Lezione10

Table of contents

Single-Layer Perceptron e Approssimazione di Funzioni

Considerando un *Single-Layer Perceptron* (SLP) con n ingressi reali e una funzione $g : \mathbb{R}^n \to \mathbb{R}$, che non andremo a dettagliare, vogliamo approssimare una funzione $f : \mathbb{R}^n \to \mathbb{R}$. Per approssimarla al meglio, disporremo un insieme di vettori in cui f è già nota, chiamato *trining set*, e un vettore contenente i rimanenti, detto *test set*.

Questi vettori avranno la forma seguente

$$\mathbf{x} = (x_1, x_2, \dots, x_{n-1}, x_n = -1)$$

con -1 finale, il *peso di bias* che sarà sempre presente.

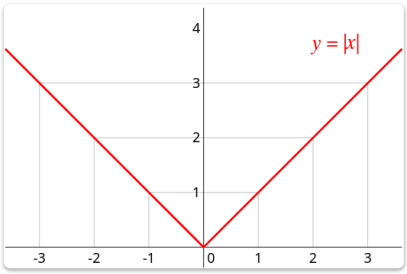
Il calcolo dell'uscita, fissato un vettore di pesi $\mathbf{w}$, sarebbe dato dalla funzione di attivazione applicata alla somma pesata dei valori del vettore d'ingresso per i suoi pesi. Possiamo anche intenderlo come il prodotto scalare euclideo.

$$o(\mathbf{w}, \mathbf{x}) = g(\mathbf{w} \cdot \mathbf{x}) = g\left(\sum_{i=1}^{n-1} w_i x_i - w_n\right)$$

Siccome abbiamo a disposizione un training set, data la natura dell'*addestramento supervisionato*: consideriamo un $\mathbf{x}$ vettore del training set; conosciuto lui, conosciamo anche $f(\mathbf{x})$. Stiamo dicendo che siamo in grado di calcolare l'*errore* (con segno) compiuto dalla rete nell'approssimazione della funzione f nel punto x_i.

$$\epsilon(\mathbf{w}, \mathbf{x}) = f(\mathbf{x}) - o(\mathbf{w}, \mathbf{x}) = f(\mathbf{x}) - g(\mathbf{w} \cdot \mathbf{x})$$

Quello che ci interessa fare è di trovare un insieme di pesi tali per cui il valore ϵ è abbastanza piccolo, inteso in valore assoluto. Diciamo così perché non ci interessa il fatto che questo tenda a $-\infty$, ma piuttosto che si avvicini il più possibile allo 0.



Siccome il valore assoluto $y = |x|$ non è derivabile in zero (2 derivate diverse), dobbiamo cercare una via d'uscita: lo dobbiamo fare perché altrimenti i calcoli del gradiente non ci saranno possibili. Invece che prendere ϵ in valore assoluto, andiamo a considerare la funzione e **errore quadratico** calcolata come

$$e(\mathbf{w}, \mathbf{x}) = \frac{1}{2}\epsilon^2(\mathbf{w}, \mathbf{x})$$

con $1/2$ la frazione che ci servirà più avanti per calcolare le derivate, permettendoci di eliminare la costante.

Essendo una funzione esponenziale, è monotona crescente e quindi possiamo cercare quei valori che la minimizzino, ovvero i pesi del vettore. Tutto questo ragionamento lo faremo considerando $g \in C^{\infty}(\mathbb{R})$, funzione di attivazione derivabile infinite volte su tutto lo spazio (la funzione logistica).

> ✎ **L'algoritmo di apprendimento che vedremo, è pensato esclusivamente per funzioni derivabili un numero arbitrario di volte (o almeno una volta $g = C^1(\mathbb{R})$). Questo perché la cancellazione nel calcolo delle derivate, risulta molto conveniente: sfrutteremo questo vantaggio.**

La nostra funzione derivabile di errore quadratico, può essere minimizzata utilizzando l'*algoritmo di discesa del gradiente*: partendo da un valore iniziale di pesi $\mathbf{w}$, cerchiamo quei valori che minimizzino l'errore usando l'equazione

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \nabla e(\mathbf{w})$$

- $\mathbf{w}$ iniziale possiamo generarlo con valori casuali;
- α è il coefficiente di apprendimento;
- $\nabla e(\mathbf{w})$ il gradiente, che nell'algoritmo viene calcolato derivata per volta.

Fissato un i-esimo peso, la derivata composta dell'errore quadratico si calcola come

$$\frac{\partial o}{\partial w_i}(\mathbf{w}) = g'(\mathbf{w} \cdot \mathbf{x}) \cdot x_i$$

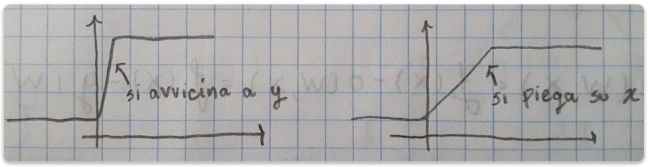
L'aggiornamento dell'i-esimo peso, quindi la **minimizzazione dell'errore per un SLP**

$$w_i \leftarrow w_i + \alpha \epsilon(\mathbf{w}) g'(\mathbf{w} \cdot \mathbf{x}) x_i = w_i + \alpha (f(x) - g(\mathbf{w} \cdot \mathbf{x})) g'(\mathbf{w} \cdot \mathbf{x}) x_i$$

- g è da intendersi strettamente funzione logistica $g(x) = \frac{1}{1+e^{-x}}$;
- la sommatoria può essere riscritta come sottrazione, sostituendo $f(x)$ con $g(\mathbf{w} \cdot \mathbf{x})$.

> ✎ **Nei libri di testo, la derivata di $g(x)$ può risultare essere riscritta come $g'(x) = g(x) \cdot g(1 - g(x))$,che è equivalente.**

Usare esclusivamente la funzione logistica non è tanto un limite siccome questa può essere utilizzata per approssimare altre funzioni. Se ci pensiamo, modificare le costanti della funzione, porterà a una modifica della flessione sull'asse delle ascisse e delle ordinate: possiamo in questo modo, approssimare la funzione di Heaviside.



Multi-Layer Perceptron e Approssimazione di Funzioni

Considerando un *Multi-Layer Perceptron* (MLP) con n ingressi ed m neuroni nascosti, utilizzando la convenzione usata per il SLP, denotiamo il vettore in ingresso iniziale

$$\mathbf{x} = (x_1, x_2, \dots, x_{n-1}, x_n = -1)$$

e il vettore in ingresso per ciascun *layer nascosto*, della stessa tipologia

$$\mathbf{y} = (y_1, y_2, \dots, y_{n-1}, y_n = -1)$$

ipotizzando di avere un'unica funzione di attivazione $g : \mathbb{R}^n \to \mathbb{R}$ per tutti quanti gli strati nascosti e neuroni. Partendo dall'input, la prima cosa che faremo sarà applicare il vettore dei pesi $\mathbf{w}_j \in \mathbb{R}^n$ che li collega verso i neuroni nascosti j-esimi, applicando il prodotto scalare e la funzione g (nuovo input del neurone dello strato nascosto)

$$y_j = g(\mathbf{w}_j \cdot \mathbf{x})$$

L'ouput vettore di pesi $\mathbf{v} \in \mathbb{R}^m$ che collega i neuroni ouput, dipendente implicitamente per il *vettore di pesi del MLP* $\mathbf{w} = (\mathbf{v}, \mathbf{w}_1, \mathbf{w}_2, \dots, \mathbf{w}_{m-1})$ viene calcolato come

$$o(\mathbf{w}, \mathbf{x}) = g(\mathbf{v} \cdot \mathbf{y})$$

Da notare come stiamo suggerendo che i calcoli siano di tipo matriciale, con $\mathbf{w}$ profondità n e $\mathbf{v}$ profondità m. In verità, alcuni pesi andranno probabilmente a essere nulli: memorizzare come matrice non è necessario, ma ha comunque sotto certe condizioni, motivo di essere. Questa matrice viene detta **tensore**.

Per calcolare l'output usiamo la formula con prodotti scalari

$$o(\mathbf{w}, \mathbf{x}) = g\left(\sum_{j=1}^{m} v_j y_j\right) = g\left(\sum_{j=1}^{m-1} v_j g\left(\sum_{i=1}^{n-1} w_{j,i} x_i - w_{j,n}\right) - v_m\right)$$

- $w_{j,i}$ la i-esima componente del vettore $\mathbf{w}_j$, vettore collegante lo strato d'input con il j-esimo neurone dello strato nascosto;
- v_j l'ouput del neurone nascosto.

Seguendo la stessa proposizione fatta per i SLP, usando $g \in C^\infty(\mathbb{R})$, introduciamo l'errore quadratico come

$$e(\mathbf{w}, \mathbf{x}) = \frac{1}{2}\epsilon^2(\mathbf{w}, \mathbf{x})$$

e calcoliamo il gradiente

$$\frac{\partial o}{\partial v_j}(\mathbf{w}) = g'(\mathbf{v} \cdot \mathbf{y}) y_j$$

L'aggiornamento dei pesi viene fatto componente per componente; la regola di aggiornamento dei pesi che collegano lo strato nascosto con l'uscita

$$v_j \leftarrow v_j + \alpha(f(\mathbf{x}) - g(\mathbf{v} \cdot \mathbf{y}))g'(\mathbf{v} \cdot \mathbf{y})y_j$$

- j indice dello strato nascosto;
- i indice del neurone.

$$\frac{\partial(\mathbf{v}\cdot\mathbf{y})}{\partial w_{j,i}} = v_j g'(\mathbf{w}_j\cdot\mathbf{x})x_i$$

La regola di aggiornamento dei pesi $w_{j,i}$ che collegano i neuroni di input con il j-esimo neurone dello strato nascosto, **minimizzazione dell'errore per un MLP** è

$$w_{j,i} \leftarrow w_{j,i} + \alpha(f(\mathbf{x}) - g(\mathbf{v}\cdot\mathbf{y}))g'(\mathbf{v}\cdot\mathbf{y})v_j g'(\mathbf{w}_j\cdot\mathbf{x})x_i$$

con

$$\mathbf{v} \leftarrow \mathbf{v} + \alpha\epsilon(\mathbf{w})g'(\mathbf{v}\cdot\mathbf{y})\mathbf{y} \qquad \mathbf{w}_j \leftarrow \mathbf{w}_j + \alpha\epsilon(\mathbf{w})g'(\mathbf{v}\cdot\mathbf{y})v_j g'(\mathbf{w}_j\cdot\mathbf{x})\mathbf{x}$$

Per ogni campione del training set, calcoleremo qualche volta la discesa del gradiente con criterio di stop arbitrario e calcolabile a piacere. La **matrice di confusione** sarà la matrice che descrive il comportamento della nostra rete, dicendoci quali sono i positivi e i falsi positivi.

Teorema di Approssimazione Universale

Ora che abbiamo visto come viene approssimata una funzione, ci serve sapere quale sia il criterio calcolabile che dica il momento in cui fermare la computazione.

Un MLP in parentela stretta a quello che abbiamo visto (a strato nascosto singolo), è capace di approssimare una qualsiasi funzione in $\mathbb{R}^n$, con arbitraria precisione.
Vediamo come questo MLP, sia lo stesso oggetto di cui abbiamo discusso fino a ora.

Fissiamo $n \in \mathbb{N}_+$ come il *numero di variabili* di qui vogliamo ottenere un'approssimazione, un compatto (insieme chiuso e limitato) $D \subset \mathbb{R}^n$ su cui vogliamo eseguire l'approssimazione di una funzione continua $f : S \to \mathbb{R}$ che magari non sfori di troppo il compatto, espandibile a un aperto $S \subseteq \mathbb{R}^n$ con $D \subset S$.
Scegliamo una funzione di attivazione continua, monotona crescente e limitata $g : \mathbb{R} \to \mathbb{R}$, ovvero le funzioni logistiche e Heaviside.

Fissato un $\epsilon \in \mathbb{R}_+$ per tutto f (approssimazione globale), il teorema garantisce l'esistenza di:

- un numero $m \in \mathbb{N}_+$ di neuroni dello strato intermedio;
- un insieme di pesi v_j e valore di bias θ_j, applicabili allo strato intermedio;
- un vettore di pesi $w_{j,i} \in \mathbb{R}$ per ogni nodo dello strato intermedio (senza bias abbinato)

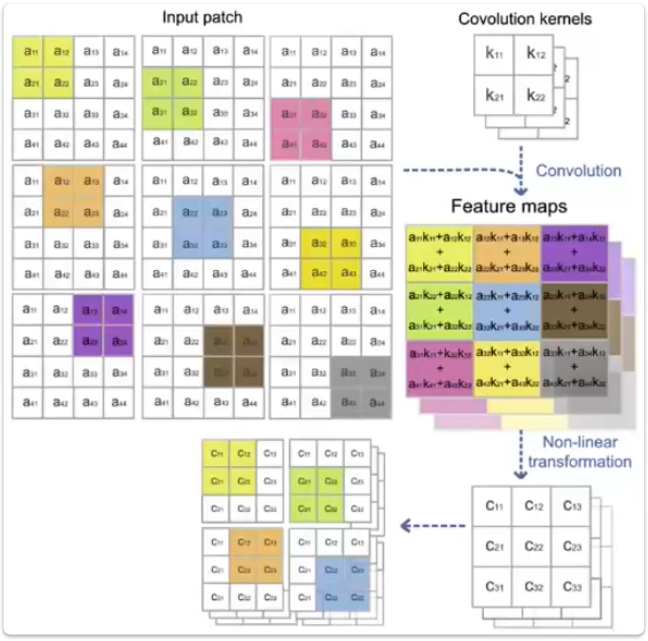
tali per cui la funzione h applicando la MLP utilizzante v_j sullo strato d'uscita e $w_{j,i}$ sullo strato d'ingresso (MLP con strato nascosto singolo), disti dalla funzione che vogliamo approssimare al più di un valore più piccolo di ϵ scelto a piacere.

$$h(\mathbf{x}) = \sum_{j=1}^{m} v_j g\left(\sum_{i=1}^{n} w_{j,i}x_i + \theta_j\right) \qquad \max_{\mathbf{x}\in D} |f(\mathbf{x}) - h(\mathbf{x})| < \epsilon$$

La matematica di quello che abbiamo scritto ci sta dicendo che è possibile costruire un MLP per approssimare una funzione tanto bene quanto vogliamo. Nel senso informatico non siamo molto felici siccome non ci viene dato nessun modo per trovare le variabili sopra citate: dobbiamo inventarci un modo per farlo.

Purtroppo non lo possiamo sapere subito quanti nodi mettere nello strato intermedio, ci sono sì delle formule che generalizzano un poco il numero da prendere ma nella pratica non è mai così. Possiamo però andare per tentativi, partendo da un m piccolo, salendo mano a mano che ϵ diminuisce.

Delle volte, la rete profonda viene utilizzata per approssimare una funzione in base a caratteristiche sue locali, che possono essere presenti in varie parti del vettore d'input. Quando parliamo di questo, parliamo di **reti convoluzionali** (o convolutive), dove un *kernel di convoluzione* viene utilizzato, per denotare soltanto le caratteristiche particolari del set.


Input patch
Covolution kernels
Convolution
Feature maps
Non-linear transformation

28/03/2023