UiO : Department of Informatics
University of Oslo

INF 5860 Machine learning for image classification

Lecture 6 : Introduction to neural nets

Anne Solberg

February 25, 2017

Reading material

- Reading material:
  - http://cs231n.github.io/neural-networks-1/
  - http://cs231n.github.io/neural-networks-2/
  - http://cs231n.github.io/optimization-2/
    Youtube: CS 231n: Lectures 4-6 covers the next 3 lectures
  - Deep learning Chapter 6.1-6.5

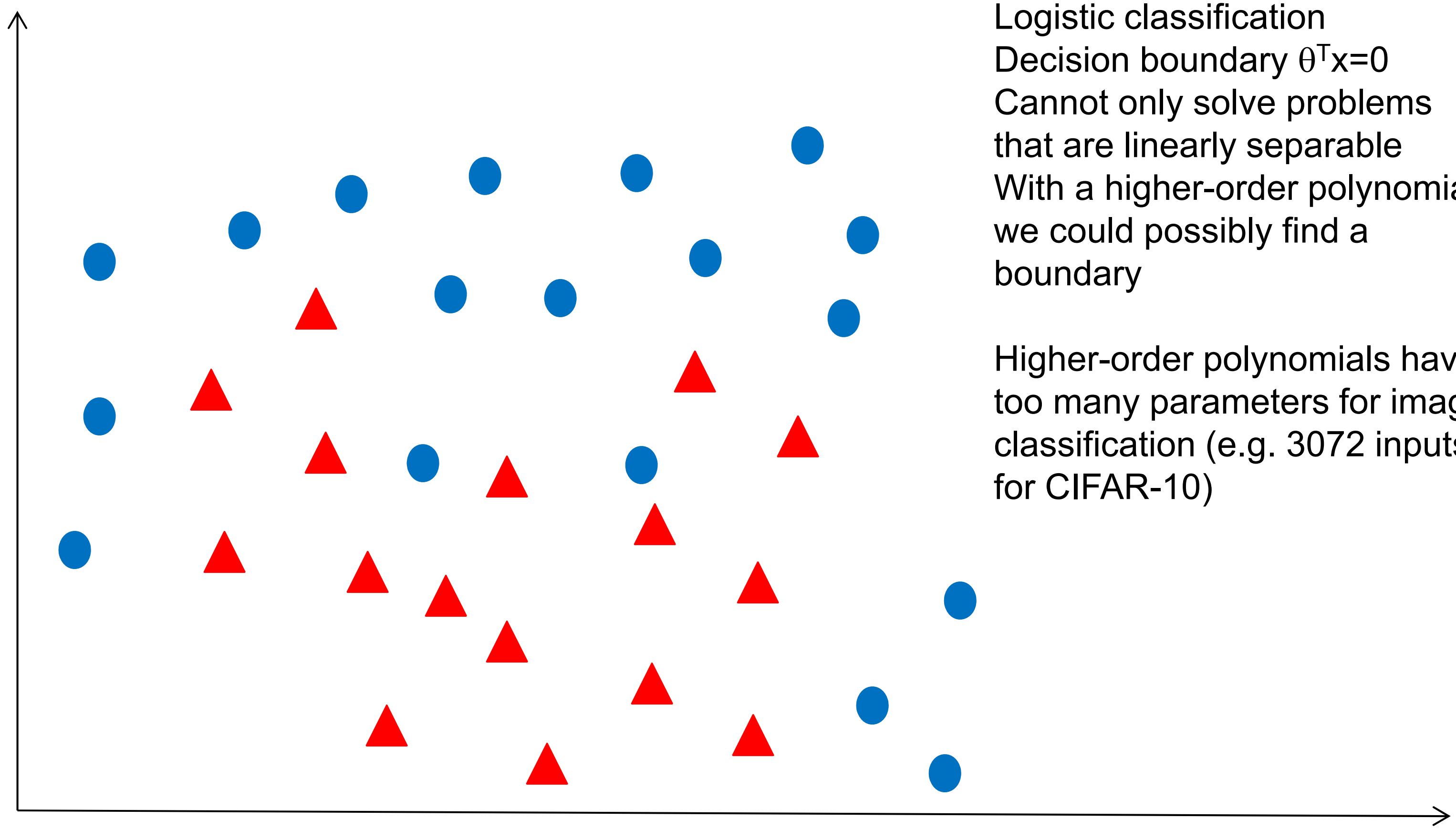
Today

- The concept of feed-forward neural nets
- Capacity of traditional feed-forward nets
- Forward propagation from input to output class labels
- Cost functions for neural net classification
- Net architecture
- Introduction to learning using backpropagation (as far as time permits)
  - Backpropagation in detail next week.

Feed-forward neural nets

- The focus today is a feed-forward neural net with few hidden layers.
- Input will be the image pixel values
  – Or features like SIFT, orientation histograms etc.
- The net will play the role of a classifier that maps the input data through some hidden layers to a score for each class.
  For a network with 2 layers, the score would be s=f(W2*f(W1*x))
  – f is a non-linear function called activation function
- Later, we will see *recurrent* networks that feeds the output back to itself.

Non-linear data example



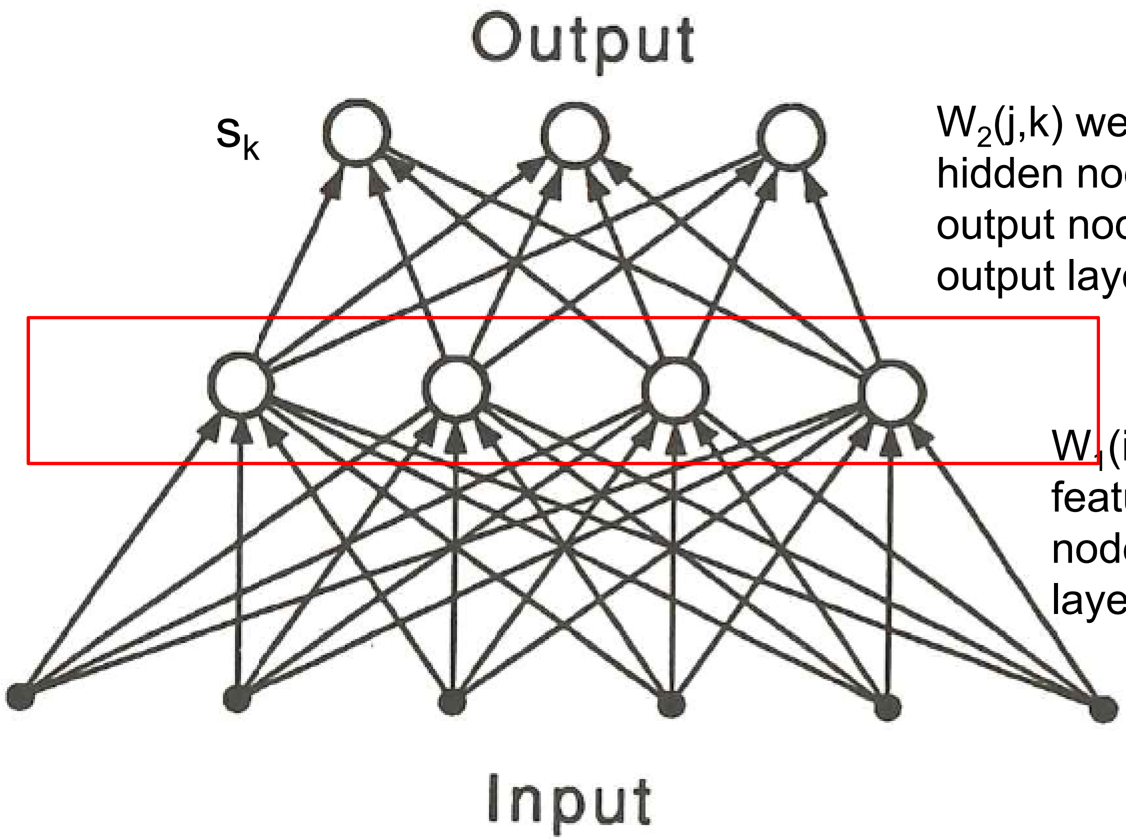
Logistic classification
Decision boundary $\theta^T x=0$
Cannot only solve problems that are linearly separable
With a higher-order polynomial we could possibly find a boundary

Higher-order polynomials have too many parameters for image classification (e.g. 3072 inputs for CIFAR-10)

Two-layer net

Hidden layer

s_k



$W_2(j,k)$ weight for hidden node k to output node j in the output layer

$W_1(i,j)$ weight for input feature j to node i in the hidden layer

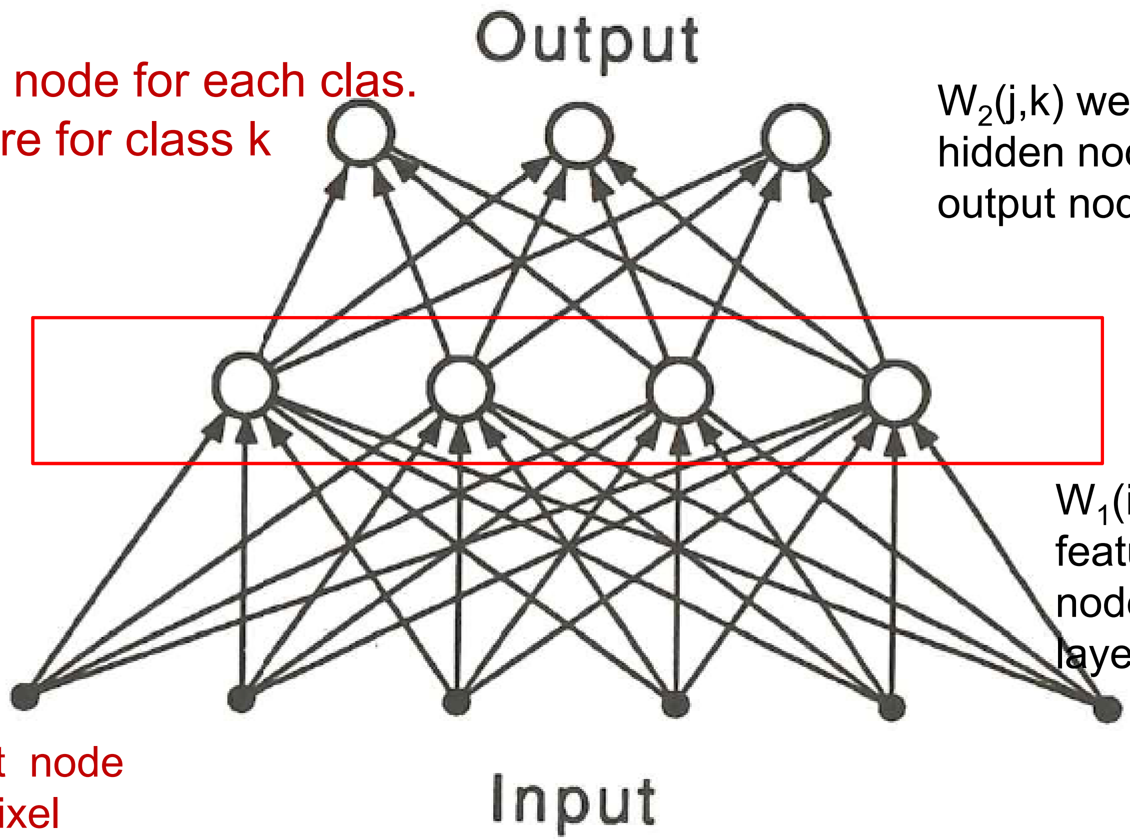
x_i

Two-layer net for image classification

One output node for each clas.
s_k is the score for class k

$W_2(j,k)$ weight for hidden node k to output node j

Hidden layer



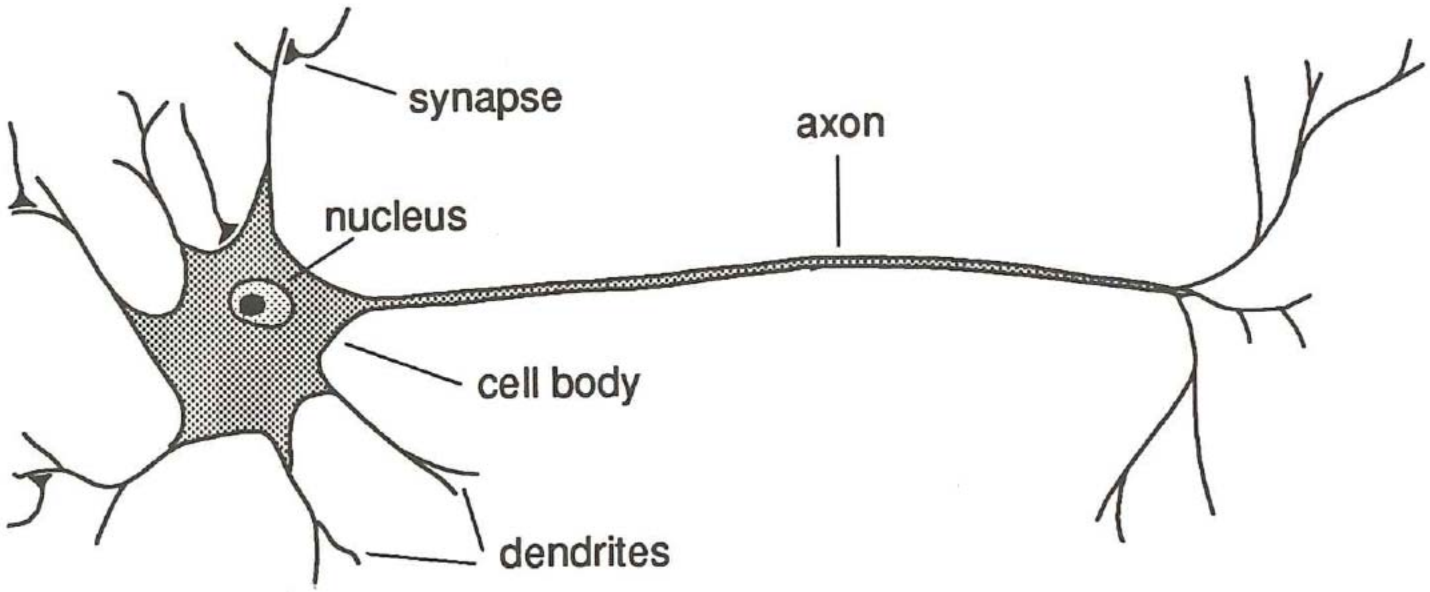
$W_1(i,j)$ weight for input feature j to node i in the hidden layer

x_i one input node for each pixel

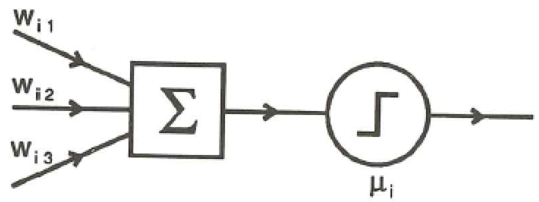
- This architecture with some hidden layers and fully forward connected layers is also called a multilayer perceptron.
- Fully connected: each node in layer i-1 is connected to each node in layer i.
- x_i is still a 1D vector of pixel values (e.g. 3072x1 for CIFAR-10)
- This network does not use any information about which pixel is a neighbor of which pixel, or any spatial features relating neighboring pixel values.
  - A Convolutional neural net will include this information and perform much better for image classification purposes.
- We can add as many hidden layers as we want, and the number of nodes in a hidden layer is a parameter we set.

Modelling one neuron

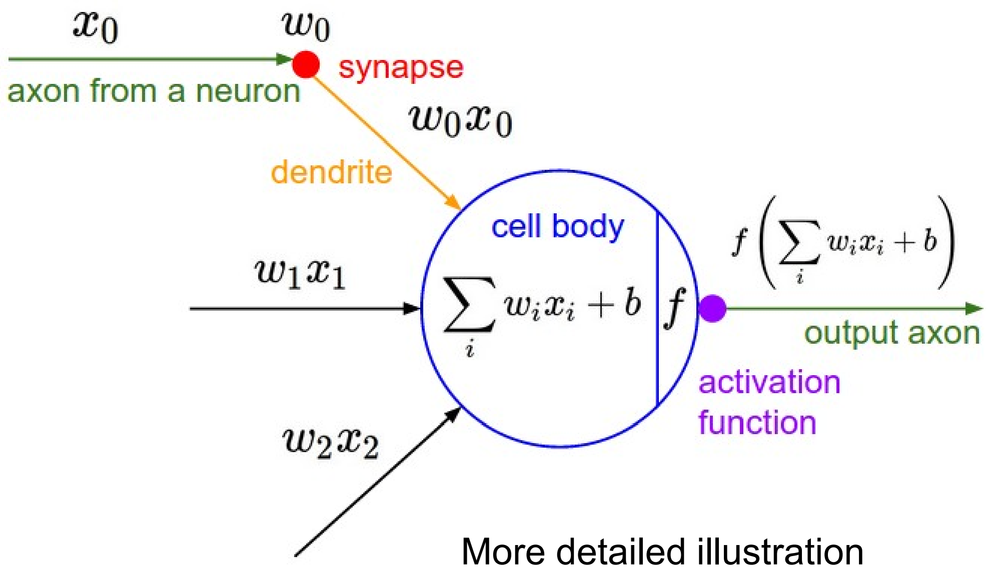
- One node in the network is inspired by a neuron in the brain.
- It received inputs from its dendrites and produce outputs along a single axon.
- We have about 86 billion neurons of different types.
- Neurons are connected by synaptic junctions or synapses.
- A cell will fire (send a pulse) if its potential reach a certain level. The frequency of firing carries information.
  - In mathematics this is modelled using activation function f.
- The weights that connect neurons are learnable.


synapse
axon
nucleus
cell body
dendrites

McColloch-Pitt's mathematical neuron model



Simple illustration



More detailed illustration

McColloch and Pitts (1943) modelled this as a binary threshold unit

Pseudo-code for forward progatating a single neuron

```
class Neuron(object):
  #
  def forward(inputs):
     #assume inputs and weights are 1D arrays
     cell_sum = np.sum(input*self.weights)+self.bias
     firing_rate = 1(1.0+math.exp(-cell_sum)) # Sigmoid activation
     return
```

Other activation functions can be better, like RELU

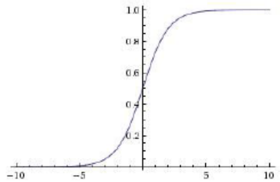
Note: not an accurate actual model for a human neuron, see http://www.sciencedirect.com/science/article/pii/S0959438814000130

Topic for a later lecture: which activation function to choose

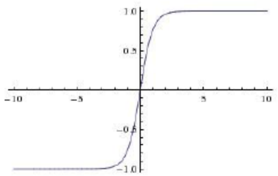
Activation Functions

Sigmoid

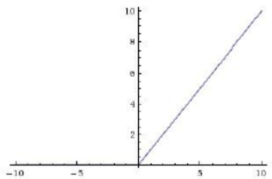
$\sigma(x) = 1/(1+e^{-x})$



tanh tanh(x)

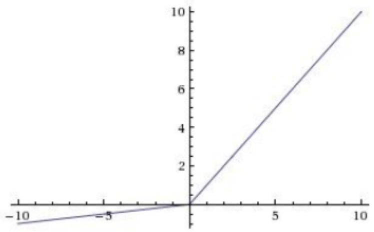


ReLU max(0,x)



Leaky ReLU

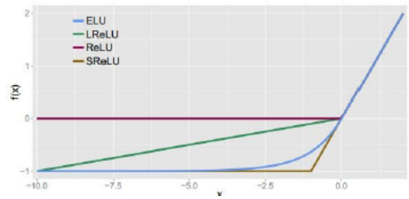
max(0.1x, x)



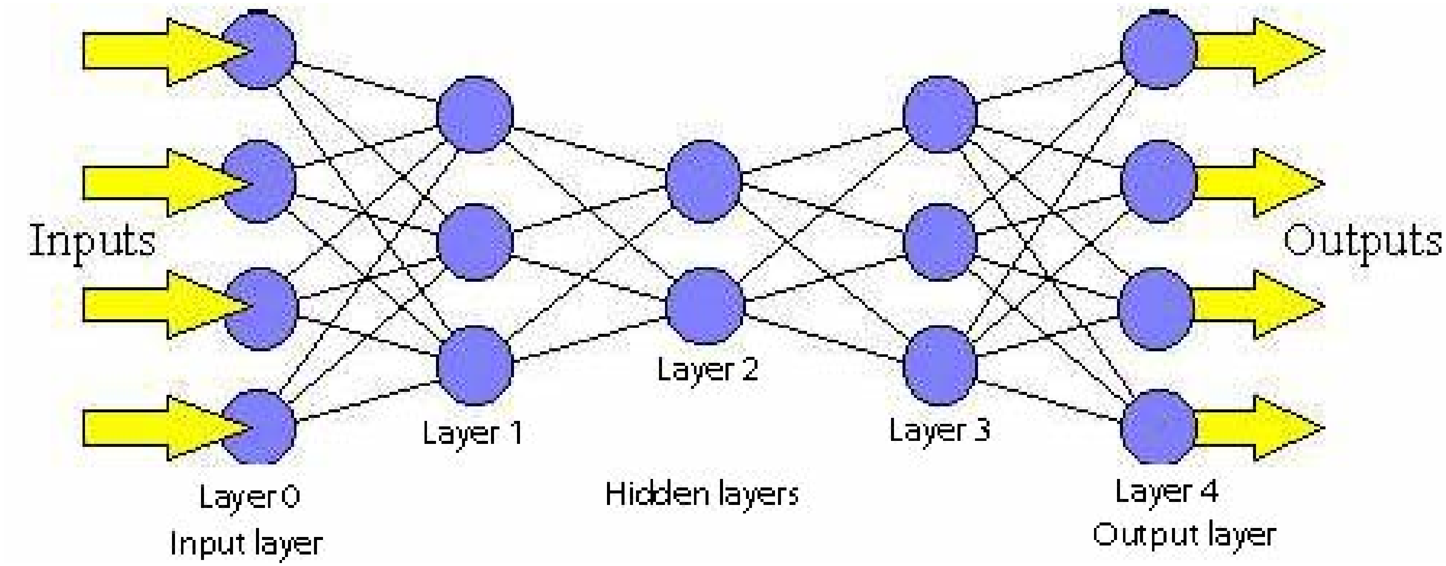
Maxout $\max(w_1^T x + b_1, w_2^T x + b$

ELU

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha\,(\exp(x) - 1) & \text{if } x \le 0 \end{cases}$$

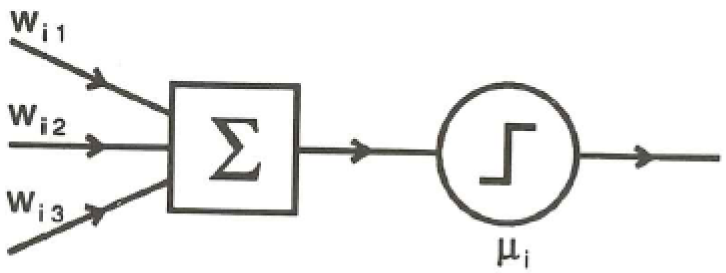


Architecture for a feed-forward net



4-layered net
- input layer not counted

One neuron as a binary logistic classifier

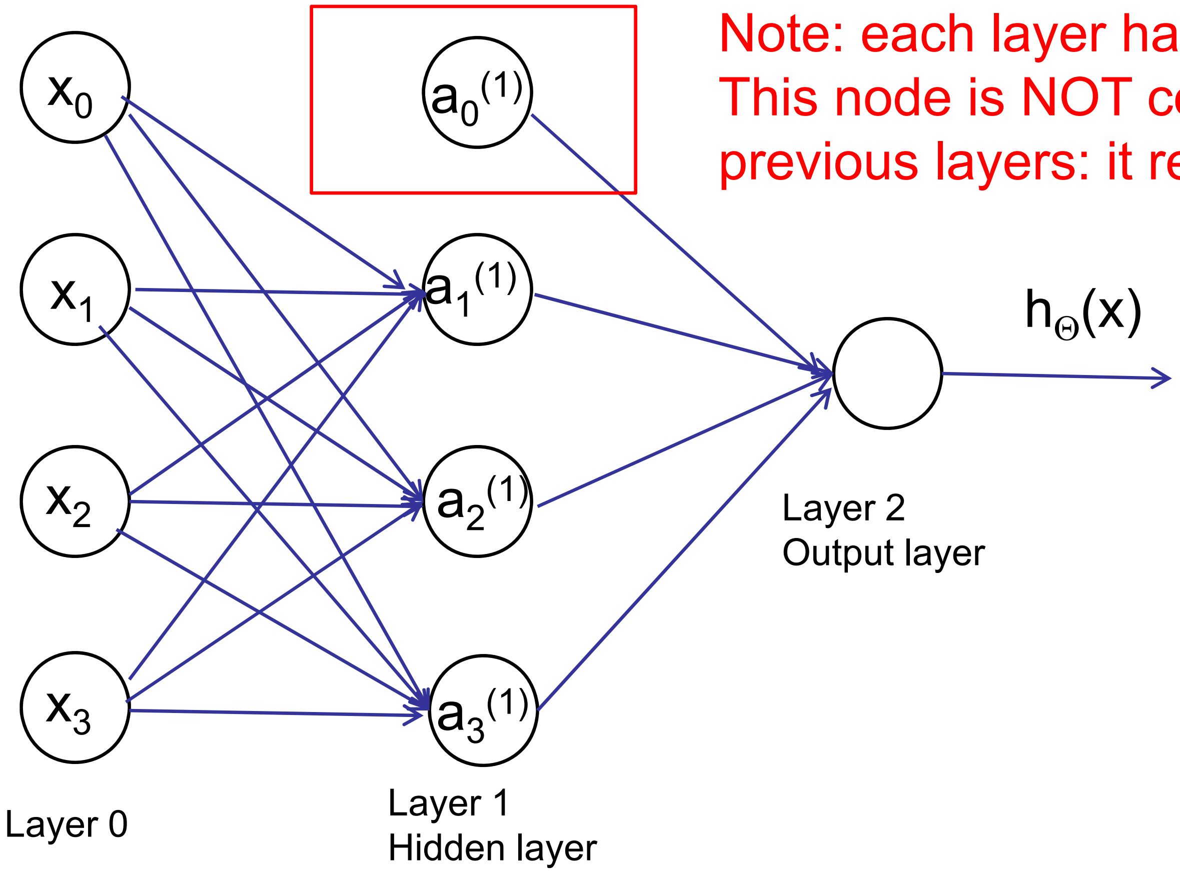


$\sigma(z) = 1/(1+e^{-x})$ is the sigmoid function

$\sigma(\sum_i w_i x_i + b)$ can be interpreted as a probability of class1 $\mathrm{P}(\mathrm{y_i} = 1 \mid \mathrm{x_i}, \mathrm{w})$, and

$\mathrm{P}(\mathrm{y_i} = 0 \mid \mathrm{x_i}, \mathrm{w}) = 1 - \mathrm{P}(\mathrm{y_i} = 1 \mid \mathrm{x_i}, \mathrm{w})$

2-layer net , 3 pixels, with notation



Note: each layer has a bias node
This node is NOT connected to previous layers: it receives NO INPUT

Notation

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

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

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

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

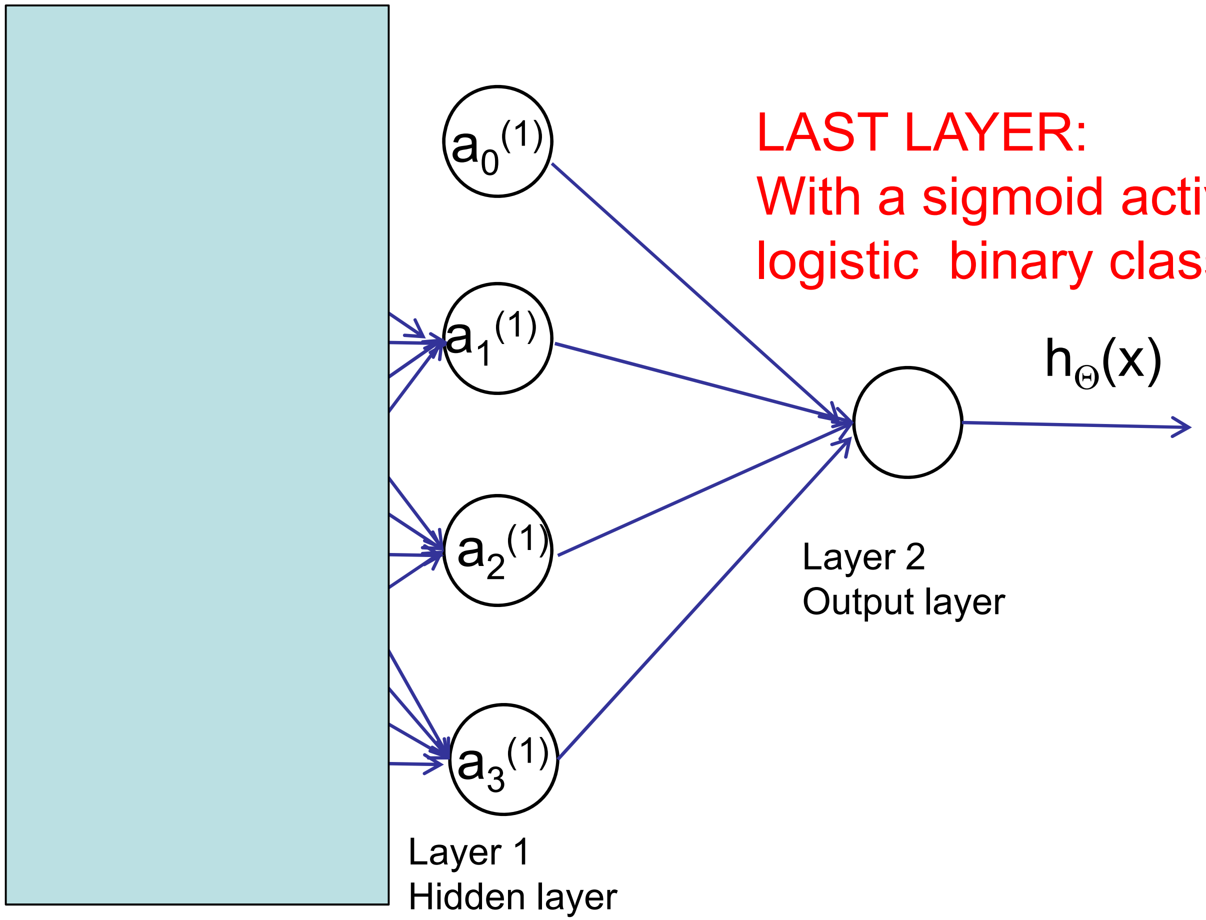
$$\mathrm{a}_1^{(1)} = g\left(\Theta_{10}^{(1)}\mathrm{x}_0 + \Theta_{11}^{(1)}\mathrm{x}_1 + \Theta_{12}^{(1)}\mathrm{x}_2 + \Theta_{13}^{(1)}\mathrm{x}_3\right)$$

$$\mathrm{a}_2^{(1)} = g\left(\Theta_{20}^{(1)}\mathrm{x}_0 + \Theta_{21}^{(1)}\mathrm{x}_1 + \Theta_{22}^{(1)}\mathrm{x}_2 + \Theta_{23}^{(1)}\mathrm{x}_3\right)$$

$$\mathrm{a}_3^{(1)} = g\left(\Theta_{30}^{(1)}\mathrm{x}_0 + \Theta_{31}^{(1)}\mathrm{x}_1 + \Theta_{32}^{(1)}\mathrm{x}_2 + \Theta_{33}^{(1)}\mathrm{x}_3\right)$$

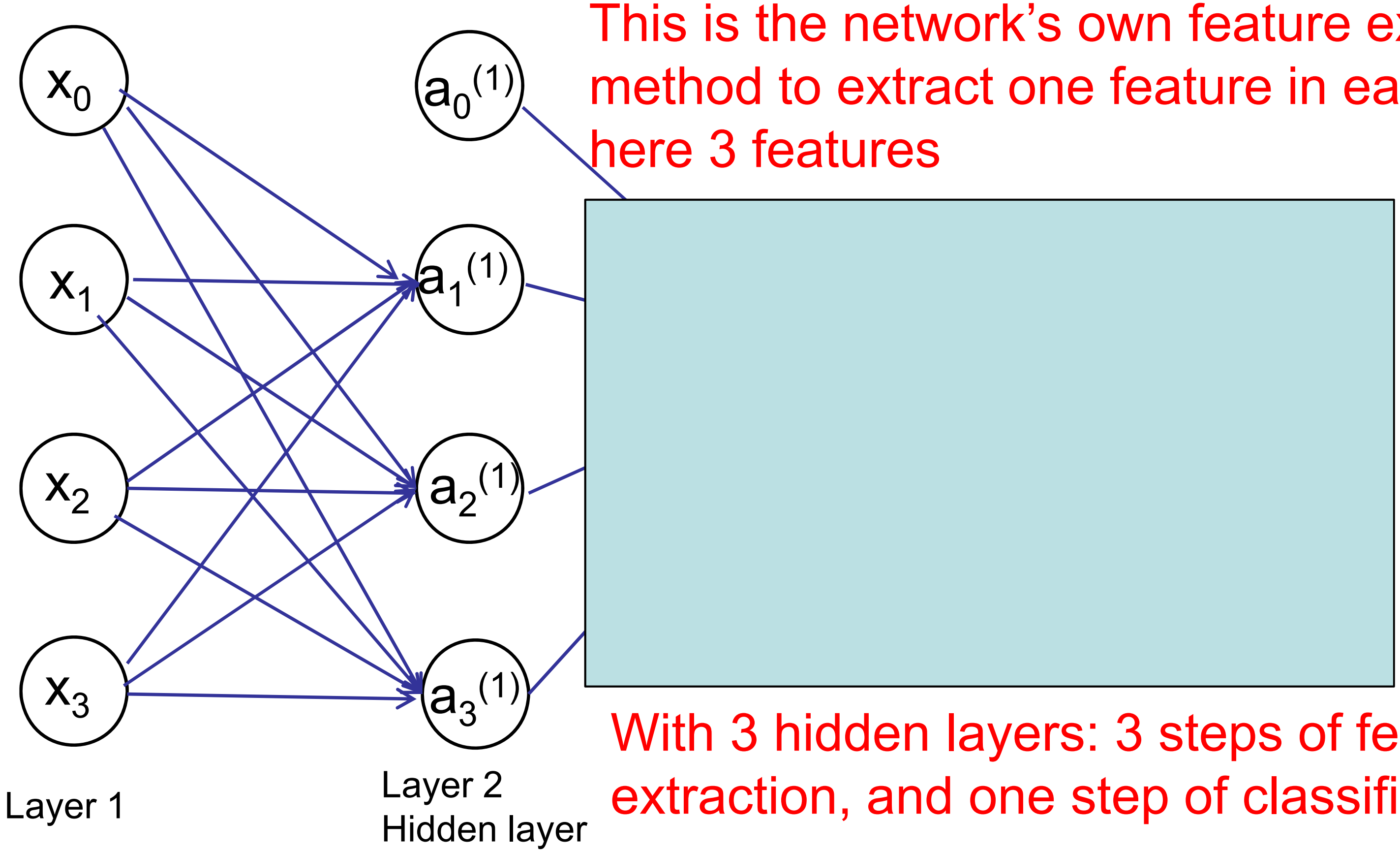
$$h_\Theta(x) = \mathrm{a}_1^{(2)} = g\left(\Theta_{10}^{(2)}\mathrm{a}_0{}^{(2)} + \Theta_{11}^{(2)}\mathrm{a}_1{}^{(2)} + \Theta_{12}^{(2)}\mathrm{a}_2{}^{(2)} + \Theta_{13}^{(2)}\mathrm{a}_3{}^{(2)}\right)$$

2-layer net – what do the layers do?



LAST LAYER:
With a sigmoid activation, h acts as a logistic binary classifier on $a_0^{(1)} \ldots a_3^{(1)}$

HIDDEN LAYER:
Each $a_1^{(1)}$ is a weighted linear combination of all input pixels $x_1, \ldots x_3$
This is the network's own feature extraction method to extract one feature in each node, here 3 features

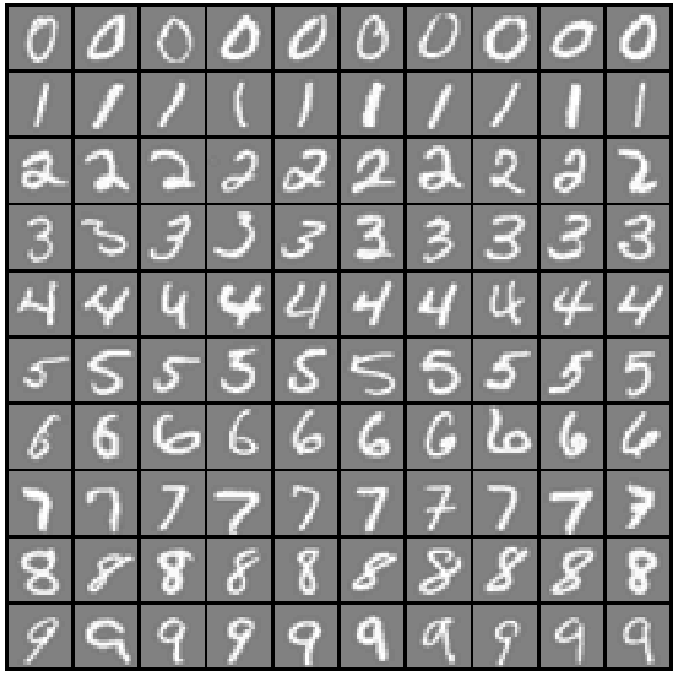


With 3 hidden layers: 3 steps of feature extraction, and one step of classification

Example net: MNIST-classification

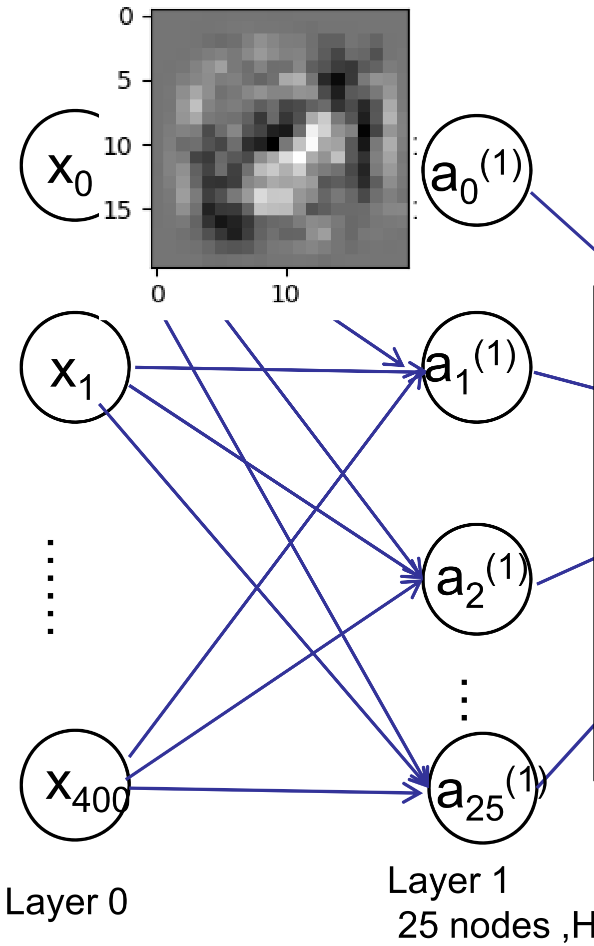
- Input 20x20 images=400 input nodes + 1 bias
- 25 nodes in hidden layer: $\Theta^{(1)}$ is 25x401 (add bias term x_0)
- 10 classes: digits ‘0’-’9’: $\Theta^{(2)}$ is 10x26 (add bias term $a_0^{(2)}$
  - One vs. all loss function

Example of trained network MNIST data



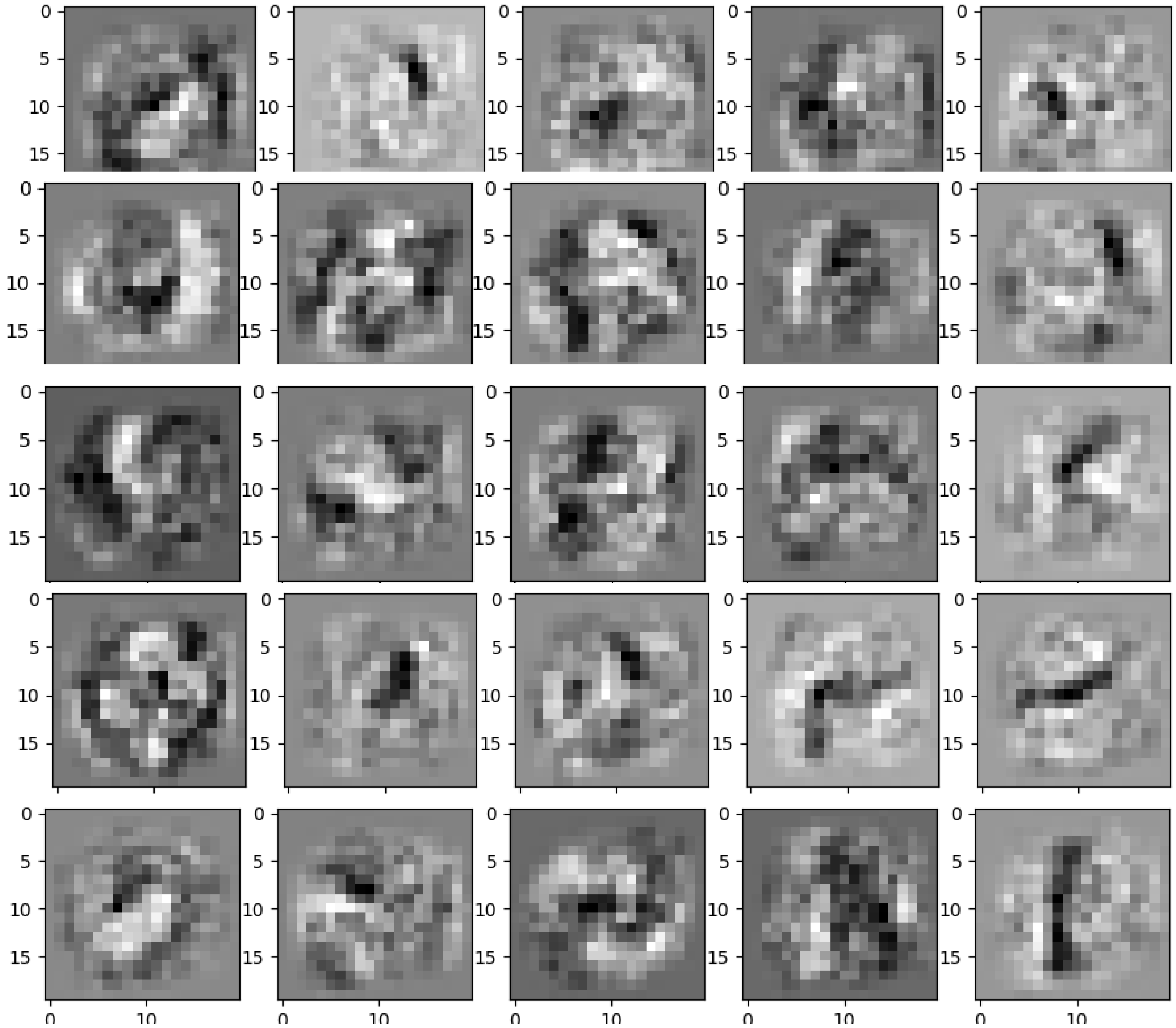
20x20 images
of handwritten digits,
10 classes

HIDDEN LAYER:
Nof. nodes is a parameter we need to select.
25 nodes represents 25 features.
Visualize these features by the weights to node $a_1^{(1)}$-$a_{25}^{(1)}$ for each of the 20x20 pixels.
(Ignore bias here)



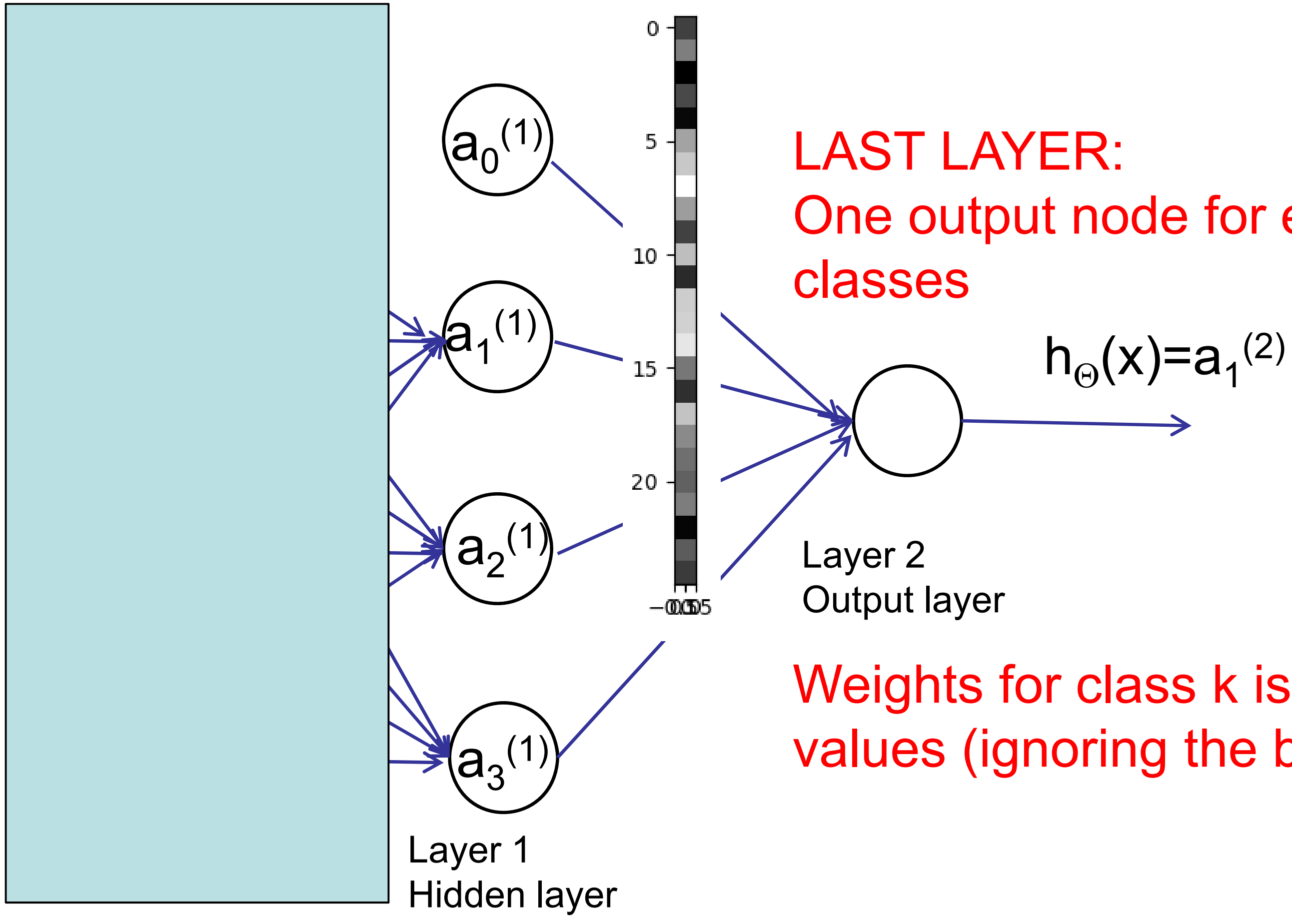
With 3 hidden layers: 3 steps of feature extraction, and one step of classification

Ui



Weights $\Theta^{(1)}$ to node 1-25

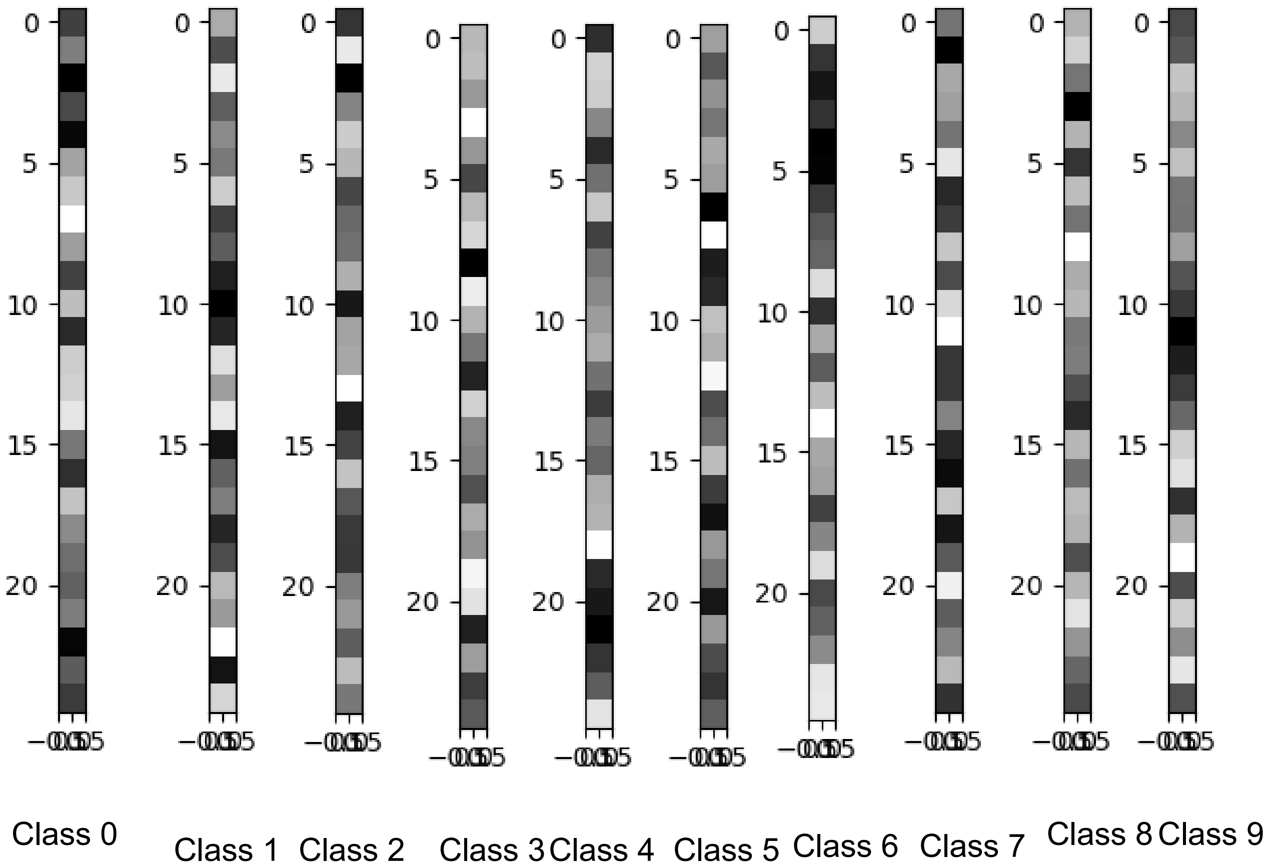
Weights for the output layer 25 hidden nodes, 10 classes



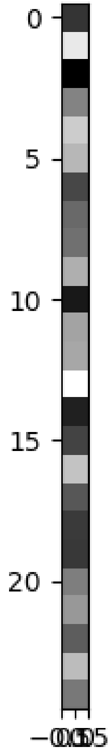
LAST LAYER:
One output node for each of the 10 classes

Weights for class k is a vector of 25 values (ignoring the bias term)

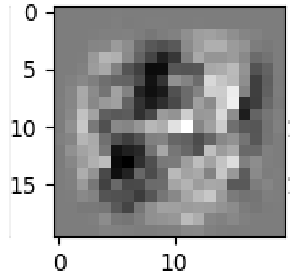
Weights for the output classes

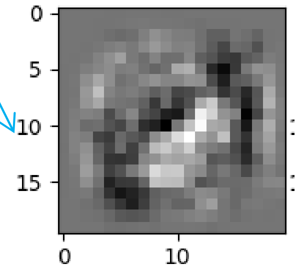


Let us look at class 2



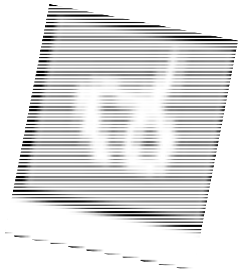
High value for feature/node 13 and 1

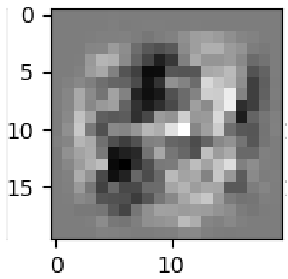




Maybe this responds to objects with a center in the right half (13) and seeing a diagonal edge????

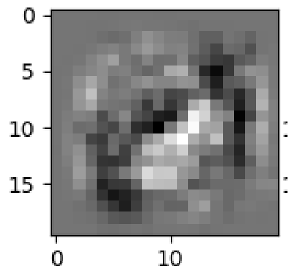
What if the object is rotated or translated?





Maybe this responds to objects with a center in the right half (13) and seeing a diagonal edge????

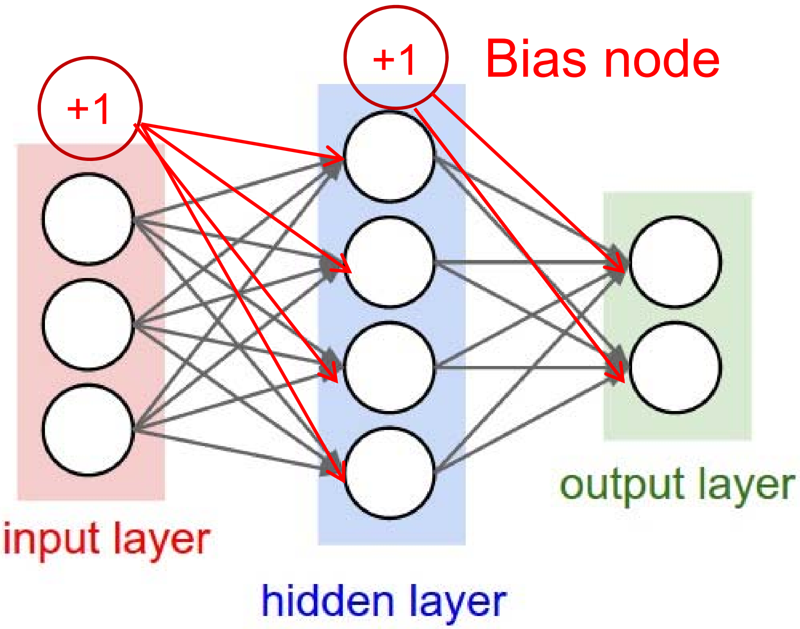
Would this be the case for the rotated object?



Some remarks

- With the output layer, it is possible to use a net with or without the activation function.
- The size of network is measured by the number of neurons.

Example feed-forward computation



- Input x: 3x1 vector

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

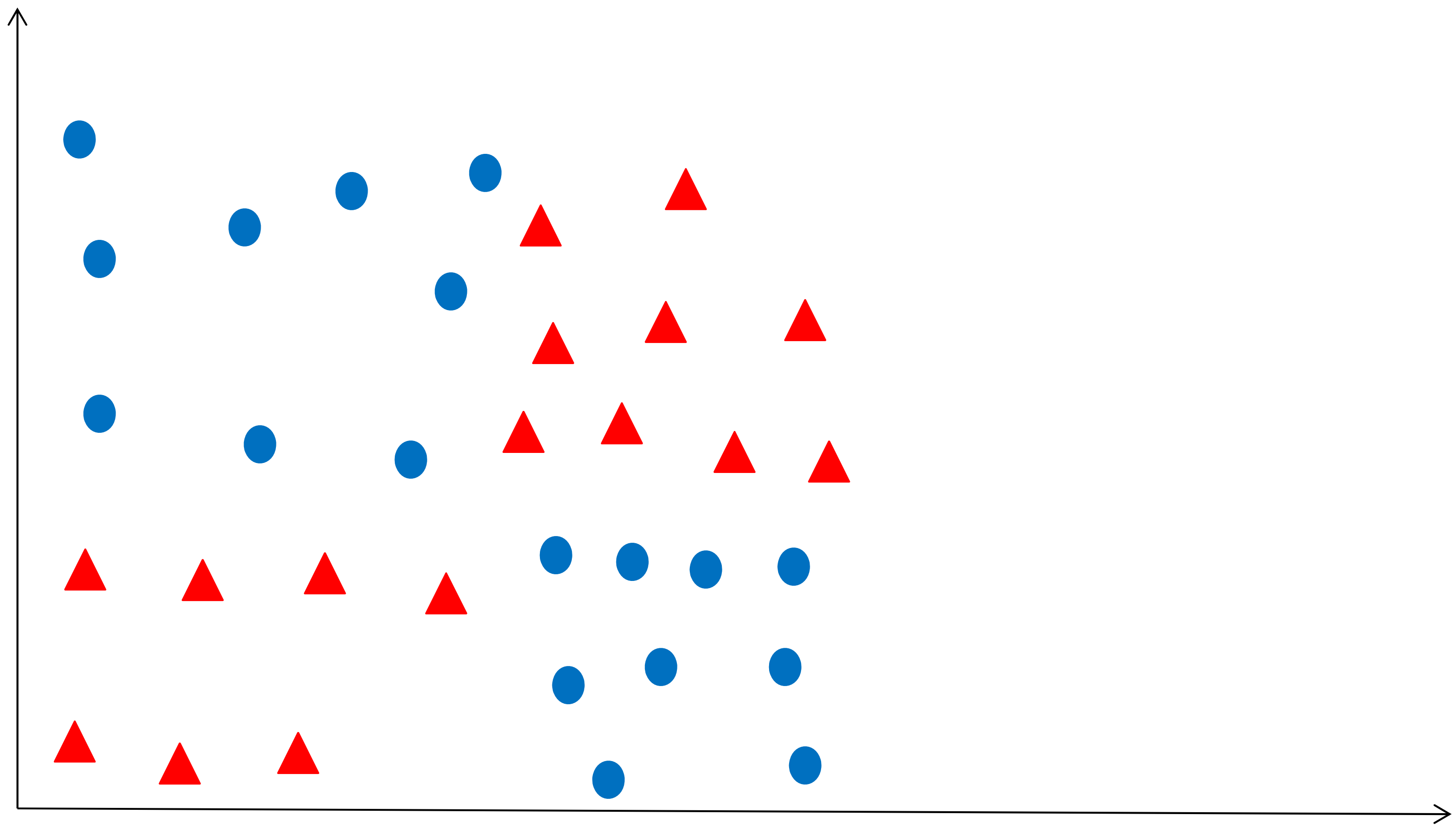
$\Theta^{(2)}: 2\times 5$ (nof. classes $\times$ nof. hidden nodes in layer 1 + 1)

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

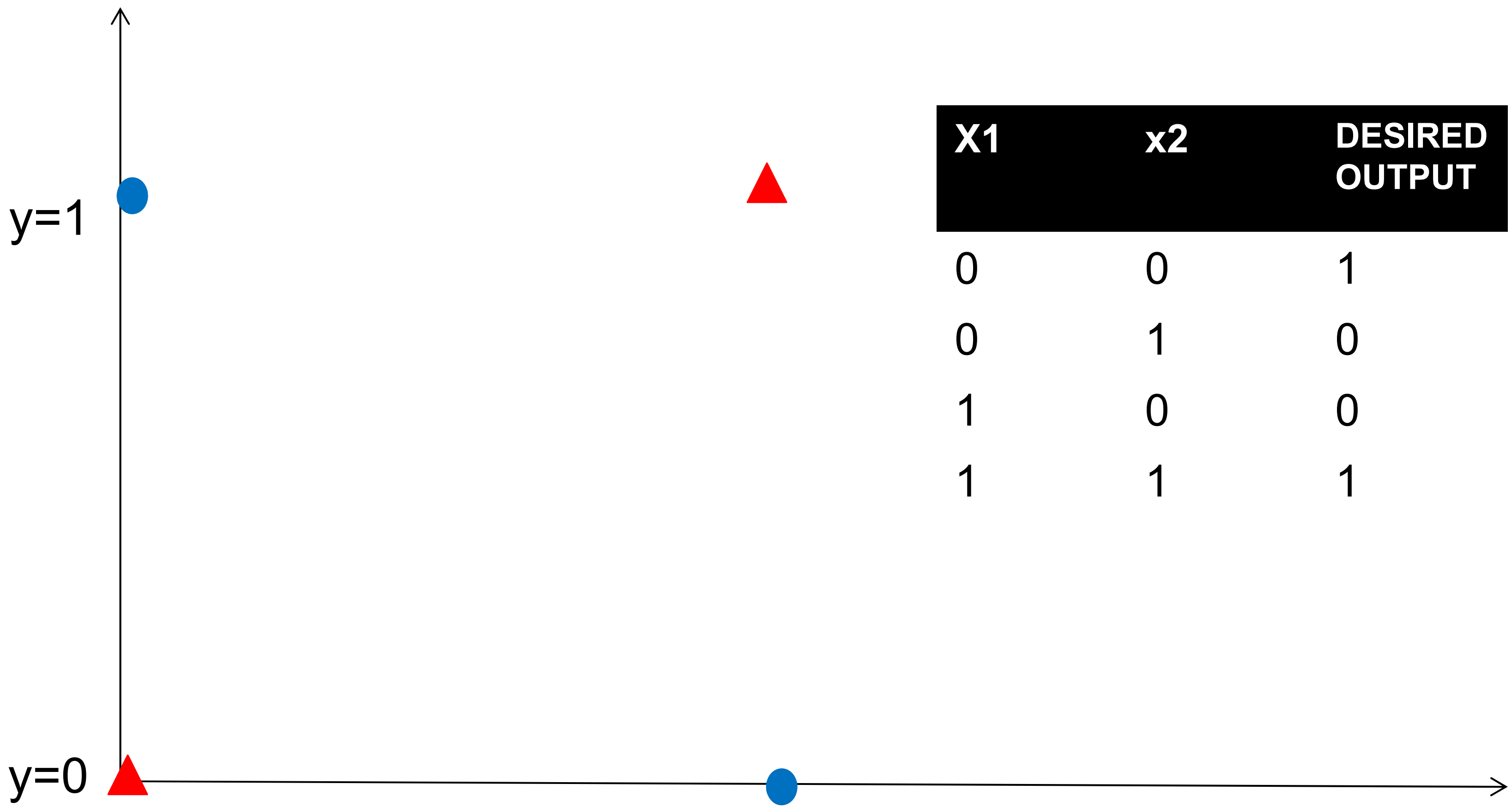
$$X=\begin{bmatrix} 1 & x_{pixel1}(n=1) & x_{pixel2}(1) & x_{pixel3}(1) \\ 1 & x_{pixel1}(n=2) & x_{pixel2}(2) & x_{pixel3}(2) \\ \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots \\ 1 & x_{pixel1}(n=N) & x_{pixel2}(N) & x_{pixel3}(N) \end{bmatrix}$$

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

Back to the non-linearly separable case

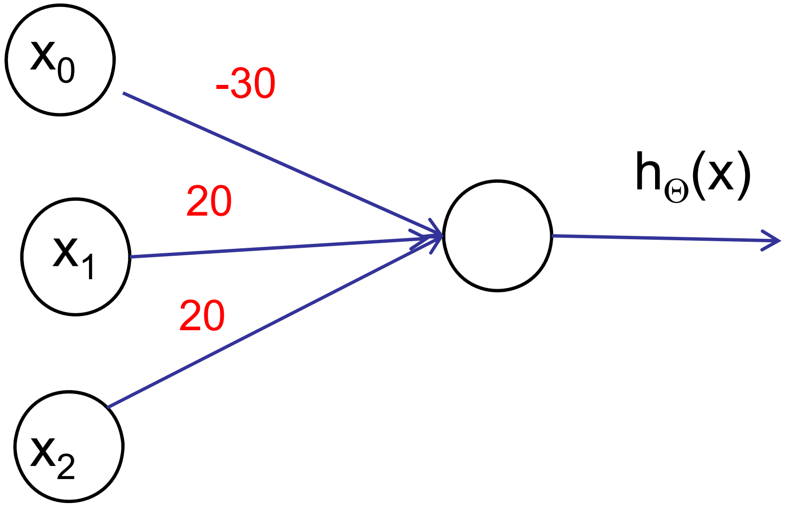


A similiar, simple example: XNOR



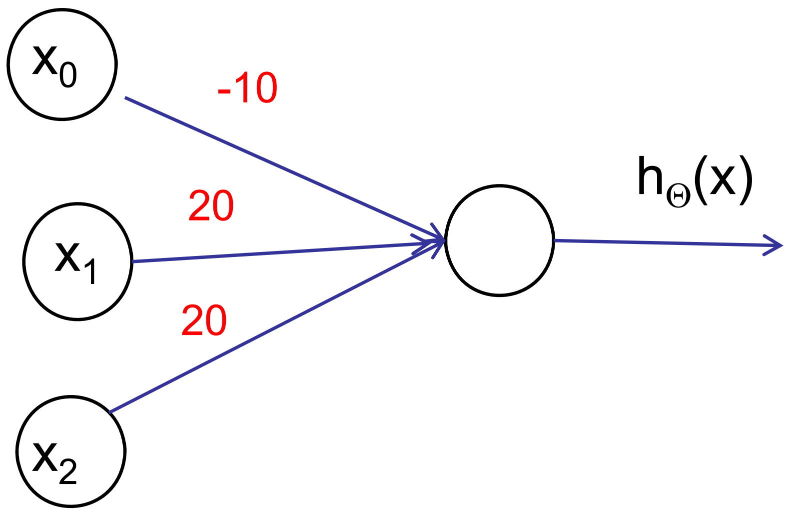
X1	x2	DESIRED OUTPUT
0	0	1
0	1	0
1	0	0
1	1	1

Coding the AND-functions



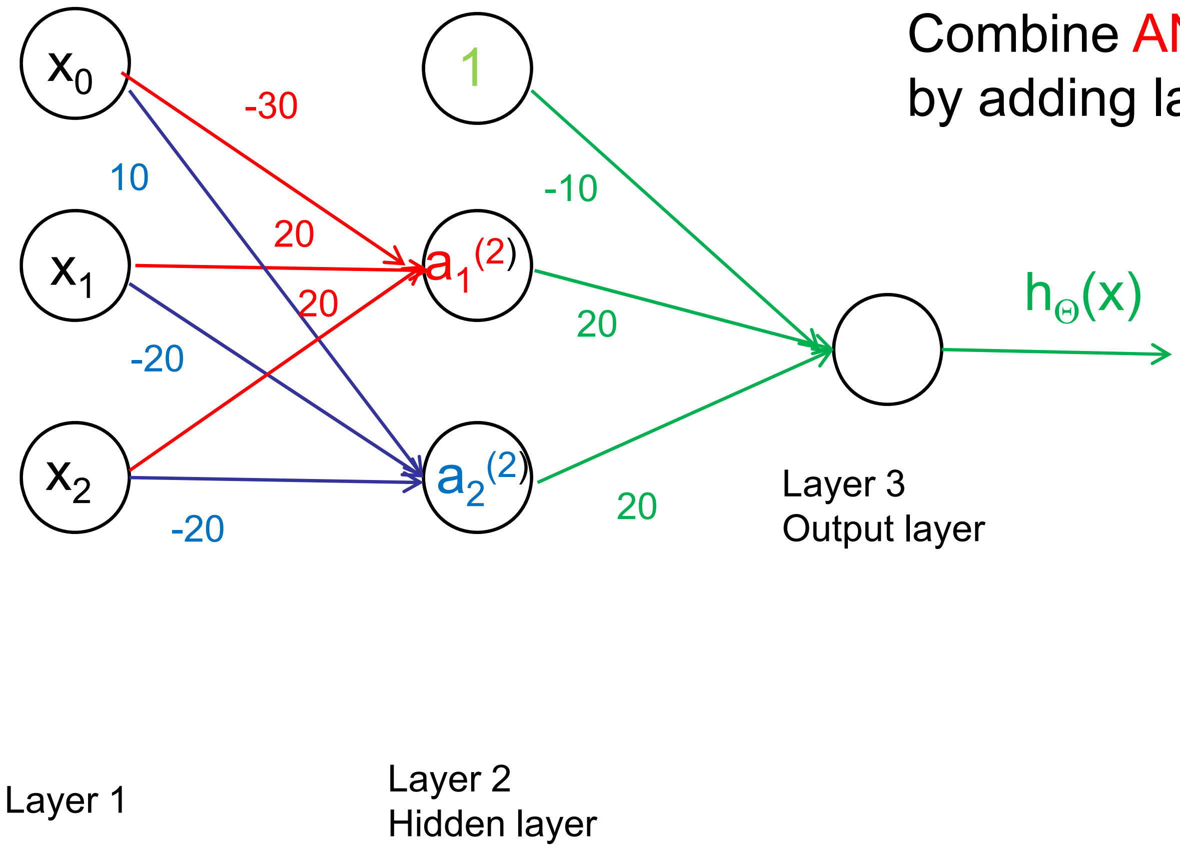
X1	x2	hΘ(x)
0	0	g(-30)≈0
0	1	g(-10)≈0
1	0	g(-10)≈0
1	1	g(10)≈1

Which logical function is this?



X1	x2	hΘ(x)=?
0	0	
0	1	
1	0	
1	1	

Creating the XNOR-function



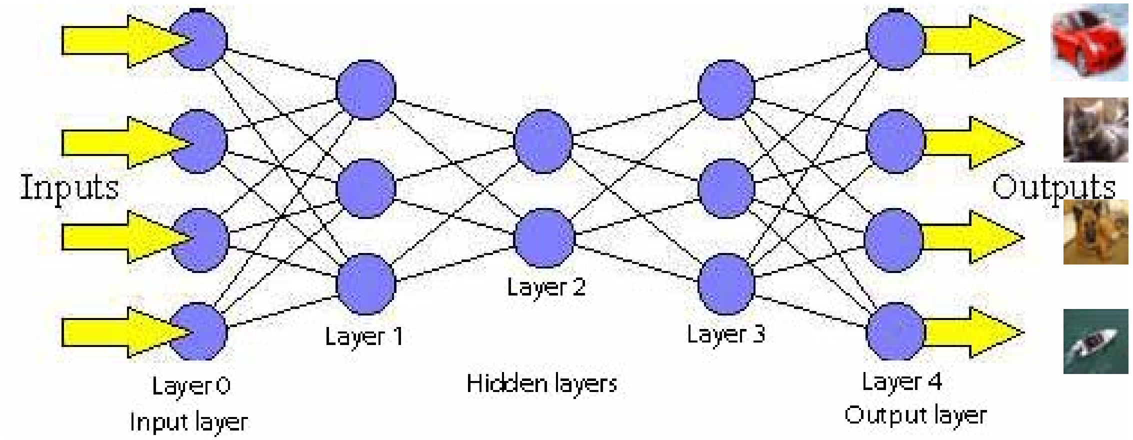
Combine AND and OR
by adding layer to get XNOR

Representation power of the net

- Fully-connected nets define a family of functions that are parameterized by the weights of the network.
- It turns out that nets with at least one hidden layer are universal approximators
  – Given any continuous function f(x) and some e>0, there is a net g(x) (with a non-linear activation) that can represent the function such that
  – |f(x)-g(x)|<e
- But why do we need more than one hidden layer?
- `In practise, 3-layers feed-forward nets often works better then 2-layers, but going deeper rarely helps.
  – This is NOT the case for Convolutional nets where depth helps, more on this later.

Multiple classes: One-vs-all

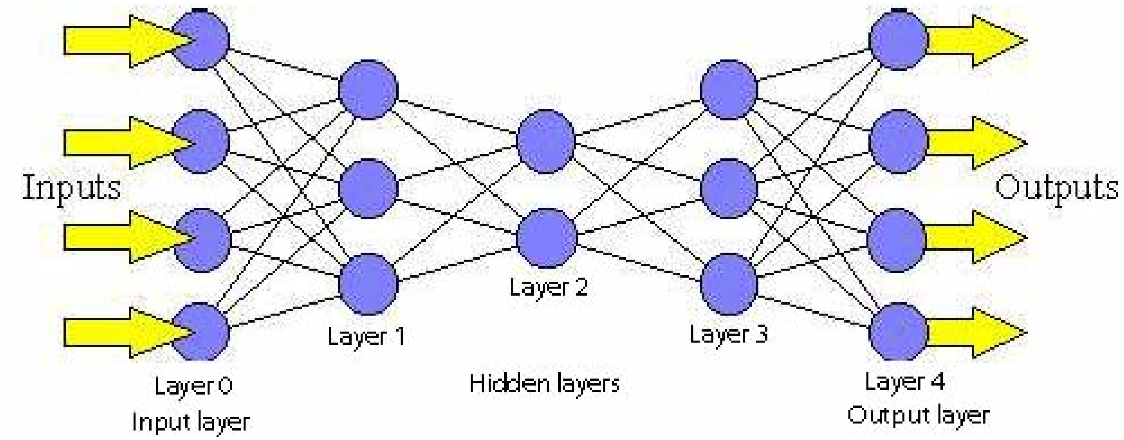
- Train one output node for each class, e.g. CAR (yes/no), CAT(yes/n0)



$$\text{Want}: \ \mathrm{h}_\Theta(x_i) \approx y_i = \begin{bmatrix}1\\0\\0\\0\end{bmatrix} \quad \mathrm{h}_\Theta(x_i) \approx y_i = \begin{bmatrix}0\\1\\0\\0\end{bmatrix} \quad \mathrm{h}_\Theta(x_i) \approx y_i = \begin{bmatrix}0\\0\\1\\0\end{bmatrix} \quad \mathrm{h}_\Theta(x_i) \approx y_i = \begin{bmatrix}0\\0\\0\\1\end{bmatrix}$$

if CAR if CAT if DOG if SHIP

Neural network classification



Binary classification:

y = 1 or 0

1 output unit

Multi-class (K classes)

$$y \in R_k, \text{e.g.} \begin{bmatrix}1\\0\\0\\0\end{bmatrix}, \begin{bmatrix}0\\1\\0\\0\end{bmatrix}, \begin{bmatrix}0\\0\\1\\0\end{bmatrix}, \begin{bmatrix}0\\0\\0\\1\end{bmatrix}$$

Cost function for neural networks

- For logistic regression it was:

$$J(\theta) = -\frac{1}{m}\left[\sum_{i=1}^{m} y(i)\log h_\theta(X(i,:) + (1-y(i))\log(1-h_\theta(X(i,:))\right] + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2$$

- For neural nets it is:

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

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

L : number of layers

s_1 : Number of units (without bias) in layer 1

Implementing the cost function

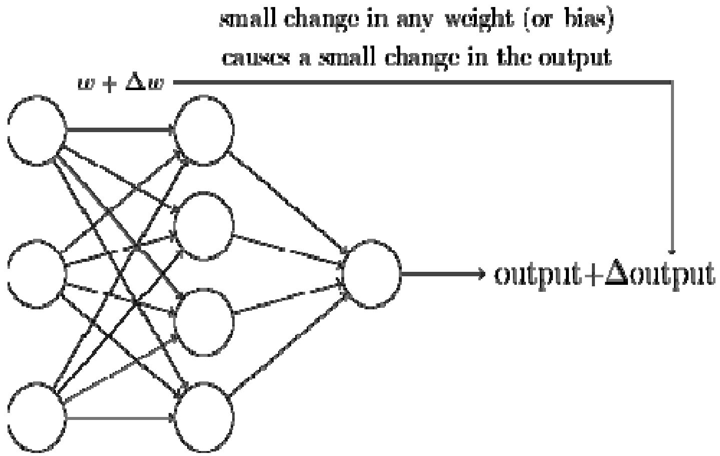
- Create an indicator matrix Y with one row per sample, where each row encodes the class as:

$$\text{IF y} = \begin{bmatrix} 2 \\ 2 \\ 1 \\ 4 \\ \vdots \\ 3 \end{bmatrix}, \quad \text{let } Y = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

- Compute the cost without regularization, then add the regularization term.
  – Use a loop over the training samples if you want.

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.



Neural net optimization problem

- Given a cost function L (or J), a set of training data (x_i, y_i), and the weights W.
- Normally we use backpropagation to compute the gradient of the cost function with respect to W
  - We can also compute it with respect to input x_i (useful for visualization)

Gradients and partial derivatives

$$f(x,y) = xy \rightarrow \frac{\partial f}{\partial x} = y \ \frac{\partial f}{\partial y} = x$$

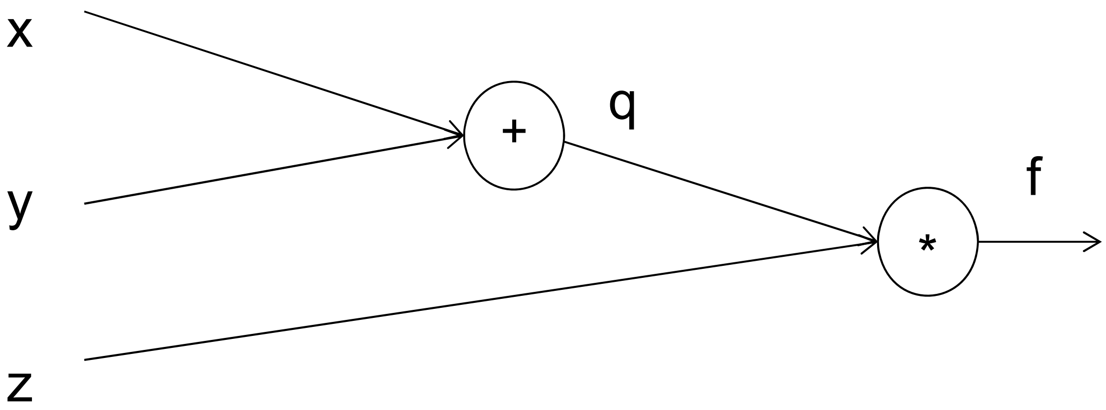
$$f(x,y) = x + y \rightarrow \frac{\partial f}{\partial x} = 1 \ \frac{\partial f}{\partial y} = 1$$

$$f(x,y) = \max(x,y) \rightarrow \frac{\partial f}{\partial x} = 1(x \geq y) \ \frac{\partial f}{\partial y} = 1(y \geq x)$$

$f(x,y,z) = (x+y)z$ Let $\mathrm{q} = \mathrm{x} + \mathrm{y}$ and $f = qz$ and use the chain rule :

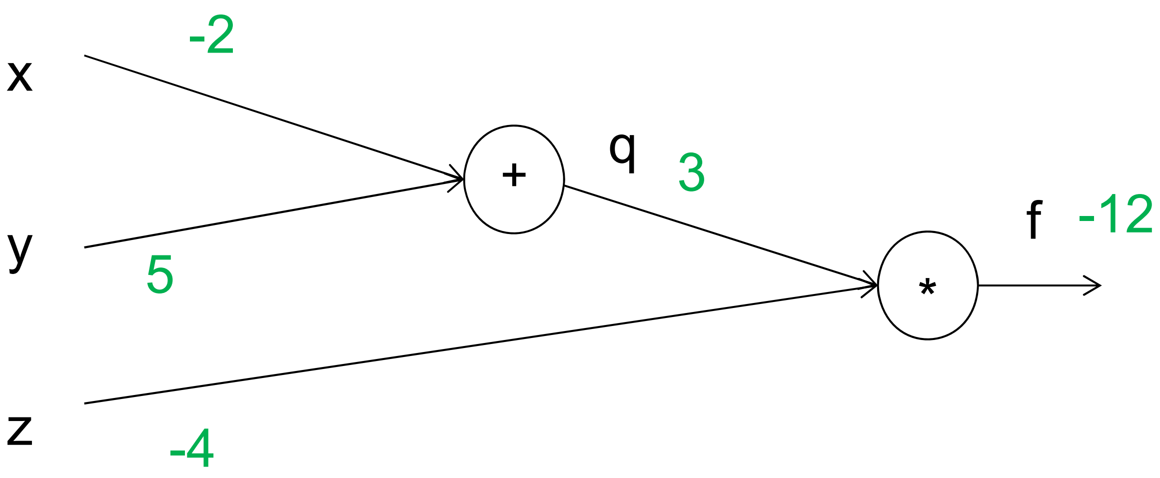
$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial x}$$

Computational graph for f=(x+y)z



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

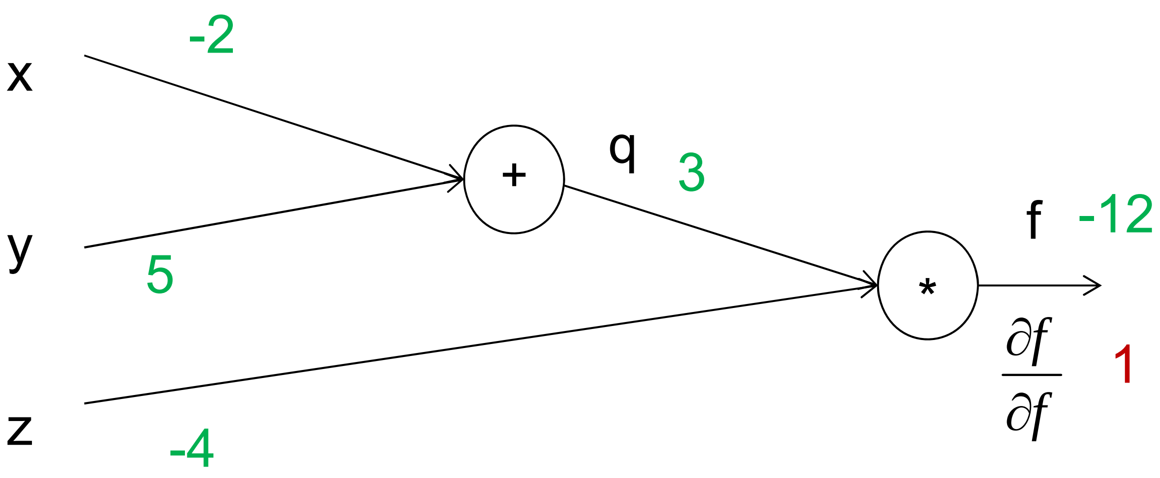
Forward propagation of one sample



One sample, x=-2, y=5, z=-4

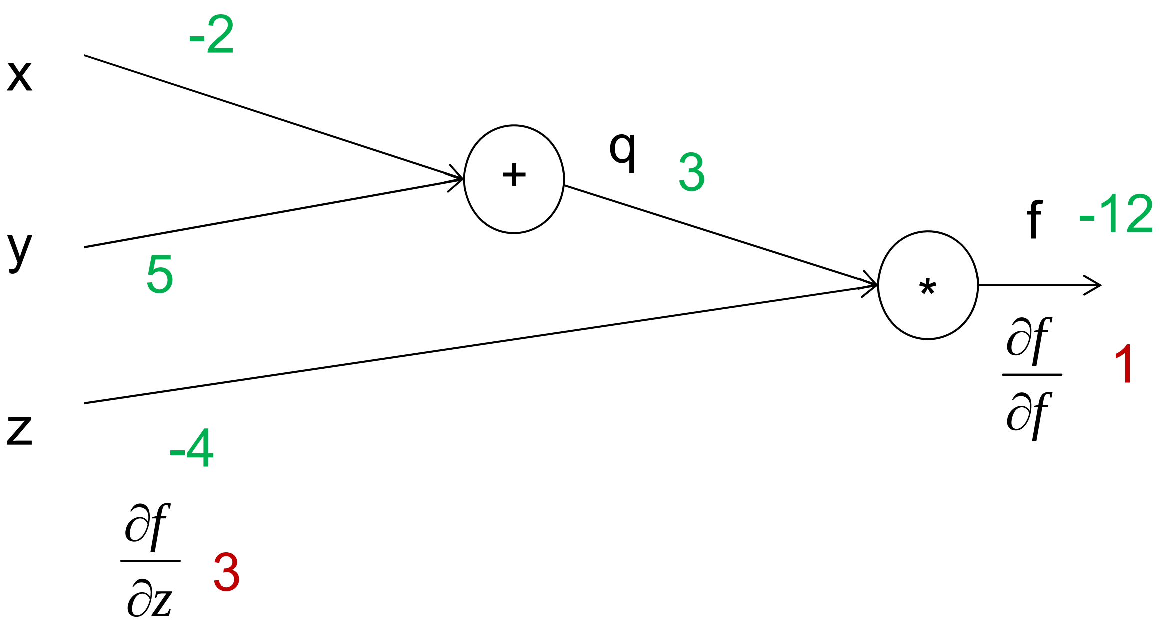
Green numbers: forward propagation
Red numbers: backwards propagation

Backwards propagation of gradients



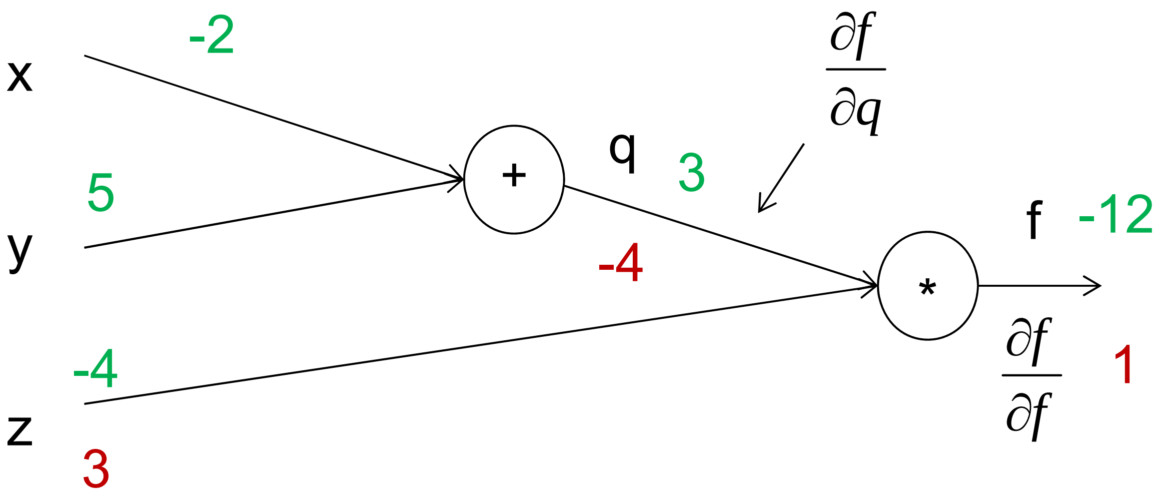
Green numbers: forward propagation
Red numbers: backwards propagation

Backwards propagation of gradients



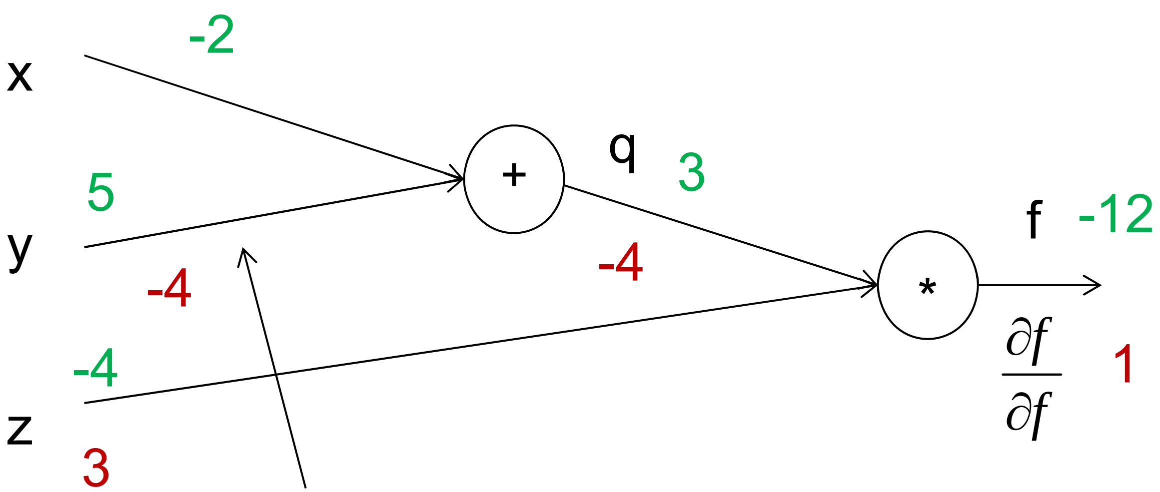
Green numbers: forward propagation
Red numbers: backwards propagation

Backwards propagation of gradients



Green numbers: forward propagation
Red numbers: backwards propagation

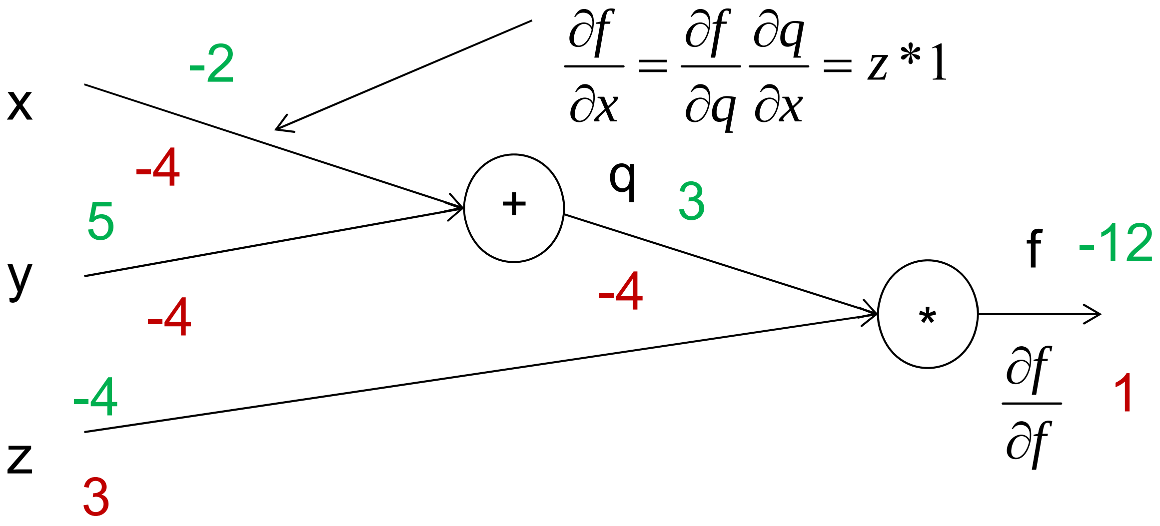
Backwards propagation of gradients



$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial y} = z * 1$$

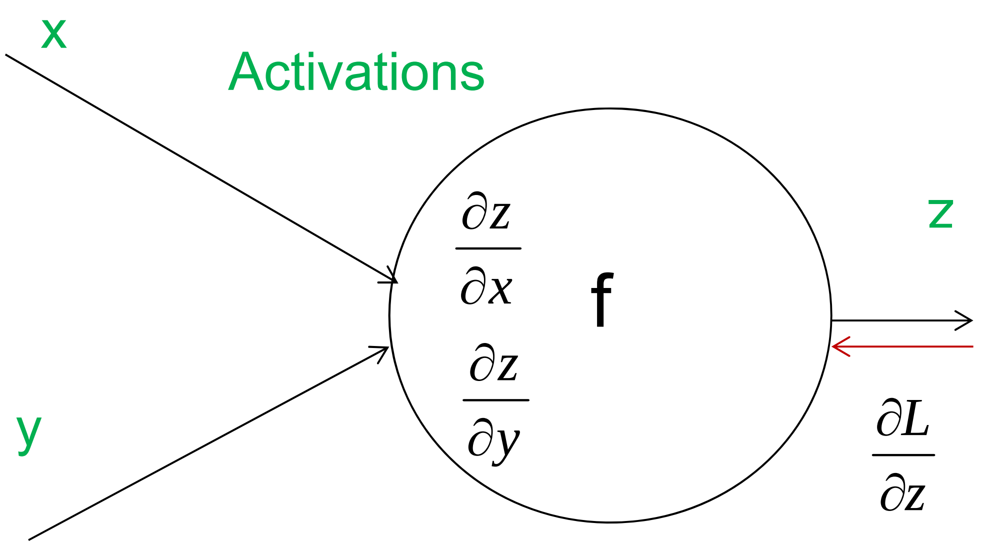
Green numbers: forward propagation
Red numbers: backwards propagation

Backwards propagation of gradients



Green numbers: forward propagation
Red numbers: backwards propagation

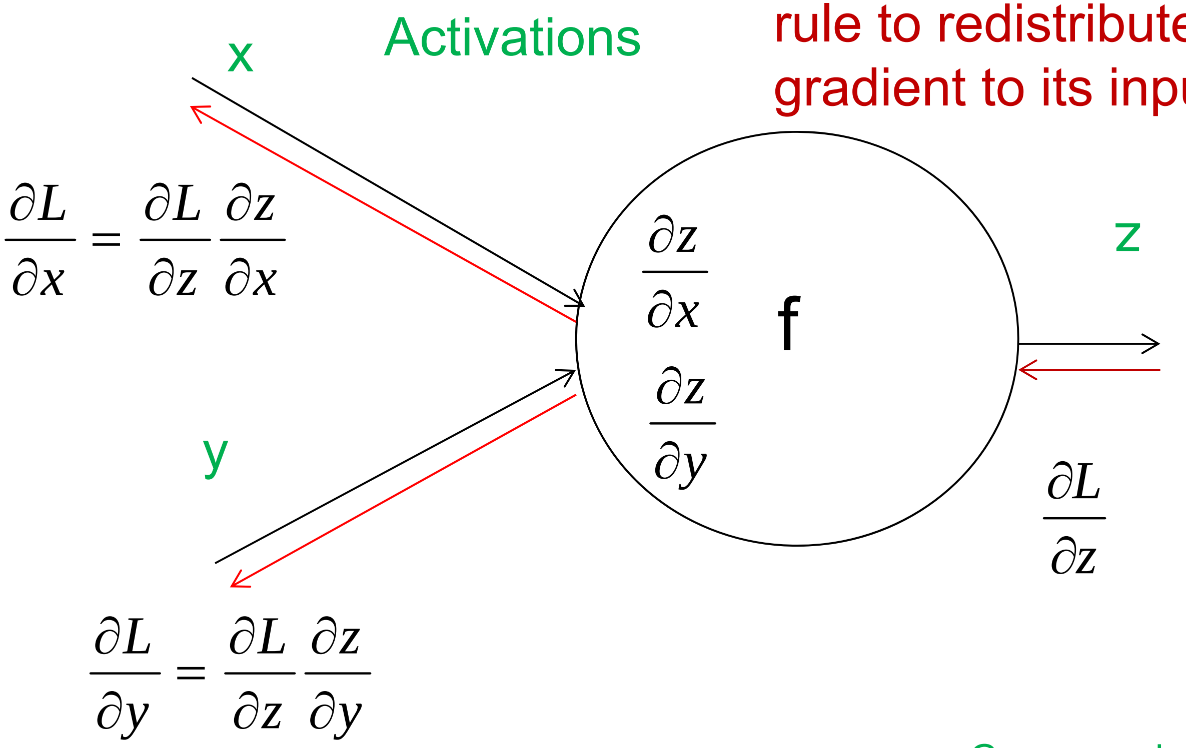
Each gate: get input x and y
Can compute output x
AND the local gradients of z



Green numbers: forward propagation
Red numbers: backwards propagation

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

The sigmoid function

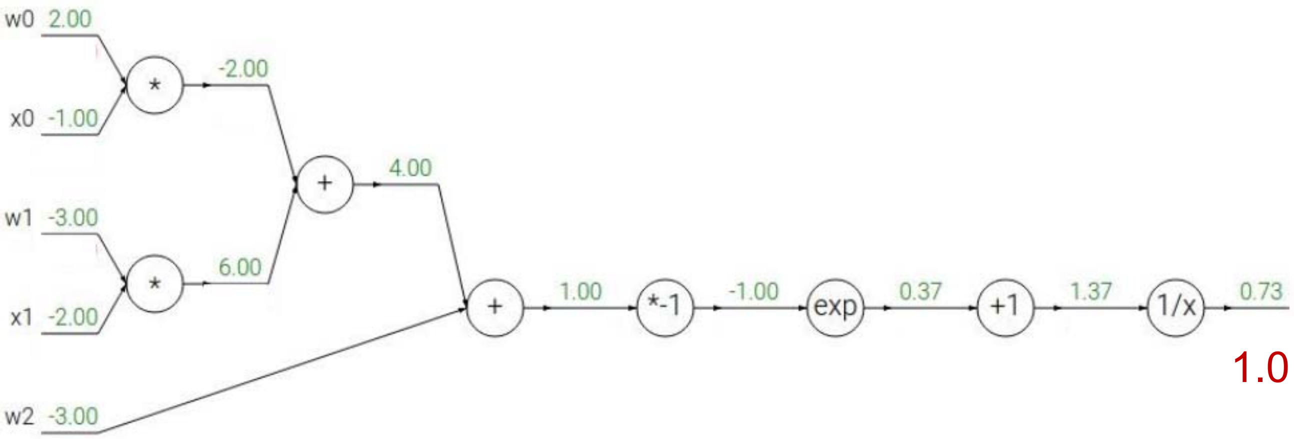
$$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$

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

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

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

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

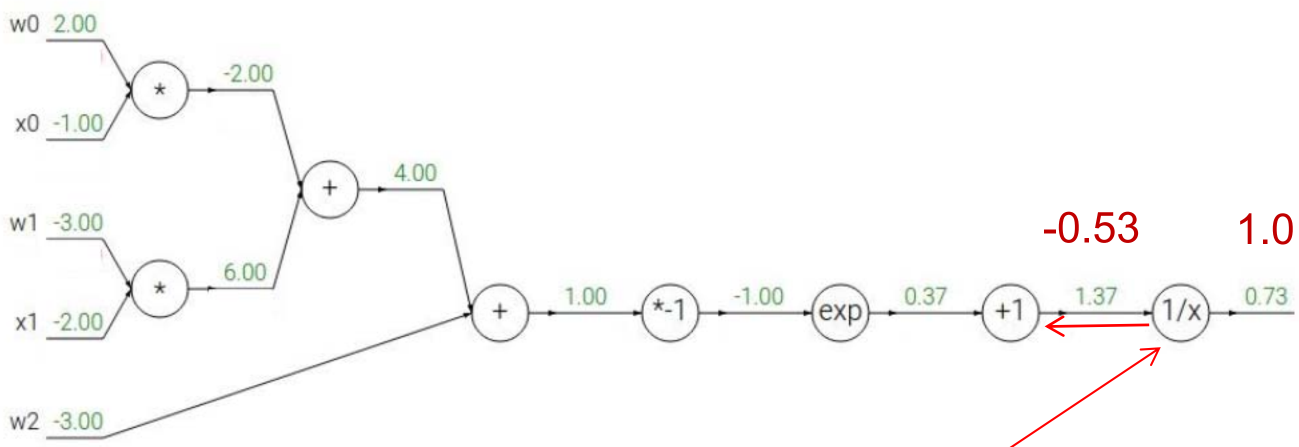


$$f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -1/x^2$$

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

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

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



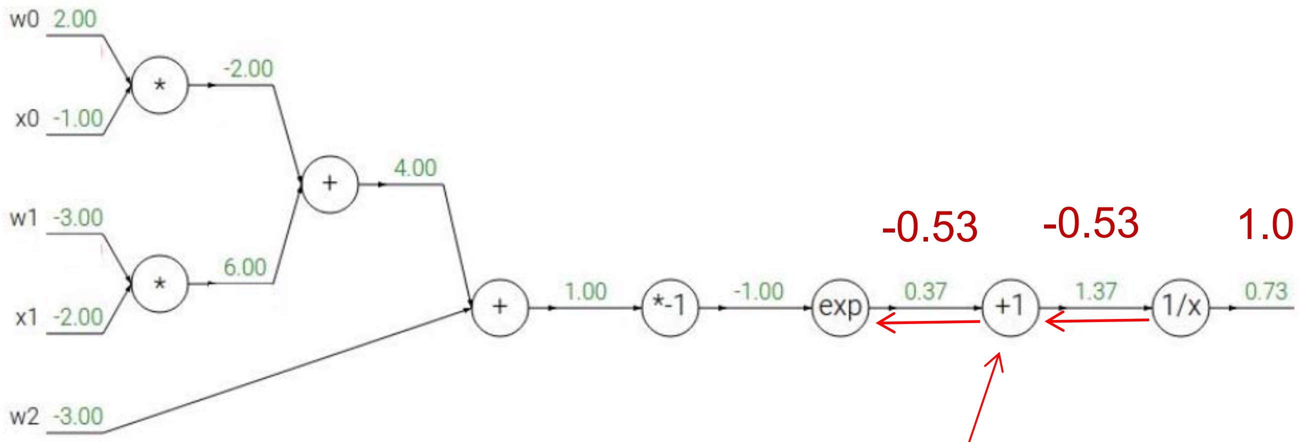
$$f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -1/x^2$$

-1/(1.37)^2*1.0

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

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

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



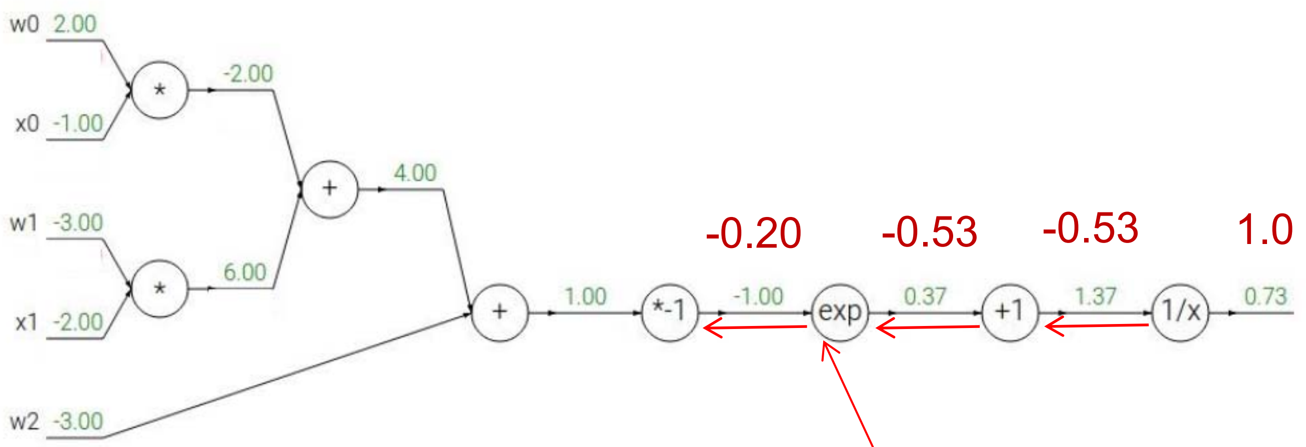
$$f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -1/x^2$$

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

1*-0.53=-0.53

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

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



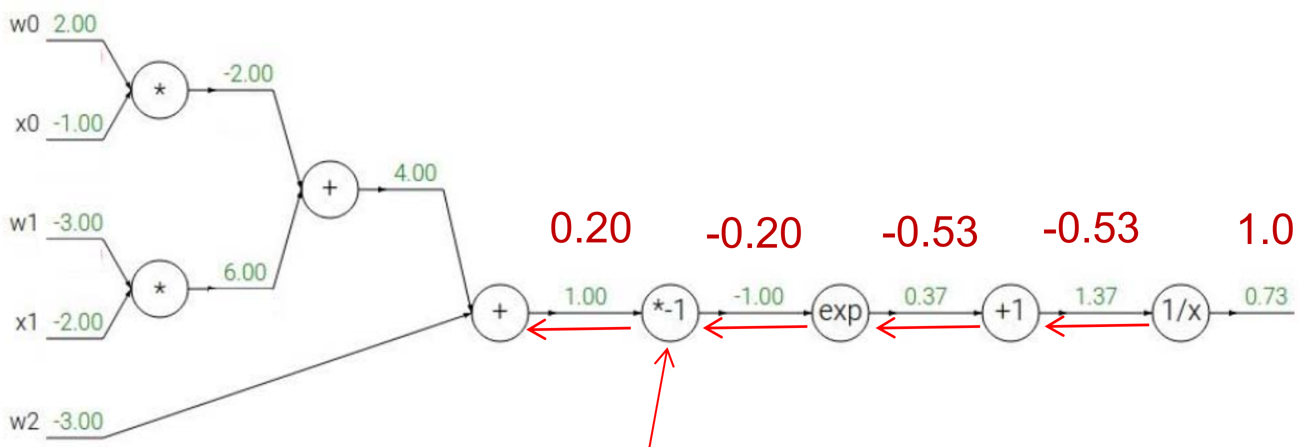
$$f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -1/x^2$$

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

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

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

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



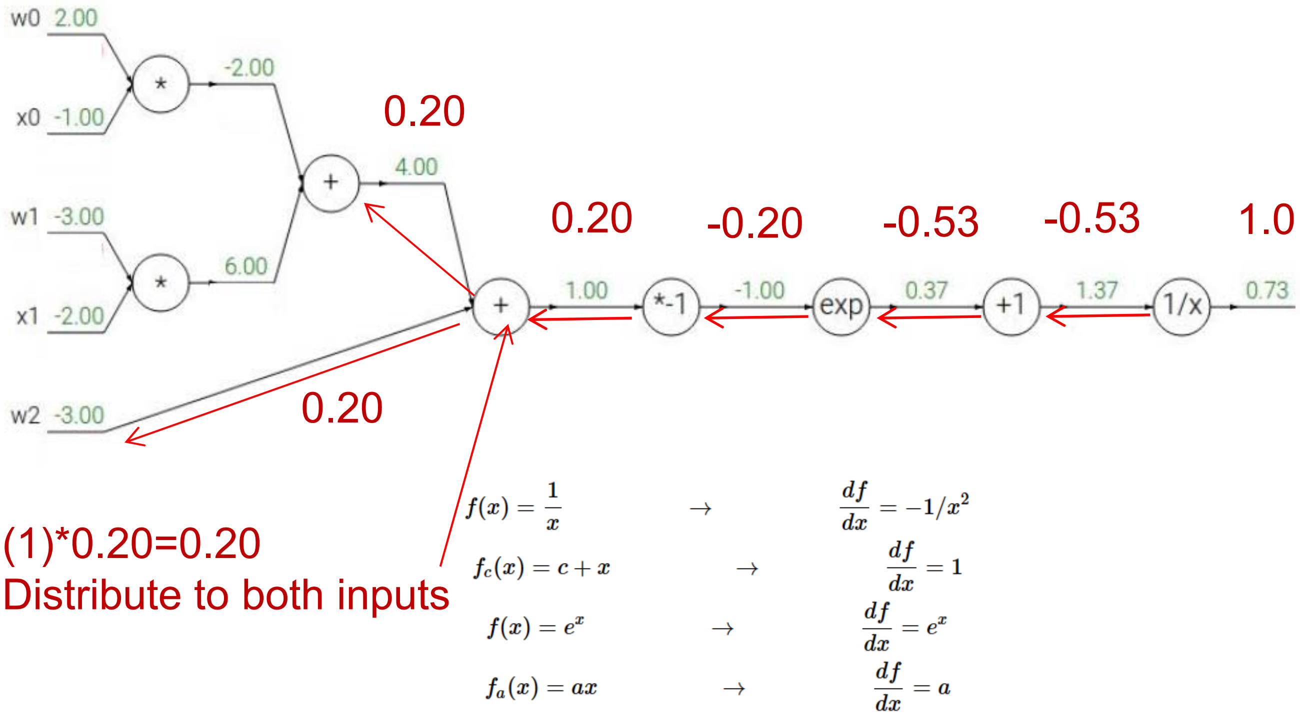
$$f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -1/x^2$$

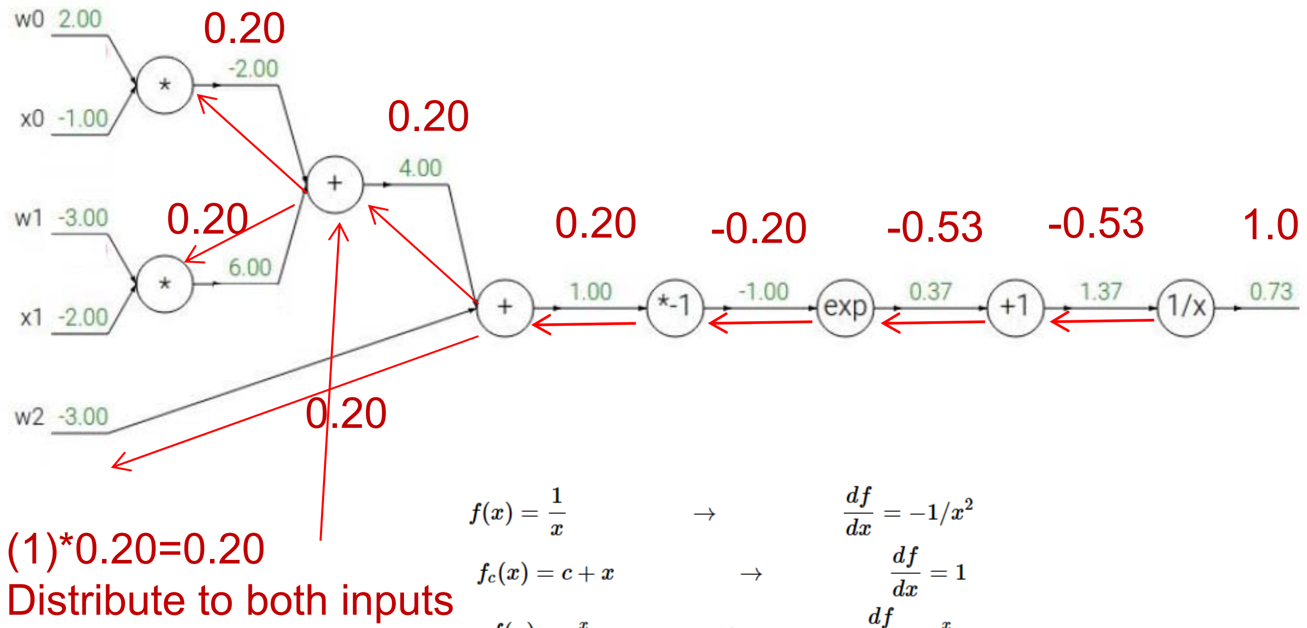
$$f_c(x) = c + x \quad \rightarrow \quad \frac{df}{dx} = 1$$

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

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

(-1)*-0.20=0.20


w0 2.00
x0 -1.00
w1 -3.00
x1 -2.00
w2 -3.00
-2.00
6.00
4.00
1.00
-1.00
0.37
1.37
0.73
*
+
*-1
exp
+1
1/x
0.20
0.20
-0.20
-0.53
-0.53
1.0
0.20
(1)*0.20=0.20
Distribute to both inputs
$f(x) = \frac{1}{x} \rightarrow \frac{df}{dx} = -1/x^2$
$f_c(x) = c + x \rightarrow \frac{df}{dx} = 1$
$f(x) = e^x \rightarrow \frac{df}{dx} = e^x$
$f_a(x) = ax \rightarrow \frac{df}{dx} = a$

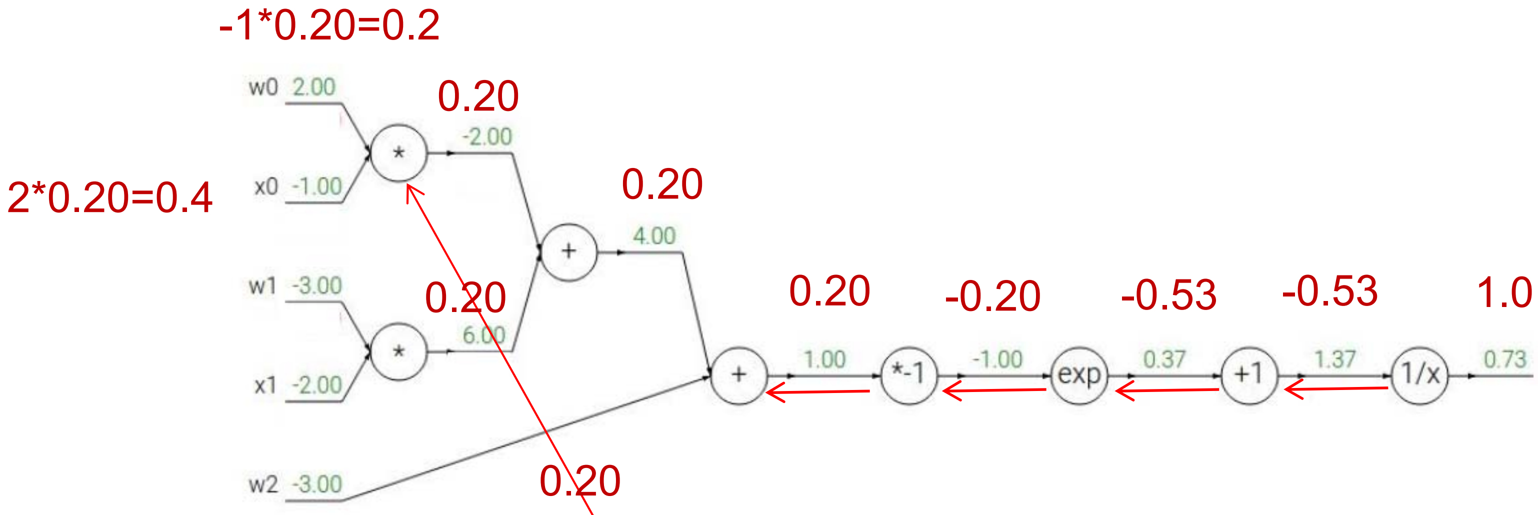


$$f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -1/x^2$$

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

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

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

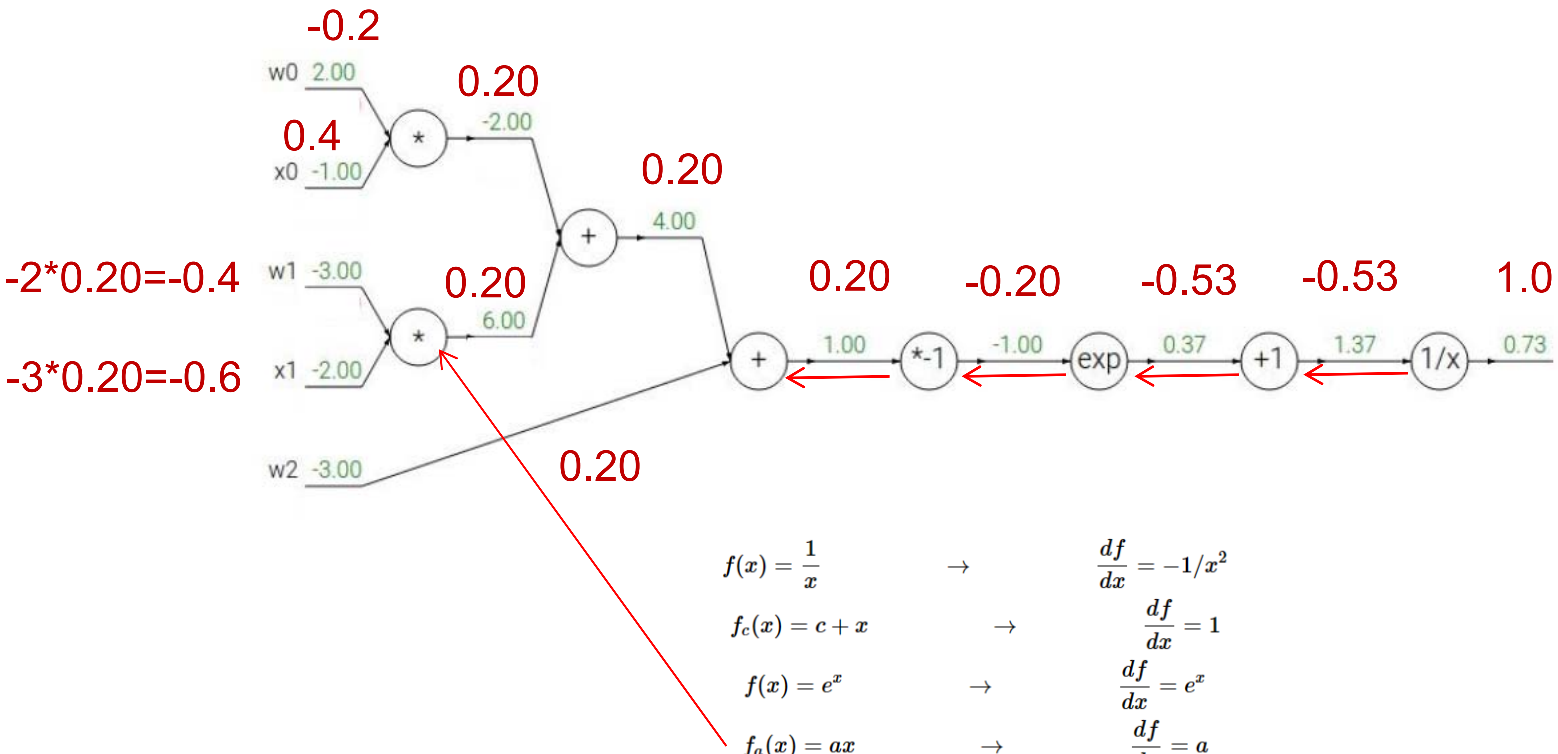


$$f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -1/x^2$$

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

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

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

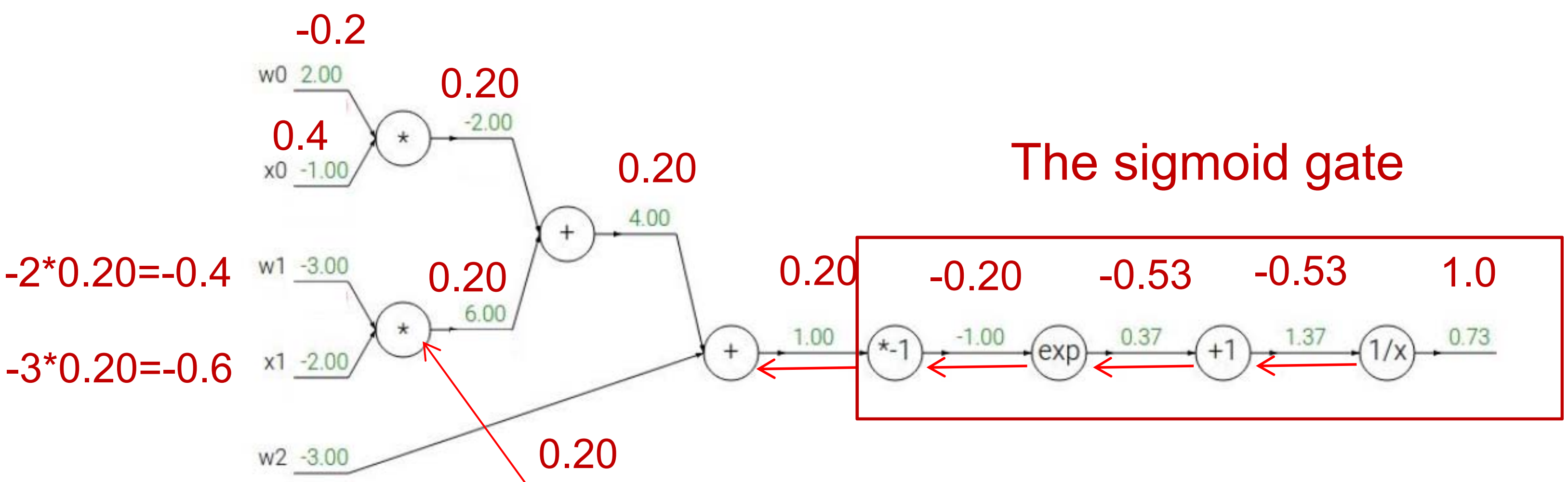


$$f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -1/x^2$$

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

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

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



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

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

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

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

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

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
```

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

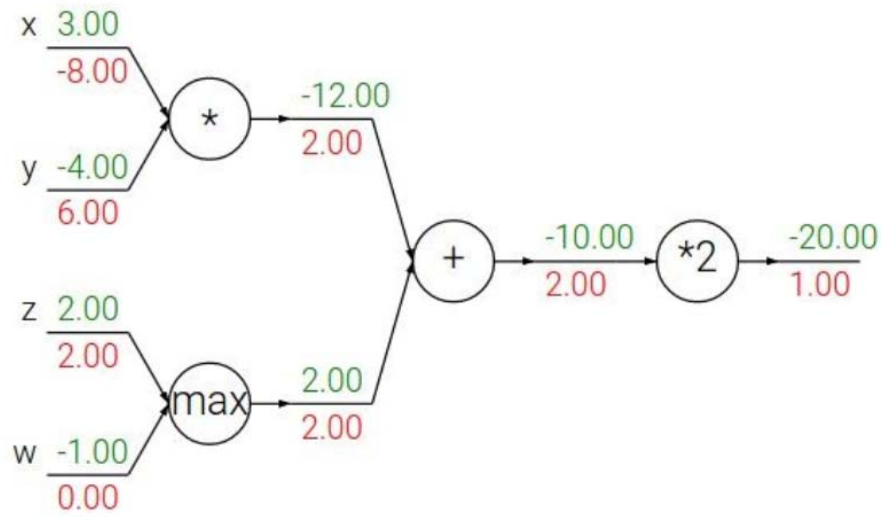
$$f(x,y) = \frac{x + \sigma(y)}{\sigma(x) + (x+y)^2}$$

- 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)
```

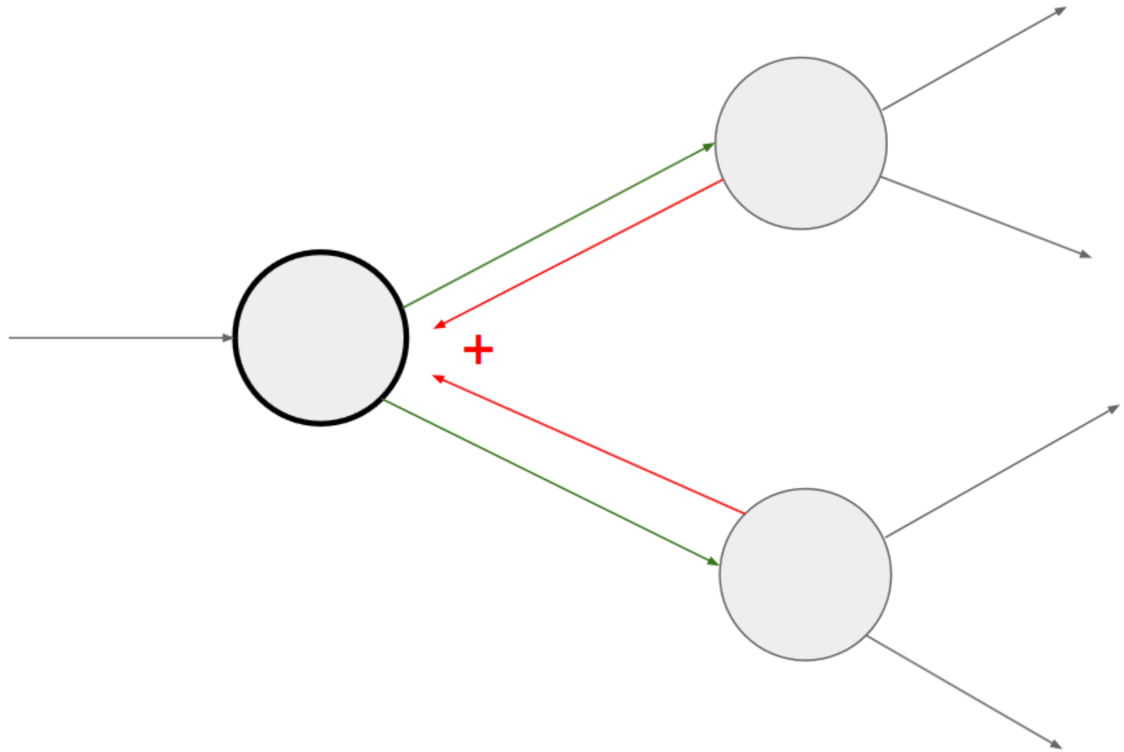
Patterns in backward flow

add gate: gradient distributor
max gate: gradient router
mul gate: gradient…. «switcher»



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

Gradients add at branches



Next week:

- Next week: Backpropagation in detail
- Vectorized implementation of backpropagation
  - Reading material:
    - http://cs231n.github.io/optimization-2/

    - Additional optional material:
    - Lecture on backpropagation in Coursera Course on Machine Learning (Andrew Ng)
    - http://yann.lecun.com/exdb/publis/pdf/lecun-98b.pdf
    - http://colah.github.io/posts/2015-08-Backprop/
    - http://neuralnetworksanddeeplearning.com/chap2.html