# [【面向代码】学习 Deep Learning（一）Neural Network](http://blog.csdn.net/dark_scope/article/details/9421061)

最近一直在看Deep Learning，各类博客、论文看得不少

但是说实话，这样做有些疏于实现，一来呢自己的电脑也不是很好，二来呢我目前也没能力自己去写一个toolbox

只是跟着Andrew Ng的[UFLDL tutorial](http://ufldl.stanford.edu/wiki/index.php/UFLDL_Tutorial) 写了些已有框架的代码(这部分的代码见[github](https://github.com/justdark/matlab_code-ufldl-exercise-))

后来发现了一个matlab的Deep Learning的[toolbox](https://github.com/rasmusbergpalm/DeepLearnToolbox)，发现其代码很简单，感觉比较适合用来学习算法

再一个就是matlab的实现可以省略掉很多数据结构的代码，使算法思路非常清晰

所以我想在解读这个toolbox的代码的同时来巩固自己学到的，同时也为下一步的实践打好基础

(本文只是从代码的角度解读算法，具体的算法理论步骤还是需要去看paper的

我会在文中给出一些相关的paper的名字，本文旨在梳理一下算法过程，不会深究算法原理和公式)

==========================================================================================

使用的代码：[**DeepLearnToolbox**](https://github.com/rasmusbergpalm/DeepLearnToolbox)  ，下载地址：[点击打开](https://github.com/rasmusbergpalm/DeepLearnToolbox)，感谢该toolbox的作者

==========================================================================================

第一章从分析NN(neural network)开始，因为这是整个deep learning的大框架，参见[UFLDL](http://ufldl.stanford.edu/wiki/index.php/UFLDL_Tutorial)

==========================================================================================

首先看一下\tests\test\_example\_NN.m ，跳过对数据进行normalize的部分，最关键的就是：

（为了注释显示有颜色，我把matlab代码中的%都改成了//）

**[cpp]** [view plaincopy](http://blog.csdn.net/dark_scope/article/details/9421061)

1. nn = nnsetup([784 100 10]);
2. opts.numepochs =  1;   //  Number of full sweeps through data
3. opts.batchsize = 100;  //  Take a mean gradient step over this many samples
4. [nn, L] = nntrain(nn, train\_x, train\_y, opts);
5. [er, bad] = nntest(nn, test\_x, test\_y);

很简单的几步就训练了一个NN，我们发现其中最重要的几个函数就是nnsetup,nntrain和nntest了

那么我们分别来分析着几个函数，\NN\nnsetup.m

## nnsetup

**[cpp]** [view plaincopy](http://blog.csdn.net/dark_scope/article/details/9421061)

1. function nn = nnsetup(architecture)
2. //首先从传入的architecture中获得这个网络的整体结构，nn.n表示这个网络有多少层，可以参照上面的样例调用nnsetup([784 100 10])加以理解
4. nn.size   = architecture;
5. nn.n      = numel(nn.size);
6. //接下来是一大堆的参数，这个我们到具体用的时候再加以说明
7. nn.activation\_function              = 'tanh\_opt';   //  Activation functions of hidden layers: 'sigm' (sigmoid) or 'tanh\_opt' (optimal tanh).
8. nn.learningRate                     = 2;            //  learning rate Note: typically needs to be lower when using 'sigm' activation function and non-normalized inputs.
9. nn.momentum                         = 0.5;          //  Momentum
10. nn.scaling\_learningRate             = 1;            //  Scaling factor for the learning rate (each epoch)
11. nn.weightPenaltyL2                  = 0;            //  L2 regularization
12. nn.nonSparsityPenalty               = 0;            //  Non sparsity penalty
13. nn.sparsityTarget                   = 0.05;         //  Sparsity target
14. nn.inputZeroMaskedFraction          = 0;            //  Used for Denoising AutoEncoders
15. nn.dropoutFraction                  = 0;            //  Dropout level (http://www.cs.toronto.edu/~hinton/absps/dropout.pdf)
16. nn.testing                          = 0;            //  Internal variable. nntest sets this to one.
17. nn.output                           = 'sigm';       //  output unit 'sigm' (=logistic), 'softmax' and 'linear'
18. //对每一层的网络结构进行初始化，一共三个参数W,vW，p，其中W是主要的参数
19. //vW是更新参数时的临时参数，p是所谓的sparsity，(等看到代码了再细讲)
20. **for** i = 2 : nn.n
21. // weights and weight momentum
22. nn.W{i - 1} = (rand(nn.size(i), nn.size(i - 1)+1) - 0.5) \* 2 \* 4 \* sqrt(6 / (nn.size(i) + nn.size(i - 1)));
23. nn.vW{i - 1} = zeros(size(nn.W{i - 1}));
25. // average activations (for use with sparsity)
26. nn.p{i}     = zeros(1, nn.size(i));
27. end
28. end

## nntrain

setup大概就这样一个过程，下面就到了train了，打开\NN\nntrain.m

我们跳过那些检验传入数据是否正确的代码，直接到关键的部分

denoising 的部分请参考论文：Extracting and Composing Robust Features with Denoising Autoencoders

**[cpp]** [view plaincopy](http://blog.csdn.net/dark_scope/article/details/9421061)

1. m = size(train\_x, 1);
2. //m是训练样本的数量
3. //注意在调用的时候我们设置了opt，batchsize是做batch gradient时候的大小
4. batchsize = opts.batchsize; numepochs = opts.numepochs;
5. numbatches = m / batchsize;  //计算batch的数量
6. assert(rem(numbatches, 1) == 0, 'numbatches must be a integer');
7. L = zeros(numepochs\*numbatches,1);
8. n = 1;
9. //numepochs是循环的次数
10. **for** i = 1 : numepochs
11. tic;
12. kk = randperm(m);
13. //把batches打乱顺序进行训练，randperm(m)生成一个乱序的1到m的数组
14. **for** l = 1 : numbatches
15. batch\_x = train\_x(kk((l - 1) \* batchsize + 1 : l \* batchsize), :);
16. //Add noise to input (for use in denoising autoencoder)
17. //加入noise，这是denoising autoencoder需要使用到的部分
18. //这部分请参见《Extracting and Composing Robust Features with Denoising Autoencoders》这篇论文
19. //具体加入的方法就是把训练样例中的一些数据调整变为0，inputZeroMaskedFraction表示了调整的比例
20. **if**(nn.inputZeroMaskedFraction ~= 0)
21. batch\_x = batch\_x.\*(rand(size(batch\_x))>nn.inputZeroMaskedFraction);
22. end
23. batch\_y = train\_y(kk((l - 1) \* batchsize + 1 : l \* batchsize), :);
24. //这三个函数
25. //nnff是进行前向传播，nnbp是后向传播，nnapplygrads是进行梯度下降
26. //我们在下面分析这些函数的代码
27. nn = nnff(nn, batch\_x, batch\_y);
28. nn = nnbp(nn);
29. nn = nnapplygrads(nn);
30. L(n) = nn.L;
31. n = n + 1;
32. end
34. t = toc;
35. **if** ishandle(fhandle)
36. **if** opts.validation == 1
37. loss = nneval(nn, loss, train\_x, train\_y, val\_x, val\_y);
38. **else**
39. loss = nneval(nn, loss, train\_x, train\_y);
40. end
41. nnupdatefigures(nn, fhandle, loss, opts, i);
42. end
44. disp(['epoch ' num2str(i) '/' num2str(opts.numepochs) '. Took ' num2str(t) ' seconds' '. Mean squared error on training set is ' num2str(mean(L((n-numbatches):(n-1))))]);
45. nn.learningRate = nn.learningRate \* nn.scaling\_learningRate;
46. end

下面分析三个函数nnff,nnbp和nnapplygrads

**nnff**

nnff就是进行feedforward pass，其实非常简单，就是整个网络正向跑一次就可以了

当然其中有dropout和sparsity的计算

具体的参见论文“Improving Neural Networks with Dropout“和[Autoencoders and Sparsity](http://ufldl.stanford.edu/wiki/index.php/Autoencoders_and_Sparsity)

**[cpp]** [view plaincopy](http://blog.csdn.net/dark_scope/article/details/9421061)

1. function nn = nnff(nn, x, y)
2. //NNFF performs a feedforward pass
3. // nn = nnff(nn, x, y) returns an neural network structure with updated
4. // layer activations, error and loss (nn.a, nn.e and nn.L)
6. n = nn.n;
7. m = size(x, 1);
9. x = [ones(m,1) x];
10. nn.a{1} = x;
12. //feedforward pass
13. **for** i = 2 : n-1
14. //根据选择的激活函数不同进行正向传播计算
15. //你可以回过头去看nnsetup里面的第一个参数activation\_function
16. //sigm就是sigmoid函数,tanh\_opt就是tanh的函数，这个toolbox好像有一点改变
17. //tanh\_opt是1.7159\*tanh(2/3.\*A)
18. **switch** nn.activation\_function
19. **case** 'sigm'
20. // Calculate the unit's outputs (including the bias term)
21. nn.a{i} = sigm(nn.a{i - 1} \* nn.W{i - 1}');
22. **case** 'tanh\_opt'
23. nn.a{i} = tanh\_opt(nn.a{i - 1} \* nn.W{i - 1}');
24. end
26. //dropout的计算部分部分 dropoutFraction 是nnsetup中可以设置的一个参数
27. **if**(nn.dropoutFraction > 0)
28. **if**(nn.testing)
29. nn.a{i} = nn.a{i}.\*(1 - nn.dropoutFraction);
30. **else**
31. nn.dropOutMask{i} = (rand(size(nn.a{i}))>nn.dropoutFraction);
32. nn.a{i} = nn.a{i}.\*nn.dropOutMask{i};
33. end
34. end
35. //计算sparsity，nonSparsityPenalty 是对没达到sparsitytarget的参数的惩罚系数
36. //calculate running exponential activations for use with sparsity
37. **if**(nn.nonSparsityPenalty>0)
38. nn.p{i} = 0.99 \* nn.p{i} + 0.01 \* mean(nn.a{i}, 1);
39. end
41. //Add the bias term
42. nn.a{i} = [ones(m,1) nn.a{i}];
43. end
44. **switch** nn.output
45. **case** 'sigm'
46. nn.a{n} = sigm(nn.a{n - 1} \* nn.W{n - 1}');
47. **case** 'linear'
48. nn.a{n} = nn.a{n - 1} \* nn.W{n - 1}';
49. **case** 'softmax'
50. nn.a{n} = nn.a{n - 1} \* nn.W{n - 1}';
51. nn.a{n} = exp(bsxfun(@minus, nn.a{n}, max(nn.a{n},[],2)));
52. nn.a{n} = bsxfun(@rdivide, nn.a{n}, sum(nn.a{n}, 2));
53. end
54. //error and loss
55. //计算error
56. nn.e = y - nn.a{n};
58. **switch** nn.output
59. **case** {'sigm', 'linear'}
60. nn.L = 1/2 \* sum(sum(nn.e .^ 2)) / m;
61. **case** 'softmax'
62. nn.L = -sum(sum(y .\* log(nn.a{n}))) / m;
63. end
64. end

**nnbp**

代码：\NN\nnbp.m

nnbp呢是进行back propagation的过程，过程还是比较中规中矩，和ufldl中的[Neural Network](http://ufldl.stanford.edu/wiki/index.php/Backpropagation_Algorithm)讲的基本一致

值得注意的还是dropout和sparsity的部分

**[cpp]** [view plaincopy](http://blog.csdn.net/dark_scope/article/details/9421061)

1. **if**(nn.nonSparsityPenalty>0)
2. pi = repmat(nn.p{i}, size(nn.a{i}, 1), 1);
3. sparsityError = [zeros(size(nn.a{i},1),1) nn.nonSparsityPenalty \* (-nn.sparsityTarget ./ pi + (1 - nn.sparsityTarget) ./ (1 - pi))];
4. end
6. // Backpropagate first derivatives
7. **if** i+1==n % in **this** **case** in d{n} there is not the bias term to be removed
8. d{i} = (d{i + 1} \* nn.W{i} + sparsityError) .\* d\_act; // Bishop (5.56)
9. **else** // in this case in d{i} the bias term has to be removed
10. d{i} = (d{i + 1}(:,2:end) \* nn.W{i} + sparsityError) .\* d\_act;
11. end
13. **if**(nn.dropoutFraction>0)
14. d{i} = d{i} .\* [ones(size(d{i},1),1) nn.dropOutMask{i}];
15. end

这只是实现的内容，代码中的d{i}就是这一层的delta值，在ufldl中有讲的

dW{i}基本就是计算的gradient了，只是后面还要加入一些东西，进行一些修改

具体原理参见论文“Improving Neural Networks with Dropout“ 以及 [Autoencoders and Sparsity](http://ufldl.stanford.edu/wiki/index.php/Autoencoders_and_Sparsity)的内容

**nnapplygrads**

代码文件：\NN\nnapplygrads.m

**[cpp]** [view plaincopy](http://blog.csdn.net/dark_scope/article/details/9421061)

1. **for** i = 1 : (nn.n - 1)
2. **if**(nn.weightPenaltyL2>0)
3. dW = nn.dW{i} + nn.weightPenaltyL2 \* nn.W{i};
4. **else**
5. dW = nn.dW{i};
6. end
8. dW = nn.learningRate \* dW;
10. **if**(nn.momentum>0)
11. nn.vW{i} = nn.momentum\*nn.vW{i} + dW;
12. dW = nn.vW{i};
13. end
15. nn.W{i} = nn.W{i} - dW;
16. end

这个内容就简单了，nn.weightPenaltyL2 是weight decay的部分，也是nnsetup时可以设置的一个参数

有的话就加入weight Penalty，防止过拟合，然后再根据momentum的大小调整一下，最后改变nn.W{i}即可

## nntest

nntest再简单不过了，就是调用一下nnpredict，在和test的集合进行比较

**[cpp]** [view plaincopy](http://blog.csdn.net/dark_scope/article/details/9421061)

1. function [er, bad] = nntest(nn, x, y)
2. labels = nnpredict(nn, x);
3. [~, expected] = max(y,[],2);
4. bad = find(labels ~= expected);
5. er = numel(bad) / size(x, 1);
6. end

**nnpredict**

代码文件：\NN\nnpredict.m

**[cpp]** [view plaincopy](http://blog.csdn.net/dark_scope/article/details/9421061)

1. function labels = nnpredict(nn, x)
2. nn.testing = 1;
3. nn = nnff(nn, x, zeros(size(x,1), nn.size(end)));
4. nn.testing = 0;
6. [~, i] = max(nn.a{end},[],2);
7. labels = i;
8. end

继续非常简单，predict不过是nnff一次，得到最后的output~~

max(nn.a{end},[],2); 是返回每一行的最大值以及所在的列数，所以labels返回的就是标号啦

(这个test好像是专门用来test 分类问题的，我们知道nnff得到最后的值即可)

## 总结

   总的来说，神经网络的代码比较常规易理解，基本上和[UFLDL](http://ufldl.stanford.edu/wiki/index.php/UFLDL_Tutorial)中的内容相差不大

   只是加入了dropout的部分和denoising的部分

   本文的目的也不奢望讲清楚这些东西，只是给出一个路线，可以跟着代码去学习，加深对算法的理解和应用能力

# [【面向代码】学习 Deep Learning（二）Deep Belief Nets(DBNs)](http://blog.csdn.net/dark_scope/article/details/9447967)

最近一直在看Deep Learning，各类博客、论文看得不少

但是说实话，这样做有些疏于实现，一来呢自己的电脑也不是很好，二来呢我目前也没能力自己去写一个toolbox

只是跟着Andrew Ng的[UFLDL tutorial](http://ufldl.stanford.edu/wiki/index.php/UFLDL_Tutorial) 写了些已有框架的代码(这部分的代码见[github](https://github.com/justdark/matlab_code-ufldl-exercise-))

后来发现了一个matlab的Deep Learning的[toolbox](https://github.com/rasmusbergpalm/DeepLearnToolbox)，发现其代码很简单，感觉比较适合用来学习算法

再一个就是matlab的实现可以省略掉很多数据结构的代码，使算法思路非常清晰

所以我想在解读这个toolbox的代码的同时来巩固自己学到的，同时也为下一步的实践打好基础

(本文只是从代码的角度解读算法，具体的算法理论步骤还是需要去看paper的

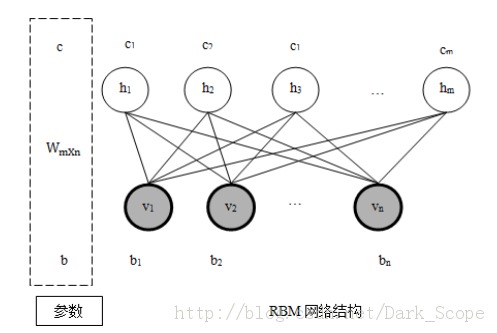
我会在文中给出一些相关的paper的名字，本文旨在梳理一下算法过程，不会深究算法原理和公式)

==========================================================================================

使用的代码：[**DeepLearnToolbox**](https://github.com/rasmusbergpalm/DeepLearnToolbox)  ，下载地址：[点击打开](https://github.com/rasmusbergpalm/DeepLearnToolbox)，感谢该toolbox的作者

==========================================================================================

今天介绍DBN的内容，其中关键部分都是(Restricted Boltzmann Machines, RBM)的步骤，所以先放一张rbm的结构，帮助理解

(图来自baidu的一个讲解ppt)

==========================================================================================

照例，我们首先来看一个完整的DBN的例子程序：

这是\tests\test\_example\_DBN.m 中的ex2

**[cpp]** [view plaincopy](http://blog.csdn.net/dark_scope/article/details/9447967)

1. //train dbn
2. dbn.sizes = [100 100];
3. opts.numepochs =   1;
4. opts.batchsize = 100;
5. opts.momentum  =   0;
6. opts.alpha     =   1;
7. dbn =dbnsetup(dbn, train\_x, opts);                //here！！！
8. dbn = dbntrain(dbn, train\_x, opts);                //here！！！
10. //unfold dbn to nn
11. nn = dbnunfoldtonn(dbn, 10);                       //here！！！
12. nn.activation\_function = 'sigm';
14. //train nn
15. opts.numepochs =  1;
16. opts.batchsize = 100;
17. nn = nntrain(nn, train\_x, train\_y, opts);
18. [er, bad] = nntest(nn, test\_x, test\_y);
19. assert(er < 0.10, 'Too big error');

其中的过程简单清晰明了，就是dbnsetup(),dbntrain()以及dbnunfoldtonn()三个函数

最后fine tuning的时候用了（一）里看过的nntrain和nntest，参见[（一）](http://blog.csdn.net/dark_scope/article/details/9421061)

## \DBN\dbnsetup.m

     这个实在没什么好说的，

     直接分层初始化每一层的rbm(受限波尔兹曼机(Restricted Boltzmann Machines, RBM))  
     同样，W,b,c是参数，vW,vb,vc是更新时用到的与momentum的变量，见到代码时再说

**[cpp]** [view plaincopy](http://blog.csdn.net/dark_scope/article/details/9447967)

1. **for** u = 1 : numel(dbn.sizes) - 1
2. dbn.rbm{u}.alpha    = opts.alpha;
3. dbn.rbm{u}.momentum = opts.momentum;
5. dbn.rbm{u}.W  = zeros(dbn.sizes(u + 1), dbn.sizes(u));
6. dbn.rbm{u}.vW = zeros(dbn.sizes(u + 1), dbn.sizes(u));
8. dbn.rbm{u}.b  = zeros(dbn.sizes(u), 1);
9. dbn.rbm{u}.vb = zeros(dbn.sizes(u), 1);
11. dbn.rbm{u}.c  = zeros(dbn.sizes(u + 1), 1);
12. dbn.rbm{u}.vc = zeros(dbn.sizes(u + 1), 1);
13. end

## \DBN\dbntrain.m

     应为DBN基本就是把rbm当做砖块搭建起来的，所以train也很简单

**[cpp]** [view plaincopy](http://blog.csdn.net/dark_scope/article/details/9447967)

1. function dbn = dbntrain(dbn, x, opts)
2. n = numel(dbn.rbm);
3. //对每一层的rbm进行训练
4. dbn.rbm{1} = rbmtrain(dbn.rbm{1}, x, opts);
5. **for** i = 2 : n
6. x = rbmup(dbn.rbm{i - 1}, x);
7. dbn.rbm{i} = rbmtrain(dbn.rbm{i}, x, opts);
8. end
9. end

  首先映入眼帘的是对第一层进行rbmtrain()，后面每一层在train之前用了rbmup，

  rbmup其实就是简单的一句sigm(repmat(rbm.c', size(x, 1), 1) + x \* rbm.W');

  也就是上面那张图从v到h计算一次，公式是Wx+c

   接下来是最关键的rbmtrain了：

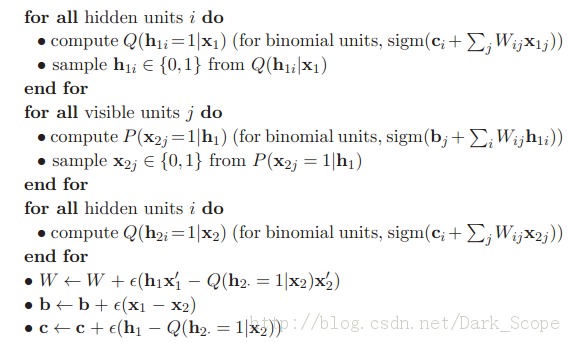
## \DBN\rbmtrain.m

        代码如下，说明都在注释里

         论文参考：【1】Learning Deep Architectures for AI   以及

                          【2】A Practical Guide to Training Restricted Boltzmann Machines

         你可以和【1】里面的这段伪代码对应一下



**[cpp]** [view plaincopy](http://blog.csdn.net/dark_scope/article/details/9447967)

1. **for** i = 1 : opts.numepochs //迭代次数
2. kk = randperm(m);
3. err = 0;
4. **for** l = 1 : numbatches
5. batch = x(kk((l - 1) \* opts.batchsize + 1 : l \* opts.batchsize), :);
7. v1 = batch;
8. h1 = sigmrnd(repmat(rbm.c', opts.batchsize, 1) + v1 \* rbm.W');            //gibbs sampling的过程
9. v2 = sigmrnd(repmat(rbm.b', opts.batchsize, 1) + h1 \* rbm.W);
10. h2 = sigm(repmat(rbm.c', opts.batchsize, 1) + v2 \* rbm.W');
11. //Contrastive Divergence 的过程
12. //这和《Learning Deep Architectures for AI》里面写cd-1的那段pseudo code是一样的
13. c1 = h1' \* v1;
14. c2 = h2' \* v2;
15. //关于momentum，请参看Hinton的《A Practical Guide to Training Restricted Boltzmann Machines》
16. //它的作用是记录下以前的更新方向，并与现在的方向结合下，跟有可能加快学习的速度
17. rbm.vW = rbm.momentum \* rbm.vW + rbm.alpha \* (c1 - c2)     / opts.batchsize;
18. rbm.vb = rbm.momentum \* rbm.vb + rbm.alpha \* sum(v1 - v2)' / opts.batchsize;
19. rbm.vc = rbm.momentum \* rbm.vc + rbm.alpha \* sum(h1 - h2)' / opts.batchsize;
20. //更新值
21. rbm.W = rbm.W + rbm.vW;
22. rbm.b = rbm.b + rbm.vb;
23. rbm.c = rbm.c + rbm.vc;
25. err = err + sum(sum((v1 - v2) .^ 2)) / opts.batchsize;
26. end
27. end

## \DBN\dbnunfoldtonn.m

      DBN的每一层训练完成后自然还要把参数传递给一个大的NN，这就是这个函数的作用

**[cpp]** [view plaincopy](http://blog.csdn.net/dark_scope/article/details/9447967)

1. function nn = dbnunfoldtonn(dbn, outputsize)
2. %DBNUNFOLDTONN Unfolds a DBN to a NN
3. %   outputsize是你的目标输出label，比如在MINST就是10，DBN只负责学习feature
4. %   或者说初始化Weight，是一个unsupervised learning，最后的supervised还得靠NN
5. **if**(exist('outputsize','var'))
6. size = [dbn.sizes outputsize];
7. **else**
8. size = [dbn.sizes];
9. end
10. nn = nnsetup(size);
11. %把每一层展开后的Weight拿去初始化NN的Weight
12. %注意dbn.rbm{i}.c拿去初始化了bias项的值
13. **for** i = 1 : numel(dbn.rbm)
14. nn.W{i} = [dbn.rbm{i}.c dbn.rbm{i}.W];
15. end
16. end

最后fine tuning就再训练一下NN就可以了

## 总结

      还是那句话，本文只是梳理一下学习路线，具体的东西还是要靠paper

      dbn主要的关键就是rbm，推荐几篇经典的文章吧，rbm可是Hinton的宝贝啊

      其中涉及到MCMC，Contrastive divergence，感觉比Autoencoder难理解多了

          [1] An Introduction to Restricted Boltzmann Machines

          [2] Learning Deep Architectures for AI                                                     Bengio大作啊

          [3] A Practical Guide to Training Restricted Boltzmann Machines              上面提到过，比较细致

          [4] A learning Algorithm for Boltzmann Machines                                      Hinton的

# [【面向代码】学习 Deep Learning（三）Convolution Neural Network(CNN)](http://blog.csdn.net/dark_scope/article/details/9495505)

最近一直在看Deep Learning，各类博客、论文看得不少

但是说实话，这样做有些疏于实现，一来呢自己的电脑也不是很好，二来呢我目前也没能力自己去写一个toolbox

只是跟着Andrew Ng的[UFLDL tutorial](http://ufldl.stanford.edu/wiki/index.php/UFLDL_Tutorial) 写了些已有框架的代码(这部分的代码见[github](https://github.com/justdark/matlab_code-ufldl-exercise-))

后来发现了一个matlab的Deep Learning的[toolbox](https://github.com/rasmusbergpalm/DeepLearnToolbox)，发现其代码很简单，感觉比较适合用来学习算法

再一个就是matlab的实现可以省略掉很多数据结构的代码，使算法思路非常清晰

所以我想在解读这个toolbox的代码的同时来巩固自己学到的，同时也为下一步的实践打好基础

(本文只是从代码的角度解读算法，具体的算法理论步骤还是需要去看paper的

我会在文中给出一些相关的paper的名字，本文旨在梳理一下算法过程，不会深究算法原理和公式)

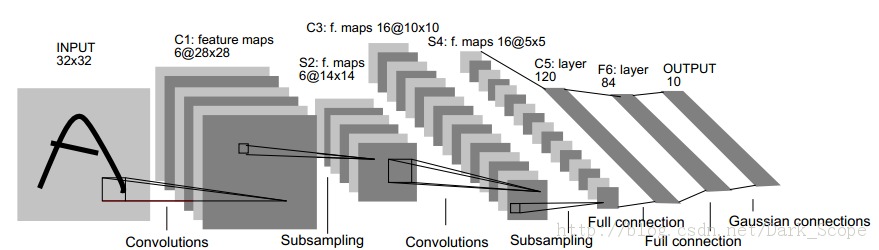
==========================================================================================

使用的代码：[**DeepLearnToolbox**](https://github.com/rasmusbergpalm/DeepLearnToolbox)  ，下载地址：[点击打开](https://github.com/rasmusbergpalm/DeepLearnToolbox)，感谢该toolbox的作者

==========================================================================================

今天是CNN的内容啦，CNN讲起来有些纠结，你可以事先看看[convolution](http://deeplearning.stanford.edu/wiki/index.php/Feature_extraction_using_convolution)和[pooling(subsampling)](http://deeplearning.stanford.edu/wiki/index.php/Pooling)，还有这篇：[tornadomeet的博文](http://www.cnblogs.com/tornadomeet/archive/2013/05/05/3061457.html)

下面是那张经典的图：



======================================================================================================

打开\tests\test\_example\_CNN.m一观

**[cpp]** [view plaincopy](http://blog.csdn.net/dark_scope/article/details/9495505)

1. cnn.layers = {
2. **struct**('type', 'i') %input layer
3. **struct**('type', 'c', 'outputmaps', 6, 'kernelsize', 5) %convolution layer
4. **struct**('type', 's', 'scale', 2) %sub sampling layer
5. **struct**('type', 'c', 'outputmaps', 12, 'kernelsize', 5) %convolution layer
6. **struct**('type', 's', 'scale', 2) %subsampling layer
7. };
8. cnn = cnnsetup(cnn, train\_x, train\_y);        //here！！！
9. opts.alpha = 1;
10. opts.batchsize = 50;
11. opts.numepochs = 1;
12. cnn = cnntrain(cnn, train\_x, train\_y, opts);  //here！！！

似乎这次要复杂了一些啊，首先是layer，有三种，i是input，c是convolution，s是subsampling

'c'的outputmaps是convolution之后有多少张图，比如上(最上那张经典的))第一层convolution之后就有六个特征图

'c'的kernelsize 其实就是用来convolution的patch是多大

's'的scale就是pooling的size为scale\*scale的区域

接下来似乎就是常规思路了，cnnsetup()和cnntrain()啦,我们来看代码

## \CNN\cnnsetup.m

主要是一些参数的作用的解释，详细的参看代码里的注释啊

**[cpp]** [view plaincopy](http://blog.csdn.net/dark_scope/article/details/9495505)

1. function net = cnnsetup(net, x, y)
2. inputmaps = 1;
3. mapsize = size(squeeze(x(:, :, 1)));
4. //尤其注意这几个循环的参数的设定
5. //numel(net.layers)  表示有多少层
6. **for** l = 1 : numel(net.layers)   //  layer
7. **if** strcmp(net.layers{l}.type, 's')
8. mapsize = mapsize / net.layers{l}.scale;
9. //subsampling层的mapsize，最开始mapsize是每张图的大小28\*28
10. //这里除以scale，就是pooling之后图的大小，这里为14\*14
11. assert(all(floor(mapsize)==mapsize), ['Layer ' num2str(l) ' size must be integer. Actual: ' num2str(mapsize)]);
12. **for** j = 1 : inputmaps //inputmap就是上一层有多少张特征图，通过初始化为1然后依层更新得到
13. net.layers{l}.b{j} = 0;
14. end
15. end
16. **if** strcmp(net.layers{l}.type, 'c')
17. mapsize = mapsize - net.layers{l}.kernelsize + 1;
18. //这里的mapsize可以参见UFLDL里面的那张图下面的解释
19. fan\_out = net.layers{l}.outputmaps \* net.layers{l}.kernelsize ^ 2;
20. //隐藏层的大小，是一个(后层特征图数量)\*(用来卷积的patch图的大小)
21. **for** j = 1 : net.layers{l}.outputmaps  //  output map
22. fan\_in = inputmaps \* net.layers{l}.kernelsize ^ 2;
23. //对于每一个后层特征图，有多少个参数链到前层
24. **for** i = 1 : inputmaps  //  input map
25. net.layers{l}.k{i}{j} = (rand(net.layers{l}.kernelsize) - 0.5) \* 2 \* sqrt(6 / (fan\_in + fan\_out));
26. end
27. net.layers{l}.b{j} = 0;
28. end
29. inputmaps = net.layers{l}.outputmaps;
30. end
31. end
32. // 'onum' is the number of labels, that's why it is calculated using size(y, 1). If you have 20 labels so the output of the network will be 20 neurons.
33. // 'fvnum' is the number of output neurons at the last layer, the layer just before the output layer.
34. // 'ffb' is the biases of the output neurons.
35. // 'ffW' is the weights between the last layer and the output neurons. Note that the last layer is fully connected to the output layer, that's why the size of the weights is (onum \* fvnum)
36. fvnum = prod(mapsize) \* inputmaps;
37. onum = size(y, 1);
38. //这里是最后一层神经网络的设定
39. net.ffb = zeros(onum, 1);
40. net.ffW = (rand(onum, fvnum) - 0.5) \* 2 \* sqrt(6 / (onum + fvnum));
41. end

## \CNN\cnntrain.m

 cnntrain就和nntrain是一个节奏了：

**[cpp]** [view plaincopy](http://blog.csdn.net/dark_scope/article/details/9495505)

1. net = cnnff(net, batch\_x);
2. net = cnnbp(net, batch\_y);
3. net = cnnapplygrads(net, opts);

cnntrain是用back propagation来计算gradient的，我们一次来看这三个函数：

**cnnff.m**

这部分计算还比较简单，可以说是有迹可循，大家最好看看[tornadomeet的博文](http://www.cnblogs.com/tornadomeet/archive/2013/05/05/3061457.html)的步骤，说得比较清楚

**[cpp]** [view plaincopy](http://blog.csdn.net/dark_scope/article/details/9495505)

1. function net = cnnff(net, x)
2. n = numel(net.layers);
3. net.layers{1}.a{1} = x;
4. inputmaps = 1;
6. **for** l = 2 : n   //  for each layer
7. **if** strcmp(net.layers{l}.type, 'c')
8. //  !!below can probably be handled by insane matrix operations
9. **for** j = 1 : net.layers{l}.outputmaps   //  for each output map
10. //  create temp output map
11. z = zeros(size(net.layers{l - 1}.a{1}) - [net.layers{l}.kernelsize - 1 net.layers{l}.kernelsize - 1 0]);
12. **for** i = 1 : inputmaps   //  for each input map
13. //  convolve with corresponding kernel and add to temp output map
14. //  做卷积，参考UFLDL，这里是对每一个input的特征图做一次卷积，再加起来
15. z = z + convn(net.layers{l - 1}.a{i}, net.layers{l}.k{i}{j}, 'valid');
16. end
17. //  add bias, pass through nonlinearity
18. //  加入bias
19. net.layers{l}.a{j} = sigm(z + net.layers{l}.b{j});
20. end
21. //  set number of input maps to this layers number of outputmaps
22. inputmaps = net.layers{l}.outputmaps;
23. elseif strcmp(net.layers{l}.type, 's')
24. //  downsample
25. **for** j = 1 : inputmaps
26. //这里有点绕绕的，它是新建了一个patch来做卷积，但我们要的是pooling，所以它跳着把结果读出来，步长为scale
27. //这里做的是mean-pooling
28. z = convn(net.layers{l - 1}.a{j}, ones(net.layers{l}.scale) / (net.layers{l}.scale ^ 2), 'valid');   //  !! replace with variable
29. net.layers{l}.a{j} = z(1 : net.layers{l}.scale : end, 1 : net.layers{l}.scale : end, :);
30. end
31. end
32. end
33. //  收纳到一个vector里面，方便后面用~~
34. //  concatenate all end layer feature maps into vector
35. net.fv = [];
36. **for** j = 1 : numel(net.layers{n}.a)
37. sa = size(net.layers{n}.a{j});
38. net.fv = [net.fv; reshape(net.layers{n}.a{j}, sa(1) \* sa(2), sa(3))];
39. end
40. //  最后一层的perceptrons，数据识别的结果
41. net.o = sigm(net.ffW \* net.fv + repmat(net.ffb, 1, size(net.fv, 2)));
43. end

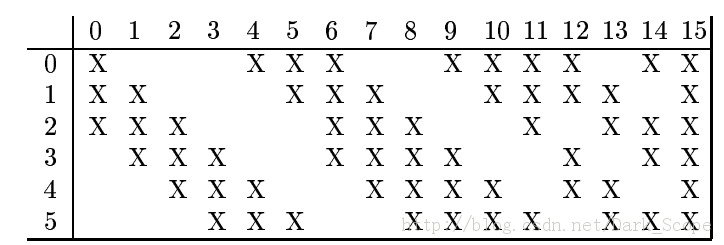
**cnnbp.m**

这个就哭了，代码有些纠结，不得已又找资料看啊，《Notes on Convolutional Neural Networks》要好一些

只是这个toolbox的代码和《Notes on Convolutional Neural Networks》里有些不一样的是这个toolbox在subsampling(也就是pooling层)没有加sigmoid激活函数，只是单纯地pooling了一下，所以这地方还需仔细辨别，这个toolbox里的subsampling是不用计算gradient的，而在Notes里是计算了的

还有这个toolbox没有Combinations of Feature Maps，也就是[tornadomeet的博文](http://www.cnblogs.com/tornadomeet/archive/2013/05/05/3061457.html)里这张表格：

具体就去看看上面这篇论文吧



然后就看代码吧：

**[cpp]** [view plaincopy](http://blog.csdn.net/dark_scope/article/details/9495505)

1. function net = cnnbp(net, y)
2. n = numel(net.layers);
3. //  error
4. net.e = net.o - y;
5. //  loss function
6. net.L = 1/2\* sum(net.e(:) .^ 2) / size(net.e, 2);
7. //从最后一层的error倒推回来deltas
8. //和神经网络的bp有些类似
9. ////  backprop deltas
10. net.od = net.e .\* (net.o .\* (1 - net.o));   //  output delta
11. net.fvd = (net.ffW' \* net.od);              //  feature vector delta
12. **if** strcmp(net.layers{n}.type, 'c')         //  only conv layers has sigm function
13. net.fvd = net.fvd .\* (net.fv .\* (1 - net.fv));
14. end
15. //和神经网络类似，参看神经网络的bp
17. //  reshape feature vector deltas into output map style
18. sa = size(net.layers{n}.a{1});
19. fvnum = sa(1) \* sa(2);
20. **for** j = 1 : numel(net.layers{n}.a)
21. net.layers{n}.d{j} = reshape(net.fvd(((j - 1) \* fvnum + 1) : j \* fvnum, :), sa(1), sa(2), sa(3));
22. end
23. //这是算delta的步骤
24. //这部分的计算参看Notes on Convolutional Neural Networks，其中的变化有些复杂
25. //和这篇文章里稍微有些不一样的是这个toolbox在subsampling(也就是pooling层)没有加sigmoid激活函数
26. //所以这地方还需仔细辨别
27. //这这个toolbox里的subsampling是不用计算gradient的，而在上面那篇note里是计算了的
28. **for** l = (n - 1) : -1 : 1
29. **if** strcmp(net.layers{l}.type, 'c')
30. **for** j = 1 : numel(net.layers{l}.a)
31. 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);
32. end
33. elseif strcmp(net.layers{l}.type, 's')
34. **for** i = 1 : numel(net.layers{l}.a)
35. z = zeros(size(net.layers{l}.a{1}));
36. **for** j = 1 : numel(net.layers{l + 1}.a)
37. z = z + convn(net.layers{l + 1}.d{j}, rot180(net.layers{l + 1}.k{i}{j}), 'full');
38. end
39. net.layers{l}.d{i} = z;
40. end
41. end
42. end
43. //参见paper，注意这里只计算了'c'层的gradient，因为只有这层有参数
44. ////  calc gradients
45. **for** l = 2 : n
46. **if** strcmp(net.layers{l}.type, 'c')
47. **for** j = 1 : numel(net.layers{l}.a)
48. **for** i = 1 : numel(net.layers{l - 1}.a)
49. 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);
50. end
51. net.layers{l}.db{j} = sum(net.layers{l}.d{j}(:)) / size(net.layers{l}.d{j}, 3);
52. end
53. end
54. end
55. //最后一层perceptron的gradient的计算
56. net.dffW = net.od \* (net.fv)' / size(net.od, 2);
57. net.dffb = mean(net.od, 2);
59. function X = rot180(X)
60. X = flipdim(flipdim(X, 1), 2);
61. end
62. end

**cnnapplygrads.m**

  这部分就轻松了，已经有grads了，依次进行梯度更新就好了

**[cpp]** [view plaincopy](http://blog.csdn.net/dark_scope/article/details/9495505)

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. net.layers{l}.k{ii}{j} = net.layers{l}.k{ii}{j} - opts.alpha \* net.layers{l}.dk{ii}{j};
7. end
8. net.layers{l}.b{j} = net.layers{l}.b{j} - opts.alpha \* net.layers{l}.db{j};
9. end
10. end
11. end
13. net.ffW = net.ffW - opts.alpha \* net.dffW;
14. net.ffb = net.ffb - opts.alpha \* net.dffb;
15. end

## cnntest.m

          好吧，我们得知道最后结果怎么来啊

**[cpp]** [view plaincopy](http://blog.csdn.net/dark_scope/article/details/9495505)

1. function [er, bad] = cnntest(net, x, y)
2. //  feedforward
3. net = cnnff(net, x);
4. [~, h] = max(net.o);
5. [~, a] = max(y);
6. bad = find(h ~= a);
8. er = numel(bad) / size(y, 2);
9. end

         就是这样~~cnnff一次后net.o就是结果

## 总结

               just code !

               这是一个89年的模型啊~~~，最近还和RBM结合起来了，做了一个Imagenet的最好成绩(是这个吧？)：

               Alex Krizhevsky.ImageNet Classification with Deep  Convolutional Neural Networks. Video and Slides, 2012  
               http://www.cs.utoronto.ca/~rsalakhu/papers/dbm.pdf

              【参考】：

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

                       【[UFLDL](http://ufldl.stanford.edu/wiki/index.php/UFLDL_Tutorial)】

                       【[Notes on Convolutional Neural Networks](http://cogprints.org/5869/1/cnn_tutorial.pdf)】

                       【[Convolutional Neural Networks (LeNet)](http://deeplearning.net/tutorial/lenet.html)】  这是deeplearning 的theano库的

# [【面向代码】学习 Deep Learning（四） Stacked Auto-Encoders(SAE)](http://blog.csdn.net/dark_scope/article/details/9942775)

最近一直在看Deep Learning，各类博客、论文看得不少

但是说实话，这样做有些疏于实现，一来呢自己的电脑也不是很好，二来呢我目前也没能力自己去写一个toolbox

只是跟着Andrew Ng的[UFLDL tutorial](http://ufldl.stanford.edu/wiki/index.php/UFLDL_Tutorial) 写了些已有框架的代码(这部分的代码见[github](https://github.com/justdark/matlab_code-ufldl-exercise-))

后来发现了一个matlab的Deep Learning的[toolbox](https://github.com/rasmusbergpalm/DeepLearnToolbox)，发现其代码很简单，感觉比较适合用来学习算法

再一个就是matlab的实现可以省略掉很多数据结构的代码，使算法思路非常清晰

所以我想在解读这个toolbox的代码的同时来巩固自己学到的，同时也为下一步的实践打好基础

(本文只是从代码的角度解读算法，具体的算法理论步骤还是需要去看paper的

我会在文中给出一些相关的paper的名字，本文旨在梳理一下算法过程，不会深究算法原理和公式)

==========================================================================================

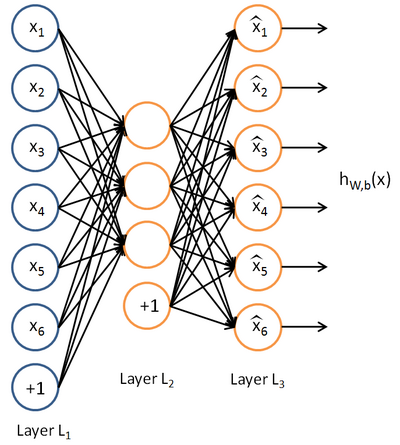
使用的代码：[**DeepLearnToolbox**](https://github.com/rasmusbergpalm/DeepLearnToolbox)  ，下载地址：[点击打开](https://github.com/rasmusbergpalm/DeepLearnToolbox)，感谢该toolbox的作者

==========================================================================================

今天介绍的呢是DL另一个非常重要的模型：SAE

把这个放在最后来说呢，主要是因为在[UFLDL tutorial](http://ufldl.stanford.edu/wiki/index.php/UFLDL_Tutorial) 里已经介绍得比较详细了，二来代码非常简单(在NN的基础之上)

先放一张autoencoder的基本结构：



基本意思就是一个隐藏层的神经网络，输入输出都是x，属于无监督学习

==========================================================================================

## 基本代码

saesetup.m

**[cpp]** [view plaincopy](http://blog.csdn.net/dark_scope/article/details/9942775)

1. function sae = saesetup(size)
2. **for** u = 2 : numel(size)
3. sae.ae{u-1} = nnsetup([size(u-1) size(u) size(u-1)]);
4. end
5. end

saetrain.m

**[cpp]** [view plaincopy](http://blog.csdn.net/dark_scope/article/details/9942775)

1. function sae = saetrain(sae, x, opts)
2. **for** i = 1 : numel(sae.ae);
3. disp(['Training AE ' num2str(i) '/' num2str(numel(sae.ae))]);
4. sae.ae{i} = nntrain(sae.ae{i}, x, x, opts);
5. t = nnff(sae.ae{i}, x, x);
6. x = t.a{2};
7. %remove bias term
8. x = x(:,2:end);
9. end
10. end

其实就是每一层一个autoencoder，隐藏层的值作为下一层的输入

## 各类变形

    为了不致于本文内容太少。。。现在单独把它的几个变形提出来说说

**sparse autoencoder：**

http://ufldl.stanford.edu/wiki/images/math/7/a/4/7a4ac86b3559db835f4357987252b088.png

   这就是ufldl讲的版本，toolbox中的代码和ufldl中练习的部分基本一致：

   在nnff.m中使用：nn.p{i} = 0.99 \* nn.p{i} + 0.01 \* mean(nn.a{i}, 1);计算

   在nnbp.m中使用

     pi = repmat(nn.p{i}, size(nn.a{i}, 1), 1);

     sparsityError = [zeros(size(nn.a{i},1),1) nn.nonSparsityPenalty \* (-nn.sparsityTarget ./ pi + (1 - nn.sparsityTarget) ./ (1 - pi))];

   计算sparsityError即可

**denoising autoencoder：**

   denoising其实就是在autoencoder的基础上，给输入的x加入噪声，就相当于dropout用在输入层

   toolbox中的也实现非常简单：

   在nntrain.m中：

      batch\_x = batch\_x.\*(rand(size(batch\_x))>nn.inputZeroMaskedFraction)

   也就是随即把大小为(nn.inputZeroMaskedFraction)的一部分x赋成0，denoising autoencoder的表现好像比sparse autoencoder要强一些

**Contractive Auto-Encoders：**

   这个变形呢是《[Contractive auto-encoders: Explicit invariance during feature extraction](http://machinelearning.wustl.edu/mlpapers/paper_files/ICML2011Rifai_455.pdf)》提出的

   这篇论文里也总结了一下autoencoder，感觉很不错

    Contractive autoencoders的模型是：

http://img.blog.csdn.net/20130813095033406?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvRGFya19TY29wZQ==/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/SouthEast

    其中：

http://img.blog.csdn.net/20130813095151218?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvRGFya19TY29wZQ==/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/SouthEast hj是表示hidden layer的函数，用它对x求导

    论文里说：这个项是

         encourages the mapping to the feature space to be contractive in the neighborhood of the training data

     具体的实现呢是：

http://img.blog.csdn.net/20130813095541312?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvRGFya19TY29wZQ==/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/SouthEast

     代码呢参看：论文作者提供的：[点击打开链接](https://github.com/srifai/cae.py/blob/master/cae/cae.py)

      主要是

**jacobian**(self，x):

**\_jacobi\_loss**():

**\_fit\_reconstruction**():

这几个函数和autoencoder有出入，其实也比较简单，就不细讲了

## 总结：

总的来说，autoencoder感觉是DL中比较好理解的一部分，所以介绍内容不长

可能你也发现了，Toolbox里还有一个文件夹叫CAE，不过这个CAE是Convolutional

Auto-Encoders

参考<http://www.idsia.ch/~ciresan/data/icann2011.pdf>，以后有时间再学习一下~