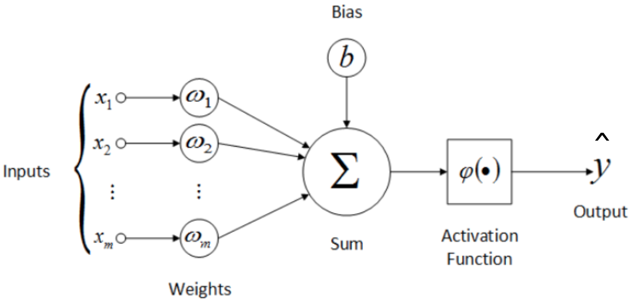
MAC 0460 / 5832
Introduction to Machine Learning

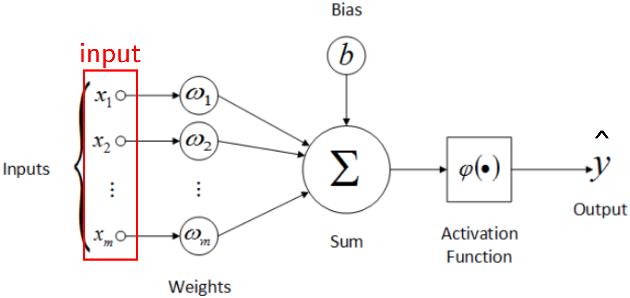
13 – Neural networks

IME/USP (26/05/2021)

Review: Logistic regression classifier

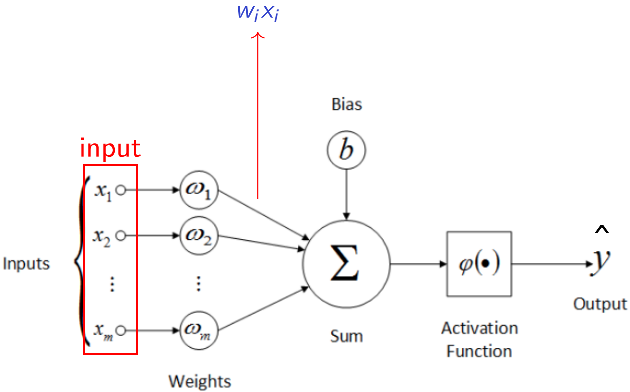


Review: Logistic regression classifier



Forward pass $\Longrightarrow$

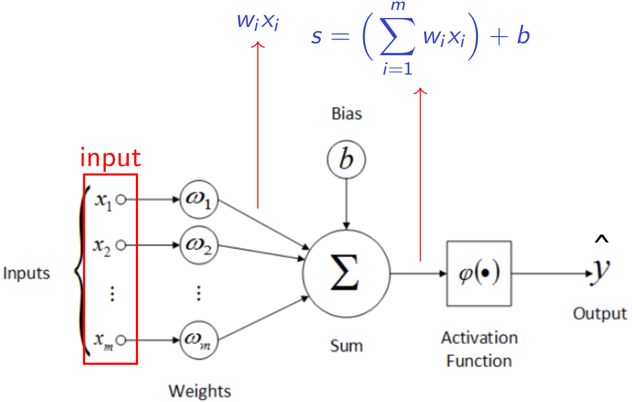
Review: Logistic regression classifier



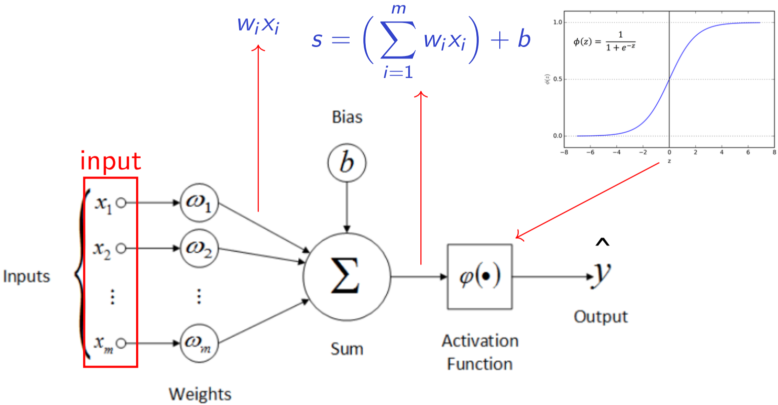
Forward pass $\Longrightarrow$

Review: Logistic regression classifier

$$s = \left(\sum_{i=1}^{m} w_i x_i \right) + b$$



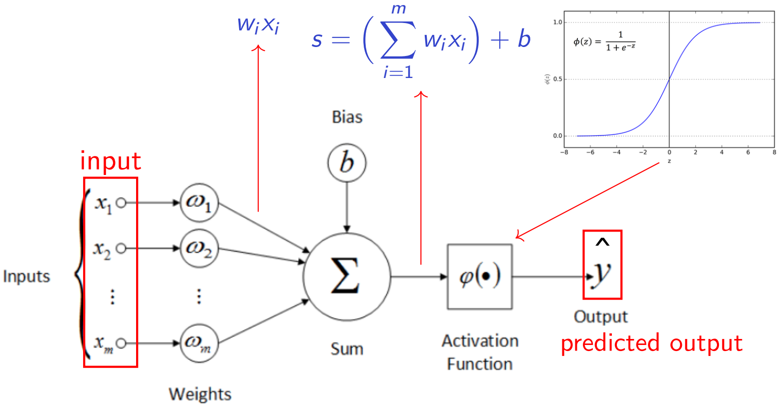
Forward pass $\Longrightarrow$



Forward pass $\Longrightarrow$

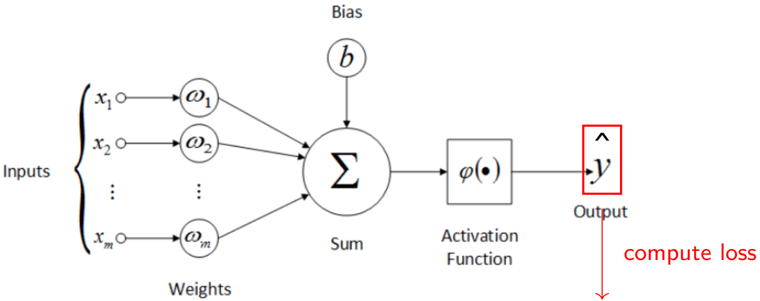
Review: Logistic regression classifier

$$s = \left(\sum_{i=1}^{m} w_i x_i\right) + b$$



Forward pass

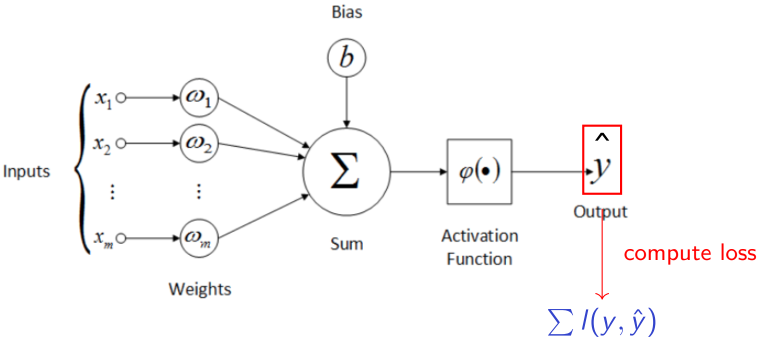
Review: Logistic regression classifier



$$\sum l(y, \hat{y})$$

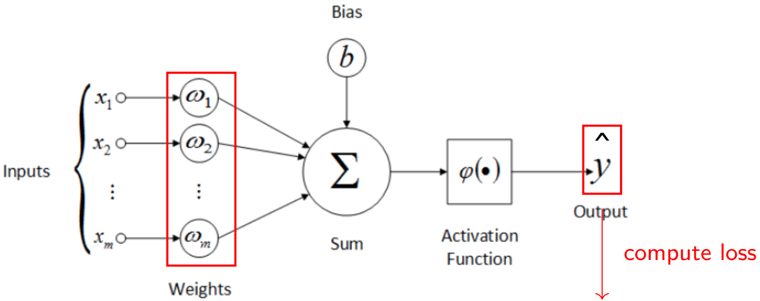
$$= -\frac{1}{N} \sum_{n=1}^{N} \left[y^{(n)} \ln \hat{y}^{(n)} + (1 - y^{(n)}) \ln(1 - \hat{y}^{(n)}) \right]$$

Review: Logistic regression classifier



Backward pass $\Longleftarrow$

Review: Logistic regression classifier



Update w to minimize $\sum l(y, \hat{y})$

Backward pass $\Longleftarrow$

Cross-entropy loss:

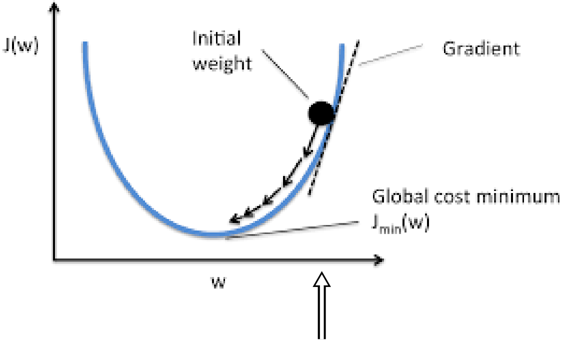
$$J(\mathbf{w}) = -\frac{1}{N}\sum_{n=1}^{N}\left[y^{(n)}\ln\hat{y}^{(n)} + (1-y^{(n)})\ln(1-\hat{y}^{(n)})\right]$$

Sketch of the **gradient descent based algorithm**:

1. Randomly choose $\mathbf{w}$
2. Compute the gradient of J with respect to $\mathbf{w}$
   (direction of steepest ascent of J at $\mathbf{w}$)
3. Update $\mathbf{w}$ to the opposite direction of the gradient
4. Repeat steps (2)-(3) until stopping criterion is met

Review: Gradient descent

Cost function

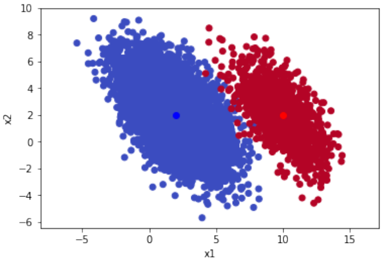


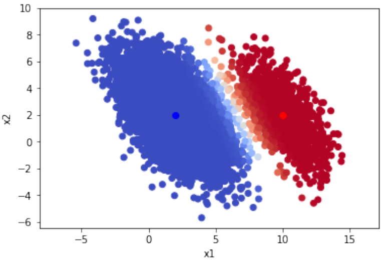
Initial weight

Logistic regression generates a linear decision boundary

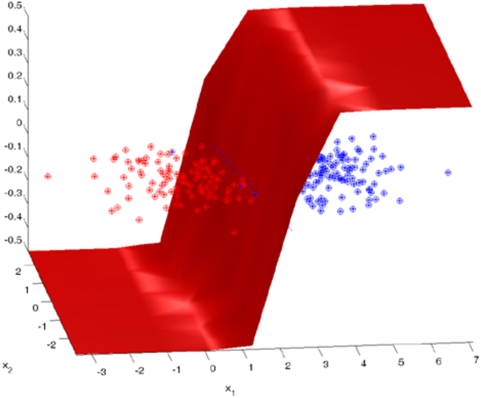
$$\mathbf{w}^T\mathbf{x} = w_1x_1 + w_2x_2 + \ldots + w_dx_d + b$$

$$\hat{y} = \hat{P}(y = 1|\mathbf{x}) = \theta(\mathbf{w}^T\mathbf{x}) = \frac{1}{1+e^{-\mathbf{w}^T\mathbf{x}}} \in [0, 1]$$





A 3D view



Source: http://strijov.com/sources/demoDataGen.php

Logistic regression generates a linear decision boundary

Why?

Decision boundary:
$\{\mathbf{x} \in \mathbb{R}^d : \hat{P}(y=1|\mathbf{x}) = \hat{P}(y=0|\mathbf{x})\} = \{\mathbf{x} \in \mathbb{R}^d : \hat{P}(y=1|\mathbf{x}) = 0.5\}$

Since $\hat{P}(y=1|\mathbf{x}) = \frac{1}{1+e^{-\mathbf{w}^T\mathbf{x}}}$, we have

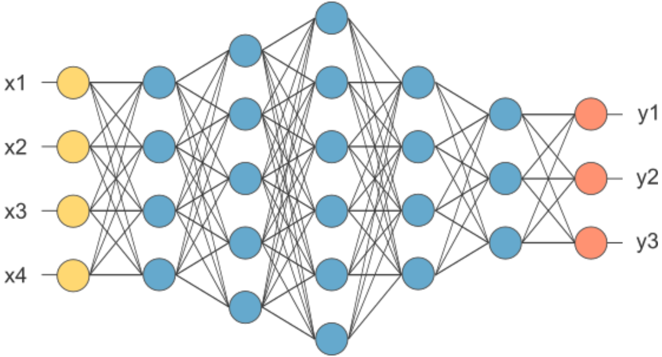
$$\frac{1}{1+e^{-\mathbf{w}^T\mathbf{x}}} = 0.5 \iff 0.5(1+e^{-\mathbf{w}^T\mathbf{x}}) = 1 \iff 1+e^{-\mathbf{w}^T\mathbf{x}} = 2$$

$$\iff e^{-\mathbf{w}^T\mathbf{x}} = 1 \iff \ln e^{-\mathbf{w}^T\mathbf{x}} = \ln 1 \iff -\mathbf{w}^T\mathbf{x} = 0$$

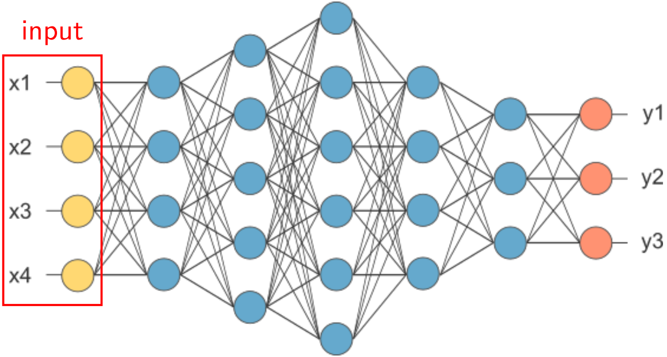
Summary: Logistic regression classifier

- deals with binary classification
- target (class label) is binary: positive=1 or negative=0
- learning formulation models output as $P(y = 1|\mathbf{x})$
- standard cost function to be optimized: cross-entropy loss
- optimization (training) is usually based on the gradient descent algorithm
- Resulting decision boundary is a hyperplane

Neural network – What it is

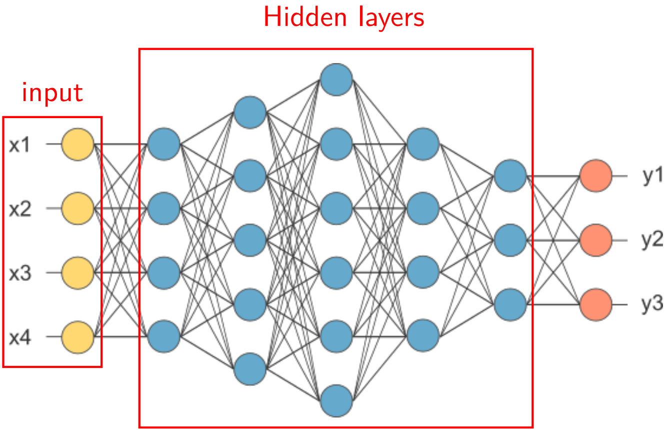


Source: https://www.kaggle.com/shokhan/neural-network-to-predict-dota-2-winner/comments



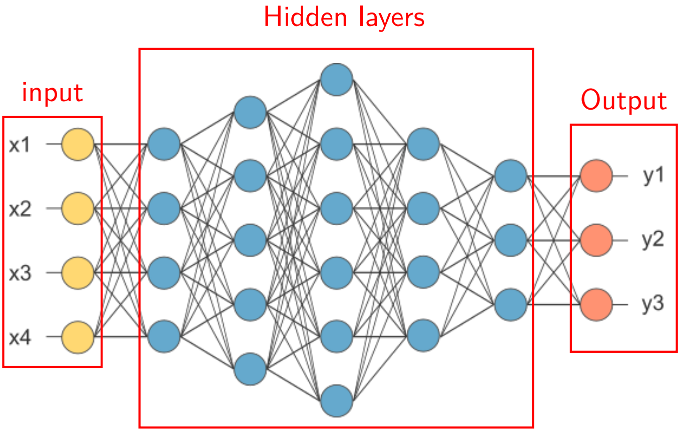
Source: https://www.kaggle.com/shokhan/neural-network-to-predict-dota-2-winner/comments

Neural network – What it is



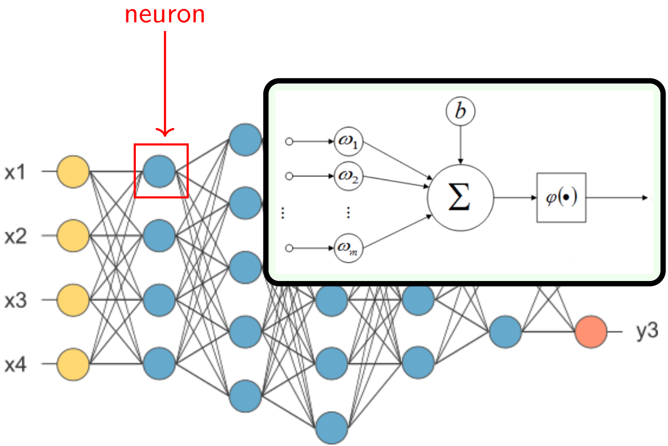
Source: https://www.kaggle.com/shokhan/neural-network-to-predict-dota-2-winner/comments

Neural network – What it is



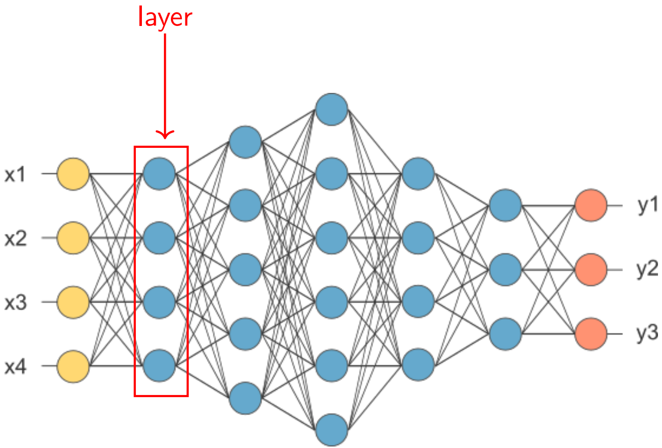
Source: https://www.kaggle.com/shokhan/neural-network-to-predict-dota-2-winner/comments

Neural network – What it is



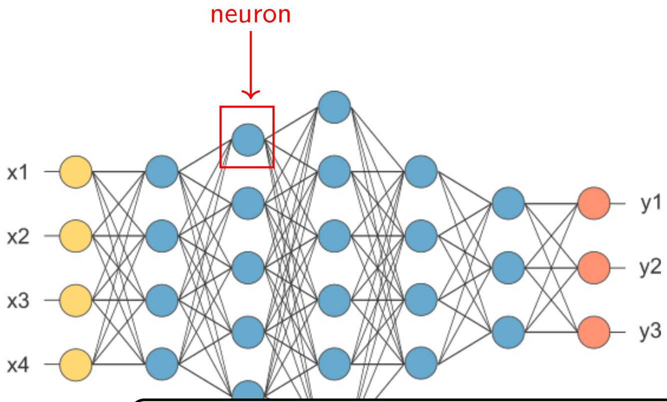
Source: https://www.kaggle.com/shokhan/neural-network-to-predict-dota-2-winner/comments

Neural network – What it is



Source: https://www.kaggle.com/shokhan/neural-network-to-predict-dota-2-winner/comments

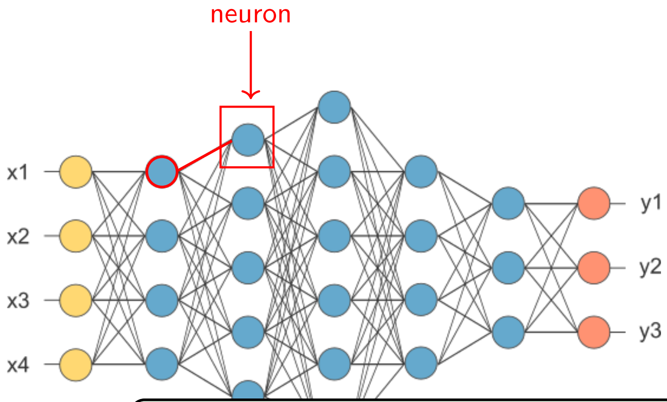
Neural network – What it is



$$\varphi\Big(w_1\varphi(s_1) + w_2\varphi(s_2) + w_3\varphi(s_3) + w_4\varphi(s_4)\Big)$$

Source: https://www.kaggle.com/shokhan/neural-network-to-predict-dota-2-winner/comments

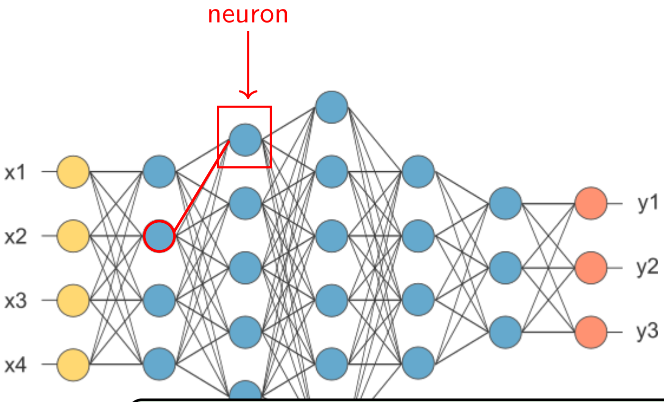
Neural network – What it is



$$\varphi\Big(w_1\varphi(s_1) + w_2\varphi(s_2) + w_3\varphi(s_3) + w_4\varphi(s_4)\Big)$$

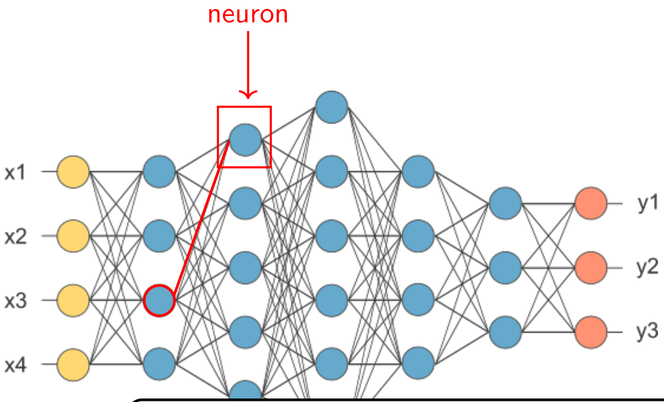
Source: https://www.kaggle.com/shokhan/neural-network-to-predict-dota-2-winner/comments

Neural network – What it is



$$\varphi\Big(w_1\varphi(s_1) + w_2\varphi(s_2) + w_3\varphi(s_3) + w_4\varphi(s_4)\Big)$$

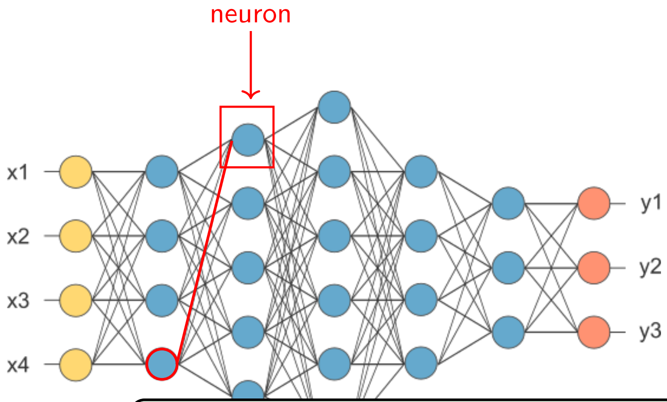
Source: https://www.kaggle.com/shokhan/neural-network-to-predict-dota-2-winner/comments



$$\varphi\Big(w_1\varphi(s_1) + w_2\varphi(s_2) + w_3\varphi(s_3) + w_4\varphi(s_4)\Big)$$

Source: https://www.kaggle.com/shokhan/neural-network-to-predict-dota-2-winner/comments

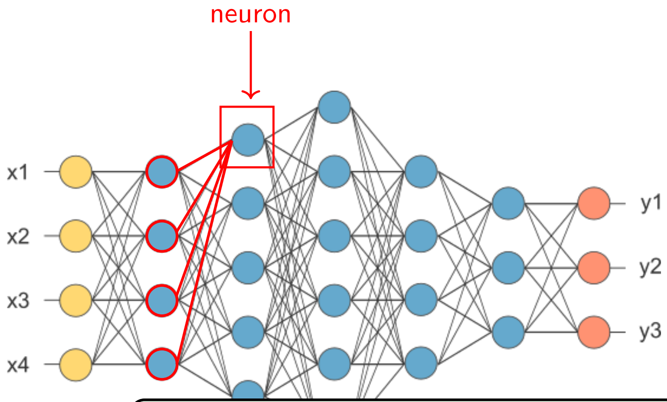
Neural network – What it is



$$\varphi\Big(w_1\varphi(s_1) + w_2\varphi(s_2) + w_3\varphi(s_3) + w_4\varphi(s_4)\Big)$$

Source: https://www.kaggle.com/shokhan/neural-network-to-predict-dota-2-winner/comments

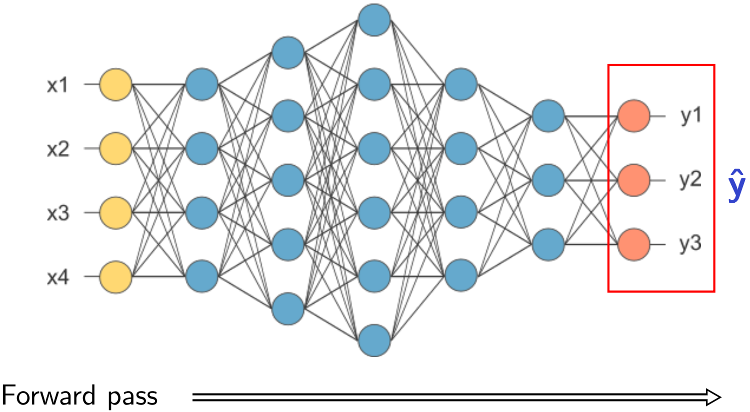
Neural network – What it is



$$\varphi\Big(w_1\varphi(s_1) + w_2\varphi(s_2) + w_3\varphi(s_3) + w_4\varphi(s_4)\Big)$$

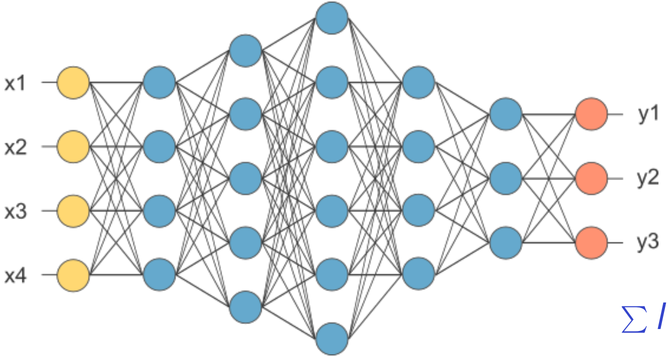
Source: https://www.kaggle.com/shokhan/neural-network-to-predict-dota-2-winner/comments

Neural network – What it is



Source: https://www.kaggle.com/shokhan/neural-network-to-predict-dota-2-winner/comments

Neural network – What it is

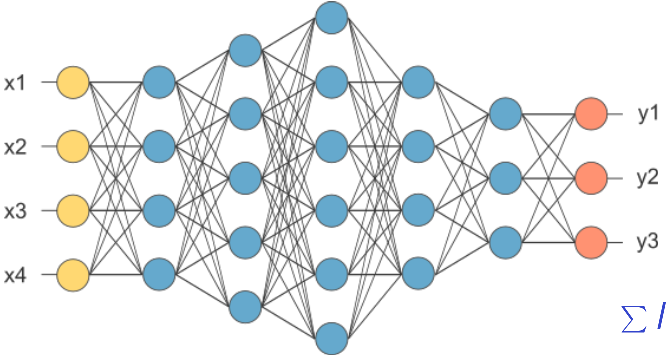


$\hat{\mathbf{y}}$

$\sum l(y_j, \hat{y}_j)$

Source: https://www.kaggle.com/shokhan/neural-network-to-predict-dota-2-winner/comments

Neural network – What it is



$\hat{\mathbf{y}}$

$\sum l(y_j, \hat{y}_j)$

Backward pass $\Longleftarrow$

Source: https://www.kaggle.com/shokhan/neural-network-to-predict-dota-2-winner/comments

Understanding neural networks

1950's and 1960's: excitement phase

1969: Minsky et al, "Perceptrons" $\rightsquigarrow$ AI winter

1987: Rumelhart et al, "Parallel Distributed Processing" $\rightsquigarrow$ backpropagation algorithm

1990: hard to train; lack of practical results $\rightsquigarrow$ AI winter **SVM**

1998: Yan LeCun, "Gradient-based Learning Applied to Document Recognition" $\rightsquigarrow$ convnets

2006: Geoffrey E. Hinton et al, "A Fast Learning Algorithm for Deep Belief Nets" $\rightsquigarrow$ effective training of deeper neural nets

2012: Convolutional neural network (Alexnet) wins the image classification competition (ImageNet)

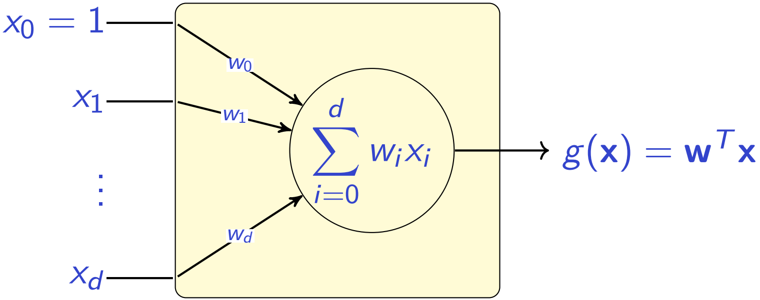
2018: Bengio, Hinton and LeCun won the Turing Award

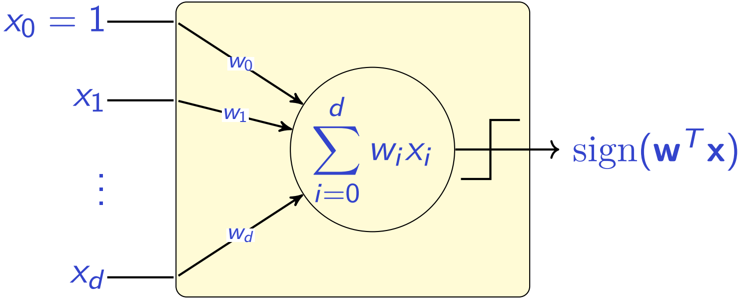
(https://awards.acm.org/about/2018-turing)

Yoshua Bengio, Geoffrey Hinton and Yann LeCun

LeCun is a mathematical sciences professor at New York University and the vice president and chief AI scientist at Facebook. Hinton is a vice president and engineering fellow at Google. Bengio is a professor at the University of Montreal and the scientific director of both Quebec's Artificial Intelligence Institute and the Institute for Data Valorization.

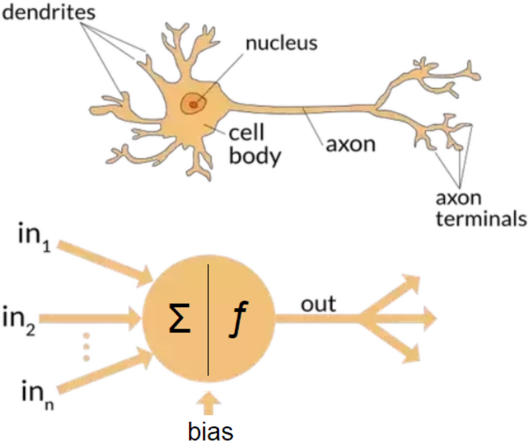
The linear machine





The perceptron can be seen as a **neuron** model

Perceptron – similarities with a biological neuron



Warning: nowadays some people doesn't like even to mention this to explain neural networks

Multilayer perceptron networks

Feedforward multilayer neural networks

Perceptron (single layer)

Linear machine $g(\mathbf{x})$ + decision ϕ

$$\text{output} = \phi(g(\mathbf{x})) = \phi(\mathbf{w}^T\mathbf{x})$$

Signal function:

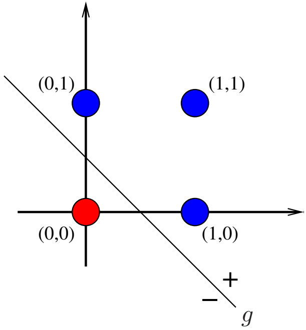
$$\phi(z) = \begin{cases} +1, & \text{se } z > 0, \\ -1 & \text{se } z \leq 0. \end{cases}$$

Step function:

$$\phi(z) = \begin{cases} 1, & \text{se } z > 0, \\ 0 & \text{se } z \leq 0. \end{cases}$$

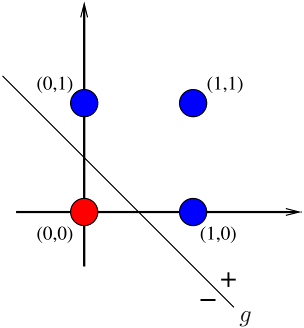
Implementation of function OR with perceptron

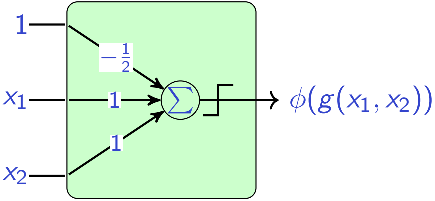
x_1	x_2	$\phi(g(x_1, x_2))$
0	0	0
0	1	1
1	0	1
1	1	1



Implementation of function OR with perceptron

x_1	x_2	$\phi(g(x_1, x_2))$
0	0	0
0	1	1
1	0	1
1	1	1

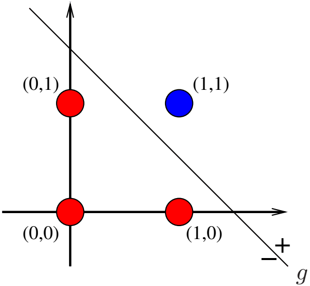




$$g(x_1, x_2) = x_1 + x_2 - \frac{1}{2}$$

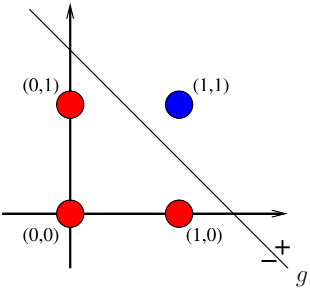
Implementation of function AND with perceptron

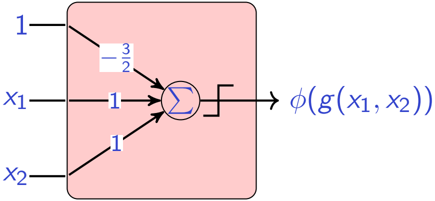
x_1	x_2	$\phi(g(x_1, x_2))$
0	0	0
0	1	0
1	0	0
1	1	1



Implementation of function AND with perceptron

x_1	x_2	$\phi(g(x_1, x_2))$
0	0	0
0	1	0
1	0	0
1	1	1

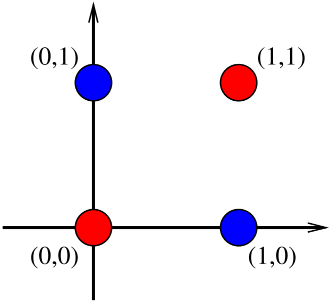




$$g(x_1, x_2) = x_1 + x_2 - \frac{3}{2}$$

XOR is not linearly separable

x_1	x_2	XOR
0	0	0
0	1	1
1	0	1
1	1	0

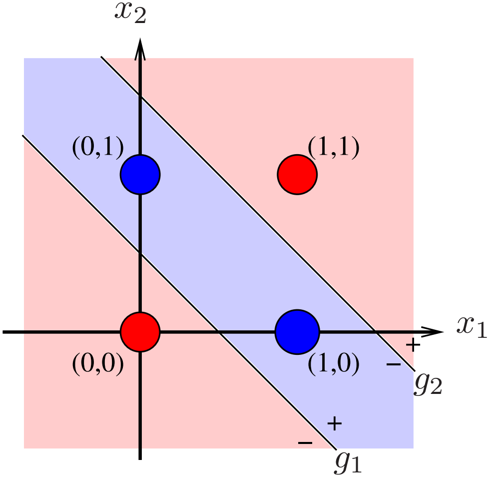


A solution for the XOR problem

Rationale: Use two linear functions

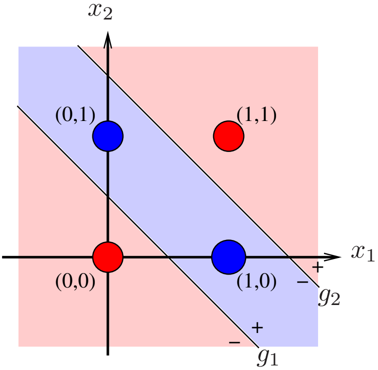
$g_1(\mathbf{x}) > 0$ and $g_2(\mathbf{x}) < 0$
$\Downarrow$
$f(\mathbf{x}) = 1$

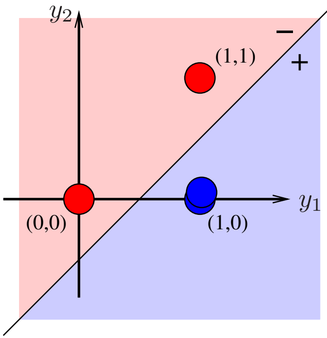
$g_1(\mathbf{x}) < 0$ or $g_2(\mathbf{x}) > 0$
$\Downarrow$
$f(\mathbf{x}) = 0$



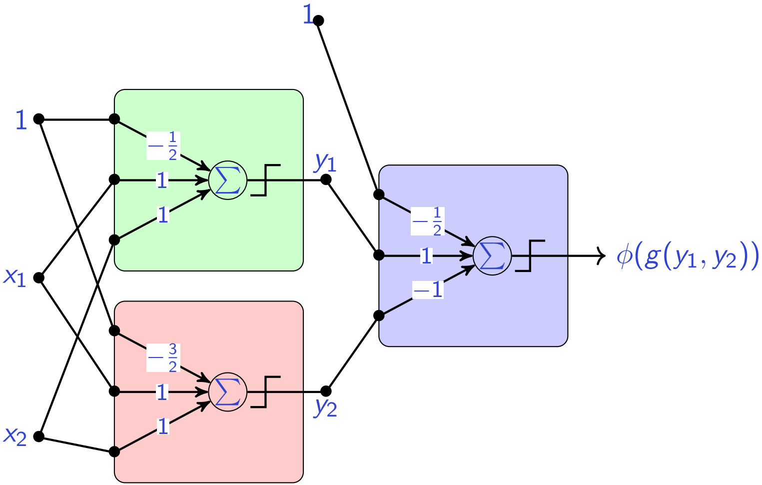
Combine the signal of two linear functions

x_1	x_2	g_1	g_2	$y_1 = \phi(g_1)$	$y_2 = \phi(g_2)$	$g(y_1, y_2)$	$\phi(g(y_1, y_2))$
0	0	-	-	0	0	-	0
0	1	+	-	1	0	+	1
1	0	+	-	1	0	+	1
1	1	+	+	1	1	-	0

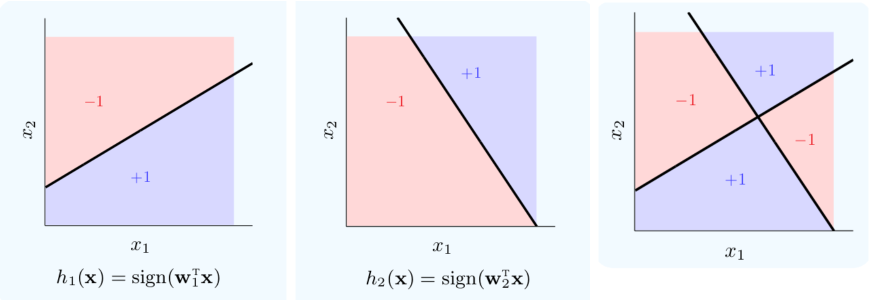




XOR function: a two-layer perceptron network



Example from Prof. Abu-Mostafa's Lecture

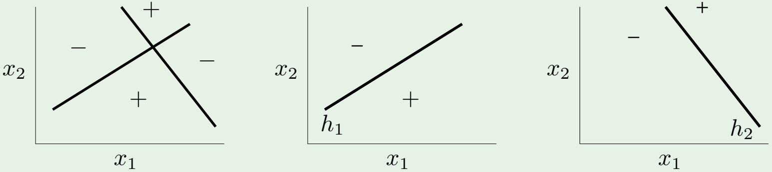


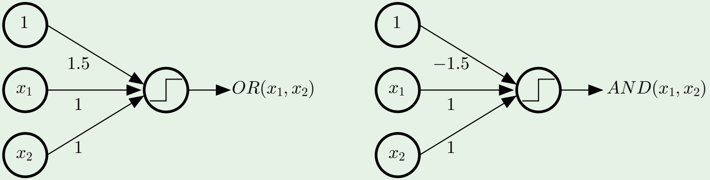
$$f = h_1\,\overline{h}_2 + \overline{h}_1\,h_2$$

Here we have four regions (in the previous example we had three regions)

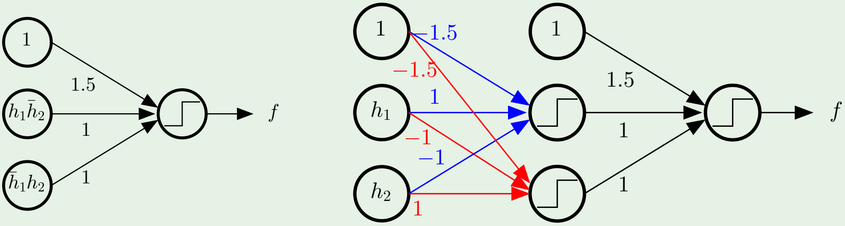
+1=TRUE and -1=FALSE
(in the previous example 1=TRUE and 0=FALSE)

Combining perceptrons

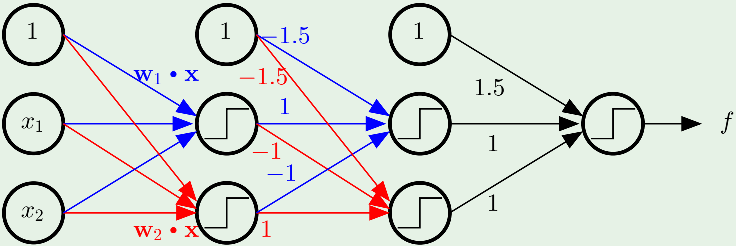




Creating layers



The multilayer perceptron



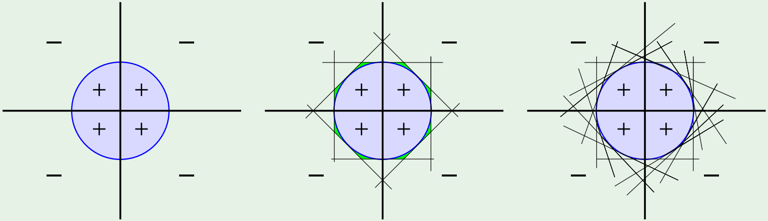
3 layers "feedforward"

The two previous examples show ways to combine

two linear functions to solve the XOR-like configuration problems

What about more complex configurations ?

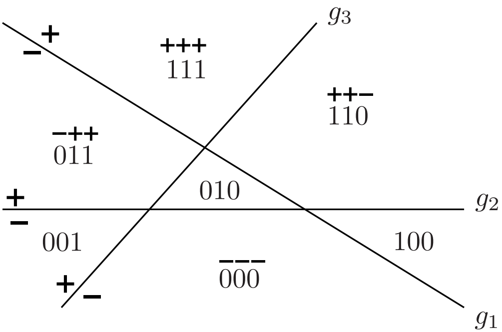
A powerful model



Target   8 perceptrons   16 perceptrons

2 red flags for **generalization** and **optimization**

Example: Let us consider $\mathbf{x} \in \mathbb{R}^2$ and $p = 3$ linear functions g_1, g_2, g_3. We have 7 regions in $\mathbb{R}^2$:



Each region can be assigned to an identity in $\{0, 1\}^3$!

General case with p linear functions:

- Let each function g_i define a perceptron $\phi(g_i(\mathbf{x}))$, and consider them as the nodes in the first hidden layer

- The output of the first layer is an element in $\mathbf{y} \in \mathbb{R}^p$

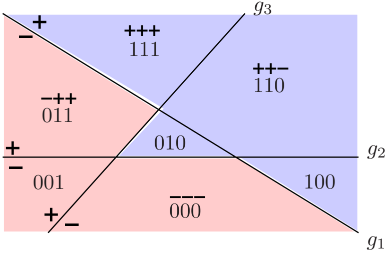
$$\mathbf{y} = \Big(\phi(g_1(\mathbf{x})), \phi(g_2(\mathbf{x})), \ldots, \phi(g_p(\mathbf{x}))\Big) = (y_1, y_2, \ldots, y_p)$$

- Since $\phi(\cdot) \in \{0, 1\}$, $(y_1, y_2, \ldots, y_p)$ is a vertex of the unitary hypercube H_p in $\mathbb{R}^p$

- This implies that all points of $X = \mathbb{R}^d$ in a particular region (among those defined by $g_i()$) will be mapped to a same vertex in H_p

More on perceptron networks

- We could employ a linear classifier on H_p – this would separate some vertices as positive and others as negatives
- The effect of that is the classification of the regions as **0** or **1**



- But the regions classes may correspond to a XOR configuration on H_p ...

Three layers perceptron network

- Instead of a single linear classifier in the second layer, we can employ k classifiers, one for each vertex corresponding to a region of class 1
(this can be easily implemented via AND function)

- Then, we add a third layer that will compute the OR of the previous layer outputs

Three layers perceptron network

- **First layer**: each node defines a hyperplane in $\mathbb{R}^d$
  The set of hyperplanes defines polyhedras (regions)
  The output of the first layer is a vertex of a hypercube in $\mathbb{R}^p$

- **Second layer:** each node selects one vertex in the hypercube, which corresponds to a region (polyhedra) in $\mathbb{R}^d$
  One node for each region of interest

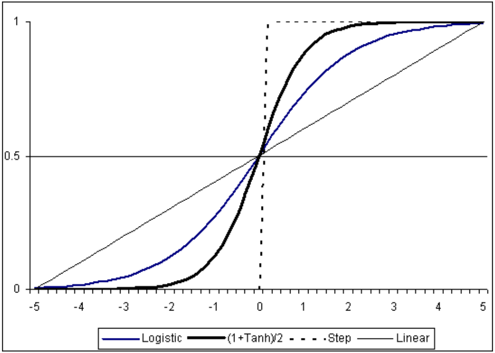
- **Third layer**: the node joins (via OR) the outputs of the previous layer
  If the input is in one of the selected regions, then the output will be positive

Conclusion: with three perceptron layers, we are able to represent any union of polyhedra defined in $\mathbb{R}^d$.

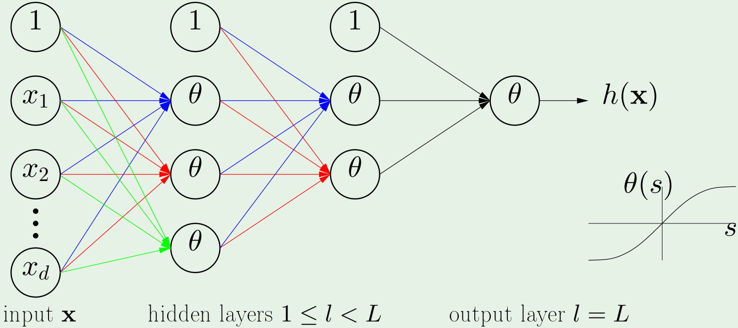
However ...

- a huge number of perceptrons might be necessary to approximate smoothly curved boundaries
- Large number of nodes in the second layer too
- No algorithm to design such network!

In each neuron (perceptron) we change the activation function ϕ (step or signal function) with a continuous differentiable function



The neural network



Universal approximation theorem

- Some theoretical works try to show that feedforward neural networks are able to approximate any continuous function

- **Cybenko, G. (1989)** "Approximations by superpositions of sigmoidal functions", Mathematics of Control, Signals, and Systems, 2 (4), 303-314 showed that any continuous function [under some not strong restrictions] can be approximated by a superposition of sigmoid functions.

- refs adicionais: http://neuron.eng.wayne.edu/tarek/MITbook/chap2/2_3.html

Universal approximation theorem

From Wikipedia: *In the mathematical theory of artificial neural networks, the universal approximation theorem states that a feed-forward network with a single hidden layer containing a finite number of neurons can approximate continuous functions on compact subsets of* $\mathbb{R}^n$*, under mild assumptions on the activation function.*

Let $\varphi : \mathbb{R} \to \mathbb{R}$ be a nonconstant, bounded, and continuous function. Let I_m denote the m-dimensional unit hypercube $[0,1]^m$. The space of real-valued continuous functions on I_m is denoted by $C(I_m)$. Then, given any $\varepsilon > 0$ and any function $f \in C(I_m)$, there exist an integer N, real constants $v_i, b_i \in \mathbb{R}$ and real vectors $w_i \in \mathbb{R}^m$ for $i = 1, \ldots, N$, such that we may define:

$$F(x) = \sum_{i=1}^{N} v_i \varphi\left(w_i^T x + b_i\right)$$

as an approximate realization of the function f; that is,

$$|F(x) - f(x)| < \varepsilon$$

for all $x \in I_m$. In other words, functions of the form $F(x)$ are dense in $C(I_m)$.

Universal approximation theorem

Theoretically, we can approximate any continuous function with a neural network with one hidden layer !

Multilayer feedforward network training

Backpropagation algorithm

Neural network training

We would like to find $\mathbf{w}$ that minimizes a cost function $J(\mathbf{w})$

Let us suppose a network with c outputs and the following loss function

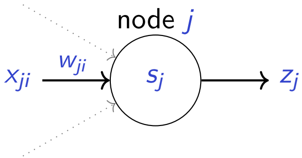
$$J(\mathbf{w}) = \frac{1}{2}\sum_{k=1}^{c}(t_k - z_k)^2$$

t_k : Expected output (target)

z_k : Predicted output (result of the forward pass)

Notations:

Let us consider a general node j in the network



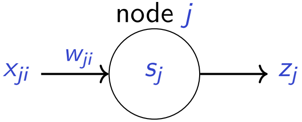
x_{ji} is the i-th input of node j

w_{ji} is the weight relative to the i-th input of node j

$$s_j = \sum_i w_{ji} x_{ji} \qquad z_j = \phi(s_j) \qquad (\ x_{ji} = z_i\ )$$

Gradient computation

Gradient of J with respect of w_{ji}:



w_{ji} influences the rest of the network through s_j:

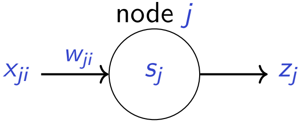
$$\frac{\partial J}{\partial w_{ji}} = \frac{\partial J}{\partial s_j}\frac{\partial s_j}{\partial w_{ji}}$$

Since $s_j = \sum_i w_{ji}\, x_{ji}$, then

$$\frac{\partial J}{\partial w_{ji}} = \frac{\partial J}{\partial s_j}\, x_{ji}$$

Gradient computation

Gradient of J with respect of w_{ji}:



w_{ji} influences the rest of the network through s_j:

$$\frac{\partial J}{\partial w_{ji}} = \frac{\partial J}{\partial s_j} \frac{\partial s_j}{\partial w_{ji}}$$

Since $s_j = \sum_i w_{ji}\, x_{ji}$, then

$$\frac{\partial J}{\partial w_{ji}} = \frac{\partial J}{\partial s_j} x_{ji}$$

Gradient computation

$$\frac{\partial J}{\partial w_{ji}} = \frac{\partial J}{\partial s_j} x_{ji} \tag{1}$$

If j is a node in the output layer, just as w_{ji} can influence the rest of the network only through s_j, s_j can influence the rest of the networks only through z_j ($z_j = \phi(s_j)$).

$$\frac{\partial J}{\partial s_j} = \frac{\partial J}{\partial z_j} \frac{\partial z_j}{\partial s_j} \tag{2}$$

If a node j is in other previous layers, then s_j affects J through all nodes k in the subsequent layer:

$$\frac{\partial J}{\partial s_j} = \sum_k \frac{\partial J}{\partial s_k} \frac{\partial s_k}{\partial s_j} \tag{3}$$

Weights related to the nodes in the output layer

Assume j is a node in the output layer

s_j affects J through z_j. Here we compute Eq. 2

$\boxed{\frac{\partial J}{\partial w_{ji}} = \frac{\partial J}{\partial s_j} x_{ji}}$

$$\frac{\partial J}{\partial s_j} = \frac{\partial J}{\partial z_j} \frac{\partial z_j}{\partial s_j}$$

Weights related to the nodes in the output layer

Assume j is a node in the output layer

s_j affects J through z_j. Here we compute Eq. 2

$$\boxed{\frac{\partial J}{\partial w_{ji}} = \frac{\partial J}{\partial s_j} x_{ji}}$$

$$\frac{\partial J}{\partial s_j} = \frac{\partial J}{\partial z_j} \frac{\partial z_j}{\partial s_j}$$

$$\boxed{\frac{\partial z_j}{\partial s_j} = \frac{\partial \phi(s_j)}{\partial s_j} = \phi'(s_j)}$$

$$z_j = \phi(s_j)$$

Weights related to the nodes in the output layer

Assume j is a node in the output layer

s_j affects J through z_j. Here we compute Eq. 2

$$\boxed{\frac{\partial J}{\partial w_{ji}} = \frac{\partial J}{\partial s_j} x_{ji}}$$

$$\frac{\partial J}{\partial s_j} = \frac{\partial J}{\partial z_j}\frac{\partial z_j}{\partial s_j}$$

$$\frac{\partial z_j}{\partial s_j} = \frac{\partial \phi(s_j)}{\partial s_j} = \phi'(s_j)$$

$$\boxed{\frac{\partial J}{\partial z_j} = \frac{\partial}{\partial z_j}\Big[\frac{1}{2}\sum_{k=1}^{c}(t_k - z_k)^2\Big] = \frac{1}{2}2(t_j - z_j)\frac{\partial (t_j - z_j)}{\partial z_j} = -(t_j - z_j)}$$

Weights related to the nodes in the output layer

Assume j is a node in the output layer

s_j affects J through z_j. Here we compute Eq. 2

$$\boxed{\frac{\partial J}{\partial w_{ji}} = \frac{\partial J}{\partial s_j} x_{ji}}$$

$$\frac{\partial J}{\partial s_j} = \frac{\partial J}{\partial z_j}\frac{\partial z_j}{\partial s_j} \qquad \frac{\partial z_j}{\partial s_j} = \frac{\partial \phi(s_j)}{\partial s_j} = \phi'(s_j)$$

$$\frac{\partial J}{\partial z_j} = \frac{\partial}{\partial z_j}\Big[\frac{1}{2}\sum_{k=1}^{c}(t_k - z_k)^2\Big] = \frac{1}{2}2(t_j - z_j)\frac{\partial (t_j - z_j)}{\partial z_j} = -(t_j - z_j)$$

Thus replacing on Eq. 1

$$\frac{\partial J}{\partial w_{ji}} = \frac{\partial J}{\partial s_j} x_{ji} = -\underbrace{(t_j - z_j)\phi'(s_j)}_{\delta_j} x_{ji}$$

Weights related to the nodes in the hidden layers

Assume j is a node in a hidden layer

We must consider all ways in which s_j affects J (every node to where its output is propagated) – here we compute Eq. 3

$\frac{\partial J}{\partial w_{ji}} = \frac{\partial J}{\partial s_j} x_{ji}$

$$\boxed{\frac{\partial J}{\partial s_j} = \sum_k \frac{\partial J}{\partial s_k} \frac{\partial s_k}{\partial s_j}} = \sum_k -\delta_k \frac{\partial s_k}{\partial s_j} = \sum_k -\delta_k \frac{\partial s_k}{\partial z_j} \frac{\partial z_j}{\partial s_j}$$

$$= \sum_k -\delta_k w_{kj} \frac{\partial z_j}{\partial s_j} = \sum_k -\delta_k w_{kj} \phi'(s_j)$$

Weights related to the nodes in the hidden layers

Assume j is a node in a hidden layer

We must consider all ways in which s_j affects J (every node to where its output is propagated) – here we compute Eq. 3

$\frac{\partial J}{\partial w_{ji}} = \frac{\partial J}{\partial s_j} x_{ji}$

Previous slide

$$\begin{aligned} \frac{\partial J}{\partial s_j} &= \sum_k \frac{\partial J}{\partial s_k} \frac{\partial s_k}{\partial s_j} = \sum_k -\delta_k \frac{\partial s_k}{\partial s_j} = \sum_k -\delta_k \frac{\partial s_k}{\partial z_j} \frac{\partial z_j}{\partial s_j} \\ &= \sum_k -\delta_k w_{kj} \frac{\partial z_j}{\partial s_j} = \sum_k -\delta_k w_{kj} \phi'(s_j) \end{aligned}$$

Weights related to the nodes in the hidden layers

Assume j is a node in a hidden layer

We must consider all ways in which s_j affects J (every node to where its output is propagated) – here we compute Eq. 3

$\boxed{\frac{\partial J}{\partial w_{ji}} = \frac{\partial J}{\partial s_j} x_{ji}}$

$$\begin{aligned}\frac{\partial J}{\partial s_j} &= \sum_k \frac{\partial J}{\partial s_k}\frac{\partial s_k}{\partial s_j} = \sum_k -\delta_k \frac{\partial s_k}{\partial s_j} = \sum_k -\delta_k \frac{\partial s_k}{\partial z_j}\frac{\partial z_j}{\partial s_j} \\ &= \sum_k -\delta_k w_{kj} \frac{\partial z_j}{\partial s_j} = \sum_k -\delta_k w_{kj} \phi'(s_j)\end{aligned}$$

s_j affects s_k through z_j

Weights related to the nodes in the hidden layers

Assume j is a node in a hidden layer

We must consider all ways in which s_j affects J (every node to where its output is propagated) – here we compute Eq. 3

$\boxed{\frac{\partial J}{\partial w_{ji}} = \frac{\partial J}{\partial s_j} x_{ji}}$

$$\begin{aligned}\frac{\partial J}{\partial s_j} &= \sum_k \frac{\partial J}{\partial s_k}\frac{\partial s_k}{\partial s_j} = \sum_k -\delta_k \frac{\partial s_k}{\partial s_j} = \sum_k -\delta_k \frac{\partial s_k}{\partial z_j}\frac{\partial z_j}{\partial s_j} \\ &= \sum_k -\delta_k w_{kj} \frac{\partial z_j}{\partial s_j} = \sum_k -\delta_k w_{kj} \phi'(s_j)\end{aligned}$$

$s_k = \sum_j w_{kj} x_{kj}$, $x_{kj} = z_j$

Weights related to the nodes in the hidden layers

Assume j is a node in a hidden layer

We must consider all ways in which s_j affects J (every node to where its output is propagated) – here we compute Eq. 3

$\frac{\partial J}{\partial w_{ji}} = \frac{\partial J}{\partial s_j} x_{ji}$

$$\begin{aligned} \frac{\partial J}{\partial s_j} &= \sum_k \frac{\partial J}{\partial s_k}\frac{\partial s_k}{\partial s_j} = \sum_k -\delta_k \frac{\partial s_k}{\partial s_j} = \sum_k -\delta_k \frac{\partial s_k}{\partial z_j}\frac{\partial z_j}{\partial s_j} \\ &= \sum_k -\delta_k w_{kj} \frac{\partial z_j}{\partial s_j} = \sum_k -\delta_k w_{kj} \phi'(s_j) \end{aligned}$$

$$z_j = \phi(s_j)$$

Weights related to the nodes in the hidden layers

Assume j is a node in a hidden layer

We must consider all ways in which s_j affects J (every node to where its output is propagated) – here we compute Eq. 3

$$\boxed{\frac{\partial J}{\partial w_{ji}} = \frac{\partial J}{\partial s_j} x_{ji}}$$

$$\begin{aligned}\frac{\partial J}{\partial s_j} &= \sum_k \frac{\partial J}{\partial s_k}\frac{\partial s_k}{\partial s_j} = \sum_k -\delta_k \frac{\partial s_k}{\partial s_j} = \sum_k -\delta_k \frac{\partial s_k}{\partial z_j}\frac{\partial z_j}{\partial s_j} \\ &= \sum_k -\delta_k w_{kj} \frac{\partial z_j}{\partial s_j} = \sum_k -\delta_k w_{kj} \phi'(s_j)\end{aligned}$$

Thus replacing on Eq. 1

$$\frac{\partial J}{\partial w_{ji}} = \frac{\partial J}{\partial s_j} x_{ji} = \underbrace{-\left[\sum_{k=1}^{c} w_{kj}\delta_k\right]\phi'(s_j)}_{\delta_j} x_{ji}$$

Summary

$$\mathbf{w}(r+1) = \mathbf{w}(r) + \Delta\mathbf{w}(r)$$

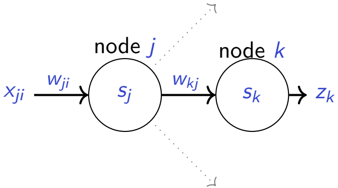
$$\Delta\mathbf{w}(r) = -\eta \nabla J(\mathbf{w})$$

If k is a node in the output layer:

$$\Delta w_{kj} = \eta \underbrace{\left(t_k - z_k\right) \phi'(s_k)}_{\delta_k} x_{kj}$$

If j is a node in the last hidden layer:

$$\Delta w_{ji} = \eta \underbrace{\left[\sum_{k=1}^{c} w_{kj}\delta_k\right] \phi'(s_j)}_{\delta_j} x_{ji}$$



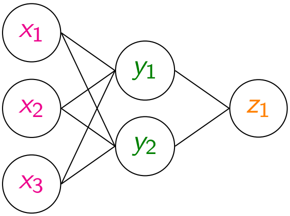
If we consider sigmoid as the activation function ϕ:

$$\delta_k = z_k(1 - z_k)(t_k - z_k)$$

$$\delta_j = z_j(1 - z_j) \sum_{k=1}^{c} w_{kj}\delta_k$$

Prof. Abu-Mostafa uses hiperbolic tangent as the activation function ϕ

Example



$$s_1 = w_{11}x_1 + w_{12}x_2 + w_{13}x_3 + b_1$$

$$y_1 = \phi(s_1)$$

$$s_2 = w_{21}x_1 + w_{22}x_2 + w_{23}x_3 + b_2$$

$$y_2 = \phi(s_2)$$

$$s_1 = w_{11}y_1 + w_{12}y_2 + b_1$$

$$z_1 = \phi(s_1)$$

Taking one example $\mathbf{x} = (x_1, x_2, x_3)$, one class ($c = 1$), and cost function

$$J(\mathbf{w}) = \frac{1}{2}\sum_k (t_k - z_k)^2$$

$$w_{kj} = w_{kj} + \Delta w_{kj}, \qquad k = 1, j = 1..2$$

$$\Delta w_{kj} = \eta \underbrace{(t_k - z_k)\,\phi'(s_k)}_{\delta_k}\, y_j$$

$$w_{ji} = w_{ji} + \Delta w_{ji}, \qquad j = 1..2, i = 1..3$$

$$\Delta w_{ji} = \eta \underbrace{\left[\sum_k w_{kj}\delta_k\right]\phi'(s_j)}_{\delta_j}\, x_i$$

Comments

- (Theoretical result) Neural networks with three layers can represent arbitrary functions (one hidden layer)
- The principle of backpropagation is the same for any cost function (we considered MSE)
  Must be differentiable
- Gradient descent may converge to a local minima
- Hidden layers can be understood as implicit transformed representations of input data
- Training neural networks is not simple because there are so many hyperparameters that need to be specified before training

Hyperparameters

- network architecture
- activation function
- cost function
- data normalization
- regularization
- Batch training $\times$ sthocastic training
- stopping criteria
- learning rate, momentum
- etc

NN Libraries

TensorFlow – https://www.tensorflow.org/

Keras – https://keras.io/

PyTorch – https://pytorch.org/

etc

Computation graphs and AutoGrad

Modern NN libraries are equipped with autograd functionalities

https://blog.paperspace.com/pytorch-101-understanding-graphs-and-automatic-differentiation/

(paper) AutoML-Zero: Evolving Machine Learning Algorithms From Scratch
https://arxiv.org/abs/2003.03384

Practice

With Keras:

deep-learning-with-python-notebooks

https://github.com/fchollet/deep-learning-with-python-notebooks

With scikit-learn:

`sklearn.neural_network.MLPClassifier`

https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html

Machine Learning with Neural Networks Using scikit-learn

https://www.pluralsight.com/guides/machine-learning-neural-networks-scikit-learn

Bibliographic reference

Online book

Neural Networks and Deep Learning, Michael Nielsen
http://neuralnetworksanddeeplearning.com/