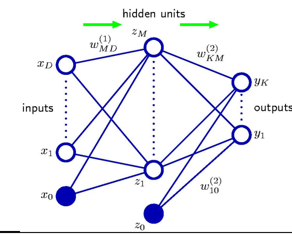
Error Backpropagation

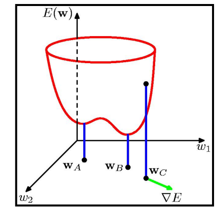
Sargur Srihari

Topics

- Neural Network Learning Problem
- Need for computing derivatives of Error function
- Forward propagation of activations
- Backward propagation of errors
- Statement of Backprop algorithm
- Use of backprop in computing the Jacobian matrix

Neural Network Learning Problem





- Goal is to learn the weights w from a labeled set of training samples
  - No. of weights is $T=(D+1)M+(M+1)K$ $=M(D+K+1)+K$
    - Where D is no of inputs, M is no of hidden units, K is no of outputs
- Learning procedure has two stages
  1. Evaluate derivatives of error function $\nabla E(\mathrm{w})$ with respect to weights $w_1,..w_T$
  2. Use derivatives to compute adjustments to weights

$$\mathrm{w}^{(\tau+1)} = \mathrm{w}^{(\tau)} - \eta \nabla E(\mathrm{w}^{(\tau)})$$

$$\nabla E(\mathrm{w}) = \begin{bmatrix} \frac{\partial E}{\partial w_0} \\ \frac{\partial E}{\partial w_1} \\ \\ \frac{\partial E}{\partial w_T} \end{bmatrix}$$

- Error Functions
  - Linear Regression

$$E(\mathrm{w}) = \frac{1}{2}\sum_{n=1}^{N}\{y(\mathrm{x}_n,\mathrm{w}) - t_n\}^2$$

$$y\,(\mathrm{x},\mathrm{w}) = \sum_{i=1}^{M} w_{ki}^{(2)} x_i + w_{k0}^{(2)} \quad \text{where} \quad k = 1,..,K$$

  - Binary Classification

$$E(\mathrm{w}) = -\sum_{n=1}^{N}\{t_n \ln y_n + (1-t_n)\ln(1-y_n)\}$$

$$y = \sigma(a) = \frac{1}{1+\exp(-a)}$$

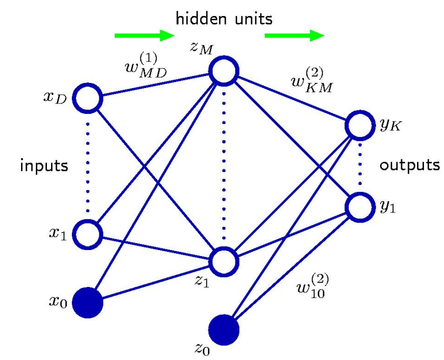
$$a = \sum_{i=1}^{M} w_{ki}^{(2)} x_i + w_{k0}^{(2)} \quad \text{where} \quad k = 1,..,K$$

  - Multiclass Classification

$$E(\mathrm{w}) = -\sum_{n=1}^{N}\sum_{k=1}^{K} t_{kn} \ln y_k(x_n,\mathrm{w})$$

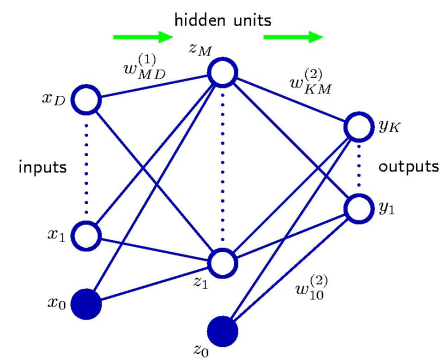
$$y_k(\mathrm{x},\mathrm{w}) = \frac{\exp(a_k)}{\sum_j \exp(a_j)}$$

Back-propagation Terminology



- Goal: Efficient technique for evaluating gradient ∇ of an error function $E(\mathrm{w})$ for a feed-forward neural network
- Backpropagation is term used for derivative computation only
- In subsequent stage derivatives are used to make adjustments to weights
- Achieved using a local message passing scheme
  - Information sent forwards and backwards alternately

Overview of Backprop algorithm



- Choose *random* weights for the network
- Feed in an example and obtain a result
- Calculate the error for each node (starting from the last stage and propagating the error backwards)
- Update the weights
- *Repeat* with other examples until the network converges on the target output

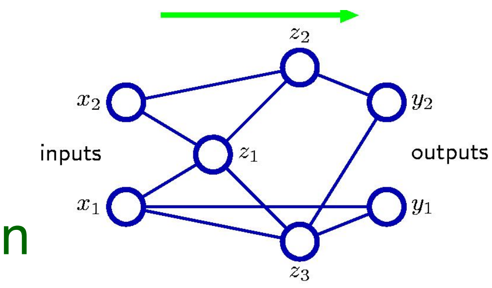
- How to divide up the errors needs a little calculus

Wide use of Backpropagation

- Can be applied to error function other than sum of squared errors
- Used to evaluate other matrices such as Jacobian and Hessian matrices
- Second stage of weight adjustment using calculated derivatives can be tackled using variety of optimization schemes substantially more powerful than gradient descent

Evaluation of Error Function Derivatives

- Derivation of back-propagation algorithm for
  - Arbitrary feed-forward topology
  - Arbitrary differentiable nonlinear activation function
  - Broad class of error functions

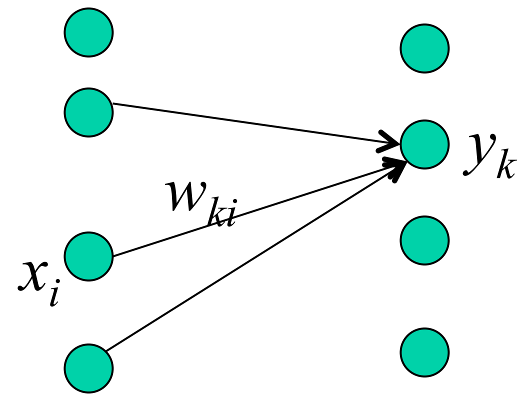


- Error functions of practical interest are sums of errors associated with each training data point

$$E(\mathrm{w}) = \sum_{n=1}^{N} E_n(\mathrm{w})$$

- We consider problem of evaluating
  - For n^{th} term in the error function $\nabla E_n(\mathrm{w})$
  - Derivatives are wrt the weights $w_1, .. w_T$
- Can be used directly for sequential optimization or accumulated over training set (for batch)

A simple Linear Model



- Outputs y_k are linear combinations of input variables x_i

$$y_k = \sum_i w_{ki} x_i$$

- Error function for a particular input x_n has the form

$$E_n = \frac{1}{2} \sum_k (y_{nk} - t_{nk})^2$$

Subscript n is for a particular input x_n which is ignored below

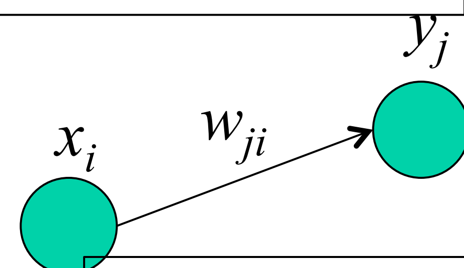
$$E = \frac{1}{2}\left(y(x,w) - t\right)^2$$

$$\frac{\partial E}{\partial w} = \left(y(x,w) - t\right)x = \delta \cdot x$$

  - where $y_{nk} = y_k(\mathrm{x}_n, \mathrm{w})$
- Gradient of Error function wrt a weight w_{ji}

$$\frac{\partial E_n}{\partial w_{ji}} = \left(y_{nj} - t_{nj}\right) x_{ni}$$

  - a local computation involving product of
    - error signal y_{nj}-t_{nj} associated with output end of link w_{ji}
    - variable x_{ni} associated with input end of link



$$\frac{\partial E}{\partial w_{ji}} = \left(y_j - t_j\right)x_i = \delta_j \cdot x_i$$

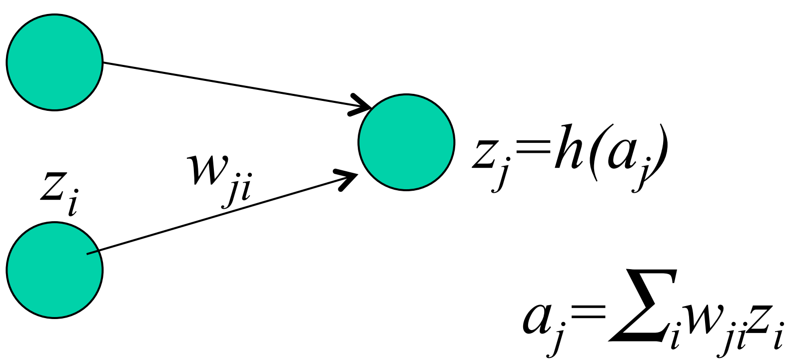
General Feed-Forward Network: Forward Propagation

- Each unit computes weighted sum of its inputs

$$a_j = \sum_i w_{ji} z_i$$

- z_i is activation of a unit (or input) that sends a connection to unit j and w_{ji} is the weight associated with the connection
- Transformed by nonlinear activation function

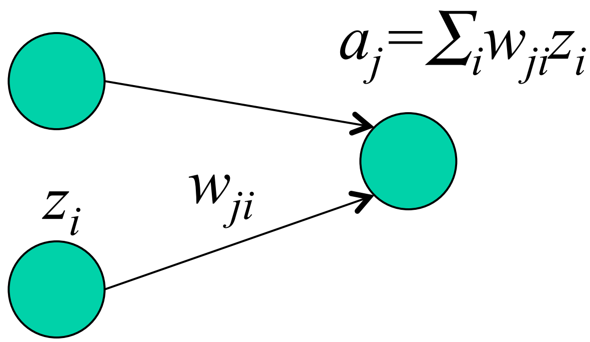
$z_j = h(a_j)$



Evaluation of Derivative E_n wrt a weight w_{ji}

- By chain rule for partial derivatives

$$\frac{\partial E_n}{\partial w_{ji}} = \frac{\partial E_n}{\partial a_j}\frac{\partial a_j}{\partial w_{ji}}$$



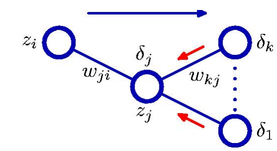
Define $\delta_j \equiv \dfrac{\partial E_n}{\partial a_j}$

$$a_j = \sum_i w_{ji} z_i$$

we have $\dfrac{\partial a_j}{\partial w_{ji}} = z_i$

- Substituting $\boxed{\dfrac{\partial E_n}{\partial w_{ji}} = \delta_j z_i}$
- Thus required derivative $\dfrac{\partial E_n}{\partial w_{ji}}$ is obtained by
  - Multiplying value of δ for the unit at output end of weight by value of z for unit at input end of weight
- Need to figure out how to calculate $\delta_j \equiv \dfrac{\partial E_n}{\partial a_j}$

Calculation of Error for hidden unit δ_j



- For output unit $\delta_k = y_k - t_k$ Since $E = \frac{1}{2}\sum_k (y_k - t_k)^2$ and $y_k = a_k = \sum w_{ki} z_i$ $\delta_k \equiv \frac{\partial E}{\partial a_k}$
- For hidden unit j
  - By chain rule

$$\delta_j \equiv \frac{\partial E_n}{\partial a_j} = \sum_k \underbrace{\frac{\partial E_n}{\partial a_k}} \underbrace{\frac{\partial a_k}{\partial a_j}}$$

We are summing partial derivatives over several variables a_k

- Substituting $\delta_k \equiv \frac{\partial E_n}{\partial a_k}$
  - We get the backpropagation formula for for error derivatives at stage j

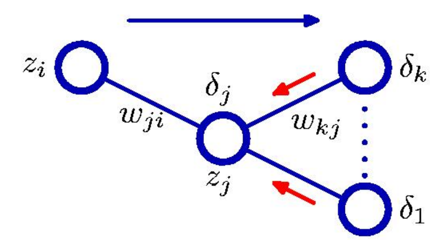
$$a_k = \sum_i w_{ki} z_i = \sum_i w_{ki} h(a_i)$$

$$\frac{\partial a_k}{\partial a_j} = \sum_k w_{kj} h'(a_j)$$

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

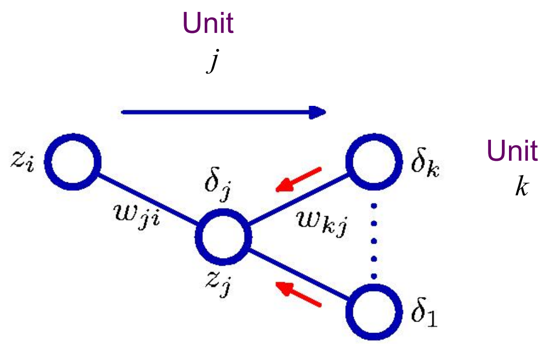
Input to activation from earlier units

error derivative at later unit k



Blue arrow for forward propagation
Red arrows indicate direction of information flow during error backpropagation

Error Backpropagation Algorithm



- Backpropagation Formula

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

- Value of δ for a particular hidden unit can be obtained by propagating the δ's backward from units higher-up in the network

1. Apply input vector x_n to network and forward propagate through network using

$$a_j = \sum_i w_{ji} z_i \quad \text{and} \quad z_j = h(a_j)$$

2. Evaluate δ_k for all output units using $\delta_k = y_k - t_k$

3. Backpropagate the δ's using

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

   to obtain δ_j for each hidden unit

4. Use $\dfrac{\partial E_n}{\partial w_{ji}} = \delta_j z_i$

   to evaluate required derivatives

A Simple Example

- Two-layer network
- Sum-of-squared error
- Output units: *linear activation* functions, i.e., multiple regression

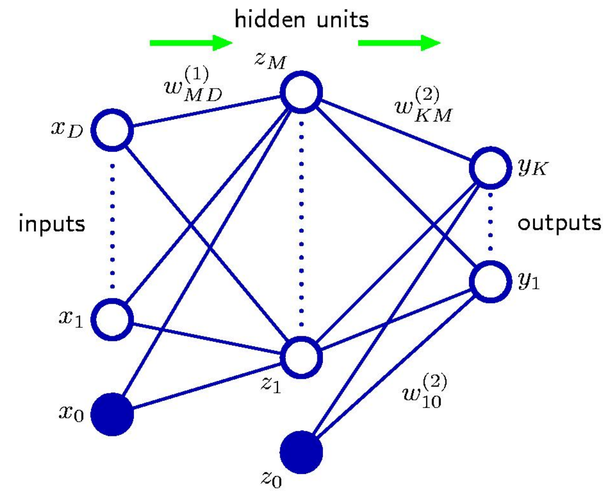
  $y_k = a_k$

- Hidden units have *logistic sigmoid* activation function

  $h(a) = \tanh(a)$

  where
  $$\tanh(a) = \frac{e^a - e^{-a}}{e^a + e^{-a}}$$

  simple form for derivative
  $$\boxed{h'(a) = 1 - h(a)^2}$$



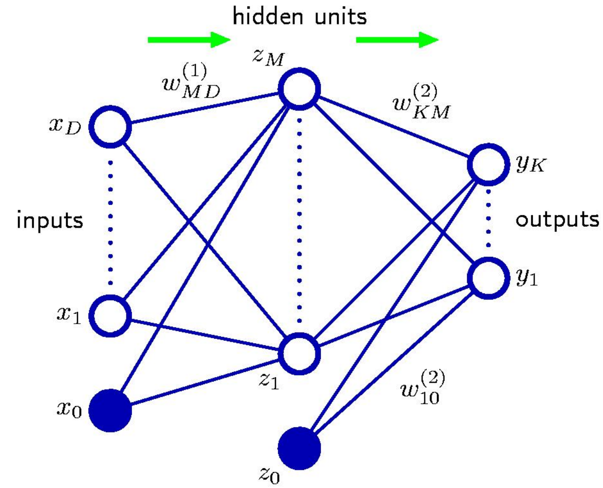
Standard Sum of Squared Error

$$E_n = \frac{1}{2}\sum_{k=1}^{K}(y_k - t_k)^2$$

y_k: activation of output unit k
t_k : corresponding target for input x_k

Simple Example: Forward and Backward Prop

For each input in training set:



- Forward Propagation

$$\begin{cases} a_j = \sum_{i=0}^{D} w_{ji}^{(1)} x_i \\ z_j = \tanh(a_j) \\ y_k = \sum_{j=0}^{M} w_{kj}^{(2)} z_j \end{cases}$$

- Output differences

$$\delta_k = y_k - t_k$$

- Backward Propagation (δs for hidden units)

$$\delta_j = (1 - z_j^2) \sum_{k=1}^{K} w_{kj} \delta_k$$

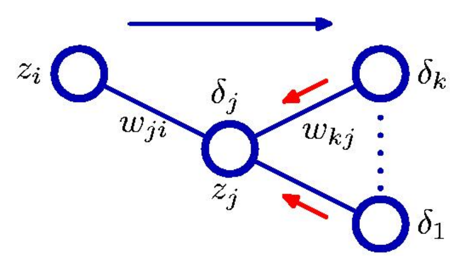
$$\boxed{\begin{gathered} \delta_j = h'(a_j) \sum_k w_{kj} \delta_k \\ h'(a) = 1 - h(a)^2 \end{gathered}}$$

- Derivatives wrt first layer and second layer weights

$$\frac{\partial E_n}{\partial w_{ji}^{(1)}} = \delta_j x_i \qquad \frac{\partial E_n}{\partial w_{kj}^{(2)}} = \delta_k z_j$$

- Batch method

$$\frac{\partial E}{\partial w_{ji}} = \sum_n \frac{\partial E_n}{\partial w_{ji}}$$



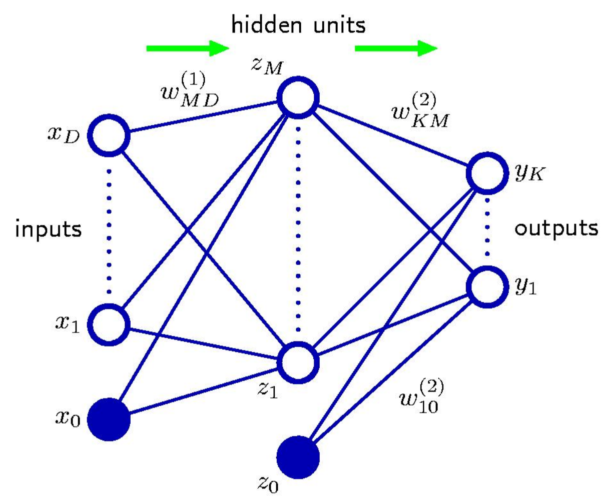
Using derivatives to update weights

- Gradient descent
  - Update the weights using $\mathrm{w}^{(\tau+1)} = \mathrm{w}^{(\tau)} - \eta \nabla E\left(\mathrm{w}^{(\tau)}\right)$
  - Where the gradient vector $\nabla E\left(\mathrm{w}^{(\tau)}\right)$ consists of the vector of derivatives evaluated using back-propagation

$$\nabla E(\mathrm{w}) = \frac{d}{d\mathrm{w}} E(\mathrm{w}) = \begin{bmatrix} \dfrac{\partial E}{\partial w_{11}^{(1)}} \\ . \\ \dfrac{\partial E}{\partial w_{MD}^{(1)}} \\ \dfrac{\partial E}{\partial w_{11}^{(2)}} \\ . \\ \dfrac{\partial E}{\partial w_{KM}^{(2)}} \end{bmatrix}$$

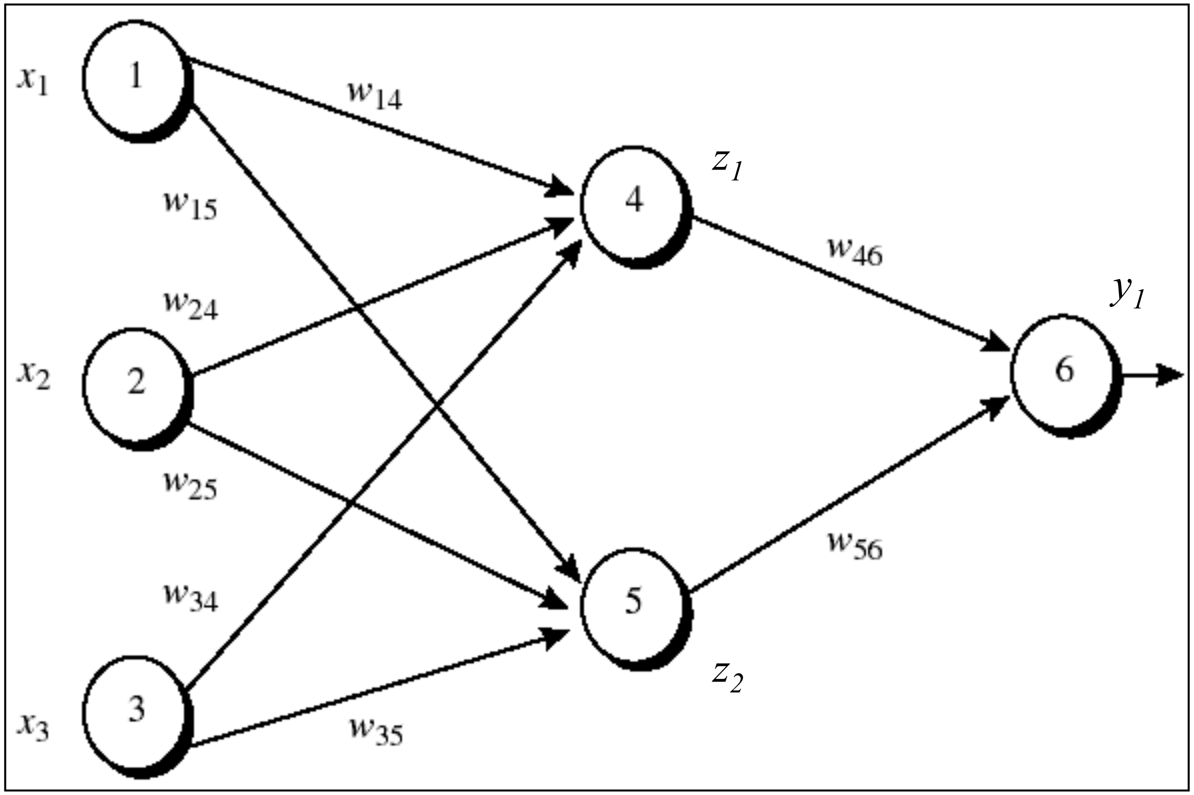
There are $W = M(D+1)+K(M+1)$ elements in the vector

Gradient $\nabla E\left(\mathrm{w}^{(\tau)}\right)$ is a W x 1 vector



Numerical example
(binary classification)

D=3
M=2
K=1
N=1



$$a_j = \sum_{i=1}^{D} w_{ji}^{(1)} x_i$$

$$z_j = \sigma(a_j)$$

$$y_k = \sum_{j=1}^{M} w_{kj}^{(2)} z_j$$

Errors

$$\delta_j = \sigma'(a_j) \sum_k w_{kj} \delta_k$$

$$\delta_k = \sigma'(a_k)\left(y_k - t_k\right)$$

Error Derivatives

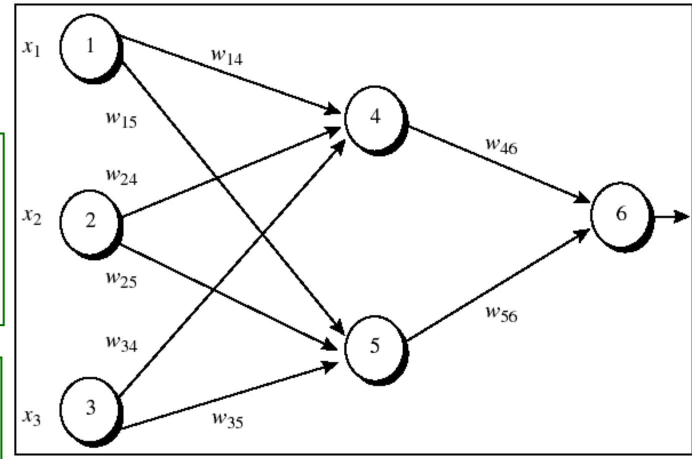
$$\frac{\partial E_n}{\partial w_{ji}^{(1)}} = \delta_j x_i \qquad \frac{\partial E_n}{\partial w_{kj}^{(2)}} = \delta_k z_j$$

- First training example, $\mathrm{x} = [1 \quad 0 \quad 1]^{\mathrm{T}}$ whose class label is $t = 1$
- The sigmoid activation function is applied to hidden layer and output layer
- Assume that the learning rate η is 0.9

Outputs, Errors, Derivatives, Weight Update

$$\delta_k = \sigma'(a_k)(y_k - t_k) = [\sigma(a_k)(1-\sigma(a_k))](1-\sigma(a_k))$$

$$\delta_j = \sigma'(a_j)\sum_k w_{jk}\delta_k = \left[\sigma(a_j)(1-\sigma(a_j))\right]\sum_k w_{jk}\delta_k$$



Initial input and weight values

x_1	x_2	x_3	w_{14}	w_{15}	w_{24}	w_{25}	w_{34}	w_{35}	w_{46}	w_{56}	w_{04}	w_{05}	w_{06}
1	0	1	0.2	-0.3	0.4	0.1	-0.5	0.2	-0.3	-0.2	-0.4	0.2	0.1

Net input and output calculation

Unit	Net input a	Output $\sigma(a)$
4	0.2 + 0 -0.5 -0.4 = -0.7	$1/(1+e^{0.7})=0.332$
5	-0.3 +0+0.2 +0.2 =0.1	$1/(1+e^{0.1})=0.525$
6	(-0.3)(0.332)-(0.2)(0.525)+0.1 = -0.105	$1/(1+e^{0.105})=0.474$

Errors at each node

Unit	δ
6	(0.474)(1-0.474)(1-0.474)=0.1311
5	(0.525)(1-0.525)(0.1311)(-0.2)=-0.0065
4	(0.332)(1-0.332)(0.1311)(-0.3)=-0.0087

Weight Update*

Weight	New value
w_{46}	-03+(0.9)(0.1311)(0.332)= -0.261
w_{56}	-0.2+(0.9)(0.1311)(0.525)= -0.138
w_{14}	0.2 +(0.9)(-0.0087)(1) = 0.192
w_{15}	-0.3 +(0.9)(-0.0065)(1) = -0.306
w_{24}	0.4+ (0.9)(-0.0087)(0) = 0.4
w_{25}	0.1+ (0.9)(-0.0065)(0) = 0.1
w_{34}	-0.5+ (0.9)(-0.0087)(1) = -0.508
w_{35}	0.2 + (0.9)(-0.0065)(1) = 0.194
w_{06}	0.1 + (0.9)(0.1311) = 0.218
w_{05}	0.2 + (0.9)(-0.0065)=0.194
w_{04}	-0.4 +(0.9)(-0.0087) = -0.408

* Positive update since we used $(t_k - y_k)$

MATLAB Implementation (Pseudocode)

- Allows for multiple hidden layers
- Allows for training in batches
- Determines gradients using back-propagation using sum-of-squared error
- Determines misclassification probability

Initializations

```
% This pseudo-code illustrates implementing a
several layer neural %network. You need to fill in
the missing part to adapt the program to %your
own use. You may have to correct minor mistakes
in the program

%% prepare for the data

load data.mat

train_x = ..
test_x = ..


train_y = ..
test_y = ..

%% Some other preparations
%Number of hidden layers

numOfHiddenLayer = 4;
```

```
s{1} = size(train_x, 1);
s{2} = 100;
s{3} = 100;
s{4} = 100;
s{5} = 2;

%Initialize the parameters
%You may set them to zero or give them small
%random values. Since the neural network
%optimization is non-convex, your algorithm
%may get stuck in a local minimum which may
%be caused by the initial values you assigned.

for i = 1 : numOfHiddenLayers
   W{i} = ..
   b{i} = ..
end
```

x is the input to the neural network,
y is the output

Training epochs, Back-propagation

The training data is divided into several batches of size 100 for efficiency

```
losses = [];
train_errors = [];
test_wrongs = [];

%Here we perform mini-batch stochastic gradient descent
%If batchsize = 1, it would be stochastic gradient descent
%If batchsize = N, it would be basic gradient descent

batchsize = 100;

%Num of batches

numbatches = size(train_x, 2) / batchsize;

%% Training part
%Learning rate alpha
alpha = 0.01;

%Lambda is for regularization
lambda = 0.001;

%Num of iterations
numepochs = 20;
```

```
for j = 1 : numepochs
   %randomly rearrange the training data for each epoch
   %We keep the shuffled index in kk, so that the input and output could
    %be matched together
    kk = randperm(size(train_x, 2));

   for l = 1 : numbatches

      %Set the activation of the first layer to be the training data
      %while the target is training labels

      a{1} = train_x(:,    kk( (l-1)*batchsize+1 : l*batchsize )    );
      y = train_y(:,    kk( (l-1)*batchsize+1 : l*batchsize )    );

      %Forward propagation, layer by layer
     %Here we use sigmoid function as an example

      for i = 2 : numOfHiddenLayer + 1
         a{i} = sigm( bsxfun(@plus, W{i-1}*a{i-1}, b{i-1}) );
      end

      %Calculate the error and back-propagate error layer by layers
     d{numOfHiddenLayer + 1} =
     -(y - a{numOfHiddenLayer + 1}) .* a{numOfHiddenLayer + 1} .* (1-a{numOfHiddenLayer + 1})

      for i = numOfHiddenLayer : -1 : 2
         d{i} = W{i}' * d{i+1} .* a{i} .* (1-a{i});
      end

      %Calculate the gradients we need to update the parameters
      %L2 regularization is used for W

      for i = 1 : numOfHiddenLayer
         dW{i} = d{i+1} * a{i}';
         db{i} = sum(d{i+1}, 2);
         W{i} = W{i} - alpha * (dW{i} + lambda * W{i});
         b{i} = b{i} - alpha * db{i};
      end
   end
```

Performance Evaluation

```
% Do some predictions to know the performance
  a{1} = test_x;
 % forward propagation

  for i = 2 : numOfHiddenLayer + 1
      %This is essentially doing W{i-1}*a{i-1}+b{i-1}, but since they
      %have different dimensionalities, this addition is not allowed in
      %matlab. Another way to do it is to use repmat

     a{i} = sigm( bsxfun(@plus, W{i-1}*a{i-1}, b{i-1}) );
  end

%Here we calculate the sum-of-square error as loss function
  loss = sum(sum((test_y-a{numOfHiddenLayer + 1}).^2)) / size(test_x, 2);

  % Count no. of misclassifications so that we can compare it
  % with other classification methods
  % If we let max return two values, the first one represents the max
  % value and second one represents the corresponding index. Since we
  % care only about the class the model chooses, we drop the max value
  % (using ~ to take the place)  and keep the index.

  [~, ind_] = max(a{numOfHiddenLayer + 1}); [~, ind] = max(test_y);
  test_wrong = sum(  ind_ ~= ind  ) / size(test_x, 2) * 100;
```

```
    %Calculate training error
    %minibatch size
    bs = 2000;
    % no. of mini-batches
    nb = size(train_x, 2) / bs;

    train_error = 0;
    %Here we go through all the mini-batches
    for ll = 1 : nb
       %Use submatrix to pick out mini-batches
       a{1} = train_x(:,    (ll-1)*bs+1 : ll*bs    );
       yy = train_y(:,    (ll-1)*bs+1 : ll*bs    );

       for i = 2 : numOfHiddenLayer + 1
          a{i} = sigm( bsxfun(@plus, W{i-1}*a{i-1}, b{i-1}) );
       end
       train_error = train_error + sum(sum((yy-a{numOfHiddenLayer + 1}).^2));
    end
    train_error = train_error /  size(train_x, 2);

    losses = [losses loss];

    test_wrongs = [test_wrongs, test_wrong];
    train_errors = [train_errors train_error];

end
```

max calculation returns value and index

Efficiency of Backpropagation

- Computational Efficiency is main aspect of back-prop
- Number of operations to compute derivatives of error function scales with total number W of weights and biases
- Single evaluation of error function for a single input requires $O(W)$ operations (for large W)
- This is in contrast to $O(W^2)$ for numerical differentiation
  - As seen next

Another Approach: Numerical Differentiation

- Compute derivatives using method of finite differences
  - Perturb each weight in turn and approximate derivatives by

$$\frac{\partial E_n}{\partial w_{ji}} = \frac{E_n(w_{ji}+\varepsilon) - E_n(w_{ji})}{\varepsilon} + O(\varepsilon) \text{ where } \varepsilon << 1$$

  - Accuracy improved by making ε smaller until round-off problems arise
  - Accuracy can be improved by using central differences

$$\frac{\partial E_n}{\partial w_{ji}} = \frac{E_n(w_{ji}+\varepsilon) - E_n(w_{ji}-\varepsilon)}{2\varepsilon} + O(\varepsilon^2)$$

  - This is $O(W^2)$
- Useful to check if software for backprop has been correctly implemented (for some test cases)

Summary of Backpropagation

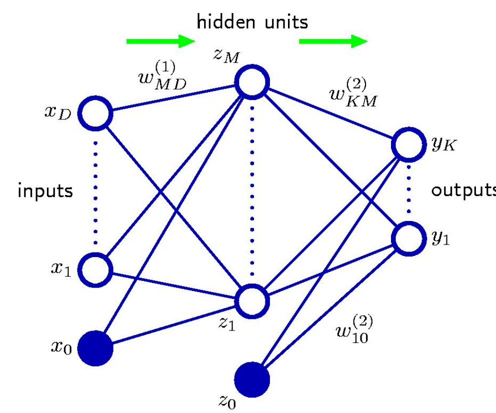
- Derivatives of error function wrt weights are obtained by propagating errors backward
- It is more efficient than numerical differentiation
- It can also be used for other computations
  - As seen next for Jacobian

The Jacobian Matrix

- For a vector valued output $\mathrm{y}=\{y_1,..,y_m\}$ with vector input $\mathrm{x}=\{x_1,..x_n\}$,
- Jacobian matrix organizes all the partial derivatives into an m x n matrix

$$J = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_n} \end{bmatrix}$$

$$J_{ki} = \frac{\partial y_k}{\partial x_i}$$



For a neural network we have a $D+1$ by K matrix

Determinant of Jacobian Matrix is referred to simply as the Jacobian

Jacobian Matrix Evaluation

- In backprop, derivatives of error function wrt weights obtained by propagating errors backward
  - We calculate $\frac{\partial E_n}{\partial w_{ji}} = \left(y_{nj} - t_{nj}\right)x_{ni}$
- Method can also be used to calculate other derivatives
- Evaluation of Jacobian matrix
  - Elements of matrix are derivatives of network outputs wrt inputs

$$J_{ki} = \frac{\partial y_k}{\partial x_i}$$

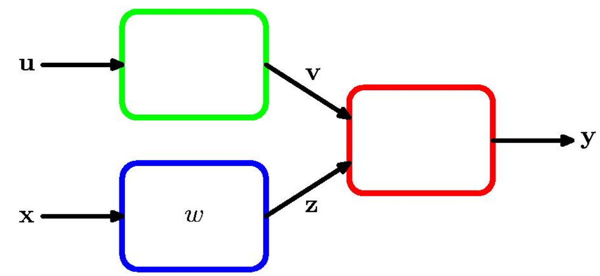
Note that for backprop we are taking the output to be y_k-t_k which is the error

  - Each derivative evaluated with other inputs fixed

Use of Jacobian Matrix

- Jacobian plays useful role in systems built from several modules
- We wish to minimize error E wrt parameter w

$$\frac{\partial E}{\partial w} = \sum_{k,j} \frac{\partial E}{\partial y_k} \underbrace{\frac{\partial y_k}{\partial z_j}} \frac{\partial z_j}{\partial w}$$



- Jacobian matrix for red module appears in the middle term
- Jacobian matrix provides measure of sensitivity of outputs to changes in input

Jacobian Matrix Computation

- Apply input vector corresponding to point in input space where the Jacobian matrix is to be found
- Forward propagate to obtain activations of the hidden and output units in the network
- For each row k of the Jacobian matrix, corresponding to output unit k:
  - Backpropagate for all the hidden units in the network
  - Finally backpropagate to the inputs
- Implementation of such an algorithm can be checked using numerical differentiation

Summary

- Neural network learning needs learning of weights from samples
- Involves two steps: determining derivative of output of a unit wrt each input
- Backpropagation is a general term for computing derivatives
  - Evaluate δ_k for all output units using $\delta_k = y_k - t_k$
  - Backpropagate the δ_k's to obtain δ_j for each hidden unit
  - Product of δ's with activations at the unit provide the derivatives for that weight
- Useful to compute a Jacobian matrix with several inputs and outputs