DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

UNIVERSITÀ DEGLI STUDI DI PADOVA

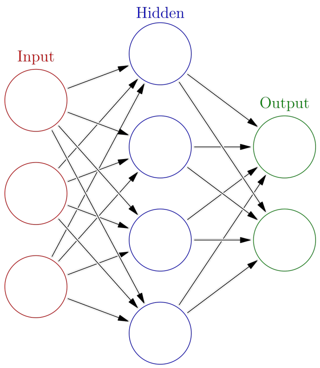
Neural Networks

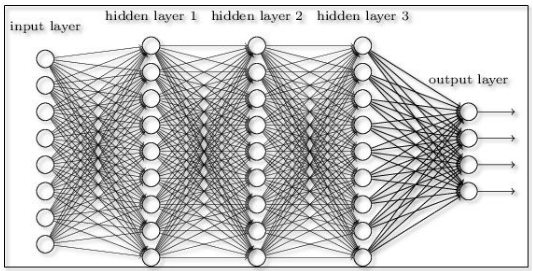
Machine Learning 2022-23

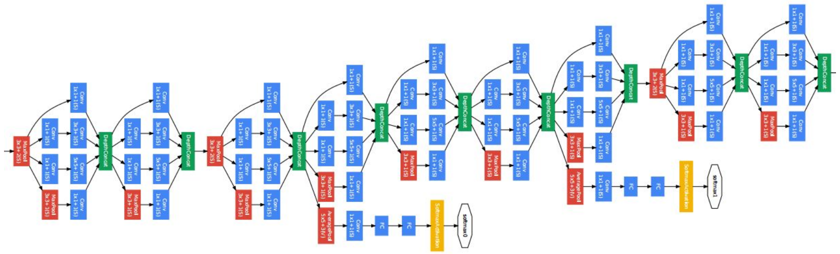
UML book chapter 20

Slides: F. Chiariotti, P. Zanuttigh, F. Vandin

Artificial Neural Networks

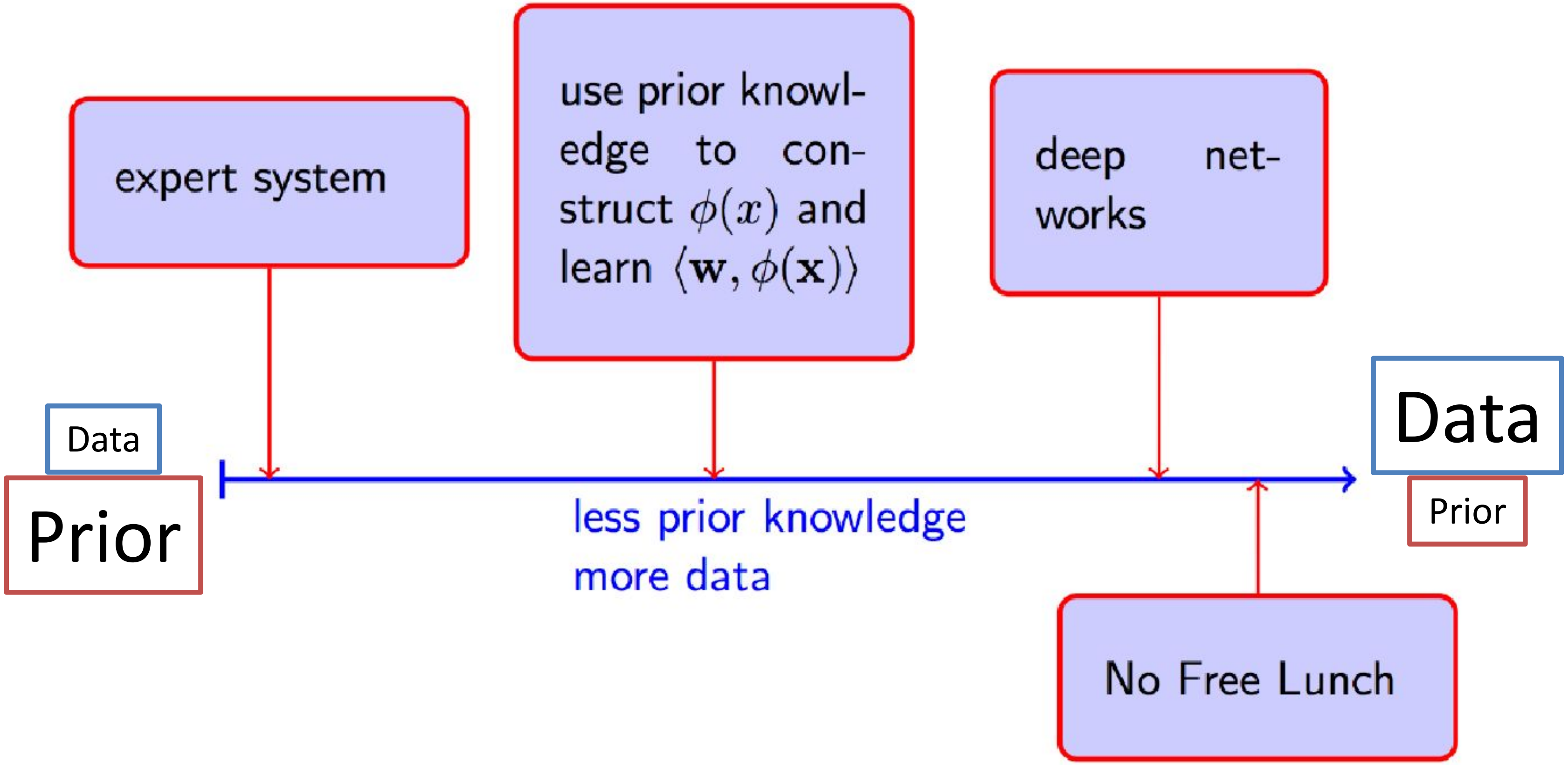






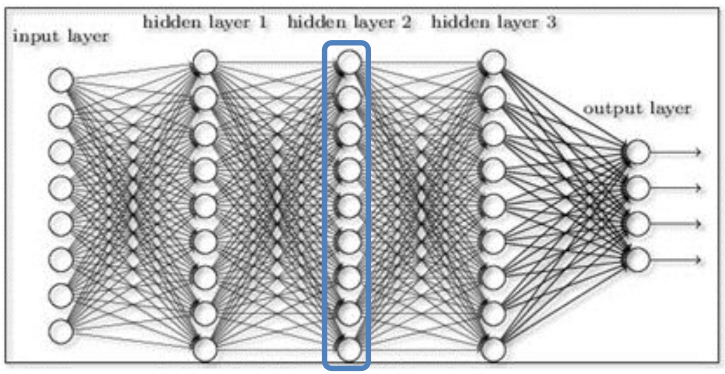
- Model of computation inspired by the structure of neural networks in the brain
- Large number of basic computing devices (neurons) connected to each other
- Neural Networks (NN) are represented with directed graphs where the nodes are the neurons and the edges corresponds to the links between the neurons
- Firstly proposed in 1940-50
- First practical applications in the 80-90s but practical results were lower than SVM and other techniques
- From 2010 on deep architectures with impressive performances

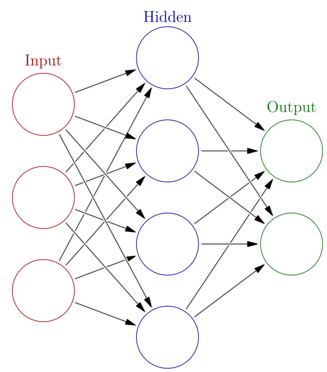
From Simple Algorithms to Deep Learning

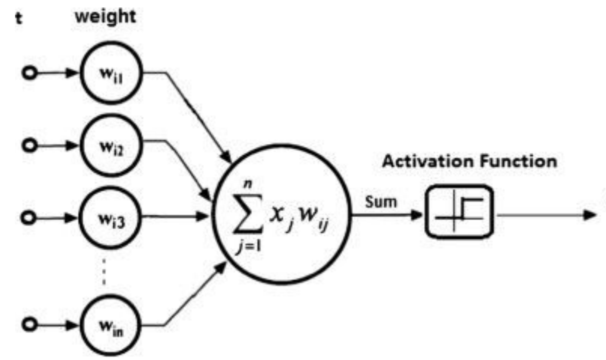


DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

Feedforward Neural Networks







Feedforward network: the graph representing the network has no cycles (data flows only in one direction)

The network is typically organized into layers: each neuron takes in input only the output of neurons of the previous layer

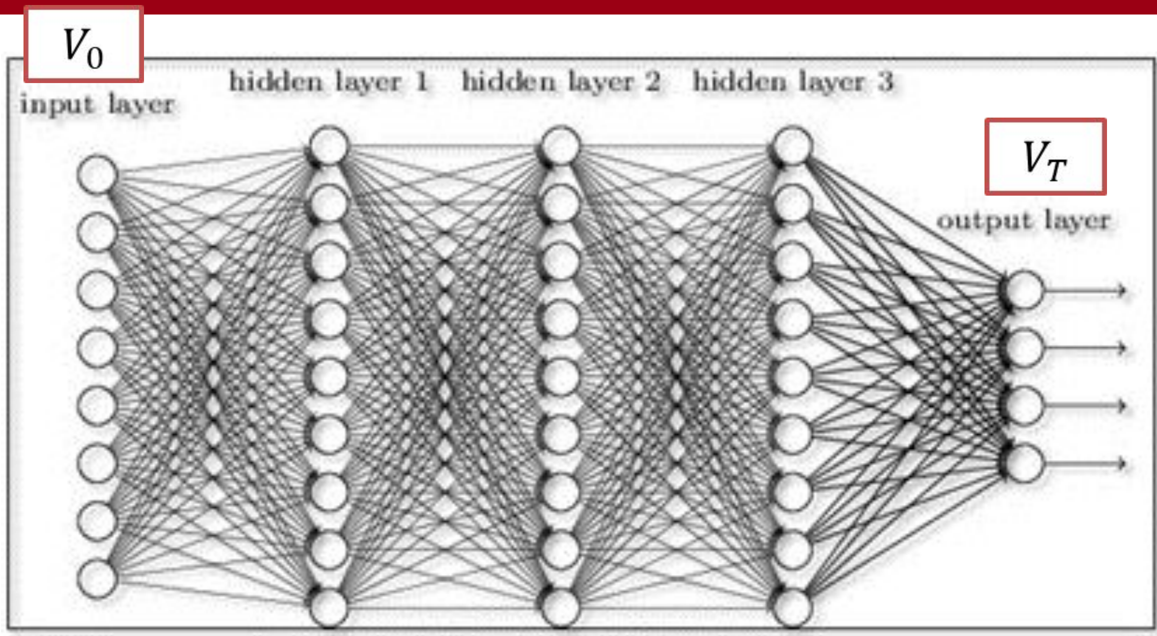
Notation (NN): Graph $G=(V,E)$ and function $w: E \rightarrow \mathbb{R}$

- V: neurons ($|V|$ is the size of the network)
- E: connections between neurons (directed edges)
- $w: E \rightarrow \mathbb{R}$ weight function over the edges (the weights w are the parameters to be learned)

Each neuron:

1. Takes in input the sum of the outputs of the connected neurons from previous layer weighted by the edge weights (w)
2. Applies to the result a simple scalar function (activation function, σ)

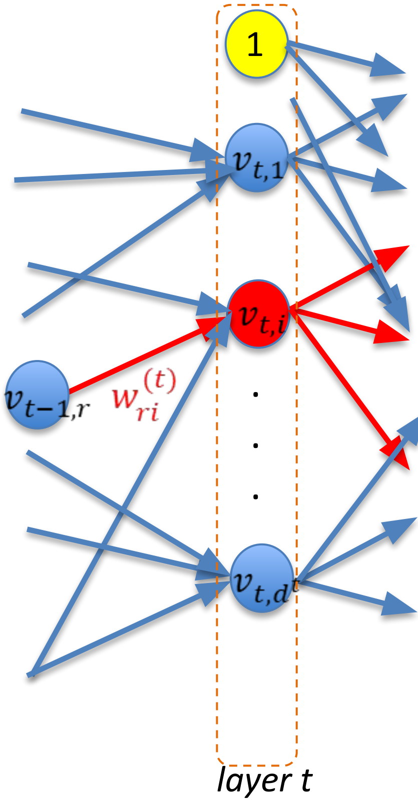
Notation (1)



Represent a network as the union of a set of (disjoint) layers: $V = \cup_{t=0}^{T} V_t$

- V_t , $t = 0, \ldots, T$: t-th layer,
- $d^t + 1$ number of nodes of layer t
  - "+1" : constant neuron (avoid bias, incorporate as in homogenous coord.)
- V_0: input layer , V_T: output layer, $V_1, \ldots, V_{T-1}$ inner (hidden) layers
- T : depth of the network
  - T=2 in "*classic*" NN, T>>2 in deep networks

Notation (2)



- $v_{t,i}$: *i-th* neuron in the *t-th* layer
- $\boldsymbol{v}^{(t)} = \left(1, v_{t,1}, \dots, v_{t,d^t}\right)^T$: all neurons of layer t
- Weights $w_{rj}^{(t+1)} = w(v_{t,r}, v_{t+1,j})$: weight of arc from neuron r of layer t to neuron j of layer t+1
- $\boldsymbol{w}_j^{(t)} = \left(w_{0j}^{(t)}, \dots, w_{d^{(t-1)}j}^{(t)}\right)^T$: all weights of arcs in input to neuron *j* of layer *t* (notice: from layer *t-1* to *t*)
- $w^{(t)}$: matrix of weights of all arcs incoming to layer t

$$w^{(t)} = \begin{bmatrix} w_{01}^{(t)} & w_{02}^{(t)} & \cdots & w_{0d^t}^{(t)} \\ w_{11}^{(t)} & w_{12}^{(t)} & & w_{1d^t}^{(t)} \\ \vdots & \vdots & \ddots & \vdots \\ w_{d^{(t-1)}1}^{(t)} & w_{d^{(t-1)}2}^{(t)} & \cdots & w_{d^{(t-1)}d^{(t)}}^{(t)} \end{bmatrix}$$

How a Neuron Works

- Compute output $o_{t,i}(\boldsymbol{x})$ of the *i-th* neuron in the *t-th* layer when ***x*** is fed to the network
  - Compact notation: use $v_{t,i}$ also to represent the output of the neuron
- The output of a neuron is a non-linear (activation) function applied to the linear combination of the inputs coming from the previous layer
  - σ: non-linear activation function
  - $a_{t+1,j} = < w_j^{(t+1)}, v^{(t)} >$ output of neuron before the activation function

$$o_{t+1,j}(x) = \sigma\left(\sum_{r:\ (v_{t,r}, v_{t+1,j}) \in E} w(v_{t,r}, v_{t+1,j})\, o_{t,r}(\boldsymbol{x}) \right) = \sigma\left(a_{t+1,j}(\boldsymbol{x})\right)$$

In vector notation:

$$v_{t+1,j} = \sigma\left(< \boldsymbol{w}_j^{(t+1)}, \boldsymbol{v}^{(t)} > \right) = \sigma(a_{t+1,j})$$

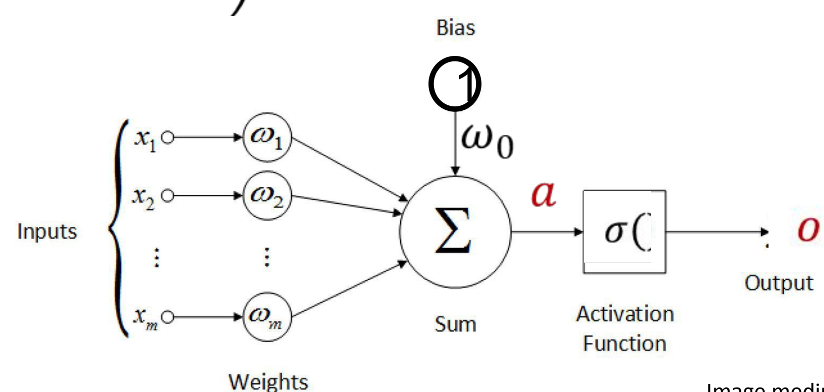


Image medium.com

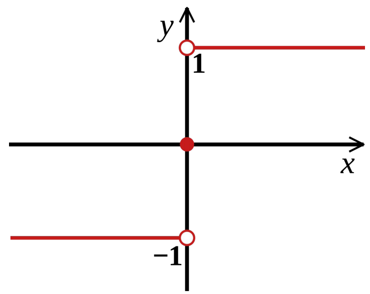
Activation Functions

- Various activation functions $\sigma(a)$ can be exploited:

1. Sign function
2. Threshold function
3. Sigmoid function
4. Hyperbolic Tangent
5. Rectified Linear Unit

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

Activation: Sign and Threshold



Outputs the sign of the input

$$\sigma(a) = sign(a)$$

$$\mathbb{R}^n \rightarrow [-1; 1]$$

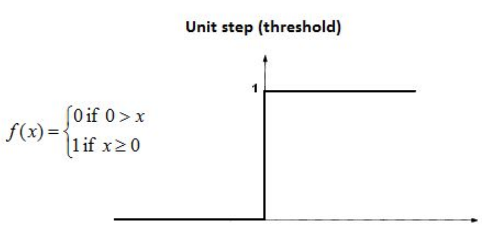
\+ Simple/fast

\+ Nice interpretation as the firing rate of a neuron

- -1 = not firing
- 1 = firing

\- Output is not smooth/continuous

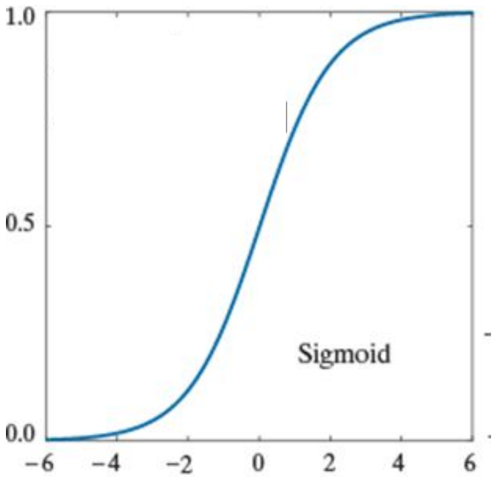
\- saturate and kill gradients, thus NN will barely learn



Threshold function: similar behaviour

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

Activation: Sigmoid



Takes a real-valued number and "squashes" it into range between 0 and 1

$$\sigma(a) = \frac{1}{1+e^{-a}}$$

$$\sigma'(a) = \sigma(a)[1-\sigma(a)]$$

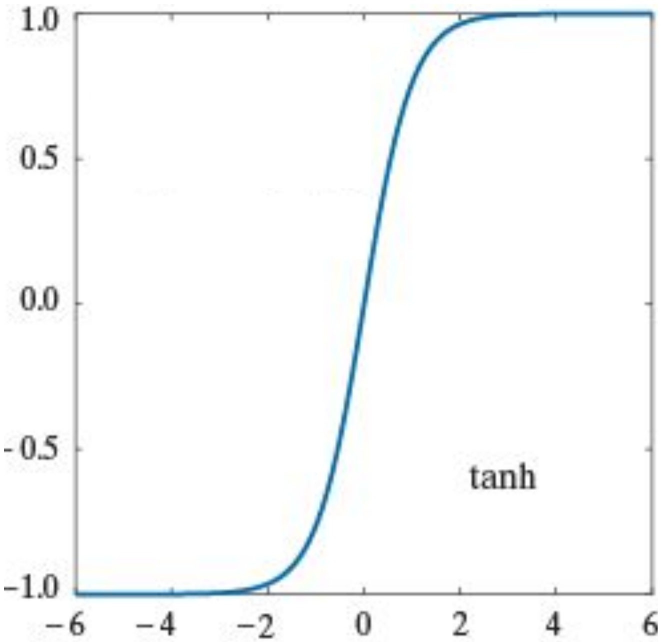
$$\mathbb{R}^n \rightarrow [0,1]$$

+ Smooth output
+ Nice interpretation as the firing rate of a neuron
    - 0 = not firing at all
    - 1 = fully firing

- Sigmoid neurons saturate and kill gradients, thus NN will have issues in learning
    - when the neuron's activation are 0 or 1 (saturate)
        - 🙁 gradient at these regions almost zero
        - 🙁 almost no signal will flow to its weights
        - 🙁 if initial weights are too large then most neurons would saturate

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

Activation: Tanh



Takes a real-valued number and "squashes" it into range between -1 and 1.

$$\sigma(a) = \tanh(a) = \frac{e^{a} - e^{-a}}{e^{a} + e^{-a}} = \frac{e^{2a} - 1}{e^{2a} + 1}$$

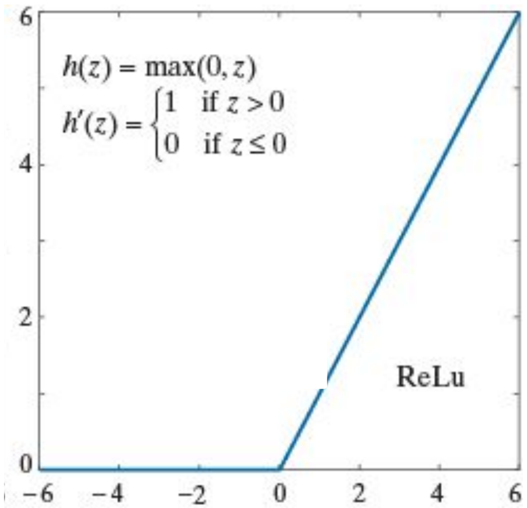
$$\sigma'(a) = 1 - [tanh(a)^2]$$

$$\mathbb{R}^n \rightarrow [0,1]$$

Tanh is a scaled and shifted sigmoid: $tanh(x) = 2sigm(2x) - 1$

- \+ Like sigmoid, tanh neurons saturate
- \- Unlike sigmoid, output is zero-centered

Activation: ReLU



Takes a real-valued number and thresholds it at zero

$$\sigma(a) = \max(0, a) = \begin{cases} a \ \ if \ a > 0 \\ 0 \ if \ a \leq 0 \end{cases}$$

$$\sigma'(a) = \begin{cases} 1 \ \ if \ a > 0 \\ 0 \ if \ a \leq 0 \end{cases}$$

$$\mathbb{R}^n \to \mathbb{R}^n_+$$

Most Deep Networks use ReLU nowadays

+ Trains much faster
  - accelerates the convergence of SGD
  - due to linear, non-saturating form
+ Less expensive operations
  - compared to *sigmoid*/*tanh* (exponentials etc.)
  - implemented by simply thresholding a matrix at zero
+ More expressive
+ Prevents the gradient vanishing problem

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

Forward Propagation

$$v^{(0)} \rightarrow v^{(1)} \rightarrow v^{(2)} \rightarrow \cdots \rightarrow v^{(T)}$$

Take an input sample and compute the output of the network

Start from the input (layer 0)…
….compute the output of layer 1, send to layer 2 and get output….

…. through all the layers up to the output layer

Input: $\mathbf{x} = (x_1, \ldots, x_d)^T$; NN with 1 output node
Output: prediction y of NN;

$v^{(0)} \leftarrow (1, x_1, \ldots, x_d)^T$; — 1st layer: read input

for $t \leftarrow 1$ **to** T **do** — From first to last layer

$a^{(t)} \leftarrow \left(w^{(t)}\right)^T v^{(t-1)}$; — linear part

$v^{(t)} \leftarrow \left(1, \sigma\left(a^{(t)}\right)^T\right)^T$; — activation function («1» for bias)

$y \leftarrow v^{(T)}$;

return y;

Learning Neural Networks

Neural Network (NN): (V,E,σ,w)

- Corresponds to a function $h_{V,E,\sigma,w}: \mathbb{R}^{|V_0-1|} \rightarrow \mathbb{R}^{|V_T|}$
- The hypothesis class of a network is defined by fixing its architecture:

$$\mathcal{H}_{V,E,\sigma} = \{h_{V,E,\sigma,w} : w \; is \; a \; mapping \; from \; E \; to \; \mathbb{R}\}$$

- V,E,σ defines the architecture of the network
- w contains the parameters that are going to be learned
- Training of the NN: finding the optimal set of weights w

Expressive Power of NN (boolean functions)

Proposition

For every d, there exists a graph (V, E) of depth 2 such that $\mathcal{H}_{V,E,\text{sign}}$ contains all functions from $\{-1, 1\}^d$ to $\{-1, 1\}$

NN can implement every boolean function!

Unfortunately the graph (V, E) is very big...

Proposition

For every d, let $s(d)$ be the minimal integer such that there exists a graph (V, E) with $|V| = s(d)$ such that $\mathcal{H}_{V,E,\text{sign}}$ contains all functions from $\{-1, 1\}^d$ to $\{-1, 1\}$. Then $s(d)$ is an exponential function of d.

Note: similar result for σ = sigmoid

Recall: Boolean functions include any function that can be implemented in a computer

Expressive Power of NN (demonstration)

Consider sign activation $\mathcal{H}_{V,E,sign}$

1. Use this 3 layers NN: $\begin{cases} INPUT: \; |V_0| = n+1 \\ HIDDEN: |V_1| = 2^n + 1 \\ OUTPUT: \; |V_2| = 1 \end{cases}$
2. Define: $\boldsymbol{u}_1, \ldots, \boldsymbol{u}_k \in \{\pm 1\}^n$: all input vectors leading to an output of 1
3. Notice: $\langle \boldsymbol{x}, \boldsymbol{u}_i \rangle = \begin{cases} n \quad\quad\quad if \; \boldsymbol{x} = \boldsymbol{u}_i & (all\ bits\ match) \\ \leq n-2 \; if \; \boldsymbol{x} \neq \boldsymbol{u}_i & (each\ mismatch\ -2\ penalty) \end{cases}$
4. Define $g_i = sign(\langle \boldsymbol{x}, \boldsymbol{u}_i \rangle - n + 1) = \begin{cases} 1 & if \, \boldsymbol{x} = \boldsymbol{u}_i \\ -1 & otherwise \end{cases}$
5. Adapt weights w to get g_i in the hidden layer
   → each hidden layer neuron looks if the input is u_i
6. Output layer: $f(\boldsymbol{x}) = sign\left(\sum_{i=1}^{k} g_i(\boldsymbol{x}) + k - 1\right)$ (sign is 1 if at least one is true)

Notice: network exponentially large, works but «brute-force» solution probably leading to overfitting

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

Expressive Power of NN (real valued functions)

Proposition

For every fixed $\varepsilon > 0$ and every Lipschitz function $f : [-1,1]^d \to [-1,1]$ it is possible to construct a neural network such that for every input $\mathbf{x} \in [-1,1]^d$ the output of the neural network is in $[f(\mathbf{x}) - \varepsilon, f(\mathbf{x}) + \varepsilon]$.

Note: first result proved by Cybenko (1989) for sigmoid activation function, requires only 1 hidden layer!

NNs are **universal approximators!**

Not part of the course

But again...

Proposition

Fix some $\varepsilon \in (0,1)$. For every d, let $s(d)$ be the minimal integer such that there exists a graph (V,E) with $|V| = s(d)$ such that $\mathcal{H}_{V,E,\sigma}$, with $\sigma =$ sigmoid, can approximate, with precision ε, every 1-Lipschitz function $f : [-1,1]^d \to [-1,1]$. Then $s(d)$ is exponential in d.

Implement Conjunction and Disjunction with NN

- NN can implement boolean AND / OR
- Consider sign activation and *k* inputs with values ±1

Conjunction (AND)

$$f(\boldsymbol{x}) = sign\left(1 - k + \sum_{i=1}^{k} \boldsymbol{x}_i\right)$$

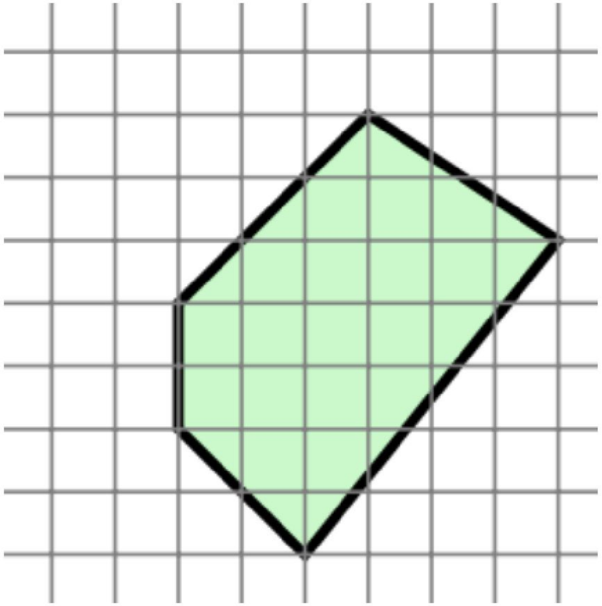
(positive if all positive, AND)

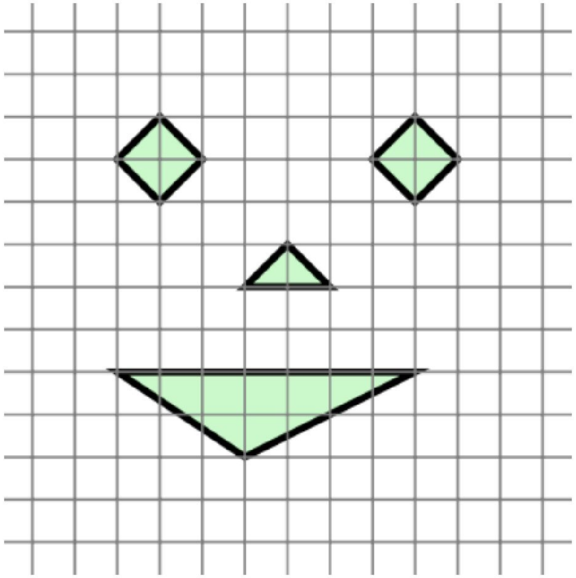
Disjunction (OR)

$$f(\boldsymbol{x}) = sign\left(k - 1 + \sum_{i=1}^{k} \boldsymbol{x}_i\right)$$

(positive if at least one positive, OR)

Expressive Power of NN (example)





- Input in $\mathbb{R}^2$, 2-layer NN
- *k* neurons, sign activation
- Each neuron: an halfspace
- Intersection of halfspaces
- Convex polytopes with k-1 faces

- Input in $\mathbb{R}^2$, 3-layer NN
- *k* neurons, sign activation
- Each neuron: an halfspace
- Intersection and unions of halfspaces
- Union of polytopes

VC dimension of NN

- With *sign* activation

VC dimension of $\mathcal{H}_{V,E,sign} = O(|E|\log|E|)$ *(no demonstration)*

- With *Sigmoid (σ)* activation

VC dimension of $\mathcal{H}_{V,E,\sigma} = O(|V|^2|E|^2)$ *(no demonstration)*

→Large NNs require a lot of data !

If we have enough data, what about the computation time ?

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

Runtime of NN

Applying the ERM rule to a NN (V,E,σ,w) is computationally difficult, even for relatively small NN...

Not part of the course

Theorem:

Hypothesis: Let $k \geq 3$. For every *d*, let *(V,E)* be a layered graph with *d* input nodes, *k+1* nodes at the (only) hidden layer (where one of them is the constant neuron), and a single output node.

Thesis: It is NP-hard to implement the ERM rule with respect to $\mathcal{H}_{V,E,sign}$

(no demonstration)

Even approximations of ERM rule are infeasible
Also by changing the activation things do not get better

Need a different strategy....

SGD and backpropagation algorithm !

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

NN optimization

- Target of ERM: given training data $(\boldsymbol{x_1}, y_1), \dots, (\boldsymbol{x_m}, y_m)$ find the weights that minimize the training error:

$$L_S(h) = \frac{1}{m} \sum_{i=1}^{m} \ell(h, (\boldsymbol{x_i}, y_i))$$

- The problem is challenging !

Idea:

1. Forward propagate the training data and compute the loss
2. Consider the loss as a function of the weights and compute the gradient of the loss w.r.t. the weights
3. Update the weights with SGD

Good Idea! But *we need the gradient of the loss w.r.t. the weights*

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

SGD fror NN: Algorithm

SGD for Neural Networks

parameters:

Number of iterations τ

Step size sequence $\eta_1, \eta_2, \dots, \eta_\tau$

Regularization parameter $\lambda > 0$

Input:

Network : layered graph *G=(V,E)*

differentiable activation function σ: $\mathbb{R} \to \mathbb{R}$

Algorithm:

chose $\boldsymbol{w}^{[1]} \in \mathbb{R}^{|E|}$ at random

(from a distribution s.t. $\boldsymbol{w}^{[1]}$ is close enough to **0**)

for $s = 1,2,..,\tau$

sample $(\boldsymbol{x}, \boldsymbol{y}) \sim \mathcal{D}$

calculate gradient $\boldsymbol{v}_s = backpropagation(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{w}, (V, E), \sigma)$

update $\boldsymbol{w}^{[s+1]} = \boldsymbol{w}^{[s]} - \eta_s(\boldsymbol{v}_s + \lambda \boldsymbol{w}^{[s]})$

regularization

adaptive learning rate

output:

$\overline{\boldsymbol{w}}$ is the best performing $\boldsymbol{w}^{[s]}$ on a validation set

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

SGD for NN: How to Compute the Gradient

Update Rule (baseline version)

layer

iteration index

$$w_{ij}^{(t)[s+1]} = w_{ij}^{(t)[s]} - \eta_s \frac{\partial L_s}{\partial w_{ij}^{(t)[s]}}$$

learning rate

gradient of the loss w.r.t each single weight

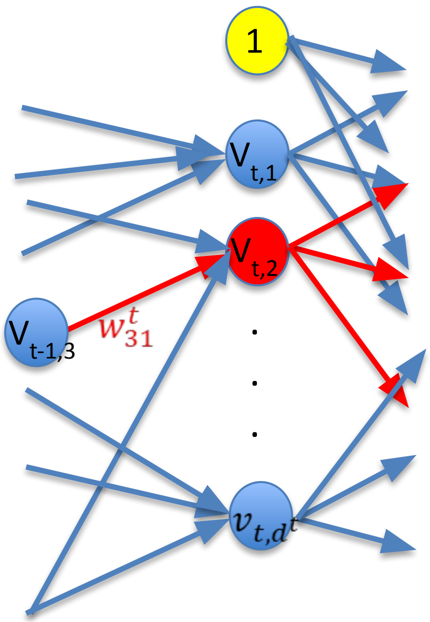
$$\frac{\partial L_S}{\partial w} = \frac{\partial}{\partial w}\left(\frac{1}{m}\sum_{i=1}^{m} \ell(h, (\boldsymbol{x}_i, y_i))\right) = \frac{1}{m}\sum_{i=1}^{m} \frac{\partial L(h,(\boldsymbol{x}_i,y_i))}{\partial w}$$

- We need the gradient w.r.t. each single weight in the network
- But we can compute the loss only on the output (i.e., after the last layer)
- Recall that each neuron contains also the non-linear activation function

$$\boldsymbol{\delta}^{(t)} = \frac{\partial L}{\partial \boldsymbol{a}^{(t)}} = \begin{bmatrix} \frac{\partial L}{\partial a_{t,1}} \\ \vdots \\ \frac{\partial L}{\partial a_{t,d^t}} \end{bmatrix}$$

$\boldsymbol{\delta}^{(t)}$: change in error w.r.t. to the weighted average before the non-linear transformation

BackPropagation (1)



- Decompose the gradient with the chain rule
  - Recall $a_{t,j} = \sum_{k=0}^{d^{(t-1)}} w_{kj}^{(t)} v_{t-1,k}$
  - Each weight $w_{ij}^{(t)}$ impacts only on $a_{t,j}$
  - We need $\boldsymbol{\delta}^{(t)} = \frac{\partial L}{\partial \boldsymbol{a}^{(t)}}$ to compute the gradient
  - σ' depends on the selected activation function

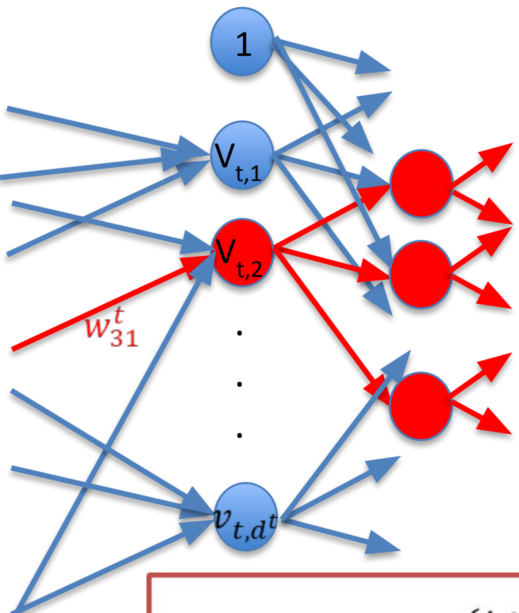
$$\frac{\partial L}{\partial w_{ij}^{(t)}} = \frac{\partial L}{\partial a_{t,j}} \frac{\partial a_{t,j}}{\partial w_{ij}^{(t)}} = \delta_j^{(t)} \frac{\partial}{\partial w_{ij}^{(t)}} \left(\sum_{k=0}^{d^{(t-1)}} w_{kj}^{(t)} v_{t-1,k} \right) = \delta_j^{(t)} \mathrm{v}_{t-1,i}$$

remains only k=i term

$$\delta_j^{(t)} = \frac{\partial L}{\partial v_{t,j}} \frac{\partial v_{t,j}}{\partial a_{t,j}} = \frac{\partial L}{\partial v_{t,j}} \sigma'(a_{t,j})$$

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

BackPropagation (2)



- Understand how the loss changes w.r.t. $v_{t,j}$
  - Change in layer *t* affects only neurons in layer *t+1* (and then each following layer up to the loss at the end)
  - Each neuron can affect all the neurons in next layer
  - Need to sum contributions to all the neurons in layer *t+1*
  - To compute $\delta^{(t)}$ we need $\delta^{(t+1)}$ (solution of the next layer)

$$\frac{\partial L}{\partial v_{t,j}} = \sum_{k=1}^{d^{(t+1)}} \frac{\partial L}{\partial a_{t+1,k}} \frac{\partial a_{t+1,k}}{\partial v_{t,j}} = \sum_{k=1}^{d^{(t+1)}} w_{jk}^{(t+1)} \delta_k^{(t+1)}$$

$$\delta_j^{(t)} = \frac{\partial L}{\partial v_{t,j}} \frac{\partial v_{t,j}}{\partial a_{t,j}} = \sigma'(a_{t,j}) \sum_{k=1}^{d^{(t+1)}} w_{jk}^{(t+1)} \delta_k^{(t+1)}$$

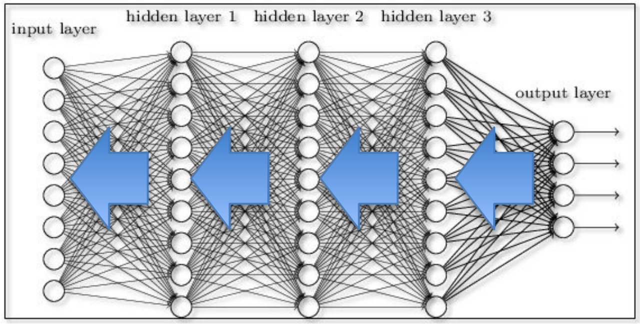
DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

BackPropagation (3)

$$\boxed{\delta_j^{(t)} = \sigma'(a_{t,j}) \sum_{k=1}^{d^{(t+1)}} w_{jk}^{(t+1)} \delta_k^{(t+1)}}$$

- The solution for each layer need the solution of the following one
- Start from the last layer ($\delta^{(L)}$ can be computed from the loss on the output)
- Backpropagate the gradients through all the layers up to the first

$$\boxed{v^{(0)} \leftarrow v^{(1)} \leftarrow v^{(2)} \leftarrow \dots \leftarrow v^{(T)}}$$



BackPropagation: Algorithm

Input: data point $(\mathbf{x}_i, y_i)$, NN (with weights $w_{ij}^{(t)}$, for $1 \leq t \leq T$)

Output: $\delta^{(t)}$ for $t = 1, \dots, T$

compute $a^{(t)}$ and $v^{(t)}$ for $t = 1, \dots T$;

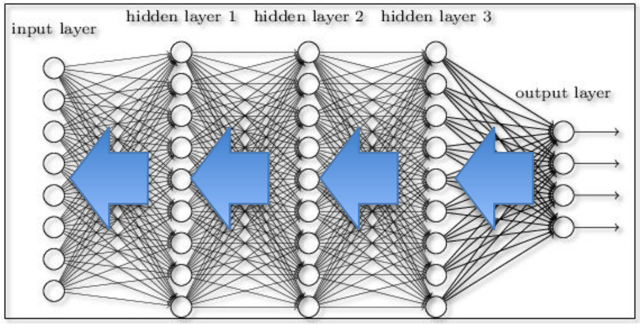
$\delta^{(L)} \leftarrow \frac{\partial L}{\partial a^{(L)}}$;

for $t = T - 1$ **downto** 1 **do**

  $\delta_j^{(\ell)} \leftarrow \sigma'(a_{t,j}) \cdot \sum_{k=1}^{d^{(\ell+1)}} w_{jk}^{(t+1)} \delta_k^{(t+1)}$ for all $j = 1, \dots, d^{(t)}$;

return $\delta^{(1)}, \dots, \delta^{(T)}$;

$$v^{(0)} \leftarrow v^{(1)} \leftarrow v^{(2)} \leftarrow \dots \leftarrow v^{(T)}$$



DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

NN Training: complete algorithm

BackPropagation algorithm with SGD

Input: training data $(x_1, y_1), \dots, (x_m, y_m)$

Output: NN weights $w_{ij}^{(t)}$

Initialize $w_{ij}^{(t)} \quad \forall i, j, t;$

for $s \leftarrow 0,1,2, \dots$ **do** // until convergence

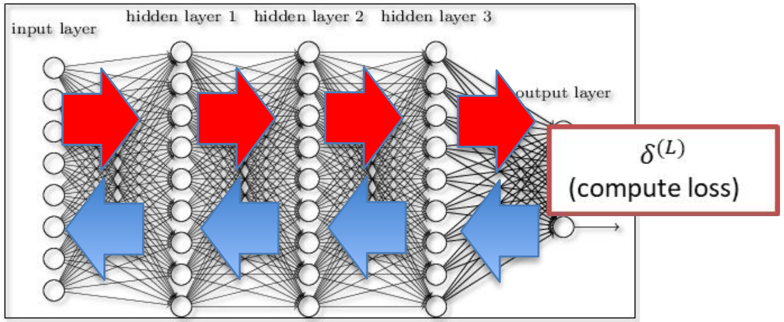
pick (x_k, y_k) at random from training data; // SGD: 1 sample at random

compute $v_{t,j} \quad \forall j, t;$ // forward propagation

compute $\delta_j^{(t)} \quad \forall j, t;$ // backward propagation

$w_{ij}^{(t)[s+1]} = w_{ij}^{(t)[s]} - \eta v_{t-1,i} \delta_j^{(t)} \quad \forall i, j, t;$ // update weights

if *converged* **then return** $w_{ij}^{(t)[s]} \quad \forall i, j, t;$



DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

NN Training Details: Pre-processing and Initialization

Pre-processing:

- Typically all inputs are normalized and centred around 0
- Both local or global normalization strategies

Initialization of the weights

- All to 0 does not work
- Random values around 0 (regime where model is roughly linear)
- Uniform or normal (Gaussian) distribution can be used
- Sometimes multiple initializations and trainings, then select best result (smallest training error)
- In deep NN "*Glorot*" initialization: normal distribution with variance inversely proportional to the sum of the number of incoming and outcoming connections of the neuron

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

NN Training details: When to Stop

When to stop?

- Small training error
- Small marginal improvement in error at each step
- Upper bound on number of iterations

Loss function usually has multiple local minima

- With highly dimensional spaces the risk is smaller than in low dimensional ones, but no guarantee
- Run stochastic gradient descent (SGD) from different (random) initial weights

Regularization

- Minimize weighted sum of the loss with the sum of all the weights
- Avoid too large weights and make optimization more stable
- Regularization parameter λ
- L1 or L2 regularization can be used

$$\boxed{L_s(h) + \frac{\lambda}{m}\sum_{i,j,t}\left|w_{ij}^{(t)}\right|}$$

$$\boxed{L_s(h) + \frac{\lambda}{m}\sum_{i,j,t}\left(w_{ij}^{(t)}\right)^2}$$