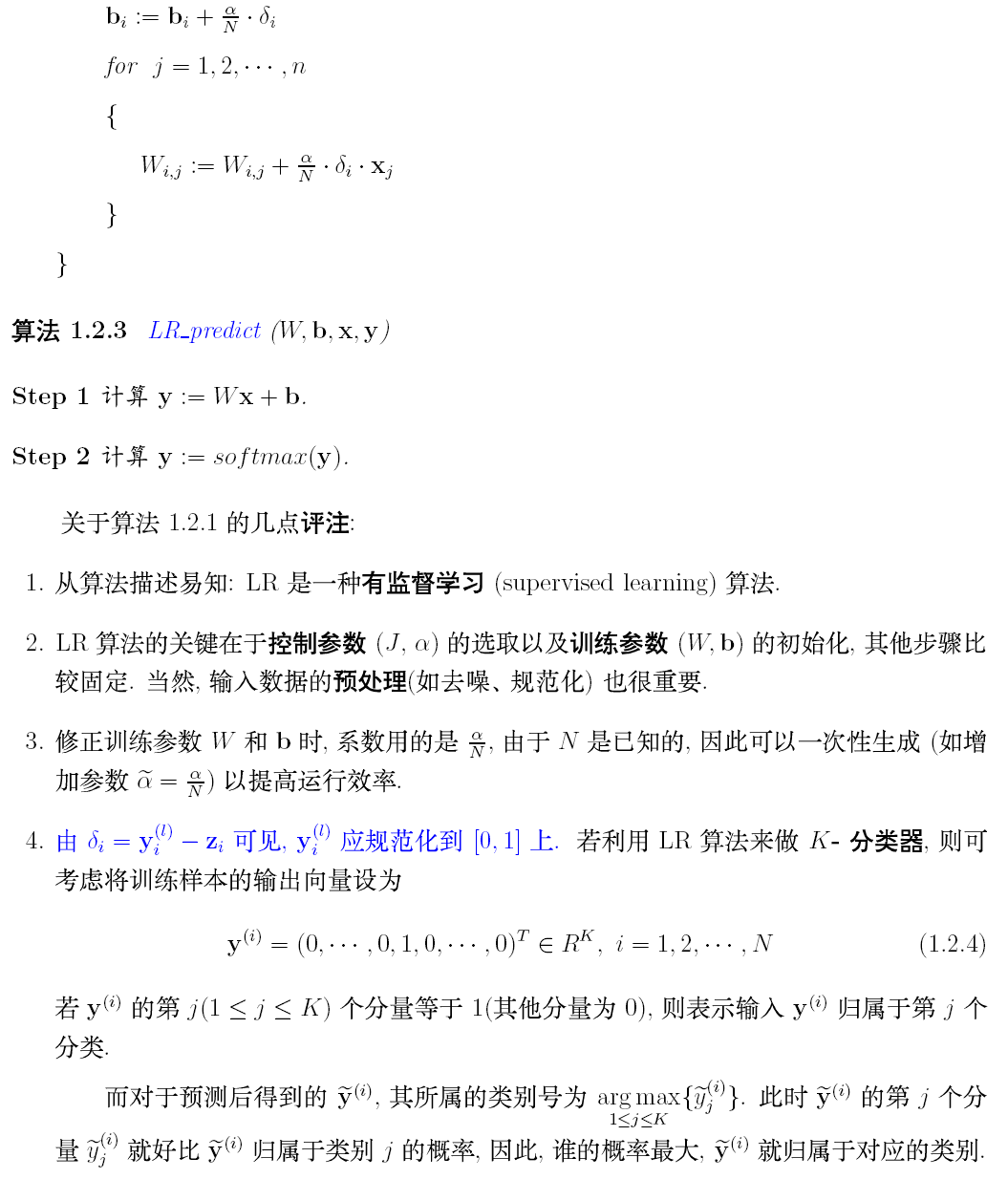
# [Yusuke Sugomori 的 C 语言 Deep Learning 程序解读](http://blog.csdn.net/itplus/article/details/9079973)

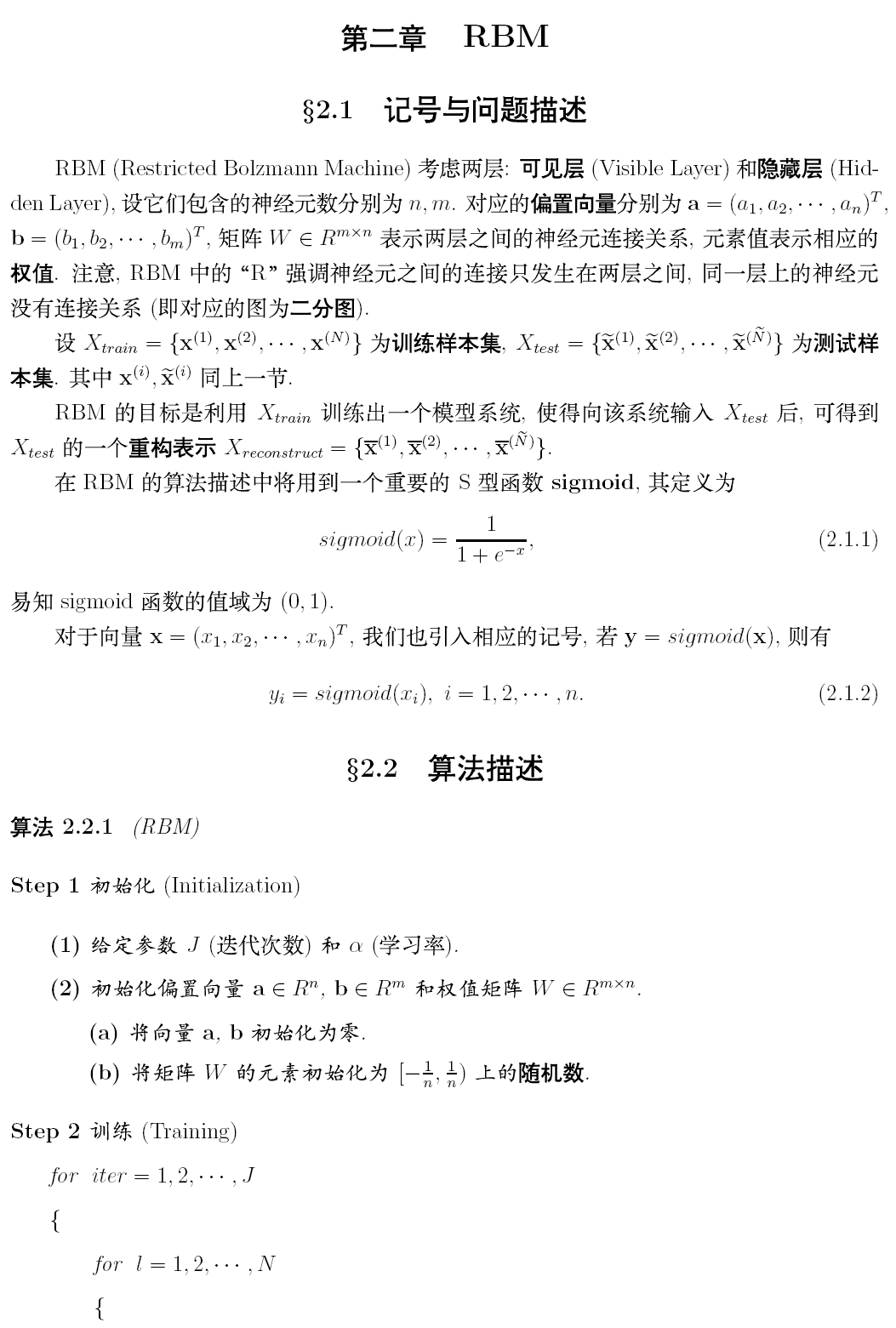
        最近开始在学习 Deep Learning 的基础知识，找了些文献来读，觉得有些晦涩。目前对于 DBN 的算法框架已经有个大概了解，但具体实现细节还不是太清楚，在网上搜到了日本学者 [Yusuke Sugomori](http://yusugomori.com/) 在 Github 上的一个 [deeplearning](https://github.com/yusugomori/DeepLearning) 项目，认真读了其 C 语言的版本，感觉收获蛮大，特将程序翻译成数学描述，希望对正在读或者将要读这个代码的朋友有些帮助。

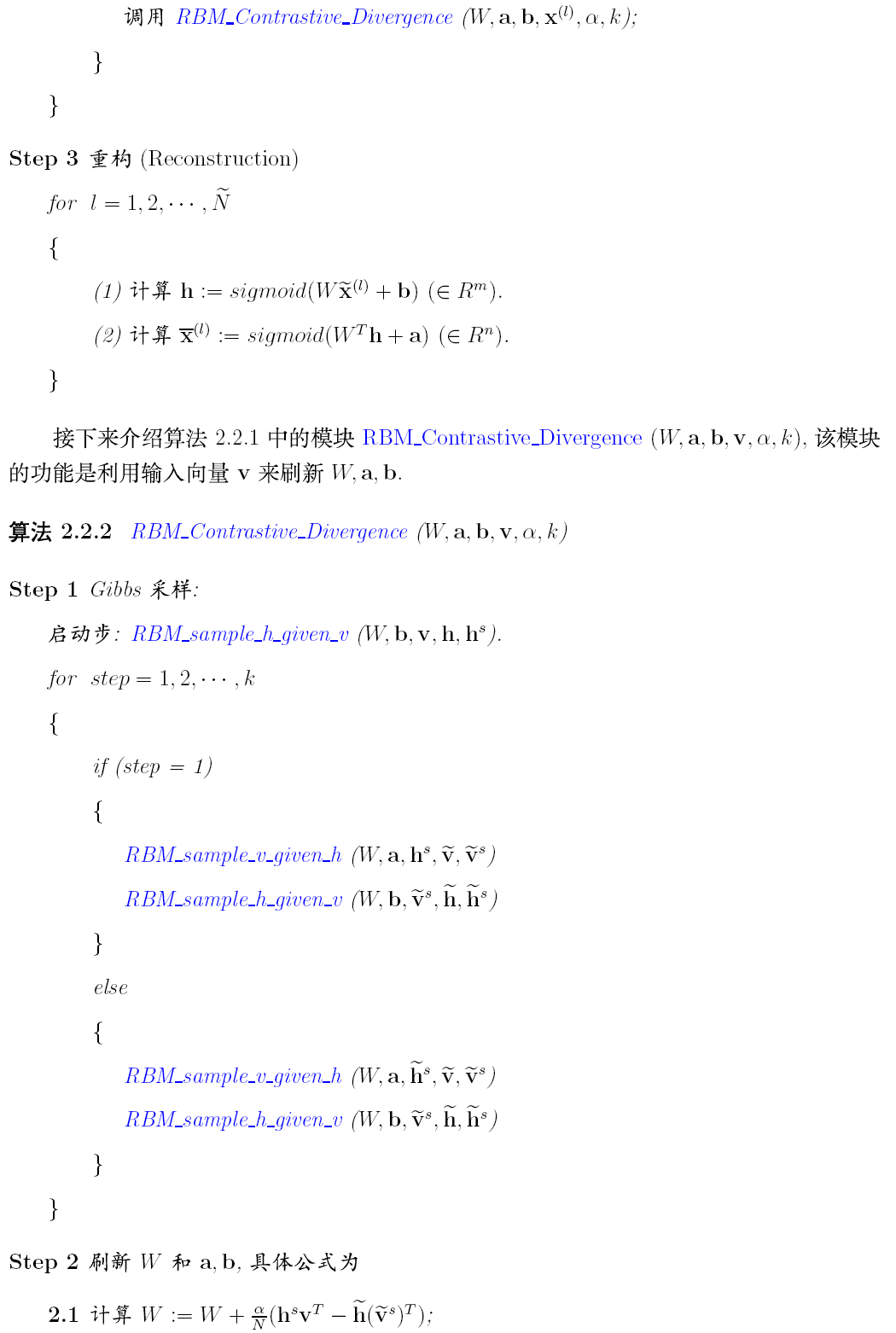
        注意，本文只是客观翻译程序，并作一些相关的评注，个人觉得这个程序与目前流行的 DBN 算法有较大差别，不能直接用来实战。学习 Deep Learning 的朋友可将其作为入门，特别是用来理解 Contrastive Divergence 和 Gibbs Sampling 比较有帮助。

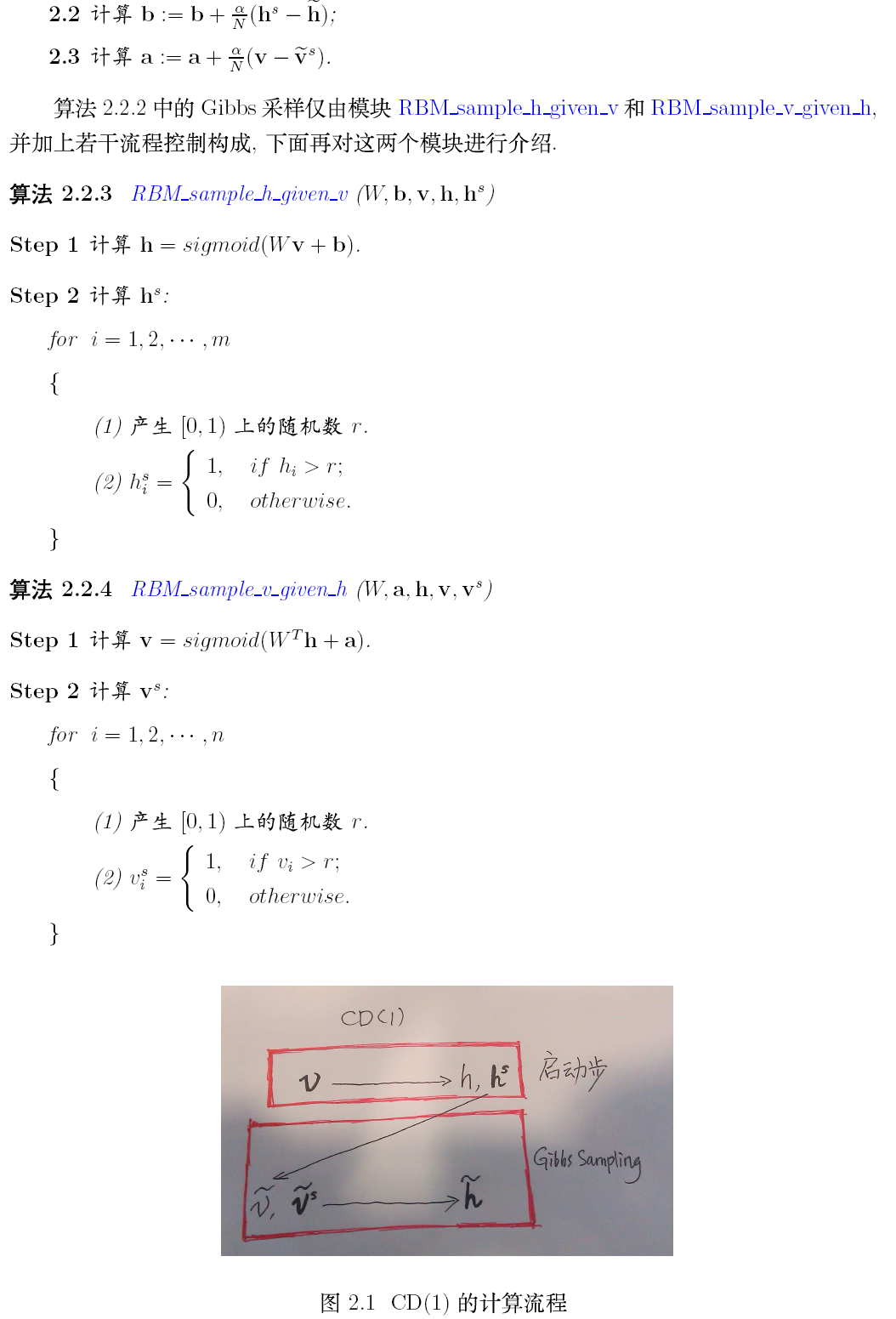


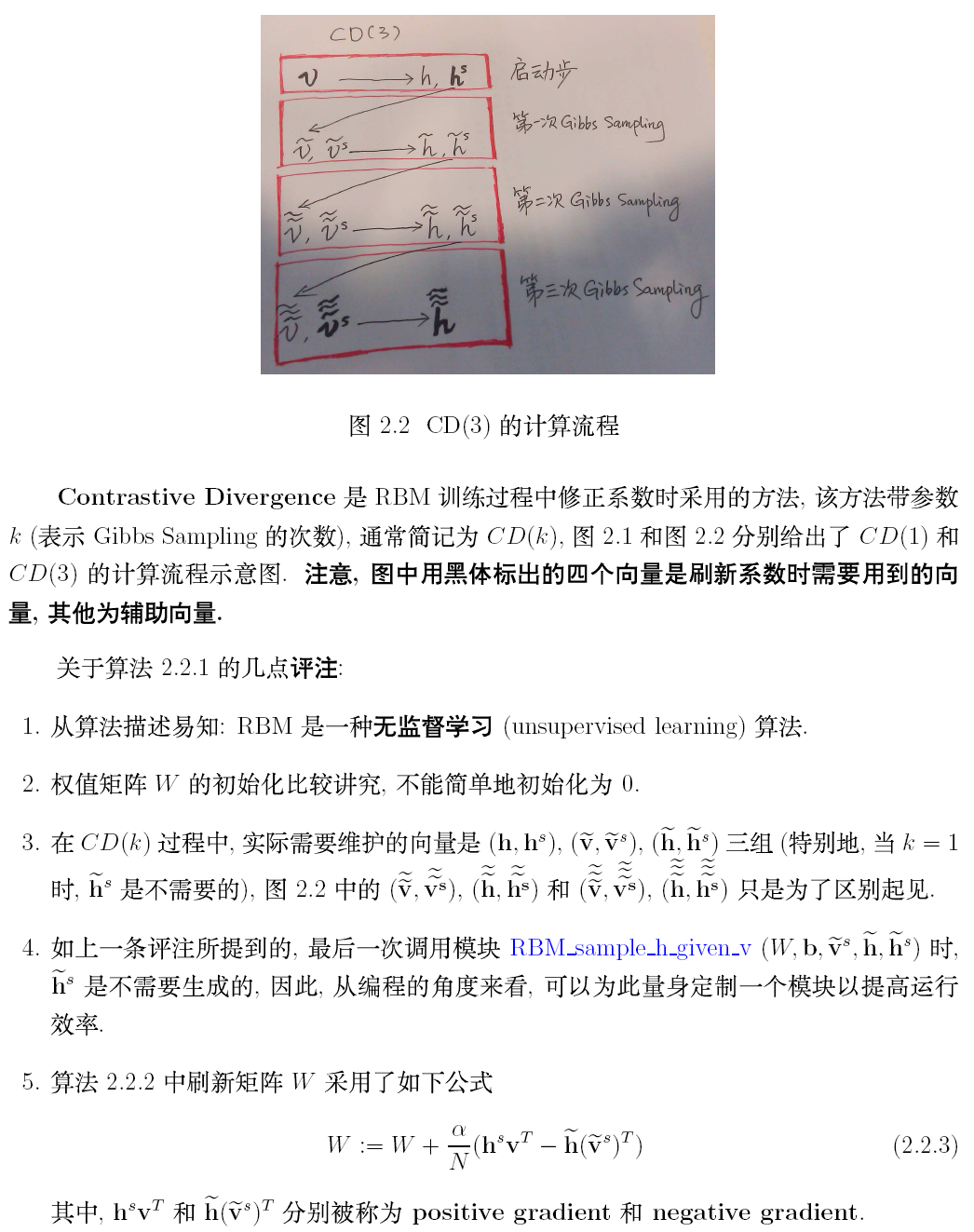


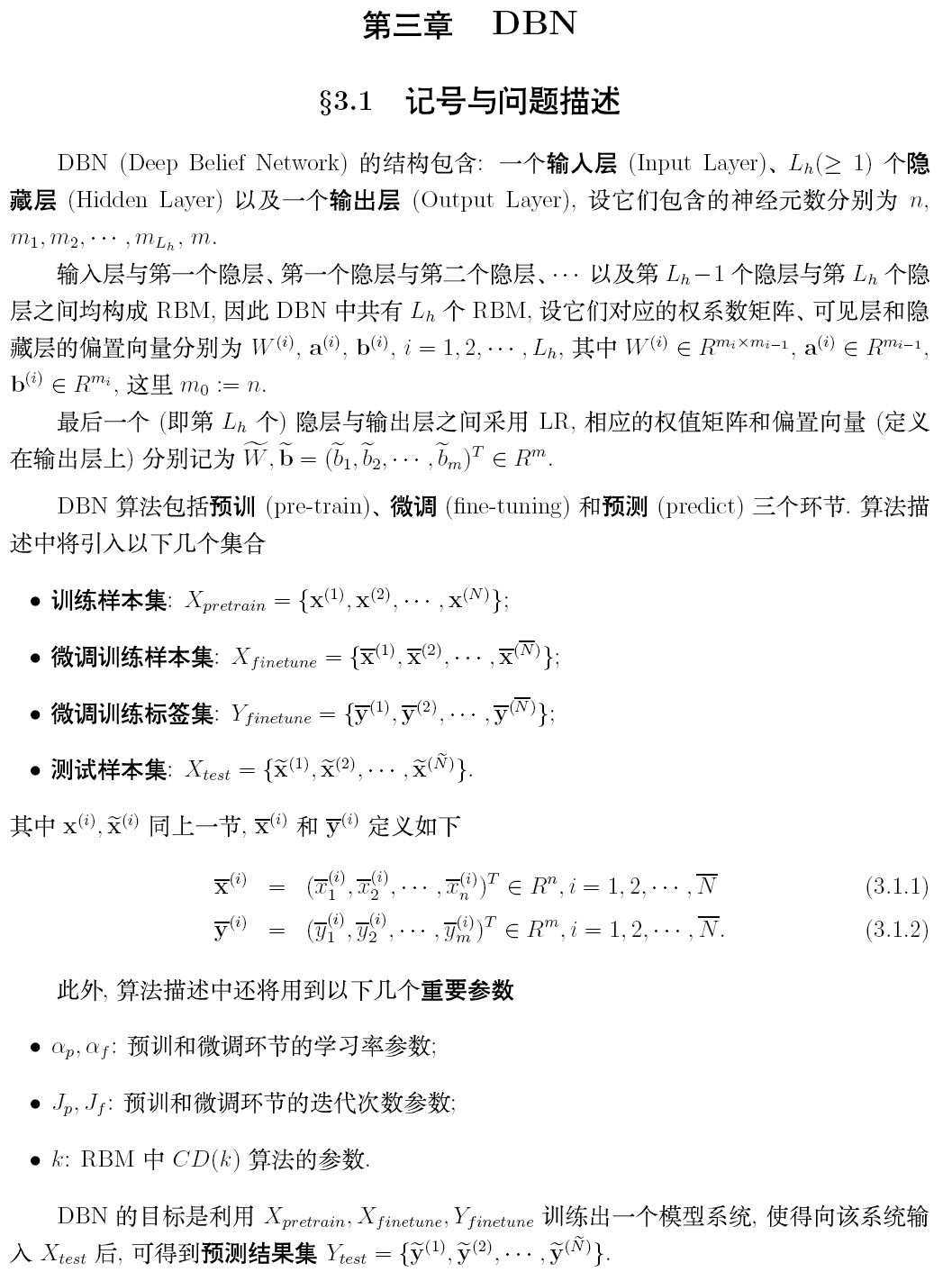


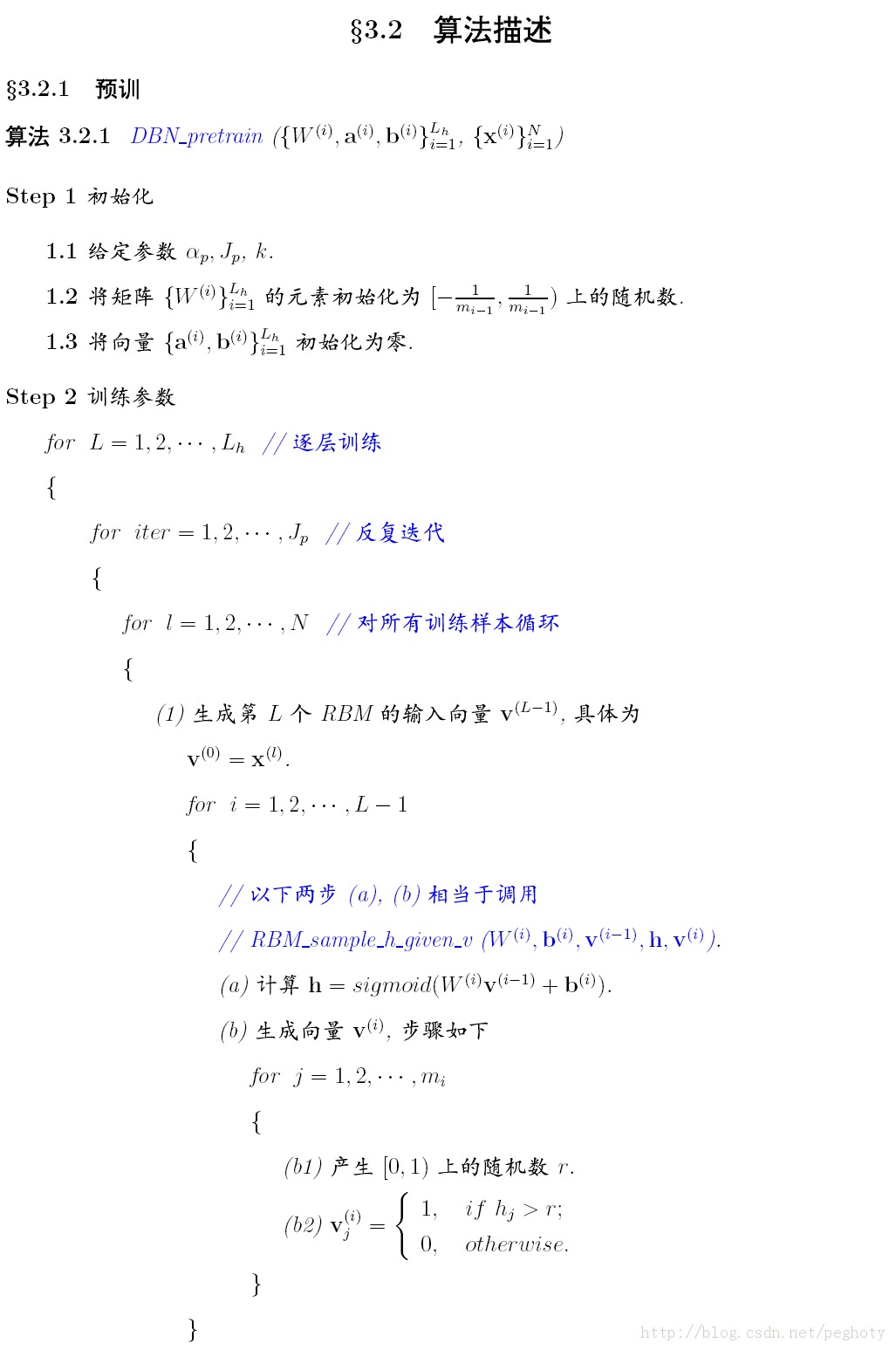


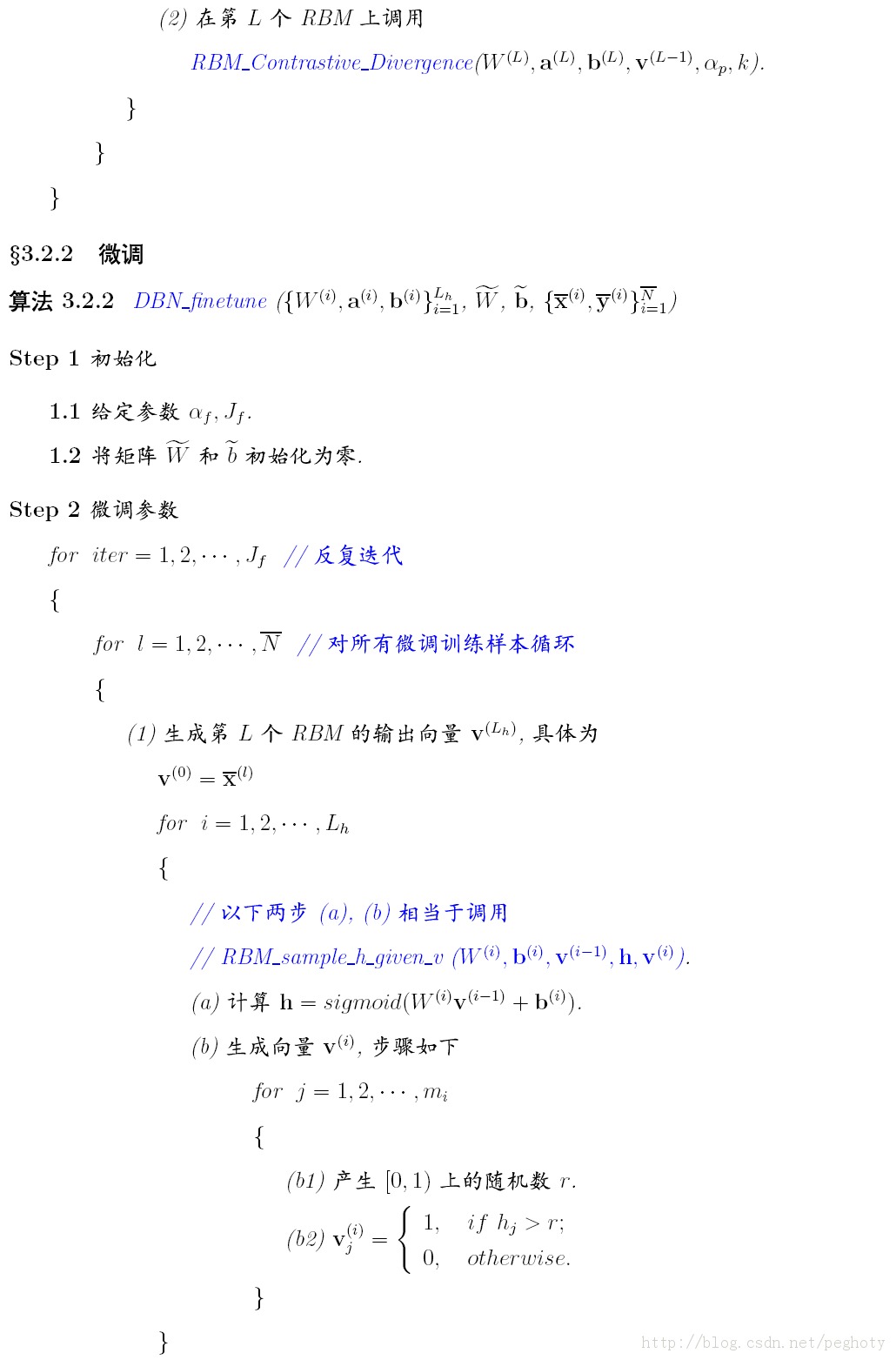


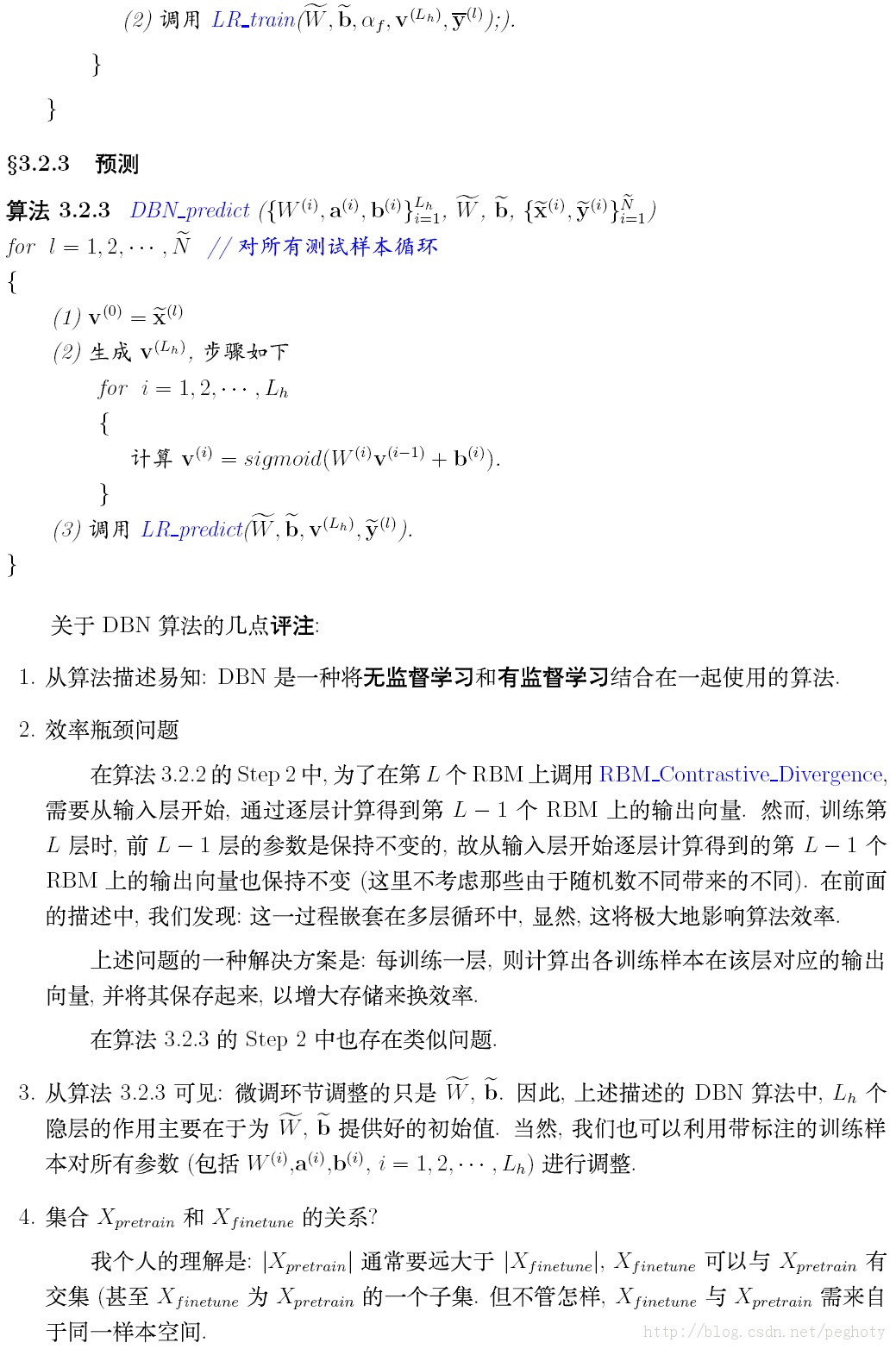












**补充**：

       文中描述的算法在框架上问题不大，但在一些细节处理上可能有些问题，例如，关于 RBM 中的参数刷新公式， fine-tuning 过程中并没有将之前的层联级微调等，望读者知晓。

# [Deep Learning论文笔记之（一）K-means特征学习](http://blog.csdn.net/zouxy09/article/details/9982495)

自己平时看了一些论文，但老感觉看完过后就会慢慢的淡忘，某一天重新拾起来的时候又好像没有看过一样。所以想习惯地把一些感觉有用的论文中的知识点总结整理一下，一方面在整理过程中，自己的理解也会更深，另一方面也方便未来自己的勘察。更好的还可以放到博客上面与大家交流。因为基础有限，所以对论文的一些理解可能不太正确，还望大家不吝指正交流，谢谢。

        本文的论文来自：

[Learning Feature Representations with K-means](http://www.stanford.edu/~acoates/papers/coatesng_nntot2012.pdf), Adam Coates and Andrew Y. Ng. In Neural Networks: Tricks of the Trade, Reloaded, Springer LNCS, 2012

         下面是自己对其中的一些知识点的理解：

**《Learning Feature Representations with K-means》**

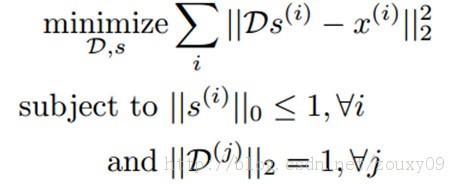
         自从Deep Learning之风盛起之时到现在，江湖上诞生了很多都可以从无标签数据中学习到深度的分级的特征的算法。大部分情况，这些算法都涉及到一个多层网络，而训练和调整这个网络需要很多tricks。最近，我们发现K-means聚类算法也可以被作为一个非常快的训练方法。它的优点是快！容易实现！当然了，K-means也不是万能神丹，它也存在自身的局限性。在本文中，我们就关注K-means的方方面面。总结了最近的K-means算法的效果和介绍使用k-means来有效地学习图像的特征的一些技巧。

**一、概述**

         非监督学习的一般流程是：先从一组无标签数据中学习特征，然后用学习到的特征提取函数去提取有标签数据特征，然后再进行分类器的训练和分类。之前说到，一般的非监督学习算法都存在很多hyper-parameters需要调整。而，最近我们发现对于上面同样的非监督学习流程中，用K-means聚类算法来实现特征学习，也可以达到非常好的效果，有时候还能达到state-of-the-art的效果。亮瞎了凡人之俗眼。

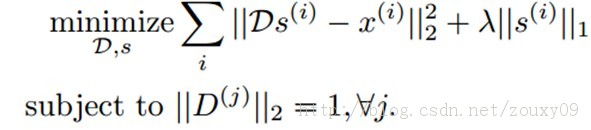
         托“bag of features ”的福，K-means其实在特征学习领域也已经略有名气。今天我们就不要花时间迷失在其往日的光芒中了。在这里，我们只关注，如果要K-means算法在一个特征学习系统中发挥良好的性能需要考虑哪些因素。这里的特征学习系统和其他的Deep Learning算法一样：直接从原始的输入（像素灰度值）中学习并构建多层的分级的特征。另外，我们还分析了K-means算法与江湖中其他知名的特征学习算法的千丝万缕的联系（天下武功出少林，哈哈）。

         经典的K-means聚类算法通过最小化数据点和最近邻中心的距离来寻找各个类中心。江湖中还有个别名，叫“矢量量化vector quantization”（[这个在我的博客上也有提到](http://blog.csdn.net/zouxy09/article/details/9153255)）。我们可以把K-means当成是在构建一个字典D∊Rnxk，通过最小化重构误差，一个数据样本x(i)∊Rn可以通过这个字典映射为一个k维的码矢量。所以K-means实际上就是寻找D的一个过程：



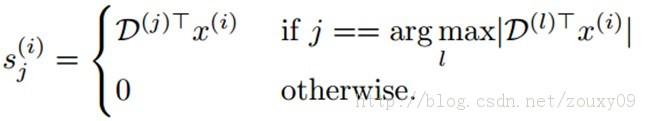
       这里，s(i)就是一个与输入x(i)对应的码矢量。D(j)是字典D的第j列。K-means毕生的目标就是寻找满足上面这些条件的一个字典D和每个样本x(i)对应的码矢量s(i)。我们一起来分析下这些条件。首先，给定字典D和码矢量s(i)，我们需要能很好的重构原始的输入x(i)。数学的表达是最小化x(i)和它的重构D s(i)。这个目标函数的优化需要满足两个约束。首先，|| s(i)||0<=1，意味着每个码矢量s(i)被约束为最多只有一个非零元素。所以我们寻找一个x(i)对应的新的表达，这个新的表达不仅需要更好的保留x(i)的信息，还需要尽可能的简单。第二个约束要求字典的每列都是单位长度，防止字典中的元素或者特征变得任意大或者任意小。否则，我们就可以随意的放缩D(j)和对应的码矢量，这样一点用都木有。

         这个算法从精神层面与其他学习有效编码的算法很相似，例如sparse coding：



         Sparse coding也是优化同样类型的重构。但对于编码复杂度的约束是通过在代价函数中增加一个惩罚项λ|| s(i)||1，以限制s(i)是稀疏的。这个约束和K-means的差不多，但它允许多于一个非零值。在保证s(i)简单的基础上，可以更准确的描述x(i)。

         虽然Sparse coding比K-means性能要好，但是Sparse coding需要对每个s(i)重复的求解一个凸优化问题，当拓展到大规模数据的时候，这个优化问题是非常昂贵的。但对于K-means来说，对s(i)的优化求解就简单很多了：



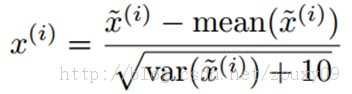
         这个求解是很快的，而且给定s求解D也很容易，所以我们可以通过交互的优化D和s（[可以了解下我博客上的EM算法的博文](http://blog.csdn.net/zouxy09/article/details/8537620)）来快速的训练一个非常大的字典。另外，K-means还有一个让人青睐的地方是，它只有一个参数需要调整，也就是要聚类的中心的个数k。

**二、数据、预处理和初始化**

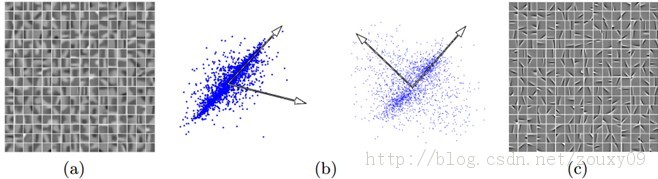
         这里我们采用的是包含很多小的图像patches的数据集，每个patch是16x16的灰度图，对每个patch样本我们将其拉成一个256维的列向量。这些patches可以在无标签图像中随机的裁剪得到。为了建立一个“完备complete”的字典（至少有256个类中心的字典），我们需要保证有足够的用以训练的patches，这样每个聚类才会包含一定合理数量的输入样本。对于16x16的灰度patch来说，样本数m=100,000个是足够的。实际上，训练一个k-means字典比其他的算法（例如sparse coding）需要的训练样本个数要多，因为在k-means中，一个样本数据只会贡献于一个聚类的中心，换句话说一个样本只能属于一个类，与其他类就毫无瓜葛了，该样本的信息全部倾注给了它的归宿（对应类的中心）。

**2.1、预处理 Pre-processing**

         在训练之前，我们需要对所有的训练样本patches先进行亮度和对比度的归一化。具体做法是，对每个样本，我们减去灰度的均值和除以标准差。另外，在除以标准差的时候，为了避免分母为0和压制噪声，我们给标准差增加一个小的常数。对于[0, 255]范围的灰度图，给方差加10一般是ok的：



         对训练数据进行归一化后，我们就可以在上面运行k-means算法以获得相应的聚类中心了（字典D的每一列），可视化在图a中，可以看到，k-means趋向于学习到低频的类边缘的聚类中心。但不幸的是，这样的特征不会有比较好的识别效果，因为邻域像素的相关性（图像的低频变化）会很强。这样K-means就会产生很多高度相关的聚类中心。而不是分散开聚类中心以更均匀的展开训练数据。所以我们需要先用白化来去除数据的相关性，以驱使K-means在正交方向上分配更多的聚类中心。



       图a是由k-means从没有经过白化处理的自然图像中学习到的聚类中心。图b展示了没有和有白化的效果。左边是没有经过白化的，因为数据存在相关性，所以聚类中心会跑偏。右边是经过白化后的，可以看到聚类中心是更加正交的，这样学习到的特征才会不一样。图c展示的是从经过白化后的图像patches学习到的聚类中心。

         实现whitening白化一个比较简单的方法是ZCA白化（[可以参考UFLDL](http://deeplearning.stanford.edu/wiki/index.php/%E7%99%BD%E5%8C%96)）。我们先对数据点x的协方差矩阵进行特征值分解cov(x)=VDVT。然后白化后的点可以表示为：

http://img.blog.csdn.net/20130815141048703?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvem91eHkwOQ==/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/SouthEast

        ɛzca是一个很小的常数。对于对比度归一化后的数据，对16x16的patch，可以设ɛzca=0.01，对8x8的patch，可以设ɛzca=0.1 。需要注意的一点是，这个常数不能太小，如果太小会加强数据的高频噪声，会使特征学习更加困难。另外，因为数据的旋转对K-means没有影响，所以可以使用其他的白化变换方法，例如PCA白化（与ZCA不同只在于其旋转了一个角度）。

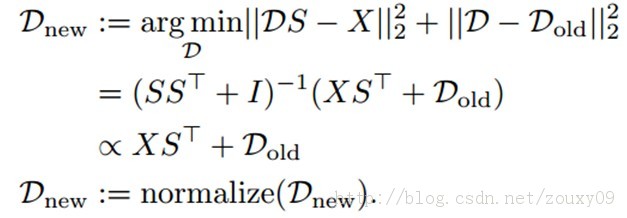
         在白化后的数据中运行k-means可以得到清晰的边缘特征。这些特征和sparse coding啊，ICA啊等等方法学到的初级特征差不多。如图c所示。另外，对于新的数据，可能需要调整归一化的参数ɛ和白化的参数ɛzca，以满足好的效果（例如，图像patches具有高对比度，低噪声和少低频波动）。

**2.2、初始化 Initialization**

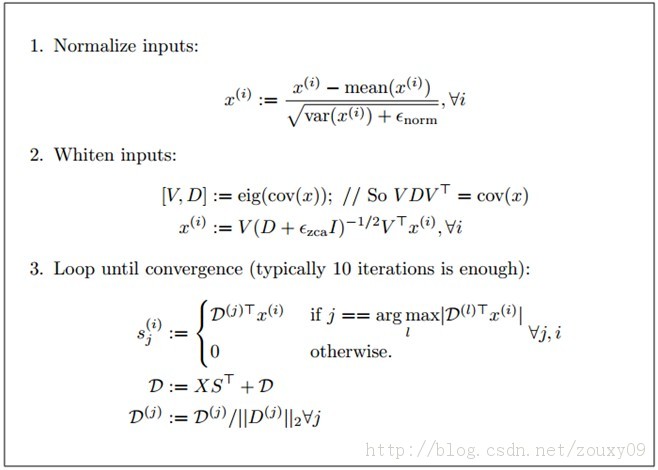
         一般的K-means聚类算法存在一个比较常见的问题，就是会出现空的聚类。通俗的讲，就是它聚类后的一个类里面居然没有样本（接近于没有）。那么很明显，这个类就一点意义都没有，留着反而还会危害人间。这我们当然得做些努力来避免这种情况的发生了。那就得找原因了吧。其实这一定情况下可以认为是中心的初始化的不恰当导致的。常用的中心初始化方法就是随机地从样本中挑k个出来作为k个初始的聚类中心。但这不是个明智的选择。它有可能会导致图像趋于稠密聚集某些区域，因为如果随机选择的patches，也就是训练样本本身就在某个区域分布非常密，那么我们随机去选择聚类中心的时候，就会出现就在这个数据分布密集的地方被选出了很多的聚类中心（因为原数据分布密集，所以被选中的概率会大）。这是不好的，因为本身这个密集的区域，一个聚类中心才是好的，你硬搞出几个聚类中心，好好的队伍就会被强拆了，还搞得其他比较分散的数据点找不到归宿，因为聚类中心离它好遥远，大部分的聚类中心都让富二代给占有了，社会资源分配不当啊，哈哈。这样还会出现一种情况，就是如果样本分布密集的话，它们都几乎从属于一个聚类中心，那么其他聚类中心就几乎没有样本属于它。这样也不好。所以，一个比较好的初始化方式就是从一个正态分布中随机初始化聚类中心，然后归一化他们到单位长度。另外，因为经过了白化的阶段，我们期望我们的数据的主要成分已经在一定程度上被组织为球面spherical分布了，而在一个球面上随机地选取一些向量结果就显得温柔多了。

         另外，还存在一些启发性的方法去改善k-means的这个问题。例如启发性得重新初始化空的聚类中心。实践中证明，这种方法对图像数据的效果比较好。当空类出现的时候，我们再随机挑选样本来替代这个空类中心，一般都是足够有效的。但好像一般也没啥必要。事实上，对于一个足够规模的实现来说，我们会训练很多的聚类中心，如果一个聚类中心包含的数据点太少了，我们直接抛弃它就好了，不要这个聚类中心了，因为俺们的聚类中心够多，多你一个不多，少你一个不少的。

         还有一种改善的方法叫做聚类中心的抑制更新damped updates。它在每次的迭代中，都会根据以下方式更新聚类中心：



         需要注意的是，这种形式的抑制并不会对“大”的聚类有很大的影响（因为XST的第j列会比Dold的第j列要大得多），它只能防止小的聚类在一次的迭代过程中被拉得太远（就像滤波的方法一样，新的值不仅受当前值的影响，还受历史值的影响，这样就会防止这个值出现突变。例如a\_old=1，新的值为5，那么a\_new=0.5\*5+0.5\*a\_old=3，也就是a\_new会跳到3而不是一下子跳到5，当然还取决于前面的权值0.5，看我们更信任新值还是历史值。但如果新值很大，那就没有特别大的效果了。例如新值为99，那么a\_new=50，它还是比a\_old=1要大多了）。我们将这个k-means训练算法概括如下：



**三、与稀疏特征学习的比较**

         上面我们提到，K-means能像ICA，sparse coding一样学习到方向性的边缘特征。那我们会怀疑，这是偶然吗（因为边缘太常见了，所以一般的学习模型都可以发现它们）？！还是说这隐含着K-means本身就具有和ICA相似的稀疏分解的能力。因为每个s(i)只能包含一个非零元素（相应的聚类中心），所以k-means会尝试去学习一个可以很好的描述整个输入图像的聚类中心。因此，它也无法保证一定可以学到和ICA或者sparse coding学到的滤波器一样。这些算法可以学习到真正的“分布式”描述，因为一张图像可以通过一个字典的几列来联合描述，而不是只通过一列。尽管如此，k-means在天时地利人和的时候依然能发现数据的稀疏投影，虽然作为一个聚类方法它好像没有明确的通过公式告诉我们它还能做这个。所以我们看到的结果也不是幻觉或者偶然。

         虽然k-means和ICA和sparse coding有着经验性的相似，但它有一个很大的缺点：它发现数据稀疏投影方向的能力很大程度上取决于输入数据的维数和数量。如果数据的维数增加，我们就需要增加大量的样本，才能得到好的结果。这点就比较难伺候了。例如，在实验中，对64x64的patch，我们用了500,000个样本来训练k-means，结果还是很烂。我们只学习到了很少的边缘。在这种情况下，就会存在很多的接近空数据点的聚类，所以就需要非常多的数据才能让k-means发挥光与热。所以，这就存在一个折衷了，是否要采用k-means，就要看我们的数据有多少维，我们能提供多少数据（一般要达到相同的结果，比sparse coding要多很多）。对于适当的维度（例如100维），使用k-means的训练速度比其他算法快那么多，那么我情愿花点力气去提供更多的样本给它。但对于很高维的数据，采用其他算法例如sparse coding会比k-means性能要好，甚至速度也要更快。

**4、在图像识别上的应用**

         在这里没什么好说的，k-means就作为一种学习特征的方法，取代一般的非监督学习过程里面的特征学习那块就好了。可以参考UFLDL中的“[卷积特征提取](http://deeplearning.stanford.edu/wiki/index.php/%E5%8D%B7%E7%A7%AF%E7%89%B9%E5%BE%81%E6%8F%90%E5%8F%96)”和“[池化](http://deeplearning.stanford.edu/wiki/index.php/%E6%B1%A0%E5%8C%96)”。另外，文中也介绍了一些选择参数的方法，很多论文也可以看到，这里也不说了。这里比较有趣的一点是，文中说到，根据观测的结果，如果我们的识别系统设计有大量有效的有标签训练数据的话，那么选择一个简单的快速的前向编码器（特征映射函数）的话，效果都可以挺好。但如果我们的有标签训练数据不多，那么采用一个复杂的编码函数会更好。例如，在大规模情况下，采用软阈值非线性函数:

http://img.blog.csdn.net/20130815141254718?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvem91eHkwOQ==/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/SouthEast

就可以工作的比较好。其中，ɑ是个可调的常数。相反，复杂的sigmoid非线性函数效果就没那么好了：

http://img.blog.csdn.net/20130815141304359?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvem91eHkwOQ==/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/SouthEast

**5、构建深度网络**

         在上面一个章节中，我们是先从一些无标签数据中随机提取一些patches，然后用k-means来学习其中的特征，得到一个字典。然后对于一个大的图像，我们用这个字典（特征提取器）去卷积图像的每个局部感受野，从而提取整幅图像的特征。然后我们再通过一个pooling步骤去对提取到的特征进行绛维。当然了，如果我们可以通过上面这些提取到的特征去学习更高层的特征，那就更好了。最简单的方法就是，对无标签数据集X中，我们通过上面的方法得到pooling后的特征后，然后得到新的数据集Z，我们再在这个数据库Z中用同样的特征学习过程去学习新的特征。从而得到第二层的特征。但这有个问题，就是这个Z一般都是非常高维的，所以需要用非常大的特征字典来学习（例如k=10000甚至更多）。但这样，对k-means的要求也高了，我们需要提供更多的样本来训练。

         该文提出了一种pair-wise “dependency test"的方法来改善这个问题。我们通过一个“能量相关”来定义数据集Z中两个特征zj和zk的依赖性。我们需要在Z的特征表达之上学习更高级的特征。使用“dependency test"，我们可以以一种相对简单的方式来选择合理的感受野：我们挑了一个特征z0，然后使用“dependency test"来寻找和z0具有很强依赖性的R特征。然后只用这R特征作为k-means算法的输入。如果我们选取的R足够小（例如100或者200），那么归一化和白化过后，再用k-means来训练一般都可以达到好的效果。因为输入的维数很小，所以我们只需要训练一百个左右的聚类中心，这样使用少量的训练样本就可以运行k-means了。

         详细的算法介绍和效果见原论文。

**六、总结**

         本文中，我们讨论了用k-means来搭建特征学习系统的一些关键点，现在总结如下：

1、需要先对数据进行均值和对比度的归一化。

2、使用白化来消去数据的相关性。注意白化参数的选择。

3、通过高斯噪声和归一化来初始化k-means的聚类中心。

4、使用抑制更新来避免空聚类和增加稳定性。

5、注意数据的维数和稀疏的影响。K-means通过寻找数据分布的”heavy-tailed"方向来找到输入数据的稀疏投影。但如果数据没有经过适当的白化，或者数据维数太高了，或者存在无效的数据，那么效果往往是不好的。

6、对于高维数据，k-means需要增加大量的样本来能得到好的效果。

7、一些外在的参数（例如pooling，编码方法的选择等等）对于系统性能的影响比学习算法本身要大。所以最好先化时间和成本在通过更多的交叉检验的方法来挑选好的参数比花更多的成本在学习算法上要明智的多。

8、对于图像识别来说，更多的聚类中心往往是有帮助的。当然，前提是我们要有足够的训练样本。

9、当有标签数据多的时候，采用一个简单的编码器。如果标签数据有限（例如每类只有百来个训练样本），那么采用一个复杂的编码器会更好。

10、尽可能地使用局部感受野。K-means是否成功，其瓶颈在于输入数据的维数。越低越好。如果无法手动的选择好的局部感受野，那么尝试去通过一个自动的依赖性测试dependency test来帮助从数据中挑选出一组低维的数据。这对于深度网络来说是很有必要的。

# [Deep Learning论文笔记之（二）Sparse Filtering稀疏滤波](http://blog.csdn.net/zouxy09/article/details/9982859)

自己平时看了一些论文，但老感觉看完过后就会慢慢的淡忘，某一天重新拾起来的时候又好像没有看过一样。所以想习惯地把一些感觉有用的论文中的知识点总结整理一下，一方面在整理过程中，自己的理解也会更深，另一方面也方便未来自己的勘察。更好的还可以放到博客上面与大家交流。因为基础有限，所以对论文的一些理解可能不太正确，还望大家不吝指正交流，谢谢。

       本文的论文来自：

[Sparse filtering](http://cs.stanford.edu/~jngiam/papers/NgiamKohChenBhaskarNg2011.pdf), J. Ngiam, P. Koh, Z. Chen, S. Bhaskar, A.Y. Ng. NIPS2011。在其论文的[支撑材料](http://cs.stanford.edu/~jngiam/papers/NgiamKohChenBhaskarNg2011_Supplementary.pdf)中有相应的Matlab代码，代码很简介。不过我还没读。

         下面是自己对其中的一些知识点的理解：

**《Sparse Filtering》**

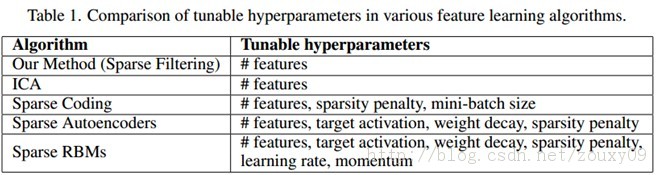
       本文还是聚焦在非监督学习Unsupervised feature learning算法。因为一般的非监督算法需要调整很多额外的参数hyperparameter。本文提出一个简单的算法：sparse filtering。它只有一个hyperparameter（需要学习的特征数目）需要调整。但它很有效。与其他的特征学习方法不同，sparse filtering并没有明确的构建输入数据的分布的模型。它只优化一个简单的代价函数（L2范数稀疏约束的特征），优化过程可以通过几行简单的Matlab代码就可以实现。而且，sparse filtering可以轻松有效的处理高维的输入，并能拓展为多层堆叠。

         sparse filtering方法的核心思想就是避免对数据分布的显式建模，而是优化特征分布的稀疏性从而得到好的特征表达。

**一、非监督特征学习**

         一般来说，大部分的特征学习方法都是试图去建模给定训练数据的真实分布。换句话说，特征学习就是学习一个模型，这个模型描述的就是数据真实分布的一种近似。这些方法包括denoising autoencoders，restricted Boltzmann machines (RBMs)， independent component analysis (ICA) 和sparse coding等等。

         这些方法效果都不错，但烦人的一点就是，他们都需要调节很多参数。比如说学习速率learning rates、动量momentum（好像rbm中需要用到）、稀疏度惩罚系数sparsity penalties和权值衰减系数weight decay等。而这些参数最终的确定需要通过交叉验证获得，本身这样的结构训练起来所用时间就长，这么多参数要用交叉验证来获取时间就更多了。我们花了大力气去调节得到一组好的参数，但是换一个任务，我们又得调节换另一组好的参数，这样就会花了俺们太多的时间了。虽然ICA只需要调节一个参数，但它对于高维输入或者很大的特征集来说，拓展能力较弱。



         本文中，我们的目标是研究一种简单并且有效的特征学习算法，它只需要最少的参数调节。虽然学习数据分布的模型是可取的，而且效果也不错，但是它往往会使学习的算法复杂化，例如：RBMs需要近似对数划分log-partition函数的梯度，这样才可能优化数据的似然函数。Sparse coding需要在每次的迭代过程中寻找活跃的基的系数，这是比较耗时的。而且，稀疏因子也是一个需要调整的参数。本文方法主要是绕过对数据分布的估计，直接分析优化特征的分布。那怎样的特征的分布才是优的呢？到这里，我们需要先关注特征的一些主要属性：population sparsity，lifetime sparsity和 high dispersal。怎样的特征才是好的特征，才是对分类或者其他任务好的特征。我们的学习算法就应该学会去提取这种特征。

**二、特征分布**

         上面讨论的特征学习算法都可以认为是生成特定的特征分布Feature distributions。例如，sparse coding只用少许的非零系数（特征）来描述每个样本。而一个面向特征分布的方法可以认为是直接优化特征分布的一些属性，使其可以更好的描述样本。

         我们引入一个特征分布矩阵，矩阵的每一行是一个特征，每一列是一个样本。每个元素http://img.blog.csdn.net/20130815142520859?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvem91eHkwOQ==/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/SouthEast表示第i个样本的第j个特征的激活值。由上面的分析可以知道，这是一个由输入到特征的映射函数（特征提取函数）来得到。

         下面我们就来讨论下什么样的特征分布才是好的：

**1）每个样本的特征应该是稀疏的（Population Sparsity）**

         每个样本都只用很少的激活（非零）特征来描述。具体来说，对于特征矩阵的每一列（一个样本）f(i)，只有很少的非零元素。其他的都是0 。例如，一幅图像可以由里面包含的一些目标来描述，如果里面存在很多可能的目标，那么在某一时刻，也只可能出现一些。我们称之为population sparsity（种群稀疏）。

**2）样本间的特征应该是稀疏的（Lifetime Sparsity）**

         好的特征应该是具有区分性的，这样才可以区分样本。例如需要区分人脸和人手，那么很明显，肤色不是区分性的特征，因为人脸和人手都有肤色。但如果看有没有眼睛，那么就很容易区分是人脸还是人手了，所以眼睛就是一个区分性的特征。所以要区分样本，就要选择样本独有的，而不是大家都有的特征。稍微学术点的表达就是，每个特征只允许在少量的样本内被激活。也就是说，在特征矩阵中，每一行（一种特征）应该只有少量的非零元素。这个特征的属性被称为lifetime sparsity（存在稀疏）。

**3）特征的分布应该是均匀的（High Dispersal）**

         对每一行（一种特征在不同样本的时候的不同取值）的特征的分布，应该和其他行的特征的分布相似，或者说每种特征都应该具有相似的统计特性。具体来说，对矩阵的每一行，我们取该行所有元素（一种特征在不同样本的时候的不同取值）的平方后的均值作为其统计特性的描述。每一行都存在一个均值，那么每行的均值都应该是一样的，这样就可以认为所有的特征都具有相似的分布。这种属性我们称之为high dispersal（高分散性）。但对于一个好的特征描述来说，这个属性并不是必要的。但它可以防止特征的退化，也就是可以防止提取到相同的特征（如果提取到相同的特征，那么特征既冗余，又没有增加信息量，所以一般都要求提取到的特征是正交的）。对于过完备的特征表达。high dispersal可以理解为只有很少的inactive不活跃的特征。例如，PCA编码一般不会满足high dispersal，因为大的特征值对应的特征向量（也就是特征code）大部分总是活跃active的。

        很多特征学习方法其实有包含上面的这些约束的。例如sparse RBM会约束一个特征的激活值靠近一个目标的值（lifetime sparsity）。ICA会归一化每个特征，还会优化特征的lifetime sparsity。Sparse autoencoder也会显式的优化lifetime sparsity。

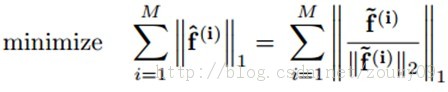
         另外，基于聚类的算法，例如K-means，是population sparsity约束的一种极端形式，它的一个聚类中心只对应一个特征，对每个样本来说，只有一个特征是激活的（只有一个值是1，其他全是0）。Triangle三角激活函数，本质上也会保证population sparsity。Sparse coding实际上也可以被视为存在population sparsity。

         本文中，我们从特征分布的角度去推导得到一种简单的特征学习算法。它仅仅需要优化high dispersal和population sparsity。因为在我们的实验中，我们发现，实现这两种特征的属性对学习一个过完备的特征表达来说已经足够了。在后面，我们会说明，这两种属性的结合实际上已经蕴含着保证特征的lifetime sparsity。

**三、Sparse filtering**

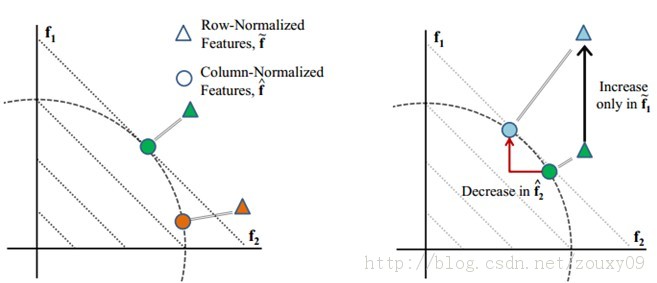
         下面我们就说明下，sparse filtering是如何捕捉到上面说的那些特性的。我们先考虑下从每个样本中计算线性特征。具体来说，我们用http://img.blog.csdn.net/20130815142701328?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvem91eHkwOQ==/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/SouthEast来表示第i个样本（特征矩阵中第i列）的第j个特征值（特征矩阵中第j行）。因为是线性特征，所以http://img.blog.csdn.net/20130815142837968?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvem91eHkwOQ==/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/SouthEast。第一步，我们先简单的对特征矩阵的行进行归一化，然后再对列进行归一化，然后再将矩阵中所有元素的绝对值求和。

         具体来说，我们先归一化每个特征为相等的激活值。具体做法是将每一个特征除以其在所有样本的二范数：http://img.blog.csdn.net/20130815142857781?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvem91eHkwOQ==/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/SouthEast。然后我们再归一化每一个样本的特征。这样，他们就会落在二范数的单位球体unit L2-ball上面了。具体做法是：http://img.blog.csdn.net/20130815142910718?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvem91eHkwOQ==/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/SouthEast。这时候，我们就可以对这些归一化过的特征进行优化了。我们使用L1范数惩罚来约束稀疏性。对于一个有M个样本的数据集， sparse filtering的目标函数表示为：



**3.1、Optimizing for population sparsity**

         其中http://img.blog.csdn.net/20130815142942843?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvem91eHkwOQ==/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/SouthEast这一项度量的就是第i个样本的特征的population sparsity，也就是限制每个样本只有很少的非零值。因为归一化的特征http://img.blog.csdn.net/20130815143000609?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvem91eHkwOQ==/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/SouthEast被约束只能落在二范数的单位球体上面，所以当这些特征是稀疏的时候，也就是样本接近特征坐标轴的时候，上面的目标函数才会最小化。反之，如果一个样本的每个特征值都差不多，那么就会导致一个很高的惩罚。可能有点难理解，我们看下图：



       左图：假设我们的特征维数是两维(f1, f2)，我们有两个样本，绿色和褐色的。每个样本都会先投影到二范数的球体上面（二维的话就是单位圆），再进行稀疏性的优化。可以看到，当样本落在坐标轴的时候，特征具有最大的稀疏性（例如，一个样本落在f2轴上，那么这个样本的表示就是(0, 1)，一个特征值为1，其他的为0，那么很明显它具有最大的稀疏性）。右图：因为归一化，特征之间会存在竞争。上面有一个只在f1特征上增加的样本。可以看到，尽管它只在f1方向上增加（绿色三角型转移到蓝色三角型），经过列归一化后（投影到单位圆上），可以看到第二个特征f2会减少（绿色圆圈转移到蓝色圆圈）。也就是说特征之间存在竞争，我变大，你就得变小。

         对特征进行归一化的一个属性就是它会隐含的进行特征间的竞争。归一化会使得如果只有一个特征分量f(i)增大，那么其他所有的特征分量的值将会减小。相似的，如果只有一个特征分量f(i)减小，那么其他所有的特征分量的值将会增大。因此，我们最小化http://img.blog.csdn.net/20130815143226437?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvem91eHkwOQ==/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/SouthEast，将会驱使归一化的特征趋于稀疏和大部分接近于0 。也就是，一些特征会比较大，其他的特征值都很小（接近于0）。因此，这个目标函数会优化特征的population sparsity。

         上面的公式与Treves-Rolls的population/life-time sparsity的度量非常相似：

http://img.blog.csdn.net/20130815143314921?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvem91eHkwOQ==/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/SouthEast

        这里F是特征的个数。这个度量常用来衡量大脑神经元激活的稀疏度。我们提出的公式可以看成是这个度量的平方根，再乘以一个尺度因子。

**3.2、Optimizing for high dispersal**

         上面说到，特征的high dispersal属性要求每个特征被恒等激活。在这里，我们粗鲁地强制每个特征的激活值平方后的均值相等。在上面sparse filtering的公式中，我们首先通过将每个特征除以它在所有样本上面的二范数来归一化每个特征，使他们具有相同的激活值：http://img.blog.csdn.net/20130815143336343?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvem91eHkwOQ==/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/SouthEast。实际上，它和约束每个特征具有相同的平方期望值有一样的效果。http://img.blog.csdn.net/20130815143353093?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvem91eHkwOQ==/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/SouthEast，因此，它已经隐含的优化了high dispersal属性。

**3.3、Optimizing for lifetime sparsity**

         我们发现，对population sparsity和 high dispersal的优化就已经隐含的优化了特征的lifetime sparsity。这其中的缘由是什么呢。首先，一个具有population sparsity的特征分布在特征矩阵里会存在很多非激活的元素（为0的元素）。而且，因为满足high dispersal，这些零元素会近似均匀的分布在所有的特征里。因此，每一个特征必然会有一定数量的零元素，从而保证了lifetime sparsity。所以，对于population sparsity和 high dispersal的优化就已经足够得到一个好的特征描述了。

**四、Deep sparse filtering**

         因为sparse filtering的目标函数是不可知的，我们可以自由地选择一个前向网络来计算这些特征。一般来说，我们都使用比较复杂的非线性函数，例如：

http://img.blog.csdn.net/20130815143451578?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvem91eHkwOQ==/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/SouthEast

或者多层网络来计算这些特征。这样，sparse filtering也算是训练深度网络的一种自然的框架了。

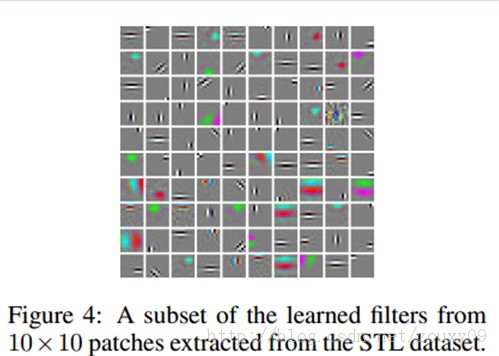
         有sparse filtering的深度网络可以使用权威的逐层贪婪算法来训练。我们可以先用sparse filtering来训练得到一个单层的归一化的特征，然后用它当成第二层的输入去训练第二层，其他层一样。实验中，我们发现，它能学习到很有意义的特征表达。

**五、实验与分析**

        在我们的实验中，我们采用软绝对值函数作为我们的激活函数：

http://img.blog.csdn.net/20130815143459906?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvem91eHkwOQ==/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/SouthEast

       其中ɛ=10^-8，然后用现有的L-BFGS算法来优化sparse filtering的目标函数直至收敛。



         其他的实验结果看原文。

**六、总结与讨论**

**6.1、与divisive normalization的联系**

         sparse filtering的population sparsity和divisive normalization有着紧密的联系。divisive normalization是一种视觉处理过程，一个神经元的响应会除以其邻域所有神经元的响应的和（或加权和）。divisive normalization在多级目标识别中是很有效的。然而，它被当成一个预处理阶段引入，而不是属于非监督学习（预训练）的一部分。实际上，sparse filtering把divisive normalization结合到了特征学习过程中，通过让特征间竞争，学习到满足population sparse的特征表达。

**6.2、与ICA 和sparse coding的联系**

         sparse filtering的目标函数可以看出ICA目标函数的归一化版本。在ICA中，目标函数是最小化线性滤波器组的响应，例如||**Wx**||1，但需要满足滤波器间相互正交的约束。正交约束保证我们学习到的特征是不同的。在sparse filtering中，我们用一个归一化的稀疏惩罚来替代这个约束，每个滤波器的响应都会除以所有滤波器响应的二范数||**Wx**||1/||**Wx**||2，在这里，就给滤波器引入了竞争，而不需要添加正交化。

         同样，我们也可以将归一化的思想运用到sparse coding框架。在sparse coding中，类似于L1/L2的稀疏惩罚已经被使用了。我们用一个归一化的惩罚，例如||**s**||1/||**s**||2来替换一般的L1范数的惩罚，例如||**s**||1。归一化惩罚具有尺度不变性，对于数据的变化可以更加鲁棒。

# [Deep Learning论文笔记之（四）CNN卷积神经网络推导和实现](http://blog.csdn.net/zouxy09/article/details/9993371)

自己平时看了一些论文，但老感觉看完过后就会慢慢的淡忘，某一天重新拾起来的时候又好像没有看过一样。所以想习惯地把一些感觉有用的论文中的知识点总结整理一下，一方面在整理过程中，自己的理解也会更深，另一方面也方便未来自己的勘察。更好的还可以放到博客上面与大家交流。因为基础有限，所以对论文的一些理解可能不太正确，还望大家不吝指正交流，谢谢。

       本文的论文来自：

[Notes on Convolutional Neural Networks,](http://cogprints.org/5869/1/cnn_tutorial.pdf) Jake Bouvrie。

         这个主要是CNN的推导和实现的一些笔记，再看懂这个笔记之前，最好具有CNN的一些基础。这里也先列出一个资料供参考：

[1] [Deep Learning（深度学习）学习笔记整理系列之（七）](http://blog.csdn.net/zouxy09/article/details/8781543)

[2] [LeNet-5, convolutional neural networks](http://yann.lecun.com/exdb/lenet/index.html)

[3][卷积神经网络](http://wenku.baidu.com/view/cd16fb8302d276a200292e22.html)

[4] [Neural Network for Recognition of Handwritten Digits](http://www.codeproject.com/Articles/16650/Neural-Network-for-Recognition-of-Handwritten-Digi)

[5] [Deep learning：三十八(Stacked CNN简单介绍)](http://www.cnblogs.com/tornadomeet/archive/2013/05/05/3061457.html)

[6] Gradient-based learning applied to document recognition.

[7]Imagenet classification with deep convolutional neural networks.

[8] UFLDL中的“[卷积特征提取](http://deeplearning.stanford.edu/wiki/index.php/%E5%8D%B7%E7%A7%AF%E7%89%B9%E5%BE%81%E6%8F%90%E5%8F%96)”和“[池化](http://deeplearning.stanford.edu/wiki/index.php/%E6%B1%A0%E5%8C%96)”。

        另外，这里有个matlab的[Deep Learning的toolbox](https://github.com/rasmusbergpalm/DeepLearnToolbox)，里面包含了CNN的代码，在下一个博文中，我将会详细注释这个代码。这个笔记对这个代码的理解非常重要。

         下面是自己对其中的一些知识点的理解：

**《Notes on Convolutional Neural Networks》**

**一、介绍**

         这个文档讨论的是CNNs的推导和实现。CNN架构的连接比权值要多很多，这实际上就隐含着实现了某种形式的规则化。这种特别的网络假定了我们希望通过数据驱动的方式学习到一些滤波器，作为提取输入的特征的一种方法。

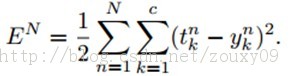
         本文中，我们先对训练全连接网络的经典BP算法做一个描述，然后推导2D CNN网络的卷积层和子采样层的BP权值更新方法。在推导过程中，我们更强调实现的效率，所以会给出一些Matlab代码。最后，我们转向讨论如何自动地学习组合前一层的特征maps，特别地，我们还学习特征maps的稀疏组合。

**二、全连接的反向传播算法**

         典型的CNN中，开始几层都是卷积和下采样的交替，然后在最后一些层（靠近输出层的），都是全连接的一维网络。这时候我们已经将所有两维2D的特征maps转化为全连接的一维网络的输入。这样，当你准备好将最终的2D特征maps输入到1D网络中时，一个非常方便的方法就是把所有输出的特征maps连接成一个长的输入向量。然后我们回到BP算法的讨论。（更详细的基础推导可以参考UFLDL中“[反向传导算法](http://deeplearning.stanford.edu/wiki/index.php/%E5%8F%8D%E5%90%91%E4%BC%A0%E5%AF%BC%E7%AE%97%E6%B3%95)”）。

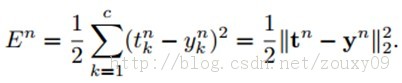
**2.1、Feedforward Pass前向传播**

         在下面的推导中，我们采用平方误差代价函数。我们讨论的是多类问题，共c类，共N个训练样本。



         这里http://img.blog.csdn.net/20130816002717187?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvem91eHkwOQ==/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/SouthEast表示第n个样本对应的标签的第k维。http://img.blog.csdn.net/20130816002727875?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvem91eHkwOQ==/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/SouthEast表示第n个样本对应的网络输出的第k个输出。对于多类问题，输出一般组织为“one-of-c”的形式，也就是只有该输入对应的类的输出节点输出为正，其他类的位或者节点为0或者负数，这个取决于你输出层的激活函数。sigmoid就是0，tanh就是-1.

         因为在全部训练集上的误差只是每个训练样本的误差的总和，所以这里我们先考虑对于一个样本的BP。对于第n个样本的误差，表示为：



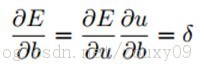
       传统的全连接神经网络中，我们需要根据BP规则计算代价函数E关于网络每一个权值的偏导数。我们用l来表示当前层，那么当前层的输出可以表示为：

http://img.blog.csdn.net/20130816002804250?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvem91eHkwOQ==/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/SouthEast

       输出激活函数f(.)可以有很多种，一般是sigmoid函数或者双曲线正切函数。sigmoid将输出压缩到[0, 1]，所以最后的输出平均值一般趋于0 。所以如果将我们的训练数据归一化为零均值和方差为1，可以在梯度下降的过程中增加收敛性。对于归一化的数据集来说，双曲线正切函数也是不错的选择。

**2.2、Backpropagation Pass反向传播**

         反向传播回来的误差可以看做是每个神经元的基的灵敏度sensitivities（灵敏度的意思就是我们的基b变化多少，误差会变化多少，也就是误差对基的变化率，也就是导数了），定义如下：（第二个等号是根据求导的链式法则得到的）



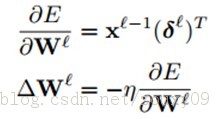
         因为∂u/∂b=1，所以∂E/∂b=∂E/∂u=δ，也就是说bias基的灵敏度∂E/∂b=δ和误差E对一个节点全部输入u的导数∂E/∂u是相等的。这个导数就是让高层误差反向传播到底层的神来之笔。反向传播就是用下面这条关系式：（下面这条式子表达的就是第l层的灵敏度，就是）

http://img.blog.csdn.net/20130816002937765?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvem91eHkwOQ==/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/SouthEast 公式（1）

         这里的“◦”表示每个元素相乘。输出层的神经元的灵敏度是不一样的：

http://img.blog.csdn.net/20130816002950875?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvem91eHkwOQ==/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/SouthEast

         最后，对每个神经元运用delta（即δ）规则进行权值更新。具体来说就是，对一个给定的神经元，得到它的输入，然后用这个神经元的delta（即δ）来进行缩放。用向量的形式表述就是，对于第l层，误差对于该层每一个权值（组合为矩阵）的导数是该层的输入（等于上一层的输出）与该层的灵敏度（该层每个神经元的δ组合成一个向量的形式）的叉乘。然后得到的偏导数乘以一个负学习率就是该层的神经元的权值的更新了：

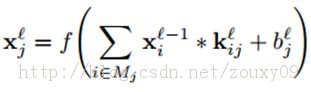
公式（2）

         对于bias基的更新表达式差不多。实际上，对于每一个权值(W)ij都有一个特定的学习率ηIj。

**三、Convolutional Neural Networks 卷积神经网络**

**3.1、Convolution Layers 卷积层**

         我们现在关注网络中卷积层的BP更新。在一个卷积层，上一层的特征maps被一个可学习的卷积核进行卷积，然后通过一个激活函数，就可以得到输出特征map。每一个输出map可能是组合卷积多个输入maps的值：



       这里Mj表示选择的输入maps的集合，那么到底选择哪些输入maps呢？有选择一对的或者三个的。但下面我们会讨论如何去自动选择需要组合的特征maps。每一个输出map会给一个额外的偏置b，但是对于一个特定的输出map，卷积每个输入maps的卷积核是不一样的。也就是说，如果输出特征map j和输出特征map k都是从输入map i中卷积求和得到，那么对应的卷积核是不一样的。

**3.1.1、Computing the Gradients梯度计算**

         我们假定每个卷积层l都会接一个下采样层l+1 。对于BP来说，根据上文我们知道，要想求得层l的每个神经元对应的权值的权值更新，就需要先求层l的每一个神经节点的灵敏度δ（也就是权值更新的公式（2））。为了求这个灵敏度我们就需要先对下一层的节点（连接到当前层l的感兴趣节点的第l+1层的节点）的灵敏度求和（得到δl+1），然后乘以这些连接对应的权值（连接第l层感兴趣节点和第l+1层节点的权值）W。再乘以当前层l的该神经元节点的输入u的激活函数f的导数值（也就是那个灵敏度反向传播的公式（1）的δl的求解），这样就可以得到当前层l每个神经节点对应的灵敏度δl了。

      然而，因为下采样的存在，采样层的一个像素（神经元节点）对应的灵敏度δ对应于卷积层（上一层）的输出map的一块像素（采样窗口大小）。因此，层l中的一个map的每个节点只与l+1层中相应map的一个节点连接。

     为了有效计算层l的灵敏度，我们需要上采样upsample 这个下采样downsample层对应的灵敏度map（特征map中每个像素对应一个灵敏度，所以也组成一个map），这样才使得这个灵敏度map大小与卷积层的map大小一致，然后再将层l的map的激活值的偏导数与从第l+1层的上采样得到的灵敏度map逐元素相乘（也就是公式（1））。

        在下采样层map的权值都取一个相同值β，而且是一个常数。所以我们只需要将上一个步骤得到的结果乘以一个β就可以完成第l层灵敏度δ的计算。

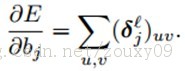
       我们可以对卷积层中每一个特征map j重复相同的计算过程。但很明显需要匹配相应的子采样层的map（参考公式（1））：

http://img.blog.csdn.net/20130816003129515?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvem91eHkwOQ==/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/SouthEast

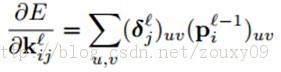
        up(.)表示一个上采样操作。如果下采样的采样因子是n的话，它简单的将每个像素水平和垂直方向上拷贝n次。这样就可以恢复原来的大小了。实际上，这个函数可以用Kronecker乘积来实现：

http://img.blog.csdn.net/20130816003155078?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvem91eHkwOQ==/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/SouthEast

       好，到这里，对于一个给定的map，我们就可以计算得到其灵敏度map了。然后我们就可以通过简单的对层l中的灵敏度map中所有节点进行求和快速的计算bias基的梯度了：

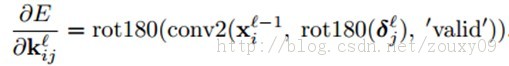
 公式（3）

       最后，对卷积核的权值的梯度就可以用BP算法来计算了（公式（2））。另外，很多连接的权值是共享的，因此，对于一个给定的权值，我们需要对所有与该权值有联系（权值共享的连接）的连接对该点求梯度，然后对这些梯度进行求和，就像上面对bias基的梯度计算一样：



       这里，http://img.blog.csdn.net/20130816003313265?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvem91eHkwOQ==/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/SouthEast是http://img.blog.csdn.net/20130816003318046?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvem91eHkwOQ==/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/SouthEast中的在卷积的时候与http://img.blog.csdn.net/20130816003328953?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvem91eHkwOQ==/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/SouthEast逐元素相乘的patch，输出卷积map的(u, v)位置的值是由上一层的(u, v)位置的patch与卷积核k\_ij逐元素相乘的结果。

      咋一看，好像我们需要煞费苦心地记住输出map（和对应的灵敏度map）每个像素对应于输入map的哪个patch。但实际上，在Matlab中，可以通过一个代码就实现。对于上面的公式，可以用Matlab的卷积函数来实现：



       我们先对delta灵敏度map进行旋转，这样就可以进行互相关计算，而不是卷积（在卷积的数学定义中，特征矩阵（卷积核）在传递给conv2时需要先翻转（flipped）一下。也就是颠倒下特征矩阵的行和列）。然后把输出反旋转回来，这样我们在前向传播进行卷积的时候，卷积核才是我们想要的方向。

**3.2、Sub-sampling Layers 子采样层**

         对于子采样层来说，有N个输入maps，就有N个输出maps，只是每个输出map都变小了。

http://img.blog.csdn.net/20130816003432812?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvem91eHkwOQ==/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/SouthEast

        down(.)表示一个下采样函数。典型的操作一般是对输入图像的不同nxn的块的所有像素进行求和。这样输出图像在两个维度上都缩小了n倍。每个输出map都对应一个属于自己的乘性偏置β和一个加性偏置b。

**3.2.1、Computing the Gradients 梯度计算**

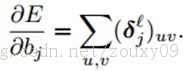
         这里最困难的是计算灵敏度map。一旦我们得到这个了，那我们唯一需要更新的偏置参数β和b就可以轻而易举了（公式（3））。如果下一个卷积层与这个子采样层是全连接的，那么就可以通过BP来计算子采样层的灵敏度maps。

         我们需要计算卷积核的梯度，所以我们必须找到输入map中哪个patch对应输出map的哪个像素。这里，就是必须找到当前层的灵敏度map中哪个patch对应与下一层的灵敏度map的给定像素，这样才可以利用公式（1）那样的δ递推，也就是灵敏度反向传播回来。另外，需要乘以输入patch与输出像素之间连接的权值，这个权值实际上就是卷积核的权值（已旋转的）。

http://img.blog.csdn.net/20130816003501218?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvem91eHkwOQ==/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/SouthEast

      在这之前，我们需要先将核旋转一下，让卷积函数可以实施互相关计算。另外，我们需要对卷积边界进行处理，但在Matlab里面，就比较容易处理。Matlab中全卷积会对缺少的输入像素补0 。

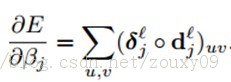
      到这里，我们就可以对b和β计算梯度了。首先，加性基b的计算和上面卷积层的一样，对灵敏度map中所有元素加起来就可以了：



       而对于乘性偏置β，因为涉及到了在前向传播过程中下采样map的计算，所以我们最好在前向的过程中保存好这些maps，这样在反向的计算中就不用重新计算了。我们定义：

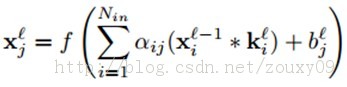
http://img.blog.csdn.net/20130816003553468?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvem91eHkwOQ==/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/SouthEast

这样，对β的梯度就可以用下面的方式计算：



**3.3、Learning Combinations of Feature Maps 学习特征map的组合**

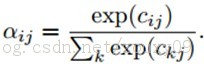
         大部分时候，通过卷积多个输入maps，然后再对这些卷积值求和得到一个输出map，这样的效果往往是比较好的。在一些文献中，一般是人工选择哪些输入maps去组合得到一个输出map。但我们这里尝试去让CNN在训练的过程中学习这些组合，也就是让网络自己学习挑选哪些输入maps来计算得到输出map才是最好的。我们用αij表示在得到第j个输出map的其中第i个输入map的权值或者贡献。这样，第j个输出map可以表示为：



         需要满足约束：

http://img.blog.csdn.net/20130816003641828?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvem91eHkwOQ==/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/SouthEast

         这些对变量αij的约束可以通过将变量αij表示为一个组无约束的隐含权值cij的softmax函数来加强。（因为softmax的因变量是自变量的指数函数，他们的变化率会不同）。

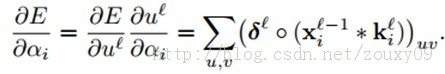


         因为对于一个固定的j来说，每组权值cij都是和其他组的权值独立的，所以为了方面描述，我们把下标j去掉，只考虑一个map的更新，其他map的更新是一样的过程，只是map的索引j不同而已。

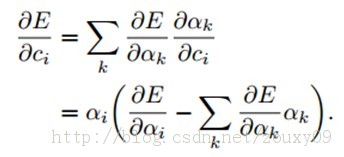
         Softmax函数的导数表示为：

http://img.blog.csdn.net/20130816003718375?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvem91eHkwOQ==/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/SouthEast

        这里的δ是Kronecker delta。对于误差对于第l层变量αi的导数为：



         最后就可以通过链式规则去求得代价函数关于权值ci的偏导数了：



**3.3.1、Enforcing Sparse Combinations 加强稀疏性组合**

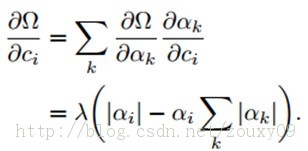
         为了限制αi是稀疏的，也就是限制一个输出map只与某些而不是全部的输入maps相连。我们在整体代价函数里增加稀疏约束项Ω(α)。对于单个样本，重写代价函数为：

http://img.blog.csdn.net/20130816003835515?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvem91eHkwOQ==/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/SouthEast

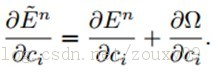
然后寻找这个规则化约束项对权值ci求导的贡献。规则化项Ω(α)对αi求导是：

http://img.blog.csdn.net/20130816003848406?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvem91eHkwOQ==/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/SouthEast

         然后，通过链式法则，对ci的求导是：



         所以，权值ci最后的梯度是：



**3.4、Making it Fast with MATLAB**

        CNN的训练主要是在卷积层和子采样层的交互上，其主要的计算瓶颈是：

1）前向传播过程：下采样每个卷积层的maps；

2）反向传播过程：上采样高层子采样层的灵敏度map，以匹配底层的卷积层输出maps的大小；

3）sigmoid的运用和求导。

         对于第一和第二个问题，我们考虑的是如何用Matlab内置的图像处理函数去实现上采样和下采样的操作。对于上采样，imresize函数可以搞定，但需要很大的开销。一个比较快速的版本是使用Kronecker乘积函数kron。通过一个全一矩阵ones来和我们需要上采样的矩阵进行Kronecker乘积，就可以实现上采样的效果。对于前向传播过程中的下采样，imresize并没有提供在缩小图像的过程中还计算nxn块内像素的和的功能，所以没法用。一个比较好和快速的方法是用一个全一的卷积核来卷积图像，然后简单的通过标准的索引方法来采样最后卷积结果。例如，如果下采样的域是2x2的，那么我们可以用2x2的元素全是1的卷积核来卷积图像。然后再卷积后的图像中，我们每个2个点采集一次数据，y=x(1:2:end,1:2:end)，这样就可以得到了两倍下采样，同时执行求和的效果。

         对于第三个问题，实际上有些人以为Matlab中对sigmoid函数进行inline的定义会更快，其实不然，Matlab与C/C++等等语言不一样，Matlab的inline反而比普通的函数定义更非时间。所以，我们可以直接在代码中使用计算sigmoid函数及其导数的真实代码。

# [Deep Learning论文笔记之（五）CNN卷积神经网络代码理解](http://blog.csdn.net/zouxy09/article/details/9993743)

自己平时看了一些论文，但老感觉看完过后就会慢慢的淡忘，某一天重新拾起来的时候又好像没有看过一样。所以想习惯地把一些感觉有用的论文中的知识点总结整理一下，一方面在整理过程中，自己的理解也会更深，另一方面也方便未来自己的勘察。更好的还可以放到博客上面与大家交流。因为基础有限，所以对论文的一些理解可能不太正确，还望大家不吝指正交流，谢谢。

       本文的代码来自githup的[Deep Learning的toolbox](https://github.com/rasmusbergpalm/DeepLearnToolbox)，（在这里，先感谢该toolbox的作者）里面包含了很多Deep Learning方法的代码。是用Matlab编写的（另外，有人翻译成了C++和python的版本了）。本文中我们主要解读下CNN的代码。详细的注释见代码。

       在读代码之前，最好先阅读下我的上一个博文：

         Deep Learning论文笔记之（四）CNN卷积神经网络推导和实现

<http://blog.csdn.net/zouxy09/article/details/9993371>

       里面包含的是我对一个作者的CNN笔记的翻译性的理解，对CNN的推导和实现做了详细的介绍，看明白这个笔记对代码的理解非常重要，所以强烈建议先看懂上面这篇文章。

         下面是自己对代码的注释：

**cnnexamples.m**

**[plain]** [view plaincopy](http://blog.csdn.net/zouxy09/article/details/9993743)

1. clear all; close all; clc;
2. addpath('../data');
3. addpath('../util');
4. load mnist\_uint8;
6. train\_x = double(reshape(train\_x',28,28,60000))/255;
7. test\_x = double(reshape(test\_x',28,28,10000))/255;
8. train\_y = double(train\_y');
9. test\_y = double(test\_y');
11. %% ex1
12. %will run 1 epoch in about 200 second and get around 11% error.
13. %With 100 epochs you'll get around 1.2% error
15. cnn.layers = {
16. struct('type', 'i') %input layer
17. struct('type', 'c', 'outputmaps', 6, 'kernelsize', 5) %convolution layer
18. struct('type', 's', 'scale', 2) %sub sampling layer
19. struct('type', 'c', 'outputmaps', 12, 'kernelsize', 5) %convolution layer
20. struct('type', 's', 'scale', 2) %subsampling layer
21. };
23. % 这里把cnn的设置给cnnsetup，它会据此构建一个完整的CNN网络，并返回
24. cnn = cnnsetup(cnn, train\_x, train\_y);
26. % 学习率
27. opts.alpha = 1;
28. % 每次挑出一个batchsize的batch来训练，也就是每用batchsize个样本就调整一次权值，而不是
29. % 把所有样本都输入了，计算所有样本的误差了才调整一次权值
30. opts.batchsize = 50;
31. % 训练次数，用同样的样本集。我训练的时候：
32. % 1的时候 11.41% error
33. % 5的时候 4.2% error
34. % 10的时候 2.73% error
35. opts.numepochs = 10;
37. % 然后开始把训练样本给它，开始训练这个CNN网络
38. cnn = cnntrain(cnn, train\_x, train\_y, opts);
40. % 然后就用测试样本来测试
41. [er, bad] = cnntest(cnn, test\_x, test\_y);
43. %plot mean squared error
44. plot(cnn.rL);
45. %show test error
46. disp([num2str(er\*100) '% error']);

**cnnsetup.m**

**[plain]** [view plaincopy](http://blog.csdn.net/zouxy09/article/details/9993743)

1. function net = cnnsetup(net, x, y)
2. inputmaps = 1;
3. % B=squeeze(A) 返回和矩阵A相同元素但所有单一维都移除的矩阵B，单一维是满足size(A,dim)=1的维。
4. % train\_x中图像的存放方式是三维的reshape(train\_x',28,28,60000)，前面两维表示图像的行与列，
5. % 第三维就表示有多少个图像。这样squeeze(x(:, :, 1))就相当于取第一个图像样本后，再把第三维
6. % 移除，就变成了28x28的矩阵，也就是得到一幅图像，再size一下就得到了训练样本图像的行数与列数了
7. mapsize = size(squeeze(x(:, :, 1)));
9. % 下面通过传入net这个结构体来逐层构建CNN网络
10. % n = numel(A)返回数组A中元素个数
11. % net.layers中有五个struct类型的元素，实际上就表示CNN共有五层，这里范围的是5
12. for l = 1 : numel(net.layers)   %  layer
13. if strcmp(net.layers{l}.type, 's') % 如果这层是 子采样层
14. % subsampling层的mapsize，最开始mapsize是每张图的大小28\*28
15. % 这里除以scale=2，就是pooling之后图的大小，pooling域之间没有重叠，所以pooling后的图像为14\*14
16. % 注意这里的右边的mapsize保存的都是上一层每张特征map的大小，它会随着循环进行不断更新
17. mapsize = floor(mapsize / net.layers{l}.scale);
18. for j = 1 : inputmaps % inputmap就是上一层有多少张特征图
19. net.layers{l}.b{j} = 0; % 将偏置初始化为0
20. end
21. end
22. if strcmp(net.layers{l}.type, 'c') % 如果这层是 卷积层
23. % 旧的mapsize保存的是上一层的特征map的大小，那么如果卷积核的移动步长是1，那用
24. % kernelsize\*kernelsize大小的卷积核卷积上一层的特征map后，得到的新的map的大小就是下面这样
25. mapsize = mapsize - net.layers{l}.kernelsize + 1;
26. % 该层需要学习的参数个数。每张特征map是一个(后层特征图数量)\*(用来卷积的patch图的大小)
27. % 因为是通过用一个核窗口在上一个特征map层中移动（核窗口每次移动1个像素），遍历上一个特征map
28. % 层的每个神经元。核窗口由kernelsize\*kernelsize个元素组成，每个元素是一个独立的权值，所以
29. % 就有kernelsize\*kernelsize个需要学习的权值，再加一个偏置值。另外，由于是权值共享，也就是
30. % 说同一个特征map层是用同一个具有相同权值元素的kernelsize\*kernelsize的核窗口去感受输入上一
31. % 个特征map层的每个神经元得到的，所以同一个特征map，它的权值是一样的，共享的，权值只取决于
32. % 核窗口。然后，不同的特征map提取输入上一个特征map层不同的特征，所以采用的核窗口不一样，也
33. % 就是权值不一样，所以outputmaps个特征map就有（kernelsize\*kernelsize+1）\* outputmaps那么多的权值了
34. % 但这里fan\_out只保存卷积核的权值W，偏置b在下面独立保存
35. fan\_out = net.layers{l}.outputmaps \* net.layers{l}.kernelsize ^ 2;
36. for j = 1 : net.layers{l}.outputmaps  %  output map
37. % fan\_out保存的是对于上一层的一张特征map，我在这一层需要对这一张特征map提取outputmaps种特征，
38. % 提取每种特征用到的卷积核不同，所以fan\_out保存的是这一层输出新的特征需要学习的参数个数
39. % 而，fan\_in保存的是，我在这一层，要连接到上一层中所有的特征map，然后用fan\_out保存的提取特征
40. % 的权值来提取他们的特征。也即是对于每一个当前层特征图，有多少个参数链到前层
41. fan\_in = inputmaps \* net.layers{l}.kernelsize ^ 2;
42. for i = 1 : inputmaps  %  input map
43. % 随机初始化权值，也就是共有outputmaps个卷积核，对上层的每个特征map，都需要用这么多个卷积核
44. % 去卷积提取特征。
45. % rand(n)是产生n×n的 0-1之间均匀取值的数值的矩阵，再减去0.5就相当于产生-0.5到0.5之间的随机数
46. % 再 \*2 就放大到 [-1, 1]。然后再乘以后面那一数，why？
47. % 反正就是将卷积核每个元素初始化为[-sqrt(6 / (fan\_in + fan\_out)), sqrt(6 / (fan\_in + fan\_out))]
48. % 之间的随机数。因为这里是权值共享的，也就是对于一张特征map，所有感受野位置的卷积核都是一样的
49. % 所以只需要保存的是 inputmaps \* outputmaps 个卷积核。
50. net.layers{l}.k{i}{j} = (rand(net.layers{l}.kernelsize) - 0.5) \* 2 \* sqrt(6 / (fan\_in + fan\_out));
51. end
52. net.layers{l}.b{j} = 0; % 将偏置初始化为0
53. end
54. % 只有在卷积层的时候才会改变特征map的个数，pooling的时候不会改变个数。这层输出的特征map个数就是
55. % 输入到下一层的特征map个数
56. inputmaps = net.layers{l}.outputmaps;
57. end
58. end
60. % fvnum 是输出层的前面一层的神经元个数。
61. % 这一层的上一层是经过pooling后的层，包含有inputmaps个特征map。每个特征map的大小是mapsize。
62. % 所以，该层的神经元个数是 inputmaps \* （每个特征map的大小）
63. % prod: Product of elements.
64. % For vectors, prod(X) is the product of the elements of X
65. % 在这里 mapsize = [特征map的行数 特征map的列数]，所以prod后就是 特征map的行\*列
66. fvnum = prod(mapsize) \* inputmaps;
67. % onum 是标签的个数，也就是输出层神经元的个数。你要分多少个类，自然就有多少个输出神经元
68. onum = size(y, 1);
70. % 这里是最后一层神经网络的设定
71. % ffb 是输出层每个神经元对应的基biases
72. net.ffb = zeros(onum, 1);
73. % ffW 输出层前一层 与 输出层 连接的权值，这两层之间是全连接的
74. net.ffW = (rand(onum, fvnum) - 0.5) \* 2 \* sqrt(6 / (onum + fvnum));
75. end

**cnntrain.m**

**[plain]** [view plaincopy](http://blog.csdn.net/zouxy09/article/details/9993743)

1. function net = cnntrain(net, x, y, opts)
2. m = size(x, 3); % m 保存的是 训练样本个数
3. numbatches = m / opts.batchsize;
4. % rem: Remainder after division. rem(x,y) is x - n.\*y 相当于求余
5. % rem(numbatches, 1) 就相当于取其小数部分，如果为0，就是整数
6. if rem(numbatches, 1) ~= 0
7. error('numbatches not integer');
8. end
10. net.rL = [];
11. for i = 1 : opts.numepochs
12. % disp(X) 打印数组元素。如果X是个字符串，那就打印这个字符串
13. disp(['epoch ' num2str(i) '/' num2str(opts.numepochs)]);
14. % tic 和 toc 是用来计时的，计算这两条语句之间所耗的时间
15. tic;
16. % P = randperm(N) 返回[1, N]之间所有整数的一个随机的序列，例如
17. % randperm(6) 可能会返回 [2 4 5 6 1 3]
18. % 这样就相当于把原来的样本排列打乱，再挑出一些样本来训练
19. kk = randperm(m);
20. for l = 1 : numbatches
21. % 取出打乱顺序后的batchsize个样本和对应的标签
22. batch\_x = x(:, :, kk((l - 1) \* opts.batchsize + 1 : l \* opts.batchsize));
23. batch\_y = y(:,    kk((l - 1) \* opts.batchsize + 1 : l \* opts.batchsize));
25. % 在当前的网络权值和网络输入下计算网络的输出
26. net = cnnff(net, batch\_x); % Feedforward
27. % 得到上面的网络输出后，通过对应的样本标签用bp算法来得到误差对网络权值
28. %（也就是那些卷积核的元素）的导数
29. net = cnnbp(net, batch\_y); % Backpropagation
30. % 得到误差对权值的导数后，就通过权值更新方法去更新权值
31. net = cnnapplygrads(net, opts);
32. if isempty(net.rL)
33. net.rL(1) = net.L; % 代价函数值，也就是误差值
34. end
35. net.rL(end + 1) = 0.99 \* net.rL(end) + 0.01 \* net.L; % 保存历史的误差值，以便画图分析
36. end
37. toc;
38. end
40. end

**cnnff.m**

**[plain]** [view plaincopy](http://blog.csdn.net/zouxy09/article/details/9993743)

1. function net = cnnff(net, x)
2. n = numel(net.layers); % 层数
3. net.layers{1}.a{1} = x; % 网络的第一层就是输入，但这里的输入包含了多个训练图像
4. inputmaps = 1; % 输入层只有一个特征map，也就是原始的输入图像
6. for l = 2 : n   %  for each layer
7. if strcmp(net.layers{l}.type, 'c') % 卷积层
8. %  !!below can probably be handled by insane matrix operations
9. % 对每一个输入map，或者说我们需要用outputmaps个不同的卷积核去卷积图像
10. for j = 1 : net.layers{l}.outputmaps   %  for each output map
11. %  create temp output map
12. % 对上一层的每一张特征map，卷积后的特征map的大小就是
13. % （输入map宽 - 卷积核的宽 + 1）\* （输入map高 - 卷积核高 + 1）
14. % 对于这里的层，因为每层都包含多张特征map，对应的索引保存在每层map的第三维
15. % 所以，这里的z保存的就是该层中所有的特征map了
16. z = zeros(size(net.layers{l - 1}.a{1}) - [net.layers{l}.kernelsize - 1 net.layers{l}.kernelsize - 1 0]);
17. for i = 1 : inputmaps   %  for each input map
18. %  convolve with corresponding kernel and add to temp output map
19. % 将上一层的每一个特征map（也就是这层的输入map）与该层的卷积核进行卷积
20. % 然后将对上一层特征map的所有结果加起来。也就是说，当前层的一张特征map，是
21. % 用一种卷积核去卷积上一层中所有的特征map，然后所有特征map对应位置的卷积值的和
22. % 另外，有些论文或者实际应用中，并不是与全部的特征map链接的，有可能只与其中的某几个连接
23. z = z + convn(net.layers{l - 1}.a{i}, net.layers{l}.k{i}{j}, 'valid');
24. end
25. %  add bias, pass through nonlinearity
26. % 加上对应位置的基b，然后再用sigmoid函数算出特征map中每个位置的激活值，作为该层输出特征map
27. net.layers{l}.a{j} = sigm(z + net.layers{l}.b{j});
28. end
29. %  set number of input maps to this layers number of outputmaps
30. inputmaps = net.layers{l}.outputmaps;
31. elseif strcmp(net.layers{l}.type, 's') % 下采样层
32. %  downsample
33. for j = 1 : inputmaps
34. %  !! replace with variable
35. % 例如我们要在scale=2的域上面执行mean pooling，那么可以卷积大小为2\*2，每个元素都是1/4的卷积核
36. z = convn(net.layers{l - 1}.a{j}, ones(net.layers{l}.scale) / (net.layers{l}.scale ^ 2), 'valid');
37. % 因为convn函数的默认卷积步长为1，而pooling操作的域是没有重叠的，所以对于上面的卷积结果
38. % 最终pooling的结果需要从上面得到的卷积结果中以scale=2为步长，跳着把mean pooling的值读出来
39. net.layers{l}.a{j} = z(1 : net.layers{l}.scale : end, 1 : net.layers{l}.scale : end, :);
40. end
41. end
42. end
44. %  concatenate all end layer feature maps into vector
45. % 把最后一层得到的特征map拉成一条向量，作为最终提取到的特征向量
46. net.fv = [];
47. for j = 1 : numel(net.layers{n}.a) % 最后一层的特征map的个数
48. sa = size(net.layers{n}.a{j}); % 第j个特征map的大小
49. % 将所有的特征map拉成一条列向量。还有一维就是对应的样本索引。每个样本一列，每列为对应的特征向量
50. net.fv = [net.fv; reshape(net.layers{n}.a{j}, sa(1) \* sa(2), sa(3))];
51. end
52. %  feedforward into output perceptrons
53. % 计算网络的最终输出值。sigmoid(W\*X + b)，注意是同时计算了batchsize个样本的输出值
54. net.o = sigm(net.ffW \* net.fv + repmat(net.ffb, 1, size(net.fv, 2)));
56. end

**cnnbp.m**

**[plain]** [view plaincopy](http://blog.csdn.net/zouxy09/article/details/9993743)

1. function net = cnnbp(net, y)
2. n = numel(net.layers); % 网络层数
4. %  error
5. net.e = net.o - y;
6. %  loss function
7. % 代价函数是 均方误差
8. net.L = 1/2\* sum(net.e(:) .^ 2) / size(net.e, 2);
10. %%  backprop deltas
11. % 这里可以参考 UFLDL 的 反向传导算法 的说明
12. % 输出层的 灵敏度 或者 残差
13. net.od = net.e .\* (net.o .\* (1 - net.o));   %  output delta
14. % 残差 反向传播回 前一层
15. net.fvd = (net.ffW' \* net.od);              %  feature vector delta
16. if strcmp(net.layers{n}.type, 'c')         %  only conv layers has sigm function
17. net.fvd = net.fvd .\* (net.fv .\* (1 - net.fv));
18. end
20. %  reshape feature vector deltas into output map style
21. sa = size(net.layers{n}.a{1}); % 最后一层特征map的大小。这里的最后一层都是指输出层的前一层
22. fvnum = sa(1) \* sa(2); % 因为是将最后一层特征map拉成一条向量，所以对于一个样本来说，特征维数是这样
23. for j = 1 : numel(net.layers{n}.a) % 最后一层的特征map的个数
24. % 在fvd里面保存的是所有样本的特征向量（在cnnff.m函数中用特征map拉成的），所以这里需要重新
25. % 变换回来特征map的形式。d 保存的是 delta，也就是 灵敏度 或者 残差
26. net.layers{n}.d{j} = reshape(net.fvd(((j - 1) \* fvnum + 1) : j \* fvnum, :), sa(1), sa(2), sa(3));
27. end
29. % 对于 输出层前面的层（与输出层计算残差的方式不同）
30. for l = (n - 1) : -1 : 1
31. if strcmp(net.layers{l}.type, 'c')
32. for j = 1 : numel(net.layers{l}.a) % 该层特征map的个数
33. % net.layers{l}.d{j} 保存的是 第l层 的 第j个 map 的 灵敏度map。 也就是每个神经元节点的delta的值
34. % expand的操作相当于对l+1层的灵敏度map进行上采样。然后前面的操作相当于对该层的输入a进行sigmoid求导
35. % 这条公式请参考 Notes on Convolutional Neural Networks
36. % for k = 1:size(net.layers{l + 1}.d{j}, 3)
37. % net.layers{l}.d{j}(:,:,k) = net.layers{l}.a{j}(:,:,k) .\* (1 - net.layers{l}.a{j}(:,:,k)) .\*  kron(net.layers{l + 1}.d{j}(:,:,k), ones(net.layers{l + 1}.scale)) / net.layers{l + 1}.scale ^ 2;
38. % end
39. net.layers{l}.d{j} = net.layers{l}.a{j} .\* (1 - net.layers{l}.a{j}) .\* (expand(net.layers{l + 1}.d{j}, [net.layers{l + 1}.scale net.layers{l + 1}.scale 1]) / net.layers{l + 1}.scale ^ 2);
40. end
41. elseif strcmp(net.layers{l}.type, 's')
42. for i = 1 : numel(net.layers{l}.a) % 第l层特征map的个数
43. z = zeros(size(net.layers{l}.a{1}));
44. for j = 1 : numel(net.layers{l + 1}.a) % 第l+1层特征map的个数
45. z = z + convn(net.layers{l + 1}.d{j}, rot180(net.layers{l + 1}.k{i}{j}), 'full');
46. end
47. net.layers{l}.d{i} = z;
48. end
49. end
50. end
52. %%  calc gradients
53. % 这里与 Notes on Convolutional Neural Networks 中不同，这里的 子采样 层没有参数，也没有
54. % 激活函数，所以在子采样层是没有需要求解的参数的
55. for l = 2 : n
56. if strcmp(net.layers{l}.type, 'c')
57. for j = 1 : numel(net.layers{l}.a)
58. for i = 1 : numel(net.layers{l - 1}.a)
59. % dk 保存的是 误差对卷积核 的导数
60. net.layers{l}.dk{i}{j} = convn(flipall(net.layers{l - 1}.a{i}), net.layers{l}.d{j}, 'valid') / size(net.layers{l}.d{j}, 3);
61. end
62. % db 保存的是 误差对于bias基 的导数
63. net.layers{l}.db{j} = sum(net.layers{l}.d{j}(:)) / size(net.layers{l}.d{j}, 3);
64. end
65. end
66. end
67. % 最后一层perceptron的gradient的计算
68. net.dffW = net.od \* (net.fv)' / size(net.od, 2);
69. net.dffb = mean(net.od, 2);
71. function X = rot180(X)
72. X = flipdim(flipdim(X, 1), 2);
73. end
74. end

**cnnapplygrads.m**

**[plain]** [view plaincopy](http://blog.csdn.net/zouxy09/article/details/9993743)

1. function net = cnnapplygrads(net, opts)
2. for l = 2 : numel(net.layers)
3. if strcmp(net.layers{l}.type, 'c')
4. for j = 1 : numel(net.layers{l}.a)
5. for ii = 1 : numel(net.layers{l - 1}.a)
6. % 这里没什么好说的，就是普通的权值更新的公式：W\_new = W\_old - alpha \* de/dW（误差对权值导数）
7. net.layers{l}.k{ii}{j} = net.layers{l}.k{ii}{j} - opts.alpha \* net.layers{l}.dk{ii}{j};
8. end
9. end
10. net.layers{l}.b{j} = net.layers{l}.b{j} - opts.alpha \* net.layers{l}.db{j};
11. end
12. end
14. net.ffW = net.ffW - opts.alpha \* net.dffW;
15. net.ffb = net.ffb - opts.alpha \* net.dffb;
16. end

**cnntest.m**

**[plain]** [view plaincopy](http://blog.csdn.net/zouxy09/article/details/9993743)

1. function [er, bad] = cnntest(net, x, y)
2. %  feedforward
3. net = cnnff(net, x); % 前向传播得到输出
4. % [Y,I] = max(X) returns the indices of the maximum values in vector I
5. [~, h] = max(net.o); % 找到最大的输出对应的标签
6. [~, a] = max(y);     % 找到最大的期望输出对应的索引
7. bad = find(h ~= a);  % 找到他们不相同的个数，也就是错误的次数
9. er = numel(bad) / size(y, 2); % 计算错误率
10. end

# [Deep Learning论文笔记之（六）Multi-Stage多级架构分析](http://blog.csdn.net/zouxy09/article/details/10007237)

自己平时看了一些论文，但老感觉看完过后就会慢慢的淡忘，某一天重新拾起来的时候又好像没有看过一样。所以想习惯地把一些感觉有用的论文中的知识点总结整理一下，一方面在整理过程中，自己的理解也会更深，另一方面也方便未来自己的勘察。更好的还可以放到博客上面与大家交流。因为基础有限，所以对论文的一些理解可能不太正确，还望大家不吝指正交流，谢谢。

        本文的论文来自：

Kevin Jarrett, Koray Kavukcuoglu, Marc’Aurelio Ranzato, and Yann LeCun, ”[What is the Best Multi-Stage Architecture for Object Recognition?](http://cs.nyu.edu/~koray/publis/jarrett-iccv-09.pdf)”, in Proc. International Conference on Computer Vision (ICCV’09), 2009

         这里面还提供了一个精简版的[Matlab代码](http://cs.nyu.edu/~koray/publis/code/randomc101.tar.gz)。实现的是random convolutional filters 和linear logistic regression classifier的两级目标识别系统。

         下面是自己对其中的一些知识点的理解：

**《What is the Best Multi-Stage Architecture for Object Recognition?》**

         对目标识别怎样的多级架构Multi-Stage Architecture还是最好的？在当前的很多目标识别系统中，特征提取阶段一般由一组滤波器，再进行非线性变换和一些类型的特征pooling层组成。大部分系统都使用一级特征提取，这时候的滤波器是hard-wired（人工选择的，硬连线的，参数不可学习调整）的，或者使用两级，这时候其中的一级或者两级的滤波器都可以通过监督或者非监督的方式学习得到。

       本文关注三个问题：

1）滤波器组后面接的non-linearities非线性算法是如何影响识别的准确性的？

2）通过监督或者非监督方式学习到的滤波器组是否比随机的滤波器组或者人工指定的滤波器要好？

3）与仅有一级的特征提取对比，两级的特征提取是否还有其他优点？

       我们证明了：

1）使用包含校正和局部对比度归一化的非线性算子对增加目标识别的准确性来说有很大帮助。

2）两级的特征提取比一级的要好。准确率更高。

3）惊喜的是，用随机初始化的滤波器组的两级系统却可以在Caltech这个数据中达到63%的识别率。当然，这里面包含了合适的非线性算子和pooling层。

4）经过监督微调，系统在NORB数据库上达到当前领先水平。而且非监督预训练后再加监督微调可以在Caltech这个数据库中达到更好的准确率（大于63%）。然后在没有处理过的MNIST数据库中，可以达到目前我们知道的最低的0.53%的错误率。

**一、概述**

         在过去这几年，对于目标识别来说，出现了很多不错的特征描述子。很多方法都是把输入图像划分为一个个规律排列的密集的patch，然后提取这些patch的特征。再以某种方式组合这些patch的特征作为这个输入图像的特征。概括的来说，这些系统的很大一部分都是这样的一个特征提取过程：输入经过一个滤波器组filter bank（一般是基于方向性的边缘检测器），再经过一个非线性算子non-linear operation（quantization, winner-take-all, sparsification, normalization, and/or point-wise saturation），然后用一个pooling操作（把实空间或者特征空间邻域的值通过一个max, average, or histogramming operator）来绛维和得到一定的不变性。例如俺们熟知的SIFT特征，它先通过对每个小patch经过方向性边缘检测器，然后用winner-take-all算子来获取最显著的方向。最后，在更大块的patch上面统计局部方向的直方图，pooling成一个稀疏向量。

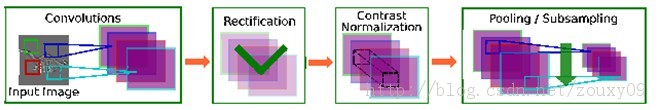
        对于一层特征提取的系统，也就是提取到了上面的这个特征后，例如SIFT，HOG等，然后直接接一个监督学习的分类器，就构成了一个目标识别系统。还有一些模型会使用两级或者更多级的特征提取器，然后再接一个监督学习分类器来构成一个比较复杂的目标识别系统。这些系统本质的差别在于：有一个或者多个特征提取层、滤波器组后使用的非线性算子、滤波器组的得到（人工选择、非监督学习还是监督学习）和顶层的分类器的选择（线性分类器还是更复杂的分类器）。

        一般对滤波器组的选择是Gabor小波，还有人选择一些简单的方向性检测滤波器组，也就是梯度算子，例如SIFT和HOG。还有一些直接通过非监督学习方法直接从训练数据中学习这些滤波器组。当在自然图像中训练的时候，学到的滤波器也是类似于Gabor边缘检测的东西。特征学习方法的一个好处就是它可以分级的学习特征。因为我们具有一定的先验知识，觉得第一级的特征就应该是边缘检测器，但第二层特征又应该是什么呢？人就没有这个类似的先验知识了。所以就比较难人工设计一个比较好二级特征提取器。所以说，二级或者多级特征必须让系统自己学习。现在出现的方法也很多了，有监督的，非监督的，或者两者联合的。

       咋一看，用像Caltech-101这些非常少的训练数据库（这个数据库要识别101类的物体，但每类只提供了很少的有标签训练数据）只用监督学习算法来训练一个完整的系统显得有点天真和不被看好，因为模型参数的个数比训练样本的个数都要多很多。所以很多人觉得只有非常认真的训练或者人工挑选好的滤波器组才可以有好的识别性能，然后再考虑非线性算子的选择。其实，这些观点，都是wrong的。

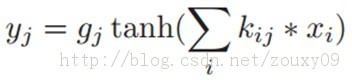
**二、模型架构**

         这部分讲述如何去构建一个分级的特征提取和分类系统。分级通过堆叠一个或者多个特征提取阶段，每个阶段包括一个滤波器组合层、非线性变换层和一个pooling层，pooling层通过组合（取平均或者最大的）局部邻域的滤波器响应，因而达到对微小变形的不变性。



**1、滤波器组层Filter Bank Layer-FCSG：**

        FCSG一般包括三部分：一组卷积滤波器（C）、再接一个sigmoid/tanh非线性变换函数（S），然后是一个可训练的增益系数（G）。分别对应下面的三个运算：



**2、校正层Rectification Layer-Rabs：**

       只是简单的一个取绝对值的操作（如果是tanh，则存在负的值，但在图像中负值是不表示的，而对于卷积来说，里面的都是绝对值越大，非线性函数输出的绝对值最大，实际意义是一样的。卷积是越相似，输出值越大）。除了绝对值算子外，我们还试过了其他的非线性算子，产生的效果差不多。

**3、局部对比度归一化层Local Contrast Normalization Layer-N：**

        该模块主要进行的是局部做减和做除归一化，它会迫使在特征map中的相邻特征进行局部竞争，还会迫使在不同特征maps的同一空间位置的特征进行竞争。在一个给定的位置进行减法归一化操作，实际上就是该位置的值减去邻域各像素的加权后的值，权值是为了区分与该位置距离不同影响不同，权值可以由一个高斯加权窗来确定。除法归一化实际上先计算每一个特征maps在同一个空间位置的邻域的加权和的值，然后取所有特征maps这个值的均值，然后每个特征map该位置的值被重新计算为该点的值除以max（那个均值，该点在该map的邻域的加权和的值）。分母表示的是在所有特征maps的同一个空间邻域的加权标准差。哦哦，实际上如果对于一个图像的话，就是均值和方差归一化，也就是特征归一化。这个实际上是由计算神经科学模型启发得到的。（这里自己有点理解，请见本文的第四节）

**4、平均池化和子采样层Average Pooling and Subsampling Layer -PA：**

       该层的作用是使得提取的特征对微小变形鲁棒，和视觉感知中的复杂细胞的角色差不多。采样窗口所有值取平均得到下采样层的值。

**5、最大值池化和子采样层Max-Pooling and Subsampling Layer -PM：**

       可以用任何一种对称的pooling操作实现对提取的特征的平移不变性。最大池与平均池相似，只是最大取代了平均。一般来说，池化窗口是不重叠的。

**三、实验与结论**

        该文做了很多实验来验证不同的模型架构的性能（组合上面的不同的层）。这里就不列举实验结果了，可以回原文查看。这里就直接回答一开始的那几个问题吧：

1）滤波器组后面接的non-linearities非线性算法是如何影响识别的准确性的？

        俺们的实验结论是，简单的矫正过的非线性算子会提高识别性能。原因可能有二。a）特征的对立polarity（也就是负值的特征）和目标的识别是无关的。b）在采用平均池化的时候，矫正层的添加会消去邻近的滤波器输出之间的cancellations，因为如果没有矫正，平均下采样只会传播输入的噪声。另外，局部归一化层也会增加性能，因为它可以使监督学习算法更快，也许是因为所有的变量都具有相似的方差了（与其他白化和去相关的方法的优点一样），这样会加快收敛速度。

2）通过监督或者非监督方式学习到的滤波器组是否比随机的滤波器组或者人工指定的滤波器要好？

         实验结果很惊喜，在两级系统中采样随机滤波器组在Caltech-101中居然达到了挺高的62.9%的识别率，但在NORB数据库中就显得有点低调了，可能这种情况只会在训练样本集较少的时候才出现。另外，非监督预训练接个监督微调会有最好的效果，尽管比单纯的全部使用监督会差点。

3）与仅有一级的特征提取对比，两级的特征提取是否还有其他优点？

         实验证明，两级比一级好。我们这里，两级系统的性能和最好的一级系统的性能：SIFT特征+PMK-SVM分类器相媲美，也许PM Kernel还隐藏着实现了我们的第二级特征提取的功能。

**四、关于local contract normalization**

        这里对这个东西再啰嗦一下。local contract normalization 这个归一化包括两个部分：局部做减和局部做除（local subtractive and divisive normalizations）。我的理解：自然图像存在低阶和高阶的统计特征，低阶（例如二阶）的统计特征是满足高斯分布的，但高阶的统计特性是非高斯分布。图像中，空间上相邻的像素点有着很强的相关性。而对于PCA来说，因为它是对协方差矩阵操作，所以可以去掉输入图像的二阶相关性，但是却无法去掉高阶相关性。而有人证明了除以一个隐含的变量就可以去除高阶相关性。你可以理解为一张图像x的像素值是一个随机变量，它由两个独立的随机变量相乘得到，分别是二阶量和高阶量相乘，二阶量的相关性可以由PCA去掉，然后高阶量（这个是隐含的，需要通过MAP最大后验估计等方法估计出来）直接用x除掉就好了。

        有论文的操作是这样：

        对输入图像的每一个像素，我们计算其邻域（例如3x3窗口）的均值，然后每个像素先减去这个均值，再除以这个邻域窗口（例如3x3窗口）拉成的9维向量的欧几里德范数（如果这个范数大于1的时候才除：这个约束是为了保证归一化只作用于减少响应（除以大于1的数值变小），而不会加强响应（除以小于1的数值变大））。也有论文在计算均值和范数的时候，都加入了距离的影响，也就是距离离该窗口中心越远，影响越小，例如加个高斯权重窗口（空间上相邻的像素点的相关性随着距离变大而变小）。

       其实在这里，很多自己也还不清楚，所以上面的不一定正确，仅供参考。还望明白的人也指点一下。谢谢。

关于local contract normalization可以参考以下两篇文章：

S. Lyu 等：Nonlinear image representation using divisive normalization.

N. Pinto等： Why is real-world visual object recognition hard?

# [Deep Learning论文笔记之（七）深度网络高层特征可视化](http://blog.csdn.net/zouxy09/article/details/10012747)

自己平时看了一些论文，但老感觉看完过后就会慢慢的淡忘，某一天重新拾起来的时候又好像没有看过一样。所以想习惯地把一些感觉有用的论文中的知识点总结整理一下，一方面在整理过程中，自己的理解也会更深，另一方面也方便未来自己的勘察。更好的还可以放到博客上面与大家交流。因为基础有限，所以对论文的一些理解可能不太正确，还望大家不吝指正交流，谢谢。

        本文的论文来自：

Dumitru Erhan, Aaron Courville, Yoshua Bengio, and Pascal Vincent. [Visualizing Higher Layer Features of a Deep Network](http://igva2012.wikispaces.asu.edu/file/view/Erhan+2009+Visualizing+higher+layer+features+of+a+deep+network.pdf). Spotlight presentation and poster at the ICML 2009 Workshop on Learning Feature Hierarchies, Montréal, Canada

         下面是自己对其中的一些知识点的理解：

**《Visualizing Higher-Layer Features of a Deep Network》**

         Deep Learning很吸引人，也很玄乎的一个点就是大家都说它可以提取到分级的逐层抽象的特征。但对我们来说，总是耳听为虚，眼见为实。所以，每当我们训练完一个深度模型后，我们还特别想把这个深度模型学到的东西给可视化出来，好弄明白它到底学到了什么东西，是不是有意义的，是不是像传说中的那样神奇。那怎样有意义的可视化呢？对了，我们用deep net来做什么的了？来提取特征的。那它提取什么特征呢？如果它像我们所说的提取的是底层到高层的特征，例如边缘，到形状，到目标等等，那就证明我们的目的是达到的。

       另外，在对深度模型定量的分析上，我们还需要一个定性的分析方法去比较不同的深度架构学习到的特征。本文的目的就是寻找深度模型所提取到的高级特征的比较好的定性解释。我们通过在几个视觉数据库中训练堆叠降噪自动编码器和DBN深信网络，并比较几种不同的高级特征可视化的方法。虽然这些特征的显示是在单元级别上面的，也许有违于直观理解，但它很容易实现，而且在不同方法上面得到的结果也是一致的。我们希望这些方法可以让研究者更清楚的理解深度学习是如何工作和为什么工作的。本文中，介绍三种可视化的方法：激活最大化、采样和线性组合法。

**一、概述**

       一些深度架构（例如DBNs）与generative procedure生成过程存在密切的联系，所以我们可以利用这个生成过程来瞥见一个单独的隐层神经元所表示的是什么。在这里，我们研究其中一个这样的采样方法。然而，一方面有时候我们很难去获得可以完全覆盖波尔兹曼或者RBM分布的样本，另一方面这种基于采样的可视化方法没办法运用到其他基于自动编码器的深度架构模型或者在每个层嵌入保留相似性的半监督学习模型上。

       一个典型的对深度架构第一层所提取的特征的定性分析方法是通过观察由模型所学习到的这些滤波器。这些滤波器是输入层到第一层的权值矩阵的权值。他们是由输入空间来表示。这样就非常方便了，因为输入是图像或者小波，他们是可以被可视化的。

       一般来说，当在数字的数据集中训练的时候，这些滤波器可以被可视化为一些不同的数字的笔画的检测器。当在自然图像中训练时，这些滤波器就相当于不同的边缘检测器了（小波滤波器）。

       本文的目标是研究一种可以可视化深度架构中任意层中的任意神经元所计算或者提取的特征的一种方法。为了达到这个目的，我们需要在输入空间（图像）中实现可视化，并且需要找到一种有效地计算方法去计算，然后使他具有通用性，也就是在不同的深度网络模型中都可以使用。在这里我们探究了几种方法。然后我们在两个数据集中对他们进行了定性的对比，研究他们之间的联系性。

       在实验过程中，一个非常让人惊喜的地方是，每一个隐层的节点对输入图像的响应，也就是输入空间的函数，居然是单峰的，也就是说，不管你随机地在什么地方初始化，最终都可以可靠地找到这个最大值，这对于迭代寻优来说是非常爽的，而且它可以将每个节点做了什么公开于天下。一览无余。

**二、模型**

      我们这里讨论两个模型，这两个模型都是在深度架构中比较常见的。第一个是DBNs，它是通过贪婪的堆叠多个RBM层得到的。我们先通过CD算法训练一个RBM，然后固定它，再将其输出做了另一个RBM的输入，去训练这一个隐层的RBM。这个过程可以通过不断重复，然后得到一个服从训练分布的非监督模型的深度架构。需要注意的一点是，它是一个数据的生成模型，可以很容易的通过一个已经训练好的模型去采集样本。

我们要讨论的第二个模型是降噪自动编码器，它是传统编码器的随机变种，它具有更强的能力。它不会学习到恒等函数（也就是h(x)=x，这个函数满足了零重构误差，但这是没有意义的）。它被强制去学习输入的本质表达。

它训练的关键是需要不断的提高生成模型的似然函数的下界。它比传统的编码机要牛逼，如果堆叠成一个深度监督架构的话，与RBMs的性能相当，甚至更牛。另一个在隐层单元比输入单元要多的情况下避免学习恒等函数的方式是对隐层code增加稀疏约束。

         我们在这里概括下Stacked Denoising Auto-Encoders的训练方法。对于一个输入x，我们给它添加随机的污染或者噪声，然后训练让降噪自动编码机学习重构原始的输入x。每个自动编码机的输出都是一个码矢h(x)。在这里和传统的神经网络一样，h(x) = sigmoid(b + W x)。这里我们用C(x)表示x的一个随机污染，我们让Ci(x) = xi或 0，换句话说，就是我们随机的在原来的x中挑选一个固定大小的子集，把他们设置为0。我们还可以添加椒盐噪声，随机的选择一定大小的子集，设置为Bernoulli(0.5)。

       在实际的图像中，对于特定像素i的像素输入xi和它的重构像素xii都可以看成该像素的伯努利概率：该位置的像素被涂成黑色的概率。我们通过交叉熵来比较该像素位置i的原始输入xi和它的重构像素xii的分布的相似性。然后需要对所有像素求和。另外，只有当输入和重构的值都在[0,1]这个范围的时候，伯努利分布才有意义。另外的选择就是选择高斯分布，这时候对应的就是均方误差规则了。

**三、Maximizing the activation 最大化激活值**

        第一个思想是很简单的：我们寻找使一个给定的隐层单元的激活值最大的输入模式。因为第一层的每一个节点的激活函数都是输入的线性函数，所以对第一层来说，它的输入模式和滤波器本身是成比例的。

        我们回顾下诺贝尔医学奖David Hubel 和Torsten Wiesel 的那个伟大的实验。他们发现了一种被称为“方向选择性细胞（Orientation Selective Cell）”的神经元细胞。当瞳孔发现了眼前的物体的边缘，而且这个边缘指向某个方向时，这种神经元细胞就会活跃。也就是说某个“特定方向神经细胞”只对这个特定方向的图像边缘存在激励或者兴奋。通俗点说就是如果我这个神经元是提取这个特征的，那么如果你这个图像满足这个特征（可以理解为和它很相似），那么神经元的输出就很大，会兴奋。（有资料表明，人的大脑高层会存在“祖母细胞”，这类细胞的某一个细胞只对特定一个目标兴奋，例如你大脑里面有个能记忆你女朋友的细胞，然后一旦你女朋友出现在你面前，你这个细胞就会兴奋，告诉大脑，啊，这个是我的女朋友！）我们如果了解过模板卷积，那么我们知道如果某个卷积模板与图像中模块越相似，那么响应就越大。相反，如果某个图像输入使得这个神经元输出激励值最大，那么我们就有理由相信，这个神经元就是提取了和这个输入差不多的特征。所以我们寻找可以使这个神经元输入最大的那个x就是我们可以可视化并且有意义的表达这个神经元学习到的特征了。

       用数学属于来表述就是，一旦完成网络训练后，参数W是确定的了，那么我们就可以寻找使得这个神经元最大化的激活值对应的x了，也就是：

http://img.blog.csdn.net/20130816232008734?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvem91eHkwOQ==/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/SouthEast

        但这个优化问题通常是一个非凸优化问题，也就是是存在很多局部最小值。最简单的方法就是通过梯度下降去寻找到一个局部最小值。这会出现两种场景：一是从不同的随机值初始化开始最后都迭代得到相同的最小值，二是得到两个或者更多的局部最小值。不管是哪种情况，该神经节点提取的特征都可以通过找到的一个或者多个最小值进行描述。如果有多个最小值，那么可以寻找使激活值最大的或者将所有的进行平均，或者把所有的都显示出来。

**四、Sampling from a unit of a Deep Belief Network 从DBN的一个节点中采样**

       我们这里用一个j层的Deep Belief Network来说明。这里层j和层j-1构成一个RBM，我们可以通过块Gibbs采样方法来对分布p(**h**j−1|**h**j) 和 p(**h**j|**h**j−1)进行连续采样（这里**h**j表示层j的所有的二值节点构成的向量）。在这个马尔科夫链中，我们限制一个节点**hij**为1，其他节点都是0 。然后在DBN中，我们从层j-1一直执行top-down采样一直到输入层。这样就会产生一个分布pj(**x**|**hij**=1)。也就是说我们用分布pj(**x**|**hij**=1)来描述hij，和第三部分相似，我们可以通过从这个分布生成或者采样足够多的样本来描述这个隐层节点或者通过计算期望E[**x**|**hij**=1]来概括这个信息。这个方法只有一个参数需要确定，也就是我们要采样多少个样本来估计这个期望值。

       在最大化激活值和计算期望E[**x**|**hij**=1] 两种方法之间是存在一个非常微妙的联系的。由条件期望的定义我们可以知道：

http://img.blog.csdn.net/20130816232029734?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvem91eHkwOQ==/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/SouthEast

        我们考虑一个极端的情况，就是这个分布全部集中在**x**+这一个点，这时候pj(**x**|**hij**=1)约等于δx+ (x)。所以其期望E[**x**|**hij**=1]= **x**+。

        事实上，我们观测到的是，尽管采样得到的样本或者它们的平均可能看起来很像训练样本，但是由激活值最大化获得的图像反而看起来更像图像的部分。所以后者可能更能准确的表达一个特定的节点做了什么。

**五、Linear combination of previous layers’ filters 上层滤波器的线性组合**

         Lee等（2008）在他们的论文中展示了一种可视化第二层隐层的节点的特征的方法。他们是基于一个神经元节点可以由和其强连接的上一层的滤波器的组合来描述的假设的。该层的某个节点的可视化可以通过上一层滤波器的线性加权得到，每个滤波器的权值就是该滤波器与这个节点之间的连接权值。

         他们用自然图像训练了一个具有激活值稀疏约束的DBNs，然后用这种方法显示它在第二层学习到的是一个角的检测器。Lee拓展了这个方法去可视化第三层学习到的东西：通过简单对第二层的滤波器进行加权，权值是第二层滤波器到第三层该节点的连接权值，而且选择的是最大的那个权值。

       这种方法是简单而有效的。但是它的一个缺点就是，对在每一层如何自动的选择合适的滤波器个数没有一个清晰的准则。另外，如果只是选择少有的几个与第一次连接最强的滤波器的话，我们有可能得到的是失望的毫无意思的混杂图像，这是因为上面这种方法本质上忽略了上一层其他没有被选择的滤波器。另一方面，这种方法还忽略了层间的非线性，这种非线性在模型中确是非常重要的一部分。

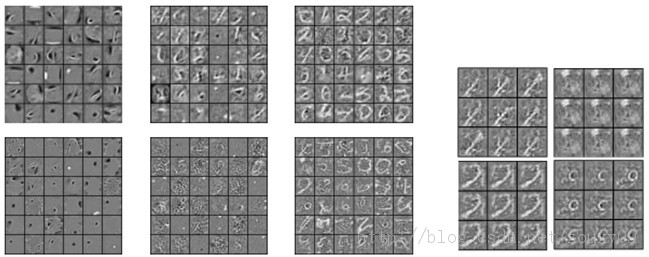
       但值得注意的是，实际上最大化一个节点的激活值的梯度更新方法与线性加权组合之前也存在微妙的联系。例如，对于第2层的第i个节点hi2= v’ sigmoid(W x)，这里v是该节点的权值，W是第一层的权值矩阵。然后∂hi2/∂x = v’diag(sigmoid(W x) ∗ (**1** −sigmoid(W x)))W。这里\*是逐元素相乘。diag是一个从一个向量创建一个对角矩阵的算子。**1**是个全一向量。如果第一层的节点没有饱和saturate，那么∂hi2/∂x就会大体的指向v’W的方向，可以用**v**i中的元素的绝对值的最大的那个来近似。（我也还没懂）

**六、实验**

**6.1、Data and setup**

我们在MINST手写体数据库和自然图像数据库中分别训练DBN和DSAE两种模型。然后再用三种方法来可视化其中一些层的一些节点所提取到的特征。

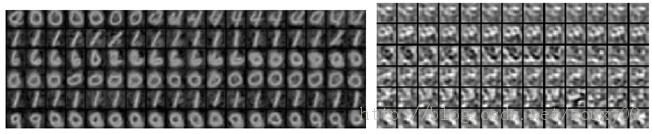
**6.2、Activation Maximization**



       在MNIST手写体数据库中用Activation maximization得到的可视化效果。左：分别是第一层（第一列）、第二层（第二列）和第三层（第三列）的36个节点所提取的特征，然后第一行是DBN训练得到的，第二行是SDAE训练得到的。右：对于DBN的第三层的一个节点，然后从9个随机初始化的值开始迭代，都可以得到同一个结果。

       一般来说，在第三层的激活函数应该是一个关于它的输入的高度非凸的函数，但是不知道是我们一直很幸运还是说恰好我们在MNIST或者自然图像中训练网络是一种很特殊的情况，我们惊喜的发现，这些节点的激活函数既然还是趋向于更加的“单峰化”。

**6.3、Sampling a unit**

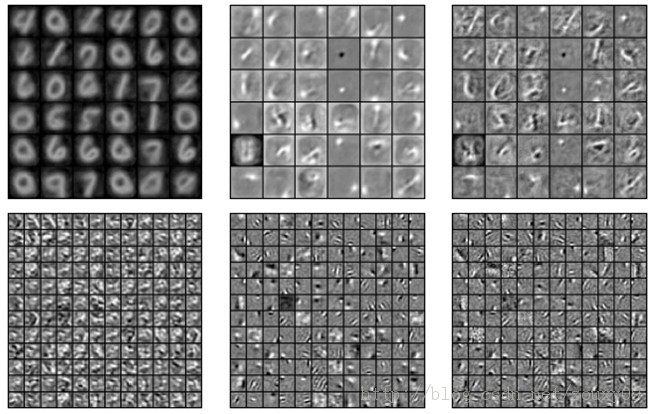


       左：从MNIST数据库中训练DBN得到的，然后采样得到第二层的其中6个节点的可视化效果。右：从自然图像中训练的。每一行的样本是每一个节点的分布采样得到的，然后每行的均值在第一列。

        值得注意的是，采样法和激活值最大化方法的结果不同，采样得到的（或者说分布生成的）样本更有可能是训练样本（数字或者patches）的潜在分布。激活值最大化方法是产生很多特征，然后要我们去决定哪些样本会匹配或者符合这些特征；采样方法是产生很多样本，然后由我们决定这些样本存在哪些共同的特征。从这个层面上面讲，这两种方法是互补的。

**6.4、Comparison of methods**

       在这里，我们对比可视化的三种方法：激活最大化、采样和线性组合法。



       我们这里展示了三种方法：左：采样；中：上次滤波器的线性组合；右：最大化节点激活值。在MINST数据库（top）和自然图像（down）上训练DBN模型。然后可视化第二层的36个节点。

       在这三种方法中，线性组合方法是以前提出来的，其他两种是我们以现有的知识吗，集当前的智慧得到的。

**附：**原论文中，提到了更多的内容，深入了解请参考原论文。另外，自己还没来得及去实现一些，等以后可以了再放上来和大家交流下。也希望大家已经实现了的也可以共享下，谢谢。

# [Deep Learning论文笔记之（八）Deep Learning最新综述](http://blog.csdn.net/zouxy09/article/details/10077055)

       自己平时看了一些论文，但老感觉看完过后就会慢慢的淡忘，某一天重新拾起来的时候又好像没有看过一样。所以想习惯地把一些感觉有用的论文中的知识点总结整理一下，一方面在整理过程中，自己的理解也会更深，另一方面也方便未来自己的勘察。更好的还可以放到博客上面与大家交流。因为基础有限，所以对论文的一些理解可能不太正确，还望大家不吝指正交流，谢谢。

         本文的论文来自：

Bengio, Y., Courville, A., & Vincent, P. (2012). [Representation Learning: A Review and New Perspectives](http://arxiv.org/pdf/1206.5538)。

         这是一篇Deep Learning比较新的综述。但是好长啊，读完了也好多不懂，之前边读边翻译了前面两节，先摆上来。后面有时间再更新后续的了。另外，因为水平有限，有些地方翻译和理解可能有错误，还望大家指正。谢谢。

       另外，对于Deep Learning这里有个reading-list，感觉很不错。大家可以参考里面的list来学习。

<http://deeplearning.net/reading-list/>

         下面是自己对其中的一些知识点的理解：

**《Representation Learning: A Review and New Perspectives》**

**摘要**

       机器学习算法的成功主要取决于数据的表达data representation。我们一般猜测，不同的表达会混淆或者隐藏或多或少的可以解释数据不同变化的因素。尽管特定的领域知识可以有助于设计或者选择数据的表达，但通过一般的先验知识来学习表达也是有效的。而且，人工智能AI的要求也迫使我们去寻找更强大的特征学习算法去实现这些先验知识。

       本文回顾非监督特征学习和深度学习领域的一些近期工作，包括概率模型的发展、自动编码机、流行学习和深度网络。通过这些分析，可以激发我们去思考一些长久以来尚未解决的问题，例如如何学习好的表达？如何选择适合的目标函数以便于计算表达？还有表达学习、密度估计和流行学习他们之间是否具有一定的几何联系？

**1、介绍**

       众所周知，机器学习方法的性能很大程度上取决于数据表达（或者特征）的选择。也正是因为这个原因，为了使得机器学习算法有效，我们一般需要在数据的预处理和变换中倾注大部分的心血。这种特征工程的工作非常重要，但它费时费力，属于劳动密集型产业。这种弊端揭露了目前的学习算法的缺点：在提取和组织数据的区分性信息中显得无能为力。特征工程是一种利用人的智慧和先验知识来弥补上述缺点的方法。为了拓展机器学习的适用范围，我们需要降低学习算法对特征工程的依赖性。这样，就可以更快的构建新的应用，更重要的是，在人工智能AI领域迈出了一大步。人工智能最基本的能力就是能理解这个世界（understand the world around us）。我们觉得，只有当它能学会如何辨别和解开在观测到的低级感知数据中隐含的解释性因素时才能达到这个目标。

       这篇文章主要讲述表达学习representation learning的，或者说学习一种数据的表达使得提取对构建分类器或者预测器有用的信息更加容易。以概率模型为例，一个好的表达总能捕捉观测输入数据的隐含解释性因素的后验概率分布。一个好的表达作为监督预测器的输入也是有用的。在表达学习的那么多不同的方法中，本文主要聚焦在深度学习方法：通过组合多个非线性变换，以得到更抽象和最终更有效的表达。这里，我们综述这个快速发展的领域，其中还会强调当前进展中的特定问题。我们认为，一些基本问题正在驱动该领域的研究。特别的，是什么导致一种表达优于另一种表达？我们应该怎样去计算它的表达，换句话来说就是，我们应该如何进行特征提取？还有就是为了学习好的表达，怎样的目标函数才是适合的？

**2、我们为什么要关心表达学习？**

         表达学习（亦被江湖称作深度学习或者特征学习）已经在机器学习社区开辟了自己的江山，成为学术界的一个新宠。在一些顶尖会议例如NIPS和ICML中都有了自己的正规军（研究它的workshops），今年（2013）还专门为它搞了一个新的会议，叫ICLR（International Conference on Learning Representations），可见它在学术界得到的宠爱招人红眼。尽管depth（深度）是这个神话的一个主要部分，但其他的先验也不能被忽视，因为有时候，先验知识会为表达的学习献上一臂之力，画上点睛之笔，更容易地学习更好的表达，这在下一章节中将会详细讨论。在表达学习有关的学术活动中最迅速的进展就是它在学术界和工业界都得到了经验性的显著性的成功。下面我们简单的聚焦几点。

**2.1、Speech Recognition and Signal Processing语音识别与信号处理**

         语音也是神经网络诞生时其最早的一个应用之一，例如卷积（或者时延）神经网络（Bengio在1993年的工作）。当然，在HMM在语音识别成功之后，神经网络也相对沉寂了不少。到现在，神经网络的复活、深度学习和表达学习的运用在语音识别领域可谓大展拳脚，重展雄风，在一些学术派和工业派人士（Dahlet al., 2010; Deng et al., 2010; Seide et al., 2011a; Mohamedet al., 2012; Dahl et al., 2012; Hinton et al., 2012）的努力下取得了突破性的成果，使得这些算法得到更大范围的应用，并且实现了产品化。例如，微软在2012年发布了它们的语音识别MAVIS (Microsoft Audio Video Indexing Service)系统的一个新版本，这个版本是基于深度学习的（Seide et al., 2011a）。对比现有的一直保持领先位置的高斯混合模型的声学建模方法，他们在四个主要的基准测试集中把错误率降低了30%左右（例如在RT03S数据库中从 27.4%的错误率降到18.5%）。在2012年，Dahl等人再次书学神话，他在一个小的大词汇量语音识别基准测试集中（Bing移动商业搜索数据库，语音长40小时）的错误率降到16%与23%之间。

         表达学习算法还被应用的音乐方面上，在四个基准测试集中，比当前领先的polyphonic transcription (Boulanger-Lewandowskiet al., 2012)在错误率上取得了5%到30%之间的提升。深度学习还赢得了MIREX (Music Information Retrieval)音乐信息检索竞赛。例如2011年的音频标注audio tagging上(Hamelet al., 2011)。

**2.2、Object Recognition目标识别**

         在2006年，深度学习的开始，主要聚焦在MNIST手写体图像分类问题上（Hinton et al.,2006; Bengioet al., 2007），它冲击了SVMs在这个数据集的霸主地位（1.4%的错误率）。最新的记录仍被深度网络占据着：Ciresanet al.(2012)声称他在这个任务的无约束版本（例如，使用卷积架构）的错误率是0.27%，为state-of-the-art。而Rifaiet al.(2011c)在MNIST的knowledge-free版本中保持着0.81%的错误率，为state-of-the-art。

         在最近几年，深度学习将其目光从数字识别移到自然图像的目标识别，而最新的突破是在ImageNet数据库中把领先的26.1%的错误率拉低到15.3% (Krizhevskyet al., 2012)。

**2.3、Natural Language Processing自然语言处理**

         除了语音识别，深度学习在自然语言处理中也有很多应用。symbolic 数据的分布式表达由Hinton在1986年引入，在2003年由Bengio等人在统计语言模型中得到第一次的发展，称为神经网络语言模型neural net language models (Bengio,2008)。它们都是基于学习一个关于每个单词的分布式表达，叫做word embedding。增加一个卷积架构，Collobertet al.(2011)开发了一个SENNA系统，它在语言建模、部分语音标记、chunking（节点识别）、语义角色标记和句法分解中共享表达。SENNA接近或者超于目前的在这些任务中的当前领先方法。但它比传统的预测器要简单和快速。学习word embeddings可以以某种方式与学习图像表达结合，这样就可以联系文本和图像。这个方法被成功运用到谷歌的图像搜索上，利用大量的数据来建立同一空间中图像与问题之间的映射(Weston et al.,2010)。在2012年，Srivastava等将其拓展到更深的多模表达。

         神经网络语言模型也被通过在隐层中增加recurrence来改进(Mikolovet al., 2011)。改进效果比当下领先的平滑n-gram语言模型不仅在复杂度上降低，还降低了语音识别的错误率（因为语言模型是语音识别系统的一个重要组成部分）。这个模型还被应用到统计机器翻译上面 (Schwenk et al., 2012; Leet al., 2013)，改进了复杂度和BLEU分数。递归自动编码机Recursive auto-encoders（产生recurrent网络）在全句释义检测full sentenceparaphrase detection上也达到了现有的领先水平，是以前技术的两倍F1分数(Socheret al., 2011a) 。表达学习还用到了单词歧义消除word sense disambiguation上(Bordeset al., 2012)，取得了准确率从67.8% 到 70.2%的提升。最后，它还被成功运用到sentimentanalysis (Glorotet al., 2011b; Socher et al., 2011b)上，并超越现有技术。

**2.4、Multi-Task and Transfer Learning, Domain Adaptation多任务和迁移学习，域自适应**

        迁移学习（传统的机器学习假设训练数据与测试数据服从相同的数据分布。如果我们有了大量的、在不同分布下的训练数据，完全丢弃这些数据也是非常浪费的。如何合理的利用这些数据就是迁移学习主要解决的问题。迁移学习可以从现有的数据中迁移知识，用来帮助将来的学习。迁移学习（Transfer Learning）的目标是将从一个环境中学到的知识用来帮助新环境中的学习任务。）是指一个学习算法可以利用不同学习任务之间的共性来共享统计的优点和在任务间迁移知识。如下面的讨论，我们假设表达学习算法具有这样的能力，因为它可以学习到能捕捉隐含因素的子集的表达，这个子集是对每个特定的任务相关的。如图1所示。这个假设被很多的经验性结果所验证，并且展现了表达学习在迁移学习场合中同样具有优异的能力。



        图1：表达学习发现了隐含的解释性因素（中间隐层红色的点）的示意图。一些解释了输入（半监督设置），一些解释了每个任务的目标。因为这些子集间会重叠，所以会贡献统计的优点，利于generalization泛化。

         给人印象深刻的是在2011年的两个迁移学习的挑战赛，都被表达学习算法夺魁。首先在由ICML2011一个workshop举办的Transfer Learning Challenge中，由无监督逐层预训练方法unsuper-vised layer-wise pre-training (Bengio, 2011; Mesnil et al.,2011)夺得。第二个挑战赛同年举办，被Goodfellow et al. (2011)夺得。在相关的domain adaptation方面，目标保持不变，但输入分布会改变(Glorot et al., 2011b; Chen et al., 2012)。在多任务学习方面multi-task learning，表达学习同样表现出了其独特的优越性（Krizhevskyet al.(2012); Collobertet al.(2011)），因为它可以在任务间共享因素。

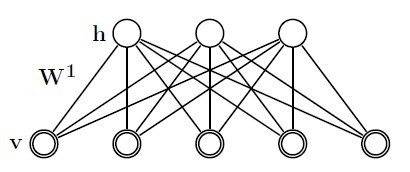
未完待续……

# [DeepLearning（深度学习）原理与实现（一）](http://blog.csdn.net/marvin521/article/details/8886971)

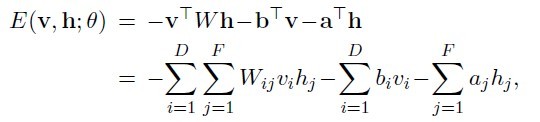
       经过三年的狂刷理论，觉得是时候停下来做些有用的东西了，因此决定开博把他们写下来，一是为了整理学过的理论，二是监督自己并和大家分享。先从DeepLearning谈起吧，因为这个有一定的实用性（大家口头传的“和钱靠的很近”大笑)，国内各个大牛也都谈了不少，我尽量从其他方面解释一下。

        DeepLearning算是多伦多大学Geoffery hinton教授第二春吧，第一春就是传统神经网络，由于传统的多层感知机很容易陷入局部最小，直接用反向传播算法（Back Propagation）求取的分类效果并不如意，原因一是特征是手工的，二就是局部最小问题。而DeepLearning引入了概率图模型里的生成模型，他可以直接自动的从训练集里提取所需要的特征，典型的模型为有限制玻尔兹曼机（Restricted Boltzmann Machines，简称RBM），自动提取的特征解决了手工特征考虑不周的因素，而且很好的初始化了神经网络权重，接着可以采用反向传播算法进行分类，实验得出了很好的效果。 因此DeepLearning被喻为下一代神经网络。今天的话题就来讨论下RBM：

       再说RBM之前，我们先提一下基于能量的模型(Engery based model),能量方法来源于热动力学，分子在高温中运动剧烈，能够克服局部约束（分子之间的一些物理约束，比如键值吸引力等），在逐步降到低温时，分子最终会排列出有规律的结构，此时也是低能量状态。受此启发，早期的模拟退火算法就是在高温中试图跳出局部最小。随机场作为物理模型之一，也引入了此方法。在马尔科夫随机场（MRF）中能量模型主要扮演着两个作用：一、全局解的度量（目标函数）；二、能量最小时的解（各种变量对应的配置）为目标解，能否把最优解嵌入到能量函数中至关重要，决定着我们具体问题求解的好坏。统计模式识别主要工作之一就是捕获变量之间的相关性，同样能量模型也要捕获变量之间的相关性，变量之间的相关程度决定了能量的高低。把变量的相关关系用图表示出来，并引入概率测度方式就够成了概率图模型的能量模型，其实实际中也可以不用概率表示，比如立体匹配中直接用两个像素点的像素差作为能量，所有像素对之间的能量和最小时的配置即为目标解。RBM作为一种概率图模型，引入概率就是为了方便采样，因为在CD（contrastive divergence）算法中采样部分扮演着模拟求解梯度的角色。  
  
  
先来看一下RBM模型定义（图一）和能量的定义（公式一）：

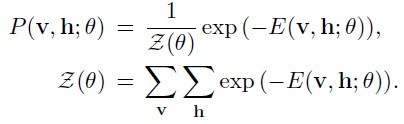


（图一）



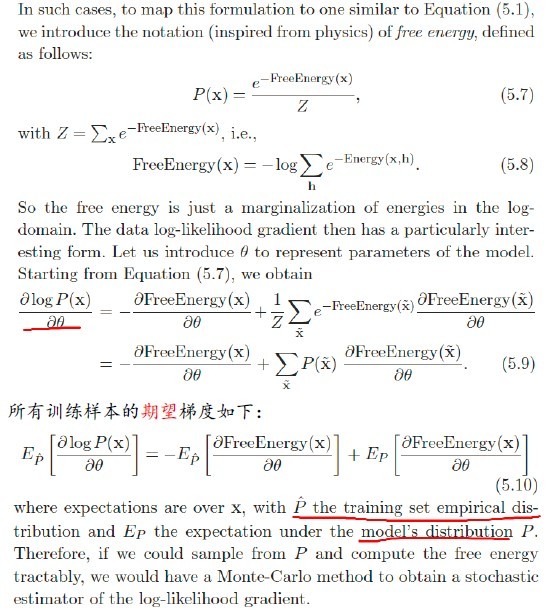
（公式一）

        其中v表示实际样本节点，以下叫可视节点，h表示隐藏节点数据，theta=｛W,b,a｝,W表示网络权重，b表示可视节点偏移，a表示隐藏节点偏移。这些参数是需要我们求解的，一般开始时我们随机初始化这些参数，对于16\*16的二值图像，可视节点有256个，每个节点可能的状态有两个（0，1）；隐藏节点的数目由自己制定，每个节点的状态也有两个（0，1）。当每个节点都有一个指定的状态时，此时的RBM有一个对应的能量。此时所有节点对应某个状态的组合成为一个配置。穷举所有配置对应的能量之和就构成了归一化常量Z（配分函数-partition function）。每个配置能量除以常量Z即为当前配置的概率。在一般计算时，为了避免数值计算问题，我们常常引入极家族函数(对数函数和指数函数)，那么模型概率定义为（公式二）：



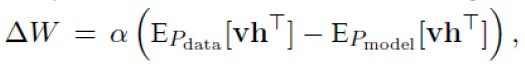
（公式二）

        由于RBM是一种生成模型(generative model),我们的目标使得概率p(x)最大时对应的参数theta,常规思路应该使用最大似然法，在求最大似然函数的极值时，我们需要用到梯度法，梯度推导如（图二）所示（大家不要被公式吓倒，CD算法不用它，=、=，只是证明求梯度有点难度）：



（图二）

公式虽然这么写，但是实际中计算归一化分割函数Z时不可取的，因为组合状态是NP-complete的。但是这个公式给出了Contrastive divergence算法的原型（公式三）：



（公式三）

        alpha是学习率，括号中第一项是训练数据集期望分布，第二项是模型期望分布，CD算法的核心是计算（或逼近）出这两项的数值，下面分别说出下面计算上式中两个期望分布，需要用到采样方法（我先给出步骤，然后解释，步骤是大家最关心的）：  
  
       第一项（数据相关期望项）：利用可视节点v1乘上权重，然后用sigmoid函数计算出隐藏节点h每个节点的概率，通过此概率对隐藏节点采样，得出隐藏节点状态h1，然后用能量公式计算出所有节点的期望能量(<v1\*h1>)作为第一项。

       第二项（模型期望项）：根据第一项中隐藏节点h1乘上权重，然后用sigmoid函数计算出可视节点v2的概率，在此通过采样得出v2 状态，同样用v2计算出h2的状态，用能量公式计算出所有节点的期望能量(<v2\*h2>)。套用上面公式，就可以求出deltaW,  
  
  
      有了类梯度，那么就可以做权重更新咯，迭代就行了，b,a类似求解。  
      这是用CD算法模拟梯度，关键在于使用什么采样方法，基于MCMC的各种采样方法(如Gibbs,MH,拒绝采样等)请酌情使用，深度理解采样方法有助于我们理解为什么CD可以模拟梯度，用hinton的话来说：这是个惊人的事实。我概括了这么一句话来给我们一个直观的理解：数据期望分布是我们整个数据的分布中心，可以看成最小点，模型期望是采样出来的，具有一定的随机性，分布在中心点周围，二者相减正好可以模拟梯度。其实求解参数theta的方法不止这一种，比如在Russlan的论文中也给出另外一种方法：用随机逼近过程来估计模型期望，用变分方法来估计数据相关期望等。碍于作者水平有限，就不讲了。   
  
     另外对DBN这个生成模型（generative model）对参数求导时，得出的结果也和CD公式相似，DBN的训练方法也可以用类似方法训练，具体可参考russlan的代码。  
  
      DBM训练方法其实就是层层通过对RBM分别训练。对于russlan论文中提到的其他衍生应用和技巧大家可以根据自己所需详细阅读。比如估计分割函数（Z）是图模型里一个重要的基石部分；autoencoders的神奇作用其实是模拟了神经科学里神经分布式描述，相当于找到了低维流形；深度网络的微调部分(fine-tune),其实通过上述方法做了无监督预训练后，结合标签部分用BP等方法做了权重微更新，值得一提的识别可以用生成模型（generative model），也可以是判别模型(discriminative model),针对具体应用大家可以试着采用不同方法。  
     我想我要说的基本说完了，整个深度学习除了需要概率图模型和神经网络基础支撑外，也用到了大量的trick。  
     哦，对了，hinton也说过，神经网络几乎可以模拟现在绝大多数的机器学习方法，包括SVM,Adaboost,当然也包括概率图模型的那一套（其中有结构化学习），这有点把图模型收编的味道，作者视野有限，不敢不认同，也不敢认同。就讲到这里吧，作者写的轻浮一些，公式都是从别人paper上扣下来的，  
而且作者水平有限，可能有些会说错,希望大家多多指正。

       最后附加一点：概率图模型的能量和概率之间的关系可以参阅[gibbs measure](http://en.wikipedia.org/wiki/Gibbs_measure)和[Hammersley–Clifford theorem](http://en.wikipedia.org/wiki/Hammersley%E2%80%93Clifford_theorem)，他们俩在能量模型到概率之间的转换上扮演着很重要的角色。还有为什么最大化P(x)或者p(v)可以使得能量最小？这个问题一开始我没搞明白，后来一个做事严谨网友（北流浪子）的花了好长时间找到答案，并且无私的与俺分享：对（图二）中的公式（5.7）两边同时取对数得到：lnp(v)=-FreeEnergy(v)-lnZ,这样当p(v)最大时，自由能FreeEnergy(v)最小，而自由能最小决定了系统达到统计平衡，详细可以参考：[Gibbs free energy](http://en.wikipedia.org/wiki/Gibbs_free_energy)。

参考文献：

         [1]  Learning Deep Generative models.Ruslan Salakhutdinov

         [2]  Learning Deep Architectures for AI.Yoshua Bengio

         [3] A Tutorial on Energy-Based Models. yann lecun

转载请注明出处：<http://blog.csdn.net/cuoqu/article/details/8886971>

# [DeepLearning（深度学习）原理与实现（二）](http://blog.csdn.net/marvin521/article/details/8887882)

下面贴出RBM C++版本的代码，一些大牛写的，结合上篇博文来加深大家对RBM理论的理解。。。

RBM类定义声明：

**[cpp]** [view plaincopy](http://blog.csdn.net/marvin521/article/details/8887882)

1. **class** RBM {
3. **public**:
4. **int** N;
5. **int** n\_visible;
6. **int** n\_hidden;
7. **double** \*\*W;
8. **double** \*hbias;
9. **double** \*vbias;
10. RBM(**int**, **int**, **int**, **double**\*\*, **double**\*, **double**\*);
11. ~RBM();
12. **void** contrastive\_divergence(**int**\*, **double**, **int**);
13. **void** sample\_h\_given\_v(**int**\*, **double**\*, **int**\*);
14. **void** sample\_v\_given\_h(**int**\*, **double**\*, **int**\*);
15. **double** propup(**int**\*, **double**\*, **double**);
16. **double** propdown(**int**\*, **int**, **double**);
17. **void** gibbs\_hvh(**int**\*, **double**\*, **int**\*, **double**\*, **int**\*);
18. **void** reconstruct(**int**\*, **double**\*);
19. };

从上面声明中可以很直观的看出和上篇文章公式符号正好完美对应。下面是代码实现部分：

**[cpp]** [view plaincopy](http://blog.csdn.net/marvin521/article/details/8887882)



**[cpp]** [view plaincopy](http://blog.csdn.net/marvin521/article/details/8887882)

1. #include <iostream>
2. #include <math.h>
3. #include "RBM.h"
4. **using** **namespace** std;
6. **double** uniform(**double** min, **double** max) {
7. **return** rand() / (RAND\_MAX + 1.0) \* (max - min) + min;
8. }
10. **int** binomial(**int** n, **double** p) {
11. **if**(p < 0 || p > 1) **return** 0;
13. **int** c = 0;
14. **double** r;
16. **for**(**int** i=0; i<n; i++) {
17. r = rand() / (RAND\_MAX + 1.0);
18. **if** (r < p) c++;
19. }
21. **return** c;
22. }
24. **double** sigmoid(**double** x) {
25. **return** 1.0 / (1.0 + exp(-x));
26. }

29. RBM::RBM(**int** size, **int** n\_v, **int** n\_h, **double** \*\*w, **double** \*hb, **double** \*vb) {
30. N = size;
31. n\_visible = n\_v;
32. n\_hidden = n\_h;
34. **if**(w == NULL) {
35. W = **new** **double**\*[n\_hidden];
36. **for**(**int** i=0; i<n\_hidden; i++) W[i] = **new** **double**[n\_visible];
37. **double** a = 1.0 / n\_visible;
39. **for**(**int** i=0; i<n\_hidden; i++) {
40. **for**(**int** j=0; j<n\_visible; j++) {
41. W[i][j] = uniform(-a, a);
42. }
43. }
44. } **else** {
45. W = w;
46. }
48. **if**(hb == NULL) {
49. hbias = **new** **double**[n\_hidden];
50. **for**(**int** i=0; i<n\_hidden; i++) hbias[i] = 0;
51. } **else** {
52. hbias = hb;
53. }
55. **if**(vb == NULL) {
56. vbias = **new** **double**[n\_visible];
57. **for**(**int** i=0; i<n\_visible; i++) vbias[i] = 0;
58. } **else** {
59. vbias = vb;
60. }
61. }
63. RBM::~RBM() {
64. **for**(**int** i=0; i<n\_hidden; i++) **delete**[] W[i];
65. **delete**[] W;
66. **delete**[] hbias;
67. **delete**[] vbias;
68. }

71. **void** RBM::contrastive\_divergence(**int** \*input, **double** lr, **int** k) {
72. **double** \*ph\_mean = **new** **double**[n\_hidden];
73. **int** \*ph\_sample = **new** **int**[n\_hidden];
74. **double** \*nv\_means = **new** **double**[n\_visible];
75. **int** \*nv\_samples = **new** **int**[n\_visible];
76. **double** \*nh\_means = **new** **double**[n\_hidden];
77. **int** \*nh\_samples = **new** **int**[n\_hidden];
79. /\* CD-k \*/
80. sample\_h\_given\_v(input, ph\_mean, ph\_sample);
82. **for**(**int** step=0; step<k; step++) {
83. **if**(step == 0) {
84. gibbs\_hvh(ph\_sample, nv\_means, nv\_samples, nh\_means, nh\_samples);
85. } **else** {
86. gibbs\_hvh(nh\_samples, nv\_means, nv\_samples, nh\_means, nh\_samples);
87. }
88. }
90. **for**(**int** i=0; i<n\_hidden; i++) {
91. **for**(**int** j=0; j<n\_visible; j++) {
92. W[i][j] += lr \* (ph\_sample[i] \* input[j] - nh\_means[i] \* nv\_samples[j]) / N;
93. }
94. hbias[i] += lr \* (ph\_sample[i] - nh\_means[i]) / N;
95. }
97. **for**(**int** i=0; i<n\_visible; i++) {
98. vbias[i] += lr \* (input[i] - nv\_samples[i]) / N;
99. }
101. **delete**[] ph\_mean;
102. **delete**[] ph\_sample;
103. **delete**[] nv\_means;
104. **delete**[] nv\_samples;
105. **delete**[] nh\_means;
106. **delete**[] nh\_samples;
107. }
109. **void** RBM::sample\_h\_given\_v(**int** \*v0\_sample, **double** \*mean, **int** \*sample) {
110. **for**(**int** i=0; i<n\_hidden; i++) {
111. mean[i] = propup(v0\_sample, W[i], hbias[i]);
112. sample[i] = binomial(1, mean[i]);
113. }
114. }
116. **void** RBM::sample\_v\_given\_h(**int** \*h0\_sample, **double** \*mean, **int** \*sample) {
117. **for**(**int** i=0; i<n\_visible; i++) {
118. mean[i] = propdown(h0\_sample, i, vbias[i]);
119. sample[i] = binomial(1, mean[i]);
120. }
121. }
123. **double** RBM::propup(**int** \*v, **double** \*w, **double** b) {
124. **double** pre\_sigmoid\_activation = 0.0;
125. **for**(**int** j=0; j<n\_visible; j++) {
126. pre\_sigmoid\_activation += w[j] \* v[j];
127. }
128. pre\_sigmoid\_activation += b;
129. **return** sigmoid(pre\_sigmoid\_activation);
130. }
132. **double** RBM::propdown(**int** \*h, **int** i, **double** b) {
133. **double** pre\_sigmoid\_activation = 0.0;
134. **for**(**int** j=0; j<n\_hidden; j++) {
135. pre\_sigmoid\_activation += W[j][i] \* h[j];
136. }
137. pre\_sigmoid\_activation += b;
138. **return** sigmoid(pre\_sigmoid\_activation);
139. }
141. **void** RBM::gibbs\_hvh(**int** \*h0\_sample, **double** \*nv\_means, **int** \*nv\_samples, \
142. **double** \*nh\_means, **int** \*nh\_samples) {
143. sample\_v\_given\_h(h0\_sample, nv\_means, nv\_samples);
144. sample\_h\_given\_v(nv\_samples, nh\_means, nh\_samples);
145. }
147. **void** RBM::reconstruct(**int** \*v, **double** \*reconstructed\_v) {
148. **double** \*h = **new** **double**[n\_hidden];
149. **double** pre\_sigmoid\_activation;
151. **for**(**int** i=0; i<n\_hidden; i++) {
152. h[i] = propup(v, W[i], hbias[i]);
153. }
155. **for**(**int** i=0; i<n\_visible; i++) {
156. pre\_sigmoid\_activation = 0.0;
157. **for**(**int** j=0; j<n\_hidden; j++) {
158. pre\_sigmoid\_activation += W[j][i] \* h[j];
159. }
160. pre\_sigmoid\_activation += vbias[i];
162. reconstructed\_v[i] = sigmoid(pre\_sigmoid\_activation);
163. }
165. **delete**[] h;
166. }

169. **void** test\_rbm() {
170. srand(0);
172. **double** learning\_rate = 0.1;
173. **int** training\_epochs = 1000;
174. **int** k = 1;
176. **int** train\_N = 6;
177. **int** test\_N = 2;
178. **int** n\_visible = 6;
179. **int** n\_hidden = 3;
181. // training data
182. **int** train\_X[6][6] = {
183. {1, 1, 1, 0, 0, 0},
184. {1, 0, 1, 0, 0, 0},
185. {1, 1, 1, 0, 0, 0},
186. {0, 0, 1, 1, 1, 0},
187. {0, 0, 1, 0, 1, 0},
188. {0, 0, 1, 1, 1, 0}
189. };

192. // construct RBM
193. RBM rbm(train\_N, n\_visible, n\_hidden, NULL, NULL, NULL);
195. // train
196. **for**(**int** epoch=0; epoch<training\_epochs; epoch++) {
197. **for**(**int** i=0; i<train\_N; i++) {
198. rbm.contrastive\_divergence(train\_X[i], learning\_rate, k);
199. }
200. }
202. // test data
203. **int** test\_X[2][6] = {
204. {1, 1, 0, 0, 0, 0},
205. {0, 0, 0, 1, 1, 0}
206. };
207. **double** reconstructed\_X[2][6];

210. // test
211. **for**(**int** i=0; i<test\_N; i++) {
212. rbm.reconstruct(test\_X[i], reconstructed\_X[i]);
213. **for**(**int** j=0; j<n\_visible; j++) {
214. printf("%.5f ", reconstructed\_X[i][j]);
215. }
216. cout << endl;
217. }
219. }


223. **int** main() {
224. test\_rbm();
225. **return** 0;
226. }

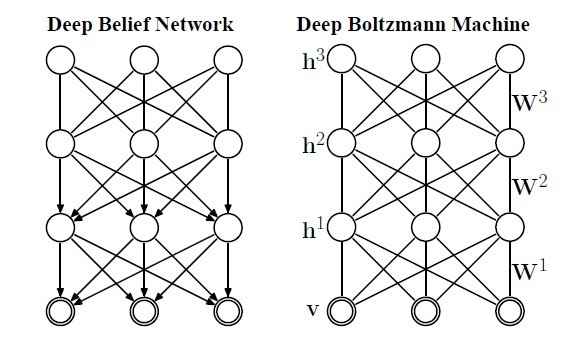
干脆把运行结果也贴出来，给那些终极极品思考者提供一些方便偷笑

**0.98472  0.67248  0.99120  0.01000  0.01311  0.01020  
0.01021  0.00720  0.99525  0.65553  0.98403  0.00497**

**转载请注明出处：**[**http://blog.csdn.net/cuoqu/article/details/8887882**](http://blog.csdn.net/cuoqu/article/details/8887882)

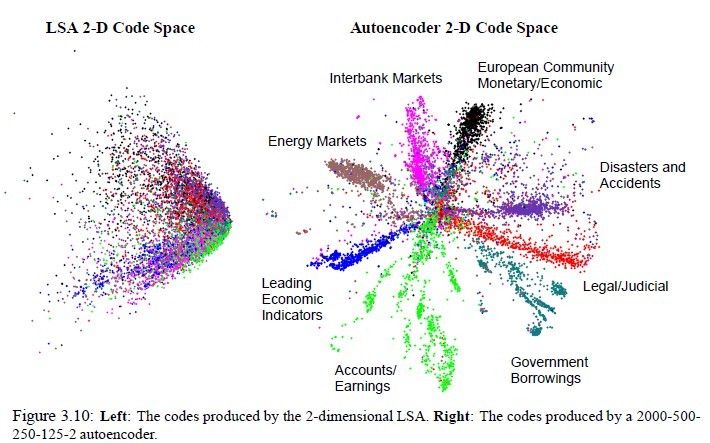
# [DeepLearning（深度学习）原理与实现（三）](http://blog.csdn.net/marvin521/article/details/8896636)

         考虑到大家有可能对深度学习的识别有点模糊，因此决定写一个短博客，简单介绍下如何识别，结合本系列的第一篇博文提到的深度学习之所以叫深度，其中之一的原因是多层RBM模仿了人脑多层神经元对输入数据进行层层预处理(值得一提的是并不是每层都是RBM，DBN就是个例外），即深层次的数据拟合，多个RBM连接起来构成DBM（deep boltzmann machines)，每层RBM的节点数自己指定，这需要一些经验,DBN和DBM在训练上没有区别，都是逐层使用CD算法训练，也叫贪心预训练算法，DBN和DBM的区别如（图一）所示。



图一

          对于二者都使用同一个算法来训练，看起来毫无区别，但是DBM有一个优势，由于RBM是无向的，这就决定了无论给定可视节点还是隐藏节点，各个节点都是独立的，可由图模型的马尔科夫性看出。作为无向图的DBM天生具有一些优秀的基因，比如当人看到一个外观性质，知道它是什么物体，同样你告诉他物体名字，他可以知道物体的外观应该是什么样子。这种互相推理的关系正好可以用无向图来表示。这种优势也顺理成章的延伸出了autoencoder（大家所谓的自编码神经网络）和栈式神经网络，最终输出的少量节点是可以推理（重建）出原来样本，也起到了降维的作用，无形中也找到了特征（编码），autoencoder的效果如图二所示。但是DBN中有些层是有向的，就不具有这种优势。



图二

           二者逐层预训练后，结合样本标签，使用BP算法进行权重微调，说白了就是在预训练后的权重基础上使用BP算法进行训练，这样得出的权重更好些。。。

下面贴出部分DBN代码，大家可以看出总体思路是按照构建DBN网络（刚构建后的每层的权重是随机生成的，从代码也能看出），贪心层层预训练，权重微调，预测（识别）这个步骤来的。另外代码中softmax其实是多变量的逻辑回归函数，注意我发的下面的代码中权重微调使用的是逻辑回归，不是BP：

**[cpp]** [view plaincopy](http://blog.csdn.net/marvin521/article/details/8896636)

1. #include <iostream>
2. #include <math.h>
3. #include "HiddenLayer.h"
4. #include "RBM.h"
5. #include "LogisticRegression.h"
6. #include "DBN.h"
7. **using** **namespace** std;

10. **double** uniform(**double** min, **double** max) {
11. **return** rand() / (RAND\_MAX + 1.0) \* (max - min) + min;
12. }
14. **int** binomial(**int** n, **double** p) {
15. **if**(p < 0 || p > 1) **return** 0;
17. **int** c = 0;
18. **double** r;
20. **for**(**int** i=0; i<n; i++) {
21. r = rand() / (RAND\_MAX + 1.0);
22. **if** (r < p) c++;
23. }
25. **return** c;
26. }
28. **double** sigmoid(**double** x) {
29. **return** 1.0 / (1.0 + exp(-x));
30. }

33. // DBN
34. DBN::DBN(**int** size, **int** n\_i, **int** \*hls, **int** n\_o, **int** n\_l) {
35. **int** input\_size;
37. N = size;
38. n\_ins = n\_i;
39. hidden\_layer\_sizes = hls;
40. n\_outs = n\_o;
41. n\_layers = n\_l;
43. sigmoid\_layers = **new** HiddenLayer\*[n\_layers];
44. rbm\_layers = **new** RBM\*[n\_layers];
46. // construct multi-layer
47. **for**(**int** i=0; i<n\_layers; i++) {
48. **if**(i == 0) {
49. input\_size = n\_ins;
50. } **else** {
51. input\_size = hidden\_layer\_sizes[i-1];
52. }
54. // construct sigmoid\_layer
55. sigmoid\_layers[i] = **new** HiddenLayer(N, input\_size, hidden\_layer\_sizes[i], NULL, NULL);
57. // construct rbm\_layer
58. rbm\_layers[i] = **new** RBM(N, input\_size, hidden\_layer\_sizes[i],\
59. sigmoid\_layers[i]->W, sigmoid\_layers[i]->b, NULL);
60. }
62. // layer for output using LogisticRegression
63. log\_layer = **new** LogisticRegression(N, hidden\_layer\_sizes[n\_layers-1], n\_outs);
64. }
66. DBN::~DBN() {
67. **delete** log\_layer;
69. **for**(**int** i=0; i<n\_layers; i++) {
70. **delete** sigmoid\_layers[i];
71. **delete** rbm\_layers[i];
72. }
73. **delete**[] sigmoid\_layers;
74. **delete**[] rbm\_layers;
75. }

78. **void** DBN::pretrain(**int** \*input, **double** lr, **int** k, **int** epochs) {
79. **int** \*layer\_input;
80. **int** prev\_layer\_input\_size;
81. **int** \*prev\_layer\_input;
83. **int** \*train\_X = **new** **int**[n\_ins];
85. **for**(**int** i=0; i<n\_layers; i++) {  // layer-wise
87. **for**(**int** epoch=0; epoch<epochs; epoch++) {  // training epochs
89. **for**(**int** n=0; n<N; n++) { // input x1...xN
90. // initial input
91. **for**(**int** m=0; m<n\_ins; m++) train\_X[m] = input[n \* n\_ins + m];
93. // layer input
94. **for**(**int** l=0; l<=i; l++) {
96. **if**(l == 0) {
97. layer\_input = **new** **int**[n\_ins];
98. **for**(**int** j=0; j<n\_ins; j++) layer\_input[j] = train\_X[j];
99. } **else** {
100. **if**(l == 1) prev\_layer\_input\_size = n\_ins;
101. **else** prev\_layer\_input\_size = hidden\_layer\_sizes[l-2];
103. prev\_layer\_input = **new** **int**[prev\_layer\_input\_size];
104. **for**(**int** j=0; j<prev\_layer\_input\_size; j++) prev\_layer\_input[j] = layer\_input[j];
105. **delete**[] layer\_input;
107. layer\_input = **new** **int**[hidden\_layer\_sizes[l-1]];
109. sigmoid\_layers[l-1]->sample\_h\_given\_v(prev\_layer\_input, layer\_input);
110. **delete**[] prev\_layer\_input;
111. }
112. }
114. rbm\_layers[i]->contrastive\_divergence(layer\_input, lr, k);
115. }
117. }
118. }
120. **delete**[] train\_X;
121. **delete**[] layer\_input;
122. }
124. **void** DBN::finetune(**int** \*input, **int** \*label, **double** lr, **int** epochs) {
125. **int** \*layer\_input;
126. // int prev\_layer\_input\_size;
127. **int** \*prev\_layer\_input;
129. **int** \*train\_X = **new** **int**[n\_ins];
130. **int** \*train\_Y = **new** **int**[n\_outs];
132. **for**(**int** epoch=0; epoch<epochs; epoch++) {
133. **for**(**int** n=0; n<N; n++) { // input x1...xN
134. // initial input
135. **for**(**int** m=0; m<n\_ins; m++)  train\_X[m] = input[n \* n\_ins + m];
136. **for**(**int** m=0; m<n\_outs; m++) train\_Y[m] = label[n \* n\_outs + m];
138. // layer input
139. **for**(**int** i=0; i<n\_layers; i++) {
140. **if**(i == 0) {
141. prev\_layer\_input = **new** **int**[n\_ins];
142. **for**(**int** j=0; j<n\_ins; j++) prev\_layer\_input[j] = train\_X[j];
143. } **else** {
144. prev\_layer\_input = **new** **int**[hidden\_layer\_sizes[i-1]];
145. **for**(**int** j=0; j<hidden\_layer\_sizes[i-1]; j++) prev\_layer\_input[j] = layer\_input[j];
146. **delete**[] layer\_input;
147. }

150. layer\_input = **new** **int**[hidden\_layer\_sizes[i]];
151. sigmoid\_layers[i]->sample\_h\_given\_v(prev\_layer\_input, layer\_input);
152. **delete**[] prev\_layer\_input;
153. }
155. log\_layer->train(layer\_input, train\_Y, lr);
156. }
157. // lr \*= 0.95;
158. }
160. **delete**[] layer\_input;
161. **delete**[] train\_X;
162. **delete**[] train\_Y;
163. }
165. **void** DBN::predict(**int** \*x, **double** \*y) {
166. **double** \*layer\_input;
167. // int prev\_layer\_input\_size;
168. **double** \*prev\_layer\_input;
170. **double** linear\_output;
172. prev\_layer\_input = **new** **double**[n\_ins];
173. **for**(**int** j=0; j<n\_ins; j++) prev\_layer\_input[j] = x[j];
175. // layer activation
176. **for**(**int** i=0; i<n\_layers; i++) {
177. layer\_input = **new** **double**[sigmoid\_layers[i]->n\_out];
179. **for**(**int** k=0; k<sigmoid\_layers[i]->n\_out; k++) {
180. // linear\_output = 0.0;  //原代码中删除此句
182. **for**(**int** j=0; j<sigmoid\_layers[i]->n\_in; j++) {
183. linear\_output = 0.0;   //原代码中添加此句
184. linear\_output += sigmoid\_layers[i]->W[k][j] \* prev\_layer\_input[j];
185. }
186. linear\_output += sigmoid\_layers[i]->b[k];
187. layer\_input[k] = sigmoid(linear\_output);
188. }
189. **delete**[] prev\_layer\_input;
191. **if**(i < n\_layers-1) {
192. prev\_layer\_input = **new** **double**[sigmoid\_layers[i]->n\_out];
193. **for**(**int** j=0; j<sigmoid\_layers[i]->n\_out; j++) prev\_layer\_input[j] = layer\_input[j];
194. **delete**[] layer\_input;
195. }
196. }
198. **for**(**int** i=0; i<log\_layer->n\_out; i++) {
199. y[i] = 0;
200. **for**(**int** j=0; j<log\_layer->n\_in; j++) {
201. y[i] += log\_layer->W[i][j] \* layer\_input[j];
202. }
203. y[i] += log\_layer->b[i];
204. }
206. log\_layer->softmax(y);

209. **delete**[] layer\_input;
210. }

213. // HiddenLayer
214. HiddenLayer::HiddenLayer(**int** size, **int** in, **int** out, **double** \*\*w, **double** \*bp) {
215. N = size;
216. n\_in = in;
217. n\_out = out;
219. **if**(w == NULL) {
220. W = **new** **double**\*[n\_out];
221. **for**(**int** i=0; i<n\_out; i++) W[i] = **new** **double**[n\_in];
222. **double** a = 1.0 / n\_in;
224. **for**(**int** i=0; i<n\_out; i++) {
225. **for**(**int** j=0; j<n\_in; j++) {
226. W[i][j] = uniform(-a, a);
227. }
228. }
229. } **else** {
230. W = w;
231. }
233. **if**(bp == NULL) {
234. b = **new** **double**[n\_out];
235. } **else** {
236. b = bp;
237. }
238. }
240. HiddenLayer::~HiddenLayer() {
241. **for**(**int** i=0; i<n\_out; i++) **delete** W[i];
242. **delete**[] W;
243. **delete**[] b;
244. }
246. **double** HiddenLayer::output(**int** \*input, **double** \*w, **double** b) {
247. **double** linear\_output = 0.0;
248. **for**(**int** j=0; j<n\_in; j++) {
249. linear\_output += w[j] \* input[j];
250. }
251. linear\_output += b;
252. **return** sigmoid(linear\_output);
253. }
255. **void** HiddenLayer::sample\_h\_given\_v(**int** \*input, **int** \*sample) {
256. **for**(**int** i=0; i<n\_out; i++) {
257. sample[i] = binomial(1, output(input, W[i], b[i]));
258. }
259. }

262. // RBM
263. RBM::RBM(**int** size, **int** n\_v, **int** n\_h, **double** \*\*w, **double** \*hb, **double** \*vb) {
264. N = size;
265. n\_visible = n\_v;
266. n\_hidden = n\_h;
268. **if**(w == NULL) {
269. W = **new** **double**\*[n\_hidden];
270. **for**(**int** i=0; i<n\_hidden; i++) W[i] = **new** **double**[n\_visible];
271. **double** a = 1.0 / n\_visible;
273. **for**(**int** i=0; i<n\_hidden; i++) {
274. **for**(**int** j=0; j<n\_visible; j++) {
275. W[i][j] = uniform(-a, a);
276. }
277. }
278. } **else** {
279. W = w;
280. }
282. **if**(hb == NULL) {
283. hbias = **new** **double**[n\_hidden];
284. **for**(**int** i=0; i<n\_hidden; i++) hbias[i] = 0;
285. } **else** {
286. hbias = hb;
287. }
289. **if**(vb == NULL) {
290. vbias = **new** **double**[n\_visible];
291. **for**(**int** i=0; i<n\_visible; i++) vbias[i] = 0;
292. } **else** {
293. vbias = vb;
294. }
295. }
297. RBM::~RBM() {
298. // for(int i=0; i<n\_hidden; i++) delete[] W[i];
299. // delete[] W;
300. // delete[] hbias;
301. **delete**[] vbias;
302. }

305. **void** RBM::contrastive\_divergence(**int** \*input, **double** lr, **int** k) {
306. **double** \*ph\_mean = **new** **double**[n\_hidden];
307. **int** \*ph\_sample = **new** **int**[n\_hidden];
308. **double** \*nv\_means = **new** **double**[n\_visible];
309. **int** \*nv\_samples = **new** **int**[n\_visible];
310. **double** \*nh\_means = **new** **double**[n\_hidden];
311. **int** \*nh\_samples = **new** **int**[n\_hidden];
313. /\* CD-k \*/
314. sample\_h\_given\_v(input, ph\_mean, ph\_sample);
316. **for**(**int** step=0; step<k; step++) {
317. **if**(step == 0) {
318. gibbs\_hvh(ph\_sample, nv\_means, nv\_samples, nh\_means, nh\_samples);
319. } **else** {
320. gibbs\_hvh(nh\_samples, nv\_means, nv\_samples, nh\_means, nh\_samples);
321. }
322. }
324. **for**(**int** i=0; i<n\_hidden; i++) {
325. **for**(**int** j=0; j<n\_visible; j++) {
326. W[i][j] += lr \* (ph\_sample[i] \* input[j] - nh\_means[i] \* nv\_samples[j]) / N;
327. }
328. hbias[i] += lr \* (ph\_sample[i] - nh\_means[i]) / N;
329. }
331. **for**(**int** i=0; i<n\_visible; i++) {
332. vbias[i] += lr \* (input[i] - nv\_samples[i]) / N;
333. }
335. **delete**[] ph\_mean;
336. **delete**[] ph\_sample;
337. **delete**[] nv\_means;
338. **delete**[] nv\_samples;
339. **delete**[] nh\_means;
340. **delete**[] nh\_samples;
341. }
343. **void** RBM::sample\_h\_given\_v(**int** \*v0\_sample, **double** \*mean, **int** \*sample) {
344. **for**(**int** i=0; i<n\_hidden; i++) {
345. mean[i] = propup(v0\_sample, W[i], hbias[i]);
346. sample[i] = binomial(1, mean[i]);
347. }
348. }
350. **void** RBM::sample\_v\_given\_h(**int** \*h0\_sample, **double** \*mean, **int** \*sample) {
351. **for**(**int** i=0; i<n\_visible; i++) {
352. mean[i] = propdown(h0\_sample, i, vbias[i]);
353. sample[i] = binomial(1, mean[i]);
354. }
355. }
357. **double** RBM::propup(**int** \*v, **double** \*w, **double** b) {
358. **double** pre\_sigmoid\_activation = 0.0;
359. **for**(**int** j=0; j<n\_visible; j++) {
360. pre\_sigmoid\_activation += w[j] \* v[j];
361. }
362. pre\_sigmoid\_activation += b;
363. **return** sigmoid(pre\_sigmoid\_activation);
364. }
366. **double** RBM::propdown(**int** \*h, **int** i, **double** b) {
367. **double** pre\_sigmoid\_activation = 0.0;
368. **for**(**int** j=0; j<n\_hidden; j++) {
369. pre\_sigmoid\_activation += W[j][i] \* h[j];
370. }
371. pre\_sigmoid\_activation += b;
372. **return** sigmoid(pre\_sigmoid\_activation);
373. }
375. **void** RBM::gibbs\_hvh(**int** \*h0\_sample, **double** \*nv\_means, **int** \*nv\_samples, \
376. **double** \*nh\_means, **int** \*nh\_samples) {
377. sample\_v\_given\_h(h0\_sample, nv\_means, nv\_samples);
378. sample\_h\_given\_v(nv\_samples, nh\_means, nh\_samples);
379. }
381. **void** RBM::reconstruct(**int** \*v, **double** \*reconstructed\_v) {
382. **double** \*h = **new** **double**[n\_hidden];
383. **double** pre\_sigmoid\_activation;
385. **for**(**int** i=0; i<n\_hidden; i++) {
386. h[i] = propup(v, W[i], hbias[i]);
387. }
389. **for**(**int** i=0; i<n\_visible; i++) {
390. pre\_sigmoid\_activation = 0.0;
391. **for**(**int** j=0; j<n\_hidden; j++) {
392. pre\_sigmoid\_activation += W[j][i] \* h[j];
393. }
394. pre\_sigmoid\_activation += vbias[i];
396. reconstructed\_v[i] = sigmoid(pre\_sigmoid\_activation);
397. }
399. **delete**[] h;
400. }

403. // LogisticRegression
404. LogisticRegression::LogisticRegression(**int** size, **int** in, **int** out) {
405. N = size;
406. n\_in = in;
407. n\_out = out;
409. W = **new** **double**\*[n\_out];
410. **for**(**int** i=0; i<n\_out; i++) W[i] = **new** **double**[n\_in];
411. b = **new** **double**[n\_out];
413. **for**(**int** i=0; i<n\_out; i++) {
414. **for**(**int** j=0; j<n\_in; j++) {
415. W[i][j] = 0;
416. }
417. b[i] = 0;
418. }
419. }
421. LogisticRegression::~LogisticRegression() {
422. **for**(**int** i=0; i<n\_out; i++) **delete**[] W[i];
423. **delete**[] W;
424. **delete**[] b;
425. }

428. **void** LogisticRegression::train(**int** \*x, **int** \*y, **double** lr) {
429. **double** \*p\_y\_given\_x = **new** **double**[n\_out];
430. **double** \*dy = **new** **double**[n\_out];
432. **for**(**int** i=0; i<n\_out; i++) {
433. p\_y\_given\_x[i] = 0;
434. **for**(**int** j=0; j<n\_in; j++) {
435. p\_y\_given\_x[i] += W[i][j] \* x[j];
436. }
437. p\_y\_given\_x[i] += b[i];
438. }
439. softmax(p\_y\_given\_x);
441. **for**(**int** i=0; i<n\_out; i++) {
442. dy[i] = y[i] - p\_y\_given\_x[i];
444. **for**(**int** j=0; j<n\_in; j++) {
445. W[i][j] += lr \* dy[i] \* x[j] / N;
446. }
448. b[i] += lr \* dy[i] / N;
449. }
451. **delete**[] p\_y\_given\_x;
452. **delete**[] dy;
453. }
455. **void** LogisticRegression::softmax(**double** \*x) {
456. **double** max = 0.0;
457. **double** sum = 0.0;
459. **for**(**int** i=0; i<n\_out; i++) **if**(max < x[i]) max = x[i];
460. **for**(**int** i=0; i<n\_out; i++) {
461. x[i] = exp(x[i] - max);
462. sum += x[i];
463. }
465. **for**(**int** i=0; i<n\_out; i++) x[i] /= sum;
466. }
468. **void** LogisticRegression::predict(**int** \*x, **double** \*y) {
469. **for**(**int** i=0; i<n\_out; i++) {
470. y[i] = 0;
471. **for**(**int** j=0; j<n\_in; j++) {
472. y[i] += W[i][j] \* x[j];
473. }
474. y[i] += b[i];
475. }
477. softmax(y);
478. }




484. **void** test\_dbn() {
485. srand(0);
487. **double** pretrain\_lr = 0.1;
488. **int** pretraining\_epochs = 1000;
489. **int** k = 1;
490. **double** finetune\_lr = 0.1;
491. **int** finetune\_epochs = 500;
493. **int** train\_N = 6;
494. **int** test\_N = 3;
495. **int** n\_ins = 6;
496. **int** n\_outs = 2;
497. **int** hidden\_layer\_sizes[] = {3, 3};
498. **int** n\_layers = **sizeof**(hidden\_layer\_sizes) / **sizeof**(hidden\_layer\_sizes[0]);
500. // training data
501. **int** train\_X[6][6] = {
502. {1, 1, 1, 0, 0, 0},
503. {1, 0, 1, 0, 0, 0},
504. {1, 1, 1, 0, 0, 0},
505. {0, 0, 1, 1, 1, 0},
506. {0, 0, 1, 1, 0, 0},
507. {0, 0, 1, 1, 1, 0}
508. };
510. **int** train\_Y[6][2] = {
511. {1, 0},
512. {1, 0},
513. {1, 0},
514. {0, 1},
515. {0, 1},
516. {0, 1}
517. };

  // construct DBN

  DBN dbn(train\_N, n\_ins, hidden\_layer\_sizes, n\_outs, n\_layers);

  // pretrain

  dbn.pretrain(\*train\_X, pretrain\_lr, k, pretraining\_epochs);

  // finetune

  dbn.finetune(\*train\_X, \*train\_Y, finetune\_lr, finetune\_epochs);

  // test data

**int** test\_X[3][6] = {

    {1, 1, 0, 0, 0, 0},

    {0, 0, 0, 1, 1, 0},

    {1, 1, 1, 1, 1, 0}

  };

**double** test\_Y[3][2];

  // test

**for**(**int** i=0; i<test\_N; i++) {

    dbn.predict(test\_X[i], test\_Y[i]);

**for**(**int** j=0; j<n\_outs; j++) {

      cout << test\_Y[i][j] << " ";

    }

    cout << endl;

  }

}

**int** main() {

  test\_dbn();

**return** 0;

}

<p>程序运行结果，是个二维的回归值：</p>

<p>0.493724  0.5062760.493724  0.5062760.493724  0.506276</p>

<p>转载请注明出处：http://blog.csdn.net/cuoqu/article/details/8896636</p>

<pre></pre>

# [DeepLearning（深度学习）原理与实现（四）](http://blog.csdn.net/marvin521/article/details/8906278)

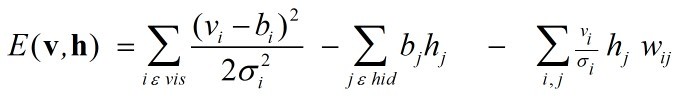
        今天就来讨论下deep learning怎么来处理real valued data。对于图像来说，二值图像毕竟是少数，更多的还是实值图像。对于这样的情况，RBM已经无法很好的处理它们，因此需要改进它，对于了解计算机视觉的人而言，想必高斯混合背景模型大家已不陌生，高斯混合模型可以很好的对实值图像建模，OpenCV中早就用高斯混合背景模型来分割物体。接下来要引出的高斯有限制玻尔兹曼机（Gaussian Restricted Boltzmann machine-GRBM）和高斯混合模型有一定的等价性。

      在上一节中，作者提到过对于RBM这个无向图模型而言，无论是给定隐藏节点还是可视节点，剩下的那层内节点之间互相独立。那么对于RBM这个图模型描述的联合分布可以分解成（公式一）的形式：

http://img.blog.csdn.net/20130509161829491

（公式一）

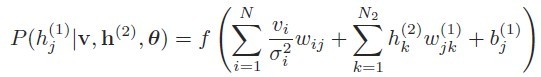
       其中P（h）可看成高斯混合系数，如果P(v|h)是高斯分布，那么这个条件下的RBM就和高斯混合分布具有等价性。因此GRBM合理登场大笑 ,下面开始进入GRBM正题,首先当然要假定P(v|h)服从高斯分布，不然没法进展下去咯，先来看下GRBM的能量公式定义（公式二）：



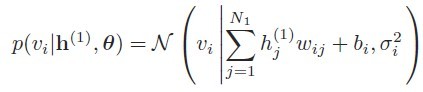
（公式二）

        从能量公式中可以很明显的看出有高斯分布的影子，当然还有其他类型的能量公式，关键看你做什么假设，假设不一样，能量公式也不一样，假设的好就会产生一篇好文章哦。其中参数要求的为：http://img.blog.csdn.net/20130509162117730。求取算法和此系列文章一方法类似，也是用CD算法，但是对于GRBM的CD算法，需要多啰嗦几句，首先我们把博文一中的求取数据期望项和模型期望项的过程改成官方称呼：分别对应为正阶段(positive phase)和负阶段(negative phase)。

         正阶段的目的就是找到给定数据的情况下，隐藏节点的配置（暂称它为"好配置"），并且使得能量降低。负阶段的目的当然就是抬升"好配置"周围的配置的能量，可能有点拗口，推荐看下博文一中的参考文献：Energy based model tutorial，可以把它总结成一句话：把和实际样本有关的配置能量降低，把fancy的样本的能量抬高，这样得到的权重是对样本的最佳描述，当然样本越多越好，所以大数据这么火偷笑。有些人会有疑问，样本多会不会导致过拟合？其实生成模型的魅力之一就在这，样本越多越好。正阶段中隐藏节点h的状态求取仍然用sigmoid函数得出概率，然后采样；但是负阶段中可视节点V的状态的概率就不是用sigmoid函数了，而是用高斯函数，二者的公式分别如（公式三）和（公式四）表示：



（公式三,f为sigmoid函数）



（公式四，高斯分布函数）

        对于参数的求取，过程只是多了sigma而已，如果你熟悉高斯混合模型，自然对sigma的实际作用也有些直观认识，庆幸的是sigma仍然也可以在CD算法中与其他参数一样同时求出来。下面给出GRBM中CD算法的部分代码(matlab)，其中invfstdInc为sigma，不要用用这段代码，而是用它解释自己对GRBM实现细节的疑问，这段代码中，CD算法引入了momentum(冲量）技巧，可以让算法收敛快些：

**[plain]** [view plaincopy](http://blog.csdn.net/marvin521/article/details/8906278)

1. %%%%%%%%% START POSITIVE PHASE %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
2. data = batchdata(:,:,batch); %nxd
3. pos\_hidprobs = 1./(1 + exp(-(data./Fstd)\*vhW - Ones\*hb)); %p(h\_j =1|data)
4. pos\_hidstates = pos\_hidprobs > rand( size(pos\_hidprobs) );
6. pos\_prods    = (data./Fstd)'\* pos\_hidprobs;
7. pos\_hid\_act  = sum(pos\_hidprobs);
8. pos\_vis\_act  = sum(data)./(fstd.^2); %see notes on this derivation
10. %%%%%%%%% END OF POSITIVE PHASE %%%%%%%%%
11. for iterCD = 1:params.nCD
13. %%%%%%%%% START NEGATIVE PHASE  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
14. negdataprobs = pos\_hidstates\*vhW'.\*Fstd+Ones\*vb;
15. negdata = negdataprobs + randn(n, d).\*Fstd;
16. neg\_hidprobs = 1./(1 + exp(-(negdata./Fstd)\*vhW - Ones\*hb ));     %updating hidden nodes again
17. pos\_hidstates = neg\_hidprobs > rand( size(neg\_hidprobs) );
19. end %end CD iterations
21. neg\_prods  = (negdata./Fstd)'\*neg\_hidprobs;
22. neg\_hid\_act = sum(neg\_hidprobs);
23. neg\_vis\_act = sum(negdata)./(fstd.^2); %see notes for details
25. %%%%%%%%% END OF NEGATIVE PHASE %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
27. errsum = errsum + sum(sum( (data-negdata).^2 ));
29. if epoch > params.init\_final\_momen\_iter,
30. momentum=params.final\_momen;
31. else
32. momentum=params.init\_momen;
33. end
35. %%%%%%%%% UPDATE WEIGHTS AND BIASES %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
37. vhWInc = momentum\*vhWInc + r/n\*(pos\_prods-neg\_prods) - r\*params.wtcost\*vhW;
38. vbInc = momentum\*vbInc + (r/n)\*(pos\_vis\_act-neg\_vis\_act);
39. hbInc = momentum\*hbInc + (r/n)\*(pos\_hid\_act-neg\_hid\_act);
41. invfstd\_grad = sum(2\*data.\*(Ones\*vb-data/2)./Fstd,1) + sum(data' .\* (vhW\*pos\_hidprobs') ,2)';
42. invfstd\_grad = invfstd\_grad - ( sum(2\*negdata.\*(Ones\*vb-negdata/2)./Fstd,1) + ...
43. sum( negdata'.\*(vhW\*neg\_hidprobs') ,2 )' );
45. invfstdInc = momentum\*invfstdInc + std\_rate(epoch)/n\*invfstd\_grad;
47. if params.SPARSE == 1 %nair's paper on 3D object recognition
48. %update q
49. if batch==1 && epoch == 1
50. q = mean(pos\_hidprobs);
51. else
52. q\_prev = q;
53. q = 0.9\*q\_prev+0.1\*mean(pos\_hidprobs);
54. end
56. p = params.sparse\_p;
57. grad = 0.1\*params.sparse\_lambda/n\*sum(pos\_hidprobs.\*(1-pos\_hidprobs)).\*(p-q)./(q.\*(1-q));
58. gradW =0.1\*params.sparse\_lambda/n\*(data'./Fstd'\*(pos\_hidprobs.\*(1-pos\_hidprobs))).\*repmat((p-q)./(q.\*(1-q)), d,1);
60. hbInc = hbInc + r\*grad;
61. vhWInc = vhWInc + r\*gradW;
62. end
64. ptot = ptot+mean(pos\_hidprobs(:));
66. vhW = vhW + vhWInc;
67. vb = vb + vbInc;
68. hb = hb + hbInc;
70. invfstd = 1./fstd;
71. invfstd =  invfstd + invfstdInc;
72. fstd = 1./invfstd;
74. fstd = max(fstd, 0.005); %have a lower bound!

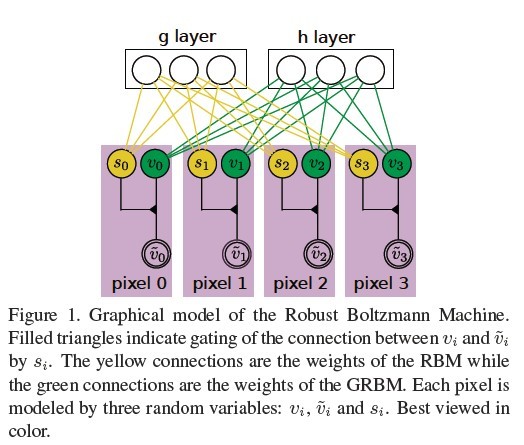
    最后来一句：别忘了这是单层GRBM，深度学习需要多层的，GDBM，读者可以畅想其魅力，尽管如此，作者仍然要保持开拓的姿态，这个模型还不能处理大脑联想这样一个简单功能。

参考文献：  
     Guassian-Bernoulli Deep Boltzmann Machine. KyungHyun Cho

# [DeepLearning（深度学习）原理与实现（五）](http://blog.csdn.net/marvin521/article/details/8929748)

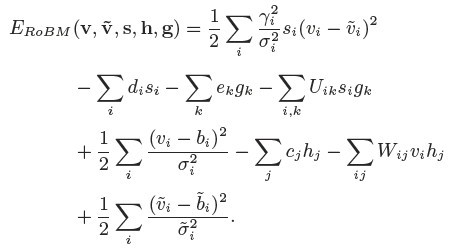
结构决定功能和协同处理：鲁棒有限制玻尔兹曼机(RoBM）

     上一篇博文中提到，GRBM训练样本越多越好，样本多蕴含的分布更具有一般性，这对其他模型也适用。但是实际样本数据中往往有大量的噪声，这或多或少的影响了GRBM的性能，工业界一般都设有清洗数据的岗位，用人脑去除噪声数据。试想：人脑为什么具有如此强大的抗噪和容错能力？其实生命科学中有一句经典的总结：结构决定功能。不同的结构的网络具有不同的功能，下面要引入的鲁棒玻尔兹曼机（Robust boltzmann machine-RoBM）就是一个很好的例证。RoBM由GRBM和RBM组成，确切的说还有一个成分：选择门。对于这个模型我不想说它怎么训练的咯，因为GRBM和RBM的训练方法前面都说过了，我只想强调下“结构决定功能”怎么体现在概率图模型上。RoBM的模型如（图一）所示：



（图一）

       在给定V-tilde(tilde就是V头上的波浪号哈，表示输入节点像素)时，S和V是独立的，同样G和H是独立的，对于GRBM和RBM的节点独立前面也已经讨论过。由于这些独立性再结合对于我们问题有利的假设（如高斯分布），能量函数定义就出来了,（对于能量函数的定义，也可以用概率图模型的clque(团)来解释，其实也是变量间的独立关系决定）。RoBM的能量函数如（公式一）所示：



（公式一）

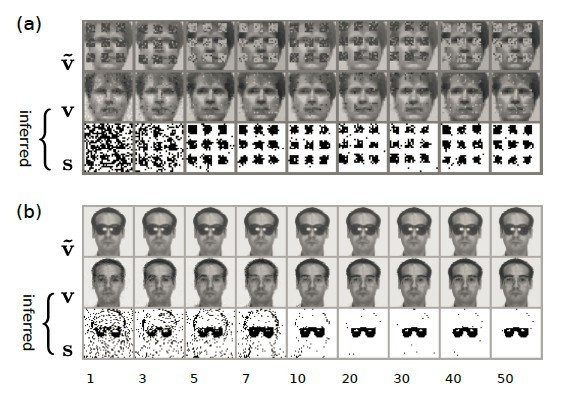
         上面公式的第一行就是牵扯到变量S,V和V-tilde的选择门项，这项其实就决定选择门的功能，控制信号的输入，比如当S等于0时，随你V和V-tilde怎么变，这块能量不会太高；当S等于1时，gamma的平方控制着V和V-tilde的差异程度，如果二者差异太大，这块能量就会偏高。

         第二行就是关于噪声S的RBM能量项（和RBM的能量公式是一样的），它控制着模型和噪声的关系，S是一个指示变量，表示该样本是否“应该”有噪声（“是否应该”这是个配置，有系统来完成）。

         第三行当然就是干净数据V的GRBM能量项了（和GRBM的能量公式是一样的），他控制着模型和干净数据的关系。

        第四行就是噪声的分布，假设噪声成高斯分布。

        这个模型的高明之处就是它先分工了噪声去除和鉴别样本是否有噪声，然后在总体框架上协调完成他们，每行的能量项分别完成某个功能，而总体能量又协调他们做的最好，形成最佳搭档。至于训练，先是用干净数据训练GRBM，然后在训练总体模型，中间的采样还跟前面有点不一样，具体看参考文献的推理(Inference)部分。实验效果如（图二）所示



（图二）

        由（图二）可以看出这个模型对输入有噪声的数据不但分离出噪声，也重建出了干净数据，最近比较有意思的是剑桥大学发布的那几个有错误的句子和段落，人脑同样可以正确的阅读他们，大脑是否是类似这样一个过程呢？开放话题，尚无结论。本来不打算写这个deep learning的应用，但是国内最大的自然语言处理群内的成员们热情的讨论勾引了俺的兴趣，要感谢他们一下哈大笑,另外这个模型的代码有点复杂，就不贴了，作者主页也挂出来了哈。

参考文献：

Robust Boltzmann Machines for Recognition and Denoising.Yi chuanTang,Ruslan,Geoffery Hinton

转载请注明出处：<http://blog.csdn.net/cuoqu/article/details/8929748>

# [Deep Learning(Logistic Regression)学习之MNIST C++实现](http://blog.csdn.net/puretech_/article/details/12215333)

## 1、这是75%完成度的SGD代码

主要的程序在八月中旬基本完成了，后边忙别的事情，实在懒得再仔细写了，目前的代码是75%完成度的，已知bug和可以改进的地方有：

1、在多次进行下降后，参数矩阵可能变成NaN值。（即由于参数计算导致除零错，目前尚未排查，估计在6%错误率左右的时候出现）

2、并发性可以进行改进。包括采用GPU进行计算的可能。

3、断点保存可以改进。目前保存每次的参数矩阵，可以从上次参数矩阵进行下降。但尚未保存每次退出时进行下降的训练例子位置等信息。

这个SGD的效果还行，应该不比原例子差，不过性能就差了不止十万八千里（囧）。但是考虑到这是第一次写这种算法，没有必要对自己吹毛求疵，而且目的是为了学习和理解SGD算法，自我安慰安慰。

原例子在<http://deeplearning.net/tutorial/gettingstarted.html>，采用python写的，主要数学工具是Theano。

这个SGD代码的程序一会儿我上载到资源区。

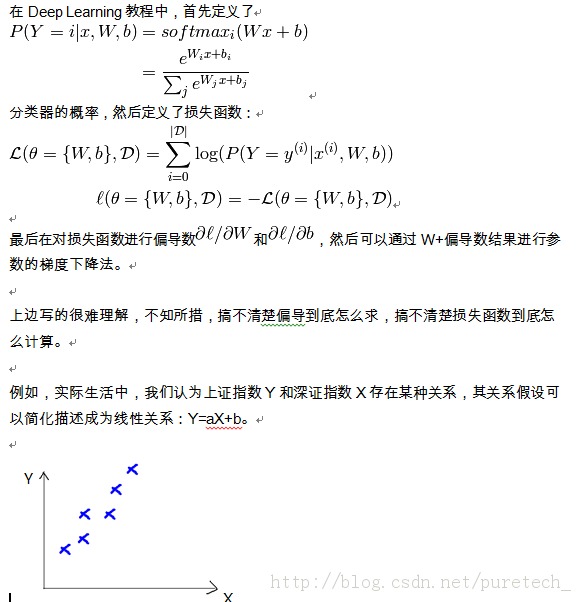
## 2、为什么要用 C++实现SGD算法

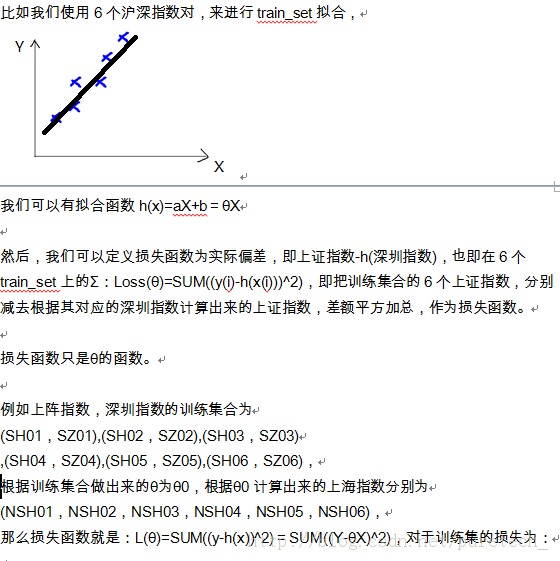
这是因为没有搞定Theano。Logistic-SGD所需环境包括Theano，但是在尝试在Win7/IronPython VS2010环境下安装Theano未果之后，只能采取手动挡来研究一下Logistic-SGD的代码了。

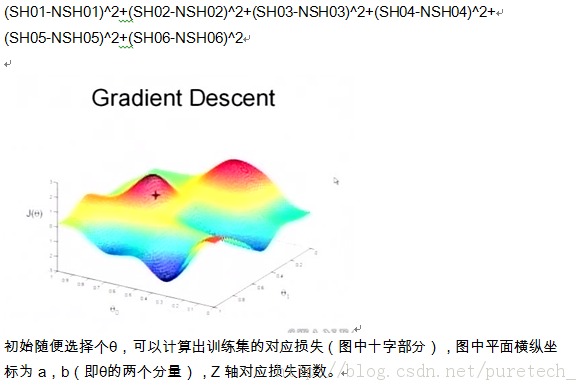
基本思路是通过IronPython+VS2010 .Net环境，在找一些数学库（用于矩阵运算等等），替换Theano的部分内容，使得程序能够跑起来。

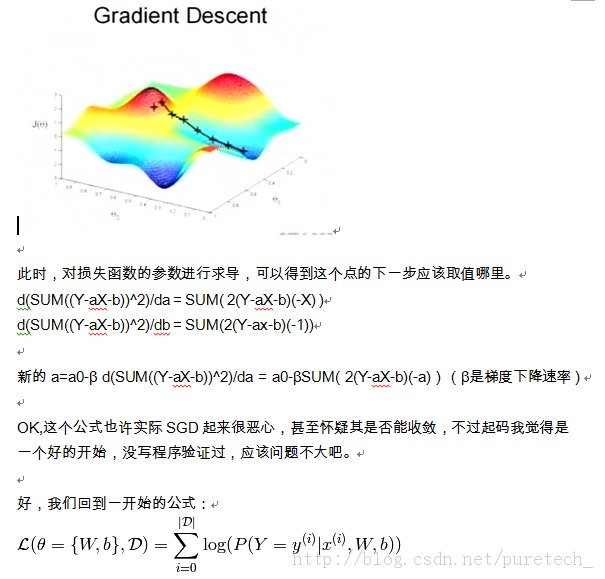
结果后来VS2010.Net操作IronPython没有搞定numpy库（抓狂，在外边直接可以用numpy，在C++初始化IronPython却不能使用numpy），加上学的深入一点后，似乎不用特别复杂的数学库，直接实现也可以。于是就有了这个程序。

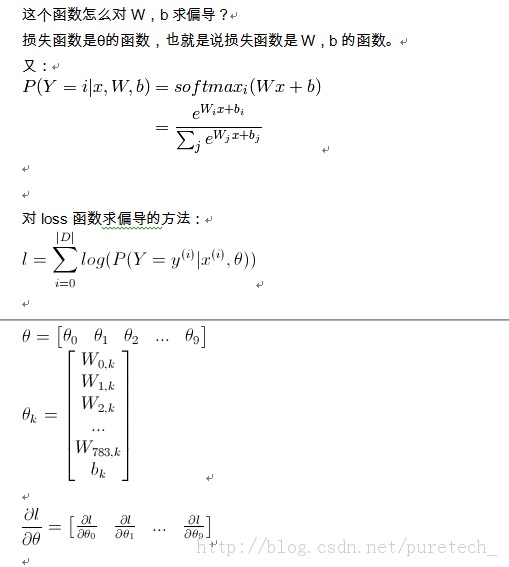
## 3、SGD理论

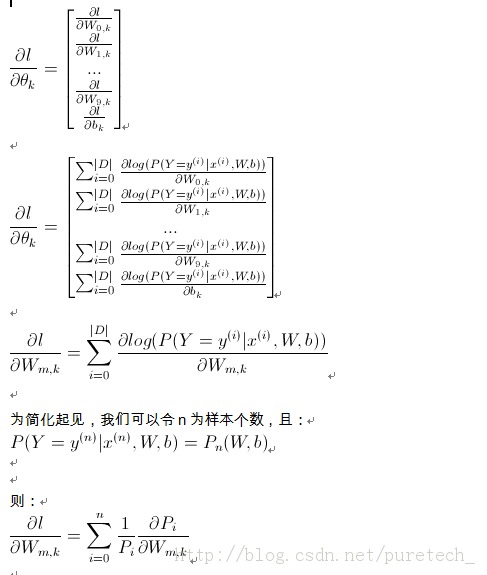


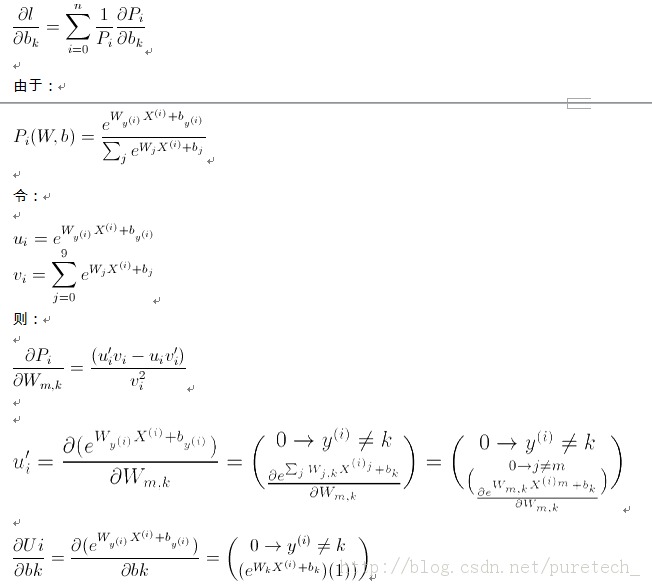


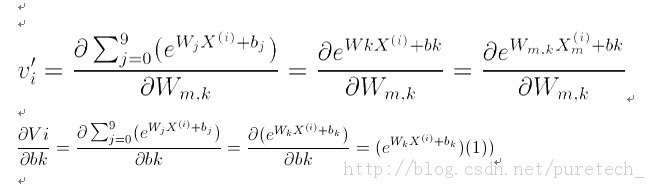












## 4、程序

原例子中的mnist.pkg.gz是压缩后的数据族，为了能够便于C++直接处理，我们首先把mnist.pkg.gz改成易于处理的数据-结果对，即：将mnist.pkg解压为6个文件：

train\_set\_x

train\_set\_y

valid\_set\_x

valid\_set\_y

test\_set\_x

test\_set\_y

对应train,valid,test三个集合（每个集合两对，即数据-结果对）。train\_set\_x中的784行对应train\_set\_y中的一行，即代表一个测试数据集合。其中train\_set\_x每一行是一个浮点数，748行（784为28x28的灰度浮点数）代表一个数字的灰度图，对应train\_set\_y的一行，即对应的数字。

这个python程序如下，很简单，写成文本文件好了，然后再写一个C++程序读这个文本文件，转成相关的 二进制文件。

import cPickle

f = open(‘mnist.pkg’, 'rb')

train\_set,valid\_set, test\_set = cPickle.load(f)

f.close()

test\_set\_x, test\_set\_y = test\_set

train\_set\_x, train\_set\_y = train\_set

valid\_set\_x, valid\_set\_y = valid\_set

fileHandle=open("test\_set\_x","w")

cnt = 0

for i in test\_set\_x:

       ifcnt% 10 ==0:

              printcnt

       cnt= cnt +1

       forj in i:

              fileHandle.write("%.6f\n"%j)

fileHandle.close()

cnt = 0

fileHandle=open("test\_set\_y","w")

for i in test\_set\_y:

       ifcnt% 10 ==0:

              printcnt

       cnt= cnt +1

       fileHandle.write("%d\n"%i)

       fileHandle=open("train\_set\_x","w")

cnt=0

for i in train\_set\_x:

       ifcnt% 10 ==0:

              printcnt

       cnt= cnt +1

       forj in i:

              fileHandle.write("%.6f\n"%j)

fileHandle.close()

fileHandle=open("train\_set\_y","w")

cnt=0

for i in train\_set\_y:

       ifcnt% 10 ==0:

              printcnt

       cnt= cnt +1

       fileHandle.write("%d\n"%i)

为了方便，已经做了二进制文件。

### 4.1、程序结构

整个程序由三部分组成：主程序、SGD主程序、文件操作支持函数组成。

数据格式在dataset.h中定义，一个集合（训练、验证、测试）为：

|  |
| --- |
| /\* DSET\_X 为每一个样例的灰度图，由784个double构成 \*/  typedef double    DSET\_X[784];    typedef struct st\_set\_array  {      int    m\_iCount; /\* 集合中样本个数 \*/      DSET\_X \*pArrayX;  /\* 每一个样本的灰度图 \*/      int    \*pArrayY;  /\* 对应的样本识别标签 \*/  }SET\_ARRAY; |

训练对应的参数w矩阵和b向量定义如下：

|  |
| --- |
| /\*   \* WMATRIX W matrix parameter   \* bVECTOR  b vector parameter   \*/  typedef DSET\_X    WMATRIX[10];  typedef double    bVECTOR[10]; |

矩阵定义其实可以用简单的wMatrix[ ][ ]来描述，但是我比较笨，懒得去思考C++二维数组的行列，索性就简单的先定义DSET\_X然后再定义一维数组，之后的操作中全部采用一维数组的操作，避免因为笨导致的问题。

定义了一个控制块，用于各个函数中传递参数使用，

这里的内容可以不求甚解。看变量名也应该可以猜到干啥的，m\_dLearnRate是学习率（就是参数delta的系数），Epoch是最大学习迭代次数。

按照整体的思想，我每次训练一批（一个BatchSize个训练样本），就调整并dump一次参数，然后训练一定批次后，用validate样本进行验证，当valid有改善时，用test样本进行验证。

中间退出的话，由于保存了每批dump的参数，下次直接可以用最新的参数进行训练。

Ok，let‘s do it。

### 4.2、主程序

主程序很简单，在minst04.cpp中（为啥是04？因为这是我第四次重写这个程序）。关键点在以下几行代码：

1、首先是读取训练样本文件

LoadDataSet(&TrainSet,&TestSet,&ValidSet);

这个是从"C:\\DEVELOP\\MNIST\\“下读取相关的二进制训练集合（LoadDataSet的函数实现也在mnist04.cpp中，其具体读取文件的函数实现在FileConvert.cpp中。

2、初始化sgd控制结构

ZeroMemory(&m\_sgd,sizeof(m\_sgd));

m\_sgd.m\_dLearnRate=0.13;         学习率0.13

m\_sgd.m\_iBatchSize=600;                   600个样本一组

m\_sgd.m\_iEpoch=1000;                最多训练1000次

m\_sgd.m\_iPatience=5000;                   。。。难以言表

m\_sgd.m\_iPatienceIncrease=2;           。。。难以言表

m\_sgd.m\_pTestSet = &TestSet;          测试集合

m\_sgd.m\_pTrainSet = &TrainSet;        训练集合

m\_sgd.m\_pValidSet = &ValidSet;         验证集合

3、读取上次dump的参数集合。

我们每次dump的参数文件在”C:\\DEVELOP\\MNIST\\parms\\“目录下，文件名以parms打头，因此读取上次dump的参数是找到最新的parms文件，然后读取之：

GetLatestFile("C:\\DEVELOP\\MNIST\\parms\\","parms\*.txt",sName);

LoadParameters(sName,m\_sgd.m\_wMatrix,m\_sgd.m\_bVector);

相关函数在FileConvert.cpp中。

读取完，马上dump一份（可以看看读的是否正确。。。-\_-!）

DumpMatrix("C:\\DEVELOP\\MNIST\\parms\\",m\_sgd.m\_wMatrix,m\_sgd.m\_bVector);

4、开始SGD

test\_sgd(&m\_sgd);

### 4.3、SGD

SGD的核心就是算几个数：

第一个是算http://img.blog.csdn.net/20131001093344937?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvcHVyZXRlY2hf/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/SouthEast

，即，给定某一个样本X，在W、b参数下，得出结果是i的概率。这个简单，在Softmax.cpp中，我们实现了两个（其实是一个意思，一个是快速查表算法Pn，一个是标准算法PnNoFast），

以PnNoFast作为样例解剖，原型：

double PnNoFast(int iY,int iItem,SET\_ARRAY \*pSet,WMATRIX w,bVECTOR b)

返回pSet集合中的第iItem个样本，在w和b参数下，当输出分类是iY时的概率。

关键代码：

pw = &w[iY];

px = &pSet->pArrayX[iItem];

dExp1 = 0;

/\* 计算矩阵相乘， Wi \* X + bi \*/

for(i=0;i<784;i++)

{

       dExp1 += (\*pw)[i]\*(\*px)[i];

}

dExp1 += b[iY];

dUpperValue=exp(dExp1);

这段代码实现了

http://img.blog.csdn.net/20131001093427062?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvcHVyZXRlY2hf/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/SouthEast

dLowerValue = 0;

for(j=0;j<10;j++)

{

       dExp1 = 0;

       pw = &w[j];

       for(i=0;i<784;i++)

       {

              dExp1 +=(\*pw)[i]\*(\*px)[i];

       }

       dExp1 += b[j];

       /\*sum e^(WjX+bj) \*/

       dLowerValue += exp(dExp1);

}

这段代码实现了http://img.blog.csdn.net/20131001093443578?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvcHVyZXRlY2hf/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/SouthEast

当然dUpperValue/dLowerValue=http://img.blog.csdn.net/20131001093508937?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvcHVyZXRlY2hf/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/SouthEast

第二个是算偏导数。

double Dpi\_dWmk(MATRIXINDEX pMatrixIndex,int iItem,int m,int k,WMATRIXwMatrix,bVECTOR bVector,SET\_ARRAY \*pSet)

{

       /\* dpi/dwmk = (u'v-uv')/v^2 \*/

       double du,dv,u,v;

       du =DuI(pMatrixIndex,iItem,m,k,wMatrix,bVector,pSet);

       dv =Dvi(pMatrixIndex,iItem,m,k,wMatrix,bVector,pSet);

       u =Ui(pMatrixIndex,iItem,k,wMatrix,bVector,pSet);

       v =Vi(pMatrixIndex,iItem,k,wMatrix,bVector,pSet);

       return (du\*v - u\*dv)/(v\*v);

}



http://img.blog.csdn.net/20131001093657546?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvcHVyZXRlY2hf/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/SouthEast

实现代码在BuildDeltaW\_b中

其核心就是三重循环，一是k从0-9循环，二是m从0-783循环，三是i从0到n循环，然后计算损失函数的导数。

dsm += pPi[i]\*Dpi\_dWmk(pMatrixIndex,i,m,k,wMatrix,bVector,pSet);

pPi[i]就是1/Pi的速查值。

test\_sgd的核心就是

循环：0-Epoch

循环：0-每一个batch

       计算这个batch的损失函数偏导数

       然后新参数=参数-学习率\*偏导数

       输出新参数矩阵

       如果该验证一下，就去验证一下，

              如果验证的有改进，就去test一下。

       判断是否可以提前结束

其他的…就不写了吧，反正test\_sgd也挺简单的。

懒…

## 5、感谢

这部分内容对我来说是最痛苦的，特别要感谢Rachel Zhang同学。Rachel耐心、细致的回答了我无数幼稚和愚笨的基础数学问题，并鼓励我尝试完成，使得这一切成为可能。

此外，还感谢<http://blog.sina.com.cn/s/blog_62339a2401015jyq.html>，其图示使得我豁然开朗，明白了梯度下降算法的意义。建议初学者可以去看看。

# Deep Belief Network(DBN)的实现(c++)

每每想到几年前写一篇关于Action Classification的paper的时候，苦于没有找到好的feature representation方法而失败了。那时候记得也是想表示成一层一层的特征出来，当时用了关联规则挖掘，效率没法克服；然后上了topic model的思想来组织图像，没做到点子上。现在想想也许当时多实验试验Hierarchical topic model的思想说不定能搞出点名堂。

去年Deep Learning被轰出来之后，在个方便都自然的取得了比较理想的效果。想到了自己之前的问题，或许可以借来用用。

所以，最近接触了Deep的思想，学习并实现了下。先从hinton最初的DBN着手，实现起来还是比较简单的。虽然有很多实现了，考虑的长远的规划，先自己实现着吧。

代码风格Too young, too simple, sometimes naive and many bugs. 不要笑话哈~；框架参考自：[yusugomori](https://github.com/yusugomori/DeepLearning)，不过他没写完的样子。。

**一、实现：**

1、Layer-wise pre-training阶段，一层一层的训练Restrict Boltzmann Machine （RBM），用的CD-k(k=1)的方法；

2、最后一层加入了softmax regression，作为最终结果输出层；

3、fine-tuning过程，即为传统的BP过程。

**二、实验**

1、用的MNIST的数据集，这里为了可以忍受自己写的幼儿园水平代码的效率，选择10000个样本training，60000个样本testing。不过传统上大家都把两者反过来，效果肯定会更好了。

2、学习速率0.1，rbm和LR的L2的惩罚相lamda开始用的0.0002，三个隐层大小[400 400 900],这些都没调

3、开始时候没有进行fine-tuning， accuracy大约在84%左右（不过这里是用的svm，当时lr还没写完。。），现在fine-tuning之后，用softmax regression对60000个样本分类结果为93%，还可以哈。不过把L2正则项的惩罚因子lamda设成0后，准确率到了95%。

4、调调参，或者用60000训练10000测试，达到95%以上应该没啥问题吧。

**三、源码**：<https://github.com/lipiji/PG_DEEP>

1、想做成像libsvm一样好用的tool，一个train可以训练出模型，一个predict可以预测你的数据，目前还在开发当中。

2、源码风格比较幼儿园，好处就是能够看懂每个细节，方便学习吧；怪自己一直没有遇到什么c++鲁棒规范实践的项目。

3、代码没有进行任何矩阵等计算的优化，优化空间很大。

**四、计划**

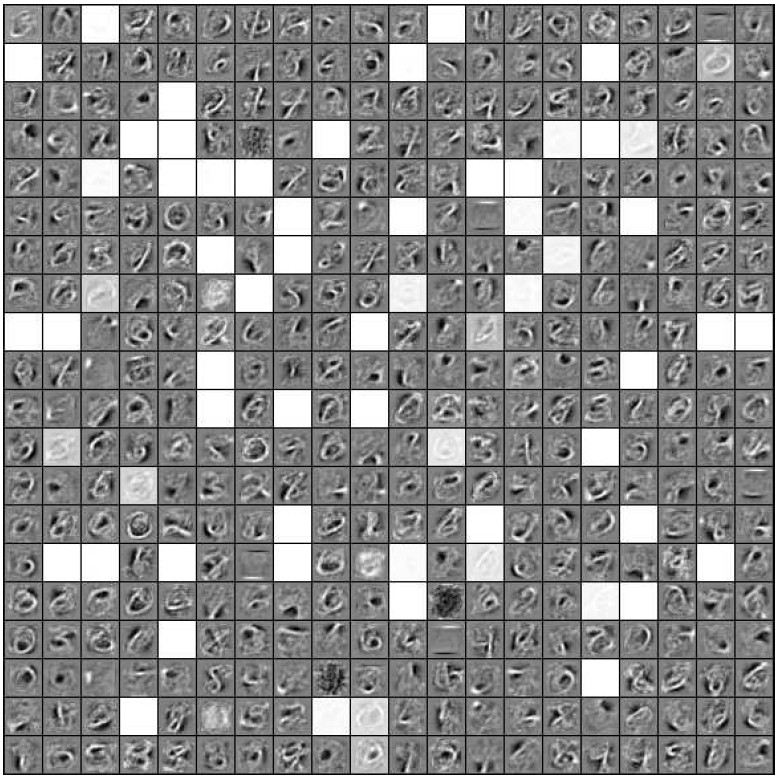
1、优化代码，理想是做一个像libsvm一样方便大家使用的tool；

2、学习实现DNN, AutoEncoder和Sparse Coding啊，convolution啊、dropout啊、GPU啊等等其他升级版本；

3、能够再次基础上扩展出vision、nlp等领域里面的那些好用的算法，例如word2vec等。

**五、附录**

1、第一层RBM的参数矩阵W1（768x400）可视化结果，W2、W3可视化跟随机噪声似的，或许应该用其他方法可视化。不过看着怎么不太舒服的样子，之所以有很多白色的区域，是因为引入正则项防止过拟合，体现了一定的稀疏性。正则项的惩罚因子设成0后，白色区域就消失了。

[](http://www.zhizhihu.com/wp-content/uploads/2013/09/dbn.jpg)

**PS：有bug大家反馈哈~**

Reference:

[1] A fast learning algorithm for deep belief nets

[2] A Practical Guide to Training Restricted Boltzmann Machines

[3] <https://github.com/yusugomori/DeepLearning>

# [Deep Learning字符识别C++程序(基于RBM)](http://blog.csdn.net/lucky_greenegg/article/details/8949578)

链接<http://download.csdn.net/detail/lucky_greenegg/5413211>中

的代码是根据DBN-RBM字符识别的MatLab程序写的C++版，

里面的测试数据已经转化为txt，其中每一行代表一张图像。完成  
手写字符识别，5000个样本，10个类1000个测试。正确率94%。  
由VS2008+OpenCV2.4.3实现。发现自己写程序已经脱离不了OpenCV了。  
由于本人纯属菜鸟，所以写出来的程序目前还只能说可以完成  
功能，什么编码风格啊、什么健壮性啊、可读性啊还无从谈起。  
经过测试，应该是可以正确运行的，结果和MatLab的差不多。  
另外，实在不想收积分，不过我自己也穷的叮当响，所以大家都不容易。

  拿自己的数据测试的话，如果要想效果好的话，需要修改参数……

请参考《A Practical Guide to Training Restricted Boltzmann Machines》

运行效果：

