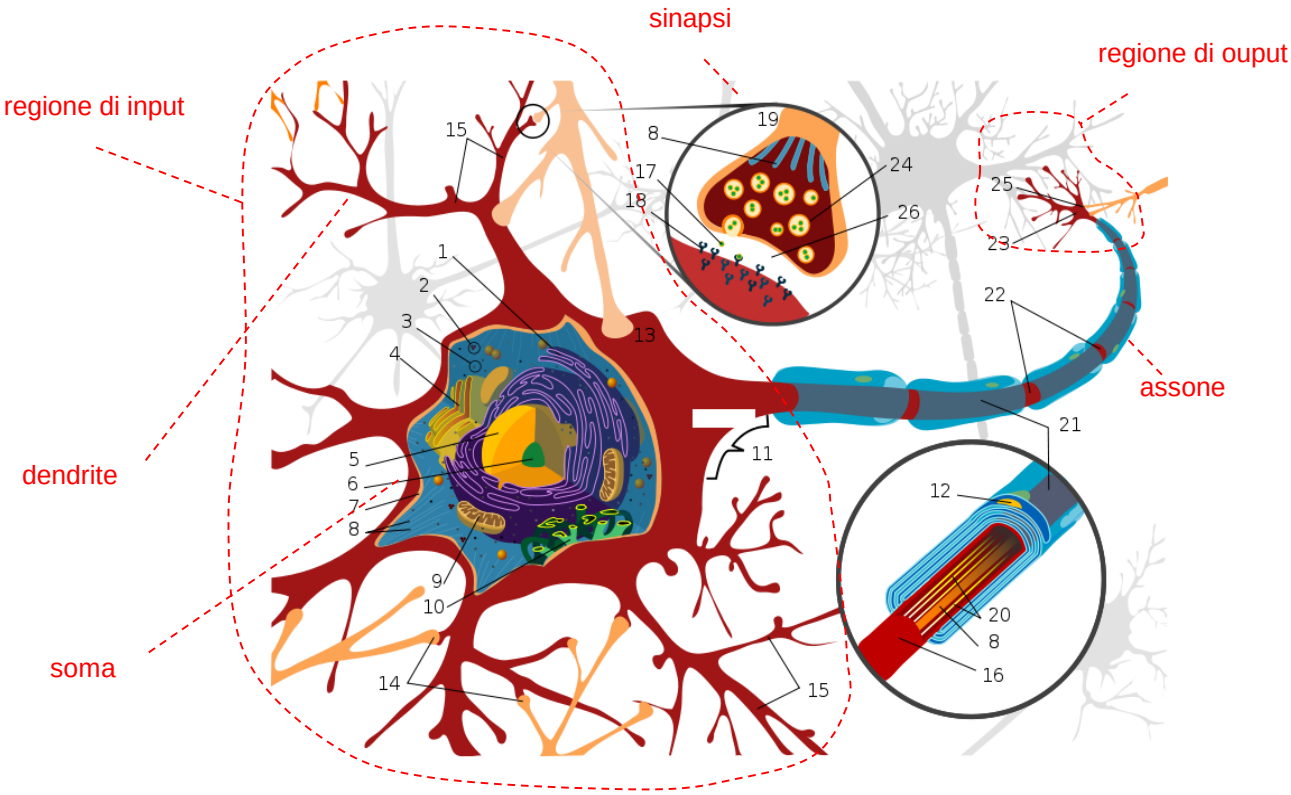
Reti Neurali

- Introduzione
  - Neuroni Biologici
  - Neuroni Artificiali
  - Tipologie di Reti
- Multilayer Perceptron (MLP)
  - Forward propagation
  - Training: Error Backpropagation
  - On-line Backpropagation
  - Stochastic Gradient Descent (SGD)
- Approfondimenti
  - Softmax e Cross-Entropy
  - Regolarizzazione (Weight Decay)
  - Momentum
  - Learning Rate adattativo

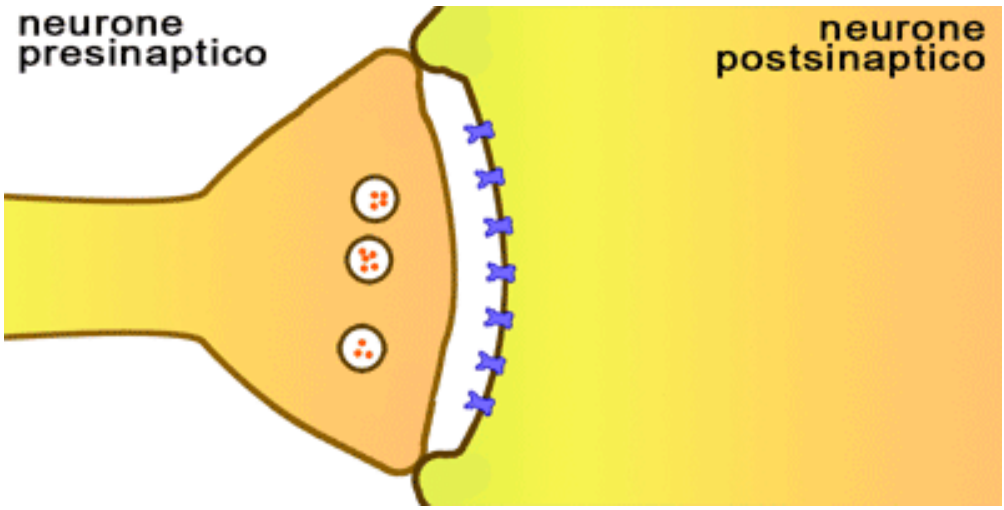
Neuroni Biologici



- Le connessioni sinaptiche o (sinapsi) agiscono come porte di collegamento per il passaggio dell'informazione tra neuroni.
- I dendriti sono fibre minori che si ramificano a partire dal corpo cellulare del neurone (detto soma). Attraverso le sinapsi i dendriti raccolgono input da neuroni afferenti e li propagano verso il soma.
- L'assone è la fibra principale che parte dal soma e si allontana da esso per portare ad altri neuroni (anche distanti) l'output.

Neuroni Biologici (2)

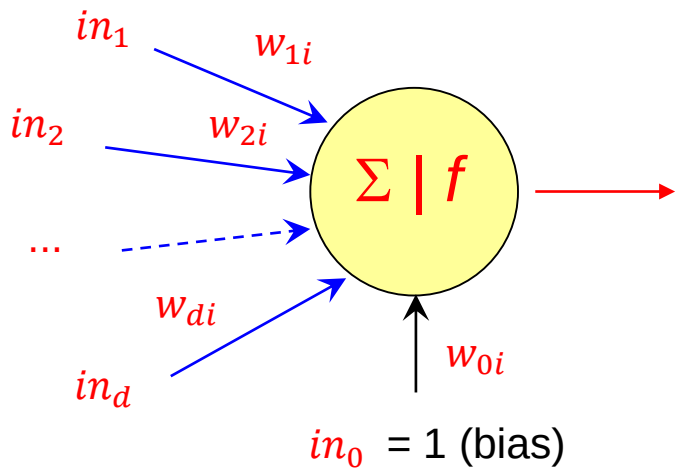
- Il passaggio delle informazioni attraverso le sinapsi avviene con processi elettro-chimici.



- L'ingresso di ioni attraverso le sinapsi dei dendriti determina la formazione di una differenza di potenziale tra il corpo del neurone e l'esterno. Quando questo potenziale supera una certa soglia si produce uno spike (impulso): il neurone propaga un breve segnale elettrico detto potenziale d'azione lungo il proprio assone: questo potenziale determina il rilascio di ioni dalle sinapsi dell'assone.
- La modifica dell'efficacia delle sinapsi (ovvero della loro capacità modulante) è direttamente collegata a processi di apprendimento e memoria.
- La modalità di trattamento delle informazioni da parte di una rete neurale dipende principalmente dalla topologia della rete (connessioni tra neuroni) e dalla modulazione delle sinapsi.

Neurone Artificiale

- Introdotto da McCulloch and Pitts, 1943.



$$net_i = \sum_{j=1\ldots d} w_{ji} \cdot in_j + w_{0i}$$

$$out_i = f(net_i)$$

- In figura è raffigurato un neurone di indice i: *una rete neurale ne contiene molti, dobbiamo distinguerli …*
  - in_1, in_2, … in_d sono i d ingressi che il neurone i riceve da assoni di neuroni afferenti.
  - w_{1i}, w_{2i}, … w_{di} sono i pesi (weight) che determinano l'efficacia delle connessioni sinaptiche dei dendriti (agiremo su questi valori durante l'apprendimento).
  - w_{0i} (detto bias) è un ulteriore peso che si considera collegato a un input fittizio con valore sempre 1; questo peso è utile per «tarare» il punto di lavoro ottimale del neurone.
  - net_i è il livello di eccitazione globale del neurone (potenziale interno);
  - $f(\cdot)$ è la funzione di attivazione che determina il comportamento del neurone (ovvero il suo output net_i) in funzione del suo livello di eccitazione net_i.

Neurone: funzione di attivazione

- Nei neuroni biologici $f(\cdot)$ è una funzione tutto-niente temporizzata: quando net_i supera una certa soglia, il neurone «spara» uno spike (impulso) per poi tornare a riposo.
  - Esistono reti artificiali (denonimate spiking neural network) che modellano questo comportamento e processano l'informazione attraverso treni di impulsi.
  - È necessaria questa «complicazione»? Oppure codificare l'informazione con impulsi (invece che con livelli continui) è solo una questione di risparmio energetico messa a punto dall'evoluzione?

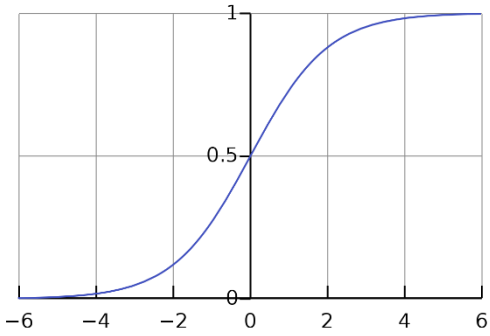
- Le reti neurali più comunemente utilizzate operano con livelli continui e $f(\cdot)$ è una funzione non-lineare ma continua e differenziabile (quasi ovunque).
  - Perché non-lineare? Se vogliamo che una rete sia in grado di eseguire un mapping (complesso) dell'informazione di input sono necessarie non-linearità.
  - Perché continua e differenziabile? Necessario per la retro-propagazione dell'errore (come scopriremo presto).

- Una delle funzioni di attivazione più comunemente utilizzate è la sigmoide nelle varianti:
  - standard logistic function (chiamata semplicemente sigmoid)
  - tangente iperbolica (tanh)

Funzione di attivazione: sigmoide

- Standard logistic function (Sigmoid), (valori in [0…1]):

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



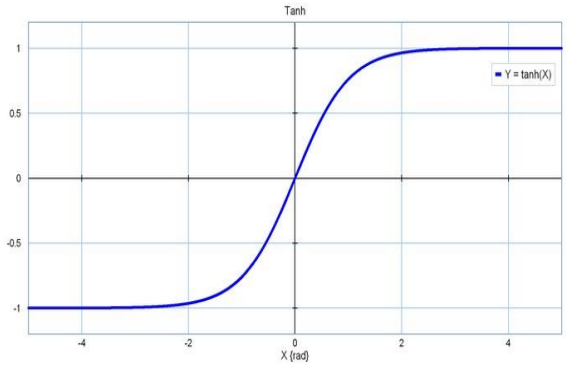
La derivata:

$$\sigma'(net) = \frac{\partial}{\partial net}\left(\frac{1}{1 + e^{-net}}\right) = \frac{e^{-net}}{(1 + e^{-net})^2} = \sigma(net)\big(1 - \sigma(net)\big)$$

- Tangente iperbolica (Tanh), (valori in [-1…1]):

Può essere ottenuta dalla funzione precedente a seguito di trasformazione di scala (×2) e traslazione (-1).

$$f(net) = \tau(net) = 2\sigma(2 \cdot net) - 1$$



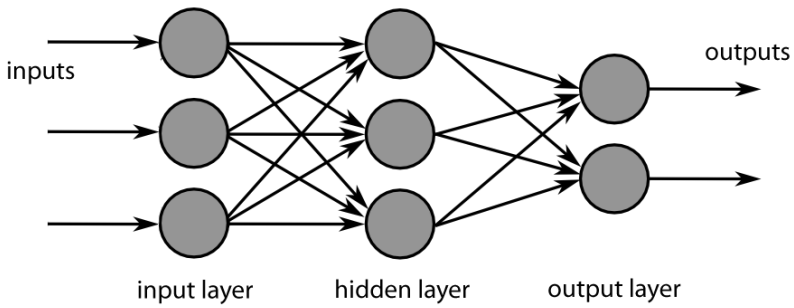
La derivata: $\tau'(net) = 1 - \tau(net)^2$

Tanh è leggermente più complessa di sigmoid ma preferibile (convergenza più rapida), in quanto simmetrica rispetto allo zero.

Tipologie di reti neurali

Le reti neurali sono composte da gruppi di neuroni artificiali organizzati in livelli. Tipicamente sono presenti: un livello di input, un livello di output, e uno o più livelli intermedi o nascosti (hidden). Ogni livello contiene uno o più neuroni.

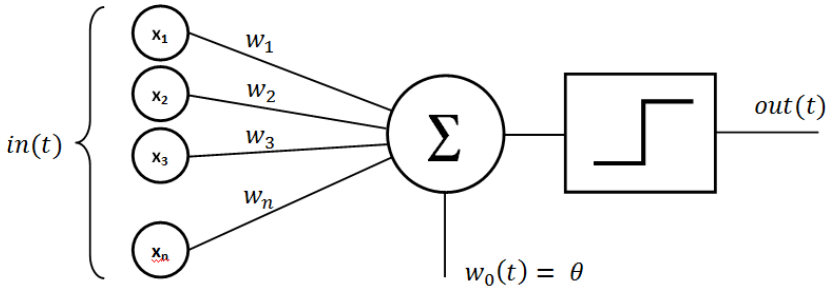
- Feedforward: nelle reti feedforward («alimentazione in avanti») le connessioni collegano i neuroni di un livello con i neuroni di un livello successivo. Non sono consentite connessioni all'indietro o connessioni verso lo stesso livello. È di gran lunga il tipo di rete più utilizzata.



- Recurrent: nelle reti ricorrenti sono previste connessioni di feedback (in genere verso neuroni dello stesso livello, ma anche all'indietro). Questo complica notevolmente il flusso delle informazioni e l'addestramento, richiedendo di considerare il comportamento in più istanti temporali (unfolding in time). D'altro canto queste reti sono più indicate per la gestione di sequenze (es. audio, video, frasi in linguaggio naturale), perchè dotate di un effetto memoria (di breve termine) che al tempo t rende disponibile l'informazione processata a $t-1$, $t-2$, ecc. Un particolare tipo di rete ricorrente è LSTM (Long Short Term Memory). *Non tratteremo queste reti nel corso.*

Multilayer Perceptron (MLP)

- Il termine perceptron (percettrone) deriva dal modello di neurone proposto da Rosenblatt nel 1956, molto simile a quello di McCulloch e Pitts.
  - Il percettrone originale utilizza una funzione di attivazione lineare a soglia (o scalino). Un singolo percettrone, o una rete di percettroni a due soli livelli (input e output), può essere addestrato con una semplice regola detta delta rule.



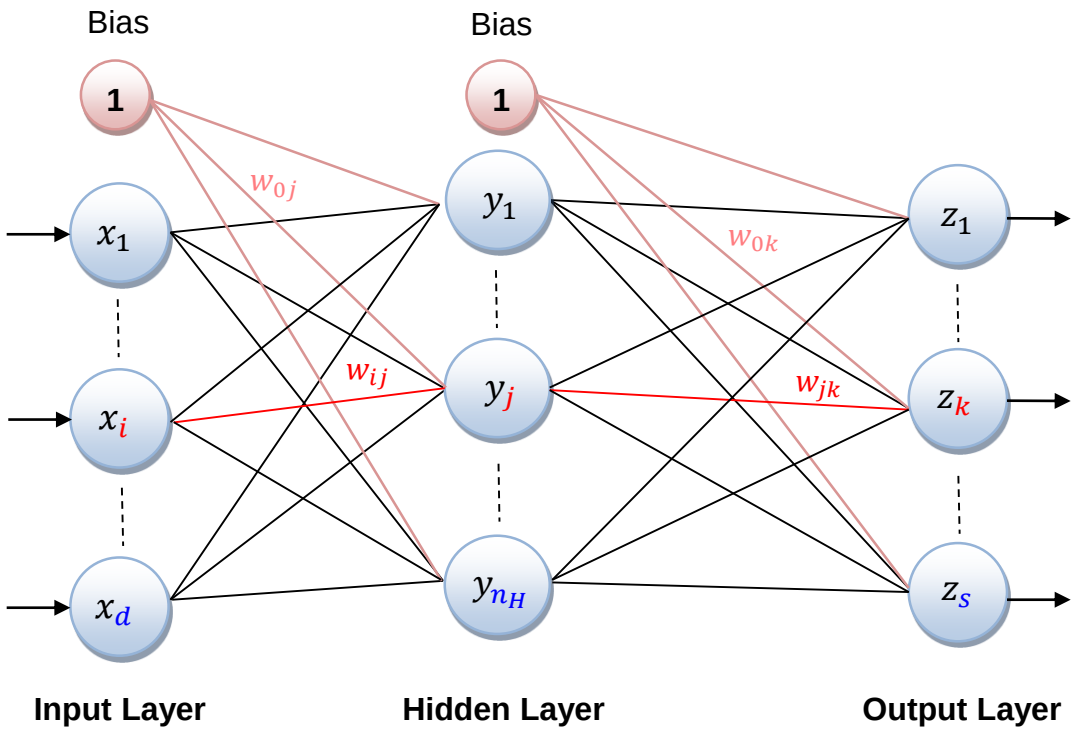
  - Una rete a due livelli di percettroni lineari a soglia è in grado di apprendere solo mapping lineari e pertanto il numero di funzioni approssimabili è piuttosto limitato.

- Un Multilayer Perceptron (MLP) è una rete feedforward con almeno 3 livelli (almeno 1 hidden) e con funzioni di attivazione non lineari.

- Un teorema noto come universal approximation theorem asserisce che ogni funzione continua che mappa intervalli di numeri reali su un intervallo di numeri reali può essere approssimata da un MLP con un solo hidden layer.
  - questa è una delle motivazioni per cui per molti decenni (fino all'esplosione del deep learning) ci si è soffermati su reti neurali a 3 livelli. D'altro canto l'esistenza teorica di una soluzione non implica che esista un modo efficace per ottenerla…

MLP: forward propagation

- Con forward propagation (o inference) si intende la propagazione delle informazioni in avanti: dal livello di input a quello di output. Una volta addestrata, una rete neurale può semplicemente processare pattern attraverso forward propagation.

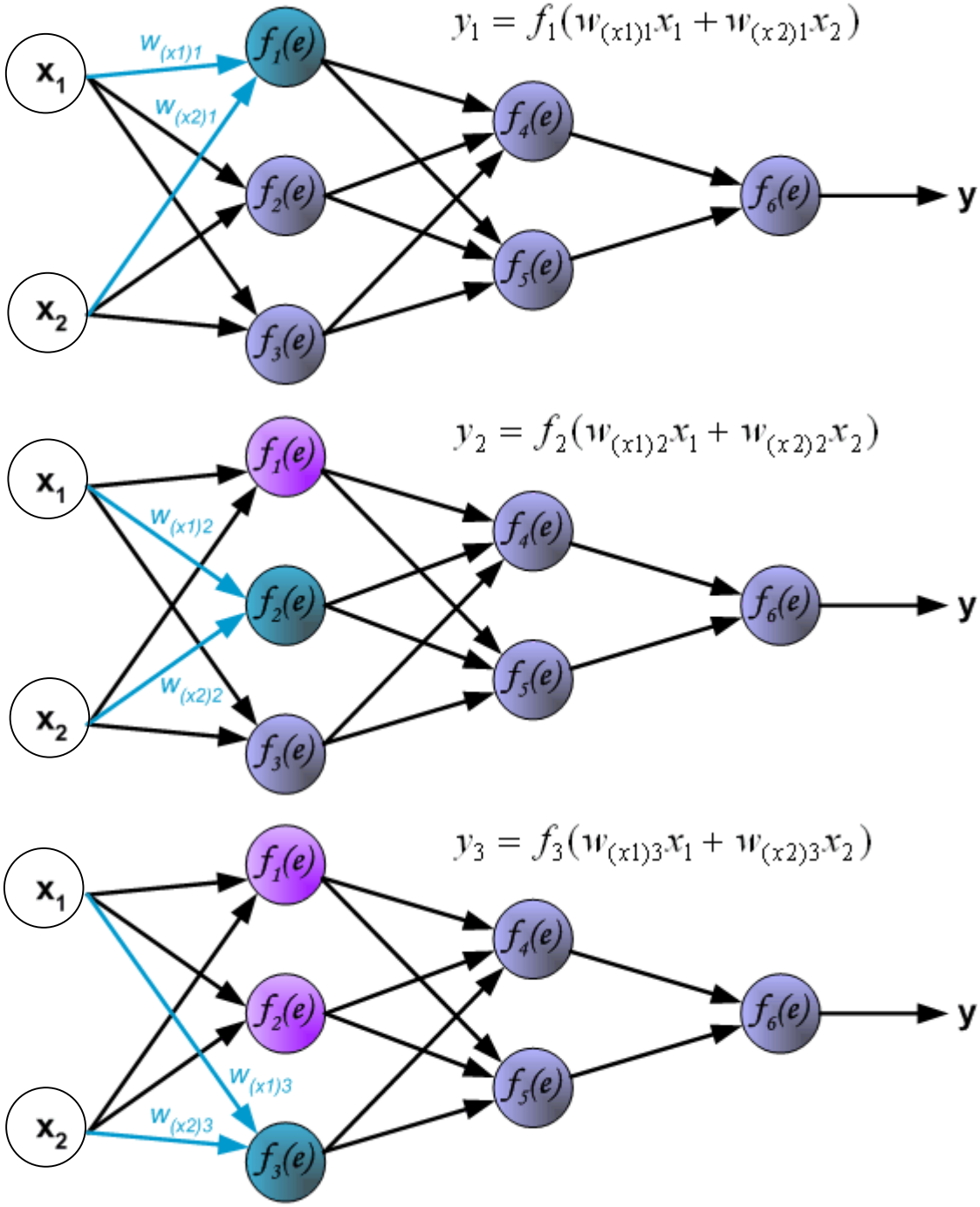


- nell'esempio una rete a 3 livelli (d:n_H:s): input d neuroni, livello nascosto (hidden) n_H neuroni, output s neuroni.
- Il numero totale di pesi (o parametri) è: $d \times n_H + n_H \times s + n_H + s$ dove gli ultimi due termini corrispondono ai pesi dei bias.
- Il k-esimo valore di output può essere calcolato come: (Eq. 1)

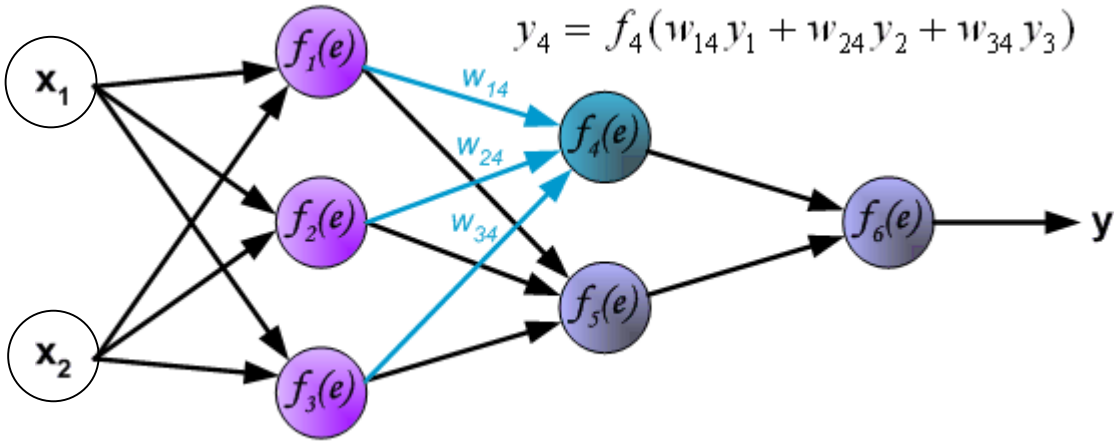
$$z_k = f\left(\sum_{j=1\ldots n_H} w_{jk} \cdot y_j + w_{0k}\right) = f\left(\sum_{j=1\ldots n_H} w_{jk} \cdot f\left(\sum_{i=1\ldots d} w_{ij} \cdot x_i + w_{0j}\right) + w_{0k}\right)$$

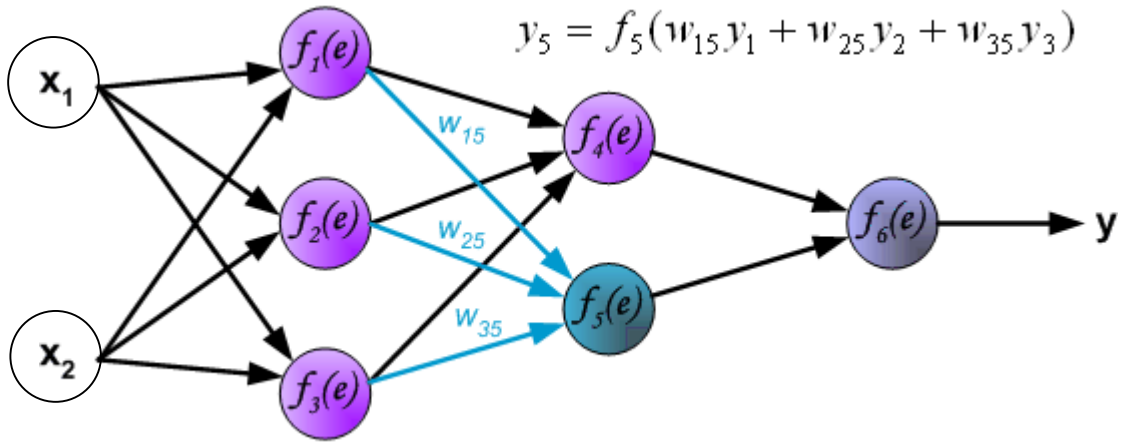
Forward propagation: esempio grafico

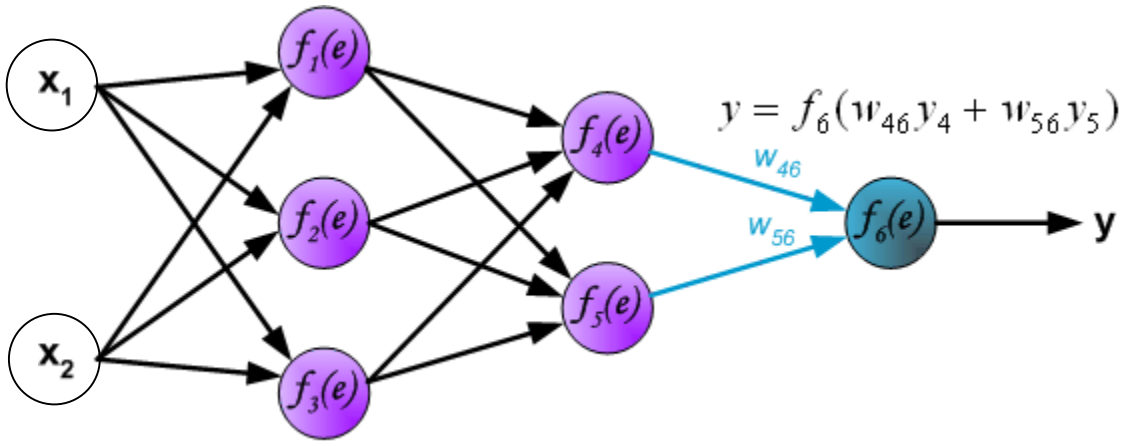
- Nell'esempio una rete a 4 livelli 2:3:2:1 (2 livelli nascosti); non sono usati bias; nella grafica dell'esempio la notazione è un po' diversa dalla precedente ma facilmente comprensibile.



…continua


$y_4 = f_4(w_{14}y_1 + w_{24}y_2 + w_{34}y_3)$
x1
x2
f1(e)
f2(e)
f3(e)
f4(e)
f5(e)
f6(e)
w14
w24
w34
y


$y_5 = f_5(w_{15}y_1 + w_{25}y_2 + w_{35}y_3)$
x1
x2
f1(e)
f2(e)
f3(e)
f4(e)
f5(e)
f6(e)
w15
w25
w35
y


$y = f_6(w_{46}y_4 + w_{56}y_5)$
x1
x2
f1(e)
f2(e)
f3(e)
f4(e)
f5(e)
f6(e)
w46
w56
y

MLP: Training

- Fissata la topologia (numero di livelli e neuroni), l'addestramento di una rete neurale consiste nel determinare il valore dei pesi **w** che determinano il mapping desiderato tra input e output.
- *Che cosa intendiamo per mapping desiderato?* Dipende dal problema che vogliamo risolvere:
  - se siamo interessati ad addestrare una rete neurale che operi come classificatore, l'output desiderato è l'etichetta corretta della classe del pattern in input.
  - se la rete neurale deve risolvere un problema di regressione, l'output desiderato è il valore corretto della variabile dipendente, in corrispondenza del valore della variabile indipendente fornita in input.

- Sebbene i primi neuroni artificiali risalgano agli anni 40', fino a metà degli anni 80' non erano disponibili algoritmi di training efficaci.
- Nel 1986 Rumelhart, Hinton & Williams hanno introdotto l'algoritmo di Error Backpropagation suscitando grande attenzione nella comunità scientifica.
- Si tratta in realtà dell'applicazione della regola di derivazione a catena, idea che però per decenni nessuno aveva applicato con successo in questo contesto.

Training Supervisionato di un Classificatore

Nel seguito focalizziamo l'attenzione su un problema di classificazione supervisionata.

- Per gestire un problema di classificazione con s classi e pattern d-dimensionali, si è soliti utilizzare una rete con $d : n_H : s$ neuroni nei tre livelli. Ovvero tanti neuroni di input quante sono le feature e tanti neuroni di output quante sono le classi. n_H è invece un iperparametro: un valore ragionevole è $n_H = {}^{1}/_{10}\, n$.
- La pre-normalizzazione dei pattern di input (es. attraverso whitening) sebbene non obbligatoria può favorire la convergenza durante l'addestramento.
- L'addestramento (supervisionato) avviene presentando alla rete pattern di cui è nota la classe e propagando (forward propagation) gli input verso gli output attraverso l'equazione (1).
- Dato un pattern di training di classe g, il vettore di output desiderato $\mathbf{t}$ (usando Tanh come funzione di attivazione) assume la forma:

$$\mathbf{t} = [-1, -1 \ldots \underset{\text{posizione } g}{1} \ldots -1]$$

- La differenza tra l'output prodotto della rete e quello desiderato è l'errore della rete. Obiettivo dell'algoritmo di apprendimento è di modificare i pesi della rete in modo da minimizzare l'errore medio sui pattern del training set.
- Prima dell'inizio dell'addestramento i pesi sono tipicamente inizializzati con valori random:
  - quelli input-hidden nel range $\pm 1/\sqrt{d}$
  - quelli hidden-output nel range $\pm 1/\sqrt{n_H}$

MLP: Error Backpropagation

Con riferimento alla rete a 3 livelli della slide 9:

- Sia $\mathbf{z} = [z_1, z_2 \ldots z_s]$ l'output prodotto dalla rete (forward propagation) in corrispondenza del pattern $\mathbf{x} = [x_1, x_2 \ldots x_d]$ di classe g fornito in input; come già detto l'output desiderato è $\mathbf{t} = [t_1, t_2 \ldots t_s]$, dove $t_i = 1$ per $i = g$, $t_i = -1$ altrimenti.
- Scegliendo come loss function la somma dei quadrati degli errori, l'errore (per il pattern $\mathbf{x}$) è:

$$J(\mathbf{w}, \mathbf{x}) \equiv \frac{1}{2} \sum_{c=1 \ldots s} (t_c - z_c)^2$$

  che quantifica quanto l'output prodotto per il pattern $\mathbf{x}$ si discosta da quello desiderato. La dipendenza dai pesi $\mathbf{w}$ è implicita in $\mathbf{z}$. L'errore $J(\mathbf{w})$ sull'intero training set è la media di $J(\mathbf{w}, \mathbf{x})$ su tutti i pattern $\mathbf{x}$ appartenenti al training set.
- $J(\mathbf{w})$ può essere ridotto modificando i pesi $\mathbf{w}$ in direzione opposta al gradiente di J. Infatti il gradiente indica la direzione di maggior crescita di una funzione (di più variabili) e muovendoci in direzione opposta riduciamo (al massimo) l'errore.
- Quando la minimizzazione dell'errore avviene attraverso passi in direzione opposta al gradiente l'algoritmo backpropagation è denominato anche gradient descent. Esistono anche tecniche di minimizzazione del secondo ordine che possono accelerare convergenza (applicabili in pratica solo a reti e training set di piccole/medie dimensioni).
- Nel seguito, consideriamo:
  - prima la modifica dei pesi w_{jk} hidden-output.
  - poi quella dei pesi w_{ij} input-hidden.

Modifica pesi hidden-output

$$\frac{\partial J}{\partial w_{jk}} = \frac{\partial}{\partial w_{jk}}\left(\frac{1}{2}\sum_{c=1\ldots s}(t_c - z_c)^2\right) = (t_k - z_k)\frac{\partial(-z_k)}{\partial w_{jk}} =$$

solo z_k è influenzato da w_{jk}

$$= (t_k - z_k)\frac{\partial\left(-f(net_k)\right)}{\partial w_{jk}} = -(t_k - z_k)\cdot\frac{f(net_k)}{\partial net_k}\cdot\frac{\partial net_k}{\partial w_{jk}} =$$

$$= -(t_k - z_k)\cdot f'(net_k)\cdot\frac{\partial\sum_{s=1\ldots n_H} w_{sk}\cdot y_s}{\partial w_{jk}} = -(t_k - z_k)\cdot f'(net_k)\cdot y_j$$

posto $\delta_k = (t_k - z_k)\cdot f'(net_k)$ (Eq. 2)

allora $\frac{\partial J}{\partial w_{jk}} = -\delta_k \cdot y_j$ (Eq. 3)

pertanto il peso w_{jk} può essere aggiornato come:

$$w_{jk} = w_{jk} + \eta\cdot\delta_k\cdot y_j \quad \text{(Eq. 4)}$$

dove η è il learning rate.

Se η troppo piccolo convergenza lenta, se troppo grande può oscillare e/o divergere. Partire con $\eta = 0.5$ e modificare il valore se non adeguato (monitorando l'andamento del loss durante le iterazioni).

N.B. se si usano i bias il peso w_{0k} si aggiorna ponendo $y_0 = 1$

Modifica pesi input-hidden

$$\frac{\partial J}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}}\left(\frac{1}{2}\sum_{c=1\dots s}(t_c - z_c)^2\right) = -\sum_{c=1\dots s}(t_c - z_c)\frac{\partial z_c}{\partial w_{ij}} =$$

$$= -\sum_{c=1\dots s}(t_c - z_c)\frac{\partial z_c}{\partial net_c}\cdot\frac{\partial net_c}{\partial w_{ij}} = -\sum_{c=1\dots s}\underbrace{(t_c - z_c)\cdot f'(net_c)}_{\delta_c \text{ vedi Eq. 2}}\cdot\frac{\partial net_c}{\partial w_{ij}}$$

$$\frac{\partial net_c}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}}\sum_{r=1\dots n_H} w_{rc}\cdot y_r = \frac{\partial}{\partial w_{ij}}\sum_{r=1\dots n_H} w_{rc}\cdot f(net_r) =$$

solo net_j è influenzato da w_{ij}

$$= \frac{\partial}{\partial w_{ij}}\left(w_{jc}\cdot f(net_j)\right) = w_{jc}\frac{\partial f(net_j)}{\partial net_j}\cdot\frac{\partial net_j}{\partial w_{ij}} =$$

$$= w_{jc}\cdot f'(net_j)\cdot\frac{\partial}{\partial w_{ij}}\sum_{q=1\dots d} w_{qj}\cdot x_q = w_{jc}\cdot f'(net_j)\cdot x_i$$

$$\frac{\partial J}{\partial w_{ij}} = -\sum_{c=1\dots s}\delta_c\cdot w_{jc}\cdot f'(net_j)\cdot x_i = -x_i\cdot f'(net_j)\sum_{c=1\dots s} w_{jc}\cdot\delta_c$$

posto $\delta_j = f'(net_j)\cdot\sum_{c=1\dots s} w_{jc}\cdot\delta_c$ (Eq. 5)

allora $\frac{\partial J}{\partial w_{ij}} = -\delta_j\cdot x_i$ (Eq. 6)

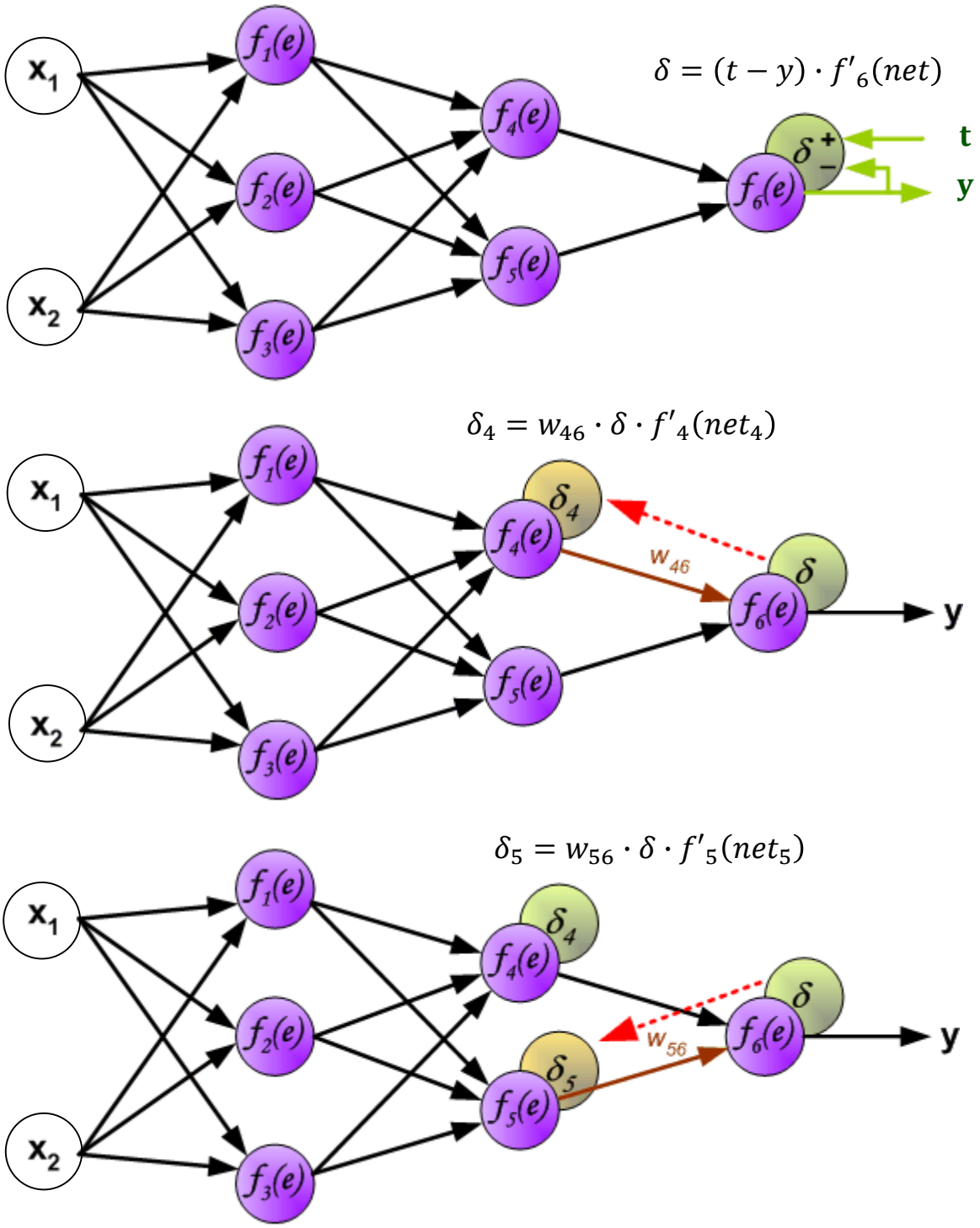
pertanto il peso w_{ij} può essere aggiornato come:

$$w_{ij} = w_{ij} + \eta\cdot\delta_j\cdot x_i \quad \text{(Eq. 7)}$$

N.B. se si usano i bias il peso w_{0j} si aggiorna ponendo $x_0 = 1$

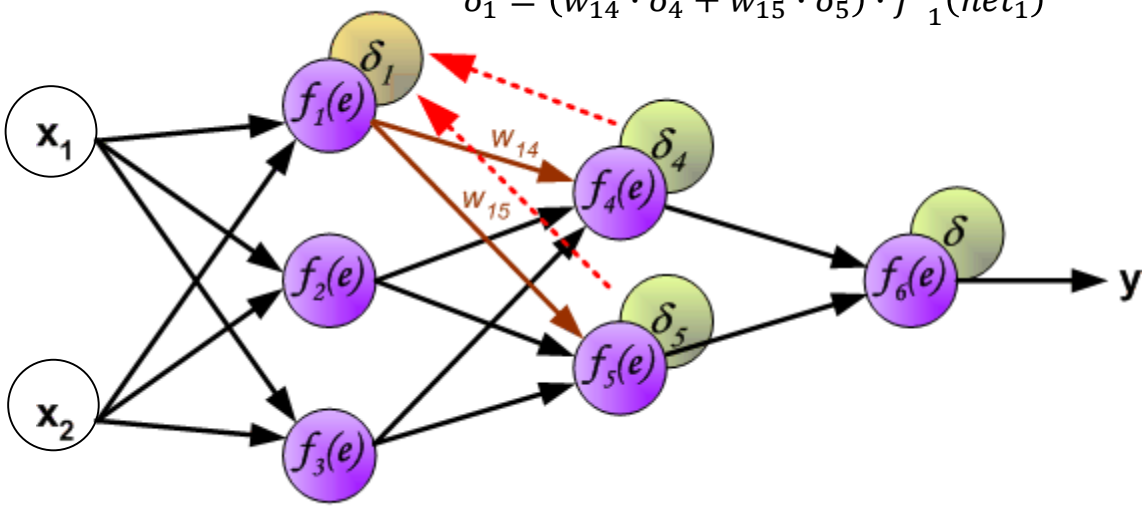
Backpropagation: esempio grafico

Stessa rete a 4 livelli su cui abbiamo eseguito forward propagation:

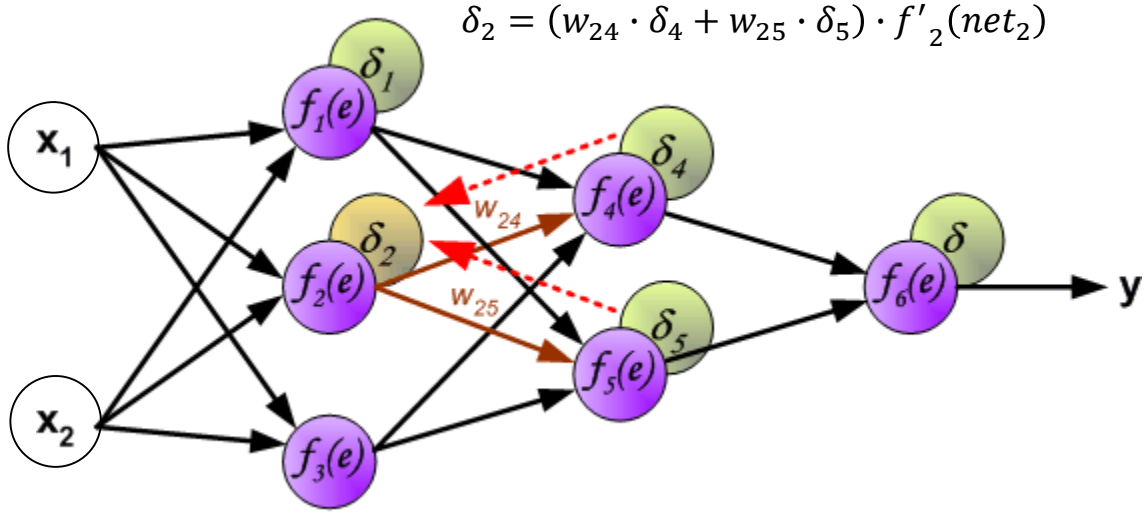


…continua

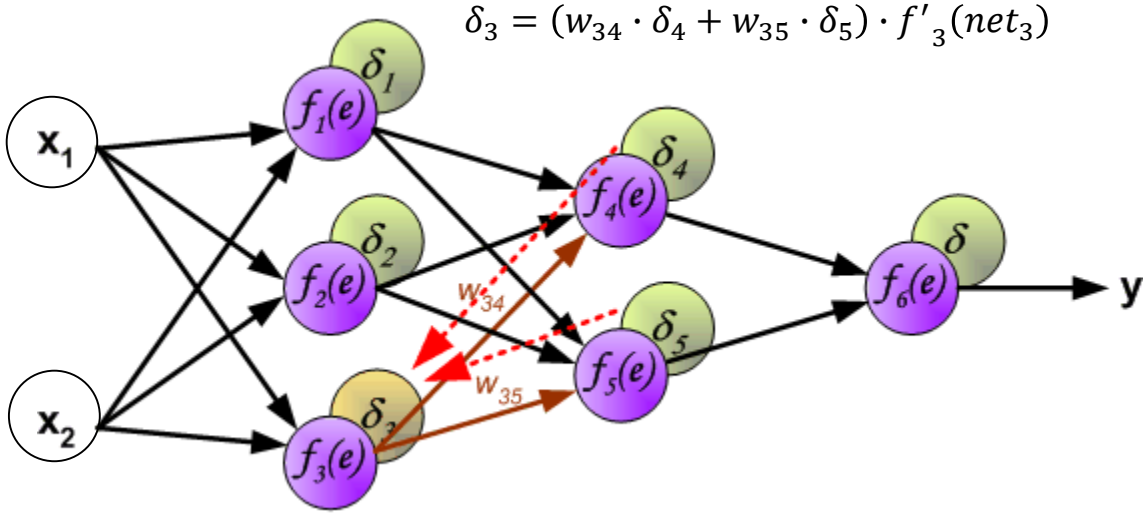
$$\delta_1 = (w_{14} \cdot \delta_4 + w_{15} \cdot \delta_5) \cdot f'_1(net_1)$$



$$\delta_2 = (w_{24} \cdot \delta_4 + w_{25} \cdot \delta_5) \cdot f'_2(net_2)$$



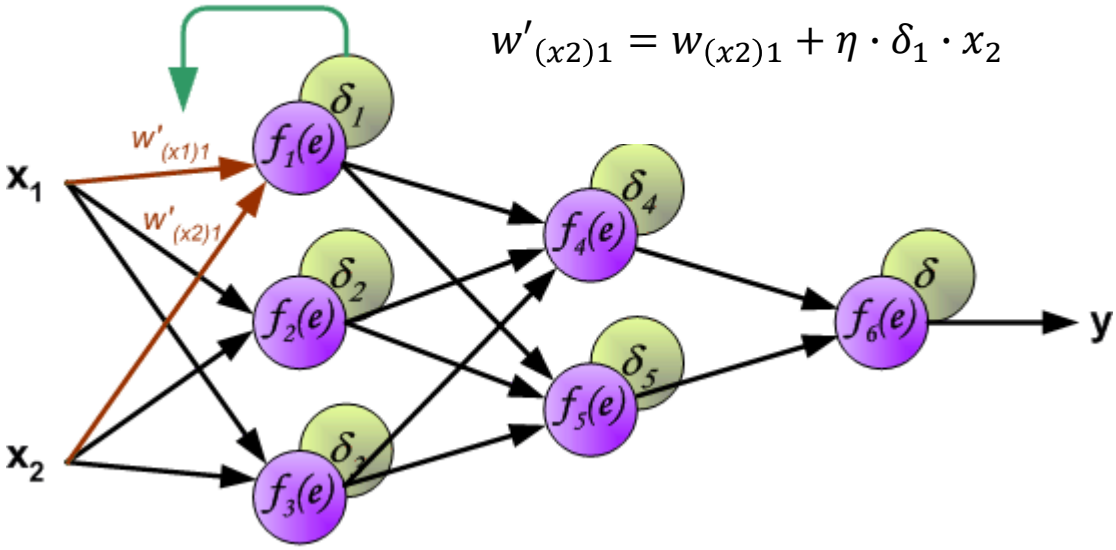
$$\delta_3 = (w_{34} \cdot \delta_4 + w_{35} \cdot \delta_5) \cdot f'_3(net_3)$$



e infine: aggiornamento pesi

$$w'_{(x1)1} = w_{(x1)1} + \eta \cdot \delta_1 \cdot x_1$$

$$w'_{(x2)1} = w_{(x2)1} + \eta \cdot \delta_1 \cdot x_2$$



…analogamente per tutti gli altri pesi

Algoritmo Backpropagation (Online)

Nella versione online i pattern del training set sono presentati sequenzialmente e i pesi sono aggiornati dopo la presentazione di ogni pattern:

Inizializza n_H, $\mathbf{w}$, η, num_{epoch}, *epoch* $\leftarrow 0$

do *epoch* $\leftarrow$ *epoch* +1

totErr $\leftarrow 0$ *// errore cumulato su tutti i pattern del TS*

for each $\mathbf{x}$ in Training Set

forward step: $\mathbf{x} \rightarrow z_k, \quad k = 1 \ldots s$ (eq. 1)

totErr $\leftarrow$ *totErr* + $J(\mathbf{w}, \mathbf{x})$

backward step: $\delta_k, \ k = 1 \ldots s$ (eq. 2), $\delta_j, \ j = 1 \ldots n_H$ (eq. 5)

aggiorna pesi hidden-output $w_{jk} = w_{jk} + \eta \cdot \delta_k \cdot y_j$ (eq. 4)

aggiorna pesi input-hidden $w_{ij} = w_{ij} + \eta \cdot \delta_j \cdot x_i$ (eq. 7)

loss $\leftarrow$ *totErr* / n *// errore medio sul TS*

Calcola accuratezza su Train Set e Validation Set

while (*not convergence and epoch* < num_{epoch})

- La convergenza si valuta monitorando l'andamento del loss e l'accuratezza sul validation set (vedi slide *Fondamenti*).
- L'approccio on-line richiede l'aggiornamento dei pesi a seguito della presentazione di ogni pattern. Problemi di efficienza (molti update di pesi) e robustezza (in caso di outliers passi in direzioni sbagliate).

Stochastic Gradient Descent (SGD)

- È l'approccio più utilizzato per implementare backpropagation:
  - a ogni epoca, gli n pattern del training set sono ordinati in modo casuale e poi suddivisi, considerandoli sequenzialmente, in gruppi (denominati mini-batch) di uguale dimensione $size_{mb}$.
    - se $size_{mb} = 1 \rightarrow$ «*stochastic*» *online*
    - se $size_{mb} = n \rightarrow$ «*full*» *batch*
  - i valori del gradiente sono (algebricamente) accumulati in variabili temporanee (una per ciascun peso); l'aggiornamento dei pesi avviene solo quando tutti i pattern di un mini-batch sono stati processati.

Inizializza n_H, $\mathbf{w}$, η, num_{epoch}, $size_{mb}$, *epoch* $\leftarrow$ 0

do *epoch* $\leftarrow$ *epoch* +1

  random sort the Training Set (n patterns)

  for each mini-batch B of $size_{mb}$

    reset gradiente

    for each $\mathbf{x}$ in B

      forward step

      backward step *// gradiente cumulato su tutti i pattern del mini-batch*

    aggiorna pesi

  Calcola Loss

  Calcola accuratezza su Training Set e Validation Set

while (*not convergence and epoch* < num_{epoch})

Terminologia (SGD)

- Attenzione alla terminologia:
  - Per epoca (epoch) si intende la presentazione (1 volta) alla rete di tutti i pattern del training set.
  - Per iterazione (iteration) si intende la presentazione (1 volta) dei pattern costituenti un mini-batch e il conseguentemente aggiornamento dei pesi.
    - $n/size_{mb}$ è il numero di iterazioni per epoca. Se non è un valore intero all'ultima iterazione si processano meno pattern.

Inizializza n_H, $\mathbf{w}$, η, num_{epoch}, $size_{mb}$, *epoch* ← 0

do *epoch* ← *epoch* +1

epoca:

random sort the Training Set (n patterns)

for each mini-batch B of $size_{mb}$

iterazione:

reset gradiente

for each $\mathbf{x}$ in B

forward step

backward step

aggiorna pesi

Calcola Loss

Calcola accuratezza su Training Set e Validation Set

while (*not convergence and epoch* < num_{epoch})

Invece di num_{epoch} e $size_{mb}$ alcuni tool (tra cui Caffe) richiedono in input il numero totale di iterazioni num_{iter} e $size_{mb}$; in questo caso il numero di epoche (non necessariamente intero) è:

$$num_{epoch} = \frac{num_{iter} \cdot size_{mb}}{n}$$

SoftMax: Output Probabilistico

- Se nel livello di output si utilizza come funzione di attivazione:
  - Tanh: i valori di uscita sono nell'intervallo [-1...1], e quindi non sono probabilità.
  - Sigmoid (standard logistic function): i valori di uscita sono compresi nell'intervallo [0…1], ma non abbiamo nessuna garanzia che la somma sui neuroni di uscita sia 1 (requisito fondamentale affinché si possano interpretare come distribuzione probabilistica).

$$\sum_{c=1\ldots s} z_c \neq 1$$

- Quando una rete neurale è utilizzata come classificatore multi-classe, l'impiego della funzione di attivazione softmax consente di trasformare i valori di net prodotti dall'ultimo livello della rete in probabilità delle classi:
  - Il livello di attivazione net_k dei singoli neuroni dell'ultimo livello si calcola nel modo consueto, ma come funzione di attivazione per il neurone k -esimo si utilizza:

$$z_k = f(net_k) = \frac{e^{net_k}}{\sum_{c=1\ldots s} e^{net_c}}$$

  - i valori z_k prodotti possono essere interpretati come probabilità delle classi: appartengono a [0…1] e la loro somma è 1.
  - l'esponenziale (utile nella combinazione con Cross Entropy Loss, vedi slide seguenti) interpreta i valori net come unnormalized log probabilities delle classi.

Cross-Entropy Loss

- Nelle slide precedenti abbiamo utilizzato per un problema di classificazione multi-classe la funzione di attivazione Tanh e la loss function Sum of Squared Error.
  - questa scelta non è ottimale, in quanto i valori di output non rappresentano probabilità, e la non imposizione del vincolo di somma a 1, rende (in genere) meno efficace l'apprendimento.

- Per un problema di classificazione multi-classe si consiglia l'utilizzo di Cross-Entropy (detta anche Multinomial Logistic Loss) come loss function:
  - La cross-entropy tra due distribuzioni discrete p e q (che fissata p misura quanto q differisce da p) è definita da:

$$H(p,q) = -\sum_{v} p(v) \cdot log\big(q(v)\big)$$

  - Nell'impiego come loss function: p (fissato) è il vettore target, mentre q il vettore di output della rete.

  - Il valore minimo di H (sempre ≥ 0) si ha quando il vettore target coincide con l'output. Attenzione nella formula il logaritmo non esiste in 0, ma gli output forniti da softmax e sigmoid non valgono mai 0 (anche se vi possono tendere asintoticamente).

  - Per approfondimenti e significato in teoria dell'Informazione: https://rdipietro.github.io/friendly-intro-to-cross-entropy-loss/

Cross-Entropy e One-hot targets

- Quando il target vector per $\mathbf{x}$ assume la forma one-hot (tutti 0 tranne un 1 per la classe corretta g):

posizione g

$$\mathbf{t} = [0, 0 \ldots 1 \ldots 0]$$

allora:

$$J(\mathbf{w}, \mathbf{x}) \equiv H(\mathbf{t}, \mathbf{z}) = -log(z_g)$$

Se z_g è ottenuta con funzione di attivazione softmax, allora:

$$J(\mathbf{w}, \mathbf{x}) = -log\left(\frac{e^{net^g}}{\sum_{c=1\ldots s} e^{net_c}}\right) = -net^g + log \sum_{c=1\ldots s} e^{net_c}$$

Per la derivazione del gradiente (backpropagation) si veda l'ottimo report "Notes on Backpropagation" di Peter Sadowski

Loss Function per Classificazione

- Classificazione multi-classe: utilizzare funzione di attivazione softmax nel livello finale e cross-entropy come loss function.
  - se i target vector $\mathbf{t}$ assumono la forma one-hot (default) si utilizza la semplificazione della slide precedente.
  - in caso di incertezza sulla classe corretta, è possibile utilizzare «soft target», con singoli valori tra 0 e 1 e somma a 1. In questo si usa formula completa di cross-entropy.

- Classificazione binaria: è possibile ricadere nel caso precedente e trattare il problema come multiclasse con 2 classi, ma risulta preferibile (spesso convergenza migliore):
  - utilizzare 1 neurone di uscita che codifica la probabilità della sola prima classe (la seconda probabilità è il complemento a 1). A tal fine si utilizza la funzione di attivazione sigmoid che produce valori in [0…1] per il primo neurone.
  - la normalizzazione a 1 della probabilità delle due classi è implicita nella semplificazione di cross-entropy adottata (~~multinomial~~ logistic regression). Si noti che con 1 neurone t, z sono valori scalari in [0...1]:

$$H(t,z) = -(t\ log(z) + (1-t)\ log(1-z))$$

- Un caso particolare è la cosiddetta classificazione multi-label dove un pattern può appartenere a più classi. Ad esempio un'immagine che contiene sia un cane che un gatto potrebbe essere classificata come appartenente a 2 classi.
  - in questo caso ogni uscita è in [0…1] ma deve essere rilassato il vincolo di somma a 1 per gli output. Si può ottenere come estensione della classificazione binaria (sopra) utilizzando attivazioni sigmoid e sommando $H(t,z)$ su più neuroni.

Loss Function per Regressione

- Consideriamo un problema di regressione dove sia la variabile indipendente (input) che quella dipendente (output) sono vettori.
  - Utilizzando la terminologia introdotta per la regressione stiamo affrontando un problema di multivariate multiple regression
  - La rete neurale è in grado di trovare un mapping non lineare tra input e output, pertanto il termine linear non si applica

- Se l'output assume valori reali non limitati si consiglia di non utilizzare funzione di attivazione nel livello finale e adottare sum of squared error come loss function. I target vector $\mathbf{t}$ non sono soggetti a vincoli:

$$\mathbf{t} = [t_1, t_2 \dots t_s]$$

Regolarizzazione

- Per ridurre il rischio di overfitting del training set da parte di una rete neurale con molti parametri (pesi), si possono utilizzare tecniche di regolarizzazione. La regolarizzazione è molto importante quando il training set non ha grandi dimensioni rispetto alla capacità del modello.
- reti neurali i cui pesi, o parte di essi, assumono valori piccoli (vicino allo zero) producono output più regolare e stabile, portando spesso a migliore generalizzazione.
- Per spingere la rete ad adottare pesi di valore piccolo si può aggiungere un termine di regolarizzazione alla loss. Ad esempio nel caso di Cross-Entropy Loss:

$$J_{Tot} = J_{Cross-Entropy} + J_{Reg}$$

  - Nella regolarizzazione L2 il termine aggiunto corrisponde alla somma dei quadrati di tutti i pesi della rete:

$$J_{Reg} = \frac{1}{2}\lambda \sum_i {w_i}^2$$

  - Nella regolarizzazione L1 si utilizza il valore assoluto:

$$J_{Reg} = \lambda \sum_i |w_i|$$

  - In entrambi i casi il parametro λ regola la forza della regolarizzazione.
  - L1 può avere un effetto sparsificante (ovvero portare numerosi pesi a 0) maggiore di L2. Infatti quando i pesi assumono valori vicini allo zero il calcolo del quadrato in L2 ha l'effetto di ridurre eccessivamente i correttivi ai pesi rendendo difficile azzerarli.

Regolarizzazione → Weight Decay

- Consideriamo la regolarizzazione L2.
  - il gradiente della J_{Tot} rispetto ad uno dei parametri della rete corrisponde al somma del gradiente di $J_{Cross-Entropy}$ e del gradiente di J_{Reg}. Quest'ultimo vale:

$$\frac{\partial J_{Reg}}{\partial w_k} = \frac{\partial}{\partial w_k}\left(\frac{1}{2}\lambda \sum_i {w_i}^2\right) = \lambda \cdot w_k$$

  - pertanto l'aggiornamento dei pesi a seguito di backpropagation include un ulteriore termine denominato weight decay (decadimento del peso) che ha l'effetto di tirarlo verso lo 0.

  - Ad esempio considerando l'aggiornamento pesi di SGD (vedi precedenti equazioni 4 e 7) :

$$w_k = w_k - \eta \cdot \frac{\partial J_{Cross-Entropy}}{\partial w_k}$$

    diventa:

$$w_k = w_k - \eta \left(\frac{\partial J_{Cross-Entropy}}{\partial w_i} + \lambda \cdot w_i\right)$$

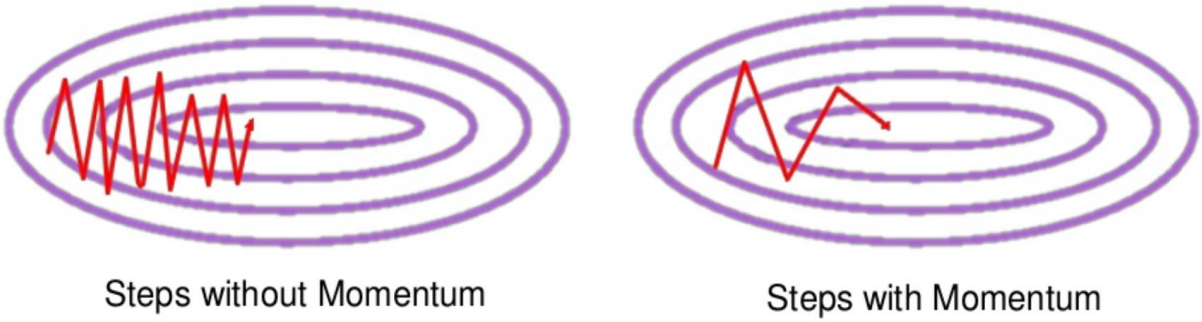
    Analogo ragionamento per la regolarizzazione L1. *Quanto vale la derivata del valore assoluto?*

Momentum

- SGD con mini-batch può determinare una discesa del gradiente a zig-zag che rallenta la convergenza e la rende meno stabile.
  - la discesa del gradiente può essere vista come la traiettoria di una biglia che rotola su una superficie scoscesa. Attratta dalla forza di gravità la biglia cerca di portarsi sul punto più basso. Nella fisica il momento conferisce alla biglia un moto privo di oscillazioni e cambi di direzione repentini.
  - in SGD per evitare oscillazioni si può emulare il comportamento fisico: si tiene traccia dell'ultimo aggiornamento Δw_k di ogni parametro w_k, e il nuovo aggiornamento è calcolato come combinazione lineare del precedente aggiornamento (che conferisce stabilità) e del gradiente attuale $\frac{\partial J}{\partial w_i}$ (che corregge la direzione):

$$\Delta w_k = \mu \cdot \Delta w_k - \eta \frac{\partial J_{Tot}}{\partial w_k}$$

$$w_k = w_k + \Delta w_k$$



Steps without Momentum

Steps with Momentum

Valore tipico del parametro μ = 0.9

Learning Rate adattativo

- Oltre a Momentum esistono altre regole per l'aggiornamento dei pesi che possono accelerare la convergenza nella discesa del gradiente (vedi http://cs231n.github.io/neural-networks-3/):
  - Nesterov Accelerate Gradient (NAG)
  - Adaptive Gradient (Adagrad)
  - Adadelta
  - Adam
  - Rmsprop

- La maggior parte di queste adatta il laerning rate globale η a ogni specifico peso. Adam è considerato lo stato dell'arte.

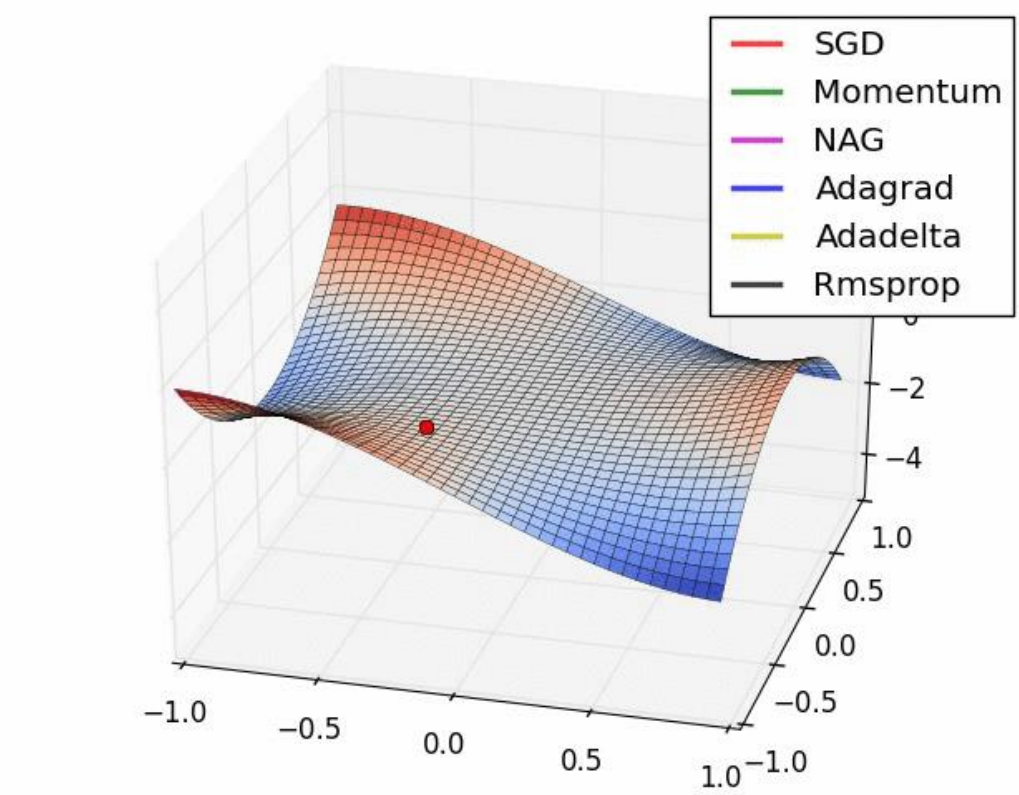


Image by Alec Radford

Tricks of the Trade

- In molti sostengono che il training di una rete neurale (complessa) sia più un'arte che una scienza.
  - Sicuramente per gestire efficacemente le molteplici scelte implementative e iperparametri: architettura, funzione di attivazione, inizializzazione, learning rate, ecc. serve esperienza (e metodo).

- Un articolo molto utile (un po' datato ma ancora attuale), che discute dal punto di vista pratico l'implementazione efficiente di backpropagation è:
  - Efficient BackProp, by Yann LeCun et al.

    http://yann.lecun.com/exdb/publis/pdf/lecun-98b.pdf

- Un articolo più recente, più specifico per reti deep, ma i cui suggerimenti sono applicabili anche a MLP è:
  - Practical Recommendations for Gradient-Based Training of Deep Architectures, by Yoshua Bengio

    https://arxiv.org/pdf/1206.5533v2.pdf

Molti concetti introdotti in queste slide possono essere sperimentati e «visivamente» compresi utilizzando il simulatore di NN disponibile al link: http://playground.tensorflow.org