# INTRODUZIONE - perceptron

**Reti neurali** **(NN)** 🡪 tecnica all’interno dell’apprendimento automatico nata a partire dagli anni ’60, hanno subito battuta di arresto e hanno avuto un nuovo boom negli anni ‘80

’90 / 2000’ 🡪 di nicchia

**2 tipi di NN:**

* 1. **Supervisionato** 🡪 si ha un insieme di addestramento formato da input e output desiderato a partire da quell’input: usate per **classificazione, riconoscimento, diagnostica, regressione**
  2. **Non** **supervisionato** 🡪 insiemi di addestramento non vengono associati alla risposta desiderata (non etichettati): impara differenti realizzazioni di input (es. imparano ad associare dalle caratteristiche degli input le istanze del problema). Usate per **clustering**

**NN** ispirato ai neuroni biologici 🡪 differenza in NN è che i neuroni al suo interno possono essere assemblati in maniera molto differente e molto vari. Il funzionamento collettivo in una NN differisce (anche) profondamente dal funzionamento dei neuroni biologici

**NN** = dispositivo parallelo e distribuito che utilizza un insieme di parametri detti **pesi** con differenti valori 🡪 obiettivo di addestramento di una nn = portare i pesi ad avere valori corretti ed adatti al loro scopo

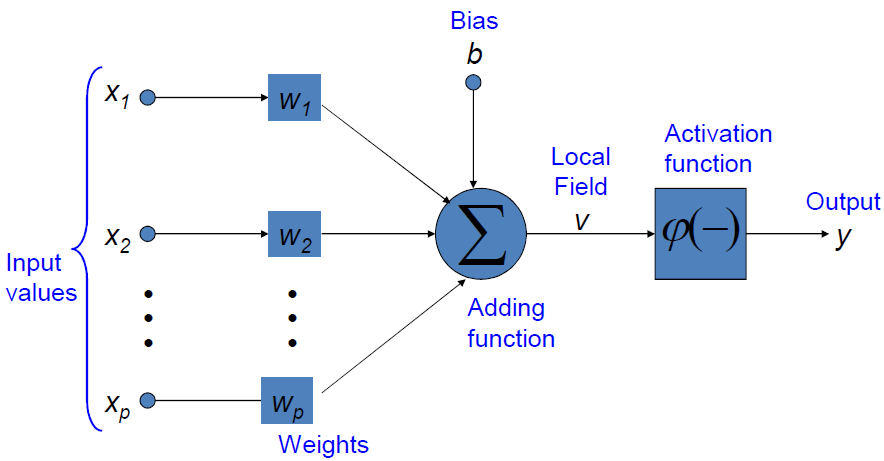
**Insieme di training =** esempi utilizzati per l’apprendimento

**Proprietà NN:**

* Non linearità (sfrutta funzioni non lineari)
* Mappaggio input/output: creato a partire dall’apprendimento
* Buona tolleranza al danneggiamento: è possibile eliminare alcuni neuroni senza portare al crollo delle performance immediato e drastico
* Addestramento incrementale: si può continuare ad addestrare la rete se si hanno dei nuovi dati disponibili

**Componenti di una NN**:

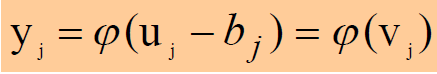
* Modello neuronale 🡪 come è strutturato il neurone
* Architettura 🡪 definisce come i neuroni sono connessi tra loro
* Algoritmo di apprendimento 🡪 usato per addestrare la NN: ogni neurone presenta i **pesi** che questo algoritmo modifica. Lo **scopo** è quello di ottenere una struttura che **generalizzi bene** quando giungeranno formulazioni nuove (ma non coincidenti con i training set)

**Il neurone**

**Ingresso neurone** = somma pesata 🡪 x1\*w1 + x2\*w2 + …

A parità dei valori x1, …, xn 🡪 ciò che entra nel neurone cambia in base ai **pesi** 🡪 uscita varia a seconda dei pesi

**Campo locale *v*** 🡪 ingresso alla **funzione di attivazione (o trasferimento)**: solitamente non lineare

Alla somma pesata contribuisce anche il **bias** = contributo additivo alla somma pesata

**Neurone** = informazione di base della NN composta da un insieme di **link** con i vari pesi ad essi associati

Y è la funzione di uscita

è la funzione di attivazione

è la somma pesata (somma degli ingressi per il loro peso)

è funzione additiva = il **bias**

è la funzione locale = ingresso del neurone

**Bias** 🡪 quantità che consente di “allontanare” da zero la somma pesata : questo per far sì che con tutti i pesi 0 o i componenti in ingresso (x1, …) la risposta del neurone non sia nulla

Il bias può essere sfruttato come input extra assieme alle altri componenti 🡪 **x0 = 1** con il peso **w0=bias** ad esso associato: questo per far si di sfruttare il peso come il bias

**La scelta di**  **determina il modello neuronale**

Una delle più utilizzate è la **sigmoide** 🡪

3 tipi di **architettura**

1. single layer feed-forward 🡪 **perceptron:** un unico livello di uscita
2. multi layer feed-forward 🡪 **multi-layer perceptron**: introduce un livello nascosto perché non riceve info dal mondo esterno e tantomeno ne fornisce al mondo esterno 🡪 permette di classificare anche insieme di dati non linearmente separabili
3. recurrent 🡪 l’uscita prodotta viene riportata in ingresso e rielaborata

**Perceptron** 🡪 è l’architettura più semplice per una NN, consiste in un solo neurone a cui è associato un algoritmo di apprendimento

**Primo modello di percettrone** 🡪 metà anni ’60, è una NN senza livelli intermedi (o unità che vedono il mondo esterno oppure forniscono info al mondo). Funzione di attivazione che vale +1 se v>0 -1 altrimenti 🡪 dove v è il famoso **campo di ingresso**

Può essere addestrato in funzione a ciò che gli viene fornito in ingresso 🡪 tramite assegnazione di valori al vettore w (quello dei pesi) in modo che il percettrone possa classificare gli esempi forniti

L’algoritmo mira ad orientare la retta in un piano bi-dimensionale affinché divida esattamente le classi da identificare 🡪 solo se l’insieme è linearmente separabile

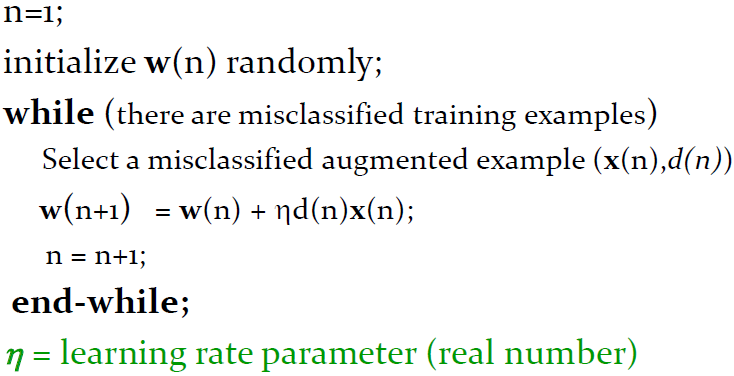
**Perceptron 🡪** usato per classificazione binaria

# perceptron

Esempio classificazione e training rule pag. 3 slides 2.0 🡪 in questo esempio si è modificata la retta verde in quella blu per far sì che P fosse classificato con la classe C1

**Training rule** 🡪 va a modificare i parametri usati per determinare la classe di un punto. In particolare, aggiunge un **learning rate parameter =**  🡪 modifica la quantità che si somma al peso w(n)

**positivo e solitamente <= 1**

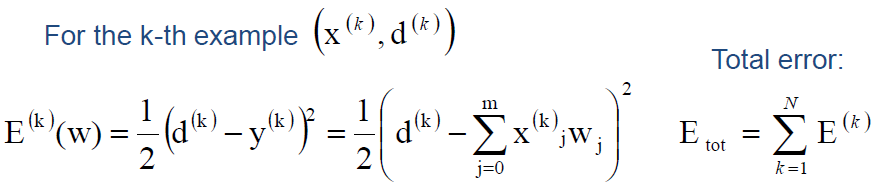
**Algoritmo di apprendimento:** d è la classe target (es. 1 se classe deve essere C1, -1 per C2)

Caso bidimensionale:dal **vettore dei pesi** 🡪 si può trovare l’**equazione della retta = decision boundary:**

**Teorema di convergenza** 🡪 (ha detto che non lo chiede all’esame [in pratica]) 🡪 garantisce che l’algoritmo del percettrone termini con successo dopo un numero finito di iterazioni se le classi C1 e C2 sono linearmente separabili. Dimostrazione pag. 5-8

**Epoca =** applicazione dell’update rule a ciascun esempio del training set (es. pag. 10) 🡪 apprendimento si interrompe quando alla fine di un’epoca non è stata applicata la training rule su nessun peso

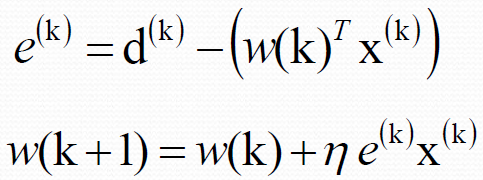
**Adaline = Adaptive Linear Element** 🡪 è un percettrone con un algoritmo di apprendimento fatto in modo da classificare tenendo conto di un determinato errore 🡪 ha come obiettivo quello di minimizzare l’errore: permette di sfruttare insiemi non linearmente separabili

**Errore Minimo Quadrato = LMS (Least Mean Square)** 

Ad ogni iterazione l’algoritmo LMS seleziona un esempio e cerca di diminuire l’errore di classificazione della rete neurale 🡪 incremental gradient descent

La direzione in cui l’errore E sta descrescendo più rapidamente = **opposto del gradiente di E** (che rappresenta la funzione dei pesi)

**Gradiente di E**:

**Update rule:** modifica il peso seguendo la direzione opposta del gradiente e

Si sta monitorando l’errore, quindi ci si vuole muovere nella direzione opposta del gradiente dell’errore (questo cosa stracazzo significa?) 🡪 estensione a più dimensioni della derivata (credo): è un vettore composto dalle derivate parziali delle funzione

# MULTI LAYER PERCEPTRON

**Multi layer perceptron (MLP)** [fa parte delle FFNN = Feed Forward NN]: consente di risolvere problemi reali e realistici con una piccola modifica 🡪 **aggiunta di un livello in più nella rete = hidden layer**

**Hidden layer** 🡪 riceve valori dal livello precedente e fornisce valori a quello successivo

Ce ne possono essere più di uno

Serve a risolvere problemi non linearmente separabili

Ogni neurone introduce una sorta di separazione all’interno dello spazio

**Modello di neurone 🡪** dotato di una funzione di attivazione 🡪**sigmoide**: con a > 0

**Algoritmo di apprendimento 🡪 backpropagation**: cerca valore ottimale dei pesi per far sì che l’errore sia minimo rispetto al target.

Addestra la rete in modo tale che quando si presenta un pattern generico *xk* 🡪 ogni valore dei neuroni di uscita sia più prossimo rispetto a quello che si desiderava

**Sfrutta 2 step:**

* 1. **Forward pass:** ci si propaga in avanti attraverso somme pesate di livello in livello e si ottiene un valore sui neuroni di uscita (inizialmente i valori di uscita saranno molto diversi da quelli voluti)
  2. **Backward pass:** partendo dai neuroni di uscita, si calcolano dei valori per modificare i pesi nella direzione che ci permette di ottenere il risultato desiderato 🡪 calcola un gradiente di errore per ogni neurone di ogni livello della NN

Backpropagation suddivide l’esecuzione in:

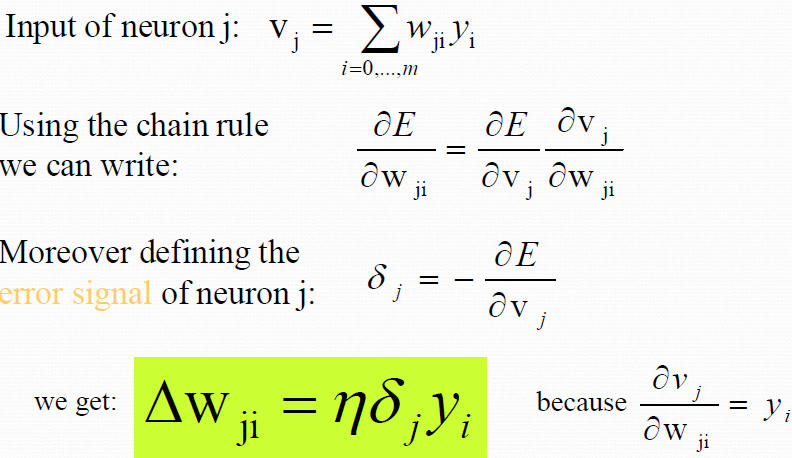
* Correzione ai pesi dei neuroni nel livello uscita
* Correzione ai pesi dei neuroni nei livelli nascosti

Per entrambi servono delle definizioni di errore:

🡪 **errore:** differenza tra ciò che si vorrebbe (d) e ciò che fornisce il livello di uscita

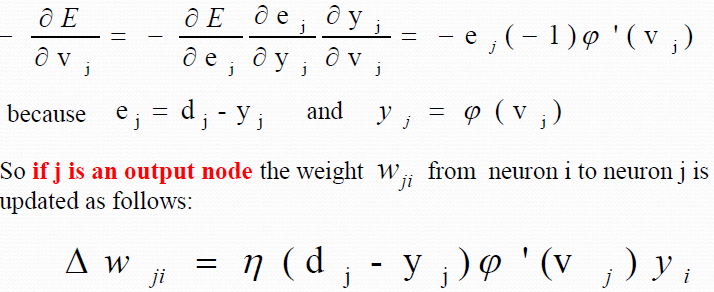
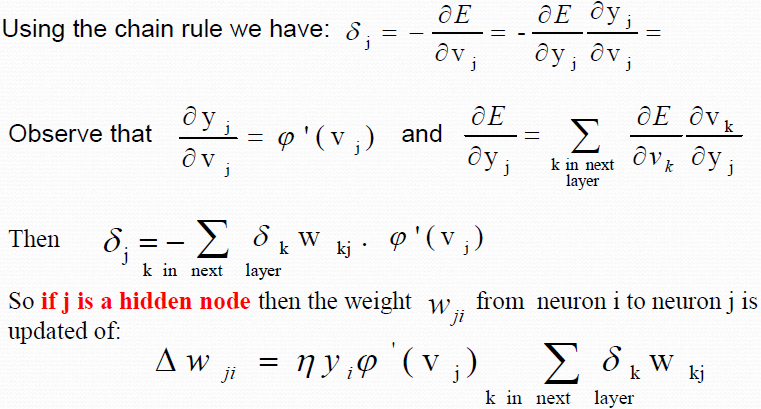
🡪 **pattern error:** definisce l’errore ennesimo (andando a sommare tutti gli errori dei vari neuroni della NN)

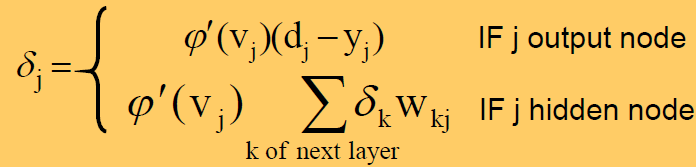
🡪 **Total Mean Squared error**: errore totale medio dei training examples

**Regola di update dei pesi** 🡪 basata sul metodo di discesa del gradiente 🡪 **direzione opposta del gradiente per minimizzare l’errore**. Si sfrutta il learning rate 🡪 ci si vuole muovere in direzione opposta a quella in cui l’errore cresce 

**Segnale di errore** (error signal) 🡪 calcolato e usato per apportare la modifica ai pesi

Ci sono 2 casi in cui si sfrutta la formula in verde:

1. Il neurone è nel **livello d’uscita**: fornisce direttamente il suo output al mondo esterno 🡪 ci si accorge subito se si commette o meno un errore
2. Il neurone è in un **livello hidden**: l’info sull’errore eventualmente commesso viene portata indietro dall’uscita fino ai neuroni più vicini all’ingresso (perché non hanno info sull’errore commesso rispetto al risultato che si vuole ottenere) 🡪 modificano uscite dei neuroni nel livello successivo perché ne modificano gli input

**La delta rule è** (per la modifica dei pesi): 

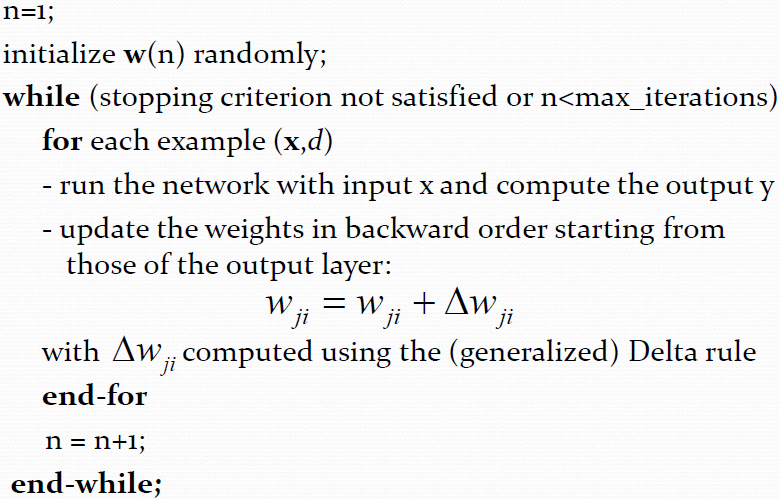
**Delta rule generalizzata** 🡪 amplia la regola aggiungendo un termine per il variare del valore di *eta (= learning rate 🡪 )* se è **piccolo** l’algoritmo impara lentamente i pesi, mentre se è **grande** si creano dei grossi cambiamenti ai pesi che possono causare un comportamento instabili con grosse oscillazioni dei valori dei pesi.

**Aggiunge un termine di momento** 🡪 🡪 (alfa compreso tra 0 ed 1 [non conosciuto a priori]) 🡪 se il momento concorda col segno del termine definito nella delta rule 🡪 i due termini si sommano e si ha un incremento più deciso del peso 🡪 velocizza il processo. Se sono attorno ad un minimo, invece si riduce la modifica

**Termine di momento** fa 2 operazioni:

* Accelera la discesa del gradiente se la direzione è stabile (aumento o diminuisco il peso per più iterazioni)
* Stabilizza 🡪 se ci si trova in prossimità del minimo e senza il momento ci sarebbero delle oscillazioni

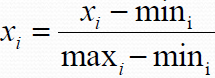
Algoritmo di backpropagation eseguito in **due modi**:

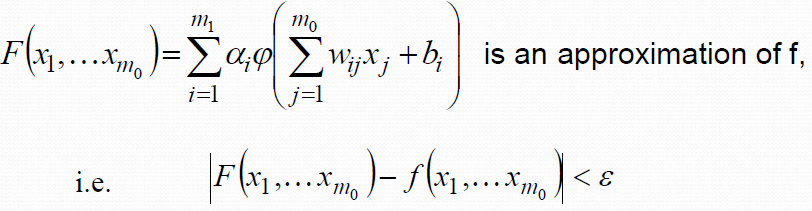
1. **Per pattern** 🡪 modifica i pesi secondo la direzione della discesa del gradiente dopo aver presentato un pattern 🡪 importante quando viene fatto l’update 🡪 ovvero quando arrivano gli input (boh porcodio)
2. **Per epoche** 🡪 i pesi sono modificati solo dopo che tutti gli esempi sono stati processati (memorizza la possibile correzione per l’esempio corrente, ma non la faccio) 🡪 mano a mano si somma la delta wji di ciascun esempio 🡪 **troppo lento!!** 🡪 gli esempi sono tutti prima di modificare i pesi, quindi si analizza ogni esempio con gli stessi pesi degli altri

**Criteri di stop** (perché non si ha un teorema di convergenza)

* **errore quadratico medio totale**: si considera che la backpropagation ha raggiunto la convergenza nel momento in cui l’errore quadratico medio sia abbastanza piccolo (es. nel range [0.1, 0.01])
* **criterio di generalizzazione**: dopo ogni epoca, si può testare la generalizzazione della rete: anche in questo caso si rientra in una fase di validation test, in cui una parte del training set non viene usato per l’aggiornamento dei pesi ma per testarne la generalizzazione. Se quest’ultima è accettabile, allora ci si può fermare con il training 🡪 cerca di evitare l’**overtraining:** situazione in cui la rete si specializza troppo sugli esempi di training e non ha buone prestazioni e performance nel validation test

**Design della NN e per il suo funzionamento**

* **rappresentazione dei dati**: gli attributi sono solitamente continui con potenzialmente diversi range 🡪 **normalizzazione** facendo sì che tutti gli attributi stiano nel range [0,1] 
* **Topologia della rete:** il numero dei livelli e di neuroni dipende dal tipo di task che si vuole eseguire. Si possono fare due operazioni generiche (o una o l’altra):
  + si parte da una rete sovradimensionata per poi rimuovere a mano a mano livelli e neuroni finchè le performance non calano 🡪 **pruning**
  + si parte da una rete piccola e si aggiungono neuroni fino a raggiungere performance soddisfacenti
* **Parametri**: tutto ciò che non è definito a priori (#neuroni, #esempi per training set, ecc) 🡪 ce ne sono molti con varie differenze tra loro
  + Iper parametri 🡪 come learning rate e termine di momento: devono essere dati, ma **non sono oggetto di addestramento**
  + In genere 🡪 **pesi** generalmente tra -1 e 1 ed è importante che ce ne siano di positivi e di negativi.
  + Learning rate dipende dalla sua applicazione, solitamente tra 0 e 1 (estremi esclusi)
  + #esempi di addestramento 🡪 legata al numero di pesi
* **Training** 🡪 consiglia come scegliere il set di esempi. 2 regole generali:
  + Regola del pollice: #training examples = almeno da 4 a 10 volte il numero dei pesi
  + 🡪 dove |W| = numero pesi e a = accuratezza che ci si aspetta sul test set
* **Validità / applicabilità** 🡪
  + Funzioni booleane 🡪 possono essere rappresentate TUTTE con una NN con un singolo livello nascosto
  + Funzioni continue
    - Limitate 🡪 approssimate con un errore arbitrariamente piccolo con una NN con un livello nascosto
    - In generale 🡪 approssimate con un accuratezza arbitraria con una NN con 2 livelli nascosti

**Approssimazione fatta da una FNN 🡪 teorema** 🡪 pagina 14 in poi. Qua metto solo la formula

Questo teorema prova l’esistenza di una approssimazione, ma non fornisce i valori dei parametri. F = NN, f = dati

Il teorema **dimostra** che un singolo livello nascosto per un MLP (Multi Layer Perceptron) è sufficiente per calcolare un’approssimazione uniforme a un dato insieme di training rappresentato dagli m0 input.

L’errore della discrepanza tra F e f 🡪 funzione decrescente al crescere di m1 (= numero di neuroni nel livello nascosto) 🡪 O(1/m1)

**Applicazioni di una FNN** 🡪 2 macro-classi:

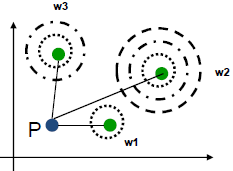
1. **Classificazione, regressione, riconoscimento di pattern e diagnosi**: per risolvere problemi reali non linearmente separabili:
   1. Classificare scritte a mano
   2. Riconoscimento di immagini (come quello facciale), riconoscimento vocale
   3. Classificazione di oggetti
   4. Analisi di segnali per determinare natura e origine
2. **Previsione / forecasting**: NN di tipo ricorrente (solitamente) 🡪 imparano dipendenze temporali, ovvero funzioni i cui input sono sequenze di misurazioni nel tempo

# RETI NEURALI A FUNZIONI RADIALI (rbf)

Sono simili agli MLP, ma in un livello nascosto possiede una funzione di attivazione/trasferimento particolare 🡪 a **simmetria radiale**

La zona individuata dalla simmetria radiale è una superficie chiusa

**Funzione radiale**: i valori forniti in uscita dipendono dalla distanza di un determinato input (quello processato in quel momento) rispetto ad un vettore immagazzinato dalla rete = “recettore di una classe”

**vettore della funzione radiale** 🡪 rappresenta il centro per la funzione radiale

**P =** pattern/vettore di ingresso

**wn = neurone radiale** 🡪 ha un centro (in verde) e attorno ha le **curve di livello**: proiezioni delle curve sulle quali l’uscita del livello hidden dà lo stesso valore 🡪 sono concentriche

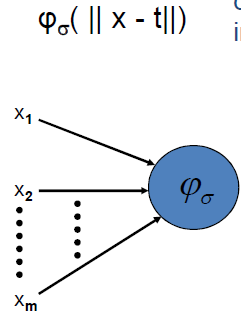
**Funzioni radiali assimilabili a campane** 🡪 punto più alto in corrispondenza del centro e decrescono (simmetricamente) a mano a mano che ci si allontana dal centro

Ogni funzione radiale **ha un centro t**

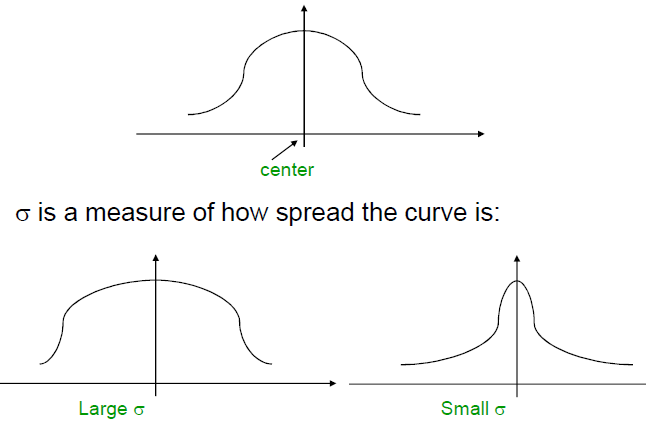
**Architettura RBF** 🡪 vi è un solo livello hidden i cui neuroni hanno **funzioni di attivazione radiali**. I valori in uscita da questo livello vengono pesati tramite i pesi per creare i valori di uscita per il livello di uscita (con tutti i suoi neuroni)

**Ingresso al livello hidden =** ingresso funzione radiale 🡪 diverso rispetto a quello di MLP 🡪 è **una distanza euclidea tra vettore x** (vettore/pattern in ingresso) e **un vettore che rappresenta il centro di una funzione radiale**

Parametri oggetto di apprendimento = i pesi w

**Iperparametri = centri della funzione radiale e larghezza (o spread)**

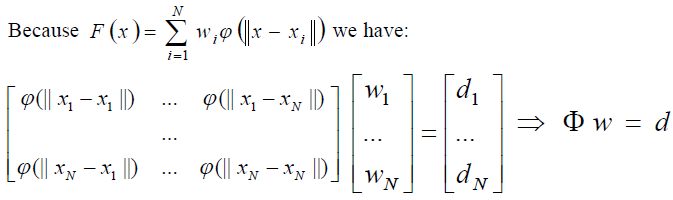
Neurone nel livello hidden 🡪 vi è un centro ed ha una larghezza = (associato al concetto della larghezza della campana)

Questo neurone è **più sensibile ai dati che si trovano ad una distanza piccola dal suo centro**

**RBF** permette di approssimare una funzione e risolvere problemi non linearmente separabili (in modo diverso da quello usato da MLP)

In RBF i neuroni di uscita fanno somma pesata di funzioni radiali (che sono non lineari)

**Interpolation problem**: dato un insieme di **N punti distinti** (x1, … xn) multidimensionali (hanno m coordinate) e un insieme di **N numeri reali** (target d1, …, dn) 🡪 una RBF consente di trovare una funzione F che valutata per un insieme di esempi di addestramento fornisce esattamente in uscita l’insieme d (target): 🡪 **condizioni di interpolazione** 🡪 (ovvero che dato il punto xi trova il target corrispondente)

Questa formalizzazione permette di capire come determinare i pesi w =**algoritmo di apprendimento**

Moltiplicare le due matrici = **calcolare somma pesata**

**Uso matrici** 🡪 notazione compatta: 🡪 **prodotto della matrice fi grande (=** ) **per il vettore** (o matrice colonna) **dei pesi deve essere uguale a d** (= matrice colonna di tutti i target desiderati)

Per sfruttare al meglio questa formula 🡪 **la matrice fi grande deve essere invertibile:** idealmente si deve dividere d per fi grande per ottenere w (ma sia fi che d sono matrici 🡪 dividere per una matrice = moltiplicare per la sua inversa)

**Teorema di Michelli** 🡪 garantisce che se gli esempi di training sono distinti (solitamente lo sono), allora la matrice fi è invertibile

Esempio problema XOR da pag 9 a 11 (da mettere in parallelo con la risoluzione proposta da MLP)

**Cosa si deve imparare per una RBF?**

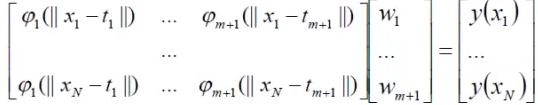
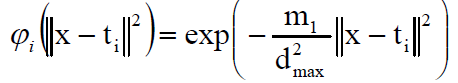
* Centri delle funzioni di attivazione
* Le larghezze delle gaussiane (campane) delle funzioni di attivazione
* I pesi da livello nascosto a livello di uscita

Ci sono **diversi algoritmi di apprendimento per RBF**

1. **Tramite pseudo-inversione**

Metodi di identificazione di centri e larghezze:

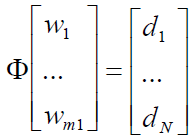
* + Centri = selezione randomica a partire dal training set
  + Spreads = scelti tramite normalizzazione 🡪

**Funzione di attivazione = gaussiana**: 

Questo prodotto di matrice riassume tutta le RBF 🡪 se voglio fare che sia oggetto di addestramento 🡪 **per ogni esempio la sovrapposizione lineare di w** (credo sia il prodotto) mi fornisca esattamente il valore di target di

**PORCAMADONNA**

Si deve quindi fare:

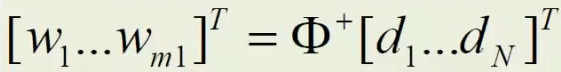






Qua c’è un problema: fi grande non è quadrata e quindi non invertibile 🡪 come si risolve? **Matrice pseudoinversa**

**Matrice pseudoinversa:** permette di ottenere la matrice identità (ma che vor dì?) 🡪 permette di ottenere i pesi come sotto (a destra) 🡪 questa matrice pseudoinvers richiede (solitamente) di memorizzare l’intera matrice



**🡪**

**Quindi passi di questo metodo:**

* + 1. Scelta dei centri in maniera randomica
    2. Usare la normalizzazione per computare la larghezza
    3. Trovare i pesi usando il metodo della pseudo-inversione

1. **Calcolo dei centri**

Metodi di identificazione di centri e larghezze

* + Centri: sfrutta un **algoritmo di clustering**
    1. Inizializzazione (random) dei centri
    2. Seleziona un esempio x dallo spazio degli input
    3. Si trova l’indice del centro che è più vicino a x:
    4. Aggiornamento dei centri 🡪 aggiungendo una quantità basata su learning rate
    5. Continua a fare i passaggi precedenti per tutti gli esempi

Questa procedura permette di spostare i centri dello spazio radiale in modo da metterne dove ci sono più esempi di training 🡪 rende le funzioni linearmente separabili

* Larghezza: scelta sempre tramite normalizzazione
* Pesi: tramite LMS usato in Adaline 🡪 possibile perché RBF rende le funzioni linearmente separabili spostando i centri nello spazio

RBF VS FFNN:

* **Entrambe** sono esempi di reti FF non lineari
* **Sono entrambi** approssimatori universali
* **Architettura diversa**
  + RBF 🡪 hanno un solo livello nascosto
  + FFNN 🡪 possono avere più livelli nascosti
* **Modello neuronale diverso**
  + RBF🡪 neurone in hidden diverso da neurone in uscita
  + FFNN 🡪 neurone hidden = neurone uscita (solitamente)
  + RBF 🡪 livello hidden = non lineare / livello output = lineare
  + FFNN 🡪 entrambi i livelli (hidden e uscita) = non lineari
* **Funzioni di attivazione diverse**
  + RBF 🡪 sfrutta una distanza euclidea tra vettore di input e il centro di un’unità
  + FFNN 🡪 sfrutta un prodotto interno di un vettore intero e il vettore dei pesi del neurone
* **Approssimazione diversa**
  + RBF 🡪 i neuroni costruiscono approssimazioni locali (si attivano se l’esempio è in un cerchio centrato sul punto di massimo)
  + FFNN 🡪 approssimazione globale perché crea un decision boundary

# 5. reti neurali extreme learning machines - elm

Sono NN particolari, molto usate in estremo oriente. Inizio anni 2000, continuano ad essere studiate soprattutto dalla comunità cinese ed orientale

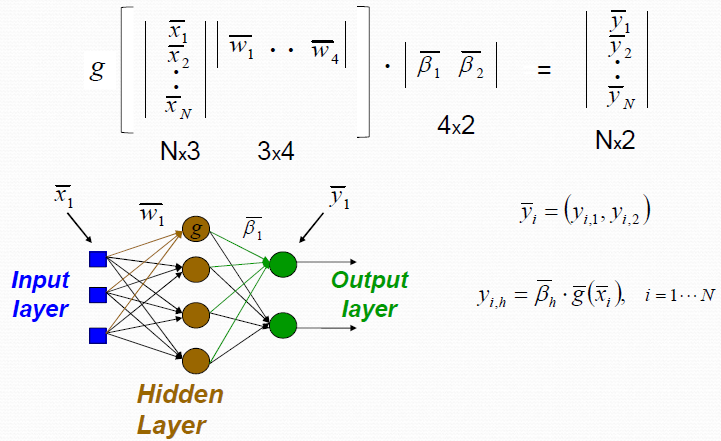
**Principio su cui si basa:** rete MLP strutturata come una radiale (ingresso, un hidden, output) ma che sfrutta funzioni sigmoidi (o in generale funzioni infinitamente differenziabili) 🡪 allora si possono settare a random i pesi del primo livello e poi calcolare direttamente i pesi di secondo livello (di uscita) sfruttando la pseudo-inversione 🡪 anziché sfruttando la backpropagation (che è efficace, ma molto lento)

**C’è un problema**: selezionando a random i pesi in ingresso, la soluzione fornita per beta (pesi di secondo livello) è funzione delle scelte fatte per w1

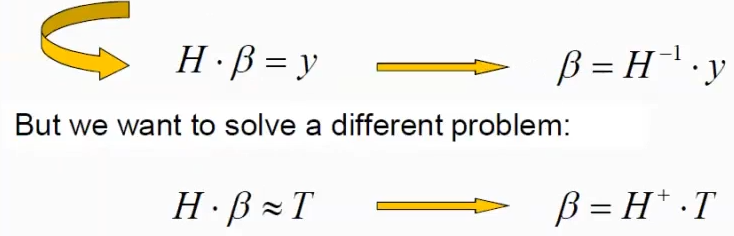
**Soluzione** = computare più volte per cambiare le scelte fatte per w1 (50/10/100, non si sa quante di preciso 🡪 scelta dello scienziato).

**Meno iperparametri** 🡪 no (= no learning rate), no termine di momento

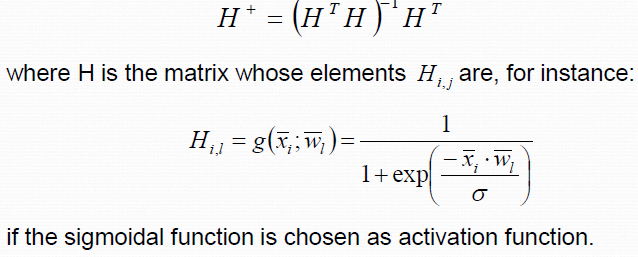
**Però c’è la pseudo-inversione 🡪** le matrici devono essere ben invertibili = non essere quasi singolari (**matrice singolare** 🡪 **determinate = 0**) 🡪 se è matematicamente invertibile, ma il determinante è quasi zero 🡪 rete instabile

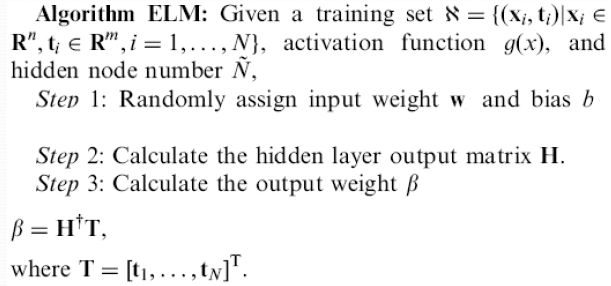
**Determinante matr quadr:** numero calcolato secondo il **teorema di Laplace**

**Il problema viene impostato per poterlo scrivere come operazione matriciale**

Per comodità 🡪 H = matrice con funzioni di attivazione sigmoidi relative a tutti gli esempi e ad un certo numero di neuroni hidden = matrice g nella figura sopra

deve essere circa **T = vettore target**

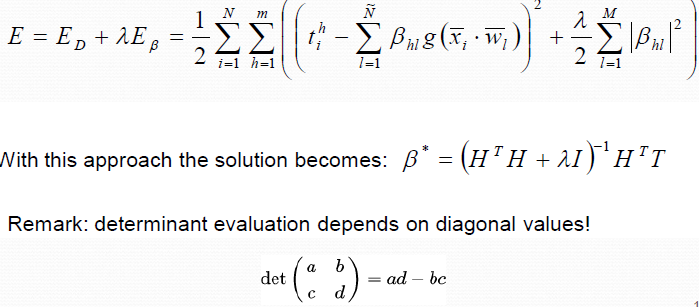
Ecco che entra in gioco la **pseudoinversa**

**Algoritmo di apprendimento**:

Procedura che va iterata più volte (minimo 50 volte) 🡪 pesante! Soprattutto perché l’inversione di matrice è dispendiosa e complessa

**Non funziona per tutte le funzioni infinitamente differenziali**

Come risolvere problemi dell’inversione di matrice (che non è sempre possibile) = determinante piccolo? 🡪 vale anche per le funzioni radiali 🡪 **regolarizzazione**

**Regolarizzazione**: non si concentra su ridurre l’errore di classificazione, ma **punta a ridurre un errore più ampio** 🡪 aggiunge un secondo termine (**lambda**): di seguito si pone la definizione dell’errore da minimizzare

Aggiungendo si evita che il determinante si avvicini a zero 🡪 questo perché il **determinante dipende fortemente dal valore della diagonale principale** (come da img sopra)

applicata alla matrice identità (1 su diagonale, 0 sul resto della matrice)

**Regolarizzazione** 🡪 porta ad un teorema: più piccola è la norma dei pesi, migliore è la performance che la rete tende ad avere

# 6. deep learning & CNN – rETI CONVOLUZIONALI

Deep learning nato a partire da MLP 🡪 introdurre più livelli, perfezionato backpropagation, modificato funzione di trasferimento

Le NN mai sviluppate fino ad una decina di anni fa per le difficolta di addestramento di 2 tipi:

1. dati per l’addestramento 🡪 GAFA (Google, Apple, Facebook e Amazon) possiedono i dati necessari, spesso li producono pure (es. ad ogni ricerca di Google).
2. hardware per l’addestramento 🡪 sviluppo alimentato dalla raccolta dei dati

**formazione di gruppi di ricerca al di fuori dell’ambito accademico** 🡪 ricerca informatica fino a 10 anni fa, sfruttava dati accademici e basta. Ora, invece, ci sono lab di alto livello che fanno ricerca in tutte le grandi azienda citate prima.

**Nuove librerie, hardware (nuove GPU), strutture di ricerca**

Big data + software + hardware = **VERA E PROPRIA RIVOLUZIONE**

**DNN = Deep Neural Network** – Reti neurali profonde

**Tanti livelli nascosti** 🡪 almeno 2, ma surclassati da modelli più profondi

**Livelli nascosti fanno operazioni diverse tra loro 🡪** organizzazione gerarchica

Perché servono? Permettono, grazie all’organizzazione gerarchica, di riusare e condividere le informazioni (simile a programmazione strutturata) 🡪 si possono selezionare determinate info/caratteristiche e scartare i dettagli non necessari 🡪 massimizzare l’invarianza

Con una rete a 1 o 2 livelli nascosti si può risolvere ogni problema, ma non in maniera efficiente

**Più livelli = rete più profonda = risoluzione di problemi complessi in maniera ottimale**

**Pro delle DNN**:

* **Representation learning:** caratteristiche importanti che fanno risolvere un problema sono trovate in maniera automatica (una volta invece lo facevano gli scienziati pre-processando i dati per trovare le feature migliori e più importanti 🡪 **apprendimento non supervisionato**
* Grazie a quello sopra 🡪 la rete neurale fa un addestramento attraverso tantissimi dati, quindi servono hardware molto potenti.

**Quanto deve essere profonda una rete? Non si sa** 🡪 ce ne sono anche con >100 livelli nascosti

Non vi è un metodo per capire quanti livelli siano corretti

**Distacco molto profondo tra reti profonde e cervello umano** (es. “solo” 12 livelli tra la retina dell’occhio e i muscoli che si attivano per prenderlo)

**Profondità = numero di livelli**

**Complessità** di una DNN dipende molti parametri tra cui profondità, numero neuroni, pesi, ecc.

**Maggiore è il numero dei pesi** (quindi delle connessioni), **maggiore è il numero di operazioni per il training** 🡪 perché addestrare = trovare e portare i pesi al loro valore ottimale

Modelli reti profonde FF:

* **CNN** 🡪 Convolutional Neural Network (ConvNet)
* **FCDNN** 🡪 completamente connesse 🡪 MLP con almeno 2 livelli nascosti
* **HTM**

**Modelli con apprendimento non supervisionato** 🡪 apprendimenti di tipo clustering, la rete organizza la struttura per fare diagnosi di similarità. Ce ne sono diversi: **auto-encoders, RBM, DBN**

**Modelli ricorrenti** 🡪 una parte del segnale prodotto viene riportato in ingresso: **RNN, LSTM**

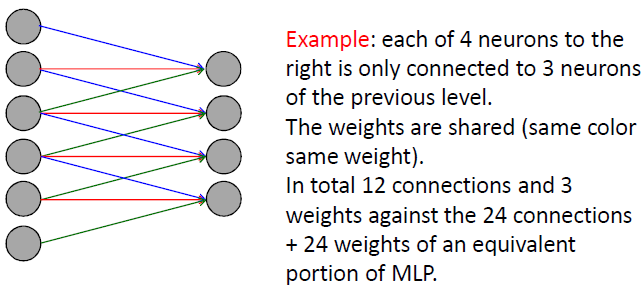
**3 vantaggi delle DNN**:

* Big Data
* GPU Computing 🡪 passare da training di mesi a solo qualche giorno
* Modelli profondi fatti ad hoc

**Prima rete CNN nel 1998** 🡪 mancavano dati e hw necessari 🡪 usata solo per problemi di piccole dimensioni

**Problemi uso funzioni sigmoidi** 🡪 scomparsa e esplosione del gradient 🡪 si passa a funzioni **relu**

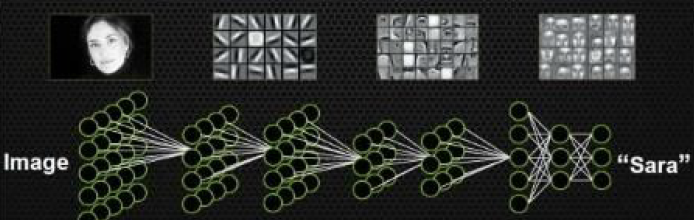
**Differenze dal MLP**:

* **Processing locale:** NO connessione completa 🡪 i neuroni sono connessi solo ad alcuni neuroni del livello precedente 🡪 ogni neurone effettua un **processing locale** (vede solo una parte del livello precedente) 🡪 riduzione enorme del numero di connessioni
* **Pesi condivisi:** neuroni di un livello sono collegati ad alcuni neuroni livello precedente 🡪 i neuroni di uno stesso livello processano tramite gli stessi pesi i neuroni del livello precedente. Vedere figura sotto per entrambe le caratteristiche (processing locale e pesi condivisi)

**Caratteristiche DNN:**

* **Esplorazione locale** dei livelli precedenti a cui un neurone è collegato
* **Condivisione dei pesi** (base del principio di convoluzione) 🡪 unico vettore di pesi
* **Imparare simultaneamente più filtri convoluzionali** 🡪 se si ha 1 livello hidden in cui tutti i neuroni processano con gli stessi pesi i neuroni del livello precedente equivale ad applicare un filtro (sempre lo stesso) a zone differenti dell’immagine

**CNN usate principalmente per processare img**

**Architettura CNN** 🡪 livelli gerarchici

Da sinistra:

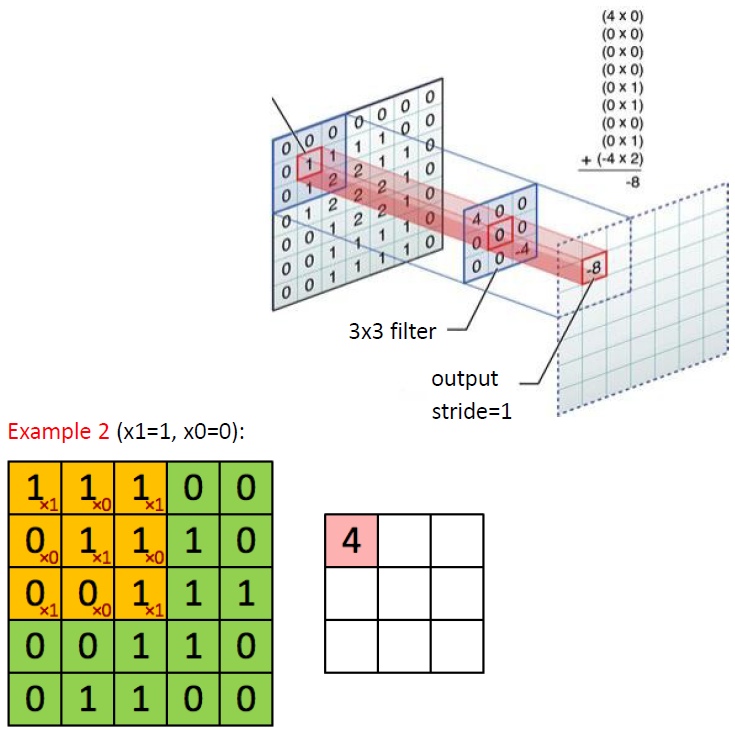
* Livello di input = immagine da analizzare 🡪 matrice dei pixel dell’immagine
* Livelli convoluzionali 🡪 connessioni locali 🡪 trovano caratteristiche di complessità sempre superiore 🡪 zone della rete si concentrano su porzioni differenti della img
* Ultimi livelli 🡪 classificano l’immagine processata 🡪 assimilabili ai livelli degli MLP

Questa struttura consente di trattare in maniera differente, diverse porzioni di una img tramite raccolta dati locale e condivisione dei pesi delle connessioni che caratterizzano i neuroni che si stanno usando

**Livello Convoluzionale di una rete CNN**

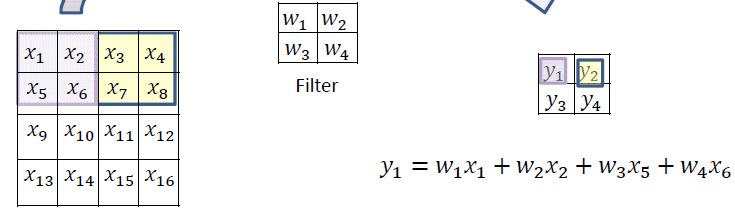
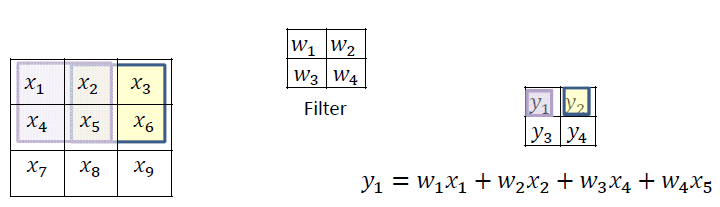
**Obiettivo**: applicare filtri digitali alle immagini

**Livello di ingresso** 🡪 matrice

**Ciascun neurone** applica un filtro 2D (matrice di pesi 🡪 inizialmente random) per generare un numero di output dato dal prodotto scalare tra la maschera 2D e la porzione coperta da tale machera della matrice di input

**Meccanismo di filtraggio:** si deve definire la dimensione del filtro e i valori dei pesi dentro alla maschera/filtro 🡪 si moltiplica poi per le corrispondenti dell’immagine di ingresso

**Stride = parametro che indica di quanto ci si sposta nell’immagine di input**

**Img sottostante: convoluzione 2D**

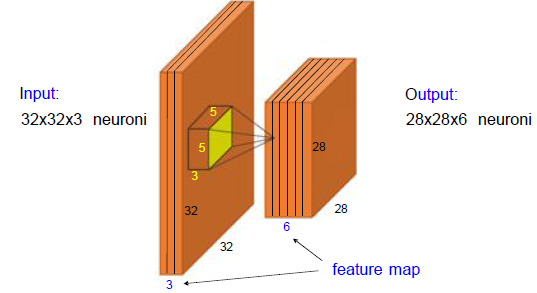
Cambio pesi per effetto dell’addestramento 🡪 cambiano tutti allo stesso modo 🡪 dopo un tot di epoche la maschera sarà differente per tutti i neuroni a quel livello

**Più livelli convoluzionali = diverse caratteristiche calcolate/catturate** 🡪 ogni livello ha neuroni che gestiscono maschere diverse

**Padding** 🡪 risolve il rischio che si ha quando si può trasbordare dall’immagine inserendo uno stride che fa uscire dai bordi 🡪 crea un bordo attorno all’immagine con un valore sempre uguale adeguato all’immagine e allo stride

**CNN** ideate per cogliere aspetti rilevanti delle immagini a cui sono dedicate

**Convoluzione 3D**

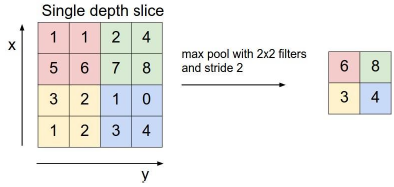
Si ha quando la matrice input assume una terza dimensione 🡪 ad esempio quando si considera una img a colori in cui ogni pixel è caratterizzato dai 3 valori RGB

In questa img 🡪 5x5x3 = 75 **valori di neuroni** presi sul totale della img di input

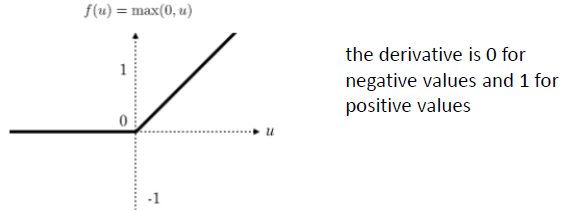
Questi 75 valori vengono pesati da un filtro a 75 dimensioni per creare un singolo neurone nel successivo blocco (o livello) convoluzionale 🡪 quest’ultimo blocco è formato da 6 livelli 28x28 🡪 in ciascuno di questi 6 livelli 🡪 tutti i neuroni processano 75 neuroni di ingresso tramite lo stesso set di 75 pesi (ovvero lo stesso filtro) 🡪 livello = **feature map**

**Livello/blocco di pooling** 🡪 genera feature map più piccole 🡪 crea aggregazione di informazioni

**Obiettivo:** dare invarianza rispetto a trasformazioni semplici (es. traslazione dell’ingresso) 🡪 **mantiene l’info necessaria per la discriminazione**

**Questo blocco con tanti livelli di pooling** 🡪 ognuno di essi prende una parte del blocco convoluzionale e la processa fornendo in uscita o la media (**average pooling**) o il livello massimo **(max pooling)** dei valori dei neuroni del livello precedente che legge

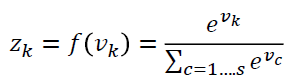
**Terzo livello: funzione ReLu**

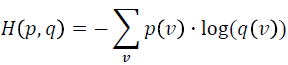
Processa tramite la funzione di trasferimento i valori posizionati su ogni neurone del blocco di pooling 🡪 nelle reti profonde questa funzione di trasferimento ha sostituito la sigmoide 🡪 **eliminato il problema di eliminazione o esplosione del gradiente 🡪 funzione ReLu (Rectified Linear)**

**Funzione ReLu** 🡪 andamento sempre crescente per i valori positivi

Se il neurone vale 0 🡪 derivata 0 🡪 non potrei fare backpropagation 🡪 il neurone si **considera spento** 🡪 info retro-propagata = info inutile

**Softmax** 🡪 funzione attribuita ai neuroni dell’ultimo livello

Ogni neurone k elabora un valore zk che è una funzione del suo campo di ingresso 🡪 **i valori di zk sono normalizzati** grazie alla struttura della formula 🡪 **la somma dei vari z, quindi, uguale a 1** 🡪 permette di vedere z come una sorta di probabilità 🡪zk serve a stabilire che l’oggetto in ingresso ha una probabilità zk di appartenere alla classe k

**Cross-entropy** (al posto di mean squared error degli MLP)🡪 funzione di perdita che valuta la differenza tra due distribuzioni 🡪 misura quanto la distribuzione in uscita è differente da quella che ci si aspetta 🡪 **misura da minimizzare nell’addestramento**

**Esempio ben fatto pag. 22 slides 6** e a seguire

**Training di modelli complessi** 🡪 richiede tempo macchina elevato (giorni o settimane)

**Classificazione post training** 🡪 molto molto rapida: 10-100 ms

**Fine-tuning** 🡪 sfrutta una rete pre-addestrata su un problema simile, si rimpiazzano alcuni livelli e si ri addestra la parte convoluzionale della rete 🡪 in questo modo vengono imparate nuove cose

**Re-using features** 🡪 si usa una rete addestrata senza ulteriore fine-tuning, si estraggono le features generate dalla rete durante i passi in avanti sul nuovo dataset. Si usano queste features per allenare una SVM o altri classificatori