

IN550 Machine Learning

Reti neurali

Vincenzo Bonifaci

Una nuova occhiata alla regressione logistica

Nella regressione logistica, abbiamo calcolato la probabilità che l'etichetta di x fosse $y = 1$ come

$$\sigma(w^\top x) = \sigma(w_0 + w_1 x_1 + \dots + w_d x_d)$$

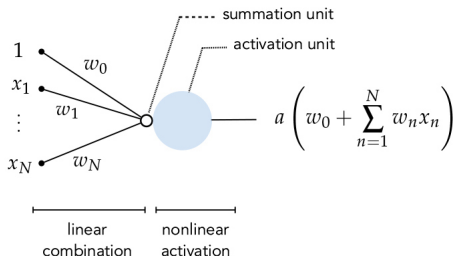
dove σ è la funzione **sigmoide logistica**

Neurone artificiale

Possiamo generalizzare questa operazione con un'unità *neurone artificiale*, che sulla base di stimoli (x_0, x_1, \dots, x_d) produce un valore di uscita

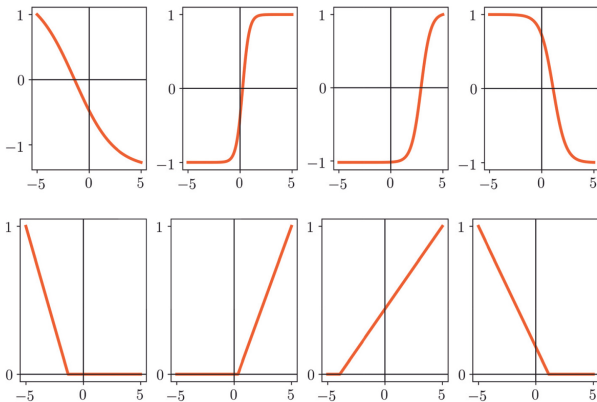
$$f = a \left(w^\top x \right) = a \left(w_0 + w_1 x_1 + \dots + w_d x_d \right)$$

dove $a : \mathbb{R} \rightarrow \mathbb{R}$ è un'opportuna *funzione di attivazione* (nonlineare) e il vettore $w \in \mathbb{R}^{d+1}$ regola la forza delle *connessioni* dagli stimoli al neurone



Altre funzioni di attivazione

$$a(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$



$$a(z) = \max(0, z)$$

Esempio: emulazione di funzioni logiche

Sia $a(z) = \max(0, z)$ (ReLU) e consideriamo due input $x_1, x_2 \in \{0, 1\}$
(True: 1, False: 0)

Allora il neurone

$$z = x_1 + x_2 - 1 \quad a(z) = \max(0, z)$$

equivale alla funzione $\text{AND}(x_1, x_2)$

mentre il neurone

$$z = 1 - x_1 \quad a(z) = \max(0, z)$$

equivale alla funzione $\text{NOT}(x_1)$

Domanda 1. Una unità ReLU può emulare la funzione $\text{OR}(x_1, x_2)$?

Domanda 2. Se codifichiamo True con $+1$ e False con -1 , un'unità con $a(z) = \text{sgn}(z)$ può emulare le funzioni AND, OR e NOT?

Possiamo combinare U_1 unità della forma

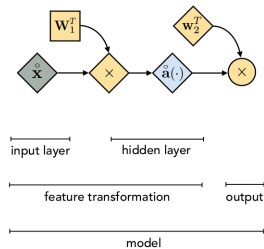
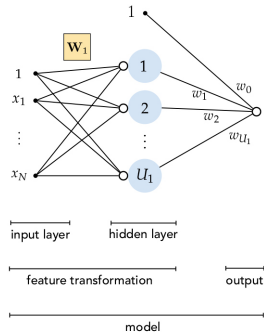
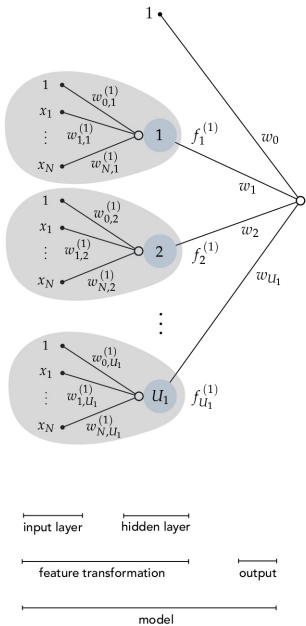
$$f_j^{[1]}(x) = a\left(w_j^{[1]\top} x\right) = a(z_j^{[1]})$$

per ottenere un output complessivo

$$w_0^{[2]} + w_1^{[2]} f_1^{[1]}(x) + \dots + w_{U_1}^{[2]} f_{U_1}^{[1]}(x) = W^{[2]} f^{[1]}$$

Tali U_1 neuroni formano uno *strato nascosto* (hidden layer):

- non sono direttamente connessi uno all'altro
- lo strato è **nascosto** nel senso che il valore corretto che gli $f_j^{[1]}(x)$ devono assumere per un dato esempio (x, y) non è noto (a differenza di quanto avviene per gli ingressi e l'uscita della rete)



Esempio: 3 ingressi, $U_1 = 2$, $U_2 = 1$

Chiamiamo lo strato di ingresso lo strato **zero**:

$$x_0 = 1, \quad x_1 = f_1^{[0]}, \quad x_2 = f_2^{[0]}, \quad x_3 = f_3^{[0]}$$

Per le unità dello strato 1 (strato nascosto) abbiamo

$$z_1^{[1]} = W_1^{[1]} x \quad f_1^{[1]} = a(z_1^{[1]})$$

$$z_2^{[1]} = W_2^{[1]} x \quad f_2^{[1]} = a(z_2^{[1]})$$

dove $W^{[1]}$ è una matrice $2 \times (3 + 1)$ di parametri

Lo strato di uscita consiste di un solo neurone:

$$z_1^{[2]} = W_1^{[2]} f^{[1]} \quad f_1^{[2]} = a(z_1^{[2]})$$

dove $W^{[2]}$ è una matrice $1 \times (2 + 1)$ di parametri e $f^{[1]}$ è il vettore calcolato dallo strato 1 (incluso il valore costante $f_0^{[1]} = 1$)

Nota. Il libro di testo scrive le matrici W in forma trasposta (ad es. $W^{[1]}$ sarebbe $(3 + 1) \times 2$).

In forma **vettorizzata** possiamo scrivere:

$$f^{[0]} = (x_0, x_1, x_2, x_3)^\top$$

$$z^{[1]} = W^{[1]} f^{[0]} \quad f^{[1]} = a(z^{[1]})$$

$$z^{[2]} = W^{[2]} f^{[1]} \quad f^{[2]} = a(z^{[2]})$$

dove

$$a(z) = (a(z_1), a(z_2), \dots)$$

La forma vettorizzata è cruciale per sfruttare appieno le risorse di calcolo disponibili: permette di sfruttare il parallelismo hardware

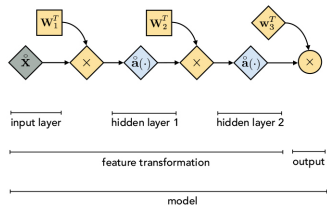
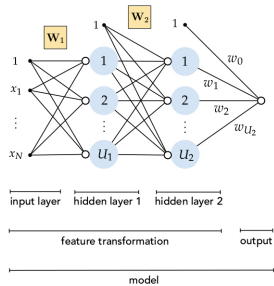
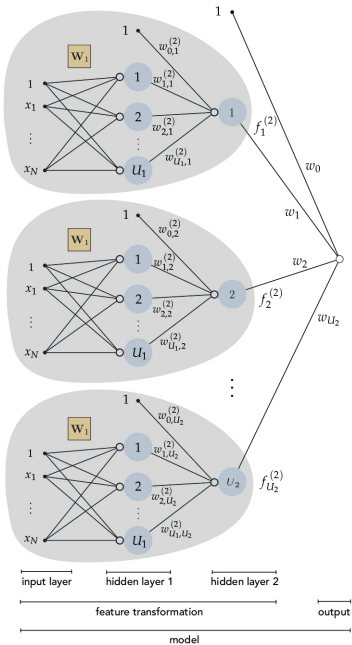
L'uscita di uno strato può fare da input per un secondo strato e così via:

$$h(x) = a \left(W^{[M]} a \left(W^{[M-1]} a \left(\dots a \left(W^{[1]} x \right) \right) \right) \right)$$

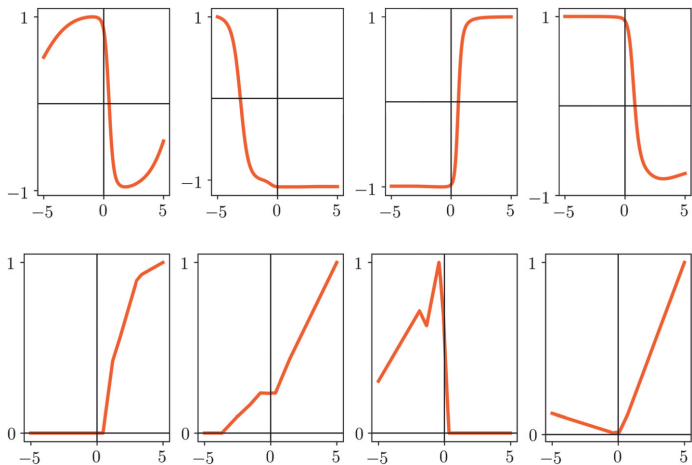
Aggiungendo strati:

- La classe delle funzioni ipotesi rappresentabili (\mathcal{H}) si espande
- Aumentano i parametri da stimare e quindi la varianza

Il termine **deep** in *deep learning* si riferisce alla **profondità** delle reti neurali costruite e quindi alla potenziale complessità delle ipotesi apprese



Esempi di funzioni rappresentabili con reti a due strati



Una rete neurale è descritta da un'architettura e dei parametri:

- *Architettura*: numero di strati, numero di neuroni in ogni strato, tipo di funzioni di attivazione in ogni strato
- *Parametri*: una matrice $W^{[k]}$ per ogni strato k

Il tipo di rete qui discussa è detta *fully-connected* in quanto il neurone di ogni strato riceve un segnale da **tutti** i neuroni dello strato precedente

Una rete neurale deve classificare immagini RGB di dimensione 64×64
($d = 64 \times 64 \times 3$)

La rete ha la seguente architettura:

- 1 strato di input, 2 strati nascosti, 1 strato di output
- Rispettivamente $(d, 3, 2, 1)$ unità in ciascuno strato
- Le funzioni di attivazione sono ReLU negli strati nascosti e sigmoide nello strato di output

Quanti parametri ha la rete in tutto?

Esempio (segue)

- Il primo strato nascosto ha $3(d + 1)$ parametri (matrice $W^{[1]}$)
- Il secondo strato nascosto ha $2(3 + 1)$ parametri (matrice $W^{[2]}$)
- Lo strato di output ha $1(2 + 1)$ parametri (matrice $W^{[3]}$)
- In totale $3d + 14 = 36878$ parametri

Propagazione in avanti (*Forward propagation*)

L'uscita di ogni strato si ottiene *propagando in avanti* l'uscita dello strato precedente:

$$z^{[k]} = W^{[k]} f^{[k-1]} \quad f^{[k]} = a(z^{[k]})$$

fino ad ottenere l'uscita dell'ultimo strato

Minimizzazione del rischio empirico nelle reti neurali

Le ipotesi delle reti neurali hanno la forma

$$h(x) = a \left(W^{[M]} a \left(W^{[M-1]} a \left(\dots a \left(W^{[1]} x \right) \right) \right) \right)$$

dove M è l'indice dello strato di uscita della rete

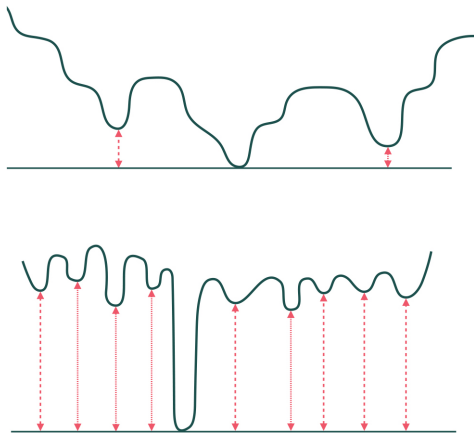
Scegliendo una funzione di costo ℓ arriviamo all'usuale rischio empirico

$$\text{RE}_S(W) = \frac{1}{m} \sum_{i=1}^m \ell(h(x^{(i)}), y^{(i)})$$

che cerchiamo di minimizzare scegliendo h (cioé le matrici W)

- L'architettura della rete è fissata a priori
- Le matrici $W^{[1]}, \dots, W^{[M]}$ sono oggetto dell'ottimizzazione
- Il problema di ottimizzazione risultante è **non convesso** in W

Non convessità della funzione rischio empirico



In pratica, metodi del primo ordine avanzati (quali Gradiente normalizzato, RMSprop, Adam) sembrano fornire ottimi risultati

Per implementare i metodi del primo ordine è sufficiente saper calcolare la derivata della funzione costo rispetto ad ogni matrice dei parametri:

$$\frac{\partial \ell}{\partial W^{[k]}}$$

Idea:

- Esprimere la definizione dell'uscita ($f^{[M]}$) in termini dell'uscita dello strato precedente ($f^{[M-1]}$) e applicare la regola della catena per le derivate

Esempio

Supponiamo l'ultimo strato abbia 1 solo neurone e

$$\ell(f^{[M]}, y) = (f^{[M]} - y)^2, \quad a(z) = \sigma(z), \quad z^{[M]} = W^{[M]}f^{[M-1]}$$

Dipendenza del costo ℓ dai pesi $W^{[M]}$:

$$\begin{aligned} \frac{\partial \ell}{\partial W^{[M]}} &= \frac{\partial \ell}{\partial f^{[M]}} \cdot \frac{\partial f^{[M]}}{\partial z^{[M]}} \cdot \frac{\partial z^{[M]}}{\partial W^{[M]}} \\ &= 2(f^{[M]} - y) \cdot \sigma'(z^{[M]}) \cdot f^{[M-1]} \end{aligned}$$

Dipendenza del costo ℓ dall'uscita dello strato precedente:

$$\begin{aligned} \frac{\partial \ell}{\partial f^{[M-1]}} &= \frac{\partial \ell}{\partial f^{[M]}} \cdot \frac{\partial f^{[M]}}{\partial z^{[M]}} \cdot \frac{\partial z^{[M]}}{\partial f^{[M-1]}} \\ &= 2(f^{[M]} - y) \cdot \sigma'(z^{[M]}) \cdot W^{[M]} \end{aligned}$$

Calcolo del gradiente: la *backpropagation*

In generale, questo ci permette di calcolare

$$\frac{\partial \ell}{\partial W^{[k]}} \text{ e } \frac{\partial \ell}{\partial f^{[k-1]}}$$

se abbiamo già calcolato

$$\frac{\partial \ell}{\partial f^{[k]}}$$

Il gradiente viene così calcolato **a ritroso** (retropropagato) dall'ultimo strato fino allo strato di input (*backpropagation*)

Le librerie software automatizzano il calcolo (*differenziazione automatica*)

Se la forward propagation richiede T operazioni aritmetiche, la backward propagation richiede $O(T)$ operazioni aritmetiche (Baur-Strassen 1983)

Altri aspetti delle reti neurali

- Strato di uscita
- Inizializzazione dei pesi
- Regolarizzazione

Va scelto in funzione del tipo di output, ad esempio:

- Regressione: 1 neurone, ReLU/identità
- Classificazione binaria: 1 neurone, sigmoide
- Classificazione multiclasse: K neuroni, esponenziale normalizzato (softmax)

Inizializzazione dei pesi

Difficoltà: gradienti che scompaiono/esplodono

Euristiche che riducono il problema in pratica:

- Pesi $W^{[k]}$ casuali, gaussiani con varianza $1/N^{[k-1]}$ dove $N^{[k-1]}$ è il numero di neuroni nello strato $k - 1$
- Pesi $W^{[k]}$ casuali, gaussiani con varianza $2/N^{[k-1]}$ (più efficace per unità ReLU)
- Pesi $W^{[k]}$ casuali, gaussiani con varianza

$$\frac{2}{N^{[k-1]} + N^{[k]}}$$

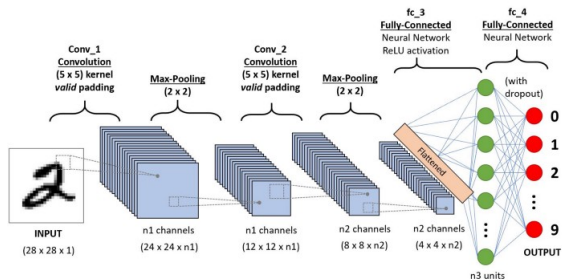
(*inizializzazione di Xavier/He*)

Partire da pesi identici non è una buona idea (troppo simmetrici)

- Regolarizzazione ℓ_2
 - La funzione obiettivo diventa $RE_S(W) + \lambda \|W\|_2^2$
 - $\lambda \geq 0$ iperparametro di regolarizzazione
- Regolarizzazione *dropout*, con fattore $\alpha \in (0, 1]$:
 - Ad ogni aggiornamento dei pesi, ogni neurone è attivo solo con probabilità α
 - I neuroni “spenti” non propagano verso lo strato successivo
 - $\alpha \in (0, 1]$ iperparametro di regolarizzazione

Architetture convolutive [Convolutional Neural Networks]

Per la **classificazione di immagini** architetture cosiddette *convolutive* sono spesso preferite all'architettura **fully connected**



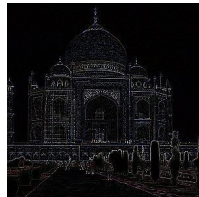
- Permettono di ridurre il numero di parametri (e quindi ridurre la varianza) rispetto all'architettura **fully connected**
- Sono basate su tecniche di elaborazione immagini: *filtri convolutivi* e *sottocampionamento*

Filtro convolutivo



0	1	0	
1	-4	1	
0	1	0	

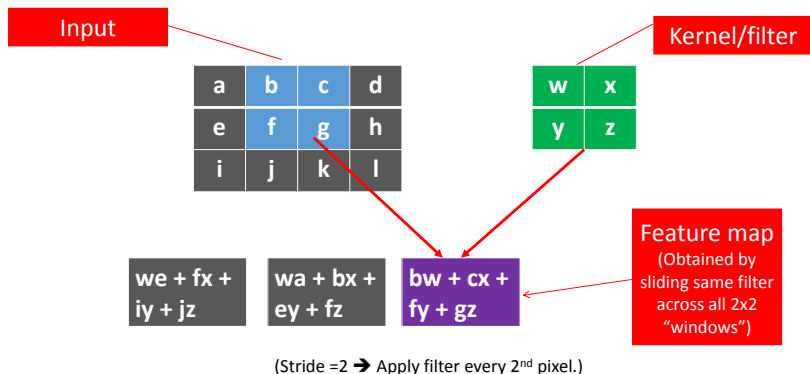
“Multiply **each** pixel value by -4 and add to it values of neighboring pixels.”



Nonzeroes wherever neighboring pixels have v. different values (“Edge Detector”)!

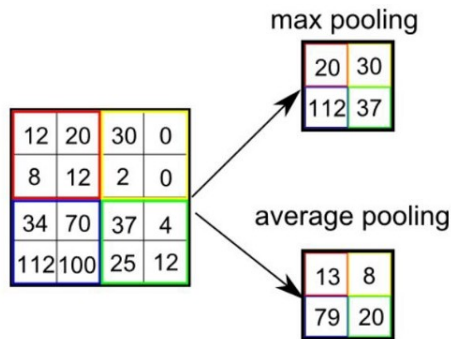
Le reti convolutive cercano di apprendere i filtri **direttamente dai dati** (il filtro in figura per esempio è 3×3 : ha solo **9 parametri**)

Strato convolutivo [convolutional layer]



L'array di output è ottenuto facendo scorrere il filtro su tutte le possibili "finestre" (in questo caso, tutti i sottoarray contigui 2×2)

Strato di sottocampionamento [pooling layer]



Riduce la quantità di dati

Spesso preferito il *max pooling* in quanto può “evidenziare” valori notevoli

Architettura complessiva

