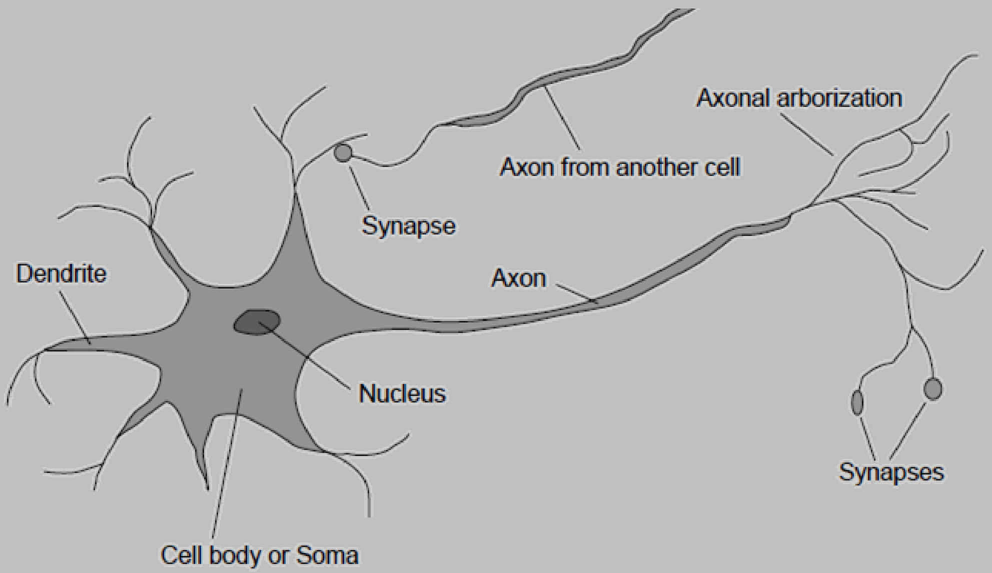
# BMB3015 ARTIFICIAL INTELLIGENCE

## NEURAL NETWORKS

# Brain Neurons

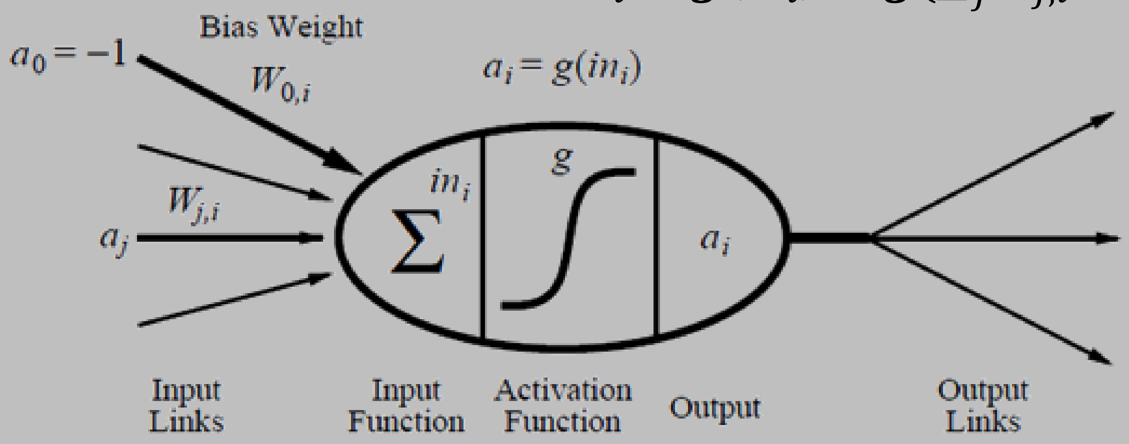
- $10^{11}$ neurons of > 20 types, $10^{14}$ synapses, 1ms-10ms cycle time
- Neurons can handle complex visual, audial or tactile signals
- Signals are noisy "spike trains" of electrical potential
- Typical processing cycle is recieve → analyse → send
- Hebbian learning (1949) asserts that neurons that fire together wire together
  - neural connections are strengthened through use
  - this may be the foundation of learning within the brain



# McCulloch-Pitts: Unit

- Single neuron ≈ percepton ≈ unit
  - acts as a linear classifier.
- Output is a " squashed" linear function of the inputs.

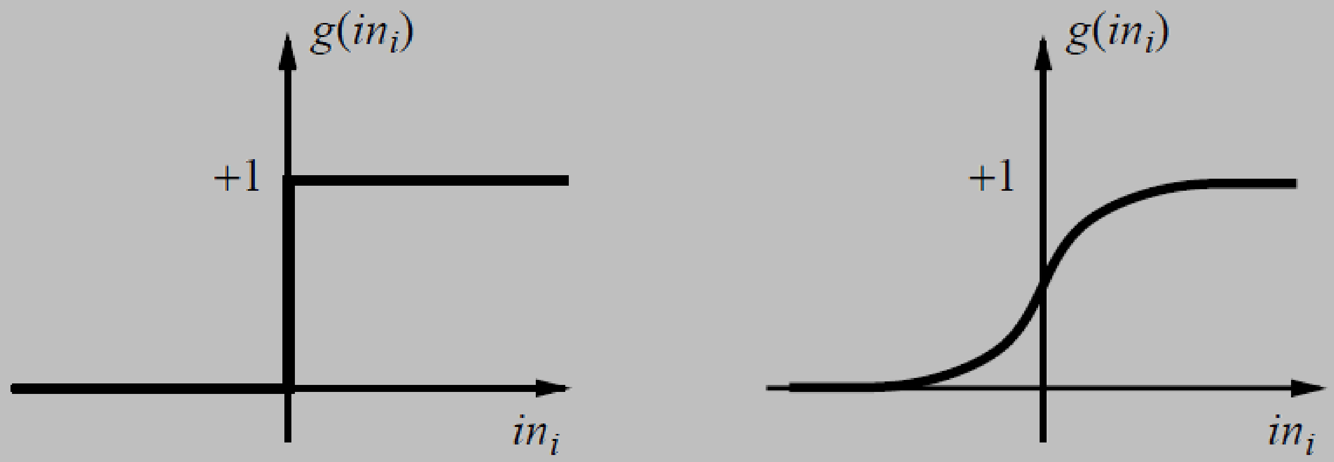
$$a_i = \boldsymbol{g}(in_i) = \boldsymbol{g}(\sum_j W_{j,i} \,.\, a_j)$$




- A gross oversimplification of real neurons, but its purpose is to develop understanding of what networks of simple units can do

# Activation Functions

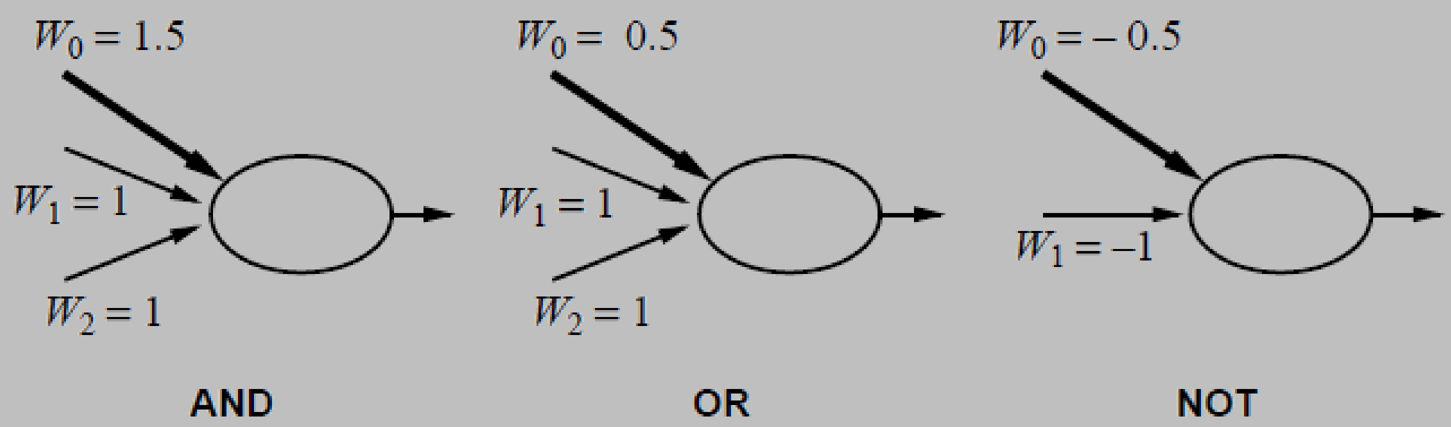
- A step function or threshold function on the left
- Sigmoid function $1/(1 + e^{-x})$ on the right
- Changing the bias weight $W_{0,i}$ moves the threshold location



(a) (b)

# Implementing Logical Functions

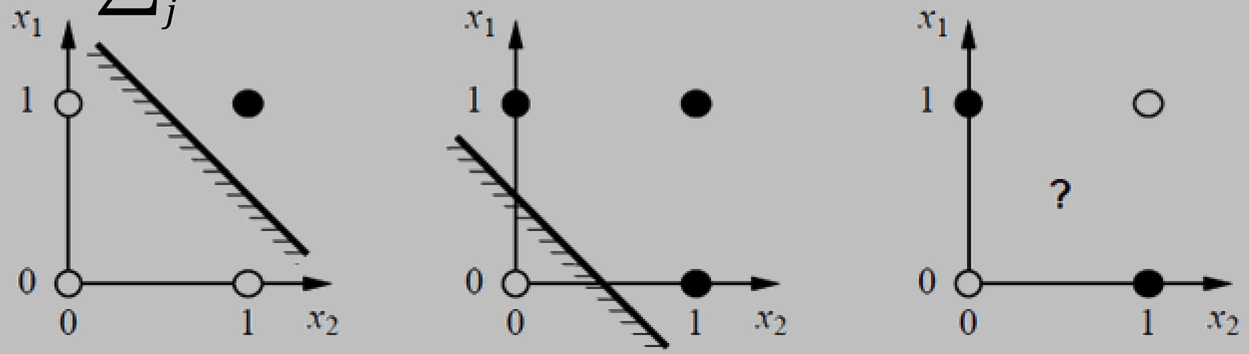
- McCulloch and Pitts:
  - every Boolean function can be implemented with a perceptron
- $h_W(\boldsymbol{x}) = \boldsymbol{g}(W_0 * -1 + W_1 x_1 + W_2 x_2)$
  - $\boldsymbol{g}$ is step function



# Expressiveness of Perceptrons

- Consider a perceptron with g = step function
- Can represent AND, OR, NOT, majority, etc., but not XOR or XNOR
- Represents a linear separator in input space:

$$\sum_j W_j . x_j > 0 \text{ or } \mathbf{W}.\mathbf{x} > 0$$




(a) $x_1$ **and** $x_2$ (b) $x_1$ **or** $x_2$ (c) $x_1$ **xor** $x_2$

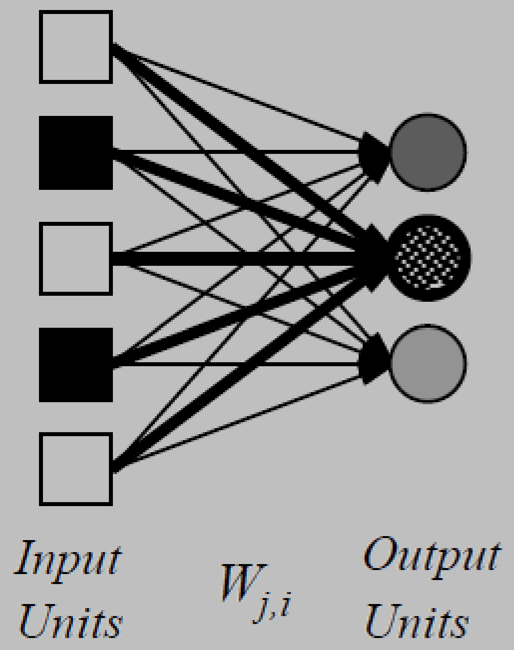
Minsky & Papert (1969) pricked the neural network balloon

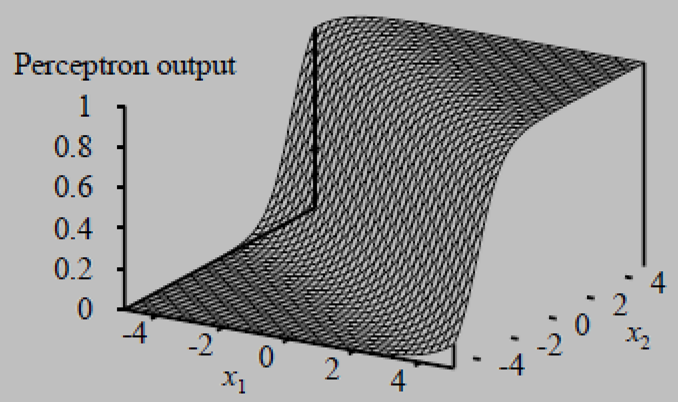
# Neural Network Architectures

- Feed-forward networks:
  - single-layer perceptrons
  - multi-layer perceptrons
- Feed-forward networks implement functions, have no internal state
- Recurrent networks:
  - Hopfield networks have symmetric weights ($W_{i,j} = W_{j,i}$)
    - $g(x) = \text{sign}(x)$ , $a_i = \pm 1$ ; holographic associative memory
  - Boltzmann machines use stochastic activation functions,
    - Similar to Markov Chain Monte Carlo (MCMC) in Bayes nets
  - Recurrent neural nets have directed cycles with delays
    - have internal state (like flip-flops), can oscillate etc.

# Single-Layer Perceptrons

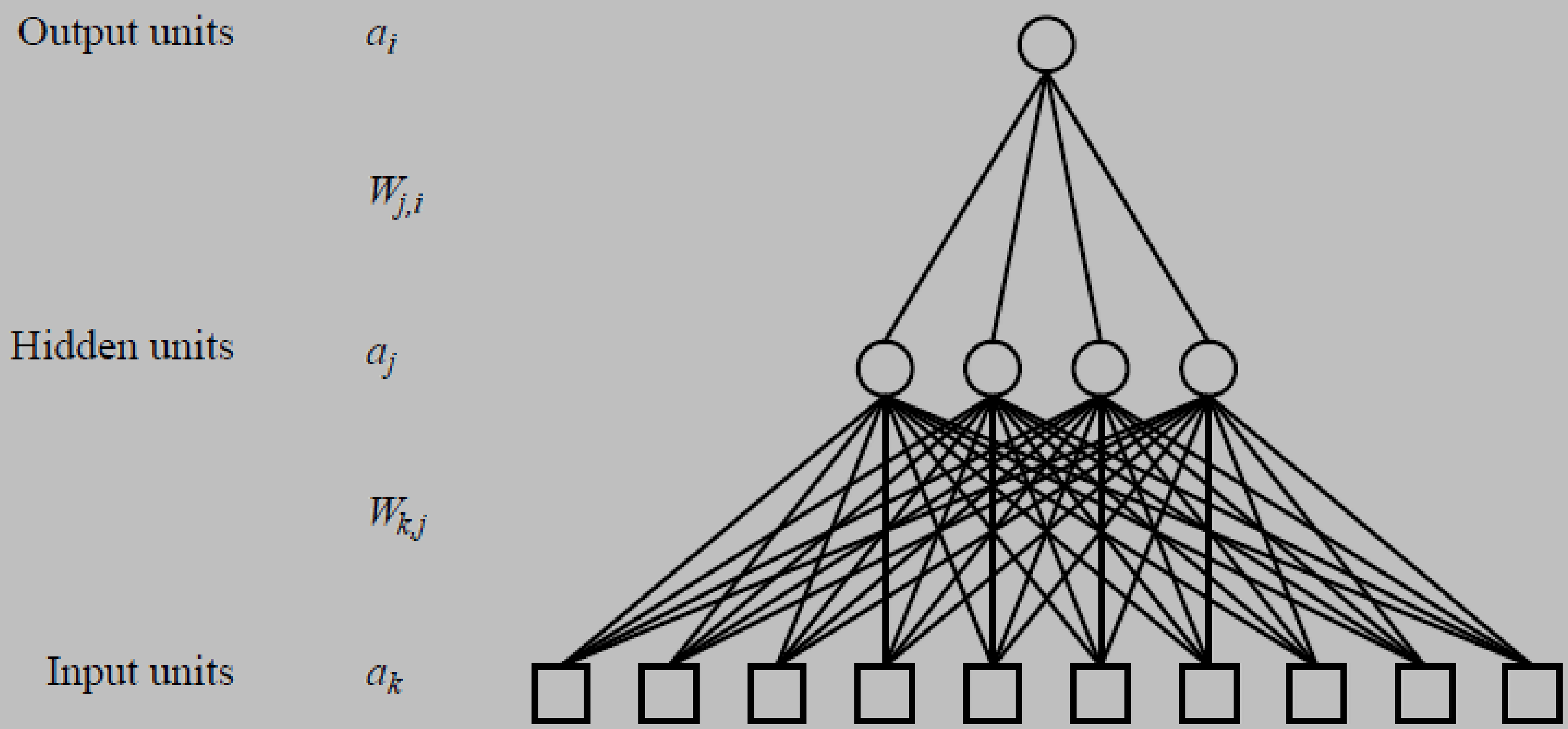
- Output units all operate separately-no shared weights
- Adjusting weights moves the location, orientation, and steepness of cliff



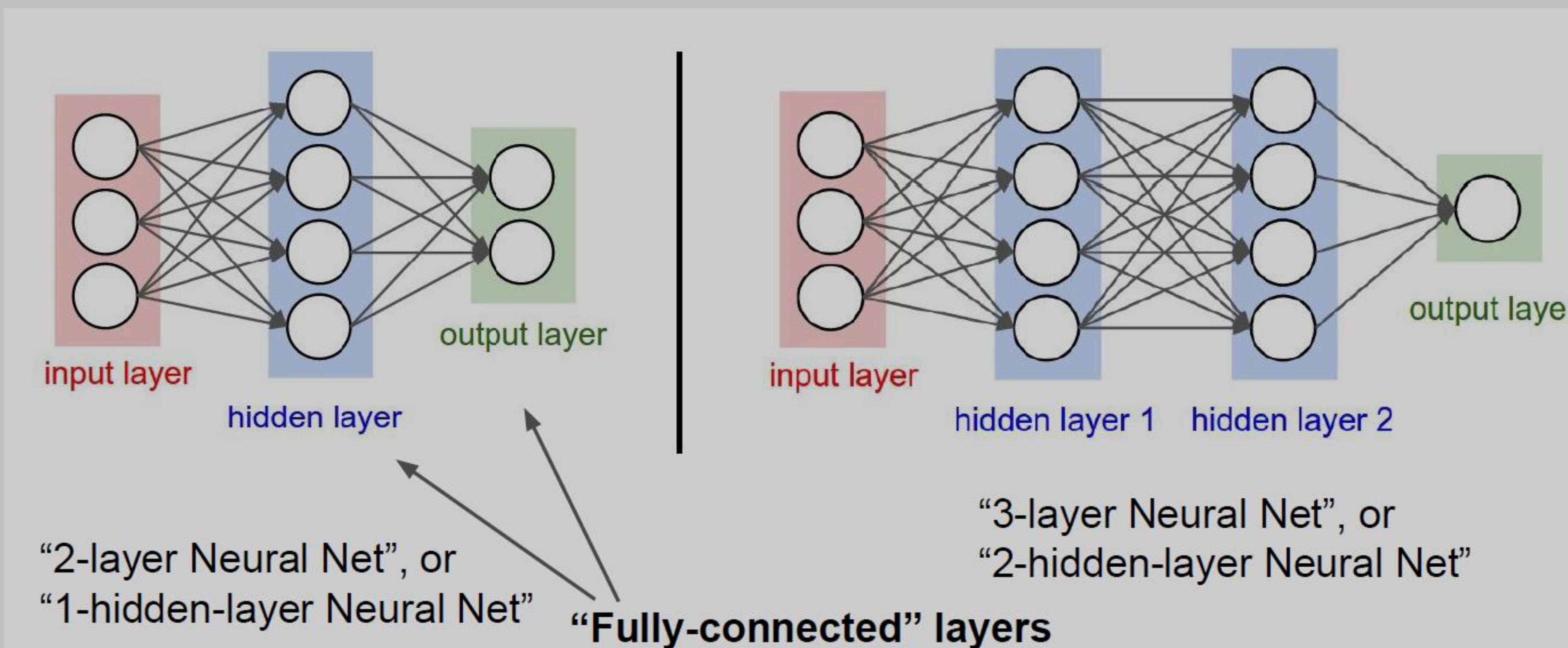
# Multi-Layer Perceptrons (MLP)

- Includes one or more hidden layers
  - Layers are usually fully connected
  - Numbers of hidden units in a hidden layer are typically chosen manually



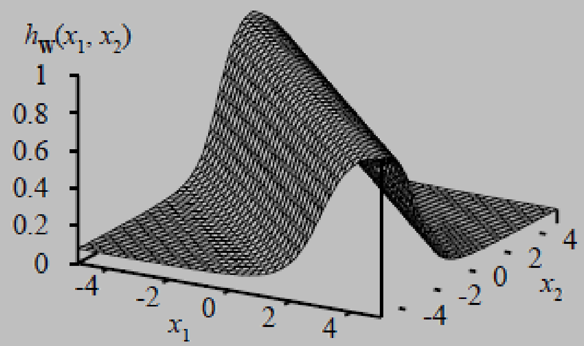
# Multi-Layer Perceptrons (MLP)

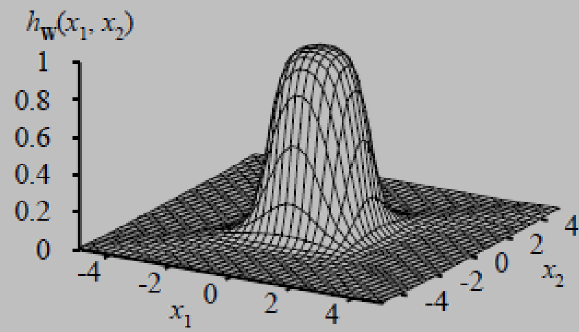
- Training dataset contain «Input» and «Output».
- Network weights represented by w are learnt through forward propagation and backpropagation.



# Expressiveness of MLPs

- All continuous functions with 2 layers and all functions with 3 layers



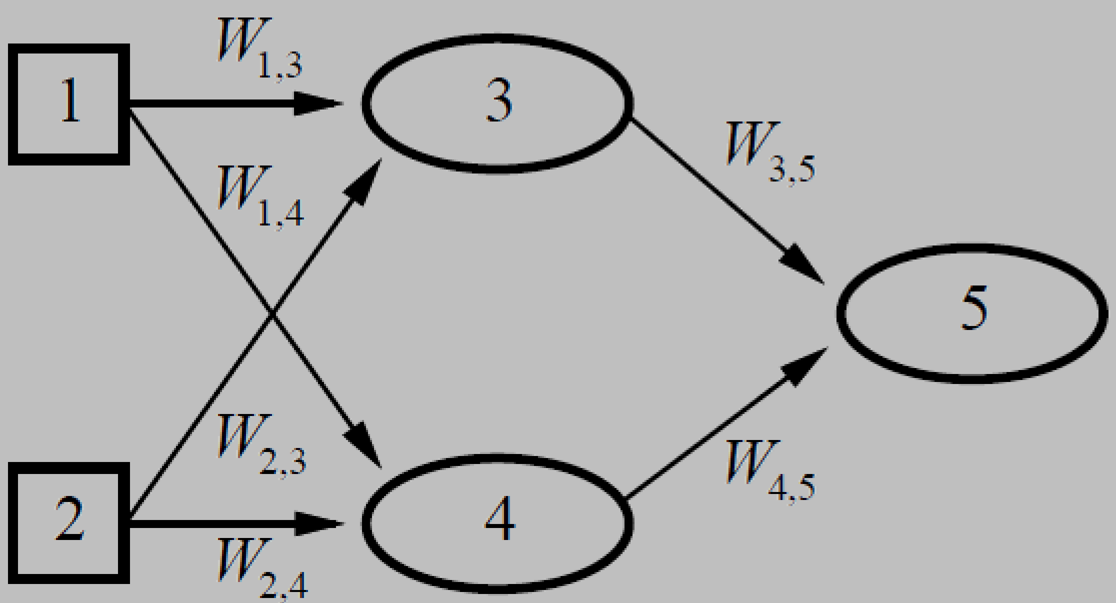

- Combine two opposite-facing threshold functions to make a ridge
- Combine two perpendicular ridges to make a bump
- Add bumps of various sizes and locations to threshold any surface
- Proof requires exponentially many hidden units

# Feedforward Example

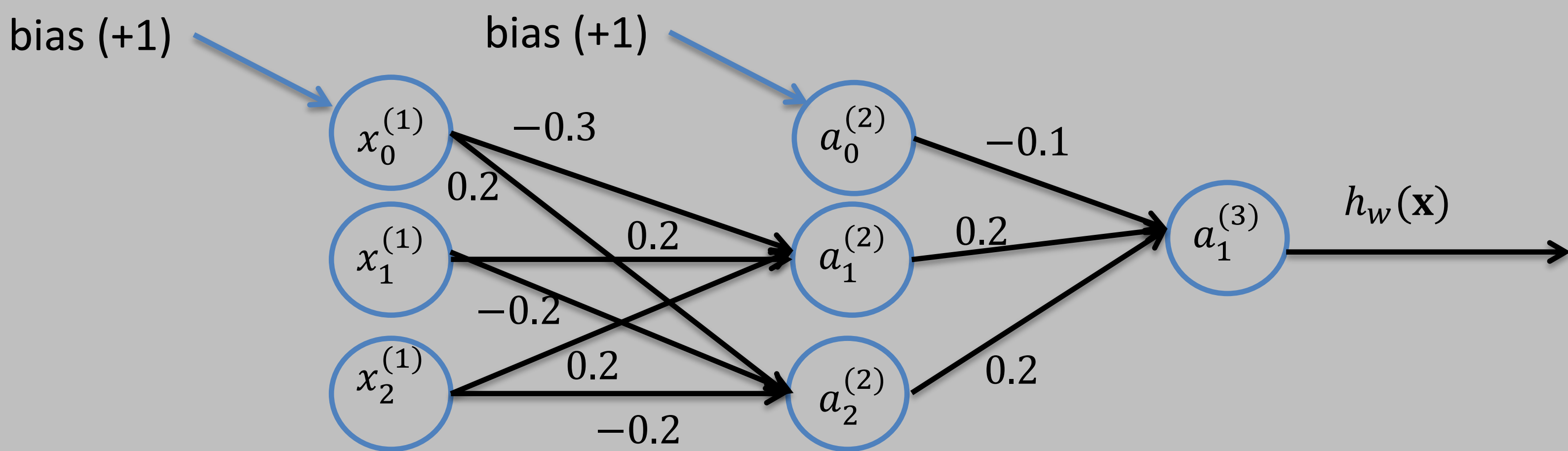
- Feed-forward network ≈ a parameterized family of nonlinear functions

$$a_5 = \boldsymbol{g}(W_{3,5}.\,a_3 + W_{4,5}.\,a_4)$$
$$= \boldsymbol{g}\left(W_{3,5}.\,\boldsymbol{g}(W_{1,3}.\,a_1 + W_{2,3}.\,a_2) + W_{4,5}.\,\boldsymbol{g}(W_{1,4}.\,a_1 + W_{2,4}.\,a_2)\right)$$

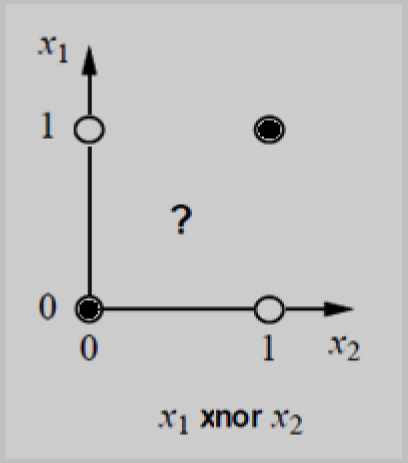
- Adjusting weights changes the function: do learning this way!



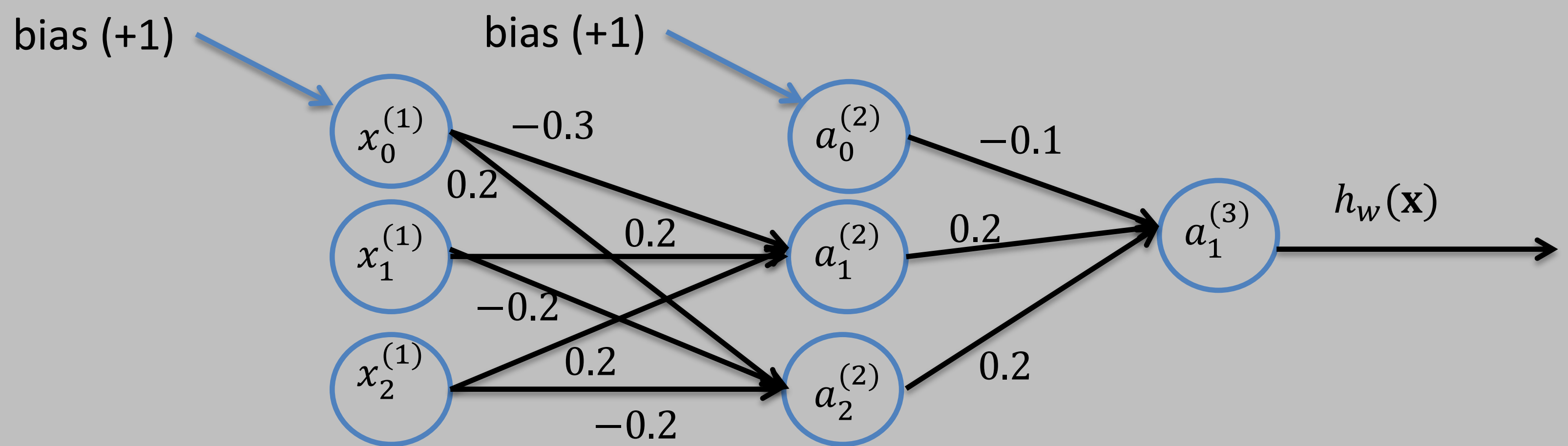
# Solution to Logical XNOR with MLP




| $x_0^{(1)} * W_{0,1}^{(1)} + x_1^{(1)} * W_{1,1}^{(1)} + x_2^{(1)} * W_{2,1}^{(1)}$ | $z_1^{(2)}$ | $a_1^{(2)}$ | $z_2^{(2)}$ | $a_2^{(2)}$ | $z_1^{(3)}$ | $a_1^{(3)}$ |
|---|---|---|---|---|---|---|
| $(1 * -0.3 + 0 * 0.2 + 0 * 0.2)$ | $-0.3$ | 0 | 0.2 | 1 | 0.1 | 1 |
| $(1 * -0.3 + 0 * 0.2 + 1 * 0.2)$ | $-0.1$ | 0 | 0 | 0 | -0.1 | 0 |
| $(1 * -0.3 + 1 * 0.2 + 0 * 0.2)$ | $-0.1$ | 0 | 0 | 0 | -0.1 | 0 |
| $(1 * -0.3 + 1 * 0.2 + 1 * 0.2)$ | 0.1 | 1 | -0.2 | 0 | 0.1 | 1 |



# Notation for Logical XNOR with MLP



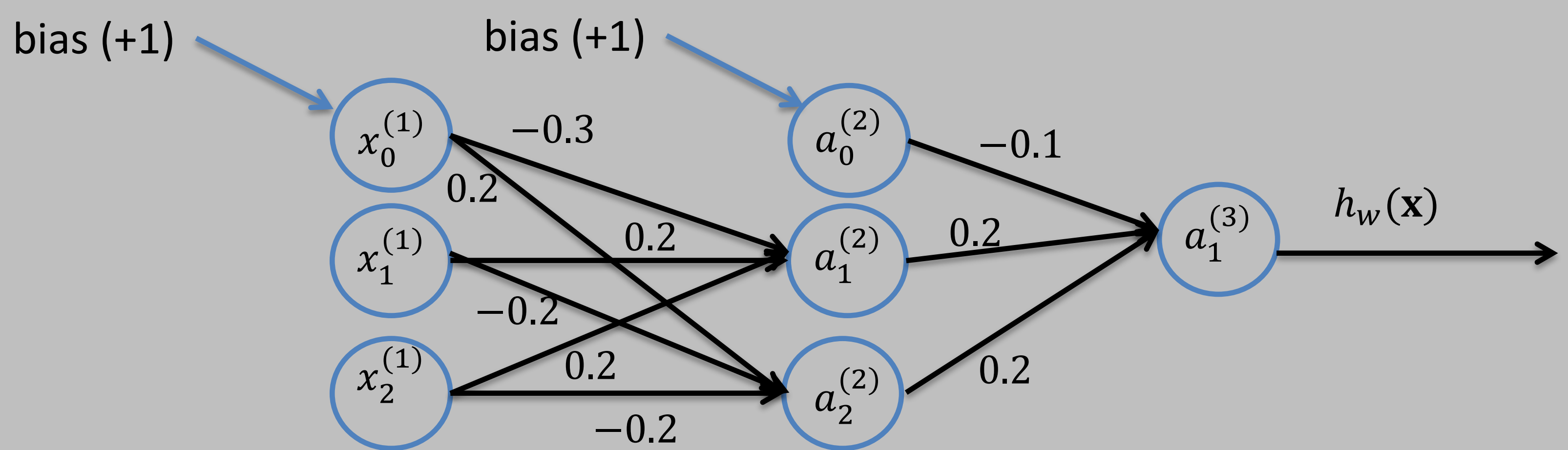

Number of neurons in layer (j+1)

$$\boldsymbol{W}^{(1)} = \begin{bmatrix} w_{10}^{(1)} & w_{11}^{(1)} & w_{12}^{(1)} \\ w_{20}^{(1)} & w_{21}^{(1)} & w_{22}^{(1)} \end{bmatrix}_{(s_{j+1})\times(s_j+1)=s_2\times(s_1+1)} \qquad \boldsymbol{X}^{(1)} = \begin{bmatrix} x_0^{(1)} \\ x_1^{(1)} \\ x_2^{(1)} \end{bmatrix}_{(s_1+1)\times 1}$$

Number of neurons in layer j

$$\boldsymbol{W}^{(1)}{}_{s_2\times(s_1+1)} \times \boldsymbol{X}^{(1)}{}_{(s_1+1)\times 1} = \boldsymbol{z}^{(2)}{}_{s_2\times 1} \qquad \boldsymbol{g}(\boldsymbol{z}^{(2)}) = \boldsymbol{a}^{(2)}{}_{s_2\times 1}$$

# Notation for Logical XNOR with MLP



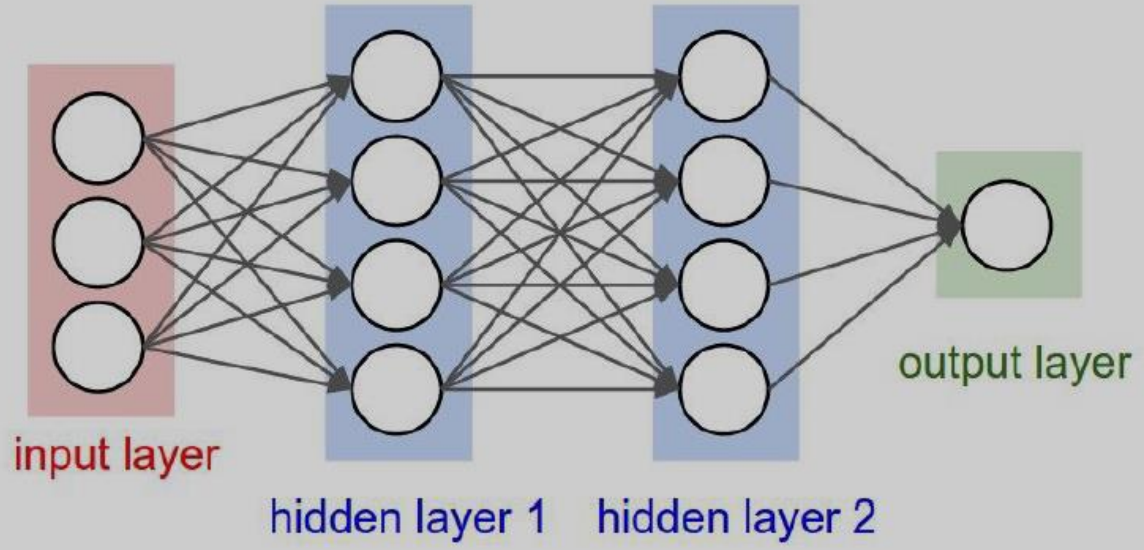

$$\boldsymbol{W}^{(2)} = \begin{bmatrix} w_{10}^{(2)} & w_{11}^{(2)} & w_{12}^{(2)} \end{bmatrix}_{s_3 \times (s_2+1)} \qquad \boldsymbol{a}^{(2)} = \begin{bmatrix} a_0^{(2)} \\ a_1^{(2)} \\ a_2^{(2)} \end{bmatrix}_{(s_2+1)\times 1}$$

$$\boldsymbol{W}^{(2)}{}_{s_3\times(s_2+1)} \times \boldsymbol{a}^{(2)}{}_{(s_2+1)\times 1} = \boldsymbol{z}^{(3)}{}_{s_3\times 1} \qquad \boldsymbol{g}(\boldsymbol{z}^{(3)}) = \boldsymbol{a}^{(3)}{}_{s_3\times 1}$$

$$h_W(\boldsymbol{x}) = \boldsymbol{g}(w_{10}^{(2)}.a_0^{(2)} + w_{11}^{(2)}a_1^{(2)} + w_{12}^{(2)}a_2^{(2)})$$

The learning procedure of a neuron is similar to logistic regression

# Feedforward of MLP




```python
# forward-pass of a 3-layer neural network:
f = lambda x: 1.0/(1.0 + np.exp(-x)) # activation function (use sigmoid)
x = np.random.randn(3, 1) # random input vector of three numbers (3x1)
h1 = f(np.dot(W1, x) + b1) # calculate first hidden layer activations (4x1)
h2 = f(np.dot(W2, h1) + b2) # calculate second hidden layer activations (4x1)
out = np.dot(W3, h2) + b3 # output neuron (1x1)
```

# Perceptron Learning

- Learn by adjusting weights to reduce error on training set
- The squared error for an example with input x and true output y is

$$E = \frac{1}{2}Err^2 \equiv \frac{1}{2}(y - h_{\mathbf{W}}(\mathbf{x}))^2 \ ,$$

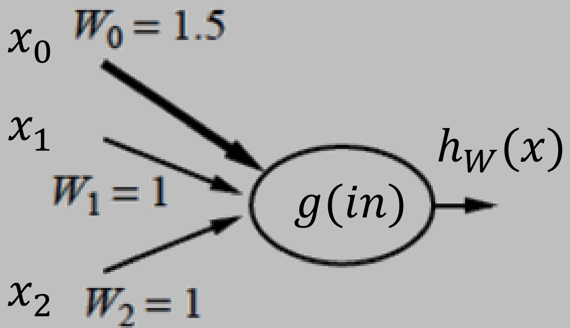
- Perform optimization search by gradient descent:

$$\begin{aligned}\frac{\partial E}{\partial W_j} &= Err \times \frac{\partial Err}{\partial W_j} = Err \times \frac{\partial}{\partial W_j}\left(y - g(\Sigma_{j=0}^{n} W_j x_j)\right) \\ &= -Err \times g'(in) \times x_j\end{aligned}$$

- Simple weight update rule:

$$W_j \leftarrow W_j + \alpha \times Err \times g'(in) \times x_j$$

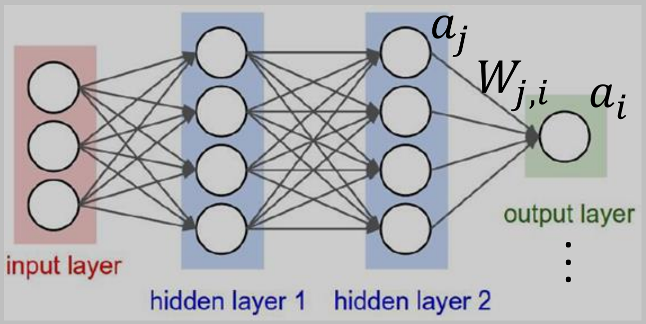
- e.g., + valued error => increase network output
  
  => increase weights on + valued errors
  
  => decrease on - valued errors



# Back-Propagation Derivation

- The squared error on a single example is defined as

$$E = \frac{1}{2}\sum_i (y_i - a_i)^2 \ ,$$




  - where the sum is over the nodes in the output layer.

$$\begin{aligned}
\frac{\partial E}{\partial W_{j,i}} &= -(y_i - a_i)\frac{\partial a_i}{\partial W_{j,i}} = -(y_i - a_i)\frac{\partial g(in_i)}{\partial W_{j,i}} \\
&= -(y_i - a_i)g'(in_i)\frac{\partial in_i}{\partial W_{j,i}} = -(y_i - a_i)g'(in_i)\frac{\partial}{\partial W_{j,i}}\left(\sum_j W_{j,i}a_j\right) \\
&= -(y_i - a_i)g'(in_i)a_j = -a_j\Delta_i
\end{aligned}$$

# Back-Propagation Learning

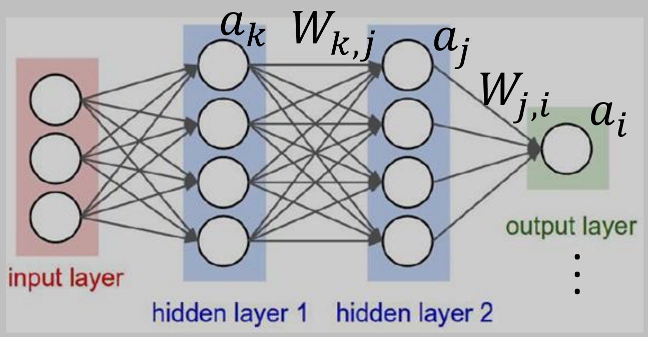
- Output layer: same as for single-layer perceptron,

$$W_{j,i} \leftarrow W_{j,i} + \alpha \times a_j \times \Delta_i$$

where $\Delta_i = Err_i \times g'(in_i)$

- Hidden layer: back-propagate the error from the output layer,

$$\Delta_j = g'(in_j) \sum_i W_{j,i} \Delta_i \ .$$



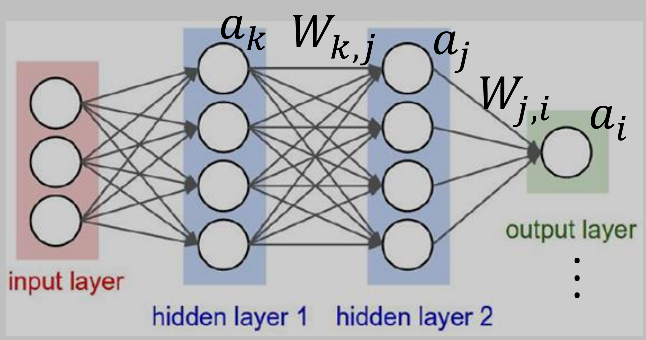

- Update rule for weights in hidden layer:

$$W_{k,j} \leftarrow W_{k,j} + \alpha \times a_k \times \Delta_j \ .$$

*( Most neuroscientists deny that back-propagation occurs in the brain )*

# Back-Propagation Derivation

$$
\begin{aligned}
\frac{\partial E}{\partial W_{k,j}} &= -\sum_i (y_i - a_i)\frac{\partial a_i}{\partial W_{k,j}} = -\sum_i (y_i - a_i)\frac{\partial g(in_i)}{\partial W_{k,j}} \\
&= -\sum_i (y_i - a_i) g'(in_i)\frac{\partial in_i}{\partial W_{k,j}} = -\sum_i \Delta_i \frac{\partial}{\partial W_{k,j}}\left(\sum_j W_{j,i} a_j\right) \\
&= -\sum_i \Delta_i W_{j,i}\frac{\partial a_j}{\partial W_{k,j}} = -\sum_i \Delta_i W_{j,i}\frac{\partial g(in_j)}{\partial W_{k,j}} \\
&= -\sum_i \Delta_i W_{j,i} g'(in_j)\frac{\partial in_j}{\partial W_{k,j}} \\
&= -\sum_i \Delta_i W_{j,i} g'(in_j)\frac{\partial}{\partial W_{k,j}}\left(\sum_k W_{k,j} a_k\right) \\
&= -\sum_i \Delta_i W_{j,i} g'(in_j) a_k = -a_k \Delta_j
\end{aligned}
$$



# Backpropagation: Simple Example

- Better idea is to use computational graphs to understand backpropagation

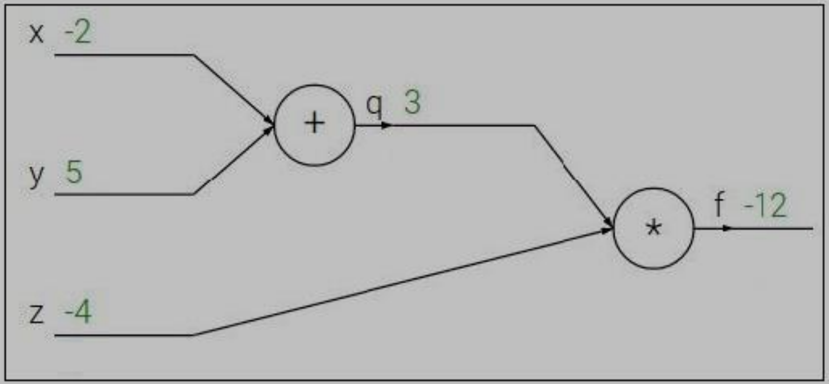
$$f(x, y, z) = (x + y)z$$

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



# Backpropagation: Simple Example

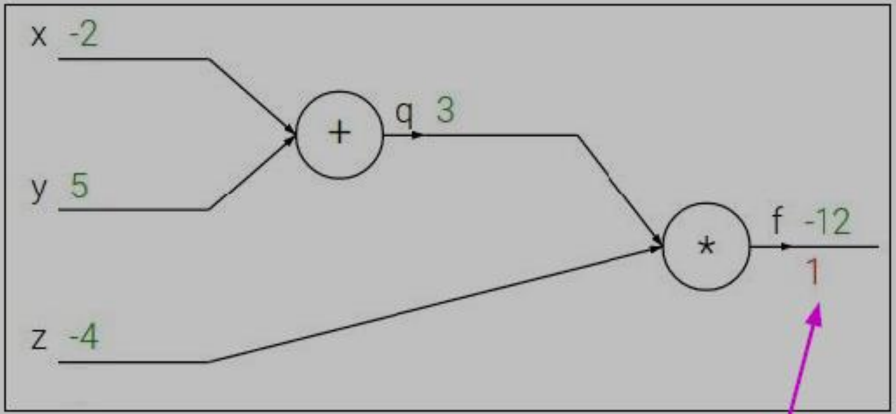
$$f(x, y, z) = (x + y)z$$

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




$$\frac{\partial f}{\partial f}$$

# Backpropagation: Simple Example

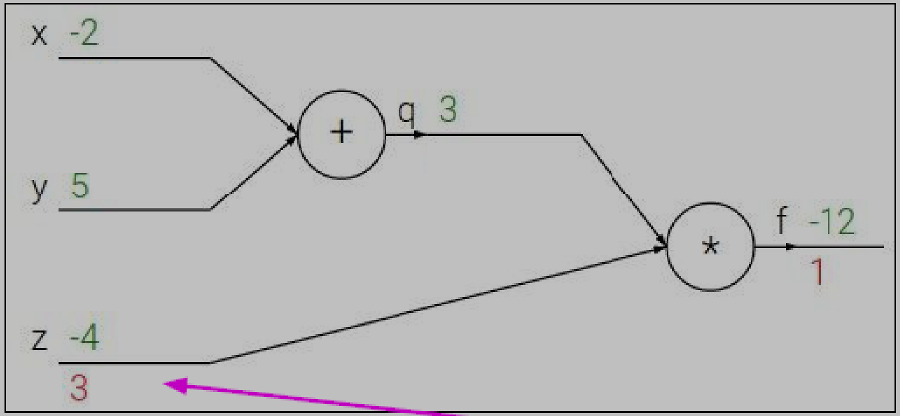
$$f(x, y, z) = (x + y)z$$

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

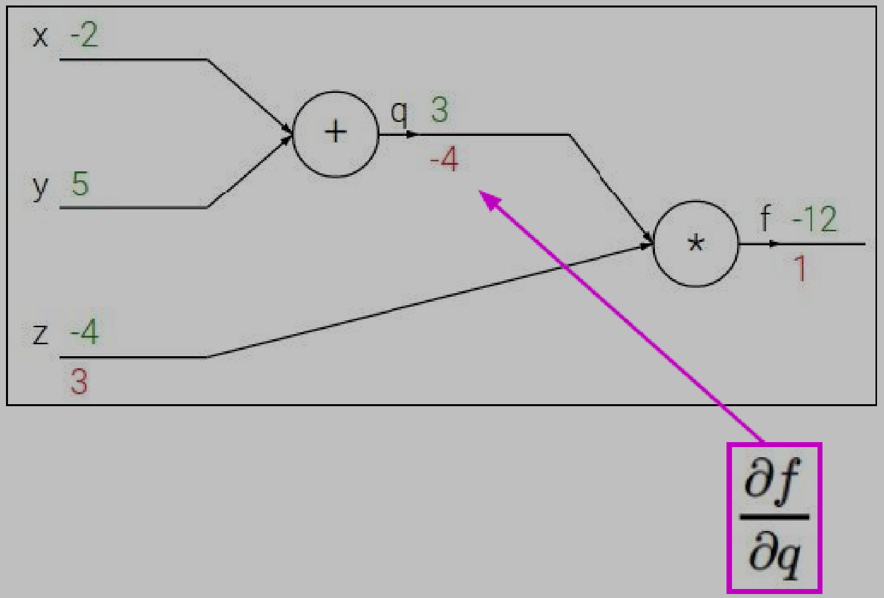



$\frac{\partial f}{\partial z}$

# Backpropagation: Simple Example

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

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

# Backpropagation: Simple Example

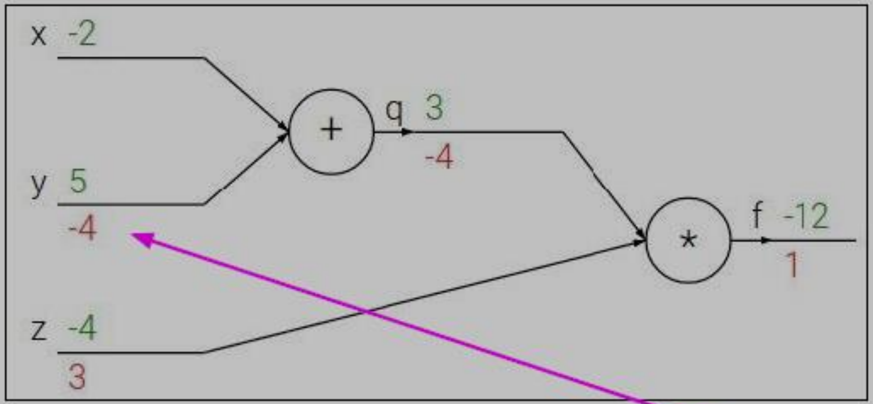
$$f(x, y, z) = (x + y)z$$

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

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

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

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




$$\boxed{\frac{\partial f}{\partial y}}$$

Chain rule:

$$\boxed{\frac{\partial f}{\partial y} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial y}}$$

Upstream gradient

Local gradient

# Backpropagation: Simple Example

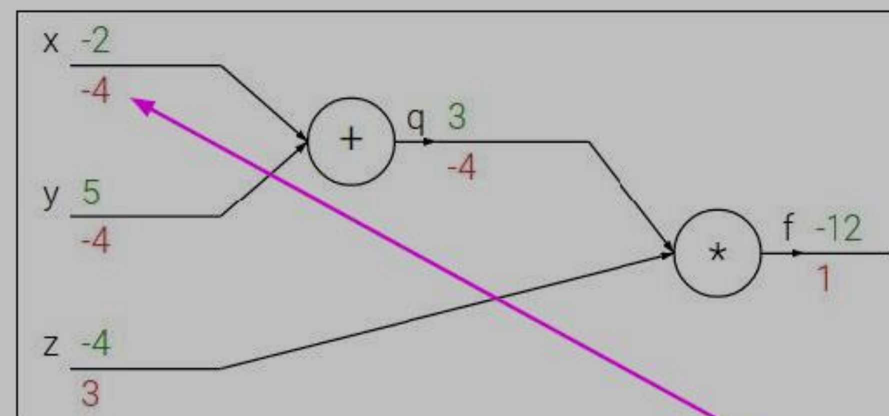
$$f(x, y, z) = (x + y)z$$

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

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

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

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

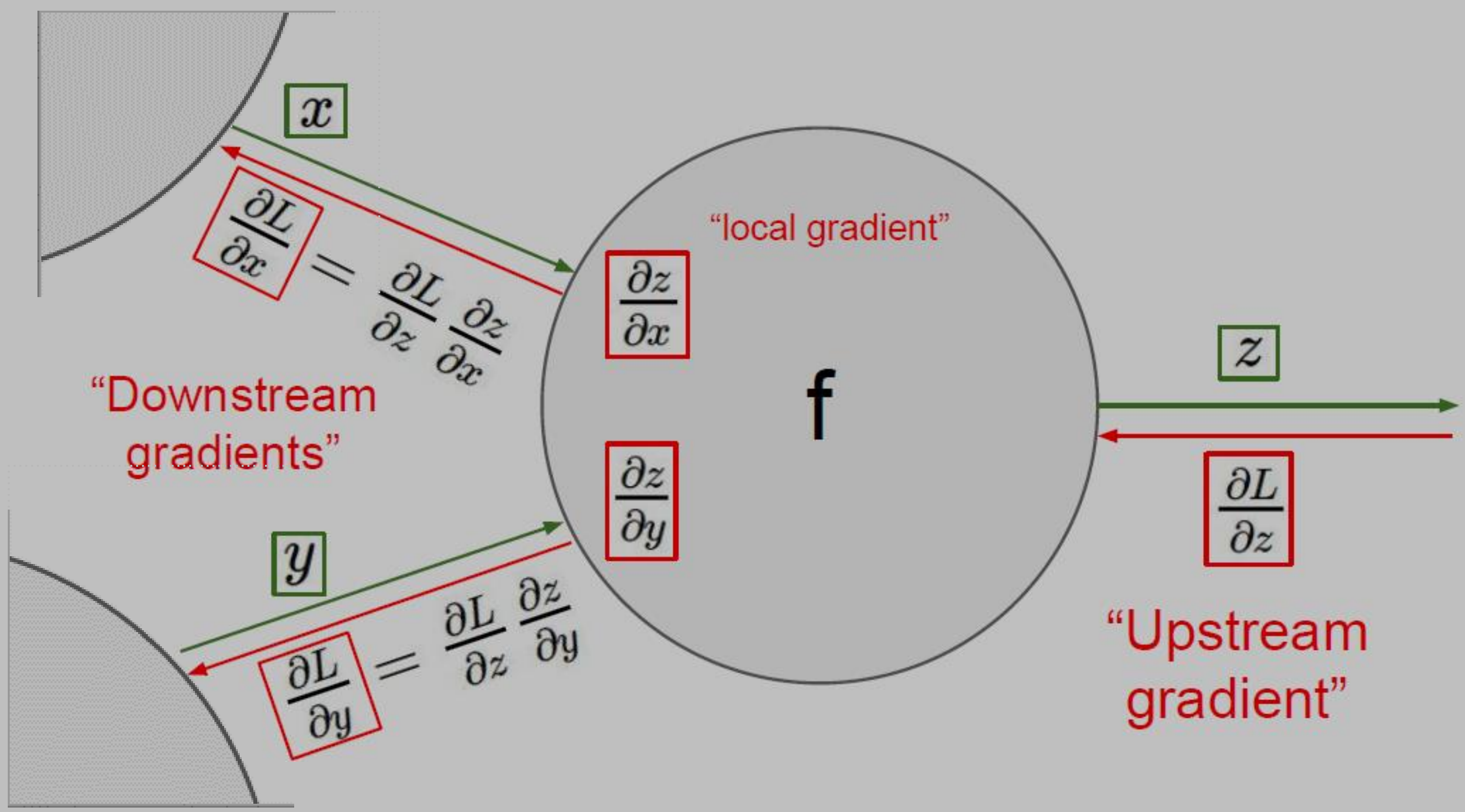



$$\boxed{\frac{\partial f}{\partial x}}$$

Chain rule:

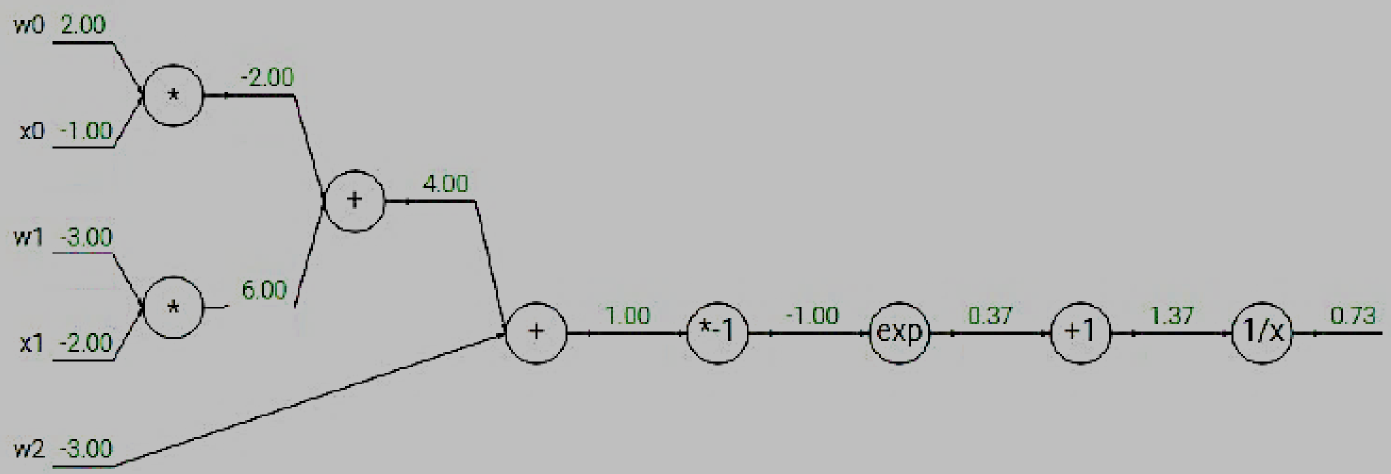
$$\boxed{\frac{\partial f}{\partial x} = \underbrace{\frac{\partial f}{\partial q}}_{\text{Upstream gradient}} \underbrace{\frac{\partial q}{\partial x}}_{\text{Local gradient}}}$$

# Backpropagating Gradients



# Backpropagation: Another Example

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




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

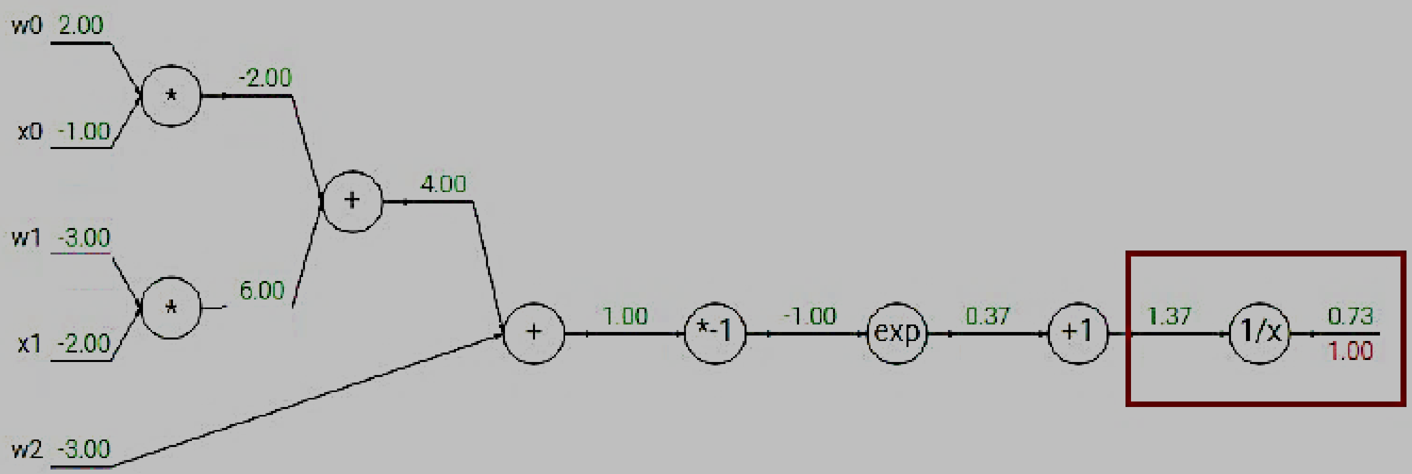
$$f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a$$

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

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

# Backpropagation: Another Example

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




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

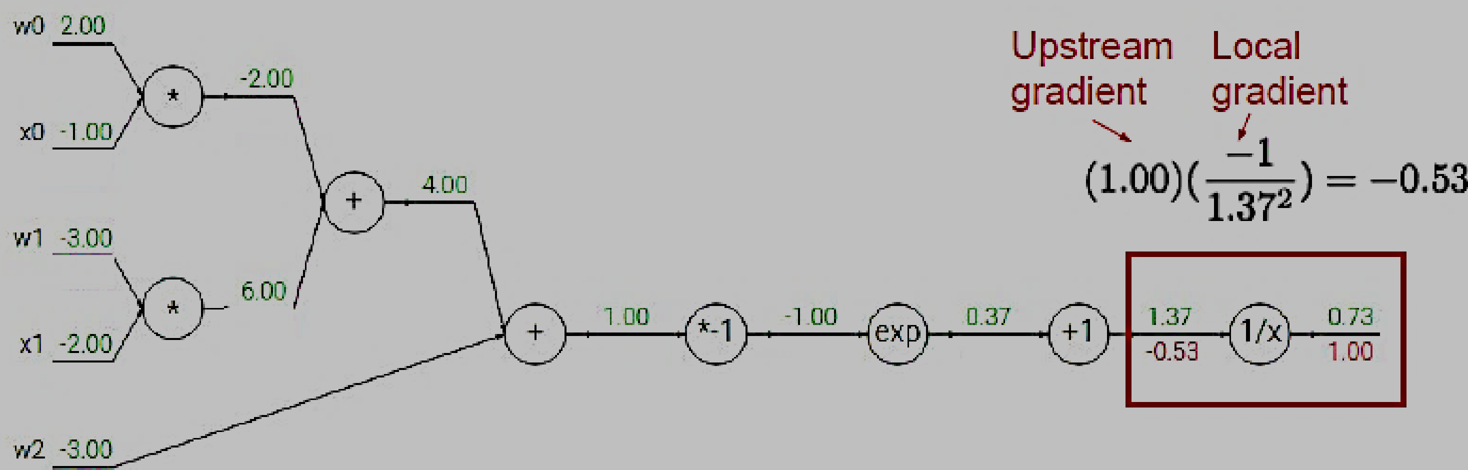
$$f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a$$

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

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

# Backpropagation: Another Example

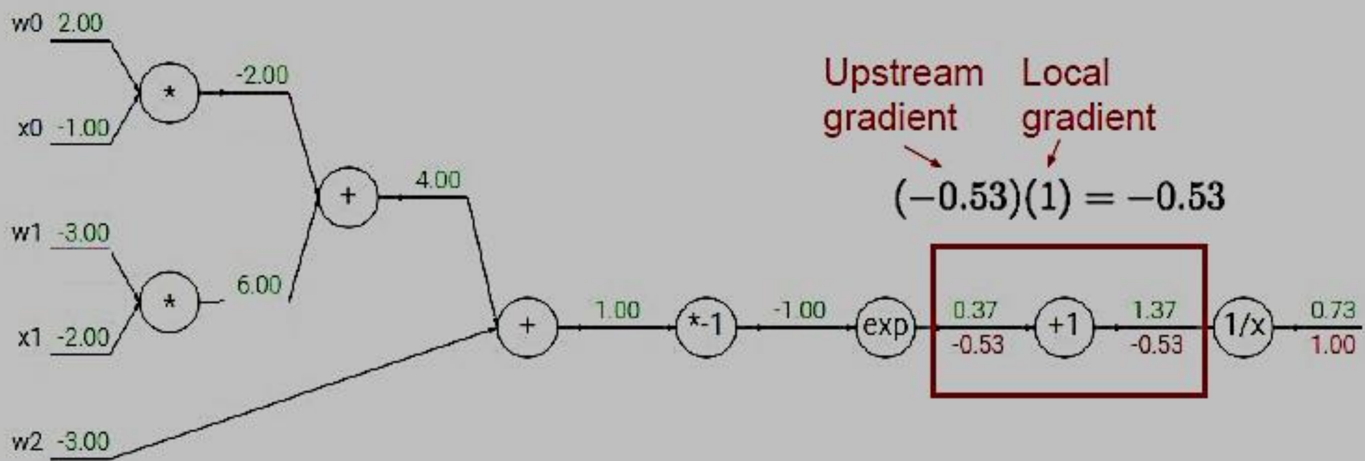
$$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$




| | | |
|---|---|---|
| $f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x$ | | $\boxed{f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -1/x^2}$ |
| $f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a$ | | $f_c(x) = c + x \quad \rightarrow \quad \frac{df}{dx} = 1$ |

# Backpropagation: Another Example

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




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

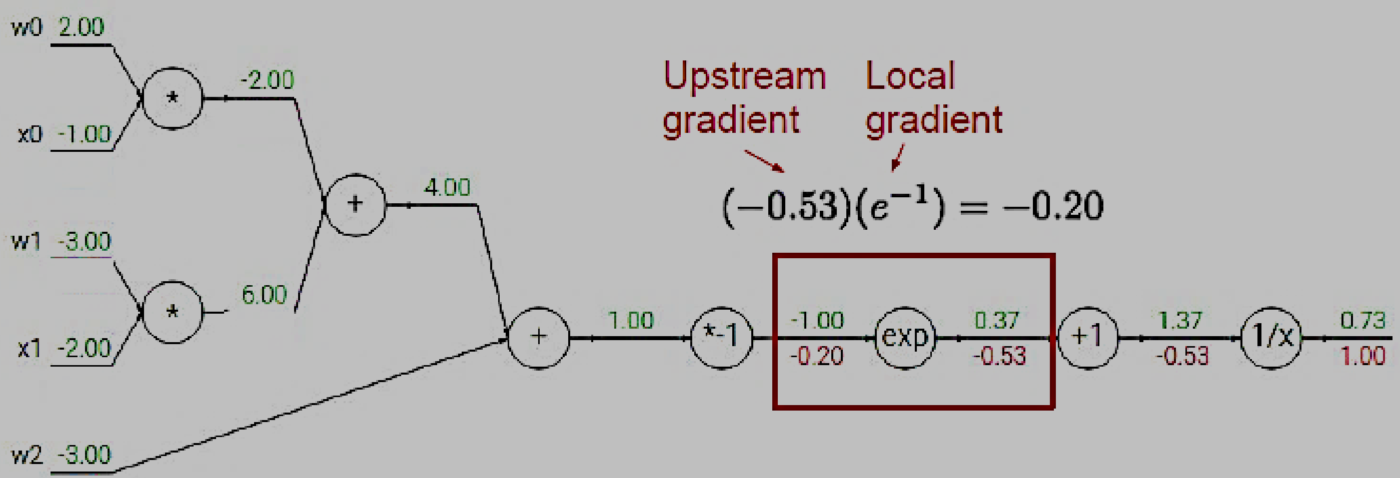
$f_a(x)=ax \quad \rightarrow \quad \frac{df}{dx}=a$

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

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

# Backpropagation: Another Example

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




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

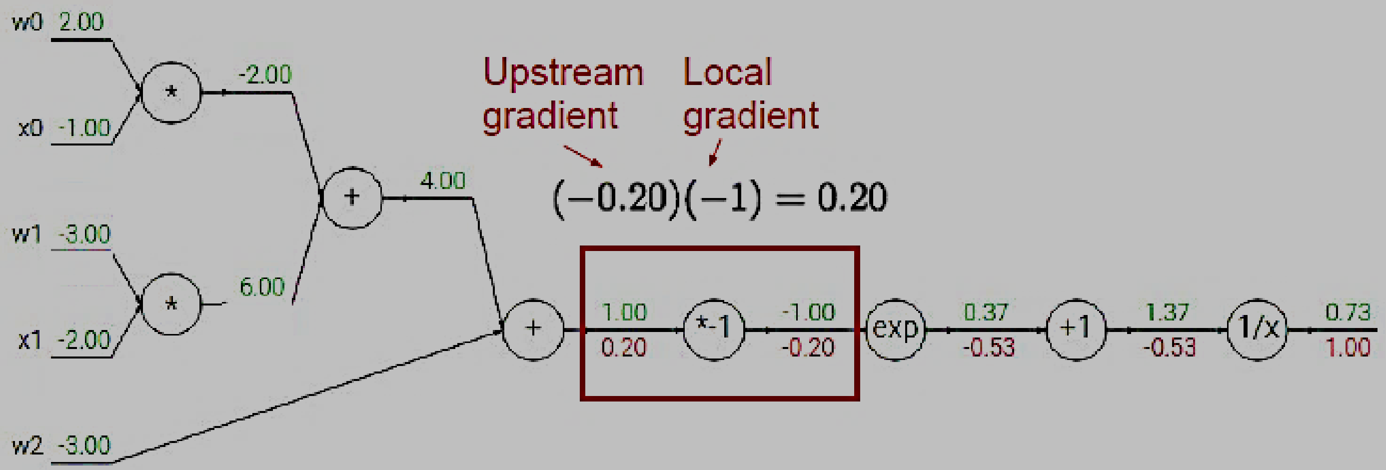
$$f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a$$

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

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

# Backpropagation: Another Example

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




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

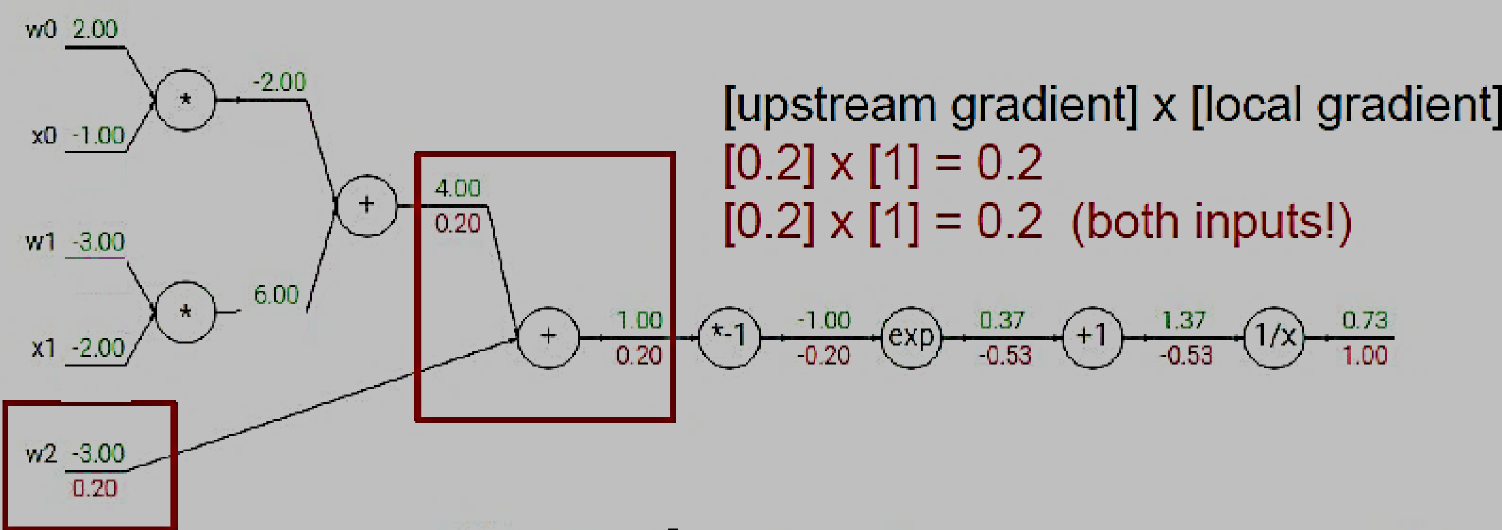
$\boxed{f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a}$

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

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

# Backpropagation: Another Example

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




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

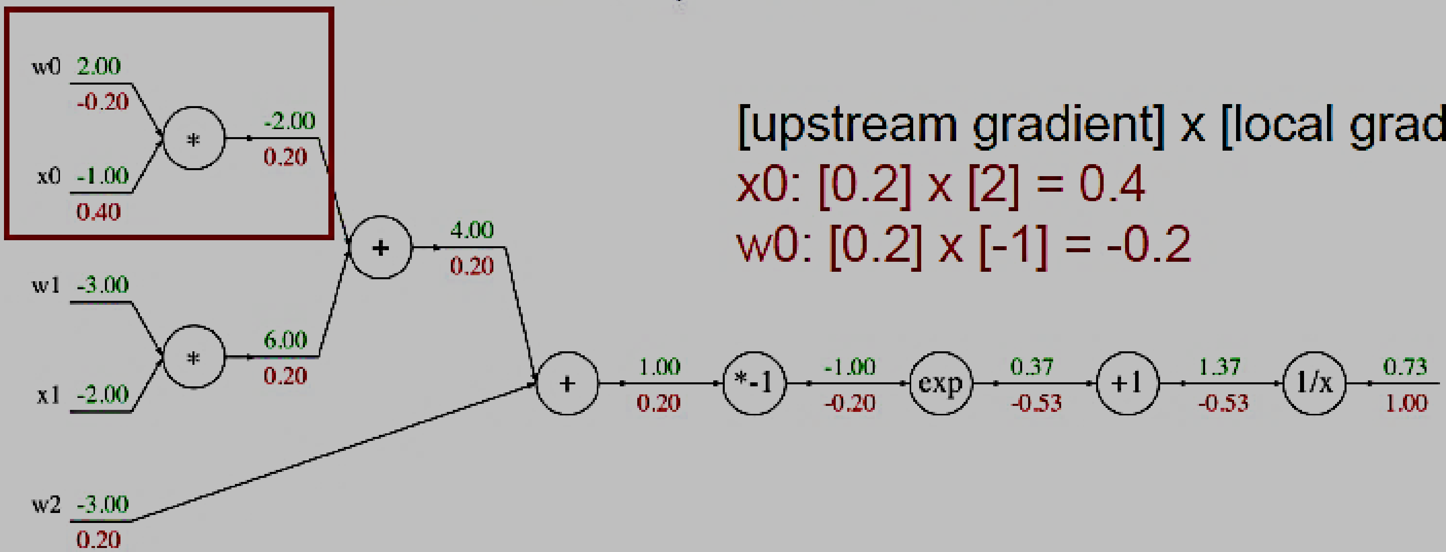
$$f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a$$

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

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

# Backpropagation: Another Example

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




[upstream gradient] x [local gradient]
x0: [0.2] x [2] = 0.4
w0: [0.2] x [-1] = -0.2

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

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

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

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

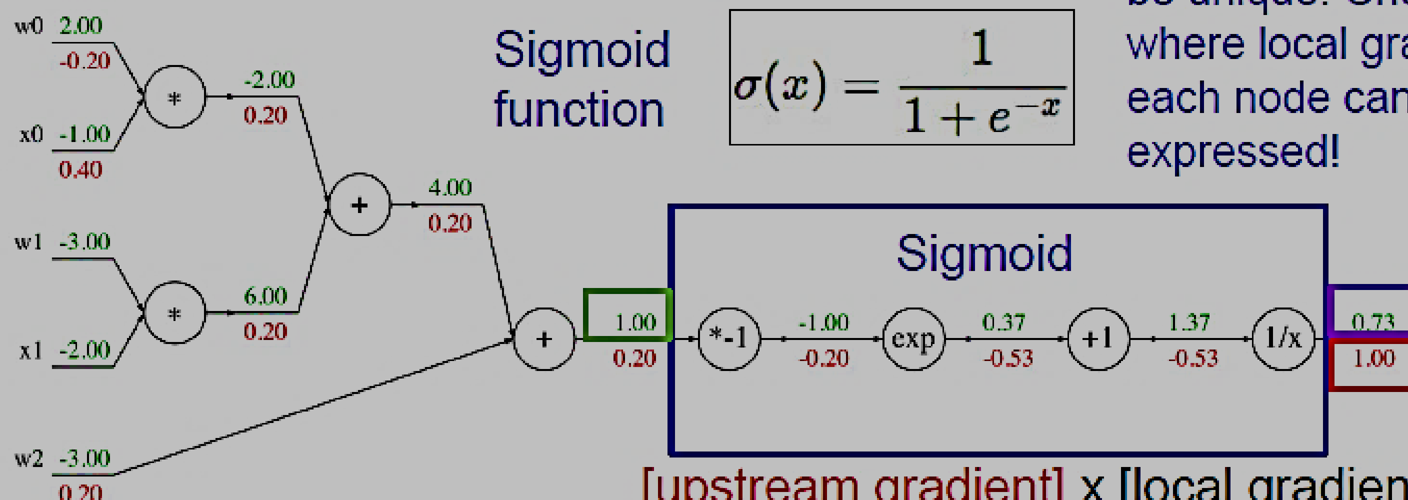
# Backpropagation: Another Example

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

Computational graph representation may not be unique. Choose one where local gradients at each node can be easily expressed!

Sigmoid function

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




[upstream gradient] x [local gradient]

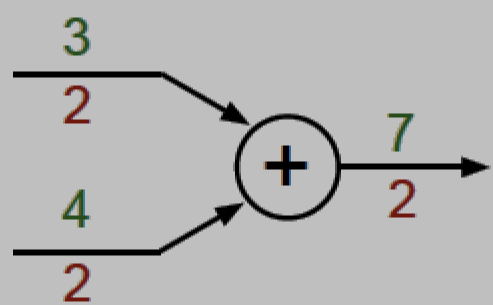
[1.00] x [(1 - 0.73) (0.73)] = 0.2

Sigmoid local gradient:

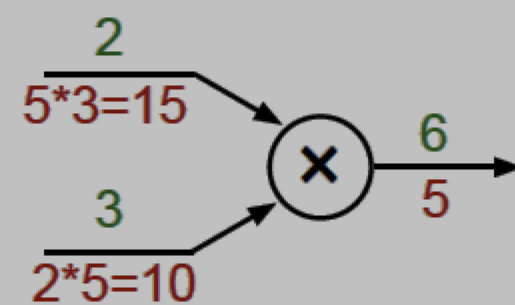
$$\frac{d\sigma(x)}{dx} = \frac{e^{-x}}{(1+e^{-x})^2} = \left(\frac{1+e^{-x}-1}{1+e^{-x}}\right)\left(\frac{1}{1+e^{-x}}\right) = (1-\sigma(x))\,\sigma(x)$$

# Patterns in Gradient Flow

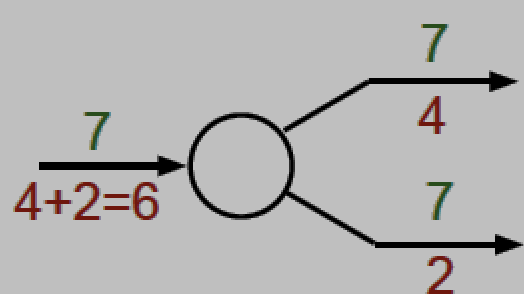
**add** gate: gradient distributor



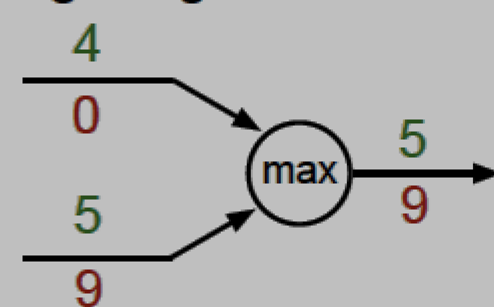

**mul** gate: “swap multiplier”



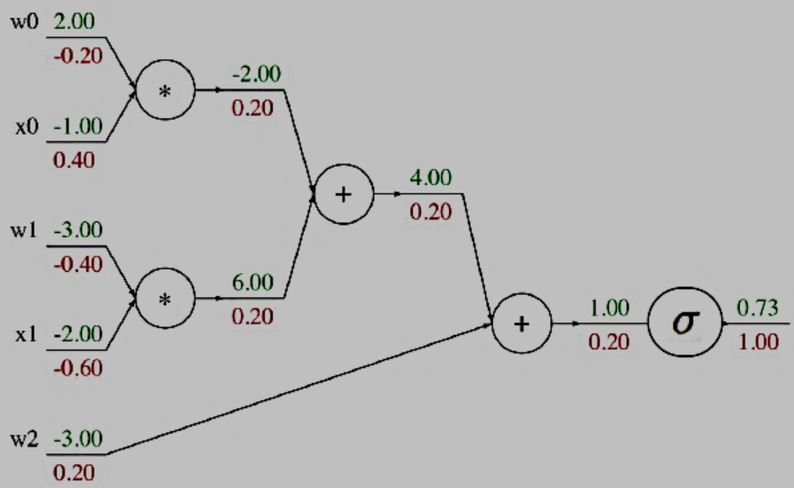

**copy** gate: gradient adder




**max** gate: gradient router



# Backpropagation Implementation




Forward pass:
Compute output

```
def f(w0, x0, w1, x1, w2):
    s0 = w0 * x0
    s1 = w1 * x1
    s2 = s0 + s1
    s3 = s2 + w2
    L = sigmoid(s3)
```

Backward pass:
Compute grads

```
    grad_L = 1.0
    grad_s3 = grad_L * (1 - L) * L
    grad_w2 = grad_s3
    grad_s2 = grad_s3
    grad_s0 = grad_s2
    grad_s1 = grad_s2
    grad_w1 = grad_s1 * x1
    grad_x1 = grad_s1 * w1
    grad_w0 = grad_s0 * x0
    grad_x0 = grad_s0 * w0
```

# Neural Networks with Matrices

- L: number of layers in the network
- $S_L$: number of neurons in network layer L without bias term
- $\boldsymbol{W}^{(L)}{}_{s_{(L+1)}\times(s_L+1)}$: the weight matrix in layer L

$$\boldsymbol{W}^{(L)} = \begin{bmatrix} w_{10}^{(L)} & w_{11}^{(L)} & \cdots & w_{1s_L}^{(L)} \\ \vdots & \vdots & \ddots & \vdots \\ w_{s_{(L+1)}0}^{(L)} & w_{s_{(L+1)}1}^{(L)} & \cdots & w_{s_{(L+1)}s_L}^{(L)} \end{bmatrix}_{s_{(L+1)}\times(s_L+1)}$$

- Training dataset: $\{(\boldsymbol{x}^{(1)},\boldsymbol{y}^{(1)}),(\boldsymbol{x}^{(2)},\boldsymbol{y}^{(2)}),\dots,(\boldsymbol{x}^{(m)},\boldsymbol{y}^{(m)})\}$

$$\boldsymbol{x}^{(i)} = \left\{x_1^{(i)},x_2^{(i)},\dots,x_N^{(i)}\right\}$$

# Neural Networks with Matrices

- For a single input to layer L+1:

$$\boldsymbol{z}^{(L+1)}{}_{s_{(L+1)}\times 1} = \boldsymbol{W}^{(L)}{}_{s_{(L+1)}\times(s_L+1)} \times \begin{bmatrix} a_0^{(L)} \\ \boldsymbol{a}^{(L)} \end{bmatrix}_{(s_L+1)\times 1}$$

- For multiple inputs to layer L+1:

$$\boldsymbol{z}^{(L+1)}{}_{s_{(L+1)}\times m} = \boldsymbol{W}^{(L)}{}_{s_{(L+1)}\times(s_L+1)} \times \begin{bmatrix} a_0^{(L)} \\ \boldsymbol{a}^{(L)} \end{bmatrix}_{(s_L+1)\times m}$$

$$\boldsymbol{a}^{(L+1)} = \mathrm{g}(\boldsymbol{z}^{(L+1)})$$

# Neural Networks with Matrices

- For binary classification output layer $s_{L=output}$ has
  - either a single neuron $\boldsymbol{y}_{\boldsymbol{out}}^{(i)}$=0 → $\boldsymbol{y}^{(i)} = 0$ or $\boldsymbol{y}_{\boldsymbol{out}}^{(i)}$=1 → $\boldsymbol{y}^{(i)} = 1$
  - or two neurons $\boldsymbol{y}_{\boldsymbol{out}}^{(i)}$=[10] → $\boldsymbol{y}^{(i)} = 0$ or $\boldsymbol{y}_{\boldsymbol{out}}^{(i)}$=[01] → $\boldsymbol{y}^{(i)} = 1$

- For multi-class classification output layer $s_{L=output} \in \Re^K$
  - $Seq_{j=1}^{K} \begin{cases} 1 & \text{iff} \quad \mathrm{j = c} \\ 0 & \text{otherwise} \end{cases} \rightarrow \boldsymbol{y}^{(i)} = c$ or
  - $Seq_{j=1}^{K} \begin{cases} 0 & \text{iff} \quad \mathrm{j = c} \\ 1 & \text{otherwise} \end{cases} \rightarrow \boldsymbol{y}^{(i)} = c$
  - For example, if $\boldsymbol{y}^{(i)} = 1$

$$\boldsymbol{y}_{\boldsymbol{out}_{K\times 1}}^{(i)} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ \vdots \end{bmatrix}_{K\times 1} \quad or \quad \boldsymbol{y}_{\boldsymbol{out}_{K\times 1}}^{(i)} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 1 \\ \vdots \end{bmatrix}_{K\times 1}$$

- Given $\left(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)}\right)$ it is checked if $h_W\left(\boldsymbol{x}^{(i)}\right) = \boldsymbol{y}_{\boldsymbol{out}}^{(i)}$

# Neural Networks Cost Function

- The generalized logistic regression cost function $J(W)$ for neural networks:

$$\mathrm{J}(\boldsymbol{W}) = \frac{-1}{m}\left[\sum_{i=1}^{m}\sum_{k=1}^{K} y_{\mathrm{k}}^{(i)} \log\left(h_{\boldsymbol{W}}\left(\boldsymbol{x}^{(i)}\right)_k\right) + (1 - y_{\mathrm{k}}^{(i)}) \log\left(1 - h_{\boldsymbol{W}}\left(\boldsymbol{x}^{(i)}\right)_k\right)\right] + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{S_l}\sum_{j=1}^{S_{l+1}}\left(w_{ji}^{(l)}\right)^2$$

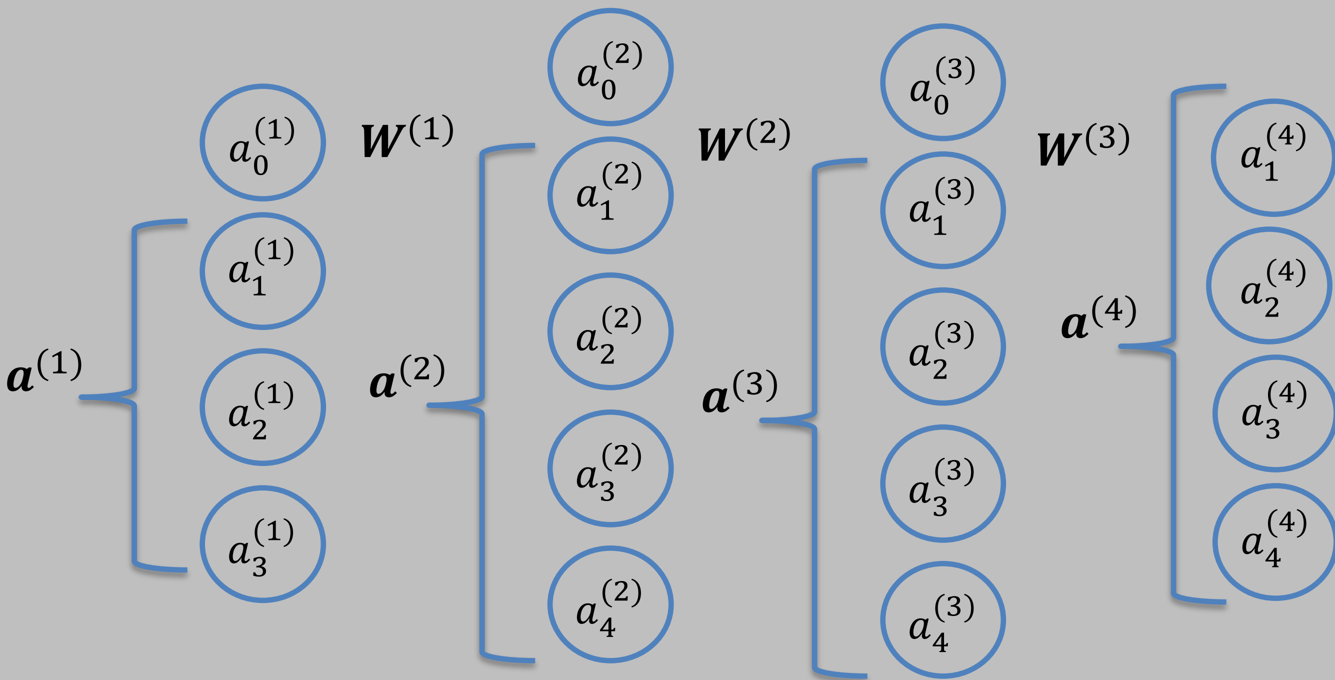
Regularization term

- A simpler version of the cost function is again preferred to reduce complexity.

$$\mathrm{J}(\boldsymbol{W}) = \frac{1}{2m}\left[\sum_{i=1}^{m}\sum_{k=1}^{K}\left(y_{\mathrm{k}}^{(i)} - h_{\boldsymbol{W}}\left(\boldsymbol{x}^{(i)}\right)_k\right)^2\right]$$

# Neural Networks Feedforward

- A 4-layer artificial neural network (ANN)




$$\boldsymbol{z}^{(2)} = \boldsymbol{W}^{(1)} \begin{bmatrix} a_0^{(1)} \\ \boldsymbol{a}^{(1)} \end{bmatrix}$$

$$\boldsymbol{a}^{(2)} = \mathrm{g}(\boldsymbol{z}^{(2)})$$

$$\boldsymbol{z}^{(3)} = \boldsymbol{W}^{(2)} \begin{bmatrix} a_0^{(2)} \\ \boldsymbol{a}^{(2)} \end{bmatrix}$$

$$\boldsymbol{a}^{(3)} = \mathrm{g}(\boldsymbol{z}^{(3)})$$

$$\boldsymbol{z}^{(4)} = \boldsymbol{W}^{(3)} \begin{bmatrix} a_0^{(3)} \\ \boldsymbol{a}^{(3)} \end{bmatrix}$$

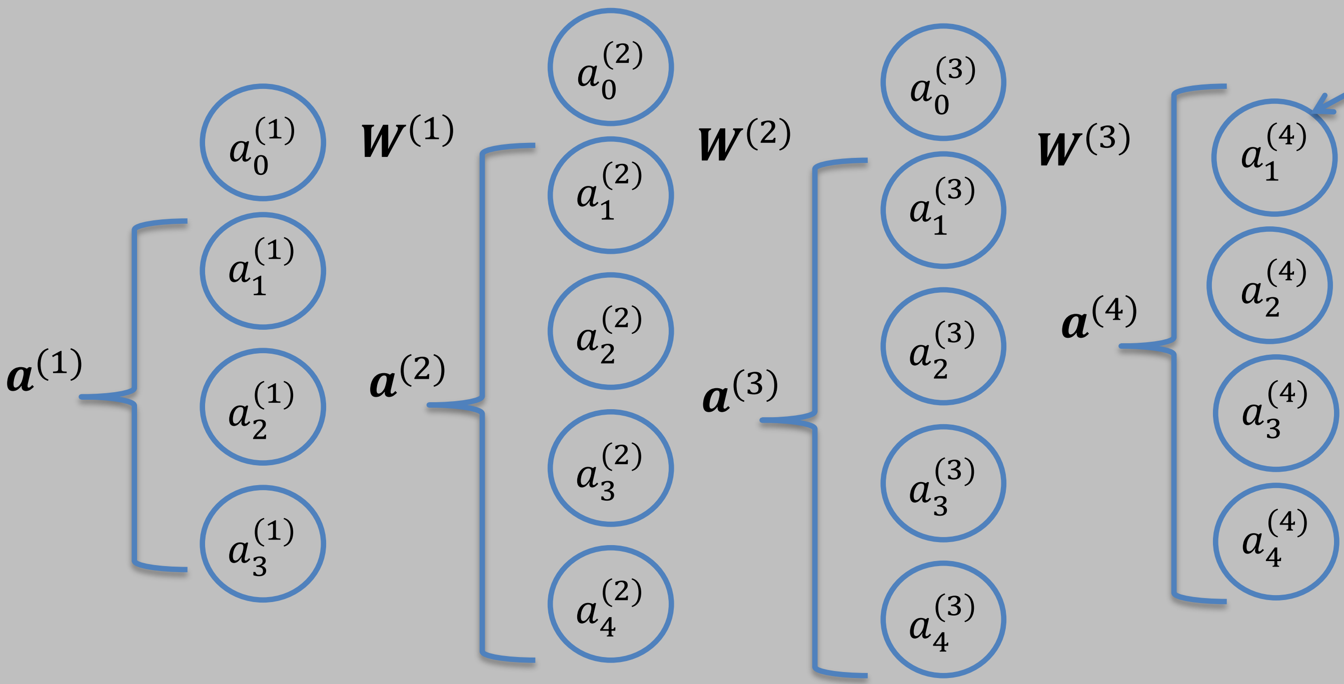
- Given sample $\left(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)}\right)$

$$\boldsymbol{a}^{(1)} = x^{(i)}$$

$$\boldsymbol{a}^{(4)} = \mathrm{g}(\boldsymbol{z}^{(4)})$$

# Neural Networks Backpropagation

- Beginning from the last layer the error of the network is computed.




for each neuron in last layer simple error without activation

$$\delta_j^{(4)} = y_j^{(i)} - a_j^{(4)}$$

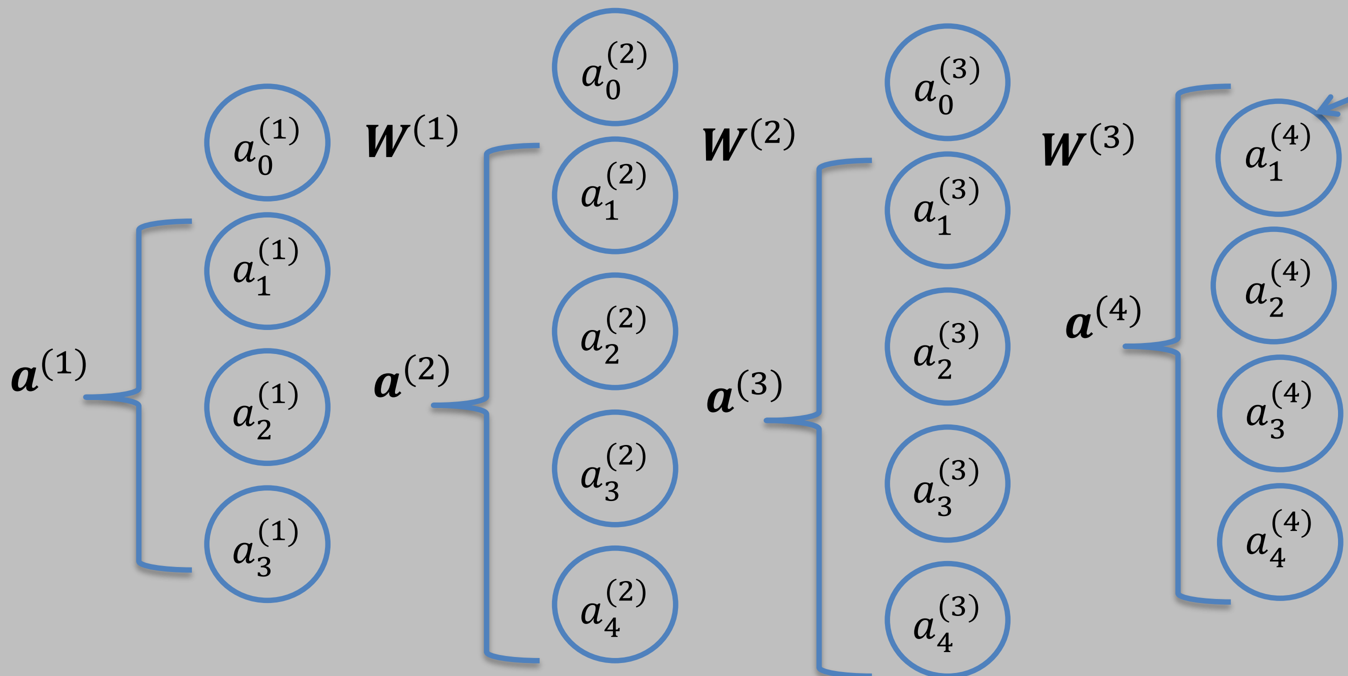
$$\delta_j^{(l)} = \frac{\partial J(\boldsymbol{W})^{(i)}}{\partial z_j^{(l)}}$$

- Given sample $\left(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)}\right)$

$$\boldsymbol{\delta}^{(\mathrm{L})} = \left(\boldsymbol{y}^{(i)} - \mathbf{a}^{(\mathrm{L})}\right).*\, \boldsymbol{g}'\left(\boldsymbol{z}^{(\mathrm{L})}\right) = \left(\boldsymbol{y}^{(i)} - \mathbf{a}^{(\mathrm{L})}\right).*\, \mathbf{a}^{(\mathrm{L})}.*\,(1 - \mathbf{a}^{(\mathrm{L})})$$

# Neural Networks Backpropagation

- Beginning from the last layer the error of the network is computed.




for each neuron in last layer, simple error without activation

$$\delta_j^{(4)} = y_j^{(i)} - a_j^{(4)}$$

$$\delta_j^{(l)} = \frac{\partial J(\boldsymbol{W})^{(i)}}{\partial z_j^{(l)}}$$

$$J(\boldsymbol{W})^{(i)} = \left(y^{(i)} - h_{\boldsymbol{W}}\left(\boldsymbol{x}^{(i)}\right)\right)^2$$

$$= \left(y^{(i)} - \mathbf{g}\left(\boldsymbol{W}^{(l)} \times \begin{bmatrix} a_0^{(l)} \\ \boldsymbol{a}^{(l)} \end{bmatrix}\right)\right)^2$$

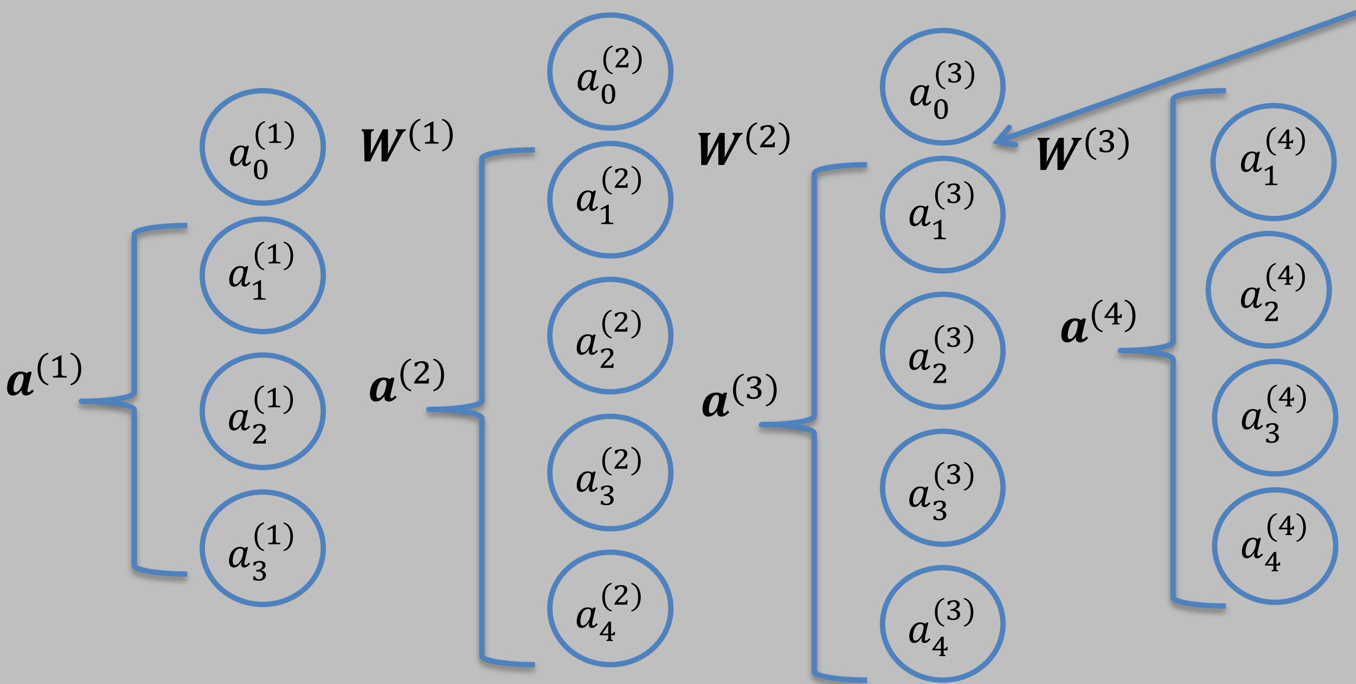
- Given sample $\left(x^{(i)}, y^{(i)}\right)$

Evaluation of derivative of sigmoid function for input $\boldsymbol{z}^{(4)}$ values

$$\boldsymbol{\delta}^{(4)} = \left(\boldsymbol{y}^{(i)} - \mathbf{a}^{(4)}\right) .* \, \mathrm{g}'\left(\boldsymbol{z}^{(4)}\right) = \left(\boldsymbol{y}^{(i)} - \mathbf{a}^{(4)}\right) .* \, \mathbf{a}^{(4)} .* \left(1 - \mathbf{a}^{(4)}\right)$$

# Neural Networks Backpropagation

- Beginning from the last layer the error of the network is computed.




for each neuron in third layer, propagated error

$$\delta_j^{(l)} = \frac{\partial J(\boldsymbol{W})^{(i)}}{\partial z_j^{(l)}}$$

$$J(\boldsymbol{W})^{(i)} = \left(y^{(i)} - h_{\boldsymbol{W}}\left(\boldsymbol{x}^{(i)}\right)\right)^2$$
$$= \left(y^{(i)} - \mathbf{g}\left(\boldsymbol{W}^{(l)} \times \begin{bmatrix} a_0^{(l)} \\ \boldsymbol{a}^{(l)} \end{bmatrix}\right)\right)^2$$

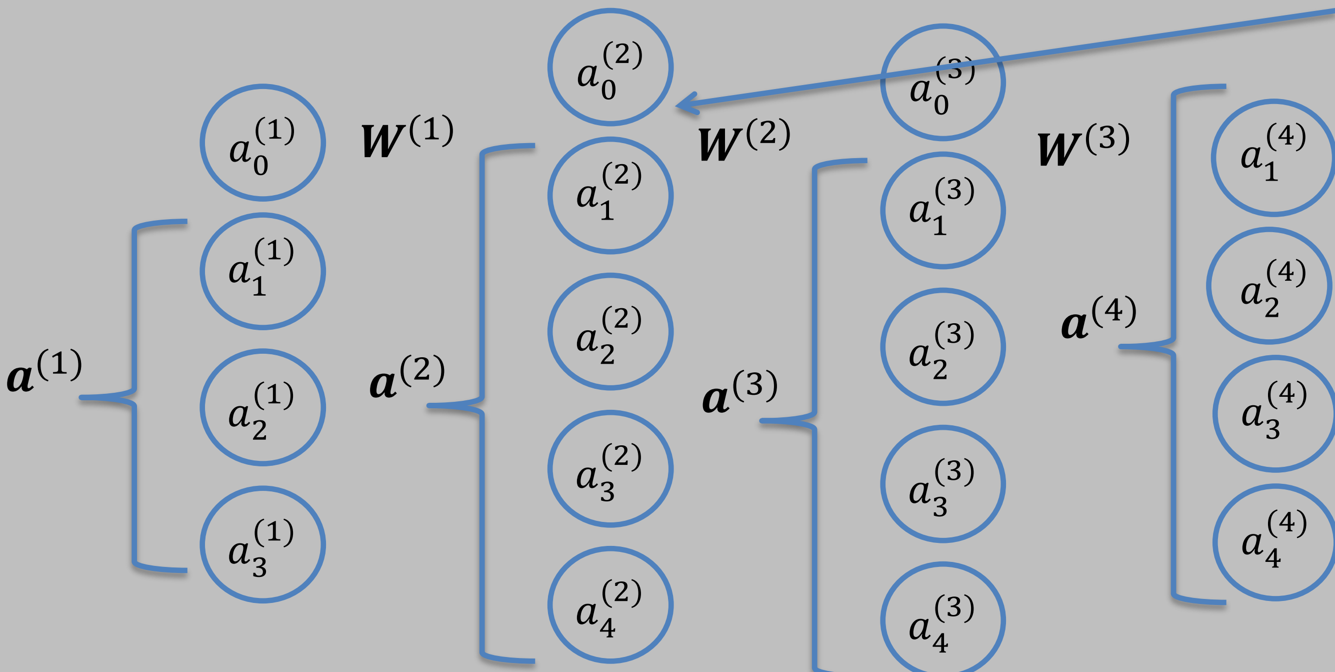
Evaluation of derivative of sigmoid function for input $\boldsymbol{z}^{(3)}$ values

- Given sample $\left(x^{(i)}, y^{(i)}\right)$

$$\boldsymbol{\delta}^{(3)} = \left(\boldsymbol{W}^{(3)}\right)^T .* \boldsymbol{\delta}^{(4)} .* \mathrm{g}'\left(\boldsymbol{z}^{(3)}\right) = \left(\boldsymbol{W}^{(3)}\right)^T .* \boldsymbol{\delta}^{(4)} .* \mathbf{a}^{(3)} .* \left(1 - \mathbf{a}^{(3)}\right)$$

# Neural Networks Backpropagation

- Beginning from the last layer the error of the network is computed.




for each neuron in second layer, propagated error

$$\delta_j^{(l)} = \frac{\partial J(\boldsymbol{W})^{(i)}}{\partial z_j^{(l)}}$$

$$J(\boldsymbol{W})^{(i)} = \left(y^{(i)} - h_{\boldsymbol{W}}\left(\boldsymbol{x}^{(i)}\right)\right)^2$$
$$= \left(y^{(i)} - \mathbf{g}\left(\boldsymbol{W}^{(l)} \times \begin{bmatrix} a_0^{(l)} \\ \boldsymbol{a}^{(l)} \end{bmatrix}\right)\right)^2$$

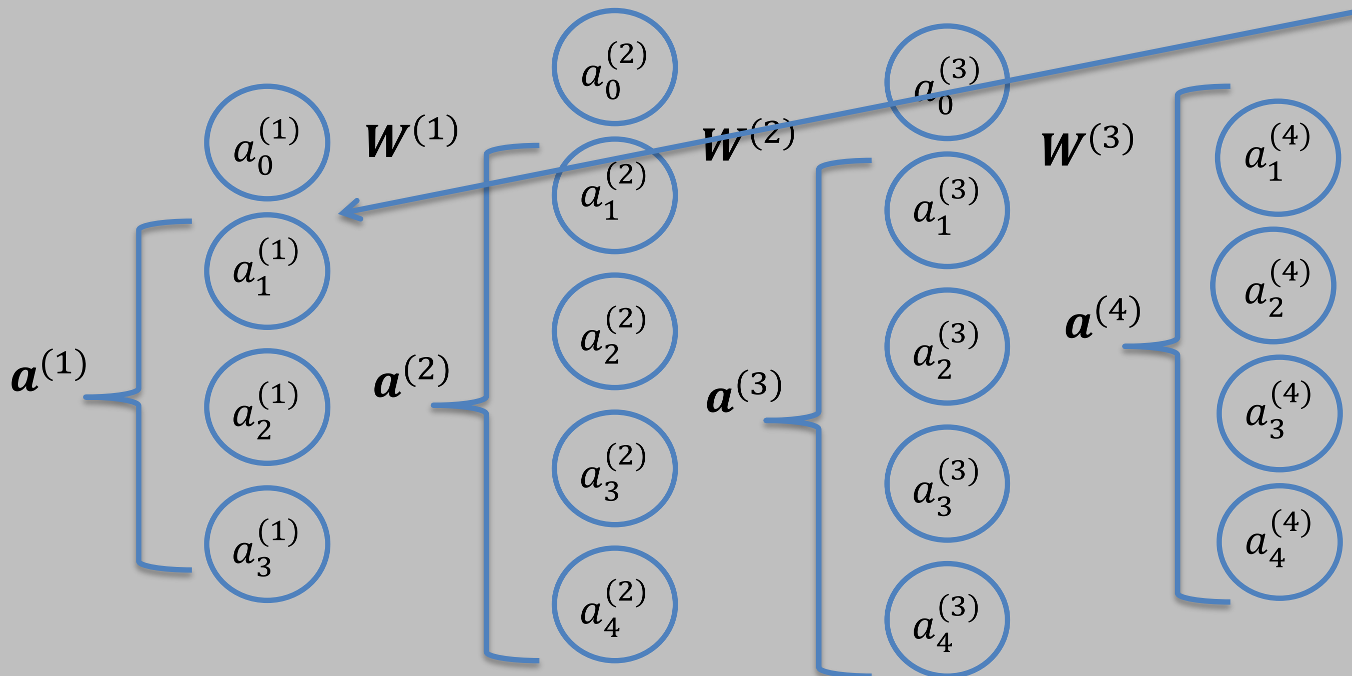
Evaluation of derivative of sigmoid function for input $\boldsymbol{z}^{(2)}$ values

- Given sample $\left(x^{(i)}, y^{(i)}\right)$

$$\boldsymbol{\delta}^{(2)} = \left(\boldsymbol{W}^{(2)}\right)^T .* \boldsymbol{\delta}^{(3)} .* \mathrm{g}'\left(\boldsymbol{z}^{(2)}\right) = \left(\boldsymbol{W}^{(2)}\right)^T .* \boldsymbol{\delta}^{(3)} .* \mathbf{a}^{(2)} .* \left(1 - \mathbf{a}^{(2)}\right)$$

# Neural Networks Backpropagation

- Beginning from the last layer the error of the network is computed.




for each neuron in first layer, propagated error

*not calculated*

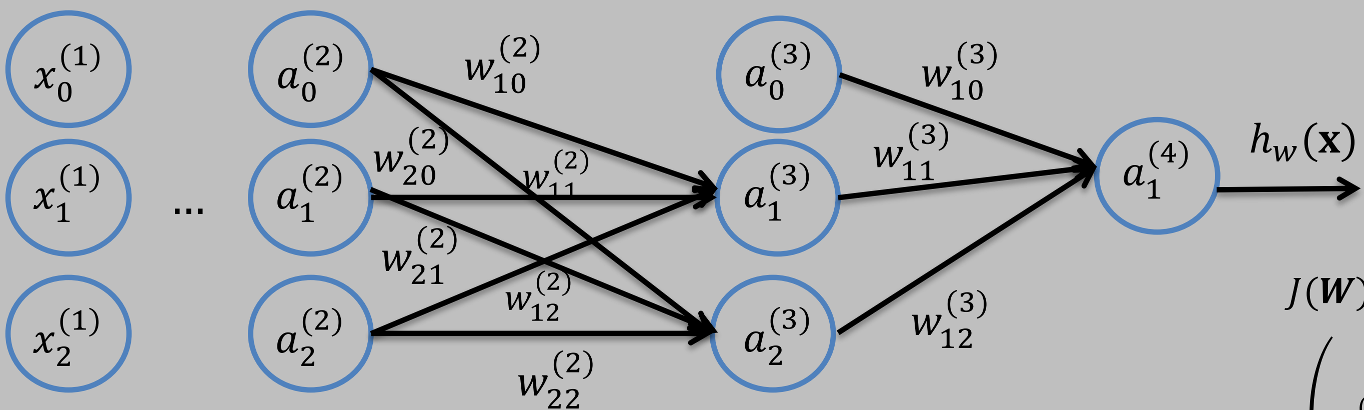
$$J(\boldsymbol{W})^{(i)} = \left(y^{(i)} - h_W\left(\boldsymbol{x}^{(i)}\right)\right)^2 = \left(y^{(i)} - \mathbf{g}\left(\boldsymbol{W}^{(l)} \times \begin{bmatrix} a_0^{(l)} \\ \boldsymbol{a}^{(l)} \end{bmatrix}\right)\right)^2$$

- Given sample $\left(x^{(i)}, y^{(i)}\right)$

$\boldsymbol{\delta}^{(1)} \rightarrow$ Not calculated because for input layer there is no error.

# Backpropagation Example

- For simplicity it is assumed that no activation function is used for any neuron!




$$\delta_j^{(l)} = \frac{\partial J(\boldsymbol{W})^{(i)}}{\partial z_j^{(l)}}$$

$$J(\boldsymbol{W})^{(i)} = \left(y^{(i)} - h_{\boldsymbol{W}}\left(\boldsymbol{x}^{(i)}\right)\right)^2$$

$$= \left(y^{(i)} - \mathbf{g}\left(\boldsymbol{W}^{(l)} \times \begin{bmatrix} a_0^{(l)} \\ \boldsymbol{a}^{(l)} \end{bmatrix}\right)\right)^2$$

$$\delta_1^{(4)} = y_1^{(i)} - a_1^{(4)}$$

$$\delta_1^{(3)} = w_{11}^{(3)} * \delta_1^{(4)}$$

$$\delta_2^{(3)} = w_{12}^{(3)} * \delta_1^{(4)}$$

$$\delta_1^{(2)} = w_{11}^{(2)} * \delta_1^{(3)} + w_{21}^{(2)} * \delta_2^{(3)}$$

$$\delta_2^{(2)} = w_{12}^{(2)} * \delta_1^{(3)} + w_{22}^{(2)} * \delta_2^{(3)}$$

# Neural Networks Weight Update

- Update of the weights without regularization term
  - Given sample $\left(x^{(i)}, y^{(i)}\right)$
  - $\frac{\partial J(\boldsymbol{W})}{\partial W_{ij}^{(l)}} = a_j^{(l)} . \delta_i^{(l+1)}$

Output of neuron j in layer $(\mathrm{l})$, input to layer $(\mathrm{l}+1)$ after multiplied by $W_{ij}^{(l)}$

Error for neuron i in layer $(\mathrm{l}+1)$

  - $W_{ij}^{(l)} \coloneqq W_{ij}^{(l)} + \Delta_{ij}^{(l)}$ where $\Delta_{ij}^{(l)}$ is the change in weight
  - $\Delta_{ij}^{(l)} := \eta \Delta_{ij}^{(l)} + \alpha \frac{\partial J(\boldsymbol{W})}{\partial W_{ij}^{(l)}} = \eta \Delta_{ij}^{(l)} + \alpha \left( a_j^{(l)} . \delta_i^{(l+1)} \right)$
  - This formula without regularization is used for the bias term $\Delta_{i0}^{(l)}$

# Neural Networks Weight Update

- Update of the weights with regularization term
  - Given sample $\left(x^{(i)}, y^{(i)}\right)$
  - $\frac{\partial J(\boldsymbol{W})}{\partial W_{ij}^{(l)}} = a_j^{(l)} . \delta_i^{(l+1)}$
  - $W_{ij}^{(l)} \coloneqq W_{ij}^{(l)} + \Delta_{ij}^{(l)}$ where $\Delta_{ij}^{(l)}$ is the change in weight
  - $\Delta_{ij}^{(l)} := \eta\Delta_{ij}^{(l)} + \alpha \frac{\partial J(\boldsymbol{W})}{\partial W_{ij}^{(l)}} = \eta\Delta_{ij}^{(l)} + \alpha \left(a_j^{(l)} . \delta_i^{(l+1)} + \lambda W_{ij}^{(l)}\right)$
  - This formula with regularization is used for the terms $\Delta_{ij}^{(l)}$ where $j \neq 0$

# Neural Networks Weight Update

- Given m samples $\left(x^{(i)}, y^{(i)}\right),\ i = 1..m$
- Incremental mode for update of the weights
  - Delta (error) for each sample is used update weights.
  - Weights are updated m times for each epoch.
  - $W_{ij}^{(l)} \coloneqq W_{ij}^{(l)} + \Delta_{ij}^{(l)}$
- Batch mode for update of the weights
  - Delta (error) for each sample is summed up to update weights.
  - Weights are updated m/batch times for each epoch.
  - $W_{ij}^{(l)} \coloneqq W_{ij}^{(l)} + \sum_{i=1}^{batch} \Delta_{ij}^{(l)}$

# Neural Networks Implementational Details

- To break the network symmetry initially $W_{ij}^{(l)}$ values must be small random numbers, e.g., in between [-0.5, 0.5]
- The number of neurons in input layer is the same as the number of features
- The number of neurons in output layer is the same as the number of classes
- Only one hidden layer is generally sufficient
  - More neurons in a hidden layer means more possibility to learn
- If more than one hidden layers are used, the number of neurons should be the same in each hidden layer

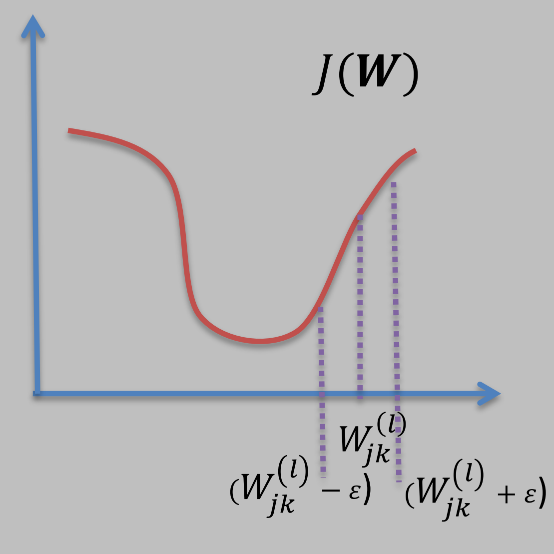
# Neural Networks Implementational Details

- How to understand whether the neural network learns?
  - During learning measure derivative of cost function $J(\boldsymbol{W})$ with respect to the parameters
  - Given $\varepsilon \cong 10^{-4}$

$$\frac{\partial J(\boldsymbol{W})}{\partial W_{jk}^{(l)}} \approx J\left(W_{10}^{(1)}, W_{11}^{(1)}, \dots, W_{jk}^{(l)} + \varepsilon, W_{53}^{(4)}, \dots\right) - J\left(W_{10}^{(1)}, W_{11}^{(1)}, \dots, W_{jk}^{(l)}, W_{53}^{(4)}, \dots\right)$$

  - If for all $W_{jk}^{(l)}$, $\frac{\partial J(\boldsymbol{W})}{\partial W_{jk}^{(l)}} \approx 0$

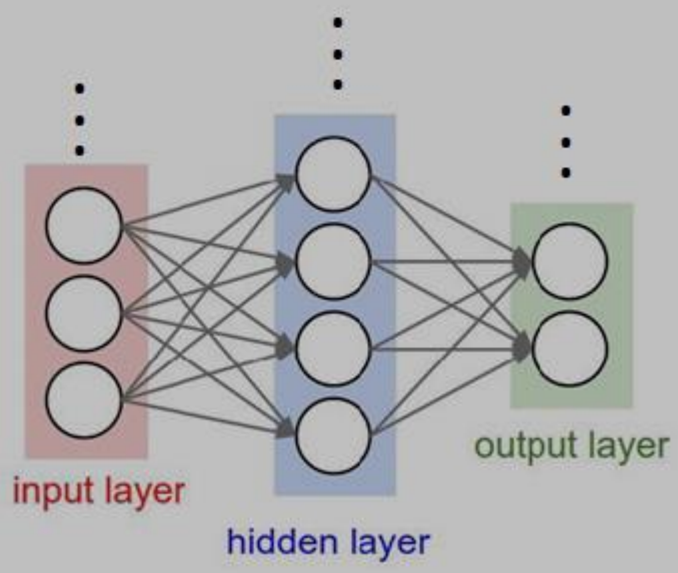
    this means that the system has converged.



# Training of Neural Network with Backpropagation

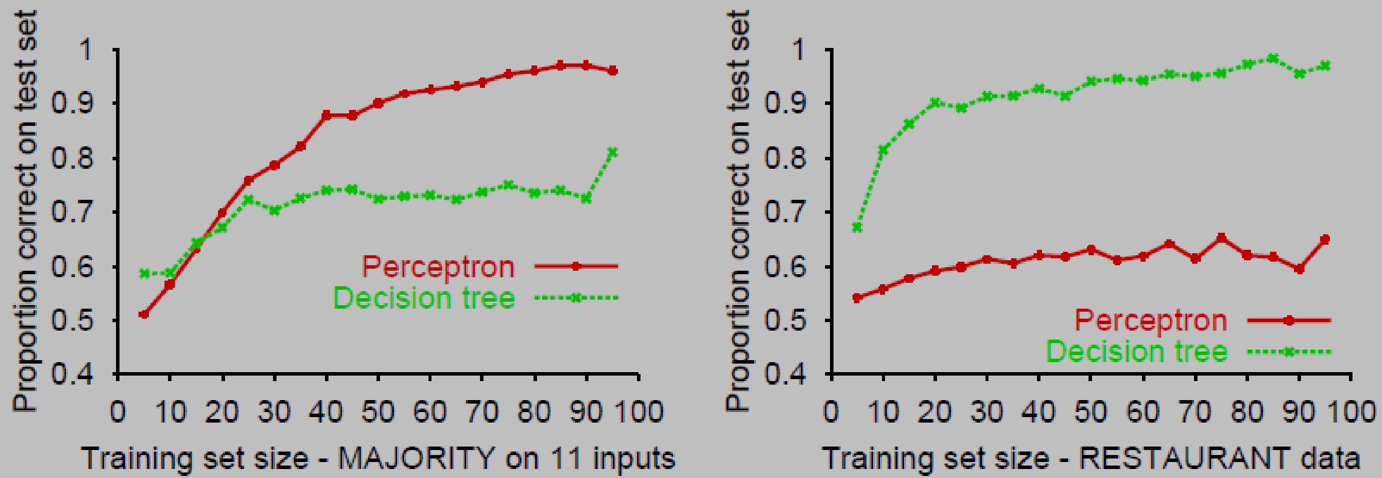
- N: number of samples
- D_in: number of input features
- H: number of neurons in hidden layer
- D_out: number of output classes

```python
import numpy as np
from numpy.random import randn

N, D_in, H, D_out = 64, 1000, 100, 10
x, y = randn(N, D_in), randn(N, D_out)
w1, w2 = randn(D_in, H), randn(H, D_out)

for t in range(2000):
  h = 1 / (1 + np.exp(-x.dot(w1)))
  y_pred = h.dot(w2)
  loss = np.square(y_pred - y).sum()
  print(t, loss)

  grad_y_pred = 2.0 * (y_pred - y)
  grad_w2 = h.T.dot(grad_y_pred)
  grad_h = grad_y_pred.dot(w2.T)
  grad_w1 = x.T.dot(grad_h * h * (1 - h))

  w1 -= 1e-4 * grad_w1
  w2 -= 1e-4 * grad_w2
```



# Perceptron Learning

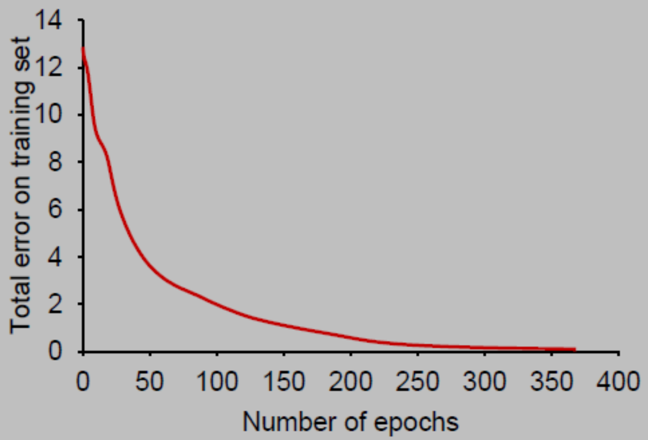
- Perceptron learning rule converges to a consistent function for any linearly separable data set




- Perceptron learns majority function easily, DTL is hopeless
- DTL learns restaurant function easily, perceptron cannot represent it

# Backpropagation Learning

- At each epoch, sum gradient updates for all examples and apply for batch mode
- Training curve for 100 restaurant examples:
  - finds exact fit



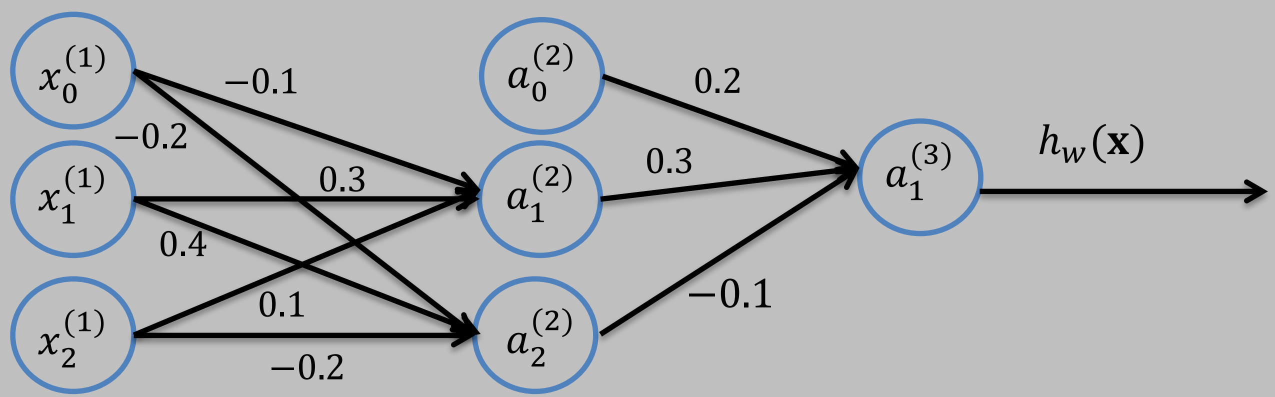

- Typical problems: slow convergence, getting stuck at local minima

# Backpropagation Learning

- Learning curve for MLP with 4 hidden units:

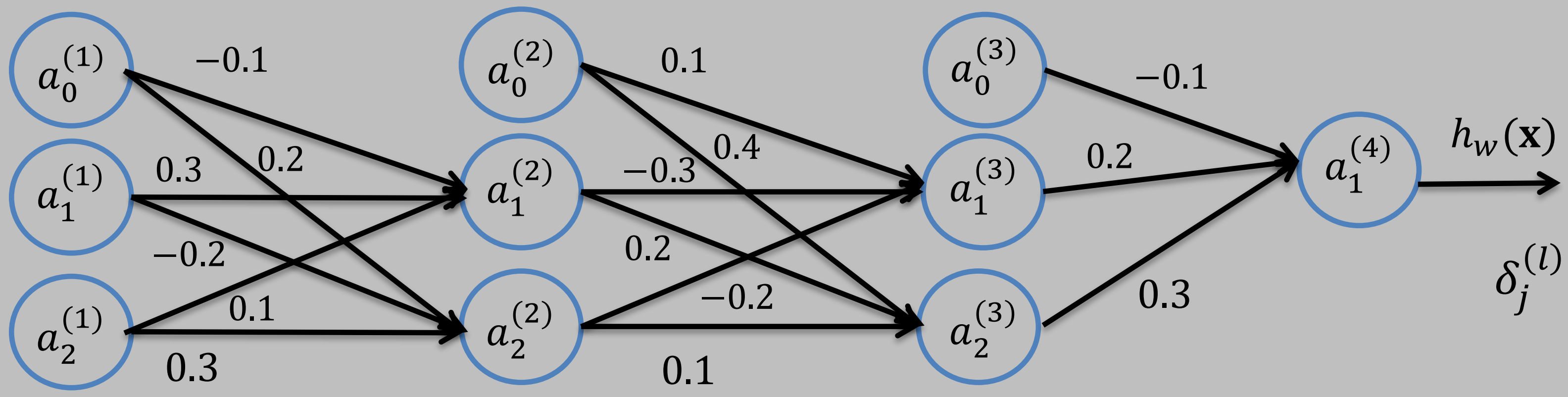

- MLPs are quite good for complex pattern recognition tasks, but resulting hypotheses cannot be understood easily

# Example I




- Neural network modelling XNOR function.
- Activation function is sigmoid function.
- Feed forward for input (1, 1) and compute $a_1^{(3)}$
- Back propagate the final error for input (1,1) and compute $\delta_1^{(2)}$

# Example II




$$\delta_j^{(l)} = \frac{\partial J(\boldsymbol{W})^{(i)}}{\partial z_j^{(l)}}$$

$$J(\boldsymbol{W})^{(i)} = \left(y^{(i)} - h_{\boldsymbol{W}}\left(\boldsymbol{x}^{(i)}\right)\right)^2$$

$$= \left(y^{(i)} - \mathbf{g}\left(\boldsymbol{W}^{(l)} \times \begin{bmatrix} a_0^{(l)} \\ \boldsymbol{a}^{(l)} \end{bmatrix}\right)\right)^2$$

$$\delta_1^{(4)} = y_1^{(i)} - a_1^{(4)}$$

$$\delta_1^{(3)} = w_{11}^{(3)} * \delta_1^{(4)}$$

$$\delta_2^{(3)} = w_{12}^{(3)} * \delta_1^{(4)}$$

$$\delta_1^{(2)} = w_{11}^{(2)} * \delta_1^{(3)} + w_{21}^{(2)} * \delta_2^{(3)}$$

$$\delta_2^{(2)} = w_{12}^{(2)} * \delta_1^{(3)} + w_{22}^{(2)} * \delta_2^{(3)}$$