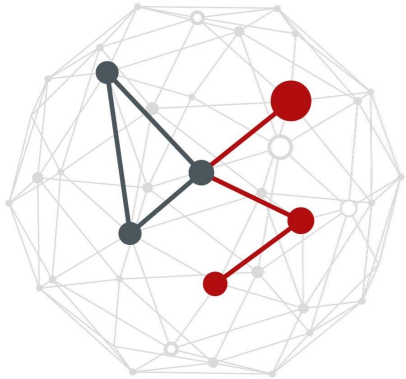
MIME
Master's degree ICT Internet Multimedia Engineering

FEED FORWARD NEURAL NETWORKS (FFNN)

Michele Rossi
michele.rossi@dei.unipd.it

Dept. of Information Engineering
University of Padova, IT



DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

DIPARTIMENTO MATEMATICA

800 ANNI 1222·2022

UNIVERSITÀ DEGLI STUDI DI PADOVA

Outline

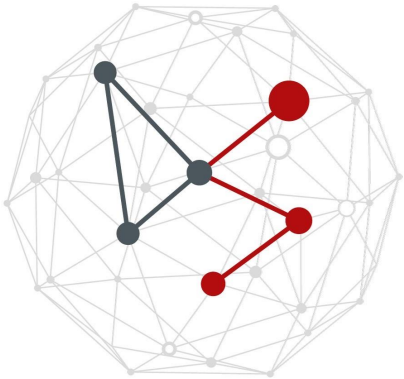
- Introduction, the perceptron algorithm
- The artificial neuron
- Feed Forward Neural Networks (FFNN)
  - Structure
  - Output layer
  - Error functions
- Training FFNN
  - backpropagation
- Bibliography

- Appendix A: calculation for softmax
- Appendix B: calculations for tanh

MIME

Master's degree ICT Internet Multimedia Engineering

INTRODUCTION



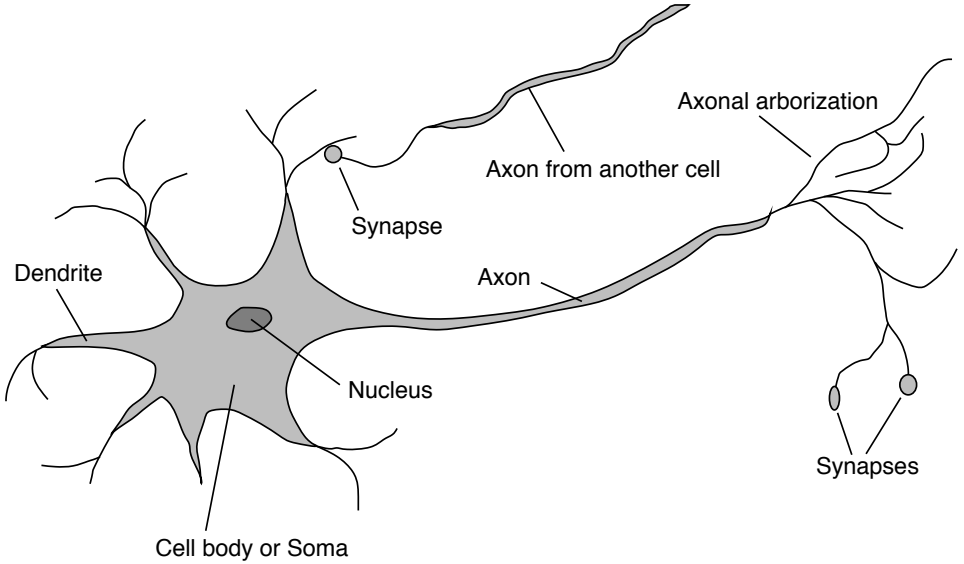
DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

DIPARTIMENTO MATEMATICA

1222 · 2022 800 ANNI

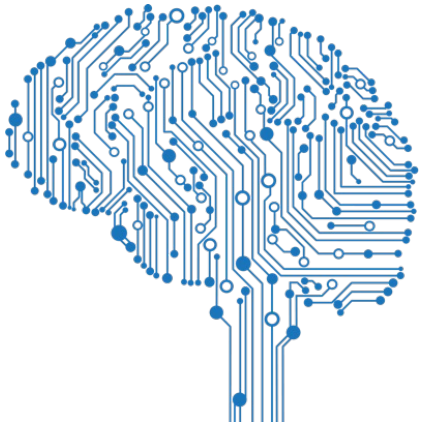
UNIVERSITÀ DEGLI STUDI DI PADOVA

The brain



- The brain
  - is responsible for all processing and memory
  - consists of a complex network of cells called neurons
  - neurons communicate by TX electrochemical signals through the net
  - each input signal to a neuron can inhibit or excite it
  - when the neuron is excited enough, it fires its own signal
- 10^{11} neurons of >20 types, 10^{14} synapses, 1-10ms cycle time
  - Signals are noisy “spike trains” of electrical potentials

Mathematical model



- Artificial Neural Network (ANN)
  - is a network of interconnected artificial neurons
  - artificial neurons communicate by sending signals to one another
  - each input to an artificial neuron can either *inhibit* or *excite* it
- Topology
  - Is simplified to keep complexity low
  - Multiple-layers of neurons, stacked

ANN (1st gen) *vs* brain

1st generation of neural networks

ANN	Brain
Feedforward	Recurrent
Fully-connected	Mostly local connections
Uniform structure	Functional modules
A few node types	Hundreds of neuron types
10-10,000 nodes	$O(10^{11})$ neurons, $O(10^{15})$ synapses
Static	Dynamic: spike trains

ANN (2ng gen) *vs* brain

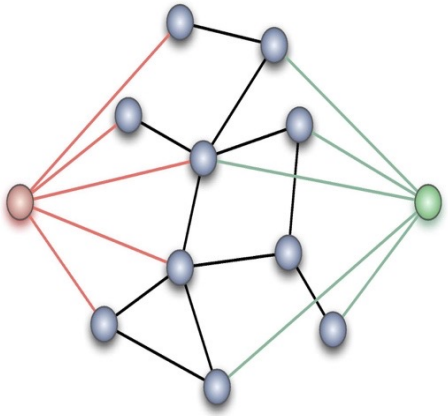
2nd generation neural networks (Deep networks)

ANN	Brain
Feedforward / recurrent	Recurrent
Fully-connected / sparse / local	Mostly local connections
Modular structure	Functional modules
Some node types	Hundreds of neuron types
Millions of nodes	$O(10^{11})$ neurons, $O(10^{15})$ synapses
Static	Dynamic: spike trains

Building an ANN

- Network topology
  - Connectivity
  - Information flow path
- Neuron types
  - Type of activation function
- Learning approach
  - Supervised
  - Unsupervised
- Learning algorithm
  - Backpropagation

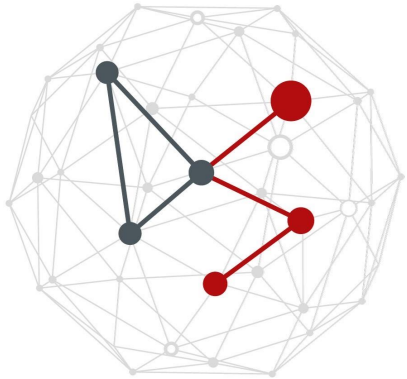
Topology



- There are many possible topologies
  - But not all are computationally tractable
  - Especially, learning in arbitrary topologies is difficult
- The most popular are
  - Feed Forward Neural Networks FFNN
  - Recurrent Neural Networks RNN
- These
  - Can be trained effectively
  - Have been used to *solve many practical problems*
  - Are being widely adopted in real products

MIME
Master's degree ICT Internet Multimedia Engineering

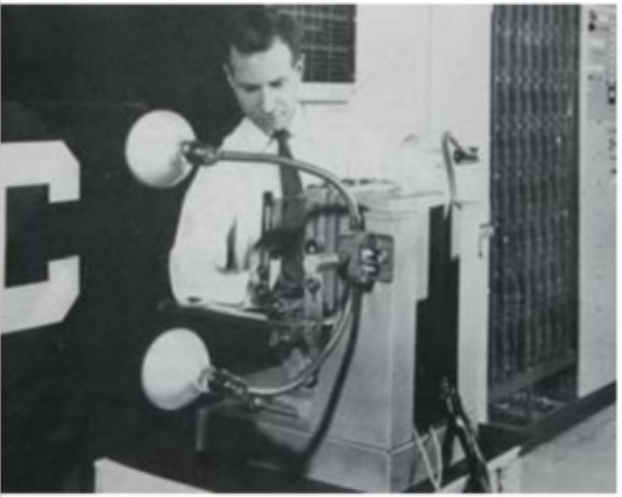
HISTORICAL NOTES: THE PERCEPTRON ALGORITHM

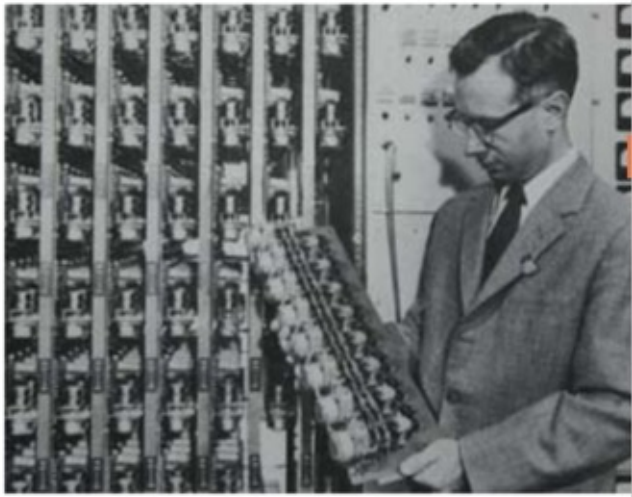


DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE
DIPARTIMENTO MATEMATICA
1222 · 2022 800 ANNI
UNIVERSITÀ DEGLI STUDI DI PADOVA

Frank Rosemblatt [Ros62]

- In 1957, Frank Rosemblatt invented the perceptron algorithm (funded by the US Office of Naval Research, ONR)
- In 1958 he built the Mark I
  - The first learning machine for image recognition
  - It had an array of 400 photocells, randomly connected to the "neurons". Weights were encoded in *potentiometers*, and weight updates during learning were performed by *electric motors*





The perceptron algorithm (1/7) [Ros62]

- Milestone in ANN theory (classification problem)
  - Input vector $\boldsymbol{x} = [x_1 \; x_2 \; \ldots \; x_D]^T$
  - Is first transformed using a non-linear transformation
    - To obtain a feature vector $\phi(\boldsymbol{x}) = [\phi_1(\boldsymbol{x}) \; \phi_2(\boldsymbol{x}) \; \ldots \; \phi_D(\boldsymbol{x})]^T$
    - Which can also be seen as a set of basis functions
  - Then, a generalized linear model is constructed, as follows:

$$y(\boldsymbol{x}) = f(\boldsymbol{w}^T \phi(\boldsymbol{x}))$$

    - $\boldsymbol{w}$ is a weight vector
    - $f(a) = f(\boldsymbol{w}^T \phi(\boldsymbol{x}))$ is a non-linear (classification) function:

$$f(a) = \begin{cases} +1 & a \geq 0 \text{ (class C1)} \\ -1 & a < 0 \text{ (class C2)} \end{cases}$$

The perceptron algorithm (2/7)

- Label
  - A variable, whose value depends on the true class to which $\boldsymbol{x}$ belongs

$$t = \begin{cases} +1 & \text{class C1} \\ -1 & \text{class C2} \end{cases}$$

- Goal
  - Find the weight vector that, however, minimizes the classification error
- Note that
  - Minimizing the number of misclassified patterns does not lead to a simple learning algorithm, as the error as a function of **w** is a piecewise constant function with discontinuities
  - In this case, adjusting **w** based on the gradients cannot be done, as the gradient is zero almost everywhere (see later)

Training set

- Discrete time n
  - Input data vector $\boldsymbol{x}_n$
  - Associated label (class) t_n
  - Training set (data and associated label)

$$\{(\boldsymbol{x_1}, t_1), (\boldsymbol{x}_2, t_2), \ldots, (\boldsymbol{x}_N, t_N)\}$$

The perceptron algorithm (3/7)

- Error function
  - We build a special error function such that (given the shape of f(a)):

$$\text{patterns } \boldsymbol{x}_n \text{ in class C1 must have: } \boldsymbol{w}^T \phi(\boldsymbol{x}_n) > 0$$

$$\text{patterns } \boldsymbol{x}_n \text{ in class C2 must have: } \boldsymbol{w}^T \phi(\boldsymbol{x}_n) < 0$$

  - Using the class coding scheme $t_n \in \{+1, -1\}$
  - We would like:

$$\boldsymbol{w}^T \phi(\boldsymbol{x}_n) t_n > 0 \,, \forall \boldsymbol{x}_n$$

  - Perceptron cost criterion: **(i)** assign zero for each pattern that is correctly classified, **(ii)** for a misclassified pattern, minimize the quantity:

$$-\boldsymbol{w}^T \phi(\boldsymbol{x}_n) t_n$$

The perceptron algorithm (4/7)

- Error function
  - The error function is thus defined as:

$$E_p(\boldsymbol{w}) = -\sum_{n \in \mathcal{M}} \boldsymbol{w}^T \phi(\boldsymbol{x}_n) t_n$$

  - where $\mathcal{M}$ is the set of misclassified patterns
  - The contribution to the error function is: **(i)** zero for patterns that are correctly classified and **(ii)** linear functions of **w** in regions of space where patterns are misclassified (see error function above)
  - This means that: the error function is piecewise linear, summing over all patterns (either correctly classified or not) leads to the cost function

$$J_p(\boldsymbol{w}) = \sum_{n} \max(0, -\boldsymbol{w}^T \phi(\boldsymbol{x}_n) t_n)$$

On the choice of the cost function …

- As the total cost, one may have chosen
  - The following 0/1 loss function

$$J_{0/1}(\boldsymbol{w}) = \sum_n \ell(\mathrm{sign}(\boldsymbol{w}^T \phi(\boldsymbol{x}_n)), t_n)$$

$$\ell(x, y) = \begin{cases} 0 & x = y \quad \text{prediction is correct} \\ 1 & x \neq y \quad \text{prediction is wrong} \end{cases}$$

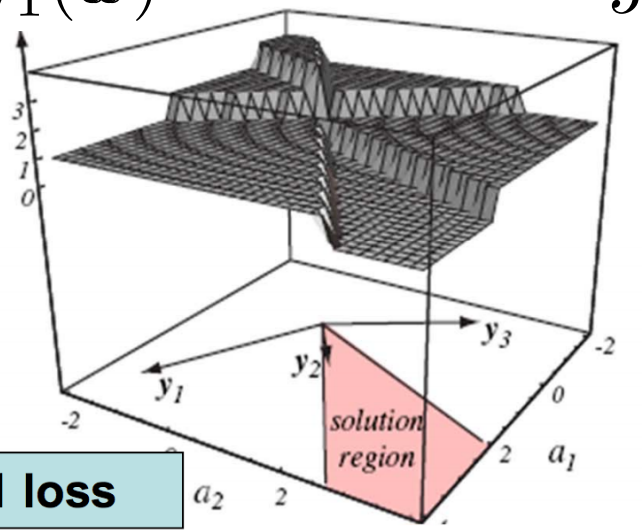
in place of

$$J_p(\boldsymbol{w}) = \sum_n \max(0, -\boldsymbol{w}^T \phi(\boldsymbol{x}_n) t_n)$$

On the choice of the cost function

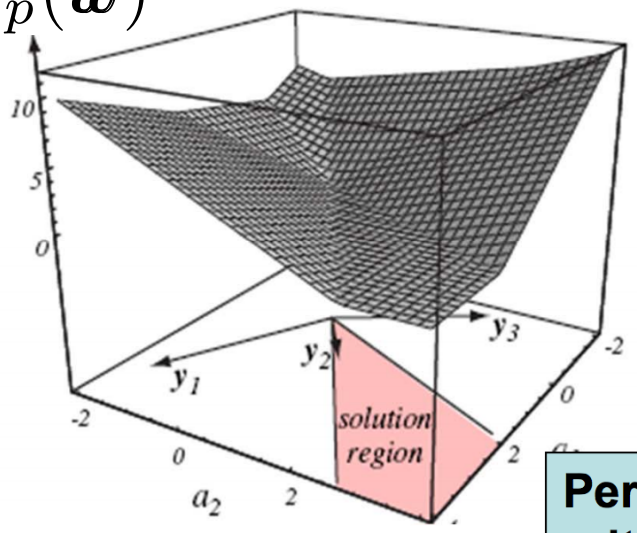
flat regions, there
the gradient is zero

$J_{0/1}(\boldsymbol{w})$



0/1 loss

$J_p(\boldsymbol{w})$



Perceptron criterion

Nice gradient

The perceptron algorithm (5/7)

- Error function
  - The total error function is then given by:

$$E_p(\boldsymbol{w}) = -\sum_{n \in \mathcal{M}} \boldsymbol{w}^T \phi(\boldsymbol{x}_n) t_n$$

  - We have that:

$$\nabla E_p(\boldsymbol{w}) = -\sum_{n \in \mathcal{M}} \phi(\boldsymbol{x}_n) t_n$$

  - For misclassified point n the gradient contribution depends on t_n

$$\begin{cases} -\phi(\boldsymbol{x}_n) & t_n > 0 \ (\mathrm{C1}) \\ \phi(\boldsymbol{x}_n) & t_n < 0 \ (\mathrm{C2}) \end{cases}$$

The perceptron algorithm (6/7)

- Weight update (gradient descent, direction is –gradient)
  - For misclassified point $\mathbf{x}_n$

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \eta\phi(\boldsymbol{x}_n) \text{ if: } t_n = +1$$
$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \eta\phi(\boldsymbol{x}_n) \text{ if: } t_n = -1$$

  - Without loss in generality, we can take the learning rate η=1
    - as the output function y(**x**) is unchanged if we multiply **w** by a constant
    - this descends form the shape of f(**a**)
- Perceptron theorem [Ros62]:
  - if there exists an exact solution
  - that is, if the data set is linearly separable
  - the above rules will find it in a finite number of steps

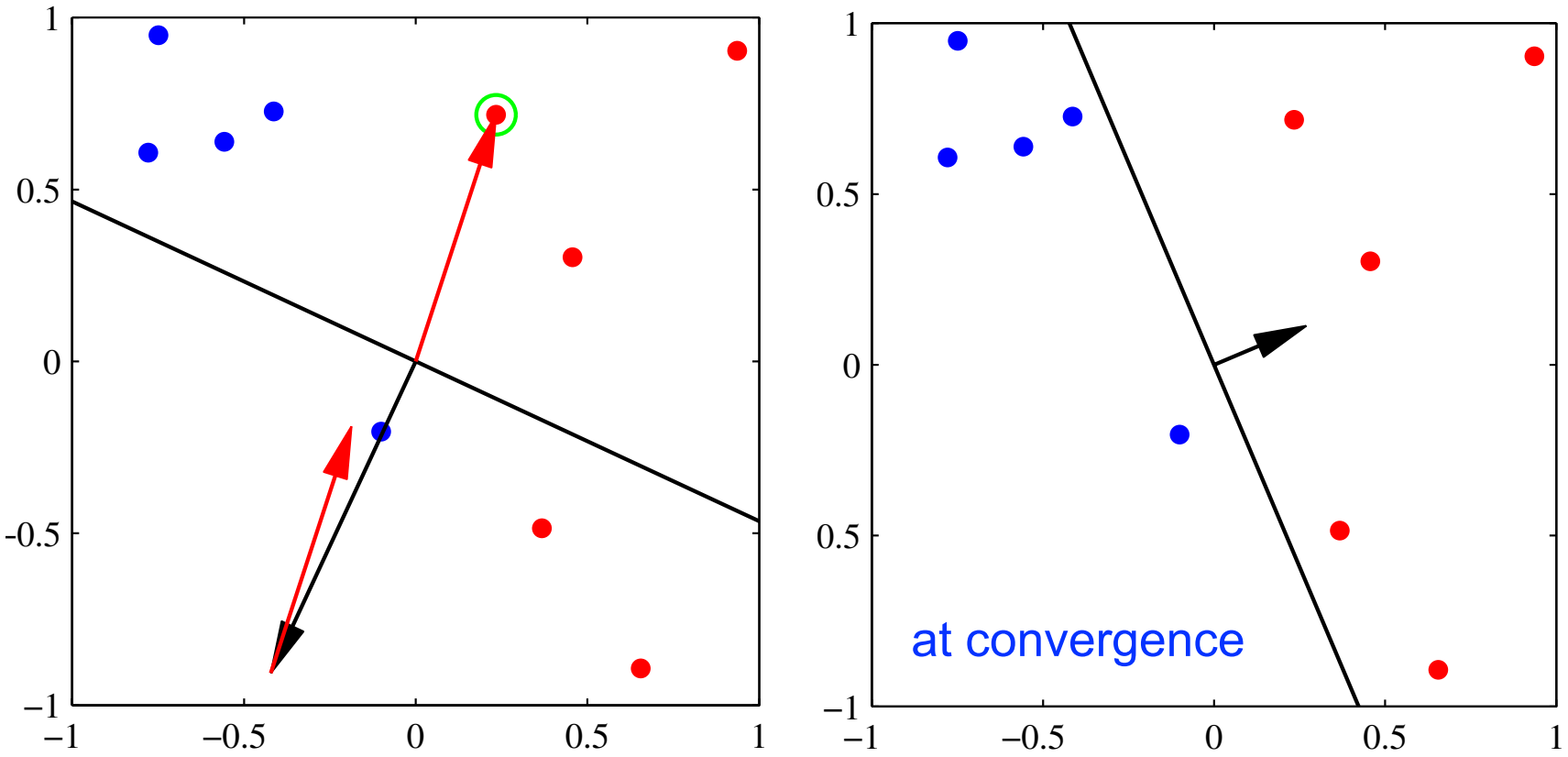
The perceptron algorithm (7/7)

- start: the weight vector $\mathbf{w}_0$ is generated at random, t=0
- test: exit if all points are correctly classified
  - pick vector **x** at random from data set
  - If **x** is in C1 and $\boldsymbol{w}^T\phi(\boldsymbol{x}) > 0$ go to test (correctly classified)
  - If **x** is in C1 and $\boldsymbol{w}^T\phi(\boldsymbol{x}) \leq 0$ go to add
  - If **x** is in C2 and $\boldsymbol{w}^T\phi(\boldsymbol{x}) < 0$ go to test (correctly classified)
  - If **x** is in C2 and $\boldsymbol{w}^T\phi(\boldsymbol{x}) \geq 0$ go to subtract
- add:

  $\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \phi(\boldsymbol{x})$ set t=t+1, go to test
- subtract:

  $\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \phi(\boldsymbol{x})$ set t=t+1, go to test

Illustration



- Boundary between two classes is linear
- Green point is misclassified → $\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \phi(\boldsymbol{x}_n)$

From perceptron to ANN

- Models for regression and classification take the form:

$$y(\boldsymbol{x}, \boldsymbol{w}) = f\left(\sum_{j=1}^{M} w_j \phi_j(\boldsymbol{x})\right) \quad (1)$$

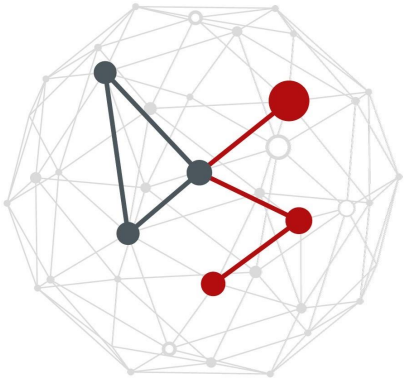
non-linear activation function

basis functions

- Function $f(\cdot)$ is:
  - non-linear: for classification problems (selects the class)
  - the identity function: for regression problems
- Goal
  - Make basis functions $\phi_j(\mathbf{x})$ depend on parameters
  - Adjust these parameters as part of the learning
  - ANN basis functions are of the form (1) so that each basis function itself is a non-linear function of a linear combination of the inputs

MIME
Master's degree ICT Internet Multimedia Engineering

THE ARTIFICIAL NEURON



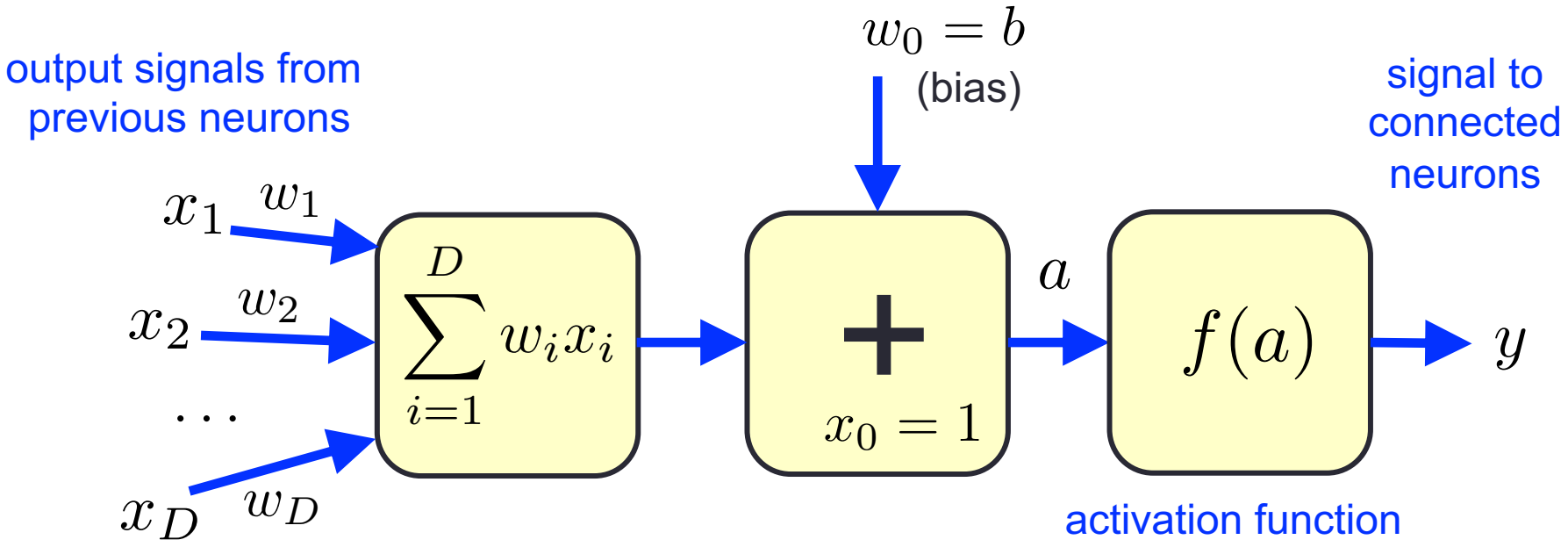
DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

DIPARTIMENTO MATEMATICA

800 1222·2022 ANNI

UNIVERSITÀ DEGLI STUDI DI PADOVA

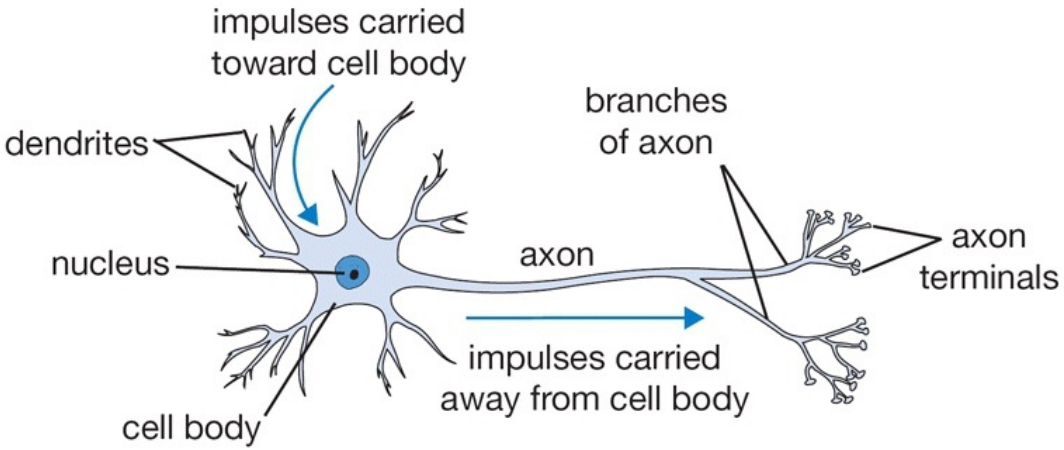
The artificial neuron



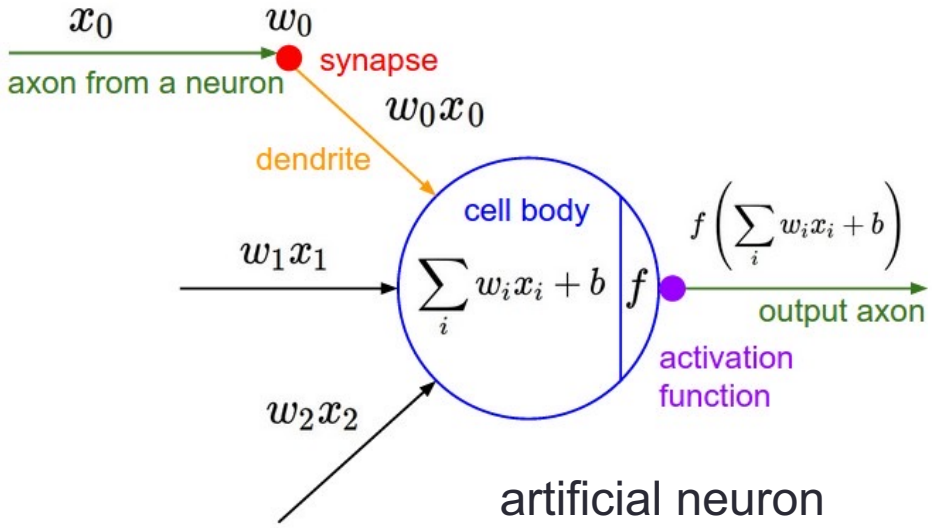
$$y = f(a) = f\left(\sum_{j=1}^{D} w_j x_j + b\right) = f\left(\sum_{j=0}^{D} w_j x_j\right)$$

The bias b is included in the weight vector, $b=w_0$, by extending the input vector, i.e., adding element $x_0=1$

Brain *vs* artificial neurons



biological neuron

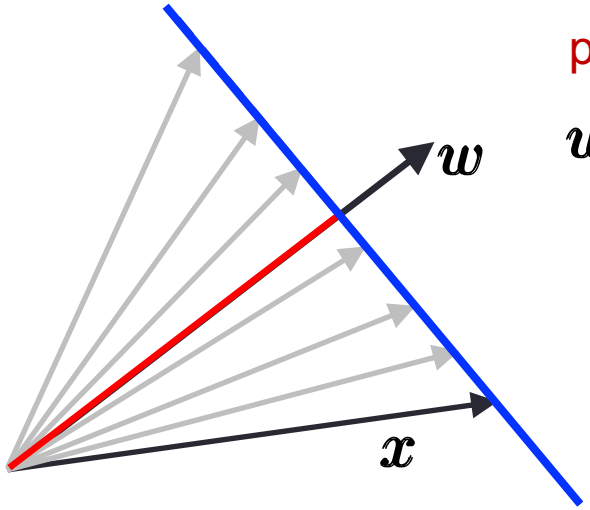


artificial neuron

Some hyperplane geometry

- Given a weight vector $\boldsymbol{w}$
- The locus of points $\boldsymbol{x}$ with a constant sum $\sum_{j=0}^{D} w_j x_j = \text{constant}$
  - Defines a hyperplane perpendicular to $\boldsymbol{w}$
- Euclidean norm (2-norm) measures vector length:

$$\|\boldsymbol{x}\| = \sqrt{\sum_{j=0}^{D} x_j^2}$$



projection onto $\boldsymbol{w}$

$$\boldsymbol{w}^T \boldsymbol{x} = \|\boldsymbol{w}\| \|\boldsymbol{x}\| \cos\theta$$

all input vectors with the same projection onto $\boldsymbol{w}$ produce the same output

the locus of points with equivalent projection defines a hyperplane perpendicular to $\boldsymbol{w}$

Hyperplane geometry: orientation

- The orientation of a hyperplane is determined by the direction of $\boldsymbol{w}$
- Let

$$\boldsymbol{e}_i = [0\,0\,0\,\ldots\,0\,1\,0\,0\,\ldots\,0]^T$$

(i-th element)

- be the unit vector aligned with the i-th coordinate axis (has single one in the i-th position and the remaining entries all zeros)

- The projection of **w** onto $\mathbf{e}_i$ returns the coordinate of **w** along the i-th axis. It is computed as:

$$\boldsymbol{w}_i = \boldsymbol{w}^T\boldsymbol{e}_i = \|\boldsymbol{w}\|\|\boldsymbol{e}_i\|\cos\theta_i = \\ = \|\boldsymbol{w}\|\cos\theta_i$$

$$\cos\theta_i = \frac{w_i}{\|\boldsymbol{w}\|} \quad (2)$$

- The orientation of the hyperplane is independent of the magnitude of **w** because the ratios in (2) remain constant if **w** is multiplied by a constant

Hyperplane geometry: bias

- As seen before, constant output surfaces $\boldsymbol{w}^T\boldsymbol{x} = \text{constant}$
  - are hyperplanes perperdicular to **w**
- Without the bias term, we have that: $\sum_{i=1}^{D} w_i x_i = 0$
- Defines a hyperplane through the origin
- The inclusion of a bias term $u = \boldsymbol{w}^T\boldsymbol{x} + b$
  - shifts the hyperplane to a distance d from the origin of: $d = -b/\|\boldsymbol{w}\|$
- Let **v** be the vector form the origin to the closest point on the plane
  - it must be normal to the plane (as it gets to the closest point)
  - and thus parallel to **w**
  - Hence, we can write: $\boldsymbol{v} = d\boldsymbol{w}/\|\boldsymbol{w}\|$
- The node hyperplane is the locus of points where $\boldsymbol{w}^T\boldsymbol{x} + b = 0$
  - **v** belongs to the hyperplane, so we have:

$$\boldsymbol{w}^T\boldsymbol{v} + b = d\boldsymbol{w}^T\boldsymbol{w}/\|\boldsymbol{w}\| + b = 0 \Rightarrow d = -b/\|\boldsymbol{w}\|$$

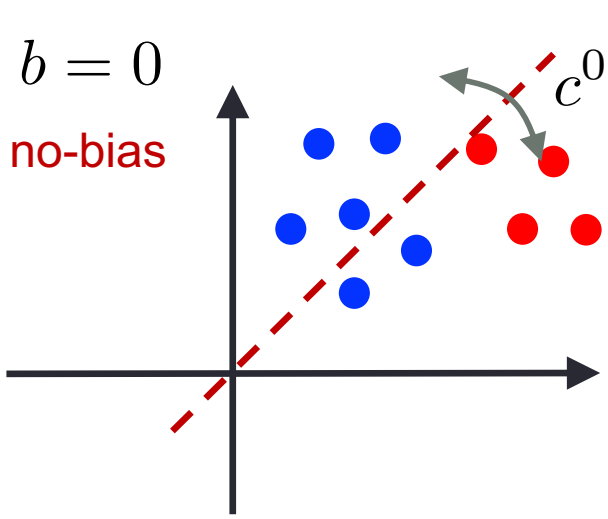
Hyperplane geometry: bias

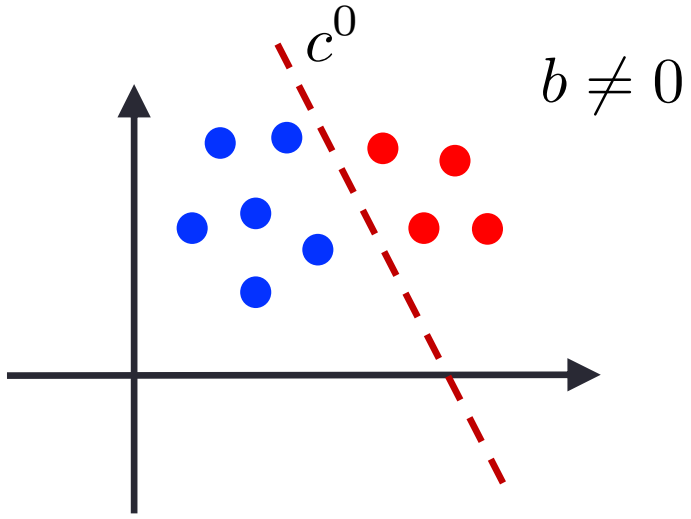
- Output of a neuron:

$$y(c) = f(c) = f\left(\sum_{i=1}^{D} w_i x_i + b\right) = f\left(\sum_{i=0}^{D} w_i x_i\right)$$

for all activation functions, the discriminating plane is that passing through zero:

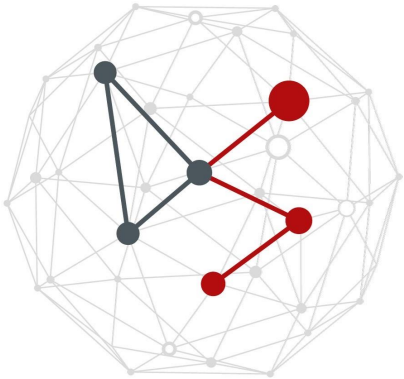
$$c^0 \triangleq \sum_i w_i x_i = 0$$





MIME
Master's degree ICT Internet Multimedia Engineering

ACTIVATION FUNCTIONS



DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE
DIPARTIMENTO MATEMATICA
800 1222·2022 ANNI
UNIVERSITÀ DEGLI STUDI DI PADOVA

Activation function

- The output y of a neuron is:

$$y = f(a) = f\left(\sum_{j=1}^{D} w_j x_j + b\right) = f\left(\sum_{j=0}^{D} w_j x_j\right)$$

- $f(\cdot)$ is an activation function
  - determining the firing intensity of the neuron
  - Several choices are possible:
    - Linear function
    - Logistic or sigmoid function
    - Hyperbolic tangent (tanh)
    - Rectified Linear Unit (ReLU)
    - Leaky ReLu

Linear activation function (1/3)

- The linear activation function
- It is a straight-line function
  - Output is proportional to input

$$y = \sum_{i=0}^{D} w_i x_i = \boldsymbol{w}^T \boldsymbol{x}$$

- Problem no. 1
  - Training is carried out using gradient descent
  - Backpropagating the error gradient from last to first layer
  - With this function, we have:

$$\partial y / \partial \boldsymbol{x} = \boldsymbol{w}$$

  - Which means that the gradient has no relationship wrt input **x**
  - **So** if an output error E is computed based on **x**
    - No change (learning) occurs based on **x**

Linear activation function (2/3)

- The linear activation function
- It is a straight-line function
  - Output is proportional to input

$$y = \sum_{i=0}^{D} w_i x_i = \boldsymbol{w}^T \boldsymbol{x}$$

- Problem no. 2
  - Think about a cascade of connected layers
  - Where each layer has linear activations
  - The activation of any layer goes into the next layer as input, and this next layer calculates weighted sum on its input and it in turn, fires based on another linear activation function
  - But, the concatenation of linear functions is still a linear function
  - Hence, stacking linear layers is equivalent to just having a single linear layer: adding layers in this way does not lead anywhere

Linear activation function (3/3)

- The linear activation function

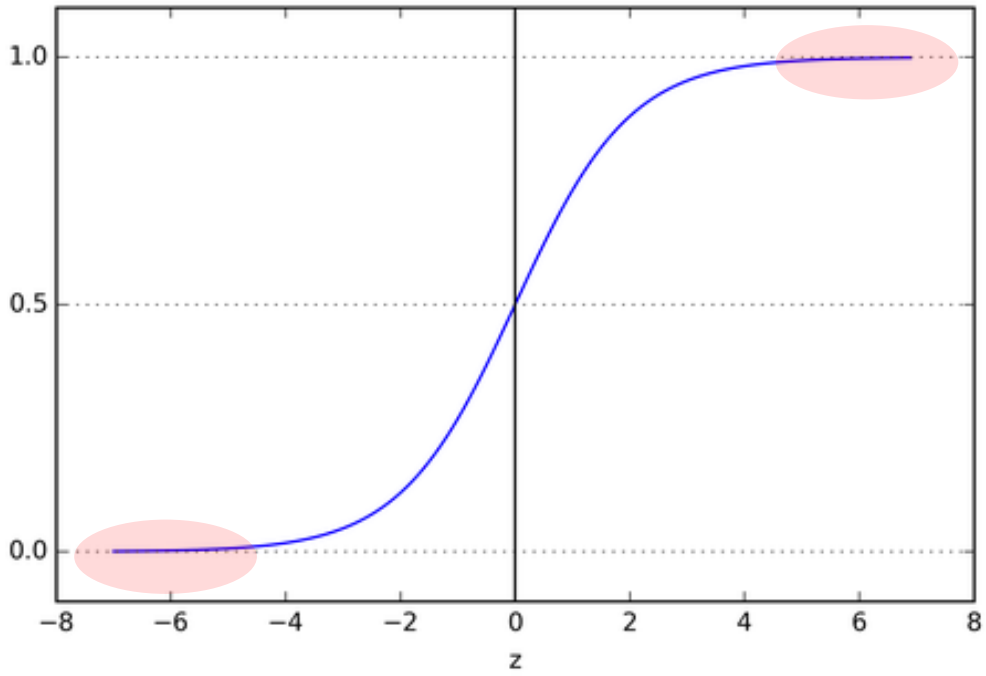
$$y = \sum_{i=0}^{D} w_i x_i = \boldsymbol{w}^T \boldsymbol{x}$$

- It is perfectly fine to use it
  - As the last layer of a neural network
  - To extract features for a final classification block (classification)
  - Or to interpolate the input (regression)

Sigmoid

- The sigmoid function $f(c) = \frac{1}{1+e^{-c}} \triangleq \sigma(c)$
  - Looks like an “S shape”
  - The output range is [0,1] (“squashes” real line onto [0,1])
  - For this reason, useful when predicting probabilities
- Properties
  - Differentiable
  - Monotonic
- Cons
  - Training can “get stuck”

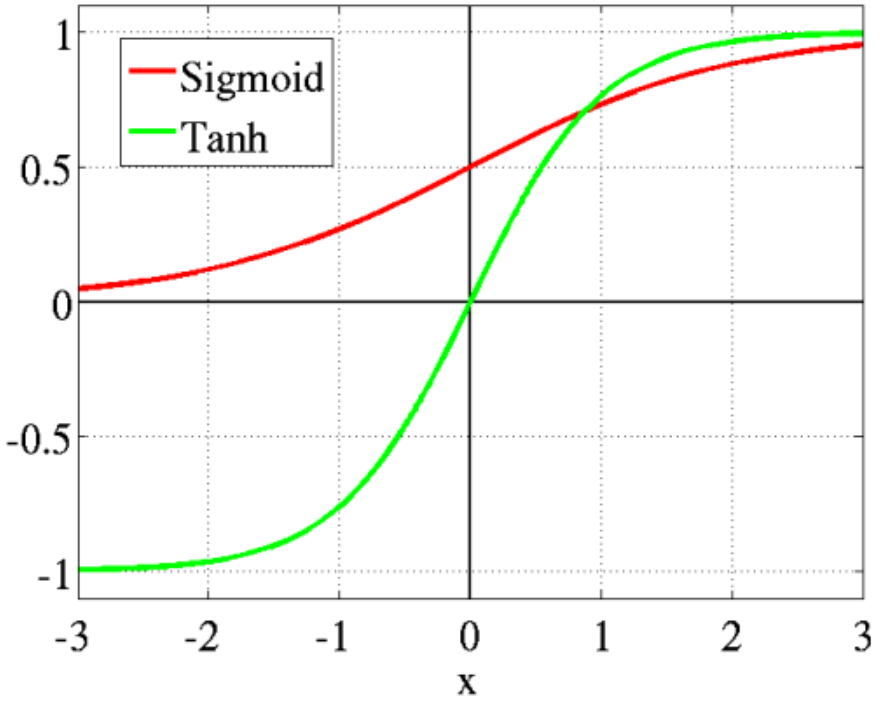
vanishing gradients



Hyperbolic tangent: tanh

- The hyperbolic tangent $\tanh(c) = \frac{\sinh(c)}{\cosh(c)} = \frac{e^c - e^{-c}}{e^c + e^{-c}}$
  - Looks like an “S shape”
  - The output range is [-1,1]
- Properties
  - Differentiable
  - Monotonic
  - Balanced dynamics
- Preferred use
  - 2-class classification

softmax preferred for multi-class problems



Problems of sigmoid (1/2)

- Problems of sigmoid function
  - Sigmoid has recently fallen out of favor
  - Two main problems

- Sigmoids saturate and kill gradients: a *very undesirable* property of the sigmoid neuron is that when the neuron's activation saturates at either tail of 0 or 1, *the gradient at these regions is almost zero*. As we shall see, during backpropagation, the local gradient will be multiplied by the gradient of this gate's output. Therefore, if the local gradient is very small, it will effectively "kill" the gradient and almost no signal will flow through the neuron to its weights and recursively to its data
- If using it: one must pay extra caution when initializing the weights of sigmoid neurons to prevent saturation. For example, if the initial weights are too large, most neurons would become saturated and the network *will barely learn*

Problems of sigmoid (1/2)

- Problems of sigmoid function
  - Sigmoid has recently fallen out of favor
  - Two main problems

- Sigmoids saturate and kill gradients: a *very undesirable* property of the sigmoid neuron is that when the neuron's activation saturates at either tail of 0 or 1, the gradient at these regions is almost zero. As we shall see, during backpropagation, the local gradient will be multiplied by the gradient of this gate's output. Therefore, if the local gradient is very small, it will effectively "kill" the gradient and almost no signal will flow through the neuron to its weights and recursively to its data
- When using it: one must pay extra caution when initializing the weights of sigmoid neurons to prevent saturation. For example, if the initial weights are too large, most neurons would become saturated and the network *will barely learn*

Problems of sigmoid (1/2)

- Problems of sigmoid function
  - Sigmoid has recently fallen out of favor
  - Two main problems

- Sigmoids saturate and kill gradients: a *very undesirable* property of the sigmoid neuron is that when the neuron's activation saturates at either tail of 0 or 1, the gradient at these regions is almost zero. As we shall see, during backpropagation, the local gradient will be multiplied by the gradient of this gate's output. Therefore, if the local gradient is very small, it will effectively "kill" the gradient and almost no signal will flow through the neuron to its weights and recursively to its data
- When using it: one must pay extra attention when initializing the weights of sigmoid neurons to prevent saturation. For example, if the initial weights are too large, most neurons would become saturated and the network *will barely learn*

Problems of sigmoid (2/2)

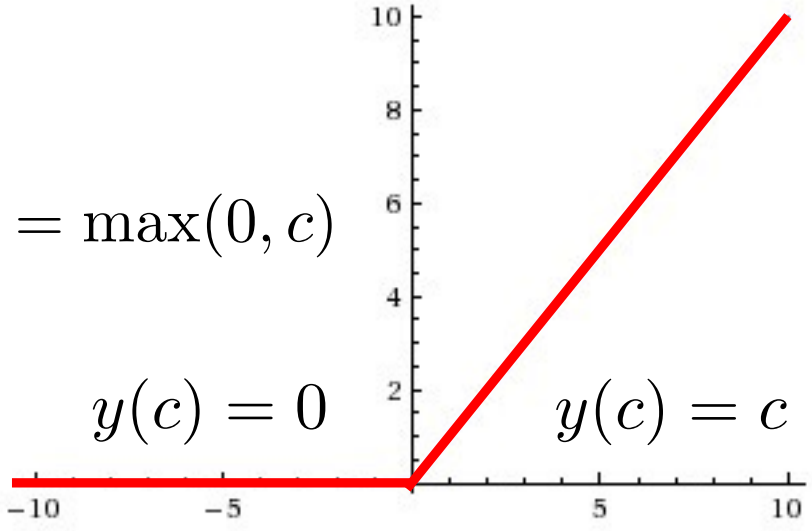
- Sigmoid outputs are non zero-centered: this is undesirable since neurons in later layers of processing in a neural network would be receiving data that is *non zero-centered*. This has implications on the dynamics during gradient descent, because if the data coming to a neuron is always positive (e.g., **x**>0 element-wise $y = \mathbf{w}^T x + b$), then the gradient on the weights **w** will, during learning (*backpropagation*) become either *all positive*, or *all negative*. This could introduce undesirable zig-zagging dynamics in the gradient updates for **w**
- While, this is an inconvenience, it has less severe consequences compared to the saturated activation problem, as it is often mitigated by updating over a large batch of data

- tanh is always preferable: its output is a scaled version of the sigmoid's, it holds:

$$\tanh(c) = 2\sigma(2c) - 1$$

ReLU (1/2)

- The rectified linear unit $\mathrm{ReLU}(c) = \max(0, c)$
  - Half-rectified
  - Equal to c for c≥0



- Imagenet [Krizhevsky12]
  - 5 conv. layers, 500k neurons, 6 million pars
  - Milestone paper for deep convolutional networks
  - Proves that ReLU leads to 6x improvement in learning time

[Krizhevsky12] A. Krizhevsky, I. Sutskever, G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," *Advances in Neural Information Processing Systems* (NIPS), 2012.

ReLU (2/2)

- Pros & cons
  - (++) it was found to greatly accelerate (e.g., a factor of 6 in [Krizhevsky12]) the convergence of stochastic gradient descent compared to the sigmoid and tanh functions
  - (+) compared to tanh and sigmoid neurons that involve expensive operations (exponentials, etc.), the ReLU can be implemented by simply thresholding a matrix of activations at zero (first reason for using it)
  - (-) Unfortunately, ReLU units can "die" during training. For example, a large gradient flowing through a ReLU neuron could cause the weights to update in such a way that the neuron will never activate on any data point again. If this happens, then the gradient flowing through the unit will forever be zero from that point on. That is, the ReLU units can irreversibly die during training. For example, you may find that *as much as 40% of your network can be "dead" if the learning rate is set too high*. A proper setting of the learning rate can solve this
- Leaky ReLU: attempt to solve this issue

ReLU (2/2)

- Pros & cons
  - (++) it was found to greatly accelerate (e.g., a factor of 6 in [Krizhevsky12]) the convergence of stochastic gradient descent compared to the sigmoid and tanh functions
  - (+) compared to tanh and sigmoid neurons that involve expensive operations (exponentials, etc.), the ReLU can be implemented by simply thresholding a matrix of activations at zero (first reason for using it)
  - (-) Unfortunately, ReLU units can "die" during training. For example, a large gradient flowing through a ReLU neuron could cause the weights to update in such a way that the neuron will never activate on any data point again. If this happens, then the gradient flowing through the unit will forever be zero from that point on. That is, the ReLU units can irreversibly die during training. For example, you may find that *as much as 40% of your network can be "dead" if the learning rate is set too high*. A proper setting of the learning rate can solve this
- Leaky ReLU: attempt to solve this issue

ReLU (2/2)

- Pros & cons

$$c = \sum_{j=1}^{D} w_j x_j + b$$

$$\text{if } b \ll 0 \rightarrow c < 0 \text{ always (w.h.p)}$$

  - (-) Unfortunately, ReLU units can "die" during training. For example, a large gradient flowing through a ReLU neuron could cause the weights to update in such a way that the neuron will never activate on any data point again. If this happens, then the gradient flowing through the unit will forever be zero from that point on. That is, the ReLU units can irreversibly die during training. For example, you may find that *as much as 40% of your network can be "dead" if the learning rate is set too high*. A proper setting of the learning rate can solve this

- Leaky ReLU: attempt to solve this issue

ReLU (2/2)

- Pros & cons

$$c = \sum_{j=1}^{D} w_j x_j + b$$

$$\text{if } b \ll 0 \rightarrow c < 0 \text{ always (w.h.p)}$$

  - (-) Unfortunately, ReLU units can "die" during training. For example, a large gradient flowing through a ReLU neuron could cause the weights to update in such a way that the neuron will never activate on any data point again. If this happens, then the gradient flowing through the unit will forever be zero from that point on. That is, the ReLU units can irreversibly die during training. For example, you may find that *as much as 40% of your network can be "dead" if the learning rate is set too high*. A proper setting of the learning rate can solve this

- Leaky ReLU: attempt to solve this issue

On the sparsity of activations

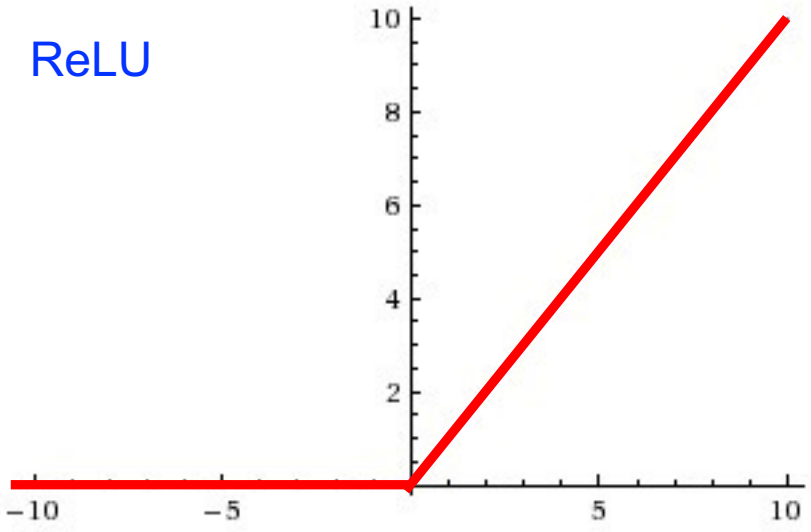
- With sigmoid or tanh
  - All neurons fire in an analog way (in [0,1] or [-1,1])
  - Hence, all activations will be processed
    - To obtain the network output as a response to an input
    - This means that activations are dense, i.e., costly
- With ReLU
  - A lot fewer neurons fire
  - As many of them will return 0
    - when the activation from the previous layer is <0
  - This means that activations are sparse, i.e., lighter computation

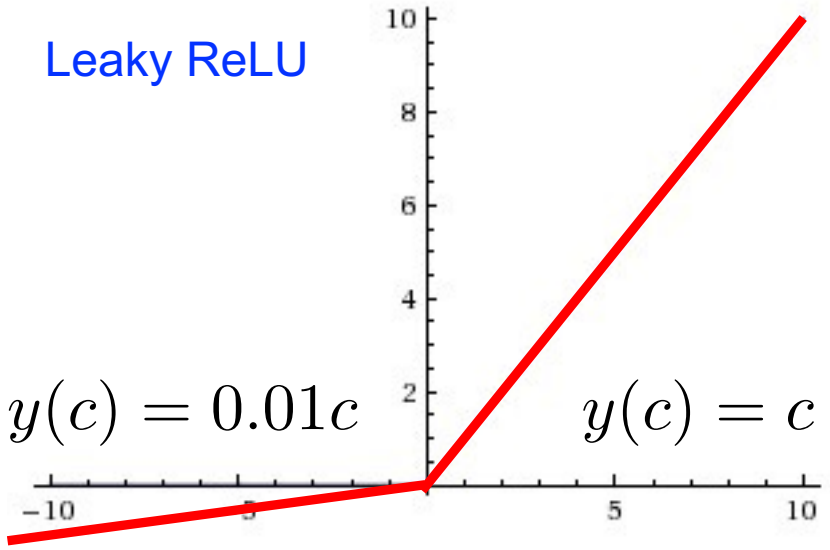
On the sparsity of activations

- With sigmoid or tanh
  - All neurons fire in an analog way (in [0,1] or [-1,1])
  - Hence, all activations will be processed
    - To obtain the network output as a response to an input
    - This means that activations are dense, i.e., costly
- With ReLU
  - A lot fewer neurons fire
  - As many of them will return 0
    - when the activation from the previous layer is <0
  - This means that activations are *sparse*, i.e., lighter computation

Leaky ReLU

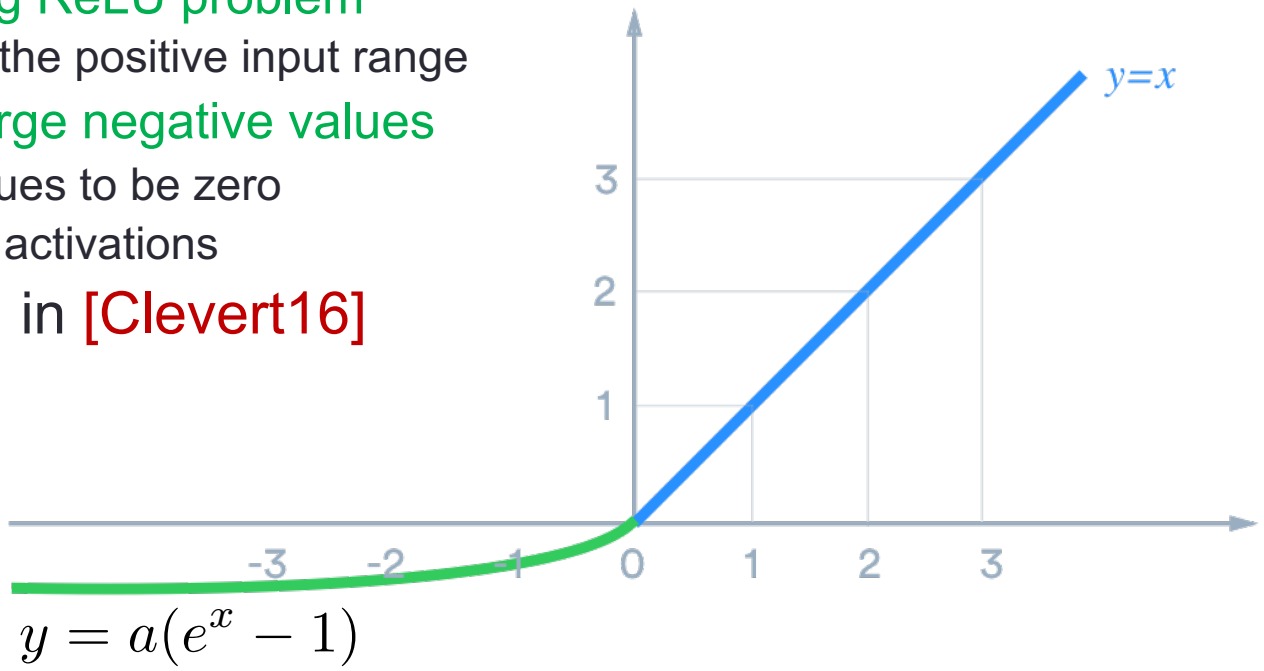
- ReLU: if c<0 → the output is zero
  - if the output of a ReLU is consistently 0 (e.g., it has a large negative b)
  - no error signal gets propagated to earlier layers - dying ReLU problem
- Leaky ReLU:
  - Mitigates this issue by adding a small negative slope for c<0
  - Solves the dying ReLU problem but we lose sparsity





Exponential Linear: ELU

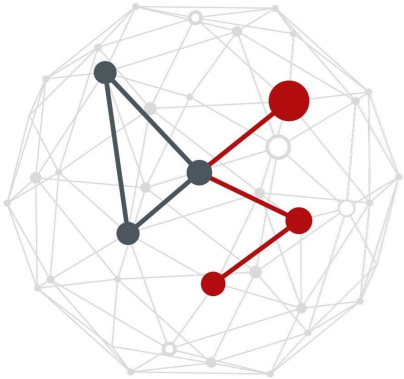
- Combines the good qualities of
  - ReLU and leaky ReLU
  - (+) Solves the dying ReLU problem
    - Same as ReLU in the positive input range
  - (+) Saturates for large negative values
    - Allowing these values to be zero
    - Leading to sparse activations
- Recently proposed in [Clevert16]



[Clevert16] D.-A.Clevert, T. Unterthiner, S. Hochreiter, “Fast and accurate deep network learning by Exponential Linear Units (ELUs),” International Conference on Learning Representations, San Juan, Puerto Rico, 2016.

MIME
Master's degree ICT Internet Multimedia Engineering

FFNN STRUCTURE



DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

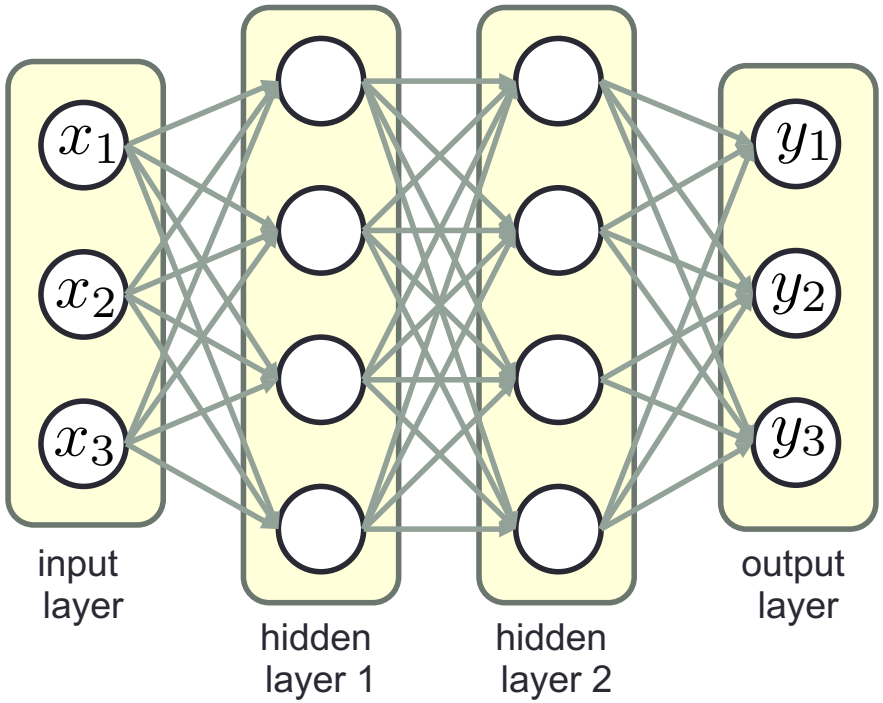
DIPARTIMENTO
MATEMATICA

800 ANNI
1222 · 2022

UNIVERSITÀ
DEGLI STUDI
DI PADOVA

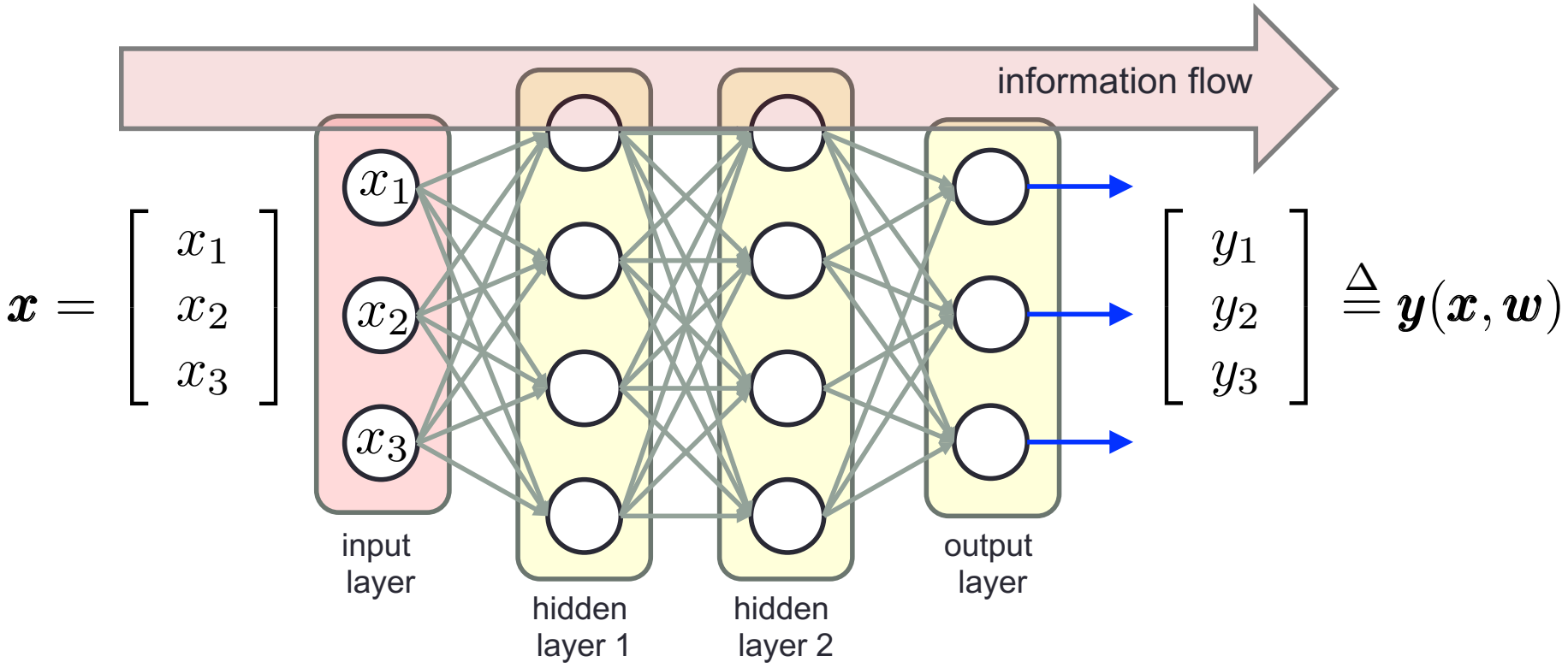
Feed Forward Neural Networks (FFNN)

- Implement functions, do not have internal states (memory cells)
- Artificial neurons organized in layers, signals flow from input to output
- Structure is fully connected (i.e., dense)



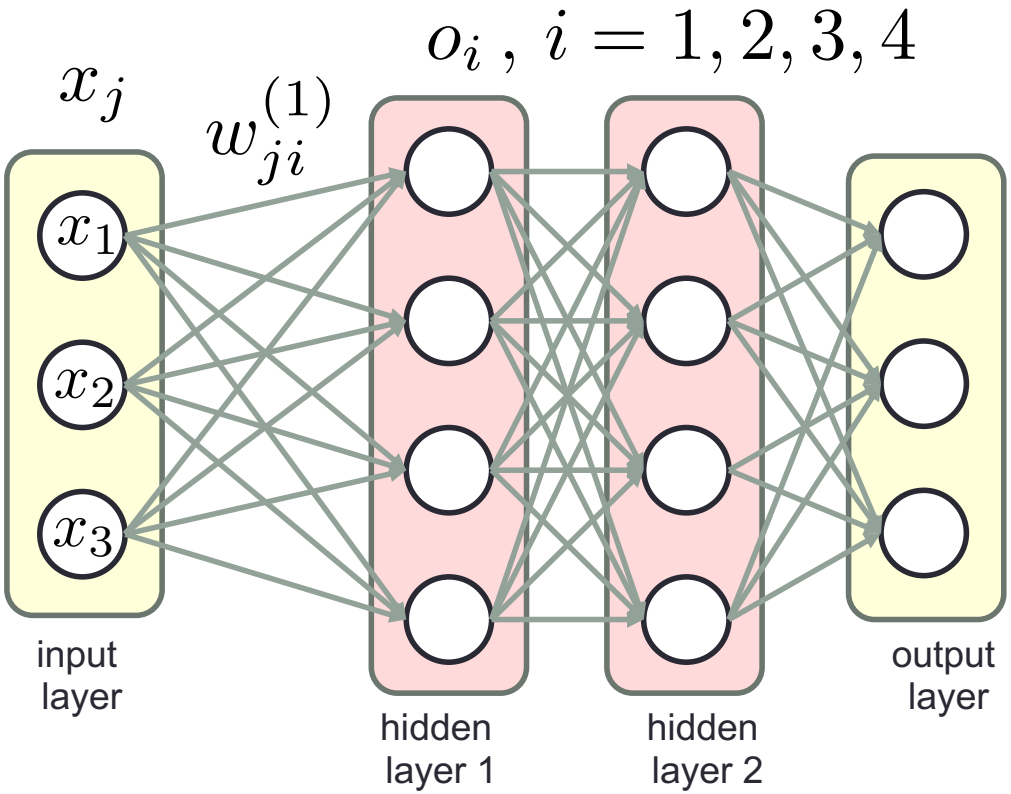
Feed Forward Neural Networks (FFNN)

- Input layer
  - Each node generates a data point
  - This data point is sent to all the neurons in the first hidden layer



Feed Forward Neural Networks (FFNN)

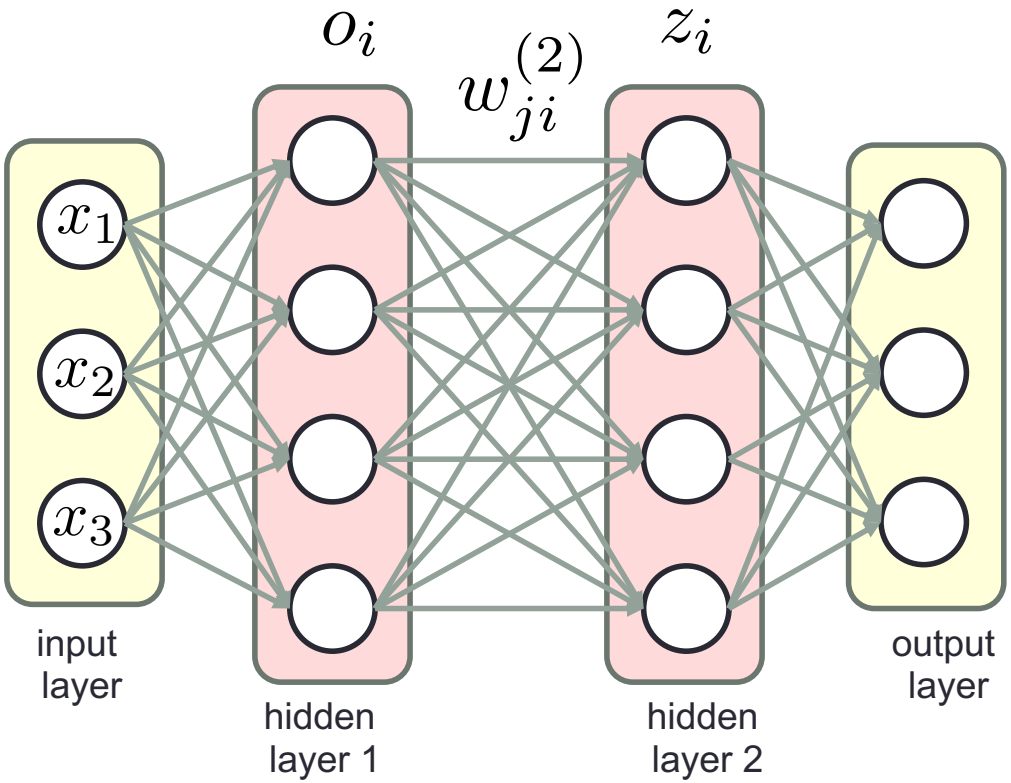
- Hidden layers
  - Each node is an artificial neuron with non-linear activation function
  - The same activation function is generally used at all nodes
  - Node combines signals from previous layer through a set of weights



$$o_i = f\left(\sum_j w_{ji}^{(1)} x_j\right)$$

Feed Forward Neural Networks (FFNN)

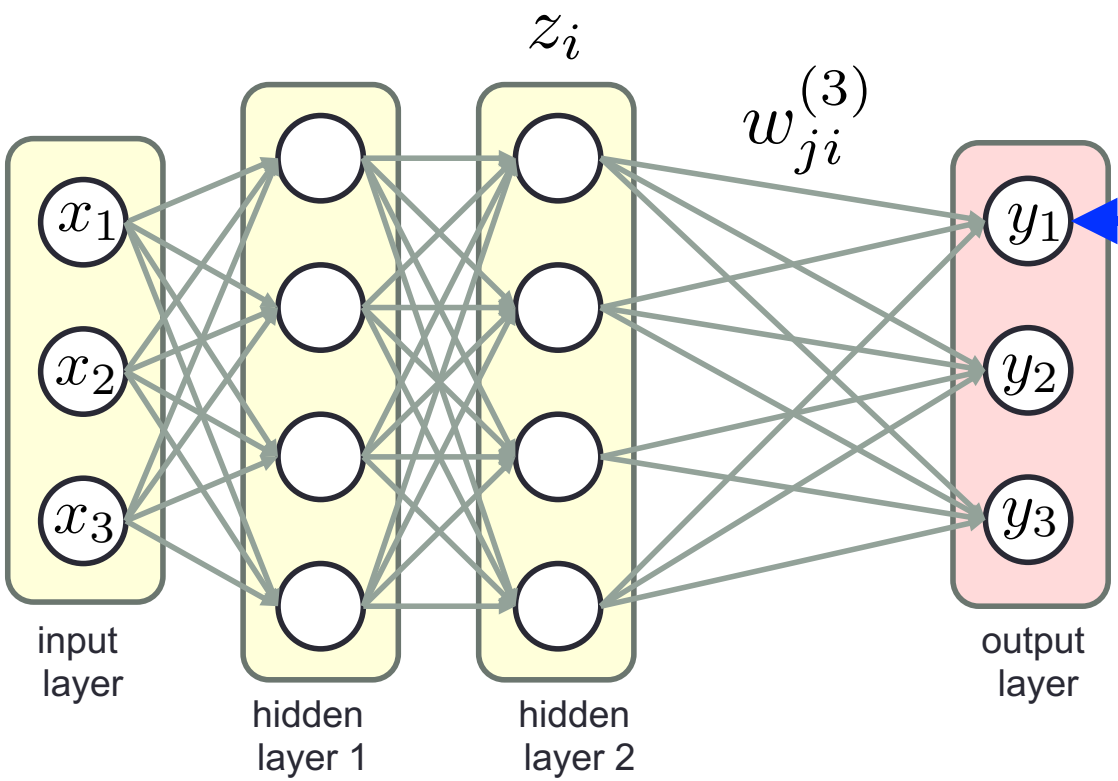
- Hidden layers
  - Each node is an artificial neuron with non-linear activation function
  - The same activation function is generally assumed at all nodes
  - Node combines signals from previous layer through a set of weights



$$z_i = f\left(\sum_j w_{ji}^{(2)} o_j\right)$$

Feed Forward Neural Networks (FFNN)

- Output layer
  - The output layer adopts a special function, which depends on the network use: (i) regression or (ii) classification



$$a_i \triangleq \sum_j w_{ji}^{(3)} z_j$$

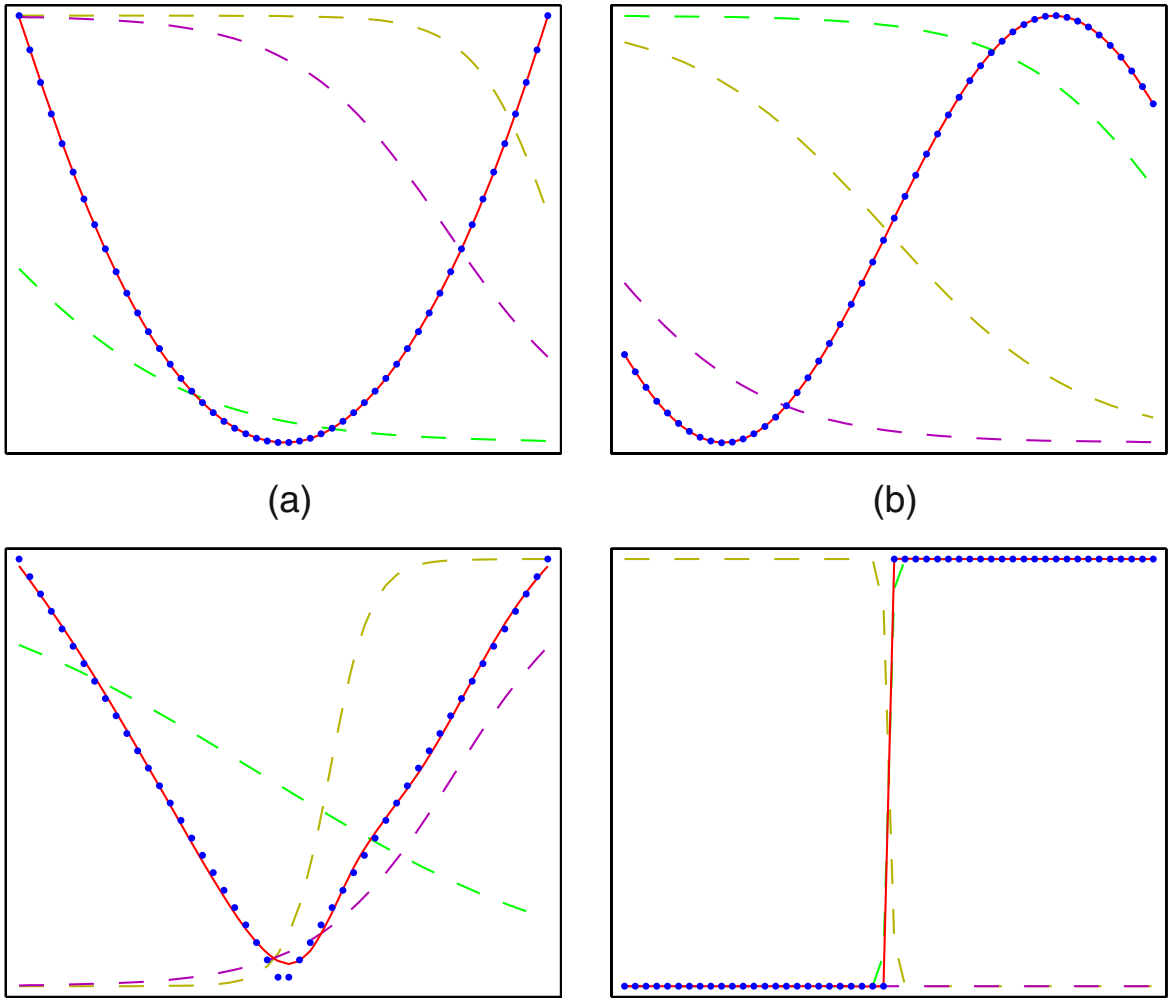
(i) for regression

$$y_i = a_i$$

(ii) for classification
"softmax" (see later)

Universal approximation capability [Cybenko89]

Illustration of the capability of a multilayer perceptron to approximate four different functions comprising (a) $f(x) = x^2$, (b) $f(x) = \sin(x)$, (c), $f(x) = |x|$, and (d) $f(x) = H(x)$ where $H(x)$ is the Heaviside step function. In each case, $N = 50$ data points, shown as blue dots, have been sampled uniformly in x over the interval $(-1, 1)$ and the corresponding values of $f(x)$ evaluated. These data points are then used to train a two-layer network having 3 hidden units with '$\tanh$' activation functions and linear output units. The resulting network functions are shown by the red curves, and the outputs of the three hidden units are shown by the three dashed curves.



Universal approximation capability [Cybenko89]

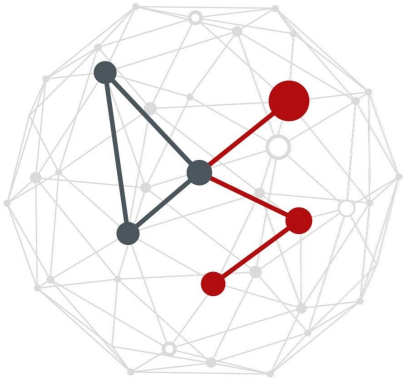
- Universal function approximators
  - To any degree of accuracy
  - Theoretically, a single hidden layer suffices
  - Provided a reasonable choice of non-linearity, e.g., sigmoid, tanh, etc.
- While this is appealing
  - A single hidden layer is almost never used
    - number of neurons in the hidden layer
    - training time and complexity
- In practice, we use multiple layers with finite number of neurons, that:
  - we are good at training with known tools (gradient descent)

MIME
Master's degree ICT Internet Multimedia Engineering

ERROR AND OUTPUT FUNCTIONS

Taken from the Bishop's book:

"Pattern Recognition and Machine Learning"



DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

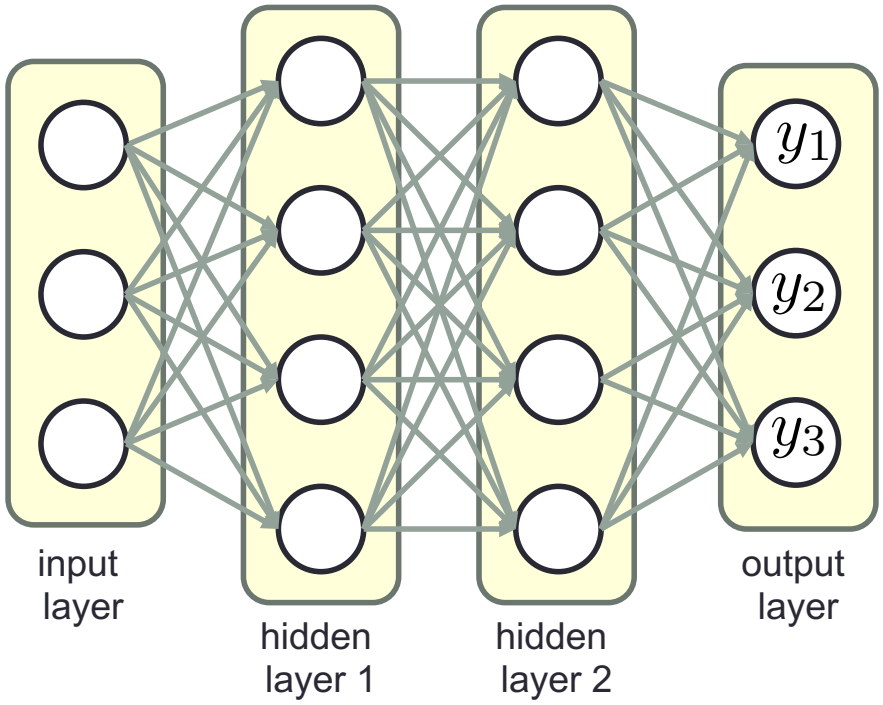
DIPARTIMENTO MATEMATICA

800 1222·2022 ANNI

UNIVERSITÀ DEGLI STUDI DI PADOVA

Notation

Input vector
at time n $\boldsymbol{x}_n$



Output vector
K output neurons

$$\boldsymbol{y}(\boldsymbol{x}_n, \boldsymbol{w}) = \begin{bmatrix} y_1(\boldsymbol{x}_n, \boldsymbol{w}) \\ y_2(\boldsymbol{x}_n, \boldsymbol{w}) \\ \vdots \\ y_K(\boldsymbol{x}_n, \boldsymbol{w}) \end{bmatrix}$$

Error & output functions: regression

- Regression problem, given a set of:
  - N input patterns
  - corresponding N target vectors $\{(\boldsymbol{x}_n, \boldsymbol{t}_n)\}\Big|_{n=1}^{N}$
- An error function is defined as:

$$E(\boldsymbol{w}) = \frac{1}{2}\sum_{n=1}^{N} \|\boldsymbol{y}(\boldsymbol{x}_n, \boldsymbol{w}) - \boldsymbol{t}_n\|^2 = \sum_{n=1}^{N} E_n(\boldsymbol{w})$$

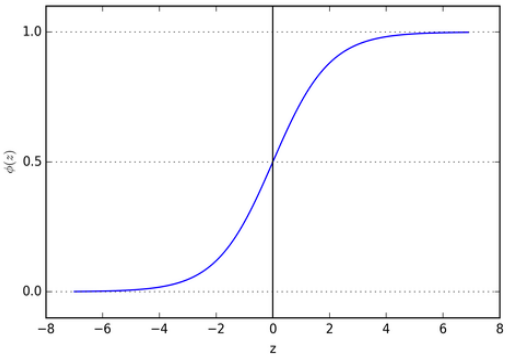
- Output activation function: is the identity, i.e.,
  - For each output neuron, we have

$$y = f(c) = c$$

Error & output functions: classification

- Binary (2-class) classification
  - t=1 denotes class C1
  - t=0 denotes class C2

[0,1]



- We consider an FFNN having a single output node
  - whose activation function is a logistic sigmoid

$$y = \sigma(c) = \frac{1}{1 + \exp(-c)}$$

  - so that: $y(\boldsymbol{x}, \boldsymbol{w}) \in [0, 1]$

- we can interpret:

$$\begin{cases} p(\mathrm{C}_1|\boldsymbol{x}) = y(\boldsymbol{x}, \boldsymbol{w}) \\ p(\mathrm{C}_2|\boldsymbol{x}) = 1 - y(\boldsymbol{x}, \boldsymbol{w}) \end{cases}$$

Error & output functions: classification

- With this:
$$\begin{cases} p(\mathrm{C}_1|\boldsymbol{x}) = y(\boldsymbol{x}, \boldsymbol{w}) \\ p(\mathrm{C}_2|\boldsymbol{x}) = 1 - y(\boldsymbol{x}, \boldsymbol{w}) \end{cases}$$
- We can write: $p(t|\boldsymbol{x}, \boldsymbol{w}) = y(\boldsymbol{x}, \boldsymbol{w})^t (1 - y(\boldsymbol{x}, \boldsymbol{w}))^{1-t}$ (3)
- If we consider a training set of N i.i.d. observations $\{(\boldsymbol{x}_n, t_n)\}$
  - The error function is given by the negative log likelihood of (3)
  - Negative → it is appealing to minimize an error function

$$E(\boldsymbol{w}) = -\ln p(t|\boldsymbol{x}, \boldsymbol{w}) = -\ln \left[\prod_{n=1}^{N} y(\boldsymbol{x}_n, \boldsymbol{w})^{t_n} (1 - y(\boldsymbol{x}_n, \boldsymbol{w}))^{1-t_n} \right]$$

$$= -\sum_{n=1}^{N} [t_n \ln y(\boldsymbol{x}_n, \boldsymbol{w}) + (1 - t_n) \ln(1 - y(\boldsymbol{x}_n, \boldsymbol{w}))] = \sum_{n=1}^{N} E_n(\boldsymbol{w})$$

cross entropy error function

Error & output functions: classification

- Multiclass classification (K classes)
  - each input vector is assigned to one of K mutually exclusive classes
  - every input vector $\boldsymbol{x}$ has an associated label $\boldsymbol{t}$ (1-of-K coding)

$$\boldsymbol{t} = [t_1, t_2, \ldots, t_K]^T \text{ with: } t_i \in \{0, 1\}, \sum_i t_i = 1$$

  - We interpret the network output from neuron k as

$$y_k(\boldsymbol{x}, \boldsymbol{w}) = p(t_k = 1 | \boldsymbol{x})$$

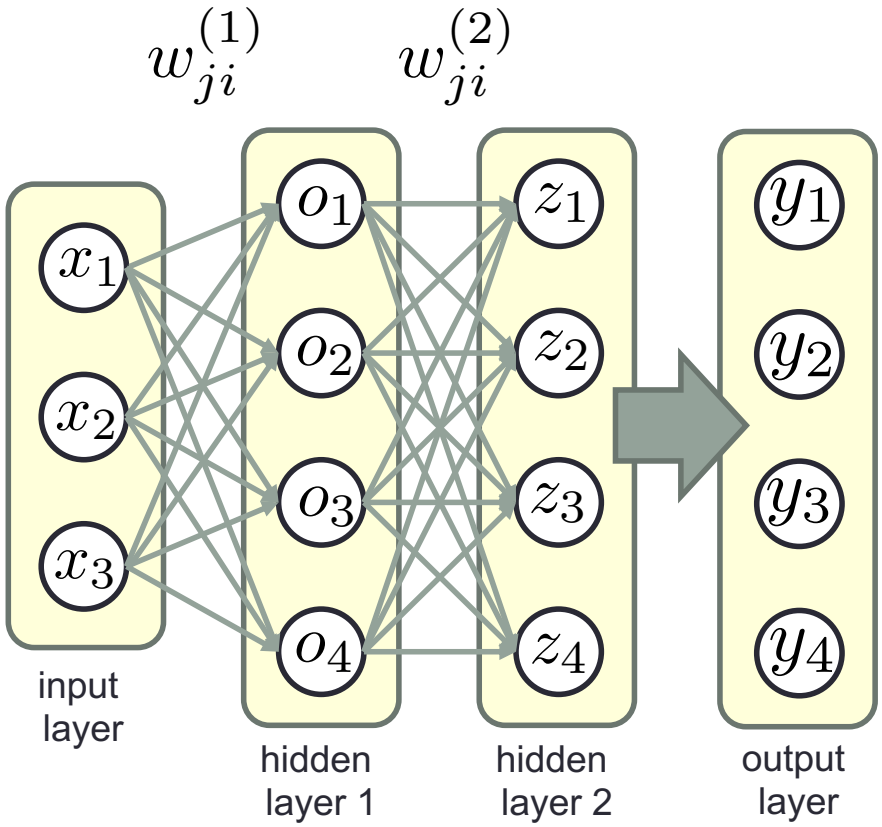
- Given N i.i.d. observations, the error function is:

$$E(\boldsymbol{w}) = -\sum_{n=1}^{N} \sum_{k=1}^{K} t_{kn} \ln(y_k(\boldsymbol{x}_n, \boldsymbol{w})) = \sum_{n=1}^{N} E_n(\boldsymbol{w})$$

multiclass cross entropy error function

Error & output functions: classification

- Softmax function



$$o_i = f\left(\sum_j w_{ji}^{(1)} x_j\right)$$

$$z_i = \sum_j w_{ji}^{(2)} o_j$$

$$y_k = \frac{\exp(z_k)}{\sum_j \exp(z_j)}$$

Error & output functions: classification

- Softmax function

$$y_k(\boldsymbol{x}, \boldsymbol{w}) = \frac{\exp(z_k(\boldsymbol{x}, \boldsymbol{w}))}{\sum_j \exp(z_j(\boldsymbol{x}, \boldsymbol{w}))}$$

- Note that:

$$\begin{cases} y_k(\boldsymbol{x}, \boldsymbol{w}) \in [0, 1] \\ \sum_k y_k(\boldsymbol{x}, \boldsymbol{w}) = 1 \end{cases}$$

- Output values represent
  - the probability that the input pattern belongs to class k

Summary

- Output layer
  - Identity for regression
  - Softmax for classification

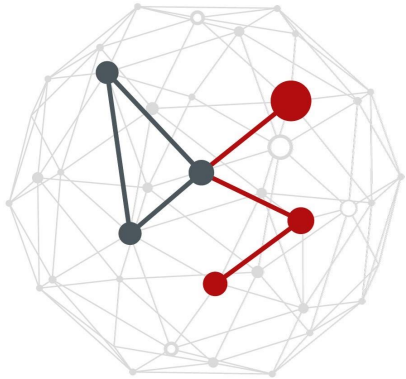
- IMPORTANT: error function
  - Can be always decomposed into the sum of N error functions over the data points (i.e., sum of pointwise errors)

MIME
Master's degree ICT Internet Multimedia Engineering

FFNN TRAINING: "BACKPROP"

Taken from Chapter 5 of [Reed99]

it is wonderful book to get the basics



DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

DIPARTIMENTO MATEMATICA

1222 · 2022 800 ANNI

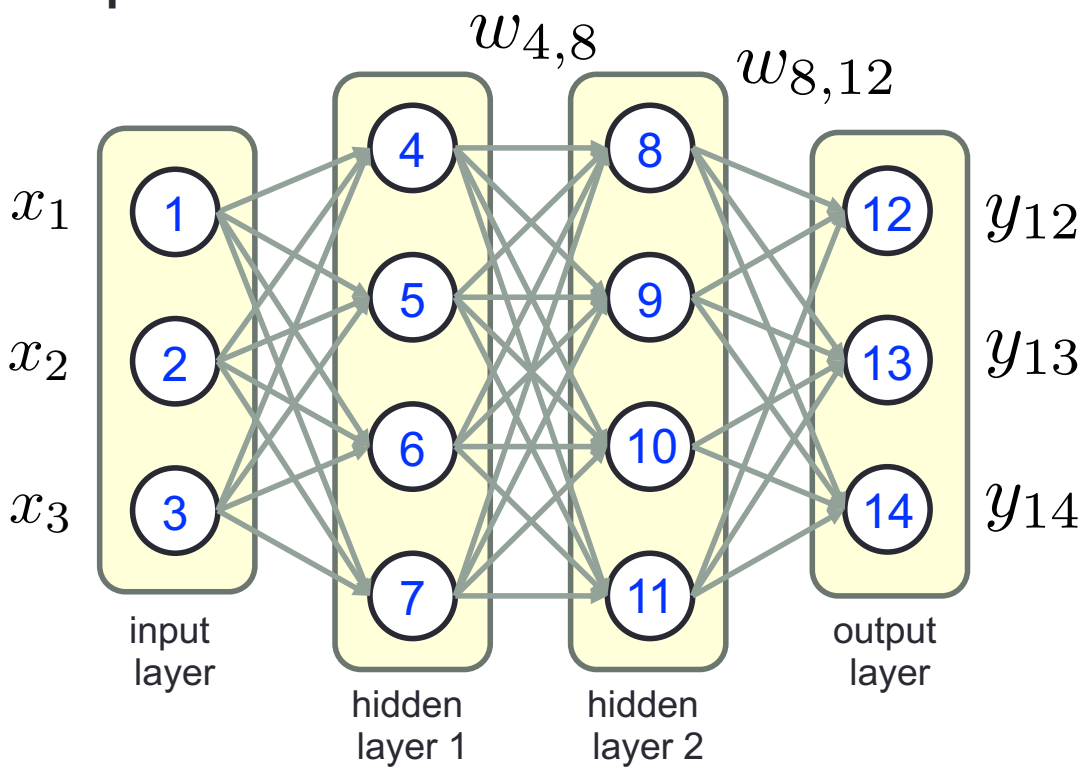
UNIVERSITÀ DEGLI STUDI DI PADOVA

Back-propagation

- Due to Seppo Linnainmaa
  - Born: 28 September 1945
  - In 1974: first PhD ever awarded in CS from Univ. of Helsinki
  - Research professor at Technical Research Centre of Finland
  - Introduced the reverse mode of automatic differentiation, to efficiently compute the derivative of a differentiable composite function that can be represented as a graph, by recursively applying the chain rule to the building blocks of the function
  - Retired in 2007

FFNN: indexing nodes and weights w_{ij}

- In a FFNN it is convenient to index nodes in a way that
  - If j>i this means that node j follows i in terms of dependencies
  - If a link does not exist between node i and node j, the weight is zero
  - **Example:** M=14 nodes



Note that:

$$w_{ij} = 0 \text{ if } i \geq j$$

In this topology:

$$w_{1,8} = 0$$

$$w_{4,13} = 0$$

$$\ldots$$

Setup

- We must have a training set - supervised learning

$$\{(\boldsymbol{x}_1, \boldsymbol{t}_1), (\boldsymbol{x}_2, \boldsymbol{t}_2), \ldots, (\boldsymbol{x}_N, \boldsymbol{t}_N)\}$$

where:
$$\begin{cases} \boldsymbol{x}_n & \text{data vector at time } n \\ \boldsymbol{t}_n & \text{target output vector at time } n \end{cases}$$

- Input and output vectors
  (I and O are number of input & output neurons):

$$\boldsymbol{x}_n = [x_{n1}, x_{n2}, \ldots, x_{nI}]^T$$

$$\boldsymbol{y}_n = [y_{n1}, y_{n2}, \ldots, y_{nO}]^T$$

Step 1 – perform forward pass

- Activation at unit (node) j

$$a_j = \sum_{i<j} w_{ij} y_i$$

sum output coming from all previous nodes (*smaller index*) in the FNN topology, for which a link exists (ensured by the weights)

$$y_j = f(a_j)$$

- Do computation for all nodes from smaller index j=1 to largest j=M, where M is the number of neurons
- Activation function can differ for each node. Although: usually one function for hidden and one for output nodes
- Regression: for output layer use $y_j = a_j$

Step 2 – error gradient (1/2)

- Error function is (computed at output layer):

$$E(\boldsymbol{w}) = \frac{1}{2}\sum_{n=1}^{N} \|\boldsymbol{y}_n - \boldsymbol{t}_n\|^2 = \frac{1}{2}\sum_{n=1}^{N}\sum_{j=1}^{O}(y_{nj} - t_{nj})^2 = \sum_{n=1}^{N} E_n(\boldsymbol{w})$$

- Calculate the derivative of the error wrt the weights
  - Given the shape of the error function (i.e., a sum)
  - the total derivative can be expressed as:

$$\frac{\partial E(\boldsymbol{w})}{\partial w_{ij}} = \sum_{n} \frac{\partial E_n(\boldsymbol{w})}{\partial w_{ij}}$$

point-wise error derivative

where:

$$E_n(\boldsymbol{w}) = \frac{1}{2}\sum_{j=1}^{O}(y_{nj} - t_{nj})^2$$

Step 2 – error gradient (2/2)

- A trick leading to an efficient computation (*chain rule*):

$$\frac{\partial E_n(\boldsymbol{w})}{\partial w_{ij}} = \sum_k \frac{\partial E_n(\boldsymbol{w})}{\partial a_k} \frac{\partial a_k}{\partial w_{ij}}$$

- where index k runs over all nodes $k = 1, 2, \ldots, M$ and

$$a_k = \sum_{i<k} w_{ik} y_i$$

- sum runs over all nodes with smaller index (preceding node k), as by construction these are possibly connected to node k
- we sum over all i<k, although some nodes are not connected to node k. These are automatically ruled out, as $w_{ik} = 0$

Error gradient: output nodes (1/3)

- It is convenient to compute a value δ_i for all output nodes i

$$\delta_i = \frac{\partial E_n(\boldsymbol{w})}{\partial a_i} = \frac{\partial E_n(\boldsymbol{w})}{\partial y_i}\frac{\partial y_i}{\partial a_i}$$

- First term (y and t are at time n, we omit the index)

$$E_n(\boldsymbol{w}) = \frac{1}{2}\sum_{j=1}^{O}(y_j - t_j)^2 \quad \Rightarrow \quad \frac{\partial E_n(\boldsymbol{w})}{\partial y_i} = y_i - t_i$$

this result descends from the considered error function (square norm), different expressions hold for other error functions

Error gradient: output nodes (2/3)

- It is convenient to compute a value δ_i for all output nodes i

$$\delta_i = \frac{\partial E_n(\boldsymbol{w})}{\partial a_i} = \frac{\partial E_n(\boldsymbol{w})}{\partial y_i}\frac{\partial y_i}{\partial a_i}$$

- For output nodes we have (remember: regression)

$$y_i = f(a_i) = a_i \quad \Rightarrow \quad \frac{\partial y_i}{\partial a_i} = 1$$

this result descends from the considered output layer activation function (linear activation function). It will differ for other choices

Error gradient: output nodes (3/3)

- It is convenient to compute a value δ_i for all output nodes i

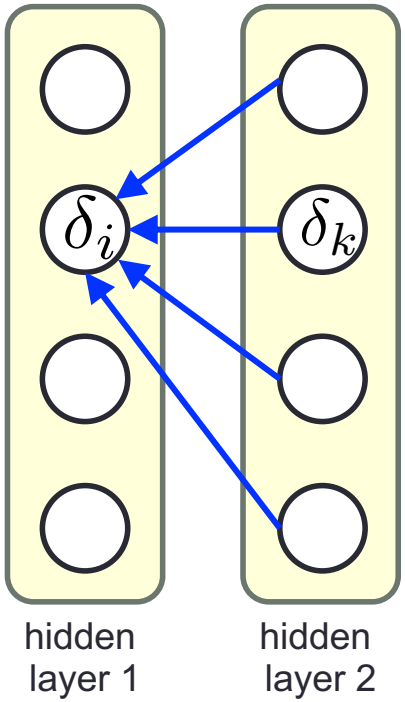
$$\delta_i = \frac{\partial E_n(\boldsymbol{w})}{\partial a_i} = \frac{\partial E_n(\boldsymbol{w})}{\partial y_i}\frac{\partial y_i}{\partial a_i}$$

- Hence, for output nodes

$$\delta_i = y_i - t_i$$

Error gradient: hidden nodes (1/4)

- For hidden nodes, δ_i is obtained indirectly: $\delta_i = \dfrac{\partial E_n(\boldsymbol{w})}{\partial a_i}$
- Hidden nodes can influence the error only through the effect on the nodes k to which they send their output signal:



We compute the deltas moving backward

- From last to first layer
- δ_i only depends on the connected nodes (k>i)
- The δ_k of those nodes is recursively propagated

Error gradient: hidden nodes (2/4)

- For hidden nodes, δ_i is obtained indirectly
- We can conveniently rewrite it as:

$$\delta_i = \frac{\partial E_n(\boldsymbol{w})}{\partial a_i} = \sum_k \frac{\partial E_n(\boldsymbol{w})}{\partial a_k} \frac{\partial a_k}{\partial a_i} = \sum_k \delta_k \frac{\partial a_k}{\partial a_i}$$

$$\text{this is } \delta_k \text{ of node } k$$

- The second factor $\partial a_k / \partial a_i$
  - it is zero if node i does not connect to node k
  - so the sum only extends to connected nodes

Error gradient: hidden nodes (3/4)

- Factor $\partial a_k / \partial a_i$
  - it is zero if node i does not connect to node k
  - Otherwise, it holds:

$$a_k = \sum_{i'<k} w_{i'k} y_{i'} = \sum_{i'<k} w_{i'k} f(a_{i'})$$

$$\Rightarrow \quad \frac{\partial a_k}{\partial a_i} = w_{ik} \frac{\partial f(a_i)}{\partial a_i} = w_{ik} f'(a_i)$$

- From the previous expression:

$$\delta_i = \sum_k \delta_k \frac{\partial a_k}{\partial a_i} = f'(a_i) \sum_{k>i} \delta_k w_{ik}$$

from previous slide

Error gradient: hidden nodes (4/4)

- For hidden nodes we have:

$$\delta_i = f'(a_i) \sum_{k>i} \delta_k w_{ik}$$

- It is a recursion, note that, as k>i
- it is calculated from node k, proceeding backwards towards i
- Hence, the name "back-propagation"

- First, δ values are calculated for the output nodes. Then, values are calculated for nodes that send connections to the output, then values are calculated for nodes two hops away from the output layer, and so forth…

Wrapping up

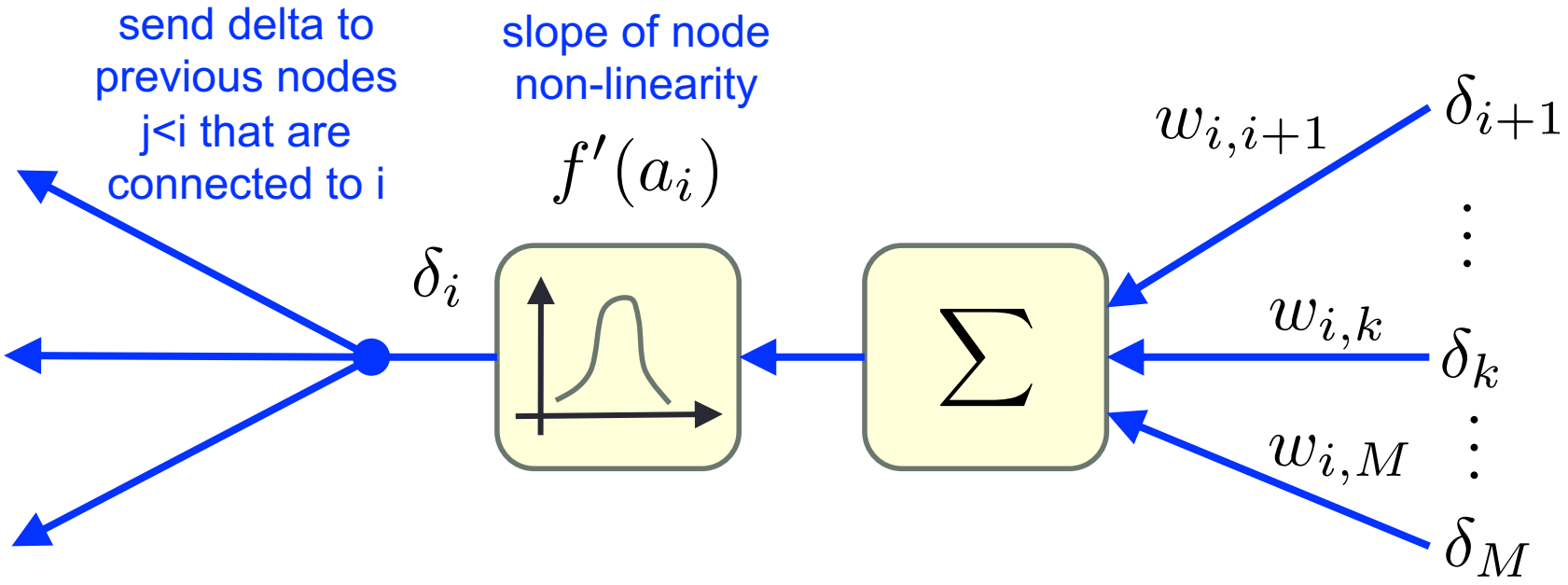
- To summarize, so far (remember: all depends on n)

$$\delta_i = \begin{cases} y_i - t_i & \text{for output nodes} \\ f'(a_i) \sum_{k>i} \delta_k w_{ik} & \text{for hidden nodes} \end{cases}$$

- Because of the way the nodes are indexed, all delta values can be updated in a single sweep through the nodes (starting from the last) in reverse order
- Backpropagation: deltas are first evaluated at output nodes based on current pattern errors, the last hidden layer is then evaluated based on the output delta values, the second-to-last hidden layer is evaluated based on the values of the last hidden layer, and so on…
- Note: backprop stops with the first hidden layer & delta values are not computed for input nodes, as these do not need to adjust the weights

Backprop diagram

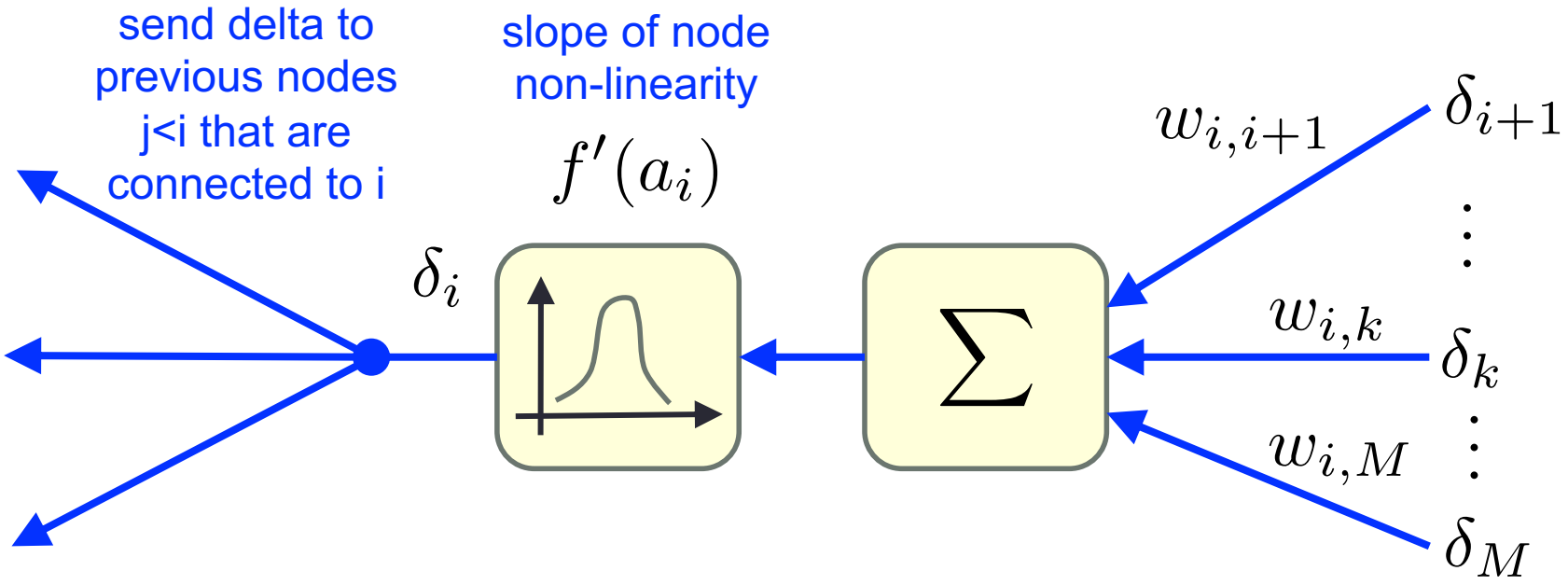
- Backpropagation (M = total number of nodes)



$$\delta_i = \begin{cases} y_i - t_i & \text{for output nodes} \\ f'(a_i) \sum_{k>i} \delta_k w_{ik} & \text{for hidden nodes} \end{cases}$$

Backprop diagram

- Backpropagation (M = total number of nodes)



Backward propagation. Node deltas are calculated in the backward phase, starting at the output nodes and proceeding backwards. At each node i, δ_i is calculated as a weighted linear combination of values δ_k, k>i, of the nodes k to which i sends output. The δ values travel backward against the normal direction of the connecting links.

Partial derivatives wrt weights

- Having obtained the node deltas, finding the derivatives of the error function with respect to the weight is easy
- In fact, we know that:

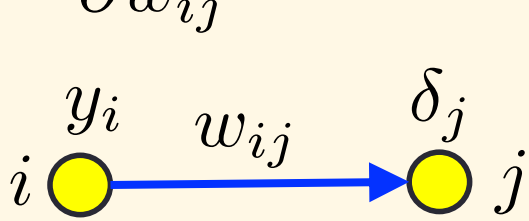
$$\frac{\partial E_n(\boldsymbol{w})}{\partial w_{ij}} = \sum_k \frac{\partial E_n(\boldsymbol{w})}{\partial a_k} \frac{\partial a_k}{\partial w_{ij}} = \sum_k \delta_k \frac{\partial a_k}{\partial w_{ij}}$$

- But, by construction we have:

$$a_k = \sum_{i<k} w_{ik} y_i$$

$$\frac{\partial a_k}{\partial w_{ij}} = \begin{cases} 0 & k \neq j \\ y_i & k = j \end{cases}$$

$$\frac{\partial E_n(\boldsymbol{w})}{\partial w_{ij}} = \delta_j y_i$$



Learning

- There are two basic weight-update variants
  - Batch-mode
    - Compute derivatives for all data points in the training set, add them up and only then performs the weight update (using the total “true” error gradient)
  - On-line (aka “stochastic gradient descent”)
    - Consider one data point at a time. For each point, partial (point-wise) error derivatives are computed and immediately used to update the weights

Batch-mode learning

- Batch-mode (all N points at once)
  - Every input pattern $\boldsymbol{x}_n$ is evaluated to get the derivatives $\frac{\partial E_n(\boldsymbol{w})}{\partial w_{ij}}$
  - These are summed up to obtain the total derivative

$$\frac{\partial E(\boldsymbol{w})}{\partial w_{ij}} = \sum_n \frac{\partial E_n(\boldsymbol{w})}{\partial w_{ij}}$$

- Algorithm
  - Measure $\boldsymbol{x}_n$ and perform forward pass obtaining $\boldsymbol{y}_n$
  - Evaluate error at the output layer $\boldsymbol{y}_n - \boldsymbol{t}_n$
  - Back-propagate error to obtain the derivatives $\partial E_n(\boldsymbol{w})/\partial w_{ij}$
  - Add these derivatives up to obtain the total derivative
  - Update the weights:

$$w_{ij}(t+1) = w_{ij}(t) - \eta \frac{\partial E(\boldsymbol{w})}{\partial w_{ij}}$$

  - Repeat

On-line (or stochastic) learning

- On-line
  - Weight updates are performed for each input pattern
  - Generally patterns are *picked at random* from the data set

- Algorithm
  - Pick a input vector $\boldsymbol{x}_n$ at random from the data set
  - Forward propagate it through the network to obtain output $\boldsymbol{y}_n$
  - Obtain error at output layer $\boldsymbol{y}_n - \boldsymbol{t}_n$
  - Back-propagate error, obtaining $\partial E_n(\boldsymbol{w})/\partial w_{ij}$
  - Update (all) weights immediately:

$$w_{ij}(t+1) = w_{ij}(t) - \eta \frac{\partial E_n(\boldsymbol{w})}{\partial w_{ij}}$$

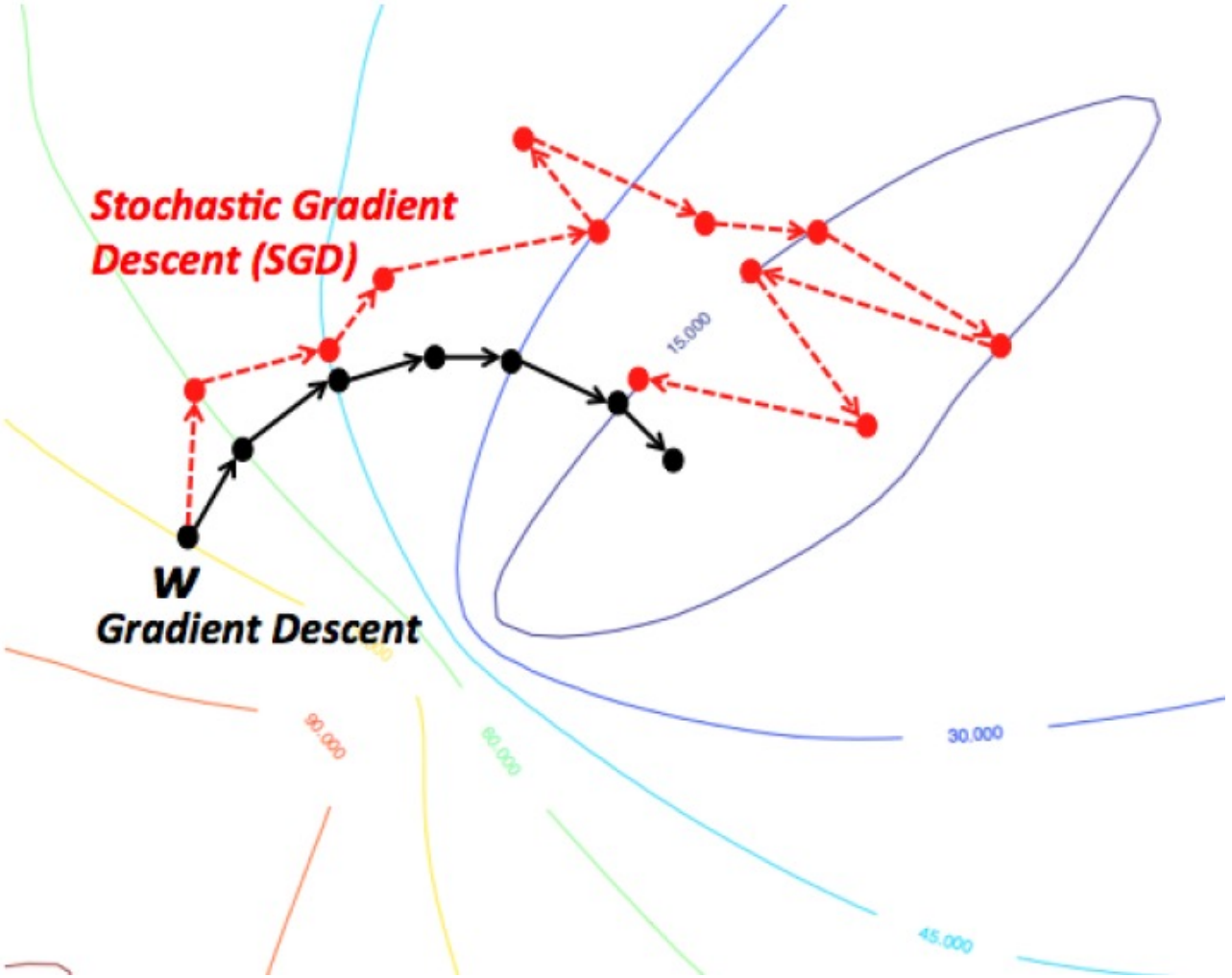
  - Repeat

On-line: pros and cons

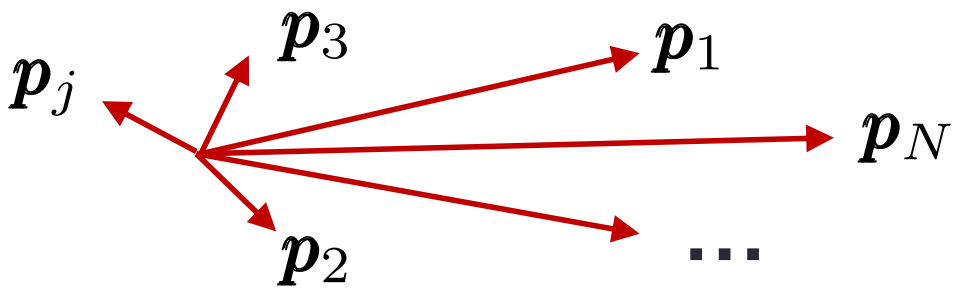
- On-line advantages
  - Weights are updated at each point
  - No need to store lots of values into memory
  - More weight updates per unit time

- On-line learning
  - While the updates add up to the true gradient
  - The local (point-wise) updates are a noisy estimate to the true gradient
  - As such, the error can
    - temporarily increase or deviate from most direct path towards minimum
    - *Instead of taking a smooth descent towards the minimum, the weight vector tends to jitter around the error surface, mostly moving downhill, but occasionally jumping uphill (hence the name "stochastic")*
  - On average, the weight change move downhill

Stochastic vs Batch GD



Pointwise error gradients: examples

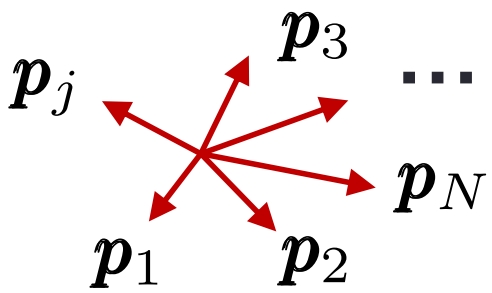


- Pointwise gradient update (one for each input data points)

$$\boldsymbol{p}_n \triangleq \frac{\partial E_n(\boldsymbol{w})}{\partial \boldsymbol{w}} = \left[\frac{\partial E_n(\boldsymbol{w})}{\partial w_{ij}}\right]_{i,j}$$

- In case there is a strong preferential direction → large learning rate makes sense, to move at a fast pace towards the minimum

Pointwise error gradients: examples



- Pointwise gradient update (one for each input data points)

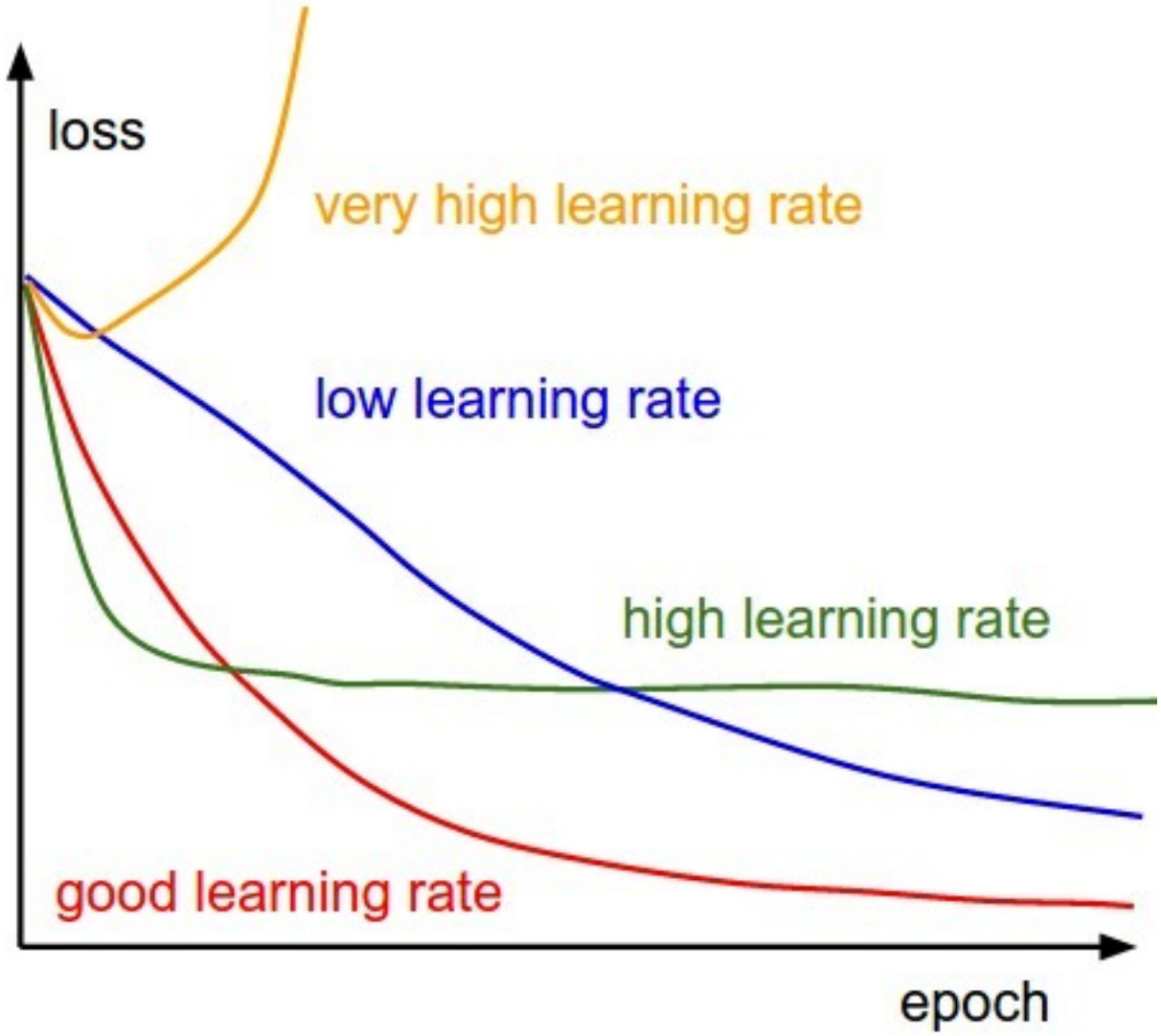
$$\boldsymbol{p}_n \triangleq \frac{\partial E_n(\boldsymbol{w})}{\partial \boldsymbol{w}} = \left[\frac{\partial E_n(\boldsymbol{w})}{\partial w_{ij}}\right]_{i,j}$$

- If there is no preferential direction → large learning rate has to be avoided, as we are probably close to a minimum → smaller learning rate would be preferred to refine the solution around the minimum

Learning rate

- Large
  - If a large majority of single pattern updates have a common direction, the learning rate should be set to a large value, this allows for some stochastic exploration of the error surface, by still converging towards the true minimum
  - With a large learning rate, the search may navigate too much (and endlessly) around the error space, leading to poor convergence
- Small
  - If subsequent weight updates do not have a common direction, this may indicate that we are close to a point of minimum
  - the (true) gradient at that point will be zero, which means that the partial gradients add up to zero, i.e., no major direction in common
  - at this point, it is wise to decrease the learning rate, allowing the search to settle down, close to the true minimum

Learning rate



Momentum

- Common modification to weight update formula
  - Stabilize the weight trajectory by combining
    - Gradient descent term
    - A fraction of previous weight change
- Modified weight update equation

$$\Delta \boldsymbol{w}(t) = -\eta \frac{\partial E_n(\boldsymbol{w})}{\partial \boldsymbol{w}}(t) + \alpha \Delta \boldsymbol{w}(t-1)\,,\ \alpha \in [0,1)$$

  - This provides a certain inertia, as the weight update tends to move in the previous direction, unless opposed by the gradient
  - Momentum tends to damp oscillations in the weight trajectory and speed up convergence in regions where gradient is small

Bibliography

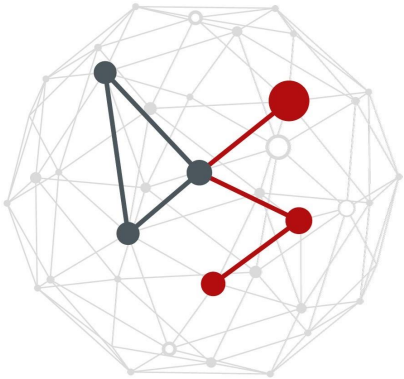
- [Ros62] Frank Rosenblatt, “Principles of neurodynamics; perceptrons and the theory of brain mechanisms,” Washington: Spartan Books, 1962.
- [Seppo70] Seppo Linnainmaa, "The representation of the cumulative rounding error of an algorithm as a Taylor expansion of the local rounding errors,” Master Thesis, 1970.
- [Cybenko89] G. Cybenko, “Approximation by superpositions of a sigmoidal function,” Mathematics of Control, Signals and Systems, Vol. 2, No. 4, December 1989.
- [Clevert16] D.-A.Clevert, T. Unterthiner, S. Hochreiter, “Fast and accurate deep network learning by Exponential Linear Units (ELUs),” International Conference on Learning Representations, San Juan, Puerto Rico, 2016.

Books

- Chapter 5 of: [Reed99] Russel D. Reed, Robert J. Marks II, “Neural Smithing: Supervised Learning in Feedforward Neural Networks,” Bradford books, 1999.
- Chapter 5 of: [Bish06] Cristopher Bishop, “Machine Learning and Pattern Recognition,” Springer Verlag, 2006.

MIME
Master's degree ICT Internet Multimedia Engineering

APPENDIX A
BACKPROP WITH SOFTMAX



DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

DIPARTIMENTO
MATEMATICA

800 ANNI
1222 · 2022

UNIVERSITÀ
DEGLI STUDI
DI PADOVA

Softmax output

- For output node i, we have

$$y_i = \frac{\exp(a_i)}{\sum_k \exp(a_k)}$$

- The expression for the deltas is still:

$$\delta_i = \frac{\partial E_n(\boldsymbol{w})}{\partial a_i} = \frac{\partial E_n(\boldsymbol{w})}{\partial y_i} \frac{\partial y_i}{\partial a_i}$$

- We compute, in the following order:

$$\frac{\partial y_i}{\partial a_i} = ? \qquad \frac{\partial E_n(\boldsymbol{w})}{\partial a_i} = ?$$

Derivative of softmax (1/4)

- For output node i, we have

$$y_i = \frac{\exp(a_i)}{\sum_k \exp(a_k)} \qquad \frac{\partial y_i}{\partial a_j} = ?$$

- Derivative of a quotient:

$$f(x) = \frac{g(x)}{h(x)} \rightarrow f'(x) = \frac{g'(x)h(x) - h'(x)g(x)}{h(x)^2}$$

- We use:

$$g(x) = \exp(a_i)\,,\; h(x) = \sum_k \exp(a_k)$$

Derivative of softmax (2/4)

if $j = i$

$$\frac{\partial y_i}{\partial a_j} = \frac{\exp(a_i)\sum_k \exp(a_k) - \exp(a_j)\exp(a_i)}{\left(\sum_k \exp(a_k)\right)^2} =$$

$$= \frac{\exp(a_i)\left(\sum_k \exp(a_k) - \exp(a_j)\right)}{\left(\sum_k \exp(a_k)\right)^2} =$$

$$= \frac{\exp(a_i)}{\sum_k \exp(a_k)} \times \frac{\left(\sum_k \exp(a_k) - \exp(a_j)\right)}{\sum_k \exp(a_k)} =$$

$$= y_i(1 - y_j) = y_i(1 - y_i) \text{ (as } i = j)$$

Derivative of softmax (3/4)

if $j \neq i$

$$\frac{\partial y_i}{\partial a_j} = \frac{0 - \exp(a_j)\exp(a_i)}{\left(\sum_k \exp(a_k)\right)^2} =$$

$$= \frac{-\exp(a_j)}{\left(\sum_k \exp(a_k)\right)} \times \frac{\exp(a_i)}{\left(\sum_k \exp(a_k)\right)} =$$

$$= -y_j y_i$$

Derivative of softmax (4/4)

- Thus, we get:

$$\frac{\partial y_i}{\partial a_j} = \begin{cases} y_i(1 - y_i) & i = j \\ -y_j y_i & i \neq j \end{cases}$$

- Compactly, using the Kronecker delta

$$\delta_{ij} = 1 \text{ if } i = j, \delta_{ij} = 0 \text{ otherwise}$$

$$\frac{\partial y_i}{\partial a_j} = y_i(\delta_{ij} - y_j)$$

Derivative of cross-entropy loss (1/2)

$$E(\boldsymbol{w}) = -\sum_{n=1}^{N}\sum_{k=1}^{K} t_{kn}\ln(y_{kn}) = \sum_{n=1}^{N} E_n(\boldsymbol{w})$$

where:

$$y_{kn} = y_k(\boldsymbol{x}_n, \boldsymbol{w})$$

- For the n-th input data vector, the error function is

$$E_n(\boldsymbol{w}) = -\sum_{k=1}^{K} t_{kn}\ln(y_{kn}) =$$

$$= -\sum_{k=1}^{K} t_k \ln(y_k) \text{ (omitting } n)$$

Derivative of cross-entropy loss (2/2)

$$\frac{\partial E_n(\boldsymbol{w})}{\partial a_i} = ?$$

$$\frac{\partial E_n(\boldsymbol{w})}{\partial a_i} = -\sum_k t_k \frac{\partial \log(y_k)}{\partial y_k} \times \frac{\partial y_k}{\partial a_i} =$$

$$= -\sum_k t_k \frac{1}{y_k} \times \frac{\partial y_k}{\partial a_i} = \underbrace{-t_i(1-y_i)}_{k=i} - \underbrace{\sum_{k \neq i} t_k \frac{1}{y_k}(-y_k y_i)}_{k \neq i} =$$

$$= -t_i + t_i y_i + \sum_{k \neq i} t_k y_i = -t_i + y_i \underbrace{(t_i + \sum_{k \neq i} t_k)}_{\text{labels use 1-of-K encoding } \sum_k t_k = 1} =$$

$$= y_i - t_i$$

same result as regression

Final result

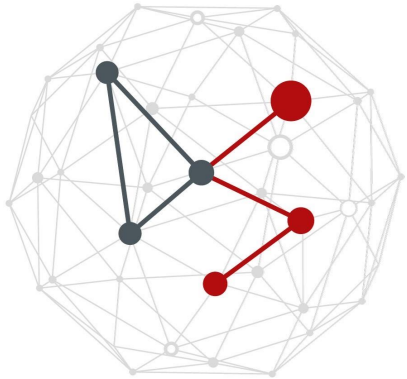
- From what shown, for a classification problem, with:
  - softmax output function
  - multiclass cross-entropy loss function
- We still have

$$\delta_i = \begin{cases} y_i - t_i & \text{for output nodes} \\ f'(a_i) \sum_{k>i} \delta_k w_{ik} & \text{for hidden nodes} \end{cases}$$

  - Partial derivatives of error functions are the same also
  - So nothing changes, although error values will differ as they will represent probabilities in this case

MIME
Master's degree ICT Internet Multimedia Engineering

APPENDIX B
HYPERBOLIC TANGENT



DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

DIPARTIMENTO
MATEMATICA

800 ANNI
1222 · 2022

UNIVERSITÀ
DEGLI STUDI
DI PADOVA

Derivative of tanh

- When the tanh activation function is used

$$f(a_i) = \frac{e^{a_i} - e^{-a_i}}{e^{a_i} + e^{-a_i}}$$

- From the derivative of the quotient, we get

$$f'(a_i) = \frac{\partial f(a_i)}{\partial a_i} = \frac{(e^{a_i} + e^{-a_i})^2 - (e^{a_i} - e^{-a_i})^2}{(e^{a_i} + e^{-a_i})^2} =$$

$$= 1 - f(a_i)^2$$

Backprop update

- When the tanh activation function is used

$$\begin{cases} y_i = f(a_i) \\ f'(a_i) = 1 - f(a_i)^2 = 1 - y_i^2 \end{cases}$$

- The backprop update rule becomes

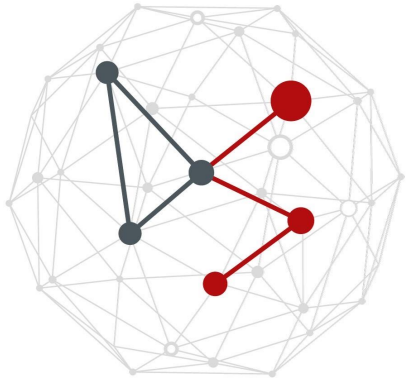
$$\delta_i = \begin{cases} y_i - t_i & \text{for output nodes} \\ (1 - y_i^2) \sum_{k>i} \delta_k w_{ik} & \text{for hidden nodes} \end{cases}$$

MIME
Master's degree ICT Internet Multimedia Engineering

FEED FORWARD NEURAL NETWORKS (FFNN)

Michele Rossi
michele.rossi@dei.unipd.it

Dept. of Information Engineering
University of Padova, IT



DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

DIPARTIMENTO MATEMATICA

800 ANNI 1222·2022

UNIVERSITÀ DEGLI STUDI DI PADOVA