python™

Unit 9

Artificial intelligence (AI)

AI

In scienze dell'informazione, Intelligenza Artificiale (IA) e' l'intelligenza dimostrata dalle macchine (in contrasto con l'intelligenza "naturale", dimostrata dagli esseri umani).

Formalmente:

Studio di "agenti intelligenti": ogni possibile tipo di dispositivo che percepisce l'ambiente circostante ed esegue azioni che massimizzano la probabilita' di raggiungere obiettivi prefissati.

Informalmente:

Il termine "intelligenza artificiale" e' spesso utilizzato per descrivere macchine (calcolatori) che "mimano"

AI

Librerie Python per IA (in particolare per DEEP LEARNING):

(ne esistono molte altre...)

- NumPy
- SciKitLearn
- MatPlotLib
- Keras (interfaccia Python al motore IA **Tensorflow**)

Integrated Development Environment (**IDE**) per analisi dati
(contiene moduli specifici per IA e per Deep earning:

- **KNIME** data analytics platform

https://www.knime.com/knime-analytics-platform

Tensori

Un **TENSORE** e’ essenzialmente, un CONTENITORE di dati.

- Numeri
- Stringhe (molto raramente)

E’ il blocco costitutivo di base di molte techiche di IA moderne… pensate ad esso come ad un contenitore di numeri…

Esistono tensori di diverse dimensioni … In AI (ed in particolare in quella branca di AI che si occupa di studiare sistemi in grado di apprendere: Apprendimanto Automatico o Machine Learning (ML) e’ frequente incontrare tensori le cui dimensioni variano da 0 a 5

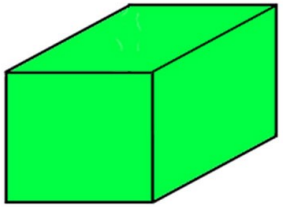
Tensori

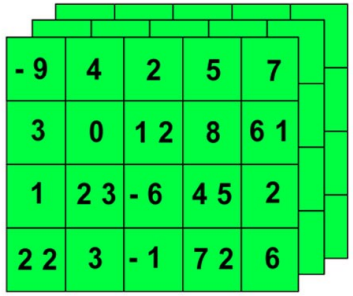
1D TENSOR/
VECTOR

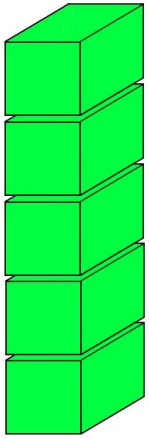
2D TENSOR /
MATRIX

- 9	4	2	5	7
3	0	1 2	8	6 1
1	2 3	- 6	4 5	2
2 2	3	- 1	7 2	6

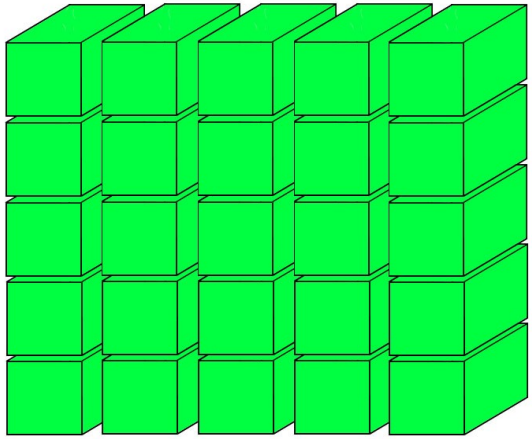
3D TENSOR/
CUBE







4D TENSOR
VECTOR OF CUBES



5D TENSOR
MATRIX OF CUBES

Tensori

- **Tensore 0D** : ogni numero ospitato in un tensore e' uno scalare. Un singolo scalare non ha dimensioni (e' un solo dato) e, quindi, e' un tensore zero dimensionale

In Python scalari che costituiscono tensori sono solitamente ospitati in **array NumPy** (NumPy e' parte di moltissime librerie python per AI).

```
import numpy
x = np.array(5)
print(x)
```

l' output e' il seguente:

```
5
```

Tensori

Il **primo step** di quasi ogni esperimento di AI (ad, in particolare di ML) e' la trasformazione dei dati di input (andamenti di mercato, immagini, video, valori di espressione di geni …) in array NumPy. Il motivo e' semplice: una volta effettuata questa trasformazione possiamo utilizzare direttamente I dati in Tensorflow (sottoforma di tensori).

Di fatto questo e' un processo di organizzazione e standardizzazione dei dati. E' importante capire che OGNI tipo PUO' essere convertito in una rappresentazione numerica! Le immagini avranno codifiche numeriche per ogni singolo pixel e questo vale anche per I video. Il testo puo' essere convertito associando ad ogni carattere un valore numerico…

Tensori

- **Tensore 1D** : e' un VETTORE.

```
x = np.array([1,2,3,4,5])
```

E' possibile ottenere il numero di assi (dimensioni della variabile) di un tensore utilizzando la funzione NumPy ndim:

```
x = np.array([1,2,3,4,5])
x.ndim
```

```
1
```

Tensori

- **Tensore 2D** : e' una MATRICE.

```
x = np.array([[5,10,15,30,25],
              [20,30,65,70,90],
              [7,80,95,20,30]])
```

Una matrice puo' essere usata per immagazzinare I dati di un set di persone. Ad esempio in una mailing list potremmo avere 10000 persone e, per ognuna di esse voler registrare Nome,Cognome, Indirizzo, Citta', Regione, Nazione, Codice postale. 7 caratteristiche per ognuna delle 10000 persone.

Ogni tensore ha una forma "shape" che si adatta perfettamente ai dati contenuti in esso. L'esempio attuale descrive un tensore 2D (10000,7). Si potrebbe essere tentati di dire 10000 righe e 7 colonne ma e' meglio di no! Un tensore puo' essere trasformato in modo che le righe diventino colonne e viceversa, a seconda delle necessita'.

Tensori

- **Tensore 3D** : e' un CUBO di dati.

```
x = np.array([[5,10,15,30,25],
              [20,30,65,70,90],
              [7,80,95,20,30]]
              [[3,0,5,0,45],
              [12,-2,6,7,90],
              [18,-9,95,120,30]]
              [[17,13,25,30,15],
              [23,36,9,7,80],
              [1,-7,-5,22,3]]])
```

Un tensore 3D puo' essere utilizzato per contenere piu' tensori 2D (in questo esempio 3 tensori 2D (matrici) da 9x9 scalari. Se avessimo 10 mailing list da 10000 persone per 7 caratteristiche allora la forma "shape" del tensore sarebbe (**10**.10000.7).

Tensori

- **Tensore 4D e 5D** :

E’ possibile impilare cubi di dati (tensori 3D) per creare “vettori di cubi di dati” … o **tensori 4D** o combinare I cubi di dati in matrici di cubi ottenendo “matrici di cubi di dati” … o tensori 5D.

Esempi di dati reali e di tensori utilizzati per immagazzinarli:

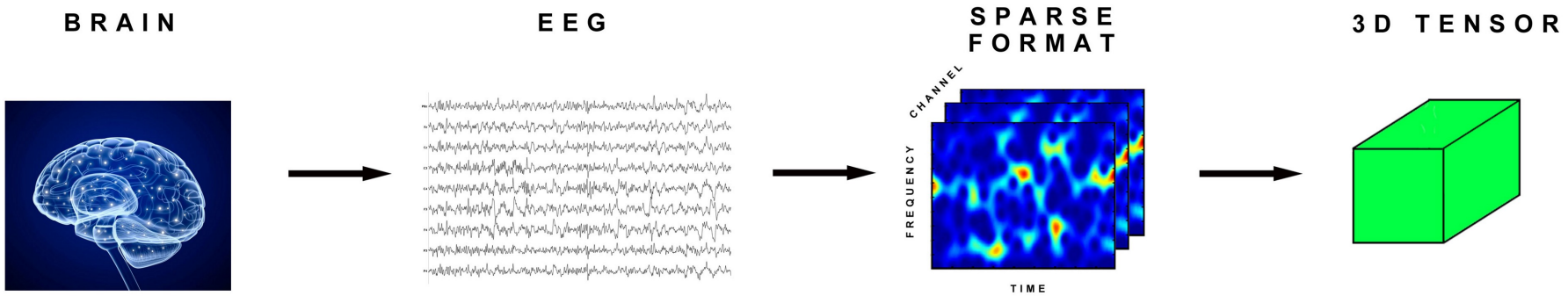
- Serie temporali → tensori 3D
- Immagini → tensori 4D
- Video → tensori 5D

In tutti questi esempi UNA DELLE DIMENSIONI corrisponde alla numerosita’ del campione

Tensori

- **Esempio: dati EEG (serie temporale) → 3D**

Codificabile tramite 3 informazioni (tempo, frequenza, canale)



Se avessimo le scansioni dell'elettroencefalogramma di piu' pazienti potremmo aggiungere una dimensione e otterremmo, cosi', un tensore 4D. Il tensore 4D non ospita solo I dati EEG ma anche TUTTI I CAMPIONI a disposizione (il dataset).

Tensori

- **Esempio: testo → 3D**

Un tweet e’ composto da 140 caratteri (forma originale) ognuno di essi puo’ essere un carattere UTF8 (moltissimi caratteri possibili) ma siamo frequentemente interessati solo ai primi 128 (testo ASCII)

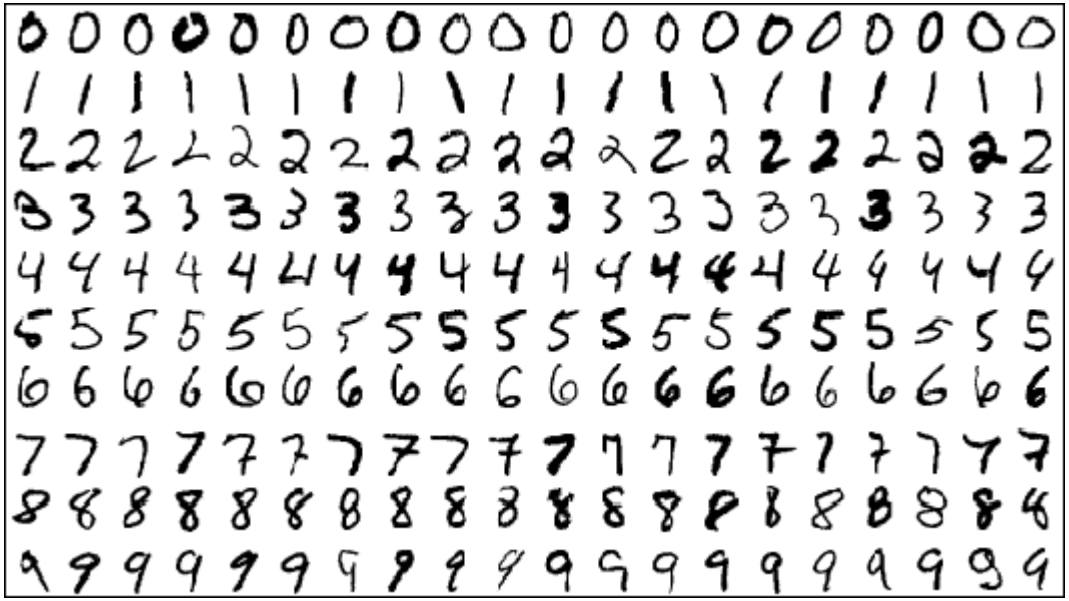
In questo caso potremmo utilizzare un tensore 2D di forma (140,128).

Ma se scarichiamo gli ultimi 10000000 di tweet da Twitter allora avremo un tensore 3D di forma (10000000,140,128)

Tensori

- **Esempio: IMMAGINI → 4D**

Serie di immagini sono contenute, di solito, in tensori 4D. Ogni immagine puo' essere rappresentata tramite 3 caratteristiche: altezza, larghezza, profondita' colore. L'immagine, di per se' , puo' essere rappresentata da un tensore 3D ma un intero set richiede una dimensione in piu', e quindi un tensore 4D.



← **MNIST dataset**

Tensori

- **Esempio: IMMAGINI → 4D … KERAS**

MNIST (dataset per il riconoscimento automatico di cifre scritte a amano) e' stato un problema di difficile soluzione per anni … Attualmente e' considerato risolto (con un 99% di accuratezza raggiungibile da una macchina). E' un buon punto di partenza per iniziare a fare esperimenti di classificazione.

Esso e' disponibile in KERAS (libreria python per AI e deep learning) come segue:

```
from keras.datasets import mnist


(train_images, train_labels), (test_images,
  test_labels) = mnist.load_data()
```

Tensori

- **Esempio: KERAS MNIST**

Il dataset e’ diviso in due contenitori:

- Training (addestramento)
- Test (test)

Ogni immagine e’ fornita insieme ad una etichetta (label) che rappresenta il vero numero scritto a mano nell’immagine (3,5,4,9,...) ed e’ stata aggiunta manualmente da un umano.

Il training set e’ utilizzato per addestrare una rete neurale ed il test set contiene I dati che la rete cerchera’ di classificare dopo l’apprendimento.

Tensori

- **Esempio: KERAS MNIST**

In MNIST ci sono 60,000 immagini di 28 x 28 pixel. Esse hanno una profondita' cromatica di **1** (**scala di grigi** ... un singolo valore scalare compreso tra 0 e 255). Tensorflow rappresenta queste immagini in questa forma : (sample_size, height, width, color_depth) corrispondente ad un tensore **4D (60000, 28, 28, 1)**

Immagini a colori: in ogni immagine RBG a colori (ad esempio un file jpg) il colore di ogni pixel e' determinato da 3 valori: R(ed), B(lue), G(reen). In tal caso avremo sempre un vettore 4D (il primo valore e' sempre la dimensione del campione), ma l'ultima dimensione (color_depth) corrisponderebbe ad un vettore da 3 valori (r,g,b) e, quindi, la forma (shape) del tensore 4D sarebbe (40000, 28, 28, 3)

Tensori

- **Esempio: video → 5D**

Un tipico video e’ rappresentato da Tensorflow con questa shape:
(sample_size, frames, width, height, color_depth)

Se consideriamo un video da 5 min (60 sec x 5 = 300) ad una risoluzione HD (1920 x 1080 pixel) a 15 frame per secondo (300 sec x 15 frame = 4500) con una profondita' cromatica di 3 (RGB) esso sarebbe rappresentato da un tensore 4D con questa forma: (4500, 1920, 1080,3).

Il quinto campo entra in gioco se consideriamo piu’ video … se avessimo 10 video arriveremmo al tensore 5D: (10,4500,1920,1030,3) … pari a 279936000000 valori … ognuno da almeno 32 bit → **1.119744 Tb** (terabyte). Addestrare una rete con un tensore cosi’ grande richiederebbe un tempo maggiore dell’eta’ dell’universo

Tensori

- **Esempio: video → 5D**

Il quinto campo entra in gioco se consideriamo piu' video … se avessimo 10 video arriveremmo al tensore 5D: (10,4500,1920,1030,3) … pari a 279936000000 valori … ognuno da almeno 32 bit → **1.119744 Tb** (terabyte). Addestrare una rete con un tensore cosi' grande richiederebbe un tempo maggiore dell'eta' dell'universo.

Quindi:

Se non siamo in grado di ridurre la dimensione del tensore (ad esempio il numero di frame, rimuovere immagini duplicate ecc.) **non saremo** in grado di effettuare l'addestramento!!!

La fase di pre processing dei dati e' molto importante

Tensori

- **Riepilogo**

I tensori non sono altro che contenitori per qualsiasi tipo di informazione

Essi permettono di rappresentare pressoche' qualsiasi cosa in forma numerica

Il primo step di ogni esperimento di AI basato su reti neurali richiede di trasformare I dati di input in tensori.

Occorre cautela per non far esplodere la dimensione dei tensori

DCNN

- **DEEP CONVOLUTIONAL NEURAL NETWORK**

Le reti convoluzionali sono gli strumenti di elezione per I problemi di computer vision (riconoscimento di immagini, guida automatica,...)

La chiave delle loro capacita' risiede nella loro capacita' di estrarre feature dalle immagini in modo del tutto automatico e di riconoscere, tra di esse, quelle rilevanti ai fini della classificazione.

Esse mimano il comportamento delle reti neurali umane e, quindi, possono ottenere performance davvero notevoli.

DCNN

- **DEEP LEARNING**

Deep learning e’ un’area di ricerca di IA in in cu l’idea forndamentale e’ quella di mettere, uno dopo l’altro, molti strati di neuroni in serie in modo da apprendere rappresentazioni dei dati progressivamente sempre piu’ informative ai fini dell’obiettivo del task (classificazione, predizione,...)

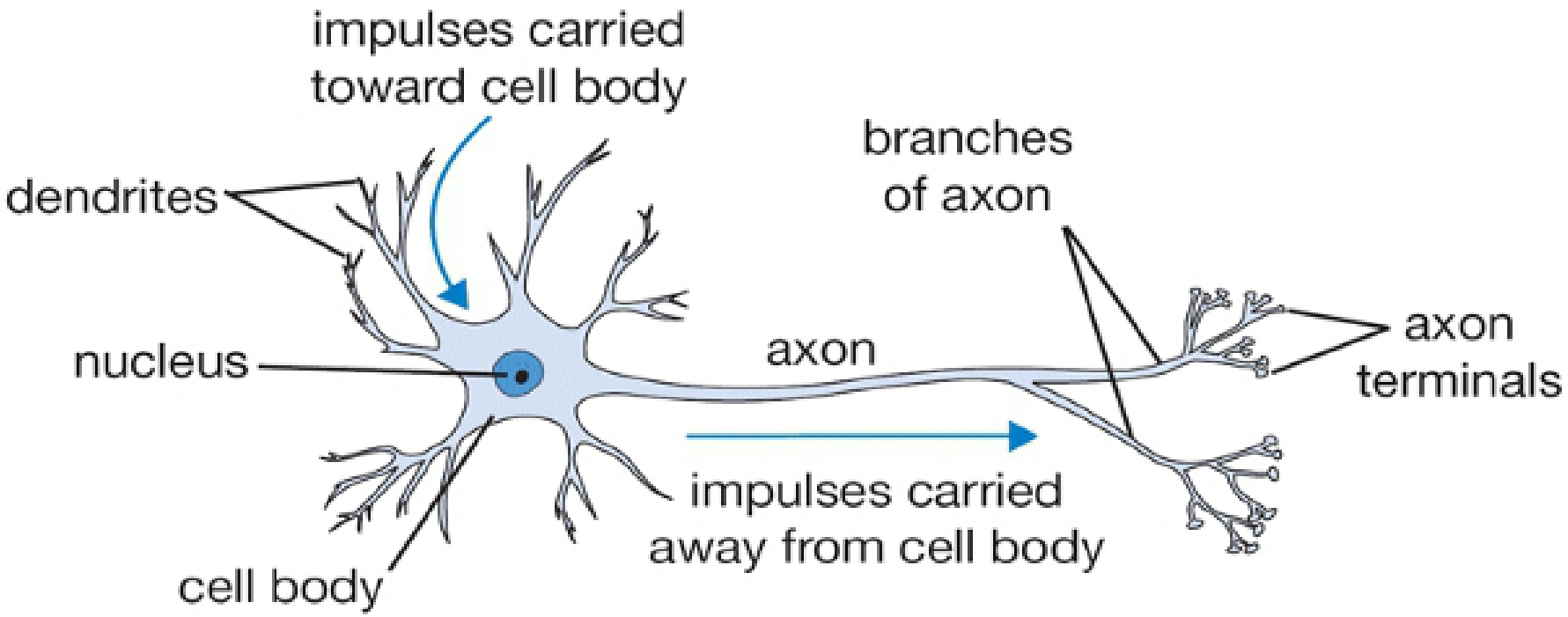
Ognuno di questi strati e’ costituito dareti neurali le quali, a loro volta, consistono in connessioni ra neuroni artificiali.

Le tecniche di deep learning sono possibilo grazie al miglioramento delle capacita’ di CPU e/o GPU. Grazie ad essi e’ possibile utilizzare **molti** strati (da qui il nome di “apprendimento profondo” o DEEP learning)

DCNN

- **NEURONI**

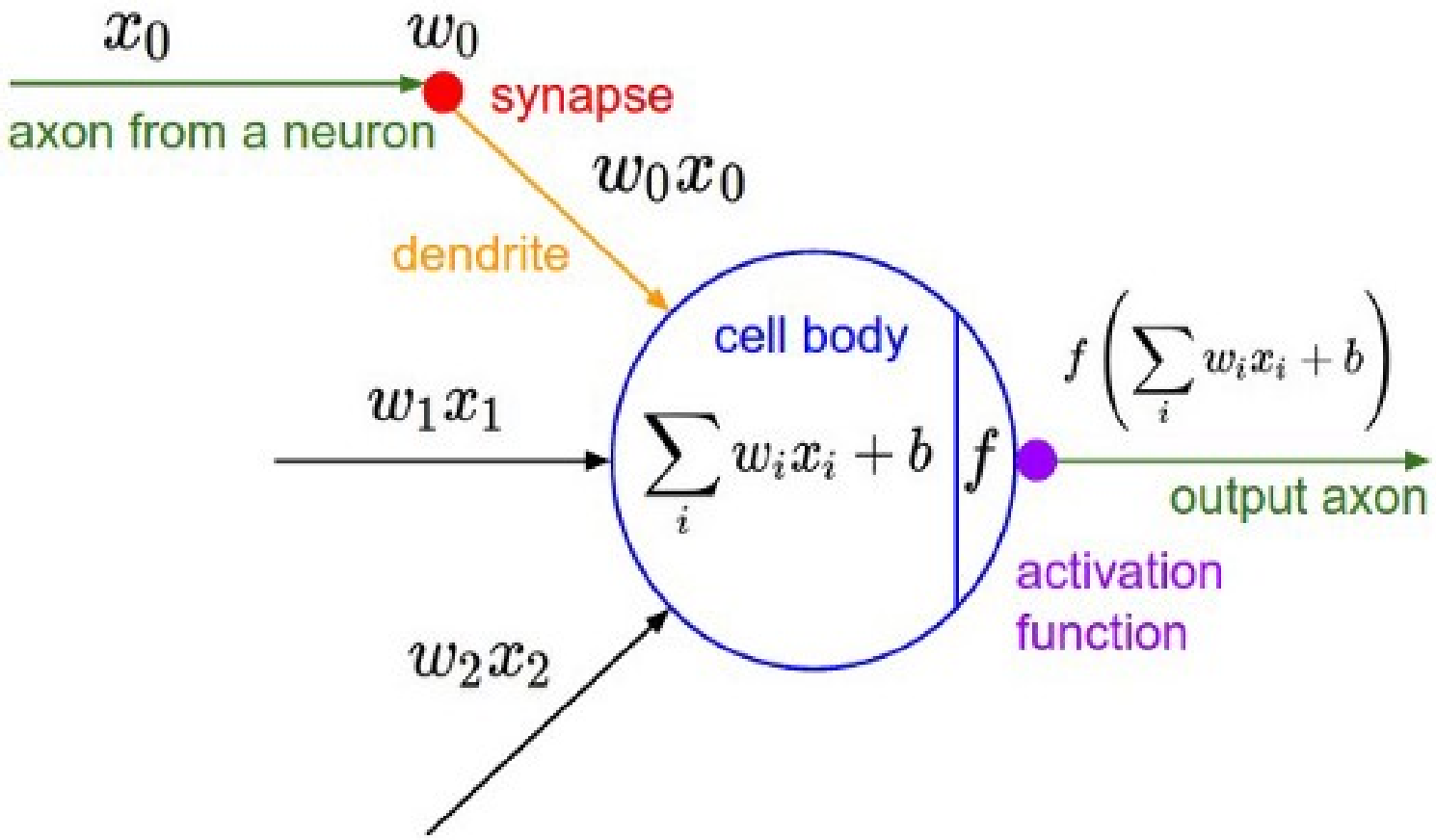
Rappresentazione di un neurone naturale:



DCNN

- **NEURONI**

Modello matematico di neurone:

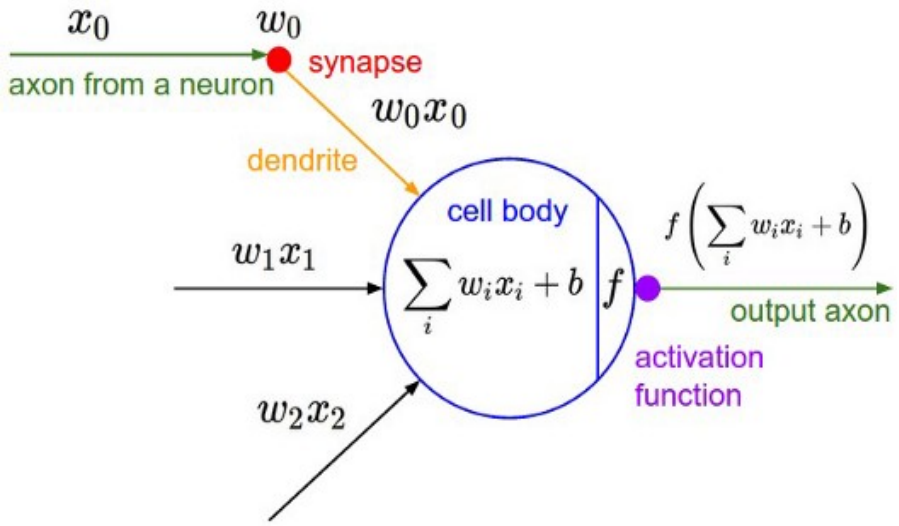


DCNN

- **NEURONI**

Modello matematico di neurone:

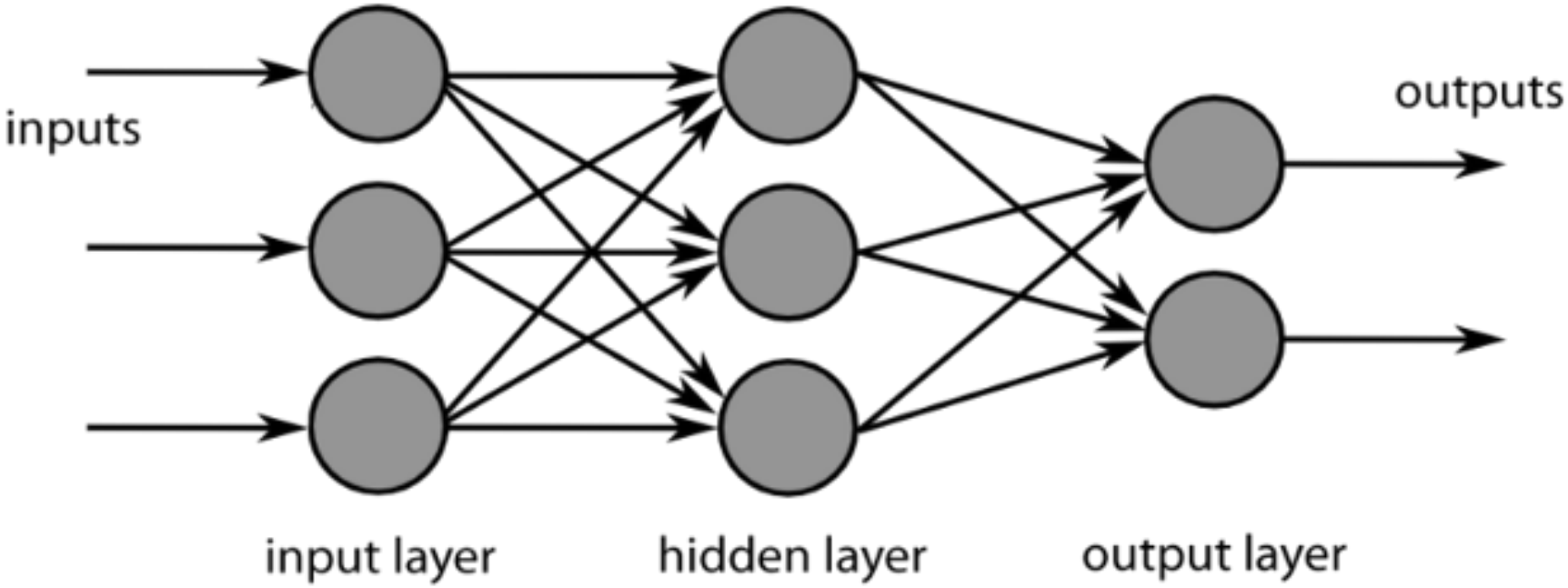
I segnali in ingresso (**x0**) viaggiano attraverso connesisoni tra neuroni. La forza delle connessioni e' rappresentata dai loro pesi (**w0**).Se I segnali in ingresso ed I pesi delle connessioni sono abbastanza forti allora il neurone si attivera' (**funzione di attivazione**) ed emettera', a sua volta, un segnale sull'assone di output.



DCNN

- **Neural Networks**

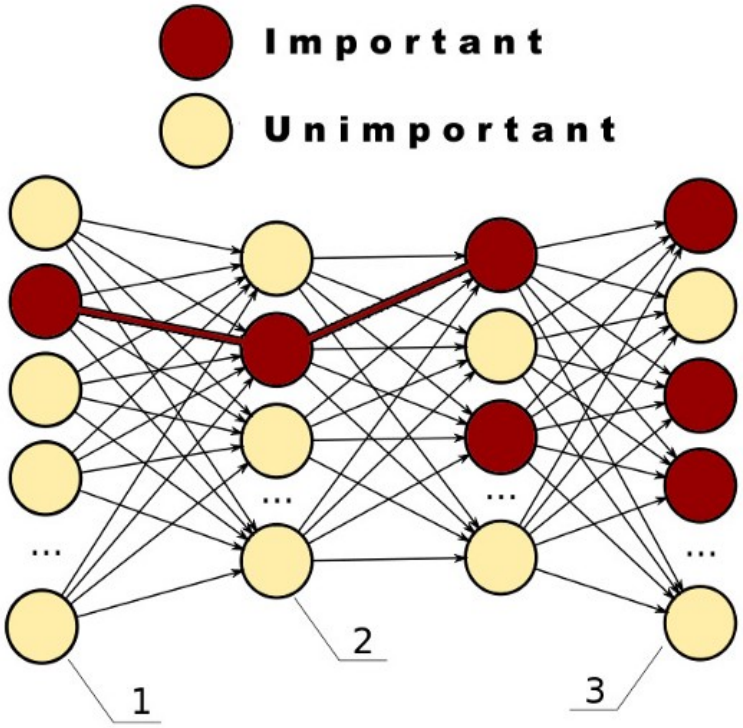
Eempio di semplice rete neurale profonda (3 strati):



DCNN

- **Neural Networks: addestramento**

Attivando alcuni neuroni e non altri e rinforzando/riducendo I **pesi di specifiche connessioni** una rete neurale **impara** cosa e' importante e cosa non lo e' ai fini del task di classificazione.



DCNN

- **DCNN: costruzione e addestramento**

Attivando alcuni neuroni e non altri e rinforzando/riducendo I **pesi di specifiche connessioni** una rete neurale **impara** cosa e' importante e cosa non lo e' ai fini del task di classificazione.

Ora proveremo a costruire ed addestrare una DCNN in python. Le caratteristiche del sistema sono:

- Addestramento
- Dati di input
- Strati (layers)
- Pesi
- Target
- Funzione loss
- Funzione di ottimizzazione
- Predizioni

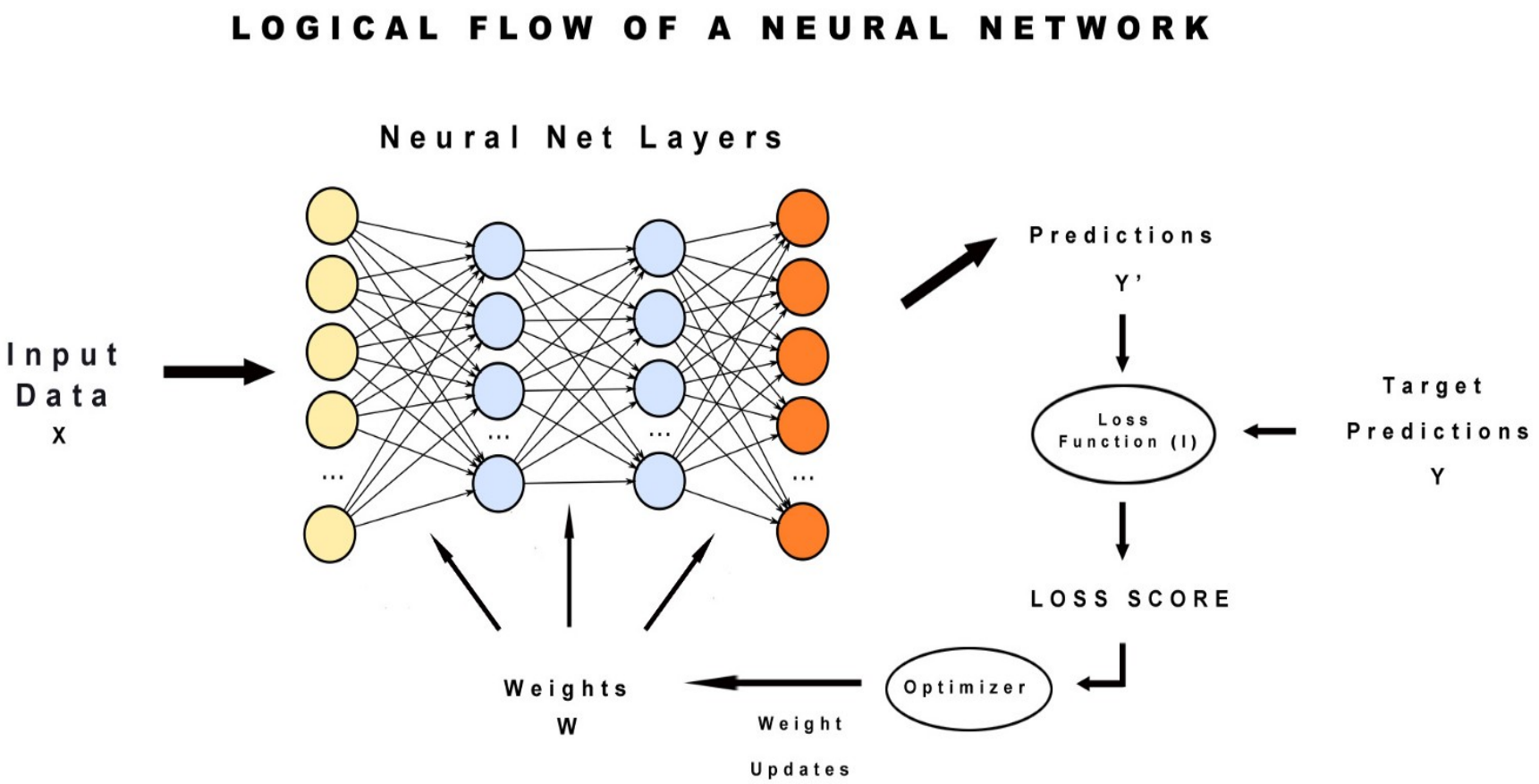
DCNN

- **DCNN: Addestramento**

L’addestramento e’ la fase in cui insegniamo alla rete cio’ che vogliamo che essa apprenda. Si puo’ dividere in 5 step ed e’ un processo ciclico:

1) Creazione di un training set, che chiameremo **x** e caricamento delle sue etichette (labels) come **target y**
2) **Fluire** i dati attraverso la rete neurale producendo i risultati (**predizioni y**)
3) Stimare la "**loss**" della DCNN che e' la **differenza** tra le y predette e i veri target y (le label)
4) Calcolare un **gradiente della loss** (l) per stimare quanto veleocemente ci stiamo muovendo verso o allontanando da i target y
5) Modificare i pesi della rete nella **direzione opposta del**

DCNN


LOGICAL FLOW OF A NEURAL NETWORK
Neural Net Layers
Input Data X
Predictions Y'
Loss Function (l)
Target Predictions Y
LOSS SCORE
Optimizer
Weights W
Weight Updates

DCNN

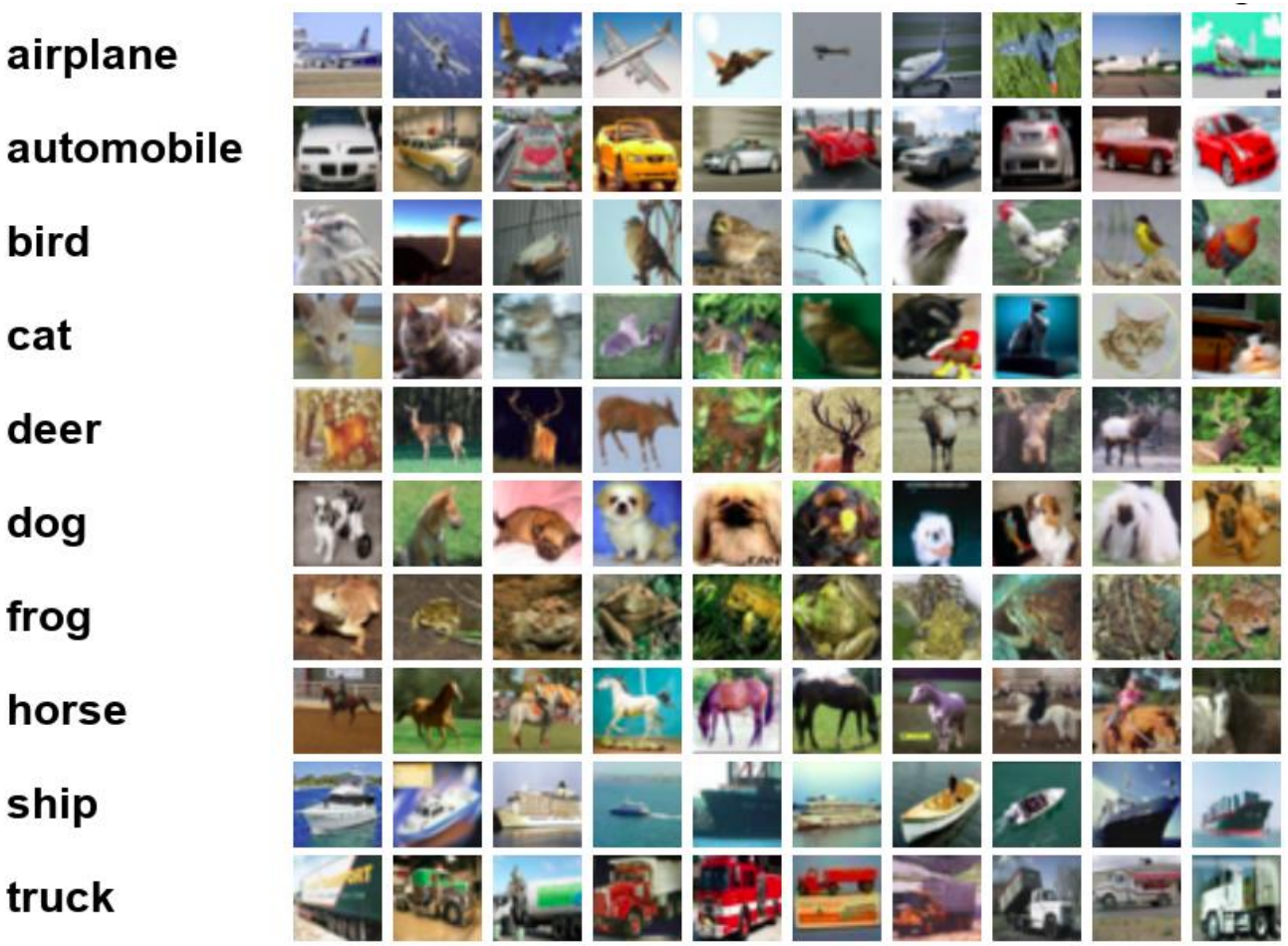
- **DCNN: Dati di input**

In questo caso il set di dati di input e’ una collezione di immagini. Maggiore e’ il loro numero e meglio e’ (le macchine necessitano di grandi quantita’ di esempi per apprendere).

Nel nostro esempio useremo il **CIFAR-10 dataset** che e' stato sviluppato dal Canadian Institute fo Advenced Research. Esso consiste di 60000 32x32 immagini a colori suddivise in 10 classi con 6000 immagini per classe. Utilizzeremo 50000 immagini per il training e 10000 per la fase di test.

DCNN

- **CIFAR-10**



DCNN - python

Inizializziamo il programma, importiamo il dataset e le varie classi di cui avremo bisogno per costruire la nostra DCNN. Keras conosce gia' come ottenere questo particolare dataset in modo automatico e quindi, non avremo molto lavoro da fare per l'acquisizione dei dati e la tensorizzazione.

```
from __future__ import print_function
import numpy as np
from keras.datasets import cifar10
from keras.callbacks import TensorBoard
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation,
  Flatten
from keras.layers import Convolution2D, MaxPooling2D
from keras.utils import np_utils
from keras import backend as K
```

DCNN - python

```
from __future__ import print_function
import numpy as np
from keras.datasets import cifar10
from keras.callbacks import TensorBoard
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation, Flatten
from keras.layers import Convolution2D, MaxPooling2D
from keras.utils import np_utils
from keras import backend as K
np.random.seed(1337)
```

La nostra DCNN parte da una configurazione casuale. Come configurazione iniziale e' buona come qualsiasi altra ma non ci aspettiamo di iniziare ad avere buone prestazioni fin dall'inizio. E' possibile che alcune configurazioni casuali producano risultati incredibilmente buoni per puro caso ... quindi **fissiamo** il generatore di numeri casuali del sistema in modo da non ottenere risultati ottimi a causa di un colpo di fortuna.

DCNN - python

- **DCNN: strati (layers)**

Ora aggiungeremo alcuni **strati di neuroni**. Molte NN utilizzano strati totalmente connessi (in cui ogni neurone e' connesso a tutti gli altri) Strati totalmente connessi sono fantastici per risolvere qualsiasi tipo di problema... ma sfortunatamente non scalano bene per problemi di riconoscimento di immagini. Costruiremo il nostro sistema utilizzando strati CONVOLUZIONALI che sono caratterizzati dal fatto che NON connettono tutti i neuroni tra di loro.

DCNN - python

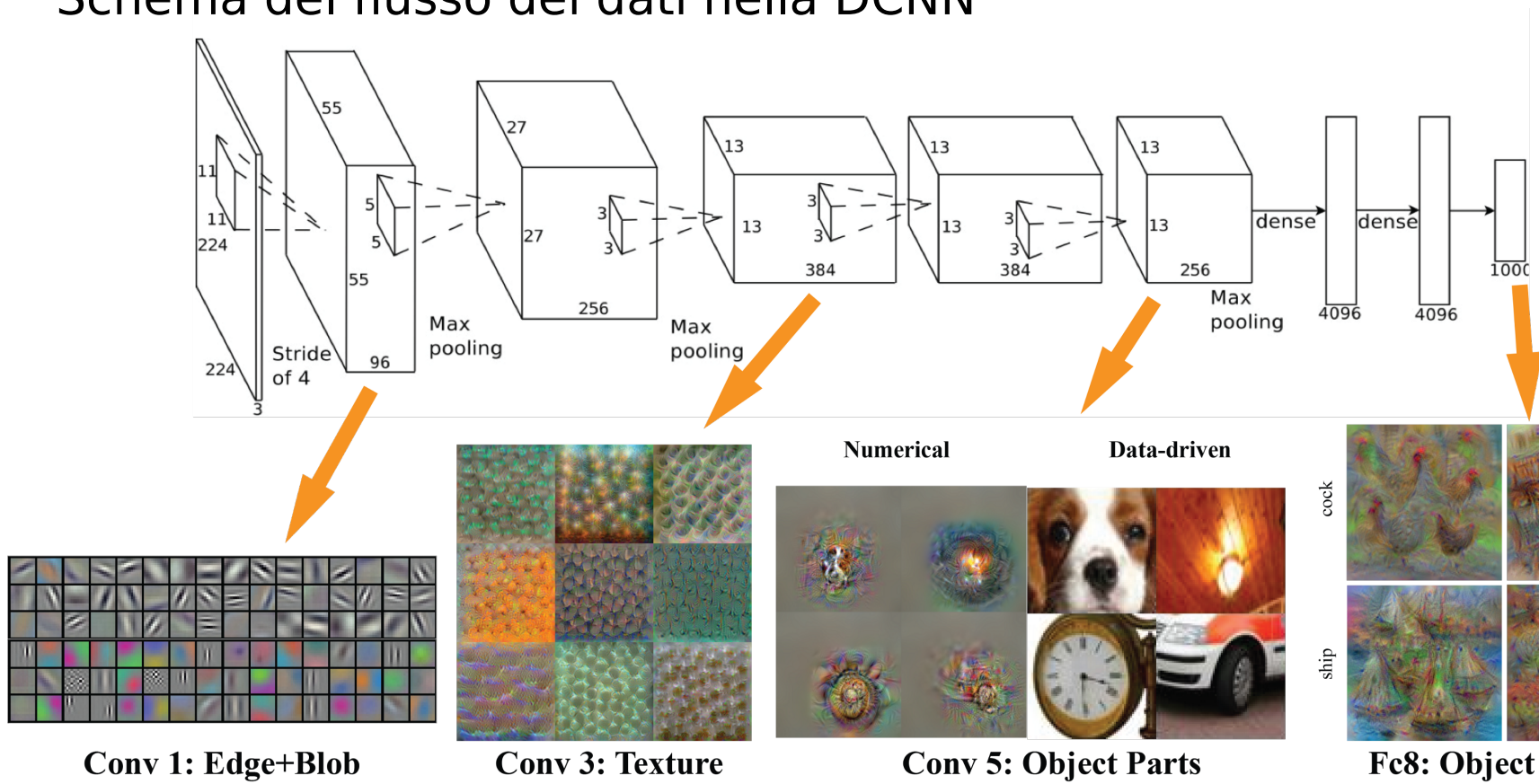
- **DCNN: strati (layers)**

“In CIFAR-10, the image are merely 32x32x3 (32 wide, 32 high, 3 color channels), so a single fully-connected neuron in a first hidden layer of a regular Neural Network would have 32*32*3 = 3072 weights. This amount still seems manageable, but clearly this fully-connected structure does not scale to larger images. For example, an image of more respectible size, e.g. 200x200x3, would lead to neurons that have 200*200*3 = 120,000 weights. Moreover, we would almost certainly want to have several such neurons, so the parameters would add up quickly! Clearly, this full connectivity is wasteful and the huge number of parameters would quickly lead to overfitting.” (fonte: Stanford Course on Computer Vision)

OVERFITTING: La rete e’ addestrata cosi’ bene sui dati di training da avere performance buonissime … ma si comporta male con immagini che non ha mai visto . In sostanza **non ha nessuna utilita’ pratica**!

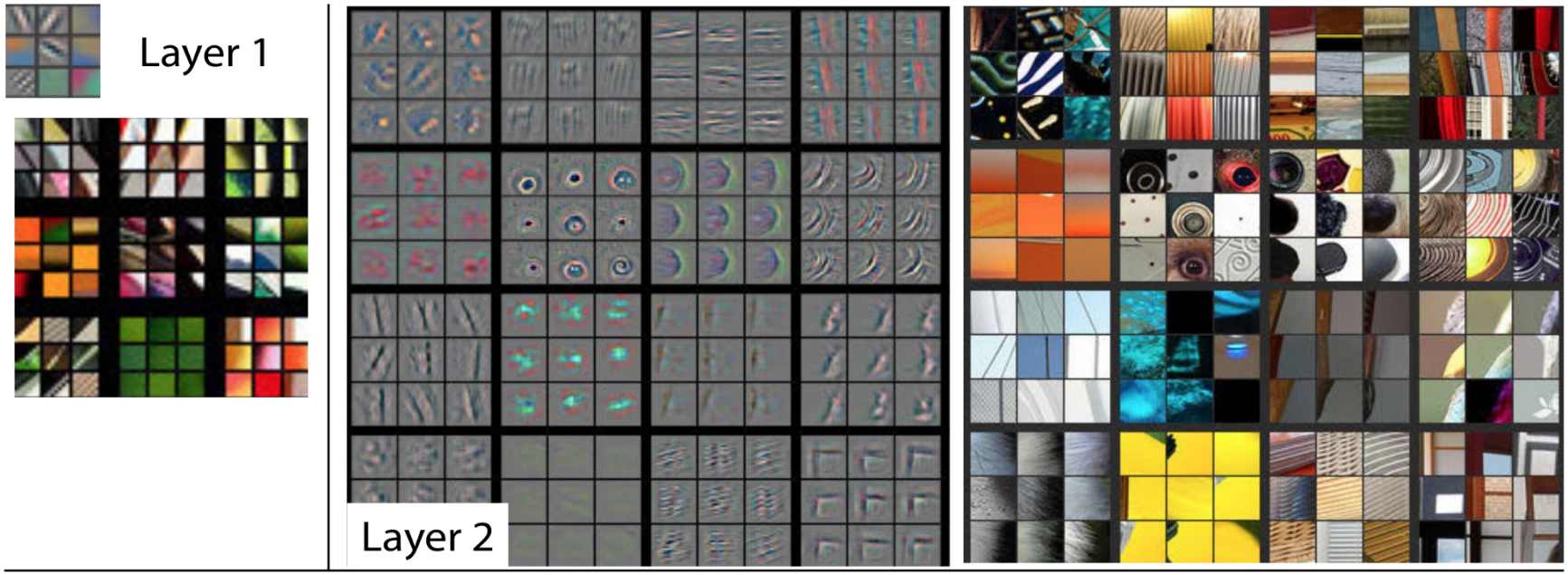
DCNN - python

Schema del flusso dei dati nella DCNN



DCNN - python

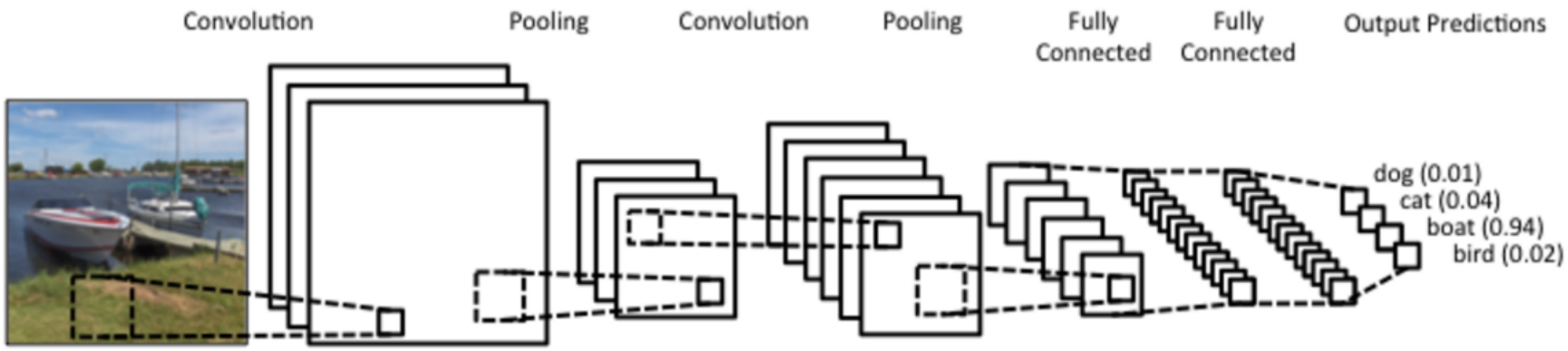
Notiamo che i primi strati "vedono" semplici motivi come linee, colori e forme molto semplici.



Man mano che i dati fluiscono attraverso gli strati di neuroni, il sistema trova pattern semplre piu' complessi come texture ed, eventualmente, deduce la classe degli oggetti rappresentati dalle immagini.

DCNN - python

Schema di tipica architettura di una Deep Convolutional Neural Network (DCNN)



DCNN - python

Aggiungeremo un terzo tipo di layer alla DCNN, il **pooling layer**. L'obiettivo di questo tipo di strato e' semplice. Essi fanno SOTTOCAMPIONAMENTO. In altre parole essi riducono l'immagine di input (rimuovendone parti) e questo riduce il carico computazionale e l'utilizzo della memoria del calcolatore. Com meno informazioni da processare e' possibie lavorare piu' facilmente con le immagini.

Questo tipo di strato, inoltre, aiuta a ridurre un particolare tipo di **overfitting** che si verifica quando la rete si concentra, per caso, su alcune anomalie delle immagini di training che non hanno nulla a che fare con la corretta classificazione. Ad esempio potremmo avere alcuni pixel causati da riflessi sulle lenti della machcina fotografica in moltre immagini di cani. La rete potrebbe concludere erroneamente che tali riflessi sono una caratteristica fondamentale della classe cane...

DCNN – definizione struttura

```
model = Sequential()

model.add(Convolution2D(nb_filters, kernel_size[0], kernel_size[1], border_mode='valid',
input_shape=input_shape))

model.add(Activation('relu'))
model.add(Convolution2D(nb_filters, kernel_size[0], kernel_size[1]))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=pool_size))
model.add(Dropout(0.25))

model.add(Convolution2D(nb_filters, kernel_size[0], kernel_size[1]))
model.add(Activation('relu'))
model.add(Convolution2D(nb_filters, kernel_size[0], kernel_size[1]))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=pool_size))
model.add(Dropout(0.25))

model.add(Flatten())
model.add(Dense(256))
model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Dense(nb_classes))
model.add(Activation('softmax'))
```

DCNN – compilazione modello

Dopo aver definito l'architettura della rete in termini di strati di neuroni specializzati per particolari task dobbiamo "compilare" il modello. In fase di compilazione **forniamo informazioni aggiuntive riguardanti la fase di addestramento**. In particolare specifichiamo:

- **Funzione obiettivo:** la funzione che permette di valutare se ci stiamo avvicinando o allontanando ad una "corretta" predizione dei dati di esempio.
- **Metodo di ottimizzazione:** algoritmo per cambiare I parametri del modello (pesi w della DCNN) in modo da migliorare le performance di predizione in fase di addestramento.
- **Metrica di valutazione:** funzione per valutare le performance del modello addestrato sui dati di TEST.

```
model.compile(loss='categorical_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])
```

DCNN – compilazione modello

Dopo aver definito l'architettura della rete in termini di strati di neuroni specializzati per particolari task dobbiamo "compilare" il modello. In fase di compilazione **forniamo informazioni aggiuntive riguardanti la fase di addestramento**. In particolare specifichiamo:

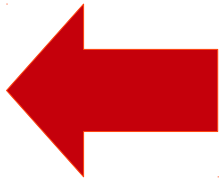
- **Funzione obiettivo:** la funzione che permette di valutare se ci stiamo avvicinando o allontanando ad una "corretta" predizione dei dati di esempio.
- **Metodo di ottimizzazione:** algoritmo per cambiare I parametri del modello (pesi w della DCNN) in modo da migliorare le performance di predizione in fase di addestramento.
- **Metrica di valutazione:** funzione per valutare le performance del modello addestrato sui dati di TEST.

```
model.compile(loss='categorical_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])
```

DCNN – compilazione modello

Funzione obiettivo: la funzione che permette di valutare se ci stiamo avvicinando o allontanando ad una “corretta” predizione dei dati di esempio.

```
model.compile(loss='categorical_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])
```



Funzione obiettivo … in questo esempio utilizziamo la categorical cross entropy. La sua formula e’ la seguente:

$$L(y,\hat{y}) = -\sum_{j=0}^{M}\sum_{i=0}^{N}(y_{ij} * log(\hat{y}_{ij}))$$

Essa e’ adatta ai casi in cui la categorizzazione e’ a singola etichetta, ossia quando ogni esempio appartiene ad una ed una sola classe. Se si usa questa funzione obiettivo nell’output layer e’ obbligatoria una funzione di attivazione **Softmax** (score **DEVONO** sommare a 1).

DCNN – compilazione modello

Categorical ctross entropy …

$$L(y,\hat{y}) = -\sum_{j=0}^{M}\sum_{i=0}^{N}(y_{ij} * log(\hat{y}_{ij}))$$

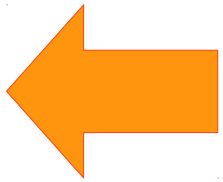
Questo e' il valore predetto. La funzione confronta la distribuzione delle predizioni (le attivazioni nello strato di output) con la vera distribuzione delle etichette in cui la probabilita' delle vara etichetta e' settata a 1 mentre la probabilita' delle altre etichette (altre classi) e' settata a 0.

Punto di vista alternativo: La vera distribuzione delle classi e' impostata tramite one-hot encoding (matrice n_esempi x n_classi) in cui la vera classe ha valore 1 ed il resto 0. La loss calcola quanto le predizioni si allontanano da questo schema: piu' il suo valore e' **basso** e piu' il tensore delel predizioni e' simile alle vere etichette.

DCNN – compilazione modello

Ottimizzatore: e' un algoritmo che dati modello, parametri e funzione loss cerca di ottimizzare il sistema cambiando I parametri (I pesi della DCNN) minimizzando la loss.

```
model.compile(loss='categorical_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])
```



Ottimizzatore ... in questo esempio utilizziamo l'algoritmo Adam.

E' possibile trovare maggiori informazioni su di essi in questo articolo:

https://arxiv.org/pdf/1412.6980v8.pdf

Il metodo e' una forma specializzata di discesa a gradiente per funzioni obiettivo (come la **minimizzazione** della nostra loss) di tipo stocastico. E' efficiente computazionalmente ed e' semplice da implementare.

DCNN – compilazione modello

Ottimizzatore: Adam (NB: Keras contiene molti ottimizzatori ronti all'uso)

Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t.

Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
$m_0 \leftarrow 0$ (Initialize 1st moment vector)
$v_0 \leftarrow 0$ (Initialize 2nd moment vector)
$t \leftarrow 0$ (Initialize timestep)
while θ_t not converged **do**
$t \leftarrow t + 1$
$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
$\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
$\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
end while
return θ_t (Resulting parameters)

DCNN – compilazione modello

Metrica: e' la metrica utilizzata per valutare le performance. Viene clcolata sia per le predizioni effettuate sul training set (durante l'addestramento) che sul test set (composto da esempi mai visti dalla rete durante l'addestramento).

```
model.compile(loss='categorical_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])
```

Metrica ... in questo esempio utilizziamo l'accuratezza. Essa e' una metrica semplice. Dopo aver ottenuto le predizioni possiamo creare una **matrice di confusione**:

		(predicted) P	(predicted) N
(true labels)	P	TP	FN
	N	FP	TN

$$\textbf{Accuracy} = \frac{TP + TN}{TP + FP + TN + FN}$$

Attenzione: accuracy NON E' adatta a dataset sbilanciati!!!!

DCNN – addestramento

E’ la fase che richiede piu’ tempo …

```
model.fit(X_train, Y_train, batch_size=batch_size, nb_epoch=nb_epoch,
          verbose=1, validation_data=(X_test, Y_test), callbacks=[tb])
```

Una volta finito l’addestramento abbiamo il modello ottimizzato disponibile nella variabile model. E’ grazie ad essa che possiamo effettuare la valutazione delle performance del modello:

```
score = model.evaluate(X_test, Y_test, verbose=0)
```