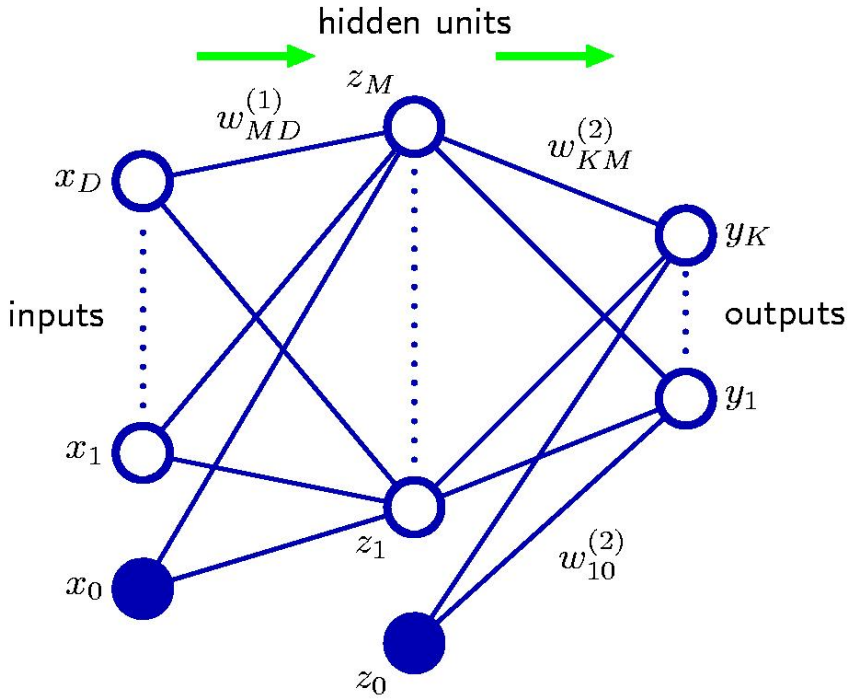
Backpropagation

Sargur Srihari

Topics in Backpropagation

1. Forward Propagation
2. Loss Function and Gradient Descent
3. Computing derivatives using chain rule
4. Computational graph for backpropagation
5. Backprop algorithm
6. The Jacobian matrix

A neural network with one hidden layer



Augmented network

No. of weights in $\boldsymbol{w}$:

$T=(D+1)M+(M+1)K$

$=M(D+K+1)+K$

D input variables $x_1,..,\ x_D$

M hidden unit activations

$$a_j = \sum_{i=1}^{D} w^{(1)}_{ji} x_i + w^{(1)}_{j0} \quad \text{where} \quad j = 1,..,M$$

Hidden unit activation functions

$$z_j = h(a_j)$$

K output activations

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

Output activation functions

$$y_k = \sigma(a_k)$$

$$y_k(\boldsymbol{x},\boldsymbol{w}) = \sigma\left(\sum_{j=1}^{M} w^{(2)}_{kj} h\left(\sum_{i=1}^{D} w^{(1)}_{ji} x_i + w^{(1)}_{j0} \right) + w^{(2)}_{k0} \right)$$

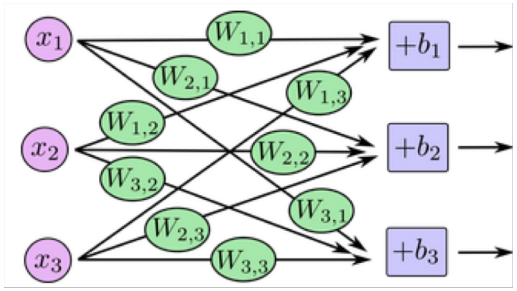
Matrix Multiplication: Forward Propagation

- Each layer is a function of layer that preceded it
  - First layer is given by $\boldsymbol{z} = h\left(\boldsymbol{W}^{(1)\mathrm{T}}\boldsymbol{x} + \boldsymbol{b}^{(1)}\right)$
  - Second layer is $\boldsymbol{y} = \sigma\left(\boldsymbol{W}^{(2)\mathrm{T}}\boldsymbol{x} + \boldsymbol{b}^{(2)}\right)$
  - Note that W is a matrix rather than a vector
    - Example with $D=3,\ M=3$

$$\boldsymbol{x} = [x_1, x_2, x_3]^{\mathrm{T}} \qquad \boldsymbol{w} = \begin{cases} W_1^{(1)} = \left[W_{11} W_{12} W_{13}\right]^T, W_2^{(1)} = \left[W_{21} W_{22} W_{23}\right]^T, W_3^{(1)} = \left[W_{31} W_{32} W_{33}\right]^T \\ W_1^{(2)} = \left[W_{11} W_{12} W_{13}\right]^T, W_2^{(2)} = \left[W_{21} W_{22} W_{23}\right]^T, W_3^{(2)} = \left[W_{31} W_{32} W_{33}\right]^T \end{cases}$$

  - Unaugmented network

First Network layer



Network layer output

$$\begin{pmatrix} W_{1,1}x_1 + W_{1,2}x_2 + W_{1,3}x_3 + b_1 \\ W_{2,1}x_1 + W_{2,2}x_2 + W_{2,3}x_3 + b_2 \\ W_{3,1}x_1 + W_{3,2}x_2 + W_{3,3}x_3 + b_3 \end{pmatrix}$$

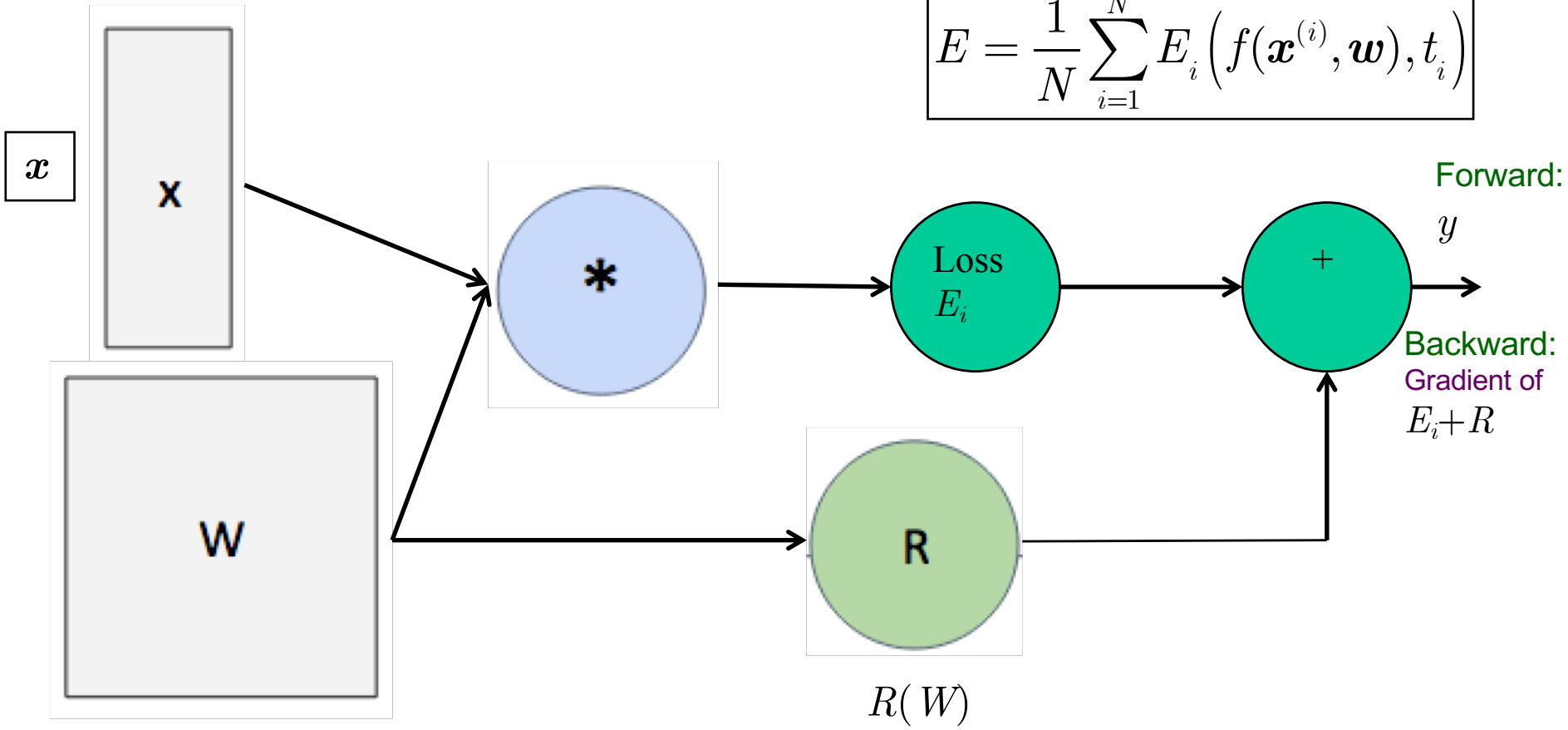
In matrix multiplication notation

$$\left(\begin{bmatrix} W_{1,1} & W_{1,2} & W_{1,3} \\ W_{2,1} & W_{2,2} & W_{2,3} \\ W_{3,1} & W_{3,2} & W_{3,3} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \right)$$

Loss and Regularization

$$y = f(\boldsymbol{x}, \boldsymbol{w})$$

$$E = \frac{1}{N}\sum_{i=1}^{N} E_i\left(f(\boldsymbol{x}^{(i)}, \boldsymbol{w}), t_i\right)$$



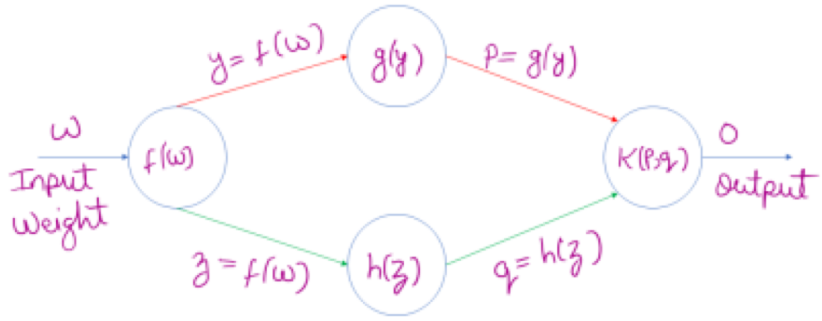
Gradient Descent

- Goal: determine weights $\boldsymbol{w}$ from labeled set of training samples
- Learning procedure has two stages
  1. Evaluate derivatives of loss $\nabla E(\boldsymbol{w})$ with respect to weights $w_1,..w_T$
  2. Use derivative vector to compute adjustments to weights

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

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

Computing derivatives using Chain Rule



$$o = K(p,q) = K(g(f(\omega)), h(f(\omega)))$$

Composition function

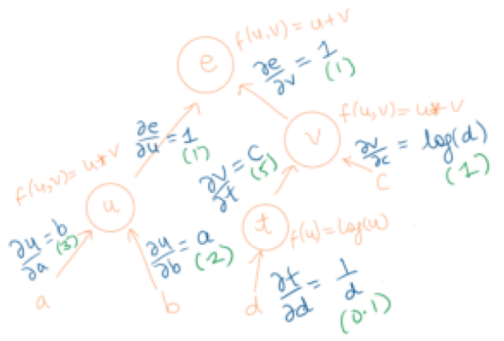
$$\frac{\partial o}{\partial \omega} = \frac{\partial o}{\partial p} \cdot \frac{\partial p}{\partial y} \cdot \frac{\partial y}{\partial \omega} + \frac{\partial o}{\partial q} \cdot \frac{\partial q}{\partial z} \cdot \frac{\partial z}{\partial \omega} \quad \text{[Chain Rule]}$$

$$= \frac{\partial K(p,q)}{\partial p} \cdot g'(y) \cdot f'(\omega) + \frac{\partial K(p,q)}{\partial q} \cdot h'(z) \cdot f'(\omega)$$

Path 1 Path 2

Back-propagation:

e.g. $a = 2, b = 3, c = 5, d = 10$



$e = a*b + c\log(d)$

$\frac{\partial e}{\partial a} = b(1) = b$

$\frac{\partial e}{\partial b} = a(1) = a$

$\frac{\partial e}{\partial c} = \log d \times 1 = \log d$

$\frac{\partial e}{\partial d} = \frac{1}{d} \times c * 1 = \frac{c}{d}$

Example of Derivative Computation

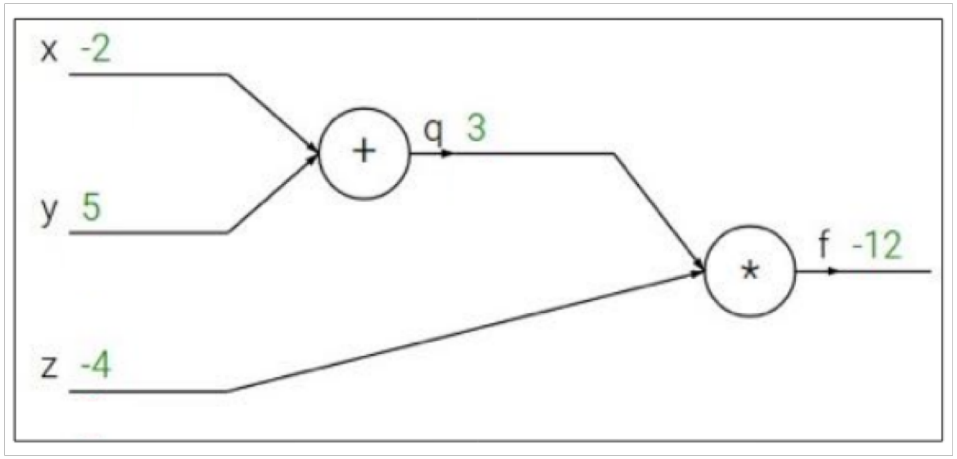
$$f(x,y,z) = (x+y)z$$

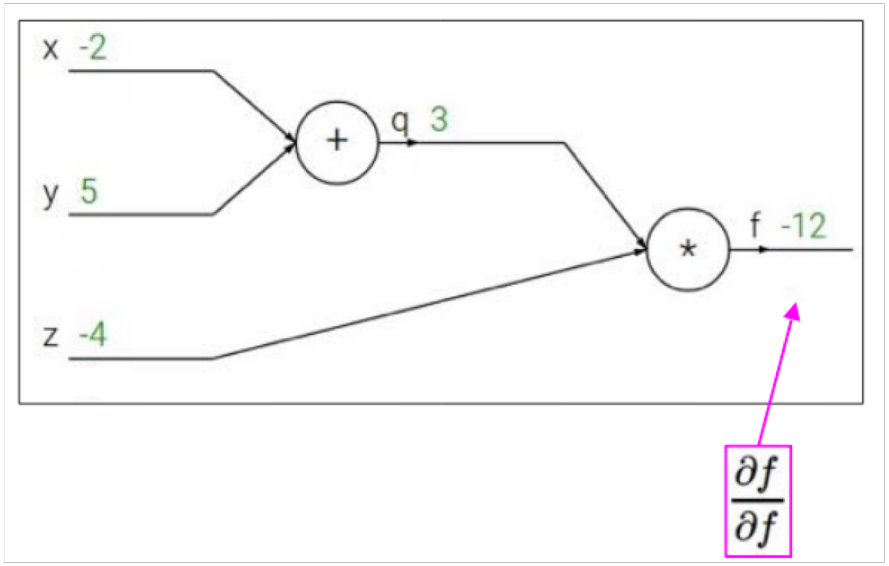
e.g. x = -2, y = 5, z = -4

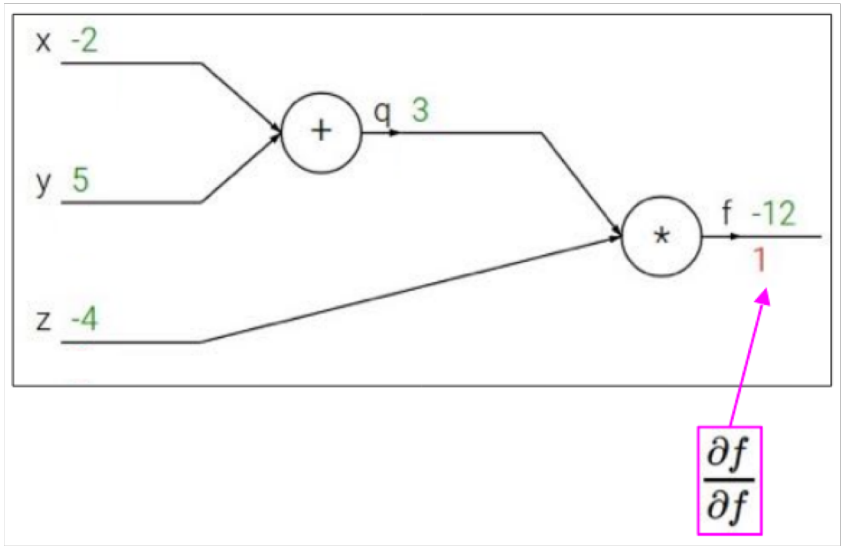
$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

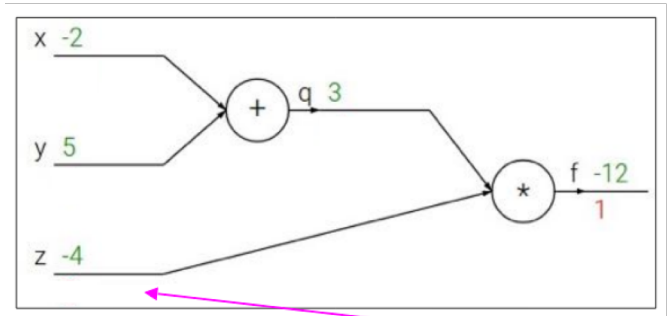
Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



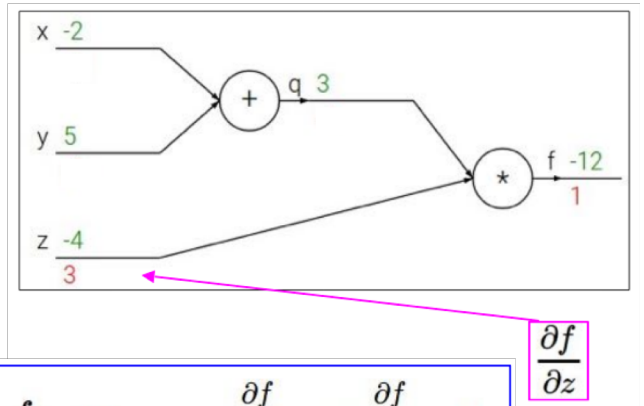




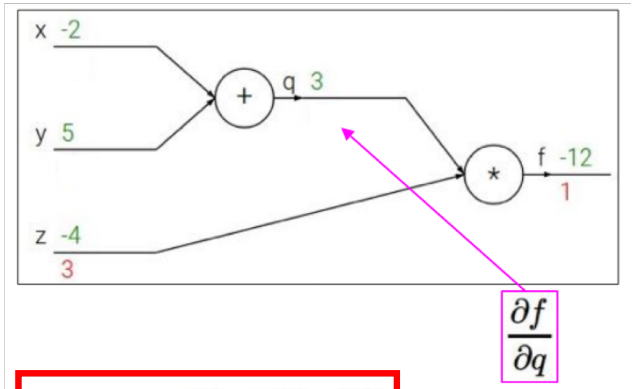
Computing derivatives of output wrt inputs



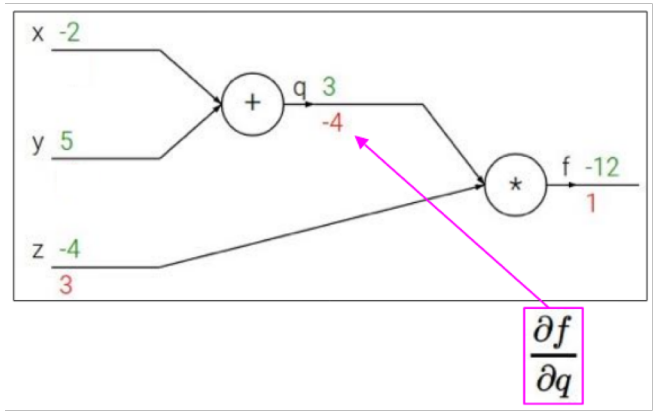
$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$

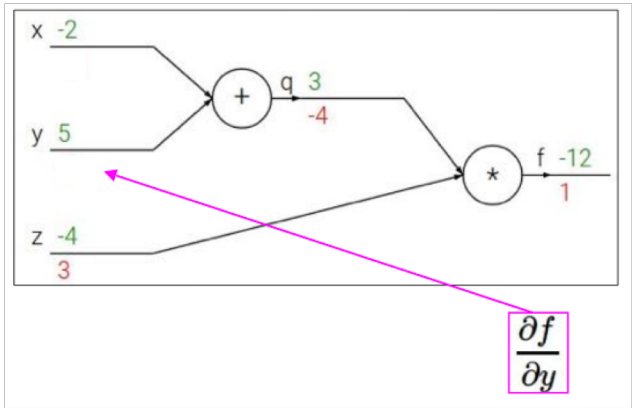


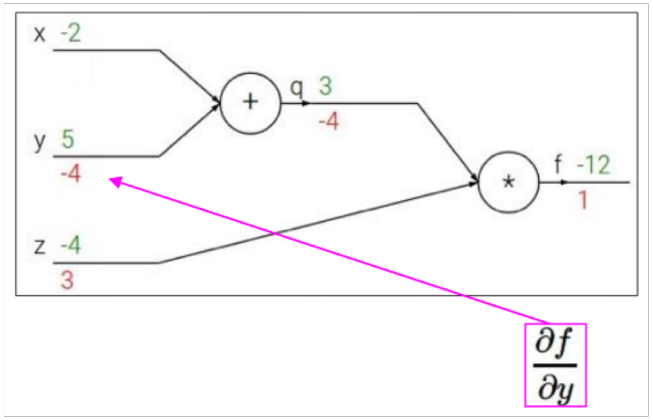
$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$

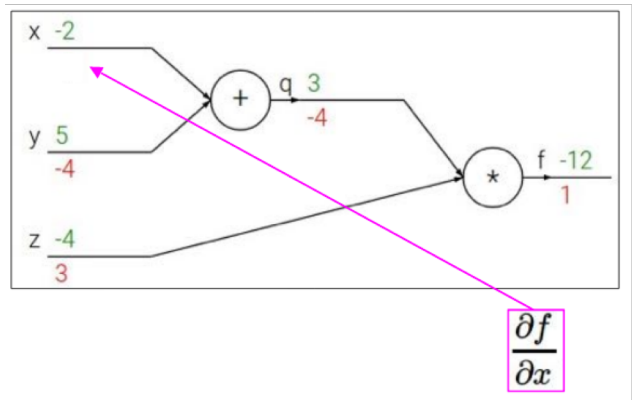


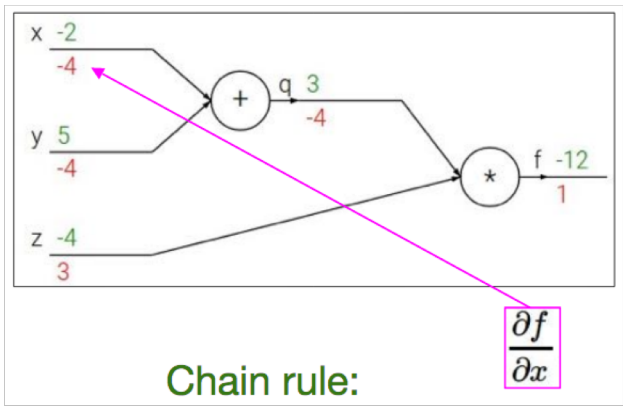
Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$





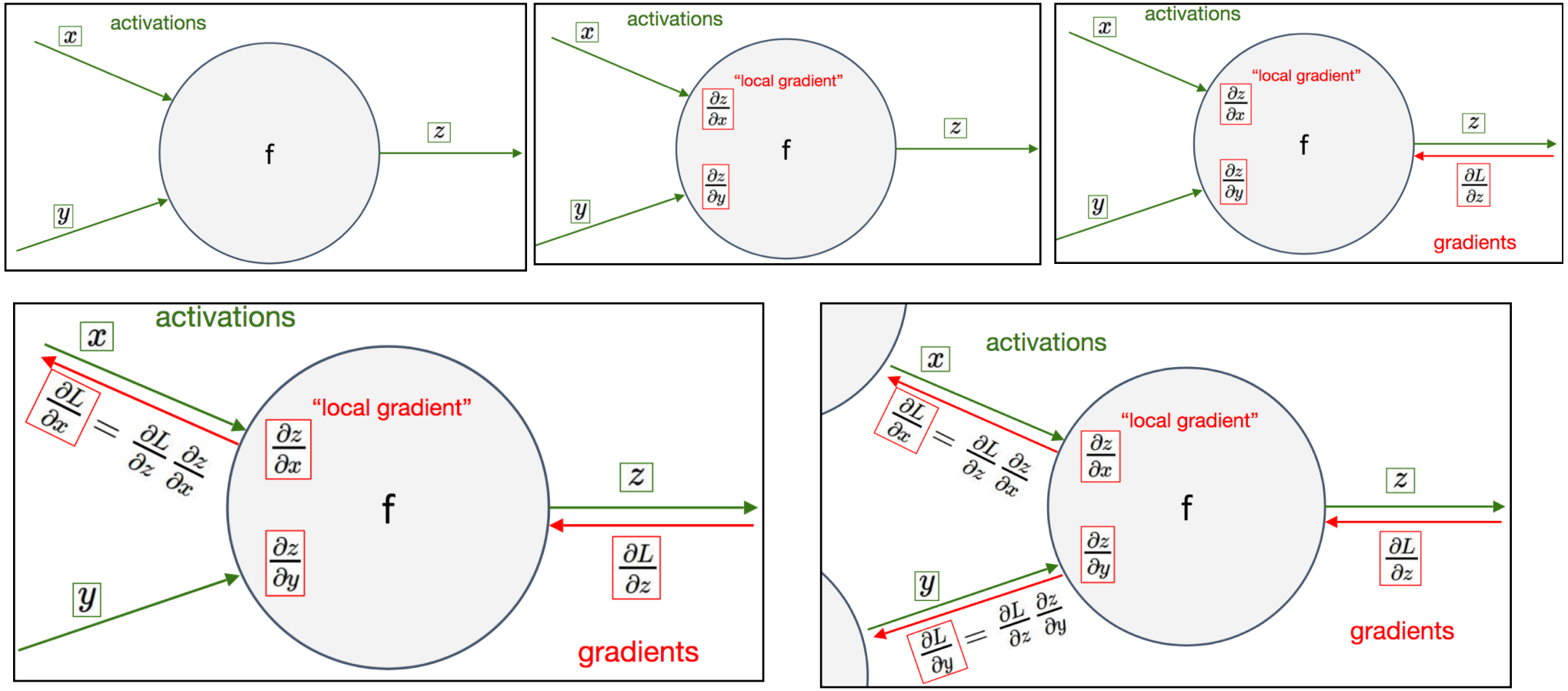






$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial x}$$

Derivatives for a neuron: $z=f(x,y)$



Composite Function

- Consider a composite function $f\left(g\left(h\left(x\right)\right)\right)$
  - i.e., an outer function f, an inner function g and a final inner function $h(x)$
- Say $f(x) = e^{\sin(x^2)}$ we can decompose it as:

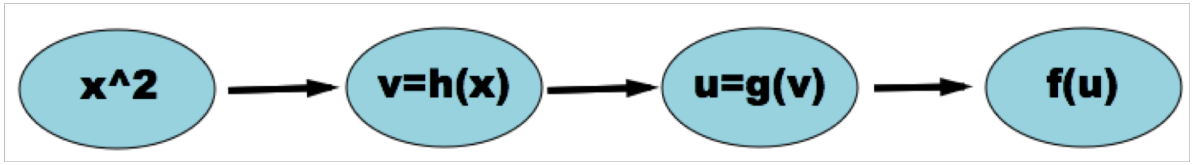
  $f\left(x\right)=e^{x}$

  $g(x)=sin\ x$ and

  $h(x)=x^2$ or

  $f\left(g(h(x))\right)=e^{\ g(h(x))}$

  - Its computational graph is



  - Every connection is an input, every node is a function or operation

Derivatives of Composite function

- To get derivatives of $f\left(g\left(h\left(x\right)\right)\right)= e^{g(h(x))}$ wrt x

1. We use the chain rule $\boxed{\frac{df}{dx}=\frac{df}{dg}\cdot\frac{dg}{dh}\cdot\frac{dh}{dx}}$ where

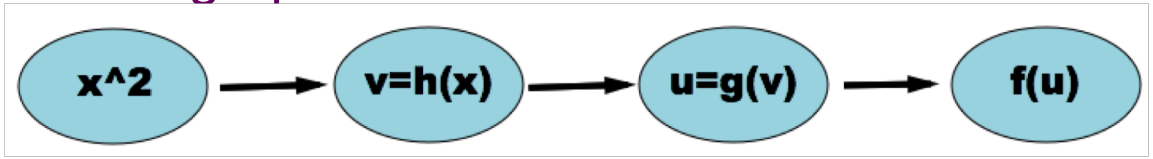
$\boxed{\frac{df}{dg}=e^{g(h(x))}}$ since $f\left(g(h(x))\right)=e^{g(h(x))}$ & derivative of e^x is e

$\boxed{\frac{dg}{dh}=\cos(h(x))}$ since $g(h(x))=sin\ h(x)$ & derivative $\sin$ is $\cos$

$\boxed{\frac{dh}{dx}=2x}$ because $h(x)=x^2$ & its derivative is $2x$

- Therefore $\boxed{\frac{df}{dx}=e^{g(h(x))}\cdot\cos\ h(x)\cdot 2x=e^{\sin x^{**}2}\cdot\cos x^2\cdot 2x}$
- In each of these cases we pretend that the inner function is a single variable and derive it as such

2. Another way to view it $f(x)=e^{\sin(x^2)}$
   - Create temp variables $u=\sin\ v$, $v=x^2$, then $f\left(u\right)=e^u$ with computational graph:



Derivative using Computational Graph

- All we need to do is get the derivative of each node wrt each of its inputs

With $u=\sin\ v$, $v=x^2$, $f\ (u)=e^u$

- We can get whichever derivative we want by multiplying the 'connection' derivatives

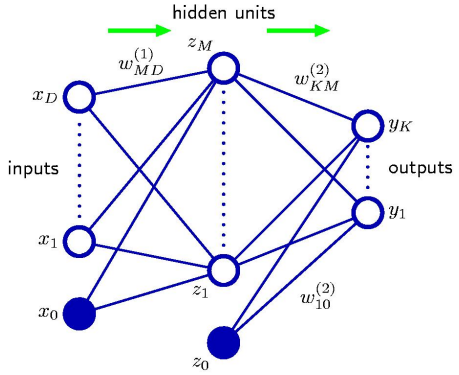
$$\boxed{\frac{dh}{dx}=2x} \qquad \boxed{\frac{dg}{dh}=\cos(h(x))} \qquad \boxed{\frac{df}{dg}=e^{g(h(x))}}$$

$$\boxed{\frac{df}{dx}=\frac{df}{dg}\cdot\frac{dg}{dh}\cdot\frac{dh}{dx}} \qquad \boxed{\begin{aligned}\frac{df}{dx}&=e^{g(h(x))}\cdot\cos\ \ h(x)\cdot 2x\\&=e^{\sin x^2}\cdot\cos x^2\cdot 2x\end{aligned}}$$

Since $f\ (x)=e^x,\ g(x)=sin\ x$ and $h(x)=x^2$

Evaluating the gradient

- Goal of this section:
  - Find an efficient technique for evaluating gradient of an error function $E(\boldsymbol{w})$ for a feed-forward neural network:



- Gradient evaluation can be performed using a local message passing scheme
  - In which information is alternately sent forwards and backwards through the network
  - Known as *error backpropagation* or simply as *backprop*

Back-propagation Terminology and Usage

- Backpropagation is term used in neural computing literature to mean a variety of different things
  - Term is used here for computing derivative of the error function wrt the weights
  - In a second separate stage the derivatives are used to compute the adjustments to be made to the weights
- 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

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

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

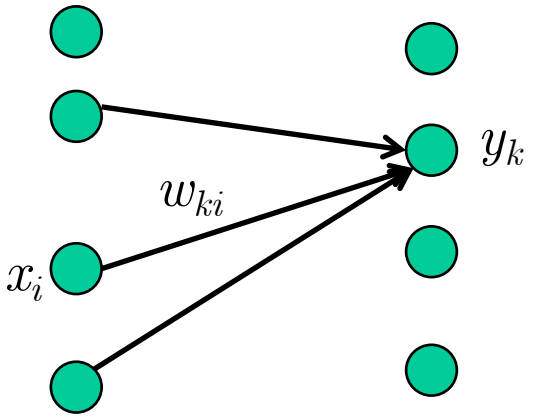
$$E(\boldsymbol{w}) = \sum_{n=1}^{N} E_n(\boldsymbol{w})$$

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

Simple Model (Multiple Linear Regression)

- Outputs y_k are linear combinations of inputs x_i

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



- Error function for a particular input $\boldsymbol{x}_n$ is

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

Where summation is over all K outputs

For a particular input x and weight w , squared error is:

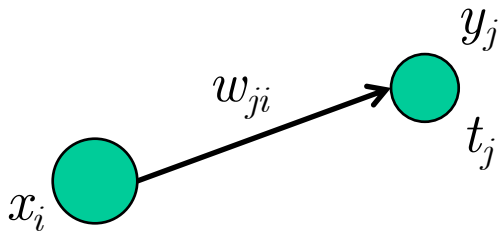
$$E = \frac{1}{2}\left(y(x,w) - t\right)^2$$

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

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

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

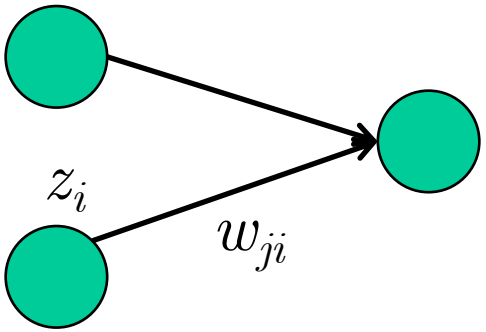
  - a local computation involving product of
    - error signal y_{nj}-t_{nj} associated with output end of link w_{ji}
    - variable x_{ni} associated with input end of link



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

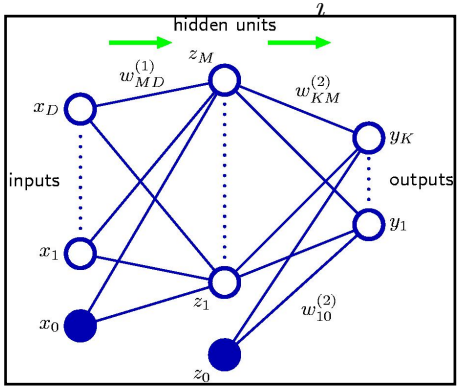
Extension to more complex multilayer Network

- Each unit computes a weighted sum of its inputs $a_j = \sum_i w_{ji} z_i$



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

$$z_j = h(a_j)$$



  - 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
- Output is transformed by a nonlinear activation function $z_j = h(a_j)$
  - The variable z_i can be an input and unit j could be an output
- For each input $\boldsymbol{x_n}$ in the training set, we calculate activations of all hidden and output units by applying above equations
  - This process is called forward propagation

Evaluation of Derivative E_n wrt a weight w_{ji}

- The outputs of the various units depend on particular input n
  - We shall omit the subscript n from network variables
  - Note that E_n depends on w_{ji} only via the summed input a_j to unit j.
  - We can therefore apply chain rule for partial derivatives to give

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

  - Derivative wrt weight is given by product of derivative wrt activity and derivative of activity wrt weight
  - We now introduce a useful notation $\delta_j \equiv \frac{\partial E_n}{\partial a_j}$
  - Where the δs are errors as we shall see
  - Using $a_j = \sum_i w_{ji} z_i$ we can write $\frac{\partial a_j}{\partial w_{ji}} = z_i$
- Substituting we get $\frac{\partial E_n}{\partial w_{ji}} = \delta_j z_i$
  - i.e., required derivative is obtained by multiplying the value of δ for the unit at the output end of the weight by the the value of z at the input end of the weight
  - This takes the same form as for the simple linear model

Summarizing evaluation of Derivative $\frac{\partial E_n}{\partial w_{ji}}$

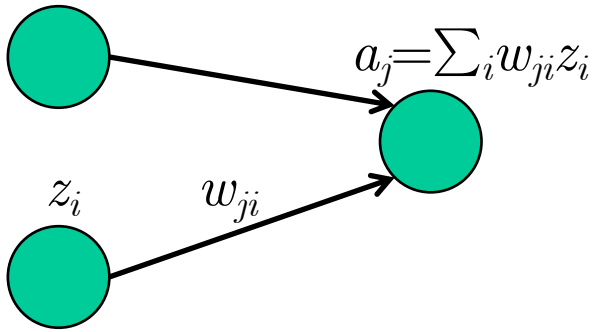
- By chain rule for partial derivatives

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

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

$a_j = \sum_i w_{ji} z_i$

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



- Substituting we get

$$\frac{\partial E_n}{\partial w_{ji}} = \delta_j z_i$$

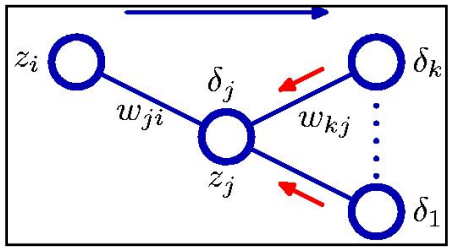
- Thus required derivative is obtained by multiplying
  1. Value of δ for the unit at output end of weight
  2. Value of z for unit at input end of weight
- Need to figure out how to calculate δ_j for each unit of network
  - For output units $\delta_j = y_j - t_j$

$$\text{If } E = \frac{1}{2}\sum_j (y_j - t_j)^2 \text{ and } y_j = a_j = \sum w_{ji} z_i \text{ then } \delta_j = \frac{\partial E}{\partial a_j} = y_j - t_j$$

For regression

  - For hidden units, we again need to make use of chain rule of derivatives to determine $\delta_j \equiv \frac{\partial E_n}{\partial a_j}$

Calculation of Error for hidden unit δ_j



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

- For hidden unit j by chain rule

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

Where sum is over all units k to which j sends connections

- Substituting $\delta_k \equiv \frac{\partial E_n}{\partial a_k}$

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

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

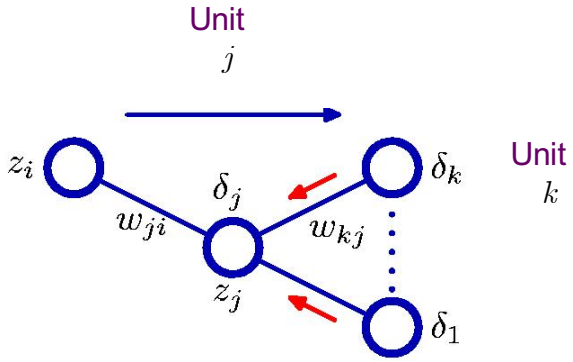
- We get the backpropagation formula for error derivatives at stage j

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

Input to activation from earlier units

error derivative at later unit k

Error Backpropagation Algorithm



- Backpropagation Formula

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

- 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 $\boldsymbol{x}_n$ 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

$$\frac{\partial E_n}{\partial w_{ji}} = \delta_j z_i$$

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

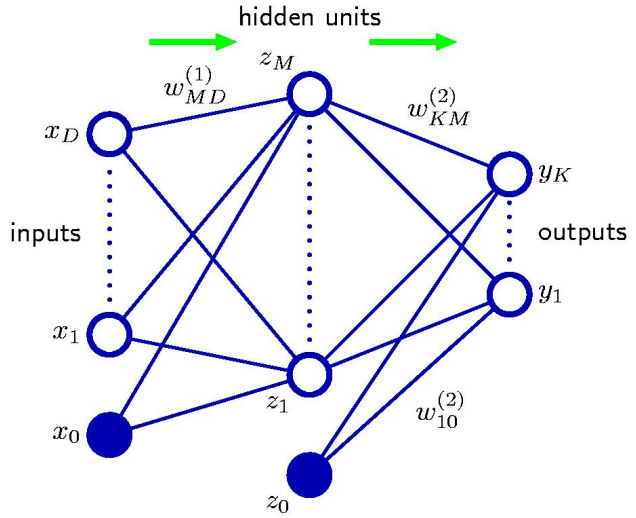
$$h(a) = \tanh(a)$$

where

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

simple form for derivative

$$h'(a) = 1 - h(a)^2$$



Standard Sum of Squared Error

$$E_n = \frac{1}{2}\sum_k \left(y_k - t_k\right)^2$$

y_k: activation of output unit k

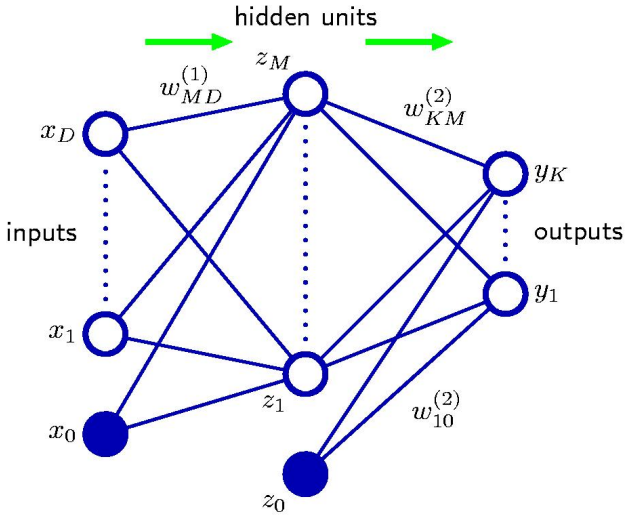
t_k : corresponding target for input x_k

Simple Example: Forward and Backward Prop

For each input in training set:

- Forward Propagation

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



- Output differences

$$\delta_k = y_k - t_k$$

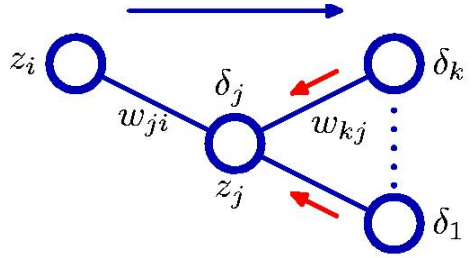
- Backward Propagation (δ s for hidden units)

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

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

- 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

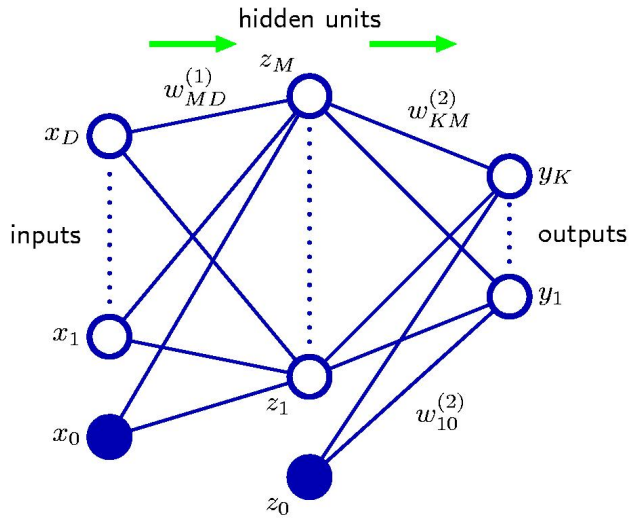
$$\frac{\partial E}{\partial w_{ji}} = \sum_n \frac{\partial E_n}{\partial w_{ji}}$$

Using derivatives to update weights

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

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

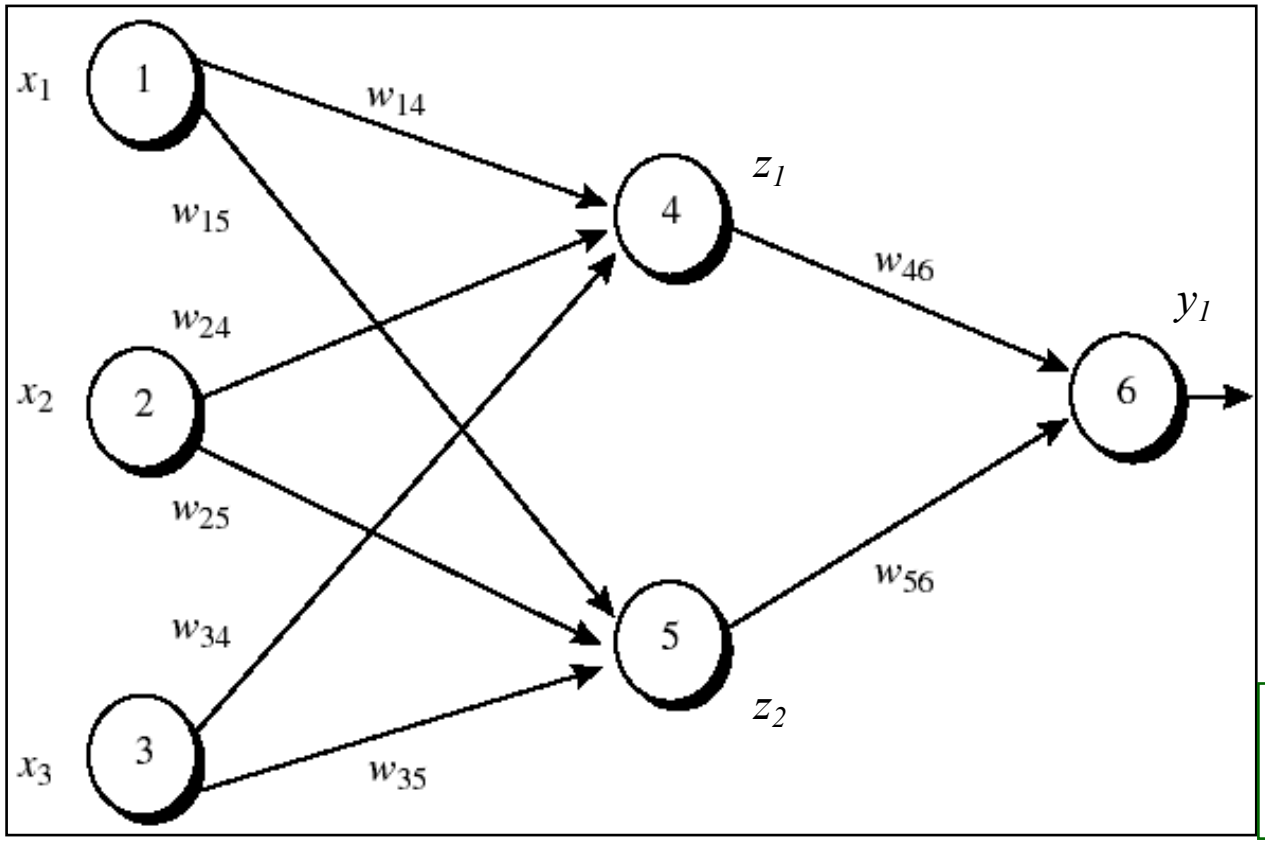
There are $W = M(D+1)+K(M+1)$ elements in the vector
Gradient $\nabla E\left(\boldsymbol{w}^{(\tau)}\right)$ is a W x 1 vector



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, $\boldsymbol{x} = [1 \quad 0 \quad 1]^{\mathrm{T}}$ 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

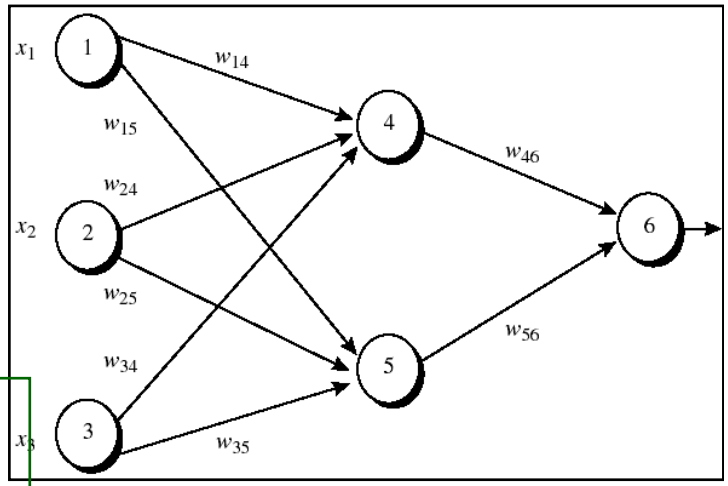
Outputs, Errors, Derivatives, Weight Update

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

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

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
- No 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)

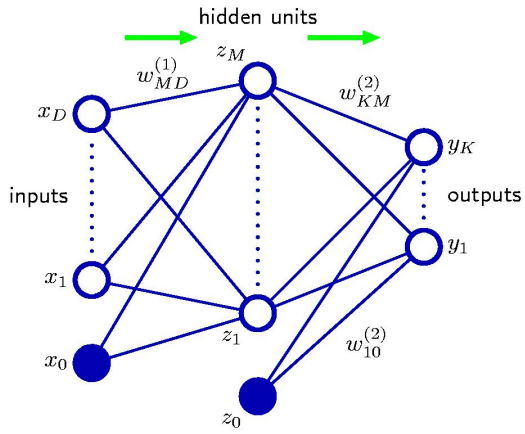
Summary of Backpropagation

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

$$J = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_n} \end{bmatrix} \qquad J_{ki} = \frac{\partial y_k}{\partial x_i}$$



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 are obtained by propagating errors backwards through the network

- The technique of backpropagation can also be used to calculate other derivatives
- Here we consider the Jacobian matrix
  - Whose elements are derivatives of network outputs wrt inputs

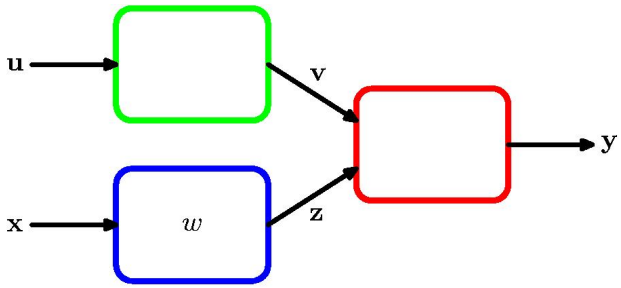
$$J_{ki} = \frac{\partial y_k}{\partial x_i}$$

  - Where each such derivative is evaluated with other inputs fixed

Use of Jacobian Matrix

- Jacobian plays useful role in systems built from several modules
  - Each module has to be differentiable
- Suppose we wish to minimize error E wrt parameter w in a modular classification system shown here:

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



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

Summary of Jacobian Matrix Computation

- Apply input vector corresponding to point in input space where the $\mathrm{Jacobian\ matrix}$ is to be found
- Forward propagate to obtain activations of the hidden and output units in the network
- For each row k of $\mathrm{Jacobian\ matrix}$, 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 in the form

$$\frac{\partial y_k}{\partial x_i} = \frac{y_k(x_i + \varepsilon) - y_k(x_i - \varepsilon)}{2\varepsilon} + O(\varepsilon^2)$$

Summary

- Neural network learning needs learning of weights from samples involves two steps:
  - Determine derivative of output of a unit wrt each input
  - Adjust weights using derivatives
- Backpropagation is a general term for computing derivatives
  - Evaluate δ_k for all output units
    - (using $\delta_k = y_k - t_k$ for regression)
  - 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
- Backpropagation is also useful to compute a Jacobian matrix with several inputs and outputs
  - Jacobian matrices are useful to determine the effects of different inputs