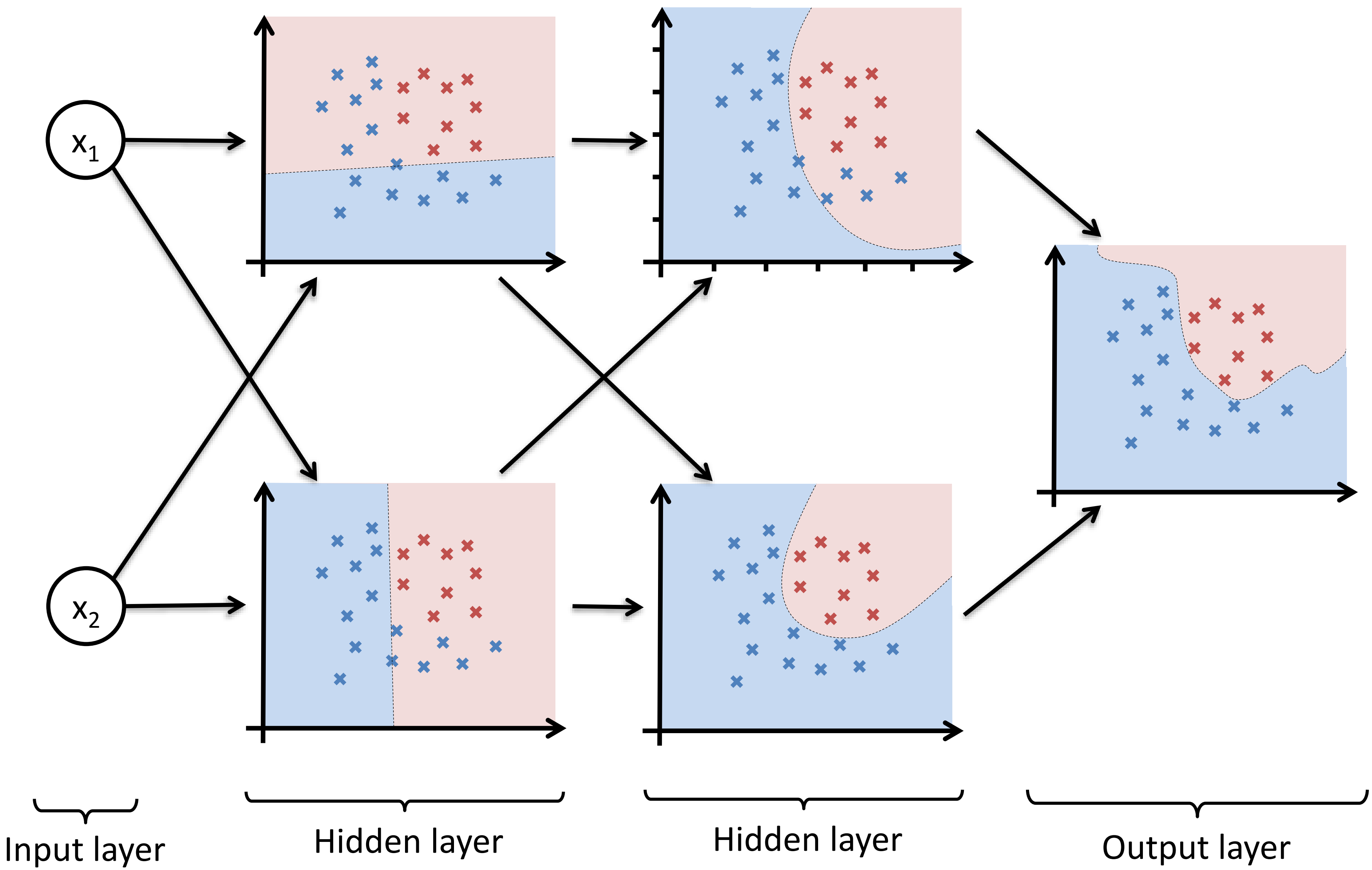
Neural Networks and Deep Learning

MLP & Backpropagation

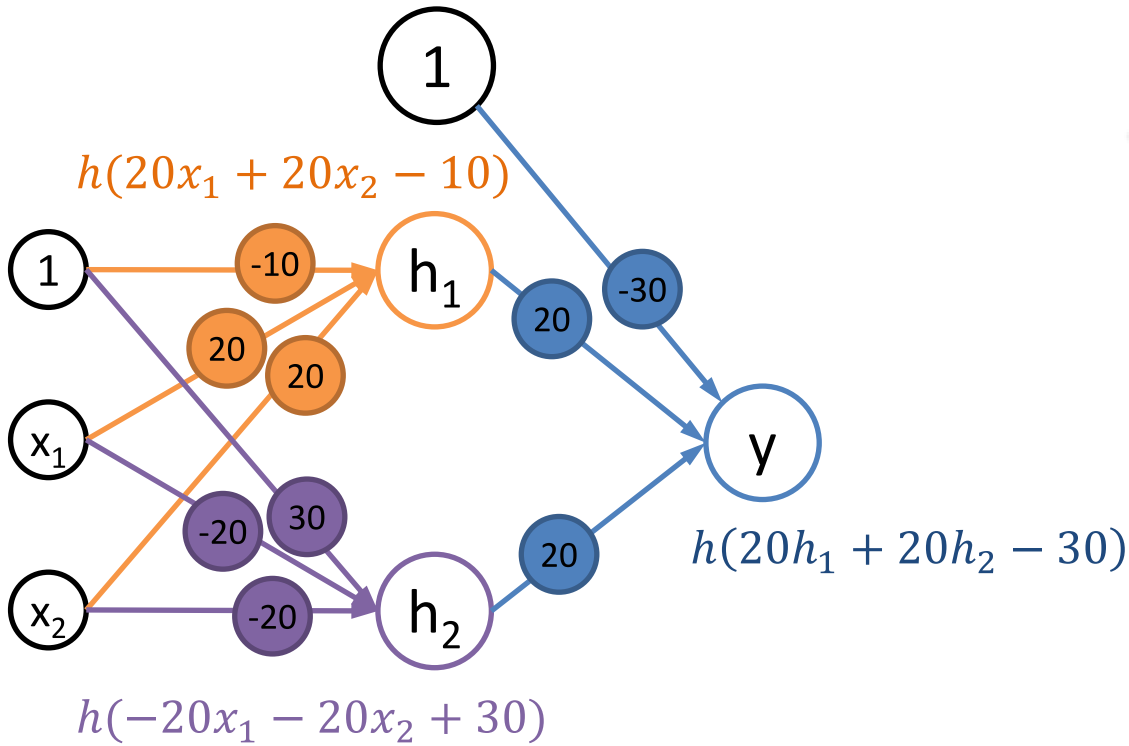
Credit to Dimosthenis Karatzas

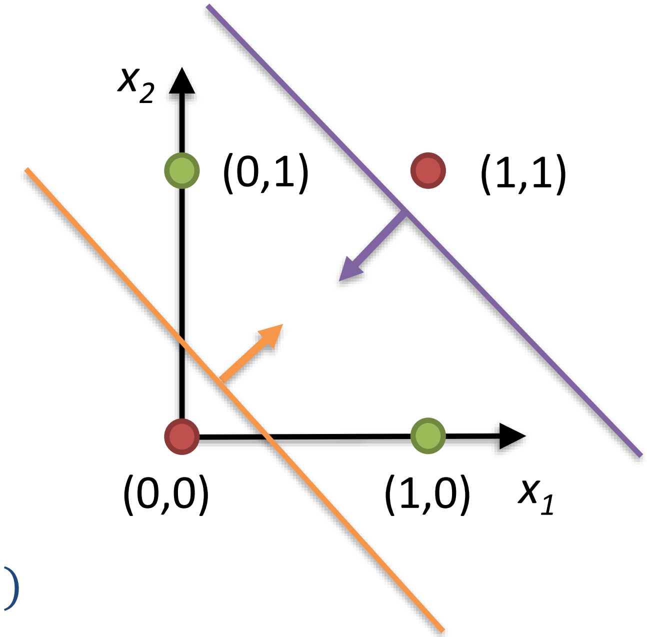
BUILDING MORE COMPLEX NETWORKS

Deep Neural Network



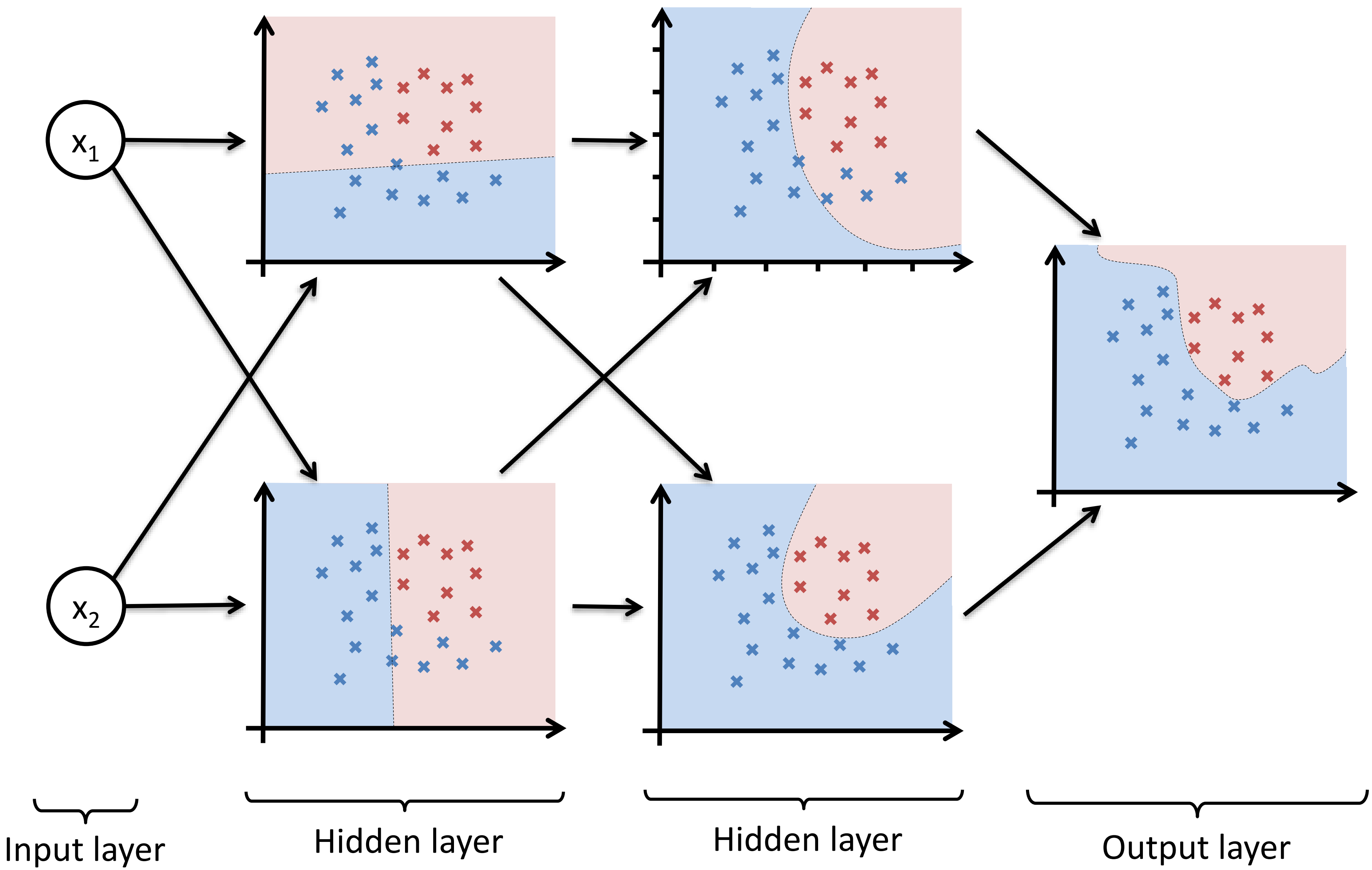
Solving the XOR problem - revisited



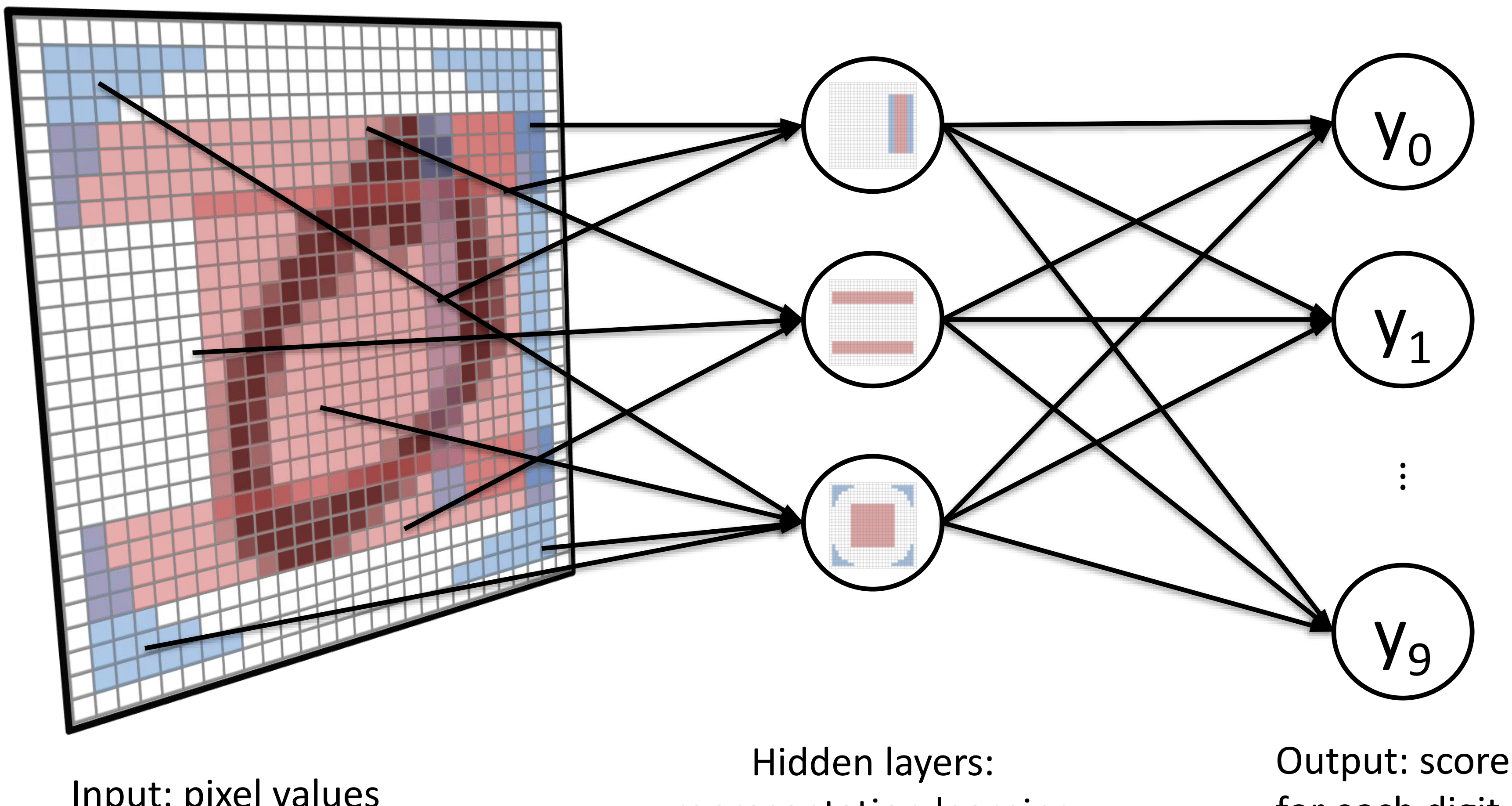


OR	**NAND**	**AND**
$\sigma(20 * 0 + 20 * 0 - 10) = 0$	$\sigma(-20 * 0 - 20 * 0 + 30) = 1$	$\sigma(20 * 0 + 20 * 1 - 30) = 0$
$\sigma(20 * 1 + 20 * 1 - 10) = 1$	$\sigma(-20 * 1 - 20 * 1 + 30) = 0$	$\sigma(20 * 1 + 20 * 0 - 30) = 0$
$\sigma(20 * 0 + 20 * 1 - 10) = 1$	$\sigma(-20 * 0 - 20 * 1 + 30) = 1$	$\sigma(20 * 1 + 20 * 1 - 30) = 1$
$\sigma(20 * 1 + 20 * 0 - 10) = 1$	$\sigma(-20 * 1 - 20 * 0 + 30) = 1$	$\sigma(20 * 1 + 20 * 1 - 30) = 1$

What do hidden layers do?

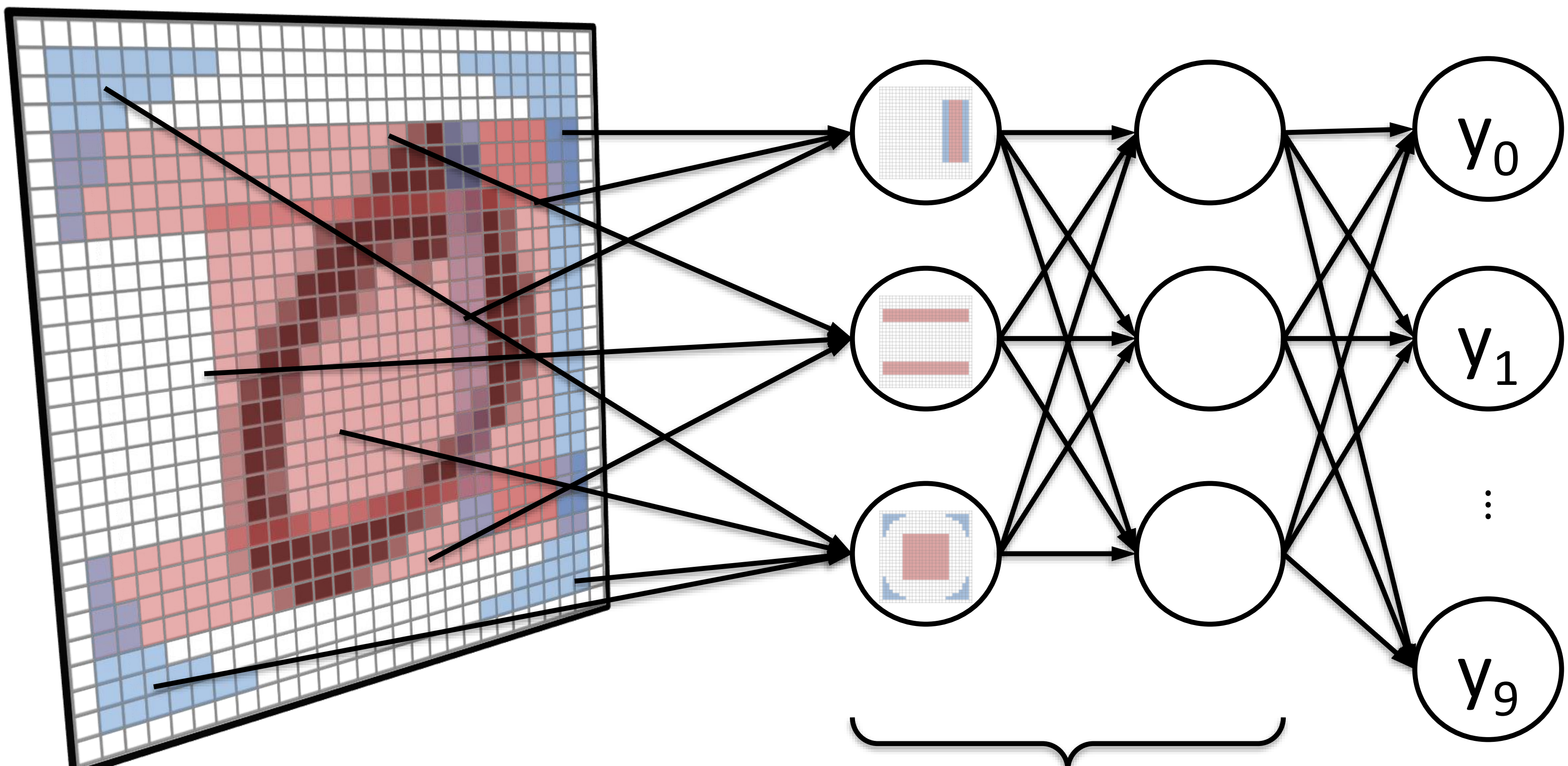


An architecture with hidden units



An architecture with hidden units

Initial hidden layers would give you low-level information, adding subsequent hidden layers the system can encode higher-level features



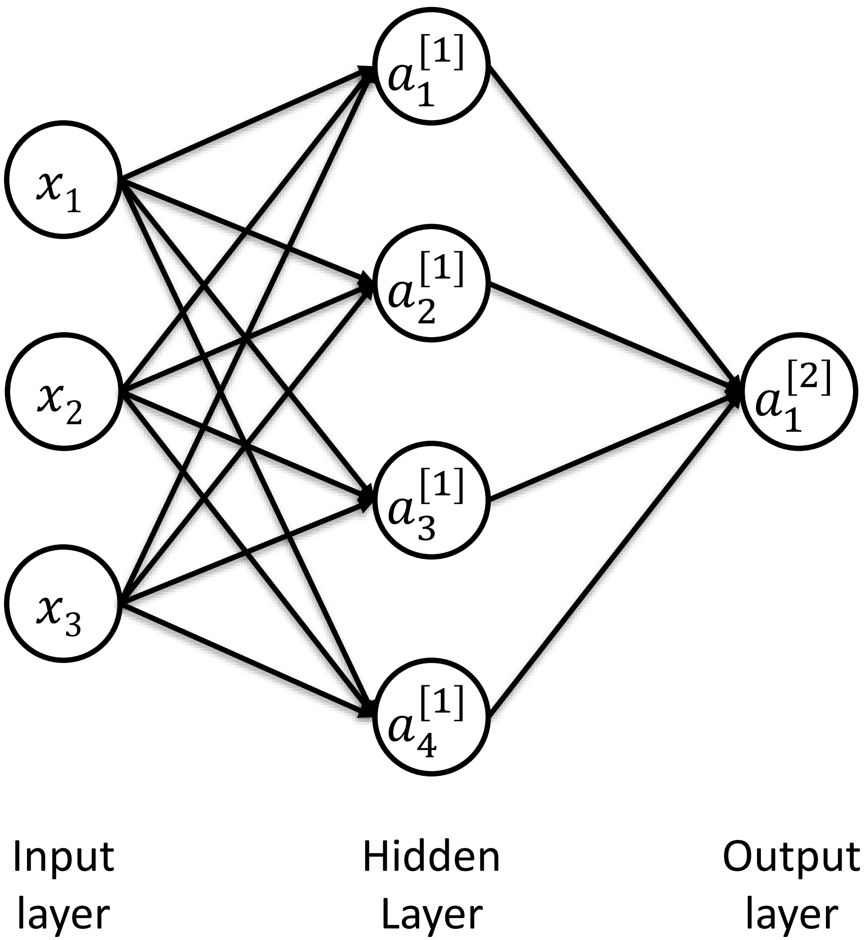
Input: pixel values

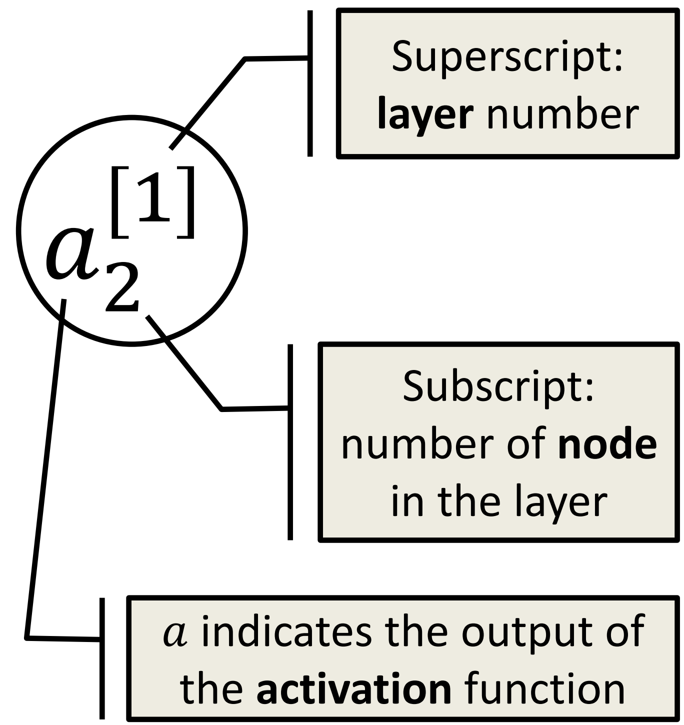
Hidden layers: representation learning

Output: score for each digit

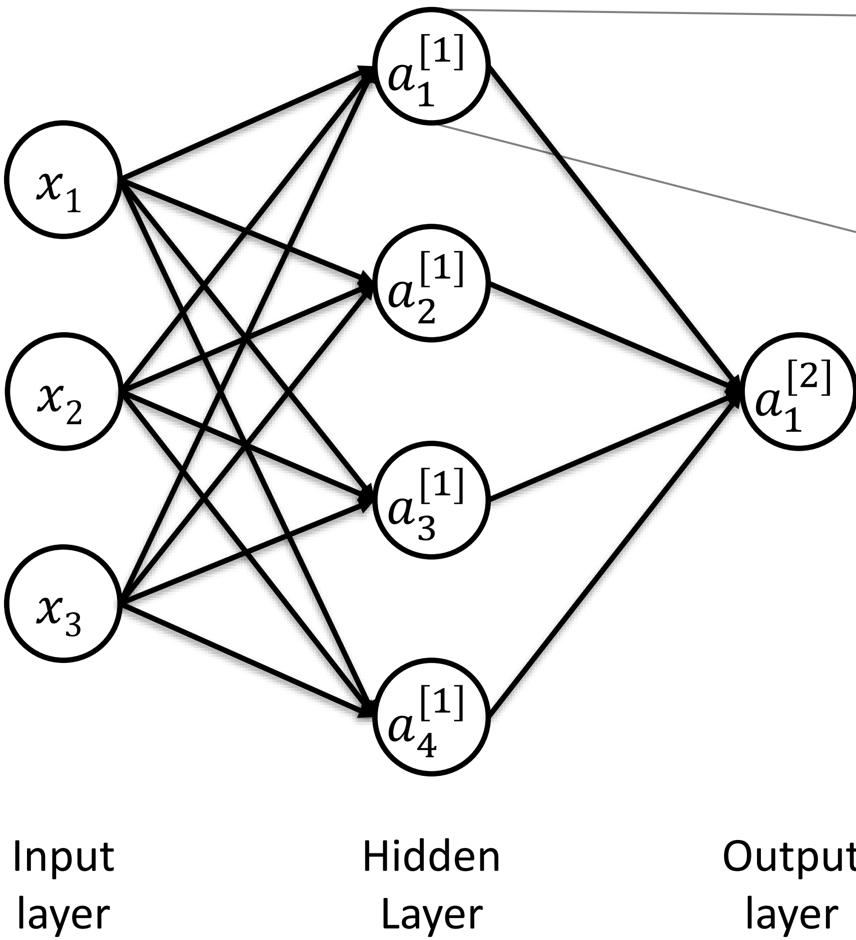
NEURAL NETWORKS NOTATION

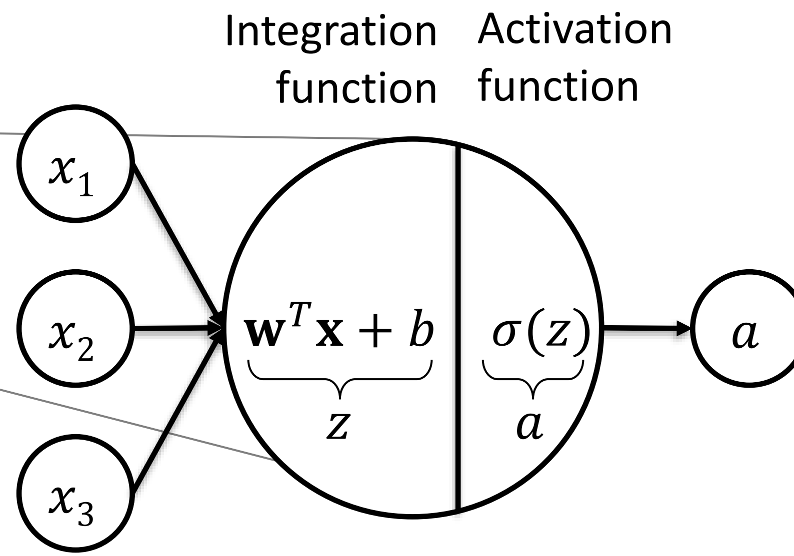
Neural Network Notation





Neural Network Notation

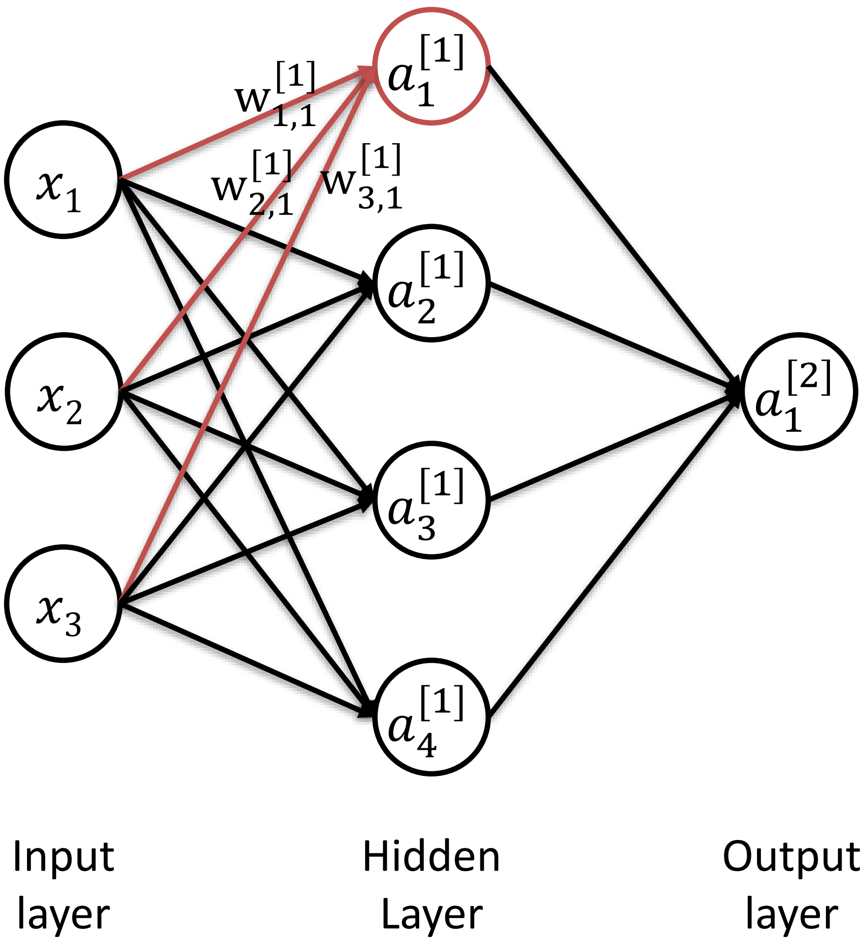




$$z = w^T x + b$$

$$a = \sigma(z)$$

Neural Network Notation

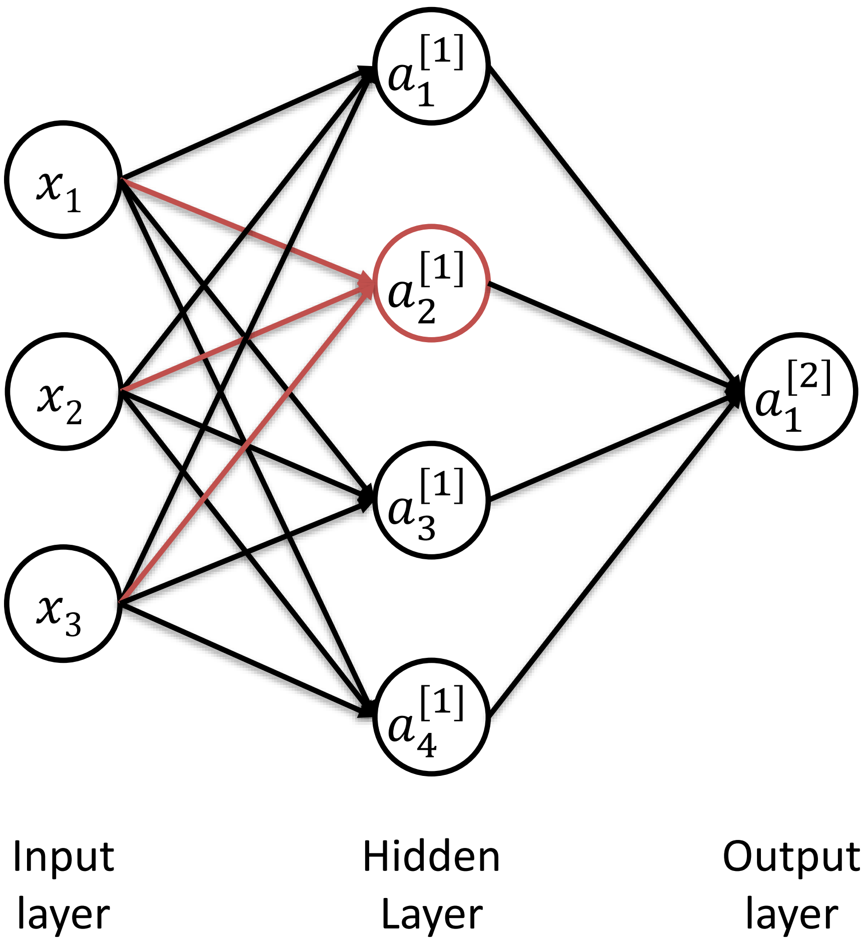


$$z_1^{[1]} = \mathrm{w}_{1,1}^{[1]} x_1 + \mathrm{w}_{1,2}^{[1]} x_2 + \mathrm{w}_{1,3}^{[1]} x_3 + b_1^{[1]}$$

$$z_1^{[1]} = \mathbf{w}_1^{[1]T} \mathbf{x} + b_1^{[1]}$$

$$a_1^{[1]} = \sigma\left(z_1^{[1]}\right)$$

Neural Network Notation

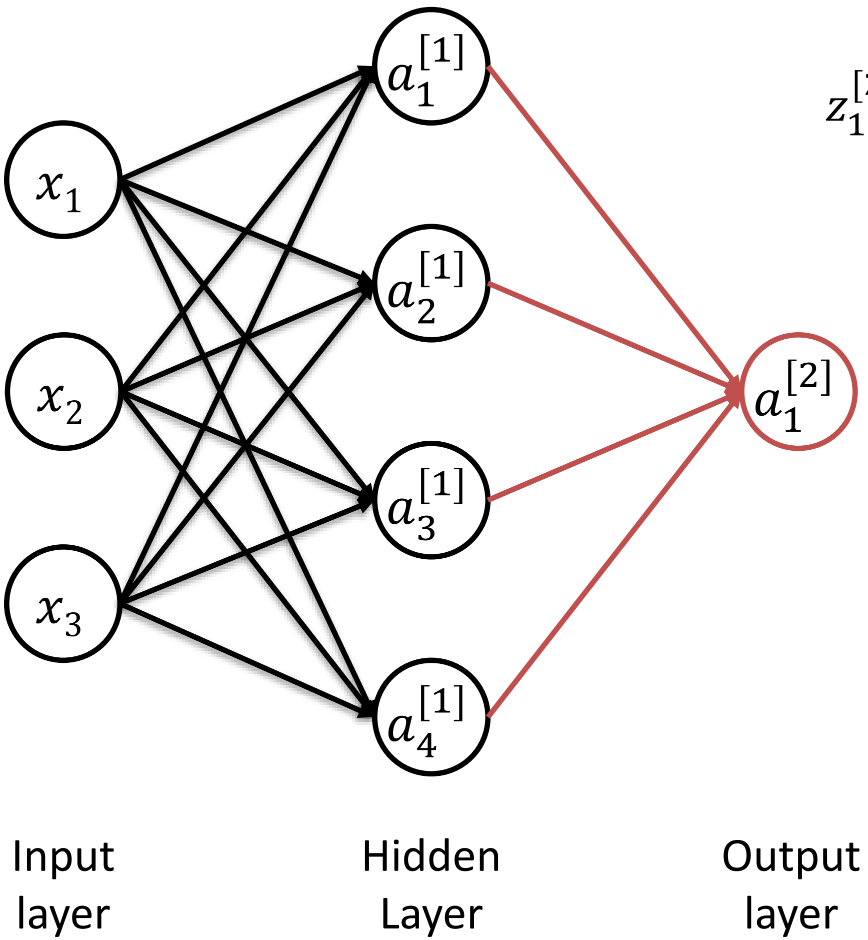


$$z_2^{[1]} = \mathrm{w}_{2,1}^{[1]} x_1 + \mathrm{w}_{2,2}^{[1]} x_2 + \mathrm{w}_{2,3}^{[1]} x_3 + b_2^{[1]}$$

$$z_2^{[1]} = \mathbf{w}_2^{[1]T} \mathbf{x} + b_2^{[1]}$$

$$a_2^{[1]} = \sigma\left(z_2^{[1]}\right)$$

Neural Network Notation

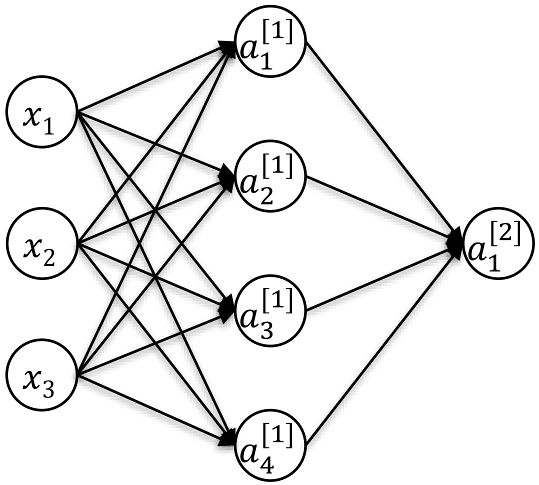


$$z_1^{[2]} = \mathrm{w}_{1,1}^{[2]} a_1 + \mathrm{w}_{1,2}^{[2]} a_2 + \mathrm{w}_{1,3}^{[2]} a_3 + \mathrm{w}_{1,4}^{[2]} a_4 + b_1^{[2]}$$

$$z_1^{[2]} = \mathbf{w}_1^{[2]T} \boldsymbol{a}^{[1]} + b_1^{[2]}$$

$$a_1^{[2]} = \sigma\left(z_1^{[2]}\right)$$

Neural Network Notation

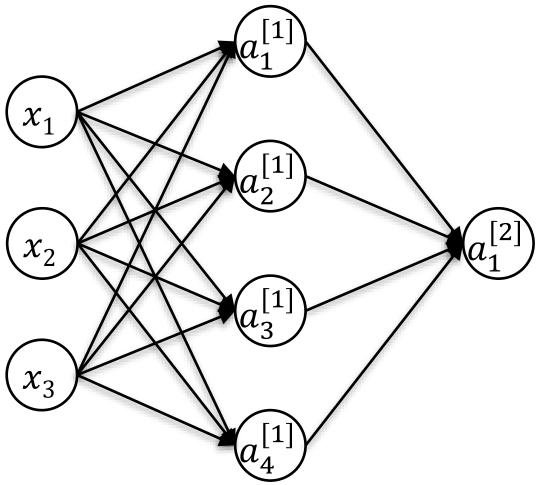


$$
\begin{aligned}
z_1^{[1]} &= \mathbf{w}_1^{[1]T}\mathbf{x} + b_1^{[1]}, & a_1^{[1]} &= \sigma\left(z_1^{[1]}\right)\\
z_2^{[1]} &= \mathbf{w}_2^{[1]T}\mathbf{x} + b_2^{[1]}, & a_2^{[1]} &= \sigma\left(z_2^{[1]}\right)\\
z_3^{[1]} &= \mathbf{w}_3^{[1]T}\mathbf{x} + b_3^{[1]}, & a_3^{[1]} &= \sigma\left(z_3^{[1]}\right)\\
z_4^{[1]} &= \mathbf{w}_4^{[1]T}\mathbf{x} + b_4^{[1]}, & a_4^{[1]} &= \sigma\left(z_4^{[1]}\right)
\end{aligned}
$$

$$
\begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \\ z_4^{[1]} \end{bmatrix} = \begin{bmatrix} - & \mathbf{w}_1^{[1]T} & - \\ - & \mathbf{w}_2^{[1]T} & - \\ - & \mathbf{w}_3^{[1]T} & - \\ - & \mathbf{w}_4^{[1]T} & - \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \\ b_4^{[1]} \end{bmatrix} \qquad \begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \\ a_3^{[1]} \\ a_4^{[1]} \end{bmatrix} = \sigma\left(\begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \\ z_4^{[1]} \end{bmatrix}\right)
$$

$$
\mathbf{z}^{[1]} = \mathbf{W}^{[1]T}\mathbf{x} + \mathbf{b}^{[1]} \qquad \boldsymbol{a}^{[1]} = \sigma\left(\mathbf{z}^{[1]}\right)
$$

Neural Network Notation



$$z_1^{[1]} = \mathbf{w}_1^{[1]T}\mathbf{x} + b_1^{[1]}, \qquad a_1^{[1]} = \sigma\left(z_1^{[1]}\right)$$
$$z_2^{[1]} = \mathbf{w}_2^{[1]T}\mathbf{x} + b_2^{[1]}, \qquad a_2^{[1]} = \sigma\left(z_2^{[1]}\right)$$
$$z_3^{[1]} = \mathbf{w}_3^{[1]T}\mathbf{x} + b_3^{[1]}, \qquad a_3^{[1]} = \sigma\left(z_3^{[1]}\right)$$
$$z_4^{[1]} = \mathbf{w}_4^{[1]T}\mathbf{x} + b_4^{[1]}, \qquad a_4^{[1]} = \sigma\left(z_4^{[1]}\right)$$

Alternatively, in row notation:

$$\left[z_1^{[1]} \quad z_2^{[1]} \quad z_3^{[1]} \quad z_4^{[1]}\right] = \left[x_1 \quad x_2 \quad x_3\right]\begin{bmatrix} | & | & | & | \\ \mathbf{w}_1^{[1]} & \mathbf{w}_2^{[1]} & \mathbf{w}_3^{[1]} & \mathbf{w}_4^{[1]} \\ | & | & | & | \end{bmatrix} + \left[b_1^{[1]} \quad b_2^{[1]} \quad b_3^{[1]} \quad b_4^{[1]}\right]$$

$$\left[a_1^{[1]} \quad a_2^{[1]} \quad a_3^{[1]} \quad a_4^{[1]}\right] = \sigma\left(\left[z_1^{[1]} \quad z_2^{[1]} \quad z_3^{[1]} \quad z_4^{[1]}\right]\right)$$

$$\mathbf{z}^{[1]T} = \mathbf{x}^T\,\mathbf{W}^{[1]} + \mathbf{b}^{[1]T} \qquad\qquad \boldsymbol{a}^{[1]T} = \sigma\left(\mathbf{z}^{[1]T}\right)$$

Neural Network Notation

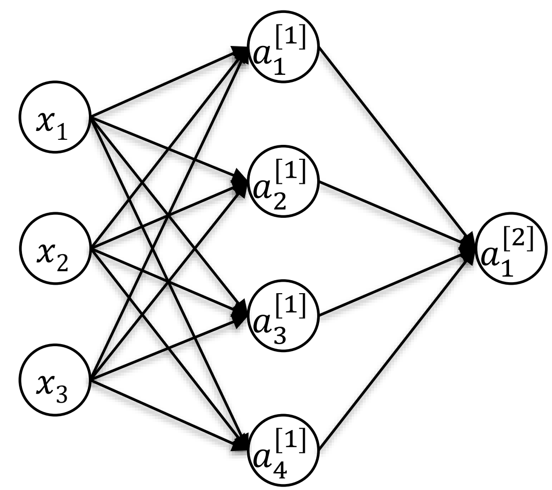
With multiple points, for layer [1]:

Sample number

Broadcasted

$$\underset{(m\times 4)}{\begin{bmatrix} - & \mathbf{z}^{(1)} & - \\ \vdots & \vdots & \vdots \\ - & \mathbf{z}^{(i)} & - \\ \vdots & \vdots & \vdots \\ - & \mathbf{z}^{(m)} & - \end{bmatrix}^{[1]}} = \underset{(m\times 3)}{\begin{bmatrix} - & \mathbf{x}^{(1)} & - \\ \vdots & \vdots & \vdots \\ - & \mathbf{x}^{(i)} & - \\ \vdots & \vdots & \vdots \\ - & \mathbf{x}^{(m)} & - \end{bmatrix}} \underset{(3\times 4)}{\begin{bmatrix} | & | & | & | \\ \mathbf{w}_1 & \mathbf{w}_2 & \mathbf{w}_3 & \mathbf{w}_4 \\ | & | & | & | \end{bmatrix}^{[1]}} + \underset{(m\times 4)}{\begin{bmatrix} - & \mathbf{b} & - \\ \vdots & \vdots & \vdots \\ - & \mathbf{b} & - \\ \vdots & \vdots & \vdots \\ - & \mathbf{b} & - \end{bmatrix}^{[1]}}$$

$$\underset{(m\times 4)}{\begin{bmatrix} - & \boldsymbol{\alpha}^{(1)} & - \\ \vdots & \vdots & \vdots \\ - & \boldsymbol{\alpha}^{(i)} & - \\ \vdots & \vdots & \vdots \\ - & \boldsymbol{\alpha}^{(m)} & - \end{bmatrix}^{[1]}} = \sigma\left(\underset{(m\times 4)}{\begin{bmatrix} - & \mathbf{z}^{(1)} & - \\ \vdots & \vdots & \vdots \\ - & \mathbf{z}^{(i)} & - \\ \vdots & \vdots & \vdots \\ - & \mathbf{z}^{(m)} & - \end{bmatrix}^{[1]}}\right)$$



Design Matrix
$(\#samples \times \#features)$

$$\mathbf{Z}^{[1]} = \mathbf{X}\,\mathbf{W}^{[1]} + \mathbf{B}^{[1]}$$

$$\mathbf{A}^{[1]} = \sigma\left(\mathbf{Z}^{[1]}\right)$$

Neural Network Notation

In general, to go from layer $[L-1]$ of k units to layer $[L]$ of n units, for a batch of m samples

$$\underset{(m\times n)}{\begin{bmatrix} - & \mathbf{z}^{(1)} & - \\ \vdots & \vdots & \vdots \\ - & \mathbf{z}^{(i)} & - \\ \vdots & \vdots & \vdots \\ - & \mathbf{z}^{(m)} & - \end{bmatrix}^{[L]}} = \underset{(m\times k)}{\begin{bmatrix} - & \boldsymbol{\alpha}^{(1)} & - \\ \vdots & \vdots & \vdots \\ - & \boldsymbol{\alpha}^{(i)} & - \\ \vdots & \vdots & \vdots \\ - & \boldsymbol{\alpha}^{(m)} & - \end{bmatrix}^{[L-1]}} \underset{(k\times n)}{\begin{bmatrix} | & \cdots & | \\ \mathbf{w}_1 & \cdots & \mathbf{w}_n \\ | & \cdots & | \end{bmatrix}^{[L]}} + \underset{(m\times n)}{\begin{bmatrix} - & \mathbf{b} & - \\ \vdots & \vdots & \vdots \\ - & \mathbf{b} & - \\ \vdots & \vdots & \vdots \\ - & \mathbf{b} & - \end{bmatrix}^{[L]}}$$

$$\underset{(m\times n)}{\begin{bmatrix} - & \boldsymbol{\alpha}^{(1)} & - \\ \vdots & \vdots & \vdots \\ - & \boldsymbol{\alpha}^{(i)} & - \\ \vdots & \vdots & \vdots \\ - & \boldsymbol{\alpha}^{(m)} & - \end{bmatrix}^{[L]}} = \sigma\left(\underset{(m\times n)}{\begin{bmatrix} - & \mathbf{z}^{(1)} & - \\ \vdots & \vdots & \vdots \\ - & \mathbf{z}^{(i)} & - \\ \vdots & \vdots & \vdots \\ - & \mathbf{z}^{(m)} & - \end{bmatrix}^{[L]}}\right)$$

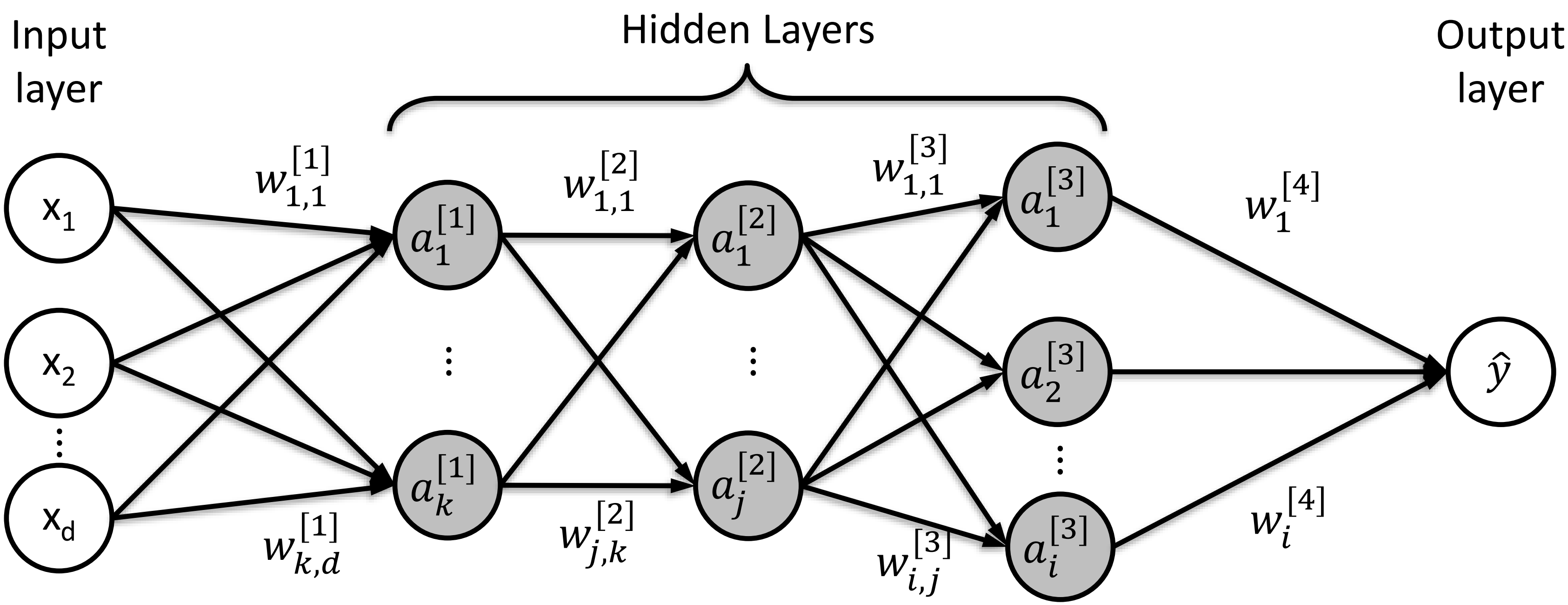
$$\mathbf{Z}^{[L]} = \mathbf{A}^{[L-1]}\mathbf{W}^{[L]} + \mathbf{B}^{[L]}$$

$$\mathbf{A}^{[L]} = \sigma\left(\mathbf{Z}^{[L]}\right)$$

Multi-layer neural networks

LEARNING WITH HIDDEN UNITS

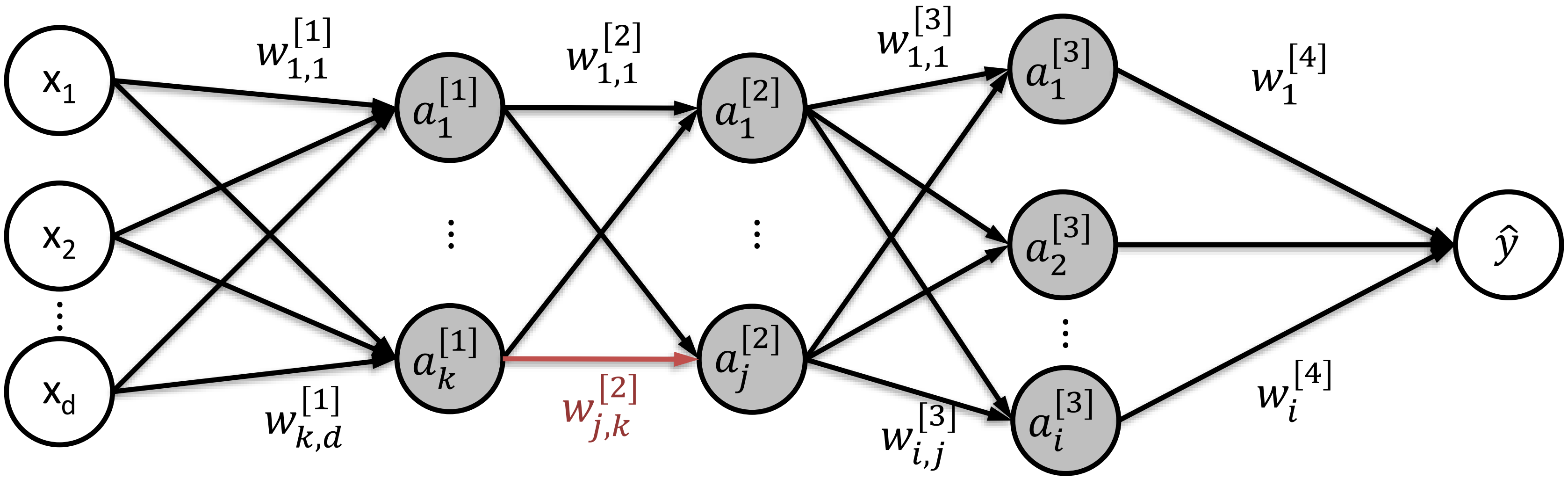
Backpropagation Algorithm



1. Receive new observation $\mathbf{x} = [x_1, x_2, \ldots, x_d]$ and target output y
2. Feed-forward: let the network calculate its predicted output $\hat{y}$
3. Get the prediction $\hat{y}$ and calculate the error (loss) e.g. $E = \frac{1}{2}(\hat{y} - y)^2$
4. **Back-propagate error**: calculate how each of the weights contributed to this error… HOW?

Backpropagation Algorithm

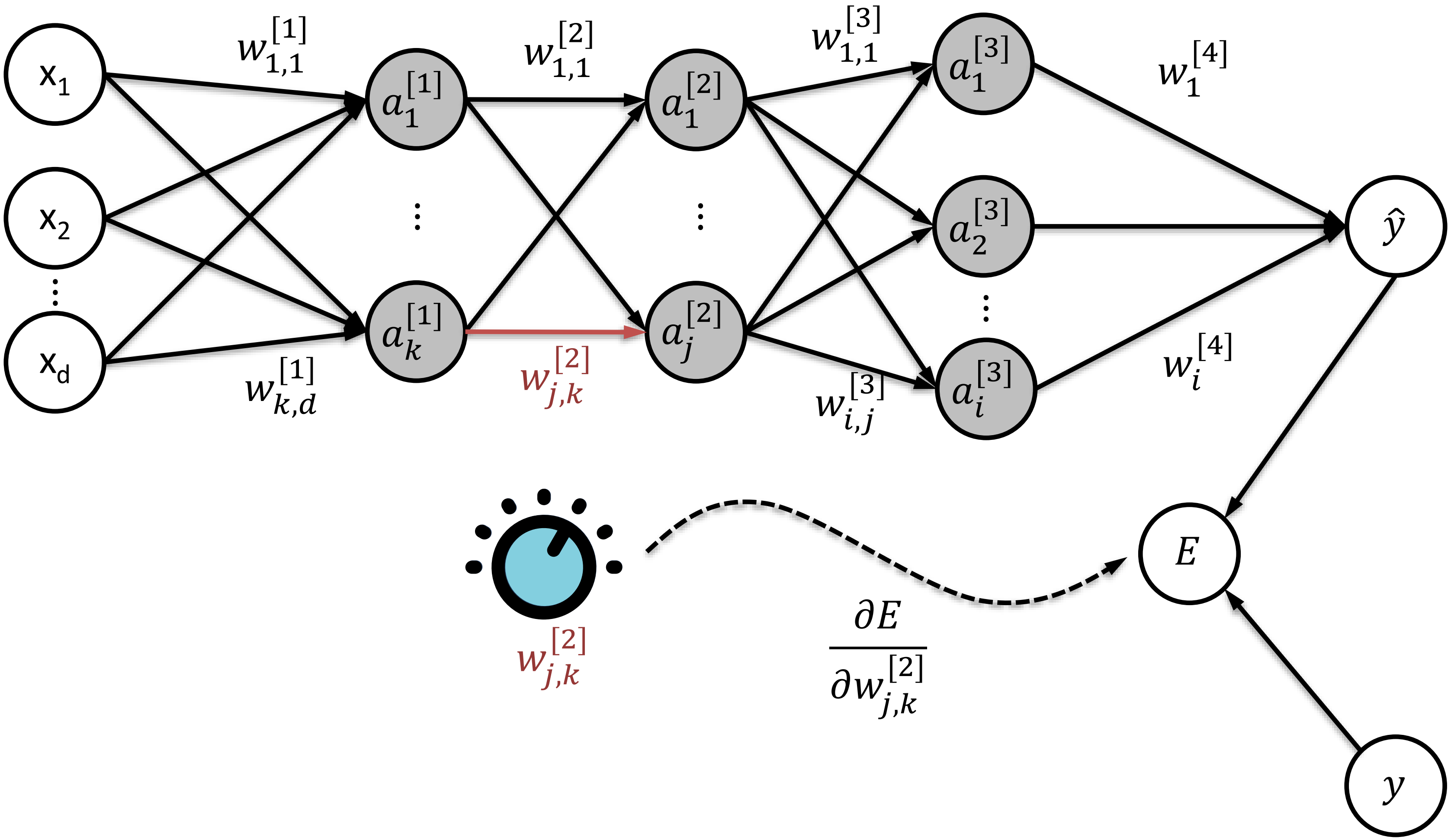
How should I change $w_{j,k}^{[2]}$?



1. Receive new observation $\mathbf{x} = [x_1, x_2, \dots, x_d]$ and target output y
2. Feed-forward: let the network calculate its predicted output $\hat{y}$
3. Get the prediction $\hat{y}$ and calculate the error (loss) e.g. $E = \frac{1}{2}(\hat{y} - y)^2$
4. **Back-propagate error**: calculate how each of the weights contributed to this error… HOW?

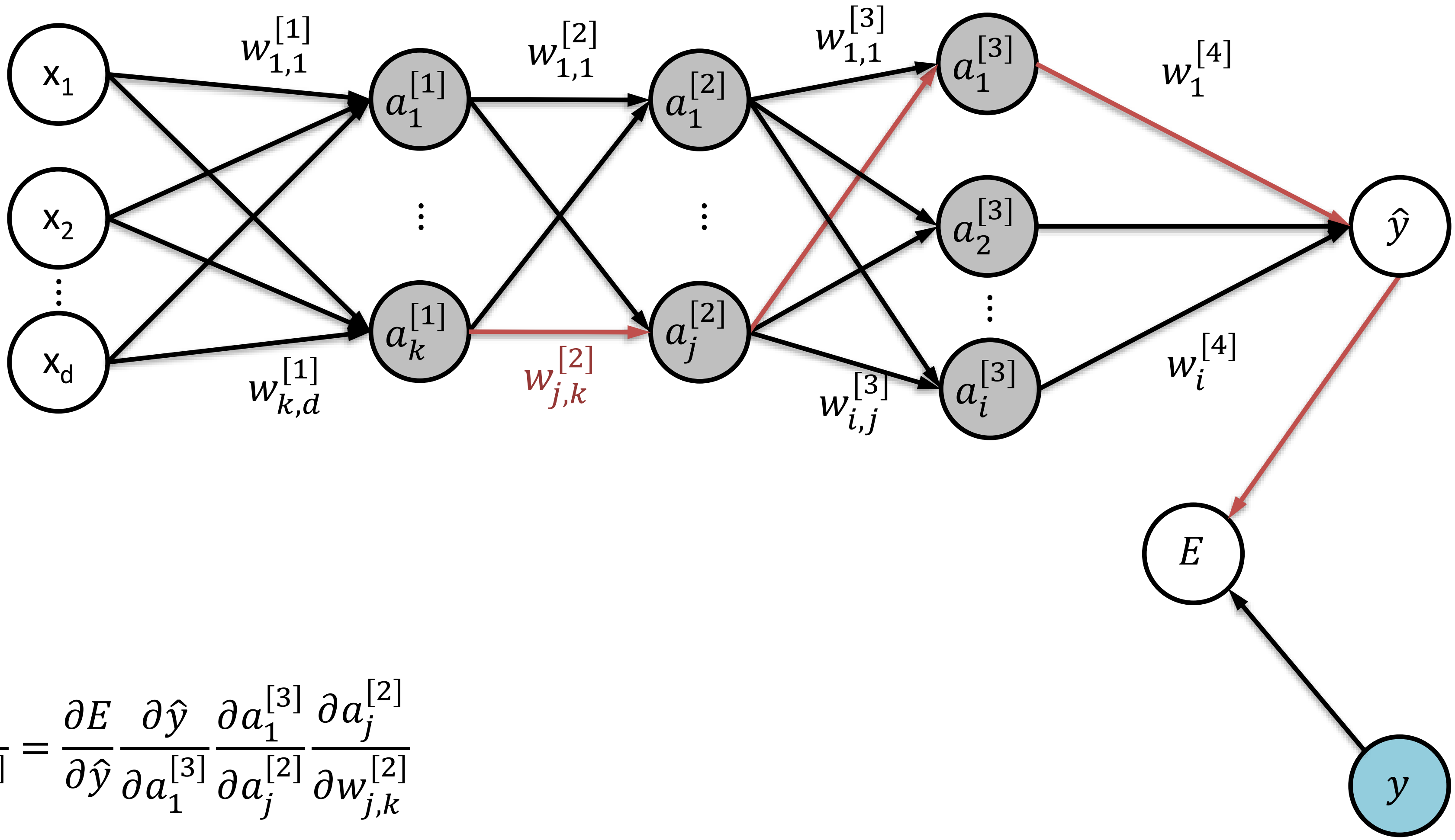
Backpropagation Algorithm

How should I change $w_{j,k}^{[2]}$?



Backpropagation Algorithm

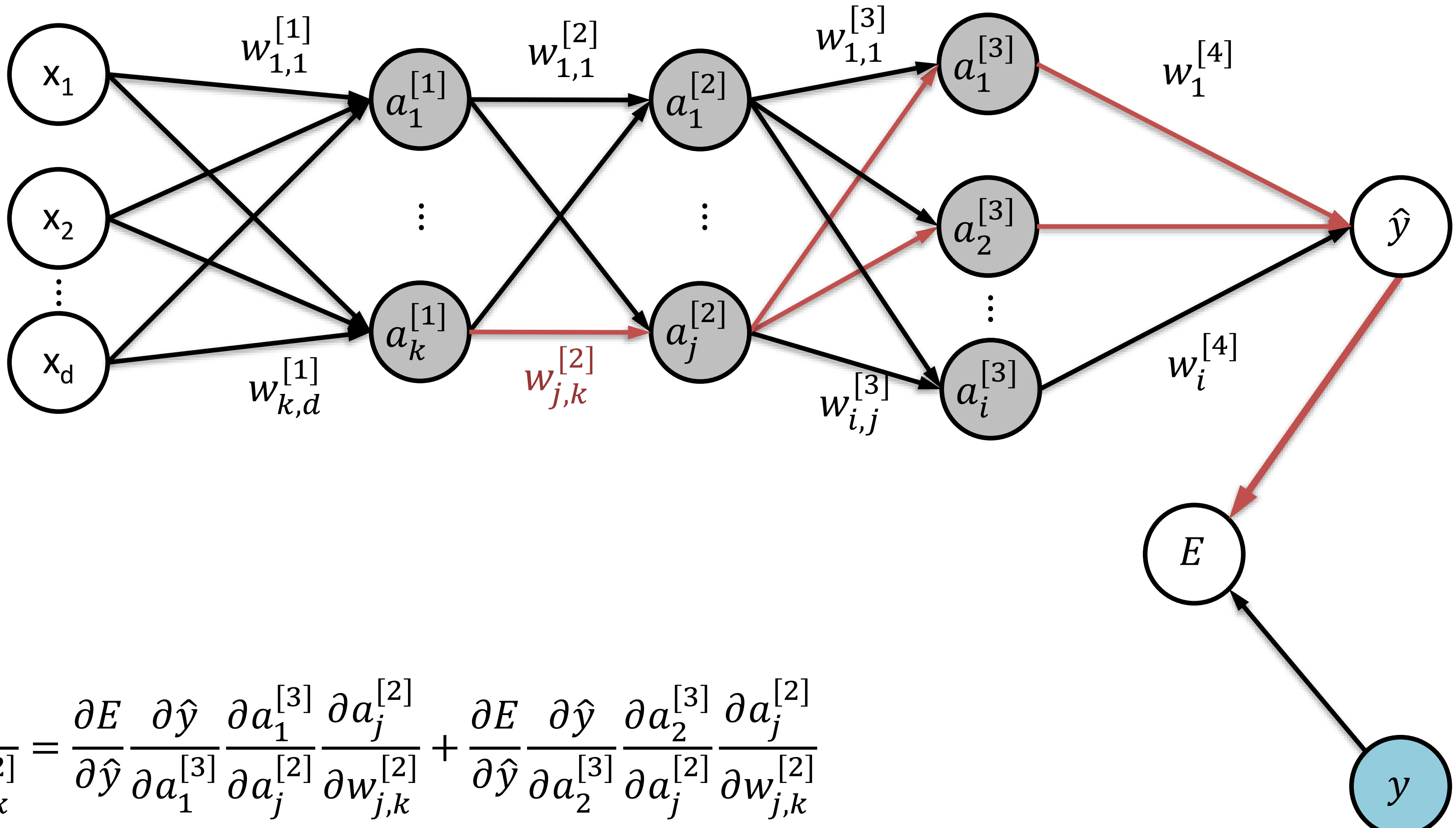
How should I change $w_{j,k}^{[2]}$?



$$\frac{\partial E}{\partial w_{j,k}^{[2]}} = \frac{\partial E}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial a_1^{[3]}} \frac{\partial a_1^{[3]}}{\partial a_j^{[2]}} \frac{\partial a_j^{[2]}}{\partial w_{j,k}^{[2]}}$$

Backpropagation Algorithm

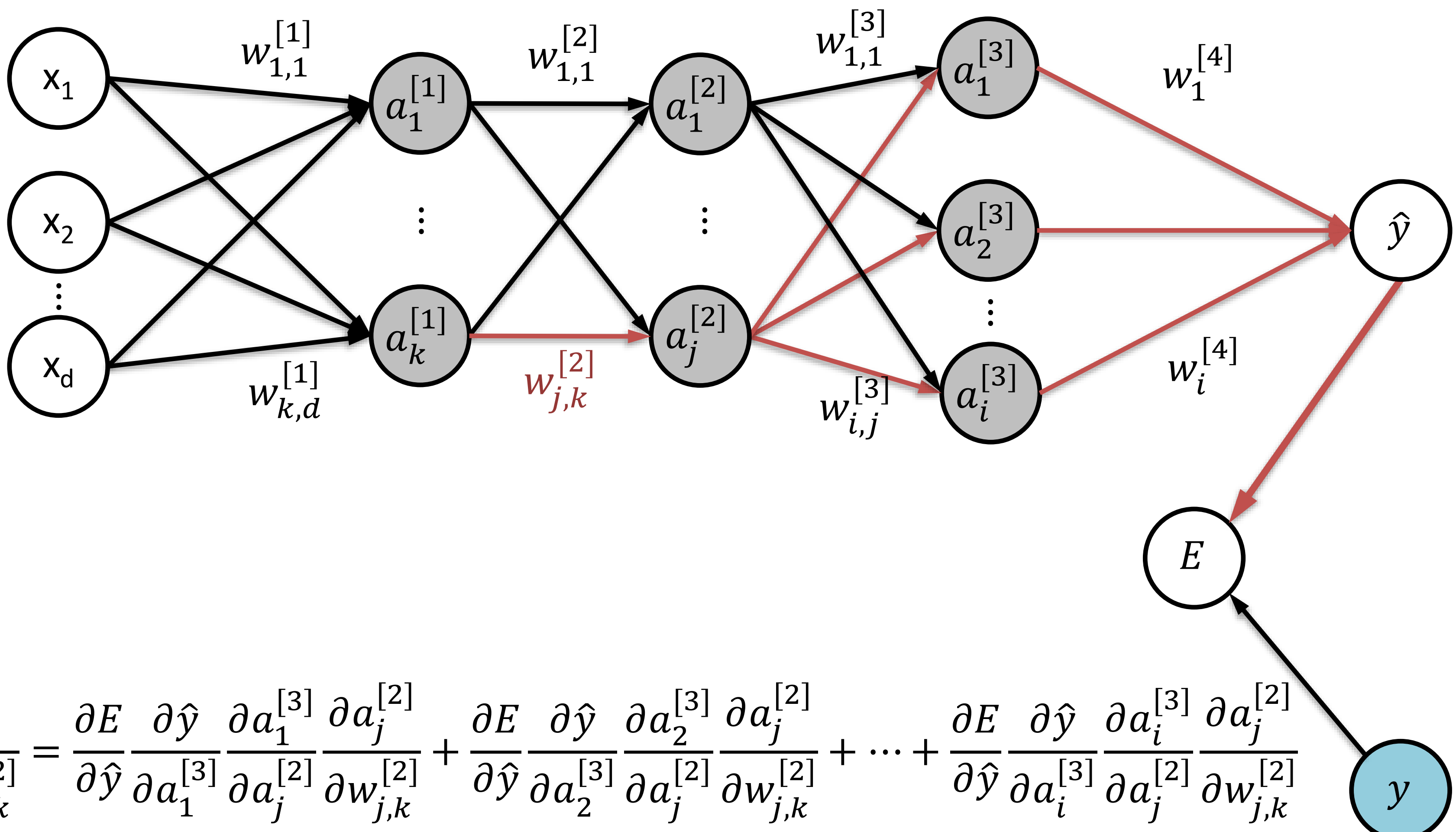
How should I change $w_{j,k}^{[2]}$?



$$\frac{\partial E}{\partial w_{j,k}^{[2]}} = \frac{\partial E}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial a_1^{[3]}} \frac{\partial a_1^{[3]}}{\partial a_j^{[2]}} \frac{\partial a_j^{[2]}}{\partial w_{j,k}^{[2]}} + \frac{\partial E}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial a_2^{[3]}} \frac{\partial a_2^{[3]}}{\partial a_j^{[2]}} \frac{\partial a_j^{[2]}}{\partial w_{j,k}^{[2]}}$$

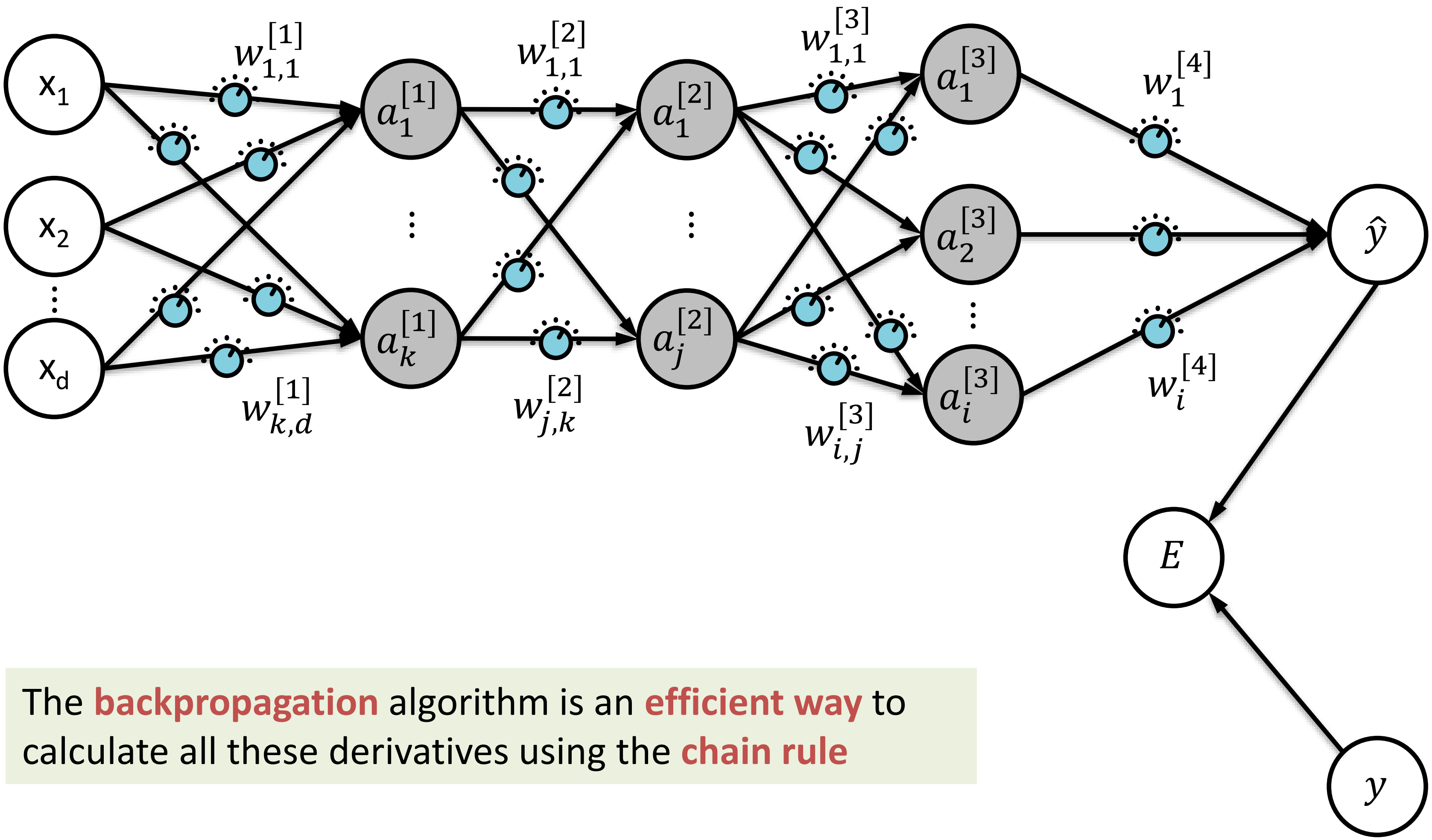
Backpropagation Algorithm

How should I change $w_{j,k}^{[2]}$?



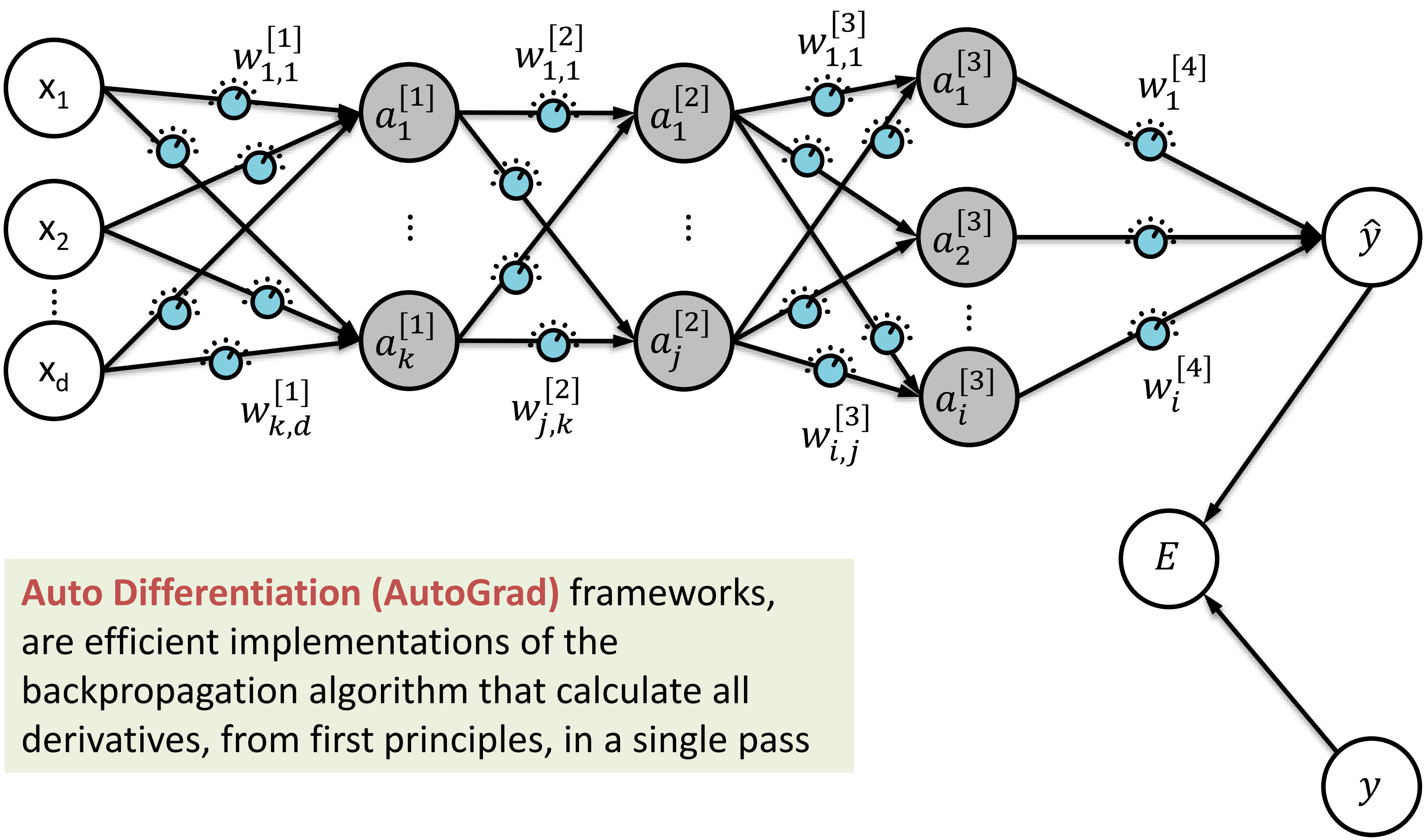
$$\frac{\partial E}{\partial w_{j,k}^{[2]}} = \frac{\partial E}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial a_1^{[3]}} \frac{\partial a_1^{[3]}}{\partial a_j^{[2]}} \frac{\partial a_j^{[2]}}{\partial w_{j,k}^{[2]}} + \frac{\partial E}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial a_2^{[3]}} \frac{\partial a_2^{[3]}}{\partial a_j^{[2]}} \frac{\partial a_j^{[2]}}{\partial w_{j,k}^{[2]}} + \cdots + \frac{\partial E}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial a_i^{[3]}} \frac{\partial a_i^{[3]}}{\partial a_j^{[2]}} \frac{\partial a_j^{[2]}}{\partial w_{j,k}^{[2]}}$$

Backpropagation Algorithm



The **backpropagation** algorithm is an **efficient way** to calculate all these derivatives using the **chain rule**

Backpropagation Algorithm



Auto Differentiation (AutoGrad) frameworks, are efficient implementations of the backpropagation algorithm that calculate all derivatives, from first principles, in a single pass

Calculating Derivatives of Composite Functions

AUTO DIFFERENTIATION

Example

```
a = 4
b = 3
c = a + b # = 4 + 3 = 7
d = a * c # = 4 * 7 = 28
```

What is the derivative of d with respect to a: $\frac{\partial d}{\partial a}$?

$d = a * c$

Solving the traditional way

Example

```
a = 4
b = 3
c = a + b # = 4 + 3 = 7
d = a * c # = 4 * 7 = 28
```

What is the derivative of d with respect to a: $\frac{\partial d}{\partial a}$?

$$d = a * c$$

$$\begin{aligned}\frac{\partial d}{\partial a} &= \frac{\partial a}{\partial a} * c + a * \frac{\partial c}{\partial a} \\ &= c + a * \frac{\partial c}{\partial a} \\ &= (a + b) + a * \frac{\partial (a + b)}{\partial a} \\ &= a + b + a * \left(\frac{\partial a}{\partial a} + \frac{\partial b}{\partial a}\right) \\ &= a + b + a * (1 + 0) \\ &= a + b + a = 2a + b \\ &= 2 * 4 + 3 = 11\end{aligned}$$

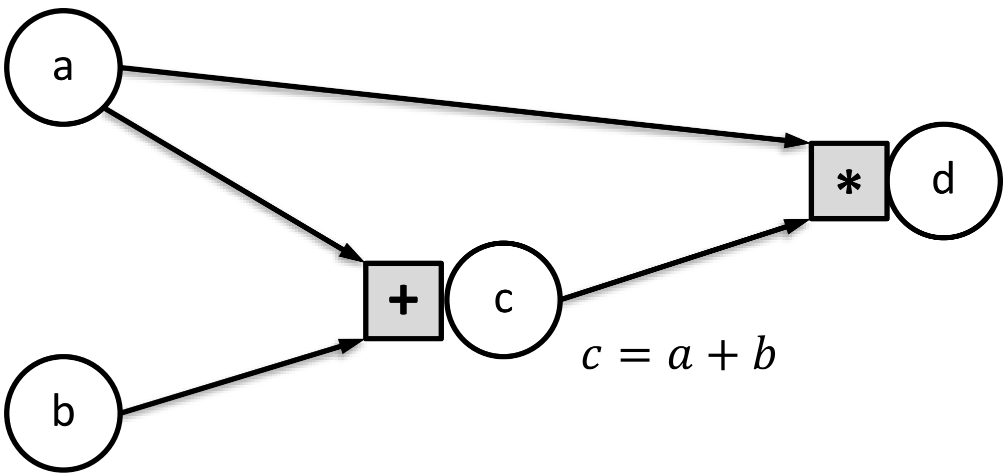
Solving the traditional way

Now, what is the derivative of d with respect to b: $\frac{\partial d}{\partial b}$?

You would have to carry out the whole process again…

Computational Graph

```
a = 4
b = 3
c = a + b # = 4 + 3 = 7
d = a * c # = 4 * 7 = 28
```

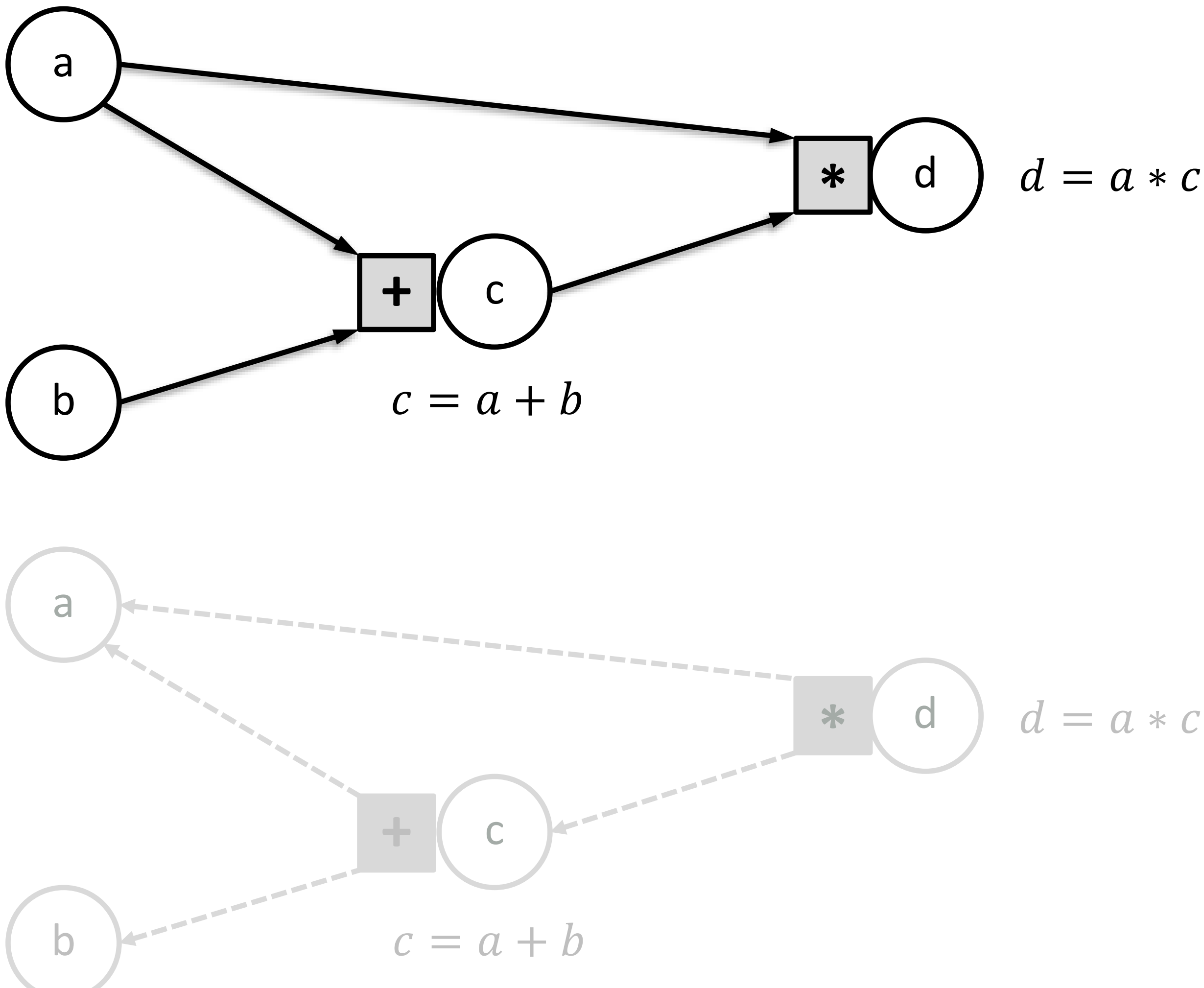


$d = a * c$

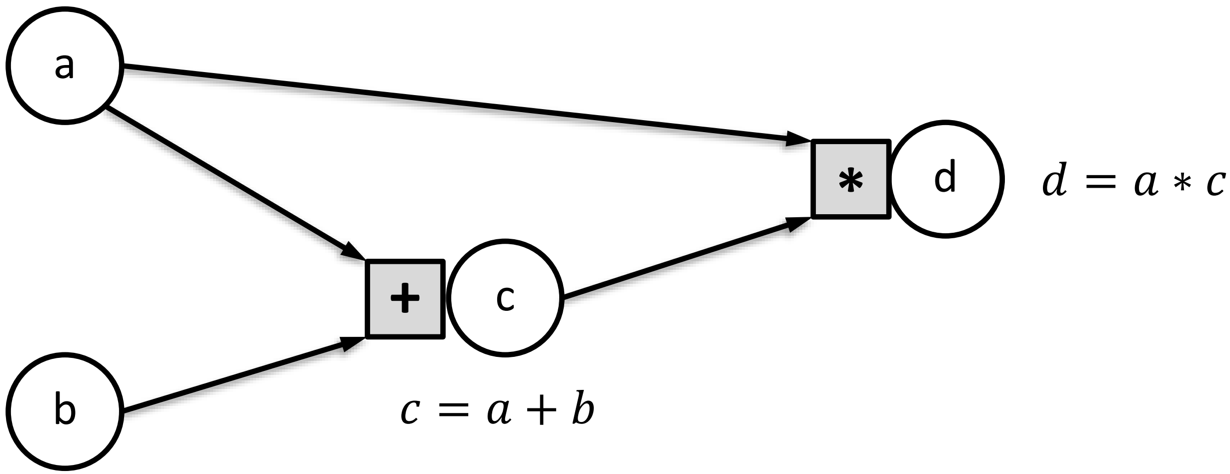
$c = a + b$

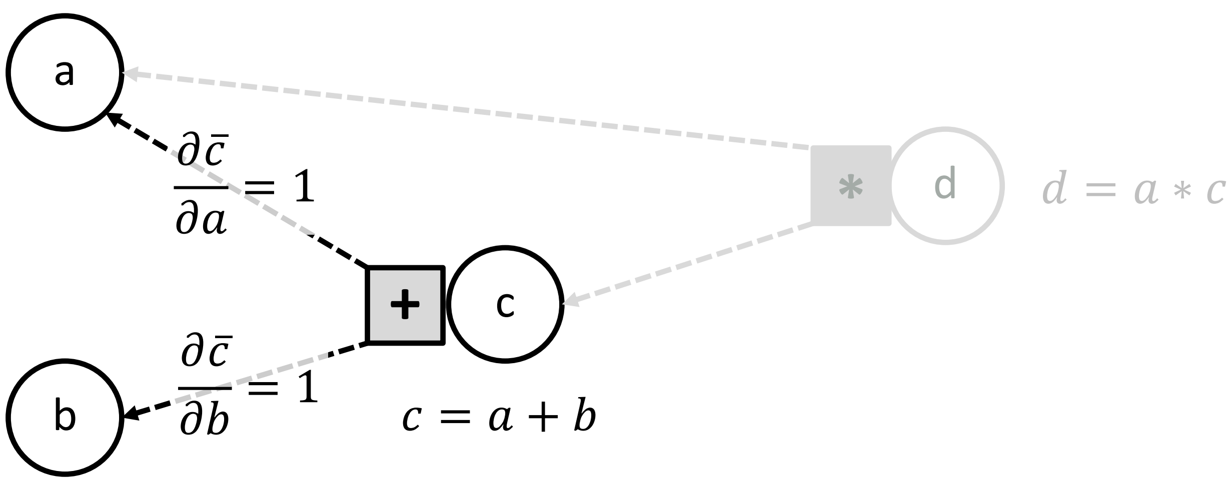
Any expression can be broken down into a series of **simple operations** that are **applied sequentially**

Local Derivatives

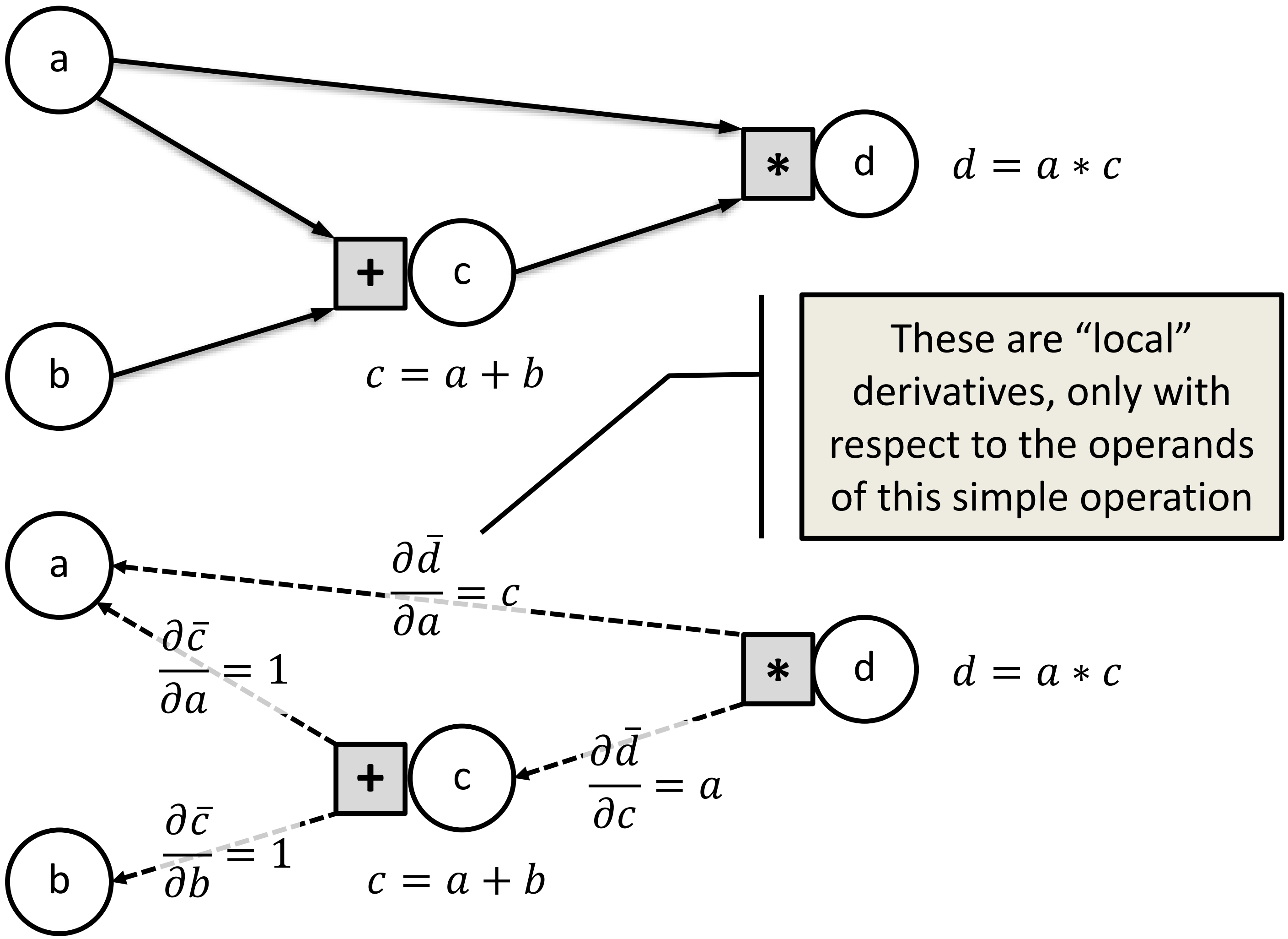


Local Derivatives



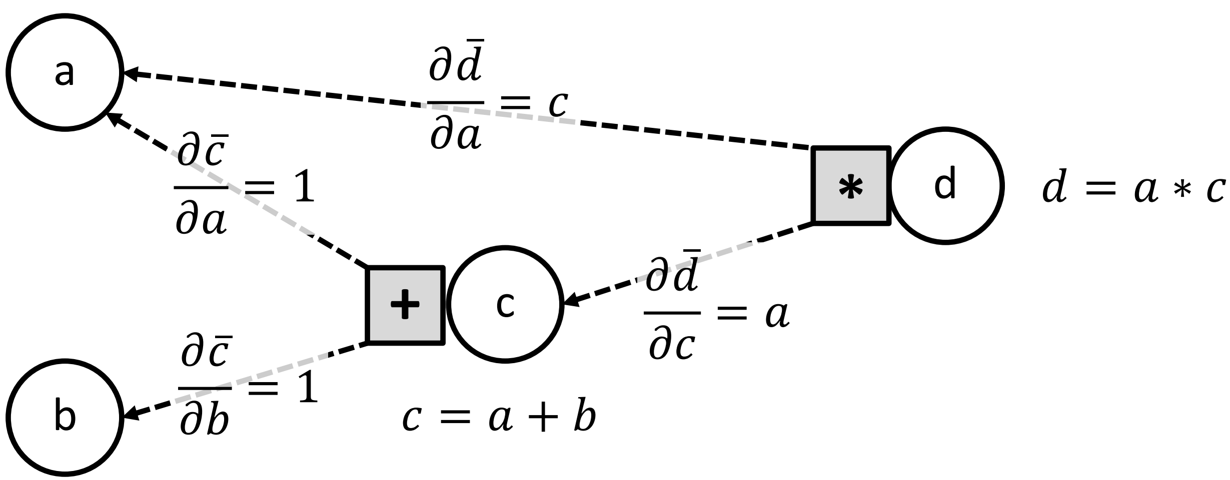


Local Derivatives



Automatic Differentiation (AutoGrad)

$$\frac{\partial d}{\partial a} = \overset{\text{Route 1}}{\frac{\partial \bar{d}}{\partial a}} + \overbrace{\frac{\partial \bar{d}}{\partial c} * \frac{\partial \bar{c}}{\partial a}}^{\text{Route 2}}$$

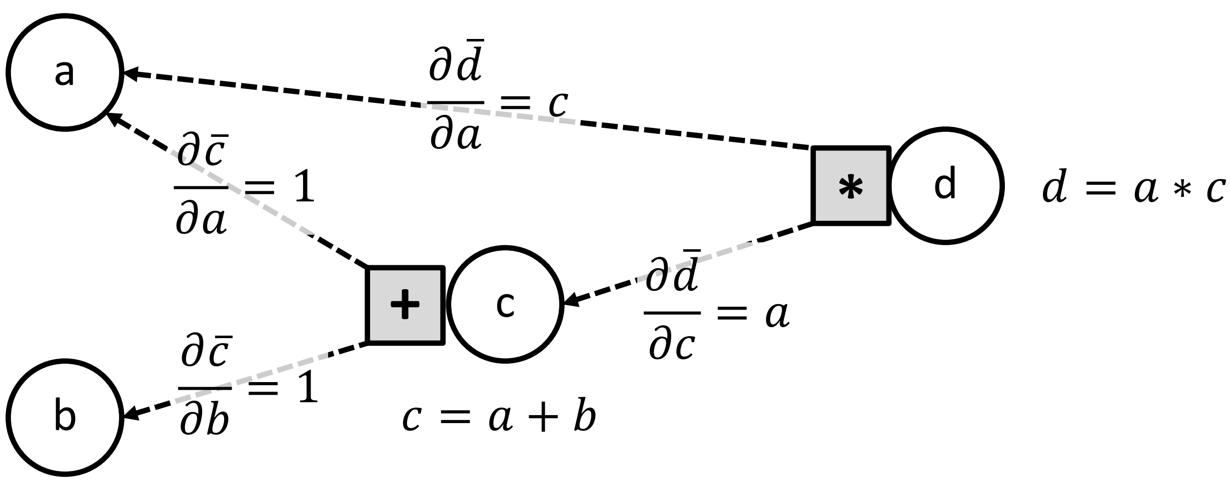


Automatic Differentiation (AutoGrad)

$$\frac{\partial d}{\partial a} = \overbrace{\frac{\partial \bar{d}}{\partial a}}^{\text{Route 1}} + \overbrace{\frac{\partial \bar{d}}{\partial c} * \frac{\partial \bar{c}}{\partial a}}^{\text{Route 2}}$$
$$= c + a * 1$$
$$= a + b + a$$
$$= 2a + b$$
$$= 11$$

To calculate **any** derivative using the computational graph:

- **Multiply** the edges of a route
- **Add** together the different routes that lead from the quantity to derive to the node of interest

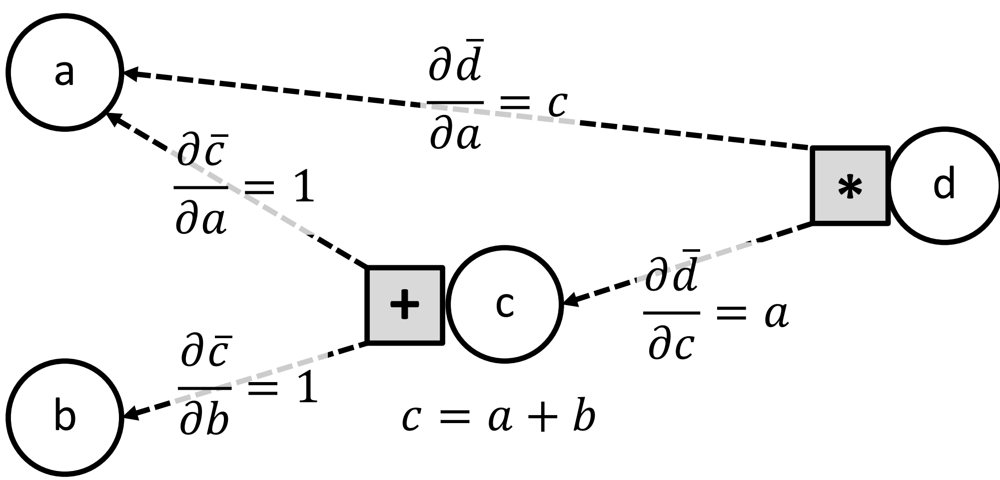


Automatic Differentiation (AutoGrad)

$$\frac{\partial d}{\partial a} = \overbrace{\frac{\partial \bar{d}}{\partial a}}^{\text{Route 1}} + \overbrace{\frac{\partial \bar{d}}{\partial c} * \frac{\partial \bar{c}}{\partial a}}^{\text{Route 2}}$$
$$= c + a * 1$$
$$= a + b + a$$
$$= 2a + b$$
$$= 11$$

To calculate **any** derivative using the computational graph:

- **Multiply** the edges of a route
- **Add** together the different routes that lead from the quantity to derive to the node of interest

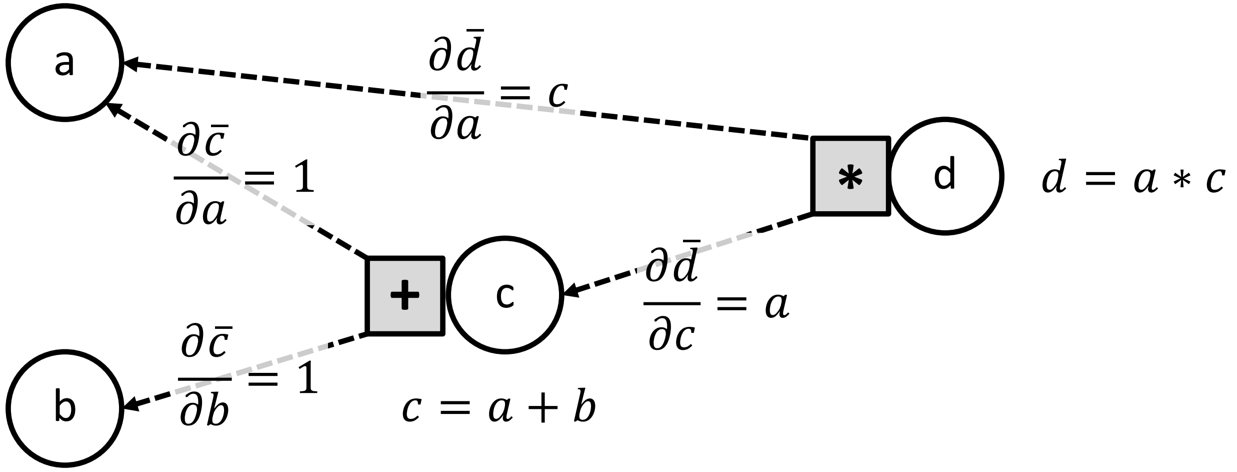


$$\left.\begin{aligned} d &= a * c \\ \frac{\partial d}{\partial a} &= \frac{\partial a}{\partial a} * c + a * \frac{\partial c}{\partial a} \\ &= c + a * \frac{\partial c}{\partial a} \\ &= (a + b) + a * \frac{\partial (a + b)}{\partial a} \\ &= a + b + a * \left(\frac{\partial a}{\partial a} + \frac{\partial b}{\partial a}\right) \\ &= a + b + a * (1 + 0) \\ &= a + b + a = 2a + b \\ &= 2 * 4 + 3 = 11 \end{aligned}\right\} \text{Solving the traditional way}$$

Automatic Differentiation (AutoGrad)

Q1: What is the derivative of d with respect to b?
Q2: What is the derivative of d with respect to c?
Q3: What is the derivative of c with respect to a?

Remember:
- **Multiply** the edges of a route
- **Add** together the different routes that lead to the node

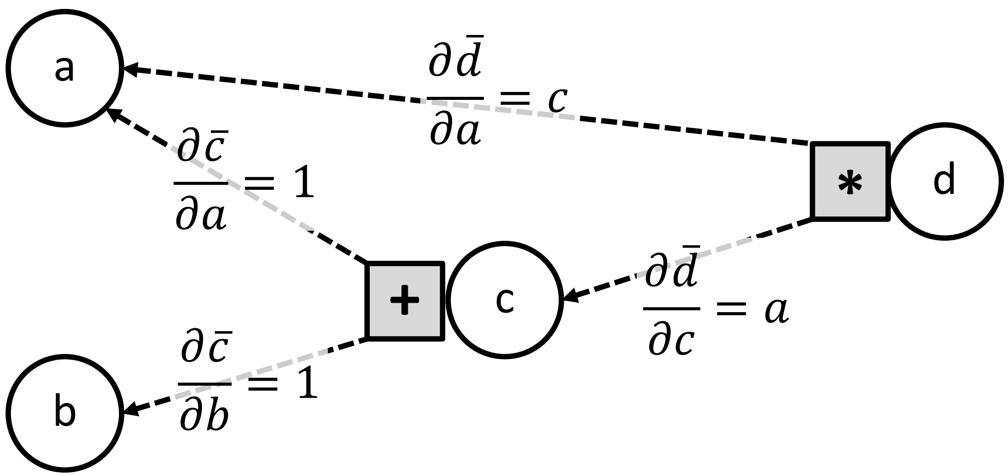


The Backwards Pass

The backwards pass is the algorithmic implementation of the backpropagation process, that calculates the derivative of d with respect to each node in the graph in a single pass

Remember:

- **Multiply** the edges of a route
- **Add** together the different routes that lead to a node

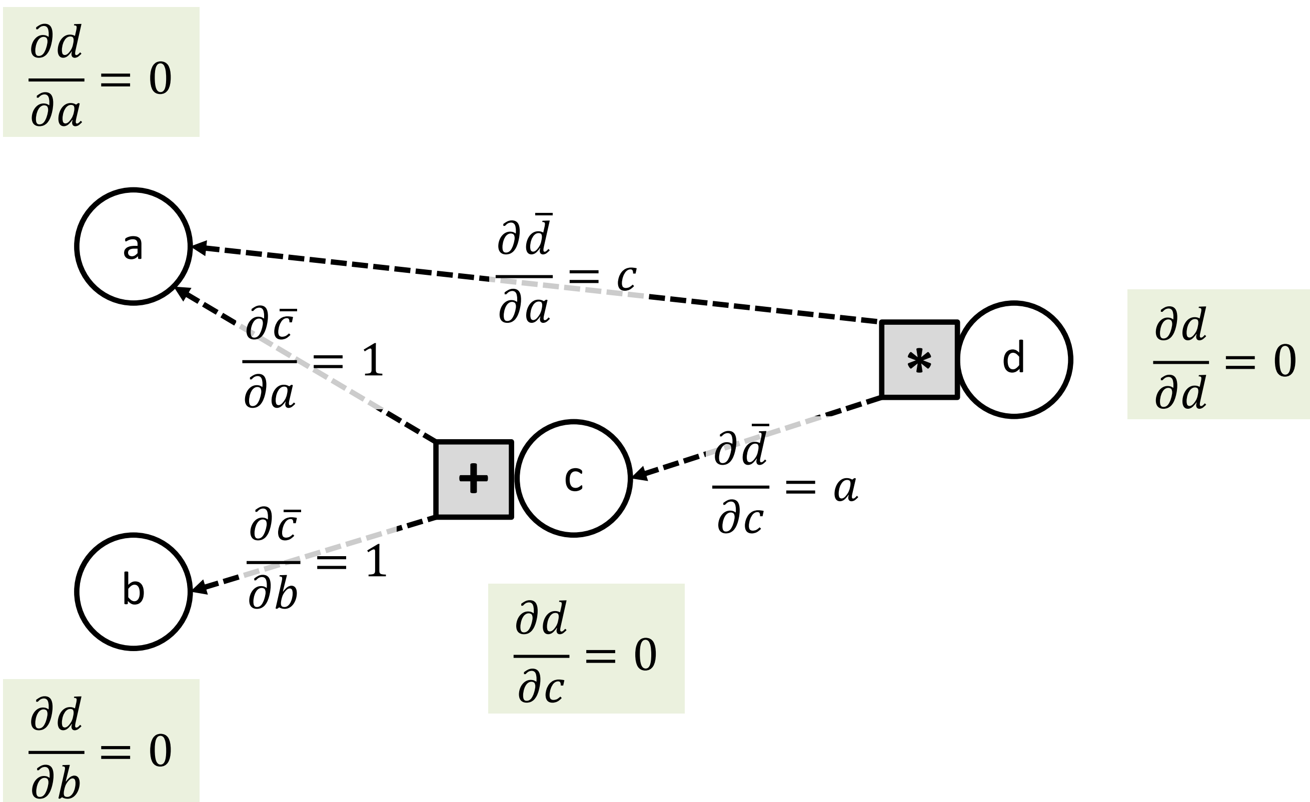


The Backwards Pass

The backwards pass is the algorithmic implementation of the backpropagation process, that calculates the derivative of d with respect to each node in the graph in a single pass

Remember:

- **Multiply** the edges of a route
- **Add** together the different routes that lead to a node



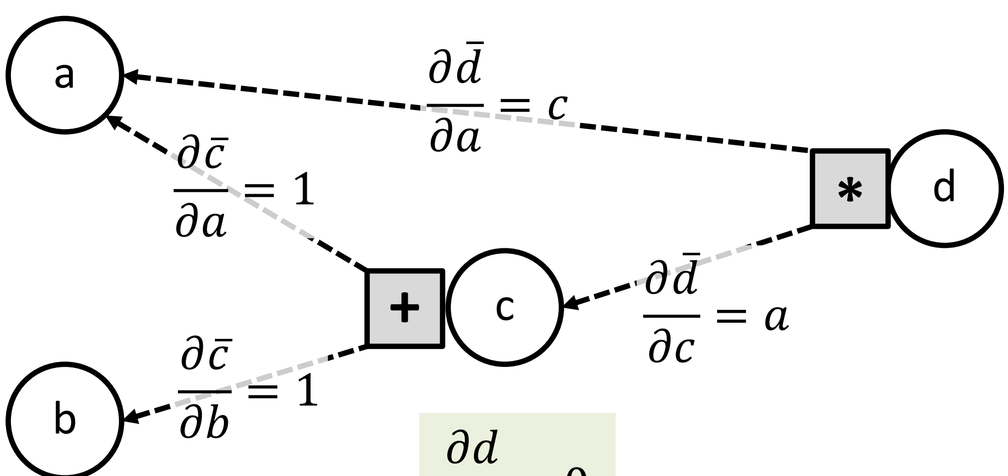
The Backwards Pass

The backwards pass is the algorithmic implementation of the backpropagation process, that calculates the derivative of d with respect to each node in the graph in a single pass

Remember:

- **Multiply** the edges of a route
- **Add** together the different routes that lead to a node

$$\frac{\partial d}{\partial a} = 0$$



$$\frac{\partial d}{\partial d} = 1$$

$$\frac{\partial d}{\partial c} = 0$$

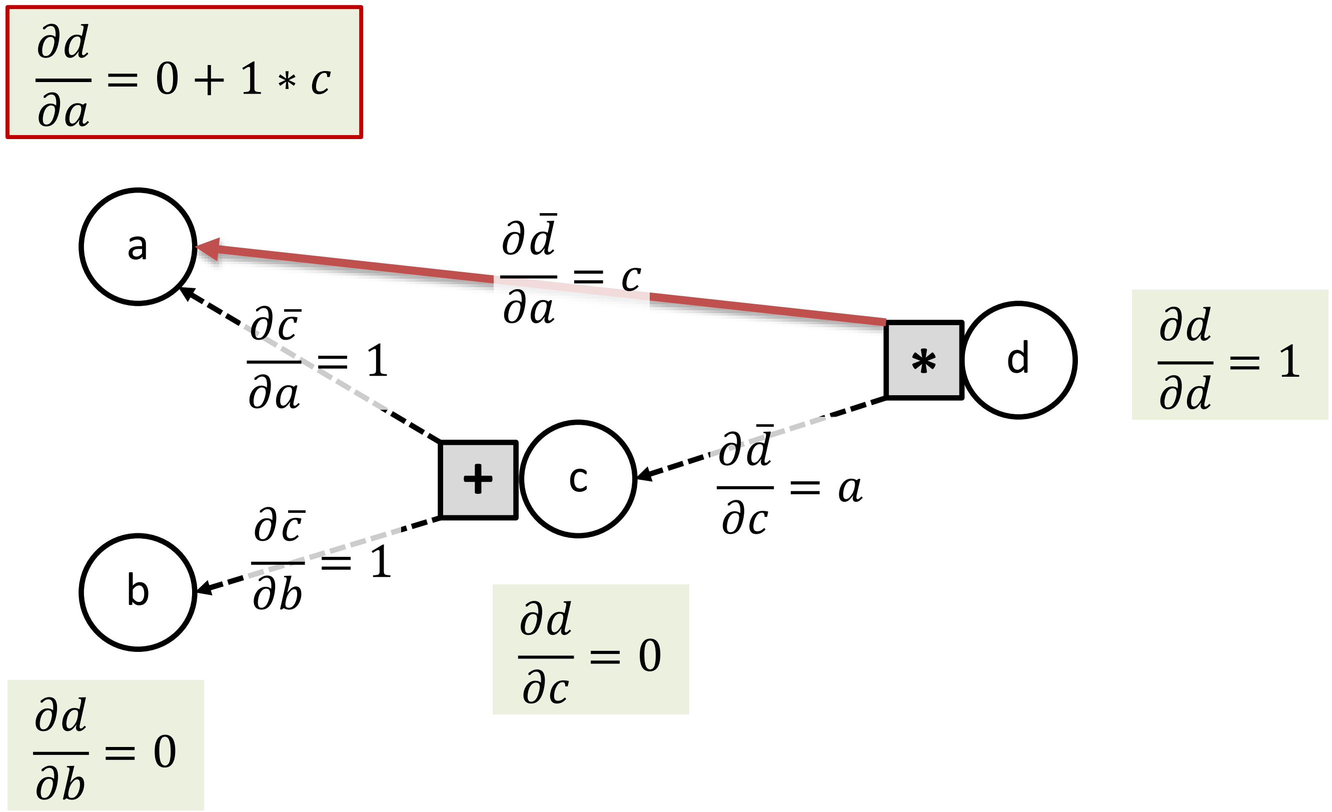
$$\frac{\partial d}{\partial b} = 0$$

The Backwards Pass

The backwards pass is the algorithmic implementation of the backpropagation process, that calculates the derivative of d with respect to each node in the graph in a single pass

Remember:

- **Multiply** the edges of a route
- **Add** together the different routes that lead to a node



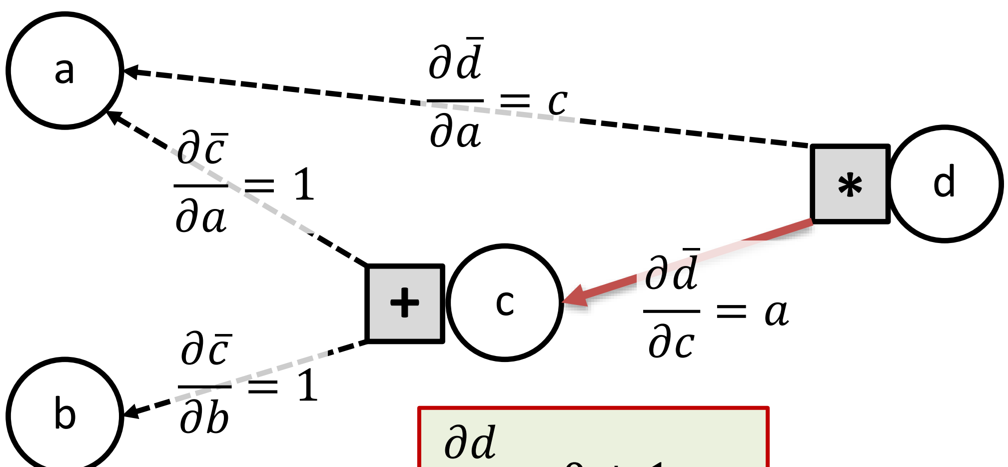
The Backwards Pass

The backwards pass is the algorithmic implementation of the backpropagation process, that calculates the derivative of d with respect to each node in the graph in a single pass

Remember:

- **Multiply** the edges of a route
- **Add** together the different routes that lead to a node

$$\frac{\partial d}{\partial a} = 0 + 1 * c$$



$$\frac{\partial d}{\partial d} = 1$$

$$\boxed{\frac{\partial d}{\partial c} = 0 + 1 * a}$$

$$\frac{\partial d}{\partial b} = 0$$

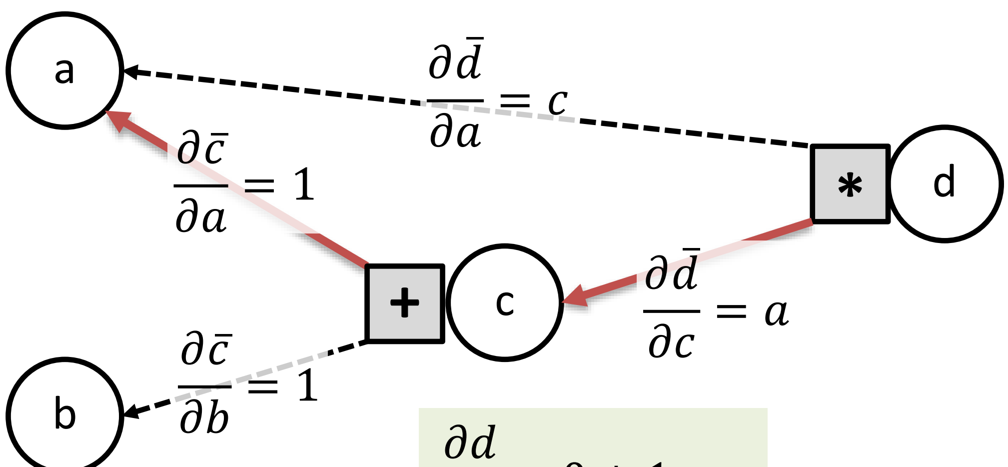
The Backwards Pass

The backwards pass is the algorithmic implementation of the backpropagation process, that calculates the derivative of d with respect to each node in the graph in a single pass

Remember:

- **Multiply** the edges of a route
- **Add** together the different routes that lead to a node

$$\boxed{\frac{\partial d}{\partial a} = 0 + 1 * c + (1 * a) * 1}$$



$$\frac{\partial d}{\partial d} = 1$$

$$\frac{\partial d}{\partial c} = 0 + 1 * a$$

$$\frac{\partial d}{\partial b} = 0$$

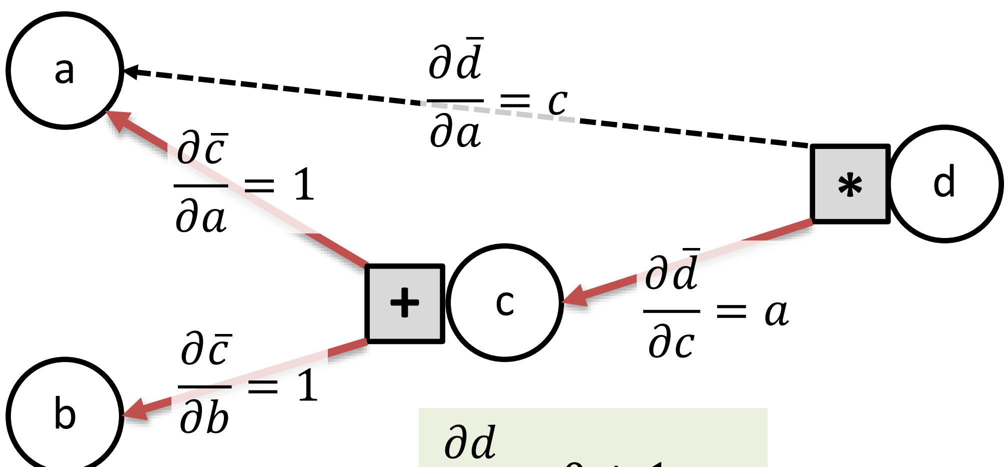
The Backwards Pass

The backwards pass is the algorithmic implementation of the backpropagation process, that calculates the derivative of d with respect to each node in the graph in a single pass

Remember:

- **Multiply** the edges of a route
- **Add** together the different routes that lead to a node

$$\frac{\partial d}{\partial a} = 0 + 1 * c + (1 * a) * 1$$



$$\frac{\partial d}{\partial d} = 1$$

$$\frac{\partial d}{\partial c} = 0 + 1 * a$$

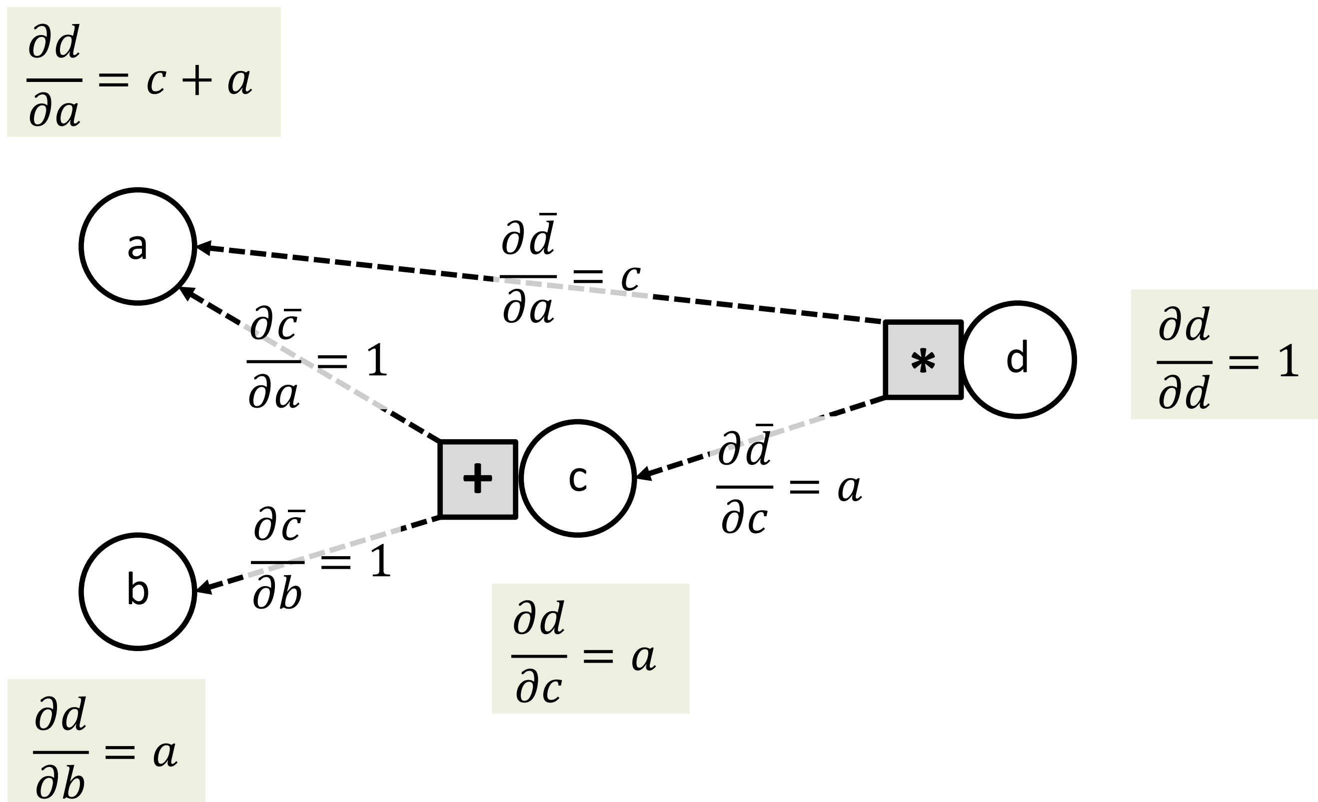
$$\boxed{\frac{\partial d}{\partial b} = 0 + (1 * a) * 1}$$

The Backwards Pass

The backwards pass is the algorithmic implementation of the backpropagation process, that calculates the derivative of d with respect to each node in the graph in a single pass

Remember:

- **Multiply** the edges of a route
- **Add** together the different routes that lead to a node

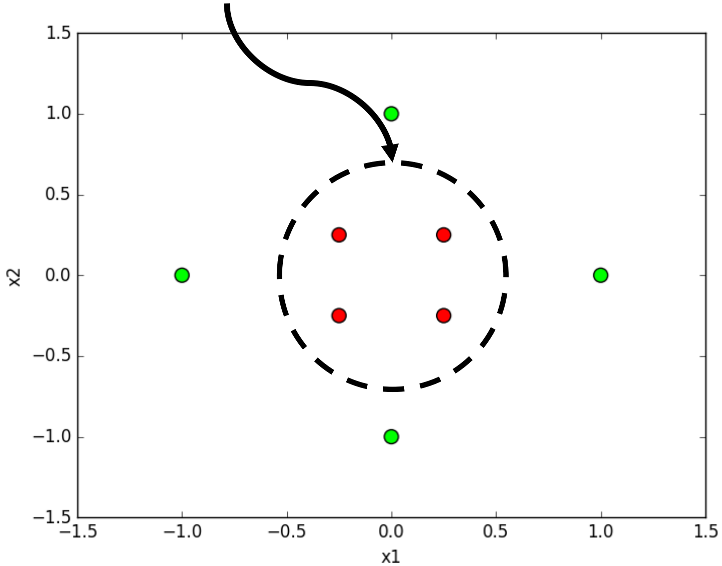


Using our AutoGrad framework

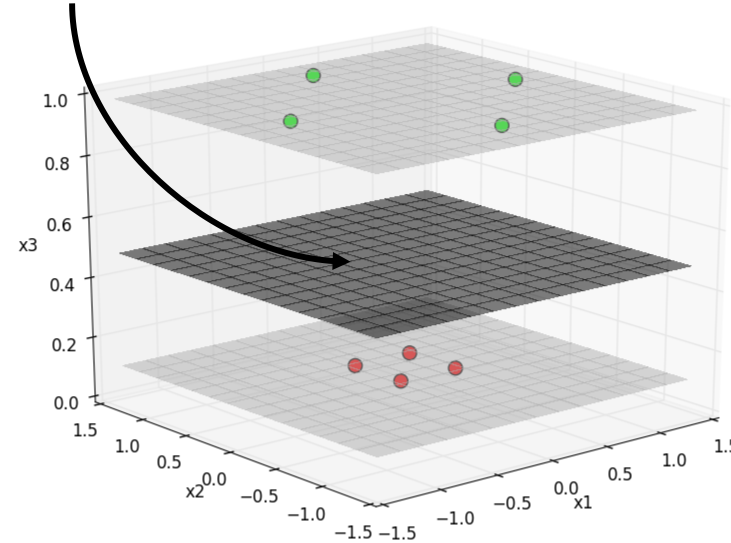
LEARNING THE PARAMETERS OF COMPOSITE FUNCTIONS

Non linear decision boundaries

$x_1^2 + x_2^2 = 0.5$

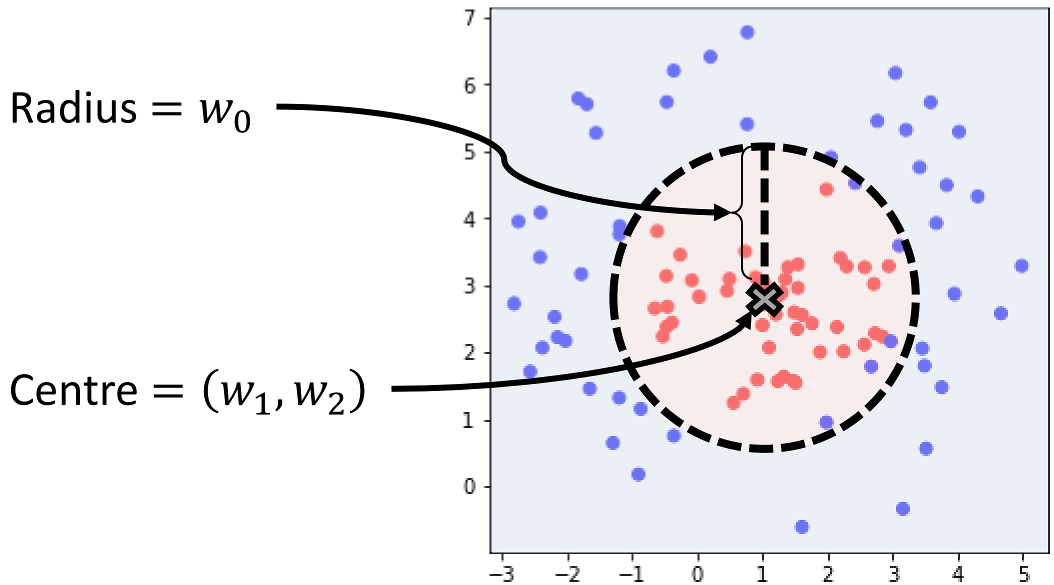


$x_3 = 0.5$



$$(x_1, x_2) \rightarrow (x_1, x_2, x_3 = x_1^2 + x_2^2)$$

Example



We have some good intuition that we are looking for a closed decision boundary. We could try with a circle – but we have no prior knowledge of where the centre is, nor the radius. These are the parameters we are looking for.

$$z = (x_1 - w_1)^2 + (x_2 - w_2)^2 - w_0^2$$

Gradient Descent

Gradient descent works as usual, but in this case, you would have to calculate a complicated derivative, including the derivative of $\partial z / \partial w_i$

$$z = (x_1 - w_1)^2 + (x_2 - w_2)^2 - w_0^2$$

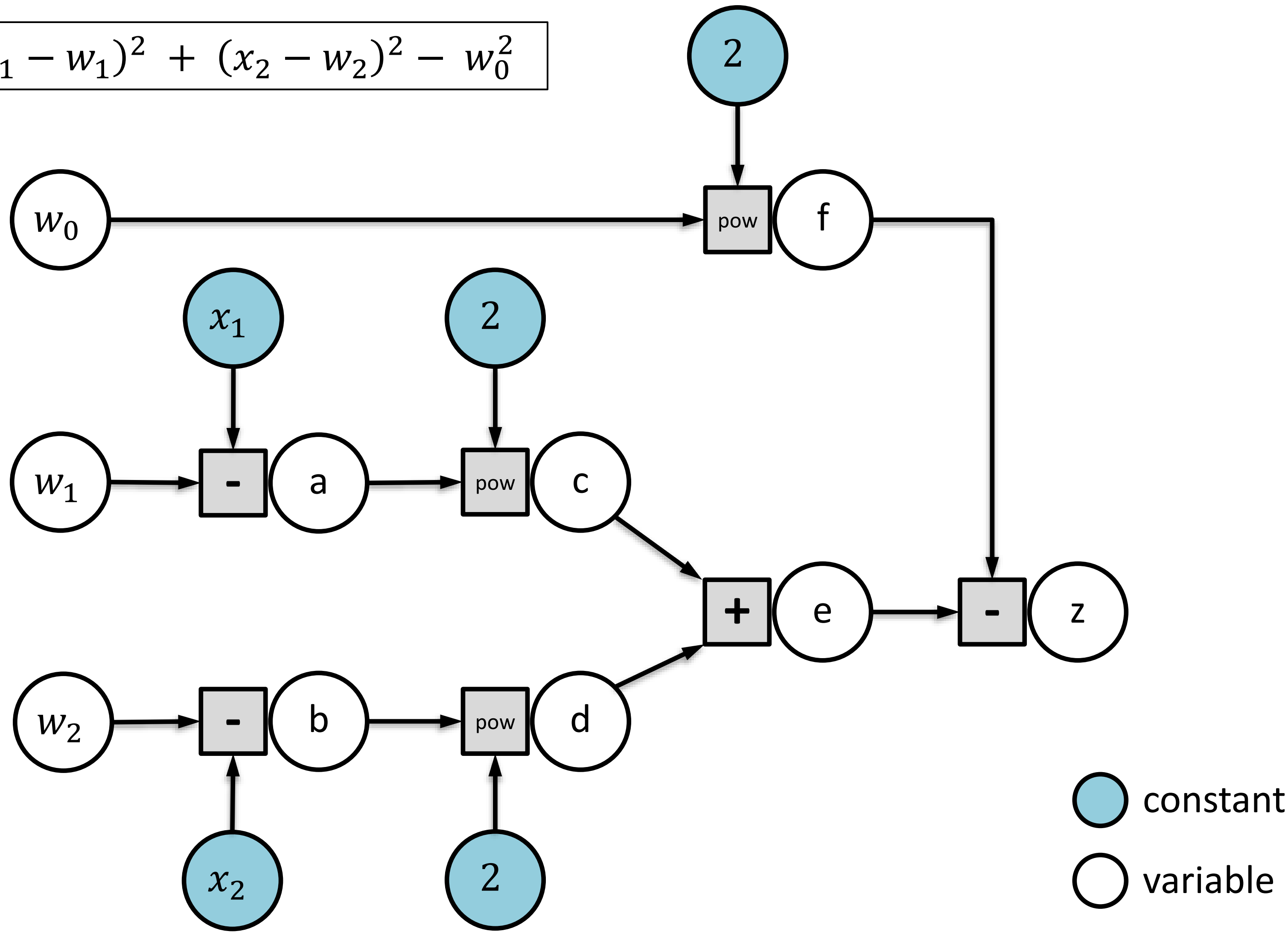
$$\frac{\partial z}{\partial w_0} = ? \quad \frac{\partial z}{\partial w_1} = ? \quad \frac{\partial z}{\partial w_2} = ?$$

AutoGrad can calculate all these derivatives for us. Then we can use normal gradient descent:

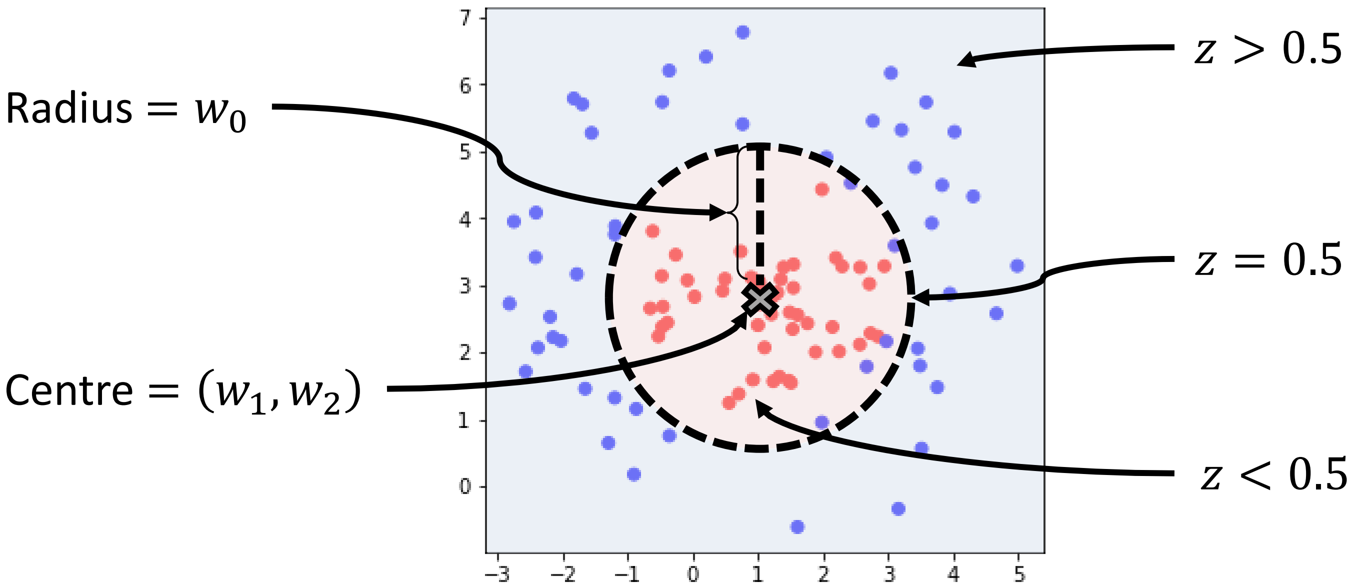
$$w_i \leftarrow w_i - \alpha \frac{\partial E}{\partial w_i}$$

Computational Graph

$$z = (x_1 - w_1)^2 + (x_2 - w_2)^2 - w_0^2$$



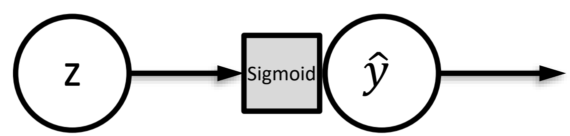
Activation Function (Sigmoid)



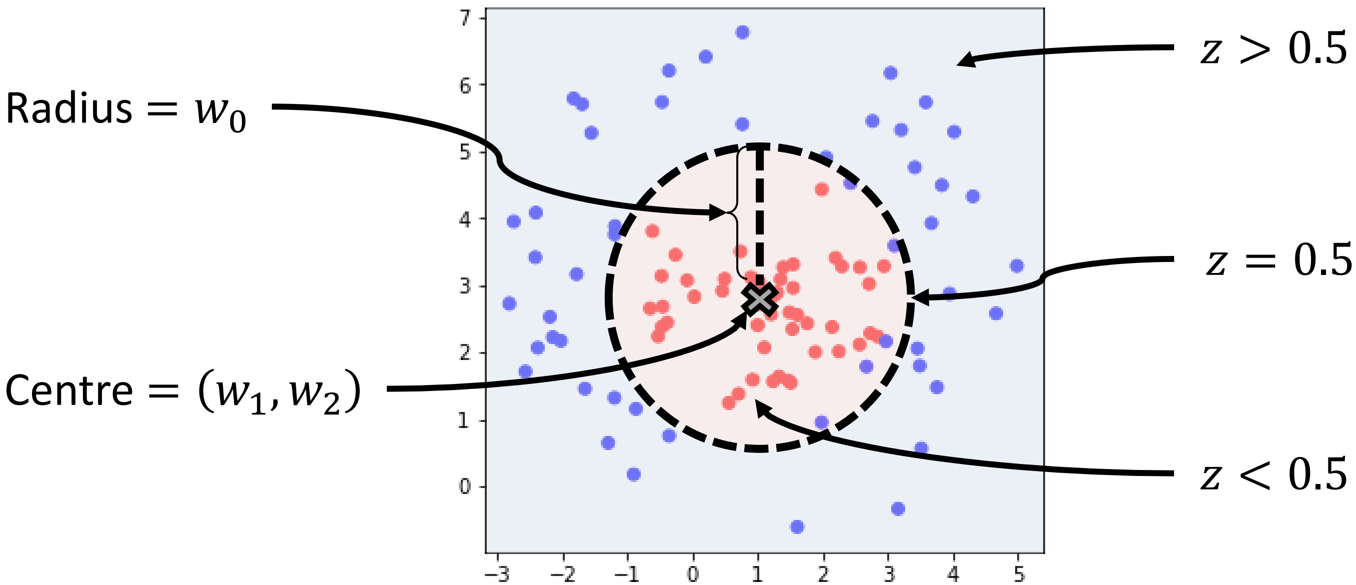
$$z = (x_1 - w_1)^2 + (x_2 - w_2)^2 - w_0^2$$

We would like wherever $z > 0.5$ to classify as **class 1**, and wherever $z < 0.5$ to classify as **class 0**. Hence, we apply a sigmoid on z.

$$\hat{y} = g(z) = \frac{1}{1 + e^{-z}}$$



Loss (binary cross-entropy loss)

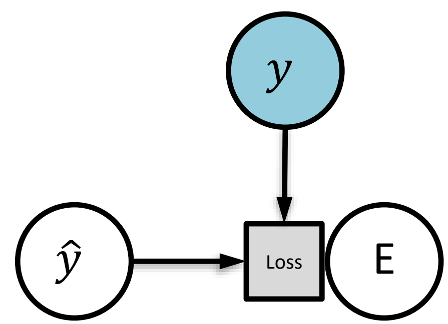


$$z = (x_1 - w_1)^2 + (x_2 - w_2)^2 - w_0^2$$

$$\hat{y} = g(z) = \frac{1}{1 + e^{-z}}$$

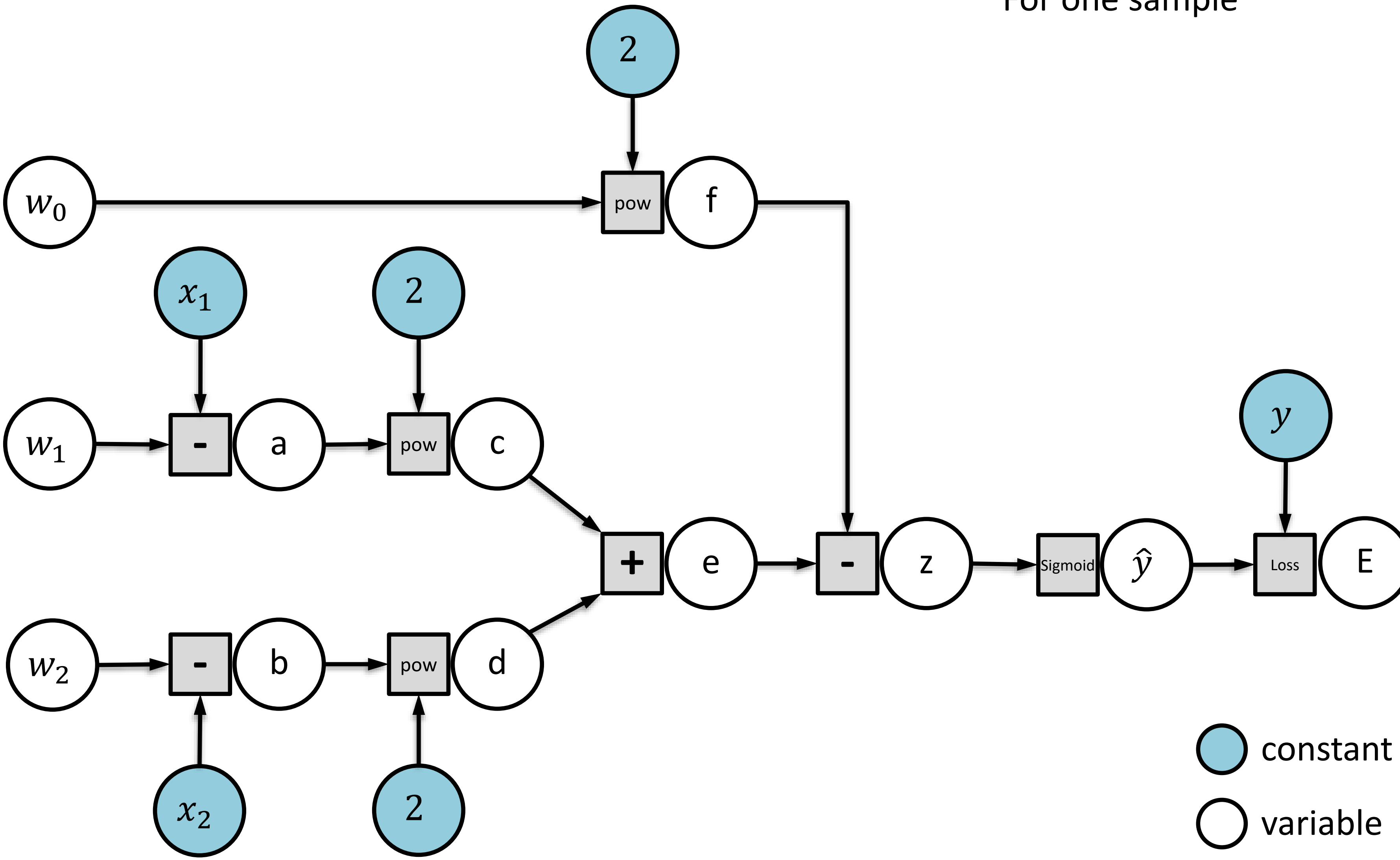
Finally, we would compare the output to the correct class using the cross-entropy loss we saw before:

$$Loss = -y \log(g(z)) - (1 - y) \log(1 - g(z))$$



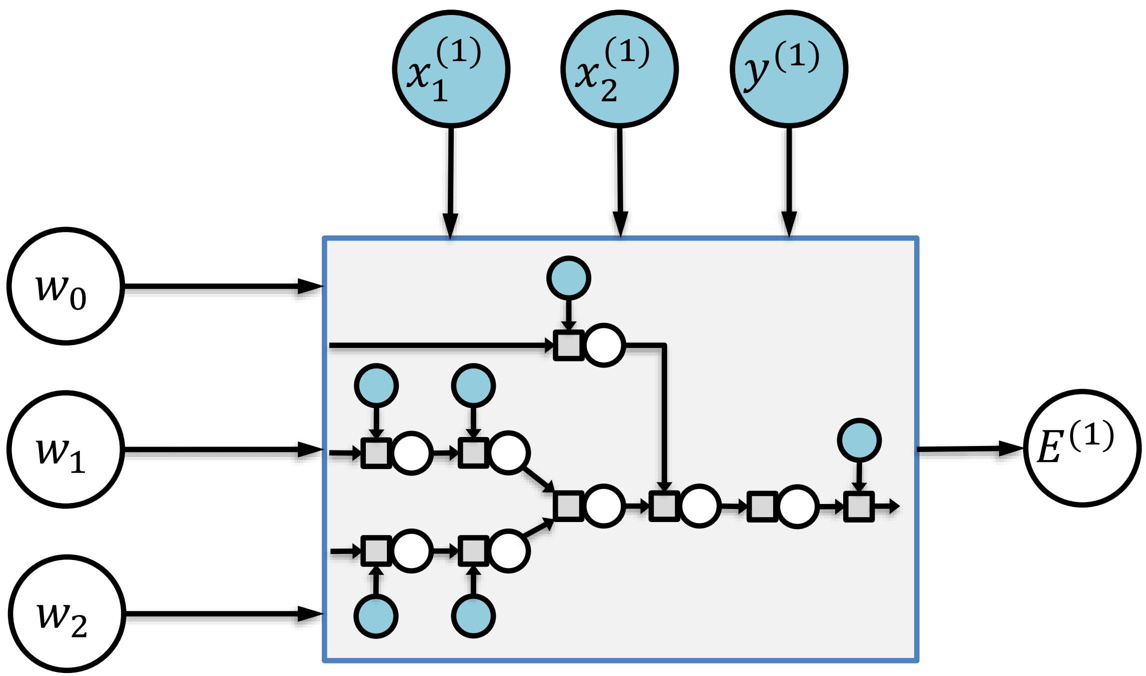
Complete Computational Graph

For one sample



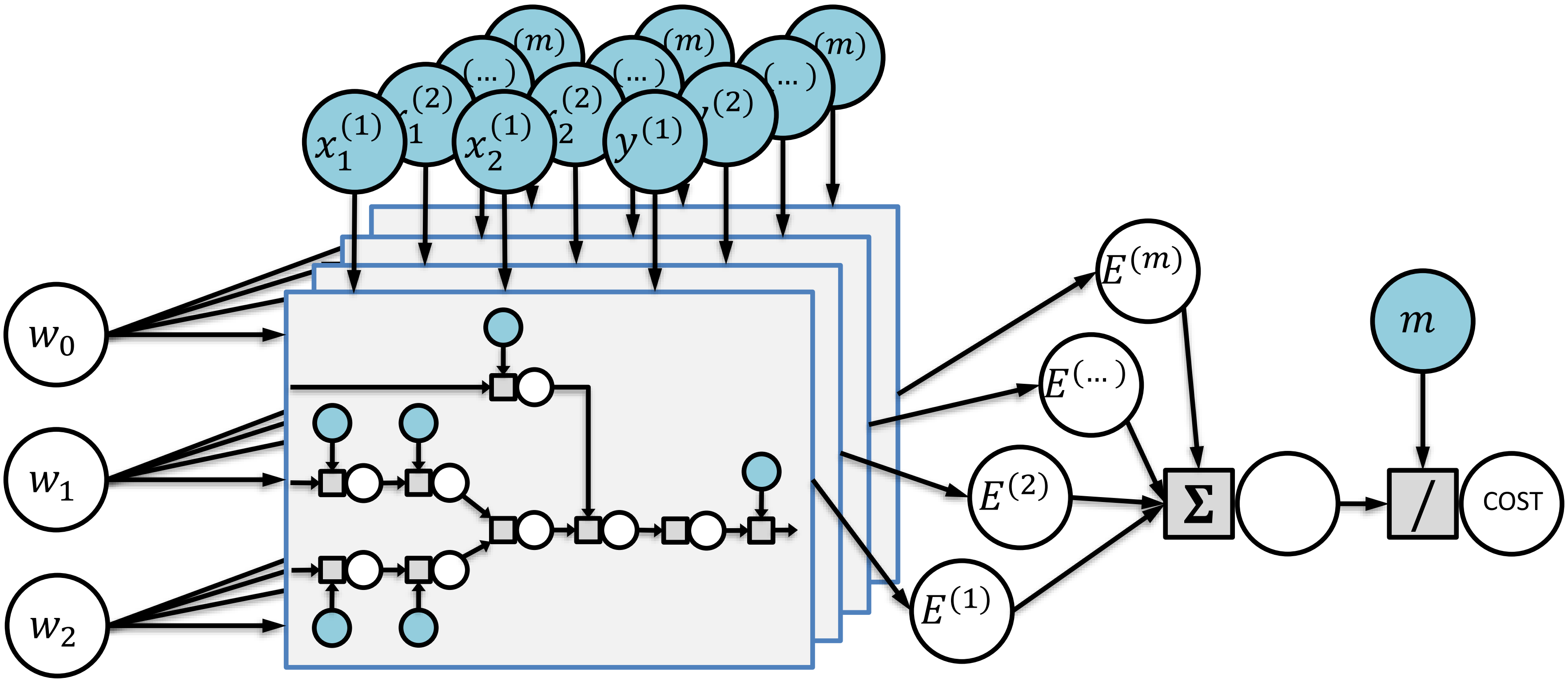
Complete Computational Graph

For one sample



Complete Computational Graph

Full batch of *m* samples



Gaining Efficiency

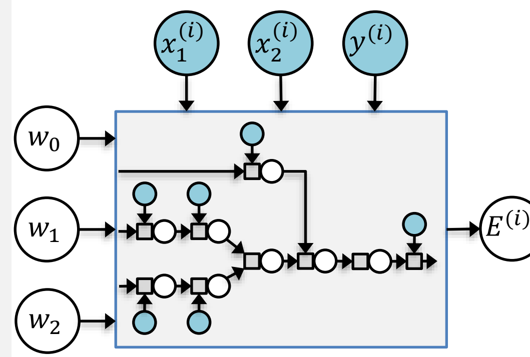
The derivative of the sum, equals the sum of derivatives… We can backpropagate errors for each sample individually, and accumulate the derivatives

Initialise gradients to zero

The backpropagated error for every point is accumulated to the derivative of each weight

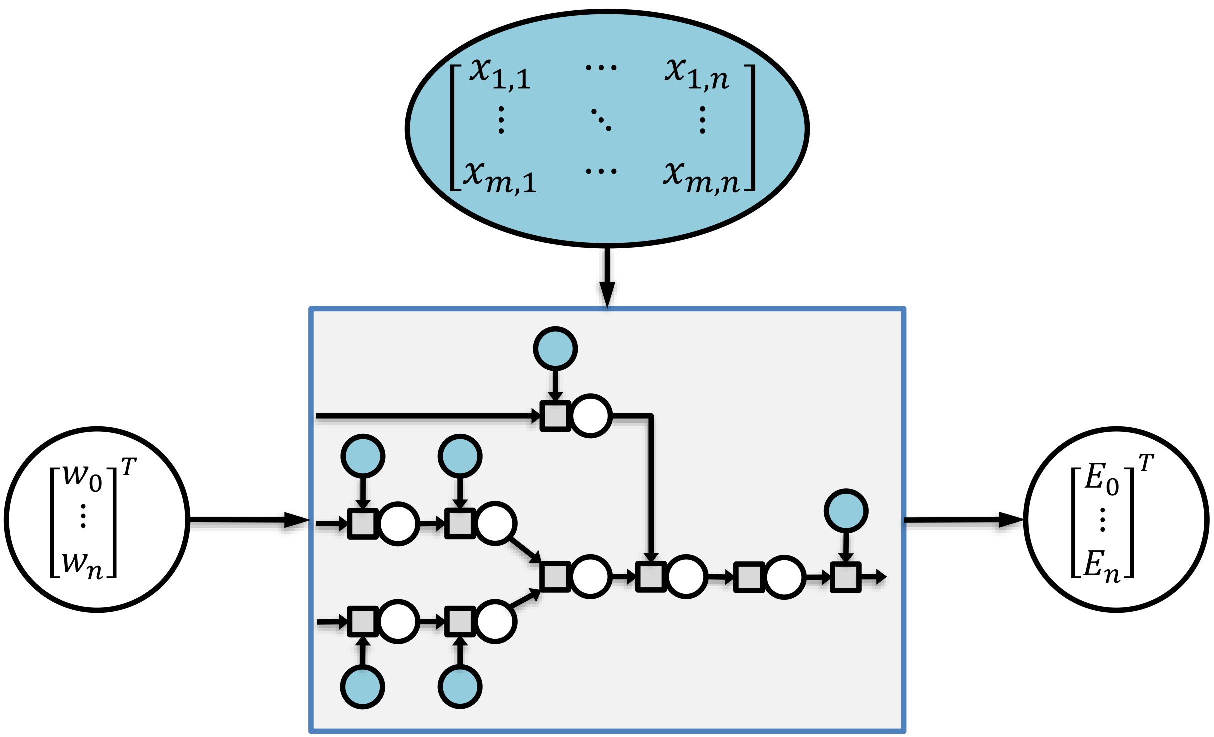
Remember to divide by the number of samples when applying gradient descent

```
for epoch in range(1, 1000):
    w0.zeroGradient()
    w1.zeroGradient()
    w2.zeroGradient()

    for x, y in trainingSamples:
        # Do forward pass
        # backpropagate error

    w0 = w0 + learningRate * w0.grad/m
    w1 = w1 + learningRate * w1.grad/m
    w2 = w2 + learningRate * w2.grad/m
```



Gaining Efficiency

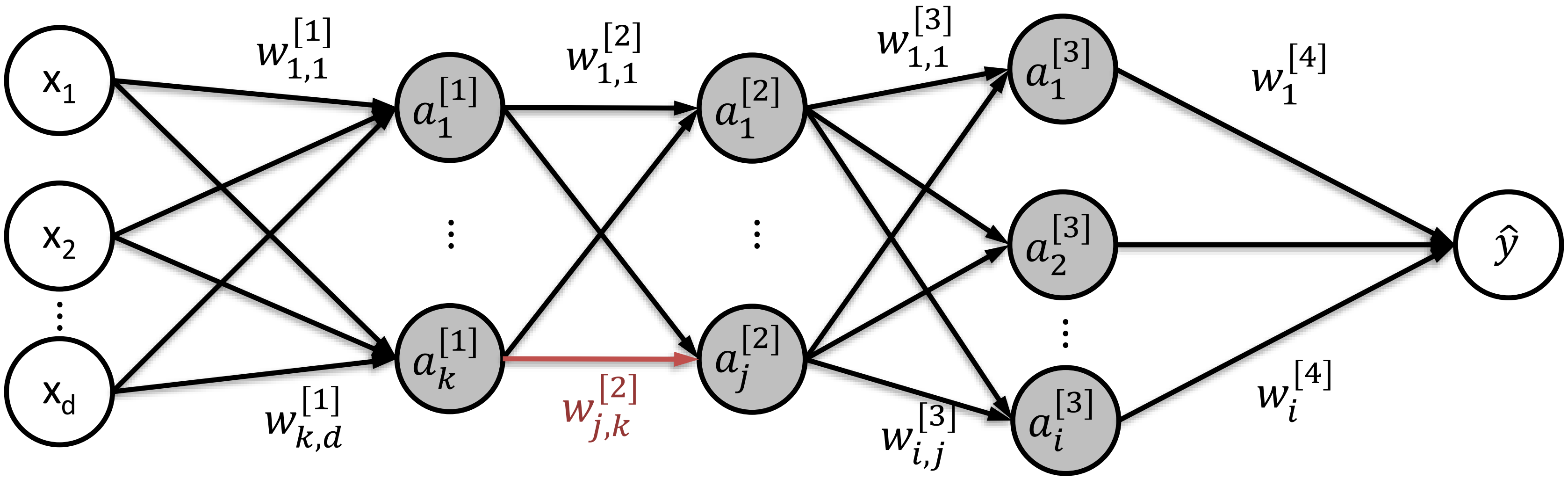
Vectorise data, and take advantage of the SIMD (Single Instruction, Multiple Data) capabilities of CPUs and GPUs



BACKPROPAGATION – TAKE TWO

Backpropagation Algorithm

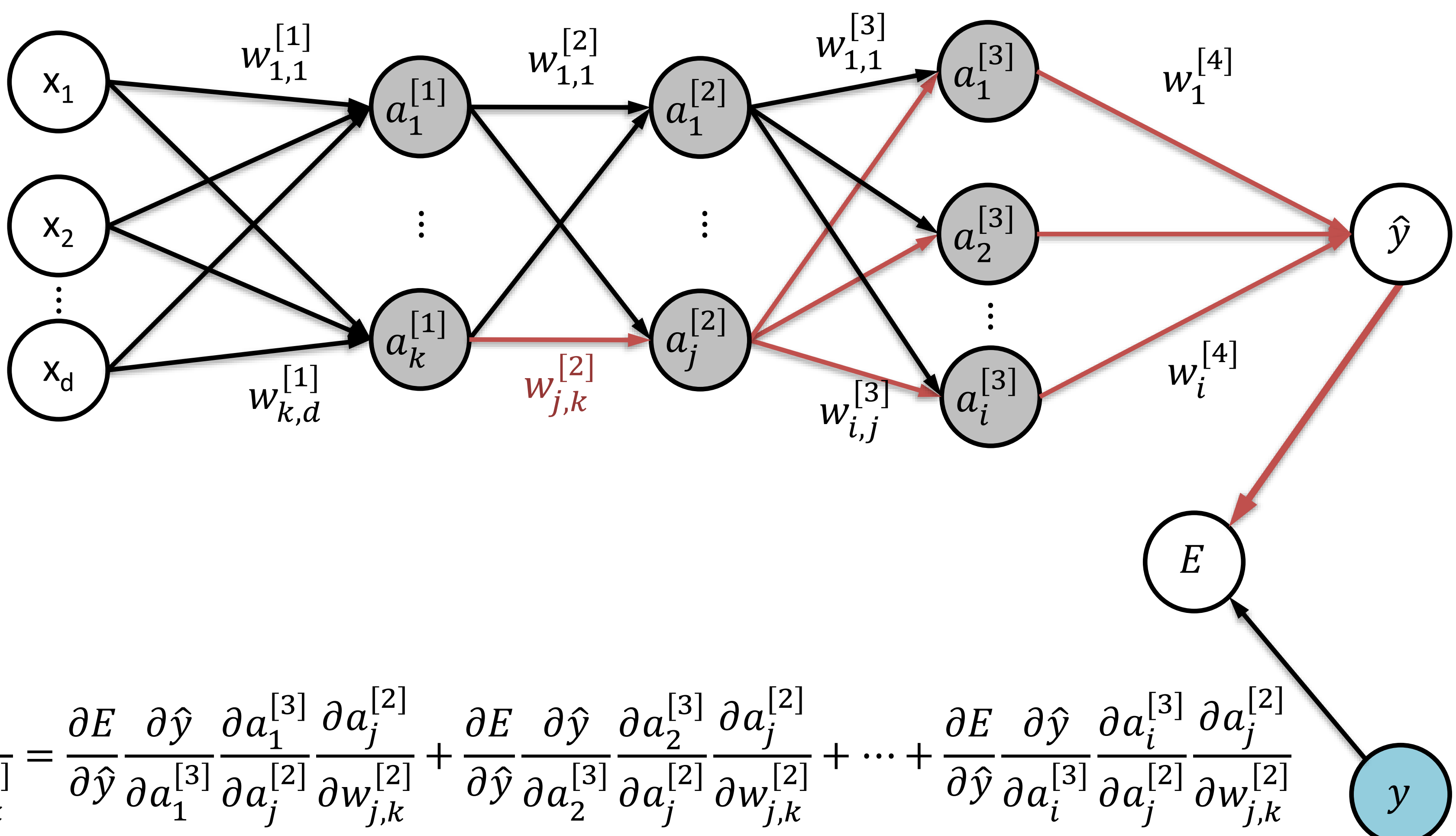
How should I change $w_{j,k}^{[2]}$?



1. Receive new observation $\mathbf{x} = [x_1, x_2, \dots, x_d]$ and target output y
2. Feed-forward: let the network calculate its output $\hat{y}$
3. Get the prediction $\hat{y}$ and calculate the error (loss) e.g. $E = \frac{1}{2}(\hat{y} - y)^2$
4. **Back-propagate error**: calculate how each of the weights contributed to this error

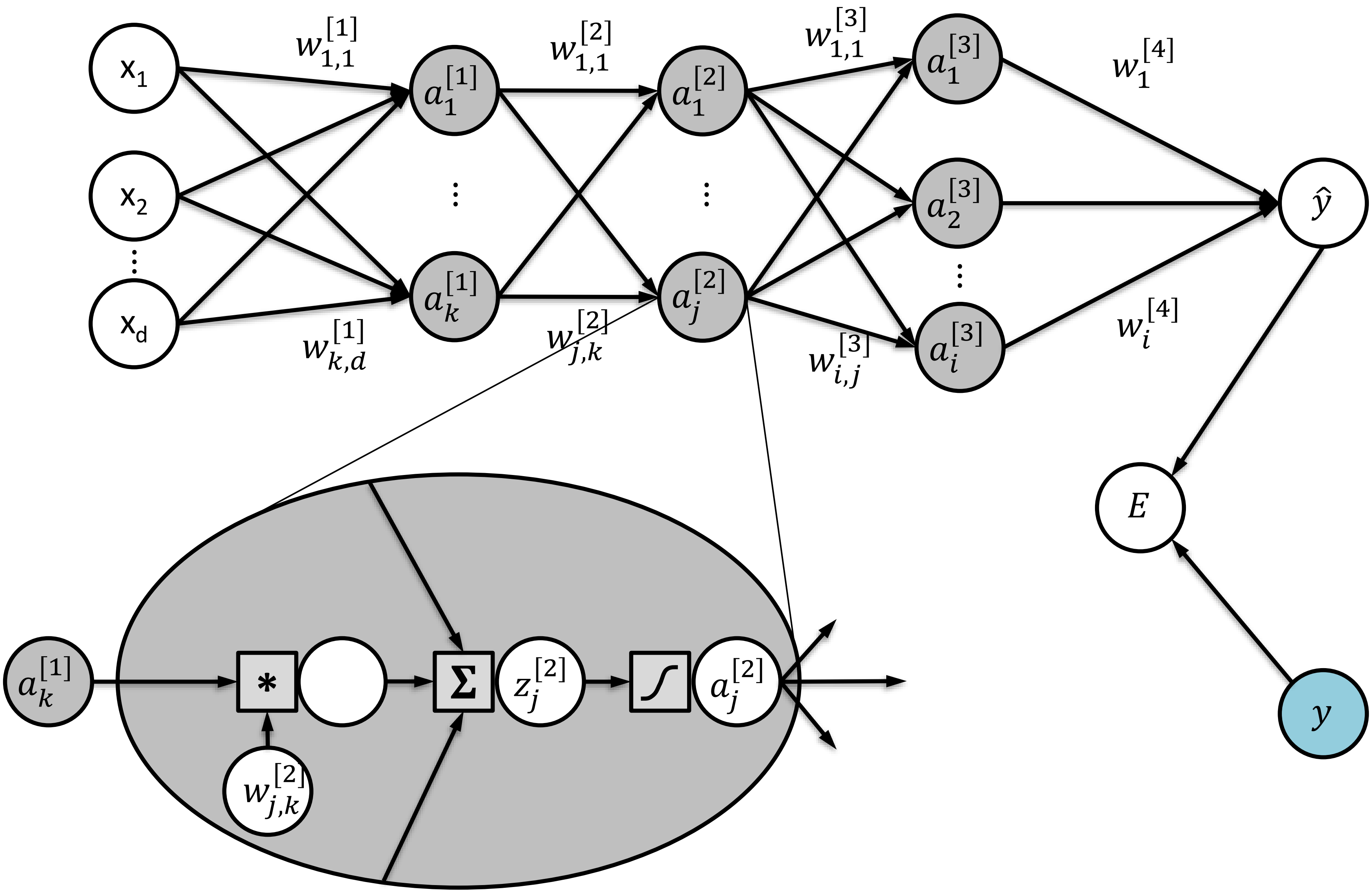
Backpropagation Algorithm

How should I change $w_{j,k}^{[2]}$?

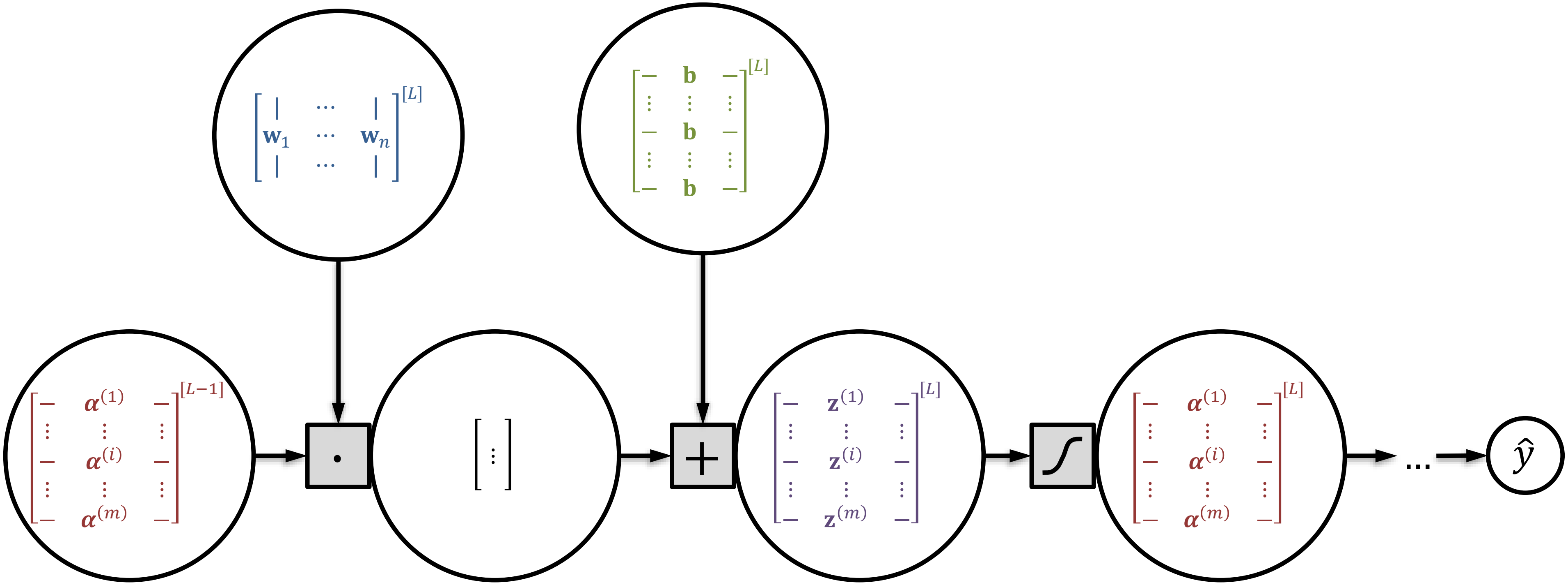


$$\frac{\partial E}{\partial w_{j,k}^{[2]}} = \frac{\partial E}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial a_1^{[3]}} \frac{\partial a_1^{[3]}}{\partial a_j^{[2]}} \frac{\partial a_j^{[2]}}{\partial w_{j,k}^{[2]}} + \frac{\partial E}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial a_2^{[3]}} \frac{\partial a_2^{[3]}}{\partial a_j^{[2]}} \frac{\partial a_j^{[2]}}{\partial w_{j,k}^{[2]}} + \cdots + \frac{\partial E}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial a_i^{[3]}} \frac{\partial a_i^{[3]}}{\partial a_j^{[2]}} \frac{\partial a_j^{[2]}}{\partial w_{j,k}^{[2]}}$$

Computation Graph of a NN



Computation Graph of a NN



In practice, all operations are vectorised and highly optimised to take advantage of the SIMD (Single Instruction, Multiple Data) capabilities of CPUs and GPUs

MATRIX DERIVATIVES

Derivatives with respect to a vector

Many times we need to calculate all the partial derivatives of a function whose input and output are both vectors.

For example, imagine the function $\boldsymbol{f}: \mathbb{R}^3 \longrightarrow \mathbb{R}^4$

$$\boldsymbol{y} = \boldsymbol{f}(\mathbf{x}) = \mathbf{W}\mathbf{x}$$

$$\boldsymbol{y} = \underset{(4\times 1)}{\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix}} = \underset{(4\times 3)}{\begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} \\ w_{2,1} & w_{2,2} & w_{2,3} \\ w_{3,1} & w_{3,2} & w_{3,3} \\ w_{4,1} & w_{4,2} & w_{4,3} \end{bmatrix}} \underset{(3\times 1)}{\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}}$$

$$\frac{d\boldsymbol{y}}{d\mathbf{x}} = ?$$

A full characterisation of the derivative of $\mathbf{y}$ with respect to $\mathbf{x}$ requires the partial derivative of **each component of $\mathbf{y}$** with respect to **each component of $\mathbf{x}$**

Derivatives with respect to a vector

A full characterisation of the derivative of $\mathbf{y}$ with respect to $\mathbf{x}$ requires the partial derivative of **each component of $\mathbf{y}$** with respect to **each component of $\mathbf{x}$**

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} = \begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} \\ w_{2,1} & w_{2,2} & w_{2,3} \\ w_{3,1} & w_{3,2} & w_{3,3} \\ w_{4,1} & w_{4,2} & w_{4,3} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

Let's compute one of these, e.g. the derivative of y_2 to x_3

$$y_2 = \sum_{j=1}^{3} w_{2,j} x_j$$

$$y_2 = w_{2,1}x_1 + w_{2,2}x_2 + w_{2,3}x_3$$

$$\frac{\partial y_2}{\partial x_3} = \frac{\partial}{\partial x_3}\left[w_{2,1}x_1 + w_{2,2}x_2 + w_{2,3}x_3\right]$$

$$\frac{\partial y_2}{\partial x_3} = 0 + 0 + \frac{\partial}{\partial x_3}\left[w_{2,3}x_3\right]$$

$$\frac{\partial y_2}{\partial x_3} = w_{2,3}$$

In general: $\dfrac{\partial y_i}{\partial x_j} = w_{i,j}$

Jacobian Matrix

We can organise all these partial derivatives into a new matrix called the **Jacobian matrix**.

$$\mathbf{y} = \mathbf{W}\mathbf{x}$$

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} = \begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} \\ w_{2,1} & w_{2,2} & w_{2,3} \\ w_{3,1} & w_{3,2} & w_{3,3} \\ w_{4,1} & w_{4,2} & w_{4,3} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

$$\frac{\partial y_i}{\partial x_j} = w_{i,j}$$

In this case the Jacobian matrix would be:

$$J = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} & \frac{\partial y_1}{\partial x_3} \\ \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2} & \frac{\partial y_2}{\partial x_3} \\ \frac{\partial y_3}{\partial x_1} & \frac{\partial y_3}{\partial x_2} & \frac{\partial y_3}{\partial x_3} \\ \frac{\partial y_4}{\partial x_1} & \frac{\partial y_4}{\partial x_2} & \frac{\partial y_4}{\partial x_3} \end{bmatrix} = \begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} \\ w_{2,1} & w_{2,2} & w_{2,3} \\ w_{3,1} & w_{3,2} & w_{3,3} \\ w_{4,1} & w_{4,2} & w_{4,3} \end{bmatrix}$$

$$\boxed{\frac{d\mathbf{y}}{d\mathbf{x}} = \mathbf{J}_{\mathbf{y}}(\mathbf{x}) = \mathbf{W}}$$

Jacobian Matrix

In general, for every function $\boldsymbol{f}: \mathbb{R}^m \longrightarrow \mathbb{R}^n$, the Jacobian matrix $\boldsymbol{J} \in \mathbb{R}^{n \times m}$ of $\boldsymbol{f}$ is defined such that $J_{i,j} = \frac{\partial f(\mathbf{x})_i}{\partial x_j}$

For a function $\boldsymbol{f}: \mathbb{R}^3 \longrightarrow \mathbb{R}^4$

The Jacobian matrix would be

$$\boldsymbol{J}_{\boldsymbol{f}}(\boldsymbol{x}) = \begin{bmatrix} \frac{\partial \boldsymbol{f}(\mathbf{x})_1}{\partial x_1} & \frac{\partial \boldsymbol{f}(\mathbf{x})_1}{\partial x_2} & \frac{\partial \boldsymbol{f}(\mathbf{x})_1}{\partial x_3} \\ \frac{\partial \boldsymbol{f}(\mathbf{x})_2}{\partial x_1} & \frac{\partial \boldsymbol{f}(\mathbf{x})_2}{\partial x_2} & \frac{\partial \boldsymbol{f}(\mathbf{x})_2}{\partial x_3} \\ \frac{\partial \boldsymbol{f}(\mathbf{x})_3}{\partial x_1} & \frac{\partial \boldsymbol{f}(\mathbf{x})_3}{\partial x_2} & \frac{\partial \boldsymbol{f}(\mathbf{x})_3}{\partial x_3} \\ \frac{\partial \boldsymbol{f}(\mathbf{x})_4}{\partial x_1} & \frac{\partial \boldsymbol{f}(\mathbf{x})_4}{\partial x_2} & \frac{\partial \boldsymbol{f}(\mathbf{x})_4}{\partial x_3} \end{bmatrix}$$

Always be careful with the numerator and denominator layout notation when doing matrix calculus!

What about row vectors?

$$\cancel{\boldsymbol{y} = \boldsymbol{f}(\mathbf{x}) = \mathbf{W}\mathbf{x}}$$

$$\cancel{\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} = \begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} \\ w_{2,1} & w_{2,2} & w_{2,3} \\ w_{3,1} & w_{3,2} & w_{3,3} \\ w_{4,1} & w_{4,2} & w_{4,3} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}}$$

$$\boldsymbol{y} = \mathbf{x}\mathbf{W}$$

$$\underset{(1 \times 4)}{\begin{bmatrix} y_1 & y_2 & y_3 & y_4 \end{bmatrix}} = \underset{(1 \times 3)}{\begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix}} \underset{(3 \times 4)}{\begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} & w_{1,4} \\ w_{2,1} & w_{2,2} & w_{2,3} & w_{2,4} \\ w_{3,1} & w_{3,2} & w_{3,3} & w_{3,4} \end{bmatrix}}$$

$$\frac{d\mathbf{y}}{d\mathbf{x}} = ?$$

Work this out at home. You should be able to show that the derivative (Jacobian) in this case is also equal to $\mathbf{W}$

Dealing with more than two dimensions

Convert to the row equivalent

Let's consider now the problem of computing the derivative with respect to the matrix $\mathbf{W}$

$$\boldsymbol{y} = \mathbf{W}\mathbf{x}$$

$$\begin{array}{ccccc} & (4 \times 1) & & (4 \times 3) & (3 \times 1) \\ \boldsymbol{y} = & \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} & = & \begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} \\ w_{2,1} & w_{2,2} & w_{2,3} \\ w_{3,1} & w_{3,2} & w_{3,3} \\ w_{4,1} & w_{4,2} & w_{4,3} \end{bmatrix} & \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \end{array}$$

$$\frac{d\boldsymbol{y}}{d\mathbf{W}} = ?$$

Let's define a 3D tensor $\mathbf{T}$, with elements: $t_{i,j,k} = \dfrac{\partial y_i}{\partial w_{k,j}}$

A full characterisation of the derivative of $\mathbf{y}$ with respect to $\mathbf{W}$ requires the partial derivative of **each component of $\mathbf{y}$** with respect to **each component of $\mathbf{W}$**

$$\begin{bmatrix} \frac{\partial y_1}{\partial w_{1,1}} & \frac{\partial y_1}{\partial w_{1,2}} & \frac{\partial y_1}{\partial w_{1,3}} \\ \frac{\partial y_2}{\partial w_{1,1}} & \frac{\partial y_2}{\partial w_{1,2}} & \frac{\partial y_2}{\partial w_{1,3}} \\ \frac{\partial y_3}{\partial w_{1,1}} & \frac{\partial y_3}{\partial w_{1,2}} & \frac{\partial y_3}{\partial w_{1,3}} \\ \frac{\partial y_4}{\partial w_{1,1}} & \frac{\partial y_4}{\partial w_{1,2}} & \frac{\partial y_4}{\partial w_{1,3}} \end{bmatrix} \begin{bmatrix} \frac{\partial y_1}{\partial w_{2,1}} & \frac{\partial y_1}{\partial w_{2,2}} & \frac{\partial y_1}{\partial w_{2,3}} \\ \frac{\partial y_2}{\partial w_{2,1}} & \frac{\partial y_2}{\partial w_{2,2}} & \frac{\partial y_2}{\partial w_{2,3}} \\ \frac{\partial y_3}{\partial w_{2,1}} & \frac{\partial y_3}{\partial w_{2,2}} & \frac{\partial y_3}{\partial w_{2,3}} \\ \frac{\partial y_4}{\partial w_{2,1}} & \frac{\partial y_4}{\partial w_{2,2}} & \frac{\partial y_4}{\partial w_{2,3}} \end{bmatrix} \begin{bmatrix} \frac{\partial y_1}{\partial w_{3,1}} & \frac{\partial y_1}{\partial w_{3,2}} & \frac{\partial y_1}{\partial w_{3,3}} \\ \frac{\partial y_2}{\partial w_{3,1}} & \frac{\partial y_2}{\partial w_{3,2}} & \frac{\partial y_2}{\partial w_{3,3}} \\ \frac{\partial y_3}{\partial w_{3,1}} & \frac{\partial y_3}{\partial w_{3,2}} & \frac{\partial y_3}{\partial w_{3,3}} \\ \frac{\partial y_4}{\partial w_{3,1}} & \frac{\partial y_4}{\partial w_{3,2}} & \frac{\partial y_4}{\partial w_{3,3}} \end{bmatrix} \begin{bmatrix} \frac{\partial y_1}{\partial w_{4,1}} & \frac{\partial y_1}{\partial w_{4,2}} & \frac{\partial y_1}{\partial w_{4,3}} \\ \frac{\partial y_2}{\partial w_{4,1}} & \frac{\partial y_2}{\partial w_{4,2}} & \frac{\partial y_2}{\partial w_{4,3}} \\ \frac{\partial y_3}{\partial w_{4,1}} & \frac{\partial y_3}{\partial w_{4,2}} & \frac{\partial y_3}{\partial w_{4,3}} \\ \frac{\partial y_4}{\partial w_{4,1}} & \frac{\partial y_4}{\partial w_{4,2}} & \frac{\partial y_4}{\partial w_{4,3}} \end{bmatrix} \quad (4 \times 3 \times 4)$$

Dealing with more than two dimensions

Convert to the row equivalent

Same as before, let's compute just one of these components, e.g. the derivative of y_2 to $w_{1,3}$

$$y_2 = \sum_{j=1}^{3} w_{2,j} x_j$$

$$y_2 = w_{2,1}x_1 + w_{2,2}x_2 + w_{2,3}x_3$$

$$\frac{\partial y_2}{\partial w_{1,3}} = \frac{\partial}{\partial w_{1,3}} \left[w_{2,1}x_1 + w_{2,2}x_2 + w_{2,3}x_3 \right]$$

$$\frac{\partial y_2}{\partial w_{1,3}} = 0$$

The only derivatives y_2 that are non-zero are the ones involving the second column of $\mathbf{W}$, the elements: $w_{i,2}$

For example:

$$\frac{\partial y_2}{\partial w_{2,3}} = \frac{\partial}{\partial w_{2,3}} \left[w_{2,1}x_1 + w_{2,2}x_2 + w_{2,3}x_3 \right] = x_3$$

In general:

$$\frac{\partial y_i}{\partial w_{i,j}} = x_j$$

Dealing with more than two dimensions

Convert to the row equivalent

Most elements will be zero, except for the elements for which $i = k$.

$$t_{i,j,k} = \begin{cases} x_j & , \text{if } i = k \\ 0 & , \text{otherwise} \end{cases}$$

$$\begin{bmatrix} \frac{\partial y_1}{\partial w_{1,1}} & \frac{\partial y_1}{\partial w_{1,2}} & \frac{\partial y_1}{\partial w_{1,3}} \\ \frac{\partial y_2}{\partial w_{1,1}} & \frac{\partial y_2}{\partial w_{1,2}} & \frac{\partial y_2}{\partial w_{1,3}} \\ \frac{\partial y_3}{\partial w_{1,1}} & \frac{\partial y_3}{\partial w_{1,2}} & \frac{\partial y_3}{\partial w_{1,3}} \\ \frac{\partial y_4}{\partial w_{1,1}} & \frac{\partial y_4}{\partial w_{1,2}} & \frac{\partial y_4}{\partial w_{1,3}} \end{bmatrix} \begin{bmatrix} \frac{\partial y_1}{\partial w_{2,1}} & \frac{\partial y_1}{\partial w_{2,2}} & \frac{\partial y_1}{\partial w_{2,3}} \\ \vdots & \vdots & \vdots \\ \frac{\partial y_4}{\partial w_{2,1}} & \frac{\partial y_4}{\partial w_{2,2}} & \frac{\partial y_4}{\partial w_{2,3}} \end{bmatrix} \begin{bmatrix} \frac{\partial y_1}{\partial w_{3,1}} & \frac{\partial y_1}{\partial w_{3,2}} & \frac{\partial y_1}{\partial w_{3,3}} \\ \vdots & \vdots & \vdots \\ \frac{\partial y_4}{\partial w_{3,1}} & \frac{\partial y_4}{\partial w_{3,2}} & \frac{\partial y_4}{\partial w_{3,3}} \end{bmatrix} \begin{bmatrix} \frac{\partial y_1}{\partial w_{4,1}} & \frac{\partial y_1}{\partial w_{4,2}} & \frac{\partial y_1}{\partial w_{4,3}} \\ \vdots & \vdots & \vdots \\ \frac{\partial y_4}{\partial w_{4,1}} & \frac{\partial y_4}{\partial w_{4,2}} & \frac{\partial y_4}{\partial w_{4,3}} \end{bmatrix}$$

$$= \begin{bmatrix} x_1 & x_2 & x_3 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ x_1 & x_2 & x_3 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ x_1 & x_2 & x_3 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ x_1 & x_2 & x_3 \end{bmatrix}$$

If $y_{i,:}$ is the i^{th} element of $\mathbf{y}$ and $W_{i,:}$ is the i^{th} row of $\mathbf{W}$ then $$\frac{\partial y_i}{\partial W_{i,:}} = \mathbf{x}$$

All the non-trivial portion of this tensor can be stored in a compact way in a 2D matrix

Multiple data points

Let's now use multiple row-vector samples $x^{(i)}$, stacked together to form a matrix $\mathbf{X}$.

$$\mathbf{Y} = \mathbf{XW}$$

If $Y_{i,:}$ is the *i*

$$\frac{\partial Y_{i,:}}{\partial X_{i,:}} = \mathbf{W}$$

Work this out at home

and $\quad \dfrac{\partial Y_{i,:}}{\partial X_{j,:}} =? \quad if\ i \neq j$

The chain rule

$$\mathbf{y} = \mathbf{V}\mathbf{W}\mathbf{x} \qquad \frac{\mathrm{d}\mathbf{y}}{\mathrm{d}\mathbf{x}} = \mathbf{V}\mathbf{W}$$

$$\mathbf{z} = \mathbf{W}\mathbf{x} \qquad \frac{\mathrm{d}\mathbf{z}}{\mathrm{d}\mathbf{x}} = \mathbf{W}$$

$$\mathbf{y} = \mathbf{V}\mathbf{z} \qquad \frac{\mathrm{d}\mathbf{y}}{\mathrm{d}\mathbf{z}} = \mathbf{V}$$

$$\frac{\mathrm{d}\mathbf{y}}{\mathrm{d}\mathbf{x}} = \frac{\mathrm{d}\mathbf{y}}{\mathrm{d}\mathbf{z}}\frac{\mathrm{d}\mathbf{z}}{\mathrm{d}\mathbf{x}} = \mathbf{V}\mathbf{W}$$

"*Vector, Matrix, and Tensor Derivatives*",
Erik Learned-Miller
http://cs231n.stanford.edu/vecDerivs.pdf

Pytorch made easy

```
import torch
dtype = torch.float
device = torch.device("cpu")
N, D_in, H, D_out = 64, 1000, 100, 10          # N is batch size; D_in is input dimension;  H is hidden dimension; D_out is output dimension.
# Create random Tensors to hold input and outputs.
# Setting requires_grad=False indicates that we do not need to compute gradients with respect to these Tensors during the backward pass.
x = torch.randn(N, D_in, device=device, dtype= torch.float)
y = torch.randn(N, D_out, device=device, dtype= torch.float)
# Create random Tensors for weights.  Setting requires_grad=True indicates that we want to compute gradients with  respect to these Tensors during the backward pass.
w1 = torch.randn(D_in, H, device=device, dtype= torch.float, requires_grad=True)
w2 = torch.randn(H, D_out, device=device, dtype= torch.float, requires_grad=True)
learning_rate = 1e-6
for t in (500):
    # Forward pass: compute predicted y using operations on Tensors; these  are exactly the same operations we used to compute the forward pass using  Tensors, but
    # we do not need to keep references to intermediate values since  we are not implementing the backward pass by hand.
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    # Compute and print loss using operations on Tensors.  Now loss is a Tensor of shape (1,), loss.item() gets the scalar value held in the loss.
    loss = (y_pred - y).pow(2).sum()
    # Use autograd to compute the backward pass. This call will compute the  gradient of loss with respect to all Tensors with requires_grad=True.  After this call
    # w1.grad and w2.grad will be Tensors holding the gradient of the loss with respect to w1 and w2 respectively.
    loss.backward()
    # Manually update weights using gradient descent. Wrap in torch.no_grad()  because weights have requires_grad=True, but we don't need to track this  in autograd.
    with torch.no_grad():
        w1 -= learning_rate * w1.grad
        w2 -= learning_rate * w2.grad
        # Manually zero the gradients after updating weights
        w1.grad.zero_()
        w2.grad.zero_()
```

Summary

- Backpropagation computes the **gradient** of the **loss function** with respect to the **weights** of the network for a single input–output example
- Auto Differentiation (AutoGrad) is at the core of modern deep learning frameworks, and enables efficient backpropagation schemes
  - Single pass process to calculate all needed derivatives
  - The key is that the process is always local (children nodes to parent nodes)
  - Scalable: we only need to compute stuff once, for all variables
  - Flexible: we can define new models easily (usually transparent from the end user)
  - Vectorizable: use matrix calculus

Still Not A Learning algorithm

- We know how to compute error derivatives for every weight on a single training point
- We got an idea about how to extend this for a whole batch of points
- We still need to see
  - **Loss functions**: How to measure our error? This depends on the task we want to solve.
  - **Activation functions**: What kind of neurons are there (neurons are defined by their integration and activation functions)?
  - **Architectures**: How to combine neurons together to build meaningful models?
  - **Optimisation**: Is batch gradient descent the best way to use these error derivatives to discover a good set of weights?
  - **Regularisation**: How do we make sure we do not overfit?
  - **Initialisation**: Where do we start our search?

More Information

- Some material on these slides has been adapted from various sources including the following highly recommended ones:
  - Andrew Ng's *Machine Learning Course*, Coursera
    https://www.coursera.org/course/ml
  - Andrew Ng's *Deep Learning Specialization,* Coursera
    https://www.coursera.org/specializations/deep-learning
  - Victor Lavrenko's *Machine Learning* Course
    https://www.youtube.com/channel/UCs7alOMRnxhzfKAJ4JjZ7Wg
  - Fei Fei Li and Andrej Karpathy's *Convolutional Neural Networks for Visual Recognition*
    http://cs231n.stanford.edu/
  - Geoff Hinton's *Neural Networks for Machine Learning,* Coursera
    https://www.coursera.org/learn/neural-networks
  - Luis Serrano's introductory videos
    https://www.youtube.com/channel/UCgBncpylJ1kiVaPyP-PZauQ