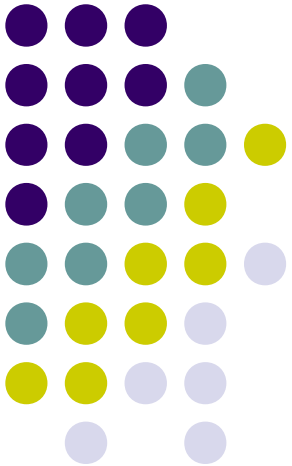
Pattern Recognition
Neural Networks

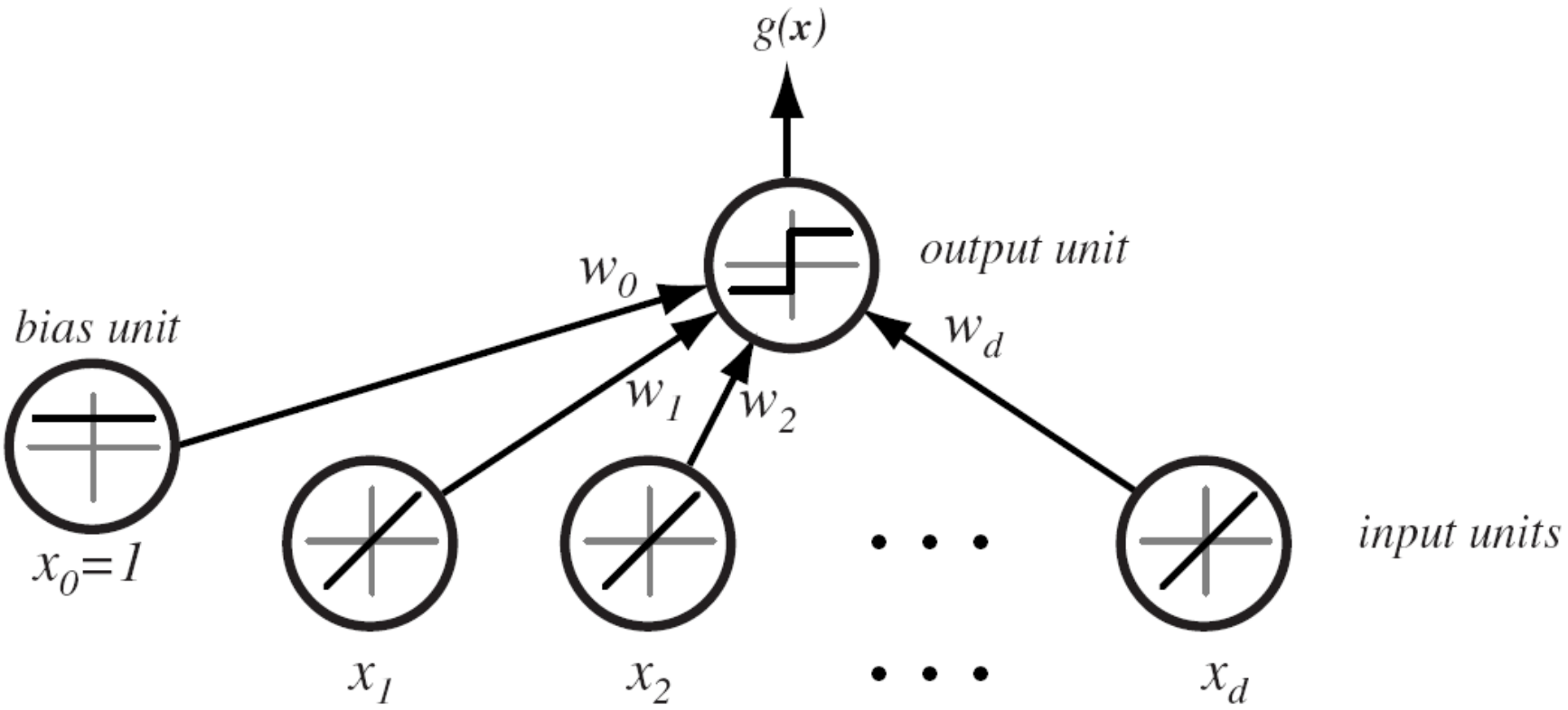
Francesco Tortorella

University of Cassino and Southern Latium

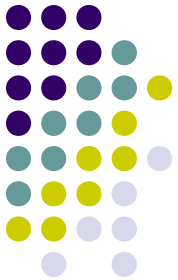
Cassino, Italy



The Perceptron model



Artificial neuron



- Neuron pre-activation (or input activation):

$$a(\mathbf{x}) = \sum_i w_i \cdot x_i + b = \mathbf{w}^T \mathbf{x} + b$$

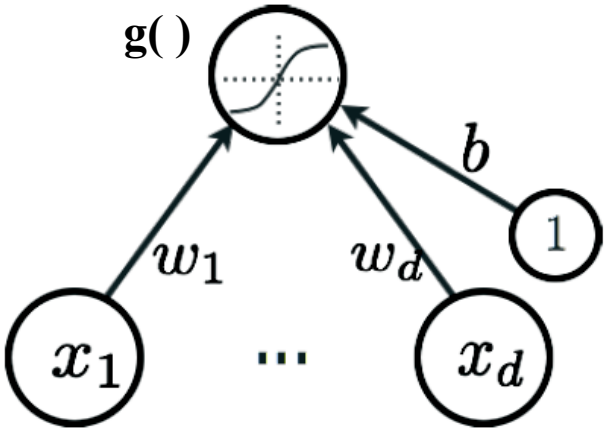
b is equivalent to w_0

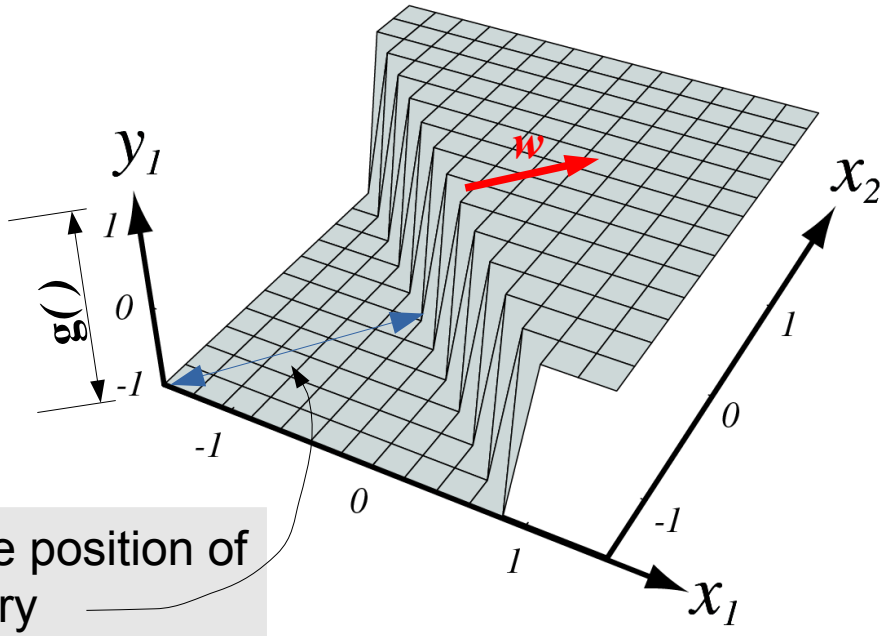
- Neuron (output) activation:

$$h(\mathbf{x}) = g\left(a(\mathbf{x})\right) = g\left(\mathbf{w}^T \mathbf{x} + b\right)$$

  - $\mathbf{w}$: connection weights
  - b : neuron bias
  - $g()$: activation function

Artificial neuron



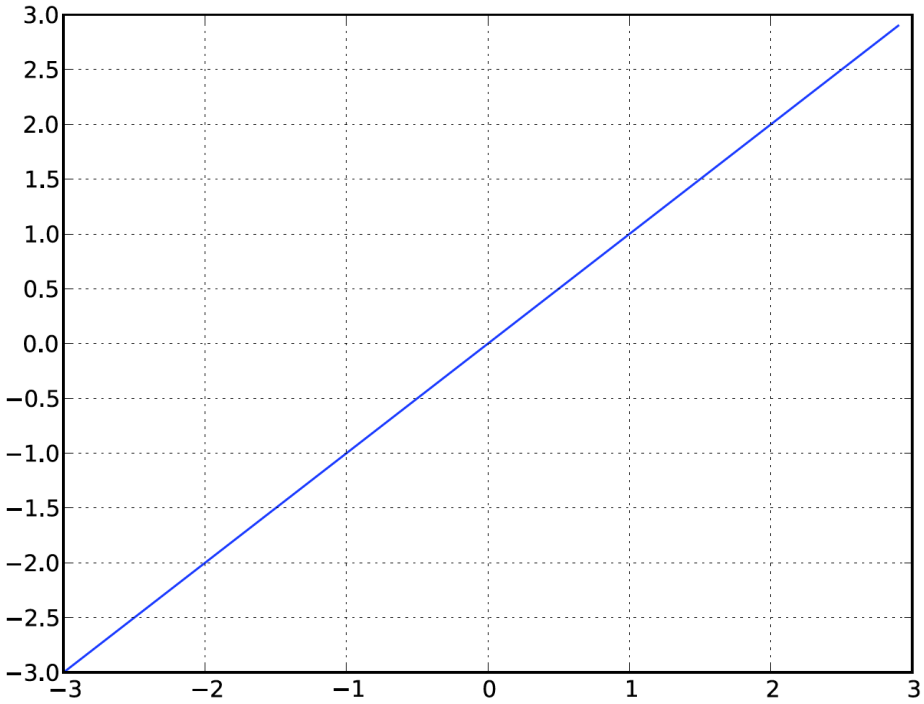


b affects the position of the boundary

Activation functions

- Linear activation function
  - Not bounded
  - ...

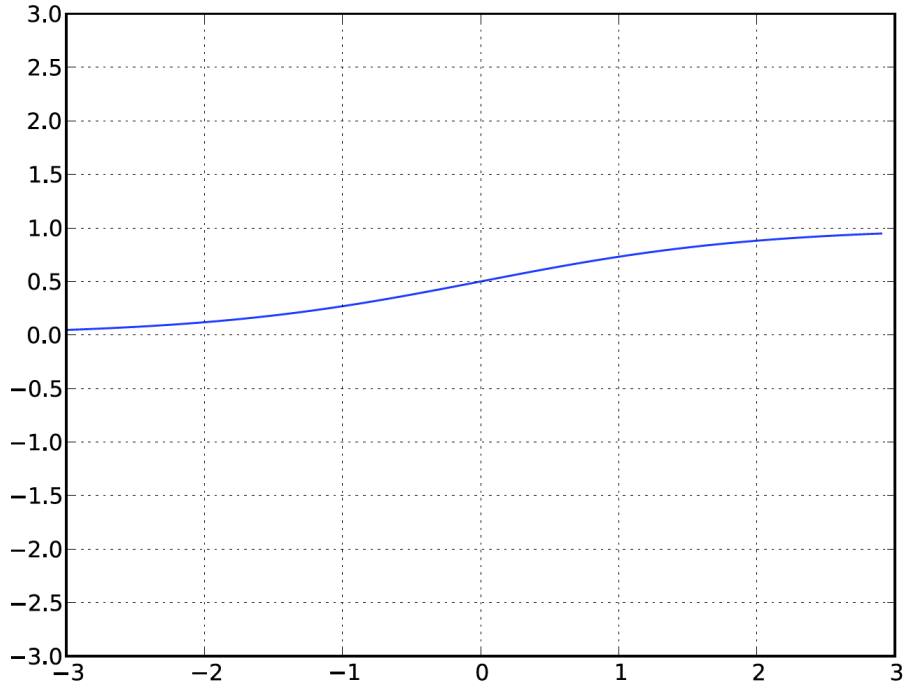
$$g(a) = a$$



Activation functions

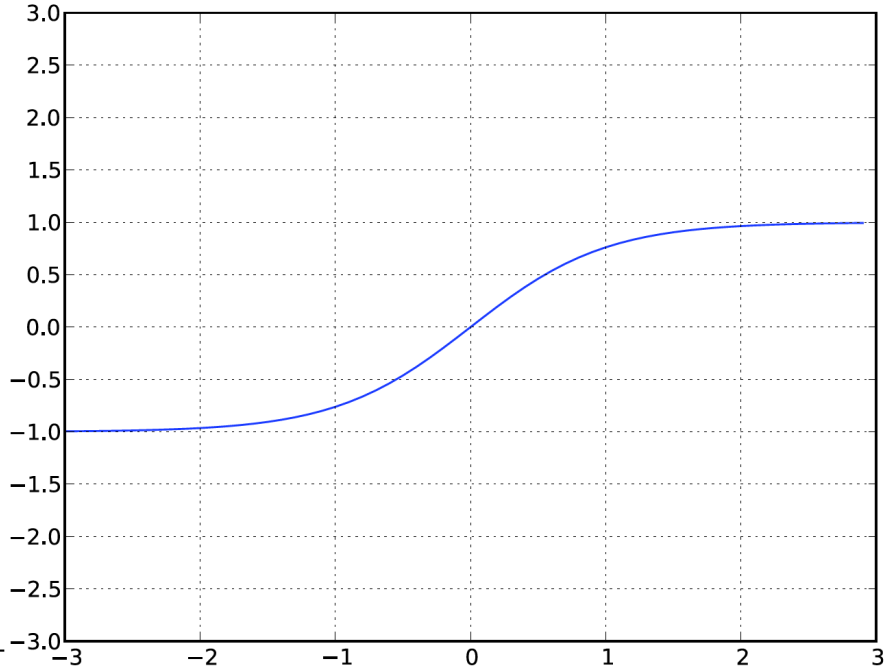
- Sigmoid activation function
  - Limit the input between 0 and 1
  - Always positive
  - Strictly increasing

$$g(a) = sigm(a) = \frac{1}{1 + exp(-a)}$$



Activation functions

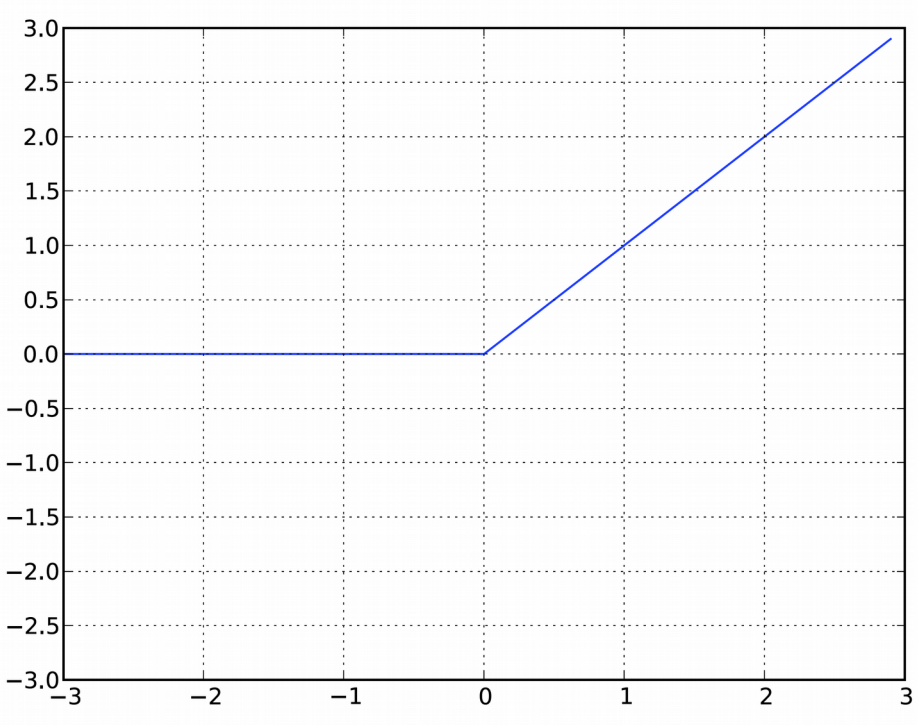
- Hyperbolic tangent (“tanh”) activation function
  - Limit the input between -1 and 1
  - Can be positive or negative
  - Strictly increasing



$$g(a) = tanh(a) = \frac{exp(a) - exp(-a)}{exp(a) + exp(-a)}$$

Activation functions

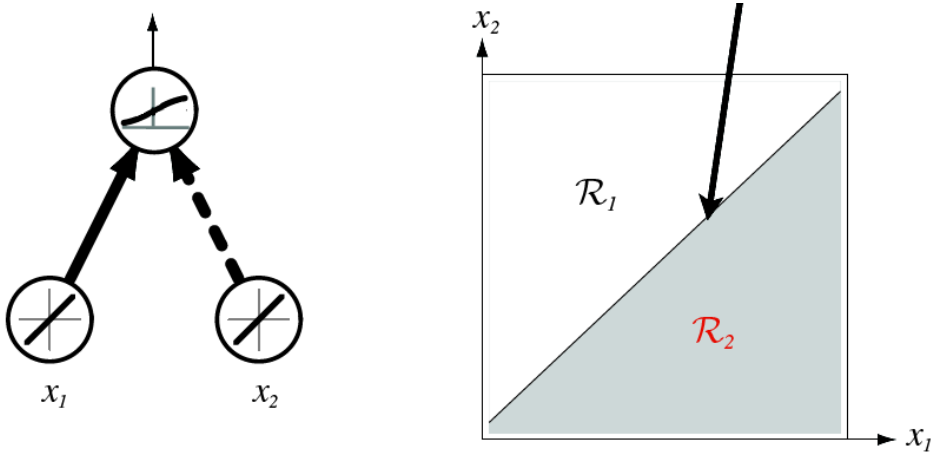
- Rectified linear activation function
  - Bounded below by 0 (always non-negative)
  - Not upper bounded
  - Strictly increasing



$$g(a) = reclin(a) = relu(a)$$
$$= \max(0, a)$$

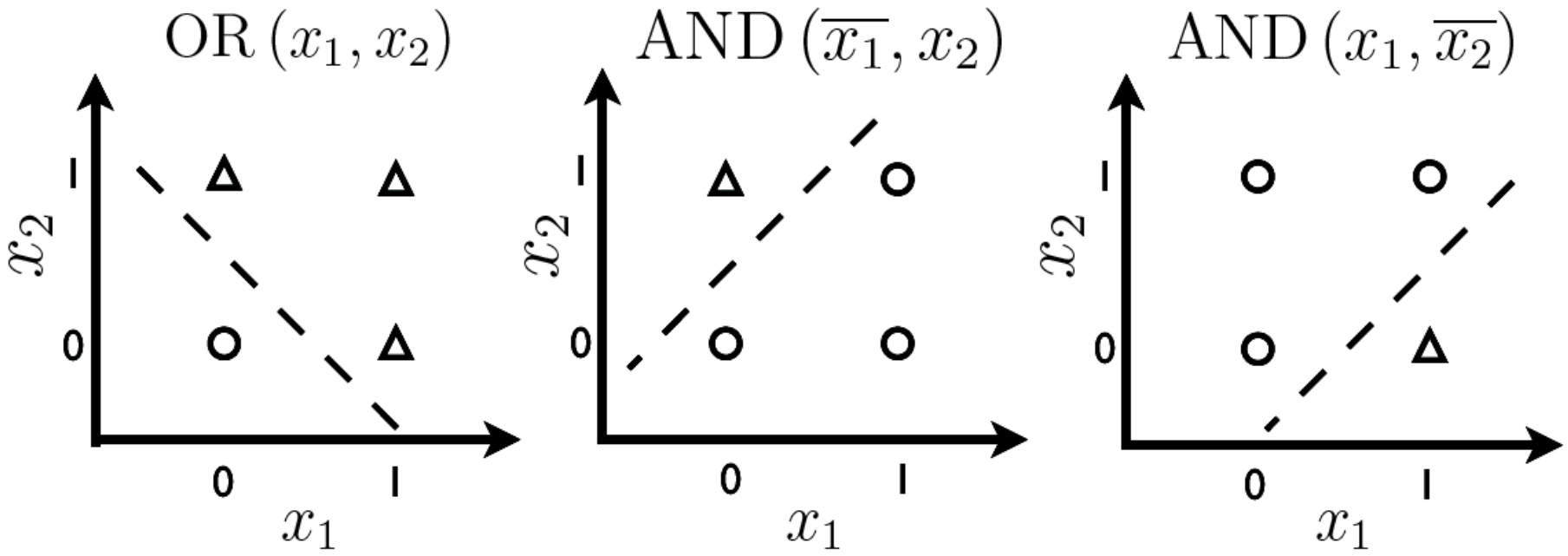
Capacity of the neuron

- Could do binary classification
- Only linear boundaries
- With sigmoid, can interpret neuron as estimating the post-probability $P(\omega_1|\mathbf{x})$

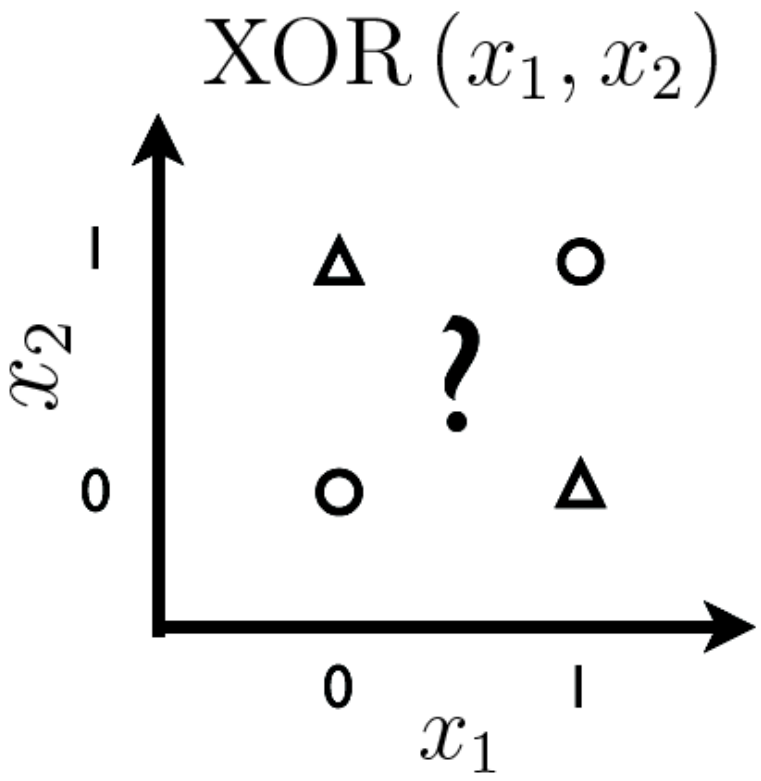


Capacity of the neuron

- Can solve only linearly separable problems

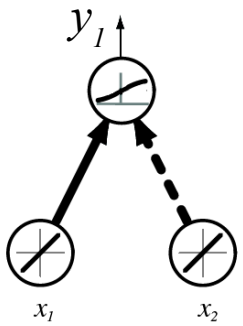


Capacity of the neuron

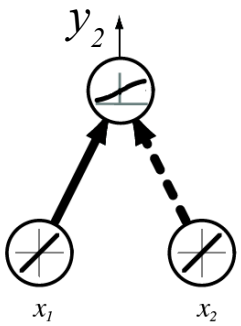


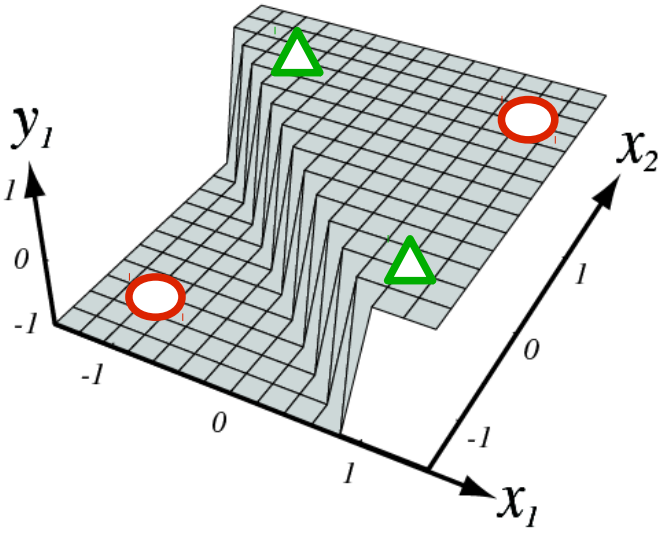
- Classical XOR problem
- What if we use two artificial neurons instead of only one?

Capacity of two neurons

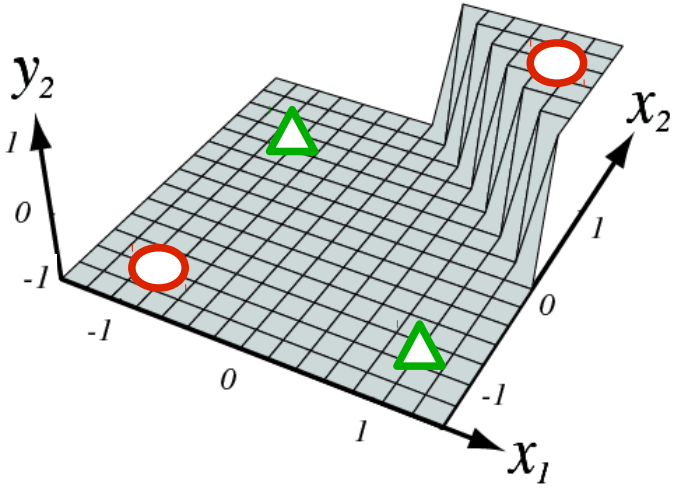


Two neurons with
- same input
- different weights
- different biases

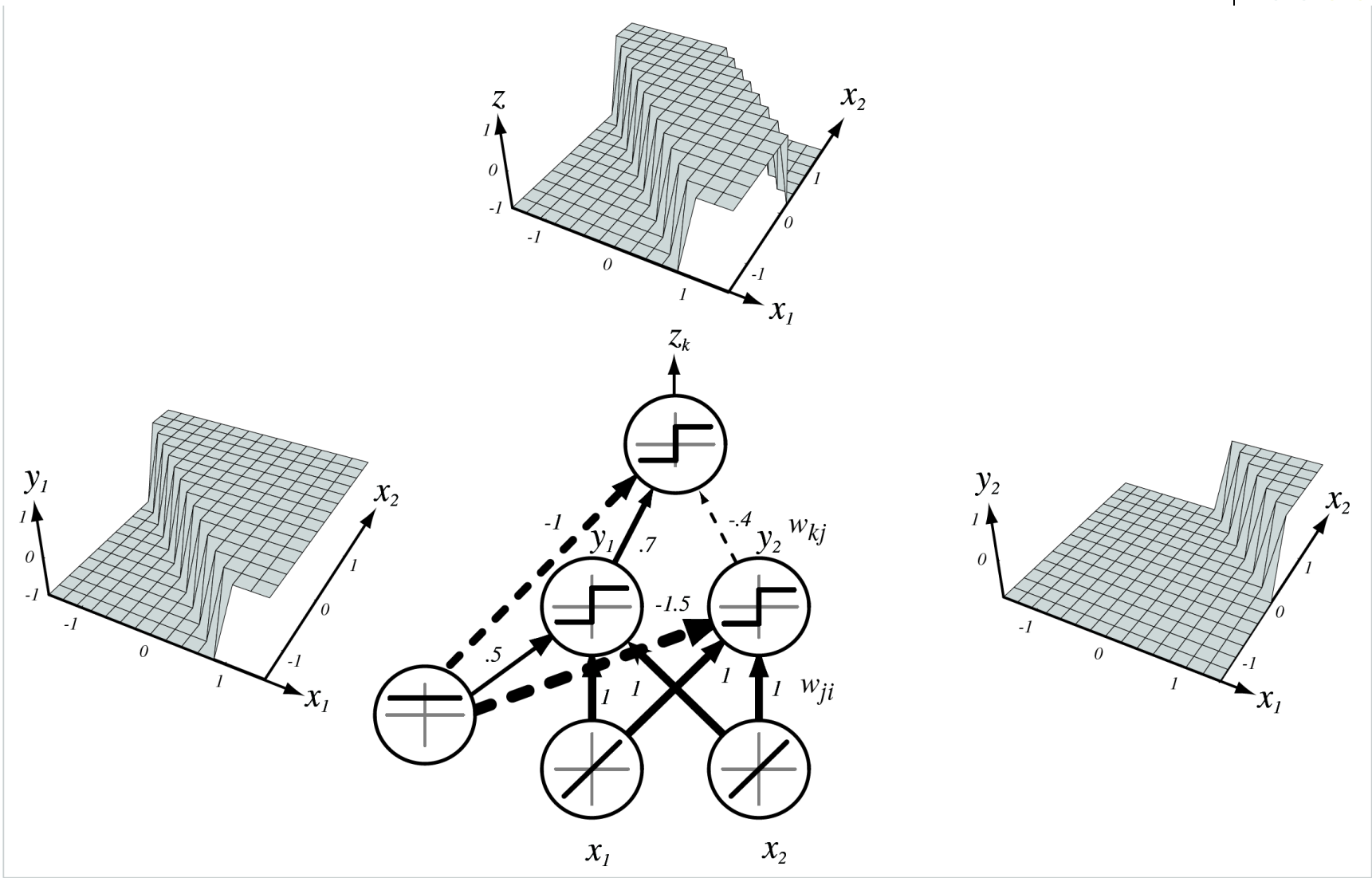




Needed a further level to combine the outputs of the two neurons



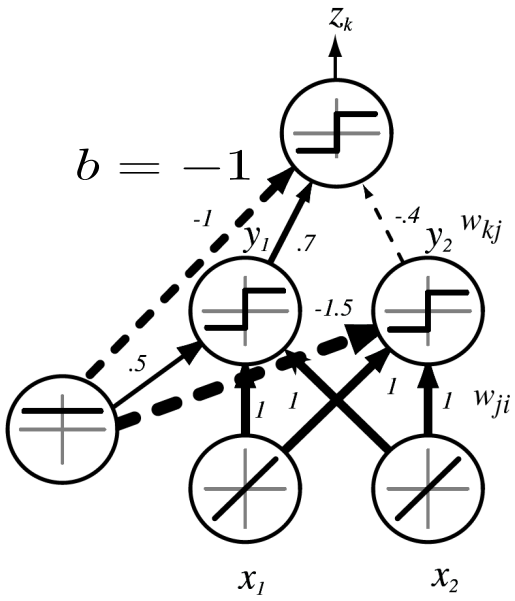
Neural Networks



Neural Networks

$$\mathbf{X} = \begin{array}{c} \begin{matrix} x_1 & x_2 \end{matrix} \\ \begin{bmatrix} -1 & -1 \\ -1 & +1 \\ +1 & -1 \\ +1 & +1 \end{bmatrix} \end{array} \quad \mathbf{Y} = \begin{bmatrix} -1 \\ +1 \\ +1 \\ -1 \end{bmatrix}$$

inputs labels



$$b = -1$$
$$\mathbf{w}^\top = \begin{bmatrix} +0.7 & -0.4 \end{bmatrix}$$

$$\mathbf{b} = \begin{bmatrix} +0.5 & -1.5 \end{bmatrix}$$
$$\mathbf{W} = \begin{bmatrix} +1 & +1 \\ +1 & +1 \end{bmatrix}$$

$$\mathbf{X} \cdot \mathbf{W} = \begin{bmatrix} -2 & -2 \\ 0 & 0 \\ 0 & 0 \\ +2 & +2 \end{bmatrix}$$

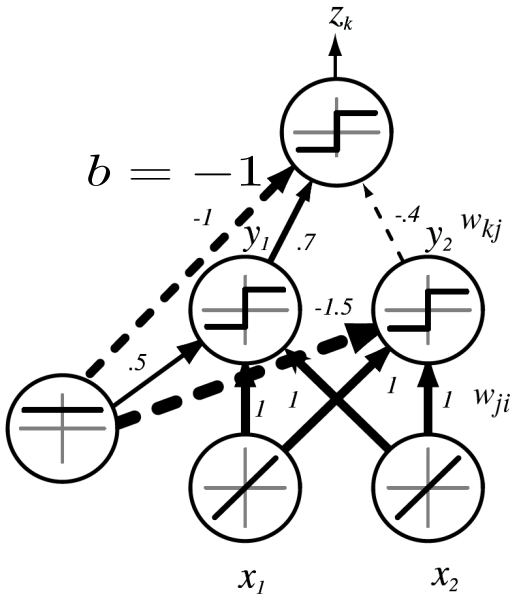
$$\mathbf{X} \cdot \mathbf{W} + \begin{bmatrix} \mathbf{b} \\ \mathbf{b} \\ \mathbf{b} \\ \mathbf{b} \end{bmatrix} = \begin{bmatrix} -1.5 & -3.5 \\ +0.5 & -1.5 \\ +0.5 & -1.5 \\ +2.5 & +0.5 \end{bmatrix}$$

$$\mathbf{G}\left(\mathbf{X} \cdot \mathbf{W} + \begin{bmatrix} \mathbf{b} \\ \mathbf{b} \\ \mathbf{b} \\ \mathbf{b} \end{bmatrix} \right) = \begin{array}{c} \begin{matrix} y_1 & y_2 \end{matrix} \\ \begin{bmatrix} -1 & -1 \\ +1 & -1 \\ +1 & -1 \\ +1 & +1 \end{bmatrix} \end{array}$$

Neural Networks

$$\mathbf{X} = \begin{array}{c} \begin{matrix} x_1 & x_2 \end{matrix} \\ \begin{bmatrix} -1 & -1 \\ -1 & +1 \\ +1 & -1 \\ +1 & +1 \end{bmatrix} \end{array} \quad \mathbf{Y} = \begin{bmatrix} -1 \\ +1 \\ +1 \\ -1 \end{bmatrix}$$

inputs **labels**



$$b = -1$$
$$\mathbf{w}^\top = \begin{bmatrix} +0.7 & -0.4 \end{bmatrix}$$

$$\mathbf{G}\left(\mathbf{X}\cdot\mathbf{W} + \begin{bmatrix} \mathbf{b} \\ \mathbf{b} \\ \mathbf{b} \\ \mathbf{b} \end{bmatrix}\right) = \begin{array}{c} \begin{matrix} y_1 & y_2 \end{matrix} \\ \begin{bmatrix} -1 & -1 \\ +1 & -1 \\ +1 & -1 \\ +1 & +1 \end{bmatrix} \end{array}$$

$$\mathbf{b} = \begin{bmatrix} +0.5 & -1.5 \end{bmatrix}$$
$$\mathbf{W} = \begin{bmatrix} +1 & +1 \\ +1 & +1 \end{bmatrix}$$

$$\begin{bmatrix} -1 & -1 \\ +1 & -1 \\ +1 & -1 \\ +1 & +1 \end{bmatrix} \cdot \mathbf{w} = \begin{bmatrix} -0.3 \\ +1.1 \\ +1.1 \\ +0.3 \end{bmatrix}$$

$$g\left(\begin{bmatrix} -1 & -1 \\ +1 & -1 \\ +1 & -1 \\ +1 & +1 \end{bmatrix} \cdot \mathbf{w} + \begin{bmatrix} b \\ b \\ b \\ b \end{bmatrix}\right) = \begin{array}{c} z \\ \begin{bmatrix} -1 \\ +1 \\ +1 \\ -1 \end{bmatrix} \end{array}$$

Neural networks

- Single hidden layer neural network
- Hidden layer pre-activation:

$$\mathbf{a}(\mathbf{x}) = \mathbf{b}^{(1)} + \mathbf{W}^{(1)}\mathbf{x}$$

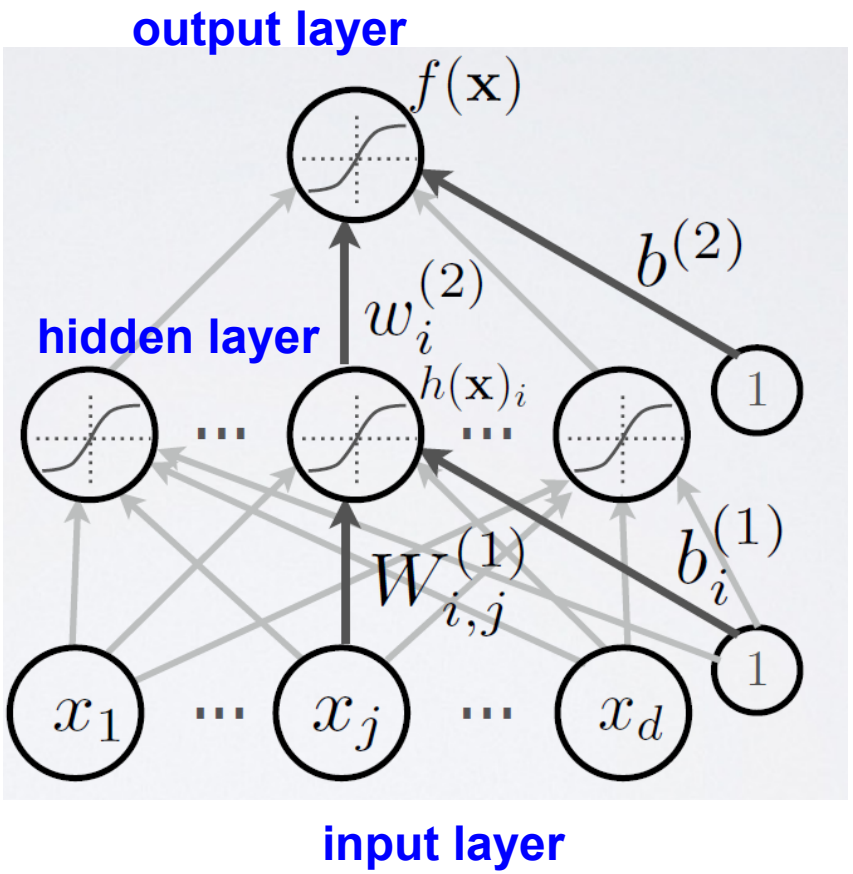
$$\left[a(\mathbf{x})_i = b_i^{(1)} + \sum_j W_{i,j}^{(1)} x_j \right]$$

- Hidden layer activation:

$$\mathbf{h}(\mathbf{x}) = \mathbf{g}(\mathbf{a}(\mathbf{x}))$$

- Output layer activation:

$$f(\mathbf{x}) = o\left(b^{(2)} + \mathbf{w}^{(2)\top}\mathbf{h}^{(1)}(\mathbf{x})\right)$$



Neural networks

- Multi-layer neural network
- Layer pre-activation:
  - for $k = 1 \dots l$

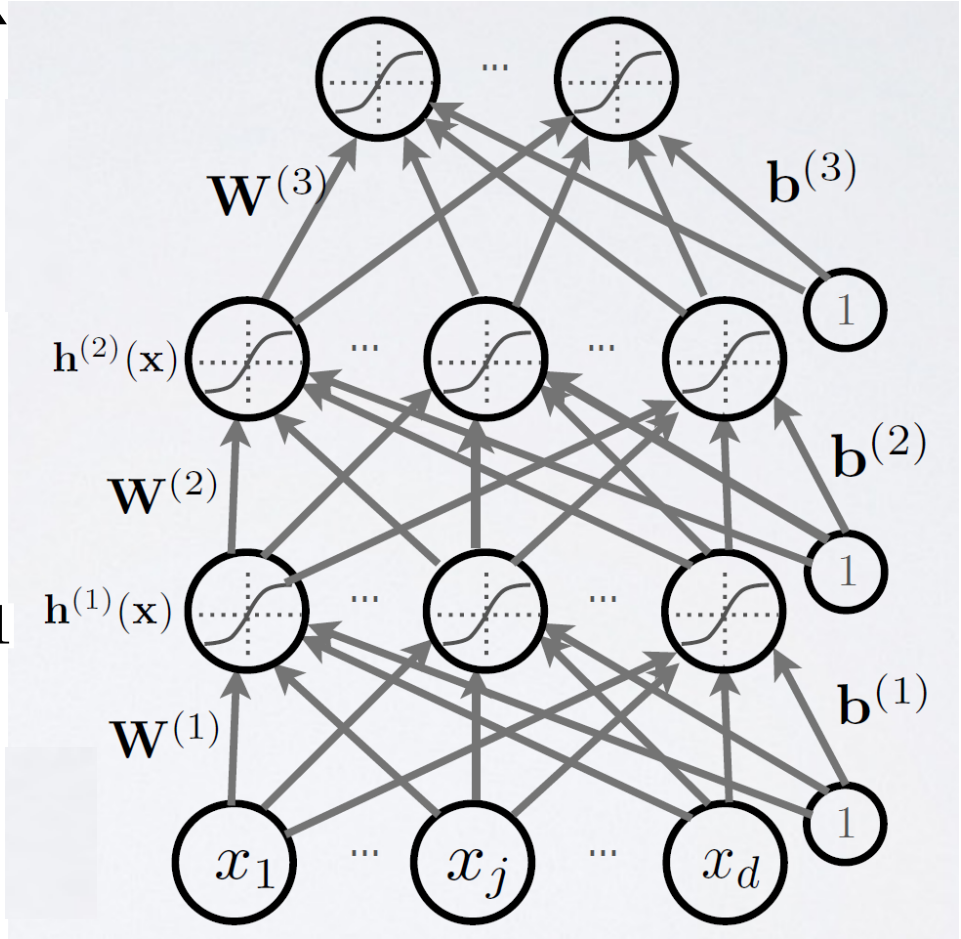
$$\mathbf{a}^{(k)}(\mathbf{x}) = \mathbf{b}^{(k)} + \mathbf{W}^{(k)}\mathbf{h}^{(k-1)}(\mathbf{x})$$

  - for $k = 0 \quad \mathbf{h}^{0}(\mathbf{x}) = \mathbf{x}$
- Hidden layer activation:

$$\mathbf{h}^{(k)}(\mathbf{x}) = \mathbf{g}(\mathbf{a}^{(k)}(\mathbf{x})) \; k = 1 \dots l-1$$

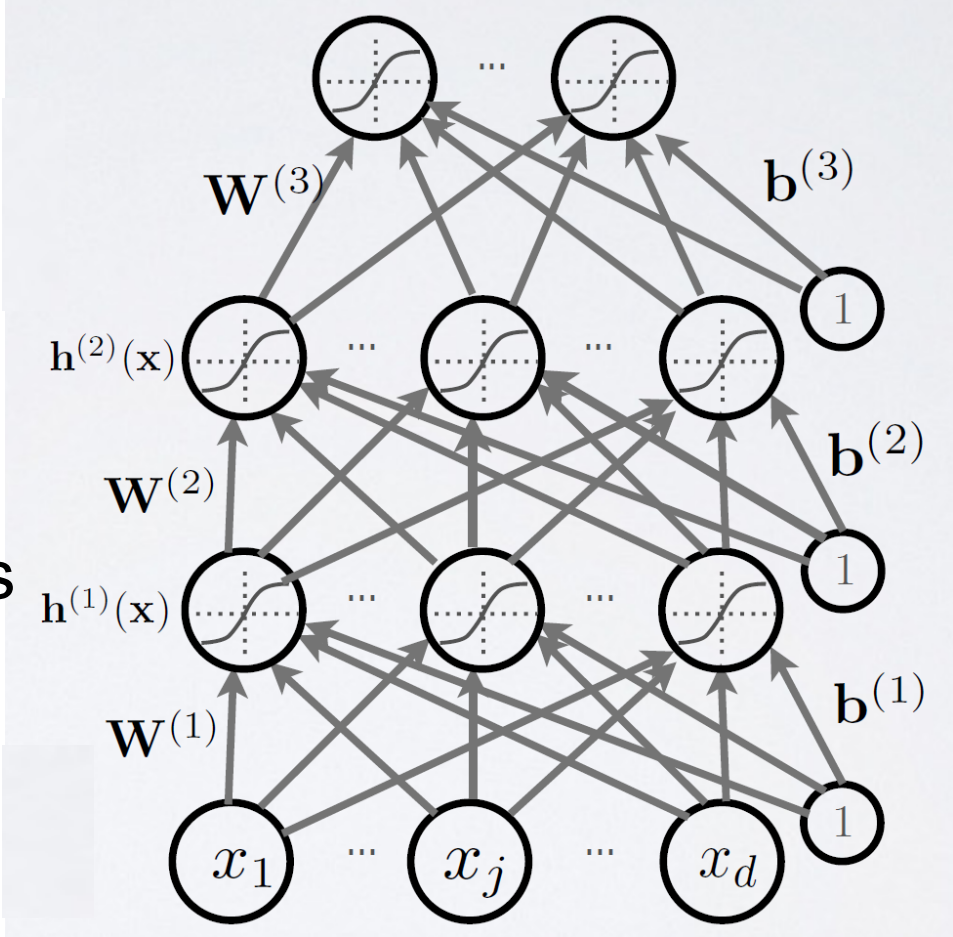
- Output layer activation:

$$\mathbf{h}^{(l)}(\mathbf{x}) = \mathbf{o}(\mathbf{a}^{(l)}(\mathbf{x})) = \mathbf{f}(\mathbf{x})$$

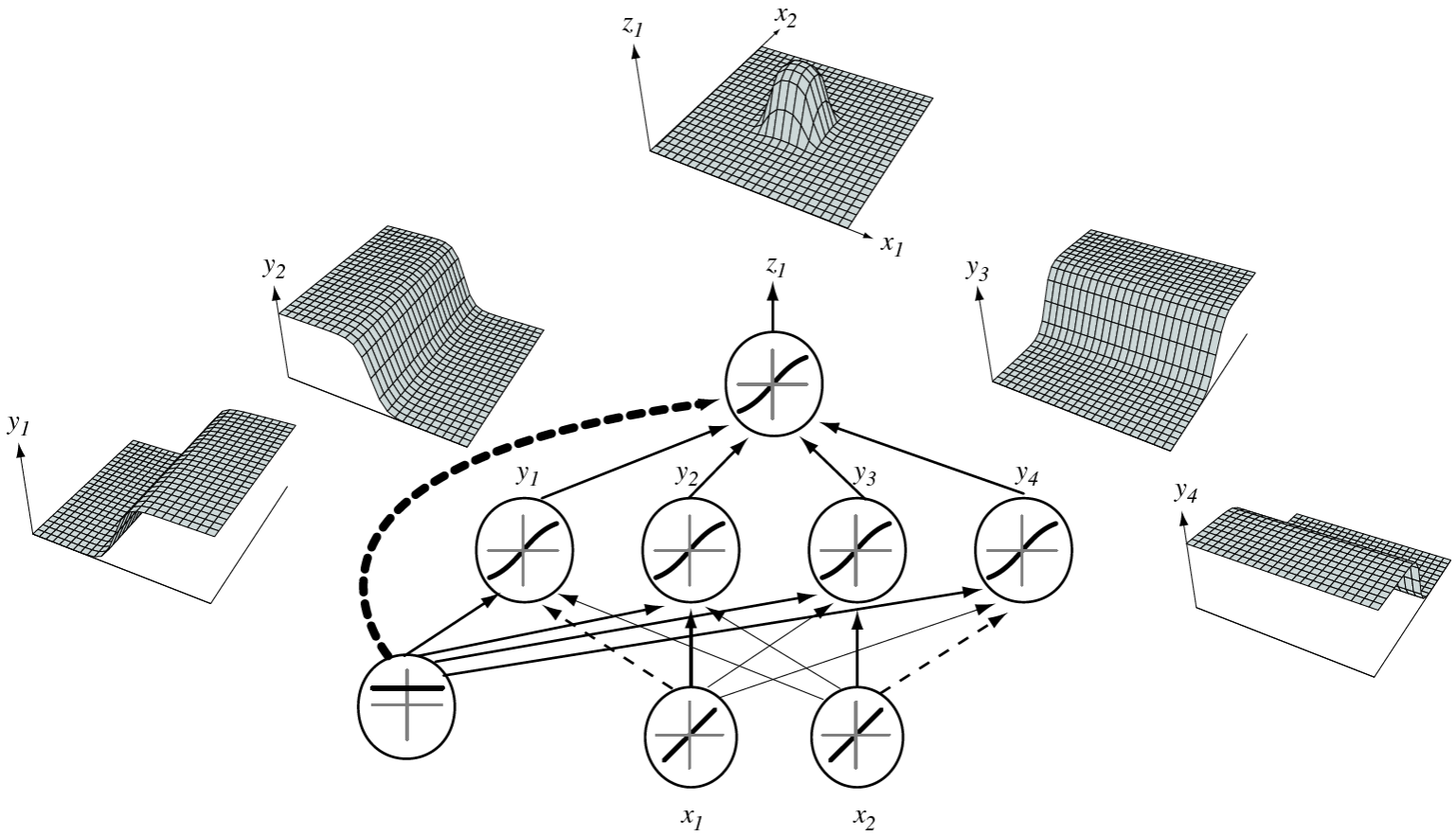


Neural networks

- Feedforward Networks
  - The information flows from the input through the intermediate layers to the output without feedback
- The number of layers gives the depth of the model
- The number of units in a layer is the width



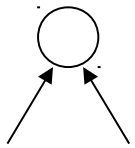
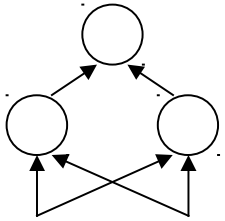
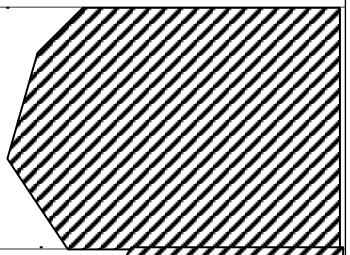
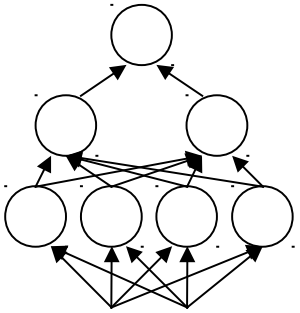
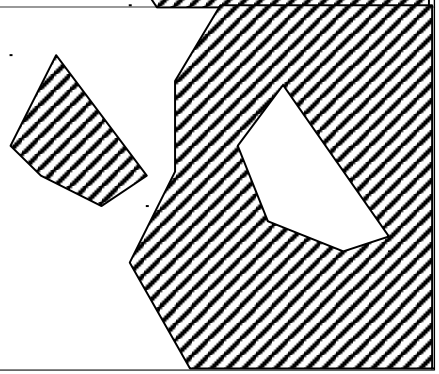
Capacity of neural network



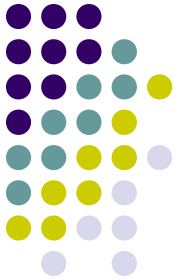
Capacity of neural network

- Universal approximation theorem (Hornik, 1991):
  - *a single hidden layer neural network with a linear output unit can approximate any continuous function arbitrarily well, given enough hidden units*
- The result applies for sigmoid, tanh and many other hidden layer activation functions
- Good result, but it doesn't help to define the learning algorithm that can find the parameter values

Capacity of neural network

Structure	Decision regions	General shapes
	Half spaces	
	Convex regions	
	Regions with arbitrary shape	

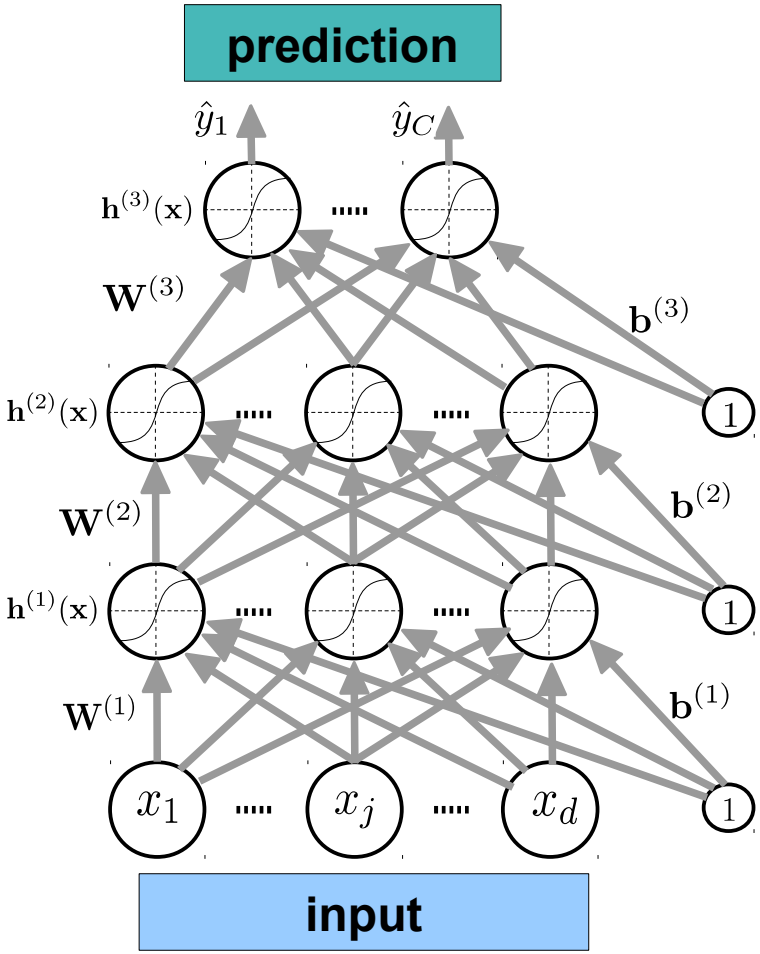
Neural network as a classifier



- Multi-class classification
  - Multiple output nodes (1 per class)
  - Coding 1-of-N (*sparse coding*)
  - Suitable to estimate the post probability $P(\omega_i|\mathbf{x})$
- To correctly estimate the postprob, the sum of the outputs must be 1
  - Softmax activation function at the output

$$softmax(a_j) = \frac{exp(a_j)}{\sum_{k=1}^{N} exp(a_k)}$$

Neural Networks: training

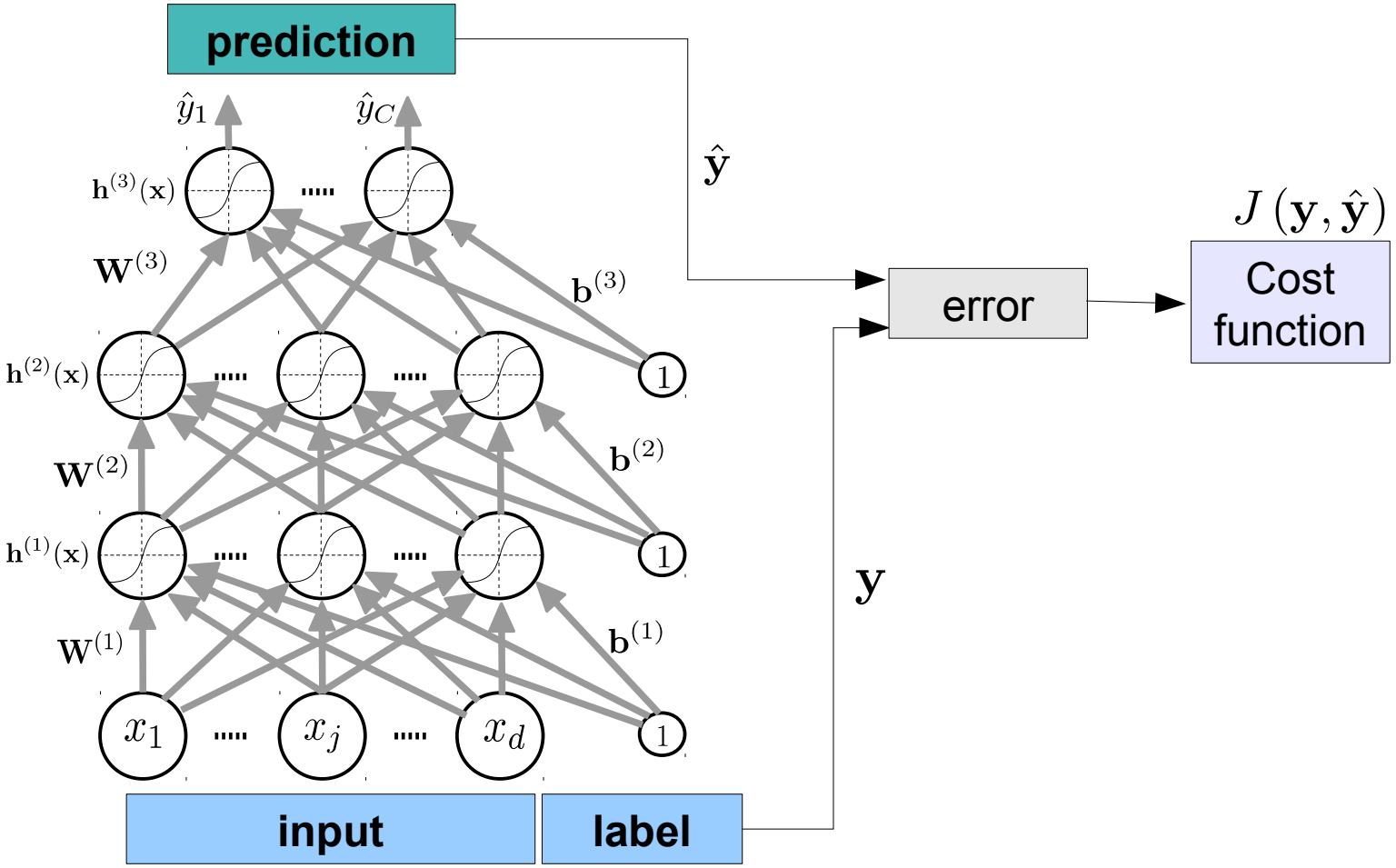


- The learning algorithms can automatically tune the weights and biases of a network of artificial neurons.
- This tuning happens in response to external stimuli, without direct intervention by a programmer.
- To this aim, we need a training set $\{\mathbf{x}, \mathbf{y}\}$
- The learning algorithm lets us find weights and biases so that the output $\hat{\mathbf{y}}$ from the network approximates the label $\mathbf{y}$ for all training inputs $\mathbf{x}$.

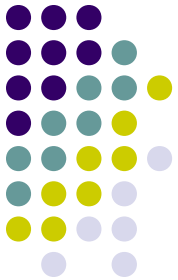
Neural Networks: training

- To quantify how well we're achieving this goal we define a *cost function* $J(\mathbf{y}, \hat{\mathbf{y}})$ that estimates the difference between the prediction and the label. Sometimes referred to as a *loss* or *objective function*.
- Obviously, the output $\hat{\mathbf{y}}$ depends on the input $\mathbf{x}$, on the weights $\mathbf{W}$ and on the biases $\mathbf{b}$

Neural Networks: training



Neural Networks: training



- Through $\hat{\mathbf{y}}$ the cost function J also depends on the parameters of the network and thus it is possible to devise a learning algorithm based on gradient descent aimed at minimizing the cost function.
- Some crucial differences with respect to the Perceptron for which the equation for weight update was:

$$w_i{}^{k+1} = w_i{}^{k} - \eta \nabla_{w_i} J(\mathbf{w})|_{\mathbf{w}^k}$$

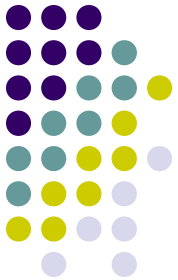
Neural Networks: training

- Problem:
  - The network has many levels with parameters in each level. How is it possible to calculate the gradient for each parameter?
- Solution:
  - Chain rule of calculus

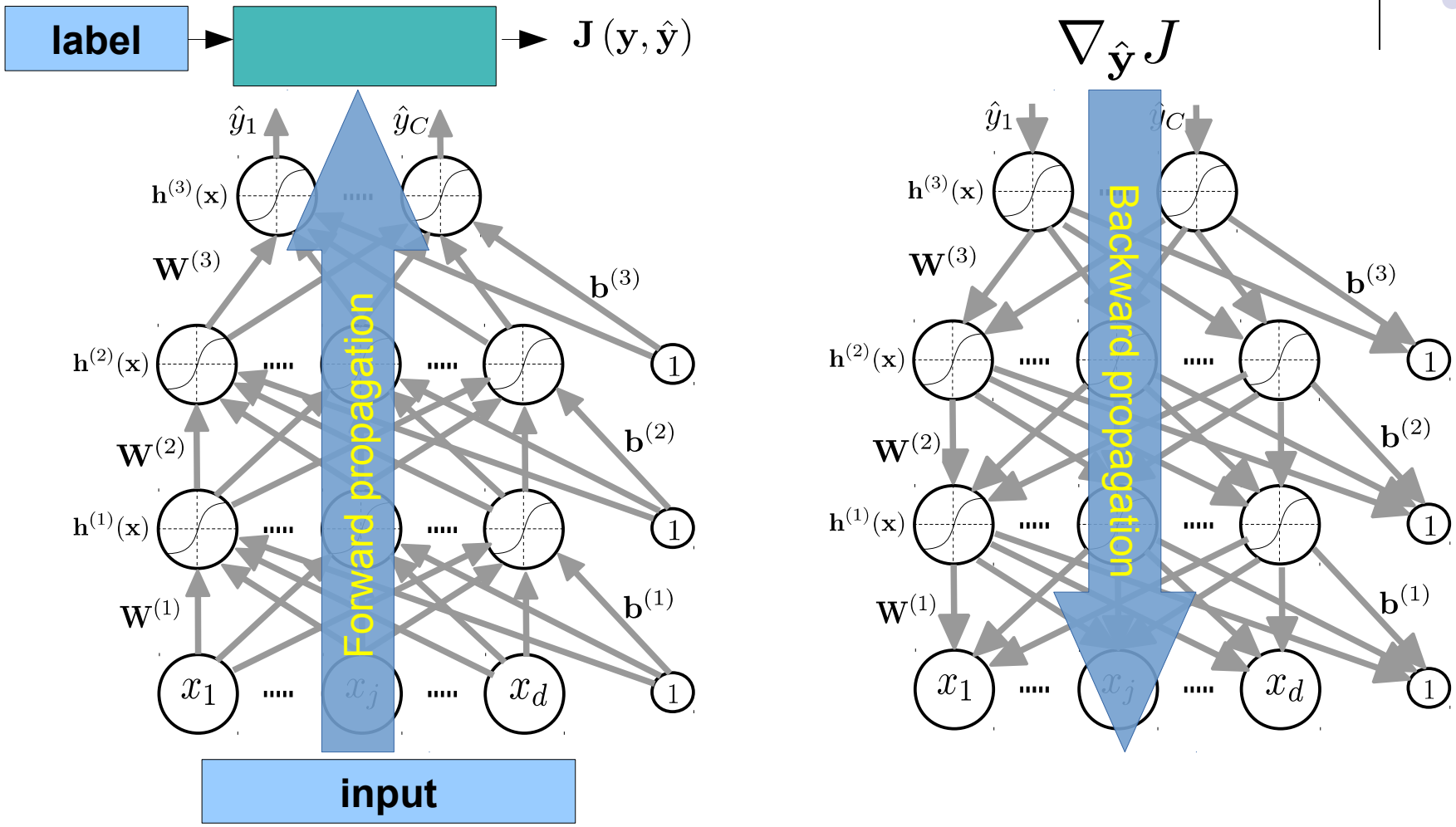
    Back-propagation algorithm

Back-propagation



- When an input $\mathbf{x}$ is fed to the network the information flows forward through the network to produce the output $\hat{\mathbf{y}}$. The inputs provide the initial information that then propagates up to the hidden units at each layer and finally produces $\hat{\mathbf{y}}$. This is called **forward propagation**.
- During training, forward propagation can continue onward until it produces a scalar cost $\mathbf{J}(\mathbf{y}, \hat{\mathbf{y}}) = \mathbf{J}(\mathbf{y}, \hat{\mathbf{y}}(\mathbf{x}, \mathbf{W}, \mathbf{b}))$
- The **back-propagation** algorithm (1986, Rumelhart et al.), often simply called **backprop**, allows the information from the cost to then flow backwards through the network, in order to compute the gradient

Back-propagation



Back-propagation

- Actually, back-propagation refers only to the method for computing the gradient for the various parameters of the network.
- In short, BP aims at *transporting* the gradient through the network.
- It is independent of the particular cost function used and of the particular method employed for calculating the gradient.

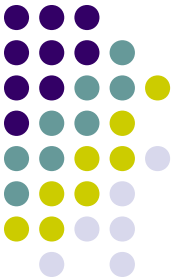
Chain rule of calculus

- Suppose we have two functions $f, g : R \rightarrow R$ with $y = g(x)$ and $z = f(y) = f(g(x))$
- Then we have:

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$$

Chain rule of calculus

Chain rule of calculus



- This extends to the multidimensional case
- Suppose $g : R^m \rightarrow R^n$ and $f : R^n \rightarrow R$
- If $\mathbf{y} = g(\mathbf{x})$ and $z = f(\mathbf{y})$ then we have:

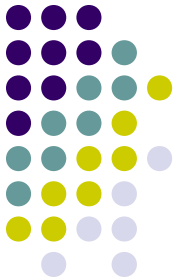
$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_i}$$

- In vector form: $\nabla_{\mathbf{x}} z = \left(\frac{\partial \mathbf{y}}{\partial \mathbf{x}} \right)^{\top} \nabla_{\mathbf{y}} z$

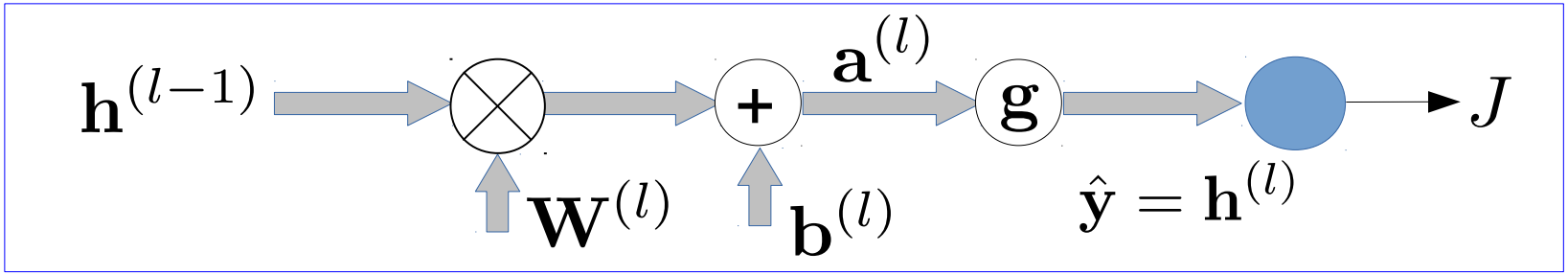
Jacobian matrix of g

gradient

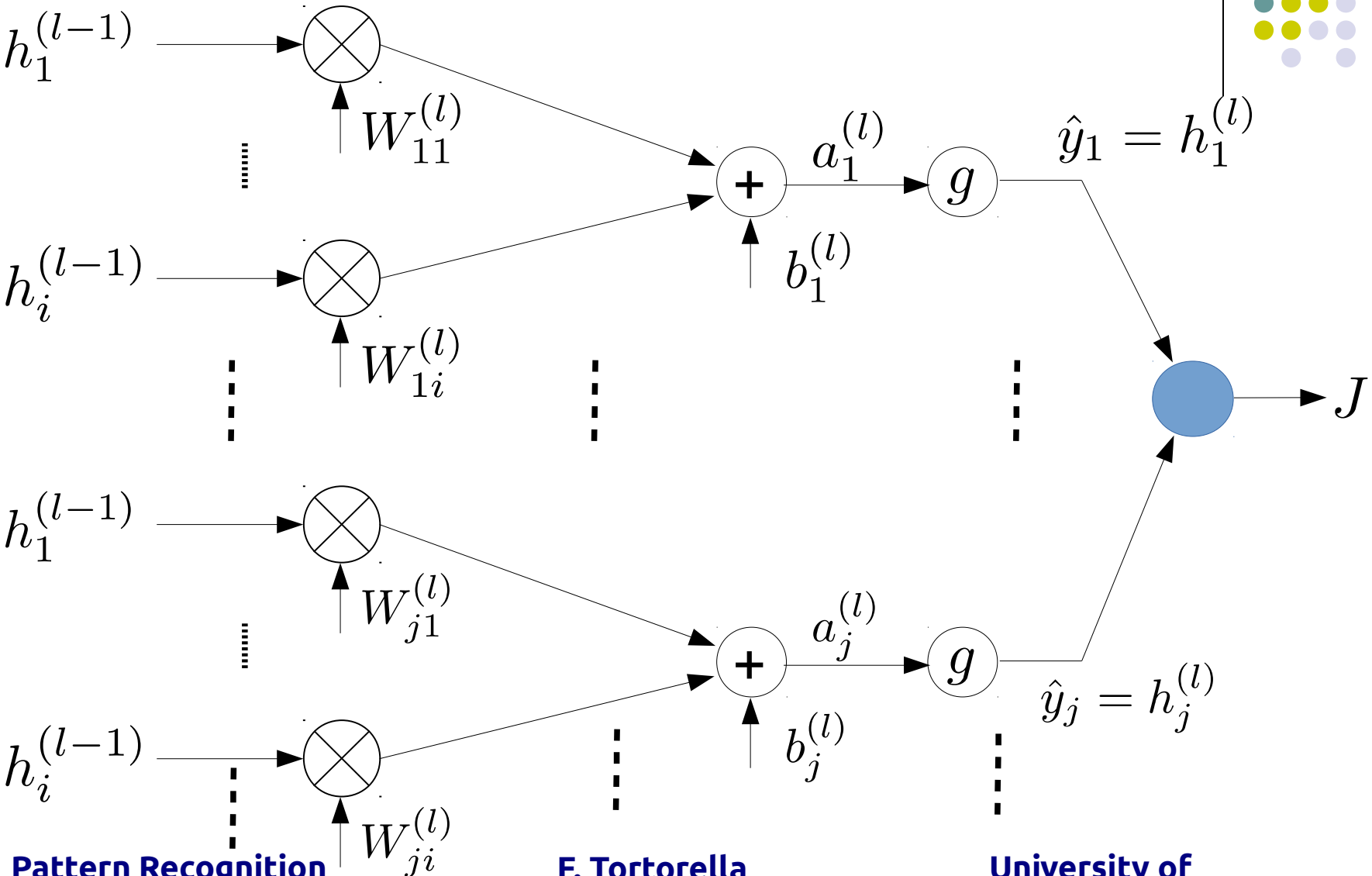
Back-propagation: the output (last) layer



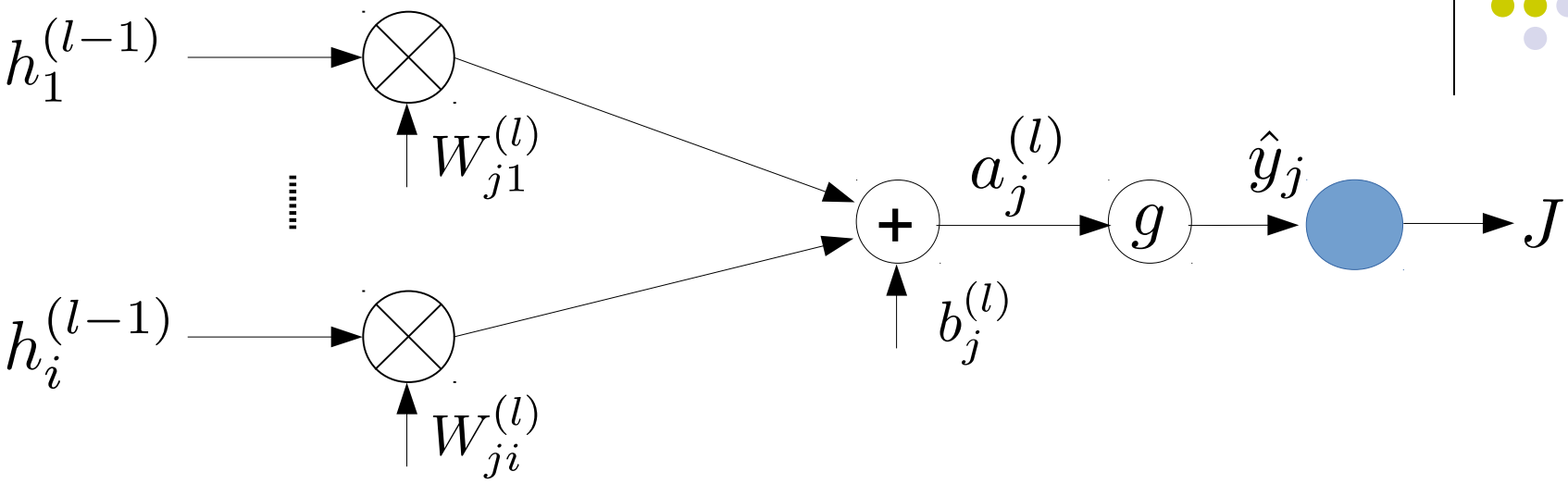
- Remember (k-th layer)
  - Layer preactivation $\mathbf{a}^{(k)}(\mathbf{x}) = \mathbf{b}^{(k)} + \mathbf{W}^{(k)}\mathbf{h}^{(k-1)}(\mathbf{x})$
  - Layer activation $\mathbf{h}^{(k)}(\mathbf{x}) = \mathbf{g}(\mathbf{a}^{(k)}(\mathbf{x}))$
- Output layer
  - Preactivation $\mathbf{a}^{(l)}(\mathbf{x}) = \mathbf{b}^{(l)} + \mathbf{W}^{(l)}\mathbf{h}^{(l-1)}(\mathbf{x})$
  - Activation $\hat{\mathbf{y}} = \mathbf{h}^{(l)}(\mathbf{x}) = \mathbf{g}(\mathbf{a}^{(l)}(\mathbf{x}))$



The output (last) layer

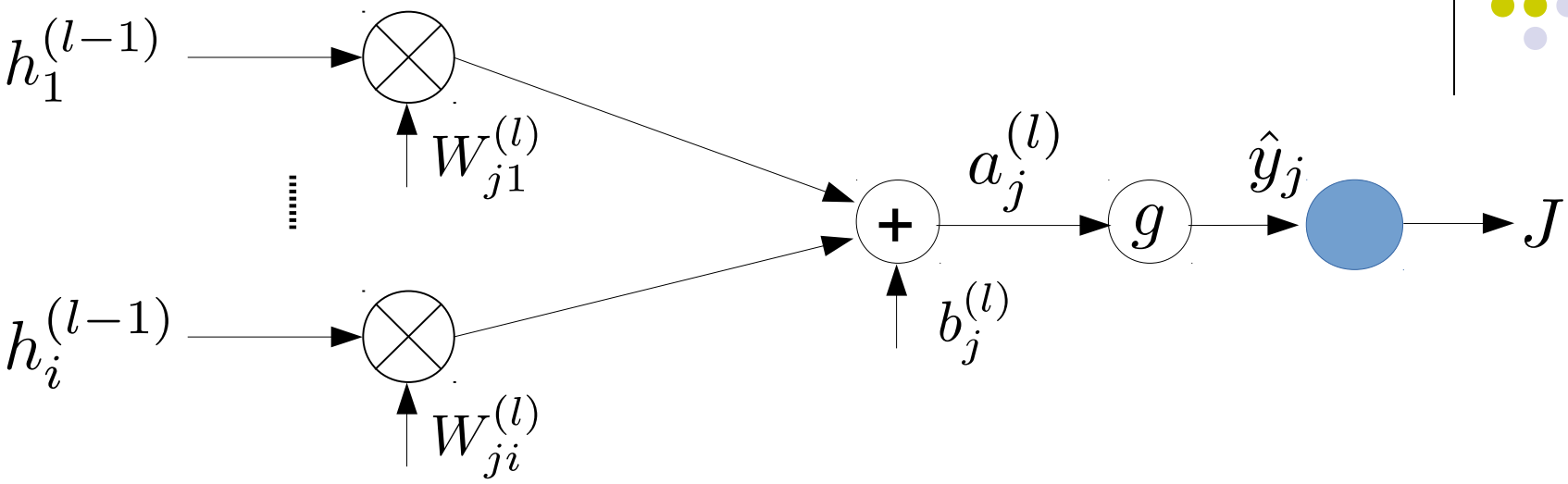


The output (last) layer



- We can immediately evaluate the gradient $\frac{\partial J}{\partial \hat{y}_i}$ once the cost function is chosen.

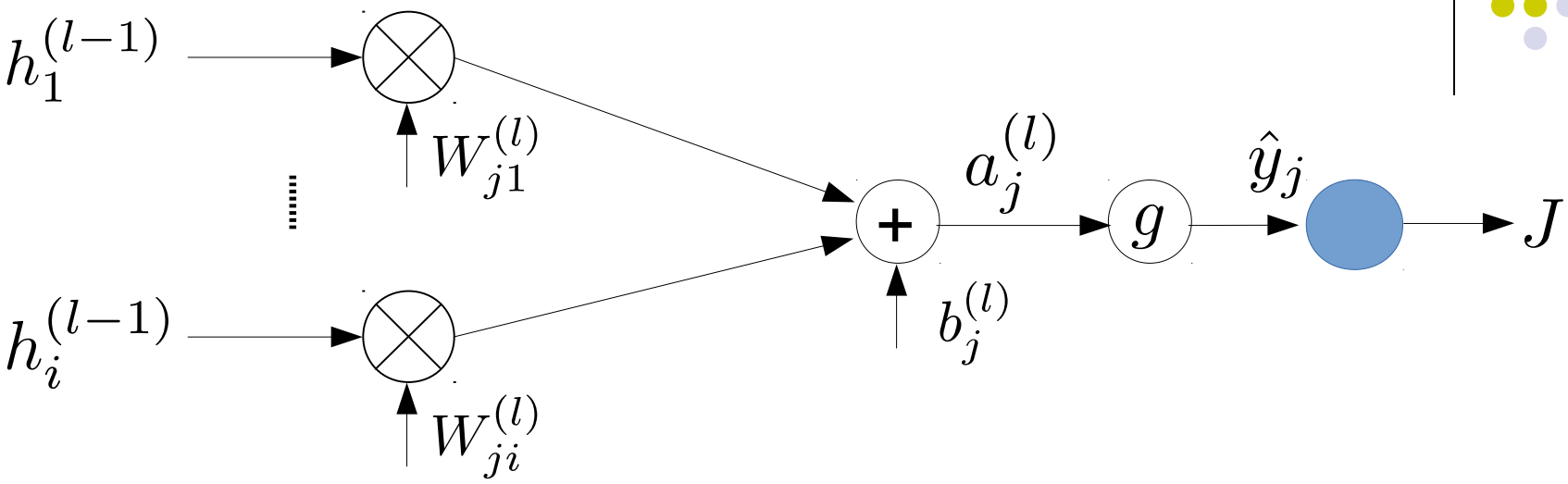
The output (last) layer



- Starting from $\frac{\partial J}{\partial \hat{y}_j}$ it is possible to evaluate the other gradients thanks to the chain rule:

$$\frac{\partial J}{\partial a_j^{(l)}} = \frac{\partial J}{\partial \hat{y}_j}\frac{d\hat{y}_j}{da_j^{(l)}} = \frac{\partial J}{\partial \hat{y}_j} g'\left(a_j^{(l)}\right)$$

The output (last) layer

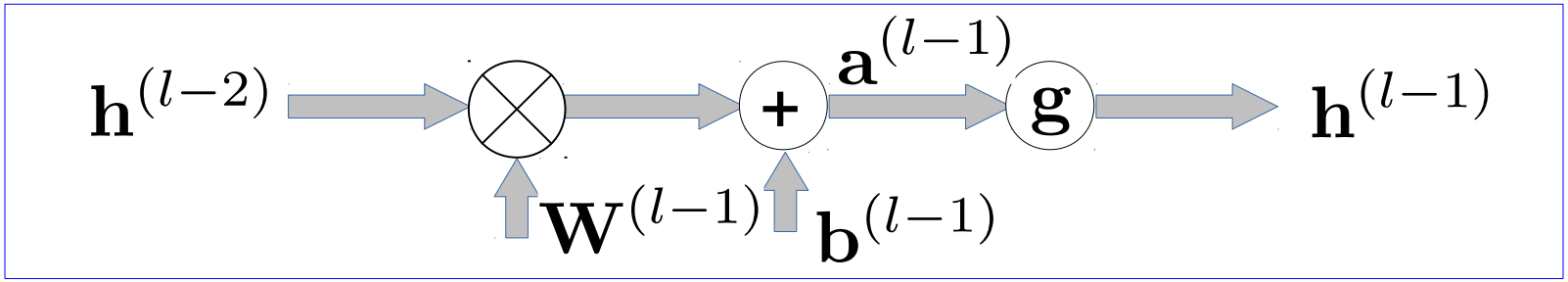


- Gradients for the parameters (weights and biases):

$$\frac{\partial J}{\partial b_j^{(l)}} = \frac{\partial J}{\partial \hat{y}_j} g'\left(a_j^{(l)}\right) \frac{\partial a_j^{(l)}}{\partial b_j^{(l)}} = \frac{\partial J}{\partial \hat{y}_j} g'\left(a_j^{(l)}\right) \quad \left[\frac{\partial a_j^{(l)}}{\partial b_j^{(l)}} = 1\right]$$

$$\frac{\partial J}{\partial W_{ji}^{(l)}} = \frac{\partial J}{\partial \hat{y}_j} g'\left(a_j^{(l)}\right) \frac{\partial a_j^{(l)}}{\partial W_{ji}^{(l)}} = \frac{\partial J}{\partial \hat{y}_j} g'\left(a_j^{(l)}\right) h_i^{(l-1)}$$

The internal layers

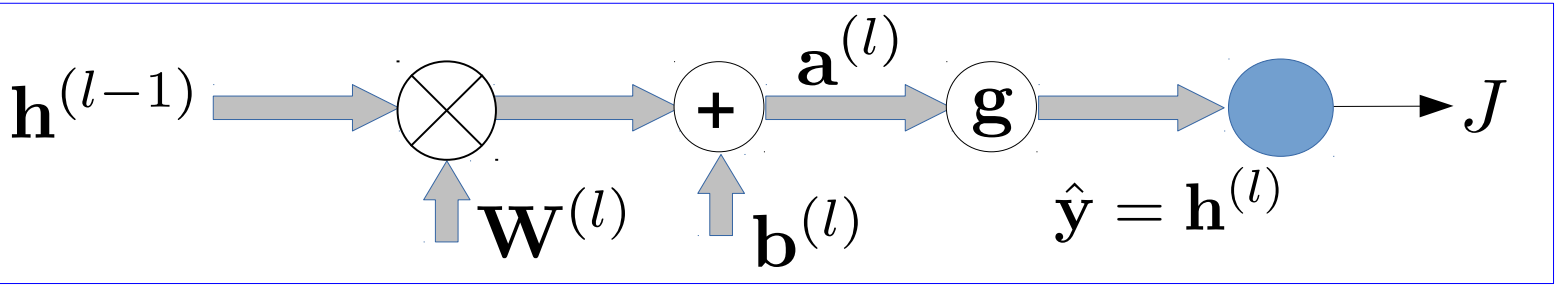


- In order to evaluate the gradients for the parameters of the internal layers, note that we could apply the same equations, once we have $\nabla_{\mathbf{h}^{(l-1)}} J$

$$\nabla_{\mathbf{b}^{(l-1)}} J = \nabla_{\mathbf{h}^{(l-1)}} J \odot g'\left(\mathbf{a}^{(l-1)}\right)$$

$$\nabla_{\mathbf{W}^{(l-1)}} J = \left[\nabla_{\mathbf{h}^{(l-1)}} J \odot g'\left(\mathbf{a}^{(l-1)}\right)\right] \mathbf{h}^{(l-2)\top}$$

The internal layers



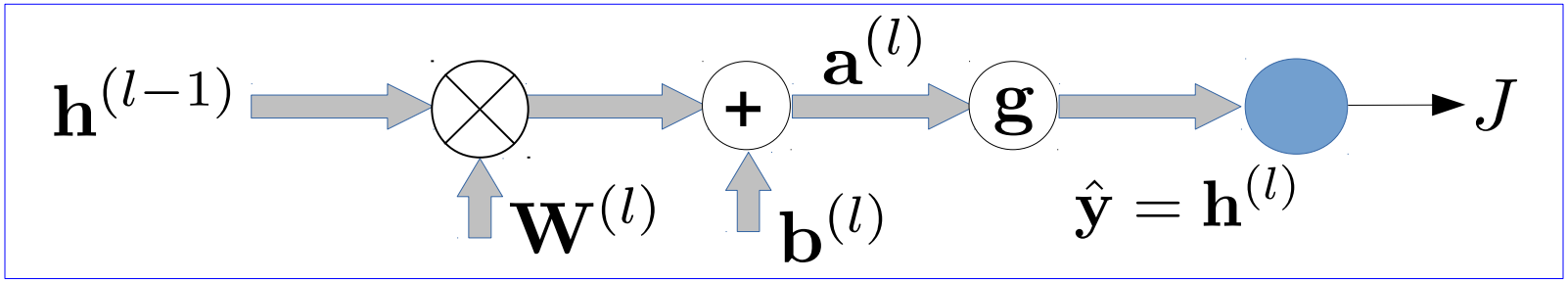
- To obtain $\nabla_{\mathbf{h}^{(l-1)}} J$ we can use the multidimensional chain rule:

$$\nabla_{\mathbf{h}^{(l-1)}} J = \left(\frac{\partial \mathbf{a}^{(l)}}{\partial \mathbf{h}^{(l-1)}} \right)^{\top} \nabla_{\mathbf{a}^{(l)}} J$$

- Since $\mathbf{a}^{(l)} = \mathbf{b}^{(l)} + \mathbf{W}^{(l)} \mathbf{h}^{(l-1)}$, we have $\frac{\partial \mathbf{a}^{(l)}}{\partial \mathbf{h}^{(l-1)}} = \mathbf{W}^{(l)}$ and thus:

$$\nabla_{\mathbf{h}^{(l-1)}} J = \left(\mathbf{W}^{(l)} \right)^{\top} \nabla_{\mathbf{a}^{(l)}} J$$

The output (last) layer



- In vector form:

$$\nabla_{\mathbf{a}^{(l)}} J = \nabla_{\hat{\mathbf{y}}} J \odot g'\left(\mathbf{a}^{(l)}\right)$$

$$\nabla_{\mathbf{b}^{(l)}} J = \nabla_{\hat{\mathbf{y}}} J \odot g'\left(\mathbf{a}^{(l)}\right)$$

$$\nabla_{\mathbf{W}^{(l)}} J = \left[\nabla_{\hat{\mathbf{y}}} J \odot g'\left(\mathbf{a}^{(l)}\right)\right] \mathbf{h}^{(l-1)\top}$$

Hadamard's Product

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \mathrm{B} = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

$$A \odot B = \begin{bmatrix} a_{11}b_{11} & a_{12}b_{12} \\ a_{21}b_{21} & a_{22}b_{22} \end{bmatrix}$$

Forward propagation

Require: Network depth, l

Require: $\boldsymbol{W}^{(i)}, i \in \{1, \ldots, l\}$, the weight matrices of the model

Require: $\boldsymbol{b}^{(i)}, i \in \{1, \ldots, l\}$, the bias parameters of the model

Require: $\boldsymbol{x}$, the input to process

Require: $\boldsymbol{y}$, the target output

$\boldsymbol{h}^{(0)} = \boldsymbol{x}$

for $k = 1, \ldots, l$ **do**

$\boldsymbol{a}^{(k)} = \boldsymbol{b}^{(k)} + \boldsymbol{W}^{(k)} \boldsymbol{h}^{(k-1)}$

$\boldsymbol{h}^{(k)} = f(\boldsymbol{a}^{(k)})$

end for

$\hat{\boldsymbol{y}} = \boldsymbol{h}^{(l)}$

$J = L(\hat{\boldsymbol{y}}, \boldsymbol{y}) + \lambda \Omega(\theta)$

Backward propagation

After the forward computation, compute the gradient on the output layer:

$\boldsymbol{g} \leftarrow \nabla_{\hat{\boldsymbol{y}}} J = \nabla_{\hat{\boldsymbol{y}}} L(\hat{\boldsymbol{y}}, \boldsymbol{y})$

for $k = l, l-1, \ldots, 1$ **do**

Convert the gradient on the layer's output into a gradient into the pre-nonlinearity activation (element-wise multiplication if f is element-wise):

$\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{a}^{(k)}} J = \boldsymbol{g} \odot f'(\boldsymbol{a}^{(k)})$

Compute gradients on weights and biases (including the regularization term, where needed):

$\nabla_{\boldsymbol{b}^{(k)}} J = \boldsymbol{g} + \lambda \nabla_{\boldsymbol{b}^{(k)}} \Omega(\theta)$

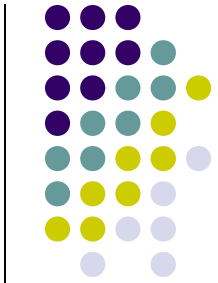
$\nabla_{\boldsymbol{W}^{(k)}} J = \boldsymbol{g}\, \boldsymbol{h}^{(k-1)\top} + \lambda \nabla_{\boldsymbol{W}^{(k)}} \Omega(\theta)$

Propagate the gradients w.r.t. the next lower-level hidden layer's activations:

$\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{h}^{(k-1)}} J = \boldsymbol{W}^{(k)\top}\, \boldsymbol{g}$

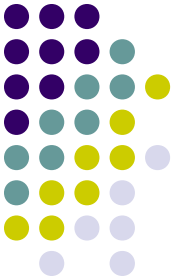
end for

Cost functions



- Different possible cost functions
- Mean Squared Error $$L(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{2}||\mathbf{y} - \hat{\mathbf{y}}||^2$$
  - Refers to a regression context
- Negative log-likelihood $$L(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_c I(y_c = 1) \ln \hat{y}_c$$
  - Sometimes referred to as cross-entropy

Stochastic Gradient Descent (SGD)



- When using a training set of m samples, the cost function is

$$J(\theta) = \frac{1}{m} \sum_{i=1}^{m} L(\mathbf{y}_i, \hat{\mathbf{y}}_i)$$

- Thus the gradient is

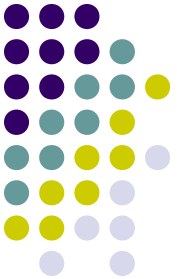
$$\nabla_\theta J = \frac{1}{m} \sum_{i=1}^{m} \nabla_\theta L(\mathbf{y}_i, \hat{\mathbf{y}}_i)$$

  - Too long time
  - Possible aggregation effects

Stochastic Gradient Descent (SGD)

- Gradient is an expectation
- May be approximately estimated using a small set of examples
- At each step of the learning process we can sample a **minibatch** of examples drawn uniformly from the training set
- The minibatch size m' ranges from one to a few hundred.
- It is usually held fixed as m grows

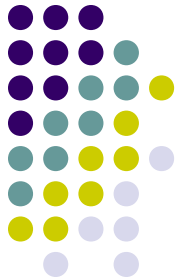
Stochastic Gradient Descent (SGD)



- The estimate of the gradient is:

$$\nabla_\theta J = \frac{1}{m'} \sum_{i=1}^{m'} \nabla_\theta L(\mathbf{y}_i, \hat{\mathbf{y}}_i)$$

Regularization

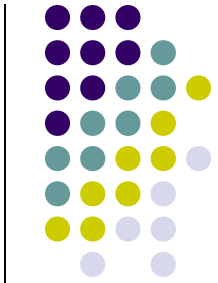


- To improve the generalization capability add a regularization term aimed at containing the size of the weights
- E.g. L2 regularization

$$\Omega(\theta) = \sum_k \sum_i \sum_j (W_{ij}^{(k)})^2 = \sum_k ||\mathbf{W}^{(k)}||^2$$

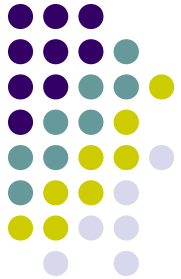
$$\nabla_{\mathbf{W}^{(k)}} \Omega(\theta) = 2\mathbf{W}^{(k)}$$

Inizialization



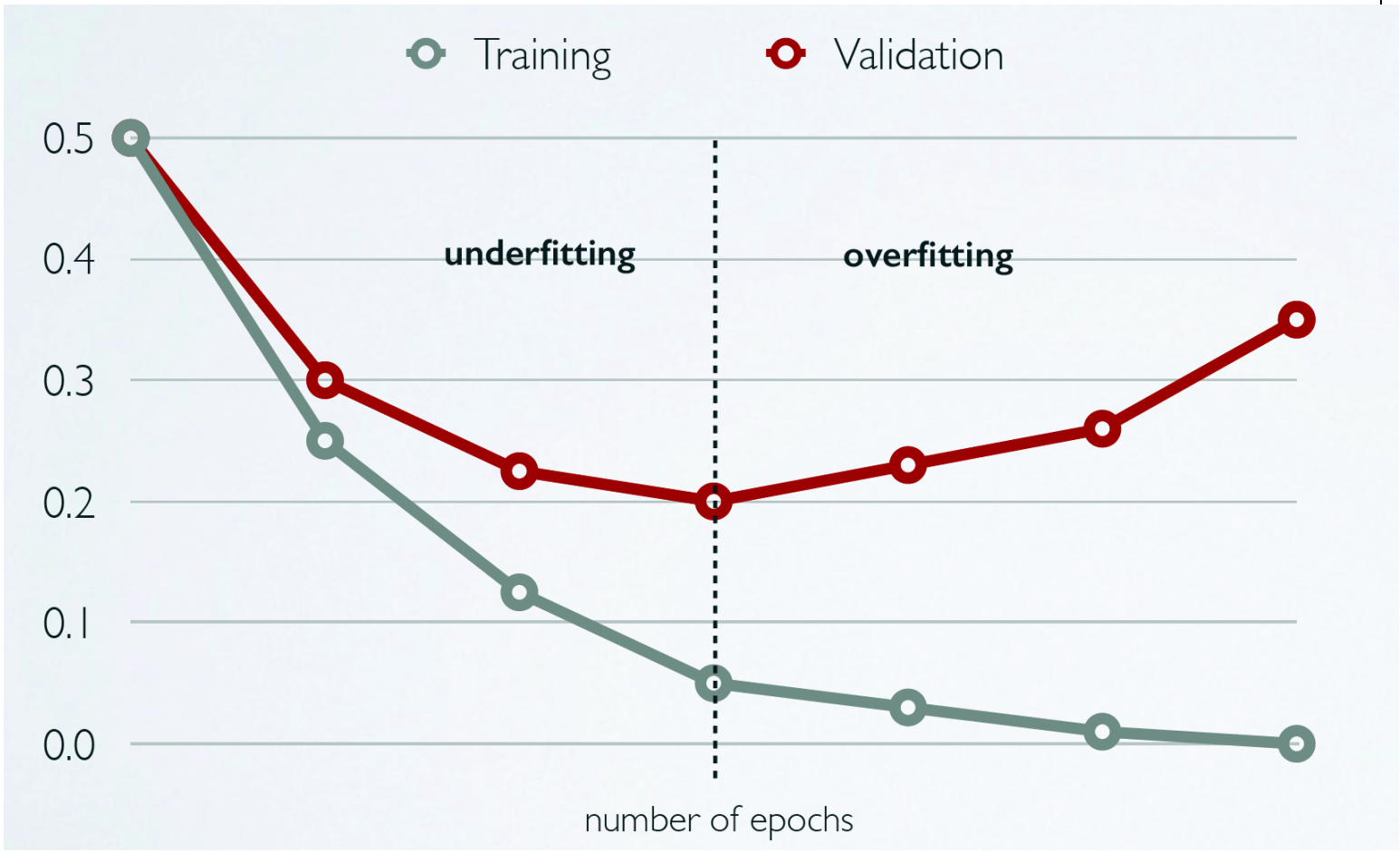
- Biases
  - Initialize all to 0
- Weights
  - can't initialize weights to 0
  - can't initialize weights to the same value
    - Need break simmetries
  - Rule of thumb
    - Sample $W_{ij}^{(k)}$ from $U[-b, b]$ with $b = \frac{\sqrt{6}}{\sqrt{H_k + H_{k-1}}}$

Neural Networks: training Model/hyperparam. selection



- Part the data in:
  - Training set – to train the model
  - Validation set – to select model/hyperparameters
  - Test set – to estimate the generalization error
- Model/hyperparameter selection
  - Grid search
  - Random search
    - Sample indipendently each hyperparameter
  - Use validation set to select the best choice

Neural Networks: training
Early stopping



Neural Networks: training normalization of data, decaying learning rate

- Normalize the real-valued data
  - Each feature is normalized so as to have zero mean and unitary standard deviation
- Decaying the learning rate
  - as we get closer to the optimum, makes sense to take smaller update steps
  - (i) start with large learning rate (e.g. 0.1)
  - (ii) maintain until validation error stops improving
  - (iii) divide learning rate by 2 and go back to (ii)

Deep Networks

- Networks with a large number of layers (deep networks) very effective for difficult problems.
- Training can be hard because of
  - Underfitting
  - Overfitting

Deep Networks Underfitting

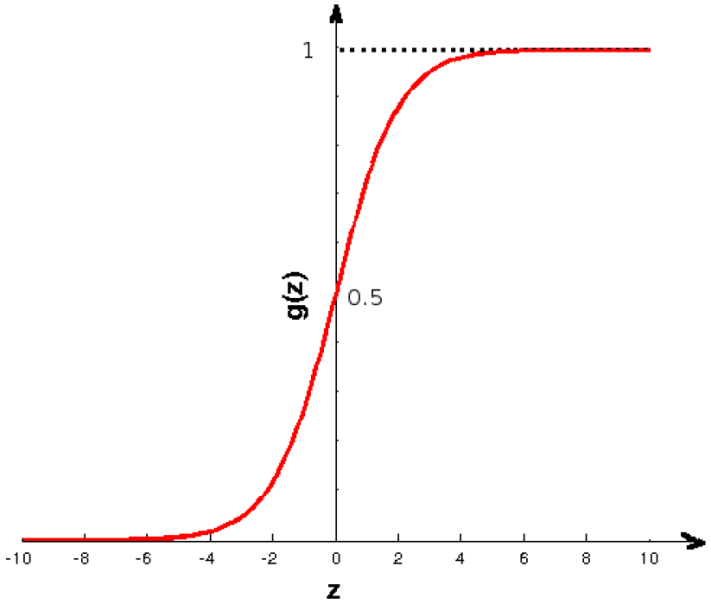
- In the backpropagation algorithm to compute the gradient for the parameters in one layer, we typically multiply the gradient from the layer above with the derivative of the activation function.

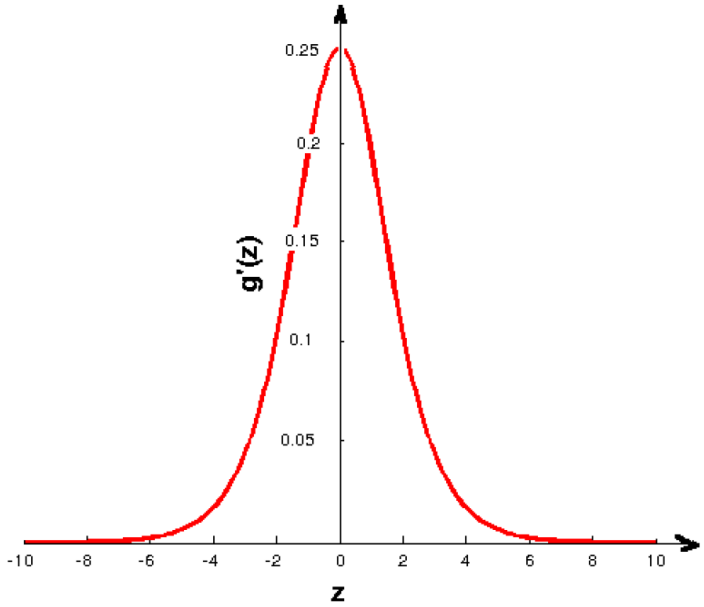
  E.g. $\nabla_{\mathbf{W}^{(l-1)}} J = \left[\nabla_{\mathbf{h}^{(l-1)}} J \odot g'\left(\mathbf{a}^{(l-1)}\right)\right] \mathbf{h}^{(l-2)\top}$

- This can give raise to the *vanishing gradient problem*

Deep Networks vanishing gradient problem

- Sigmoid activation function and its derivative

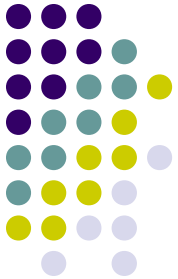




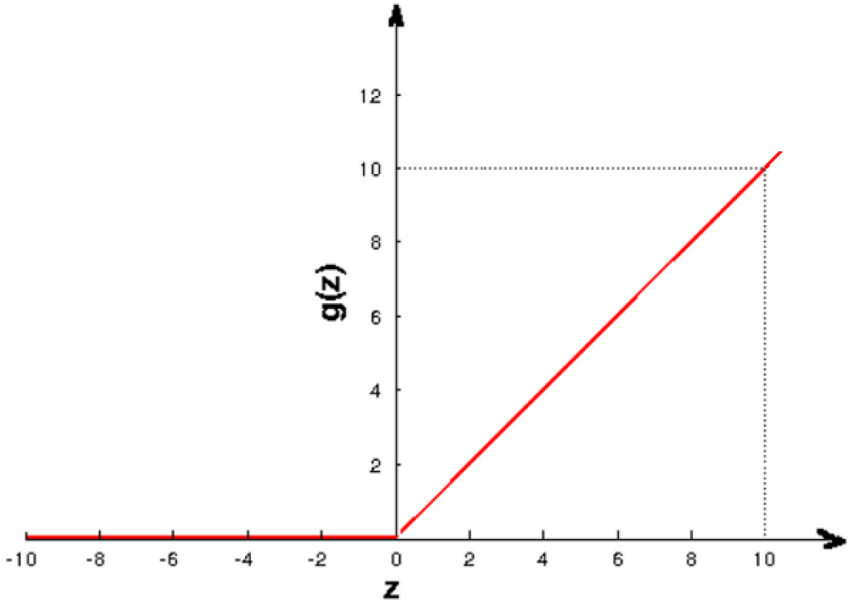
Deep Networks vanishing gradient problem

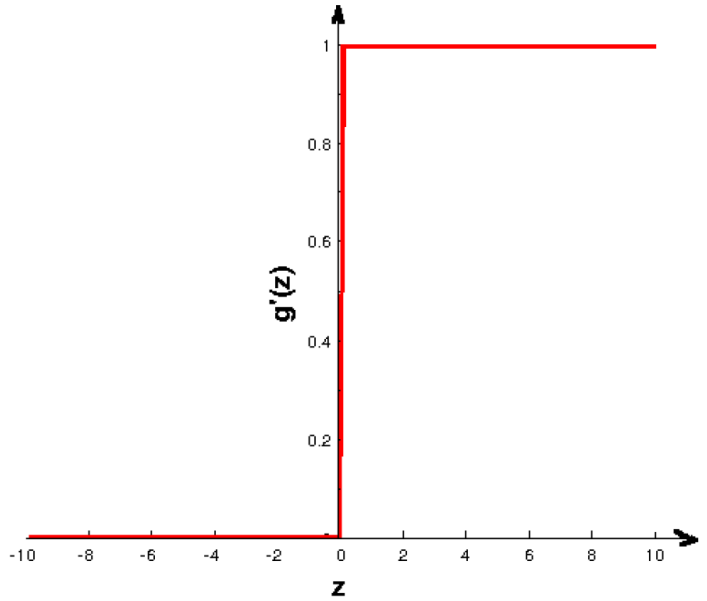
- Since the derivative of the sigmoid function has very small values (near zero) everywhere except for when the input has values of 0, the lower layers will likely to have smaller gradients in terms of magnitude, compared to the higher layers.
- This makes it difficult to change the parameters of the neural networks with stochastic gradient descent.

Deep Networks vanishing gradient problem

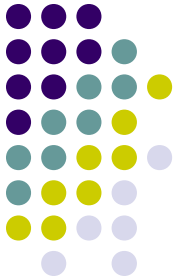


- This problem can be addressed by the use of rectified linear activation function. In this case the derivative can have many nonzero values.





Deep Networks Overfitting



- Deep neural networks have a lot of parameters (high capacity) and thus tend to overfit, especially when the training set is small.
- To overcome overfitting, there are many strategies:
  - Penalize the weights by adding a penalty on the norm of the weights in the network to the objective function J
  - Use unlabeled data to train a different network known as an autoencoder and then use the weights to initialize the network (pre-training)
  - Use dropout

Deep Networks
Unsupervised pre-training

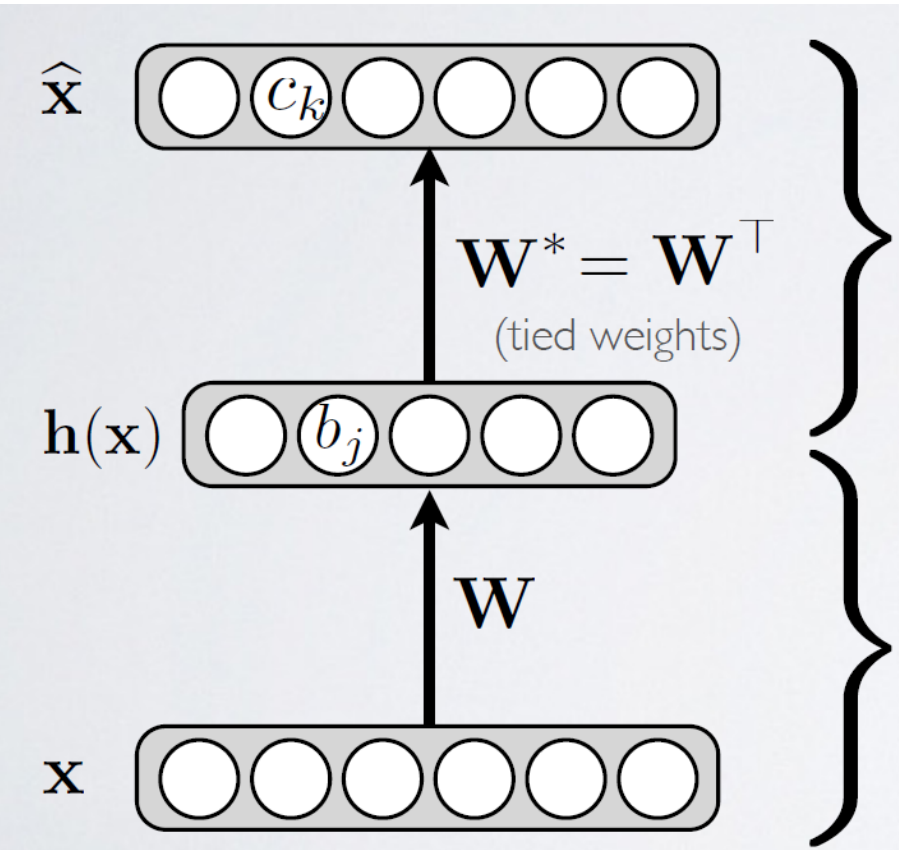
- Initialize hidden layers using unsupervised learning.
  - force network to represent latent structure of input distribution
  - encourage hidden layers to encode that structure

This is a character

This is not a character

Autoencoder

- Feed-forward neural network trained to reproduce its input at the output layer



Decoder

$$\hat{\mathbf{x}} = o(\hat{\mathbf{a}}(\mathbf{x})) = \underbrace{\text{sigm}}_{\text{for binary inputs}}(\mathbf{c} + \mathbf{W}^*\mathbf{h}(\mathbf{x}))$$

Encoder

$$\mathbf{h}(\mathbf{x}) = g(\mathbf{a}(\mathbf{x})) = \text{sigm}(\mathbf{b} + \mathbf{W}\mathbf{x})$$

Autoencoder

- Loss function
  - Binary inputs (cross-entropy)

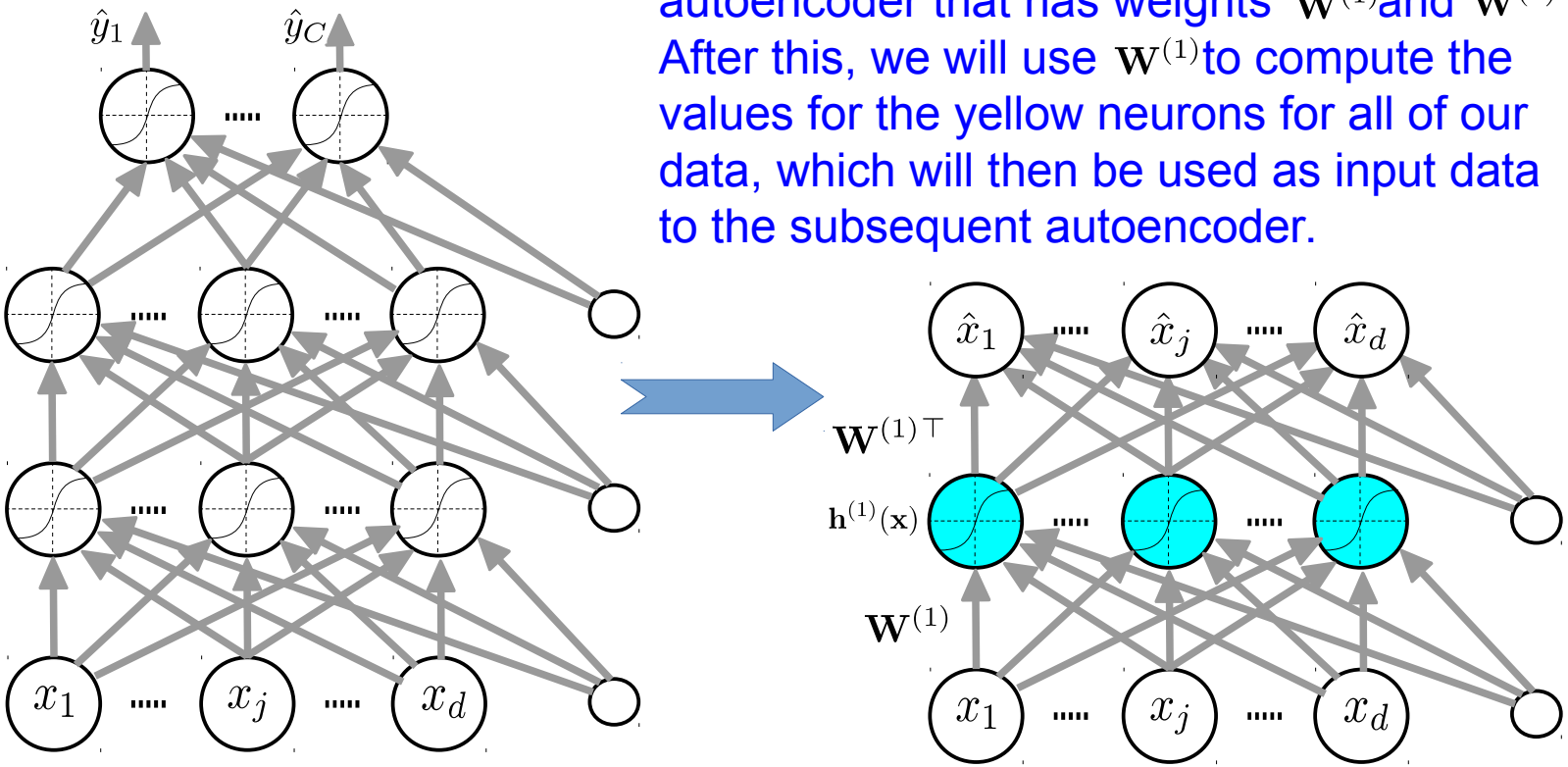
$$l(f(\mathbf{x})) = -\sum_k \left(x_k \log(\hat{x}_k) + (1 - x_k) \log(1 - \hat{x}_k)\right)$$

  - Real-valued inputs

$$l(f(\mathbf{x})) = \frac{1}{2} \sum_k (\hat{x}_k - x_k)^2$$

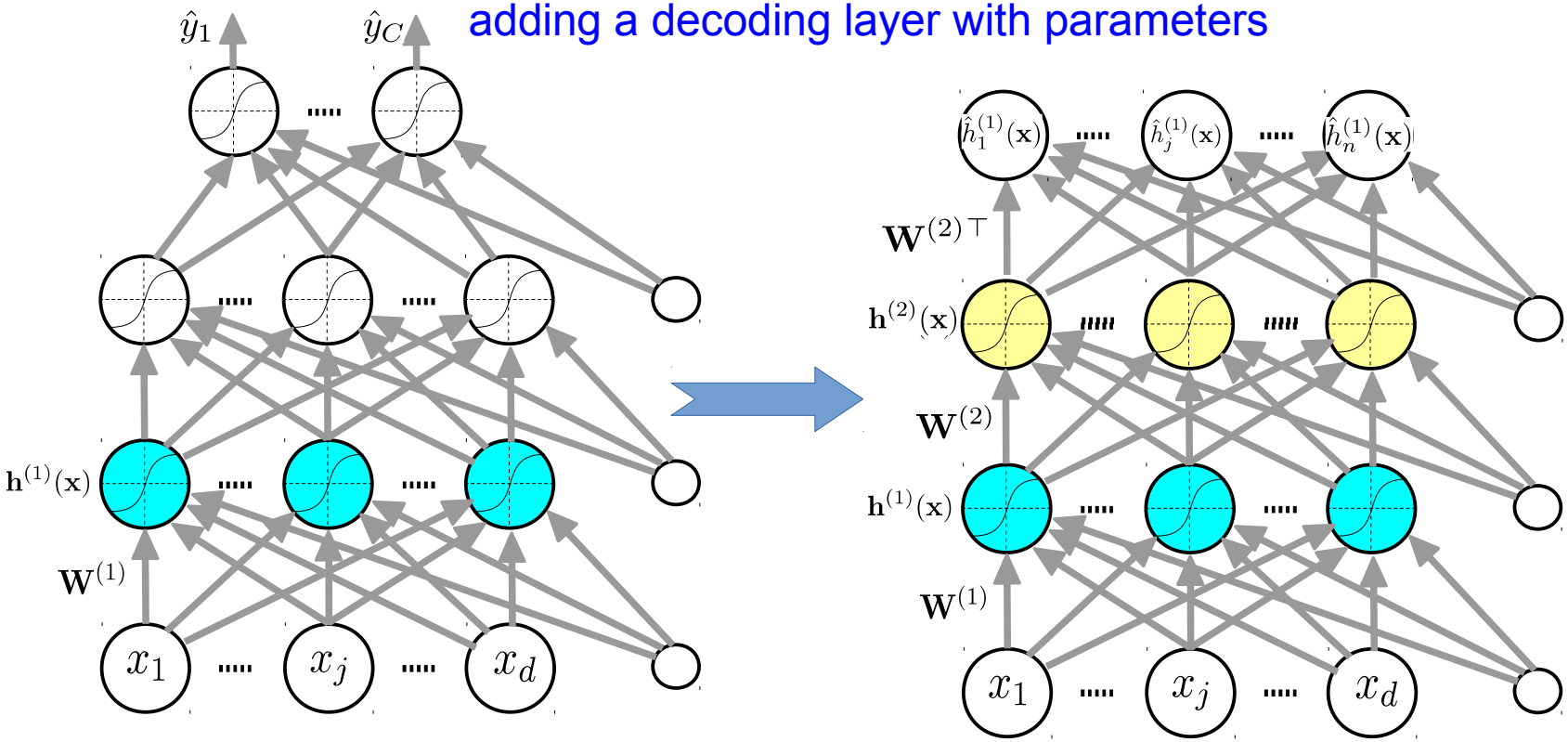
Unsupervised pre-training

To train the cyan neurons, we will train an autoencoder that has weights $\mathbf{W}^{(1)}$ and $\mathbf{W}^{(1)\top}$ After this, we will use $\mathbf{W}^{(1)}$ to compute the values for the yellow neurons for all of our data, which will then be used as input data to the subsequent autoencoder.

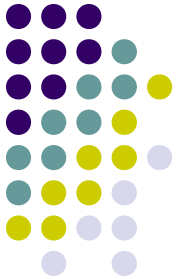


Unsupervised pre-training

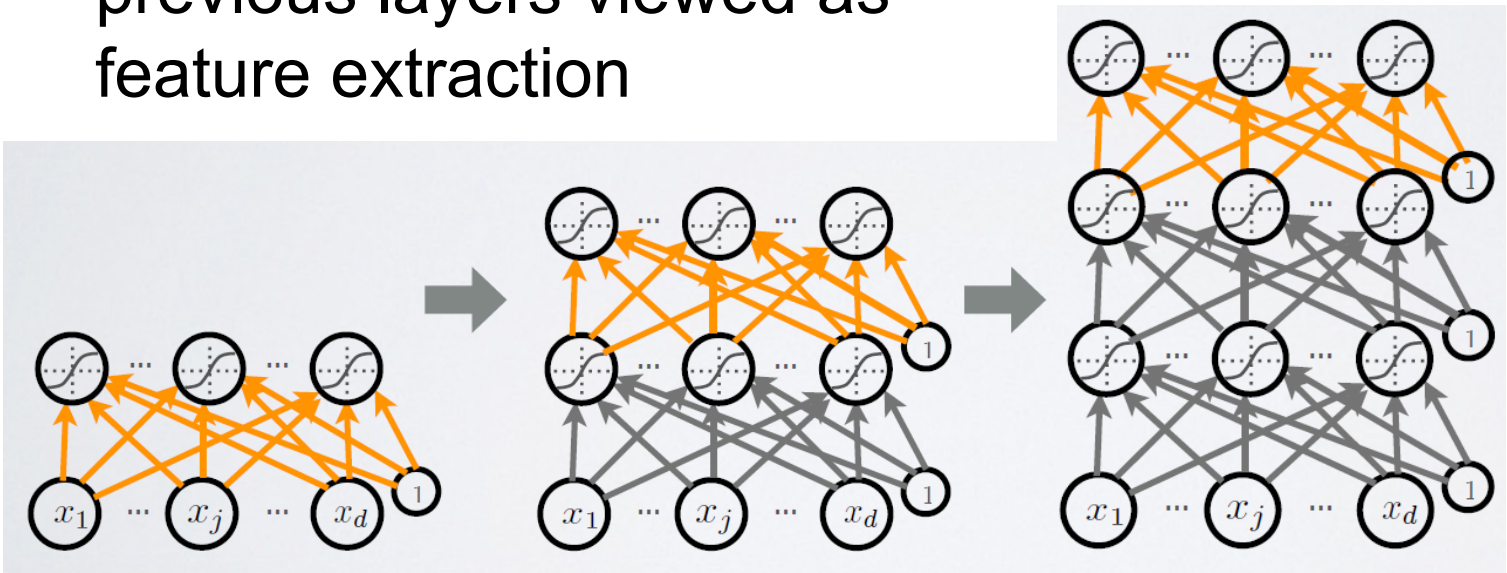
The parameters of the decoding process $\mathbf{W}^{(1)\top}$ will be discarded. The subsequent autoencoder uses the values for the cyan neurons as inputs, and trains an autoencoder to predict those values by adding a decoding layer with parameters



Unsupervised pre-training

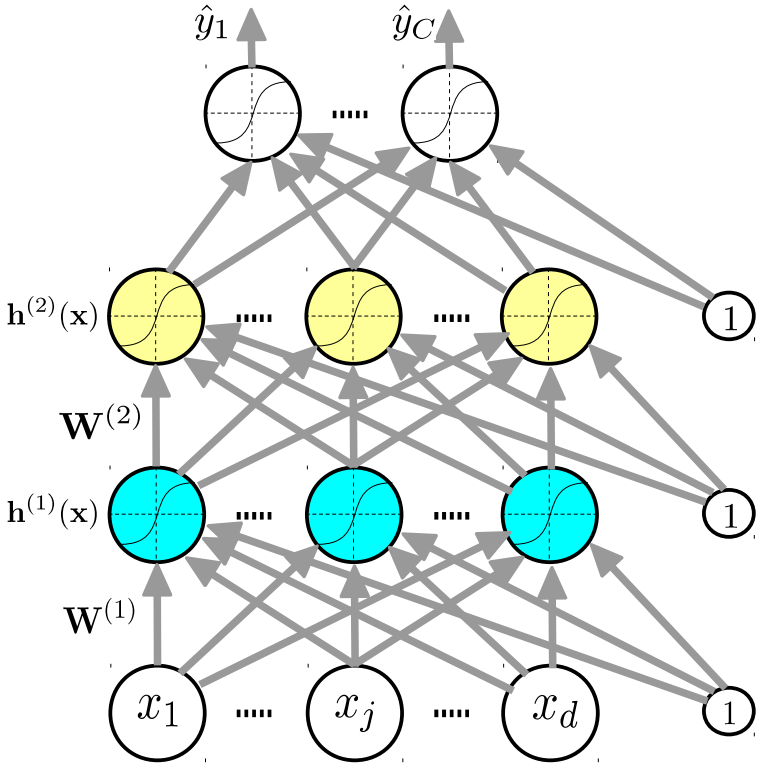


- A greedy, layer-wise procedure
  - train one layer at a time, from first to last, with unsupervised criterion
  - fix the parameters of previous hidden layers
  - previous layers viewed as feature extraction



Fine-tuning

- Once all layers are pre-trained
  - add output layer
  - train the whole network using supervised learning
- Supervised learning is performed as in a regular feed-forward network
  - forward propagation, backpropagation and update
- We call this last phase **fine-tuning**
  - all parameters are "tuned" for the supervised task at hand
  - representation is adjusted to be more discriminative



Pre-training + fine tuning

- for $l=1$ to L
  - build unsupervised training set (with $\mathbf{h}^{(0)}(\mathbf{x}) = \mathbf{x}$):

$$\mathcal{D} = \left\{ \mathbf{h}^{(l-1)}(\mathbf{x}^{(t)}) \right\}_{t=1}^{T}$$

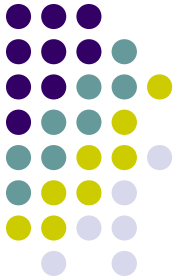
  - train "greedy module" (RBM, autoencoder) on $\mathcal{D}$
  - use hidden layer weights and biases of greedy module to initialize the deep network parameters $\mathbf{W}^{(l)}$, $\mathbf{b}^{(l)}$

pre-training

- Initialize $\mathbf{W}^{(L+1)}$, $\mathbf{b}^{(L+1)}$ randomly (as usual)
- Train the whole neural network using (supervised) stochastic gradient descent (with backprop)

fine-tuning

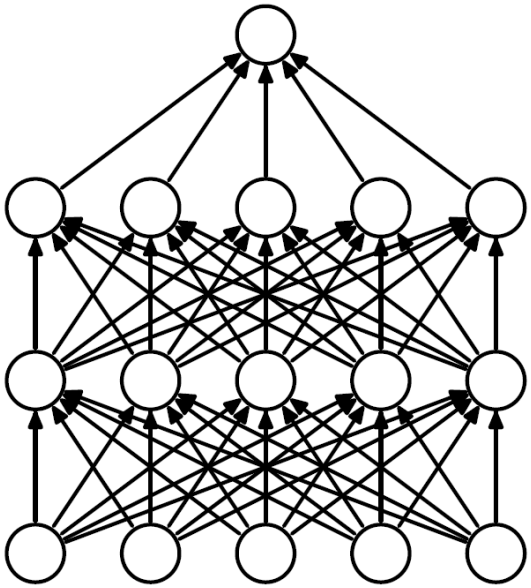
Dropout



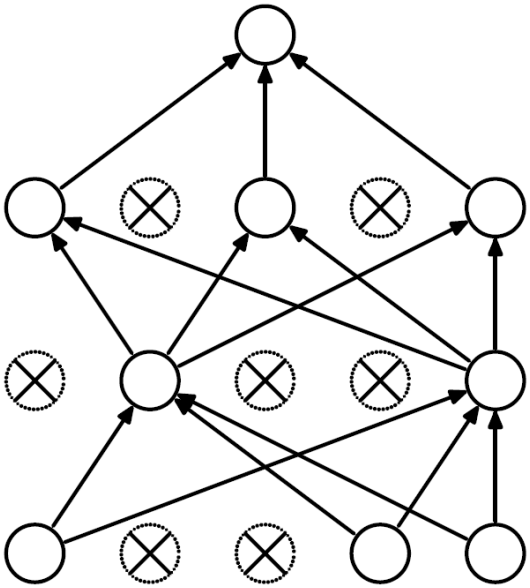
- It is a technique for regularization that relies on modifying the network itself.
- With dropout, the training process starts by randomly (and temporarily) deleting the hidden neurons in the network with fixed probability p, while leaving the input and output neurons untouched.
- The input x is forward-propagated through the modified network, and then the result is backpropagated, also through the modified network.
- After doing this over a mini-batch of examples, the appropriate weights and biases are updated.
- The processis repeated, first restoring the dropout neurons, then choosing a new random subset of hidden neurons to delete, estimating the gradient for a different mini-batch, and updating the weights and biases in the network.

Dropout

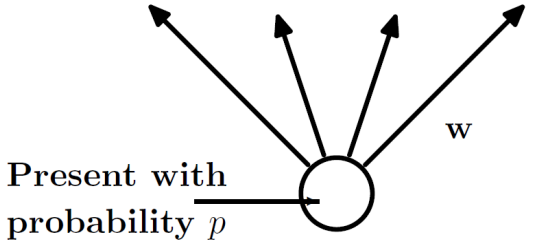
- At test time, all neurons are kept (no dropout), but the activations are scaled by p (probability of dropping each neuron).



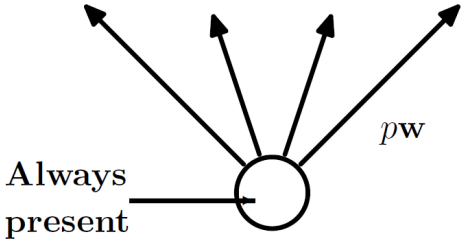
(a) Standard Neural Net



(b) After applying dropout.

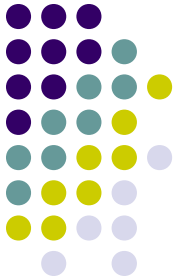


(a) At training time



(b) At test time

Why dropout?



- With dropout, the weights of the nodes learned through backpropagation become somewhat more insensitive to the weights of the other nodes and learn to decide the outcome independent of the other neurons (hidden neurons cannot co-adapt to other hidden neurons)
- Dropout can also be thought of as an ensemble consisting of all sub-networks that can be constructed by removing non-output units from an underlying base network. When a neuron is dropped, we are basically subsampling a part of the neural network and we are training it on the current mini-batch.
- In every iteration of training, we will subsample a different part of the network and train that network on the datapoints. Thus what we have essentially is an ensemble of models that share some parameters.

Why dropout?