Introduction to Neural Networks (I)

Perceptron, Multilayer Feedforward Networks, Backpropagation

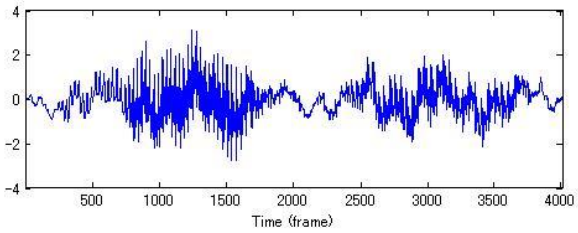
Dimosthenis Karatzas (dimos@cvc.uab.es)

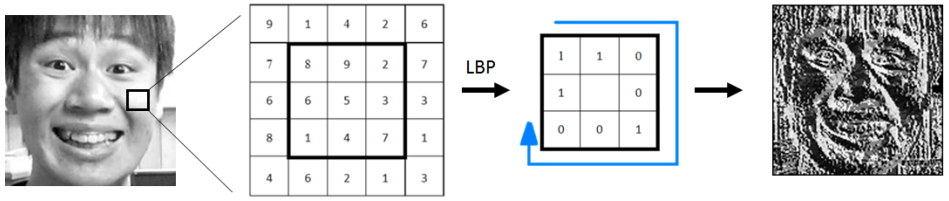
What have we learnt up to now?

$$y = \mathrm{sgn}(\mathbf{w}^T\mathbf{x} + \mathbf{b})$$

Machine learning solved: it's all about finding the right weights **w**

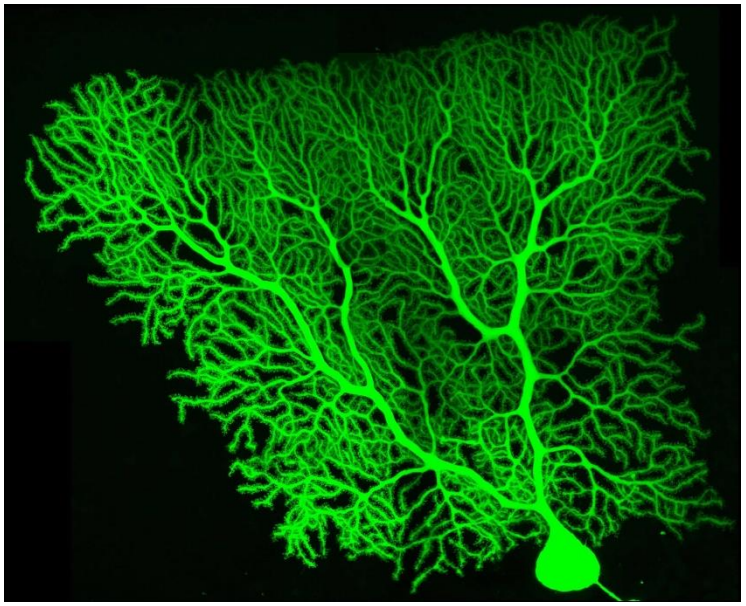
With one handicap: this only works *IF* you have chosen correctly your features **x**

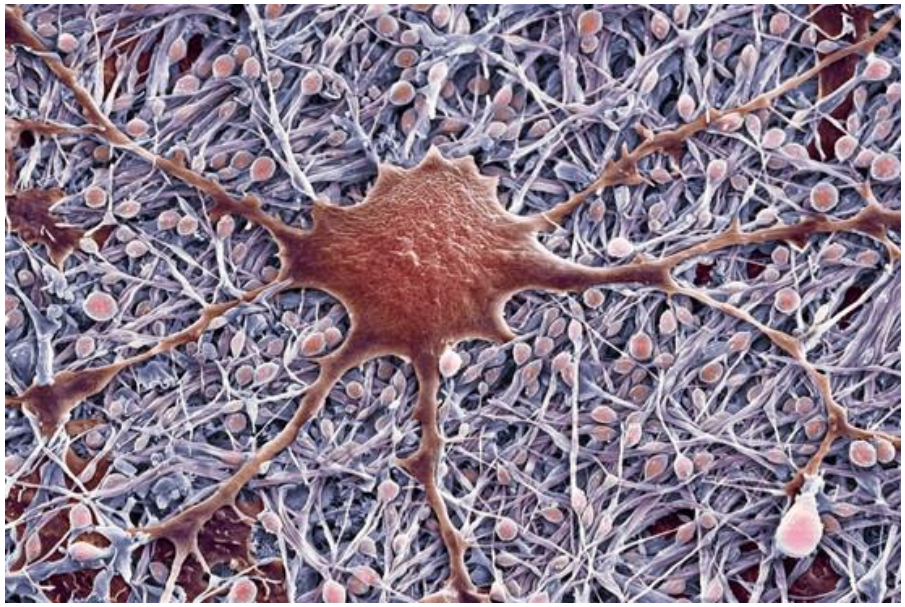




It would be nice to be able to "learn" what are the best features to use automatically: come up with a universal (not domain-dependent) algorithm that is able to represent any kind of data in the best way for each problem.

Neurons





The human brain comprises about 10^{11} neurons and 10^{14} synapses

A hugely parallel machine that has an effective bandwidth to stored knowledge much better than a modern workstation

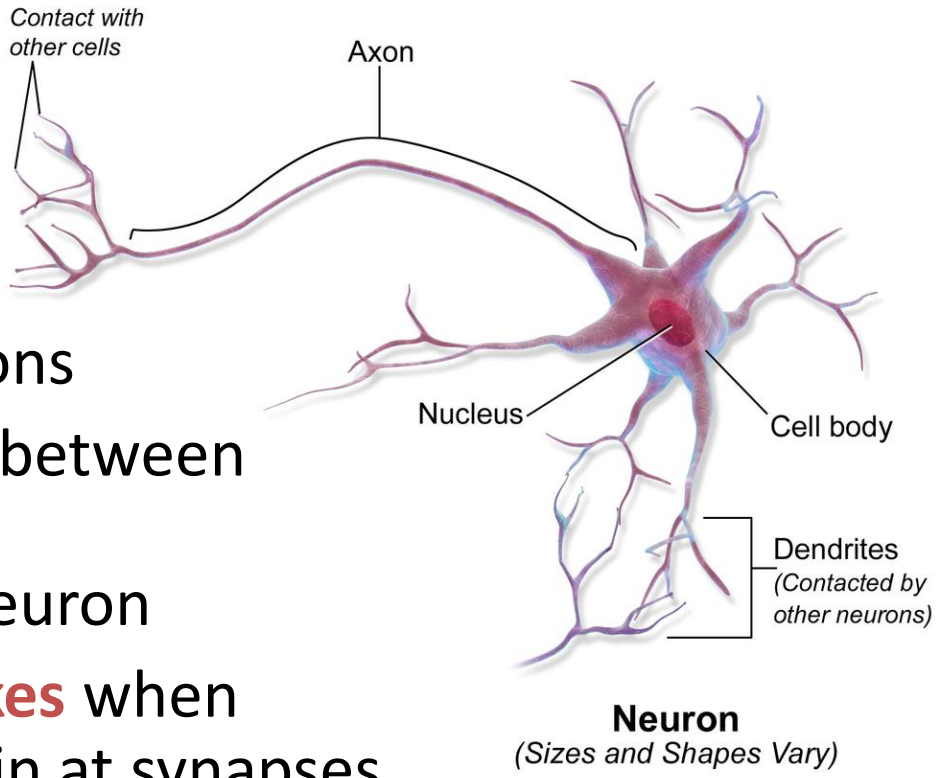
Neurons

Axon: branches and sends messages to other neurons

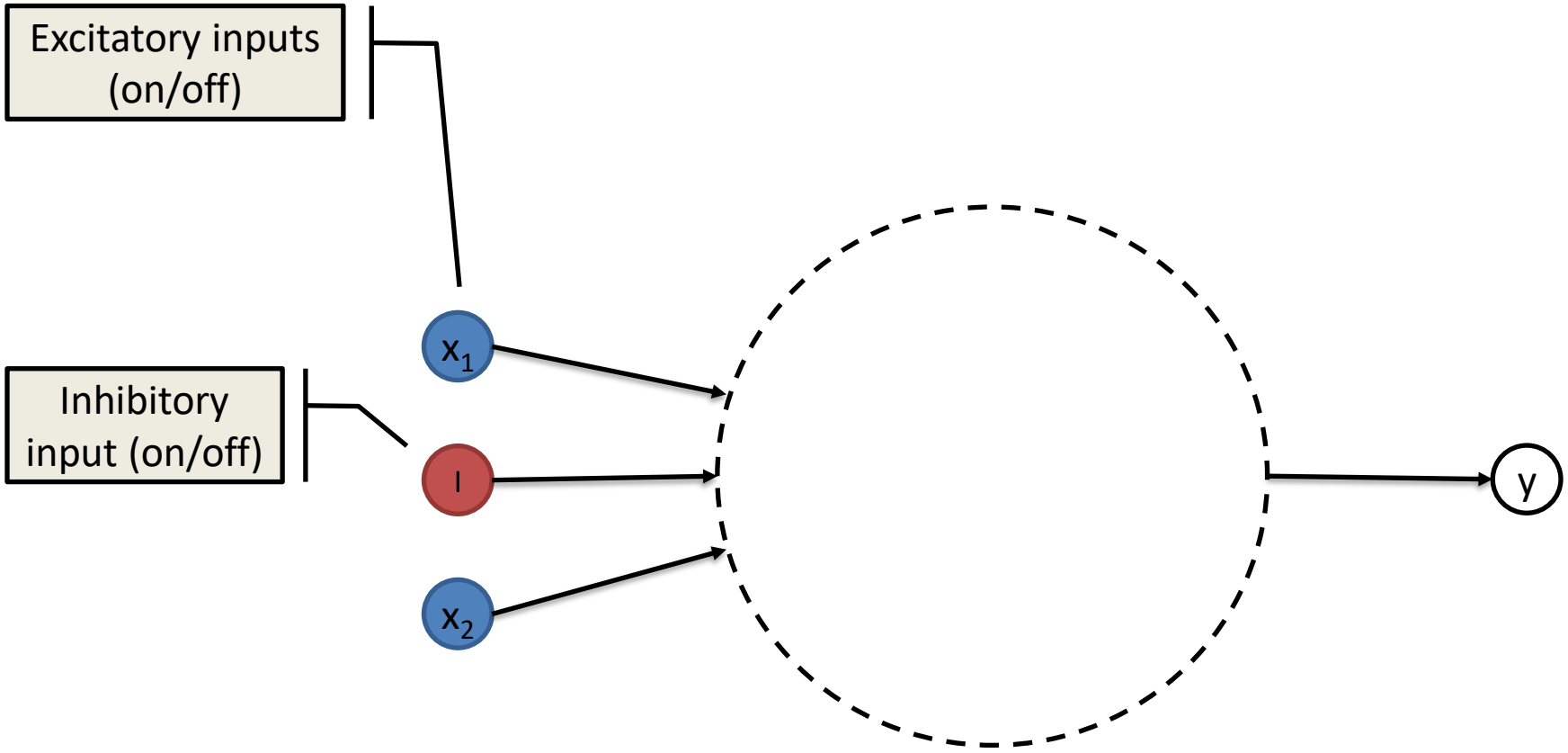
Dendritic tree: receives messages from other neurons

Synapses: are connections between an axon of a neuron and a dendritic tree of another neuron

A neuron can generate **spikes** when enough charge has flowed in at synapses (depolarise the "*axon hillock*")

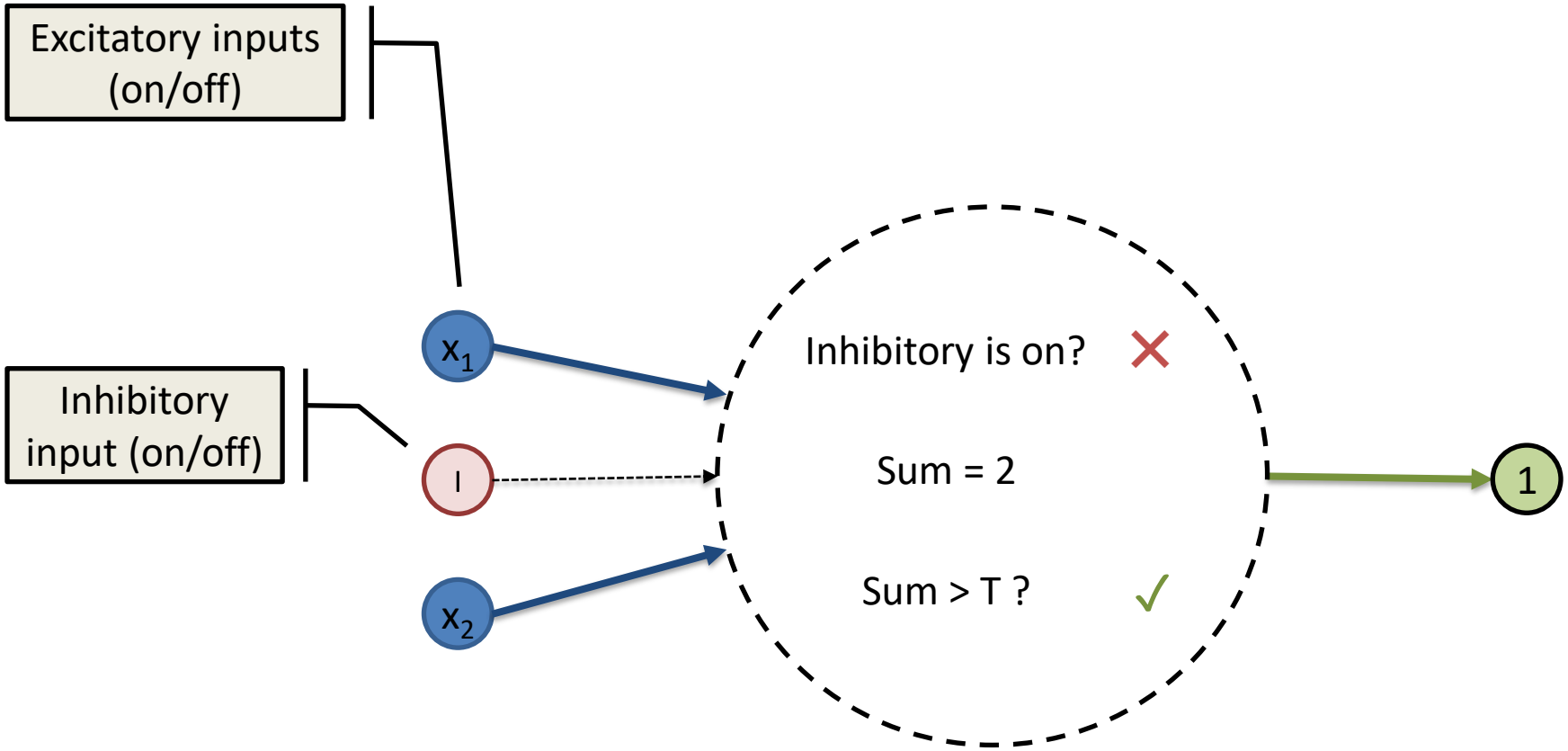


McCulloch-Pitts model (1943)



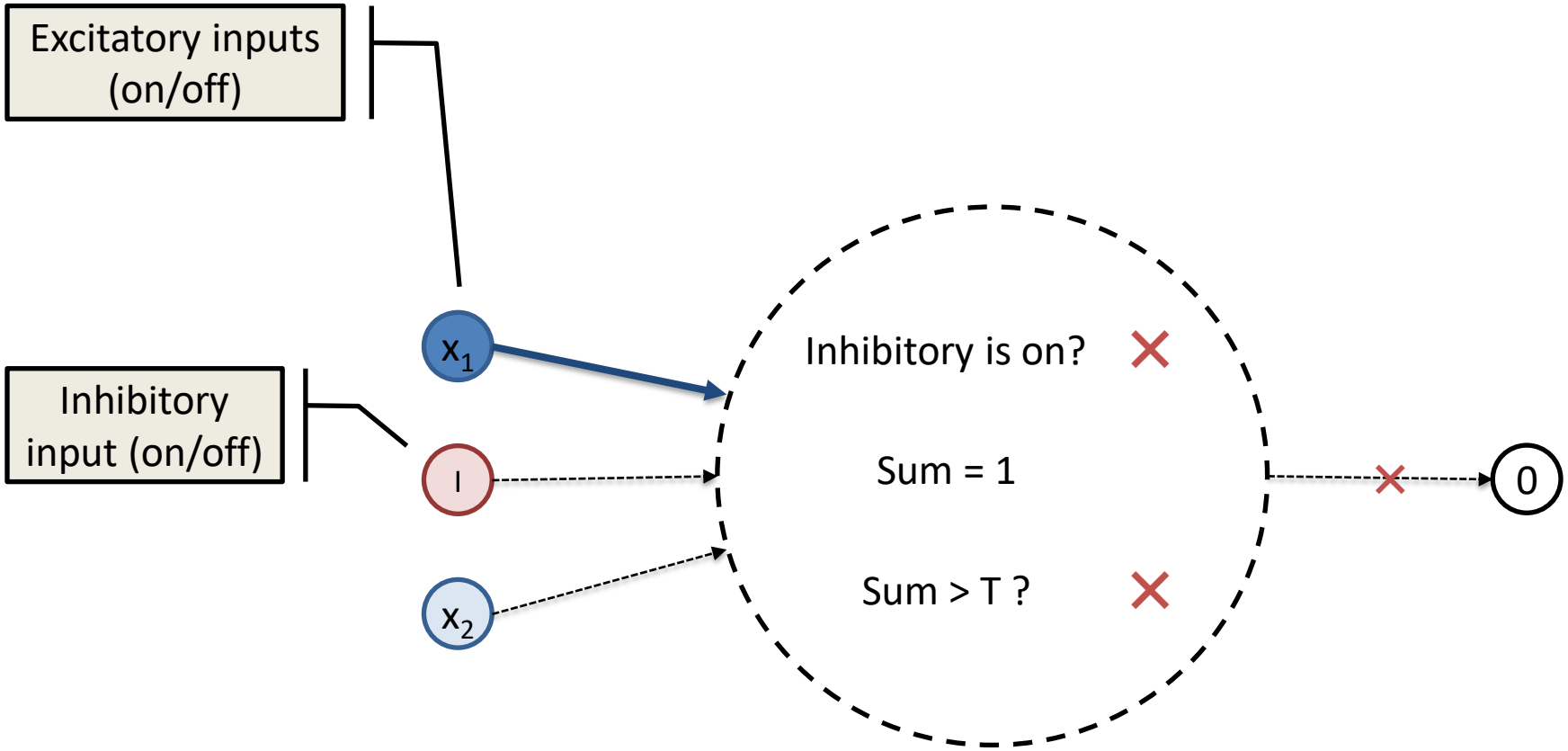
1. Binary output
2. Neuron fires if sum of activated excitatory neurons is above a threshold T (original version T = 1)
3. But any activated inhibitory neuron prevents activation

McCulloch-Pitts model (1943)



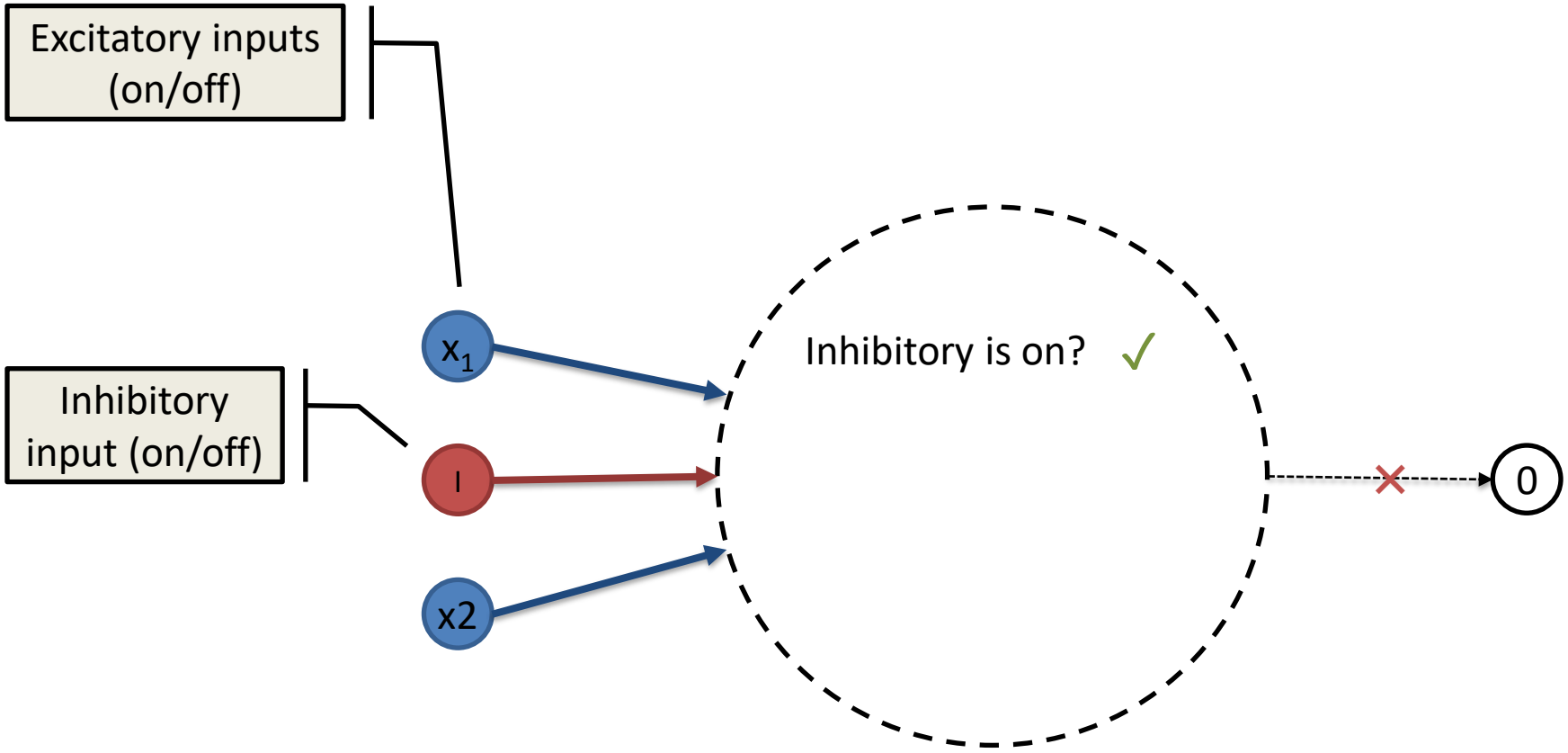
1. Binary output
2. Neuron fires if sum of activated excitatory neurons is above a threshold T (original version T = 1)
3. But any activated inhibitory neuron prevents activation

McCulloch-Pitts model (1943)



1. Binary output
2. Neuron fires if sum of activated excitatory neurons is above a threshold T (original version T = 1)
3. But any activated inhibitory neuron prevents activation

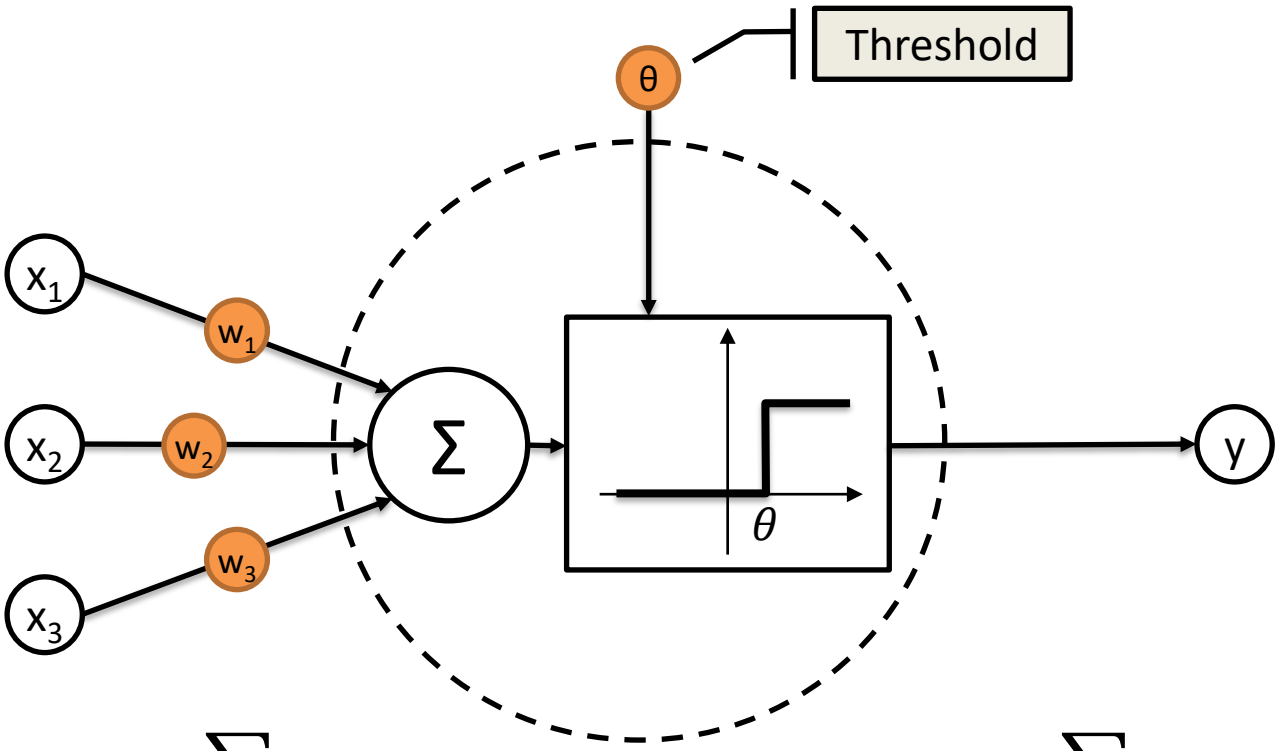
McCulloch-Pitts model (1943)



1. Binary output
2. Neuron fires if sum of activated excitatory neurons is above a threshold T (original version T = 1)
3. But any activated inhibitory neuron prevents activation

PERCEPTRON

Binary Threshold Neurons

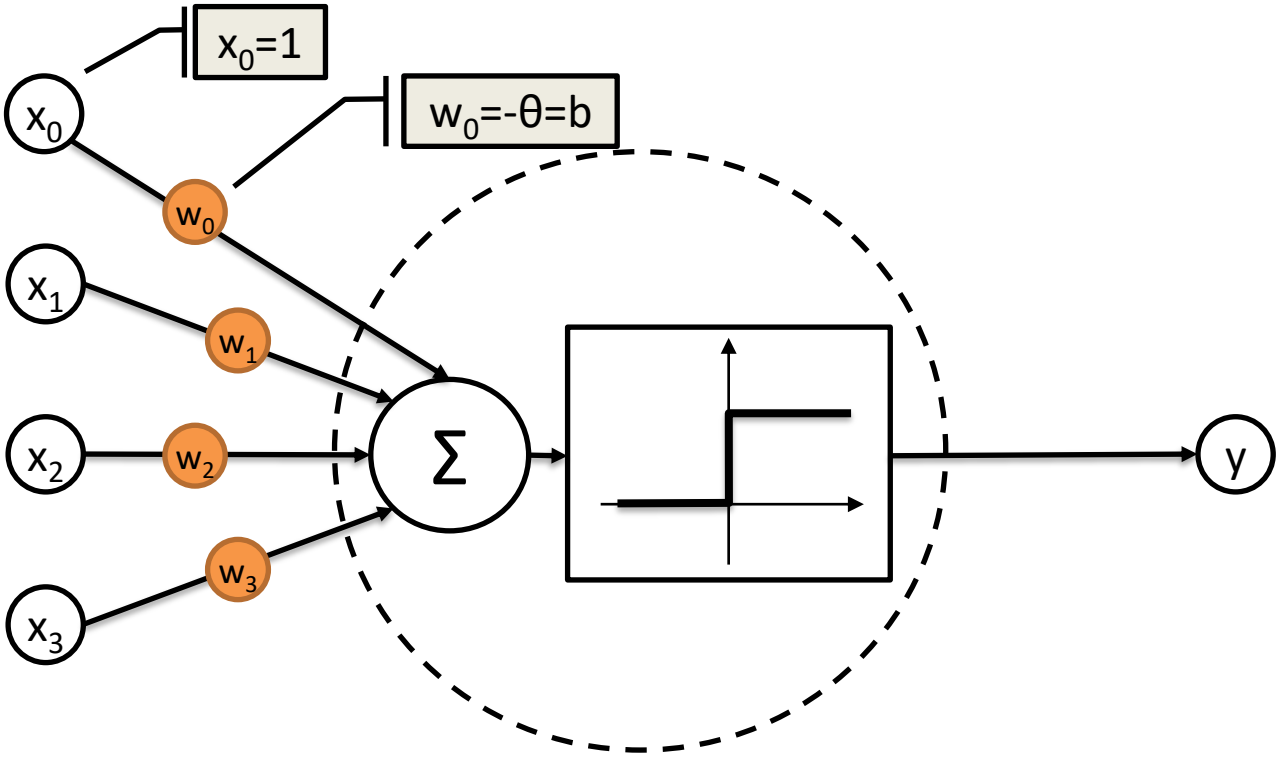


$$z = \sum_i w_i x_i \qquad z = b + \sum_i w_i x_i$$

$$\theta = -b$$

$$y = \begin{cases} 1 & \text{if } z \geq \theta \\ 0 & \text{otherwise} \end{cases} \qquad y = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

Binary Threshold Neurons



$$w_0 = -\theta = b$$

$$x_0 = 1$$

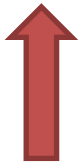
$$z = \sum_i w_i x_i$$

$$y = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

Standard paradigm of classification

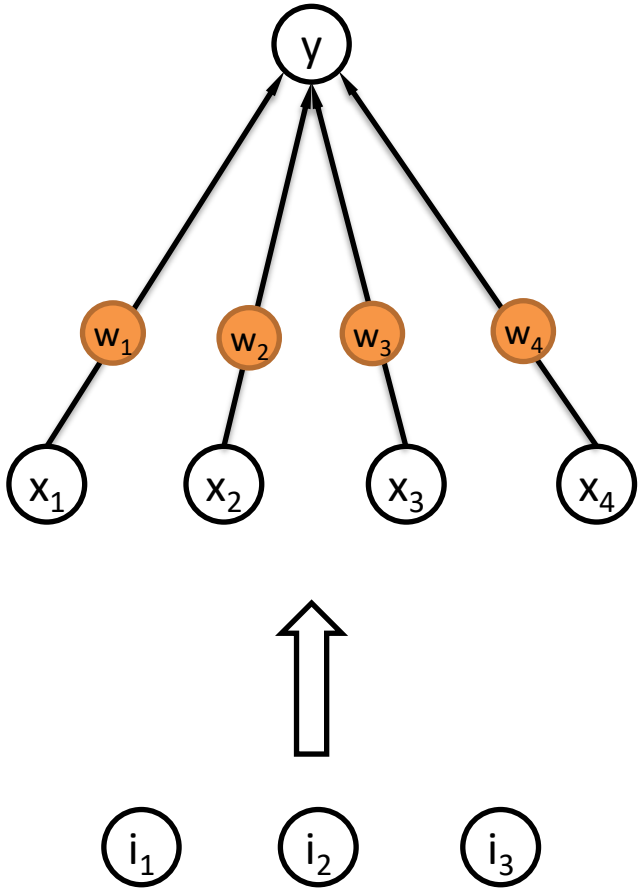
If this quantity is above some threshold, decide that the input vector is a positive example of the target class

Learn how to weight each of the features to get a single scalar quantity



Convert the raw input vector into a vector of feature activations using hand-coded weights or programs

The standard perceptron architecture



Perceptron Convergence Procedure

Add an extra component (x_0) with value 1 to each input vector. The weight of this component is the "bias" weight, equal to minus the threshold

Pick training cases using any policy that ensures that every training sample is seen once before any sample is seen again - i.e. a we can repeat cases, but **uniformly**.

For each training sample run the network

- If the output unit is correct, do not change weights
- If the output unit **incorrectly outputs zero** (should be positive, but returns negative), **add the input vector to the weight vector**
- If the output unit **incorrectly outputs one** (should be negative, but returns positive), **subtract the input vector from the weight vector**

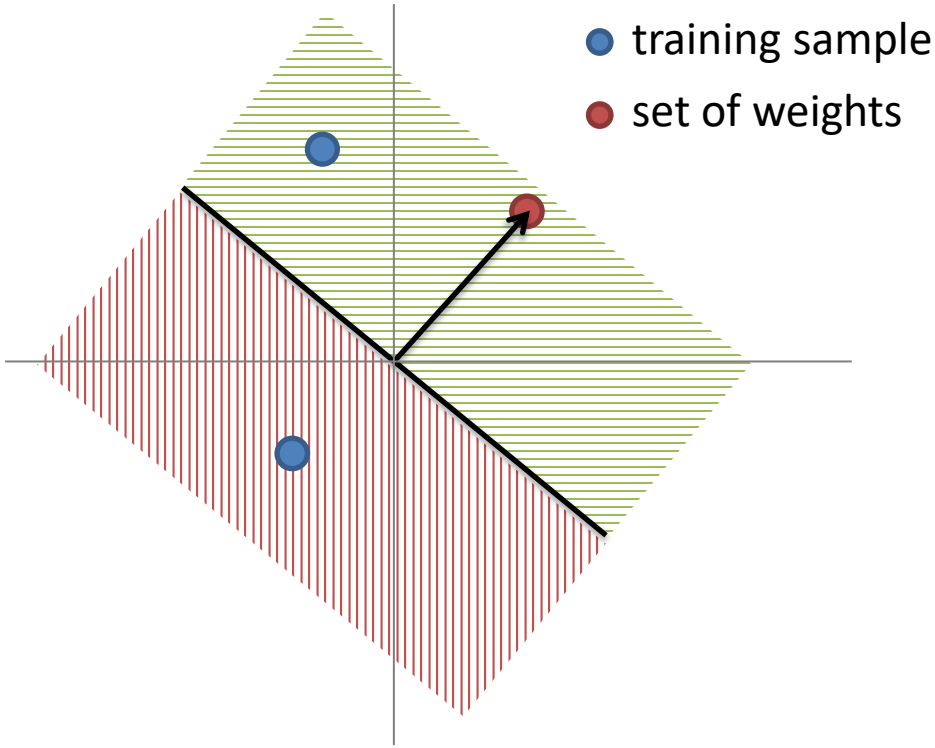
This is guaranteed to find a set of weights that gets the right answer for all the training cases **if any such set exists**

A Geometrical View

We will be switching our view between the data and the parameter space

In the normal (feature) space:

- Each training sample is a point
- Each set of weights defines a plane
- Training samples that this set of weights classifies as positive lie on one side, while training samples that this set of weights classify as negative lie on the other side



Assume that we have eliminated the threshold (all planes go through the origin)

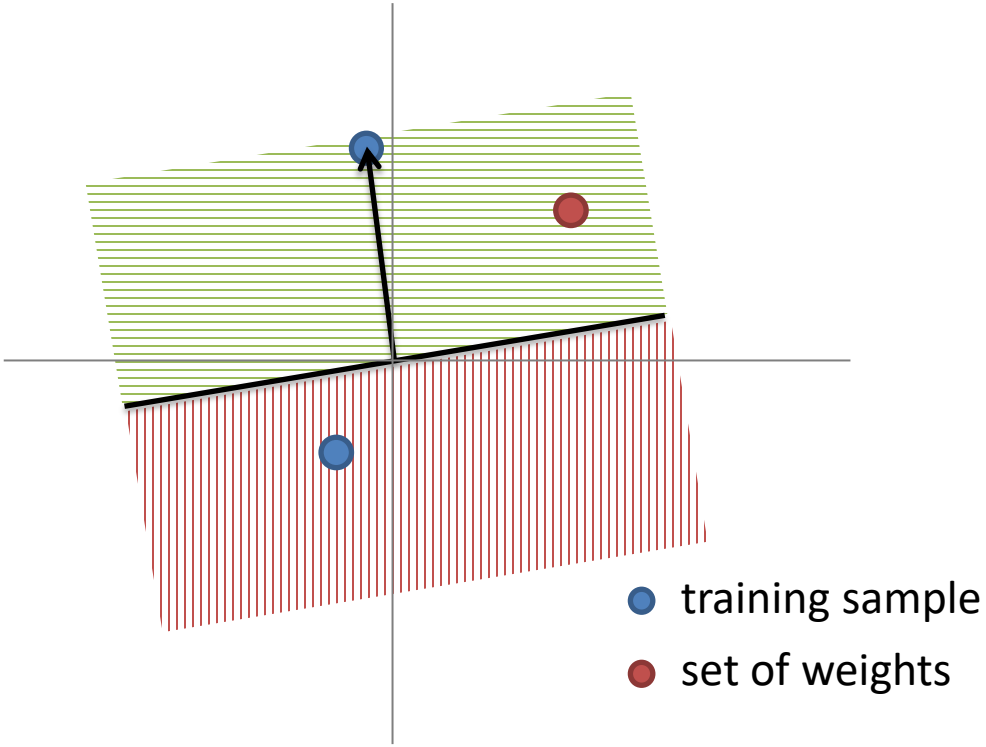
$$z = \sum_i w_i x_i \qquad y = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

A Geometrical View

We will be switching our view between the data and the parameter space

In the parameter (weight) space:

- Each set of weights is a point
- Each training sample defines a plane
- Weights that classify it correctly lie on one side, while weights that do not lie on the other side



Assume that we have eliminated the threshold (all planes go through the origin)

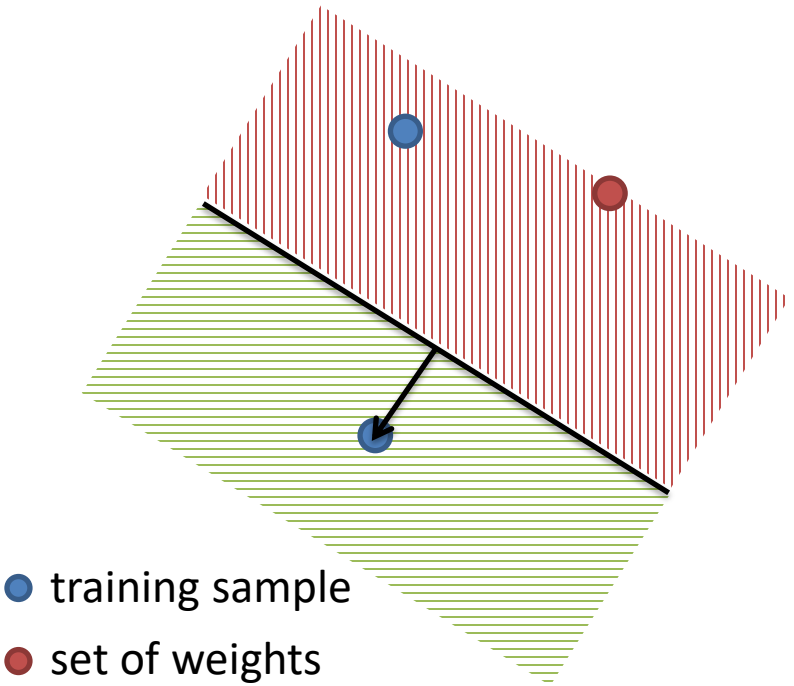
$$z = \sum_i w_i x_i \qquad y = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

A Geometrical View

We will be switching our view between the data and the parameter space

In the parameter (weight) space:

- Each set of weights is a point
- Each training sample defines a plane
- Weights that classify it correctly lie on one side, while weights that do not lie on the other side



Assume that we have eliminated the threshold (all planes go through the origin)

$$z = \sum_i w_i x_i \qquad y = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

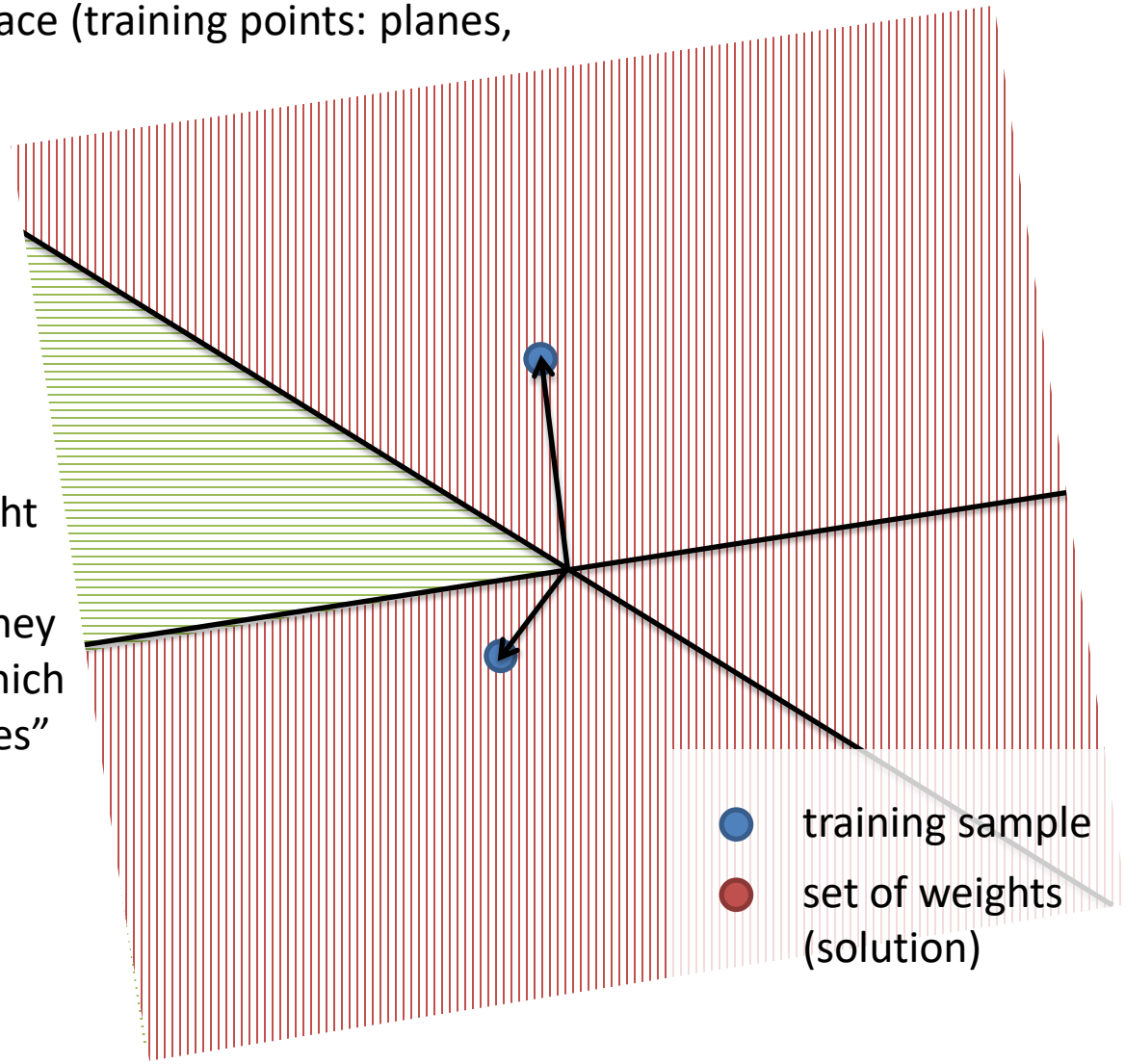
The cone of feasible solutions

We are in the parameter space (training points: planes, weights: points)

To classify all training samples correctly we need to find a set of weights on the correct side of all the planes

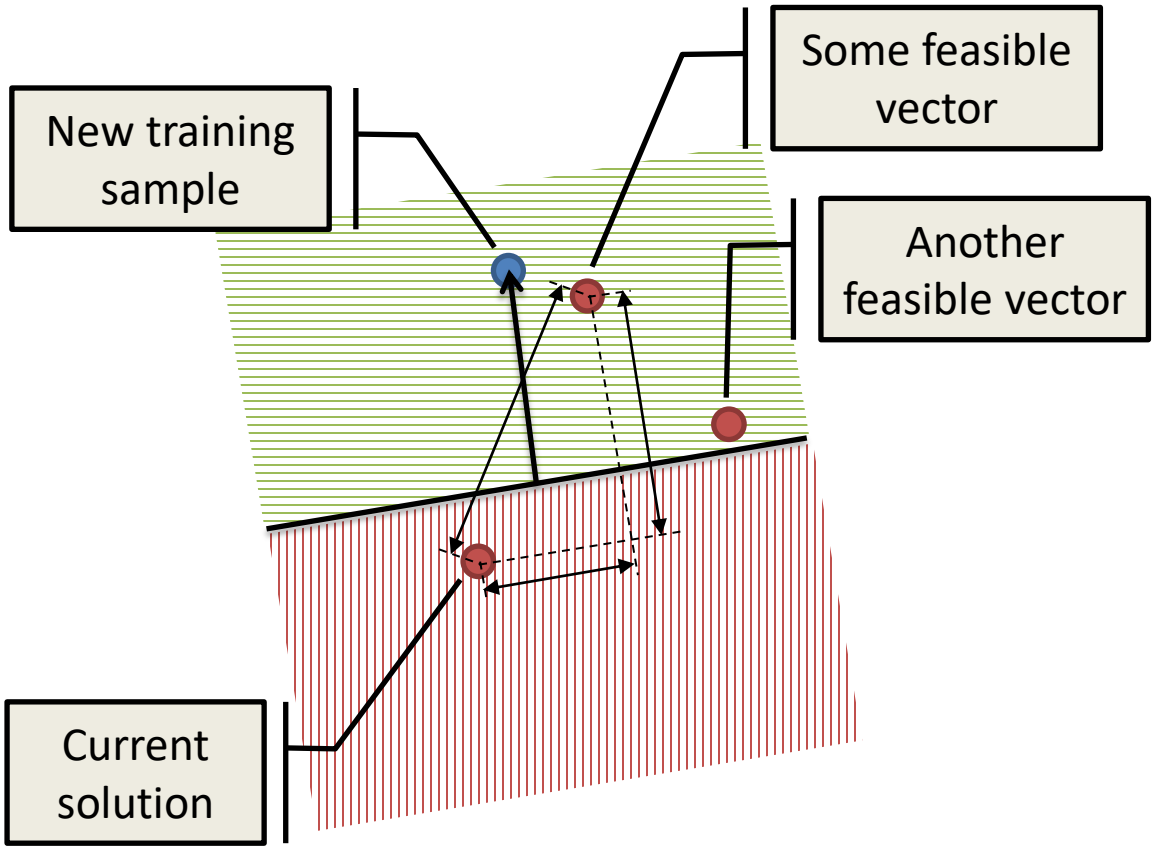
If there are any sets of weight that can give us the correct responses for all samples, they will lie in the hyper-cone which is the union of all “good sides”

The average of two good solutions is a good solution (the problem is convex)



Why does the learning procedure work?

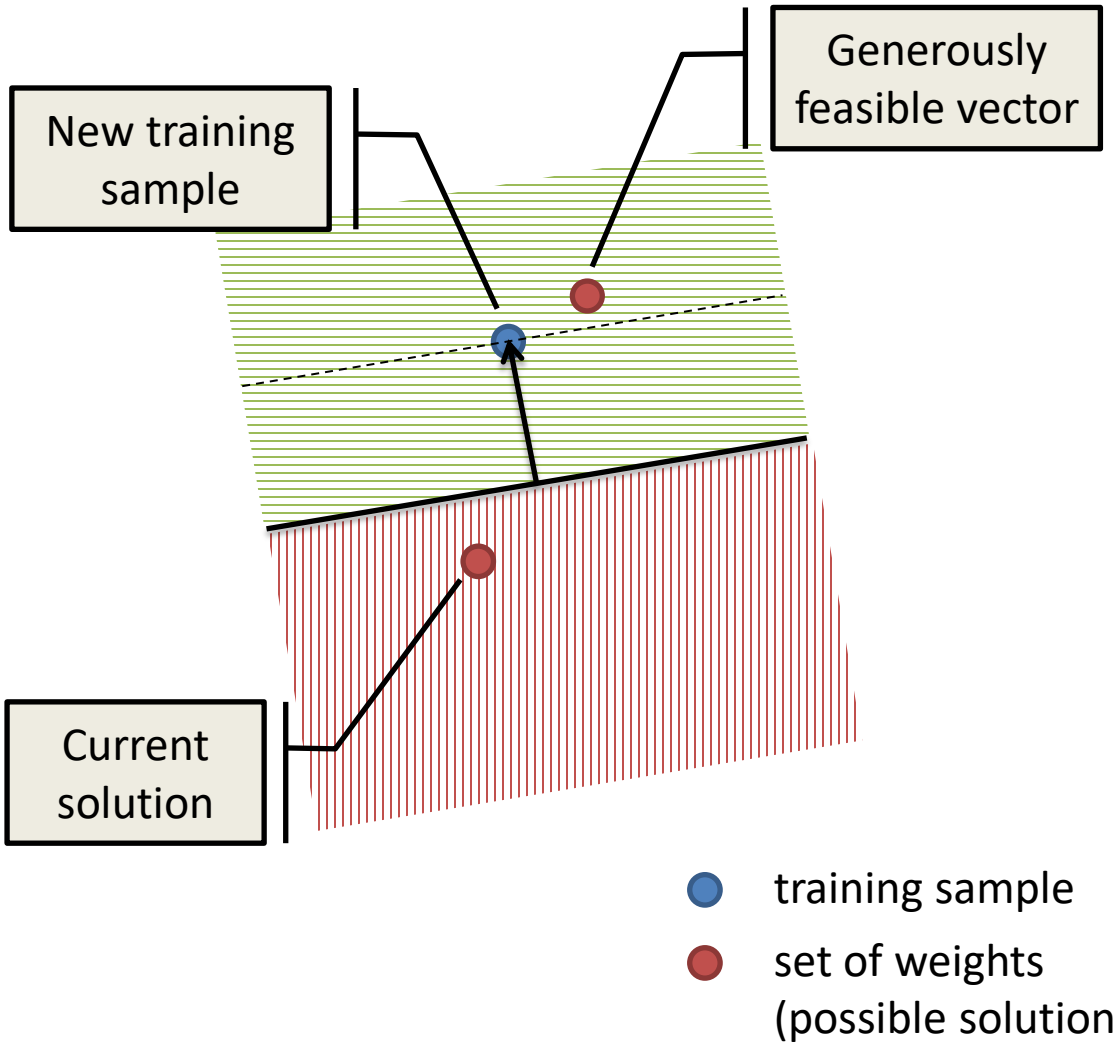
Claim: Every time the perceptron makes a mistake, the learning algorithm moves the current weight vector closer to all feasible weight vectors



Why does the learning procedure work?

Consider "generously feasible" weight vectors that lie within the feasible region by a margin at least as great as the length of the input vector that defines each constraint plane

Real claim: Every time the perceptron makes a mistake the squared distance to all of these generously feasible vectors is always reduced by at least the squared length of the update vector



Limitations of Perceptrons

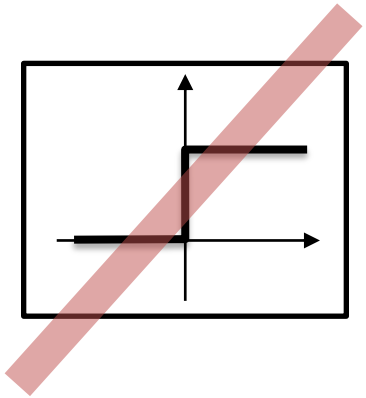
- Basic algorithm only works with classes that are linearly separable
  - It emphasizes that the difficult bit of learning is to learn the right features. If you can design the right features, you can do almost everything
- Many variations of the Perceptron algorithm:
  - Perceptron of optimal stability **Min-Over** algorithm (Krauth and Mezard, 1987), **AdaTron** (Anlauf and Biehl, 1989): aims at finding the largest separating margin in separable problems.
  - **Pocket algorithm** with ratchet (Gallant, 1990) keeps the best solution (number of errors, stable over a number of iterations) seen so far "in its pocket". The pocket algorithm then returns the solution in the pocket, rather than the last solution.
  - **Maxover algorithm** (Wendemuth, 1995) Update is based on a function acting on data point and weight that has to satisfy certain properties.
  - **Voted Perceptron** (Freund and Schapire, 1999), Ensemble of linear classifiers, each set of weights gets a vote relative to the number of iterations it survived before it had to be updated.

WHERE TO NOW?

Where to now?

A more flexible learning algorithm

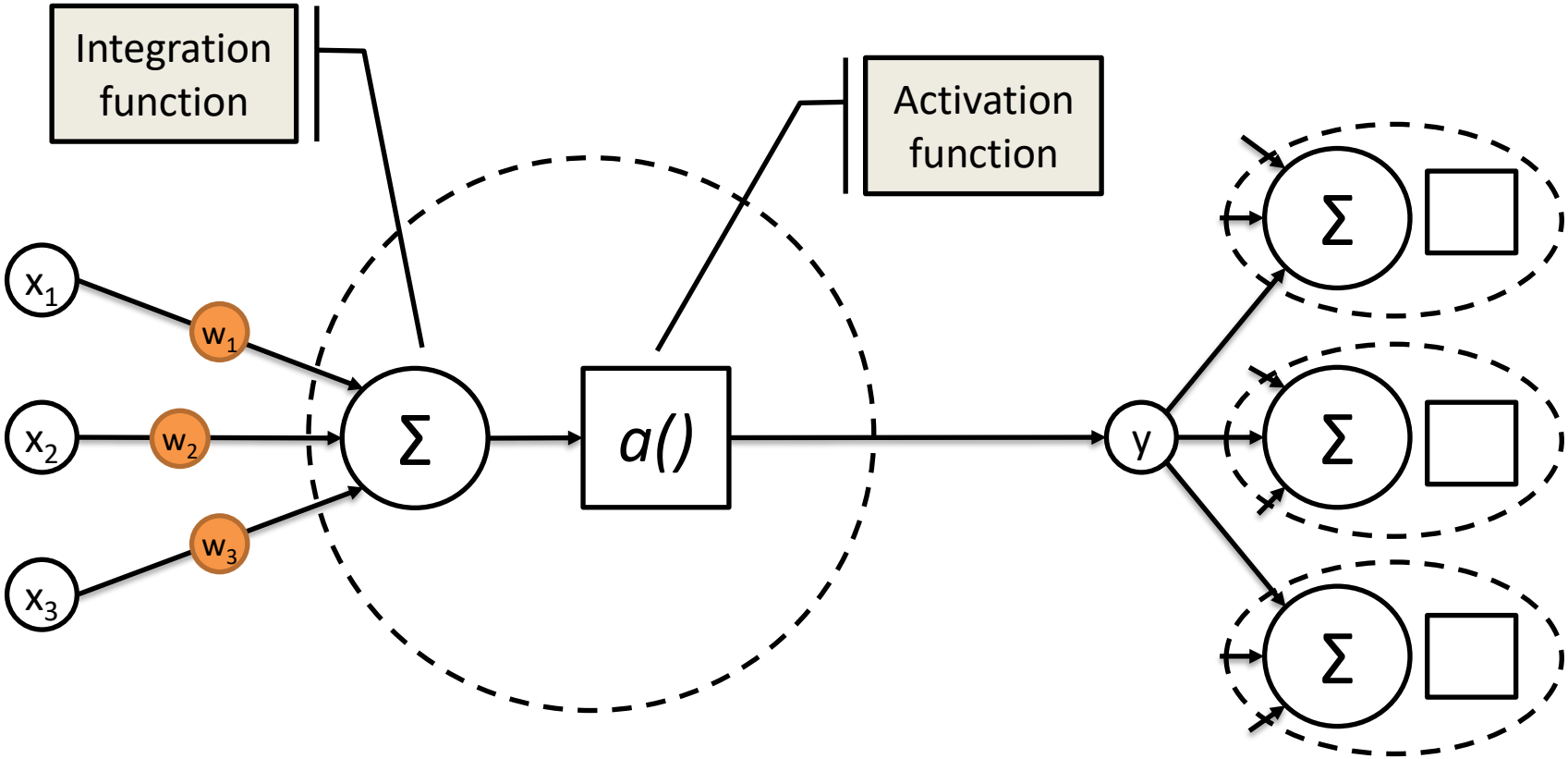
Gradient Descent



Do more than just linear

Hidden Layers

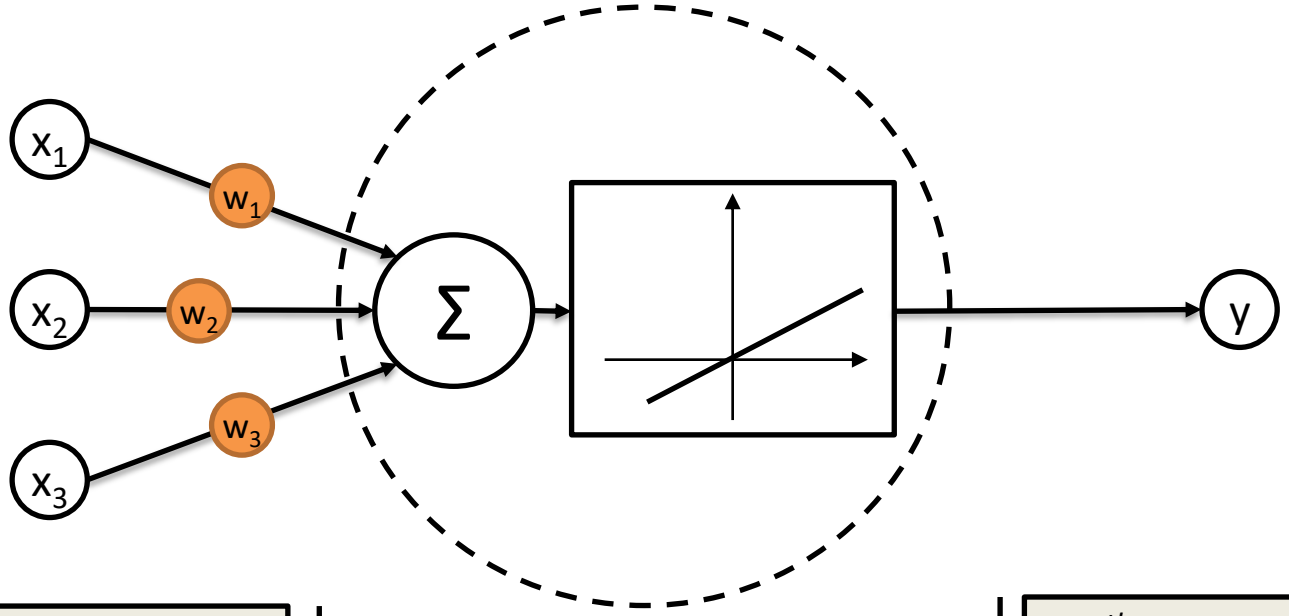
A computational model of an idealised neuron



*(There is also a **bias** term that is summed, which is not always explicitly shown)*

Linear Neurons

aka: Linear regression



Integration value

$$z = b + \sum_i w_i x_i$$

i^{th} input

bias

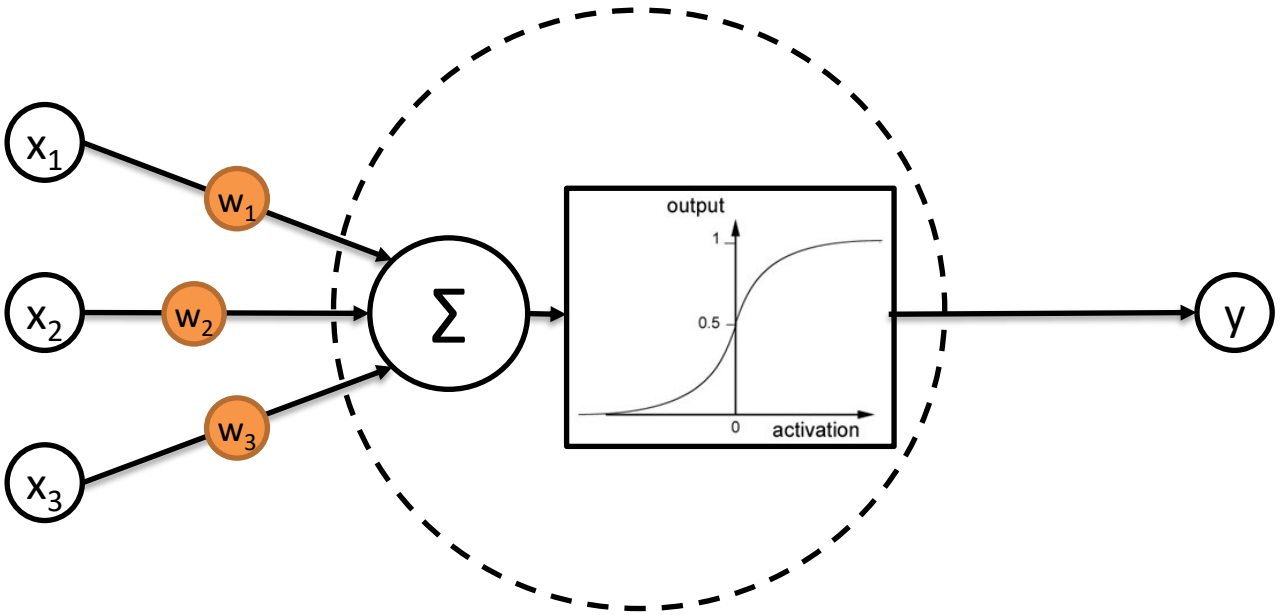
weight on i^{th} input

$$y = \alpha z$$

output

Logistic (Sigmoid) Neurons

aka: Logistic regression



Real-valued output – smooth and bounded between [0, 1]

$$z = b + \sum_i w_i x_i$$

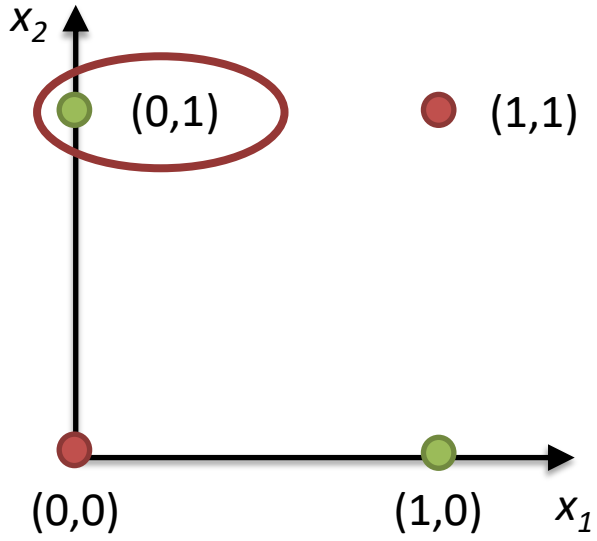
$$y = \frac{1}{1 + e^{-z}}$$

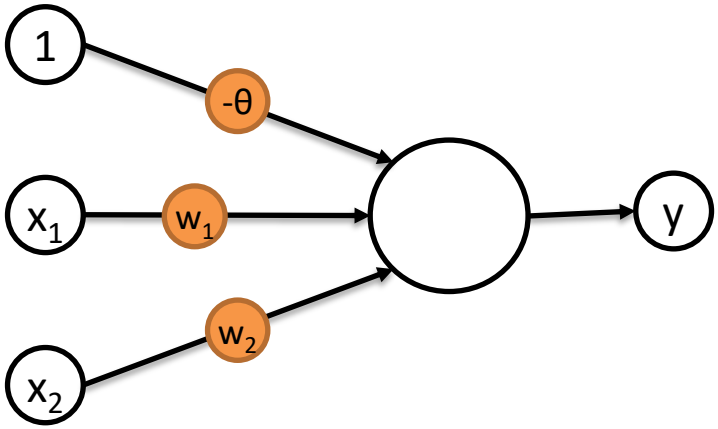
Solving the XOR problem

Imagine the following training set:

- Positive cases if x_1 is the same as x_2
- Negative cases if x_1 is different to x_2

$$w_2 < \theta$$





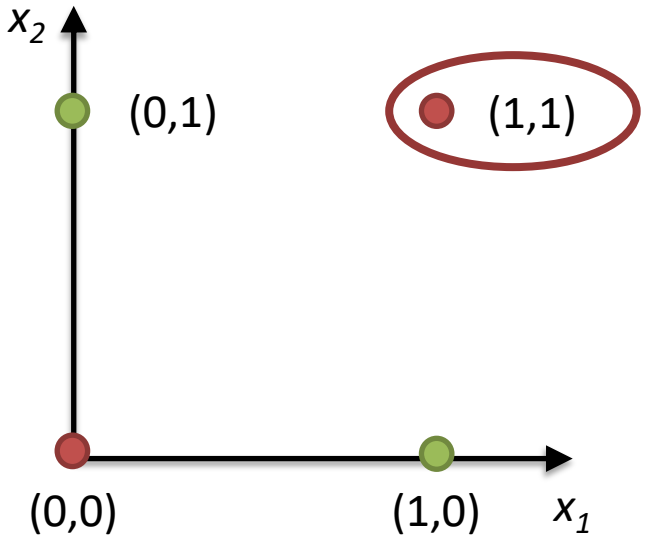
$$y = \begin{cases} 1 & \sum_i w_i x_i \geq \theta \\ 0 & \sum_i w_i x_i < \theta \end{cases}$$

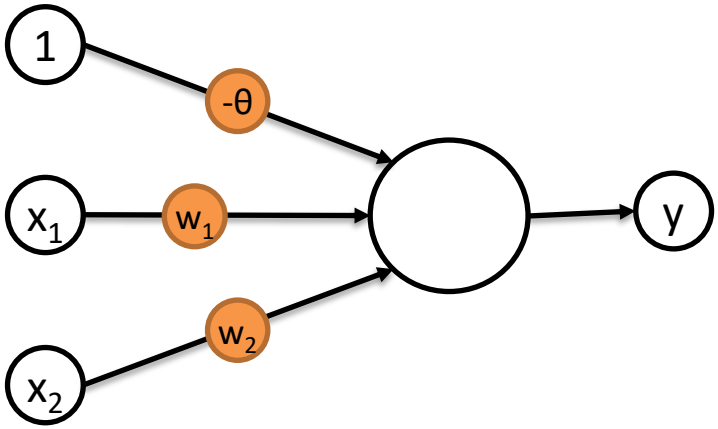
Solving the XOR problem

Imagine the following training set:

- Positive cases if x_1 is the same as x_2
- Negative cases if x_1 is different to x_2

$w_2 < \theta$ $\qquad$ $w_1 + w_2 \geq \theta$





$$y = \begin{cases} 1 & \sum_i w_i x_i \geq \theta \\ 0 & \sum_i w_i x_i < \theta \end{cases}$$

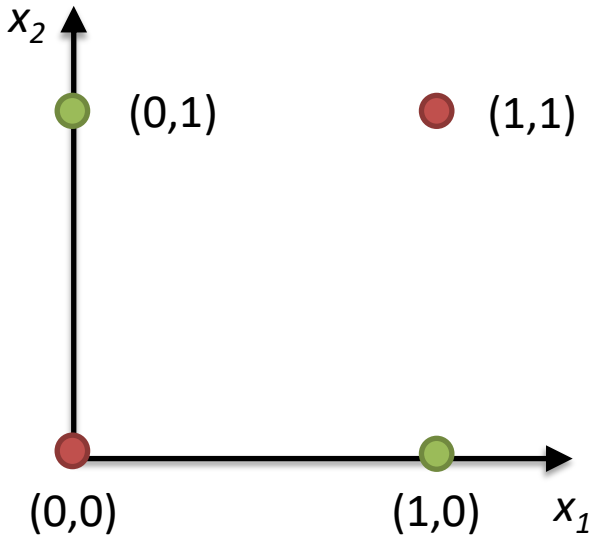
Solving the XOR problem

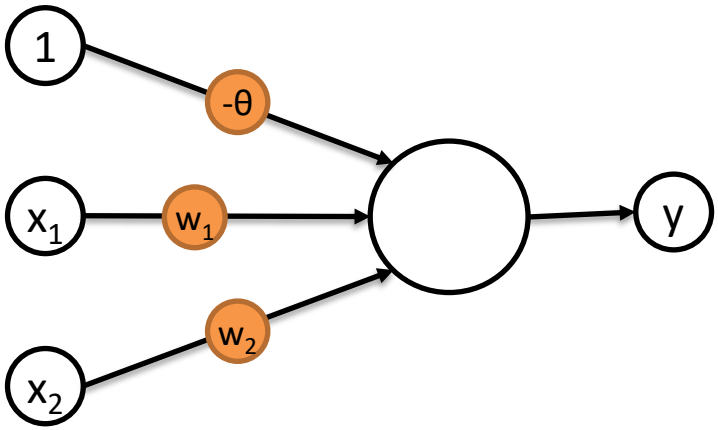
Imagine the following training set:

- Positive cases if x_1 is the same as x_2
- Negative cases if x_1 is different to x_2

$w_2 < \theta$ $\qquad$ $w_1 + w_2 \geq \theta$

$0 \geq \theta$ $\qquad$ $w_1 < \theta$



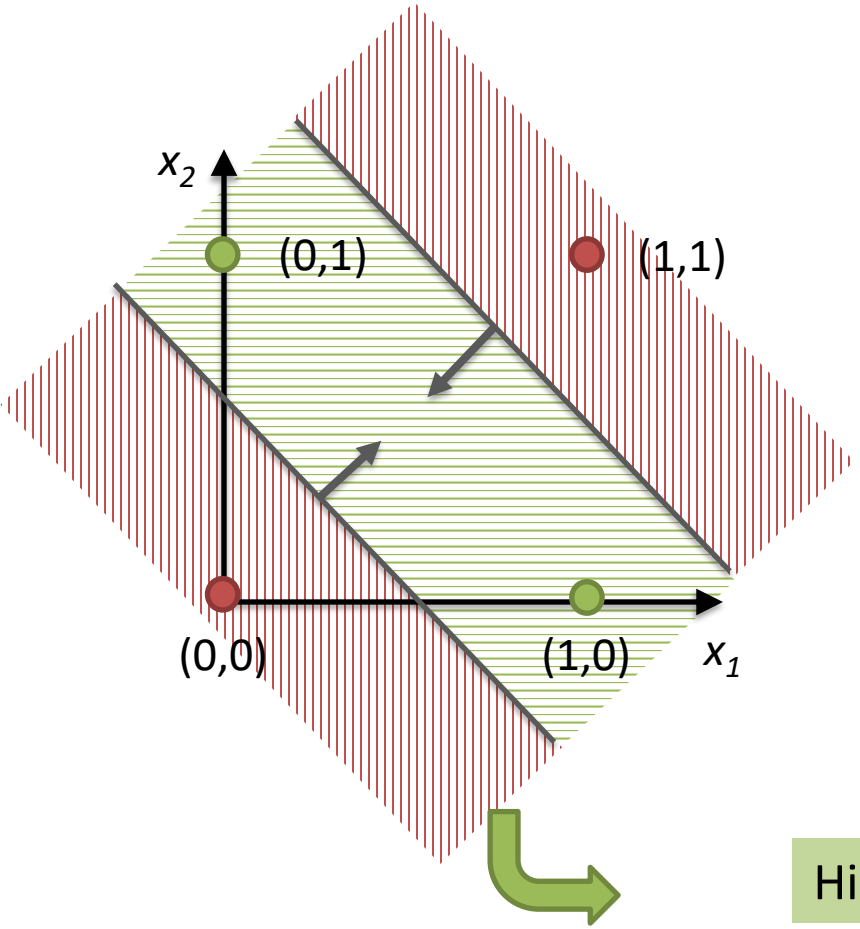


$$y = \begin{cases} 1 & \sum_i w_i x_i \geq \theta \\ 0 & \sum_i w_i x_i < \theta \end{cases}$$

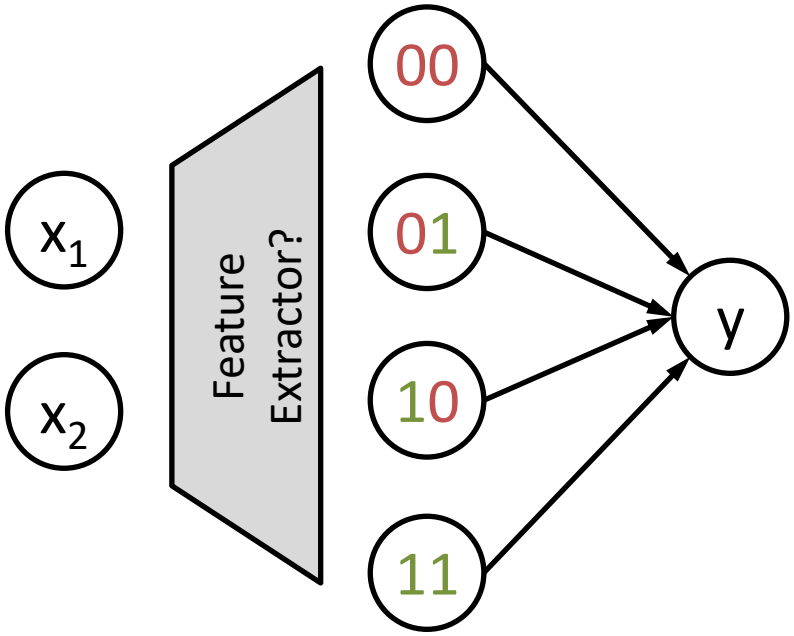
Not linearly separable. Remember, all we do is still: $y = \text{sgn}(\mathbf{w}^T\mathbf{x} + \mathbf{b})$

Two ways to solve this problem (two ways of thinking about it)

Use multiple lines



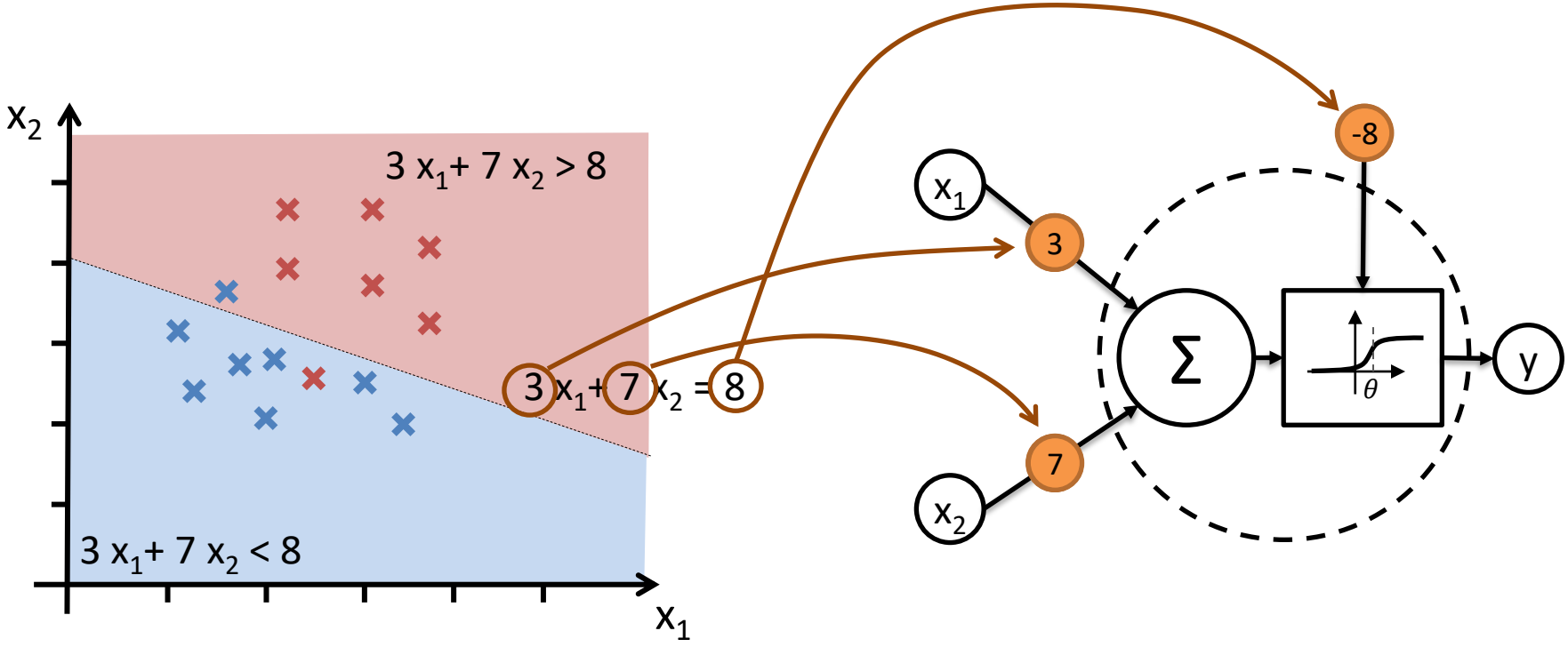
Design the right features



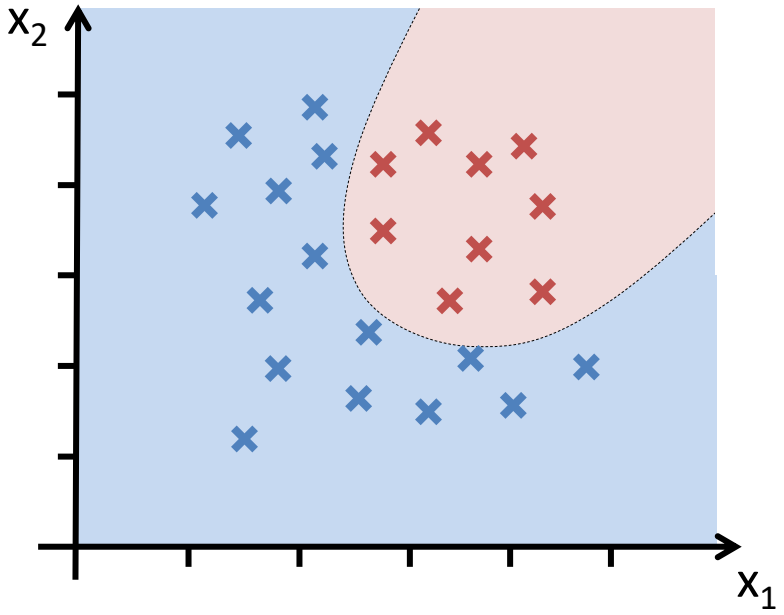
Hidden layers

BUILDING MORE COMPLEX NETWORKS

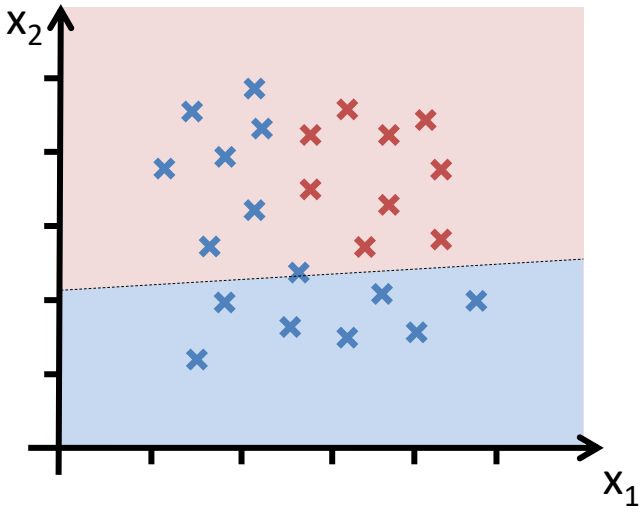
A graphical representation of what a logistic neuron does

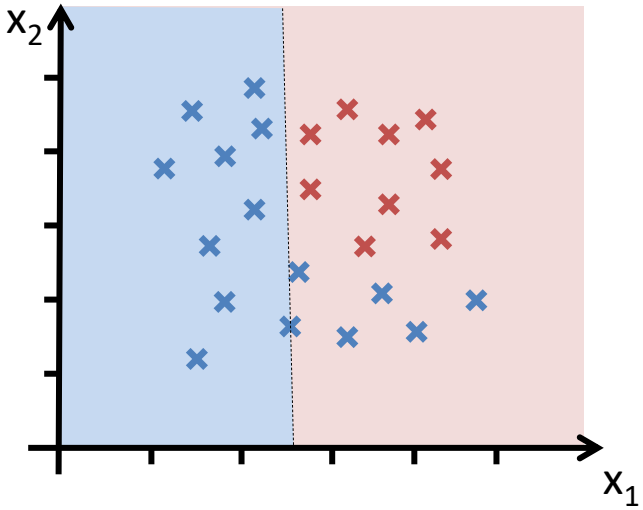


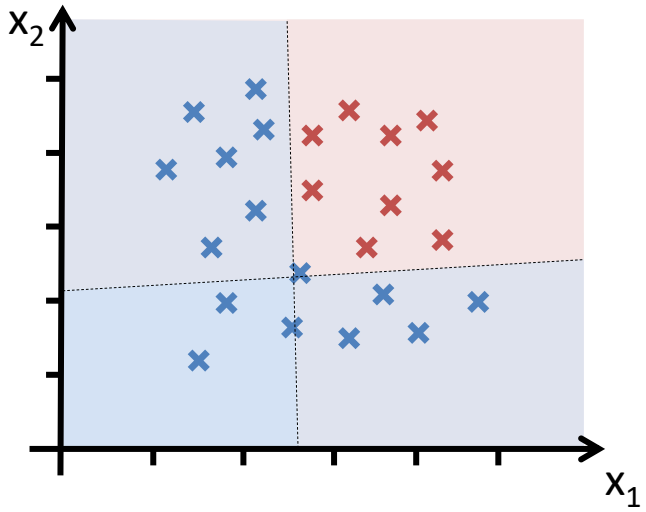
Non linear decision boundaries



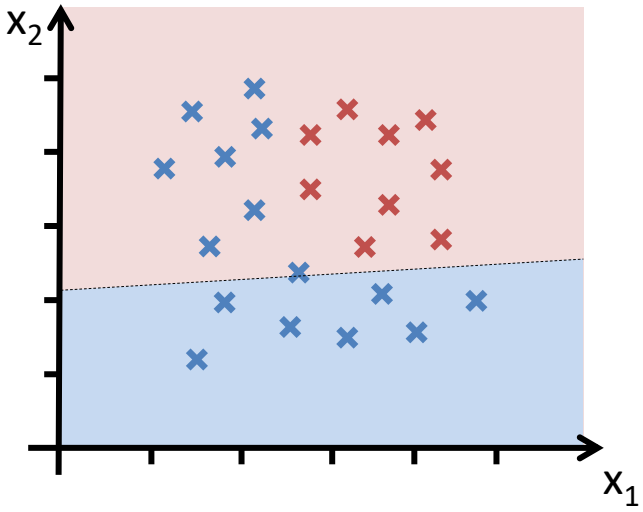
Combining linear decision boundaries

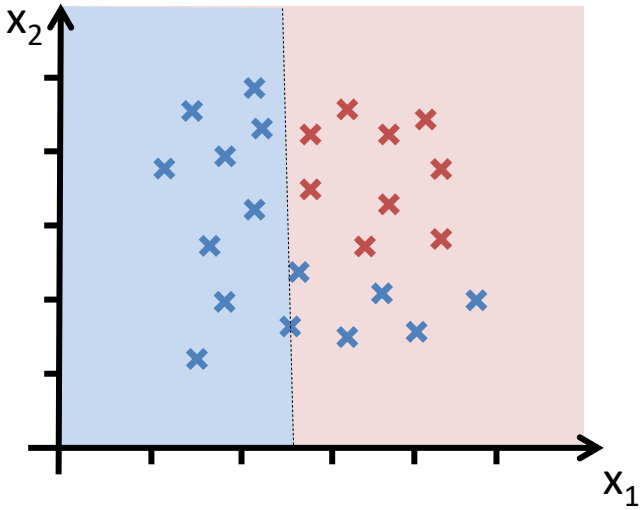


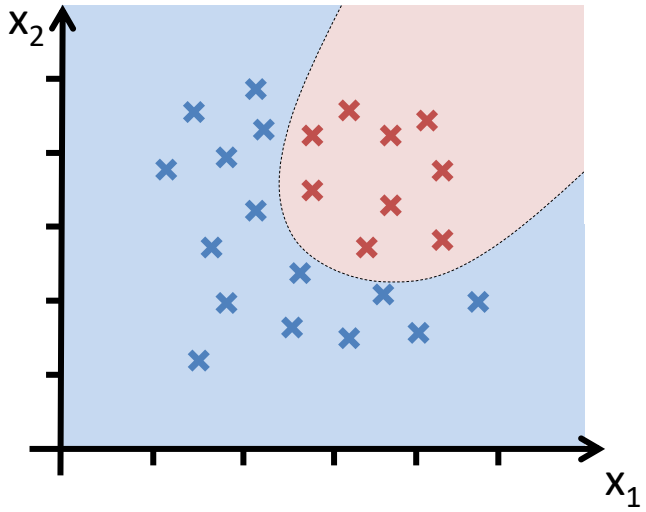




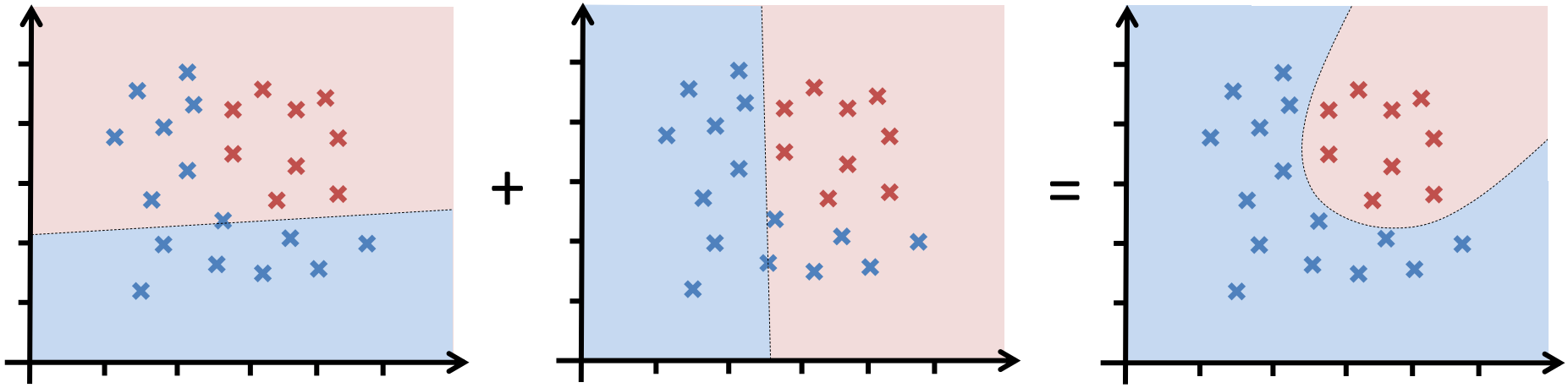
Combining linear decision boundaries



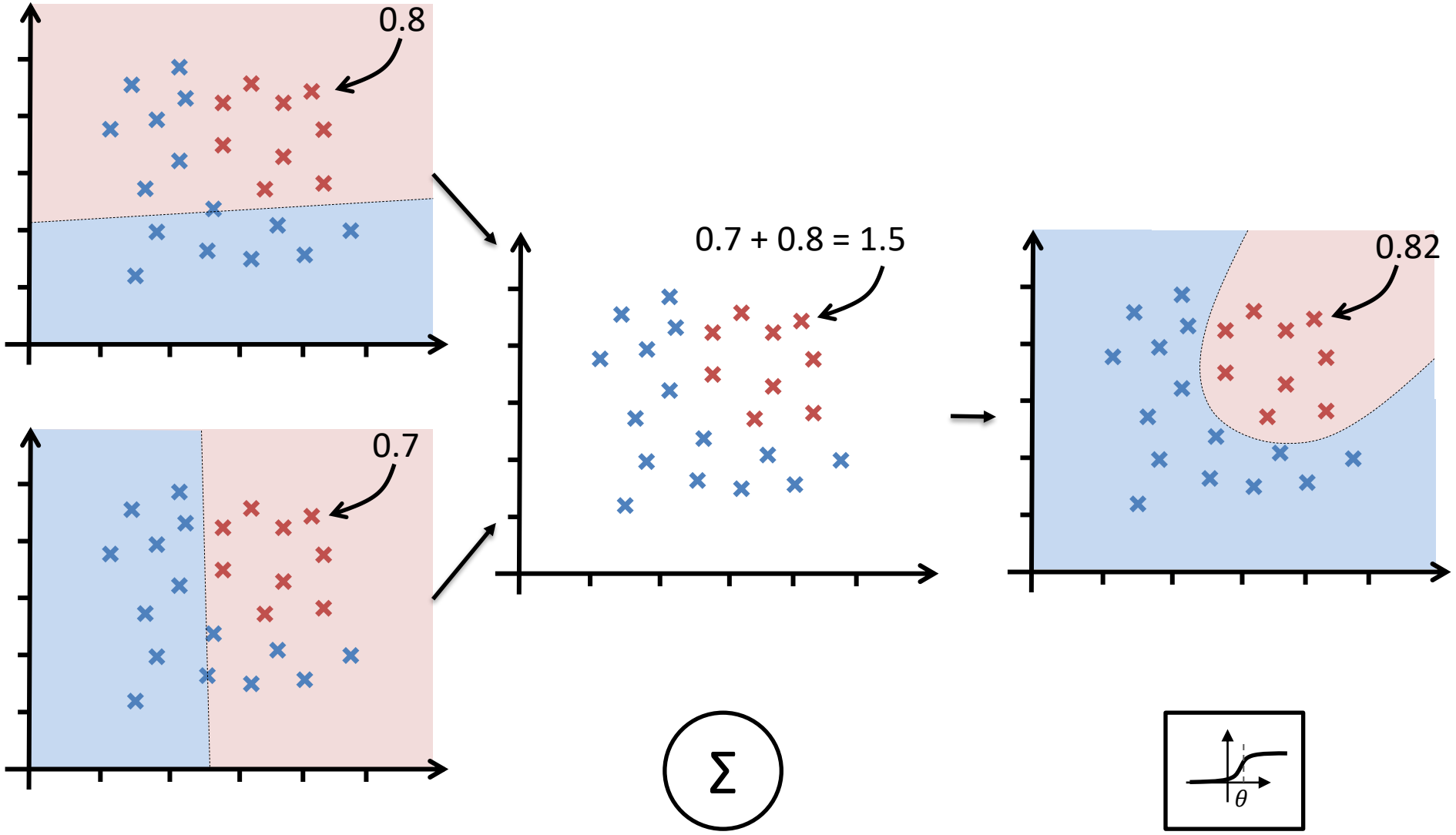




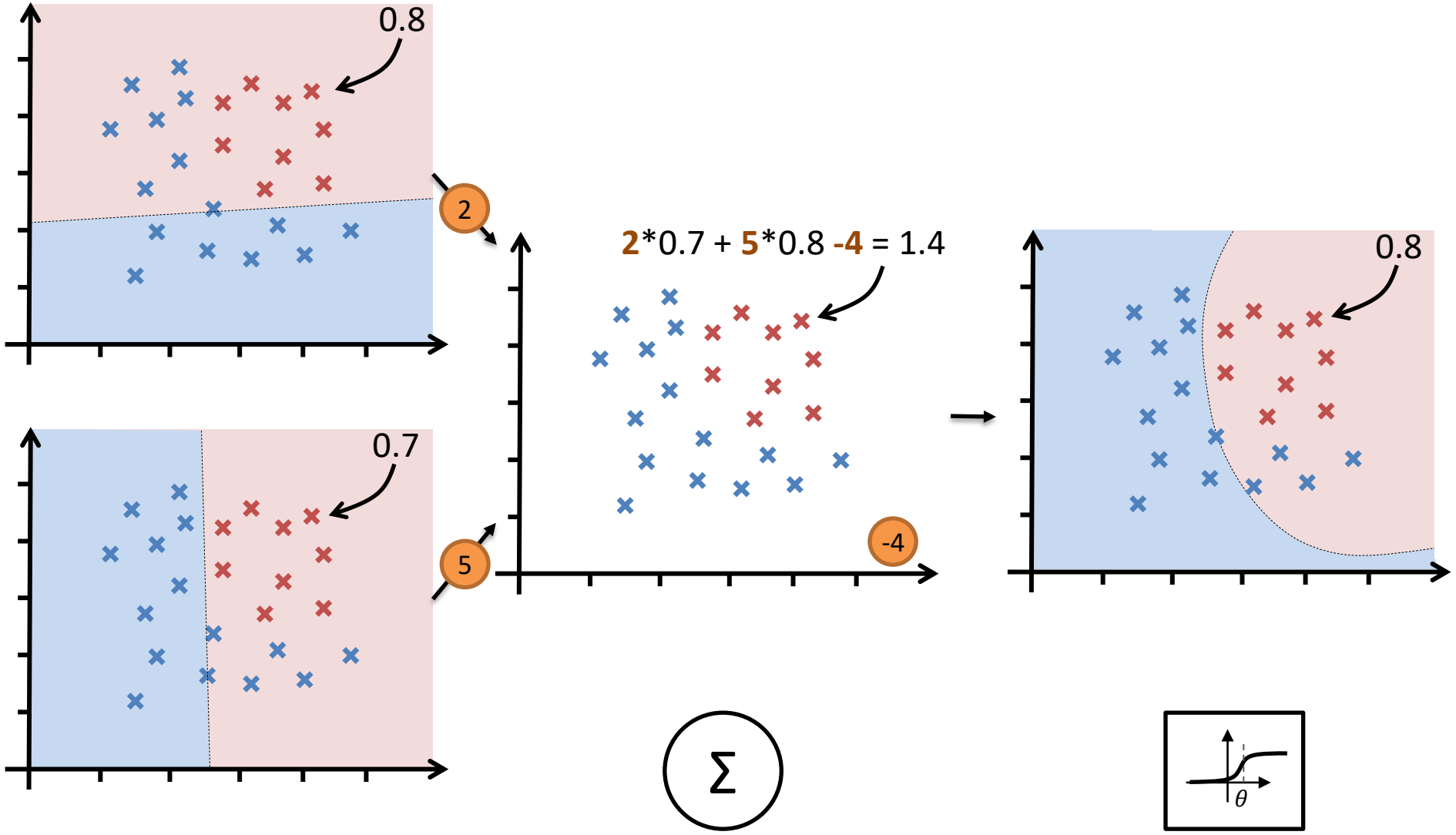
Combining linear decision boundaries



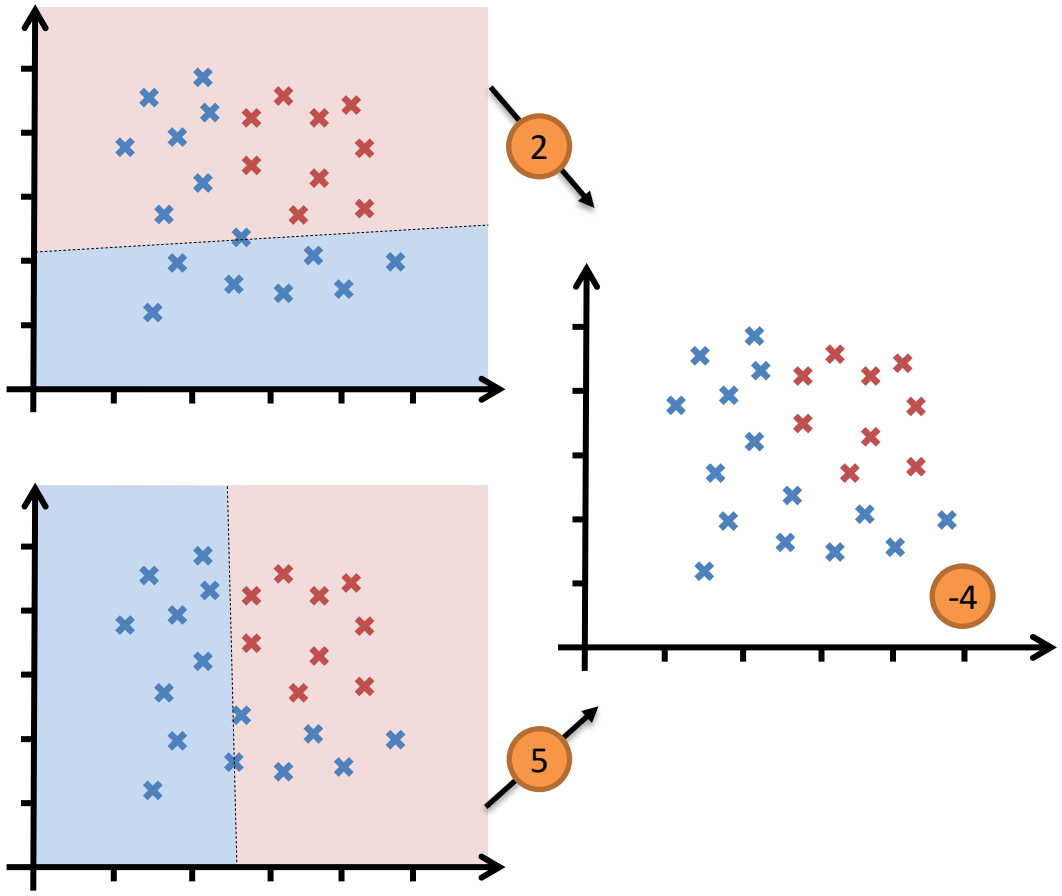
Neural Network

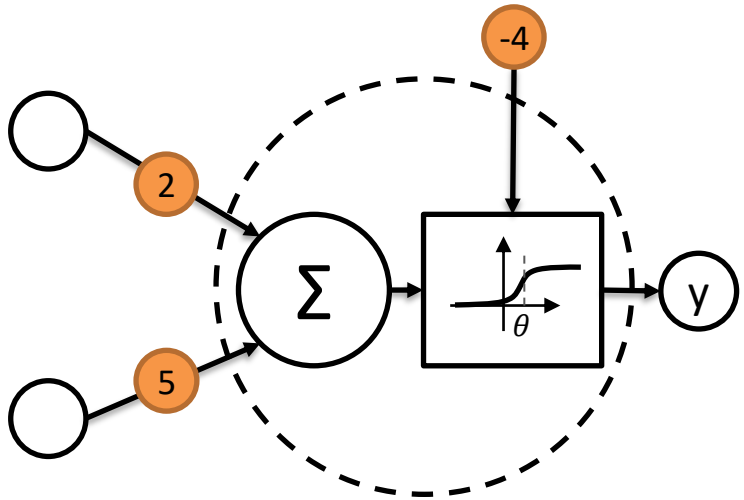


Neural Network

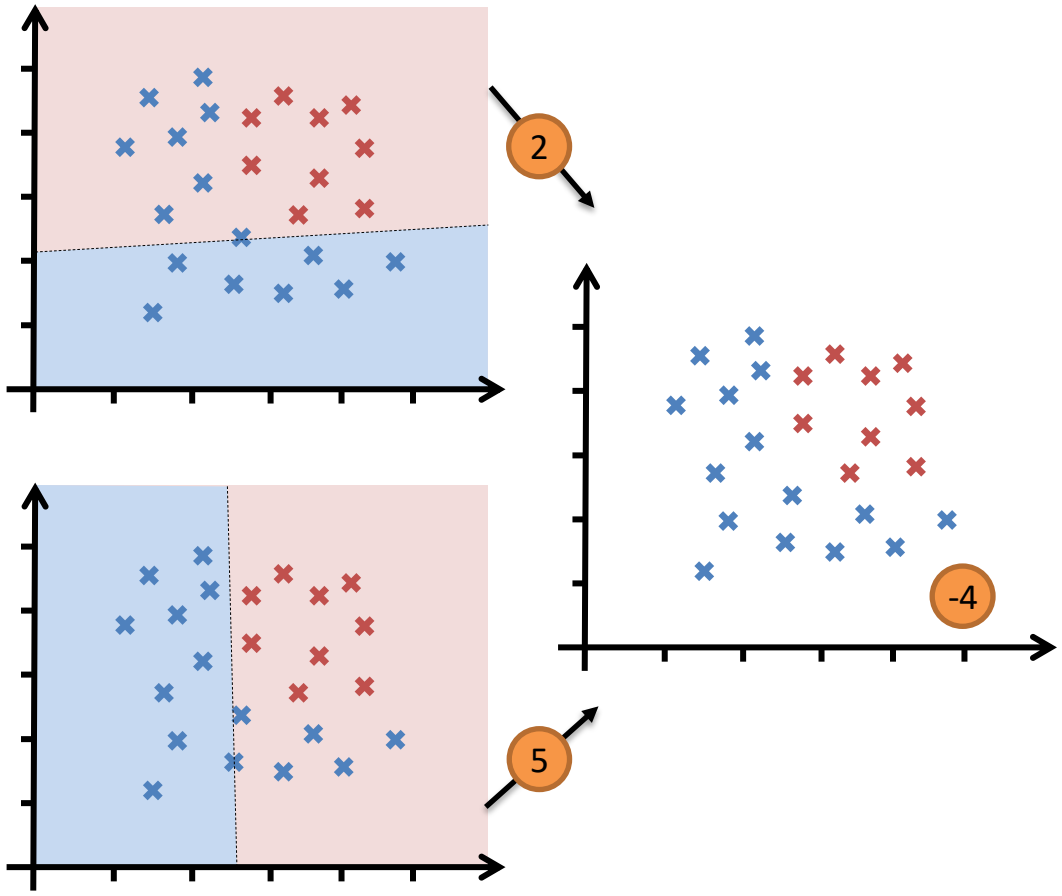


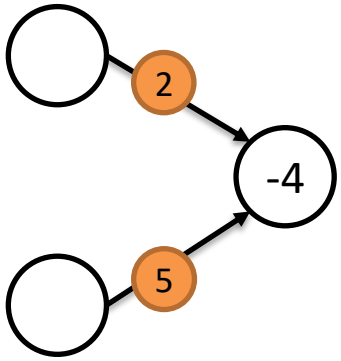
Neural Network



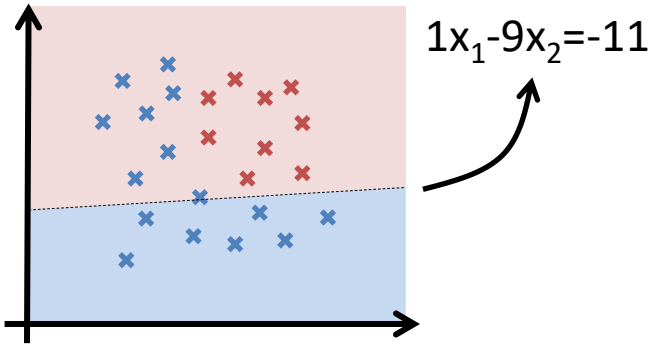


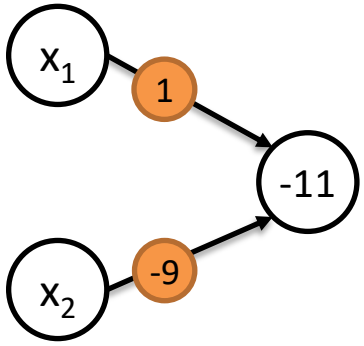
Neural Network

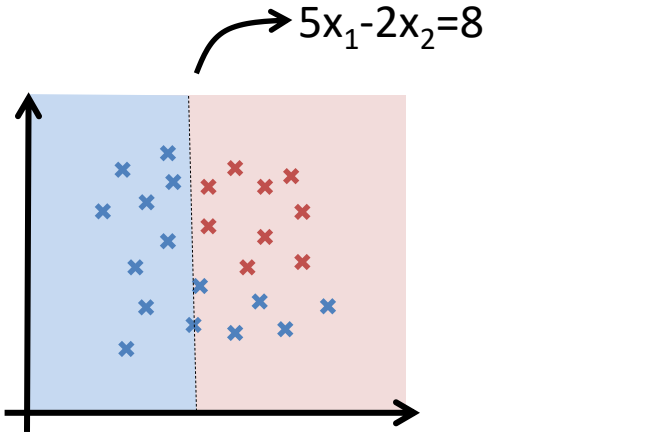


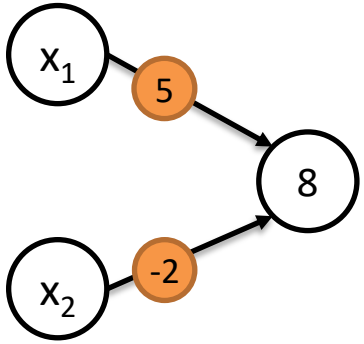


Neural Network

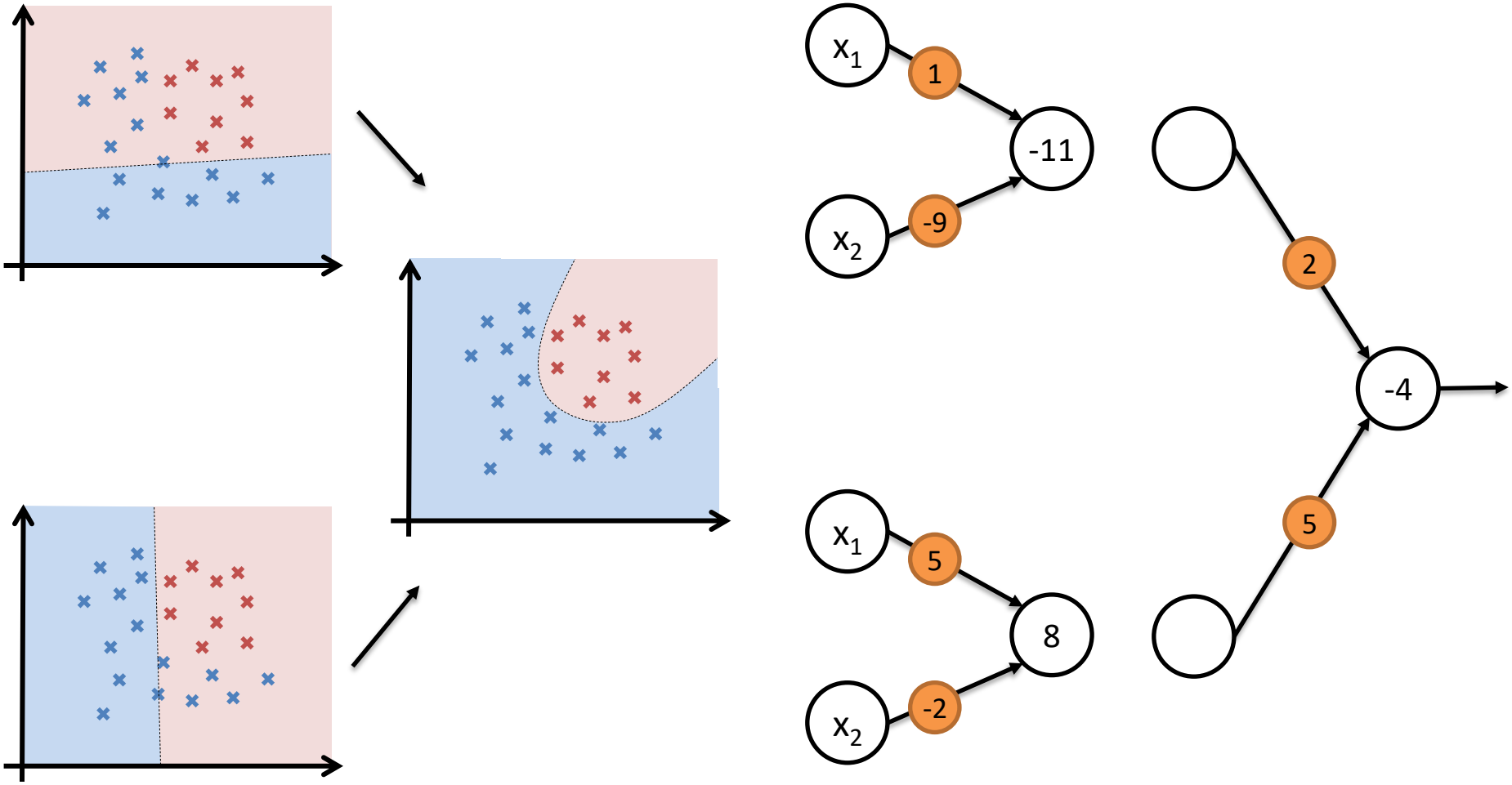




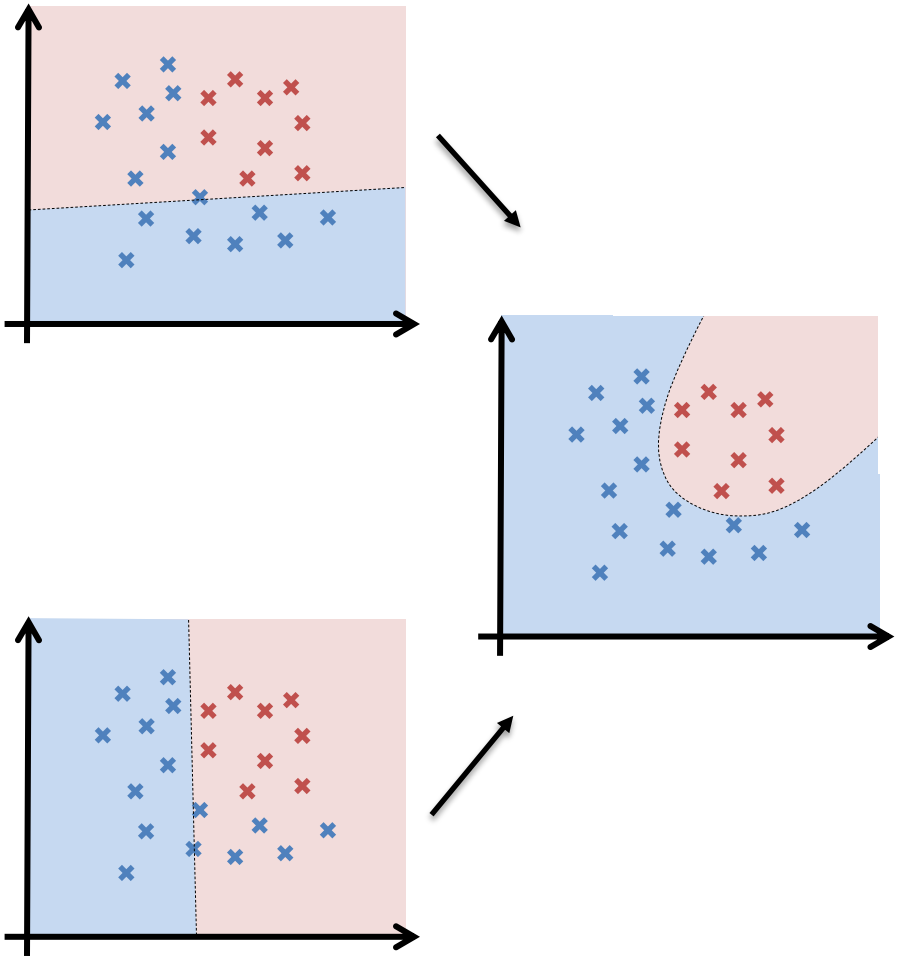


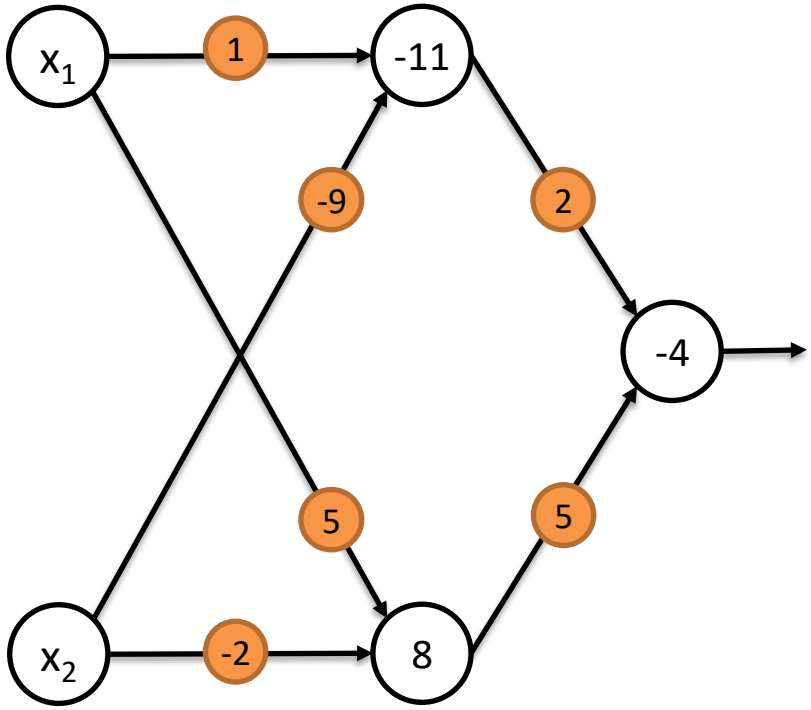


Neural Network

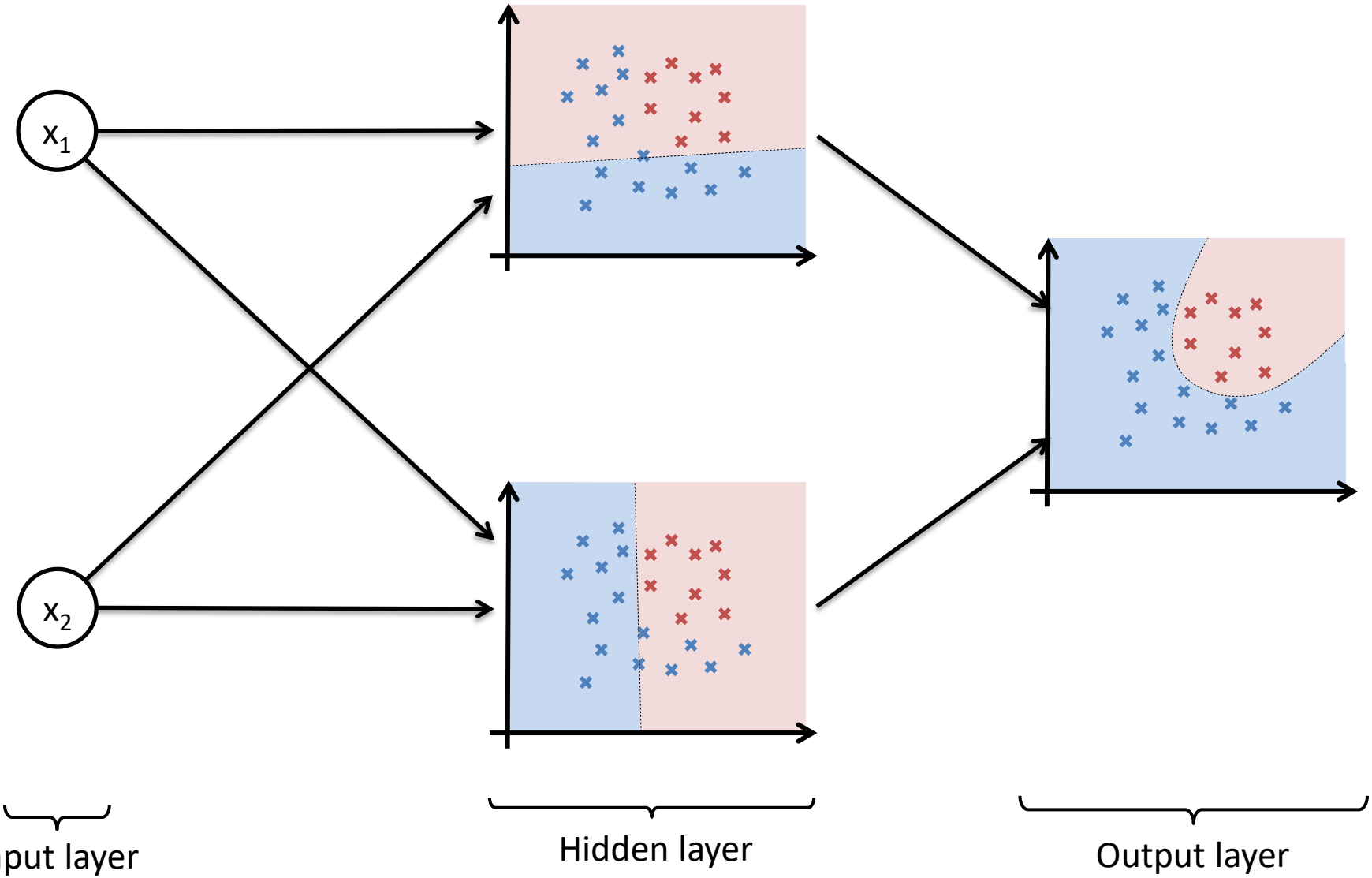


Neural Network

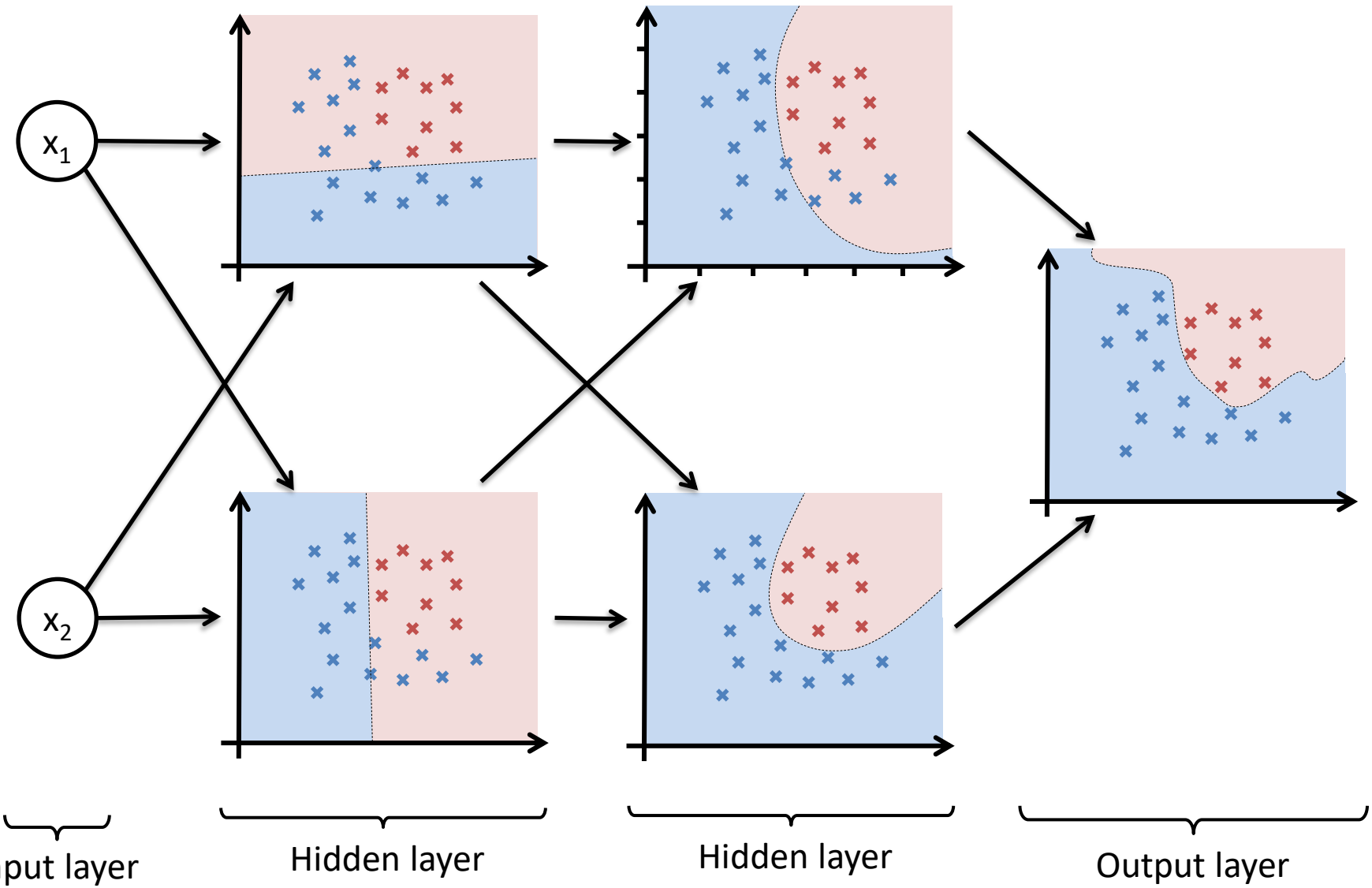




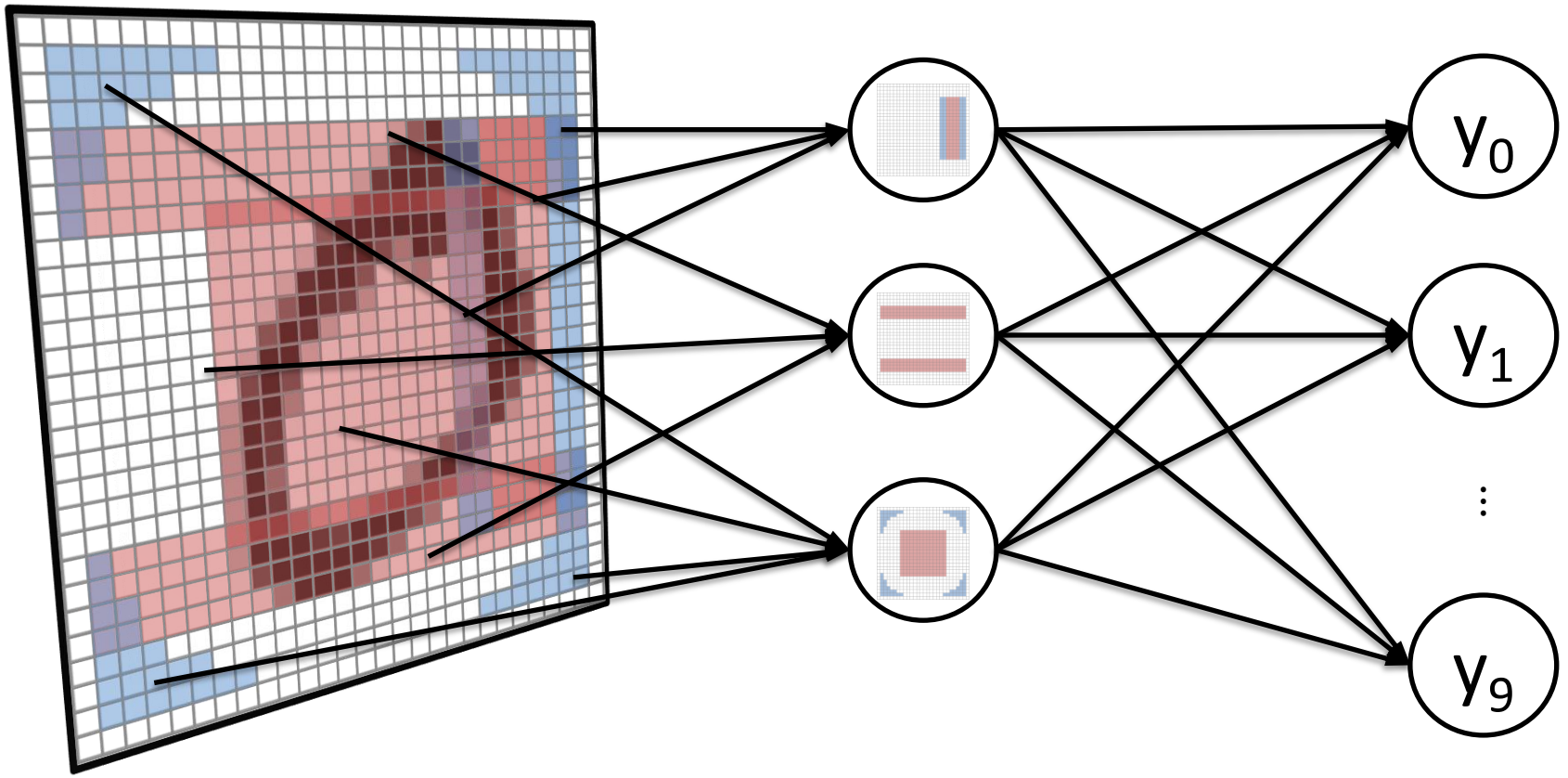
Neural Network



Deep Neural Network



Representation learning



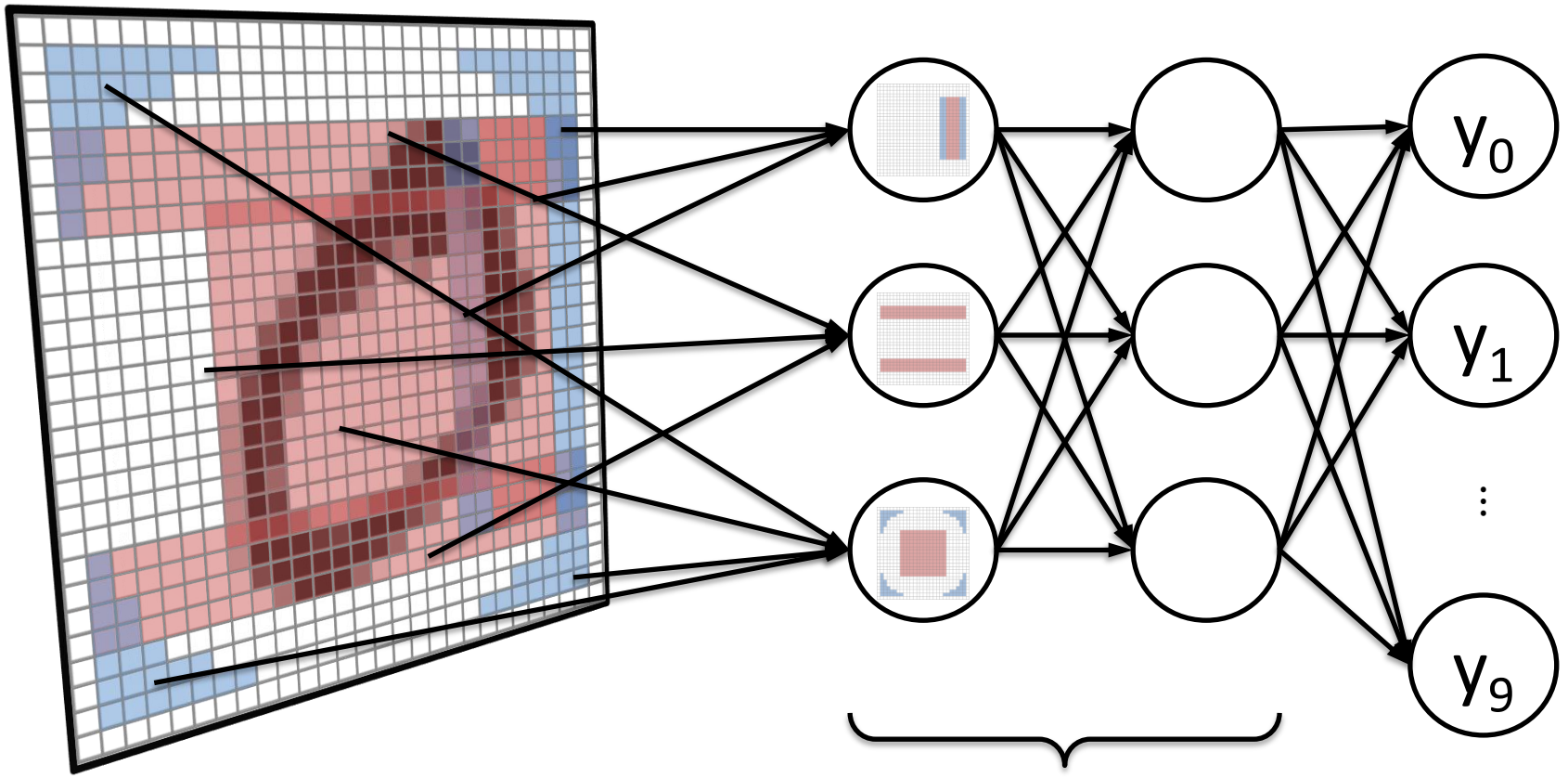
Input: pixel values

Hidden layers:
representation learning

Output: score
for each digit

Representation learning

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

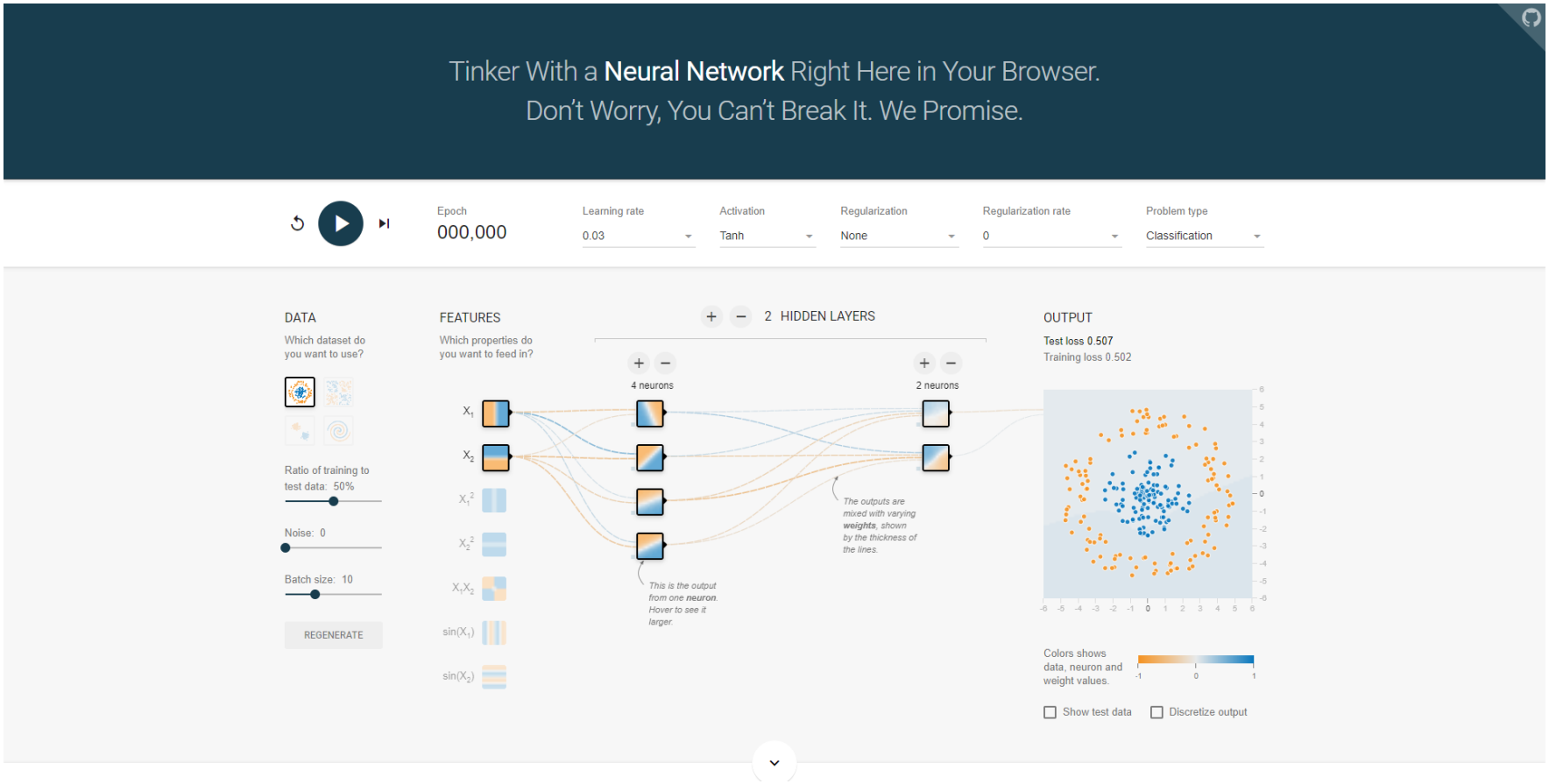


Input: pixel values

Hidden layers: representation learning

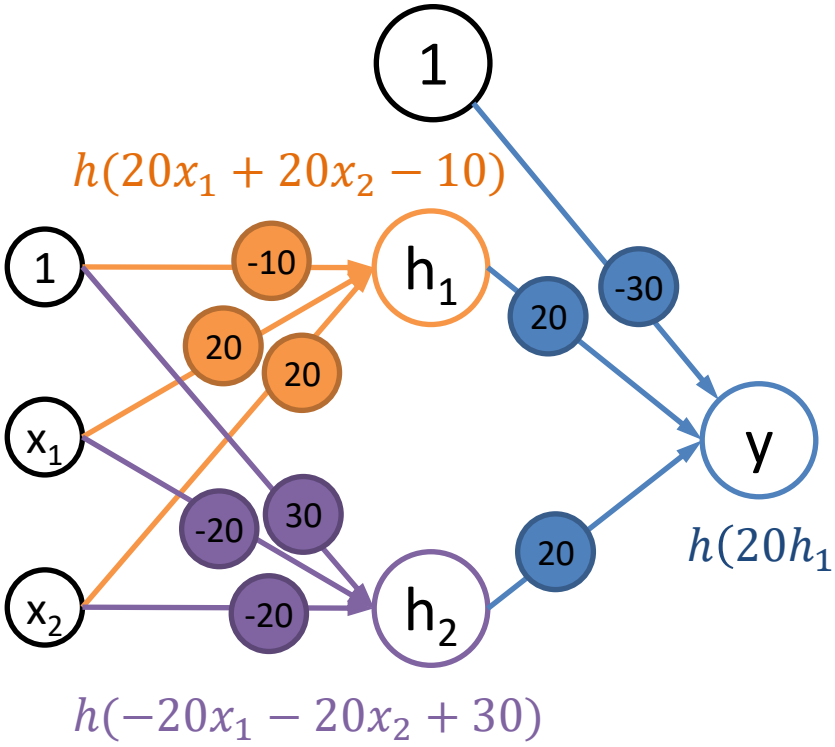
Output: score for each digit

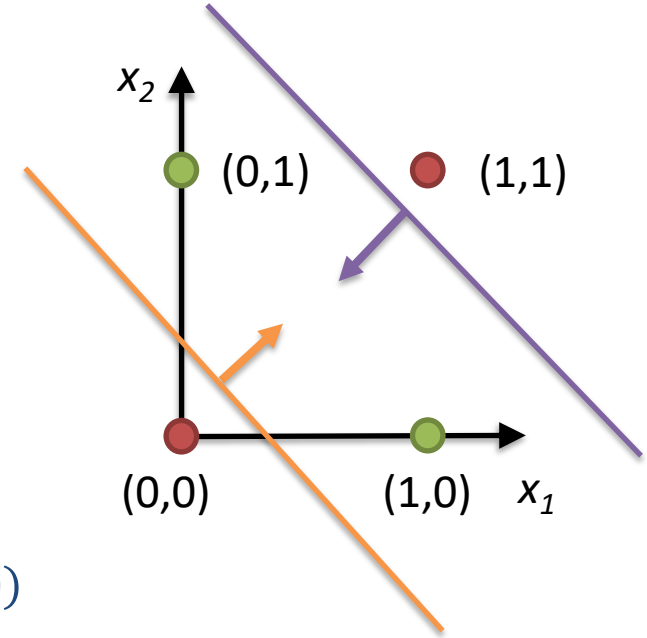
Playground



https://playground.tensorflow.org/

Solving the XOR problem - revisited





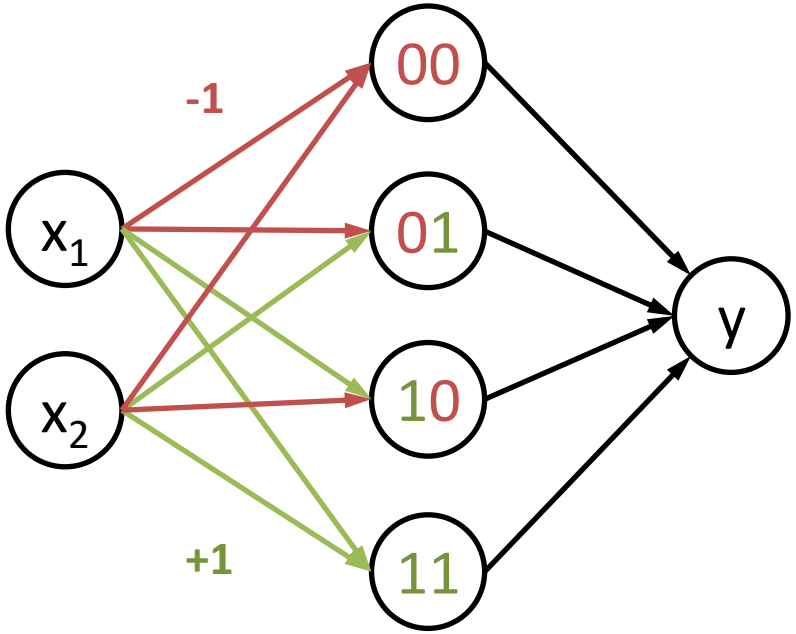
OR	NAND	AND
$\sigma(20 * 0 + 20 * 0 - 10) = 0$	$\sigma(-20 * 0 - 20 * 0 + 30) = 1$	$\sigma(20 * 0 + 20 * 1 - 30) = 0$
$\sigma(20 * 1 + 20 * 1 - 10) = 1$	$\sigma(-20 * 1 - 20 * 1 + 30) = 0$	$\sigma(20 * 1 + 20 * 0 - 30) = 0$
$\sigma(20 * 0 + 20 * 1 - 10) = 1$	$\sigma(-20 * 0 - 20 * 1 + 30) = 1$	$\sigma(20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

Representation power of Neural Nets

A Neural Network with a single hidden layer can represent:

- Any bounded continuous function (to arbitrary ε)
  - Universal Approximation Theorem [Cybenko 1989]
- Any Boolean function (exactly)
  - May require 2^d hidden units for input $x_1, \dots, x_d$

E.g. assume that:
True: +1
False: -1

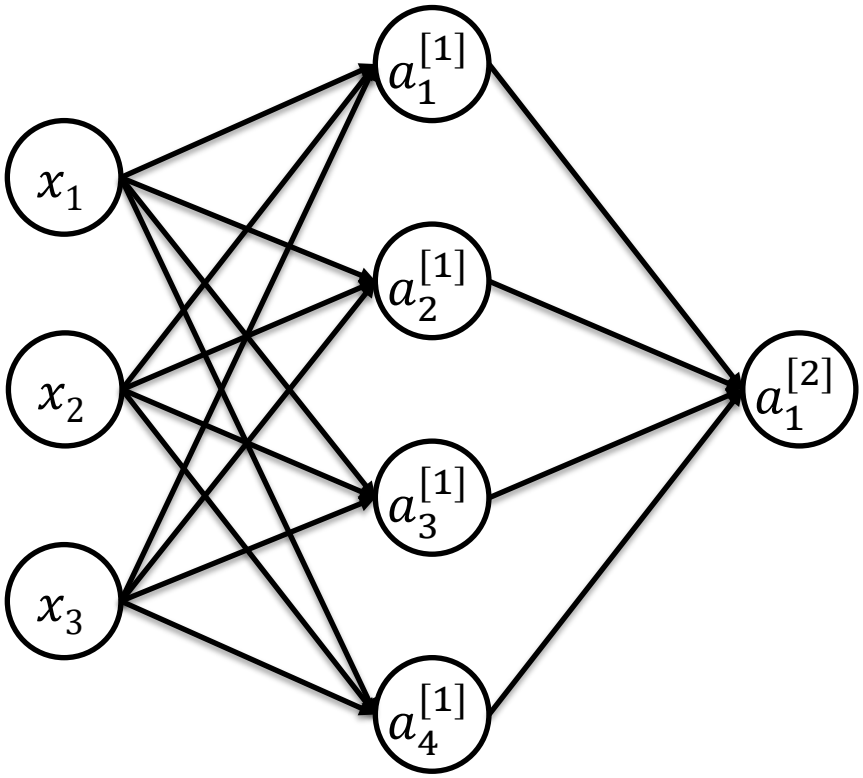


(food for thought: what bias and activation function would you use for this to work?)

Intuitive proof: http://neuralnetworksanddeeplearning.com/chap4.html

NEURAL NETWORKS NOTATION

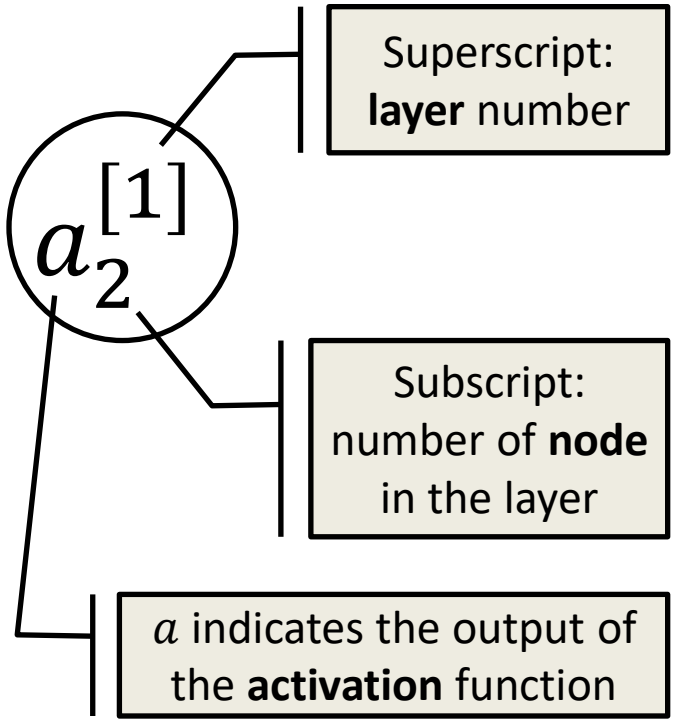
Neural Network Notation



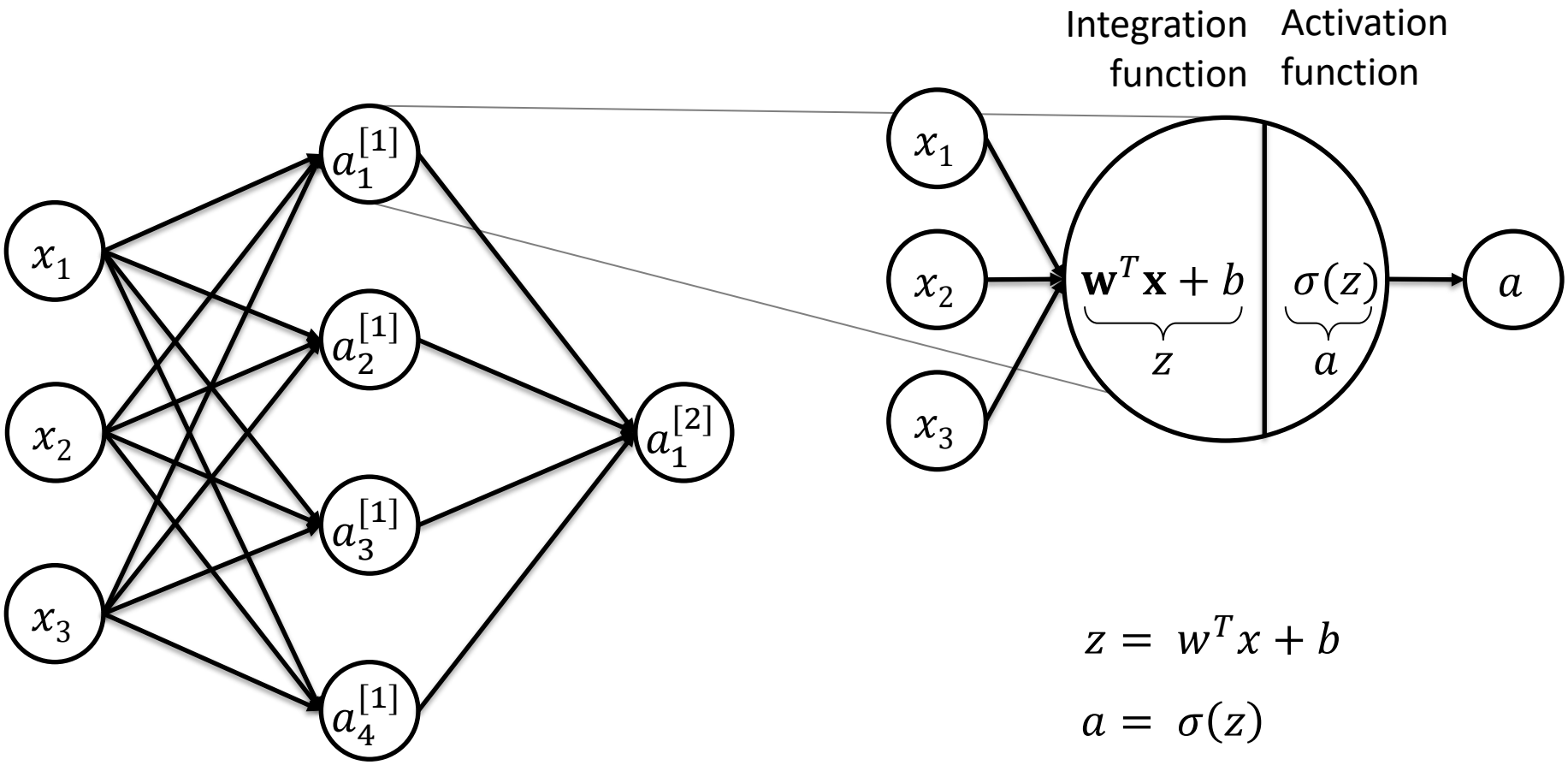
Input layer

Hidden Layer

Output layer



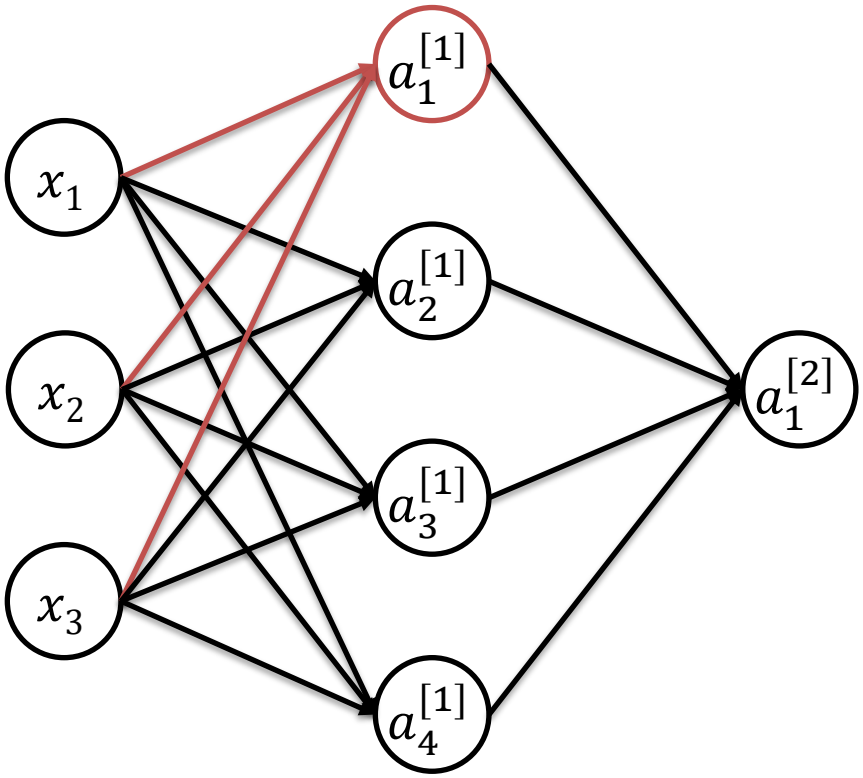
Neural Network Notation



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

$$a = \sigma(z)$$

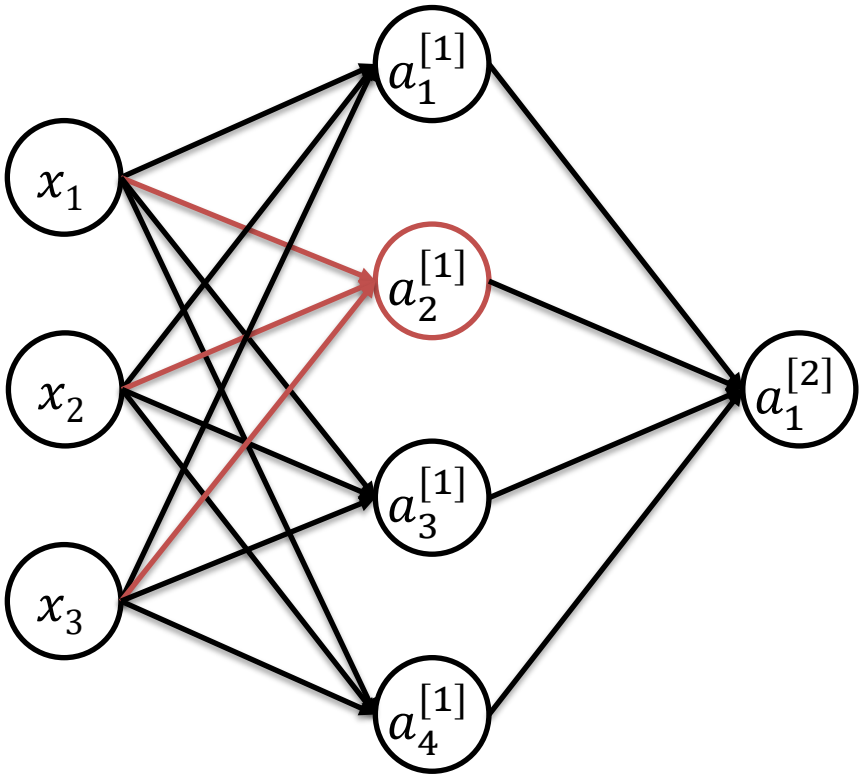
Neural Network Notation



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

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

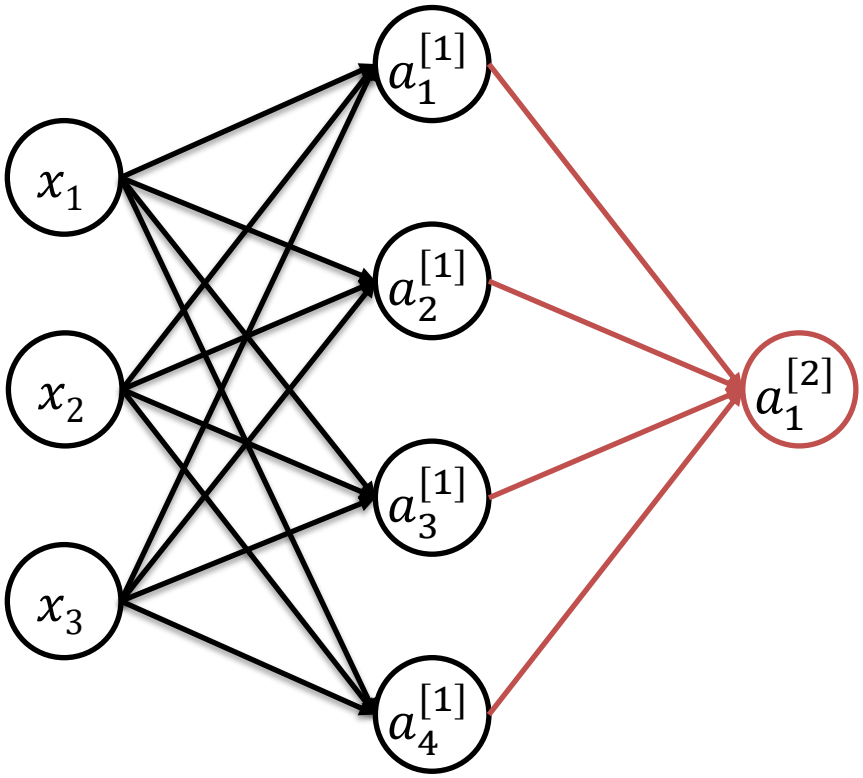
Neural Network Notation



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

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

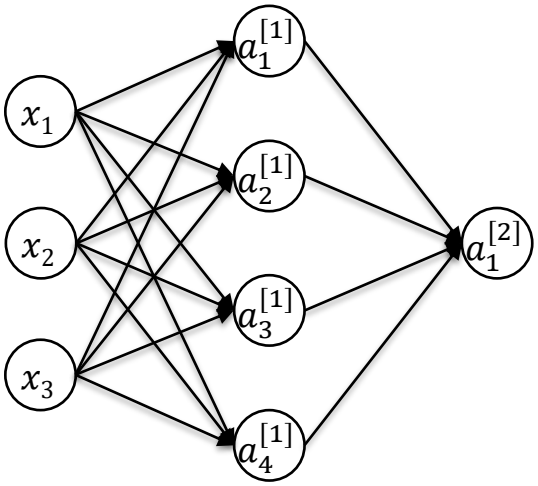
Neural Network Notation



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

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

Neural Network Notation



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

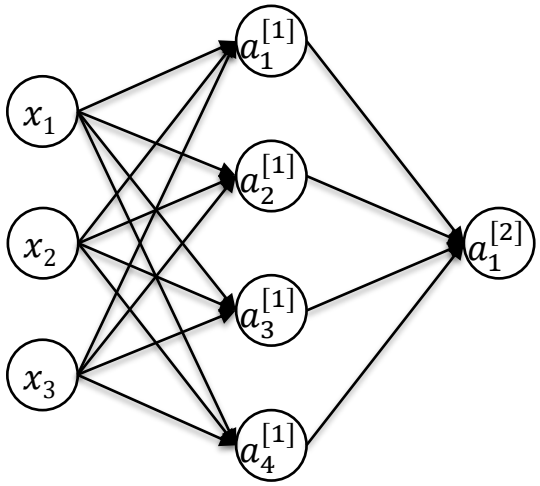
$$\begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \\ z_4^{[1]} \end{bmatrix} = \begin{bmatrix} - & \mathbf{w}_1^{[1]T} & - \\ - & \mathbf{w}_2^{[1]T} & - \\ - & \mathbf{w}_3^{[1]T} & - \\ - & \mathbf{w}_4^{[1]T} & - \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1^{[1]}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

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

$$\mathbf{z}^{[1]} = \mathbf{W}^{[1]T}\mathbf{x} + \mathbf{b}^{[1]}$$

$$\boldsymbol{a}^{[1]} = \sigma\left(\mathbf{z}^{[1]}\right)$$

Neural Network Notation



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

Alternatively, in row notation:

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

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

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

Neural Network Notation

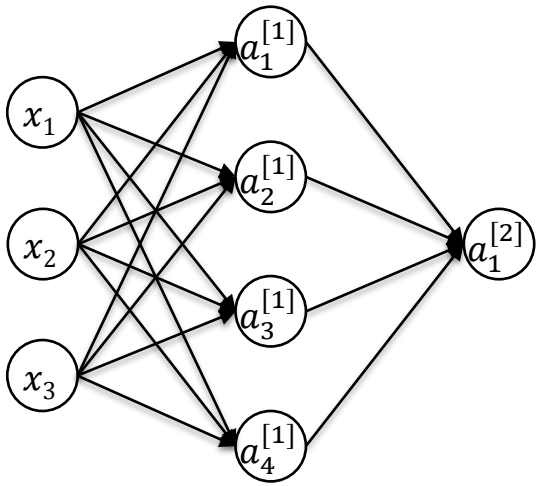
With multiple points, for layer [1]:

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

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

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

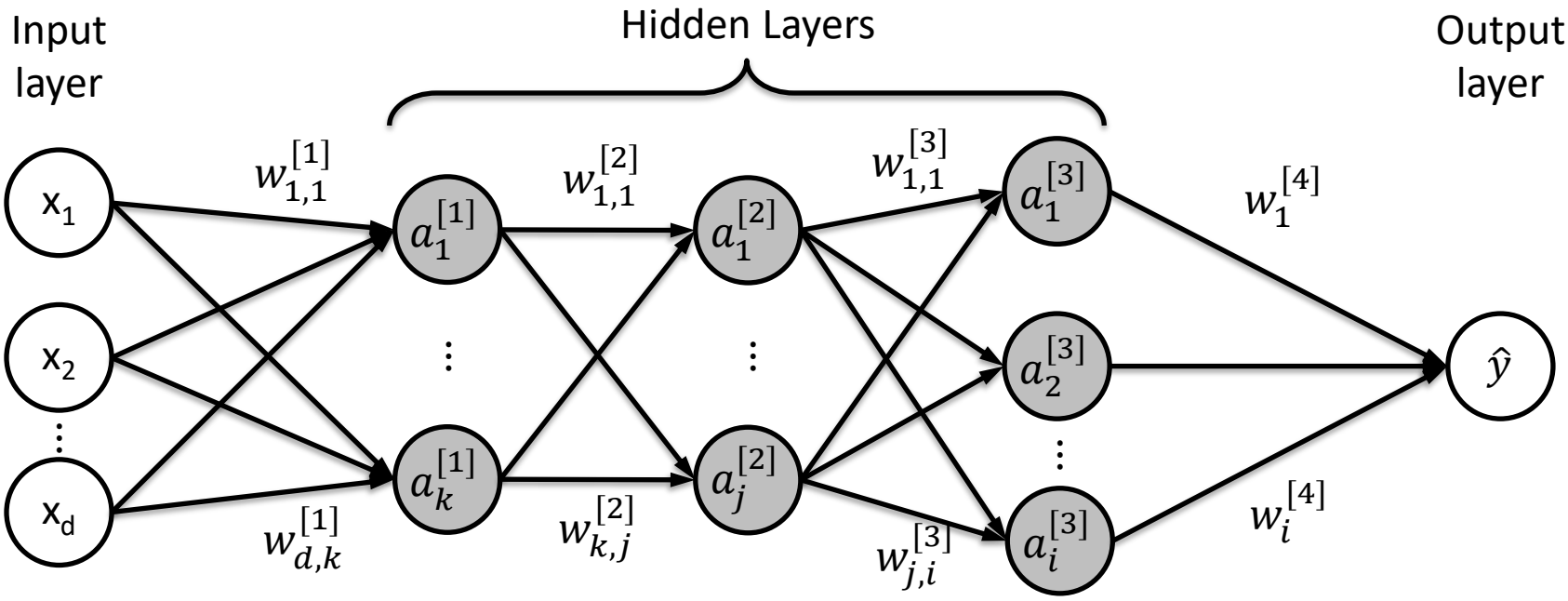
$$\mathbf{Z}^{[L]} = \mathbf{A}^{[L-1]}\mathbf{W}^{[L]} + \mathbf{B}^{[L]}$$

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

Multi-layer neural networks

LEARNING WITH HIDDEN UNITS

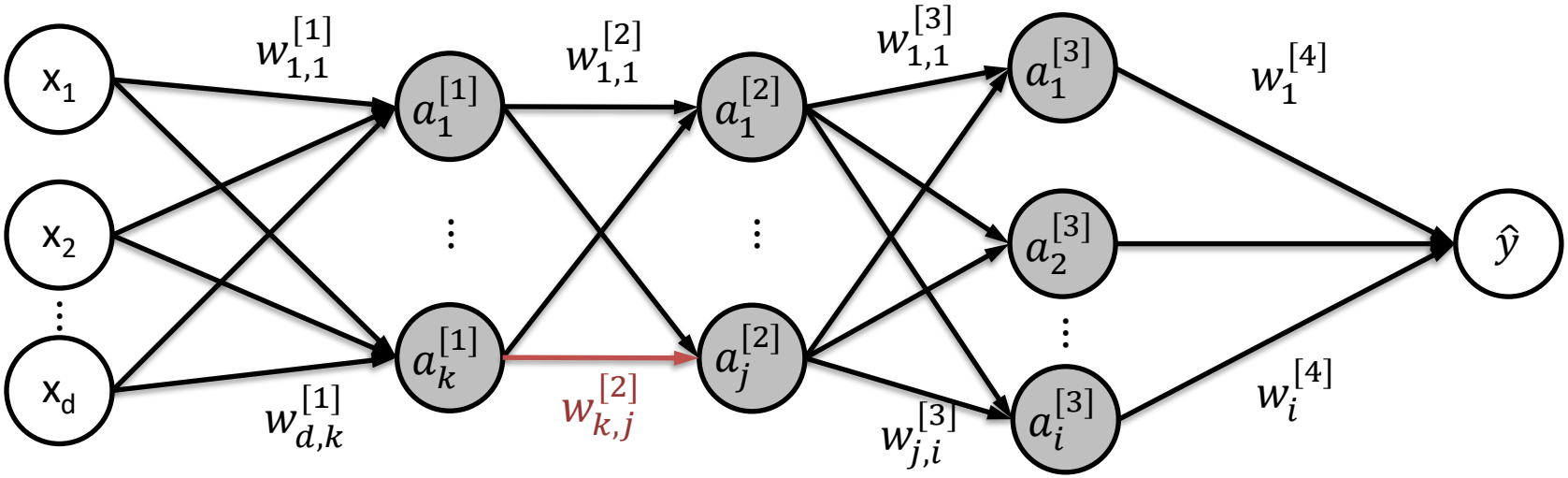
Backpropagation Algorithm



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

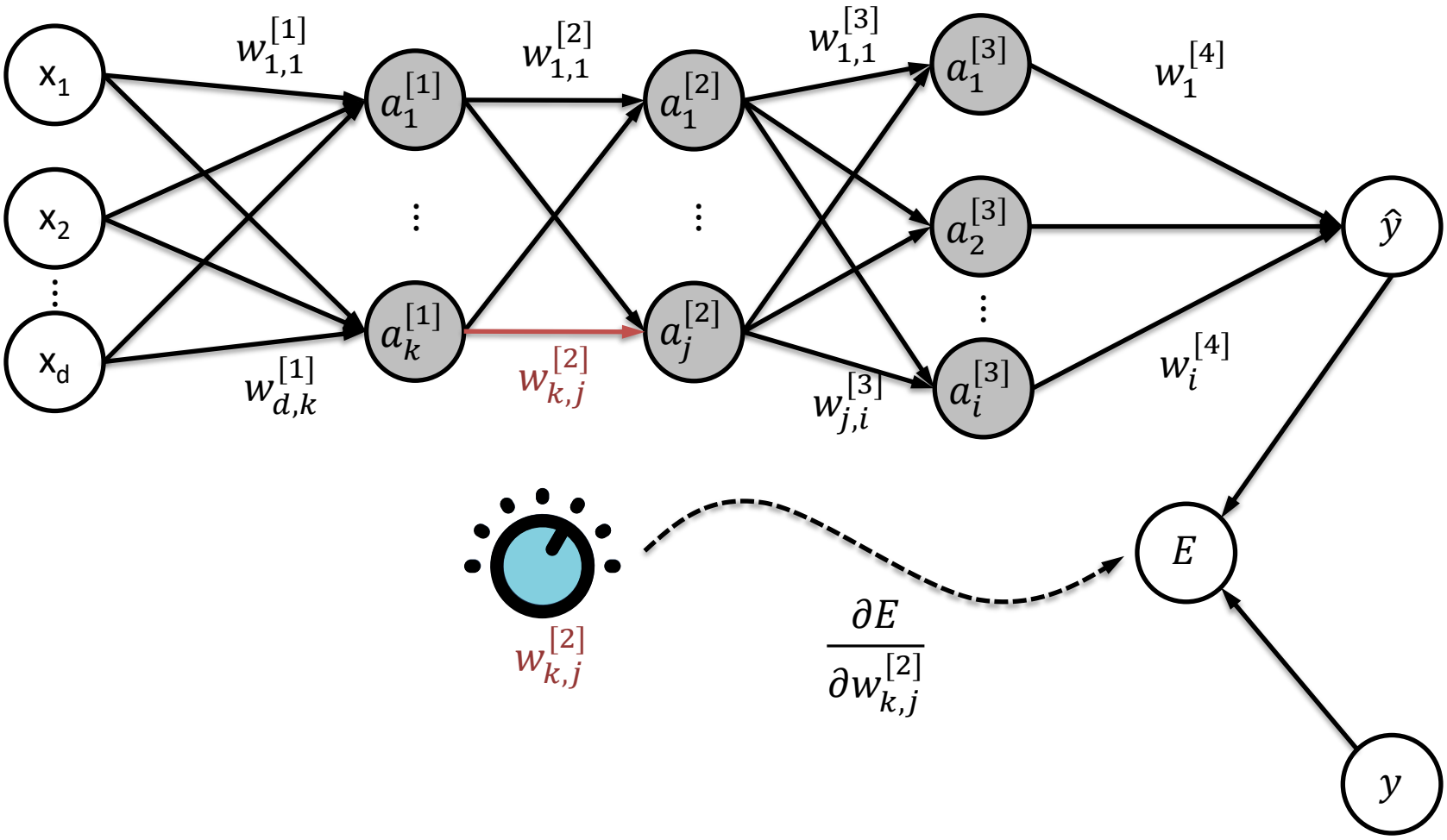
How should I change $w_{j,k}^{[2]}$?



1. Receive new observation $\mathbf{x} = [x_1, x_2, \ldots, x_d]$ 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?

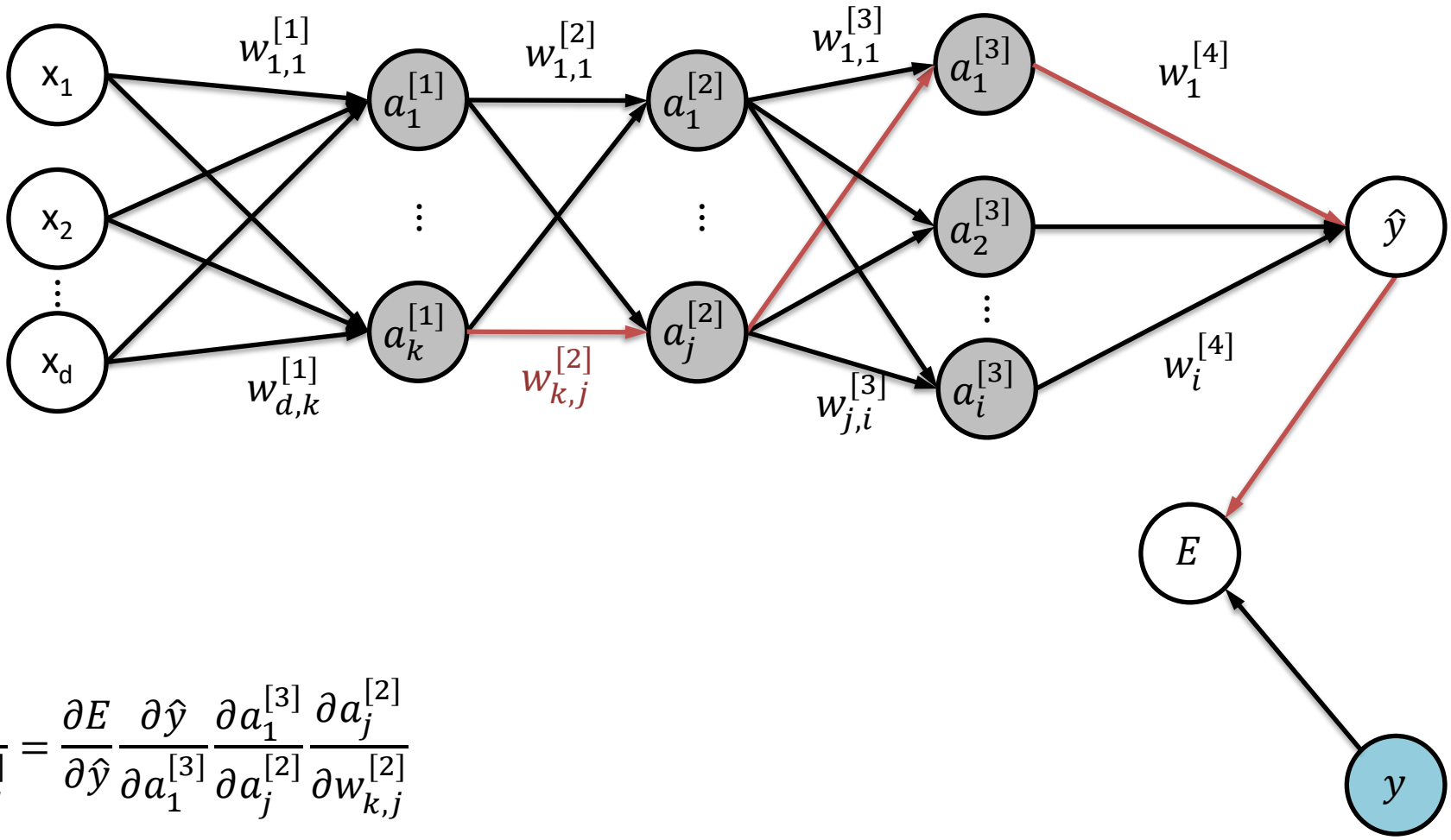
Backpropagation Algorithm

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



Backpropagation Algorithm

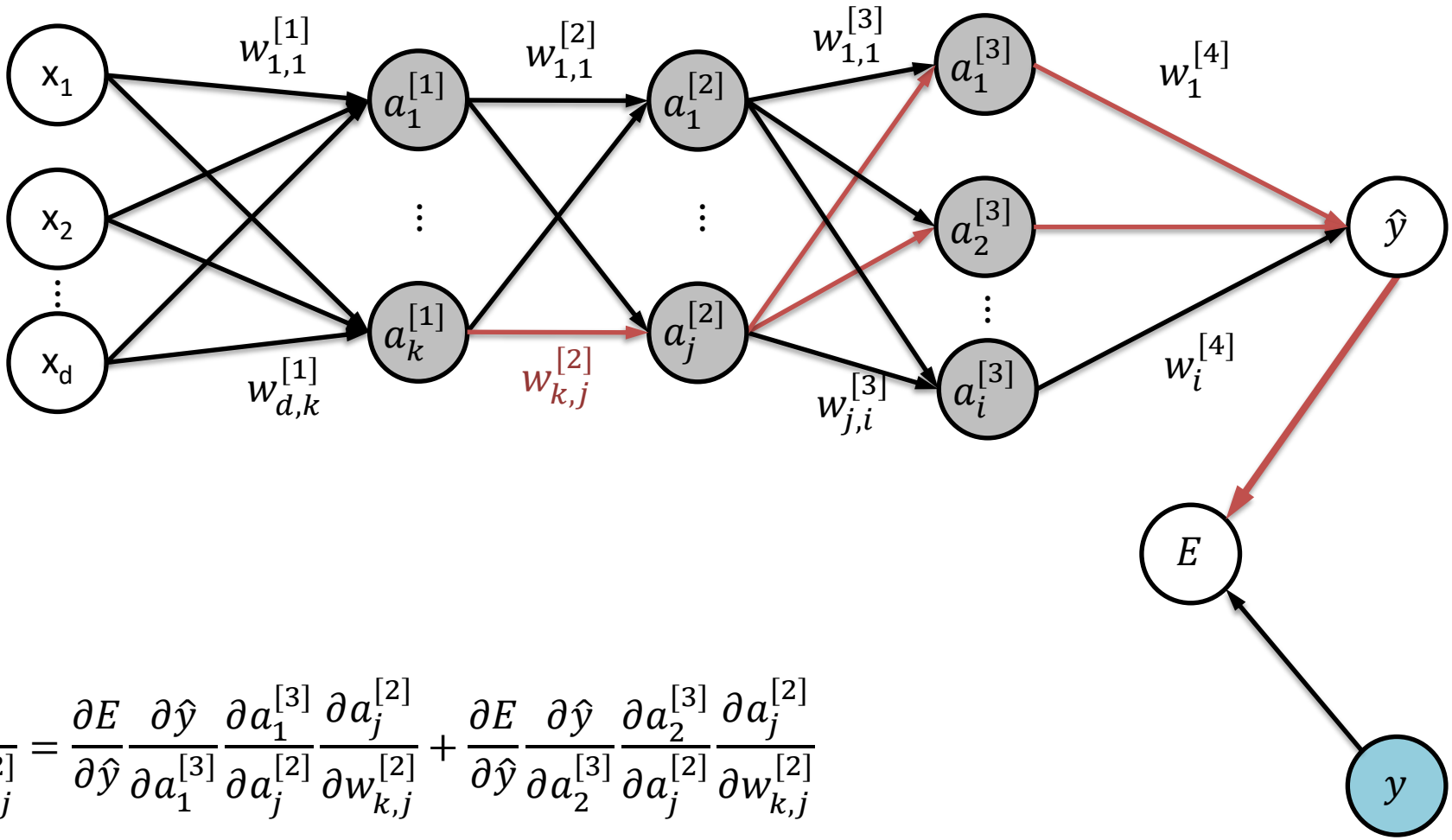
How should I change $w_{j,k}^{[2]}$?



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

Backpropagation Algorithm

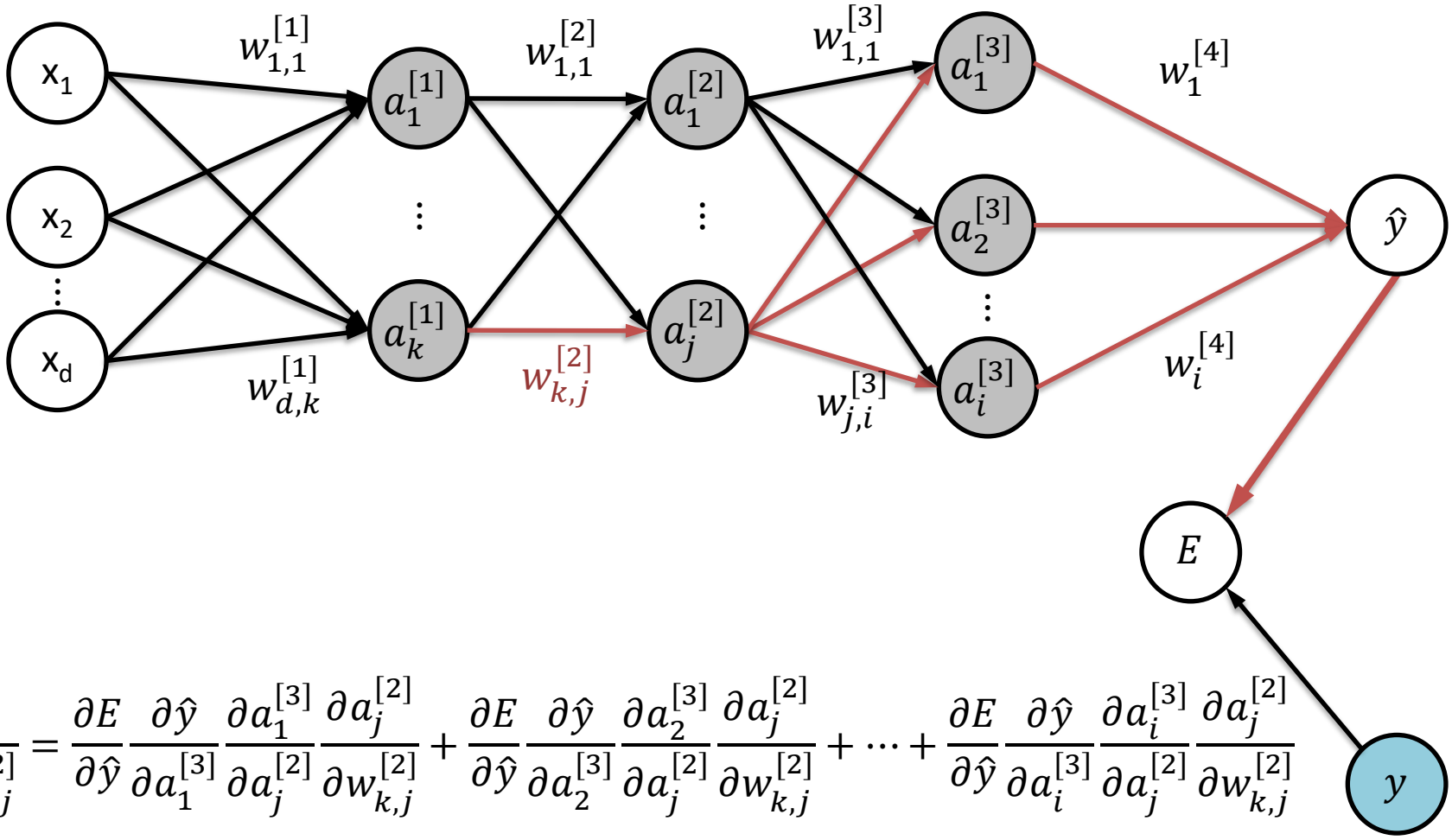
How should I change $w_{j,k}^{[2]}$?



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

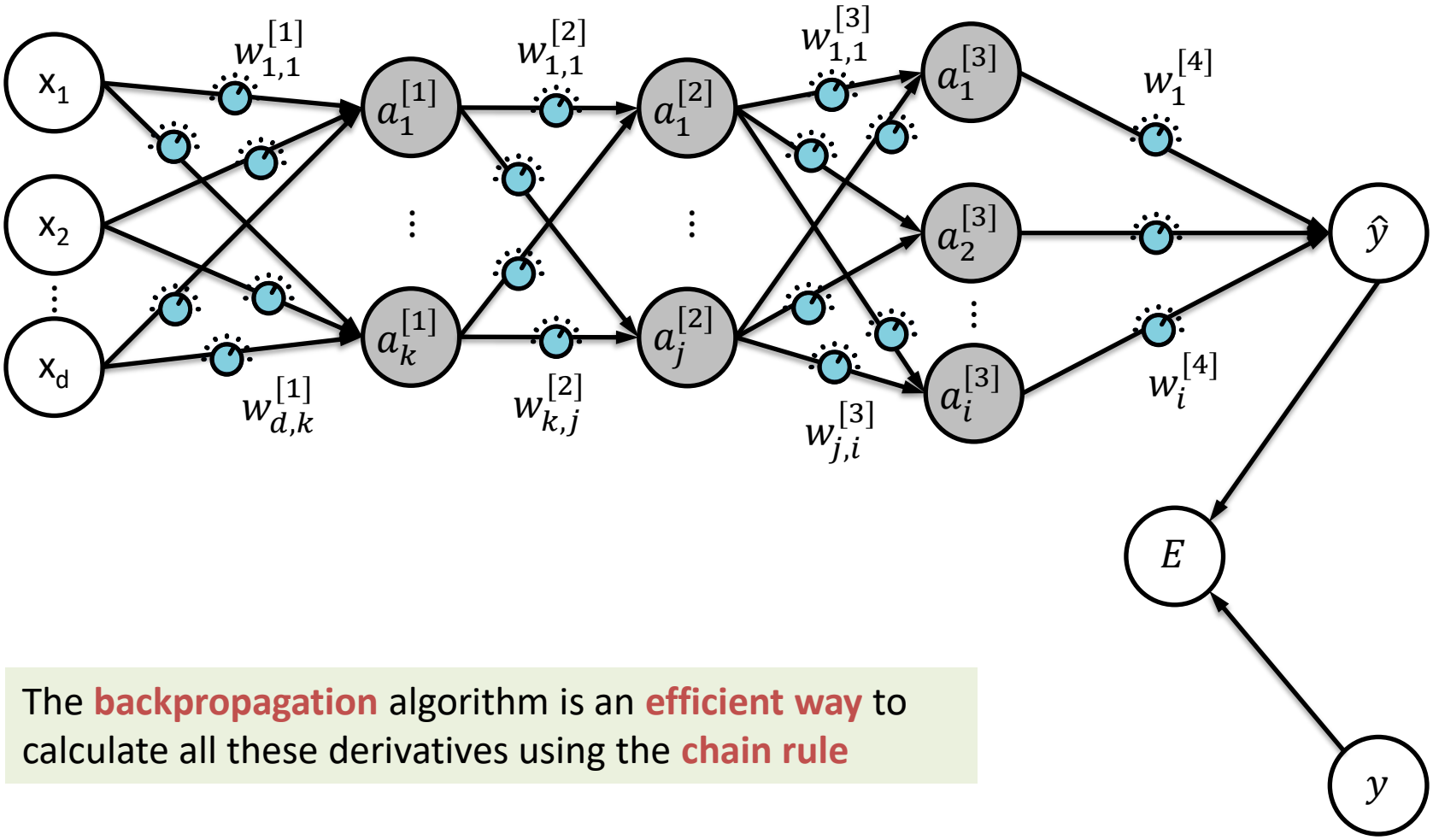
Backpropagation Algorithm

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



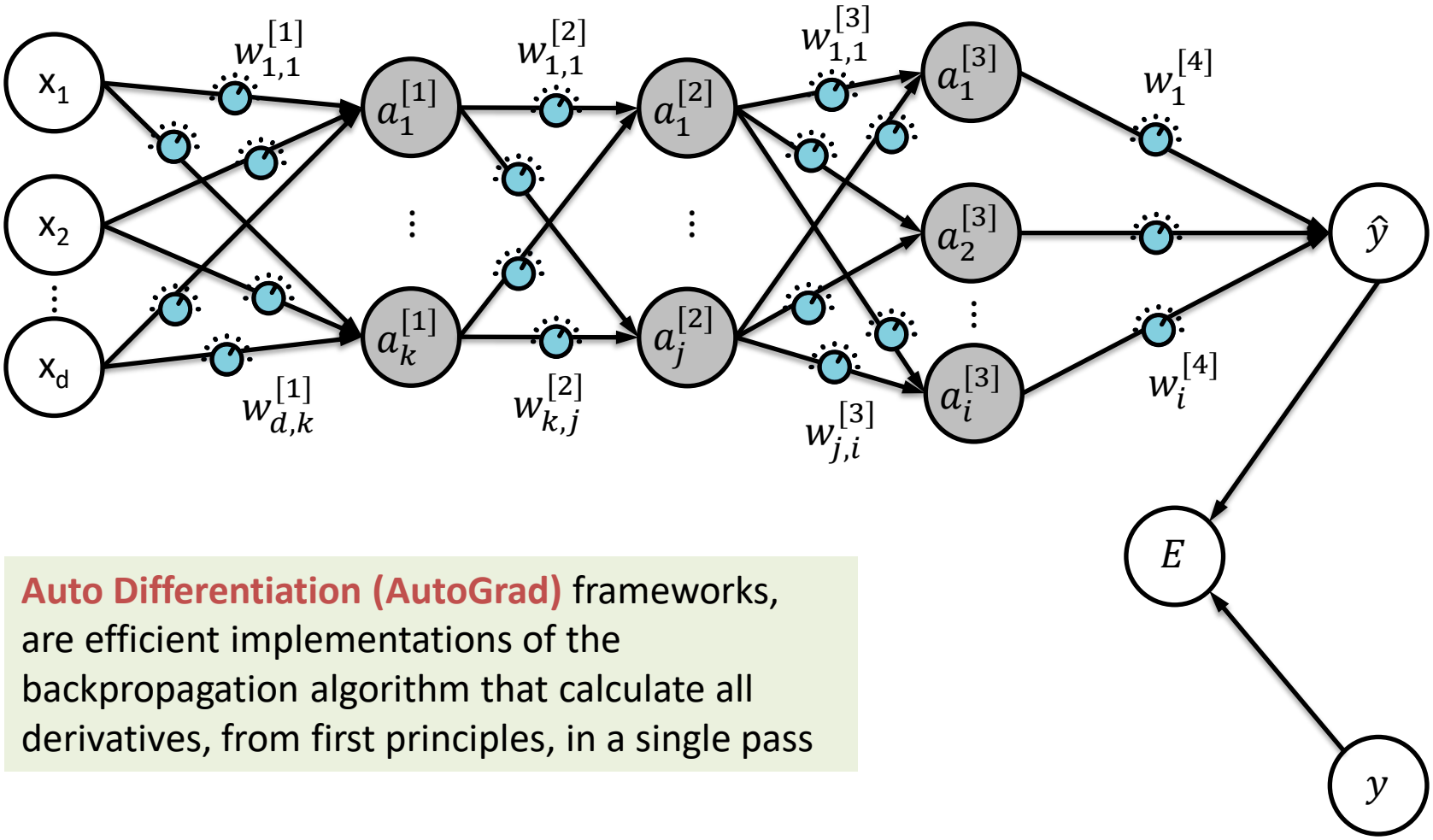
$$\frac{\partial E}{\partial w_{k,j}^{[2]}} = \frac{\partial E}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial a_1^{[3]}} \frac{\partial a_1^{[3]}}{\partial a_j^{[2]}} \frac{\partial a_j^{[2]}}{\partial w_{k,j}^{[2]}} + \frac{\partial E}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial a_2^{[3]}}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

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

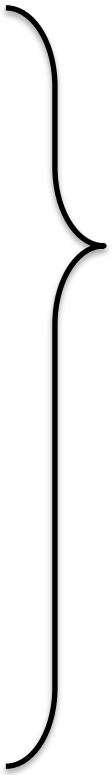
AUTO DIFFERENTIATION

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: $\frac{\partial d}{\partial a}$?

$$d = a * c$$

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

$$= c + a * \frac{\partial c}{\partial a}$$

$$= (a + b) + a * \frac{\partial (a + b)}{\partial a}$$

$$= a + b + a * \left(\frac{\partial a}{\partial a} + \frac{\partial b}{\partial a}\right)$$

$$= a + b + a * (1 + 0)$$

$$= a + b + a = 2a + b$$

$$= 2 * 4 + 3 = 11$$

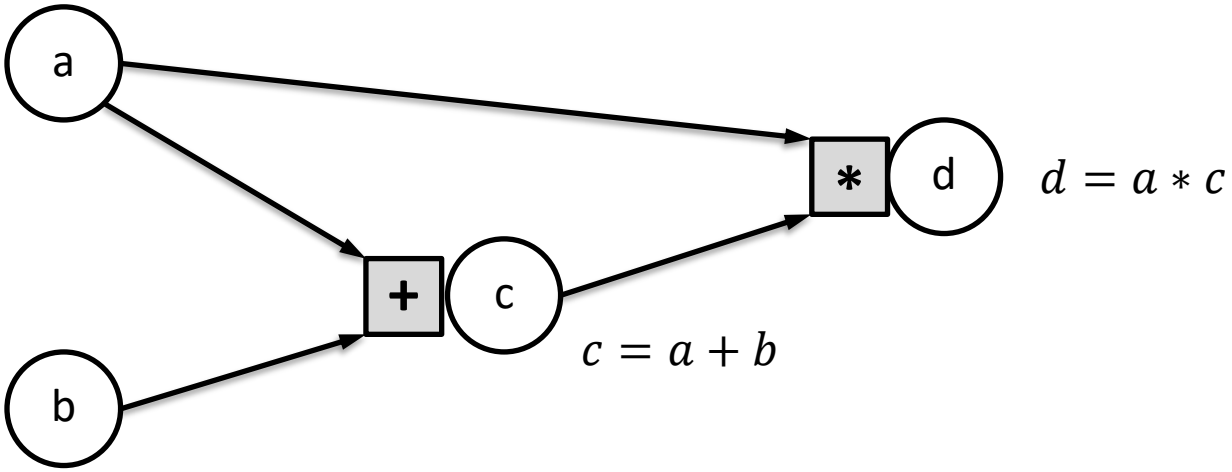
Solving the traditional way

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…

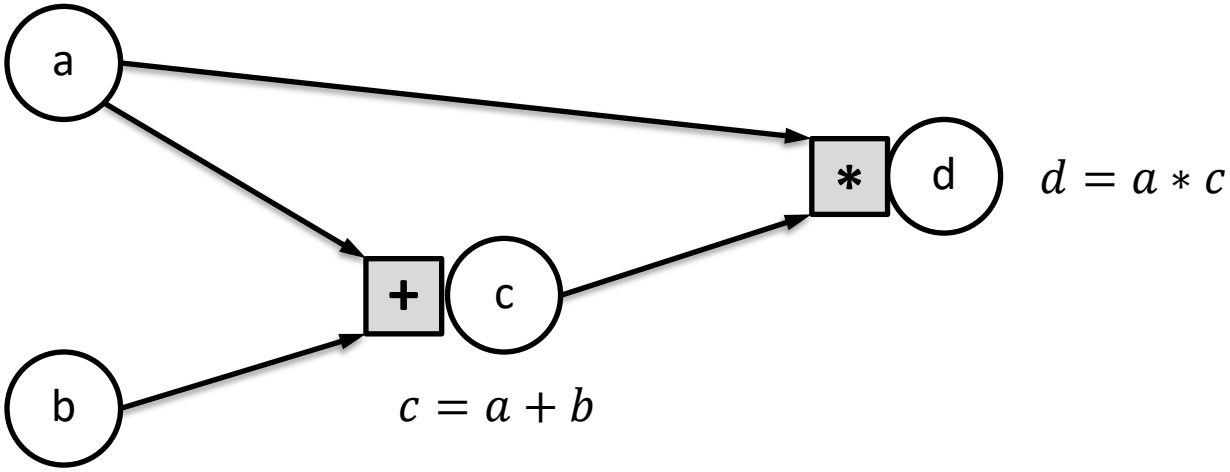
Computational Graph

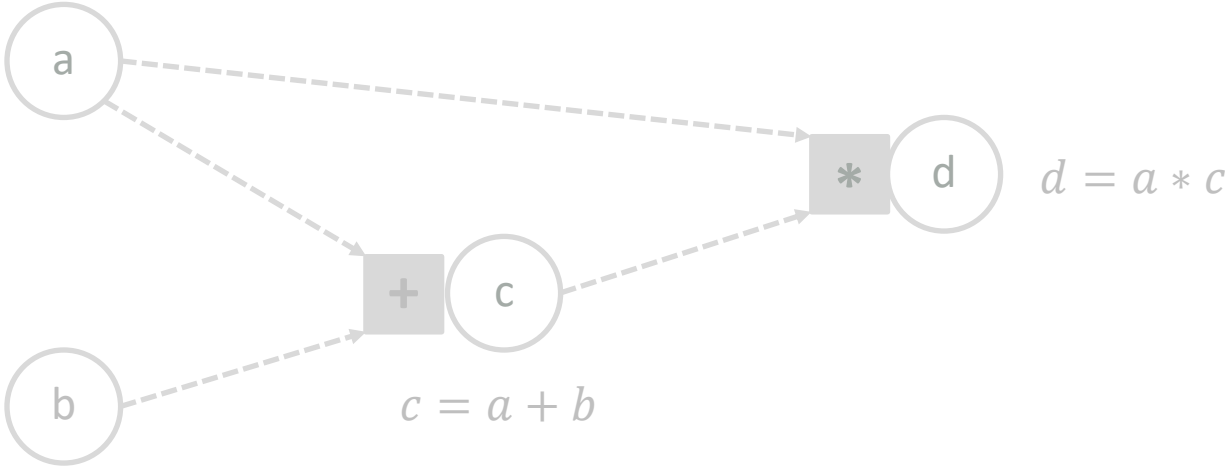
```
a = 4
b = 3
c = a + b # = 4 + 3 = 7
d = a * c # = 4 * 7 = 28
```



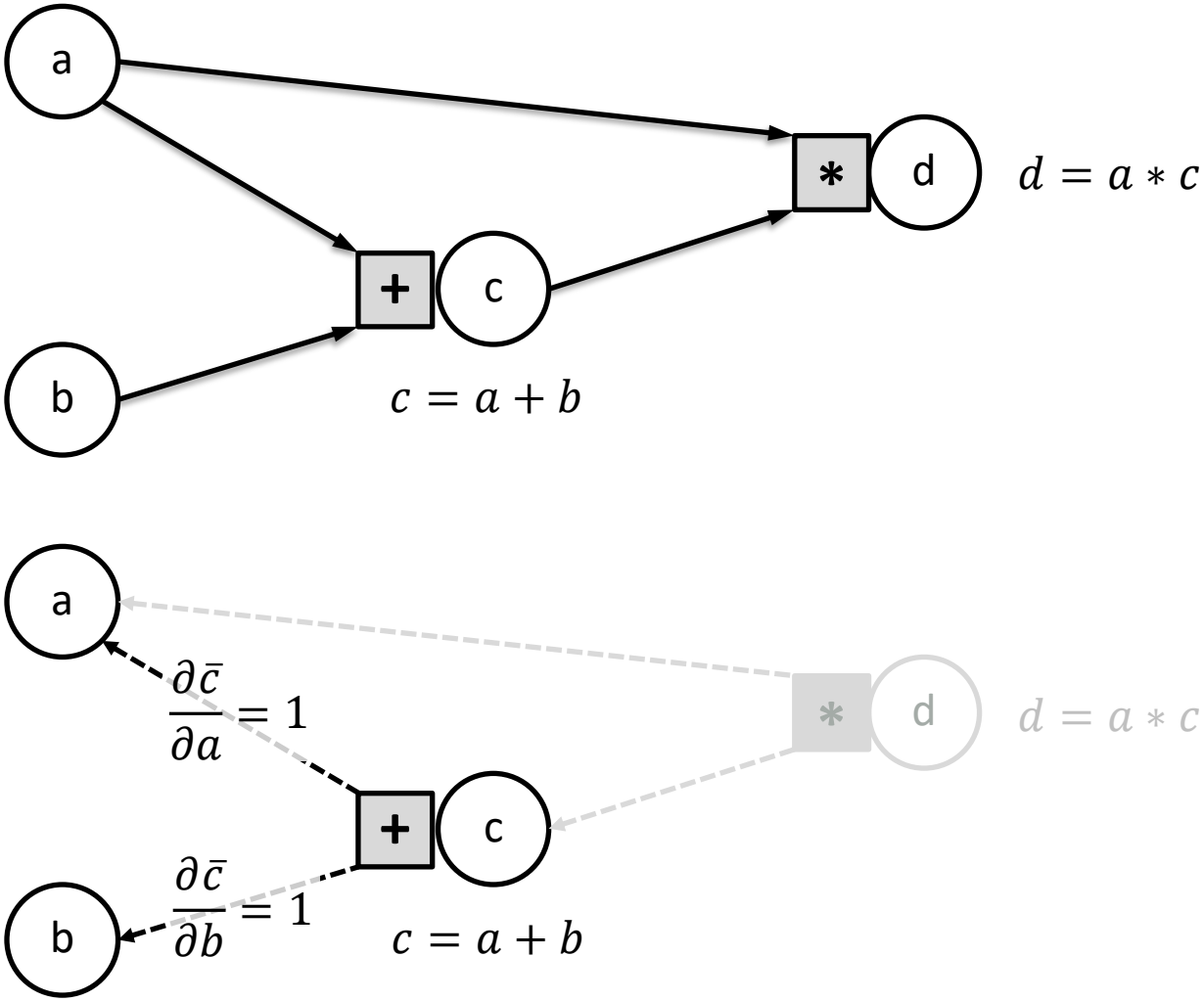
Any expression can be broken down into a series of **simple operations** that are **applied sequentially**

Local Derivatives

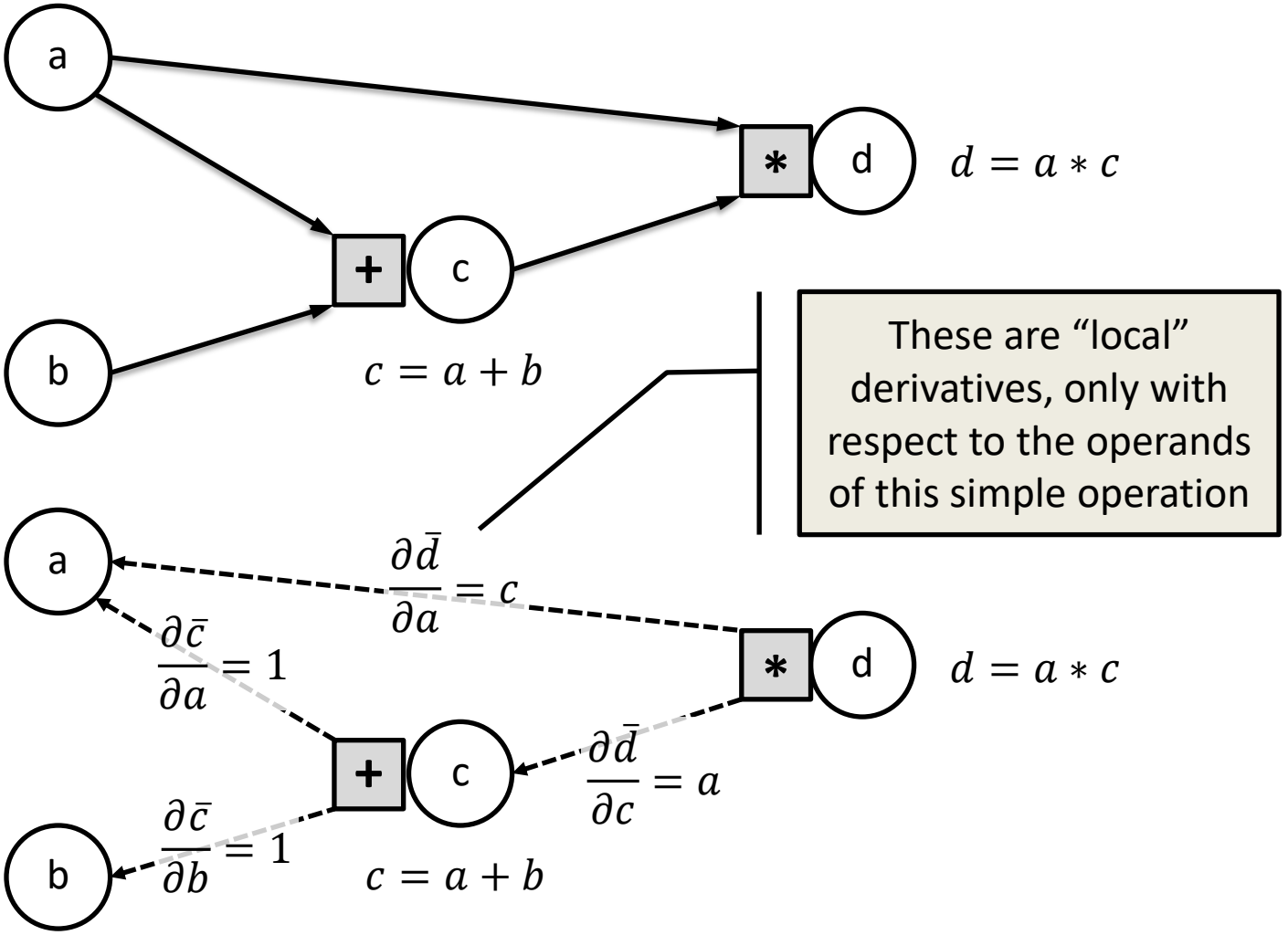




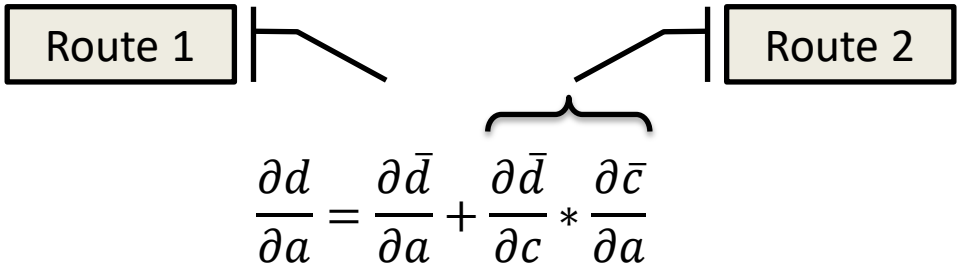
Local Derivatives

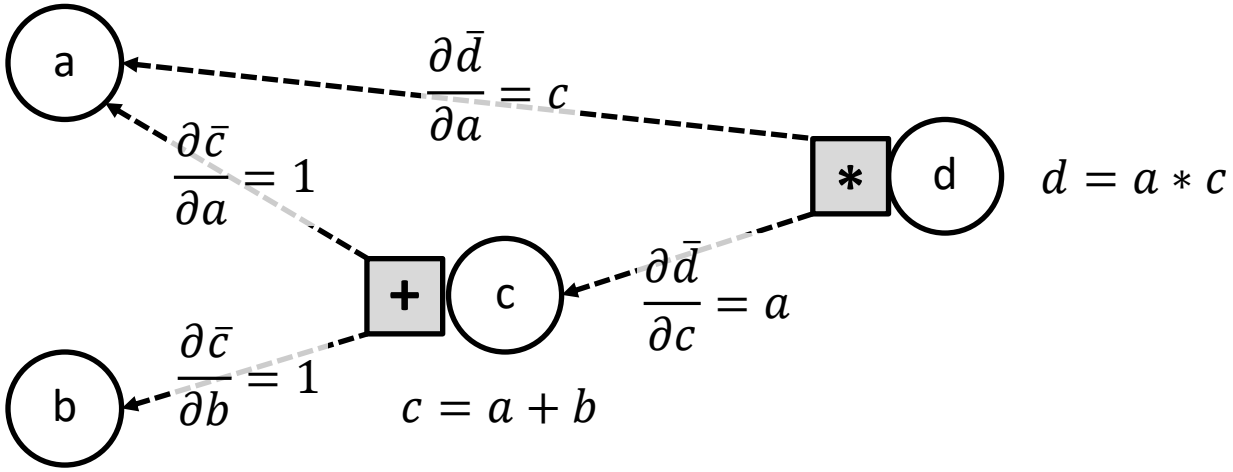


Local Derivatives



Automatic Differentiation (AutoGrad)



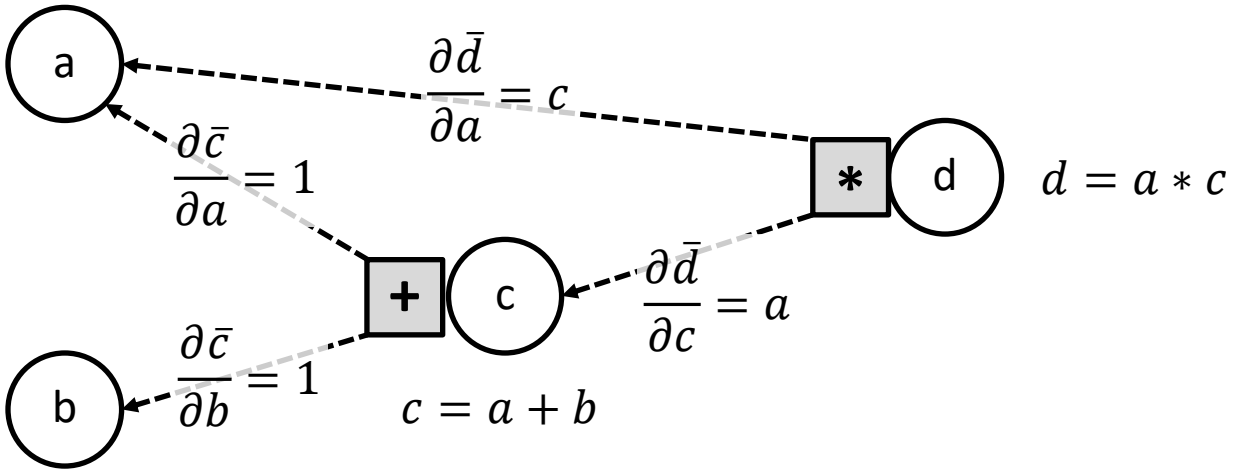


Automatic Differentiation (AutoGrad)

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

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

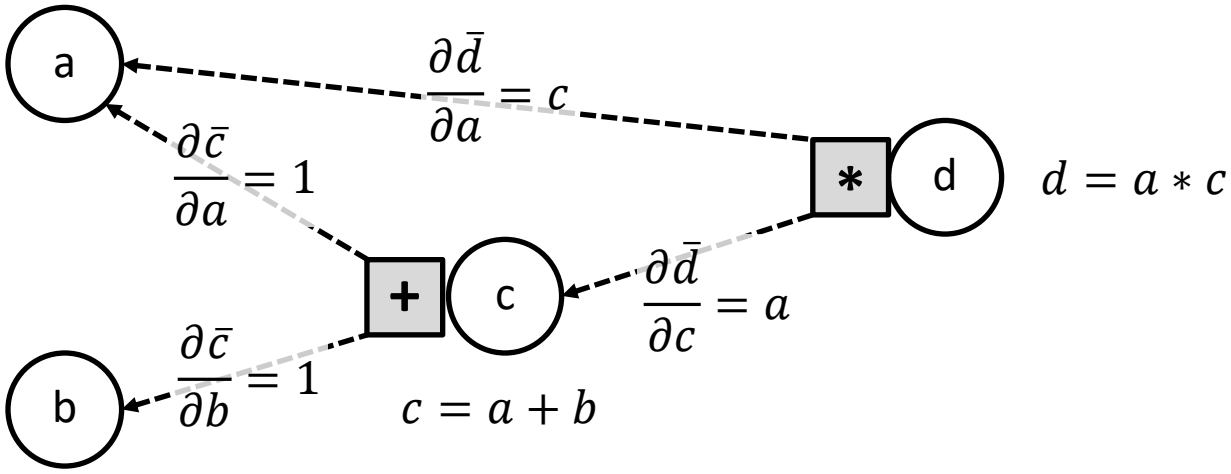
- **Multiply** the edges of a route
- **Add** together the different routes that lead from the quantity to derive to the node of interest



Automatic Differentiation (AutoGrad)

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

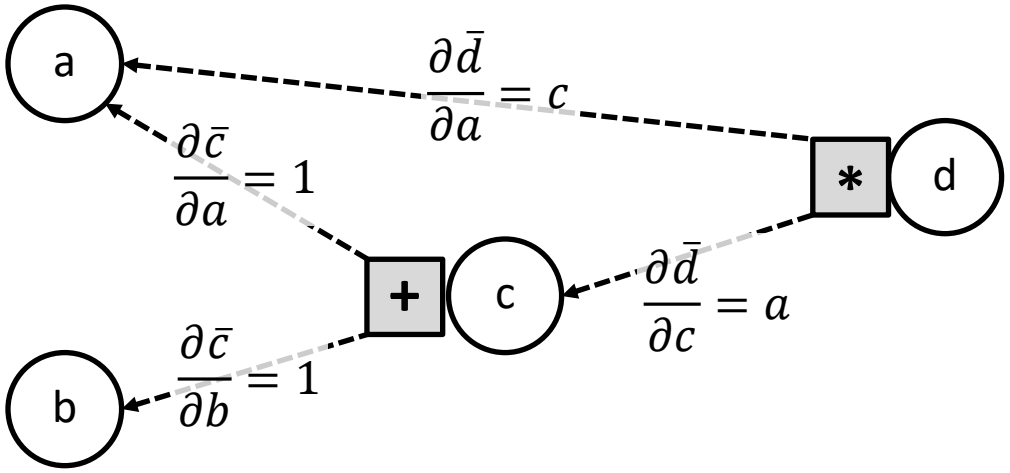


The Backwards Pass

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



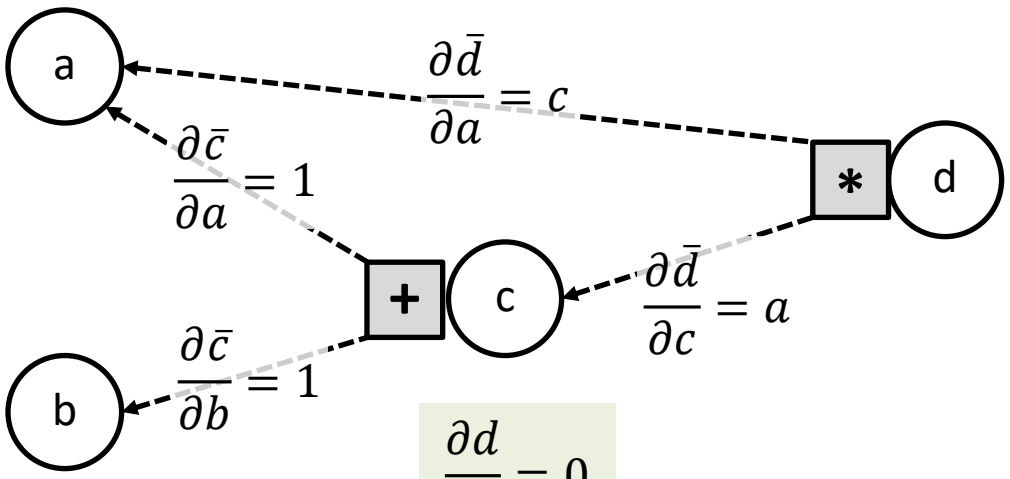
The Backwards Pass

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

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



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

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

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

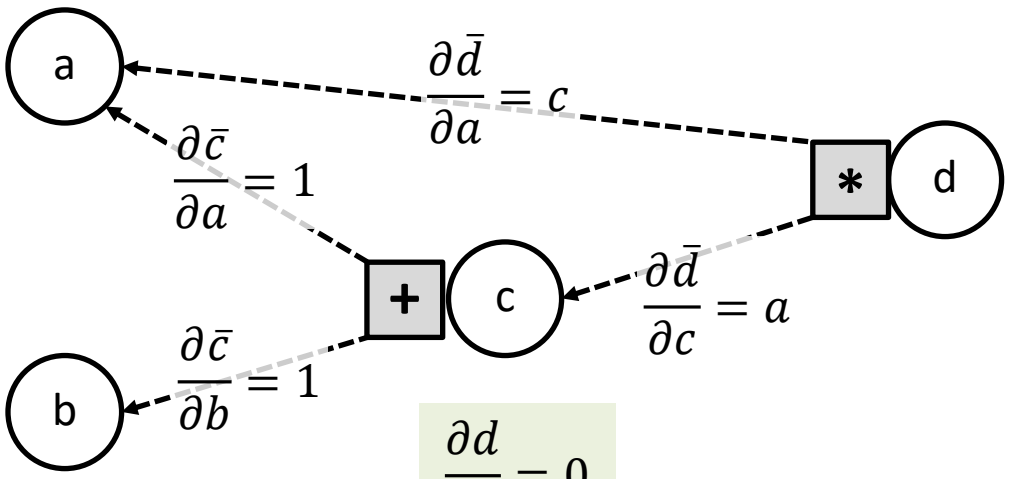
The Backwards Pass

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

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



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

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

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

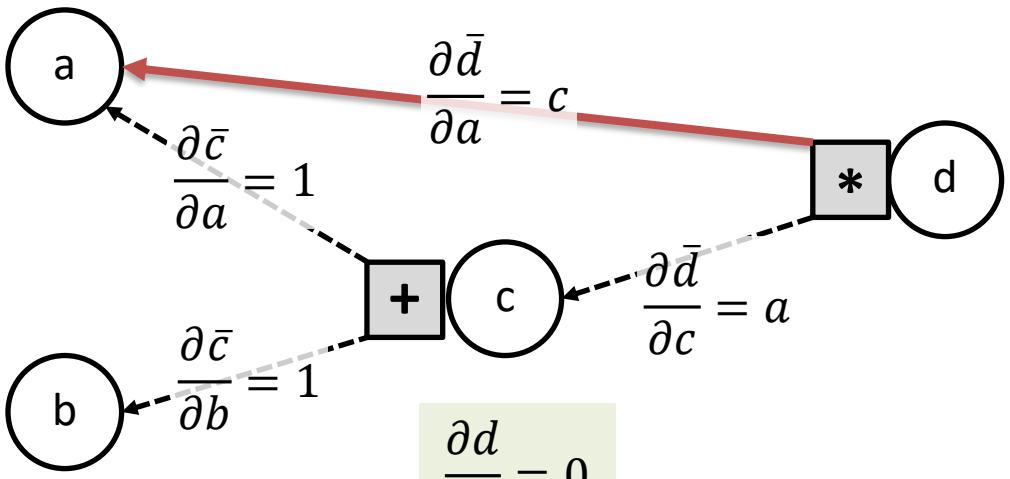
The Backwards Pass

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

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



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

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

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

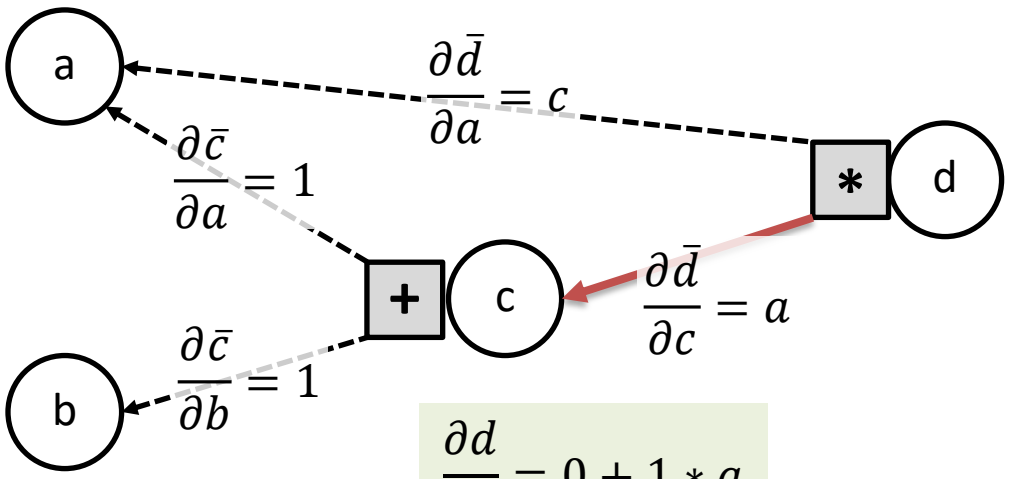
The Backwards Pass

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

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



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

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

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

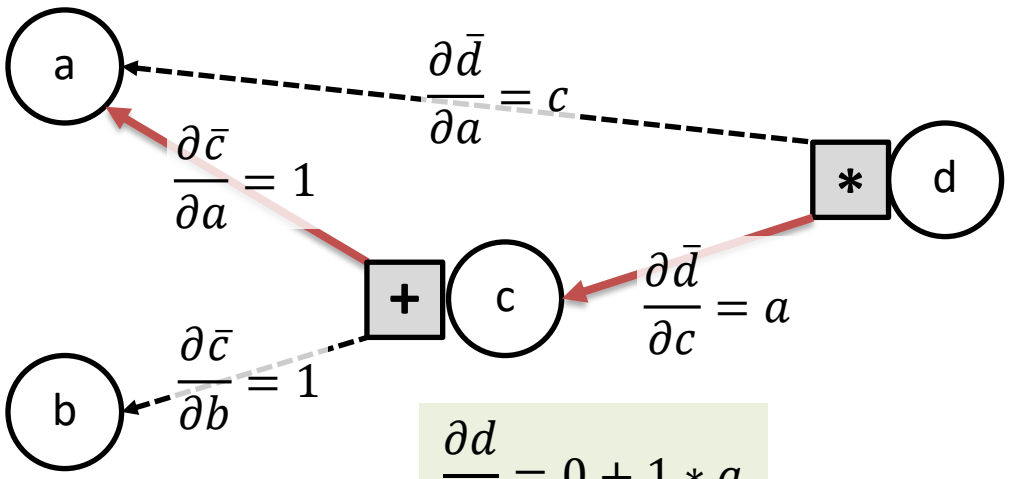
The Backwards Pass

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

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



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

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

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

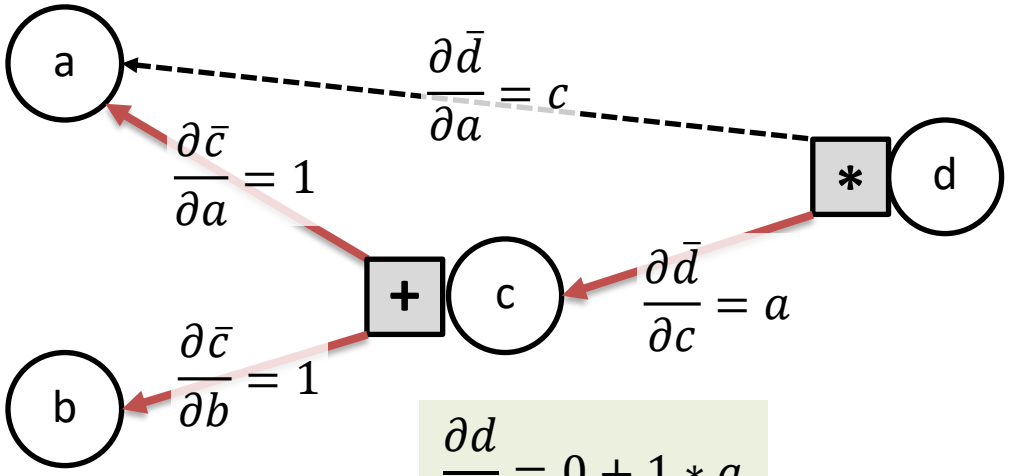
The Backwards Pass

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

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



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

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

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

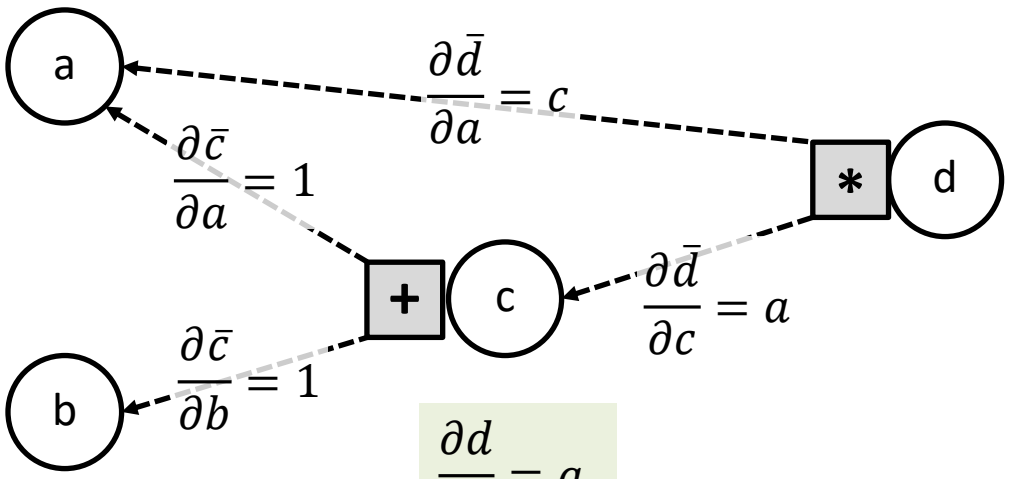
The Backwards Pass

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

$$\frac{\partial d}{\partial a} = c + a$$



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

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

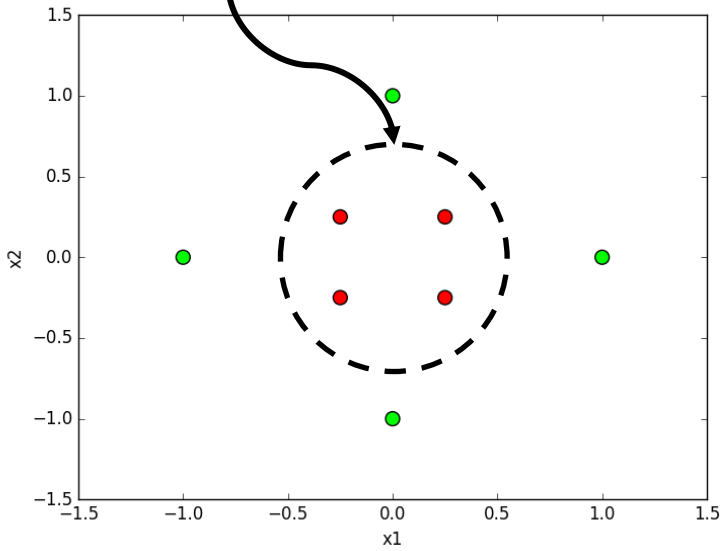
$$\frac{\partial d}{\partial b} = a$$

Using our AutoGrad framework

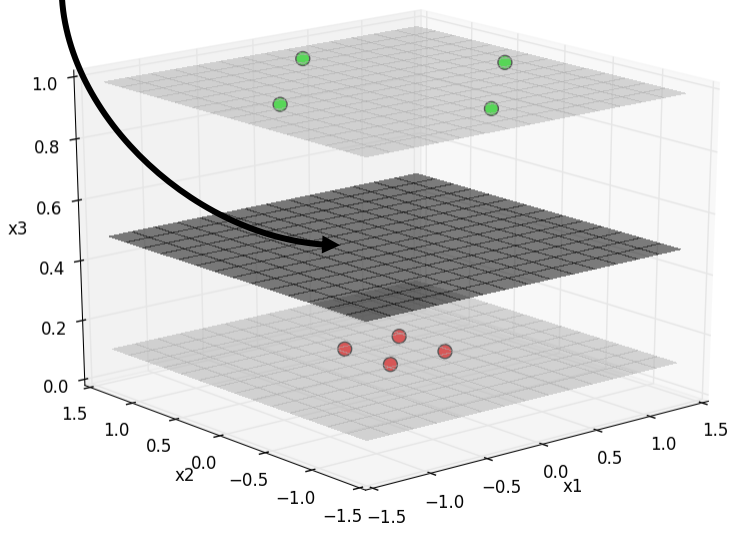
LEARNING THE PARAMETERS OF COMPOSITE FUNCTIONS

Non linear decision boundaries

$x_1^2 + x_2^2 = 0.5$

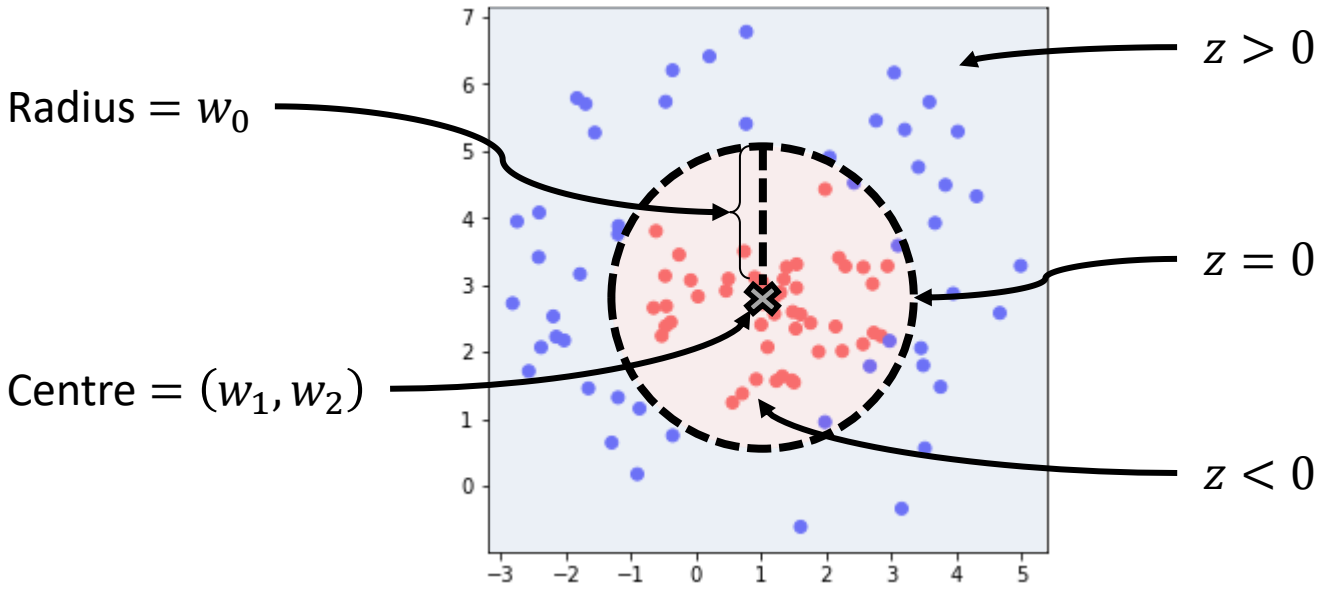


$x_3 = 0.5$



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

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 $\partial z / \partial w_i$

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

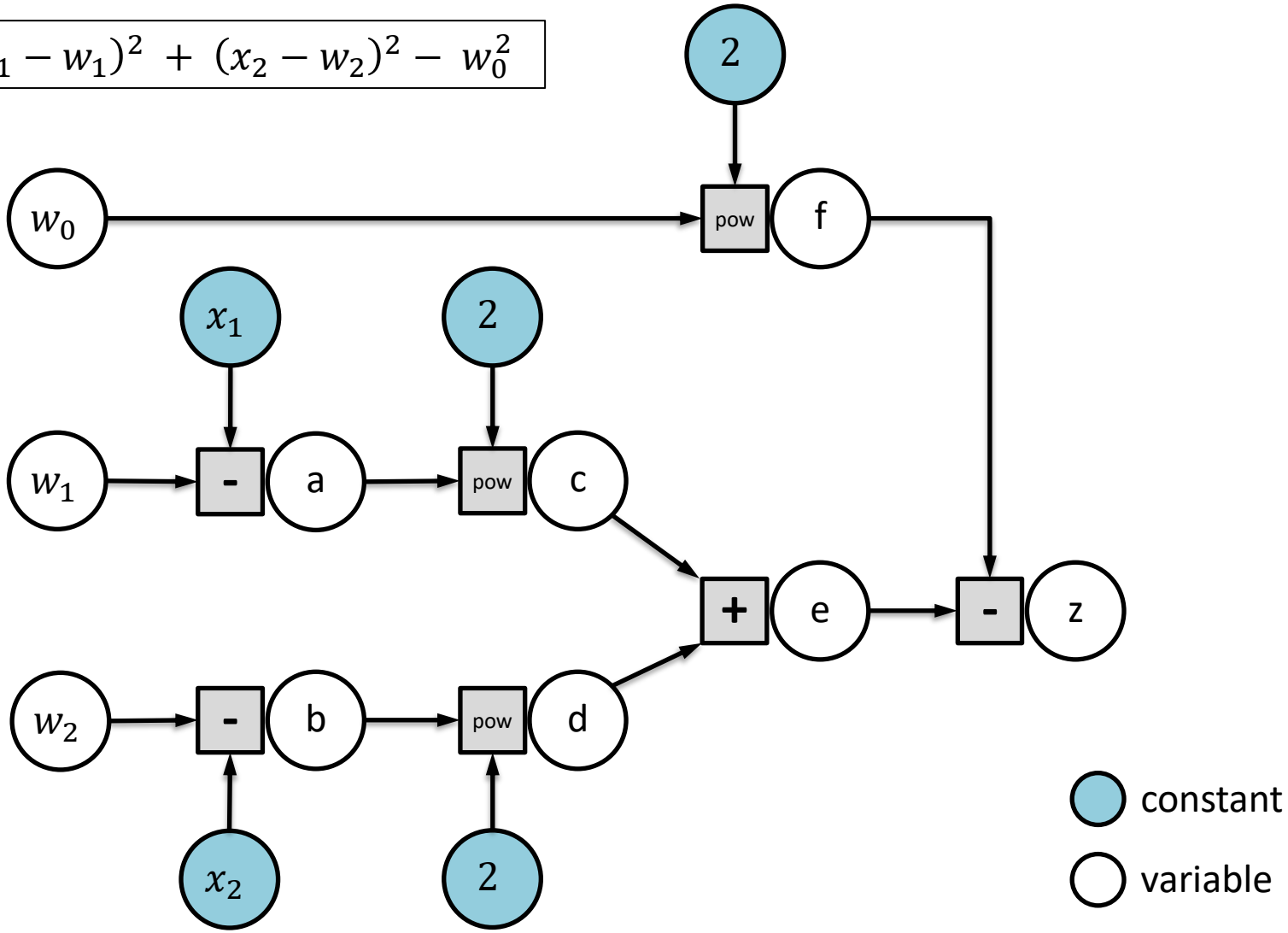
$$\frac{\partial z}{\partial w_0} = ? \quad \frac{\partial z}{\partial w_1} = ? \quad \frac{\partial z}{\partial w_2} = ?$$

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

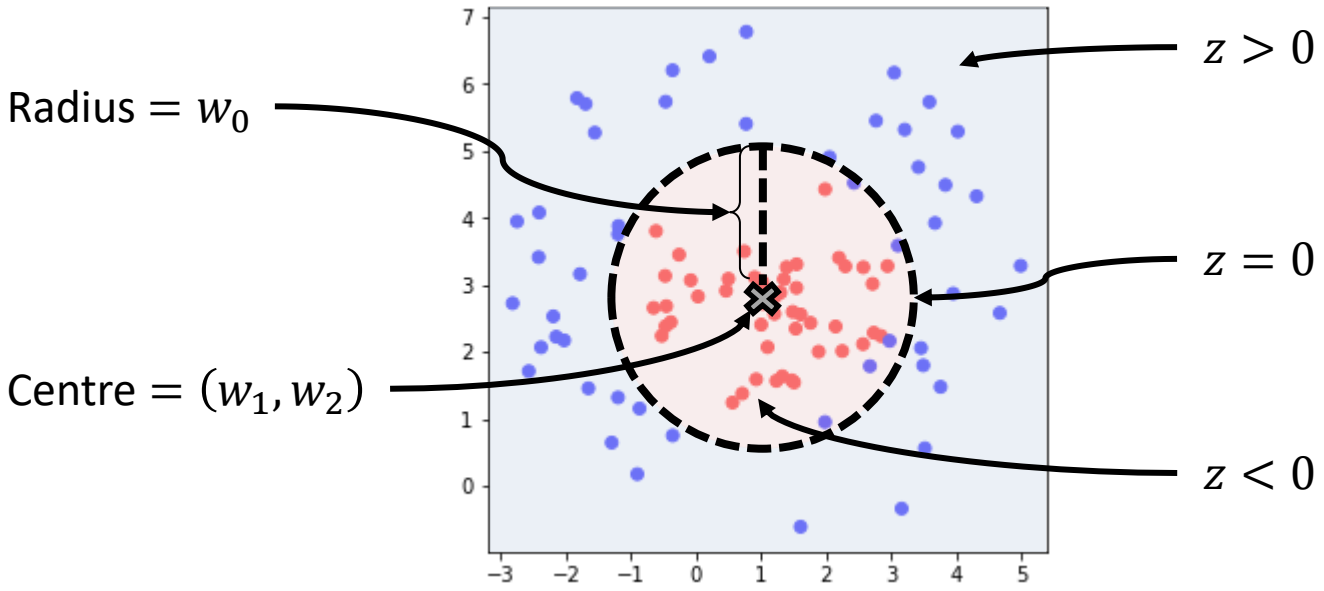
$$w_i \leftarrow w_i - \alpha \frac{\partial E}{\partial w_i}$$

Computational Graph

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



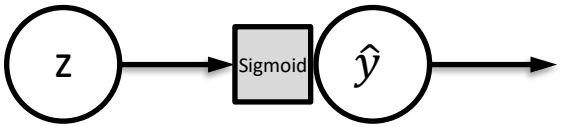
Activation Function (Sigmoid)



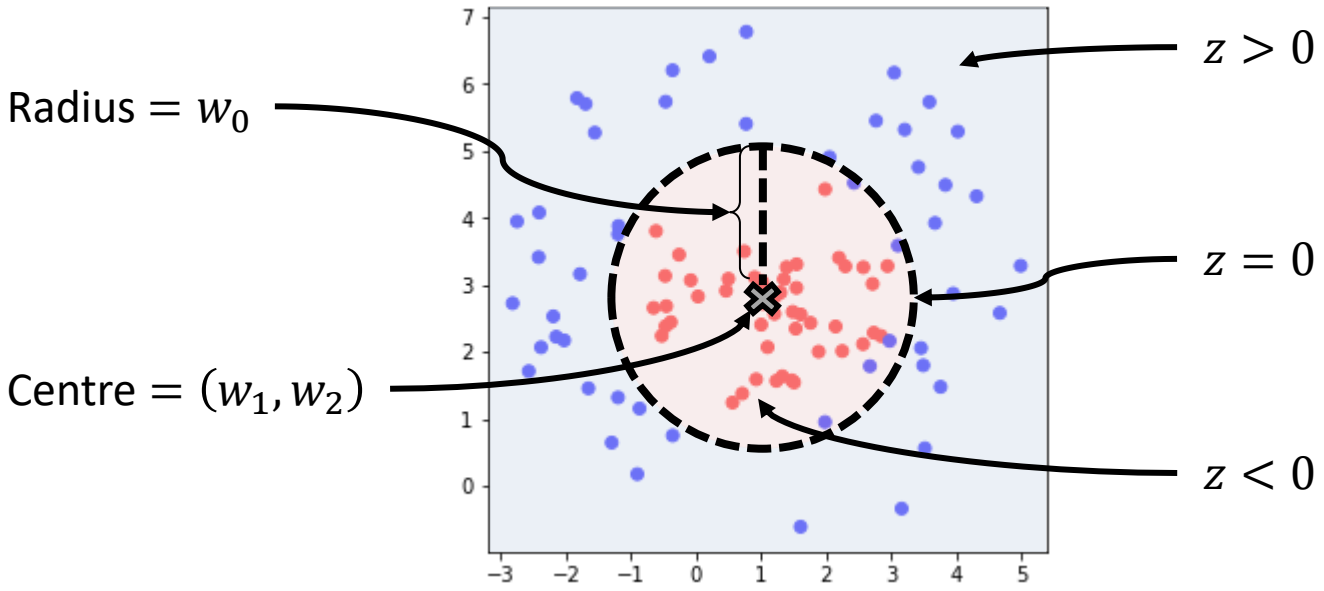
$$z = (x_1 - w_1)^2 + (x_2 - w_2)^2 - w_0^2$$

We would like wherever $z > 0$ to classify as **class 1**, and wherever $z < 0$ 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)

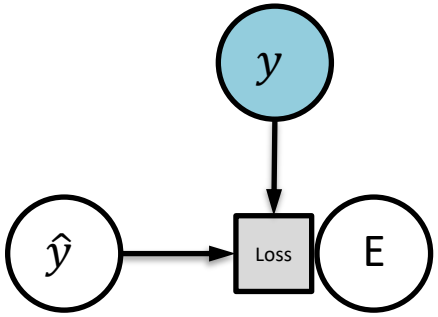


$$z = (x_1 - w_1)^2 + (x_2 - w_2)^2 - w_0^2$$

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

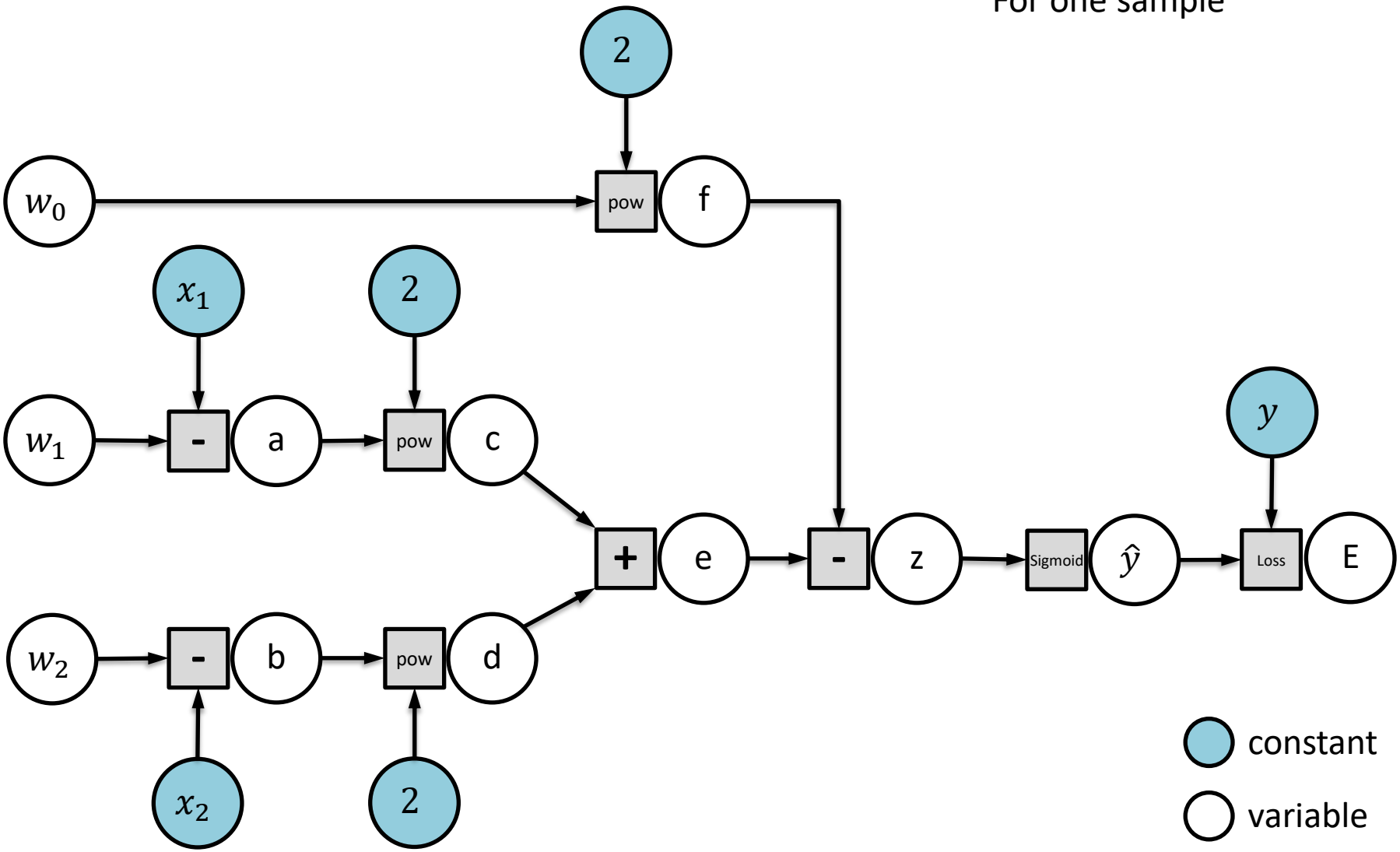
Finally, we would compare the output to the correct class using the cross-entropy loss we saw before:

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



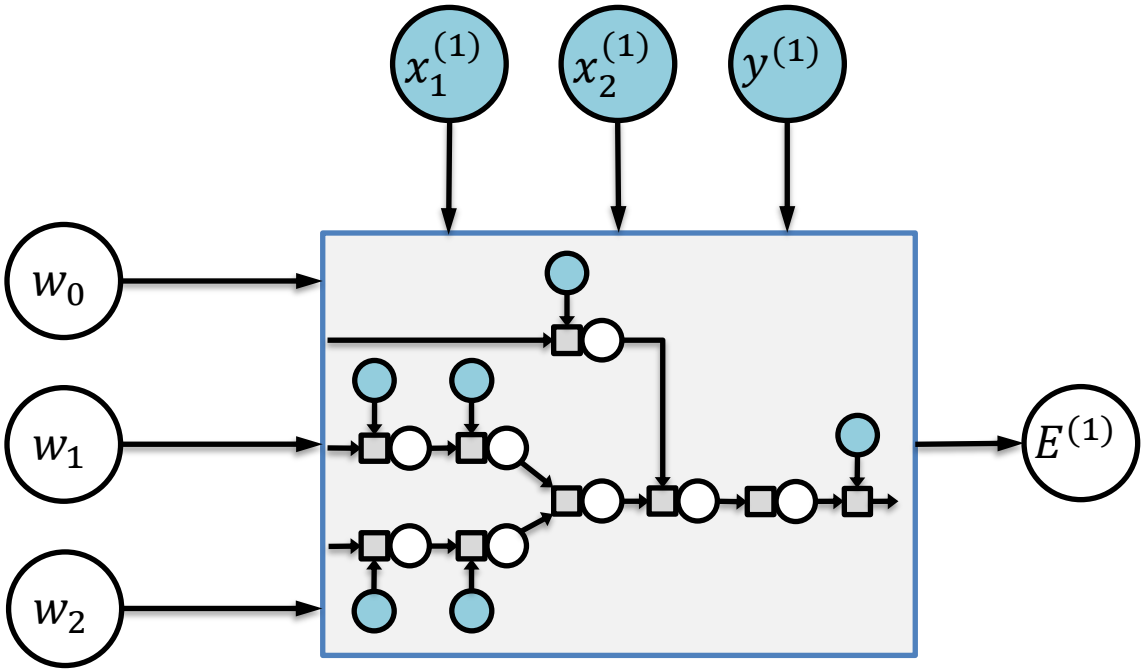
Complete Computational Graph

For one sample



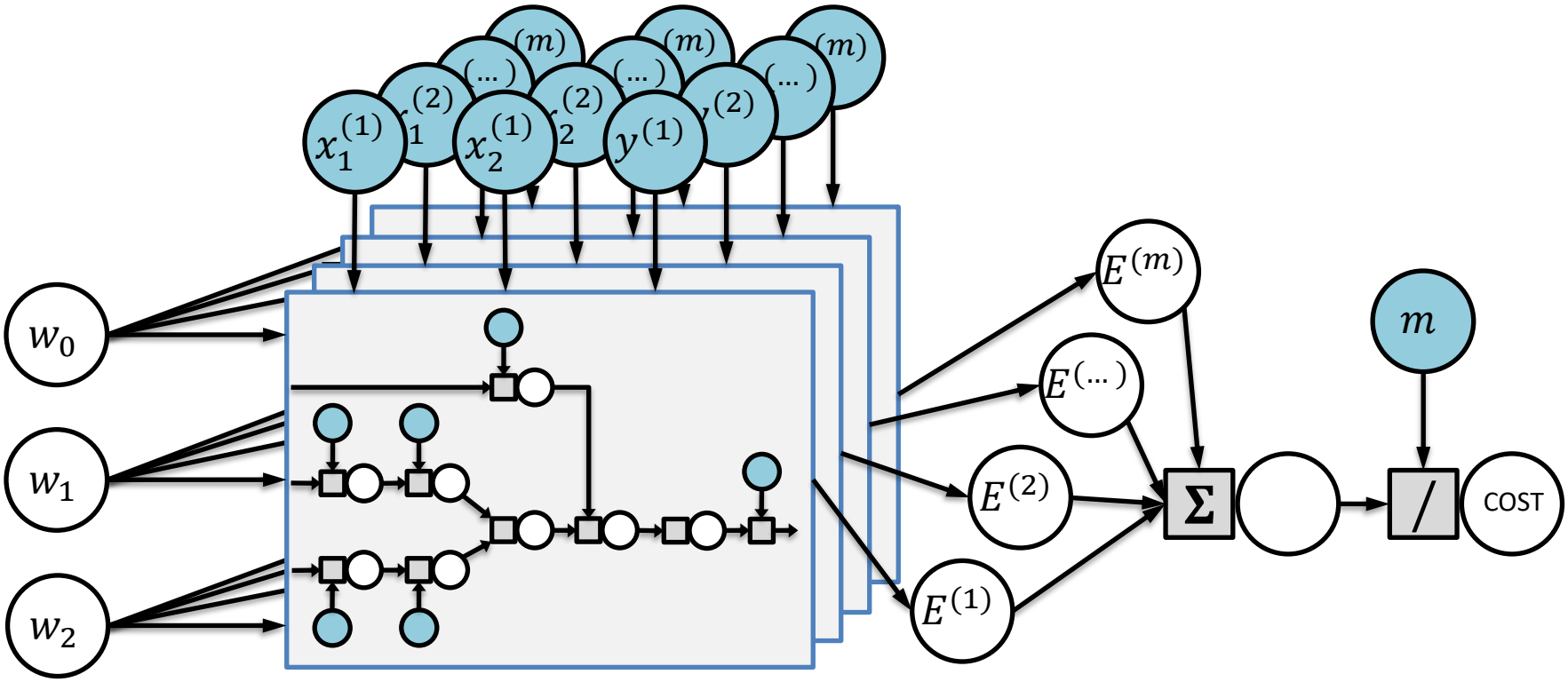
Complete Computational Graph

For one sample



Complete Computational Graph

Full batch of *m* samples



Gaining Efficiency

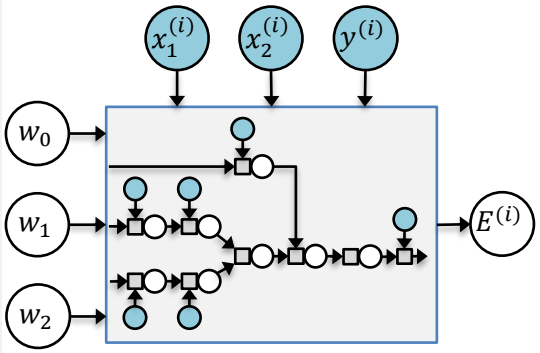
The derivative of the sum, equals the sum of derivatives… We can backpropagate errors for each sample individually, and accumulate the derivatives

Initialise gradients to zero

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

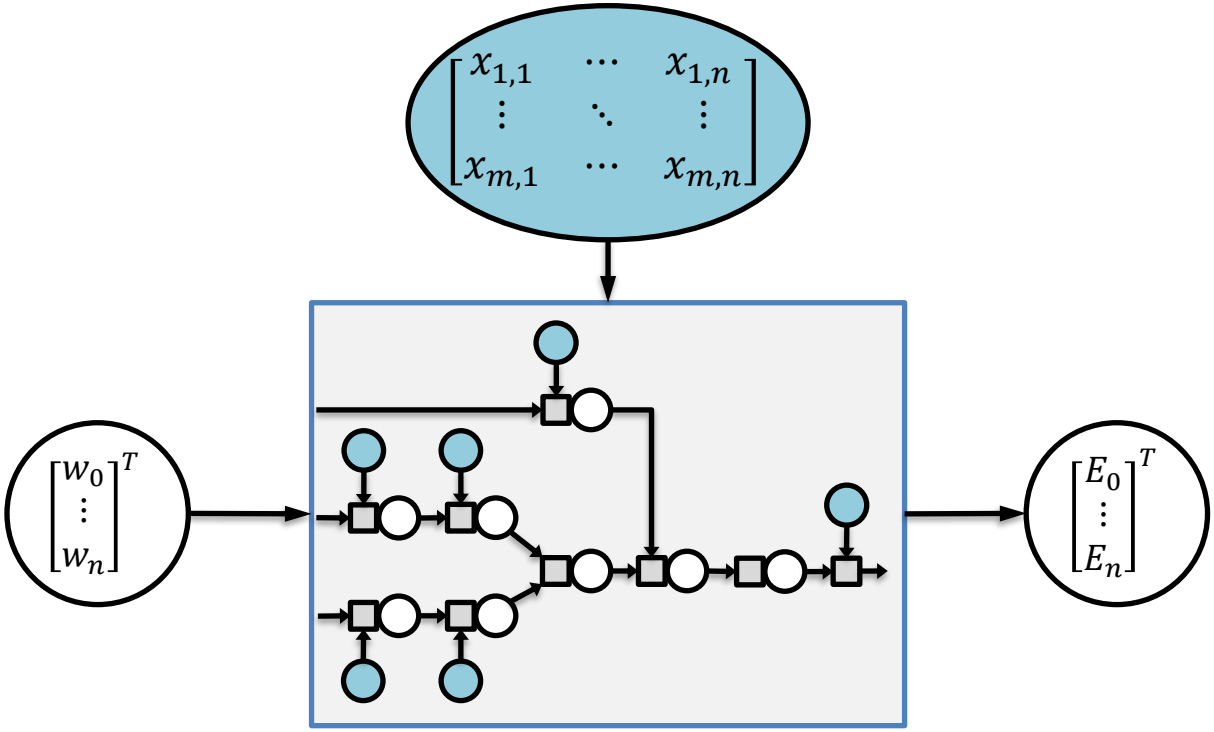
Remember to divide by the number of samples when applying gradient descent

```
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

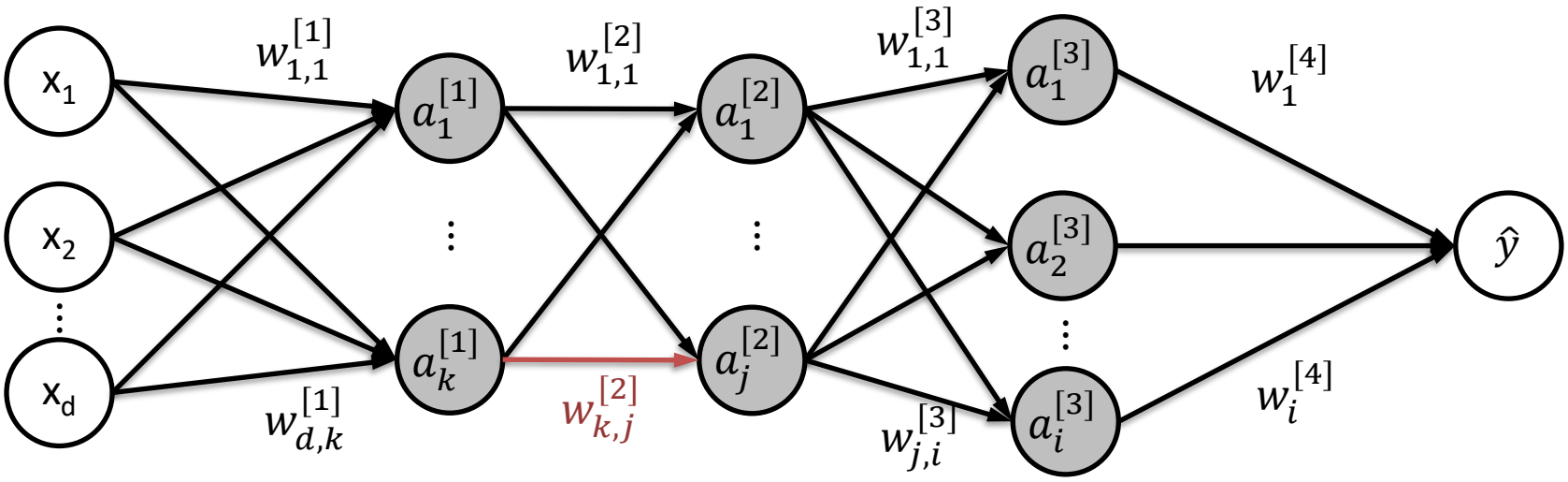
Vectorise data, and take advantage of the SIMD (Single Instruction, Multiple Data) capabilities of CPUs and GPUs



BACKPROPAGATION – TAKE TWO

Backpropagation Algorithm

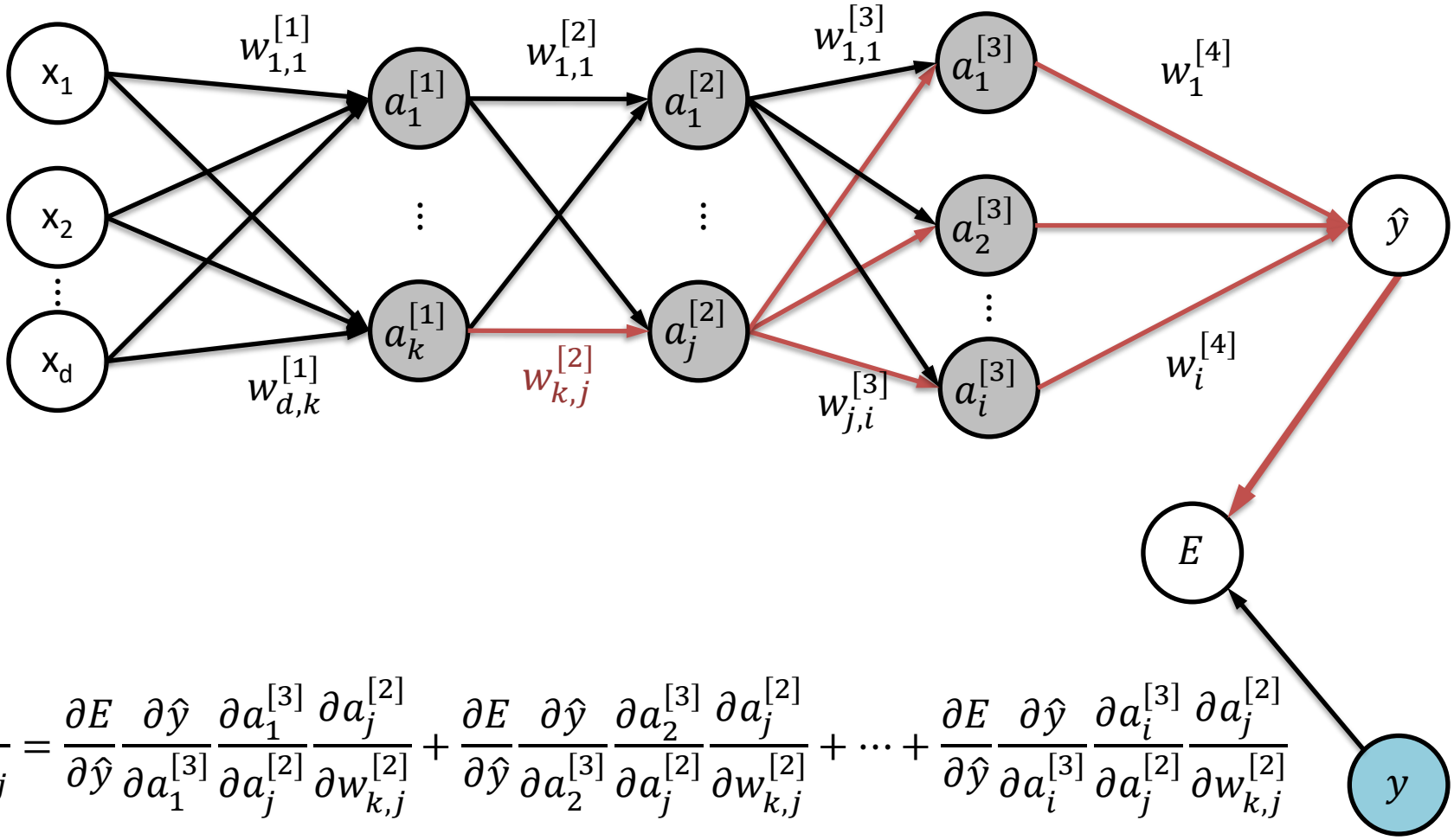
How should I change $w_{j,k}^{[2]}$?



1. Receive new observation $\mathbf{x} = [x_1, x_2, \ldots, x_d]$ 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

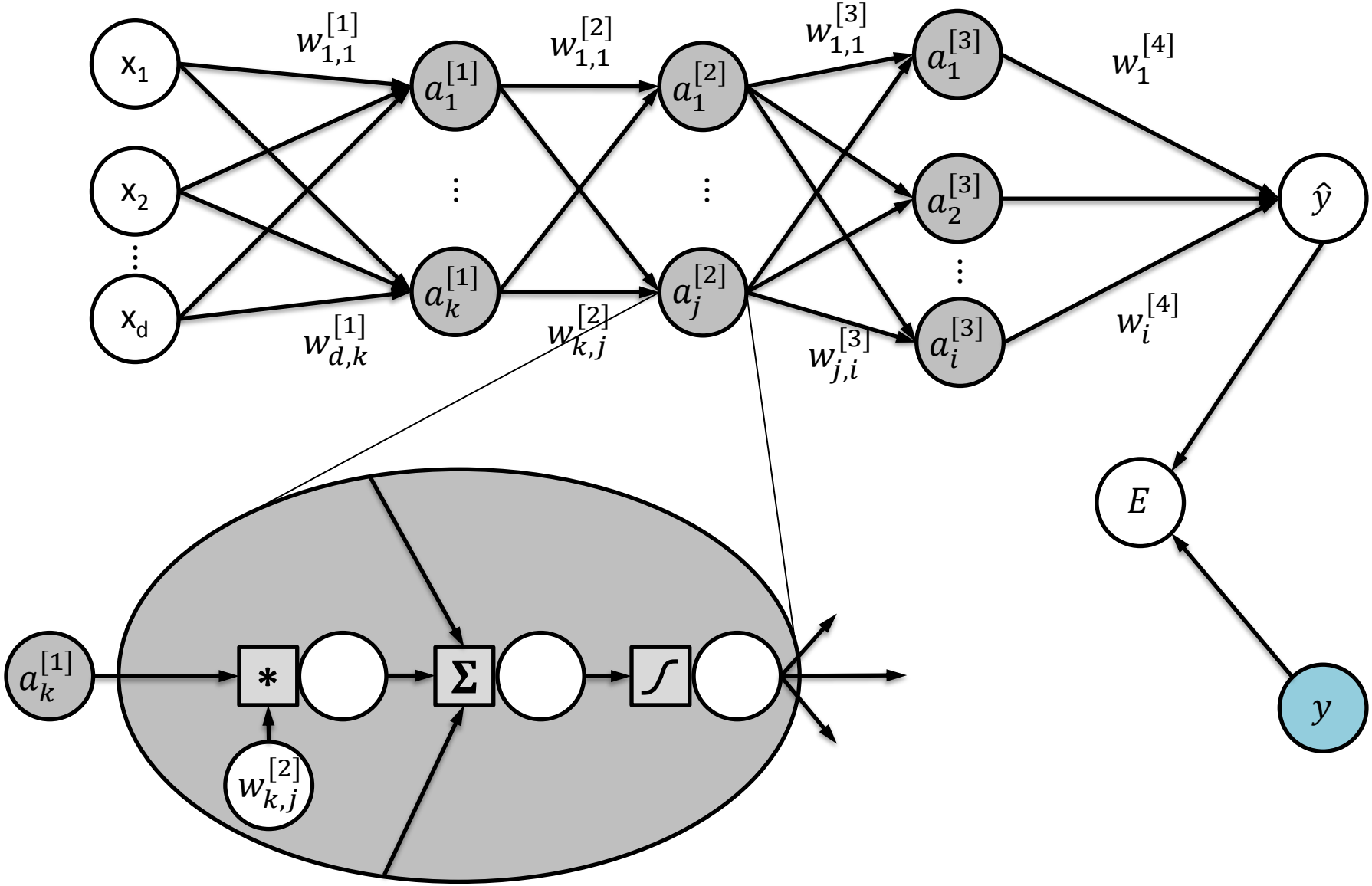
Backpropagation Algorithm

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

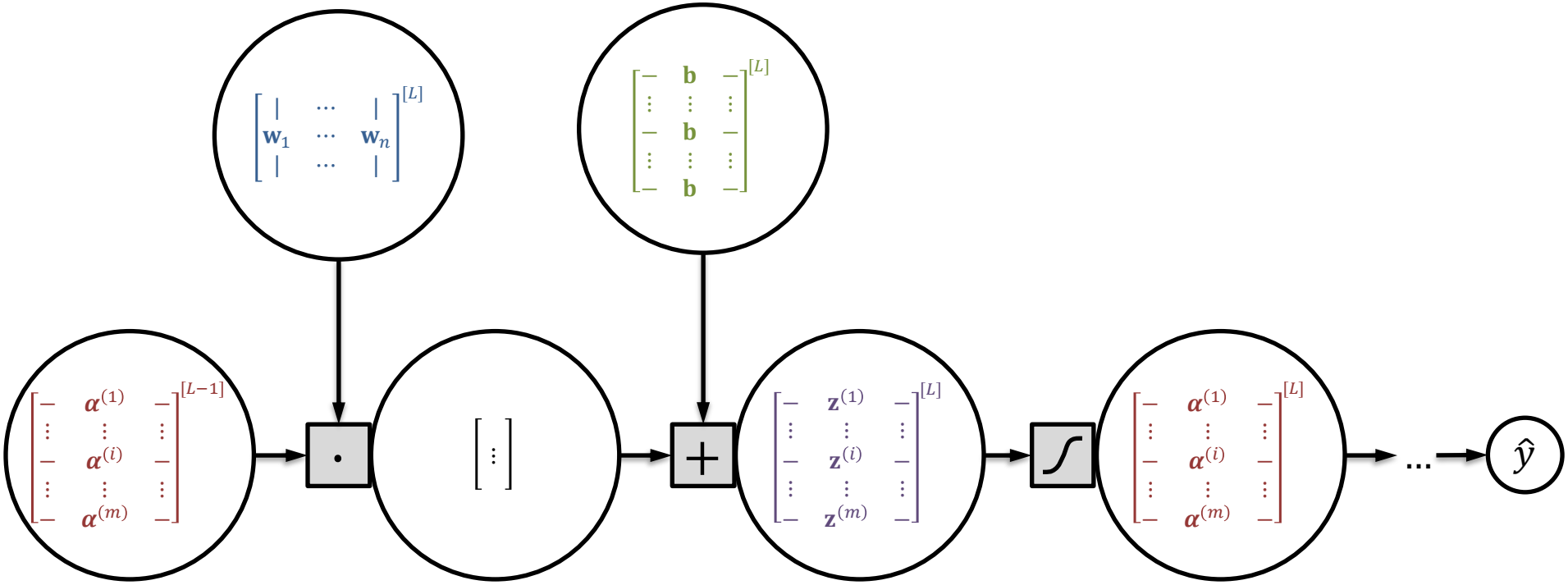


$$\frac{\partial E}{\partial w_{k,j}} = \frac{\partial E}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial a_1^{[3]}}\frac{\partial a_1^{[3]}}{\partial a_j^{[2]}}\frac{\partial a_j^{[2]}}{\partial w_{k,j}^{[2]}} + \frac{\partial E}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial a_2^{[3]}}\frac{\partial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

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 CALCULUS

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

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

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

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

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 y** with respect to **each component of 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: $\frac{\partial y_i}{\partial x_j} = w_{i,j}$

Jacobian Matrix

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

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

In this case the Jacobian matrix would be:

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

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

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

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?

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

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

$$\frac{d\mathbf{y^T}}{d\mathbf{x^T}} = \left(\frac{d\mathbf{y}}{d\mathbf{x}}\right)^{\boldsymbol{T}} = \mathbf{W^T}$$

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

Dealing with more than two dimensions

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

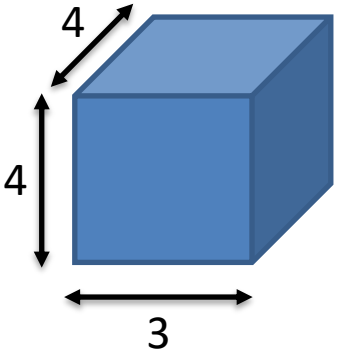
$$\boldsymbol{y} = \mathbf{W}\mathbf{x}$$

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

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

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

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



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

Dealing with more than two dimensions

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 row of $\mathbf{W}$, the elements: $w_{2,i}$

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

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

$$\mathbf{Y} = \mathbf{X}\mathbf{W}$$

If $Y_{i,:}$ is the *i*th row of $\mathbf{Y}$ and $X_{i,:}$ is the *i*th row of $\mathbf{X}$ it can be shown that:

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

The chain rule

The chain rule applies as expected here:

$$\mathbf{y} = \mathbf{VWx} \qquad \frac{d\mathbf{y}}{d\mathbf{x}} = \mathbf{VW}$$

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

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

$$\frac{d\mathbf{y}}{d\mathbf{x}} = \frac{d\mathbf{y}}{d\mathbf{z}}\frac{d\mathbf{z}}{d\mathbf{x}} = \mathbf{VW}$$

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

More info on Matrix Calculus

"*The Matrix Calculus You Need For Deep Learning*",
Terence Parr and Jeremy Howard
https://arxiv.org/pdf/1802.01528.pdf

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

Wikipedia page on Matrix Calculus
https://en.wikipedia.org/wiki/Matrix_calculus

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, to calculate derivatives for all variables in the computation graph
  - Flexible: we can define new models easily (usually transparent from the end user)
  - Vectorisable: use matrix calculus

Still Not A Learning algorithm

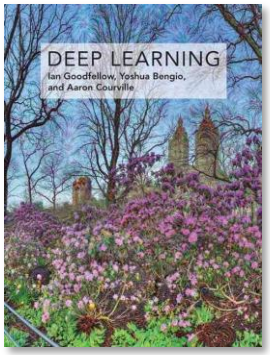
- We know how to compute error derivatives for every weight on a single training point
- We got an idea about how to extend this for a whole batch of points
- We still need to see
  - **Loss functions**: How to measure our error? This depends on the task we want to solve.
  - **Activation functions**: 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?

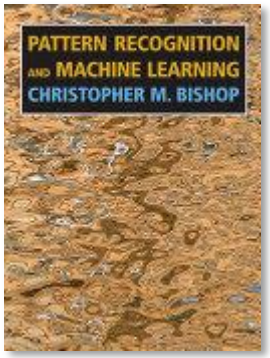
Next Week

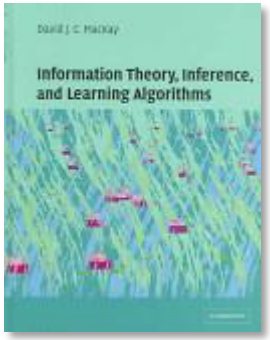
With L. Gomez

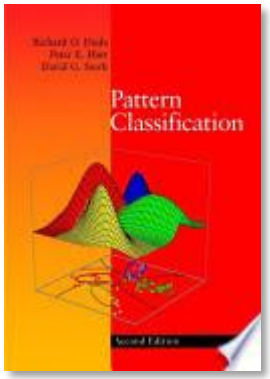
Resources (I)

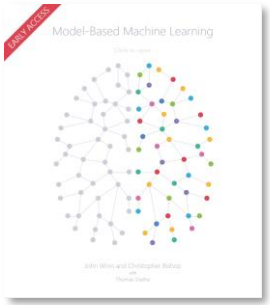
- *I. Goodfellow, Y. Bengio, A. Courville, "Deep Learning", MIT Press, 2016*
  http://www.deeplearningbook.org/
- *C. Bishop, "Pattern Recognition and Machine Learning", Springer, 2006*
  http://research.microsoft.com/en-us/um/people/cmbishop/prml/index.htm
- *D. MacKay, "Information Theory, Inference and Learning Algorithms", Cambridge University Press, 2003*
  http://www.inference.phy.cam.ac.uk/mackay/
- *R.O. Duda, P.E. Hart, D.G. Stork, "Pattern Classification", Wiley & Sons, 2000*
  http://books.google.com/books/about/Pattern_Classification.html?id=Br33IRC3PkQC
- *J. Winn, C. Bishop, "Model-Based Machine Learning", early access*
  http://mbmlbook.com/











Further Info

- Many of the slides of these lectures have been adapted from various highly recommended online lectures and courses:
  - 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, (*ex Coursera)
    https://www.youtube.com/playlist?list=PLiPvV5TNogxKKwvKb1RKwkq2hm7ZvpHz0
  - Luis Serrano's introductory videos
    https://www.youtube.com/channel/UCgBncpylJ1kiVaPyP-PZauQ
  - Michael Nielsen's *Neural Networks and Deep Learning*
    http://neuralnetworksanddeeplearning.com/