



UNIVERSITÀ  
DEGLI STUDI  
DI PALERMO



# Introduzione al Deep Learning

CORSO DI BIG DATA

a.a. 2020/2021

Prof. Roberto Pirrone

# Sommario

- Generalità
- Multi-Layer Perceptron
- Aspetti architetturali
  - Funzioni di costo e unità di uscita
  - Unità nascoste
- Addestramento:
  - Stochastic Gradient Descent
  - Backpropagation
- Regolarizzazione e Ottimizzazione
- Reti Convoluzionali
- Autoencoder
- Generative Adversarial Networks
- Reti Ricorrenti
- Transformers

# Generalità

- Il Deep Learning (DL) nasce essenzialmente per fornire un framework di addestramento di algoritmi che abbiano buona capacità di generalizzazione nei compiti più sfidanti dell’Intelligenza Artificiale quali la Visione Artificiale o il Riconoscimento del Parlato
- Questi compiti non sono, di fatto, risolvibili con gli algoritmi tradizionali di Machine Learning (ML) incluse le reti neurali artificiali classiche, poco profonde o *shallow*

# Generalità

- I problemi più rilevanti da un punto di vista computazionale, a questo riguardo, sono:
  - Incapacità di rappresentare funzioni ad elevata complessità per mappare gli ingressi nelle uscite desiderate
  - Curse of dimensionality legato al fatto che le configurazioni rilevanti degli ingressi crescono esponenzialmente con le dimensioni e quindi possono essere molte di più degli esempi di addestramento

# Generalità

- La necessità di calcolare funzioni complesse sui dati di input nasce dal fatto che, spesso, la capacità di generalizzazione di un algoritmo di ML dipende dai cosiddetti *prior* ovvero dalle assunzioni che si fanno in termini dello spazio delle ipotesi sulla forma funzionale di  $f$
- Termini di regolarizzazione espressi come condizioni di *smoothing* esplicitamente inclusi nelle funzioni di Loss
- Veri e propri prior in senso probabilistico

# Generalità

- In genere un questi prior tendono a preferire l'apprendimento di una funzione che rimane pressoché invariata in un piccolo intorno del punto  $x$ .

$$f(x) \approx f(x + \epsilon)$$

- Ovviamente questo limita la capacità di generalizzazione
  - Servono molti campioni ben distribuiti nelle zone di elevata variazione di  $f$
  - All'aumentare delle dimensioni, anche questi prior non garantiscono smoothness lungo tutte le dimensioni
  - *Servono algoritmi che apprendano funzioni più complesse da pochi dati*

# Generalità

- Il DL gestisce la complessità facendo alcune assunzioni sullo spazio delle ipotesi:
  - I dati sono generati dalla composizione di molti fattori semplici (feature) aggregati secondo diversi livelli di una gerarchia
    - Es. La percezione di un volto si basa sulla percezione di una forma ovoidale e di un certo pigmento di pelle, ma anche di alcune caratteristiche specifiche come presenza di occhi, naso e bocca, e, successivamente, della forma e del colore di ciascuna caratteristica con le sue varianti

# Generalità

- Il DL gestisce la complessità facendo alcune assunzioni sullo spazio delle ipotesi:
  - Non tutte le configurazioni dei dati di ingresso sono rilevanti per il problema
  - La funzione  $f$  assume valori rilevanti non su tutto  $\mathbb{R}^d$ , ma su un *manifold* cioè su un sottoinsieme connesso di punti in  $\mathbb{R}^d$  che hanno di per sé una topologia con dimensionalità più bassa (*manifold hypothesis*)
    - Manifold → varietà topologica, ad es. una superficie 2D o una curva 1D in  $\mathbb{R}^3$

# Multi-Layer Perceptron

- Le Reti Neurali Profonde (*Deep Neural Networks – DNN*) sono basate principalmente sul *Multi-Layer Perceptron* (MLP)
- Una serie di strati di unità computazionali (*neuroni*) che calcolano una funzione del proprio vettore di ingresso (*funzione di attivazione*)
- Le reti sono *dense (fully connected)*, cioè ogni neurone di uno strato riceve gli ingressi da *tutti* i neuroni dello strato precedente

# Multi-Layer Perceptron

- Apprende una classificazione sui dati di ingresso:

$$y = f^*(\mathbf{x})$$

- L'architettura della rete è data da una serie di  $n$  strati (la *profondità* della rete) in cui si trovano numeri diversi di unità (la *larghezza* della rete)
- Gli strati interni della rete sono detti strati nascosti (*hidden*) le cui attivazioni indicheremo con  $h$
- Nel complesso la rete approssima  $f^*$  come  $f(\mathbf{x}; \theta)$  e viene addestrata a trovare i parametri  $\theta$  ottimi per la migliore approssimazione di  $f^*$

# Multi-Layer Perceptron

- $f(\mathbf{x}; \theta)$  e viene calcolata concatenando gli effetti delle funzioni di attivazione di ogni strato

$$f(\mathbf{x}) = f^{(n)}(\dots f^{(3)}(f^{(2)}(f^{(1)}(\mathbf{x}))))$$

- Nel DL la forma funzionale di  $f$  si definisce come un operatore lineare applicato ad un mapping non lineare  $\phi(\mathbf{x})$  degli ingressi

$$y = f(\mathbf{x}; \theta, \mathbf{w}) = \phi(\mathbf{x}; \theta)^\top \mathbf{w}$$

- La rete, quindi, apprende anche la rappresentazione ottima per  $\phi$

# Aspetti architetturali

- L'architettura di una DNN è molto complessa ed è isomorfa ad un grafo
- Si parla in questo caso di «grafo di computazione» che viene generato sia per il calcolo delle attivazioni «in avanti» attraverso la rete sia per il calcolo dei gradienti e degli errori «all'indietro» nell'addestramento con backpropagation
- I framework di DL si appoggiano esplicitamente a un componente di «graph compiling» per mappare le operazioni sul hardware

# Aspetti architetturali

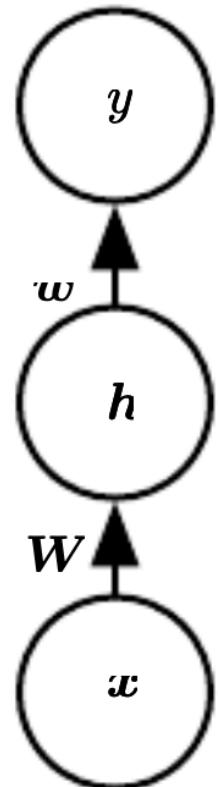
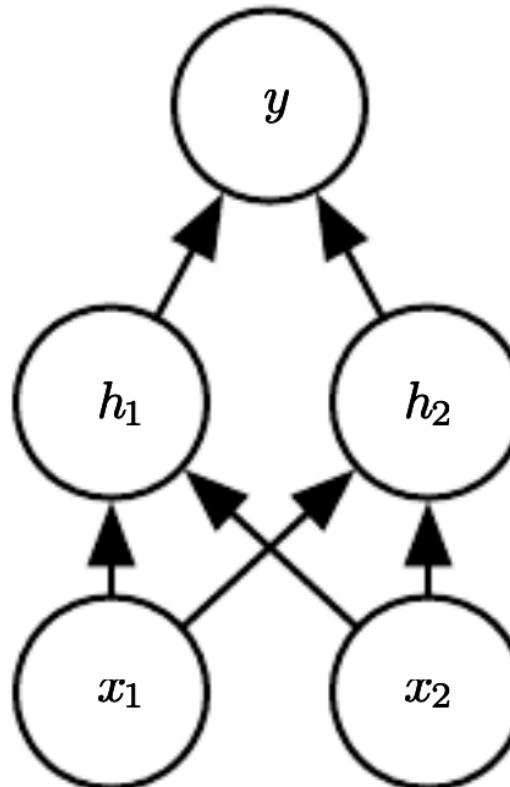
- Esempio: Rete per il calcolo dello XOR

$$y = \mathbf{h}^\top \mathbf{w} + b$$

$$\mathbf{h} = \max\{0, \mathbf{W}^\top \mathbf{x} + \mathbf{c}\}$$

Rectified Linear Unit (ReLU)

Ogni colonna di  $\mathbf{W}$  rappresenta  
il vettore di pesi  $\mathbf{W}_{:,i}$  che connettono  
gli ingressi all'unità nascosta  $h_i$



# Aspetti architetturali

- La scelta delle funzioni di attivazione delle unità di uscita e di quelle nascoste dipende da alcuni fattori
  - Consentire la stima di funzioni non lineari complesse
  - Soddisfare le forme funzionali di uno stimatore MLE o MAP riguardo alla distribuzione di probabilità dei valori delle uscite
  - Avere un gradiente ampio e ben stimabile per supportare gli algoritmi di apprendimento che si basano sulla discesa lungo il gradiente dell'errore commesso sulle uscite

# Aspetti architetturali

- Nella scelta della funzione di costo abbiamo le stesse opzioni degli altri modelli di ML
  - La cross-entropia fornisce la stima MLE di una distribuzione di probabilità condizionale  $p(\mathbf{y}|\mathbf{x})$

$$J(\boldsymbol{\theta}) = -\mathbb{E}_{\mathbf{x}, \mathbf{y} \sim \hat{p}_{\text{data}}} \log p_{\text{model}}(\mathbf{y} \mid \mathbf{x})$$

$$p_{\text{model}}(\mathbf{y} \mid \mathbf{x}) = \mathcal{N}(\mathbf{y}; f(\mathbf{x}; \boldsymbol{\theta}), \mathbf{I})$$

Stimatore della media  
della Gaussiana

$$J(\boldsymbol{\theta}) = \frac{1}{2} \mathbb{E}_{\mathbf{x}, \mathbf{y} \sim \hat{p}_{\text{data}}} \|\mathbf{y} - f(\mathbf{x}; \boldsymbol{\theta})\|^2 + \text{const}$$

**MSE**

# Aspetti architetturali

- Possiamo essere interessati a stimare semplicemente *una statistica*  $f^*(\mathbf{x}; \theta)$  su  $\mathbf{y}$
- Il problema di apprendimento diventa quello di stimare la funzione ottima  $f^*$

$$f^* = \arg \min_f \mathbb{E}_{\mathbf{x}, \mathbf{y} \sim p_{\text{data}}} \|\mathbf{y} - f(\mathbf{x})\|^2$$

$$f^*(\mathbf{x}) = \mathbb{E}_{\mathbf{y} \sim p_{\text{data}}} (\mathbf{y} | \mathbf{x}) [\mathbf{y}] \quad \text{Il MSE stima la media di } \mathbf{y}$$

# Aspetti architetturali

- Possiamo essere interessati a stimare semplicemente *una statistica*  $f^*(\mathbf{x}; \theta)$  su  $\mathbf{y}$
- Il problema di apprendimento diventa quello di stimare la funzione ottima  $f^*$

$$f^* = \arg \min_f \mathbb{E}_{\mathbf{x}, \mathbf{y} \sim p_{\text{data}}} \|\mathbf{y} - f(\mathbf{x})\|_1$$

Il Mean Absolute Error stima la *mediana* di  $\mathbf{y}$

# Aspetti architetturali

- Scelta della funzione di attivazione delle unità di uscita
- Assumiamo che la stima sia appresa dagli strati nascosti in termini dei parametri, per cui avremo che  $\hat{h} = f(\mathbf{x}; \theta)$
- Unità lineari di uscita possono essere usate per stimare la media di  $y$ , supponendo una distribuzione Gaussiana e usando il MSE

$$\begin{aligned}\hat{y} &= \mathbf{W}^\top \mathbf{h} + \mathbf{b} \\ p(\mathbf{y}|\mathbf{x}) &= \mathcal{N}(\mathbf{y}; \hat{y}, \mathbf{I})\end{aligned}$$

Non stimiamo alcuna matrice di covarianza  $\Sigma$ , altrimenti le unità lineari non bastano

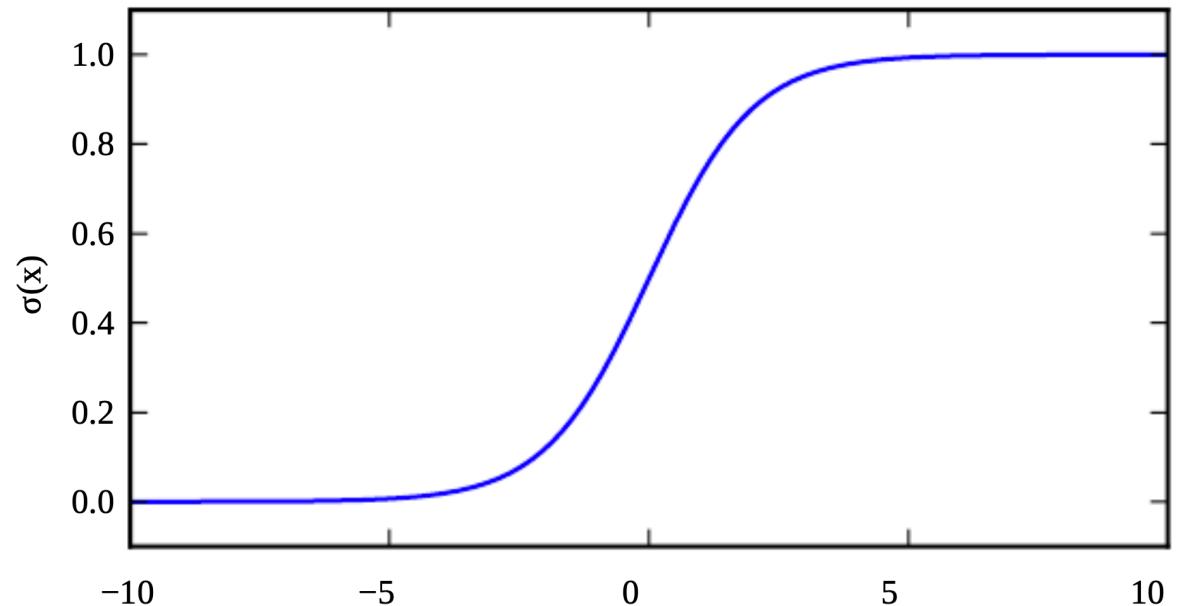
# Aspetti architetturali

- Nel caso di una classificazione binaria si utilizzerà una sola unità di uscita che fornisce la MLE di  $P(y=1 \mid \mathbf{x}) \in [0,1]$  che è una distribuzione di Bernoulli
- Assumeremo che l'ingresso all'unità sia sempre lineare  $z = \mathbf{W}^\top \mathbf{h} + b$
- Il valore  $z$  fornirà la stima della *log-probabilità non normalizzata* di  $y$  e rappresenta la reale uscita dei layer profondi della rete

# Aspetti architetturali

- Una attivazione lineare con soglia non va bene perché il gradiente si annulla non appena si va al di fuori dell'intervallo [0,1]
- Si utilizzerà una funzione di attivazione sigmoidale

$$\sigma(x) = \frac{1}{1 + \exp(-x)} \equiv \frac{\exp(x)}{\exp(x) + \exp(0)}$$



# Aspetti architetturali

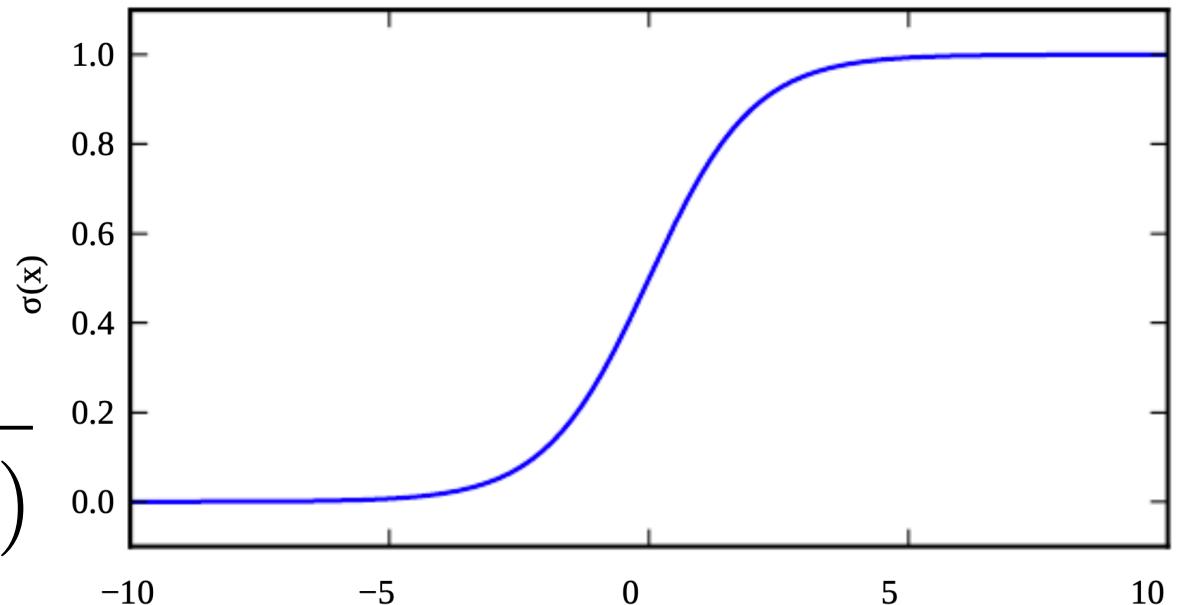
- Possiamo assumere che la log-probabilità non normalizzata sia lineare in  $y$  e  $z$  ( $y = 1$  perché è la probabilità che stiamo stimando)

$$\log \tilde{P}(y) = yz$$

$$\tilde{P}(y) = \exp(yz)$$

$$P(y) = \frac{\exp(yz)}{\sum_{y'=0}^1 \exp(y'z)}$$

$$P(y) = \sigma((2y - 1)z) \equiv \sigma(z) = \sigma(W^\top h + b)$$

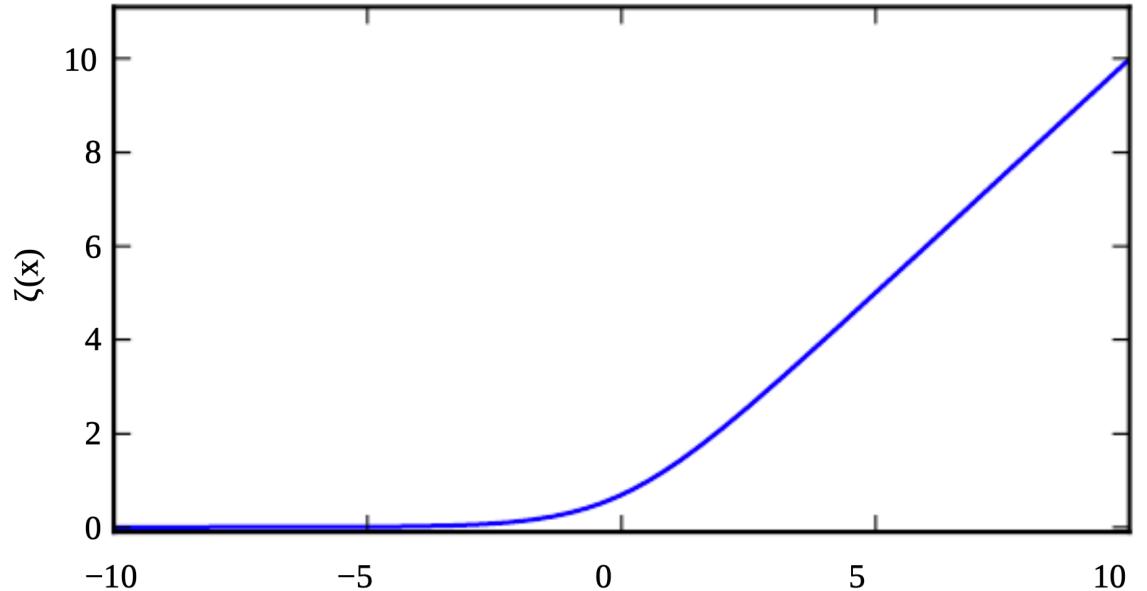


# Aspetti architetturali

- Ricaviamo la loss function corrispondente all'apprendimento MLE

$$\begin{aligned} J(\theta) &= -\log P(y \mid x) \\ &= -\log \sigma((2y - 1)z) \\ &= \zeta((1 - 2y)z) \end{aligned}$$

*softplus*



*Il logaritmo dell'esponenziale fornisce una funzione lineare  
che ha un buon comportamento per il gradiente*

# Aspetti architetturali

- Nel caso di una classificazione multiclasse stimiamo una distribuzione Multinoulli

$$\hat{\mathbf{y}} = \{\hat{y}_i = P(y = i | \mathbf{x})\}$$

$$\sum_i \hat{y}_i = 1$$

- Estendiamo l'uso della funzione sigmoidale con l'applicazione della funzione *softmax*

# Aspetti architetturali

- La funzione softmax, usando una loss per apprendimento MLE, soddisfa i requisiti per avere somma 1 su tutti i componenti anche quelli relativi al calcolo del gradiente

$$z = \mathbf{W}^\top \mathbf{h} + \mathbf{b} = \{z_i = \log \tilde{P}(y = i | \mathbf{x})\}$$

$$\text{softmax}(\mathbf{z})_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

Poiché stimiamo probabilità che sono di volta in volta concentrate su una sola classe  $i$ ,  $z_j$  è molto piccolo se diverso da  $\max_j z_j$

$$\log \text{softmax}(\mathbf{z})_i = z_i - \log \sum_j \exp(z_j) \approx z_i - \max_j z_j$$

# Aspetti architetturali

- Nel momento in cui  $\max_j z_j$  corrisponde all'esempio corretto  $z_i$ , la log-likelihood negativa per quell'esempio tende a 0 e quindi non da contributo alla funzione di costo
- Questo induce una stabilizzazione numerica per evitare problemi nel caso di valori di  $z$  estremamente grandi in magnitudine

$$\begin{aligned}\text{softmax}(z) &= \text{softmax}(z + c) \\ &= \text{softmax}(z - \max_i z_i)\end{aligned}$$

# Aspetti architetturali

- Attivazione delle unità nascoste
  - Anche in questo caso la rete applicherà una funzione di attivazione  $g(\cdot)$  ad una trasformazione affine sui suoi ingressi
$$h = g(W^\top x + b)$$
  - In genere  $b$  viene inizializzato, in ogni strato, con dei valori piccoli, ad es. 0.1
  - La Rectified Linear Unit (ReLU) assume  $g(z) = \max(0, z)$

# Aspetti architetturali

- ReLU e sue varianti
  - La ReLU è ottima per costruire non linearità attraverso il meccanismo della composizione strato per strato, ma non è differenziabile per  $z=0$
  - Da un punto di vista numerico, in  $z=0$  si usa la derivata destra o la sinistra di  $g(\cdot)$  che sono entrambe definite
    - C'è sempre un errore numerico nel calcolo esatto di un valore di  $z$ , per cui si può sempre assumere che  $g(0)$  corrisponda in realtà a un valore  $g(0 + \varepsilon)$  con  $\varepsilon$  molto piccolo cui corrisponde un gradiente perfettamente definito

# Aspetti architetturali

- ReLU e sue varianti

$$h_i = g(z, \alpha)_i = \max(0, z_i) + \alpha_i \min(0, z_i)$$

- Unità lineare:  $\alpha_i = 1$
- Leaky ReLU:  $\alpha_i$  molto piccolo (ad es. 0.01)
- Parametric ReLU:  $\alpha_i$  è un parametro da apprendere

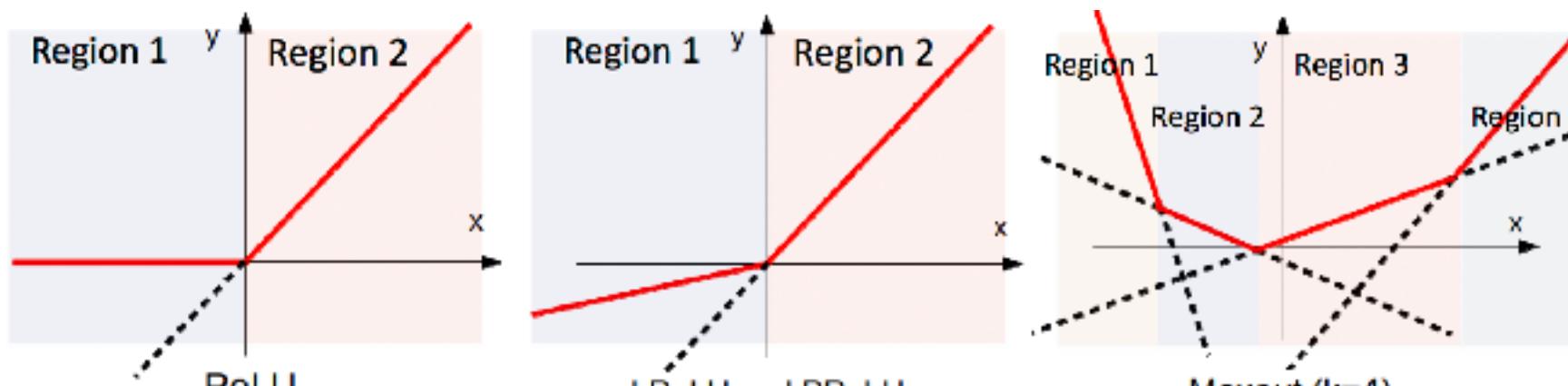
# Aspetti architetturali

- Maxout

$$g(z)_i = \max_{j \in \mathbb{G}^{(i)}} z_j = \max_{j \in \mathbb{G}^{(i)}} (\mathbf{W}_{:,j}^\top \mathbf{x} + b_j)$$

$$\mathbb{G}^{(i)} = \{(i-1)k+1, \dots, ik\}$$

Apprende una funzione lineare a tratti degli ingressi su  $k$  regioni



# Aspetti architetturali

- Altre funzioni di attivazione per le unità nascoste

- Sigmoide
- Tangente iperbolica
- Softplus
- Radial Basis Function (RBF)

$$h_i = \exp\left(-\frac{1}{\sigma_i^2} \|\mathbf{W}_{:,i} - \mathbf{x}\|^2\right)$$

# Aspetti architetturali

- Altre considerazioni sulla struttura dei layer
  - In principio il Teorema dell'Approssimazione Universale ci garantisce che

«Una qualunque rete neurale con layer di uscita lineare e uno strato di unità nascoste con attivazione che satura agli estremi del suo intervallo di definizione approssima qualunque funzione (Borel-misurabile) da un qualunque spazio a dimensione finita ad un altro, con un errore piccolo quanto si vuole *posto che abbia un numero sufficiente di unità nascoste*»

*Ogni funzione continua in qualunque sottoinsieme chiuso e limitato  
di  $\mathbb{R}^n$  è Borel-misurabile*

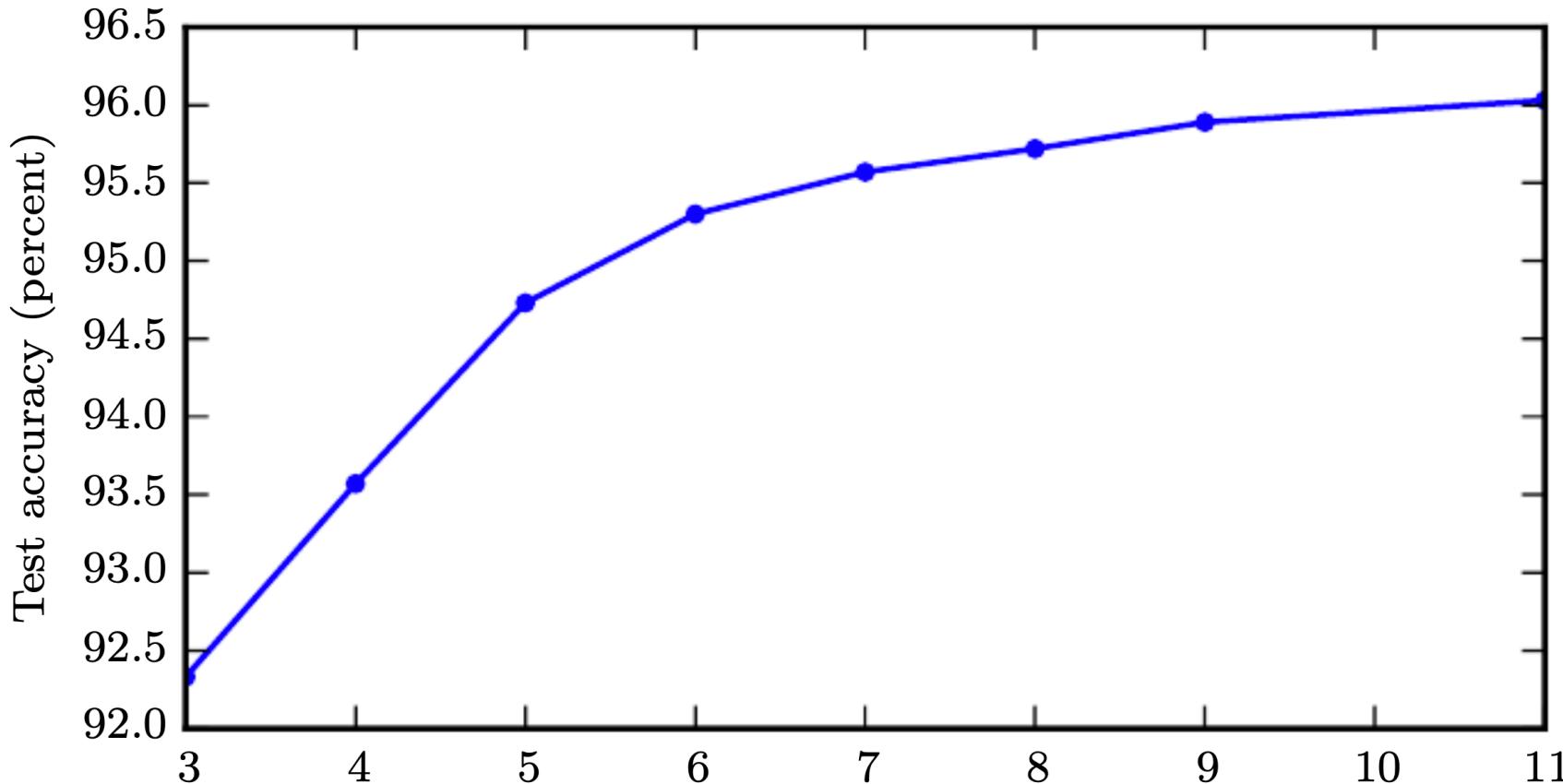
# Aspetti architetturali

- Ovviamente una rete del genere sarebbe enormemente *larga*
- La soluzione, storicamente, è stata quella di andare in profondità, utilizzando più strati nascosti
  - Si assume che possiamo approssimare una funzione complessa come composizione di funzioni lineari a tratti
  - Il pattern che dobbiamo predire dipende da un insieme di *feature complesse* che, a loro volta, dipendono da altre *feature più semplici* e così via
- Ovviamente emerge un problema di numero di parametri che esplode esponenzialmente

# Aspetti architetturali

- I layer non devono essere necessariamente densi
  - Nelle Reti Neurali Convoluzionali (Convolutional Neural Networks – CNN) ogni unità del layer  $i+1$  è connessa *alle sole unità del layer  $i$  che formano il supporto per un kernel di convoluzione* utilizzato per calcolare gli ingressi all'unità stessa
  - Forte riduzione dei parametri e maggiore efficienza con reti molto profonde le quali, appunto, sono possibili proprio per effetto del ridotto numero di connessioni

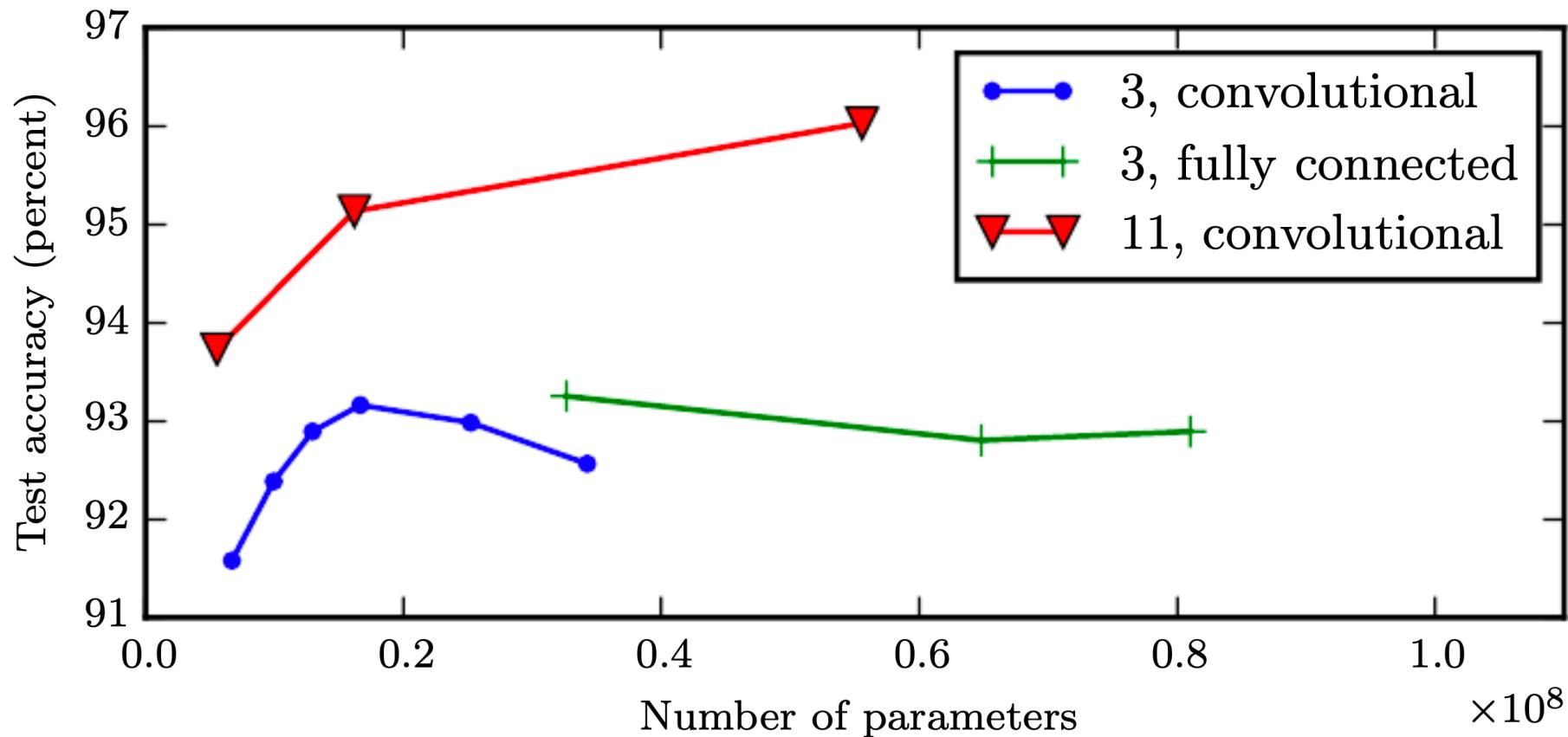
# Aspetti architetturali



Incremento dell'accuratezza  
all'aumentare degli strati  
su un data set di immagini di  
numeri civici, acquisite da  
Google Street View

Fonte:  
<https://arxiv.org/abs/1312.6082>

# Aspetti architetturali

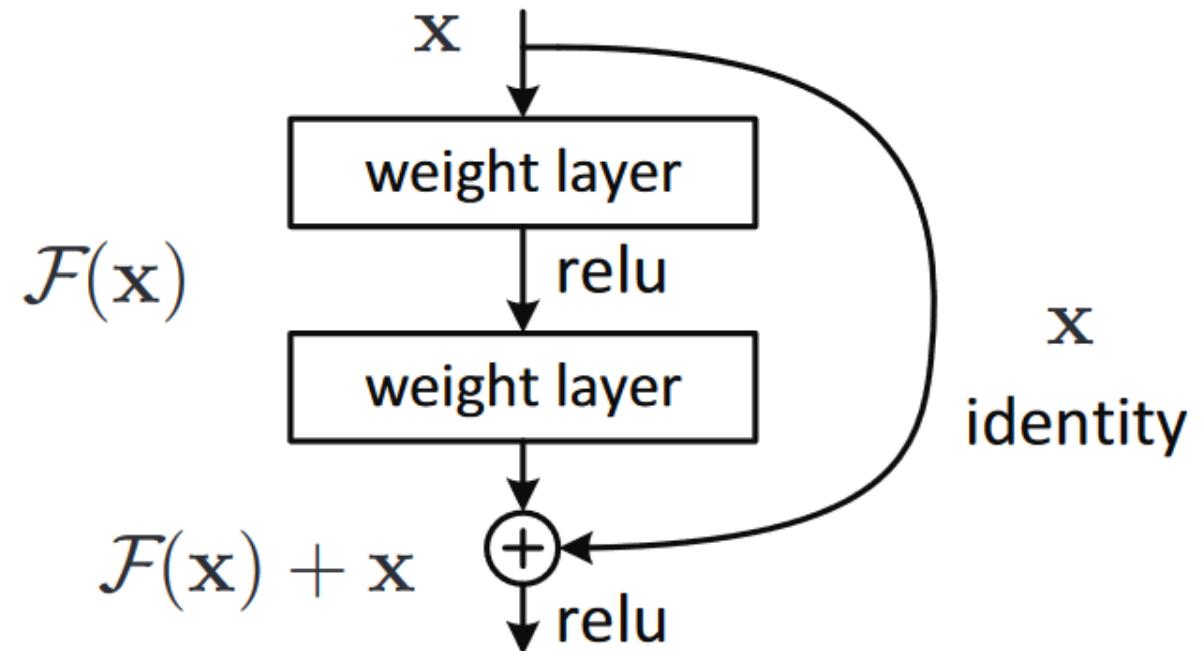


Diminuzione del numero dei parametri all'aumentare della profondità della rete su un data set di immagini di numeri civici, acquisite da Google Street View

Fonte:  
<https://arxiv.org/abs/1312.6082>

# Aspetti architetturali

- Spesso si usano le cosiddette *skip connections*: un layer è connesso «in avanti» a un layer non successivo
  - Aiuta a gestire il problema del «vanishing gradient»
  - Un blocco del genere è detto anche «residual block»



Il gradiente del residual block aggiunge un termine costante a  $\partial\mathcal{F}/\partial x$  mal condizionato e il blocco apprenderà  $\mathcal{H}(x)=\mathcal{F}(x)+x$

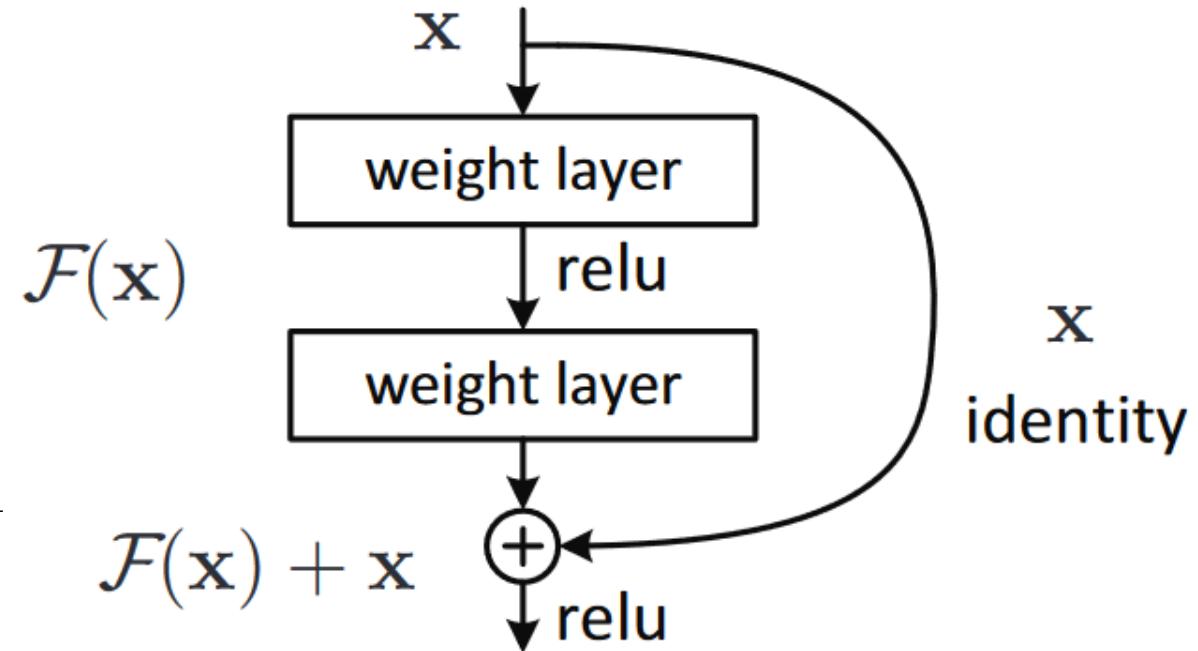
# Aspetti architetturali

- Dopo  $t$  layer la funzione  $\mathcal{F}$  appresa tende a divenire proporzionale a  $\mathbf{W}^t$  e il suo gradiente a  $t\mathbf{W}^{t-1}$

$$\mathbf{W} = (\mathbf{V} \operatorname{diag}(\boldsymbol{\lambda}) \mathbf{V}^{-1})$$

$$\mathbf{W}^t = (\mathbf{V} \operatorname{diag}(\boldsymbol{\lambda}) \mathbf{V}^{-1})^t = \mathbf{V} \operatorname{diag}(\boldsymbol{\lambda})^t \mathbf{V}^{-1}$$

- $\lambda_i \gg 1 \rightarrow$  «exploding gradients»
- $\lambda_i \ll 1 \rightarrow$  «vanishing gradients»



Il gradiente del residual block aggiunge un termine costante a  $\partial\mathcal{F}/\partial x$  mal condizionato e il blocco apprenderà  $\mathcal{H}(x)=\mathcal{F}(x)+x$

# Addestramento

- L'addestramento di una DNN, come anche nel caso del MLP, avviene utilizzando un algoritmo di discesa lungo il gradiente di una funzione di costo  $J(\theta, x)$  rispetto ai parametri del modello
- Si usa lo *Stochastic Gradient Descent*
- Sarà necessario quindi calcolare  $\nabla_{\theta} J(\theta, x)$  per minimizzare  $J$

# Addestramento

- Il calcolo di  $\nabla_{\theta} J(\theta, x)$  riguarda tutti i parametri, anche quelli negli strati più nascosti
- Viene utilizzato un algoritmo per calcolare i gradienti in maniera ricorsiva utilizzando un grafo computazionale
- Questo algoritmo è la *Backpropagation*

# Addestramento

- Backpropagation

- Si basa sul calcolo del gradiente di funzione composta  $z = f(g(\mathbf{x}))$
- Estendibile al caso di funzioni di tensori n-dimensionali  $\mathbf{X}$  in cui  $j$  è una n-upla di indici degli elementi componenti

$$\mathbf{y} = g(\mathbf{x}) \text{ and } z = f(\mathbf{y})$$

$$\nabla_{\mathbf{x}} z = \left( \frac{\partial \mathbf{y}}{\partial \mathbf{x}} \right)^T \nabla_{\mathbf{y}} z,$$

$$\nabla_{\mathbf{X}} z = \sum_j (\nabla_{\mathbf{X}} Y_j) \frac{\partial z}{\partial Y_j}$$

# Addestramento

- Backpropagation
- Algoritmo di calcolo delle attivazioni per una funzione di loss con termine di regolarizzazione

**Require:** Network depth,  $l$

**Require:**  $\mathbf{W}^{(i)}, i \in \{1, \dots, l\}$ , the weight matrices of the model

**Require:**  $\mathbf{b}^{(i)}, i \in \{1, \dots, l\}$ , the bias parameters of the model

**Require:**  $\mathbf{x}$ , the input to process

**Require:**  $\mathbf{y}$ , the target output

$$\mathbf{h}^{(0)} = \mathbf{x}$$

**for**  $k = 1, \dots, l$  **do**

$$\mathbf{a}^{(k)} = \mathbf{b}^{(k)} + \mathbf{W}^{(k)} \mathbf{h}^{(k-1)}$$

$$\mathbf{h}^{(k)} = f(\mathbf{a}^{(k)})$$

**end for**

$$\hat{\mathbf{y}} = \mathbf{h}^{(l)}$$

$$J = L(\hat{\mathbf{y}}, \mathbf{y}) + \lambda \Omega(\theta)$$

# Addestramento

After the forward computation, compute the gradient on the output layer:

$$\mathbf{g} \leftarrow \nabla_{\hat{\mathbf{y}}} J = \nabla_{\hat{\mathbf{y}}} L(\hat{\mathbf{y}}, \mathbf{y})$$

**for**  $k = l, l - 1, \dots, 1$  **do**

Convert the gradient on the layer's output into a gradient on the pre-nonlinearity activation (element-wise multiplication if  $f$  is element-wise):

$$\mathbf{g} \leftarrow \nabla_{\mathbf{a}^{(k)}} J = \mathbf{g} \odot f'(\mathbf{a}^{(k)})$$

Compute gradients on weights and biases (including the regularization term, where needed):

$$\nabla_{\mathbf{b}^{(k)}} J = \mathbf{g} + \lambda \nabla_{\mathbf{b}^{(k)}} \Omega(\theta)$$

$$\nabla_{\mathbf{W}^{(k)}} J = \mathbf{g} \mathbf{h}^{(k-1)\top} + \lambda \nabla_{\mathbf{W}^{(k)}} \Omega(\theta)$$

Propagate the gradients w.r.t. the next lower-level hidden layer's activations:

$$\mathbf{g} \leftarrow \nabla_{\mathbf{h}^{(k-1)}} J = \mathbf{W}^{(k)\top} \mathbf{g}$$

**end for**

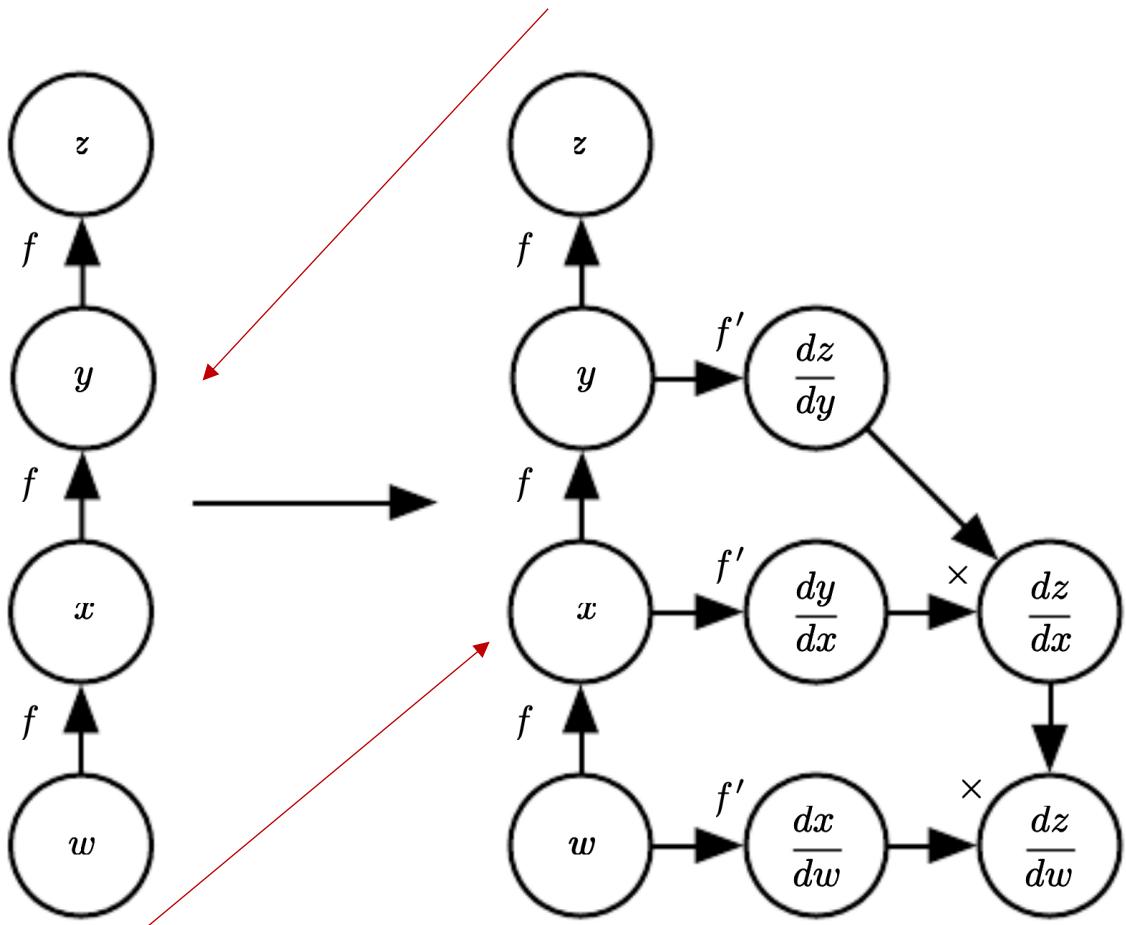
# Addestramento

- Backpropagation

- Il calcolo delle derivate composte può essere svolto attraverso il grafo computazionale

$$\begin{aligned}\frac{\partial z}{\partial w} &= \frac{\partial z}{\partial y} \frac{\partial y}{\partial x} \frac{\partial x}{\partial w} \\ &= f'(y) f'(x) f'(w) \\ &= f'(f(f(w))) f'(f(w)) f'(w)\end{aligned}$$

(Py)Torch usa la *symbol-to-number differentiation*: gli ingressi  $x$  fluiscano nel grafo e via via si calcolano le derivate



Tensorflow usa la *symbol-to-symbol differentiation*: gli ingressi  $x$  fluiscano in un grafo di computazione simbolica delle attivazioni e delle derivate

# Addestramento

**Require:**  $\mathbb{T}$ , the target set of variables whose gradients must be computed.

**Require:**  $\mathcal{G}$ , the computational graph

**Require:**  $z$ , the variable to be differentiated

Let  $\mathcal{G}'$  be  $\mathcal{G}$  pruned to contain only nodes that are ancestors of  $z$  and descendants of nodes in  $\mathbb{T}$ .

Initialize `grad_table`, a data structure associating tensors to their gradients

`grad_table`[ $z$ ]  $\leftarrow 1$

**for**  $\mathbf{V}$  in  $\mathbb{T}$  **do**

`build_grad`( $\mathbf{V}, \mathcal{G}, \mathcal{G}', \text{grad\_table}$ )

**end for**

Return `grad_table` restricted to  $\mathbb{T}$

Uso del grafo computazionale

# Addestramento

Require:  $\mathbf{V}$ , the variable whose gradient should be added to  $\mathcal{G}$  and `grad_table`

Require:  $\mathcal{G}$ , the graph to modify

Require:  $\mathcal{G}'$ , the restriction of  $\mathcal{G}$  to nodes that participate in the gradient

Require: `grad_table`, a data structure mapping nodes to their gradients

**if**  $\mathbf{V}$  is in `grad_table` **then**

    Return `grad_table`[ $\mathbf{V}$ ]

**end if**

$i \leftarrow 1$

**for**  $\mathbf{C}$  in `get_consumers`( $\mathbf{V}, \mathcal{G}'$ ) **do**

$\text{op} \leftarrow \text{get\_operation}(\mathbf{C})$

$\mathbf{D} \leftarrow \text{build_grad}(\mathbf{C}, \mathcal{G}, \mathcal{G}', \text{grad\_table})$

$\mathbf{G}^{(i)} \leftarrow \text{op.bprop}(\text{get\_inputs}(\mathbf{C}, \mathcal{G}'), \mathbf{V}, \mathbf{D})$

$i \leftarrow i + 1$

**end for**

$\mathbf{G} \leftarrow \sum_i \mathbf{G}^{(i)}$

`grad_table`[ $\mathbf{V}$ ] =  $\mathbf{G}$

Insert  $\mathbf{G}$  and the operations creating it into  $\mathcal{G}$

Return  $\mathbf{G}$

Uso del grafo computazionale:  
Souboutine `build_grad`

# Addestramento

- Stochastic Gradient Descent (SGD)
  - L'algoritmo di Gradient Descent stabilisce che una funzione  $f(\mathbf{x})$  può essere minimizzata muovendo  $\mathbf{x}$  lungo la direzione del gradiente di  $f$ :

$$\mathbf{x} \leftarrow \mathbf{x} - \varepsilon \nabla_{\mathbf{x}} f$$

- Il gradiente di  $J(\boldsymbol{\theta})$  è la media di tutti i valori di gradiente per tutti gli  $m$  ingressi: il costo computazionale è  $O(m)$  che può diventare molto elevato

$$J(\boldsymbol{\theta}) = \mathbb{E}_{\mathbf{x}, y \sim \hat{p}_{\text{data}}} L(\mathbf{x}, y, \boldsymbol{\theta}) = \frac{1}{m} \sum_{i=1}^m L(\mathbf{x}^{(i)}, y^{(i)}, \boldsymbol{\theta})$$

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \frac{1}{m} \sum_{i=1}^m \nabla_{\boldsymbol{\theta}} L(\mathbf{x}^{(i)}, y^{(i)}, \boldsymbol{\theta})$$

# Addestramento

- Stochastic Gradient Descent (SGD)
  - L'algoritmo SGD campiona  $m'$  ingressi costituendo un *minibatch*  $\mathbb{B}$  di alcune decine/centinaia di esempi sui quali si calcola il gradiente e si fa l'aggiornamento dell'algoritmo
  - Si elimina la dipendenza dal numero dei campioni

$$\mathbf{g} = \frac{1}{m'} \nabla_{\boldsymbol{\theta}} \sum_{i=1}^{m'} L(\mathbf{x}^{(i)}, y^{(i)}, \boldsymbol{\theta}) \quad \mathbb{B} = \{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m')}\}$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon \mathbf{g}$$

# Regolarizzazione

- Le tecniche di regolarizzazione sono mirate a ridurre l'errore di generalizzazione e non quello di addestramento
- Vedremo alcune tecniche tra le più diffuse
  - Uso di norme come termini di regolarizzazione
  - Data augmentation
  - Multi-task learning
  - Early stopping
  - Dropout

# Regolarizzazione

- Uso della norma  $L^2$ 
  - Il termine di regolarizzazione non interessa i bias, ma solo i pesi ed è un termine quadratico (per esempio il weight decay)
  - La regolarizzazione sui bias porta a underfitting

$$\tilde{J}(\mathbf{w}; \mathbf{X}, \mathbf{y}) = \frac{\alpha}{2} \mathbf{w}^\top \mathbf{w} + J(\mathbf{w}; \mathbf{X}, \mathbf{y})$$

$$\mathbf{w} \leftarrow \mathbf{w} - \epsilon (\alpha \mathbf{w} + \nabla_{\mathbf{w}} J(\mathbf{w}; \mathbf{X}, \mathbf{y}))$$

# Regolarizzazione

- Uso della norma  $L^2$ 
  - Assumiamo  $J$  sempre quadratica in un piccolo intorno del suo minimo  $\mathbf{w}^*$

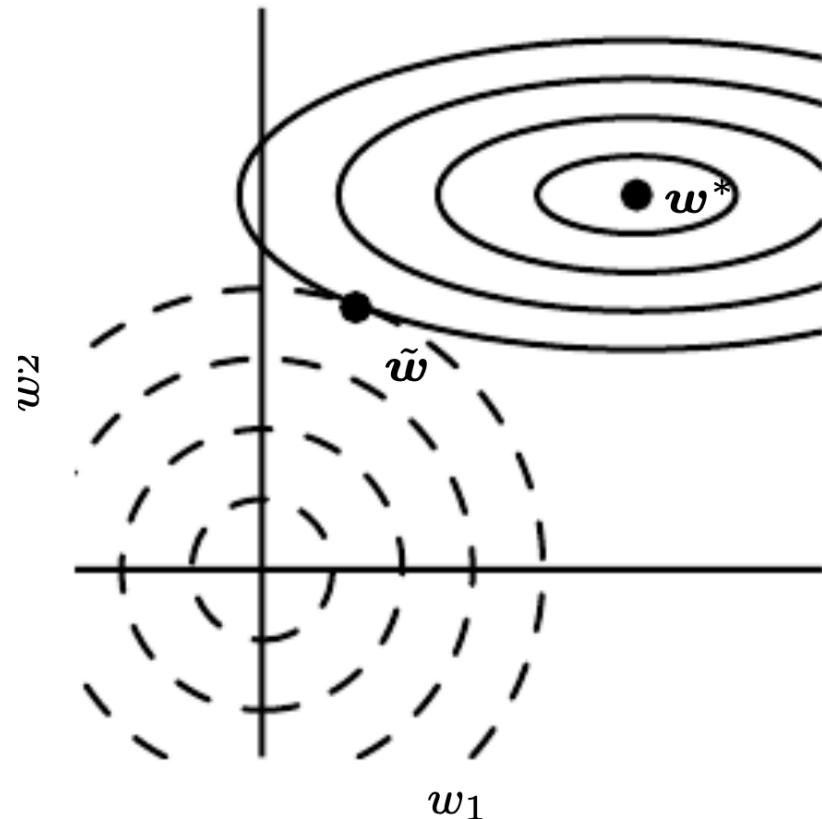
$$\hat{J}(\boldsymbol{\theta}) = J(\mathbf{w}^*) + \frac{1}{2}(\mathbf{w} - \mathbf{w}^*)^\top \mathbf{H}(\mathbf{w} - \mathbf{w}^*) \quad \mathbf{H} \text{ Matrice Hessiana di } J$$

- Si può mostrare che il termine di regolarizzazione individua un punto di minimo  $\hat{\mathbf{w}}$  che è una versione scalata di  $\mathbf{w}^*$  lungo gli autovalori di  $\mathbf{H}$  secondo dei coefficienti dati da:

$$\frac{\lambda_i}{\lambda_i + \alpha}$$

# Regolarizzazione

- Uso della norma  $L^2$ 
  - Esempio bidimensionale in cui  $\hat{w}$  si trova all'equilibrio tra i minimi di  $J$  e del termine di regolarizzazione
  - Viene fortemente scalato il parametro nella direzione di minore variazione di  $J$  nell'intorno di  $w^*$
  - Viene preservata la tendenza di  $J$  a variare lungo certe direzioni preferenziali

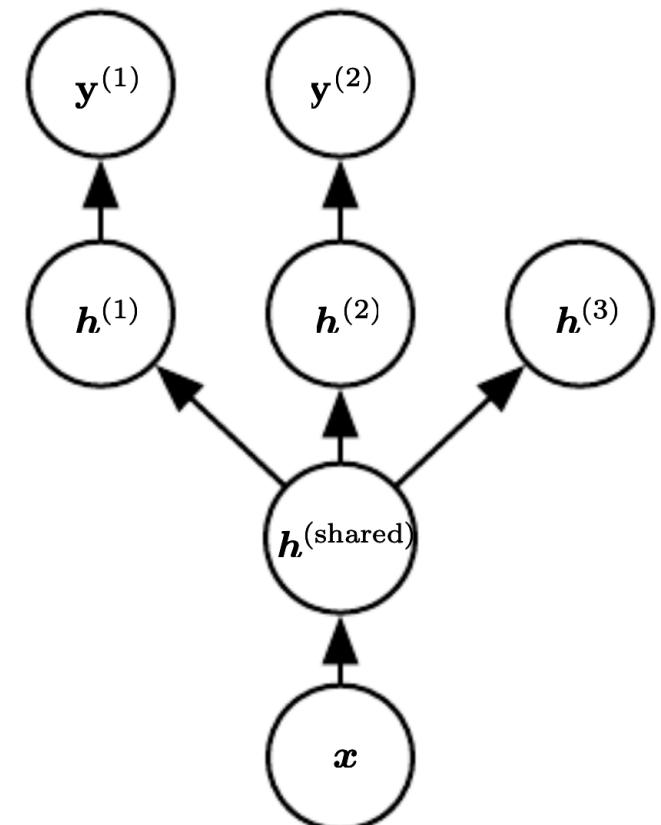


# Regolarizzazione

- Data augmentation
  - Il numero dei campioni viene aumentato artificiosamente per consentire una migliore generalizzazione
  - Si possono usare trasformazioni dei dati che possono essere indotte dal dominio di applicazione (ad es. trasformazioni affini su immagini)
  - Si può iniettare rumore nei dati di ingresso (ha l'effetto di una regolarizzazione  $L^2$ ) o anche nei pesi degli strati nascosti

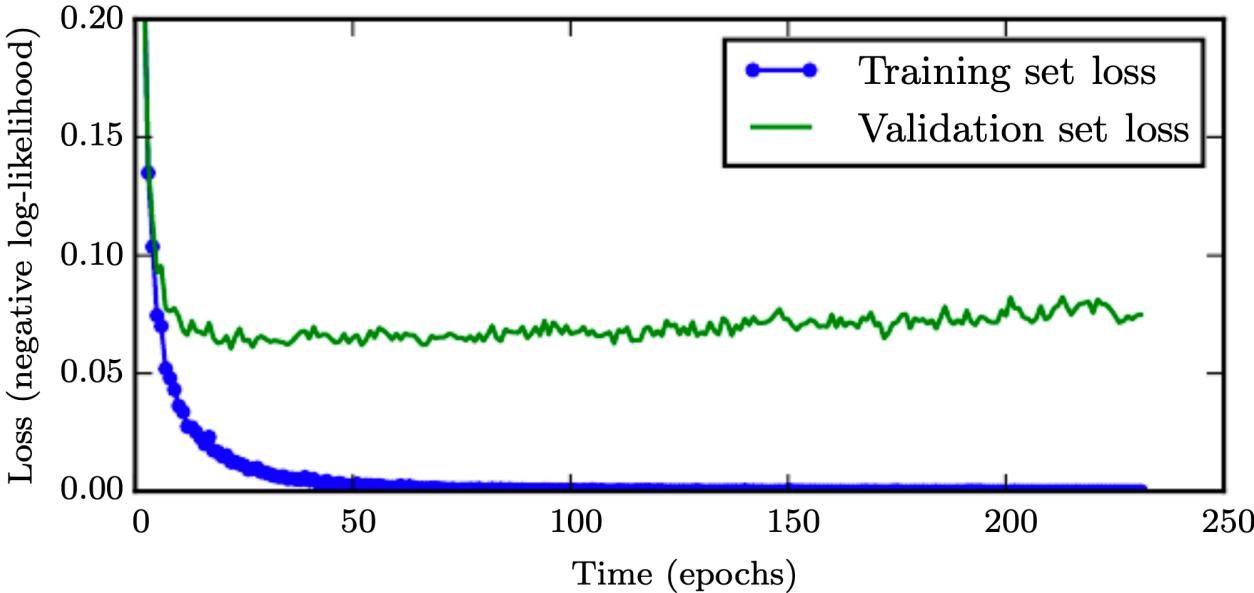
# Regolarizzazione

- Multi-task learning
  - Condivisione della rete per addestrarla su diversi task affini
  - Parametri specifici per ogni task nei layer vicini alle uscite
  - Parametri condivisi nei layer di ingresso



# Regolarizzazione

- Early stopping
  - Un addestramento che si protrae per un numero eccessivo di epoche porta a overfitting e riduce la capacità di generalizzare
  - L'early stopping è un algoritmo per fermare l'addestramento quando si osserva che la loss, valutata su un apposito validation set, comincia a risalire



Esperimento su una rete con unità maxout addestrata sul data set MNIST

# Regolarizzazione

- Early stopping
  - Si utilizza un parametro di *pazienza* che indica il numero di epoche che l'algoritmo attende per osservare un definitivo incremento dell'errore commesso sul set di validazione

Let  $n$  be the number of steps between evaluations.

Let  $p$  be the “patience,” the number of times to observe worsening validation set error before giving up.

Let  $\theta_o$  be the initial parameters.

```
 $\theta \leftarrow \theta_o$ 
 $i \leftarrow 0$ 
 $j \leftarrow 0$ 
 $v \leftarrow \infty$ 
 $\theta^* \leftarrow \theta$ 
 $i^* \leftarrow i$ 
while  $j < p$  do
    Update  $\theta$  by running the training algorithm for  $n$  steps.
     $i \leftarrow i + n$ 
     $v' \leftarrow \text{ValidationSetError}(\theta)$ 
    if  $v' < v$  then
         $j \leftarrow 0$ 
         $\theta^* \leftarrow \theta$ 
         $i^* \leftarrow i$ 
         $v \leftarrow v'$ 
    else
         $j \leftarrow j + 1$ 
    end if
end while
```

Best parameters are  $\theta^*$ , best number of training steps is  $i^*$ .

# Regolarizzazione

- **Early stopping**
  - Let  $\mathbf{X}^{(\text{train})}$  and  $\mathbf{y}^{(\text{train})}$  be the training set.
  - Split  $\mathbf{X}^{(\text{train})}$  and  $\mathbf{y}^{(\text{train})}$  into  $(\mathbf{X}^{(\text{subtrain})}, \mathbf{X}^{(\text{valid})})$  and  $(\mathbf{y}^{(\text{subtrain})}, \mathbf{y}^{(\text{valid})})$  respectively.
  - Run early stopping (algorithm 7.1) starting from random  $\boldsymbol{\theta}$  using  $\mathbf{X}^{(\text{subtrain})}$  and  $\mathbf{y}^{(\text{subtrain})}$  for training data and  $\mathbf{X}^{(\text{valid})}$  and  $\mathbf{y}^{(\text{valid})}$  for validation data. This returns  $i^*$ , the optimal number of steps.
  - Set  $\boldsymbol{\theta}$  to random values again.
  - Train on  $\mathbf{X}^{(\text{train})}$  and  $\mathbf{y}^{(\text{train})}$  for  $i^*$  steps.
- Strategia di addestramento 1: si ottiene  $i^*$  dall'early stopping e si riaddestra su tutto il training set; non è garantito il raggiungimento della loss ottenuta dall'early stopping

# Regolarizzazione

- **Early stopping**

Let  $\mathbf{X}^{(\text{train})}$  and  $\mathbf{y}^{(\text{train})}$  be the training set.

Split  $\mathbf{X}^{(\text{train})}$  and  $\mathbf{y}^{(\text{train})}$  into  $(\mathbf{X}^{(\text{subtrain})}, \mathbf{X}^{(\text{valid})})$  and  $(\mathbf{y}^{(\text{subtrain})}, \mathbf{y}^{(\text{valid})})$  respectively.

Run early stopping (algorithm 7.1) starting from random  $\boldsymbol{\theta}$  using  $\mathbf{X}^{(\text{subtrain})}$  and  $\mathbf{y}^{(\text{subtrain})}$  for training data and  $\mathbf{X}^{(\text{valid})}$  and  $\mathbf{y}^{(\text{valid})}$  for validation data. This updates  $\boldsymbol{\theta}$ .

$$\epsilon \leftarrow J(\boldsymbol{\theta}, \mathbf{X}^{(\text{subtrain})}, \mathbf{y}^{(\text{subtrain})})$$

**while**  $J(\boldsymbol{\theta}, \mathbf{X}^{(\text{valid})}, \mathbf{y}^{(\text{valid})}) > \epsilon$  **do**

    Train on  $\mathbf{X}^{(\text{train})}$  and  $\mathbf{y}^{(\text{train})}$  for  $n$  steps.

**end while**

- Strategia di addestramento 2: si riaddestra su tutto il training set a partire dai parametri  $\boldsymbol{\theta}^*$  ottenuti dall'early stopping controllando che la loss calcolata sui parametri via via aggiornati scenda al di sotto di quella ottenuta con l'early stopping

# Regolarizzazione

- Early stopping
  - Anche in questo caso si ottiene un effetto di regolarizzazione analogo all'uso di una norma  $L^2$
  - Si può mostrare che il troncamento dell'addestramento a  $\tau$  epoche, mentre la soluzione procede verso il minimo  $\mathbf{w}^*$  di  $J$ , ha un effetto inversamente proporzionale al coefficiente di regolarizzazione:

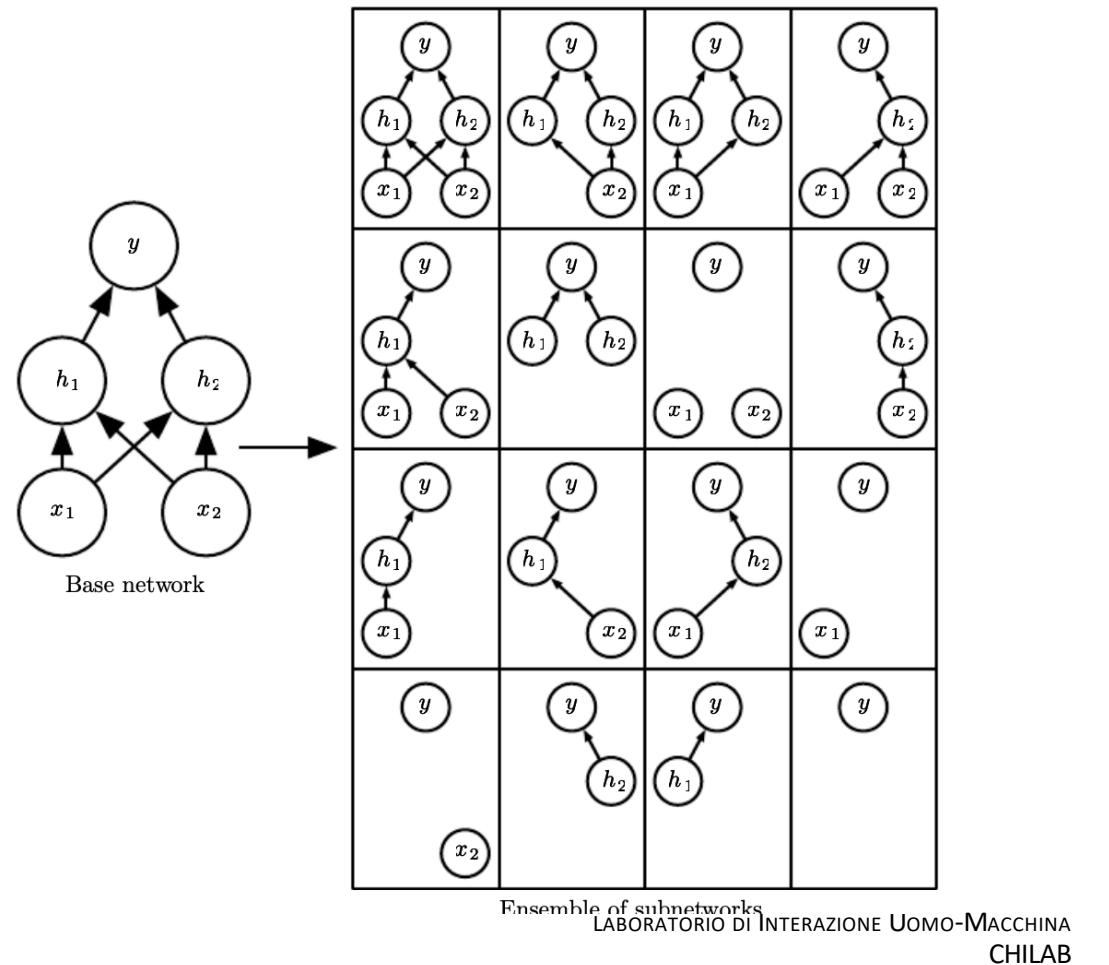
$$\alpha \approx \frac{1}{\tau \epsilon}$$

# Regolarizzazione

- Dropout
  - E' noto che i metodi di «ensemble di modelli», come il bagging, hanno una performance più accurata perché ogni modello ha una performance lievemente diversa dagli altri sullo stesso test set per cui la media/maggioranza delle predizioni risulta essere più accurata della singola predizione
  - Il dropout è una tecnica statistica per fare pruning delle connessioni tra le unità generando di fatto un ensemble di modelli che vengono addestrati contemporaneamente

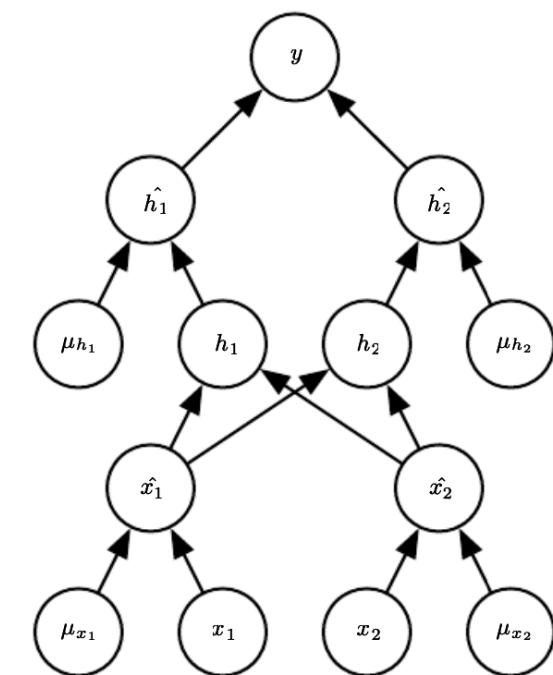
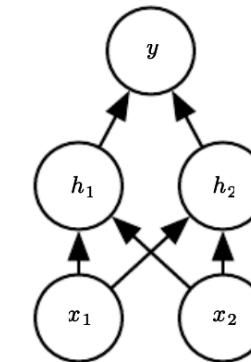
# Regolarizzazione

- Dropout
  - Rimuovendo via via le diverse connessioni si ottengono le sottoreti della rete di partenza



# Regolarizzazione

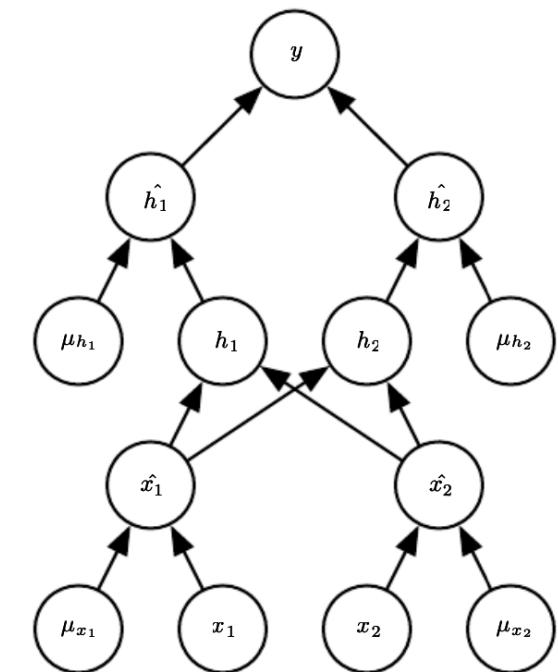
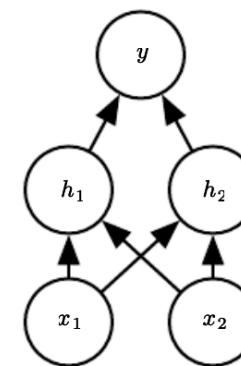
- Dropout
  - Sia  $\mu$  il vettore binario che maschera le attivazioni delle unità per cui ogni unità si attiverà come  $\mu_i h_i$
  - Ogni unità sarà inclusa nella rete con una certa probabilità: questo è un iperparametro
    - Dropout 0.8 per le unità di ingresso
    - Dropout 0.5 per le unità nascoste
  - ad ogni minibatch viene campionata casualmente una maschera diversa nel rispetto del valore di dropout con distribuzione  $p(\mu)$



# Regolarizzazione

- Dropout
  - Siamo in presenza di un ensemble di modelli, al variare di  $\mu$ , per cui si dovrà minimizzare globalmente una funzione di costo che è  $\mathbb{E}_{\mu}[J(\mathbf{x}; \boldsymbol{\theta}, \mu)]$ 
    - Si fa una stima con poche decine di campioni per  $\mu$
    - Ogni modello viene addestrato con campionamento con rimpiazzo dal data set
  - La predizione di ogni singolo modello  $p(y|\mathbf{x}, \mu)$  viene inserita nel modello di ensemble, per esempio, con la media pesata

$$\sum_{\mu} p(\mu) p(y|\mathbf{x}, \mu)$$



# Ottimizzazione

- L'ottimizzazione per le DNN riguarda tutto l'insieme di algoritmi e di tecniche per accelerare la convergenza della procedura di minimizzazione del funzionale di costo
- In linea teorica si vuole minimizzare il cosiddetto *rischio empirico* cioè i valore atteso della loss posto che traiamo i nostri campioni di addestramento da una *distribuzione empirica* e non da quella vera

$$\mathbb{E}_{\mathbf{x}, \mathbf{y} \sim \hat{p}_{\text{data}}(\mathbf{x}, \mathbf{y})}[L(f(\mathbf{x}; \boldsymbol{\theta}), \mathbf{y})] = \frac{1}{m} \sum_{i=1}^m L(f(\mathbf{x}^{(i)}; \boldsymbol{\theta}), y^{(i)})$$

# Ottimizzazione

- Gli approcci alla minimizzazione più diffusi ed efficienti assumono che la funzione obiettivo sia *localmente approssimabile con una funzione quadratica*
  - Il gradient descent è ottimo esattamente sotto l'assunzione quadratica
  - Il metodo di Newton calcola esplicitamente la matrice Hessiana  $H$  della funzione obiettivo per effettuare la minimizzazione «ad un passo» e ha la medesima assunzione
  - Problemi di mal condizionamento numerico di  $H$  e di presenza di plateau o punti sella nella funzione di costo

# Ottimizzazione

- La minimizzazione diretta del rischio empirico porta in genere a overfitting
  - Il calcolo di *tutti* i gradienti è molto oneroso
  - La rete si addestrerà per valori di elevata capacità e tenderà a minimizzare il solo training error
- Si minimizzerà una funzione diversa selezionando un minibatch di campioni per volta e utilizzando l'algoritmo di Stochastic Gradient Descent (SGD) e le sue varianti

# Ottimizzazione

- SGD

**Require:** Learning rate schedule  $\epsilon_1, \epsilon_2, \dots$

**Require:** Initial parameter  $\theta$

$k \leftarrow 1$

**while** stopping criterion not met **do**

    Sample a minibatch of  $m$  examples from the training set  $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$  with corresponding targets  $\mathbf{y}^{(i)}$ .

    Compute gradient estimate:  $\hat{\mathbf{g}} \leftarrow \frac{1}{m} \nabla_{\theta} \sum_i L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$

    Apply update:  $\theta \leftarrow \theta - \epsilon_k \hat{\mathbf{g}}$

$k \leftarrow k + 1$

**end while**

# Ottimizzazione

- SGD
  - $\hat{g}$  calcolato sul minibatch consente all'algoritmo di convergere bene risparmiando molto onere computazionale
  - In genere il learning rate  $\varepsilon$  viene fatto decrescere linearmente
  - Il minibatch serve da regolarizzazione
  - Si presta ad essere eseguito in parallelo su architetture multi-core
  - Spesso si usano valori potenza di 2 da 32 a 256

# Ottimizzazione

- SGD con momentum
  - Si usa un termine di momento che accumula una media mobile dei passati gradienti
  - Il momento di Nesterov interessa anche l'aggiornamento di  $\theta$

$$\begin{aligned}\mathbf{v} &\leftarrow \alpha \mathbf{v} - \epsilon \nabla_{\boldsymbol{\theta}} \left( \frac{1}{m} \sum_{i=1}^m L(\mathbf{f}(\mathbf{x}^{(i)}; \boldsymbol{\theta}), \mathbf{y}^{(i)}) \right) \\ \boldsymbol{\theta} &\leftarrow \boldsymbol{\theta} + \mathbf{v}.\end{aligned}$$

$$\begin{aligned}\mathbf{v} &\leftarrow \alpha \mathbf{v} - \epsilon \nabla_{\boldsymbol{\theta}} \left[ \frac{1}{m} \sum_{i=1}^m L\left(\mathbf{f}(\mathbf{x}^{(i)}; \boldsymbol{\theta} + \alpha \mathbf{v}), \mathbf{y}^{(i)}\right) \right] \\ \boldsymbol{\theta} &\leftarrow \boldsymbol{\theta} + \mathbf{v},\end{aligned}$$

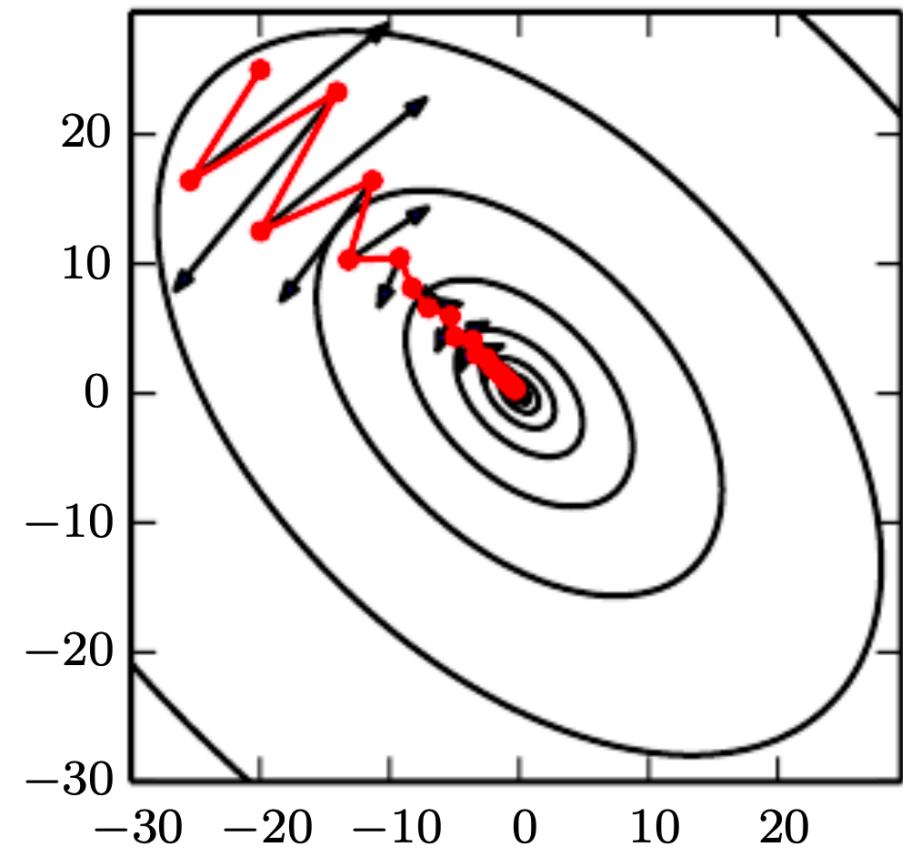
# Ottimizzazione

- SGD con momentum

- Il momentum fa tendere la soluzione verso direzioni a minore curvatura per evitare che la soluzione diverge

- La velocità della soluzione è: 
$$\frac{\epsilon \|\mathbf{g}\|}{1 - \alpha}$$

- $\alpha$  tipicamente 0.5, 0.9, 0.99



- Le frecce nere sono le direzioni del gradient descent
- I tratti rossi sono quelli del momentum

# Ottimizzazione

- Algoritmi con learning rate variabile
  - Adattano il learning rate scalandolo con un termine legato all'accumulazione dei quadrati dei gradienti lungo l'addestramento (momento del secondo ordine di  $\mathbf{g}$ )
  - AdaGrad: usa l'intera storia dei gradienti
  - RMSProp: usa una media mobile su una finestra con termine di decay e usa il momento di Nesterov
  - Adam: usa una regolarizzazione con i momenti del primo e del secondo ordine di  $\mathbf{g}$

# Ottimizzazione

**Require:** Global learning rate  $\epsilon$

**Require:** Initial parameter  $\theta$

**Require:** Small constant  $\delta$ , perhaps  $10^{-7}$ , for numerical stability

Initialize gradient accumulation variable  $r = \mathbf{0}$

**while** stopping criterion not met **do**

    Sample a minibatch of  $m$  examples from the training set  $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$  with corresponding targets  $\mathbf{y}^{(i)}$ .

    Compute gradient:  $\mathbf{g} \leftarrow \frac{1}{m} \nabla_{\theta} \sum_i L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$ .

    Accumulate squared gradient:  $\mathbf{r} \leftarrow \mathbf{r} + \mathbf{g} \odot \mathbf{g}$ .

    Compute update:  $\Delta \theta \leftarrow -\frac{\epsilon}{\delta + \sqrt{\mathbf{r}}} \odot \mathbf{g}$ . (Division and square root applied element-wise)

    Apply update:  $\theta \leftarrow \theta + \Delta \theta$ .

**end while**

# Ottimizzazione

**Require:** Global learning rate  $\epsilon$ , decay rate  $\rho$ , momentum coefficient  $\alpha$

**Require:** Initial parameter  $\theta$ , initial velocity  $v$

Initialize accumulation variable  $r = \mathbf{0}$

**while** stopping criterion not met **do**

    Sample a minibatch of  $m$  examples from the training set  $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$  with corresponding targets  $\mathbf{y}^{(i)}$ .

    Compute interim update:  $\tilde{\theta} \leftarrow \theta + \alpha v$ .

    Compute gradient:  $\mathbf{g} \leftarrow \frac{1}{m} \nabla_{\tilde{\theta}} \sum_i L(f(\mathbf{x}^{(i)}; \tilde{\theta}), \mathbf{y}^{(i)})$ .

    Accumulate gradient:  $r \leftarrow \rho r + (1 - \rho) \mathbf{g} \odot \mathbf{g}$ .

    Compute velocity update:  $v \leftarrow \alpha v - \frac{\epsilon}{\sqrt{r}} \odot \mathbf{g}$ .   ( $\frac{1}{\sqrt{r}}$  applied element-wise)

    Apply update:  $\theta \leftarrow \theta + v$ .

**end while**

# Ottimizzazione

**Require:** Step size  $\epsilon$  (Suggested default: 0.001)

**Require:** Exponential decay rates for moment estimates,  $\rho_1$  and  $\rho_2$  in  $[0, 1]$ .  
(Suggested defaults: 0.9 and 0.999 respectively)

**Require:** Small constant  $\delta$  used for numerical stabilization (Suggested default:  $10^{-8}$ )

**Require:** Initial parameters  $\theta$

    Initialize 1st and 2nd moment variables  $s = \mathbf{0}$ ,  $r = \mathbf{0}$

    Initialize time step  $t = 0$

**while** stopping criterion not met **do**

        Sample a minibatch of  $m$  examples from the training set  $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$  with corresponding targets  $\mathbf{y}^{(i)}$ .

        Compute gradient:  $\mathbf{g} \leftarrow \frac{1}{m} \nabla_{\theta} \sum_i L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$

$t \leftarrow t + 1$

        Update biased first moment estimate:  $\mathbf{s} \leftarrow \rho_1 \mathbf{s} + (1 - \rho_1) \mathbf{g}$

        Update biased second moment estimate:  $\mathbf{r} \leftarrow \rho_2 \mathbf{r} + (1 - \rho_2) \mathbf{g} \odot \mathbf{g}$

        Correct bias in first moment:  $\hat{\mathbf{s}} \leftarrow \frac{\mathbf{s}}{1 - \rho_1^t}$

        Correct bias in second moment:  $\hat{\mathbf{r}} \leftarrow \frac{\mathbf{r}}{1 - \rho_2^t}$

        Compute update:  $\Delta\theta = -\epsilon \frac{\hat{\mathbf{s}}}{\sqrt{\hat{\mathbf{r}}} + \delta}$     (operations applied element-wise)

        Apply update:  $\theta \leftarrow \theta + \Delta\theta$

**end while**

# Reti Convoluzionali

- Le Reti Neurali Convoluzionali (Convolutional Neural Networks – CNN) sono reti neurali mirate ad analizzare dati che abbiano una topologia a griglia
  - Immagini
  - Sequenze monodimensionali con finestra di analisi fissa
    - Campionamento di serie temporali
    - Serie discrete (frasi)
    - Molecole
    - ...

# Reti Convoluzionali

- La connessione tra due strati successivi usa l'operatore di convoluzione

$$S(i, j) = (K * I)(i, j) = \sum_m \sum_n I(i - m, j - n) K(m, n)$$

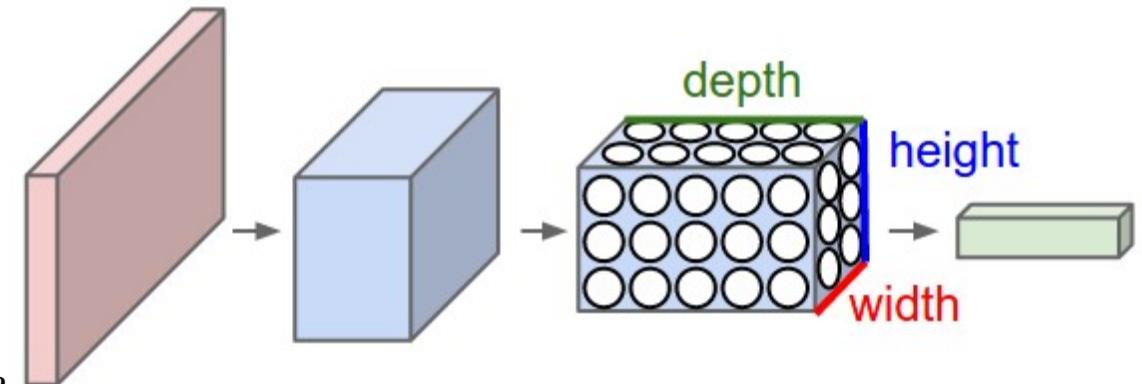
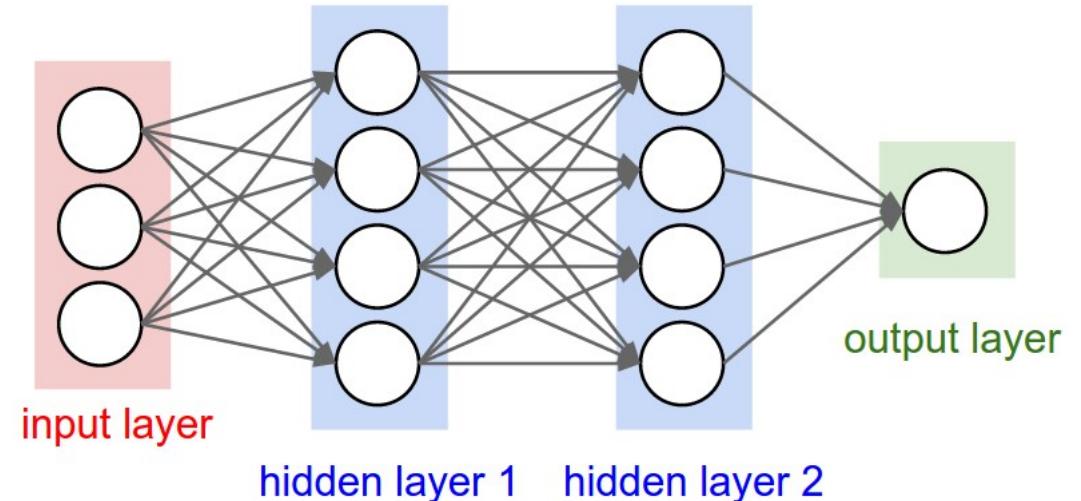
- L'operatore è applicato in realtà *sempre* a un tensore n-dimensionale anche nel caso che l'input sia monodimensionale
- In genere, dimensioni le del tensore di ingresso  $\mathbf{X}_{b,w,h,d}$  sono *batch, ampiezza, larghezza e profondità* (o canali)
  - Un vettore  $\mathbf{x} \in \mathbb{R}^d \rightarrow \mathbf{X}_{b,d,1,1}$

# Reti Convoluzionali

- La struttura del tensore deriva esplicitamente dal fatto che per le immagini la dimensione della profondità è quella dei tre «canali» R,G,B

- La dimensione del batch non rientra direttamente nell'operazione di convoluzione

$$Y_{i,j,k} = \sum_l \sum_m \sum_n X_{i-l, j-m, k-n} K_{l,m,n}$$

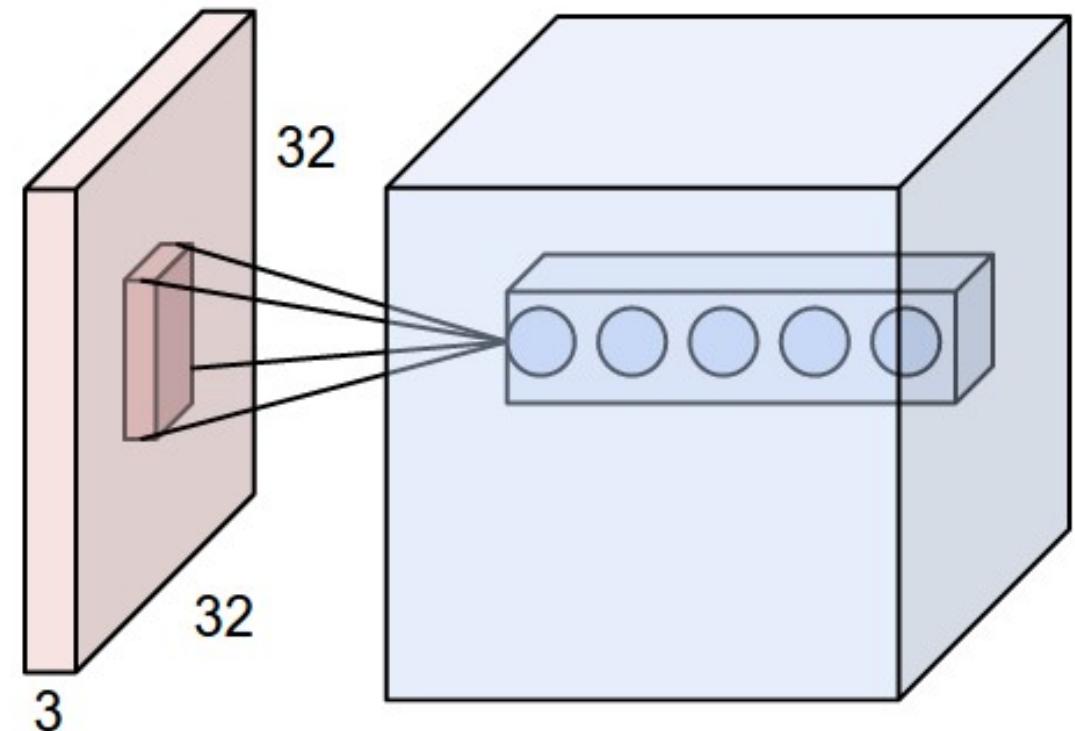


Fonte <https://cs231n.github.io/convolutional-networks/>

# Reti Convoluzionali

- La dimensione  $s$  del filtro lungo le dimensioni della griglia è un *iperparametro*
- Lungo la direzione della profondità la convoluzione è sempre completa (fully-connected) e pari alla profondità  $d$  dell'input
- Ogni layer convoluzionale può avere comunque *una sua profondità* diversa da quella dell'input e degli altri layer e che è un altro *iperparametro*

$$Y_{i,j,k} = \sum_{l=1}^s \sum_{m=1}^s \sum_{n=1}^d X_{i-l,j-m,k-n} K_{l,m,n}$$



Fonte <https://cs231n.github.io/convolutional-networks/>

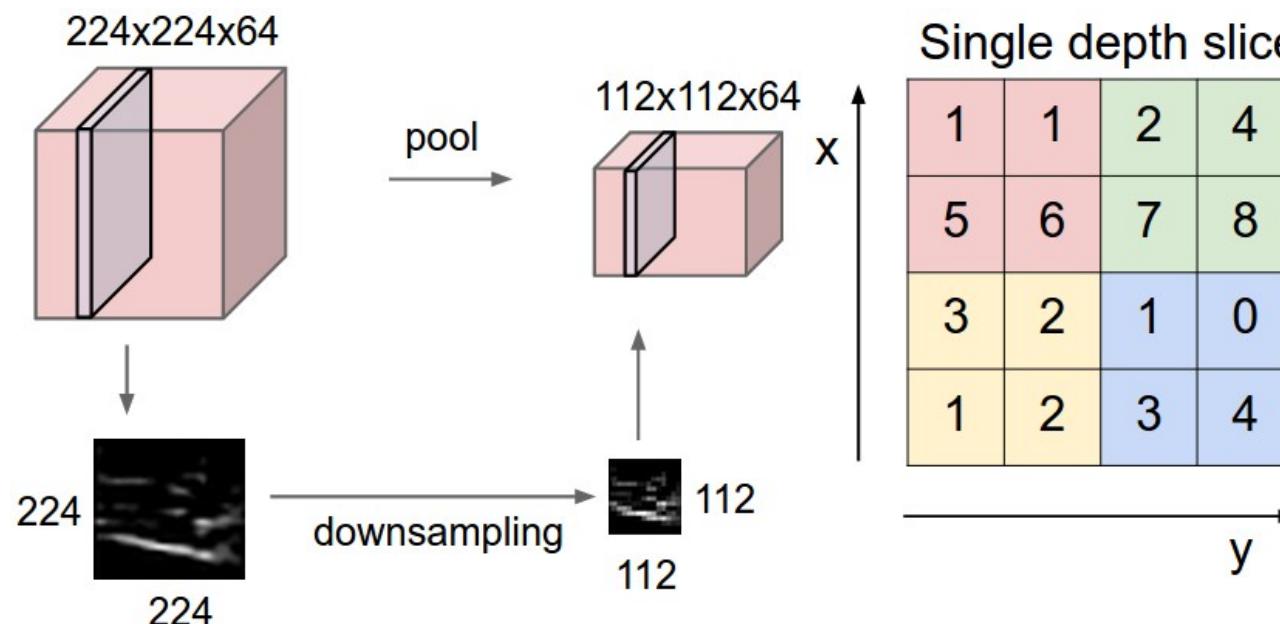
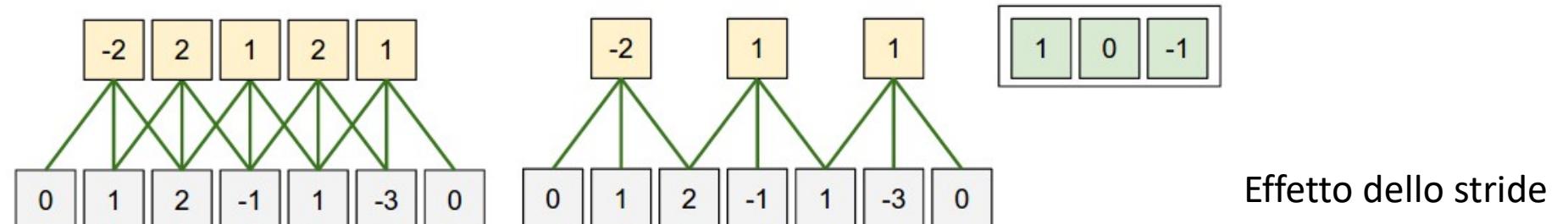
# Reti Convoluzionali

- Un layer convoluzionale «complesso» è in genere una sequenza di tre sub-layer
  - Convoluzionale: raccoglie gli input su una regione limitata e serve a estrarre caratteristiche locali
    - Nel senso della profondità vengono generate differenti *feature map*
  - ReLU: classico layer DNN per il calcolo della non linearità
    - Spesso il layer complesso è dato solo da Conv + ReLU
  - Pool (opzionale): effettua una sotto-campionamento degli ingressi per ottenere delle feature map a scala più piccola
    - Orientato ad ottenere feature aggregate invarianti alla posizione

# Reti Convoluzionali

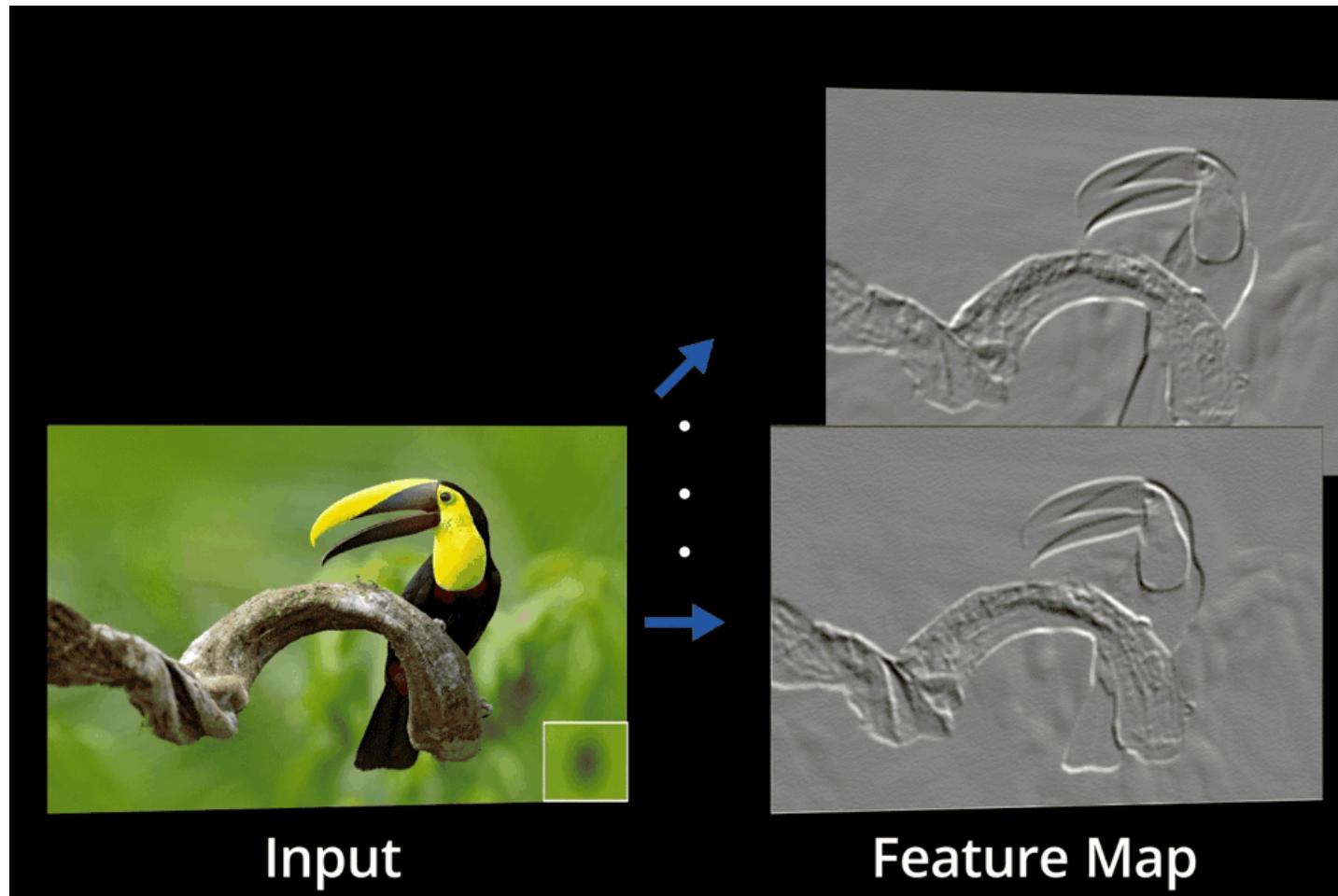
- Il layer di Pooling filtra la feature map in un piccolo intorno (generalmente 2x2) applicando diversi filtri. I più utilizzati sono
  - Max pooling
  - Average pooling
- Il sotto-campionamento si può ottenere anche applicando uno *stride* cioè uno step di salto nell'applicazione dei kernel convoluzionali o del Pooling
  - Spesso è necessario introdurre uno *zero-padding* cioè una opportuna cornice di valori nulli attorno alla feature map al fine di adeguare il passo del kernel e la sua dimensione con quelle della feature map

# Reti Convoluzionali



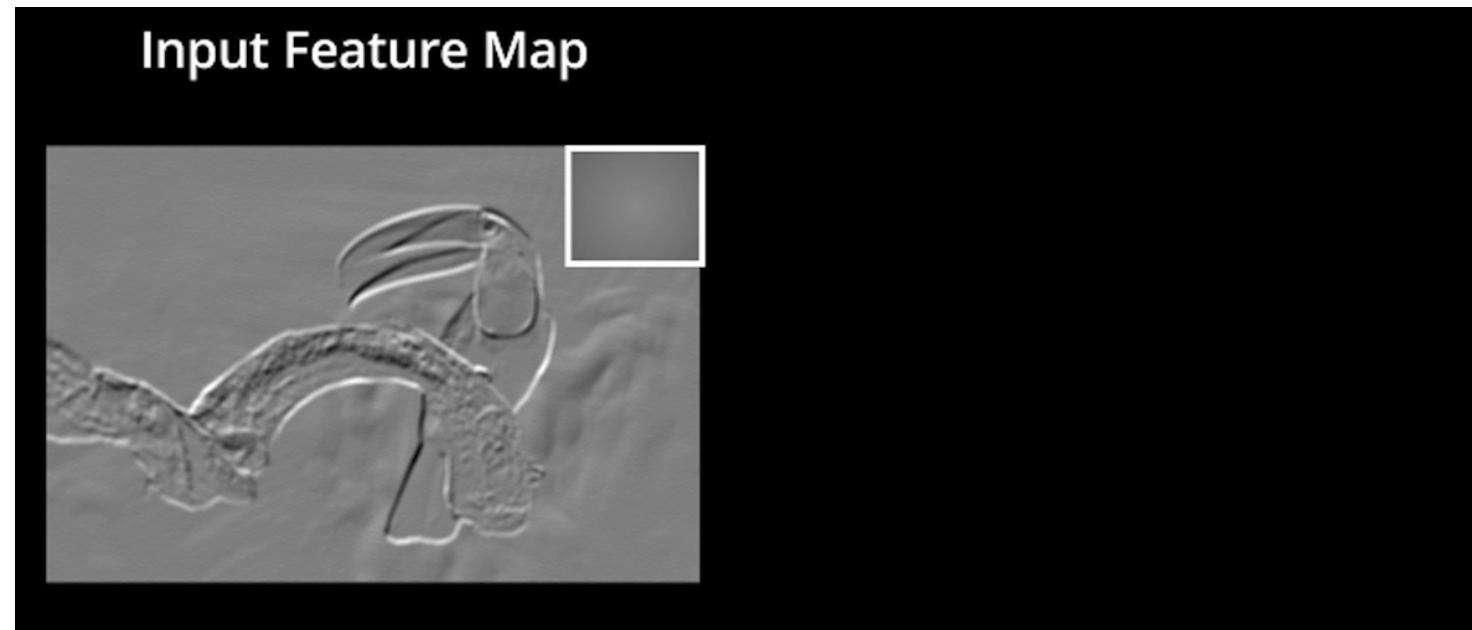
# Reti Convoluzionali

Conv layer



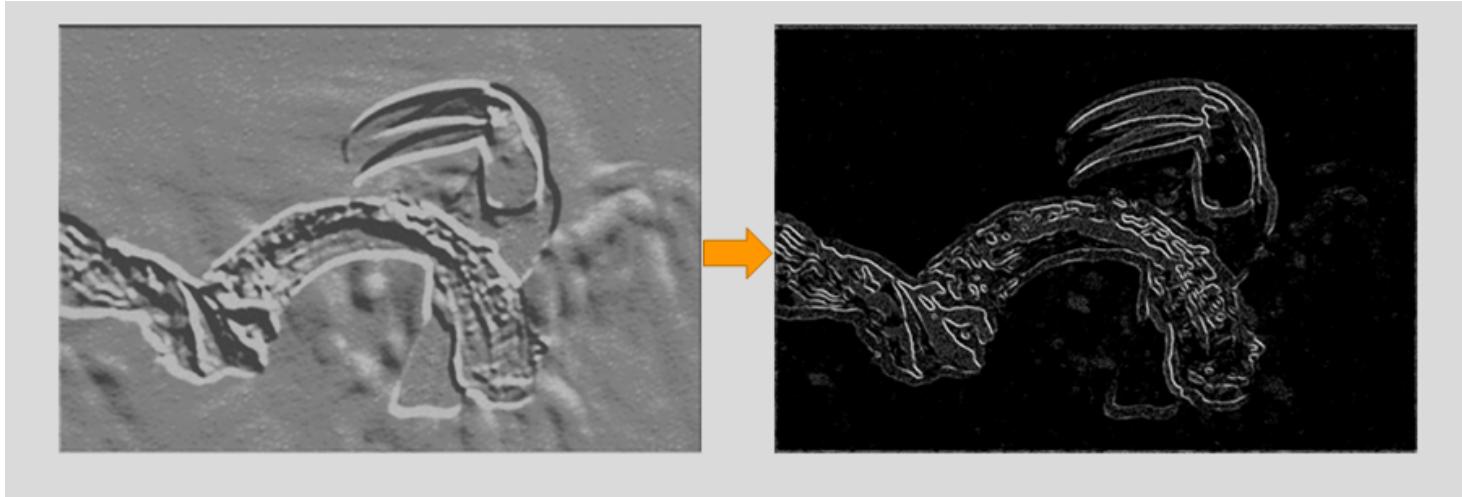
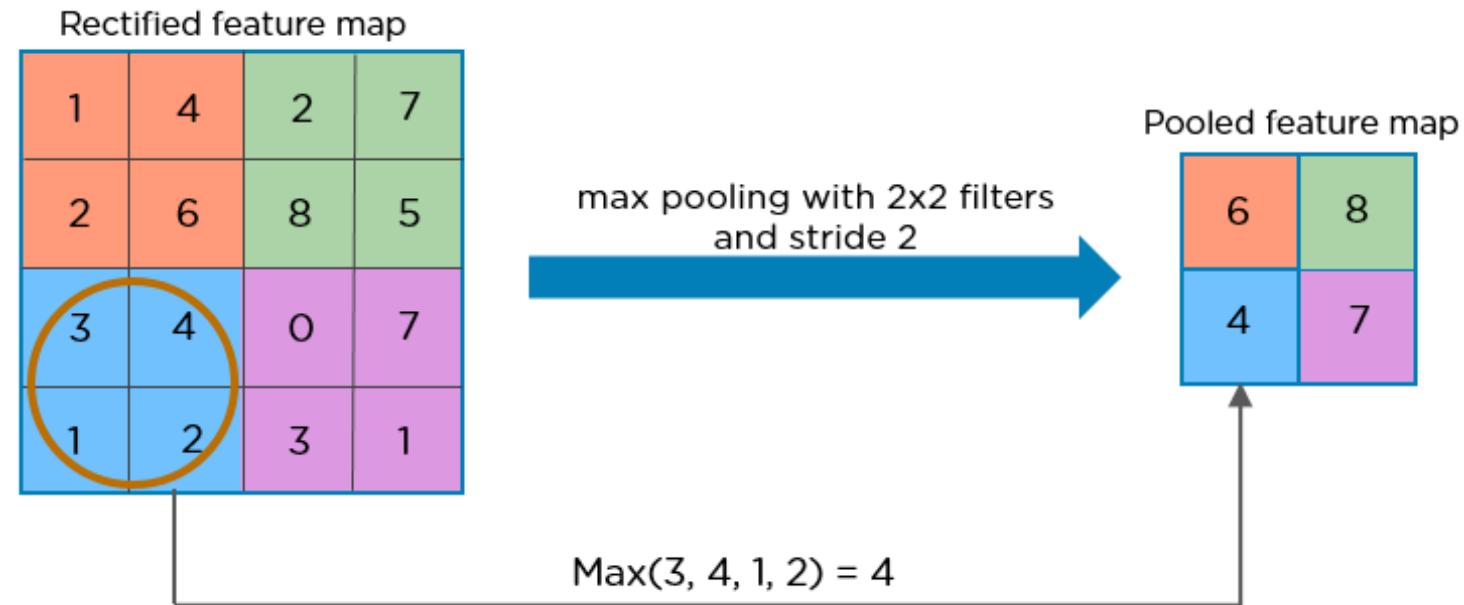
# Reti Convoluzionali

ReLU layer



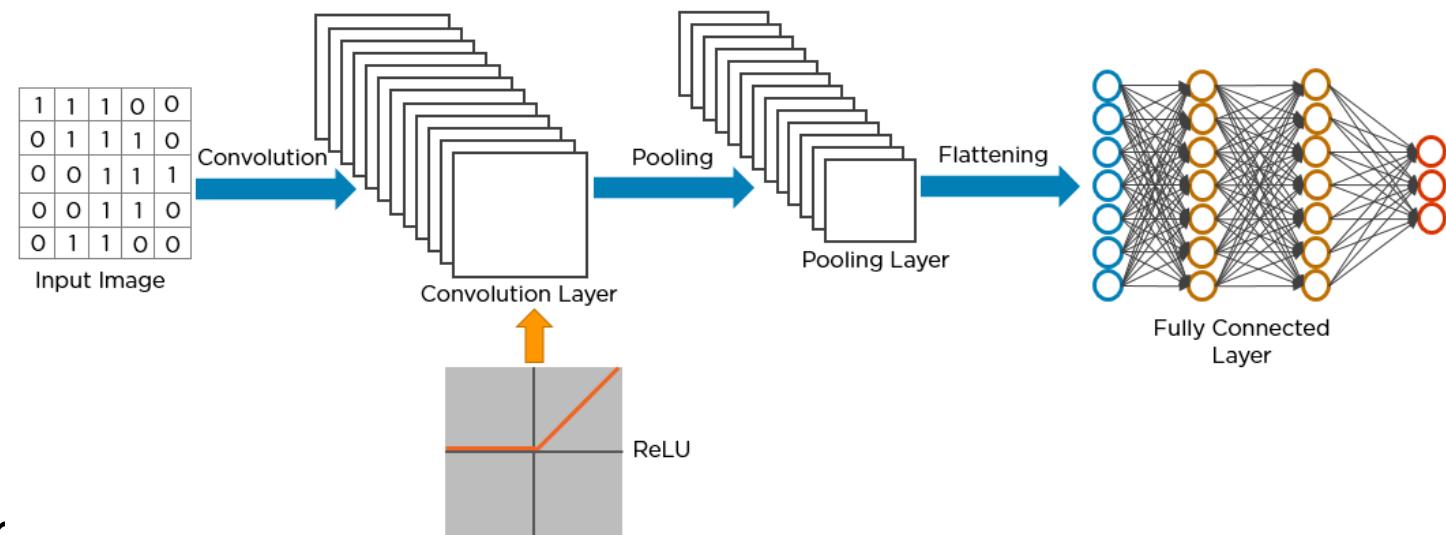
# Reti Convoluzionali

Max Pool 2x2 layer

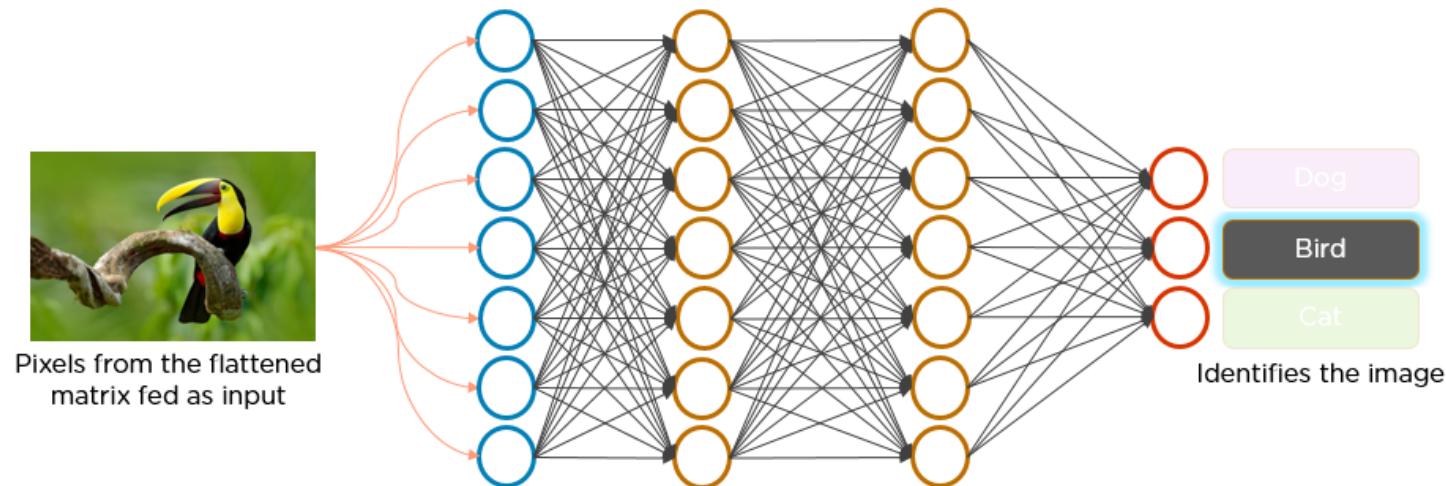


Fonte <https://www.simplilearn.com/tutorials/deep-learning-tutorial/convolutional-neural-network>

# Reti Convoluzionali

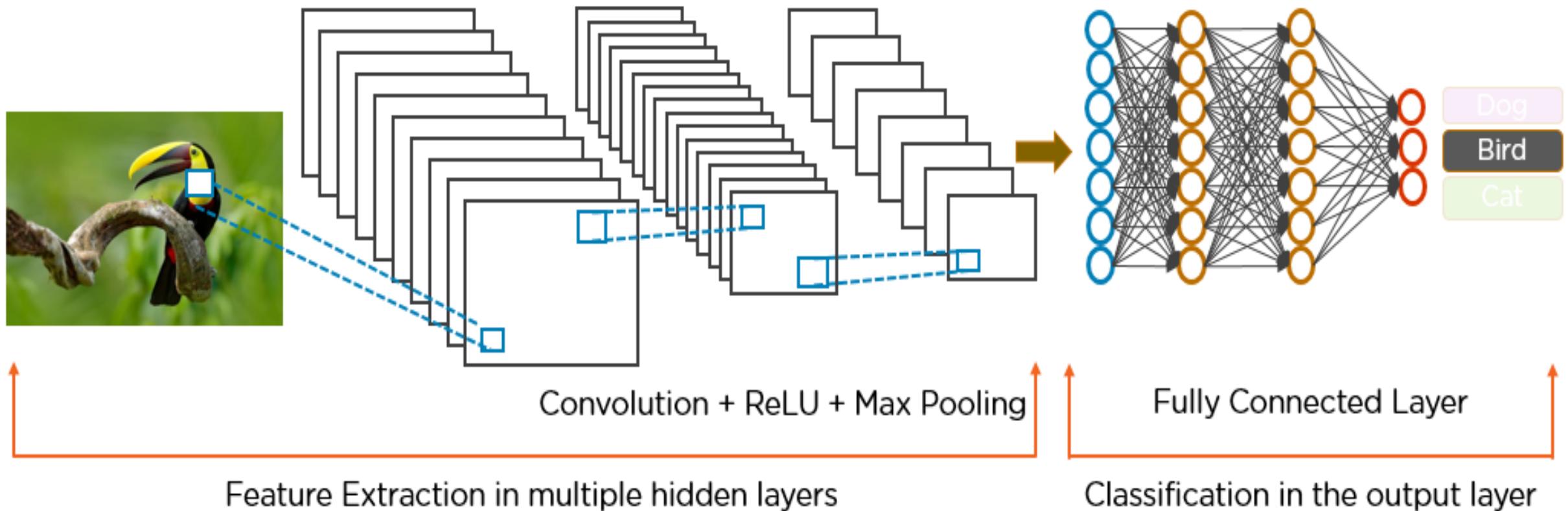


Flattening + Dense layer



Fonte <https://www.simplilearn.com/tutorials/deep-learning-tutorial/convolutional-neural-network>

# Reti Convoluzionali



La presenza di più layer complessi, in genere con molte feature map e con diversi livelli di sotto-campionamento consente di estrarre caratteristiche gerarchicamente sempre più aggregate e invarianti a scala e posizione

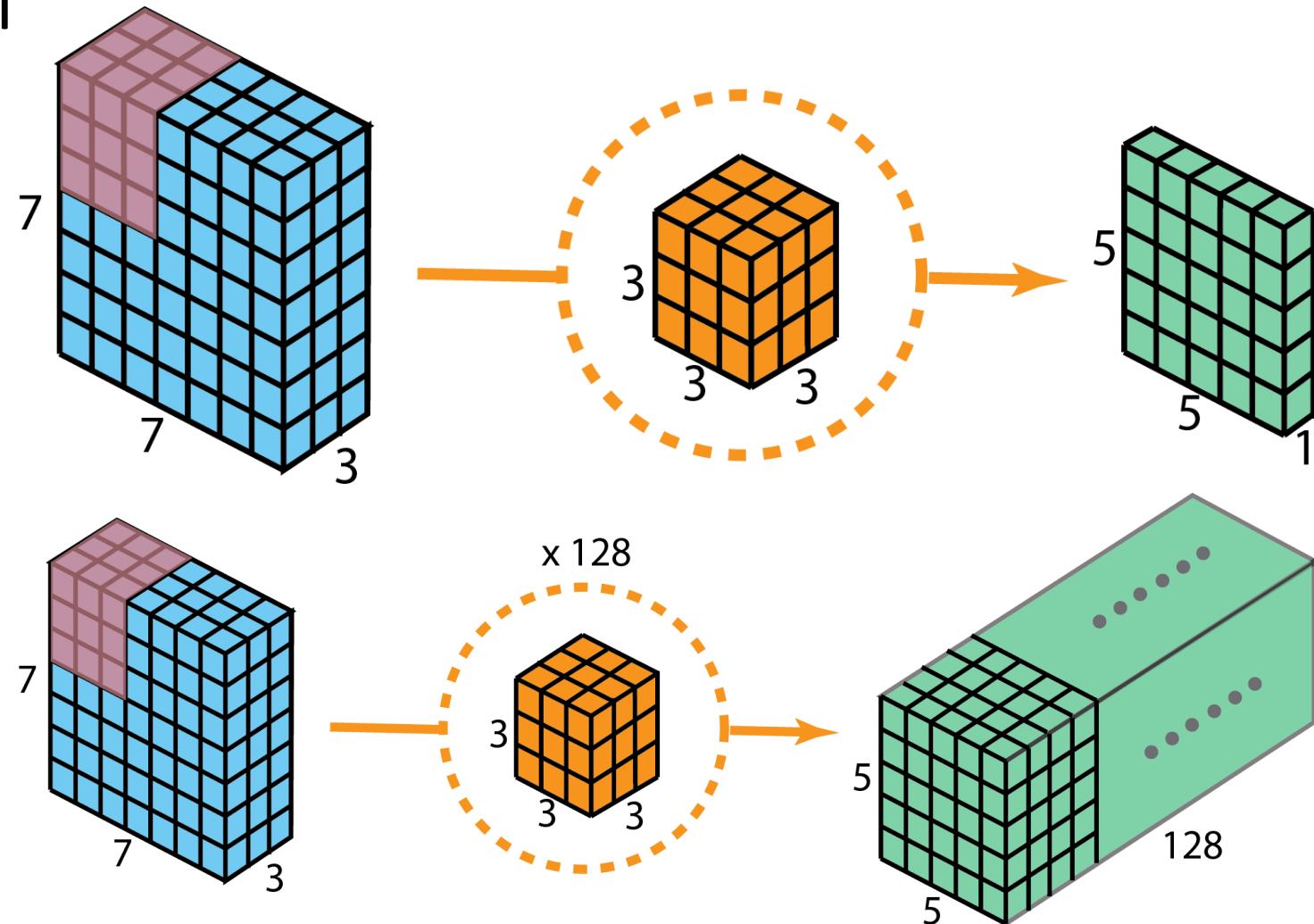
# Reti Convoluzionali

- Le CNN sono di derivazione biologica e si basano sugli studi percettivi della visione umana, ma le CNN moderne si sono sviluppate dal 2012 in poi
  - LeNet, AlexNet, VGGNet, GoogleNet, ResNet, ...
  - Reti sempre più profonde: spesso questi modelli generalisti addestrati a riconoscere decine di classi di oggetti sono usate come «backbone» di architetture più complesse che si addestrano tramite *transfer learning*
  - Numero di parametri che esplode
  - Necessità di reinterpretare l'operatore di convoluzione

# Reti Convoluzionali

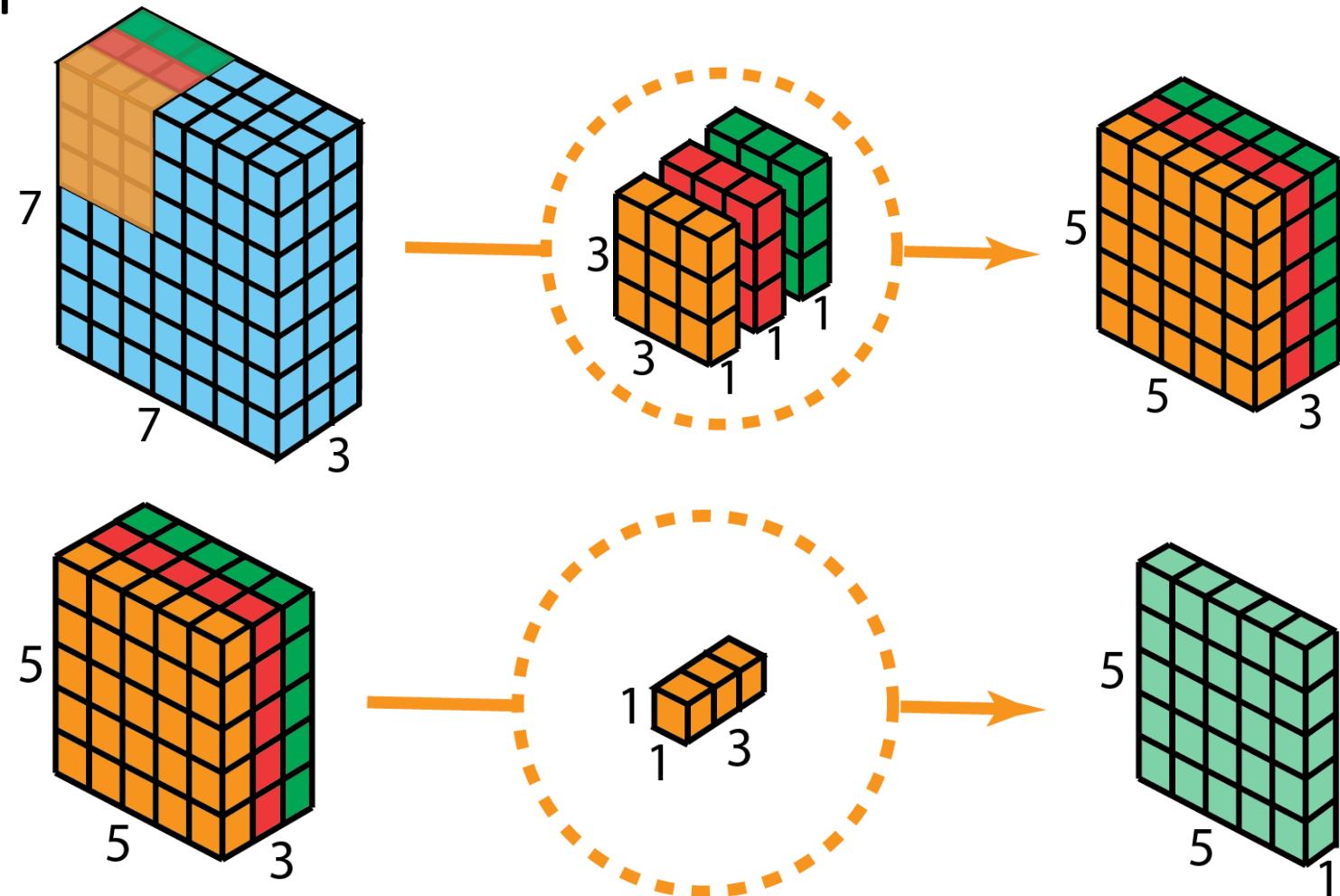
- Depthwise separable convolution
  - Una convoluzione da un layer  $w_i \times h_i \times d_i$  a uno  $w_o \times h_o \times d_o$  con un kernel  $s \times s \times d$ , comporta un numero di moltiplicazioni pari a:

$$d_o \times (s \times s \times d_i) \times w_o \times h_o$$

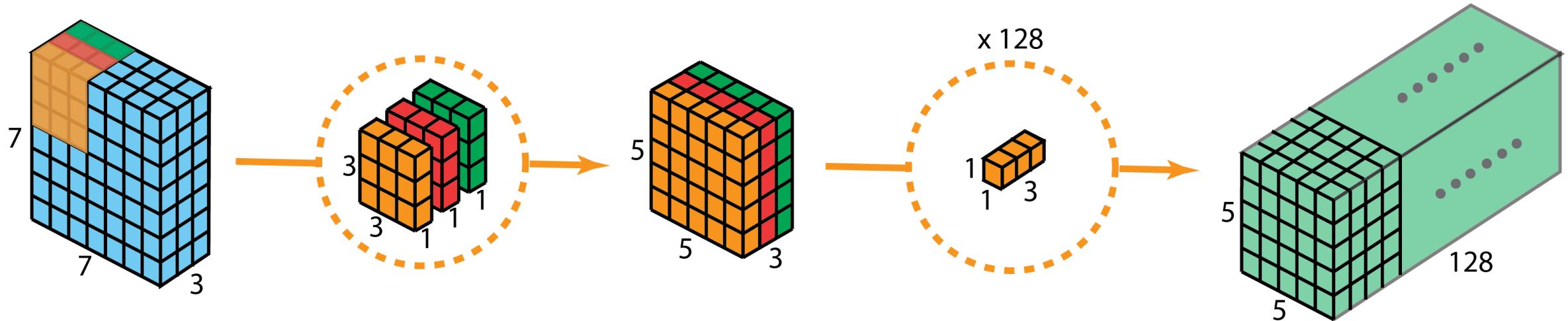


# Reti Convoluzionali

- Depthwise separable convolution
  - La convoluzione viene svolta in due fasi:
    - $d_i$  convoluzioni  $s \times s \times 1$
    - 1 convoluzione  $1 \times 1 \times d_i$
  - $d_o$  kernel  $1 \times 1 \times d_i$  forniscono il volume finale



# Reti Convoluzionali



- Costo della DSC in rapporto alla convoluzione normale

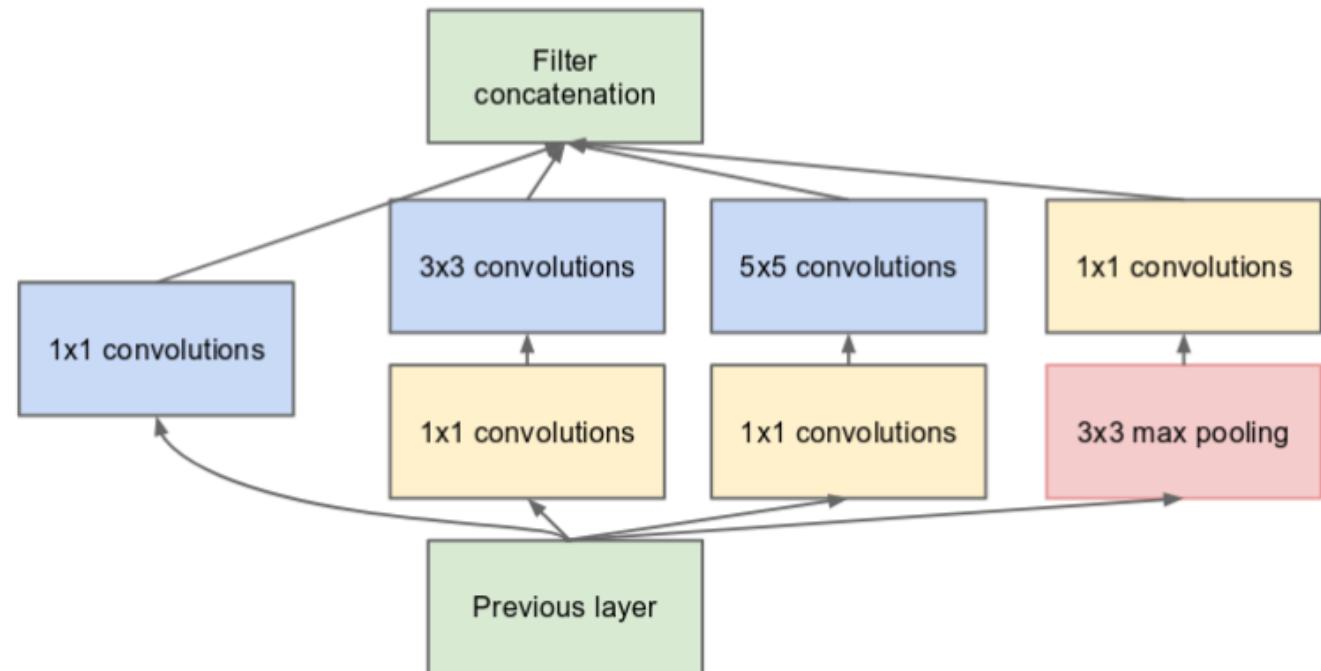
$$\begin{aligned} \frac{d_i(s \times s \times 1)w_o h_o + d_o(1 \times 1 \times d_i)w_o h_o}{d_o(s \times s \times d_i)w_o h_o} &= \\ = \frac{d_i s^2}{d_i d_o s^2} + \frac{d_o d_i}{d_o s^2 d_i} &= \frac{1}{d_o} + \frac{1}{s^2} \approx \frac{1}{s^2}, \quad d_o \gg s \end{aligned}$$

# Reti Convoluzionali

- Inception module e DSC (GoogleNet)

- I layer realizzati con Inception partono dall'idea di analizzare contemporaneamente l'input a più scale

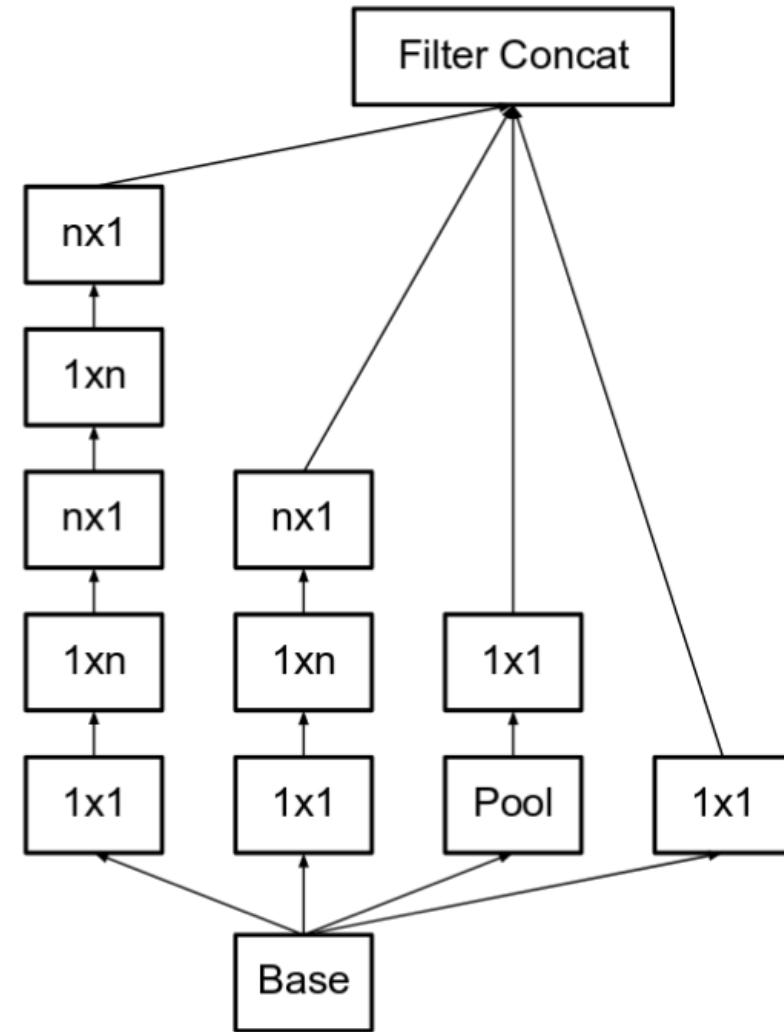
- Vengono introdotti dei blocchi convoluzionali  $1 \times 1$  per ridurre la dimensionalità dell'input



(b) Inception module with dimension reductions

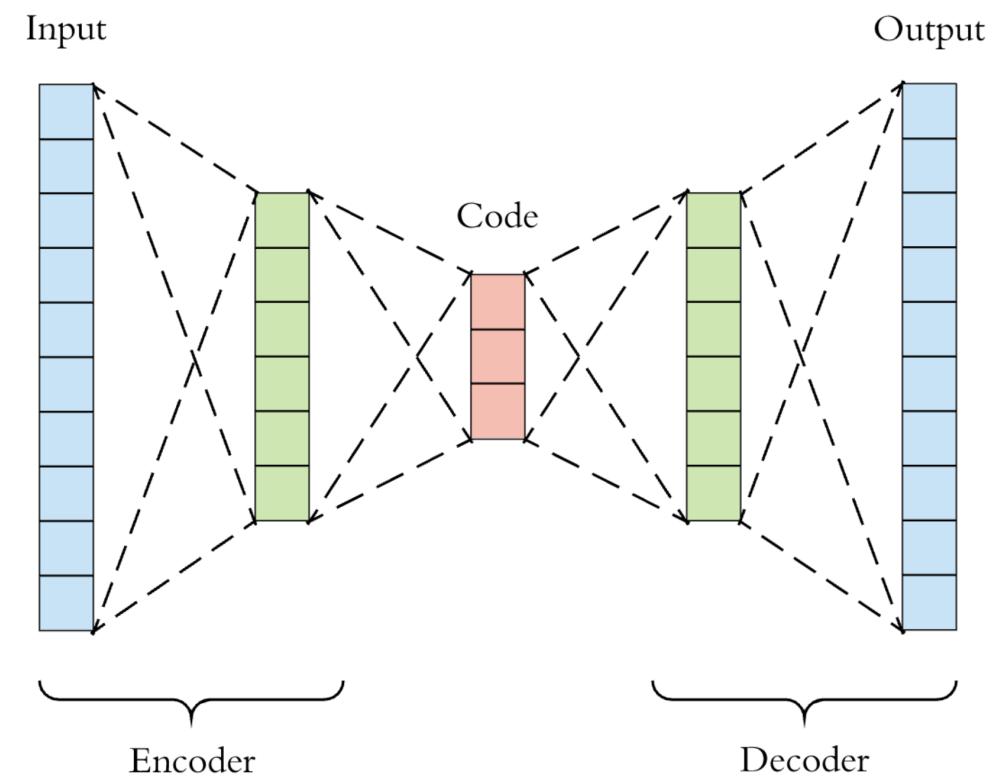
# Reti Convoluzionali

- Inception module e DSC (GoogleNet)
  - Il blocco  $5 \times 5$  viene fattorizzato in due  $3 \times 3$
  - I blocchi  $3 \times 3$  vengono fattorizzati in  $1 \times 3$  e  $3 \times 1$  (DSC in due dimensioni)
  - La rete ResNet (Residual Network) usa le skip connection su un blocco inception



# Autoencoder

- Un autoencoder è una coppia di reti CNN in cui sono presenti due parti (*encoder* e *decoder*) che hanno struttura simmetrica e che sono addestrate su coppie input/output *uguali*
- Lo strato più interno (strato di codifica) apprende una rappresentazione degli ingressi in uno *spazio latente a ridotta dimensionalità*
  - Attivazioni lineari → PCA
  - Si usano le ReLU



# Autoencoder

- Apprendimento

$$\phi : \mathcal{X} \rightarrow \mathcal{F}, \text{ encoder}$$

$$\psi : \mathcal{F} \rightarrow \mathcal{X}, \text{ decoder}$$

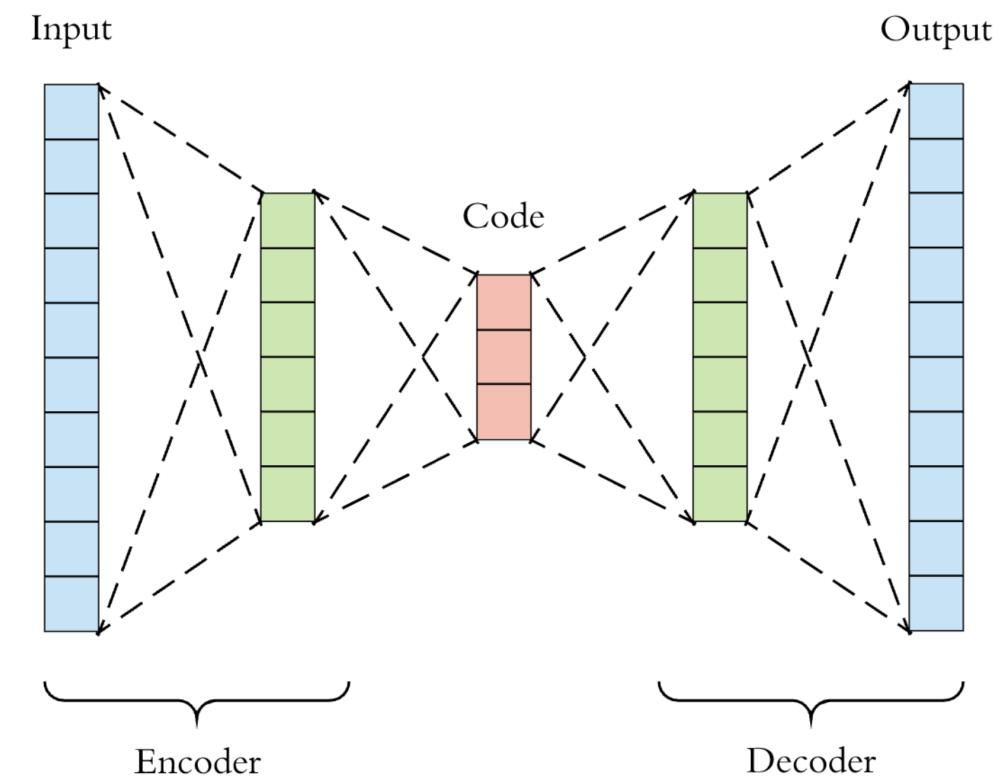
$$\phi, \psi = \arg \min_{\phi, \psi} \|X - (\psi \circ \phi)X\|^2$$

$$\mathbf{z} = \sigma(\mathbf{Wx} + \mathbf{b})$$

$$\mathbf{x}' = \sigma'(\mathbf{W}'\mathbf{z} + \mathbf{b}')$$

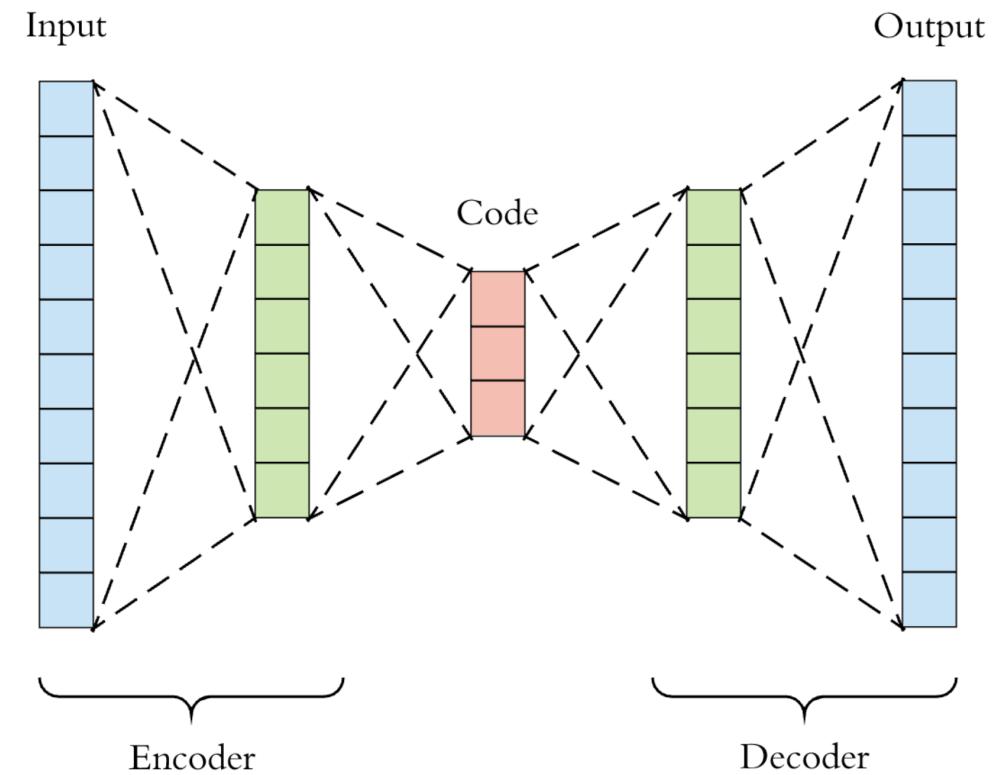
$$\mathcal{L}(\mathbf{x}, \mathbf{x}') = \|\mathbf{x} - \mathbf{x}'\|^2 = \|\mathbf{x} - \sigma'(\mathbf{W}'(\sigma(\mathbf{Wx} + \mathbf{b})) + \mathbf{b}')\|^2$$

Uso delle ReLU



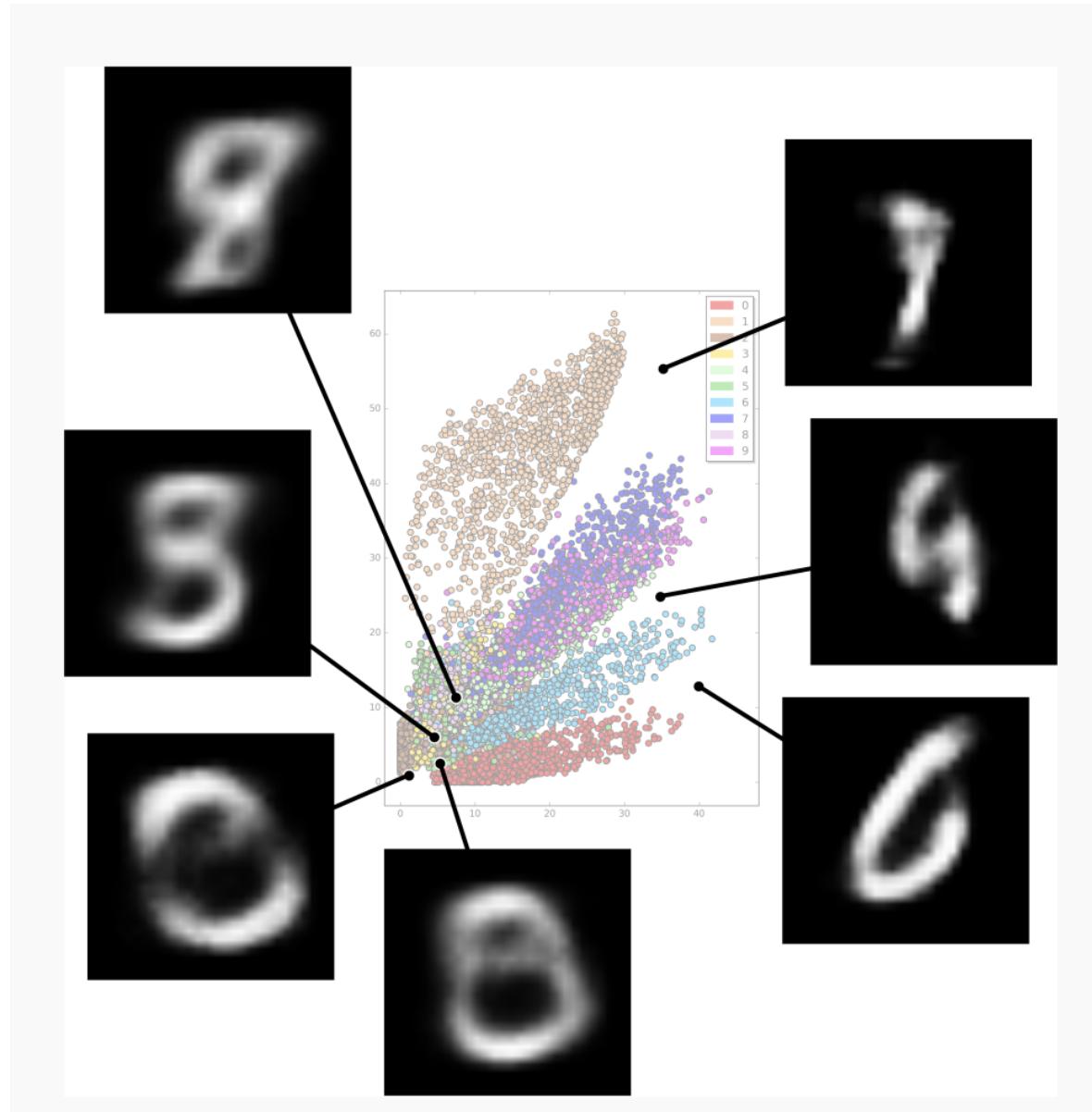
# Autoencoder

- Denoising autoencoder
  - Apprendono su input rumorosi, ma usano una loss riferita al dato con rumore
  - Regolarizzazione che diminuisce l'overfit
- Sparse autoencoder
  - Apprendono un vettore di codifica sparso con molte unità, ma che non si attivano tutte
  - Anche qui regolarizzazione
- Contractive autoencoder
  - Apprendimento con una regolarizzazione esplicita della loss



# Autoencoder

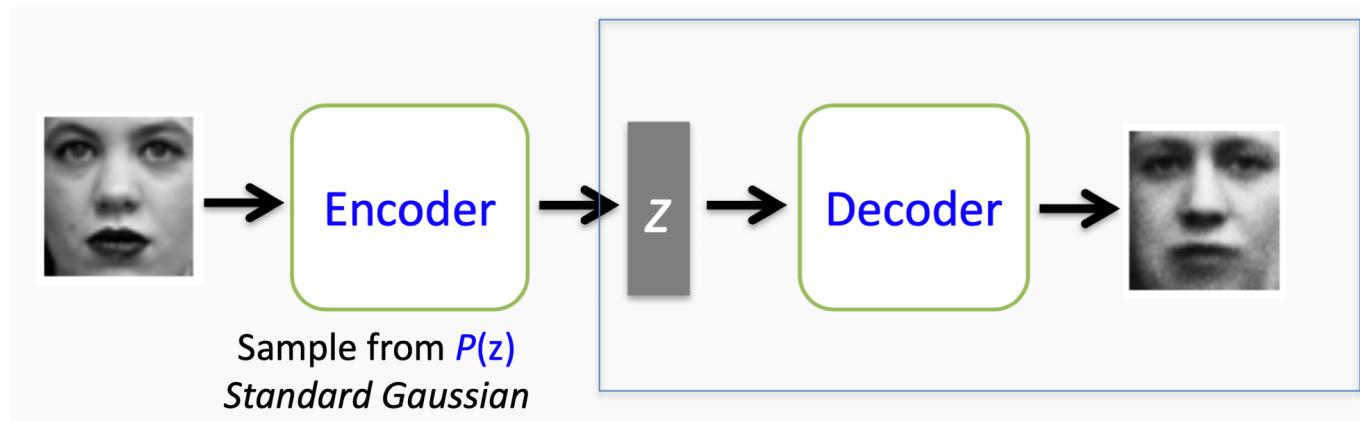
- Limiti
  - Spazio di ingresso discreto
  - Gap nello spazio latente: mancanza di rappresentazione
  - Necessità di una distribuzione continua di probabilità



# Autoencoder

- Variational autoencoder (VAE)

- Stima di una distribuzione di probabilità  $p(z|x)$  dalla quale generare  $x'$  non osservato a partire dalle variabili latenti  $z$
- Stiamo facendo una stima MAP di  $p(z|x)$

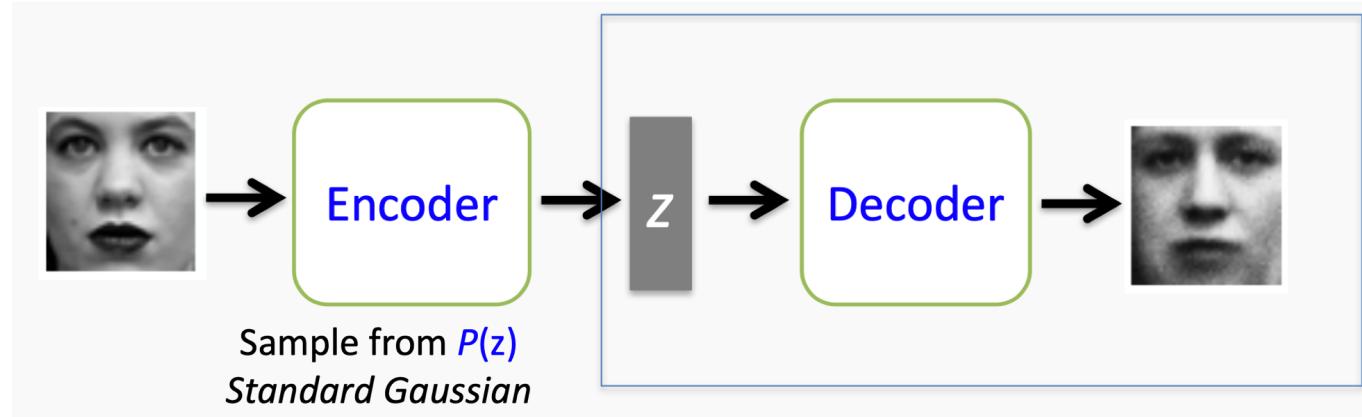


# Autoencoder

- Variational autoencoder (VAE)

- Useremo la minimizzazione della divergenza KL rispetto ad una *gaussiana* (per semplicità di calcolo)

$$q^*(\mathbf{z}) = \arg \min_{q \sim Q} \text{KL}(q(\mathbf{z}) \| p(\mathbf{z} \mid \mathbf{x}))$$



# Autoencoder

- Variational autoencoder (VAE)

- Useremo la minimizzazione della divergenza KL rispetto ad una *gaussiana* (per semplicità di calcolo)

$$\begin{aligned} \text{KL}(q(\mathbf{z}) \| p(\mathbf{z} \mid \mathbf{x})) &= \mathbb{E}_{\mathbf{z} \sim q} \log q(\mathbf{z}) - \mathbb{E}_{\mathbf{z} \sim q} \log p(\mathbf{z} \mid \mathbf{x}) \\ &= \underbrace{\mathbb{E}_{\mathbf{z} \sim q} \log q(\mathbf{z})}_{(a) \quad -1 * \text{ELBO}} - \underbrace{\mathbb{E}_{\mathbf{z} \sim q} \log p(\mathbf{z}, \mathbf{x})}_{(b)} + \underbrace{\log p(\mathbf{x})}_{(b)} \\ &= -\text{ELBO}(q) + \log p(\mathbf{x}) \end{aligned}$$

ELBO: Evidence Lower Bound

Massimizzare solo il termine ELBO implica minimizzare la KL

# Autoencoder

- Variational autoencoder

- Useremo la minimizzazione della divergenza KL rispetto ad una *gaussiana* (per semplicità di calcolo)

$$p(z \mid X) \approx q(z) = \prod_{i=1}^N q_i(z_i)$$

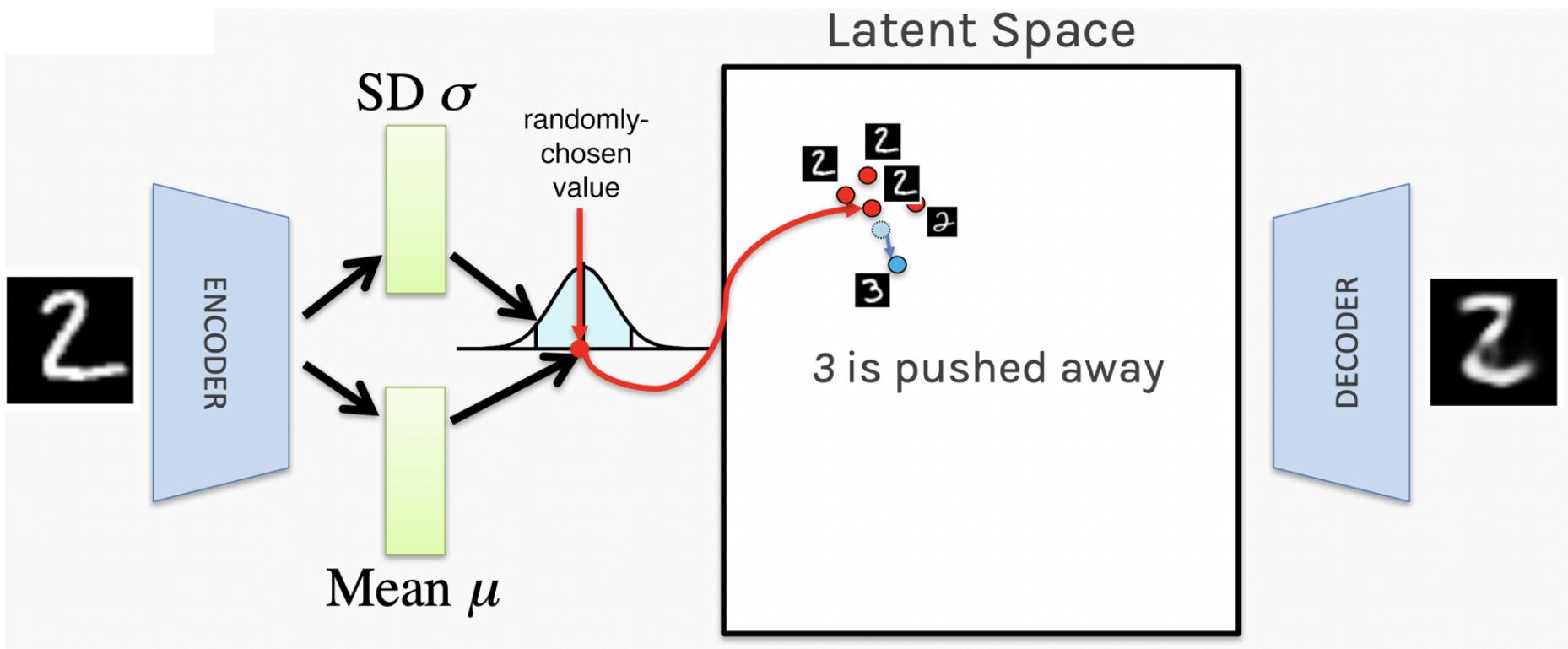
$$z = \mu + \sigma * \epsilon; \quad \epsilon \sim \mathcal{N}(0, 1)$$

Scelgo la mia fattorizzazione in base  
al numero di cluster che mi aspetto

Si sceglie di rappresentare la  
Gaussiana riparametrizzandola al  
fine di calcolare più efficientemente  
i gradienti

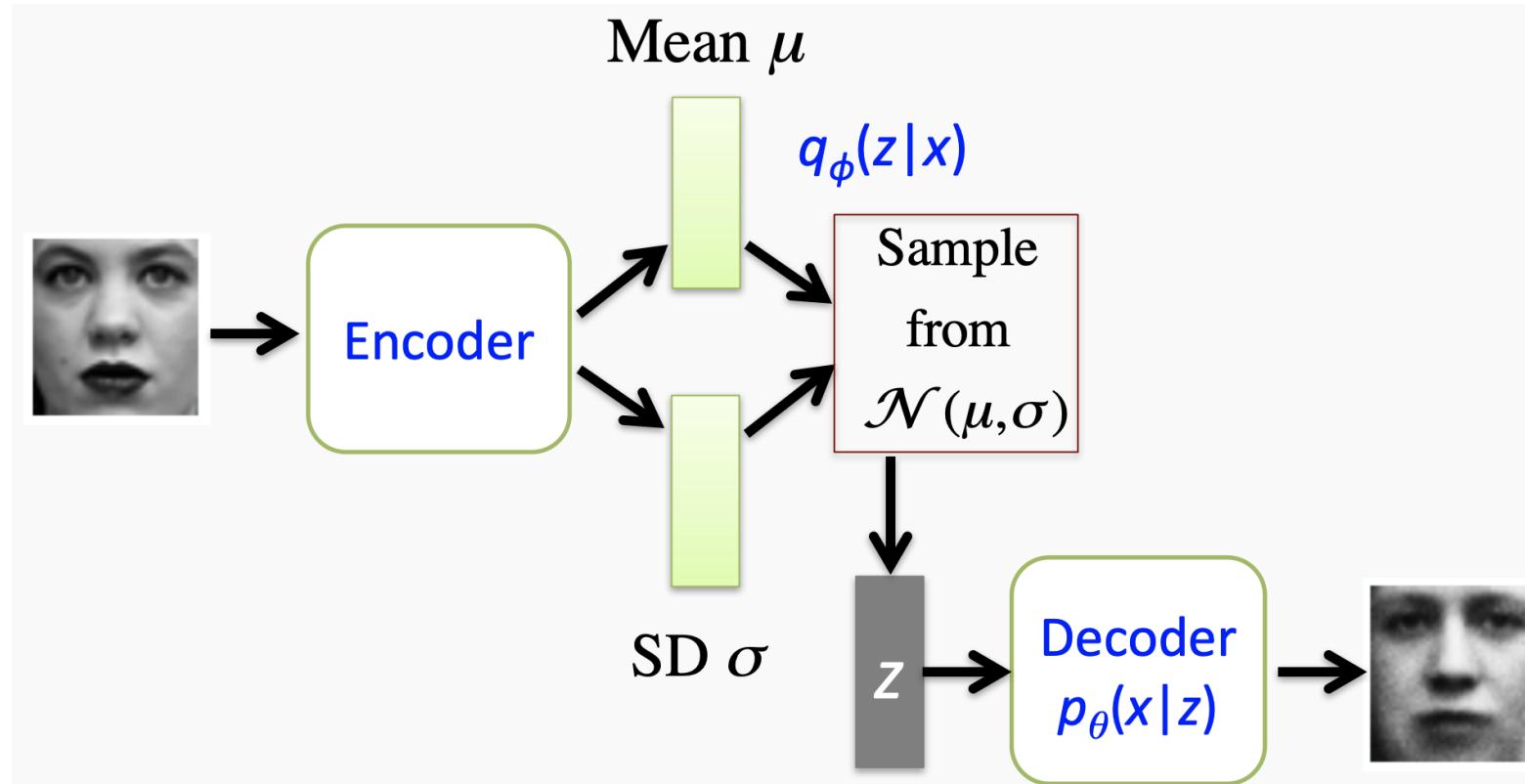
# Autoencoder

- Variational autoencoder



# Autoencoder

- Variational autoencoder

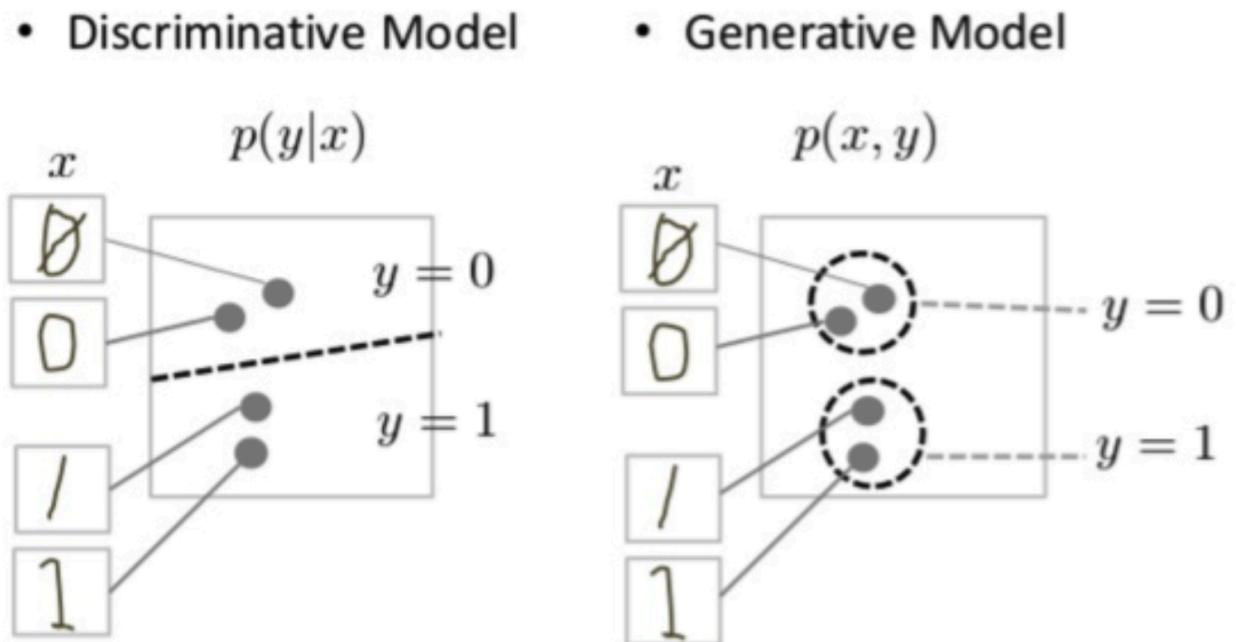


# Generative Adversarial Networks (GAN)

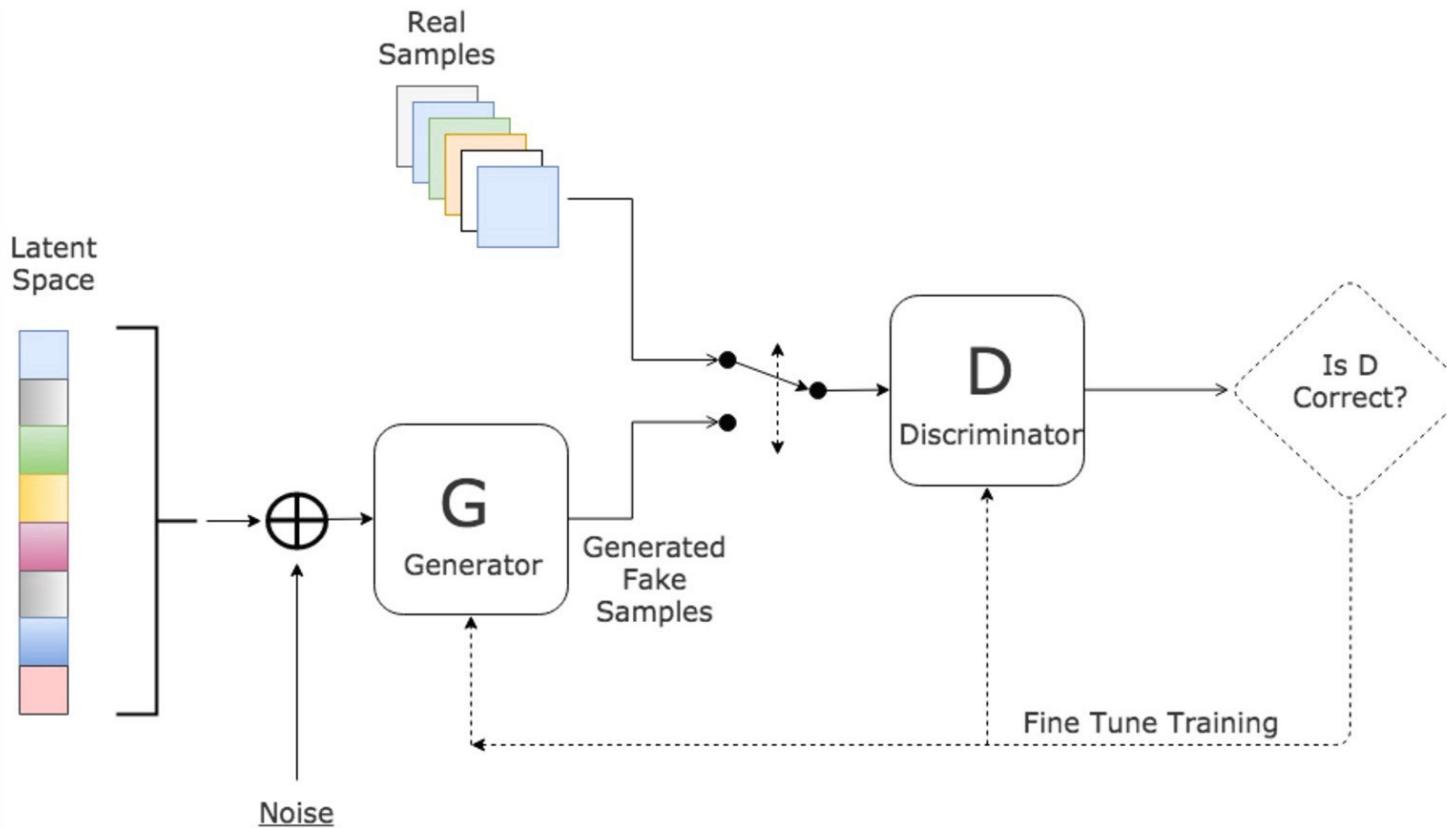
- I VAE sono stati usati spesso per tentare di generare campioni sintetici e realistici tratti dallo spazio di ingresso
  - Generazione di «Celebrity faces»
- Le GAN sono delle architetture che usano due reti, un *generatore* e un *discriminatore*, per competere l'una contro l'altra in al fine di generare campioni realistici

# Generative Adversarial Networks (GAN)

- Il generatore è non supervisionato e apprende la probabilità congiunta  $p(x,y)$  di ingressi e uscite
  - Viene addestrato a partire da rumore gaussiano a produrre esempi realistci
- Il discriminatore è supervisionato e apprende  $p(y|x)$ 
  - Viene addestrato a discriminare gli esempi reali e quelli sintetici prodotti dal generatore



# Generative Adversarial Networks (GAN)



Generatore:  $G(\mathbf{z}, \theta_g)$   
Discriminatore:  $D(\mathbf{x}, \theta_d)$

Apprendono con un approccio minmax

$$\min_G \max_D V(D, G) = \mathbb{E}_{\mathbf{x} \sim p_{\text{data}}(\mathbf{x})} [\log D(\mathbf{x})] + \mathbb{E}_{\mathbf{z} \sim p_{\mathbf{z}}(\mathbf{z})} [\log(1 - D(G(\mathbf{z})))].$$

# Generative Adversarial Networks (GAN)

---

**Algorithm 1** Minibatch stochastic gradient descent training of generative adversarial nets. The number of steps to apply to the discriminator,  $k$ , is a hyperparameter. We used  $k = 1$ , the least expensive option, in our experiments.

---

**for** number of training iterations **do**

**for**  $k$  steps **do**

- Sample minibatch of  $m$  noise samples  $\{\mathbf{z}^{(1)}, \dots, \mathbf{z}^{(m)}\}$  from noise prior  $p_g(\mathbf{z})$ .
- Sample minibatch of  $m$  examples  $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$  from data generating distribution  $p_{\text{data}}(\mathbf{x})$ .
- Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^m \left[ \log D(\mathbf{x}^{(i)}) + \log (1 - D(G(\mathbf{z}^{(i)}))) \right].$$

**end for**

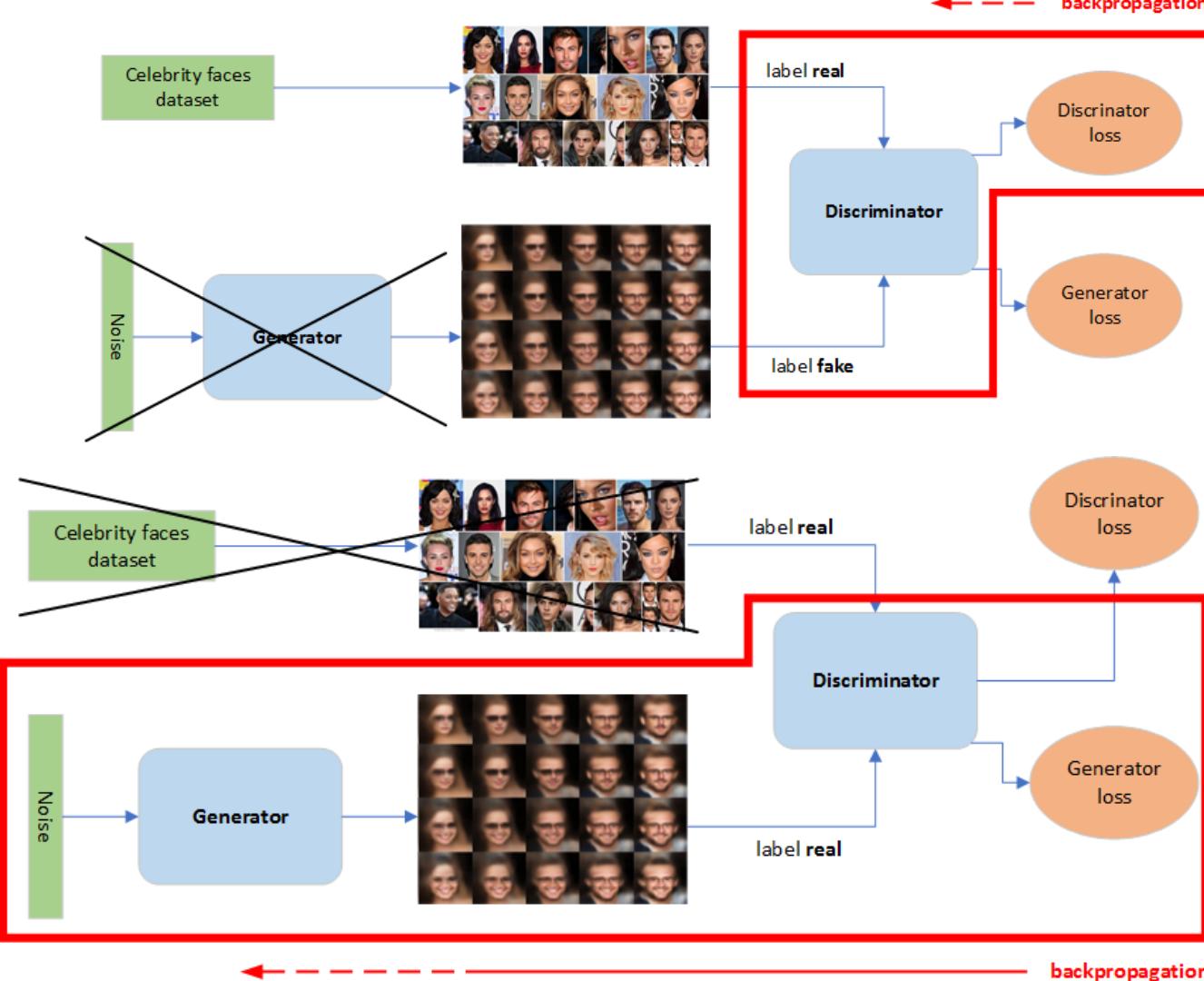
- Sample minibatch of  $m$  noise samples  $\{\mathbf{z}^{(1)}, \dots, \mathbf{z}^{(m)}\}$  from noise prior  $p_g(\mathbf{z})$ .
- Update the generator by descending its stochastic gradient:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^m \log (1 - D(G(\mathbf{z}^{(i)}))).$$

**end for**

The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.

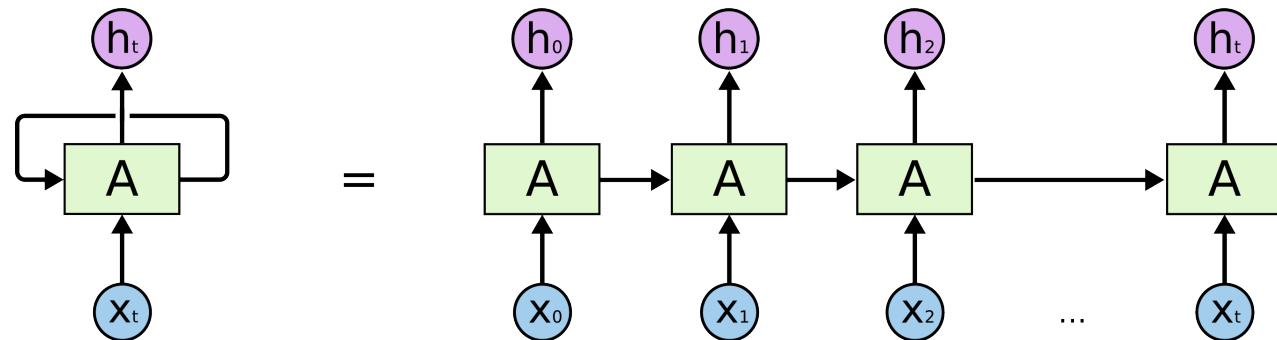
# Generative Adversarial Networks (GAN)



1. Campioniamo  $m$  campioni di rumore  $\mathbf{z}$  e dati reali  $\mathbf{x}$
2. Addestriamo  $D$  su  $\mathbf{x}$  e  $G(\mathbf{z})$  che non si addestra
3. Campioniamo di nuovo  $m$  campioni di rumore  $\mathbf{z}$
4. Addestriamo  $G$  sulla base della loss di  $D(G(\mathbf{z}))$

# Reti ricorrenti

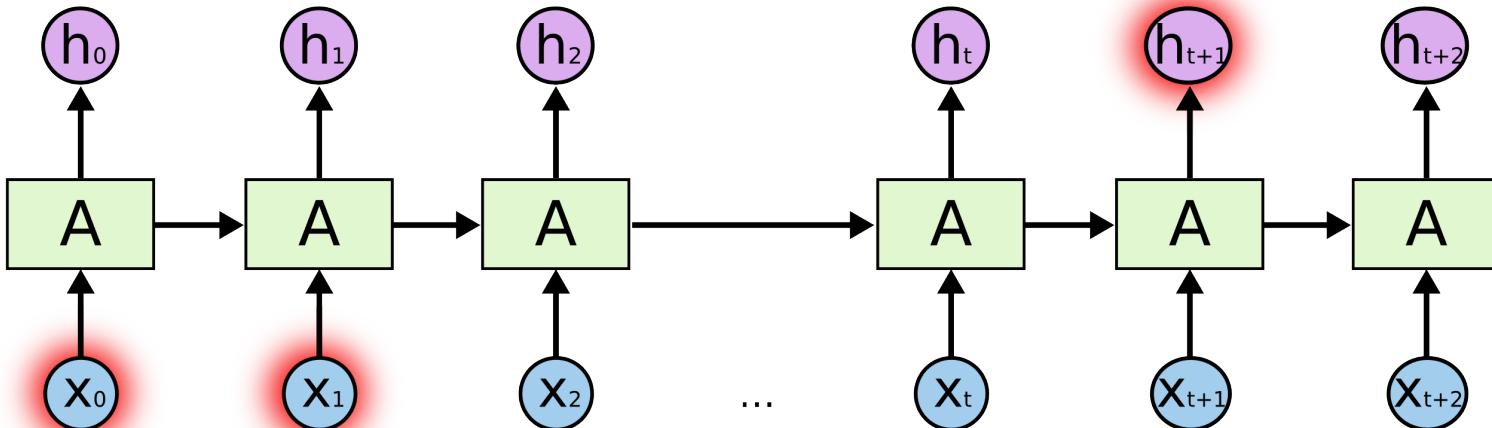
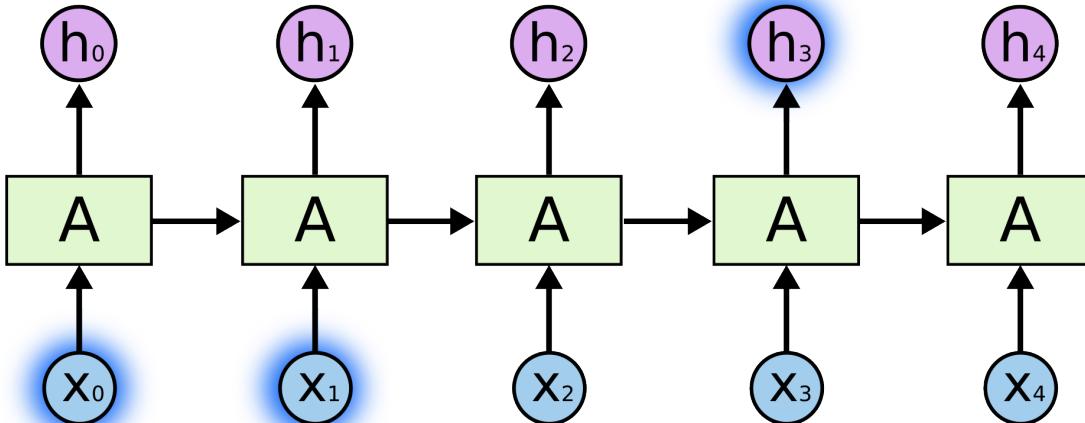
- Le Reti Neurali Ricorrenti (Recurrent Neural Networks – RNN) analizzano sequenze di ingressi riproponendo in ingresso le uscite calcolate al passo precedente



- Possono essere viste come catene di reti identiche che analizzano, di volta in volta,  $x_t$  e  $h_{t-1}$  per produrre  $h_t$

# Reti ricorrenti

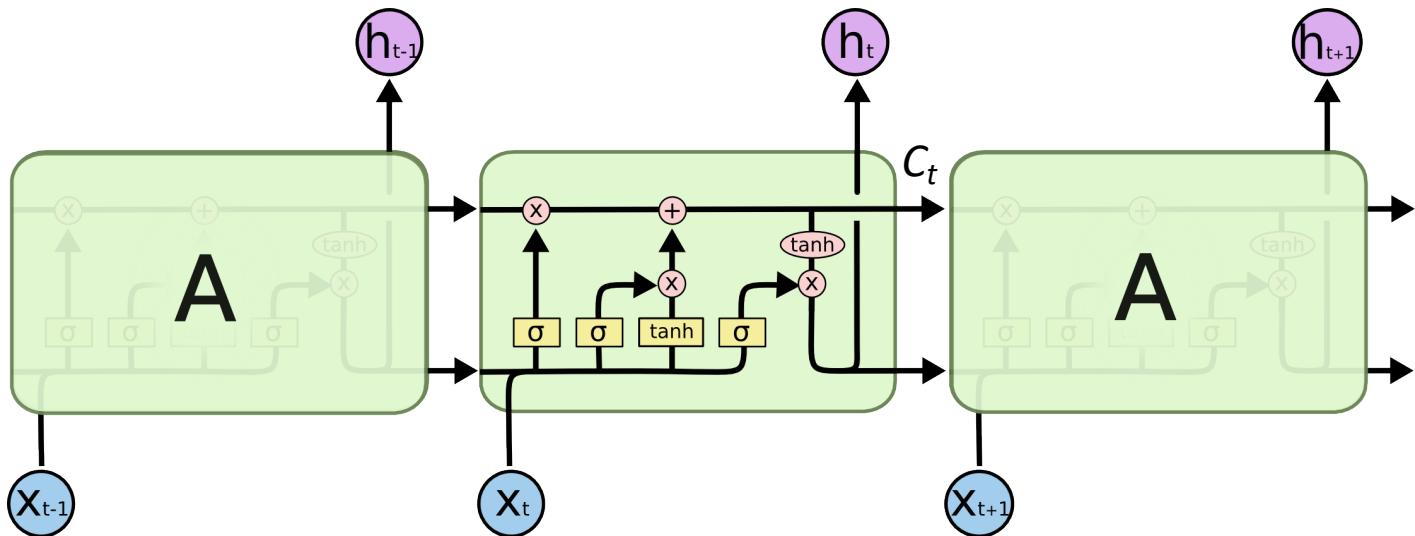
- Long Term Dependency
  - Le RNN non riescono ad apprendere dipendenze da ingressi molto lontani
- Problemi di vanishing gradients



# Reti ricorrenti

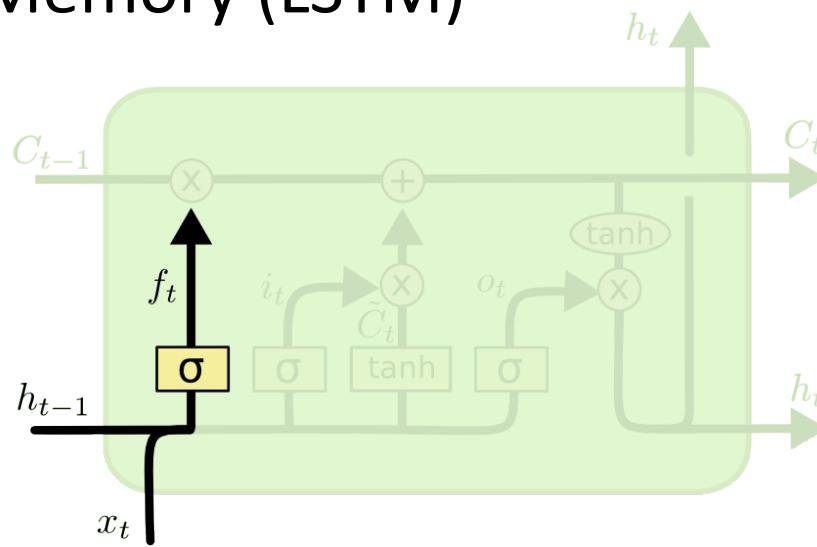
- Long-Short Term Memory (LSTM)

- Le unità apprendono dipendenze a lungo termine attraverso l'uso di uno *stato*  $C_t$  della cella che viene propagato verso le successive e tiene conto del fatto che la cella vuole ricordare o dimenticare le dipendenze pregresse



# Reti ricorrenti

- Long-Short Term Memory (LSTM)

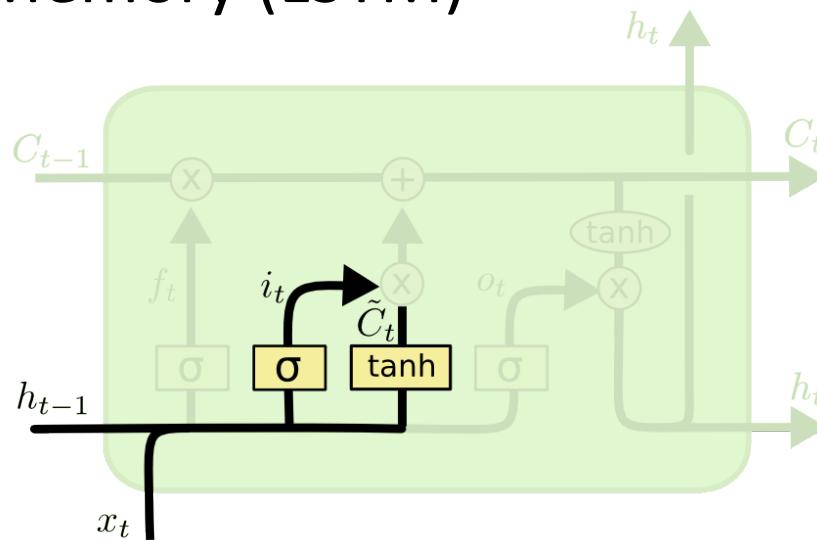


$$f_t = \sigma (W_f \cdot [h_{t-1}, x_t] + b_f)$$

- $f_t$  è il *forget gate*: tiene memoria di  $C_{t-1}$ 
  - Sarà questo termine a mettere l'accento su alcune parti della sequenza o a dimenticarle (ad es. selezionare l'attenzione sul verbo della frase)

# Reti ricorrenti

- Long-Short Term Memory (LSTM)

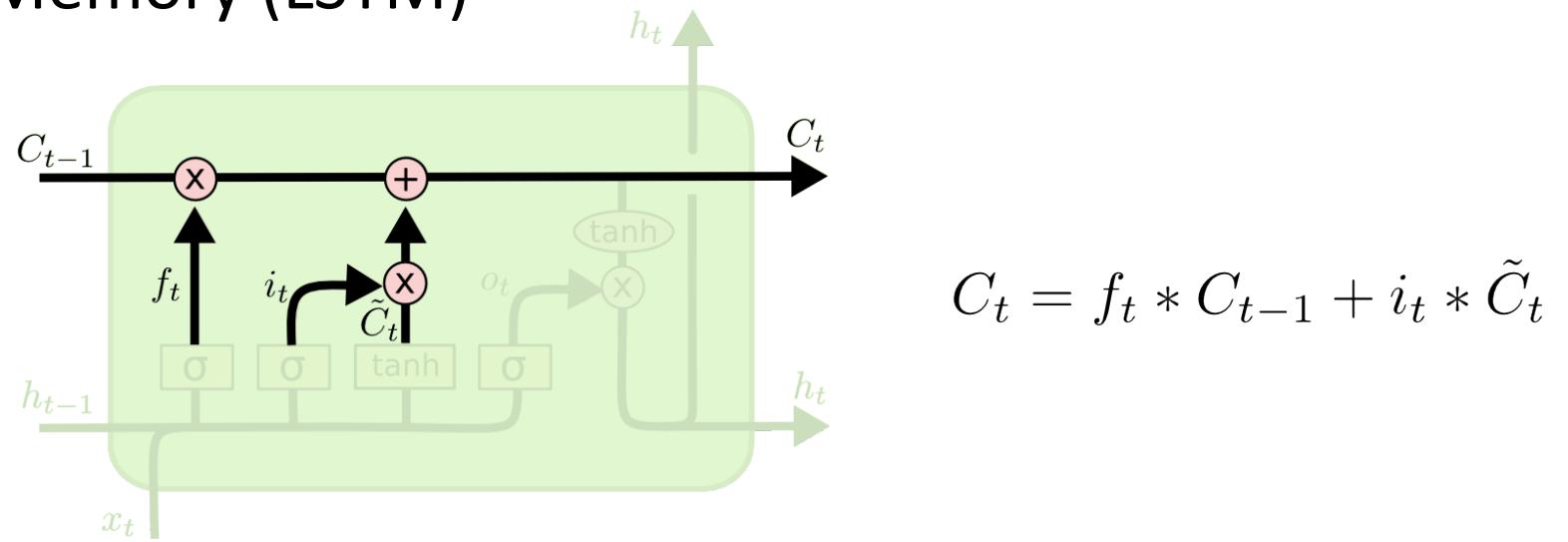


$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$

- $\hat{C}_t$  è lo stato candidato della cella
  - La tangente iperbolica garantisce valori in  $[-1,1]$  evitando che questa attivazione accumulata esploda lungo la catena di celle
- $i_t$  è l'*input gate*: va a modulare lo stato coandidato  $\hat{C}_t$  con l'ingresso alla cella

# Reti ricorrenti

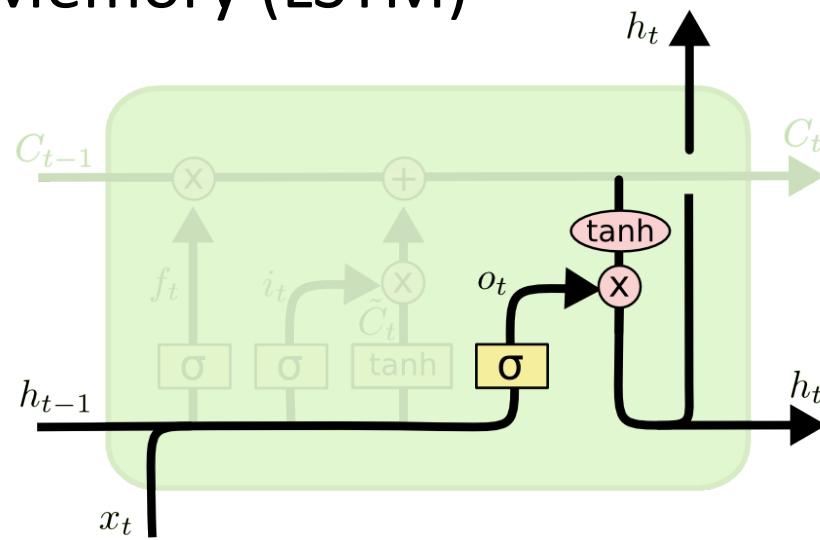
- Long-Short Term Memory (LSTM)



- $C_t$  è lo stato della cella ottenuto pesando lo stato corrente con il termine candidato generato dalla cella stessa

# Reti ricorrenti

- Long-Short Term Memory (LSTM)

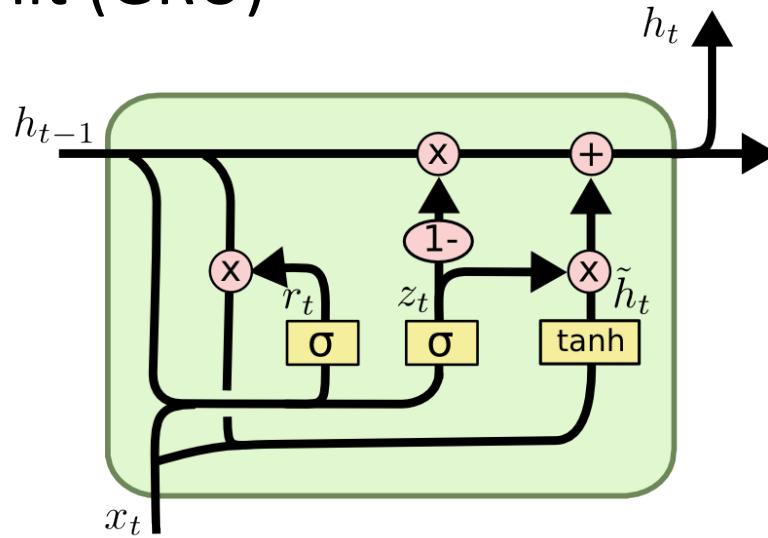


$$o_t = \sigma (W_o [ h_{t-1}, x_t ] + b_o)$$
$$h_t = o_t * \tanh (C_t)$$

- $o_t$  filtra lo stato aggiornato  $C_t$  per fornire l'uscita  $h_t$  (anche qui si usa tanh)

# Reti ricorrenti

- Gate Recurrent Unit (GRU)



- Unisce l'input e il forget gate in un unico update gate
  - $r_t$  va a pesare  $h_{t-1}$  nella generazione dell'uscita candidata  $\hat{h}_t$
  - $z_t$  modula l'uscita precedente con quella candidata
- Molto più efficiente della cella LSTM

$$z_t = \sigma (W_z \cdot [h_{t-1}, x_t])$$
$$r_t = \sigma (W_r \cdot [h_{t-1}, x_t])$$
$$\tilde{h}_t = \tanh (W \cdot [r_t * h_{t-1}, x_t])$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

# Reti ricorrenti

- Le uscite vanno sempre passate ad uno stadio di classificazione denso
- Spesso si utilizzano varianti *bidirezionali* delle LSTM o GRU
  - Nel Natural Language Processing consentono di catturare il *contesto sinistro e destro* di una parola che è una informazione più ricca al fine di classificarla in termini di task linguistici

# Transformer

- Nel campo della traduzione automatica si sono sviluppate reti dedicate a trasformare sequenze di simboli (frase in una lingua) in nuove sequenze di simboli (frase nella lingua target)
- Queste architetture si basano su coppie encoder-decoder che hanno struttura ricorrente
- Poiché è necessario, in questi casi focalizzarsi, su parti rilevanti della frase che siano punti chiave per la traduzione e comunque comprendere le *dipendenze* tra le parti del discorso è nato il meccanismo dell'*attenzione*

# Transformer

- Attenzione
  - *Es. «la cagnetta si rifiutava di attraversare la strada perché era impaurita»*
  - Chi era impaurita? La cagnetta o la strada? Non è ovvio per un algoritmo
  - «era impaurita» **dipende** da «la cagnetta» cioè è necessario ricordare la sotto-sequenza «la cagnetta» quando di processa «era impaurita»

# Transformer

- Attenzione
  - Implementazione tradizionale
  - Gli encoder RNN passano *tutti i loro stati nascosti* ai decoder RNN e non solamente quello generato dall'ultimo encoder

# Transformer

- Gli encoder e i decoder sono preceduti da un meccanismo di self-attention che serve ad analizzare l'intorno di ogni termine
- Si utilizza un meccanismo di mascheratura dei termini della frase per addestrare il Transformer a predire il termine mascherato

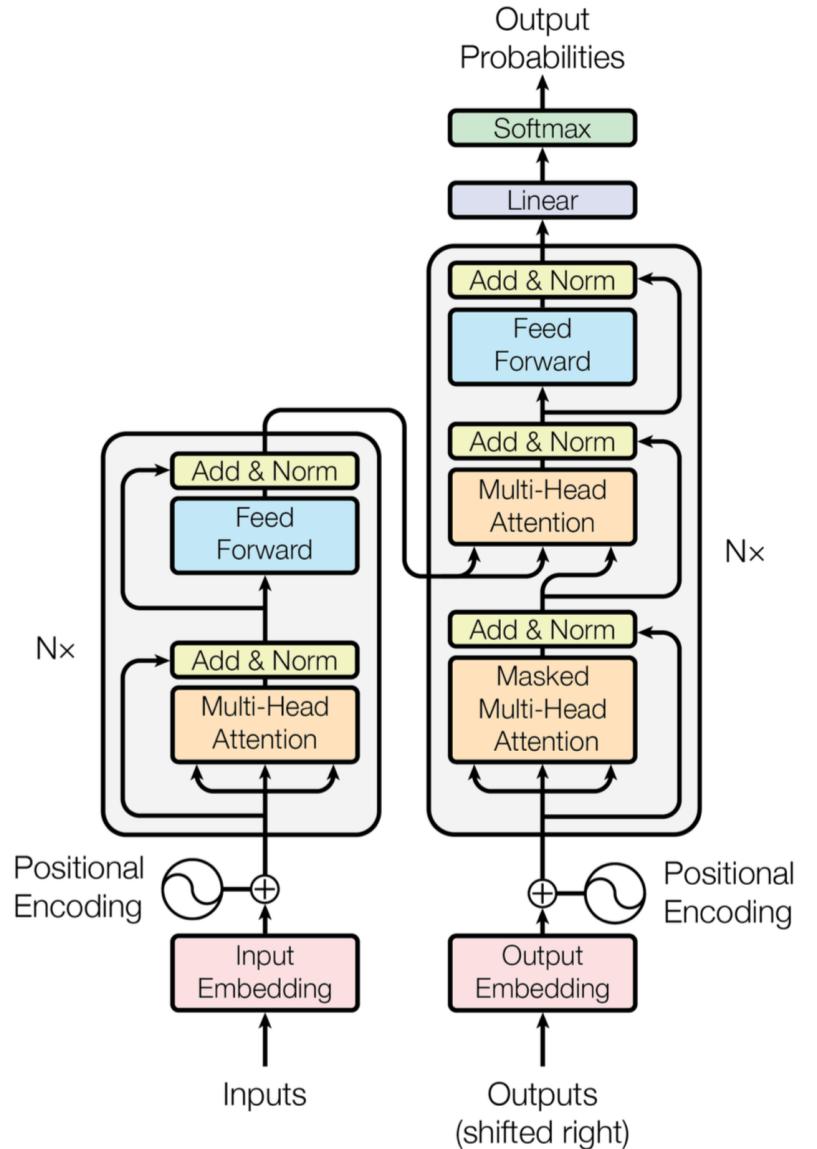


Figure 1: The Transformer - model architecture.

# Transformer

- $Q$  vettore delle query  
(rappresentazione delle singole parole)
- $K$  vettore delle chiavi  
(rappresentazione di tutte le parole della frase)
- $V$  vettore dei valori (di nuovo tutte le parole della frase)

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Fonte <https://medium.com/inside-machine-learning/what-is-a-transformer-d07dd1fbec04>

La multi-head attention è addestrato su diverse combinazioni in parallelo di  $Q$ ,  $K$  e  $V$

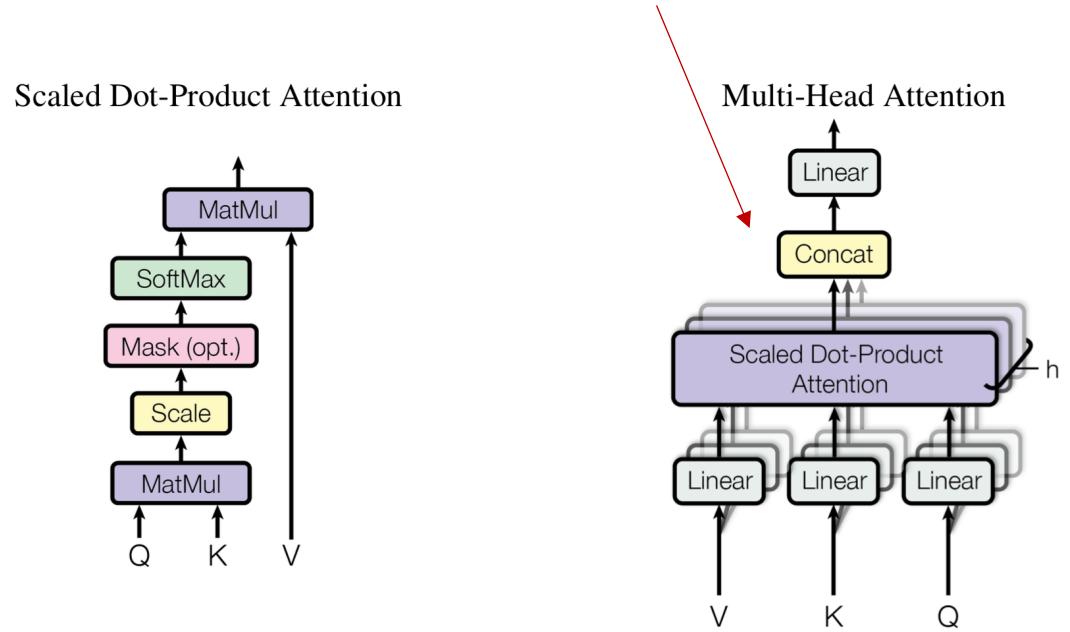


Figure 2: (left) Scaled Dot-Product Attention. (right) Multi-Head Attention consists of several attention layers running in parallel.

Termine di «attenzione» che modula una query con tutte le chiavi