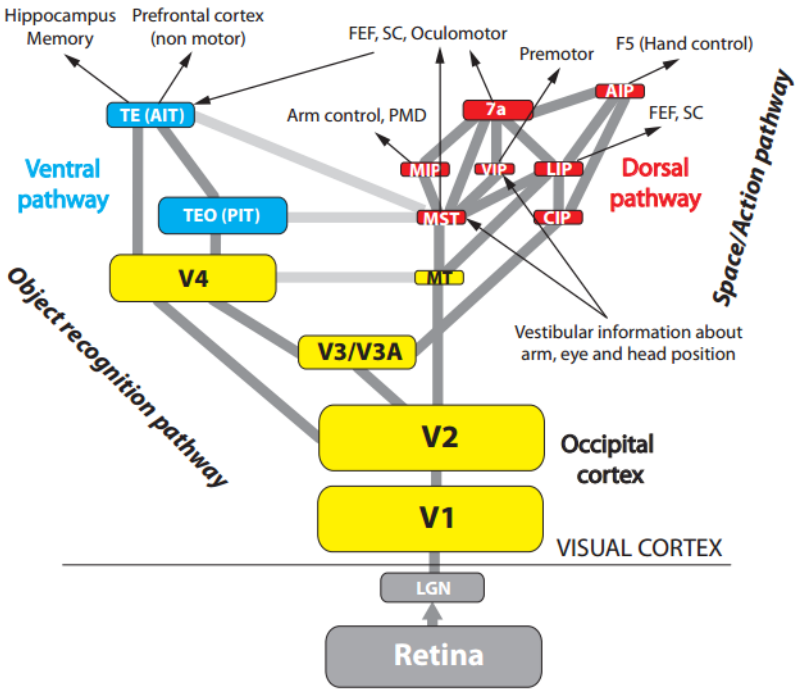
Deep Learning

- Introduzione
  - Perché deep?
  - Livelli e complessità
  - Tipologie di DNN
  - Ingredienti necessari
  - Da MLP a CNN
- Convolutional Neural Networks (CNN)
  - Architettura
  - Volumi e Convoluzione 3D
  - Relu
  - Pooling
  - Esempi di reti
  - In pratica
  - Training e Transfer Learning
- Deep Reinforcement Learning
  - Q-Learning
  - Deep Q-Learning

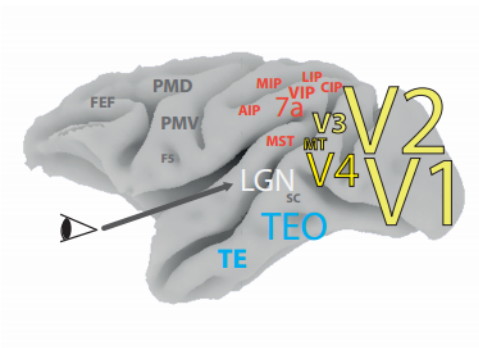
Perchè deep?

Con il termine DNN (Deep Neural Network) si denotano reti «profonde» composte da molti livelli (almeno 2 hidden) organizzati gerarchicamente.

- Le implicazioni di universal approximation theorem e la difficoltà di addestrare reti con molti livelli hanno portato per lungo tempo a focalizzarsi su reti con un solo livello hidden.
  - L'esistenza di soluzioni non implica efficienza: esistono funzioni computabili con complessità polinomiale operando su k livelli, che richiedono una complessità esponenziale se si opera su $k-1$ livelli (Hastad, 1986).
- L'organizzazione gerarchica consente di condividere e riusare informazioni (un po' come la programmazione strutturata). Lungo la gerarchia è possibile selezionare feature specifiche e scartare dettagli inutili (al fine di massimizzare l'invarianza).
- Il nostro sistema visivo opera su una gerarchia di livelli (deep):

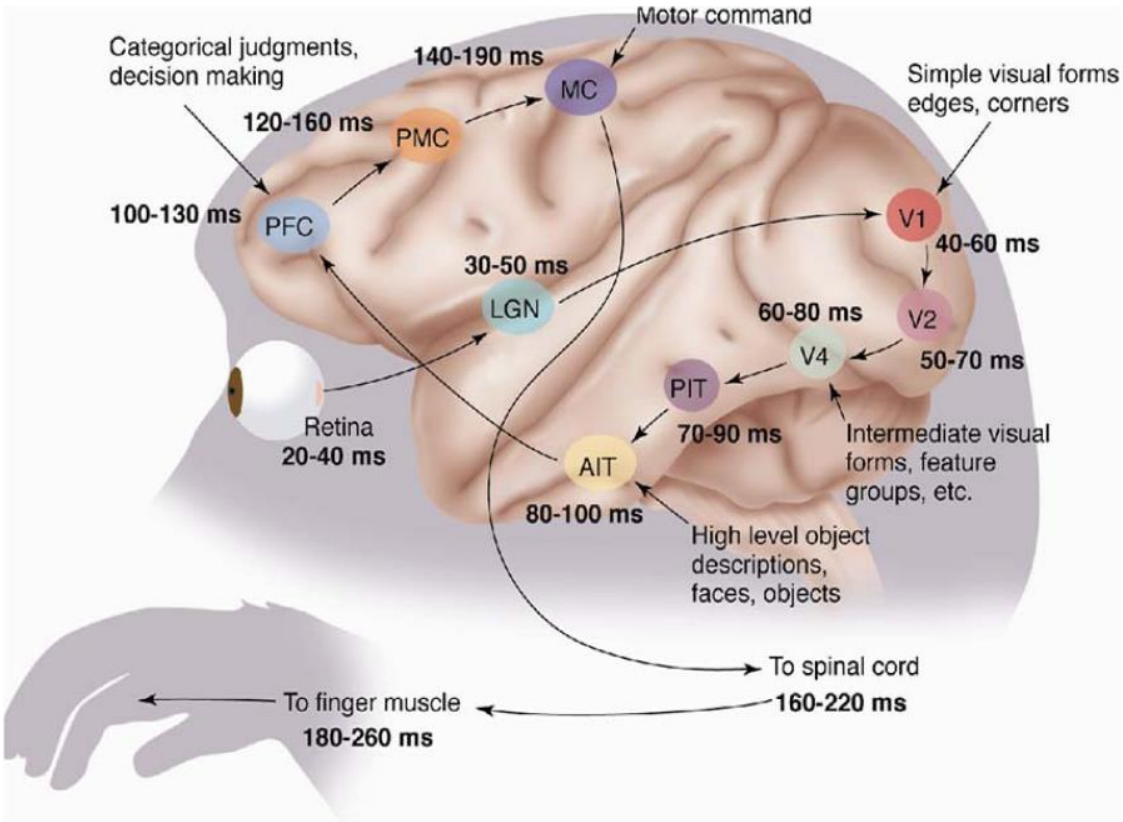


[Kruger et al. 2013]



ma quanto deep?

- Le DNN oggi maggiormente utilizzate consistono di un numero di livelli compreso tra 7 e 50.
  - Reti più profonde (100 livelli e oltre) hanno dimostrato di poter garantire prestazioni leggermente migliori, a discapito però dell'efficienza.
  - «solo» una decina di livelli tra la retina e i muscoli attuatori (altrimenti saremmo troppo lenti a reagire agli stimoli).

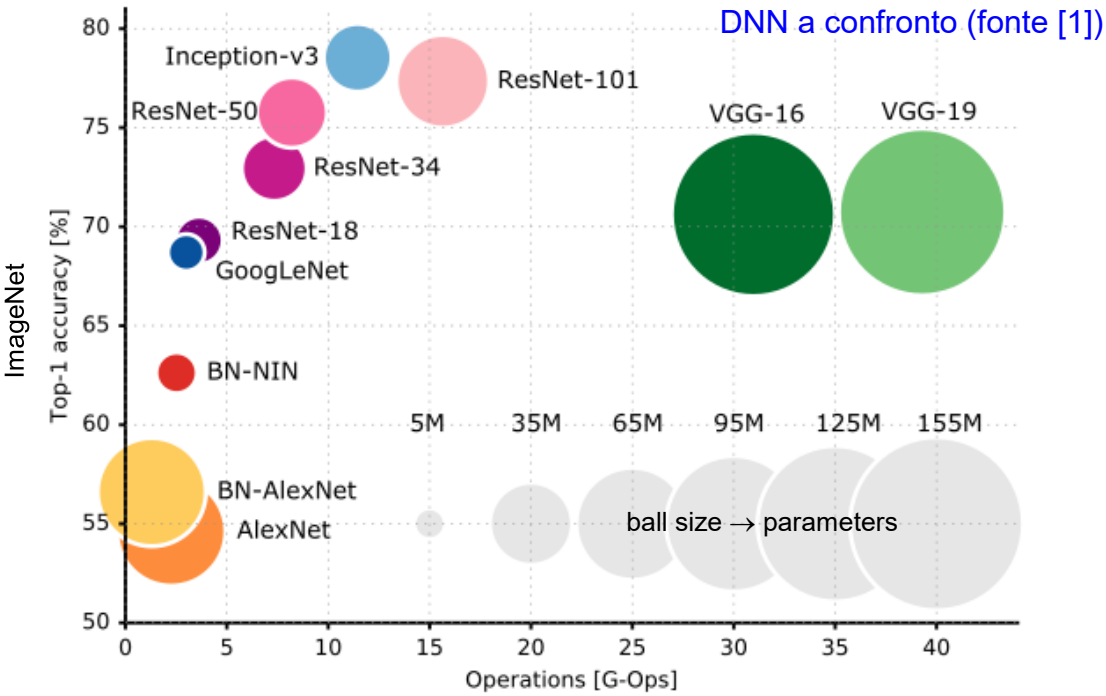


[Simon Thorpe 1996]

Livelli e Complessità

- La profondità (numero di livelli) è solo uno dei fattori di complessità. Numero di neuroni, di connessioni e di pesi caratterizzano altresì la complessità di una DNN.
- Maggiore è il numero di pesi (ovvero di parametri da apprendere) maggiore è la complessità del training. Al tempo stesso un elevato numero di neuroni (e connessioni) rende forward e back propagation più costosi, poiché aumenta il numero (G-Ops) di operazioni necessarie.
  - AlexNet: 8 livelli, 650K neuroni e 60M parametri
  - VGG-16: 16 livelli, 15M neuroni e 140M parametri
  - Corteccia umana: 21×10^9 neuroni e 1.5×10^{14} sinapsi

DNN a confronto (fonte [1])



(forward propagation)

[1] Canziani et al. 2016, *An Analysis of Deep Neural Network Models for Practical Applications*

Principali tipologie di DNN

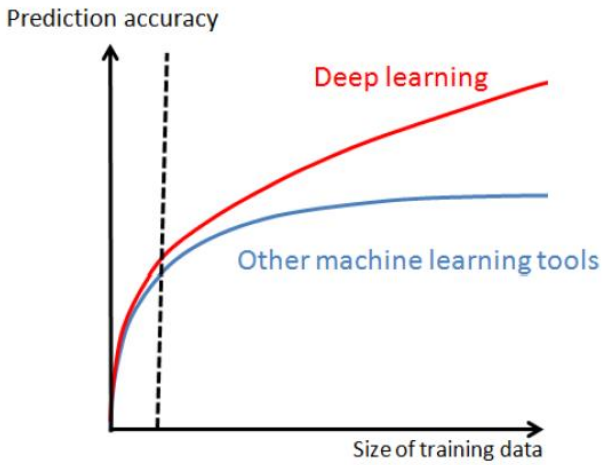
- Modelli feedforward «discriminativi» per la classificazione (o regressione) con training prevalentemente supervisionato:
  - CNN - Convolutional Neural Network (o ConvNet)
  - FC DNN - Fully Connected DNN (MLP con almeno due livelli hidden)
  - HTM - Hierarchical Temporal Memory
- Training non superivisionato (modelli «generativi» addestrati a ricostruire l'input, utili per pre-training di altri modelli e per produrre feature salienti):
  - Stacked (de-noising) Auto-Encoders
  - RBM - Restricted Boltzmann Machine
  - DBN - Deep Belief Networks
- Modelli ricorrenti (utilizzati per sequenze, speech recognition, sentiment analysis, natural language processing,…):
  - RNN - Recurrent Neural Network
  - LSTM - Long Short-Term Memory
- Reinforcement learning (per apprendere comportamenti):
  - Deep Q-Learning

Ingredienti necessari

CNN ottengono già nel 1998 buone prestazioni in problemi di piccole dimensioni (es. riconoscimento caratteri, riconoscimento oggetti a bassa risoluzione), ma bisogna attendere il 2012 (AlexNet) per un radicale cambio di passo. AlexNet non introduce rilevanti innovazioni rispetto alle CNN di LeCun del 1998, ma alcune condizione al contorno sono nel frattempo cambiate:

- BigData: disponibilità di dataset etichettati di grandi dimensioni (es. ImageNet: milioni di immagini, decine di migliaia di classi).

La superiorità delle tecniche di deep learning rispetto ad altri approcci si manifesta quando sono disponibili grandi quantità di dati di training.

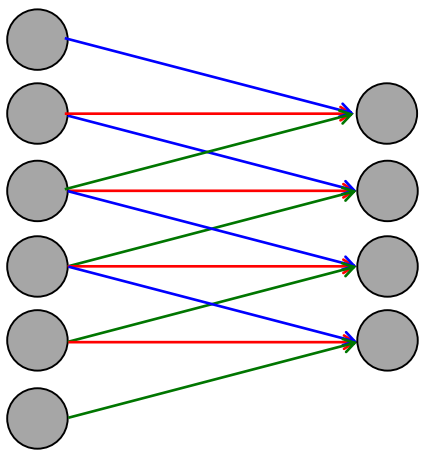


- GPU computing: il training di modelli complessi (profondi e con molti pesi e connessioni) richiede elevate potenze computazionali. La disponibilità di GPU con migliaia di core e GB di memoria interna ha consentito di ridurre drasticamente i tempi di training: da mesi a giorni.

- Vanishing (or exploding) gradient: la retro propagazione del gradiente (fondamentale per backpropagation) è problematica su reti profonde se si utilizza la sigmoide come funzione di attivazione. Il problema può essere gestito con attivazione Relu (descritta in seguito).

Da MLP a CNN

- Hubel & Wiesel (1962) scoprono la presenza, nella corteccia visiva del gatto, di due tipologie di neuroni:
  - Simple cells: agiscono come feature detector locali (fornendo selettività)
  - Complex cells: fondono (pooling) gli output di simple cell in un intorno (garantendo invarianza).
- Neocognitron (Fukushima, 1980) è una delle prime reti neurali che cerca di modellare questo comportamento.
- Convolutional Neural Networks (CNN) introdotte da LeCun et al., a partire dal 1998. Le principali differenze rispetto a MLP:
  - processing locale: i neuroni sono connessi solo localmente ai neuroni del livello precedente. Ogni neurone esegue quindi un'elaborazione locale. Forte riduzione numero di connessioni.
  - pesi condivisi: i pesi sono condivisi a gruppi. Neuroni diversi dello stesso livello eseguono lo stesso tipo di elaborazione su porzioni diverse dell'input. Forte riduzione numero di pesi.



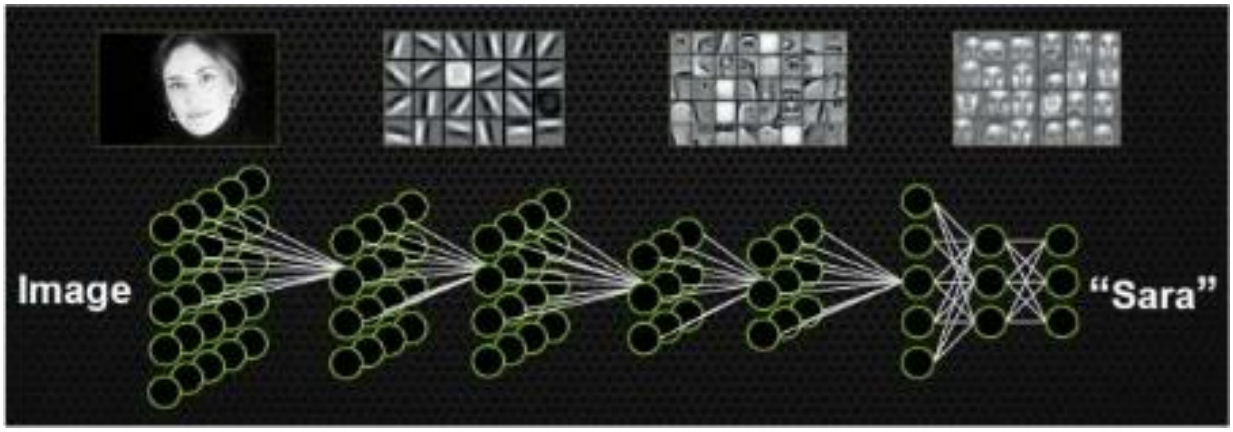
Esempio: ciascuno dei 4 neuroni a destra è connesso solo a 3 neuroni del livello precedente. I pesi sono condivisi (stesso colore stesso peso). In totale 12 connessioni e 3 pesi contro le 24 connessioni + 24 pesi di una equivalente porzione di MLP.

  - alternanza livelli di feature extraction e pooling.

CNN: Architettura

Esplicitamente progettate per processare immagini, per le quali elaborazione locale, pesi condivisi, e pooling non solo semplificano il modello, ma lo rendono più efficace rispetto a modelli fully connected. Possono essere utilizzate anche per altri tipi di pattern (es. speech).

- Architettura: una CNN è composta da una gerarchia di livelli. Il livello di input è direttamente collegato ai pixel dell'immagine, gli ultimi livelli sono generalmente fully-connected e operano come un classificatore MLP, mentre nei livelli intermedi si utilizzano connessioni locali e pesi condivisi.

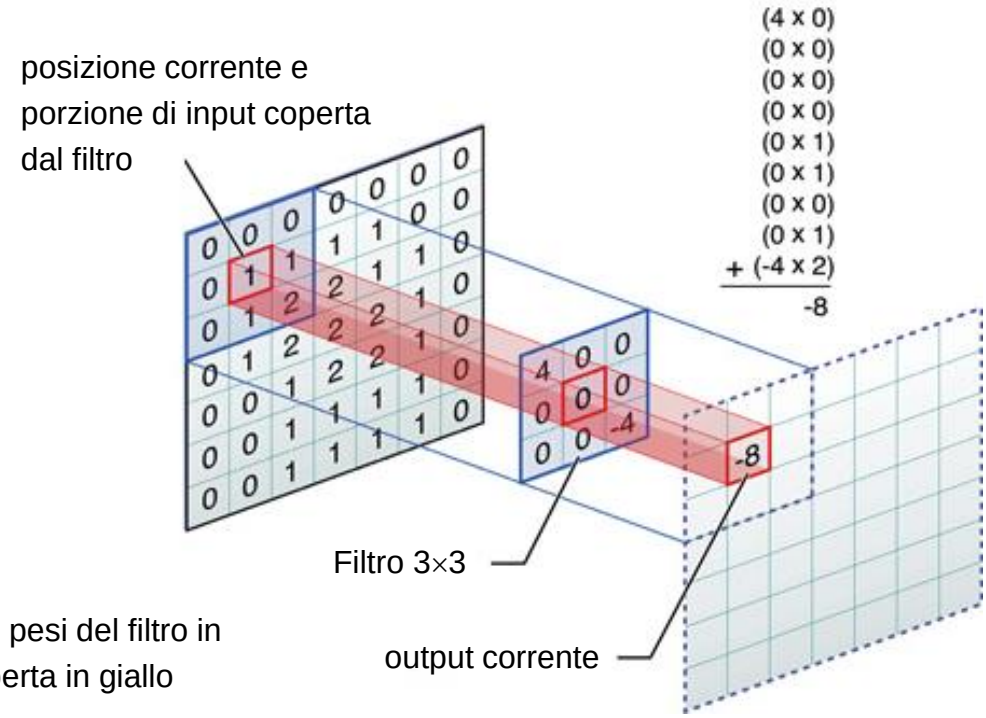


- Il campo visivo (receptive field) dei neuroni aumenta muovendosi verso l'alto nella gerarchia.

- Le connessioni locali e condivise fanno sì che i neuroni processino nello stesso modo porzioni diverse dell'immagine. Si tratta di un comportamento desiderato, in quando regioni diverse del campo visivo contengono lo stesso tipo di informazioni (bordi, spigoli, porzioni di oggetti, ecc.).

Convoluzione

- La convoluzione è una delle più importanti operazioni di image processing attraverso la quale si applicano filtri digitali.
- Un filtro digitale (un piccola maschera 2D di pesi) è fatta scorrere sulle diverse posizioni di input; per ogni posizione viene generato un valore di output, eseguendo il prodotto scalare tra la maschera e la porzione dell'input coperta (entrambi trattati come vettori).



Esempio (animato): pesi del filtro in rosso, porzione coperta in giallo

1 ×1	1 ×0	1 ×1	0	0
0 ×0	1 ×1	1 ×0	1	0
0 ×1	0 ×0	1 ×1	1	1
0	0	1	1	0
0	1	1	0	0

4		

Esempi applicazione filtri a Immagini

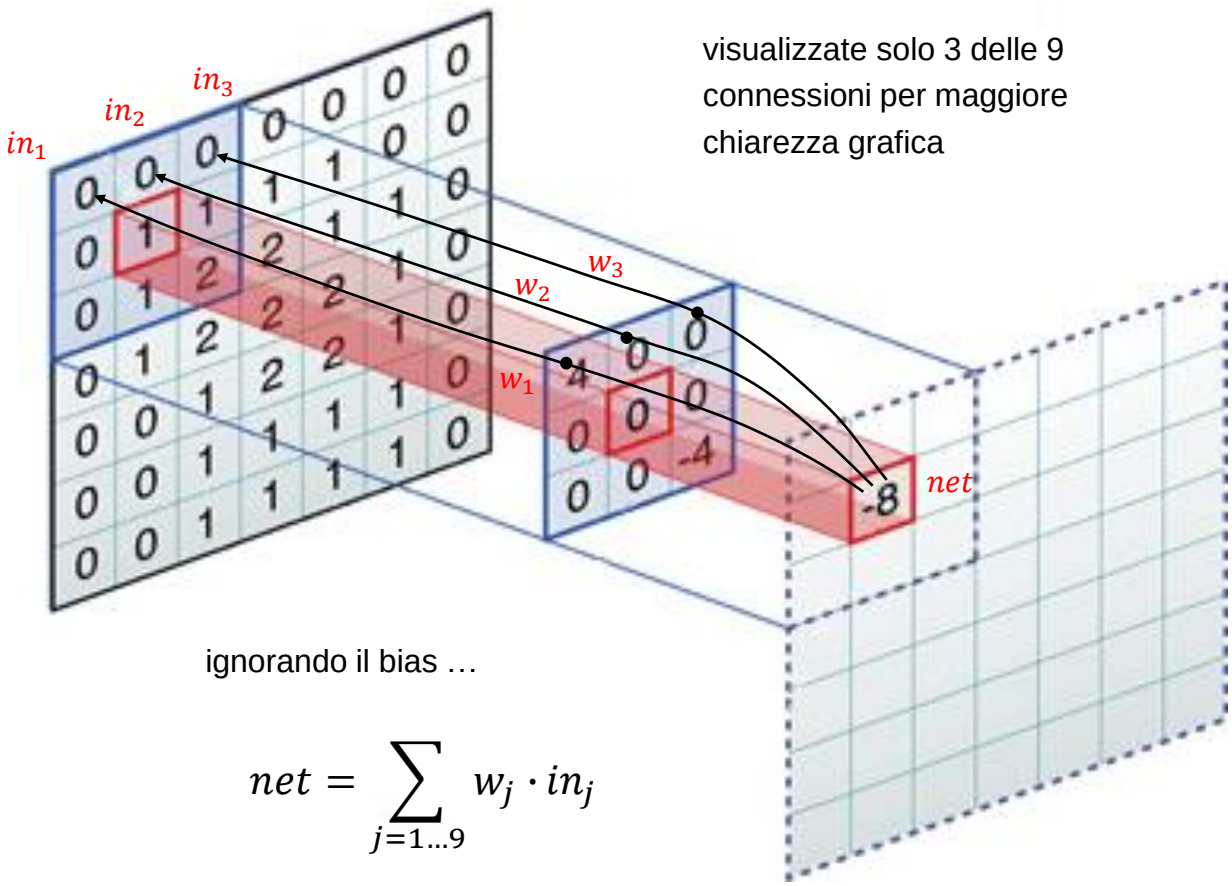
Immagine input | Filtro | Immagine output

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

$$\begin{bmatrix} 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \end{bmatrix}$$

Neuroni come convolutori

- Consideriamo i pixel come neuroni e le due immagini di input e di output come livelli successivi di una rete. Dato un filtro 3×3, se colleghiamo un neurone ai 9 neuroni che esso «copre» nel livello precedente, e utilizziamo i pesi del filtro come pesi delle connessioni w, notiamo che un classico neurone (di una MLP) esegue di fatto una convoluzione.



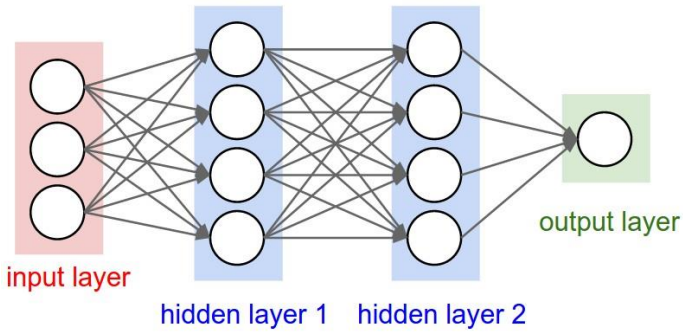
visualizzate solo 3 delle 9 connessioni per maggiore chiarezza grafica

ignorando il bias …

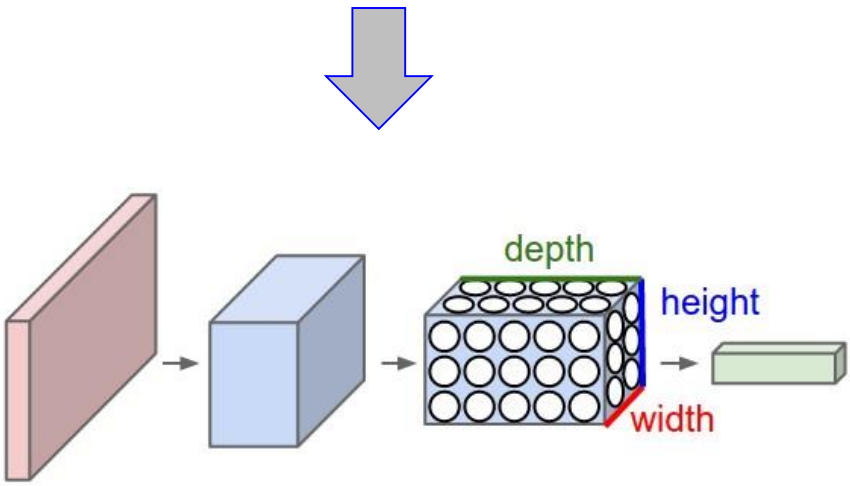
$$net = \sum_{j=1\ldots9} w_j \cdot in_j$$

CNN: Volumi

- Volumi: I neuroni di ciascun livello sono organizzati in griglie o volumi 3D (si tratta in realtà di una notazione grafica utile per la comprensione delle connessioni locali).



MLP: organizzazione lineare dei neuroni nei livelli

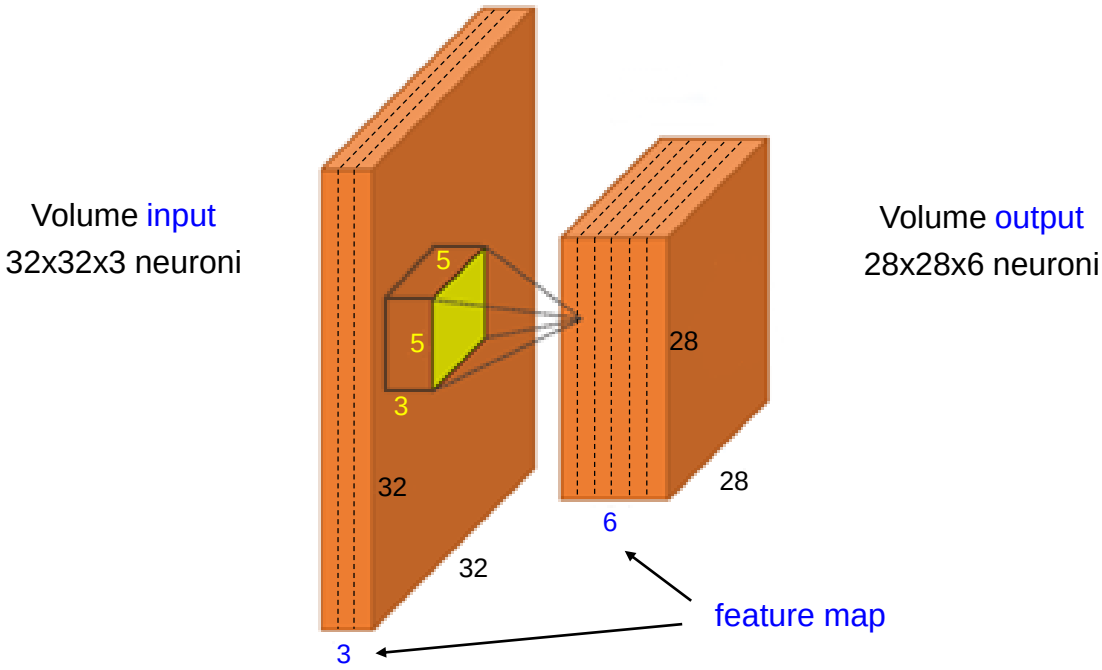


CNN: i livelli sono organizzati come griglie 3D di neuroni

- sui piani width - height si conserva l'organizzazione spaziale «retinotipica» dell'immagine di input.
- la terza dimensione depth (come vedremo) individua le diverse feature map.

CNN: Convoluzione 3D

- Il filtro opera su una porzione del volume di input. Nell'esempio ogni neurone del volume di output è connesso a 5x5x3=75 neuroni del livello precedente.



- Ciascuna «fetta» di neuroni (stessa depth) denota una feature map. Nell'esempio troviamo:
  - 3 feature map (dimensione 32x32) nel volume di input.
  - 6 feature map (dimensione 28x28) nel volume di output.

- I pesi sono condivisi a livello di feature map. I neuroni di una stessa feature map processano porzioni diverse del volume di input nello stesso modo. Ogni feature map può essere vista come il risultato di uno specifico filtraggio dell'input (filtro fisso).

- Nell'esempio il numero di connessioni tra i due livelli è (28x28x6) x (5x5x3) = 352800, ma il numero totale di pesi è 6 x (5x5x3 + 1) = 456. *In analoga porzione di MLP quanti pesi?*

bias

CNN: Convoluzione 3D (2)

- Quando un filtro 3D viene fatto scorrere sul volume di input, invece di spostarsi con passi unitari (di 1 neurone) si può utilizzare un passo (o $Stride$) maggiore. Questa operazione riduce la dimensione delle feature map nel volume di output e conseguentemente il numero di connessioni.
  - Sui livelli iniziali della rete per piccoli stride (es. 2, 4), è possibile ottenere un elevato guadagno in efficienza a discapito di una leggera penalizzazione in accuratezza.
- Ulteriore possibilità (per regolare la dimensione delle feature map) è quella di aggiungere un bordo (valori zero) al volume di input. Con il parametro $Padding$ si denota lo spessore (in pixel) del bordo.
- Sia W_{out} la dimensione (orizzontale) della feature map di output e W_{in} la corrispondente dimensione nell'input. Sia inoltre F la dimensione (orizzontale del filtro). Vale la seguente relazione:

$$W_{out} = \frac{(W_{in} - F + 2 \cdot Padding)}{Stride} + 1$$

  Nell'esempio precedente: $Stride = 1$, $Padding = 0$, $W_{in} = 32$, $F = 5 \rightarrow W_{out} = 28$. Analoga relazione lega i parametri verticali.

- Il corso on-line di Andrej Karpathy (Stanford - OpenAI) introduce questi concetti in modo molto chiaro ed esaustivo.

  http://cs231n.github.io/convolutional-networks/

  In particolare vedi:

  - Esempio (animato) su convoluzione 3D
  - Esempi di volumi considerando Stride e Padding.

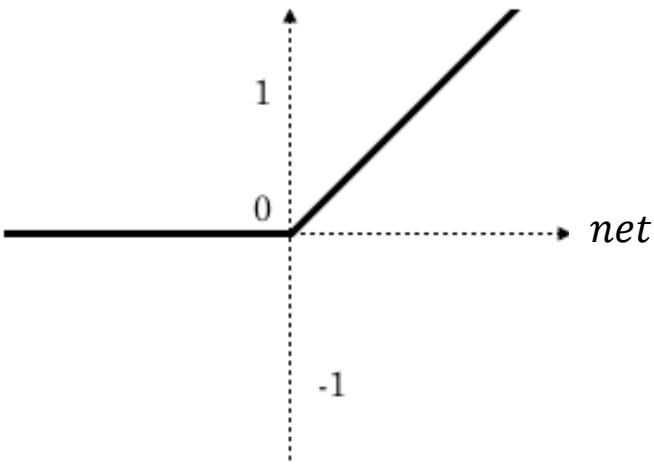
Funzione di attivazione: Relu

- Nelle reti MLP la funzione di attivazione (storicamente) più utilizzata è la sigmoide. Nelle reti profonde, l'utilizzo della sigmoide è problematico per la retro propagazione del gradiente (problema del vanishing gradient):
  - La derivata della sigmoide è tipicamente minore di 1 e l'applicazione della regola di derivazione a catena porta a moltiplicare molti termini minori di 1 con la conseguenza di ridurre parecchio i valori del gradiente nei livelli lontani dall'output. Per approfondimenti ed esempi:

    http://neuralnetworksanddeeplearning.com/chap5.html
  - La sigmoide ha un comportamento saturante (allontanandosi dallo 0). Nelle regioni di saturazione la derivata vale 0 e pertanto il gradiente si annulla.

- L'utilizzo di Relu (Rectified Linear) come funzione di attivazione risolve il problema:

$$f(net) = max(0, net)$$



La derivata vale 0 per valori negativi o nulli di net e 1 per valori positivi

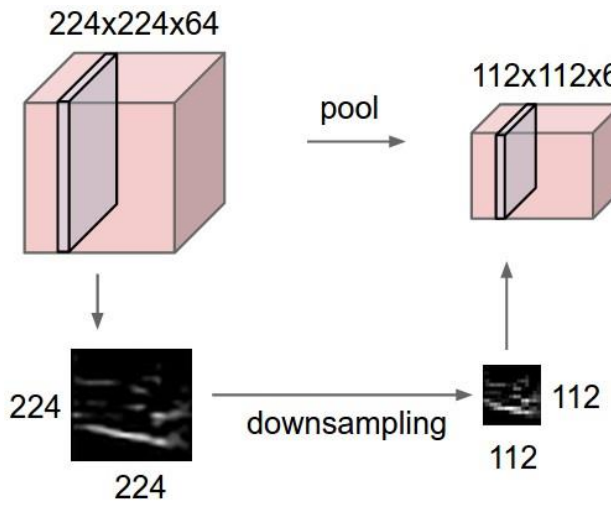
Per valori positivi nessuna saturazione

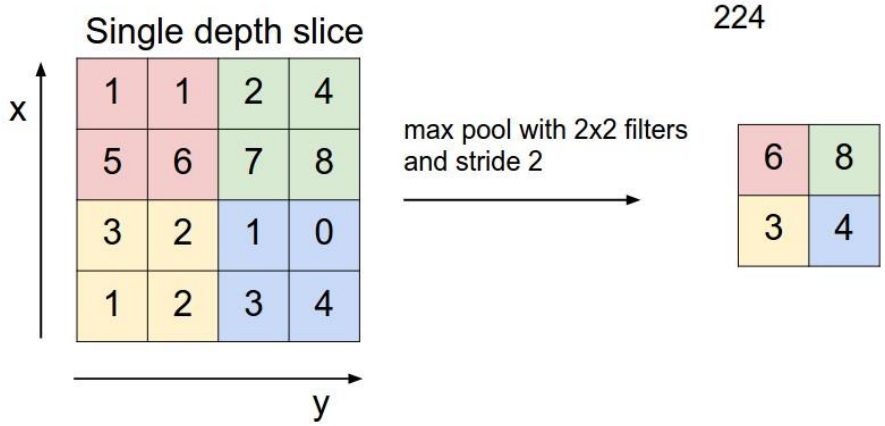
Porta ad attivazioni sparse (parte dei neuroni sono spenti) che possono conferire maggiore robustezza

CNN: Pooling

- Un livello di pooling esegue un'aggregazione delle informazioni nel volume di input, generando feature map di dimensione inferiore. Obiettivo è conferire invarianza rispetto a semplici trasformazioni dell'input mantenendo al tempo stesso le informazioni significative ai fini della discriminazione dei pattern.
- L'aggregazione opera (generalmente) nell'ambito di ciascuna feature map, cosicché il numero di feature map nel volume di input e di output è lo stesso. Gli operatori di aggregazione più utilizzati sono la media (Avg) e il massimo (Max): entrambi «piuttosto» invarianti per piccole traslazioni. Questo tipo di aggregazione non ha parametri/pesi da apprendere.
- Nell'esempio un max-pooling con Filtri 2×2 e $Stride = 2$.

L'equazione precedente per il calcolo di W_{out} è valida anche per pooling (considerando $Padding$ = 0)





Ricomponiamo i pezzi

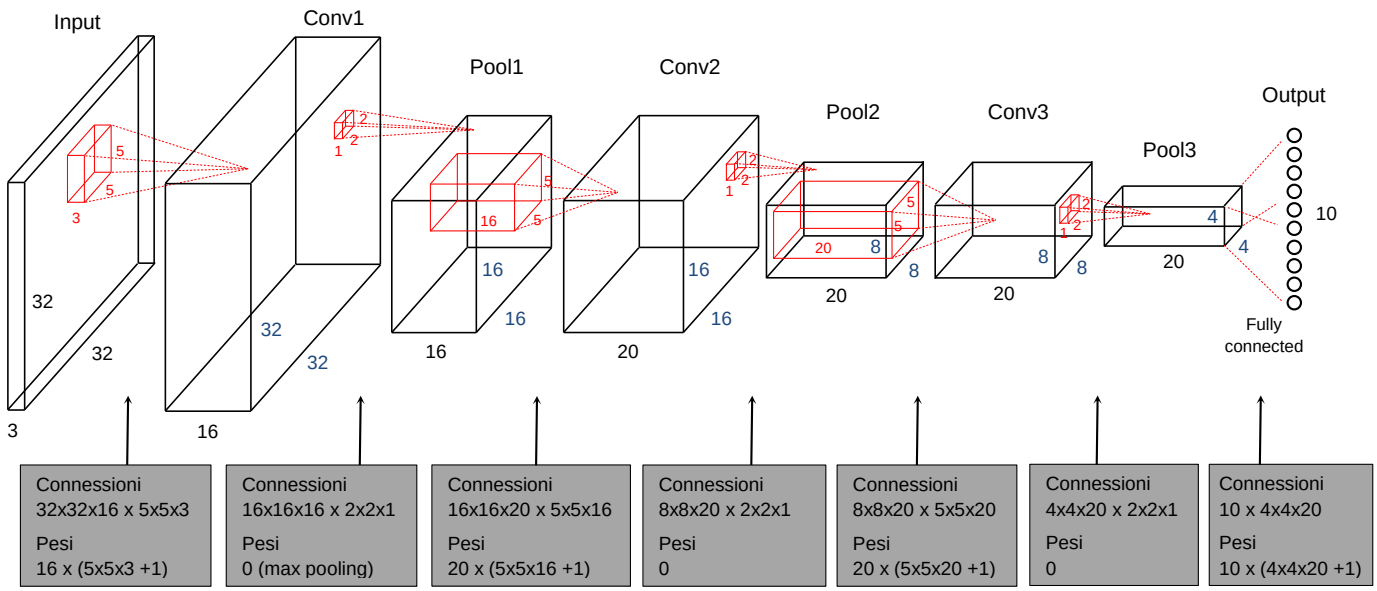
- Esempio 1: Cifar-10 (Javascript running in the browser).

  http://cs.stanford.edu/people/karpathy/convnetjs/demo/cifar10.html

  Architettura

  - Input: Immagini RGB 32x32x3;
  - Conv1: Filtri:5x5, FeatureMaps:16, stride:1, pad:2, attivazione: Relu
  - Pool1: Tipo: Max, Filtri 2x2, stride:2
  - Conv2: Filtri:5x5, FeatureMaps:20, stride:1, pad:2, attivazione: Relu
  - Pool2: Tipo: Max, Filtri 2x2, stride:2
  - Conv3: Filtri:5x5, FeatureMaps:20, stride:1, pad:2, attivazione: Relu
  - Pool3: Tipo: Max, Filtri 2x2, stride:2
  - Output: Softmax, NumClassi: 10

  Disegniamo la rete e calcoliamo neuroni sui livelli, connessioni e pesi



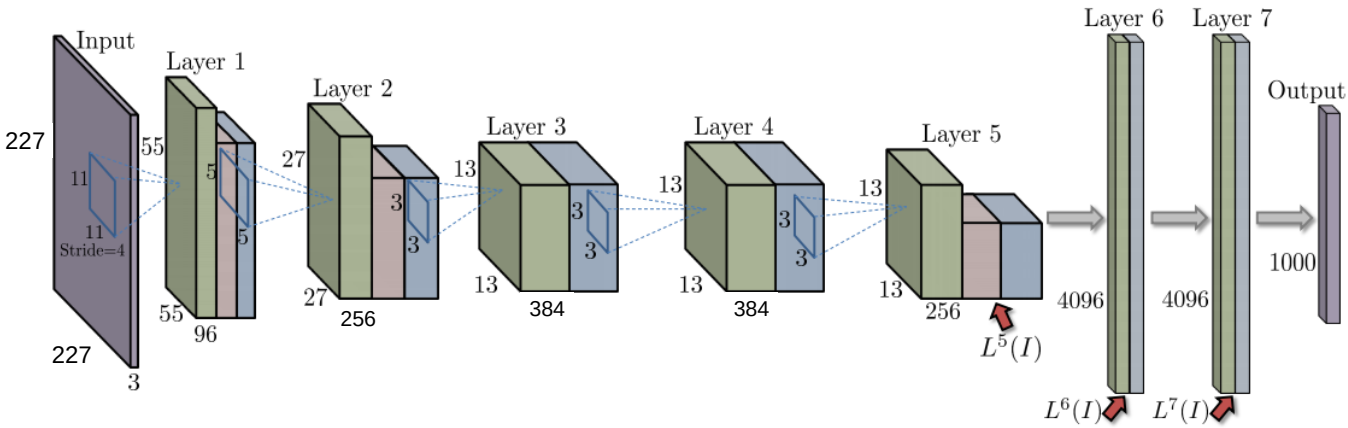
Neuroni totali: 31.562 (incluso livello input)

Connessioni totali: 3.942.784

Pesi totali: 22.466 (inclusi bias)

Ricomponiamo i pezzi (2)

- Esempio 2: CaffeNet (AlexNet porting in Caffe).



Codici colore

- Viola: Input (immagini 227x227x3) e Output (1000 classi di ImageNet)
- Verde: Convoluzione
- Rosa: Pooling (max)
- Blu: Relu activation

Note

- Layer 6, 7 e 8: Fully connected
- Layer 8: Softmax (1000 classi)
- Stride: 4 per il primo livello di convoluzione, poi sempre 1
- Filtri: Dimensioni a scalare: da 11x11 a 3x3
- Feature Map: Numero crescente muovendosi verso l'output
- L^5, L^6, L^7, denotano feature riutilizzabili per altri problemi (vedi transfer-learning e [1]).
- Numero totale di parametri: 60M circa

[1] Babenko et al., *Neural Codes for Image Retrieval*, 2014.

In pratica

- L'implementazione di CNN (da zero) è certamente possibile. Il passo forward non è nemmeno complesso da codificare. D'altro canto la progettazione/sviluppo di software che consente:
  - il training/inference di architetture diverse a partire da una loro descrizione di alto livello (non embedded nel codice)
  - di effettuare il training con backpropagation del gradiente (rendendo disponibili le numerose varianti, parametrizzazioni e tricks disponibili)
  - ottimizzare la computazione su GPU (anche più di una)

  richiede molto tempo/risorse per lo sviluppo/debug.

- Fortunatamente sono disponibili numerosi framework (spesso open-source) che consentono di operare su CNN. Tra i più utilizzati:
  - Caffe (Berkley) – *lo useremo in laboratorio*
  - Theano (Bengio's group – Montreal)
  - Torch (LeCun's group and other contributors)
  - TensorFlow (Google)
  - Digits (Nvidia) – *anche senza programmazione*

Training e Transfer Learning

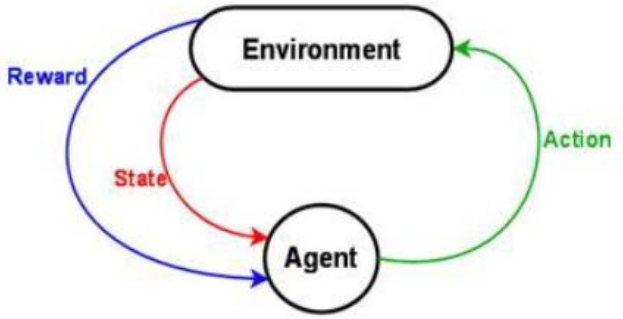
- Il training di CNN complesse (es. AlexNet) su dataset di grandi dimensioni (es. ImageNet) può richiedere giorni/settimane di tempo macchina anche se eseguito su GPU.
  - Fortunatamente, una volta che la rete è stata addestrata, il tempo richiesto per la classificazione di un nuovo pattern (propagazione forward) è in genere veloce (es. 10-100 ms).
- Inoltre il training di una CNN su un nuovo problema, richiede un training set etichettato di notevoli dimensioni (spesso non disponibile). In alternativa al training da zero, possiamo perseguire due strade (Transfer Learning):
  - Fine-Tuning: si parte con una rete pre-trained addestrata su un problema simile e:
    1. si rimpiazza il livello di output con un nuovo livello di output softmax (adeguando il numero di classi).
    2. come valori iniziali dei pesi si utilizzano quelli della rete pre-trained, tranne che per le connessioni tra il penultimo e ultimo livello i cui pesi sono inizializzati random.
    3. si eseguono nuove iterazioni di addestramento (SGD) per ottimizzare i pesi rispetto alle peculiarità del nuovo dataset (non è necessario che sia di grandi dimensioni).
  - Riutilizzo Features: si utilizza una rete esistente (pre-trained) senza ulteriore fine-tuning. Si estraggono (a livelli intermedi) le feature generate dalla rete durante il passo forward (vedi L^5, L^6, L^7 nell'esempio CaffeNet). Si utilizzano queste feature per addestrare un classificatore esterno (es. SVM) a classificare i pattern del nuovo dominio applicativo.

Per approfondimenti: http://cs231n.github.io/transfer-learning/

Reinforcement Learning

L'obiettivo è apprendere un comportamento ottimale a partire dalle esperienze passate.

- Un agente esegue azioni (a) che modificano l'ambiente, provocando passaggi da uno stato (s) all'altro. Quando l'agente ottiene risultati positivi riceve una ricompensa o reward (r) che però può essere temporalmente ritardata rispetto all'azione, o alla sequenza di azioni, che l'hanno determinata.



- Un episodio (o game) è una sequenza finita di stati, azioni, reward:

$$s_0, a_0, r_1, s_1, a_1, r_2, s_2 \ldots, s_{n-1}, a_{n-1}, r_n, s_n$$

- In ciascun stato s_{t-1}, l'obiettivo è scegliere l'azione ottimale a_{t-1}, ovvero quella che massimizza il future reward R_t:

$$R_t = r_t + r_{t+1} + \cdots + r_n$$

- In molte applicazioni reali l'ambiente è stocastico (i.e., non è detto che la stessa azione determini sempre la stessa sequenza di stati e reward), pertanto applicando la logica del «*meglio un uovo oggi che una gallina domani*» si pesano maggiormente i reward temporalmente vicini (discounted future reward):

$$R_t = r_t + \gamma \cdot r_{t+1} + \gamma^2 \cdot r_{t+2} + \cdots + \gamma^{n-t} \cdot r_n \quad \text{(con } 0 \leq \gamma \leq 1\text{)}$$

Q-Learning

- Il discounted future reward può essere definito ricorsivamente:

$$R_t = r_t + \gamma \cdot R_{t+1}$$

- Nel Q-learning la funzione $Q(s,a)$ indica l'ottimalità (o qualità) dell'azione a quando ci si trova in stato s. Volendo massimizzare il discounted future reward si pone:

$$Q(s_t, a_t) = max\, R_{t+1}$$

- Nell'ipotesi che la funzione $Q(s,a)$ sia nota, quando ci si trova in stato s, si può dimostrare che la policy ottimale è quella che sceglie l'azione a^* tale che:

$$a^* = \underset{a}{arg\, max}\, Q(s,a)$$

- Il punto cruciale è dunque l'apprendimento della funzione Q. Data una transizione (quaterna) $< s_t, a_t, r_{t+1}, s_{t+1} >$ possiamo scrivere:

$$Q(s_t, a_t) = max\, R_{t+1} = max(r_{t+1} + \gamma \cdot R_{t+2}) =$$

$$Q(s_t, a_t) = r_{t+1} + \gamma \cdot max\, R_{t+2} = r_{t+1} + \gamma \cdot Q(s_{t+1}, a_{t+1})$$

L'azione a_{t+1} (che non fa parte della quaterna) sarà scelta con la policy ottimale precedente, ottenendo:

$$Q(s_t, a_t) = r_{t+1} + \gamma \cdot \underset{a}{max}\, Q(s_{t+1}, a)$$

nota come Equazione di Bellman.

Q-Learning (2)

L'algoritmo di apprendimento di Q sfrutta l'equazione di Bellman:

inizializza $Q(s,a)$ in modo casuale

esegui m episodi

$t = 0$

do

seleziona l'azione ottimale $a_t = \underset{a}{arg\,max}\, Q(s_t, a)$

esegui a_t e osserva r_{t+1} e s_{t+1}

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha \cdot \left(r_{t+1} + \gamma \cdot \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right)$$

$t = t + 1$

while (episodio corrente non terminato)

end episodi

Dove α è il learning rate: se $\alpha = 1$ l'aggiornamento di $Q(s_t, a_t)$ è eseguito esattamente con l'equazione di Bellman, se $\alpha < 1$, la modifica va nella direzione suggerita dall'equazione di Bellman (ma con passi più piccoli).

Valori tipici iniziali: $\gamma = 0.9$, $\alpha = 0.5$ (α è in genere progressivamente ridotto durante l'apprendimento)

- Problema (pratico): quanto è grande Q ?

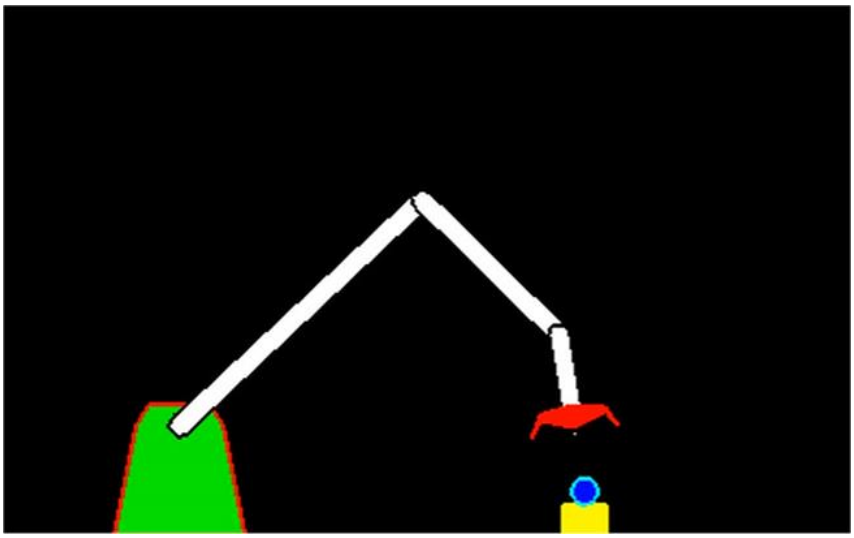
  Tanti valori quanti sono i possibili stati $\times$ le possibili azioni.

Q-Learning: esempio Grasping

Sviluppato con OpenAI Gym + Box2D (*https://gym.openai.com/*)

[Credits: Giammarco Tosi]

Un braccio robotico con tre giunti e pinza, deve prendere una pallina posta su un piedistallo di altezza (Y) e posizione (X) casuali.



Stato: codificato con Δx e Δy della pinza rispetto alla pallina + stato della pinza (aperto/chiuso).

Azioni: ruota a destra/sinistra su ognuno dei tre giunti, inverti stato pinza (chiudi se aperta e viceversa) $\rightarrow$ 7 azioni.

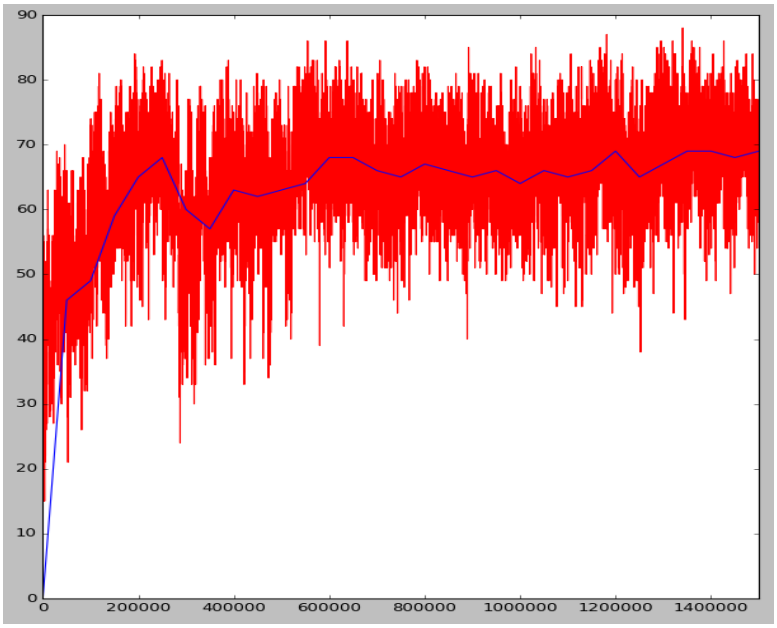
Reward:

- Avvicinamento alla pallina: +1
- Allontanamento dalla pallina: -1
- Movimento a pinza chiusa = -0.2
- Cattura pallina = +100

Q-Learning: esempio Grasping (2)

Q-learning:

- Memorizzazione esplicita della tabella Q
- Parametri addestramento: $\varepsilon = 0.1$, $\gamma = 0.6$, $\alpha = 0.2$



Il grafico rappresenta (in rosso) la percentuale di episodi vinti (pallina catturata) durante l'addestramento ogni 100 episodi. In blu è riportata la percentuale di episodi vinti ogni 5000 episodi.

Video di esempio:

- Pre-training: http://bias.csr.unibo.it/maltoni/ml/Demo/Qarm_pre.wmv
- Post-training: http://bias.csr.unibo.it/maltoni/ml/Demo/Qarm_post.wmv

Deep Q-Learning

Nel 2013 ricercatori della società Deep Mind (immediatamente acquisita da Google) pubblicano l'articolo Playing Atari with Deep Reinforcement Learning dove algoritmi di reinforcement learning sono utilizzati con successo per addestrare un calcolatore a giocare (a livello super-human) a numerosi giochi della consolle Atari.

- La cosa di per sé non sarebbe straordinaria, se non per il fatto che lo stato s osservato dall'agente non consiste di feature numeriche game-specific (es. *la posizione della navicella, la sua velocità*), ma di semplici immagini dello schermo (raw pixel). Questo permette tra l'altro allo stesso algoritmo di apprendere giochi diversi semplicemente giocando.

- Considerando lo stato s formato da 4 immagini dello schermo (a 256 livelli di grigio) e risoluzione 86×86, il numero di stati è $256^{84\times84\times4} \approx 10^{67970}$, più del numero di atomi nell'universo conosciuto! Impossibile gestire esplicitamente una tabella Q di tali dimensioni.

- L'idea consiste nell'approssimare la funzione Q con una rete neurale deep (CNN) che, per ogni stato di input, fornisce in output un valore di qualità per ogni possibile azione. Per maggiori dettagli si veda l'eccellente introduzione di T. Matiisen:

  http://neuro.cs.ut.ee/demystifying-deep-reinforcement-learning/

  https://www.nervanasys.com/demystifying-deep-reinforcement-learning/

- Ulteriori raffinamenti hanno portato allo sviluppo di AlphaGo che nel 2016 ha battuto a Go il campione umano Lee Sedol.

- Altro esempio (codice sorgente in Python): Deep Reinforcement Learning: Pong from Pixels

  http://karpathy.github.io/2016/05/31/rl/