The building blocks of deep learning models - part 1

A light overview

Filippo Biscarini
Senior Scientist
CNR, Milan (Italy)

Nelson Nazzicari
Research fellow
CREA, Lodi (Italy)

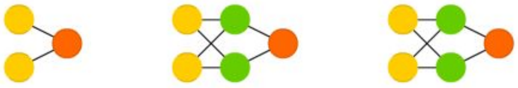
Deep learning: the building blocks

1. Function approximation
2. The neural network model:
   a. the “neuron”
   b. the network
3. Activation functions

This session

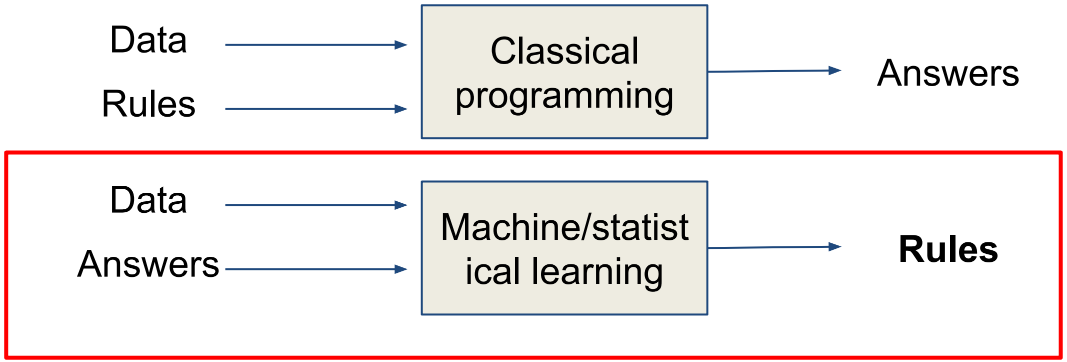
4. Cost functions
5. Gradient descent (and solvers/optimizers)
6. Forward propagation and the backward propagation algorithm

Through the logistic regression example

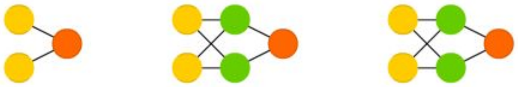


Function approximation

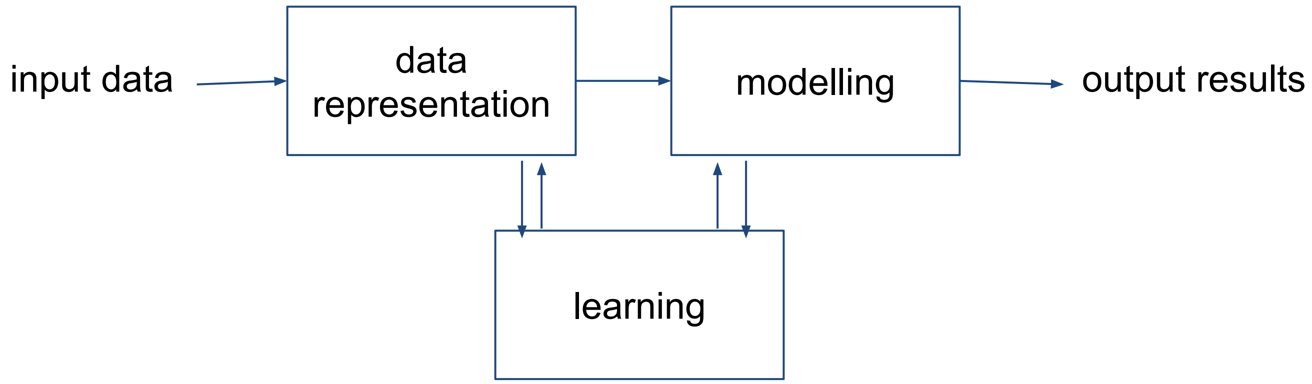
Function approximation?



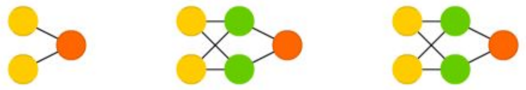
- **unknown function** that maps input data to output results (answers):
  - » $y = f(x)$
- learn this function → **function approximation**
- f(x) can be **nonlinear** and quite **complex**



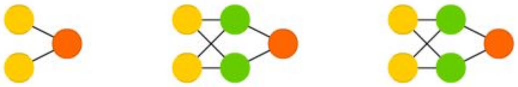
Function approximation



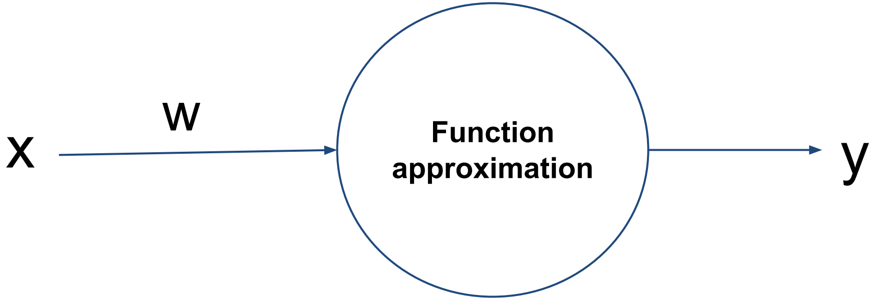
- NNs are good at finding functions that accurately map x to y
- deep neural networks (NNs) are powerful **function approximators**
  - » $y = f(x)$
- ¡complex highly non-linear functions can lead to problems with **generalization!**



The neural network model

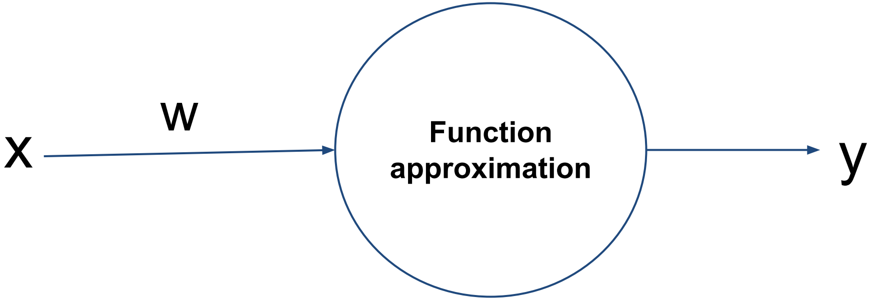


Neural network, the basic unit: the “neuron”

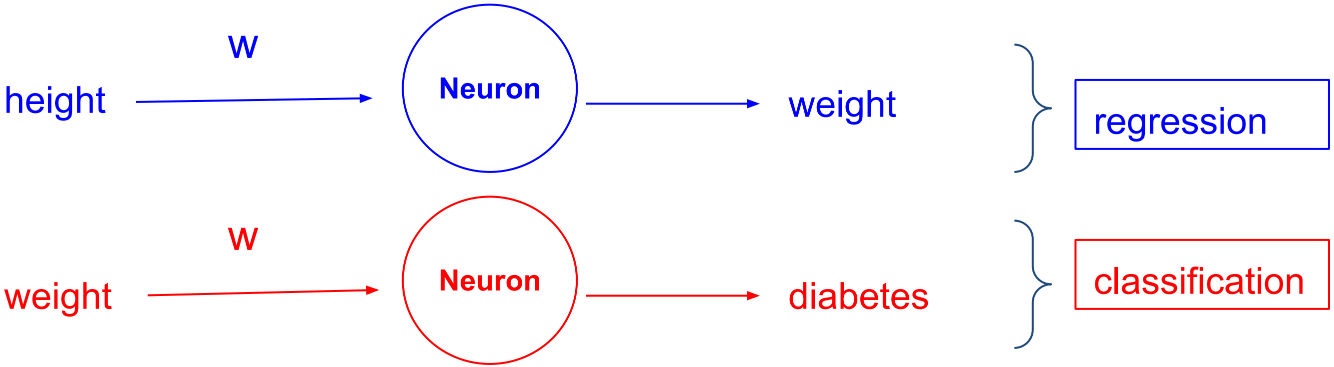


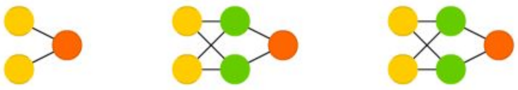
- Mc Culloch & Pitts (1943)
- **perceptron** (“neuron”):
  - dendrites
  - neuron
  - axon

Neural network, the basic unit: the “neuron”

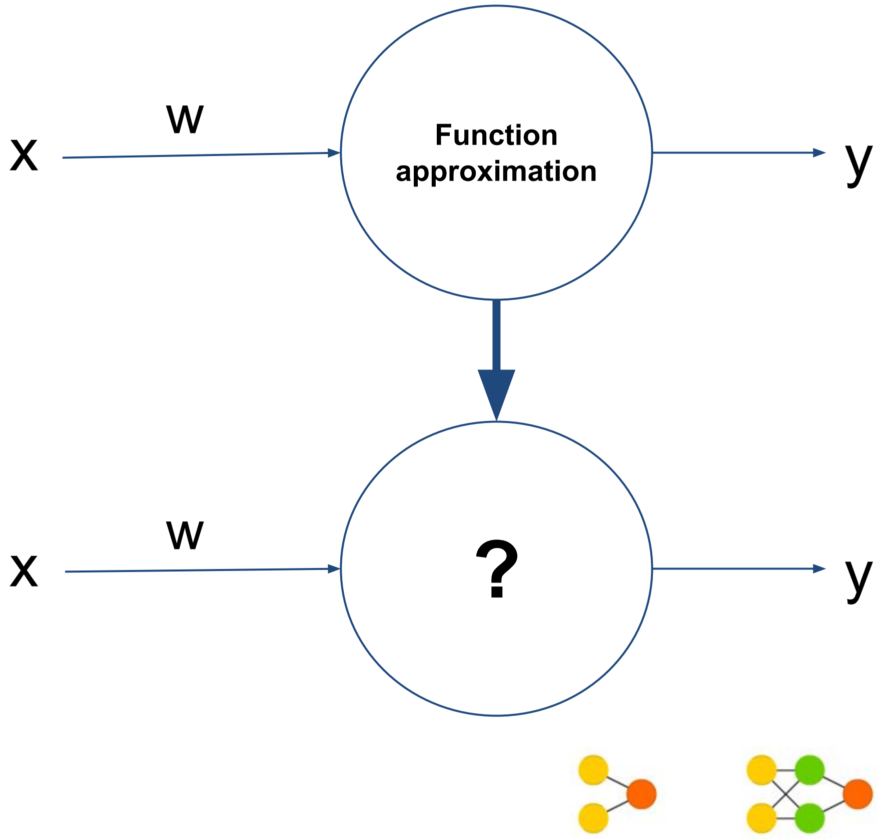


- Mc Culloch & Pitts (1943)
- **perceptron** (“neuron”):
  - dendrites
  - neuron
  - axon



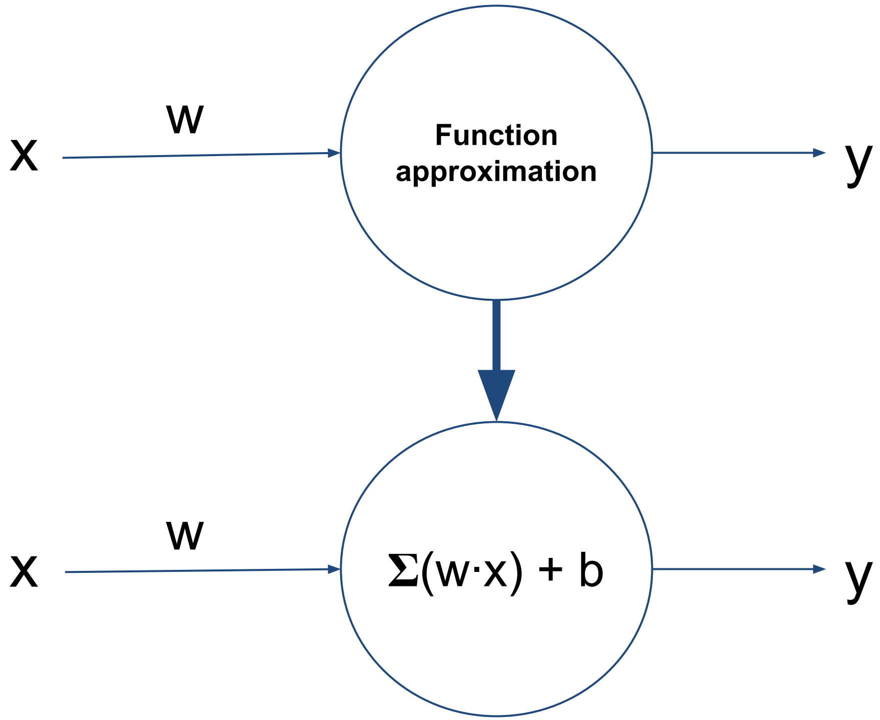


Neural network, the basic unit: the “neuron”



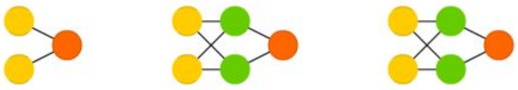
- Mc Culloch & Pitts (1943)
- **perceptron** (“neuron”):
  - dendrites
  - neuron
  - axon

Neural network, the basic unit: the “neuron”



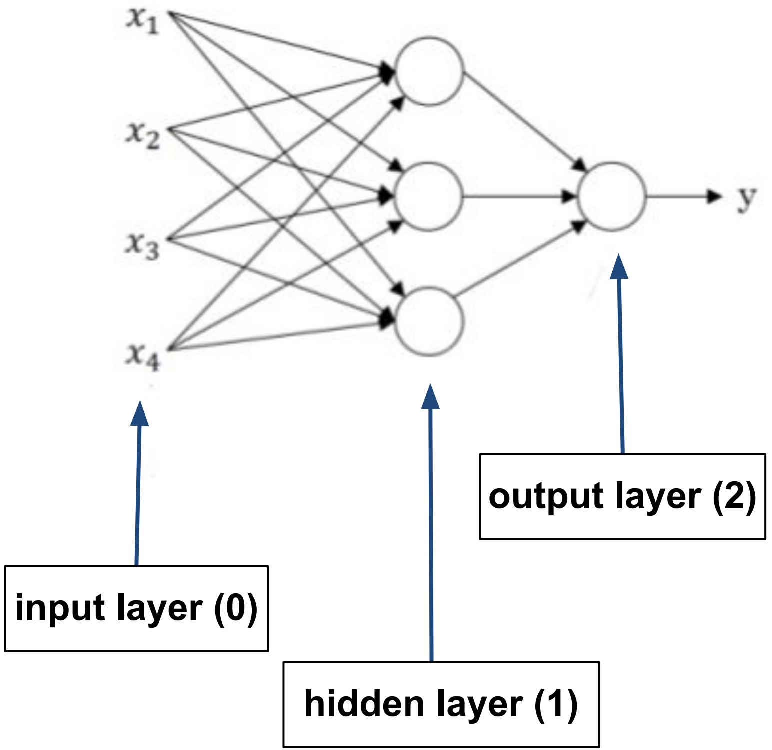
- Mc Culloch & Pitts (1943)
- **perceptron** (“neuron”):
  - dendrites
  - neuron
  - axon
- **learning the weights**

- e.g. linear combination of weights*features + bias
- fancy way to perform linear regression
- solved through NN rather than OLS or ML

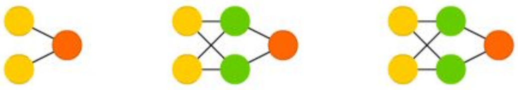


Anatomy of a neural network

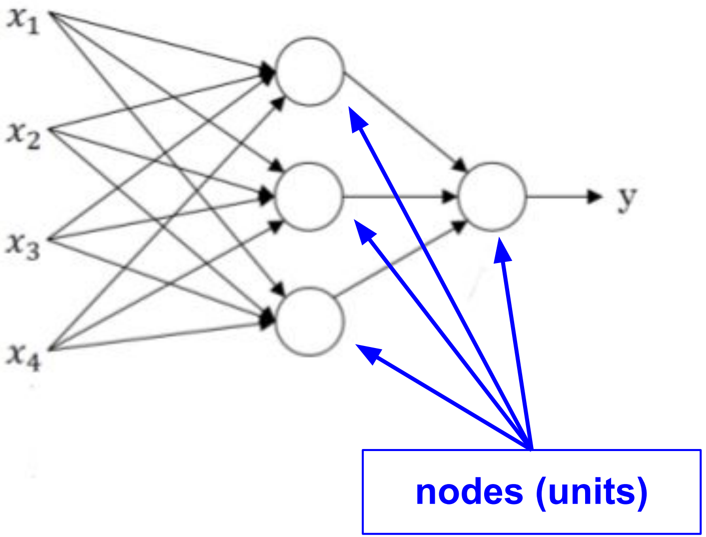
Fully connected (dense) neural network



- two-layer NN (not strictly "deep"):
  - input layer: [0]
  - hidden layer: [1]
  - output layer: [2]

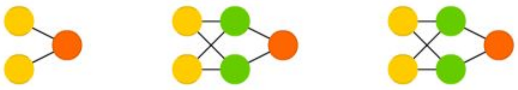


Fully connected (dense) neural network

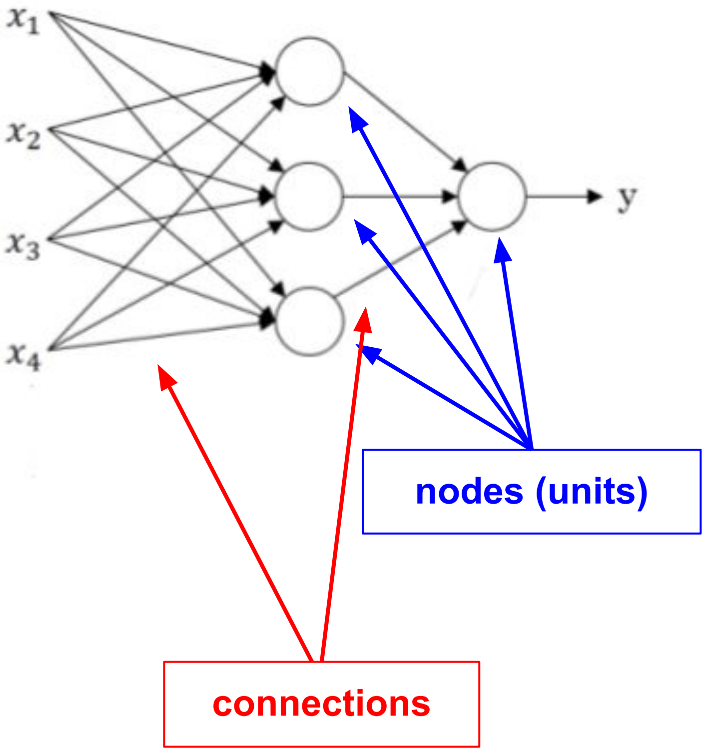


IMPORTANT!

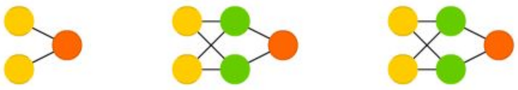
- each hidden unit takes in input all x features
- replicates the predictive model as many times as there are units ("neurons")
- if the approximated function is linear regression, each unit will fit a different linear regression model
  - e.g.: 3 units → 3 regression models



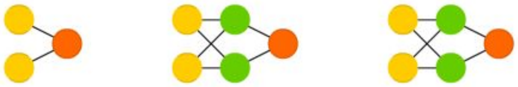
Fully connected (dense) neural network



- **two-layer NN** (not strictly "deep"):
    - input layer: [0]
    - hidden layer: [1]
    - output layer: [2]
- all features connected to all "neurons" in the hidden layer
- the NN will decide which variables to use (and how) in each node (by **learning the weights**)



Activation functions



Activation functions: what?

<u>“neuron” (unit)</u>

$$\Sigma(w \cdot x) + b \longrightarrow z \longrightarrow g(z)$$

- g(z): **activation function**

Activation functions: what?

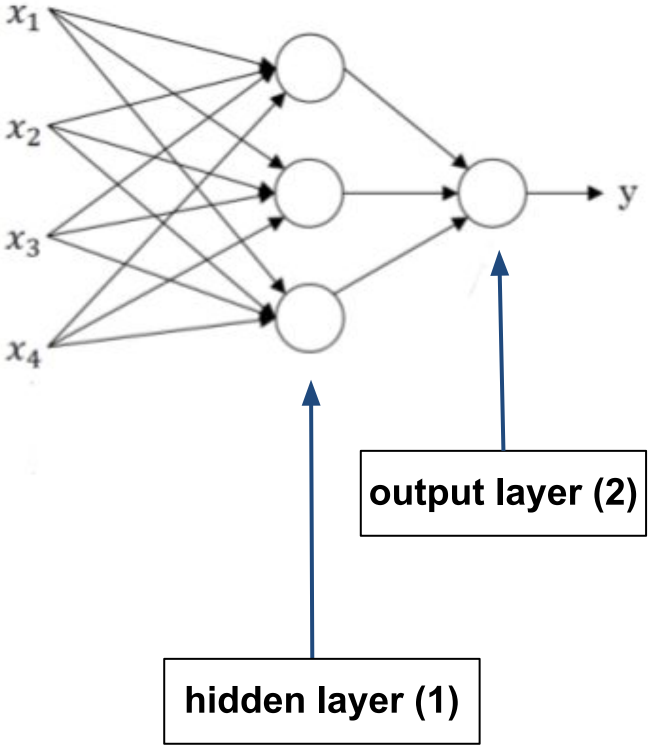
“neuron” (unit)

1. $z = \Sigma(w \cdot x) + b$
2. $g(z)$

y

- g(z): **activation function**
- the unit actually processes both the combination of weights and features and the activation function
- the output can be i) the final prediction, or ii) the intermediate output of a hidden layer

Activation functions: when and where?



- **when**: each time a unit is activated: input data (initial features, intermediate output) is processed and output is transferred to the next layer (or final output) through an activation function
- **where**: hidden layers and output layer

Activation functions: which?

- Identity function
- Logistic function
- Hyperbolic tangent function
- ReLU (Rectified Linear Unit) function
- Softmax function

Activation functions: identity function

1. $z = \Sigma(w \cdot x) + b$
2. $I(z)$

y

- identity function: a.k.a. **linear activation** function
- returns the value z that comes from the combination of input features and learned weights
- **never used**, except for the output layer in regression problems

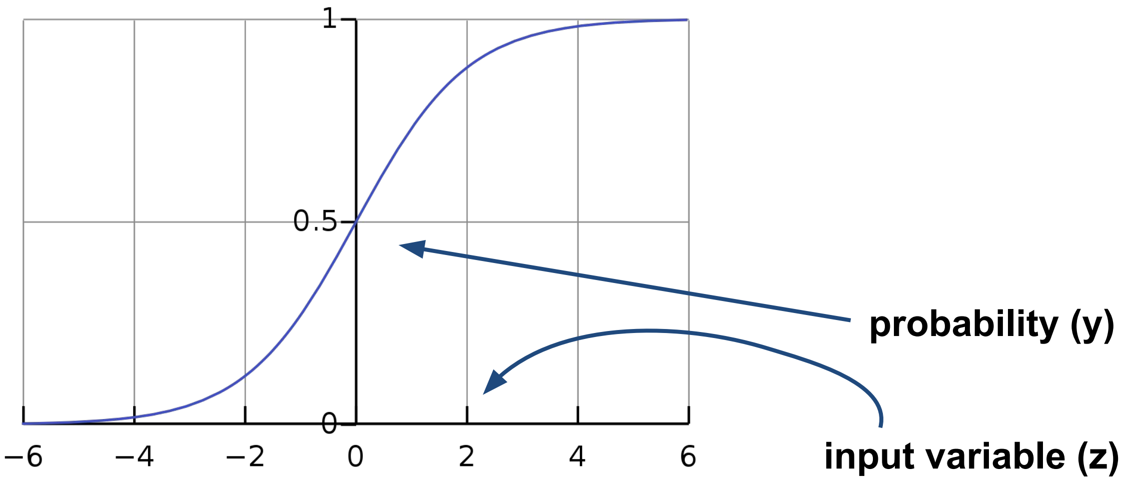
Activation functions: logistic function

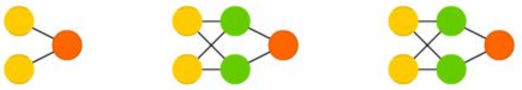
1. $z = \Sigma(w \cdot x) + b$
2. $\sigma(z)$

y

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

- logistic (**sigmoid**) function
- converts real input in [-∞,+∞] to output in the range [0,1]



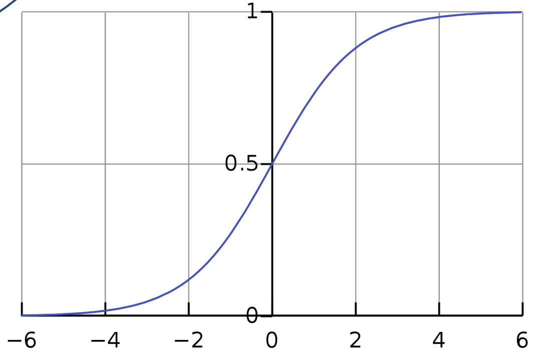


Activation functions: logistic function

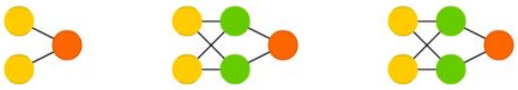
1. $z = \Sigma(w \cdot x) + b$
2. $\sigma(z)$

→ y

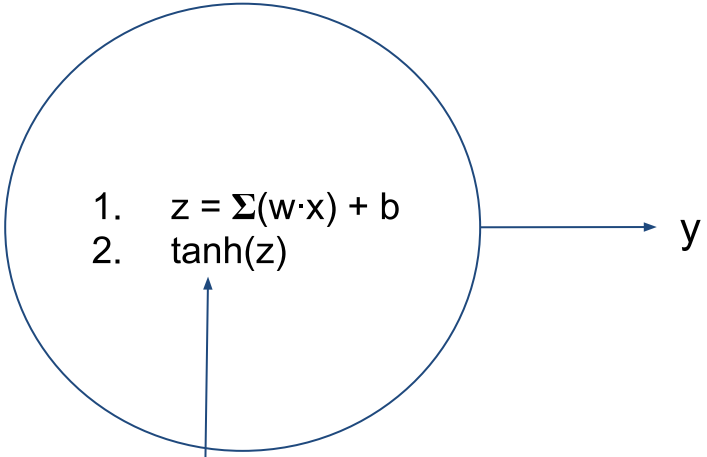
$$\frac{1}{1+e^{-z}}$$



- historically very popular
- now less popular → problems with gradient descent (solution of the model)
- when z is very large or very small derivatives are close to 0 → **slow descent**
- still used for the output layer in binary classification problems (and also for specialised hidden layers/units)



Activation functions: tanh

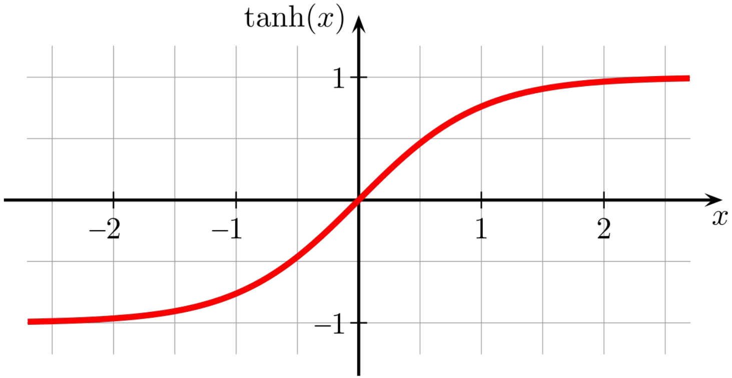


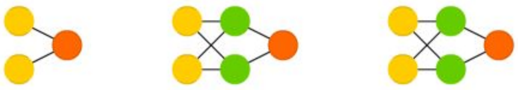
$$\frac{e^{2z}-1}{e^{2z}+1}$$

- hyperbolic tangent function
- rescaling of the logistic function:

  tanh = 2**σ**(2z)-1 [proof here]

- output in **[-1,+1]**, **mean 0,** ~ "centering of the data"





From: https://commons.wikimedia.org/wiki/File:Hyperbolic_Tangent.svg

Activation functions: tanh

1. z = Σ(w·x) + b
2. tanh(z)

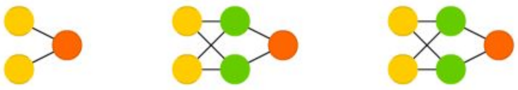
y

$$\frac{e^{2z}-1}{e^{2z}+1}$$

- hyperbolic tangent function
- rescaling of the logistic function:

  tanh = 2σ(2z)-1 [proof here]

- output in **[-1,+1]**, **mean 0,** ~ "centering of the data"
- more efficient learning in the intermediate hidden layers
- still suffers from similar limitations as σ(z) when z is very large or small
- used in specialized layers/units (e.g. RNN)



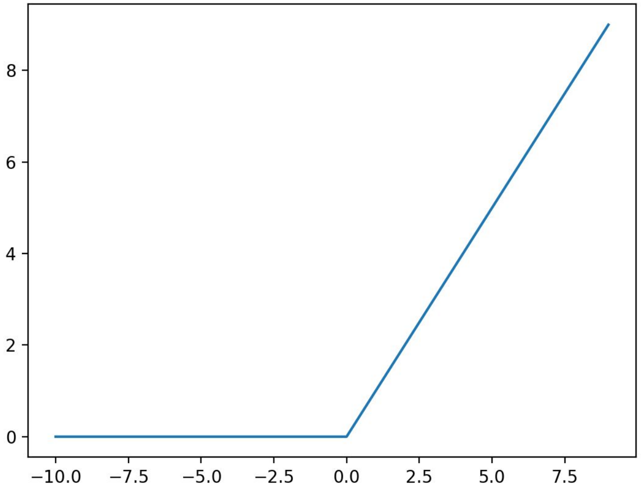
Activation functions: ReLU

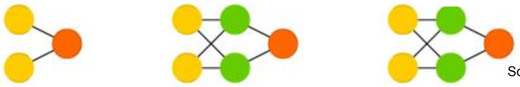
1. $z = \Sigma(w \cdot x) + b$
2. ReLU(z)

y

$$\max(0, z)$$

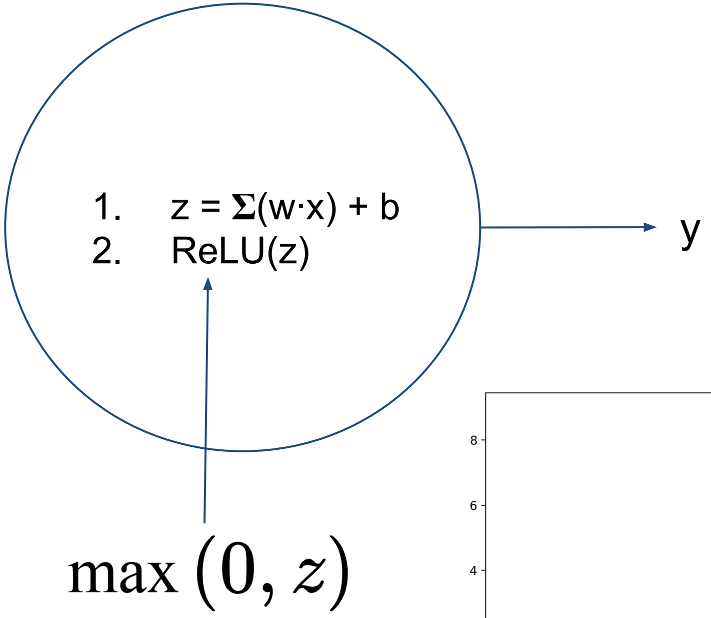
- derivative is 0 for $z < 0$, 1 for $z > 0$
- most common activation function (default choice in many cases)
- much faster and efficient learning of DL models

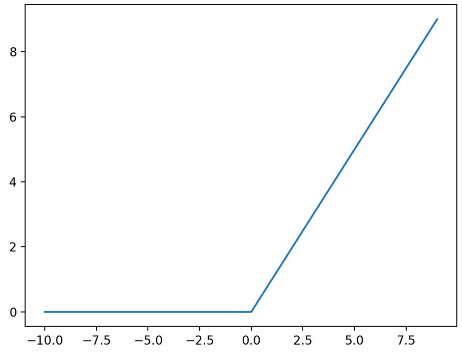




Source: https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks/

Activation functions: ReLU





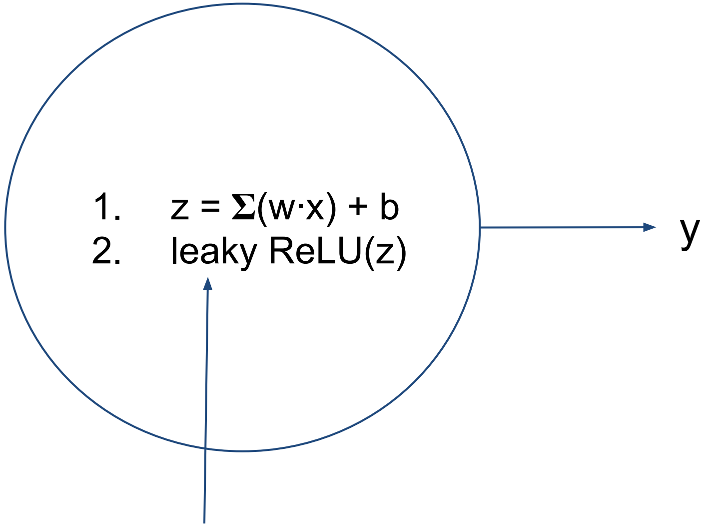
Pros of ReLU activation:

- easy to compute
- sparse representation: many output values will be exactly 0 (unlike sigmoid and tanh, which tends asymptotically to 0)
- reduces vanishing gradients → faster learning (training of multi-layered NNs)

→ ReLU is one of the ingredients that made deep learning possible

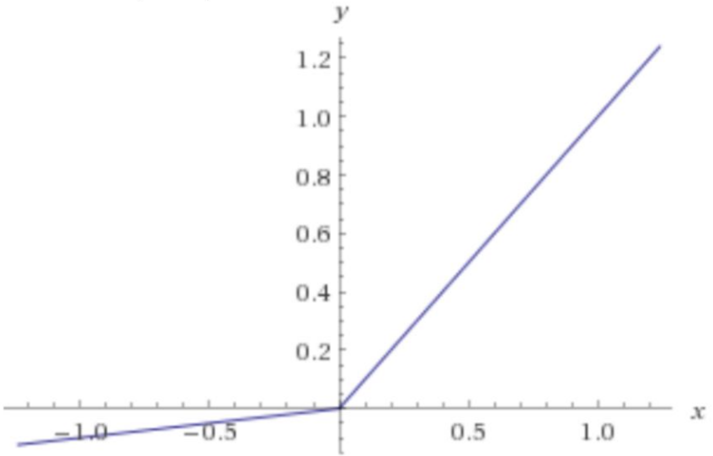
Source: https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks/

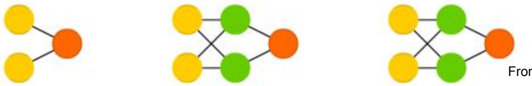
Activation functions: leaky ReLU



$$\max\,(0.01z,\, z)$$

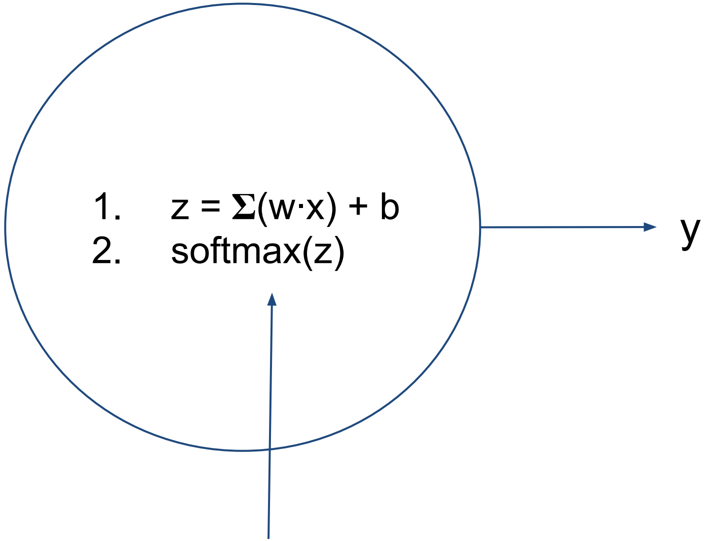
- uses a slight slope for z < 0
- can help when there are too many flat neurons (0 slopes, “dying neurons”), e.g.:
  - large negative bias
  - learning rate is too large





From: http://theprofessionalspoint.blogspot.com/2019/06/dying-relu-causes-and-solutions-leaky.html

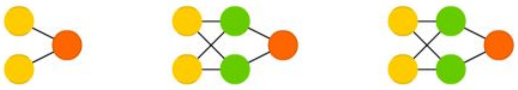
Activation functions: softmax



$$\mathrm{softmax}\left(z_i\right) = \frac{exp(x_i)}{\sum_{j=1}^{k} exp(x_j)}$$

- returns a probability distribution over the target classes in a **multiclass classification** problem
- k classes
- negative inputs converted to non-negative values (exponential function)
- each input will be in the interval [0,1]
- same denominator → normalization (sum to 1)

- Softmax is used in the output layer of multinomial classification problems
- Softmax is differentiable → backpropagation for optimization of the weights (parameters of the deep learning model)



Activation functions: why not linear?

- the linear (identity) activation function is never used: why?
- has to do with **function approximation**: NNs (deep learning) are excellent at finding complex non-linear relationships in the data (e.g. between features and target variables)
- with the identity activation function, the intermediate output of each layer will just be a linear combination of the input, and so no matter how many hidden layers you have, the final output **ŷ** will be a **linear combination** of the initial features **X**
- deep learning would then just be a very expensive way of doing linear regression!

$$\begin{cases} y_1 = w_1 x + b_1 \\ y_2 = w_2 y_1 + b_2 \end{cases} \rightarrow y_2 = w_2(w_1 x + b_1) + b_2 = \underbrace{(w_2 w_1)}_{\mathbf{w'}} x + \underbrace{(w_2 b_1 + b_2)}_{\mathbf{b'}}$$