UiO : Department of Informatics
University of Oslo

INF 5860 Machine learning for image classification

Lecture : Backpropagation – learning in neural nets

Anne Solberg

March 3, 2017

Reading material

- Reading material:
  - http://cs231n.github.io/optimization-2/
  - Additional optional material:
  - Lecture on backpropagation in Coursera Course on Machine Learning (Andrew Ng)
    CS 231n on youtube: lecture 4
  - http://yann.lecun.com/exdb/publis/pdf/lecun-98b.pdf
  - http://colah.github.io/posts/2015-08-Backprop/
  - http://neuralnetworksanddeeplearning.com/chap2.html

Notation- forward propagation

Assume that the input is layer 0

$a_i^{(j)}$ - activation of unit i and layer j

$\Theta^{(j)}$ - matrix of weights controlling function mapping from layer j - 1 to j

$\Theta^{(j)}$ has dimension (nodes in layer (j)) $\times$ (nodes in layer (j - 1) + 1)

s_{j-1} nodes in layer j - 1, s_j nodes in layer j: $\Theta^{(j)}$ has size $s_j \times (s_{j-1} + 1)$

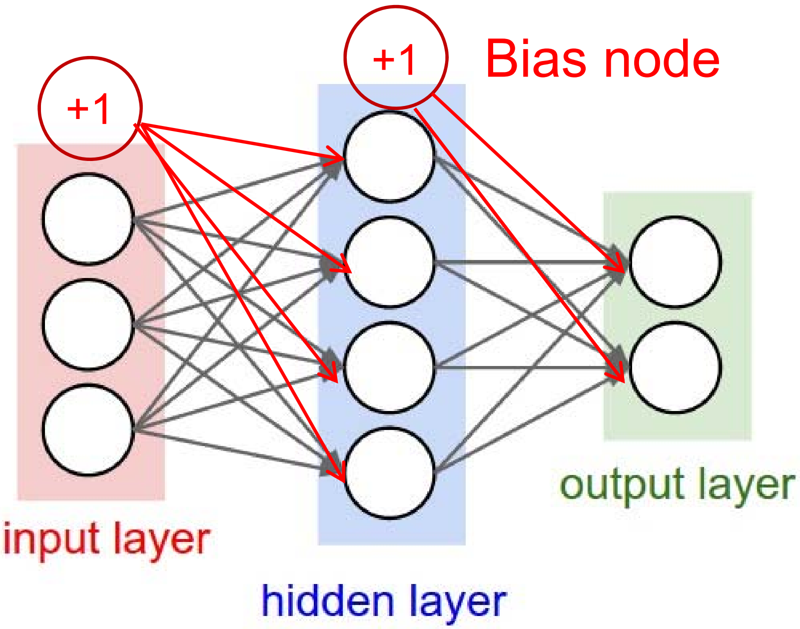
$$a_1^{(1)} = g\left(\Theta_{10}^{(1)} x_0 + \Theta_{11}^{(1)} x_1 + \Theta_{12}^{(1)} x_2 + \Theta_{13}^{(1)} x_3\right)$$

$$a_2^{(1)} = g\left(\Theta_{20}^{(1)} x_0 + \Theta_{21}^{(1)} x_1 + \Theta_{22}^{(1)} x_2 + \Theta_{23}^{(1)} x_3\right)$$

$$a_3^{(1)} = g\left(\Theta_{30}^{(1)} x_0 + \Theta_{31}^{(1)} x_1 + \Theta_{32}^{(1)} x_2 + \Theta_{33}^{(1)} x_3\right)$$

$$h_\Theta(x) = a_1^{(2)} = g\left(\Theta_{10}^{(2)} a_0^{(1)} + \Theta_{11}^{(2)} a_1^{(1)} + \Theta_{12}^{(2)} a_2^{(1)} + \Theta_{13}^{(2)} a_3^{(1)}\right)$$

Example feed-forward computation



- Input x: 3x1 vector

$\Theta^{(1)}: 4\times 4$ (nof. hidden nodes in layer 1 $\times$ nof. inputs + 1)

$\Theta^{(2)}: 2\times 5$ (nof. classes $\times$ nof. hidden nodes in layer 1 + 1)

If we have N training samples we can predict all n = 1....N at one time :

$$X=\begin{bmatrix} 1 & x_{pixel1}(n=1) & x_{pixel2}(1) & x_{pixel3}(1) \\ 1 & x_{pixel1}(n=2) & x_{pixel2}(2) & x_{pixel3}(2) \\ \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots \\ 1 & x_{pixel1}(n=N) & x_{pixel2}(N) & x_{pixel3}(N) \end{bmatrix}$$

```
z1 = Theta1.dot(X)
a1 = sigmoid(z1)
#Append 1 to a1 before computing z2
Continue with layer 2……….
```

Cost function for one-vs-all neural networks

For a neural nets with one-vs-all :

$$\text{Output}: a^L = h_\Theta(x) \in R^K$$

$$J(\Theta) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{k=1}^{K} y_k(i)\log h_{\theta k}(X(i,:)) + (1-y_k(i))\log(1-h_{\theta k}(X(i,:)))\right] + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_j+1}(\Theta_{ji}^{(l)})^2$$

L : number of layers

s_l : Number of units (without bias) in layer l

$$J(\Theta) = \text{LossTerm} + \lambda * \text{RegularizationTerm}$$

Remark: two variations are common:
- Regularize all weights including the bias terms (sum from 0)
- Avoid regularizing the bias terms (sum from 1)

In practise, this choice do not matter.

Cost function for softmax neural networks

For a neural net with softmax loss function :

$$\text{Output}: a^L = h_\Theta(x) = \begin{bmatrix} P(y=1 \mid x,\Theta) \\ P(y=2 \mid x,\Theta) \\ \vdots \\ P(y=K \mid x,\Theta) \end{bmatrix} = \frac{1}{\sum_{k=1}^{K} e^{\Theta_k^T x}} \begin{bmatrix} e^{\Theta_1^T x} \\ e^{\Theta_2^T x} \\ \vdots \\ e^{\Theta_K^T x} \end{bmatrix}$$

$$J(\Theta) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{k=1}^{K} 1\{y_i = k\}\log\left(\frac{e^{\Theta_k^T x_i}}{\sum_{k=1}^{K} e^{\Theta_k^T x_i}}\right)\right] + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{.j}+1}(\Theta_{ji}^{(l)})^2$$

L : number of layers

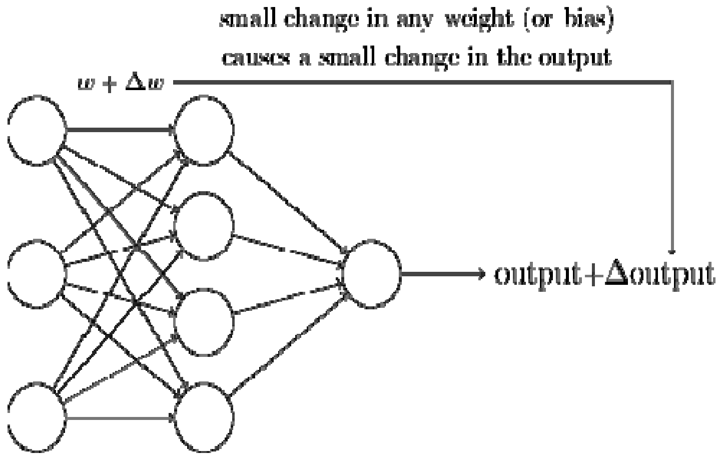
s_l : Number of units (without bias) in layer l

$J(\Theta) = \text{LossTerm} + \lambda * \text{RegularizationTerm}$

Remark: two variations are common,
See previous slide:

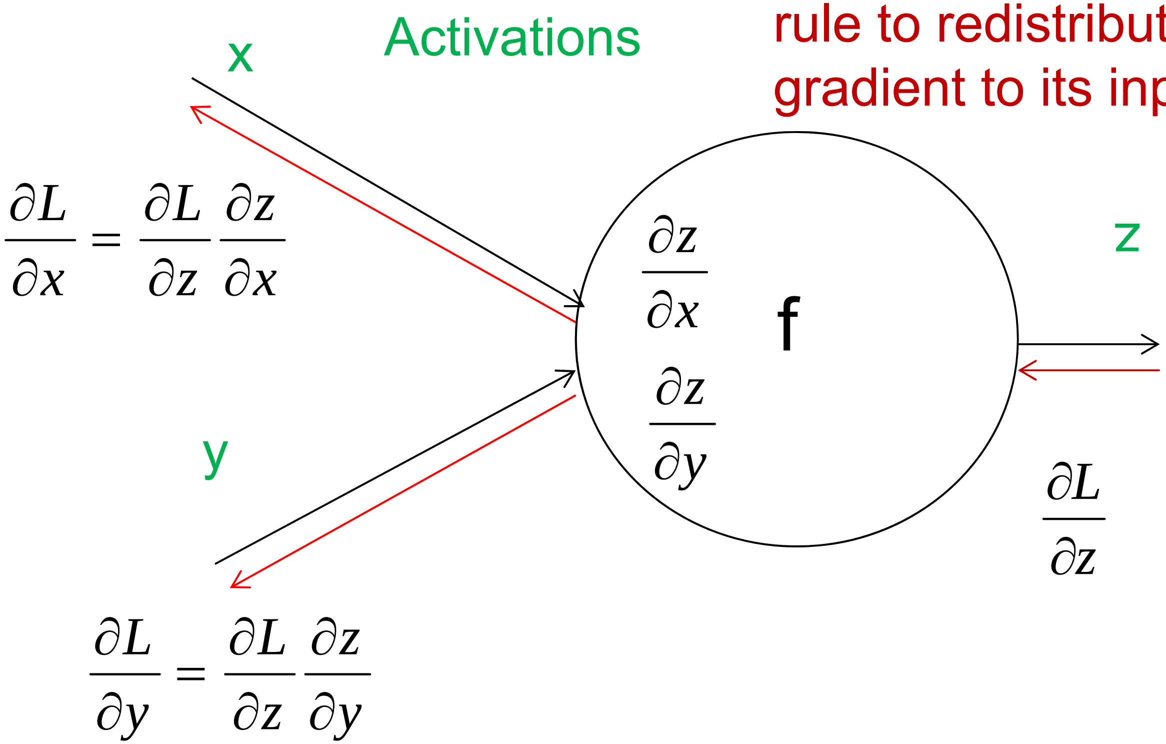
Introduction to backpropagation and computational graphs

- We now have a network architecture and a cost function.
- A learning algorithm for the net should give us a way to change the weights in such a manner that the output is closer to the correct class labels.
- The activation function should assure that a small change in weights results in a small change in ouputs.
- Backpropagation use partial derivatives to compute the derivative of the cost function J with respect to all the weights.



During backpropagation, the node will learn $\frac{\partial L}{\partial z}$

The gate uses chain rule to redistribute this gradient to its inputs



Green numbers: forward propagation
Red numbers: backwards propagation

A more complicated graph example

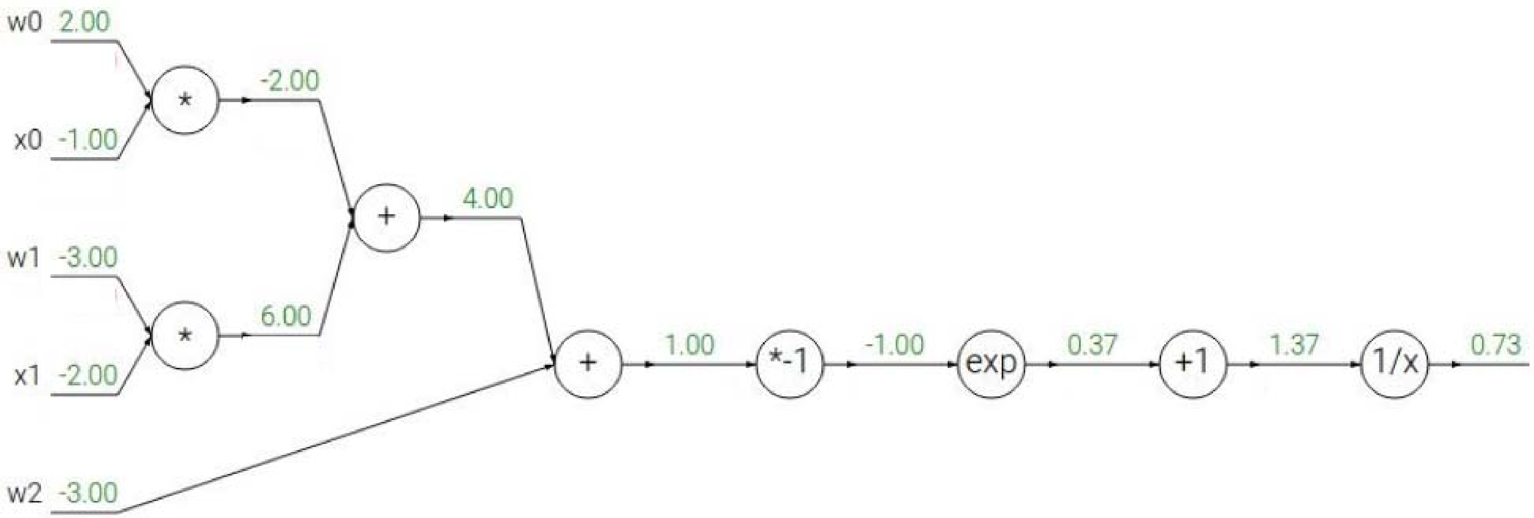
$$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$

$$f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -1/x^2$$

$$f_c(x) = c + x \quad \rightarrow \quad \frac{df}{dx} = 1$$

$$f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x$$

$$f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a$$



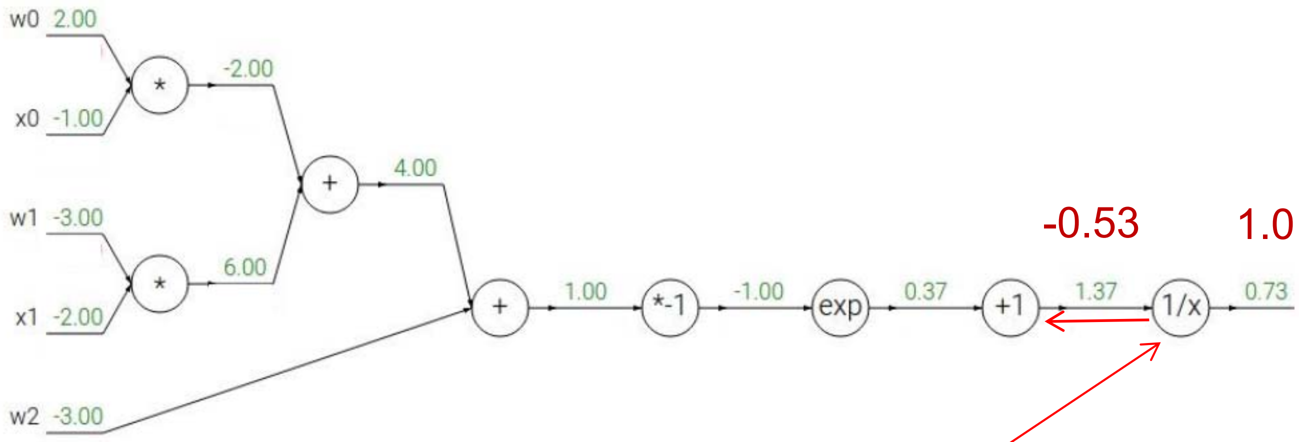
1.0

$$f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -1/x^2$$

$$f_c(x) = c + x \quad \rightarrow \quad \frac{df}{dx} = 1$$

$$f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x$$

$$f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a$$



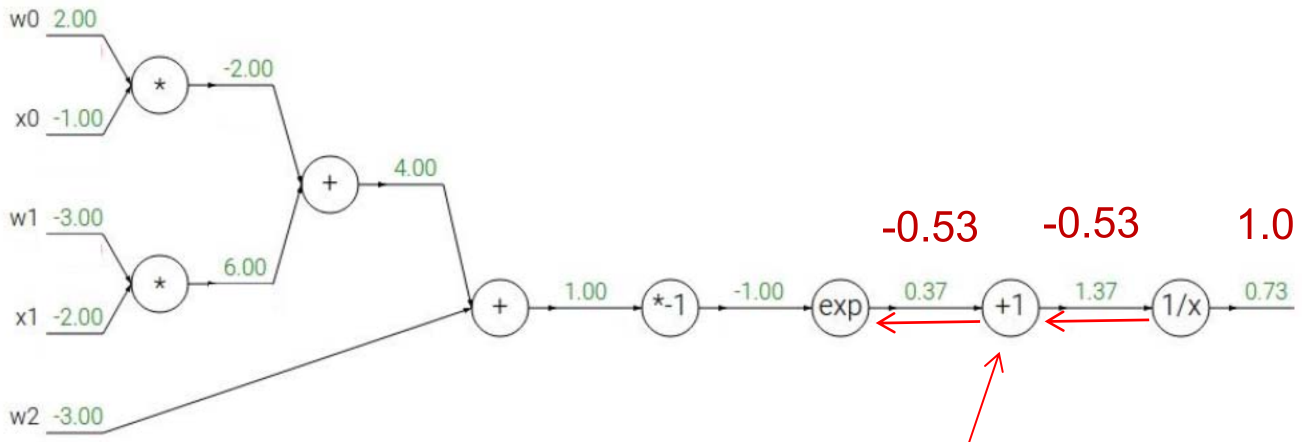
$$f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -1/x^2$$

-1/(1.37)^2*1.0

$$f_c(x) = c + x \quad \rightarrow \quad \frac{df}{dx} = 1$$

$$f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x$$

$$f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a$$



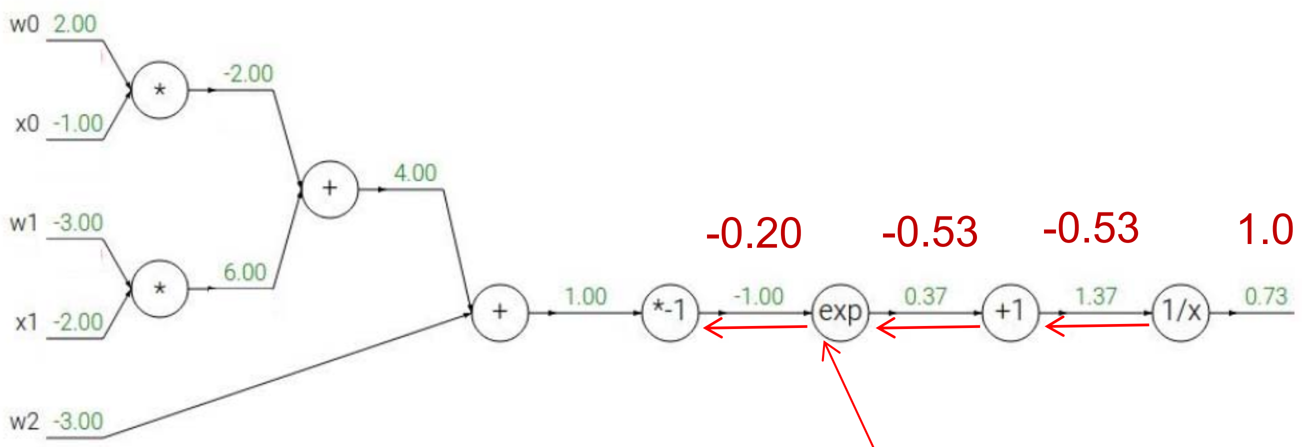
$$f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -1/x^2$$

$$f_c(x) = c + x \quad \rightarrow \quad \frac{df}{dx} = 1$$

1*-0.53=-0.53

$$f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x$$

$$f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a$$



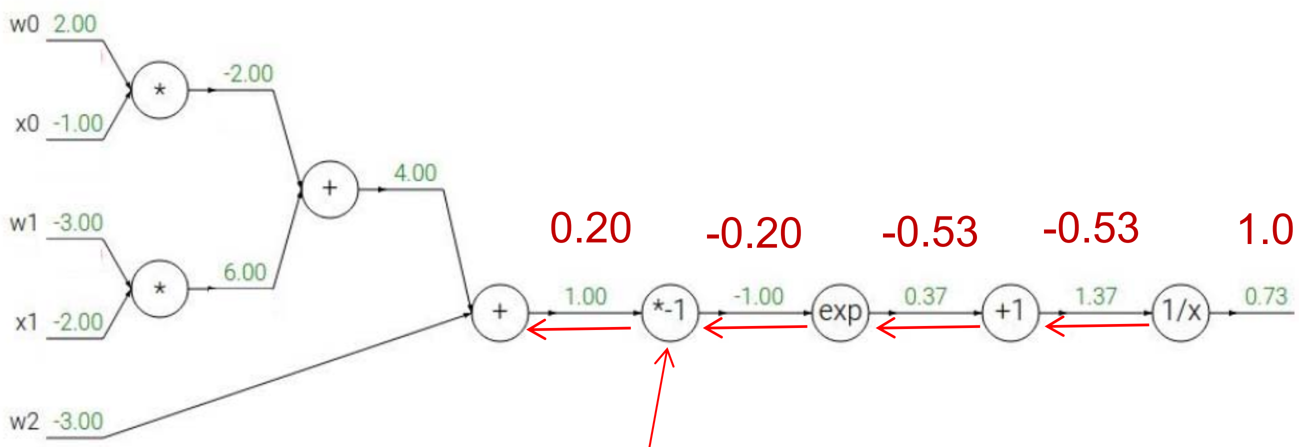
$f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -1/x^2$

$f_c(x) = c + x \quad \rightarrow \quad \frac{df}{dx} = 1$

$f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x$

$f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a$

$e^{-1} * -0.53 = -0.20$



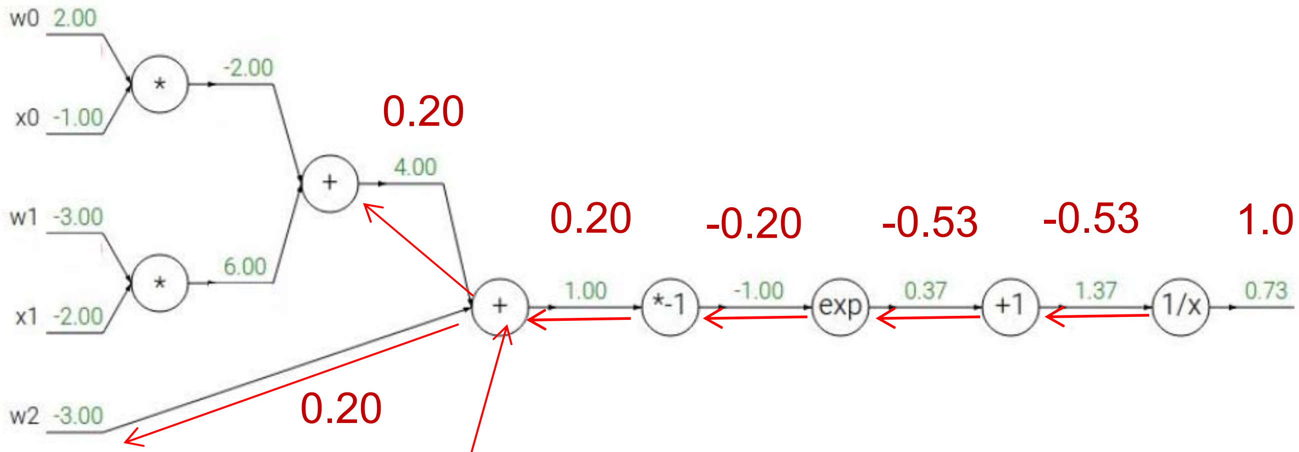
$$f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -1/x^2$$

$$f_c(x) = c + x \quad \rightarrow \quad \frac{df}{dx} = 1$$

$$f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x$$

$$f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a$$

(-1)*-0.20=0.20



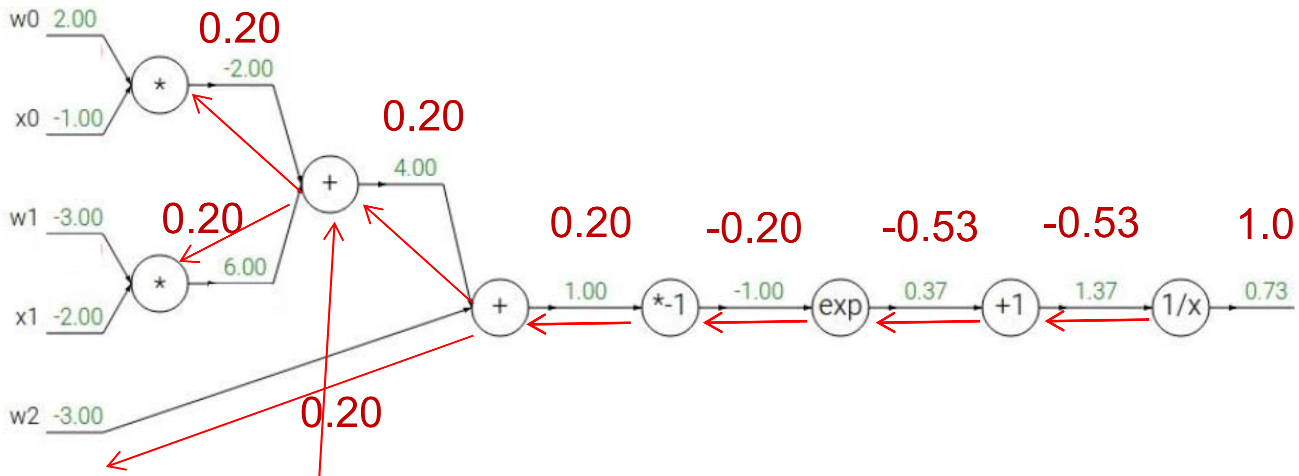
(1)*0.20=0.20
Distribute to both inputs

$$f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -1/x^2$$

$$f_c(x) = c + x \quad \rightarrow \quad \frac{df}{dx} = 1$$

$$f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x$$

$$f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a$$



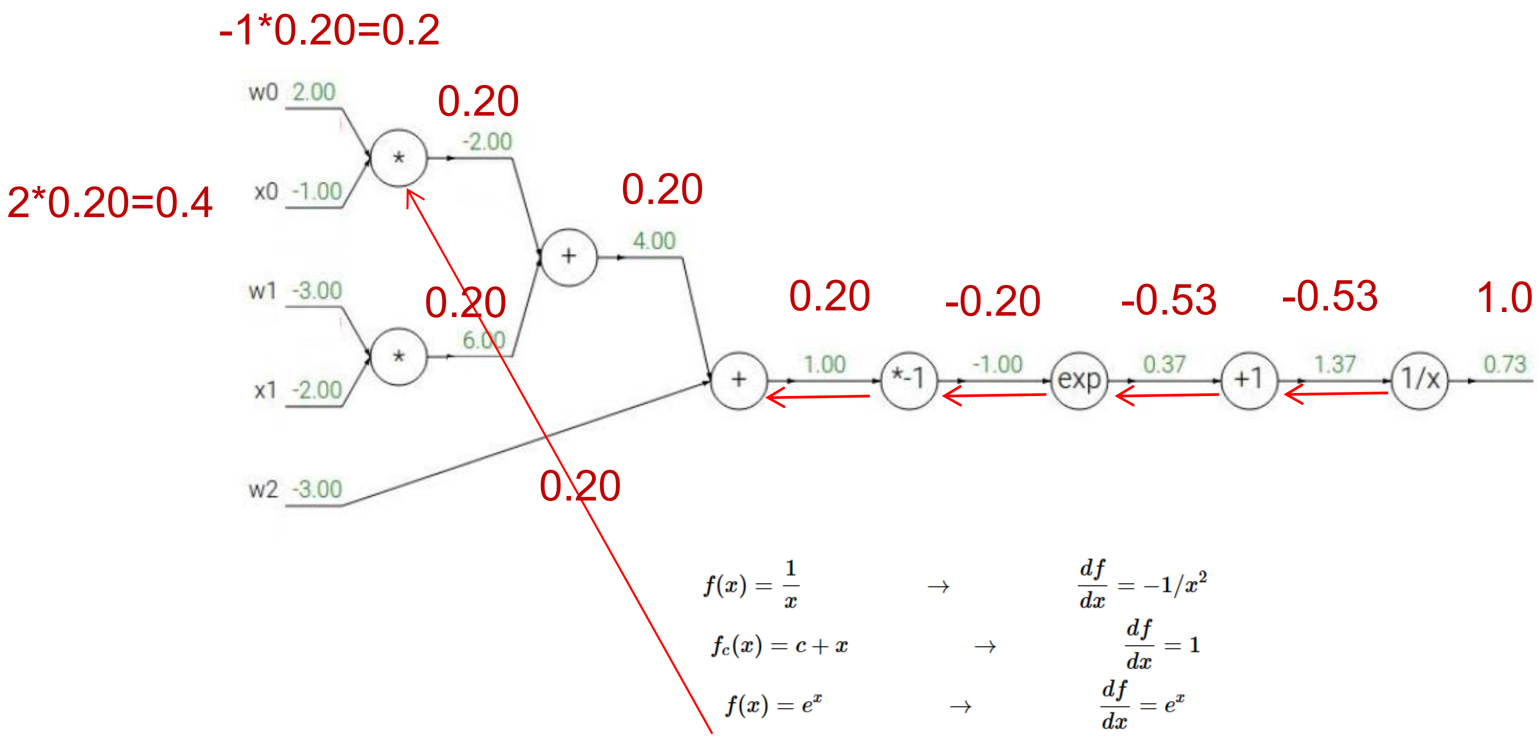
(1)*0.20=0.20
Distribute to both inputs

$$f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -1/x^2$$

$$f_c(x) = c + x \quad \rightarrow \quad \frac{df}{dx} = 1$$

$$f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x$$

$$f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a$$

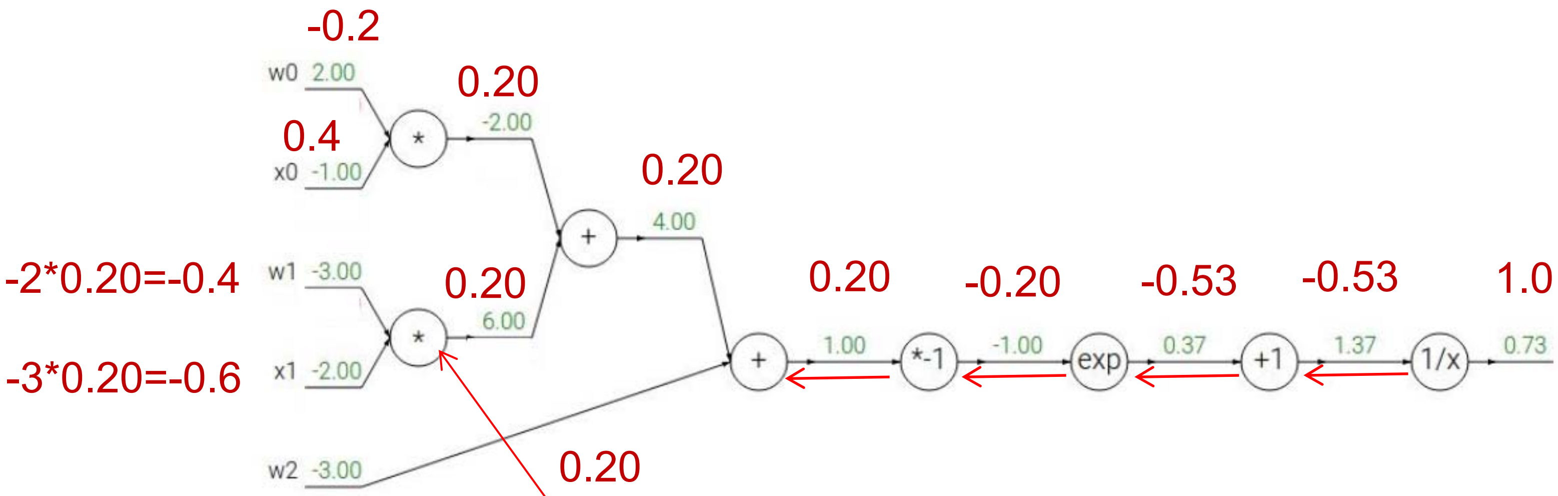


$$f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -1/x^2$$

$$f_c(x) = c + x \quad \rightarrow \quad \frac{df}{dx} = 1$$

$$f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x$$

$$f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a$$

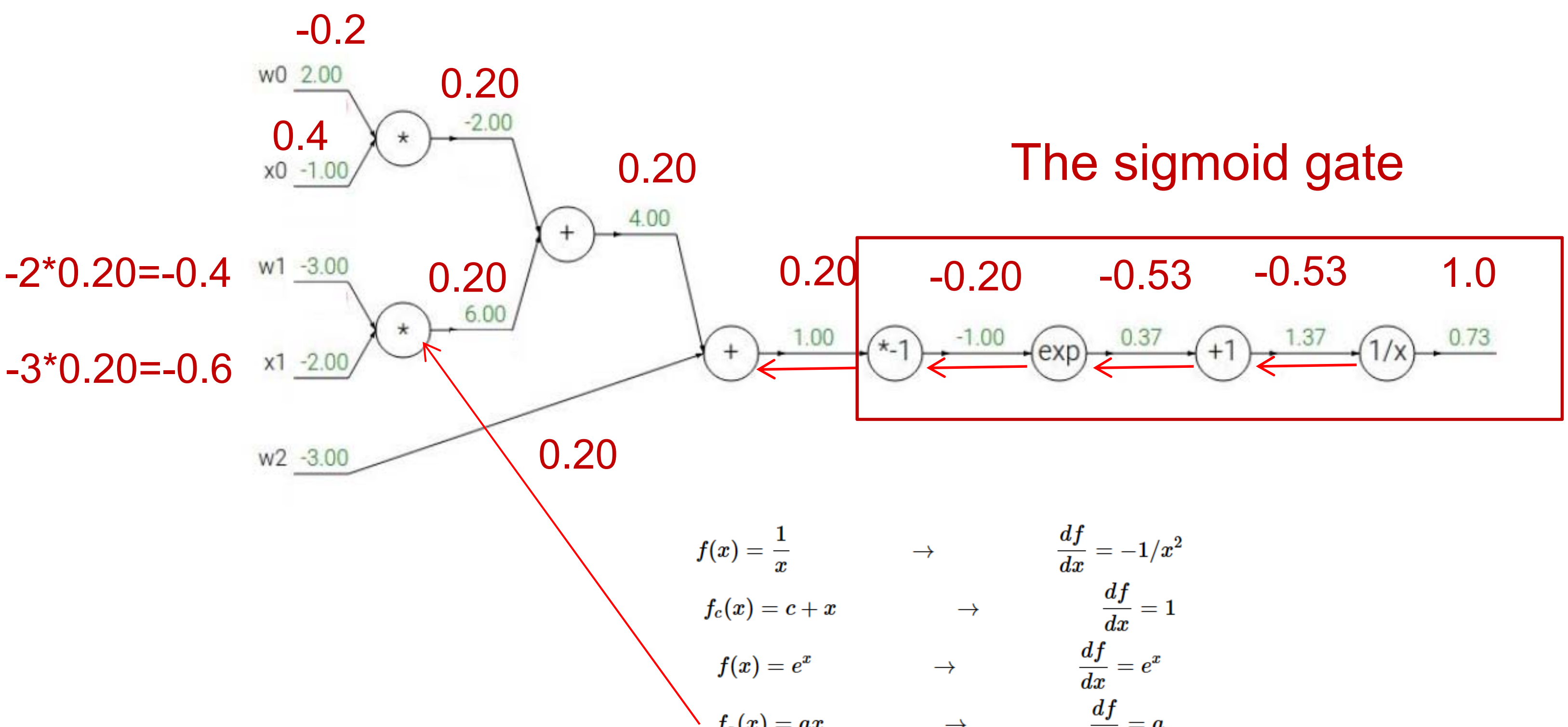


$$f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -1/x^2$$

$$f_c(x) = c + x \quad \rightarrow \quad \frac{df}{dx} = 1$$

$$f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x$$

$$f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a$$



$$f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -1/x^2$$

$$f_c(x) = c + x \quad \rightarrow \quad \frac{df}{dx} = 1$$

$$f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x$$

$$f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a$$

The sigmoid gate

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

$$\frac{d\sigma(x)}{dx} = \frac{e^{-x}}{(1+e^{-x})^2} = \left(\frac{1+e^{-x}-1}{1+e^{-x}}\right)\left(\frac{1}{1+e^{-x}}\right) = (1-\sigma(x))\,\sigma(x)$$

Output: 0.73
Derivative of the sigmoid gate: (1-0.73)0.73=0.20
This is the same result as above.

Forward and backward for a single neuron

```
w = [2,-3,-3] # assume some random weights and data
x = [-1, -2]

# forward pass
dot = w[0]*x[0] + w[1]*x[1] + w[2]
f = 1.0 / (1 + math.exp(-dot)) # sigmoid function

# backward pass through the neuron (backpropagation)
ddot = (1 - f) * f # gradient on dot variable, using the sigmoid gradient derivation
dx = [w[0] * ddot, w[1] * ddot] # backprop into x
dw = [x[0] * ddot, x[1] * ddot, 1.0 * ddot] # backprop into w
# we're done! we have the gradients on the inputs to the circuit
```

Remark: an efficient implemetation will store inputs and intermediates during forward, so that they are available for backprop.

A more tricky example

$$f(x,y) = \frac{x + \sigma(y)}{\sigma(x) + (x+y)^2}$$

- Stage the forward pass into simple operations that we now the derivative of:

```
x = 3 # example values
y = -4

# forward pass
sigy = 1.0 / (1 + math.exp(-y)) # sigmoid in numerator    #(1)
num = x + sigy # numerator                                #(2)
sigx = 1.0 / (1 + math.exp(-x)) # sigmoid in denominator #(3)
xpy = x + y                                               #(4)
xpysqr = xpy**2                                           #(5)
den = sigx + xpysqr # denominator                         #(6)
invden = 1.0 / den                                        #(7)
f = num * invden # done!                                  #(8)
```

A more tricky example

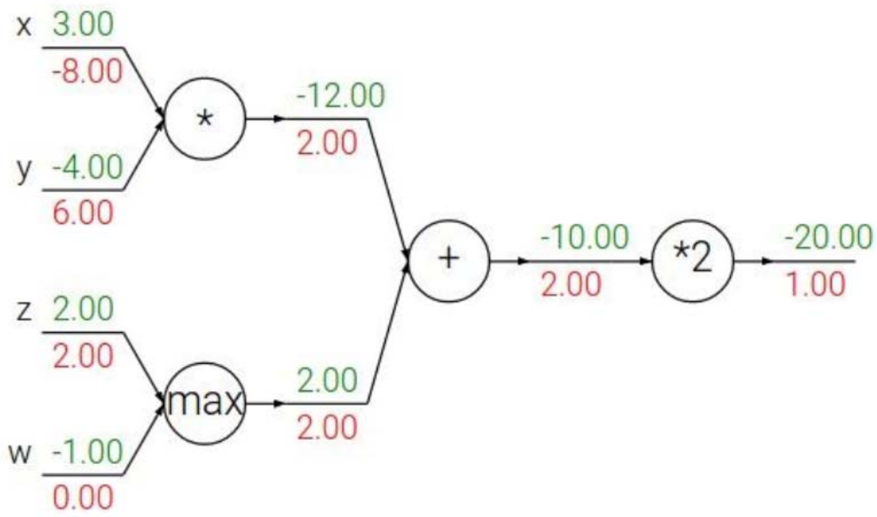
$$f(x,y) = \frac{x + \sigma(y)}{\sigma(x) + (x+y)^2}$$

- In the backwards pass: compute the derivative of all these terms:

```
# backprop f = num * invden
dnum = invden # gradient on numerator                             #(8)
dinvden = num                                                     #(8)
# backprop invden = 1.0 / den
dden = (-1.0 / (den**2)) * dinvden                                #(7)
# backprop den = sigx + xpysqr
dsigx = (1) * dden                                                #(6)
dxpysqr = (1) * dden                                              #(6)
# backprop xpysqr = xpy**2
dxpy = (2 * xpy) * dxpysqr                                        #(5)
# backprop xpy = x + y
dx = (1) * dxpy                                                   #(4)
dy = (1) * dxpy                                                   #(4)
# backprop sigx = 1.0 / (1 + math.exp(-x))
dx += ((1 - sigx) * sigx) * dsigx # Notice += !! See notes below  #(3)
# backprop num = x + sigy
dx += (1) * dnum                                                  #(2)
dsigy = (1) * dnum                                                #(2)
# backprop sigy = 1.0 / (1 + math.exp(-y))
dy += ((1 - sigy) * sigy) * dsigy                                 #(1)
```

Patterns in backward flow

add gate: gradient distributor
max gate: gradient router
mul gate: be careful
f=x*y means that
df/dx=y and df/dy=x



Remark on multiplier gate:
If a gate get one large and one small input, backprop will use the big input to cause a large change on the small input, and vice versa. This is partly why feature scaling is important

The optimization problem

Given a loss function J and a feed - forward net with L layers with weights $\Theta^{(l)}$

We want to minimize J using gradient descent

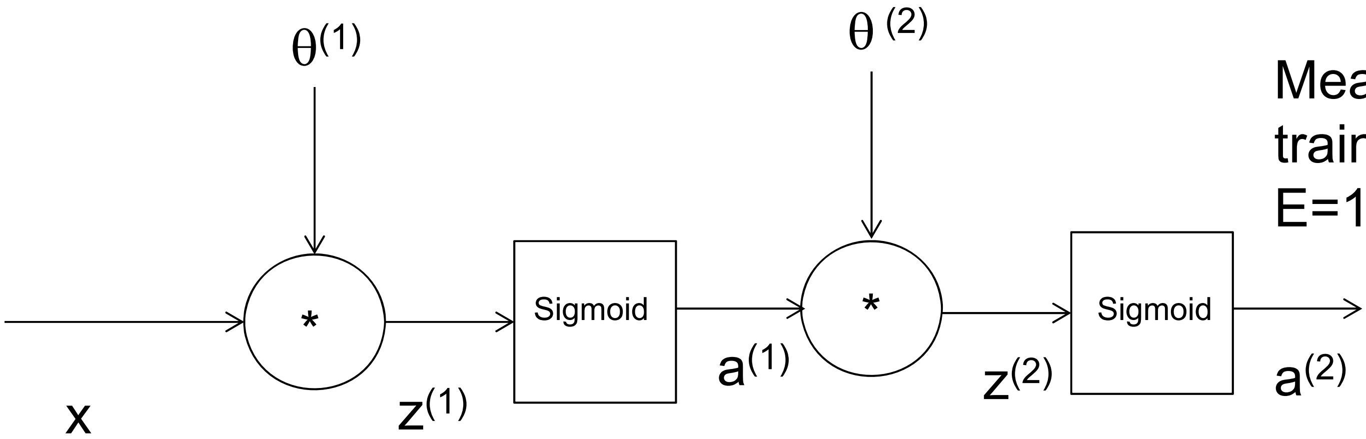
Need the derivatives of J with respect to every $\Theta_{m,n}^{(l)}$

Backpropagation: recursive application of the chain rule on a computational graph to compute the gradients of all input/parameters/intermediates

Implementation:

- Forward: compute the result of the node operation and save the intermediates needed for gradient computation
- Backwards: apply the chain tule to compute the gradients of the loss function with respect to the input of each node.

A very simple net with one input



Measure of error on training data
$E=1/2(y-a^{(2)})^2$

Assume that we want to minimize the square error between the output $a^{(2)}$ and the true class y
$E=1/2(y-a^{(2)})^2$ (Mean square error in this example)
Compute the partial derivatives with respect to $\theta^{(1)}$ and $\theta^{(2)}$, $\frac{\partial E}{\partial \theta^{(1)}{}_1}$ and $\frac{\partial E}{\partial \theta^{(2)}}$
and use use gradient descent to update $\theta^{(1)}$
and $\theta^{(2)}$

$$\frac{\partial E}{\partial \theta^{(2)}} = \frac{\partial E}{\partial a^{(2)}} \frac{\partial a_2^{(2)}}{\partial \theta^{(2)}} = \left(a^{(2)} - y\right) \frac{\partial a^{(2)}}{\partial z^{(2)}} \frac{\partial z_2^{(2)}}{\partial \theta^{(2)}}$$

$$= \left(a^{(2)} - y\right) \frac{\partial a_2^{(2)}}{\partial z^{(2)}} a^{(1)}$$

$a^{(2)}$ applies the sigmoid function g(z) so $\frac{\partial a^{(2)}}{\partial z^{(2)}} = g'(z^{(2)}) = g(z^{(2)})\left(1 - g(z^{(2)})\right)$

$$\frac{\partial E}{\partial \theta^{(1)}} = \frac{\partial E}{\partial a^{(2)}}\frac{\partial a^{(2)}}{\partial \theta^{(1)}} = \left(a^{(2)} - y\right)\frac{\partial a^{(2)}}{\partial z^{(2)}}\frac{\partial z^{(2)}}{\partial \theta^{(1)}}$$

$$= \left(a^{(2)} - y\right)g(z^{(2)})\left(1 - g(z^{(2)})\right)\frac{\partial z^{(2)}}{\partial a^{(1)}}\frac{\partial a^{(1)}}{\partial \theta^{(1)}}$$

$$= \left(a^{(2)} - y\right)g(z^{(2)})\left(1 - g(z^{(2)})\right)\theta^{(2)}\frac{\partial a^{(1)}}{\partial z^{(1)}}\frac{\partial z^{(1)}}{\partial_1^{(1)}}$$

$$= \left(a^{(2)} - y\right)g(z^{(2)})\left(1 - g(z^{(2)})\right)\theta^{(2)}g(z^{(1)})\left(1 - g(z_1^{(1)})\right)\frac{\partial z^{(1)}}{\partial \theta^{(1)}}$$

$$\left(a^{(2)} - y\right)g(z^{(2)})\left(1 - g(z^{(2)})\right)\theta^{(2)}g(z^{(1)})\left(1 - g(z^{(1)})\right)x$$

$a^{(1)}$ applies the sigmoid function g(z) so $\frac{\partial a^{(1)}}{\partial z^{(1)}} = g'(z^{(1)}) = g(z^{(1)})\left(1 - g(z^{(1)})\right)$

From scalars to vectors

- In the example x was a scalar, and $\frac{\partial E}{\partial \theta}$ was a vector with one element pr. weight.
- When working with vector input, for each layer $\frac{\partial E}{\partial \Theta^l}$ will be a matrix.
- Deriving the vector/matrix version of backpropagation is more tedious, but follows the same principle.
- A good source is

http://neuralnetworksanddeeplearning.com/chap2.html

- We now present the vector algorithm

Backpropagation algorithm for a single training sample (x_i, y_i)

For now, ignore the regularization (set $\lambda = 0$)

For a 3-layer net:

Let $\delta_j^{(3)} = a_j^{(3)} - y_j$

Let $\delta^{(3)} = a^{(3)} - y$ be the vector of $\delta_j^{(3)}$ $j = 1,..s_j$, where s_j is the number of nodes in layer j

$$\text{Compute } \delta^{(2)} = \left(\left(\Theta^{(3)} \right)^T \delta^{(3)} \right) .* g'(z^{(2)})$$

$$\delta^{(1)} = \left(\left(\Theta^{(2)} \right)^T \delta^{(2)} \right) .* g'(z^{(1)})$$

Note that this is the elementwise product, or Hadamard-product of two vectors

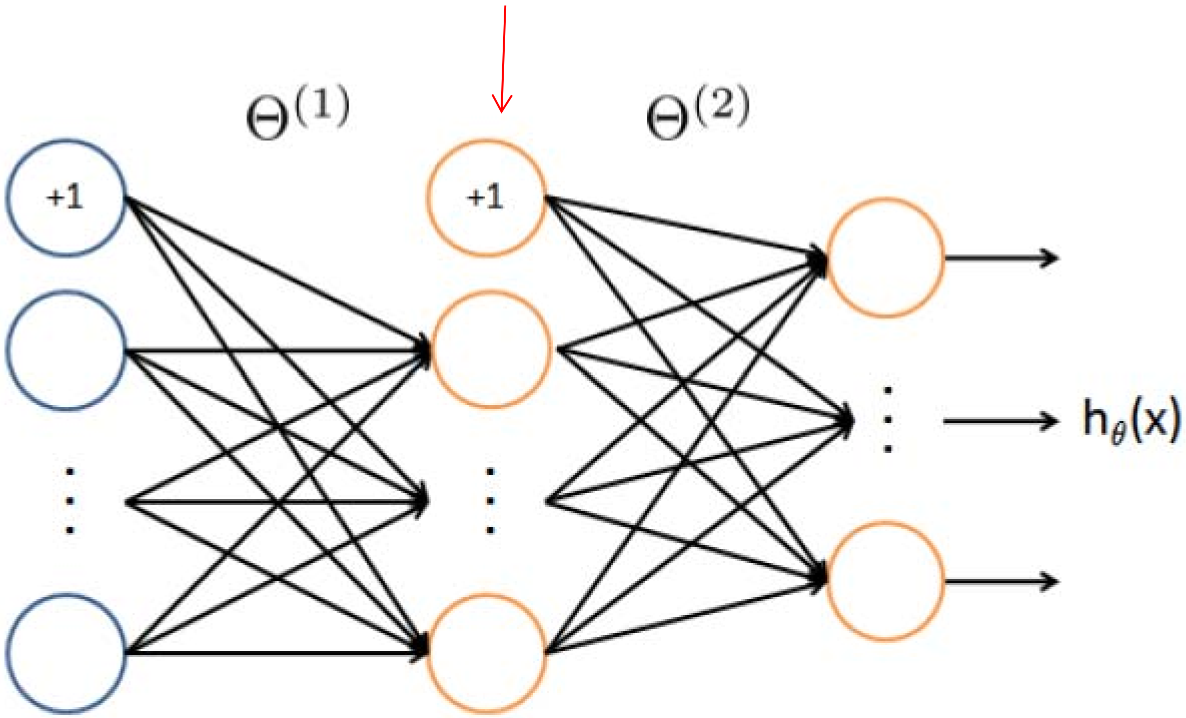
With this notation, $\frac{\partial J}{\partial \Theta_{ij}^{(l)}} = a_j^{(l)} \delta_i^{l+1}$

Derivative of loss function

- In backpropagation, we need the derivative of the loss functions with respect to the activation of the output layer a_i^L.
- If we ignore the regularization term, the derivative of the logistic loss function for sample *i* can be shown to be $(a_i^L - y_i)$
  - See http://stats.stackexchange.com/questions/219241/gradient-for-logistic-loss-function
- For softmax, ignoring the regularization term, the derivative of the softmax loss is also $(a_i^L - y_i)$
  - See http://math.stackexchange.com/questions/945871/derivative-of-softmax-loss-function

NOTE: a_i^L is computed differently

Notice that the bias nodes do not receive input from previous layer.
Thus, they should NOT be used in backpropagation



$$\delta^{(1)} = \left(\left(\Theta^{(2)}\right)^T \delta^{(2)}\right) .* g'(z^{(1)}) \qquad \delta_k^{(2)} = a_k^{(2)} - yind(k)$$

Including the regularization term

$$J(\Theta) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{k=1}^{K} y_k(i)\log h_{\theta k}(X(i,:)) + (1-y_k(i))\log(1-h_{\theta k}(X(i,:)))\right] + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{j}+1}(\Theta_{ji}^{(l)})^2$$

$$J(\Theta) = \text{LossTerm} + \lambda * \text{RegularizationTerm}$$

Backpropagation update including the regularization:

$$\frac{\partial J}{\partial \Theta_{ij}^{(l)}} = D_{ij}^{(l)} = \frac{1}{m}\Delta_{ij}^{(l)} \quad \text{for j} = 0\text{, here the convention is that we do not regularize the bias terms}$$

$$\frac{\partial J}{\partial \Theta_{ij}^{(l)}} = D_{ij}^{(l)} = \frac{1}{m}\Delta_{ij}^{(l)} + \frac{\lambda}{m}\Theta_{ij}^{(l)} \quad \text{for j} =\geq 1$$

Note that i is indexed from 1, and j from 0 (it gets input from the bias in the previous layer)

Remark: softmax will have the same regularization term

Backpropagation with a loop over training data

Training set $\{(x_1, y_1), \ldots..(x_m, y_m)\}$

Set $\Delta_{ij}^{(l)} = 0$ for all i, j, l

for $i = 1 : m$

Set $a^{(0)} = x_i$

Do forward propagation to compute $a^{(0)}, l = 1, \ldots L-1$

Compute $\delta_k^{(L-1)} = a_k^{(L-1)} - yind(k)_i$, yind is an indicator function, $= 1$ if $y_i = k$ and 0 otherwise

Compute $\delta^{(L-2)}, \ldots.\delta^{(1)} as\ \delta^{(l)} = \left((\Theta^{(l)})^T \delta^{(l+1)}\right) .* g'(z^{(l)})$

Set $\Delta_{ij}^{(l)} = \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$

$$D_{ij}^{(l)} = \frac{1}{m} \Delta_{ij}^{(l)} + \lambda \Theta_{ij}^{(l)}, \text{ if } j \neq 0$$

$$D_{ij}^{(l)} = \frac{1}{m} \Delta_{ij}^{(l)}, \qquad \text{if } j = 0$$

Here, $\frac{\partial J}{\partial \Theta_{ij}^{(l)}} = D_{ij}^{(l)}$

Checking dimensions

$$\delta^{(2)} = \left(\left(\Theta^{(2)}\right)^T \delta^{(3)}\right) .* g'(z^{(2)})$$

- Note that in backpropagation, we use Θ^T
- When implementing this shape() is your best friend ☺

- Think of a net with one hidden layer (layer 1) with 25 nodes + bias, and output layer with 10 nodes (10 classes)
- $\Theta^{(2)}$ has dimension 10x26 including bias, and $(\Theta^{(2)})^T$ is 26x10
- $\delta^{(2)}$ has dimension 10x1

- REMARK: we can either ignore the bias terms in backpropagation, or compute $\delta_0^{(1)}$ also (resulting in a 26x1 vector), but later ignore the $\delta_0^{(1)}$ values
  - When doing backpropagation from layer 2 to layer 1, ignore the bias in (index 0 of layer 2) and backpropagate $(\Theta^{(2)})^T(1:25,0:9)$
- $\delta^{(1)}$ then has dimension [(25x10)x(10x1)]$\cdot*$(25x1) = 25x1

Assumptions behind backpropragation

1. The loss function should be expressed as a sum or average over all training samles.
   - This is true for all the functions we have studied so far
   - We will be able to compute $\frac{\partial L}{\partial \Theta_{ij}^{l}}$ for a single training example, and then average over all samples.

$$\text{Output}: \mathrm{h}_{\Theta}(\mathrm{x}) \in \mathrm{R}^{\mathrm{K}}$$

$$J(\Theta) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{k=1}^{K} y_k(i)\log h_{\theta k}(X(i,:)) + (1-y_k(i))\log(1-h_{\theta k}(X(i,:)))\right] + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{.j}+1}(\Theta_{ji}^{(l)})^2$$

L : number of layers

s_l : Number of units (without bias) in layer l

Assumptions behind backpropragation

2. The loss function must be expressed as a function of the outputs of the net.
   - This allows us to change the weights and measure how similar y_i and the output $h_\Theta(x)$ is.

$$\text{Output}: h_\Theta(x) \in R^K$$

$$L(\Theta) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{k=1}^{K} y_k(i)\log h_{\theta k}(X(i,:)) + (1-y_k(i))\log(1-h_{\theta k}(X(i,:)))\right] + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{.j}+1}(\Theta_{ji}^{(l)})^2$$

L : number of layers

s_l : Number of units (without bias) in layer l

Gradient checking

- When implementing backpropagation, we use gradient checking to verify the implementation.
- When the code works, we turn off gradient checking.
- But what is it?

Gradient checking: numerical estimation of the gradient

- The gradient of a function is defined as:

$$\frac{d}{d\theta} J(\theta) = \lim_{\varepsilon \to 0} \frac{J(\theta+\varepsilon) - J(\theta-\varepsilon)}{2\varepsilon}$$

- When we have the cost function implemented, we can easily approximate the gradient θ as

$$\frac{J(\theta+\varepsilon) - J(\theta-\varepsilon)}{2\varepsilon}$$

Procedure for gradient checking

- 'Unroll' Θ_1, Θ_2,…into a 1-d vector $\theta = [\theta_1, \ldots \theta_n]$
- Approximate

$$\frac{\partial J}{\partial \theta_1} = \frac{J(\theta_1 + \varepsilon, \theta_2, \ldots \theta_n) - J(\theta_1 - \varepsilon, \theta_2, \ldots \theta_n)}{2\varepsilon}$$

$$\frac{\partial J}{\partial \theta_2} = \frac{J(\theta_1, \theta_2 + \varepsilon, \ldots \theta_n) - J(\theta_1, \theta_2 - \varepsilon, \ldots \theta_n)}{2\varepsilon}$$

$$\vdots$$

$$\frac{\partial J}{\partial \theta_n} = \frac{J(\theta_1, \theta_2, \ldots \theta_n + \varepsilon) - J(\theta_1, \theta_2, \ldots \theta_n - \varepsilon)}{2\varepsilon}$$

- Check that the difference between this partial derivative and the one from backpropagation is smaller than a threshold.

Regarding gradient checking:

- Computing the approximated gradient is computationally much slower than backpropagation:
  - Use gradient checking for a small example when debugging the backpropagation code.
  - Once it works, turn off gradient checking and proceed with training the entire data set.

Random initialization of weights

- All weights must be initialized to small, but different random numbers.
  - More on why next week.

Training a neural network

- Choose an architecture:
  - Number of inputs: dimension of feature vector or image
  - Number of outputs: number of classes
  - 1-2 hidden layers.
    - For simplicity: use the same number of nodes in each hidden layer
  - More on practial details in the next two lectures.

Training a network

1. Randomly initialize each weight to small numbers
2. Implement forward propagation to get the output
3. Implement code to compute the cost function $J(\theta)$
4. Implement backprop to compute the partial derivatives

   for i=1:m

   Perform forward propagation and backpropagation for sample x_i,y_i
5. Use gradient checking to compate numerical estimates and backpropagation gradients. Afterward, disable gradient checking.
6. Use gradient descent (or optimization methods) with backpropagation to minimize J.

Weekly exercise:

- A detailed programming exercise, with descriptions on the operations, will be available.
- Implementing backpropagation is central to Mandatory exercise 1
  - No solution in python will be given, but test data with known results.

Next weeks:

- Training in practice, useful tricks.
- Babysitting the training process
- Parameter updates
- Activation functions
- Weight initialization
- Preprocessing
- Evaluation

Main reading material: http://cs231n.github.io/neural-networks-3/