Introduzione ai modelli statistici per il *deep learning*

Alessandro Aere

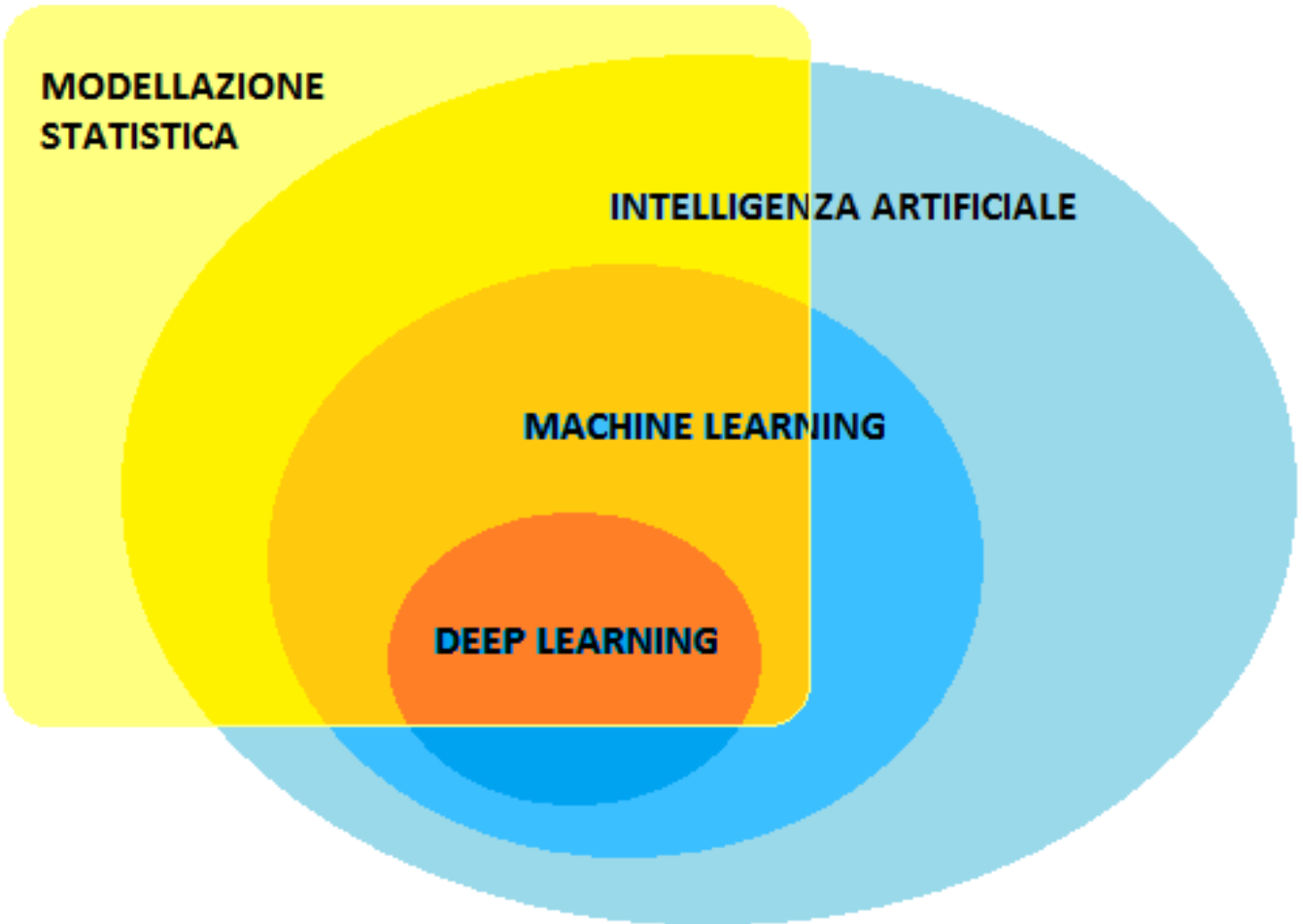
Università degli Studi di Padova

11 maggio 2018

Indice

Descrizione del contesto

Il *deep learning* ha cominciato a svilupparsi a partire dal 2010, ed è nato in un contesto informatico.

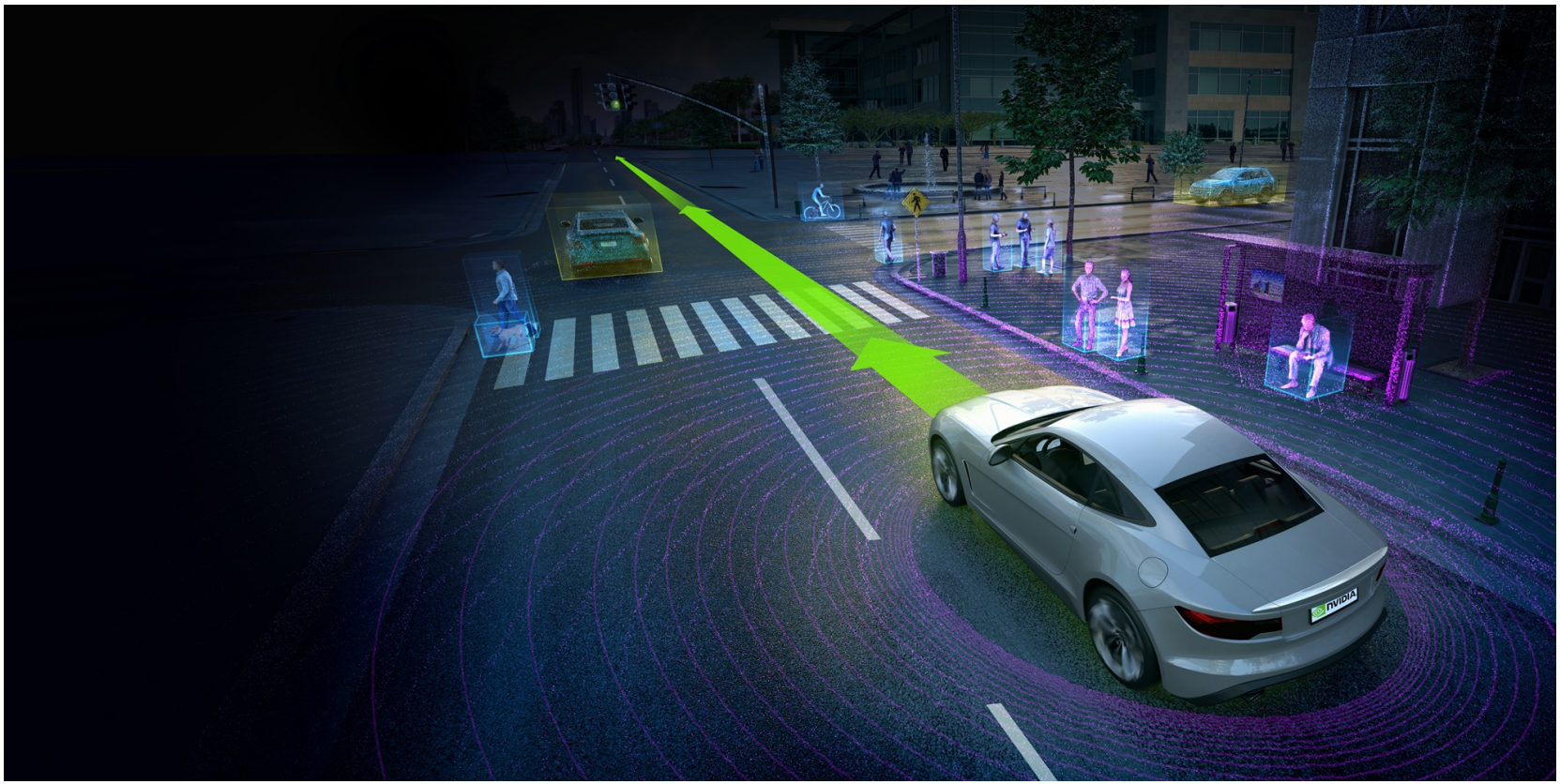


Descrizione del contesto

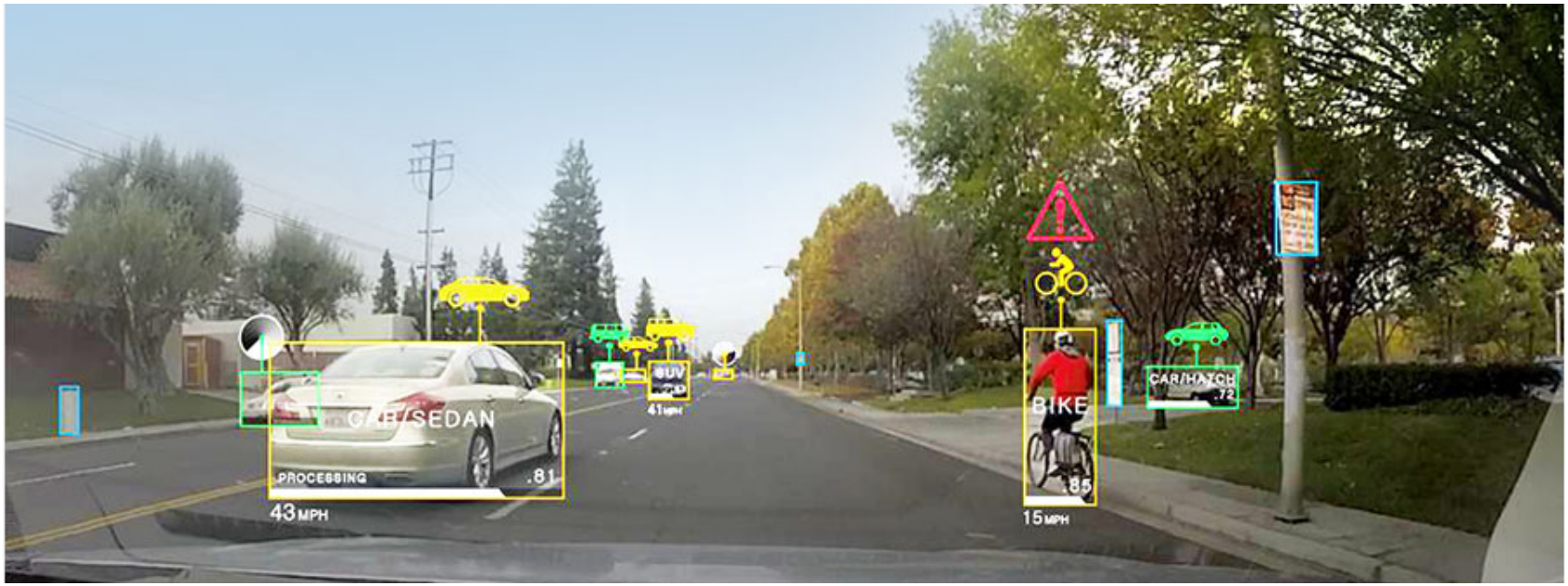
I più rilevanti utilizzatori di *deep learning*:

Una moderna applicazione del *deep learning*

Le *self-driving cars* utilizzano tecnologie di intelligenza artificiale, in particolare il *deep learning*.



Una moderna applicazione del *deep learning*



- Le immagini provenienti dai sensori sono i dati ricevuti in input.
- La macchina elabora i dati grazie a tecniche di *deep learning*, fornendo come output una classificazione degli oggetti rappresentati nell'immagine.
- Sulla base di questa classificazione, la macchina prende le decisioni di conseguenza.

Alcuni campi di applicazione del *deep learning*

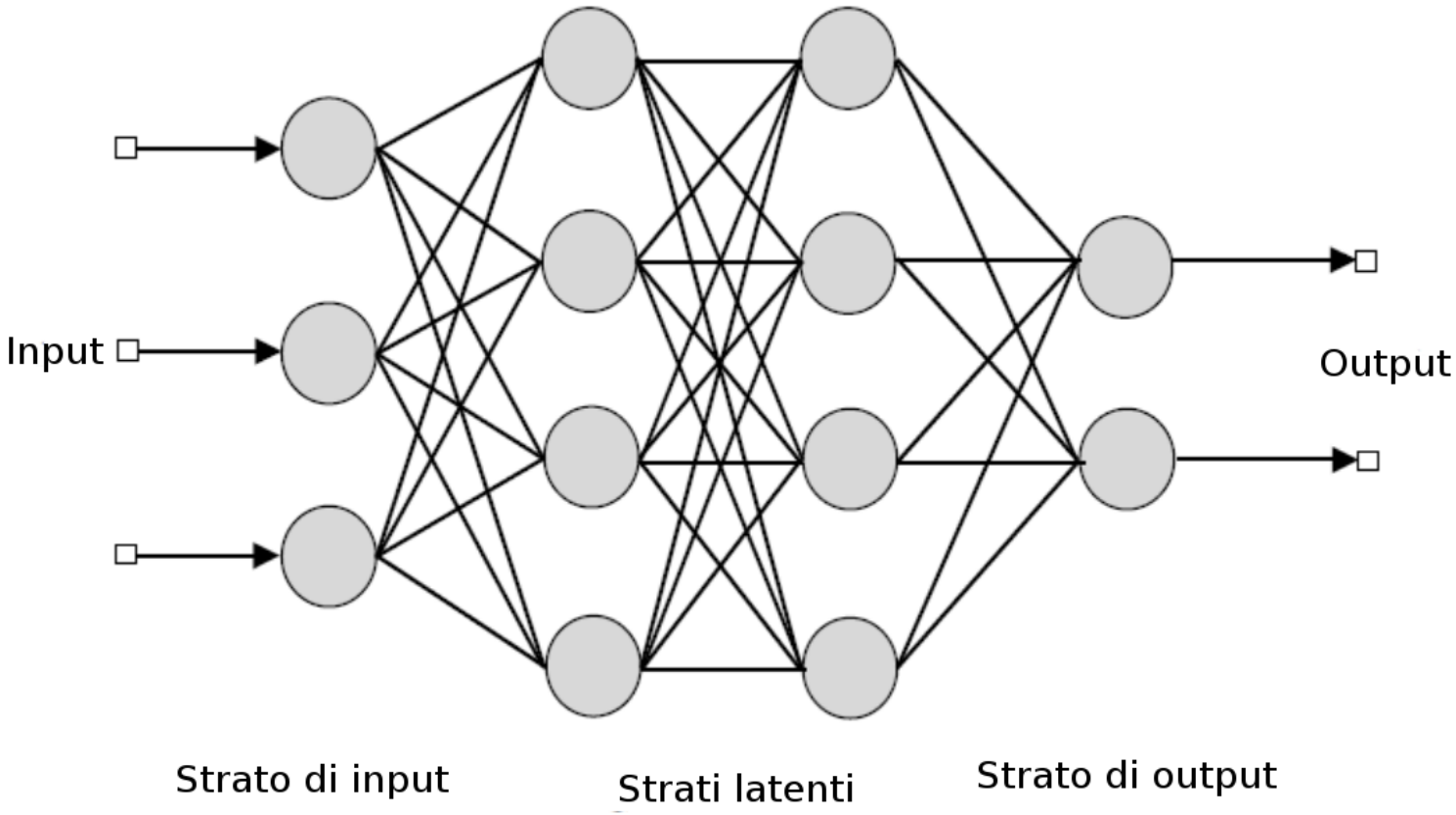
Alcuni campi applicativi del *deep learning* sono:

- Riconoscimento di immagini;
- Riconoscimento vocale;
- Elaborazione del linguaggio naturale;
- Previsione di effetto di farmaci;
- Ricostruzione di circuiti celebrali;
- Previsione gli effetti di mutazioni nel DNA non codificato;
- Analisi dei dati dell'acceleratore di particelle.

Indice

La rete neurale multi-strato (FFNN)

In seguito è raffigurata la struttura della *feed-forward neural network* (FFNN), i cui principali elementi sono i **nodi** e gli **archi**. Ad ogni arco è associato un **parametro**.



La rete neurale multi-strato (FFNN)

Sia

- $a_j^{(l)}$: valore del nodo j-esimo dello strato l-esimo;
- $w_{ij}^{(l)}$: coefficiente associato all'arco che collega il nodo i-esimo dello strato l-esimo con il nodo j-esimo dello strato $(l+1)$-esimo.

Come è legato il generico strato l con lo strato precedente $l-1$?

$$z_j^{(l)} = w_{0j}^{(l-1)} + \sum_{i=1}^{p_{l-1}} w_{ij}^{(l-1)} a_i^{(l-1)}$$

$$a_j^{(l)} = g^{(l)}(z_j^{(l)})$$

dove $g^{(l)}(\cdot)$ viene chiamata **funzione di attivazione**.

La rete neurale multi-strato (FFNN)

Sia

- $\mathbf{x}$ lo strato di input;
- $\mathbf{a}^{(l)}$ lo strato latente l-esimo;
- $W^{(l)}$ la matrice di parametri che succede lo strato l-esimo;
- $\mathbf{y}$ lo strato di output.

Nella forma vettoriale, la relazione tra il generico strato l e lo strato precedente $l-1$ diventa

$$\mathbf{z}^{(l)} = W^{(l-1)}\mathbf{a}^{(l-1)}$$

$$\mathbf{a}^{(l)} = g^{(l)}(\mathbf{z}^{(l)}).$$

La relazione che lega il vettore di input $\mathbf{x}$ con quello di output $\mathbf{y}$ è

$$\mathbf{y} = g^{(L)}\{W^{(L-1)}g^{(L-1)}[\dots W^{(2)}g^{(2)}(W^{(1)}\mathbf{x})]\}$$

La rete neurale multi-strato (FFNN)

Problema di regressione univariato

- Si ha tipicamente un solo nodo di output.
- Una opportuna scelta della funzione di attivazione applicata all'ultimo strato latente è la **funzione identità**:

$$g^{(L)}(\mathbf{z}^{(L)}) = \mathbf{z}^{(L)}$$

Problema di classificazione

- Il numero di nodi di output coincide con il numero di classi della variabile risposta.
- Una opportuna scelta della funzione di attivazione applicata all'ultimo strato latente è la **funzione logistica multinomiale** (softmax):

$$g^{(L)}(\mathbf{z}^{(L)}) = \frac{e^{\mathbf{z}^{(L)}}}{\sum_{j=1}^{K} e^{\mathbf{z}^{(L)}}}$$

Stima dei parametri

Stima dei parametri $\hat{\mathbf{W}}$ della rete neurale:

$$\hat{\mathbf{W}} = \arg\min \left\{ \frac{1}{n} \sum_{i=1}^{n} L[y_i, f(x_i; \mathbf{W})] \right\}$$

Principale funzioni di perdita per problemi di **regressione**:

- **Errore quadratico medio**, $\text{MSE}(\mathbf{W}) = \frac{1}{n} \sum_{i=1}^{n} (y_i - f(x_i; \mathbf{W}))^2$

Principale funzioni di perdita per problemi di **classificazione**:

- **Cross-entropia**, $\text{H}(\mathbf{W}) = -\sum_{i=1}^{n} \sum_{k=1}^{K} y_{ik} \log f_k(x_i; \mathbf{W})$

Algoritmo di *backpropagation*

L'algoritmo largamente più utilizzato per stimare le reti neurali, sia a strato singolo che multi-strato, è la *backpropagation*. Questo algoritmo

- ha la capacità di stimare i parametri con un basso costo computazionale;
- è iterativo (ogni iterazione dell'algoritmo viene chiamata **epoca**);
- è composto da due fasi: con il *passo in avanti* si ottiene $\hat{f}(x_i; \mathbf{W})$, tenendo fisso $\mathbf{W}$, mentre con il *passo all'indietro* vengono aggiornati i parametri;
- necessita solamente del calcolo del gradiente primo, il quale si ottiene in modo efficiente nel *passo all'indietro*.

Logica dell'algoritmo di *backpropagation*

Passo in avanti:

1. Calcolare il valore dei nodi, utilizzando i valori correnti dei parametri.

Passo all'indietro:

- Fase di propagazione:
  2. L'obiettivo è quello di ricavare le derivate parziali della funzione di perdita rispetto ai parametri. Per alleggerire il costo computazionale, si ricavano prima quelle rispetto a $\mathbf{z}$, definite $\delta_2, ..., \delta_L$. Di queste è necessario calcolare solamente δ_L, quella riferita all'ultimo strato.
  3. Il gradiente viene propagato all'indietro, in modo ricorsivo, attraverso l'**equazione di back-propagation** (operazione lineare).
  4. Avendo $\delta_2, ..., \delta_L$ è possibile ricavare le derivate parziali della funzione di perdita, rispetto ai parametri, con una semplice operazione lineare.
- Fase di aggiornamento:
  5. Aggiornare i parametri usando il **gradient descent** e le derivate calcolate al punto 4.

6. Usare i nuovi valori per i parametri nell'iterazione successiva (**epoca**).

Fase di propagazione

Punto 2

La logica consiste nel ricavare alcune quantità, denominate $\delta_L, ..., \delta_2$, utili per calcolare le derivate parziali in modo iterativo. Il generico elemento δ_l si ottiene come $\partial L[y_i, \hat{f}(x_i; \mathbf{W})]$ rispetto a $\mathbf{z}^{(l)}$.
Applicando la "regola della catena" si può scrivere δ_L come

$$\begin{aligned}
\delta^{(L)} &= \frac{\partial L[y_i, \hat{f}(x_i; \mathbf{W})]}{\partial \mathbf{z}^{(L)}} \\
&= \frac{\partial L[y_i, \hat{f}(x_i; \mathbf{W})]}{\partial \hat{f}(x_i; \mathbf{W})} \frac{\partial \hat{f}(x_i; \mathbf{W})}{\partial \mathbf{z}^{(L)}} \\
&= \frac{\partial L[y_i, \hat{f}(x_i; \mathbf{W})]}{\partial \hat{f}(x_i; \mathbf{W})} \circ \dot{g}^{(L)}(\mathbf{z}^{(L)}),
\end{aligned}$$

dove $\dot{g}^{(L)}(\mathbf{z}^{(L)})$ indica la derivata prima di $g^{(L)}(\mathbf{z}^{(L)})$ e il simbolo $\circ$ indica il prodotto di Hadamard (o prodotto elemento per elemento).

Fase di propagazione

Punto 3

È possibile scrivere la quantità $\delta^{(l)}$ come

$$\begin{aligned}
\delta^{(l)} &= \frac{\partial L[y_i, \hat{f}(x_i; \mathbf{W})]}{\partial \mathbf{z}^{(l)}} \\
&= \frac{\partial \mathbf{a}^{(l)}}{\partial \mathbf{z}^{(l)}} \frac{\partial \mathbf{z}^{(l+1)}}{\partial \mathbf{a}^{(l)}} \frac{\partial L[y, \hat{f}(x_i; \mathbf{W})]}{\partial \mathbf{z}^{(l+1)}} \\
&= \frac{\partial \mathbf{a}^{(l)}}{\partial \mathbf{z}^{(l)}} \frac{\partial \mathbf{z}^{(l+1)}}{\partial \mathbf{a}^{(l)}} \delta^{(l+1)} \\
&= \dot{g}^{(l)}(\mathbf{z}^{(l)}) \circ \left(W^{(l)'} \delta^{(l+1)} \right),
\end{aligned}$$

dove $\dfrac{\partial \mathbf{z}^{(l+1)}}{\partial \mathbf{a}^{(l)}} = W^{(l)'}$ è il gradiente primo di $\mathbf{z}^{(l+1)}$ rispetto ad $\mathbf{a}^{(l)}$.
Questa espressione viene chiamata **equazione di backpropagation**.

Fase di propagazione

Punto 4

Avendo $\delta_2, ..., \delta_L$, è possibile ricavare le derivate

$$\begin{aligned}\frac{\partial L[y_i, f(x_i; \mathbf{W})]}{\partial W^{(l)}} &= \frac{\partial L[y_i, f(x_i; \mathbf{W})]}{\partial \mathbf{z}^{(l+1)}} \frac{\partial \mathbf{z}^{(l+1)}}{\partial W^{(l)}} \\ &= \delta^{(l+1)} \mathbf{a}^{(l)'},\end{aligned}$$

dove $\frac{\partial \mathbf{z}^{(l+1)}}{\partial W^{(l)}} = \mathbf{a}^{(l)'}$ è il gradiente primo di $\mathbf{z}^{(l+1)}$ rispetto ad $W^{(l)}$.

Nota

Questa è la fase di propagazione per una generica osservazione (x_i, y_i), con $i = 1, ..., n$. La procedura va svolta per ogni osservazione.

Fase di aggiornamento: *gradient descent*

Definizione

Il **gradient descent** è una tecnica numerica iterativa, che permette di trovare il punto di ottimo di una funzione in più variabili.

L'aggiornamento dei parametri, al passo 5, avviene secondo la formula

$$W_{t+1}^{(l)} = W_t^{(l)} - \eta \cdot \Delta L(W_t^{(l)}; x, y), \qquad \text{per } l = 1, ..., L-1$$

dove

$$\Delta L(W_t^{(l)}; x, y) = \frac{1}{n} \sum_{i=1}^{n} \frac{\partial L[y_i, f(x_i; W)]}{\partial W_t^{(l)}}.$$

Learning rate

Il parametro di regolarizzazione $\eta \in (0, 1]$ viene chiamato *learning rate*, e determina la grandezza dello spostamento.

Mini-batch gradient descent

Problema

L'utilizzo di tutti i dati per effettuare un solo passo di aggiornamento comporta costi computazionali notevoli e rallenta di molto la procedura di stima.

Soluzione

Viene introdotta la tecnica del *mini-batch gradient descent*. Ciò consiste nel suddividere il *dataset* in sottocampioni di numerosità fissata $m \ll n$, dopo una permutazione casuale dell'intero insieme di dati.
L'aggiornamento viene quindi attuato utilizzando ciascuno di questi sottoinsiemi, attraverso la formula

$$W_{t+1}^{(l)} = W_t^{(l)} - \eta \cdot \Delta L(W_t^{(l)}; x^{(i:i+m)}, y^{(i:i+m)}),$$

dove $(i : i + m)$ è l'indice per riferirsi al sottoinsieme di osservazioni che vanno dalla i-esima alla $(i + m)$-esima.

Ulteriori sviluppi del *gradient descent*

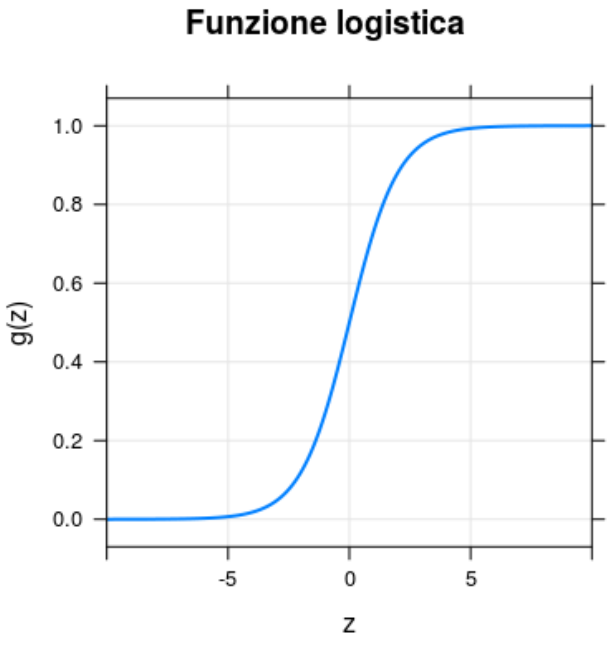
In seguito, sono stati sviluppati altri ottimizzatori per effettuare l'aggiornamento dei parametri, in modo da:

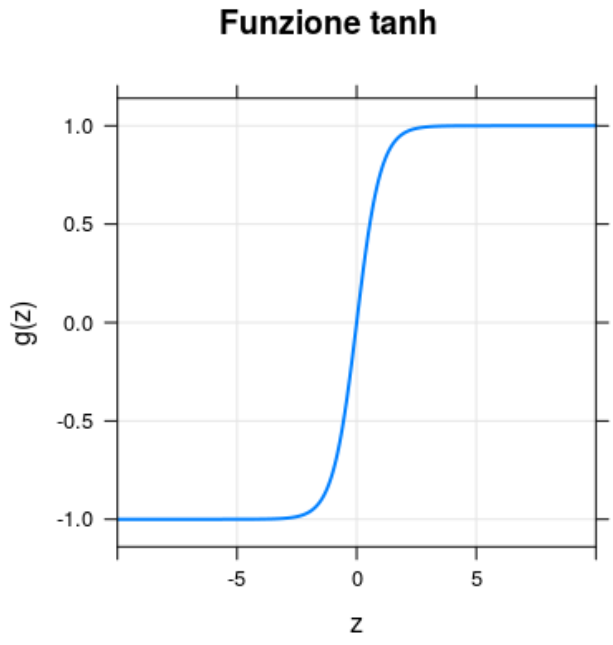
- scegliere il *learning rate* in modo adattivo, evitando la fase di regolarizzazione;
- permettere l'uso di diversi *learning rate* a seconda del parametro a cui sono affiancati;
- ridurre la propensione a rimanere intrappolati in minimi locali.

I principali ottimizzatori utilizzati nel *deep learning* sono:

- Adagrad
- Adadelta
- **Adam**

Le classiche funzioni di attivazione delle reti neurali





Funzione logistica (sigmoidale)

$$\text{logistica}(z) = \frac{1}{1 + e^{-z}}.$$

Tangente iperbolica

$$\begin{aligned}\tanh(z) =& \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}} \\ =& 2 \cdot \text{logistica}(2z) - 1.\end{aligned}$$

La funzione di attivazione ReLU

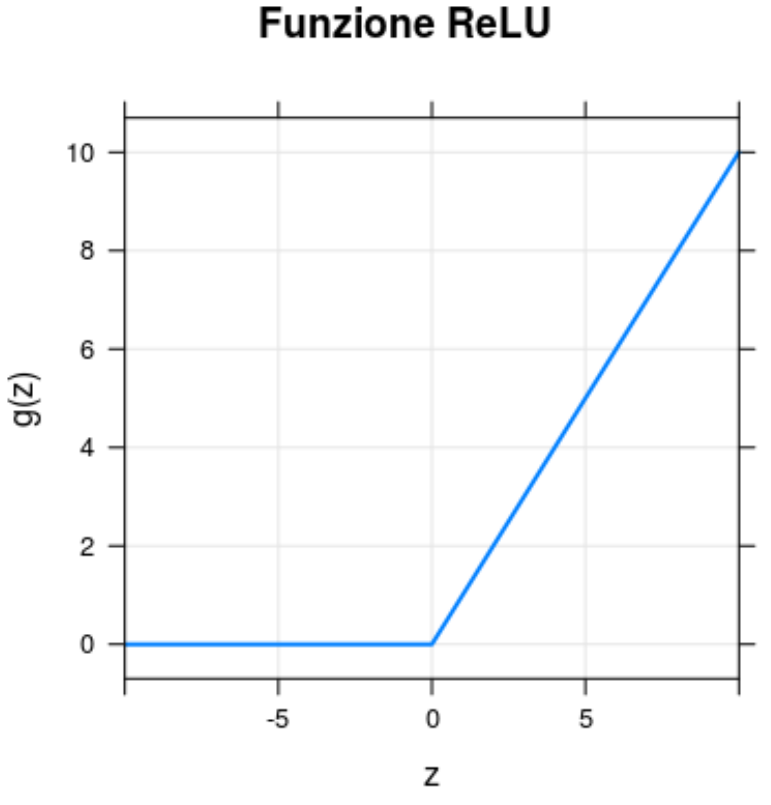
Problema 1

Una funzione di attivazione limitata può ridurre la flessibilità del modello. Cambiamenti anche rilevanti di $\mathbf{z}$, ma lontani dallo 0, corrispondono a variazioni quasi inesistenti della funzione.

Soluzione

Si può utilizzare la funzione di attivazione *rectified linear unit* (ReLU). Quest'ultima è definita come

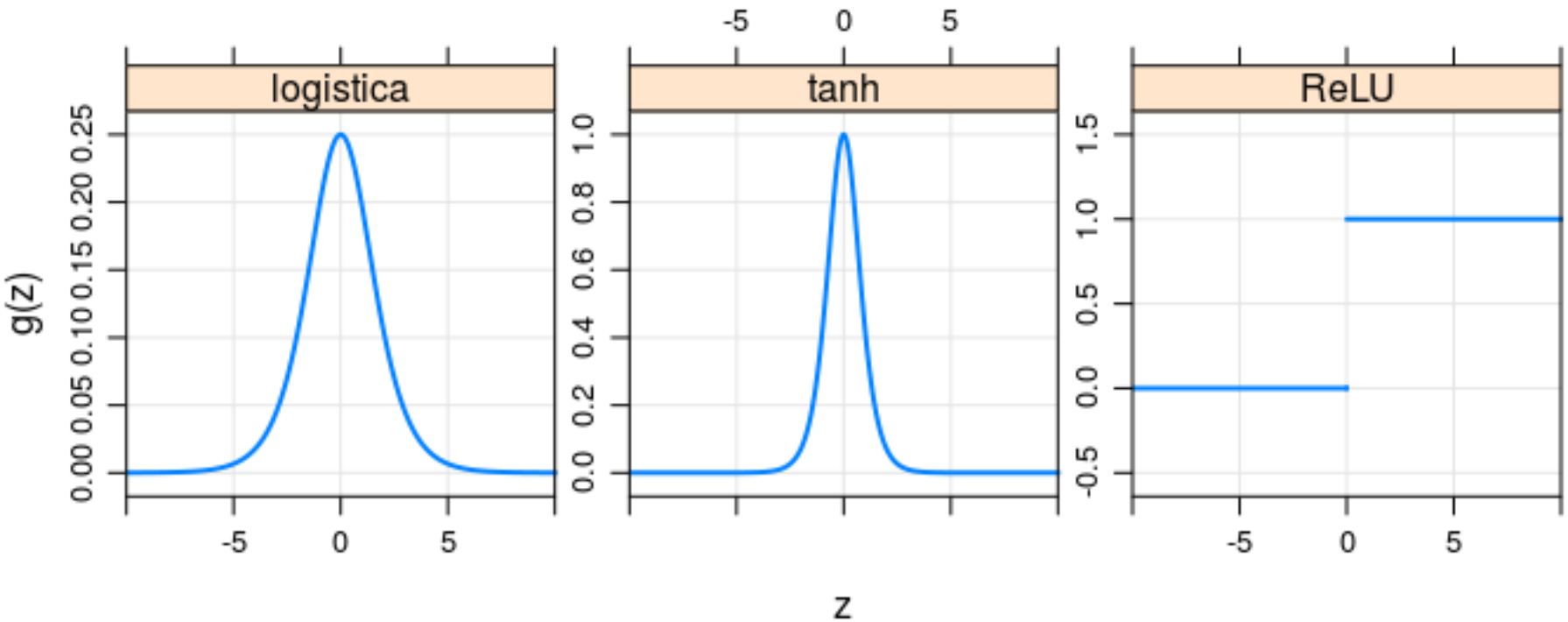
$$g(z) = z_+ = \max(0, z)$$



La funzione di attivazione ReLU

Problema 2: "scomparsa del gradiente"

Nella fase di stima, quando i valori di **z** si avvicinano agli asintoti orizzontali della funzione di attivazione, il gradiente di questa funzione tende a 0.



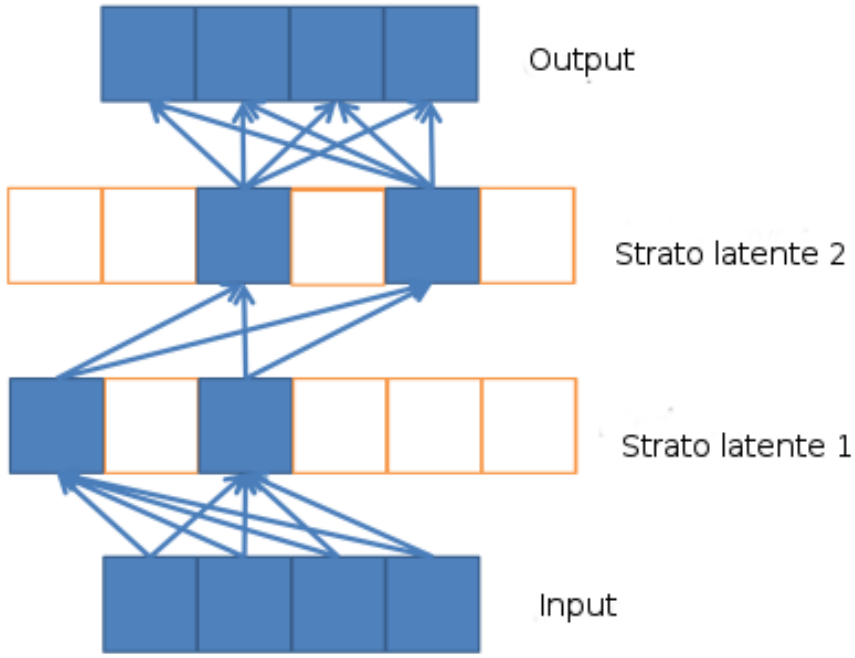
La funzione di attivazione ReLU

Osservazione

La funzione ReLU corrisponde alla funzione "parte positiva", che viene utilizzata nella *spline di regressione* lineare, come **funzione di base**.

Vantaggi della funzione ReLU:

- è lineare e non limitata;
- non soffre della "scomparsa del gradiente";
- costi computazionali minimi;
- viene introdotta sparsità nelle matrici di parametri;
- capacità di adattarsi localmente ai dati.



Teorema di approssimazione universale

Approssimatore universale

Ripley (1996) dimostra che una rete neurale a singolo strato è un approssimatore universale.

Teorema

Ogni funzione continua $f : \mathbb{R}^n \to \mathbb{R}$ *può essere approssimata uniformemente da una rete neurale a singolo strato, con nodi di output lineari e funzioni di attivazione non lineari, monotone crescenti e limitate.*

Delalleau and Bengio (2011) affermano che una rete neurale multi-strato può essere riscritta come una rete neurale a singolo strato con un arbitrario numero di nodi.

Un punto di forza di una rete neurale multi-strato

Perché allora utilizzare una una rete neurale multi-strato?

Molti autori, come ad esempio

- Delalleau and Bengio (2011),
- Eldan and Shamir (2015),
- Telgarsky (2016),
- Liang and Srikant (2016)

dimostrarono che approssimare una funzione, utilizzando una rete neurale multi-strato, a parità di errore di approssimazione (modello con la stessa distorsione), si ha un **guadagno esponenziale** in termini di numero di nodi (e quindi di parametri), rispetto ad una rete neurale a singolo strato.

Compromesso varianza-distorsione

È considerato migliore un modello che, a parità di distorsione, ha il minor numero di parametri, poiché la sua varianza è inferiore.

Compromesso varianza-distorsione

La selezione del modello ottimale, in termini di accuratezza previsiva, deve essere condotta facendo un *compromesso* tra *varianza* e *distorsione*.

Un primo modo per effettuare questo compromesso è quello di regolare la **complessità del modello** scegliendo quello che minimizza l'errore di previsione nell'*insieme di verifica*.
La complessità del modello è stabilita dal numero di parametri, il quale è legato in modo deterministico al numero di **nodi** e di **strati latenti**.

Early stopping

Un secondo modo è bloccare l'algoritmo di *backpropagation* dopo un certo numero di epoche.

Altri metodi di regolarizzazione

Esistono altri metodi per effettuare questo compromesso, come ad esempio i **metodi di penalizzazione** ed il **dropout**.

Metodi di penalizzazione

Applicando il metodo di penalizzazione, la stima dei parametri $\hat{\mathbf{W}}$, diventa quindi

$$\hat{\mathbf{W}} = \arg\min \left\{ \frac{1}{n} \sum_{i=1}^{n} L[y_i, f(x_i; \mathbf{W})] + \lambda J(\mathbf{W}) \right\},$$

dove $J(\mathbf{W})$ è un *termine di regolarizzazione* non negativo, mentre $\lambda \geq 0$ è un *parametro di regolarizzazione*.

Le penalità più utilizzate

- *weight decay*, o penalità L_2;
- penalità L_1, o *lasso*;
- *fused lasso*;
- *group lasso*.

La penalità *weight decay*

La penalità è composta dalla norma quadratica di $\mathbf{W}$, il tensore tridimensionale dei parametri:

$$J(\mathbf{W}) = \frac{1}{2}\|\mathbf{W}\|_2^2 = \frac{1}{2}\sum_{l=1}^{L-1}\sum_{i=1}^{p_l}\sum_{j=1}^{p_{l+1}}\left(w_{ij}^{(l)}\right)^2.$$

Il gradiente della funzione di perdita, rispetto al peso $w_{ij}^{(l)}$, è

$$\frac{1}{n}\sum_{i=1}^{n}\frac{\partial L[y_i, f(x_i;\mathbf{W})]}{\partial w_{ij}^{(l)}} + \lambda w_{ij}^{(l)}.$$

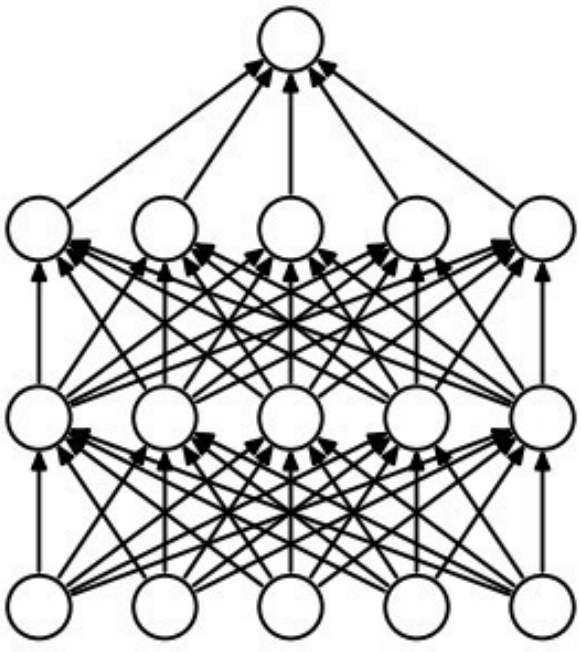
Nella fase di stima, l'aggiornamento dei parametri tramite la discesa del gradiente, non ha costo computazionale aggiuntivo:

$$w_{t+1,ij}^{(l)} = w_{t,ij}^{(l)} - \eta\left(\Delta L(w_{t,ij}^{(l)}) + \lambda w_{t,ij}^{(l)}\right).$$

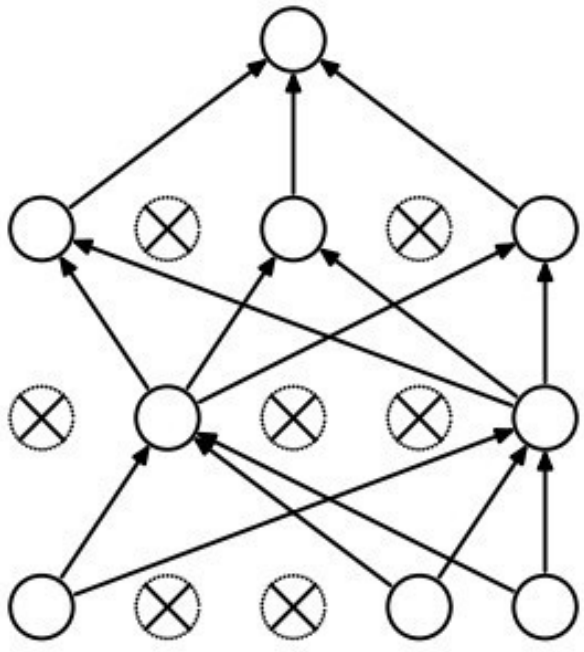
Dropout

Il metodo *dropout*

La tecnica del *dropout* consiste nel porre, ad ogni iterazione della procedura di *backpropagation*, una porzione di nodi pari a zero. Questa porzione viene scelta casualmente.



Rete neurale standard



Dopo l'applicazione del *dropout*

Dropout

Dropout nella stima

Con l'applicazione del *dropout*, il j-esimo nodo dell'$(l+1)$-esimo strato latente è ricavato nel seguente modo:

$$r_i^{(l)} \text{ realizzazione della v.c. } R \sim Bernoulli(p),$$

$$z_j^{(l+1)} = w_{0j}^{(l)} + \sum_{i=1}^{p_l} r_i^{(l)} w_{ij}^{(l)} a_i^{(l)},$$

$$a_j^{(l+1)} = g^{(l+1)}(z_j^{(l+1)}).$$

Dropout nella previsione

Si utilizza la struttura della rete originale con tutti i nodi, in cui i parametri stimati vengono pre-moltiplicati per la probabilità p.

Dropout

Caratteristiche del *dropout*

- il *dropout* è un'approssimazione del risultato che si otterrebbe attraverso la combinazione di classificatori;
- il *dropout* è nato pensando alla logica del campionamento casuale delle variabili della *random forest*;
- viene risolto il problema dell'*overfitting* e si ha anche un netto miglioramento dell'accuratezza previsiva;
- la probabilità p di conservare un nodo nel modello è un parametro di regolarizzazione;
- Il *dropout* esprime tutta la sua efficacia con un elevato numero di osservazioni.

Indice

Convolutional neural network (CNN)

Le *convolutional neural networks* (CNN) sono una classe di reti neurali, che funziona in modo ottimale nella classificazione di immagini.

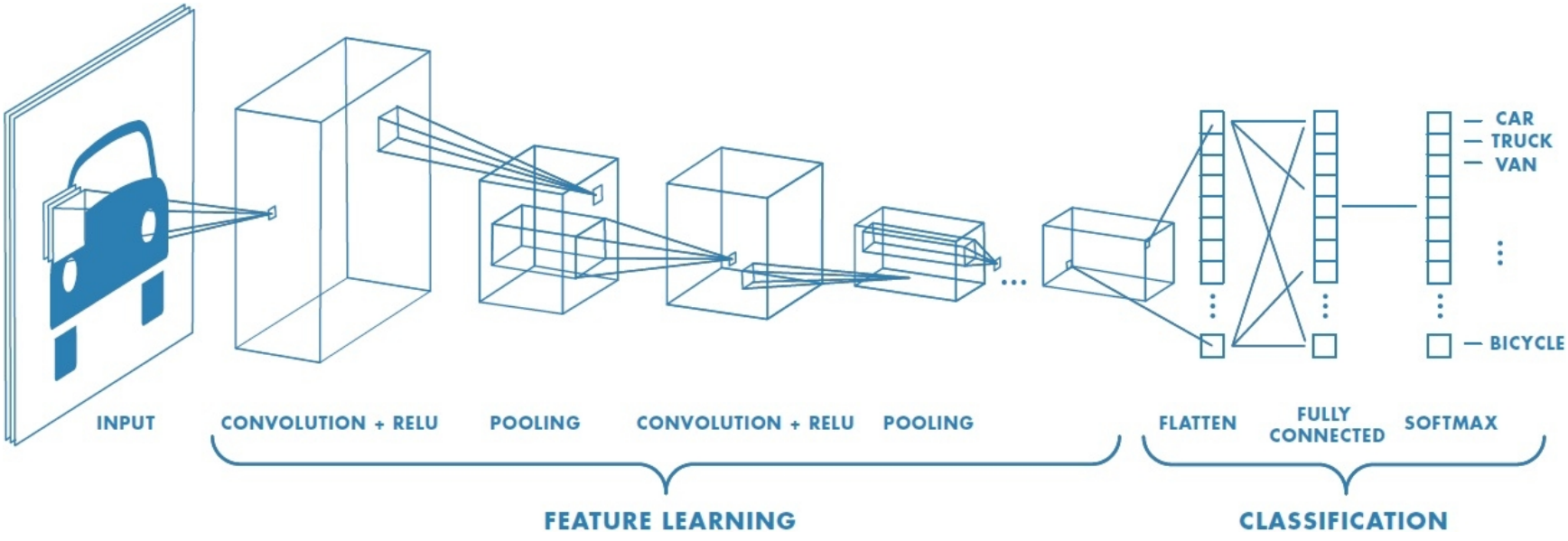
Struttura del dato

Un'immagine possiede la struttura di un tensore a 3 dimensioni: le prime due dimensioni rispecchiano la disposizione dei *pixel*, mentre la terza dimensione è la rappresentazione del colore (RGB).

La struttura della CNN è divisa in due parti:

- la prima serie di strati dopo l'input alterna uno **strato convoluzionale** con uno **strato di pooling**;
- la seconda serie di strati sono **fully-connected**, cioè esattamente come quelli delle *feed-forward neural networks*.

Convolutional neural network (CNN)



Lo strato convoluzionale è costituito da p "versioni" differenti dell'immagine in entrata. Ognuna di queste "versioni" è il risultato dell'applicazione di un **filtro**. Il filtro viene moltiplicato (prodotto-interno) ad ogni sotto-immagine delle stesse dimensioni del filtro. I valori del filtro sono i parametri della rete.

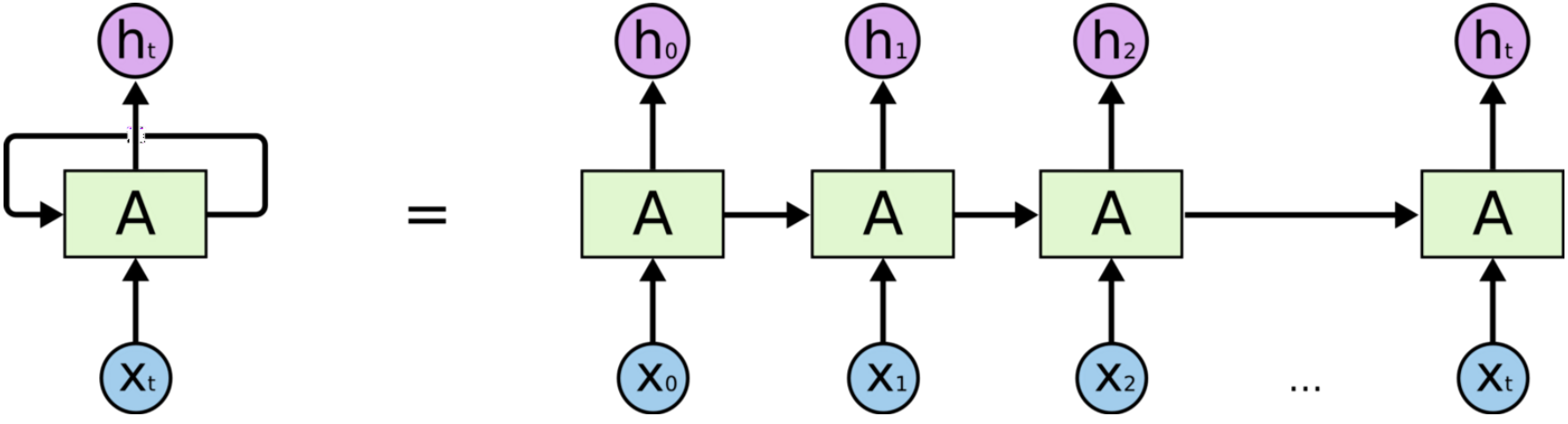
Lo strato di *pooling* suddivide ogni immagine in piccole parti di dimensione $r \times r$, e di ognuna di queste prende il valore **massimo**. Con ciascun valore massimo ricostruisce un'immagine di dimensioni ridotte.

Recurrent neural network (RNN)

Una *recurrent neural network* (RNN) è un tipo di rete neurale, che funziona in modo ottimale nell'analisi di dati sequenziali, come serie storiche o dati testuali.

Una RNN possiede uno o più **strati ricorrenti**

Uno strato ricorrente è uno strato latente che, non solo connette lo strato di input con lo strato latente successivo, ma connette anche lo strato latente al tempo precedente con lo strato latente al tempo successivo.



Recurrent neural network (RNN)

Problema

- L'utilizzo di strati ricorrenti con una finestra temporale (insieme di strati latenti passati connessi con lo strato latente corrente) ampia fa esplodere il numero di parametri da stimare;
- le informazioni importanti faticano a propagarsi a lungo nel tempo.

Soluzione

- *Long-Short Term Memory* (LSTM)
- *Gated Recurrent Unit* (GRU)

Indice

La libreria Keras di R

Le caratteristiche principali della libreria Keras sono:

- permette la stima di tutte le classi di modelli per il *deep learning* supervisionato e non supervisionato;
- si serve del calcolo parallelo;
- può utilizzare l'unità di elaborazione grafica (GPU) per la stima del modello;
- si interfaccia alla libreria `TensorFlow` di `python`.

Consiglio!

Le librerie implementate in altri linguaggi di programmazione, come ad esempio `python`, hanno una miglior gestione della memoria ed una maggior velocità computazionale.

Analisi di *big data*

Verrà stimata una rete neurale multi-strato (FFNN) utilizzando il *dataset* *Reuters Corpus Volume* I (RCV1), una raccolta di articoli.

- Lo scopo è classificare l'appartenenza alla categoria aziendale/industriale (variabile risposta binomiale).
- L'insieme di stima è composto da 200 000 osservazioni (articoli).
- Ci sono 2 000 variabili esplicative e rappresentano le parole presenti nell'articolo (ogni variabile indica la presenza o assenza di una determinata parola).
- verrà calcolato il tasso di errata classificazione nell'insieme di verifica, composto da 80 000 osservazioni.

Preparazione dei dati

```
library(keras)

# Caricamento dei dati
X.train <- read.csv("X_train_RCV1.csv", head = F)
y.train <- read.csv("y_train_RCV1.csv", head = F)
X.test <- read.csv("X_test_RCV1.csv", head = F)
y.test <- read.csv("y_test_RCV1.csv", head = F)

X.train <- as.matrix(X.train)
y.train <- array(as.integer(y.train[, 1]))
y.train <- to_categorical(y.train, 2)
X.test <- as.matrix(X.test)
y.test <- array(as.integer(y.test[, 1]))
y.test <- to_categorical(y.test, 2)
```

La costruzione della rete neurale

```
# Architettura della rete
model <- keras_model_sequential()
model %>%
  layer_dense(units = 1000, activation = "relu", input_
      shape = c(nrow(X.train))) %>%
  layer_dropout(rate = 0.5) %>%
  layer_dense(units = 500, activation = "relu") %>%
  layer_dropout(rate = 0.5) %>%
  layer_dense(units = 2, activation = "softmax")

# Parametri di ottimizzazione
model %>% compile(
  loss = "categorical_crossentropy",
  optimizer = "adam",
  metrics = c("accuracy")
)
```

Stima del modello e previsione

```
# Stima del modello
model %>% fit(
  X.train,
  y.train,
  epochs = 100,
  batch_size = 128,
  validation_data = list(X.test, y.test)
)

# Previsione
model %>% evaluate(X.test, y.test)
model %>% predict_classes(X.test)
model %>% predict_proba(X.test)
```

Riferimenti utili