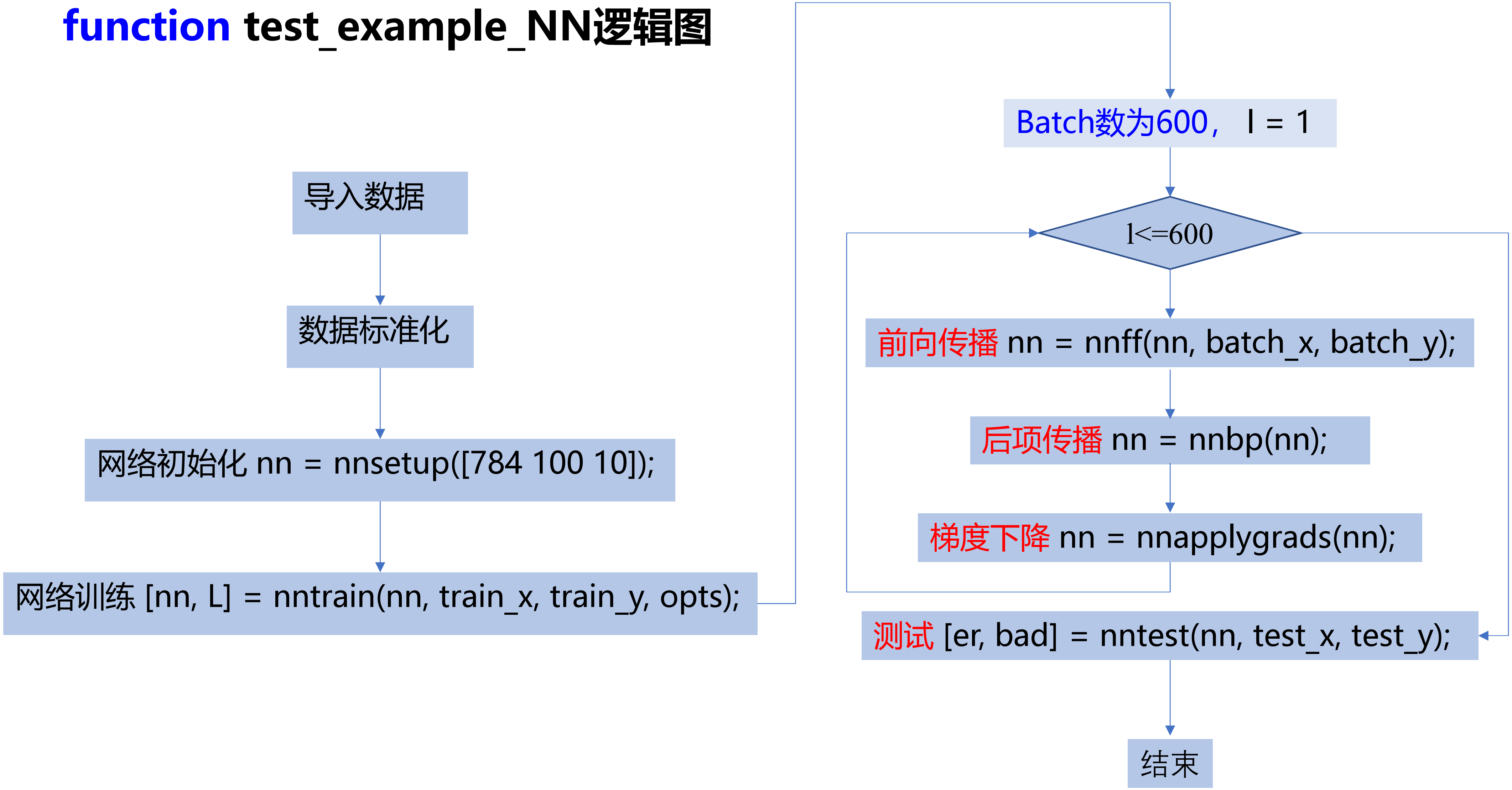
深度网络基本功

第一讲 Neural Network

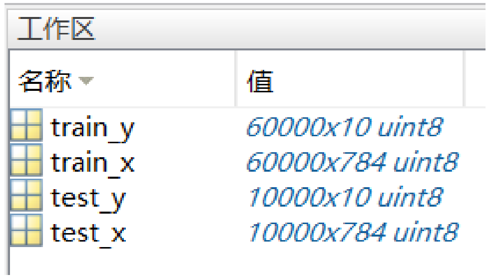
武德安

电子科技大学

function test_example_NN逻辑图



```
function test_example_NN
load mnist_uint8;

train_x = double(train_x) / 255;
test_x  = double(test_x)  / 255;
train_y = double(train_y);
test_y  = double(test_y);

% normalize
[train_x, mu, sigma] = zscore(train_x); %find the mean and variance
test_x = normalize(test_x, mu, sigma);
%% ex1 vanilla neural net
rand('state',0)
nn = nnsetup([784 100 10]); % 初始化
opts.numepochs =  1;   %  Number of full sweeps through data，  ，波数（一波=全体训练数据跑一遍）
opts.batchsize = 100;  %  Take a mean gradient step over this many samples
[nn, L] = nntrain(nn, train_x, train_y, opts); %训练

[er, bad] = nntest(nn, test_x, test_y); %测试

assert(er < 0.08, 'Too big error');
```



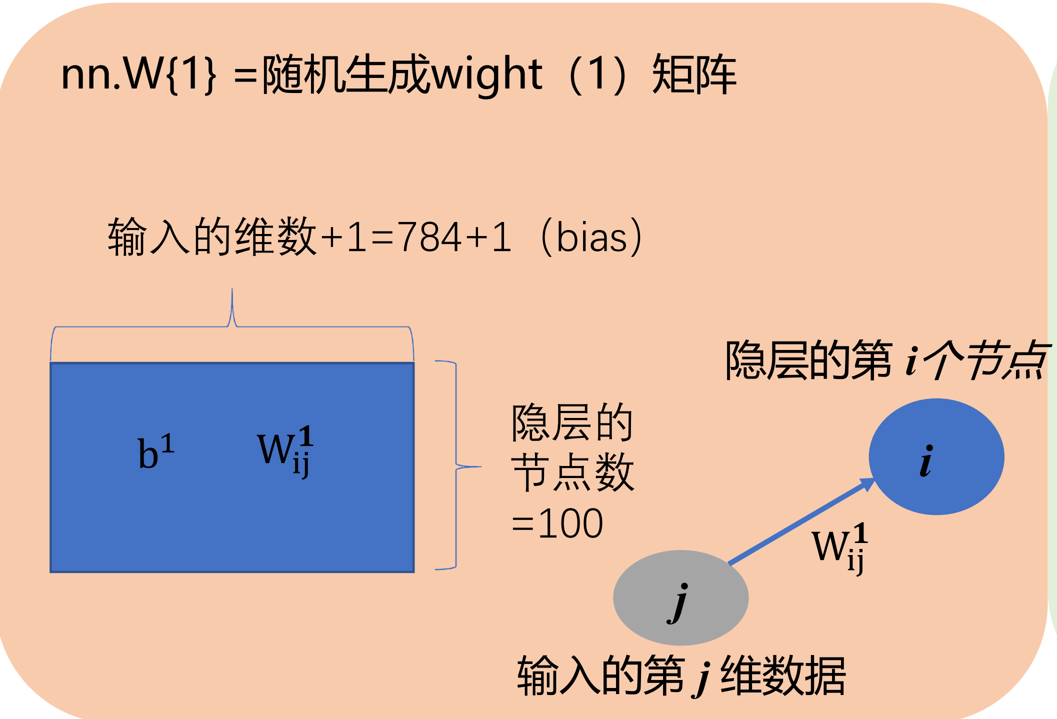
```
function nn = nnsetup(architecture)
%NNSETUP creates a Feedforward Backpropagate
Neural Network
% nn = nnsetup(architecture) returns an neural
network structure with n=numel(architecture)
% layers, architecture being a n x 1 vector of layer
sizes e.g. [784 100 10]
```

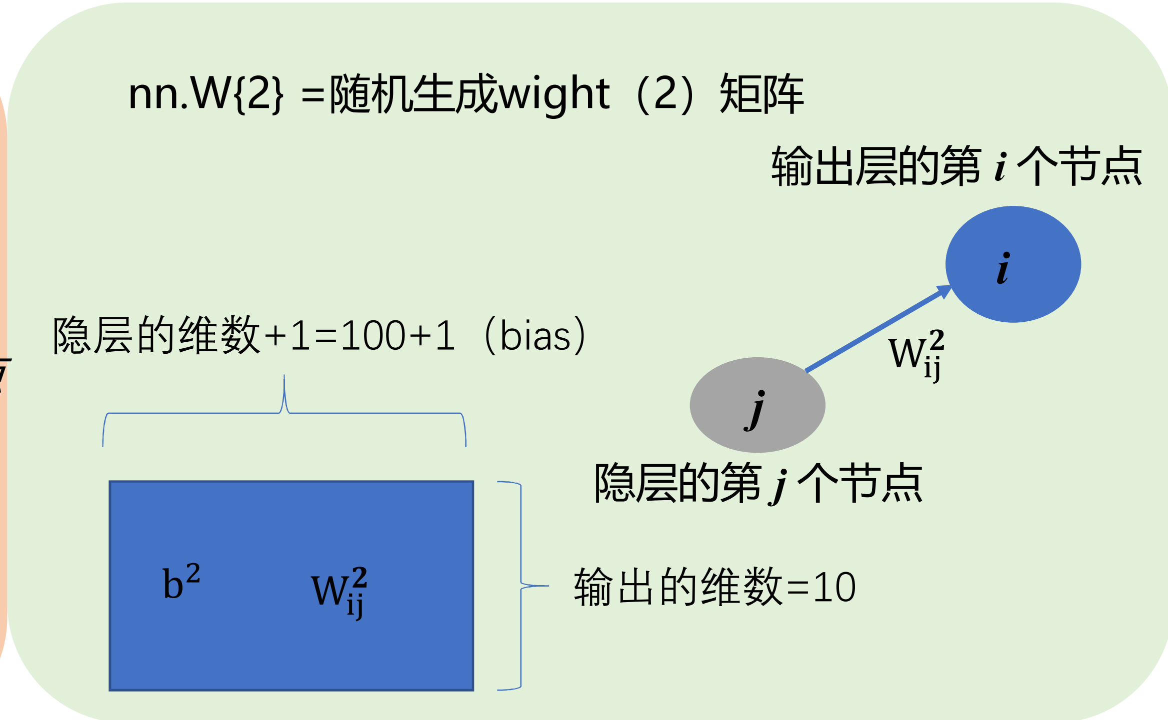
以下定义网络结构

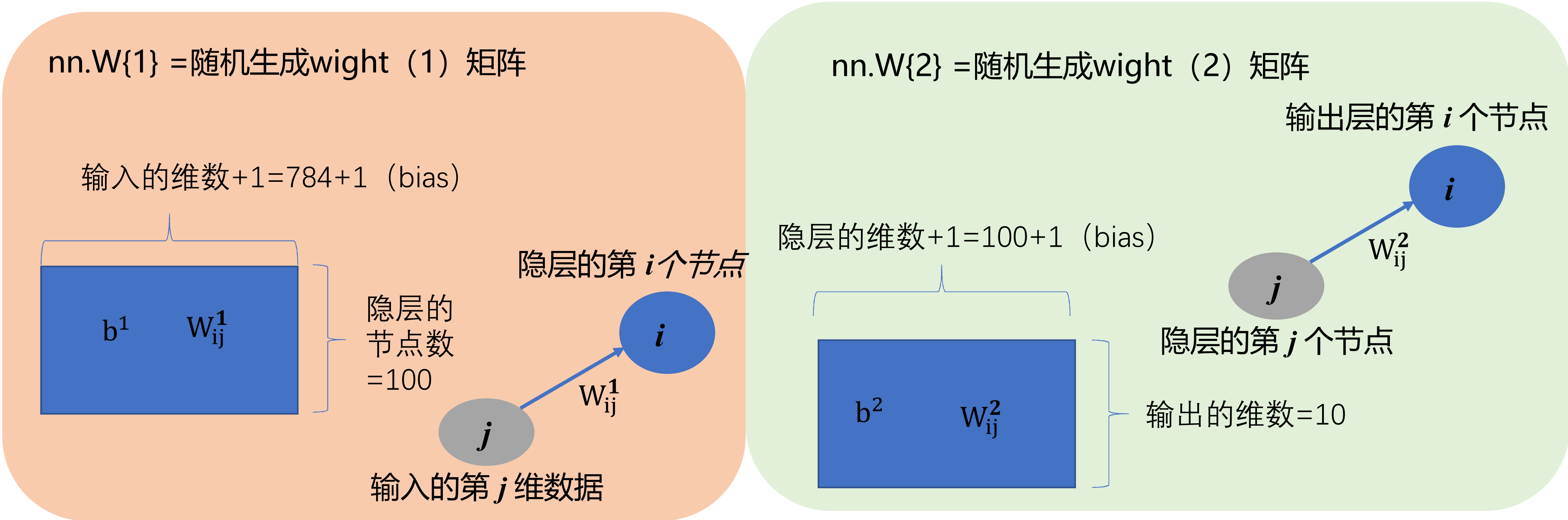
```
nn.size     = architecture;     % architecture= [784 100 10]
nn.n      = numel(nn.size);    % 网络层数=3
nn.activation_function              = 'tanh_opt';   %  Activation functions of hidden layers: 'sigm' (sigmoid) or
'tanh_opt' (optimal tanh).
nn.learningRate                     = 2;            %  learning rate Note: typically needs to be lower when using 'sigm'
activation function and non-normalized inputs.
nn.momentum                         = 0.5;          %  Momentum
nn.scaling_learningRate             = 1;            %  Scaling factor for the learning rate (each epoch)
nn.weightPenaltyL2                  = 0;            %  L2 regularization
nn.nonSparsityPenalty               = 0;            %  Non sparsity penalty
nn.sparsityTarget                   = 0.05;         %  Sparsity target
nn.inputZeroMaskedFraction          = 0;            %  Used for Denoising AutoEncoders
nn.dropoutFraction                  = 0;            %  Dropout level
(http://www.cs.toronto.edu/~hinton/absps/dropout.pdf)
nn.testing                          = 0;            %  Internal variable. nntest sets this to one.
nn.output                           = 'sigm';       %  output unit 'sigm' (=logistic), 'softmax' and 'linear'
```

```
for i = 2 : nn.n  % i=2->3
        % weights and weight momentum
        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)));
        nn.vW{i - 1} = zeros(size(nn.W{i - 1}));

        % average activations (for use with sparsity)
        nn.p{i}     = zeros(1, nn.size(i));
    end
end
```

网络权重初始化





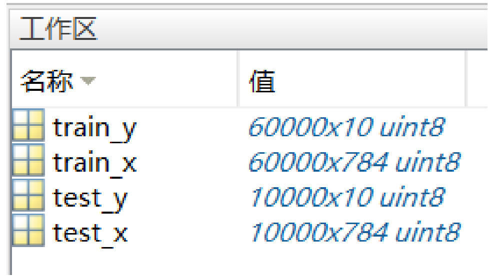


$$W^1 = \begin{matrix} W_{10}^1 & W_{11}^1 & \cdots & W_{1\,784}^1 \\ W_{20}^1 & W_{21}^1 & \cdots & W_{2\,784}^1 \\ W_{100\,0}^1 & W_{100\,0}^1 & \cdots & W_{100\,784}^1 \end{matrix}$$

$$W^2 = \begin{matrix} W_{10}^2 & W_{11}^2 & \cdots & W_{1\,784}^2 \\ W_{20}^2 & W_{21}^2 & \cdots & W_{2\,784}^2 \\ W_{100\,0}^2 & W_{100\,0}^2 & \cdots & W_{100\,784}^2 \end{matrix}$$

```
function [nn, L]  = nntrain(nn, train_x, train_y, opts, val_x, val_y)
%NNTRAIN trains a neural net
% [nn, L] = nnff(nn, x, y, opts) trains the neural network nn with input x and output y for opts.numepochs epochs, with minibatches of size
% opts.batchsize. Returns a neural network nn with updated activations, % errors, weights and biases, (nn.a, nn.e, nn.W, nn.b) and L, the
sum % squared error for each training minibatch.

assert(isfloat(train_x), 'train_x must be a float');
assert(nargin == 4 || nargin == 6,'number ofinput arguments must be 4 or 6')

loss.train.e               = [];
loss.train.e_frac          = [];
loss.val.e                 = [];
loss.val.e_frac            = [];
opts.validation = 0;
if nargin == 6
    opts.validation = 1;
end
fhandle = [];
if isfield(opts,'plot') && opts.plot == 1
    fhandle = figure();
end
```

```
m = size(train_x, 1);   % m=60000, 训练样本的个数

batchsize = opts.batchsize;  % batchsize=100
numepochs = opts.numepochs; %训练的epoch数=1

numbatches = m / batchsize;  % batches的个数=60000/100=600

assert(rem(numbatches, 1) == 0, 'numbatches must be a integer');

L = zeros(numepochs*numbatches,1); %L=[600*1的向量，都为0]
```



```
n = 1;
for i = 1 : numepochs %波数=1
    tic; %开始计时
                                                    共60000个元素
    kk = randperm(m); % 对1-60000做一个随机排列，例如  [2, 300, 56, 37, 4 ,3000,...]

    for l = 1 : numbatches %batch数为600
        batch_x = train_x(kk((l - 1) * batchsize + 1 : l * batchsize), :);
        % l=1时，batch_x=train_x( kk(0*100+1: 1*100), : ) ， 随机取train_x() 矩阵中的100行作为一个batch输入
        % l=2时，batch_x=train_x( kk(1*100+1: 2*100), : ) ， 随机取train_x() 矩阵中的100行作为一个batch输入
        %Add noise to input (for use in denoising autoencoder)
        if(nn.inputZeroMaskedFraction ~= 0)
            batch_x = batch_x.*(rand(size(batch_x))>nn.inputZeroMaskedFraction);
        end
        batch_y = train_y(kk((l - 1) * batchsize + 1 : l * batchsize), :);
        % 对应于batch_x随机样本x的取法，l=1时，batch_y=train_y( kk(0*100+1: 1*100), : ) ， 随机取train_y() 矩阵中的100行
作为一个batch。 l=2时，batch_y=train_y( kk(1*100+1: 2*100), : ) 。
        nn = nnff(nn, batch_x, batch_y); %前向传播
        nn = nnbp(nn); %后项传播
        nn = nnapplygrads(nn);  % 梯度下降
        L(n) = nn.L;
        n = n + 1;
    end
```

工作区

名称	值
train_y	60000x10 uint8
train_x	60000x784 uint8
test_y	10000x10 uint8
test_x	10000x784 uint8

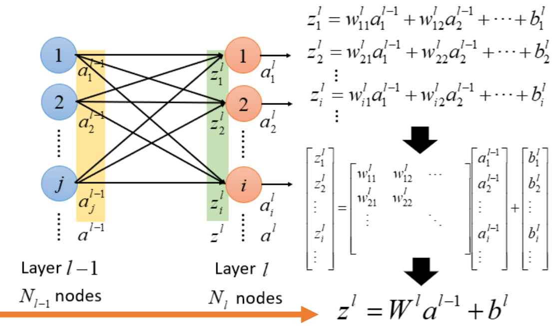
```
t = toc; %停止计时

   if opts.validation == 1 %若需要交叉验证
      loss = nneval(nn, loss, train_x, train_y, val_x, val_y); %评价误差
      str_perf = sprintf('; Full-batch train mse = %f, val mse = %f', loss.train.e(end), loss.val.e(end));
   else
      loss = nneval(nn, loss, train_x, train_y);
      str_perf = sprintf('; Full-batch train err = %f', loss.train.e(end));
   end
   if ishandle(fhandle)
      nnupdatefigures(nn, fhandle, loss, opts, i);
   end

   disp(['epoch ' num2str(i) '/' num2str(opts.numepochs) '. Took ' num2str(t) ' seconds' '. Mini-batch mean squared error on 
training set is ' num2str(mean(L((n-numbatches):(n-1)))) str_perf]);
   nn.learningRate = nn.learningRate * nn.scaling_learningRate;
end
end
```

```
function nn = nnff(nn, x, y) %nn是初始化之后的网络构造，x是100*784的batch_x, y是100*10点batch_y
%NNFF performs a feedforward pass
% nn = nnff(nn, x, y) returns an neural network structure with updated
% layer activations, error and loss (nn.a, nn.e and nn.L)
n = nn.n;  % nn.n=3 网络层数
m = size(x, 1);  %m是batchsize=100
x = [ones(m,1) x]; %在100*784的行列式batch_x的第一列追加100*1的列向量1=[1, 1, 1,…,1]T
nn.a{1} = x; %网络nn的第一层输入为 nn.a{1}=x
```

$$W^1=\begin{bmatrix} W^1_{10} & W^1_{11} & \cdots & W^1_{1\,784} \\ W^1_{20} & W^1_{21} & \cdots & W^1_{2\,784} \\ W^1_{100\,0} & W^1_{100\,0} & \cdots & W^1_{100\,784} \end{bmatrix}$$

```
%feedforward pass
    for i = 2 : n-1  %从2到2，loop内只走一次
        switch nn.activation_function
            case 'sigm'
                % Calculate the unit's outputs (including the bias term)
                nn.a{i} = sigm(nn.a{i - 1} * nn.W{i - 1}'  );
                    %注释：nn.a{2}=sigm（x W(1)T）
            case 'tanh_opt'
                nn.a{i} = tanh_opt(nn.a{i - 1} * nn.W{i - 1}');
        end
```

Relations between Layer Outputs

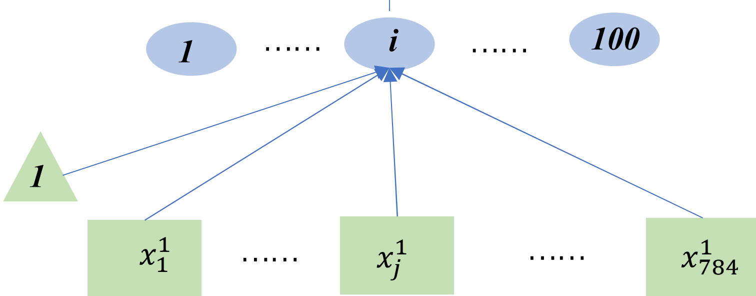


$$\sigma(z_i^1)=\sigma\left(\sum_{j=0}^{784} W_{ij}^1 x_j^1\right)$$

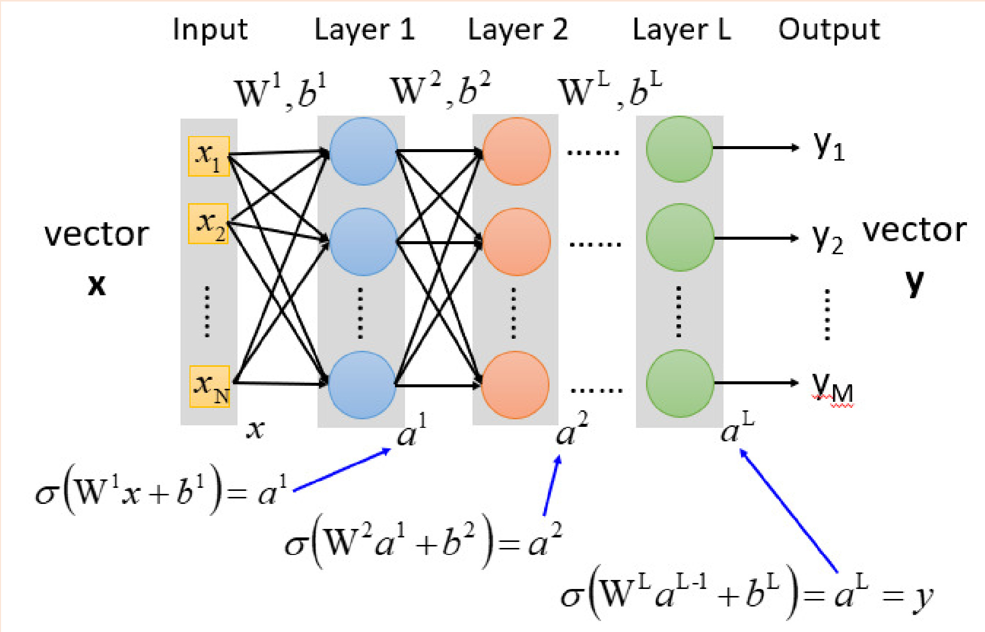
注释：此处两种写法不同的原因是上边李宏毅的写法输入x，或a都用的是列向量。而程序中用的是行向量

$$\begin{bmatrix} 1 & x_1^1 & \cdots & x_{784}^1 \\ \vdots & & \ddots & \vdots \\ 1 & x_1^{100} & \cdots & x_{784}^{100} \end{bmatrix} \begin{bmatrix} W^1_{10} & W^1_{20} & W^1_{100\,0} \\ W^1_{11} & W^1_{21} & W^1_{100\,1} \\ W^1_{1\,784} & W^1_{2\,784} & W^1_{100\,784} \end{bmatrix} = \begin{bmatrix} z_1^1 & z_2^1 & z_{100}^1 \\ z_1^2 & z_2^2 & z_{100}^2 \\ z_1^{100} & z_2^{100} & z_{100}^{100} \end{bmatrix}$$

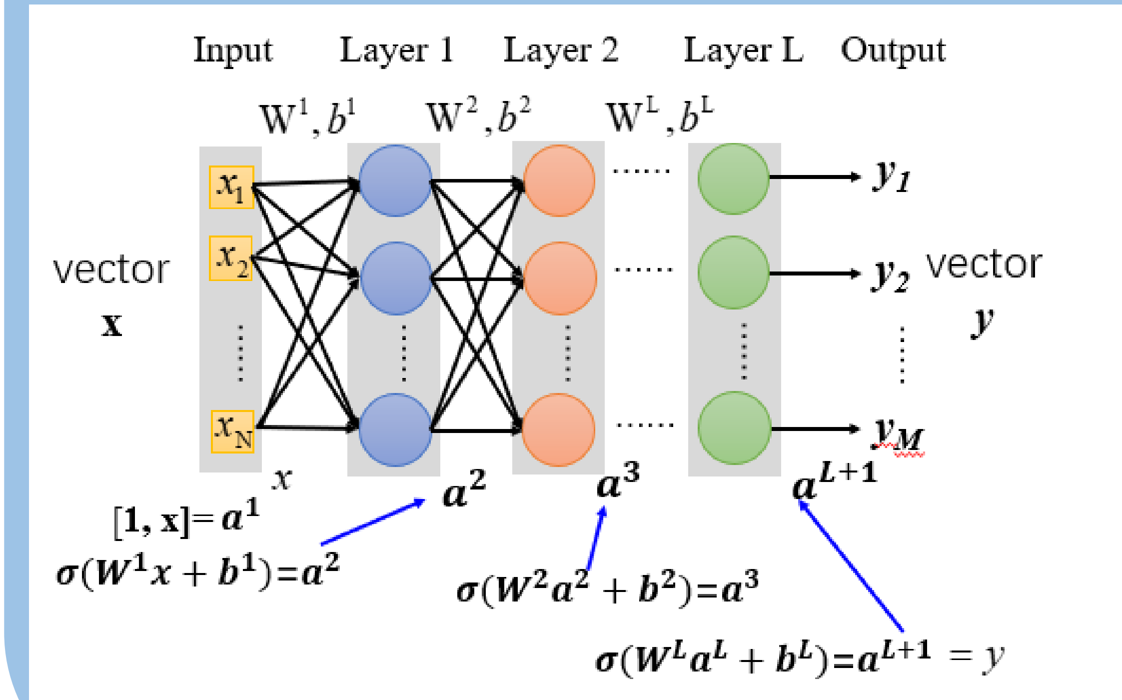
100*785　　785*100 $(W^1)^T$　　100*100



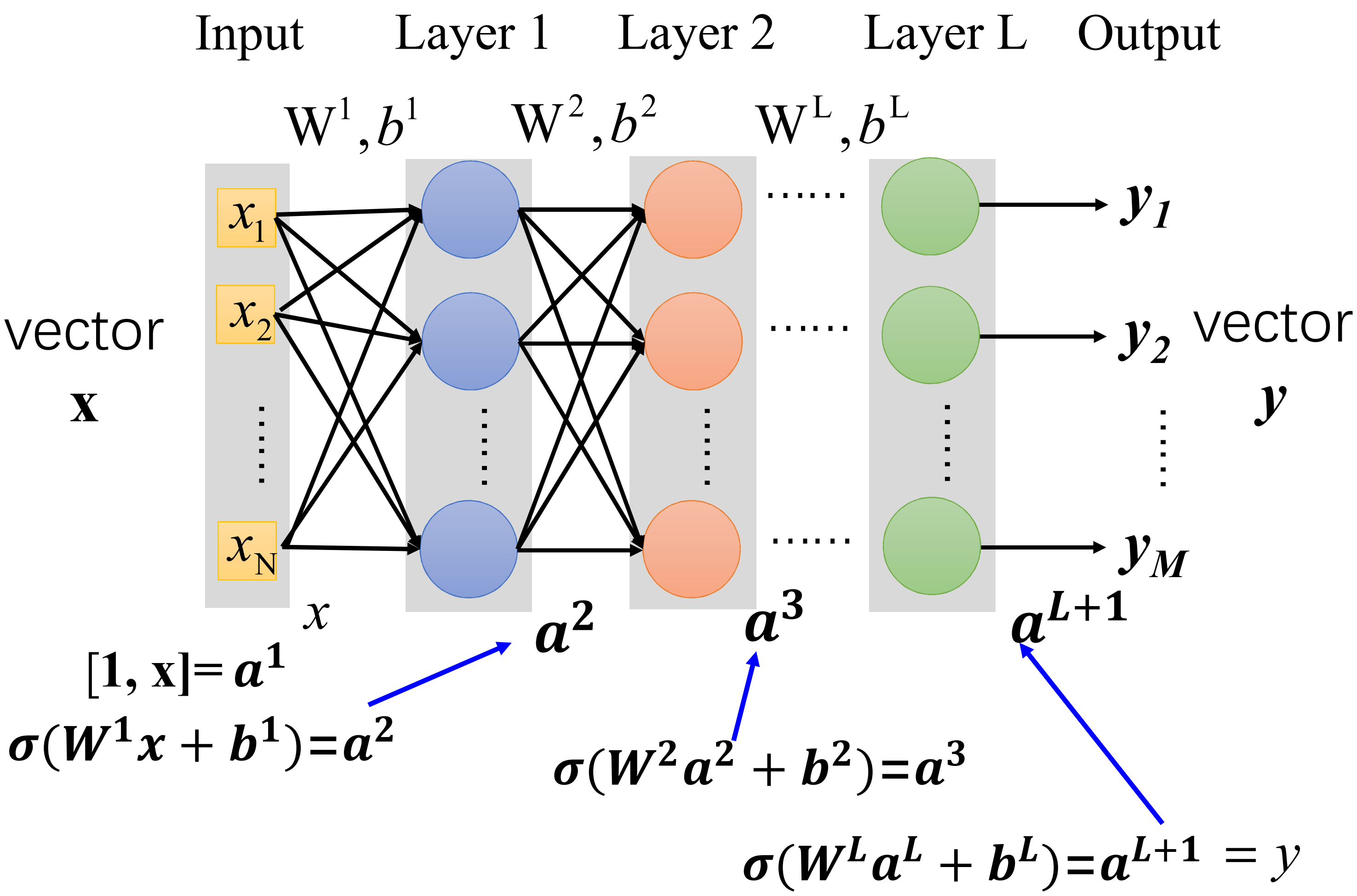
李宏毅的网络符号



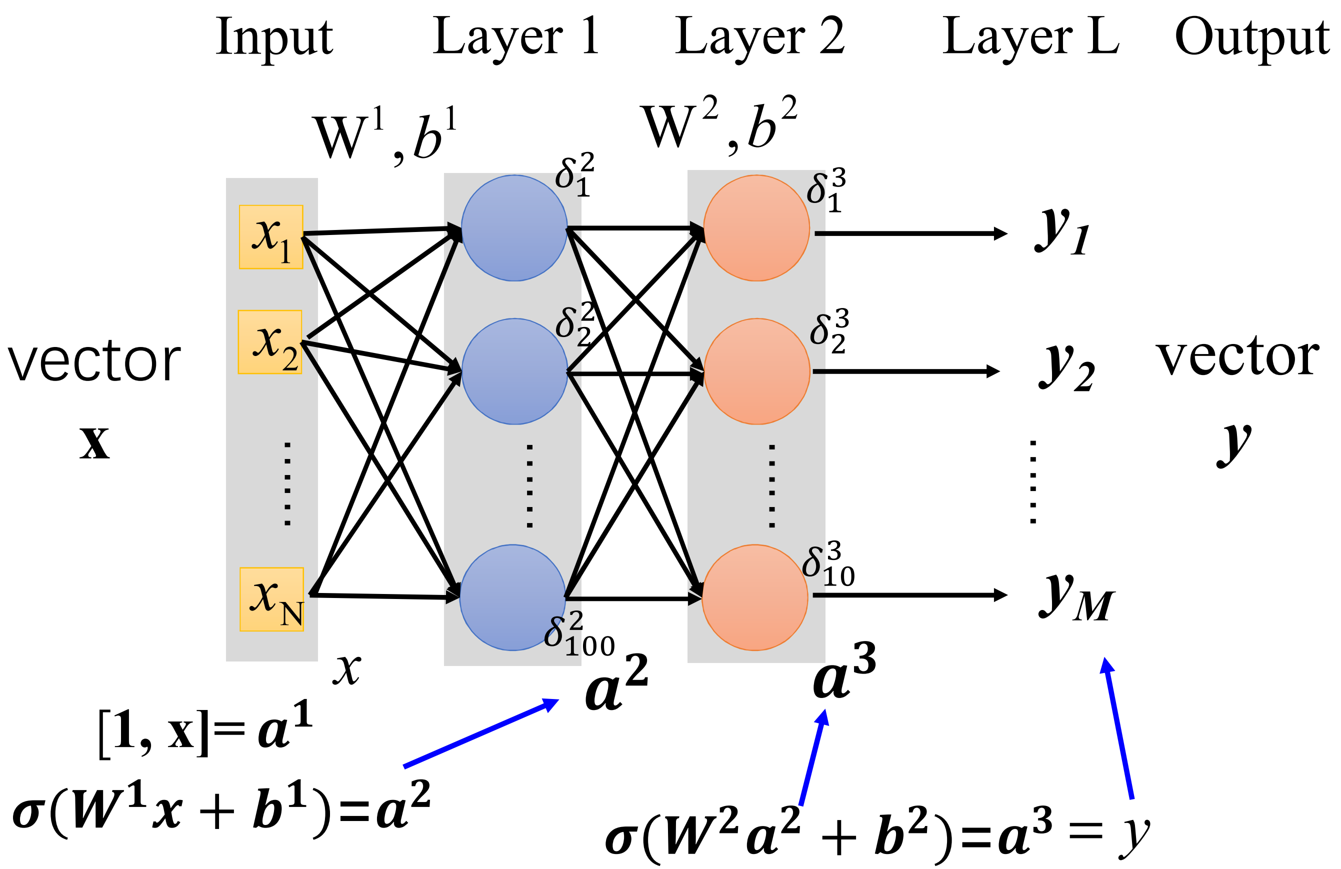
本程序的网络符号



(草稿1)本程序的网络符号



(草稿2)本程序的网络符号



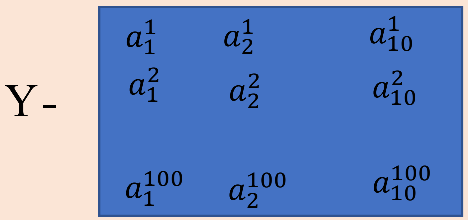
```
%dropout
if(nn.dropoutFraction > 0)
    if(nn.testing)
        nn.a{i} = nn.a{i}.*(1 - nn.dropoutFraction);
    else
        nn.dropOutMask{i} = (rand(size(nn.a{i}))>nn.dropoutFraction);
        nn.a{i} = nn.a{i}.*nn.dropOutMask{i};
    end
end
```

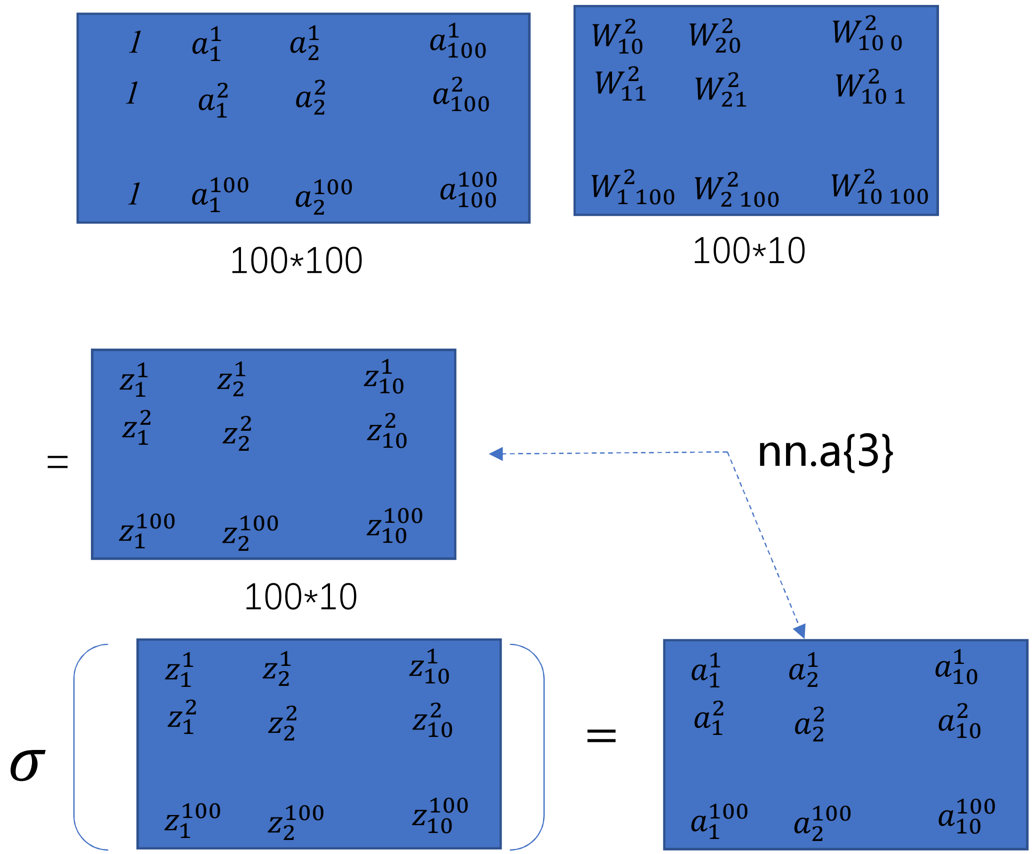
```
 %calculate running exponential activations for use with
sparsity
    if(nn.nonSparsityPenalty>0)
        nn.p{i} = 0.99 * nn.p{i} + 0.01 * mean(nn.a{i}, 1);
    end

    %Add the bias term
    nn.a{i} = [ones(m,1) nn.a{i}]; %在nn.a{2} 的第一列见一列1，
对应bias项，注意这个操作只有在输出项nn.a{3}处不用加

end  %此处前项传递（除了输出层）循环结尾处
```

$$\sigma\left(\begin{matrix} z_1^1 & z_2^1 & z_{100}^1 \\ z_1^2 & z_2^2 & z_{100}^2 \\ & & \\ z_1^{100} & z_2^{100} & z_{100}^{100} \end{matrix}\right) = \begin{matrix} a_1^1 & a_2^1 & a_{100}^1 \\ a_1^2 & a_2^2 & a_{100}^2 \\ & & \\ a_1^{100} & a_2^{100} & a_{100}^{100} \end{matrix}$$

$$\text{nn.a\{i\}} = \begin{matrix} 1 & a_1^1 & a_2^1 & a_{100}^1 \\ 1 & a_1^2 & a_2^2 & a_{100}^2 \\ & & & \\ 1 & a_1^{100} & a_2^{100} & a_{100}^{100} \end{matrix}$$

```
switch nn.output
    case 'sigm'
        nn.a{n} = sigm(nn.a{n - 1} * nn.W{n - 1}'   );% nn.a{3}为输出层
    case 'linear'
        nn.a{n} = nn.a{n - 1} * nn.W{n - 1}';
    case 'softmax'
        nn.a{n} = nn.a{n - 1} * nn.W{n - 1}';
        nn.a{n} = exp(bsxfun(@minus, nn.a{n}, max(nn.a{n},[],2)));
        nn.a{n} = bsxfun(@rdivide, nn.a{n}, sum(nn.a{n}, 2));
        %之所以用以上函数注解见下及后页
end

%error and loss
nn.e = y - nn.a{n}; %误差矩阵

switch nn.output
    case {'sigm', 'linear'}
        nn.L = 1/2 * sum(sum(nn.e .^ 2)) / m;
    case 'softmax'
        nn.L = -sum(sum(y .* log(nn.a{n}))) / m;
end
end
```





Matlab Help: Example: If X = [2 8 4; 7 3 9] then
max(X,[],1) is [7 8 9],
max(X,[],2) is [8; 9] and
max(X,5) is [5 8 5; 7 5 9].

Examples:
If X = [0 1 2; 3 4 5]
then sum(X, 1) is [3 5 7] and sum(X, 2) is [3; 12]
sum([1,2; 3, 4]) is [4 6]
sum(sum([1,2; 3, 4])) is [10]

Computing Log-Sum-Exp

This post is about a computational trick that everyone should know, but that doesn't tend to be explicitly taught in machine learning courses. Imagine that we have a set of N values, $\{x_n\}_{n=1}^{N}$ and we want to compute the quantity

$$z = \log \sum_{n=1}^{N} \exp\{x_n\}. \tag{1}$$

This comes up all the time when you want to parameterize a multinomial distribution using a softmax, e.g., when doing logistic regression and you have more than two unordered categories. If you want to compute the log likelihood, you'll find such an expression due to the normalization constant. Computing this naively can be a recipe for disaster, due to underflow or overflow, depending on the scale of the x_n. Consider a simple example, with the vector $[0\ 1\ 0]$. This seems pretty straightforward, and we get about 1.55. Now what about $[1000\ 1001\ 1000]$. This seems like it should also be straightforward, but instead our computer gives us back $\inf$. If we try $[-1000\ \ -999\ \ -1000]$ we get $-\inf$. What's happening here? Well, in your typical 64-bit double, $\exp\{1000\} = \inf$ and $\exp\{-1000\} = 0$ due to overflow and underflow, respectively. Even though the log would make the numbers reasonably scaled again with infinite precision, it doesn't work on a real computer with typical floating point operations. What to do?

The trick to resolve this issue is pretty simple and exploits the identity:

$$\log \sum_{n=1}^{N} \exp\{x_n\} = a + \log \sum_{n=1}^{N} \exp\{x_n - a\} \tag{2}$$

for any a. This means that you are free to shift the center of the exponentiated variates up and down, however you like. A typical thing to do is to make

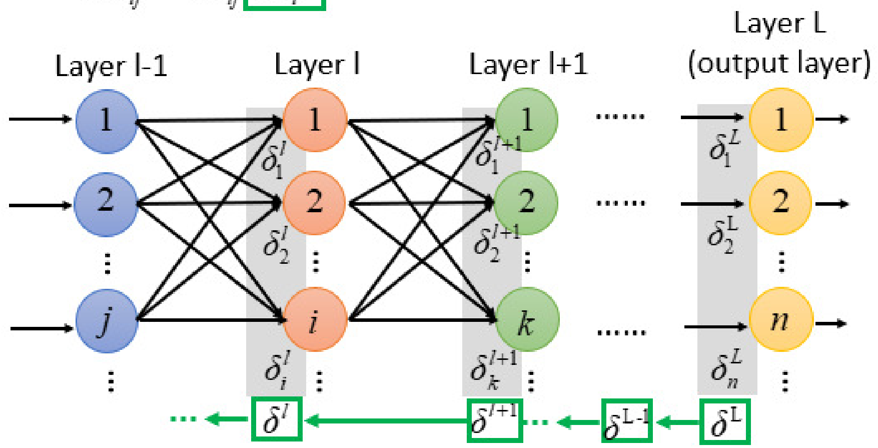
$$a = \max_n x_n, \tag{3}$$

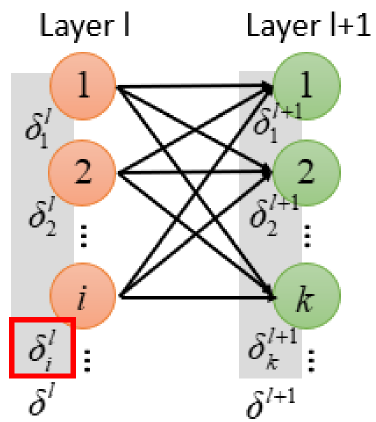
which ensures that the largest value you exponentiate will be zero. This means you will definitely not overflow and even if the rest underflow you will still get a reasonable value.

$\partial C^r / \partial w_{ij}^l$ - Second Term

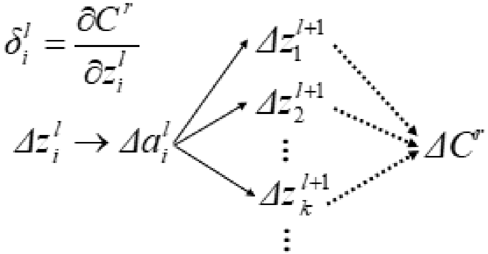
$$\frac{\partial C^r}{\partial w_{ij}^l} = \frac{\partial z_i^l}{\partial w_{ij}^l} \boxed{\frac{\partial C^r}{\partial z_i^l}} \rightarrow \delta_i^l$$

1. How to compute δ^L
2. The relation of δ^l and δ^{l+1}





$$\delta_i^l = \frac{\partial C^r}{\partial z_i^l}$$



$$\delta_i^l = \frac{\partial C^r}{\partial z_i^l} = \frac{\partial a_i^l}{\partial z_i^l} \sum_k \frac{\partial z_k^{l+1}}{\partial a_i^l} \boxed{\frac{\partial C^r}{\partial z_k^{l+1}}} \rightarrow \delta_k^{l+1}$$

```
function nn = nnbp(nn)
%NNBP performs backpropagation
% nn = nnbp(nn) returns an neural network structure with updated weights

    n = nn.n; % nn.n=3 网络层数
    sparsityError = 0;
    switch nn.output
        case 'sigm'
            d{n} = - nn.e .* (nn.a{n} .* (1 - nn.a{n})); %注释： d{n} =delta（3）
        case {'softmax','linear'}
            d{n} = - nn.e;
    end
```

$\partial C^r/\partial w_{ij}^l$ - Second Term

$$\frac{\partial C^r}{\partial w_{ij}^l} = \frac{\partial z_i^l}{\partial w_{ij}^l}\boxed{\frac{\partial C^r}{\partial z_i^l}} \rightarrow \delta_i^l$$

1. How to compute δ^L

2. The relation of δ^l and δ^{l+1}

$$\delta_n^L = \frac{\partial C^r}{\partial z_n^L} = \frac{\partial y_n^r}{\partial z_n^L}\frac{\partial C^r}{\partial y_n^r} = \sigma'(z_n^L)\frac{\partial C^r}{\partial y_n^r}$$

δ^L?

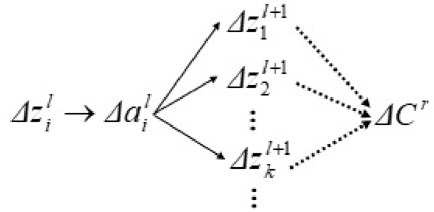
$$\sigma'(z^L) = \begin{bmatrix} \sigma'(z_1^L) \\ \sigma'(z_2^L) \\ \vdots \\ \sigma'(z_n^L) \\ \vdots \end{bmatrix} \qquad \nabla C^r(y^r) = \begin{bmatrix} \partial C^r/\partial y_1^r \\ \partial C^r/\partial y_2^r \\ \vdots \\ \partial C^r/\partial y_n^r \\ \vdots \end{bmatrix}$$

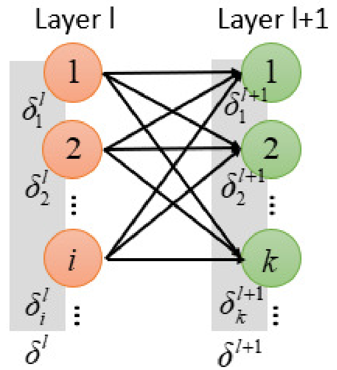
$$\delta^L = \underline{\sigma'(z^l)} \bullet \underline{\nabla C^r(y^r)}$$

element-wise multiplication

$\partial C^r/\partial w_{ij}^l$ - Second Term

$$\frac{\partial C^r}{\partial w_{ij}^l} = \frac{\partial z_i^l}{\partial w_{ij}^l}\boxed{\frac{\partial C^r}{\partial z_i^l}} \rightarrow \delta_i^l$$



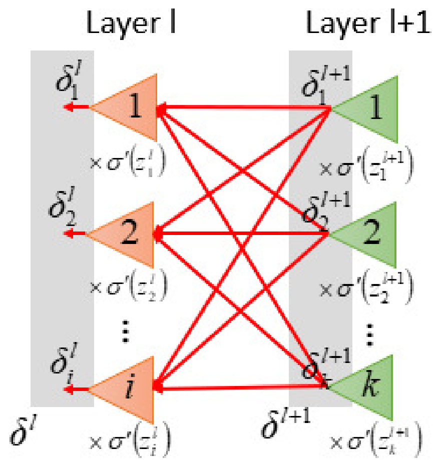


$$\delta_i^l = \frac{\partial a_i^l}{\partial z_i^l}\sum_k \frac{\partial z_k^{l+1}}{\partial a_i^l}\delta_k^{l+1}$$

$\sigma'(z_i^l)$

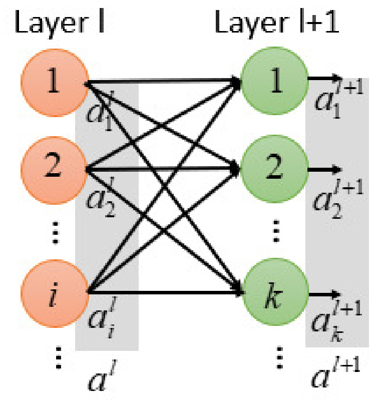
$$z_k^{l+1} = \sum_i w_{ki}^{l+1} a_i^l + b_k^{l+1}$$

$$\delta_i^l = \sigma'(z_i^l)\sum_k w_{ki}^{l+1}\delta_k^{l+1}$$



$$\delta^l = \sigma'(z^l)\bullet(W^{l+1})^T\delta^{l+1}$$

Compare



$$a^{l+1} = \sigma(W^{l+1}a^l + b^{l+1})$$

```
for i = (n - 1) : -1 : 2  %注释：i 从2到2，loop内只转一次
      % Derivative of the activation function
      switch nn.activation_function
         case 'sigm'
            d_act = nn.a{i} .* (1 - nn.a{i}); %激活函数的导数，Matlab
help:   X.*Y denotes element-by-element multiplication
%注释：  d_act = nn.a{2} .* (1 - nn.a{2});
         case 'tanh_opt'
            d_act = 1.7159 * 2/3 * (1 - 1/(1.7159)^2 * nn.a{i}.^2);
      end

      if(nn.nonSparsityPenalty>0)
         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))];
      end

      % Backpropagate first derivatives
      if i+1==n % in this case in d{n} there is not the bias term to
be removed， i+1=2+1==3==n ， 见p.13页注解，只有输出项nn.a{3}
不加bias项对应的第一列，1行向量 ， %注释： d{2} =delta（2）,
d_act=h’ (a)
       d{i} = (d{i + 1} * nn.W{i} + sparsityError) .* d_act;
 % Bishop (5.56)
      else % in this case in d{i} the bias term has to be removed
         d{i} = (d{i + 1}(:,2:end) * nn.W{i} + sparsityError) .* d_act;
      end
      if(nn.dropoutFraction>0)
         d{i} = d{i} .* [ones(size(d{i},1),1) nn.dropOutMask{i}];
      end
   end
```

$$\frac{\partial C^r}{\partial w^l_{ij}} = \frac{\partial z^l_i}{\partial w^l_{ij}} \boxed{\frac{\partial C^r}{\partial z^l_i}}$$

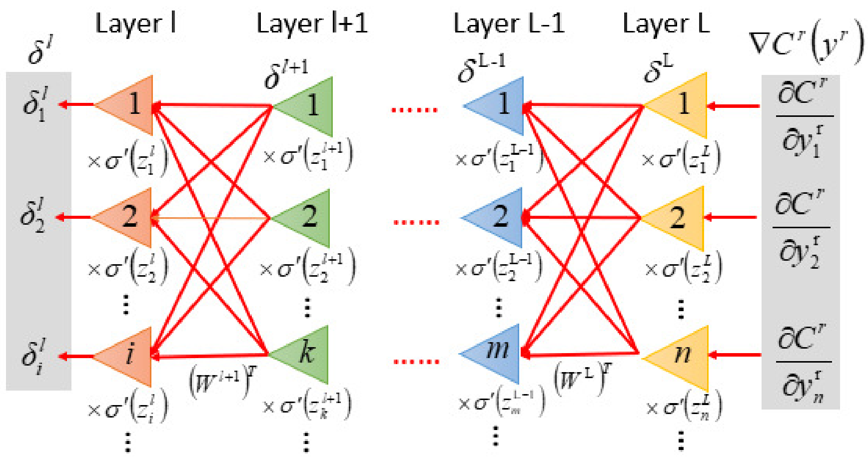
δ^l

1. How to compute δ^L

$$\Rightarrow \delta^L = \sigma'(z^L) \bullet \nabla C^r(y^r)$$

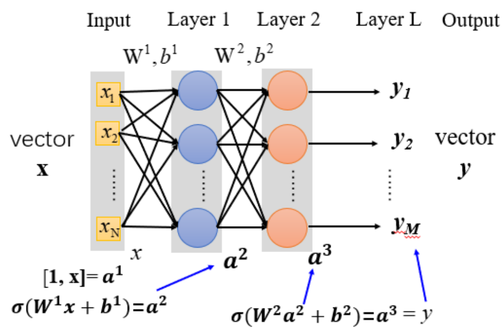
2. The relation of δ^l and δ^{l+1}

$$\Rightarrow \delta^l = \sigma'(z^l) \bullet (W^{l+1})^T \delta^{l+1}$$



backpropagation formula

$$\delta_j = h'(a_j) \sum_k w_{kj} \delta_k \qquad (5.56)$$



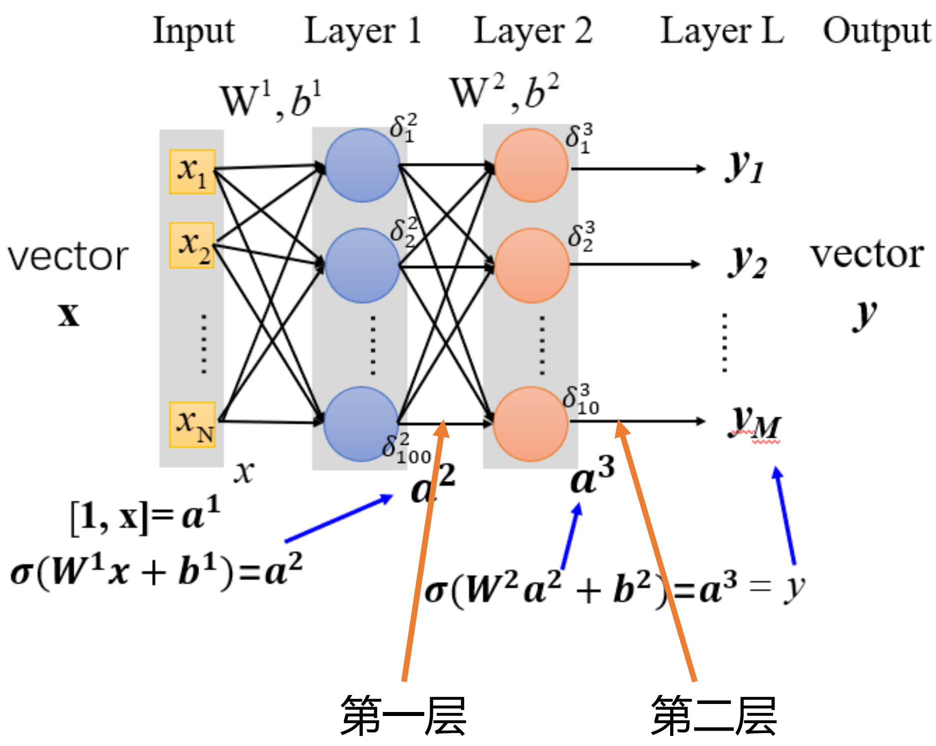
```
for i = 1 : (n - 1)
      if i+1==n
         nn.dW{i} = (d{i + 1}' * nn.a{i}) / size(d{i + 1}, 1);
      else
         nn.dW{i} = (d{i + 1}(:,2:end)' * nn.a{i}) / size(d{i + 1}, 1);
      end
   end
end
```

% 注释：nn.dW{1} = (d{2}(:,2:end)' * nn.a{1}) / size(d{2}, 1);

% 注释： nn.dW{2} = (d{3}' * nn.a{2}) / size(d{3}, 1)

% 注释： d{i}为i层的delta

nn.dW{i}为W(i)的调整量



```
function nn = nnapplygrads(nn)
%NNAPPLYGRADS updates weights and biases with
calculated gradients
% nn = nnapplygrads(nn) returns an neural network
structure with updated
% weights and biases

    for i = 1 : (nn.n - 1)
        if(nn.weightPenaltyL2>0)
            dW = nn.dW{i} + nn.weightPenaltyL2 *
[zeros(size(nn.W{i},1),1) nn.W{i}(:,2:end)];
        else
            dW = nn.dW{i};
        end

        dW = nn.learningRate * dW; %梯度变化最速方向

        if(nn.momentum>0)
            nn.vW{i} = nn.momentum*nn.vW{i} + dW;
            dW = nn.vW{i};
        end

        nn.W{i} = nn.W{i} - dW; %梯度下降法
    end
end
```