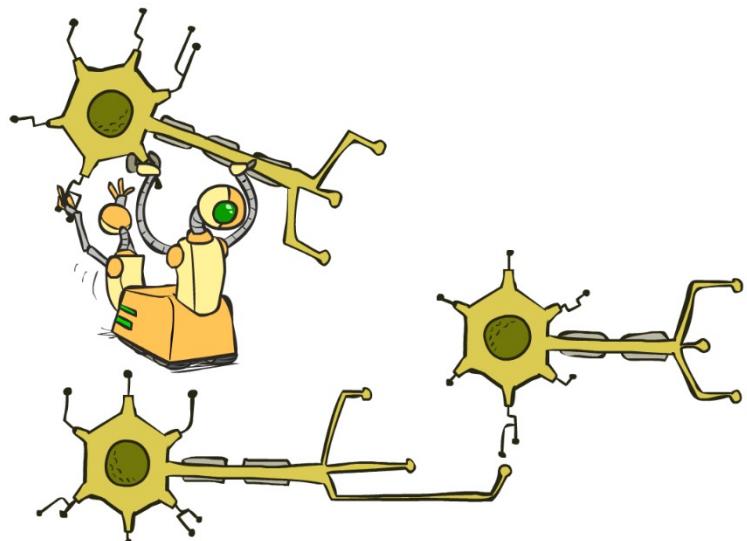


# Neural Networks

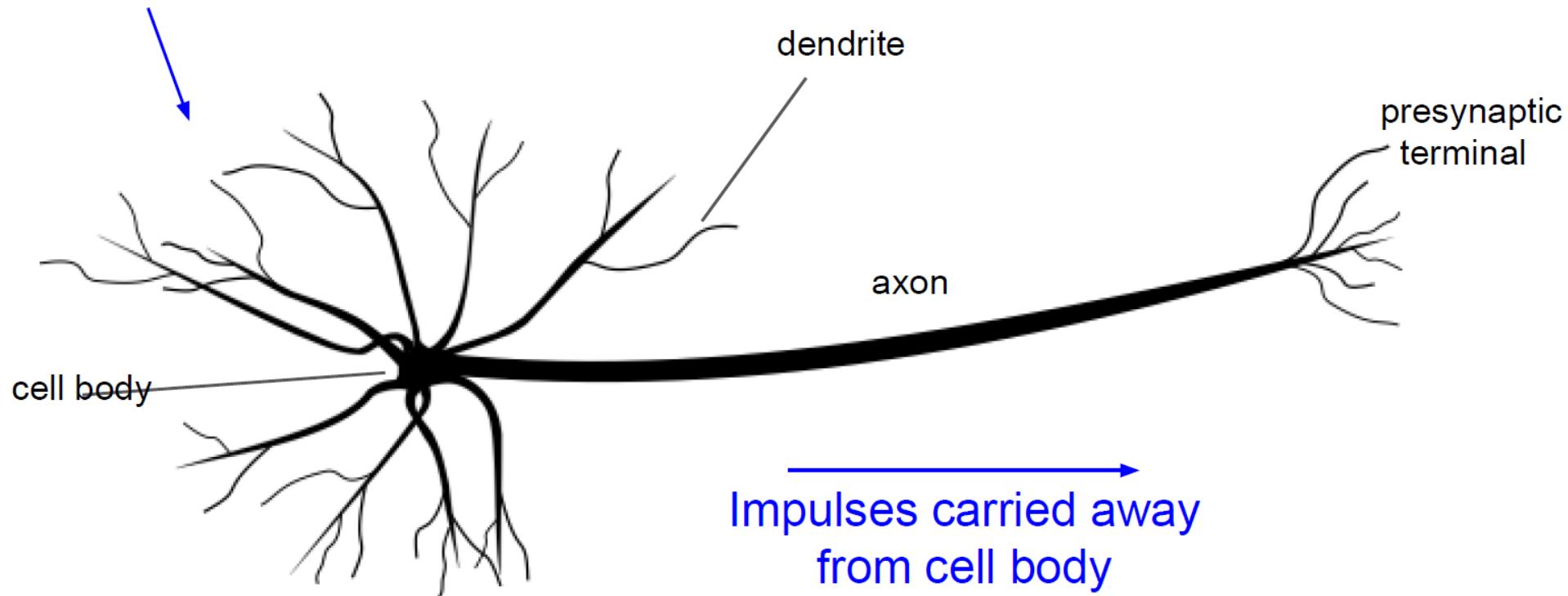
CE-417: Artificial Intelligence  
Sharif University of Technology  
Fall 2023

Soleymani

Some slides are based on Anca Dragan's slides, CS188, UC Berkeley  
and some adapted from Bhiksha Raj, II-785, CMU 2019.



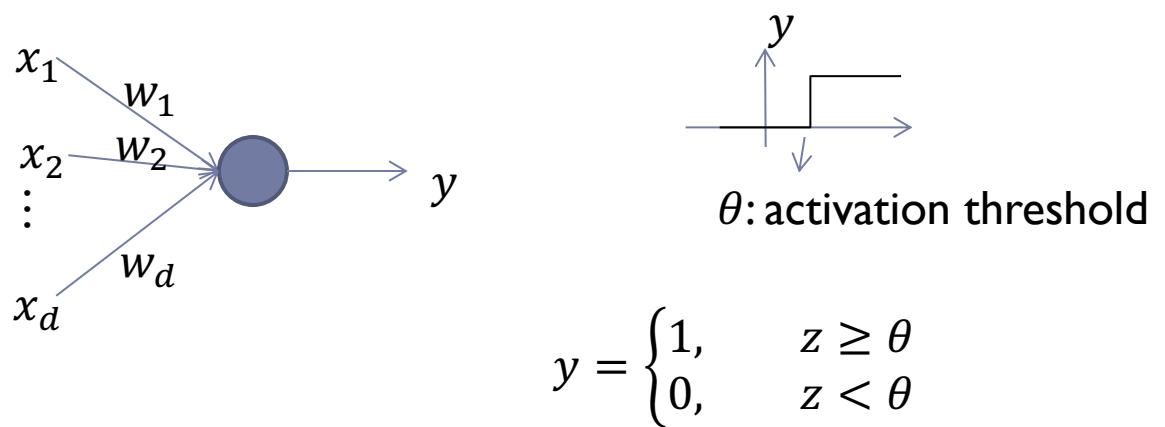
Impulses carried toward cell body



Impulses carried away  
from cell body

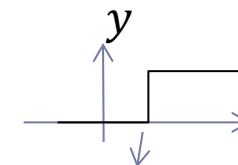
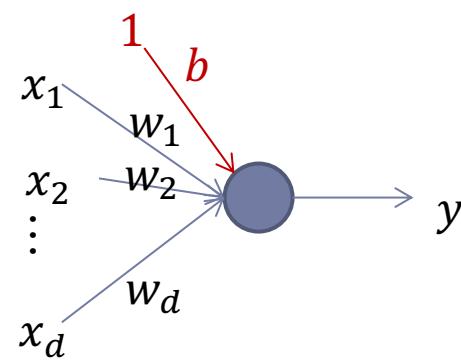
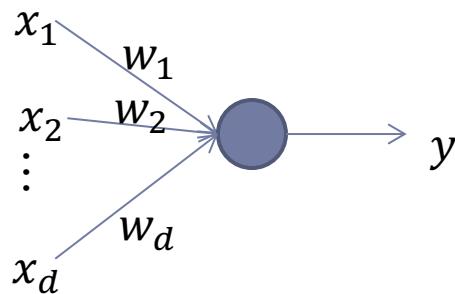
[This image](#) by Felipe Perucho  
is licensed under [CC-BY 3.0](#)

# McCulloch-Pitts neuron: binary threshold



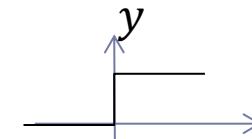
# McCulloch-Pitts neuron: binary threshold

Equivalent to



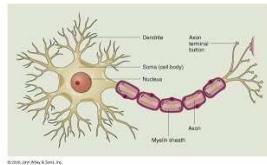
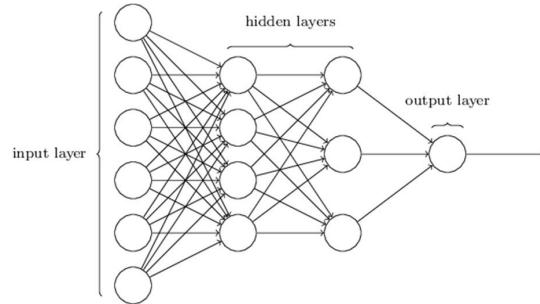
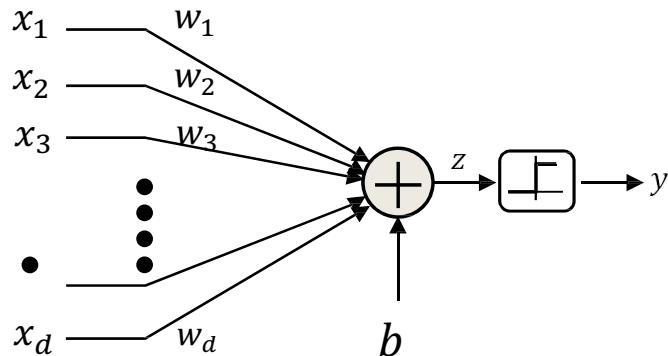
$\theta$ : activation threshold

$$y = \begin{cases} 1, & z \geq \theta \\ 0, & z < \theta \end{cases}$$



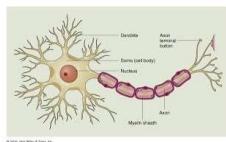
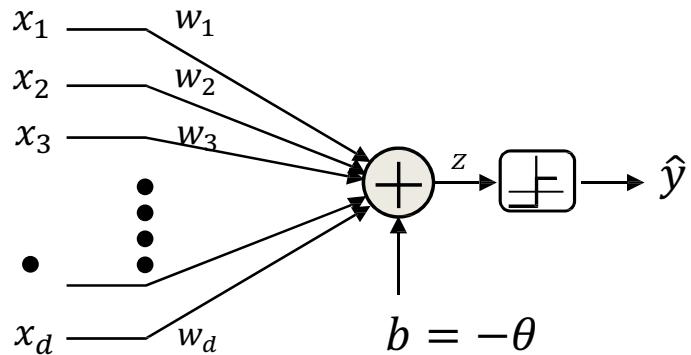
bias:  $b = -\theta$

# Neural nets and the brain



- Neural nets are composed of networks of computational models of neurons called perceptrons

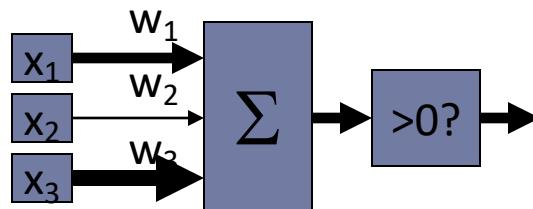
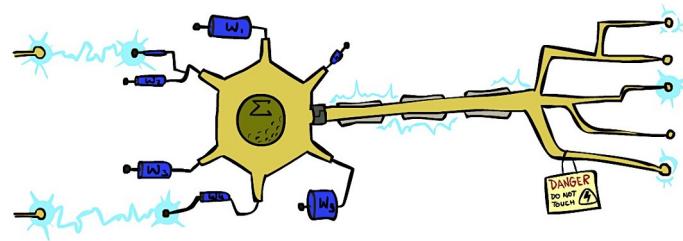
# The perceptron



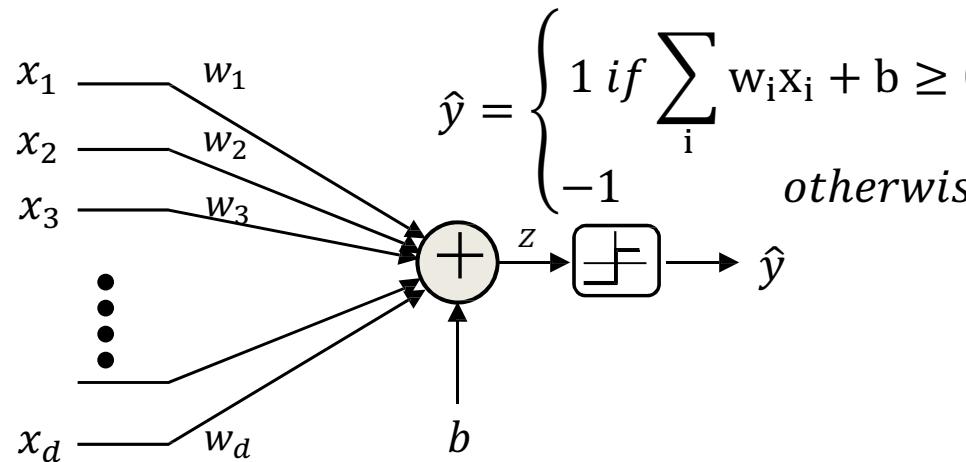
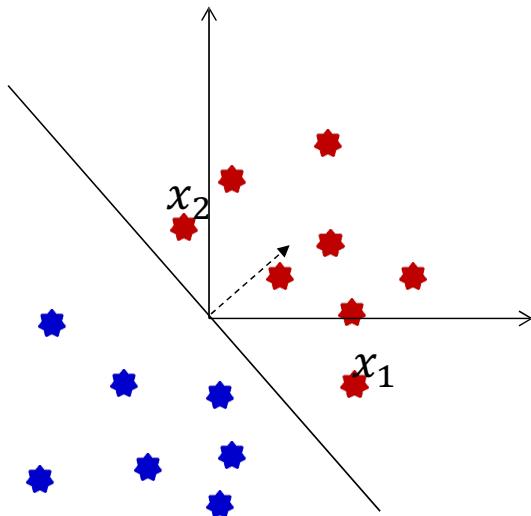
$$\hat{y} = \begin{cases} 1 & \text{if } \sum_i w_i x_i \geq \theta \\ -1 & \text{else} \end{cases}$$

# Reminder: Linear Classifiers

- Inputs are **feature values**
  - Each feature has a **weight**
  - Sum is the **activation**
- 
- If the activation is:
    - Positive, output +1
    - Negative, output -1



# Learning the perceptron



- Given a number of input output pairs, learn the weights and bias
  - Learn  $W = [w_1, \dots, w_d]^T$  and  $b$ , given several  $(x, y)$  pairs

# Perceptron Algorithm: Summary

- Cycle through the training instances
- Only update  $w$  on misclassified instances
- If instance misclassified:
  - If instance is positive class

$$w = w + x^{(n)}$$

- If instance is negative class

$$w = w - x^{(n)}$$

# Perceptron vs. Delta Rule

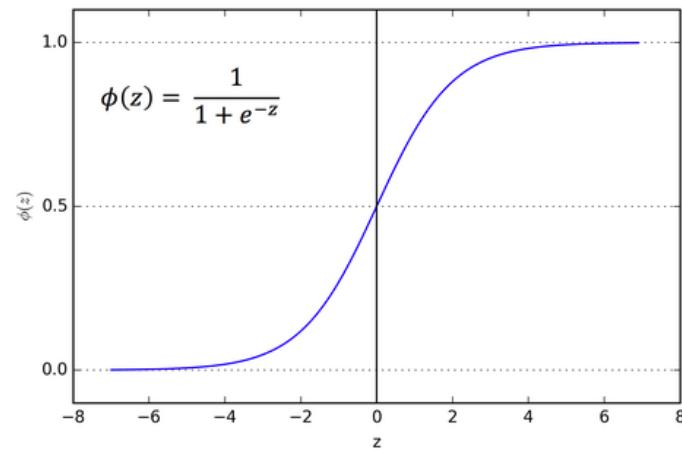
- Perceptron learning rule:
  - guaranteed to succeed if training examples are linearly separable
- Delta rule:
  - guaranteed to converge to the hypothesis with the minimum squared error
  - can also be used for regression problems

$$\mathbf{w} = \mathbf{w} + \eta(y^{(n)} - \mathbf{w}^T \mathbf{x}^{(n)}) \mathbf{x}^{(n)}$$

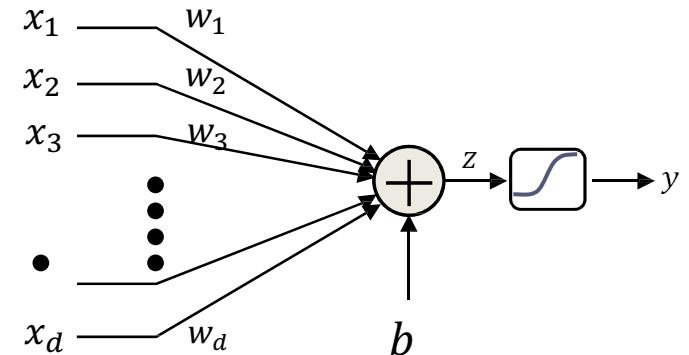
# How to get probabilistic decisions?

- Activation:
- If  $z = \mathbf{w}^T \mathbf{x}$  very positive  $\rightarrow$  want probability going to 1
- If  $z = \mathbf{w}^T \mathbf{x}$  very negative  $\rightarrow$  want probability going to 0
- Sigmoid function

$$\phi(z) = \frac{1}{1 + e^{-z}}$$



# Logistic regression



Maximum likelihood estimation:

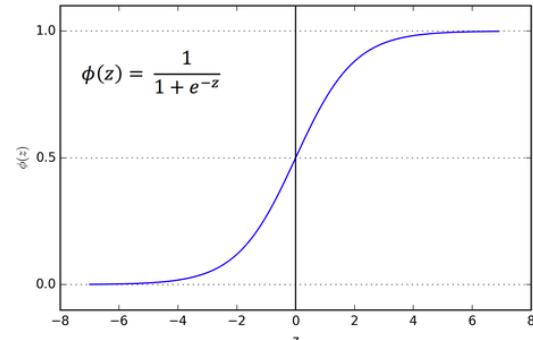
$$\max_w ll(w) = \max_w \sum_i \log P(y^{(i)} | x^{(i)}; w)$$

with:

$$P(y = +1 | x; w) = \frac{1}{1 + e^{-w^T x}} = \sigma(w^T x)$$

$$P(y = 0 | x; w) = 1 - \frac{1}{1 + e^{-w^T x}} = 1 - \sigma(w^T x)$$

= Logistic Regression



# Multi-class logistic regression

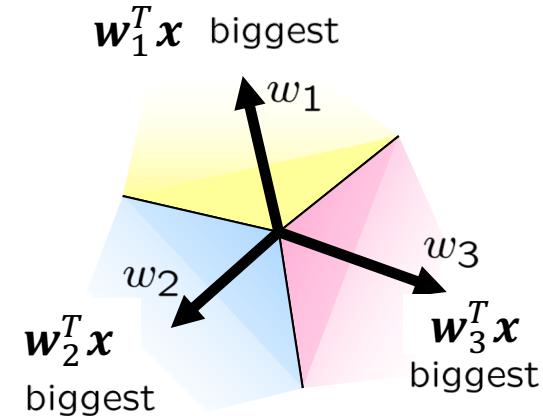
- Maximum likelihood estimation:

$$\max_w \text{ll}(w) = \max_w \sum_i \log P(y^{(i)} | x^{(i)}; w)$$

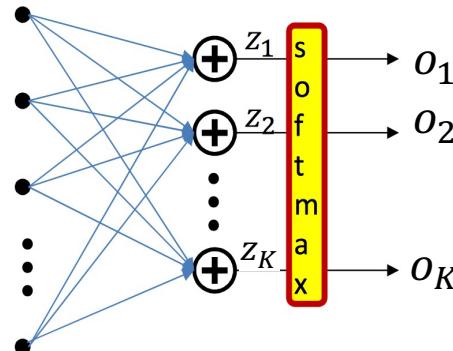
with:

$$P(y^{(i)} | x^{(i)}; w) = \frac{e^{\mathbf{w}_{y^{(i)}}^T \mathbf{x}^{(i)}}}{\sum_{k=1}^K e^{\mathbf{w}_k^T \mathbf{x}^{(i)}}}$$

= Multi-Class Logistic Regression



# Softmax activation function



$$o_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

- Softmax vector activation is often used at the output of multi-class classifier nets

$$z_i = \sum_j w_{ji}^{(l)} a_j^{(n-1)}$$

$$o_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

- This can be viewed as the probability  $o_i = P(\text{class} = i | \mathbf{x})$

# Batch Gradient Ascent on the Log Likelihood Objective

$$\max_w \text{ ll}(w) = \max_w \underbrace{\sum_i \log P(y^{(i)} | x^{(i)}; w)}_{g(w)}$$

- init  $w$
- for iter = 1, 2, ...

$$w \leftarrow w + \alpha * \sum_i \nabla \log P(y^{(i)} | x^{(i)}; w)$$

# Stochastic Gradient Ascent on the Log Likelihood Objective

$$\max_w \text{ll}(w) = \max_w \sum_i \log P(y^{(i)} | x^{(i)}; w)$$

**Observation:** once gradient on one training example has been computed, might as well incorporate before computing next one

- init  $w$
- for iter = 1, 2, ...
  - pick random  $j$

$$w \leftarrow w + \alpha * \nabla \log P(y^{(j)} | x^{(j)}; w)$$

# Mini-Batch Gradient Ascent on the Log Likelihood Objective

$$\max_w \text{ll}(w) = \max_w \sum_i \log P(y^{(i)} | x^{(i)}; w)$$

**Observation:** gradient over small set of training examples (=mini-batch) can be computed, might as well do that instead of a single one

- init  $w$
- for iter = 1, 2, ...
  - pick random subset of training examples  $J$

$$w \leftarrow w + \alpha * \sum_{j \in J} \nabla \log P(y^{(j)} | x^{(j)}; w)$$

# Limitation of single layer network

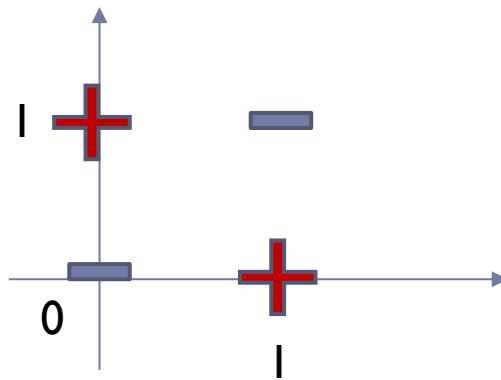
- Single layer networks is equivalent to template matching
  - Weights for each class as a template for that class.
- The ways in which a digit can be written are much too complicated to be captured by simple template
- Thus, networks without hidden units are very limited in the mappings that they can learn

# The history of Perceptron

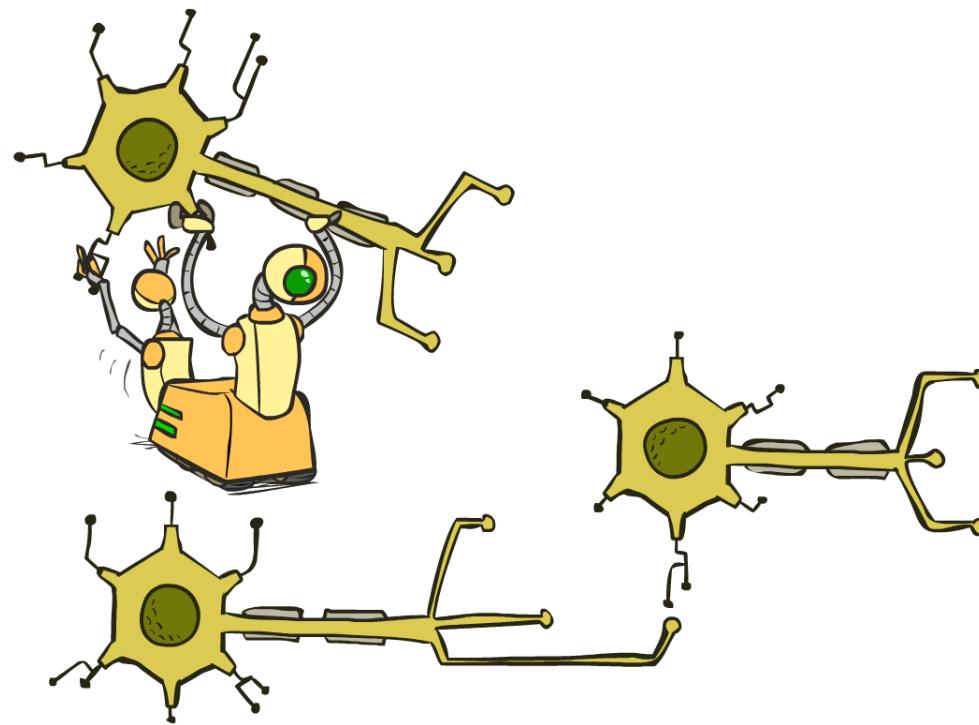
- They were popularized by Frank Rosenblatt in the early 1960's.
  - They appeared to have a very powerful learning algorithm.
  - Lots of grand claims were made for what they could learn to do.
- In 1969, Minsky and Papert published a book called “Perceptrons” that analyzed what they could do and showed their limitations.
  - Many people thought these limitations applied to all neural network models.

# What binary threshold neurons cannot do

- A binary threshold output unit cannot even tell if two single bit features are the same!
- A geometric view of what binary threshold neurons cannot do
- The positive and negative cases cannot be separated by a plane



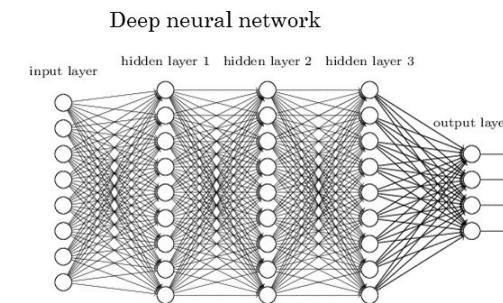
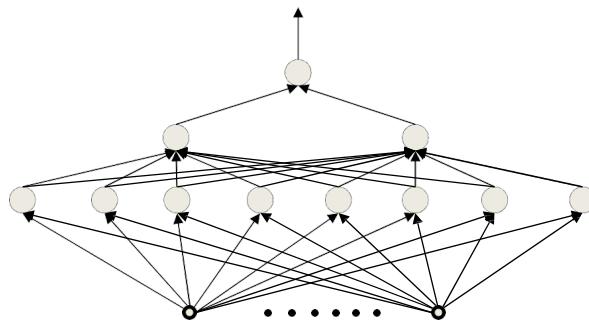
# Neural Networks



# Networks with hidden units

- Networks without hidden units are very limited in the input-output mappings they can learn to model.
  - More layers of linear units do not help. Its still linear.
  - Fixed output non-linearities are not enough.
- We need multiple layers of adaptive, non-linear hidden units. But how can we train such nets?

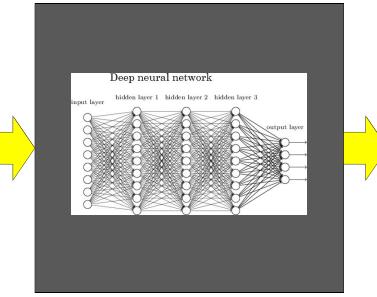
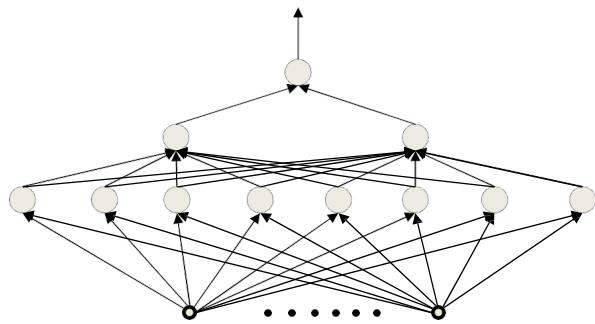
# The multi-layer perceptron



- A network of perceptrons
  - Generally “layered”

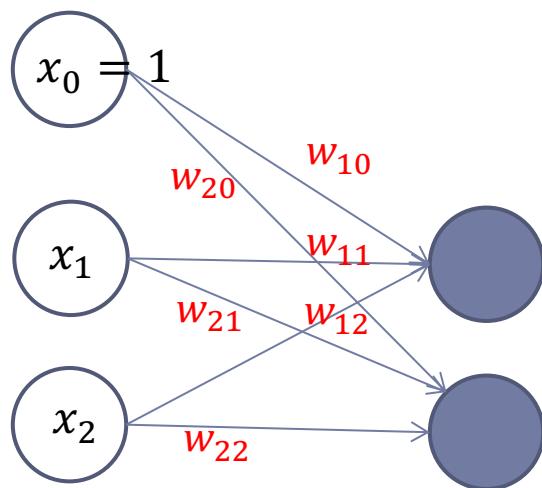


# The multi-layer perceptron



- Inputs are real or Boolean stimuli
- Outputs are real or Boolean values
  - Can have multiple outputs for a single input
- What can this network compute?
  - What kinds of input/output relationships can it model?

# Linear model



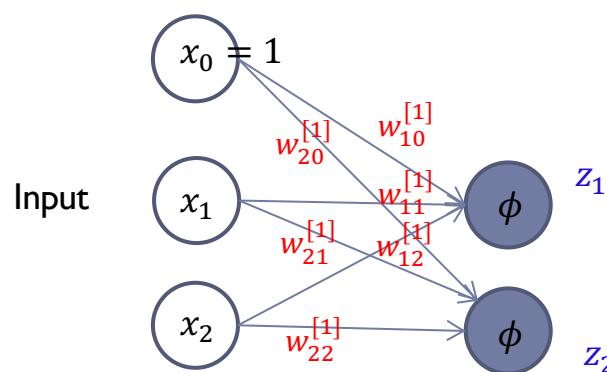
Input

Output:  $Wx$

$$W = \begin{bmatrix} w_{10} & w_{11} & \dots & w_{1d} \\ \vdots & \ddots & \dots & \vdots \\ w_{k0} & w_{k1} & \dots & w_{kd} \end{bmatrix} \quad x = \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ \vdots \\ x_d \end{bmatrix}$$

# Multi-layer Neural Network

$$W^{[1]} = \begin{bmatrix} w_{10}^{[1]} & w_{11}^{[1]} & w_{12}^{[1]} \\ w_{20}^{[1]} & w_{21}^{[1]} & w_{22}^{[1]} \end{bmatrix}$$



$\phi$ : fixed activation function

Examples:

$$\phi(z) = \max(0, z)$$

$$\phi(z) = \frac{1}{1 + e^{-z}}$$

$$z_j = \phi \left( \sum_{i=0}^d w_{ji}^{[1]} x_i \right)$$

Matrix form:  $\mathbf{z} = \phi(W^{[1]} \mathbf{x})$

# Multi-layer Neural Network

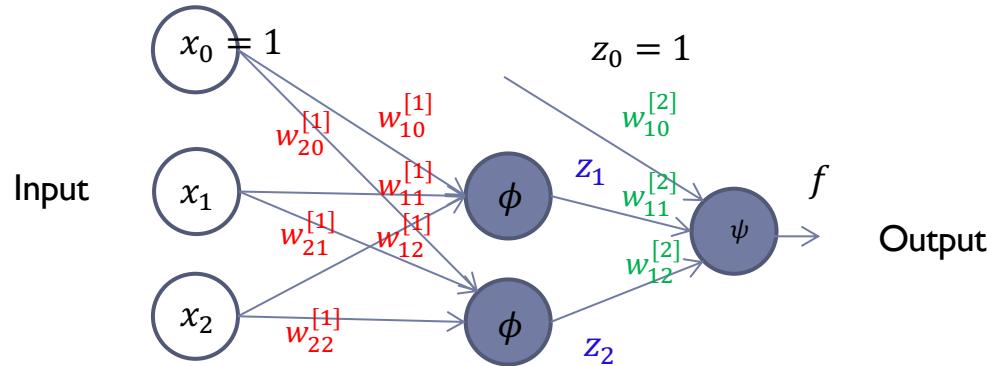
$$\mathbf{W}^{[1]} = \begin{bmatrix} w_{10}^{[1]} & w_{11}^{[1]} & w_{12}^{[1]} \\ w_{20}^{[1]} & w_{21}^{[1]} & w_{22}^{[1]} \end{bmatrix}$$

$\phi$ : fixed activation function

Examples:

$$\phi(z) = \max(0, z)$$

$$\phi(z) = \frac{1}{1 + e^{-z}}$$



$$z_j = \phi \left( \sum_{i=0}^d w_{ji}^{[1]} x_i \right)$$

$$f = \psi \left( \sum_{j=0}^2 w_{kj}^{[2]} z_j \right) = \left( \sum_{j=0}^2 w_{kj}^{[2]} \phi \left( \sum_{i=0}^d w_{ji}^{[1]} x_i \right) \right)$$

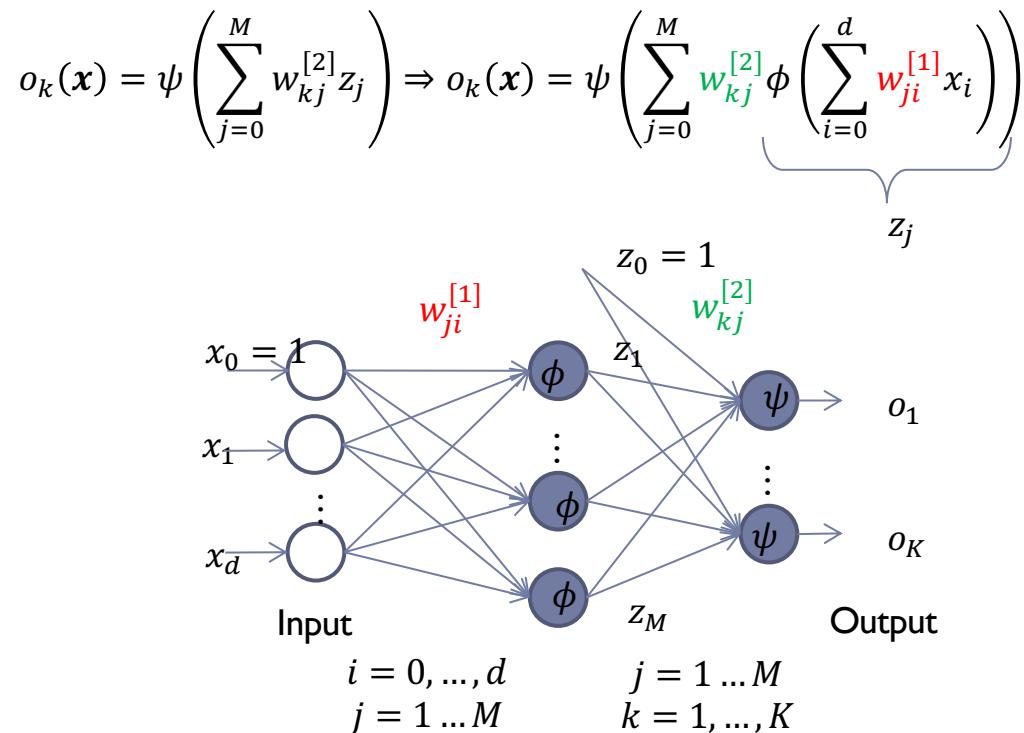
Matrix form:

$$\mathbf{z} = \phi(\mathbf{W}^{[1]} \mathbf{x})$$

$$\begin{aligned} f(\mathbf{x}) &= \psi(\mathbf{W}^{[2]} \mathbf{z}) \\ &= \psi(\mathbf{W}^{[2]} \phi(\mathbf{W}^{[1]} \mathbf{x})) \end{aligned}$$

# MLP with single hidden layer

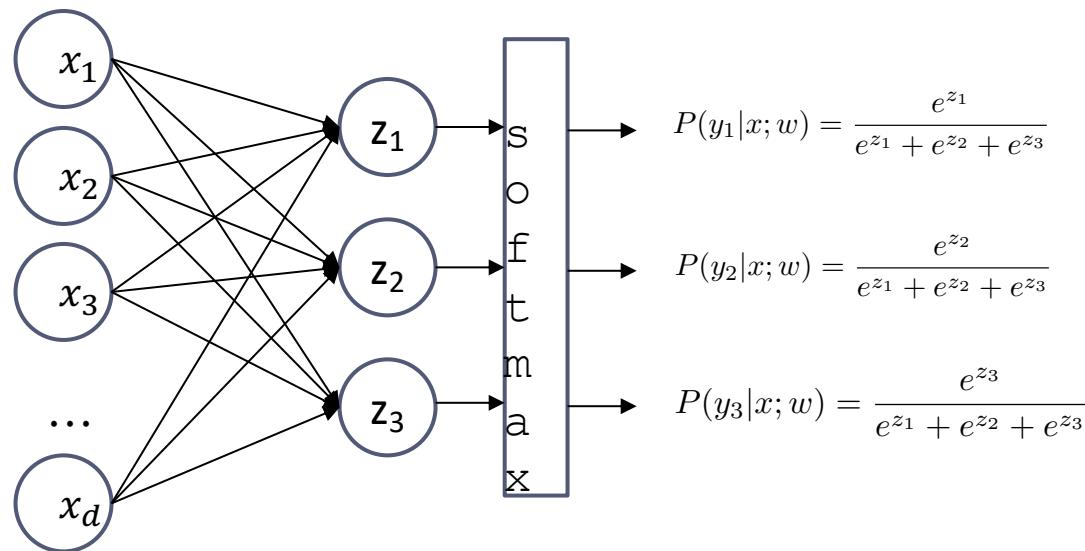
- Two-layer MLP (Number of layers of adaptive weights is counted)



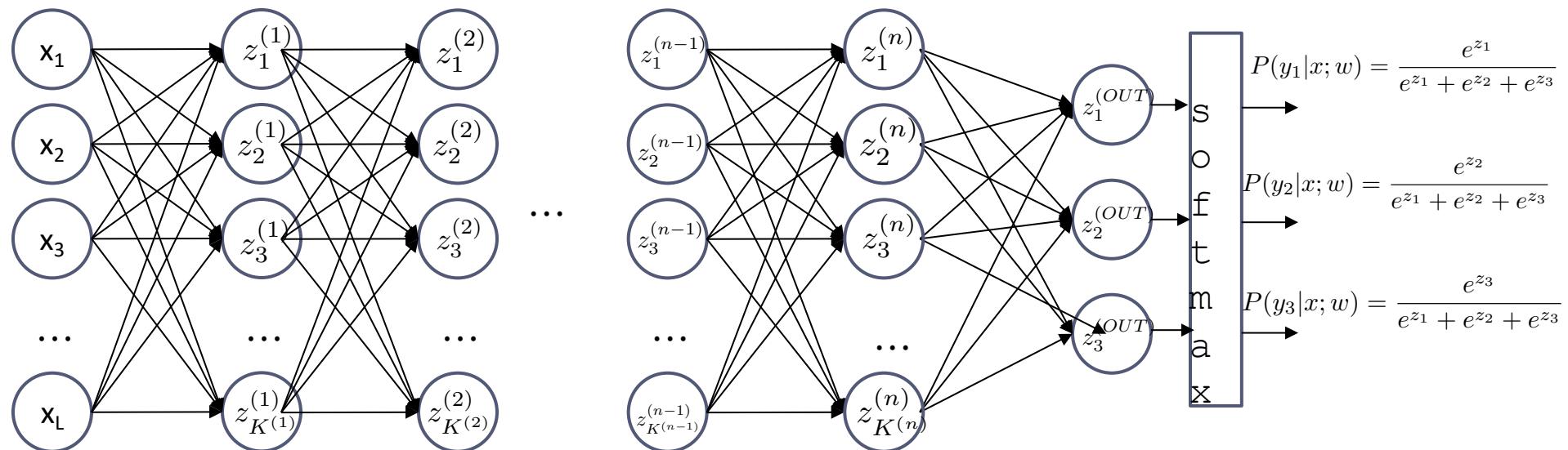
- Thus, we don't need expert knowledge or time consuming tuning of hand-crafted features
  - The form of the nonlinearity (basis functions  $f_j$ ) is adapted from the training data

# Multi-class Logistic Regression

= special case of neural network



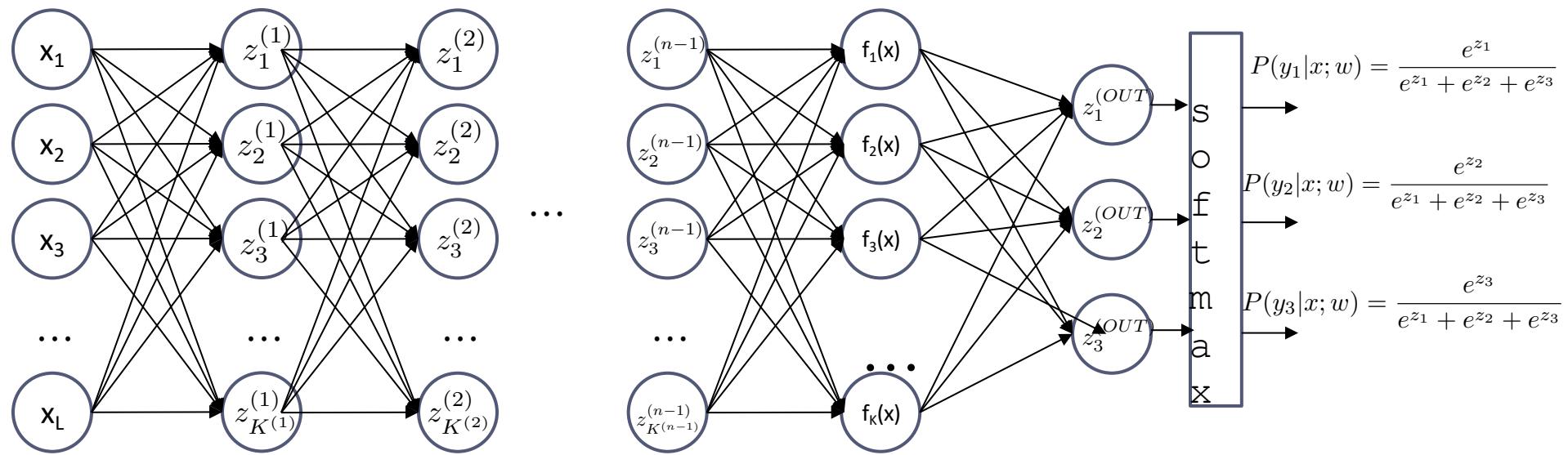
# Deep Neural Network = Also learn the features!



$$z_i^{(k)} = \phi\left(\sum_j W_{i,j}^{(k-1,k)} z_j^{(k-1)}\right)$$

**$\phi$  = nonlinear activation function**

# Deep Neural Network = Also learn the features!



$$z_i^{(k)} = \phi\left(\sum_j W_{i,j}^{(k-1,k)} z_j^{(k-1)}\right)$$

$\phi$  = nonlinear activation function

# Neural Networks Properties

- **Theorem (Universal Function Approximators).** A two-layer neural network with a sufficient number of neurons can approximate any continuous function to any desired accuracy.
- **Practical considerations**
  - Can be seen as learning the features
  - Large number of neurons
    - Danger for overfitting

# Universal Function Approximation Theorem\*

**Hornik theorem 1:** Whenever the activation function is *bounded and nonconstant*, then, for any finite measure  $\mu$ , standard multilayer feedforward networks can approximate any function in  $L^p(\mu)$  (the space of all functions on  $R^k$  such that  $\int_{R^k} |f(x)|^p d\mu(x) < \infty$ ) arbitrarily well, provided that sufficiently many hidden units are available.

**Hornik theorem 2:** Whenever the activation function is *continuous, bounded and non-constant*, then, for arbitrary compact subsets  $X \subseteq R^k$ , standard multilayer feedforward networks can approximate any continuous function on  $X$  arbitrarily well with respect to uniform distance, provided that sufficiently many hidden units are available.

- **In words:** Given any continuous function  $f(x)$ , if a 2-layer neural network has enough hidden units, then there is a choice of weights that allow it to closely approximate  $f(x)$ .

Cybenko (1989) “Approximations by superpositions of sigmoidal functions”

Hornik (1991) “Approximation Capabilities of Multilayer Feedforward Networks”

Leshno and Schocken (1991) “Multilayer Feedforward Networks with Non-Polynomial Activation Functions Can Approximate Any Function”

# MLP Universal Approximator

- A feed-forward network with a single hidden layer and linear outputs can approximate any continuous function on a compact domain to an arbitrary accuracy
  - under mild assumptions on the activation function
    - e.g., sigmoid activation functions (Cybenko, 1989)
  - when sufficiently large (but finite) number of hidden units is used

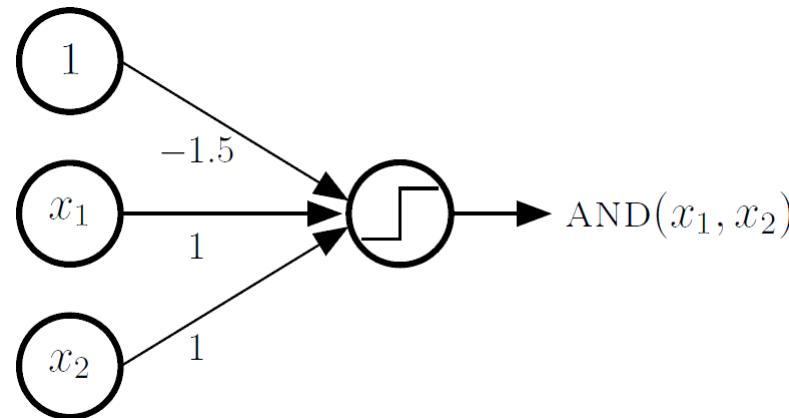
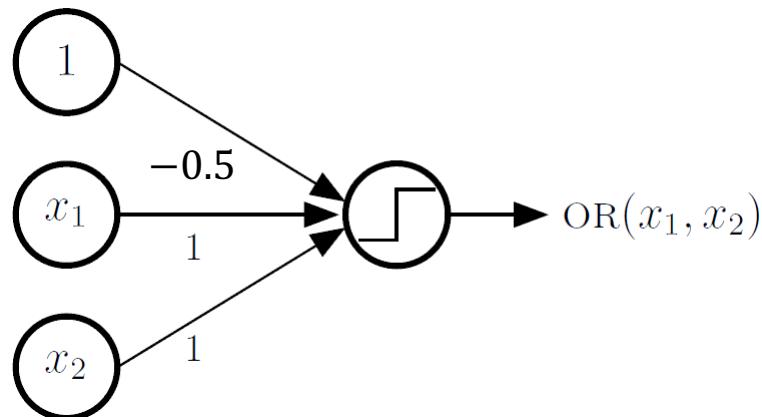
$$F_k(x) = \sum_{j=1}^M w_{kj}^{[2]} \phi \left( \sum_{i=0}^d w_{ji}^{[1]} x_i \right)$$

- It is of greater theoretical interest than practical
  - the construction of such a network requires the nonlinear activation functions and weight values which are unknown

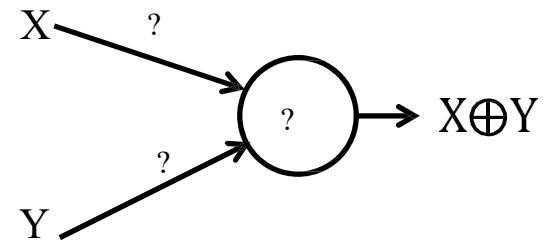
# MLPs approximate functions

- MLPs can compose Boolean functions
- MLPs as universal classifiers
- MLPs as universal approximators (of real-valued functions)

# AND & OR networks



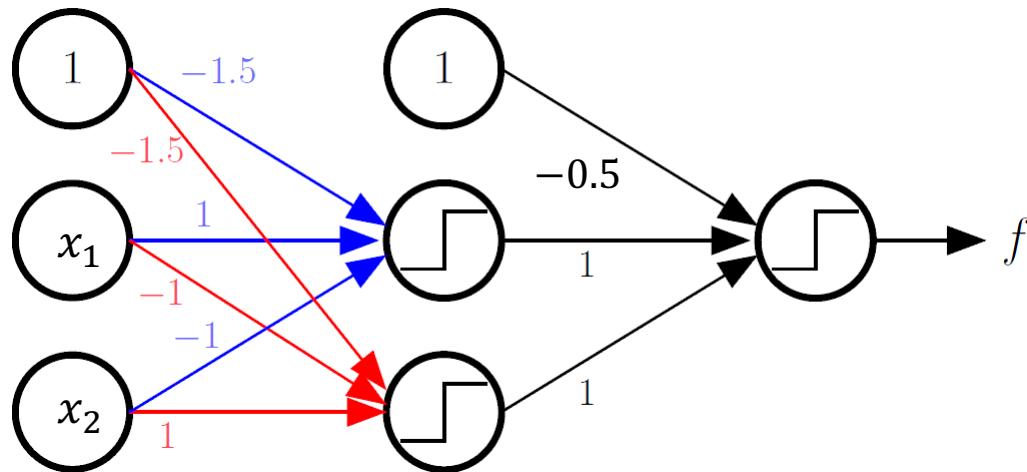
# The perceptron is not enough



- Cannot compute an XOR

# XOR example

$$f = XOR(x_1, x_2) = OR(AND(x_1, \bar{x}_2), AND(\bar{x}_1, x_2))$$



Input variables that are True are considered as 1 and False ones as -1

# General Boolean functions

- Every Boolean function can be represented by a network with a single hidden layer
  1. Consider the truth table of the Boolean function
  2. Write Boolean function as OR of ANDs, with one AND for each positive entry in the truth table.
  3. Construct a 2-layer network that is composed of OR of ANDs (first layer contains ANDs and second layer contains OR)
- It might need an exponential number of hidden units

# How many layers for a Boolean MLP?

Truth table shows all input combinations  
for which output is 1

Truth Table					
X <sub>1</sub>	X <sub>2</sub>	X <sub>3</sub>	X <sub>4</sub>	X <sub>5</sub>	Y
0	0	1	1	0	1
0	1	0	1	1	1
0	1	1	0	0	1
1	0	0	0	1	1
1	0	1	1	1	1
1	1	0	0	1	1

$$y = \bar{X}_1 \bar{X}_2 X_3 X_4 \bar{X}_5 + \bar{X}_1 X_2 \bar{X}_3 X_4 X_5 + \bar{X}_1 X_2 X_3 \bar{X}_4 \bar{X}_5 + \\ X_1 \bar{X}_2 \bar{X}_3 \bar{X}_4 X_5 + X_1 \bar{X}_2 X_3 X_4 X_5 + X_1 X_2 \bar{X}_3 \bar{X}_4 X_5$$

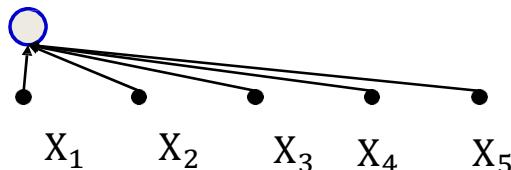
- Expressed in disjunctive normal form

# How many layers for a Boolean MLP?

Truth Table					
X <sub>1</sub>	X <sub>2</sub>	X <sub>3</sub>	X <sub>4</sub>	X <sub>5</sub>	Y
0	0	1	1	0	1
0	1	0	1	1	1
0	1	1	0	0	1
1	0	0	0	1	1
1	0	1	1	1	1
1	1	0	0	1	1

Truth table shows all input combinations  
for which output is 1

$$y = \bar{X}_1 \bar{X}_2 X_3 X_4 \bar{X}_5 + \bar{X}_1 X_2 \bar{X}_3 X_4 X_5 + \bar{X}_1 X_2 X_3 \bar{X}_4 \bar{X}_5 + \\ X_1 X_2 \bar{X}_3 X_4 X_5 + X_1 \bar{X}_2 X_3 X_4 X_5 + X_1 X_2 \bar{X}_3 \bar{X}_4 X_5$$



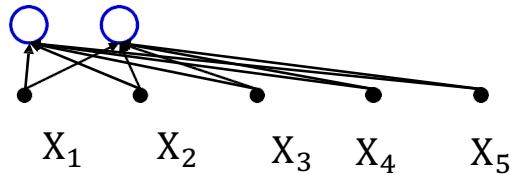
- Expressed in disjunctive normal form

# How many layers for a Boolean MLP?

Truth table shows all input combinations  
for which output is 1

Truth Table					
X <sub>1</sub>	X <sub>2</sub>	X <sub>3</sub>	X <sub>4</sub>	X <sub>5</sub>	Y
0	0	1	1	0	1
0	1	0	1	1	1
0	1	1	0	0	1
1	0	0	0	1	1
1	0	1	1	1	1
1	1	0	0	1	1

$$y = \bar{X}_1 \bar{X}_2 X_3 X_4 \bar{X}_5 + \boxed{\bar{X}_1 X_2 \bar{X}_3 X_4 X_5} + \bar{X}_1 X_2 X_3 \bar{X}_4 \bar{X}_5 + \\ X_1 \bar{X}_2 \bar{X}_3 \bar{X}_4 X_5 + X_1 X_2 X_3 X_4 X_5 + X_1 X_2 \bar{X}_3 \bar{X}_4 X_5$$



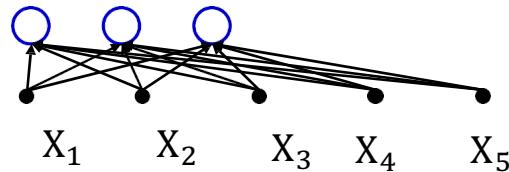
- Expressed in disjunctive normal form

# How many layers for a Boolean MLP?

Truth table shows all input combinations  
for which output is 1

Truth Table					
X <sub>1</sub>	X <sub>2</sub>	X <sub>3</sub>	X <sub>4</sub>	X <sub>5</sub>	Y
0	0	1	1	0	1
0	1	0	1	1	1
0	1	1	0	0	1
1	0	0	0	1	1
1	0	1	1	1	1
1	1	0	0	1	1

$$y = \bar{X}_1 \bar{X}_2 X_3 X_4 \bar{X}_5 + \bar{X}_1 X_2 \bar{X}_3 X_4 X_5 + \boxed{\bar{X}_1 X_2 X_3 \bar{X}_4 \bar{X}_5} + \\ X_1 \bar{X}_2 \bar{X}_3 \bar{X}_4 X_5 + X_1 \bar{X}_2 X_3 X_4 X_5 + X_1 X_2 X_3 \bar{X}_4 X_5$$



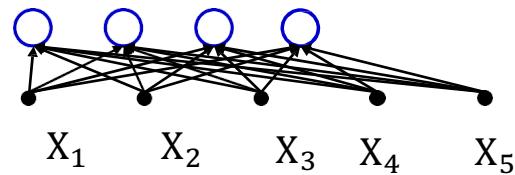
- Expressed in disjunctive normal form

# How many layers for a Boolean MLP?

Truth table shows all input combinations  
for which output is 1

Truth Table					
X <sub>1</sub>	X <sub>2</sub>	X <sub>3</sub>	X <sub>4</sub>	X <sub>5</sub>	Y
0	0	1	1	0	1
0	1	0	1	1	1
0	1	1	0	0	1
1	0	0	0	1	1
1	0	1	1	1	1
1	1	0	0	1	1

$$y = \bar{X}_1 \bar{X}_2 X_3 X_4 \bar{X}_5 + \bar{X}_1 X_2 \bar{X}_3 X_4 X_5 + \bar{X}_1 X_2 X_3 \bar{X}_4 \bar{X}_5 + \boxed{X_1 \bar{X}_2 \bar{X}_3 \bar{X}_4 X_5} + X_1 \bar{X}_2 X_3 X_4 X_5 + X_1 X_2 \bar{X}_3 \bar{X}_4 X_5$$



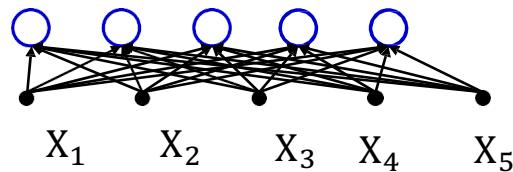
- Expressed in disjunctive normal form

# How many layers for a Boolean MLP?

Truth table shows all input combinations  
for which output is 1

Truth Table					
X <sub>1</sub>	X <sub>2</sub>	X <sub>3</sub>	X <sub>4</sub>	X <sub>5</sub>	Y
0	0	1	1	0	1
0	1	0	1	1	1
0	1	1	0	0	1
1	0	0	0	1	1
1	0	1	1	1	1
1	1	0	0	1	1

$$y = \bar{X}_1 \bar{X}_2 X_3 X_4 \bar{X}_5 + \bar{X}_1 X_2 \bar{X}_3 X_4 X_5 + \bar{X}_1 X_2 X_3 \bar{X}_4 \bar{X}_5 + \\ X_1 \bar{X}_2 \bar{X}_3 \bar{X}_4 X_5 + \boxed{X_1 \bar{X}_2 X_3 X_4 X_5} + X_1 X_2 \bar{X}_3 \bar{X}_4 X_5$$



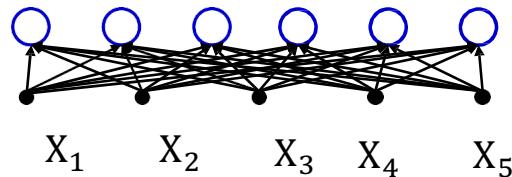
- Expressed in disjunctive normal form

# How many layers for a Boolean MLP?

Truth table shows all input combinations  
for which output is 1

Truth Table					
X <sub>1</sub>	X <sub>2</sub>	X <sub>3</sub>	X <sub>4</sub>	X <sub>5</sub>	Y
0	0	1	1	0	1
0	1	0	1	1	1
0	1	1	0	0	1
1	0	0	0	1	1
1	0	1	1	1	1
1	1	0	0	1	1

$$y = \bar{X}_1 \bar{X}_2 X_3 X_4 \bar{X}_5 + \bar{X}_1 X_2 \bar{X}_3 X_4 X_5 + \bar{X}_1 X_2 X_3 \bar{X}_4 \bar{X}_5 + \\ X_1 \bar{X}_2 \bar{X}_3 \bar{X}_4 X_5 + X_1 \bar{X}_2 X_3 X_4 X_5 + \boxed{X_1 X_2 \bar{X}_3 \bar{X}_4 X_5}$$



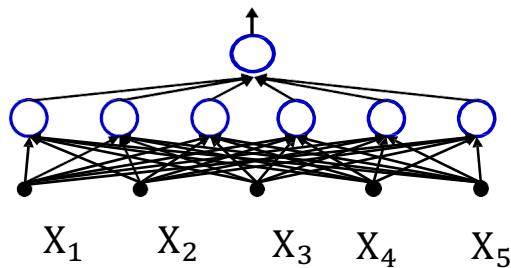
- Expressed in disjunctive normal form

# How many layers for a Boolean MLP?

Truth Table					
X <sub>1</sub>	X <sub>2</sub>	X <sub>3</sub>	X <sub>4</sub>	X <sub>5</sub>	Y
0	0	1	1	0	1
0	1	0	1	1	1
0	1	1	0	0	1
1	0	0	0	1	1
1	0	1	1	1	1
1	1	0	0	1	1

Truth table shows all input combinations  
for which output is 1

$$y = \bar{X}_1 \bar{X}_2 X_3 X_4 \bar{X}_5 + \bar{X}_1 X_2 \bar{X}_3 X_4 X_5 + \bar{X}_1 X_2 X_3 \bar{X}_4 \bar{X}_5 + \\ X_1 \bar{X}_2 \bar{X}_3 \bar{X}_4 X_5 + X_1 \bar{X}_2 X_3 X_4 X_5 + X_1 X_2 \bar{X}_3 \bar{X}_4 X_5$$

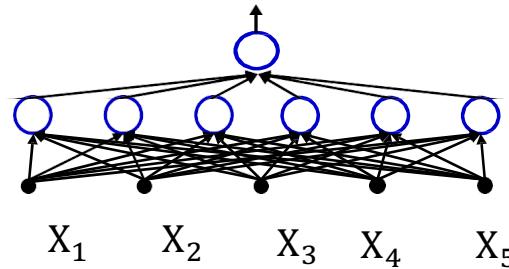


- Expressed in disjunctive normal form

# How many layers for a Boolean MLP?

Truth Table					
X <sub>1</sub>	X <sub>2</sub>	X <sub>3</sub>	X <sub>4</sub>	X <sub>5</sub>	Y
0	0	1	1	0	1
0	1	0	1	1	1
0	1	1	0	0	1
1	0	0	0	1	1
1	0	1	1	1	1
1	1	0	0	1	1

$$y = \bar{X}_1 \bar{X}_2 X_3 X_4 \bar{X}_5 + \bar{X}_1 X_2 \bar{X}_3 X_4 X_5 + \bar{X}_1 X_2 X_3 \bar{X}_4 \bar{X}_5 + \\ X_1 \bar{X}_2 \bar{X}_3 \bar{X}_4 X_5 + X_1 \bar{X}_2 X_3 X_4 X_5 + X_1 X_2 \bar{X}_3 \bar{X}_4 X_5$$

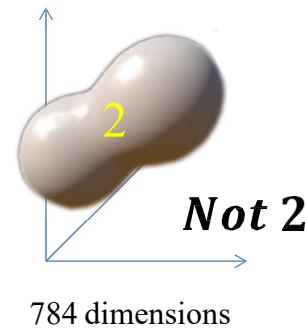
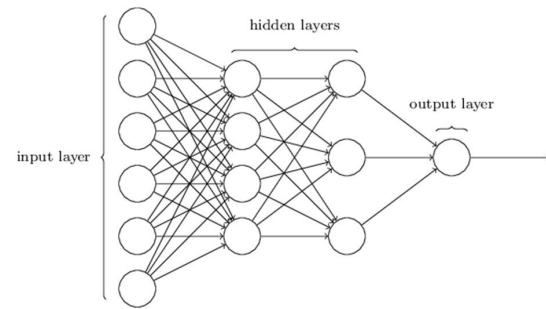
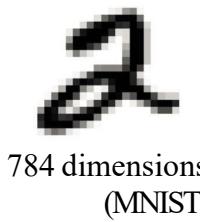


- Any truth table can be expressed in this manner!
- A one-hidden-layer MLP is a Universal Boolean Function
- But what is the largest number of perceptrons required in the single hidden layer for an N-input-variable function?

# MLPs approximate functions

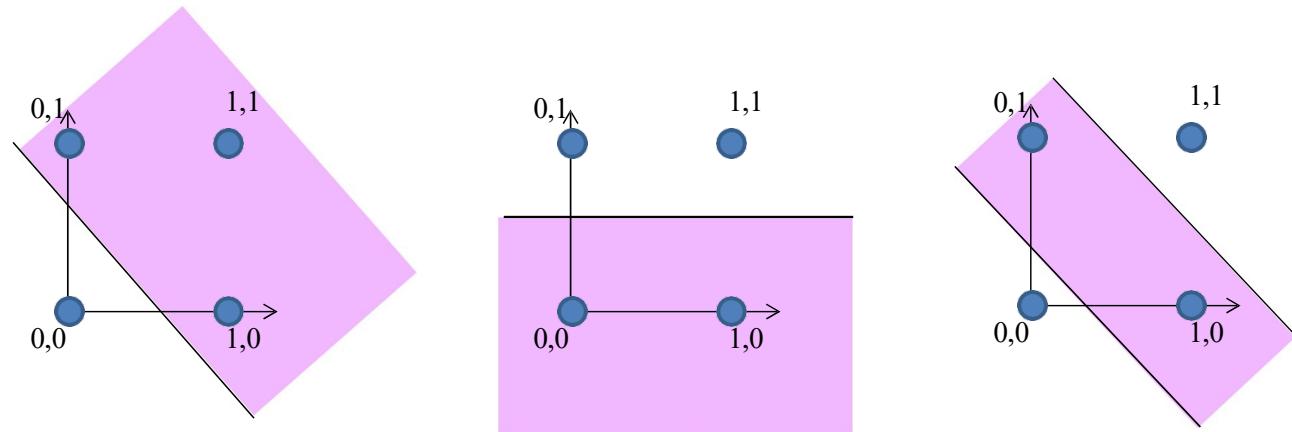
- MLPs can compose Boolean functions
- MLPs as universal classifiers
- MLPs as universal approximators (of real-valued functions)

# The MLP as a classifier



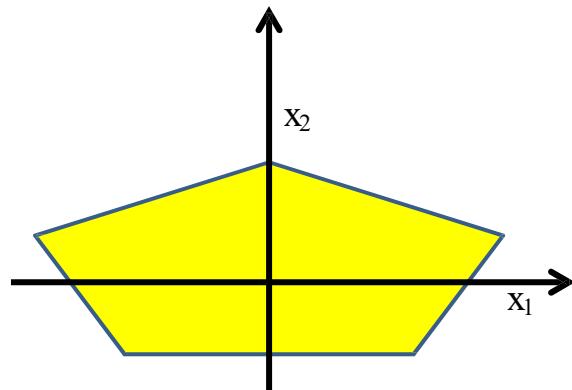
- MLP as a function over real inputs
- MLP as a function that finds a complex “decision boundary” over a space of reals

# Boolean functions with a real perceptron



- Boolean perceptrons are also linear classifiers
  - Purple regions are 1

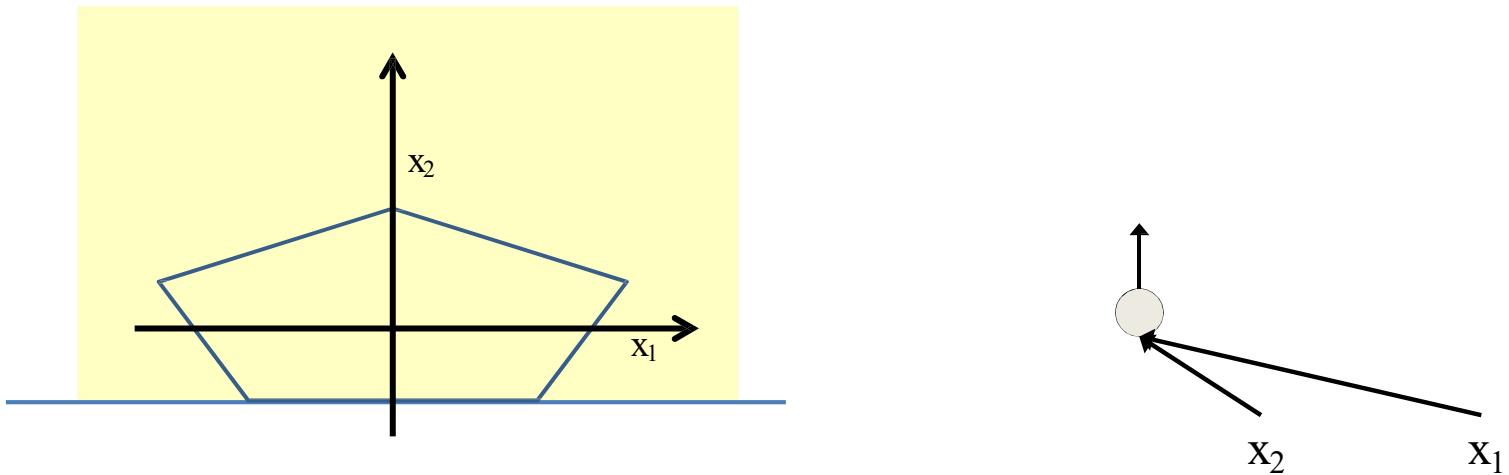
# Composing complicated “decision” boundaries



Can now be composed into “networks” to compute arbitrary classification “boundaries”

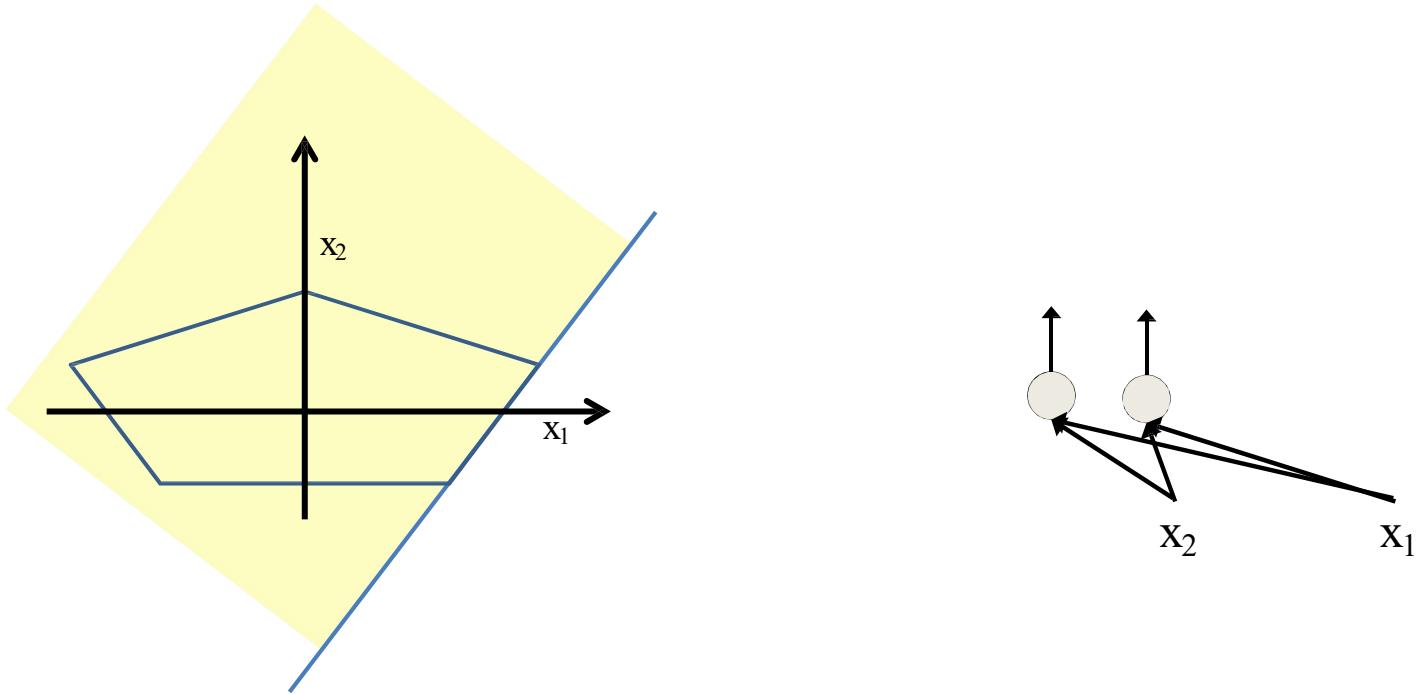
- Build a network of units with a single output that fires if the input is in the coloured area

# Booleans over the reals



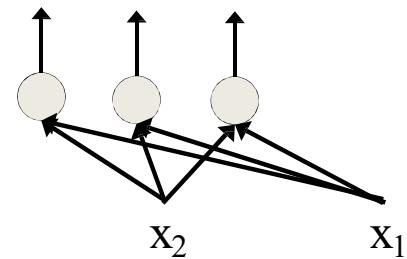
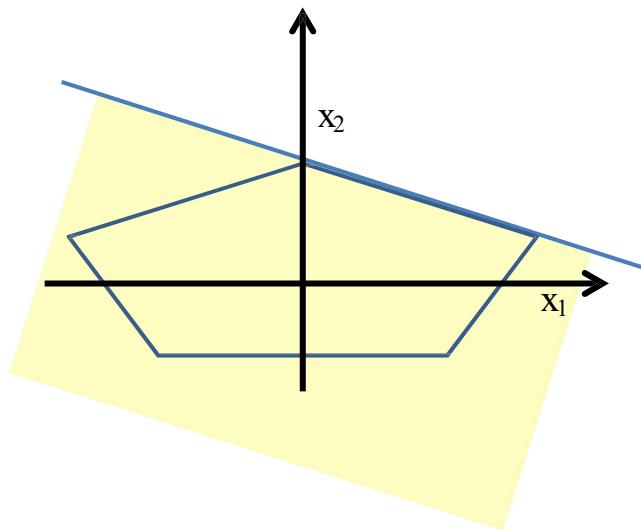
- The network must fire if the input is in the coloured area

# Booleans over the reals



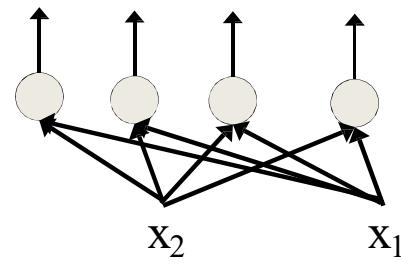
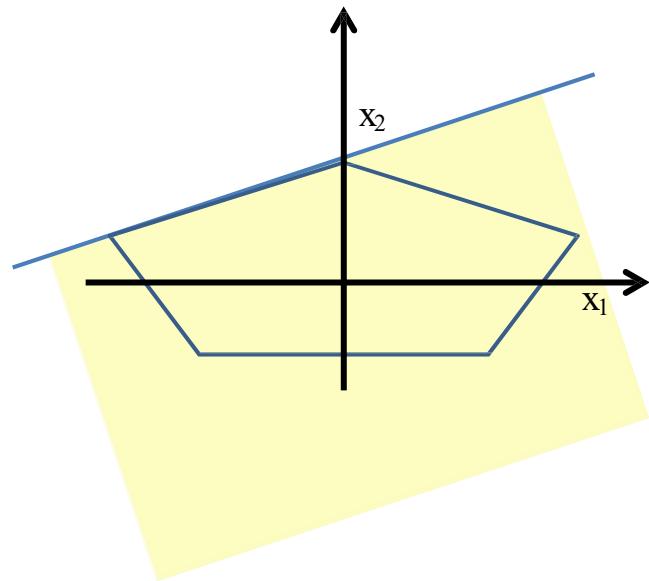
- The network must fire if the input is in the coloured area

# Booleans over the reals



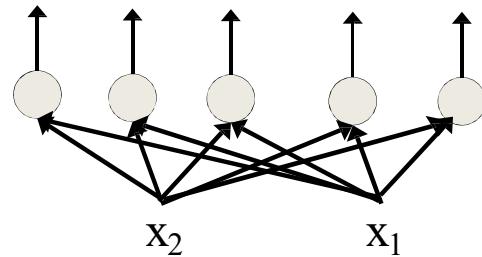
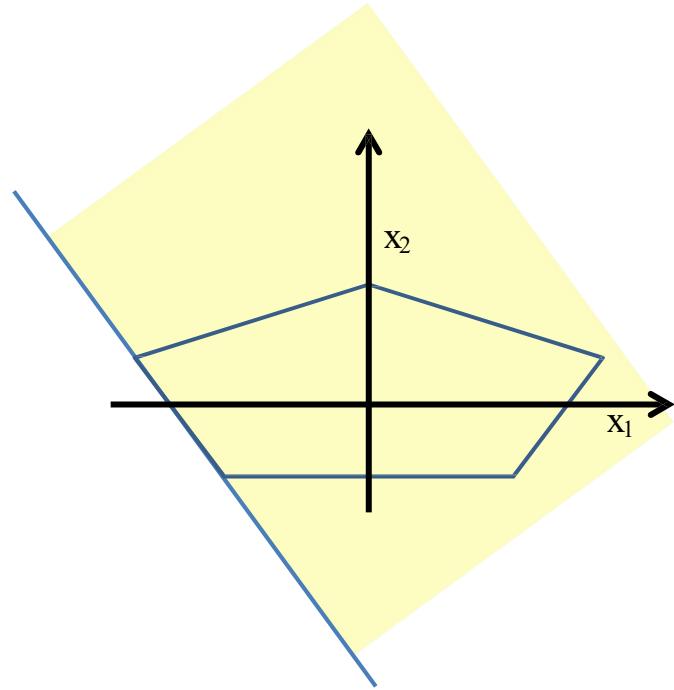
- The network must fire if the input is in the coloured area

# Booleans over the reals



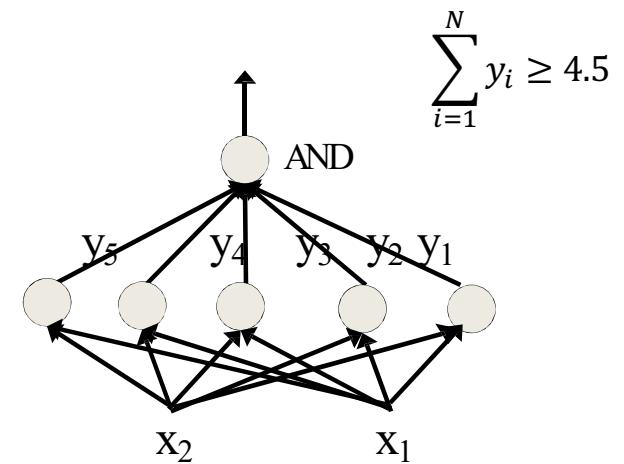
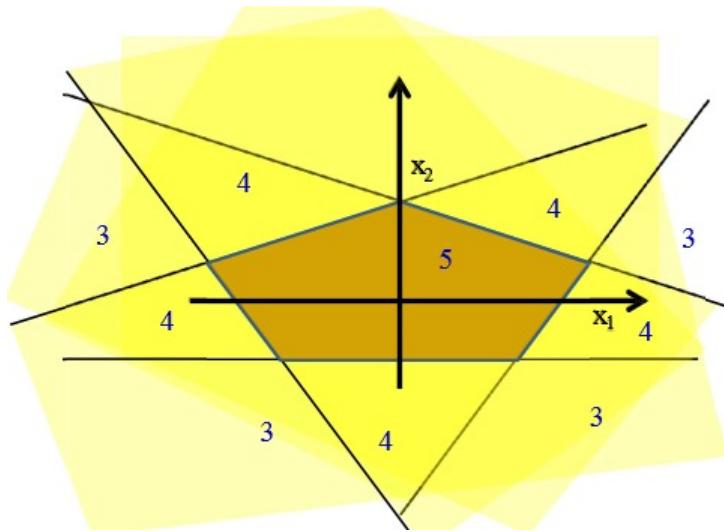
- The network must fire if the input is in the coloured area

# Booleans over the reals



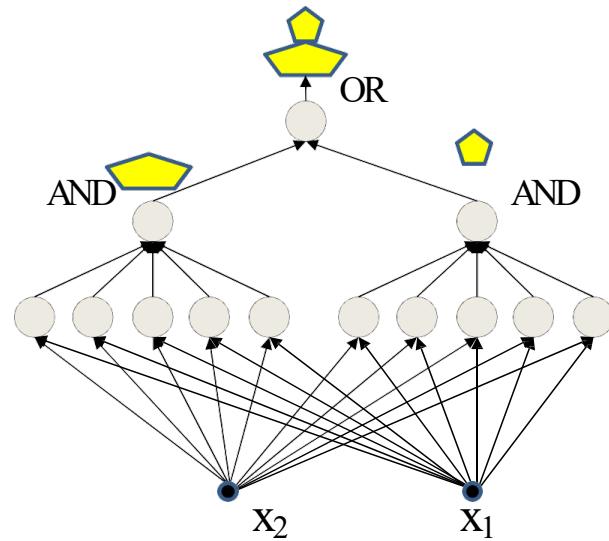
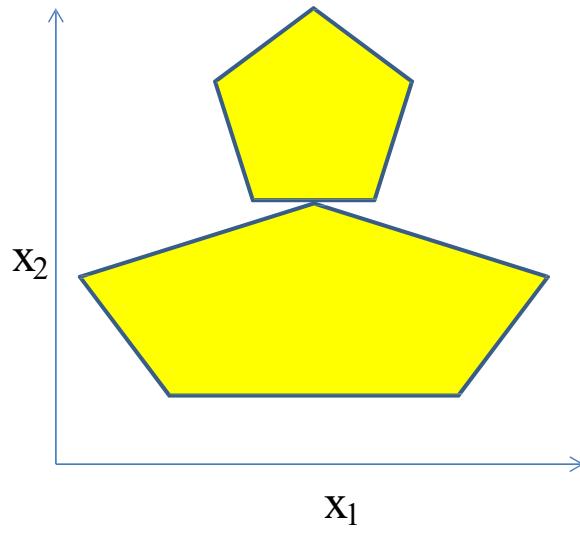
- The network must fire if the input is in the coloured area

# Booleans over the reals



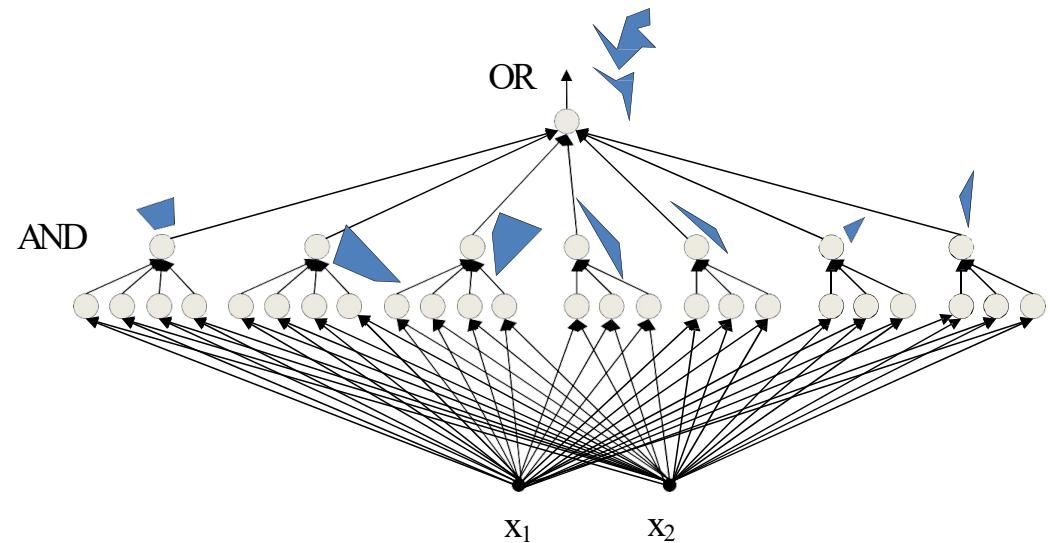
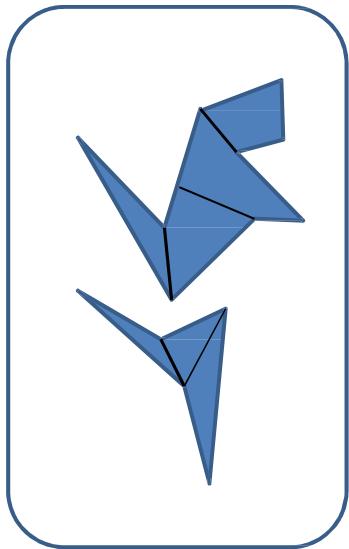
- The network must fire if the input is in the coloured area

# More complex decision boundaries



- Network to fire if the input is in the yellow area
  - “OR” two polygons
  - A third layer is required

# Complex decision boundaries



- Can compose arbitrarily complex decision boundaries
  - With only one hidden layer!
  - **How?**

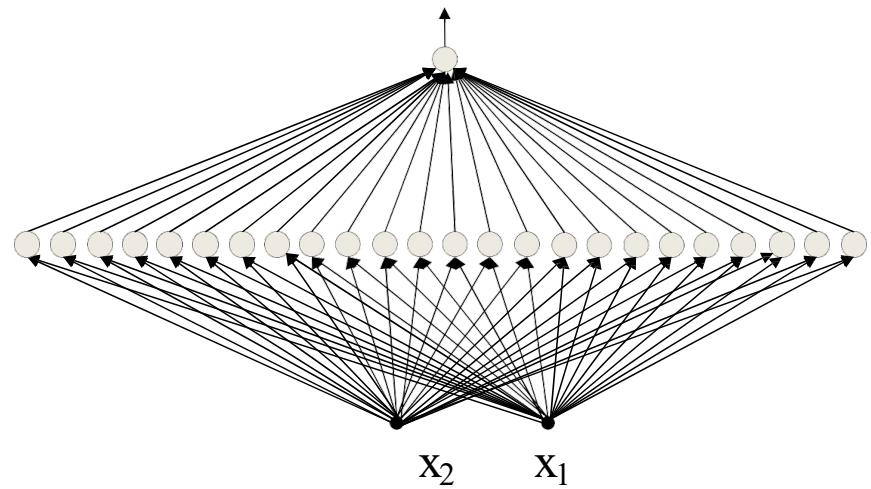
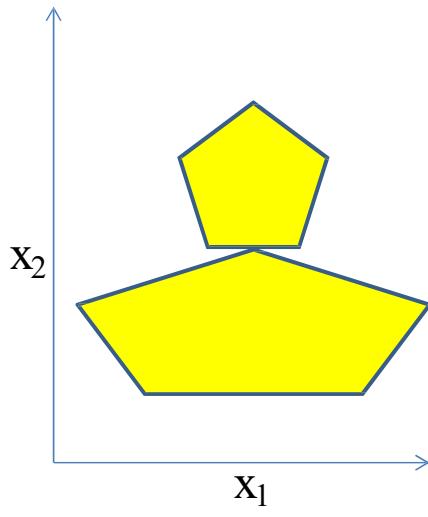
# MLP with Different Number of Layers

MLP with unit step activation function

Decision region found by an output unit.

Structure	Type of Decision Regions	Interpretation	Example of region
Single Layer (no hidden layer)	Half space	Region found by a hyper-plane	
Two Layer (one hidden layer)	Polyhedral (open or closed) region	Intersection of half spaces	
Three Layer (two hidden layers)	Arbitrary regions	Union of polyhedrals	

Exercise: compose this with one hidden layer

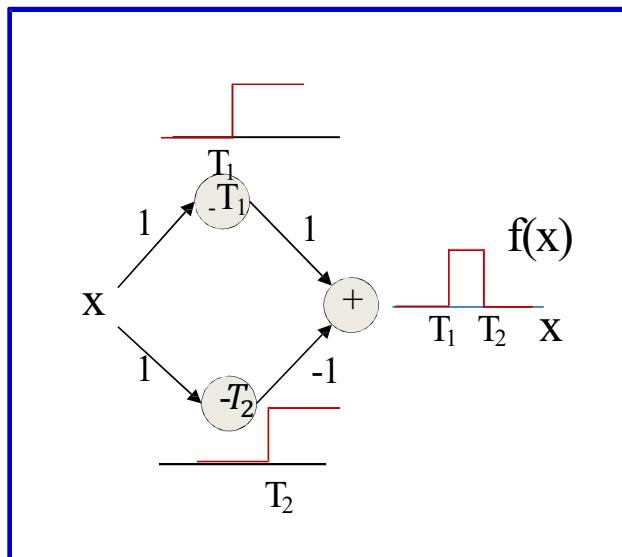


- How would you compose the decision boundary to the left with only one hidden layer?

# MLPs approximate functions

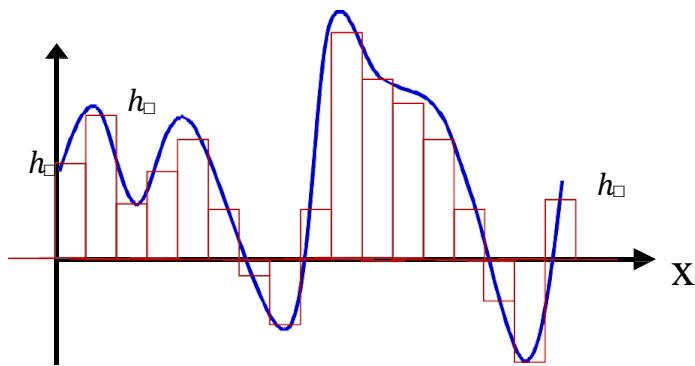
- MLPs can compose Boolean functions
- MLPs as universal classifiers
- MLPs as universal approximators (of real-valued functions)

# MLPs as a continuous-valued regression



- A simple 3-unit MLP with a “summing” output unit can generate a “square pulse” over an input
  - Output is 1 only if the input lies between  $T_1$  and  $T_2$
  - $T_1$  and  $T_2$  can be arbitrarily specified

# MLP as a continuous-valued regression



- A simple 3-unit MLP can generate a “square pulse” over an input
- An MLP with many units can model an arbitrary function over an input
  - To arbitrary precision
    - Simply make the individual pulses narrower
- A one-layer MLP can model an arbitrary function of a single input

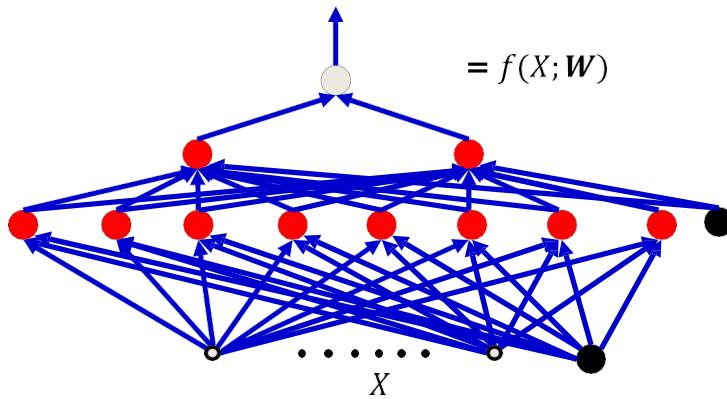
# Summary

- MLPs are universal Boolean function
- MLPs are universal classifiers
- MLPs are universal function approximators
- An MLP with two (or even one) hidden layers can approximate anything to arbitrary precision
  - But could be exponentially or even infinitely wide in its inputs size

# How to adjust weights for multi layer networks?

- **How can we train such multi-layer networks?**
  - We need to adapt all the weights, not just the last layer.
  - adapting the weights entering hidden units is equivalent to learning features.
    - seems difficult to learn them since the target output of hidden units is not specified (only we access the target output of the whole network).

# What we learn : The parameter of the network



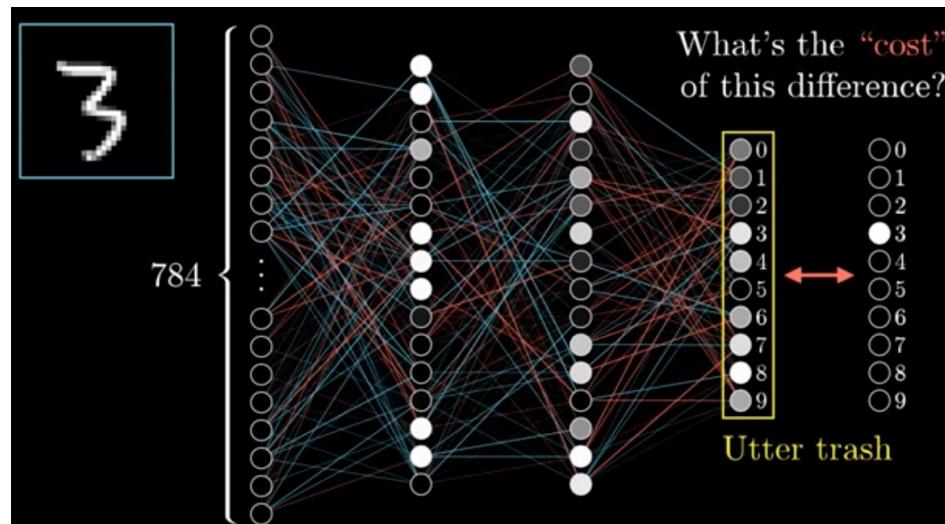
- Given: the architecture of the network
- The parameters of the network: The weights and biases
  - The weights associated with the blue arrows in the picture
- Learning the network: Determining the values of these parameters such that the network computes the desired function

# Training multi-layer networks

- **Back-propagation**
  - Training algorithm that is used to adjust weights in multi-layer networks
    - The backpropagation algorithm is based on gradient descent
      - The direction of the most rapid decrease in the cost function
    - Use chain rule to efficiently compute gradients

# Find the weights by optimizing the cost

- Start from random weights and then adjust them iteratively to get lower cost.
- Update the weights according to the gradient of the cost function



Source: <http://3b1b.co>

# Learning problem

- Given: the architecture of the network
- Training data: A set of input-output pairs

$$(\mathbf{x}^{(1)}, \mathbf{y}^{(1)}), (\mathbf{x}^{(2)}, \mathbf{y}^{(2)}), \dots, (\mathbf{x}^{(N)}, \mathbf{y}^{(N)})$$

- We want to find the function  $g$  on the input space to get the output
  - We consider a neural network as a parametric function  $g(\mathbf{x}; \mathbf{W})$

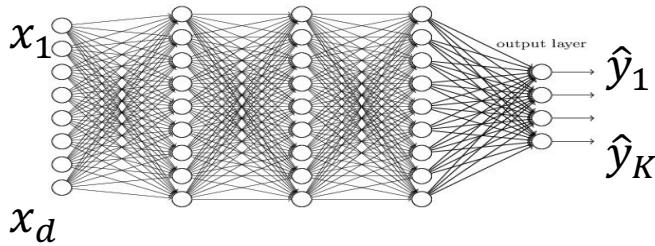
# Problem setup

- Given: the architecture of the network
- Training data: A set of input-output pairs  
 $(\mathbf{x}^{(1)}, \mathbf{y}^{(1)}), (\mathbf{x}^{(2)}, \mathbf{y}^{(2)}), \dots, (\mathbf{x}^{(N)}, \mathbf{y}^{(N)})$
- We want to find the function  $f$ 
  - We consider a neural network as a parametric function  $g(\mathbf{x}; \mathbf{W})$
- We need a **loss function** to show how penalizes the obtained output  $g(\mathbf{x}; \mathbf{W})$  when the desired output is  $\mathbf{y}$

$$\frac{1}{N} \sum_{n=1}^N loss(g(\mathbf{x}^{(n)}; \mathbf{W}), \mathbf{y}^{(n)})$$

# Choosing cost function: Examples

- Regression problem
  - SSE



$$E = \sum_{n=1}^N E_n$$

$$E_n = \sum_{k=1}^K (\hat{y}_k^{(n)} - y_k^{(n)})^2$$

- Classification problem
  - Cross-entropy

$$\text{loss}_n = \sum_{k=1}^K -y_k^{(n)} \log \hat{y}_k^{(n)}$$

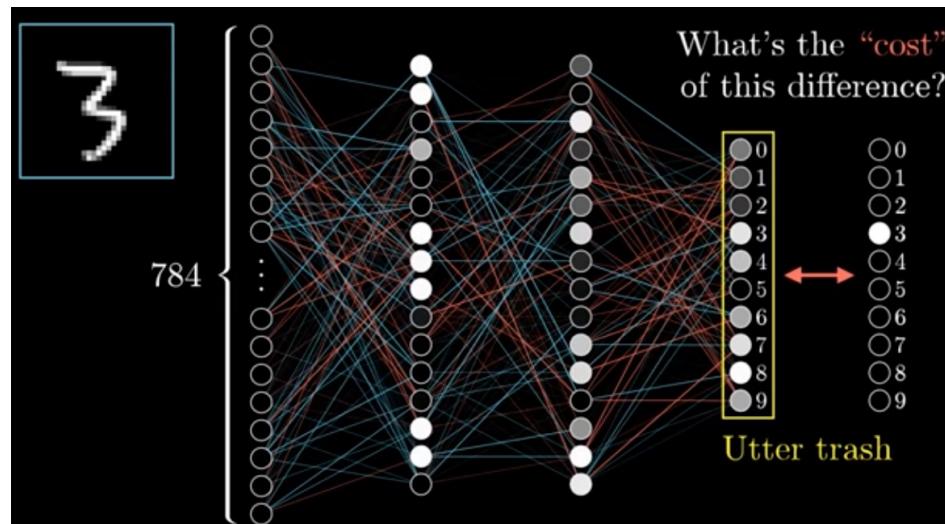
Output is found by a softmax layer  $\hat{y}_k = \frac{e^{z_k}}{\sum_{k=1}^K e^{z_k}}$

# How to adjust weights for multi layer networks?

- We need multiple layers of adaptive, non-linear hidden units. But how can we train such nets?
  - We need an efficient way of adapting all the weights, not just the last layer.
  - Learning the weights going into hidden units is equivalent to learning features.
  - This is difficult because nobody is telling us directly what the hidden units should do.

# Find the weights by optimizing the cost

- Start from random weights and then adjust them iteratively to get lower cost.
- Update the weights according to the gradient of the cost function

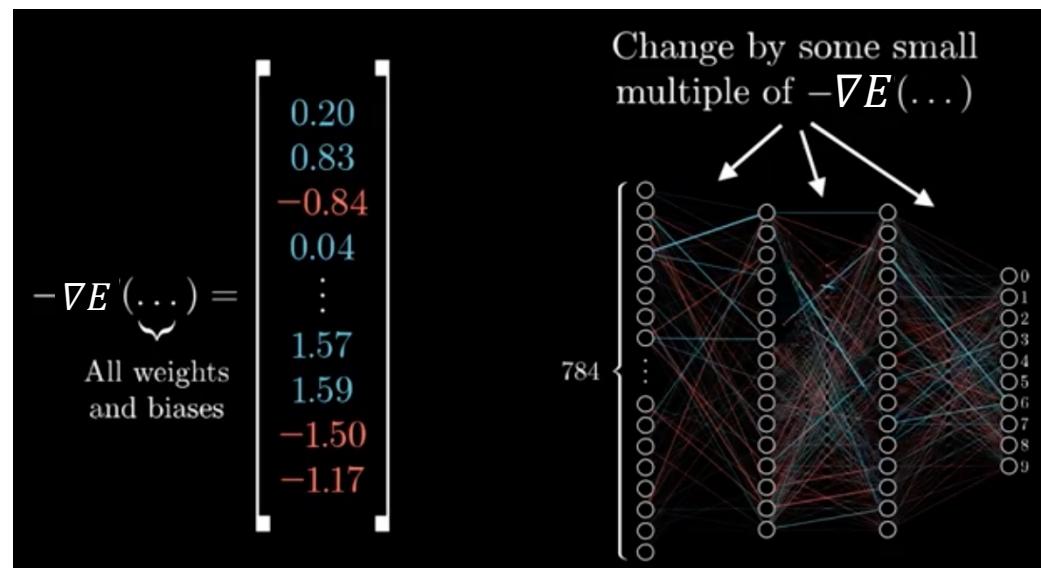


Source: <http://3b1b.co>

# How does the network learn?

- Which changes to the weights do improve the most?

Source: <http://3b1b.co>



- The magnitude of each element shows how sensitive the cost is to that weight or bias.

# Training multi-layer networks

- Back-propagation
  - Training algorithm that is used to adjust weights in multi-layer networks (based on the training data)
  - The back-propagation algorithm is based on gradient descent
  - Use chain rule and dynamic programming to efficiently compute gradients

# Training Neural Nets through Gradient Descent

$$E = \sum_{n=1}^N loss(\hat{\mathbf{y}}^{(n)}, \mathbf{y}^{(n)})$$

Total training error:

- Gradient descent algorithm
- Initialize all weights and biases  $\{w_{ij}^{[k]}\}$ 
  - Using the extended notation : the bias is also weight
- Do :
  - For every layer  $k$  for all  $i, j$  update:
    - $w_{i,j}^{[k]} = w_{i,j}^{[k]} - \eta \frac{dE}{dw_{i,j}^{[k]}}$
- Until  $E$  has converged

Assuming the bias is also represented as a weight

# The derivative

Total training error:

$$E = \sum_{n=1}^N loss(\hat{y}^{(n)}, y^{(n)})$$

Computing the derivative •

Total derivative:

$$\frac{dE}{dw_{i,j}^{[k]}} = \sum_{n=1}^N \frac{dloss(\hat{y}^{(n)}, y^{(n)})}{dw_{i,j}^{[k]}}$$

# Training by gradient descent

- Initialize all weights  $\{w_{ij}^{[k]}\}$
- Do :
  - For all  $i, j, k$ , initialize  $\frac{dE}{dw_{i,j}^{[k]}} = 0$
  - For all  $n = 1:N$ 
    - For every layer  $k$  for all  $i, j$ :
      - Compute  $\frac{d \text{loss}(\hat{y}^{(n)}, y^{(n)})}{dw_{i,j}^{[k]}}$
      - $\frac{dE}{dw_{i,j}^{[k]}} += \frac{d \text{loss}(\hat{y}^{(n)}, y^{(n)})}{dw_{i,j}^{[k]}}$
    - For every layer  $k$  for all  $i, j$ :

$$w_{i,j}^{[k]} = w_{i,j}^{[k]} - \frac{\eta}{T} \frac{dE}{dw_{i,j}^{[k]}}$$

# Returning to our problem

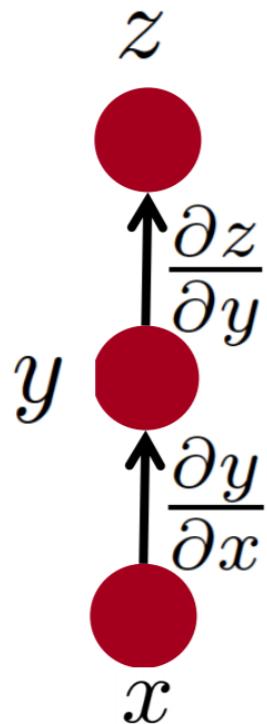
How to compute •  $\frac{d \text{ loss}(\hat{\mathbf{y}}, \mathbf{y})}{dw_{i,j}^{[k]}}$

# Training multi-layer networks

- **Back-propagation**
  - Training algorithm that is used to adjust weights in multi-layer networks
    - The backpropagation algorithm is based on gradient descent
      - The direction of the most rapid decrease in the cost function
  - **Use chain rule to efficiently compute gradients**

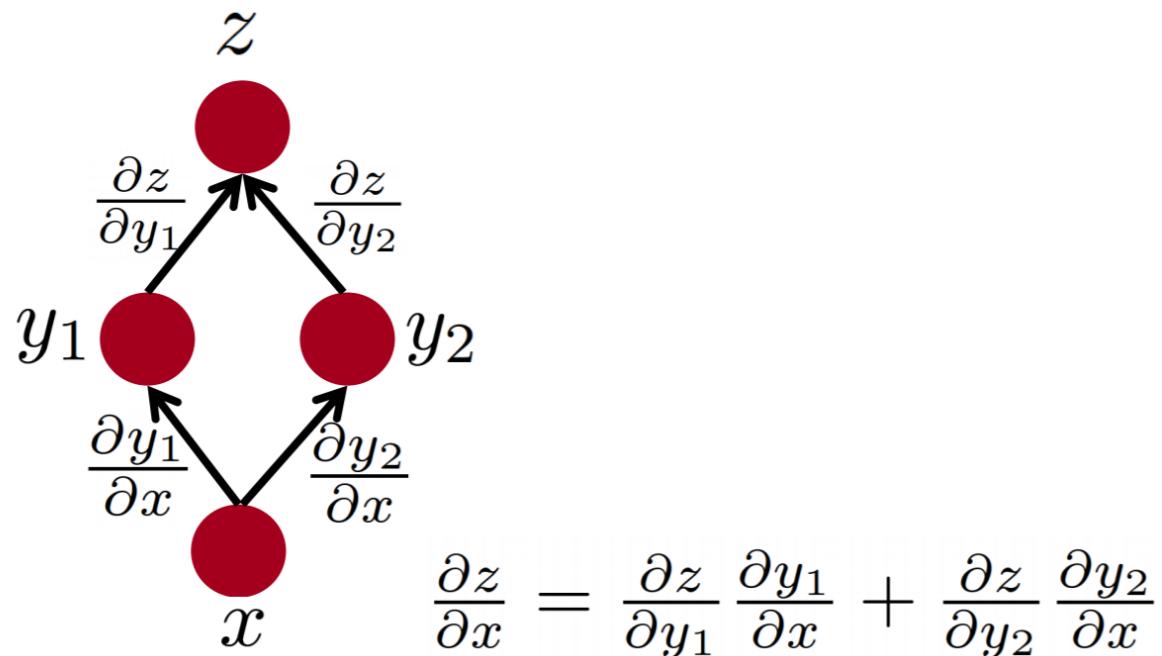
# Simple chain rule

- $z = f(g(x))$
- $y = g(x)$



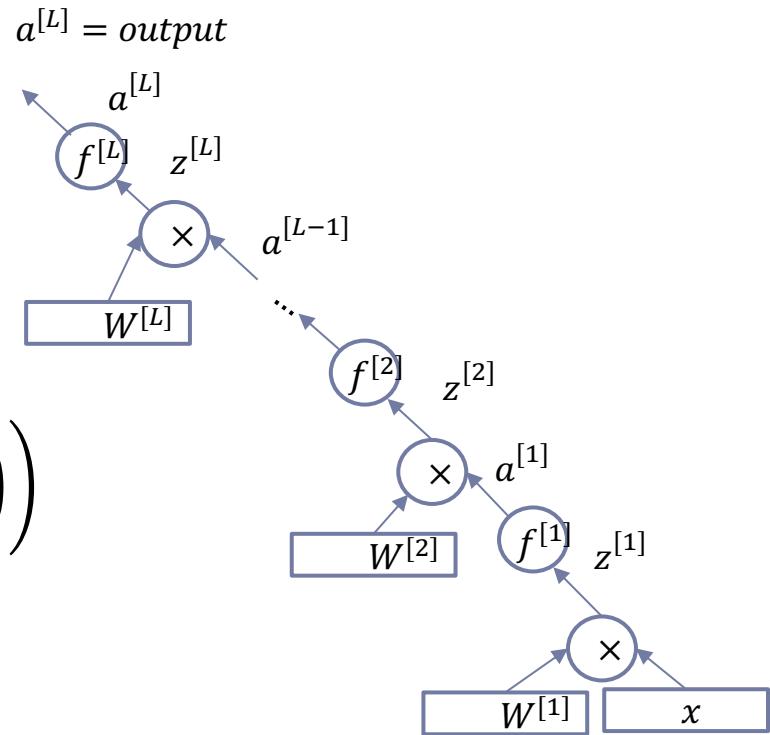
$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y} \frac{\partial y}{\partial x}$$

# Multiple paths chain rule



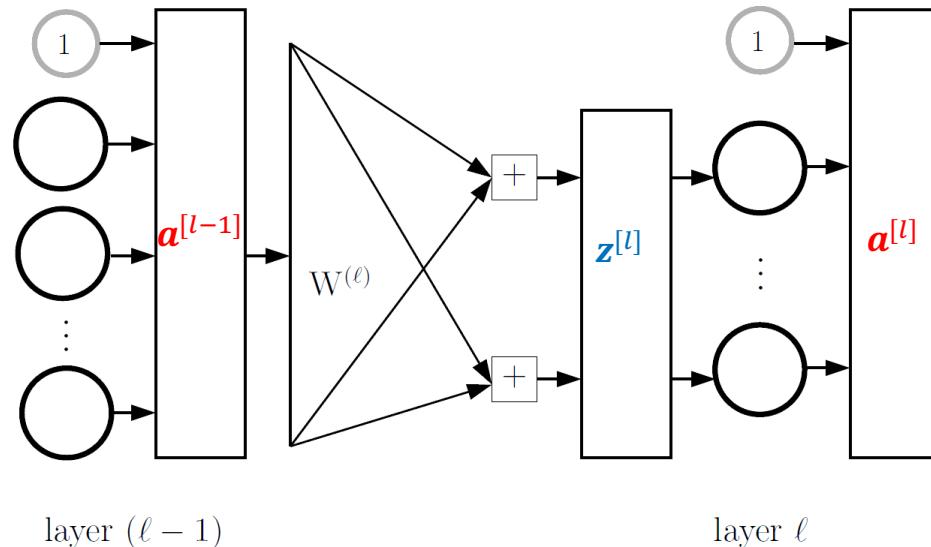
# Multi-layer network

$$\begin{aligned} \text{Output} &= a^{[L]} \\ &= f(z^{[L]}) \\ &= f(W^{[L]}a^{[L-1]}) \\ &= f(W^{[L]}f(W^{[L-1]}a^{[L-2]})) \\ &= f\left(W^{[L]}f\left(W^{[L-1]} \dots f\left(W^{[2]}f\left(W^{[1]}x\right)\right)\right)\right) \end{aligned}$$



# Backpropagation: Notation

- $a^{[0]} \leftarrow Input$
- $output \leftarrow a^{[L]}$



# Backpropagation: Last layer gradient

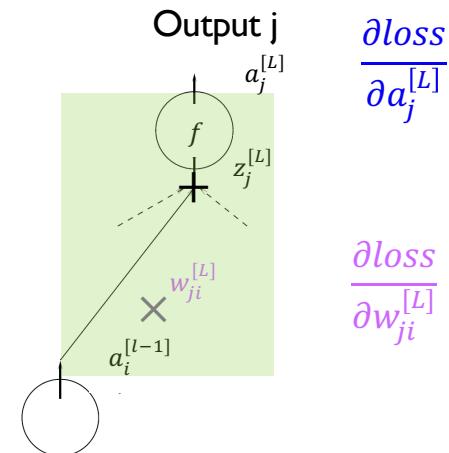
For squared error loss:

$$\begin{aligned} loss &= \frac{1}{2} \sum_j (o_j - y_j)^2 \\ o_j &= a_j^{[L]} \end{aligned}$$

$$\frac{\partial loss}{\partial a_j^{[L]}} = (a_j^{[L]} - y_j)$$

$$\begin{aligned} a_i^{[L]} &= f(z_i^{[L]}) \\ z_j^{[L]} &= \sum_{i=0}^M w_{ji}^{[L]} a_i^{[L-1]} \end{aligned}$$

$$\frac{\partial loss}{\partial w_{ji}^{[L]}} = ?$$



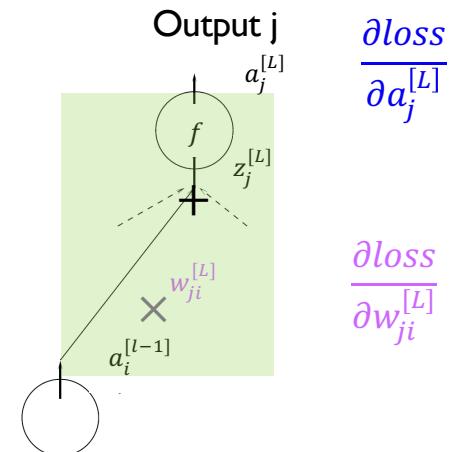
# Backpropagation: Last layer gradient

For squared error loss:

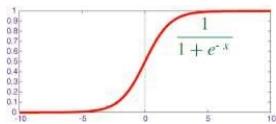
$$\begin{aligned} \text{loss} &= \frac{1}{2} \sum_j (o_j - y_j)^2 & \frac{\partial \text{loss}}{\partial a_j^{[L]}} &= (a_j^{[L]} - y_j) \\ o_j &= a_j^{[L]} & a_j^{[L]} &= f(z_j^{[L]}) \\ && z_j^{[L]} &= \sum_{i=0}^M w_{ji}^{[L]} a_i^{[L-1]} \end{aligned}$$

$$\frac{\partial \text{loss}}{\partial w_{ji}^{[L]}} = \frac{\partial \text{loss}}{\partial a_j^{[L]}} f'(z_j^{[L]}) \frac{\partial z_j^{[L]}}{\partial w_{ji}^{[L]}}$$

$$= \frac{\partial \text{loss}}{\partial a_j^{[L]}} f'(z_j^{[L]}) a_i^{[L-1]}$$

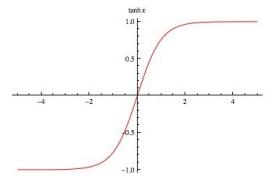


# Activations and their derivatives



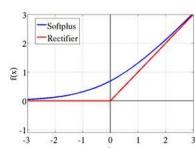
$$f(z) = \frac{1}{1 + \exp(-z)}$$

$$f'(z) = f(z)(1 - f(z))$$



$$f(z) = \tanh(z)$$

$$f' f'(z) = 1 - f^2(z)$$



$$f(z) = \begin{cases} 0, & z < 0 \\ z, & z \geq 0 \end{cases}$$

$$f'(z) = \begin{cases} 1, & z \geq 0 \\ 0, & z < 0 \end{cases}$$

$$f(z) = \log(1 + \exp(z))$$

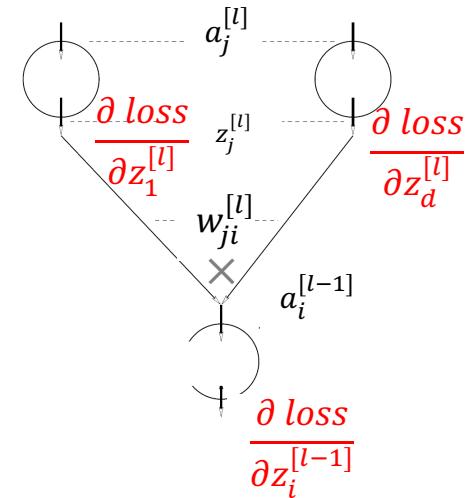
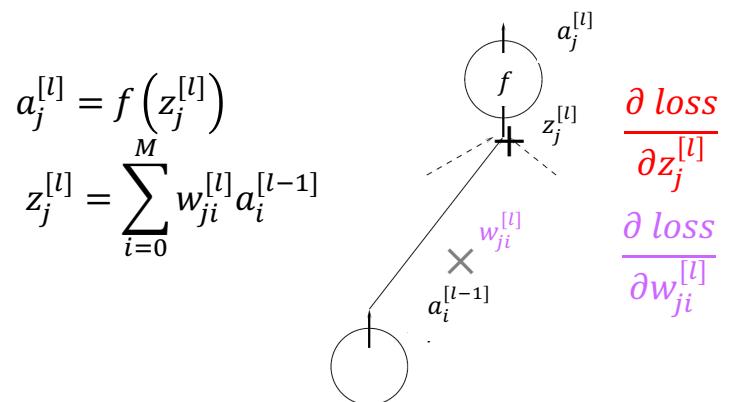
$$f'(z) = \frac{1}{1 + \exp(-z)}$$

- Some popular activation functions and their derivatives

# Previous layers gradients

$$\frac{\partial \text{loss}}{\partial w_{ji}^{[l]}} = \frac{\partial \text{loss}}{\partial z_j^{[l]}} \frac{\partial z_j^{[l]}}{\partial w_{ji}^{[l]}} = \frac{\partial \text{loss}}{\partial z_j^{[l]}} a_i^{[l-1]}$$

$$\begin{aligned} \frac{\partial \text{loss}}{\partial z_i^{[l-1]}} &= \frac{\partial a_i^{[l-1]}}{\partial z_i^{[l-1]}} \sum_{j=1}^{d^{[l]}} \frac{\partial \text{loss}}{\partial z_j^{[l]}} \times \frac{\partial z_j^{[l]}}{\partial a_i^{[l-1]}} \\ &= f'(z_i^{[l-1]}) \sum_{j=1}^{d^{[l]}} \frac{\partial \text{loss}}{\partial z_j^{[l]}} \times w_{ji}^{[l]} \end{aligned}$$

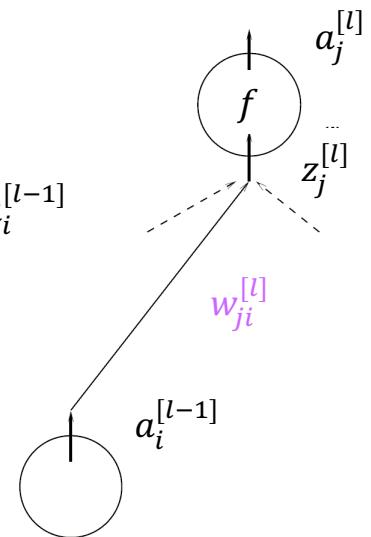


# Backpropagation:

$$\begin{aligned}\frac{\partial \text{loss}}{\partial w_{ji}^{[l]}} &= \boxed{\frac{\partial \text{loss}}{\partial z_j^{[l]}}} \times \frac{\partial z_j^{[l]}}{\partial w_{ji}^{[l]}} \\ &= \boxed{\delta_j^{[l]}} \times a_i^{[l-1]}\end{aligned}$$

$$a_j^{[l]} = f(z_j^{[l]})$$

$$z_j^{[l]} = \sum_{i=0}^M w_{ji}^{[l]} a_i^{[l-1]}$$



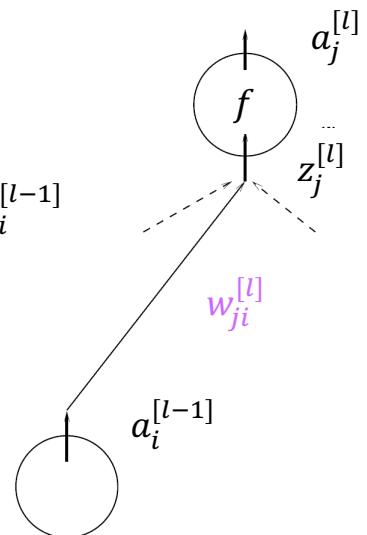
# Backpropagation:

$$\begin{aligned}\frac{\partial \text{loss}}{\partial w_{ji}^{[l]}} &= \boxed{\frac{\partial \text{loss}}{\partial z_j^{[l]}}} \times \frac{\partial z_j^{[l]}}{\partial w_{ji}^{[l]}} \\ &= \boxed{\delta_j^{[l]}} \times a_i^{[l-1]}\end{aligned}$$

- ▶  $\delta_j^{[l]} = \frac{\partial \text{loss}}{\partial z_j^{[l]}}$  is the **sensitivity** of the loss to  $z_j^{[l]}$
- ▶ Sensitivity vectors can be obtained by running a backward process in the network architecture (hence the name backpropagation.)  
We will compute  $\delta_i^{[l-1]}$  from  $\delta_j^{[l]}$ :

$$\delta_i^{[l-1]} = f'(z_i^{[l-1]}) \sum_{j=1}^{d^{[l]}} \delta_j^{[l]} \times w_{ji}^{[l]}$$

$$\begin{aligned}a_j^{[l]} &= f(z_j^{[l]}) \\ z_j^{[l]} &= \sum_{i=0}^M w_{ji}^{[l]} a_i^{[l-1]}\end{aligned}$$



# Backward process on sensitivity vectors

- For the final layer  $l = L$ :

$$\delta_j^{[L]} = \frac{\partial \text{loss}}{\partial z_j^{[L]}}$$

- Compute  $\delta^{[l-1]}$  from  $\delta^{[l]}$ : by running a backward process in the network architecture:

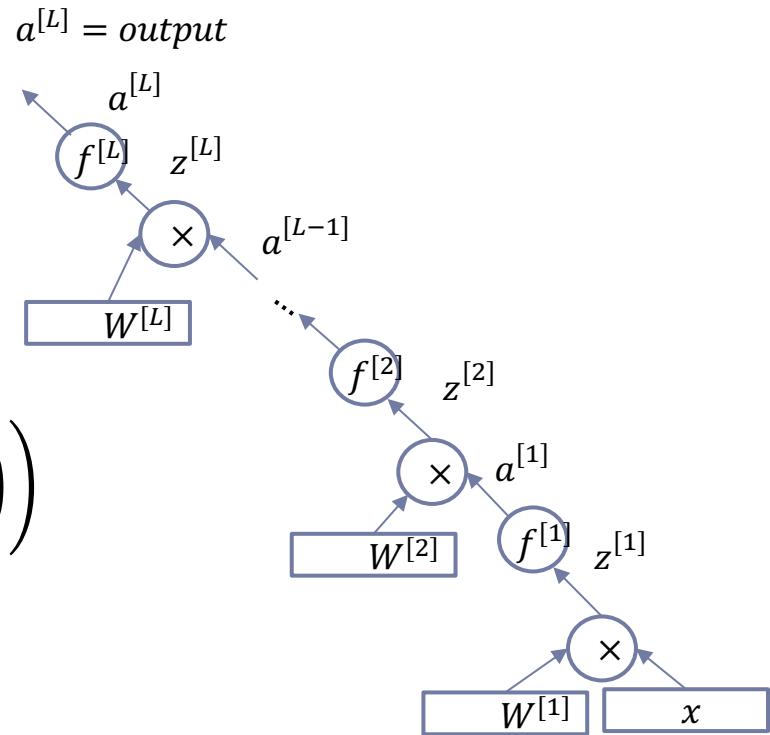
$$\delta_i^{[l-1]} = f' \left( z_i^{[l-1]} \right) \sum_{j=1}^{d^{[l]}} \delta_j^{[l]} \times w_{ji}^{[l]}$$

# Backpropagation Algorithm

- Initialize all weights to small random numbers.
- **While not satisfied**
- **For** each training example **do**:
  1. Feed forward the training example to the network and compute the outputs of all units in forward step ( $z$  and  $a$ ) and the loss
  2. For each unit find its  $\delta$  in the backward step
  3. Update each network weight  $w_{ji}^{[l]}$  as  $w_{ji}^{[l]} \leftarrow w_{ji}^{[l]} - \eta \frac{\partial \text{loss}}{\partial w_{ji}^{[l]}}$  where  $\frac{\partial \text{loss}}{\partial w_{ji}^{[l]}} = \delta_j^{[l]} \times a_i^{[l-1]}$

# Multi-layer network: Matrix notation

$$\begin{aligned} \text{Output} &= a^{[L]} \\ &= f(z^{[L]}) \\ &= f(W^{[L]}a^{[L-1]}) \\ &= f(W^{[L]}f(W^{[L-1]}a^{[L-2]})) \\ &= f\left(W^{[L]}f\left(W^{[L-1]} \dots f\left(W^{[2]}f\left(W^{[1]}x\right)\right)\right)\right) \end{aligned}$$



# Multi-layer network: Matrix notation

$$Output = a^{[L]}$$

$$= f(z^{[L]})$$

$$= f(W^{[L]}a^{[L-1]})$$

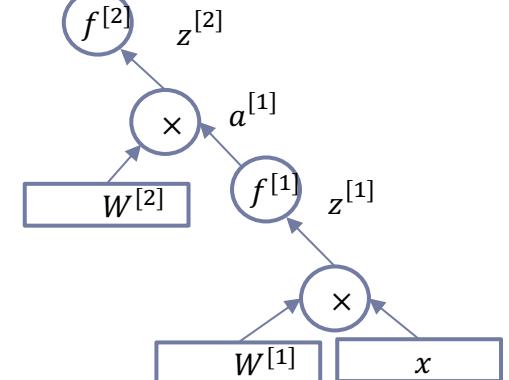
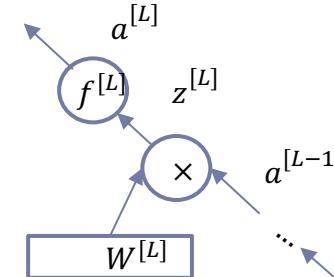
$$= f(W^{[L]}f(W^{[L-1]}a^{[L-2]}))$$

$$= f\left(W^{[L]}f\left(W^{[L-1]} \dots f\left(W^{[2]}f\left(W^{[1]}x\right)\right)\right)\right)$$

$$\frac{\partial loss}{\partial W^{[l]}} = \frac{\partial loss}{\partial z^{[l]}} a^{[l-1]^T}$$

$$\frac{\partial loss}{\partial z^{[l]}} = f'(z^{[l]})W^{[l+1]^T} \frac{\partial loss}{\partial z^{[l+1]}}$$

$$a^{[L]} = output$$



# Mini-batch gradient descent

- Large datasets
  - Divide dataset into smaller batches containing one subset of the main training set
  - Weights are updated after seeing training data in each of these batches

# Gradient descent methods

Stochastic gradient

Stochastic mini-batch gradient

Batch gradient

Batch size=1

e.g., Batch size= 32, 64, 128, 256

Batch size=n  
(the size of training set)

n: whole no of training data

bs: the size of batches

$m = \left\lceil \frac{n}{bs} \right\rceil$ : the number of batches

Batch 1 $X^{\{1\}}, Y^{\{1\}}$	Batch 2 $X^{\{2\}}, Y^{\{2\}}$								Batch m $X^{\{m\}}, Y^{\{m\}}$

# Mini-batch gradient descent

For epoch=1,...,k

For t=1,...,m

Forward propagation on  $X^{\{t\}}$

$$J^{\{t\}} = \frac{1}{m} \sum_{n \in Batch_t} L(\hat{Y}_n^{\{t\}}, Y_n^{\{t\}}) + \lambda R(W)$$

Backpropagation on  $J^{\{t\}}$  to compute gradients  $dW$

For  $l = 1, \dots, L$

$$W^{[l]} = W^{[l]} - \alpha dW^{[l]}$$

I epoch:  
Single pass  
over all  
training  
samples

$$\begin{aligned} A^{[0]} &= X^{\{t\}} \\ \text{For } l &= 1, \dots, L \\ Z^{[l]} &= W^{[l]} A^{[l-1]} \\ A^{[l]} &= f^{[l]}(Z^{[l]}) \\ \hat{Y}_n^{\{t\}} &= A_n^{[L]} \end{aligned}$$

Vectorized computation

Batch 1 $X^{\{1\}}, Y^{\{1\}}$	Batch 2 $X^{\{2\}}, Y^{\{2\}}$								Batch m $X^{\{m\}}, Y^{\{m\}}$

# Training issues

- The backpropagation algorithm is an efficient way of computing the derivative of the cost function w.r.t. each of the weights
- However, many issues must be considered to have successful training:
  - Optimization issues
  - Generalization issues

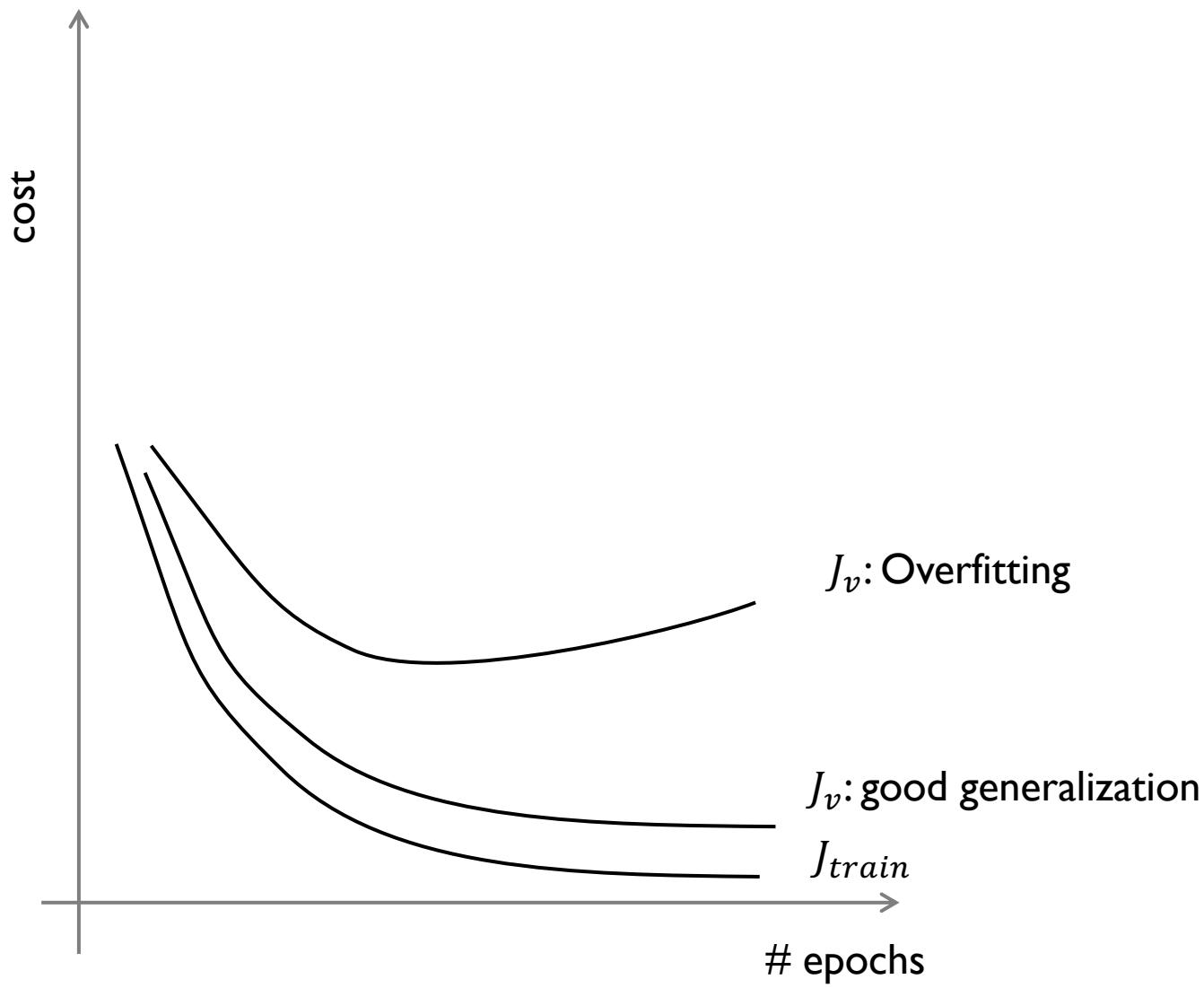
# Generalization techniques

- Regularization or weight-decay
- Hyper-parameter tuning
- Weight-sharing
  - e.g., CNNs
- Model ensemble
- Pre-training
- Data augmentation
- Dropout
- Batch Normalization
- ...

# Hyperparameter tuning of NNs

- **Architecture**
  - Type of layers
  - Activation functions
  - Number of layers
  - Number of hidden units in each layer
  - ...
- **Optimization**
  - Optimizer
  - Batch size
  - Learning rate
  - ...

# Generalization



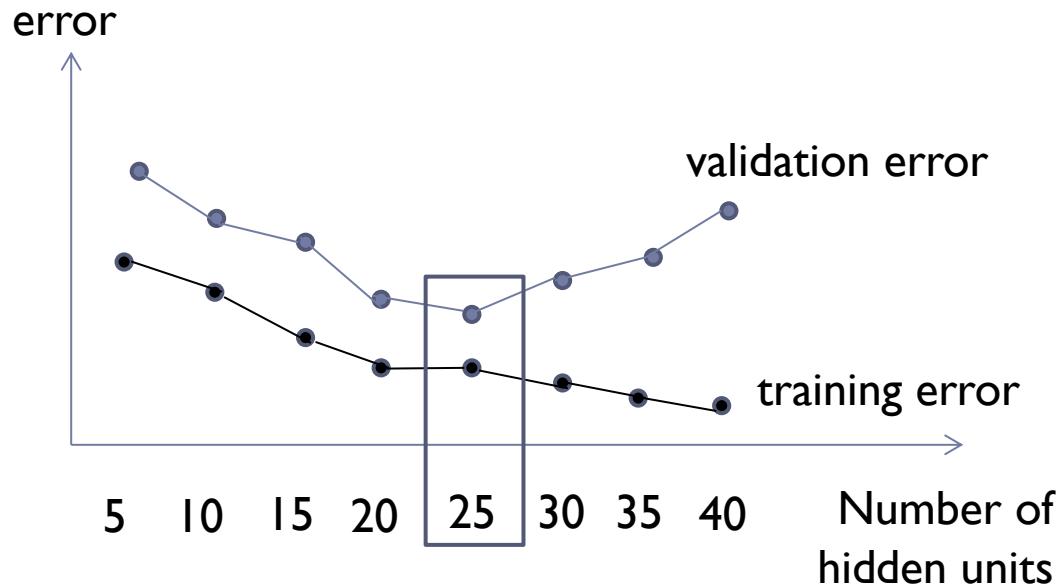
# Regularization

$$J(W) = \frac{1}{N} \sum_{n=1}^N L^{(n)}(W) + \lambda R(W)$$

- $R(W)$ : is defined based on the norm of the weights vectors
  - Example:  $R(W) = \sum_l \sum_{i,j} w_{ij}^{[l]} {}^2$

# Hyper-parameter tuning: Example Number of Hidden Units

- Shows the expressive power the network
  - Can specify the total numbers of weights that are the number of freedom degree
- Select among networks with different no. of hidden units by training these networks and then evaluating them on a validation set
  - For large networks and large training set, it is inefficient.

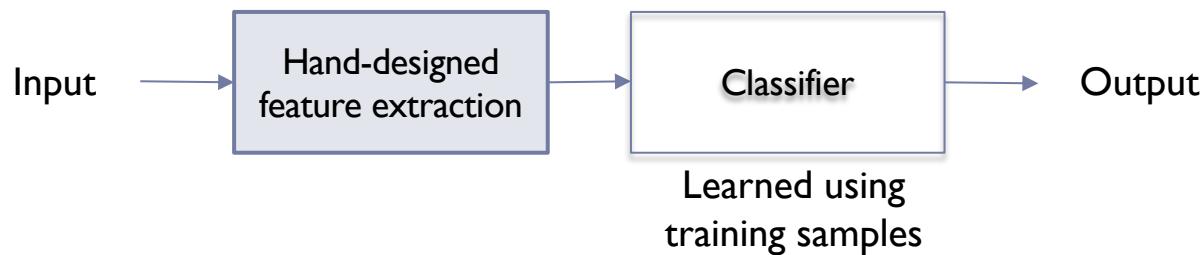


# Deep Learning

- Learning a computational models consists of multiple processing layers
  - learn representations of data with multiple levels of abstraction.
- Dramatically improved the state-of-the-art in many speech, vision and NLP tasks (and also in many other domains)

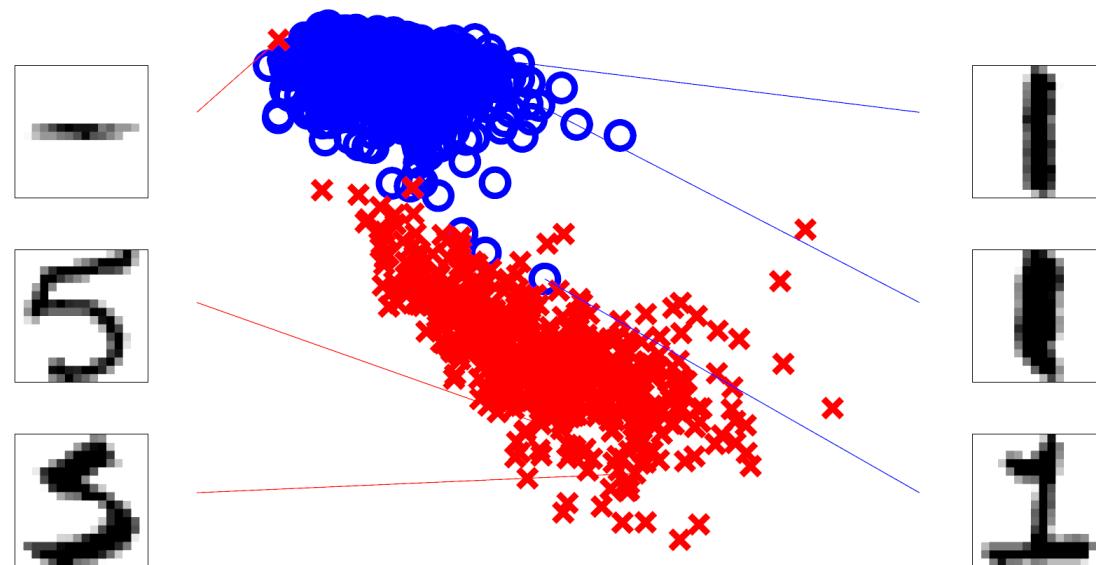
# Machine Learning Methods

- Conventional machine learning methods:
  - try to learn the mapping from the input features to the output by samples
  - However, they need appropriately designed hand-designed features



# Example

- $x_1$ : intensity
- $x_2$ : symmetry



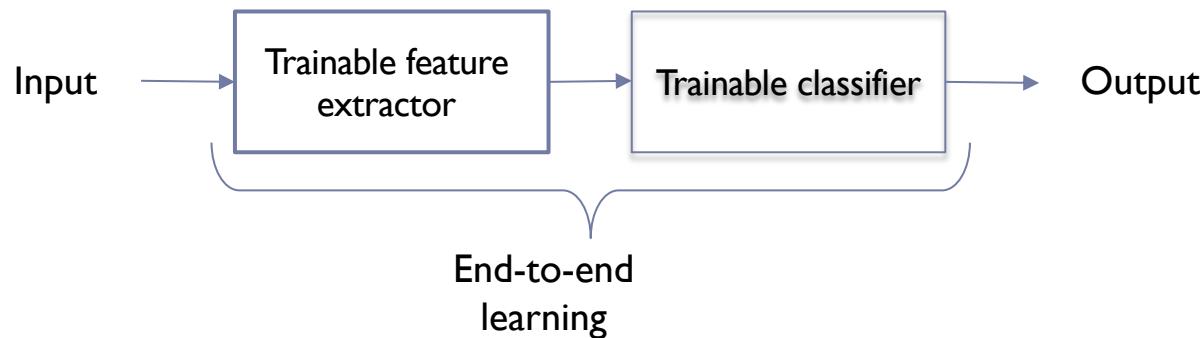
[Abu Mostafa, 2012]

# Representation of Data

- Performance of traditional learning methods depends heavily on the representation of the data.
  - **Most efforts were on designing proper features**
- However, designing hand-crafted features for inputs like image, videos, time series, and sequences is not trivial at all.
  - It is difficult to know which features should be extracted.
    - Sometimes, it needs long time for a community of experts to find (an incomplete and over-specified) set of these features.

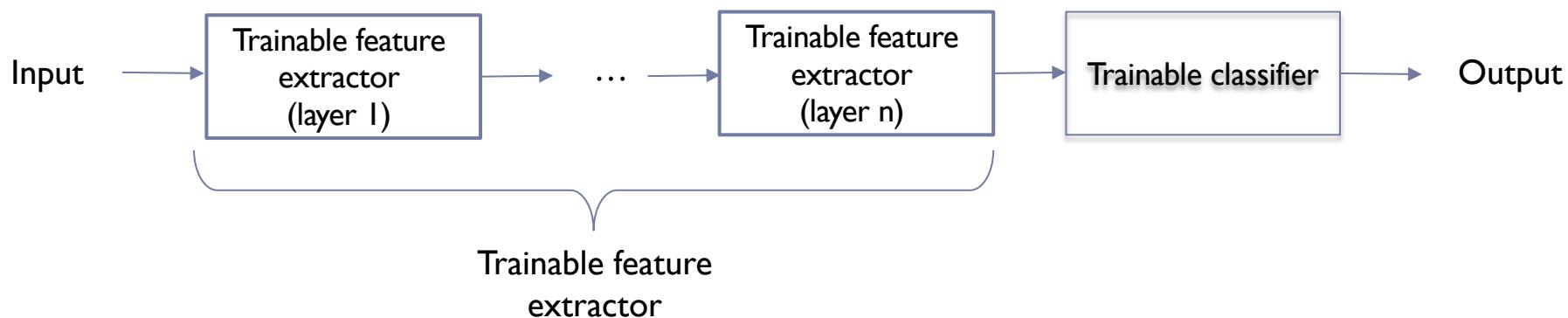
# Representation Learning

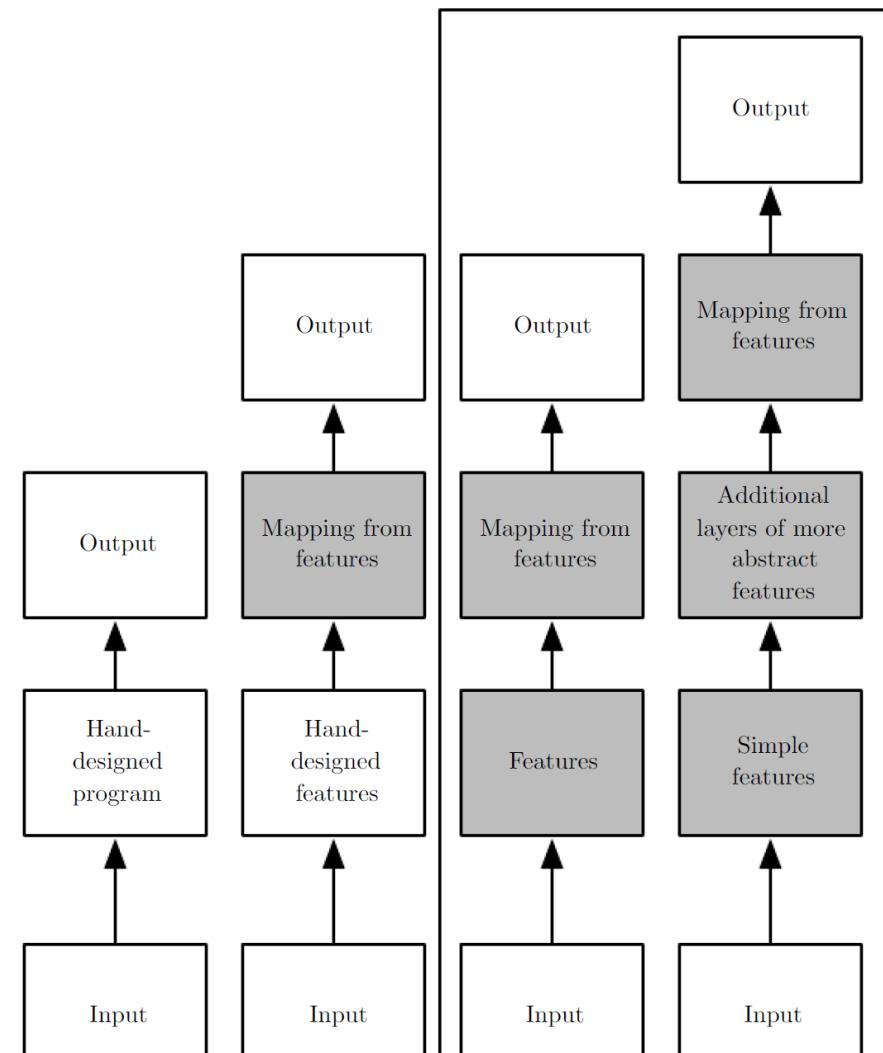
- Using learning to discover both:
  - the representation of data from input features
  - and the mapping from representation to output



# Deep Learning Approach

- Deep breaks the desired complicated mapping into a series of nested simple mappings
  - each mapping described by a layer of the model.
  - each layer extracts features from output of previous layer
- shows impressive performance on many Artificial Intelligence tasks





[Deep Learning book]

# Deep Representations: The Power of Compositionality

- **Compositionality is useful to describe the world efficiently**
  - Learned function seen as a composition of simpler operations
  - Hierarchy of features, concepts, leading to more abstract factors enabling better generalization
    - each concept defined in relation to simpler concepts
    - more abstract representations computed in terms of less abstract ones.
  - Again, theory shows this can be exponentially advantageous
- Deep learning has great power and flexibility by learning to represent the world as a nested hierarchy of concepts

# Deep learning

- Use networks with **many layers**
- A single hidden layer with enough units can approximate any target network
  - More layers more closely mimics human learning
  - We may need far less number of nodes when we use deep networks
- A hierarchy of internal representations for the input.
  - The first layer constructs a low-level representation;
  - More complex representations in terms of simpler representation of the previous layer

# Boolean functions

- Input: N Boolean variable
- How many neurons in a one hidden layer MLP is required?
- More compact representation of a Boolean function
  - “Karnaugh Map”
    - representing the truth table as a grid
    - Grouping adjacent boxes to reduce the complexity of the Disjunctive Normal Form (DNF) formula

	X, Y 00	01	10	11
W, Z 00	I	I	I	I
01				
10		I	I	
11	I			I

# Worst case

- Which truth tables cannot be reduced further simply?
- Largest width needed for a single-layer Boolean network on  $N$  inputs
  - Worst case:  $2^{N-1}$ 
    - Example: Parity function

~~$X, Y$~~   
 $W, Z$

	00	01	11	10
00	1	0	1	0
01	0	1	0	1
11	1	0	1	0
10	0	1	0	1

$X \oplus Y \oplus Z \oplus W$

# Using deep network: Parity function on N inputs

- Simple MLP with one hidden layer:

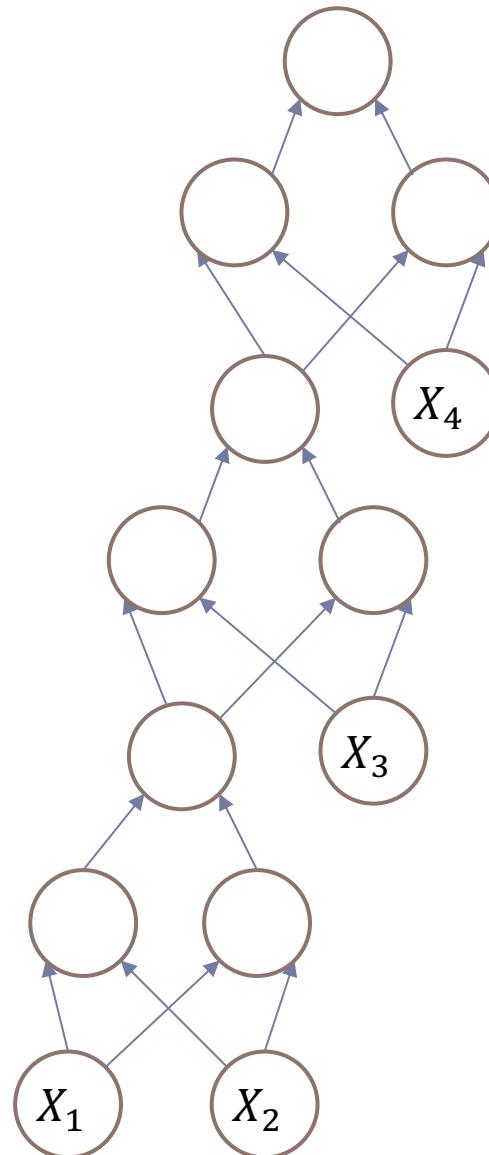
$2^{N-1}$  Hidden units

$(N + 2)2^{N-1}$  Weights and biases

- $f = X_1 \oplus X_2 \oplus \dots \oplus X_N$

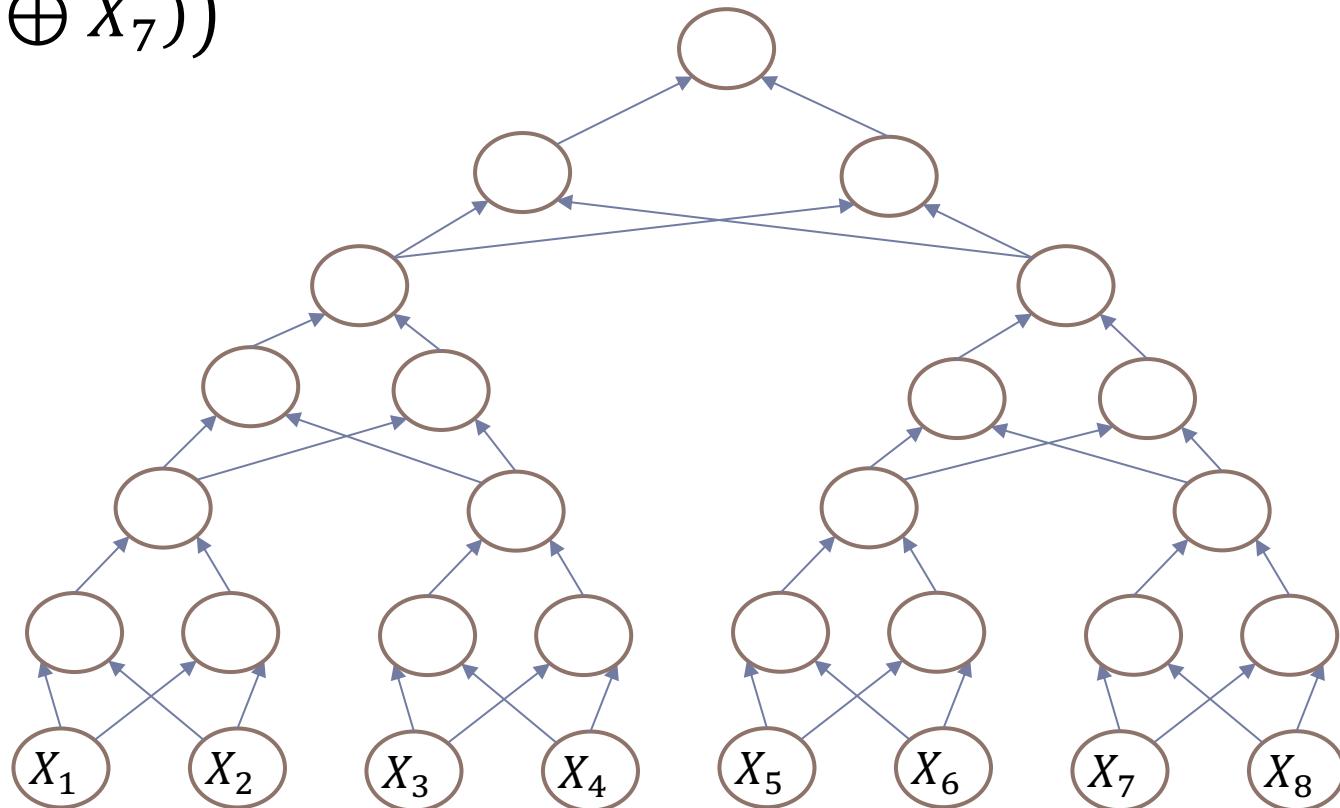
$3(N - 1)$  Hidden nodes

$9(N - 1)$  Weights and biases



# A better architecture

- Only requires  $2\log N$  layers
- $f = ((X_1 \oplus X_2) \oplus (X_3 \oplus X_4)) \oplus ((X_4 \oplus X_5) \oplus (X_6 \oplus X_7))$

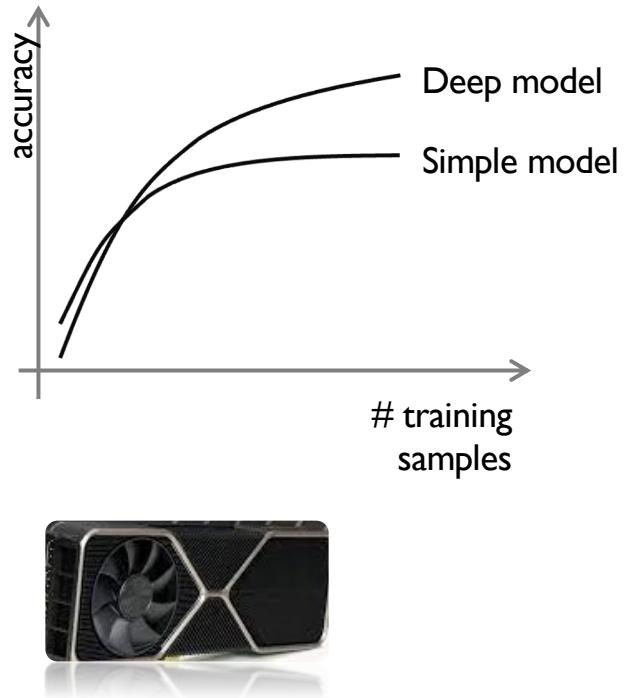


# Boolean function: Wide vs. deep network

- MLP with a single hidden layer is a universal Boolean function
- However, a single-layer network might need an exponential number of hidden units w.r.t. the number of inputs
- Deeper networks may require far fewer neurons than the single hidden layer network
  - Linear w.r.t. the number of inputs when that is deep enough

# Why does deep learning become popular?

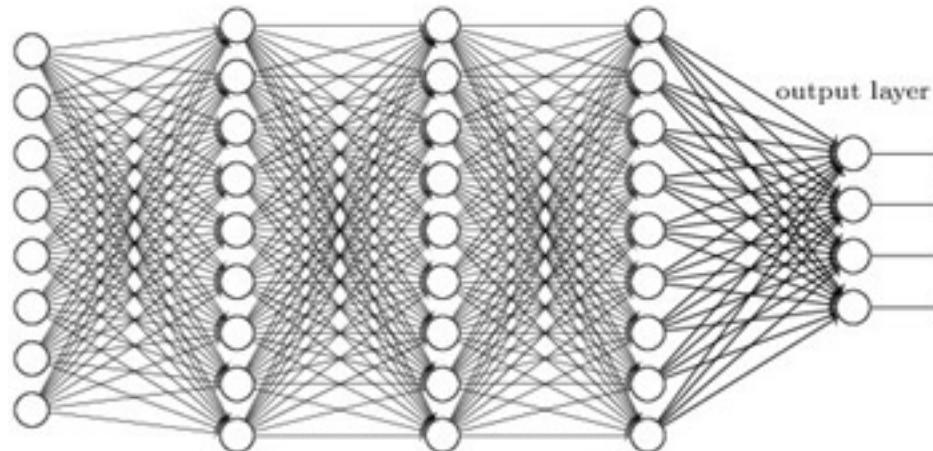
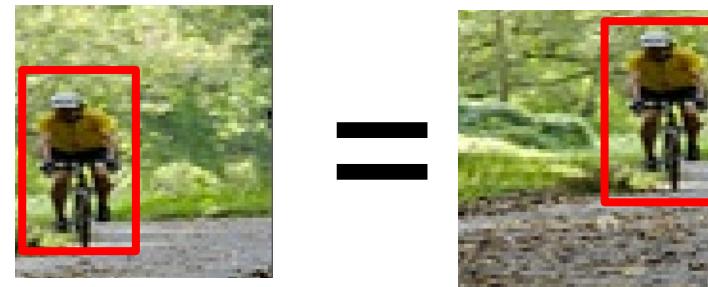
- Data: Large datasets
- Hardware: Availability of the computational resources to run much larger models
- Algorithm
  - New training techniques
  - New models
  - Frameworks



# Deep learning architectures

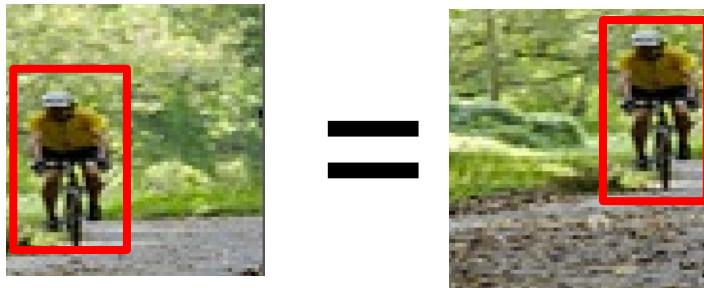
- Fully connected
- Convolutional Neural Networks (CNNs)
- Recurrent Neural Networks (RNNs)
- Transformers

# A problem of fully connected layers



- Is MLP proper for classifying images in which objects may have different locations?

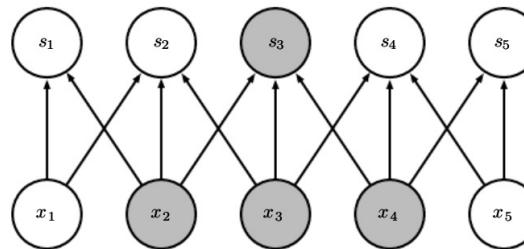
# The need for *shift invariance*



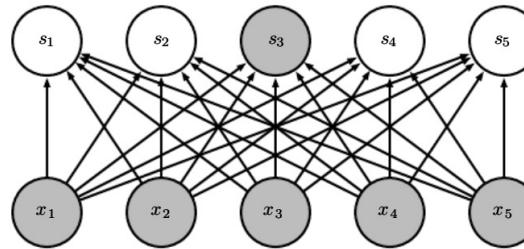
- In many problems the *location* of a pattern is not important
  - Only the presence of the pattern
- MLPs are sensitive to the location of the pattern
- Requirement: Network must be *shift invariant*

# Convolutional layers

- Locality and weight sharing



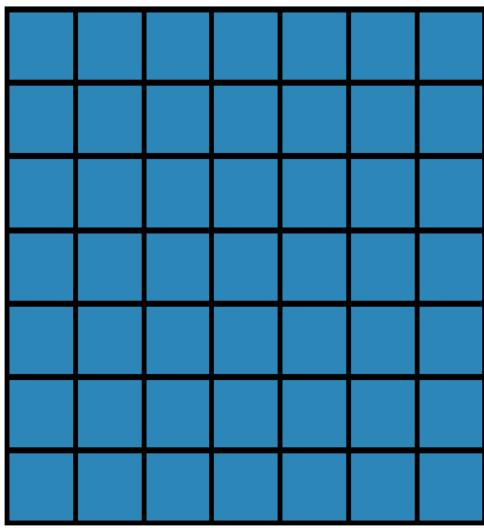
convolution



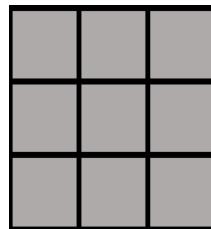
fully connected

[Goodfellow et al. 2016]

# Convolutional filter

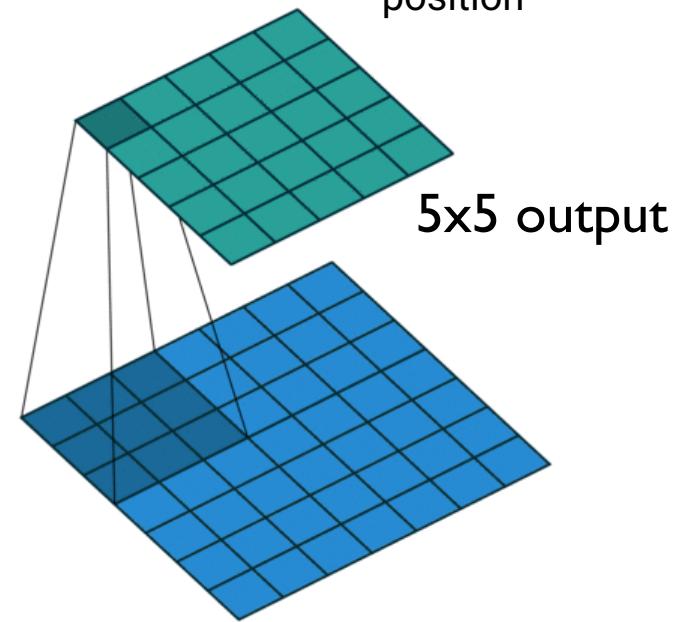


7x7 input



3x3 filter

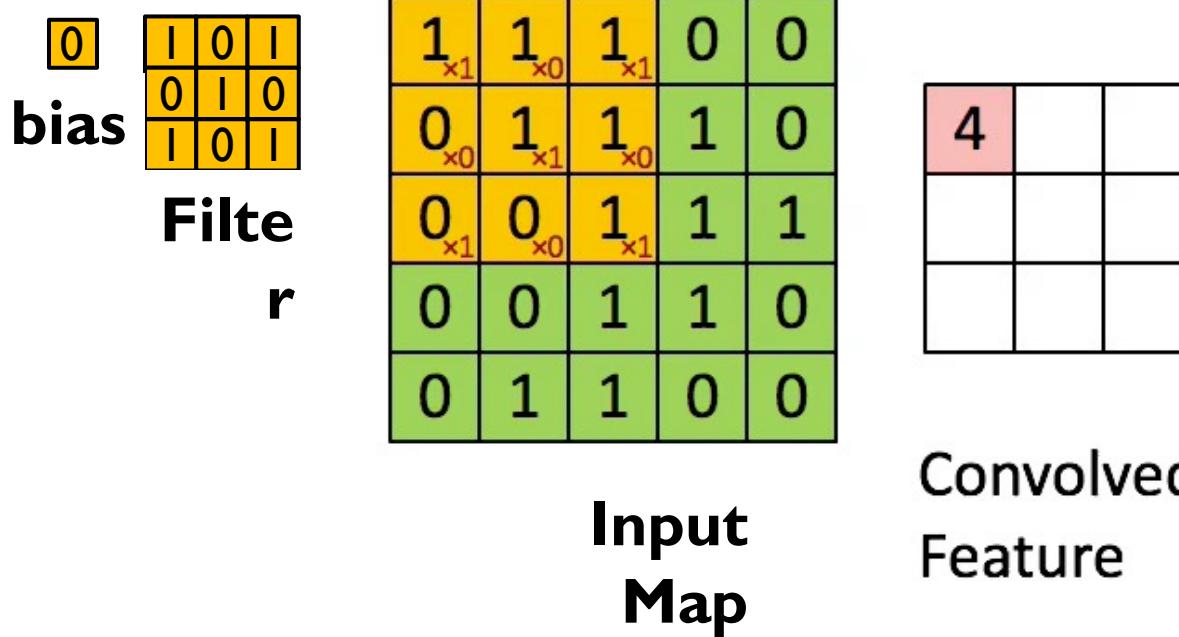
Gives the responses of that filter at every spatial position



Source:

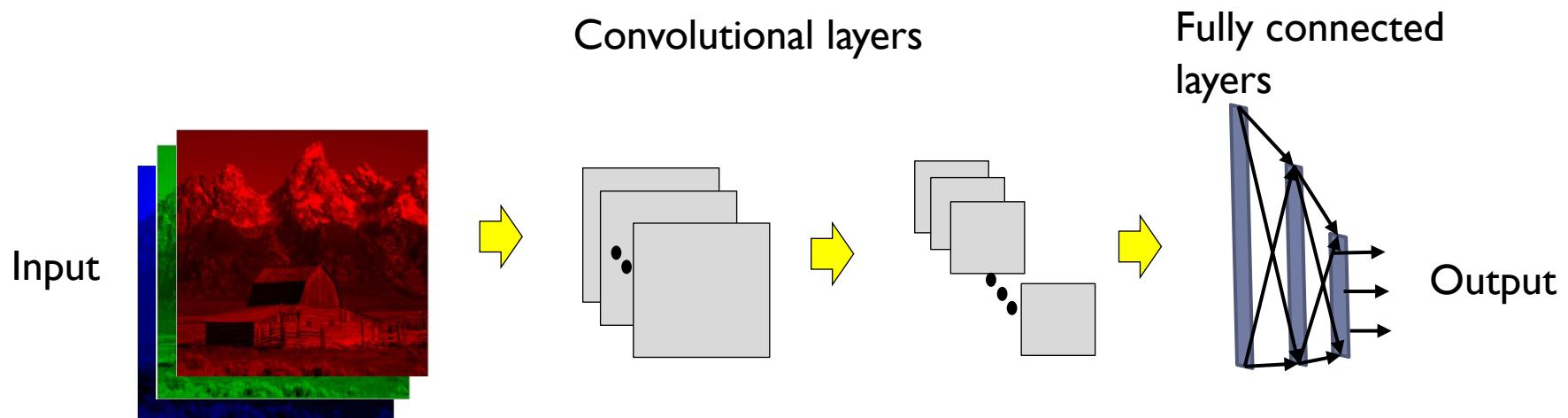
<http://iamaaditya.github.io/2016/03/one-by-one-convolution/>

# What is a convolution?



- Scanning an image with a “filter”
  - At each location, the “filter and the underlying map values are multiplied component wise, and the products are added along with the bias

# Convolutional Neural Network (CNN)



# Summary

- Neural nets are universal approximators
- Backpropagation is a training algorithm for neural nets
- Training issues must be considered
  - Optimization and generalization issues
- Convolutional layers as an example of inductive biases that improve generalization are introduced.