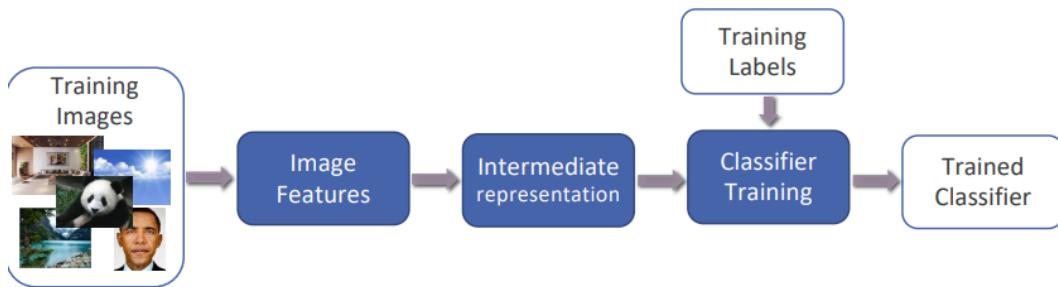


Lezione 6 - CNN - 11/11/2024

Convolutional neural networks



Fin ora abbiamo questo processo per il **pattern recognition**: le immagini, da cui prendiamo le features, possiamo poi usare optionalmente una rappresentazione intermedia (tipo bag of words) e poi trainiamo un classifier con dei labels associati e arriviamo al trained classifier.

La parte importante è che ogni data sample ha la sua descrizione, che è in un suo spazio dimensionale

A small image of a white flower on a black background is followed by a horizontal arrow pointing to the right. To the right of the arrow is the mathematical expression $f_i = f(x_i), f_i \in R^n$. Below the image is the text "i-th sample x_i ".

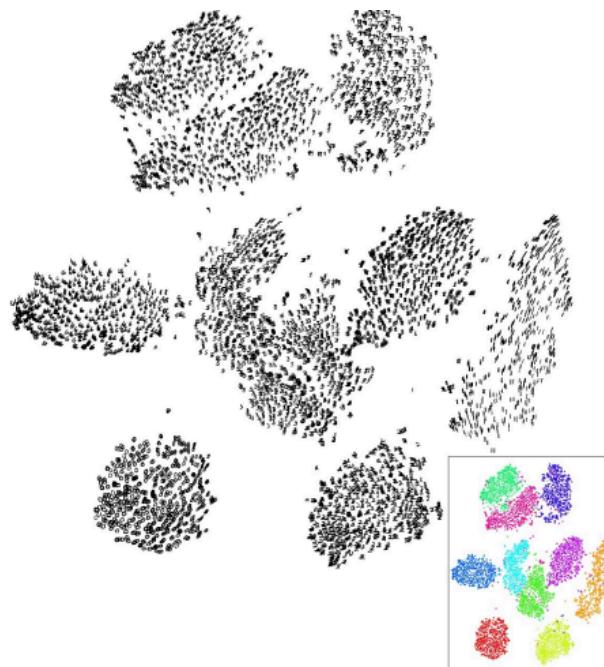
Visto che n può essere grande (vettori grandi) spesso si usano metodi che riducono la dimensionalità (PCA, ICA, ...), così il nuovo descrittore è mappato ad un nuovo spazio vettoriale.

Che **classifiers** possiamo usare?

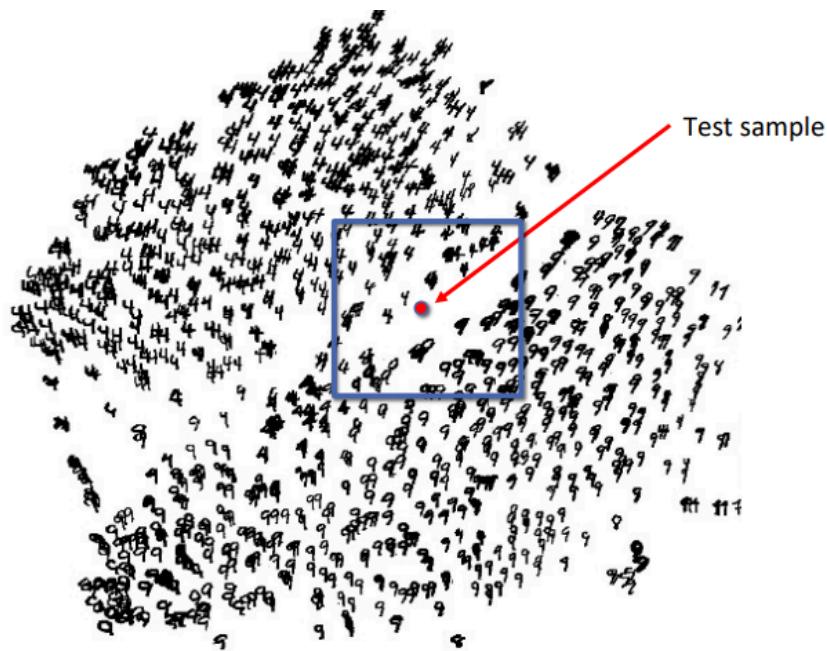
1-NN (nearest neighbour)

- All the descriptors are seen as points in the k-dimensional vector space
- A distance function is chosen to induce a topology in the space, e.g.:
 - Euclidean distance (L2)
 - Taxicab or Manhattan distance (L1)
 - Hamming distance
 - Earth mover distance
 - Chi-squared distance

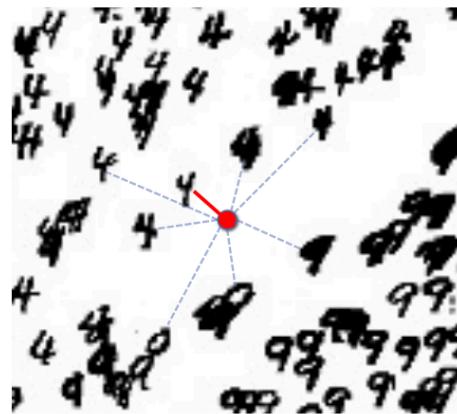
Cambia in base a come misuriamo le distanze



Questa è una rappresentazione del MNIST dataset in 2 dimensioni



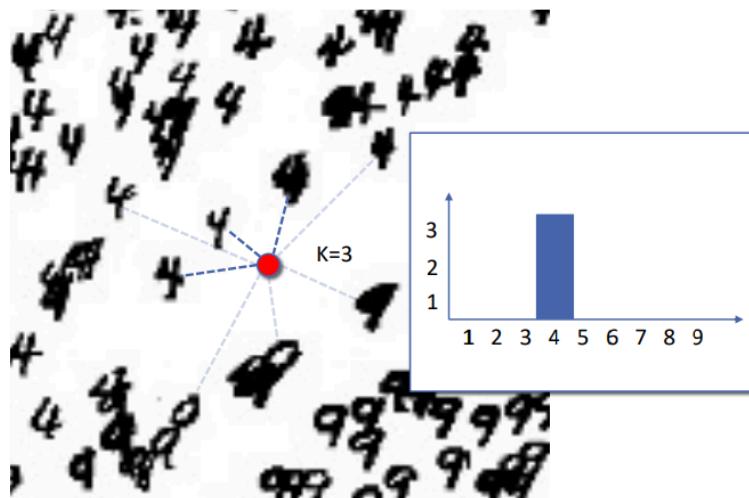
I 9 e i 4 sono molto vicine perché è facile confonderle. Ora diciamo che abbiamo questo test sample, come funziona 1-NN per fare la predizione?



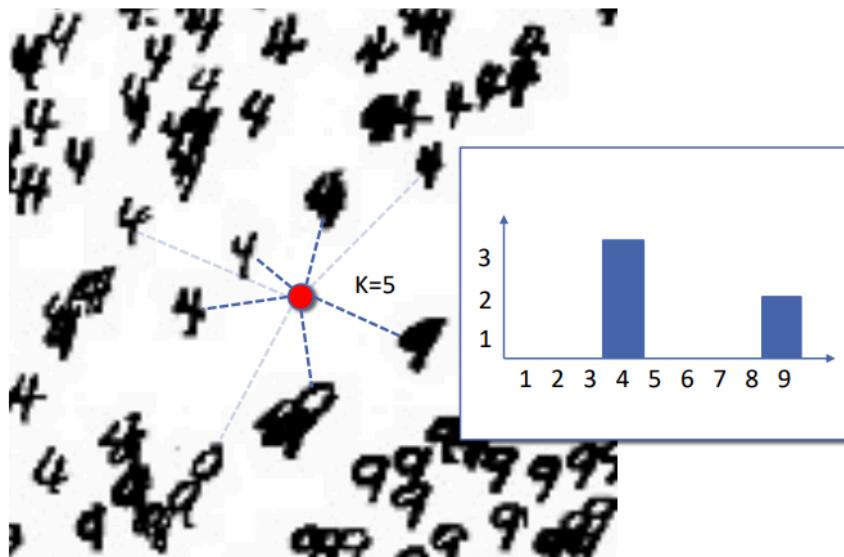
Va a vedere la distanza tra quelli intorno, e trova quella più corta, e sceglie quella classe come predizione.

k-NN

k-NN funziona allo stesso modo ma usa più di una distanza.



k dice quanti vicini consideriamo (i più vicini). Si va quindi a contare quanti esempi ci sono per ciascuna classe, così ciascuna classe avrà un confidence level percentuale. In questo caso 100% confidence sulla classe 4.

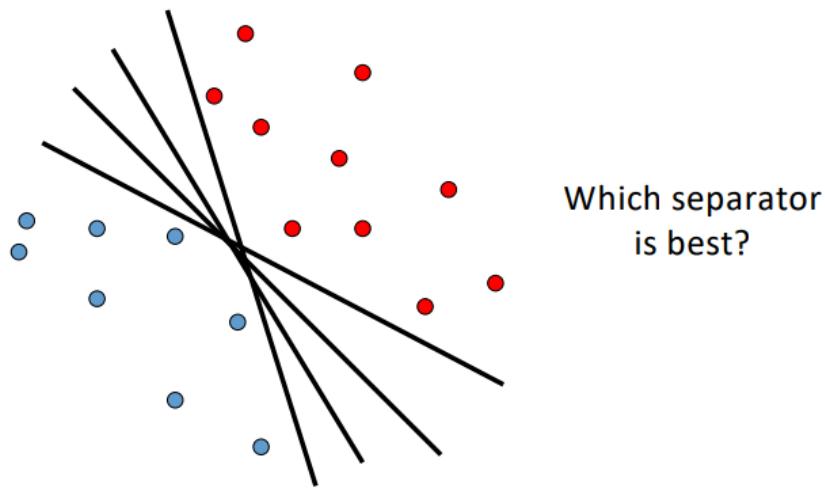


Qui abbiamo 60% di confidenza sulla classe 4 e 40% sulla classe 9, quindi ci dà più informazione perchè ci dice che non è molto confident. Questa informazione può essere utile, in base alla task.

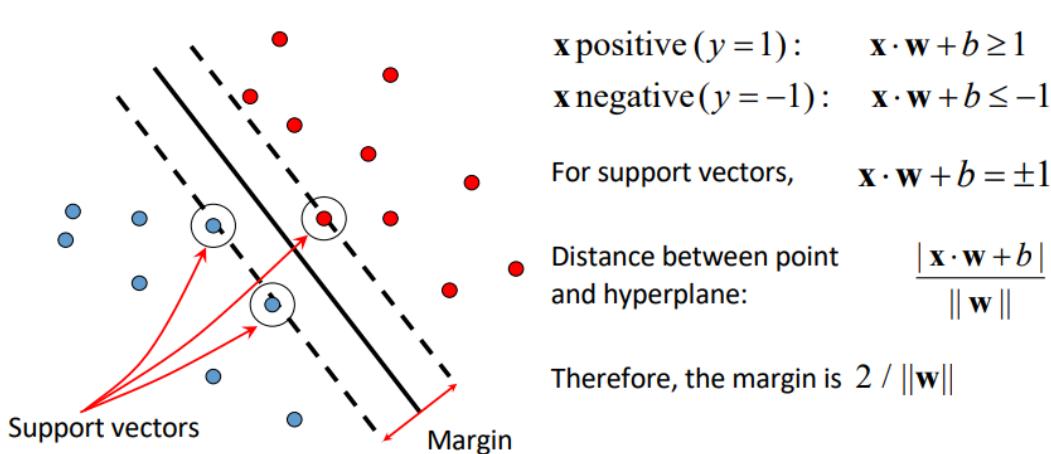
Perchè questo classifier è trainable? Perchè dobbiamo scegliere il parametro k e la distanza da usare, quindi possiamo considerarlo "trainabile". Quindi spesso facciamo una cross-validation per scegliere i parametri migliori.

SVM Support Vector Machine

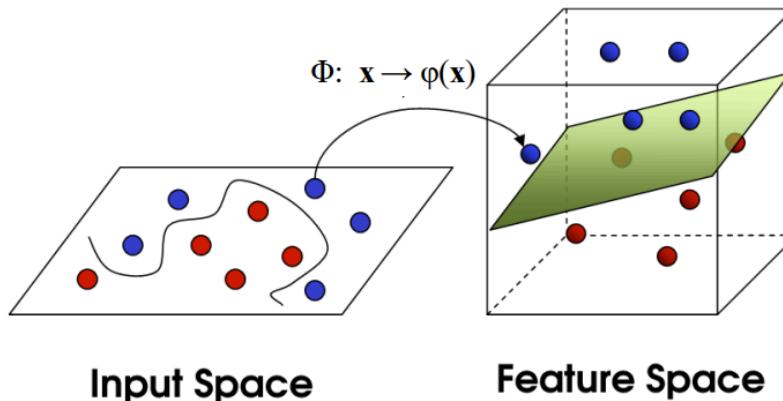
L'idea di questo tipo di classifier parte dall'osservazione che quando abbiamo 2 classi diverse, e vogliamo separarle con una linea, anche se i dati sono linearmente separabili, abbiamo una famiglia infinita di linee che separano i dati allo stesso modo. Quindi come facciamo a scegliere la linea migliore che separa i dati?



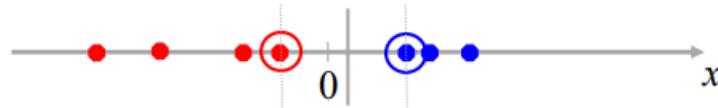
L'idea è quella di trovare la linea che massimizza il margine tra le due classi. La distanza tra le 2 linee tratteggiate è chiamato il **margine**. Quindi noi vogliamo massimizzare il margine.



Il **kernel trick** è usato per mappare i dati dell'input space, dove i dati possono essere non separabili, in un feature space dove sono linearmente separabili.



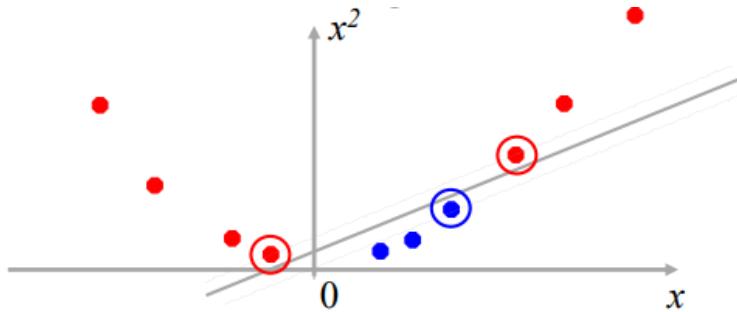
Questo è un esempio: in una dimensione dobbiamo trovare un punto di modo che una classe sia a destra e una a sinistra. Anche qui vogliamo scegliere il punto medio tra i punti più vicini di ciascuna classe, per avere il margine massimo.



In questo caso i dati non sono linearmente separabili in una dimensione. Quindi l'idea è quella di mappare questi dati in uno spazio di dimensioni più grandi.



Per esempio, facciamo la potenza di 2 di ciascun dato. In questo spazio, possiamo separare i dati linearmente.



Le SVM funzionano anche per separazioni **non lineari**. Abbiamo altri kernel come il kernel polinomiale (che il prof non ha visto usato in problemi reali), oppure il **gaussian kernel** (Radial Basis Function, **RBF**), che può funzionare bene tunando il sigma.

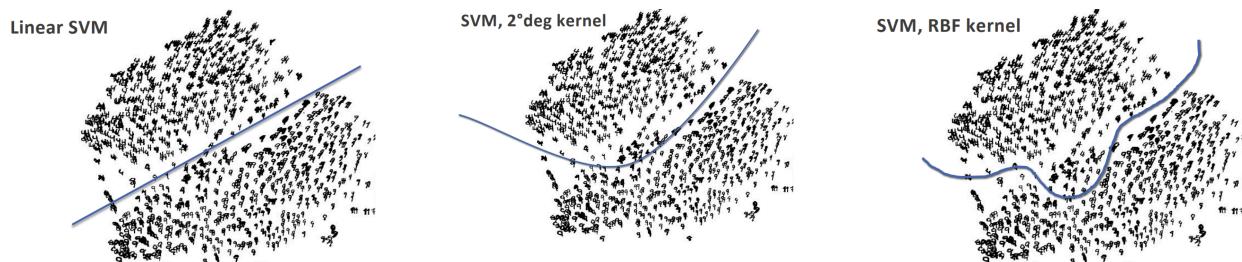
Instead of explicitly computing the lifting transformation $\varphi(\mathbf{x})$, define a kernel function K such that

$$K(\mathbf{x}, \mathbf{y}) = \varphi(\mathbf{x}) \cdot \varphi(\mathbf{y})$$

Polynomial kernel: $K(\mathbf{x}, \mathbf{y}) = (c + \mathbf{x} \cdot \mathbf{y})^d$

Gaussian kernel (a.k.a. Radial Basis Function, RBF):

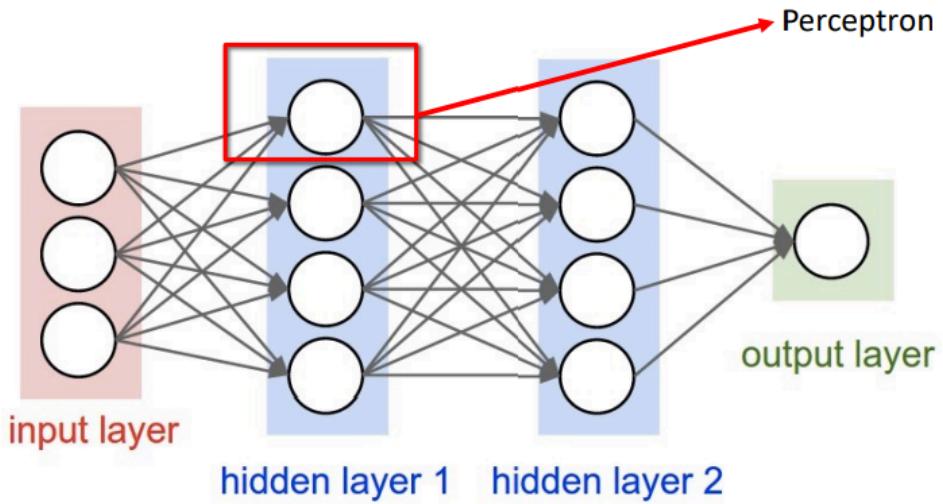
$$K(\mathbf{x}, \mathbf{y}) = \exp\left(-\frac{1}{\sigma^2} \|\mathbf{x} - \mathbf{y}\|^2\right)$$



Anche qui bisogna trovare i parametri migliori.

Neural network

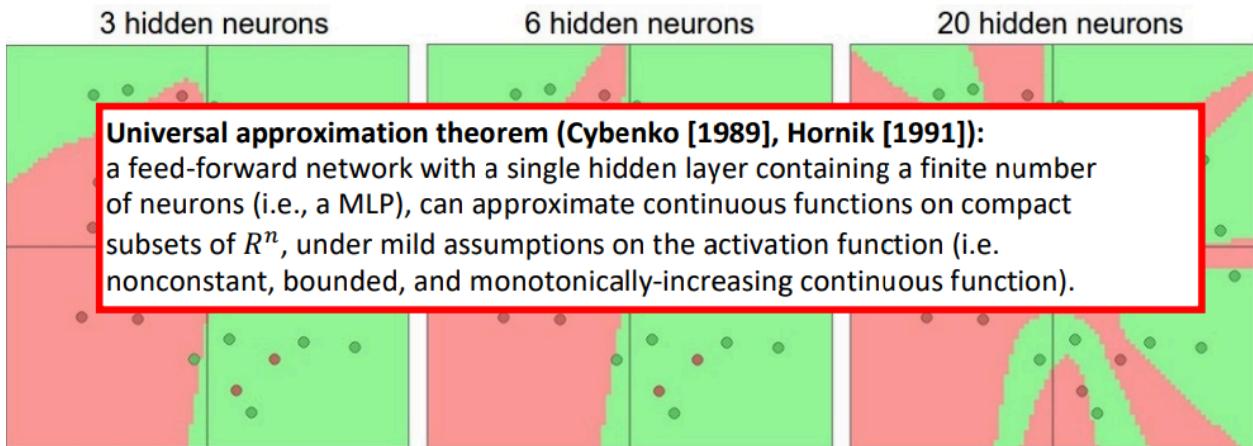
Come facciamo a creare dei classificatori non lineari, usando un percettrone (che è lineare)? Facendo un modello con più layers.



Aggiungendo più layers raggiungiamo classificatori più potenti.

Abbiamo delle proprietà interessanti su modelli che hanno un solo layer:

In base a quanti neuroni abbiamo in quel layer, possiamo aumentare la capacità del modello, ovvero la sua capacità di separare le classi. Con 3 neuroni il modello non è abbastanza non lineare, idem con 6 anche se migliora. Con 20 le decision boundaries sono molto più complessi, separano in multiple regioni, e ora il set è classificato correttamente.



Abbiamo questo teorema:

Non è un teorema di convergenza. Ci dice che abbiamo un numero finito di neuroni, per il quale questo è vero. Se abbiamo una funzione di attivazione non costante, bounded (quindi non va a +/- infinito), e monotonically-increasing

continuous, allora esiste un numero finito di neuroni che approssima la funzione. Dice semplicemente che questa rete esiste.

La strategia è quella di usare multipli layers, di modo che il classificatore sia non lineare.

Però questo teorema è importante, perchè per le SVM per esempio non esiste un teorema che dice che esiste un sigma che classifica correttamente.

Trovare il minimo di una funzione

Esempio di partenza:

Suppose you want to find the minimum of a function, what would you do?

$$f(x) = x^2$$

1. Calculate the derivative of $f(x)$ wrt x

$$\frac{df}{dx} = 2x$$

2. Set it to 0

$$\frac{df}{dx} = 2x \triangleq 0 \rightarrow x = 0$$

How would you build a system that given a function and its derivative, finds a minimum?

Facciamo la derivata, risolviamo per $x = 0$.

Calcolare la derivata di una funzione è semplice, abbiamo delle regole meccaniche. La parte difficile è risolvere per $x=0$. Può capitare che la derivata è molto complessa. L'idea è quella di non calcolare la soluzione, ma piuttosto costruire una cosa alternativa che man mano si avvicina alla soluzione.

IDEA: use the gradient to minimize the function

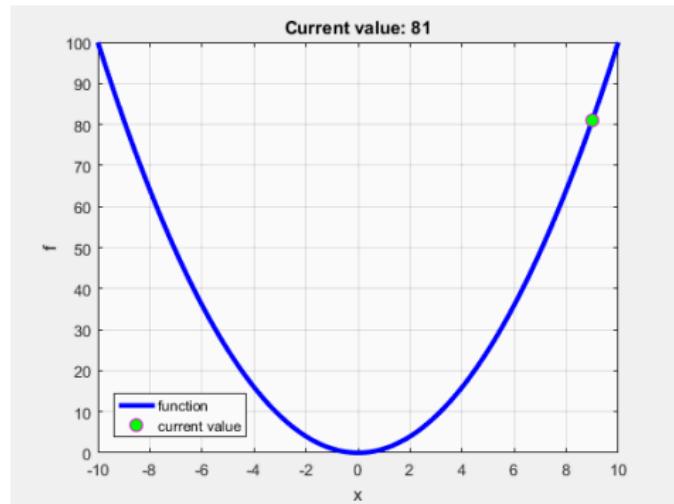
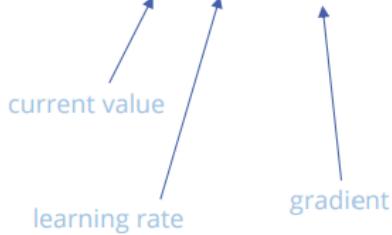
$$f(x) = x^2 \quad \frac{df}{dx} = 2x$$

1. Initialize x at a random value

2. while $f'(x) \neq 0$

3. update x using gradient

$$x_{t+1} = x_t - lr * \frac{df}{dx}(x_t)$$



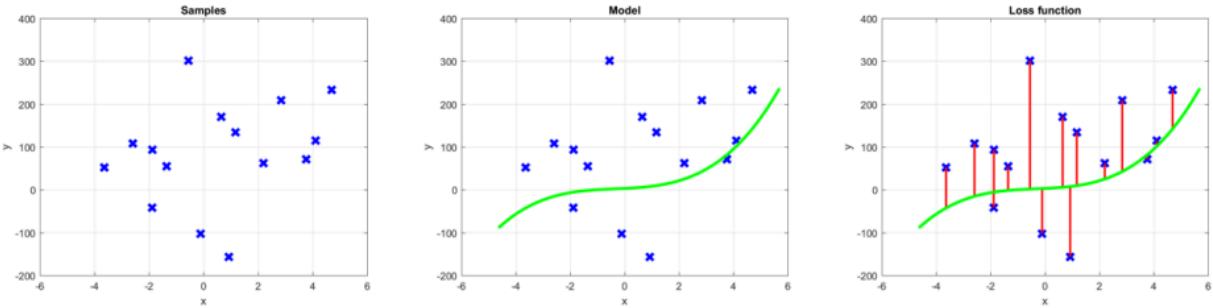
Quindi inizializziamo x a valori random, e finchè non arriviamo ad avere la derivata = 0, continuiamo ad aggiornare x in base al valore della derivata in quel punto, moltiplicato per il learning rate.

Questo è computazionalmente meno costoso di computarla direttamente.

Il learning rate diminuisce man mano che ci avviciniamo alla soluzione.

Quindi abbiamo il random value iniziale e il learning rate.

Supponiamo che ora vogliamo fissare questi punti, e vogliamo fissare un polinomio di terzo grado in questi punti. La loss function è la somma per tutti i punti, di quanto è diversa la ground truth rispetto alla nostra funzione. Quindi nel grafico è la somma delle linee rosse.



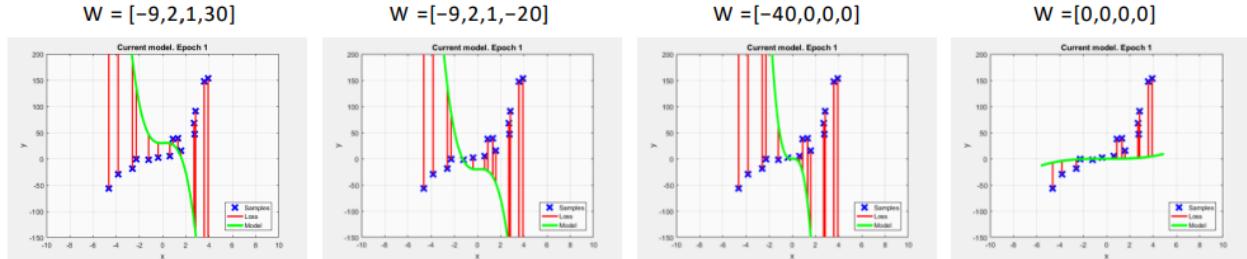
$$[(x_1, y_1), (x_i, y_i), \dots, (x_N, y_N)]$$

$$f = w_1 x^3 + w_2 x^2 + w_3 x + w_4$$

$$l = \sum_{i=1}^N (z_i - y_i)^2$$

In base al punto random scelto all'inizio, può succedere che i modelli convergano a velocità diverse, ma in questo caso tutte raggiungono lo stesso risultato. Quindi l'inizializzazione non è importante.

Different Initializations

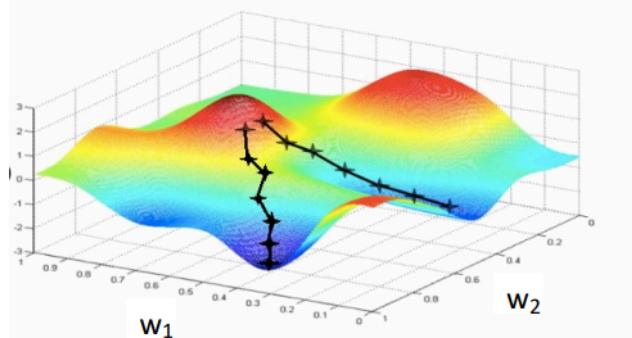


D'altro canto, il learning rate è molto importante, se è troppo grande la funzione può non convergere.

Però qui avevamo 4 pesi. Come facciamo ad ottimizzarli quando abbiamo milioni di pesi?

L'idea è la stessa, vogliamo minimizzare la somma delle differenze tra i valori reali e le predizioni, quindi aggiorniamo tutti i pesi facendo la discesa del gradiente, usando la stessa formula (derivata). Ora abbiamo un cambiamento di notazione, perché abbiamo una funzione multi-dimensionale, quindi usiamo delle derivate parziali.

Update weights by **gradient descent**: $\mathbf{w} \leftarrow \mathbf{w} - \alpha \frac{\partial E}{\partial \mathbf{w}}$



In particolare, per fare questa discesa del gradiente, dobbiamo calcolare la derivata di tutti i percetroni che abbiamo nella nostra architettura della rete. Oltre ai percetroni, abbiamo anche la non linearità, quindi abbiamo una restrizione del tipo di non linearità da applicare al neurone, perché dobbiamo poter applicare la discesa del gradiente.

In particolare, se vogliamo scrivere l'output del nn rispetto all'input, questo sarebbe una funzione molto non lineare, perché l'input passa per molti percetroni, quindi sarebbe una funzione composta molto grande. Per calcolare questa derivata, usiamo la back propagation. Iniziamo calcolando i gradienti dai layers vicini all'output, propagando l'informazione verso gli input. Questa si chiama chain rule.

$$E(\mathbf{w}) = \sum_{j=1}^N (y_j - f_{\mathbf{w}}(\mathbf{x}_j))^2$$

Il secondo ingrediente è l'uso della discesa del gradiente stocastica. Noi dovremmo calcolare le derivate rispetto a tutti i punti del dataset, per la discesa del gradiente, aggiornando solo dopo averli considerati tutti. Invece, usiamo un sottoinsieme chiamato batch, e li usiamo per avere multipli update in ciascuna epoca. Il punto importante è che la discesa del gradiente classica potrebbe rimanere bloccata in un minimo locale, perché usa tutti i punti. La discesa del gradiente stocastica invece, visto che usa un set più piccolo, e visto che questi batches sono random, può succedere che ad un certo punto ci sia una

configurazione che ci fa scappare dai limiti locali. Questo è un altro vantaggio, oltre al fatto che possiamo fare più updates.

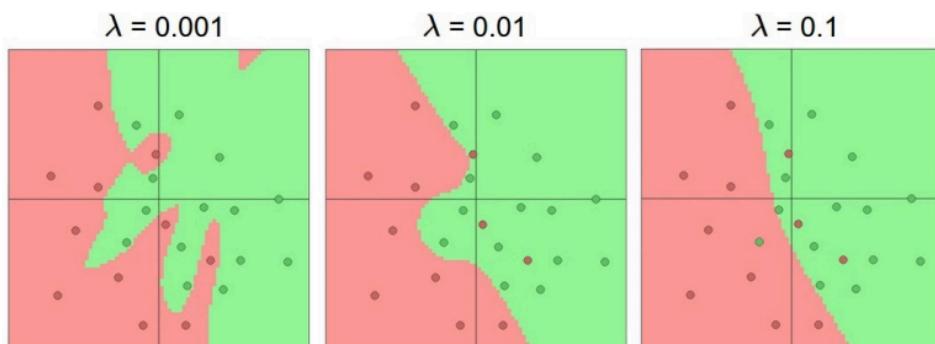
è comune avere un modello fisso, ma con pochi dati, e quindi c'è un grosso rischio di overfitting.

Come possiamo intervenire, per evitare l'overfitting (il modello fitta perfettamente il training set ma non riesce a fare predizioni).

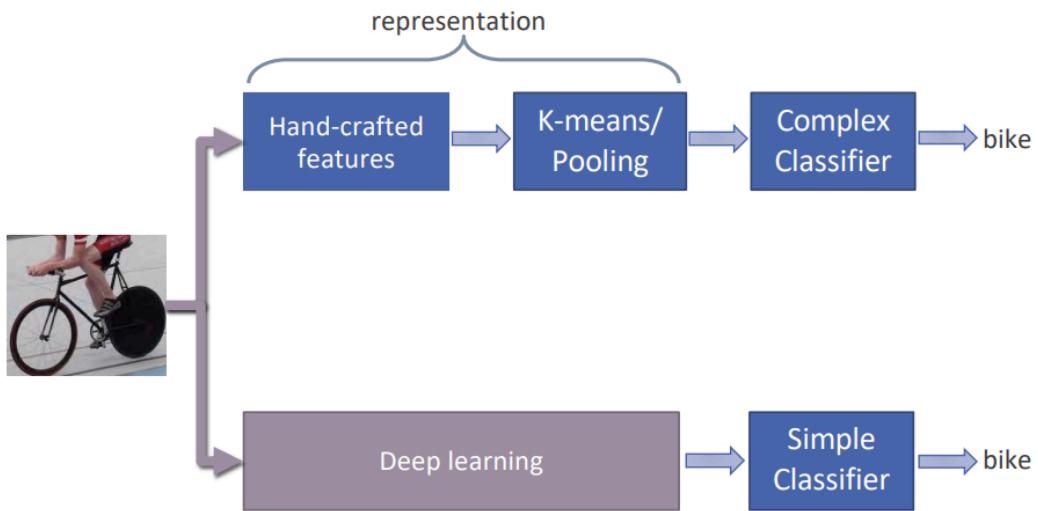
Un modo comune è di usare una penalità aggiunta alla loss function.

$$E(f) = \sum_{i=1}^N (y_i - f(\mathbf{x}_i))^2 + \frac{\lambda}{2} \sum_j w_j^2$$

Quando abbiamo pesi molto grandi, il modello segue molto bene i punti del training set (come nella prima immagine), di solito però vogliamo un modello più generale, come la seconda immagine. Questo è l'effetto del parametro lambda, ovvero quanto penalizziamo i pesi grandi.



Hand-crafted vs learned features



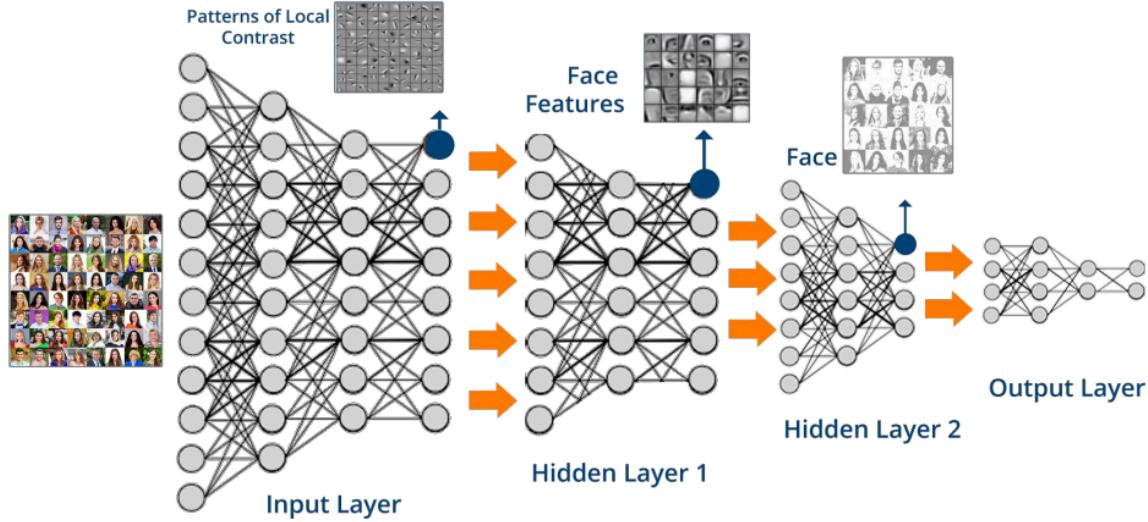
La parte di representation veniva fatta a mano. Ora abbiamo un cambio di paradigma, abbiamo il deep learning che sostituisce questa parte, e alla fine abbiamo un classificatore semplice, lineare.

Deep learning

Il deep learning è una nuova area di machine learning che lo avvicina all'artificial intelligence. Al giorno d'oggi quando senti AI, significa modelli di deep learning trainati. Non è corretto perchè AI in realtà è più broad, però è usato così dal pubblico generalmente.

Partendo dall'input layer, creiamo astrazioni sempre più grandi finchè raggiungiamo il layer di output. Ogni stage è una trasformazione non lineare, allenabile, del layer precedente.

Per esempio l'input è l'immagine, i primi layer estraggono patterns di contrasto locale, poi altri layers calcolano delle trasformazioni trovando delle features della faccia, poi altri layers calcolano un'astrazione che trovano le facce.



Questo funziona per tante tasks diverse

Image Recognition

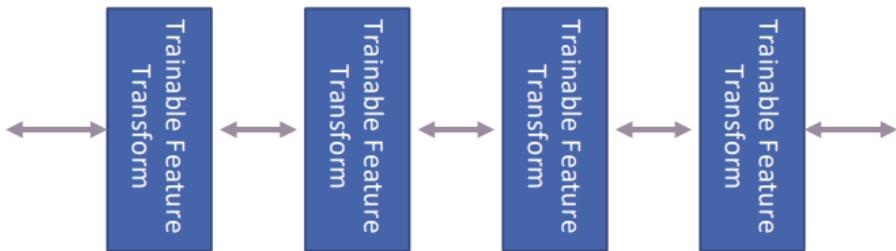
Pixel → Edge → Texton → Motif → Part → Object

Speech

Sample → Spectral band → Sound → ... → Phonema → Word

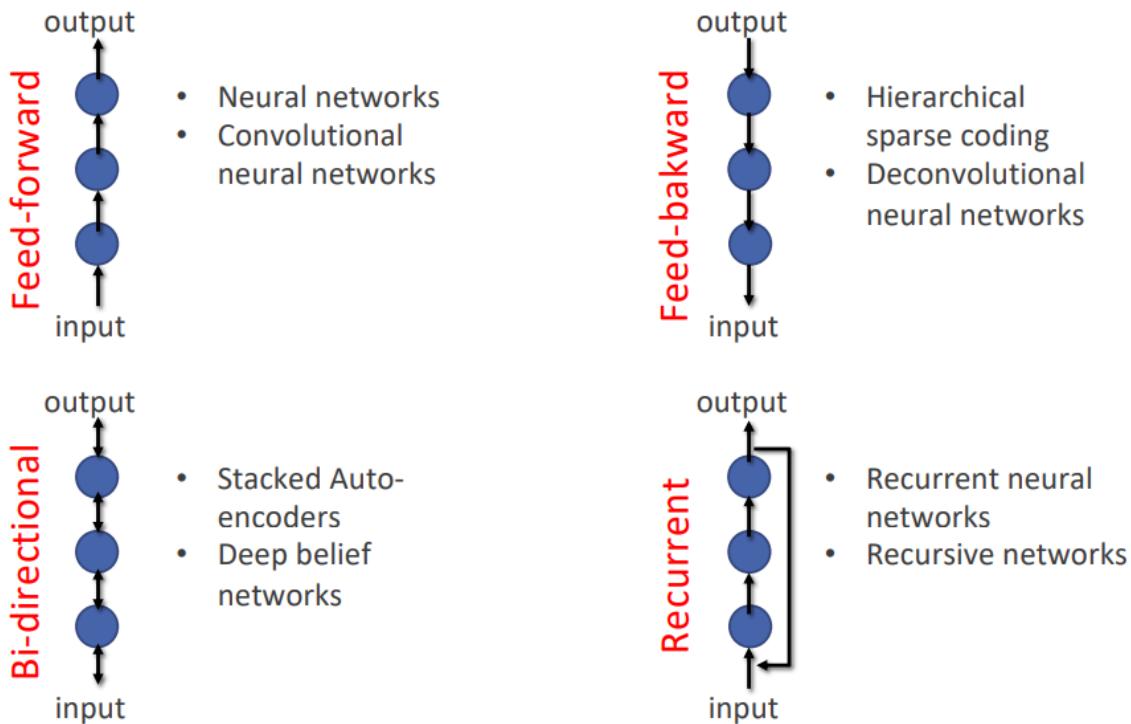
Text

Character → Word → ... → Clause → Sentence → Story



Abbiamo questa gerarchia per tutte.

Abbiamo 4 tipi di architetture deep.



Feed forward, l'informazione va in avanti, dall'input all'output, dove vogliamo fare una task tipo regression, classification. Le convolutional nn sono di questo tipo.

Faded-backward sono l'opposto, vanno dall'output all'input, nelle deconvolutional nn vogliamo per esempio decostruire l'output.

Bi-directional è un mix delle due, l'informazione va in entrambe le direzioni.

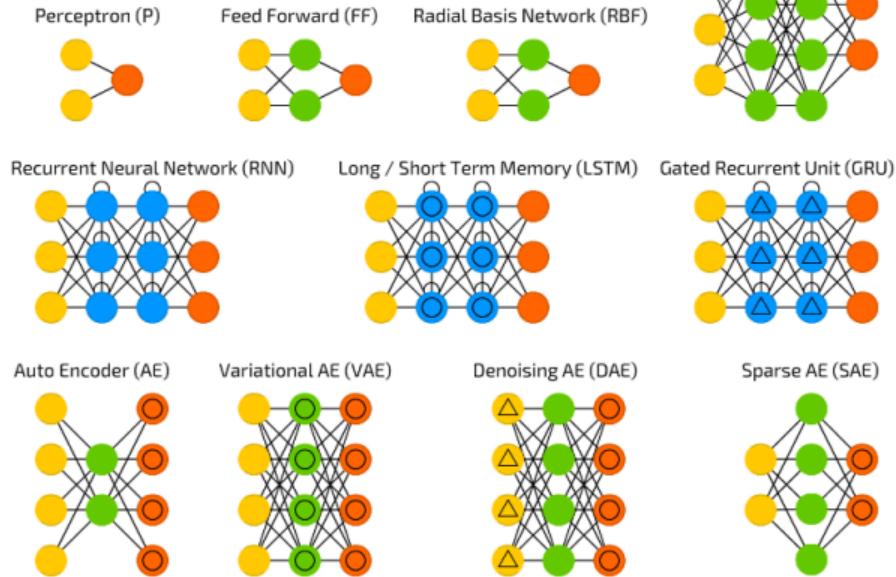
Recurrent, l'informazione va dall'input all'output ma ad un certo punto, alcuna informazione è data al prossimo input. Usata per dati sequenziali, di modo che le computazioni fatte fin ora sono anche usate per il prossimo datapoint, come una "memoria". Può essere usata per audio recognition, banking data, video data.

In realtà abbiamo molte altre architetture

A mostly complete chart of Neural Networks

©2016 Fjodor van Veen - asimovinstitute.org

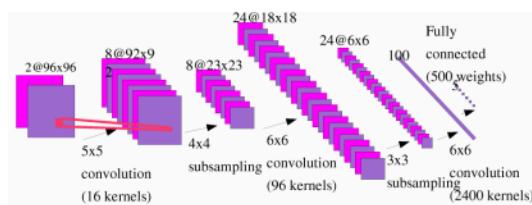
- Backfed Input Cell
- Input Cell
- △ Noisy Input Cell
- Hidden Cell
- Probabilistic Hidden Cell
- △ Spiking Hidden Cell
- Output Cell
- Match Input Output Cell
- Recurrent Cell
- Memory Cell
- △ Different Memory Cell
- Kernel
- Convolution or Pool



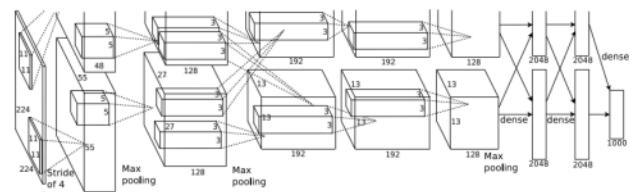
Convolutional Neural Networks (CNNs)

Sono tra le architetture più usate in computer vision.

L'idea è la stessa di quella classiche (stackano stages di feature extractors), qui vediamo 2 tipi di architettura molto usati:



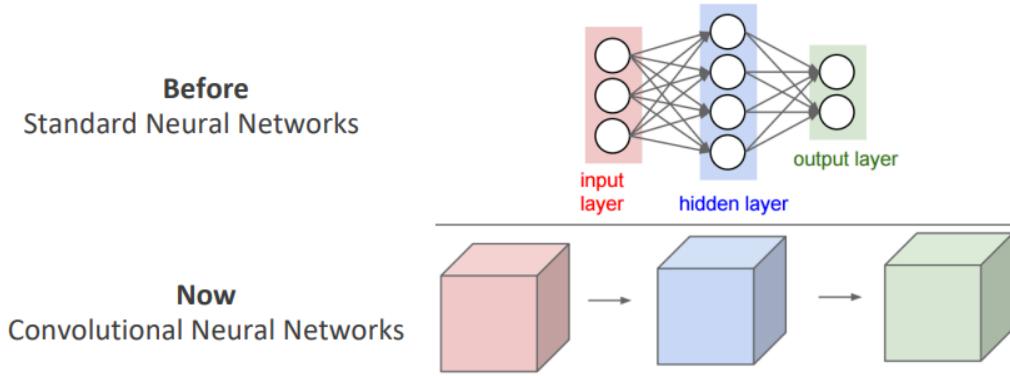
LeNet [1]



AlexNet [2]

Vedremo queste nella prossima lezione.

La differenza sta in come sono organizzati i pesi, prima erano un'informazione bidimensionale, ora sono in 3 dimensioni, sono volumi di pesi.



Supponiamo di avere il nostro input che è un'immagine $32 \times 32 \times 3$. Immagina che vogliamo creare un neurone classico, che quindi è fully-connected all'input, quindi avremo $32 \times 32 \times 3$ connessioni, ovvero più di 3 mila pesi solo per un neurone.

Si può pensare come la costruzione di un classificatore non lineare fa esplodere il numero di parametri.

L'idea è quella di non avere questa full connectivity, ma invece colleghiamo soltanto una parte piccole dell'input. In questo caso colleghiamo solo $5 \times 5 \times 3$, quindi 75 pesi.

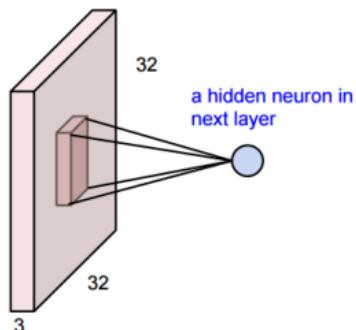


Image: $32 \times 32 \times 3$ volume

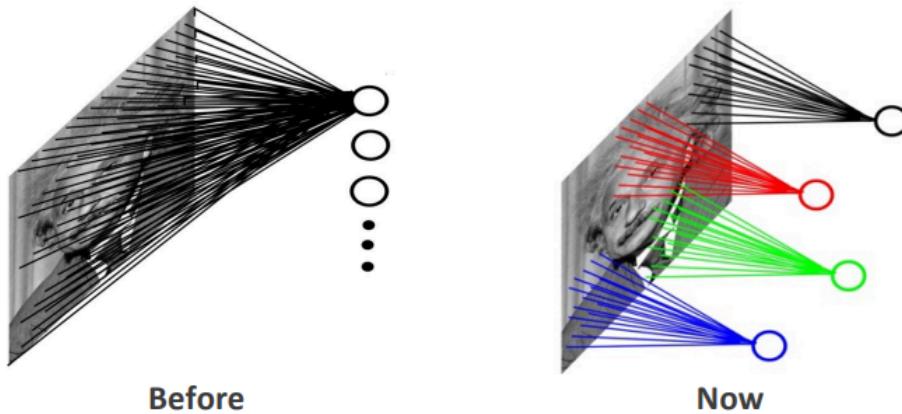
Before: full connectivity $32 \times 32 \times 3 = 3072$ weights

Now: one neuron will connect to, e.g. $5 \times 5 \times 3$ patch and only have $5 \times 5 \times 3 = 75$ weights

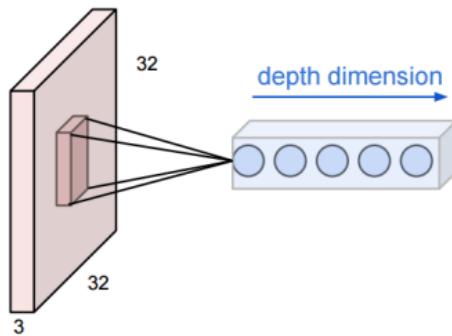
Note that connectivity is:

- Local in space (5×5 inside 32×32)
- Full in space (all 3 depth channels)

Questa connettività è **local in space** (5×5 nel 32×32), ma **full in depth** (uguale in tutti i 3 canali di profondità).



Prima avevamo un hidden layer di 200 neuroni, ora abbiamo un volume di output di 200. Ciascun neurone ci darà una dimensione di profondità nell'output. Ogni neurone produce una feature map che è impilata (stacked).

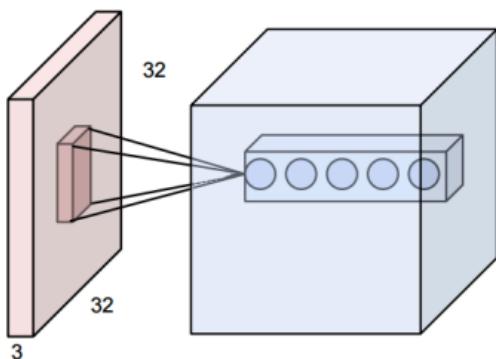


Before: hidden layer of 200 neurons

Now: output volume of depth 200

Multiple neurons all looking at the same region of the input volume, stacked along depth

Quindi questa non è più un'informazione 1×1 , ma è un volume.



These form a single $[1 \times 1 \times \text{depth}]$ «depth column» in the output feature map volume

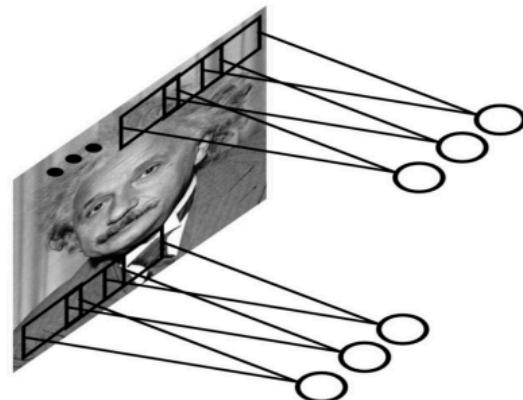
Avere neuroni diversi che guardano a parti diverse dell'immagine è inefficiente. Quindi abbiamo un neurone che guarda ad una parte dell'immagine, ma anche in

zone diverse. Lo stesso neurone guarda a diverse porzioni, questo si chiama weights sharing. è come se avessimo più neuroni, ma invece è solo uno.

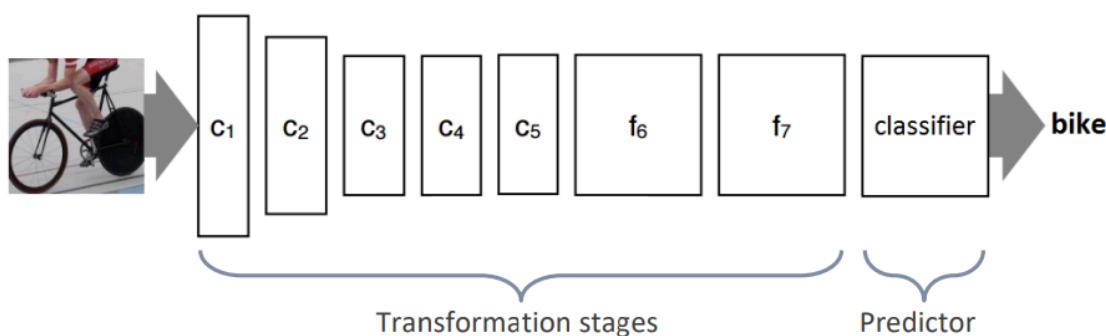
All Neural Network weights arranged in **3 dimensions**

Convolutional Neural Networks are still Neural Networks but:

- Local connectivity
- Weights sharing
 - Weights in the layer are shared across spatial positions



Questa è l'architettura di una CNN



Nella prima parte abbiamo convolutions, che estraggono informazioni e gerarchia dall'immagine, poi arriviamo ad un certo punto ad una informazione monodimensionale e abbiamo un network più normale fino al classifier finale.

Idee chiave:

Feed-forward feature extraction

Supervised training of convolutional filters by back-propagating classification error

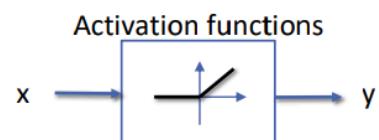
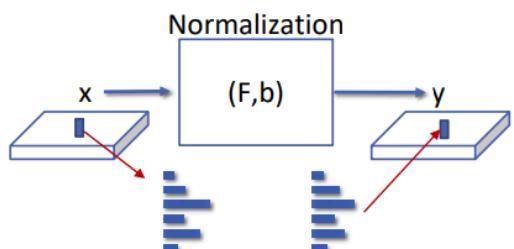
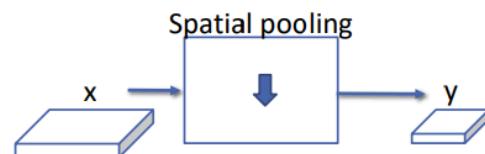
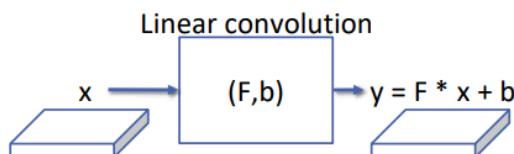
Respect to traditional Neural Networks, CNNs have other special layers

- Spatial pooling
- Local response normalization

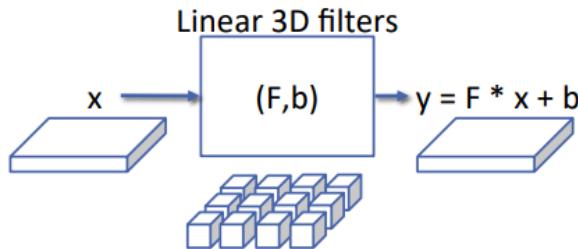
These layers are useful to

- Reduce computational burden
- Increase invariance
- Ease the optimization

Questa è un'overview dei componenti:



Linea convolution



- Linear
- Local
- Translation invariant
- Filter bank to form a richer representation of the data

- Input $x = H \times W \times K$ array
- Filter bank $F = H' \times W' \times K \times Q$ array
- Output $y = (H - H' + 1) \times (W - W' + 1) \times Q$ array

$$y_{ijq} = y_q + \sum_{u=0}^{H-1} \sum_{v=0}^{W-1} \sum_{k=1}^K x_{u+i,v+j,k} F_{u,v,k,q}$$

1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

Image

4		

Convolved Feature

L'output è raggiunto facendo la convoluzione dell'input con il filtro, e poi applicando il bias.

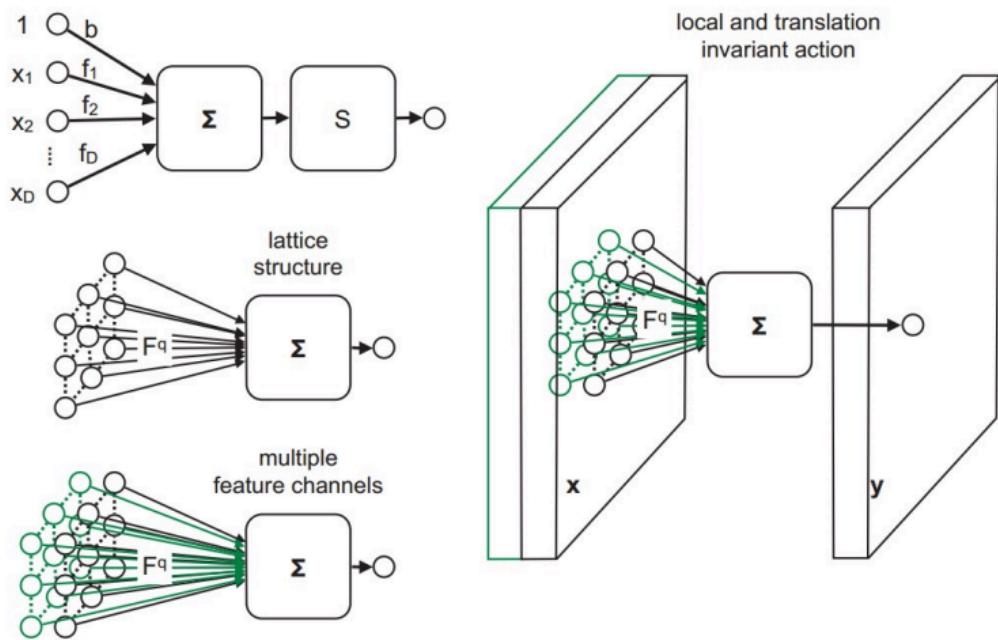
Ora il nostro filtro è un kernel 3×3 . Come facciamo a calcolare il valore in una posizione? Partendo da in alto a sinistra, prendiamo la zona 3×3 , pixel per pixel moltiplichiamo il valore del filtro con il valore nell'immagine, poi sommiamo tutto e raggiungiamo il risultato, che mettiamo in alto a sx.

Poi andiamo alla prossima posizione, spostiamo il filtro uno a destra, calcoliamo come prima, e otteniamo il secondo valore. Continuiamo con questo sliding pattern, spostandoci (in questo caso) di una posizione.

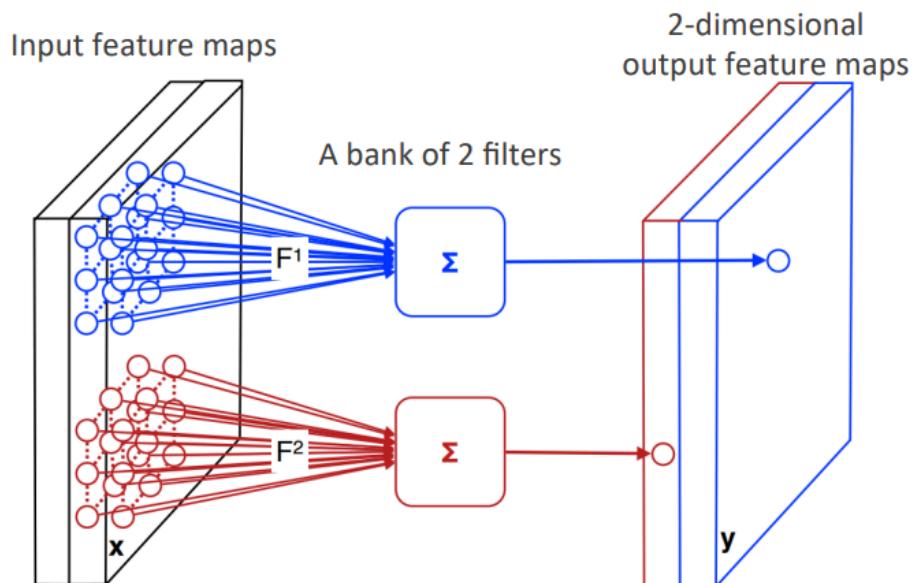
La quantità di elementi che saltiamo da una computazione all'altra è chiamata **stride** del filtro (potrebbe essere più di 1 il movimento del filtro prima di calcolare il prossimo valore).

L'output avrà sicuramente una profondità che dipende dal numero di filtri considerati. La larghezza e altezza dipendono dall'immagine?

Quindi nella formula finale, sommiamo anche tra i diversi layers, applicando alla fine il termine di bias che è diverso per ogni livello di profondità.



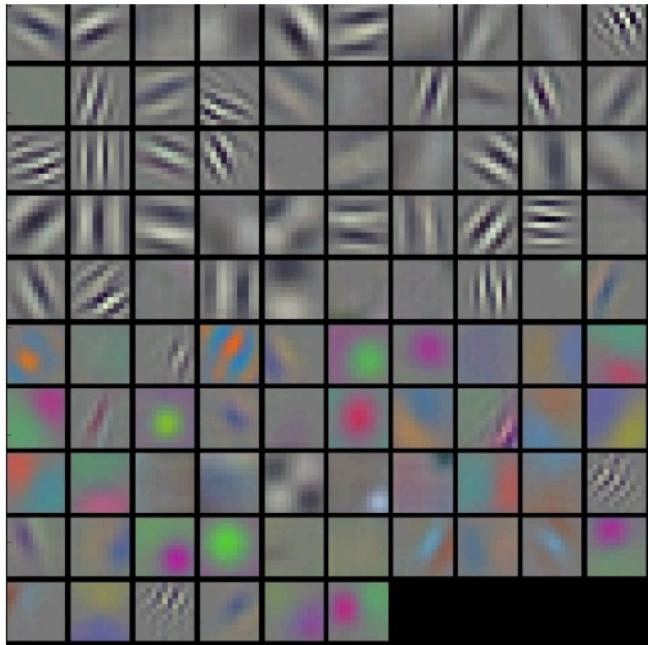
Un filtro, un output plane.



Se abbiamo più filtri (F^1, F^2), ognuno avrà un peso diverso. Verranno quindi calcolati più output planes.

In Alexnet abbiamo 96 filtri nel primo layer convoluzionale, ognuno ha una dimensione 11×11.

Questi filtri calcolano diversi edges a diverse orientazioni a diverse frequenze. Quindi alcuni cercano bordi grandi, altri piccoli. Poi abbiamo dei filtri che estraggono colori, opponents. E poi altri che cercano patterns più complessi.

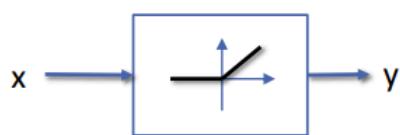


A bank of 96 filters of the first convolutional layer of AlexNet trained on ILSVRC2012 dataset

- Each one is a 11×11 pixels 3D filter (it applies to a RGB image)

Dopo ogni filtro abbiamo l'activation function, è la stessa delle reti neurali normali. Possono essere usate tutte, basta che siano differenziabili, bounded e monotonic increasing.

- Scalar non-linearity



$$y = \frac{1}{1 + e^{-x}}$$

Sigmoid

$$y = \tanh x$$

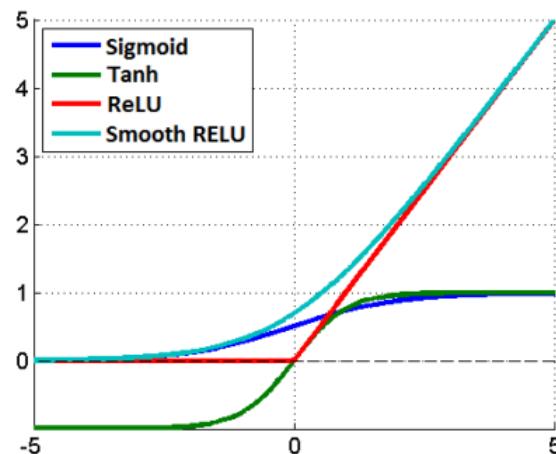
Hyperbolic tangent

$$y = \max\{0, x\}$$

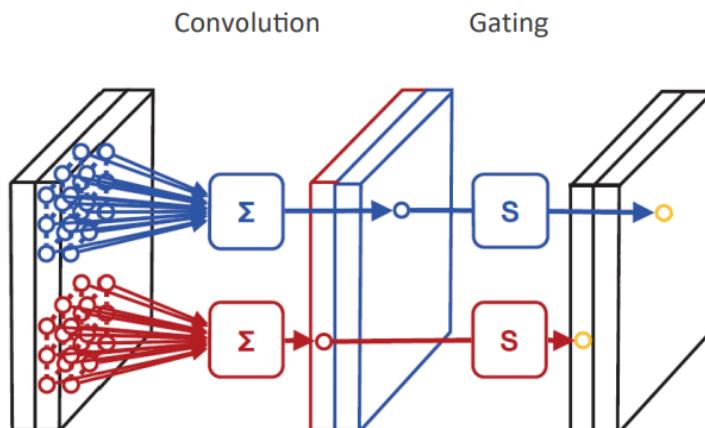
Rectified Linear Unit (ReLU)

$$y = \log(1 + e^x)$$

Smooth ReLU



Dopo aver fatto la convoluzione, applichiamo la non-linearità con l'activation function



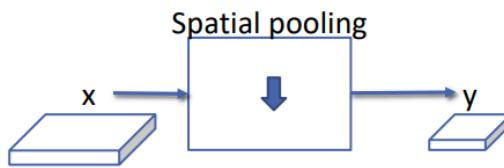
Filters are followed by non-linear operators (e.g. gating function)

Spatial pooling

Questo layer diminuisce la dimensione spaziale. è usato per fare un encoding della varianza e per rendere la computazione più semplice per il prossimo layer.

Abbiamo 2 versioni: max pooling e average pooling. Quando abbiamo scelto la dimensione (2×2 in questo caso) max pooling sceglie il valore massimo nell'area che stiamo considerando, e si muove per una stride che è uguale alla dimensione della cella (2×2 in questo caso). Average pooling invece fa la media, e funziona allo stesso modo con lo stride.

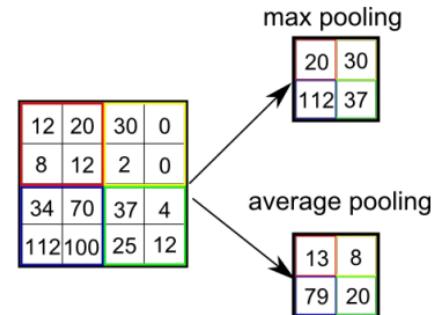
Queste sono applicate canale per canale indipendentemente. Il numero di canali di profondità rimane invariato, perché lavora solo nella dimensione spaziale.



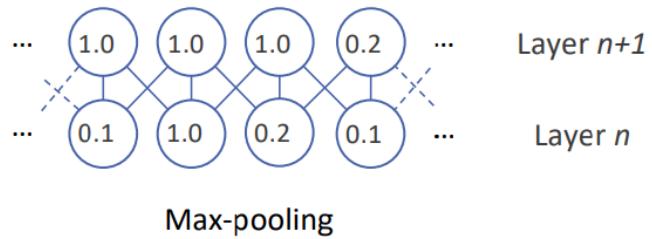
$$y_{ijk} = \max_{pq \in \Omega_{ij}} x_{pqk} \quad \text{Max pooling}$$

$$y_{ijk} = \text{avg}_{pq \in \Omega_{ij}} x_{pqk} \quad \text{Average pooling}$$

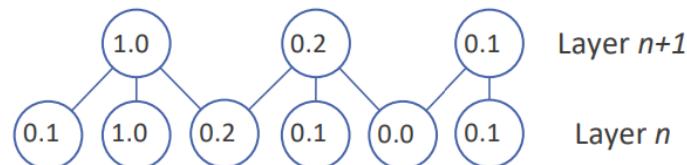
- Pooling and sub-sampling
- Encodes translation invariance
- Pooling computes the average/max of the features in a neighbourhood
- It is applied channel-by-channel



- Aggregate to achieve translation invariance (gain robustness to the exact spatial location of features)



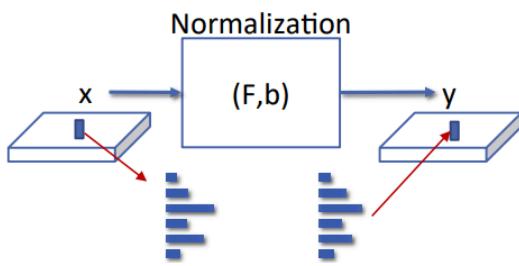
- Subsampling to reduce spatial scale and computation



Local Response Normalization (LRN)

Questi layers non cambiano la dimensione dell'input, ma normalizzano i pesi in alcuni modi. Questo è usato per migliorare la varianza, l'ottimizzazione, e per avere più sparsity nella rete.

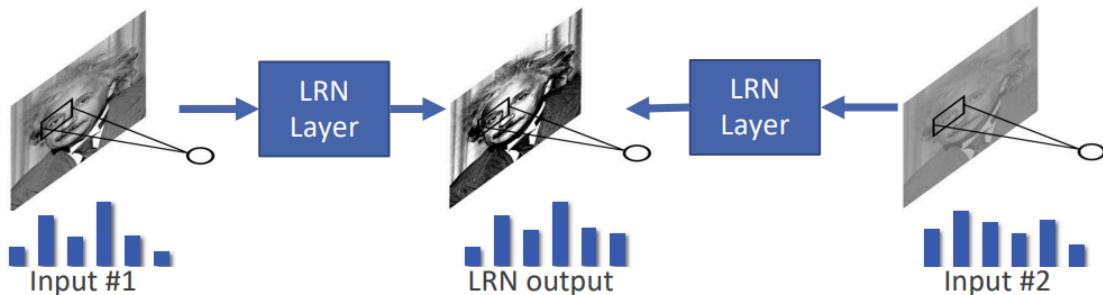
Nell'esempio vogliamo vedere che è la stessa persona, non ci interessa il contrasto. Normalizzando i valori, facciamo in modo che nel prossimo layers i dati siano più easy.



Contrast normalization

Effects:

- Improves invariance
- Improves optimization
- Improves sparsity



We want the same response

Abbiamo 2 tipi:

Within channel lavora su ciascun canale da solo. Quindi normalizza quelli in quel canale, indipendentemente per ciascun canale.

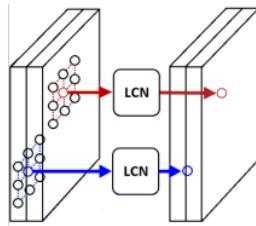
Across channel invece normalizza tra tutti i canali contemporaneamente.

WITHIN CHANNEL

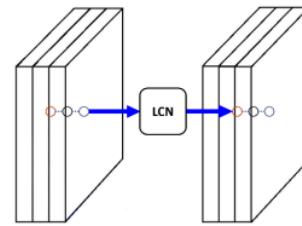
- Operates independently on different feature channels
- Rescales each input feature basing on a local neighborhood

ACROSS CHANNELS

- Operates independently at each spatial location and groups of channels
- Normalizes groups $G(k)$ of feature channels
- Groups are usually defined in a sliding window manner



$$y_{ijk} = x_{ijk} \left(\kappa + \alpha \sum_{(u,v) \in N(i,j)} x_{uvk}^2 \right)^{-\beta}$$

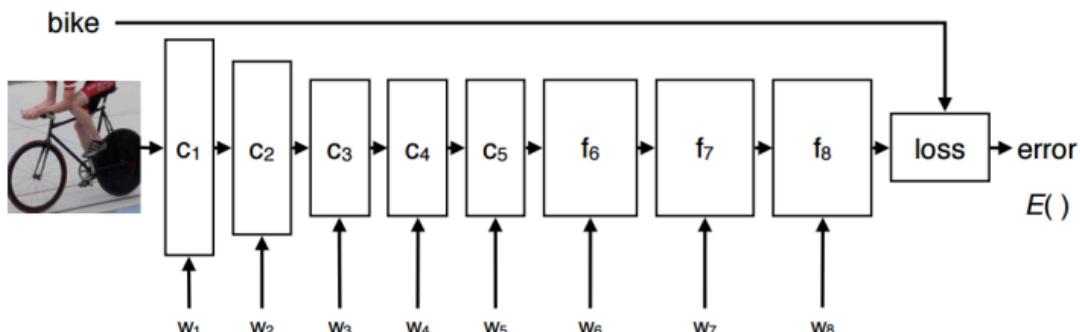


$$y_{ijk} = x_{ijk} \left(\kappa + \alpha \sum_{q \in G(k)} x_{ijq}^2 \right)^{-\beta}$$

Training CNNs

Come facciamo il training? Nello stesso modo delle reti normali. Vogliamo trovare i pesi w che minimizzano l'errore tra i labels reali e quelli stimati. Usiamo SGD (Stochastic Gradient Descent) e backpropagation.

Vogliamo minimizzare l'errore, rispetto a tutti i pesi in tutti i canali diversi.



$$\operatorname{argmin} E(\mathbf{w}_1, \mathbf{w}_2, \dots, \mathbf{w}_8)$$

Questo è fatto nello stesso modo, facciamo le derivate parziali rispetto a tutti i pesi diversi.

Quindi facciamo la parte forward, poi vogliamo aggiornare i pesi tornando indietro.

