COMPENDIO DI RETI NEURALI - A.A 2018/2019

A cura di Cesare Iurlaro

https://linkedin.com/in/cesare-iurlaro

cesareiurlaro@gmail.com

1 – Percettrone

- Funzione di attivazione e architettura
- *Algoritmo di apprendimento*
  - **Delta rule**
  - **Teorema di convergenza: enunciato e dimostrazione**
- Limiti del percettrone

2 – Adaline

- Funzione di attivazione e architettura
- *Algoritmo di apprendimento* ***LMS***
  - **Filtering adattivo**
  - **Funzione costo**
  - **Problema unconstraint optimization (definizione + condizione di ottimalità)**
    - **Approccio local iterative descent**
      - **Steepest Descent (Discesa del gradiente)**
        - **Stima del gradiente in *LMS***

3 – MLP

- Funzione di attivazione e architettura
- *Algoritmo di apprendimento* ***Back-Propagation***
  - **Forward/Backward phase**
  - **Funzioni costo: istantaneo e medio**
  - **Stima del gradiente in *Back-Propagation***
    - **Gradiente locale output/hidden**
- Metodi di apprendimento (epoche/pattern)
- Problemi di η e soluzioni
- Condizioni di terminazione (variazione dell'errore o generalizzazione)
- Topologia della rete (pruning/aggiunta iterativa)
- Rete come approssimatore di funzioni (booleane, continue a tratti, continue)
- **Teorema di esistenza dell'approssimazione universale**

4 – RBF

- Funzione di attivazione e architettura
  - Parametri
  - Approccio ibrido
- Teorema di Cover
- Problema dell'interpolazione
- Teorema di Micchelli
  - Funzioni per cui è valido il teorema
- *Algoritmo di apprendimento* (*determinazione dei parametri*)
  - ***Metodo della pseudoinversa* ottimizzato per calcolo parallelo e distribuito**
  - ***Hybrid Learning Process***

5 – ELM

- Funzione di attivazione e architettura
- Teoremi di ELM
- *Algoritmo di apprendimento*
  - ***Pseudoinversa***
  - ***Pseudoinversa di SVD per gestire singolarità, evitando l'inversione matriciale***

6 – CNN

- Introduzione
  - Studio delle reti neurali
  - Arrivo di AlexNet
  - Fattori del suo successo
- Funzione di attivazione e architettura
  - ***Convoluzione e livelli convoluzionali***
  - ***Pooling***
  - ***Relu e vantaggi***
- Scelte in architetture recenti
  - Soft-max layer
  - Cross-entropy loss function
- *Apprendimento*
  - **Transfer learning**
  - **Features re-using**

7 – SOM

- Architettura
- Processi di una SOM
  - **Competizione** (discriminazione del *BMU*)
  - **Cooperazione** (selezione del *vicinato topologico*)
  - **Adattamento sinaptico** (aggiornamento vettore pesi)
    - **Fasi del processo adattivo (ordinamento e convergenza)**
- *Algoritmo di apprendimento*
  - **Decadimento esponenziale iterativo di η e σ**

8 – Hopfield & Boltzmann

- Definizione di energy based model
  - **Funzione di energia globale**
  - **Energy gap**
  - **Memoria stabile e memoria fondamentale**
- Pattern completion e algoritmo non deterministico di information withdrawal
- Teorema di convergenza
- Principio di Hebb
- *Algoritmo di apprendimento*
  - ***Limitazioni***
  - ***Boltzmann: Simulated annealing***
- *RBM*
  - ***Architettura***
  - ***Algoritmo di apprendimento***

- *Esempi di calcolo*

0 – Introduzione alle Reti Neurali (Neural Networks)

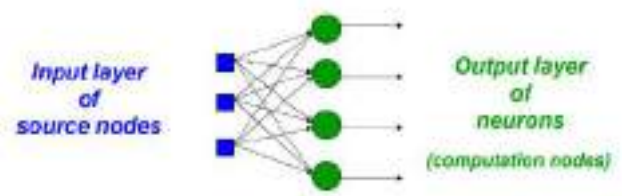
Elemento computazionale massicciamente distribuito e parallelo che memorizza in un vettore pesi conoscenza sperimentale.

Allenamento: processo di miglioramento delle prestazioni, realizzato sottoponendo la rete ad esempi (conoscenza sperimentale).

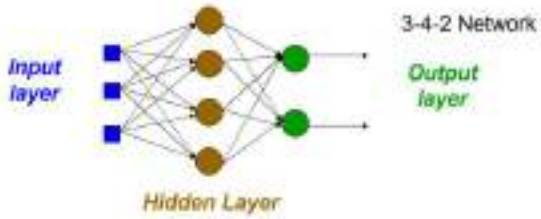
Benefici:
- Non-linearity
- Input-Output mapping realizzato attraverso l'apprendimento.
- Fault tolerance: le prestazioni degradano gradualmente.
- Incremental Training: è possibile continuare l'apprendimento se sono disponibili nuovi dati.

Componenti di una *Neural Network*:
- Architettura: insieme di neuroni collegamenti tramite link. A ciascun link è associato un peso.
- Neurone: unità computazionale dell'*architettura.*
- Algoritmo di apprendimento: algoritmo usato per allenare la *Neural Network*.

Classi di architetture:
- Feed Forward (Single-Layer o Multi-Layer): i neuroni sono organizzati a livelli in un grafo aciclico



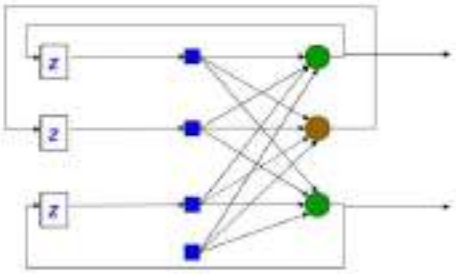
Single-Layer Feed Forward



Fully Connected Multi-Layer Feed Forward

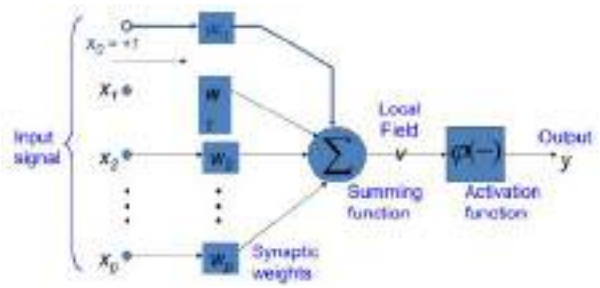
La presenza di *Hidden Layer* permette di estrarre dinamiche di più alto ordine proiettando i dati in spazi differenti.
Ciò assume particolare importanza all'aumentare dei nodi al *livello di input*.

- Recurrent: architettura caratterizzata da ***unit delay operators,*** componenti che controllano dinamicamente il sistema. Rendono possibili dinamiche temporali.



Recurrent Neural Network

Architettura del neurone:



- **Link:** caratterizzano l'input del neurone associato a un peso w_i.
- **Combinazione lineare u_j:** funzione che computa la somma degli input pesata in funzione dei pesi.

$$u_j = \sum_{i=1}^{p} w_{ji} \cdot x_i$$

- **Bias $w_{j0} = b_j$:** intercetta che applica una trasformazione alla combinazione lineare u_j: $v_j = u_j - b_j$.

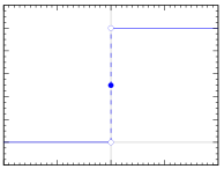
- **Funzione di attivazione:** funzione che limita il l'intervallo dei valori possibili che l'output può assumere.

$$y_j = \varphi(u_j - b_j) = \varphi(v_j)$$

La scelta della funzione φ (*step*, *ramp*, *sigmoid* o *gaussian*) caratterizza il *modello del neurone*.

1 – Percettrone

Funzione φ a gradino

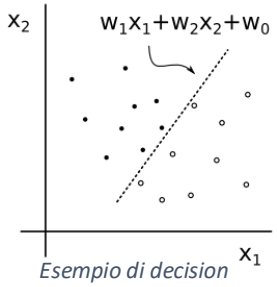


Il più semplice *modello* di *Rete Neurale*; classifica esempi in uno spazio linearmente separabile.

Induced local field $v_j = \boldsymbol{w}^T \boldsymbol{x}$: *combinazione lineare* del **vettore pesi** $\boldsymbol{w}$ e delle p componenti di $\boldsymbol{x}$.
Il vettore $\boldsymbol{x}$ è detto *esempio*, *istanza* o *pattern input*.

Il percettrone è caratterizzato dalla funzione di attivazione *a gradino*: $\varphi(v_j) = \begin{cases} = 1 & v_j > 0 \\ = -1 & v_j < 0 \end{cases}$

Dati il vettore pesi $\boldsymbol{w}$, è possibile visualizzare il **decision boundary** (ovvero l'iperpiano che divide le istanze in funzione delle classi di appartenenza) tracciato sullo spazio delle istanze.



Esempio di decision boundary in due dimensioni

Apprendimento del percettrone – Supervisionato

Sia $\boldsymbol{w}(t)$ il vettore pesi $\boldsymbol{w}$ all'iterazione t ed η il *learning rate* (*coefficiente di apprendimento*).
Sia inoltre la seguente la **regola di aggiornamento**: $\boldsymbol{w}(k+1) = \boldsymbol{w}(k) + \eta\, \boldsymbol{x}(k)\, d(k)$

$$\Longrightarrow \Delta \boldsymbol{w} = \boldsymbol{w}(k+1) - \boldsymbol{w}(k) = \eta\, \boldsymbol{x}(k)\, d(k)$$

L'algoritmo di apprendimento del percettrone modifica iterativamente $\boldsymbol{w}$ nel seguente modo:

1) Inizializza $\boldsymbol{w}$ con valori casuali
2) *While* (la presente iterazione n presenta esempi erroneamente classificati) {
   1. Seleziona un esempio classificato male
   2. Aggiorna i pesi con la *regola di aggiornamento*

}

Osservazione: L'algoritmo converge quando la condizione di terminazione viene rispettata, ovvero quando $\boldsymbol{w}$ è tale per cui il decision boundary classifichi correttamente **tutti** gli esempi.

Teorema di convergenza: Siano le classi C_1, C_2 linearmente separabili; allora l'algoritmo di apprendimento del percettrone converge in un numero finito di iterazioni, ovvero esiste un iperpiano che separi correttamente C_1 e C_2.

Dimostrazione:
Assumiamo $\eta = 1$ e $\boldsymbol{w}(0) = 0$.
Sia C_1l'insieme delle istanze per cui $d_{C_1} = 1$ e C_2 l'insieme di istanze per cui $d_{C_2} = -1$. $d(\boldsymbol{x})$: *desired output dell'istanza* $\boldsymbol{x}$.
Sia C l'insieme ottenuto dall'unione $C_1 \cup C_2$.

1) **Passo preliminare di correzione:** $\forall \boldsymbol{x}(i) \in C_2$ applichiamo la trasformazione $\boldsymbol{x}(i) = -\boldsymbol{x}(i)$, cioè tale che $\forall n, d(n) = 1$
2) **Definizione del problema di ricerca:** trovare un vettore pesi $\boldsymbol{w}_*$tale che $\forall \boldsymbol{x}(i) \in C$, $\boldsymbol{w}_*^T \boldsymbol{x}(i) > \boldsymbol{0}$, cioè tale che $\forall \boldsymbol{x}(i)$, $\boldsymbol{y}(i) = 1$
3) Sia k la k-esima iterazione. Poiché:
   a. $\forall n, d(n) = 1$
   b. $\eta = 1$

   Allora per la regola di aggiornamento $\boldsymbol{w}(k) + \eta\, \boldsymbol{x}(k)\, d(k)$, vale che:

$$\forall k : \ \boldsymbol{w}(k+1) = \boldsymbol{w}(k) + \boldsymbol{x}(k) \ = \ \boldsymbol{x}(1) + \boldsymbol{x}(2) + \ \dots + \boldsymbol{x}(k)$$

A) **Determinazione del *lower bound*;** sia $\alpha = min\|\boldsymbol{x}(i)\|$ l'istanza con il modulo minore.

$$\boldsymbol{w}(k+1) = \boldsymbol{x}(1) + \boldsymbol{x}(2) + \ \dots + \boldsymbol{x}(k)$$
$$\boldsymbol{w}_*^T \cdot \boldsymbol{w}(k+1) = \boldsymbol{w}_*^T \cdot [\boldsymbol{x}(1) + \boldsymbol{x}(2) + \ \dots + \boldsymbol{x}(k)]$$
$$\boldsymbol{w}_*^T \cdot \boldsymbol{w}(k+1) \geq k\alpha$$
$$\left\|\boldsymbol{w}_*^T \cdot \boldsymbol{w}(k+1)\right\|^2 \geq (k\alpha)^2$$

Disuguaglianza di Cauchy-Shwartz: $|(u,v)| \leq \|u\| \cdot \|v\|$ $\Longrightarrow \left\|\boldsymbol{w}_*^T\right\|^2 \cdot \|\boldsymbol{w}(k+1)\|^2 \ \geq \left\|\boldsymbol{w}_*^T \cdot \boldsymbol{w}(k+1)\right\|^2 \geq (k\alpha)^2$

$$\|\boldsymbol{w}(k+1)\|^2 \geq \left({}^{k\alpha}/_{\|\boldsymbol{w}_*^T\|}\right)^2$$

B) **Determinazione dell'*upper bound*;** sia $\beta = max\|\boldsymbol{x}(i)\|^2$ l'istanza con il modulo maggiore.

$$\boldsymbol{w}(k+1) = \boldsymbol{w}(k) + \boldsymbol{x}(k)$$
$$\|\boldsymbol{w}(k+1)\|^2 = \|\boldsymbol{w}(k) + \boldsymbol{x}(k)\|^2$$
$$\|\boldsymbol{w}(k+1)\|^2 = \|\boldsymbol{w}(k)\|^2 + \|\boldsymbol{x}(k)\|^2 + 2 \cdot \boldsymbol{w}^T(k) \cdot \boldsymbol{x}(k)$$

$\boldsymbol{w}(k+1) \Longrightarrow \boldsymbol{x}(k)$ non classificato correttamente $\Longrightarrow 2 \cdot \boldsymbol{w}^T(k) \cdot \boldsymbol{x}(k) \leq 0$
$\Longrightarrow \|\boldsymbol{w}(k+1)\|^2 \leq \|\boldsymbol{w}(k)\|^2 + \|\boldsymbol{x}(k)\|^2 = \sum_{i=0}^{k} \|\boldsymbol{x}(i)\|^2 \Longrightarrow \ \|\boldsymbol{w}(k+1)\|^2 \leq \sum_{i=0}^{k} \|\boldsymbol{x}(i)\|^2$

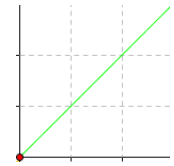
$$\|\boldsymbol{w}(k+1)\|^2 \leq k\beta$$

---

4) Da $\boldsymbol{A}$ e $\boldsymbol{B}$ segue che: $\left({}^{k\alpha}/_{\|\boldsymbol{w}_*^T\|}\right)^2 \leq \|\boldsymbol{w}(k+1)\|^2 \leq k\beta$, ovvero che l'algoritmo termina in un numero finito di iterazioni.

Limite del percettrone: non risolve problemi non linearmente separabili (es: *problema dello XOR*)

2 – Adaline: Adaptive Linear Element


Funzione φ ramp

Modello di Rete Neurale caratterizzato dalla funzione di attivazione lineare *ramp* $y = \varphi(v)$.

Utilizza l'algoritmo di apprendimento supervisionato con *filtering adattivo* ***LMS*** (***Least Mean Square***).

Filtering process: computazione dei segnali di *output* $y(i)$ e di *errore* $e(i)$, dato un vettore *input* $\boldsymbol{x}_i$.
Adaptive process: aggiustamento automatico del vettore pesi $\boldsymbol{w}_i$ in funzione dell'*errore* $e(i)$.

Problema unconstraint-optimization*:* calcolare $\boldsymbol{w}^* = argmin_{\boldsymbol{w}}\, E(\boldsymbol{w})$. $E(\boldsymbol{w})$: *funzione costo* che misura la bontà di un vettore $\boldsymbol{w}$

Condizione necessaria per l'ottimalità: $\nabla E(\boldsymbol{w}^*) = \boldsymbol{0}$ (notare che $\boldsymbol{0}$ è un *vettore di zeri*)

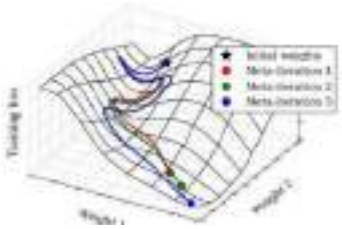
- ∇ è l'**operatore gradiente**
- $\nabla E(\boldsymbol{w}) = \left[\frac{\partial E}{\partial \boldsymbol{w}_1}, \ \dots\ , \frac{\partial E}{\partial \boldsymbol{w}_m}\right]$ è il **gradiente** (o **vettore gradiente**) della funzione costo

Gradiente di una funzione: vettore le cui componenti sono le derivate parziali della funzione rispetto agli assi di riferimento.
- La **direzione** del gradiente identifica la *direzione più ripida* della funzione.

Ciascuna componente j del gradiente definisce due informazioni:
1. La ***magnitudine*** con cui ciascun peso $\boldsymbol{w}_j$ influenza l'errore $E(\boldsymbol{w})$.
2. Il ***segno*** necessario a $\boldsymbol{w}_j$ per ridurre l'errore (se $\boldsymbol{w}_j$ è più grande o piccolo di come dovrebbe essere).

Local iterative descent: approccio per problemi di *unconstraint-optimization.*
Consiste nella *riduzione iterativa* della *funzione costo*. Ad ogni k-esima iterazione, vale che: $E\big(\boldsymbol{w}(k+1)\big) < E\big(\boldsymbol{w}(k)\big)$.

- ***Metodo Steepest Descent (discesa del gradiente)*:** tecnica di ottimizzazione con approccio di *local iterative descent* che iterativamente modifica il vettore pesi $\boldsymbol{w}(k)$ al fine di massimizzare la riduzione dell'errore in esso presente.


Esempio di discesa del gradiente

La *regola di aggiornamento* di questo metodo prende il nome di **steepest descent delta rule**:

$$\boldsymbol{w}(k+1) = \boldsymbol{w}(k) - \eta\big(\nabla E(\boldsymbol{w})\big)$$
$$\Longrightarrow \Delta\boldsymbol{w} = \boldsymbol{w}(k+1) - \boldsymbol{w}(k) = -\eta\big(\nabla E(\boldsymbol{w})\big)$$

Ossia, ad ogni iterazione, $\boldsymbol{w}$ viene aggiornato in funzione del gradiente della funzione costo (stessa direzione e verso opposto) ponderato su $\eta \in \mathbb{R}$, che è il tasso di apprendimento.

› **Algoritmo LMS (1960):** algoritmo che effettua *filtering adattivo lineare*. Usa il *metodo steepest descent* per **minimizzare il valore** della funzione costo quadratico *istantaneo* $E(\boldsymbol{w}) = \frac{1}{2}e^2(k)$, attraverso la ***stima*** del vettore gradiente *istantaneo* $\widehat{\boldsymbol{g}}(k)$:

Notare che l'algoritmo minimizza l'errore non attraverso il calcolo del gradiente effettivo, ma di una sua stima!
Pertanto anche il vettore pesi calcolato è una stima, e viene indicato come $\widehat{\boldsymbol{w}}$.

$$E(\widehat{\boldsymbol{w}}) = \frac{1}{2}e^2(k) \Longrightarrow \frac{\partial E(\widehat{\boldsymbol{w}})}{\partial e(k)} = \frac{\partial\left(\frac{1}{2}e^2(k)\right)}{\partial e(k)} = e(k)$$

$$e(k) = d(k) - \widehat{\boldsymbol{w}}^{\boldsymbol{T}}(k)\boldsymbol{x}(k) \Longrightarrow \frac{\partial e(k)}{\partial \widehat{\boldsymbol{w}}(k)} = -\boldsymbol{x}(k)$$

$$\widehat{\boldsymbol{g}}(k) = \frac{\partial E(\widehat{\boldsymbol{w}})}{\partial \widehat{\boldsymbol{w}}} = \frac{\partial E(\widehat{\boldsymbol{w}})}{\partial e(k)} \cdot \frac{\partial e(k)}{\partial \widehat{\boldsymbol{w}}} = -\boldsymbol{x}(k)e(k)$$

Dalla *steepest descent delta rule* segue la LMS delta rule:

$$\widehat{\boldsymbol{w}}(k+1) = \widehat{\boldsymbol{w}}(k) + \eta\, \boldsymbol{x}(k)\, e(k)$$
$$\Longrightarrow \Delta\widehat{\boldsymbol{w}} = \widehat{\boldsymbol{w}}(k+1) - \widehat{\boldsymbol{w}}(k) = \eta\, \boldsymbol{x}(k)\, e(k)$$

Algoritmo LMS:

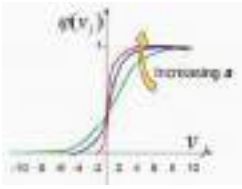
1) Inizializza casualmente $\boldsymbol{w}(k)$.
2) *While* (E_{tot} insoddisfacente $\wedge\ k <$ max_iterations) {
   1. *Itera su un esempio* k : $\big(x^{(k)}, d^{(k)}\big)$
   2. *Computa l'****errore istantaneo*** k*-esimo* : $\boldsymbol{e}^{(k)} = d(k) - \boldsymbol{x}^{\boldsymbol{T}}(k)\, \widehat{\boldsymbol{w}}(k)$
   3. *Aggiorna i pesi con la* ***LMS Delta rule***: $\boldsymbol{w}(k+1) = \boldsymbol{w}(k) + \eta\, \boldsymbol{x}(k)\, \boldsymbol{e}^{(k)}$

   }

3 – MLP: Multilayer Perceptron

La struttura di ***MLP*** è definita tramite una sequenza di *layer* di *percettroni* e gode delle seguenti caratteristiche:

- Il modello di ciascun neurone nella rete include una funzione di attivazione *non lineare* e *differenziabile*, tipicamente la funzione di attivazione *sigmoide* $\varphi(v) = \frac{1}{1+e^{-av}}$, dove $v = \boldsymbol{w}_{ji}\,\boldsymbol{\varphi}$.
- **Uno o più** *hidden layer*: nodi che non sono né di input né di *output*.
- Esibisce un alto livello di connettività.



Funzione φ sigmoide

Gli *hidden layer* effettuano una ***trasformazione non lineare*** dei dati in input in uno spazio chiamato *spazio delle feature.*

Rete neurale feed-forward ($\boldsymbol{FFNN}$)**:** modello di rete neurale dove le connessioni tra le unità non formano cicli.

Tipicamente l'algoritmo di *apprendimento supervisionato* in una *rete* di questo tipo è l'***algoritmo di back-propagation (1985),*** che in particolari degenera nell'*algoritmo LMS.* Tale algoritmo consiste in due iterazioni:

1) **Forward phase:** fissati i pesi della rete, il segnale di input si propaga attraverso la rete fino all'output.
2) **Backward phase:** l'output della rete viene comparato con la risposta attesa e ne viene computato l'errore. Tale errore viene propagato all'indietro per la rete e ne modifica i pesi.

- $\mathcal{T} = \{\boldsymbol{x}(k),\ \boldsymbol{d}(k)\}_{k=1}^{N}$ il ***training set*** di N *istanze,* etichettato*.*
- $y_j(k)$ l'***output*** del ***j-esimo neurone nell'output layer*** allo stimolo dell'istanza $\boldsymbol{x}(k)$.
- $e_j(k) = d_j(k) - y_j(k)$ l'***errore*** del ***j-esimo neurone nell'output layer*** allo stimolo dell'istanza $\boldsymbol{x}(k)$.

- $E(k) = \frac{1}{2}\sum_j e_j^2\,(k)$ l'***errore totale istantaneo*** dell'intero *output layer* allo stimolo dell'istanza $\boldsymbol{x}(k)$.
- $E_{av}(N) = \frac{1}{N}\sum_{k=1}^{N} E(k)$ l'***errore medio sul training set*** (o ***rischio empirico***)

Osservazione: $E(k)$ è la somma, per ogni *neurone output*, della funzione costo che era presente in *Adeline*.

Stima del gradiente nell'algoritmo di back-propagation:

$$\frac{\partial E(\widehat{\boldsymbol{w}})}{\partial e_j(k)} = \boldsymbol{e}_j \qquad \frac{\partial e_j(k)}{\partial \boldsymbol{y}_j(k)} = -1$$

$$\frac{\partial \boldsymbol{y}_j(k)}{\partial \boldsymbol{v}_j(k)} = \boldsymbol{\varphi}'_j(v_j) \qquad \frac{\partial \boldsymbol{v}_j(k)}{\partial \widehat{\boldsymbol{w}}_{ji}(k)} = \boldsymbol{y}_i$$

A seconda della tipologia di *layer* a cui appartengono i neuroni (*hidden* oppure *output*), il *gradiente locale* assume forme diverse. Questo è vero poiché agli *output layer* sono associate informazioni che non lo sono agli *hidden layer*: i *desired output* $d(n)$.

- **Gradiente computato nei neuroni *output*:**

$$\widehat{\boldsymbol{g}}_{Out}(k) = \frac{\partial E}{\partial \widehat{\boldsymbol{w}}_{ji}} = \frac{\partial E}{\partial e_j}\cdot\frac{\partial e_j}{\partial y_j}\cdot\frac{\partial y_j}{\partial v_j}\cdot\frac{\partial v_j}{\partial \widehat{\boldsymbol{w}}_{ji}} = -\boldsymbol{e}_j\,\boldsymbol{\varphi}'_j(v_j)\,\boldsymbol{y}_i$$

Dalla *steepest descent delta rule* segue la ***Back-Propagation*** delta rule (per i neuroni di ***output***):

$$\widehat{\boldsymbol{w}}_{ji}(k+1) = \widehat{\boldsymbol{w}}_{ji}(k) + \eta\,\boldsymbol{e}_j\,\boldsymbol{\varphi}'_j(v_j)\,\boldsymbol{y}_i$$

Il fattore $\boldsymbol{e}_j\,\boldsymbol{\varphi}'_j(v_j)$ prende il nome di ***gradiente locale*** (o segnale d'errore) del j-esimo *neurone di* ***output*** *;* si indica con $\boldsymbol{\delta}_j$:

$$\Longrightarrow \Delta\boldsymbol{w}_{ji} = \boldsymbol{w}_{ji}(k+1) - \boldsymbol{w}_{ji}(k) = \eta\,\boldsymbol{\delta}_j\,\boldsymbol{y}_i$$

- **Gradiente computato nei neuroni *hidden*:**

Q : $insieme\ dei\ neuroni\ del\ livello\ successivo;\ q \in Q$

$$\frac{\partial E}{\partial y_j} = \sum_q e_q\cdot\frac{\partial e_q}{\partial y_j} = \sum_q e_q\cdot\frac{\partial e_q}{\partial y_q}\cdot\frac{\partial y_q}{\partial v_q}\cdot\frac{\partial v_q}{\partial y_j} = \sum_q e_q\cdot\left[-\mathbf{1}\cdot\boldsymbol{\varphi}'_q(v_q)\right]\cdot\widehat{\boldsymbol{w}}_{qj} = -\sum_q e_q\,\boldsymbol{\varphi}'_q(v_q)\,\widehat{\boldsymbol{w}}_{qj} \overset{\text{def}}{=} -\sum_q \boldsymbol{\delta}_q\,\widehat{\boldsymbol{w}}_{qj}$$

$$\widehat{\boldsymbol{g}}_{Hid}(k) = \frac{\partial E}{\partial \widehat{\boldsymbol{w}}_{ji}} = \frac{\partial E}{\partial y_j}\cdot\frac{\partial y_j}{\partial v_j}\cdot\frac{\partial v_j}{\partial \widehat{\boldsymbol{w}}_{ji}} = \left[-\sum_q \boldsymbol{\delta}_q\,\widehat{\boldsymbol{w}}_{qj}\right]\cdot\boldsymbol{\varphi}'_j(v_j)\cdot y_i$$

Dalla *steepest descent delta rule* segue la ***Back-Propagation*** delta rule (per i neuroni ***hidden***):

$$\widehat{\boldsymbol{w}}_{ji}(k+1) = \widehat{\boldsymbol{w}}_{ji}(k) + \eta\left(\sum_q \boldsymbol{\delta}_q\,\widehat{\boldsymbol{w}}_{qj}\right)\cdot\boldsymbol{\varphi}'_j(v_j)\cdot\boldsymbol{y}_i$$

Il fattore $\left(\sum_q \boldsymbol{\delta}_q\,\widehat{\boldsymbol{w}}_{qj}\right)\cdot\boldsymbol{\varphi}'_j(v_j)$ prende il nome di ***gradiente locale*** del j-esimo *neurone* ***hidden*** *;* si indica con $\boldsymbol{\delta}_j$:

$$\Longrightarrow \Delta\boldsymbol{w}_{ji} = \boldsymbol{w}_{ji}(k+1) - \boldsymbol{w}_{ji}(k) = \eta\,\boldsymbol{\delta}_j\,\boldsymbol{y}_i$$

Metodi apprendimento:

- Per epoche (batch): i pesi vengono modificati dopo la presentazione (*non ordinata*) di tutti gli N esempi nel *training set* $\mathcal{T}$. Un'intera presentazione prende il nome di ***epoca di allenamento***. In altre parole, la *funzione costo* è definita da $E_{av}(N)$.
- Per pattern (online): i pesi vengono modificati dopo la presentazione *ordinata* di ciascun esempio nel *training set* $\mathcal{T}$. La *funzione costo* è definita da $E(k)$.

Problema: η piccolo implica apprendimenti lenti, η grande implica cambiamenti più consistenti ma potrebbe causare comportamenti instabili con oscillazioni del vettore pesi se > 1.

Tecniche per aggirare il problema:

1. Introdurre un ***termine momentum*** $\alpha\Delta\boldsymbol{w}_{ji}(k-1)$ nella regola di aggiornamento dei pesi che prenda in considerazione gli aggiornamenti precedenti. Tale nuova regola prende il nome di ***Delta rule generalizzata:***

$$\Delta\boldsymbol{w}_{ji}(k) = \alpha\Delta\boldsymbol{w}_{ij}(k-1) + \eta\,\boldsymbol{\delta}_j\,\boldsymbol{y}_i\,; \quad 0 \leq \alpha < 1$$

   Il *termine momentum* ha l'effetto di accelerare l'apprendimento in presenza di discesa costante e di stabilizzarlo in presenza di oscillazioni.

2. ***Aggiustamento adattivo di η*** attraverso l'uso di euristiche:
   a. *Euristica 1*: assegnare un η diverso a ciascun peso
   b. *Euristica 2*: variare η da un'iterazione a lla successiva

L'apprendimento continua finché la condizione di terminazione non è soddisfatta.

Possibili condizioni di terminazione:

- *Variazioni dell'errore quadratico medio totale:*
  L'algoritmo converge se il tasso assoluto di cambiamento di $E_{av}(N)$ per epoca ricade nell'intervallo [0.1, 0.01] (suff. piccolo).

- *Criteri basati sulla generalizzazione:*
  L'algoritmo converge se le *performance di generalizzazione* sono adeguate.

Algoritmo di back-propagation:

1) Inizializza il vettore $\boldsymbol{w}(k)$ casualmente
2) Presenta gli esempi di training (per epoca o per pattern)
3) Computa in avanti, ottenendo l'output y_j e l'errore e_j per ogni j-esimo *neurone di output*
4) Computa all'indietro per aggiornare i pesi (differentemente a seconda che i neuroni siano *hidden* o *output*)
5) Torna a (2), se la *condizione di terminazione* non è rispettata

Caratteristiche di una Rete Neurale:

- *Rappresentazione dei dati* (dominio delle feature)
  Dipende dal problema, generalmente valori continui (reali), potenzialmente racchiusi in un intervallo limitato.
  La **normalizzazione** è una trasformazione tale per cui le *feature* delle istanze del *training set* siano comprese nell'intervallo [0, 1].
  $$\boldsymbol{x}_i = \frac{\boldsymbol{x}_i - min_i}{max_i - min_i}$$
- *Topologia della rete* (numero di livelli e di neuroni)
  Dipende dal *task.* Nella pratica viene decisa tramite *trial and error*.

  Per determinarla possono essere usati due tipi di algoritmi adattivi:
  - **Pruning**: se le prestazioni sono soddisfacenti, si rimuovono neuroni e link finché continuano ad esserlo
  - **Aggiunta di neuroni e link:** se le prestazioni non sono soddisfacenti, finché non lo diventano

- *Parametri della rete* (pesi, learning rate, numero di esempi del *training set,* ecc.)
  - In generale, i **pesi iniziali** sono casuali; tipicamente con valori compresi negli intervalli $[-1, 1]$ e $[-0{,}5, 0.5]$.
  - Il valore di η dipende dall'applicazione. Tipicamente nell'intervallo $[0.1, 0.9]$ oppure adattato durante l'allenamento.

- *Apprendimento:*
  - **"Regola del pollice"**: il numero di *istanze di training* dovrebbe essere da 4 a 10 volte il numero di pesi nella rete.
  - **Altra regola**: $N > \frac{|W|}{(1-a)}$, dove $|W|$ è il numero di pesi e a è l'*accuratezza* attesa del test set.

- *Approssimazione:*
  - **Funzioni booleane:** rappresentabili in un singolo *hidden layer.*
  - **Funzioni continue limitate a tratti:** approssimabili con un errore arbitrariamente piccolo in un singolo *hidden layer.*
  - **Funzioni continue:** approssimabili con un errore arbitrariamente piccolo in due *hidden layer.*

Teorema di esistenza dell'approssimazione universale:
Sia $\varphi(\cdot)$ una funzione continua non costante, limitata e monotona crescente.
Sia I_{m_0} un ipercubo unitario $[0, 1]^{m_0}$ m_0-dimensionale.

Allora, data una funzione $f \in \boldsymbol{C}(I_{m_0})$ e un $\varepsilon > 0$, esistono un intero m_1 e un insieme di costanti reali $\boldsymbol{\alpha}_i$, $\boldsymbol{b}_i$ *e* $\boldsymbol{w}_{ij}$ tali che:

$$F(x_1, \dots x_{m_0}) = \sum_{i=1}^{m_1} \boldsymbol{\alpha}_i \varphi\left(\sum_{j=1}^{m_0} \boldsymbol{w}_{ij}\boldsymbol{x}_j + \boldsymbol{b}_i\right)$$

$$i.e. \qquad \left|F(x_1, \dots x_{m_0}) - f(x_1, \dots x_{m_0})\right| < \varepsilon$$

$F(x_1, \dots x_{m_0})$ rappresenta l'*output* dell'*MLP* con:
- m_0 *nodi input*
- m_1 *nodi hidden*
- w_{ij} pesi sinaptici per *nodi hidden*
- b_i bias per *nodi hidden*
- α_i pesi sinaptici per *nodi output*

Osservazioni sul teorema: il teorema afferma che un singolo *neurone hidden è sufficiente* ad una *MLP* per computare un'approssimazione uniforme per un *training set* $\{\boldsymbol{x_1}, \dots \boldsymbol{x_{m_0}}\}$.

Nel 1993, Barron ha stabilito la proprietà di approssimazione di *MLP*, valutando il **tasso di decrescita dell'errore** $\boldsymbol{O(1/m_1)}$.

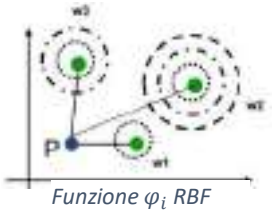
Le *FNN* possono essere applicate per risolvere problemi non linearmente separabili, per apprendere funzioni non lineari (regressione) e in particolare funzioni il cui input è una sequenza di misurazioni nel tempo (time series).

4 – RBF: Radial-Basis Function Networks

Modello di RN caratterizzato dalla *funzione di attivazione lineare* $\varphi(\boldsymbol{v}) = \boldsymbol{w}_1\boldsymbol{\varphi}_1(\|\boldsymbol{x} - \boldsymbol{t}_1\|) + \cdots + \boldsymbol{w}_{m1}\boldsymbol{\varphi}_{m1}(\|\boldsymbol{x} - \boldsymbol{t}_{m1}\|)$, che è la *combinazione lineare di* m *funzioni di base radiale* $\boldsymbol{\varphi}_i$ e dal fatto di avere **un solo** *hidden layer*.
Le *Reti* ***RBF*** sono utilizzate per *task di regressione* e *classificazione complessa* (non *lineare*).

Una *rete RBF* implementa l'approccio, differente dall'*approssimazione stocastica* delle reti *MLP*, chiamato **approccio ibrido:**

1) **Trasforma** un insieme di pattern non linearmente separabili in modo da renderli più probabilmente linearmente separabili.
2) **Classifica** l'insieme dei pattern trasformati.



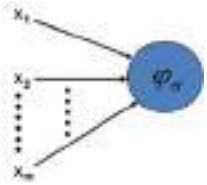
Funzione φ_i *RBF*

$\|\boldsymbol{x} - \boldsymbol{t}_i\|$ è la distanza del vettore $\boldsymbol{x}$ dal vettore $\boldsymbol{t}_i$

- I neuroni dell'*hidden layer*, usando *una funzione RBF,* applicano una trasformazione non lineare dall'*input space* all'***hidden*** (**o *feature***) ***space*** (il quale tipicamente è molto dimensionale).
  In una funzione **di base *radiale* (*RBF*)**, l'*output* è in funzione della distanza dal *vettore prefissato* $\boldsymbol{t}_i$.
- Il neurone dell'*output layer* **combina linearmente** l'*output* dei neuroni dell'*hidden layer precedente* e il peso di ciascuno di essi.

Teorema di Cover della separabilità dei patter: il numero di neuroni dell'*hidden layer* e la probabilità che il problema sia linearmente separabile nel *feature space*, sono direttamente proporzionali.

Modello neurone hidden:



$\boldsymbol{\varphi}_\sigma(\|\boldsymbol{x} - \boldsymbol{t}\|)$, dove $\boldsymbol{t}$ e σ sono parametri:
$\boldsymbol{t}$ è il parametro *centro*
σ è il parametro *spread* (*ampiezza*)

Se la funzione è gaussiana, la *sensibilità* del neurone può essere manipolata attraverso la variazione del parametro σ, il cui valore è inversamente proporzionale alla sensibilità.

Problema dell'interpolazione:
Dato un *dataset* di N diversi punti $\{\boldsymbol{x}_i \in \mathbb{R}^m, i = 1 \cdots N\}$ e un vettore ***desired output*** **(target)** $\boldsymbol{d}$ con N componenti reali $\{\boldsymbol{d}_i \in \mathbb{R}, i = 1 \cdots N\}$, trovare una funzione $f : \mathbb{R}^m \to \mathbb{R}$ che soddisfi la **condizione di interpolazione:** $F(\boldsymbol{x}_i) = \boldsymbol{d}_i$

Se $F(x) = \sum_{i=1}^{N} \boldsymbol{w}_i\boldsymbol{\varphi}(\|\boldsymbol{x} - \boldsymbol{x}_i\|)$, allora:

$$\begin{bmatrix} \boldsymbol{\varphi}_1(\|x_1 - t_1\|) & \cdots & \boldsymbol{\varphi}_{m1}(\|x_1 - t_{m1}\|) \\ \vdots & & \vdots \\ \boldsymbol{\varphi}_1(\|x_N - t_1\|) & \cdots & \boldsymbol{\varphi}_{m1(\|x_N - t_{m1}\|)} \end{bmatrix} \begin{bmatrix} \boldsymbol{w}_1 \\ \cdots \\ \boldsymbol{w}_{m1} \end{bmatrix} = \begin{bmatrix} \boldsymbol{d}_1 \\ \cdots \\ \boldsymbol{d}_{m1} \end{bmatrix} \Rightarrow \boldsymbol{\Phi} \cdot \boldsymbol{w}^T = \boldsymbol{d}$$

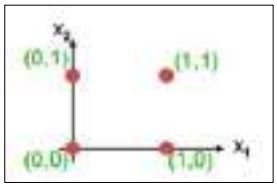
Teorema di Micchelli:
Sia $\{\boldsymbol{x}_i\}_{i=1}^{N}$ un insieme di punti $\in \mathbb{R}^m$ e distinti. Allora la **matrice Φ di interpolazione** tale per cui ciascuna componente ij sia $\varphi_{ij} = \varphi(\|\boldsymbol{x}_j - \boldsymbol{x}_i\|)$, se di dimensioni $N \times N$, è **non singolare**; ovvero il suo *determinante è* $\neq 0$ (il suo *rango non massimo*).

Tipi di $\boldsymbol{\varphi}$ per cui è valido il teorema di Micchelli:

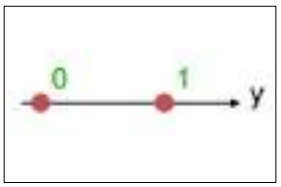
- **Multiquadratica:** $\varphi(r) = (r^2 + c^2)^{1/2}$
- **Multiquadratica inversa:** $\varphi(r) = \frac{1}{(r^2+c^2)^{1/2}}$ $\quad r = \|\boldsymbol{x} - \boldsymbol{t}\|$; $\quad c > 0$
- **Gaussiana (normale):** $\varphi(r) = exp\left(-\frac{r^2}{2\sigma^2}\right)$ $\quad \sigma > 0$

Esempio: *problema dello XOR*

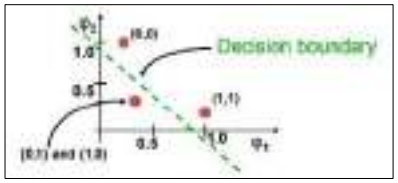
Input space:



Output space:



Mappati nello **spazio delle feature** $\langle \boldsymbol{\varphi_1}, \boldsymbol{\varphi_2} \rangle$**,** C_1 e C_2 diventano *linearmente separabili* e il classificatore lineare che riceve gli input $\varphi_1(\boldsymbol{x})$ e $\varphi_2(\boldsymbol{x})$ può risolvere il problema.



Parametri di una *Rete RBF* necessari ad apprendere*:*

1) Il centro della *funzione di attivazione RBF*
2) *L'ampiezza* della funzione *RBF,* se *Gaussiana*
3) *I pesi* dall'*hidden* all'*output layer*

Stabilire i parametri di *una rete RBF:*

1) <u>Metodo della matrice pseudo inversa:</u>

- *<u>Centri</u>:* selezionati **casualmente** dal training set
- *<u>Ampiezze dei neuroni hidden</u>:* scelte attraverso **normalizzazione** $\sigma = \frac{\text{Maximum distance between any 2 centers}}{\sqrt{\text{number of centers}}} = \frac{d_{\max}}{\sqrt{m_1}}$

  Funzione di attivazione del *neurone hidden i:* $\boldsymbol{\varphi}_i(\|\boldsymbol{x} - \boldsymbol{t}_i\|^2) = exp\left(-\frac{m_1}{d_{\max}^2} \cdot \|\boldsymbol{x} - \boldsymbol{t}_i\|^2\right)$

- *<u>Pesi</u>*: computabili attraverso l'**algoritmo della matrice pseudo-inversa**:

  Per ogni istanza $(\boldsymbol{x}_i, d_i)$, vorremmo che $\varphi(x_i) = d_i$.

  Questo vincolo può essere imposto attraverso l'*equazione matriciale* $\boldsymbol{\Phi w} = \boldsymbol{d}$:

$$\begin{bmatrix} \boldsymbol{\varphi}_1(\|\boldsymbol{x}_1 - \boldsymbol{t}_1\|) & \cdots & \boldsymbol{\varphi}_{m1}(\|\boldsymbol{x}_1 - \boldsymbol{t}_{m1}\|) \\ \vdots & & \vdots \\ \boldsymbol{\varphi}_1(\|\boldsymbol{x}_N - \boldsymbol{t}_1\|) & \cdots & \boldsymbol{\varphi}_{m1}(\|\boldsymbol{x}_N - \boldsymbol{t}_{m1}\|) \end{bmatrix} \begin{bmatrix} \boldsymbol{w}_1 \\ \cdots \\ \boldsymbol{w}_{m1} \end{bmatrix} = [\boldsymbol{d}_1 \quad \cdots \quad \boldsymbol{d}_N]^{\mathrm{T}}$$

  Che può essere riscritta come $\boldsymbol{\Phi}^T \boldsymbol{\Phi w} = \boldsymbol{\Phi}^T \boldsymbol{d}$.

> **Matrice pseudo-inversa $\boldsymbol{\Phi}^+$ (o pseudo-inversa di *Moore-Penrose*):**
> generalizzazione della *matrice inversa* al caso in cui $\boldsymbol{\Phi}$ non sia quadrata. **Definizione:** $\boldsymbol{\Phi}^+ \equiv (\boldsymbol{\Phi}^T \boldsymbol{\Phi})^{-1} \boldsymbol{\Phi}^T$.

Obiettivo: calcolare il *vettore pesi* che segue dall'*equazione matriciale* precedente $[\boldsymbol{w}_1 \dots \boldsymbol{w}_{m1}]^T = \boldsymbol{\Phi}^+ [\boldsymbol{d}_1 \dots \boldsymbol{d}_N]^T$
Problema: computare $\boldsymbol{\Phi}^+$ normalmente richiede di mantenere in memoria l'intera matrice $\boldsymbol{\Phi}$.
Idea: Dividere $\boldsymbol{\Phi}$ in Q sottoinsiemi, ciascuno con un arbitrario numero a_i di righe $(1 \le i \le Q)$.

Algoritmo:

a. Costruire un insieme di *<u>nuove matrici</u>* $\boldsymbol{A}_i$ tali che ciascuna di esse sia composta da righe di zeri tali che: $\boldsymbol{\Phi} = \sum_{i=1}^{Q} \boldsymbol{A}_i$

b. Sostituire questa definizione di $\boldsymbol{\Phi}$ nella definizione di $\boldsymbol{\Phi}^+$, ottenendo così:

$$\boldsymbol{w} = (\boldsymbol{\Phi}^T \boldsymbol{\Phi})^{-1} \boldsymbol{\Phi}^T \boldsymbol{d} = \left(\sum_{j.i\,=\,1}^{Q} \boldsymbol{A}_j^T \boldsymbol{A}_i\right)^{-1} \left(\sum_{j=1}^{Q} \boldsymbol{A}_j^T\right) \boldsymbol{d}$$

c. Definire la matrice $\boldsymbol{H} = \sum_{j.i=1}^{Q} \boldsymbol{A}_j^T \boldsymbol{A}_i$

> **Osservazioni:**
> - Tutti i prodotti $\boldsymbol{A}_j^T \cdot \boldsymbol{A}_i$ producono una matrice quadrata (ovvero $M \times M$)
> - Per via della costruzione di $\boldsymbol{A}_i$, tutti i prodotti $\boldsymbol{A}_j^T \cdot \boldsymbol{A}_i$ con $j \neq i$ producono una matrice *quadrata nulla*
> - Sia $\widehat{\boldsymbol{A}}_i$ un insieme di **matrici ridotte**, ovvero un insieme di matrici tali che ciascuna sia composta da una sola riga, quella non nulla, della matrice $\boldsymbol{A}_i$ da cui è generata. Otteniamo che $\boldsymbol{A}_i^T \boldsymbol{A}_i \equiv \widehat{\boldsymbol{A}}_i^T \widehat{\boldsymbol{A}}_i$.

d. Computare $\boldsymbol{H^{-1}} = \left(\sum_{j.i\,=\,1}^{Q} \boldsymbol{A}_j^T \boldsymbol{A}_i\right)^{-1}$:

1. Memorizzare $\widehat{\boldsymbol{A}}_i$ (dimensioni di $\widehat{\boldsymbol{A}}_i$: $a_i \times M$)
2. Computare $\widehat{\boldsymbol{A}}_i^T \widehat{\boldsymbol{A}}_i$
3. Aggiungere il prodotto ottenuto al punto **(2)** a una lista $\boldsymbol{v}$;

Dopo Q iterazioni, $\boldsymbol{v} = \boldsymbol{H}$. <u>Con quest'algoritmo viene mantenuta in memoria una sola $\widehat{\boldsymbol{A}}_i$ per volta.</u>
Inoltre, i prodotti possono essere computati in parallelo.

e. Computare $\boldsymbol{w} = \boldsymbol{H}^{-1} \left(\sum_{j=1}^{Q} \boldsymbol{A}_j^T\right) \boldsymbol{d}$:

Sia Q l'insieme di **vettori ridotti** $\widehat{\boldsymbol{d}}_i$, tali per cui ciascun $\widehat{\boldsymbol{d}}_i$ sia un insieme di componenti del vettore $\boldsymbol{d}$
Analogamente al passo precedente vale $\boldsymbol{A}_j^T \boldsymbol{d} \equiv \widehat{\boldsymbol{A}}_j^T \widehat{\boldsymbol{d}}_j$ e quindi: $\boldsymbol{w} = \boldsymbol{H^{-1}} \left(\sum_{j=1}^{Q} \widehat{\boldsymbol{A}}_j^T\right) \widehat{\boldsymbol{d}}_j$

1. Memorizzare $\widehat{\boldsymbol{A}}_j$
2. Computare $\widehat{\boldsymbol{A}}_j^T \boldsymbol{d}_j$
3. Aggiungere il termine calcolato al punto **(2)** a una lista
4. Moltiplicare il termine ottenuto per $\boldsymbol{H}^{-1}$

2) <u>Hybrid Learning Process</u>

- *<u>Centri</u>: determinati con un algoritmo di clustering*
    a. *<u>Inizializzazione</u>*: inizializza un vettore $\boldsymbol{t}$ le cui $\boldsymbol{m}_1$ componenti (inizialmente casuali) sono cluster
    b. *<u>Campionamento</u>:* rappresenta $\boldsymbol{x}$ dallo *spazio dell'input*
    c. *<u>Similarity matching</u>:* trova l'*indice del cluster più vicino* $k(\boldsymbol{x}) = \arg\min_k \|\boldsymbol{x}(n) - \boldsymbol{t}_k(n)\|$
    d. *<u>Aggiornamento:</u>* avvicina $\boldsymbol{t}_{k(\boldsymbol{x})}$ ad $\boldsymbol{x}$ di una misura pari al *learning rate* η ponderato per la distanza di $\boldsymbol{x}$ da $\boldsymbol{t}$

$$\boldsymbol{t}_k(n+1) = \begin{cases} \boldsymbol{t}_k(n) + \eta[\boldsymbol{x}(n) - \boldsymbol{t}_k(n)] & se\ k = k(\boldsymbol{x}) \\ \boldsymbol{t}_k(n) & altrimenti \end{cases}$$

    e. <u>Continuazione</u>: finché vi sono cambiamenti del centro consistenti, incrementa n di 1 e ricomincia dal punto **(2).**

- *<u>Ampiezze dei neuroni hidden</u>:* scelte attraverso **normalizzazione**
- *<u>Pesi</u>:* computati attraverso l'***algoritmo LMS***

3) <u>Metodo della discesa del gradiente</u> *per trovare i centri, lo spread e i pesi, minimizzando l'errore quadratico istantaneo*

- *<u>Centri</u>:* $\boldsymbol{\Delta t}_j = -\eta_{\boldsymbol{t}_j} \frac{\partial E}{\partial \boldsymbol{t}_j}$
- *<u>Ampiezze dei neuroni hidden</u>:* $\boldsymbol{\Delta \sigma}_j = -\eta_{\boldsymbol{\sigma}_j} \frac{\partial E}{\partial \boldsymbol{\sigma}_j}$
- *<u>Pesi</u>:* $\boldsymbol{\Delta w_{ij}} = -\eta_{\boldsymbol{w}_{ij}} \frac{\partial E}{\partial \boldsymbol{w}_{ij}}$

Comparazione tra *RBF* e *FFNN*:

- Entrambe le architetture sono ***stratificate in modo non lineare***
- Entrambe sono considerabili ***approssimatori universali***

<u>Architettura:</u>

- *Reti RBF* hanno un **singolo** *hidden layer*
- *Reti FFNN* possono avere **molteplici** *hidden layer*

<u>Modello dei neuroni:</u>

- Nelle *Reti RBF* i **modelli** dei neuroni *hidden* sono **diversi** dal modello del neurone in *output*
- Nelle *Reti FFNN* neuroni *hidden* e neurone di *output* tipicamente condividono un **modello comune**

- Nelle *Reti RBF* gli ***hidden layer*** sono **non lineari** e l'***output layer*** è **linerare**
- Nelle *Reti FFNN* tipicamente sia gli ***hidden layer*** che l'***output layer*** sono **non lineari**

<u>Funzione di attivazione dei neuroni *hidden*:</u>

- Nelle *Reti RBF* computa la **distanza euclidea** tra i vettori in input e il centro di quella unità.
- Nelle *Reti FFNN* computa il **prodotto interno** dei vettori in input e il vettore peso di quel neurone

<u>Approssimazione:</u>

- Nelle *Reti RBF* la funzione gaussiana costruisce un'**approssimazione locale** dell'associazione non lineare I/O
- Nelle *Reti FFNN* costruisce un'**approssimazione globale** dell'associazione non lineare I/O

5 – ELM: Extreme Learning Machine

***Rete* SLF**: *Single hidden Layer Feedforward Neural Network*

ELM: algoritmo di apprendimento per *Reti SLF* con **prestazioni computazionali** fino a centinaia di volte superiori dei tradizionali algoritmi di apprendimento basati sulla *backpropagation* e con **prestazioni di generalizzazione** migliori.

Definizione: una funzione si dice *infinitamente differenziabile* in un punto se la sua n-esima derivata esiste in quel punto $\forall n \in \mathbb{N}$.

Teoremi principali di ELM:
Ipotesi:
1. Sia data una *Rete SLF* con N neuroni *hidden e* sia $g: \mathbb{R} \to \mathbb{R}$ la loro *funzione di attivazione infinitamente differenziabile*.
2. Siano N un numero arbitrario di istanze distinte $(\boldsymbol{x}_i, \boldsymbol{t}_i)$, dove $\boldsymbol{x}_i \in \mathbb{R}^n$ e $\boldsymbol{t}_i \in \mathbb{R}^m$, e $\boldsymbol{T}$ il vettore di componenti $\boldsymbol{t}_i$.
3. Siano $\boldsymbol{\beta_i}$ il **vettore** dei **pesi** che associa l'i-esimo neurone *hidden* ai neuroni *output* e $\boldsymbol{\beta}$ il vettore di componenti $\boldsymbol{\beta_i}$**.**
4. Sia ε un qualsiasi numero arbitrariamente piccolo positivo.

1) Allora, per ogni $\boldsymbol{w}_i \in \mathbb{R}^n$ tra neuroni *input* ed *hidden* e $b_i \in \mathbb{R}$ scelti casualmente in funzione di una certa distribuzione di probabilità continua, è garantito che esista la matrice $\boldsymbol{H}$ *output* dell'*hidden layer* invertibile tale che valga $\|\boldsymbol{H\beta} - \boldsymbol{T}\| = 0$.

2) Allora, esiste $\widetilde{N} \leq N$ tale che $|\boldsymbol{H}_{N\times\widetilde{N}} \cdot \boldsymbol{\beta}_{\widetilde{N}\times m} - \boldsymbol{T}_{N\times m}| < \varepsilon$, dove $\widetilde{N}$ è il numero di neuroni *hidden*

Quindi: se le *funzioni di attivazione* dell'*hidden layer* di una sono *infinitamente differenziabili*, allora:
1) I **vettori pesi *input*** e ***hidden layer*** possono essere inizializzati casualmente
2) La *Rete SLF* può essere considerata come un sistema lineare e il **vettore pesi *output* $\boldsymbol{\beta}$** può essere determinato analiticamente attraverso l'**operazione inversa generalizzata $\boldsymbol{H}^+$** delle matrici di *output* dell'*hidden layer* $\boldsymbol{H}$.
Questo è molto utile in quanto non sempre $\boldsymbol{H}$ è quadrata**.**

Sia $\boldsymbol{\varphi}_i(v) = \sum_{h=1}^{\widetilde{N}} \boldsymbol{\beta}_h \cdot g(\boldsymbol{x}_j)$ la funzione di attivazione del i-esimo neurone *output* in risposta al j-esimo *input*.
Sebbene sia possibile il calcolo di $\boldsymbol{\beta}$ in funzione di $\boldsymbol{H}^{-1}$, non è quello a cui siamo interessati poiché $\boldsymbol{H}$ non è invertibile per $\widetilde{N} < N$.

$$\begin{bmatrix} g(\overline{\boldsymbol{x}_1} \cdot \overline{\boldsymbol{w}_1}) & \cdots & g(\overline{\boldsymbol{x}}_1 \cdot \overline{\boldsymbol{w}}_{\widetilde{N}}) \\ \vdots & & \vdots \\ g(\overline{\boldsymbol{x}_N} \cdot \overline{\boldsymbol{w}_{\widetilde{N}}}) & \cdots & g(\overline{\boldsymbol{x}}_N \cdot \overline{\boldsymbol{w}}_{\widetilde{N}}) \end{bmatrix} \begin{bmatrix} \beta_{11} \cdots \beta_{1m} \\ \cdots \\ \beta_{\widetilde{N}1} \cdots \beta_{\widetilde{N}m} \end{bmatrix} = \begin{bmatrix} y_{11} \cdots y_{1m} \\ \cdots \\ y_{N1} \cdots y_{Nm} \end{bmatrix}$$

$$\Rightarrow \boldsymbol{H} \cdot \boldsymbol{\beta} = \boldsymbol{y} \Rightarrow \boldsymbol{\beta} = \boldsymbol{H}^{-1} \cdot \boldsymbol{y}$$

Bensì vogliamo risolvere la seguente equazione matriciale: $\boldsymbol{H} \cdot \boldsymbol{\beta} \approx \boldsymbol{T} \Rightarrow \boldsymbol{\beta} = \boldsymbol{H}^+ \cdot \boldsymbol{T}$, che è invece sempre risolvibile (ma $\boldsymbol{T} \approx \boldsymbol{y}$).
Ovvero effettuare **regressione** e non **interpolazione**.

Algoritmo ELM:
1. Assegna valori casuali a $\boldsymbol{w}_i$ e b_i, $\quad i = 1, \dots, \widetilde{N}$ (l'*upper bound* del numero di nodi richiesti è N)
2. Computa $\boldsymbol{H}$
3. Computa la minima *norma least-square* $\boldsymbol{\beta} = \boldsymbol{H}^{+}\boldsymbol{T}$

L'algoritmo converge per ogni funzione di attivazione g *infinitamente differenziale*.
Minore è la norma del vettore pesi, maggiori sono le performance di generalizzazione.

Questo metodo è affetto da severe limitazioni quando la matrice è quasi *singolare*.
Matrice singolare: matrice *quadrata* ma *non invertibile,* poiché il suo *determinante* è $= 0$ (ovvero il suo *rango non è pieno*)

Approcci:
1) Aggiunta di un *termine di regolarizzazione,* così che il costo funzionale sia esprimibile come:

$$E = E_D + \lambda E_W = \frac{1}{2}\sum_{j=1}^{N}\sum_{k=1}^{Q}\left(\left(\boldsymbol{t}_j^k - \sum_{i=1}^{M} \boldsymbol{w}_{ki}\phi(c_i \cdot x_j + b_i)\right)^2 + \frac{\lambda}{2}\sum_{i=1}^{M}|\boldsymbol{w}_{ki}|^2\right)$$

La cui soluzione è: $\mathbf{w}^* = (\mathbf{H}^{\mathrm{T}}\mathbf{H} + \lambda\mathbf{I})^{-1}\mathbf{H}^{\mathrm{T}}$

2) *Calcolo della **S**ingular **V**alue **D**ecomposition di $\boldsymbol{H}$:*

Metodo computazionalmente più efficace. Inoltre, poiché non è necessario il *termine di regolarizzazione*, è possibile realizzare una *minimizzazione dell'errore dipendente dai dati* più efficace.

Definizioni:

- **Matrice unitaria:** matrice $\boldsymbol{U}$ che soddisfa la condizione $\boldsymbol{U}^{\dagger}\boldsymbol{U} = \boldsymbol{U}\boldsymbol{U}^{\dagger} = \boldsymbol{I}$
- **Matrice diagonale:** matrice in cui solamente i valori della diagonale principale possono essere diversi da 0.
- **Matrice trasposta coniugata:** matrice $\boldsymbol{V}$ che soddisfa la condizione $\boldsymbol{V}^{\dagger} = (\boldsymbol{V}^{\boldsymbol{T}})^{*} = (\boldsymbol{V}^{*})^{\boldsymbol{T}}$

Singular Value Decomposition ($\boldsymbol{SVD}$)*: fattorizzazione* di una matrice basata sull'uso di ***autovettori.***

Data una matrice $\boldsymbol{M}$ di dimensioni $m \times n$, si tratta di una scrittura del tipo*:* $\boldsymbol{M} = \boldsymbol{U}\boldsymbol{\Sigma}(\boldsymbol{V}^{T})^{*} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^{\dagger}$, dove*:*

- $\boldsymbol{U}$ è una **matrice unitaria** di dimensioni $m \times m$
- $\boldsymbol{\Sigma}$ è una **matrice diagonale** di dimensioni $m \times n$ le cui componenti vengono indicate come σ_{ij}
- $\boldsymbol{V}^{\dagger}$ è la matrice **trasporta coniugata** di una *matrice unitaria* $\boldsymbol{V}$ di dimensioni $n \times n$

Le componenti sulla diagonale di $\boldsymbol{\Sigma}$ sono detti **valori singolari** di $\boldsymbol{M}$**.**
Le colonne di $\boldsymbol{U}$ sono dette **vettori singolari sinistri** di $\boldsymbol{M}$ (o **autovettori** di $\boldsymbol{M}\boldsymbol{M}^{*}$)**.**
Le colonne di $\boldsymbol{V}$ sono dette **vettori singolari destri** di $\boldsymbol{M}$ (o **autovettori** di $\boldsymbol{M}^{*}\boldsymbol{M}$)**.**

Usando questa strategia, la *matrice pseudoinversa* $\mathbf{H}^{+}$ è definita come: $\mathbf{H}^{+} = \boldsymbol{V}\boldsymbol{\Sigma}^{+}\boldsymbol{U}^{\boldsymbol{T}}$**,** ovvero evitando l'inversione *matriciale*.

- $\boldsymbol{\Sigma}^{+}$ è una **matrice diagonale** le cui componenti *non nulle* assumono il valore $\frac{1}{\sigma_{ij}}$

L'inversione *numerica* di valori molto bassi σ_{ij} in $\boldsymbol{\Sigma}$ potrebbe rendere instabile l'algoritmo.

6 – CNN: Convolutional Neural Networks

Deep Neural Network (DNN): reti caratterizzate da un'ampia profondità e da una struttura gerarchica di *layer*.

- La **profondità** (numero di *layer*) è solo un fattore di complessità. **Neuroni**, **connessioni** e conseguentemente **numero di pesi** caratterizzano anche la complessità di un $\boldsymbol{DNN}$.
- L'**organizzazione gerarchica** consente di condividere e riutilizzare le informazioni. Lungo la gerarchia è possibile selezionare una caratteristica specifica e scartare i dettagli non necessari (al fine di massimizzare l'*invarianz*a).

Cenni storici
Per motivi di **efficienza**, gli studi inerenti le reti neurali per molto tempo si sono concentrati su reti con un singolo *hidden layer*: funzioni computabili con complessità polinomiale su n *layer* possono richiedere complessità esponenziale con un differente numero di *layer*. Questo perché all'aumentare del numero di pesi presenti nella rete, aumentano anche le risorse computazionali necessarie al suo addestramento.

Principali modelli DNN:
Addestramento supervisionato, modelli discriminativi:

- $\boldsymbol{CNN}$ - $\boldsymbol{Convolutional\ Neural\ Networks}$
- $FC\ DNN$ - $Fully\ Connected\ DNN$ (MLP with at least two hidden layers)
- HTM - $Hierarchical\ Temporal\ Memory$

Addestramento supervisionato, modelli generativi (utili per il pre-training):

- $Stacked$ (de-noising) Auto-Encoders
- RBM - Restricted Boltzmann Machine
- DBN - Deep Belief Networks

Modelli ricorrenti,

- RNN - Recurrent Neural Network
- $LSTM$ - Long Short-Term Memory

Motivi del successo delle DNN:
Le CNN sono state introdotte nel 1998 e hanno ottenuto un modesto successo per problemi di piccole dimensioni.
Nel 2012, con l'arrivo della rete convoluzionale $\boldsymbol{AlexNet}$, si è verificato un cambiamento radicale.
$AlexNet$ non introdusse innovazioni importanti rispetto alla CNN, ma alcuni fattori ne hanno decretato il successo:

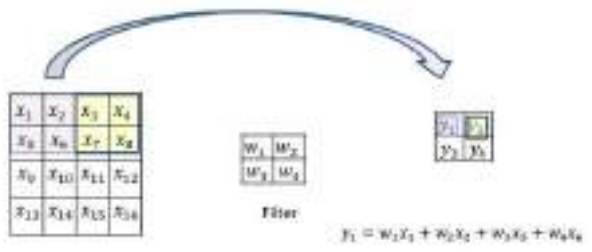
- La presenza di ***BigData*** (*dataset* con immense quantità di istanze etichettate)
- L'introduzione delle **unità di calcolo grafico** ($\boldsymbol{GPU}$): architetture che a differenza delle *unità di calcolo centrali* (CPU) favoriscono l'elaborazione di calcoli paralleli e che hanno drasticamente ridotto i tempi necessari all'apprendimento.
- La sostituzione della funzione di attivazione *sigmoide* in favore della **Relu**.

Convoluzione: operazione matematica attraverso la quale vengono applicati *filtri* ad una matrice.
L'applicazione viene effettuata, iterativamente e per scorrimento, a diverse porzioni dell'input attraverso il prodotto scalare.
Filtro: *maschera tensoriale di pesi*.

Architettura di una CNN: gerarchia di *layer*

- ***Convolutional layers****: sequenza di layer* che esegue la scansione dell'input applicando *filtri di convoluzione*.
  L'*output* prende il nome di *feature map.* Le dimensioni della *feature map* dipendono da:
  - ***Dimensioni del filtro:*** la profondità della *feature map* è uguale alla profondità del filtro.
  - ***Stride*****:** numero di unità di cui si muove per scorrimento la finestra del filtro.
  - ***Zero-padding:*** presenza di vettori colonna o riga nulli ai bordi della matrice di *input*.

Esempio:



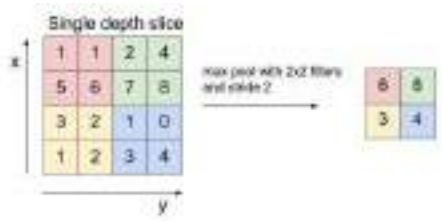
- ***Pooling layers:*** *sequenza di layer* che esegue *downsampling* (approssimazione di una sequenza in *input*).

Il layer di pooling esegue <u>*aggregazione di informazioni*</u>, generando *feature map* di dimensioni minori dell'*input*.

***Obiettivo*:** trasformare l'*input* mantenendo *invarianza* (ovvero mantenendo solo informazioni significative per la discriminazione).
Esempi di pooling: <u>pooling massimo</u> e <u>pooling medio</u>; prendono rispettivamente valore massimo e valore medio della sequenza.
Gli operatori che lo effettuano, avg e max, prendono il nome di *operatori aggregati*.

Anche in questo caso l'*output* prende il nome di *feature map*; le sue dimensioni dipendono dalle dimensioni del filtro.

Questo tipo di aggregazione <u>non necessita di parametri/pesi per apprendere</u>.



Nell'algoritmo di *backpropagation* i gradienti ai vari livelli vengono moltiplicati tramite la *chain rule*.
Scomparsa del gradiente: gli n fattori $\in (0,1) \rightarrow$ il prodotto decresce *esponenzialmente* rispetto a n (profondità della rete).
Esplosione del gradiente: gli n fattori assumono valori elevati $\rightarrow$ il prodotto decresce *esponenzialmente* rispetto a n.

- ***Fully connected layers:*** *sequenza di layer* che compone una tradizionale *Rete MLP.*

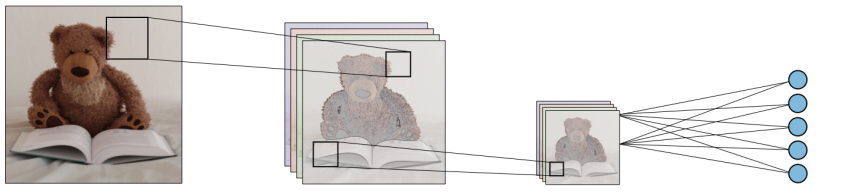
Funzione di attivazione:
- ***Relu (Rectified Linear):*** funzione di attivazione che risolve i problemi di scomparsa ed esplosione del gradiente.
  La derivata è 0 per valori negativi, 1 per quelli positivi. Questo porta ad *attivazioni sparse* e *robustezza* della rete.

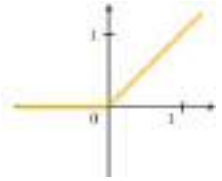
Scelte comuni nelle architetture *CNN* moderne:
- ***Soft-max layer:*** ultimo livello della rete, *completamente connesso* al precedente.
  Ciascun neurone i computa una funzione logistica generalizzata: prende come *input* il vettore di punteggi $\boldsymbol{x} \in \mathbb{R}^n$ generati dal livello precedente e genera un vettore di probabilità $\boldsymbol{p} \in \mathbb{R}^n$, dove n è il numero di neuroni.

$$\boldsymbol{p_i} = \frac{e^{\boldsymbol{x}_j}}{\sum_{j=1}^{n} e^{\boldsymbol{x}_j}}$$

- ***Cross-entropy loss function:*** funzione che quantifica la differenza tra due distribuzioni p e q.

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





Funzione di attivazione Relu
$\varphi = max(0, z)$

Caratteristiche di *CNN*:
<u>*Local Processing*</u>:
- Ciascun neurone effettua un processamento locale, ovvero differente da quello degli altri neuroni allo stesso livello.

<u>*Shared weights*</u>*:*
- Gli strati di una *feature map* condividono lo stesso *vettore pesi*.

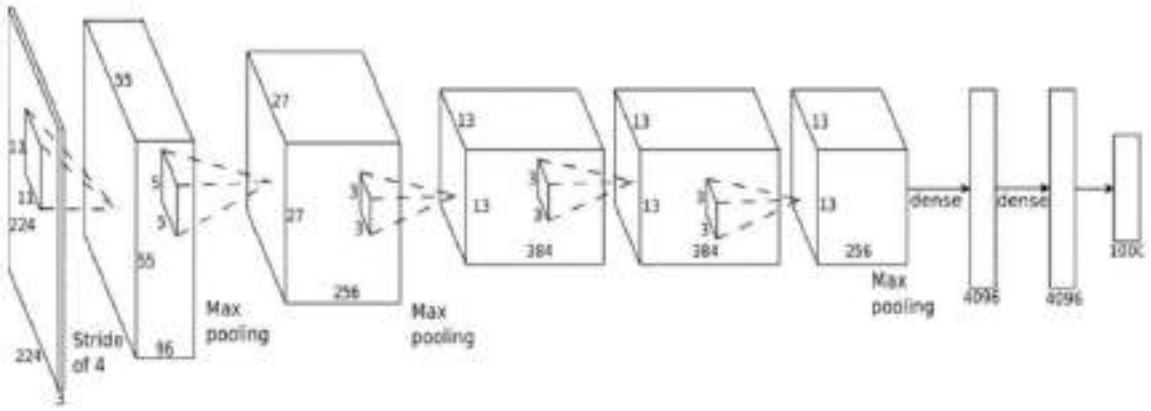
Insieme hanno contribuito a una forte riduzione del numero di pesi necessari alla rete per fare apprendimento.

Allenamento di una *CNN*: molto lento, a causa della necessità di dataset molto grandi.
Fortunatamente, una volta terminato l'apprendimento della rete, apprendere nuovi *pattern* è generalmente un'operazione più veloce. Il problema rimane la possibilità di usufruire di un dataset di dimensioni opportune.

Transfer leaning: memorizzare le conoscenze acquisite durante la risoluzione di un problema e applicarle a un problema diverso.

- ***Fine-tuning*:** apprendimento di un modello a partire da una *rete precedentemente allenata* su un problema simile:
  1) Sostituire l'*output layer* con un nuovo *layer* (tipicamente aggiornando il numero delle classi).
  2) Conservare i pesi della *rete* già allenata, tranne quelli agli ultimi due *layer*.
  3) Effettuare ulteriori iterazioni di apprendimento per ottimizzare i pesi in rispetto alle peculiarità del nuovo dataset

- ***Features re-using*:** a partire da una *rete precedentemente allenata* (su cui non è stato effettuato *fine-tuning*):
  1) Estrarre da un *layer intermedio* le *feature* generate dalla *rete*
  2) Usare tali *feature* per allenare un classificatore esterno (ad esempio $\boldsymbol{SVM}$) a classificare nuovi pattern.



7 – SOM: Self-Organizing Maps

Modello di Rete Neurale caratterizzato un tipo di apprendimento **non supervisionato:** l'**apprendimento competitivo**, il quale prevede che i neuroni di *output* competano tra loro per essere attivati.
Il neurone di *output* che vince la competizione prende il nome di *neurone vincitore***.**

Nelle SOM i neuroni sono posti in un *reticolo* che definisce un **ordine topologico**, tipicamente **mono/bi-dimensionale**.

Tali neuroni si ***regolano selettivamente*** alla ricezione di ***stimoli*** (ovvero di *pattern input*), la quale avviene in modo continuo fino al termine dell'apprendimento. Tali *stimoli* modificano la posizione dei neuroni, i quali catturano le regolarità delle *feature* e producono un sistema di coordinate significativo rispetto alla loro varianza che prende il nome di ***feature map***.

> "$The\ brain\ is\ organized\ in\ such\ a\ way\ that\ different\ sensory\ inputs\ are\ represented\ by\ topologically\ ordered\ maps.\ Sensory\ inputs\ such\ as\ tactile, visual\ and\ acoustic\ inputs\ are\ mapped\ onto\ different\ cerebral\ cortex\ areas.$"

Ciascun *input pattern* presentato alla rete produce l'attivazione di una regione localizzata della *feature map*.
La posizione e la natura di dell'attivazione sono in funzione delle *feature* dell'*input pattern*.

Affinché l'addestramento della rete si sviluppi correttamente, essa deve essere stimolata attraverso input sufficientemente diversi tra loro. La presenza di ridondanza è fondamentale, infatti gli *stimoli* in fase di apprendimento vengono **campionati** con una certa distribuzione di probabilità a partire dall'*input space*.

A ciascuna feature dell'*input pattern* $\boldsymbol{x}$ è connesso ciascun neurone j della rete, al quale è associato a un vettore pesi $\boldsymbol{w}_j$.

Processi di una SOM:

1) Competizione: per ogni *input pattern* $\boldsymbol{x}$, i neuroni nella rete computano i rispettivi valori di una *funzione discriminante* $i(\boldsymbol{x})$.
Tale *funzione discriminante*, $i(\boldsymbol{x}) = arg\ \min_j \|\boldsymbol{x} - \boldsymbol{w}_j\|$, designa il ***neurone vincitore*** i (o ***Best Matching Unit,*** $\boldsymbol{BMU}$).
Minimizzare la norma 2 (distanza euclidea) e massimizzare il prodotto scalare sono criteri che coincidono.

> ***Selezione del neurone vincitore***:
> ***1.*** La rete viene stimolata da $\boldsymbol{x}$
> ***2.*** Per ogni j, la rete calcola $\boldsymbol{w}_j^T \boldsymbol{x}$
> ***3.*** Poi seleziona il neurone associato al prodotto maggiore (distanza euclidea minore).
> Tale neurone sarà il centro di un **vicinato topologico** di *neuroni eccitati*.

2) Cooperazione: il *neurone vincitore* determina la posizione spaziale del **vicinato topologico** di *neuroni eccitati*.

> ***Lateral interaction:***
> "$In\ the\ human\ brain\ a\ neuron\ that\ is\ firing\ tends\ to\ excite\ the\ neurons\ in\ its\ immediate\ neighborhood\ more\ than\ those\ farther\ away\ from\ it, which\ is\ intuitively\ satisfying.$"

Sulla base di questo principio è stata modellata la **selezione del vicinato topologico**, centrato nel *neurone vincitore* i circondato da un insieme di *neuroni eccitati* j.
Una buona scelta (non l'unica possibile) di *funzione di selezione del vicinato topologico* h_{ji} è la funzione gaussiana:

$$h_{j,i(\boldsymbol{x})} = exp\left(-\frac{d^2{}_{j,i}}{2\sigma^2}\right)$$

$d_{j,i}$ è la *distanza laterale* tra i e il neurone vincitore j, ovvero la distanza nel *feature map* (**non** nello *spazio dell'input*).
σ è il parametro che regola la **larghezza effettiva** del *vicinano topologico.*

3) Adattamento sinaptico: meccanismo che, aggiustando i *pesi sinaptici* $\boldsymbol{w}_j$ dei *neuroni eccitati*, incrementa i valori della *funzione discriminante* in relazione all'*input pattern* $\boldsymbol{x}$. Tali aggiustamenti enfatizzano la risposta del *neurone vincitore* e del *vicinato topologico* a futuri stimoli simili ad $\boldsymbol{x}$.

$$\widehat{\boldsymbol{w}}(n+1) = \widehat{\boldsymbol{w}}_j(n) + \eta(n)\ h_{j,i(\boldsymbol{x})}(n)\ [\boldsymbol{x}(n) - \boldsymbol{w_j}(n)]$$
$$\Longrightarrow \Delta \boldsymbol{w} = \boldsymbol{w}(n+1) - \boldsymbol{w}(n) = \eta(n) h_{j,i(\boldsymbol{x})}(n)[\boldsymbol{x}(n) - \boldsymbol{w_j}(n)] = \eta \cdot h_{j,i(\boldsymbol{x})} \cdot (\boldsymbol{x} - \boldsymbol{w_j})$$

Fasi del processo adattivo:

1) Fase di *ordinamento*, o di ***auto-organizzazione*** (~ 1000 iterazioni)
Fase in cui avviene l'*ordinamento topologico* dei vettori peso e in cui i parametri η e σ_o assumono i seguenti valori:
$\eta = 0.1$ (*lower bound* di decrescita: 0.01) $\sigma_o =$ raggio del reticolo
2) Fase di convergenza (~ [500 $*$ # $neuroni$] iterazioni)
Fase in cui aumenta la precisione di corrispondenza tra *pattern input* e neuroni e in cui i parametri η e σ si stabilizzano intorno a $\eta = 0.1$ e $\sigma =$ valore tale che $\mathrm{h_{j,i(x)}}$ contenga solo i neuroni immediatamente più vicini al BMU.

Algoritmo di apprendimento:

1) Inizializza il vettore pesi, desiderabilmente con valori bassi e diversi per ciascun neurone.
2) Campiona uno *stimolo* a partire dall'*input space* seguendo una certa distribuzione di probabilità*.
   1. Similarity matching: ricerca il *neurone vincitore* (BMU).
   2. Aggiorna i vettori peso dei *neuroni eccitati*.
3) Torna a (2), se ci sono stati cambiamenti significativi nell'aggiornamento dei pesi.

L'algoritmo di apprendimento prevede un ***decadimento esponenziale iterativo in*** n di σ e η in funzione del *parametro temporale* τ

$$\eta(n) = \eta_o \, exp\left(-\frac{n}{\tau}\right) \qquad \sigma(n) = \sigma_o \, exp\left(-\frac{n}{\tau}\right)$$

* La distribuzione di probabilità del campionamento influenza la *feature map*, all'interno della quale gli stimoli più frequenti saranno *meglio* ("*con una maggiore risoluzione*") rappresentati.

8 – Hopfield & Boltzman Networks

***Energy based model*:** modello che si avvale di una *funzione*, detta *di energia globale*; tale funzione, dati due neuroni i, j e il peso $\boldsymbol{w}_{ij}$ della loro connessione, associa un valore (*energia*) alla loro specifica configurazione di stati.

Le *Reti di Hopfield* sono il più semplice tipo di *energy based model* e sono composte da **unità** (*neuroni*) **binarie di soglia,** collegate tra loro da **connessioni ricorrenti** (*bidirezionali*). Tali reti sono caratterizzate dalla seguente **funzione di *energia globale***:

$$E = -\sum_i s_i b_i - \sum_{i<j} s_i s_j \boldsymbol{w}_{ij}$$

dove b_i è *bias* di ciascuna unità i, s_i è il suo *stato* e $\boldsymbol{w}_{ij}$ è il *peso del collegamento* tra le unità i e j.

Ciascuna *unità* i può **quantificare localmente** la propria influenza sull'*energia globale*, ovvero il proprio ***energy gap***:

$$v_i(n) = \Delta E_i = E(s_i = -1) - E(s_i = 1) = b_i + \sum_j s_j \boldsymbol{w}_{ij}$$

Inoltre, la *funzione di attivazione* φ *è:* $\varphi\left(v_j(n)\right) = \begin{cases} 1 & \text{if } v_j(n) > 0 \\ -1 & \text{if } v_j(n) < 0 \end{cases}$

Possibili *outcome* di reti ricorrenti di unità non lineari:
- Raggiungimento di uno stato stabile
- Oscillazione
- Perseguimento di traiettorie caotiche (non prevedibili)

Analisi del comportamento di una rete ricorrente con <u>connessioni simmetriche</u> e <u>opportune *funzioni di attivazione*</u>:
- Emerge una *funzione di energia globale*.
- ***La rete si stabilizza nei punti di minimo*** della *funzione di energia globale.*

Il tipico *task* che le *Reti di Hopfield* sono in grado di eseguire è il ***pattern completion***: data una porzione sufficiente di informazione memorizzata, recuperarne la completezza. Lo realizzano attraverso un'operazione che prende il nome di ***information withdrawal***.

Algoritmo <u>non deterministico</u> di *information withdrawal* (attraverso la minimizzazione della funzione di energia):
1) Calcola l'*energia globale* della configurazione di stati.
2) Applica ad un'*unità* <u>casuale</u> la *funzione di attivazione.*
3) Torna a (1), se la rete non si è *stabilizzata* (ovvero se la sua *configurazione* non corrisponde a un punto di minimo).

<u>Teorema di convergenza</u>: l'algoritmo ***converge sempre.***

<u>Dimostrazione in assenza di bias</u>:

Sia N il numero di unità della rete. Allora esistono 2^N possibili configurazioni della rete.

A ciascuna configurazione, attraverso la *funzione di energia senza bias* $E = -\frac{1}{2}\sum_i \sum_j s_i s_j \boldsymbol{w}_{ij}$, viene assegnato un valore.

Osservazione: (s_i, s_j *concordi* $\wedge\, \boldsymbol{w}_{ij} > 0$) $\vee$ (s_i, s_j *discordi* $\wedge\, \boldsymbol{w}_{ij} < 0$) $\Longrightarrow E > E'$, ovvero *l'energia globale decresce*.

Siano E ed E' due configurazioni successive nell'apprendimento: $E - E' = -\sum_{j\neq k} s_j(s_k - s_k')\boldsymbol{w}_{kj}$, k unità aggiornata.
- $s_k = 1 \wedge s_k' = -1 \Longrightarrow s_k - s_k{}' > 0 \wedge \sum_j \boldsymbol{w}_{kj} s_j < 0$
- $s_k = -1 \wedge s_k' = 1 \Longrightarrow s_k - s_k{}' < 0 \wedge \sum_j \boldsymbol{w}_{kj} s_j > 0$

Essendo il numero di possibili configurazioni della rete finito, <u>è dimostrato che l'algoritmo converge</u>.

Osservazioni:
1. La *funzione di energia globale* può avere più di un punto di minimo (o ***memoria stabile***), ma non è garantito che l'algoritmo converga nel punto di ***minimo assoluto** (o **memoria fondamentale).***
2. La regola di aggiornamento non è *error-driven*.

***Principio di Hebb* per l'apprendimento:**
<u>*Rafforzare*</u> le connessioni tra unità aventi <u>*stesso stato*</u> e <u>*indebolire*</u> le connessioni tra unità con <u>*stato opposto*</u>.

Controparte biologica: il cervello
- Rafforza le sinapsi tra unità spesso attive allo stesso tempo.
- Indebolisce quelle tra i neuroni che non sono attivi simultaneamente.

<u>Algoritmo di apprendimento</u>:

Memorizzazione: date M memorie fondamentali, calcola ogni collegamento $\boldsymbol{w}_{ji}$ tale che $j \neq i :\ \boldsymbol{w}_{ji} = \frac{1}{M}\sum_{k=1}^{M} f_k(i) f_k(j)$

Qualità: *pattern completion*, generalizzano input simili, permettono l'estrazione di prototipi, fault tolerant, Hebb rule, context effect.

Difetti: <u>*minimi spuri*</u>, *sinapsi simmetriche* e *stati stabili* (non corrisposti biologicamente - *stati transitori*), ***<u>capacità limitata</u>***.

Capacità limitata delle reti di *Hopfield*: N unità → $0.15N$ memorie → uso *non efficiente* della memoria dovuto al comportamento della *funzione di energia globale*, la quale **fonde due minimi** se la loro distanza è inferiore a una certa soglia.

Modifiche che migliorano la capacità delle reti di Hopfield:

- **Aggiunta di unità *hidden* alla *Rete di Hopfield standard***:

  Approccio teorizzato da *Hopfield*, *Feinstein* e *Palmer*, che però non ne hanno approfondito l'analisi e ispirato agli studi di *Crick* e *Mitchison* sull'importanza della *fase REM* nel sonno.
  Realizzare una fase di "*unlearning*" a seguito di apprendimento al fine di superare eventuali *minimi locali*.

  - Apprendimento: determinazione di nuovi minimi della *funzione di energia globale*.
  - Unlearning: modifica dell'apprendimento effettuato.

  Le ***unità hidden*** rappresentano l'*interpretazione percettiva* che la rete fa dell'input.
  I ***pesi*** rappresentano i vincoli sulle *interpretazioni*.
  La ***funzione d'energia globale*** misura la *bontà dell'interpretazione*.

- ***Pseudo-likelihood* tecnique:**

  Approccio, scoperto dalla fisica *Elizabeth Gardiners*, che consiste nel sostituire la regola di aggiornamento standard con l'applicazione della *regola di convergenza del percettrone* su ciascuna unità della rete.

L'apprendimento con quest'approccio:

1. **Evita minimi locali** (ovvero *soluzioni sub-ottime*) con l'introduzione di *non determinismo*.

   Simulated annealing (*Kirkpatrick*): euristica che introduce nell'apprendimento un *fattore di rumore* (*noise*).
   La sua implementazione richiede che le ***unità binarie*** siano ***stocastiche*** e non più **di soglia** (che invece sono *deterministiche*)**.**
   Stocasticamente ad ogni iterazione decide con probabilità p se transire dalla *configurazione di stati* attuale.
   Probabilità di transizione: $p_j = \frac{1}{1+e^{-\Delta E_j/T}}$
   Osservazioni:
   – p è una *funzione logistica normale*
   – T (***temperatura***) è un parametro *di regolarizzazione*

2. **Massimizza** la ***verosimiglianza* (*likelihood*) *congiunta*** delle istanze $\boldsymbol{v}$ del *training set*, a ciascuna delle quali il modello assegna la *verosomiglianza* $\boldsymbol{p_v}$:

$$\max_{\boldsymbol{w}} \prod_{\boldsymbol{v}} \boldsymbol{p_v} \equiv \min_{\boldsymbol{w}} \sum_{\boldsymbol{v}} log\, \boldsymbol{p_v}$$

$$\frac{\partial\, log(\boldsymbol{p_v})}{\partial \boldsymbol{w}_{\mathrm{ij}}} = \langle s_i \cdot s_j \rangle_{\boldsymbol{v}} - \langle s_i \cdot s_j \rangle_{\boldsymbol{model}} \Longrightarrow \Delta \boldsymbol{w}_{\mathrm{ij}} \propto \langle s_i \cdot s_j \rangle_{\boldsymbol{data}} - \langle s_i \cdot s_j \rangle_{\boldsymbol{model}}$$

   $\langle s_i \cdot s_j \rangle_{\boldsymbol{data}}$ è il **termine di apprendimento**, ovvero il **valore atteso** di $s_i \cdot s_j$ dato dal vettore $\boldsymbol{data}$ (*media* del *training set)*.
   $\langle s_i \cdot s_j \rangle_{\boldsymbol{model}}$ è il **termine di unlearning,** ovvero **valore atteso** di $s_i \cdot s_j$ quando la rete non viene sottoposta ad *input*.

   Osservazione: tale l'algoritmo di allenamento eredita la stessa *inefficienza* dell'algoritmo del percettrone.

Restricted Boltmann Machines (RBM): semplificazione delle tradizionali *Reti di Boltmann*.
Sono caratterizzate da **un solo *hidden layer***, e dal fatto che le ***unità*** presenti sullo ***stesso livello non*** sono ***collegate tra loro***.

Algoritmo di apprendimento di una RBM:

1) Sottopone l'*RBM* all'*input*
2) Attende che la *rete* raggiunga l'equilibrio termico, ovvero che la configurazione di stati non transisca più
3) Applica la regola di aggiornamento $\Delta \boldsymbol{w}_{ij} = \varepsilon\big(\langle s_i \cdot s_j \rangle^0 - \langle s_i \cdot s_j \rangle^\infty\big)$

Tale algoritmo è di per sé più efficace dell'algoritmo di apprendimento delle *Boltzman Machine*, poiché l'indipendenza dei neuroni sullo stesso livello permette di aggiornarli in parallelo. Esiste tuttavia una versione ulteriormente migliorata di tale algoritmo:

Contrastive Divergence Algorithm:

1) Computa $\langle s_i \cdot s_j \rangle^0$ e $\langle s_i \cdot s_j \rangle^1$, rispettivamente data e reconstructor.
2) Applica la regola di aggiornamento $\Delta \boldsymbol{w}_{ij} = \varepsilon\big(\langle s_i \cdot s_j \rangle^0 - \langle s_i \cdot s_j \rangle^1\big)$, che non segue il *gradiente della verosimiglianza* ma che è comunque efficace in quanto sfrutta la differenza di energia tra i due addendi (temperatura bassa nel primo, alta nel secondo).

Osservazione: i pesi catturano *feature globali* del *training set*, quindi la ricostruzione a partire da classi non familiari finisce comunque con l'obbedire alle regolarità delle feature catturate.