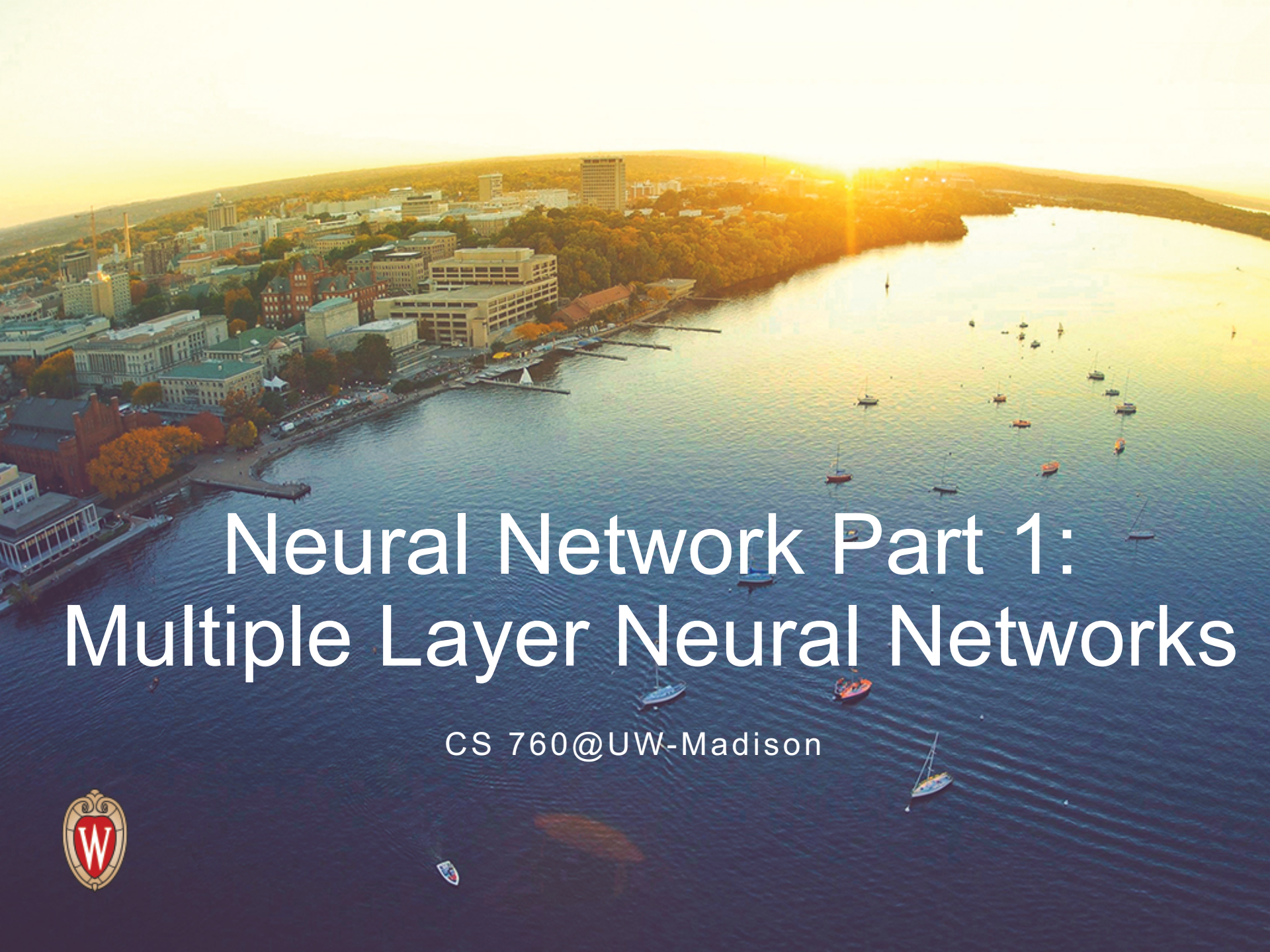


Neural Network Part 1: Multiple Layer Neural Networks

CS 760@UW-Madison

Goals for the lecture

you should understand the following concepts

- perceptrons
- the perceptron training rule
- linear separability
- multilayer neural networks
- stochastic gradient descent
- backpropagation

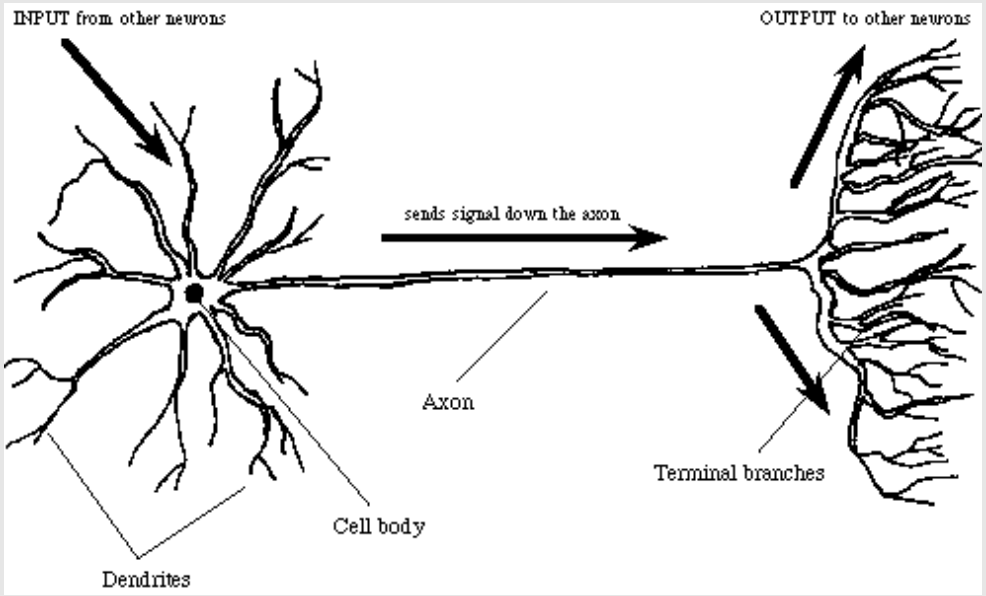
Goals for the lecture

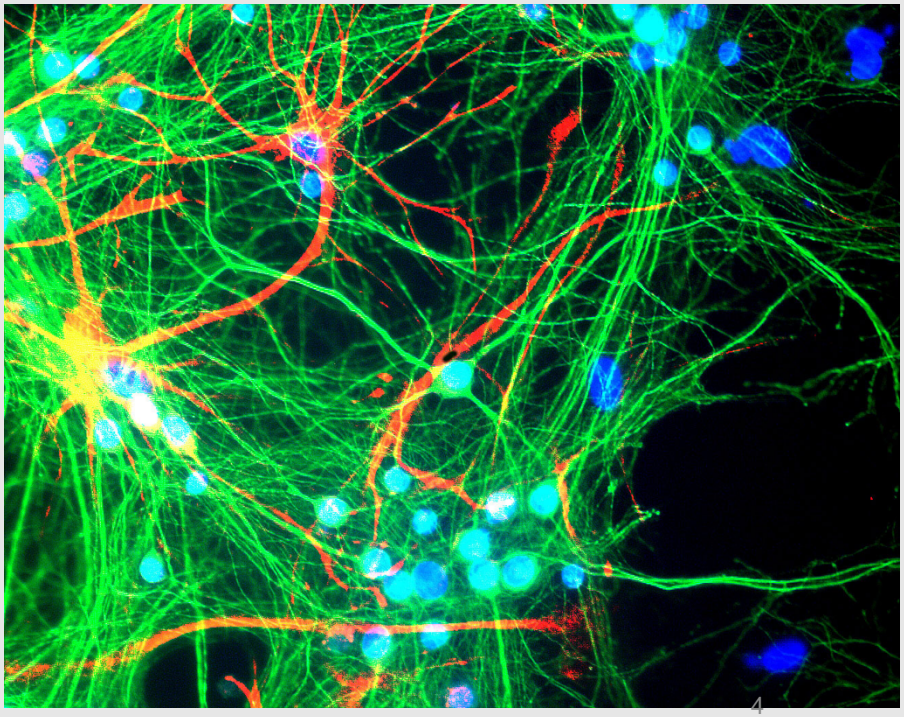
you should understand the following concepts

- regularization
- different views of regularization
- norm constraint
- dropout
- data augumentation
- early stopping

Neural networks

- a.k.a. *artificial neural networks*, *connectionist models*
- inspired by interconnected neurons in biological systems
  - simple processing units
  - each unit receives a number of real-valued inputs
  - each unit produces a single real-valued output

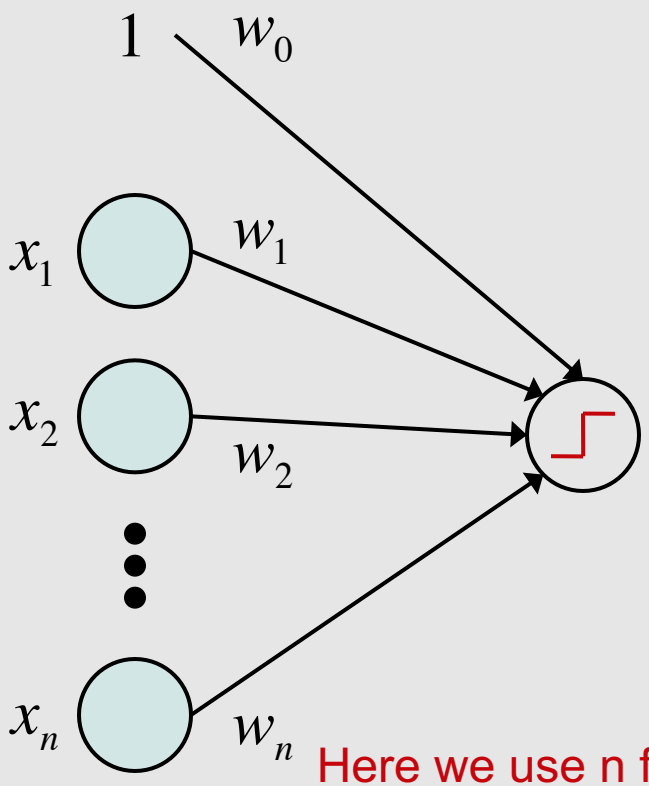





Perceptron

Perceptrons

[McCulloch & Pitts, 1943; Rosenblatt, 1959; Widrow & Hoff, 1960]



$$o = \begin{cases} 1 & \text{if } w_0 + \sum_{i=1}^{n} w_i x_i > 0 \\ 0 & \text{otherwise} \end{cases}$$

Here we use n for dimension

input units:
represent given $\boldsymbol{x}$

output unit:
represents binary classification

The perceptron training rule

1. initialize weights w=0

2. iterate through training instances until convergence

2a. calculate the output for the given instance

$$o = \begin{cases} 1 & \text{if } w_0 + \sum_{i=1}^{n} w_i x_i > 0 \\ 0 & \text{otherwise} \end{cases}$$

2b. update each weight

$$\Delta w_i = (y - o) x_i$$

$$w_i \leftarrow w_i + \Delta w_i$$

Representational power of perceptrons

perceptrons can represent only *linearly separable* concepts

$$o = \begin{cases} 1 & \text{if } w_0 + \sum_{i=1}^{n} w_i x_i > 0 \\ 0 & \text{otherwise} \end{cases}$$

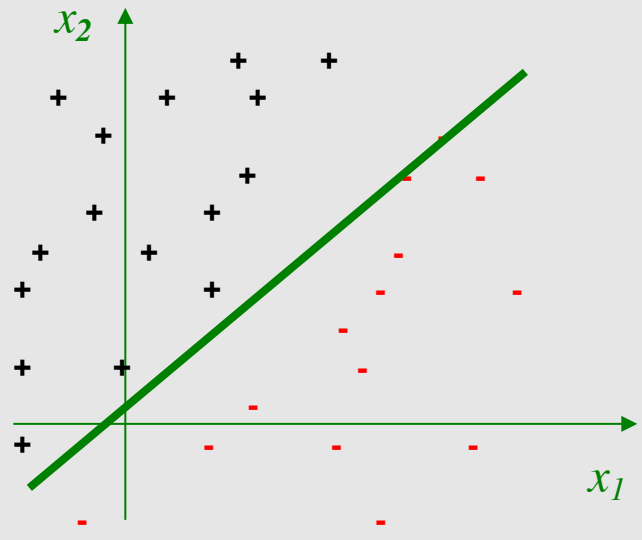
decision boundary given by:

$$1 \text{ if } w_0 + w_1 x_1 + w_2 x_2 > 0$$

also write as: **wx** > 0

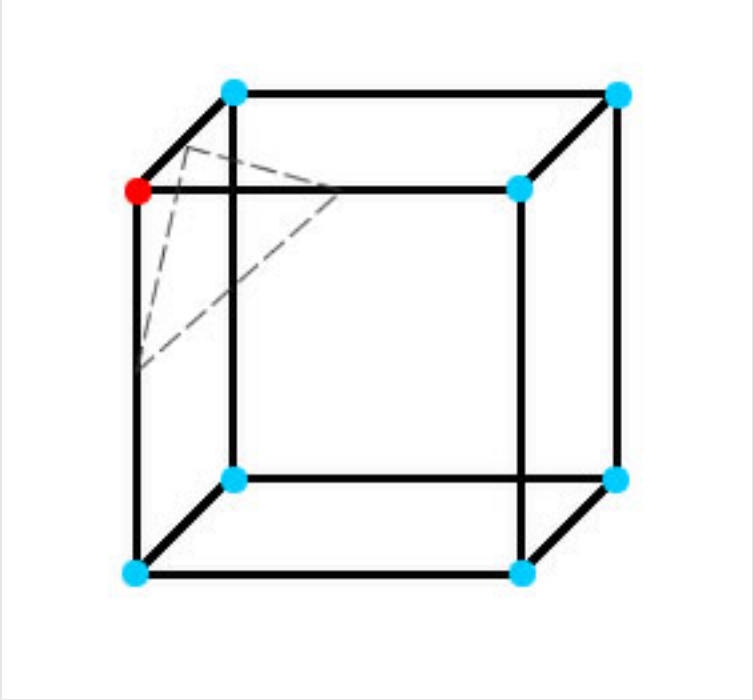
$$w_1 x_1 + w_2 x_2 = -w_0$$

$$x_2 = -\frac{w_1}{w_2} x_1 - \frac{w_0}{w_2}$$



Representational power of perceptrons

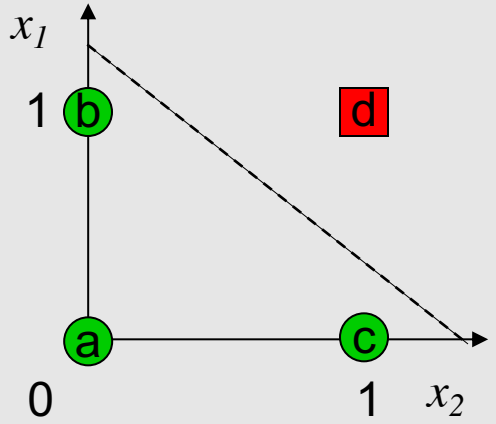
- in previous example, feature space was 2D so decision boundary was a line
- in higher dimensions, decision boundary is a hyperplane



Some linearly separable functions

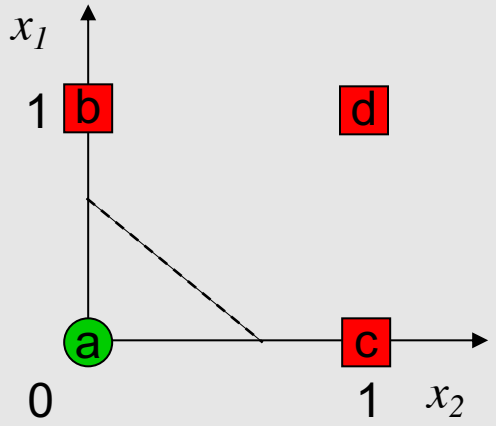
AND

	x_1	x_2	y
a	0	0	0
b	0	1	0
c	1	0	0
d	1	1	1



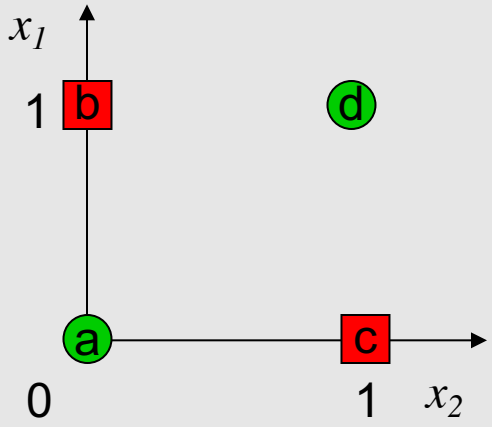
OR

	x_1	x_2	y
a	0	0	0
b	0	1	1
c	1	0	1
d	1	1	1

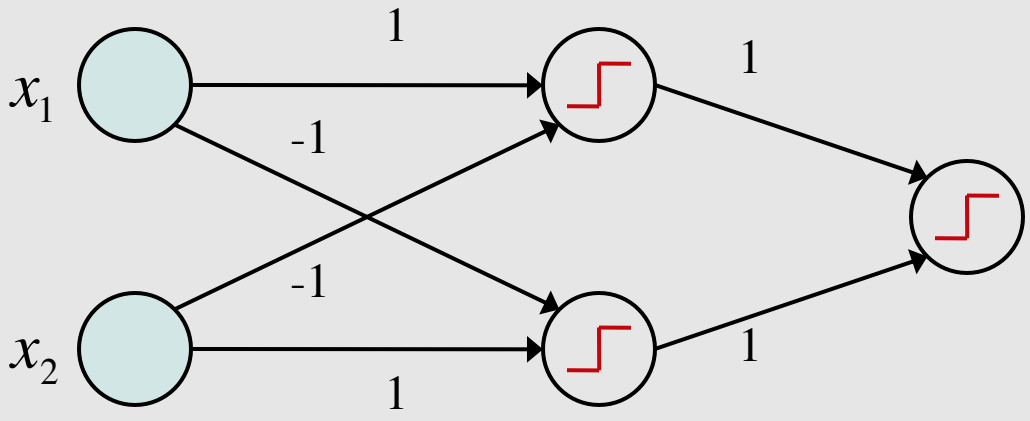


XOR is not linearly separable

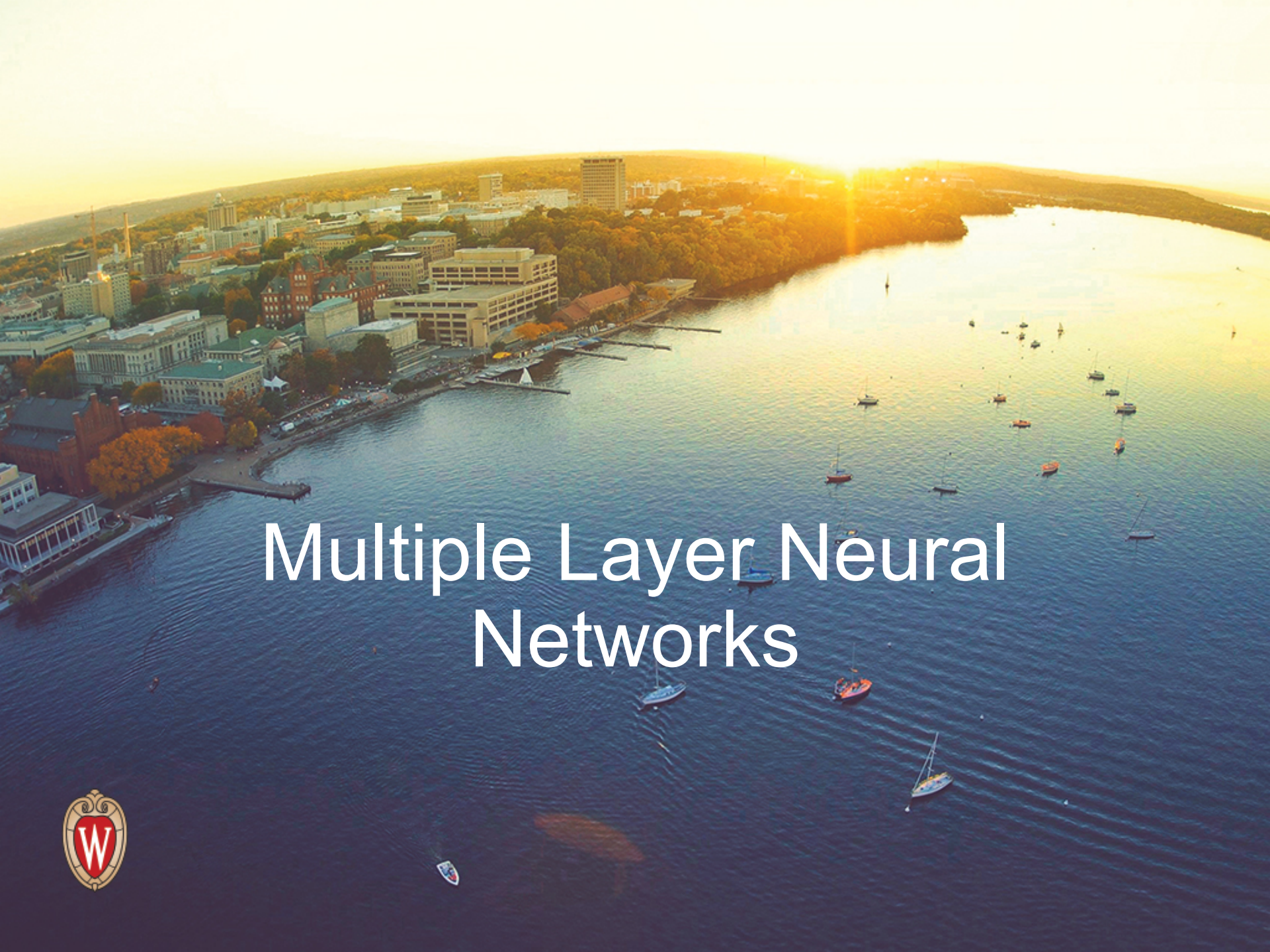
	x_1	x_2	y
a	0	0	0
b	0	1	1
c	1	0	1
d	1	1	0



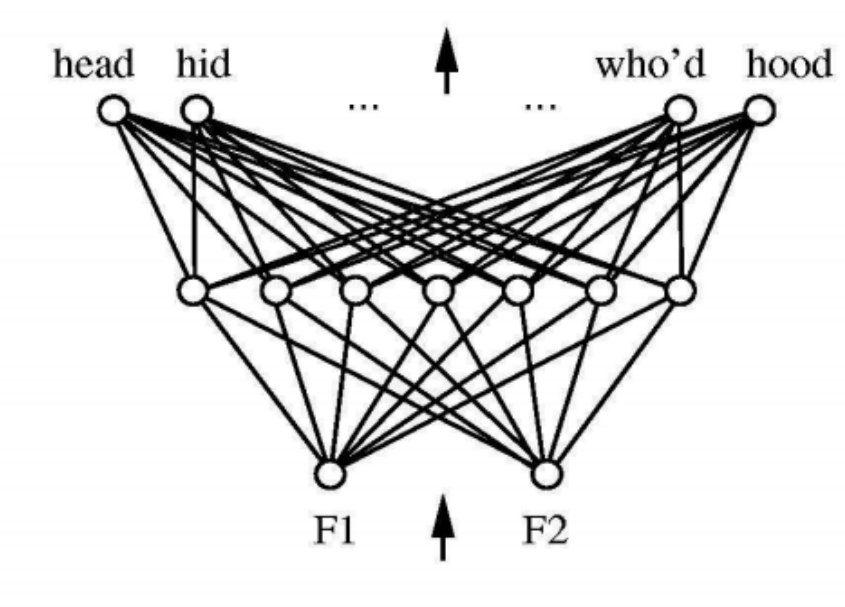
a multilayer perceptron can represent XOR



assume $w_0 = 0$ for all nodes


Multiple Layer Neural Networks

Example multilayer neural network



output units

hidden units

input units

figure from Huang & Lippmann, *NIPS* 1988

input: two features from spectral analysis of a spoken sound

output: vowel sound occurring in the context "h__d"

Decision regions of a multilayer neural network

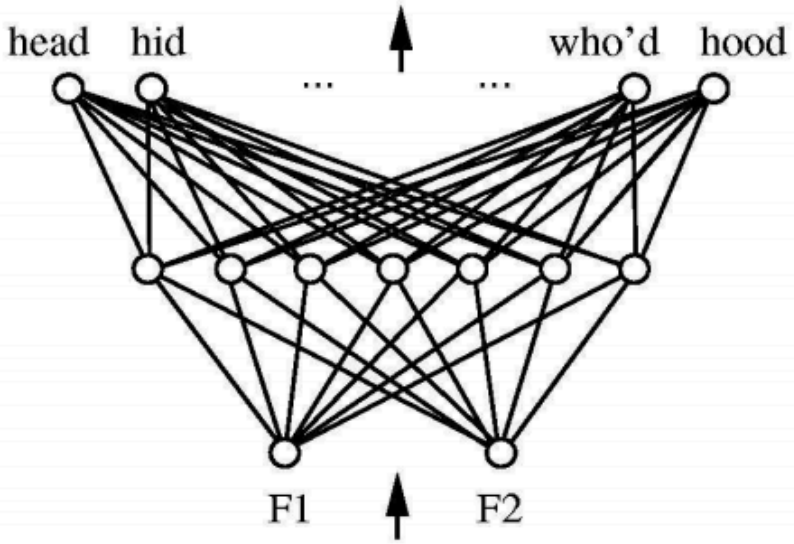
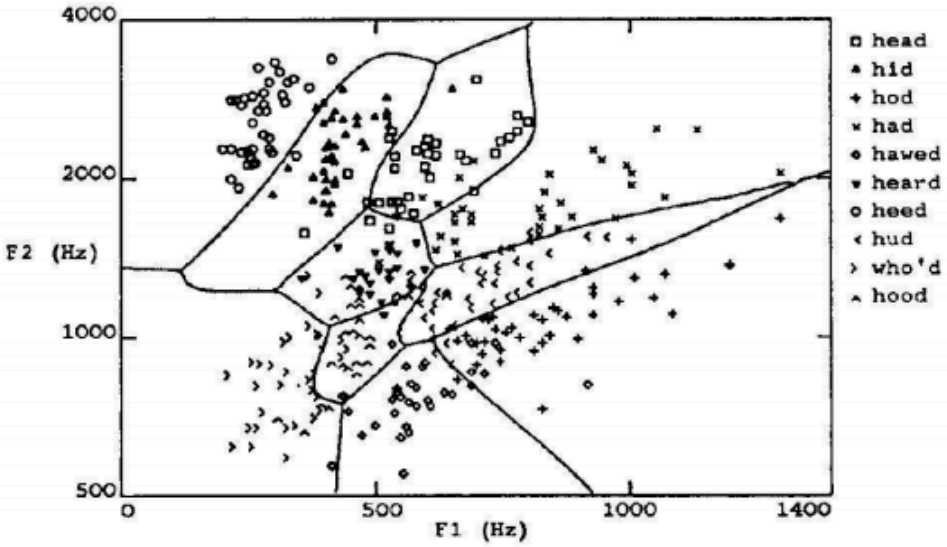


figure from Huang & Lippmann, *NIPS* 1988



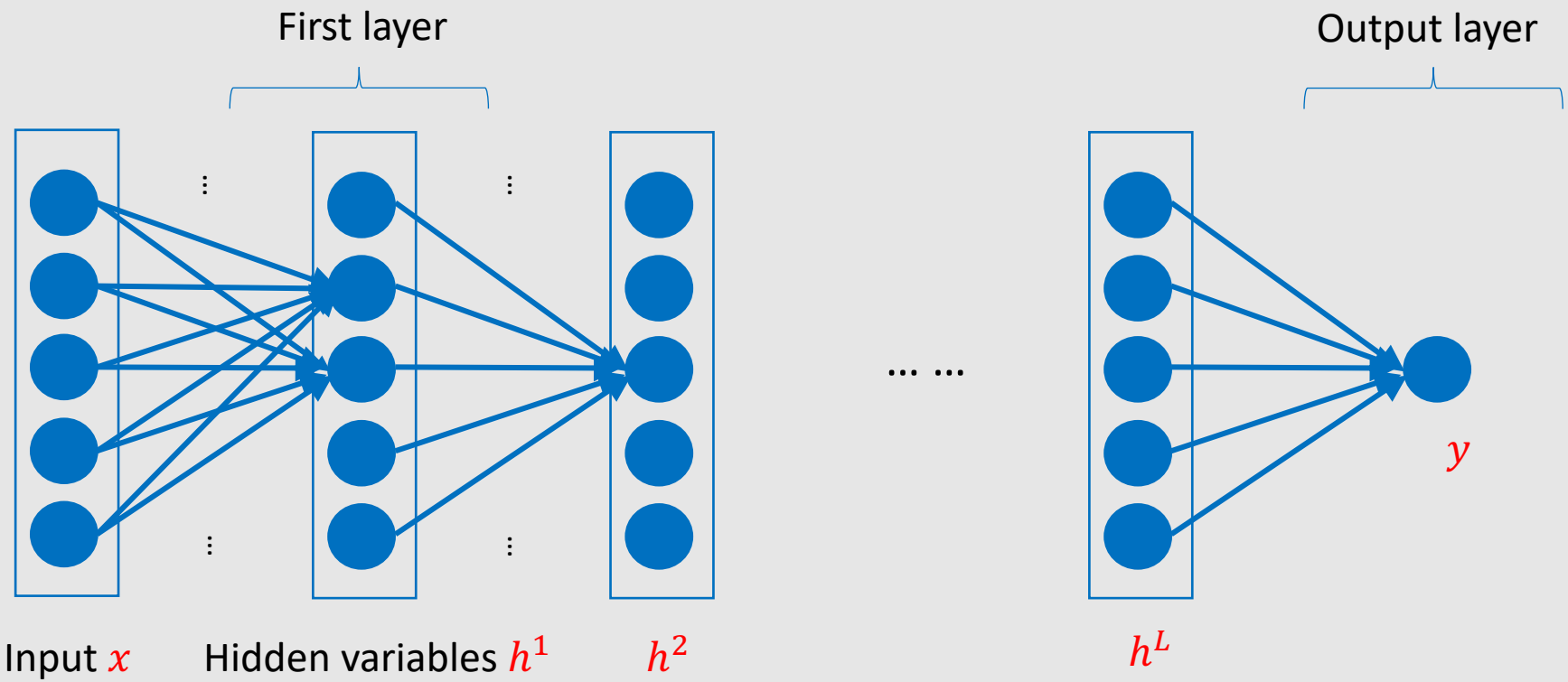
input: two features from spectral analysis of a spoken sound

output: vowel sound occurring in the context "h__d"

Components

- Representations:
  - Input
  - Hidden variables
- Layers/weights:
  - Hidden layers
  - Output layer

Components



Input

- Represented as a vector

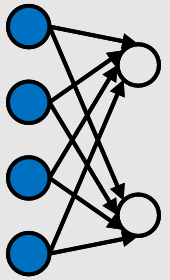
- Sometimes require some preprocessing, e.g.,
  - Subtract mean
  - Normalize to [-1,1]

Expand

Input (feature) encoding for neural networks

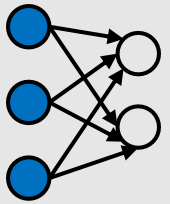
nominal features are usually represented using a *1-of-k* encoding

$$A = \begin{bmatrix}1\\0\\0\\0\end{bmatrix} \quad C = \begin{bmatrix}0\\1\\0\\0\end{bmatrix} \quad G = \begin{bmatrix}0\\0\\1\\0\end{bmatrix} \quad T = \begin{bmatrix}0\\0\\0\\1\end{bmatrix}$$



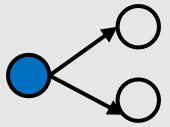
ordinal features can be represented using a *thermometer* encoding

$$\text{small} = \begin{bmatrix}1\\0\\0\end{bmatrix} \quad \text{medium} = \begin{bmatrix}1\\1\\0\end{bmatrix} \quad \text{large} = \begin{bmatrix}1\\1\\1\end{bmatrix}$$



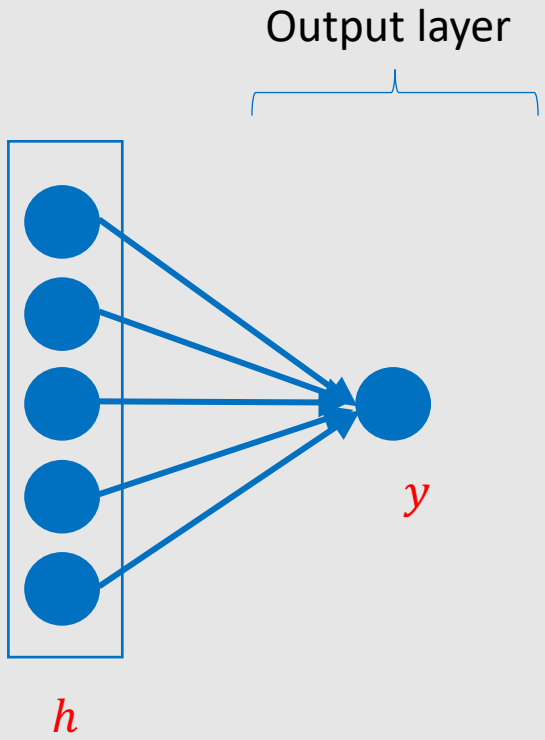
real-valued features can be represented using individual input units (we may want to scale/normalize them first though)

$$\text{precipitation} = \begin{bmatrix}0.68\end{bmatrix}$$



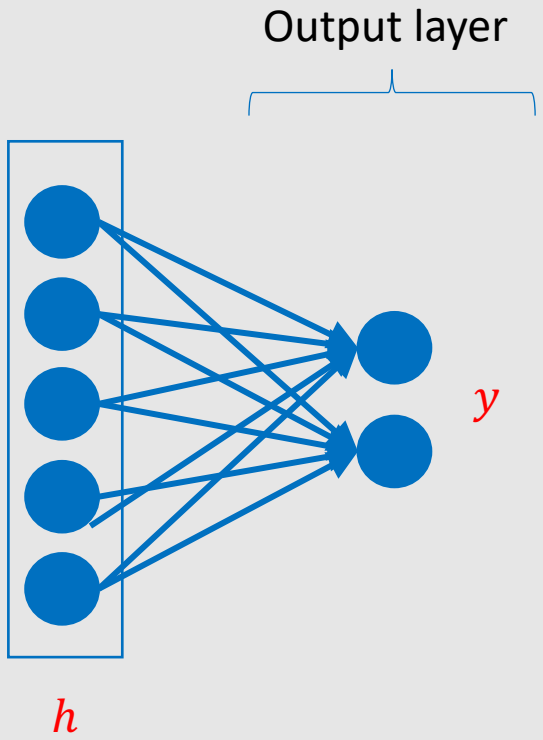
Output layers

- Regression: $y = w^T h + b$
- Linear units: no nonlinearity



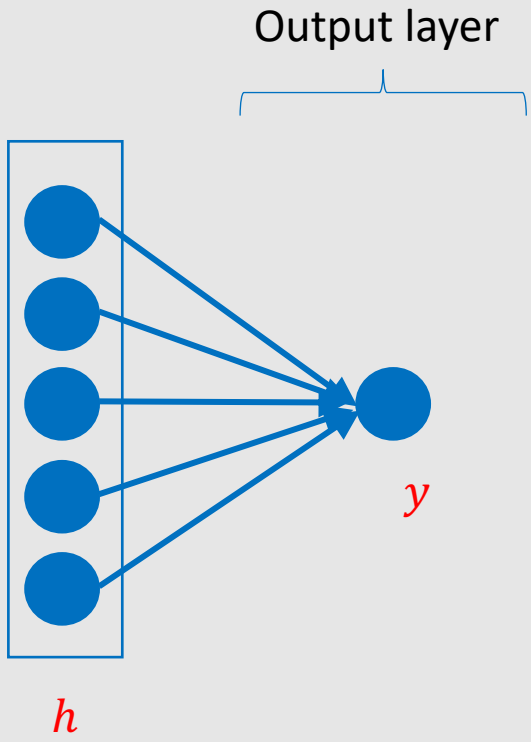
Output layers

- Multi-dimensional regression: $y = W^T h + b$
- Linear units: no nonlinearity



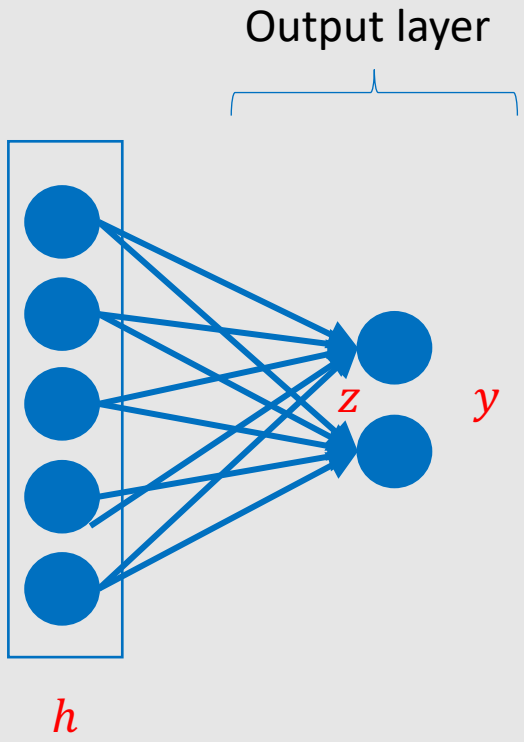
Output layers

- Binary classification: $\mathrm{P}(y = 1|h) = \sigma(w^T h + b)$
- Corresponds to using logistic regression on h



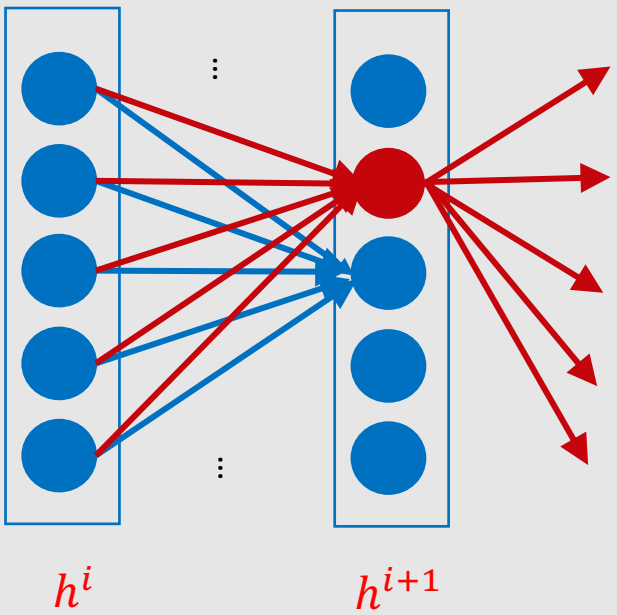
Output layers

- Multi-class classification:
- $\mathrm{P}(y|h) = \mathrm{softmax}(z)$ where $z = W^T h + b$
- Corresponds to using multi-class logistic regression on h



Hidden layers

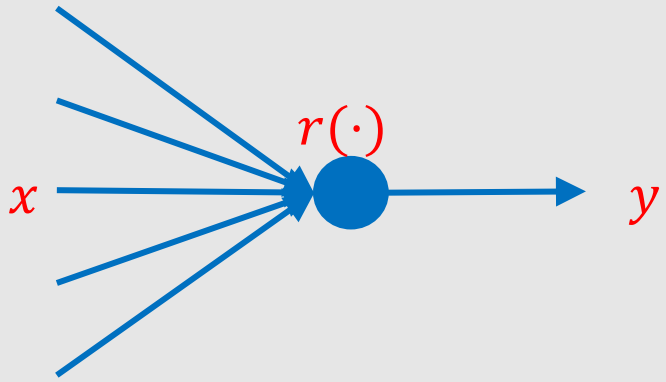
- Neuron takes weighted linear combination of the previous layer
- outputs one value for the next layer



Hidden layers

- $y = r(w^T x + b)$

- Typical activation function r
  - Threshold $\mathrm{t}(z) = \mathbb{I}[z \geq 0]$
  - Sigmoid $\sigma(z) = 1/(1 + \exp(-z))$
  - Tanh $\tanh(z) = 2\sigma(2z) - 1$



Hidden layers

- Problem: saturation

Too small gradient

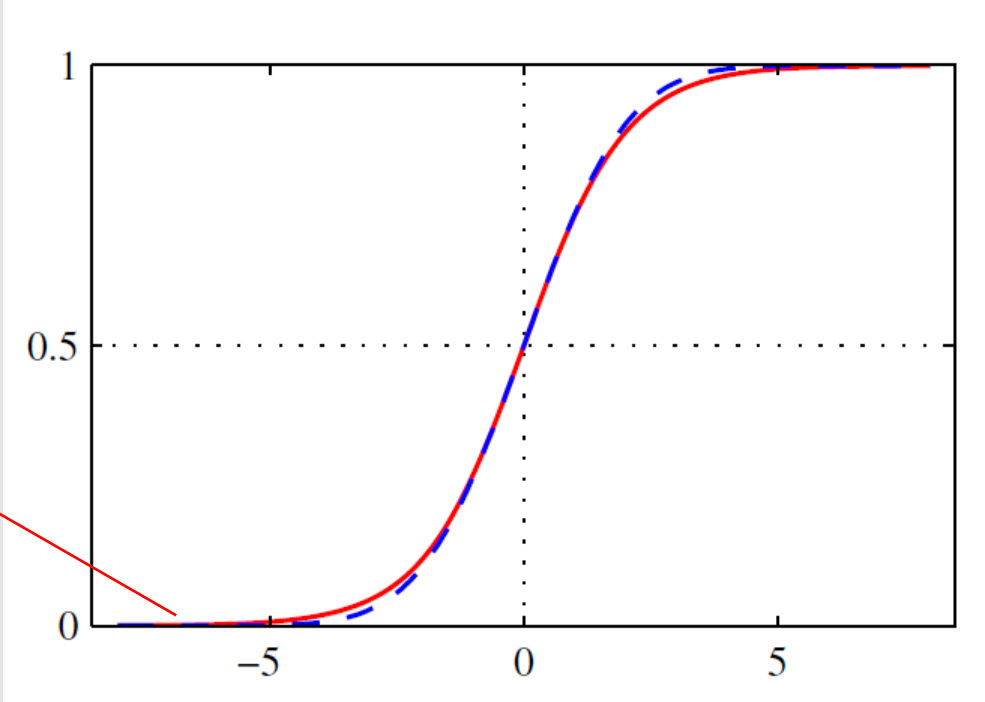


Figure borrowed from *Pattern Recognition and Machine Learning*, Bishop

Hidden layers

- Activation function ReLU (rectified linear unit)
  - $\mathrm{ReLU}(z) = \max\{z, 0\}$

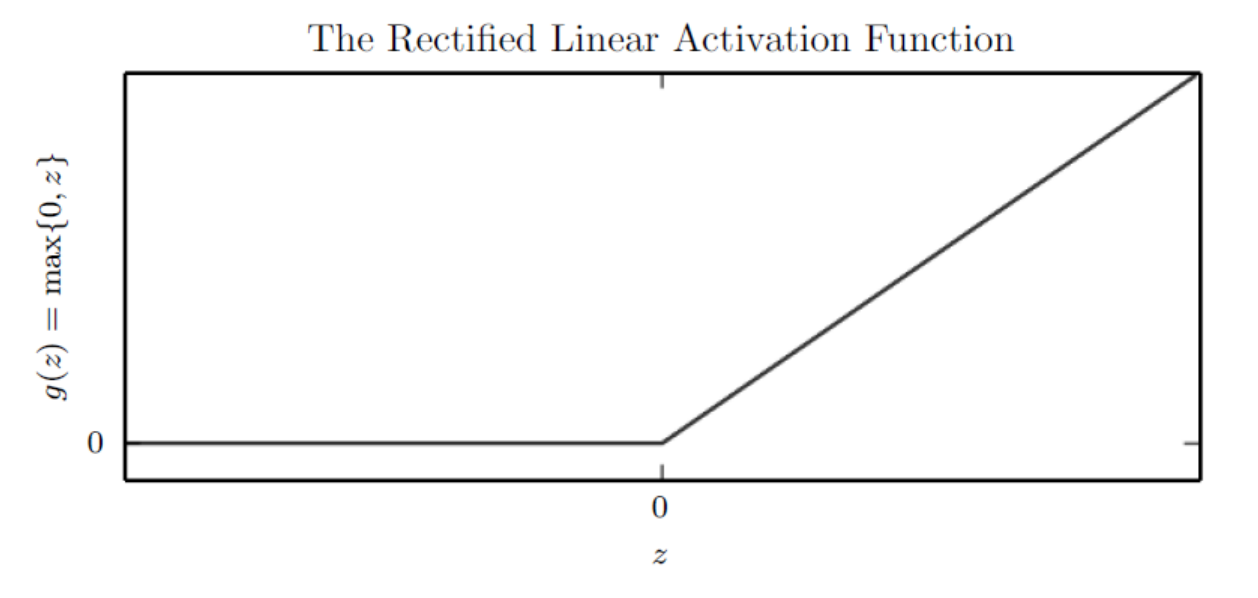


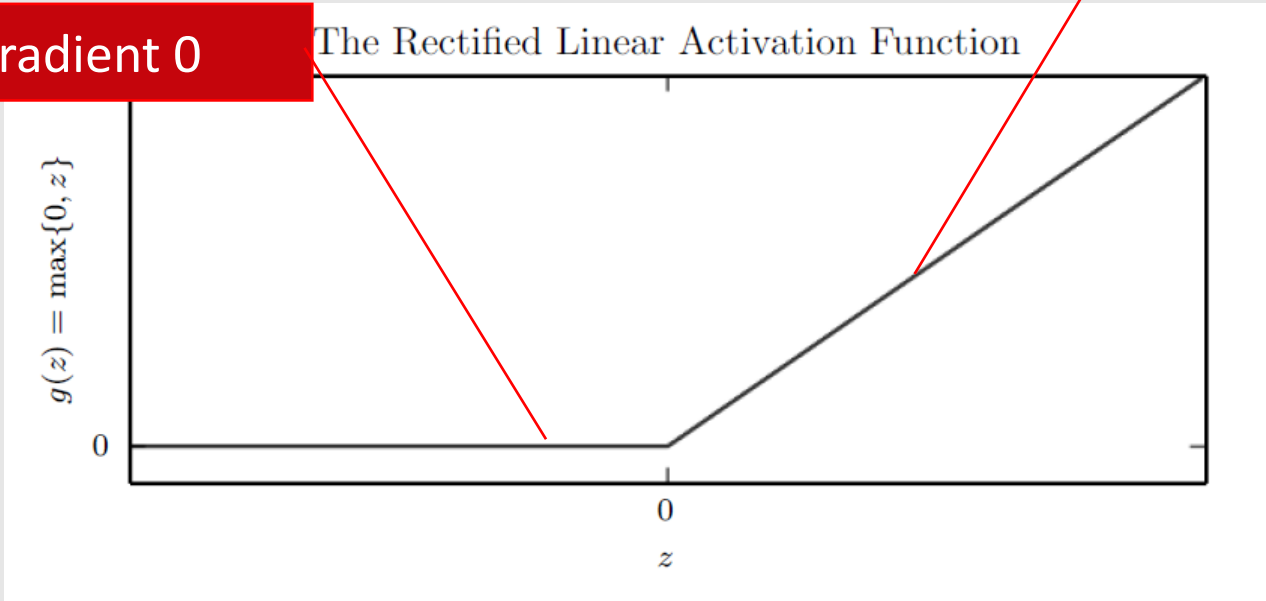
Figure from *Deep learning*, by Goodfellow, Bengio, Courville.

Hidden layers

- Activation function ReLU (rectified linear unit)
  - $\mathrm{ReLU}(z) = \max\{z, 0\}$

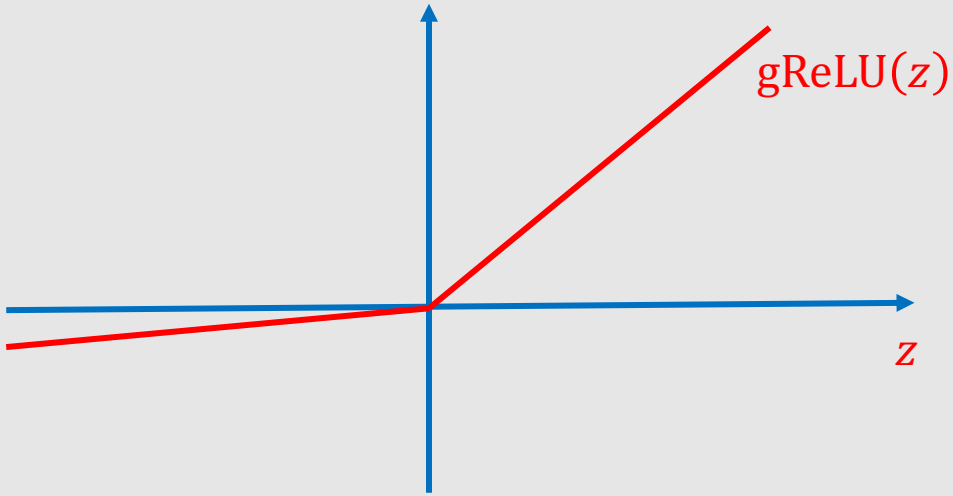
Gradient 1

Gradient 0



Hidden layers

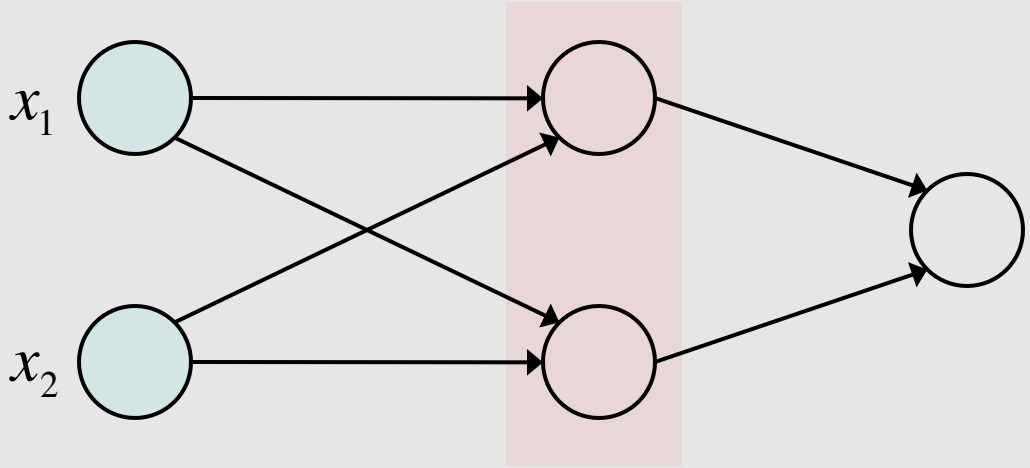
- Generalizations of ReLU $\mathrm{gReLU}(z) = \max\{z, 0\} + \alpha \min\{z, 0\}$
  - Leaky-$\mathrm{ReLU}(z) = \max\{z, 0\} + 0.01 \min\{z, 0\}$
  - Parametric-$\mathrm{ReLU}(z)$: α learnable




Backpropagation

Learning in multilayer networks

- work on neural nets fizzled in the 1960's
  - single layer networks had representational limitations (linear separability)
  - no effective methods for training multilayer networks



how to determine error signal for *hidden units?*

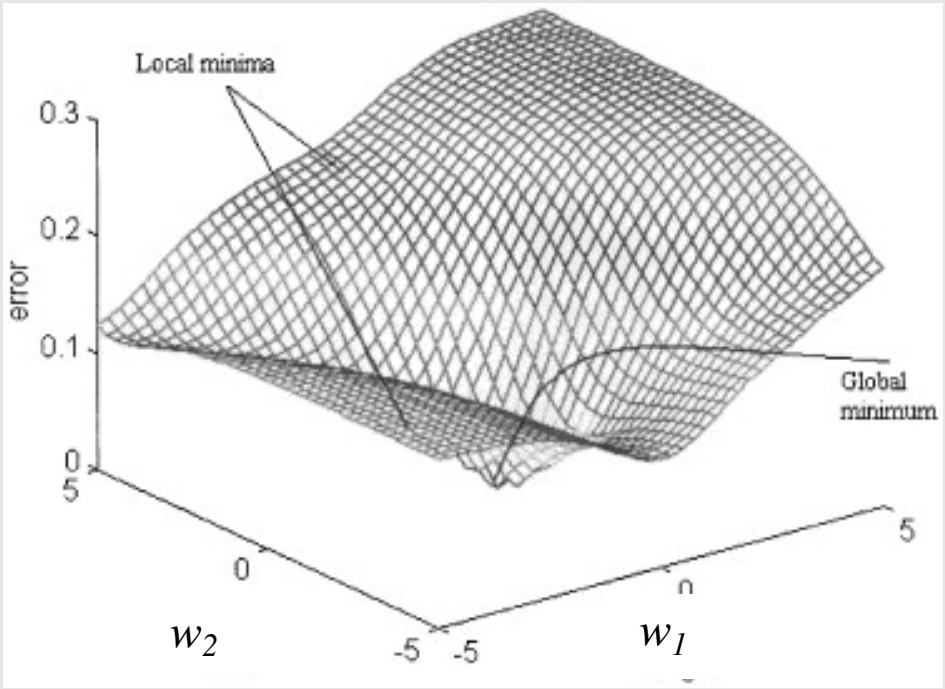
- revived again with the invention of *backpropagation* method [Rumelhart & McClelland, 1986; also Werbos, 1975]
  - key insight: require neural network to be differentiable; use *gradient descent*

Learning in multilayer networks

- learning techniques nowadays
  - random initialization of the weights
  - stochastic gradient descent (can add momentum)
  - regularization techniques
    - norm constraint
    - dropout
    - batch normalization
    - data augmentation
    - early stopping
    - pretraining
    - …

Gradient descent in weight space

Given a training set $D = \{(x^{(1)}, y^{(1)}),\ldots,(x^{(m)}, y^{(m)})\}$ we can specify an error measure that is a function of our weight vector $\boldsymbol{w}$



$$E(\boldsymbol{w}) = \frac{1}{2}\sum_{d\in \mathrm{D}}\left(y^{(d)} - o^{(d)}\right)^2$$

figure from Cho & Chow, *Neurocomputing* 1999

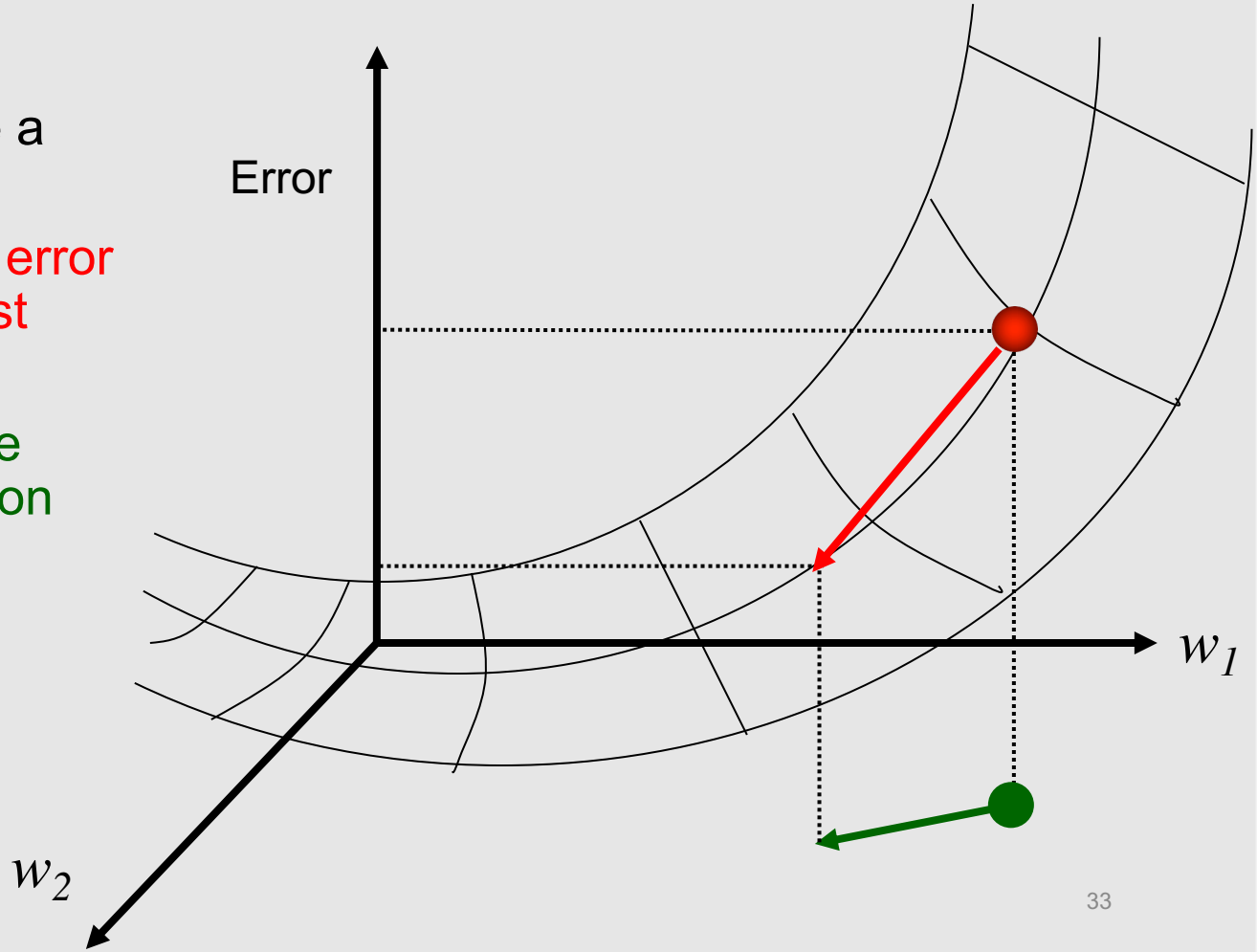
This error measure defines a surface over the hypothesis (i.e. weight) space

Gradient descent in weight space

gradient descent is an iterative process aimed at finding a minimum in the error surface

on each iteration

- current weights define a point in this space
- find direction in which error surface descends most steeply
- take a step (i.e. update weights) in that direction



Gradient descent in weight space

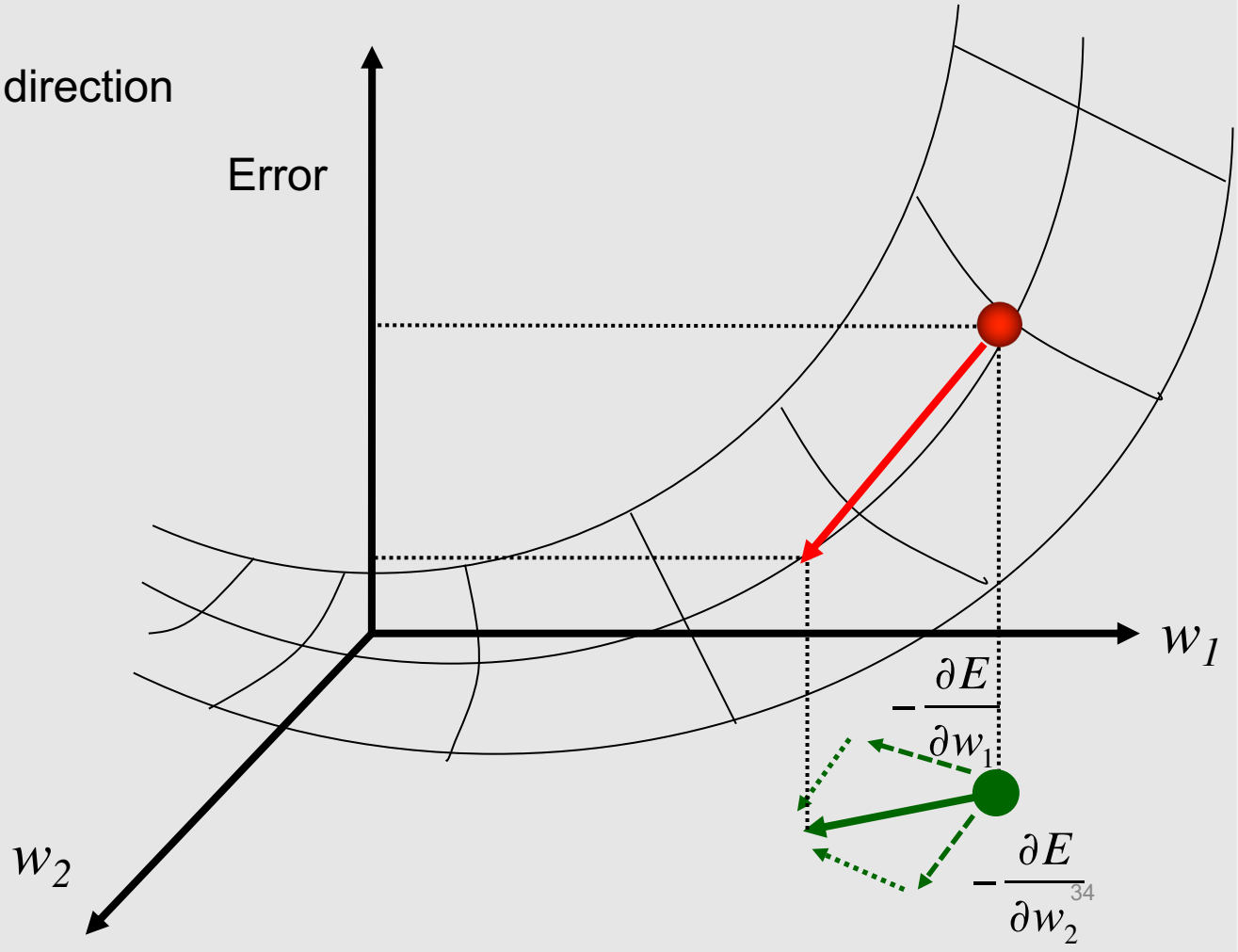
calculate the gradient of E:

$$\nabla E(\boldsymbol{w}) = \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \cdots, \frac{\partial E}{\partial w_n} \right]$$

take a step in the opposite direction

$$\Delta \boldsymbol{w} = -\eta \, \nabla E(\boldsymbol{w})$$

$$\Delta w_i = -\eta \, \frac{\partial E}{\partial w_i}$$



Batch neural network training

given: network structure and a training set $D = \{(x^{(1)}, y^{(1)}),...,(x^{(m)}, y^{(m)})\}$

initialize all weights in $\boldsymbol{w}$ to small random numbers

until stopping criteria met do

initialize the error $E(\boldsymbol{w}) = 0$

for each $(\boldsymbol{x}^{(d)}, y^{(d)})$ in the training set

input $\boldsymbol{x}^{(d)}$ to the network and compute output $o^{(d)}$

increment the error $E(\boldsymbol{w}) = E(\boldsymbol{w}) + \frac{1}{2}\left(y^{(d)} - o^{(d)}\right)^2$

calculate the gradient

$$\nabla E(\boldsymbol{w}) = \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \cdots, \frac{\partial E}{\partial w_n}\right]$$

update the weights

$$\Delta \boldsymbol{w} = -\eta \nabla E(\boldsymbol{w})$$

Online vs. batch training

- Standard gradient descent (batch training): calculates error gradient for the entire training set, before taking a step in weight space

- *Stochastic gradient descent* (online training): calculates error gradient for a single instance, then takes a step in weight space
  - much faster convergence
  - less susceptible to local minima

Online neural network training (stochastic gradient descent)

given: network structure and a training set $D = \{(x^{(1)}, y^{(1)}),...,(x^{(m)}, y^{(m)})\}$

initialize all weights in $\boldsymbol{w}$ to small random numbers

until stopping criteria met do

for each $(\boldsymbol{x}^{(d)}, y^{(d)})$ in the training set

input $\boldsymbol{x}^{(d)}$ to the network and compute output $o^{(d)}$

calculate the error $E(\boldsymbol{w}) = \frac{1}{2}\left(y^{(d)} - o^{(d)}\right)^2$

calculate the gradient

$$\nabla E(\boldsymbol{w}) = \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \cdots, \frac{\partial E}{\partial w_n}\right]$$

update the weights

$$\Delta \boldsymbol{w} = -\eta \nabla E(\boldsymbol{w})$$

Taking derivatives in neural nets

recall the chain rule from calculus

$$y = f(u)$$
$$u = g(x)$$

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial u}\frac{\partial u}{\partial x}$$

we'll make use of this as follows

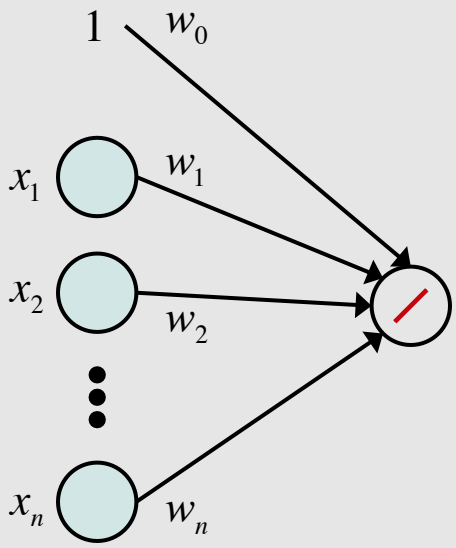
$$\frac{\partial E}{\partial w_i} = \frac{\partial E}{\partial o}\frac{\partial o}{\partial net}\frac{\partial net}{\partial w_i}$$

$$net = w_0 + \sum_{i=1}^{n} w_i x_i$$

Gradient descent: simple case

Consider a simple case of a network with one linear output unit and no hidden units:

$$o^{(d)} = w_0 + \sum_{i=1}^{n} w_i x_i^{(d)}$$



let's learn w_i's that minimize squared error

$$E(\boldsymbol{w}) = \frac{1}{2} \sum_{d \in \mathrm{D}} \left(y^{(d)} - o^{(d)} \right)^2$$

batch case

$$\frac{\partial E}{\partial w_i} = \frac{\partial}{\partial w_i} \frac{1}{2} \sum_{d \in \mathrm{D}} \left(y^{(d)} - o^{(d)} \right)^2$$

online case

$$\frac{\partial E^{(d)}}{\partial w_i} = \frac{\partial}{\partial w_i} \frac{1}{2} \left(y^{(d)} - o^{(d)} \right)^2$$

Stochastic gradient descent: simple case

let's focus on the online case (stochastic gradient descent):

$$\frac{\partial E^{(d)}}{\partial w_i} = \frac{\partial}{\partial w_i}\frac{1}{2}\left(y^{(d)} - o^{(d)}\right)^2$$

$$= \left(y^{(d)} - o^{(d)}\right)\frac{\partial}{\partial w_i}\left(y^{(d)} - o^{(d)}\right)$$

$$= \left(y^{(d)} - o^{(d)}\right)\left(-\frac{\partial o^{(d)}}{\partial w_i}\right)$$

$$= -\left(y^{(d)} - o^{(d)}\right)\frac{\partial o^{(d)}}{\partial net^{(d)}}\frac{\partial net^{(d)}}{\partial w_i} = -\left(y^{(d)} - o^{(d)}\right)\frac{\partial net^{(d)}}{\partial w_i}$$

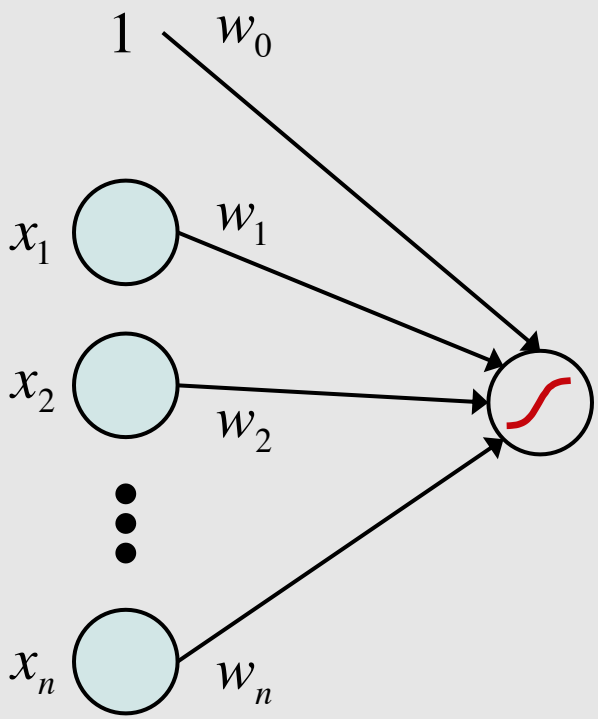
$$= -\left(y^{(d)} - o^{(d)}\right)x_i^{(d)}$$

Gradient descent with a sigmoid

Now let's consider the case in which we have a sigmoid output unit and no hidden units:

$$net^{(d)} = w_0 + \sum_{i=1}^{n} w_i x_i^{(d)}$$

$$o^{(d)} = \frac{1}{1 + e^{-net^{(d)}}}$$



useful property:

$$\frac{\partial o^{(d)}}{\partial net^{(d)}} = o^{(d)}(1 - o^{(d)})$$

Stochastic GD with sigmoid output unit

$$\frac{\partial E^{(d)}}{\partial w_i} = \frac{\partial}{\partial w_i}\frac{1}{2}\left(y^{(d)} - o^{(d)}\right)^2$$

$$= \left(y^{(d)} - o^{(d)}\right)\frac{\partial}{\partial w_i}\left(y^{(d)} - o^{(d)}\right)$$

$$= \left(y^{(d)} - o^{(d)}\right)\left(-\frac{\partial o^{(d)}}{\partial w_i}\right)$$

$$= -\left(y^{(d)} - o^{(d)}\right)\frac{\partial o^{(d)}}{\partial net^{(d)}}\frac{\partial net^{(d)}}{\partial w_i}$$

$$= -\left(y^{(d)} - o^{(d)}\right)o^{(d)}\left(1 - o^{(d)}\right)\frac{\partial net^{(d)}}{\partial w_i}$$

$$= -\left(y^{(d)} - o^{(d)}\right)o^{(d)}\left(1 - o^{(d)}\right)x_i^{(d)}$$

42

Backpropagation

- now we've covered how to do gradient descent for single-layer networks with
  - linear output units
  - sigmoid output units

- how can we calculate $\frac{\partial E}{\partial w_i}$ for every weight in a multilayer network?

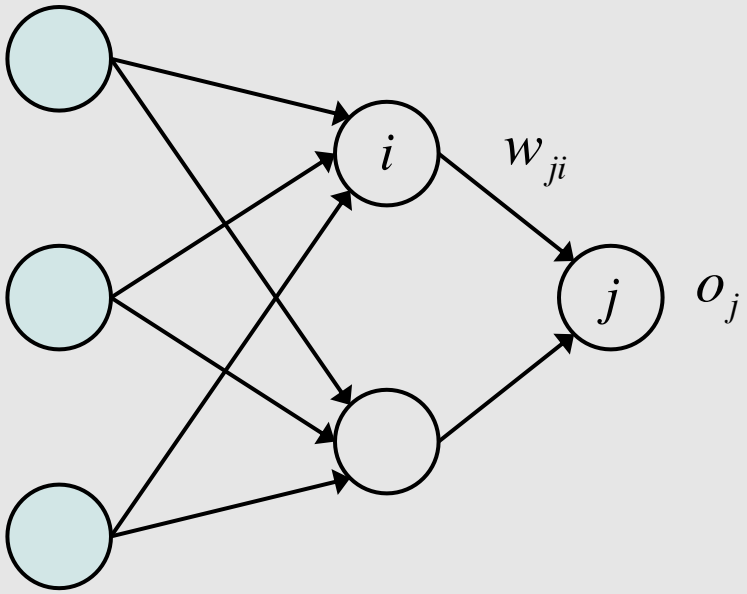
  ➔ backpropagate errors from the output units to the hidden units

Backpropagation notation

let's consider the online case, but drop the $^{(d)}$ superscripts for simplicity

we'll use

- subscripts on y, o, net to indicate which unit they refer to
- subscripts to indicate the unit a weight emanates from and goes to



Backpropagation

each weight is changed by

$$\Delta w_{ji} = -\eta \frac{\partial E}{\partial w_{ji}}$$

$$= -\eta \frac{\partial E}{\partial net_j} \frac{\partial net_j}{\partial w_{ji}}$$

$$= \eta \delta_j o_i$$

x_i if i is an input unit

where $\delta_j = -\frac{\partial E}{\partial net_j}$

Backpropagation

each weight is changed by $\Delta w_{ji} = \eta \delta_j o_i$

where $\delta_j = -\frac{\partial E}{\partial net_j}$

$\delta_j = o_j(1 - o_j)(y_j - o_j)$ if j is an output unit

same as single-layer net with sigmoid output

$\delta_j = o_j(1 - o_j) \sum_k \delta_k w_{kj}$ if j is a hidden unit

sum of backpropagated contributions to error

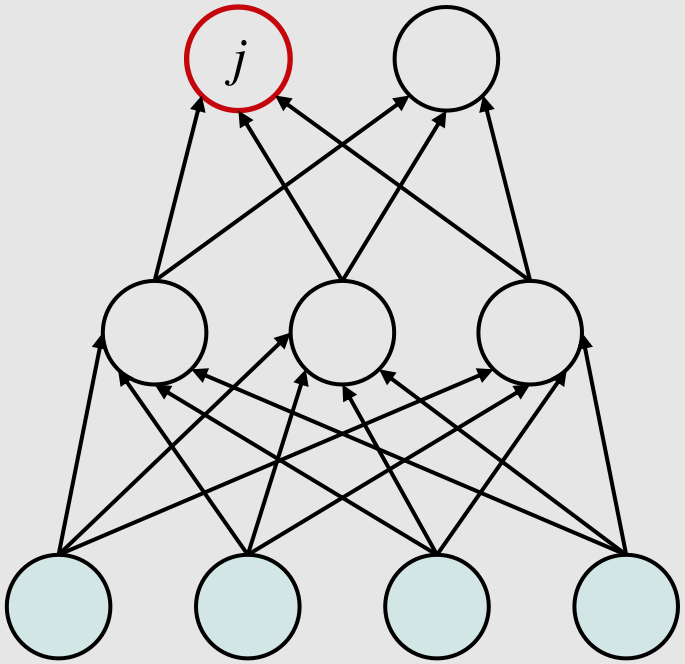
Backpropagation illustrated

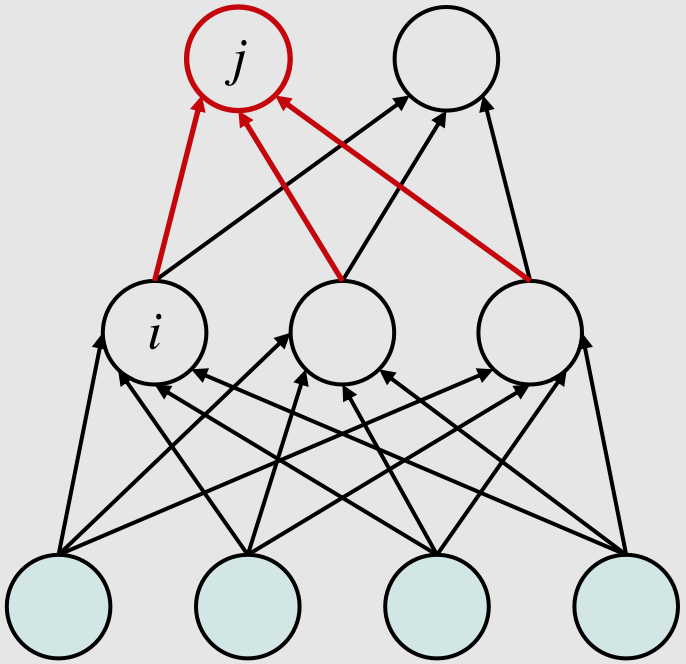
1. calculate error of output units

$$\delta_j = o_j(1 - o_j)(y_j - o_j)$$

2. determine updates for weights going to output units

$$\Delta w_{ji} = \eta \delta_j o_i$$

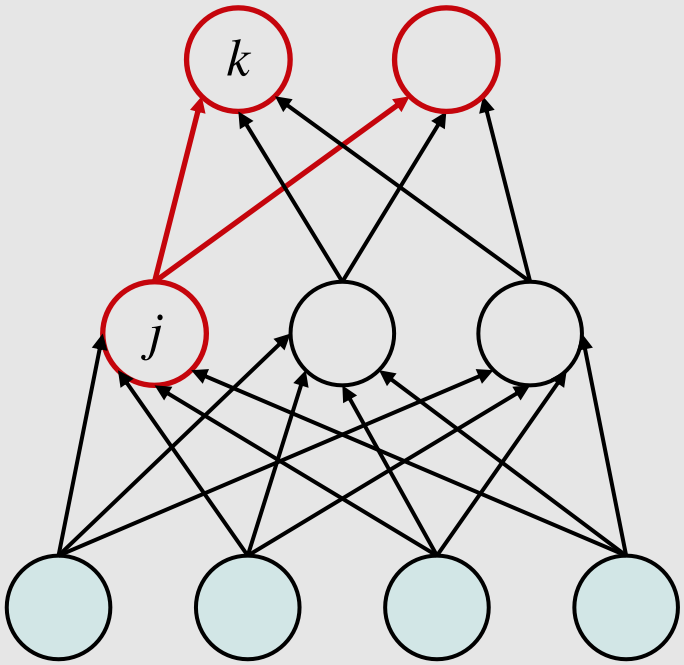




Backpropagation illustrated

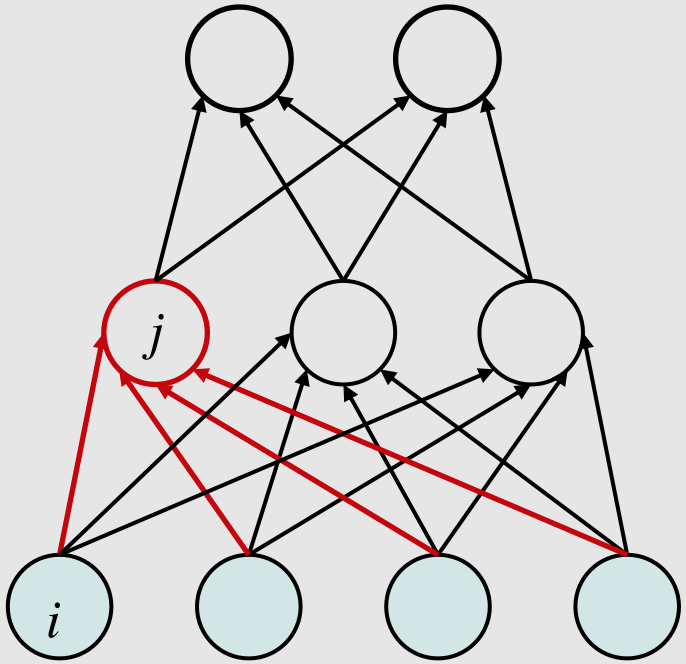
3. calculate error for hidden units

$$\delta_j = o_j(1 - o_j) \sum_k \delta_k w_{kj}$$



4. determine updates for weights to hidden units using hidden-unit errors

$$\Delta w_{ji} = \eta \delta_j o_i$$



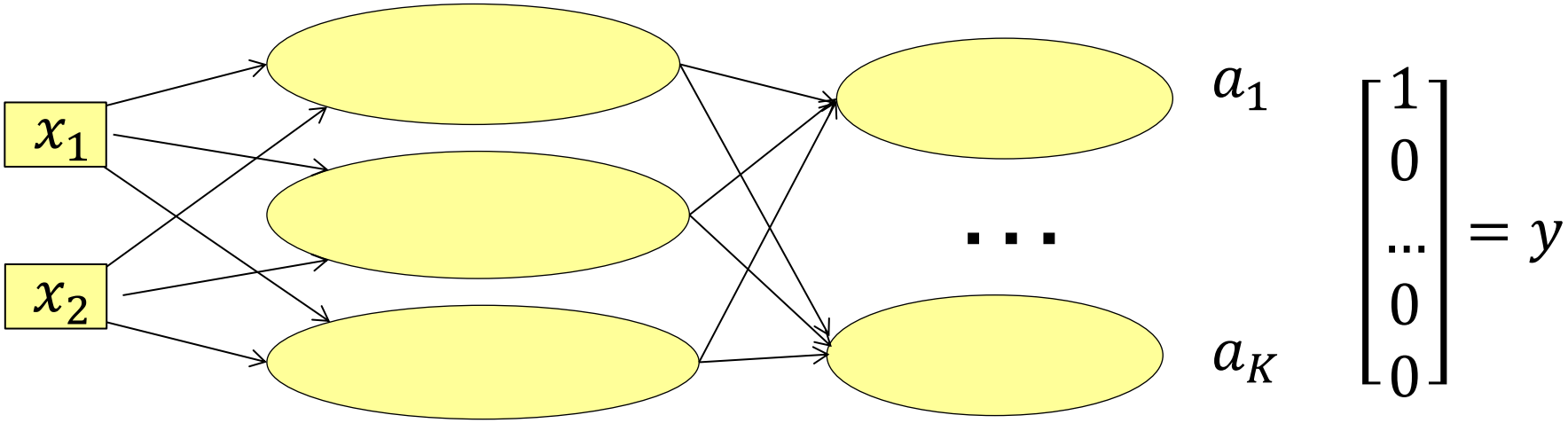

More Illustration of
Backpropagation

Learning in neural network

- Again we will minimize the error (K outputs):

$$E = \frac{1}{2}\sum_{x \in D} E_x\,, \qquad E_x = \|y - a\|^2 = \sum_{c=1}^{K} (a_c - y_c)^2$$

- x: one training point in the training set D
- a_c: the c-th output for the training point x
- y_c: the c-th element of the label indicator vector for x



Learning in neural network

- Again we will minimize the error (K outputs):

$$E = \frac{1}{2}\sum_{x \in D} E_x\,, \qquad E_x = \|y - a\|^2 = \sum_{c=1}^{K} (a_c - y_c)^2$$

- x: one training point in the training set D
- a_c: the c-th output for the training point x
- y_c: the c-th element of the label indicator vector for x
- Our variables are all the weights w on all the edges
  - Apparent difficulty: we don't know the 'correct' output of hidden units

Learning in neural network

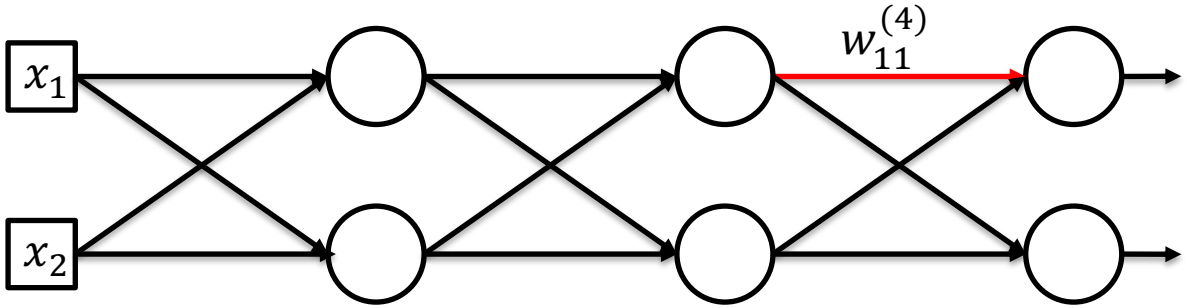
- Again we will minimize the error (K outputs):

$$E = \frac{1}{2}\sum_{x \in D} E_x\,, \qquad E_x = \|y - a\|^2 = \sum_{c=1}^{K} (a_c - y_c)^2$$

- x: one training point in the training set D
- a_c: the c-th output for the training point x
- y_c: the c-th element of the label indicator vector for x
- Our variables are all the weights w on all the edges
  - Apparent difficulty: we don't know the 'correct' output of hidden units
  - It turns out to be OK: we can still do gradient descent. The trick you need is the chain rule
  - The algorithm is known as **back-propagation**

Gradient (on one data point)

Layer (1) Layer (2) Layer (3) Layer (4)

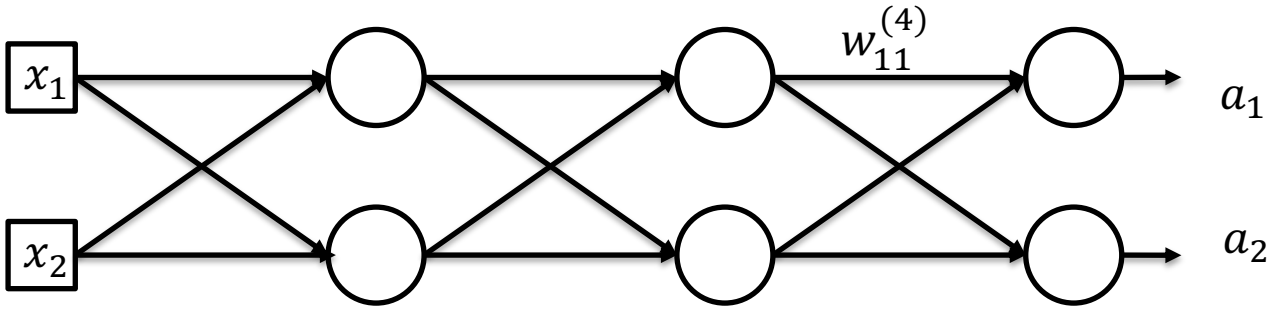


E_x

want to compute $\frac{\partial E_x}{\partial w_{11}^{(4)}}$

Gradient (on one data point)

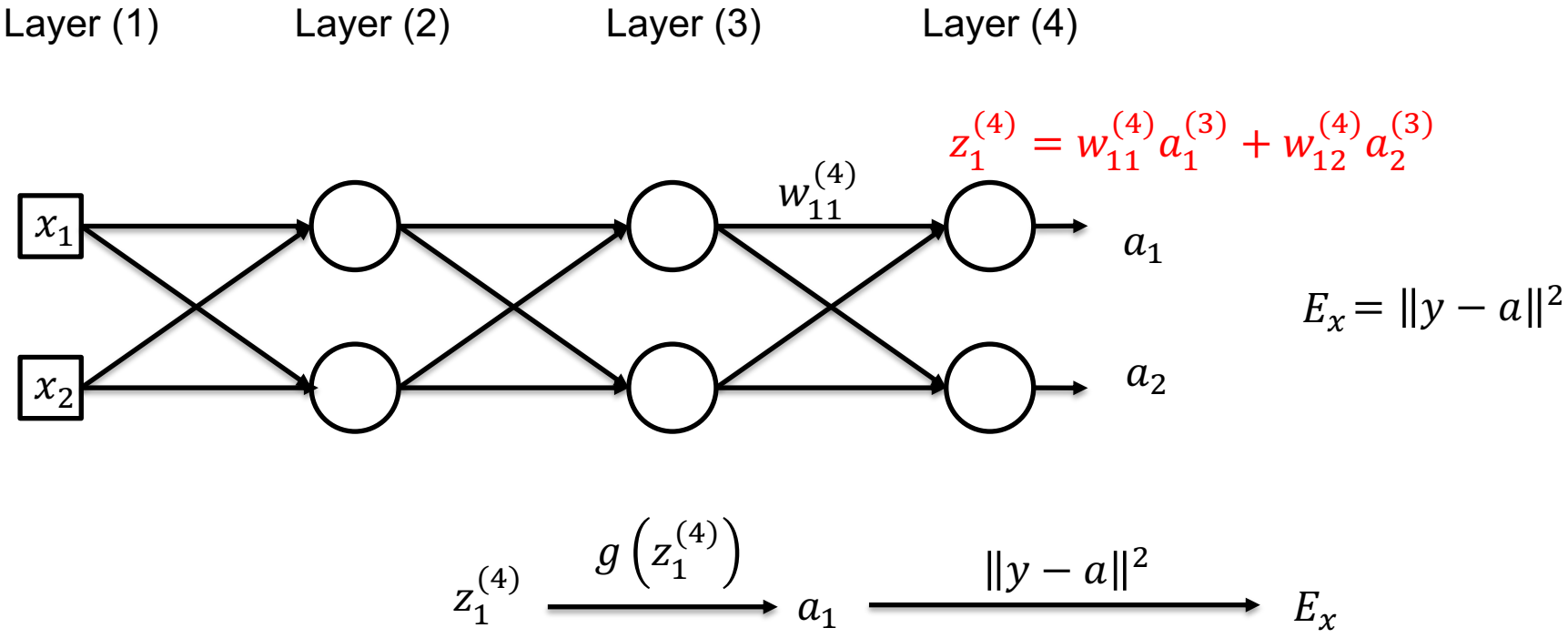
Layer (1) Layer (2) Layer (3) Layer (4)



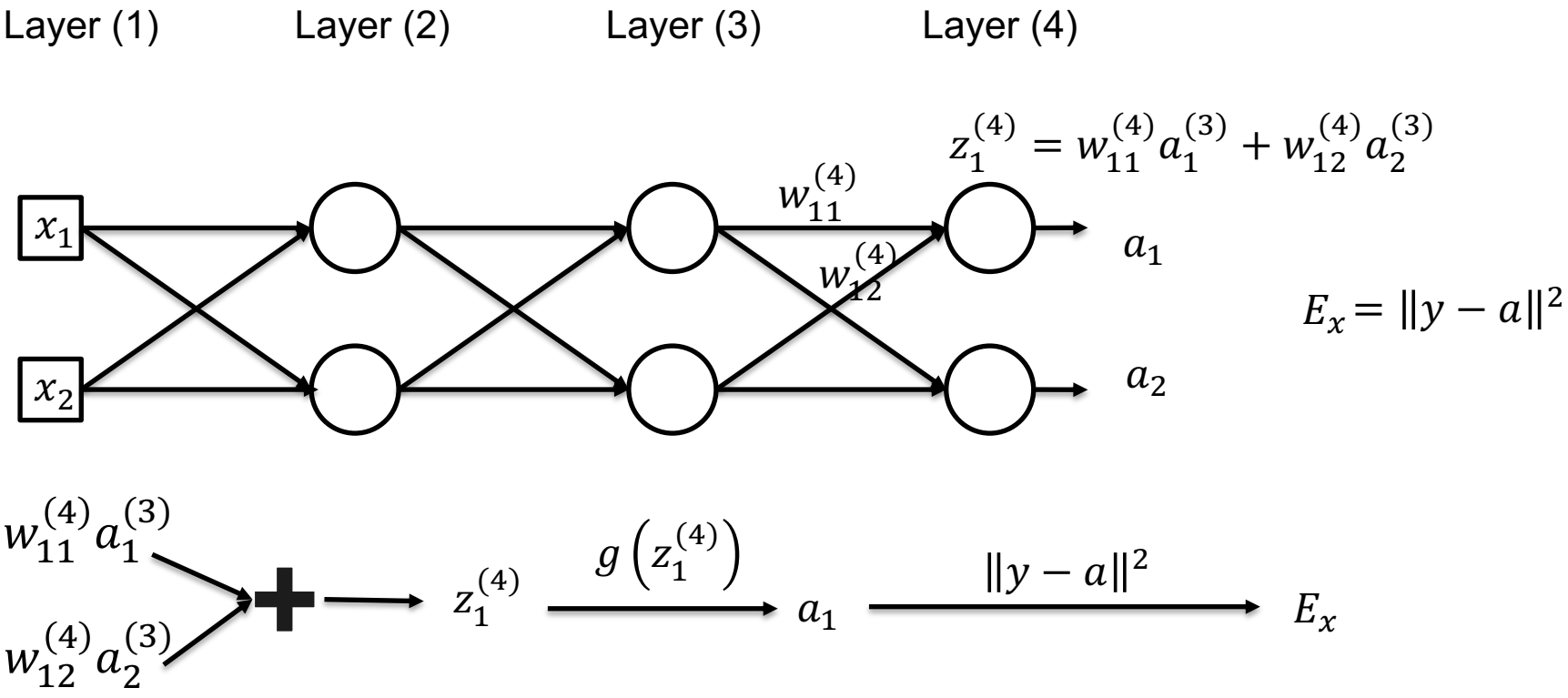
$$E_x = \|y - a\|^2$$

$$a_1 \xrightarrow{\|y-a\|^2} E_x$$

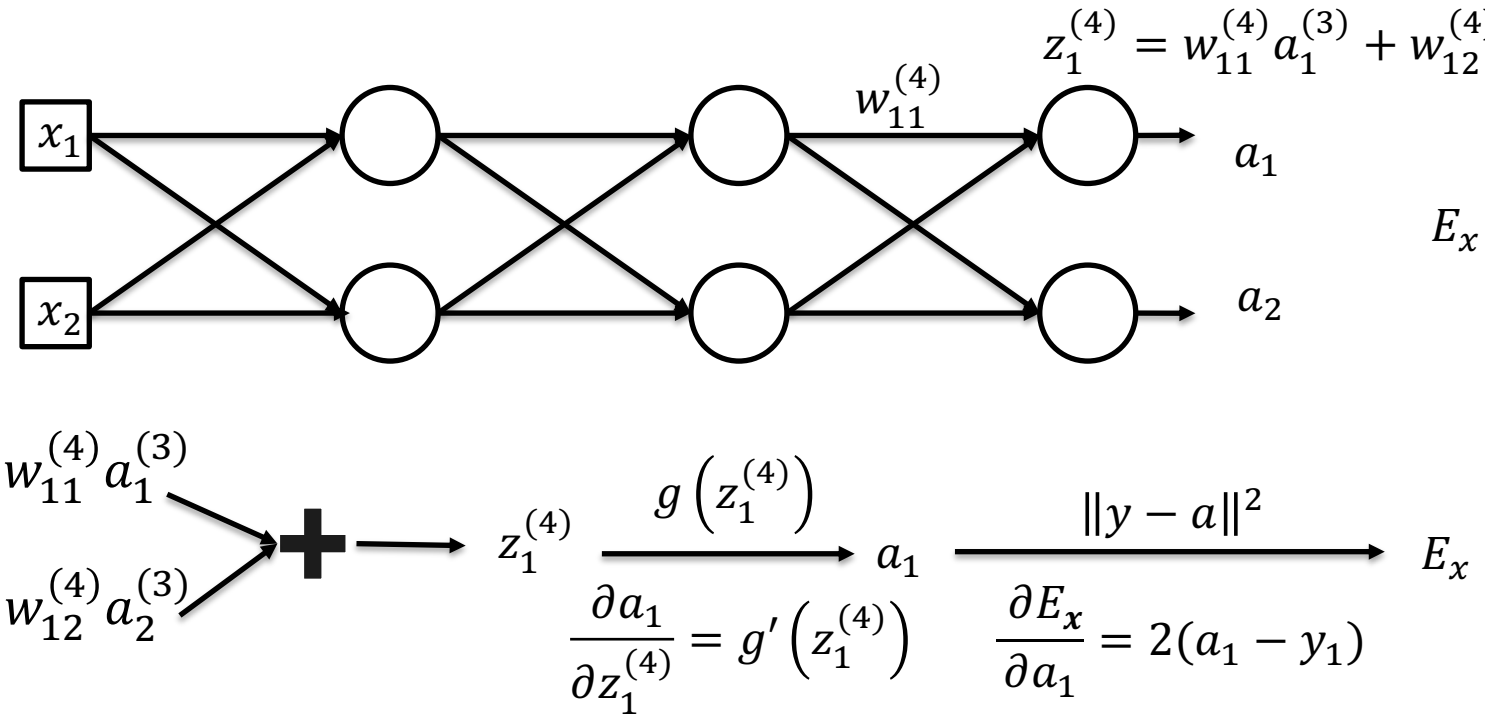
Gradient (on one data point)



Gradient (on one data point)



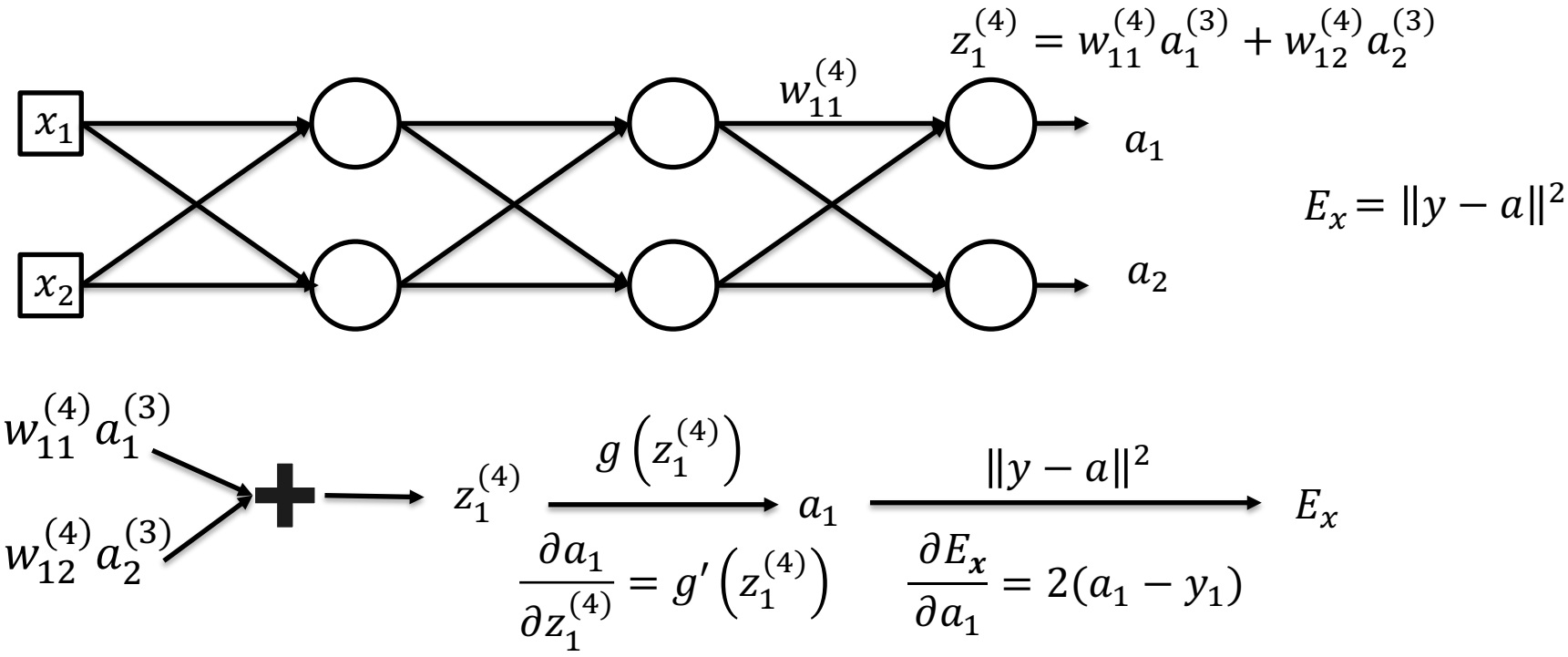
Gradient (on one data point)



By Chain Rule:

$$\frac{\partial E_x}{\partial w_{11}^{(4)}} = \frac{\partial E_x}{\partial a_1} \frac{\partial a_1}{\partial z_1^{(4)}} \frac{\partial z_1^{(4)}}{\partial w_{11}^{(4)}}$$

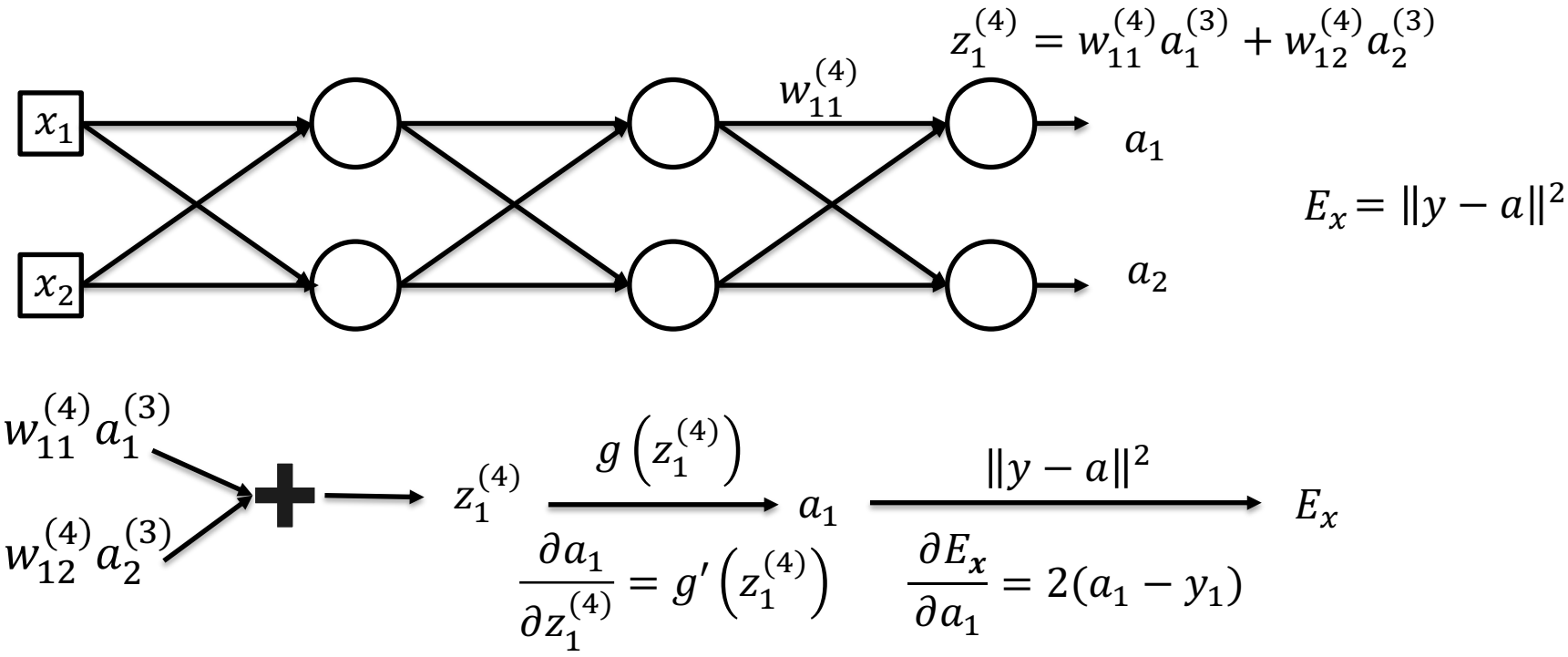
Gradient (on one data point)



By Chain Rule:

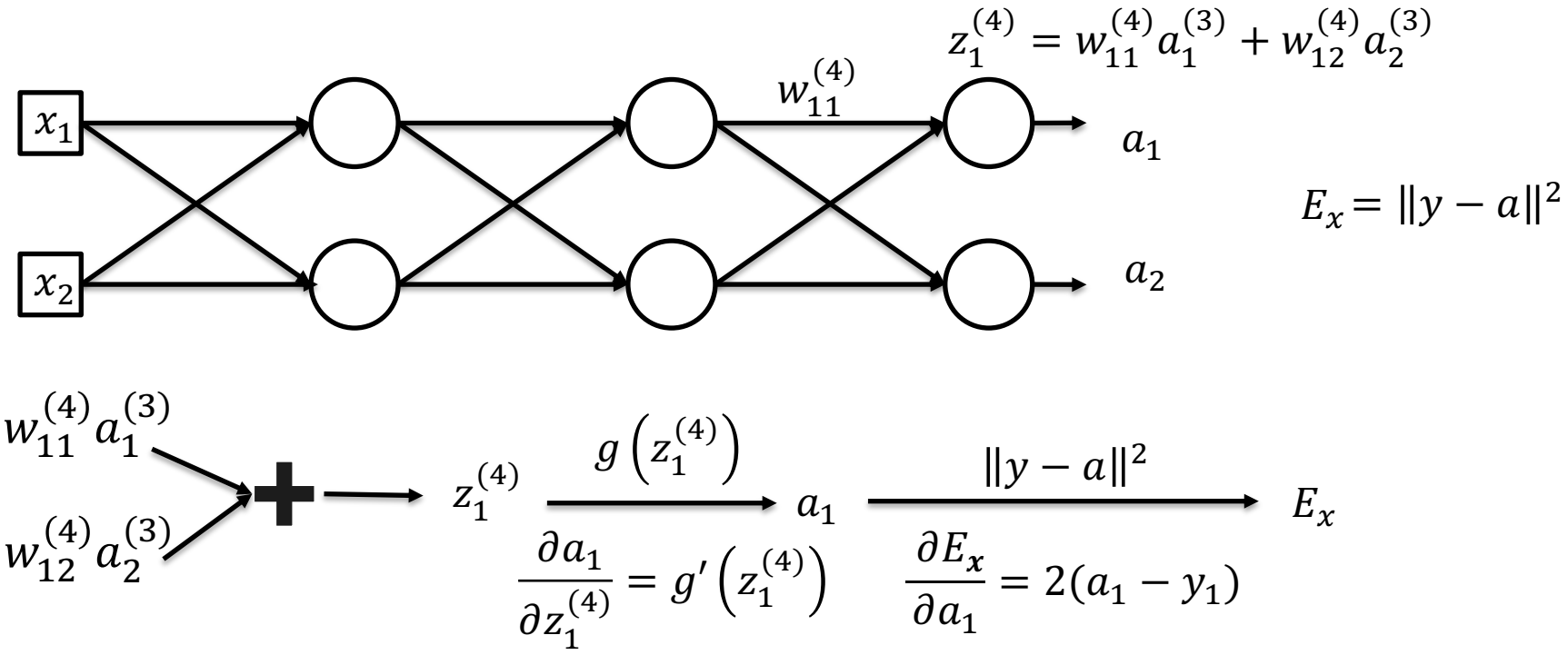
$$\frac{\partial E_x}{\partial w_{11}^{(4)}} = 2(a_1 - y_1) g'\left(z_1^{(4)}\right) \frac{\partial z_1^{(4)}}{\partial w_{11}^{(4)}}$$

Gradient (on one data point)



By Chain Rule: $$\frac{\partial E_x}{\partial w_{11}^{(4)}} = 2(a_1 - y_1) g'\left(z_1^{(4)}\right) a_1^{(3)}$$

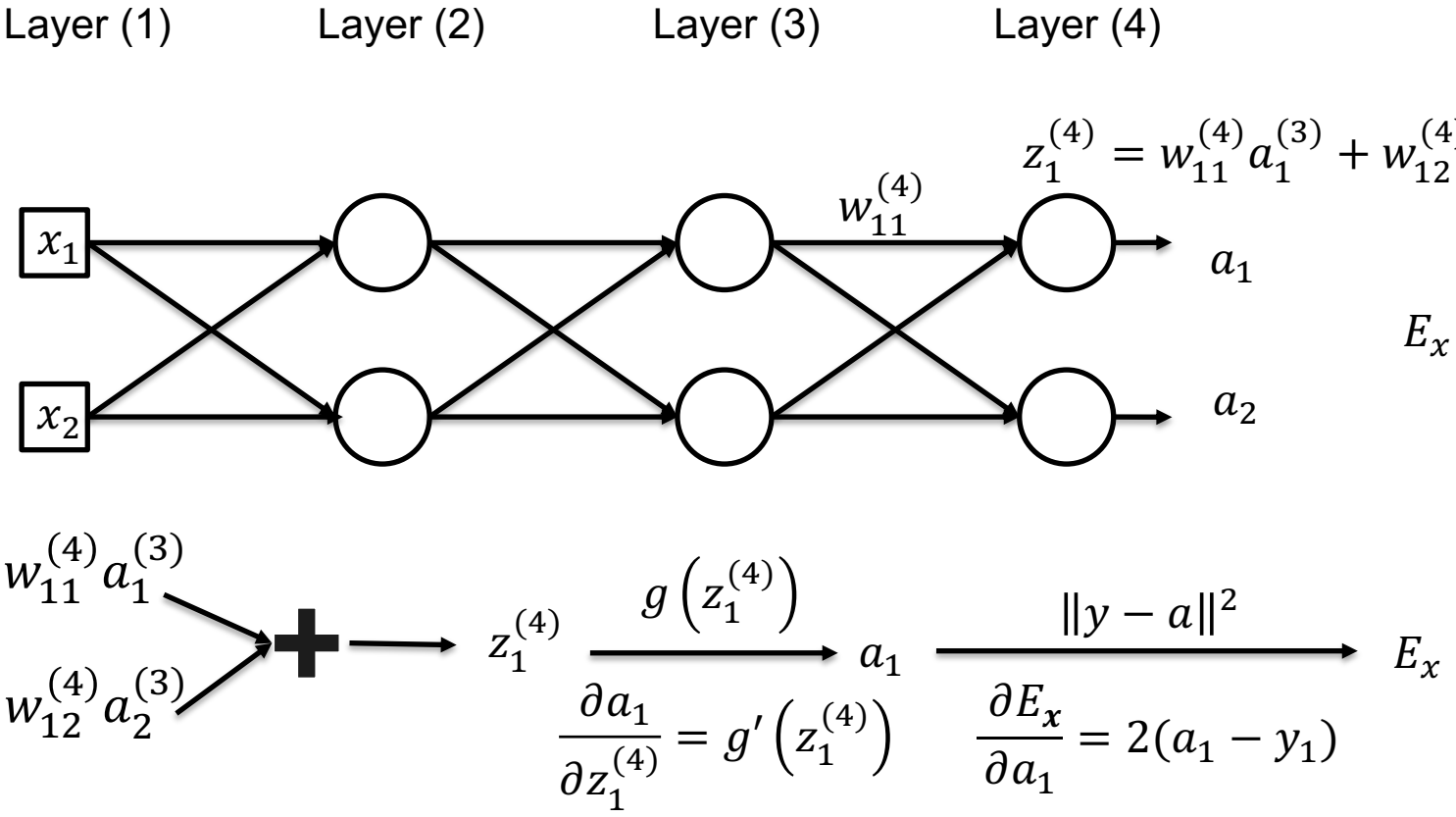
Gradient (on one data point)



By Chain Rule:

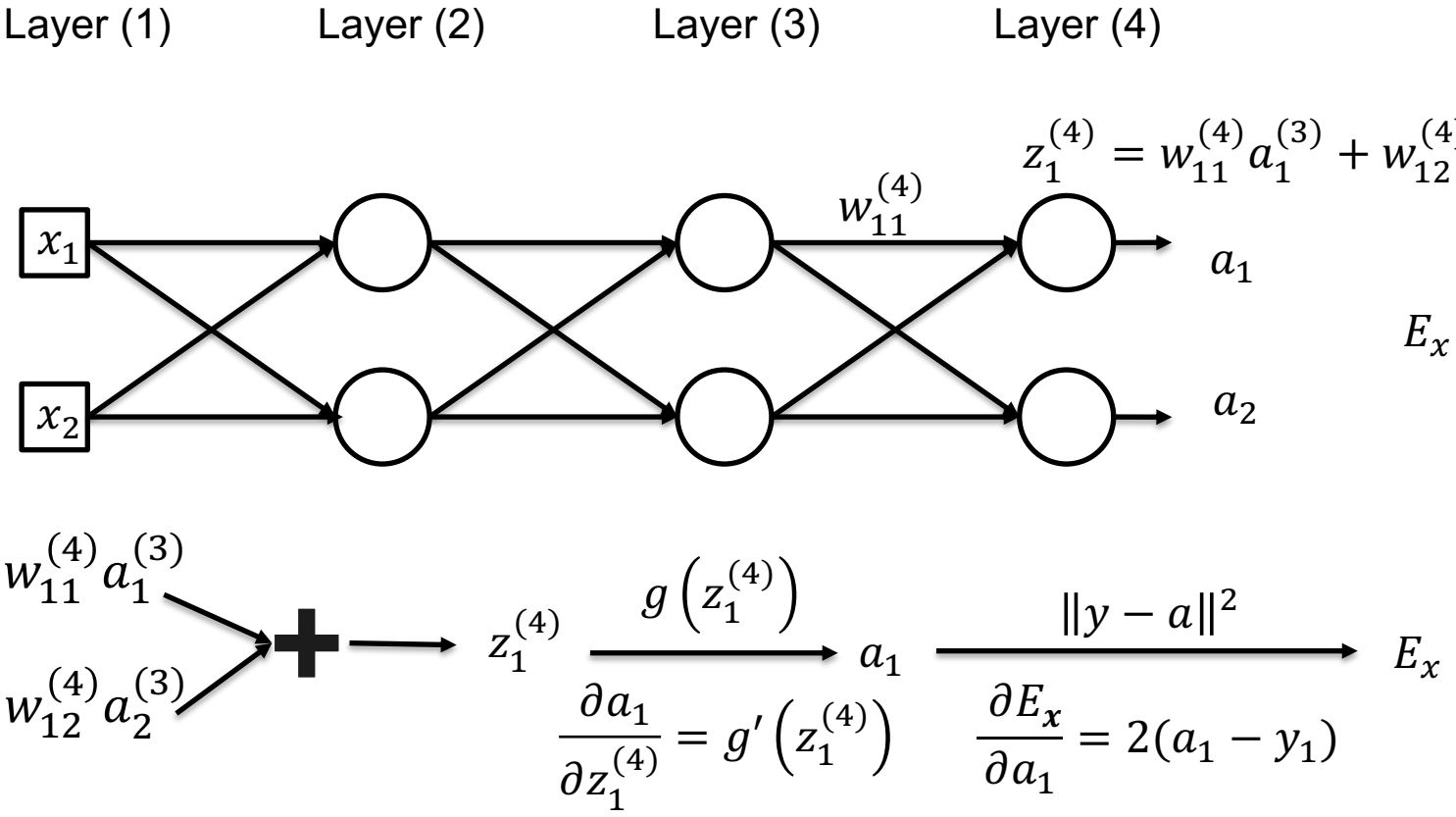
$$\frac{\partial E_x}{\partial w_{11}^{(4)}} = 2(a_1 - y_1) g\left(z_1^{(4)}\right)\left(1 - g\left(z_1^{(4)}\right)\right) a_1^{(3)}$$

Gradient (on one data point)



By Chain Rule: $$\frac{\partial E_x}{\partial w_{11}^{(4)}} = 2(a_1 - y_1) a_1 (1 - a_1) a_1^{(3)}$$

Gradient (on one data point)

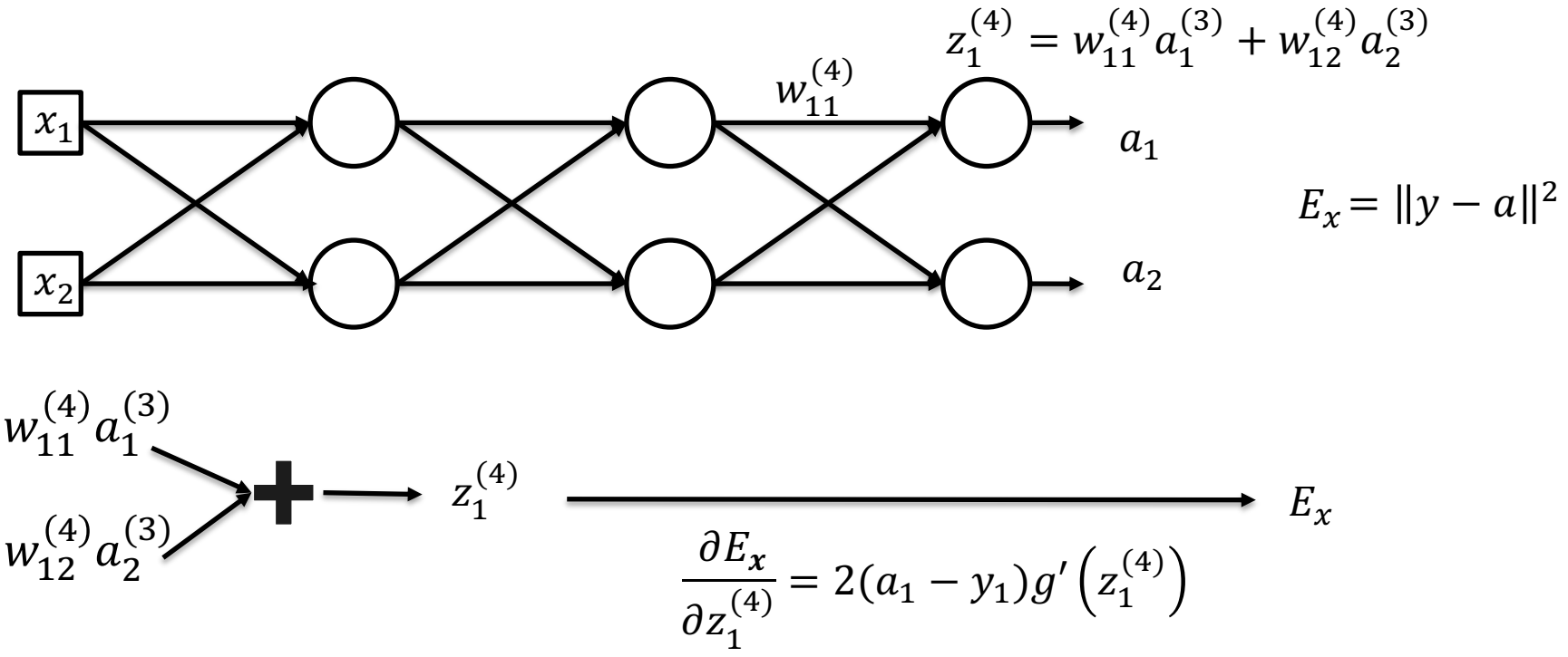


By Chain Rule: $$\frac{\partial E_x}{\partial w_{11}^{(4)}} = 2(a_1 - y_1) a_1 (1 - a_1) a_1^{(3)}$$

Can be computed by network activation

Backpropagation

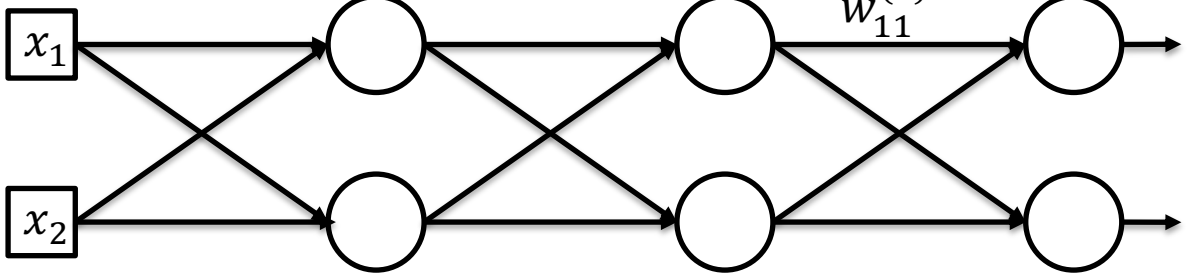
Layer (1) Layer (2) Layer (3) Layer (4)



By Chain Rule: $\frac{\partial E_x}{\partial w_{11}^{(4)}} = 2(a_1 - y_1) a_1 (1 - a_1) a_1^{(3)}$

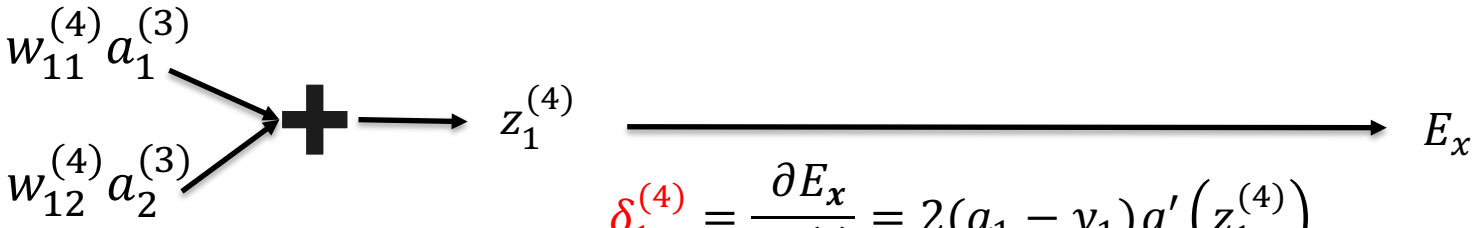
Backpropagation

Layer (1) Layer (2) Layer (3) Layer (4)



$$z_1^{(4)} = w_{11}^{(4)} a_1^{(3)} + w_{12}^{(4)} a_2^{(3)}$$

$$E_x = \|y - a\|^2$$

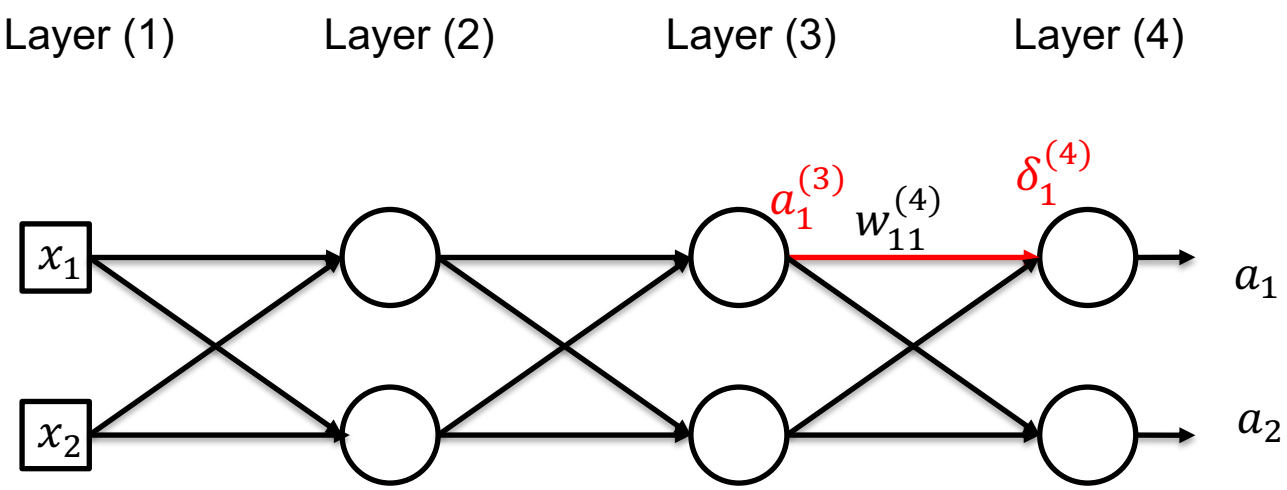


$$\delta_1^{(4)} = \frac{\partial E_x}{\partial z_1^{(4)}} = 2(a_1 - y_1) g'\left(z_1^{(4)}\right)$$

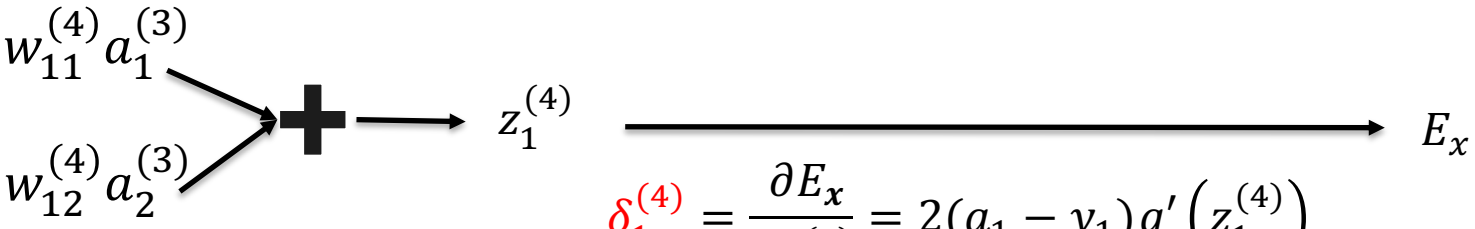
By Chain Rule:

$$\frac{\partial E_x}{\partial w_{11}^{(4)}} = 2(a_1 - y_1) a_1 (1 - a_1) a_1^{(3)}$$

Backpropagation



$$E_x = \|y - a\|^2$$

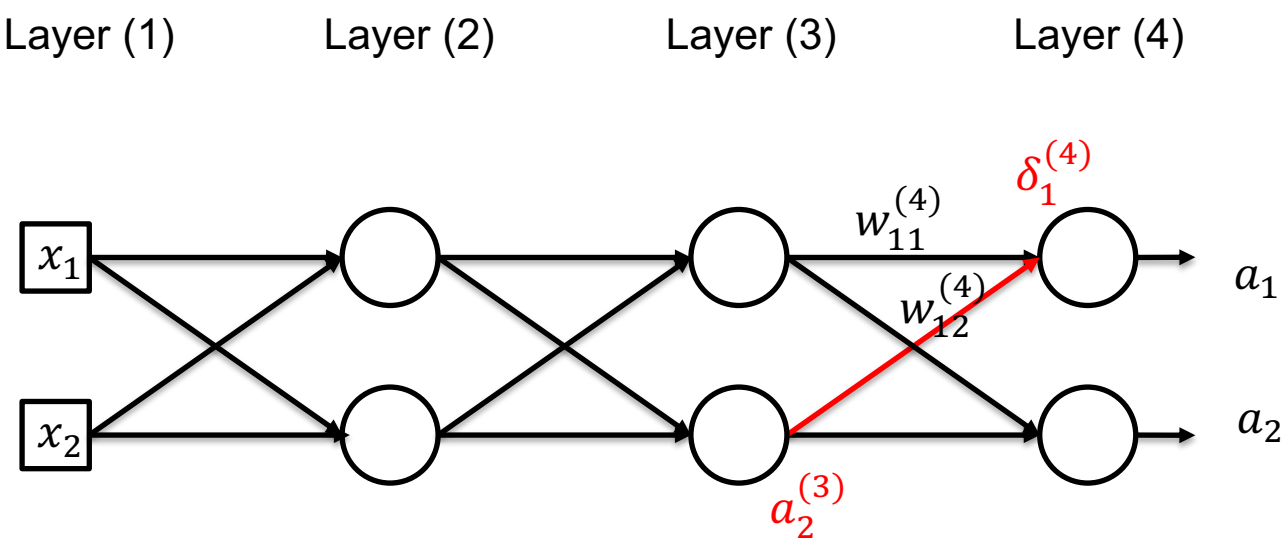


$$\delta_1^{(4)} = \frac{\partial E_x}{\partial z_1^{(4)}} = 2(a_1 - y_1) g'\left(z_1^{(4)}\right)$$

By Chain Rule:

$$\frac{\partial E_x}{\partial w_{11}^{(4)}} = \delta_1^{(4)} a_1^{(3)}$$

Backpropagation



$$E_x = \|y - a\|^2$$

$$w_{11}^{(4)} a_1^{(3)}$$
$$w_{12}^{(4)} a_2^{(3)}$$
$$z_1^{(4)} \longrightarrow E_x$$

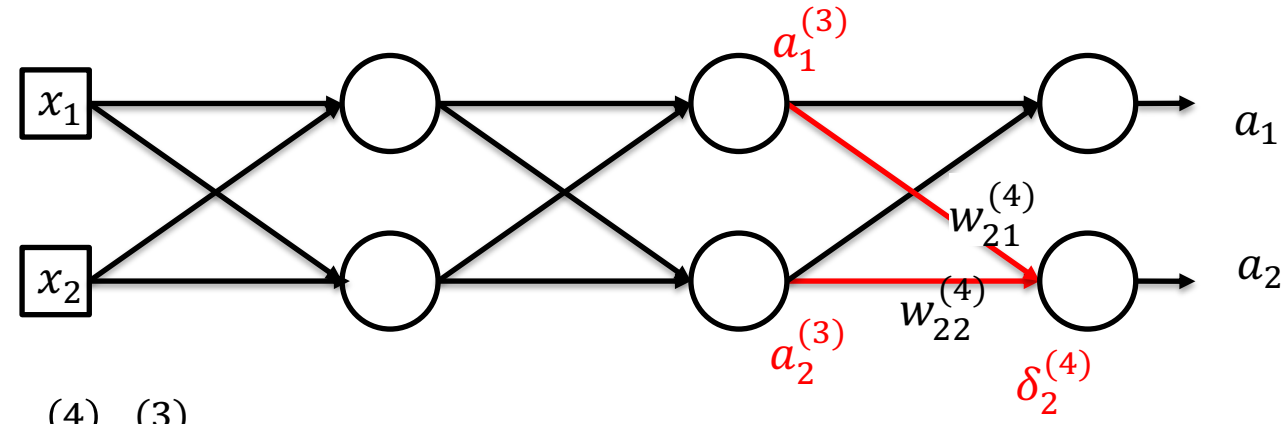
$$\delta_1^{(4)} = \frac{\partial E_x}{\partial z_1^{(4)}} = 2(a_1 - y_1) g'\left(z_1^{(4)}\right)$$

By Chain Rule:

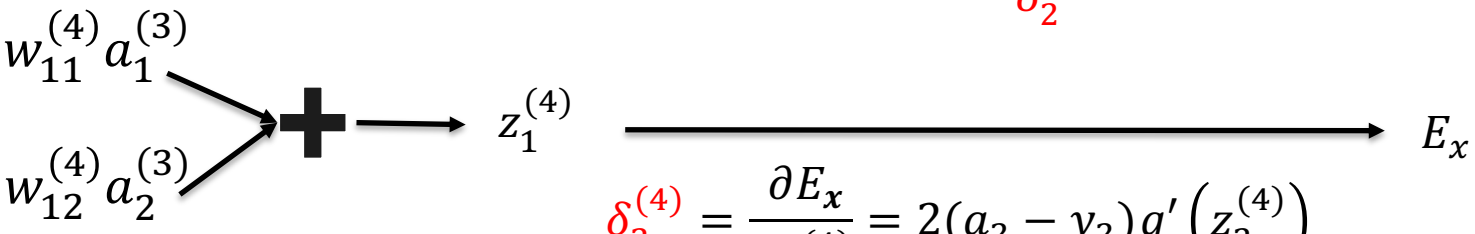
$$\frac{\partial E_x}{\partial w_{11}^{(4)}} = \delta_1^{(4)} a_1^{(3)}, \qquad \frac{\partial E_x}{\partial w_{12}^{(4)}} = \delta_1^{(4)} a_2^{(3)}$$

Backpropagation

Layer (1) Layer (2) Layer (3) Layer (4)



$$E_x = \|y - a\|^2$$



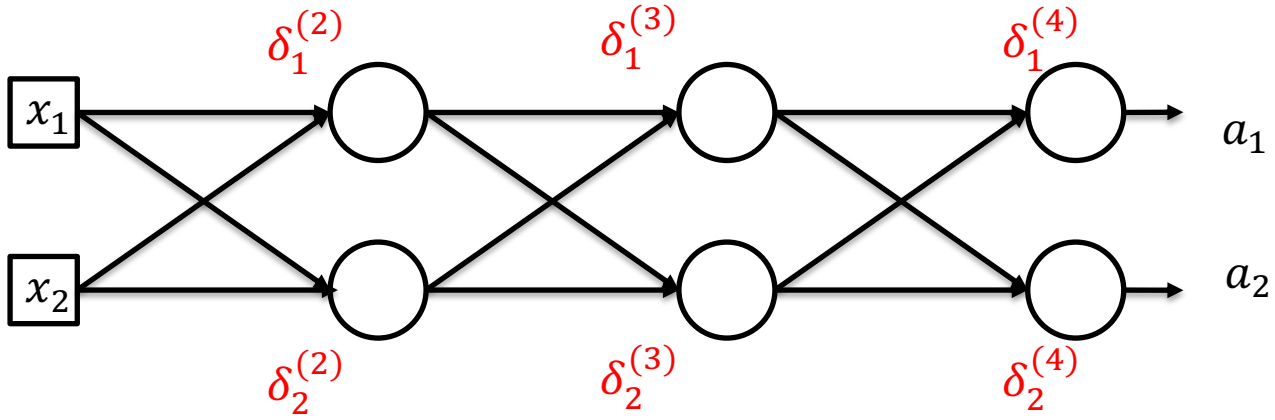
$$\delta_2^{(4)} = \frac{\partial E_x}{\partial z_2^{(4)}} = 2(a_2 - y_2) g'\left(z_2^{(4)}\right)$$

By Chain Rule:

$$\frac{\partial E_x}{\partial w_{21}^{(4)}} = \delta_2^{(4)} a_1^{(3)}, \qquad \frac{\partial E_x}{\partial w_{22}^{(4)}} = \delta_2^{(4)} a_2^{(3)}$$

Backpropagation

Layer (1) Layer (2) Layer (3) Layer (4)



$$E_x = \|y - a\|^2$$

Thus, for any weight in the network:

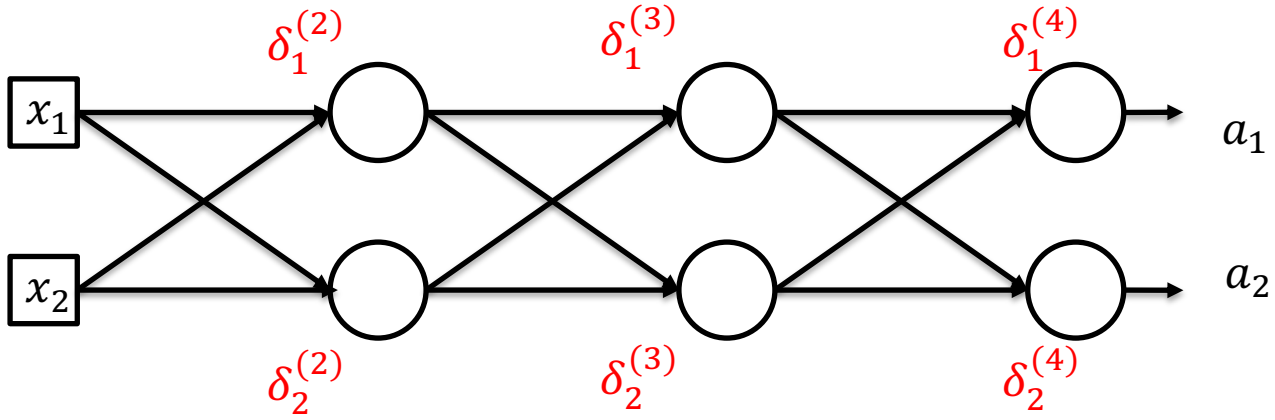
$$\frac{\partial E_x}{\partial w_{jk}^{(l)}} = \delta_j^{(l)} a_k^{(l-1)}$$

$\delta_j^{(l)}$: δ of j^{th} neuron in Layer l

$a_k^{(l-1)}$: Activation of k^{th} neuron in Layer $l-1$

$w_{jk}^{(l)}$: Weight from k^{th} neuron in Layer $l-1$ to j^{th} neuron in Layer l

Exercise



$$E_x = \|y - a\|^2$$

Show that for any bias in the network:

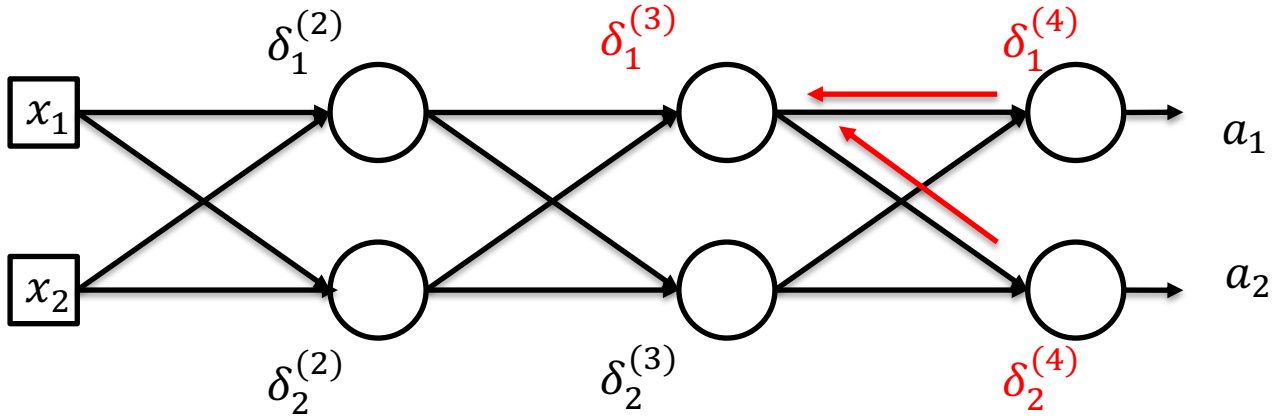
$$\frac{\partial E_x}{\partial b_j^{(l)}} = \delta_j^{(l)}$$

$\delta_j^{(l)}$: δ of j^{th} neuron in Layer l

$b_j^{(l)}$: bias for the j^{th} neuron in Layer l, i.e., $z_j^{(l)} = \sum_k w_{jk}^{(l)} a_k^{(l-1)} + b_j^{(l)}$

Backpropagation of δ

Layer (1) Layer (2) Layer (3) Layer (4)



$$E_x = \|y - a\|^2$$

Thus, for any neuron in the network:

$$\delta_j^{(l)} = \sum_k \delta_k^{(l+1)} w_{kj}^{(l+1)} \; g'\left(z_j^{(l)}\right)$$

$\delta_j^{(l)}$: δ of j^{th} Neuron in Layer l

$\delta_k^{(l+1)}$: δ of k^{th} Neuron in Layer $l+1$

$g'\left(z_j^{(l)}\right)$: derivative of j^{th} Neuron in Layer l w.r.t. its linear combination input

$w_{kj}^{(l+1)}$: Weight from j^{th} Neuron in Layer l to k^{th} Neuron in Layer $l+1$

Gradient descent with Backpropagation

1. Initialize Network with Random Weights and Biases
2. For each Training Image:
   a. Compute Activations for the Entire Network
   b. Compute δ for Neurons in the Output Layer using Network Activation and Desired Activation

$$\delta_j^{(L)} = 2\left(y_j - a_j\right)a_j(1 - a_j)$$

   c. Compute δ for all Neurons in the previous Layers

$$\delta_j^{(l)} = \sum_k \delta_k^{(l+1)} w_{kj}^{(l+1)} a_j^{(l)}(1 - a_j^{(l)})$$

   d. Compute Gradient of Cost w.r.t each Weight and Bias for the Training Image using δ

$$\frac{\partial E_x}{\partial w_{jk}^{(l)}} = \delta_j^{(l)} a_k^{(l-1)} \qquad \frac{\partial E_x}{\partial b_j^{(l)}} = \delta_j^{(l)}$$

Gradient descent with Backpropagation

3. Average the Gradient w.r.t. each Weight and Bias over the Entire Training Set

$$\frac{\partial E}{\partial w_{jk}^{(l)}} = \frac{1}{n}\sum \frac{\partial E_{\boldsymbol{x}}}{\partial w_{jk}^{(l)}} \qquad \frac{\partial E}{\partial b_{j}^{(l)}} = \frac{1}{n}\sum \frac{\partial E_{\boldsymbol{x}}}{\partial b_{j}^{(l)}}$$

4. Update the Weights and Biases using Gradient Descent

$$w_{jk}^{(l)} \leftarrow w_{jk}^{(l)} - \eta \frac{\partial E}{\partial w_{jk}^{(l)}} \qquad b_{j}^{(l)} \leftarrow b_{j}^{(l)} - \eta \frac{\partial E}{\partial b_{j}^{(l)}}$$

5. Repeat Steps 2-4 till Cost reduces below an acceptable level

THANK YOU

Some of the slides in these lectures have been adapted/borrowed from materials developed by Yingyu Liang, Mark Craven, David Page, Jude Shavlik, Tom Mitchell, Nina Balcan, Matt Gormley, Elad Hazan, Tom Dietterich, Pedro Domingos, and Mohit Gupta.